Operating Systems

Lecture Notes

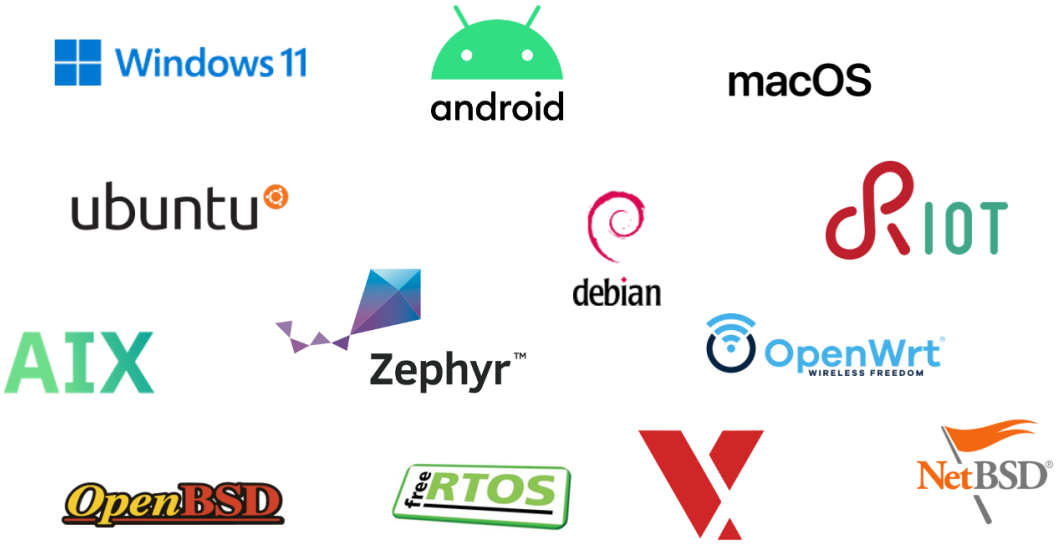
Jürgen Schönwälder

February 19, 2024

Abstract

This memo provides annotated slides for the Computer Science module "Operating Systems" offered at the Constructor University (formerly Jacobs University Bremen). The topics covered are processes, threads, synchronization and coordination, deadlocks, scheduling, linking, memory management, inter-process communication, file systems, devices, and virtual machines. Knowing how operating systems realize a number of basic abstractions on top of the naked hardware and which strategies they apply while managing resources is crucial for any programmer who wants to write programs that can be executed efficiently.

Students are expected to have a working knowledge of the C programming language and a basic understanding of data representations and computer architecture. A key learning goal, and for some students a learning challenge, is to get used to concurrency and non-sequential control flows.



CONSTRUCTOR UNIVERSITY

https://cnds.jacobs-university.de/courses/os-2023

Table of Contents

Source Codes Examples

Part I

Operating Systems

We start by defining what we recognize as an operating system and afterwards we discuss general operating system requirements and services. We briefly define different types of operating systems and we look at software architectures that were used to construct operating systems. Since the discussion of these topics tends to be a bit 'academic', we also look at different implementations of "hello world" programs in order to get an idea about the difference between system calls and library calls, or how different ways to compile and link programs influence the size of executables.

By the end of this part, students should be able to

- define what an operating system is;
- sketch the hardware and software layers on a computer system;
- explain the difference between library calls and system calls;
- understand buffered stream I/O provided by the C library;
- name services provided by operating systems;
- describe the difference between user and system mode;
- summarize different ways to enter the operating system kernel;
- distinguish between concurrency and parallelism;
- recall the separation of mechanisms from policies design principle;
- contrast different types of operating systems;
- describe different operating system architectures.

Section 1: Definition, Requirements and Services

What is an Operating System?

- An operating system is similar to a government. . . Like a government, the operating system performs no useful function by itself. (A. Silberschatz, P. Galvin)
- The most fundamental of all systems programs is the operating system, which controls all the computer's resources and provides the basis upon which the application programs can be written. (A.S. Tanenbaum)
- An operating system (OS) is system software that manages computer hardware and software resources, and provides common services for computer programs. (Wikipedia, 2023-05-13)

For computer scientists, the operating system is the system software providing an abstraction on which application software can be written while hiding the details of the hardware components from the application programmer and making application programs portable. For ordinary people, the operating system is often associated with the (graphical) user interface running on top of what computer scientists understand as the operating system. This is understandable since the operating system underlying the graphical user interface is largely invisible to ordinary people. In this course, we do not discuss user interface or usability aspects. The goal of this course is to explain how an operating system provides the services necessary to execute programs and how essential abstractions provided to application programmers are realized.

A second important topic we are discussing is concurrency. To achieve good performance, it is necessary to exploit concurrency at the hardware level. And this is meanwhile not only true for operating systems but also for applications since the number of processor cores is increasing steadily. Hence, we will study primitives that support the implementation of concurrent programs.

A large number of operating systems have been implemented since the 1960s. They differ significantly in their functionality since they target different environments. Some examples of operating systems:

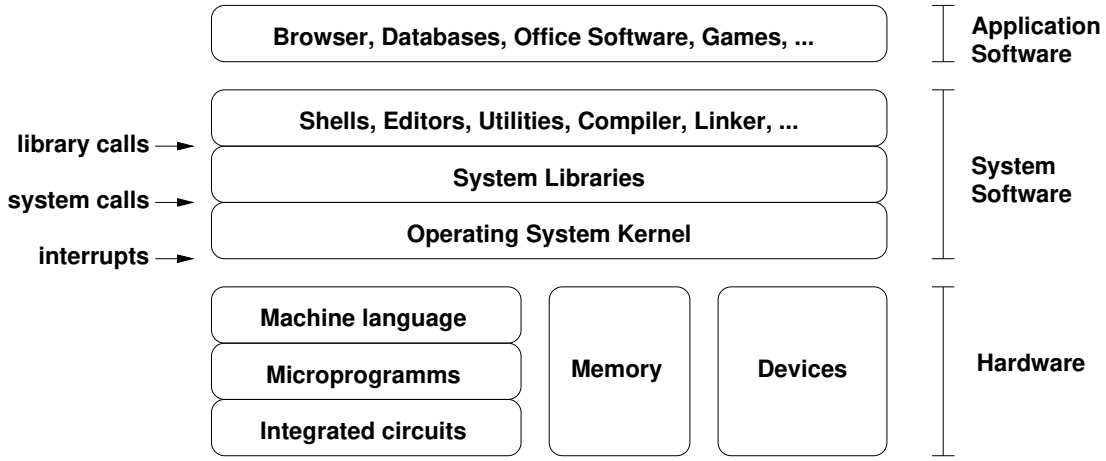
- Unix (AT&T), Solaris (Sun), HP-UX (HP), AIX (IBM), MAC OS X (Apple)
- BSD, NetBSD, FreeBSD, OpenBSD, Linux (e.g., Debian, Ubuntu, Fedora)
- Windows (Microsoft), MAC OS (Apple), OS/2 (IBM)
- MVS (IBM), OS/390 (IBM), BS 2000 (Siemens)
- VxWorks (Wind River Systems), Embedded Linux (e.g., OpenWrt), Embedded BSD
- iOS (Apple), Android (Google), Symbian (Nokia)
- Zephyr, FreeRTOS, RIOT, Contiki, TinyOS, Tock

Implementing and maintaining an operating system is a huge effort. This has lead to some consolidation. For hardware manufacturers it is often cheaper to contribute to an open source operating system instead of developing and maintaining their own operating system.

Further online information:

- **YouTube**: AT&T Archives: The UNIX Operating System

Hardware vs. System vs. Application



From the operating system perspective, the hardware is mainly characterized by the machine language (also called the instruction set) of the main processors, the memory system, and the I/O busses and interfaces to devices.

The operating system is part of the system software, which includes next to the operating system kernel system libraries and tools like command interpreters and in some cases development tools like editors, compilers, linkers, and various debugging and troubleshooting tools. Operating system distributions usually add software package management functionality to simplify and automate the installation, maintenance, and removal of (application) software.

Applications are build on top of the system software, primarily by using application programming interfaces (APIs) exposed by system libraries. Complex applications often use libraries that wrap system libraries in order to provide more abstract interfaces, to supply generally useful data structures, and to enhance portability by hiding differences of system libraries from application programmers. Examples of such libraries are:

- GLib[1] originating from the Gnome project
- Apache Portable Runtime (APR)[2] originating from the Apache web server
- Netscape Portable Runtime[3] (NSR) originating from the Mozilla web browser
- QtCore of the Qt Framework[4]

Some of these libraries make it possible to write applications that can be compiled to run on very different operating systems, e.g., Linux, Windows and MacOS. Some more recent programming languages like Rust provide a standard library that is designed to hide system differences.

Let us look at some “hello world” programs to better understand library and system calls and the difference between statically and dynamically linked programs.

[1] https://wiki.gnome.org/Projects/GLib
[2] https://apr.apache.org/
[3] https://developer.mozilla.org/en-US/docs/Mozilla/Projects/NSPR
[4] http://doc.qt.io/

```c
/*
 * hello-naive.c --
 *
 *      This program uses the stdio library to print a short message.
 *
 * Exercise:
 *
 * On Linux, run the program with ltrace and strace. Explain the
 * output produced by ltrace and strace.
 */

#include <stdio.h>

int
main(void)
{
    printf("Hello World\n");
    return 0;
}
```

Listing 1: Naive hello world program using C library functions

The program in Listing 1 is pretty much the standard “hello world” program written in C. If you compile it, you will by default get a shared executable where the system’s C library is linked to the executable at program startup time. This makes the executable size reasonably small. (But it is possible to produce much smaller “hello world” programs if small size is desirable.)

The program *always* returns 0, indicating successful execution of the program. There can be situations where the printing of the string fails but this is not reflected in the exit code.

```c
/*
 * hello-stdio.c --
 *
 *      This program uses the stdio library to print a short message.
 *      Note that we check the return code of puts() and that
 *      we flush() the buffered output stream manually to check
 *      whether writing to stdout actually worked.
 *
 * Exercise:
 *
 * On Linux, run the program with ltrace and strace. Explain the
 * output produced by ltrace and strace.
 */

#include <stdio.h>
#include <stdlib.h>

int
main(void)
{
    const char msg[] = "Hello World";
    int n;

    n = puts(msg);
    if (n == EOF) {
        perror("puts");
        return EXIT_FAILURE;
    }

    if (fflush(stdout) == EOF) {
        perror("fflush");
        return EXIT_FAILURE;
    }

    return EXIT_SUCCESS;
}
```

Listing 2: Proper hello world program using C library functions

The program in Listing 2 improves our first naive "hello world" program by properly checking whether the printing of the message was successful. If a problem occured while printing the characters, the program returns a non-zero exit status to indicate that a failure occurred. Furthermore, an error message is written to the standard error.

A common problem is that inexperienced programmers do not properly separate the standard output from the standard error. This can be a catastrophic programming error if output generated by a program is passed on to other programs that suddenly get confused by receiving error messages. Hence, always properly separate a program's output from error messages or debug messages.

```c
/*
 * hello-write.c --
 *
 *      This program invokes the Linux write() system call.
 *
 * Exercise:
 *
 * Statically compile and run the program. Look at the assembler code
 * generated (objdump -S, or gcc -S).
 */

#include <stdlib.h>
#include <unistd.h>

int
main(void)
{
    const char msg[] = "Hello World\n";
    ssize_t n;

    n = write(STDOUT_FILENO, msg, sizeof(msg));
    if (n == -1 || n != sizeof(msg)) {
        const char err[] = "write system call failed\n";
        (void) write(STDERR_FILENO, err, sizeof(err));
        return EXIT_FAILURE;
    }

    return EXIT_SUCCESS;
}
```

Listing 3: Proper hello world program using the write() system call

The program in Listing 3 avoids the usage of the buffered I/O streams provided by the C library. It uses instead the `write()` system call directly to write the message to the standard output. Note that we have to identify the standard output by a file descriptor (a small number identifying an open file). The `STDOUT_FILENO` preprocessor macro resolves to the number of the standard output file descriptor. On Unix systems, the well-known file descriptors are `STDIN_FILENO` (0), `STDOUT_FILENO` (1), and `STDERR_FILENO` (2).

The `write()` system call returns the number of bytes written or -1 to indicate a system call error. At the system call level, it is common practice to indicate errors by returning negative numbers. In order to check whether the writing of the message has failed, we check whether the system call execution failed or we got a short write.

Failing system calls usually leave an error number in the global variable `errno`. Note that `errno` is not modified if a system call succeeds. Hence, `errno` should only be checked after a system call failure. There is a collection of well-defined system call error names that can be accessed by including `errno.h`.

```c
/*
 * hello-syscall.c --
 *
 *      This program invokes the Linux write() system call by using
 *      the generic syscall library function.
 */

#define _GNU_SOURCE

#include <stdlib.h>
#include <unistd.h>
#include <syscall.h>

int
main(void)
{
    const char msg[] = "Hello World\n";
    ssize_t n;

    n = syscall(SYS_write, 1, msg, sizeof(msg));
    if (n == -1 || n != sizeof(msg)) {
        return EXIT_FAILURE;
    }

    return EXIT_SUCCESS;
}
```

Listing 4: Proper hello world program using the Linux syscall() interface

The program in Listing 4 invokes the `write()` system call directly, i.e., without calling the `write()` wrapper function provided by the C library. This example is Linux specific and most likely not portable. Note that the system call is identified by the constant `SYS_write`. This constant is used by the operating system kernel to index into a system call table in order to identify the function implementing the system call in the kernel. This design resembles how an interrupt number is used to index into an interrupt vector to locate the function responsible for handling the interrupt.

General Requirements

- An operating system
  - should be efficient and introduce little overhead;
  - should be robust against malfunctioning application programs;
  - should protect data and programs against unauthorized access;
  - should protect data and programs against hardware failures;
  - should manage resources in a way that avoids shortages or overload conditions.
- Some of these requirements can be contradictory.
- Hence, trade-off decisions must be made while designing an operating system.

Protecting the operating system against malfunctioning applications or isolating applications against each other does have an impact on performance. Similarly, hiding hardware failures from applications usually requires the allocation and management of additional resources. Hence, operating system designers often have to find engineering solutions requiring trade-off decisions.

Services for Application Programs

- Loading of programs, cleanup after program execution
- Management of the execution of multiple programs
- High-level input/output operations (`write()`, `read()`, ...)
- Logical file systems (`open()`, `close()`, `mkdir()`, `unlink()`, ...)
- Control of peripheral devices (keyboard, display, pointer, camera, ...)
- Interprocess communication primitives (signals, pipes, ...)
- Support of basic communication protocols (TCP/IP)
- Checkpoint and restart primitives
- ...

What are the system services needed to execute a hello world program?

Services for System Operation

- User identification and authentication
- Access control mechanisms
- Support for cryptographic operations and the management of keys
- Control functions (e.g., forced abort of processes)
- Testing and repair functions (e.g., file systems checks)
- Monitoring functions (observation of system behavior)
- Logging functions (collection of event logs)
- Accounting functions (collection of usage statistics)
- System generation and system backup functions
- Software management functions
- ...

When did you do your last backup? When did you check the last time that your backup is complete and sufficient to restore your system? Is the backup process you are using automated?

When did you last update your software? Is your software update process automated?

Section 2: Fundamental Concepts

User Mode

In user mode,

- the processor executes machine instructions of (user space) processes;
- the instruction set of the processor is restricted to the so called *unprivileged instruction set*;
- the set of accessible registers is restricted to the so called *unprivileged register set*;
- the memory addresses used by a process are typically mapped to physical memory addresses by a memory management unit;
- direct access to hardware components is protected by using hardware protection where possible;
- direct access to the state of other concurrently running processes is restricted.

The programs that we write and use every day are all running as processes in user mode. Even processes with special priviledges still run in user mode (they just have additional privileges).

System Mode

In system mode,

- the processor executes machine instructions of the operating system kernel;
- all instructions of the processor can be used, the so called *privileged instruction set*;
- all registers are accessible, the so called *privileged register set*;
- direct access to physical memory addresses and the memory address mapping tables is enabled;
- direct access to the hardware components of the system is enabled;
- the direct manipulation of the state of processes is possible.

The operating system kernel generally runs in system mode while processes execute in user mode. By enforcing a hardware assisted separation of the operating system kernel from user space processes, the kernel can protect itself against malfunctioning processes. A robust and well debugged kernel will never die due to a misbehaving user space process. (But as we will see soon, there can be situations where user space processes make a system practically unusable, e.g., by making the kernel really busy, but strictly speaking the kernel still does what it was designed to do in such situations – just slowly.)

Embedded systems sometimes lack the hardware support that is necessary to enforce a clear separation of user mode from system mode. Such systems are by design less robust than systems that can use hardware assisted separation since programming errors in application code (or malware in application code) can impact the behavior of the entire system.

Entering the Operating System Kernel

- System calls (supervisor calls, software traps)
  - Synchronous to the running process
  - Parameter transfer via registers, the call stack or a parameter block
- Hardware traps
  - Synchronous to a running process (devision by zero)
  - Forwarded to a process by the operating system
- Hardware interrupts
  - Asynchronous to the running processes
  - Call of an interrupt handler via an interrupt vector
- Software interrupts
  - Asynchronous to the running processes

The operating system kernel exists to support applications and to coordinate resource requests. As such, the operating system kernel is not constantly running but instead most of the time waiting for something to happen that requires the kernel's intervention.

- System calls are invoked by a process when the process needs services provided by the operating system kernel. A system call looks like a library function call but the mechanics of performing a system call are way more complex since a system call requires a transition from user mode into system mode.
- Hardware traps are signaled by a hardware component (i.e., via hardware interrupts) but caused by the execution of a user-mode process. A hardware trap occurs because a user space process was trying to do something that is not well defined or not allowed. When a hardware trap occurs, the user space process is stopped and the kernel investigates which process was causing the trap and which action needs to be taken to resolve the situation.
- Hardware interrupts are any hardware interrupts that are not triggered by a user space process. For example, an interrupt may signal that a network packet has been received. When an interrupt occurs, a running user space process may be stopped and the kernel investigates how the interrupt needs to be handled.
- Software interrupts are signaling a user space process that something exceptional has happened. A user space process, when receiving a software interrupt, may change its normal execution path and jump into a special function that handles the software interrupt. On Unix systems, software interrupts are implemented as signals.

Note that system calls are much more expensive than library calls since system calls require a transition from user mode to system mode and finally back to user mode. Efficient programs therefore tend to minimize the system calls they need to perform.

Concurrency versus Parallelism

Definition (concurrency)

An application or a system making progress on more than one task at the same time is using *concurrency* and is called *concurrent*.

Definition (parallelism)

An application or a system executing more than one task at the same time is using *parallelism* and is called *parallel*.

- Concurrency does not require parallel execution.
- Example: A web server running on a single CPU handling multiple clients.

As someone said (unknown source):

> Concurrency is like having a juggler juggle many balls. Regardless of how it seems, the juggler is only catching/throwing one ball per hand at a time. Parallelism is having multiple jugglers juggle balls simultaneously.

Concurrency improves efficiency since waiting times can be used for doing other useful things. An operating system kernel organizes a concurrent world and usually is internally concurrent as well. On computing hardware that has multiple CPU cores, concurrent programs and operating systems can explore the parallelism enabled by the hardware.

The Go programming languages was designed to make it easy to write concurrent programs. A Go program can easily have thousands of concurrent activities going on that are executed by the Go runtime using a typically much smaller number of operating system level “threads” that explore the parallelism enabled by multi-core CPUs.

Separation of Mechanisms and Policies

- An important design principle is the separation of policy from mechanism.
- Mechanisms determine *how* to do something.
- Policies decide `what` will be done.
- The separation of policy and mechanism is important for flexibility, especially since policies are likely to change.

Good operating system designs (or good software designs in general) separate mechanisms from policies. Instead of hard-wiring certain policies in an implementation of a function, it is better to expose mechanism with which different policies and be enforced.

Examples:

- An operating system implements a packet filter, which provides mechanisms to filter packets based on a variety of properties of a packet. The exact policies detailing which types of packets are filtered is provided as a set of packet filter rules at runtime.
- An operating system kernel provides mechanisms to enforce access control rules on file system objects. The definition of the access control rules, i.e., the access control policy, is left to be provided by the user of the system.

Good separation of mechanisms and policies leads to systems that can be adapted to different usage scenarios in flexible ways.

Section 3: Types of Operating Systems

Operating systems can be classified by the types of computing environments they are designed to support:

- Batch processing operating systems
- General purpose operating systems
- Parallel operating systems
- Distributed operating systems
- Real-time operating systems
- Embedded operating systems

Subsequent slides provide details about these different operating system types.

Batch Processing Operating Systems

- Characteristics:
  - Batch jobs are processed sequentially from a job queue
  - Job inputs and outputs are saved in files or printed
  - No interaction with the user during the execution of a batch program
- Batch processing operating systems were the early form of operating systems.
- Batch processing functions still exist today, for example to execute jobs on super computers.

General Purpose Operating Systems

- Characteristics:
  - Multiple programs execute simultaneously (multi-programming, multi-tasking)
  - Multiple users can use the system simultaneously (multi-user)
  - Processor time is shared between the running processes (time-sharing)
  - Input/output devices operate concurrently with the processors
  - Network support but no or very limited transparency
- Examples:
  - Linux, BSD, Solaris, ...
  - Windows, MacOS, ...

We often think of general purpose operating systems when we talk about operating systems. While general purpose operating systems do play an important role, we often neglect the large number of operating systems we find in embedded devices.

Parallel Operating Systems

- Characteristics:
  - Support for a very large number of tightly integrated processors
  - Symmetrical
    - Each processor has a full copy of the operating system
  - Asymmetrical
    - Only one processor carries the full operating system
    - Other processors are operated by a small operating system stub to transfer code and tasks
- Massively parallel systems are a niche market and hence parallel operating systems are usually very specific to the hardware design and application area.

Distributed Operating Systems

- Characteristics:
  - Support for a medium number of loosely coupled processors
  - Processors execute a small operating system kernel providing essential communication services
  - Other operating system services are distributed over available processors
  - Services can be replicated in order to improve scalability and availability
  - Distribution of tasks and data transparent to users (single system image)
- Examples:
  - Amoeba (Vrije Universiteit Amsterdam)
  - Plan 9 (Bell Labs, AT&T)

Some distributed operating systems aimed at providing a single system image to the user where a user interacts with a system that hides the fact that the underlying hardware is a loosely coupled collection of many computers. The idea was to provide transparency by hiding where computations take place or where data is actually stored and by masking failures that occur in the system.

Real-time Operating Systems

- Characteristics:
  - Predictability
  - Logical correctness of the offered services
  - Timeliness of the offered services
  - Services are to be delivered not too early, not too late
  - Operating system executes processes to meet time constraints
- Examples:
  - QNX
  - VxWorks
  - RTLinux, RTAI, Xenomai
  - Windows CE

A hard real-time operating system guarantees to always meet time constraints. A soft real-time operating system guarantees to meet time constraints most of the time. Note that a real-time system does not require a super fast processor or something like that. What is required is predictability and this implies that for every operating system function there is a defined upper time bound by which the function has to be completed. The operating system never blocks in an uncontrolled manner.

Hard real-time operating systems are required for many things that interact with the real word such as robots, medical devices, computer controlled vehicles (cars, planes, ...), and many industrial control systems.

Embedded Operating Systems

- Characteristics:
  - Usually real-time systems, sometimes hard real-time systems
  - Very small memory footprint (even today!)
  - No or limited user interaction
  - 90-95 % of all processors are running embedded operating systems
- Examples:
  - Embedded Linux, Embedded BSD
  - Symbian OS, Windows Mobile, iPhone OS, BlackBerry OS, Palm OS
  - Cisco IOS, JunOS, IronWare, Inferno
  - Contiki, TinyOS, RIOT, Mbed OS, Tock

Special variants of Linux and BSD systems have been developed to support embedded systems and they are gaining momentum. On mobile phones, the computing resources are meanwhile big enough that mobile phone operating systems tend to become variants of general purpose operating systems. There are, however, a fast growing number of systems that run embedded operating systems as the Internet is reaching out to connect things (Internet of Things).

Some notable Linux variants:

- OpenWRT[5] (low cost network devices)
- Raspbian[6] (Raspberry Pi)

[5] https://openwrt.org/
[6] https://www.raspbian.org/

Evolution of Operating Systems

- 1st Generation (1945-1955): Vacuum Tubes
  - Manual operation, no operating system
  - Programs are entered via plugboards
- 2nd Generation (1955-1965): Transistors
  - Batch systems automatically process job queues
  - The job queue is stored on magnetic tapes
- 3rd Generation (1965-1980): Integrated Circuits
  - Spooling (Simultaneous Peripheral Operation On Line)
  - Multiprogramming and Time-sharing
- 4th Generation (1980-2000): VLSI
  - Personal computer (CP/M, MS-DOS, Windows, Mac OS, Unix)
  - Network operating systems (Unix)
  - Distributed operating systems (Amoeba, Mach, V)

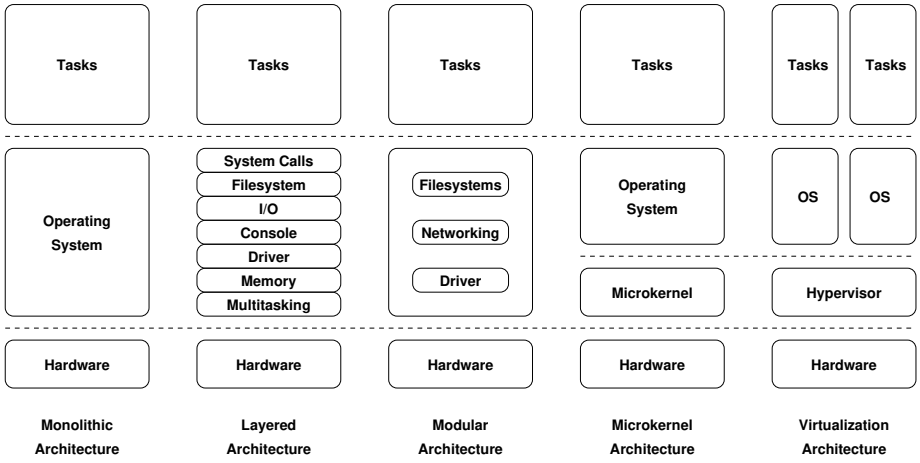
The development since 2000 is largely driven by virtualization techniques such as virtual machines or containers and software systems that manage very large collections of virtual machines and containers. Some notable open source systems:

- OpenStack[7]
- OpenNebula[8]
- Docker[9]
- Kubernetes[10]

[7] https://www.openstack.org/
[8] https://opennebula.org/
[9] https://www.docker.com/
[10] https://kubernetes.io/

Section 4: Operating System Architectures

Operating System Architectures



Monolithic Kernel Architecture: A monolithic kernel is a collection of functions without a structure (the big mess). All services are implemented in the kernel with the same privilege level. Monolithic kernels are difficult to maintain and often lack reliability since they are hard to debug. A programming mistake anywhere in the code can cause arbitrary side effects and failures in other parts of the code. Monolithic architectures can be very time and space efficient. Monolithic kernels are often found on embedded systems where often the interface between the kernel and application code blurs. In fact, some operating systems for embedded systems choose to compile everything in a single compiler run so that the compiler can do optimizations across source file boundaries.

Layered Kernel Architecture: In the early days of kernel designs, several projects tried to construct strictly layered kernels where each new layer adds functionality to the layer below and the layers are clearly separated [12]. The idea was that layered architectures are a rigorous implementation of a stacked machine perspective and easier to maintain. The downside of the strictly layered architecture is the overhead of going through multiple layer interfaces even for relatively simple functions.

Modular Kernel Architecture: A modular kernel architecture divides the kernel into several modules. Modules can be platform independent. The architecture enforces a certain separation of the modules in order to increase reliability and robustness. However, it is still a monolithic kernel architecture with a single priviledge level where programming mistakes can have drastic consequences. Modular kernels can achieve performance close to pure monolithic kernels while allowing the kernel code to grow significantly in size and complexity. The Linux kernel uses a modular architecture.

Microkernel Architecture: Microkernel architectures implement basic multi-tasking, memory management, and inter-process communication facilities in the microkernel. All other operating system functions are implemented outside of the microkernel. The goal of this design is to improve the robustness since failures in device drivers do not necessarily lead to failures of the entire operating system.

Virtualization Architecture: Virtual machines were invented in the 1970s (IBM VM/370 in 1979) and reinvented in the 1990s (VMware in 1992, XEN in 2003). The idea is to have a very small software layer running on top of the hardware that virtualizes the hardware. The goal in the 1970s was to run different operating systems concurrently on a single computer. Virtual machines were reinvented in the 1990s when PC hardware became powerful enough to support virtualization. Meanwhile, virtualization technology is a foundation of cloud data centers that are often able to migrate running virtual machines from one physical computer to another. Virtual machine technology can also be used in several meaningful ways on desktop computers, but this is not yet as popular as the usage on the server (data center) side.

Kernel Modules / Extensions

- Implement large portions of the kernel like device drivers, file systems, networking protocols etc. as loadable kernel modules
- During the boot process, load the modules appropriate for the detected hardware and necessary for the intended purpose of the system
- A single software distribution can support many different hardware configurations while keeping the (loaded) kernel size small
- Potential security risks since kernel modules must be trusted (some modern kernels only load signed kernel modules)
- On high security systems, consider disabling kernel modules and instead building custom kernel images

Kernel modules are the simplest way to learn writing kernel code on Linux since they can be developed easily outside of the Linux kernel source tree.

```c
/*
 * This is a sample hello world Linux kernel module. To compile it on
 * Debian or Ubuntu, you need to install the Linux kernel headers:
 *
 *   sudo apt-get install linux-headers-$(uname -r)
 *
 * Then type make and you are ready to install the kernel module:
 *
 *   sudo insmod ./hello.ko
 *   sudo lsmod
 *   sudo rmmod hello
 *
 * To inspect the module try this:
 *
 *   sudo modinfo ./hello.ko
 */

#include <linux/kernel.h>
#include <linux/module.h>

MODULE_AUTHOR("Juergen Schoenwaelder");
MODULE_LICENSE("Dual BSD/GPL");
MODULE_DESCRIPTION("Simple hello world kernel module.");

static char *msg = "hello world";
module_param(msg, charp, 0000);
MODULE_PARM_DESC(msg, "A message to emit upon module initialization");

static const char* modname = __this_module.name;

static int __init hello_init(void)
{
    printk(KERN_DEBUG "%s: initializing...\n", modname);
    printk(KERN_INFO "%s: %s\n", modname, msg);
    return 0;
}

static void __exit hello_exit(void)
{
    printk(KERN_DEBUG "%s: exiting...\n", modname);
}

module_init(hello_init);
module_exit(hello_exit);
```

Listing 5: Hello world from within the kernel (Linux kernel module)

Selected Relevant Standards

Organization	Standard	Year
ANSI/ISO	C Language (ISO/IEC 9899:1999)	1999
ANSI/ISO	C Language (ISO/IEC 9899:2011)	2011
ANSI/ISO	C Language (ISO/IEC 9899:2018)	2018
IEEE	Portable Operating System Interface (POSIX:2001)	2001
IEEE	Portable Operating System Interface (POSIX:2008)	2008
IEEE	Portable Operating System Interface (POSIX:2017)	2017

The table lists standards that are currently important. Historically, there have been many standardization efforts, some became irrelevant, others became part of other standards. The organizations driving standards range from companies (AT&T) over industry consortia (X/Open, Open Group) to independent standards developing organizations (IEEE, ISO).

The C library used on many Linux systems supports multiple standards. Source code can declare to which standards it complies by defining a preprocessor symbol before including any header files:

```
1  #define _POSIX_SOURCE             /* POSIX standards and ISO C */
2
3  #define _POSIX_C_SOURCE 200112L   /* POSIX 1003.1-2001 */
4  #define _POSIX_C_SOURCE 200809L   /* POSIX 1003.1-2008 */
5
6  #define _ISOC99_SOURCE            /* ISO/IEC 9899:1999 */
7  #define _ISOC11_SOURCE            /* ISO/IEC 9899:2011 */
8
9  #define _DEFAULT_SOURCE           /* collection of standards */
10 #define _GNU_SOURCE               /* collection of standards and extensions */
```

POSIX P1003.1 Standard

Name	Title
P1003.1a	System Interface Extensions
P1003.1b	Real Time Extensions
P1003.1c	Threads
P1003.1d	Additional Real Time Extensions
P1003.1j	Advanced Real Time Extensions
P1003.1h	Services for Reliable, Available, and Serviceable Systems
P1003.1g	Protocol Independent Interfaces
P1003.1m	Checkpoint/Restart
P1003.1p	Resource Limits
P1003.1q	Trace

The POSIX standards most relevant for this module are P1003.1a and P1003.1c standards.

Some Useful Linux System Utilities

Command	Description
`strace`	trace system calls and signals
`ltrace`	trace library calls (and system calls)
`time`	run programs and summarize system resource usage
`readelf`	display information about ELF files
`objdump`	display information from object files
`nm`	list symbols from object files
`ldd`	print shared object dependencies
`stat`	display file or file system status
`xxd`	make a hexdump or do the reverse

Examples:

```
# trace all system calls triggered by the execution of /bin/hello
strace /bin/echo hello world

# trace all library calls triggered by the execution of /bin/hello
ltrace /bin/echo hello world

# measure the execution time of /bin/echo
time /bin/echo hello world

# provide detailed resource usage information for /bin/echo
/usr/bin/time -v /bin/echo hello world

# read the information stored in the executable /bin/hello
readelf -a /bin/echo

# disassemble the program /bin/echo
objdump -d /bin/echo

# display the different sections of /bin/echo
objdump -s /bin/echo

# list the shared libraries /bin/echo depends on
ldd /bin/echo

# display the meta-information about the file /bin/echo
stat /bin/echo

# display the raw content of the file /bin/echo
xxd /bin/echo
```

Note that commands like `echo` and `time` are often built into the shell directly. In order to execute the full flavored versions of these commands, it is necessary to specify the (full) path to the executable file. For details and options of the commands, please consult the manual pages.

Part II
Hardware

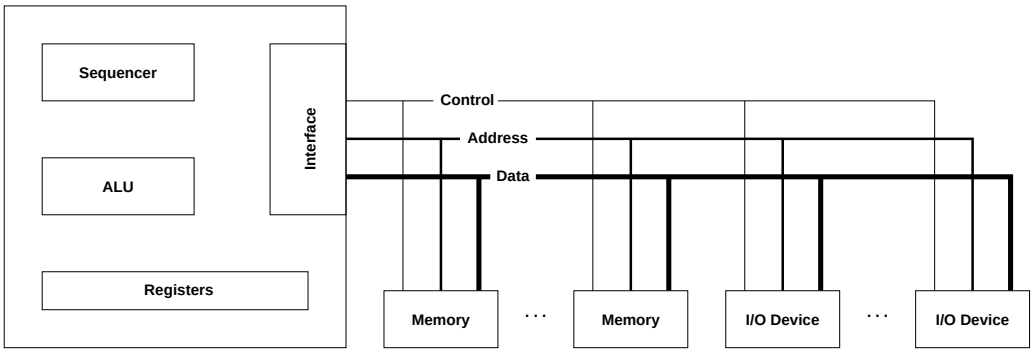
In this part we review some basic concepts of computer architecture that are relevant for understanding operating systems. This is mostly a refresher of material covered by other introductory modules. Note that the model of a computer we are using here is a significant simplification, real systems are much more complex.

By the end of this part, students should be able to

- describe the von Neumann computer architecture;
- summarize CPU registers and instruction sets;
- sketch the memory hierarchy;
- explain caching and when caching mechanisms are effective;
- map elements of a program to the different memory segments;
- illustrate how function calls are carried out;
- outline I/O programming techniques;
- demonstrate how interrupts are handled.

Section 5: Computer Architecture and Processors

Computer Architecture (von Neumann)



- Today's common computer architecture uses busses to connect memory and I/O systems to the central processing unit (CPU)

The central processing unit (CPU) is connected to the main memory and other devices using the system bus. The system bus consists of the data bus, an address bus, and a control bus. Data is carried over the data bus to/from the address carried over the address bus. The control bus signals the direction of the data transfer and when the transfer takes place. The usage of shared system busses to connect components of a computer requires arbitration, synchronization, interrupts, priorities.

A CPU consists of a command sequencer fetching instructions, an arithmetic logic unit (ALU), and a set of registers. A CPU is primarily characterized by its instruction set. Modern CPUs often have multiple cores, i.e., multiple ALUs and register sets that can work concurrently.

CPU Registers and Instruction Sets

- Typical CPU registers:
  - Processor status register
  - Instruction register (current instruction)
  - Program counter (current or next instruction)
  - Stack pointer (top of stack)
  - Special privileged registers
  - Dedicated registers
  - Universal registers
- Non-privileged instruction set:
  - General purpose set of CPU instructions
- Privileged instruction set:
  - Access to special resources such as privileged registers or memory management units
  - Subsumes the non-privileged instruction set

CPUs used by general purpose computers usually support multiple privilege levels. The Intel x86 architecture, for example, supports four privilege levels (protection rings 0...3). Note that CPUs for small embedded systems often do not support multiple privilege levels and this has serious implications on the robustness an operating system can achieve. In the following, we focus primarily on operating systems that run on hardware supporting multiple CPU privilege levels. Hardware-assisted privilege levels or protection modes is slowly but surely becoming more widely available in embedded hardware to enable some level of trusted computing.

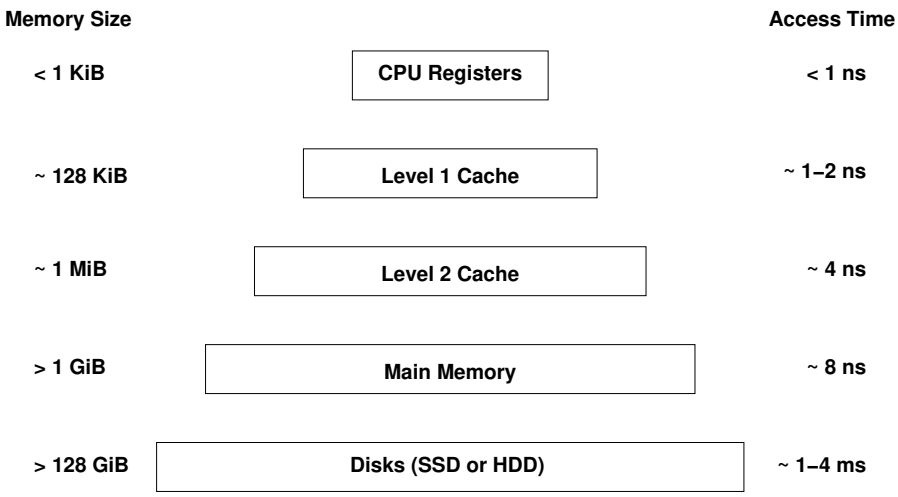
Today, most of our desktop systems and servers use processors implementing the x86-64 instruction set. This is a 64-bit extension of the original 32-bit x86 instruction set developed by the Intel Corporation (US). The x86-64 instruction set was defined by the US-based company Advanced Micro Devices (AMD), hence it was also initially known as amd64.

Many mobile devices use ARM processors. ARM Limited (UK) does not produce and sell processors but they merely make money by selling licenses for their processor designs. Companies often extend the licensed processor design with additional features that are tailored to their products.

Since recently, there is a push towards open-source processor architectures that are not covered by commercial licenses. A prominent example is the RISC-V instruction set developed by a project led by the University of Berkeley. The RISC-V processor design has been released under a BSD license and is gaining traction and has very good support by open source development tools. The development is meanwhile coordinated by the non-profit RISC-V International association (based in Switzerland), also known as the RISC-V Foundation.

Section 6: Memory, Caching, Segments, Stacks

Memory Sizes and Access Times



There is a trade-off between memory speed and memory size. CPU registers are very fast to access and update. The main memory is comparatively slow but much larger. Since the CPU has to wait for the slow main memory, most CPUs have additional cache memory on the chip that is faster than the main memory but also much smaller. As a consequence, a CPU runs only a full speed if the cache memories have the “right” data cached. If a program accesses main memory in a way that violates the assumptions made by the caching logic, the program will run slowly. Modern compilers try to optimize code in order to maximize cache hits.

In a similar way, unused main memory is often used as a cache for data stored on bigger but slower disks. In addition, disks may be used to extend the main memory to sizes that are larger than the physically present main memory.

Caching

- Caching is a general technique to speed up memory access by introducing smaller and faster memories which keep a copy of frequently / soon needed data
- *Cache hit*: A memory access which can be served from the cache memory
- *Cache miss*: A memory access which cannot be served from the cache and requires access to slower memory
- *Cache write through*: A memory update which updates the cache entry as well as the slower memory cell
- *Delayed write*: A memory update which updates the cache entry while the slower memory cell is updated at a later point in time

There are several caches in modern computing systems. Data essentially moves through the cache hierarchy until it is finally manipulated in CPU registers. To run CPUs at maximum speed, it is necessary that data that is needed in the next instructions is properly cached since otherwise CPUs have to wait for data to be retrieved from slow memory systems. In order to fill caches properly, CPUs have gone as far as executing machine instructions in a speculative way (e.g., while waiting for a slow memory transfer). Speculative execution has lead to a number of attacks on caches (Spectre).

Locality

- Cache performance relies on:
  - *Spatial locality*:
    Nearby memory cells are likely to be accessed soon
  - *Temporal locality*:
    Recently addressed memory cells are likely to be accessed again soon
- Iterative languages generate linear sequences of instructions (spatial locality)
- Functional / declarative languages extensively use recursion (temporal locality)
- CPU time is in general often spend in small loops/iterations (spatial and temporal locality)
- Data structures are organized in compact formats (spatial locality)

Operating systems often use heuristics to control resources. A common assumption is that application programs have spatial and temporal locality when it comes to memory access. For programs that do not have locality, operating systems may make rather poor resource allocation decisions.

As a programmer, it is useful to be aware of resource allocation strategies used by the operating system if the goal is to write highly efficient application programs.

Memory Segments

Segment	Description
text	machine instructions of the program
data	static and global variables and constants, may be further devided into initialized and uninitialized data
heap	dynamically allocated data structures
stack	automatically allocated local variables, management of function calls (parameters, results, return addresses, automatic variables)

- Memory used by a program is usually partitioned into different segments that serve different purposes and may have different access rights

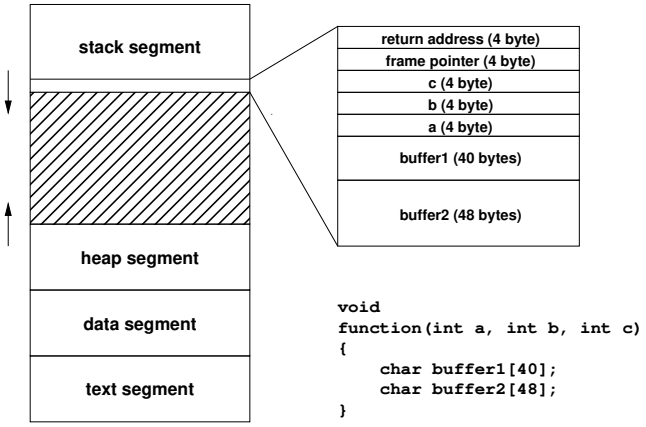
The text segment usually has a fixed size and is read-only and executable. The initialized data segment usually also has a fixed size and it may be partially read-only (constants) and partially read-write (global and static variables). The uninitialized data segment is read-write and it usually also has a fixed size.

The heap segment stores dynamically allocated data structures. It is read-write and it can grow and shrink (but shrinking is rare in practice). The stack segment grows with every function call and it shrinks with every function return. It is usually read-write although it used to be also executable for a long time, leading to many security issues.

On modern operating systems, the memory layout of running programs is often randomized (address space randomization). This is done in order to make it a bit harder for writers of malware and attack code to know (or predict) where in memory specific data is stored.

Stack Frames

- Every function call adds a stack frame to the stack
- Every function return removes a stack frame from the stack
- Stack frame layout is processor specific (here Intel x86)



Stacks are necessary for realizing nestable function calls. We often take it for granted that stack space is available when we call a function. This, however, is not necessarily always the case. Hence, as a good programmer, it makes sense to limit the size of automatic variables allocated on the stack.

The x86 assembly code related to the C function shown on the slide may look as follows:

```
function:
        pushq   %rbp              ; push frame pointer on the stack
        movq    %rsp, %rbp        ; stack pointer becomes base pointer
        movl    %edi, -100(%rbp)  ; copy a (passed via edi) to the stack
        movl    %esi, -104(%rbp)  ; copy b (passed via esi) to the stack
        movl    %edx, -108(%rbp)  ; copy c (passed via edx) to the stack
                                  ; ...
        popq    %rbp              ; pop the frame pointer from the stack
        ret                       ; pop return address from stack and jump

main:
                                  ; ...
        movl    $3, %edx          ; load value into register edx (parameter c)
        movl    $2, %esi          ; load value into register esi (parameter b)
        movl    $1, %edi          ; load value into register edi (parameter a)
        call    function          ; push return address on stack and jump
```

Note that `function` does not "reserve" the space that it is using for the data on the stack. It is using the so called "red zone", which can be used without "reserving it" as long as no other functions are called by a "leaf function". If `function` would call another function, then it would have to update the stack pointer to make sure function local data is preserved.

The `main` assembly code loads values into the registers that are used to pass parameters to a function (which is defind in the processor's calling convention). The `main` function apparently has no local automatic data, since otherwise it would have to adjust the stack pointer in order to protect the data.

For a "near" function call, the `call` instruction pushes the eip register (the instruction pointer) to the stack and sets the eip register to the starting address of the function's code. For a "near" function return, the `ret` instruction pops the eip register (the return address) from the stack.

Example

```
static int foo(int a)
{
    static int b = 5;
    int c;

    c = a * b;
    b += b;
    return c;
}

int main(int argc, char *argv[])
{
    return foo(foo(1));
}
```

- What is returned by `main()`?
- Which memory segments store the variables?

In the example, `b` is stored in the initialized data segment (since it is static), `a` and `c` are stored in the stack frame of a `foo()` function call, `argc` and `argv` are stored in the stack frame of the `main()` function call.

Further online information:

- **YouTube**: Assembly, System Calls, and Hardware in C++

Stack Smashing Attacks

```
#include <string.h>

static void foo(char *bar)
{
    char c[12];
    strcpy(c, bar);  // no bounds checking
}

int main(int argc, char *argv[])
{
    for (int i = 1; i < argc; i++) foo(argv[i]);
    return 0;
}
```

- Overwriting a function return address on the stack
- Returning into a 'landing area' (typically sequences of NOPs)
- Landing area is followed by shell code (code to start a shell)

Since programming languages such as C or C++ do not restrict memory access to properly allocated data objects, it is the programmer's responsibility to ensure that buffers are never overrun or underrun and that pointers point to valid memory areas. Unfortunately, many programs fail to implement this correctly, partly due to laziness, partly due to programming errors. As a consequence, programs written in C or C++ often contain bugs that can be exploited to change the control flow of a program. While there are some defense techniques that make it more difficult to exploit such programming bugs, there are also an increasing number of tools that can systematically find such programming problems.

For C and C++ programmers, there is no alternative to developing the discipline to always ensure that uncontrolled access to memory is prevented, i.e., making it a habit to always write robust code.

Section 7: Devices and Interrupts

Basic I/O Programming

- *Status driven*: the processor polls an I/O device for information
  - Simple but inefficient use of processor cycles
- *Interrupt driven*: the I/O device issues an interrupt when data is available or an I/O operation has been completed
  - *Program controlled*: Interrupts are handled by the processor directly
  - *Program initiated*: Interrupts are handled by a DMA-controller and no processing is performed by the processor (but the DMA transfer might steal some memory access cycles, potentially slowing down the processor)
  - *Channel program controlled*: Interrupts are handled by a dedicated channel device, which is usually itself a micro-processor

Devices are essential for almost every computer. Typical classes of devices are:

- Clocks, timers
- User-interface devices (displays, keyboards, ...)
- Document I/O devices (scanner, printer, ...)
- Multimedia devices (audio and video equipment)
- Network interfaces (Ethernet, WiFi, Bluetooth, Mobile, ...)
- Mass storage devices
- Sensors and actuators in control applications
- Security tokens and biometric sensors

Device drivers are often the biggest component of general purpose operating system kernels.

Interrupts

- Interrupts can be triggered by hardware and by software
- Interrupt control:
  - grouping of interrupts
  - encoding of interrupts
  - prioritizing interrupts
  - enabling / disabling of interrupt sources
- Interrupt identification:
  - interrupt vectors, interrupt states
- Context switching:
  - mechanisms for CPU state saving and restoring

Interrupt Service Routines

- Minimal hardware support (supplied by the CPU)
  - Save essential CPU registers
  - Jump to the vectorized interrupt service routine
  - Restore essential CPU registers on return
- Minimal wrapper (supplied by the operating system)
  - Save remaining CPU registers
  - Save stack-frame
  - Execute interrupt service code
  - Restore stack-frame
  - Restore CPU registers


```c
typedef void (*interrupt_handler)(void);

void handler_a(void)
{
    save_cpu_registers();
    save_stack_frame();
    interrupt_a_handling_logic();
    restore_stack_frame();
    restore_cpu_registers();
}

void handler_b(void)
{
    save_cpu_registers();
    save_stack_frame();
    interrupt_b_handling_logic();
    restore_stack_frame();
    restore_cpu_registers();
}

/*
 * The interrupt vector is indexed by the interrupt number. Every element
 * contains a pointer to a function handling this specific interrupt.
 */

interrupt_handler interrupt_vector[] =
{
    handler_a,
    handler_b,
    // ...
}

#ifdef HARDWARE
/*
 * The following logic executed by the hardware when an interrupt has
 * arrived and the execution of an instruction is complete:
 */

void interrupt(int x)
{
    handler = NULL;
    save_essential_registers();      // includes instruction pointer
    if (valid(x)) {
        handler = interrupt_vector[x];
    }
    if (handler) handler();
    restore_essential_registers();   // includes instruction pointer
}
#endif
```

Part III

Processes and Threads

Processes are a key abstraction provided by operating systems. A process is simply a program under execution. The operating system kernel manages all properties of a process and all resources assigned to a process by maintaining several data structures in the kernel. These data structures change constantly, for example when new processes are created, when running processes allocate or deallocate resources, or when processes are terminated. There are user space tools to inspect the information maintained in the kernel data structures. But note that these tools usually show you a snapshot only and the snapshot may not even be consistent.

Processes are relatively heavy-weight objects since every process has its own memory, its own collection of open files, etc. In order to exploit hardware with multiple CPU cores, it is desirable to exploit multiple cores within a single process, i.e., within the same memory image. This led to the introduction of threads, which represent a thread of execution within a process.

By the end of this part, students should be able to

- explain how operating systems manage the concurrent execution of programs;
- describe the difference between processes and threads;
- apply the POSIX APIs to create and manage processes;
- use the POSIX APIs to create and manage threads;
- illustrate the notion of context switches;
- describe how processes and threads are represented in an operating system kernel;
- outline how a basic command interpreter (a shell) works.

Section 8: Processes

Process Definition

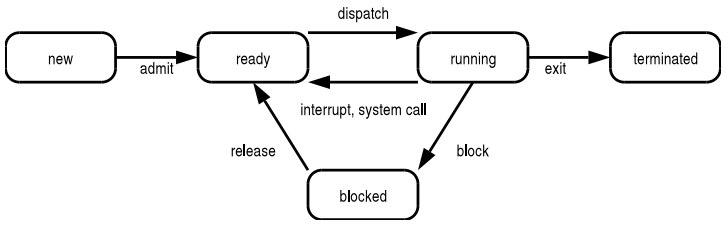
Definition (process)

A process is an instance of a program under execution. A process uses/owns resources (e.g., CPU, memory, files) and is characterized by the following:

1. A sequence of machine instructions determining the behavior (*control flow*) of the running program
2. The current *internal state* of the running program defined by the content of the registers of the processors, the stack, the heap, and the data segments
3. The *external state* of the process defined by the state of other resources used by the running program (e.g., open files, open network connections, running timers, state of devices)

On a Unix system, the shell command `ps` provides a list of all processes on the system. There are many options that can be used to select the information displayed for the processes on the system. Tools like `top` are also popular for displaying process lists.

Processes: State Machine View

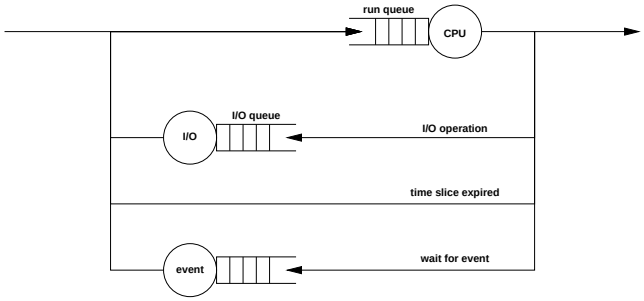


- *new*: just created, not yet admitted
- *ready*: ready to run, waiting for CPU
- *running*: executing, holds a CPU
- *blocked*: not ready to run, waiting for a resource
- *terminated*: just finished, not yet removed

If you run the command line utility `top`, you will see the processes running on the system sorted by some criteria, e.g., the current CPU usage. In the example below, the process state can be seen in the column `S` and the letters mean `R` = running, `S` = sleeping, `I` = idle.

```
top - 20:21:12 up 3 days,  7:16,  1 user,  load average: 0.00, 0.00, 0.00
Tasks:  85 total,   1 running,  84 sleeping,   0 stopped,   0 zombie
%Cpu(s):  0.3 us,  0.3 sy,  0.0 ni, 84.7 id, 14.6 wa,  0.0 hi,  0.0 si,  0.0 st
MiB Mem :    987.5 total,    155.3 free,    132.8 used,    699.4 buff/cache
MiB Swap:   1997.3 total,   1997.3 free,      0.0 used.    687.3 avail Mem

  PID USER      PR  NI    VIRT    RES    SHR S  %CPU  %MEM     TIME+ COMMAND
21613 schoenw   20   0   16964   4776   3620 R   0.3   0.5   0:00.01 sshd
    1 root      20   0  170612  10348   7804 S   0.0   1.0   0:10.49 systemd
    2 root      20   0       0      0      0 S   0.0   0.0   0:00.01 kthreadd
    3 root       0 -20       0      0      0 I   0.0   0.0   0:00.00 rcu_gp
    4 root       0 -20       0      0      0 I   0.0   0.0   0:00.00 rcu_par_gp
```

Processes: Queueing Model View

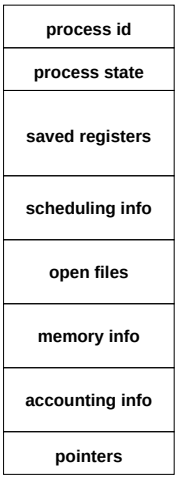


- Processes are enqueued if they wait for resources or events
- Dequeuing strategies can have strong performance impact
- Queueing models can be used for performance analysis and prediction

Unix systems usually keep track of the length of the run queue, i.e., the queue of processes that are runnable and waiting for getting a CPU assigned. The run queue length is typically measured (and smoothed) over 1, 5, and 15 minute intervals and displayed as a the system's load average (see the `top` output on the previous page).

Process Control Block

- Processes are internally represented by a data structure called a process control block (PCB)
  - Process identification
  - Process state
  - Saved registers during context switches
  - Scheduling information (priority)
  - Assigned memory regions
  - Open files or network connections
  - Accounting information
  - Pointers to other PCBs
- PCBs are often enqueued at a certain state or condition

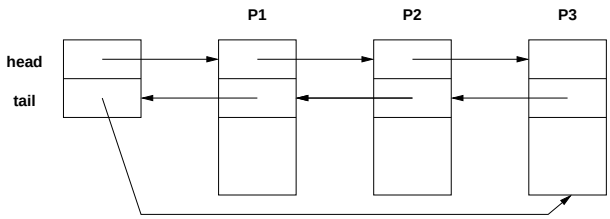


In the Linux kernel, the process control block is defined by the C `struct task_struct`, which is defined in `include/linux/sched.h`. It is a very long struct and it may be interesting to read through its definition to get an idea how central this structure is for keeping the information related to processes organized in the kernel.

Utilities like `ps` or `top` display process information that is obtained from the in-kernel process control blocks. Since user space utilities do not have access to in-kernel data structures, it is necessary to find ways to expose kernel data to user-space programs. In early Unix versions, a common approach was to give selected user space programs access to kernel memory and then user space processes would obtain information directly from kernel data structures. This is, however, tricky from a security perspective and it implies a very tight coupling of user space utilities to kernel data structures. In the 1990's it became popular to expose kernel information via a special file system. User space tools like `ps` or `top` obtain information by reading files in a special process file system and the kernel responds to `read()` and `write` system calls directed to this file system by exposing kernel data, often in a textual easy to parse format. On Linux, the process file system is usually mounted on `/proc` and it exposes every Linux process (task) as a directory (named by the task identifier). Within the directory, there are numerous files that provide information about the state of the process.

Obviously, there is no "atomic" way to read this file system. But in general, it is impossible to take a consistent snapshot of kernel data from user space unless the kernel provides specific features to support the creation of such snapshots. This means the information user space utilities show must be considered an approximation of reality but not the reality.

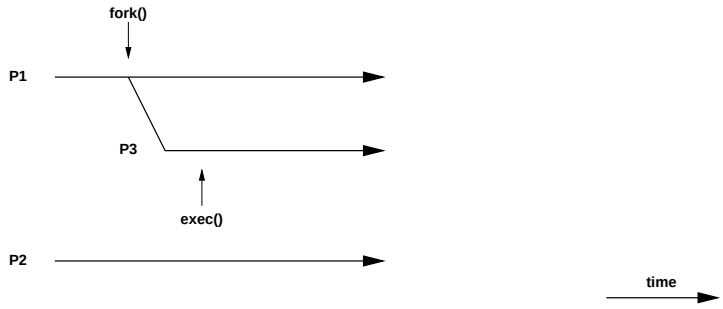
Process Lists



- PCBs are often organized in doubly-linked lists or tables
- PCBs can be queued easily and efficiently using pointer operations
- Run queue length of the CPUs is a good load indicator
- The system load is often defined as the exponentially smoothed average of the run queue length over 1, 5 and 15 minutes

Iterating over the process list is tricky even if you are inside the kernel since the process list can change during the iteration unless precautions are taken that prevent changes during the iteration. The same is true for many members of the data structure representing a process. Kernel programming requires to take care of concurrency issues and it is often required to obtain a number of read and/or write locks in a certain order to complete an activity.

Process Creation

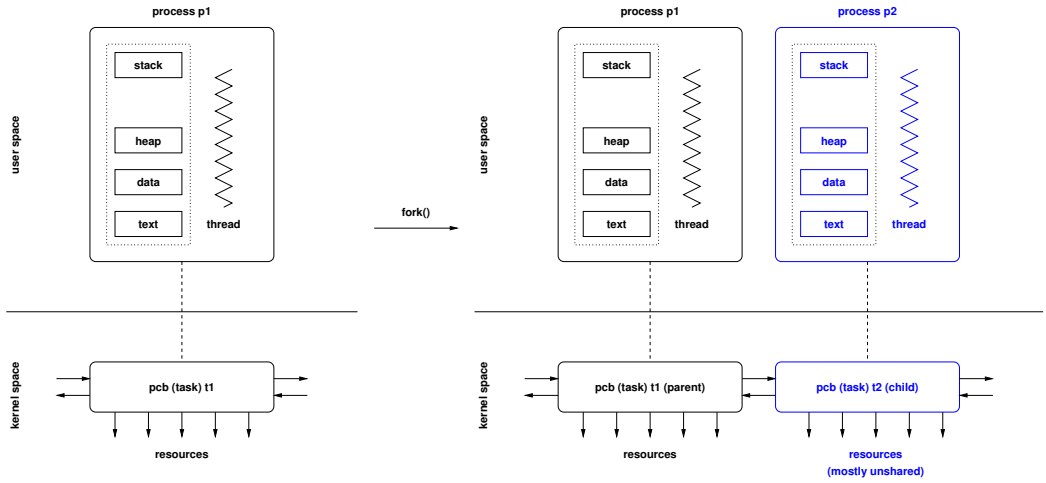


- The `fork()` system call creates a new child process
  - which is an exact copy of the parent process,
  - except that the result of the system call differs
- The `exec()` system call replaces the current process image with a new image.

The fact that creation of a child process and the loading of a new image are two separate system calls is an extremely flexible design since programs creating new child processes can setup the properties of the new child process before it loads a new process image. While of course some properties are always reset to sane defaults during an `exec()` system call, other properties are passed through.

An example where this is used is input and output redirection, which is implemented by modifying the input and output channels of the newly created process before the `exec()` system call is executed.

Process Creation: `fork()`

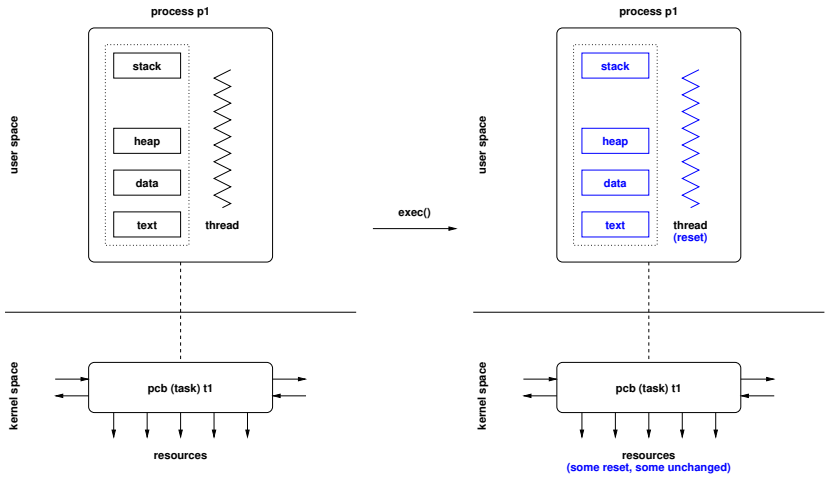


This slides aims to explain the effect of the `fork()` system call. The following happens if a process with its own memory image and thread of control invokes the `fork()` system call:

1. A new process control block is allocated in the kernel and linked to the other process control blocks.
2. A new unused process identifier is assigned to the new process control block.
3. The memory image and the status of the thread invoking the `fork()` system call is (conceptually) cloned and linked to the new process control block.
4. Most of the resources assigned to the original process are copied to the new process, in many cases by cloning resources, but in some cases by sharing resources.
5. Both processes finally return from the `fork()` system call. The process that invoked the `fork()` system call returns the newly allocated process identifier of the child process, the new process returns the value 0.

After the `fork()` has been executed, both processes proceed independently (except for any shared resources they might still have).

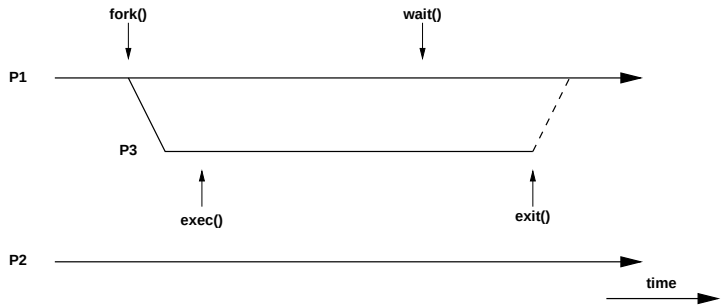
Process Image Replacement: `exec()`



This slides aims to explain the effect of the `exec()` system calls. The `exec()` system calls load a new process image. This essentially means that the running program is replaced by another program. Hence, all memory segments of the process invoking the `exec()` system call are loaded (text and data) from an executable file or reset (heap and stack). Furthermore, the thread of control is reset to start from the entry point of the newly loaded program.

Note that the entry point of a program is not your `main()` function but rather some other function that eventually will call your `main()` function.

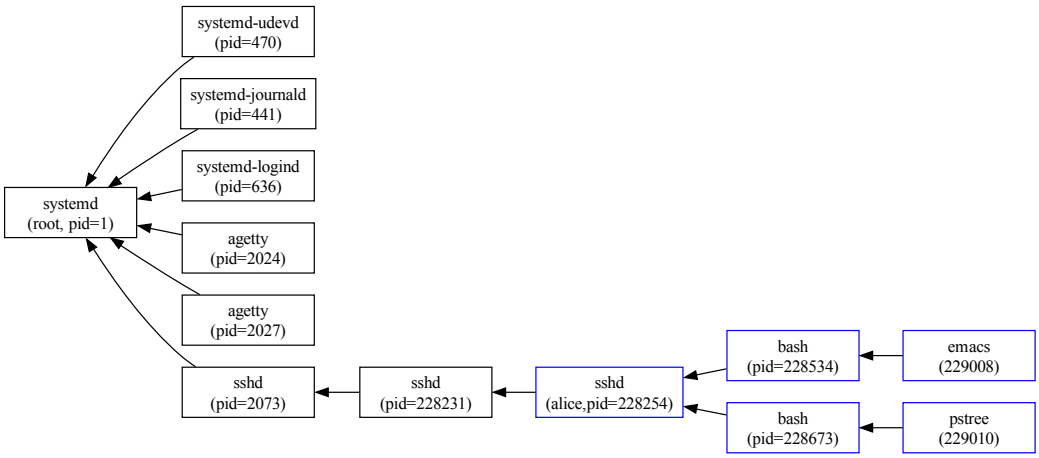
Process Termination



- Processes can terminate themself by calling `exit()`
- The `wait()` system call suspends execution until a child terminates (or a signal arrives)
- Terminating processes return a numeric status code

Strictly speaking, `exit()` is not really a system call but a library function call, which eventually calls the `_exit()` system call. By calling `exit()`, a program returns control to a library function to carry out any cleanup operations before the kernel is asked to exit the process.

Process Trees



The first process is created when the system is initialized. All other processes are created using `fork()`, which leads to a process tree. Processes have an associated user identifier, which is by default inherited and a `fork()` is executed. The (effective) user identity determines access rights and permissions. The PCBs of processes usually maintain a pointer to the PCB of the parent process.

Since processes are organized in a tree, the question arises what happens if a process exists that still has child processes. There are several different possible solutions:

- The exit of the parent process causes all child processes to exit as well.
- The parent process is not allowed to exit until all child processes have exited.
- The parent process is allowed to exit but the child processes have to get a new parent process.

On Unix systems, orphaned processes get a new parent process assigned by the kernel, which is the first process that was created when the system was initialized. On older systems, this process (with process identifier 1) is typically called `initd`. On more recent systems, you may find instead that this process is called `systemd` or `launchd`.

On Linux systems, you can take a quick look at the process tree by running the utility `pstree`. Some process viewers like `htop` can also show the process tree.

POSIX API (fork, exec)

```
#include <unistd.h>

extern char **environ;

pid_t getpid(void);
pid_t getppid(void);

pid_t fork(void);

int execl(const char *path, const char *arg, ...);
int execlp(const char *file, const char *arg, ...);
int execle(const char *path, const char *arg, ..., char * const envp[]);
int execv(const char *path, char *const argv[]);
int execvp(const char *file, char *const argv[]);
int execve(const char *path, char *const argv [], char *const envp[]);
```


On Unix systems, a process has environment variables. These environment variables are stored as an array of strings, where each string has `key=value` format. The utility `env` displays the environment variables of your shell process. Some variables are important since they control where programs are found on your computer or in which language you prefer to interact with programs. In particular, the `PATH` environment variable controls where the shell or the kernel looks for an executable implementing a certain command. A messed up `PATH` environment variable can lead to serious surprises. For any shell scripts that are executed with special privileges, it is of high importance to set the `PATH` environment variable to a sane value before any commands are executed.

The `envp` parameter of the `execve()` and `execle()` calls can be used to control the environment that will be used when executing a new process image.

POSIX API (exit, wait)

```
#include <stdlib.h>
#include <unistd.h>

void exit(int status);
int atexit(void (*function)(void));
void _exit(int status);

pid_t wait(int *status);
pid_t waitpid(pid_t pid, int *status, int options);

#include <sys/time.h>
#include <sys/resource.h>
#include <sys/wait.h>

pid_t wait3(int *status, int options, struct rusage *rusage);
pid_t wait4(pid_t pid, int *status, int options, struct rusage *rusage);
```


The various `wait()` functions document that the original design was found lacking and hence additional parameters were added over time. For most pratical purposes, the waitpid() call is the one you may want to use. The waitpid() call is essentially the same as the `wait4()` call with the last parameter set to `NULL` (and it requires fewer header files to be included).

Listing 6 demonstrates how processes are created and waited for by using the POSIX fork and wait system calls in the C programming language. Listing 7 shows the same program written in the Rust programming language.

```c
/*
 * echo/echo-fork.c --
 *
 *      A simple program to fork processes and to wait for them.
 */

#define _POSIX_C_SOURCE 200809L

#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/wait.h>

static void work(const char *msg)
{
    (void) printf("%s ", msg);
    exit(EXIT_SUCCESS);
}

int main(int argc, char *argv[])
{
    int stat, status = EXIT_SUCCESS;
    pid_t pids[argc];

    for (int i = 1; i < argc; i++) {
        pids[i] = fork();
        if (pids[i] == -1) {
            perror("fork() failed");
            status = EXIT_FAILURE;
            continue;
        }
        if (pids[i] == 0) {
            work(argv[i]);
        }
    }

    for (int i = 1; i < argc; i++) {
        if (pids[i] > 0) {
            if (waitpid(pids[i], &stat, 0) == -1) {
                perror("waitpid() failed");
                status = EXIT_FAILURE;
            }
        }
    }

    (void) puts("");
    if (fflush(stdout) || ferror(stdout)) {
        perror("write failed");
        status = EXIT_FAILURE;
    }

    return status;
}
```

Listing 6: Forking processes and waiting for them to finish (C)

```rust
//! A multi-process program to echo command line arguments.

use nix::sys::wait;
use nix::unistd;
use std::env;
use std::process;

fn work(arg: String) {
    print!("{arg} ");
    process::exit(0);
}

fn main() {
    let mut vec = Vec::new();
    let mut status = 0;
    let mut it = env::args().skip(1).peekable();

    if it.peek().is_none() {
        return;
    }

    for arg in it {
        match unsafe{unistd::fork()} {
            Ok(unistd::ForkResult::Parent { child, .. }) => vec.push(child),
            Ok(unistd::ForkResult::Child) => work(arg),
            Err(msg) => {
                eprintln!("fork() failed: {}", msg);
                status = 1;
            }
        }
    }

    for child in vec {
        match wait::waitpid(child, None) {
            Ok(_) => (),
            Err(msg) => {
                eprintln!("waitpid() failed: {}", msg);
                status = 1;
            }
        }
    }
    println!();

    if status > 0 {
        process::exit(status);
    }
}
```

Listing 7: Forking processes and waiting for them to finish (Rust)

Sketch of a Command Interpreter

```
while (1) {
    show_prompt();                  /* display prompt */
    read_command();                 /* read and parse command */
    pid = fork();                   /* create new process */
    if (pid < 0) {                  /* continue if fork() failed */
        perror("fork");
        continue;
    }
    if (pid != 0) {                 /* parent process */
        waitpid(pid, &status, 0);   /* wait for child to terminate */
    } else {                        /* child process */
        execvp(args[0], args, 0);   /* execute command */
        perror("execvp");           /* only reach on exec failure */
        _exit(1);                   /* exit without any cleanups */
    }
}
```


A basic command interpreter (usually called a shell) is very simple to implement. Ignoring all extra features that a typical good shell has, the core of a minimal shell is simply a loop that reads a command and if it is a valid command, the shell forks a child process that then executes the command while the shell waits for the child process to terminate. Listing 9 shows the core loop written in C. You can find the complete source code in the source code archive.

```c
/*
 * msh/msh.c --
 *
 *      This file contains the simple and stupid shell (msh).
 */

#define _POSIX_C_SOURCE 200809L

#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <errno.h>
#include <sys/types.h>
#include <unistd.h>
#include <sys/wait.h>
#include <assert.h>

#include "msh.h"

int
main(void)
{
    pid_t pid;
    int status;
    int argc;
    char **argv;

    while (1) {
        msh_show_prompt();
        msh_read_command(stdin, &argc, &argv);
        if (argv[0] == NULL || strcmp(argv[0], "exit") == 0) {
            break;
        }
        if (strlen(argv[0]) == 0) {
            continue;
        }
        pid = fork();
        if (pid == -1) {
            fprintf(stderr, "%s: fork: %s\n", progname, strerror(errno));
            continue;
        }
        if (pid == 0) {                 /* child */
            execvp(argv[0], argv);
            fprintf(stderr, "%s: execvp: %s\n", progname, strerror(errno));
            _exit(EXIT_FAILURE);
        } else {                        /* parent */
            if (waitpid(pid, &status, 0) == -1) {
                fprintf(stderr, "%s: waitpid: %s\n", progname, strerror(errno));
            }
        }
    }

    return EXIT_SUCCESS;
}
```

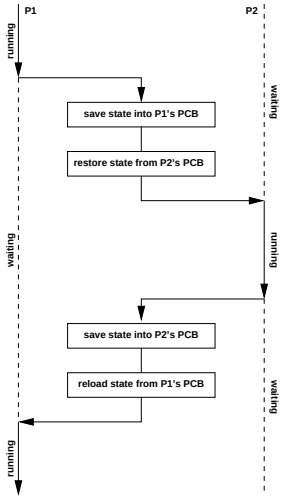
Listing 8: Minimal command interpreter (shell) (C)

```rust
//! An implementation of a very minimal shell

use anyhow::Error;
use is_terminal::IsTerminal;
use std::io::{self, Write};
use std::process;

fn is_interactive() -> bool {
    std::io::stdin().is_terminal()
}

fn show_error(error: Error) {
    eprintln!("msh: {error}");
    if !is_interactive() {
        std::process::exit(1);
    }
}

fn show_prompt() {
    if is_interactive() {
        print!("msh > ");
        io::stdout().flush().expect("msh: could not flush stdout");
    }
}

fn run(line: &str) {
    let mut argv: Vec<&str> = line.split_whitespace().collect();
    if !argv.is_empty() {
        let prog = argv.remove(0);
        if let "exit" = prog {
            std::process::exit(0);
        } else {
            let mut cmd = process::Command::new(prog);
            match cmd.args(argv).spawn() {
                Ok(mut child) => {
                    child.wait().expect("msh: child wasn't running");
                }
                Err(error) => show_error(error.into()),
            }
        }
    }
}

fn main() {
    loop {
        let mut input = String::new();
        show_prompt();
        match io::stdin().read_line(&mut input) {
            Ok(0) => {
                if is_interactive() {
                    println!();
                }
                break;
            }
            Ok(_) => run(input.trim()),
            Err(error) => show_error(error.into()),
        }
    }
}
```

Listing 9: Minimal command interpreter (shell) (Rust)

Context Switch

- Save the state of the running process/thread
- Reload the state of the next running process/thread
- Context switch overhead is an important operating system performance metric
- Switching processes can be expensive if memory must be reloaded
- Preferable to continue a process or thread on the same CPU



A context switch is the process of storing the state of a process or thread, so that it can be restored and resume execution at a later point. Context switches happen frequently and hence it is important that they can be carried out with low overhead.

A system call may be seen as a context switch as well where a user-space program does a context switch into the operating system kernel. However, most of the time, when we talk about context switches we talk about context switches between user space processes or threads.

Section 9: Threads

Threads

- Threads are individual control flows, typically within a process (or within a kernel)
- Every thread has its own private stack (so that function calls can be managed for each thread separately)
- Multiple threads share the same address space and other resources
  - Fast communication between threads
  - Fast context switching between threads
  - Often used for very scalable server programs
  - Multiple CPUs can be used by a single process
  - Threads require synchronization (see later)
- Some operating systems provide thread support in the kernel while others implement threads in user space

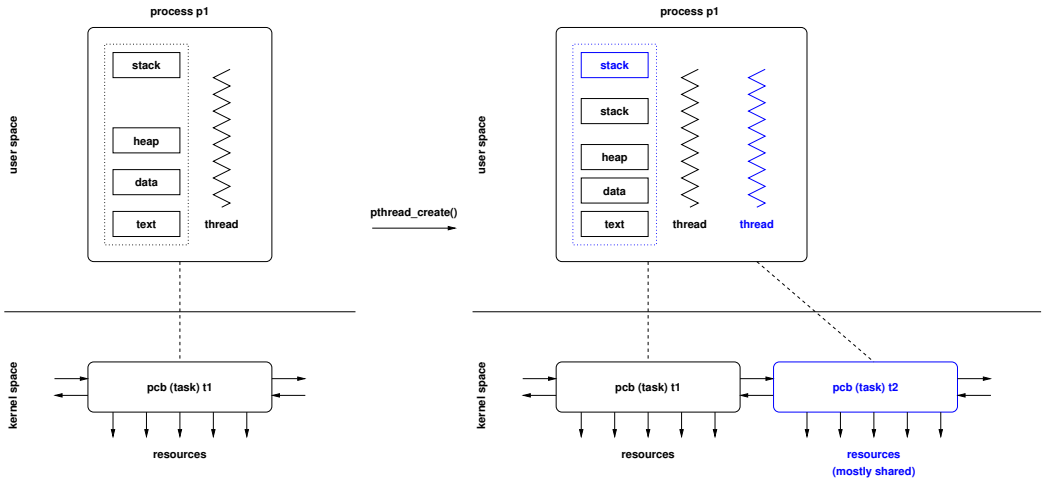
A thread is the smallest sequence of programmed instructions that can be managed independently (by the operating system kernel). A process has a single thread of control executing a sequence of machine instructions.

Threads extend this basic model by enabling processes with multiple threads of control. Note that the execution of threads is concurrent and hence the execution order is in general non-deterministic. Never make any assumption about thread execution order. On systems with multiple processor cores, threads within a process may execute concurrently at the hardware level.

While it is great to be able to utilize multiple cores, writing and debugging concurrent programs is difficult. As Ned Batchelder once stated this nicely:

> Some people, when confronted with a problem, think, “I know, I’ll use threads,” and then two they hav erpoblesms.

Thread Creation: pthread_create



This slide aims to explain the effect of the `pthread_create()` call. The following happens if a process with its own memory image and thread of control invokes the `pthread_create()` call (which usually is a wrapper around a system specific system call):

1. new process control block is allocated in the kernel and linked to the other process control blocks.
2. A new unused process identifier is assigned to the new process control block.
3. A new stack segment is added to the memory image of the calling process.
4. Most of the resources assigned to the original process (thread) are shared with the new thread.
5. The new thread is initialized to start executing at the function provided by the `pthread_create()` call.

Note that each thread of control needs its own stack since each thread of control does independent function calls.

Furthermore, note that all data in the memory image is accessible by all threads. This enables super fast exchange of data between threads but this can also lead to subtle race conditions.

POSIX API (pthreads)

```
#include <pthread.h>

typedef ... pthread_t;
typedef ... pthread_attr_t;

int pthread_create(pthread_t *thread, pthread_attr_t *attr,
                   void * (*start) (void *), void *arg);
void pthread_exit(void *retval);
int pthread_cancel(pthread_t thread);
int pthread_join(pthread_t thread, void **retvalp);

int pthread_cleanup_push(void (*func)(void *), void *arg)
int pthread_cleanup_pop(int execute)
```


Listing 10 demonstrates how threads are created and joined using the POSIX thread API for the C programming language. Listing 11 shows the same program written in the Rust programming language (with minimal error handling).

```c
/*
 * echo/echo-pthread.c --
 *
 *      A simple program to start and join threads.
 */

#define _POSIX_C_SOURCE 200809L

#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <pthread.h>

static void* work(void *data)
{
    char *msg = (char *) data;
    (void) printf("%s ", msg);
    return NULL;
}

int main(int argc, char *argv[])
{
    int rc, status = EXIT_SUCCESS;
    pthread_t tids[argc];

    for (int i = 1; i < argc; i++) {
        rc = pthread_create(&tids[i], NULL, work, argv[i]);
        if (rc) {
            fprintf(stderr, "pthread_create() failed: %s\n", strerror(rc));
            status = EXIT_FAILURE;
            memset(&tids[i], 0, sizeof(tids[i]));
        }
    }

    for (int i = 1; i < argc; i++) {
        if (tids[i]) {
            rc = pthread_join(tids[i], NULL);
            if (rc) {
                fprintf(stderr, "pthread_join() failed: %s\n", strerror(rc));
                status = EXIT_FAILURE;
            }
        }
    }

    (void) puts("");
    if (fflush(stdout) || ferror(stdout)) {
        perror("write failed");
        status = EXIT_FAILURE;
    }

    return status;
}
```

Listing 10: Creating threads and joining them (C)

```rust
//! A multi-threaded program to echo command line arguments.

use std::env;
use std::thread;

fn work(arg: String) {
    print!("{arg} ");
}

fn main() {
    let mut vec = Vec::new();
    let mut it = env::args().skip(1).peekable();

    if it.peek().is_none() {
        return;
    }

    for arg in it {
        let handle = thread::spawn(move || {
            work(arg);
        });
        vec.push(handle);
    }

    for handle in vec {
        handle.join().unwrap();
    }
    println!();
}
```

Listing 11: Creating threads and joining them (Rust)

Processes and Threads in the Linux Kernel

- Linux internally treats processes and threads as so called tasks
- Linux distinguishes three different types of tasks:
  1. idle tasks (also called idle threads)
  2. kernel tasks (also called kernel threads)
  3. user tasks
- Tasks are in one of the states *running*, *interruptible*, *uninterruptible*, *stopped*, *zombie*, or *dead*
- A special `clone()` system call is used to create processes and threads

Processes and Threads in the Linux Kernel

- Linux tasks (processes) are represented by a `struct task_struct` defined in `<linux/sched.h>`
- Tasks are organized in a circular, doubly-linked list with an additional hashtable, hashed by process id (pid)
- Non-modifying access to the task list requires the usage of the `tasklist_lock` for `READ`
- Modifying access to the task list requires the usage the `tasklist_lock` for `WRITE`
- System calls are identified by a number
- The `sys_call_table` contains pointers to functions implementing the system calls

The kernel module shown in Listing 12 iterates over all tasks in the Linux kernel. Note that the Linux kernel has a collection of macros and functions implementing commonly used data structures.

```c
/*
 * This is a simple tasks list iteration demo.
 */

#include <linux/kernel.h>
#include <linux/module.h>
#include <linux/sched.h>
#include <linux/sched/task.h>
#include <linux/sched/signal.h>

MODULE_AUTHOR("Juergen Schoenwaelder");
MODULE_LICENSE("Dual BSD/GPL");
MODULE_DESCRIPTION("Simple task list iteration kernel module.");

static const char* modname = __this_module.name;

static void print_tasks(void)
{
    struct task_struct *task;

    rcu_read_lock();
    for_each_process(task) {
        printk(KERN_INFO "%s: %8d %s\n", modname, task->pid, task->comm);
    }
    rcu_read_unlock();
}

static int __init tasks_init(void)
{
    printk(KERN_DEBUG "%s: initializing...\n", modname);
    print_tasks();
    return 0;
}

static void __exit tasks_exit(void)
{
    printk(KERN_DEBUG "%s: exiting...\n", modname);
}

module_init(tasks_init);
module_exit(tasks_exit);
```

Listing 12: Iterating over all tasks in the kernel (Linux kernel module)

Part IV

Synchronization

Concurrent threads or processes require synchronization in order to coordinate access to shared resources. The general idea is that the multiple processes or threads handshake at a certain point in their execution, in order to reach an agreement or commit to a certain sequence of action. Synchronization is in particular a major concern with threads or processes that share memory since concurrent access to memory must be coordinated.

When we talk about synchronization, we usually refer to two different problems. The first problem, the mutual exclusion problem, is about ensuring that critical sections of a program's code are only executed by one concurrent thread at a time. The second problem, the coordination problem, is about ensuring that threads coordinate their actions by waiting for each other when this is necessary.

By the end of this part, students should be able to

- understand race conditions and the notion of critical sections;
- describe solutions for classic synchronization problems;
- explain limitations of ad-hoc mechanisms to solve synchronization problems;
- understand the semantics of semaphores;
- solve synchronization problems using semaphores;
- recall the definition of mutexes and condition variables;
- solve synchronization problems using mutexes and condition variables;
- outline how message passing can be used to solve synchronization problems;
- explain synchronization pattern;
- explain how to solve synchronization pattern with semaphores;
- illustrate how to solve synchronization pattern with mutexes and condition variables;
- use the POSIX APIs for thread synchronization (mutexes, condition variables, barriers).

Section 10: Race Conditions and Critical Sections

Race Conditions

Definition (race condition)

A *race condition* is a situation where the result produced by concurrent processes (or threads) accessing and manipulating shared resources (e.g., shared variables) depends unexpectedly on the order of the execution of the processes (or threads).

- Protection against race conditions is a very important issue within operating system kernels, but equally important in many application programs
- Protection against race conditions is difficult to test (execution order usually depends on many factors that are hard to control)
- High-level programming constructs move the generation of correct low-level race protection into the compiler

In this section, we focus on race conditions related to shared variables or more generally data stored in shared memory. Race conditions, however, are not limited to data stored in shared memory. Another classic example are time-of-check to time-of-use race conditions (TOCTOU, TOCTTOU or TOC/TOU) in file systems where a check is made at some point in time to determine whether a certain action should be performed at some later point in time. An adversary can exploit the time between the check and the use to change the file system in order to gain an advantage.

```
1  import os
2  import time
3
4  def collect(dir, age):
5      """Collect all files in dir that are older than age days."""
6      obsoletes = []
7      for root, _, files in os.walk(dir):
8          for name in files:
9              fn = os.path.join(root, name)
10             if os.path.getmtime(fn) < time.time() - age * 86400:
11                 obsoletes.append(fn)
12     return obsoletes
13
14 def delete(obsoletes):
15     """Unlink all files listed in obsoletes."""
16     for fn in obsoletes:
17         os.unlink(fn)
18
19 delete(collect("/tmp", 1))
```

The Python script appears to be harmless but it has a race condition. An adversary can exploit the race condition between collecting file names and deleting them by creating a file `/tmp/foo/passwd` with an old modification time and then between the time of check and the time of use delete `/tmp/foo` and create a symbolic link `/tmp/foo` pointing to `/etc`. The unlink of `/tmp/foo/passwd` will then unlink `/etc/passwd`.

Bounded-Buffer Problem (incorrect naive solution)

```
const int N;
shared item_t buffer[N];
shared int in = 0, out = 0, count = 0;

void producer()
{
    produce(&item);
    while (count == N) sleep(1);
    buffer[in] = item;
    in = (in + 1) % N;
    count = count + 1;
}
```

```
void consumer() {
{
     while (count == 0) sleep(1);
     item = buffer[out];
     out = (out + 1) % N;
     count = count - 1;
     consume(item);
 }
```


The bounded-buffer problem, sometimes also called the producer-consumer problem, is one of the classic synchronization problems. Unlike some other classic problems, it is of high practical relevance since bounded-buffers are frequently used to separate processes that execute at different speeds. Communication channels between processes that can store a limited number of messages are also bounded-buffers.

The basic idea is the following:

- Producer processes put data into a common fixed-size buffer while consumer processes read data out of the buffer.
- Producers must wait if the buffer is full, consumers must wait if the buffer is empty.
- The buffer is organized in a circular fashion. The `in` and `out` indexes indicate where the next item is inserted or removed.

A general design goal is to avoid busy waiting loops. In an ideal solution, a process that can't proceed would enter a dormant state and be woken up when the process can proceed again. The (incorrect) solution on the slide uses sleep cycles to avoid busy waiting but they still waste CPU cycles and they delay reaction in case a process can proceed again.

Bounded-Buffer Problem (race condition)

- Pseudo machine code for `count = count + 1` and `count = count - 1`:

```
P1: load   reg_a,count          C1: load   reg_b,count
P2: incr   reg_a                C2: decr   reg_b
P3: store  reg_a,count          C3: store  reg_b,count
```

- Lets assume `count` has the value 5. What happens to `count` in the following execution sequences?
  a) `P1, P2, P3, C1, C2, C3` leads to the value 5
  b) `P1, P2, C1, C2, P3, C3` leads to the value 4
  c) `P1, P2, C1, C2, C3, P3` leads to the value 6
- Every situation, in which multiple processes (threads) manipulate shared resources, can lead to race conditions

With threads sharing memory, race conditions are created by programmers easily without getting noticed, leading to spurious failures of programs that are often difficult to reproduce and debug.

Race conditions may also be caused by the use of certain library functions. A particular problem are library functions that are not reentrant. A function is called reentrant if multiple invocations can safely run concurrently. A classic example is the `strtok()` function, which keeps internal state between function calls. For some of these library functions, there are reentrant replacements. This is usually described in the manual pages.

Listing 13 shows a C program that demonstrates that data races are a real problem and not something exotic happening only very rarely.

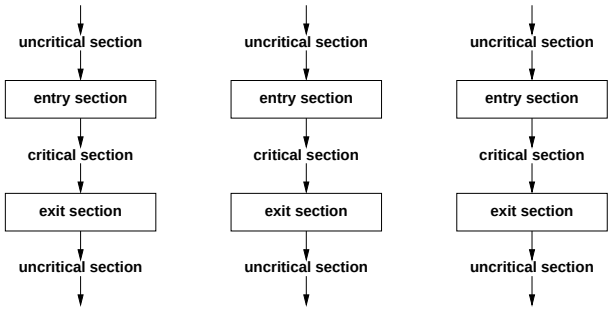
```c
/*
 * race/race.c --
 *
 *      A simple program demonstrating race conditions. Note that it
 *      is system specific how frequently race conditions occur.  Run
 *      this program using
 *
 *          watch -n 0.5 -d "./race | sort -n | xargs -n 20"
 *
 *      and lean back and you may see numbers suddenly changing.
 */

#define _POSIX_C_SOURCE 200809L

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <pthread.h>

static unsigned int cnt = 0;              /* shared variable */
static const struct timespec delay = { .tv_sec = 0, .tv_nsec = 123456 };

static void* count(void *ignored)
{
    (void) ignored;

    for (int i = 0; i < 10; i++) {
        nanosleep(&delay, NULL);
        printf("%d\n", ++cnt);
    }
    return NULL;
}

int main(void)
{
    const unsigned int num = 10;
    unsigned int i;
    int rc, status = EXIT_SUCCESS;
    pthread_t tids[num];

    for (i = 0; i < num; i++) {
        rc = pthread_create(&tids[i], NULL, count, NULL);
        if (rc) {
            fprintf(stderr, "pthread_create() failed: %s\n", strerror(rc));
            status = EXIT_FAILURE;
            memset(&tids[i], 0, sizeof(tids[i]));
        }
    }

    for (i = 0; i < num; i++) {
        rc = pthread_join(tids[i], NULL);
        if (rc) {
            fprintf(stderr, "pthread_join() failed: %s\n", strerror(rc));
            status = EXIT_FAILURE;
        }
    }
    return status;
}
```

Listing 13: Data race conditions in multi-threaded programm (C)

Critical Sections

Definition (critical section)

A *critical section* is a code segment that can only be executed by one process at a time. The execution of critical sections by multiple processes is *mutually exclusive*.



Critical-Section Problem

- Entry and exit sections must protect critical sections
- The *critical-section problem* is to design a protocol that the processes can use to cooperate
- A solution must satisfy the following requirements:
  1. *Mutual Exclusion*: No two processes may be simultaneously inside the same critical section.
  2. *Progress*: No process outside its critical sections may block other processes.
  3. *Bounded-Waiting*: No process should have to wait forever to enter its critical section.
- General solutions are not allowed to make assumptions about execution speeds or the number of CPUs present in a system.

Section 11: Synchronization Mechanisms

Disabling Interrupts

```
disable_interrupts();
critical_section();
enable_interrupts();
```

- The simplest solution is to disable all interrupts during the critical section
- Nothing can interrupt the execution of the critical section
- Can usually not be used in user-space
- Problematic on systems with multiple processors or cores
- Not usable if interrupts are needed in the critical section
- Very efficient and sometimes used in some special cases

Strict Alternation

```
/* process 0 */
uncritical_section();
while (turn != 0) sleep(1);
criticial_section();
turn = 1;
uncritical_section();
```

```
/* process 1 */
uncritical_section();
while (turn != 1) sleep(1);
critical_section()
turn = 0;
uncritical_section();
```

- Two processes share a variable called `turn`, which holds the values 0 and 1
- Strict alternation ensures mutual exclusion
- Can be extended to handle alternation between N processes
- Fails to satisfy the progress requirement, solution enforces strict alternation

Peterson's Algorithm

```
uncritical_section();
interested[i] = true;
turn = j;
while (interested[j] && turn == j) sleep(1);
criticial_section();
interested[i] = false;
uncritical_section();
```

- Two processes *i* and *j* share a variable `turn` (which holds a process identifier) and a boolean array `interested`, indexed by process identifiers
- Modifications of `turn` (and `interested`) are protected by a loop to handle concurrency issues

Peterson's algorithm satisfies mutual exclusion, progress and bounded-waiting requirements and it can be extended to handle N processes. It is, however, difficult to implement, in particular on dynamic systems, where processes can join and leave dynamically.

Spin-Locks

```
enter:  tsl     register, flag  ; copy shared variable flag to
                                ; register and set flag to 1
        cmp     register, #0    ; was flag 0?
        jnz     enter           ; if not 0, lock was set, try again
        ret                     ; critical region entered

leave:  move    flag, #0        ; clear lock by storing 0 in flag
        ret                     ; critical region left
```

- *Spin-locks* cause the processor to spin while waiting for the lock (busy waiting)
- They are sometimes used to synchronize shared-memory multi-processor cores
- They require atomic test-and-set machine instructions on shared memory cells

Note: Reentrant locks do not harm if you already hold a lock.

Spin-Locks Critique

- Busy waiting wastes processor cycles
- Busy waiting can lead to *priority inversion*:
  - Consider processes with high and low priority
  - Processes with high priority are preferred over processes with lower priority by the scheduler
  - Once a low priority process enters a critical section, processes with high priority will be slowed down more or less to the low priority
  - Depending on the scheduler, complete starvation is possible
- Goal: Find alternatives that do not require busy waiting

Section 12: Semaphores

Semaphores

Definition (semaphore)

A *semaphore* is a protected integer variable, which can only be manipulated by the *atomic* operations `up()` and `down()`:

```
down(s) {
    s = s - 1;
    if (s < 0) queue_this_process_and_block();
}

up(s) {
    s = s + 1;
    if (s <= 0) dequeue_and_wakeup_a_process();
}
```


Semaphores were introduced by Edsger Dijkstra in the 1960s [10]. Dijkstra called the operations `P()` (passeer, proberen) and `V()` (vrijgeven verhoogen), other popular names are `wait()` and `signal()`. The term semaphore goes back to the signaling system used by railroads for trains: semaphores were used to indicate whether a certain segment of a railroad is currently used by a train or not. Nowadays, signaling is done electronically and good old semaphores cease to exist.

The semaphore operations `up()` and `down()` must be atomic. On uniprocessor machines, semaphores can be implemented by either disabling interrupts during the `up()` and `down()` operations or by using a correct software solution (e.g., Peterson's algorithm).

On multiprocessor machines, semaphores are usually implemented by using spin-locks, which themself use special machine instructions that cause concurrent CPUs and cores to synchronize. Semaphores are therefore often implemented on top of more primitive synchronization mechanisms.

Semaphores are the classic approach to achieve mutual exclusion and to solve coordination problems. They are a popular tool to think about synchronization problems and to describe algorithms, they work great for assignments, exams, or job interviews. They are, however, less popular for implementing programs that exploit system-level concurrency. For writing production code, mutexes and condition variables tend to be more popular. This may be partially due to the design and popularity of the POSIX thread API.

Allen B. Downey's "Little Book of Semaphores" [13] is a great collection of synchronization problems. The book also describes solutions using semaphores, including wrong solutions, which are often very informative.

Critical Sections with Semaphores

```
semaphore mutex = 1;

uncritical_section();          uncritical_section();
down(&mutex);                  down(&mutex);
critical_section();            critical_section();
up(&mutex);                    up(&mutex);
uncritical_section();          uncritical_section();
```

- Rule of thumb: Every access to a shared data object must be protected by a `mutex` semaphore for the shared data object as shown above
- However, some synchronization and coordination problems require more creative usage of semaphores

Bounded-Buffer with Semaphores

```
const int N; shared int in = 0, out = 0, count = 0;
shared item_t buffer[N]; semaphore mutex = 1, empty = N, full = 0;

void producer()
{
    produce(&item);
    down(&empty);
    down(&mutex);
    buffer[in] = item;
    in = (in + 1) % N;
    up(&mutex);
    up(&full);
}
```

```
void consumer()
{
   down(&full);
   down(&mutex);
   item = buffer[out];
   out = (out + 1) % N;
   up(&mutex);
   up(&empty);
   consume(item);
 }
```


The semaphores serve different purposes:

- The semaphore `mutex` protects the critical section.
- The semaphore `empty` counts empty buffer space.
- The semaphore `full` counts used buffer space.

Readers / Writers Problem

- A data object is to be shared among several concurrent processes
- Multiple processes (the readers) should be able to read the shared data object simultaneously
- Processes that modify the shared data object (the writers) may only do so if no other process (reader or writer) accesses the shared data object
- Several variations exist, mainly distinguishing whether either reader or writers gain preferred access

$\Longrightarrow$ Starvation can occur in many solutions and is not taken into account here

The readers-writers problem is another classic synchronization problem of high practical relevance. Consider a kernel thread traversing the task list maintained in the kernel. There is no problem if other kernel threads do the same concurrently as long as no thread is making changes to the task list.

Readers / Writers with Semaphores

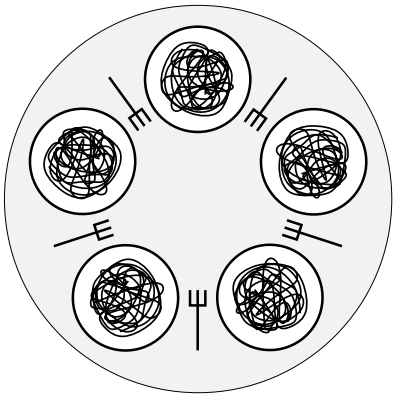
```
shared object data; shared int readcount = 0;
semaphore mutex = 1, writer = 1;

void reader()
{
    down(&mutex);
    if (++readcount == 1) down(&writer);
    up(&mutex);
    read_shared_object(&data);
    down(&mutex);
    if (--readcount == 0) up(&writer);
    up(&mutex);
}
```

```
void writer()
{
    down(&writer);
    write_shared_object(&data);
    up(&writer);
}
```


Many readers can cause starvation of writers. Finding solutions for the readers-writers problem that are starvation free may be a good weekend exercise. For more details, see [8].

Dining Philosophers

- Philosophers sitting on a round table either think or eat
- Philosophers do not keep forks while thinking
- A philosopher needs two forks (left and right) to eat
- A philosopher can only pick up one fork at a time



There are different solutions for the dining philosophers problem.

- Resource hierarchy solution

  The solution assumes that you can establish an order on the forks. Philosophers then acquire locks for the forks always with lowest number first. This means that in case N-1 philosophers pick up a fork, the Nth philosopher has to wait. This solution was proposed by Dijkstra and it does not require any centralized component.

- Arbitrator solution

  An arbitrator is introduced who is coordinating access to forks. A philosopher can only pick up forks if the arbitrator allows to do so. The arbitrator may be implemented using a mutual exclusion semaphore. The solution on the following slide implements the arbitrator solution.

- Request messages solution

  This solution assumes that philosophers can send messages to each other. A philosopher not able to pick up a needed fork will send a request message to the philosopher holding the fork. A philosopher receiving a message while holding a fork will finish eating, then clean the fork and send it over to the philosopher who requested the fork.

Dining Philosophers with Semaphores

```
const int N;              /* number of philosophers  */
shared int state[N];      /* thinking (default), hungry or eating */
semaphore mutex = 1;      /* mutex semaphore to protect state */
semaphore s[N] = 0;       /* semaphore for each philosopher */

void philosopher(int i)
{
    while (true) {
        think(i);
        take_forks(i);
        eat(i);
        put_forks(i);
    }
}
```

```
void test(int i)
{
    if (state[i] == hungry
        && state[(i-1)%N] != eating
        && state[(i+1)%N] != eating) {
        state[i] = eating;
        up(&s[i]);
    }
}
```


The `test()` function tests whether philosopher `i` can eat and conditionally unblocks his semaphore.

Dining Philosophers with Semaphores

```
void take_forks(int i)
{
    down(&mutex);
    state[i] = hungry;
    test(i);
    up(&mutex);
    down(&s[i]);
}
```

```
void put_forks(int i)
{
    down(&mutex);
    state[i] = thinking;
    test((i-1)%N);
    test((i+1)%N);
    up(&mutex);
}
```


The function `take_forks()` introduces a hungry state and waits for the philosopher's semaphore. The function `put_forks()` gives the neighbors a chance to eat.

Binary Semaphores

- Binary semaphores are semaphores that only take the two values 0 and 1.
- Counting semaphores can be implemented by means of binary semaphores:

```
    shared int c;
    binary_semaphore mutex = 1, wait = 0, barrier = 1;

void down()                          void up()
{                                    {
    down(&barrier);                      down(&mutex);
    down(&mutex);                        c = c + 1;
    c = c - 1;                           if (c <= 0) {
    if (c < 0) {                             up(&wait);
        up(&mutex);                      }
        down(&wait);                     up(&mutex);
    } else {                         }
        up(&mutex);
    }
    up(&barrier);
}
```


Apparently, binary semaphores, sometimes also called locks, are sufficient to implement counting semaphores.

However, a warning needs to be placed here that implementing synchronization primitives is subtle on today's computing systems. The reason is that the execution of instructions can be reordered by optimizing compilers and CPUs since modern memory systems may process read and write operations in non-sequential order. For a discussion of problems caused by so called memory models of modern hardware and programming languages, see [4].

Section 13: Critical Regions, Condition Variables, Messages

Critical Regions

```
shared struct buffer {
    item_t pool[N]; int count; int in; int out;
}

region buffer when (count < N)
{
    pool[in] = item;
    in = (in + 1) % N;
    count = count + 1;
}
```

```
region buffer when (count > 0)
{
    item = pool[out];
    out = (out + 1) % N;
    count = count - 1;
}
```

- Simple programming errors (omissions, permutations) with semaphores usually lead to difficult to debug synchronization errors
- By introducing language constructs, the number of errors can be reduced

Monitors

- Idea: Encapsulate the shared data object and the synchronization access methods into a monitor
- Processes can call the procedures provided by the monitor
- Processes can not access monitor internal data directly
- A monitor ensures that only one process is active in the monitor at every given point in time
- Monitors are special programming language constructs
- Compilers generate proper synchronization code
- Monitors were developed well before object-oriented languages became popular

Monitors were introduced by C.A.R. Hoare [16] in the 1970s.

Condition Variables

- Condition variables are special monitor variables that can be used to solve more complex coordination and synchronization problems
- Condition variables support the two operations `wait()` and `signal()`:
  - The `wait()` operation blocks the calling process on the condition variable `c` until another process invokes `signal()` on `c`. Another process may enter the monitor while a process is waiting to be signaled.
  - The `signal()` operation unblocks a process waiting on the condition variable `c`. The calling process must leave the monitor before the signaled process continues.
- Condition variables are not counters. A `signal()` on `c` is ignored if no processes is waiting on `c`

Condition variables come with different semantics. Consider a thread in a critical region waiting on a condition variable. While waiting, the thread will leave the critical region and it will enter the critical region again before returning to continue.

The *Hoare semantics of condition variables* [16] assume that the signaling thread causes the woken thread to run immediately upon being woken up. The signal is thus a guarantee that the world has changed as desired and the signaled thread can rely on the fact that the world is in the desired state. Implementation experience with this approach then later let to relaxed semantics.

The *Mesa semantics of condition variables* [17] assume that a woken thread is not necessarily running immediately, it is instead scheduled regularly like any other threads are scheduled. Hence, when the thread returns from the wait call, there is a chance that the condition is not true anymore since another thread might have changed the state of the world inbetween. In other words, the signal is not a guarantee that the condition is as desired, it is merely a hint that the state of the world has changed and the woken thread needs to verify that the condition indeed holds.

Today's systems usually use the weaker Mesa semantics for condition variables and this requires that programmers check a condition and wait in a loop. The general pattern is (in pseudo code notation):

```
lock(&mutex);
while (! condition) {
    wait(&condvar, &mutex);
}
unlock(&mutex);
```

With the stronger Hoare semantics, a loop would not be necessary. In practice, you *always* check and wait for a condition using a loop.

Bounded-Buffer with Monitors

```
monitor BoundedBuffer
{
  condition full, empty;
  int count = 0;
  item_t buffer[N];

  void enter(item_t item)
  {
      if (count == N) wait(&full);
      buffer[in] = item;
      in = (in + 1) % N;
      count = count + 1;
      if (count == 1) signal(&empty);
  }

  item_t remove()
  {
      if (count == 0) wait(&empty);
      item = buffer[out];
      out = (out + 1) % N;
      count = count - 1;
      if (count == N-1) signal(&full);
      return item;
  }
}
```

Messages

- Exchange of messages can be used for synchronization
- Two primitive operations:

  ```
  send(destination, message)
  recv(source, message)
  ```

- Blocking message systems block processes in these primitives if the peer is not ready for a rendevous
- Storing message systems maintain messages in special mailboxes called message queues. Processes only block if the remote mailbox is full during a `send()` or the local mailbox is empty during a `recv()`
- Some programming languages (e.g., go) use message queues as the primary abstraction for synchronization (e.g., go routines and channels)

Bounded-Buffer with Messages

```
void init() { for (i = 0; i < N; i++) { send(&producer, &m); } }

void producer()
{
    produce(&item);
    recv(&consumer, &m);
    pack(&m, item);
    send(&consumer, &m)
}
```

```
void consumer()
{
    recv(&producer, &m);
    unpack(&m, &item)
    send(&producer, &m);
    consume(item);
}
```

- Messages are used as tokens which control the exchange of items
- Consumers initially generate and send a number of tokens to the producers
- Mailboxes are used as temporary storage space and must be large enough to hold all tokens / messages

Equivalence of Mechanisms

- Are there synchronization problems which can be solved only with a subset of the mechanisms?
- Or are all the mechanisms equivalent?
- Constructive proof technique:
  - Two mechanisms A and B are equivalent if A can emulate B and B can emulate A
  - In both proof directions, construct an emulation (does not have to be efficient - just correct ;-)

Section 14: Synchronization Pattern

Semaphore Pattern: Mutual Exclusion

A critical section may only be executed by a single thread.

```
semaphore_t s = 1;

thread()
{
  /* do something */
  down(&s);
  /* critical section */
  up(&s);
  /* do something */
}
```

Semaphore Pattern: Multiplex

A section may be executed concurrently with a certain fixed limit of N concurrent threads. (This is a generalization of the mutual exclusion pattern, which is essentially multiplex with $N = 1$.)

```
semaphore_t s = N;

thread()
{
  /* do something */
  down(&s);
  /* multiplex section */
  up(&s);
  /* do something */
}
```

Semaphore Pattern: Signaling

A thread waits until some other thread signals a certain condition.

```
semaphore_t s = 0;

waiting_thread()
{
  /* do something */
  down(&s);
  /* do something */
}
```

```
signaling_thread()
{
  /* do something */
  up(&s);
  /* do something */
}
```

Semaphore Pattern: Rendezvous

Two threads wait until they both have reached a certain state (the rendezvous point) and afterwards they proceed independently again. (This can be seen as using the signaling pattern twice.)

```
semaphore_t s1 = 0, s2 = 0;

thread_A()                    thread_B()
{                             {
  /* do something */            /* do something */
  up(&s2);                      up(&s1);
  down(&s1);                    down(&s2);
  /* do something */            /* do something */
}
```

Semaphore Pattern: Simple Barrier

A barrier requires that all threads reach the barrier before they can proceed.
(Generalization of the rendevous pattern to N threads.)

```
shared int count = 0;
semaphore_t mutex = 1, turnstile = 0;

thread()
{
  /* do something */
  down(&mutex);
  count++;
  if (count == N) {
    for (int j = 0; j < N; j++) {
      up(&turnstile);          /* let N threads pass through the turnstile */
    }
    count = 0;
  }
  up(&mutex);
  down(&turnstile);            /* block until opened by the Nth thread */
  /* do something */
}
```

Semaphore Pattern: Double Barrier

This solution allows threads to do something while passing through the barrier, which is sometimes needed.

```
shared int count = 0;
semaphore_t mutex = 1, turnstile1 = 0, turnstile2 = 1;

{
  /* do something */

  down(&mutex);
  count++;
  if (count == N) {
    down(&turnstile2);        /* close turnstile2 (which was left open) */
    up(&turnstile1);          /* open turnstile1 for one thread */
  }
  up(&mutex);
  down(&turnstile1);          /* block until opened by the last thread */
  up(&turnstile1);            /* every thread lets another thread pass */

  /* do something controlled by a barrier */
```

Semaphore Pattern: Double Barrier (cont.)

```
  /* do something controlled by a barrier */

  down(&mutex);
  count--;
  if (count == 0) {
    down(&turnstile1);        /* close turnstile1 again */
    up(&turnstile2);          /* open turnstile2 for one thread */
  }
  up(&mutex);
  down(&turnstile2);          /* block until opened by the last thread */
  up(&turnstile2);            /* every thread lets another thread pass */
                              /* (turnstile2 is left open) */

  /* do something */
}
```

Section 15: Synchronization in C

POSIX Mutex Locks

```
#include <pthread.h>

typedef ... pthread_mutex_t;
typedef ... pthread_mutexattr_t;

int pthread_mutex_init(pthread_mutex_t *mutex, pthread_mutexattr_t *mutexattr);
int pthread_mutex_destroy(pthread_mutex_t *mutex);

int pthread_mutex_lock(pthread_mutex_t *mutex);
int pthread_mutex_trylock(pthread_mutex_t *mutex);
int pthread_mutex_timedlock(pthread_mutex_t *mutex, struct timespec *abstime);
int pthread_mutex_unlock(pthread_mutex_t *mutex);
```


Mutex locks are a simple mechanism to achieve mutual exclusion in critical sections. The work like binary semaphores and they are often used in conjunction with condition variables (see next page).

Listing 14 demonstrates the usage of mutex locks in C. Listing 15 shows the same program written in the Rust programming language (with minimal error handling).

There are some tools that can detect certain thread synchronization problems:

- `valgrind --tool=drd` detects data races
- `valgrind --tool=helgrind` detects misuse of the pthread API
- `clang -fsanitize=thread` instruments code to detect data races
- `mutrace` collects statistics about lock usage

```c
/*
 * pthread/pthread-mutex.c --
 *
 *      A simple demonstration of pthread mutexes. This example
 *      lacks error handling code to keep it short.
 */

#define _POSIX_C_SOURCE 200809L

#include <pthread.h>

typedef struct {
    unsigned int counter;       /* shared counter */
    pthread_mutex_t mutex;      /* mutex protecting the counter */
} counter_t;

static void* work(void *arg)
{
    counter_t *c = (counter_t *) arg;

    (void) pthread_mutex_lock(&c->mutex);
    c->counter++;
    (void) pthread_mutex_unlock(&c->mutex);
    return NULL;
}

int main(int argc, char *argv[])
{
    pthread_t tids[argc];
    counter_t cnter = { .counter = 0, .mutex = PTHREAD_MUTEX_INITIALIZER };

    (void) argv;

    for (int i = 1; i < argc; i++) {
        (void) pthread_create(&tids[i], NULL, work, &cnter);
    }
    for (int i = 1; i < argc; i++) {
        (void) pthread_join(tids[i], NULL);
    }
    return 0;
}
```

Listing 14: Demonstration of pthread mutexes (C)

```rust
/*
 * pthread/src/bin/pthread-mutex.rs --
 *
 *      A simple demonstration of mutexes in Rust. This example lacks
 *      error handling code to keep it short. In Rust, an Arc is a
 *      thread-safe reference-counting pointer (Atomically Reference
 *      Counted). Note that the unlock is implicit when the obtained
 *      lock goes out of scope.
 */

use std::env;
use std::sync::{Arc, Mutex};
use std::thread;

fn work(arc: Arc<Mutex<u64>>) {
    let mut c = arc.lock().unwrap();
    *c += 1;
}

fn main() {
    let counter = Mutex::new(0);
    let arc = Arc::new(counter);

    let mut handles = Vec::new();
    for _ in env::args().skip(1) {
        let a = arc.clone();
        handles.push(thread::spawn(move || work(a)));
    }

    for handle in handles {
        handle.join().unwrap();
    }
}
```

Listing 15: Demonstration of mutexes (Rust)

POSIX Condition Variables

```
#include <pthread.h>

typedef ... pthread_cond_t;
typedef ... pthread_condattr_t;

int pthread_cond_init(pthread_cond_t *cond, pthread_condattr_t *condattr);
int pthread_cond_destroy(pthread_cond_t *cond);

int pthread_cond_wait(pthread_cond_t *cond, pthread_mutex_t *mutex);
int pthread_cond_timedwait(pthread_cond_t *cond, pthread_mutex_t *mutex,
                           struct timespec *abstime);

int pthread_cond_signal(pthread_cond_t *cond);
int pthread_cond_broadcast(pthread_cond_t *cond);
```


Condition variables can be used to bind the entrance into a critical section protected by a mutex to a condition. The common pattern is to acquire a mutex and then to wait in a loop until a condition is true. While the condition is false, the mutex will be temporarily released in a call of `pthread_cond_wait()`.

Listing 16 demonstrates the usage of condition variables in C. Listing 17 shows the same program written in the Rust programming language (with minimal error handling).

Note that it is not easy to implement condition variables efficiently on top of semaphores [3]. Hence, it is useful to have condition variables implemented natively. Another interesting read is a paper describing fast user-level locking in Linux [14].

```c
/*
 * pthread/pthread-cond.c --
 *
 *      A simple demonstration of pthread condition variables. This
 *      example lacks error handling code to keep it short.
 */

#define _POSIX_C_SOURCE 200809L

#include <pthread.h>

typedef struct {
    unsigned int counter;       /* shared counter */
    pthread_mutex_t mutex;      /* mutex protecting the counter */
    pthread_cond_t cond;        /* condition variable */
} counter_t;

static void* even(void *arg)
{
    counter_t *c = (counter_t *) arg;
    (void) pthread_mutex_lock(&c->mutex);
    while (c->counter % 2 != 0) {
        (void) pthread_cond_wait(&c->cond, &c->mutex);
    }
    c->counter++;
    (void) pthread_mutex_unlock(&c->mutex);
    (void) pthread_cond_signal(&c->cond);
    return NULL;
}

static void* odd(void *arg)
{
    counter_t *c = (counter_t *) arg;
    (void) pthread_mutex_lock(&c->mutex);
    while (c->counter % 2 == 0) {
        (void) pthread_cond_wait(&c->cond, &c->mutex);
    }
    c->counter++;
    (void) pthread_mutex_unlock(&c->mutex);
    (void) pthread_cond_signal(&c->cond);
    return NULL;
}

int main(int argc, char *argv[])
{
    pthread_t tids[2*argc];
    counter_t cnter = { .counter = 0, .mutex = PTHREAD_MUTEX_INITIALIZER,
                        .cond = PTHREAD_COND_INITIALIZER };
    (void) argv;

    for (int i = 1; i < argc; i++) {
        (void) pthread_create(&tids[2*i], NULL, even, &cnter);
        (void) pthread_create(&tids[2*i+1], NULL, odd, &cnter);
    }
    for (int i = 1; i < argc; i++) {
        (void) pthread_join(tids[2*i], NULL);
        (void) pthread_join(tids[2*i+1], NULL);
    }
    return 0;
}
```

Listing 16: Demonstration of pthread condition variables (C)

```rust
/*
 * pthread/src/bin/pthread-cond.rs --
 *
 *      A simple demonstration of condition variables in Rust. This
 *      example lacks error handling code to keep it short.
 */

use std::env;
use std::sync::{Arc, Condvar, Mutex};
use std::thread;

struct Counter {
    mutex: Mutex<u64>,
    cvar: Condvar,
}

fn even(me: &str, arc: Arc<Counter>) {
    let mut ctr = arc
        .cvar
        .wait_while(arc.mutex.lock().unwrap(), |c| *c % 2 != 0)
        .unwrap();
    eprintln!("{me}: even: {}", *ctr);
    *ctr += 1;
    arc.cvar.notify_all();
}

fn odd(me: &str, arc: Arc<Counter>) {
    let mut ctr = arc
        .cvar
        .wait_while(arc.mutex.lock().unwrap(), |c| *c % 2 == 0)
        .unwrap();
    eprintln!("{me}: odd:  {}", *ctr);
    *ctr += 1;
    arc.cvar.notify_all();
}

fn main() {
    let ctr = Counter {
        mutex: Mutex::new(0),
        cvar: Condvar::new(),
    };
    let data = Arc::new(ctr);

    let mut handles = Vec::new();
    for arg in env::args().skip(1) {
        let c = data.clone();
        let a = arg.clone();
        handles.push(thread::spawn(move || even(&a, c)));
        let c = data.clone();
        handles.push(thread::spawn(move || odd(&arg, c)));
    }

    for handle in handles {
        handle.join().unwrap();
    }
}
```

Listing 17: Demonstration of condition variables (Rust)

POSIX Read-Write Locks

```
#include <pthread.h>

typedef ... pthread_rwlock_t;
typedef ... pthread_rwlockattr_t;

int pthread_rwlock_init(pthread_rwlock_t *restrict rwlock,
                        const pthread_rwlockattr_t *attr);
int pthread_rwlock_destroy(pthread_rwlock_t *rwlock);

int pthread_rwlock_rdlock(pthread_rwlock_t *rwlock);
int pthread_rwlock_tryrdlock(pthread_rwlock_t *rwlock);
int pthread_rwlock_timedrdlock(pthread_rwlock_t *rwlock, struct timespec *atime);
int pthread_rwlock_wrlock(pthread_rwlock_t *rwlock);
int pthread_rwlock_trywrlock(pthread_rwlock_t *rwlock);
int pthread_rwlock_timedwrlock(pthread_rwlock_t *rwlock, struct timespec *atime);
int pthread_rwlock_unlock(pthread_rwlock_t *rwlock);
```


Listing 18 demonstrates the usage of read-write locks in C.

```c
/*
 * pthread/pthread-rwlock.c --
 *
 *      A simple demonstration of pthread rwlocks. This example lacks
 *      error handling code to keep it short.
 */

#define _POSIX_C_SOURCE 200809L

#include <pthread.h>

typedef struct {
    unsigned int counter;       /* shared counter */
    pthread_rwlock_t rwlock;    /* read/write lock for the counter */
} counter_t;

static void* reader(void *arg)
{
    counter_t *c = (counter_t *) arg;

    (void) pthread_rwlock_rdlock(&c->rwlock);
    (void) (c->counter + c->counter);
    (void) pthread_rwlock_unlock(&c->rwlock);
    return NULL;
}

static void* writer(void *arg)
{
    counter_t *c = (counter_t *) arg;

    (void) pthread_rwlock_wrlock(&c->rwlock);
    c->counter++;
    (void) pthread_rwlock_unlock(&c->rwlock);
    return NULL;
}

int main(int argc, char *argv[])
{
    pthread_t tids[2*argc];
    counter_t cnter = { .counter = 0, .rwlock = PTHREAD_RWLOCK_INITIALIZER };
    (void) argv;

    for (int i = 1; i < argc; i++) {
        (void) pthread_create(&tids[2*i], NULL, reader, &cnter);
        (void) pthread_create(&tids[2*i+1], NULL, writer, &cnter);
    }
    for (int i = 1; i < argc; i++) {
        (void) pthread_join(tids[2*i], NULL);
        (void) pthread_join(tids[2*i+1], NULL);
    }
    return 0;
}
```

Listing 18: Demonstration of pthread rwlocks (C)

```rust
/*
 * pthread/src/bin/pthread-rwlock.rs --
 *
 *      A simple demonstration of read-write locks in Rust. This
 *      example lacks error handling code to keep it short.
 */

use std::env;
use std::sync::{Arc, RwLock};
use std::thread;

fn reader(me: &str, arc: Arc<RwLock<u64>>) {
    let c = arc.read().unwrap();
    eprintln!("{me}: reader: {}", *c);
}

fn writer(me: &str, arc: Arc<RwLock<u64>>) {
    let mut c = arc.write().unwrap();
    *c += 1;
    eprintln!("{me}: writer: {}", *c);
}

fn main() {
    let ctr = RwLock::new(0);
    let arc = Arc::new(ctr);

    let mut handles = Vec::new();

    for arg in env::args().skip(1) {
        let r = arc.clone();
        let a = arg.clone();
        handles.push(thread::spawn(move || reader(&a, r)));
        let w = arc.clone();
        handles.push(thread::spawn(move || writer(&arg, w)));
    }

    for handle in handles {
        handle.join().unwrap();
    }
}
```

Listing 19: Demonstration of pthread rwlocks (Rust)

POSIX Barriers

```
#include <pthread.h>

typedef ... pthread_barrier_t;
typedef ... pthread_barrierattr_t;

int pthread_barrier_init(pthread_barrier_t *barrier,
                         pthread_barrierattr_t *barrierattr,
                         unsigned count);
int pthread_barrier_destroy(pthread_barrier_t *barrier);

int pthread_barrier_wait(pthread_barrier_t *barrier);
```


Barriers block threads until the required number of threads have called `pthread_barrier_wait()`.

Listing 20 demonstrates the usage of barriers in C. Listing 21 shows the same program written in the Rust programming language (with minimal error handling).

```c
/*
 * pthread/pthread-barrier.c --
 *
 *      A simple demonstration of pthread barriers. This example lacks
 *      error handling code to keep it short.
 */

#define _POSIX_C_SOURCE 200809L

#include <pthread.h>
#include <stdio.h>

typedef struct {
    pthread_barrier_t enter;            /* gate to enter work */
    pthread_barrier_t leave;            /* gate to leave work */
} gates_t;

typedef struct {
    char *name;                         /* name of the worker */
    gates_t *gates;                     /* gates to use */
} worker_t;

static void* work(void *arg)
{
    worker_t *me = (worker_t *) arg;

    printf("%s arriving\n", me->name);
    (void) pthread_barrier_wait(&me->gates->enter);
    printf("%s working\n", me->name);
    (void) pthread_barrier_wait(&me->gates->leave);
    printf("%s leaving\n", me->name);
    return NULL;
}

int main(int argc, char *argv[])
{
    pthread_t tids[argc];
    gates_t gates;
    worker_t worker[argc];

    (void) pthread_barrier_init(&gates.enter, NULL, argc-1);
    (void) pthread_barrier_init(&gates.leave, NULL, argc-1);

    for (int i = 1; i < argc; i++) {
        worker[i].name = argv[i];
        worker[i].gates = &gates;
        (void) pthread_create(&tids[i], NULL, work, &worker[i]);
    }
    for (int i = 1; i < argc; i++) {
        (void) pthread_join(tids[i], NULL);
    }
    return 0;
}
```

Listing 20: Demonstration of pthread barriers (C)

```rust
/*
 * pthread/src/bin/pthread-barrier.rs --
 *
 *      A simple demonstration of barriers in Rust. This example lacks
 *      error handling code to keep it short.
 */

use std::env;
use std::sync::{Arc, Barrier};
use std::thread;

fn work(me: &str, opening: Arc<Barrier>, closing: Arc<Barrier>) {
    println!("arriving {}", me);
    opening.wait();
    println!("working {}", me);
    closing.wait();
    println!("leaving {}", me);
}

fn main() {
    let mut handles = Vec::new();
    let argc = env::args().len() - 1;
    let opening = Arc::new(Barrier::new(argc));
    let closing = Arc::new(Barrier::new(argc));

    for arg in env::args().skip(1) {
        let o = Arc::clone(&opening);
        let c = Arc::clone(&closing);
        handles.push(thread::spawn(move || work(&arg, o, c)));
    }

    for handle in handles {
        handle.join().unwrap();
    }
}
```

Listing 21: Demonstration of barriers (Rust)

POSIX Semaphores

```
#include <semaphore.h>

typedef ... sem_t;

int sem_init(sem_t *sem, int pshared, unsigned int value);
int sem_destroy(sem_t *sem);

int sem_wait(sem_t *sem);
int sem_trywait(sem_t *sem);
int sem_post(sem_t *sem);
int sem_getvalue(sem_t *sem, int *sval);

sem_t* sem_open(const char *name, int oflag);
sem_t* sem_open(const char *name, int oflag, mode_t mode, unsigned int value);
int int sem_close(sem_t *sem);
int sem_unlink(const char *name);
```


The POSIX API offers unnamed semaphores (created with `sem_init()`) and named semaphores (created with (`sem_open()`). Named semaphores exist in the file system and hence they are not bound to the lifetime of a process.

Listing 22 demonstrates the usage of unnamed semaphores in C.

```c
/*
 * pthread/pthread-sem.c --
 *
 *      A simple demonstration of pthread semaphores. This example
 *      lacks error handling code to keep it short.
 */

#define _POSIX_C_SOURCE 200809L

#include <pthread.h>
#include <semaphore.h>

typedef struct {
    unsigned int counter;       /* shared counter */
    sem_t semaphore;            /* mutex semaphore protecting the counter */
} counter_t;

static void* work(void *arg)
{
    counter_t *c = (counter_t *) arg;

    (void) sem_wait(&c->semaphore);
    c->counter++;
    (void) sem_post(&c->semaphore);
    return NULL;
}

int main(int argc, char *argv[])
{
    pthread_t tids[argc];
    counter_t cnter = { .counter = 0  };
    (void) argv;

    (void) sem_init(&cnter.semaphore, 0, 1);

    for (int i = 1; i < argc; i++) {
        (void) pthread_create(&tids[i], NULL, work, &cnter);
    }
    for (int i = 1; i < argc; i++) {
        (void) pthread_join(tids[i], NULL);
    }
    return 0;
}
```

Listing 22: Demonstration of pthread semaphores

POSIX Message Queues

```
#include <mqueue.h>

typedef ... mqd_t;

mqd_t mq_open(const char *name, int oflag);
mqd_t mq_open(const char *name, int oflag, mode_t mode, struct mq_attr *attr);

int mq_getattr(mqd_t mqdes, struct mq_attr *attr);
int mq_setattr(mqd_t mqdes, const struct mq_attr *newattr, struct mq_attr *oldattr)

int mq_close(mqd_t mqdes);
int mq_unlink(const char *name);
```

- Message queues can be used to exchange messages between threads and processes running on the same system efficiently

Message queues are a very important software components for building distributed applications. The POSIX message queues exist in the kernel and hence are restricted to a single system.

There are far more flexible message queues that can work efficient locally (i.e., between threads) and in a distributed applicaiton (i.e., processes running on different computers). Interested readers should lookup ZeroMQ[11] and nanomsg[12].

[11] https://zeromq.org/
[12] https://nanomsg.org/

POSIX Message Queues

```
#include <mqueue.h>

int mq_send(mqd_t mqdes, const char *msg_ptr, size_t msg_len,
            unsigned int msg_prio);
int mq_timedsend(mqd_t mqdes, const char *msg_ptr, size_t msg_len,
                 unsigned int msg_prio, const struct timespec *atimeout);

ssize_t mq_receive(mqd_t mqdes, char *msg_ptr, size_t msg_len,
                   unsigned int *msg_prio);
ssize_t mq_timedreceive(mqd_t mqdes, char *msg_ptr, size_t msg_len,
                   unsigned int *msg_prio, const struct timespec *atimeout);

int mq_notify(mqd_t mqdes, const struct sigevent *sevp);
```


Message queues notifications can be delivered in different ways, e.g., as signals or in a thread-like fashion.

Atomic Operations in the Linux Kernel

```
struct ... atomic_t;

int  atomic_read(atomic_t *v);
void atomic_set(atomic_t *v, int i);
void atomic_add(int i, atomic_t *v);
void atomic_sub(int i, atomic_t *v);
void atomic_inc(atomic_t *v);
void atomic_dec(atomic_t *v);

int atomic_add_negative(int i, atomic_t *v);
int atomic_sub_and_test(int i, atomic_t *v);
int atomic_inc_and_test(atomic_t *v)
int atomic_dec_and_test(atomic_t *v);
```


Sometimes it is sufficient to have atomic operations on integers. The `atomic_t` is essentially 24 bit wide since some processors use the remaining 8 bits of a 32 bit word for locking purposes.

Atomic Operations in the Linux Kernel

```
void set_bit(int nr, unsigned long *addr);
void clear_bit(int nr, unsigned long *addr);
void change_bit(int nr, unsigned long *addr);

int test_and_set_bit(int nr, unsigned long *addr);
int test_and_clear_bit(int nr, unsigned long *addr);
int test_and_change_bit(int nr, unsigned long *addr);
int test_bit(int nr, unsigned long *addr);
```


The kernel also provides bit operations that are not atomic (prefixed with two underscores). The bit operations are the only portable way to set bits atomically. On some processors, the non-atomic versions might be faster.

Spin Locks in the Linux Kernel

```
typedef ... spinlock_t;

void spin_lock_init(spinlock_t *l);

void spin_lock(spinlock_t *l);
void spin_unlock(spinlock_t *l);
void spin_unlock_wait(spinlock_t *l);
int  spin_trylock(spinlock_t *l)

int  spin_is_locked(spinlock_t *l);
```

Read-Write Locks in the Linux Kernel

```
typedef ... rwlock_t;

void rwlock_init(rwlock_t *rw);

void read_lock(rwlock_t *rw);
void read_unlock(rwlock_t *rw);

void write_lock(rwlock_t *rw);
void write_unlock(rwlock_t *rw);
int  write_trylock(rwlock_t *rw);

int  rwlock_is_locked(rwlock_t *rw);
```

Semaphores in the Linux Kernel

```
struct ... semaphore;

void sema_init(struct semaphore *sem, int val);
void init_MUTEX(struct semaphore *sem);
void init_MUTEX_LOCKED(struct semaphore *sem);

void down(struct semaphore *sem);
int  down_interruptible(struct semaphore *sem);
int  down_trylock(struct semaphore *sem);

void up(struct semaphore *sem);
```


Linux kernel semaphores are counting semaphores:

- `init_MUTEX(s)` equals `sema_init(s, 1)`
- `init_MUTEX_LOCKED(s)` equals `sema_init(s, 0)`

Section 16: Synchronization in Java and Go

Synchronization in Java

- Java supports mutual exclusion of code blocks by declaring them synchronized:

```
synchronized(expr) {
    // 'expr' must evaluate to an Object
}
```

- Java supports mutual exclusion of critical sections of an object by marking methods as `synchronized`, which is in fact just syntactic sugar:

```
synchronized void foo()         { /* body */ }
void foo() { synchronized(this) { /* body */ } }
```

- Additional `wait()`, `notify()` and `notifyAll()` methods can be used to coordinate critical sections

Listing 23 demonstrates how Java synchronization mechanisms can be used to implement a bounded buffer.

```java
/*
 * bounded/BoundedBuffer.java --
 *
 *      Bounded buffer (producer / consumer) problem solution with
 *      Java synchronized methods.
 */

import java.lang.Thread;

class BoundedBuffer
{
    private final int size;
    private int count = 0, out = 0, in = 0;
    private int[] buffer;

    public BoundedBuffer(int size) {
        this.in = 0;
        this.out = 0;
        this.count = 0;
        this.size = size;
        this.buffer = new int[this.size];
    }

    public synchronized void insert(int i)
    {
        try {
            while (count == size) {
                wait();
            }
            buffer[in] = i;
            in = (in + 1) % size;
            count++;
            notifyAll();    // wakeup all waiting threads
        } catch (InterruptedException e) {
            Thread.currentThread().interrupt();
        }
    }

    public synchronized int remove()
    {
        try {
            while (count == 0) {
                wait();
            }
            int r = buffer[out];
            out = (out + 1) % size;
            count--;
            notifyAll();    // wakeup all waiting threads
            return r;
        } catch (InterruptedException e) {
            Thread.currentThread().interrupt();
            return -1;
        }
    }
}
```

Listing 23: Implementation of a bounded buffer in Java

Synchronization in Go

- Light-weight "goroutines" that are mapped to an operating system level thread pool
- Channels provide message queues between goroutines
- Philosophy: Do not communicate by sharing memory; instead, share memory by communicating
- Inspired by Hoare's work on Communicating Sequential Processes (CSP)

Listing 24 demonstrates how Go synchronization mechanisms can be used to implement a bounded buffer.

```
/*
 * bounded/bounded.go --
 *
 *      Bounded buffer (producer / consumer) problem solution using
 *      go channels. Well, a go channel in fact is a bounded buffer.
 *      Anyway, this code is in analogy to the C version and it
 *      primarily serves to demonstrate how channels can be used to
 *      avoid the usage of explicit synchronization primitives.
 */

package main

import (
        "flag"
        "fmt"
)

const (
        size = 12
)

var nc = flag.Int("c", 1, "number of consumers")
var np = flag.Int("p", 1, "number of producers")
var ve = flag.Bool("v", false, "verbose output")

func producer(b chan int, g <-chan int, n chan<- bool) {
        for {
                v := <-g
                b <- v
                n <- true
        }
}

func consumer(b chan int, d chan<- int, r <-chan bool) {
        for {
                <-r
                v := <-b
                d <- v
        }
}

func generator() (<-chan int, chan<- bool) {
        s := make(chan int)
        n := make(chan bool)
        cnt := 0
        go func() {
                for {
                        cnt++
                        s <- cnt
                        <-n
                }
        }()
        return s, n
}

func discarder() (chan<- int, <-chan bool) {
        d := make(chan int)
        r := make(chan bool)
        cnt := 0
        go func() {
                for {
                        r <- true
                        v := <-d
                        cnt++
                        if *ve {
```

Part V

Deadlocks

Concurrent threads and processes can deadlock if they start to wait on each other. In this part, we study the conditions that are necessary for deadlocks to occur and we then look at different ways to deal with possible deadlocks. The first approach is deadlock prevention where resources are allocated in such a way that deadlocks can never occur. The second approach is deadlocks avoidance where resources are allocated in such a way that system states in which deadlocks can occur are avoided. The third approach is deadlock detection where resources are allocated freely but deadlocks are detected and then recovery procedures are used to resolve them. In order to discuss deadlocks and deadlock handling strategies, we will introduce resource allocation graphs and wait-for graphs. Furthermore, we will introduce algorithms to detect deadlocks or to determine whether system states are safe or unsafe.

By the end of this part, students should be able to

- recognize deadlock situations;
- describe the four necessary deadlock conditions;
- define resource allocation graphs and wait-for graphs;
- illustrate deadlock prevention strategies and why they are of limited use;
- illustrate deadlock avoidance strategies;
- execute the Banker's algorithm;
- explain deadlock detection algorithms;
- summarize distributed deadlock detection approaches.

Section 17: Deadlocks

Deadlocks

```
semaphore s1 = 1, s2 = 1;

void p1()                          void p2()
{                                  {
    down(&s1);                         down(&s2);
    down(&s2);                         down(&s1);
    critical_section();                critical_section();
    up(&s2);                           up(&s1);
    up(&s1);                           up(&s2);
}                                  }
```

- Executing the functions `p1` and `p2` concurrently can result in a deadlock when both processes have executed the first `down()` operation
- Deadlocks also occur if processes do not release semaphores/locks

The possibility for a deadlock may be easy to spot on an example as short as this. But real-world code is usually complex and it is not always easy to decide which parts of the code are executed with locks held or not. Code analysis tools can help but they are sometimes not able to construct all possible execution paths.

Deadlocks

```
class A
{
    public synchronized a1(B b)
    {
        b.b2();
    }

    public synchronized a2(B b)
    {
    }
}
```

```
class B
{
    public synchronized b1(A a)
    {
        a.a2();
    }

    public synchronized b2(A a)
    {
    }
}
```

- Deadlocks can also be created by careless use of higher-level synchronization mechanisms

This example demonstrates that higher-level synchronization mechanisms like Java's synchronized methods do not necessarily prevent the occurance of deadlocks.

Necessary Deadlock Conditions

Definition (necessary deadlock conditions)

A deadlock on a resource can arise if and only if all of the following conditions hold simultaneously:

- *Mutual exclusion*:
  Resources cannot be used simultaneously by several processes
- *Hold and wait*:
  Processes apply for a resource while holding another resource
- *No preemption*:
  Resources cannot be preempted, only the process itself can release resources
- *Circular wait*:
  A circular list of processes exists where every process waits for the release of a resource held by the next process

E.C. Coffman stated the necessarily conditions for deadlocks [7], hence they are also known as the Coffman conditions. E.C. Coffman also discussed deadlock prevention and avoidance strategies.

Section 18: Resource Allocation Graphs

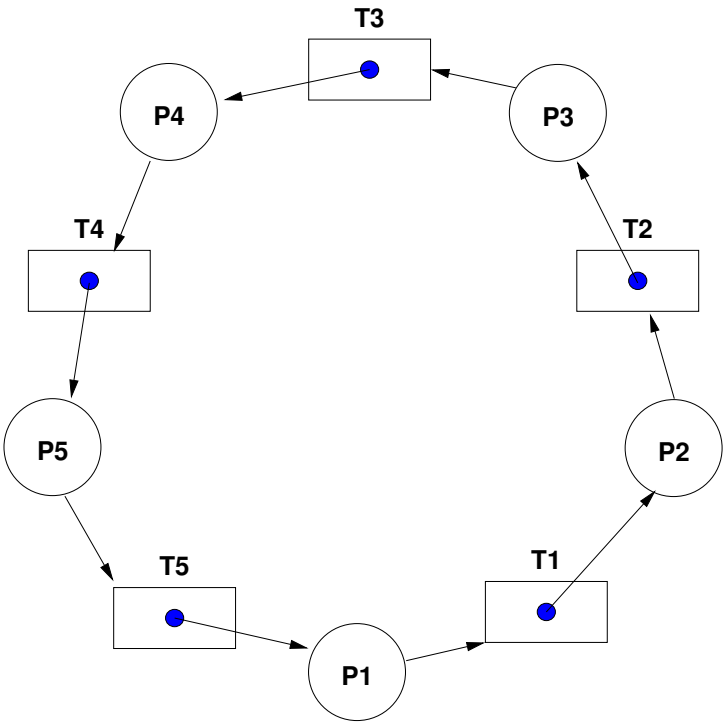
Resource-Allocation Graph (RAG)

Definition (resource-allocation graph)

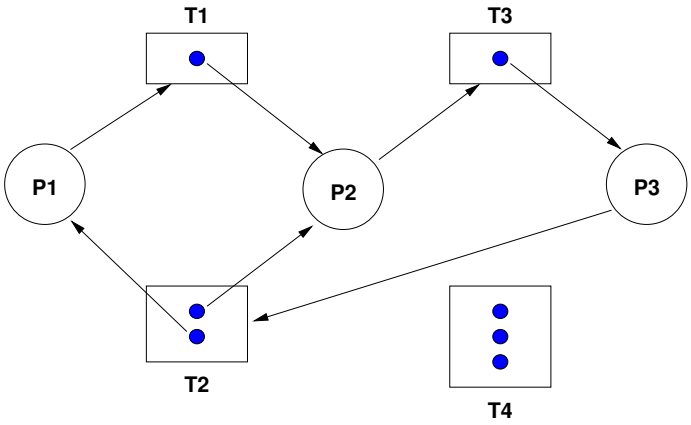
A *resource-allocation graph* is a directed graph $RAG = (V, E)$. The vertices V are partitioned into a set of processes P_i, a set of resource types T_i, and a set of resource instances R_i. Resource instances belong to a resource type. The set of edges E is partitioned into a set of resource assignments E_a, a set of resource requests E_r, and a set of future resource claims E_c.

- A directed edge $e \in E_a$ from a resource instance R_i to a process P_i indicates that the instance R_i has been assigned to P_i.
- A directed edge $e \in E_r$ from a process P_i to a resource type T_i indicates that P_i is requesting a resource of type T_i.
- A directed edge $e \in E_c$ from a process P_i to a resource type T_i indicates that P_i will be requesting a resource of type T_i in the future.

Below is a RAG showing a deadlock situation in a naive solution of the dining philosophers problem where every philosopher picks up the fork on the left.



Resource-Allocation Graph (RAG)

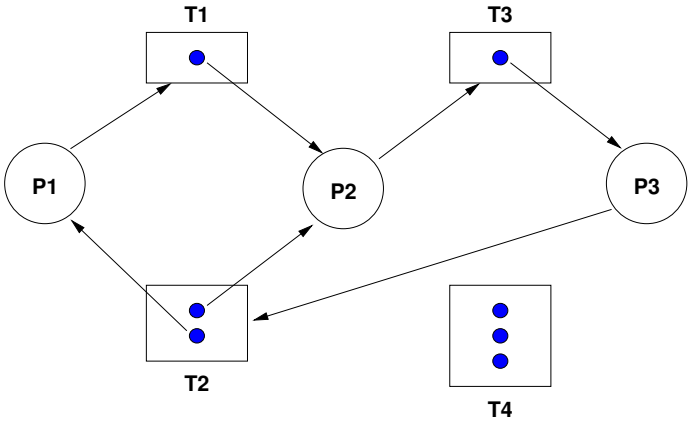


$$
\begin{aligned}
RAG &= \{V, E\} \\
V &= P \cup T \cup R \\
E &= E_a \cup E_r \cup E_c \\
P &= \{P_1, P_2, \ldots, P_n\} \\
T &= \{T_1, T_2, \ldots, T_m\} \\
R &= \{R_1, R_2, \ldots, R_m\} \\
E_a &= \{R_j \rightarrow P_i\} \\
E_r &= \{P_i \rightarrow T_j\} \\
E_c &= \{P_i \rightarrow T_j\}
\end{aligned}
$$

RAG Properties

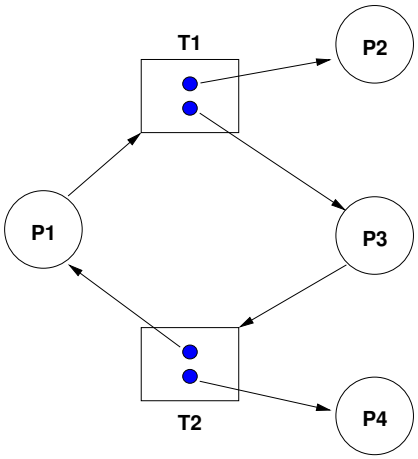
- Properties of a Resource-Allocation Graph:
  - A cycle in the RAG is a necessary condition for a deadlock
  - If each resource type has exactly one instance, then a cycle is also a sufficient condition for a deadlock
  - If resource types have multiple instances, then a cycle is not a sufficient condition for a deadlock
- Dashed claim arrows (E_c) can express that a future claim for an instance of a resource is already known
- Information about future claims can help to avoid situations which can lead to deadlocks

RAG Example #1



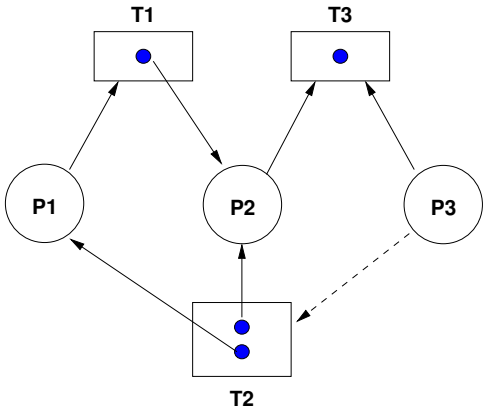
- Cycle 1:
  $P_1 \to T_1 \to P_2 \to T_3 \to P_3 \to T_2 \to P_1$
- Cycle 2:
  $P_2 \to T_3 \to P_3 \to T_2 \to P_2$
- $\{P_1, P_2, P_3\}$ are deadlocked

RAG Example #2



- Cycle:
  $P_1 \rightarrow T_1 \rightarrow P_3 \rightarrow T_2 \rightarrow P_1$
- $\{P_1, P_3\}$ are not deadlocked
- P_4 may break the cycle by releasing its instance of T_2

RAG Example #3



- P_2 and P_3 both request T_3
- To which process should the resource be assigned?
- Assigning an instance of T_3 to P_2 avoids a future deadlock

Section 19: Deadlock Strategies

Deadlock Strategies

- *Prevention*:
  The system is designed such that deadlocks can never occur
- *Avoidance*:
  The system assigns resources so that deadlocks are avoided
- *Detection and recovery*:
  The system detects deadlocks and recovers itself
- *Ignorance*:
  The system does not care about deadlocks and the user has to take corrective actions

Deadlock Prevention

- Idea: Ensure that at least one of the necessary conditions cannot hold
  1. Prevent mutual exclusion:
     Some resources are intrinsically non-sharable
  2. Prevent hold and wait:
     Low resource utilization and starvation possible
  3. Prevent no preemption:
     Preemption can not be applied to some resources such as printers or tape drives
  4. Prevent circular wait:
     Leads to low resource utilization and starvation if the imposed order does not match process requirements
- Deadlock prevention is only feasible in special cases

Deadlock Avoidance

Definition (safe state)

A resource allocation state is *safe* if the system can allocate resources to each process (up to its claimed maximum) and still avoid a deadlock.

Definition (unsafe state)

A resource allocation state is *unsafe* if the system cannot prevent processes from requesting resources such that a deadlock can occur.

- Note: An unsafe state does not necessarily lead to a deadlock.
- Assumption: For every process, the maximum resource claims are known a priori.
- Idea: Only grant resource requests that can not lead to a deadlock situation.

Banker's Algorithm Notation

Symbol	Description	Name
$n \in \mathbb{N}$	number of processes	
$m \in \mathbb{N}$	number of resource types	
$t \in \mathbb{N}^m$	total number of resource instances	*total*
$u \in \mathbb{N}^m$	number of used resource instances	*used*
$a \in \mathbb{N}^m$	number of available resource instances	*avail*
$M \in \mathbb{N}^{n \times m}$	$m_{i,j}$ maximum claim of type j by process i	*max*
$A \in \mathbb{N}^{n \times m}$	$a_{i,j}$ resources of type j allocated to process i	*alloc*
$N \in \mathbb{N}^{n \times m}$	$n_{i,j}$ maximum needed resources of type j by process i	*need*
R	set of processes that can get their needed resources	*ready*

The Banker's algorithm was presented by Edsger W. Dijkstra [11]. Given the notation, some important relationships can be identified easily:

$t = u + a$	total t is the sum of the used and available resources
$M = A + N$	max M is the sum of the allocated and maximum needed resources
$u = colsum(A)$	allocated resources a is the columnar sum of the allocation matrix A

Lets consider a system that has $m = 4$ different resource types and the total number of resource instances of each resource type is given by $t = (6, 8, 10, 12)$, i.e., there are 6 instances of the first resource type, 8 instanced of the second and so on. We have $n = 5$ running processes. For every process, the maximum resource requests are known and given by the matrix M:

$$M = \begin{bmatrix} 3 & 1 & 2 & 5 \\ 3 & 2 & 5 & 7 \\ 2 & 6 & 3 & 1 \\ 5 & 4 & 9 & 2 \\ 1 & 3 & 8 & 9 \end{bmatrix}$$

The first row says that the first process may request up to 3 instances of the first resource type, 1 instance of the second resource type, etc. A given state of the system is described by the allocation matrix A.

$$A = \begin{bmatrix} 0 & 0 & 2 & 1 \\ 1 & 0 & 1 & 2 \\ 1 & 2 & 1 & 1 \\ 3 & 4 & 0 & 0 \\ 0 & 0 & 4 & 2 \end{bmatrix}$$

The column sums of A give us $u = (5, 6, 8, 6)$. With u, we can calculate the available resources $a = t - u = (6, 8, 10, 12) - (5, 6, 8, 6) = (1, 2, 2, 6)$.

Safe-State Algorithm

```
1:  function isSafe(total, max, alloc)
2:      loop
3:          need ← max − alloc                          ▷ Needed resources
4:          avail ← total − colsum(alloc)               ▷ Currently available resources
5:          ready ← filter(need, avail)                 ▷ Processes that can get their resources
6:          if ready ≡ ∅ then
7:              return (alloc ≡ ∅)                      ▷ Safe if alloc is empty
8:          end if
9:          proc ← select(ready)                        ▷ Select a process that is ready
10:         alloc ← remove(alloc, proc)                 ▷ Remove process from alloc
11:         max ← remove(max, proc)                     ▷ Remove process from max
12:     end loop
13: end function
```

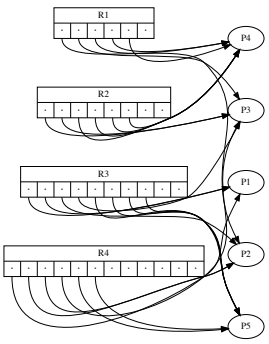

is the system state described by the following allocation matrix A safe?

$$A = \begin{bmatrix} 0 & 0 & 2 & 1 \\ 1 & 0 & 1 & 2 \\ 1 & 2 & 1 & 1 \\ 3 & 4 & 0 & 0 \\ 0 & 0 & 4 & 2 \end{bmatrix}$$

To answer this question, we calculate how many resources are still available, $a = (1, 2, 2, 6)$, and the remaining maximum need N for all processes:

$$N = M - A = \begin{bmatrix} 3 & 1 & 2 & 5 \\ 3 & 2 & 5 & 7 \\ 2 & 6 & 3 & 1 \\ 5 & 4 & 9 & 2 \\ 1 & 3 & 8 & 9 \end{bmatrix} - \begin{bmatrix} 0 & 0 & 2 & 1 \\ 1 & 0 & 1 & 2 \\ 1 & 2 & 1 & 1 \\ 3 & 4 & 0 & 0 \\ 0 & 0 & 4 & 2 \end{bmatrix} = \begin{bmatrix} 3 & 1 & 0 & 4 \\ 2 & 2 & 4 & 5 \\ 1 & 4 & 2 & 0 \\ 2 & 0 & 9 & 2 \\ 1 & 3 & 4 & 7 \end{bmatrix}$$

Given the available resources $a = (1, 2, 2, 6)$, there is no process that can get all his needs. Hence, the set of processes that are ready is empty, ($R = \emptyset$). Since there are still allocations left, the system is in an unsafe state. A system using the Banker's algorithm should not have gotten into this state. The resource allocation graph looks like this:



Resource-Request Algorithm

```
1:  function RESQUESTRESOURCES(total, max, alloc, request)
2:      need ← max − alloc                          ▷ Needed resources
3:      avail ← total − colsum(alloc)               ▷ Currently available resources
4:      if request > need then
5:          error illegal                           ▷ Request exceeds available resources
6:      end if
7:      if request ≤ avail then
8:          alloc′ ← alloc + request                ▷ Pretend to grant the request
9:          if isSafe(total, max, alloc′) then      ▷ Check whether the new state is safe
10:             return True                         ▷ Grant the resource request since its safe
11:         end if
12:     end if
13:     return False                                ▷ Request not granted at this point in time
14: end function
```


Assume the system is in the state described by the following allocation matrix:

$$A = \begin{bmatrix} 1 & 0 & 2 & 1 \\ 1 & 1 & 2 & 5 \\ 1 & 2 & 3 & 1 \\ 1 & 1 & 1 & 1 \\ 1 & 0 & 2 & 2 \end{bmatrix} \qquad N = M - A = \begin{bmatrix} 3 & 1 & 2 & 5 \\ 3 & 2 & 5 & 7 \\ 2 & 6 & 3 & 1 \\ 5 & 4 & 9 & 2 \\ 1 & 3 & 8 & 9 \end{bmatrix} - \begin{bmatrix} 1 & 0 & 2 & 1 \\ 1 & 1 & 2 & 5 \\ 1 & 2 & 3 & 1 \\ 1 & 1 & 1 & 1 \\ 1 & 0 & 2 & 2 \end{bmatrix} = \begin{bmatrix} 2 & 1 & 0 & 4 \\ 2 & 1 & 3 & 2 \\ 1 & 4 & 0 & 0 \\ 4 & 3 & 8 & 1 \\ 0 & 3 & 6 & 7 \end{bmatrix}$$

How should the system react if process 4 requests an instance of resource 4?

Lets assume the request can be granted and we check whether the new state is safe.

$$A' = \begin{bmatrix} 1 & 0 & 2 & 1 \\ 1 & 1 & 2 & 5 \\ 1 & 2 & 3 & 1 \\ 1 & 1 & 1 & 2 \\ 1 & 0 & 2 & 2 \end{bmatrix} \qquad N' = M - A' = \begin{bmatrix} 3 & 1 & 2 & 5 \\ 3 & 2 & 5 & 7 \\ 2 & 6 & 3 & 1 \\ 5 & 4 & 9 & 2 \\ 1 & 3 & 8 & 9 \end{bmatrix} - \begin{bmatrix} 1 & 0 & 2 & 1 \\ 1 & 1 & 2 & 5 \\ 1 & 2 & 3 & 1 \\ 1 & 1 & 1 & 2 \\ 1 & 0 & 2 & 2 \end{bmatrix} = \begin{bmatrix} 2 & 1 & 0 & 4 \\ 2 & 1 & 3 & 2 \\ 1 & 4 & 0 & 0 \\ 4 & 3 & 8 & 0 \\ 0 & 3 & 6 & 7 \end{bmatrix}$$

We have to find a sequence such that all processes can obtain their needed resources and then terminate. The current set of available resources is $a = (1, 4, 0, 1)$. This allows process 3 to obtain all needed resources and when it will terminate, process 3 returns all allocated resources and this allows other processes to obtain their needed resources. In the example, we go through the following sequence:

$a = (1, 4, 0, 1)$	$R = \{3\}$	termination of process 3
$a = (2, 6, 3, 2)$	$R = \{2\}$	termination of process 2
$a = (3, 7, 5, 7)$	$R = \{1\}$	termination of process 1
$a = (4, 7, 7, 8)$	$R = \{5\}$	termination of process 5
$a = (5, 7, 9, 10)$	$R = \{4\}$	termination of process 4
$a = (6, 8, 10, 12)$	$R = \{\}$	stop

Since there is a sequence that allows all processes to receive their still needed resources, the new state is safe and the resource request can be granted.

Deadlock Detection

- Idea:
  - Assign resources without checking for unsafe states
  - Periodically run an algorithm to detect deadlocks
  - Once a deadlock has been detected, use an algorithm to recover from the deadlock
- Recovery:
  - Abort one or more deadlocked processes
  - Preempt resources until the deadlock cycle is broken
- Issues:
  - Criterias for selecting a victim?
  - How to avoid starvation?

Deadlock-Detection Algorithm

```
1: function isDeadlocked(total, need, alloc)
2:     loop
3:         avail ← total − colsum(alloc)          ▷ Currently available resources
4:         ready ← filter(need, avail)            ▷ Processes that can get their resources
5:         if ready ≡ ∅ then
6:             return (alloc ≢ ∅)                 ▷ Deadlocked if alloc is not empty
7:         end if
8:         proc ← select(ready)                   ▷ Select a process that is ready
9:         alloc ← remove(alloc, proc)            ▷ Remove process from alloc
10:    end loop
11: end function
```


This is a slight variation of the safe-state algorithm. It does not require to know the maximum resource requests. But then it is also limited to the detection of deadlocks, it does not help to prevent deadlocks.

Wait-For Graph (WFG)

Definition (wait-for graph)

A *wait-for graph* is a directed graph $WFG = (V, E)$. The vertices V represent processes and an edge $e = (P_i, P_j) \in E$ indicates that process P_i is waiting for process P_j.

- A cycle in a wait-for graph indicates a deadlock
- Resource allocation graphs (RAGs), where every resource type has only a single instance, can be easily transformed into wait-for graphs (WFGs)
- Constructing and maintaining WFG graphs is relatively expensive

Desktop operating systems like Windows or Linux usually do not try to detect or avoid deadlocks. It is up to the designers of application software to find design solutions that are deadlock free.

Operationally, deadlocks are sometimes identified through system monitoring and or they are dealt with by regularly restarting processes:

- If the throughput of a system goes down and at the same time the load of a system goes down, then a deadlock can be a possible explanation.
- Processes often leak resources (i.e., they acquired resources but forgot to release them). In some operational settings, processes are restarted periodically in order to free any leaked resources. (This is also called sofware aging and software rejuvenation.) Periodic restarts of processes often have the side effect that they resolve deadlocks.

Distributed Deadlock Detection

- *Path-pushing algorithms* detect distributed deadlocks by maintaining a global WFG. Nodes push paths to a central deadlock detector or their neighbors.
- *Edge-chasing algorithms* verify the presence of a cycle in a distributed graph structure by propagating special messages (called probes) along the edges of the graph.
- *Diffusion computation* algorithms detect deadlocks by diffusing the computation via an echo algorithm. They superimpose the detection on a distributed computation.
- *Global state detection algorithms* detect snapshots by analyzing a consistent snapshot of a distributed system.

Detection of deadlocks in distributed systems has been a significant research topic.

Part VI

Scheduling

A scheduler decides which processes or threads are receiving resources in order to continue with their computations. The decisions taken by a scheduler determine throughput and responsiveness of applications running on an operating system. We will focus on CPU scheduling, i.e., the assignment of CPU resources to processes and threads. There are many other schedulers in modern operating systems like I/O schedulers or network packet transmission schedulers that we will not discuss here.

By the end of this part, students should be able to

- describe the purpose and goals of a scheduler;
- explain the concept of a fair-share scheduler;
- distinguish preemptive and non-preemptive schedulers;
- differentiate between deterministic and probabilistic schedulers;
- construct Gantt diagrams for deterministic schedules;
- create FCFS, LPTF, SPTF, and SRTF schedules;
- determine round-robin and multilevel feedback queue schedules;
- distinguish real-time scheduling algorithms.

Section 20: Scheduler

Scheduler and CPU Scheduler

Definition (scheduler)

A *scheduler* (or a scheduling discipline) is an algorithm that distributes resources to parties, which simultaneously and asynchronously request them.

Definition (cpu scheduler)

A *CPU scheduler* is a scheduler, which distributes CPU time to processes (or threads) that are ready to execute.

There are many scheduler in an operating system. The CPU scheduler takes a special role because it decides who CPU time is distributed to processes and threads, i.e., which computations proceed. However, there are other resources that an operating system has to manage and where competition needs to be handled.

- I/O devices are usually much slower than the processing core and hence I/O requests may queue up. An I/O scheduler determines in which order I/O requests are processed. Depedending on the characteristics of the I/O device, clever I/O scheduling can improve I/O throughput (and bad I/O scheduling can harm throughput).
- Network communication typically involves queues in front of outgoing network interfaces, either within an end host or within switches and routers in the network infrastructure. Various scheduling strategies are used in order to handle such queues.
- Operating systems typically have to schedule regular maintenance functions. While the maintenance functions typically are not time critical, the goal is often to schedule them in such a way that the impact on the regular usage of the system is minimized.

Scheduler Goals

Goal	Description
Fairness	Every process receives a fair amount of the resources available
Efficiency	Keep resources busy whenever there are processes ready to run
Response Time	Minimize the response time for interactive applications
Wait Time	Minimize the time spend waiting instead of executing
Turnaround Time	Minimize the time from process creation until termination
Throughput	Maximize the number of processes completed over a time interval
Scalability	Low overhead of the scheduler itself

- Some of these goals are conflicting, hence trade-off decisions must be taken
- Taking good (not perfect) scheduling decisions quickly is often the way to go

Fair-Share Scheduler

Definition (fair-share scheduler)

A fair-share scheduler is a scheduler that aims to distribute resources fairly between users of a system as opposed to equal distribution among the parties requesting resources.

- Fair-share scheduling avoids that users "game the system" by splitting work over many processes in order to obtain overall a higher CPU share than other users.
- Fair-share scheduling is also important for managing network resources since otherwise users may start many concurrent network connections in order to obtain a larger share of the available network bandwidth.

Preemptive vs. Non-preemptive

Definition (preemptive scheduler)

A *preemptive* scheduler can interrupt a running process or thread and assign the assigned resources (e.g., CPU time) to another process.

Definition (non-preemptive scheduler)

A *non-preemptive* scheduler waits for the process or thread to yield resources (e.g., the CPU) once they have been assigned to the process or thread.

- Non-preemptive schedulers cannot guarantee fairness
- Preemptive schedulers are harder to design

Preemptive schedulers might preempt processes or threads at times where the preemption is costly (e.g., in the middle of a critical section).

Preemptive schedulers are the norm on computing systems with full-fledged processors. On embedded systems, the situation is often different.

Deterministic vs. Probabilistic

Definition (deterministic scheduler)

A *deterministic* scheduler knows the resource requests of the processes and threads and optimizes the resource assignment to optimize system behavior (e.g., maximize throughput).

Definition (probabilistic scheduler)

A *probabilistic* scheduler describes process and thread behavior using certain probability distributions (e.g., process arrival rate distribution, service time distribution) and optimizes the overall system behavior based on these probabilistic assumptions.

- Deterministic schedulers are relatively easy to analyze
- Probabilistic schedulers must be analyzed using stochastic models (queuing models)

Finding optimal schedules often leads to algorithms with an undesirable complexity. In order to limit the computational overhead caused by schedulers, operating systems often prefer heuristics that are fast to compute but which may not always produce optimal results.

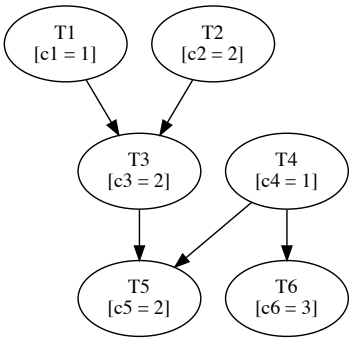
Metrics of a Schedule

Metric	Description
a_i	Arrival time of process i
e_i	End time or completion time of process i
t_i	Turnaround time of process i, obviously $t_i = e_i - a_i$
c_i	Compute or execution time of process i
w_i	Waiting time of pocess i, obviously $w_i = t_i - c_i$
$\bar{t}$	Average turnaround time, $\bar{t} = \frac{1}{n}\sum_i t_i$
$\bar{w}$	Average waiting time, $\bar{w} = \frac{1}{n}\sum_i w_i$
L	Length of a schedule, obviously $L = \max_i e_i$

Deterministic Scheduling

- A *deterministic schedule* S for a set of processors $P = \{P_1, P_2, \ldots, P_m\}$ and a set of tasks $T = \{T_1, T_2, \ldots, T_n\}$ with the execution times $c = \{c_1, c_2, \ldots c_n\}$ and a set D of dependencies between tasks is a temporal assignment of the tasks to the processors.
- A *precedence graph* $G = (T, E)$ is a directed acyclic graph which defines dependencies between tasks. The vertices of the graph are the tasks T. An edge from T_i to T_j indicates that task T_j may not be started before task T_i is complete.

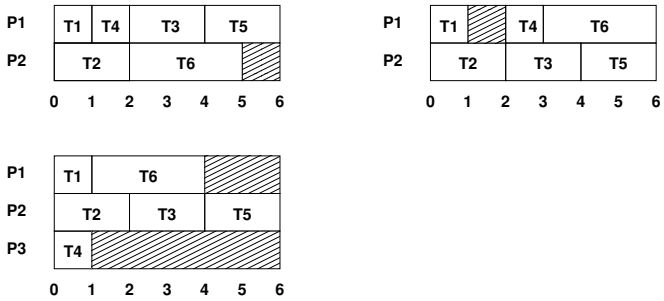
Deterministic Scheduling Example



$$G = (T, E)$$
$$T = \{ T_1, T_2, T_3, T_4, T_5, T_6 \}$$
$$E = \{ (T_1 \prec T_3), (T_2 \prec T_3), (T_3 \prec T_5), (T_4 \prec T_5), (T_4 \prec T_6) \}$$
$$c = (c_1, c_2, c_3, c_4, c_5, c_6) = (1, 2, 2, 1, 2, 3)$$
$$n = |T| = 6$$

$$P = \{ P_1, P_2 \}$$
$$m = |P| = 2$$

Gantt Diagrams



- Both schedules for $m = 2$ processors have the same length and the same average turnaround and waiting times.
- The schedule for $m = 3$ processors has the same length but different average turnaround and waiting times.

- First schedule (top left):

$$L = 6$$
$$\bar{t} = \frac{1}{6}(1+2+2+4+5+6) = \frac{10}{3}$$
$$\bar{w} = \frac{1}{6}(0+0+1+2+2+4) = \frac{9}{6}$$

- Second schedule (top right):

$$L = 6$$
$$\bar{t} = \frac{1}{6}(1+2+4+3+6+6) = \frac{11}{3}$$
$$\bar{w} = \frac{1}{6}(0+0+2+2+4+3) = \frac{11}{6}$$

- Third schedule (bottom left):

$$L = 6$$
$$\bar{t} = \frac{1}{6}(1+1+2+4+4+6) = \frac{9}{3}$$
$$\bar{w} = \frac{1}{6}(0+0+0+1+2+4) = \frac{7}{6}$$

Section 21: Scheduling Strategies

First-Come, First-Served (FCFS)

- Assumptions:
  - No preemption of running processes
  - Arrival times of processes are known
- Principle:
  - Processors are assigned to processes on a first come first served basis (under observation of any precedences)
- Properties:
  - Straightforward to implement
  - Average waiting time can become quite large

Longest Processing Time First (LPTF)

- Assumptions:
  - No preemption of running processes
  - Execution times of processes are known
- Principle:
  - Processors are assigned to processes with the longest execution time first
  - Shorter processes are kept to fill "gaps" later
- Properties:
  - LPTF schedules are good approximations for optimal schedules in respect to the schedule length. For the length L_L of an LPTF schedule and the length L_O of an optimal schedule with m processors, the following holds:

$$L_L \leq \left(\frac{4}{3} - \frac{1}{3m}\right) \cdot L_O$$

For a derivation of the equation see [15].

Shortest Processing Time First (SPTF)

- Assumptions:
  - No preemption of running processes
  - Execution times of processes are known
- Principle:
  - Processors are assigned to processes with the shortest execution time first
- Properties:
  - The SPTF algorithm produces schedules with the minimum average waiting time for a given set of processes and non-preemptive scheduling
  - Also known as Shortst Job First (SJF)

Shortest Remaining Time First (SRTF)

- Assumptions:
  - Preemption of running processes
  - Execution times of the processes are known
- Principle:
  - Processors are assigned to processes with the shortest remaining execution time first
  - New arriving processes with a shorter execution time than the currently running processes will preempt running processes
- Properties:
  - The SRTF algorithm produces schedules with the minimum average waiting time for a given set of processes and preemptive scheduling

Round Robin (RR)

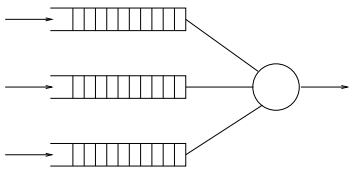
- Assumptions:
  - Preemption of running processes
  - Execution times or the processes are unknown
- Principle:
  - Processes are assigned to processors using a FCFS queue
  - After a small unit of time (time slice), the running processes are preempted and added to the end of the FCFS queue
- Properties:
  - time slice $\to \infty$: FCFS scheduling
  - time slice $\to 0$: processor sharing (idealistic)
  - Choosing a "good" time slice is important

Round Robin Variations

- Use separate queues for each processor
  - keep processes assigned to the same processor
- Use a short-term queue and a long-term queue
  - limit the number of processes that compete for the processor on a short time period
- Different time slices for different types of processes
  - degrade impact of processor-bound processes on interactive processes
- Adapt time slices dynamically
  - can improve response time for interactive processes

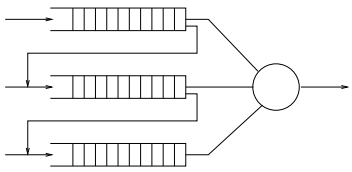
$\Longrightarrow$ Tradeoff between responsiveness and throughput

Multilevel Queue Scheduling



- Principle:
  - Multiple queues for processes with different priorities
  - Processes are permanently assigned to a queue
  - Each queue has its own scheduling algorithm
  - Additional scheduling between the queues necessary
- Properties:
  - Overall queue scheduling important (static vs. dynamic partitioning)

Multilevel Feedback Queue Scheduling



- Principle:
  - Multiple queues for processes with different priorities
  - Processes can move between queues
  - Each queue has its own scheduling algorithm
- Properties:
  - Very general and configurable scheduling algorithm
  - Queue up/down grade critical for overall performance

Real-time Scheduling

- *Hard real-time systems* must complete a critical task within a guaranteed amount of time
  - Scheduler needs to know exactly how long each operating system function takes to execute
  - Processes are only admitted if the completion of the process in time can be guaranteed
- *Soft real-time systems* require that critical tasks always receive priority over less critical tasks
  - Priority inversion can occur if high priority soft real-time processes have to wait for lower priority processes in the kernel
  - One solution is to give processes a high priority until they are done with the resource needed by the high priority process (priority inheritance)

Earliest Deadline First (EDF)

- Assumptions:
  - Deadlines for the real-time processes are known
  - Execution times of operating system functions are known
- Principle:
  - The process with the earliest deadline is always executed first
- Properties:
  - Scheduling algorithm for hard real-time systems
  - Can be implemented by assigning the highest priority to the process with the first deadline
  - If processes have the same deadline, other criterias can be considered to schedule the processes

Linux Scheduler System Calls

```
#include <unistd.h>

int nice(int inc);

#include <sched.h>

int sched_setscheduler(pid_t pid, int policy, const struct sched_param *p);
int sched_getscheduler(pid_t pid);
int sched_setparam(pid_t pid, const struct sched_param *p);
int sched_getparam(pid_t pid, struct sched_param *p);
int sched_get_priority_max(int policy);
int sched_get_priority_min(int policy);
int sched_rr_get_interval(pid_t pid, struct timespec *tp);
int sched_setaffinity(pid_t pid, unsigned int len, unsigned long *mask);
int sched_getaffinity(pid_t pid, unsigned int len, unsigned long *mask);
int sched_yield(void);
```

Part VII
Linking

Larger programs consists of multiple source files that are compiled separately into multiple object files. Collections of related object files that are generally useful may be put together into a reusable library. When an executable file is created, the object files derived from the source code of the program plus the relevant content of any used libraries needs to be linked together.

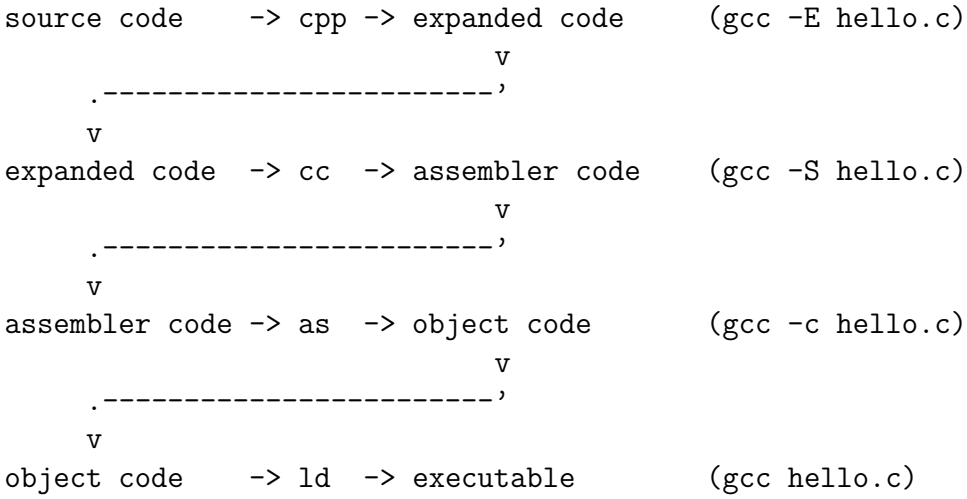
In this section we take a closer look at the linker and how it resolves symbols in order to produce an executable that has no undefined symbols anymore. We will also discuss that some of the linking may be deferred to the program load time, so called dynamic linking.

By the end of this part, students should be able to

- describe the functions performed by a linker;
- distinguish different object code file types;
- recognize different kinds of linker symbols;
- construct shared and static libraries;
- explain different interpositioning teachniques.

Section 22: Linker

Stages of the C Compilation Process



Compiling C source code is traditionally a four-stage process. Modern compilers often integrate stages for efficiency reasons but it is still possible (and instructive) to look at the artifacts created during the various stages.

Reasons for using a Linker

- Modularity
  - Programs can be we written as a collection of small files
  - Creating collections (libraries) of reusable functions
- Efficiency
  - Separate compilation of a subset of small files saves time on large projects
  - Smaller executables by linking only functions that are needed
  - Even smaller executables by lazily linking some functions at program start time

A general goal is to not repeat program logic. If a certain logic is needed multiple times, factor it into a collection of reusable functions. It is good practice to define unit test cases for collections of reusable functions. Modularity helps to reduce maintenance cost.

Once you have factored code into smaller modules, you also get the benefit of reduced build times. Good build tools will determine the dependencies between source code files and then rebuild only what needs to be rebuild. On large projects, compilation time can add up. Try to compile a Linux kernel yourself or a C compiler or a Web browser to get an idea how long builds can be.

What does a Linker do?

- Symbol resolution
  - Programs define and reference symbols (variables or functions)
  - Symbol definitions and references are stored in object files
  - Linker associate each symbol reference with exactly one symbol definition
- Relocation
  - Merge separate code and data sections into combined sections
  - Relocate symbols from relative locations to their final absolute locations
  - Update all references to these symbols to reflect their new positions

Relocation becomes simpler if the compiler produced so called position independent code, i.e., code that can be placed into different locations of memory without having to adjust it. The opposite of position independent code is absolute code, which has to be positioned into a specific memory location to function correctly or which needs to be adopted by the linker.

Sometimes the work of the linker can be simplified by using some indirection. For example, instead of resolving all calls of a function at link time, the function calls may be done via a function pointer table such that at link or resolution time only the function pointer table needs to be updated.

Object Code File Types

- Relocatable object files (.o files)
  - Contain code and data that can be combined with other relocatable object files
- Library archive files (.a files)
  - Contain code and data of many object files with an additional index
- Shared library/object files (.so files)
  - Special type of relocatable object or library files
  - Can be linked by a dynamic linker at program start time
  - Can be loaded dynamically during the execution of a program
- Executable object files
  - Contain code and data in a form that can be loaded directly into process memory

The `file` utility can be used to quickly determine the type of a file. The utility displays useful meta-information for the various object code file types.

The `objdump` tool can be used to inspect the content of object code files. The `-s` option provides an overview of the sections stored in an object file.

```
$ objdump -s hello-naive.o

hello-naive.o:     file format elf64-x86-64

Contents of section .text:
 0000 554889e5 488d3d00 000000e8 00000000  UH..H.=.........
 0010 b8000000 005dc3                      .....].
Contents of section .rodata:
 0000 48656c6c 6f20576f 726c6400           Hello World.
Contents of section .comment:
 0000 00474343 3a202844 65626961 6e20382e  .GCC: (Debian 8.
 0010 332e302d 36292038 2e332e30 00        3.0-6) 8.3.0.
Contents of section .eh_frame:
 0000 14000000 00000000 017a5200 01781001  .........zR..x..
 0010 1b0c0708 90010000 1c000000 1c000000  ................
 0020 00000000 17000000 00410e10 8602430d  .........A....C.
 0030 06520c07 08000000                    .R......
```

The `-t` option displays the symbols defined in or referenced from an object file.

Executable and Linkable Format

- Standard unified binary format for all object files
- ELF header provides basic information (word size, endianess, machine architecture, structure of the ELF file, ...)
- Program header table describes zero or more segments used at runtime
- Section header table provides information about zero or more sections
- Separate sections for `.text`, `.rodata`, `.data`, `.bss`, `.symtab`, `.rel.text`, `.rel.data`, `.debug` and many more
- The `readelf` tool can be used to read ELF format
- The tool `objdump` can process ELF formatted object files
- The `size` tool provides a quick overview of section sizes
- The `strip` tool can remove symbols from ELF files

A common file format is the Executable and Linkable Format (ELF) format. The format is flexible, extensible, and cross-platform. An ELF file starts with an ELF header and is followed by a sequence of sections. An ELF header contains information like this:

```
$ readelf -e hello-naive.o
ELF Header:
  Magic:   7f 45 4c 46 02 01 01 00 00 00 00 00 00 00 00 00
  Class:                             ELF64
  Data:                              2's complement, little endian
  Version:                           1 (current)
  OS/ABI:                            UNIX - System V
  ABI Version:                       0
  Type:                              REL (Relocatable file)
  Machine:                           Advanced Micro Devices X86-64
  Version:                           0x1
  Entry point address:               0x0
  Start of program headers:          0 (bytes into file)
  Start of section headers:          696 (bytes into file)
  Flags:                             0x0
  Size of this header:               64 (bytes)
  Size of program headers:           0 (bytes)
  Number of program headers:         0
  Size of section headers:           64 (bytes)
  Number of section headers:         13
  Section header string table index: 12
```

A quick overview of the main sections and their sizes is provided by the `size` utility.

```
$ size hello-naive
   text    data     bss     dec     hex filename
   1457     584       8    2049     801 hello-naive
```

Calling `objdump -S` disassembles the code stored in the sections that contain machine code.

Further online information:

- **Wikipedia**: Executable and Linkable Format

Linker Symbols

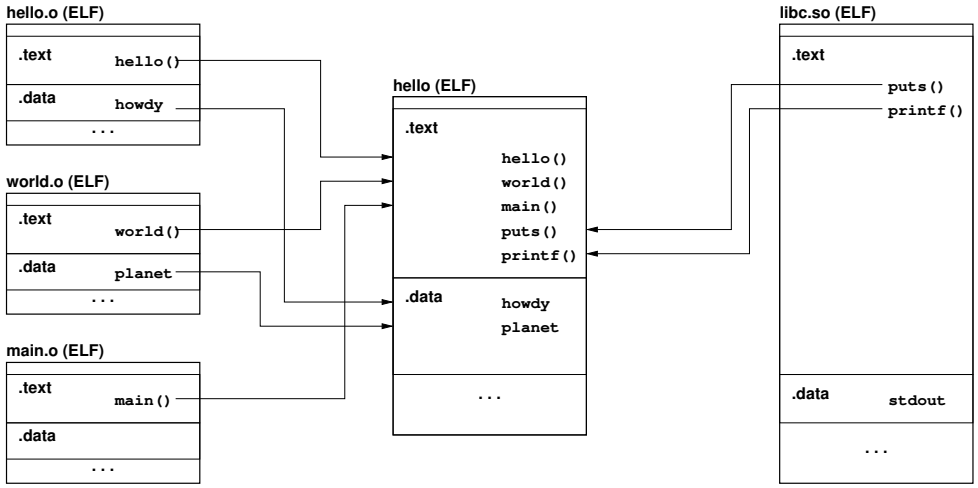
- Global symbols
  - Symbols defined by a module that can be referenced by other modules
- External symbols
  - Global symbols that are referenced by a module but defined by some other module
- Local symbols
  - Symbols that are defined and referenced exclusively by a single module

- Tools:
  - The tool `nm` displays the (symbol table) of object files in a traditional format
  - The newer tool `objdump -t` does the same for ELF object files

The tool `nm` can be used to obtain a quick overview of the defined and referenced symbols. In the example below, we can see that `hello-naive.o` defined the symbol `main` (T = symbol in the text section) and has the undefined symbol `puts`.

```
$ nm hello-naive.o
                 U _GLOBAL_OFFSET_TABLE_
0000000000000000 T main
                 U puts

$ objdump -t hello-naive.o

hello-naive.o:     file format elf64-x86-64

SYMBOL TABLE:
0000000000000000 l    df *ABS* 0000000000000000 hello-naive.c
0000000000000000 l    d  .text 0000000000000000 .text
0000000000000000 l    d  .data 0000000000000000 .data
0000000000000000 l    d  .bss 0000000000000000 .bss
0000000000000000 l    d  .rodata 0000000000000000 .rodata
0000000000000000 l    d  .note.GNU-stack 0000000000000000 .note.GNU-stack
0000000000000000 l    d  .eh_frame 0000000000000000 .eh_frame
0000000000000000 l    d  .comment 0000000000000000 .comment
0000000000000000 g     F .text 0000000000000017 main
0000000000000000         *UND* 0000000000000000 _GLOBAL_OFFSET_TABLE_
0000000000000000         *UND* 0000000000000000 puts
```

Example: Linking main.o, hello.o, world.o



The `hello.c` module uses a static (local) character string as a default greeting.

```
1  /* hello-ld/hello.c */
2
3  #include "common.h"
4
5  static const char *howdy = "hello";
6
7  void hello(char *greeting)
8  {
9      printf("%s ", (! greeting) ? howdy : greeting);
10 }
```

Similarly, the `world.c` module uses a static (local) character string as a default region.

```
1  /* hello-ld/world.c */
2
3  #include "common.h"
4
5  static const char *planet = "world";
6
7  void world(char *region)
8  {
9      puts((! region) ? planet : region);
10 }
```

And finally, here is a `main.c` module calling the functions defined in the other two modules.

```
1  /* hello-ld/main.c */
2
3  #include "common.h"
4
5  int main(void)
6  {
7      hello(NULL); world(NULL); return 0;
8  }
```

Strong and Weak Symbols and Linker Rules

- Strong Symbols
  - Functions and initialized global variables
- Weak Symbols
  - Uninitialized global variables
- Linker Rule #1
  - Multiple strong symbols with the same name are not allowed
- Linker Rule #2
  - Given a strong symbol and multiple weak symbols with the same name, choose the strong symbol
- Linker Rule #3
  - If there are multiple weak symbols with the same name, pick an arbitrary one

The linker resolves symbols without any knowledge about the types associated with the symbols. In fact, all type information is gone at the time the linker is called and hence, there is no type safe linking in C.

Early C++ compilers did translate the C++ source code to C code. In order to support features such as overloaded functions, it was necessary to encode type information into the symbols. This is called name mangling. Here is an example:

```
1  int f()
2  {
3      return 0;
4  }
5
6  int f(int i)
7  {
8      return i;
9  }
10
11 int main ()
12 {
13     return f(f());
14 }
```

On a Gnu/Linux system, here is what `nm` finds in the object file:

```
$ nm overload.o
0000000000000017 T main
000000000000000b T _Z1fi
0000000000000000 T _Z1fv
$ nm -C overload.o
0000000000000017 T main
000000000000000b T f(int)
0000000000000000 T f()
```

Note that name mangling algorithms are compiler specific.

Linker Puzzles

- Link time error due to two definitions of `p1`:

```
a.c: int x; p1() {}
b.c:        p1() {}
```

- Reference to the same uninitialized variable `x`:

```
a.c: int x; p1() {}
b.c: int x; p2() {}
```

- Reference to the same initialized variable `x`:

```
a.c: int x=1; p1() {}
b.c: int x;   p2() {}
```

- Writes to the double `x` likely overwrites `y`:

```
a.c: int x; int y; p1() {}
b.c: double x;     p2() {}
```

Section 23: Libraries

Static Libraries

- Collect related relocatable object files into a single file with an index (called an archive)
- Enhance the linker so that it tries to resolve external references by looking for symbols in one more more archives
- If an archive member file resolves a reference, link the archive member file into the executable (which may produce additional references)
- The archive format allows for incremental updates
- Example:

```
ar -rs libfoo.a foo.o bar.o
```


Linking of static libraries duplicates the object code. The linker essentially copies code from the library into executables. Creating deep copies of object code becomes a problem if some code was found to be buggy. To roll out a bug fix, the library needs to be updated (easy) and all programs that use the library need to be linked again (can become expensive if a library is very widely used).

Shared Libraries

- Idea: Delay the linking until program start and then link against the most recent matching versions of the required libraries
- At traditional link time, an executable file is prepared for dynamic linking (i.e., information is stored indicating which shared libraries are needed) while the final linking takes place when an executable is loaded into memory
- Benefits:
  1. Smaller executables since common code is not copied into executables
  2. Shared libraries can be updated without relinking all executables
  3. Library machine code and data can be stored in shared memory
  4. Programs can load additional object code dynamically at runtime

There is one caveat with shared libraries: Loading untrusted libraries can lead to big surprises. A program that is linked against shared libraries has to trust the runtime system to provide genuine shared libraries. An attacker might modify shared libraries in order to provide backdoors into arbitrary problems.

POSIX API (dlopen, dlclose, dlsym, dlerror)

```
#include <dlfcn.h>

#define RTLD_LAZY   ...   /* resolve symbols lazyly */
#define RTLD_NOW    ...   /* resolve symbols now */
#define RTLD_GLOBAL ...   /* enable global symbol resolution */
#define RTLD_LOCAL  ...   /* enable local symbol resolution */

void *dlopen(const char *filename, int flags);
int dlclose(void *handle);

#define RTLD_DEFAULT ...  /* find first occurance of symbol */
#define RTLD_NEXT    ...  /* fin next occurance of symbol */

void *dlsym(void *handle, const char *symbol);

char *dlerror(void);      /* obtain human readable error string */
```


Listing 25 demonstrates how a C program can load a shared library, search for a symbol in the library, and then use it as any other regular symbol. The flag `LD_LAZY` requests a lazy binding where symbols are resolved only when needed. The flag `LD_NOW` requests that all symbols are resolved during the `dlopen()` call. The `dlerror()` function returns the most recent error since the last call to `dlerror()`.

```c
/*
 * hello-dll.c --
 *
 *      This program uses the dynamic linking API to locate the puts()
 *      function in the C library and then uses it to print a message.
 */

#define _POSIX_C_SOURCE 200809L

#include <stdio.h>
#include <stdlib.h>
#include <dlfcn.h>

#if defined(__linux__)
#define LIBC "/lib/x86_64-linux-gnu/libc.so.6"
#elif defined(__APPLE__)
#define LIBC "/usr/lib/libc.dylib"
#else
#define LIBC "libc.so"
#endif

int main(void)
{
    void *handle;
    void (*println) (char *);
    char *error;

    handle = dlopen(LIBC, RTLD_LAZY);
    if (! handle) {
        fprintf(stderr, "%s\n", dlerror());
        exit(EXIT_FAILURE);
    }

    dlerror();
#pragma GCC diagnostic push
#pragma GCC diagnostic ignored "-Wpedantic"
    println = dlsym(handle, "puts");
#pragma GCC diagnostic pop
    if ((error = dlerror()) != NULL)  {
        fprintf(stderr, "%s\n", error);
        exit(EXIT_FAILURE);
    }

    println("hello world");
    (void) dlclose(handle);

    return EXIT_SUCCESS;
}
```

Listing 25: Demonstration of the dynamic linking API

Section 24: Interpositioning

Interpositioning

- Intercepting library calls can be useful for many reasons
  - *Debugging*: tracing memory allocations / leaks
  - *Profiling*: study typical function arguments
  - *Sandboxing*: emulate a restricted view on a file system
  - *Hardening*: simulate failures to test program robustness
  - *Privacy*: add encryption into I/O calls
  - *Hacking*: give a program an illusion to run in a different context
  - *Spying*: oops
- Library call interpositioning can be done at compile-time, link-time and load-time.

Intercepting library calls can be a very powerful debugging and testing tool. But interpositioning can also be a dangerous tool if used for malicious purposes.

Compile-time Interpositioning

- Change symbols at compile time so that library calls can be intercepted
- Typically done in C using `#define` pre-processor substitutions, sometimes contained in special header files
- This technique is restricted to situations where source code is available
- Example:

```
#define malloc(size) dbg_malloc(size, __FILE__, __LINE__)
#define free(ptr) dbg_free(ptr, __FILE__, __LINE__)

void *dbg_malloc(size_t size, char *file, int line);
void dbg_free(void *ptr, char *file, int line);
```

Link-time Interpositioning

- Tell the linker to change the way symbols are matched
- The GNU linker supports the option `--wrap=symbol`, which causes references to `symbol` to be resolved to `__wrap_symbol` while the real symbol remains accessible as `__real_symbol`.
- The GNU compiler allows to pass linker options using the `-Wl` option.
- Example:

```
/* gcc -Wl,--wrap=malloc -Wl,--wrap=free */
void * __wrap_malloc (size_t c)
{
    printf("malloc called with %zu\n", c);
    return __real_malloc (c);
}
```

Load-time Interpositioning

- The dynamic linker can be used to pre-load shared libraries
- This may be controlled by setting the `LD_PRELOAD` (Linux) or `DYLD_INSERT_LIBRARIES` (MacOS) environment variable
- Example:

  ```
  LD_PRELOAD=./libmymalloc.so vim
  ```


Load-time interpositioning can be a very powerful tool since it can be applied to any executable that uses shared libraries. The example shown in Listing 26 can be used to make a program believe it is executing at a different point in time.

Linux developers can use a program called `fakeroot` that runs a command in an environment where the command believes to have root privileges for file manipulation. This can be used to construct archives with proper file ownerships without having to work with a root account.

```c
/*
 * datehack/datehack.c --
 *
 * Build using cmake. Use as follows:
 *
 * LD_PRELOAD=./build/datehack.so date                                    (Linux)
 * DYLD_INSERT_LIBRARIES=./build/libdatehack.dylib date                   (MacOS)
 *
 * See fakeroot <http://freecode.com/projects/fakeroot> for a project
 * making use of LD_PRELOAD for good reasons.
 *
 * http://hackerboss.com/overriding-system-functions-for-fun-and-profit/
 */

#define _GNU_SOURCE
#include <time.h>
#include <dlfcn.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/types.h>
#include <stdio.h>

#define TIME_OFFSET (1 * 24 * 60 * 60 )

static struct tm *(*orig_localtime)(const time_t *timep);
static int (*orig_clock_gettime)(clockid_t clk_id, struct timespec *tp);

struct tm *localtime(const time_t *timep)
{
    time_t t = *timep - TIME_OFFSET;
    return orig_localtime(&t);
}

int clock_gettime(clockid_t clk_id, struct timespec *tp)
{
    int rc = orig_clock_gettime(clk_id, tp);
    if (tp) {
        tp->tv_sec -= TIME_OFFSET;
    }
    return rc;
}

__attribute__ ((constructor))
static void _init(void)
{
#pragma GCC diagnostic push
#pragma GCC diagnostic ignored "-Wpedantic"
    orig_localtime = dlsym(RTLD_NEXT, "localtime");
    if (! orig_localtime) {
        abort();
    }

    orig_clock_gettime = dlsym(RTLD_NEXT, "clock_gettime");
    if (! orig_clock_gettime) {
        abort();
    }
#pragma GCC diagnostic pop
}
```

Listing 26: Load-time library call interpositioning example

Part VIII

Memory

Every process needs memory to store machine instructions and to store data. The operating system kernel is in charge to assign memory to processes. Since main memory is finite, the operating system kernel needs to handle competing requests such that good performance can be achieved while establishing some degree of fairness.

Memory sizes have grown significantly over the last couple years and compared to 20 years ago, we have plenty of memory at our disposal today. But our applications are also consuming more memory and hence memory management has still a great impact on the overall performance of a system.

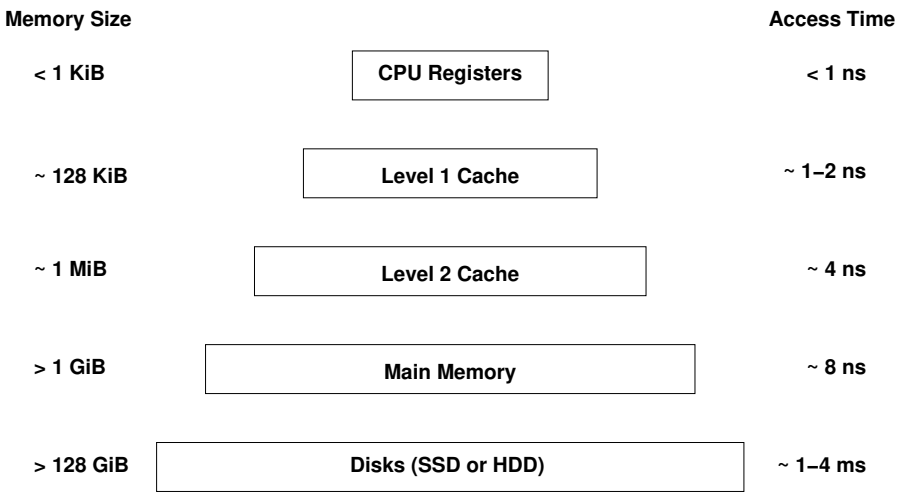
As we will see towards the end of this part, memory management and CPU scheduling can become most effective if they work hand in hand.

By the end of this part, students should be able to

- articulate the concept of logical and physical address spaces;
- sketch the difference between internal and external fragmentation;
- explain segmentation and relevant positioning strategies;
- apply positioning strategies to concrete situations;
- demonstrate how a buddy system allocator works;
- explain paging and and relevant loading and replacement strategies;
- apply replacement strategies to concrete situations;
- describe how an operating system kernel handles page faults;
- motivate the class of stack algorithms avoiding Belady's anomaly;
- understand the working-set model and how it can be applied to avoid thrashing;
- outline how process images are loaded by mapping various memory segments.

Section 25: Translation of Memory Addresses

Memory Sizes and Access Times



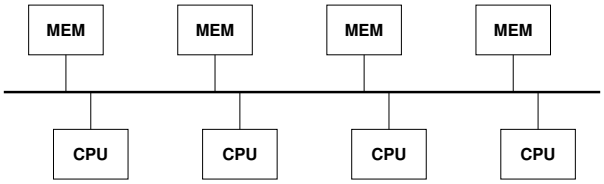
In the following, we will focus on the main memory. Organizing the usage of the available physical memory is a key function of an operating system. A mismatch of the memory management strategies used by an operating system kernel with the assumptions made by the developer of an application can slow down applications significantly.

In the following we focus on systems that run full-featured CPUs (desktops, notebooks, tablets, server, ...) and that have memory management units. The techniques we discuss may not be applicable to embedded systems using microcontrollers that have no hardware support for memory management (and memory protection).

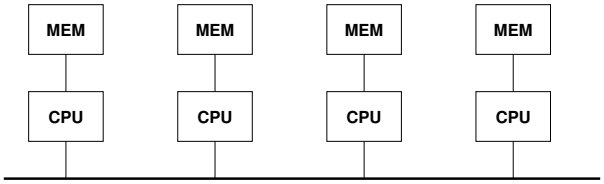
Main Memory

- Properties:
  - An ordered set of words or bytes
  - Each word or byte is accessible via a unique address
  - CPUs and I/O devices access the main memory
  - Running programs are (at least partially) loaded into main memory
  - CPUs usually can only access data in main memory directly (everything goes through main memory)
- Memory management of an operating system
  - allocates and releases memory regions
  - decides which process is loaded into main memory
  - controls and supervises main memory usage

Our model of a computer systems is a bit simplified since we assume that all CPUs have access to the entire memory in the same way.

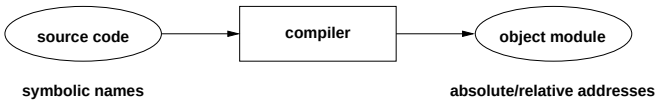


This is called the unified memory access (UMA) model of a computer. The UMA model model naturally leads to a high degree of contention on the memory system.

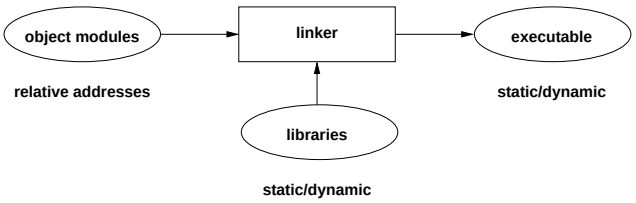


To reduce contention, engineers have invented non-uniform memory models (NUMA), where every CPU can access only a part of the memory directly and it has to interact with some other CPU to access memory directly accessible to that CPU. A big challenge is to organize cache coherence in NUMA architectures. An operating system should be aware of the internal structure of a NUMA system since it is desirable to place code and data into the memory that is directly accessible by the CPU a process is running on. A NUMA architecture also implies that a process should stay on the same CPU in order to execute efficiently. This is called the CPU affinity of a process.

Translation of Memory Addresses



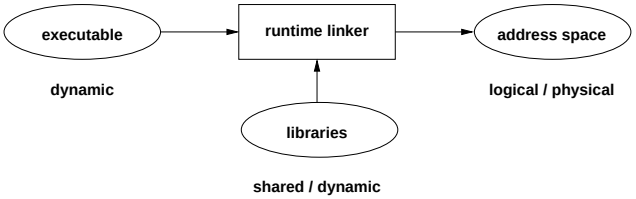
- A *compiler* translates variable / function names into absolute or relative addresses



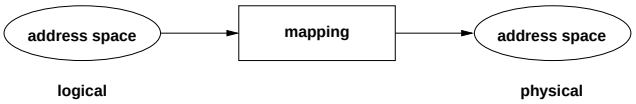
- A *linker* binds multiple object modules and libraries into an executable

As programmers, we like to give variables and functions meaningful names since this helps us to understand our source code. We leave it to compilers, linkers, and the operating system to resolve our names to memory addresses.

Translation of Memory Addresses



- A *runtime linker* loads dynamic (shared) libraries at program startup time

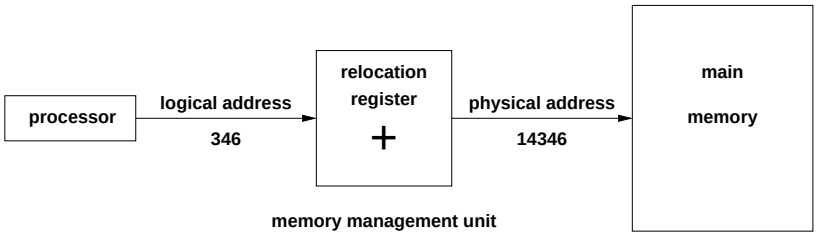


- A *memory management unit* maps logical addresses to physical addresses

Most of the advanced computing systems have a memory management unit that maps logical address spaces into a physical address space and that provides some level of memory protection. Modern systems often randomize memory space mappings such that the logical memory layout of a program changes on every invocation. This randomization makes it more difficult to write attacks that exploit vulnerabilities of a program.

Memory Management Tasks

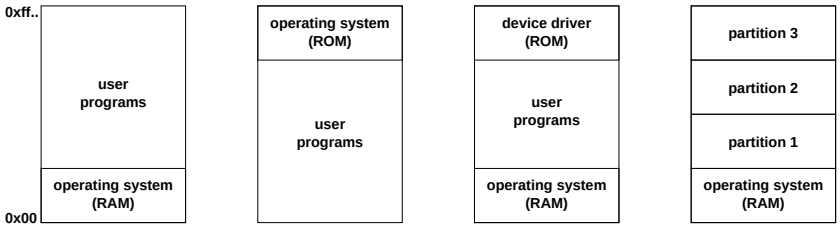
- Dynamic memory allocation for processes
- Creation and maintenance of memory regions shared by multiple processes (shared memory)
- Protection against erroneous / unauthorized access (isolation)
- Mapping of logical addresses to physical addresses



The simplest form of a memory management unit is using a relocation register to add a value to logical addresses in order to obtain corresponding physical addresses.

Since the memory management unit needs to do its work at the speed in which the CPU can access memory, the complexity of the mapping is limited by the timinig constraints that must be met.

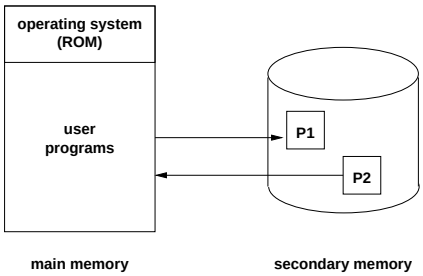
Memory Partitioning



- Memory space is often divided into several regions or partitions, some of them serve special purposes
- Multiple processes can be held in memory (as long as they fit)
- Partitioning is not very flexible (but may be good enough for embedded systems)

Memory partitioning can be easily implemented using relocation registers. In order to give running processes separate logical address spaces, the relocation register is updated on every context switch.

Swapping Principle

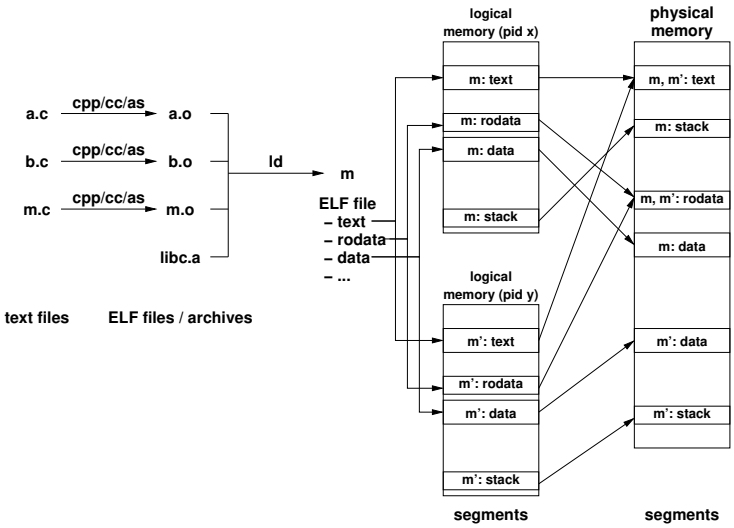


- Address space of a process is moved to a big (but slow) secondary storage system
- Swapped-out processes should not be considered runable by the scheduler
- Often used to handle (temporary) memory shortages

Swapping significantly slows down the execution of processes since data must be copied to a slow secondary storage devices and later the data needs to be read back. Swapping has a high price in particular if memory segments are large.

Section 26: Segmentation

Segmentation Overview



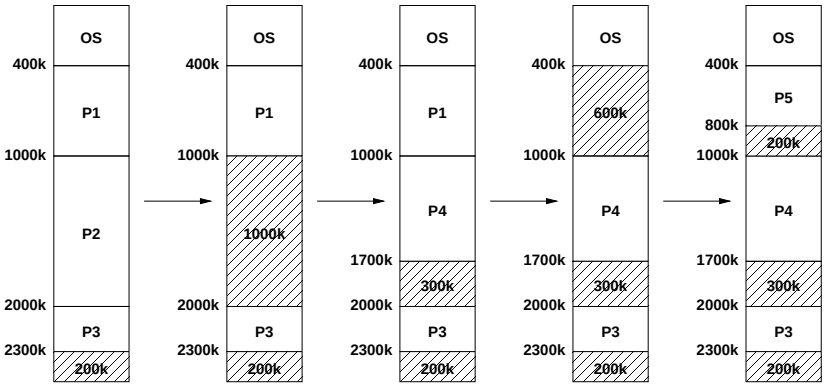
The slide shows a program `m` that was compiled from the source files `a.c`, `b.c`, and `m.c`. The linker (ld) links the object files `a.o`, `b.o`, and `m.o` and the C library `libc.a` into the executable file `m`. The exectable file, usually stored in ELF format, includes segments for the machine code (the text segment), a read-only data segment (the rodata segment), and the writeable data segment (data). When the program is executed twice, we obtain two processes, each with its own logical address space. (The `exec()` family of system calls is responsible for loading the segments from the ELF file into memory.)

A memory management system using segmentation will map these segments plus additional ones like the stack segment into physical memory. Note that all segments have different lengths. Furthermore, read-only segments can easily be shared (text segments are typically read-only).

Segmentation

- Main memory is partitioned by the operating system into memory segments of variable length
  - Different segments can have different access rights
  - Segments may be shared between processes
  - Segments may grow or shrink
  - Applications may choose to only hold the currently required segments in memory (sometimes called overlays)
- Addition and removal of segments will over time lead to small unusable holes (external fragmentation)
- Positioning strategy for new segments influences efficiency and longer term behavior

External Fragmentation



- In the general case, there is more than one suitable hole to hold a new segment — which one to choose?

Positioning Strategies ({best, worst, first, next} fit)

- *best fit*:
  - Allocate the smallest hole that is big enough
  - Large holes remain intact, many small holes
- *worst fit*:
  - Allocate the largest hole
  - Holes tend to become equal in size
- *first fit*:
  - Allocate the first hole from the top that is big enough
  - Simple and relatively efficient due to limited search
- *next fit*:
  - Allocate the next big enough hole from where the previous next fit search ended
  - Hole sizes are more evenly distributed

The positioning strategies best-fit, worst-fit, first-fit and next-fit have different strengths and weaknesses. It is difficult to say which one is better than another one since it depends on the workload and the properties of the resulting memory allocation requests.

Positioning Strategies (buddy system)

- Segments and holes always have a size of 2^i bytes (internal fragmentation)
- Holes are maintained in k lists such that holes of size 2^i are maintained in list i
- Holes in list i can be efficiently merged to a hole of size 2^{i+1} managed by list $i+1$
- Holes in list i can be efficiently split into two holes of size 2^{i-1} managed by list $i-1$
- Buddy systems are fast because only small lists have to be searched
- Internal fragmentation can be costly
- Sometimes used by user-space memory allocators (`malloc()`)

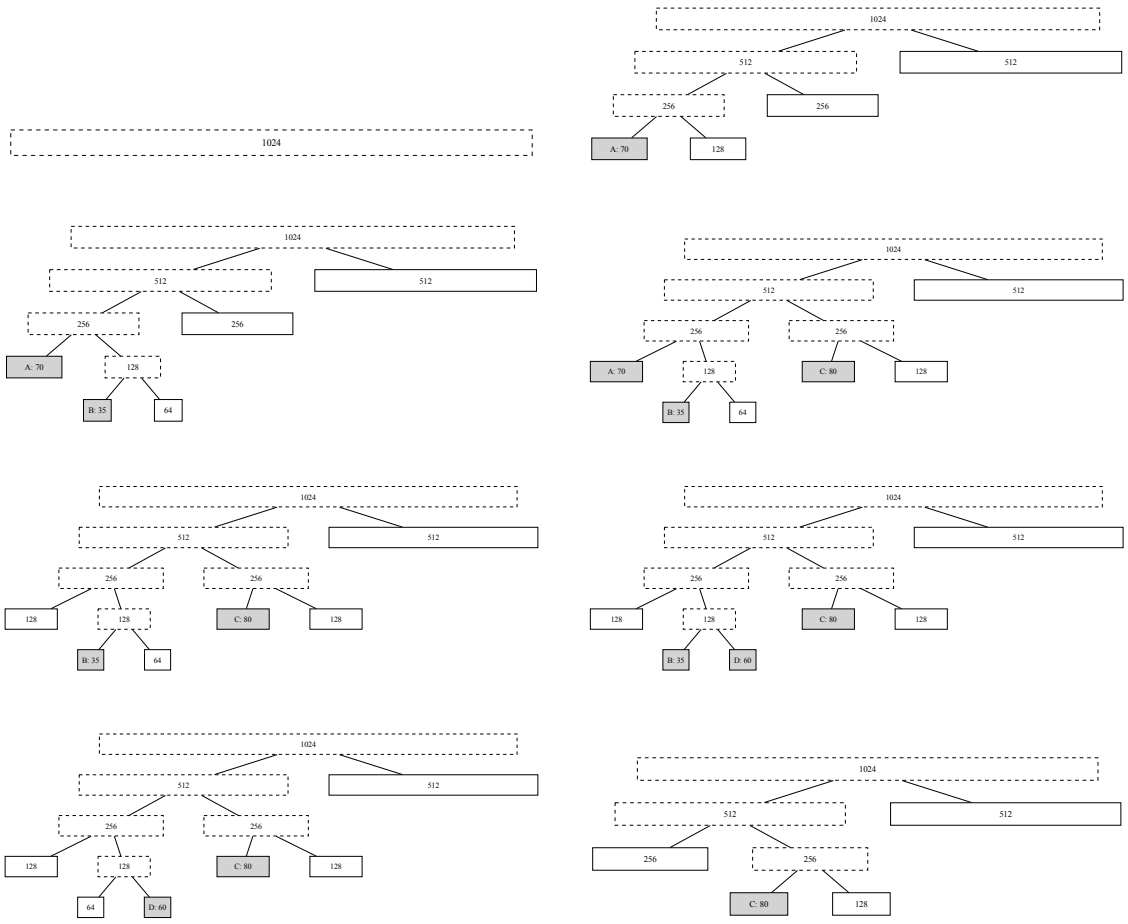
The buddy-system moves away from allocating the requested segment sizes since it always rounds up the next power of two. This leads to internal fragmentation. Whether the internal fragmentation is lost or not depends on the purpose and properties of a segment. A read-only segment is likely fixed in size and hence internal fragmentation may be lost memory. A read-write segment used as a heap may be bigger than initially requested but the additional memory may end up being used later during the execution of a process.

Buddy System Example

- Consider the processes A, B, C and D with the memory requests $70k$, $35k$, $80k$ and $60k$:

	1024					
A →	A	128		256		512
B →	A	B	64	256		512
C →	A	B	64	C	128	512
A ←	128	B	64	C	128	512
D →	128	B	D	C	128	512
B ←	128	64	D	C	128	512
D ←	256			C	128	512
C ←	1024					

The buddy system internally manages a binary tree structure. When allocated memory is released, then free chunks are joined together according to the tree structure. Here is how the underlying tree evolves in the example shown on the slide.



Slap Memory Allocator

- Many allocations are small, short-lived and frequent
- Performance can be improved by keeping allocations around and to reuse them whenever possible
- Support the allocation and caching of initialized objects
- Delay the deinitialization until a memory object is finally dropped
- Idea: Generalize the idea of caching allocated memory objects
- The Linux kernel got a SLAP allocator in the mid 1990s and later improved versions for non-unified memory access (NUMA) systems

Many allocations are short-lived and of similar sizes. A common technique to improve efficiency is to cache allocated memory objects on free lists and to reuse them without returning memory to the memory allocator. Such object caches are often maintained inside of libraries or specific kernel modules. Jeff Bonwick [5] proposed to generalize this by providing a general interface that can be used to instantiate object caches. His proposal is called a *slab allocator* and it manages caches of memory objects that may hold initialized memory objects.

A slab allocator can work well in combination with a buddy system allocator where the buddy system allocator is used to allocate memory space for the caches and the slab allocator is used to manage object allocations from the caches.

Segmentation Analysis

- *fifty percent rule*:
  Let n be the number of segments and h the number of holes. For large n and h and a system in equilibrium:
  $$h \approx \frac{n}{2}$$
- *unused memory rule*:
  Let s be the average segment size and ks the average hole size for some $k > 0$. With a total memory of m bytes, the fraction f of memory occupied by holes is:
  $$f = \frac{k}{k+2}$$

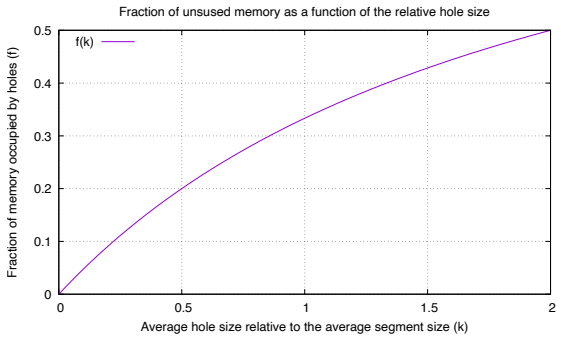
The memory m holds n segments of average size s. It follows that the unused memory is given by $m - ns$. If we assume that the average hole size is ks for some $k > 0$ and we have $h = \frac{n}{2}$ holes in equilibrium, then the unused memory is also given by $\frac{n}{2}ks$. This leads to:

$$\begin{aligned} m - ns &= \frac{n}{2}ks \\ m &= \frac{n}{2}ks + ns \\ &= ns(\frac{k}{2} + 1) \end{aligned}$$

The fraction f of unused memory is given by:

$$\begin{aligned} f &= \frac{\frac{n}{2}ks}{m} \\ &= \frac{\frac{n}{2}ks}{ns(\frac{k}{2}+1)} \\ &= \frac{\frac{k}{2}}{\frac{k}{2}+1} \\ &= \frac{k}{k+2} \end{aligned}$$

Segmentation Analysis

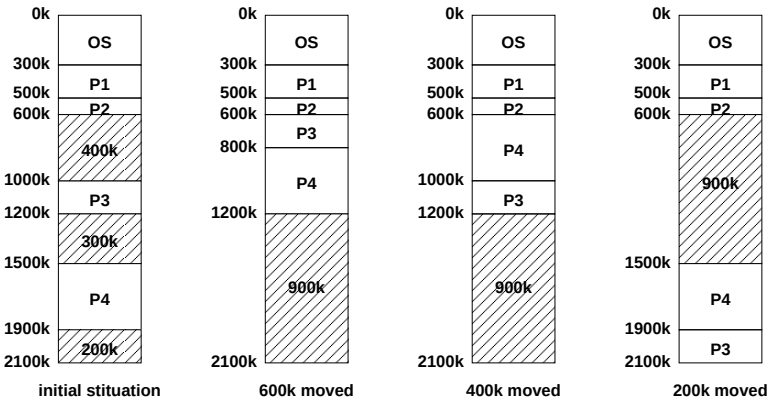


$\Longrightarrow$ As long as the average hole size is a significant fraction of the average segment size, a substantial amount of memory will be wasted

If the holes are on average half of the size of an average segment ($k = 0.5$), then 20% of the memory will not be used. This is a pretty bad result.

Compaction

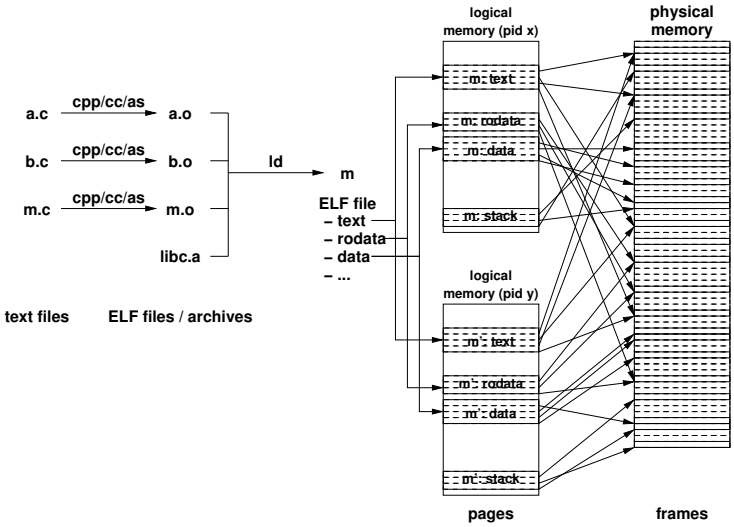
- Moving segments in memory allows to turn small holes into larger holes (and is usually quite expensive)
- Finding a good compaction strategy is not easy



If a memory allocation system is suffering from too many small external fragments, one could come up with the idea to compact memory again by moving memory segments around in memory. Relocating memory is of course expensive and the costs depend on how memory is moved. Finding a good compaction strategy that minimizes the memory moved can be tricky.

Section 27: Paging

Paging Overview



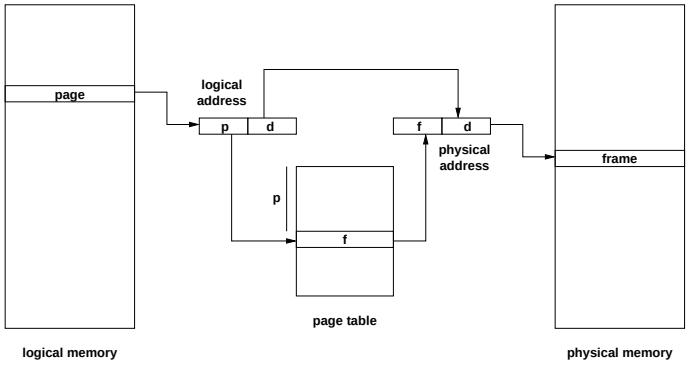
The slide shows a program `m` that was compiled from the source files `a.c`, `b.c`, and `m.c`. The linker (ld) links the object files `a.o`, `b.o`, and `m.o` and the C library `libc.a` into the executable file `m`. The exectable file, usually stored in ELF format, includes segments for the machine code (the text segment), a read-only data segment (the rodata segment), and the writeable data segment (data). When the program is executed twice, we obtain two processes, each with its own logical address space. (The `exec()` family of system calls is responsible for loading the segments from the ELF file into memory.)

A memory management system using paging will slice the segments into fixed-sized pages and load them into frames, that have been created by partitioning the physical memory into fixed-sizes frames. Since all pages and frames have the same sizes, the mapping can be very flexible. The pages of read-only segments can easily be shared (text segments are typically read-only). Paging offers a number of advantages. For example, not all pages of a logical memory segment have to be loaded at the same time and segment sizes can easily be extended by adding additional pages. The downside of paging systems is that the mapping is much more complex and hence the hardware needed to achieve memory mappings at CPU speed is more complex.

Paging Idea

- General Idea:
  - Physical memory is organized in frames of fixed size
  - Logical memory is organized in pages of the same fixed size
  - Page numbers are mapped to frame numbers using a (very fast) page table mapping mechanism
  - Pages of a logical address space can be scattered over the physical memory
- Motivation:
  - Avoid external fragmentation and compaction
  - Allow fixed size pages to be moved into / out of physical memory

Paging Model and Hardware



- A logical address is a tuple (p, d) where p is an index into the page table and d is an offset within page p
- A physical address is a tuple (f, d) where f is the frame number and d is an offset within frame f

Paging Properties

- Address translation must be very fast (in some cases, multiple translations are necessary for a single machine instruction)
- Page tables can become quite large (a 32 bit address space with a page size of 4096 bytes requires a page table with 1 million entries)
- Additional information in the page table:
  - Protection bits (read/write/execute)
  - Dirty bit (set if page was modified)
- Not all pages of a logical address space must be resident in physical memory to execute the process
- Access to pages not in physical memory causes a page fault which must be handled by the operating system

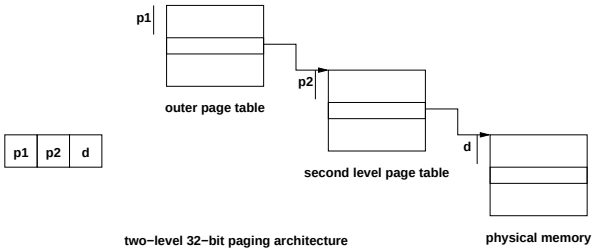
Handling Page Faults

1. MMU detects a page fault and raises an interrupt
2. Operating system saves the registers of the process
3. Mark the process blocked (waiting for page)
4. Determination of the address causing the page fault
5. Verify that the logical address usage is valid
6. Select a free frame (or a used frame if no free frame)
7. Write used frame to secondary storage (if modified)
8. Load page from secondary storage into the free frame
9. Update the page table in the MMU
10. Restore the instruction pointer and the registers
11. Mark the process runnable and call the scheduler

Paging Characteristics

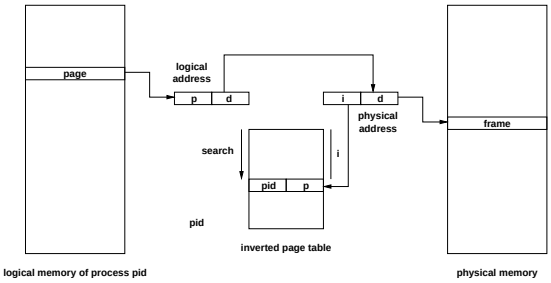
- Limited internal fragmentation (last page)
- Page faults are costly due to slow I/O operations
- Try to ensure that the "essential" pages of a process are always in memory
- Try to select used frames (victims) which will not be used in the future
- During page faults, other processes can execute
- What happens if the other processes also cause page faults?
- In the extreme case, the system is busy swapping pages into memory and does not do any other useful work (thrashing)

Multilevel Paging



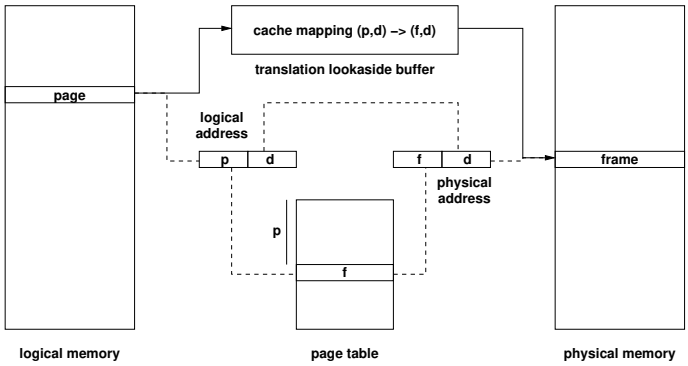
- Paging can be applied to page tables as well
- SPARC 32-bit architecture supports three-level paging
- Motorola 32-bit architecture (68030) supports four-level paging
- Caching essential to alleviate delays introduced by multiple memory lookups

Inverted Page Tables



- The inverted page table has one entry for each frame
- Page table size determined by size of physical memory
- Entries contain page address and process identification
- The non-inverted page table is stored in paged memory
- Lookups require to search the inverted page table

Translation Lookaside Buffers (TLBs)



- A TLB acts as a cache mapping logical addresses (p, d) to physical addresses (f, d)
- TLB lookup failures may be handled by the kernel in software

A Translation Lookups Buffer (TLB) acts like a cache for page table lookups. A page number extracted from a logical address is given to the TLB and it returns either the associated frame number (a TLB hit) or it signals a TLB miss to the operating system.

The operating system (often in software) looks up the missing TLB entry, loads it into the TLB, and restarts the instruction. If the given address is not valid, then an error is signaled to the running process.

Combined Segmentation and Paging

- Segmentation and paging have different strengths and weaknesses
- Combined segmentation and paging allows to take advantage of the different strengths
- Some architectures supported paged segments or even paged segment tables
- MMUs supporting segmentation and paging leave it to the operating systems designer to decide which strategy is used
- Note that fancy memory management schemes do not work for real-time systems...

Section 28: Virtual Memory

Virtual Memory

- Virtual memory is a technique that allows the execution of processes that may not fit completely in memory
- Motivation:
  - Support virtual address spaces much larger than the physical address space
  - Programmers are less bound by memory constraints
  - Only small portions of an address space are typically used at runtime
  - More programs can be in memory if only the essential data resides in memory
  - Faster context switches if resident data is small
- Most virtual memory systems are based on paging, but virtual memory systems based on segmentation are feasible

Loading Strategies

- Loading strategies determine when pages are loaded into memory:
  - *swapping*:
    Load complete address spaces (does not work for virtual memory)
  - *demand paging*:
    Load pages when they are accessed the first time
  - *pre-paging*:
    Load pages likely to be accessed in the future
  - *page clustering*:
    Load larger clusters of pages to optimize I/O
- Most systems use demand paging, sometimes combined with pre-paging

Replacement Strategies

- Replacement strategies determine which pages are moved to secondary storage in order to free frames
  - Local strategies assign a fixed number of frames to a process (page faults only affect the process itself)
  - Global strategies assign frames dynamically to all processes (page faults may affect other processes)
- Paging can be described using reference strings:

$w = r[1]r[2]\ldots r[t]\ldots$	sequence of page accesses
$r[t]$	page accessed at time t
$s = s[0]s[1]\ldots s[t]\ldots$	sequence of resident pages
$s[t]$	set of pages resident at time t
$x[t]$	set of pages paged-in at time t
$y[t]$	set of pages paged-out at time t

Replacement Strategies

- *First in first out (FIFO)*:
  Replace the page which is the longest time in memory
- *Second chance (SC)*:
  Like FIFO but skip pages that have been used since the last page fault
- *Least frequently used (LFU)*:
  Replace the page which has been used least frequently
- *Least recently used (LRU)*:
  Replace the page which has not been used for the longest period of time
- *Belady's optimal algorithm (BO)*:
  Replace the page which will not be used for the longest period of time

Belady's algorithm requires to look into the future and hence it is of mostly theoretical value. It can be shown that Belady's algorithm minimizes page faults by postponing them as much as possible into the future.

LRU is often used to approximate Belady's algorithm. If the past behaviour of a program is a good approximation of its behaviour in the future, then LRU should provide good results.

A problem of LRU is that it is costly to implement. Hardware to keep track of a complete reference string would be very expensive. Hence, LFU may be used as an approximation of LRU since counting page accesses is much more viable than keeping track of a reference string.

FIFO is a relatively simple to implement algorithm but not very smart in keeping pages frequently needed in memory. The second chance algorithm improves that by taking some information about recent page accesses into account. The motiviation here is that recently accessed pages are likely also accessed in the near future.

Belady's Anomaly (FIFO Replacement Strategy)

string	1	2	3	4	1	2	5	1	2	3	4	5
frame 0	1	1	1	4	4	4	5	5	5	5	5	5
frame 1		2	2	2	1	1	1	1	1	3	3	3
frame 2			3	3	3	2	2	2	2	2	4	4
faults	x	x	x	x	x	x	x			x	x	

string	1	2	3	4	1	2	5	1	2	3	4	5
frame 0	1	1	1	1	1	1	5	5	5	5	4	4
frame 1		2	2	2	2	2	2	1	1	1	1	5
frame 2			3	3	3	3	3	3	2	2	2	2
frame 3				4	4	4	4	4	4	3	3	3
faults	x	x	x	x			x	x	x	x	x	x

- For the same reference string $w = 1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5$
  - FIFO with $m = 3$ frames leads to 9 page faults
  - FIFO with $m = 4$ frames leads to 10 page faults
- Belady's anomaly: Increasing memory may lead to an increase of page faults for certain page replacement strategies.

Increasing physical memory size should never increase the page fault rate. If you buy more memory, you expect things to become faster, not slower. However, Belady discovered that this is not always the case [2].

Stack Algorithms

- Every reference string w can be associated with a sequence of stacks such that the pages in memory are represented by the first m elements of the stack
- A stack algorithm is a replacement algorithm with the following properties:
  1. The last used page is on the top
  2. Pages which are not used never move up
  3. Pages below the used page do not move
- Let $S_m(w)$ be the memory state reached by the reference string w and the memory size m. Then for every stack algorithm, the following holds:

$$S_m(w) \subseteq S_{m+1}(w)$$

- Replacement strategies that are stack algorithm do not suffer from Belady's anomaly

LRU Algorithm

- LRU is a stack algorithm (while FIFO is not)
- LRU with counters:
  - CPU increments a counter for every memory access
  - Page table entries have a counter that is updated with the CPU's counter on every memory access
  - The page with the smallest counter is the LRU page
- LRU with a stack:
  - Keep a stack of page numbers
  - Whenever a page is used, move its page number on the top of the stack
  - The page number at the bottom identifies LRU page
- Exact algorithms are generally difficult to implement at CPU/MMU speed

Memory Management and Scheduling

- Interaction of memory management and scheduling:
  - Processes should not get the CPU if the probability for page faults is high
  - Processes should not remain in main memory if they are waiting for an event which is unlikely to occur in the near future
- How to estimate the probability of future page faults?
- Does the approach work for all programs equally well?
- Fairness?

Memory management aspects are difficult to analyze on real operating systems. On GNU/Linux systems, the GNU `time` command can provide information about the resource usage of processes.

```
$ /usr/bin/time -v date
Tue Oct 13 23:33:03 CEST 2020
        Command being timed: "date"
        User time (seconds): 0.00
        System time (seconds): 0.00
        Percent of CPU this job got: 66%
        Elapsed (wall clock) time (h:mm:ss or m:ss): 0:00.00
        Average shared text size (kbytes): 0
        Average unshared data size (kbytes): 0
        Average stack size (kbytes): 0
        Average total size (kbytes): 0
        Maximum resident set size (kbytes): 2076
        Average resident set size (kbytes): 0
        Major (requiring I/O) page faults: 0
        Minor (reclaiming a frame) page faults: 89
        Voluntary context switches: 1
        Involuntary context switches: 1
        Swaps: 0
        File system inputs: 0
        File system outputs: 0
        Socket messages sent: 0
        Socket messages received: 0
        Signals delivered: 0
        Page size (bytes): 4096
        Exit status: 0
```

The Linux kernel distinguishes between minor and major page faults. A major page fault causes some I/O to a storage device while a minor page fault does not. The output shown above indicates 89 minor page faults and 0 major page faults for running the `date` program. The page size used by the system (Linux 4.19.0-8-amd64, Debian 4.19.98-1+deb10u1) is 4096 bytes.

Locality

- Locality describes the property of programs to use only a small subset of the memory pages during a certain part of the computation
- Programs are typically composed of several localities, which may overlap
- Reasons for locality:
  - Structured and object-oriented programming (functions, small loops, local variables)
  - Recursive programming (functional / declarative programs)
- Some applications (e.g., data bases or mathematical software handling large matrices) show only limited locality

Working-Set Model

- The *Working-Set* $W_p(t, T)$ of a process p at time t with parameter T is the set of pages which were accessed in the time interval $[t - T, t)$
- A memory management system follows the working-set model if the following conditions are satisfied:
  - Processes are only marked runnable if their full working-set is in main memory
  - Pages belonging to the working-set of a running process are not removed from memory
- Example ($T = 10$):

$$w = \ldots \underline{2, 6, 1, 5, 7, 7, 7, 7, 5, 1}, 6, 2, 3, 4, 1, 2, \underline{3, 4, 4, 4, 3, 4, 3, 4, 4, 4}, 1, 3, 2, 3, 4, 3, \ldots$$

$$W_p(t_1, 10) = \{1, 2, 5, 6, 7\} \qquad W_p(t_2, 10) = \{3, 4\}$$

The working set model was introduced by Peter J. Denning in 1968 [9].

Working-Set Properties

- The performance of the working-set model depends on the parameter T:
  - If T is too small, many page faults are possible and thrashing can occur
  - If T is too big, unused pages might stay in memory and other processes might be prevented from becoming runnable
- Determination of the working-set:
  - Mark page table entries whenever they are used
  - Periodically read and reset these marker bits to estimate the working-set
- Adaptation of the parameter T:
  - Increase / decrease T depending on page fault rate

POSIX API (mmap, munmap, msync, mlock, munlock)

```
#include <sys/mman.h>

#define PROT_EXEC  ...    /* memory is executable */
#define PROT_READ  ...    /* memory is readable */
#define PROT_WRITE ...    /* memory is writable */
#define PROT_NONE  ...    /* no access */

#define MAP_SHARED    ... /* memory may be shared between processes */
#define MAP_PRIVATE   ... /* memory is private to the process */
#define MAP_ANONYMOUS ... /* memory is not tied to a file descriptor */

void* mmap(void *start, size_t length, int prot, int flags, int fd, off_t offset);
int munmap(void *start, size_t length);

int msync(void *start, size_t length, int flags);
int mprotect(const void *addr, size_t len, int prot);

int mlock(const void *addr, size_t len);
int munlock(const void *addr, size_t len);
```


The memory mapping API enables us to allocate larger chunks of memory. There are several different mappings:

- Anonymous mappings are plain memory mappings. The mapping is by default not accessible to other processes.
- Mappings can be marked as shared. A shared mapping makes updates visible to other processes mapping the same region (most likely at different logical addresses).
- Mappings can be backed up by a file. Programs can use this to read file content by mapping (portions of) a file into memory. It is also possible to make changes and to synchronize the changes back to the underlying file.
- Locking memory mappings tells the kernel to keep pages resident in physical memory.
- Shared memory mappings require synchronization primitives that are shared between processes.

Memory mappings have associated protection bits indicating whether mapped memory content can be read, written, or executed. Listing 27 demonstrates anonymous memory mappings.

```c
/*
 * memmap/mapit.c --
 *
 *      The mapit() function allocates memory using mmap().
 */

#define _POSIX_C_SOURCE 200809L
#define _DEFAULT_SOURCE
#define _DARWIN_C_SOURCE

#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/mman.h>
#include "memmap.h"

void mapit(int flags)
{
    unsigned char *p;
    long pagesize, length;

    pagesize = sysconf(_SC_PAGE_SIZE);
    if (pagesize == -1) {
        perror("sysconf");
        exit(EXIT_FAILURE);
    }

    length = 10 * pagesize;                     /* 10 pages - why 10? */
    pmap("after main() is called", flags);

    p = mmap(NULL, length, PROT_READ | PROT_WRITE,
             MAP_ANONYMOUS | MAP_PRIVATE, -1, 0);
    if (p == MAP_FAILED) {
        perror("mmap");
        exit(EXIT_FAILURE);
    }
    pmap("after calling mmap()", flags);

    if (flags & MEMMAP_L_FLAG) {                /* lock the pages */
        if (mlock(p, length) == -1) {
            perror("mlock");
            exit(EXIT_FAILURE);
        }
        pmap("after calling mlock()", flags);
    }

    p[0] = 0xff;                                /* dirty the first page */
    pmap("after writing p[0]", flags);

    p[length-1] = 0xff;                         /* dirty the last page */
    pmap("after writing p[length-1]", flags);

    if (munmap(p, length) == -1) {
        perror("munmap");
        exit(EXIT_FAILURE);
    }
    pmap("after calling munmap()", flags);
}

```

Listing 27: Demonstration of anonymous memory mappings

Part IX

Communication

Communication between processes is essential in order to break problems into smaller pieces and to enable distributed computing. Operating system kernels provide different inter-process communication primitives, ranging from very primitive and basic ones to highly general ones that enable processes to communicate over the global Internet with other processes. This part, we will first look at basic signals that can be used to signal exceptional conditions. We then look at pipes, a popular communication mechanism between processes running on the same kernel. Finally, we look at the sockets that enable local as well as global communication.

By the end of this part, students should be able to

- outline how signals interrupt the normal control flow;
- explain the difference between C and POSIX signal handling APIs;
- write programs that react to signals properly;
- describe how pipes enable data flows between processes;
- illustrate how file descriptor tables modifications can impact the standard input and output;
- use sockets as generic communication endpoints;
- differentiate socket types and families of socket addresses;
- write programs using connection-less and connection-oriented communication;
- implement I/O multiplexing and event loops;
- distinguish between blocking and non-blocking I/O;
- describe the abstraction of asynchronous I/O.

Inter-Process Communication

- An operating system has to provide inter-process communication primitives in the form of system calls and APIs
- Signals:
  - Software equivalent of hardware interrupts
  - Signals interrupt the normal control flow, but they do not carry any data (except the signal number)
- Pipes:
  - Uni-directional channel between two processes
  - One process writes, the other process reads data
- Sockets:
  - General purpose communication endpoints
  - Multiple processes, global (Internet) communication

Section 29: Signals

Signals

- Signals are a very limited communication mechanism
- Signals are either *synchronous* or *asynchronous* to the program execution
- Basic signals are part of the standard C library
  - Signals for runtime exceptions (division by zero)
  - Signals created by external events
  - Signals explicitly created by the program itself
- POSIX signals are more general and powerful
  - Sending signals between processes
  - Better control of signal delivery (blocking signals)
  - Better control of signal handling behavior
- If in doubt, use the POSIX signal API to make code portable

As a C/C++ programmer, you are used to signals. If your program uses an invalid pointer, you likely get a segmentation fault, which is actually signalled by the kernel to your program via a `SIGSEGV` signal. You also know that you can interrupt an application running in your terminal by pressing CTRL-C, which lets the kernel deliver a `SIGINT` signal to your application. If you divide a number by zero, you may receive a `SIGFPE` signal.

Signals can interrupt the current control flow of a program. In such a situation, the CPU stops executing the regular sequence of machine instructions and it starts executing the machine instructions of the signal handler. Once the signal handler returns, the CPU continues executing the regular sequence of machine instructions. Since signals can in general happen at any point in time, it is important to control them to avoid race conditions.

C Library Signal API

```
#include <signal.h>

typedef ... sig_atomic_t;
typedef void (*sighandler_t)(int);

sighandler_t signal(int signum, sighandler_t handler);
int raise(int signum);

#define SIGABRT ...     /* abnormal  termination */
#define SIGFPE  ...     /* floating-point exception */
#define SIGILL  ...     /* illegal instruction */
#define SIGINT  ...     /* interactive interrupt */
#define SIGSEGV ...     /* segmentation violation */
#define SIGTERM ...     /* termination request */

#define SIG_IGN ...     /* handler to ignore the signal */
#define SIG_DFL ...     /* default handler for the signal */
#define SIG_ERR ...     /* handler returned on error situations */
```


The signal API of the C library is very limited and implementations behave differently. On some systems, signal handlers are deinstalled when a signal is delivered. On other systems, a signal handler remains in place until it is changed explicitly. Furthermore, the C library functions are limited to a single running program since the C library does not have a notion of processes. Hence, for most use cases, it is advisable to use the POSIX signal API.

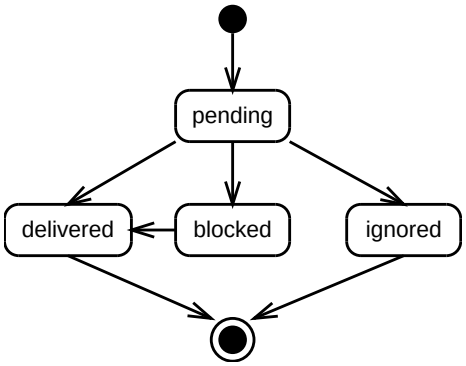
Listing 28 shows a simple version of `cat` that ignores `SIGINT` signals. Note that the signal handler is reinstalled whenever a signal has been received. Also note that signals can interrupt I/O system calls. To resolve this, the copy loop has been wrapped into another loop that clears any errors that are caused by interrupted I/O system calls.

```c
/*
 * signals/catc.c --
 *
 *        This program demonstrates the C library functions for handling
 *        signals. Note that signal handlers need to be reinstalled and
 *        that signals can interrupt I/O system calls. As a consequence,
 *        the main loop needs to handle I/O failures caused by signals.
 */

#include <stdlib.h>
#include <stdio.h>
#include <signal.h>
#include <unistd.h>
#include <errno.h>

static void sig_handler(int);

static void sig_install(void)
{
    if (signal(SIGINT, sig_handler) == SIG_ERR) {
        perror("signal");
        exit(EXIT_FAILURE);
    }
}

static void sig_handler(int signum)
{
    if (signum == SIGINT) {
        fprintf(stderr, "catc: sig_handler: Interrupt\n");
        sig_install();
    } else {
        fprintf(stderr, "catc: sig_handler: %d\n", signum);
    }
}

int main(void)
{
    char c;

    sig_install();
    while (! feof(stdin)) {
        while ((c = getc(stdin)) != EOF) {
            (void) putc(c, stdout);
        }
        if (ferror(stdin)) {
            if (errno == EINTR) {
                clearerr(stdin);
                continue;
            }
            break;
        }
    }
    if (ferror(stdin) || fflush(stdout) == EOF) {
        return EXIT_FAILURE;
    }
    return EXIT_SUCCESS;
}
```

Listing 28: Demonstration of the C library signals API

POSIX Signal Delivery

- Signals start in the state *pending* and are usually *delivered* to the process
- Signals can be *blocked* by processes
- Blocked signals are *delivered* when unblocked
- Signals can be ignored if they are not needed



The state machine model enables us to control the delivery of signals. Signals can be in the pending state if the kernel has not yet delivered them. Note that kernels may drop signals if a signal of the same type is still pending. The delivery of pending signals may be blocked. Blocked signals will be delivered when the blocking reason disappears. Signals can also be ignored, in which case the kernel does not interrupt the control flow of processes. There are some signals (like `SIGKILL`) that cannot be ignored.

POSIX Signal API

```
#include <signal.h>

typedef void (*sighandler_t)(int);
typedef ... sigset_t;
typedef ... siginfo_t;

#define SIG_DFL ...             /* default handler for the signal */
#define SIG_IGN ...             /* handler to ignore the signal */

#define SA_NOCLDSTOP ...        /* do not create SIGCHLD signals when a child is stopped */
#define SA_NOCLDWAIT ...        /* do not create SIGCHLD signals when a child terminates */
#define SA_ONSTACK   ...        /* use an alternative stack */
#define SA_RESTART   ...        /* restart interrupted system calls */

struct sigaction {
    sighandler_t sa_handler;    /* handler function */
    void         (*sa_sigaction)(int, siginfo_t *, void *);  /* handler function */
    sigset_t     sa_mask;       /* signals to block while executing handler */
    int          sa_flags;      /* flags to control behavior */
};
```


Signal handlers usually use the stack of the interrupted process. This is sometimes not desirable. It is possible to setup a separate stack for signal handling and then the `SA_ONSTACK` flag requests that the alternate stack is used.

POSIX Signal API

```
int sigaction(int signum, const struct sigaction *action,
              struct sigaction *oldaction);
int kill(pid_t pid, int signum);

int sigemptyset(sigset_t *set);
int sigfillset(sigset_t *set);
int sigaddset(sigset_t *set, int signum);
int sigdelset(sigset_t *set, int signum);
int sigismember(const sigset_t *set, int signum);

#define SIG_BLOCK   ...
#define SIG_UNBLOCK ...
#define SIG_SETMASK ...

int sigprocmask(int how, const sigset_t *set, sigset_t *oldset);
int sigpending(sigset_t *set);
int sigsuspend(const sigset_t *mask);
```


The function `sigaction()` registers a function to be executed when a specific signal is being delivered. During the execution of a signal function, the triggering signal and any signals specified in the signal mask are blocked.

The function `kill()` sends a signal to a process or process group:

- If `pid > 0`, the signal is sent to process pid.
- If `pid == 0`, the signal is sent to every process in the process group of the current process
- If `pid == -1`, the signal is sent to every process except for process 1 (init)
- If `pid < -1`, the signal is sent to every process in the process group `-pid`

Listing 29 shows a simple version of `cat` that ignores `SIGINT` signals, this time written using the POSIX signal API. The implementation requests that I/O system calls are automatically restarted and the signal handler does not need to be reinstalled upon signal delivery.

```c
/*
 * signals/catp.c --
 *
 *        This program demonstrates the POSIX library functions for
 *        handling signals. The POSIX API allows programs much more
 *        control over the behaviour of signals and signal handlers. We
 *        request that system calls are restarted, which simplifies the
 *        main loop since we do not have to deal I/O call interrupts.
 */

#define _POSIX_C_SOURCE 200809L

#include <stdlib.h>
#include <stdio.h>
#include <signal.h>
#include <unistd.h>
#include <string.h>

static void sig_action(int signum, siginfo_t *siginfo, void *unused);

static void sig_install(void)
{
    struct sigaction sa;

    sa.sa_sigaction = sig_action;
    sigemptyset(&sa.sa_mask);
    sa.sa_flags = SA_SIGINFO | SA_RESTART;
    if (sigaction(SIGINT, &sa, NULL) == -1) {
        perror("sigaction");
        exit(EXIT_FAILURE);
    }
}

static void sig_action(int signum, siginfo_t *siginfo, void *unused)
{
    (void) unused;
    if (siginfo && siginfo->si_code <= 0) {
        fprintf(stderr, "catp: sig_action: %s (from pid %d, uid %d)\n",
                strsignal(signum), siginfo->si_pid, siginfo->si_uid);
    } else {
        fprintf(stderr, "catp: sig_action: %s\n", strsignal(signum));
    }
}

int main(void)
{
    char c;

    sig_install();
    while ((c = getc(stdin)) != EOF) {
        (void) putc(c, stdout);
    }
    if (ferror(stdin) || fflush(stdout) == EOF) {
        return EXIT_FAILURE;
    }
    return EXIT_SUCCESS;
}
```

Listing 29: Demonstration of POSIX library signals

Properties of POSIX Signals

- Implementations can merge multiple identical signals
- Signals can not be counted reliably
- Signals do not carry any data / information except the signal number
- Signal functions are typically very short since the real processing of the signaled event is usually deferred to a later point in time of the execution when the state of the program is known to be consistent
- Variables modified by signals should be volative and signal atomic
- `fork()` inherits signal functions, `exec()` resets signal functions (for security reasons and because the process gets a new memory image)
- Threads in general share the signal actions, but every thread may have its own signal mask

Child processes deliver a `SIGCHLD` signal to their parent process upon termination (but also when they are stopped or resumed). By default, the `SIGCHLD` signal is ignored. This requires, however, that the parent process picks up status codes returned by child processes using one of the `wait()` system calls. Terminated child processes turn into zombie processes if a parent process does not wait for child processes.

If a parent process explicitly sets the action of the `SIGCHLD` to `SIG_IGN`, then child processes do not turn into zombie processes when they terminate. This is a subtle difference between the default behaviour and explicitely requesting the default behaviour.

Listing 30 demonstrates that signals are truly asynchronous. Data accessed by a signal handler can be in an inconsistent state. Hence, it is strongly recommended to only update variables with a signal atomic data type from a signal handler.

```c
/*
 * signals/surprise.c --
 *
 *        Depending on your system, you might see that signals interrupt
 *        even simple assignments. The alarm() call arranges for a
 *        SIGALRM to be delivered to the calling process after a number
 *        of seconds. Compile without optimizations to see the effect.
 */

#define _POSIX_C_SOURCE 200809L

#include <stdlib.h>
#include <signal.h>
#include <stdint.h>
#include <stdio.h>
#include <inttypes.h>
#include <unistd.h>

typedef struct twins { int64_t a, b; } twins_t;
static const twins_t zeros = { 0, 0 }, ones = { 1, 1 };
static twins_t twins;
static volatile sig_atomic_t go_on = 1;

static void handler(int signum)
{
    if (signum == SIGALRM) {
        go_on = (twins.a == twins.b);
        if (go_on) {
            (void) alarm(1);                        /* request next alarm */
        } else {
            printf("twins = {%" PRId64 ",%" PRId64 "}\n", twins.a, twins.b);
        }
    }
}

int main(void)
{
    struct sigaction sa;
    sa.sa_handler = handler;
    sigemptyset(&sa.sa_mask);
    sa.sa_flags = SA_RESTART;
    if (sigaction(SIGALRM, &sa, NULL) == -1) {
        perror("sigaction");
        return EXIT_FAILURE;
    }

    (void) alarm(1);                               /* request first alarm */
    while (go_on) {
        twins = zeros;
        twins = ones;
    }

    printf("twins = {%" PRId64 ",%" PRId64 "}\n", twins.a, twins.b);
    return EXIT_SUCCESS;
}
```

Listing 30: Demonstration of signal generated data races

Signal Pattern: Flagging Behaviour Changes

```
#include <signal.h>

static volatile sig_atomic_t keep_going = 1;

static void
catch_signal(int signum)
{
    keep_going = 0;         /* defer the handling of the signal */
}

int
main(void)
{
    signal(SIGINT, catch_signal);
    while (keep_going) {
        /* ... do something ... */
    }
    /* ... cleanup ... */
    return 0;
}
```


Since signals such as `SIGINT` or `SIGTERM` arrive asynchronously, it is often not a good idea to take an action when the signal arrives. Instead, it is recommended to simply set a flag that is regularly checked in the program to change its behavior. This ensures that the program takes an action when the state of the program is consistent and well-defined.

Note that the flag variable should be of type `sig_atomic_t` to handle any possible race conditions. Furthermore, it is suggested to use the storage class `volatile` to tell the compiler that the variable may change in ways that the compiler cannot predict. This implies that the compiler has to read the variable's value from memory whenever it is needed and the compiler can't simply decide to hold the variable's value cached in a register or even to optimize the variable away.

Signal Pattern: Catching Terminating Signals

```
volatile sig_atomic_t fatal_error_in_progress = 0;

static void
fatal_error_signal(int signum)
{
    if (fatal_error_in_progress) {
        raise(signum);
        return;
    }

    fatal_error_in_progress = 1;
    /* ... cleanup ... */

    signal(signum, SIG_DFL);        /* install the default handler */
    raise(signum);                  /* and let it do its job */
}
```


A pattern for catching terminating signals such as `SIGINT` or `SIGTERM` is to install the default signal handler after the signal has been handled and to request that the signal is delivered again. This has the nice effect that the program will actually be terminated by the signal, which will then also be communicated as part of the status code to other processes waiting for the program to terminate.

The template code shown above assumes that it can be called by multiple terminating signals and this is why it only allows the first signal to execute the cleanup code.

Listing 31 demonstrates how the `sleep(3)` library function can be implemented using a timer signal.

```c
/*
 * sleep/sleep.c --
 *
 *        Implementation of sleep() using POSIX signal functions.
 */

#define _POSIX_C_SOURCE 2

#include <stdlib.h>
#include <signal.h>
#include <unistd.h>

static volatile sig_atomic_t wake_up = 0;

static void catch_alarm(int signum)
{
    (void) signum;              /* unused signum parameter */
    wake_up = 1;
}

unsigned int sleep(unsigned int seconds)
{
    struct sigaction sa, old_sa;
    sigset_t mask, old_mask;

    sa.sa_handler = catch_alarm;
    sigemptyset(&sa.sa_mask);
    sa.sa_flags = 0;

    /* Be nice and save the original signal handler so that it can be
     * restored when we are done. */
    sigaction(SIGALRM, &sa, &old_sa);

    /* After resetting wake_up, ask the system to send us a SIGALRM at
     * an appropriate time. */
    wake_up = 0;
    alarm(seconds);

    /* First block the signal SIGALRM. After safely checking wake_up,
     * suspend until a signal arrives. Note that sigsuspend may return
     * on other signals (according to the old mask). If wake_up is
     * finally true, cleanup by unblocking the blocked signals. */
    sigemptyset(&mask);
    sigaddset(&mask, SIGALRM);
    sigprocmask(SIG_BLOCK, &mask, &old_mask);

    /* No SIGALRM will be delievered here since this signal is blocked.
     * This means we have a safe region here until we suspend below... */
    while (! wake_up) {
        sigsuspend(&old_mask);
    }

    /* Cleanup by restoring the original state. */
    sigprocmask(SIG_UNBLOCK, &mask, NULL);
    sigaction(SIGALRM, &old_sa, NULL);
    return 0;
}
```

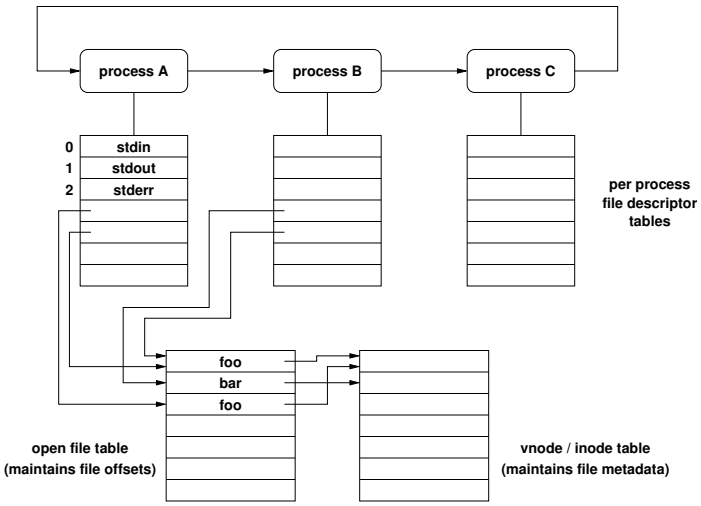
Listing 31: Implementation of the sleep() library function

Section 30: Pipes

Pipes enable us to compose processes. The concept of pipelines was introduced by Douglas McIlroy in 1973 during the development of the Unix operating system. The idea of pipes shaped the philosophy of the Unix operating system, which was summarized by Peter H. Salus in 1994 [21].

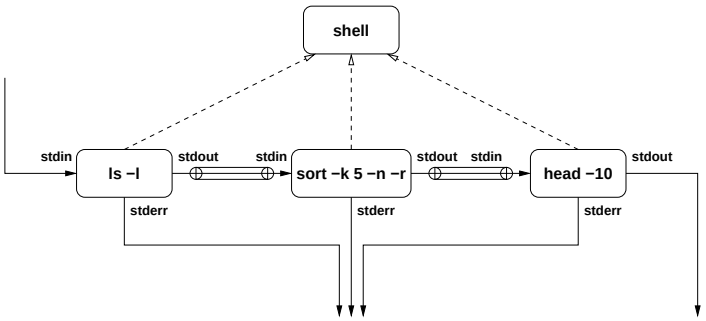
1. Write programs that do one thing and do it well.
2. Write programs to work together.
3. Write programs to handle text streams, because that is a universal interface.

Processes, File Descriptors, Open Files, ...



On a Unix / POSIX system, user space programs refer to an I/O “channel” using small numbers, called file descriptors, that are essentially an index into the file descriptor table. Every process has its own file descriptor table and the file descriptor table of a process is copied when a child process is created. Hence, a child process inherits the file descriptors of the parent process when it is created. The entries in the file descriptor table then refer to other tables that maintain further information about an I/O “channel”. For regular files, the kernel maintains an open file table, which then further refers to tables that are file system specific.

Pipes at the Shell Command Line




```
# list the 10 largest files in the
# current directory
ls -l | sort -k 5 -n -r | head -10
```


Pipes are kernel objects that support unidirectional communication from the write end of a pipe to the read end of a pipe. When the kernel creates a new pipe, it allocates two new file descriptors for the two endpoints of the pipe in the file descriptor table.

In the example shown above, the shell parses the command line and it sees two pipe symbols. This tells the shell that is has to create two pipes and that it needs to fork three child processes, one excuting the `ls` command, one executing the `sort` command, and one executing the `head` command. After forking child processes and before doing the `exec()` calls, the shell has to arrange the file descriptors such that the standard output of the first child process goes into the write end of the first pipe, that the standard input of the second child process is the read end of the first pipe and the standard output of the second child process is the write end of the second pipe, and that the standard input of the third child process is the read end of the second pipe.

POSIX Pipes

```
#include <unistd.h>

int pipe(int filedes[2]);
int dup(int oldfd);
int dup2(int oldfd, int newfd);

#include <stdio.h>

FILE *popen(const char *command, const char *type);
int pclose(FILE *stream);
```

- The `popen()` and `pclose()` library functions are wrappers to open a pipe to a child process executing the given command

The `pipe()` system call allocates a pipe in the kernel and it returns the two file descriptors refering to the read end of the pipe (first file descriptor) and the write end of the pipe (second file descriptor). The `dup2()` system call duplicates the old file descriptor to be (in addition) the new file descriptor.

Listing 32 demonstrates how a pipe can be used to copy data from a parent process to a child process. The program shown in Listing 33 does the same but it first duplicates the pipe file description and then uses the `cat` program to do the copying.

```c
/*
 * pipe/pipe.c --
 *
 *        This program sends data read from the standard input into a
 *        pipe connected to a child process. The child process copies
 *        data from the pipe to the standard output.
 */

#define _POSIX_C_SOURCE 200809L

#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <unistd.h>

static void copy(int sfd, int dfd)
{
    char buf[128];
    size_t len;

    while ((len = read(sfd, buf, sizeof(buf))) > 0) {
        (void) write(dfd, buf, len);
    }
}

int main(void)
{
    int pfd[2];
    pid_t pid;

    if (pipe(pfd) == -1) {
        perror("pipe");
        return EXIT_FAILURE;
    }

    pid = fork();
    if (pid == -1) {
        perror("fork");
        return EXIT_FAILURE;
    }
    if (pid == 0) {
        (void) close(pfd[1]);
        copy(pfd[0], STDOUT_FILENO);
        (void) close(pfd[0]);
    } else {
        (void) close(pfd[0]);
        copy(STDIN_FILENO, pfd[1]);
        (void) close(pfd[1]);
    }

    return EXIT_SUCCESS;
}
```

Listing 32: Demonstration of the pipe system call

```c
/*
 * pipe/pipex.c --
 *
 *        This program sends data read from the standard input into a
 *        pipe connected to a child process. The child process copies
 *        data from the pipe to the standard output. This version simply
 *        uses the cat utility after duplicating the file descriptors.
 */

#define _POSIX_C_SOURCE 200809L

#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <unistd.h>

/*
 * Duplicate the source fd (sfd) to the destination fd (dfd), close
 * other fds referring to the pipe, and finally execute "cat".
 */

static void cat(int *pfd, int sfd, int dfd)
{
    if (dup2(sfd, dfd) == -1) {
        perror("dup2");
        exit(EXIT_FAILURE);
    }
    (void) close(pfd[0]);
    (void) close(pfd[1]);
    execlp("cat", "cat", NULL);
    perror("execl");
    exit(EXIT_FAILURE);
}

int main(void)
{
    int pfd[2];
    pid_t pid;

    if (pipe(pfd) == -1) {
        perror("pipe");
        return EXIT_FAILURE;
    }

    pid = fork();
    if (pid == -1) {
        perror("fork");
        return EXIT_FAILURE;
    }
    if (pid == 0) {
        cat(pfd, pfd[0], STDIN_FILENO);
    } else {
        cat(pfd, pfd[1], STDOUT_FILENO);
    }

    return EXIT_SUCCESS;
}
```

Listing 33: Demonstration of the pipe and dup2 system call

```c
/*
 * pipe/mail.c --
 *
 *        This program sends an email by opening a pipe to sendmail.
 */

#define _POSIX_C_SOURCE 200809L

#include <stdio.h>
#include <stdlib.h>

typedef struct inet_mail {
    char *from;
    char *to;
    char *subject;
    char *content;
} inet_mail_t;

static int send(inet_mail_t *mail)
{
    FILE *pipe;
    int rc;

    pipe = popen("sendmail -t", "w");
    if (! pipe) {
        return -1;
    }
    fprintf(pipe, "From: %s\n", mail->from);
    fprintf(pipe, "To: %s\n", mail->to);
    fprintf(pipe, "Subject: %s\n", mail->subject);
    fprintf(pipe, "\n%s", mail->content);
    fflush(pipe);
    rc = ferror(pipe) ? -1 : 0;
    if (pclose(pipe) == -1) {
        rc = -1;
    }
    return rc;
}

int main(void)
{
    inet_mail_t mail = {
        .from = "<me@example.com>",
        .to = "<you@example.com>",
        .subject = "test email - please ignore",
        .content = "Hi,\n\n"
                   "This is a test email. Please excuse.\n\n"
                   "Chears, me.\n",
    };

    if (send(&mail) == -1) {
        fprintf(stderr, "mail: sending failed\n");
        return EXIT_FAILURE;
    }
    return EXIT_SUCCESS;
}
```

Listing 34: Using a pipe to send an email message

Named Pipes

- Named pipes are file system objects and arbitrary processes can read from or write to a named pipe (subject to file system permissions)
- Named pipes are created using the `mkfifo()` system call (or shell command)
- A simple example:

```
$ mkfifo pipe
$ ls > pipe &
$ less < pipe
```

- An interesting example:

```
$ mkfifo pipe1 pipe2
$ echo -n x | cat - pipe1 > pipe2 &
$ cat < pipe2 > pipe1
```


Pipes can only exist between processes which have a common parent process, which created the pipes. Named pipes are pipes that have a name in the file system name space. The file system permissions control which processes have access to named pipes.

Section 31: Sockets

Sockets

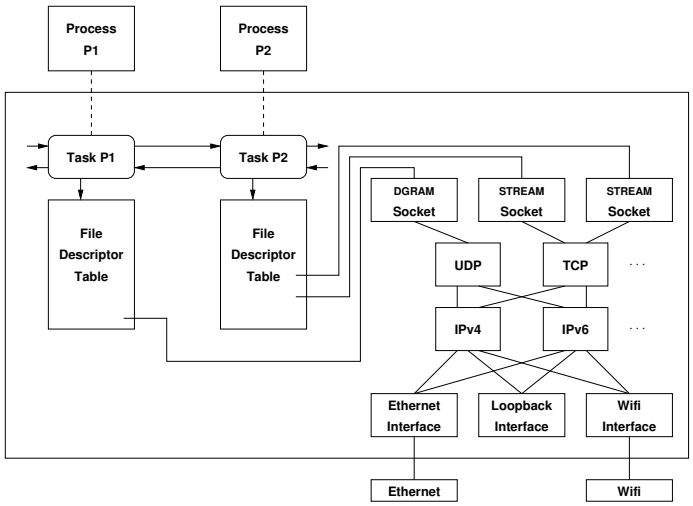
- Sockets are abstract communication endpoints with a rather small number of associated function calls
- The socket API consists of
  - address formats for various network protocol families
  - functions to create, name, connect, destroy sockets
  - functions to send and receive data
  - functions to convert human readable names to addresses and vice versa
  - functions to multiplex I/O on several sockets
- Sockets are the de-facto standard communication API provided by operating systems

We focus here on sockets that are communication endpoints of the Internet. While sockets can be used as endpoints of other commuication networks, we leave such details to more specializated courses.

In order to understand Internet sockets, we need some basic knowledge how the Internet works. Lets start with some terminology:

- The Internet is a *packet-switched* network. Data is sent in self-contained packets, where each packet carries full addressing information. Packets are (in principle) forwarded by the network independently, i.e., different packets may take different routes and packets may arrive out of order. It is possible that some packets never arrive or that packets are getting damaged and lost.
- A *node* is a computer with one or more *network interfaces*. A *router* is a node that forwards packets between interfaces while a *host* is a node that is not forwarding packets. (Routers establish the network topology while hosts are the leafs of the network topology.)
- A network interface can have multiple *IP addresses*. An IP packet is sent from a source network interface to a destination network interface and carries both the source and the destination IP address.
- There are currently two relevant IP address formats: IP version 4 (IPv4) addresses are 32 bits long and typically written in dotted-quad notation (e.g., `192.0.2.1`). IP version 6 (IPv6) addresses are 128 bits long and typically written in a hexadecimal notation (e.g., `2001:db8:0:0:0:0:0:1` or short `2001:db8::1`).
- A node can run many different applications (and supporting protocols) concurrently. IP packets carry additional information to properly deliver the packets to the right communication endpoints on a node. The information used to (de)multiplex packets are 16-bit *port numbers*.
- Hence, the address of an Internet communication endpoint consists of the combination of an IP address (32 or 128 bits) and a port number (16 bits).
- Note that addresses and port numbers must be encoded in network byte order (big endian format). Portable code must ensure that any necessary byte order conversions take place.

Sockets in the Kernel



On Unix-like systems, user-space processes refer to a socket by a file descriptor. The entry of the file descriptor table identified by the file descriptor refers to an data structure representing a socket. Sockets are associated with a transport protocol, usually UDP or TCP. The transport protocol itself is associated with a network protocol, usually IPv4 or IPv6. The network protocols exchange data through network interfaces. Network interfaces are abstract interfaces for concrete networking technologies, such as wired Ethernet networks or wireless WiFi local area networks. The loopback interface is a software interface that simply relays data back, providing a means to communicate locally between different processes, or even within the same process.

Socket Types

- Stream sockets (`SOCK_STREAM`) represent bidirectional communication endpoints providing reliable byte stream service
- Datagram sockets (`SOCK_DGRAM`) represent bidirectional communication endpoints providing unreliable connectionless message service
- Reliable delivered message sockets (`SOCK_RDM`) are bidirectional communication endpoints providing reliable connectionless message service
- Sequenced packet sockets (`SOCK_SEQPACKET`) are bidirectional communication endpoints providing reliable connection-oriented message service
- Raw sockets (`SOCK_RAW`) represent communication endpoints which can send/receive (raw) interface layer datagrams

Only two of these socket types are used widely on the Internet:

- Stream sockets are used in combination with the Transmission Control Protocol (TCP) of the Internet protocol suite.
- Datagram sockets are widely used in combination with the User Datagram Protocol (UDP) of the Internet protocol suite.

Both, TCP and UDP are transport layer protocols supporting a large collection of different application protocols. These two protocols are commonly implemented in an operating system kernel and sockets provide the interface to the protocol implementation residing in the kernel.

There are newer transport layer protocols such as DCCP or SCTP that enjoy only limited adoption. Several years ago, Google started pushing for a new transport protocol, meanwhile called QUIC. QUIC is designed to be implemented in user space, i.e., on top of the UDP transport protocol typically residing in kernel space. A significant portion of the traffic between Google browsers and Google servers is already running over QUIC.

Generic Socket Addresses

```
#include <sys/socket.h>

struct sockaddr {
    uint8_t     sa_len          /* address length (BSD) */
    sa_family_t sa_family;      /* address family */
    char        sa_data[...];   /* data of some size */
};

struct sockaddr_storage {
    uint8_t     ss_len;         /* address length (BSD) */
    sa_family_t ss_family;      /* address family */
    char        padding[...];   /* padding of some size */
};
```


A `struct sockaddr` represents an abstract generic address. Pointers to abstract generic addresses are sometimes down casted to a struct for a concrete address family or they are up casted from a struct for a concrete address family. A generic socket address (`struct sockaddr`) consists of a socket address family (`sa_family_t`). On BSD systems, a generic socket address also contains a length field (`uint8_t`). (This is from an architectural point of view the right thing to do but since early versions of the socket API did not specify such a length field, programmers cannot count on this length field being present on non-BSD systems.)

The `struct sockaddr_storage` can be used to allocate space for addresses of any address family supported by a system. This pseudo address format is useful for writing portable code without making assumptions about address formats and sizes.

```
1  {
2      struct sockaddr *sa;
3
4      sa = (struct sockaddr *) malloc(sizeof(struct sockaddr_storage));
5      if (! sa) {
6          // ...
7      }
8      memset(sa, 0, sizeof(struct sockaddr_storage));
9
10     // ...
11
12     free(sa);
13 }
```

IPv4 Socket Addresses

```
#include <sys/socket.h>
#include <netinet/in.h>

typedef ... sa_family_t;
typedef ... in_port_t;

struct in_addr {
    uint8_t  s_addr[4];         /* IPv4 address */
};

struct sockaddr_in {
    uint8_t     sin_len;        /* address length (BSD) */
    sa_family_t sin_family;     /* address family */
    in_port_t   sin_port;       /* transport layer port number */
    struct in_addr sin_addr;    /* network layer IPv4 address */
};
```


An IPv4 socket address (`struct sockaddr_in`) consists of the (optional) length and family fields followed by a port number (`in_port_t`) and an IPv4 address (`struct in_addr`).

The following code fragment dynamically allocates an IPv4 socket address and initialized it with the IPv4 address `192.0.2.1` and the port number `8080`.

```
1  {
2      struct sockaddr *sa;
3      struct sockaddr_in *sin;
4      char ip[INET_ADDRSTRLEN];
5
6      sa = (struct sockaddr *) malloc(sizeof(struct sockaddr_storage));
7      if (! sa) {
8          // ...
9      }
10     memset(sa, 0, sizeof(struct sockaddr_storage));
11
12     sin = (struct sockaddr_in *) sa;
13     sin->sin_family = AF_INET;
14     sin->port = htons(8080);     /* network byte order */
15     inet_pton(sin->sin_family, "192.0.2.1", &(sin->sin_addr));
16
17     inet_ntop(sin->sin_family, &(sa.sin_addr), ip, sizeof(ip));
18     printf("The socket address is %s port %d\n", ip, ntohs(sin->sin_port));
19
20     free(sa);
21 }
```

IPv6 Socket Addresses

```
#include <sys/socket.h>
#include <netinet/in.h>

typedef ... sa_family_t;
typedef ... in_port_t;

struct in6_addr {
    uint8_t  s6_addr[16];       /* IPv6 address */
};

struct sockaddr_in6 {
    uint8_t     sin6_len;       /* address length (BSD) */
    sa_family_t sin6_family;    /* address family */
    in_port_t   sin6_port;      /* transport layer port number */
    uint32_t    sin6_flowinfo;  /* network layer flow information */
    struct in6_addr sin6_addr;  /* network layer IPv6 address */
    uint32_t    sin6_scope_id;  /* network layer scope identifier */
};
```


An IPv4 socket address (`struct sockaddr_in6`) consists of the (optional) length and family fields followed by a port number (`in_port_t`) and an IPv6 address (`struct in6_addr`). And IPv6 address in addition carries a flow information field (`uint32_t`) and a scope identifier (`uint32_t`). Unless you have special needs, these fields should be set to zero.

The following code fragment dynamically allocates an IPv6 socket address and initialized it with the IPv6 address `2001:db8::1` and the port number `8080`.

```
1  {
2      struct sockaddr *sa;
3      struct sockaddr_in6 *sin6;
4      char ip[INET6_ADDRSTRLEN];
5
6      sa = (struct sockaddr *) malloc(sizeof(struct sockaddr_storage));
7      if (! sa) {
8          // ...
9      }
10     memset(sa, 0, sizeof(struct sockaddr_storage));
11
12     sin6 = (struct sockaddr_in6 *) sa;
13     sin6->sin_family = AF_INET6;
14     sin6->port = htons(8080);   /* network byte order */
15     inet_pton(sin6->sin6_family, "2001:db8::1", &(sin6->sin6_addr));
16
17     inet_ntop(sin6->sin6_family, &(sa6.sin6_addr), ip, sizeof(ip));
18     printf("The socket address is %s port %d\n", ip, ntohs(sin6->sin6_port));
19
20     free(sa);
21 }
```

Mapping Names to Addresses 1/2

```
#include <sys/types.h>
#include <sys/socket.h>
#include <netdb.h>

struct addrinfo {
    int              ai_flags;       /* flags controlling mappings */
    int              ai_family;      /* address family */
    int              ai_socktype;    /* socket type (e.g., SOCK_STREAM, SOCK_DGRAM)
    int              ai_protocol;    /* transport protocol number */
    size_t           ai_addrlen;     /* length of the socket address */
    struct sockaddr *ai_addr;        /* socket address */
    char            *ai_canonname;   /* possible canonical name */
    struct addrinfo *ai_next;        /* next pointer to form a linked list */
};
```


Addresses work very well for machines but they do not work well for humans. We can easily remember `amazon.com` or `google.com` or `ebay.com`, we usually do not remember the addresses used by these services. And these addresses may even change dynamically, so it is best to map names to addresses dynamically. The technology we use for that is called the Domain Name System (DNS). We do not go into the details how the DNS works here and leave that for a computer networking course. What we need to know is that we have names, more specifically domain names, that resolve to one or more addresses in one or more address families.

To represent one address mapping, we use the `struct addrinfo`. This structure provides everything we need to initialize a socket address. We can use the standard library function `getaddrinfo()` to perform a name lookup. This function will return an ordered list of addresses that we can use to communicate with an endpoint belonging to a given name. Note that not all address may actually work, so we should use a smart strategy to find out which one is usable [**?**].

A bit of a warning: The mapping provided by the domain name system has to be considered insecure. While there are extensions providing signed name resolutions, they are (still) not widely deployed and used. Hence it is crucial to verify via some other means whether the endpoint you communicate with is really the endpoint you wanted to communicate with.

Mapping Names to Addresses 2/2

```
#define AI_PASSIVE     ...
#define AI_CANONNAME   ...
#define AI_NUMERICHOST ...

int getaddrinfo(const char *node, const char *service,
                const struct addrinfo *hints, struct addrinfo **res);

void freeaddrinfo(struct addrinfo *res);

const char *gai_strerror(int errcode);
```


Some older text books describe somewhat outdated address mapping functions, which are not recommended to be used anymore. They main advantage of the new address mapping functions is that they make it easier to write code that can handle multiple existing address families and likely also any future address families. When IPv6 was introduced, a major problem was that many programs had hard wired assumptions that the Internet uses 32-bit IPv4 addresses. Rewriting applications to support multiple address families (without annoying users with long timeouts) during the transition phase from IPv4 to IPv6 has taken many years. Fast fallbacks to a different protocol version, in case the preferred address family is not working as expected, requires quite some user space code when implemented on top of the basic socket API.

Mapping Addresses to Names

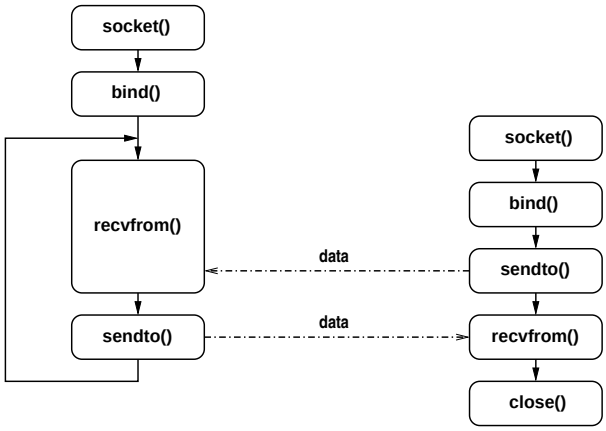
```
#include <sys/types.h>
#include <sys/socket.h>
#include <netdb.h>

#define NI_NOFQDN       ...
#define NI_NUMERICHOST  ...
#define NI_NAMEREQD     ...
#define NI_NUMERICSERV  ...
#define NI_NUMERICSCOPE ...
#define NI_DGRAM        ...

int getnameinfo(const struct sockaddr *sa, socklen_t salen,
                char *host, size_t hostlen, char *serv, size_t servlen,
                int flags);
const char *gai_strerror(int errcode);
```


Listing 35 demonstrates how names can be converted into a list of IP addresses without writing any version specific code. It is reasonably short and elegant. Unfortunately, the `NI_MAXHOST` and `NI_MAXSERV` definitions do not exist on all systems.

```c
/*
 * socket/examples/showip.c --
 *
 *      Print the IP addresses for each node name provided on the
 *      command line.
 */

#define _POSIX_C_SOURCE 200809L

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netdb.h>

#ifndef NI_MAXHOST
#define NI_MAXHOST      1025
#endif
#ifndef NI_MAXSERV
#define NI_MAXSERV      32
#endif

static void showip(char *name)
{
    int rc;
    struct addrinfo hints, *res, *p;
    char host[NI_MAXHOST], serv[NI_MAXSERV];

    memset(&hints, 0, sizeof(hints));
    hints.ai_family = AF_UNSPEC;
    hints.ai_socktype = SOCK_STREAM;
    rc = getaddrinfo(name, NULL, &hints, &res);
    if (rc) {
        fprintf(stderr, "getaddrinfo: %s\n", gai_strerror(rc));
        return;
    }
    for (p = res; p; p = p->ai_next) {
        rc = getnameinfo(p->ai_addr, p->ai_addrlen,
                         host, sizeof(host), serv, sizeof(serv),
                         NI_NUMERICHOST | NI_NUMERICSERV);
        if (rc) {
            fprintf(stderr, "getnameinfo: %s\n", gai_strerror(rc));
            continue;
        }
        printf("%s\t%s\n", name, host);
    }
    (void) freeaddrinfo(res);
}

int main(int argc, char *argv[])
{
    for (int i = 1; i < argc; i++) {
        showip(argv[i]);
    }
    return EXIT_SUCCESS;
}
```

Listing 35: Resolving names to IP addresses

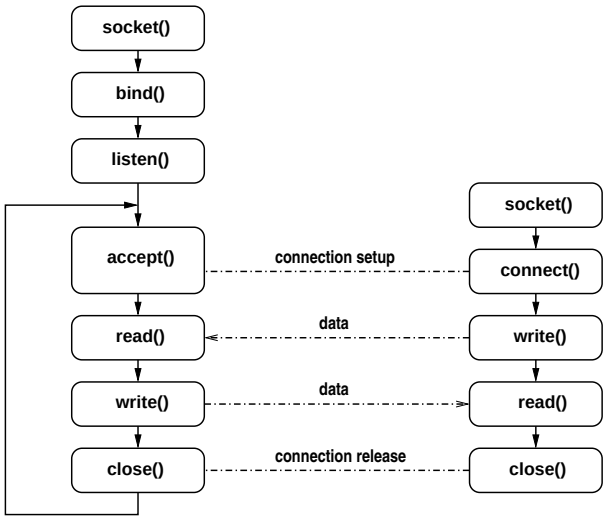
Connection-Less Communication



Connection-less datagram communication is extremely lightweight since no connection needs to be established or torn down. Furthermore, a single datagram socket can be used to communicate with many peers concurrently since the remote address can be provided as arguments to the `sendt()` and `recvfrom()` system calls. The downside of the datagram socket service is that it is unreliable, which means that datagrams can be lost, duplicated or reordered and applications have to deal with this. Furthermore, there are restrictions on the size of datagrams, which means applications may have to implement suitable fragmentation and reassembly functions.

Listings 36, 37, 38 and 39 show the core functions of a datagram (UDP) chat client. Note that the exchange of the messages is unreliable and messages are not protected from eavesdroppers.

Connection-Oriented Communication



Connection-oriented stream communication is in many situations much easier to use since it provides a bidirectional reliable byte-stream. Data is automatically segmented into datagrams and lost and reordered datagrams are automatically handled. Furthermore, connection-oriented communication protocols usually implements congestion control, mechanisms to avoid congestion collapse within the network. The price for this advanced service is some overhead to establish connections and to tear them down at the end.

Listings 40, 41, 42, 43 and 44 show the core functions of a stream (TCP) chat client. Note that messages are not protected from eavesdroppers.

Socket API Summary 1/3

```
#include <sys/types.h>
#include <sys/socket.h>
#include <unistd.h>

#define SOCK_STREAM    ...    /* stream socket */
#define SOCK_DGRAM     ...    /* datagram socket */
#define SOCK_RAW       ...    /* raw socket, requires privileges */
#define SOCK_RDM       ...    /* reliable delivered message socket */
#define SOCK_SEQPACKET ...    /* sequenced packet socket */

#define AF_INET  ...          /* IPv4 address family */
#define AF_INET6 ...          /* IPv6 address family */

#define PF_INET  ...          /* IPv4 protocol family */
#define PF_INET6 ...          /* IPv6 protocol family */
```

Socket API Summary 2/3

```
int socket(int domain, int type, int protocol);
int bind(int socket, struct sockaddr *addr, socklen_t addrlen);
int connect(int socket, struct sockaddr *addr, socklen_t addrlen);
int listen(int socket, int backlog);
int accept(int socket, struct sockaddr *addr, socklen_t *addrlen);

ssize_t write(int socket, void *buf, size_t count);
int send(int socket, void *msg, size_t len, int flags);
int sendto(int socket, void *msg, size_t len, int flags,
           struct sockaddr *addr, socklen_t addrlen);

ssize_t read(int socket, void *buf, size_t count);
int recv(int socket, void *buf, size_t len, int flags);
int recvfrom(int socket, void *buf, size_t len, int flags,
             struct sockaddr *addr, socklen_t *addrlen);
```

Socket API Summary 3/3

```
int shutdown(int socket, int how);
int close(int socket);

int getsockopt(int socket, int level, int optname,
               void *optval, socklen_t *optlen);
int setsockopt(int socket, int level, int optname,
               void *optval, socklen_t optlen);

int getsockname(int socket, struct sockaddr *addr, socklen_t *addrlen);
int getpeername(int socket, struct sockaddr *addr, socklen_t *addrlen);
```


All socket API functions operate on abstract socket addresses. Not all socket API functions make equally sense for all socket types.

Multiplexing (select)

```
#include <sys/select.h>

typedef ... fd_set;

FD_ZERO(fd_set *set);
FD_SET(int fd, fd_set *set);
FD_CLR(int fd, fd_set *set);
FD_ISSET(int fd, fd_set *set);

int select(int n, fd_set *readfds, fd_set *writefds, fd_set *exceptfds,
           struct timeval *timeout);
int pselect(int n, fd_set *readfds, fd_set *writefds, fd_set *exceptfds,
           struct timespec *timeout, sigset_t sigmask);
```


The `select()` system call works with arbitrary file descriptors. It can be used to implement the main loop of event-driven programs. There are other system calls offering similar functionality and that offer better performance when very large file descriptor sets need to be managed.

While it can be fun to write your own event loop, it is often much faster to use one of generic and heavily optimized event loops, like for example the `libevent` library. These libraries not only handle file descriptor events, they usually also support timer events and they often integrate signal handling. Furthermore, some of these library have APIs that can be used to integrate multiple event loops.

Non-blocking I/O (fcntl)

```
#include <unistd.h>
#include <fcntl.h>

#define F_GETFD ...     /* get file descriptor flags */
#define F_SETFD ...     /* set file descriptor flags */

#define O_NONBLOCK ...  /* non-blocking I/O */

int fcntl(int fd, int cmd, ... /* arg */ );
```

- I/O operations that would normally block fail with an EAGAIN error if `O_NONBLOCK` has been set on the file descriptor
- `fcntl()` can manipulate many more file descriptor properties

The following C code turns the file descriptor `fd` into a non-blocking file descriptor:

```
1    int flags = fcntl(fd, F_GETFL, 0);
2    fcntl(fd, F_SETFL, flags | O_NONBLOCK);
```

Once a file descriptor is non-blocking, system calls that would normally block (e.g., blocking until data is readable or a connection has been established) will fail with `errno` set to `EAGAIN` or `EWOULDBLOCK`. Non-blocking I/O ensures that programs are not getting blocked on an I/O operation, which is important for servers or interactive clients since both need to ensure that they are responsive. The price for this is, however, that the logic of the server of application needs to cater for an event-driven execution model where a logic operation is executed in a sequence of smaller code chunks that are called in an event-driven fashion and the intermediate state needs to be saved explicitely. (Some modern programming languages provide support for asynchronous programming, where the creation and maintenance of the underlying state machine is done by the compiler and mostly hidden from the programmer.)

Non-blocking I/O plays an important role for all programs that have to handle many concurrent I/O interactions. In some programming frameworks, programmers are encouraged or even forced to use multiple threads to deal with I/O system calls that can block the calling thread for a long time. Using threads for I/O is, however, in many cases not efficient since I/O bound programs don't get faster by exploiting computational concurrency. Multiple threads are useful for exploiting CPU resources, I/O is, however, not CPU bound (unless you do heavy cryptographic operations in software).

```c
/*
 * socket/lib/udp-connect.c --
 *
 *      Create a UDP socket and "connect" it to a remote server.
 */

#define _POSIX_C_SOURCE 201112L

#include <stdio.h>
#include <string.h>
#include <unistd.h>
#include <errno.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netdb.h>

#include "udp.h"

int udp_connect(const char *host, const char *port)
{
    struct addrinfo hints, *ai_list, *ai;
    int rc, fd = 0;

    memset(&hints, 0, sizeof(hints));
    hints.ai_family = AF_UNSPEC;
    hints.ai_socktype = SOCK_DGRAM;
    rc = getaddrinfo(host, port, &hints, &ai_list);
    if (rc != 0) {
        fprintf(stderr, "getaddrinfo: %s\n", gai_strerror(rc));
        return -1;
    }
    for (ai = ai_list; ai; ai = ai->ai_next) {
        fd = socket(ai->ai_family, ai->ai_socktype, ai->ai_protocol);
        if (fd < 0) {
            switch (errno) {
            case EAFNOSUPPORT:
            case EPROTONOSUPPORT:
                continue;
            default:
                perror("socket");
                continue;
            }
        } else {
            if (connect(fd, ai->ai_addr, ai->ai_addrlen) == -1) {
                (void) close(fd);
                perror("connect");
                continue;
            }
        }
        break;  /* we were successful, break out of the loop */
    }
    freeaddrinfo(ai_list);
    if (ai == NULL) {
        fprintf(stderr, "failed to connect to '%s' port '%s'\n", host, port);
        return -1;
    }
    return fd;
}
```

Listing 36: Creating a connected datagram (UDP) socket

```c
/*
 * socket/lib/udp-read-send.c --
 *
 *        Read data from a file descriptor and send it as a datagram.
 */

#define _POSIX_C_SOURCE 201112L

#include <stdio.h>
#include <unistd.h>
#include <sys/socket.h>

#include "udp.h"

int udp_read_send(int sfd, int dfd)
{
    char buf[1024];
    int len, rc;

    len = read(sfd, buf, sizeof(buf));
    if (len == -1) {
        perror("read");
        return -1;
    }
    if (len == 0) {
        return 0;
    }
    rc = send(dfd, buf, len, 0);
    if (rc == -1) {
        perror("send");
        return -1;
    }
    return rc;
}
```

Listing 37: Reading data and sending it as a datagram

```c
/*
 * socket/lib/udp-recv-write.c --
 *
 *        Receive a datagram and write data to a file descriptor.
 */

#define _POSIX_C_SOURCE 201112L

#include <stdio.h>
#include <unistd.h>
#include <sys/socket.h>

#include "udp.h"

int udp_recv_write(int sfd, int dfd)
{
    char buf[1024];
    ssize_t len;
    int rc;

    len = recv(sfd, buf, sizeof(buf), 0);
    if (len == -1) {
        perror("recv");
        return -1;
    }
    if (len == 0) {
        return 0;
    }
    rc = write(dfd, buf, len);
    if (rc == -1) {
        perror("write");
        return -1;
    }
    return rc;
}
```

Listing 38: Receiving a datagram and writing its data

```c
/*
 * socket/lib/udp-chat.c --
 *
 *        Chat with a UDP server, reading stdin / writing stdout.
 */

#define _POSIX_C_SOURCE 201112L

#include <stdio.h>
#include <unistd.h>
#include <sys/socket.h>
#include <sys/select.h>

#include "udp.h"

int udp_chat(int fd)
{
    const int maxfd = (fd > STDIN_FILENO ? fd : STDIN_FILENO);
    int rc;
    fd_set fdset;

    while (1) {
        FD_ZERO(&fdset);
        FD_SET(STDIN_FILENO, &fdset);
        FD_SET(fd, &fdset);
        if (select(1 + maxfd, &fdset, NULL, NULL, NULL) == -1) {
            perror("select");
            return -1;
        }

        if (FD_ISSET(fd, &fdset)) {
            rc = udp_recv_write(fd, STDOUT_FILENO);
            if (rc <= 0) {
                return rc;
            }
        }

        if (FD_ISSET(STDIN_FILENO, &fdset)) {
            rc = udp_read_send(STDIN_FILENO, fd);
            if (rc <= 0) {
                return rc;
            }
        }
    }

    return 0;
}
```

Listing 39: Chat with a datagram server, reading from stdin and writing to stdout

```c
/*
 * socket/lib/tcp-connect.c --
 *
 *      Establish a TCP connection to a remote server.
 */

#define _POSIX_C_SOURCE 201112L

#include <stdio.h>
#include <string.h>
#include <unistd.h>
#include <errno.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netdb.h>

#include "tcp.h"

int tcp_connect(const char *host, const char *port)
{
    struct addrinfo hints, *ai_list, *ai;
    int rc, fd = 0;

    memset(&hints, 0, sizeof(hints));
    hints.ai_family = AF_UNSPEC;
    hints.ai_socktype = SOCK_STREAM;
    rc = getaddrinfo(host, port, &hints, &ai_list);
    if (rc != 0) {
        fprintf(stderr, "getaddrinfo: %s\n", gai_strerror(rc));
        return -1;
    }
    for (ai = ai_list; ai; ai = ai->ai_next) {
        fd = socket(ai->ai_family, ai->ai_socktype, ai->ai_protocol);
        if (fd < 0) {
            switch (errno) {
            case EAFNOSUPPORT:
            case EPROTONOSUPPORT:
                continue;
            default:
                perror("socket");
                continue;
            }
        } else {
            if (connect(fd, ai->ai_addr, ai->ai_addrlen) == -1) {
                (void) close(fd);
                perror("connect");
                continue;
            }
        }
        break;  /* we were successful, break out of the loop */
    }
    freeaddrinfo(ai_list);
    if (ai == NULL) {
        fprintf(stderr, "failed to connect to '%s' port '%s'\n", host, port);
        return -1;
    }
    return fd;
}
```

Listing 40: Connecting a stream (TCP) socket

```c
/*
 * socket/lib/tcp-read.c --
 *
 *        The tcp_read() function behaves like read(), but it handles
 *        interrupted system calls, short reads, and non-blocking file
 *        descriptors.
 */

#define _POSIX_C_SOURCE 201112L

#include <unistd.h>
#include <errno.h>
#include <fcntl.h>

#include "tcp.h"

ssize_t tcp_read(int fd, void *buf, size_t count)
{
    size_t nread = 0;
    int flags;

    flags = fcntl(fd, F_GETFD);
    if (flags == -1) {
        return -1;
    }

    while (count > 0) {
        int r = read(fd, buf, count);
        if (r < 0 && errno == EINTR) {
            continue;
        }
        if (r < 0 && (errno == EAGAIN || errno == EWOULDBLOCK)) {
            return nread;
        }
        if (r < 0) {
            return r;
        }
        if (r == 0) {
            return nread;
        }
        buf = (unsigned char *) buf + r;
        count -= r;
        nread += r;
        if ((flags & O_NONBLOCK) == 0) {
            return nread;
        }
    }

    return nread;
}
```

Listing 41: Handling interrupted read system calls and short reads

```c
/*
 * socket/lib/tcp-write.c --
 *
 *        The tcp_write() function behaves like write(), but it handles
 *        interrupted system calls, short writes, and non-blocking file
 *        descriptors.
 */

#define _POSIX_C_SOURCE 201112L

#include <unistd.h>
#include <errno.h>
#include <fcntl.h>

#include "tcp.h"

ssize_t tcp_write(int fd, const void *buf, size_t count)
{
    size_t nwritten = 0;
    int flags;

    flags = fcntl(fd, F_GETFD);
    if (flags == -1) {
        return -1;
    }

    while (count > 0) {
        int r = write(fd, buf, count);
        if (r < 0 && errno == EINTR) {
            continue;
        }
        if (r < 0 && (errno == EAGAIN || errno == EWOULDBLOCK)) {
            return nwritten;
        }
        if (r < 0) {
            return r;
        }
        if (r == 0) {
            return nwritten;
        }
        buf = (unsigned char *) buf + r;
        count -= r;
        nwritten += r;
        if ((flags & O_NONBLOCK) == 0) {
            return nwritten;
        }
    }

    return nwritten;
}
```

Listing 42: Handling interrupted write system calls and short writes

```c
/*
 * socket/lib/tcp-read-write.c --
 *
 *      Copy data from a source a destination file descriptor using
 *      tcp_read() and tcp_write().
 */

#define _POSIX_C_SOURCE 201112L

#include <stdio.h>
#include <unistd.h>

#include "tcp.h"

int tcp_read_write(int sfd, int dfd)
{
    char buf[1024];
    int len, rc;

    len = tcp_read(sfd, buf, sizeof(buf));
    if (len <= 0) {
        return len;
    }
    rc = tcp_write(dfd, buf, len);
    return rc;
}
```

Listing 43: Copy data from a source a destination file descriptor

```c
/*
 * socket/lib/tcp-chat.c --
 *
 *        Chat with a TCP server, reading stdin / writing stdout.
 */

#define _POSIX_C_SOURCE 201112L

#include <stdio.h>
#include <unistd.h>
#include <sys/select.h>
#include <fcntl.h>

#include "tcp.h"

int tcp_chat(int fd)
{
    const int maxfd = (fd > STDIN_FILENO ? fd : STDIN_FILENO);
    int rc;
    fd_set fdset;

    (void) fcntl(fd, F_SETFL, O_NONBLOCK);
    (void) fcntl(STDIN_FILENO, F_SETFL, O_NONBLOCK);

    while (1) {
        FD_ZERO(&fdset);
        FD_SET(STDIN_FILENO, &fdset);
        FD_SET(fd, &fdset);
        if (select(1 + maxfd, &fdset, NULL, NULL, NULL) == -1) {
            perror("select");
            return -1;
        }

        if (FD_ISSET(fd, &fdset)) {
            rc = tcp_read_write(fd, STDOUT_FILENO);
            if (rc <= 0) {
                return rc;
            }
        }

        if (FD_ISSET(STDIN_FILENO, &fdset)) {
            rc = tcp_read_write(STDIN_FILENO, fd);
            if (rc <= 0) {
                return rc;
            }
        }
    }

    return 0;
}
```

Listing 44: Chat with a stream server, reading from stdin and writing to stdout

```c
/*
 * socket/chat/chat.c --
 *
 *      A basic chat program sending messages read from standard input
 *      to a server and printing messages received from a server.
 */

#define _POSIX_C_SOURCE 201112L

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>

#include "tcp.h"
#include "udp.h"

#define USE_TCP  0x1
#define USE_UDP  0x2

static void usage(FILE *stream, int status)
{
    (void) fprintf(stream, "usage: chat host port\n");
    exit(status);
}

int main(int argc, char *argv[])
{
    int c, fd, use = USE_TCP;

    while ((c = getopt(argc, argv, "ut")) != -1) {
        switch (c) {
        case 'u':
            use = USE_UDP;
            break;
        case 't':
            use = USE_TCP;
            break;
        case '?':
        default:
            usage(stdout, EXIT_SUCCESS);
        }
    }
    argc -= optind;
    argv += optind;

    if (argc != 2) {
        usage(stderr, EXIT_FAILURE);
    }

    if (use == USE_TCP) {
        if ((fd = tcp_connect(argv[0], argv[1])) == -1) {
            return EXIT_FAILURE;
        }
        if (tcp_chat(fd) == -1) {
            (void) tcp_close(fd);
            return EXIT_FAILURE;
        }
        (void) tcp_close(fd);
    }
    if (use == USE_UDP) {
        if ((fd = udp_connect(argv[0], argv[1])) == -1) {
            return EXIT_FAILURE;
        }
        if (udp_chat(fd) == -1) {
            (void) udp_close(fd);
            return EXIT_FAILURE;
        }
        (void) udp_close(fd);
    }
    return EXIT_SUCCESS;
}
```

Listing 45: Main function of the chat client

```c
/*
 * socket/lib/tcp-listen.c --
 *
 *        Create a listening TCP endpoint. Avoid IPv4 mapped addresses.
 */

#define _POSIX_C_SOURCE 201112L

#include <stdio.h>
#include <string.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netdb.h>

#include "tcp.h"

int tcp_listen(const char *host, const char *port)
{
    struct addrinfo hints, *ai_list, *ai;
    int rc, fd = 0, on = 1;

    memset(&hints, 0, sizeof(hints));
    hints.ai_flags = AI_PASSIVE;
    hints.ai_family = AF_UNSPEC;
    hints.ai_socktype = SOCK_STREAM;
    rc = getaddrinfo(host, port, &hints, &ai_list);
    if (rc) {
        fprintf(stderr, "getaddrinfo: %s\n", gai_strerror(rc));
        return -1;
    }

    for (ai = ai_list; ai; ai = ai->ai_next) {
        fd = socket(ai->ai_family, ai->ai_socktype, ai->ai_protocol);
        if (fd < 0) {
            continue;
        }
#ifdef IPV6_V6ONLY
        if (ai->ai_family == AF_INET6) {
            (void) setsockopt(fd, IPPROTO_IPV6, IPV6_V6ONLY, &on, sizeof(on));
        }
#endif
        setsockopt(fd, SOL_SOCKET, SO_REUSEADDR, &on, sizeof(on));
        if (bind(fd, ai->ai_addr, ai->ai_addrlen) == 0) {
            break;
        }
        (void) close(fd);
    }
    freeaddrinfo(ai_list);
    if (ai == NULL) {
        fprintf(stderr, "failed to bind to '%s' port %s\n", host, port);
        return -1;
    }

    if (listen(fd, 42) < 0) {
        perror("listen");
        (void) close(fd);
        return -1;
    }
    return fd;
}

```

Listing 46: Creating a listening TCP socket

```c
/*
 * socket/chatd/clnt.c --
 *
 *        Creation and deletion of clients, providing a broadcast API.
 */

#define _POSIX_C_SOURCE 201112L

#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <unistd.h>
#include <stdarg.h>
#include <event2/event.h>

#include "tcp.h"
#include "clnt.h"

static clnt_t *clients = NULL;

clnt_t* clnt_new(void)
{
    clnt_t *clnt;

    clnt = calloc(1, sizeof(clnt_t));
    if (! clnt) {
        perror("calloc");
        return NULL;
    }
    clnt->next = clients;
    clients = clnt;
    return clnt;
}

void clnt_del(clnt_t *me)
{
    clnt_t *clnt;
    int cfd = me->fd;

    event_del(me->event);
    (void) tcp_close(me->fd);
    if (me == clients) {
        clients = me->next;
        (void) free(me);
    } else {
        for (clnt = clients; clnt && clnt->next != me; clnt = clnt->next) ;
        if (clnt->next == me) {
            clnt->next = me->next;
            (void) free(me);
        }
    }
    clnt_bcast("server: clnt-%d left\n", cfd);
}

void clnt_bcast(const char *format, ...)
{
    va_list ap;
    char buf[1024];
    int len, rc;
    clnt_t *clnt, *gone = NULL;

    va_start(ap, format);
    len = vsnprintf(buf, sizeof(buf), format, ap);
    if (len > 0) {
        for (clnt = clients; clnt; clnt = clnt->next) {
            rc = tcp_write(clnt->fd, buf, len);
            if (rc <= 0) gone = clnt;
        }
    }
    if (gone) clnt_del(gone);
    va_end(ap);
}
```

Listing 47: Creation and deletion of clients and broadcast API

```c
/*
 * socket/chatd/clnt-event.c --
 *
 *        Client related event callbacks.
 */

#define _POSIX_C_SOURCE 201112L

#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <unistd.h>
#include <fcntl.h>
#include <event2/event.h>

#include "tcp.h"
#include "clnt.h"

void clnt_read(evutil_socket_t evfd, short evwhat, void *evarg)
{
    char buf[1024];
    int len;
    clnt_t *me = evarg;

    (void) evwhat;

    len = tcp_read(evfd, buf, sizeof(buf));
    if (len <= 0) {
        clnt_del(me);
        return;
    }

    clnt_bcast("clnt-%d: %.*s", evfd, len, buf);
}

void clnt_join(evutil_socket_t evfd, short evwhat, void *evarg)
{
    int cfd;
    clnt_t *clnt;
    struct event_base *evb = evarg;

    (void) evwhat;

    cfd = tcp_accept(evfd);
    if (cfd == -1) {
        return;
    }
    (void) fcntl(cfd, F_SETFL, O_NONBLOCK);
    clnt = clnt_new();
    if (! clnt) {
        return;
    }

    clnt->fd = cfd;
    clnt->event = event_new(evb, cfd, EV_READ|EV_PERSIST, clnt_read, clnt);
    (void) event_add(clnt->event, NULL);

    clnt_bcast("server: clnt-%d joined\n", cfd);
}
```

Listing 48: Client related event callbacks

```c
/*
 * socket/chatd/chatd.c --
 *
 *      A basic TCP chat daemon using libevent.
 */

#define _POSIX_C_SOURCE 201112L

#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>
#include <signal.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <event2/event.h>

#include "tcp.h"
#include "clnt.h"

typedef struct {
    char *address;
    int fd;
} listen_t;

static void usage(FILE *stream, int status)
{
    (void) fprintf(stream, "usage: chatd port\n");
    exit(status);
}

int main(int argc, char *argv[])
{
    struct event_base *evb;
    struct event *ev;
    listen_t *iface, interfaces[] = {
        { .address = "0.0.0.0" },               /* IPv4 any address */
        { .address = "::" },                    /* IPv6 any address */
        { .address = NULL }                     /* end marker */
    };

    if (argc != 2) {
        usage(stderr, EXIT_FAILURE);
    }

    if (signal(SIGPIPE, SIG_IGN) == SIG_ERR) {
        perror("signal");
        return EXIT_FAILURE;
    }

    evb = event_base_new();
    if (! evb) {
        fprintf(stderr, "event_base_new: failed\n");
        return EXIT_FAILURE;
    }
    for (iface = interfaces; iface->address; iface++) {
        iface->fd = tcp_listen(iface->address, argv[1]);
        if (iface->fd == -1) {
            continue;
        }
        ev = event_new(evb, iface->fd, EV_READ|EV_PERSIST, clnt_join, evb);
        event_add(ev, NULL);
    }
    if (event_base_loop(evb, 0) == -1) {
        fprintf(stderr, "event_base_loop: failed\n");
```

Part X

File Systems

We are used to store our data in named files. We have files for text documents, for calculation sheets, for source code, for program code, for images, for music, for videos, and many other digital objects. In order to deal with a large amount of files, we can organize files that relate to each other into directories (or folders). Finding a good organization of files is often surprisingly difficult and usually the organization of files takes time to develop.

The operating system kernel provides us with the abstraction of a hierarchical file system where data objects can be named and easily be found by a human. The operating system kernel allows us to create new files, to change files, to rename files, to delete files, and to associate permissions with file system objects. We are so used to these operations that we often forget that the underlying storage components (e.g., hard-drives or flash-drives), only provide us with numbered data blocks of fixed size, something that is barely useful for humans to work with.

Since file systems are fundamental for the storage of data, it is crucial that file systems are robust (we do not want to loose data) and efficient.

By the end of this part, students should be able to

- explain how a hiearchical file system name space is used to name files;
- describe the purpose of the special links named `.` and `..`;
- distinguish hard links from soft links and to explain their advantages and disadvantages;
- outline how a kernel resolves file descriptors to the data blocks storing the content of files;
- implement programs using file system specific system calls;
- explain file locking operations and semantics;
- outline how file system specific events can be obtained;
- discuss the advantages and disadvantages of a file system constructed using index nodes;
- sketch how file system operations translate to block updates in a file system using index nodes;
- illustrate how virtual file system APIs facilitate the integration of different kinds of file systems.

Section 32: File System Concepts

File Types

- Files are persistent containers for the storage of data
- Unstructured files:
  - Container for a sequence of bytes
  - Applications interpret the contents of the byte sequence
  - File name extensions may be used to identify content types (`.txt`, `.c`, `.pdf`)
  - Some file formats use internal "magic numbers" in addition to extensions
- Structured files:
  - Sequential files
  - Index-sequential files
  - B-tree files

$\Longrightarrow$ Only some operating systems support structured files

Support for structured files in the operating system kernel is meanwhile rather uncommon. It is much easier to support various file types through user space libraries. In situations where performance is most important and for workloads where regular generic file systems may not perform best, applications sometimes choose to bypass the in kernel file system support and they rather work with block devices where all input and output is under full application control.

Special Files

- Files representing devices:
  - Represent devices as files (`/dev/mouse`)
  - Distinction between block and character device files
  - Special operations to manipulate devices (`ioctl`)
- Files representing processes:
  - Represent processes (and more) as files (`/proc`)
  - Simple interface between kernel and system utilities
- Files representing communication endpoints:
  - Named pipes (fifos) and local domain sockets
  - Internet connections (`/net/tcp`) (Plan 9)
- Files representing graphical user interface objects:
  - Plan 9 represents all windows of a GUI as files

Unix/Linux systems represent devices in the `/dev` file system. On most modern distributions, the `/dev` file system is a special file system exporting device information as it is known by the kernel. This keeps the `/dev` file system reasonably small since devices files for devices that do not exist can be avoided.

The `/proc` file system is commonly used to expose information that the kernel maintains about running processes. On Linux systems, the `/proc` file system has grown substantially over the years.

Exposing network connections or even elements (widgets) of graphical user interfaces has so far been more experimental and has not been adopted widely.

Directories

- Hierarchical file system name spaces
  - Files are the leaves of the hierarchy
  - Directories are the nodes spanning the hierarchy
- Names of files and directories on one level of the hierarchy usually have to be unique (beware of uppercase/lowercase differences and character sets)
- Absolute names are formed by concatenating directory and file names
- Directories may be realized
  - as special file system objects or
  - as regular files with special contents

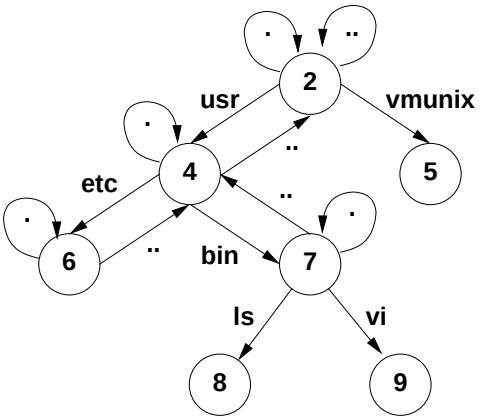
$\Longrightarrow$ Embedded operating systems sometimes only support flat file name spaces, or only read-only file systems, or no file systems at all

Applications tend to interact with the file system a lot and hence file systems have to be fast. To achieve fast access to frequently used data, file systems often use data caches residing in main memory. There are often even multiple caches involved such as block I/O buffer caches and dedicated directory caches for fast name lookups.

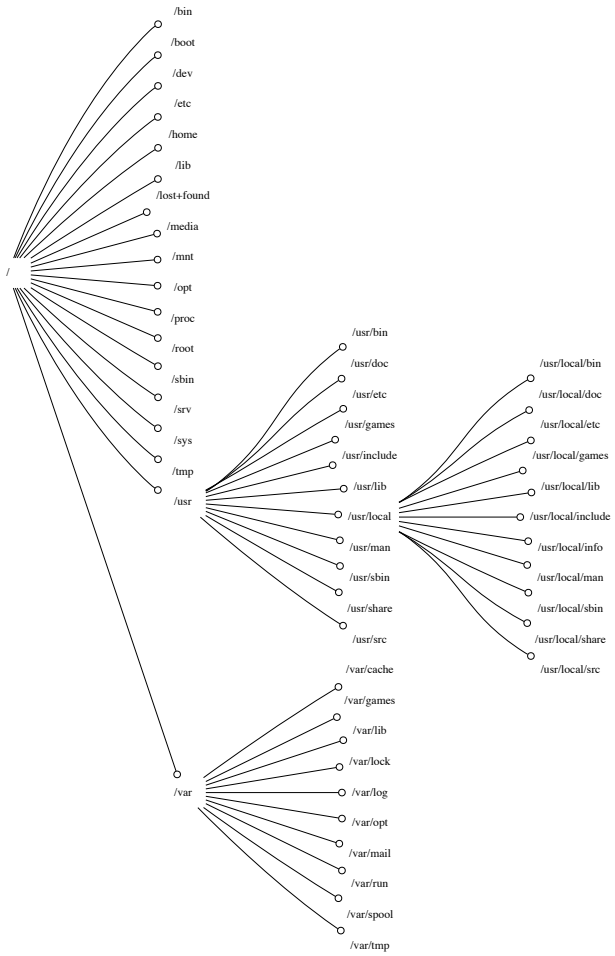
When it comes to file system modifications, operating systems often do not wait until a write operation has been carried out on the underlying storage medium. Instead, they signal a sucessful write to the application while the actual update is still only in a memory cache and waiting to be committed. Agressive caching can boost performance but it can also lead to data loss and file system inconsistencies if a system is not cleanly shut down. Modern file system designs try to find a balance between speed and robustness in cases of system failure.

Since file systems can become inconsistent, there are usually tools for each file system to identify inconsistencies and to repair them. On some file systems, files that were accidentally lost may be recovered and they may be located in special directories, like the `lost+found` directory that exists on several Linux file systems.

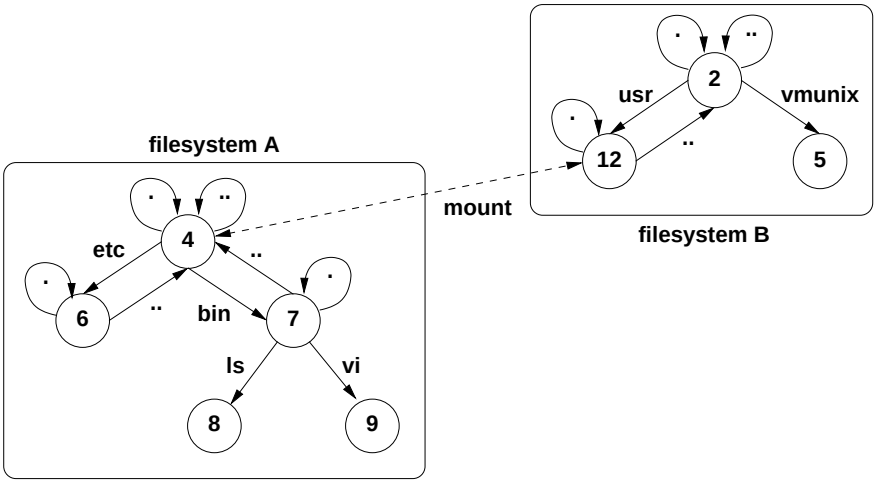
Unix Directory Structure



The structure of a typical Linux file system:



Mounting



Mounting is the process of making a directory from another file system (usually residing on some other storage system) available as part of a local file system name space. Mounting allows to build logical file system name spaces that span multiple devices. Note that the mounted file systems may be of different file system types. Mounted file systems may also reside on remote systems.

There is a large body of research and engineering work on networked and distributed file systems. A networked file system provides access to files stored on a remote computer over a computer network. A distributed file system supports distributed and often replicated storage of file content with the goal to reduce access times and to improve robustness against failures.

Hard Links and Soft Links (Symbolic Links)

Definition (hard link)

A *hard link* is a directory entry that associates a name with a file system object. The association is established when the link is created and fixed afterwards.

Definition (soft link)

A *soft link* or *symbolic link* is a directory entry storing a reference to a file system object in the form of an absolute or relative path. The reference is resolved at runtime.

- Links make file system object accessible under several different names
- Soft links may resolve to different file system objects (or none) depending on the current state of the file system
- Soft links can turn strictly hierarchical name spaces into directed graphs

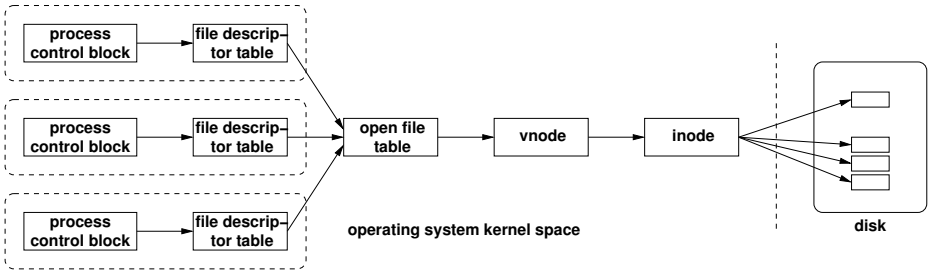
Large file systems often store replicated content. Modern file systems provide data deduplication features that detect duplicated content and replace duplicates with internal links to a single copy of the data.

A file system object is accessible as long as there is at least one hard link to it. File systems maintain an internal link count in order to keep track of how many hard links refer to a file system object. A file system can garbage collect the storage associated with a file when (i) the last hard link disappears and (ii) the last open file descriptor refering to the file is closed. As a consequence of this model, there is no "remove file" system call, there is only an "unlink" system call that removes a hard link.

File Usage Pattern

- File usage patterns heavily depend on the applications and the environment
- Typical file usage pattern of "normal" users:
  - Many small files (less than 10K)
  - Reading is more dominant than writing
  - Access is most of the time sequential and not random
  - Most files are short lived
  - Sharing of files is relatively rare
  - Processes usually use only a few files
  - Distinct file classes
- Totally different usage pattern exist (e.g., databases)

Processes and Files



- Every process control block maintains a pointer to the file descriptor table
- File descriptor table entries point to an entry in the open file table
- Open file table entries point to virtual inodes (vnodes)
- The vnode points to the inode (if it is a local file)

It is important to understand the relationship of file descriptor tables and the open file table.

- Every process has its own file descriptor table.
- File descriptors can be copied (duplicated).
- File descriptors can refer to open files but also other objects supporting I/O like pipes or sockets.
- When a new process is created, the parent's file descriptor table is (at least conceptually) copied to initialize the child's file descriptor table.
- Entries in the open file table keep track of the current position in the file (the current offset) and the file access mode (whether the file is open for reading or writing or both).
- It is possible to open a file several times with different access modes or to maintain different positions in the file.
- Entries in the open file table may be shared among processes. Reference counting is used to determine how long an open file has to remain open.

To investigate the files used by a running process, tools like `lsof` are quite handy.

Special File Systems

- Process file systems (e.g., profcs)
- Device file systems (e.g., devfs, udev)
- File systems exposing kernel information (e.g., sysfs)
- Ephemeral file systems (e.g., tmpfs)
- Union mount file systems (e.g., unionfs, overlayfs)
- User space file systems (e.g., fuse)
- Auto mounting file systems (e.g., autofs)
- Network file systems (e.g., nfs, cifs/smb)
- Distributed file systems (e.g., afs, lustre)

There is a large list of special purpose file systems.

Section 33: File System Programming Interface

Standard File System Operations

```
#include <stdlib.h>

int rename(const char *oldpath, const char *newpath);

#include <unistd.h>

ssize_t write(int fd, const void *buf, size_t count);
ssize_t read(int fd, void *buf, size_t count);
int close(int fd);
int link(const char *oldpath, const char *newpath);
int unlink(const char *pathname);
int access(const char *pathname, int mode);
int symlink(const char *oldpath, const char *newpath);
int readlink(const char *path, char *buf, size_t bufsiz);
```


Most of the C functions are C or POSIX standards and well portable across operating systems. The exact semantics of file system names are, however, less portable. File systems use different characters to separate names in a path, some file systems expose drive names, some file systems consider lowercase and uppercase letters to be the same, some file systems are rather unclear about the supported character sets (although UTF-8 may eventually succeed here as well but it will take a very long time to get rid of legacy file systems).

The `access()` system call should be used with care, it often leads to time-of-check to time-of-use (TOCTOU) problems, where a race exists between the check whether a file is accessible and a subsequent use of the file.

Standard File System Operations

```
#include <sys/types.h>
#include <sys/stat.h>
#include <unistd.h>

int open(const char *pathname, int flags);
int open(const char *pathname, int flags, mode_t mode);
int creat(const char *pathname, mode_t mode);
int mkfifo(const char *pathname, mode_t mode);
int stat(const char *file_name, struct stat *buf);
int fstat(int fd, struct stat *buf);
int lstat(const char *file_name, struct stat *buf);
int chmod(const char *path, mode_t mode);
int fchmod(int fd, mode_t mode);
int chown(const char *path, uid_t owner, gid_t group);
int fchown(int fd, uid_t owner, gid_t group);
int lchown(const char *path, uid_t owner, gid_t group);
```


A file system object has an associated owner (identified by a `uid_t` value, a numeric user identifier) and an associated group (identified by a `gid_t`, a numeric group identifier). The numeric user identifier and the numeric group identifier are scoped by the local system, which makes is difficult to share file systems over the network unless these numeric identifiers are managed to be consistent across system boundaries. Some modern networked file systems use the associated user names and group names as primary identifiers, which requires that the names are consistent.

A file system has an associated mode (identified by a numeric `mode_t` value). The mode carries basic access permissions for the owner of the file, the group the file is associated with, and everybody else on the system.

The `open()` call is problematic in some usage situations. For example, an application opening multiple files using relative file names may suffer from a race condition if the file system changes while the files are opened. A new `openat()` call has been introducted to fix this problem by referring to the base directory using an open file descriptor.

Standard Directory Operations

```
#include <sys/stat.h>
#include <sys/types.h>

int mkdir(const char *pathname, mode_t mode);
int rmdir(const char *pathname);
int chdir(const char *path);
int fchdir(int fd);

#include <dirent.h>

DIR *opendir(const char *name);
struct dirent *readdir(DIR *dir);
void rewinddir(DIR *dir);
int closedir(DIR *dir);
```

```c
/*
 * ls/ls.c --
 *
 *        A very basic implementation of 'ls' demonstrating how to read
 *        directories and obtain status information about files using
 *        POSIX interfaces.
 */

#define _POSIX_C_SOURCE 200112L

#include <stdlib.h>
#include <stdio.h>
#include <unistd.h>

#include "ls.h"

void
ls(const char *path, int flags)
{
    DIR *d;
    struct dirent *e;

    d = opendir(path);
    if (! d) {
        perror("opendir");
        return;
    }

    if (chdir(path) == -1) {
        perror("chdir");
        return;
    }

    while (1) {
        e = readdir(d);
        if (! e) {
            break;
        }
        if ((flags & LS_FLAG_ALL) || e->d_name[0] != '.') {
            if (flags & LS_FLAG_LONG) {
                show(e, flags);
            } else {
                puts(e->d_name);
            }
        }
    }

    (void) closedir(d);
}
```

Listing 50: Demonstration of directory operations

File Locking Operations (1/2)

```
#include <fcntl.h>

#define F_RDLCK ...   /* request a shared read lock */
#define F_WRLCK ...   /* request an exclusive write lock */
#define F_UNLCK ...   /* request to unlock */

#define SEEK_SET ...  /* lock region relative to file start */
#define SEEK_CUR ...  /* lock region relative to current position */
#define SEEK_END ...  /* lock region relative to file end */

#define F_SETLK  ...  /* acquire/release a lock, fail if lock unavailable */
#define F_SETLKW ...  /* acquire/release a lock, wait if lock unavailable */
#define F_GETLK  ...  /* investigate whether a lock is available */
```


POSIX systems traditionally support only advisory locks. Cooperating processes can acquire and release locks and coordinate their access to file content. However, processes that are unaware of locks (i.e., that choose to not coordinate access to file content with other processes) will not be prevented from accessing file content and thus such processes can cause problems.

The alternative to advisory locks are mandatory locks, where I/O system calls fail or block if processes would violate locks. Mandatory locks are often provided as non-standard extensions.

POSIX locks associate locks with file system nodes and process identifiers. This has a number of consequences:

- POSIX locks will not be shared between parent and child processes.
- A process opening a file multiple times will automatically see the POSIX locks shared across these multiple open files.
- Threads automatically share locks, i.e., POSIX file locks can't be used to coordinate access to shared files between concurrent threads.

Note that POSIX locks can be upgraded in an atomic way, i.e., a shared read lock can be turned into an exclusive write lock. Note that the operating system may drop locks if multiple processes try to update a shared lock to an exclusive lock concurrently.

File Locking Operations (2/2)

```
#include <fcntl.h>

struct flock {
    // ...
    short l_type;    /* one of F_RDLCK or F_WRLCK or F_UNLCK */
    short l_whence;  /* one of SEEK_SET or SEEK_CUR or SEEK_END */
    off_t l_start;   /* starting offset for lock region */
    off_t l_len;     /* number of bytes of the lock region */
    pid_t l_pid;     /* PID of process blocking our lock (set by F_GETLK) */
    ...
};

int fcntl(int fd, int cmd, ...);
```


On Linux systems, the `flock` utility can be used to acquire locks from the command line or shell scripts. The `lslocks` utility provides an overview about the locks that currently exist on the system. The following shell script sleeps for 10 seconds after obtaining an exclusive lock on a temporary file. A subsequent attempt to obtain the lock is delayed until the sleep command has finished.

```
1  file=$(tempfile)
2  flock $file sleep 10 &
3  sleep 1
4  lslocks
5  flock $file date
6  lslocks
7  unlink $file
```

Note that `fcntl` locks are not without problems, in particular in large programs where files may be opened many times without much control over them: Closing one of the file descriptors may accidentally remove a lock that is expected to be held at other places of the program. Sometimes people work around this by using the existence of additional files as a coarse grained file locking mechanism or by using non-standard locking APIs that bind locks to specific file descriptors.

```c
/*
 * locking/locks.c --
 *
 *        Demonstration of POSIX advisory locks using the fcntl API.
 */

#define _POSIX_C_SOURCE 201112L

#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <unistd.h>
#include <fcntl.h>

#include "locks.h"

int lock_file_read(int fd, off_t start, off_t len)
{
    struct flock fl;
    int rc;

    memset(&fl, 0, sizeof(fl));
    fl.l_type = F_RDLCK;
    fl.l_whence = SEEK_SET;
    fl.l_start = start;
    fl.l_len = len;
    rc = fcntl(fd, F_SETLKW, &fl);        /* potentially wait for the lock */
    if (rc == -1) {
        perror("fcntl");
    }
    return rc;
}

int lock_file_write(int fd, off_t start, off_t len)
{
    struct flock fl;
    int rc;

    memset(&fl, 0, sizeof(fl));
    fl.l_type = F_WRLCK;
    fl.l_whence = SEEK_SET;
    fl.l_start = start;
    fl.l_len = len;
    rc = fcntl(fd, F_SETLKW, &fl);        /* potentially wait for the lock */
    if (rc == -1) {
        perror("fcntl");
    }
    return rc;
}

int unlock_file(int fd, off_t start, off_t len)
{
    struct flock fl;
    int rc;

    memset(&fl, 0, sizeof(fl));
    fl.l_type = F_UNLCK;
    fl.l_whence = SEEK_SET;
    fl.l_start = start;
    fl.l_len = len;
    rc = fcntl(fd, F_SETLK, &fl);
    if (rc) {
        perror("fcntl");
    }
    return rc;
}
```

Listing 51: Demonstration of fcntl file locking

Memory Mapped Files

```
#include <sys/mman.h>

void* mmap(void *start, size_t length, int prot, int flags,
          int fd, off_t offset);
int munmap(void *start, size_t length);
int msync(void *start, size_t length, int flags);
int mprotect(const void *addr, size_t len, int prot);
int mlock(const void *addr, size_t len);
int munlock(const void *addr, size_t len);
```

- Direct mapping of regular files into virtual memory
- Enables extremely fast input/output and data sharing
- Mapped files can be protected and locked (regions)
- Changes made in memory are written to files during `unmap()` or `msync()` calls

File System Events

- Modern applications like to monitor file systems for changes.
- There are many system specific APIs, such as
  - `inotify` on Linux,
  - `kqueue` on *BSD,
  - `File System Events` on MacOS,
  - `ReadDirectoryChangesW` on Microsoft Windows.
- The APIs differ significantly in their functionality and whether they scale up to monitor large file system spaces.
- There are first attempts to build wrapper libraries that encapsulate system specific APIs (see for example `libfswatch`).
- A simple command line tool is `fswatch`.

Being able to monitor file system changes and to react to them became important when graphical user interfaces appeared. Graphical user interfaces often visualize the content of directories (or the desktop, which is often just a special directory) and ideally the visual representation should stay in sync with the actual state of the file system.

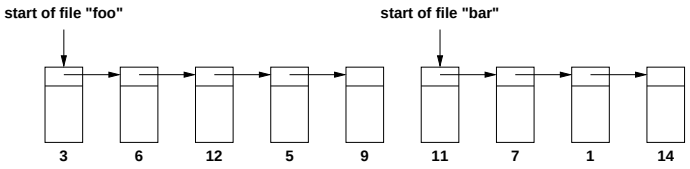
Other applications that like to monitor file system changes are programs that automatically synchronize file system content. A data synchronization service that got widely known and used was called dropbox. The dropbox software essentially monitors file systems for changes and then these changes are propagated to replicas.

There are some command line utilities that can be used to monitor file systems events or to trigger actions if certain file system events take place. The `fswatch` program detects file system events and writes messages to the standard output. The `entr` command can be used to run programs when certain file system events are detected. This can be used, for example, to trigger a compilation run when a source file is modified.

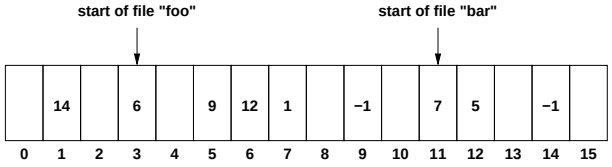
Section 34: File System Implementation

Block Allocation Methods using Lists

- Linked list allocation example:



- Indexed linked list allocation example:



Contiguous allocation

- Files stored as a contiguous block of data on the disk

+ Fast data transfers, simple to implement

+ Random access is easy and fast to implement

- File sizes often not known in advance

- Fragmentation on the underlying block storage device

Linked list allocation

- Every data block contains a pointer (number) to the next data block

+ No fragmentation on the underlying block storage device

+ Fast sequential access

- Random access is relatively slow (traversal of the list)

- Unnatural data block size (due to the space needed for the pointer)

Linked list allocation using an index

- The linked list is maintained in an index array outside of the data blocks

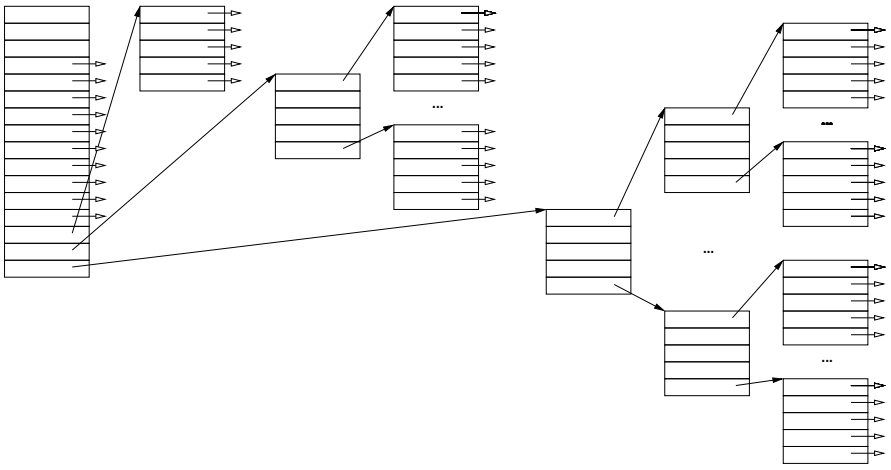
+ Index tables can remain in main memory for fast access

+ Random access is reasonably fast

+ Entire data blocks are available for data

- Memory usage for files with large index tables

Block Allocation Method using Index Nodes



Allocation using index nodes (inodes)

- Small index nodes (inodes) store pointers to the first few disk blocks plus pointers to
  - an inode with data pointers (single indirect)
  - an inode with pointers to inodes (double indirect)
  - an inode with pointers to inodes with pointers to inodes (triple indirect)

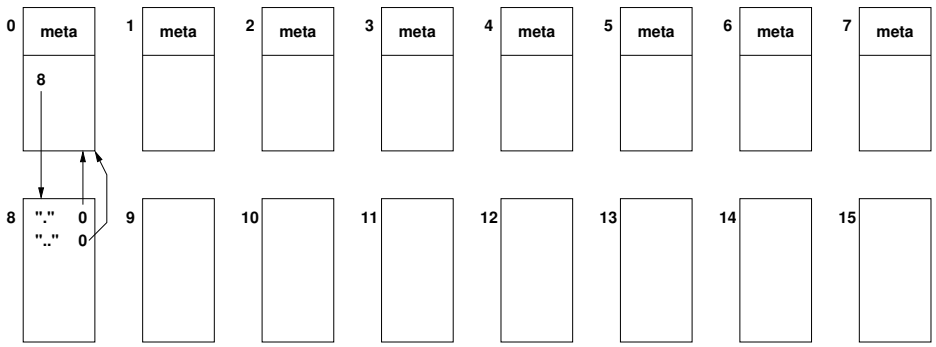
+ Fast sequential access
+ Random access is reasonably fast
+ Entire data blocks are available for data
+ Very efficient for small files
+ Provides space in the first inode to store metadata
- Caching of index nodes desirable

Several Unix file systems (4.4 BSD or the Linux extended file system) use inodes to manage block allocations. The `ls` utility shows the index node number when the `-i` command line option is used.

The metadata stored in an index node (inode) can be accessed using the `stat()` system call or the `stat` command line utility. Below you can see typical `stat` command output:

```
$ stat /
  File: /
  Size: 4096       Blocks: 8          IO Block: 4096   directory
Device: ca01h/51713d Inode: 2           Links: 25
Access: (0755/drwxr-xr-x)  Uid: (    0/    root)   Gid: (    0/    root)
Access: 2020-01-03 14:41:16.726606968 +0100
Modify: 2020-11-08 12:41:39.567554013 +0100
Change: 2020-11-08 12:41:39.567554013 +0100
 Birth: -
```

Index Node File System Example



- Block device with 16 equal-sized blocks (numbered 0...15)
- Blocks (0...7) are reserved for inodes, the remaining blocks are data blocks
- The root directory is always found in inode 0

We focus on the block references that the file system maintains. The initial situation can be described as follows:

```
inode   0: { 8 }                    // inode 0 refers to dnode(s) 8
inode   i: undef                    // inode i has undefined content (i in {1..7})
dnode   8: { (".", 0), ("..", 0) } // dnode 8 has 2 directory entries (to inode 0)
dnode   i: undef                    // dnode i has undefined content (i in {9..15})
freeblks: 0111111101111111          // free blocks bitmap (0 = used, 1 = free)
```

We are now making a number of changes to the file system and we write down (using the notation shown above) which inodes and/or dnodes are updated.

a) A file `/a` is created in the root directory, which occupies two data blocks. Assuming a block size of 4k, this can be done using the shell command `dd bs=4k count=2 if=/dev/random /a`.

```
inode   1: { 9, 10 }
dnode   9: random data
dnode  10: random data
dnode   8: { (".", 0), ("..", 0), ("a", 1) }
freeblks: 0011111100011111
```

b) A directory `/d` is created in the root directory, that is `mkdir /d`.

```
inode   2: { 11 }
dnode  11: { (".", 2), ("..", 0) }
dnode   8: { (".", 0), ("..", 0), ("a", 1), ("d", 2) }
freeblks: 0001111100001111
```

c) Create a hard link such that `/a` is also accessible as `/d/a`, that is `ln /a /d/a`.

```
dnode  11: { (".", 2), ("..", 0), ("a", 1) }
```

d) Remove (unlink) the file named `/a`, that is `rm /a`.

```
dnode   8: { (".", 0), ("..", 0), ("d", 2) }
```

e) Create a copy of the file `/d/a` in `/d/b`, that is `cp /d/a /d/b`.

```
inode   3: { 12, 13 }
dnode  12: data copied from dnode 9
dnode  13: data copied from dnode 10
dnode  11: { (".", 2), ("..", 0), ("a", 1), ("b", 3) }
```

```
freeblks: 0000111100000011
```

f) Create a symbolic link `/d/c` resolving to `/d/a`, that `ln -s /d/a /d/c`. Different solutions are possible. First slow links (the path is stored in a dnode):

```
inode  4: { 14 }
dnode 14: "/d/a"
dnode 11: { (".", 2), ("..", 0), ("a", 1), ("b", 3), ("c", 4) }
freeblks: 0000011100000001
```

Next fast links (the path is stored in the inode):

```
inode  4: { "/d/a" }
dnode 11: { (".", 2), ("..", 0), ("a", 1), ("b", 3), ("c", 4) }
freeblks: 0000011100000011
```

Finally, a solution storing the path in the directory itself:

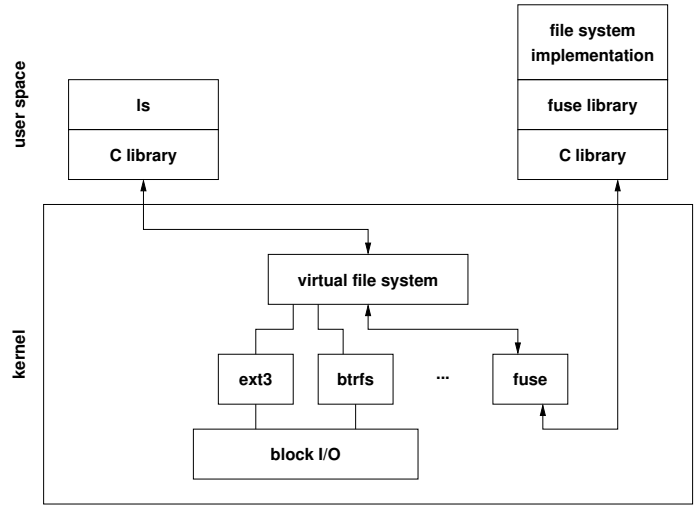
```
dnode 11: { (".", 2), ("..", 0), ("a", 1), ("b", 3), ("c", "/d/a") }
```

When making updates to a file system, there is often a trade-off here between performance and robustness. For example, one strategy is to cache the block updates and to write them in an order that gives best write performance. Another strategy is to execute the block writes in an order that minimizes the risk of file system inconsistencies caused by an abrupt system failure. There is a trade-off here between performance and robustness.

Free-Space Management

- *Free block lists*:
  - Manage free blocks in a linked free list
  - Efficient if there are only few free blocks
- *Free block bitmaps*:
  - Use a single bit for every block to indicate whether it is in use or not
  - Bitmap can be held in memory to make allocations and deallocations very fast
  - Sometimes useful to keep redundant bitmaps in order to recover from errors

Virtual File Systems (VFS)



- Abstract (virtual) file system interface
- Simplifies support for many different file systems
- Common functions implemented at the virtual file system interface
- Concrete file systems may reside in user space or on remote systems

On Linux, the virtual file system is defined by structures containing function pointers for the different file system operations (see `<linux/fs.h>`)

- `struct super_operations`
- `struct inode_operations`
- `struct file_operations`

A concrete file system implements suitable functions, initializes structures with the function pointers, and finally registers the structures in the kernel.

The file systems in user space (fuse) implementation exposes a simplified version of the virtual-file-system API to user space processes so that new file system ideas can be prototyped entirely in user space.

Further online information:

- **Wikipedia**: Filesystem in Userspace

Part XI

Devices

An operating system kernel has to organize input and output to a large number of diverse devices. Devices range from block storage devices to displays, keyboards, pointing devices, network adapters, printers, scanners, cameras, sound devices, just to name a few. As a consequence, the code handling the specifics of different devices is usually the largest part of an operating system kernel.

Devices can be classified into block devices and character devices:

- Block devices operate on fixed sized blocks of data. Mass data storage devices like hard disks or solid state disks are typical examples of block devices.
- Character devices operate on individual bytes or sequences or bytes of variable length. A large number of devices fall into this category, ranging from keyboards to sound cards.

A key challenge is to organize input and output to devices in such a way that best performance can be achieved. In addition, operating system kernels provide common programming interfaces for applications in order to minimize device specific code in application programs.

By the end of this part, students should be able to

- summarize the difference between character and block devices;
- describe how devices are represented in a file system;
- explain benefits of different buffering schemes;
- outline how RAIDs can improve performance and reliability;
- sketch the differences between mirroring and striping;
- express the principles of storage virtualization and logical volume management;
- explain the use of serial lines to connect character devices;
- name the modes of terminal device drivers;
- demonstrate how pseudo terminals can be used to implement a terminal program.

Section 35: Goals and Design Considerations

Design Considerations

- Device Independence
  - Hide low-level device details from applications
  - Be as generic as possible to encourage generic applications
  - Enable applications to exploit device specific characteristics
- Efficiency
  - Efficiency is of great concern, in particular for I/O bound applications
- Error Reporting
  - I/O operations have a high failure probability
  - Meaningful error reporting to applications
  - Proper error reporting and logging for system administrators

Efficiency: Buffering Schemes

- Data is passed without any buffering from user space to the device (unbuffered I/O)
- Data is buffered in user space before it is passed to the device
- Data is buffered in user space and then again in kernel space before it is passed to the device
- Data is buffered multiple times in order to improve efficiency or to avoid side effects (e.g., flickering in graphics systems)
- Circular buffers can help to decouple data producers and data consumers without copying data
- Vectored I/O (scatter/gather I/O) uses a single function call to write data from multiple buffers to a single data stream or to read data from a data stream into multiple buffers


```c
/*
 * hello-writev.c --
 *
 *      This program which invokes the Linux writev() system call,
 *      which is useful if the data to be written is stored at
 *      different memory locations.
 */

#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <sys/uio.h>

int
main(void)
{
    struct iovec iov[2];
    ssize_t n;

    iov[0].iov_base = "Hello ";
    iov[0].iov_len = strlen(iov[0].iov_base);
    iov[1].iov_base = "World\n";
    iov[1].iov_len = strlen(iov[1].iov_base);

    n = writev(STDOUT_FILENO, iov, 2);
    if (n == -1 || (size_t) n != iov[0].iov_len + iov[1].iov_len) {
        return EXIT_FAILURE;
    }

    return EXIT_SUCCESS;
}
```

Listing 52: Hello world program using vectored I/O

Efficiency: I/O Programming Styles

Definition (programmed input/output)

The CPU executes a program to copy data to/from the I/O device and it is unavailable for other tasks until an I/O action has completed.

Definition (interrupt-driven input/output)

Interrupts drive the I/O process, the CPU can do other things while the device is executing I/O requests, but the CPU has to process interrupts to steer the I/O action.

Definition (direct-memory-access input/output)

A direct-memory-access (DMA) controller moves data in/out of memory and notifies the CPU when an I/O action has completed; the CPU does not need to process any interrupts during the I/O action.

Error Reporting

- Provide a consistent and meaningful way to report errors and runtime exceptions to applications
- On POSIX systems, system calls indicate errors via special return values while a (thread) global variable `errno` provides details about the nature of the error
- Note: The variable `errno` stores the last error code and does not get cleared when a system call completes without an error
- Runtime errors not relating to a specific system call are handled in application or library specific ways
- Non-interactive programs should use logging facilities, e.g., `syslog` on Unix systems, to report errors

Generating good error messages takes time and effort but eventually pays off during the lifetime of a program. Good error messages are

- clear and not ambiguous
- concise and meaningful
- specific and relevant
- indicate where the error was detected
- describe the necessary details of the action that failed
- avoiding jargon that not everyone will understand
- easy to read out during a phone call
- never mixed into regular output
- written to error or logging facilities
- ...

Representation of Devices

Definition (block device)

A *block device* is a device where the natural unit of work is a fixed length data block.

Definition (character device)

A *character device* is a device where the natural unit of work is a byte.

Definition (device identification)

Devices are identified by their *device type* (block or character device), their *major device number* (identifying the responsible device driver), and their *minor device number* (identifying the device instance handled by the device driver).

On Unix systems, devices are represented as special objects in the file system (usually located in `/dev` directory). Many systems use a special file system to populate the `/dev` directory with devices that are meaningful for the system configuration. Some common device files:

- `/dev/null` – A character device accepting and discarding all data written to it. Indicating an end-of-file when read.
- `/dev/zero` – A character device accepting and discarding all data written to it. Produces an endless stream of NUL characters when read.
- `/dev/full/` – A character device returning ENOSPC when written. Produces an endless stream of NUL characters when read.
- `/dev/random/` – A character device returning random bytes when read. The device may block if the kernel runs out of entropy.
- `/dev/urandom/` – A character device returning random bytes when read. The device does not block if the kernel runs out of entropy.
- `/dev/tty` – A special character device that resolves to the tty of the current process, i.e., it potentially resolves to different ttys for different processes.
- `/dev/mem` – A character device representing the physical memory.
- `/dev/hd*` – Block devices representing classic IDE disk drives.
- `/dev/sd*` – Block devices representing modern disk drives.
- `/dev/tty*` – Character devices representing physical (serial) terminals.

Section 36: Storage Devices and RAIDs

Storage Media

- Magnetic disks (floppy disks, hard disks):
  - Data storage on rotating magnetic disks
  - Division into tracks, sectors and cylinders
  - Usually multiple (moving) read/write heads
- Solid state disks:
  - Data stored in solid-state memory (no moving parts)
  - Processing on a memory unit emulates hard disk interface
- Optical disks (CD, DVD, Blu-ray):
  - Read-only vs. recordable vs. rewritable
  - Very robust and relatively cheap
- Magnetic tapes (or tesa tapes):
  - Used mainly for backups and archival purposes
  - Very robust, high capacity, and relatively cheap

RAID

- Redundant Array of Inexpensive Disks (1988)
- Observation:
  - CPU speed grows exponentially
  - Main memory sizes grow exponentially
  - I/O performance increases slowly
- Solution:
  - Use lots of cheap disks to replace expensive disks
  - Redundant information to handle high failure rate
- Common on almost all small to medium size file servers
- Can be implemented in hardware or software

The acronym RAID did originally expand to Redundant Array of Inexpensive Disks [20]. It was later changed to Redundant Array of Independent Disks.

RAID Level 0 (Striping)

- Striped disk array where the data is broken down into blocks and each block is written to a different disk drive
- I/O performance is greatly improved by spreading the I/O load across many channels and drives
- Best performance is achieved when data is striped across multiple controllers with only one drive per controller
- No parity calculation overhead is involved
- Very simple design
- Easy to implement
- Failure of just one drive will result in all data in an array being lost

RAID Level 1 (Mirroring)

- Data is written to all disks in an array
- Twice the read transaction rate of single disks
- Same write transaction rate as single disks
- 100% redundancy of data means no rebuild is necessary in case of a disk failure
- Transfer rate per block is equal to that of a single disk
- Can sustain multiple simultaneous drive failures
- Simplest RAID storage subsystem design
- High disk overhead and thus relatively inefficient

RAID Level 5 (Distributed Parity)

- Data blocks are written onto data disks
- Parity for blocks is generated and recorded in a distributed location
- Parity is checked on reads
- High read data transaction rate
- Data can be restored if a single disk fails
- If two disks fail simultaneously, all data is lost
- Block read transfer rate equal to that of a single disk
- Controller design is more complex
- Widely used in practice

Section 37: Storage Virtualization

Logical Volume Management

Definition (Physical Volume)

A physical volume is a disk raw partition as seen by the operating system (hard disk partition, raid array, storage area network partition).

Definition (Volume Group)

A volume group pools several physical volumes into one logical unit.

Definition (Logical Volume)

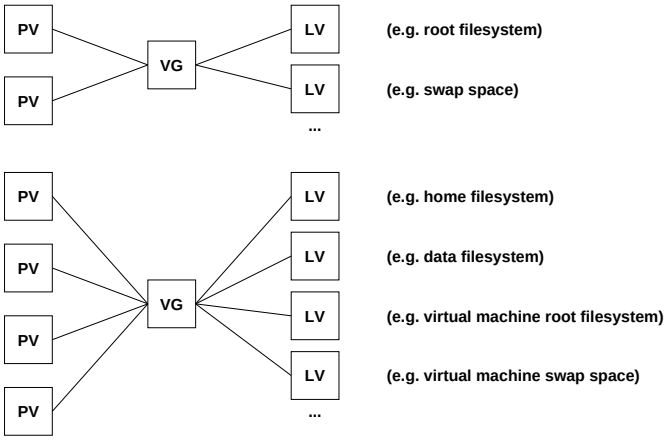
A logical volume resides in a volume group and provides a block device, which can be used to create a file system.

Logical volume management separates the logical storage layout from the physical storage layout. This simplifies the management of logical volumes, it becomes easy to create, remove, or resize logical volumes should the need arise.

Logical volumes provide many interesting features, in addition to the decoupling of logical storage layout from the physical storage layout:

- It is easy and efficient to resize logical volumes. The size of volumes can be increased as needed. Of course, after resizing the volume, the embedded file system must be resized as well to take advantage of the additional capacity.
- It is possible to take snapshots of logical volumes. Any writes to data blocks after the snapshot go into newly allocated blocks (copy-on-write). This feature of logical volume management systems can be used to make efficient consistent backups of logical volumes with extremely short interruptions to create the snapshot.
- The physical extends used to store data on physical devices can be moved. This can be used to migrate data from a physical device to another physical device without interrupting the system.

Logical Volume Management (Linux)



PV = physical volume, VG = volume group, LV = logical volume

Overview of some of the more frequently used LVM commands on Linux:

```
pvs             # display information about physical volumes
vgs             # display information about volume groups
lvs             # display information about logical volumes

pvcreate        # create physical volume
pvremote        # remove physical volume

vgcreate        # create volume groups
vgrename        # rename volume groups
vgremove        # remove volume groups

lvcreate        # create logical volumes
lvextend        # add space to a logical volume
lvreduce        # reduce the size of a logical volume
lvremove        # remove logical volumes
```

Networked Storage

Definition (storage area networks)

A storage area network (SAN) provides access to block devices over a network.

- Simplifies the sharing of storage between (frontend) computers
- Dedicated network technologies (Fibre Channel, iSCSI, ...)
- Relative expensive but fast technology

Definition (network attached storage)

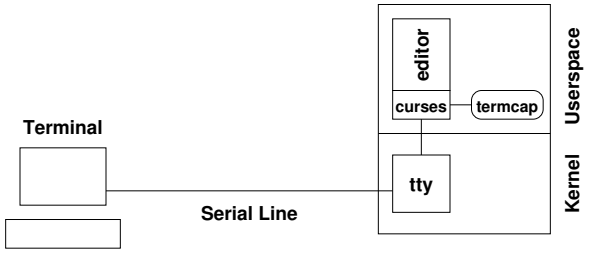
Network attached storage (NAS) provides access to file systems over a network.

- Sharing of file systems between multiple computers over a network
- Many different protocols (NFS, SMB/CIFS, AFP, ...)
- Relatively cheap but slow technology

NAS technology is relatively cheap and often sufficient in small office environments or private homes. SAN technology is commonly used by cloud computing infrastructures to separate storage for servers. Advanced solutions can provide support for high-availability storage and computing systems that can cope with failures by utilizing redundant hardware infrastructure.

Section 38: Terminal Devices

Traditional Character Terminal Devices



- Character terminals were traditionally connected via serial lines
- The kernel's device driver represents the terminal to user space programs
- Applications often use libraries, which are aware of terminal capabilities, to achieve terminal device independence

A classic terminal of the late 1970s was the VT100 terminal produced by the Digital Equipment Co-operation (DEC). The VT100 terminal supported 24 lines with 80 characters each. Due to its success in the market place, the terminal control sequences of the VT100 did become an de-facto standard for terminals that others manufacturers extended with additional control sequences in order to support additional features.

Since terminal control sequences started to differ quickly, it was desirable to provide an interface that makes it easy to adapt applications to new terminals supporting different control sequences. This problem has been dealt with by providing terminal capability descriptions in so called terminal capability databases so that applications (or better libraries used by applications) can lookup and adapt control sequences dynamically at runtime.

The `curses` library is an example of a software library that provides abstractions for text-based user interfaces (windows, menus, forms, scrollable text boxes, ...). The library also tries to minimize latency by carefully selecting control sequences to achieve a desired change on the terminal. Some editors and command line tools are built on top of `curses`, or the more recent version `ncurses`.

Serial Communication (RS232)

- Data transfer via two lines (TX/RX) using different voltage levels
- A *start bit* is used to indicate the beginning of the serial transmission of a word
- Parity bits may be sent (even or odd parity) to detect transmission errors
- One or several *stop bits* may be used after each word to allow the receiver to process the word
- *Flow control* can be implemented either using dedicated lines (RTS/CTS) or by sending special characters (XON/XOFF)
- Common settings: 8 data bits, 1 stop bit, no parity

Serial lines are still used to connect certain devices to computers. A typical challenge of using serial lines is that it is necessary to know the proper settings. It often takes trial and error (or a lot of experience) to find suitable settings.

Nowadays, many of the serial line connectors have been replaced by USB connectors. The Universal Serial Bus (USB) is a much more powerful and flexible serial bus system and USB connectors take much less space. However, USB technology is also much more complex and thus expensive and due to the simplicity of RS232, it can still be found in many (industrial) deployments.

Terminal Characteristics

- Serial lines were traditionally used to connect terminals to a computer
- Terminals understand different sets of control sequences (escape sequences) to control curser positioning or clearing of (parts of) the display
- Traditionally, terminals had different (often fixed) numbers of rows and columns they could display
- Keyboards were attached to the terminal and terminals did send different key codes, depending on the attached keyboard
- Teletypes were printers with an attached or builtin keyboard

Terminal Devices

- Unix systems represent terminals as `tty` devices.
  - In *raw mode*, no special processing is done and all received characters are directly passed on to the application
  - In *cooked mode*, the device driver preprocesses received characters, generating signals for control character sequences and buffering input lines
  - In *cbreak mode* (*rare mode*), the device driver does not buffer characters but still generates signals for some control characters sequences.
- Terminal capabilities are described in the (termcap, terminfo) databases.
- The `TERM` environment variable selects the terminal definition to use.
- Network terminals use the same mechanisms, but they are represented as pseudo tty devices, often called `ptys`.

On many Linux systems, terminfo is installed as part of the base distribution. The `TERM` variable indicates which terminal is in use. To obtain the terminal characteristics from the terminfo files, one can use the following shell command:

```
1    infocmp -L $TERM | less
```

Programs like `vim` or `top` are linked against the `tinfo` library providing access to the information stored in the terminfo files.

Communication over the network sometimes requires to represent a network connection as a terminal (e.g., `ssh`). To support this, kernels provide so called pseudo ttys, that behave a bit like bidirectional pipes but emulate terminal device behavior. A pseudo tty is a pair of a secondary and a primary tty. The secondary emulates a hardware text terminal device while the primary provides the interface to control the terminal. In a remote login scenario (`ssh`), the shell on the remote system interacts with a secondary tty while the daemon implementing the SSH network protocol interacts with the primary tty. Pseudo ttys have many other uses, e.g., to implement software terminals on a graphical user interface or to automate programs that expect to run on a terminal.

Portable and Efficient Terminal Control

- Curses is a terminal control library enabling the construction of text user interface applications
- The curses API provides functions to position the cursor and to write at specific positions in a virtual window
- The refreshing of the virtual window to the terminal is program controlled
- Based on the terminal capabilities, the curses library can find the most efficient sequence of control codes to achieve the desired result
- The curses library also provides functions to switch between raw and cooked input mode and to control function key mappings
- The `ncurses` implementation provides a library to create panels, menus, and input forms.

Listing 53 shows how the `ncurses` library can be used to write a less boring hello world program. The `initscr()` function determines the terminal type and initializes all curses data structures. The `cbreak()` function puts the `tty` (`pty`) into cbreak (rare) mode while the `noecho()` function turns the automatic echoing of typed characters off. The `nodelay()` function makes the `getch()` function used to read characters non-blocking.

The `move()` function moves the curser position to a specific position on the screen. The `mvaddch()` and `mvaddstr()` functions move the curser position as well and then output either a character or a string. The `clearol()` function clears the text from the current curser position to the end of the line.

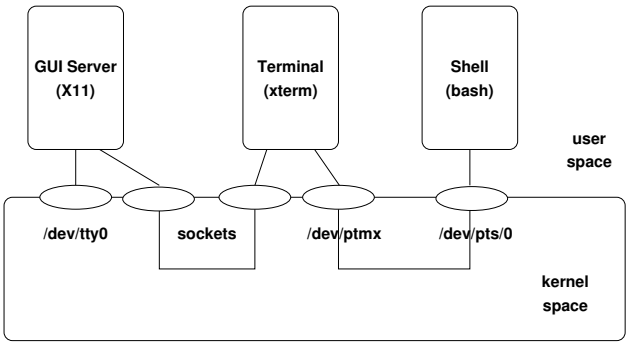
The `ncurses` library uses a buffer for screen updates. All movements and updates are performed on the buffer until they are complete. The `refresh()` call synchronizes the buffer with the screen. This is where the terminal characteristics are used to calculate efficient control sequences to update the screen.

The function `delwin()` deletes a window and the function `endwin()` restores the terminal mode into its original settings.

```c
/*
 * hello/hello-ncurses.c --
 *
 *        A hello world program using the [n]curses library.
 */

#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <curses.h>

static void cleanup(void)
{
    endwin();
}

int main(void)
{
    const char *spin[] = { "o<", "o-", NULL };
    const char *msg = "Hello, world!";
    WINDOW *win;
    int c, y, x, my, mx;

    if ((win = initscr()) == NULL) {
        fprintf(stderr, "Error initialising ncurses.\n");
        exit(EXIT_FAILURE);
    }
    atexit(cleanup);

    cbreak();
    noecho();
    nodelay(stdscr, TRUE);
    curs_set(0);
    my = LINES/2, mx = COLS/2 - strlen(msg)/2;

    for (y = 0; y < my; y++) {
        mvaddstr(y, mx, msg);
        refresh(); napms(100);
        move(y, mx); clrtoeol();
    }
    mvaddstr(my, mx, msg);
    refresh();
    for (x = 0; (c = getch()) == ERR && x < (int) (mx + strlen(msg)); x++) {
        mvaddstr(my, x, spin[x%2]);
        refresh(); napms(100);
        mvaddch(my, x, ' ');
    }
    for (y = my; y < LINES; y++) {
        mvaddstr(y, x, "oo");
        refresh(); napms(100);
        move(y, x); clrtoeol();
    }
    delwin(win);

    return EXIT_SUCCESS;
}
```

Listing 53: Hello world program using ncurses terminal control

Pseudo Terminal Devices



- The terminal device (`/dev/pts/0`) behaves like a traditional terminal device
- The pseudoterminal device (obtained by opening the pseudoterminal device pair multiplexer `/dev/ptmx`) controls the interaction with the terminal device

The slide shows how a terminal program provides a terminal device to a shell. Here is a high-level description how a terminal program on a Linux system sets things up:

1. The terminal program opens `/dev/ptmx`. This results in the allocation of a pseudoterminal device pair consisting of a pseudo terminal device and a terminal device. The file descriptor returned by the open system call refers to the pseudo terminal device of the newly allocated pseudoterminal device pair.
2. The terminal program obtains the name of associated terminal device (by calling `ptsname()`. On the slide, the terminal device of the pseudoterminal device pair is named `/dev/pts/0`.
3. The terminal program enables the pseudoterminal device pair by setting the correct permissions (`grantpt()`) and unlocking it (`unlockpt()`).
4. The terminal program opens the terminal device to be used by the shell.
5. The terminal program forks a child process.
6. The child process duplicates the file descriptor of the terminal device descriptor to the standard input, standard output and standard error file descriptors, closes all unneeded open file descriptors, and finally executes the shell.

The shell now interacts with the terminal device of the pseudoterminal device pair, which behaves like a traditional terminal device. The terminal program receives data from the graphical user interface server (typically via a socket) and writes the data into the pseudoterminal device of the pseudoterminal device pair. The data is copied by the kernel and appears on the shell's terminal device. Similarly, the shell writtes output to the shell's terminal device. The kernel copies that data to the pseudoterminal device. The terminal program reads the data and then interacts with the graphical user interface server via the socket to display the data.

A remote login (e.g., via `ssh`) works in a similar fashion. The remote login daemon on the remote system (e.g., `sshd`) uses a socket to interact with the client program and it forwards data received through the socket via the pseudoterminal device to the terminal device used by the shell. Similarly, output generated by the shell appears on the pseudoterminal device of the remote login daemon and is then copied into the socket to send it to the client.

Part XII

Virtualization

Virtual machines and container technology form the basis of today's cloud computing platforms and are meanwhile commonly used even on small server machines. The usage of virtual machine technology on desktop systems is also steadily increasing.

By the end of this part, students should be able to

- explain the difference between emulation and virtualization;
- distinguish between bare metal, hosted, os-level, and user-level virtualization;
- outline how Linux namespaces can be used to virtualize resources;
- sketch the use of control groups to partition resources;
- describe the differences between Docker containers and images;
- summarize the purpose of an orchestrator such as Kubernetes.

Section 39: Terminology and Architectures

Virtualization Concepts in Operating Systems

- Virtualization has already been seen several times in operating system components:
  - virtual memory
  - virtual file systems
  - virtual block devices (LVM, RAID)
  - virtual terminal devices (pseudo ttys)
  - virtual network interfaces (not covered here)
  - ...
- What we are talking about now is running multiple operating systems on a single computer concurrently.
- The basic idea is to virtualize the hardware, but we will see that there are differences in what is actually virtualized.

Virtualization is essentially a method that uses an additional indirection to decouple software components. If designed well, the performance overhead of a virtualization solution is well bounded and the gained flexibility is worth the resources invested for realizing the indirection.

Note that virtualization is also heavily used outside computing systems: Virtualization is widely used in computer networks, e.g., virtual local area networks (VLANs) or virtual private networks (VPNs).

Emulation

- Emulation of processor architectures on different platforms
  - Transition between architectures (e.g., PPC ⇒ Intel ⇒ ARM)
  - Faster development and testing of software for embedded devices
  - Development and testing of code for different target architectures
  - Usage of software that cannot be ported to new platforms
- QEMU (`http://www.qemu.org/`)
  - Full system emulation and user mode (process) emulation
  - Support for many different processor architectures
  - Dynamic translation to native code
  - Open source license

QEMU is very widely used and an essential tool for testing cross-compiled code and for developing tools for new CPU architectures. Recent QEMU versions support open hardware CPU designs such as RISC-V. QEMU achieves very good performance by translating emulated machine code dynamically into machine code executed natively by the host. In system emulation mode, QEMU emulates different drives, network interfaces, serial and parallel interfaces, keyboard and graphics cards, etc.

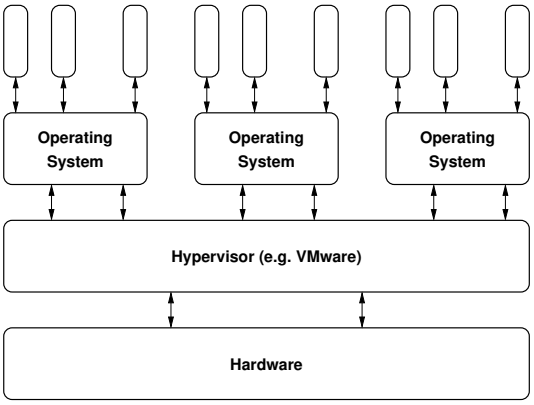
Type I (bare metal) Hardware Virtualization

- Virtualization of the physical hardware
  - Running multiple operating systems concurrently
  - Consolidation (replacing multiple physical machines by a single machine)
  - Separation of concerns and improved robustness
  - High-availability (live migration, tandem systems, ...)
- Examples:
  - VMware (`http://www.vmware.com/`)
  - Kernel-based Virtual Machines (`https://www.linux-kvm.org/`)
  - ...

Type I Hardware virtualization is the most general approach. The goal is usually to run unmodified operating systems on virtualized hardware components.

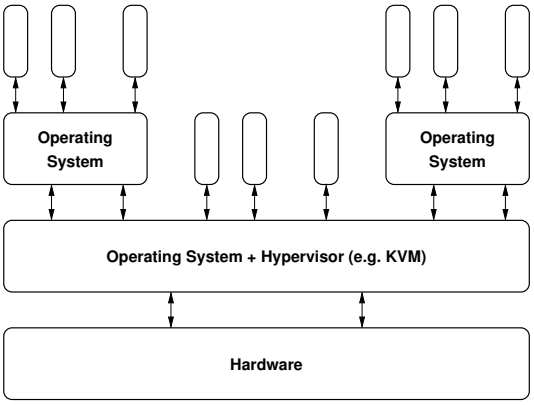
Note that the definition of a Type I hypervisor is somewhat fuzzy. While some solutions provide pure hypervisors and which full-features operating system kernels run, there are also solutions where the hypervisor functionality is a part of an ordinary operating system kernel. The Linux kernel-based virtual machines (KVM) fall into this category.

Example: VMware



- VMware (USA)
- 1998 VMware founded
- VMware ESXi (Hypervisor)
- VMware vSphere
- VMware Workstation
- closed source

Example: KVM

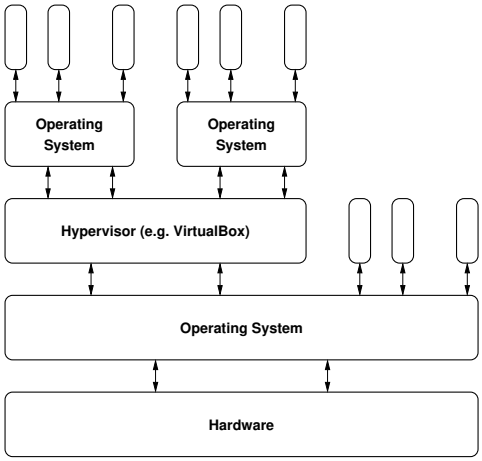


- Qumranet (Israel)
- 2007 integration in Linux
- 2008 bought by Red Hat
- OS Kernel Extension
- QEMU for device emulation
- OpenStack, Amazon, ...

Type II (hosted) Hardware Virtualization

- Virtualization on top of an operating system
  - Running multiple operating systems concurrently
  - Common solution for desktop systems
  - Usually less efficient than type I virtualization
- Examples:
  - VMware (`http://www.vmware.com/`)
  - VirtualBox (`https://www.virtualbox.org/`)
  - Parallels (`http://www.parallels.com/`)
  - ...

Example: VirtualBox



- InnoTek (Germany)
- 2007 open source (GPL)
- 2008 bought by Sun
- 2010 bought by Oracle
- core open source (GPL)
- extensions closed source
- Linux, Solaris
- Windows, MacOS

There are many alternatives to VirtualBox, in particular on the Desktop side with tighter integration into the native user interface.

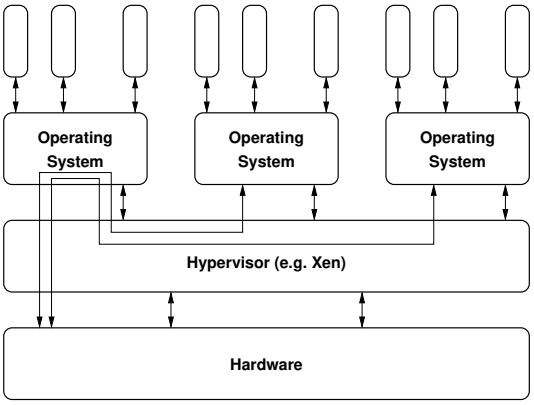
The Vagrant system is an open source project for managing virtual machines from the command line. It appeared in 2010 and it can use among other technologies VirtualBox as a backend virtualization system. Vagrant has a very simple configuration file syntax and it allows to create and share so called boxes (virtual machine images).

Paravirtualization

- Minimal hypervisor controlling guest operating systems
  - Minimal code complexity of the hypervisor
  - Reasonably efficient solution
  - Driver complexity moved to a single special guest operating system
  - Simplified device abstractions for all other operating systems
  - May require OS support and/or hardware support
- Examples:
  - Xen (`http://www.xenproject.org/`)

Paravirtualization tries to find a middle-ground between hardware virtualization and OS-level virtualization. The Xen system [1] is a well documented paravirtualization system.

Example: Xen



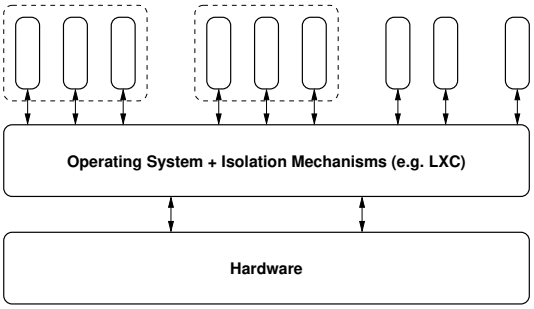
- University of Cambridge (UK)
- 2003 release 1.0 open source
- 2004 XenSource founded
- 2007 bought by Citrix Systems
- 2013 Linux Foundation
- Mircokernel Design

In the early days, the University of Cambridge published research papers about the Xen design, e.g., [1]. One of the special features of Xen was that it could migrate running virtual machines from one hypervisor to another with an almost unnoticeable downtime. Live migration is a quite complex but also quite common function of virtualization systems. Being able to relocate running virtual machines on the fly is essential on installations where a high level of availability must be guaranteed.

OS-Level Virtualization (Container)

- Multiple isolated operating system user-space instances
  - Efficient separation using namespaces and control groups
  - Robustness with minimal performance overhead
  - Reduction of system administration complexity
  - Restricted to a single operating system interface
- Examples:
  - Linux Container (LXC) (`https://linuxcontainers.org/`)
  - Linux VServer (`http://linux-vserver.org/`)
  - BSD Jails
  - Solaris Zones

Example: LXC



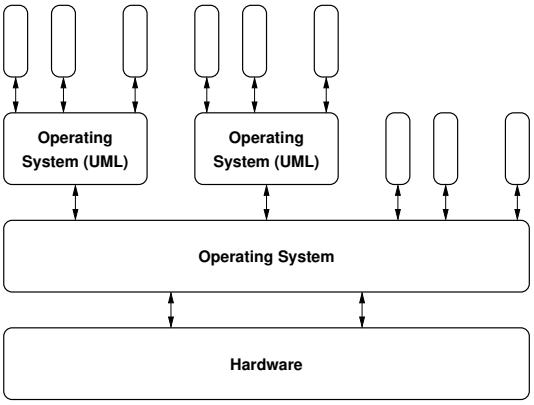
- 2008 initial release
- open source (LGPL, GPL)

Very similar to LXC was the Linux VServer, which was primarily designed and implemented by two individuals. The Linux VServer became quickly popular in web hosting environments as a cheap alternative to fully virtualized systems. While in principle very efficient, resource isolation was limited and if could happen that a neighboring virtual server did negatively impact your virtual server. The PlanetLab distributed system research testbed, established in 2002 and supported by hundreds of research institutions world-wide, did use Linux VServer technology extensively.

User-Level Virtualization

- Executing kernels as processes in user space
  - Simplify kernel development and debugging
  - Efficiency problems, rarely used in production
  - Often restricted to a single operating system
- Examples:
  - User-mode Linux (`http://user-mode-linux.sourceforge.net/`)

Example: UML



- 2003 integration in Linux

User-mode linux is not a very efficient virtualization solution and it had for a long time security limitations.

Section 40: Namespaces and Resource Management

Linux Namespaces

Linux namespaces isolate all global system resources. Existing namespaces:

- control group namespaces (see later)
- system V IPC and message queue namespaces
- network namespaces
- mount point namespaces
- process id namespaces
- time namespaces
- user and group id namespaces
- hostname and NIS namespaces

Command line tools can be used to manage namespaces:

- lsns: list namespaces
- nsenter: run a program with namespaces of other processes
- unshare: run a program with some namespaces unshared

The namespaces of a process can be found in `/proc/$pid/ns`.

Network namespaces can be used to instantiate multiple network stacks in the kernel. This has been used to create emulators such as mininet, that can emulate large computer networks on a regular host or notebook.

Linux Control Groups

A control group (cgroup) is a collection of processes under a set of resource limits.
Control groups are hierarchical and control resources such as memory, CPU, block I/O, or network usage.
Controller (subsystems) have been implemented to control the following resources:

- cpu scheduling and accounting
- cpu pinning (assigning specific CPUs to specific tasks)
- suspending or resuming tasks
- memory limits
- block I/O
- network packet tagging setting network traffic priorities
- namespaces
- performance analysis data collection

Control happens through a special cgroup file system. The file system hierarchy exposes the control group hierarchy. You can get an overview via the `/proc` file system: `/proc/cgroups` lists the controller supported by the kernel and `/proc/$pid/cgroup` lists the control groups a process belongs to. The `cgcreate` comand line tool can be used to create new control groups and `cgexec` can execute a process in a given control group. The `cgclassify` tool can move a set of processes to a given control group.

Section 41: Docker and Kubernetes

Docker

- Open-source software to manage container
- Moby is the base component and written in Go
- Docker appeared in 2013 and is managed by Docker Inc.
- Container were initially using Linux container (LCX)
- Meanwhile Docker uses its own libcontainer framework

There are alternatives to docker such as podman. However, at the time of this writing, Docker is clearly dominating.

Docker Terminology

- An image is a read-only template of a container. An image consists of layers. Images are portable and can be stored in repositories.
- A container is an active (running) instance of an image. An image can have many concurrently running container.
- A layer is a part of an image, it may consist of a command or files that are added to an image.
- A Dockerfile is a text file defining how an image is constructed.
- A repository is a collection of (versioned) images.
- A registry (like Docker Hub) manages repositories.

Kubernetes (K8s)

- Kubernetes is an orchestrator automating the deployment, scaling, and management of containerized applications on a cluster of hosts.
- Supporting several container tools, including Docker.
- Kubernetes appeared in 2014, initially developed by Google.
- Maintained by the Cloud Native Computing Foundation (CNCF).
- A core component is a key-value store called etcd.

A description of different orchestrator designs that led to the design of Kubernetes can be found in [6].

Kubernetes Terminology

- A pod consists of one or more containers that are co-located on a host machine.
- A service is a set of pods that work together.
- A replica set defines the number of pod instances that should be maintained.

References

[1] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield. Xen and the Art of Virtualization. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles*, October 2003.

[2] L.A. Belady, R.A. Nelson, and G.S. Shedler. An Anomaly in Space-Time Characteristics of Certain Programs Running in a Paging Machine. *Communications of the ACM*, 12(6):349–353, June 1969.

[3] A.D. Birrell. *Implementing Condition Variables with Semaphores*, pages 29–37. Springer, December 2004.

[4] Hans-J. Boehm. Threads cannot be implemented as a library. *SIGPLAN Notices*, 40(6):261–268, June 2005.

[5] Jeff Bonwick. The Slab Allocator: An Object-Caching Kernel. In *USENIX Summer 1994 Technical Conference (USENIX Summer 1994 Technical Conference)*, Boston, MA, June 1994. USENIX Association.

[6] Brendan Burns, Brian Grant, David Oppenheimer, Eric Brewer, and John Wilkes. Borg, omega, and kubernetes. *Communications of the ACM*, 59(5):50–57, April 2016.

[7] E. G. Coffman, M. Elphick, and A. Shoshani. System deadlocks. *ACM Computing Surveys*, 3(2):67–78, June 1971.

[8] P.J. Courtois, F. Heymans, and D.L. Parnas. Concurrent control with “readers” and “writers”. *Communications of the ACM*, 14(10):667–668, October 1971.

[9] P.J. Denning. The Working Set Model for Program Behavior. *Communications of the ACM*, 11(5):323–333, May 1968.

[10] E. W. Dijkstra. Cooperating Sequential Processes. *Programming Languages*, 1968.

[11] E. W. Dijkstra. *The mathematics behind the Banker's Algorithm*, pages 308–319. Springer-Verlag, 1982.

[12] Edsger W. Dijkstra. The structure of the ”THE”-multiprogramming system. *Commununications of the ACM*, 11(5):341–346, May 1968.

[13] A.B. Downey. *The Little Book of Semaphores*. Green Tea Press, 2.2.1 edition, 2016.

[14] H. Franke, R. Russell, and M. Kirkwood. Fuss, Futexes and Furwocks: Fast Userlevel Locking in Linux. In *Ottawa Linux Symposium 2002*, June 2002.

[15] R.L. Graham. Bounds on Multiprocessing Timing Anomalies. *SIAM Journal on Applied Mathematics*, 17(2):416–429, 1969.

[16] C.A.R. Hoare. Monitors: An operating system structuring concept. *Communications of the ACM*, 17(10), October 1974.

[17] B.W. Lampson and D.R. Redell. Experience with processes and monitors in mesa. *Communications of the ACM*, 23(2), February 1980.

[18] R. Love. *Linux Kernel Development*. Sams Publishing, 2003.

[19] M. K. McKusick, K. Bostic, M. J. Karels, and J. S. Quarterman. *The Design and Implementation of the 4.4 BSD Operating System*. Addison Wesley, 1996.

[20] D.A. Patterson, G. Gibson, and R.H. Katz. A case for redundant arrays of inexpensive disks (raid). In *Proceedings of the 1988 ACM SIGMOD International Conference on Management of Data (SIGMOD '88)*, page 109–116. Association for Computing Machinery, June 1988.

[21] P. H. Salus. *A Quarter Century of UNIX*. Addison Wesley, 1994.