# Operating Systems

Jürgen Schönwälder

December 5, 2019

**Abstract**

This memo contains annotated slides for the course “Operating Systems”. This is largely work in progress since annotating a large collection of slides is an effort that takes time.

## Contents

**I Introduction** 4

Definition and Requirements / Services 5

Types of Operating Systems 13

Operating System Architectures 21

**II Hardware** 26

Common Computer Architecture 27

I/O Systems and Interrupts 31

Memory 38

**III Processes and Threads** 46

Fundamental Concepts 47

Processes 52

Threads 67

**IV Synchronization** 73

Race Conditions and Critical Sections 74

Basic Synchronization Mechanisms 81

Semaphores 87

Semaphore Pattern 98

Critical Regions, Condition Variables, Messages 106

Synchronization in Java and Go and POSIX APIs 114

V Deadlocks 131

Deadlocks 132

Resource Allocation Graphs 136

Deadlock Strategies 142

VI Scheduling 157

CPU Scheduling 158

CPU Scheduling Strategies 165

VII Linking 178

Linker 179

Libraries 188

Interpositioning 191

VIII Memory Management 197

Memory Systems and Translation of Memory Addresses 198

Segmentation 206

Paging 216

Virtual Memory 226

IX Inter-Process Communication 239

Signals 241

Pipes 252

Sockets 263

X File Systems 278

General File System Concepts 279

File System Programming Interface 288

File System Implementation 294

XI Input/Output and Devices 302

Goals and Design Considerations 303

Storage Devices and RAIDs 309

Storage Virtualization 318

Terminal Devices 322

XII Virtual Machines 328

Terminology 329

XIII Distributed Systems 336

Definition and Models 337

Remote Procedure Calls 346

Distributed File Systems 355

Distributed Message Queues 363

## Part I

# Introduction

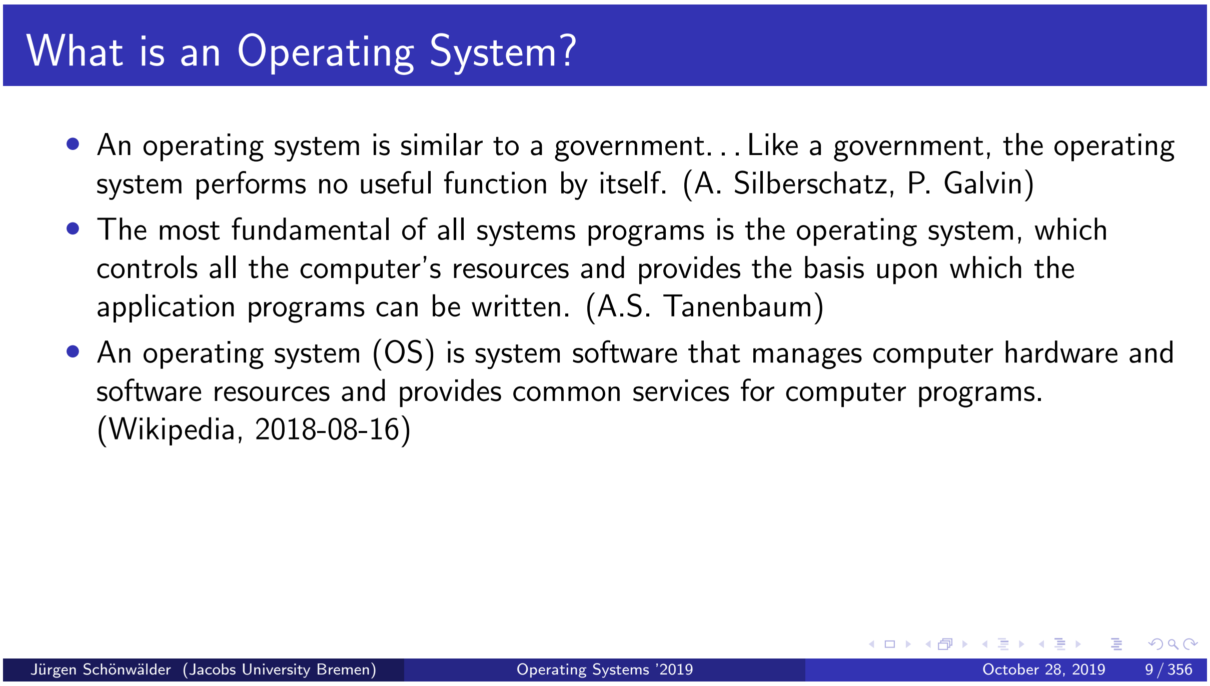
We start by defining what we understand as an operating system and we discusses general operating system requirements and services. Afterwards, we briefly discuss different types of operating systems and we look at software architectures that were used to construct operating system.

# Section 1: Definition and Requirements/Services

1 Definition and Requirements/Services

2 Types of Operating Systems

3 Operating System Architectures



## 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)

Jürgen Schönwälder (Jacobs University Bremen) Operating Systems '2019 October 28, 2019 9 / 356

For computer scientists, the operating system is the system software, which provides an abstraction on which application software can be written, hiding the details of a collection of hardware components from the application programmer and making application programs portable.

For ordinary people, the operating system is often associated with the (graphical) user interface running on top of what computer scientists understand as the operations system. This is understandable since the operating system underlying the graphical user interface is largely invisible to ordinary people.

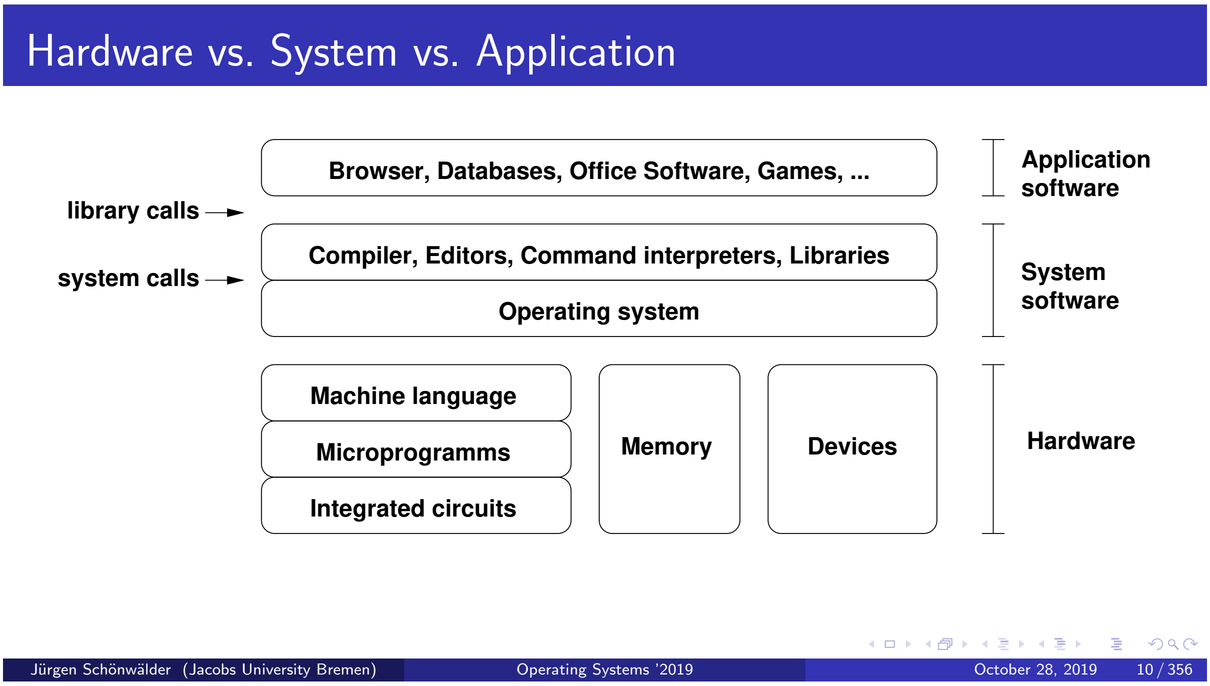
In this course, we do not discuss user interface or usability aspects. The goal of this course is to explain how an operating systems provides the services necessary to execute programs and how essentially abstractions provided to programmers of applications are realized.

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

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

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

Implementing and maintaining on operating system is a huge effort and this has lead to some consolidation of the operating systems that are actually uses. For hardware manufacturers it is often cheaper to contribute to an open source operating system instead of developing and maintaining their own operating system.



## Hardware vs. System vs. Application

library calls →

system calls →

Browser, Databases, Office Software, Games, ...

Application software

Compiler, Editors, Command interpreters, Libraries

Operating system

System software

Machine language

Microprogramms

Integrated circuits

Memory

Devices

Hardware

Jürgen Schönwälder (Jacobs University Bremen) Operating Systems '2019 October 28, 2019 10 / 356

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

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

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

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

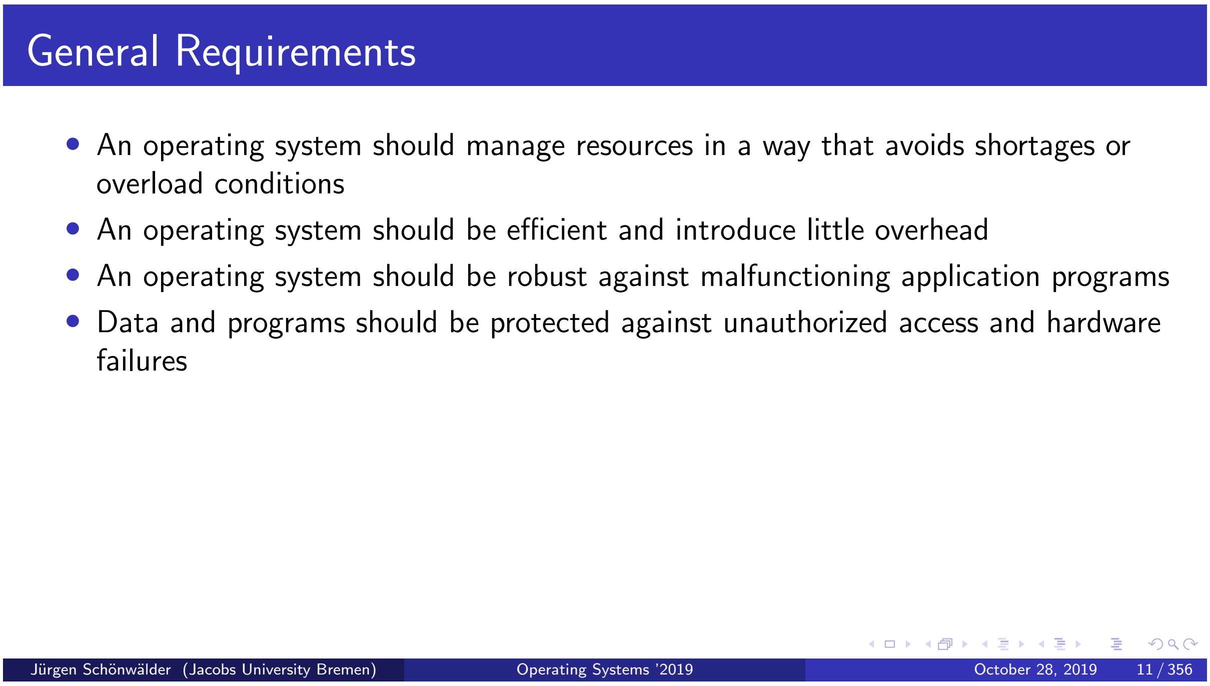
Some of these libraries make it possible to write applications that can be compiled to run on very different operating systems, e.g., Linux, Windows and MacOS.

[1] https://wiki.gnome.org/Projects/GLib

[2] https://apr.apache.org/

[3] https://developer.mozilla.org/en-US/docs/Mozilla/Projects/NSPR

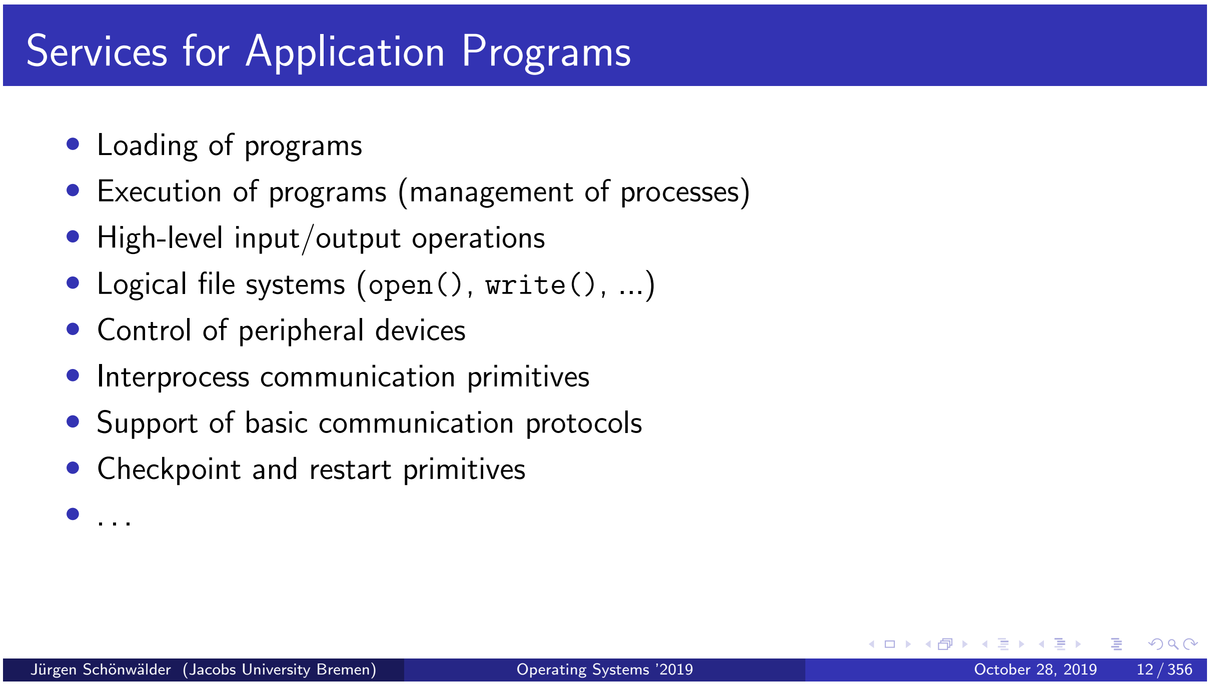
[4] http://doc.qt.io/



## General Requirements

- An operating system should manage resources in a way that avoids shortages or overload conditions
- An operating system should be efficient and introduce little overhead
- An operating system should be robust against malfunctioning application programs
- Data and programs should be protected against unauthorized access and hardware failures

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



## Services for Application Programs

- Loading of programs
- Execution of programs (management of processes)
- High-level input/output operations
- Logical file systems (`open()`, `write()`, ...)
- Control of peripheral devices
- Interprocess communication primitives
- Support of basic communication protocols
- Checkpoint and restart primitives
- ...

Jürgen Schönwälder (Jacobs University Bremen) Operating Systems '2019 October 28, 2019 12 / 356

What are the system services that are needed to execute a hello world program? Discuss different ways to implement a hello world program in C and their advantages and disadvantages.

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

#include <stdio.h>

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

Listing 1: Naive hello world program using C library functions

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

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

int
main(int argc, char *argv[])
{
    const char msg[] = "Hello World";
    int n;

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

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

    return EXIT_SUCCESS;
}
```

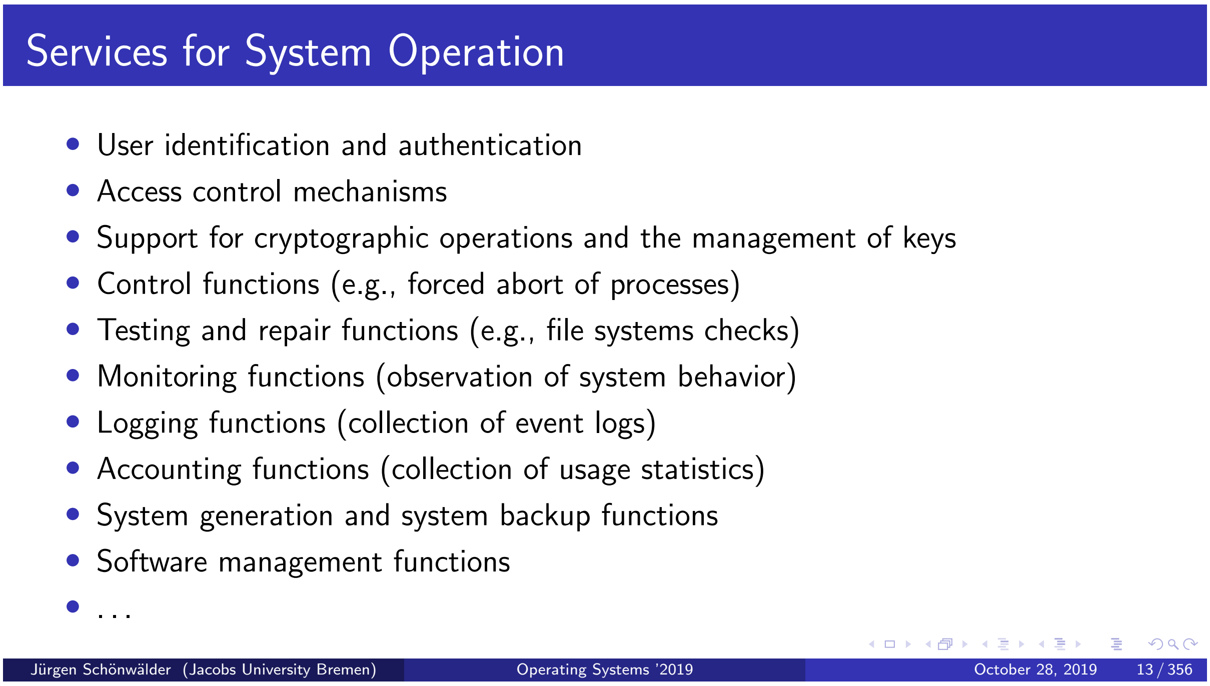
Listing 2: Proper hello world program using C library functions

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

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

int
main(int argc, char *argv[])
{
    const char msg[] = "Hello World\n";
    ssize_t n;

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

    return EXIT_SUCCESS;
}
```

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

```c
/*
 * hello-syscall.c --
 *
 *        This file contains a program which directly invokes a Linux
 *        write() system call by using the syscall library function.
 */

#define _GNU_SOURCE

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

int
main(int argc, char *argv[])
{
    const char msg[] = "Hello World\n";
    ssize_t n;

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

    return EXIT_SUCCESS;
}
```

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



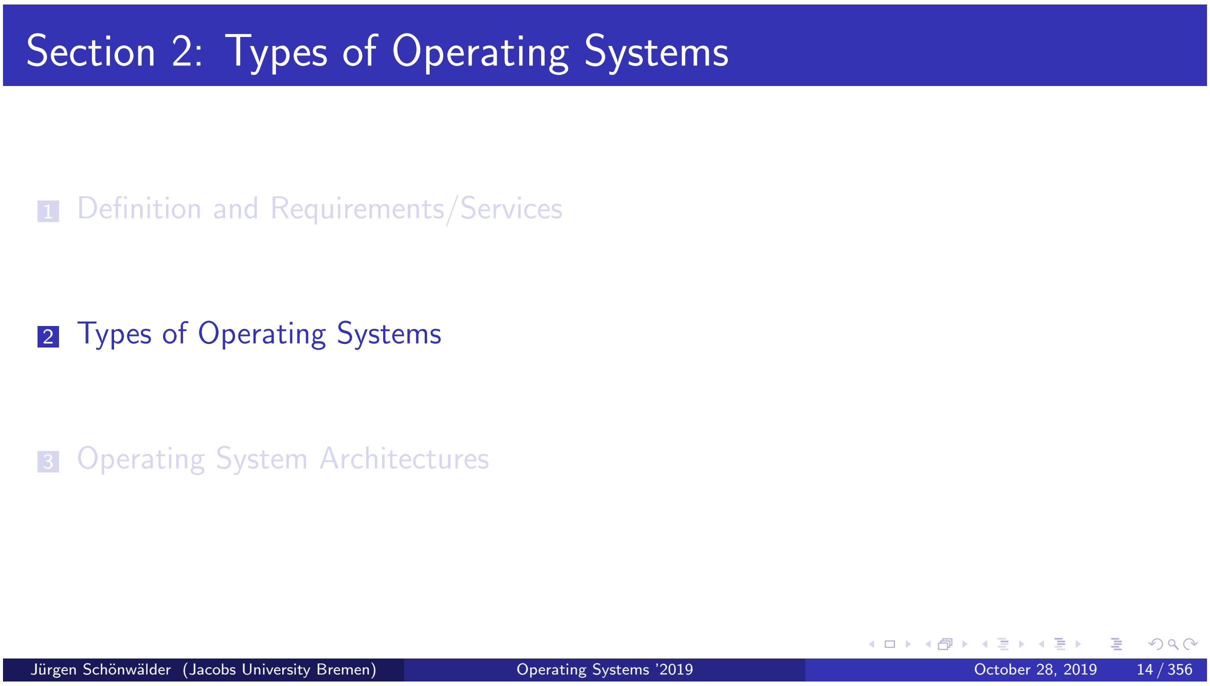
## Services for System Operation

- 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
- ...

Jürgen Schönwälder (Jacobs University Bremen) Operating Systems '2019 October 28, 2019 13 / 356

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

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



## Section 2: Types of Operating Systems

1 Definition and Requirements/Services

2 Types of Operating Systems

3 Operating System Architectures

Jürgen Schönwälder (Jacobs University Bremen) Operating Systems '2019 October 28, 2019 14 / 356

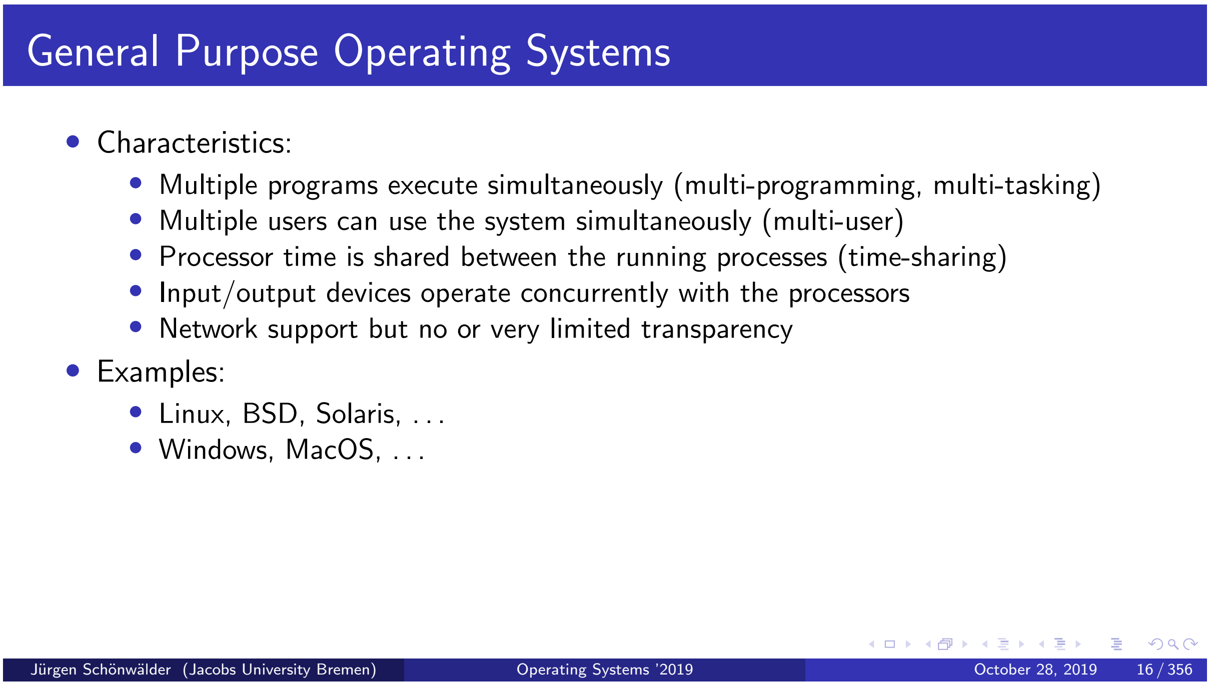
Operating systems can be classified by the types of computing environments they are designed to support:

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

Subsequent slides provide details about these different operating system types.

## Batch Processing Operating Systems

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



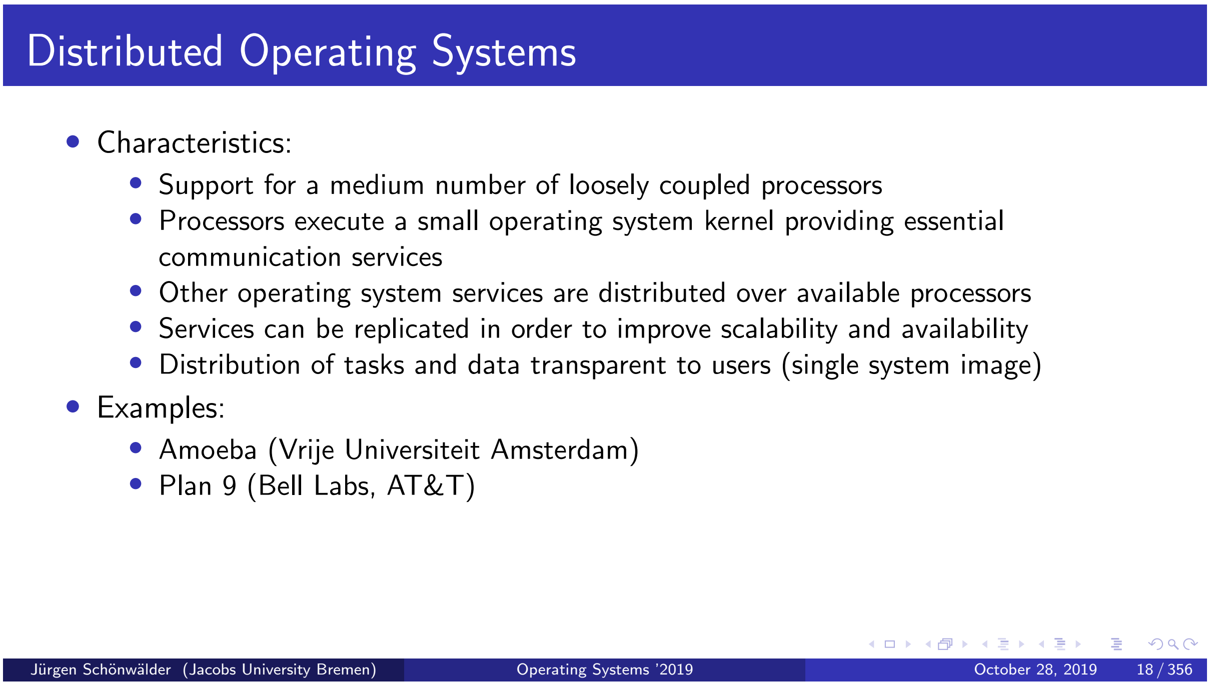
## General Purpose Operating Systems

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

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

# Parallel Operating Systems

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

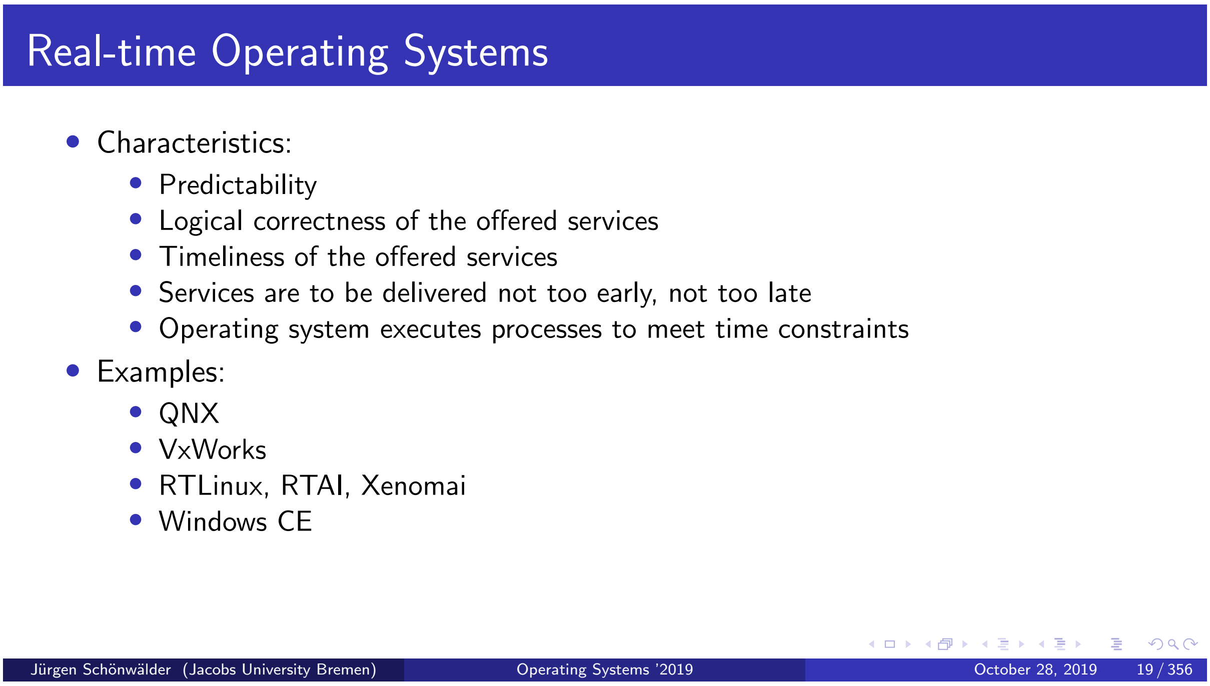


## Distributed Operating Systems

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

Jürgen Schönwälder (Jacobs University Bremen) Operating Systems '2019 October 28, 2019 18 / 356

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

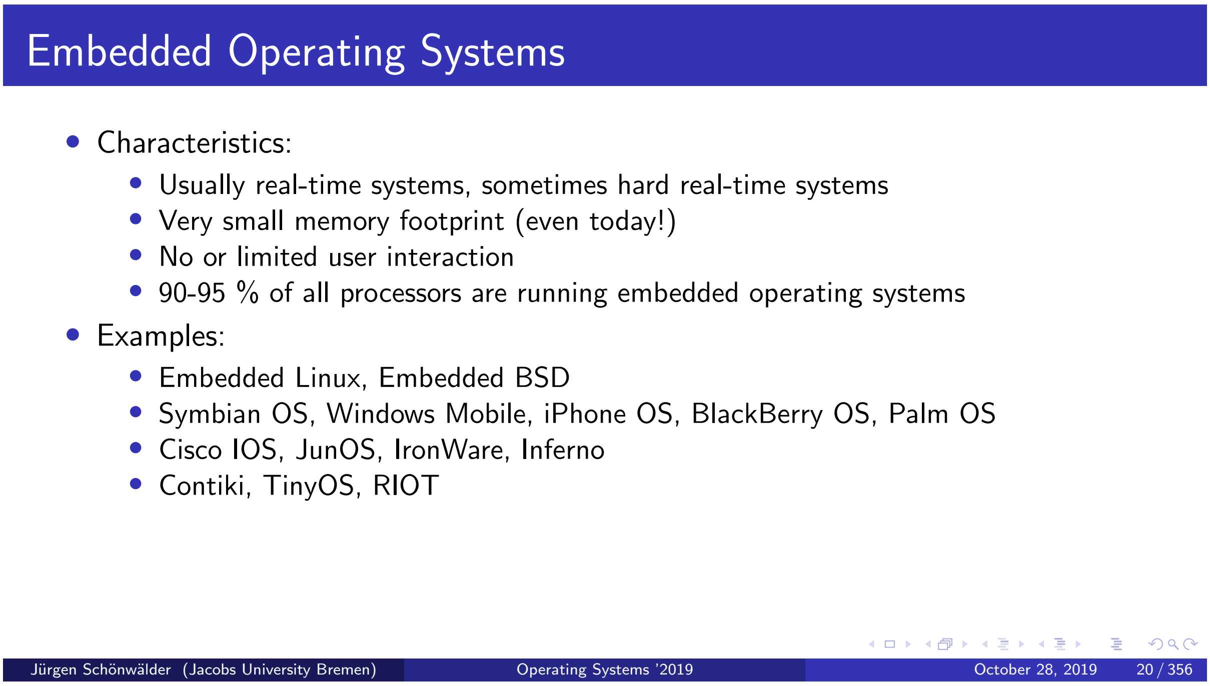


## Real-time Operating Systems

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

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

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



## Embedded Operating Systems

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

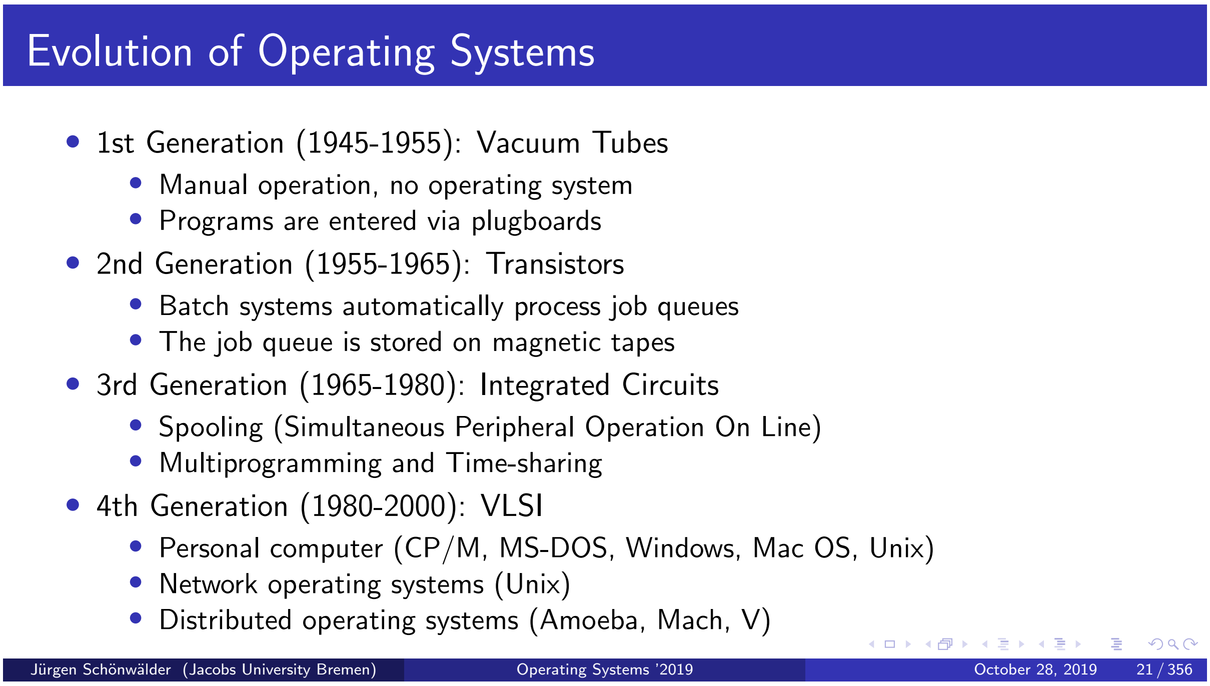
Special variants of Linux and BSD systems have been developed to support embedded systems and they are gaining momentum. On mobile phones, the computing resources are meanwhile big enough that mobile phone operating systems tend to become variants of general purpose operating systems. There are, however,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

Some notable Linux variants:

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

[5] https://openwrt.org/

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



## Evolution of Operating Systems

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

Jürgen Schönwälder (Jacobs University Bremen) Operating Systems '2019 October 28, 2019 21 / 356

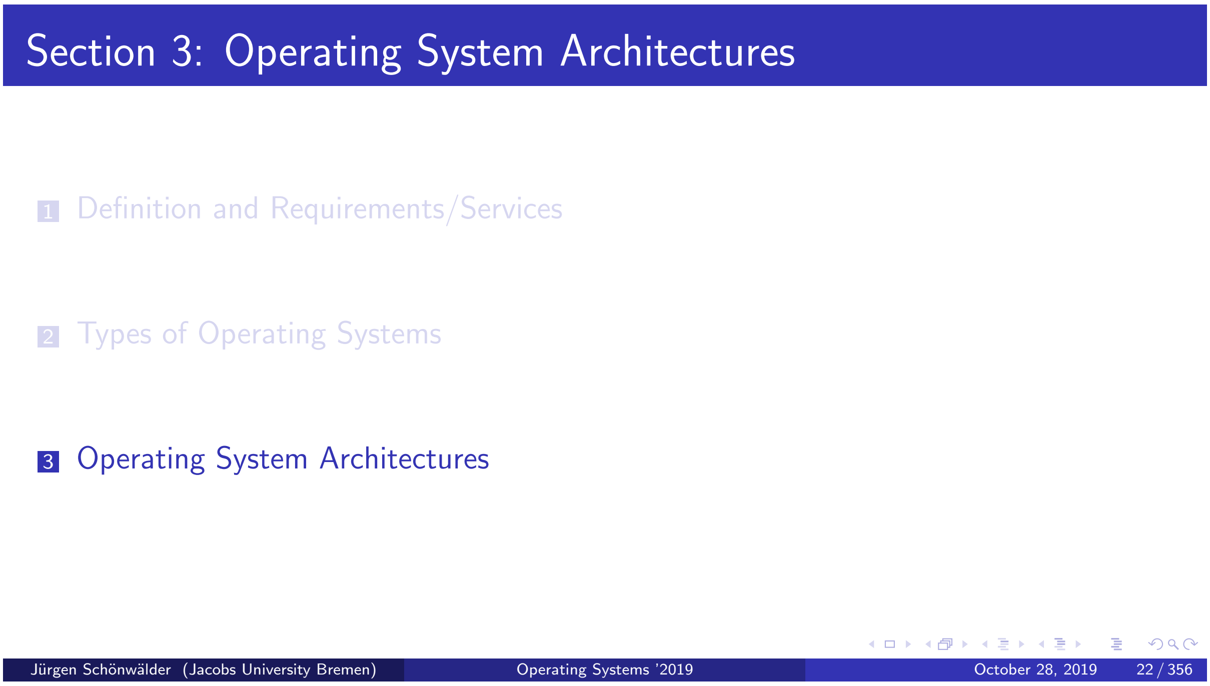
The development since 2000 is largely driven by virtualization techniques such as virtual machines or containers and software systems that manage very large collections of virtual machines and containers. Some notable open source systems:

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

[7] https://www.openstack.org/

[8] https://opennebula.org/

[9] https://kubernetes.io/



## Section 3: Operating System Architectures

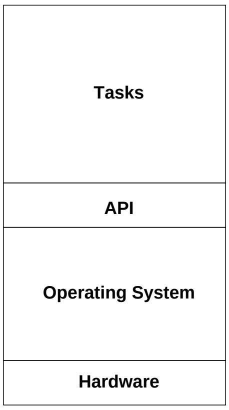
1. Definition and Requirements/Services
2. Types of Operating Systems
3. Operating System Architectures

Operating systems can be classified by their software architecture:

- Monolithic operating systems
- Monolithic modular operating systems
- Monolithic layered operating systems
- Virtual machines
- Client/Server operating systems
- Distributed operating systems

# Monolithic Operating Systems

- A collection of functions without a structure (the big mess)
- Typically not-portable, difficult to maintain, lack of reliability
- All services are in the kernel with the same privilege level
- Monolithic systems can be highly efficient



Tasks

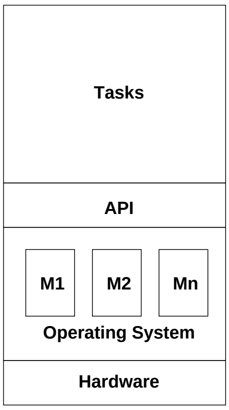
API

Operating System

Hardware

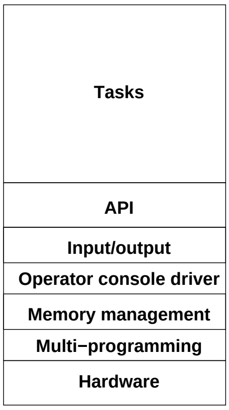
# Monolithic and Modular Operating Systems

- Modules can be platform independent
- Easier to maintain and to develop
- Increased reliability / robustness
- All services are in the kernel with the same privilege level
- May reach high efficiency
- Example: Linux



# Monolithic and Layered Operating Systems

- Easily portable, significantly easier to maintain
- Often reduced efficiency because of the need to go through many layered interfaces
- Rigorous implementation of the stacked virtual machine perspective
- Services offered by the various layers are important for the overall performance
- Example: THE (Dijkstra, 1968)



Tasks

API

Input/output

Operator console driver

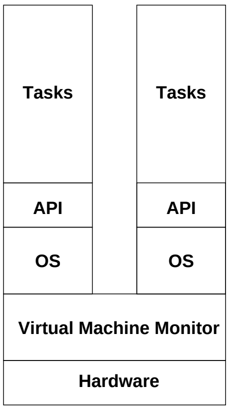
Memory management

Multi−programming

Hardware

## Virtual Machines

- Virtualization of the hardware
- Multiple operating systems can execute concurrently
- Separation of multi-programming from other operating system services
- Examples: IBM VM/370 ('79), VMware (1990s), XEN (2003)



## Part II

# Hardware

In this part we review some basic concepts of computer architecture that are relevant for understanding operating systems.

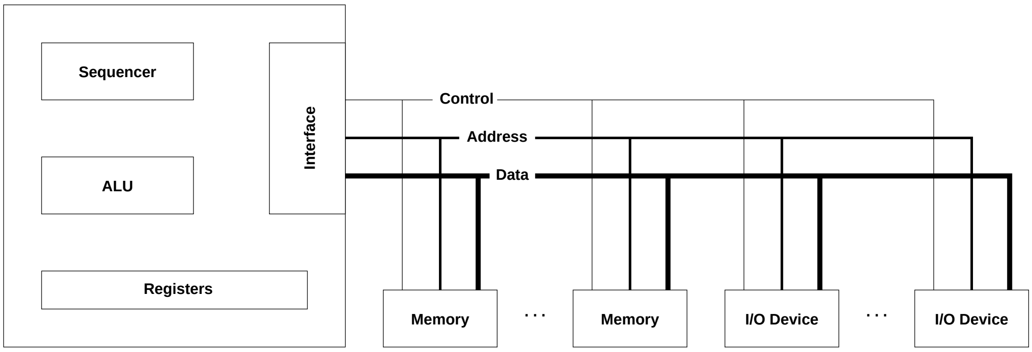
# Section 4: Common Computer Architecture

4 Common Computer Architecture

5 I/O Systems and Interrupts

6 Memory

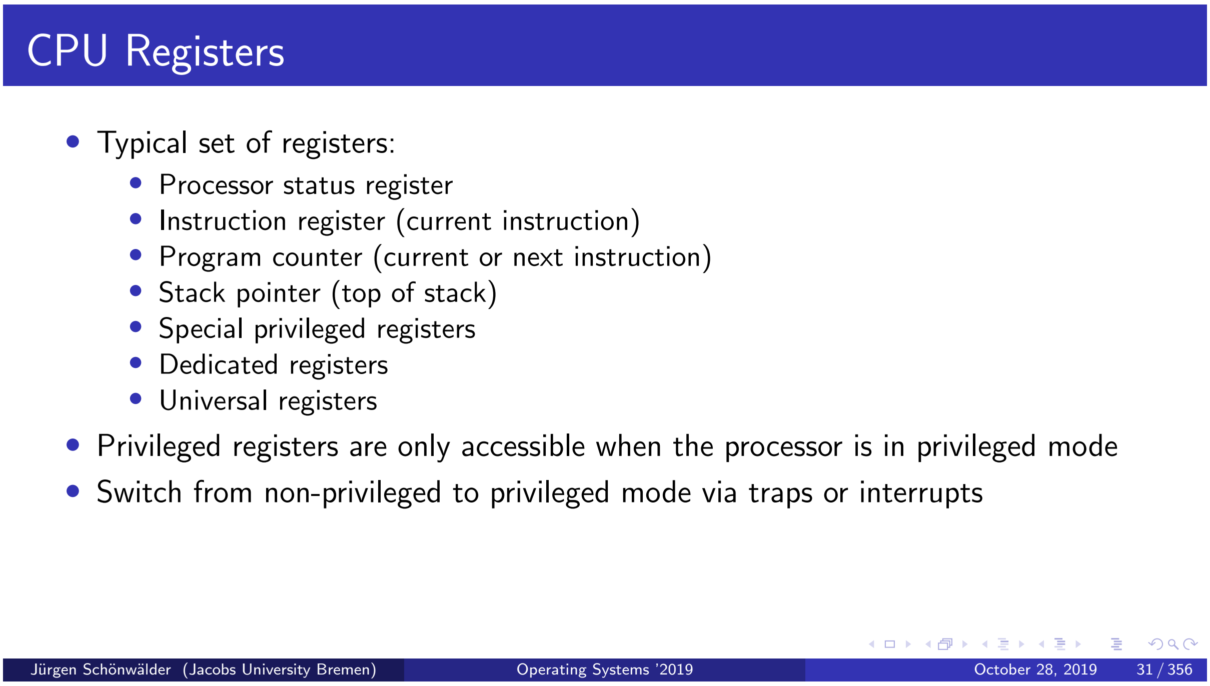
## Common Computer Architecture



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

The usage of shared busses to connect components of a computer requires arbitration, synchronization, interrupts, priorities.

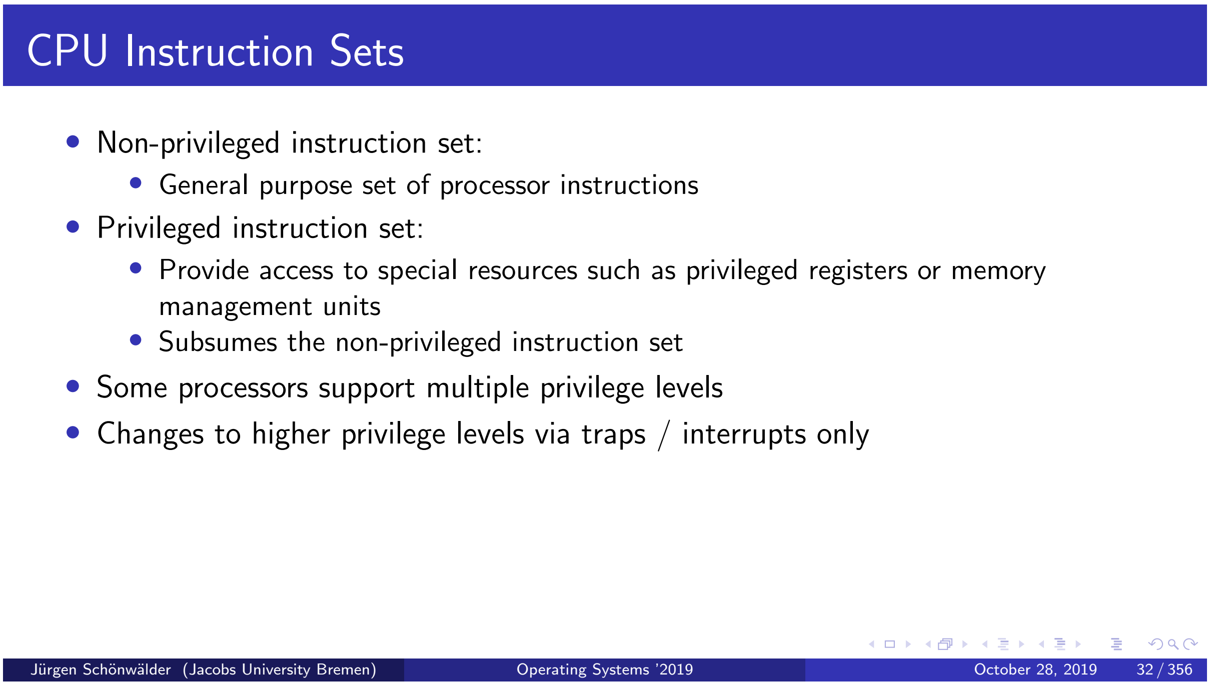
A CPU consists of a command sequencer fetching instructions, an arithmetic logic unit (ALU), and a set of registers. Data is carried over the data bus to/from the address carried over the address bus. The control bus signals the direction of the data transfer and when the transfer takes place.



## CPU Registers

- Typical set of 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
- Privileged registers are only accessible when the processor is in privileged mode
- Switch from non-privileged to privileged mode via traps or interrupts

CPUs used by general purpose computers usually support multiple privilege levels. The Intel x86 architecture, for example, supports four privilege levels (protection rings 0. . . 3). Note that CPUs for small embedded systems often do not support multiple privilege levels and this has serious implications on the robustness an operating system can achieve. In the following, we focus on operating systems that run on hardware that supports multiple CPU privilege levels.



## CPU Instruction Sets

- Non-privileged instruction set:
  - General purpose set of processor instructions
- Privileged instruction set:
  - Provide access to special resources such as privileged registers or memory management units
  - Subsumes the non-privileged instruction set
- Some processors support multiple privilege levels
- Changes to higher privilege levels via traps / interrupts only

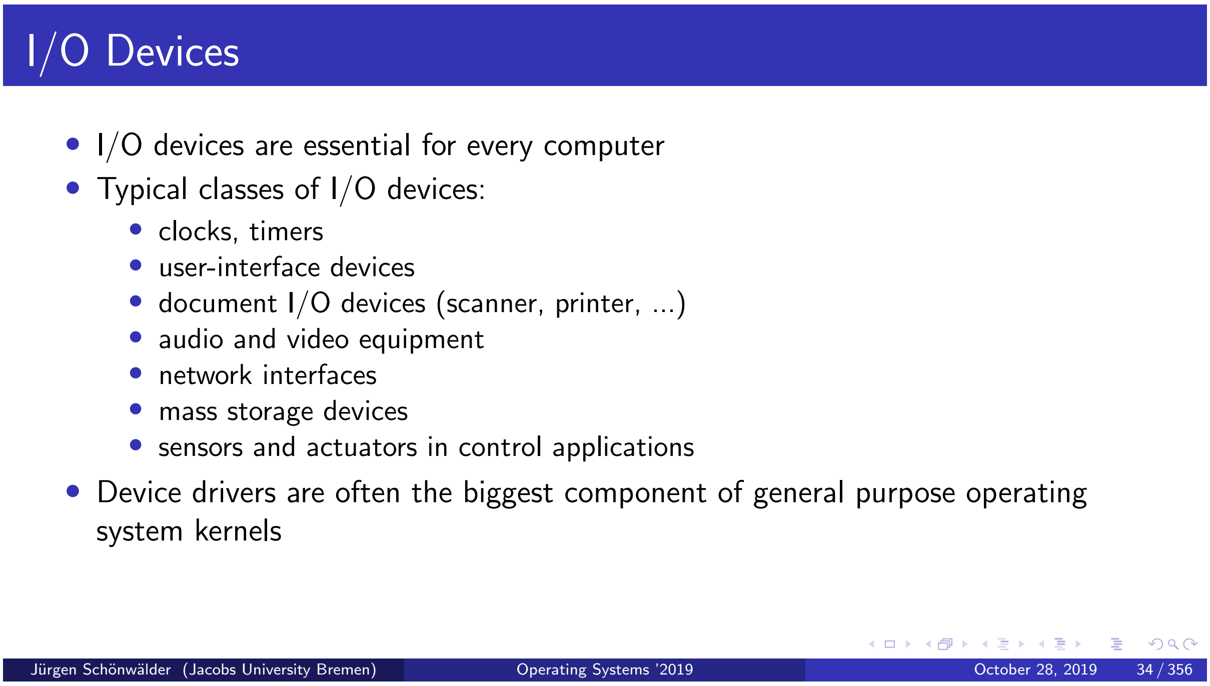
In the following, we assume that an unprivileged instruction set is available to run application programs and a privileged instruction is available to execute core operating system functionality.

# Section 5: I/O Systems and Interrupts

4 Common Computer Architecture

5 I/O Systems and Interrupts

6 Memory



## I/O Devices

- I/O devices are essential for every computer
- Typical classes of I/O devices:
    - clocks, timers
    - user-interface devices
    - document I/O devices (scanner, printer, ...)
    - audio and video equipment
    - network interfaces
    - mass storage devices
    - sensors and actuators in control applications
- Device drivers are often the biggest component of general purpose operating system kernels

## Basic I/O Programming

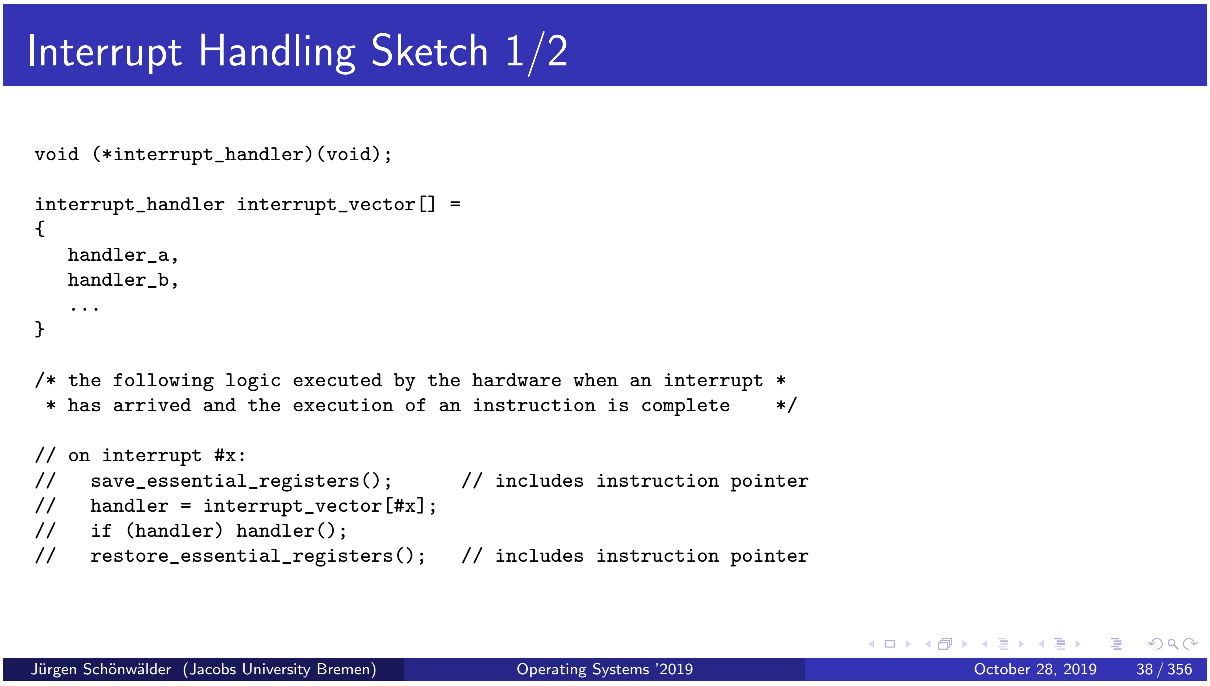
- *Status driven*: the processor polls an I/O device for information
  - Simple but inefficient use of processor cycles
- *Interrupt driven*: the I/O device issues an interrupt when data is available or an I/O operation has been completed
  - *Program controlled*: Interrupts are handled by the processor 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



# Interrupt Handling Sketch 1/2

```
void (*interrupt_handler)(void);

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

/* the following logic executed by the hardware when an interrupt *
 * has arrived and the execution of an instruction is complete    */

// on interrupt #x:
//   save_essential_registers();      // includes instruction pointer
//   handler = interrupt_vector[#x];
//   if (handler) handler();
//   restore_essential_registers();   // includes instruction pointer
```

# Interrupt Handling Sketch 2/2

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

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

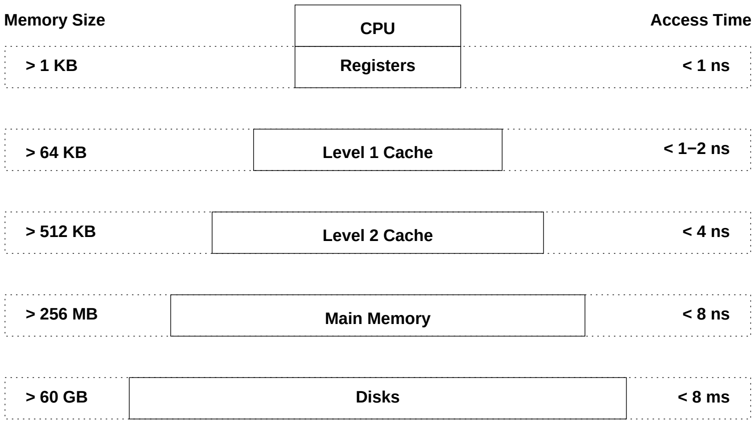
## Section 6: Memory

4 Common Computer Architecture

5 I/O Systems and Interrupts

6 Memory

# Memory Sizes and Access Times



| Memory Size | CPU | Access Time |
|---|---|---|
| > 1 KB | Registers | < 1 ns |
| > 64 KB | Level 1 Cache | < 1–2 ns |
| > 512 KB | Level 2 Cache | < 4 ns |
| > 256 MB | Main Memory | < 8 ns |
| > 60 GB | Disks | < 8 ms |

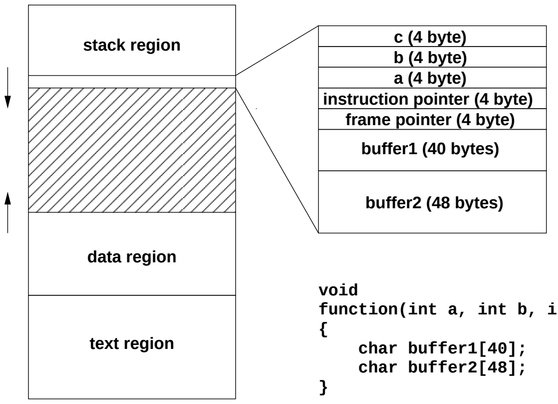
- There is a trade-off between memory speed and memory size.

## Memory Segments

| Segment | Description |
|---|---|
| text | machine instructions of the program |
| data | static 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) |

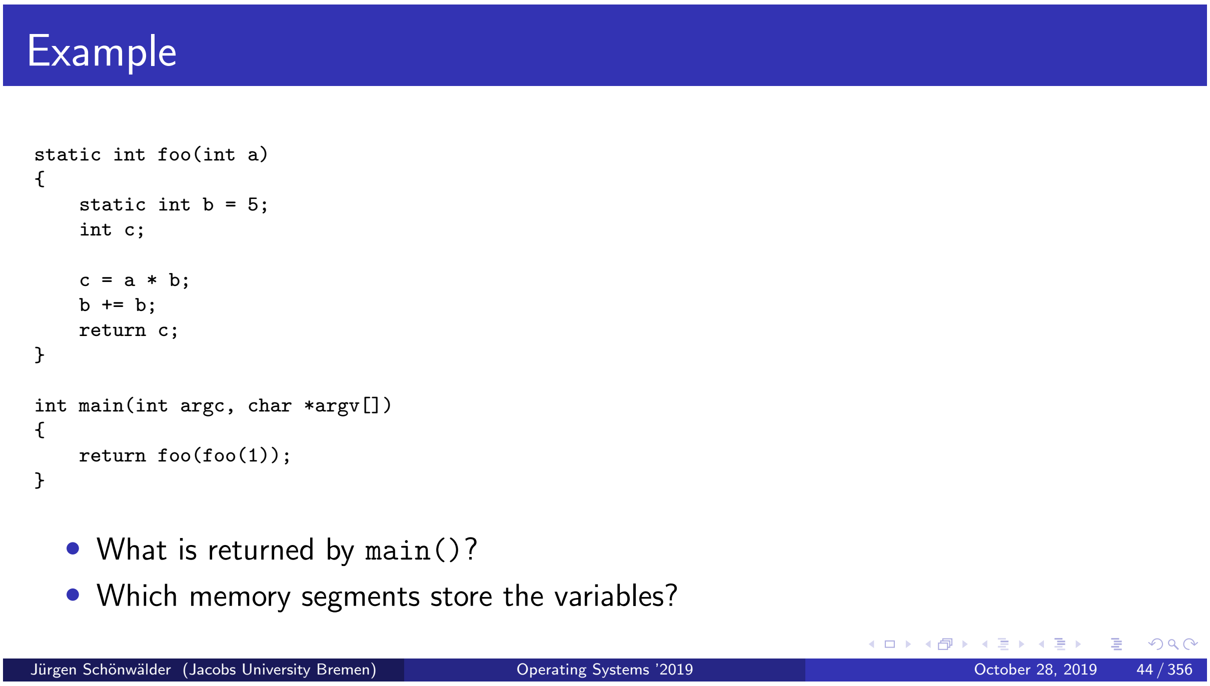
- Memory used by a program is usually partitioned into different segments that serve different purposes and may have different access rights

# Stack Frames



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

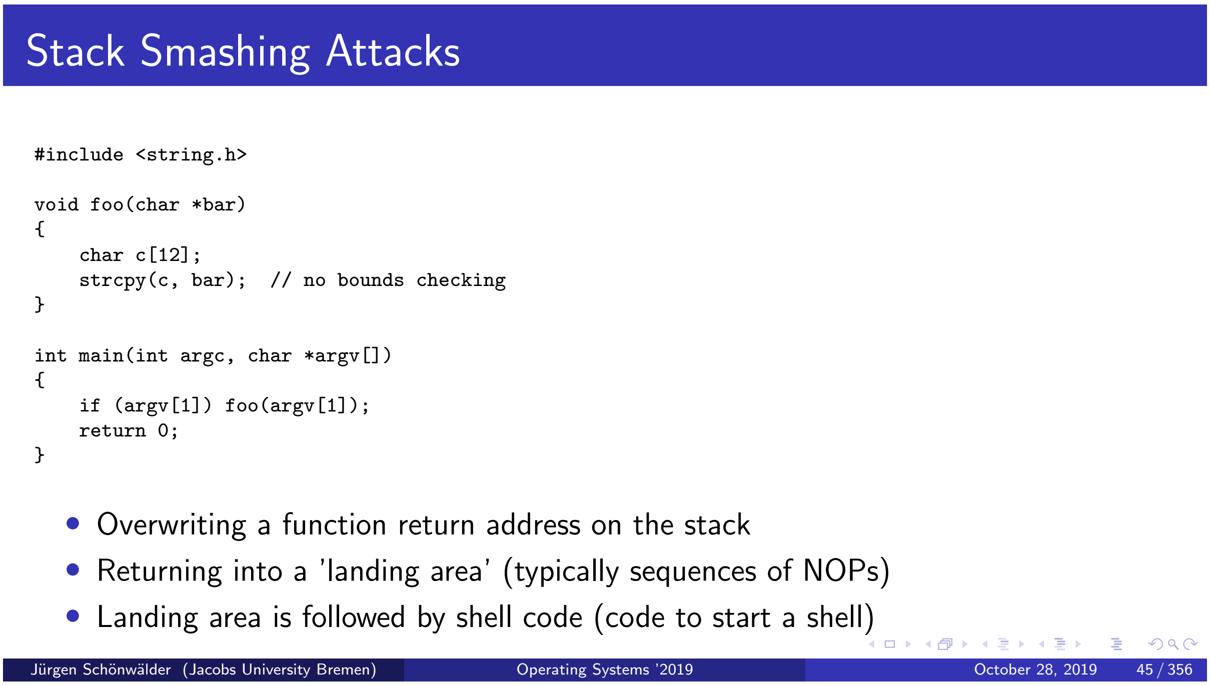
- Every function call leaves an entry (stack frame) on the stack
- Stack frame layout is processor specific (here Intel x86)



## 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?



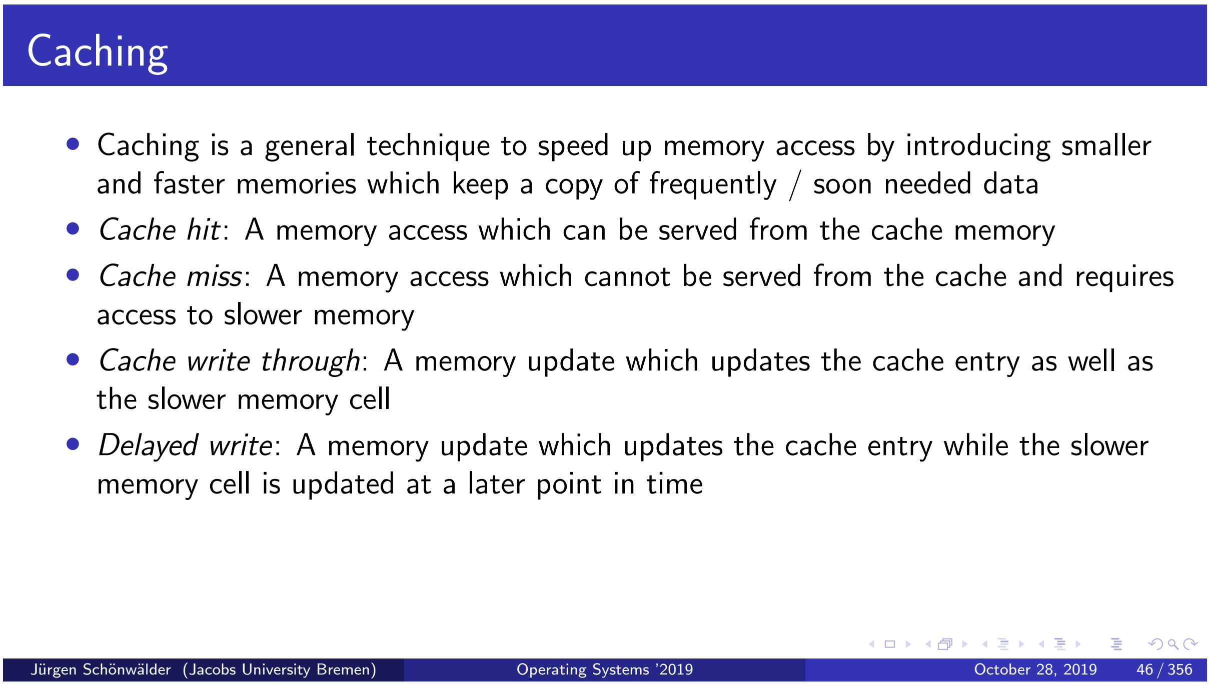
## Stack Smashing Attacks

```
#include <string.h>

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

int main(int argc, char *argv[])
{
    if (argv[1]) foo(argv[1]);
    return 0;
}
```

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

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

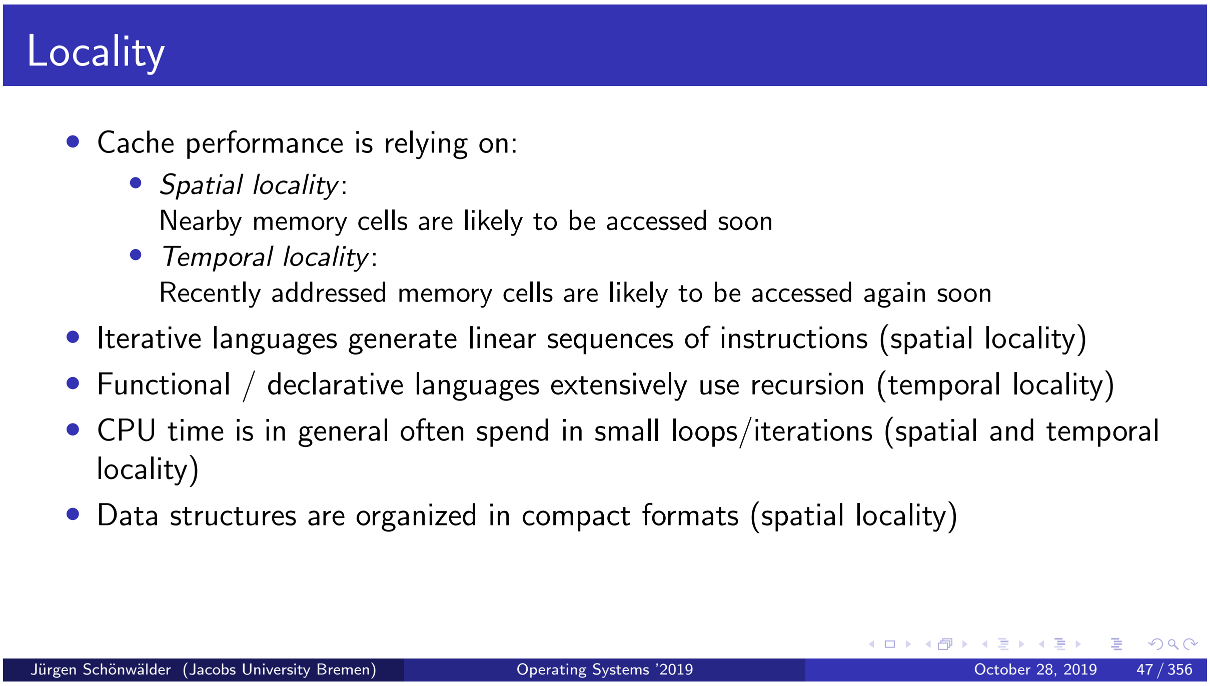
For C and C++ programmers, there is no alternative to developing the discipline to always ensure that uncontrolled access to memory is prevented, i.e., making it a habit to always write robust code.



## Caching

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

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



## Locality

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

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

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

## Part III

# Processes and Threads

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

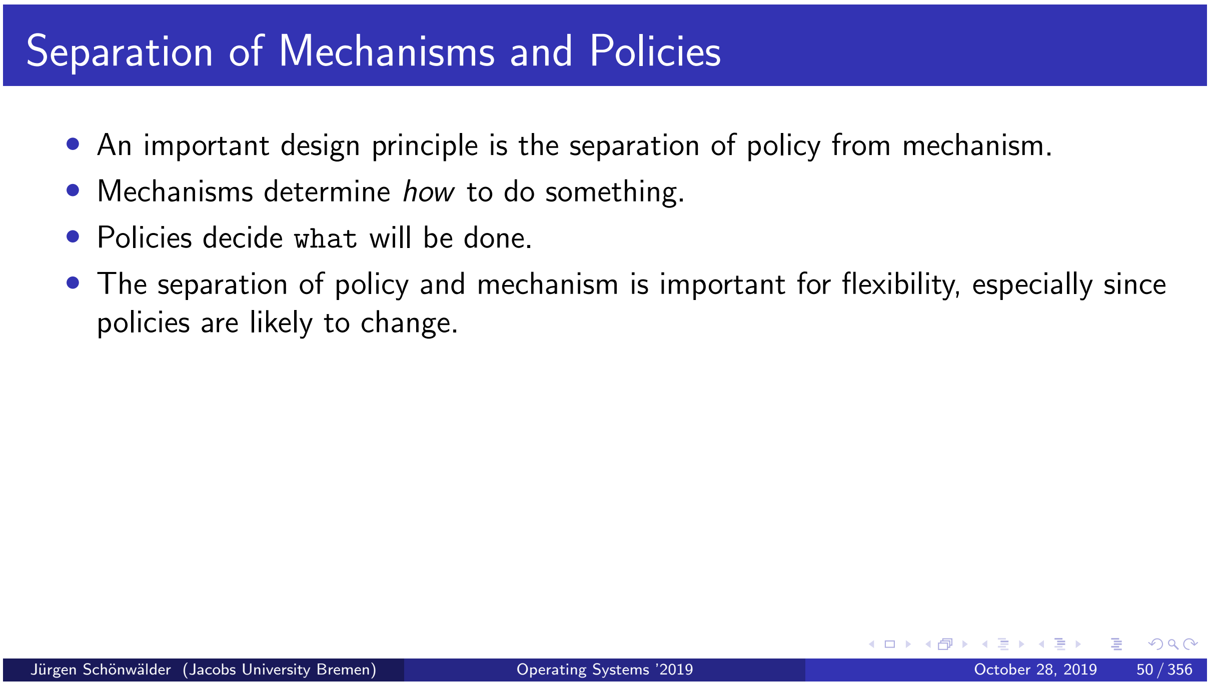
Processes are relatively heavy-weight objects since every process has its own memory, his own collection of open files, etc. In order to exploit hardware with multiple CPU cores, it is desirable to exploit multiple cores within a single process, i.e., within the same memory image. The lead to the introduction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

# Section 7: Fundamental Concepts

7 Fundamental Concepts

8 Processes

9 Threads



## 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.

Jürgen Schönwälder (Jacobs University Bremen) Operating Systems '2019 October 28, 2019 50 / 356

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

Examples:

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

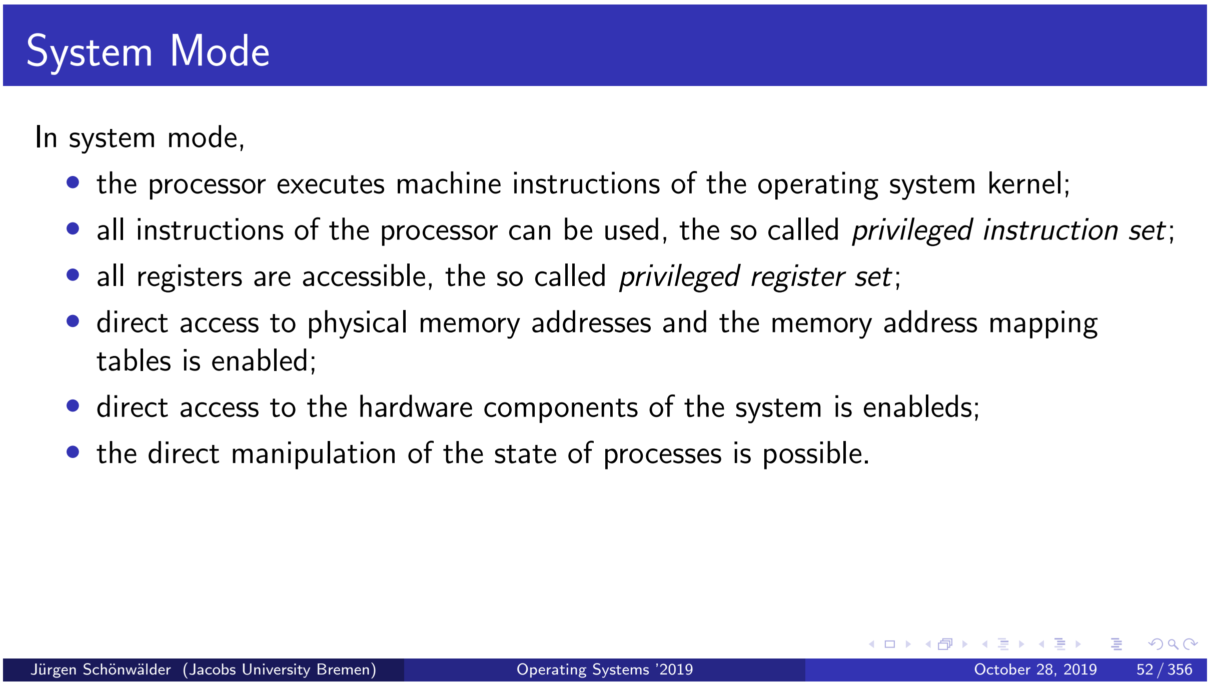
Good separation of mechanisms and policies leads to systems that can be adapted to different usage scenarios in flexible ways.

## User Mode

In user mode,

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

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



## System Mode

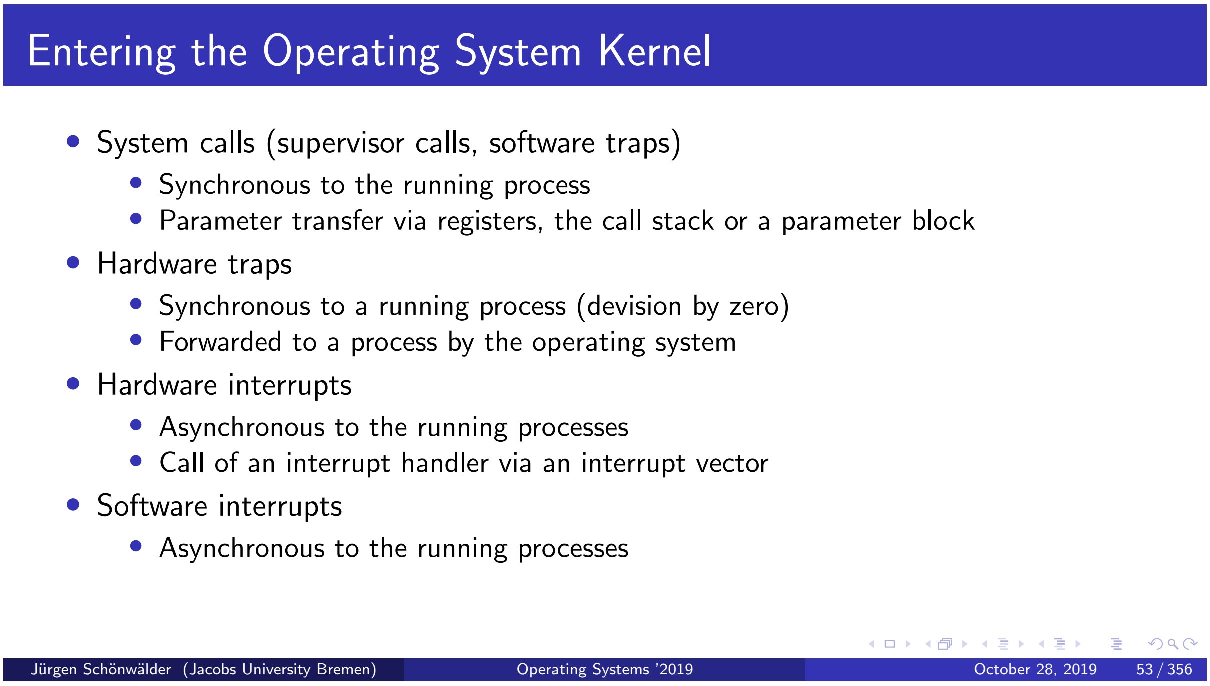
In system mode,

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

Jürgen Schönwälder (Jacobs University Bremen) Operating Systems '2019 October 28, 2019 52 / 356

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

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



## Entering the Operating System Kernel

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

Jürgen Schönwälder (Jacobs University Bremen) Operating Systems '2019 October 28, 2019 53 / 356

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

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

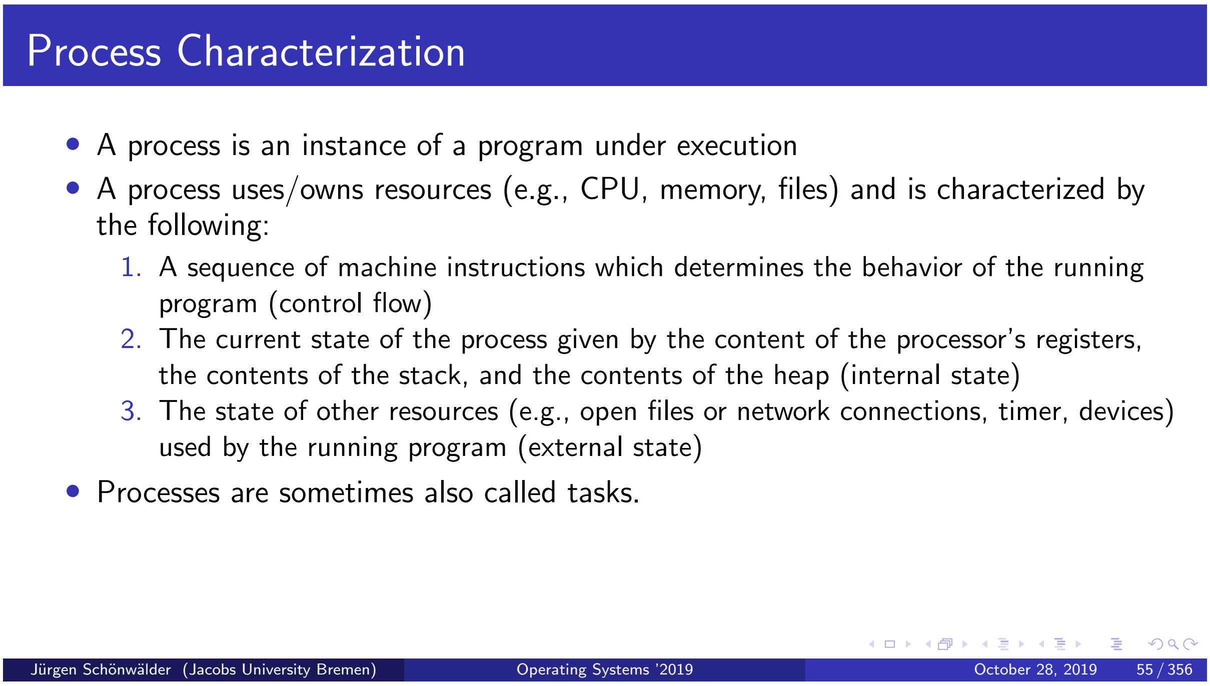
Note that system calls are much more expensive than library calls since system calls require a transition from user mode to system mode and finally back to user mode. Efficient programs therefore tend to minimize the system calls they need to perform.

# Section 8: Processes

7 Fundamental Concepts

8 Processes

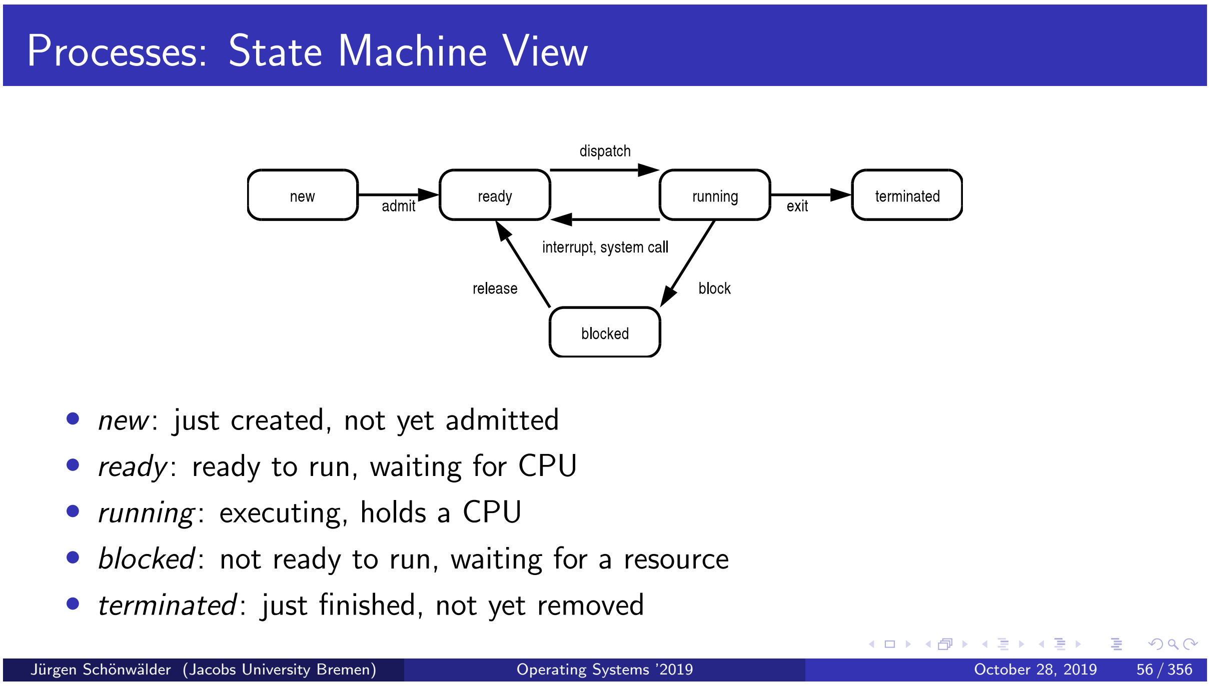
9 Threads



## Process Characterization

- 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 which determines the behavior of the running program (control flow)
  2. The current state of the process given by the content of the processor's registers, the contents of the stack, and the contents of the heap (internal state)
  3. The state of other resources (e.g., open files or network connections, timer, devices) used by the running program (external state)
- Processes are sometimes also called tasks.

On a Unix system, the shell command `ps` provides a list of all processes on the system. There are many options that can be used to select the information displayed for the processes on the system.



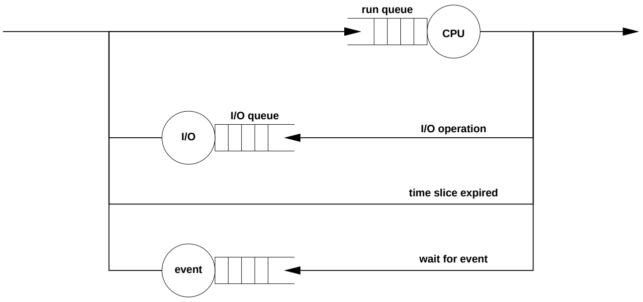
## Processes: State Machine View

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

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

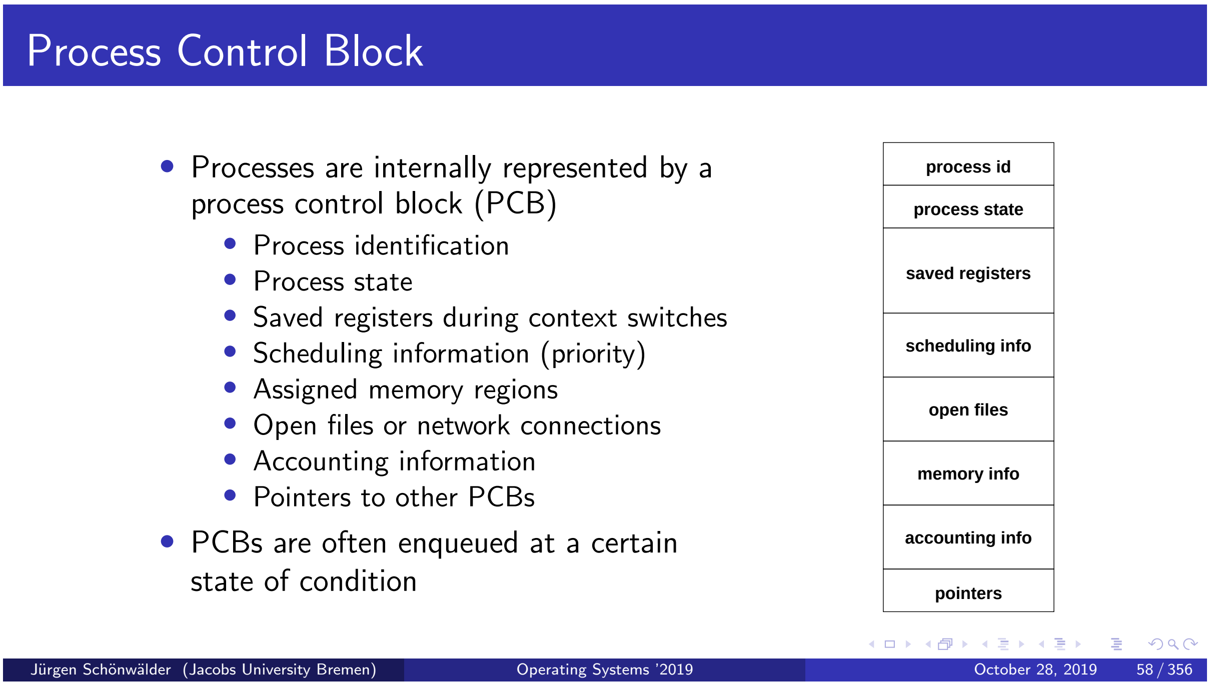
```
top - 20:21:12 up 3 days,  7:16,  1 user,  load average: 0.00, 0.00, 0.00
Tasks:  85 total,   1 running,  84 sleeping,   0 stopped,   0 zombie
%Cpu(s):  0.3 us,  0.3 sy,  0.0 ni, 84.7 id, 14.6 wa,  0.0 hi,  0.0 si,  0.0 st
MiB Mem :    987.5 total,    155.3 free,    132.8 used,    699.4 buff/cache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

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

# Processes: Queueing Model View



- Processes are enqueued if resources are not readily available or if processes wait for events
- Dequeuing strategies have strong performance impact
- Queueing models can be used for performance analysis

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



## Process Control Block

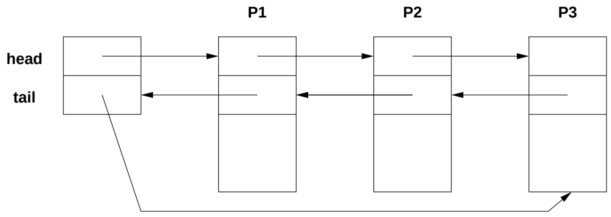
- Processes are internally represented by a process control block (PCB)
  - Process identification
  - Process state
  - Saved registers during context switches
  - Scheduling information (priority)
  - Assigned memory regions
  - Open files or network connections
  - Accounting information
  - Pointers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

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

Jürgen Schönwälder (Jacobs University Bremen) Operating Systems '2019 October 28, 2019 58 / 356

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

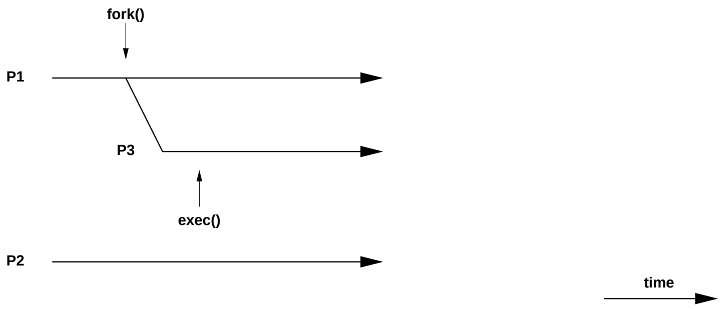
# Process Lists



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

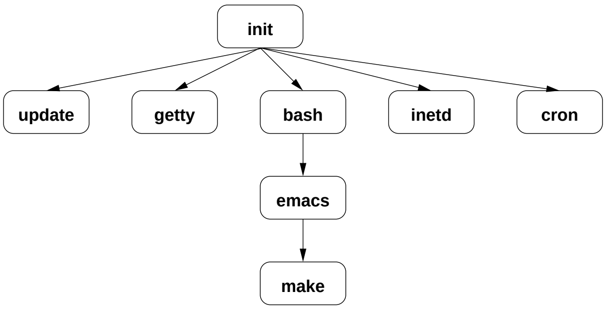
Iterating over the process list is tricky since the process list can change during the iteration unless one takes precautions that prevent changes during the iteration. The same is true for many members of the data structure representing a process. Kernel programming requires to take care of concurrency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

# Process Creation



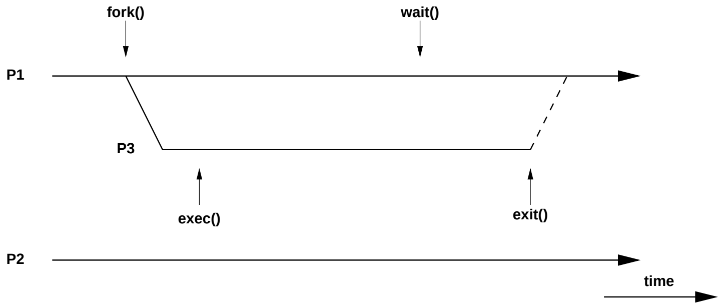
- The `fork()` system call creates a new child process
  - which is an exact copy of the parent process,
  - except that the result of the system call differs
- The `exec()` system call replaces the current process image with a new process image.

# Process Trees

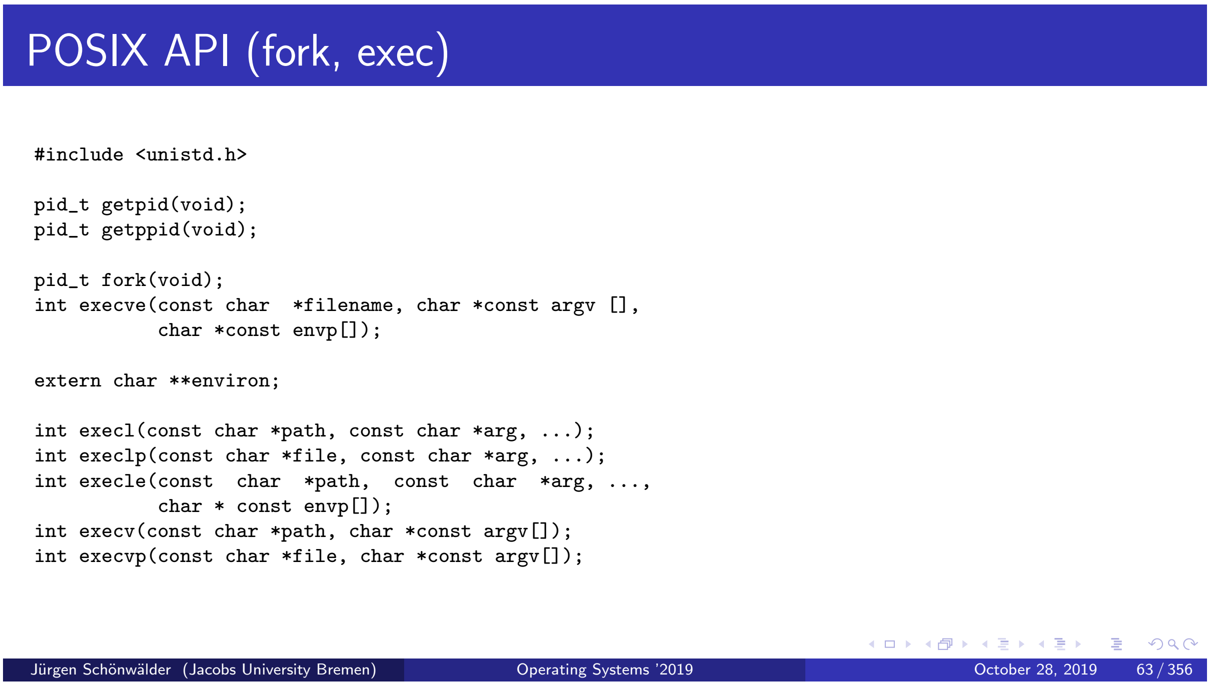


- First process is created when the system is initialized
- All other processes are created using `fork()`, which leads to a process tree
- PCBs often contain pointers to parent PCBs

# Process Termination



- Processes can terminate themself by calling `exit()`
- The `wait()` system call allows processes to wait for the termination of a child process
- Terminating processes return a numeric result code



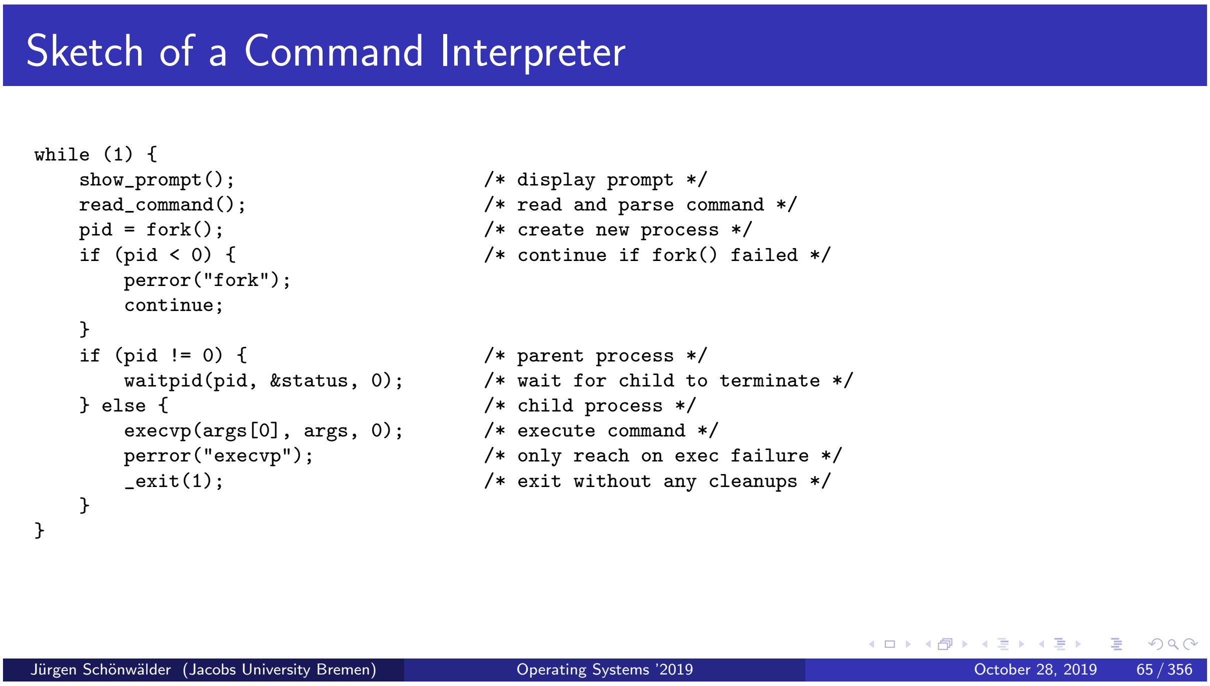
## POSIX API (fork, exec)

```
#include <unistd.h>

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

pid_t fork(void);
int execve(const char  *filename, char *const argv [],
           char *const envp[]);

extern char **environ;

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[]);
```

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

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

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

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

    for (i = 1; i < argc; i++) {
        pids[i] = fork();
        if (pids[i] == -1) {
            fprintf(stderr, "failed to fork: %s\n", strerror(errno));
            continue;
        }
        if (pids[i] == 0) {
            work(argv[i]);
        }
    }
    for (i = 1; i < argc; i++) {
        if (pids[i] > 0) {
            (void) waitpid(pids[i], &status, 0);
        }
    }
    (void) printf("\n");

    return EXIT_SUCCESS;
}
```

# POSIX API (exit, wait)

```
#include <stdlib.h>

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

#include <unistd.h>

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

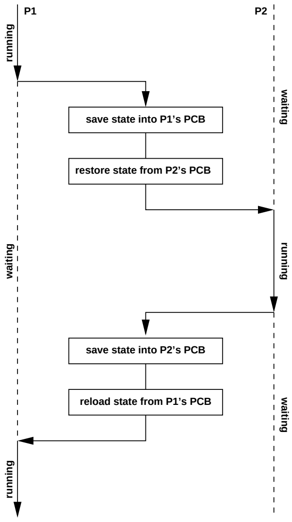
```
while (1) {
    show_prompt();                     /* display prompt */
    read_command();                    /* read and parse command */
    pid = fork();                      /* create new process */
    if (pid < 0) {                     /* continue if fork() failed */
        perror("fork");
        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 */
    }
}
```

Jürgen Schönwälder (Jacobs University Bremen) Operating Systems '2019 October 28, 2019 65 / 356

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

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

static const char *progname = "msh";

static void
show_prompt(void)
{
    printf("%s > ", progname);
    fflush(stdout);
}

static char*
next_token(char **s)
{
    char *token, *p;

    assert(s && *s);

    for (p = *s; *p && isspace(*p); p++) {
        *p = '\0';
    }
    token = p;
    for (; *p && !isspace(*p); p++) ;
    for (; *p && isspace(*p); p++) {
        *p = '\0';
    }
    *s = p;
    return token;
}

static void
read_command(FILE *stream, int *argc, char ***args)
{
#ifndef BUFLEN
#define BUFLEN 512
#endif
    static char line[BUFLEN];
    static char* argv[BUFLEN];
    char *p;

    assert(argc && args);

    memset((char *) argv, 0, sizeof(argv));
    p = fgets(line, sizeof(line), stream);
    for (*argc = 0; p && *p; (*argc)++) {
        argv[*argc] = next_token(&p);
    }
    *args = argv;
}

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

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

    return EXIT_SUCCESS;
}
```

# 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

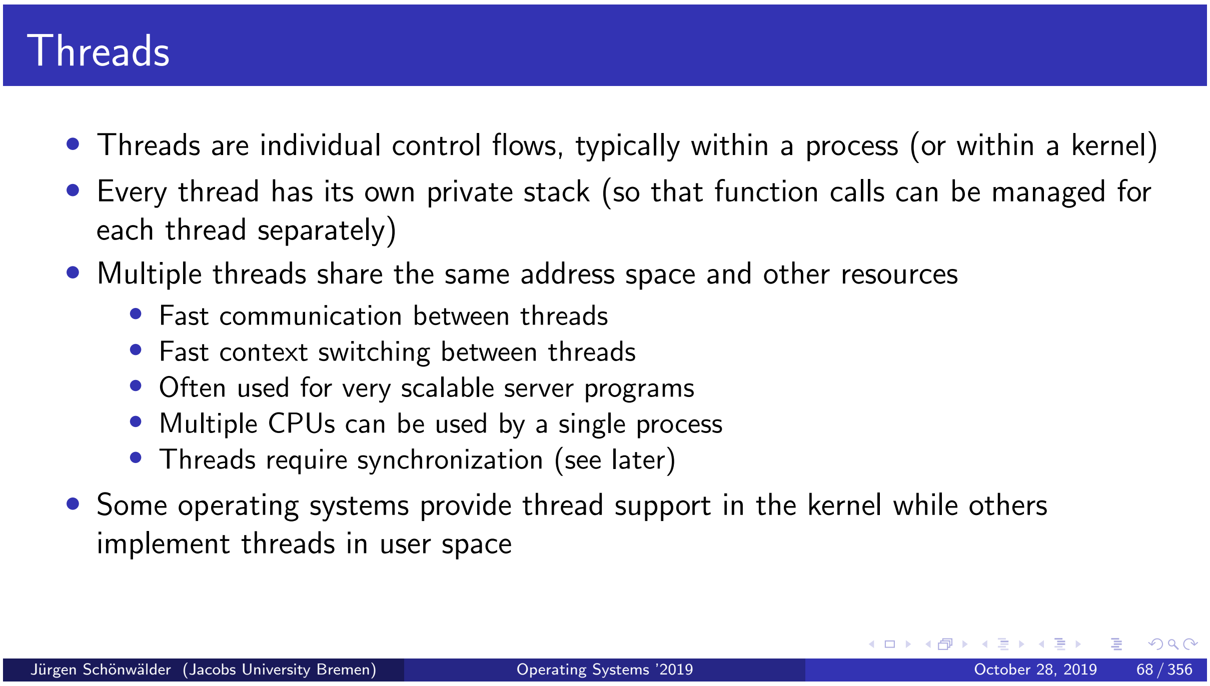


# Section 9: Threads

7 Fundamental Concepts

8 Processes

9 Threads

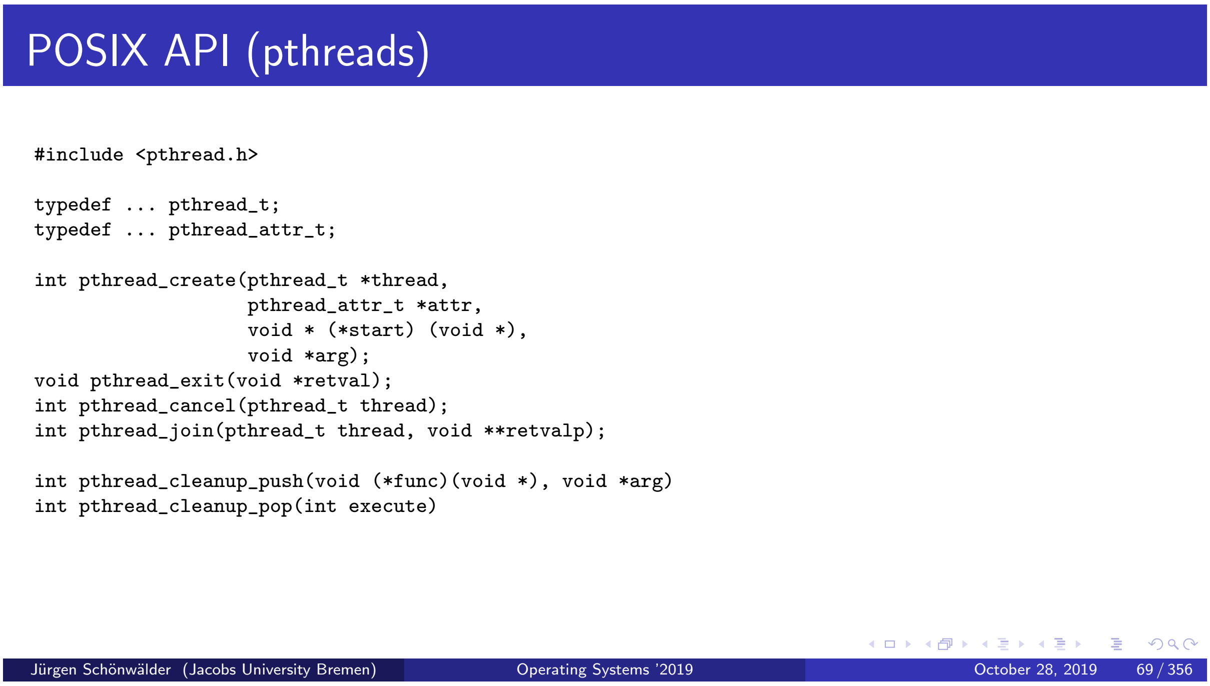


## Threads

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

A thread is the smallest sequence of programmed instructions that can be managed independently (by the operating system kernel).

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



# 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)
```

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

#define _REENTRANT
#define _DEFAULT_SOURCE

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

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

int
main(int argc, char *argv[])
{
    int i, r;
    pthread_t tids[argc];

    for (i = 1; i < argc; i++) {
        r = pthread_create(&tids[i], NULL, work, argv[i]);
        if (r) {
            fprintf(stderr, "failed to create thread: %s\n", strerror(r));
        }
    }

    for (i = 1; i < argc; i++) {
        if (tids[i]) {
            (void) pthread_join(tids[i], NULL);
        }
    }
    (void) printf("\n");

    return EXIT_SUCCESS;
}
```

# Processes and Threads in Linux (2.6.x)

- 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 Linux (2.6.x)

- 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 IV

# Synchronization

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

# Section 10: Race Conditions and Critical Sections

10 Race Conditions and Critical Sections

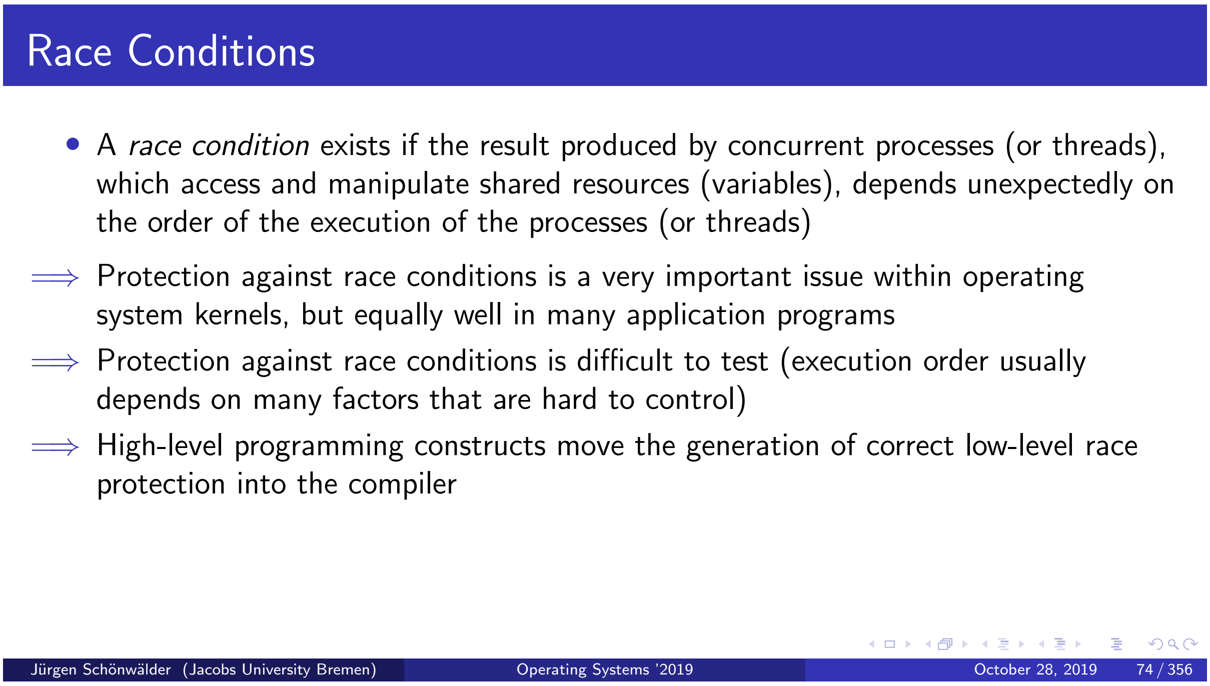
11 Synchronization Mechanisms

12 Semaphores

13 Semaphore Pattern

14 Critical Regions, Condition Variables, Messages

15 Synchronization in Java and Go and POSIX APIs



# Race Conditions

- A *race condition* exists if the result produced by concurrent processes (or threads), which access and manipulate shared resources (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 well in many application programs

⟹ Protection against race conditions is difficult to test (execution order usually depends on many factors that are hard to control)

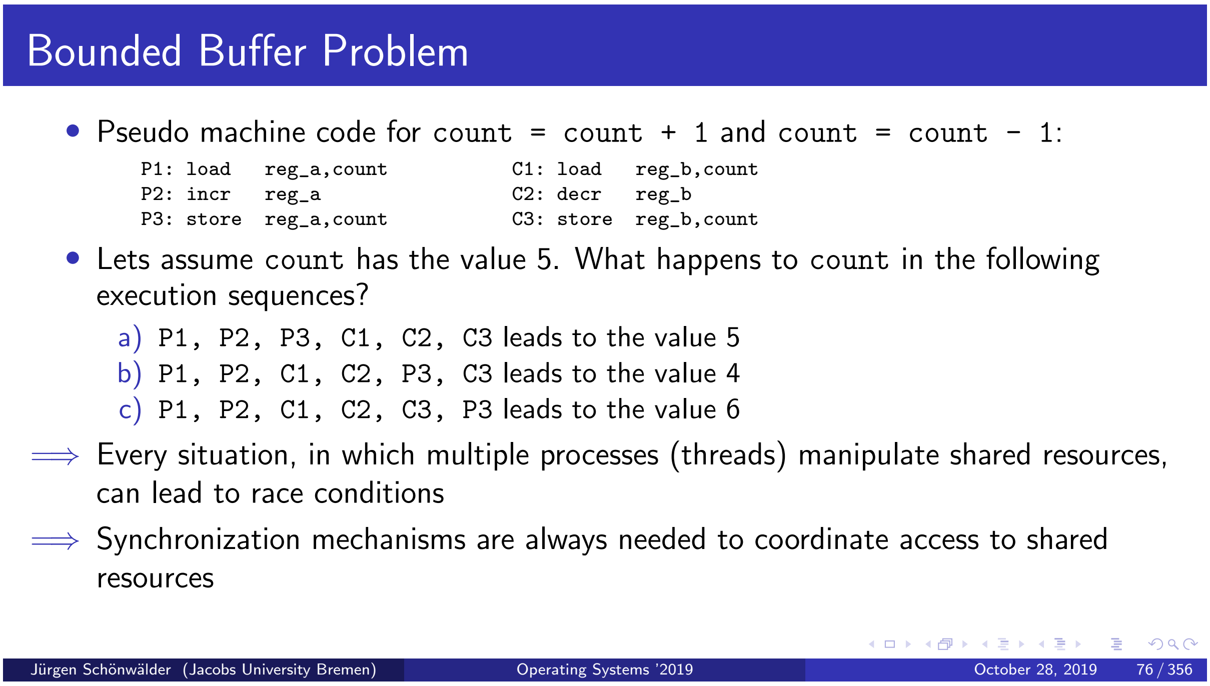
⟹ High-level programming constructs move the generation of correct low-level race protection into the compiler

# Bounded Buffer Problem

- Two processes share a common fixed-size buffer
- The producer process puts data into the buffer
- The consumer process reads data out of the buffer
- The producer must wait if the buffer is full
- The consumer must wait if the buffer is empty

```
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);
}                                       }
```

⟹ This solution has a race condition and is not correct!



## Bounded Buffer Problem

- 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
```

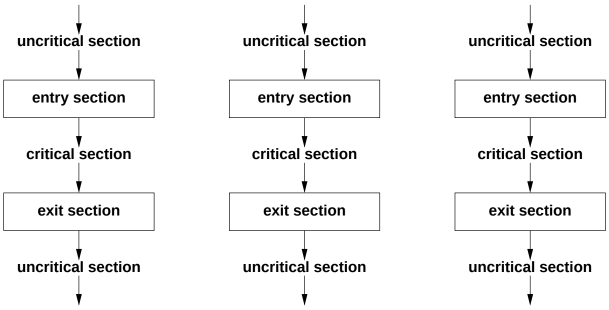
- Lets assume `count` has the value 5. What happens to `count` in the following execution sequences?
  a) `P1, P2, P3, C1, C2, C3` leads to the value 5
  b) `P1, P2, C1, C2, P3, C3` leads to the value 4
  c) `P1, P2, C1, C2, C3, P3` leads to the value 6

⟹ Every situation, in which multiple processes (threads) manipulate shared resources, can lead to race conditions

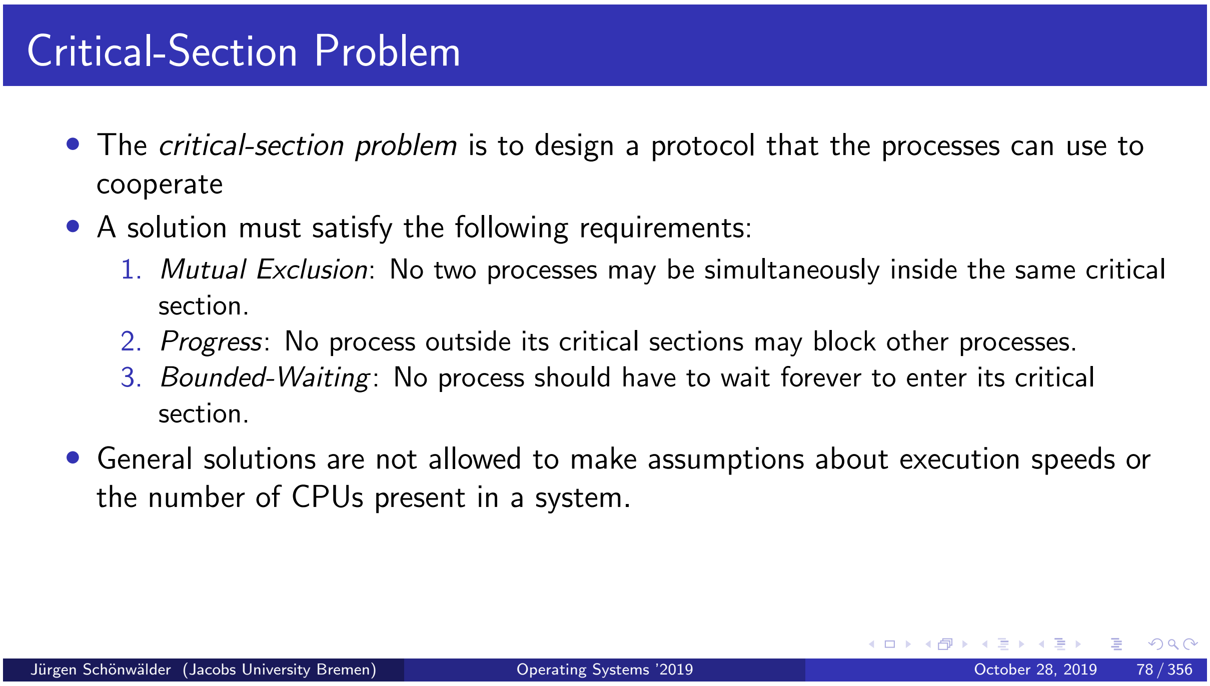
⟹ Synchronization mechanisms are always needed to coordinate access to shared resources

```c
/*
 * race/race.c --
 *
 *      A simple program demonstrating race conditions. Note that it
 *      is system specific how frequently race conditions occur.  Run
 *      this program using
 *
 *          watch -n 0.5 -d "./race | xargs -n 1 | sort -n  | xargs"
 *
 *      and lean back and you may see numbers suddenly changing.
 */

#define _POSIX_C_SOURCE 200809L

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

static unsigned int c = 0;          /* shared variable */

static void *
count(void *ignored)
{
    int i;
    const struct timespec ts = { .tv_sec = 0, .tv_nsec = 123456 };

    for (i = 0; i < 10; i++) {
        nanosleep(&ts, NULL);
        c++;
        printf(" %d", c);
    }
    return NULL;
}

int
main(int argc, char *argv[])
{
    const unsigned int num = 8;
    int i, rc;
    pthread_t t[num];

    for (i = 0; i < num; i++) {
        rc = pthread_create(&t[i], NULL, count, NULL);
        if (rc) {
            fprintf(stderr, "thread creation failed\n");
            exit(EXIT_FAILURE);
        }
    }

    for (i = 0; i < num; i++) {
        (void) pthread_join(t[i], NULL);
    }
    printf("\n");

    return EXIT_SUCCESS;
}
```

## Critical Sections

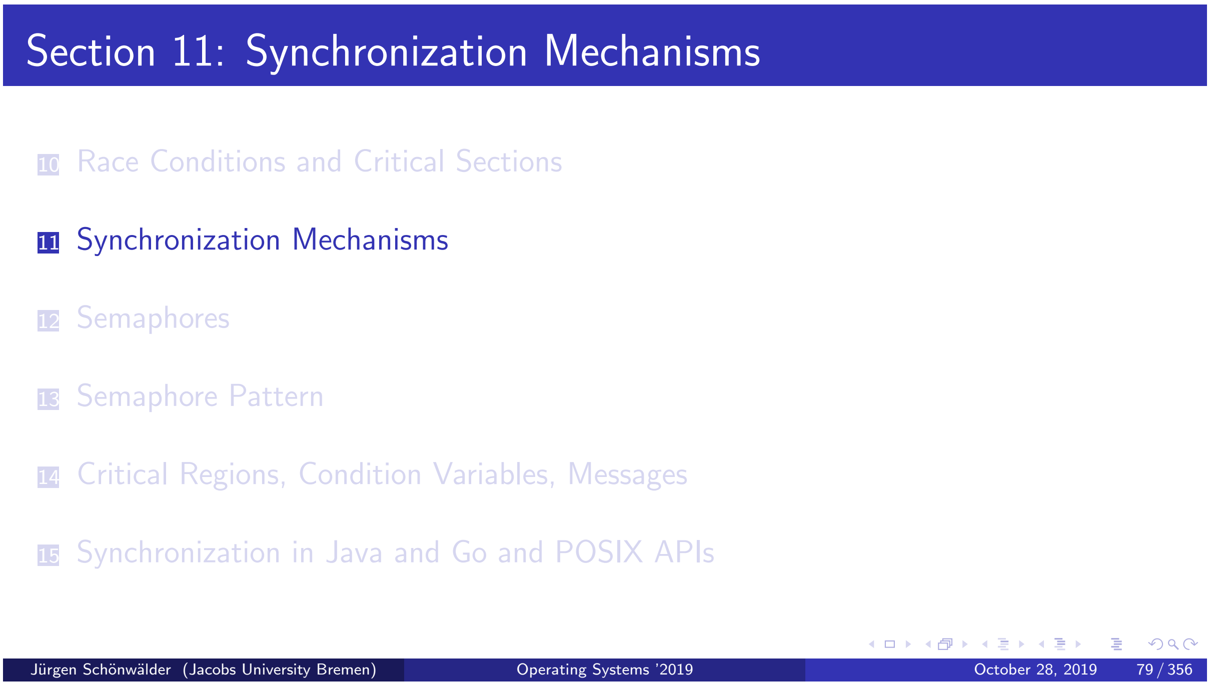


- A *critical section* is a segment of code that can only be executed by one process at a time
- The execution of critical sections by multiple processes is *mutually exclusive* in time
- Entry and exit sections must protect critical sections



## Critical-Section Problem

- 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

10 Race Conditions and Critical Sections

11 Synchronization Mechanisms

12 Semaphores

13 Semaphore Pattern

14 Critical Regions, Condition Variables, Messages

15 Synchronization in Java and Go and POSIX APIs

## Disabling Interrupts

- The simplest solution is to disable all interrupts during the critical section so that nothing can interrupt the critical section

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

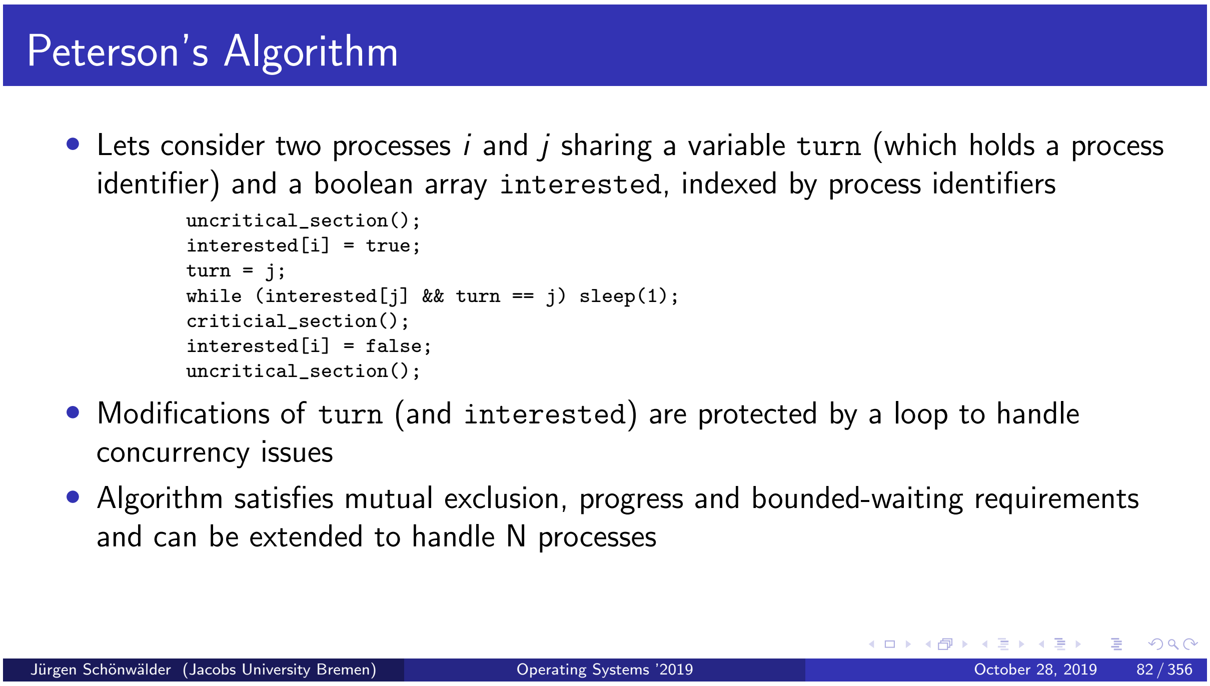
- Can usually not be used in user-space
- Problematic on systems with multiple processors
- Not usable if interrupts are needed in the critical section
- Very efficient and sometimes used in some special cases

# Strict Alternation

- Lets consider two processes sharing a variable called `turn`, which holds the values 0 and 1

```
/* 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();
```

- Ensures mutual exclusion
- Can be extended to handle alternation between N processes
- Does not satisfy the progress requirement since the solution enforces strict alternation



# Peterson's Algorithm

- Lets consider two processes *i* and *j* sharing a variable `turn` (which holds a process identifier) and a boolean array `interested`, indexed by process identifiers

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

- Modifications of `turn` (and `interested`) are protected by a loop to handle concurrency issues
- Algorithm satisfies mutual exclusion, progress and bounded-waiting requirements and can be extended to handle N processes

# Spin-Locks

- So called *spin-locks* are locks which cause the processor to spin while waiting for the lock
- Spin-locks are often used to synchronize multi-processor systems with shared memory
- Spin-locks require atomic test-and-set machine instructions on shared memory cells
- Reentrant locks do not harm if you already hold a lock

```
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, a lock was set, so try again
        ret                      ; critical region entered

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

## 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

⟹ Find alternatives which do not require busy waiting

## Section 12: Semaphores

10 Race Conditions and Critical Sections

11 Synchronization Mechanisms

12 Semaphores

13 Semaphore Pattern

14 Critical Regions, Condition Variables, Messages

15 Synchronization in Java and Go and POSIX APIs

# Semaphores

- A *semaphore* is a protected integer variable which can only be manipulated by the atomic operations `up()` and `down()`
- High-level definition of the *behavior* of semaphores:

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

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

- Dijkstra called the operations `P()` and `V()`, other popular names are `wait()` and `signal()`

# Critical Sections with Semaphores

- Critical sections are easy to implement 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 problems require more creative usage of semaphores for proper coordination

# Bounded Buffer with Semaphores

```
      const int N;
      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);
}                                    }
```

- Semaphore `mutex` protects the critical section
- Semaphore `empty` counts empty buffer space
- Semaphore `full` counts used buffer space

# Readers / Writers Problem

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

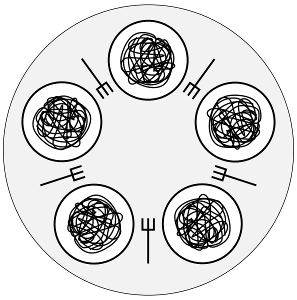
⟹ Starvation can occur in many solutions and is not taken into account here

# Readers / Writers with Semaphores

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

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

⟹ Many readers can cause starvation of writers

# 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 may not pick up only one fork at a time

# Dining Philosophers with Semaphores

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

void philosopher(int i)         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]);
    }                               }
}                               }
```

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

# Dining Philosophers with Semaphores

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

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

- The function `take_forks()` introduces a hungry state and waits for the philosopher's semaphore
- The function `put_forks()` gives the neighbors a chance to eat
- Starvation of philosophers? Fairness?
- What about trying to pick forks after waiting randomly?

## Implementation of Semaphores

- The semaphore operations `up()` and `down()` must be atomic
- On uniprocessor machines, semaphores can be implemented by either disabling interrupts during the `up()` and `down()` operations or by using a correct software solution (e.g., Peterson's algorithm)
- On multiprocessor machines, semaphores are usually implemented by using spin-locks, which themself use special machine instructions
- Semaphores are therefore often implemented on top of more primitive synchronization mechanisms

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

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

# Section 13: Semaphore Pattern

10 Race Conditions and Critical Sections

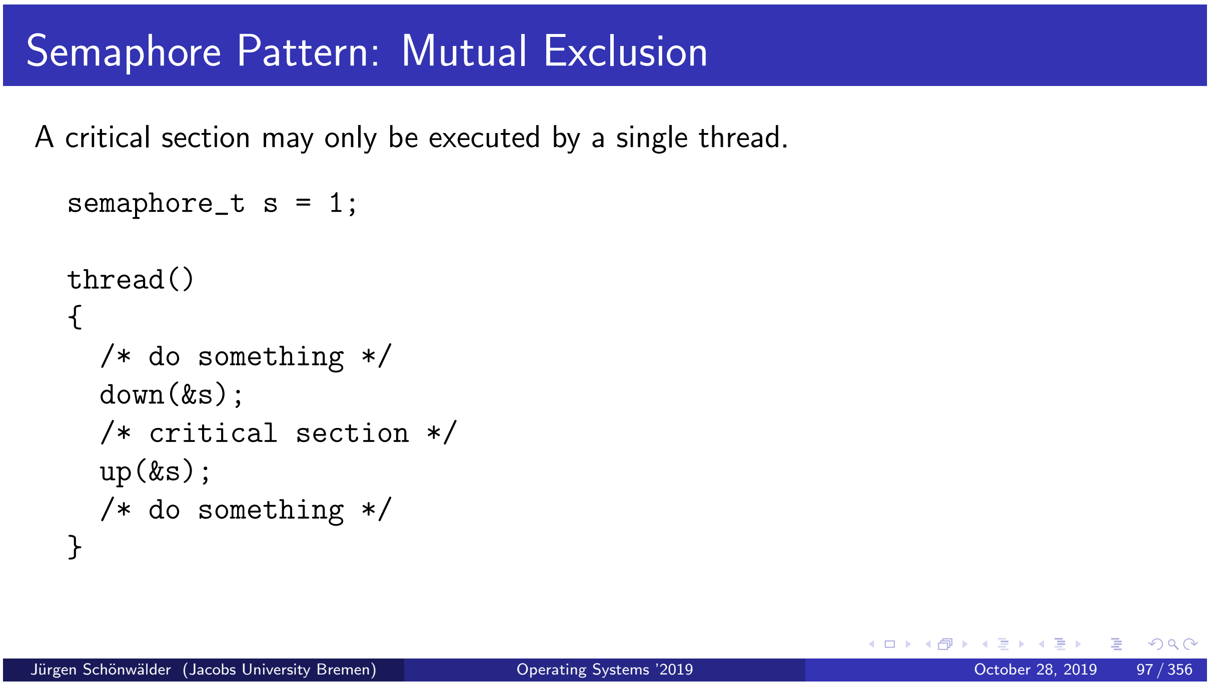
11 Synchronization Mechanisms

12 Semaphores

13 Semaphore Pattern

14 Critical Regions, Condition Variables, Messages

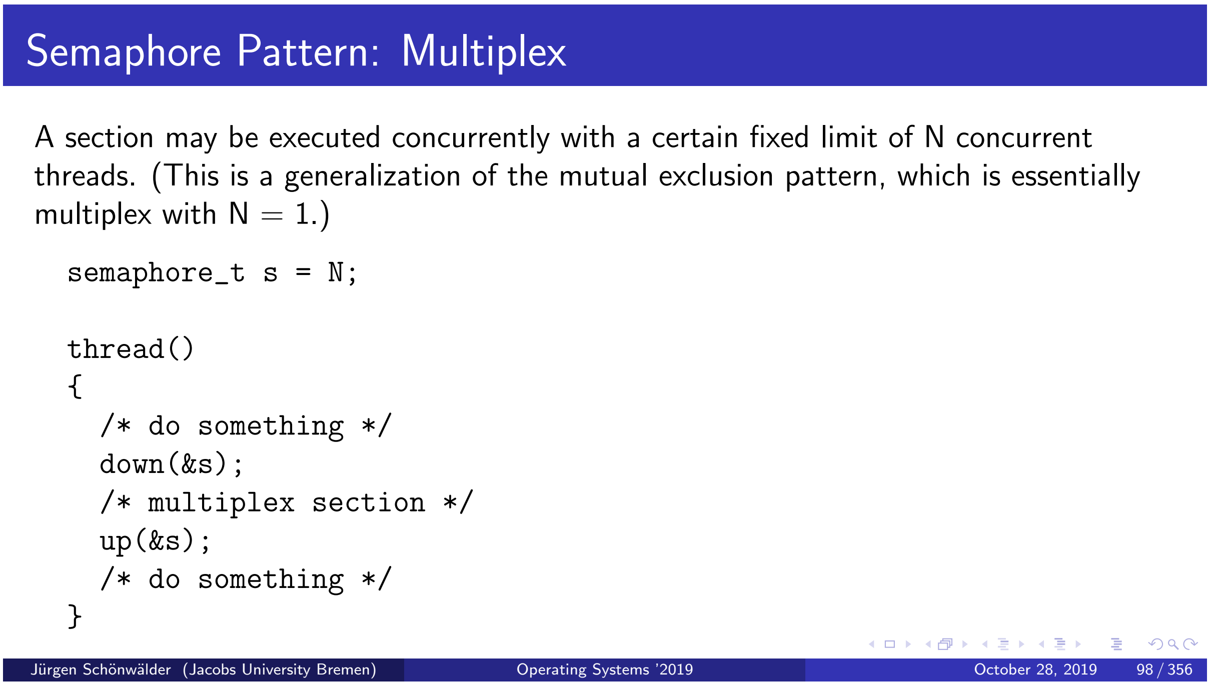
15 Synchronization in Java and Go and POSIX APIs



## 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

Next a solution allowing 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 14: Critical Regions, Condition Variables, Messages

10 Race Conditions and Critical Sections

11 Synchronization Mechanisms

12 Semaphores

13 Semaphore Pattern

14 Critical Regions, Condition Variables, Messages

15 Synchronization in Java and Go and POSIX APIs

# Critical Regions

- Simple programming errors (omissions, permutations) with semaphores usually lead to difficult to debug synchronization errors
- Idea: Let the compiler do the tedious work

```
       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;
}                               }
```

- Reduces the number of synchronization errors, does not eliminate synchronization errors

## 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 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)

## Bounded Buffer with Messages

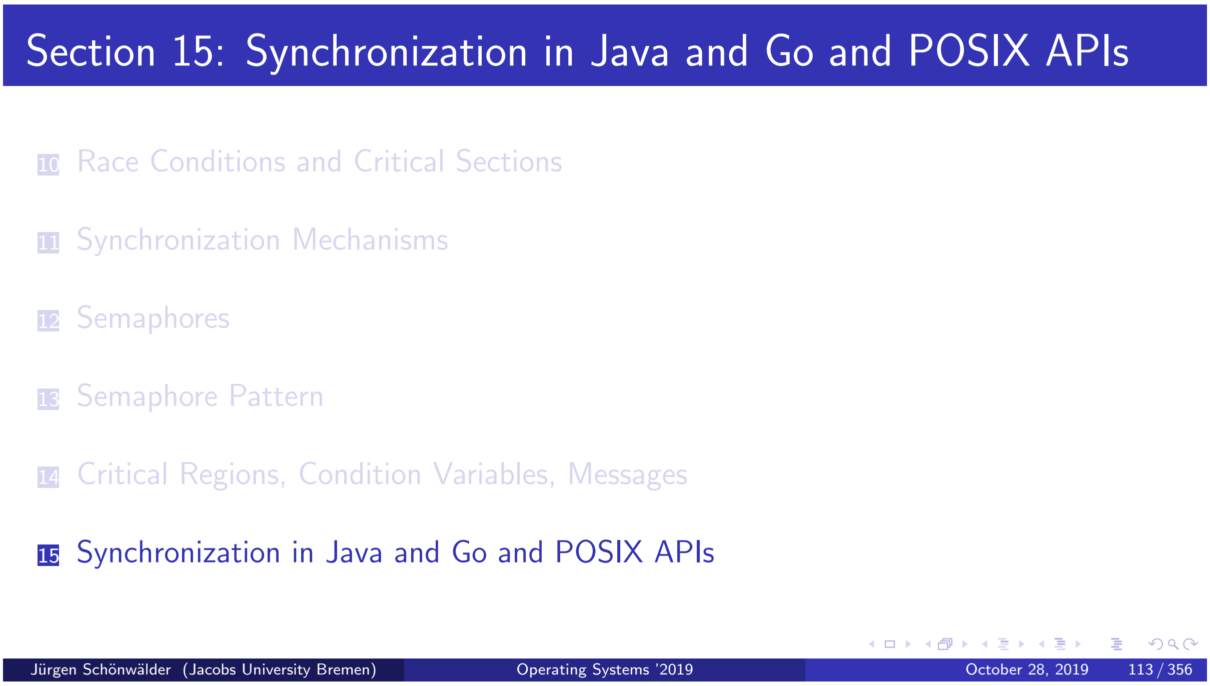
- Messages are used as tokens which control the exchange of items
- Consumers initially generate and send a number of tokens to the producers

```
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);
}                               }
```

- 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 15: Synchronization in Java and Go and POSIX APIs

10 Race Conditions and Critical Sections

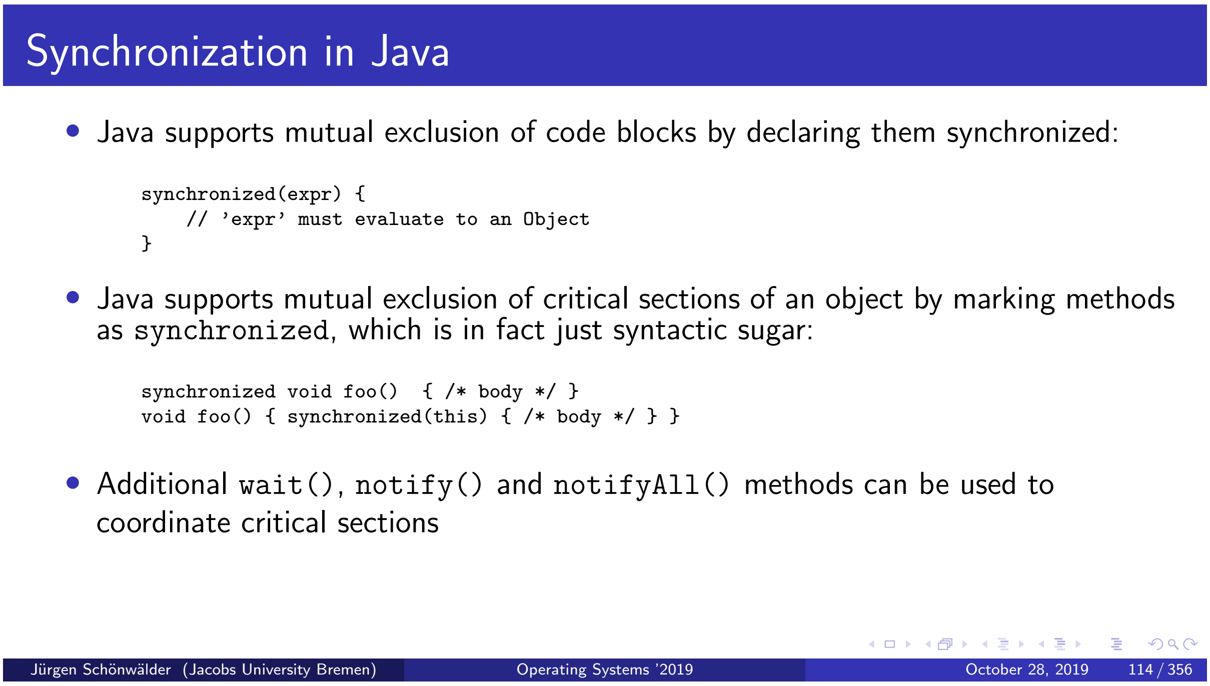
11 Synchronization Mechanisms

12 Semaphores

13 Semaphore Pattern

14 Critical Regions, Condition Variables, Messages

15 Synchronization in Java and Go and POSIX APIs



## 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

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

import java.lang.Thread;

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

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

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

/*
 * bounded/BoundedBufferTest.java --
 *
 *        Bounded buffer (producer / consumer) problem solution with
 *      Java synchronized methods. This test driver uses synchronized
 *      expressions to generate an increasing sequence of numbers
 *      concurrently.
 */

import java.lang.Runnable;
import java.lang.Thread;
import java.lang.ThreadGroup;

public class BoundedBufferTest
{
    private static BoundedBuffer buffer = new BoundedBuffer(8);

    private static class Producer implements Runnable
    {
        private static Integer fake = 0;
        private static int next = 0;

        public void run()
        {
            while (true) {
                synchronized (fake) {
                    buffer.insert(next++);
                }
            }
        }
    }

    private static class Consumer implements Runnable
    {
        public void run()
        {
            while (true) {
                int val = buffer.remove();
                System.out.println(val);
            }
        }
    }

    public static void main(String[] args)
    {
        BoundedBufferTest test = new BoundedBufferTest();
        ThreadGroup producer = new ThreadGroup("producer");
        ThreadGroup consumer = new ThreadGroup("consumer");

        for (int i = 0; i < 5; i++) {
            Thread t = new Thread(producer, new Producer());
            t.start();
        }

        for (int i = 0; i < 2; i++) {
            Thread t = new Thread(consumer, new Consumer());
            t.start();
        }
    }
}
```

## Synchronization in Go

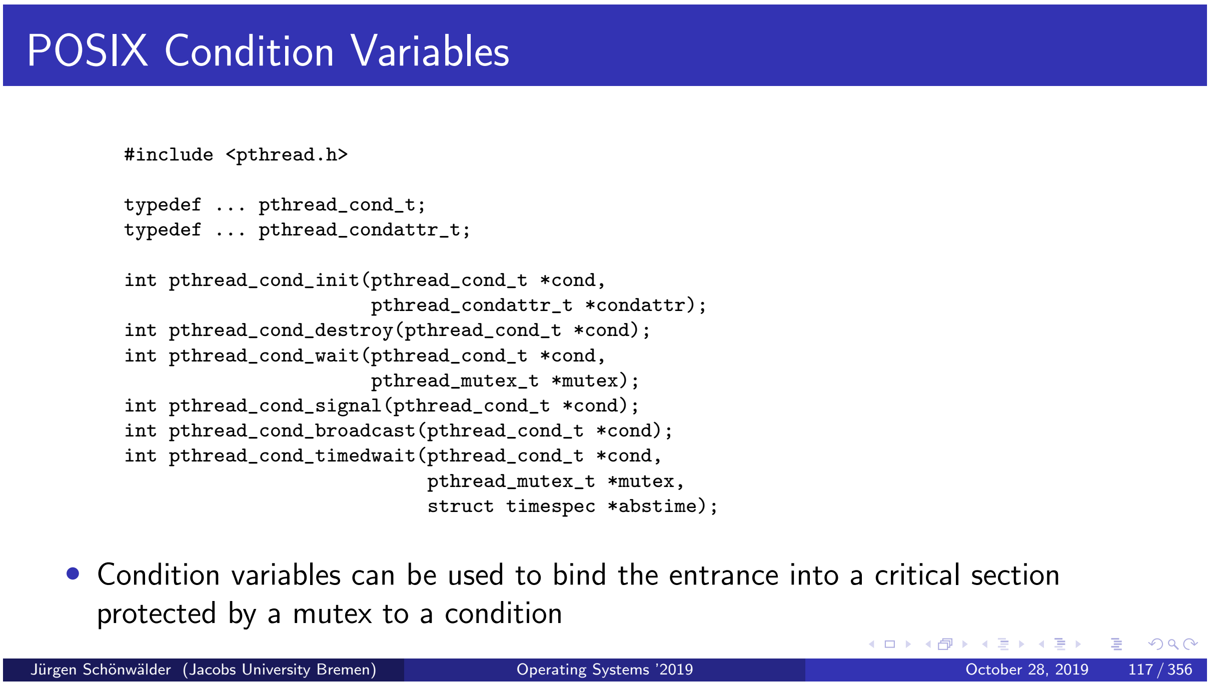
- Light-weigth "goroutines" that are typically mapped to an operating system level thread pool
- Channels provide message queues between goroutines
- Philosophy: Do not communicate by sharing memory; instead, share memory by communicating
- Inspired by Hoare's work on Communicating Sequential Processes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

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

package main

import (
        "flag"
        "fmt"
)

const (
        size = 12
)

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

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

func discarder() (chan<- int, <-chan bool) {
        d := make(chan int)
        r := make(chan bool)
        cnt := 0
        go func() {
                for {
                        r <- true
                        v := <-d
                        cnt++
                        if *ve {
                                fmt.Printf(".")
                        }
                        if cnt != v {
                                panic(fmt.Sprintf("unexpected number %d (expected %d)", v, cnt))
                        }
                }
        }()
        return d, r
}

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

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

func run(nc int, np int) {
        b := make(chan int, size) // bounded buffer
        g, n := generator()       // lock-step generator
        d, r := discarder()       // lock-step discarder
        for i := 0; i < np; i++ {
                go producer(b, g, n)
        }
        for i := 0; i < nc; i++ {
                go consumer(b, d, r)
        }
}

func main() {
        flag.Parse()
        run(*nc, *np)
        <-make(chan struct{}) // block on a channel that never delivers
}
```

# 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_unlock(pthread_mutex_t *mutex);
int pthread_mutex_trylock(pthread_mutex_t *mutex);
int pthread_mutex_timedlock(pthread_mutex_t *mutex,
                            struct timespec *abstime);
```

- Mutex locks are a simple mechanism to achieve mutual exclusion in critical sections



## 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_signal(pthread_cond_t *cond);
int pthread_cond_broadcast(pthread_cond_t *cond);
int pthread_cond_timedwait(pthread_cond_t *cond,
                           pthread_mutex_t *mutex,
                           struct timespec *abstime);
```

- Condition variables can be used to bind the entrance into a critical section protected by a mutex to a condition

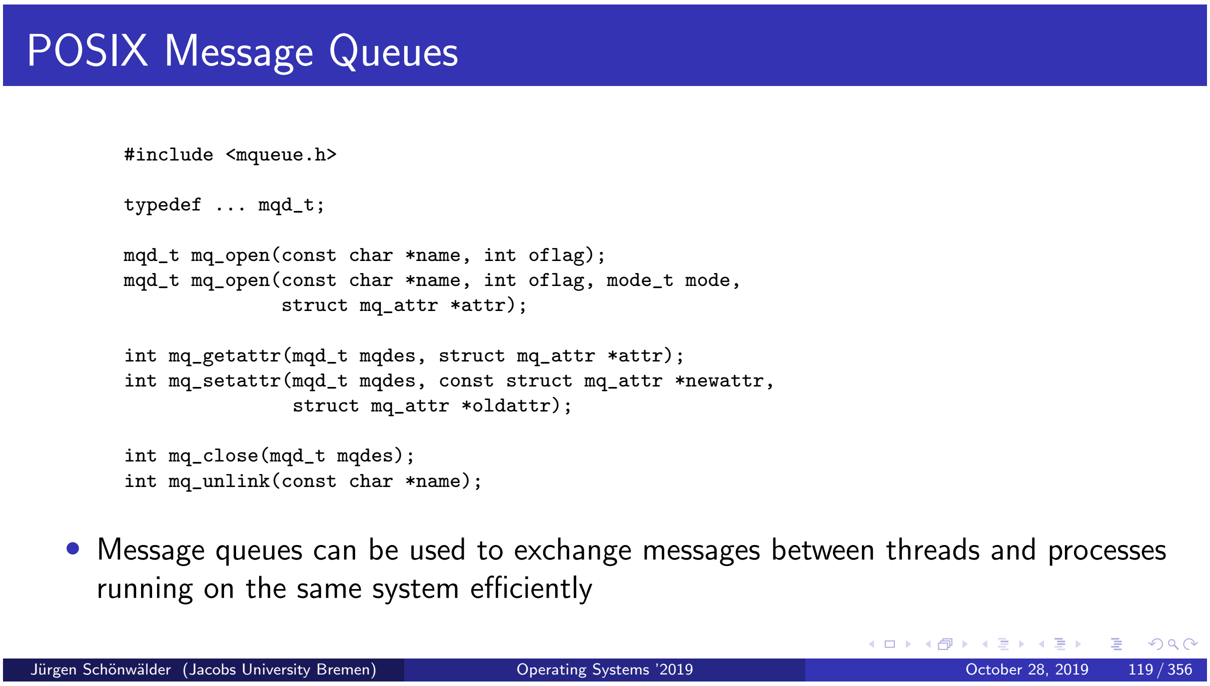
```
/*
 * bounded/bounded.c --
 *
 *      Bounded buffer (producer / consumer) problem solution with
 *      pthreads and condition variables.
 */

#define _REENTRANT
#define _DEFAULT_SOURCE
#define _XOPEN_SOURCE

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

#define BUFFER_SIZE 12

typedef struct buffer {
    unsigned int    count;
    unsigned int    data[BUFFER_SIZE];
    int             in;
    int             out;
    pthread_mutex_t mutex;
    pthread_cond_t  empty;
    pthread_cond_t  full;
} buffer_t;

static buffer_t shared_buffer = {
    .count = 0,
    .in    = 0,
    .out   = 0,
    .mutex = PTHREAD_MUTEX_INITIALIZER,
    .empty = PTHREAD_COND_INITIALIZER,
    .full  = PTHREAD_COND_INITIALIZER
```

```
};

static const char *progname = "bounded";

static unsigned int
next()
{
    static unsigned int cnt = 0;
    return ++cnt;
}

static void
check(unsigned int num)
{
    static unsigned int cnt = 0;
    assert(num == ++cnt);
}

static void*
producer(void *data)
{
    buffer_t *buffer = (buffer_t *) data;

    while (1) {
        (void) pthread_mutex_lock(&buffer->mutex);
        while (buffer->count == BUFFER_SIZE) {
            (void) pthread_cond_wait(&buffer->empty, &buffer->mutex);
        }
        buffer->data[buffer->in] = next();
        buffer->in = (buffer->in + 1) % BUFFER_SIZE;
        buffer->count++;
        (void) pthread_cond_signal(&buffer->full);
        (void) pthread_mutex_unlock(&buffer->mutex);
    }
    return NULL;
}

static void*
consumer(void *data)
{
    buffer_t *buffer = (buffer_t *) data;

    while (1) {
        (void) pthread_mutex_lock(&buffer->mutex);
        while (buffer->count == 0) {
            (void) pthread_cond_wait(&buffer->full, &buffer->mutex);
        }
        check(buffer->data[buffer->out]);
        buffer->out = (buffer->out + 1) % BUFFER_SIZE;
        buffer->count--;
        (void) pthread_cond_signal(&buffer->empty);
        (void) pthread_mutex_unlock(&buffer->mutex);
    }
    return NULL;
}

static int
run(int nc, int np)
{
    int err, n = nc + np;
    pthread_t thread[n];

    for (int i = 0; i < n; i++) {
        err = pthread_create(&thread[i], NULL,
                             i < nc ? consumer : producer, &shared_buffer);
        if (err) {
            fprintf(stderr, "%s: %s(): unable to create thread %d: %s\n",
                    progname, __func__, i, strerror(err));
            return EXIT_FAILURE;
        }
    }

    for (int i = 0; i < n; i++) {
        if (thread[i]) {
            err = pthread_join(thread[i], NULL);
            if (err) {
                fprintf(stderr, "%s: %s(): unable to join thread %d: %s\n",
                        progname, __func__, i, strerror(err));
            }
        }
    }

    return EXIT_SUCCESS;
}

int
main(int argc, char *argv[])
{
    int c, nc = 1, np = 1;

    while ((c = getopt(argc, argv, "c:p:h")) >= 0) {
        switch (c) {
        case 'c':
            if ((nc = atoi(optarg)) <= 0) {
                fprintf(stderr, "number of consumers must be > 0\n");
                exit(EXIT_FAILURE);
            }
            break;
        case 'p':
            if ((np = atoi(optarg)) <= 0) {
                fprintf(stderr, "number of producers must be > 0\n");
                exit(EXIT_FAILURE);
            }
            break;
        case 'h':
            printf("Usage: %s [-c consumers] [-p producers] [-h]\n", progname);
            exit(EXIT_SUCCESS);
        }
    }

    return run(nc, np);
}
```

# POSIX Barriers

```
#include <pthread.h>

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

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

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



## POSIX Message Queues

```
#include <mqueue.h>

typedef ... mqd_t;

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

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

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

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

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

There are far more flexible message queues that can work efficient locally (i.e., between threads) and in a distributed applicaiton (i.e., processes running on different computers). Interested readers should lookup ZeroMQ[10] and nanomsg[11].

[10] http://zeromq.org/

[11] https://nanomsg.org/

# POSIX Message Queues

```
#include <mqueue.h>

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

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 *abs_timeout);

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

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

# POSIX Semaphores

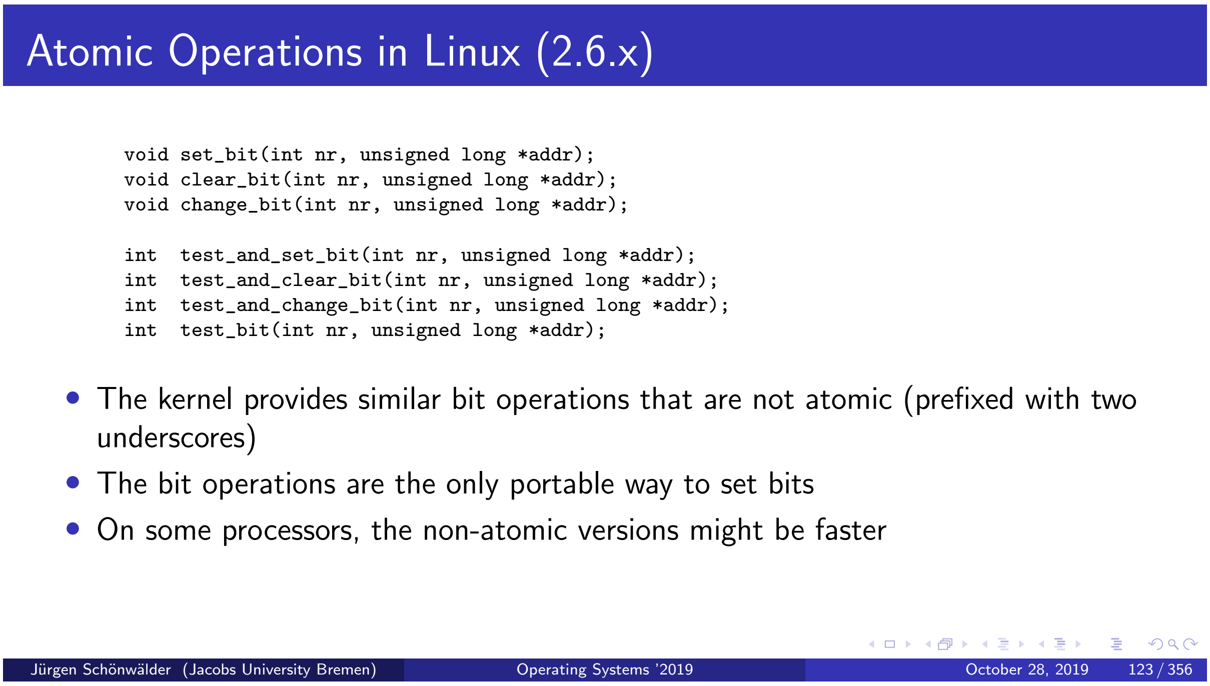
```
#include <semaphore.h>

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_post(sem_t *sem);
int sem_trywait(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);
```

- Unnamed semaphores are created with (`sem_init()`)
- Named semaphores are created with (`sem_open()`)

# Atomic Operations in Linux (2.6.x)

```
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);
```

- The `atomic_t` is essentially 24 bit wide since some processors use the remaining 8 bits of a 32 bit word for locking purposes



# Atomic Operations in Linux (2.6.x)

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

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

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

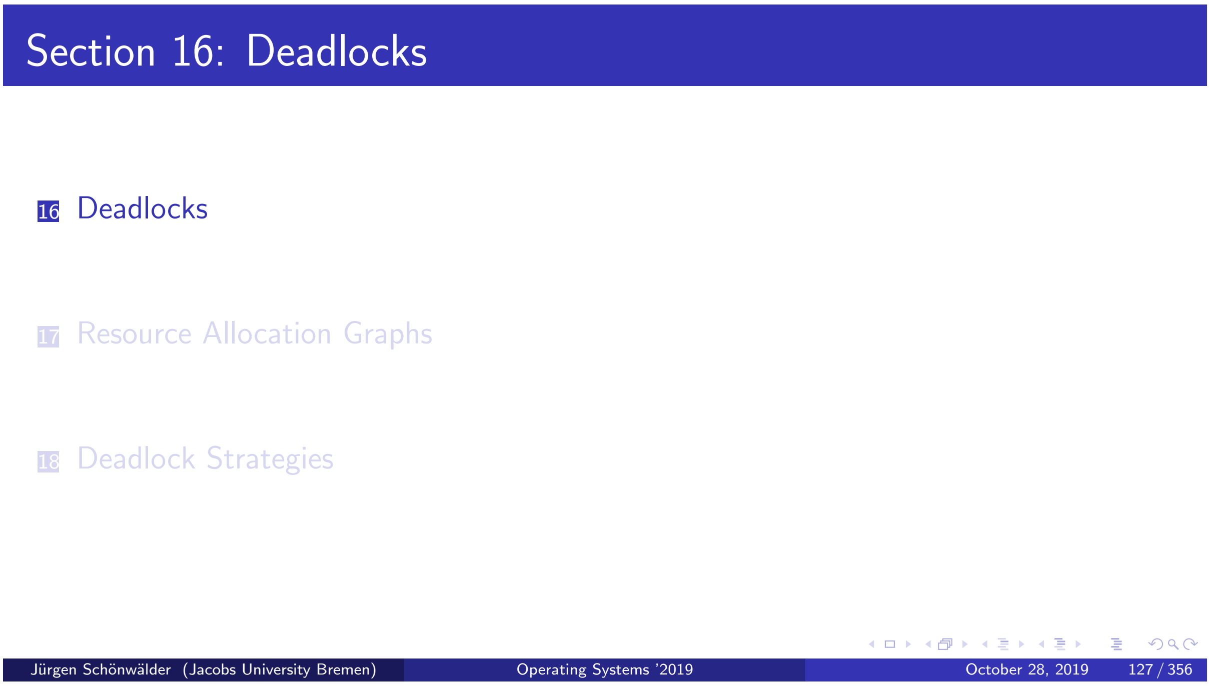
# Semaphores in Linux (2.6.x)

```
struct ... semaphore;

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

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

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

- Linux kernel semaphores are counting semaphores
- `init_MUTEX(s)` equals `sema_init(s, 1)`
- `init_MUTEX_LOCKED(s)` equals `sema_init(s, 0)`

Part V

# Deadlocks



# Section 16: Deadlocks

16 Deadlocks

17 Resource Allocation Graphs

18 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 lead to 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
- Should the operating system not prevent deadlocks?

## Necessary Deadlock Conditions

- *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

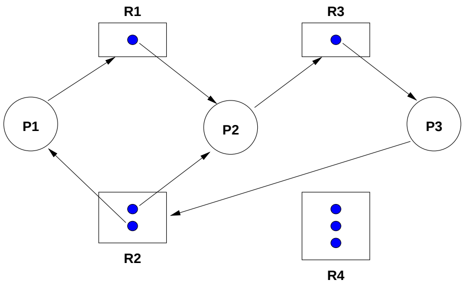
# Section 17: Resource Allocation Graphs

16 Deadlocks

17 Resource Allocation Graphs

18 Deadlock Strategies

# Resource-Allocation Graph (RAG)



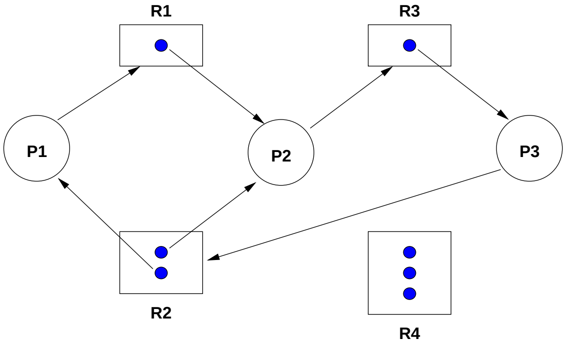
$$RAG = \{V, E\}$$
$$V = P \cup R$$
$$E = E_c \cup E_r \cup E_a$$

| | |
|---|---|
| $P = \{P_1, P_2, \ldots, P_n\}$ | (processes) |
| $R = \{R_1, R_2, \ldots, R_m\}$ | (resource types) |
| $E_c = \{P_i \rightarrow R_j\}$ | (resource claims (future)) |
| $E_r = \{P_i \rightarrow R_j\}$ | (resource requests (current)) |
| $E_a = \{R_i \rightarrow P_j\}$ | (resource assignments) |

# RAG Properties

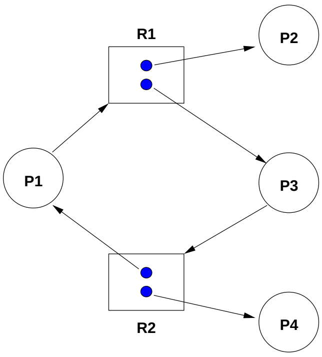
- Properties of a Resource-Allocation Graph:
  - A cycle in the RAG is a necessary condition for a deadlock
  - If each resource type has exactly one instance, then a cycle is also a sufficient condition for a deadlock
  - If each resource type has several instances, then a cycle is not a sufficient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

# RAG Example #1



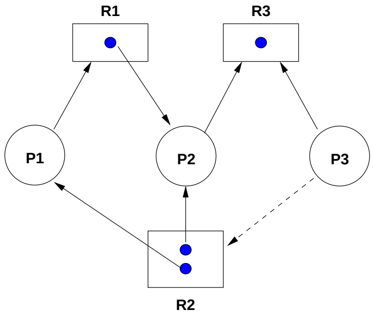
- Cycle 1: $P_1 \rightarrow R_1 \rightarrow P_2 \rightarrow R_3 \rightarrow P_3 \rightarrow R_2 \rightarrow P_1$
- Cycle 2: $P_2 \rightarrow R_3 \rightarrow P_3 \rightarrow R_2 \rightarrow P_2$
- Processes $P_1$, $P_2$ and $P_3$ are deadlocked

## RAG Example #2



- Cycle: $P_1 \rightarrow R_1 \rightarrow P_3 \rightarrow R_2 \rightarrow P_1$
- Processes $P_1$ and $P_3$ are not deadlocked
- $P_4$ may release its instance of $R_2$, breaking the cycle

# RAG Example #3



- $P_2$ and $P_3$ both request $R_3$. To which process should the resource be assigned?
- Assign $R_3$ to $P_2$ to avoid a future deadlock situation

# Section 18: Deadlock Strategies

16 Deadlocks

17 Resource Allocation Graphs

18 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

- Ensure that at least one of the necessary conditions cannot hold
- Prevent mutual exclusion:
  Some resources are intrinsically non-sharable
- Prevent hold and wait:
  Low resource utilization and starvation possible
- Prevent no preemption:
  Preemption can not be applied to some resources such as printers or tape drives
- Prevent circular wait:
  Leads to low resource utilization and starvation if the imposed order does not match process requirements

⟹ Prevention is not feasible in the general case

# Deadlock Avoidance

- Definitions:
  - A state is *safe* if the system can allocate resources to each process (up to its claimed maximum) and still avoid a deadlock
  - A state is *unsafe* if the system cannot prevent processes from requesting resources such that a deadlock occurs
- 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

- There are $n$ processes and $m$ resource types
- Let $i \in 1, \ldots, n$ and $j \in 1, \ldots m$
- $total[j]$: list of total number of resources of type $j$
- $max[i, j]$: maximum number of resources of type $j$ that can be claimed by process $i$
- $alloc[i, j]$: number of resources of type $j$ allocated to process $i$
- $avail[j]$: number of available resources of type $j$
- $need[i, j]$: number of still to be requested resources of type $j$ by process $i$
- $ready$: list of processes that can get the needed resources allocated

# Safe-State Algorithm

```
function IsSafe($total, max, alloc$)
    loop
        $need \leftarrow max - alloc$                         ▷ Needed resources
        $avail \leftarrow total - colsum(alloc)$              ▷ Currently available resources
        $ready \leftarrow filter(need, avail)$                ▷ Processes that can get their resources
        if $ready \equiv \emptyset$ then
            return $(alloc \equiv \emptyset)$                 ▷ Safe if alloc is empty, otherwise unsafe
        end if
        $proc \leftarrow select(ready)$                       ▷ Select a process that is ready
        $alloc \leftarrow remove(alloc, proc)$                ▷ Remove process from alloc
        $alloc \leftarrow remove(max, proc)$                  ▷ Remove process from max
    end loop
end function
```

## Resource-Request Algorithm

```
function RESQUESTRESOURCES($total, max, alloc, request$)
    $need \leftarrow max - alloc$                              ▷ Needed resources
    $avail \leftarrow total - colsum(alloc)$                   ▷ Currently available resources
    if $request > need$ then
        error $illegal$                                        ▷ Request exceeds available resources
    end if
    if $request \geq avail$ then
        $alloc' \leftarrow alloc + request$                    ▷ Pretend to grant the request
        if $isSafe(total, max, alloc')$ then                   ▷ Check whether the new state is safe
            return $True$                                      ▷ Grant the resource request since its safe
        end if
    end if
    return $False$                                             ▷ Request not granted at this point in time
end function
```

# Banker's Algorithm Example

- System description:

$$m = 4 \quad \text{resource types}$$

$$n = 5 \quad \text{processes}$$

$$Total = (6, 8, 10, 12)$$

$$Max \quad = \quad \begin{pmatrix} 3 & 1 & 2 & 5 \\ 3 & 2 & 5 & 7 \\ 2 & 6 & 3 & 1 \\ 5 & 4 & 9 & 2 \\ 1 & 3 & 8 & 9 \end{pmatrix}$$

# Banker's Algorithm Example

- Can the system get into the state described by the following allocation matrix?

$$Alloc \quad = \quad \begin{pmatrix} 0 & 0 & 2 & 1 \\ 1 & 0 & 1 & 2 \\ 1 & 2 & 1 & 1 \\ 3 & 4 & 0 & 0 \\ 0 & 0 & 4 & 2 \end{pmatrix}$$

# Banker's Algorithm Example

- Check whether the given state is safe:

$$Avail = (1, 2, 2, 6)$$

$$Need = \begin{pmatrix} 3 & 1 & 0 & 4 \\ 2 & 2 & 4 & 5 \\ 1 & 4 & 2 & 0 \\ 2 & 0 & 9 & 2 \\ 1 & 3 & 4 & 7 \end{pmatrix}$$

- The system may never reach this state!

# Banker's Algorithm Example

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

$$Alloc = \begin{pmatrix} 1 & 0 & 2 & 1 \\ 1 & 1 & 2 & 5 \\ 1 & 2 & 3 & 1 \\ 1 & 1 & 1 & 1 \\ 1 & 0 & 2 & 2 \end{pmatrix}$$

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

# Banker's Algorithm Example

- Assume the request can be granted:

$$
\mathit{Alloc} = \begin{pmatrix} 1 & 0 & 2 & 1 \\ 1 & 1 & 2 & 5 \\ 1 & 2 & 3 & 1 \\ 1 & 1 & 1 & 2 \\ 1 & 0 & 2 & 2 \end{pmatrix} \qquad \mathit{Need} = \begin{pmatrix} 2 & 1 & 0 & 4 \\ 2 & 1 & 3 & 2 \\ 1 & 4 & 0 & 0 \\ 4 & 3 & 8 & 0 \\ 0 & 3 & 6 & 7 \end{pmatrix}
$$

- Is it still possible to satisfy the maximum claims?

# Banker's Algorithm Example

- Maximum claims can be satisfied as shown below:

| *Avail* | Action |
|---|---|
| $(1,4,0,1)$ | termination of process 3 |
| $(2,6,3,2)$ | termination of process 2 |
| $(3,7,5,7)$ | termination of process 1 |
| $(4,7,7,8)$ | termination of process 5 |
| $(5,7,9,10)$ | termination of process 4 |
| $(6,8,10,12)$ | stop |

- The new state is safe and the request can be granted.

# Deadlock Detection

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

# Detection Algorithm

1. Initialize:

    $Work \leftarrow Avail$

    $\forall i = 1, \ldots, n : Finish[i] \leftarrow false$

2. Select:

    *Find a process i such that for* $j = 1, \ldots, m$ $Finish[i] = false \wedge Request[i,j] \leq Work[j]$

    *If no such process i exists, go to step 4.*

3. Update:

    $Work[j] \leftarrow Work[j] + Alloc[i,j]$ *for* $j = 1, \ldots, m$, $Finish[i] \leftarrow true$, *go to step 2.*

4. Finish:

    *Deadlock if* $Finish[i] = false$ *for some* $i$, $1 \leq i \leq n$

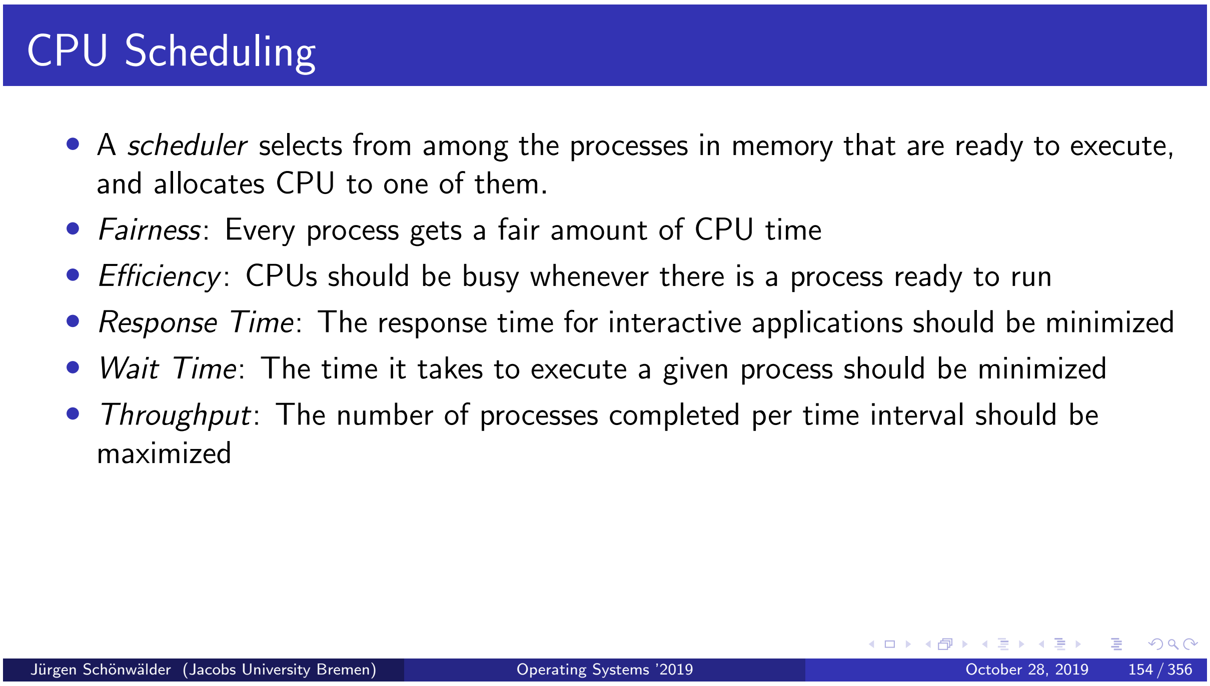
# Part VI

# Scheduling

# Section 19: CPU Scheduling

19 CPU Scheduling

20 CPU Scheduling Strategies



## CPU Scheduling

- A *scheduler* selects from among the processes in memory that are ready to execute, and allocates CPU to one of them.
- *Fairness*: Every process gets a fair amount of CPU time
- *Efficiency*: CPUs should be busy whenever there is a process ready to run
- *Response Time*: The response time for interactive applications should be minimized
- *Wait Time*: The time it takes to execute a given process should be minimized
- *Throughput*: The number of processes completed per time interval should be maximized

# Preemptive Scheduling

- A *preemptive* scheduler can interrupt a running process and assign the CPU to another process
- A *non-preemptive* scheduler waits for the process to give up the CPU once the CPU has been assigned to the process
- Non-preemptive schedulers cannot guarantee fairness
- Preemptive schedulers are harder to design
- Preemptive schedulers might preempt the CPU at times where the preemption is costly (e.g., in the middle of a critical section)

## Deterministic vs. Probabilistic

- A *deterministic* scheduler knows the execution times of the processes and optimizes the CPU assignment to optimize system behavior (e.g., maximize throughput)
- A *probabilistic* scheduler describes process behavior with certain probability distributions (e.g., process arrival rate distribution) and optimizes the overall system behavior based on these probabilistic assumptions

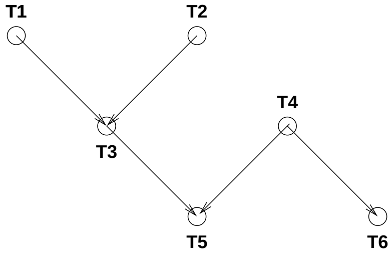
- Deterministic schedulers are relatively easy to analyze
- Finding optimal schedules is a complex problem
- Probabilistic schedulers must be analyzed using stochastic models (queuing models)

## Deterministic Scheduling

- A *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 $t = \{t_1, t_2, \ldots t_n\}$ 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.

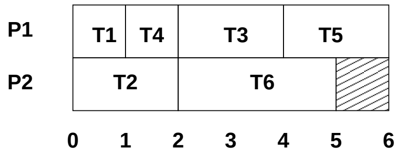
# Deterministic Scheduling Example

- $T = \{T_1, T_2, T_3, T_4, T_5, T_6\}, n = 6$
- $t_1 = t_4 = 1, t_2 = t_3 = t_5 = 2, t_6 = 3$
- $G = (T, E)$
- $E = \{(T_1, T_3), (T_2, T_3), (T_3, T_5), (T_4, T_5), (T_4, T_6)\}$
- $P = \{P_1, P_2\}, m = 2$



# Gantt Diagrams

- Schedules are often visualized using Gantt diagrams:



- Let $e = \{e_1, e_2, \dots, e_n\}$ denote the termination time of the task $t_i \in T$ in the schedule $S$. The length of the schedule $t(S)$ and the average wait time $\overline{e}$ are defined as follows:

$$t(S) = \max_{1 \leq i \leq n} \{e_i\} \qquad \overline{e} = \frac{1}{n} \sum_{i=1}^{n} e_i$$

# Section 20: CPU Scheduling Strategies

19 CPU Scheduling

20 CPU Scheduling Strategies

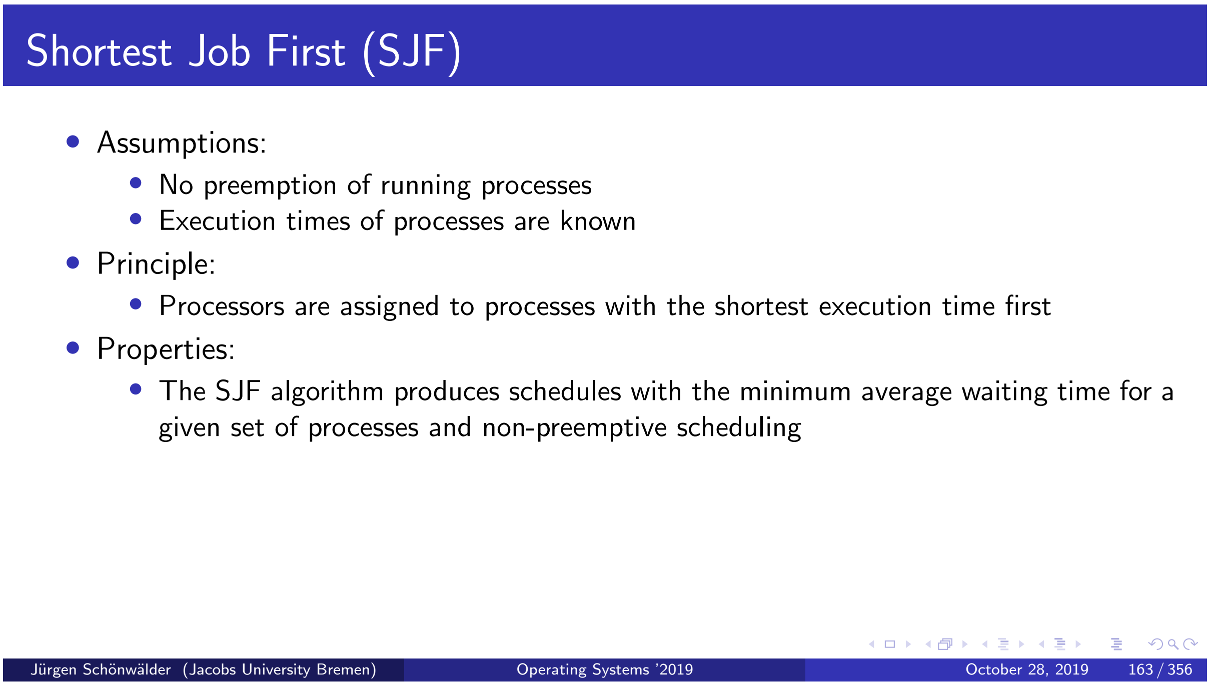
# First-Come, First-Served (FCFS)

- Assumptions:
  - No preemption of running processes
  - Arrival and execution 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 wait 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:
  - For the length $t(S_L)$ of an LPTF schedule $S_L$ and the length $t(S_O)$ of an optimal schedule $S_O$, the following holds:

$$t(S_L) \leq (\frac{4}{3} - \frac{1}{3m}) \cdot t(S_O)$$



## Shortest Job First (SJF)

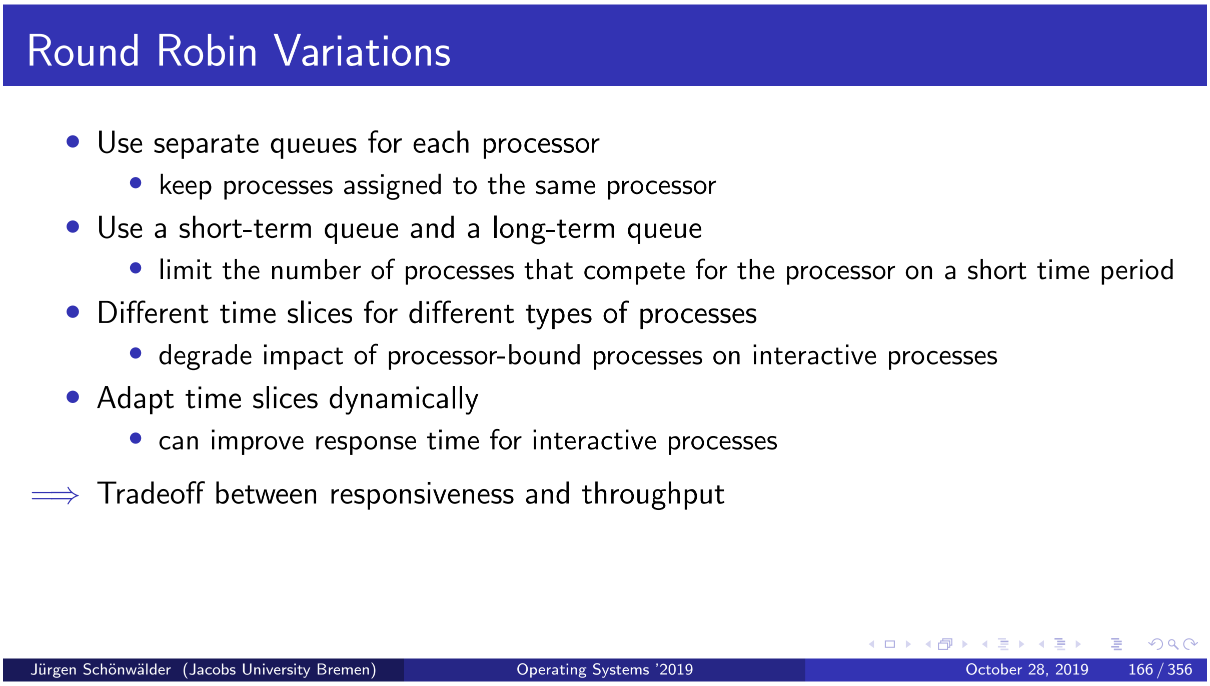
- Assumptions:
  - No preemption of running processes
  - Execution times of processes are known
- Principle:
  - Processors are assigned to processes with the shortest execution time first
- Properties:
  - The SJF algorithm produces schedules with the minimum average waiting time for a given set of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

# 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 $\rightarrow \infty$: FCFS scheduling
  - time slice $\rightarrow 0$: processor sharing (idealistic)
  - Choosing a “good” time slice is important

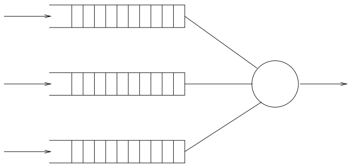


# Round Robin Variations

- 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

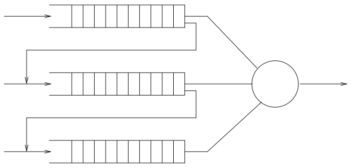
⟹ Tradeoff between responsiveness and throughput

# 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 (2.6.x)

```
#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);
```

# Parallel Computing Libraries

- OpenMP
  - Parallel computer API for shared memory systems. The OpenMP API is operating system independent and provides high-level compiler contructs (via C pragmas) for managing threads.
- OpenCL / OpenCL C
  - Parallel programming of heterogenous plattforms consisting of CPUs, GPUs, and DSPs. OpenCL provides access to GPUs for non graphical computing.
- Open MPI
  - A message passing library for distributed parallel computing based on the Message-Passing Interface standard (MPI)

# Part VII

# Linking

# Section 21: Linker

21 Linker

22 Libraries

23 Interpositioning

# C Compilation Process

```
C preprocessor -> expanded C code     (gcc -E hello.c)
       v                 v
  C compiler   -> assembler code      (gcc -S hello.c)
       v                 v
   assembler   -> object code         (gcc -c hello.c)
       v                 v
    linker     -> executable          (gcc hello.c)
```

- Compiling C source code is traditionally a four-stage process.
- Modern compilers often integrate stages for efficiency reasons.

## Reasons for using a Linker

- Modularity
  - Programs can be we written as a collection of small files
  - Building a collection 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 associates 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 load time or run-time

# Executable and Linkable Format

- Standard unified binary format for all object files
- ELF header provides basic information (word size, endianess, machine architecture, . . . )
- 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 traditional 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 Rules:
  - Rule 1: Multiple strong symbols with the same name are not allowed
  - Rule 2: Given a strong symbol and multiple weak symbols with the same name, choose the strong symbol
  - 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 overwrite `y`:

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

# Section 22: Libraries

21 Linker

22 Libraries

23 Interpositioning

## Static Libraries

- Collect related relocatable object files into a single file with an index (called an archive)
- Enhance 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

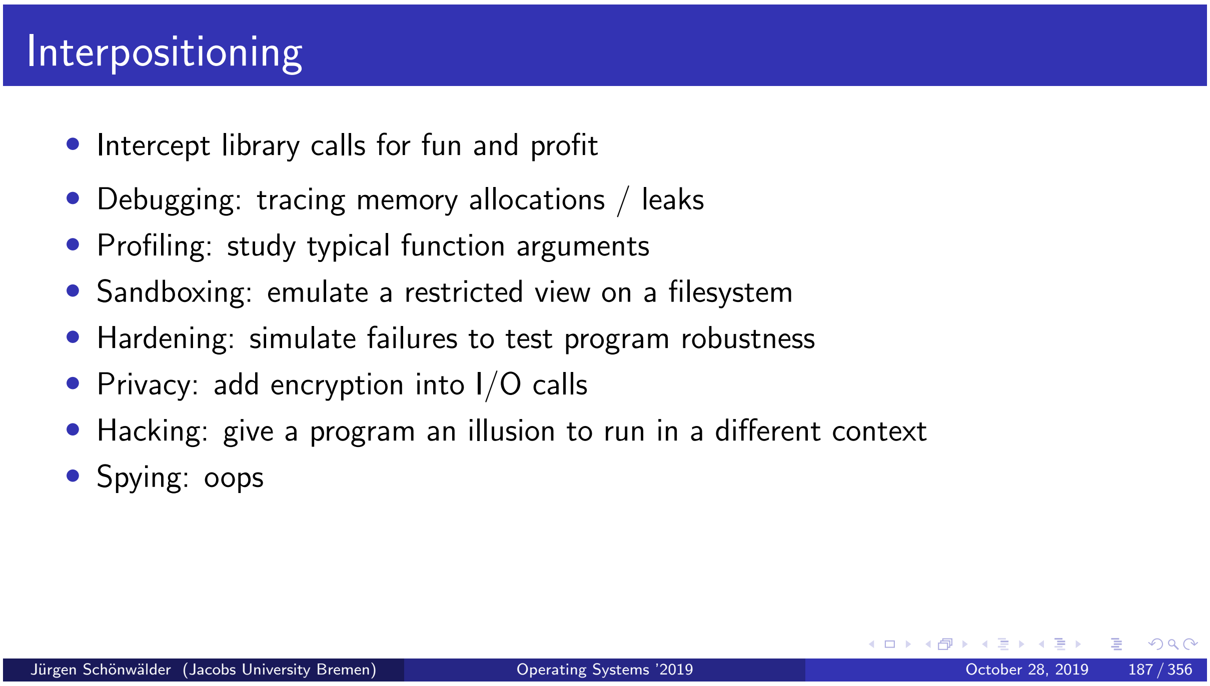
- Static linking duplicates library code by copying it into executables
- Bug fixes in libraries require to re-link all executables
- Solution: Delay the linking until program start and then link against the most recent matching versions of the required libraries
- At traditional link time, an executable 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
- Benefit #1: Library machine code can be stored in memory shared by multiple processes
- Benefit #2: Programs can load additional code dynamically while the program is running
- Caveat: Loading untrusted libraries can lead to big surprises

# Section 23: Interpositioning

21 Linker

22 Libraries

23 Interpositioning



## Interpositioning

- Intercept library calls for fun and profit
- Debugging: tracing memory allocations / leaks
- Profiling: study typical function arguments
- Sandboxing: emulate a restricted view on a filesystem
- 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

Jürgen Schönwälder (Jacobs University Bremen) Operating Systems '2019 October 28, 2019 187 / 356

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

# Compile-time Interpositioning

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

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

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

## Link-time Interpositioning

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

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

## Load-time Interpositioning

- The dynamic linker can be used to pre-load shared libraries
- This may be controlled via setting the LD_PRELOAD environment variable
- Example:
  `LD_PRELOAD=./libmymalloc.so vim`

```c
/*
 * datehack/datehack.c --
 *
 * gcc -Wall -fPIC -DPIC -c datehack.c
 * ld -shared -o datehack.so datehack.o -ldl (Linux)
 * ld -dylib -o datehack.dylib datehack.o -ldl (MacOS)
 *
 * LD_PRELOAD=./datehack.so date (Linux)
 * DYLD_INSERT_LIBRARIES=./datehack.dylib date (MacOS)
 *
 * See fakeroot <http://freecode.com/projects/fakeroot> for a project
 * making use of LD_PRELOAD for good reasons.
 *
 * http://hackerboss.com/overriding-system-functions-for-fun-and-profit/
 */

#define _GNU_SOURCE
#include <time.h>
#include <dlfcn.h>

#include <stdlib.h>
#include <unistd.h>
#include <sys/types.h>

struct tm *(*orig_localtime)(const time_t *timep);

int (*orig_clock_gettime)(clockid_t clk_id, struct timespec *tp);

struct tm *localtime(const time_t *timep)
{
    time_t t = *timep - 60 * 60 * 24;
    return orig_localtime(&t);
}

int clock_gettime(clockid_t clk_id, struct timespec *tp)
{
    int rc = orig_clock_gettime(clk_id, tp);
    if (tp) {
        tp->tv_sec -= 60 * 60 * 24;
    }
    return rc;
}

void
_init(void)
{
    orig_localtime = dlsym(RTLD_NEXT, "localtime");
    if (! orig_localtime) {
        abort();
    }

    orig_clock_gettime = dlsym(RTLD_NEXT, "clock_gettime");
    if (! orig_clock_gettime) {
        abort();
    }
}
```

## Part VIII

# Memory Management

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

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

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

# Section 24: Memory Systems and Translation of Memory Addresses

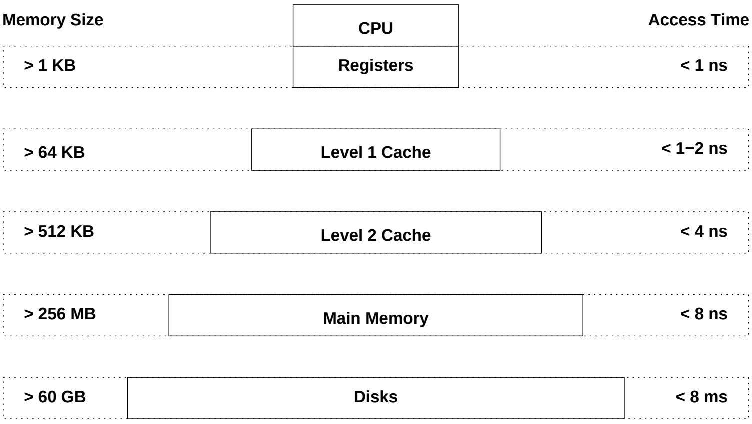
24 Memory Systems and Translation of Memory Addresses

25 Segmentation

26 Paging

27 Virtual Memory

## Memory Systems



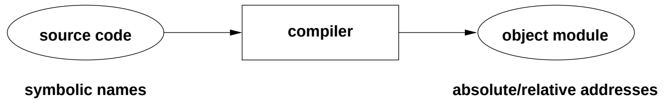
| Memory Size | CPU | Access Time |
|---|---|---|
| > 1 KB | Registers | < 1 ns |
| > 64 KB | Level 1 Cache | < 1–2 ns |
| > 512 KB | Level 2 Cache | < 4 ns |
| > 256 MB | Main Memory | < 8 ns |
| > 60 GB | Disks | < 8 ms |

- In the following, we will focus on the main memory

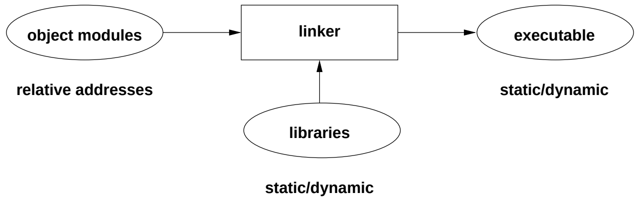
## Main Memory

- 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

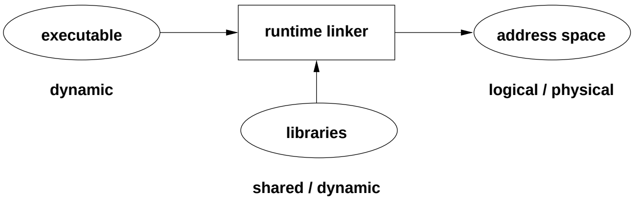


- Compiler translates symbolic addresses (variable / function names) into absolute or relative addresses

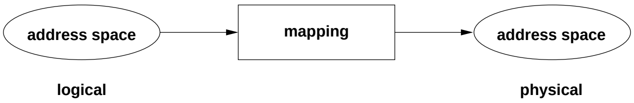


- Linker binds multiple object modules (with relative addresses) and referenced libraries into an executable

## Translation of Memory Addresses



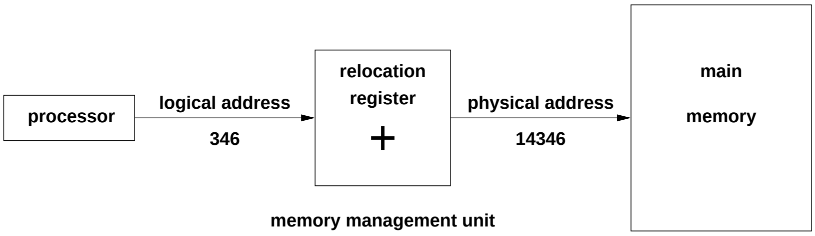
- Runtime linker binds executable with dynamic (shared) libraries at program startup time



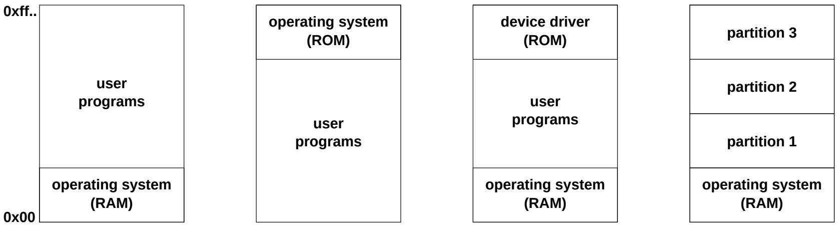
- Hardware memory management unit (MMU) maps the logical address space into the physical address space

# 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

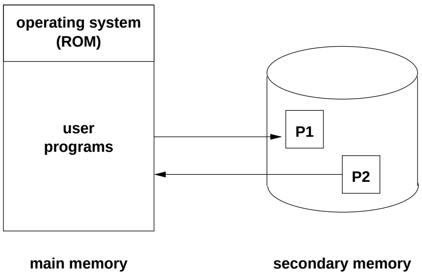


# Memory Partitioning



- Memory space is often divided into several regions or partitions, some of them serve special purposes
- Partitioning enables the OS to hold multiple processes in memory (as long as they fit)
- Static partitioning is not very flexible (but might 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

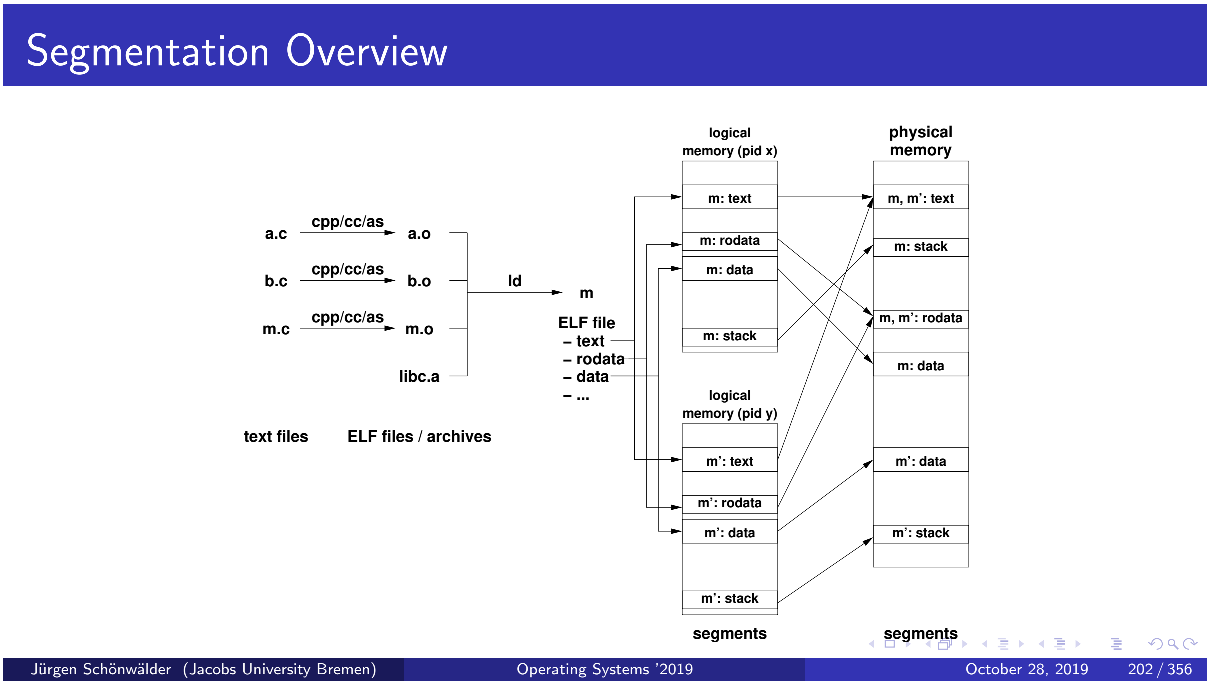
# Section 25: Segmentation

24 Memory Systems and Translation of Memory Addresses

25 Segmentation

26 Paging

27 Virtual Memory



## Segmentation Overview

Jürgen Schönwälder (Jacobs University Bremen) Operating Systems '2019 October 28, 2019 202 / 356

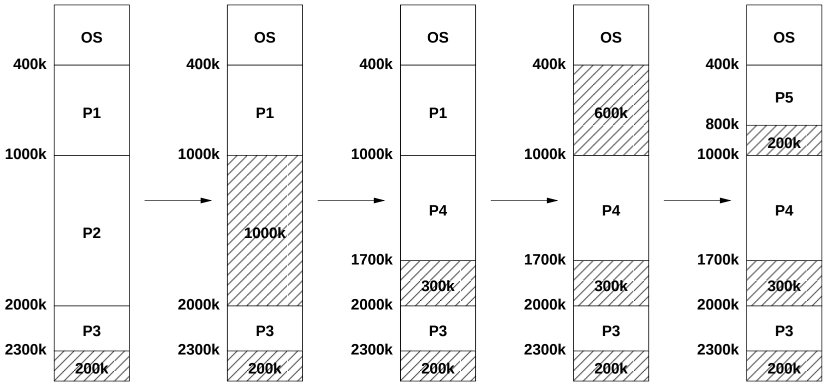
The slide shows a program `m` that was compiled from the source files `a.c`, `b.c`, and `m.c`. The linker (ld) links the object files `a.o`, `b.o`, and `m.o` and the C library `libc.a` into the executable file `m`. The exectable file, usually stored in ELF format, includes segments for the machine code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

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

## Segmentation

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

# External Fragmentation



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

# Positioning Strategies

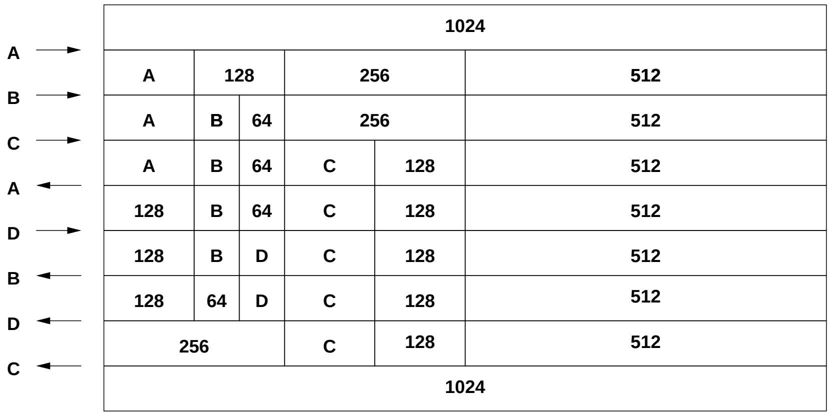
- *best fit*:
  - Allocate the smallest hole that is big enough
  - Large holes remain intact, many small holes
- *worst fit*:
  - Allocate the largest hole
  - Holes tend to become equal in size
- *first fit*:
  - Allocate the first hole from the top that is big enough
  - Simple and relatively efficient 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*:
  - 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
  - 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 $60k$:

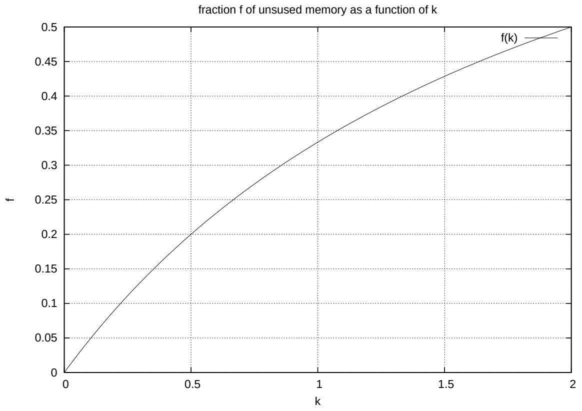


| | | | | | | |
|---|---|---|---|---|---|---|
| | 1024 | | | | | |
| A → | A | 128 | | 256 | | 512 |
| B → | A | B | 64 | 256 | | 512 |
| C → | A | B | 64 | C | 128 | 512 |
| A ← | 128 | B | 64 | C | 128 | 512 |
| D → | 128 | B | D | C | 128 | 512 |
| B ← | 128 | 64 | D | C | 128 | 512 |
| D ← | 256 | | | 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 $n/2$ holes occupy $m - ns$ bytes:
  $$(n/2) \cdot ks = m - ns \iff m = ns(1 + k/2)$$
  The fraction $f$ of memory occupied by holes is:
  $$f = \frac{nks/2}{m} = \frac{nks/2}{ns(1 + k/2)} = \frac{k}{k + 2}$$

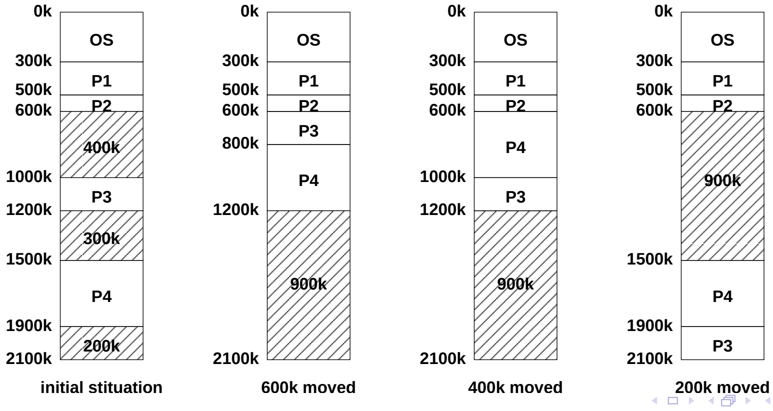
## Segmentation Analysis



$\Longrightarrow$ As long as the average hole size is a significant fraction of the average process 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



## Section 26: Paging

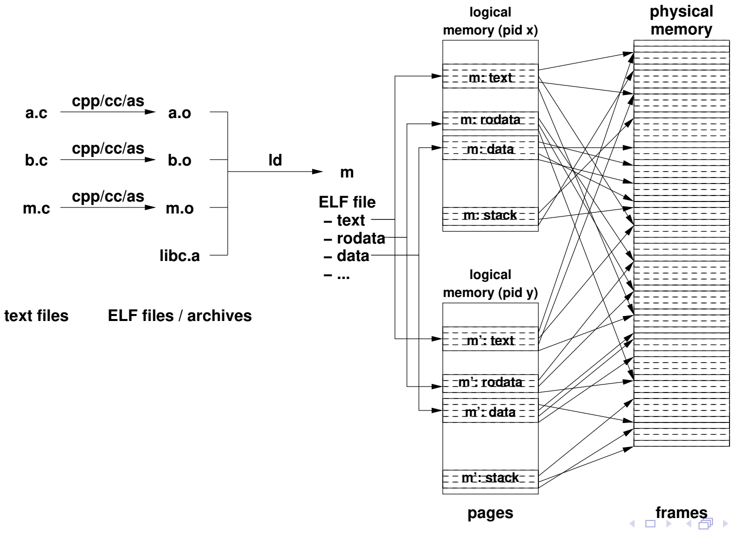
24 Memory Systems and Translation of Memory Addresses

25 Segmentation

26 Paging

27 Virtual Memory

## Paging Overview



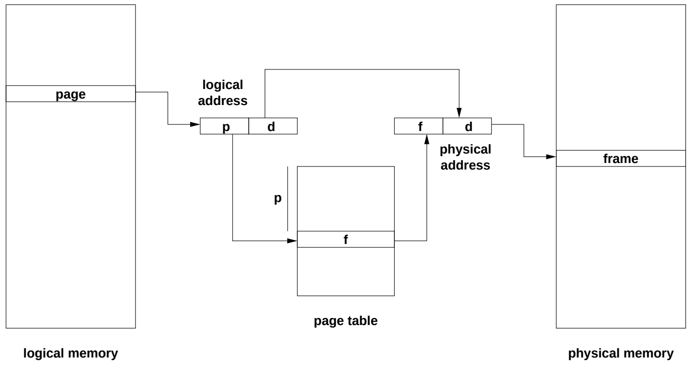
The slide shows a program `m` that was compiled from the source files `a.c`, `b.c`, and `m.c`. The linker (ld) links the object files `a.o`, `b.o`, and `m.o` and the C library `libc.a` into the executable file `m`. The exectable file, usually stored in ELF format, includes segments for the machine code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

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

## Paging Idea

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

# Paging Model and Hardware



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

## Paging Properties

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

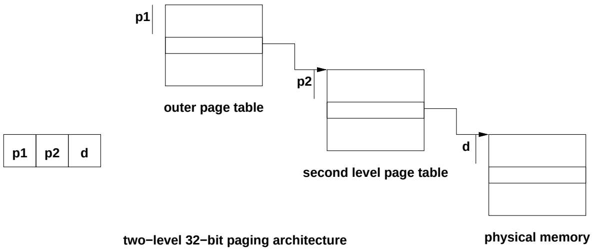
# Handling Page Faults

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

# Paging Characteristics

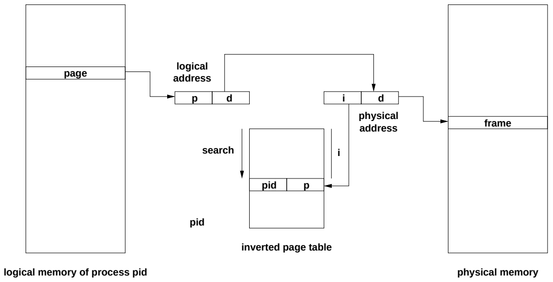
- Limited internal fragmentation (last page)
- Page faults are costly due to slow I/O operations
- Try to ensure that the “essential” pages of a process are always in memory
- Try to select used frames (victims) which will not be used in the future
- During page faults, other processes can execute
-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

# Multilevel Paging



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

# Inverted Page Tables



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

# 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 27: Virtual Memory

24 Memory Systems and Translation of Memory Addresses

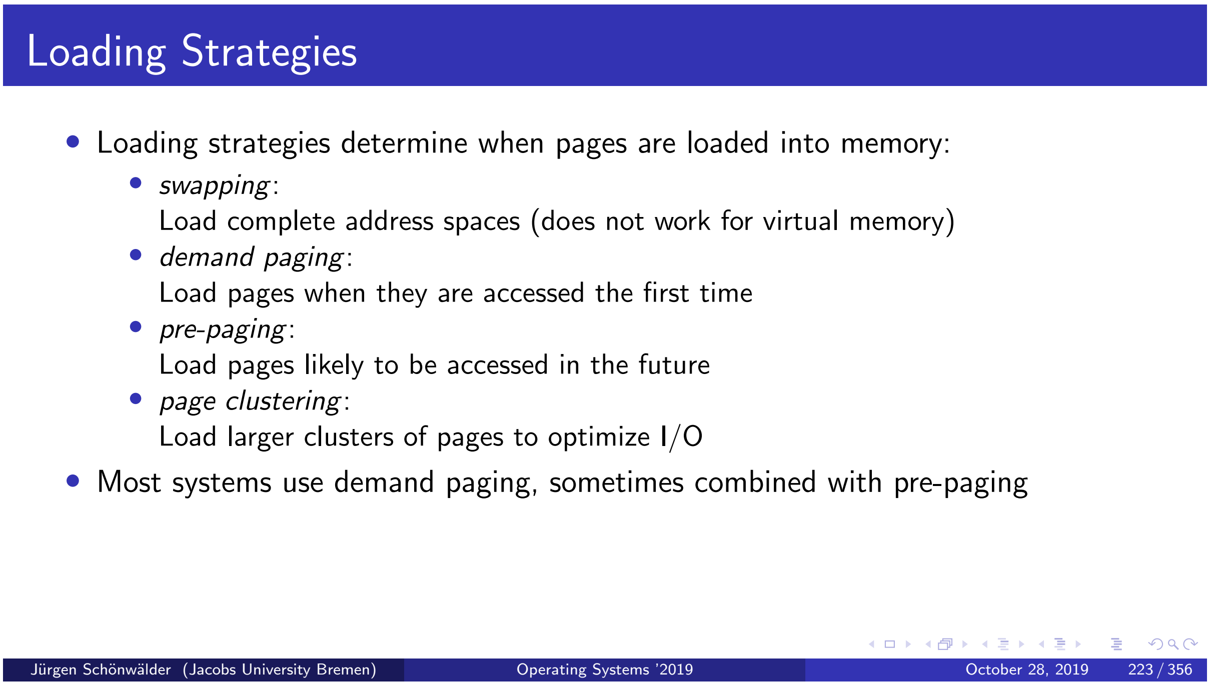
25 Segmentation

26 Paging

27 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, except that pages are skipped which 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 (in the past)
- *Belady's optimal algorithm (BO)*:
  Replace the page which will not be used for the longest period of time (in the future)

# Belady's Anomaly

- Increasing memory size should decrease page fault rate
- Consider $w = 123412512345$, FIFO replacement strategy and the memory sizes $m = 3$ and $m = 4$:

```
s[0]  = {}               s[0]  = {}
s[1]  = {1}     *        s[1]  = {1}       *
s[2]  = {1 2}   *        s[2]  = {1 2}     *
s[3]  = {1 2 3} *        s[3]  = {1 2 3}   *
s[4]  = {2 3 4} *        s[4]  = {1 2 3 4} *
s[5]  = {3 4 1} *        s[5]  = {1 2 3 4}
s[6]  = {4 1 2} *        s[6]  = {1 2 3 4}
s[7]  = {1 2 5} *        s[7]  = {2 3 4 5} *
s[8]  = {1 2 5}          s[8]  = {3 4 5 1} *
s[9]  = {1 2 5}          s[9]  = {4 5 1 2} *
s[10] = {2 5 3} *        s[10] = {5 1 2 3} *
s[11] = {5 3 4} *        s[11] = {1 2 3 4} *
s[12] = {5 3 4}          s[12] = {2 3 4 5} *
```

- 9 page faults for $m = 3$, 10 page faults for $m = 4$

# 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

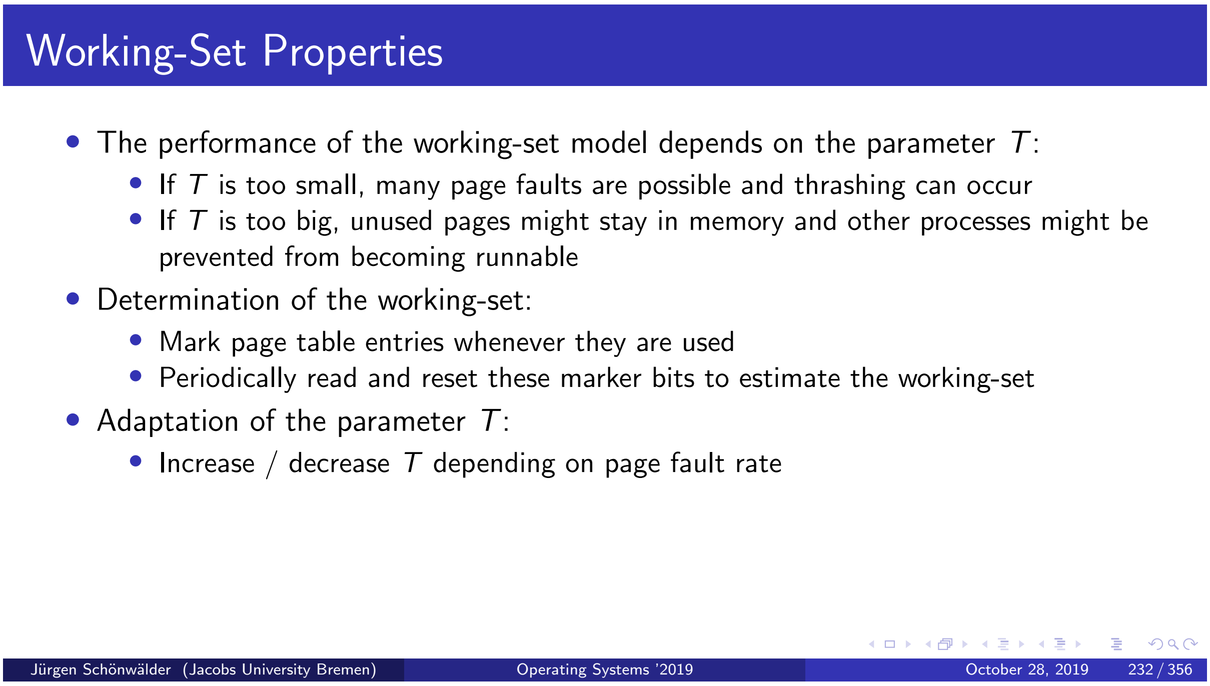
- Locality describes the property of programs to use only a small subset of the memory pages during a certain part of the computation
- Programs are typically composed of several localities, which may overlap
- Reasons for locality:
  - Structured and object-oriented programming (functions, small loops, 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 ($T = 10$):

$$w = \ldots \underline{2615777751}623412\underline{3444343444}13234443444 \ldots$$

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



## 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 IX

# Inter-Process Communication

# Inter-Process Communication

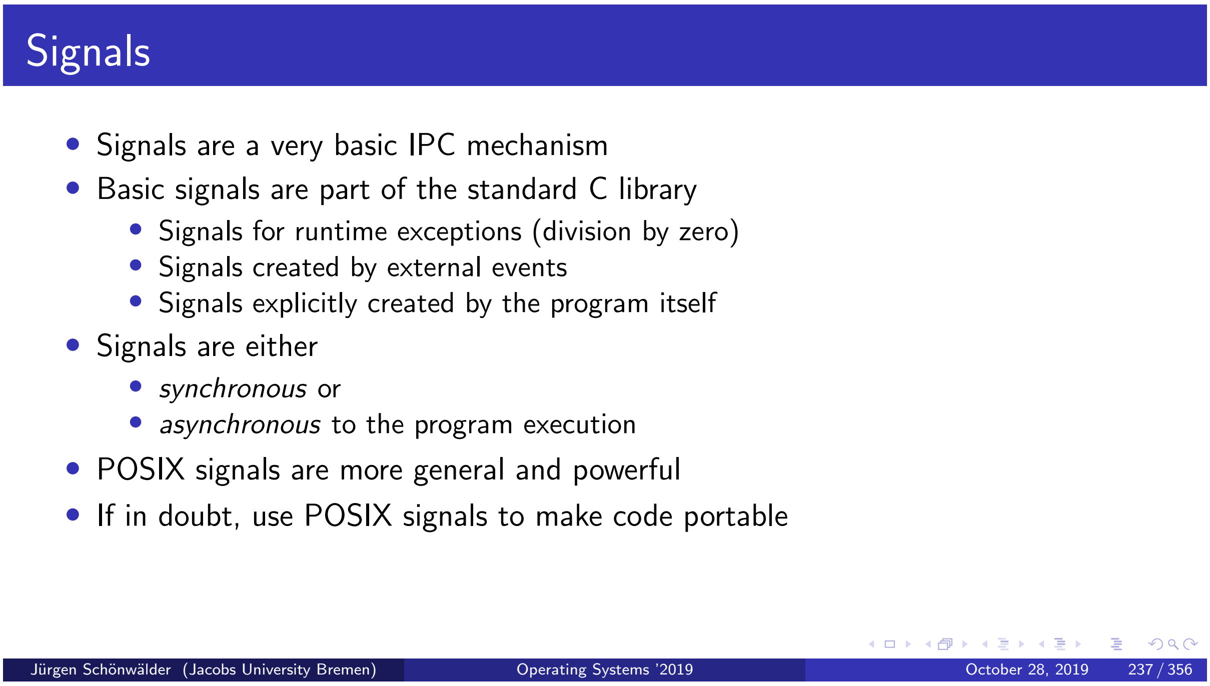
- An operating system has to provide inter-process communication primitives in the form of system calls and APIs
- Signals:
  - Software equivalent of hardware interrupts
  - Signals interrupt the normal control flow, but they do not carry any data (except the signal number)
- Pipes:
  - Uni-directional channel between two processes
  - One process writes, the other process reads data
- Sockets:
  - General purpose communication endpoints
  - Multiple processes, global (Internet) communication

# Section 28: Signals

28 Signals

29 Pipes

30 Sockets



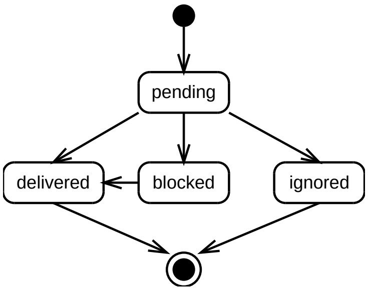
## Signals

- Signals are a very basic IPC mechanism
- 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
- Signals are either
  - *synchronous* or
  - *asynchronous* to the program execution
- POSIX signals are more general and powerful
- If in doubt, use POSIX signals 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 */
#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);
```

# Posix Signal API

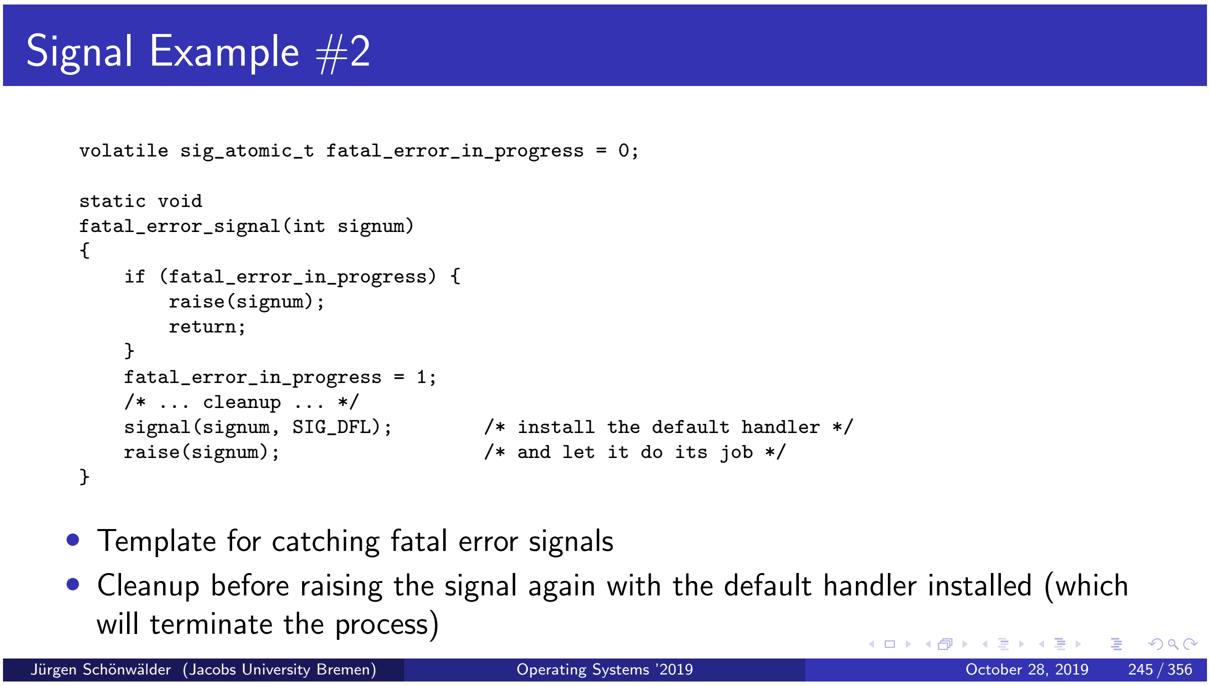
- The function `sigaction()` registers a function to be executed when a specific signal has been received
- During the execution of a signal function, the triggering signal and any signals specified in the signal mask are blocked
- The function `kill()` sends a signal to a process or process group:
  - If `pid > 0`, the signal is sent to process pid.
  - If `pid == 0`, the signal is sent to every process in the process group of the current process
  - If `pid == -1`, the signal is sent to every process except for process 1 (init)
  - If `pid < -1`, the signal is sent to every process in the process group `-pid`

# 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 signalled 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 must 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 Example #1

```
#include <signal.h>

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 Example #2

```
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 */
}
```

- Template for catching fatal error signals
- Cleanup before raising the signal again with the default handler installed (which will terminate the process)

Jürgen Schönwälder (Jacobs University Bremen) Operating Systems '2019 October 28, 2019 245 / 356

The following example demonstrates how the `sleep(3)` library function can be implemented using a timer signal.

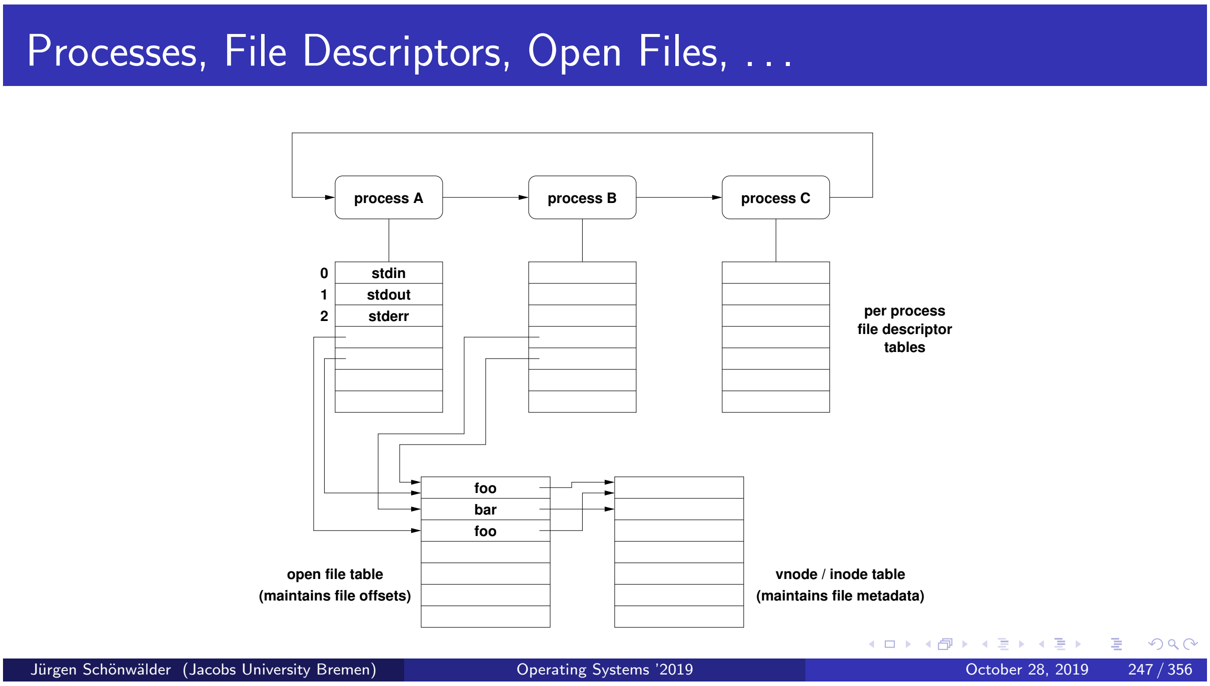
```
/*
 * sleep/sleep.c --
 *
 *        This little example demonstrates how to use the POSIX signal
 *        functions to wait reliably for a signal.
 */

#define _POSIX_C_SOURCE 2

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

static volatile sig_atomic_t wake_up = 0;

static void
catch_alarm(int sig)
{
    wake_up = 1;
}

unsigned int
sleep(unsigned int seconds)
{
    struct sigaction sa, old_sa;
    sigset_t mask, old_mask;

    sa.sa_handler = catch_alarm;
    sigemptyset(&sa.sa_mask);
    sa.sa_flags = 0;

    /*
     * Be nice and save the original signal handler so that it can be
     * restored when we are done.
     */
    sigaction(SIGALRM, &sa, &old_sa);

    /*
     * After resetting wake_up, ask the system to send us a SIGALRM at
     * an appropriate time.
     */

    wake_up = 0;
    alarm(seconds);

    /*
     * First block the signal SIGALRM. After safely checking wake_up,
     * suspend until a signal arrives. Note that sigsuspend may return
     * on other signals. If wake_up is finally true, cleanup by
     * unblocking the blocked signals.
     */

    sigemptyset(&mask);
    sigaddset(&mask, SIGALRM);
    sigprocmask(SIG_BLOCK, &mask, &old_mask);

    /*
     * No SIGALRM will be delievered here since this signal is
     * blocked. This means we have a safe region here until we
     * suspend below...
     */

    while (! wake_up) {
        /*
         * Wait for SIGALRM (assumed to be unblocked in old_mask).
         * While waiting, some other signals may get delivered since
         * the old mask is restored while waiting...
         */
        sigsuspend(&old_mask);
    }

    /*
     * Cleanup by restoring the original state.
     */

    sigprocmask(SIG_UNBLOCK, &mask, NULL);
    sigaction(SIGALRM, &old_sa, NULL);
    return 0;
}
```

# Section 29: Pipes

28 Signals

29 Pipes

30 Sockets

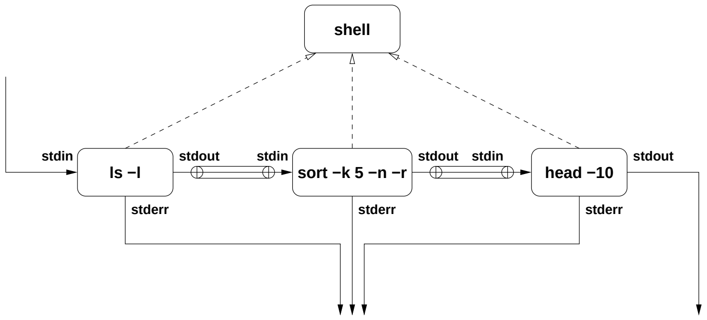


## Processes, File Descriptors, Open Files, . . .

Jürgen Schönwälder (Jacobs University Bremen) Operating Systems '2019 October 28, 2019 247 / 356

On a Unix / Posix system, user space programs refer to an I/O “channel” using small numbers, called file descriptors, that are essentially an index into the file descriptor table. Every process has its own file descriptor table and the file descriptor table of a process is copied when a child process is created. Hence, a child process inherits the file descriptors of the parent process when it is created. The entries in the file descriptor table then refer to other tables that maintain further information about an I/O “channel”. For regular files, the kernel maintains an open file table, which then further refers to tables that are file system specific.

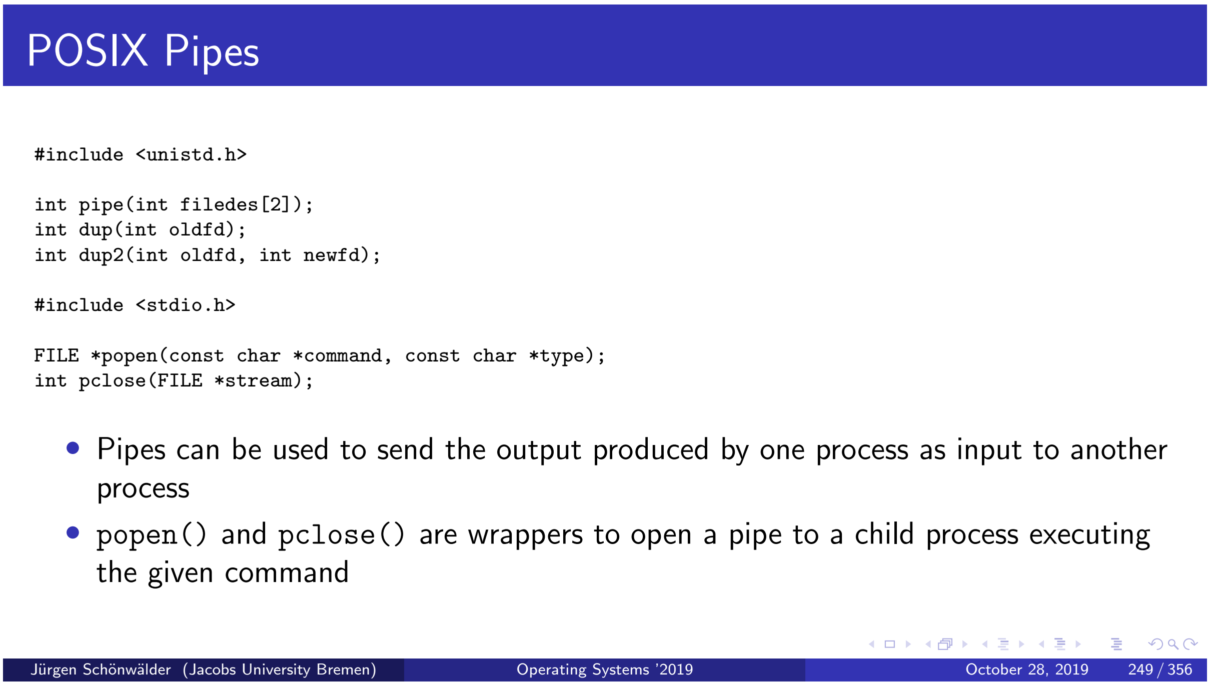
# Pipes at the Shell Command Line



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

Pipes are kernel objects that support unidirectional communication from the write end of a pipe to the read end of a pipe. When the kernel creates a new pipe, it allocates two new file descriptors for the two endpoints of the pipe in the file descriptor table.

In the example shown above, the shell parses the command line and it sees two pipe symbols. This tells the shell that is has to create two pipes and that it needs to fork three child processes, one excuting the `ls` command, one executing the `sort` command, and one executing the `head` command. After forking child processes and before doing the `exec()` calls, the shell has to arrange the file descriptors such that the standard output of the first child process goes into the write end of the first pipe, that the standard input of the second child process is the read end of the first pipe and the standard output of the second child process is the write end of the second pipe, and that the standard input of the third child process is the read end of the second pipe.



## POSIX Pipes

```
#include <unistd.h>

int pipe(int filedes[2]);
int dup(int oldfd);
int dup2(int oldfd, int newfd);

#include <stdio.h>

FILE *popen(const char *command, const char *type);
int pclose(FILE *stream);
```

- Pipes can be used to send the output produced by one process as input to another process
- `popen()` and `pclose()` are wrappers to open a pipe to a child process executing the given command

The `pipe()` system call allocates a pipe in the kernel and it returns the two file descriptors refering to the read end of the pipe (first file descriptor) and the write end of the pipe (second file descriptor). The `dup2()` system call duplicates the old file descriptor to be (in addition) the new file descriptor.

The `popen()` and `pclose()` library functions can be implemented as follows:

```
/*
 * popen.c --
 *
 *      Implementation of popen(3) and pclose(3) functions. This
 *      version keeps information about forked processes, but it is
 *      not thread safe.
 */

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

typedef struct pinfo {
    FILE         *file;
    pid_t         pid;
    struct pinfo *next;
} pinfo;

static pinfo *plist = NULL;

FILE*
popen(const char *command, const char *mode)
{
    int fd[2];
    pinfo *cur, *old;

    if (mode[0] != 'r' && mode[0] != 'w') {
```

```c
        errno = EINVAL;
        return NULL;
    }

    if (mode[1] != 0) {
        errno = EINVAL;
        return NULL;
    }

    if (pipe(fd)) {
        return NULL;
    }

    cur = (pinfo *) malloc(sizeof(pinfo));
    if (! cur) {
        close(fd[0]);
        close(fd[1]);
        errno = ENOMEM;
        return NULL;
    }

    cur->pid = fork();
    switch (cur->pid) {

    case -1:                    /* fork() failed */
        close(fd[0]);
        close(fd[1]);
        free(cur);
        return NULL;

    case 0:                     /* child */
        for (old = plist; old; old = old->next) {
            close(fileno(old->file));
        }

        if (mode[0] == 'r') {
            dup2(fd[1], STDOUT_FILENO);
        } else {
            dup2(fd[0], STDIN_FILENO);
        }
        close(fd[0]);   /* close other pipe fds */
        close(fd[1]);

        execl("/bin/sh", "sh", "-c", command, (char *) NULL);
        _exit(1);

    default:                    /* parent */
        if (mode[0] == 'r') {
            close(fd[1]);
            if (!(cur->file = fdopen(fd[0], mode))) {
                close(fd[0]);
            }
        } else {
            close(fd[0]);
            if (!(cur->file = fdopen(fd[1], mode))) {
                close(fd[1]);
            }
        }
        cur->next = plist;
        plist = cur;
    }

    return cur->file;
}

int
pclose(FILE *file)
{
    pinfo *last, *cur;
    int status;
    pid_t pid;

    /* search for an entry in the list of open pipes */

    for (last = NULL, cur = plist; cur; last = cur, cur = cur->next) {
        if (cur->file == file) break;
    }
    if (! cur) {
        errno = EINVAL;
        return -1;
    }

    /* remove entry from the list */

    if (last) {
        last->next = cur->next;
    } else {
        plist = cur->next;
    }

    /* close stream and wait for process termination */

    fclose(file);
    do {
        pid = waitpid(cur->pid, &status, 0);
    } while (pid == -1 && errno == EINTR);

    /* release the entry for the now closed pipe */

    free(cur);

    if (WIFEXITED(status)) {
        return WEXITSTATUS(status);
    }
    errno = ECHILD;
    return -1;
}
```

# Pipe Example: paging some text

```
static int
page(char *pager, char *text)
{
    ssize_t len, cnt;
    int status, pid, fd[2];

    status = pipe(fd);
    if (status == -1) {
        perror("pipe");
        return EXIT_FAILURE;
    }

    pid = fork();
    if (pid == -1) {
        perror("fork");
        return EXIT_FAILURE;
    }
```

# Pipe Example

```
if (pid == 0) {
    close(fd[1]);
    status = dup2(fd[0], STDIN_FILENO);
    if (status == -1) {
        perror("dup2");
        return EXIT_FAILURE;
    }
    close(fd[0]);
    execl(pager, pager, NULL);
    perror("execl");
    _exit(EXIT_FAILURE);
} else {
    close(fd[0]);
    status = dup2(fd[1], STDOUT_FILENO);
    if (status == -1) {
        perror("dup2");
        return EXIT_FAILURE;
    }
```

# Pipe Example

```
        close(fd[1]);
        for (len = strlen(text); len; len -= cnt, text += cnt) {
            cnt = write(STDOUT_FILENO, text, len);
            if (cnt == -1) {
                perror("write");
                return EXIT_FAILURE;
            }
        }
        close(1);
        do {
            if (waitpid(pid, &status, 0) == -1) {
                perror("waitpid");
                exit(EXIT_FAILURE);
            }
        } while (!WIFEXITED(status) && !WIFSIGNALED(status));
    }
    return EXIT_SUCCESS;
}
```

# Named Pipes

- Pipes can only exist between processes which have a common parent process who created the pipe
- Named pipes are file system objects and arbitrary processes can read from or write to a named pipe
- Named pipes are created using the `mkfifo()` function
- A simple example:

```
$ mkfifo pipe
$ ls > pipe &
$ less < pipe
```

- An interesting example:

```
$ mkfifo pipe1 pipe2
$ echo -n x | cat - pipe1 > pipe2 &
$ cat < pipe2 > pipe1
```

# Python Example

```
#!/usr/bin/env python

import os, sys

r, w = os.pipe()
pid = os.fork()
if pid:
    os.close(w)
    r = os.fdopen(r)   # turn r into a file object
    txt = r.read()
    os.waitpid(pid, 0) # make sure the child process gets cleaned up
else:
    os.close(r)
    w = os.fdopen(w, 'w')
    w.write("here's some text from the child")
    w.close()
    print "child: closing"
    sys.exit(0)

print "parent: got it; text =", txt
```

# Section 30: Sockets

28 Signals

29 Pipes

30 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

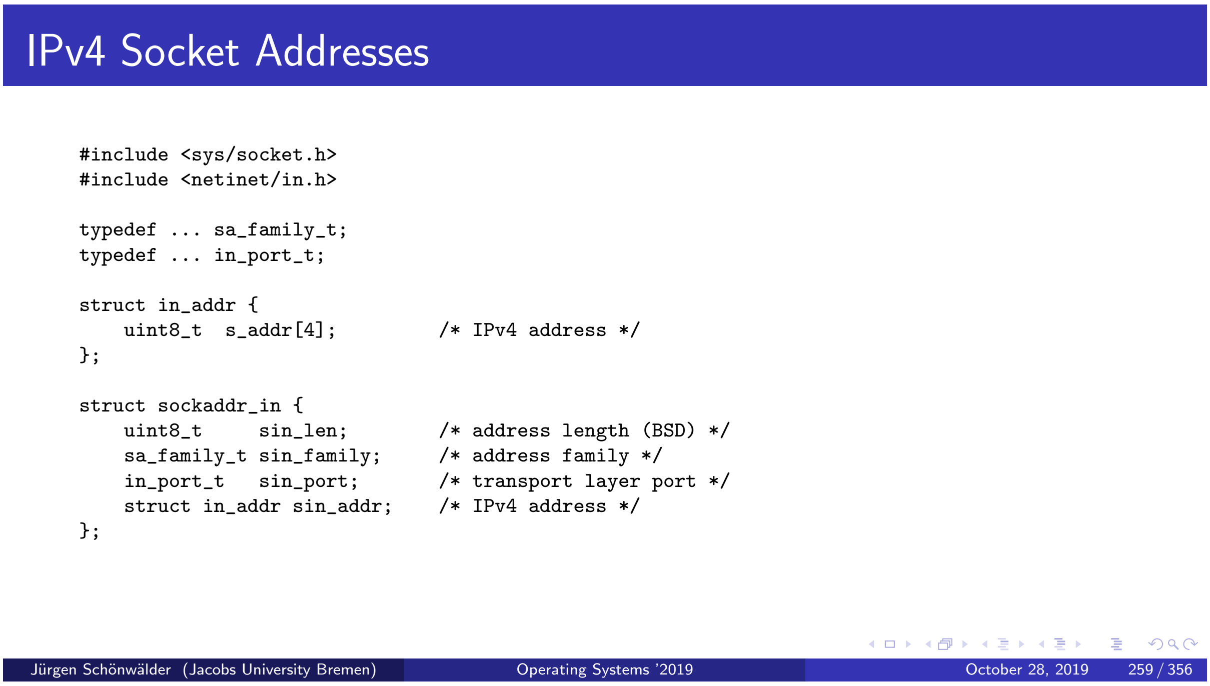
## Socket Types

- 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 */
};
```

- A `struct sockaddr` represents an abstract address, typically casted to a struct for a concrete address format
- A `struct sockaddr_storage` provides storage space



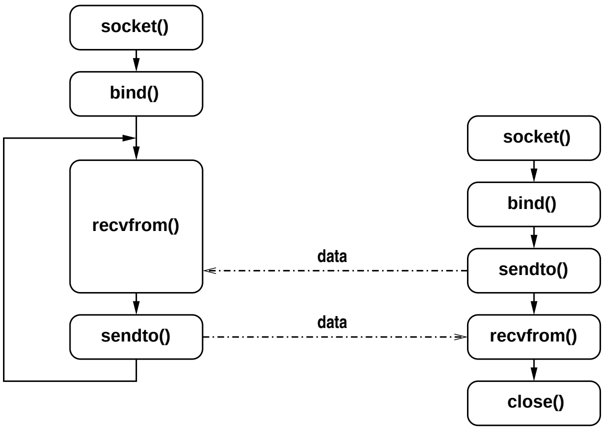
# IPv4 Socket Addresses

```
#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 */
    struct in_addr sin_addr;    /* IPv4 address */
};
```

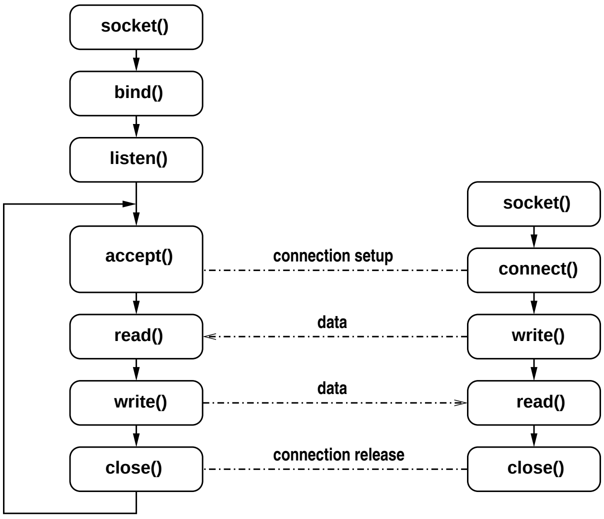
# IPv6 Socket Addresses

```
#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 */
    uint32_t    sin6_flowinfo;  /* flow information */
    struct in6_addr sin6_addr;  /* IPv6 address */
    uint32_t    sin6_scope_id;  /* scope identifier */
};
```

# Connection-Less Communication



## Connection-Oriented Communication



# Socket API Summary

```
#include <sys/types.h>
#include <sys/socket.h>
#include <unistd.h>

#define SOCK_STREAM    ...
#define SOCK_DGRAM     ...
#define SOCK_RAW       ...
#define SOCK_RDM       ...
#define SOCK_SEQPACKET ...

#define AF_LOCAL ...
#define AF_INET  ...
#define AF_INET6 ...

#define PF_LOCAL ...
#define PF_INET  ...
#define PF_INET6 ...
```

# Socket API Summary

```
int socket(int domain, int type, int protocol);
int bind(int socket, struct sockaddr *addr,
         socklen_t addrlen);
int connect(int socket, struct sockaddr *addr,
            socklen_t addrlen);
int listen(int socket, int backlog);
int accept(int socket, struct sockaddr *addr,
           socklen_t *addrlen);

ssize_t write(int socket, void *buf, size_t count);
int send(int socket, void *msg, size_t len, int flags);
int sendto(int socket, void *msg, size_t len, int flags,
           struct sockaddr *addr, socklen_t addrlen);

ssize_t read(int socket, void *buf, size_t count);
int recv(int socket, void *buf, size_t len, int flags);
int recvfrom(int socket, void *buf, size_t len, int flags,
             struct sockaddr *addr, socklen_t *addrlen);
```

# Socket API Summary

```
int shutdown(int socket, int how);
int close(int socket);

int getsockopt(int socket, int level, int optname,
               void *optval, socklen_t *optlen);
int setsockopt(int socket, int level, int optname,
               void *optval, socklen_t optlen);
int getsockname(int socket, struct sockaddr *addr,
                socklen_t *addrlen);
int getpeername(int socket, struct sockaddr *addr,
                socklen_t *addrlen);
```

- All API functions operate on abstract socket addresses
- Not all functions make equally sense for all socket types

# Mapping Names to Addresses

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

#define AI_PASSIVE     ...
#define AI_CANONNAME   ...
#define AI_NUMERICHOST ...

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;
};
```

# Mapping Names to Addresses

```
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);
```

- Many books still document the old name and address mapping functions
  - `gethostbyname()`
  - `gethostbyaddr()`
  - `getservbyname()`
  - `getservbyaddr()`

  which are IPv4 specific and should not be used anymore

# 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);
```

# 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);
```

- `select()` works with arbitrary file descriptors
- `select()` frequently used to implement the main loop of event-driven programs

# Part X

# File Systems

We are used to store our data in named files. We have files for text documents, for calculation sheets, for source code, for program code, for images, for music, for videos, and many other digital objects. In order to deal with a large amount of files, we can organize files that relate to each other into directories (or folders). Finding a good organization of files is often surprisingly difficult and usually the organization of files takes time to develop.

The operating system kernel provides us with the abstraction of a hierarchical file system where data objects can be named and easily be found by a human. The operating system kernel allows us to create new files, to change files, to rename files, to delete files, and to associate permissions with file system objects. We are so used to these operations that we often forget that the underlying storage components (e.g., hard-drives or flash-drives), only provide us with numbered data blocks of fixed size, something that is barely useful for humans to work with.

Since file systems are fundamental for the storage of data, it is crucial that file systems are robust (we do not want to loose data) and efficient.

# Section 31: General File System Concepts

31 General File System Concepts

32 File System Programming Interface

33 File System Implementation

## 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 are often used to identify the type of contents (`.txt`, `.c`, `.pdf`)
- Structured files:
  - Sequential files
  - Index-sequential files
  - B-tree files

⟹ Only some operating systems support structured files

## Special Files

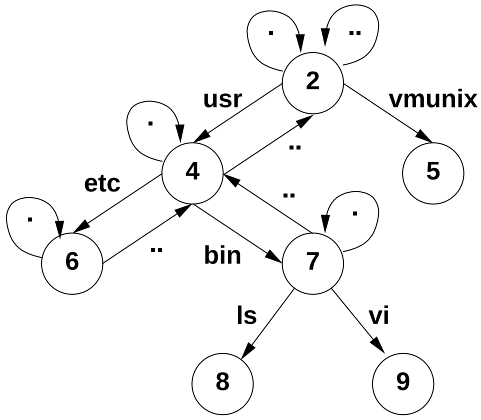
- Files representing devices:
  - Represent devices as files (`/dev/mouse`)
  - Distinction between block and character device files
  - Special operations to manipulate devices (`ioctl`)
- Files representing processes:
  - Represent processes (and more) as files (`/proc`)
  - Simple interface between kernel and system utilities
- Files representing communication endpoints:
  - Named pipes and fifos
  - Internet connection (`/net/tcp`) (Plan 9)
- Files representing graphical user interface windows:
  - Plan 9 represents all windows of a GUI as files

# Directories

- Hierarchical 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 and character sets)
- Absolute names formed through concatenation of directory and file names
- Directories may be realized
  - as special file system objects or
  - as regular files with special contents

⟹ Small and embedded operating systems sometimes only support flat file name spaces

# Unix Directory Structure

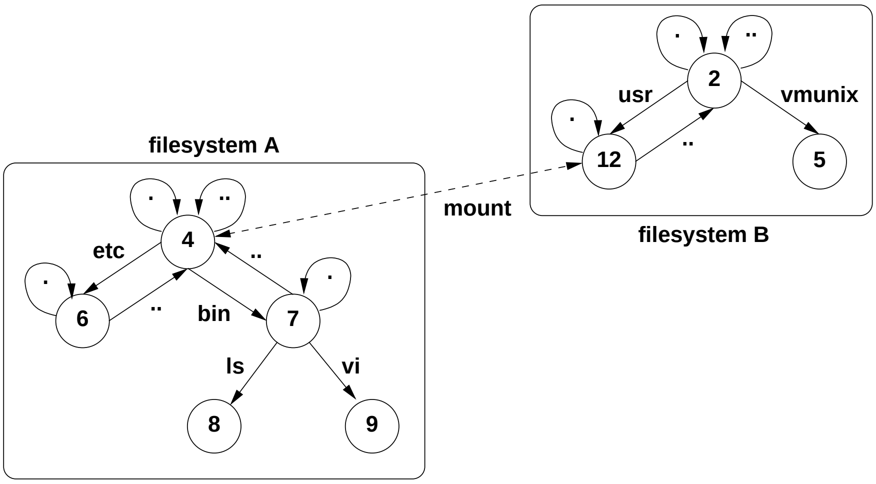


## Mounting

- Mounting is the process of importing a directory (usually residing on some other storage system) into the existing file system name space
- Enables logical name spaces that span multiple devices
- Mounted file systems can be different
- Mounted directories may reside on remote systems

⟹ More details on networked and distributed file systems are provided in a distributed systems course

# Mounting



filesystem A
.
..
4
etc
.
6
..
..
bin
.
7
ls
vi
8
9
mount
.
..
2
usr
vmunix
.
12
..
5
filesystem B

## Links

- Access a single file or directory under different names
- Two common types of links:
  - Hard links register a file under two different names
  - Soft links store the path (pointer) of the real file
- Links usually turn hierarchical name spaces into directed graphs. What about cycles in the graph?

# File Usage Pattern

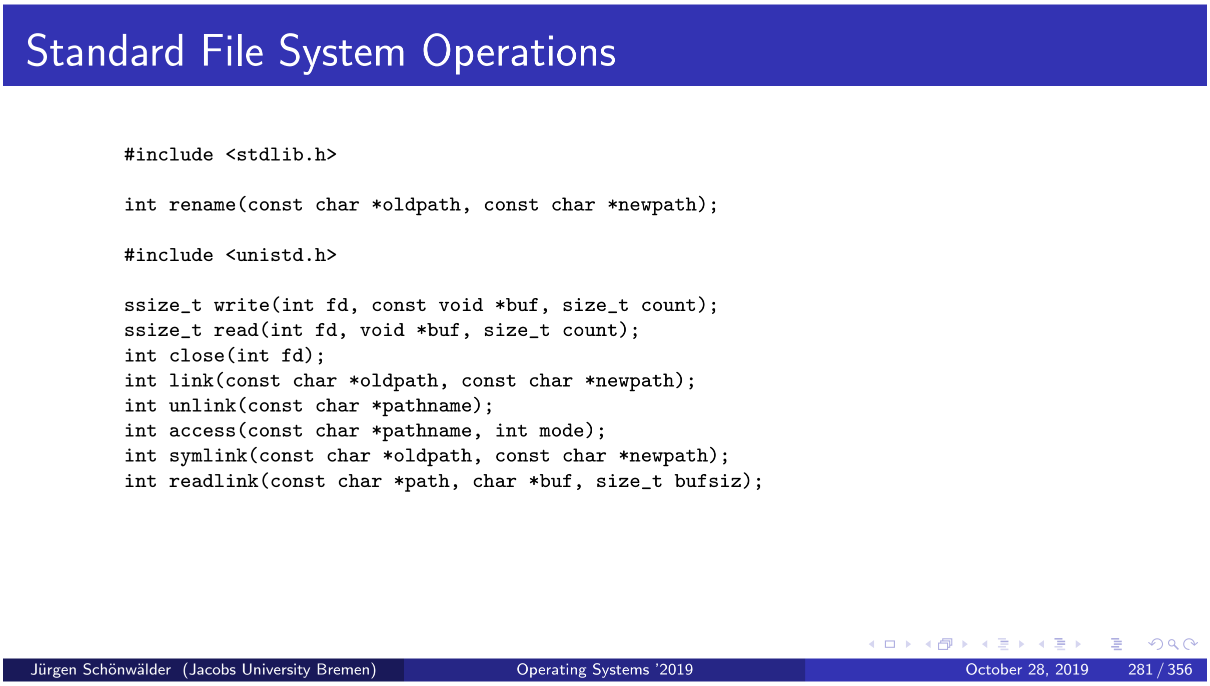
- File usage patterns heavily depend on the applications and the environment
- Typical file usage pattern of “normal” users:
  - Many small files (less than 10K)
  - Reading is more dominant than writing
  - Access is most of the time sequential and not random
  - Most files are short lived
  - Sharing of files is relatively rare
  - Processes usually use only a few files
  - Distinct file classes
- Totally different usage patterns for e.g. databases

# Section 32: File System Programming Interface

31 General File System Concepts

32 File System Programming Interface

33 File System Implementation



# Standard File System Operations

```
#include <stdlib.h>

int rename(const char *oldpath, const char *newpath);

#include <unistd.h>

ssize_t write(int fd, const void *buf, size_t count);
ssize_t read(int fd, void *buf, size_t count);
int close(int fd);
int link(const char *oldpath, const char *newpath);
int unlink(const char *pathname);
int access(const char *pathname, int mode);
int symlink(const char *oldpath, const char *newpath);
int readlink(const char *path, char *buf, size_t bufsiz);
```

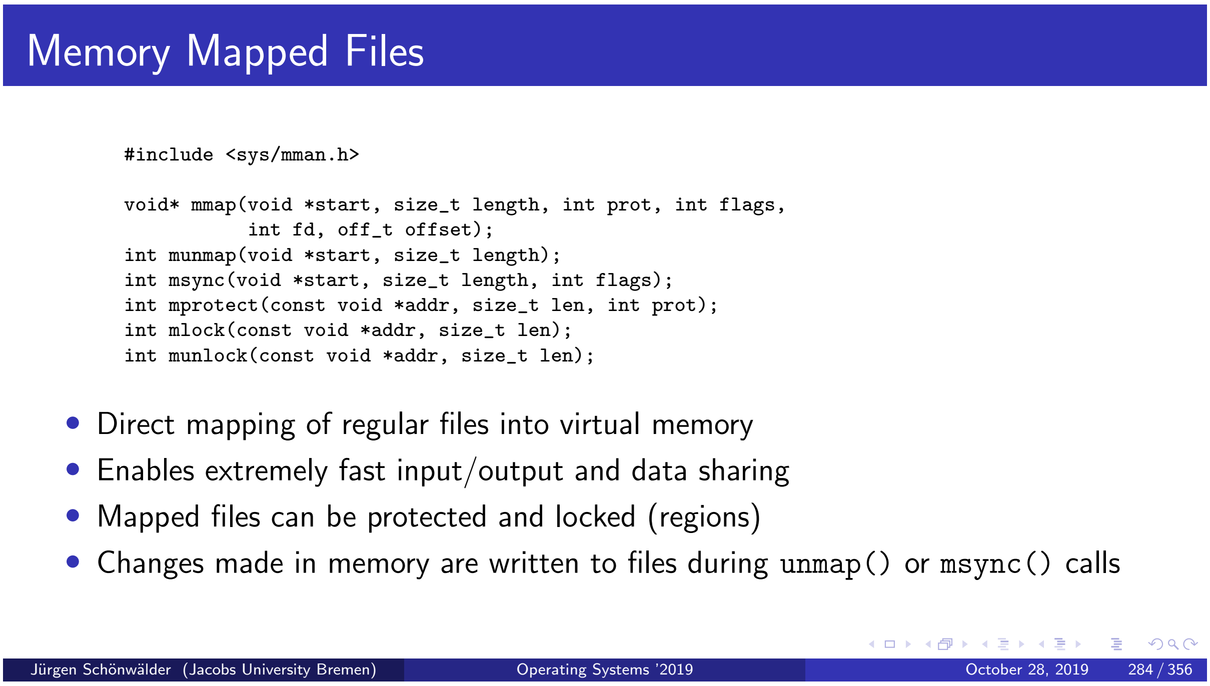
Most of the C functions are C or POSIX standards and quite portable across operating systems.

# Standard File System Operations

```
#include <sys/types.h>
#include <sys/stat.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 filedes, struct stat *buf);
int lstat(const char *file_name, struct stat *buf);

int chmod(const char *path, mode_t mode);
int fchmod(int fildes, mode_t mode);

#include <unistd.h>

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);
```



# Memory Mapped Files

```
#include <sys/mman.h>

void* mmap(void *start, size_t length, int prot, int flags,
           int fd, off_t offset);
int munmap(void *start, size_t length);
int msync(void *start, size_t length, int flags);
int mprotect(const void *addr, size_t len, int prot);
int mlock(const void *addr, size_t len);
int munlock(const void *addr, size_t len);
```

- Direct mapping of regular files into virtual memory
- Enables extremely fast input/output and data sharing
- Mapped files can be protected and locked (regions)
- Changes made in memory are written to files during `unmap()` or `msync()` calls

# File System Events

- Modern applications like to monitor file systems for changes.
- There are many system specific APIs, such as
  - `inotify` on Linux,
  - `kqueue` on *BSD,
  - `File System Events` on MacOS,
  - `ReadDirectoryChangesW` on Microsoft Windows.
- The APIs differ significantly in their functionality and whether they scale up to monitor large filesystem 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 33: File System Implementation

31 General File System Concepts

32 File System Programming Interface

33 File System Implementation

## Block Allocation Methods

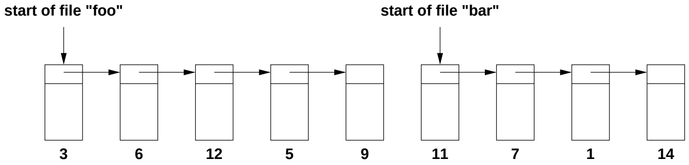
- *Contiguous allocation*:
  - Files stored as a contiguous block of data on the disk
  - Fast data transfers, simple to implement
  - File sizes often not known in advance
  - Fragmentation on disk
- *Linked list allocation*:
  - Every data block of a file contains a pointer (number) to the next data block
  - No fragmentation on disk
  - Reasonable sequential access, slow random access
  - Unnatural data block size (due to the space needed for the index)

## Block Allocation Methods

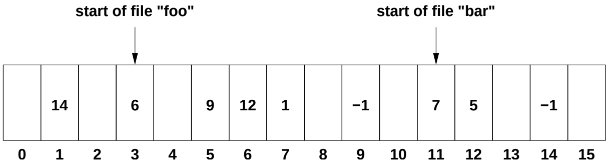
- *Linked list allocation using an index*:
  - The linked list is maintained in an index array outside of the data blocks
  - Index tables remain in main memory for fast access
  - Random access is reasonably faster
  - Significant memory usage by large index tables
  - Entire data blocks are available for data
- *Allocation using index nodes (inodes)*:
  - Small index nodes (inodes) store pointers to the first few disk blocks plus pointers to
    - an inode with data pointers (single indirect)
    - an inode with pointers to inodes (double indirect)
    - an inode with pointers to inodes with pointers to inodes (triple indirect)

# Block Allocation Methods

- Linked list allocation example:



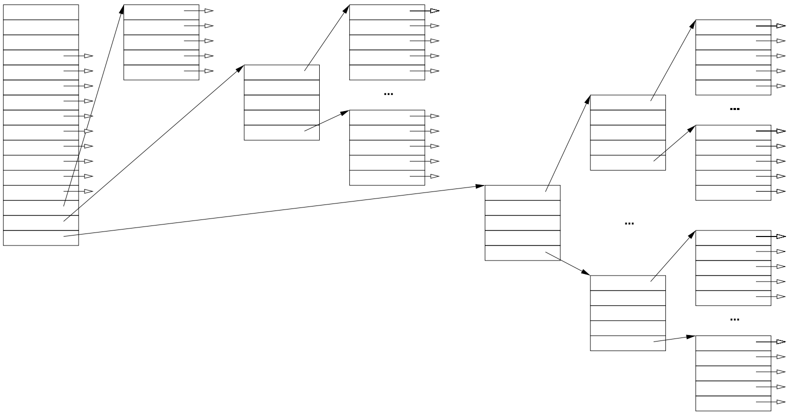
- Indexed linked list allocation example:



| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
|  | 14 |  | 6 |  | 9 | 12 | 1 |  | −1 |  | 7 | 5 |  | −1 |  |

# Block Allocation Methods

- Index node (inode) allocation example:



- Used on many Unix file systems (4.4 BSD and others)

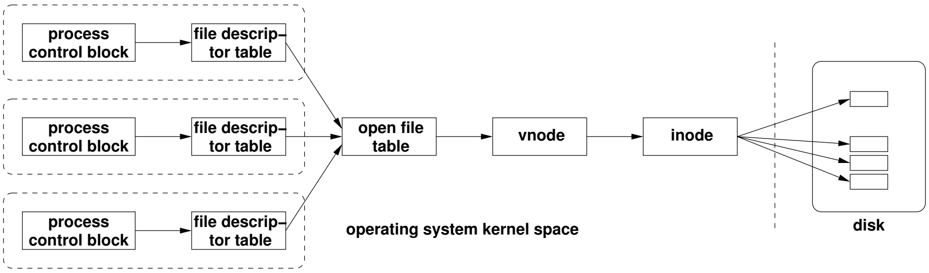
## Free-Space Management

- *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)

- Provide an abstract (virtual) file system interface
- Common functions (e.g., caching) can be implemented on the virtual file system interface
- Simplifies support for many different file systems
- A virtual file system interface is often realized as a collection of function pointers
- Example Linux (`<linux/fs.h>`)
    - `struct super_operations`
    - `struct inode_operations`
    - `struct file_operations`

# 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)

Part XI

# Input/Output and Devices

## Section 34: Goals and Design Considerations

34 Goals and Design Considerations

35 Storage Devices and RAIDs

36 Storage Virtualization

37 Terminal Devices

## 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 producer and data consumer 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 to 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

- Block devices represent devices where the natural unit of work is a fixed length data block (e.g., disks)
- Character devices represent devices where the natural unit of work is a character or a byte
- On Unix systems, devices are represented as special objects in the file system (usually mounted on `/dev`)
- Devices are identified by their type and their major and minor device number: the major number is used by the kernel to identify the responsible driver and the minor number to identify the device instance
- The `ioctl()` system call can be used by user-space applications to invoke device specific operations

# Section 35: Storage Devices and RAIDs

34 Goals and Design Considerations

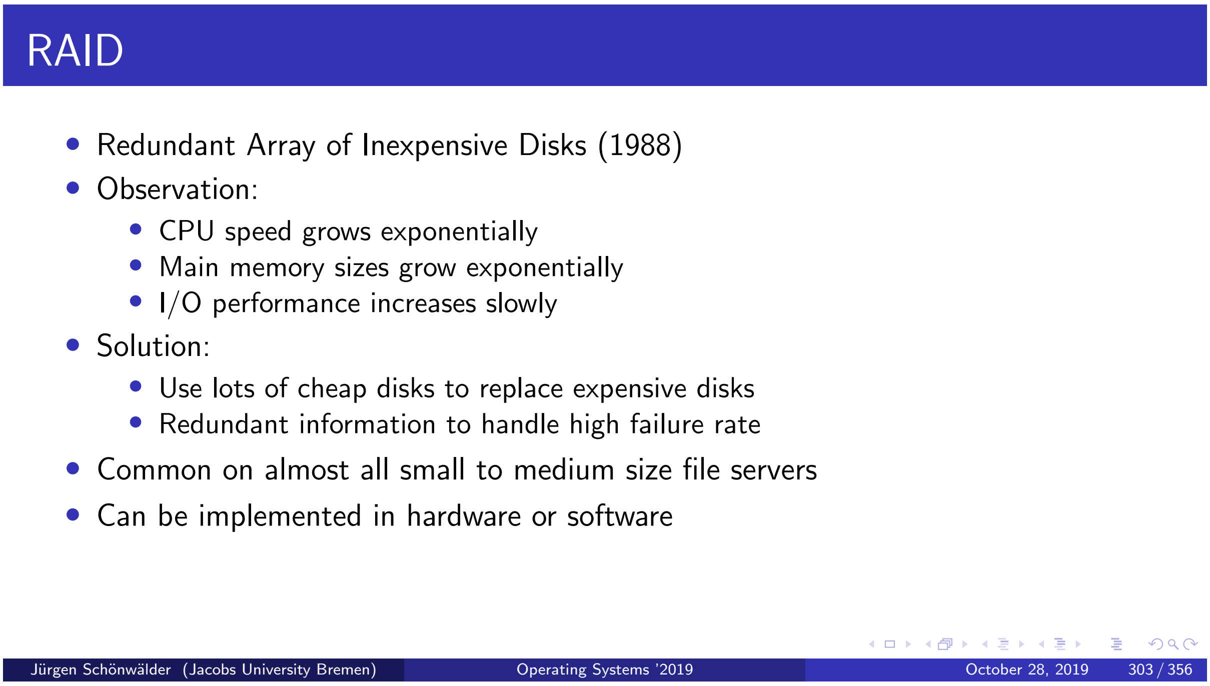
35 Storage Devices and RAIDs

36 Storage Virtualization

37 Terminal Devices

# 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 2 (Striping + ECC)

- Write data to data disks
- Write error correcting codes (ECC) to ECC disks
- Read and correct data on the fly
- High data transfer rates possible
- The higher the data transfer rate required, the better the ratio of data disks to ECC disks
- Relatively simple controller design
- High ratio of ECC disks to data disks
- Entry level cost very high

# RAID Level 3 (Striping + Parity)

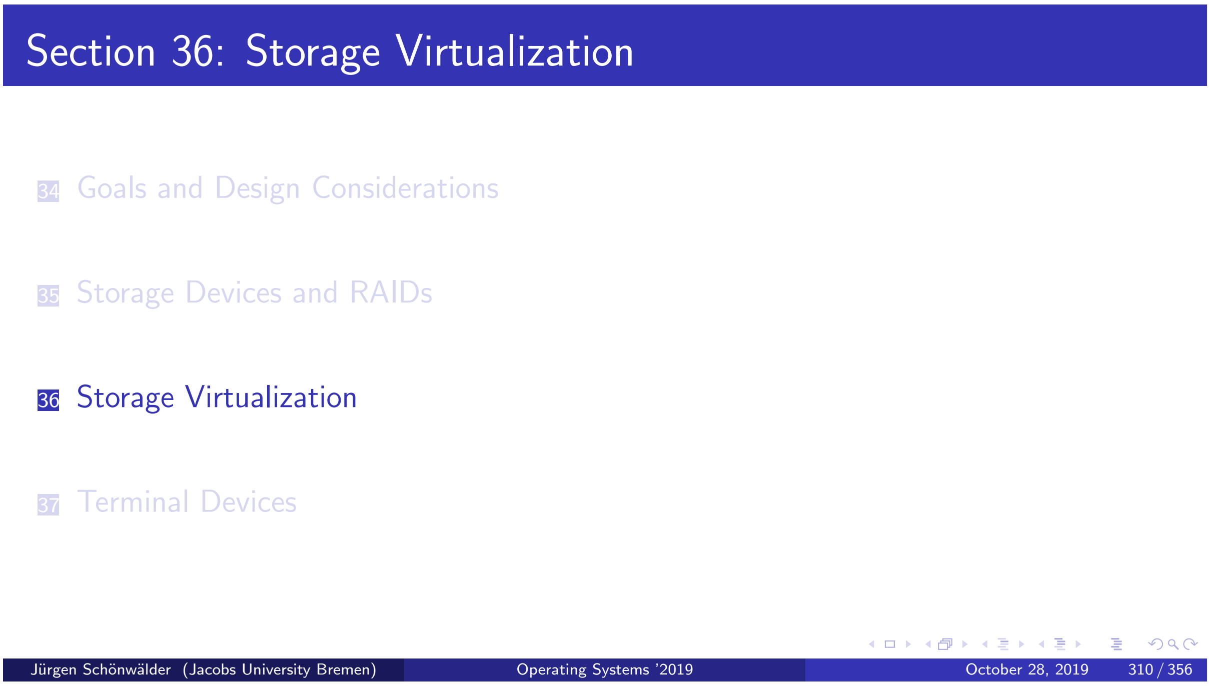
- The data block is subdivided (”striped”) and written on the data disks
- Stripe parity is generated on writes, recorded on the parity disk and checked on reads
- High read and write data transfer rate
- Low ratio of ECC (parity) disks to data disks
- Transaction rate equal to that of a single disk drive at best
- Controller design is fairly complex

## RAID Level 4 (Parity)

- Data blocks are written onto data disks
- Parity for disk blocks is generated on writes and recorded on the shared parity disk
- Parity is checked on reads
- High read data transaction rate
- Low ratio of ECC (parity) disks to data disks
- 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 fairly complex

# RAID Level 5 (Distributed Parity)

- Data blocks are written onto data disks
- Parity for blocks is generated on writes 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 36: Storage Virtualization

34 Goals and Design Considerations

35 Storage Devices and RAIDs

36 Storage Virtualization

37 Terminal Devices

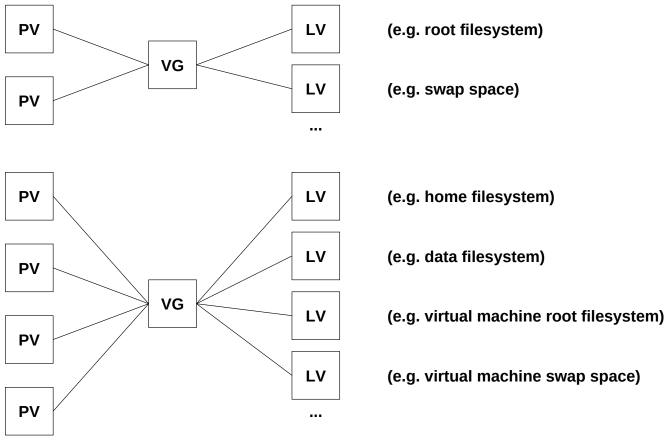
# Logical Volume Management

- *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

⟹ Separation of the logical storage layout from the physical storage layout

⟹ Simplifies modification of logical volumes (create, remove, resize, snapshot)

# Logical Volume Management (Linux)



(e.g. root filesystem)

(e.g. swap space)

(e.g. home filesystem)

(e.g. data filesystem)

(e.g. virtual machine root filesystem)

(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 37: Terminal Devices

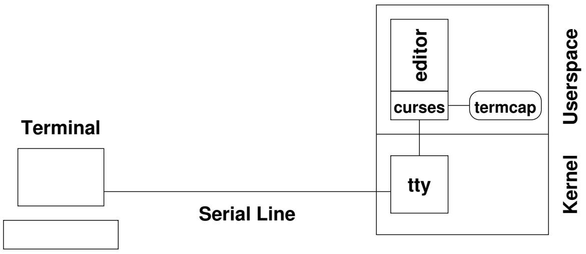
34 Goals and Design Considerations

35 Storage Devices and RAIDs

36 Storage Virtualization

37 Terminal Devices

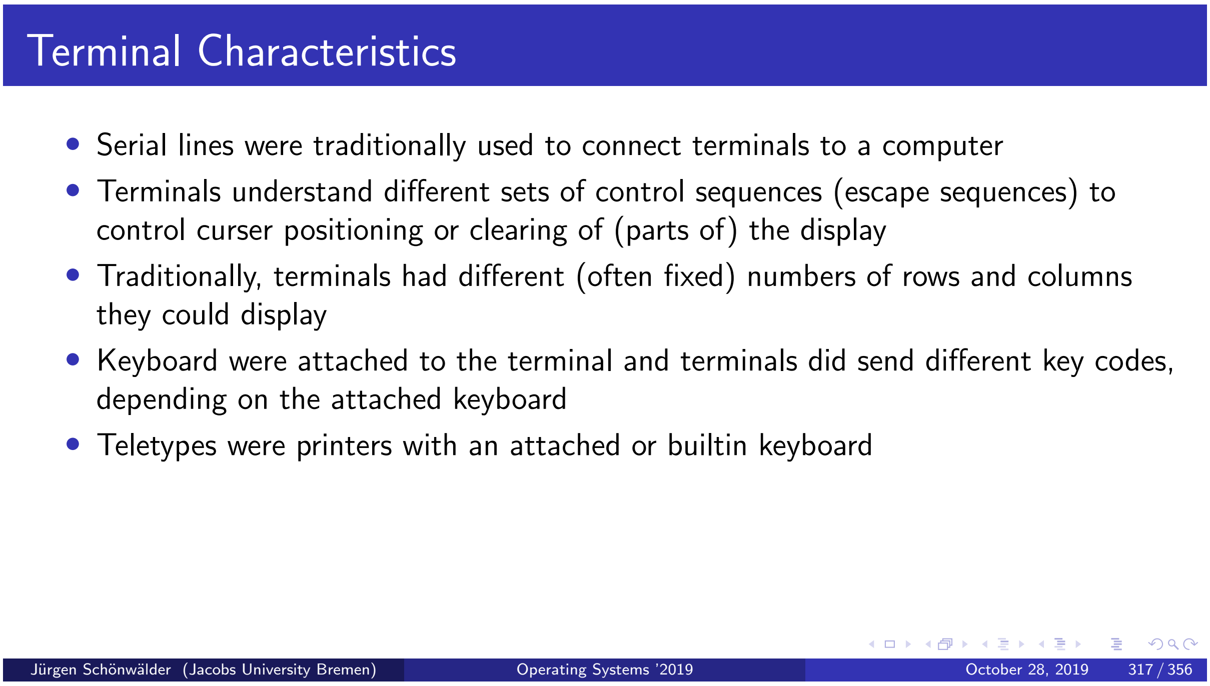
## Traditional Character Terminal Devices



- 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

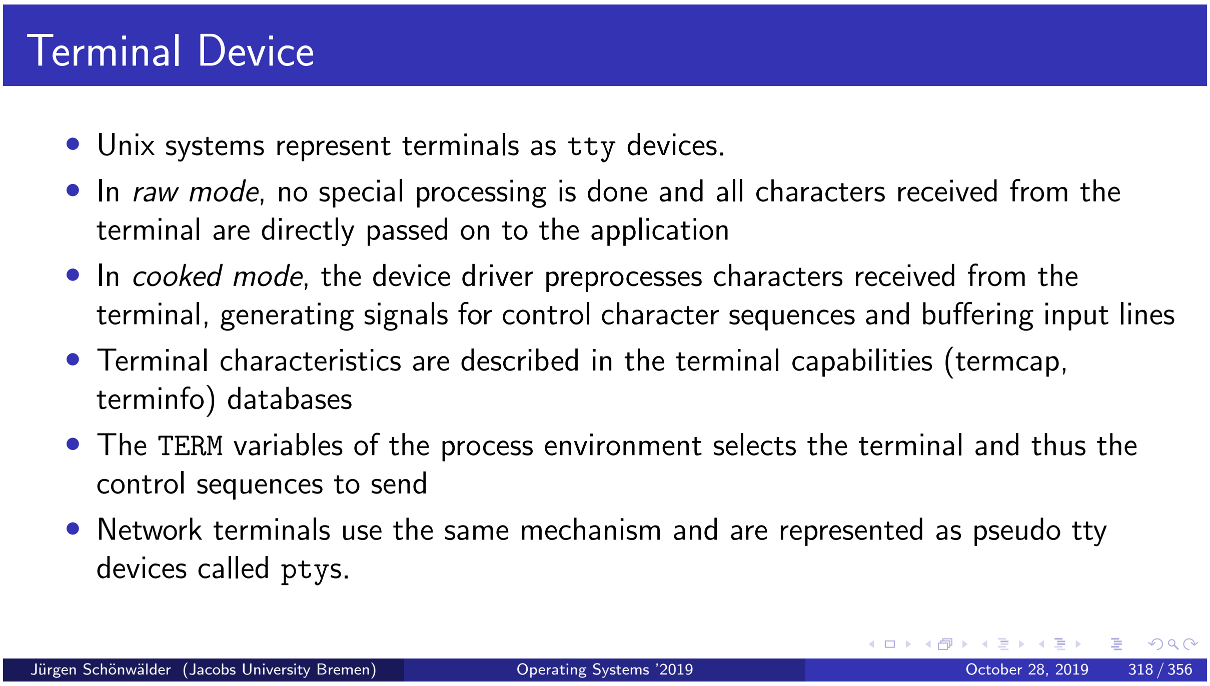
## Serial Communication (RS232)

- 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
- Keyboard 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 Device

- Unix systems represent terminals as `tty` devices.
- In *raw mode*, no special processing is done and all characters received from the terminal are directly passed on to the application
- In *cooked mode*, the device driver preprocesses characters received from the terminal, generating signals for control character sequences and buffering input lines
- Terminal characteristics are described in the terminal capabilities (termcap, terminfo) databases
- The `TERM` variables of the process environment selects the terminal and thus the control sequences to send
- Network terminals use the same mechanism and are represented as pseudo tty devices called `ptys`.

On many Linux systems, terminfo is installed as part of the base distribution. The `TERM` variable indicates which terminal is in use. To obtain the terminal characteristics from the terminfo files, one can use the following shell command:

```
infocmp -L $TERM | less
```

Programs like `vim` or `top` are linked against the `tinfo` library providing access to the information stored in the terminfo files.

Communication over the network sometimes requires to represent a network connection as a terminal (e.g., `ssh`). To support this, kernels provide so called pseudo ttys, that behave a bit like bidrectional pipes but emulate terminal device behavior. A pseudo tty is a pair of a slave and a master tty. The slave emulates a hardware text terminal device while the master provides the interface to control the terminal. In a remote login scenario (`ssh`), the shell on the remote system interacts with a slave tty while the daemon implementing the SSH network protocol interacts with the master tty. Pseudo ttys have many other uses, e.g., to implement software terminals on a graphical user interface or to automate programs that expect to run on a terminal.

## Portable and Efficient Terminal Control

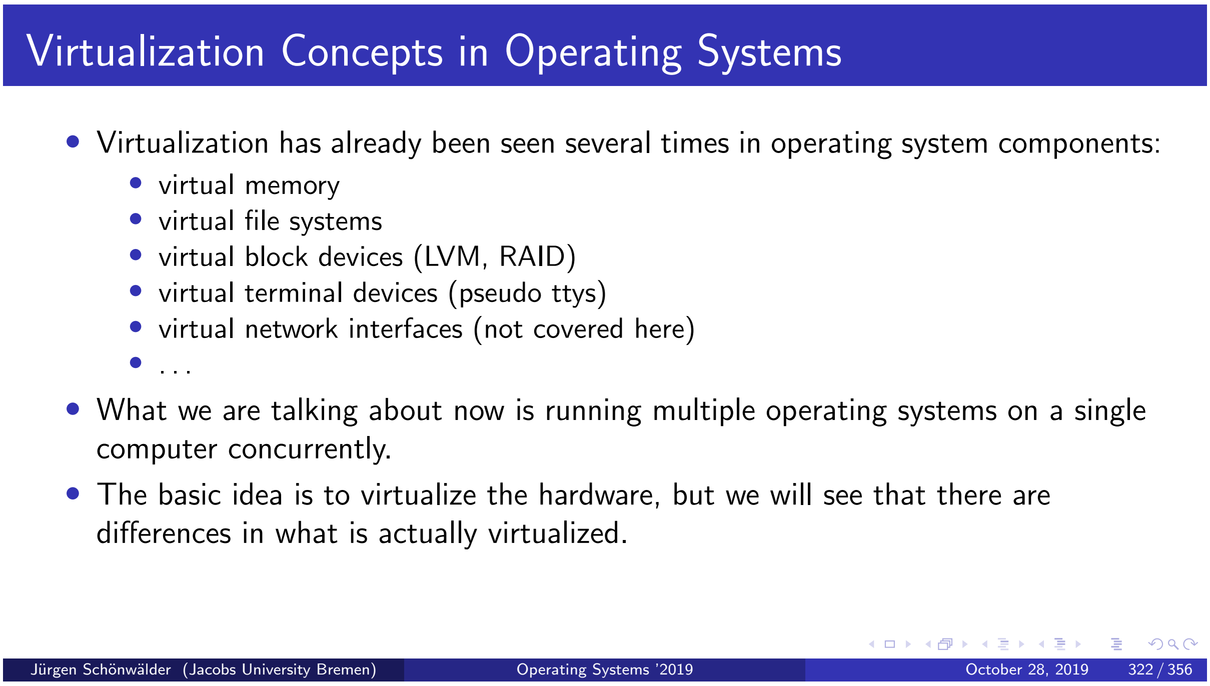
- Curses is a terminal control library enabling the construction of text user interface applications
- The curses API provides functions to position the cursor and to write at specific positions in a virtual window
- The refreshing of the virtual window to the terminal is program controlled
- Based on the terminal capabilities, the curses library can find the most efficient sequence of control codes to achieve the desired result
- The curses library also provides functions to switch between raw and cooked input mode and to control function key mappings
- The `ncurses` implementation provides a library to create panels, menus, and input forms.

# Part XII

# Virtual Machines

# Section 38: Terminology

38 Terminology



## 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 ⇒ Intel)
  - Faster development and testing of embedded software
  - Development and testing of code for different target architectures
  - Usage of software that cannot be ported to new platforms
- Examples:
  - QEMU `http://www.qemu.org/`
    - full system emulation and user mode (process) emulation
    - support for many different processor architectures
    - dynamic translation to native code

# Hardware Virtualization

- Virtualization of the physical hardware (aka hardware virtualization)
  - 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:

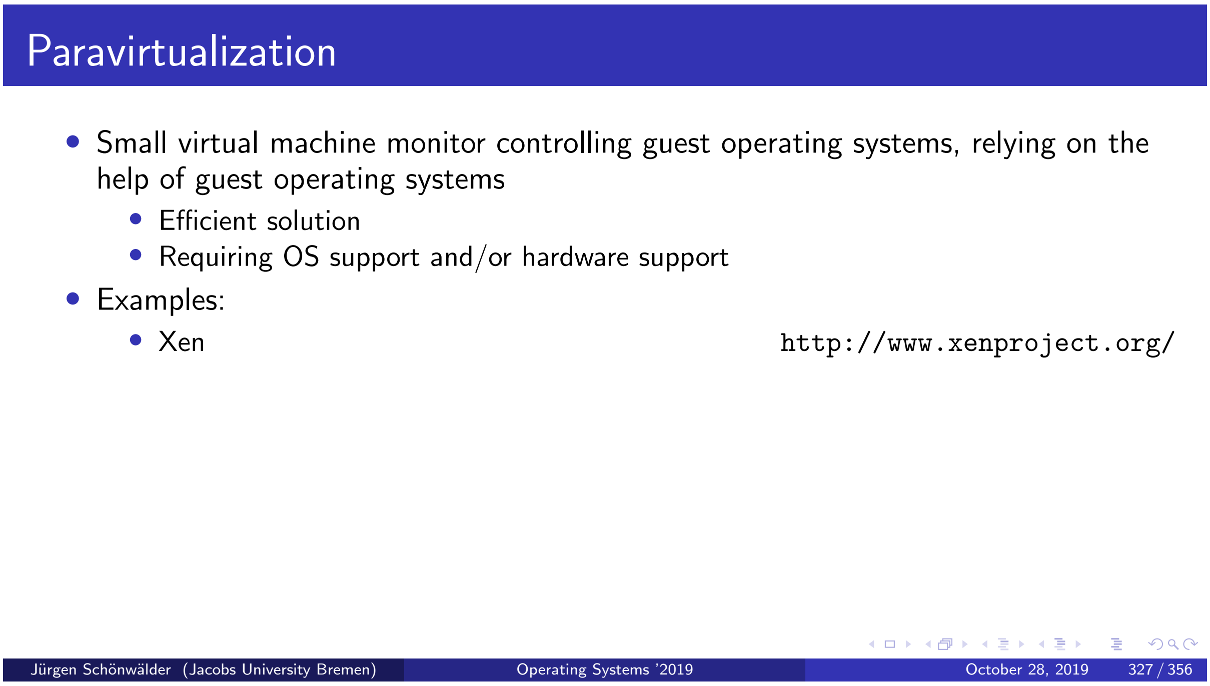
| | |
|---|---|
| VMware | `http://www.vmware.com/` |
| VirtualBox | `https://www.virtualbox.org/` |
| Parallels | `http://www.parallels.com/` |
| Linux KVM | `http://www.linux-kvm.org/` |
| . . . | |

## User-Level Virtualization

- Virtualization of kernels in user space
  - Simplify kernel development and debugging
- Examples:
  - User-mode Linux `http://user-mode-linux.sourceforge.net/`

# OS-Level Virtualization

- Multiple virtual operating system interfaces provided by a single operating system
  - Efficient separation using different namespaces
  - Robustness with minimal loss of performance
  - Reduction of system administration complexity
- Examples:
  - Linux Container
  - Linux VServer
  - BSD Jails
  - Solaris Zones



## Paravirtualization

- Small virtual machine monitor controlling guest operating systems, relying on the help of guest operating systems
  - Efficient solution
  - Requiring OS support and/or hardware support
- Examples:
  - Xen http://www.xenproject.org/

Paravirtualization tries to find a middle-ground between hardware virtualization and OS-level virtualization. The Xen system [1] is a well documented paravirtualization system.

Part XIII

# Distributed Systems

# Section 39: Definition and Models

39 Definition and Models

40 Remote Procedure Calls

41 Distributed File Systems

42 Distributed Message Queues

## 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

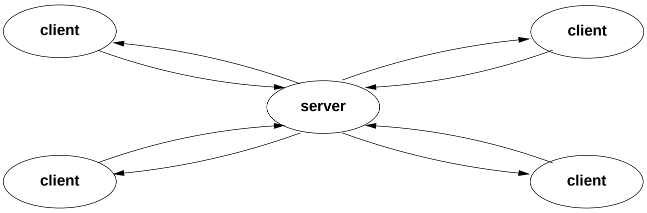
Special design challenges (increasingly important):

- Context-awareness and energy-awareness

# Distributed vs. Centralized

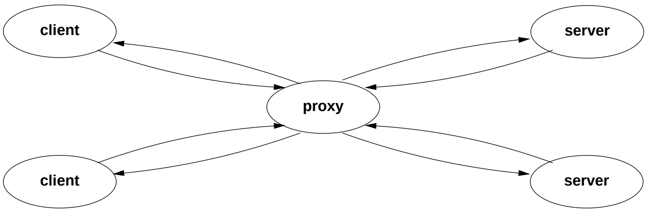
- *Lack of knowledge of global state*
  Nodes in a distributed system have access only to their own state and not to the global state of the entire system
- *Lack of a global time frame*
  The events constituting the execution of a centralized algorithm are totally ordered by their temporal occurance. 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



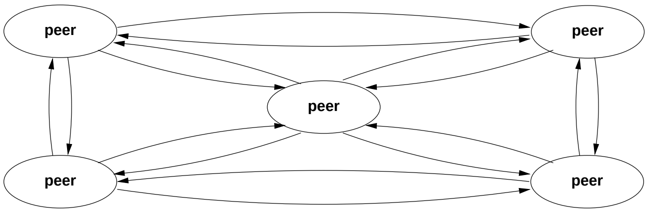
- Clients requests services from servers
- Synchronous: clients wait for the response before they proceed with their computation
- Asynchronous: clients proceed with computations while the response is returned by the server

# Proxies

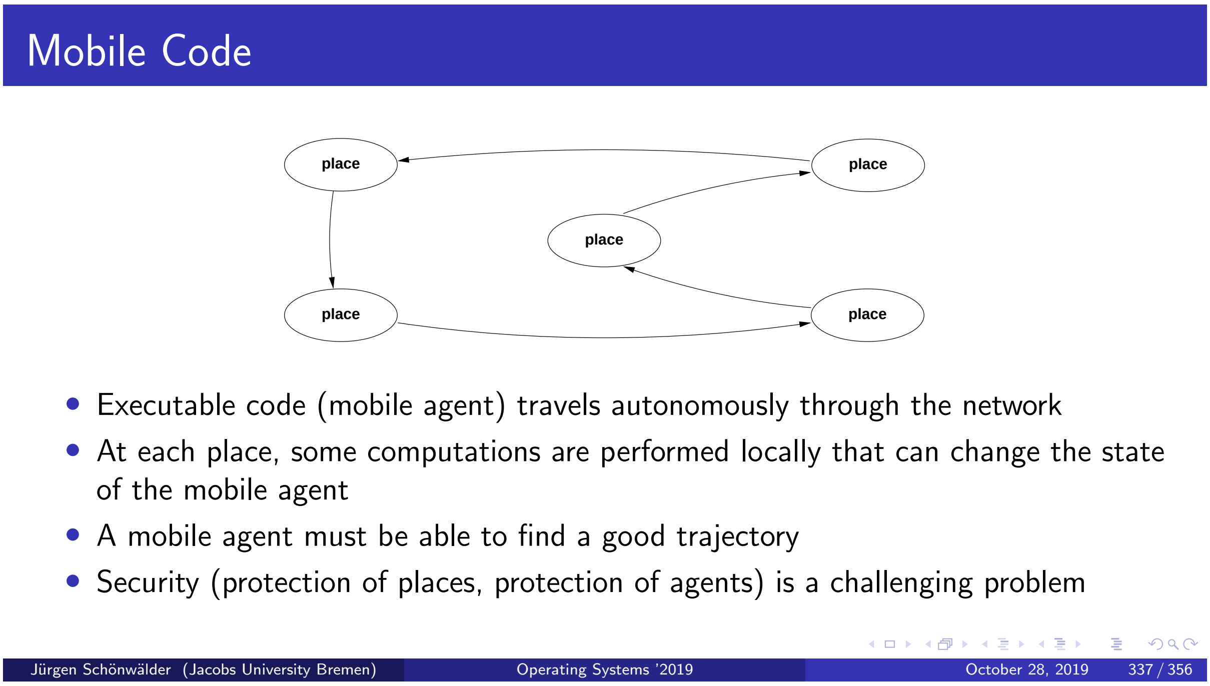


- Proxies can increase scalability
- Proxies can increase availability
- Proxies can increase protection and security
- Proxies and help solving versioning issues

## 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

Andrzej Bieszczad proposed a taxonomy for mobile code [2], which was a hot topic of research at the end of the 1990s:

- *Applets*: downloadable applications
- *Servlets*: uploadable services
- *Extlets*: uploadable or downloadable features
- *Deglets*: delegated tasks
- *Netlets*: autonomous tasks
- *Piglets*: malicious mobile code

Nowadays, we have a lot of mobile code on web pages, usually in the form of JavaScript.

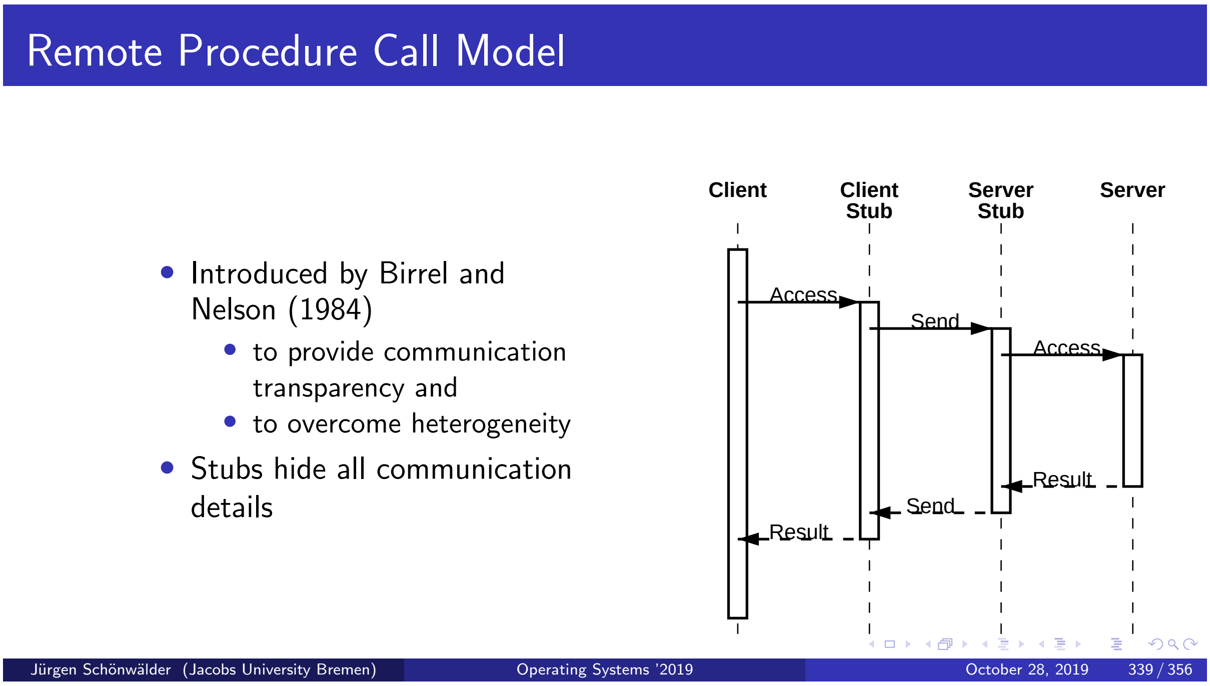
# Section 40: Remote Procedure Calls

39 Definition and Models

40 Remote Procedure Calls

41 Distributed File Systems

42 Distributed Message Queues



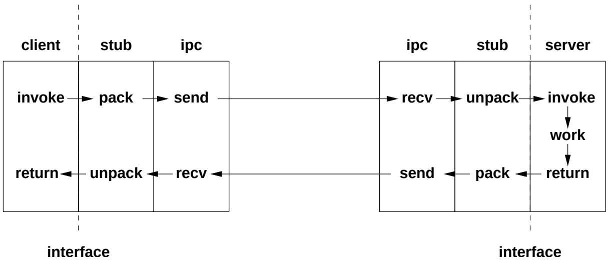
## Remote Procedure Call Model

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

Jürgen Schönwälder (Jacobs University Bremen) Operating Systems '2019 October 28, 2019 339 / 356

The remote procedure model was described in a paper by A. Birrell and P. Nelson [3], which belongs to one of the most cited papers in computer science. A few years later, a survey of remote procedure calls appeared [5], documenting the popularity of this work back in the 1980s.

# Stub Procedures

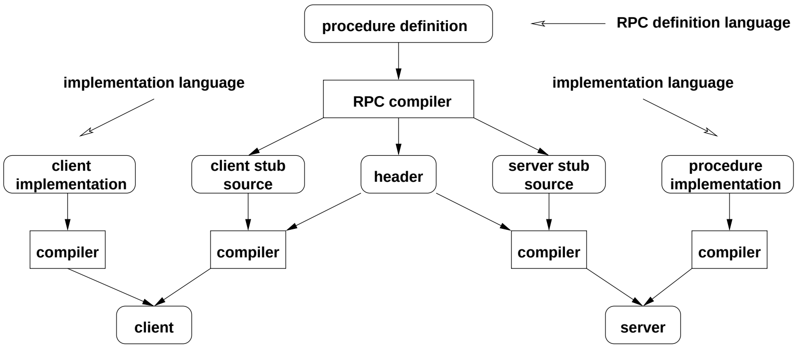


- Client stubs provide a local interface which can be called like any other local procedure
- Server stubs provide the server interface which calls the server's implementation of the procedure provided by a programmer and returns any results back to the client

## 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

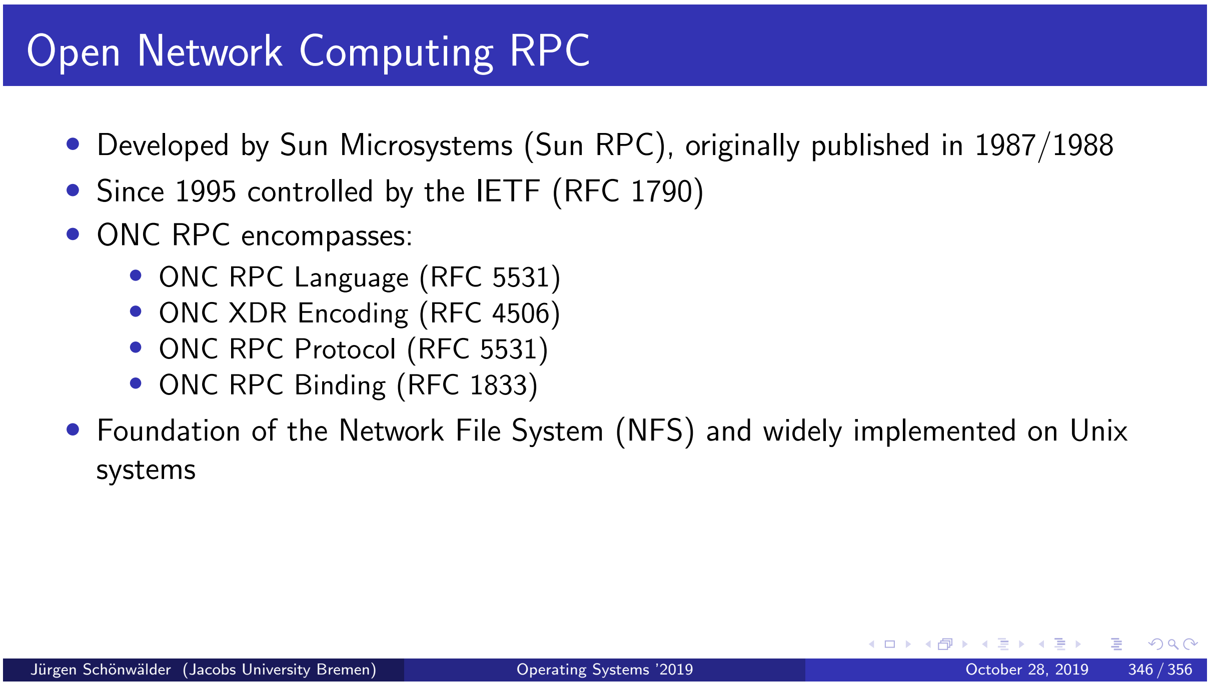
- A client needs to locate and bind to a server in order to use RPCs
- This usually requires to lookup the transport endpoint for a suitable server in some sort of name server:
    1. The name server uses a well know transport address
    2. A server registers with the name server when it starts up
    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

The Open Network Computing (ONC) Remote Procedure Call (RPC) is defined in RFC 5531 [6]. It uses the external data representation (XDR) defined in RFC 4506 [4].

There are many other RPC systems these days, some use XML encoding, some use JavaScript encoding, yet others use newer binary encodings such as Google's gRPC, which uses Google's protocol buffers for data encoding.

# Section 41: Distributed File Systems

39 Definition and Models

40 Remote Procedure Calls

41 Distributed File Systems

42 Distributed Message Queues

# 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 transparency
  - Access transparancy
  - Replication transparency
  - Failure transparency
  - Mobility transparency
  - Scaling transparency
- 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

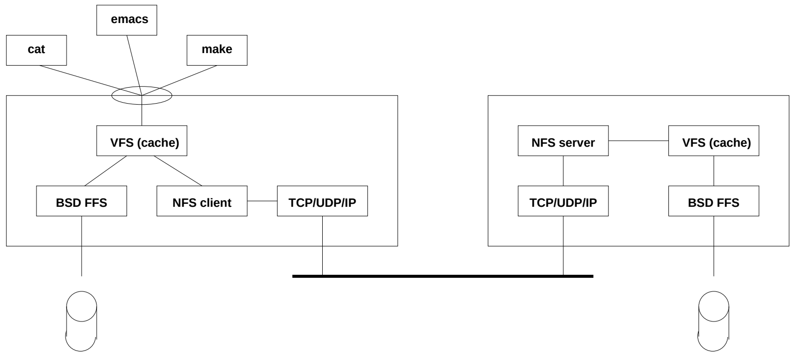
# Stateless vs. Stateful Servers

- 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)

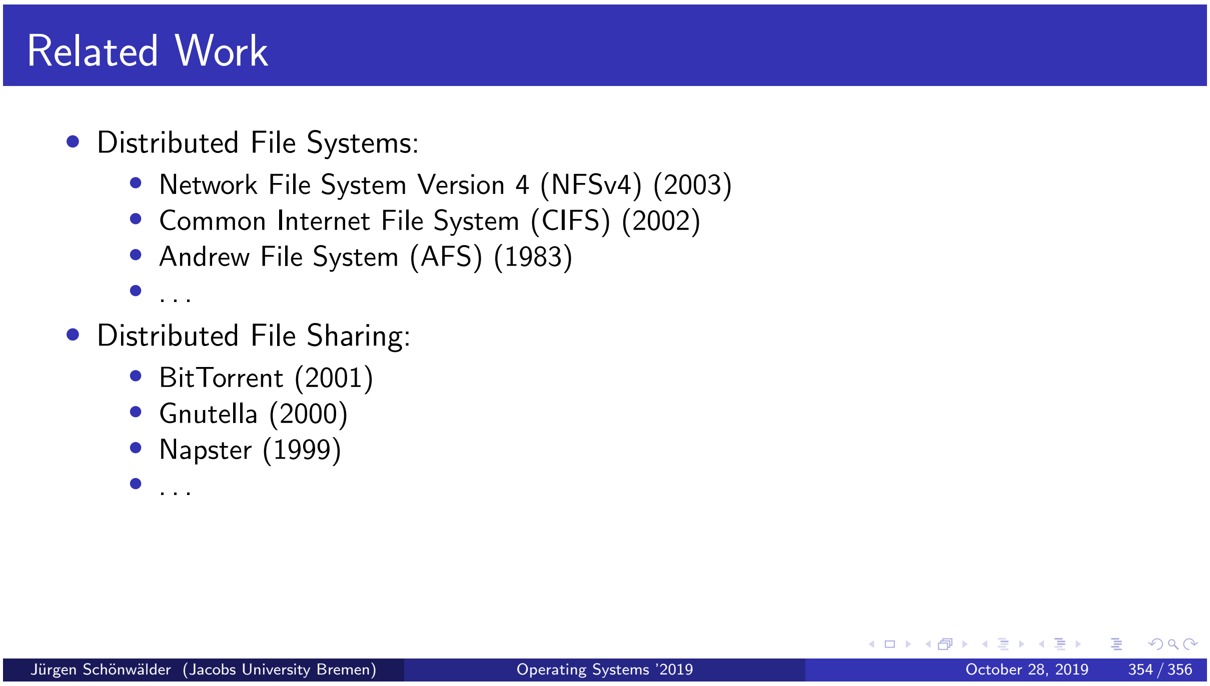
## Operating System Integration



- 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 42: Distributed Message Queues

39 Definition and Models

40 Remote Procedure Calls

41 Distributed File Systems

42 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 . . .

# References

[1] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield. Xen and the Art of Virtualization. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles*, October 2003.

[2] A. Bieszczad, B. Pagurek, and T. White. Mobile Agents for Network Management . *IEEE Communications Surveys*, 1(1), 1998.

[3] A. Birrell and P. Nelson. Implementing Remote Procedure Calls. *ACM Transactions on Computer Systems*, 2(1):39–59, 1984.

[4] M. Eisler. XDR: External Data Representation Standard. RFC 4506, May 2006.

[5] B. H. Tay and A. L. Ananda. A Survey of Remote Procedure Calls. *Operating Systems Review*, 24(3):68–79, July 1990.

[6] R. Thurlow. RPC: Remote Procedure Call Protocol Specification Version 2. RFC 5531, Sun Microsystems, May 2009.