Operating Systems 2022

Jürgen Schönwälder

February 20, 2023

`https://cnds.jacobs-university.de/courses/os-2022/`

C›ONSTRUCTOR
UNIVERSITY

Objectives

- Understand how an operating systems manages to turn a collection of independent hardware components into a useful abstraction
- Understand concurrency issues and be able to solve synchronization problems
- Knowledge about basic resource management algorithms
- Understand how memory management systems work and how they may impact the performance of systems
- Knowledge of inter-process communication mechanisms (signals, pipes, sockets)
- Understand trade-offs in the design of file systems
- Understand virtualization and container technology

Intended Learning Outcomes

- explain the differences between processes, threads, application programs, libraries, and operating system kernels;
- describe well-known mutual exclusion and coordination problems;
- use semaphores to achieve mutual exclusion and solve coordination problems;
- use mutual exclusion locks and condition variables to solve synchronization and coordination problems;
- illustrate how deadlocks can be avoided, detected, and resolved;
- summarize the different mechanisms to realize virtual memory and their trade-offs;
- solve basic inter-process communication problems using signals and pipes;
- use socket inter-process communication primitives;
- multiplex I/O activities using suitable system calls and libraries;
- describe file system programming interfaces and the design of file systems at the operating system kernel level;
- explain how memory mapping can improve I/O performance;
- restate the functionality of a linker and the difference between static linking and dynamic linking;
- outline how different device types are supported by Unix-like kernels;
- discuss virtualization mechanisms such as containers or virtual machines.

Topics and Timeline

I	Introduction	1 week
II	Hardware	0.5 weeks
III	Processes and Threads	0.5 weeks
IV	Synchronization	2 weeks
V	Deadlocks	0.5 weeks
VI	Scheduling	0.5 weeks
VII	Linking	0.5 weeks
VIII	Memory Management	1.5 weeks
IX	Inter-Process Communication	3 weeks
X	File Systems	1 week
XI	Input/Output and Devices	1 week
XII	Virtual Machines	1 week
XIII	Distributed Systems	1 week

Assessment

- Module achievement (during the semester):
  - 50% of 10 (weekly) assignments correctly solved
  - 2 additional (weekly) assignments can be used to makeup points
  - Students without module achievement are not allowed to sit for the exam
  - Submit homework solutions regularly from the beginning

- Written examination (December 2022 and/or January 2023):
  - Duration: 120 min (closed book)
  - Scope: All intended learning outcomes of the module

- You can audit the module. To earn an audit, you have to pass a short oral interview about key concepts introduced in this module at the end of the semester.

Assignments

- We will post weekly homework assignments
- Assignments reinforce what has been discussed in class
- Assignments will be small individual assignments (but may take time to solve)
- Solving assignments will prepare you for the written examination
- Solutions must be submitted individually via Moodle
- Teaching assistants will review the assignments
- Assignments tell you whether you understood the material

Study Groups

- I strongly suggest to form study groups.
- It helps to discuss questions and the course materials in study groups, in particular when you are getting stuck.
- Discussions in a study group can help to understand what is demanded by a problem.
- Study group members may try different approaches to solve a problem and you can benefit from that.
- However, note that submissions must be individual solutions.
- It is OK to sketch a possible solution in a study group, then work out the details of the solution yourself.

Deadlines

- Deadlines will be strict (don't bother to ask for extensions)
- Make sure you submit the right document. We grade what was submitted, not what could have been submitted.
- Submit early — avoid last minute changes or software/hardware problems.
- Official excuses by the student records office will extend the deadlines, but not more than the time covered by the excuse.
- A word on medical excuses: Use them when you are ill. Do not use them as a tool to gain more time.
- You want to be taken serious if you are seriously ill. Misuse of excuses can lead to a situation where you are not taken too serious when you deserve to be taken serious.

Reading Material

- A. Silberschatz, P.B. Galvin, B. Peter, G. Gagne: "Applied Operating System Concepts", John Wiley, 2000

- A.S. Tanenbaum, H. Bos: "Modern Operating Systems", Prentice Hall, 4th edition, Pearson, 2015

- W. Stallings: "Operating Systems: Internals and Design Principles", 8th edition, Pearson, 2014

- R. Love: "Linux Kernel Development", 3rd edition, Addison Wesley, 2010

- R. Love: "Linux System Programming: Talking Directly to the Kernel and C Library", 2nd edition, O'Reilly, 2013

Part 1: Introduction

1 Definition, Requirements and Services

2 Fundamental Concepts

3 Types of Operating Systems

4 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, 2018-08-16)

Hardware vs. System vs. Application

Browser, Databases, Office Software, Games, ...	**Application Software**

library calls →

system calls →

interrupts →

Shells, Editors, Utilities, Compiler, Linker, ...	**System Software**
System Libraries	
Operating System Kernel	

Machine language	**Memory**	**Devices**	**Hardware**
Microprograms			
Integrated circuits			
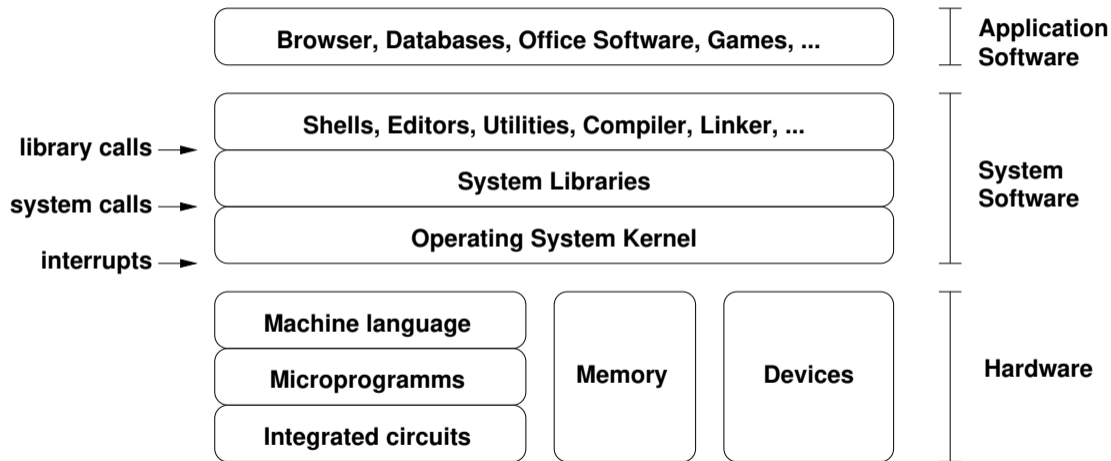
General Requirements

- 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.

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
- . . .

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
- . . .

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.

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.

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

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.

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.

Section 3: Types of Operating Systems

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, . . .

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)

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

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

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)

Section 4: Operating System Architectures

Operating System Architectures
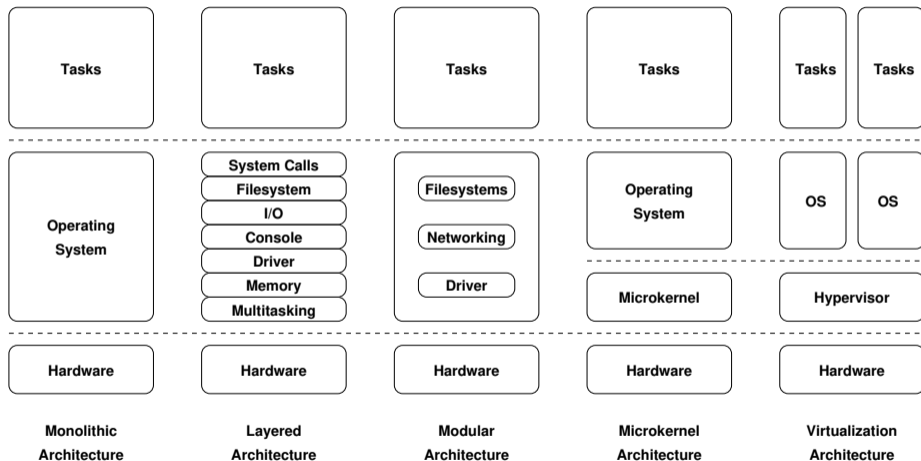
Kernel Modules / Extensions

- 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

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

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

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

Part 2: Hardware

Section 5: Computer Architecture and Processors

5 Computer Architecture and Processors

6 Memory, Caching, Segments, Stacks

7 Devices and Interrupts

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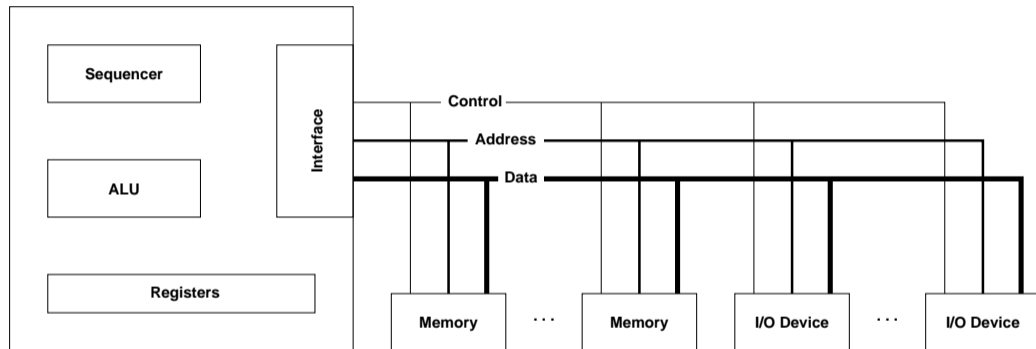
- Today's common computer architecture uses busses to connect memory and I/O systems to the central processing unit (CPU)

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

Section 6: Memory, Caching, Segments, Stacks

Memory Sizes and Access Times

Memory Size		Access Time
< 1 KiB	CPU Registers	< 1 ns
~ 128 KiB	Level 1 Cache	~ 1–2 ns
~ 1 MiB	Level 2 Cache	~ 4 ns
> 1 GiB	Main Memory	~ 8 ns
> 128 GiB	Disks (SSD or HDD)	~ 1–4 ms

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

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)

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

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)

stack segment

return address (4 byte)
frame pointer (4 byte)
c (4 byte)
b (4 byte)
a (4 byte)
buffer1 (40 bytes)
buffer2 (48 bytes)

heap segment

data segment

text segment

```
void
function(int a, int b, int c)
{
    char buffer1[40];
    char buffer2[48];
}
```
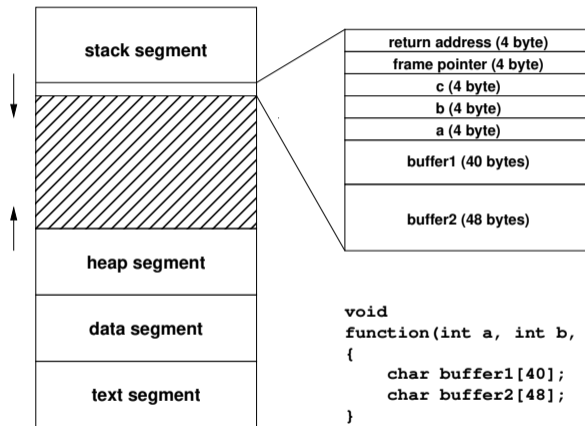
Example

```
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?

Stack Smashing Attacks

```c
#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)

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

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

Part 3: Processes and Threads

8 Processes

9 Threads

Section 8: Processes

8 Processes

9 Threads

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 (*control flow*) determining the behavior 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)

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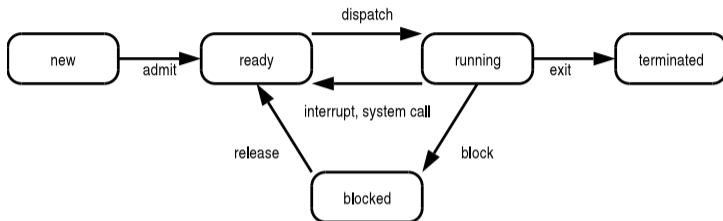
- *new*: just created, not yet admitted
- *ready*: ready to run, waiting for CPU
- *running*: executing, holds a CPU
- *blocked*: not ready to run, waiting for a resource
- *terminated*: just finished, not yet removed

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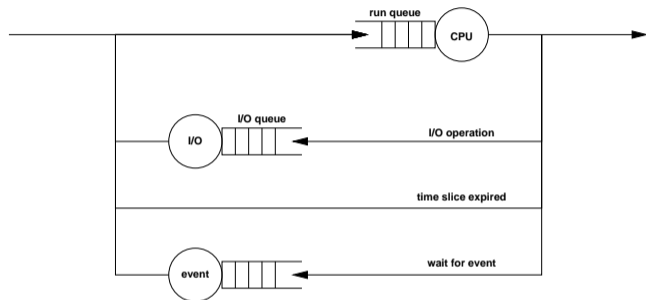
- Processes are enqueued if they wait for resources or events
- Dequeuing strategies can have strong performance impact
- Queueing models can be used for performance analysis and prediction

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

process id
process state
saved registers
scheduling info
open files
memory info
accounting info
pointers

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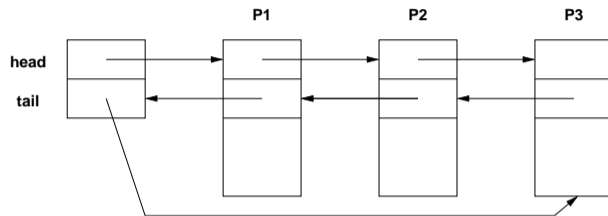
- PCBs are often organized in doubly-linked lists or tables
- PCBs can be queued easily and efficiently using pointer operations
- Run queue length of the CPUs is a good load indicator
- The system load is often defined as the exponentially smoothed average of the run queue length over 1, 5 and 15 minutes

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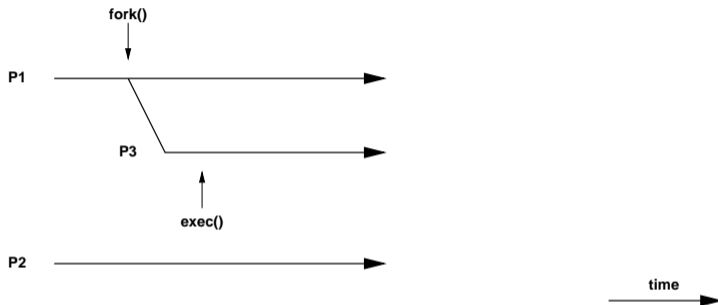
- The fork() system call creates a new child process
  - which is an exact copy of the parent process,
  - except that the result of the system call differs
- The exec() system call replaces the current process image with a new image.

Process Creation: `fork()`
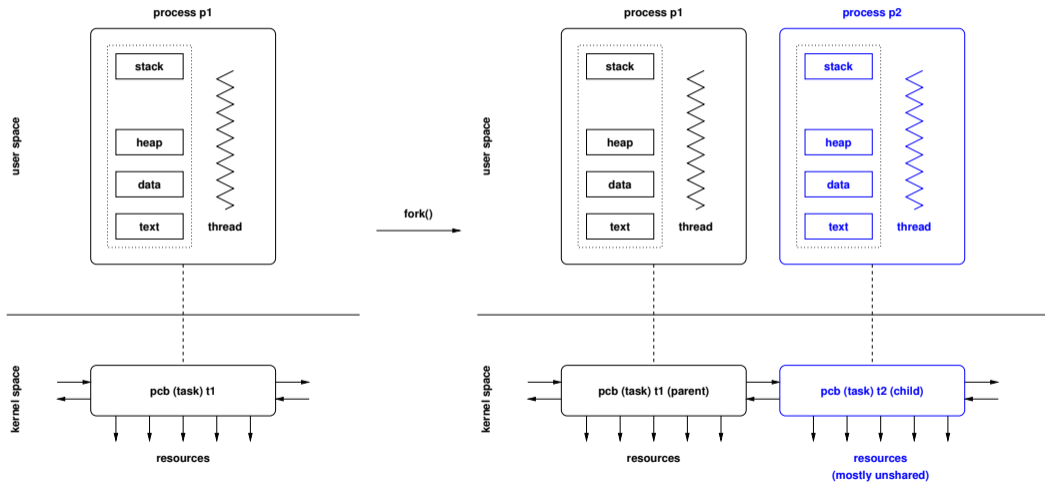
Process Image Replacement: exec()
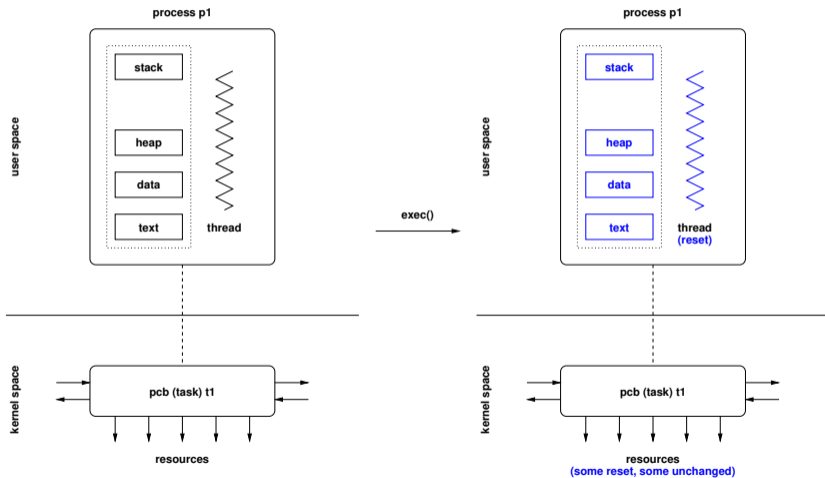
Process Termination
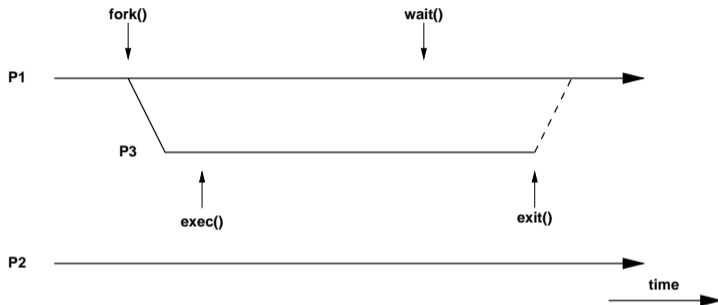
- Processes can terminate themself by calling exit()
- The wait() system call suspends execution until a child terminates (or a signal arrives)
- Terminating processes return a numeric status code

Process Trees
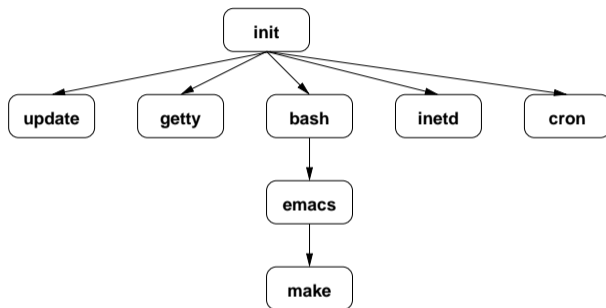
- The first process is created when the system is initialized
- All other processes are created using `fork()`, which leads to a process tree
- PCBs often contain pointers to parent PCBs

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[]);
```

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);
```

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 */
    }
}
```

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
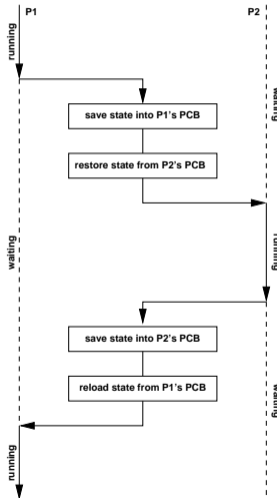
Section 9: Threads

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

Thread Creation: `pthread_create`
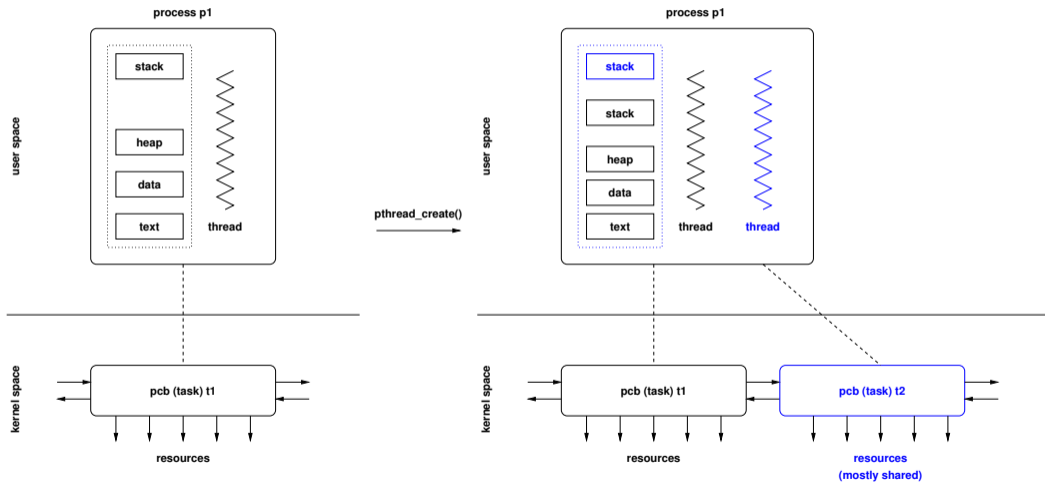
POSIX API (pthreads)

```
#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)
```

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

Part 4: Synchronization

10 Race Conditions and Critical Sections

11 Synchronization Mechanisms

12 Semaphores

13 Critical Regions, Condition Variables, Messages

14 Synchronization Pattern

15 Synchronization in C

16 Synchronization in Java and Go

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

Bounded-Buffer Problem (incorrect naive solution)

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

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

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

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*.
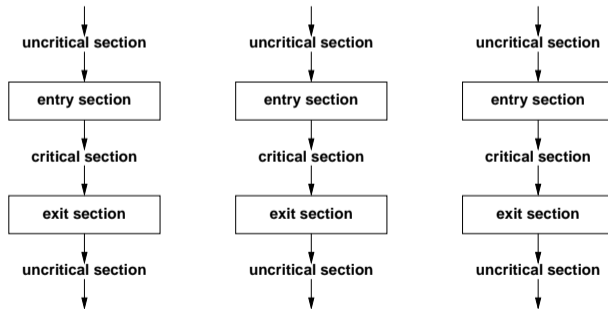
Critical-Section Problem

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

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

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();
}
```

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

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

Readers / Writers with Semaphores

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

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

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
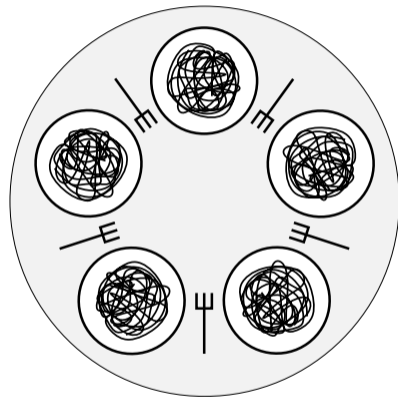
Dining Philosophers with Semaphores

```c
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)        void test(int i)
{                              {
    while (true) {                 if (state[i] == hungry
        think(i);                      && state[(i-1)%N] != eating
        take_forks(i);                 && state[(i+1)%N] != eating) {
        eat(i);                        state[i] = eating;
        put_forks(i);                  up(&s[i]);
    }                              }
}                              }
```

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);
}
```

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);
}
```

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

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

Bounded-Buffer with Monitors

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

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

Messages

- Message systems support the synchronization of processes that do not have shared memory
- Message systems can be implemented in user space and without special compiler support
- Message systems usually require that
  - messages are not lost during transmission
  - messages are not duplicated during transmission
  - addresses are unique
  - processes do not send arbitrary messages to each other
- Message systems are often slower than shared memory mechanisms
- POSIX message queues provide synchronization between threads or processes

Bounded-Buffer with Messages

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

void producer()                    void consumer()
{                                  {
    produce(&item);                    recv(&producer, &m);
    recv(&consumer, &m);               unpack(&m, &item)
    pack(&m, item);                    send(&producer, &m);
    send(&consumer, &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()              signaling_thread()
{                             {
  /* do something */            /* do something */
  down(&s);                     up(&s);
  /* do something */            /* 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);
```

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);
```

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);
```

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);
```

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);
```

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

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);
```

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);
```

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);
```

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);
```

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

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)

Part 5: Deadlocks

Section 17: Deadlocks

17 Deadlocks

18 Resource Allocation Graphs

19 Deadlock Strategies

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

Deadlocks

```
class A                                class B
{                                      {
    public synchronized a1(B b)            public synchronized b1(A a)
    {                                      {
        b.b2();                                a.a2();
    }                                      }

    public synchronized a2(B b)            public synchronized b2(A a)
    {                                      {
    }                                      }
}                                      }
```

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

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

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.

Resource-Allocation Graph (RAG)
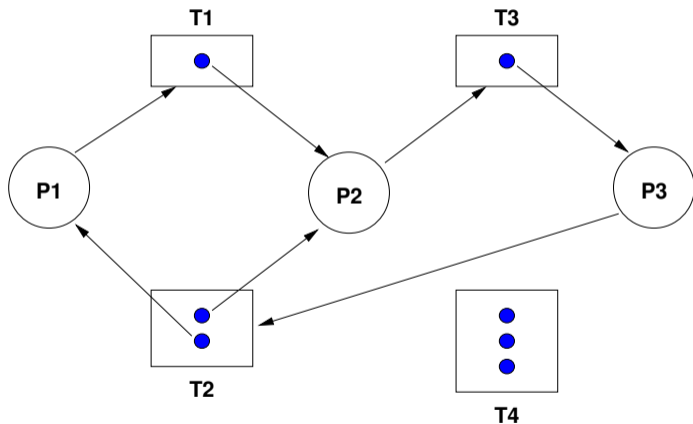
$$RAG = \{V, E\}$$
$$V = P \cup T \cup R$$
$$E = E_a \cup E_r \cup E_c$$
$$P = \{P_1, P_2, \ldots, P_n\}$$
$$T = \{T_1, T_2, \ldots, T_m\}$$
$$R = \{R_1, R_2, \ldots, R_m\}$$
$$E_a = \{R_j \rightarrow P_i\}$$
$$E_r = \{P_i \rightarrow T_j\}$$
$$E_c = \{P_i \rightarrow T_j\}$$

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
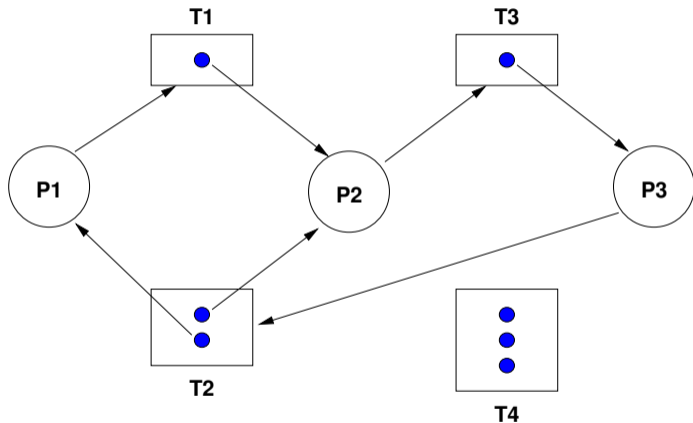
- Cycle 1:
  $P_1 \rightarrow T_1 \rightarrow P_2 \rightarrow T_3 \rightarrow$
  $P_3 \rightarrow T_2 \rightarrow P_1$
- Cycle 2:
  $P_2 \rightarrow T_3 \rightarrow P_3 \rightarrow T_2 \rightarrow P_2$
- $\{P_1, P_2, P_3\}$ are deadlocked
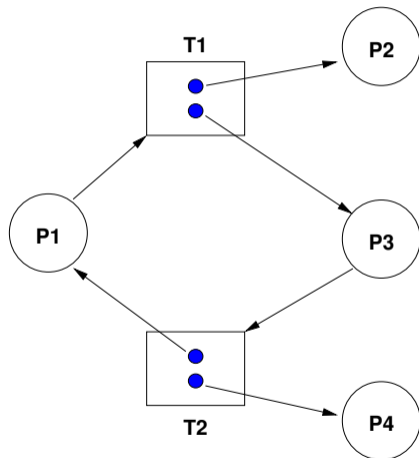
- Cycle:
  $P_1 \rightarrow T_1 \rightarrow P_3 \rightarrow T_2 \rightarrow P_1$
- $\{P_1, P_3\}$ are not deadlocked
- P_4 may break the cycle by releasing its instance of T_2
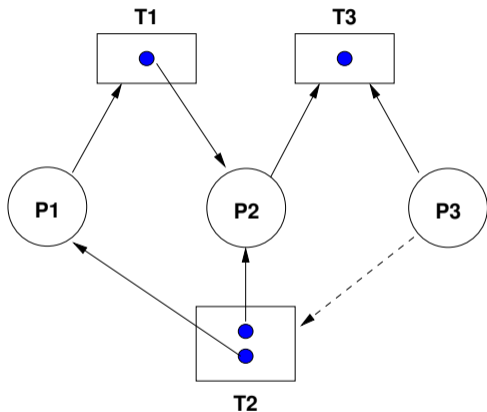
- P_2 and P_3 both request T_3
- To which process should the resource be assigned?
- Assigning an instance of T_3 to P_2 avoids a future deadlock

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*
$a \in \mathbb{N}^m$	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*

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
```

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
```

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
```

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

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.

Part 6: Scheduling

Section 20: Scheduler

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.

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
- Perfect is the enemy of the good
- 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

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)

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
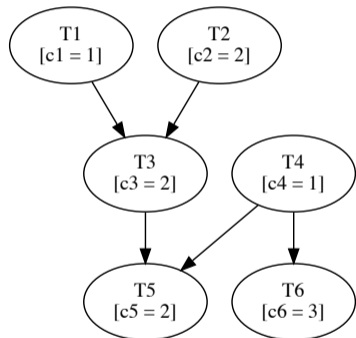
$T = \{T_1, T_2, T_3, T_4, T_5, T_6\}$

$n = 6$

$c = (c_1, c_2, c_3, c_4, c_5, c_5) = (1, 2, 2, 1, 2, 3)$

$G = (T, E)$

$E = \{(T_1, T_3), (T_2, T_3), (T_3, T_5),$
$\quad\quad (T_4, T_5), (T_4, T_6)\}$

$P = \{P_1, P_2\}$

$m = 2$

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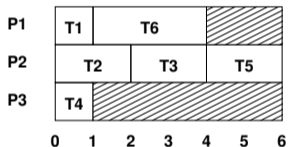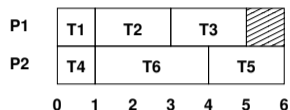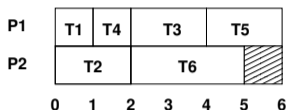
- Both schedules for $m = 2$ processors have the same length and the same average turnaround and waiting times.
- The schedule for $m = 3$ processors has the same length but different average turnaround and waiting times.

Section 21: Scheduling Strategies

20 Scheduler

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:

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

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

$\implies$ Tradeoff between responsiveness and throughput

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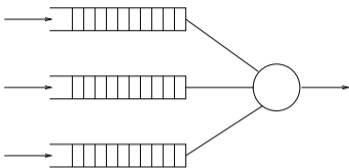
- Principle:
  - Multiple queues for processes with different priorities
  - Processes are permanently assigned to a queue
  - Each queue has its own scheduling algorithm
  - Additional scheduling between the queues necessary
- Properties:
  - Overall queue scheduling important (static vs. dynamic partitioning)

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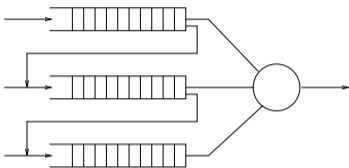
- Principle:
  - Multiple queues for processes with different priorities
  - Processes can move between queues
  - Each queue has its own scheduling algorithm
- Properties:
  - Very general and configurable scheduling algorithm
  - Queue up/down grade critical for overall performance

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 7: Linking

Section 22: Linker

Stages of the C Compilation Process

```
source code    -> cpp -> expanded code    (gcc -E hello.c)
                              v
    .----------------------'
    v
expanded code  -> cc  -> assembler code   (gcc -S hello.c)
                              v
    .----------------------'
    v
assembler code -> as  -> object code      (gcc -c hello.c)
                              v
    .----------------------'
    v
object code    -> ld  -> executable       (gcc hello.c)
```
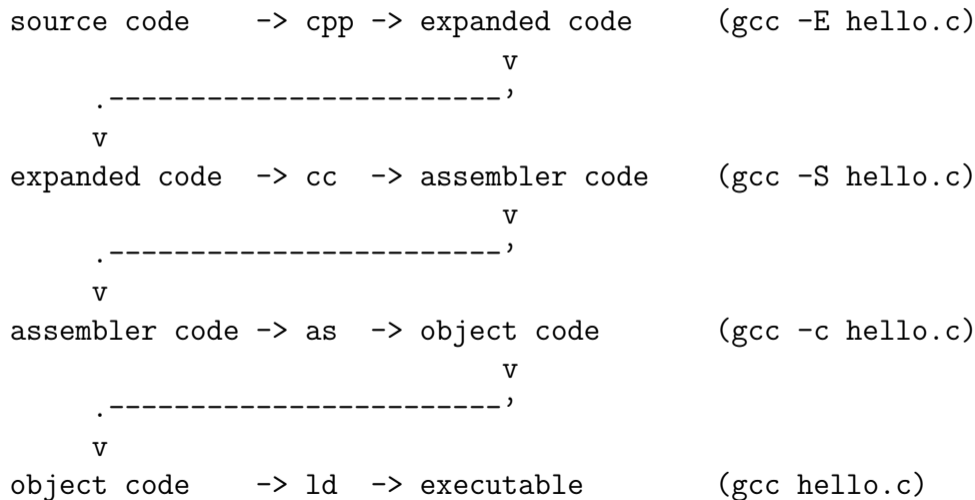
Reasons for using a Linker

- Modularity
  - Programs can be we written as a collection of small files
  - Creating collections of easily reusable functions
- Efficiency
  - Separate compilation of a subset of small files saves time on large projects
  - Smaller executables by linking only functions that are actually used

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

Object Code File Types

- Relocatable object files (.o files)
  - Contains code and data in a form that can be combined with other relocatable object files
- Executable object files
  - Contains code and data in a form that can be loaded directly into memory
- Shared object files (.so files)
  - Special type of relocatable object file that can be loaded into memory and linked dynamically at either executable load time or at run-time

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

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

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

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
  ```

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

Dynamic Linking Loader API

```
#include <dlfcn.h>

#define ... RTLD_LAZY        /* resolve symbols lazily when needed */
#define ... RTLD_NOW         /* resolve all symbols at load time */
#define ... RTLD_GLOBAL      /* make symbols globally available */
#define ... RTLD_LOCAL       /* keel symbols local to the library */

void *dlopen(const char *filename, int flag);
char *dlerror(void);
void *dlsym(void *handle, const char *symbol);
int dlclose(void *handle);
```

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.

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
  ```

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 */
```

Part 8: Memory

Section 25: Translation of Memory Addresses

Memory Sizes and Access Times

Memory Size		Access Time
< 1 KiB	CPU Registers	< 1 ns
~ 128 KiB	Level 1 Cache	~ 1–2 ns
~ 1 MiB	Level 2 Cache	~ 4 ns
> 1 GiB	Main Memory	~ 8 ns
> 128 GiB	Disks (SSD or HDD)	~ 1–4 ms

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

Translation of Memory Addresses
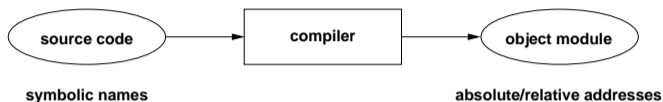
- Compiler translates variable / function names into absolute or relative addresses
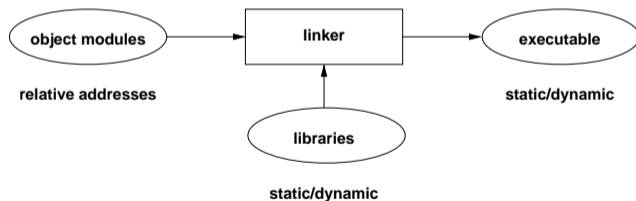
- Linker binds multiple object modules and libraries into an executable

Translation of Memory Addresses
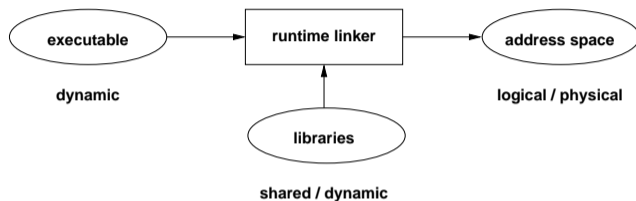
- Runtime linker loads dynamic (shared) libraries at program startup time
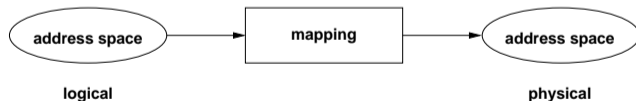
- Memory management unit maps logical addresses to physical addresses

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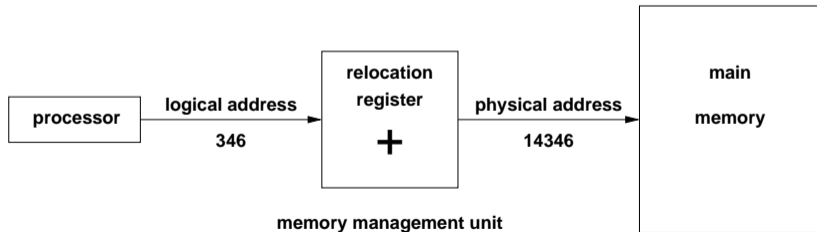
Memory Partitioning
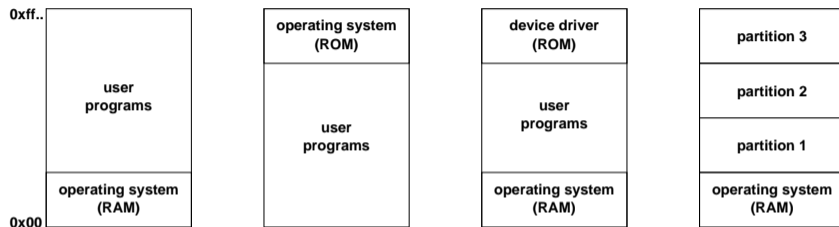
- Memory space is often divided into several regions or partitions, some of them serve special purposes
- Multiple processes can be held in memory (as long as they fit)
- Partitioning is not very flexible (but may be good enough for embedded systems)

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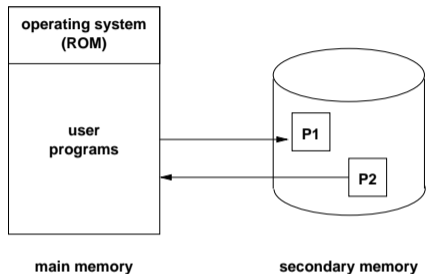
- Address space of a process is moved to a big (but slow) secondary storage system
- Swapped-out processes should not be considered runable by the scheduler
- Often used to handle (temporary) memory shortages

Section 26: Segmentation

Segmentation Overview
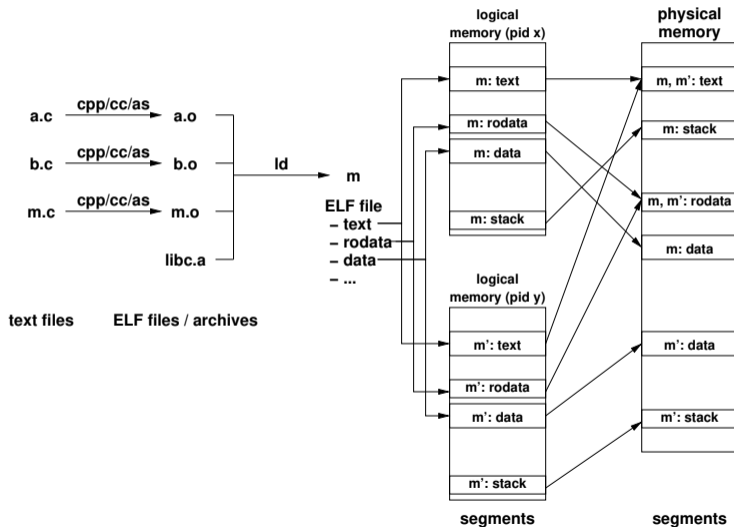
Segmentation

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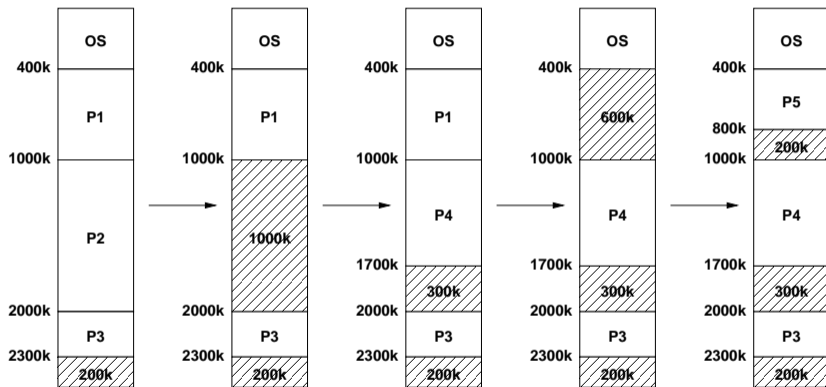
- In the general case, there is more than one suitable hole to hold a new segment — which one to choose?

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

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()`)

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					
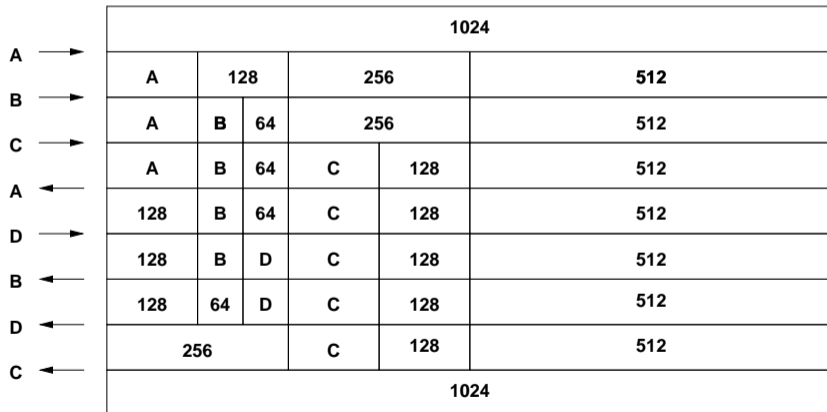
Segmentation Analysis

- *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:

$$f = \frac{k}{k + 2}$$

Fraction of unsused memory as a function of the relative hole size
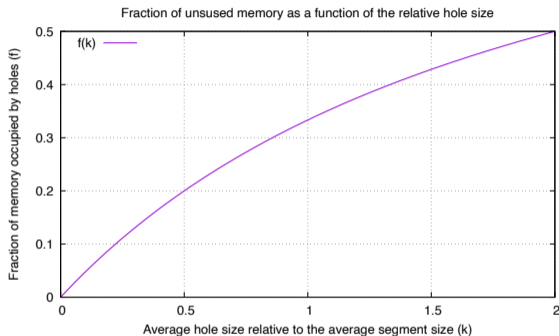
$\implies$ As long as the average hole size is a significant fraction of the average segment size, a substantial amount of memory will be wasted

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
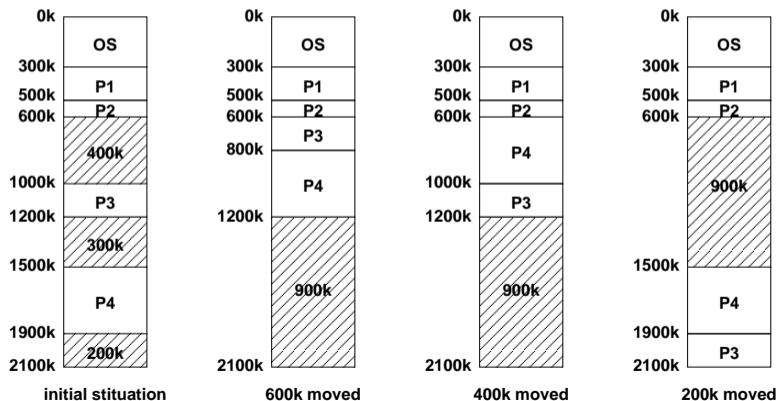
initial stituation 600k moved 400k moved 200k moved

Section 27: Paging

Paging Overview
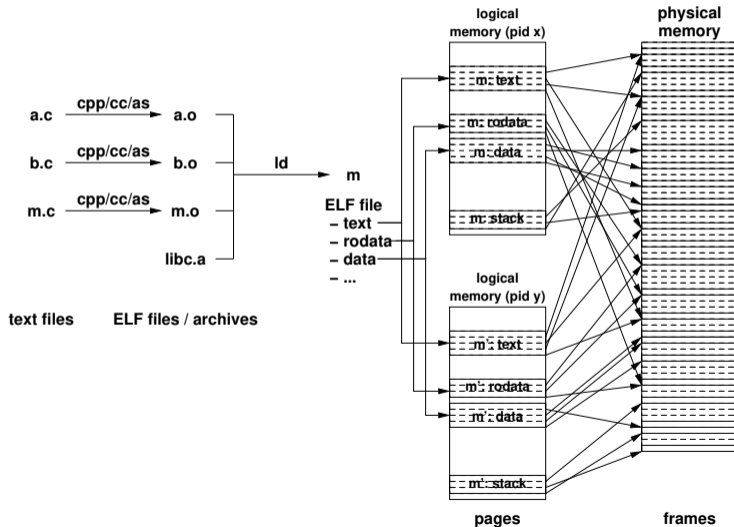
Paging Idea

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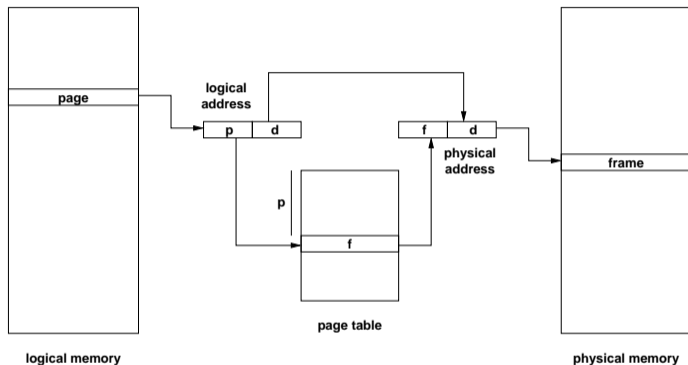
- A logical address is a tuple (p, d) where p is an index into the page table and d is an offset within page p
- A physical address is a tuple (f, d) where f is the frame number and d is an offset within frame f

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

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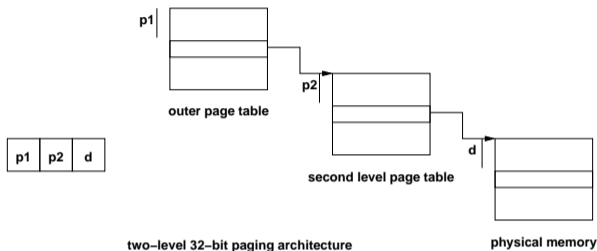
- Paging can be applied to page tables as well
- SPARC 32-bit architecture supports three-level paging
- Motorola 32-bit architecture (68030) supports four-level paging
- Caching essential to alleviate delays introduced by multiple memory lookups

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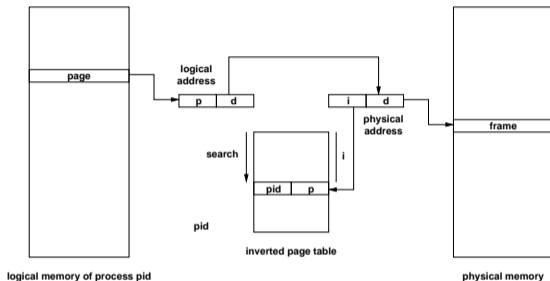
- The inverted page table has one entry for each frame
- Page table size determined by size of physical memory
- Entries contain page address and process identification
- The non-inverted page table is stored in paged memory
- Lookups require to search the inverted page table

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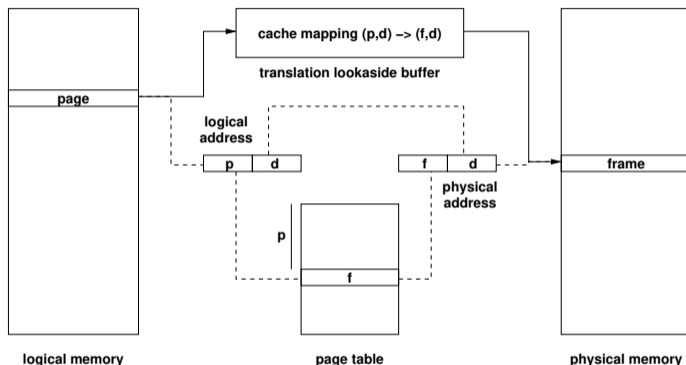
- A TLB acts as a cache mapping logical addresses (p, d) to physical addresses (f, d)
- TLB lookup failures may be handled by the kernel in software

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 that are much larger than the physical address space available
    - 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 loaded pages

$s[t]$ set of pages loaded at time t

$x[t]$ pages paged in at time t

$y[t]$ 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 Anomaly (FIFO Replacement Strategy)

```
| string  | 1 2 3 4 1 2 5 1 2 3 4 5 |      | 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 0 | 1 1 1 1 1 1 5 5 5 5 4 4 |
| frame 1 |   2 2 2 1 1 1 1 1 3 3 3 |      | frame 1 |   2 2 2 2 2 2 1 1 1 1 5 |
| frame 2 |     3 3 3 2 2 2 2 2 4 4 |      | 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     |      |---------+-------------------------|
                                            | 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.

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
- For every stack algorithm, the following holds true:

$$S_m(w) \subseteq S_{m+1}(w)$$

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
  - 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
  - Page number at the bottom identifies LRU page
- In general 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 must 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?

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 which belong to the working-set of a running process are not removed from memory
- Example ($T = 10$):

$$w = \ldots \underline{2, 6, 1, 5, 7, 7, 7, 7, 5, 1}, 6, 2, 3, 4, 1, 2, \underline{3, 4, 4, 4, 3, 4, 3, 4, 4, 4}, 1, 3, 2, 3, 4, 3, \ldots$$

$$W_p(t_1, 10) = \{1, 2, 5, 6, 7\} \qquad\qquad W_p(t_2, 10) = \{3, 4\}$$

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);
```

Part 9: Communication

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 IPC 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 handling behavior
- If in doubt, use the POSIX signal API to make code portable

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 */
```

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
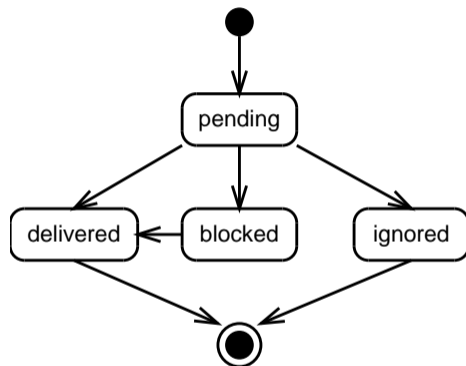
POSIX Signal API

```
#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 */
};
```

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);
```

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 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

Signal Pattern: Flagging Behaviour Changes

```c
#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;
}
```

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 */
}
```

Section 30: Pipes
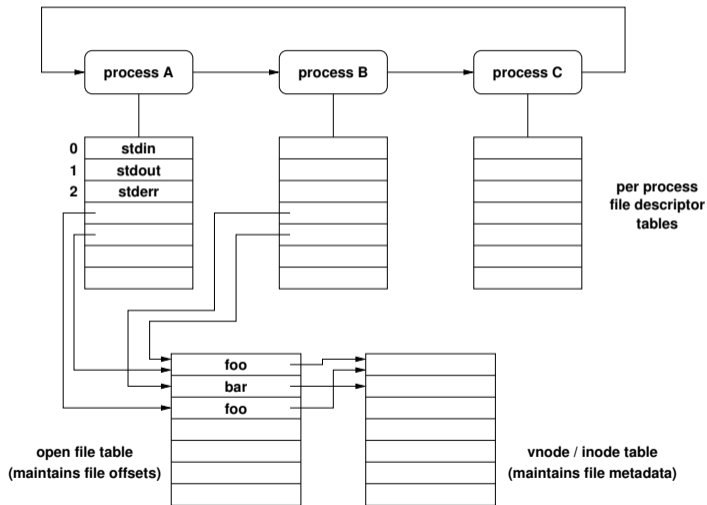
Pipes at the Shell Command Line
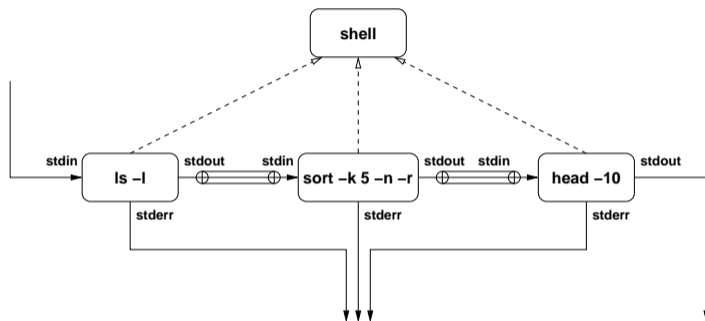

```
# list the 10 largest files in the
# current directory
ls -l | sort -k 5 -n -r | head -10
```

POSIX Pipes

```c
#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

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
```

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

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

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 */
};
```

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 */
};
```

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 */
};
```

Mapping Names to Addresses 1/2

```c
#include <sys/types.h>
#include <sys/socket.h>
#include <netdb.h>

struct addrinfo {
    int             ai_flags;
    int             ai_family;
    int             ai_socktype;
    int             ai_protocol;
    size_t          ai_addrlen;
    struct sockaddr *ai_addr;
    char            *ai_canonname;
    struct addrinfo *ai_next;
};
```

```
#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);
```

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);
```

Connection-Less Communication
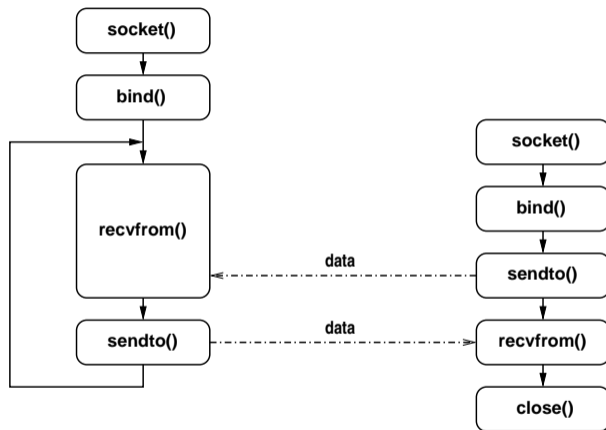
Connection-Oriented Communication
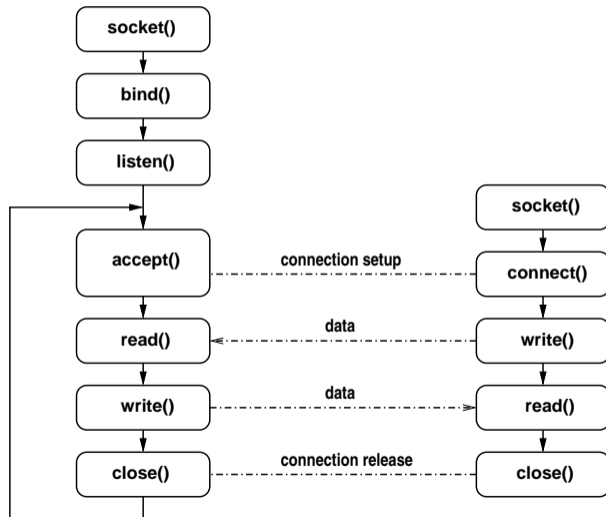
Socket API Summary 1/3

```
#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);
```

```
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);
```

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);
```

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

Part 10: File Systems

32 General File System Concepts

33 File System Programming Interface

34 File System Implementation

Section 32: General 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

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

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
- ⟹ Embedded operating systems sometimes only support flat file name spaces, or only read-only file systems, or no file systems at all

Unix Directory Structure
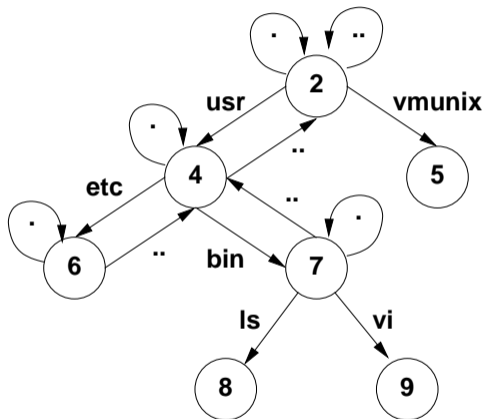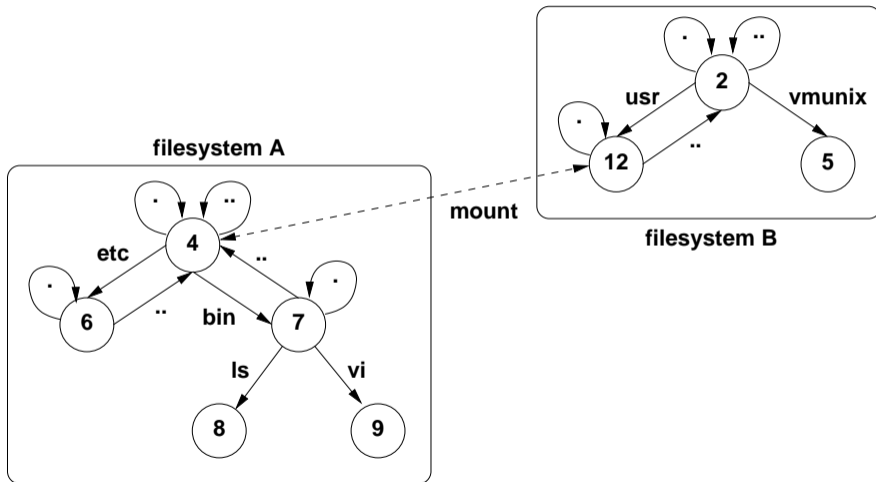
Hard Links and Soft Links (Symbolic Links)

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

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
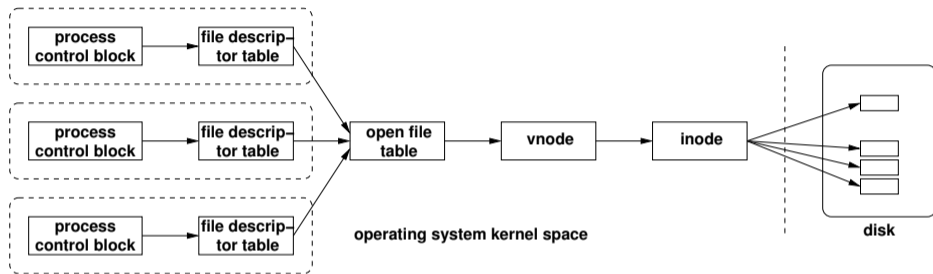
- Every process control block maintains a pointer to the file descriptor table
- File descriptor table entries point to an entry in the open file table
- Open file table entries point to virtual inodes (vnodes)
- The vnode points to the inode (if it is a local file)

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)

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);
```

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);
```

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);
```

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 */
```

```
#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, ...);
```

Memory Mapped Files

```c
#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`.

Section 34: File System Implementation

Block Allocation Methods using Lists

- Linked list allocation example:
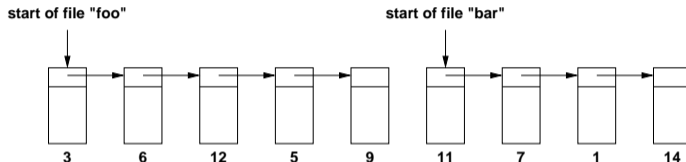
- Indexed linked list allocation example:
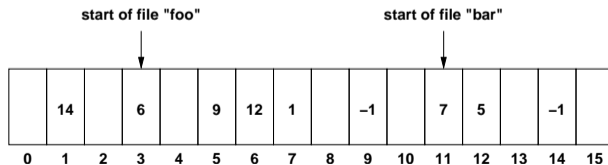
Block Allocation Method using Index Nodes
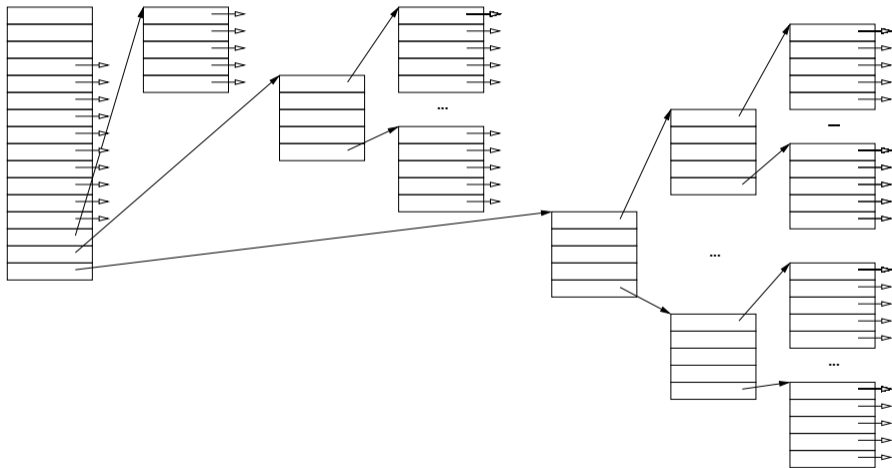
Index Node File System Example
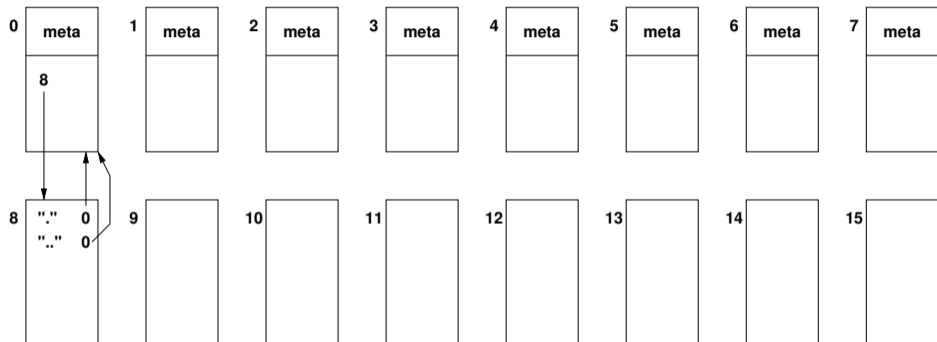
- Block device with 16 equal-sized blocks (numbered 0...15)
- Blocks (0...7) are reserved for inodes, the remaining blocks are data blocks
- The root directory is always found in inode 0

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)
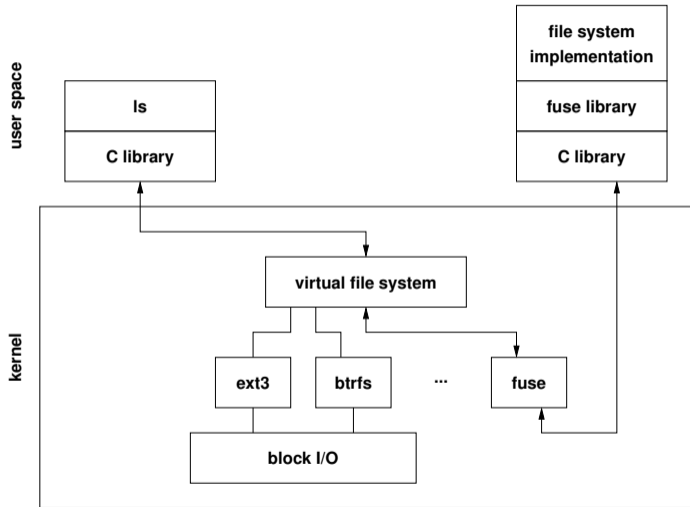
- Abstract (virtual) file system interface
- Simplifies support for many different file systems
- Common functions implemented at the virtual file system interface
- Concrete file systems may reside in user space or on remote systems

Part 11: Devices

Section 35: Goals and Design Considerations

Design Considerations

- Device Independence
  - User space applications should work with as many similar devices as possible without requiring any changes
  - Some user space applications may want to exploit specific device characteristics
  - Be as generic as possible while allowing applications to explore specific features of certain devices
- Efficiency
  - Efficiency is of great concern since many applications are I/O bound and not CPU bound
- Error Reporting
  - I/O operations have a high error probability and proper reporting of errors to applications and system administrators is crucial

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

Efficiency: I/O Programming Styles

- *programmed input/output*:
  The CPU does everything (copying data to/from the I/O device) and blocks until I/O is complete

- *interrupt-driven input/output*:
  Interrupts drive the I/O process, the CPU can do other things while the device is busy

- *direct-memory-access input/output*:
  A DMA controller moves data in/out of memory and notifies the CPU when I/O is complete, the CPU does not need to process any interrupts during the I/O process

Error Reporting

- Provide a consistent and meaningful (!) way to report errors and exceptions to applications (and to system administrators)
- This is particularly important since I/O systems tend to be error prone compared to other parts of a computer
- On POSIX systems, system calls report errors via special return values and a (thread) global variable `errno` (`errno` stores the last error code and does not get cleared when a system call completes without an error)
- Runtime errors that do not relate to a specific system call are reported to a logging facility, usually via `syslog` on Unix systems

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.

- Devices are identified by their
  - type (block or character device)
  - major device number, which identifies the responsible device driver
  - minor device number, which identifies the device instance handled by the device driver

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)
  - Memory unit emulates hard disk interface
- Optical disks (CD, DVD, Blu-ray):
  - Read-only vs. recordable vs. rewritable
  - Very robust and relatively cheap
  - Division into tracks, sectors and cylinders
- Magnetic tapes (or tesa tapes):
  - Used mainly for backups and archival purposes
  - Not further considered in this lecture

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

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)

- 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

- *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)
- *Volume Group*: A volume group pools several physical volumes into one logical unit
- *Logical Volume*: A logical volume resides in a volume group and provides a block device, which can be used to create a file system

$\implies$ Separation of the logical storage layout from the physical storage layout

$\implies$ Simplifies modification of logical volumes (create, remove, resize, snapshot)

Logical Volume Management (Linux)


```
PV ───┐
       ├─ VG ─── LV        (e.g. root filesystem)
PV ───┘        └ LV        (e.g. swap space)
                 ...
```

```
PV ───┐
PV ───┤
       ├─ VG ─── LV        (e.g. home filesystem)
PV ───┤        ├ LV        (e.g. data filesystem)
PV ───┘        ├ LV        (e.g. virtual machine root filesystem)
               └ LV        (e.g. virtual machine swap space)
                 ...
```
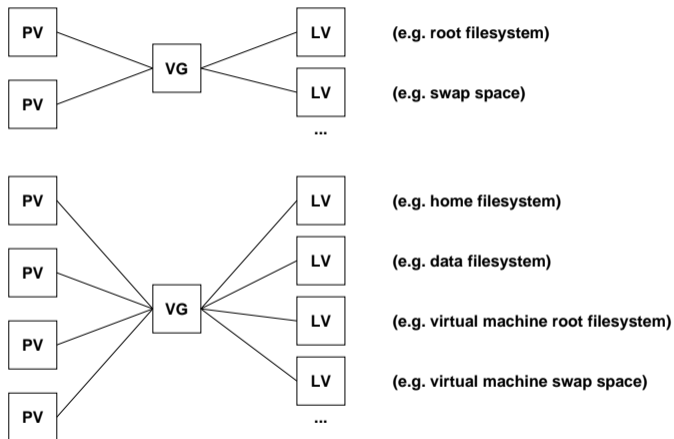
PV = physical volume, VG = volume group, LV = logical volume

Networked Storage

- Storage Area Networks (SAN)
  - A storage area network detaches block devices from computer systems through a fast communication network
  - Simplifies the sharing of storage between (frontend) computers
  - Dedicated network protocols (Fibre Channel, iSCSI, . . .)
  - Relative expensive technology
- Network Attached Storage (NAS)
  - Access to a logical file system over the network
  - Sharing of file systems between multiple computers over a network
  - Many different protocols: NFS, SMB/CIFS, . . .

Section 38: Terminal Devices

Traditional Character Terminal Devices
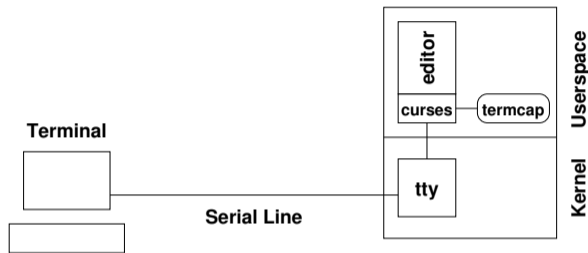
- Character terminals were connected via serial lines
- The device driver in the kernel represents the terminal to user space programs (via a tty device file)
- Applications often use a library that knows about terminal capabilities to achieve terminal device independence

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

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, and the `TERM` environment variable selects the terminal and thus the control sequences to use
- Network terminals use the same mechanisms but are represented as pseudo tty devices, usually called `ptys`.

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.

Pseudo Terminal Devices
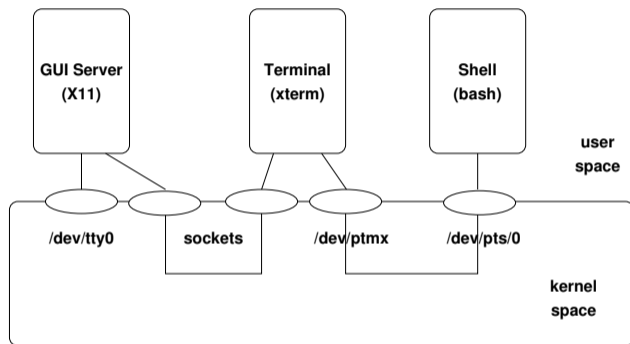
- The terminal device (`/dev/pts/0`) behaves like a traditional terminal device
- The pseudoterminal device (obtained by opening the pseudoterminal device pair multiplexer `/dev/ptmx`) controls the interaction with the terminal device

Part 12: Virtualization

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.

Emulation

- Emulation of processor architectures on different platforms
  - Transition between architectures (e.g., PPC $\Rightarrow$ Intel $\Rightarrow$ 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

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/`)
  - . . .
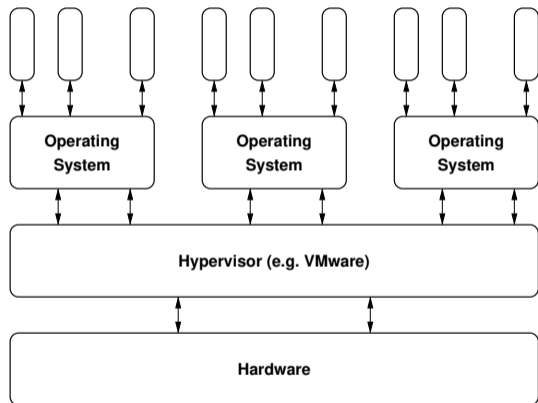
Example: VMware

- VMware (USA)
- 1998 VMware founded
- VMware ESXi (Hypervisor)
- VMware vSphere
- VMware Workstation
- closed source

Example: KVM
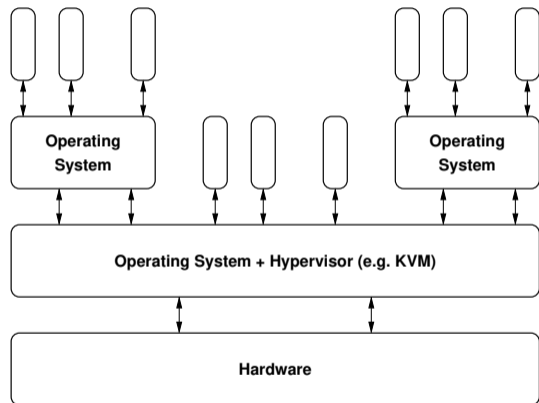
- Qumranet (Israel)
- 2007 integration in Linux
- 2008 bought by Red Hat
- OS Kernel Extension
- QEMU for device emulation
- OpenStack, Amazon, . . .

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/`)
  - . . .
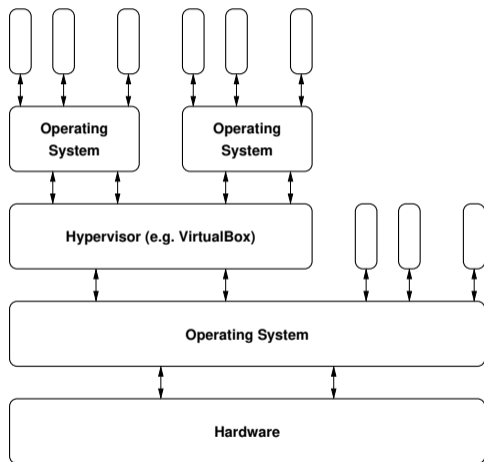
- InnoTek (Germany)
- 2007 open source (GPL)
- 2008 bought by Sun
- 2010 bought by Oracle
- core open source (GPL)
- extensions closed source
- Linux, Solaris
- Windows, MacOS

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/)

Example: Xen
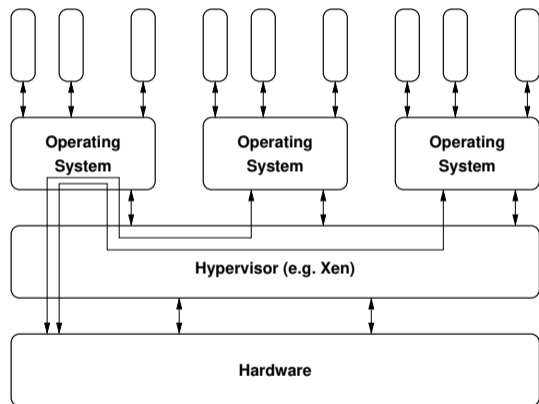
- University of Cambridge (UK)
- 2003 release 1.0 open source
- 2004 XenSource founded
- 2007 bought by Citrix Systems
- 2013 Linux Foundation
- Mircokernel Design

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

Operating System + Isolation Mechanisms (e.g. LXC)

Hardware
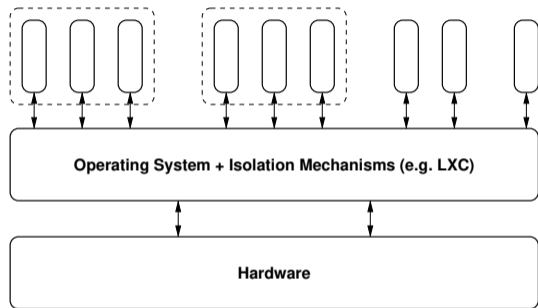
- 2008 initial release
- open source (LGPL, GPL)

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
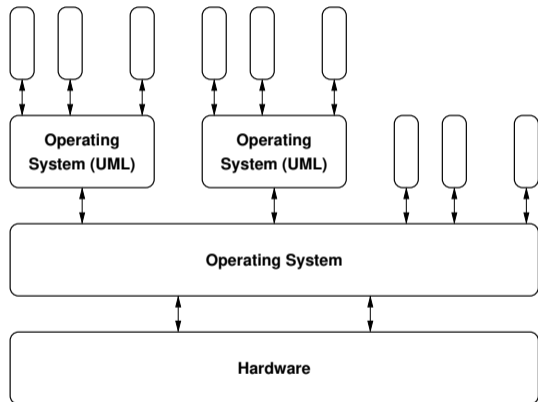
- 2003 integration in Linux

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

Linux Control Groups

A control group (cgroup) is a collection of processes that are bound by 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

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

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.

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.

Part 13: Distributed Systems

Section 42: Definition and Models

What is a Distributed System?

- A distributed system is one in which the failure of a computer you didn't even know existed can render your own computer unusable. (Lesley Lamport, 1992)

- A distributed system is several computers doing something together. (M.D. Schroeder, 1993)

- An interconnected collection of autonomous computers, processes, or processors. (G. Tel, 2000)

- A distributed system is a collection of processors that do not share memory or a clock. (A. Silberschatz, 1994)

Why Distributed Systems?

- Information exchange
- Resource sharing
- Increased reliability through replication
- Increased performance through parallelization
- Simplification of design through specialization
- Cost reduction through open standards and interfaces

Challenges

General challenges for the design of distributed systems:

- Efficiency
- Scalability
- Security
- Fairness
- Robustness
- Transparency
- Openness

Special design challenges (increasingly important):

- Context-awareness and energy-awareness

Distributed vs. Centralized

- *Lack of knowledge of global state*
  Nodes in a distributed system have access only to their own state and not to the global state of the entire system

- *Lack of a global time frame*
  The events constituting the execution of a centralized algorithm are totally ordered by their temporal occurance. Such a natural total order does not exist for distributed algorithms

- *Non-determinism*
  The execution of a distributed system is usually non-deterministic due to speed differences of system components

Client-Server Model
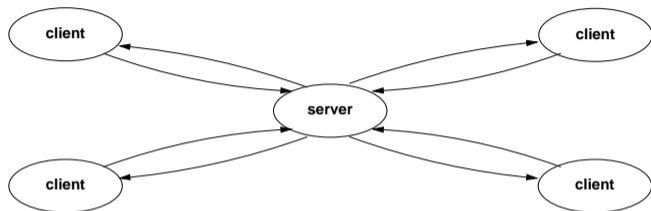
- Clients requests services from servers
- Synchronous: clients wait for the response before they proceed with their computation
- Asynchronous: clients proceed with computations while the response is returned by the server
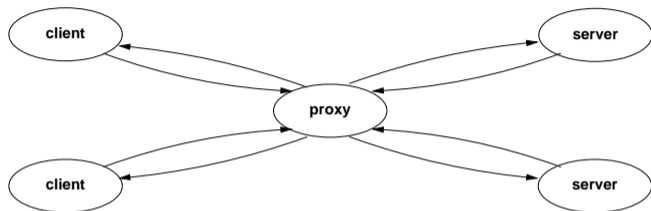
Proxies

- Proxies can increase scalability
- Proxies can increase availability
- Proxies can increase protection and security
- Proxies and help solving versioning issues

Peer-to-Peer Model
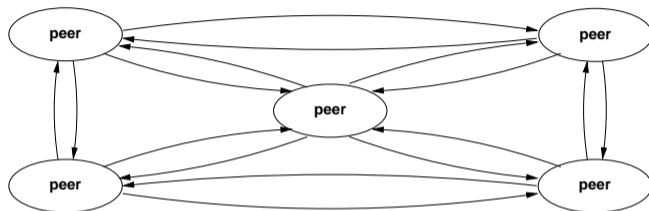
- Every peer provides client and server functionality
- Avoids centralized components
- Able to establish new (overlay) topologies dynamically
- Requires control and coordination logic on each node
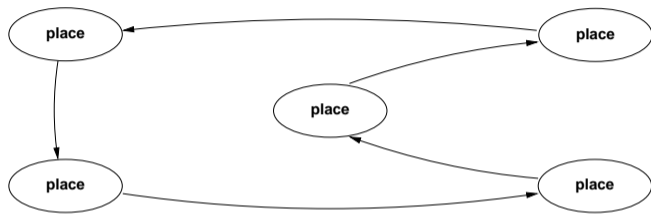
Mobile Code

- Executable code (mobile agent) travels autonomously through the network
- At each place, some computations are performed locally that can change the state of the mobile agent
- A mobile agent must be able to find a good trajectory
- Security (protection of places, protection of agents) is a challenging problem
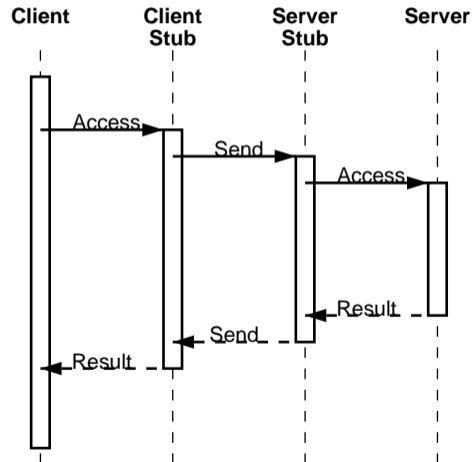
Section 43: Remote Procedure Calls
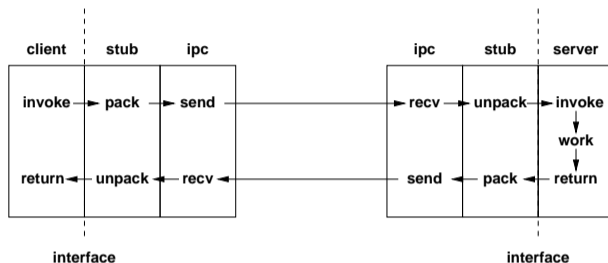
Remote Procedure Call Model

- Introduced by Birrel and Nelson (1984)
  - to provide communication transparency and
  - to overcome heterogeneity
- Stubs hide all communication details
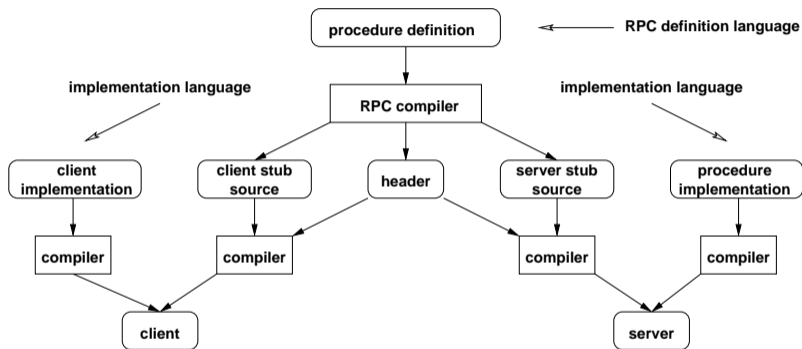
Stub Procedures

- Client stubs provide a local interface which can be called like any other local procedure
- Server stubs provide the server interface which calls the server's implementation of the procedure provided by a programmer and returns any results back to the client

Marshalling

- Marshalling is the technical term for transferring data structures used in remote procedure calls from one address space to another
- Serialization of data structures for transport in messages
- Conversion of data structures from the data representation of the calling process to that of the called process
- Pointers can be handled to some extend by introducing call-back handles, which can be used to make an call-back RPCs from the server to the client in order to retrieve the data pointed to

RPC Definition Languages

- Formal language to define the type signature of remote procedures
- RPC compiler generates client / server stubs from the formal remote procedure definition

RPC Binding

- A client needs to locate and bind to a server in order to use RPCs
- This usually requires to lookup the transport endpoint for a suitable server in some sort of name server:
  1. The name server uses a well know transport address
  2. A server registers with the name server when it starts up
  3. A client first queries the name server to retrieve the transport address of the server
  4. Once the transport address is known, the client can send RPC messages to the correct transport endpoint

RPC Semantics

- *May-be*:
  - Client *does not* retry failed requests
- *At-least-once*:
  - Client *retries* failed requests, server re-executes the procedure
- *At-most-once*:
  - Client *may* retry failed requests, server detects retransmitted requests and responds with cached reply from the execution of the procedure
- *Exactly-once*:
  - Client *must* retry failed requests, server detects retransmitted requests and responds with cached reply from the execution of the procedure

Local vs. Remote Procedure Calls

- Client, server and the communication channel can fail independently and hence an RPC may fail
- Extra code must be present on the client side to handle RPC failures gracefully
- Global variables and pointers can not be used directly with RPCs
- Passing of functions as arguments is close to impossible
- The time needed to call remote procedures is orders of magnitude higher than the time needed for calling local procedures

Open Network Computing RPC

- Developed by Sun Microsystems (Sun RPC), originally published in 1987/1988
- Since 1995 controlled by the IETF (RFC 1790)
- ONC RPC encompasses:
  - ONC RPC Language (RFC 5531)
  - ONC XDR Encoding (RFC 4506)
  - ONC RPC Protocol (RFC 5531)
  - ONC RPC Binding (RFC 1833)
- Foundation of the Network File System (NFS) and widely implemented on Unix systems

Section 44: Distributed File Systems

Distributed File Systems

- A *distributed file system* is a part of a distributed system that provides a user with a unified view of the files on the network
- Transparancy features (not necessarily all supported):
  - Location transparancy
  - Access transparancy
  - Replication transparancy
  - Failure transparancy
  - Mobility transparancy
  - Scaling transparancy
- Recently: File sharing (copying) via peer-to-peer protocols

Design Issues

- Centralized vs. distributed data
  - Consistency of global file system state
  - If distributed, duplications (caching) or division
- Naming
  - Tree vs. Directed Acyclic Graph (DAG) vs. Forest
  - Symbolic links (file system pointers)
- File sharing semantics
  - Unix (updates are immediately visible)
  - Session (updates visible at end of session)
  - Transaction (updates are bundled into transactions)
  - Immutable (write once, change never)
- Stateless vs. stateful servers

Stateless vs. Stateful Servers

- Stateless Server:
  + Fault tolerance
  + No open/close needed (less setup time)
  + No data structures needed to keep state
  + No limits on open files
  + Client crashes do not impact the servers
- Stateful Server:
  + Shorter request messages
  + Better performance with buffering
  + Readahead possible
  + Idempotency is easier to achieve
  + File locking possible

Network File System Version 3

- Original Idea:
  - Wrap the file system system calls into RPCs
  - Stateless server, little transparency support
  - Unix file system semantics
  - Simple and straight-forward to implement
  - Servers are dumb and clients are smart
- Stateless server
- Mount service for mounting/unmounting file systems
- Additional locking service (needs to be stateful)
- NFSv3 is defined in RFC 1813 (June 1995)

Operating System Integration
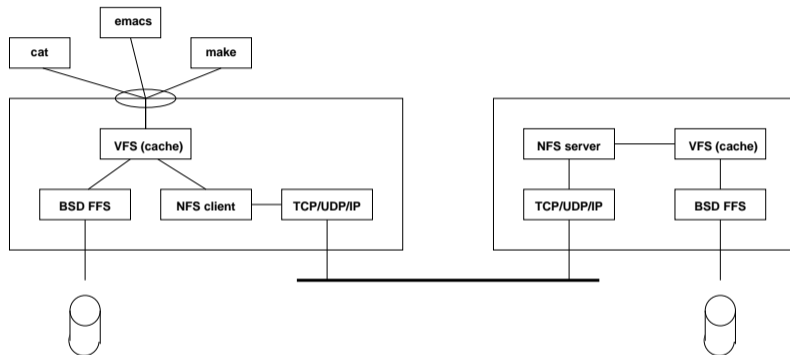
- Early implementations used user-space deamons
- NFS runs over UDP and TCP, currently TCP is preferred
- NFS uses a fixed port number (no portmapper involved)

NFSv3 Example (Simplified!)

```
C: PORTMAP GETPORT mount          # mount bayonne:/export/vol0 /mnt
S: PORTMAP GETPORT port
C: MOUNT /export/vol0
S: MOUNT FH=0x0222
C: PORTMAP GETPORT nfs            # dd if=/mnt/home/data bs=32k \
S: PORTMAP GETPORT port           # count=1 of=/dev/null
C: FSINFO FH=0x0222
S: FSINFO OK
C: GETATTR FH=0x0222
S: GETATTR OK
C: LOOKUP FH=0x0222 home
S: LOOKUP FH=0x0123
C: LOOKUP FH=0x0123 data
S: LOOKUP FH=0x4321
C: ACCESS FH=0x4321 read
S: ACCESS FH=0x4321 OK
C: READ FH=0x4321 at 0 for 32768
S: READ DATA (32768 bytes)
```

Related Work

- Distributed File Systems:
  - Network File System Version 4 (NFSv4) (2003)
  - Common Internet File System (CIFS) (2002)
  - Andrew File System (AFS) (1983)
  - . . .
- Distributed File Sharing:
  - BitTorrent (2001)
  - Gnutella (2000)
  - Napster (1999)
  - . . .

Section 45: Distributed Message Queues

Typical Design Goals for Distributed Systems

- Distributed systems should be asynchronous (avoid blocking)
- Distributed systems should be designed to tolerate failures
- Distributed workflows should be adaptable at runtime (scaling up, scaling down)
- Distributed systems should be programming language agnostic
- Distributed systems should be deployable in a wide range of configurations (ranging from all components on a single system to all components distributed over many systems)
- Distributed systems should be designed to support program analysis and debugging

Message Passing and Message Queuing Frameworks

- Advanced Message Queuing Protocol (AMQ) is an open standard application layer protocol for message-oriented middleware (core developed in 2004-2006)
- ZeroMQ (ØMQ) is an asynchronous messaging library for distributed and concurrent applications. It provides message queues and it be used without a dedicated message broker (core developed in 2007-2011, written in C++)
- nanomsg is a is a high-level socket library that provides several common communication patterns that can be used over several transport mechanisms (developed since 2011, written in C)
- MQTT ...