

Course TÖL401G: Stýrikerfi / Operating Systems 1. Introduction

Mainly based on slides and figures subject of
copyright by Silberschatz, Galvin and Gagne

Chapter Objectives

- Describe the role and responsibilities of an operating system (OS).
- Describe the general organization of a computer system and the role of interrupts.
- Describe the components in a modern multiprocessor computer system.
- Illustrate the transition from user mode to kernel mode.
- *Discuss how operating systems are used in various computing environments.*
- Provide examples of free and open-source operating systems.

Why Study Operating Systems?

- Only few will ever create or modify an operating system:
- Why, then, study operating systems and how they work?

- Simply because all code runs on top of an operating system!
 - Knowledge of how operating systems work is crucial for
 - effective (=works at all),
 - Efficient (=works fast, without wasting resource),
 - secure (=no vulnerabilities)
 - programming of any application running on top of an OS.

Contents

1. What is an Operating System?
2. Computer-System Organisation
3. Computer-System Architecture
4. Operating-System Structure
5. Operating-System Operations
6. *Process Management*
7. *Memory Management*
8. *Storage Management*
9. Protection and Security
10. *Distributed Systems*
11. Special-Purpose Systems
12. Computing Environments
13. Open-Source Operating Systems
14. Summary

Just a brief overview on these topics in this chapter 1, remainder of this course dedicates detailed chapters on each of these topics.

1.1 What is an Operating System?

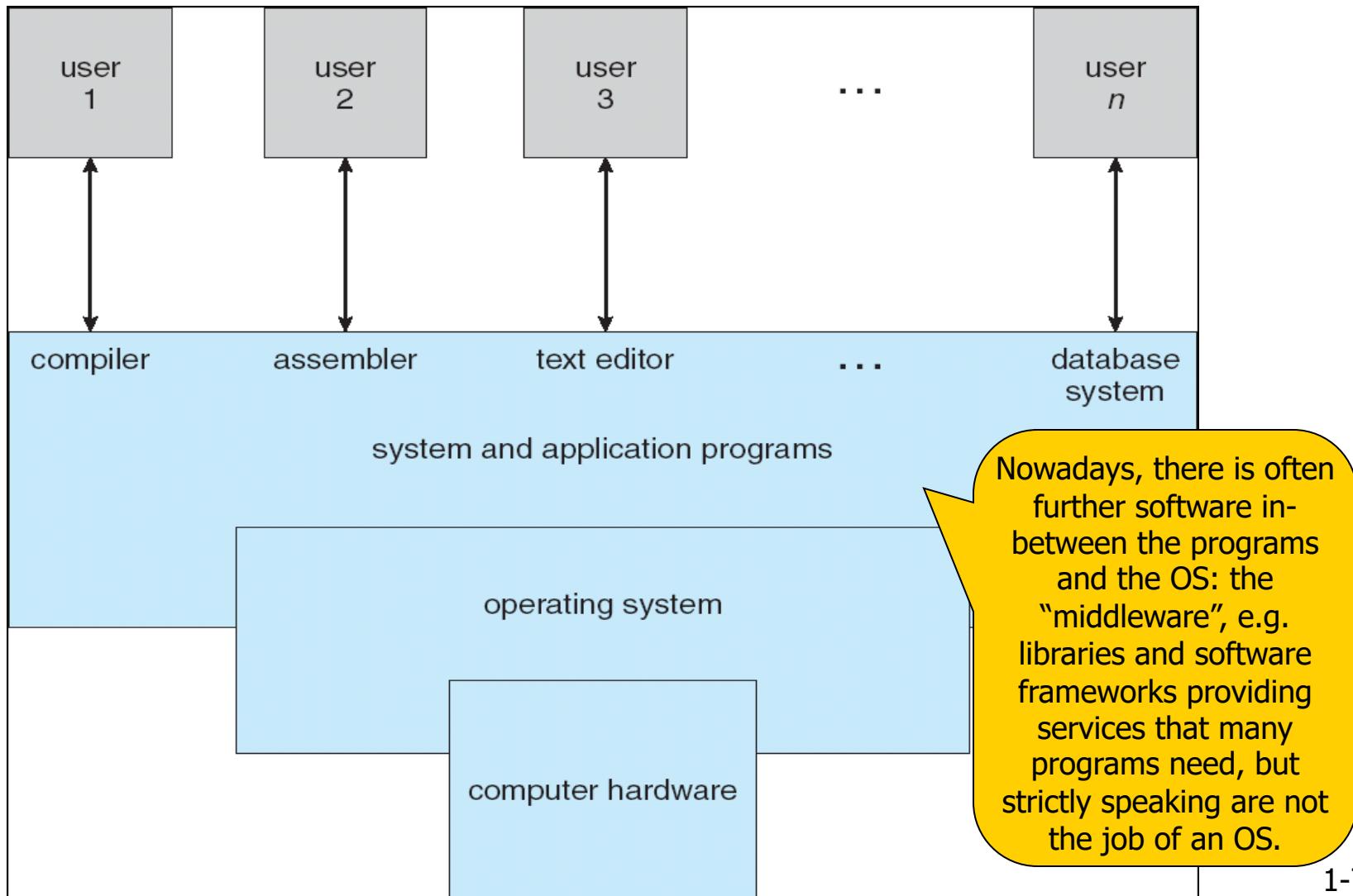
- An operating system (OS) is a program that acts as an intermediary between a user of a computer (and the used application programs) and the computer hardware.
 - Provides environment for other programs & users to do useful work.
- Operating system goals:
 - Make the computer system convenient to use.
 - Use the computer hardware (& software) resources in an efficient manner.

Computer System Structure (1)

- A computer system can be divided into four components:
 - **Hardware** – provides basic computing resources:
 - CPU (Central Processing Unit), memory, I/O (Input/Output) devices.
 - **Operating system**:
 - Controls and coordinates use of hardware among various applications and users.
 - **Programs** – define the ways in which the system resources are used to solve the computing problems of the users.
 - **System programs**: copy files, list directory contents, format disc, etc.
 - **Application programs**: Word processors, compilers, web browsers, database systems, video games, etc.
 - **Users**:
 - People, machines, other computers.
- Relationship of these components as shown on next slide.

Computer System Structure (2)

Relationship of Components



Operating System Definitions (1)

Different definitions exist depending on view point:

- **User view** (Top-down view):
 - OS abstracts away hardware detail in order to **ease the use of the involved hard- and software**. Maximise the work the user performs.
- **System view** (Bottom-up view):
 - OS is a **resource allocator**:
 - Manages all resources.
 - **Resource**: hard- and software components that are relevant for program execution: CPU, Memory, I/O devices, Processes, etc.
 - Decides between conflicting requests for efficient and fair resource use.
 - OS is a **control program**:
 - Controls execution of programs to prevent errors and improper use of the computer.

Operating System Definitions (2)

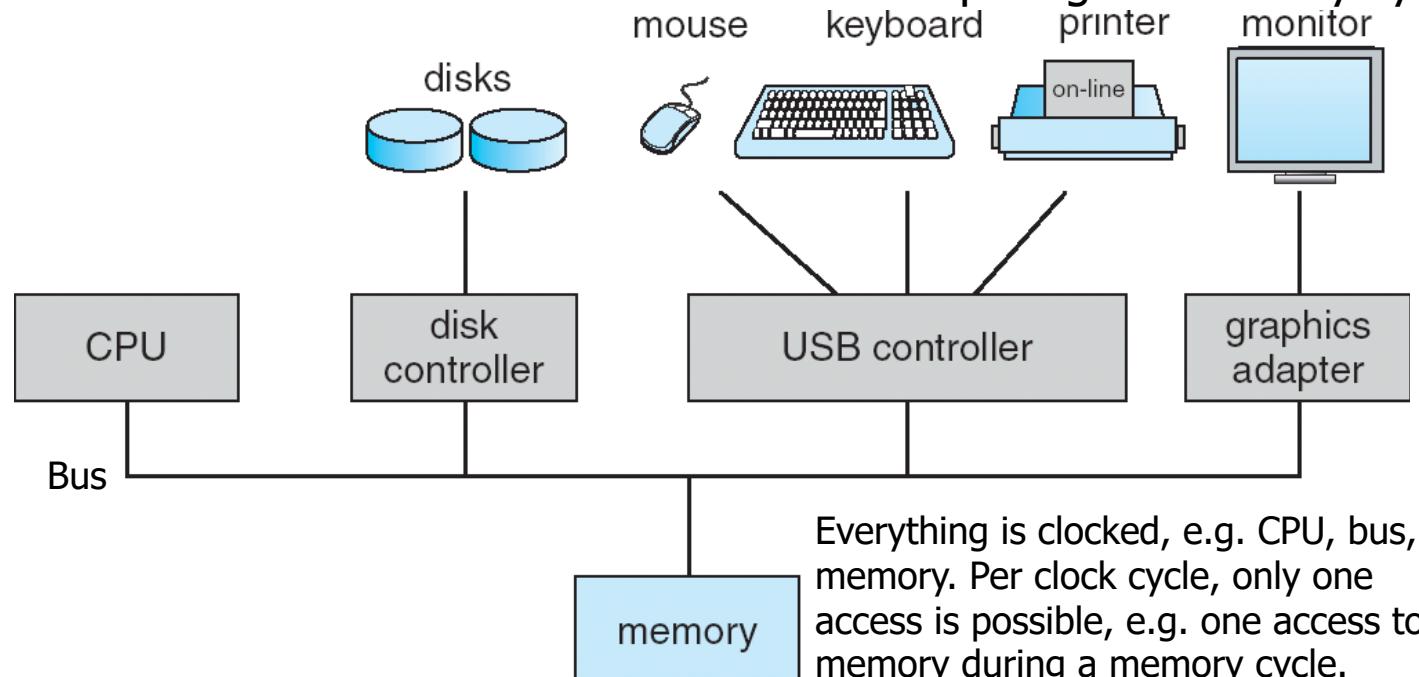
No universally accepted definition exists.

Instead, even various further definitions exist, e.g.:

- OS is “**Everything a vendor ships when you order an operating system.**”
 - Trivia: *Microsoft was found guilty of shipping too much, e.g. bundling Internet Explorer browser, thus preventing competition.*
- “**The one program running at all times on the computer is the **kernel**.** Everything else is either a system program (ships with the operating system) or an application program (installed separately).”
 - System program: deals with resources managed by the OS.
 - E.g. system program to partition a storage device, create/format a filesystem, copy a file, list directory contents

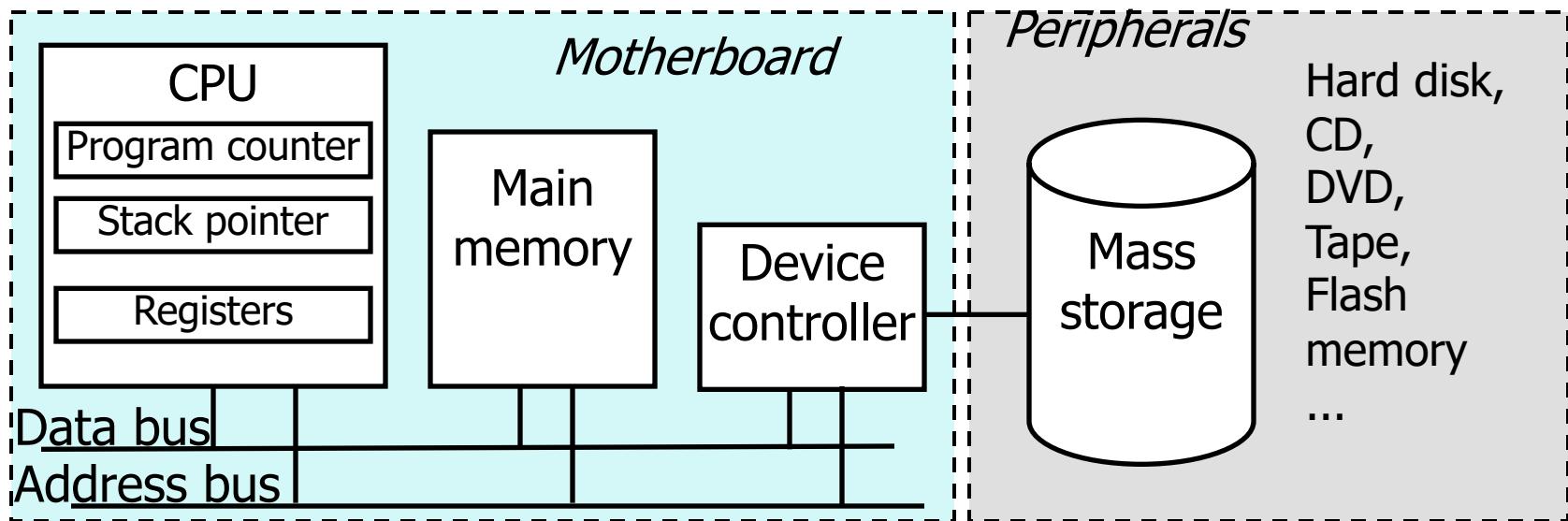
1.2 Computer System Organisation

- Computer-system operation:
 - One or more CPUs, device controllers connected through common bus providing access to shared memory.
 - Concurrent execution of CPUs and devices competing for memory cycles.



- Further details on next slides

Excursion: Von Neumann Architecture



- CPU can only access locations (& execute instructions) in main memory.
 - Mass storage only indirectly accessible via device controller.
- Program counter: points to location of next instruction to be executed.
- Stack pointer: points to top of stack data structure used by CPU to save return address (program counter value) when calling a sub-routine.
- Registers: limited number of general purpose registers that hold, e.g., values of calculations or addresses of data stored in main memory.

Excursion: Bits and Bytes, k vs. K, etc.

- **1 Bit**: smallest unit of information: 0 or 1.
 - (Bit often abbreviated as "b".)
- **1 Byte** = 8 Bits.
 - (Byte abbreviated as "B", e.g. 10 B = 10 Bytes.)
 - Using binary system, possible to represent 2^8 (256) different values.
- **Word size**: number of bits each CPU register and the data bus has:
 - E.g. today's 64 Bit computers have a word size of 64 Bits \Rightarrow in each memory access cycle, 64 Bits can be moved and 2^{64} different values can be stored in a CPU register.
- **1 k** (kilo)= 1000
 - E.g. 1 km = 1000 m
- **1 K** (kilo as binary prefix, using uppercase letter) = 1024
 - E.g. 1 KB = 1024 B
- **Kilo** (K) = 1024 , **Mega** (M) = 1024^2 , **Giga** (G) = 1024^3 , **Tera** (T), **Peta** (P), **Exa**(E)
- For mental math: 2^{10} (=1024) roughly 10^3 (=1000).
 - (E.g. when you are in the final exam without a pocket calculator...)

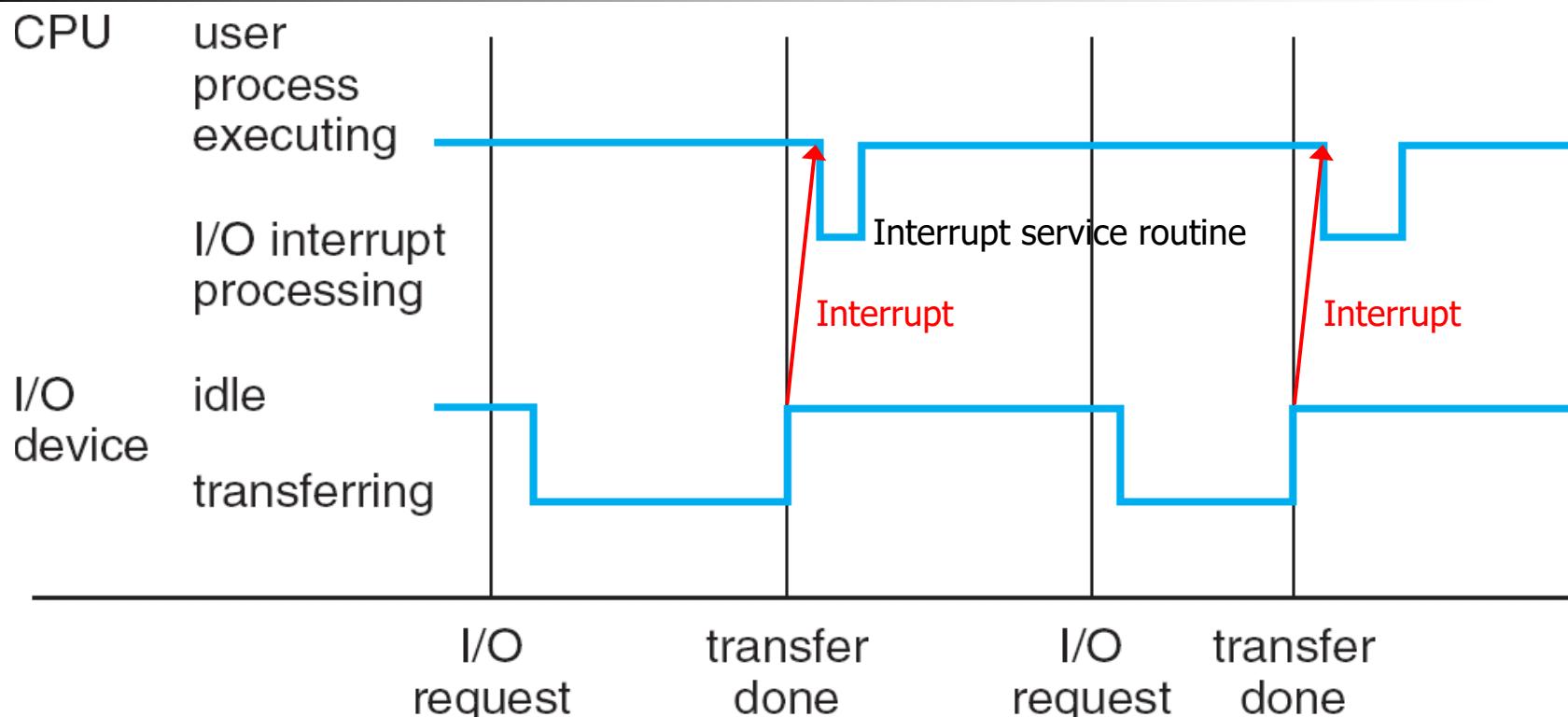
1.2.1 Computer-System Operation

- I/O devices and the CPU can execute concurrently.
- Each device controller is in charge of a particular device type.
- Each device controller has a local buffer for the data that is currently subject of input or output.
- For inputting and outputting data, CPU copies data via the bus between the local buffers of a device controller and main memory.
- Actual I/O is between local buffer of controller and the actual I/O device.
- Device controller informs CPU that it has finished its operation by causing an *interrupt*.

Common Functions of Interrupts

- **Interrupt** transfers control from any instruction that is currently executed by the CPU to the **interrupt service routine**.
 - *Interrupt vector* (=array/table in main memory) contains the addresses of all the service routines (=instructions) for each type/priority of interrupt.
 - Interrupt architecture must **save the address of the interrupted instruction**. Once interrupt service routine completes ex. resumes at the saved address.
 - Usually, interrupts of a lower priority are *disabled/masked* while interrupt of a higher priority is being processed: avoid lower priority interrupts prevent the higher priority interrupt service routine from being completed.
- **Hardware** interrupt such as from the s I/O system
- **SW trap/exception** is a **software-generated interrupt** caused either by an error (e.g. division by zero) or a user/application program request.
 - **System calls** (i.e. calling an OS function) typically by means of a software-generated interrupt.
- An **operating system** is *interrupt driven*.

Interrupt Timeline



- Interrupt processing is **time critical**:
 - Interrupt service routine must have finished before the next interrupt calls it again: If not, previous I/O data would not get completely processed & thus lost.
 - An I/O controller that raises an interrupt, typically needs the data quickly to be processed otherwise the data gets lost (e.g. if I/O device provides new data before the interrupt service routine grabbed the old data from the device controller).

Interrupt Handling

- Operating system
 - provides interrupt service **routines**,
 - initializes interrupt vector accordingly.
 - Configures on multicore systems an APIC (Advanced Programmable Interrupt Controller) to which core(s) which interrupts are routed.
- Interrupt service routine **preserves the state of the CPU** by saving registers and the program counter (typically already saved by the CPU hardware when an interrupt is detected) **and restores it afterwards**.
 - Different segments of code determine what action should be taken for each different type of interrupt, e.g.:
 - Transferring data between main memory and local buffer of device controller.
 - Executing an OS function (**system call**).

1.4 Operating System Structure

- **Multiprogramming** needed for efficiency of (old) **batch systems**:
 - Single user cannot keep CPU and I/O devices busy at all times.
 - Multiprogramming organizes jobs (code and data) so CPU always has one to execute.
 - A subset of total jobs in system is kept in memory.
 - One job selected and run via **job scheduling**.
 - When job has to wait (for I/O for example), OS switches to another job.
 - Rarely used anymore today.
- **Timesharing (multitasking)**: extension of multiprogramming in which CPU switches jobs so frequently (**not only when job has to wait**) that users can interact with each job while it is running, creating **interactive** computing.
 - Response time should be < 1 second.
 - Each user has at least one program (process) executing in memory.
 - If several jobs ready to run at the same time: **CPU scheduling**.
 - If processes don't fit altogether in memory, swapping moves them in and out to run.
 - Virtual memory allows execution of processes not completely in memory.
 - Timesharing used in today's PC operating systems.

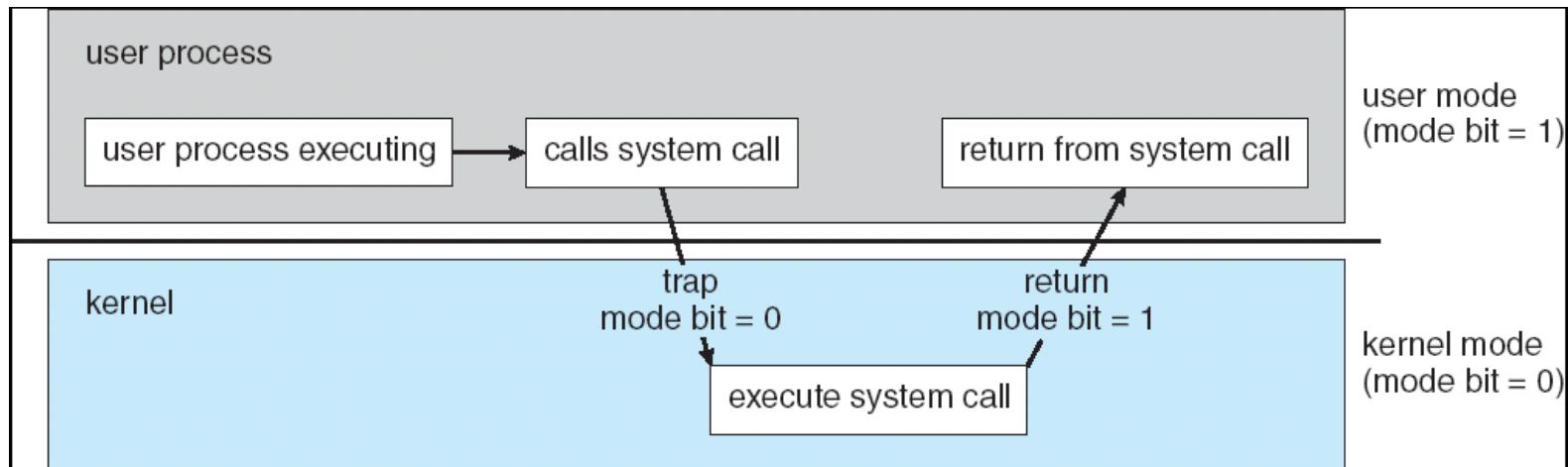
1.5 Operating-System Operations

- An operating system is interrupt driven:
 - **Hardware interrupt**, e.g.:
 - Service an I/O device controller
 - Periodic timer that initiates frequently switching of multitasking jobs.
 - **Software interrupt**, e.g.:
 - Error (e.g. Division by zero)
 - Request for operating system service by application ("system call") using an **exception** or **trap**.

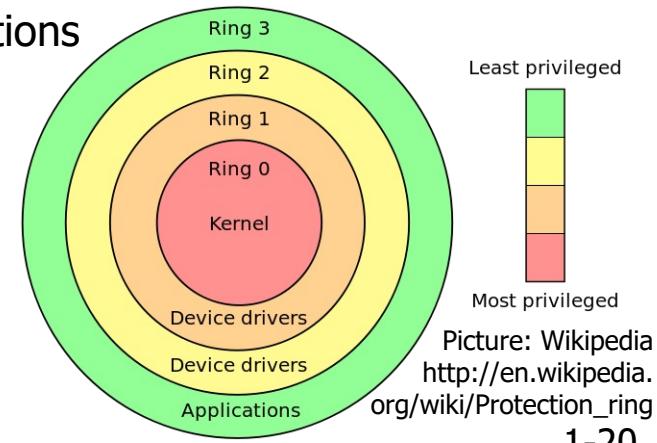
Dual-Mode Operation

- Dual-mode operation allows OS to protect itself and other system components (e.g. processes modifying each other or the operating system):
 - Two modes: User mode and kernel mode (supervisor mode).
 - Mode bit provided by CPU (CPU status bit):
 - Provides ability to distinguish when system is running user code ("User space") or kernel code ("Kernel space").
 - Some CPU instructions are privileged: only executable in kernel mode.
 - Typically, certain memory regions (e.g. containing kernel data structures or access to I/O devices) may only be accessed in kernel mode.
 - Each interrupt changes mode automatically to kernel mode, return from interrupt resets it previous (=typically user) mode.
 - As system calls are called by issuing a software interrupt, the CPU is automatically in kernel mode when a system call shall be executed.
 - **No other way to switch to kernel mode than by an interrupt!**

Transition from User to Kernel Mode



- Memory hardware typically restricts memory access in user mode.
 - ➡ User process can only access its own memory ("Memory protection").
 - Process that runs wild, cannot overwrite instructions or data of other processes or even the kernel.
- A CPU may have more than just the two modes, then typically called "rings".
 - There may be even a ring -1 for running a ring 0 kernel in a virtual machine hypervisor (=ring -1). → Ch. 2



Interrupts and Timer

- Not only a system call made by an application switches into kernel mode, but also each interrupt. E.g.:
 - Interrupt raised by I/O device controller:
 - Now, the kernel needs to serve that I/O device controller.
 - Computer hardware is designed to allow I/O device controller only to be accessed in kernel mode.
 - Otherwise an ordinary user process might circumvent the OS and directly access the hardware.
 - Convenient that interrupt switches to kernel mode needed by kernel!
 - Timer used for multitasking(/timesharing):
 - Set timer interrupt after specific period (e.g. every 20ms).
 - Timer interrupt transfers control to operating system:
 - OS gains control even if process is stuck in an infinite loop.
 - CPU scheduling may take place and select next process to be executed.

1.6 Process Management

- A **process** is a **program in execution**. It is a unit of work within the system.
 - A program is a **passive entity** (program file stored on mass storage device),
 - A process is an **active entity**.
- A **process needs resources** to accomplish its task:
 - CPU, memory, I/O, files, initial data.
 - Process termination requires operating system to reclaim any reusable resources.
- **Single-threaded process** has **one program counter** specifying location of next instruction to execute:
 - Process executes instructions sequentially, one at a time, until completion.
- **Multi-threaded process** has **one program counter per thread**.
 - More on threads in chapter 4.
- A typical system has many processes, few users, one operating system running concurrently on one or more CPUs/cores.
 - Concurrency by multiplexing the CPUs among the processes / threads (multitasking/timesharing).

Process Management Activities

- Process management activities of operating system:
 - Creating and deleting both user and system processes,
 - Scheduling processes (and threads) on the CPUs,
 - Suspending and resuming processes,
 - Providing mechanisms for process synchronization,
 - Providing mechanisms for process communication.

1.7 Main Memory Management

- **Instructions** of a process need to be in main memory in order to execute process.
- **Data** of a process need to be in main memory during processing.
- **Memory management** determines what is in main memory when optimizing CPU and resource utilization and computer response to users.
- **Memory management activities:**
 - Keeping track of which parts of main memory are currently being used and by whom.
 - Deciding which processes (or parts thereof) and data to move into and out of main memory.
 - Allocating and deallocating main memory space as needed.

Low-Level Mass-Storage Management

- Traditionally, disks (now trend to SSD) used to store data that does not fit completely in main memory (e.g. virtual memory) or data that must be kept for a long period of time (e.g. files).
 - Proper management is of central importance.
 - Entire speed of computer operation depends on disk subsystem and the operating system's algorithms used to manage it.
- Low-level mass-storage management activities:
 - Free-space management,
 - Storage allocation,
 - Disk scheduling.
- Some storage needs not be that fast:
 - Tertiary storage includes optical storage, magnetic tape.
 - Seldom-used data, thus speed not crucial. But: still must be managed!

Caching

- Important principle, performed at many levels in a computer (in hardware, operating system, software).
- Information in use copied from slower to faster storage temporarily.
- Faster storage (cache) checked first to see if information is there:
 - If it is, information used directly from the cache (fast).
 - If not, data copied to cache and used there.
- Cache smaller than storage being cached (as faster cache memory is more expensive than slower memory).
 - Cache management: important design problem.
 - Cache size and replacement policy.
- Caching at different levels, e.g.:
 - CPU cache as cache for main memory.
 - Main memory as a cache for secondary storage.
 - Web browser cache as cache for network transmission.

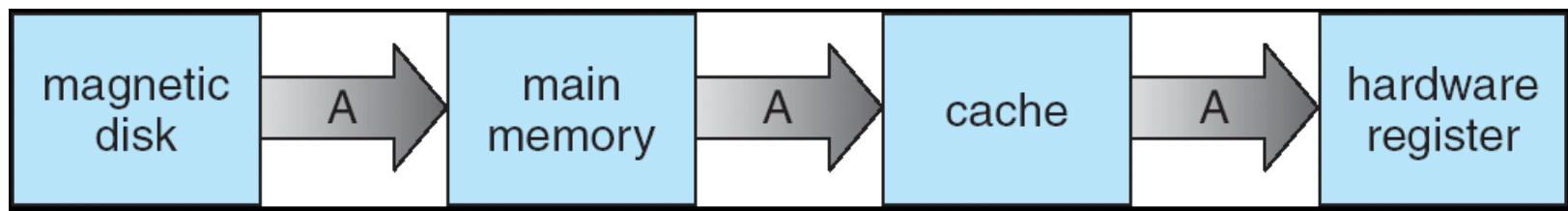
Performance of Various Levels of Storage

Level	1	2	3	4	5
Name	registers	cache	main memory	solid state disk	magnetic disk
Typical size	< 1 KB	< 16MB	< 64GB	< 1 TB	< 10 TB
Implementation technology	custom memory with multiple ports CMOS	on-chip or off-chip CMOS SRAM	CMOS DRAM	flash memory	magnetic disk
Access time (ns)	0.25 - 0.5	0.5 - 25	80 - 250	25,000 - 50,000	5,000,000
Bandwidth (MB/sec)	20,000 - 100,000	5,000 - 10,000	1,000 - 5,000	500	20 - 150
Managed by	compiler	hardware	operating system	operating system	operating system
Backed by	cache	main memory	disk	disk	disk or tape

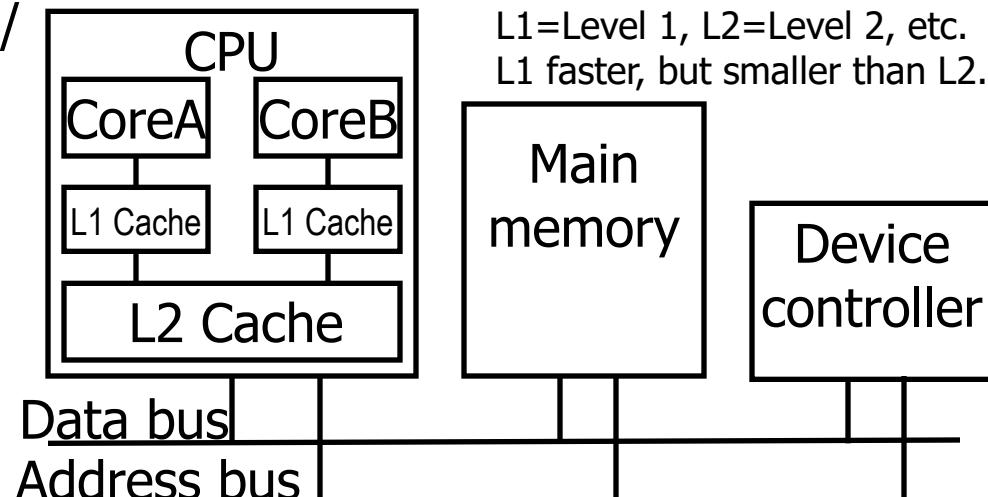
- In the past, cost was the main driver that prevented to use larger main memory.
- Then, the maximum memory addressable by a 32 Bit CPU became a restriction:
 - 32 Bit CPU uses 32 Bit to address memory: maximum memory addressable with 32 Bit: 4GB
- Today's 64 Bit CPUs allows to address larger memory.

Migration of Integer A from Disk to CPU Register

- Care must be taken to use most recent value, no matter where it is stored in the storage hierarchy.



- Multiprocessor environment must provide **cache coherency** in hardware such that all CPUs/cores have the most recent value in their cache.
 - E.g.: L1 cache of Core A must be updated if Core B changes a value in main memory that is cached by L1 cache of Core A.



1.9 Protection and Security

- **Protection:** any mechanism for controlling access of processes or users to resources defined by the OS. [Protection within OS'es](#)
- **Security:** defense of the system against internal and external attacks.
 - Huge range: denial-of-service, worms, viruses, identity theft, theft of service, ...[Cyber security](#).
 - To realize protection and security, operating systems generally distinguish among users, to determine who can do what:
 - **User identities** ([user IDs](#), security IDs) include name and associated number, one per user.
 - User ID then associated with all files, processes of that user to determine access control.
 - **Group identifier** ([group ID](#)) allows set of users to be defined and to manage access control for whole groups of users.
 - Group ID also associated with each process, file.
 - **Privilege escalation** allows user to change to effective ID with more rights.

1.11 Special Purpose Systems: Embedded systems

- Embedded computers today found everywhere:
 - Cars, DVD players, microwave ovens, etc.
- Have **very specific tasks**.
 - No general purpose system.
 - **Restricted hardware**, e.g.
 - No mass storage device,
 - User interface just a few buttons and status light.
 - OS provides **limited features**, e.g.
 - No mass storage management,
 - Little user interface.

Special Purpose Systems: Real-Time Operating System

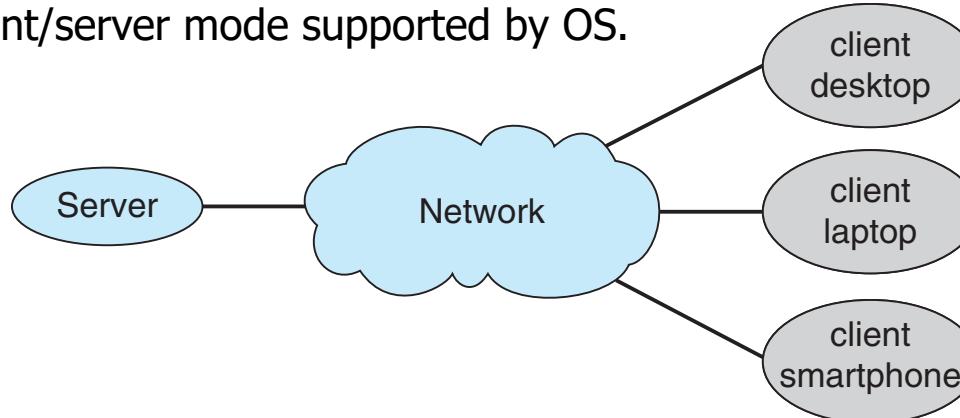
- Embedded systems often run real-time OS.
- Used in controlling devices: processing *must* be done within a defined time constraint, otherwise the system will fail. E.g.:
 - Industrial control system,
 - Traffic control system,
 - Medical systems,
 - Multi-media system (audio/video must be delivered in time to avoid stuttering).
- Real-time OS needs to support processing with well-defined fixed time constraints (ROS-kernel $\leq 1\text{ms}$).
 - Must be taken into account when designing e.g. CPU scheduler.

Special Purpose Systems: Mobile Computing

- Smartphones and tablets allow mobile computing.
 - On one hand restricted hardware (e.g. no keyboard, no mouse, no hard disk) in comparison to a PC,
 - but on the other hand even more hardware (e.g. GPS, compass, accelerometer).
 - Tablets and laptops converge and handheld devices have nowadays functionalities like an ordinary computer.
 - Generic OS (e.g. Android based on Linux), however with
 - support for power saving (due to limited battery),
 - special user interface (touch instead of physical keyboard or mouse),
 - wireless (WiFi) and cellular data network (3G, 4G, 5G) support,
 - limited storage capacity support (no hard-disk drivers).

1.12 Computing Environments: Client-Server Computing

- Client-Server Computing / Distributed System:
 - Computer as **servers**, responding to requests generated by **clients**.
 - Compute-server provides an interface to client to request services, e.g. database.
 - File-server provides interface for clients to store and retrieve files.
 - Client/server mode supported by OS.



- Web-based Computing (special application of Client-Server computing):
 - Web server; Web browser as client to access powerful cloud services.
 - Web has become ubiquitous: most devices networked to allow web access.
 - E.g. Chromebook: Linux-based OS, mainly providing a web browser.

1.13 Free and Open-Source vs. Closed Source Operating Systems

- **Free and Open-source software (FOSS)**: Source code of OS (e.g. Linux) can be accessed (open-source) and modified and redistributed for free (free software).
 - Good for learning about OS implementation.
 - Download the source code of the Linux kernel from <http://kernel.org> and have a look at it.
 - Possible to fix bugs and extend on your own.
 - Arguably, more secure: many eyes can view code to find problems.
- **Closed-source**: Only compiled binary available (e.g. Microsoft Windows).
 - Need to rely on support of vendor to get bugs fixed.
- **Hybrid**: E.g. Apple Mac OS X: Darwin OS kernel is FOSS, system programs and middleware (e.g. GUI) are not.

History of Unix(-like) Operating Systems

1969

Together with MS Windows
(which has no Unix history),
you find all today's
relevant OSes here.

1971 to 1973

1974 to 1975

1978

1979

1980

1981

1982

1983

1984

1985

1986

1987

1988

1989

1990

1991

1992

1993

1994

1995

1996

1997

1998

1999

2000

2001 to 2004

2005

2006 to 2007

2008

2009

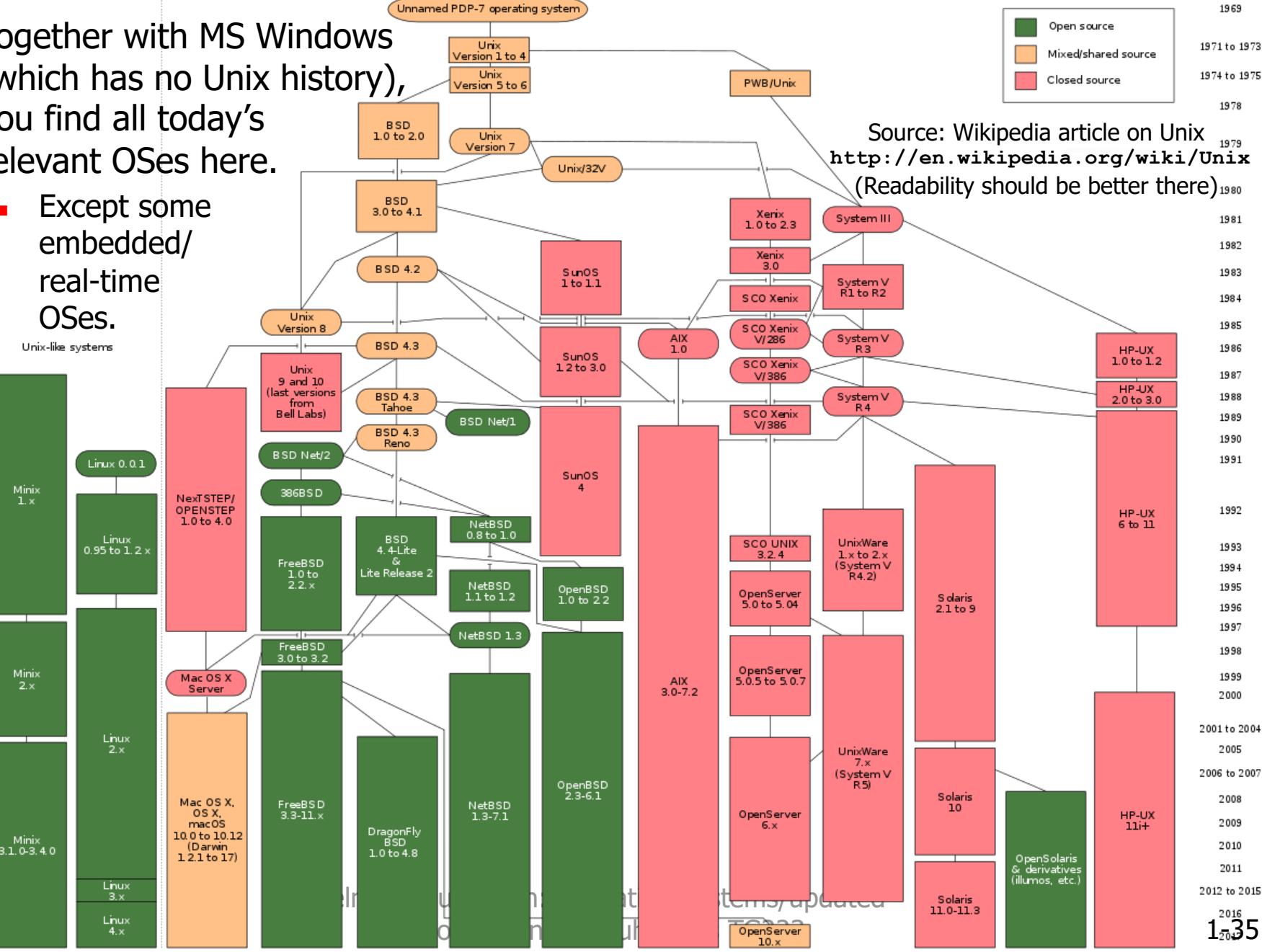
2010

2011

2012 to 2015

2016

2017



1.14 Summary

- Operating system mediates between user/application and underlying hardware.
 - Manages all the resources.
 - Lots of responsibilities.
 - We will have a closer look on each of them in the following chapters.

Course TÖL401G: Stýrikerfi / Operating Systems

2. Operating System Structures

Mainly based on slides and figures subject of
copyright by Silberschatz, Galvin and Gagne

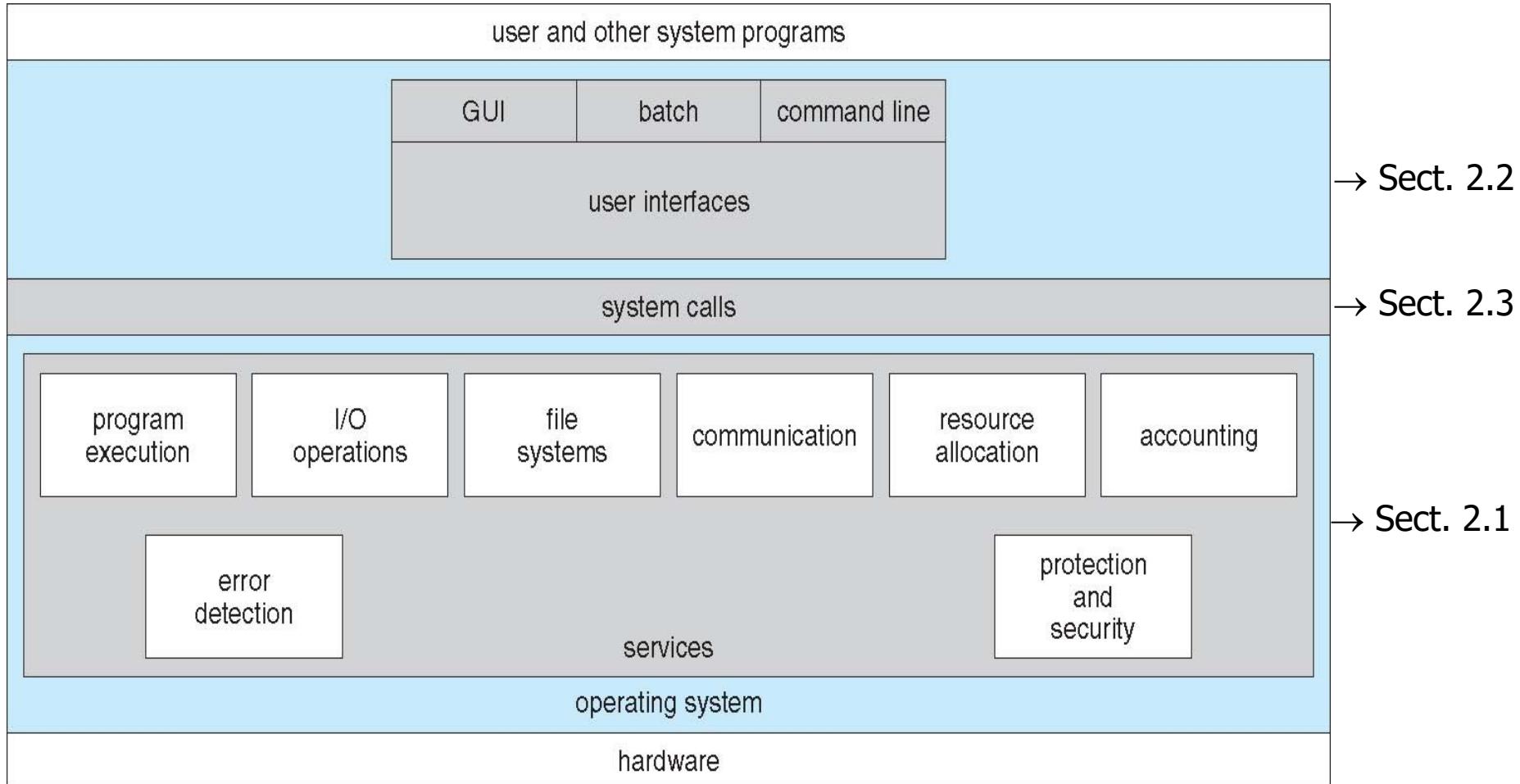
Chapter Objectives

- Identify services provided by an operating system.
- Illustrate how system calls are used to provide operating system services.
- Compare and contrast monolithic, layered, microkernel, modular, and hybrid strategies for designing operating systems.
- Illustrate the process for booting an operating system.
- Learn tools for monitoring operating systems.

Contents

1. Operating System Services
2. User Interface of Operating Systems
3. System Calls
4. Types of System Calls
5. System Programs
6. Operating Systems Design and Implementation
7. Operating Systems Structure
8. Virtual Machines
9. Java
10. Operating System Debugging
11. Configuration and Generation of Operating Systems
12. System Boot
13. Summary

Operating System Services, User Interfaces, and System Calls: Overview



2.1 Operating System Services: Services for the user/application (1)

- Operating system provides services that are helpful to the user and application programmer:
 - User interface:
 - More on this in section 2.2.
 - Program execution:
 - The system must be able to load a program into memory and to run that program, to end execution either normally or abnormally (indicating an error).
 - I/O (Input/Output) operations:
 - A running program may require I/O, which may involve a file (see below) or a particular I/O device.
 - File-system manipulation:
 - Programs need to read and write files and directories, create and delete them, search them, list file information, permission management.

Operating System Services: Services for the user/application (2)

- Further helpful operating system services:
 - Communications:
 - Processes may exchange information, on the same computer or between computers over a network.
 - Communications may be via shared memory or through message passing (packets moved by the OS).
 - Error detection:
 - OS needs to be constantly aware of possible errors:
 - May occur in CPU & memory hardware, in I/O devices or user program.
 - For each type of error, OS should take the appropriate action to ensure correct and consistent computing.
 - Debugging facilities can greatly enhance the user's and programmer's abilities to efficiently use the system.

Operating System Services: Services for the operation of the system itself (1)

- OS services for ensuring the efficient operation (=efficient sharing of resources) of the system itself:
 - **Resource allocation:**
 - Resources need to be allocated to users or jobs running concurrently.
 - Allocation strategies (and corresponding implementation) dependent on type of resources:
 - Allocation strategy may be specific to resource, e.g. CPU scheduler, memory management, file storage.
 - Generic allocation strategies for other resources, e.g. all I/O devices may be allocated and released in the same way.
 - **Accounting:**
 - To keep track of which users use how much and what kinds of computer resources (e.g. for billing or for identifying bottlenecks).

Operating System Services: Services for the operation of the system itself (2)

- Further OS services for ensuring the operation of the system itself:
 - Protection and security:
 - Access of information stored in a multiuser or networked computer system must be controllable.
 - Concurrent processes should not interfere with each other.
 - Protection: involves ensuring that all access to system resources from system users is controlled.
 - Security: prevent access from outsiders. Requires user authentication, extends to defending external I/O devices from invalid access attempts.

2.2 User Interface of Operating Systems

- Almost all OS have a user interface (UI).
 - Varies between
 - Interactive UI, e.g.:
 - Command-Line Interface (CLI), see next slide.
 - Graphical User Interface (GUI), see slide after next slide.
 - Non-interactive UI, e.g.:
 - Batch interface:
 - User specifies all details of a batch job in advance to the actual batch processing. (Using some job submission language that is similar to the command-line interface.)
 - No prompt for further input after the processing has started.
 - User receives the output when all the processing is done.

Command-Line Interface (CLI)

- CLI allows direct command entry via keyboard.
 - Output as simple characters on the monitor.
- Sometimes CLI implemented in kernel, sometimes by a separate system program (**shell**).
 - Sometimes multiple flavors of shells implemented.
- Primarily fetches a user textual command and executes it.
 - Sometimes **commands built-in** into shell,
 - Sometimes commands are just names of **external programs** that will be executed in a new process (eg.:from a system library).
 - Adding new features doesn't require shell modification.

Graphical User Interface (GUI)

- User-friendly graphical interface based on **desktop metaphor** (invented at Xerox PARC in the early 1970ies).
 - Usually mouse, keyboard, and monitor.
 - Icons represent files, programs, actions, etc.
 - Various mouse buttons in objects interface cause various actions.
 - Mobile systems lack a mouse: touch interface with gestures.
- Many systems now include both CLI and GUI interfaces:
 - Microsoft Windows is GUI with CLI “command” shell
 - Apple Mac OS X as “Aqua” GUI interface with UNIX kernel underneath and shells available.
 - GNU/Linux is CLI with optional GUI interfaces (Gnome, KDE, etc.)
 - Android is Linux kernel with touch GUI and frameworks from Google.

2.3 System Calls

- Include program interface to services provided by the OS.
 - Technically realised as software interrupts (traps) callable via special **assembly-language trap instruction**.
 - Application Binary Interface (ABI).
 - Mostly accessed by programs **via** an assembly-language independent library providing high-level **Application Programming Interface (API)** rather than direct system call use.
 - API may be operating system specific or/and programming language specific (see next slide).
 - (Note that system-call names used in this course are often POSIX names (→next slide), but sometimes, we abstract from a certain operating system and use rather generic system-call names.)

System Calls: APIs

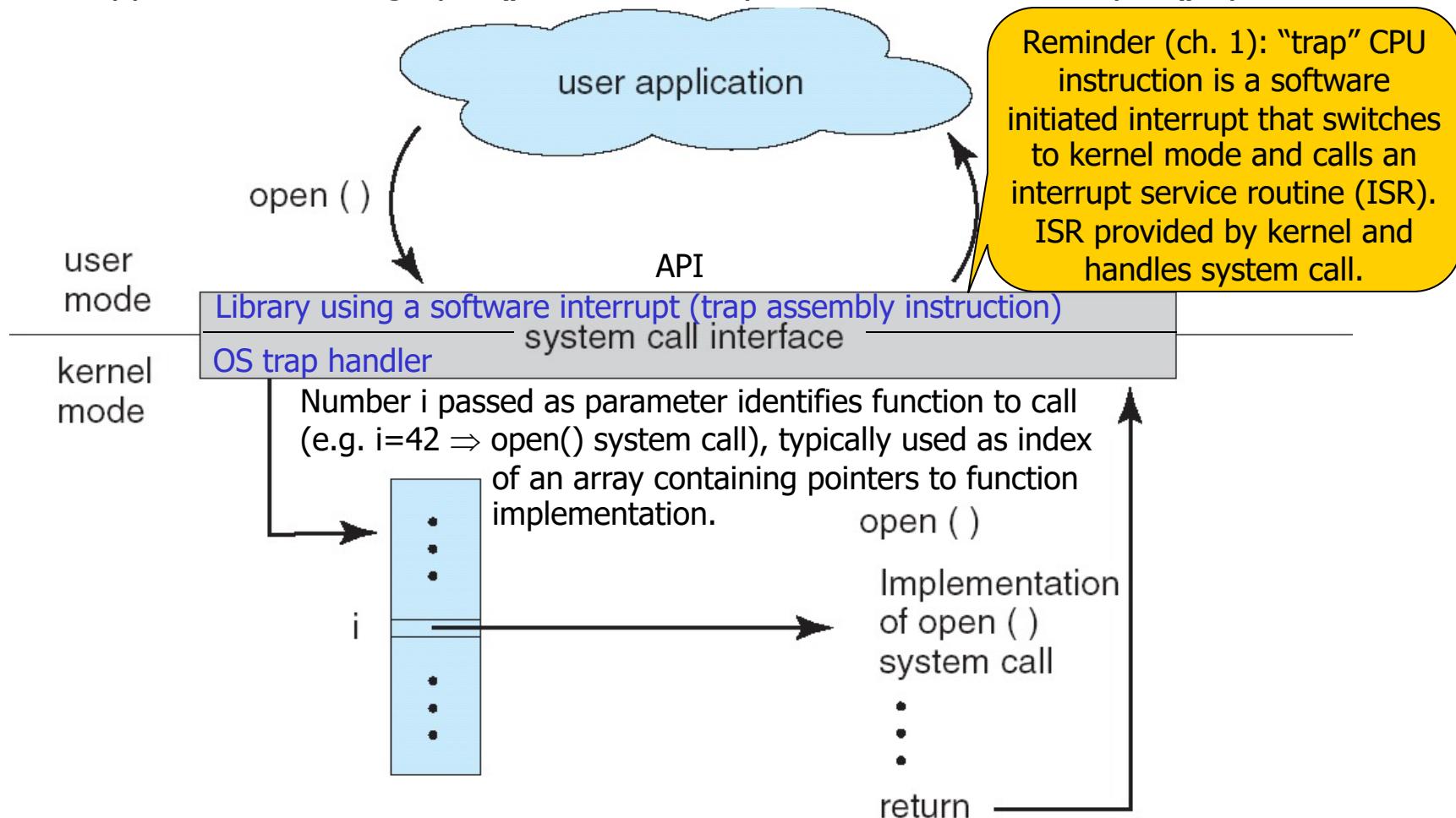
- Most common OS-specific APIs:
 - Windows API for Microsoft Windows.
 - Corresponding libraries available for all major MS programming languages.
 - POSIX API for POSIX-based systems, i.e. virtually all commercial versions of UNIX, e.g. Mac OS X.
 - (POSIX is a standard for the API of UNIX-like operating systems.)
 - C library that maps directly to system calls of POSIX-based systems.
 - For non-POSIX systems, a POSIX API library may try to emulate POSIX functionality (e.g. CYGWIN for MS Windows or MS Windows Subsystem for Linux).
- Many programming languages come with operating system-independent APIs for accessing common OS services (for specific OS services, still the OS API needs to be used). E.g.
 - Standard C library,
 - Java API for the Java virtual machine (JVM).
 - Internally, these make then system calls using OS-specific API.

System Call Implementation

- Typically, a **number** is associated with each system call.
 - Number identifies associated routine in OS kernel.
 - Number is used as an index of a table that contains addresses of functions that implement the corresponding system call.
- The system call **API** invokes intended system call in OS kernel **by passing number (and additional parameters)** using a **trap assembly instruction**. Result of the system call is **retrieved** by API and **returned to caller**.
- Caller needs not knowing how system call is implemented
Just needs to obey API and understand what OS will do as result.
 - Most details of OS interface hidden from programmer by API .

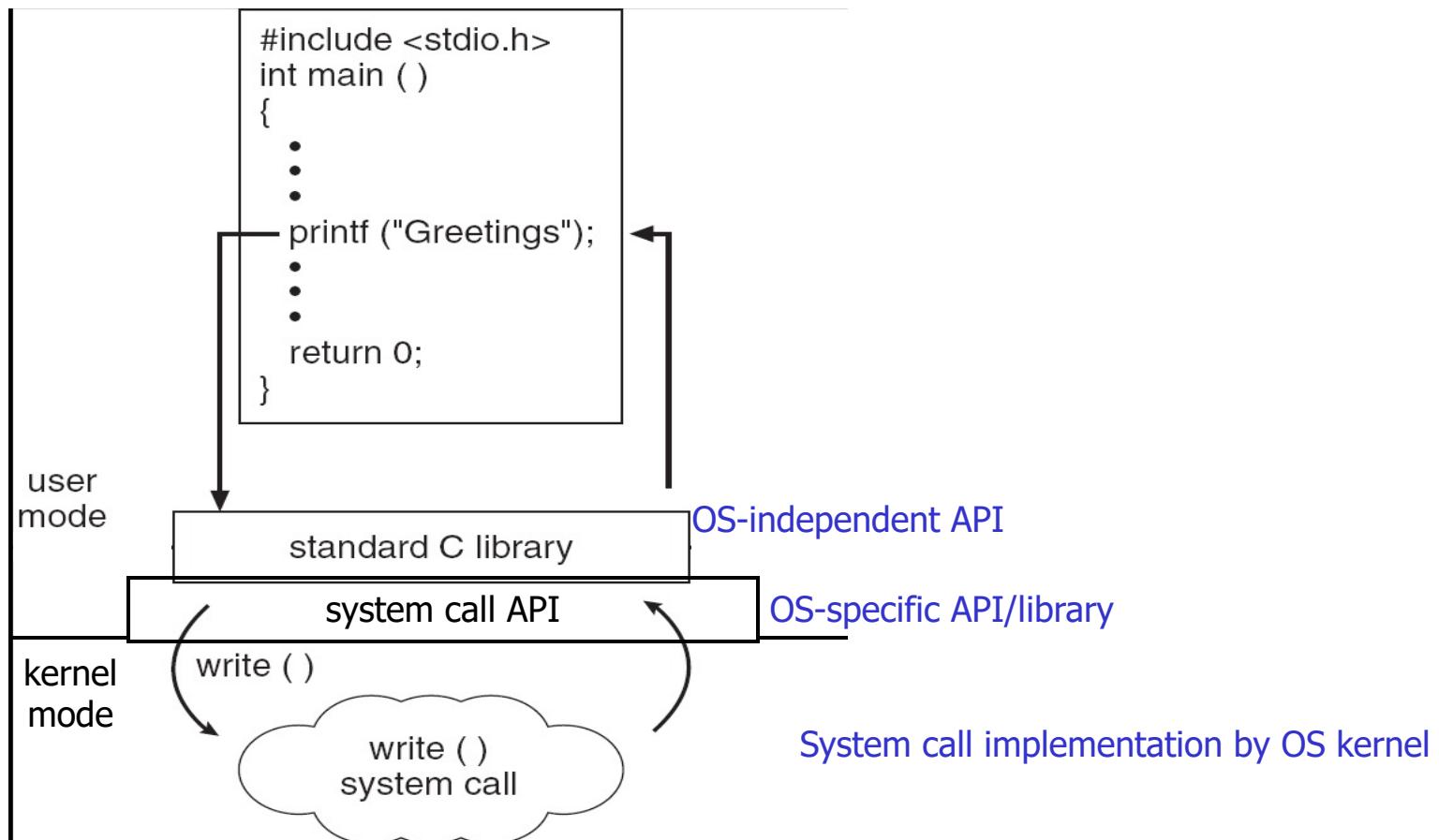
API – System Call – OS Relationship

- User application invoking open() call offered by API, which calls the open() system call:



Standard C Library Example

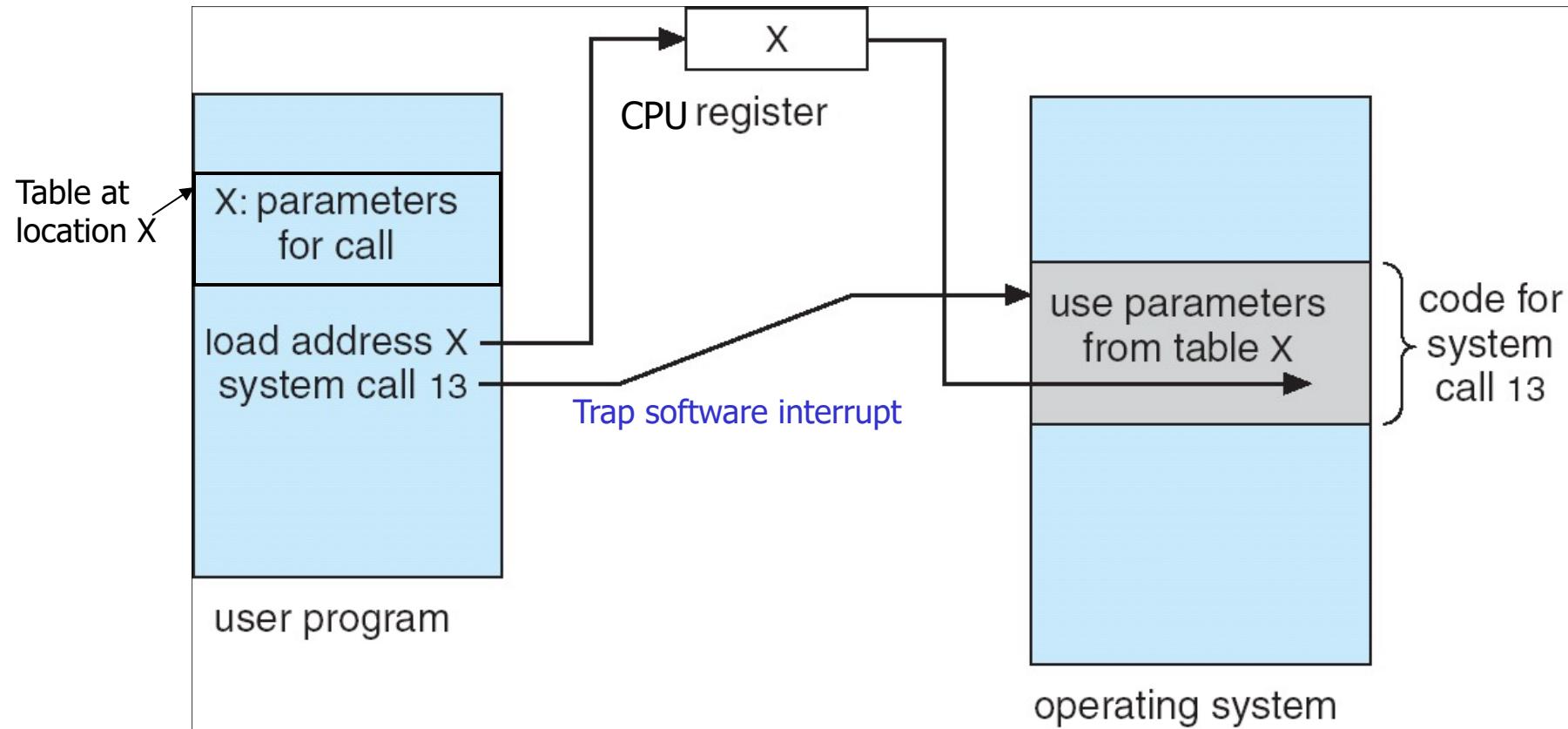
- C program invoking printf() Standard C library call, which calls write() system call:



System Call Parameter Passing

- Three possible approaches passing parameters to the OS?
 - Pass the parameters in **CPU registers**. Simple, but not sufficient
 - If more parameters than registers shall be passed.
 - Parameters stored in a **block**, or table, **in memory**, and **address of block passed as a parameter in a register** (see next slide).
 - This approach taken by Linux and Solaris.
 - Parameters placed, or pushed, onto the **stack** (pointed to by the Stack Pointer (SP) CPU register) by the program and popped off the stack by the operating system.
 - Block and stack methods do not limit the number or length of parameters being passed.

Parameter Passing via Table



Difference API ↔ ABI

- Application Programming Interface (API) is the interface used by a high-level programming language program.
 - If API changes (e.g. new function/method name or change of parameters) then the whole application program source needs to be adjusted to be able to re-compiled.
- Application Binary Interface (ABI) is the interface used on assembly/machine code level.
 - ABI change: different number used for an existing system call or different order or format of parameters (=caller & callee assume different parameter passing tables).
 - All compiled programs (=compiled into binary code and making use of the ABI) will fail when trying to call this system call.
 - As long as the API did not change, it is enough to recompile the source code of the application program to deal with ABI changes: the library that implements the API will then internally use the new ABI. (Of course library, needs to updated to the new ABI, first.)
- ABI changes not only a problem at OS level, but also for Java, C++ etc:
 - E.g. method parameter (=API & ABI change) change in Java class B
⇒ Java class A calling method from B, but compiled using old API contains byte code that passes now wrong parameters (=relying on old ABI).
- Both, API and ABI changes should be avoided (rather add additional calls).

Documentation of POSIX System Calls

If man pages for system calls are not installed on your Linux system, install packages: `manpages-dev` `manpages-posix-dev` (your Linux distribution may use different names).

- On most POSIX systems (Linux, Mac OS X etc.), a manual page ("man page") can be called from command line to obtain further information of how to use a system call from within a C program:
 - `man -S s x` provides documentation for system call `x`:
 - Required C include files,
 - Parameters and return values,
 - Short description,
 - Related system calls,
 - And many further useful information.
 - `s` specifies section of manual, e.g.:
 - 1 Shell command or system program,
 - 2 System calls (Platform specific system calls as provided by operating system),
 - 3 Library calls (Functions provided by C standard library).
 - If `-S s` is not used as parameter, the first section having an entry for `x` is used.
- Example:
 - `man kill` : description of shell command kill (kill first found in section 1),
 - `man -S 2 kill` : description of system call kill.

How to read man page of an API

As an example of a standard API, consider the `read()` function that is available in UNIX and Linux systems. The API for this function is obtained from the `man` page by invoking the command

```
man read
```

on the command line. A description of this API appears below:

```
#include <unistd.h>

ssize_t      read(int fd, void *buf, size_t count)
```

return value function name parameters

A program that uses the `read()` function must include the `unistd.h` header file, as this file defines the `ssize_t` and `size_t` data types (among other things). The parameters passed to `read()` are as follows:

- `int fd`—the file descriptor to be read
- `void *buf`—a buffer where the data will be read into
- `size_t count`—the maximum number of bytes to be read into the buffer

On a successful read, the number of bytes read is returned. A return value of 0 indicates end of file. If an error occurs, `read()` returns `-1`.

2.4 Types of System Calls

- Roughly six major categories of system calls (most of them refer to related OS services):
 - Process control,
 - File management,
 - Device management,
 - Information maintenance,
 - Communications,
 - Protection.
- (Most of these will be covered in detail in later chapters.
On the following slides, only a brief overview is given.)

Process Control System Calls

- End, abort (=abnormal halt) process.
- Load, execute process.
- Create process, terminate process.
- Get/set process attributes.
- Wait for a certain amount of time.
- Signal event, wait for event.
- Allocate memory, free memory.

2.5 System Programs/ System Services

- System programs are delivered together with an OS and provide a convenient environment for program development and execution.
 - Some of them are simply user interfaces to system calls; others are considerably more complex.
 - Most users' view of the operation system is defined by system programs, not the actual system calls.
- In the past, these were mainly command line tools.
 - (e.g. `format` to create a new file system, `cp` to copy a file.)
- Nowadays, these are provided as a system service via a GUI.

System Programs

- System programs can be divided into (details on the next slides...):
 - File management,
 - Status information,
 - File modification,
 - Programming language support,
 - Program loading and execution,
 - Communications,
 - Application programs,
 - Background programs.

Categories of System Programs (1)

- **File management:**
 - Create, delete, copy, rename, print, dump, list, and generally manipulate files and directories.
- **Status information:**
 - Some ask the system for info:
 - date, time, amount of available memory, disk space, number of users.
 - Typically, these programs format and print the output to the terminal or other output devices
 - Some systems (e.g. MS Windows) implement a registry:
 - Used to store and retrieve configuration information.
- **File modification:**
 - Text editors to create and modify files.
 - Special commands to search contents of files or perform transformations of the text.
- **Programming-language support:**
 - Compilers, assemblers, debuggers and interpreters.

Categories of System Programs (2)

- Program loading and execution:
 - Command interpreter (shell).
 - Linker, tracing of system calls.
- Communications:
 - Provide the mechanism for creating virtual connections among processes, users, and computer systems.
 - Allow users to send messages to one another's screens, browse web pages, send electronic-mail messages, log in remotely, transfer files from one machine to another.
- Application programs:
 - Additional programs for solving application specific problems (Word processor, graphic editor, games, etc.)
- Background programs ("daemons"):
 - Print spooler for buffering print jobs while printer is busy, syslog for logging events, execution of commands at scheduled times ("cron jobs"), etc.

2.6 Operating System Design and Implementation

- There is no standard design for operating systems.
 - Internal structure of different operating systems can vary widely.
- Proven approach:
 - Start by defining goals and specifications:
 - Choice of hardware, type of OS (batch, multitasking, etc.).
 - Identify User goals and System goals:
 - **User goals:** operating system should be convenient to use, easy to learn, reliable, safe, and fast.
 - **System goals:** operating system should be easy to design, implement, and maintain, as well as flexible, reliable, error-free, and efficient.

Operating System Design and Implementation: Mechanisms and Policies

- Separate policy from mechanism:
 - A fixed mechanism is responsible for executing a variable policy.
 - Example:
 - A scheduling mechanism assigns CPU time to processes.
 - The scheduling policy specifies the criteria to use for scheduling, e.g.
 - Multitasking scheduling policy: switch process every 20ms.
 - Batch scheduling policy: switch process just when a process is waiting for I/O to finish.
 - Policies are likely to change (just like in politics).
 - A policy-independent mechanism may support different policies without needing to re-implement the mechanism.
 - Separation of policy from mechanism is a very important principle, it allows maximum flexibility if policy decisions are likely to change.

Operating System Design and Implementation: Implementation

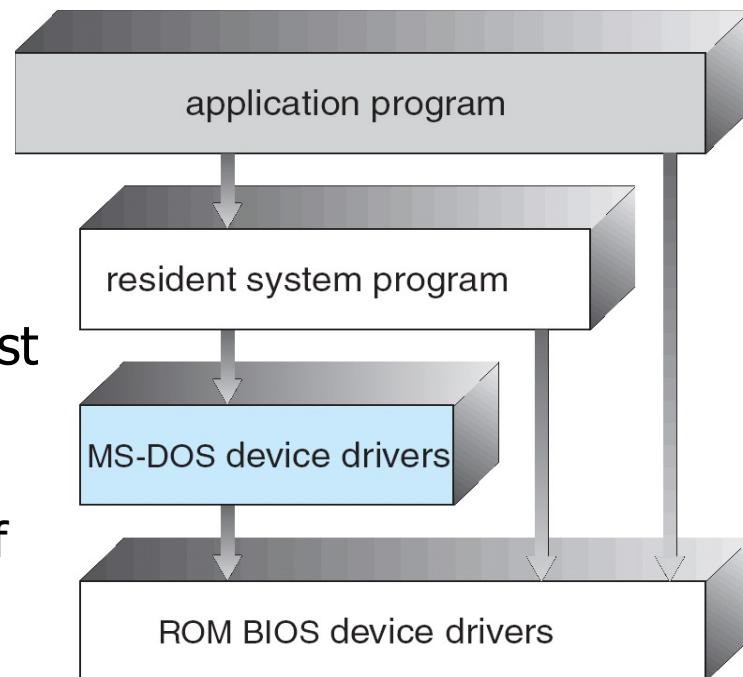
- Initially, operating systems implemented using assembly language.
- Nowadays, operating systems usually implemented using higher-level languages, in particular C.
 - Small parts using assembly language, e.g. code responsible for saving and restoring CPU registers (context switch, e.g. as part of interrupt handler).
 - Advantage of using higher-level programming languages:
 - OS is easier to port to other CPUs.
 - Disadvantage of using higher-level programming languages:
 - Reduced speed of operating system.
 - Compromise: implement time critical parts in assembly language, e.g. scheduler.

2.7 Operating System Structure

- Different design of operating systems results in operating system structures
(i.e. how components of an OS are arranged and interconnected):
 - Simple structure: **Monolithic system**,
 - **Layered system**,
 - **Microkernel system**,
 - **Module-based system**.
- (We will have a closer look at each of them on the next slides...)

Simple Structure: Monolithic System

- Operating system not divided into separated modules or layers.
 - Layers may exist, but an application may circumvent layers and thus directly access hardware, making the system vulnerable.
- Example:
 - MS-DOS: written to provide the most functionality in the least space.
 - Although MS-DOS has some structure, its interfaces and levels of functionality are not well separated.
 - Direct access to hardware possible.

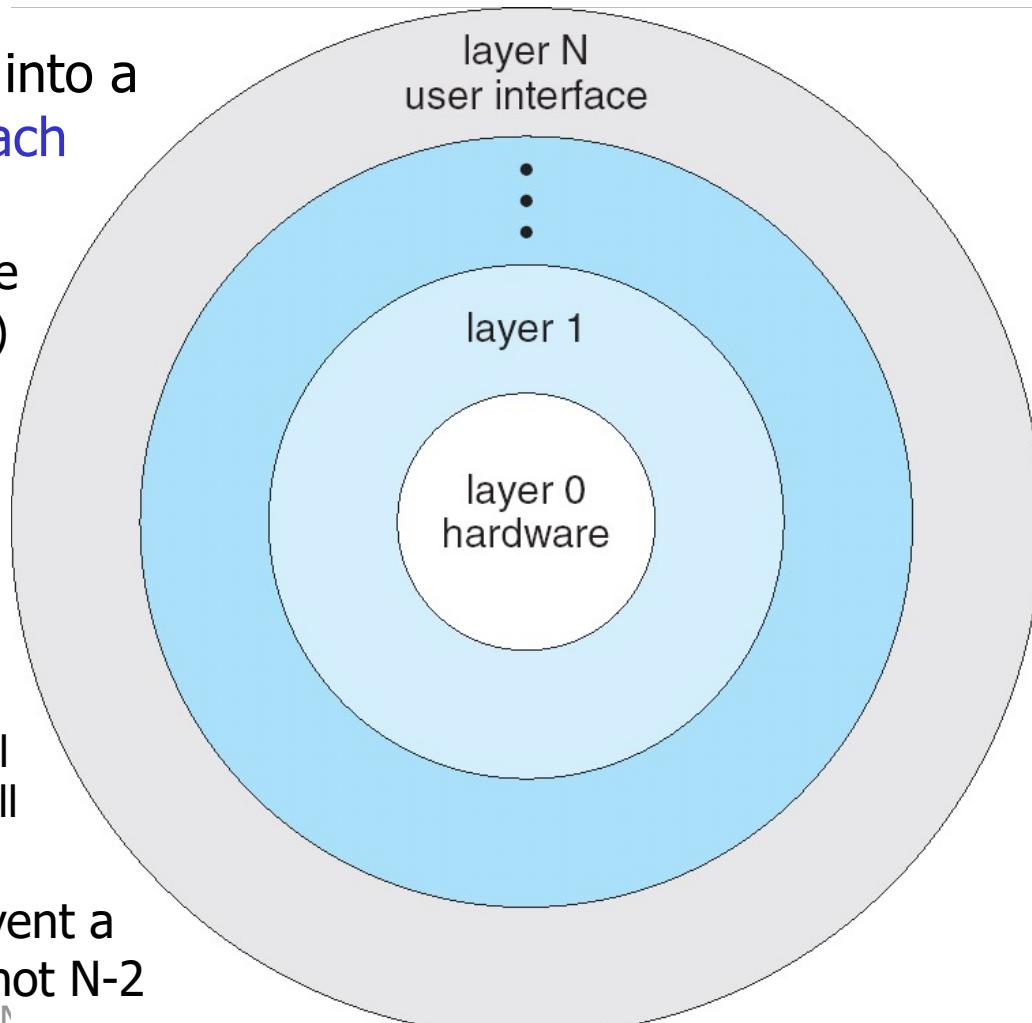


Layered Approach

- Operating system is divided into a number of layers (levels), each built on top of lower layers.

- Bottom layer (layer 0) is the hardware; highest (layer N) is the user interface.
- Layers are designed such that each layer uses functions (operations) and services of only lower-level layers.
 - E.g.: While layer 2 may call layer 1, layer 1 may not call layer 2.

- It is not possible to circumvent a layer: N can only call N-1, not N-2 (only via N-1).



Helmut

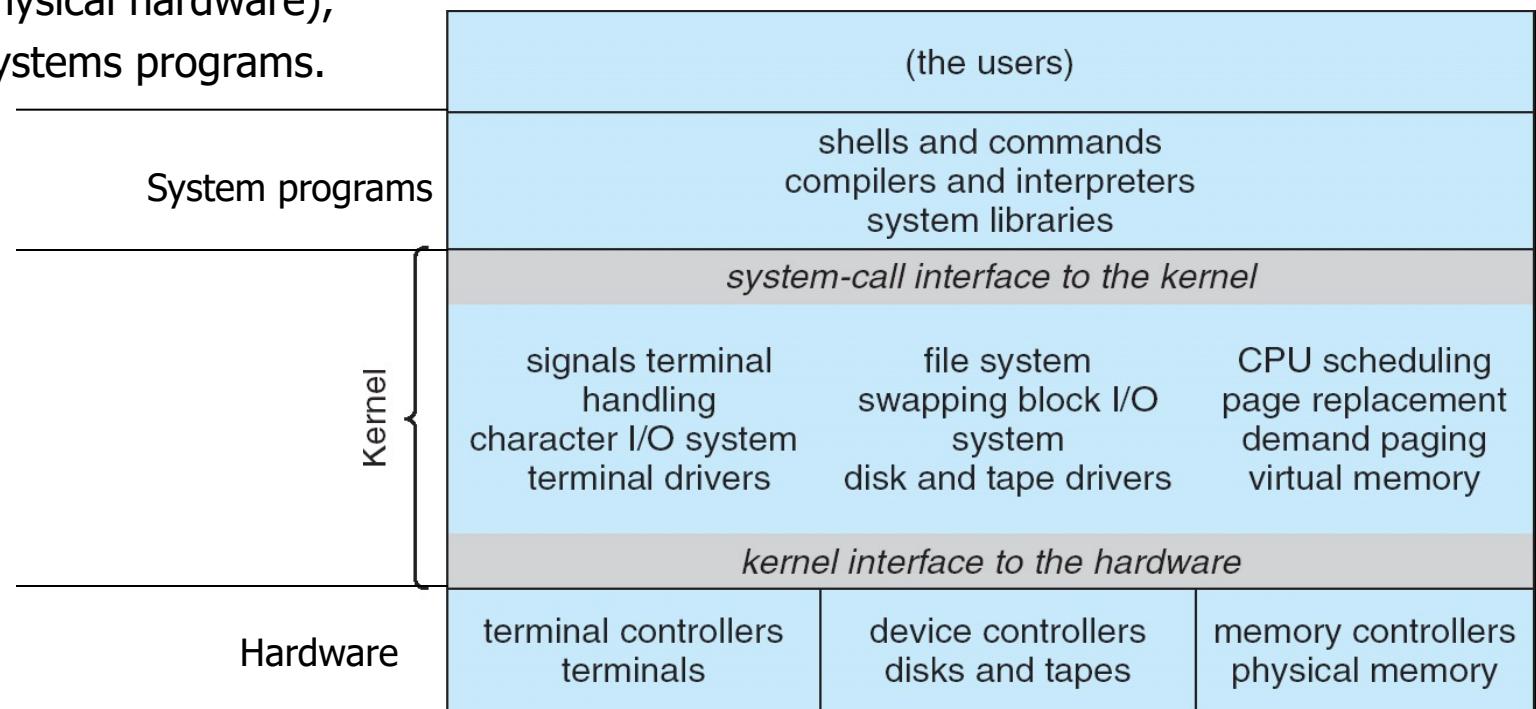
Layered Approach: Advantages and Disadvantages

- **Advantages:**
 - Layers hide complexity.
 - Development is eased.
 - Layers are easy to exchange.
 - E.g. replacing the layer directly above the hardware may be sufficient to port an operating system to another hardware.

- **Disadvantages:**
 - Good separation of layers may be difficult.
 - If component A needs to call component B and component B needs to call component A, they must reside in the same layer.
 - Low speed as each layer involves an additional function call adding some overhead.

Layered Approach: Example: UNIX

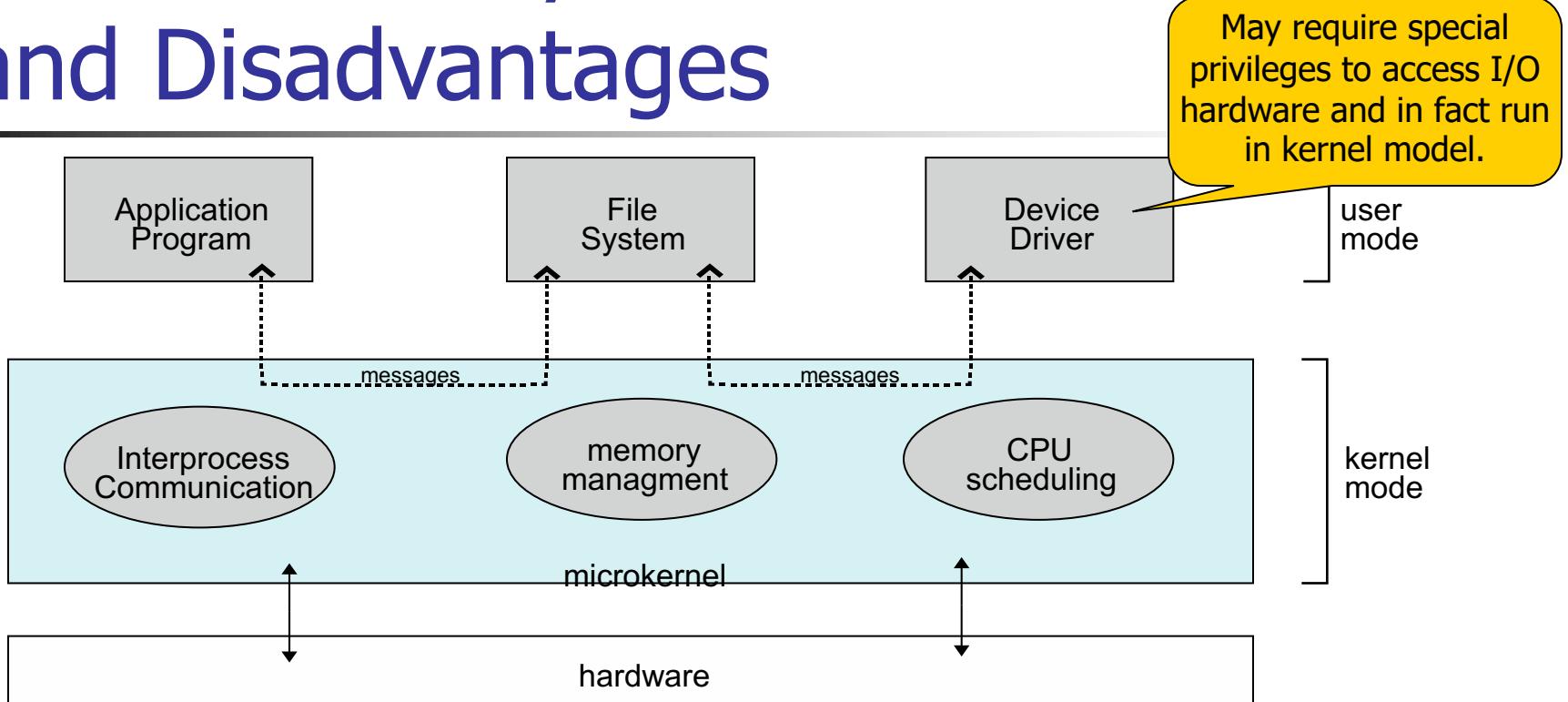
- The original UNIX operating system had limited structuring:
Only two separable layers (hence, you could consider it also as a simple structure, quite monolithic system):
 - The kernel (everything below the system-call interface and above the physical hardware),
 - Systems programs.



Microkernel System

- Moves as much from the kernel into “user” space (i.e. ordinary processes running in user mode instead of kernel mode).
 - Microkernel (running in kernel mode) consists only of minimal functionality for process and memory management and communications.
 - All additional functionality is provided by user space modules.
 - Communication takes place between user space modules using message passing (via microkernel system calls for inter-process communication).
 - E.g. instead of accessing an internal file system layer via a system call, a user process that provides file system functionality needs to be contacted by sending a message to it that requests file access.
 - File system process may in turn send a message to a device driver process.

Microkernel System: Advantages and Disadvantages



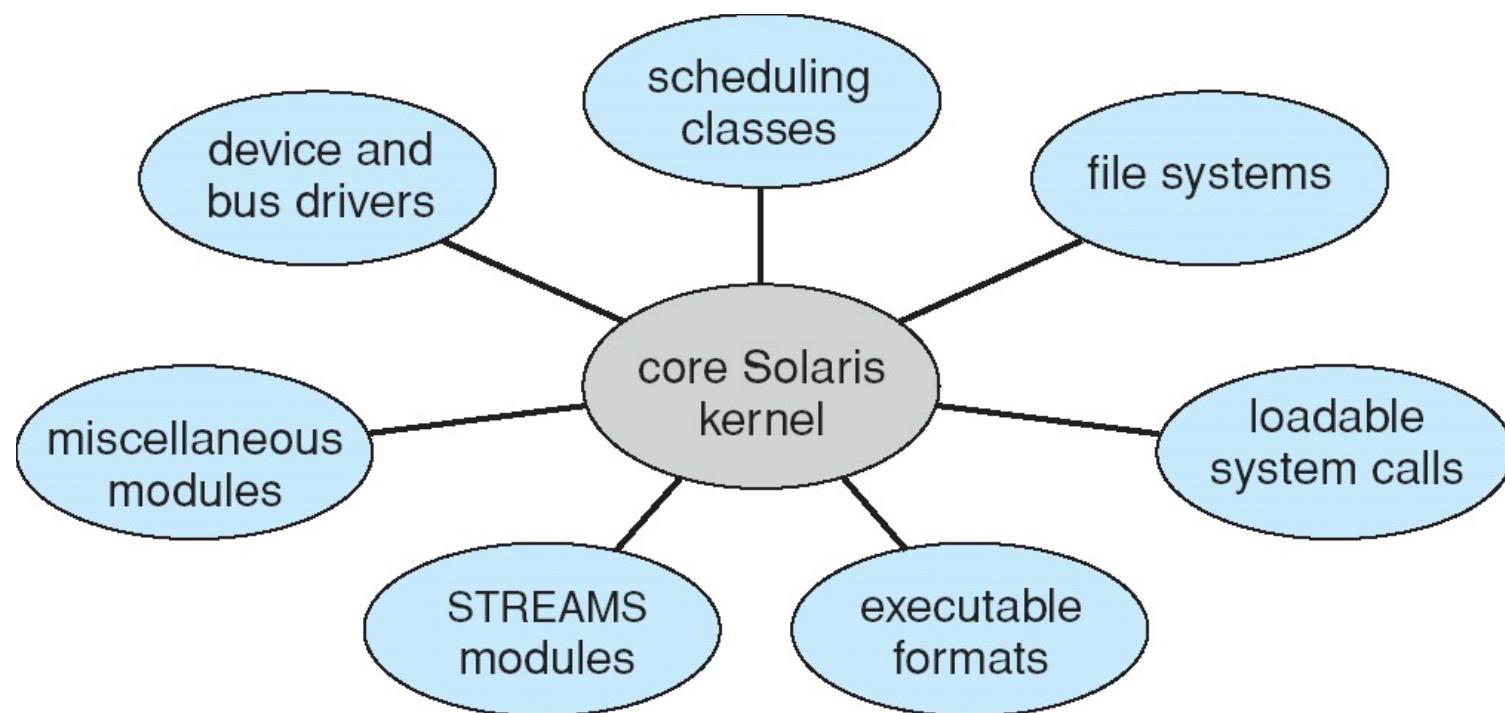
- **Advantages:**
 - Easier to extend a microkernel system: just add user space module.
 - Easier to port the operating system to new architectures: just port kernel.
 - More secure & more reliable (less code is running in kernel mode).
- **Disadvantages:**
 - Huge performance overhead due to user space to kernel space (and back) communication.

Module-based Operating System

- Most modern operating systems (e.g. Linux, Solaris, Mac OS X, even Windows) implement kernel modules:
 - Kernel consists only of core functionality.
 - Functionality can be added by loading kernel modules at run-time.
 - Similar to object-oriented approach:
 - Each class of kernel module (file system, device driver etc.) has a well defined interface.
 - Each kernel module implements this interface.
 - Mixture of Microkernel and Layers, but
 - Faster than microkernel: no message passing required for communication, instead: kernel and kernel modules run in kernel space and may use ordinary function calls to call each other.
 - More flexible than layered approach: each class of kernel module may call functions from another class of kernel module (no strict hierarchical layers), additional kernel modules may be added at run-time.

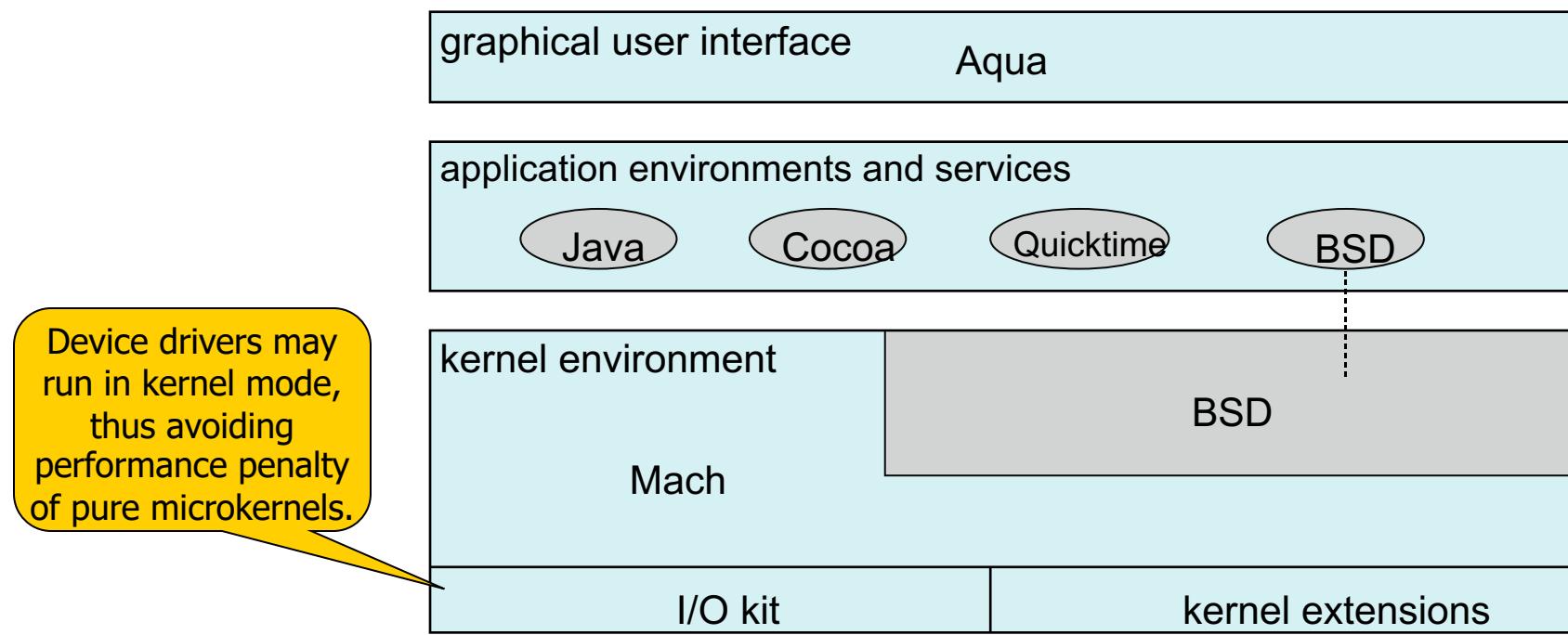
Example: Solaris Modular Approach

- 7 different classes of kernel modules. Allows at run-time to, e.g.,
 - exchange scheduler,
 - add support for devices,
 - add support for further file systems.



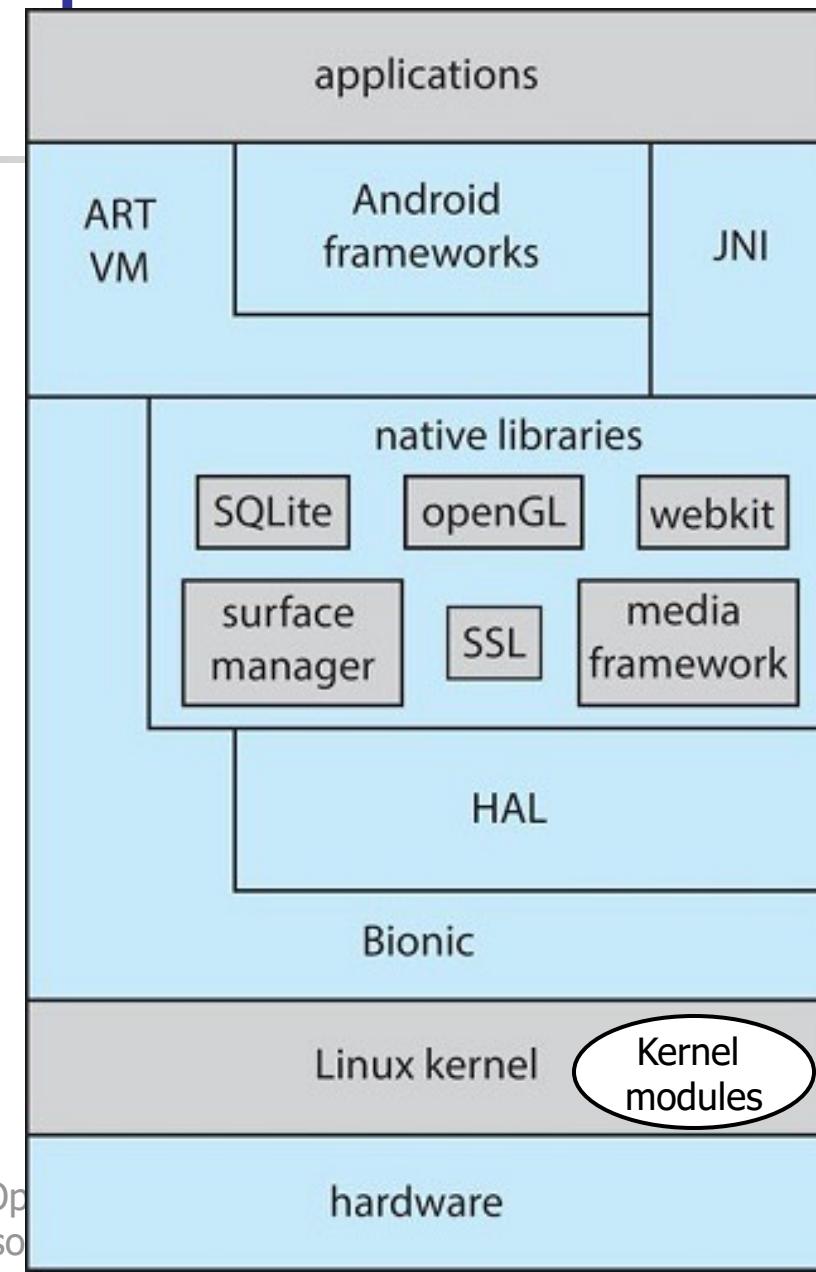
Hybrid Systems example: Apple Mac OS X (iOS is similar)

- Hybrid structure (microkernel and layered):
 - Based on [Mach microkernel](#).
 - Some common services running as user space processes.
 - Monolithic/Layered BSD kernel provides BSD Unix API on top of microkernel (to allow to use the more common BSD Unix API instead of Mach API).
 - Furthermore: support for kernel modules.



Hybrid Systems example: Android

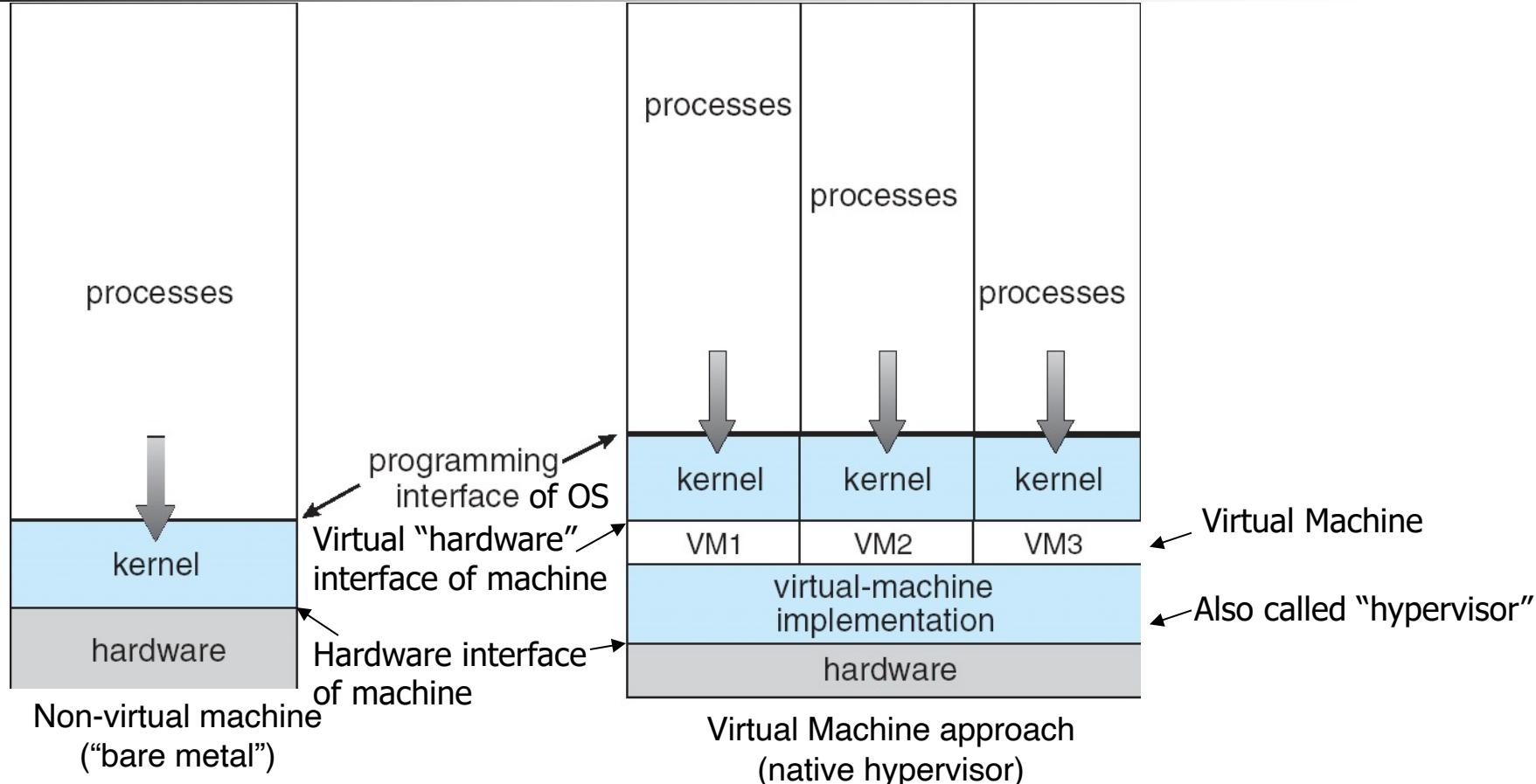
- Module-based Linux kernel with higher layers of libraries and frameworks,
- Android Runtime (ART) virtual machine interprets CPU-independent bytecode (comparable to Java).
 - Also possible to call C code of “native” libraries (=not bytecode, but C code compiled to machine code of the CPU, e.g. built-in SQLite database) via the Java Native Interface (JNI).
 - Hardware Abstraction Layer (HAL) to use hardware in an abstract way, i.e. use a camera without knowing details of it.
 - Bionic is the standard C library used by, e.g. native libraries (that are implemented in C) for doing, e.g., string or math operations that are not part of the C language itself but of a library.



2.8 Virtual Machines (VMs)

- An operating system (the kernel) runs on top of a machine:
 - CPU, memory, I/O devices, etc. (i.e. the bare hardware).
- A **virtual machine manager/monitor** (VMM) or **hypervisor** is a low-level software that provides an interface (=a virtual machine) identical to the underlying bare hardware.
 - Extreme case: a machine code interpreter simulating another CPU.
 - Software running on top of ("inside") the virtual machine has the impression of running directly on the real hardware.
- Using virtual machine technology, multiple operating systems may run in parallel on the same real machine.
 - Just like a timesharing/multi-tasking (→ch. 5) OS that shares the CPU, memory and devices between different processes, a hypervisor shares all the hardware resources between the different operating systems running inside their virtual machine.

Non-Virtual Machine vs. Virtual Machines

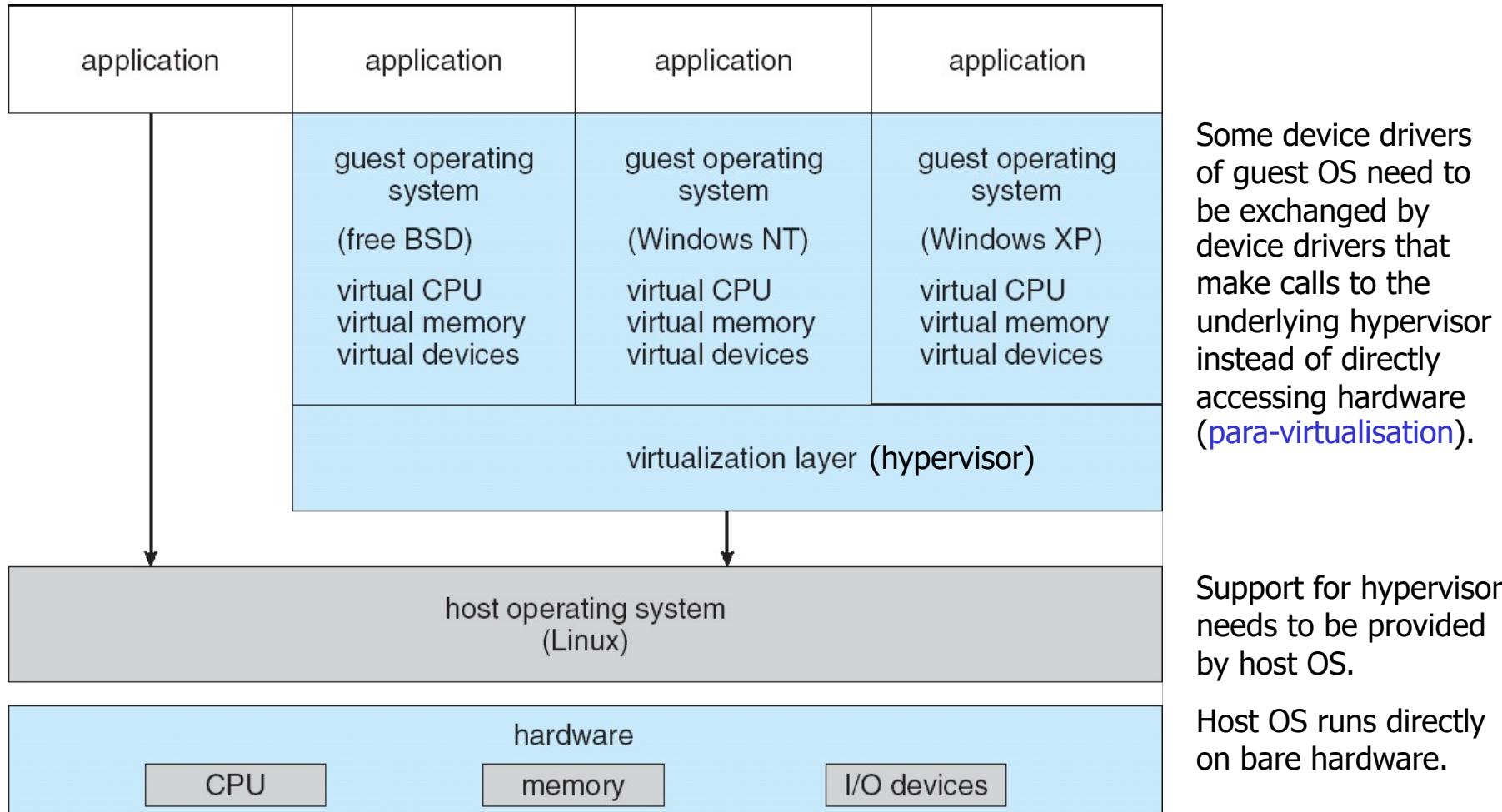


- **Native hypervisor:** Runs on top of bare hardware – all OSes run on top of it.
- **Hosted hypervisor:** Runs on top of an ordinary OS that runs on top of bare hardware.

Helmut Neukirchen: Operating

Systems/updated: I.Hjorleifsson ingolfuh@hi.is

Example: Hosted hypervisor (e.g. VMware Workstation Player)



Advantages/Disadvantages

- **Advantages:**
 - Protection between different VMs: problems (e.g. virus) in guest OS does not affect host OS (even not the hard disk: hard disks are only virtual).
 - New/different OS can easily be installed/tested in addition to existing OS and may even run in parallel to it.
 - E.g. best of Windows, Linux, Mac worlds on one machine in parallel.
 - Guest OS (and its processes) may be suspended and resumed again, e.g. for replacing hardware without reboot of guest OS or for migration to another (less busy) server. (Cloud computing is heavily based on virtualisation.)
 - Easier OS development/debugging/experiments:
 - No need to exit development environment: just start new OS in parallel on VM.
 - Debugger may be used on host OS to debug guest OS.
 - No fears that files on host hard disk get destroyed if guest OS may only access virtual disk.
 - If hypervisor implements even a complete CPU in software ("emulation"), machine code for a completely different CPU becomes executable (but slow).
- **Disadvantages:**
 - Slightly slower (rule of thumb: 80% of the native speed) (reasons: see next slides).
 - Often not all possible hardware devices supported.

Implementation of VMs: CPU virtualisation

- Hypervisor must virtualise all hardware resources:
 - CPU:
 - Hypervisor has CPU scheduler for sharing CPU time between guest operating systems.
 - Similar to CPU scheduler of timesharing/multi-tasking OS.
 - User mode/kernel mode: Guest OS must not be able to access resources directly using kernel mode. (Nevertheless, guest OS assumes that all of its interrupt handlers are running in kernel mode and thus tries to access physical devices instead of virtualised ones.)
 - VM must provide a virtual kernel mode and virtual user mode. However, even in virtual kernel mode, the guest OS is actually in physical user mode.
 - If guest OS performs an action that requires physical kernel mode, this will lead to an interrupt: interrupt handler of hypervisor is called: hypervisor (running in physical kernel mode) will analyse action and perform a corresponding action on behalf of the guest OS and finally resumes execution of guest OS instructions (in virtual kernel mode).

Modern CPUs may provide additional features to support virtualisation, e.g. additional CPU modes ("real kernel mode" for hypervisor, virtualised kernel mode for OS, user mode.)

Implementation of VMs: Memory and device virtualisation

- Hypervisor must virtualise all hardware resources:
 - Memory:
 - Guest OS must not be able to access memory of other operating systems.
 - Virtual memory gives each guest OS the impression of having exclusive access to memory, even though the memory is actually separated.
 - (Based on the same techniques that an OS uses to provide virtual memory to its application. → chapters 8 and 9)
 - Devices:
 - Guest OS must not access physical devices. (Access to devices is only possible in physical kernel mode, thus hypervisor is able to detect this → previous slide.)
 - Depending on the type of device, either a controlled access to the shared physical device is performed or a separate virtual device is simulated for each guest OS. E.g.
 - Keyboard, mouse event are forwarded to guest OS that has focus.
 - Virtualised hard disk of each guest OS is just a file on a physical file system.

Software Container, e.g. Docker

- Assume, both VM1 and VM2 run the same Linux kernel (incl. kernel modules) and system libraries (=you do not want to run different OSes):
 - Waste of disk space and RAM.
 - Store two times identical kernel, modules, libraries on two different virtual file systems, keeping them two times in RAM.
- Software container do the virtualisation at a higher layer (inside kernel):
 - Load only one kernel into memory: the kernel does the virtualisation.
 - Only disk and RAM space for a single kernel needed.
 - Kernel may even support to share identical files (such as common libraries) as well as having own file versions in each container.
 - No need to care about: application 1 runs only with library version 1, application 2 only with library version 2 ⇒ Instead, each application has its own container with own libraries and other software it depends on (=eases installation).
 - No overhead for virtualising hardware accesses.:
 - The single kernel runs already on the bare hardware.
 - Kernel needs however to be able to separate container from each other.
 - E.g. processes in Container1 should not be able to modify files from Container 2.

2.9 Java

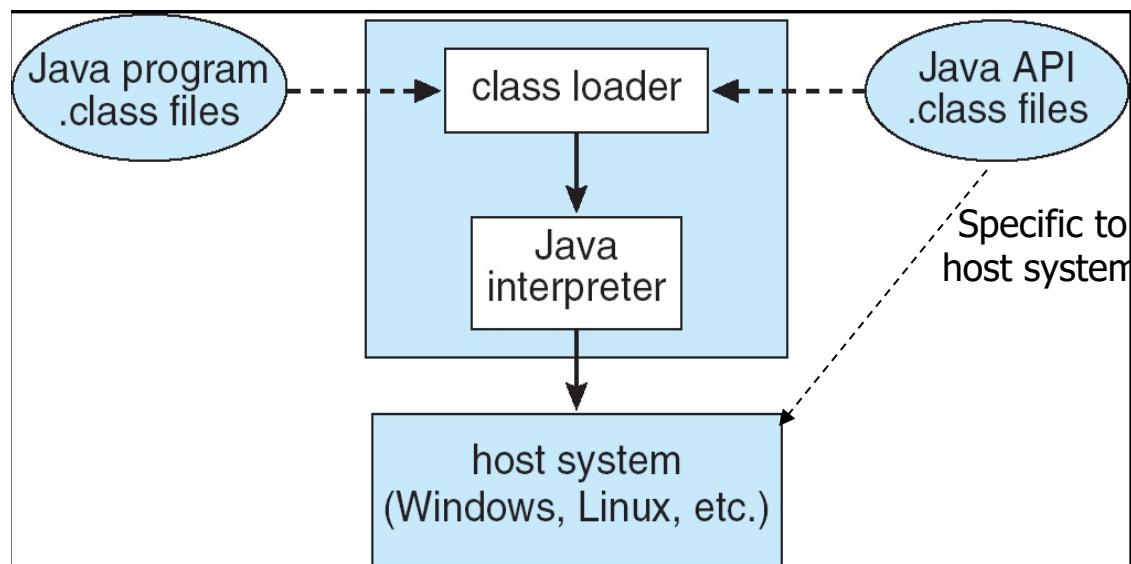
- Not immediately related to operating systems (or at least to OS kernels), but an interesting application of virtual machines:

Java technology:

- Java programming language,
 - Is compiled into bytecode (.class files).
- Java libraries (API),
 - Provides access to high-level services and operating system-level services via an operating-system independent API.
 - Implementation of that interface (i.e. the Java library), however, is operating-system dependent: OS-level services of Java API need to be mapped on system calls of respective OS.
- Java virtual machine.
 - Is able to execute bytecode (more on next slide).

Java Virtual Machine

- Java virtual machine (JVM):
 - Does not run directly on hardware, but on top of OS.
 - Available for various platforms (i.e. operating systems and CPU types).
 - Virtual machine concept (together with OS-specific Java API implementation) allows to execute bytecode without modification on all supported platforms:
 - Interpreter for bytecode ("emulation" of Java CPU) (or just-in-time compiler).



2.12 System Boot

- How is an OS started booted?
 - (“Booting” = starting a computer by loading the kernel.)
- When power initialized on system, CPU starts execution at a fixed memory location (location hard-coded into CPU).
 - CPU runs in kernel mode after power-on.
- At that memory location, a computer hardware has a ROM or EPROM that contains machine code instructions, generally known as **firmware**.
 - In older PCs: BIOS (Basic Input Output System)
 - More recently: UEFI (Unified Extensible Firmware Interface)
 - + “Compatibility Support Module” (CSM) for compatibility for programs expecting a BIOS.

System Boot: Bootstrap Program/Boot Loader

- Firmware contains **bootstrap program / bootstrap loader**:
 - Just enough functionality to locate the kernel on mass storage device, loading it into memory, and starting it.
 - Often, a two-step process where the fixed firmware bootstrap program just loads a small **boot block** from fixed location of mass storage device and then starts the **boot code** contained in boot block.
 - BIOS was only able to load boot block from one of the firsts blocks of a storage.
 - UEFI is smart enough to locate, load and start bootstrap code everywhere.
 - Contents of boot block can be easily changed (located on mass storage, not in firmware).
 - E.g. GRUB (GRand Unified Bootloader) on Linux systems
 - As CPU runs in **kernel mode**, **boot code** can access hardware.

System Boot: Kernel

- Boot code in boot block then loads kernel from a storage device and hands over control to it.
- The first instruction of the operating system kernel is started (=the OS is now running)
 - Kernel initialises hardware.
 - (in addition to basic initialisation done by BIOS/UEFI.)
 - A root filesystem (=the filesystem that contains most important system programs) is mounted, i.e. OS can access it.
 - Kernel starts system processes (e.g. user interface for logging in) and background processes (Unix: “daemons”/ Microsoft: “service”) by loading them from the root filesystem.

2.13 Summary

- OS provides number of service to user/application and itself.
 - Services to user accessible via user interface.
 - May be used to execute system programs.
 - Services to applications accessible via system calls.
 - Instead of using the API offered by the OS, often a programming language-specific API (e.g. C standard lib) is used.
 - Application remains portable as long as programming language-specific API is available on other system. However, for OS specific services, OS API needs to be called.
- Several types of OS designs: today, module-based operating systems are dominant.
 - Supporting the exchange of policies keeps OS mechanisms flexible.
- While virtual machines are usually slightly slower, they have many advantages, e.g. running multiple OS in parallel.
- Different stages of booting involve progressively smarter programs.
 - From firmware (in the past via boot block) to kernel.

Course TÖL401G: Stýrikerfi / Operating Systems 3. Processes

Mainly based on slides and figures subject of
copyright by Silberschatz, Galvin and Gagne

Chapter Objectives

- Identify the separate components of a process and illustrate how they are represented and scheduled in an operating system.
- Describe how processes are created and terminated in an operating system, including developing programs using POSIX system calls that perform these operations.
- Describe and contrast interprocess communication using shared memory and message passing.
- Describe programs that use POSIX pipes and POSIX shared memory to perform interprocess communication.
- Describe client–server communication using sockets and including how to create client/server programs using the Java socket API.

Contents

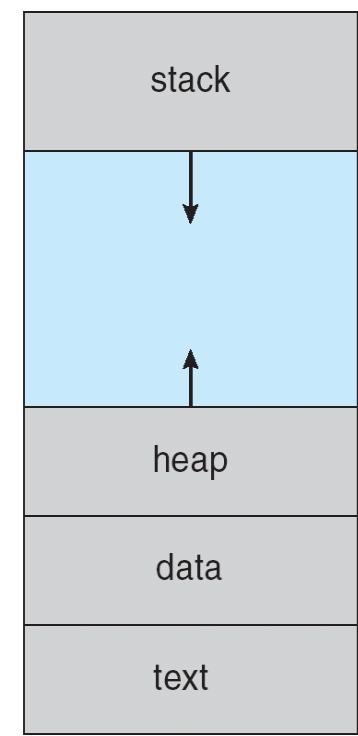
-
1. Process Concept
 2. Process Scheduling
 3. Operations on Processes
 4. Interprocess Communication
 5. Communication in Client-Server Systems
 6. Summary

For users of any edition of Silberschatz et al.:
The section on Interprocess communication differs
slightly from the book and contains differing material.

3.1 Process Concept

- Process: a program in execution.
 - A program is a passive entity:
 - E.g. a list of instructions stored on a hard disk.
 - A process is an active entity:
 - List of **instructions** loaded into main memory (**text section**),
 - **Program counter** (PC) specifying next instruction to execute,
 - **Stack** (for calling sub-routines or storing local variables),
 - **Data section** (for global variables)
 - Set of **further associated resources**, e.g.
 - Heap (memory dynamically allocated during process run time),
 - Opened files, ...
- Synonyms for process:
 - Batch system world: **job**,
 - Time-shared systems: **task**.

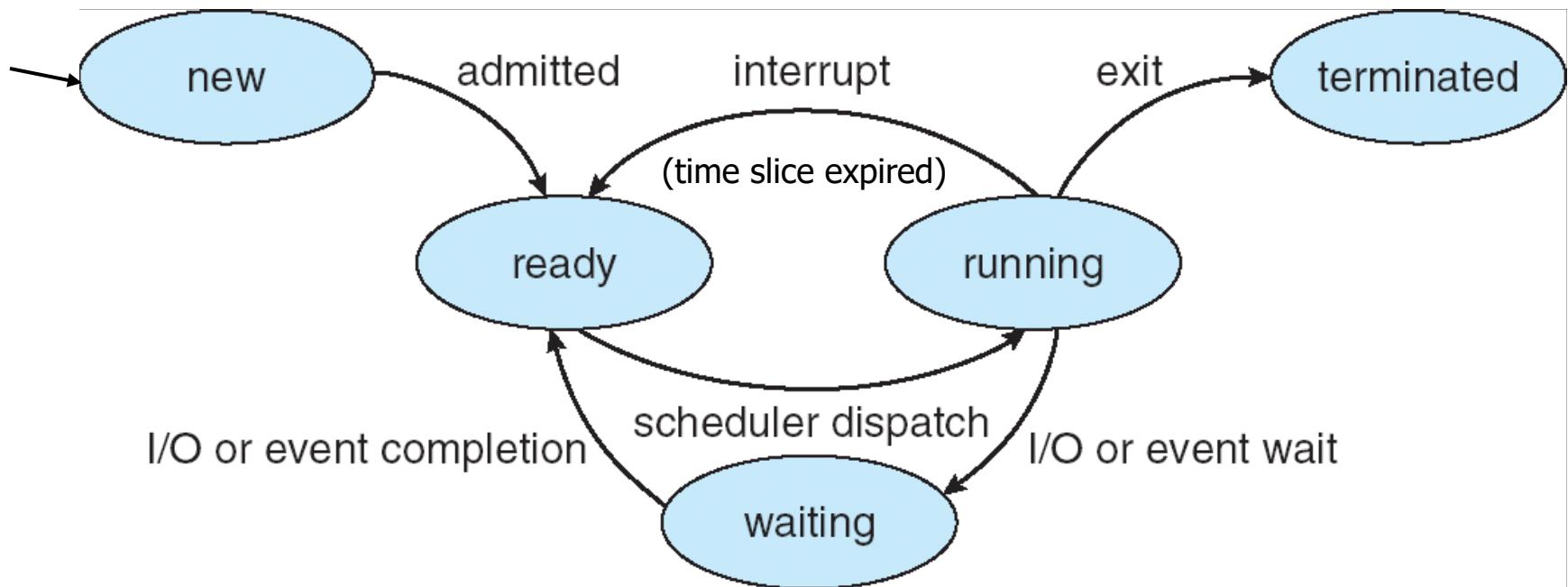
Address space
of a process:



Process State

- As a process executes, it changes its **state**:
 - **New**: The process is being created.
 - **Running**: Instructions are being executed.
 - On a single processor system, at most one process may be in state running. (On multi processor/multi core systems, each processor/core may execute a process.)
 - **Waiting**: The process cannot execute instructions, because it is waiting for some event to occur.
 - E.g. waiting for I/O completion.
 - **Ready**: The process is waiting to be assigned to a processor.
 - **Terminated**: The process has finished execution.
 - Note: depending on the OS, the name of states may vary (e.g. "blocked" instead of "waiting") or further more detailed states may exist.

Process State Transition Diagram

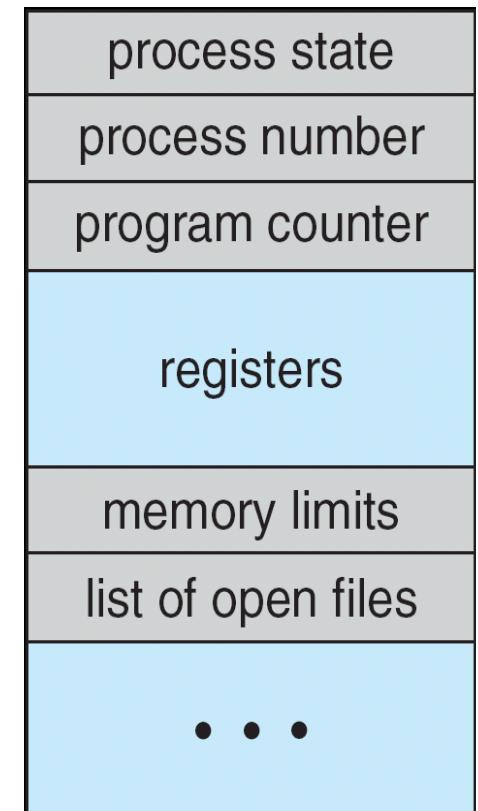


- State changes from running to ready, if time slice expires.
 - I.e. interrupted by timer of scheduler in order to give CPU time to another process and therefore interrupts current running process.
 - Each process gets a “slice” of the CPU time (=time sharing →3.2).

Process Control Block (PCB)

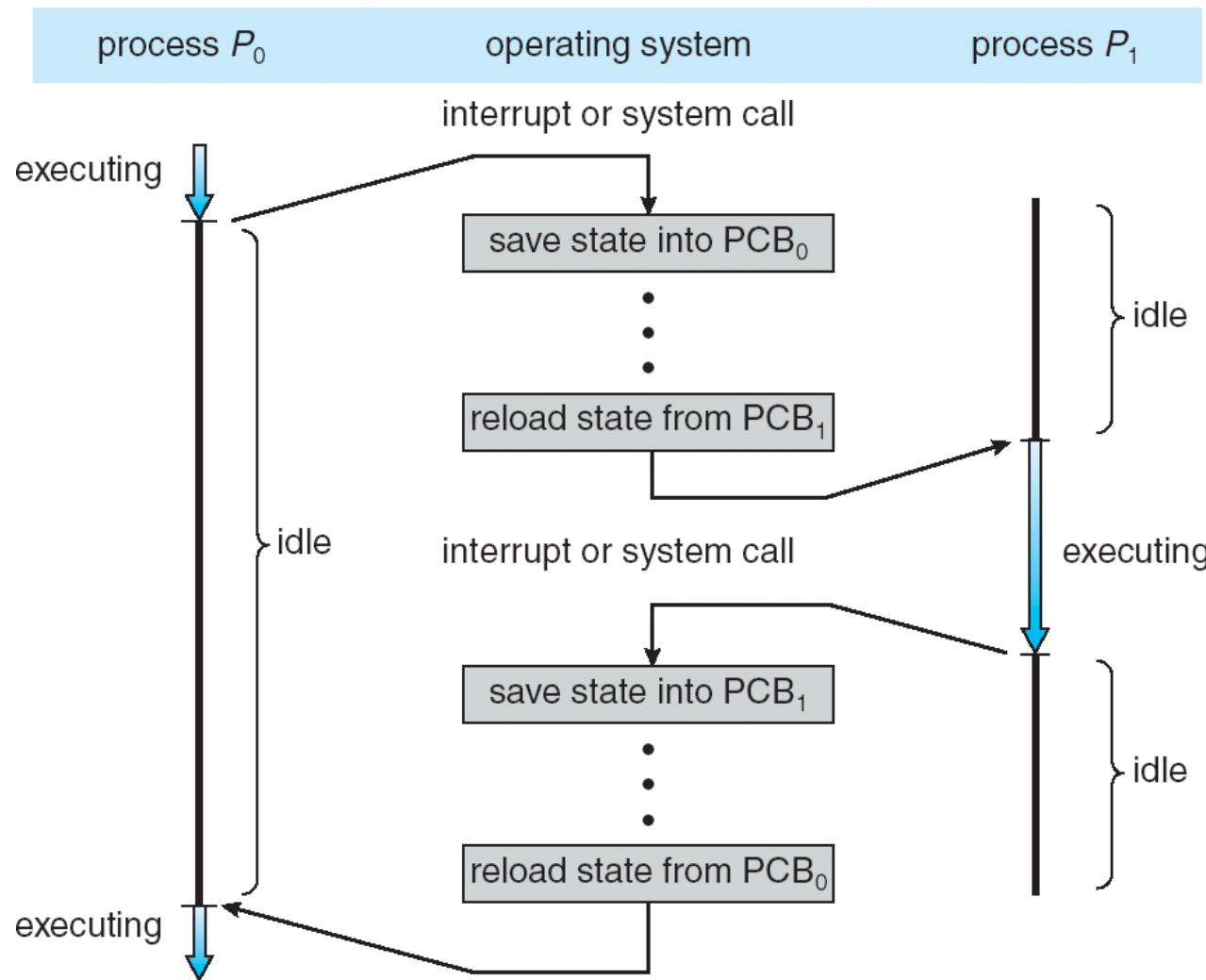
- State of process (and further information) stored by OS in **process control block (PCB)**:

- Process state,
- Process number/Process Id,
- Program counter and all other CPU registers,
 - Saved when interrupt occurs,
restored when execution continues.
- CPU scheduling information,
 - Process priority, pointer to scheduling queues, ...
- Memory-management information,
 - Allocated memory, address space (memory limits/
memory mapping) of process, ...
- Accounting information,
 - Used CPU time, ...
- I/O status information.
 - I/O devices reserved by process, opened files, ...



- Program counter, CPU registers, and memory management information are called **hardware context** of a process.

CPU Switch From Process to Process (Context Switch)



Note: here, “save/reload state” refers not to the process state but to the hardware context.
(The process state changes of course as well as part of context switch, e.g. from “running” to “ready” and vice versa.)

d

3.2 Process Scheduling

- Process scheduler is responsible for switching between processes:
 - Multiprogramming (batch system, e.g. High-Performance Computing/Supercomputing):
 - Goal: maximise CPU (& other resource's) utilisation.
 - Switch to another process as soon as a process is waiting.
 - Time sharing (multi tasking system):
 - Goal: allow interaction between users and processes.
 - Switch frequently between processes (interrupt process even if it is not waiting). Each process gets a time slice of the CPU.
 - (More on scheduling algorithms in chapter 5...)

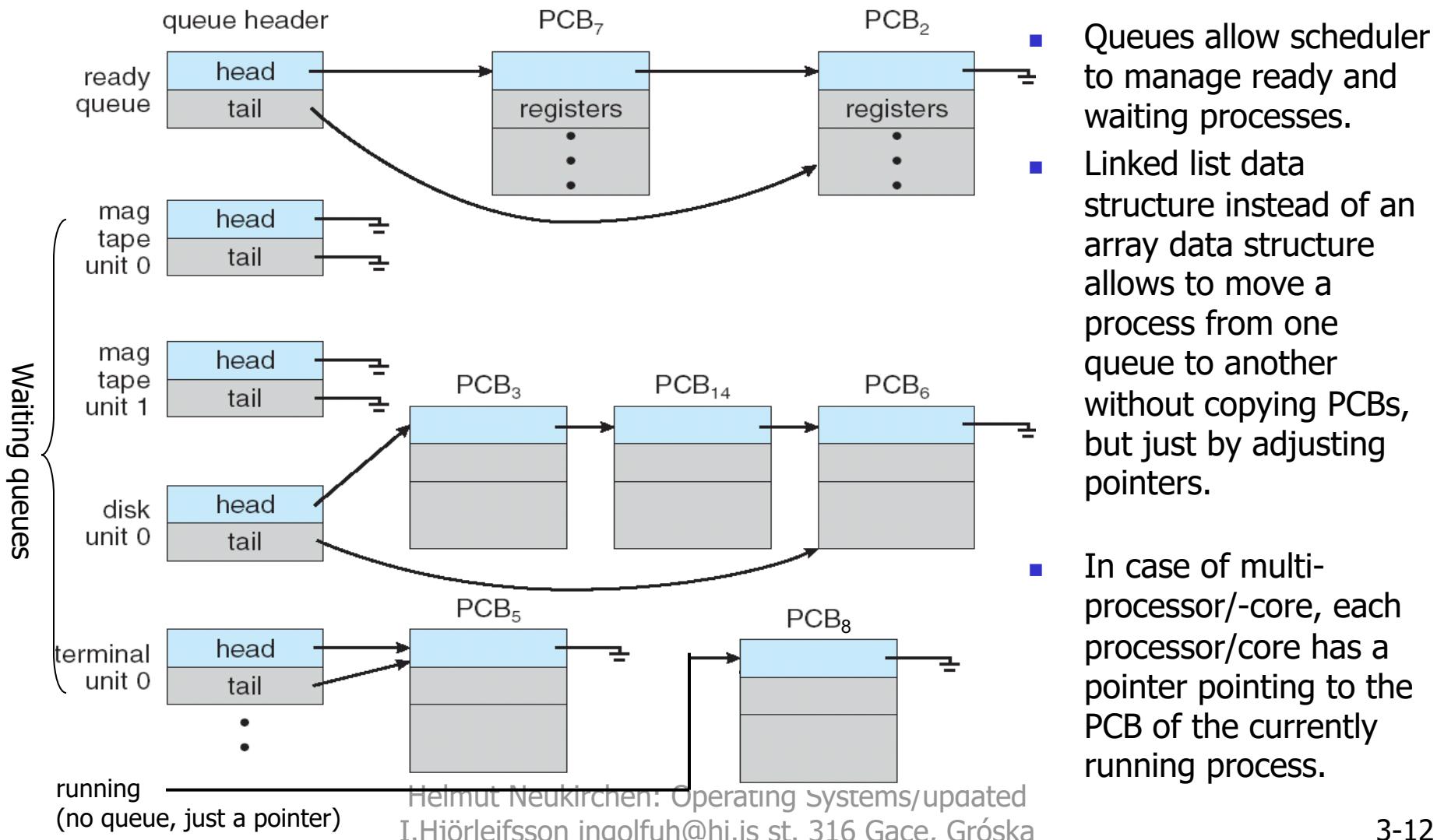
Scheduler Activations

- Process scheduler is activated when:
 1. Process gets blocked because it is waiting for some event, e.g. completion of I/O:
 - Transition from running to waiting state.
 2. Time slices expires, i.e. timer interrupt occurred (in case of timesharing system, i.e. preemptive):
 - Transition from running to ready state.
 3. I/O completes, i.e. interrupt of I/O device signal completion (preemptive):
 - Transition from waiting to ready.
 4. Process terminates (e.g. makes `exit()` system call or abnormal termination) (`exit()` is nonpreemptive, but abnormal termination may be preemptive):
 - Transition from running to terminated state/removal of process from system.

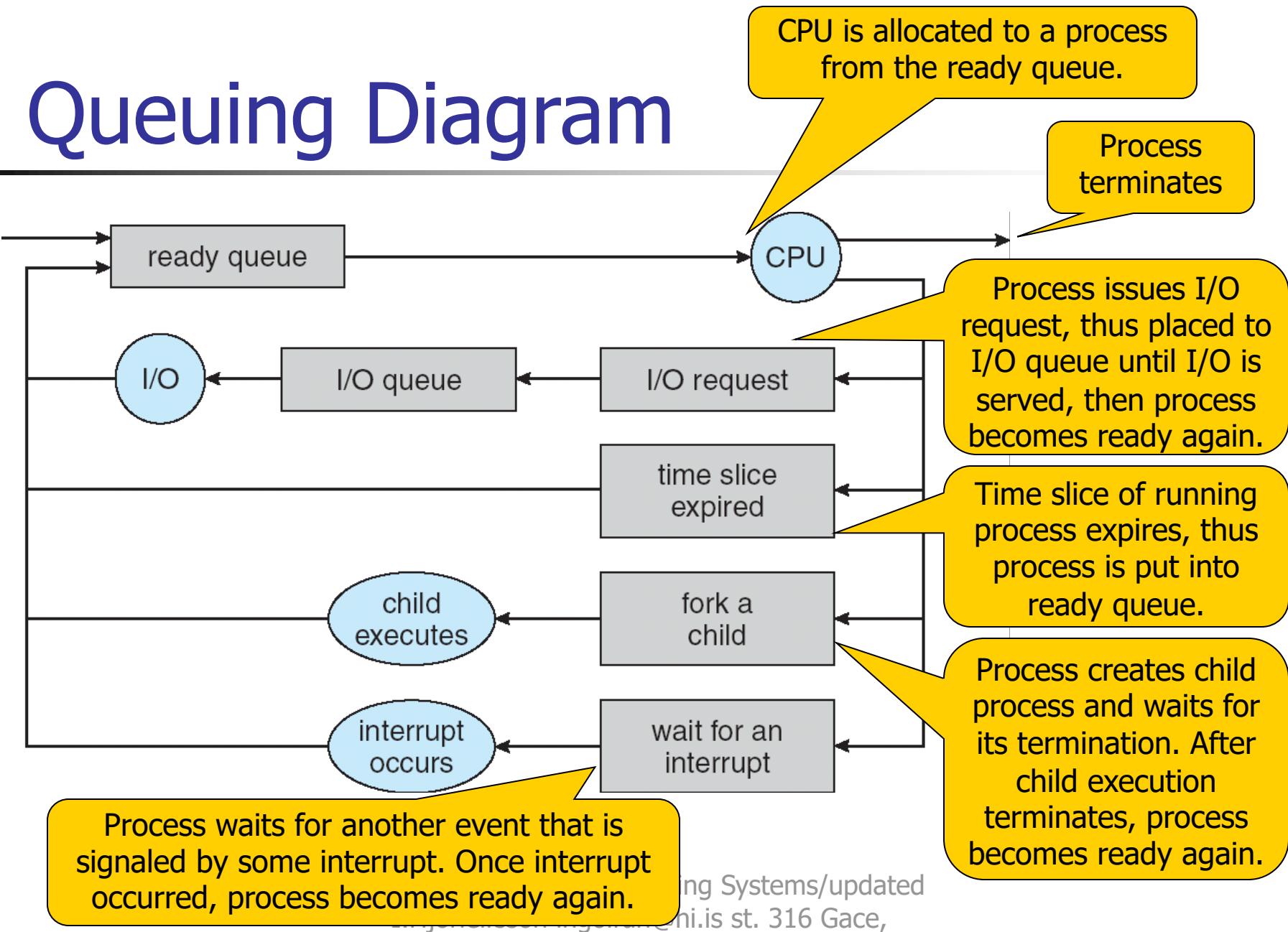
Process Scheduling Queues

- When switching to another process, the process scheduler selects (dispatches) a ready process that will be executed by the CPU.
 - Scheduler needs to be able to find all ready processes.
- Scheduler maintains several queues for an efficient scheduling:
 - Job queue: set of all processes in the system.
 - As soon as a process enters the system it is added to job queue.
 - Ready queue: set of all processes in main memory, ready to execute.
 - Device queues: set of processes waiting for an I/O device.
 - Several processes may be waiting for the same I/O device:
one queue for each device.
 - Processes migrate among the various queues depending on whether they are ready or waiting.
- Queues are implemented as a linked list data structure:
 - Queue header is pointing to PCB of first (and final) process in the queue.
 - Each PCB contains a pointer to the next process in the particular queue.
 - (See next slide for example.)

Ready Queue, Various I/O Device Queues, Pointer to Running Process



Queuing Diagram



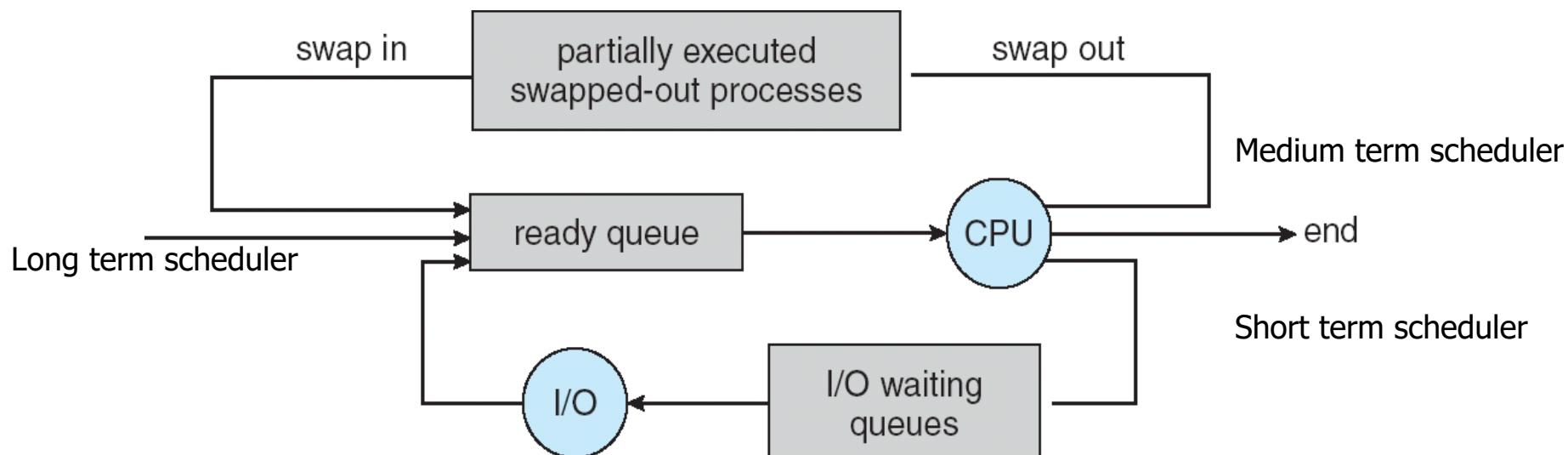
Process waits for another event that is signaled by some interrupt. Once interrupt occurred, process becomes ready again.

Batch System Schedulers

- In multi tasking systems, all processes started by a user are kept in memory at the same time.
 - Usually only one scheduler (the CPU scheduler).
- However, in batch systems only a selected number (=**degree of multiprogramming**) of the submitted batch jobs are loaded into memory and started as process. (Once a process terminates, a new batch job is loaded into memory and started.) Two schedulers:
 - **Long-term scheduler** (or **job scheduler**): selects which processes from the submitted batch jobs should be loaded into memory (and thus brought into the ready queue).
 - **Short-term scheduler** (or **CPU scheduler**): selects which process from the ready queue should be executed next and get the CPU.

Medium Term Scheduling/Swapping

- Sometimes, degree of multiprogramming needs to be temporarily reduced (e.g. if processes require a lot of memory).
 - A **medium term scheduler** may be used to identify processes that should not be considered by short-term scheduler for some time:
 - Processes are removed from main memory to hard disk (swapped out) and later-on, e.g. if another process terminates, the swapped-out state is put into main memory again (swapped in).



Process Schedulers

- Short-term scheduler is invoked very frequently (milliseconds) ⇒ must be fast.
- Long-term scheduler is invoked very infrequently (seconds, minutes) ⇒ may be slow.
- The long-term (/medium term) scheduler controls the degree of multi-programming (how many programs are executed at the same time).
- Processes can be described as either:
 - I/O-bound process: spends more time doing I/O than computations, many short CPU bursts.
 - CPU-bound process: spends more time doing computations; few, but very long CPU bursts.
 - Long-term scheduler should try to achieve a good process mix of I/O and CPU-bound processes to utilise both CPU and I/O devices.
 - E.g.: submitters of batch jobs need to classify their jobs as either I/O- or CPU-bound

Context Switch

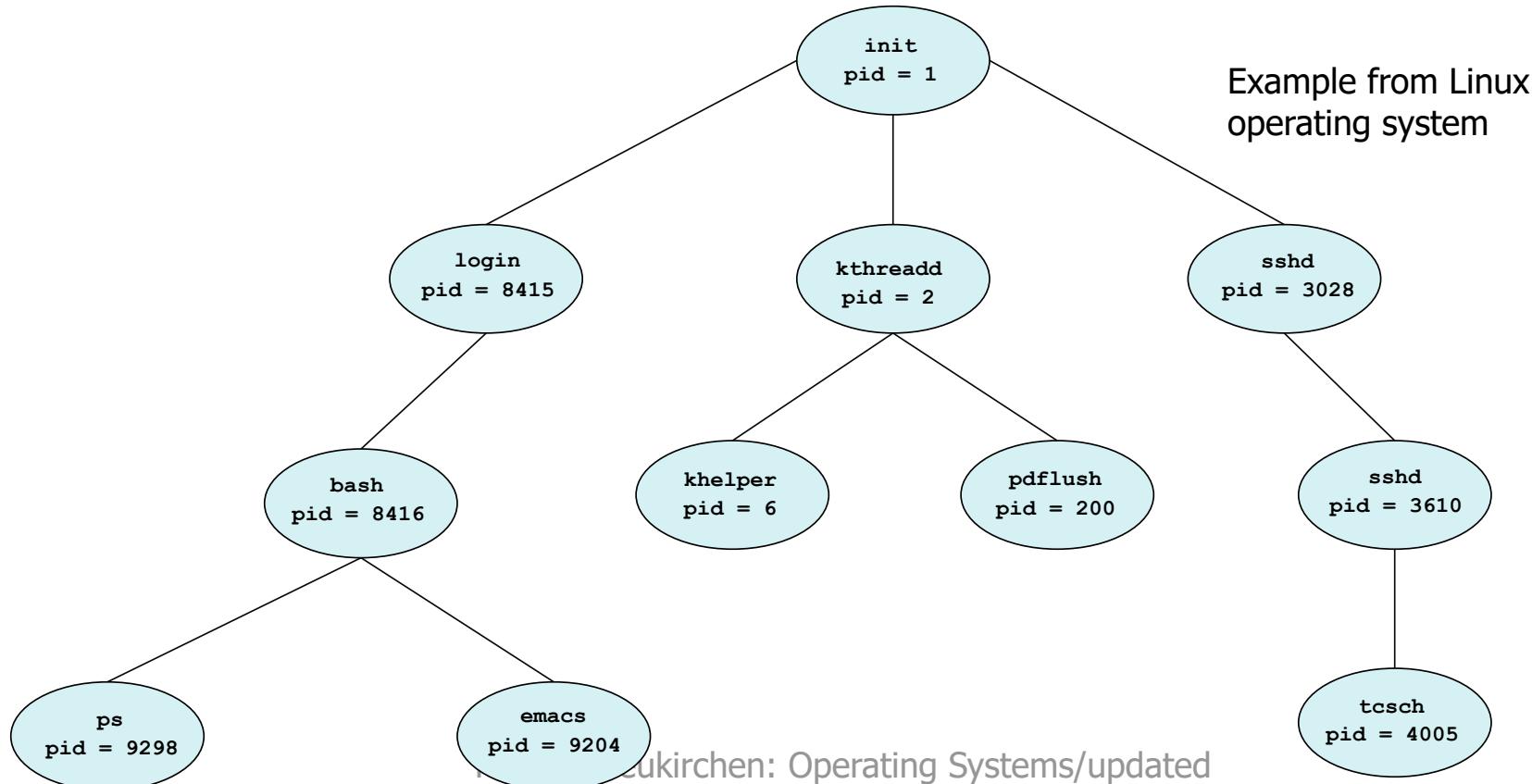
- When CPU switches to another process, the system must save the (hardware) state of the old process (also: interrupt processing) and load the saved (hardware) state for the new process: **Context switch**.
 - Status of CPU (program counter and all other CPU registers),
 - Status of memory controller hardware (memory limits/memory mapping).
 - (Other information from the PCB usually needs not to be saved/restored as part of context switch, because it refer to states where the PCB is anyway the primary location for maintaining and storing them, e.g. process Id.)
- Time needed for **context-switch is overhead**.
 - Time slices between 10ms and 100ms mean between 100 and 10 context switches per second.
 - Usually, a context switch takes between $3\mu s$ ($=0.003ms$) and 1ms (depending on scenarios, OS and hardware: nowadays typically $30\mu s$).
 - <http://blog.tsunanet.net/2010/11/how-long-does-it-take-to-make-context.html>
 - Overhead may be reduced by using threads (switches only CPU status, but not memory controller hardware status).>Chapter 4

3.3 Operations on Processes

- Operating system must include support to create and terminate processes dynamically.
- To identify a process, each process has a **process identifier (PId)**.
 - PId is just an integer number.
 - Usually, the first process created by the operating system ("init" process) has PId 1, all further processes get increasing PIds.
- Usually, there is a parent-child relationship between processes.

Process Creation: Tree of processes

- Parent process creates children processes, which, in turn create other processes, forming a tree of processes.

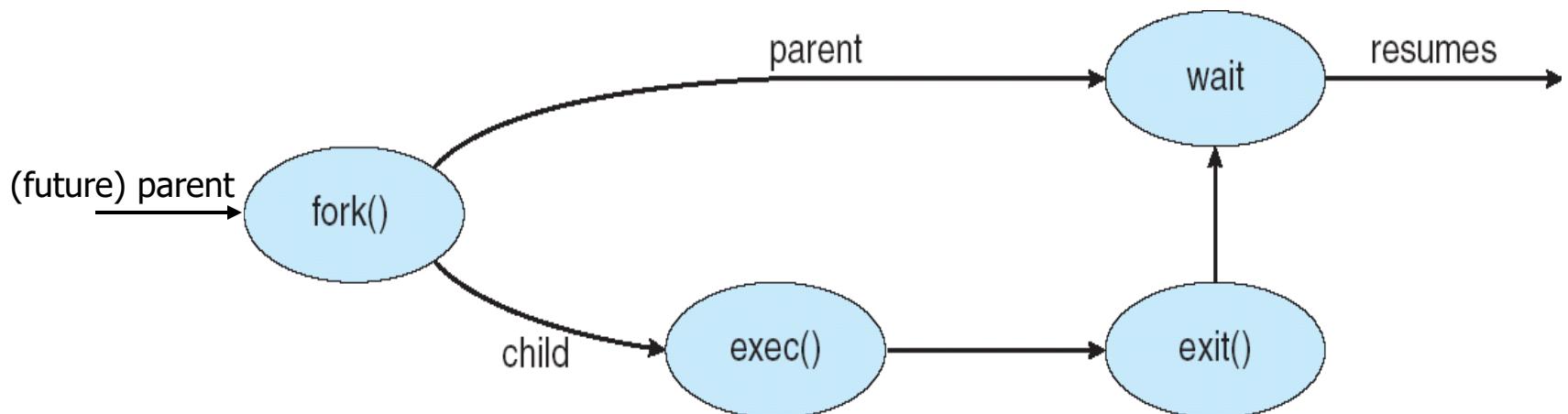


Process Creation: Design decisions

- Several possibilities concerning sharing of resources, execution, and address space:
- Resource sharing:
 - Parent and children share all resources or
 - Children share subset of parent's resources or
 - Parent and child share no resources.
- Execution:
 - Parent and children execute concurrently or
 - Parent waits until children terminate.
- Address space:
 - Child address space (instructions, data, heap, stack) duplicate of parent or
 - Child has a new program loaded into it.

Process Creation: POSIX

- Approach (chosen design decision) depends on operating system.
- POSIX example:
 - `fork` system call creates new process as a copy of parent process.
 - `exec` system call used after a fork to replace the process' memory space with a new program.

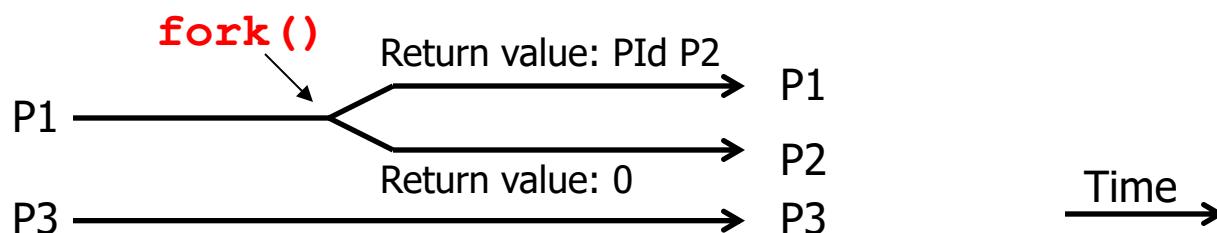


Process Termination: POSIX

- Process executes last statement and voluntarily requests from operating system to be deleted (**exit**).
 - Return data from child to parent (return value provided by **exit** system call – may be retrieved by parent via **wait**).
 - Process' resources are deallocated by operating system.
- Parent may terminate execution of children processes (**kill**), e.g. because
 - child has exceeded allocated resources,
 - task assigned to child is no longer required,
 - parent is exiting.
 - Some operating system do not allow child to continue if its parent terminates. (However, POSIX does.)
 - In that case, all children would be automatically terminated if parent terminates (cascading termination).

POSIX System Calls for Process Creation in Detail (1)

- POSIX `fork()` system call for creating a new process:
 - New process (**child**) is an exact copy of the creating process (**parent**), but has a new PId.
 - Child and Parent execute the same instructions after the `fork()`.
 - Return value of `fork()` system call differs for parent and child: based on this, a process can identify whether it is the child (return value: 0) or the parent (return value: PId of Child).

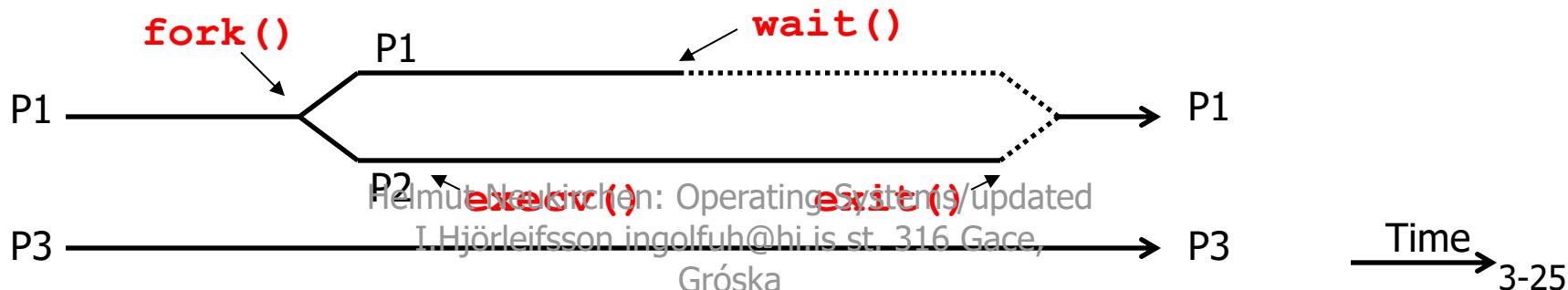


POSIX System Calls for Process Creation in Detail (2)

- “Exact copy” of processes when `forking` means:
 - Copy of instructions (text section), `data`, `stack`, `heap`.
 - Partial copy of PCB, e.g.:
 - CPU registers,
 - Opened files.
 - E.g. for communication between parent and child using pipe files (\rightarrow 3.5).
 - Other parts of PCB are not copied but `adjusted`, e.g.:
 - New Process Identifier (`PId`) for child,
 - `Address space` (child process is a copy: address space is copied into another part of the RAM)
- If it is not intended that child process executes after a `fork()` the same instructions as the parent:
 - (Child-)Process may use one of the `exec()` system calls (e.g. `execv()`) to replace its instructions and data with those from a programme file.
 - E.g. used by a shell to start other programs.

POSIX System Calls for Process Termination in Detail (1)

- Possible process terminations:
 - Normal termination using `exit()` system call (intentionally by client),
 - Termination to indicate an error using `exit()` system call (intentionally by client),
 - Termination due to a severe process error, e.g. division by 0 (by operating system),
 - Termination using `kill()` system call (by another process).
- Parent may wait for the termination of child process using `wait()` system call.
 - The status code provided by child process when calling `exit()` is passed back to parent as return value: this status code is stored in a table of processes.
 - If parent has not yet called `wait()`, it might do so in future and process table entry for child (storing the status code) needs to be kept: child process is during that time in process state `zombie` until `wait()` is called.
 - If parent process terminates without calling `wait()`, the init process (PId=1) becomes parent of that child. (Init calls anyway periodically `wait()` thus removing any zombies.)



POSIX System Calls for Process Termination in Detail (2)

- Usage of `kill()` system call in detail:
 - `kill (pid, SIGKILL)`
sends signal `SIGKILL` to process `pid`, thus terminating it.
 - Signals send via `kill()` do not necessarily kill a process.
E.g.:
 - `SIGSTOP` puts a process to sleep,
 - `SIGCONT` awakes process again.
- Operating system considers protection:
 - A process of user **A** is not allowed to kill process of user **B**.

Example of using POSIX system calls: fork(), exit(), execv(), wait()

- C program using fork(), etc.
 - (Java does only support creating threads, but not forking processes.)

```
#include <unistd.h>
#include <stdlib.h>
#include <sys/wait.h>

int err, status;

if (fork() == 0) {
    /* Child process */
    err = execv("/bin/program", NULL);
    exit(err);
}
else {
    /* Parent process */
    wait(&status);
    ...
}
```

Create new process

Replace instruction
and data of process
with those of
program in file
/bin/program

This line of code should never
get executed, except that
execv() fails, e.g. because
file does not exist.

Wait for child to terminate. **status** will contain the
parameter passed by child to **exit()**.

Multiprocess applications: Web browser example

- In the past, many web browsers ran as single process:
 - If one of the web sites (in a tab) causes trouble, entire browser can hang or crash.
- Nowadays, Web browsers are multiprocess, e.g. Google Chrome, with three different types of processes:
 - One main **Browser** process manages user interface, disk and network I/O, and creates/coordinates the other processes.
 - **For each web page** (e.g. in a tab): **Renderer** process for each web pages dealing with HTML and Javascript.
 - **For each plug-in** (e.g. Flash): **Plug-in** process running the plug-in code.
 - Separate Renderer and Plug-in processes: provide **sandbox**, minimizing effect of security exploits (problem occurs only in that process).
- Interprocess communication needed between processes (e.g. HTML to be rendered to Renderer process) → next section.



3.4 Interprocess Communication (IPC)

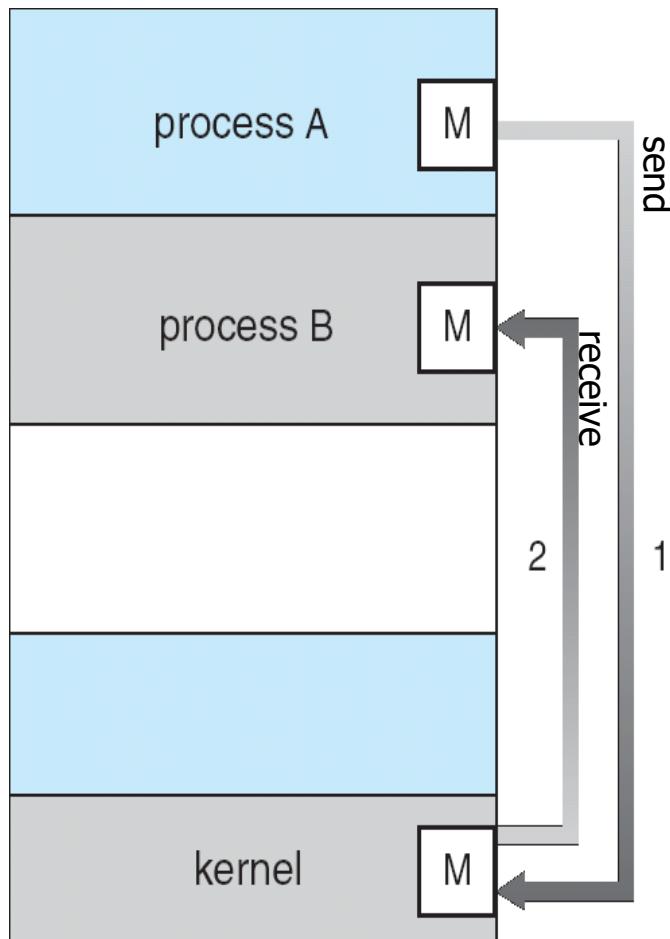
- Concurrently running processes may be independent or cooperating.
- An **independent process** is not related to any other process.
 - Does not communicate or share data with any other processes.
- A **cooperating process** communicates or shares data with other processes (directly, not via files).
 - Reasons for process cooperation:
 - Information sharing: several users work together on the same data.
 - Computation speed-up: using parallel processing on a multi-processor/-core system.
 - Modularity: as, e.g., used in microkernel OS design.
 - Convenience: e.g. Compiler process translates source code to textual assembly instructions, assembler process takes them and converts them to machine code.
 - Process cooperation requires mechanisms for **interprocess communication (IPC)** and **synchronisation of actions (process synchronisation)**.
 - IPC will be discussed in this and the following section.
 - Process synchronisation will be treated later in a chapter on its own (ch. 6).

Two fundamental models of IPC: Message Passing vs. Shared Memory (1)

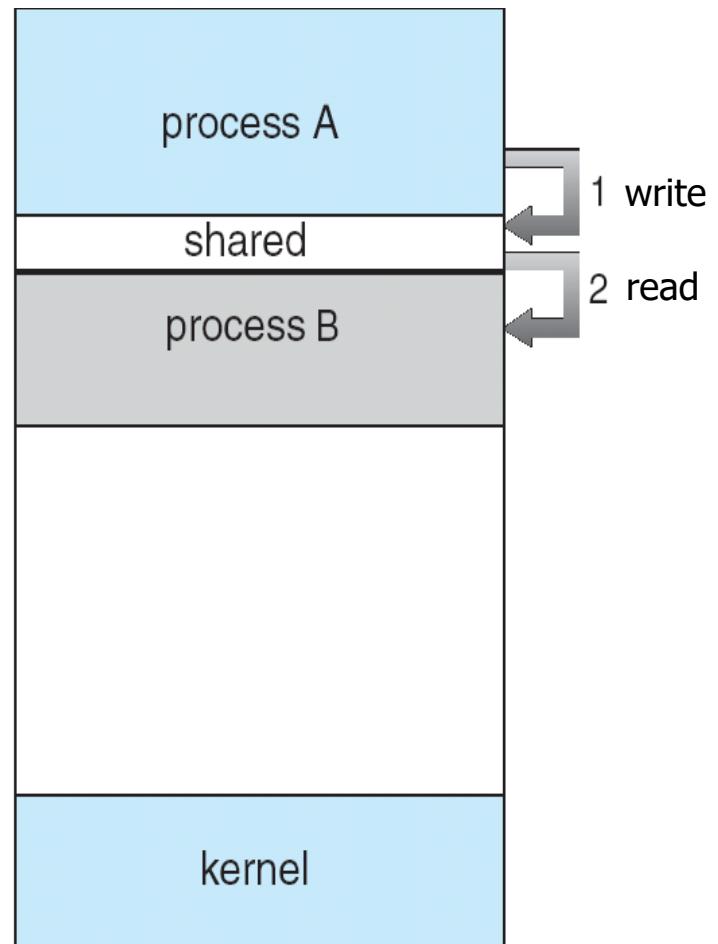
- For communication, processes need to exchange data:
 - Either using **shared memory** that is requested by the communicating processes from the operating system via system calls.
 - Once the shared memory area has been established, ordinary (=fast) memory accesses can be used to read and write shared data without further OS support.
 - Processes need to synchronise access to shared memory: suitable infrastructure needs to be implemented by processes.
 - Preferable if huge chunks of data need to be exchanged.
 - Or using **message passing** facilities of operating system:
 - OS has an internal buffer that can be accessed by different processes via send & receive system calls to exchange data.
 - Operating system provides communication infrastructure and thus synchronises access to exchanged data.
 - Preferable if only smaller chunks of data need to be exchanged.

Message Passing vs. Shared Memory (2)

Message passing:



Shared memory:



Shared-Memory Systems, e.g. POSIX

- Usually, an OS prevents two processes from accessing each other's memory. System calls are needed to **remove memory protection and share memory between processes**.
- POSIX system calls for shared memory, e.g.:
 - `int shmget (long key, int size, int flag)`: get an Id (will be the return value) for a shared memory area using a `key` (on which communicating processes have to agree on – e.g. using message passing).
 - `void* shmat (int id, char* addr, int flag)`: map shared memory area to process memory at `addr` using the `id` that identifies the previously created shared memory area.
- (We will have a look in later chapters on how an OS does realise protected and shared memory and on how to synchronise access to a shared buffer.)
- While communication via shared memory is fast, it requires that the communicating processes reside on the same machine.

Message-Passing Systems

- Mechanism provided by OS to allow processes to communicate and to synchronize their actions without using shared variables/shared memory: **Messages are exchanged between processes.**
- IPC provides two operations to processes that want to communicate:
 - **send(*message*)**
 - **receive(*message*)**
- If processes *P* and *Q* wish to communicate, they need to:
 1. Establish a **communication link** between them.
 - Communicating processes on two different machines: network connection.
 - Communicating processes on the same machines: usually a buffer of the operating system.
 - However, this buffer is no shared memory between processes that can directly accessed by the processes. (Rather, only send & receive can be used to add and retrieve data from the head and tail of the buffer.)
 2. Exchange messages via the communication link using send & receive.

Helmut Neukirchen: Operating Systems/updated

I.Hjörleifsson ingolfuh@hi.is st. 316 Gace,
Gróska

Design Decisions when Implementing an OS with Message Passing IPC

- Communication:
 - direct (processes need to know each other) vs.
 - indirect communication (processes communicate via a well-known mailbox):
- Communication:
 - synchronous (send & receive may block if buffer full or empty respectively) vs.
 - asynchronous (non-blocking send & receive).
- Buffer size:
 - no buffer,
 - automatic buffering (either bounded or unbounded).
- Further details on following slides...

Direct vs. Indirect Communication:

Direct communication

- Direct communication: Processes must name each other explicitly.
 - **send**(P , message): send a message to process P ,
 - **receive**(Q , message): receive a message from process Q .
 - Disadvantage: processes must know the PId of the other process.
 - In practice, e.g. a server process may write its PId to a file with a well-known file name. All client processes may then retrieve PId of server process from that file and send message to server.
- Properties of communication link:
 - Links are established automatically.
 - A link is associated with exactly one pair of communicating processes.
 - Between each pair there exists exactly one link.
 - Not possible to have multiple independent links between a pair. (Because no further information than the PId would be available to distinguish links.)
 - The link may be unidirectional, but is usually bi-directional.

Direct vs. Indirect Communication: Indirect Communication (1)

- **Indirect communication:** Messages are directed and received from **mailboxes** (also referred to as **ports**):
 - **send**(*A, message*): send a message to mailbox A
 - **receive**(*A, message*): receive a message from mailbox A
 - **Advantage:** processes only need to agree on Id of mailbox.
 - Mailbox Ids must be well-known to avoid that different application accidentally use the same Id. Mailbox must be created first (→next slide)
- Properties of communication link:
 - Link established only if processes decide to share a common mailbox.
 - A link/mailbox may be associated with many processes.
 - Each pair of processes may share several communication links (by using multiple mailboxes at the same time).
 - Link may be unidirectional or bi-directional.

Direct vs. Indirect Communication: Indirect Communication (2)

- For supporting indirect communication, an OS must provide the following operations:
 - Create a new mailbox using a certain Id,
 - Send and Receive messages through a mailbox,
 - Destroy a mailbox.

Synchronization: Blocking vs. Non-Blocking Send & Receive

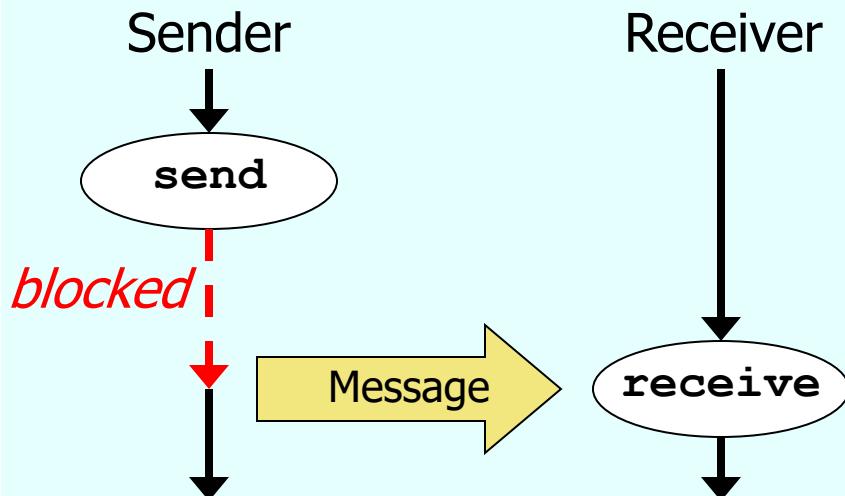
- Send & receive operations may be blocking or non-blocking:
 - **Blocking:**
 - **Blocking send:** sender is blocked until the message is received by the receiving process or put by the OS into some buffer/mailbox.
 - **Blocking receive:** a process that issues a receive operation is blocked until a message is available.
 - **Non-blocking:**
 - **Non-blocking send:** sender continues immediately after sending.
 - **Non-blocking receive:** receiver receives a valid message or gets “null” (or some error return value) if no message is available.

(More examples on next slides...)

Synchronization: Blocking Send & Receive

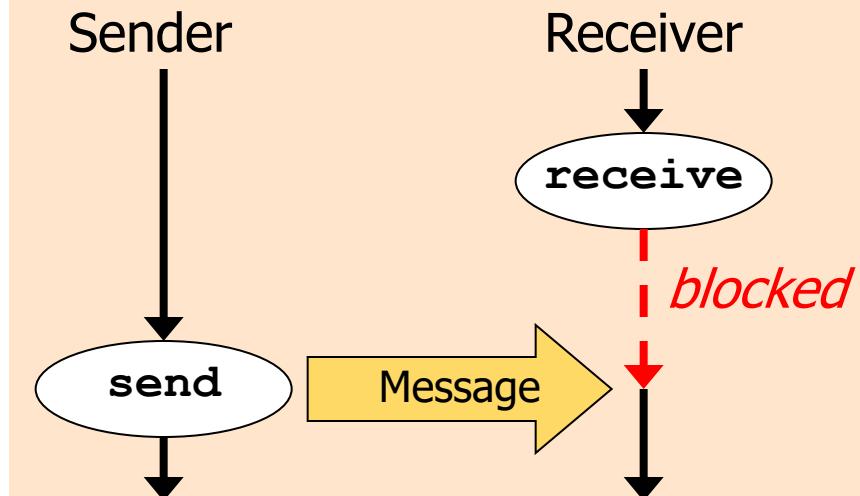
■ Blocking Send

- Sender waits/blocks until receiver (or buffer) is ready to receive.



■ Blocking Receive

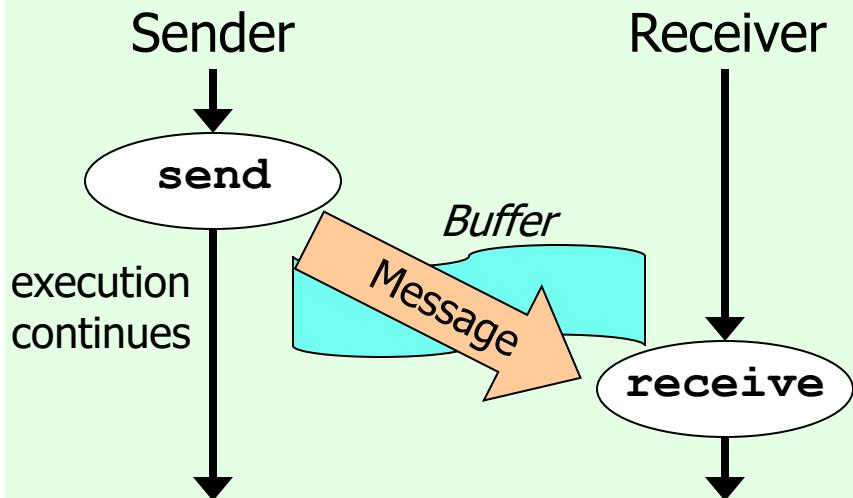
- Receiver waits/blocks until Sender sends a message.



Synchronization: Non-Blocking Send & Receive

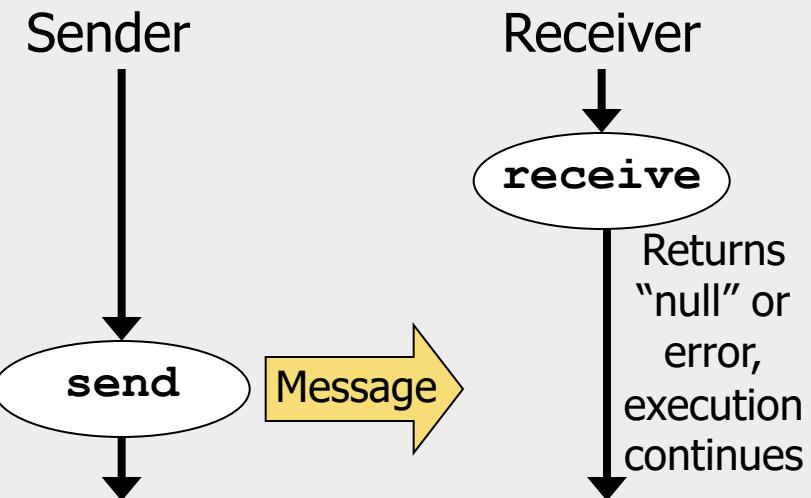
■ Non-Blocking Send

- Sender does not wait for receiver. Message is stored in buffer or (in case of no buffering): message gets lost if receiver is not ready.



■ Non-Blocking Receive

- Receiver does not wait for message. If no message is available, receive operation returns "null" or an error.



Buffering: Buffer size

- Buffer queue provided by OS for each communication link.
- Possible buffer capacities:
 1. **Zero capacity**: 0 messages buffer size, i.e. no buffer at all.
 - If blocking send is used, sender must wait for receiver until receiver retrieves/receives the message.
 2. **Bounded capacity**: finite length of n messages.
 - If blocking send is used, sender must wait if communication link (or its buffer respectively) capacity exceeded.
 3. **Unbounded capacity**: infinite length.
 - Even if blocking send is used, sender never has to wait.

Synchronous vs. Asynchronous communication

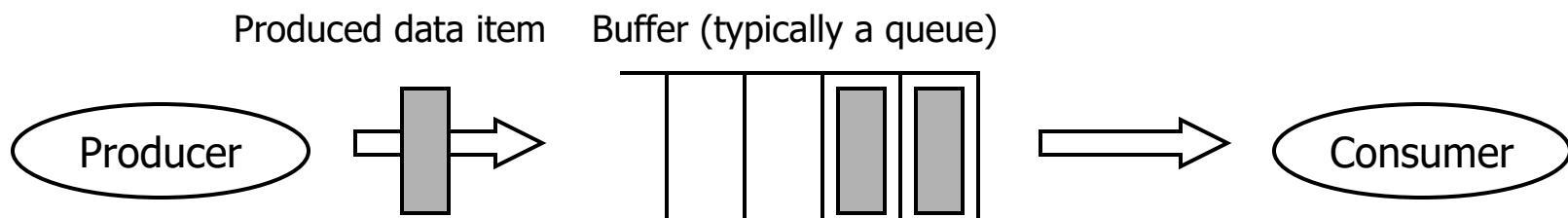
- **Synchronous communication:**
 - Close temporal coupling between send and reception events.
 - E.g. telephone: receiver hears immediately what sender says.
 - **Can be used for synchronisation** between both sender and receiver.
- **Asynchronous communication:**
 - No close temporal coupling send and reception events.
 - E.g. mail: it may take some time until receiver gets what sender posted.
 - **Only partially suitable for synchronisation** between sender and receiver:
 - Receiver does not know exactly when message was sent.
 - Sender does not know when receiver actually receives message.

Synchronization of Processes

- Rendezvous communication:
 - Blocking send und blocking Receive with zero capacity buffer.
 - Often synonymously called **synchronous communication**, because can be used for synchronisation of processes:
 - Sender only continues when receiver received message.
 - Receiver only continues when sender sent message.
- By default, most message passing communication mechanisms use **non-blocking send** (or blocking send with a buffer) and **blocking receive**.
 - Often, this is called **asynchronous communication**. Can only be used to take care that receiver does not start an action before sender does.
 - Receiver only knows that sender must have sent message at any time in the past. Sender knows nothing about the reception.

Producer-Consumer Problem

- A classical example to illustrate the usage/study the applicability of an interprocess communication/synchronisation mechanism:
- Concurrent producer and consumer processes share a common buffer.
 - **Producer:**
 - Puts (inserts) produced data items into buffer until it is full.
 - Waits if buffer is full.
 - Resumes when buffer is not full any more.
 - **Consumer:**
 - Consumes (removes) data items from buffer.
 - Waits if buffer is empty (no items available).
 - Resumes when new items are available.



- Example: Output of `dir` command is piped as input to `more` command.

Variants of the Producer-Consumer Problem

- There are two flavours of the producer-consumer problem:
 - **Unbounded-buffer**: Buffer is infinitely large (i.e. producer never has to wait).
 - Note: Computers are finite machines with in particular a finite amount of memory, hence an infinitely large buffer cannot be implemented. However, “unbounded-buffer” simply means that the buffer is large enough for all considered cases. (E.g. a Java vector that can grow instead of an array of fixed size.)
 - **Bounded-buffer**: assumes that there is a fixed buffer size (i.e. a producer may have to wait if buffer is full).

Solution of Producer-Consumer Problem Using Message Passing

- Assumption:
 - non-blocking send with unbounded buffer, non-blocking receive.
 - (Other solutions with other combinations of blocking/non-blocking possible, too.)
- Excerpt of a possible Java solution:

- Producer process:

```
Mailbox mbox = new Mailbox("ProducerConsumerMailbox");
```

```
while (true) {  
    Data message = new Data();  
    mbox.send(message);  
}
```

Infinite buffer renders problem trivial (sender never has to wait)!

Use Mailbox with string as ID.

Produce data.

Put data into unbounded buffer.

- Consumer process:

```
Mailbox mbox = new Mailbox("ProducerConsumerMailbox");
```

```
while (true) {  
    Data message = (Data) mbox.receive();  
    if (message!=null) {  
        // do something with consumed message  
    }  
}
```

Use Mailbox with string as ID.

Non-blocking receive

If receive did return data, otherwise ignore.

Solution of Producer-Consumer Problem Using Message Passing

- The Java code on the previous slide assumes that some class **Mailbox** is available (e.g. provided by the Operating System).
 - A possible implementation of class **Mailbox** could look like shown below (code of constructor for creating/using a mailbox with some ID not shown):

```
public class Mailbox {  
    private Vector queue; // Unbounded buffer  
  
    // This implements a non-blocking send  
    public void send(Object item) {  
        queue.addElement(item);  
    }  
  
    // This implements a non-blocking receive  
    public Object receive() {  
        if (queue.size() == 0)  
            return null;  
        else  
            return queue.remove(0);  
    }  
}
```

3.5 Communication in Client-Server Systems

- Scope of different interprocess communication mechanisms:
 - Shared memory: same machine.
 - Based on sharing physical memory.
 - Message passing: same machine or different machines connected via a network possible, but same type of operating system required.
 - Based on message passing mechanism that is specifically implemented by each operating system (may be different on each OS).
- How to communicate between processes running on different machines having different operating systems?
 - Standardised communication mechanisms required, e.g. the Internet Protocol (IP).
 - Heavily used in client-server systems:
 - Server, e.g. running on Unix machine, offers service (e.g. HTTP server delivering WWW pages).
 - Client, e.g. running on an MS Windows machine accesses service (e.g. via a web browser).

Sockets

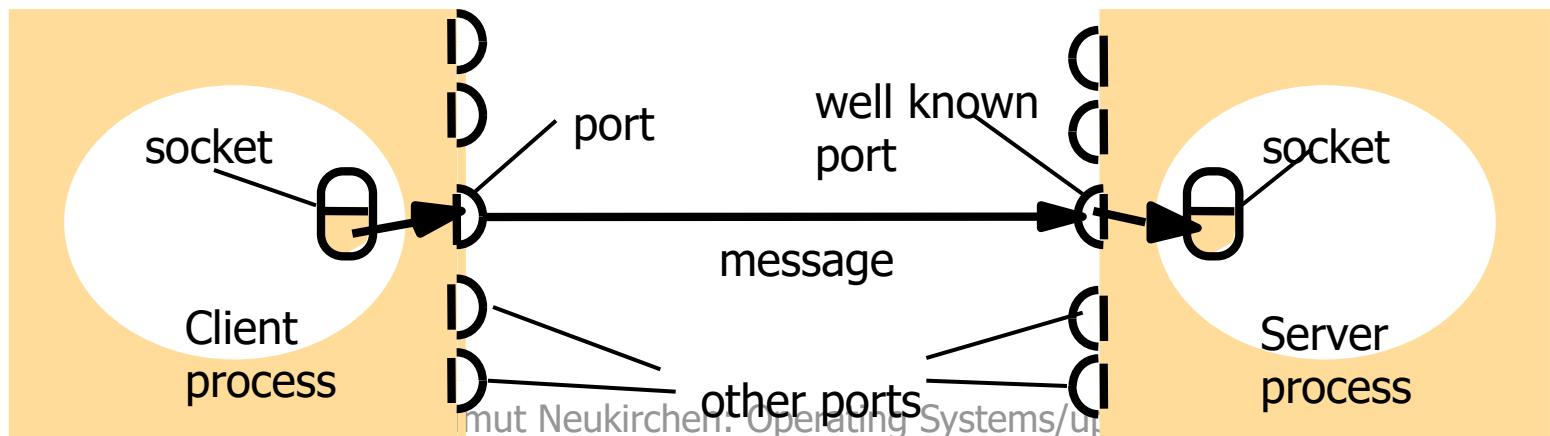
- A generic programming model for non-local and local interprocess communication. Available on all major operating systems:
 - Allows to write source code that runs on many operating systems.
 - Usually provided by the kernel.
 - (But may also be provided by a library on top of the kernel.)
- Sockets are endpoints of communication.
 - Communication takes place between a pair of sockets.
 - The operating system binds the sockets of the sending and receiving process to a port and address. (→slides after next slide)
 - (Comparable to indirect communication using mailboxes).
 - Different communication styles supported:
 - Datagram type (connection-less unreliable) and Stream type (connection-oriented reliable). (→next slide)

Excursion: Reliable Connection-oriented vs. Unreliable Connection-less Communication

- **Unreliable, connection-less communication:**
 - A single message is immediately sent from one process to another (**datagram**). (Datagram has maximum size, e.g. \approx 64KB)
 - Like sending a letter/telegram: sender & intended receiver do not know whether message was lost (e.g. due to network problems) or not.
 - Fast: no overhead involved.
 - Example: User Datagram Protocol (**UDP**) from the IP family.
- **Reliable, connection-oriented communication:**
 - First, a connection needs to be set up. Using this, an endless **stream** of data can be transmitted between a pair of processes.
 - Connection is reliable, because reception of transmitted data needs to be acknowledged by receiver. If sender does not get an acknowledgement, data is retransmitted.
 - Slower: Connection set-up and acknowledgment significant overhead.
 - Example: **Transmission Control Protocol (TCP)** from the IP family.

Sockets, addresses and ports

- Each IP address identifies a machine on the Internet.
 - IPv4 address: 4 x 8 Bits, IPv6: 16 x 8 Bits.
- For making a server process reachable, server needs to listen at a port.
 - Clients contact the agreed well known port number at the server's IP address.
- To enable the server to reply to each client, the client socket has also a port number.
 - Client port number is dynamically assigned by the operating system.
- A process may use several sockets (& ports) at the same time.



Internet address = 138.37.94.248

Internet address = 138.37.88.249

More about port numbers

- Port numbers: 16 Bit integer, i.e. 65536 different ports available per machine.
- Well Known Port Numbers are used for standard services.
 - Example: HTTP server process typically binds port 80 to its socket.
 - All client processes will use as well port 80 as target address for contacting that HTTP server process.
 - Well Known Port Numbers are in the range [0,1023].
 - An operating system does only allow processes running with superuser privileges to bind their socket to a port number <1024.
 - Prevents that any ordinary (malicious) user can start a process that claims to be an “official” service on that machine.
- The range [1024, 49151] (“registered ports”) can be used by server processes started by ordinary users.
- Dynamically assigned port number of client sockets are in the range [49152, 65535].

Helmut Neukirchen: Operating Systems/updated
I.Hjörleifsson ingolfuh@hi.is st. 316 Gace,
Gróska

Java API for Socket Communication

Port numbers and IP addresses

- Java Application Programming Interface (API) supports interprocess communication based on sockets.
- How to specify IP address and Port numbers in the Java world:
 - Port numbers are represented by Integers: Java built-in type `int`
 - IP Addresses have an own data type: class `InetAddress` from package `java.net.InetAddress` (IPv4 and IPv6 supported).
 - For translating domain names (e.g. `www.hi.is`) into IP addresses using the Domain Names Service (DNS), the static method `getByName(String host)` `throws UnknownHostException` can be applied to type `InetAddress`:

```
import java.net.InetAddress;
import java.net.UnknownHostException;
public class InetAddressDemo {
    public static void main(String[] args) {
        try {
            InetAddress anInetAddress = InetAddress.getByName("www.hi.is");
            System.out.println(anInetAddress.getHostAddress());
        } catch (UnknownHostException e) {
        }
    }
}
```

Helmut Neukirchen: Operating Systems/updated
I.Hjörleifsson ingolfuh@hi.is st. 316 Gace,
Gróska

Java API for Connection-less Sockets

- Class **DatagramSocket** from package `java.net.DatagramSocket` represents a socket of datagram type.
 - Once a socket has been constructed, the port number stays fixed.
 - The same socket can be used for sending and receiving via the assigned port.
- Constructors and Methods:
 - `DatagramSocket()`: creates a socket on local machine and assigns free port to it.
 - `DatagramSocket(int port)`: creates a socket that is bound to local port `port`.
 - `void send(DatagramPacket dp) throws IOException`: sends datagram packet `dp` that must have set a destination IP address and destination port.
 - `void receive(DatagramPacket dp) throws IOException`: receives datagram packet and stores it in `dp` that provides space for storing the message. IP address and port of the originating socket can be retrieved from `dp` afterwards.
 - `void close()`: close socket.
- (More about datagram packets on next slide...)

Java API for Packets Transmitted via a Connection-less Socket

- Class **DatagramPacket** from package `java.net.DatagramPacket` represents a datagram packet.
 - May be used for sending and receiving via a **DatagramSocket**.
- Constructors and Methods:
 - **DatagramPacket(byte[] buf, int length)**: creates a packet that can be used for receiving. `buf` must provide space for received message. If received message is longer than `length`, it will be truncated. Origin of message can be retrieved after reception (see getters below).
 - **DatagramPacket(byte[] buf, int length, InetAddress address, int port)**: creates a packet that can be used for sending to IP address `address` and port `port`.
 - `byte[] getData()`: returns the contents of the message.
 - `int getLength()`: returns the length of the message.
 - `InetAddress getAddress()`: returns IP address to which datagram is being sent or from which the datagram was received.
 - `int getPort()`: returns the port number to which datagram is being sent or from which the datagram was received.

Java Example for Connection-less Server Process

```
import java.net.*;
import java.io.*;

public class ConnectionLessServer {
    public static void main(String args[]) {
        DatagramSocket aSocket = null;
        try {
            aSocket = new DatagramSocket(6789); // create socket at agreed port
            byte[] buffer = new byte[1000];
            while (true) {
                DatagramPacket request = new DatagramPacket(buffer, buffer.length);
                aSocket.receive(request);
                DatagramPacket reply = new DatagramPacket(request.getData(), request.getLength(),
                                                request.getAddress(), request.getPort());
                aSocket.send(reply);
            }
        } catch (SocketException e) {
            System.out.println("Socket: " + e.getMessage());
        } catch (IOException e) {
            System.out.println("IO: " + e.getMessage());
        } finally {
            if (aSocket != null)
                aSocket.close();
        }
    }
}
```

■ Server just sends the received message back to the same IP address and port from where it has been received.

Server runs in an endless loop. (Abort with e.g. CTRL-C)

Needs to be changed if already used by some other running server (Error "Socket: Address already in use").

Java Example for Connection-less Client Process

```
import java.io.*;
import java.net.*;
public class ConnectionLessClient {
    public static void main(String args[]) {
        // args[0]: message contents, args[1]: destination hostname
        DatagramSocket aSocket = null;
        try {
            aSocket = new DatagramSocket();
            byte[] message = args[0].getBytes();
            InetAddress aHost = InetAddress.getByName(args[1]);
            int serverPort = 6789; // agreed port
            DatagramPacket request = new DatagramPacket(message, message.length,
                aHost, serverPort);
            aSocket.send(request);
            byte[] buffer = new byte[1000];
            DatagramPacket reply = new DatagramPacket(buffer, buffer.length);
            aSocket.receive(reply);
            System.out.println("Reply: " + new String(reply.getData()));
        } catch (SocketException e) {
            System.out.println("Socket: " + e.getMessage());
        } catch (IOException e) {
            System.out.println("IO: " + e.getMessage());
        } finally {
            if (aSocket != null)
                aSocket.close();
        }
    }
}
```

- Client sends message via socket to agreed port and waits on same socket for reply.

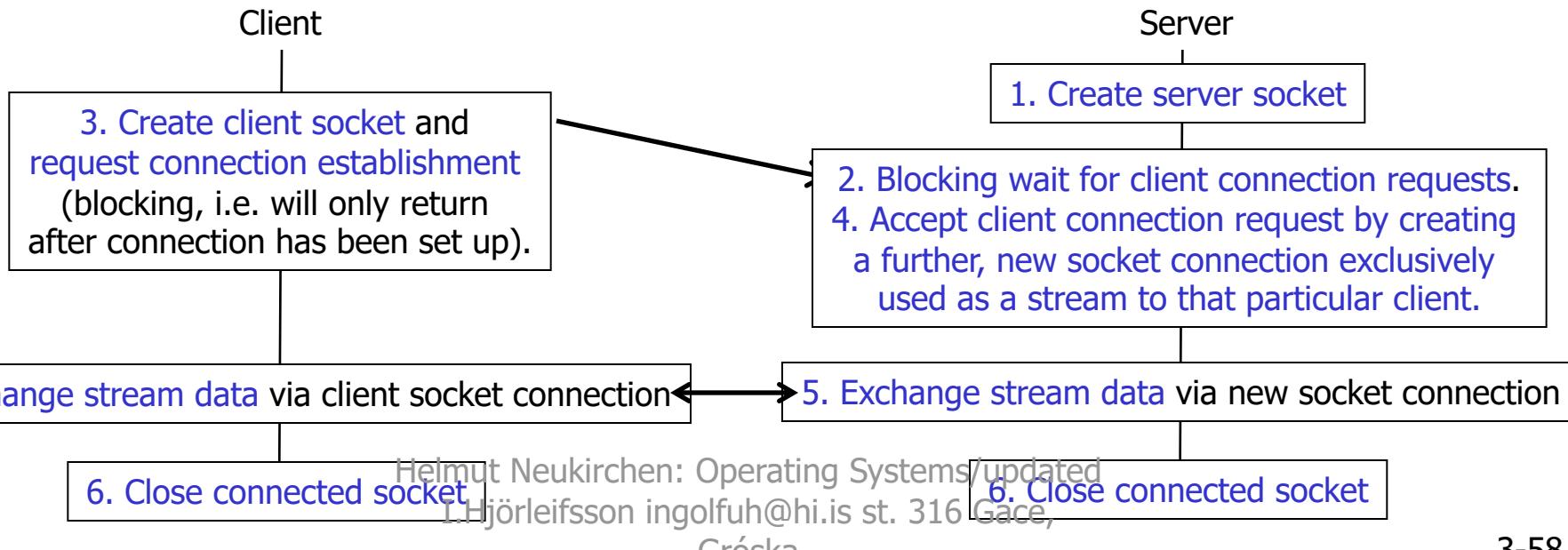
Command line parameters.

Needs to be the same as the server's port.

Helmut Neukirchen: Operating Systems/updated
I.Hjörleifsson ingolfuh@hi.is st. 316 Gace,
Gróska

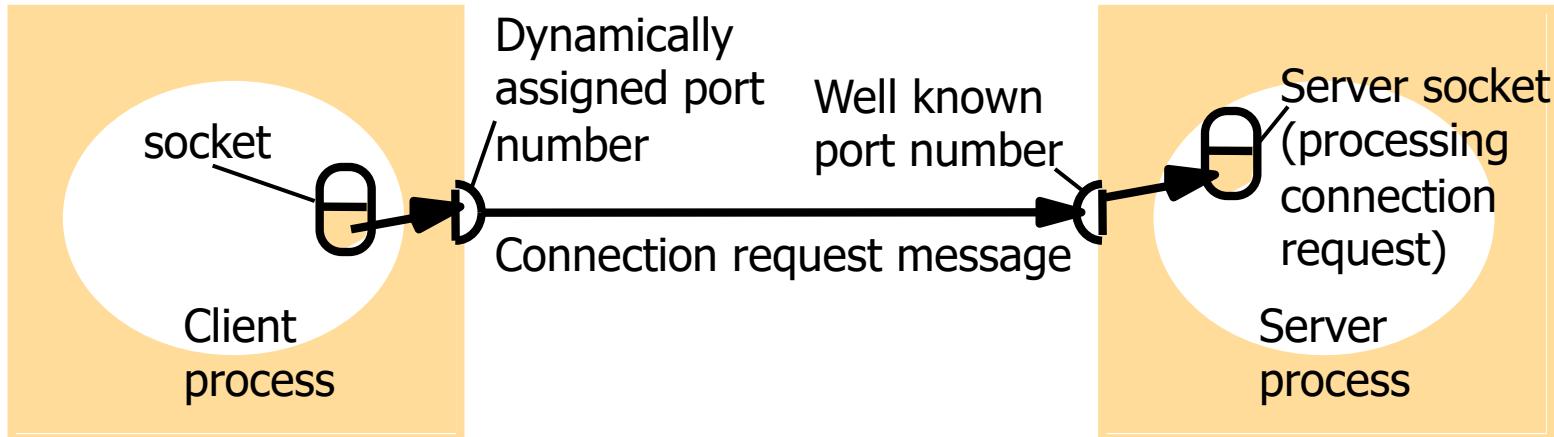
Connection-oriented Sockets for Servers and Clients (1)

- A connection (=bi-directional stream for endless amount of bytes) is initiated by the client and accepted by the server that is waiting for connection requests.
 - Server socket is listening for connection requests from a client socket.
 - Once this connection is established (=request is accepted), a new "connected" socket is created at the server side via which client (using the same client socket via which connection was requested) and server may exchange streams of data.
 - Original server socket can still be used to accept further connection requests.

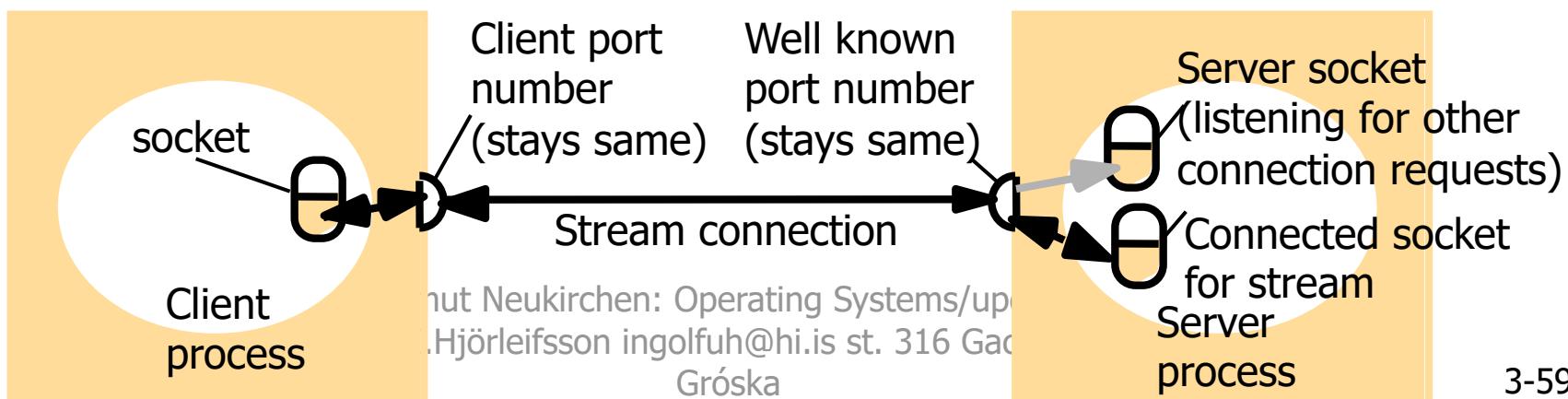


Connection-oriented Sockets for Servers and Clients (2)

- 1. Establishing connection by connecting to server socket listening for connection requests:

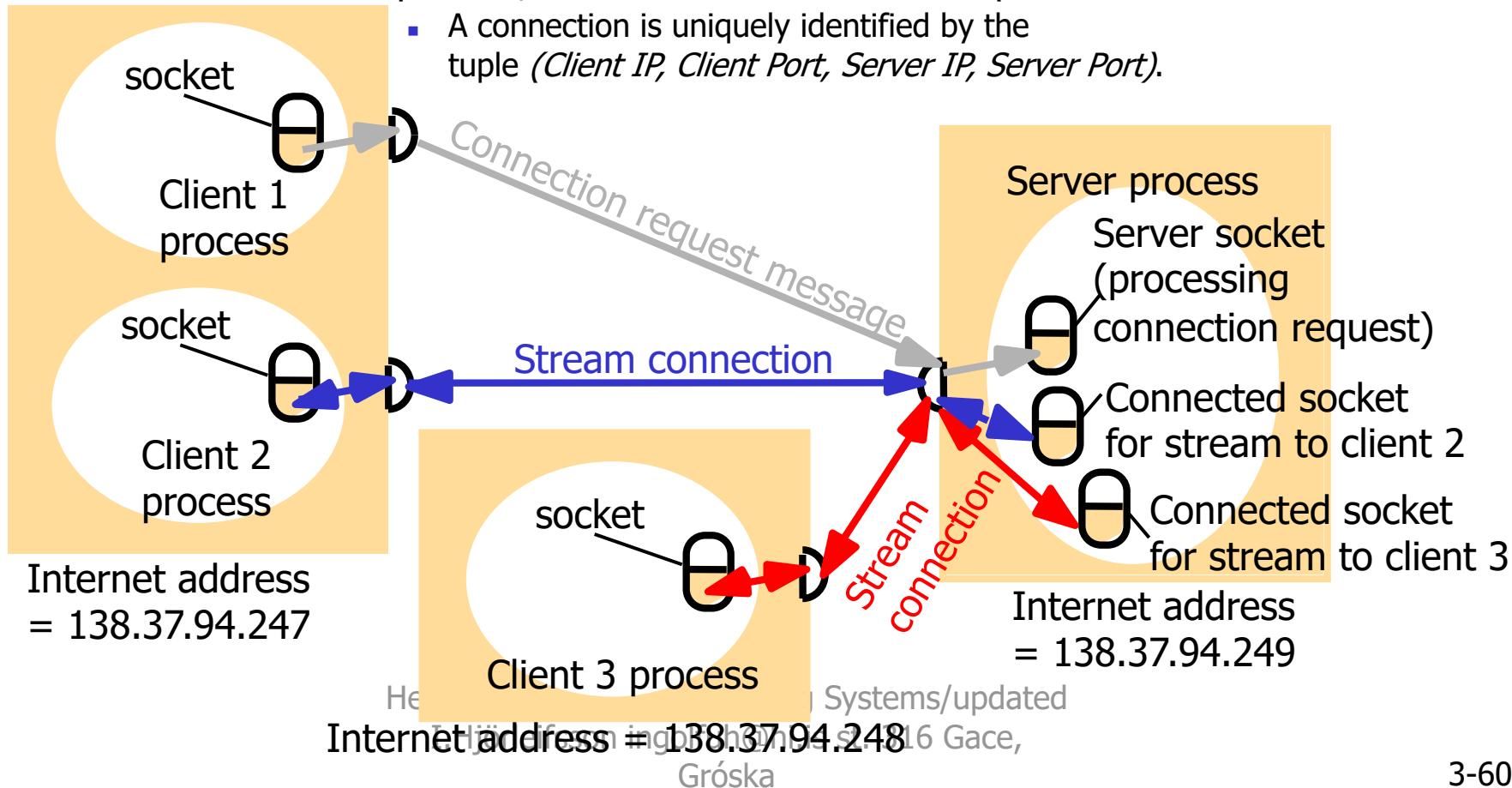


- 2. Established connection via created connected socket at server side, same socket at client side (further connection requests to server socket possible, resulting each time in a further connected socket at server for handling that new stream):



Connection-oriented Sockets for Servers and Clients (3)

- Multiple connections to same server port possible at the same time.
 - Server OS (in charge of ports) can keep connections apart and route them to the right socket of server process, because the clients differ in port number and/or IP number.
 - A connection is uniquely identified by the tuple (*Client IP, Client Port, Server IP, Server Port*).



Java API for Connection-oriented Server Sockets

- Class **ServerSocket** from package `java.net.ServerSocket` represents a connection-oriented server socket.
 - Is only used for listening to connection request (that is used to establish the connection used for sending/receiving the actual data stream).
- Constructors and Methods:
 - **ServerSocket(int port) throws IOException**: creates a socket that is bound to local port `port`.
 - **Socket accept() throws IOException**: listens in a blocking-style for a connection to be accepted. Creates and returns a further connected socket that can be used for the actual stream communication. (See next slide...)

Java API for Connection-oriented Sockets and Stream Communication

- Class **Socket** from package **java.net.Socket** represents a plain socket.
- Constructor:
 - **Socket(String host, int port) throws UnknownHostException, IOException**: Creates a socket for a client and connects it to target host and port of a server socket. Such a connected socket can be used for stream communication.
 - If socket has been constructed and returned by the **ServerSocket.accept()** method, it is also a connected one that can be used for stream communication.
- Methods for applying ordinary Java stream communication (read, write etc.):
 - **InputStream getInputStream() throws IOException**: gets input stream of socket.
 - **OutputStream getOutputStream() throws IOException**: gets output stream of socket.
 - The stream objects returned by **getInputStream()** and **getOutputStream()** can be used as arguments of constructors that create suitable Java input and output streams, e.g. **DataInputStream** and **DataOutputStream** which allow binary representations of primitive Java data type values to be read and written in a machine independent manner using **read** and **write** methods (stream can be used to transmit infinite amount of bytes).

Java Example for Connection-Oriented Server Process

```
import java.net.*;
import java.io.*;

public class ConnectionOrientedServer {
    public static void main(String args[]) {
        try {
            int serverPort = 7896; // the server port
            ServerSocket listenSocket = new ServerSocket(serverPort);
            while (true) {
                Socket clientSocket = listenSocket.accept();
                Connection c = new Connection(clientSocket); // Handle request
            }
        } catch (IOException e) {
            System.out.println("Listen socket:" + e.getMessage());
        }
    }
}
```

- Server just accepts connection and creates class to handle further processing (→next slide).

Needs to be changed if already used by some other server (Error "Socket: Address already in use") or your old server is still running and needs to be terminated first.

The ConnectionOrientedServer class is only responsible for accepting the initial connection of each client. The actual communication (& service provision) is handled by a separate class Connection (→next slide).

Java Example for Connection-Oriented Server Class

```
import java.net.*;
import java.io.*;
class Connection {
    DataInputStream in;
    DataOutputStream out;
    Socket clientSocket;
    public Connection(Socket aClientSocket) {
        try {
            clientSocket = aClientSocket;
            in = new DataInputStream(clientSocket.getInputStream());
            out = new DataOutputStream(clientSocket.getOutputStream());
            handleRequest();
        } catch (IOException e) { System.out.println("Connection:" + e.getMessage()); }
    }
    public void handleRequest() {
        try { // an echo server
            String data = in.readUTF(); // read a Unicode-encoded text string from the stream
            out.writeUTF(data);
        } catch (EOFException e) { System.out.println("EOF:" + e.getMessage()); }
        catch (IOException e) { System.out.println("readline:" + e.getMessage()); }
        finally {
            try {
                clientSocket.close();
            } catch (IOException e) {/* close failed */}
        }
    }
}
```

- Actual communication (& service provision) after accepting connection: send the received stream data back via the established connection.

Connecting to the input and output stream of the socket.

The actual server routine (is called as part of the constructor).

Read/write stream operations are used.

Note: `readUTF/writeUTF` have the string size as a first value, then the actual text string follows – therefore, `readUTF` knows how many bytes it needs to receive from the stream before returning the string.

Java Example for Connection-oriented Client Process

```
import java.net.*;
import java.io.*;

public class ConnectionOrientedClient {
    public static void main(String args[]) {
        // args[0]: message contents, args[1]: destination hostname
        Socket aSocket = null;
        try {
            int serverPort = 7896; Needs to be the same as the server's port.
            aSocket = new Socket(args[1], serverPort);
            DataInputStream in = new DataInputStream(aSocket.getInputStream());
            DataOutputStream out = new DataOutputStream(aSocket.getOutputStream());
            out.writeUTF(args[0]); // UTF is a Unicode-based string encoding
            String data = in.readUTF(); // read a UTF string from the stream
            System.out.println("Received: " + data);
        } catch (UnknownHostException e) { System.out.println("Socket:" + e.getMessage()); }
        } catch (EOFException e) { System.out.println("EOF:" + e.getMessage()); }
        } catch (IOException e) { System.out.println("readline:" + e.getMessage()); }
    } finally {
        if (aSocket != null)
            try {
                aSocket.close();
            } catch (IOException e) { System.out.println("close:" + e.getMessage()); }
        }
    }
}
```

- Client sends message via socket to agreed port and waits on same socket for reply.

Read/write stream operations are used.

Helmut Neukirchen: Operating Systems/updated
I.Hjörleifsson ingolfuh@hi.is st. 316 Gace,
Gróska

Excursion: Java package concept used in samples

- Source code of these Java examples is provided in Canvas (*.zip) and in video.
- To separate the different Java examples used in the different chapters and sections, the hierarchical **Java package concept** is used: e.g., classes **ConnectionLessClient** and **ConnectionLessServer** are contained in package
ch3Processes.connectionLessSocket.
 - To start a class using the JVM (`java` command), the **package name needs to be used as prefix** of that class: `java ch3Processes.connectionLessSocket.ConnectionLessClient`
 - In the file system, **each package is mapped to a directory**, e.g. class `ch3Processes.connectionLessSocket.ConnectionLessClient` is contained in a file with path
`ch3Processes/connectionLessSocket/ConnectionLessClient.class` (or `.java`)
 - The **package file path will get added to the Java Classpath**, e.g. if the bytecode of class `ConnectionLessClient` is contained in file
`/home/helmut/workspace/OperatingSystems/bin/ch3Processes/connectionLessSocket/ConnectionLessClient.class`, the JVM needs to be started using
`java -cp /home/helmut/workspace/OperatingSystems/bin ch3Processes.connectionLessSocket.ConnectionLessClient`

No line
break
here

→ Note: Microsoft systems use \ than / as directory separator. For example, if the folder `ch3Processes` is located in `C:\Users\jon ingi`, then, you have to use:

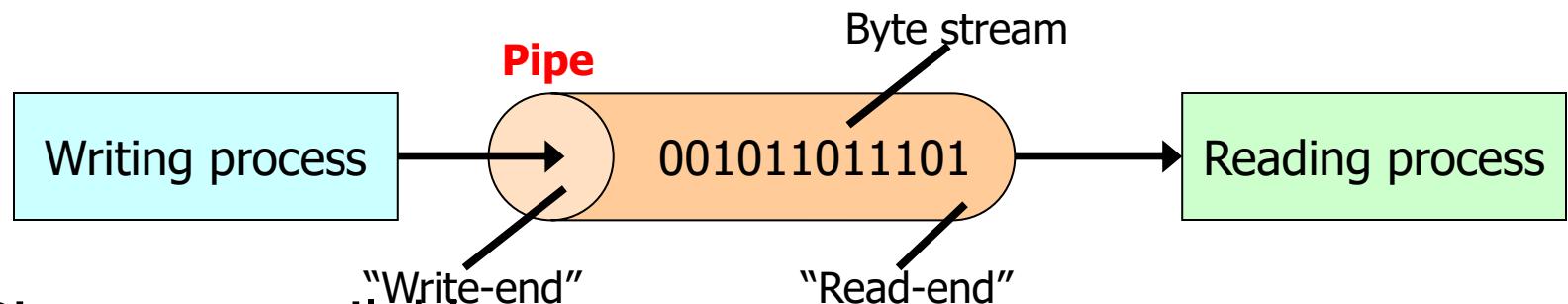
`java -cp "C:\Users\jon ingi" ch3Processes.connectionLessSocket.ConnectionLessClient`

↑
1.Hjörleifsson ingolfuh@hi.is st. 316 Gace,

Double quotes if path contains blanks. On MS Windows, it must not end with "\", i.e. ...ingi", not ...ingi\" 5

Pipes

- Pipes are the most simple interprocess communication mechanism:
 - Transmit a stream of bytes in FIFO First In, First Out processes.
 - Ordinary read/write operations like for files can be used.



- Pipes are available:
 - to a programmer using system calls of POSIX and MS Windows.
 - to the end user using the pipe operator "|" in a command line shell, e.g.: `dir | more`

Characteristics of Pipes

- Pipes
 - have a **limited size** (Linux: 4KByte) and thus their content can be easily kept in main memory (in contrast to sharing data via a file).
 - If pipe full: reader has to read before further data can be written.
 - may be **named** or **unnamed/anonymous**:
 - A **named pipe** is mapped to a file name in the file system. Any process that knows the file name can access that pipe (which is not a real file).
 - An **unnamed/anonymous pipe** requires a parent-child relationship between the communicating processes to be able to refer to and share the same pipe (→process creation using fork).
 - may allow **unidirectional** communication only or **bidirectional** communication.
 - may be restricted to processes running on the **same machine** or may support communication over a **network** (e.g. POSIX named pipes together with the **netcat** system program).

POSIX

Unnamed Pipes

- Unidirectional & Anonymous pipes:
 - System call: `pipe(int fd[2])`
 - Creates pipe (stored in memory managed by OS); returns a file handle ("file descriptor") for the read-end (`fd[0]`), and a file handle for the write-end (`fd[1]`).
 - These handles can be used like opened files.
 - Write-end may only be used to write to pipe, read-end to read from.
 - Usually, parent process creates pipe using above system call and then forks.
 - Child process is a copy of parent, thus it inherits the file handles that refer to the same anonymous pipe. Pipe (owned by OS) not copied, just the handles to it.
 - What one process writes to the pipe can be read by the other process from the pipe.
 - Pipe has limited size: if a process is writing faster to the pipe than the other process is reading, pipe fills up: writing process gets blocked until some space is available again (due to reading). Reading process gets blocked if pipe is empty.
 - A process that is writing shall close its read-end, a process that is reading shall close its write-end (to let the OS know the direction of the pipe).

POSIX Unnamed Pipes: C example

```
main() {
    char buffer[5]; /* Buffer for received data */
    int fd[2];      /* File descriptors for read-end (fd[0])
                        and write-end (fd[1]) of pipe */
    pipe(fd); /* Create unnamed, directional pipe: fd[0] and fd[1] are returned */
    if (fork() == 0) /* Fork child process that gets a copy of file descriptors */
    { /* Child process as writer */
        close(fd[0]); /* Close unused read-end (refers only to read-end of child) */
        write(fd[1], "Test", 5); /* Write 5 byte string to write-end of pipe */
        ...
        exit(0); /* Terminate child process */
    }
    else
    { /* Parent process as reader */
        close(fd[1]); /* Close write-end (refers only to write-end of parent) */
        read(fd[0], buffer, 5); /* Read 5 bytes from read-end of pipe */
        printf("Read: %s\n", buffer);
        ...
        exit(0); /* Terminate parent process */
    }
} /* When processes terminate, OS will close all opened files. Once the last
   file descriptor for a pipe is closed, unnamed pipe ceases to exist. */
```

3.6 Summary

- Process: program in execution.
 - Process Control Block (PCB) represents a process by storing all relevant information of a process.
 - Process states: new, ready, running, waiting, terminated.
 - Non-running processes placed in queues.
- Scheduler responsible for switching between processes (context switch).
- System calls for creating and terminating processes.
- Process cooperation requires interprocess communication:
 - Shared memory,
 - Message passing,
 - Sockets,
 - Only Java Socket API covered – POSIX socket API different: not object-oriented.
 - (*Remote Procedure Call/Remote Method Invocation*),
 - Pipes.

Helmut Neukirchen: Operating Systems/updated
I.Hjörleifsson ingolfuh@hi.is st. 316 Gace,
Gróska

Course TÖL401G: Stýrikerfi / Operating Systems 4. Threads

Mainly based on slides and figures subject of
copyright by Silberschatz, Galvin and Gagne

Chapter Objectives

- Identify the basic components of a thread, and contrast threads and processes.
- Describe the major benefits and significant challenges of designing multithreaded processes.
- Describe different multithreading models.
- Design multithreaded applications using the POSIX Pthreads and Java threading APIs.
- Have heard about implicit threading approaches.
- Know about issues of using threads.

Contents

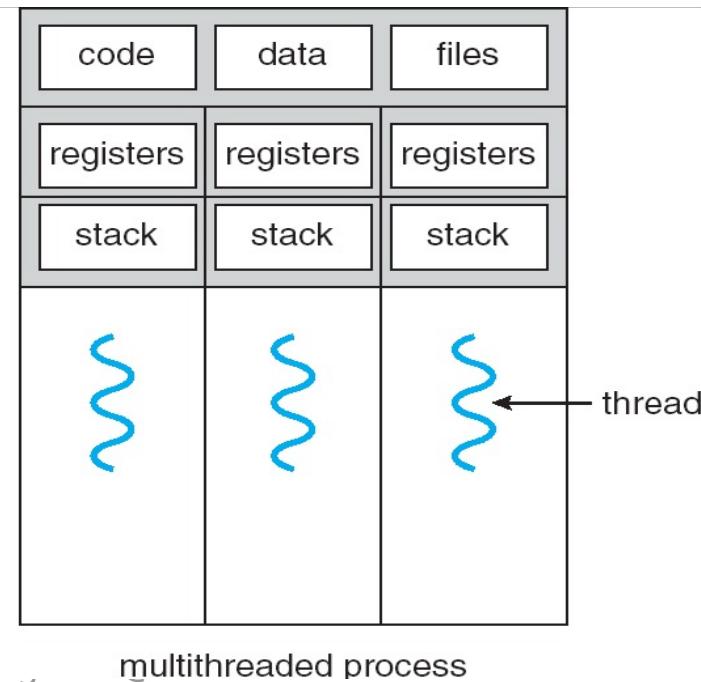
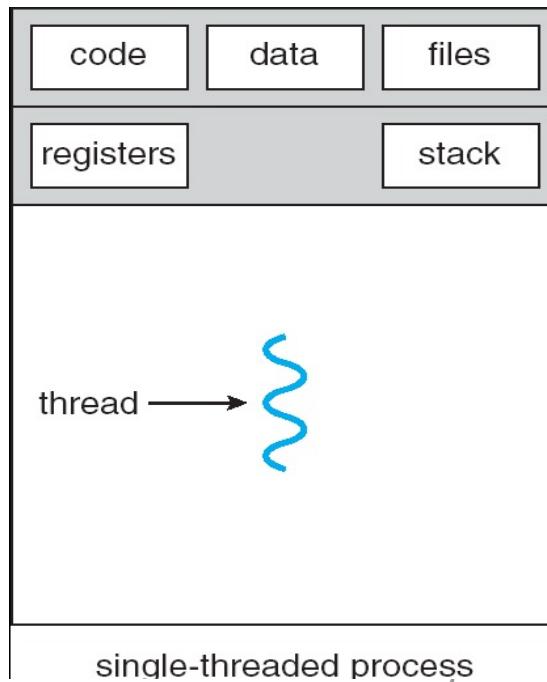
1. Overview
2. Multithreading Models
3. Thread libraries
4. Java Threads
5. Implicit Threading
6. Threading Issues
7. Operating-System Example: Linux Threads
8. Summary

4.1 Overview Threads

- Threads are some sort of “process within a process”.
- Threads are parallel flows of control
 - **within** one process.
 - that are not separated from each other, instead
 - they share resources of the process (in particular the memory of the process).
- Threads are also called **lightweight processes**:
 - Creation of a thread and context switch is more efficient/faster compared to processes (as long as switching threads within the same process – switching threads of different processes: full context switch):
 - Only a minimal part of the hardware context needs to be saved and restored: Just the CPU registers (incl. program counter (PC) and stack pointer).
 - No switch of address space (hardware for memory management that controls accesses to memory needs not to be reprogrammed during a context switch).

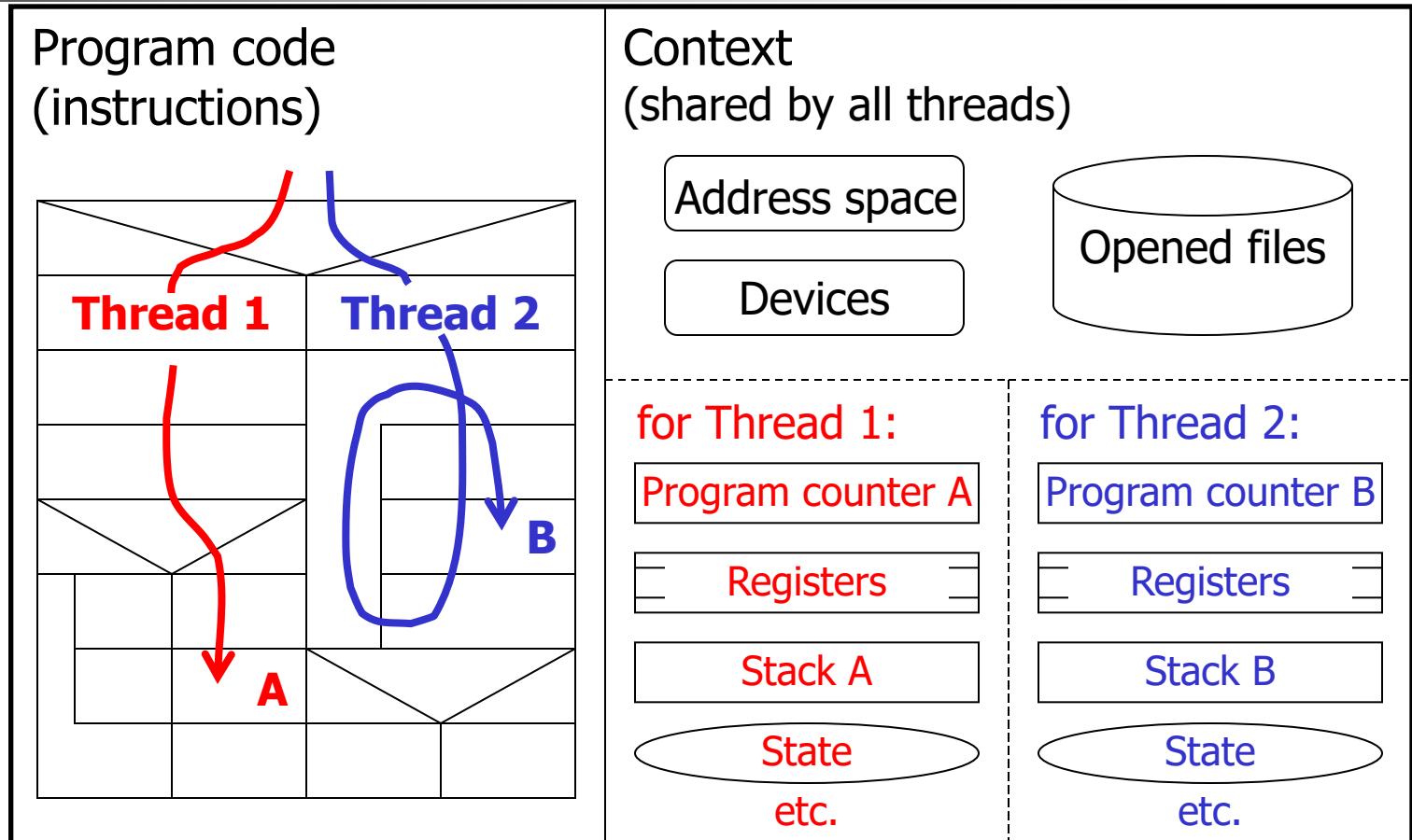
Single and Multithreaded Processes

- On operating systems that support threads, each process has at least one thread that executes the instructions of that process.
- Further threads may be created on request.
 - If process terminates (=one thread calls `exit()`), all contained threads are terminated.
 - However, if a thread just finishes, the surrounding process does not terminate (as long as at least one thread is still running.) – But a crashing thread may crash the whole surrounding process (that's why Google Chrome browser is multiprocess, not multithreaded).



A Closer Look on a Multi-threaded Process

Process (Task)

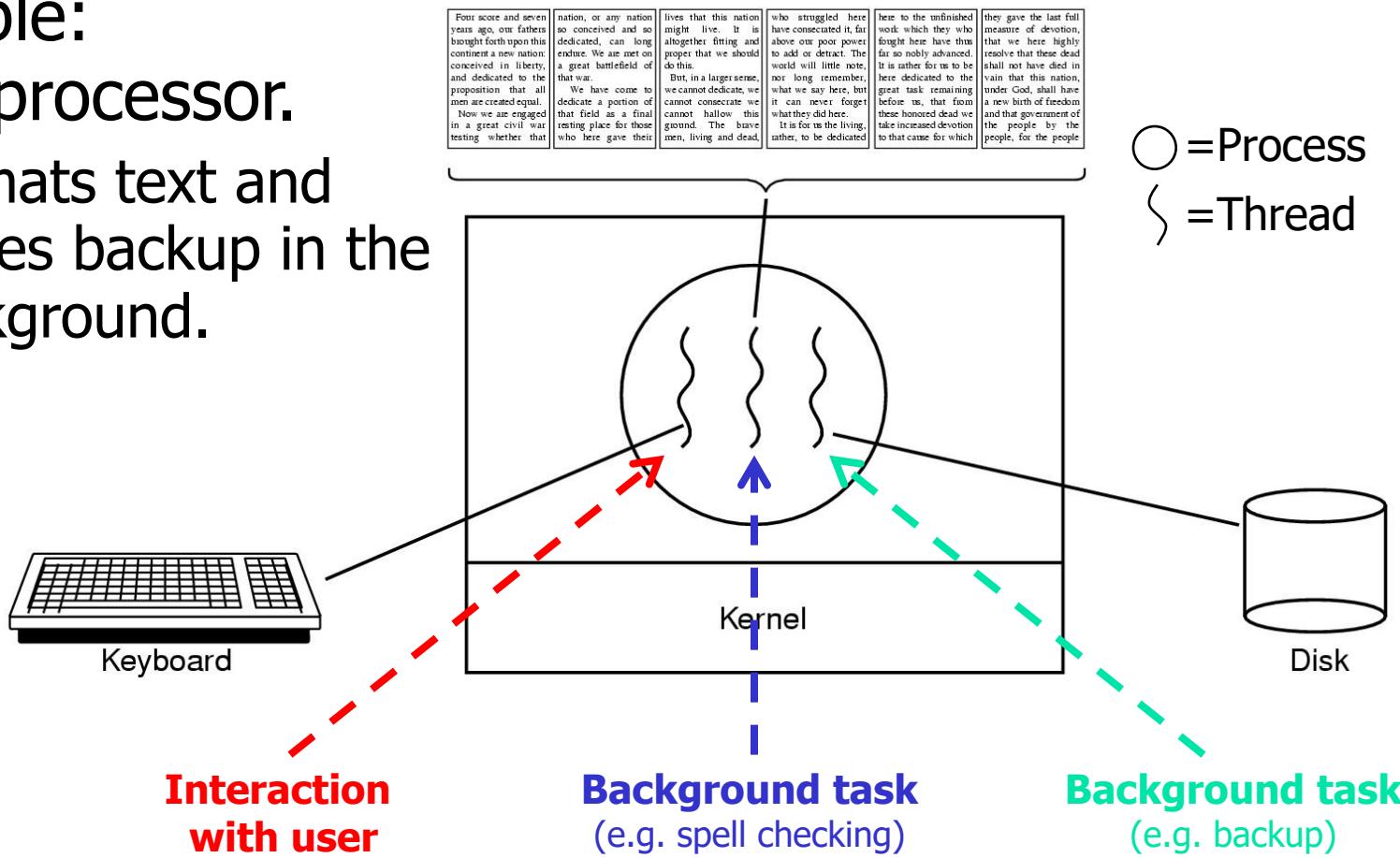


- In addition to process information stored in PCB, thread information (program counter, registers, thread state) needs to be stored for each thread in a thread control block. No process state anymore, instead each thread has a state.

Application of Threads (1)

- Example:
Word processor.

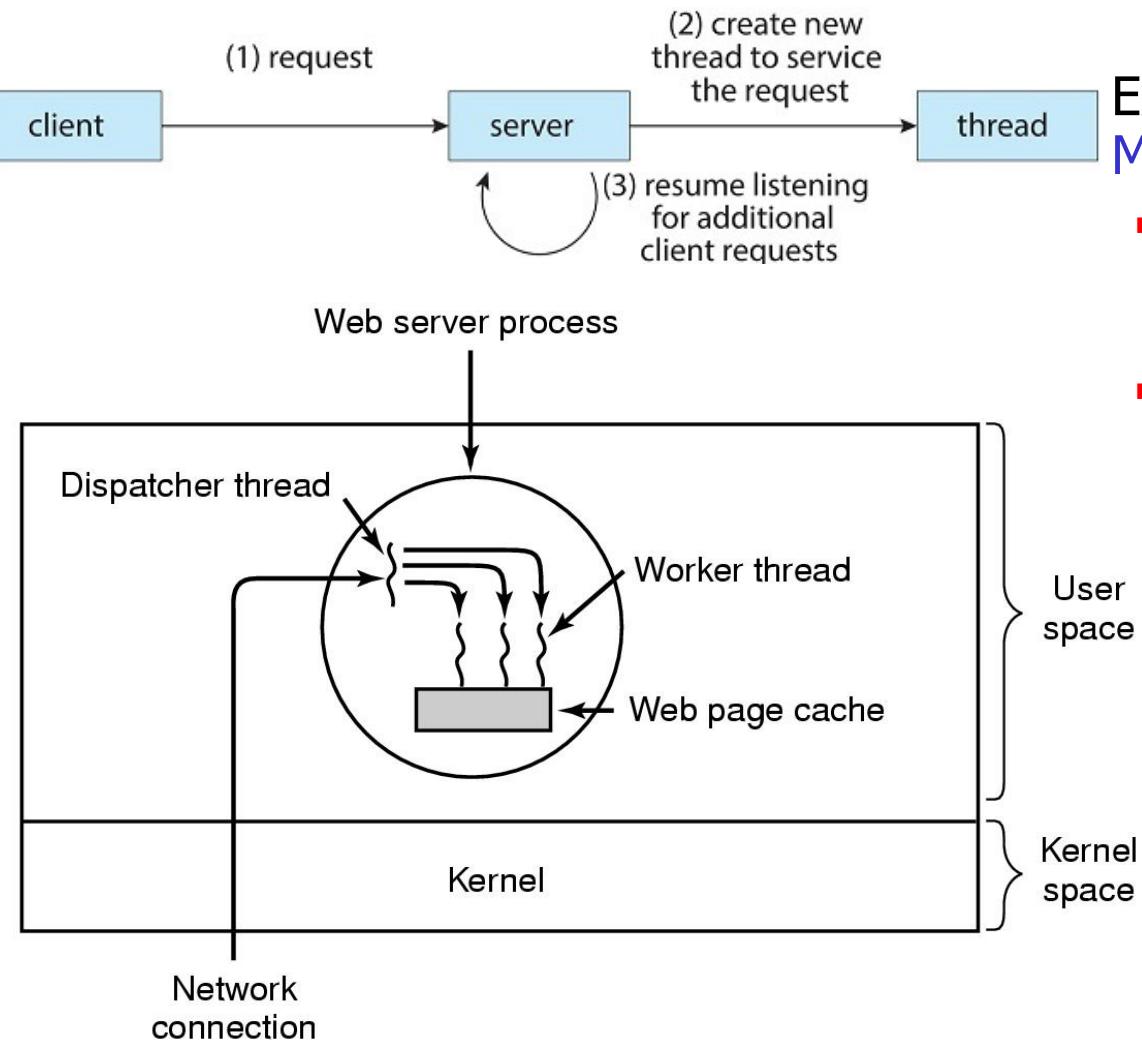
- Formats text and makes backup in the background.



Application of Threads (2)

- Example: Web server:
 - Incoming request for HTML page:
 - HTML page must be read from disk (or cache) and sent to client.
 - What if a further request arrives during that time?
- Possibilities for designing a server, e.g.:
 1. One process (i.e. one thread):
 - No concurrency, blocking system calls.
 - While processing request (involves blocking system calls, e.g. to read file from disk), no new requests possible. (Client blocked or even aborts due to timeout).
 2. One process with multiple threads (multi-threaded server):
 - Concurrency by threads, blocking system calls are no problem:
 - While one thread processes request (which may involve blocking system calls), a further thread may process further requests.

Application of Threads (3)



Example for possibility 2: Multi-threaded Web Server

- One *dispatcher thread* accepts requests and distributes work to worker threads.
- *Worker threads* load Web page using (blocking) system calls.
 - Variant 1: Each request creates a new thread that terminates itself again.
 - Creating thread and terminating it after request processing finished adds overhead.
 - Variant 2: A pool of worker threads is created once at the beginning: thread from pool is either idle or processes a request.
 - See next slide.

Application of Threads (4)

- Dispatcher thread:
- Each worker thread from thread pool:

```
while (TRUE) {  
    get_next_request(&buf);  
    handoff_work(&buf);  
}
```

(a)

Coordination

```
while (TRUE) {  
    wait_for_work(&buf)  
    look_for_page_in_cache(&buf, &page);  
    if (page_not_in_cache(&page))  
        read_page_from_disk(&buf, &page);  
    return_page(&page);  
}
```

(b)

Read will block until
disk I/O has finished.

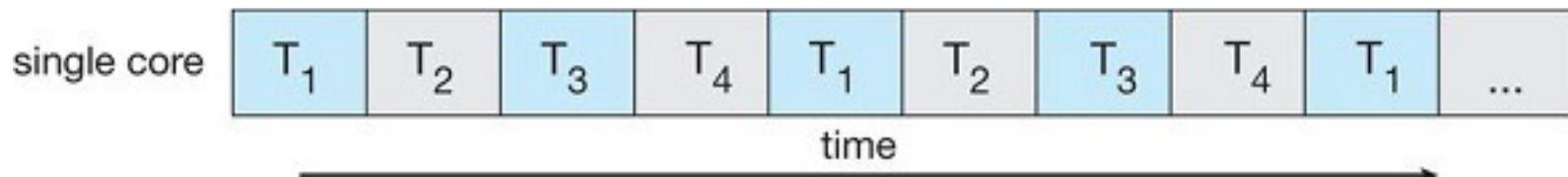
- Not shown:
 - Creation of thread pool.
 - Coordination between dispatcher and worker threads
(=Implementation of **handoff_work()** & **wait_for_work()**).
 - **wait_for_work()** will put worker thread to sleep if no work is available.
 - **handoff_work()** will put dispatcher thread to sleep if all workers are busy.
 - Note: This is essentially the producer-consumer problem with one producer and multiple consumers.

Benefits of Threads

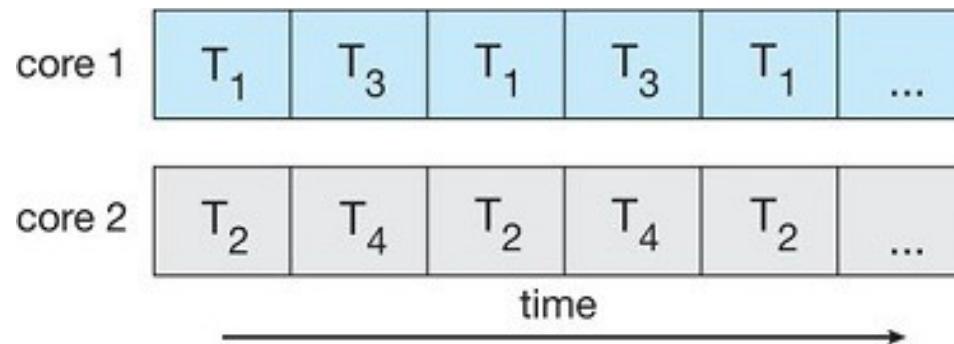
- **Responsiveness:**
 - A single-threaded process would not be able to perform user interactions while doing some computation. However, a multi-threaded process may start one thread for computation, one thread for user interaction, etc.
 - A multi-threaded server is able to handle multiple client requests at the same time (even if request processing involves blocking system calls).
- **Resource sharing:**
 - Memory of process is shared between threads. No system calls required for creating shared memory area or for message passing.
 - Nevertheless, synchronisation between threads may be required using system calls. (See later chapter on synchronisation.)
- **Economy (reduced overhead):**
 - Creating threads is faster than creating processes. Context switch (between threads of same process) is faster for threads than for processes.
- **Scalability/Utilization of multiprocessor/multicore architectures:**
 - Each thread can be executed by a different processor/core, achieving a speed-up by parallel processing (\rightarrow next slide).

Concurrency vs. Parallelism

- Concurrent execution on single-core system:
 - Scheduler uses time slices to interleave threads/processes, i.e. they run concurrently (e.g. fight for the same resources), but not really in parallel.



- Multi-core/multi processor system, allows to turn concurrency into real parallelism:



- If there are more threads/processes than cores/processors, still scheduler needs to interleave time slices (i.e. again concurrency, but true parallelism at least for a subset of the execution).

Multicore (Multiprocessor) Programming

- Threads help to utilise multicore (multiprocessor) systems:
 - If only one process (or thread) would be running, all the other cores/processors would be idle.
 - By writing programs that make use of multithreading, a significant speed-up can be achieved on multicore (multiprocessor) systems.
- Challenges in multicore (multiprocessor) programming:
 - Dividing activities: which activities can run in parallel?
 - Balance: overhead of thread handling/communication/synchronisation may outweigh performance gain (if too small tasks are divided on threads).
 - Data splitting: not only a challenge how to split activities, but also how to divide data processed by different threads.
 - Data dependency: if a thread depends on data produced by another thread, synchronisation between threads needed (& reduced concurrency).
 - Testing and debugging: inherently more difficult than single-threaded applications.

4.2 Multithreading Models

Kernel Threads vs. User Threads

- Two possibilities to provide thread support:
 - **Kernel threads:**
 - Thread management provided by the kernel ("kernel space").
 - CPU scheduler responsible for context switch of processes and threads.
 - **User threads:**
 - Thread management provided by a user-level library ("user space").
 - OS is not aware that processes uses multiple threads.
 - Process creates threads via user-level library that manages these.
 - A thread runs until it voluntarily decides to give control to another thread of the same process:
 - Thread has to call the user-level thread library to hand over control to another thread: ordinary (=fast) function call, instead of "expensive" scheduler interrupts and context switch.
- ⇒ Less overhead, but no automatic periodic switching between threads.

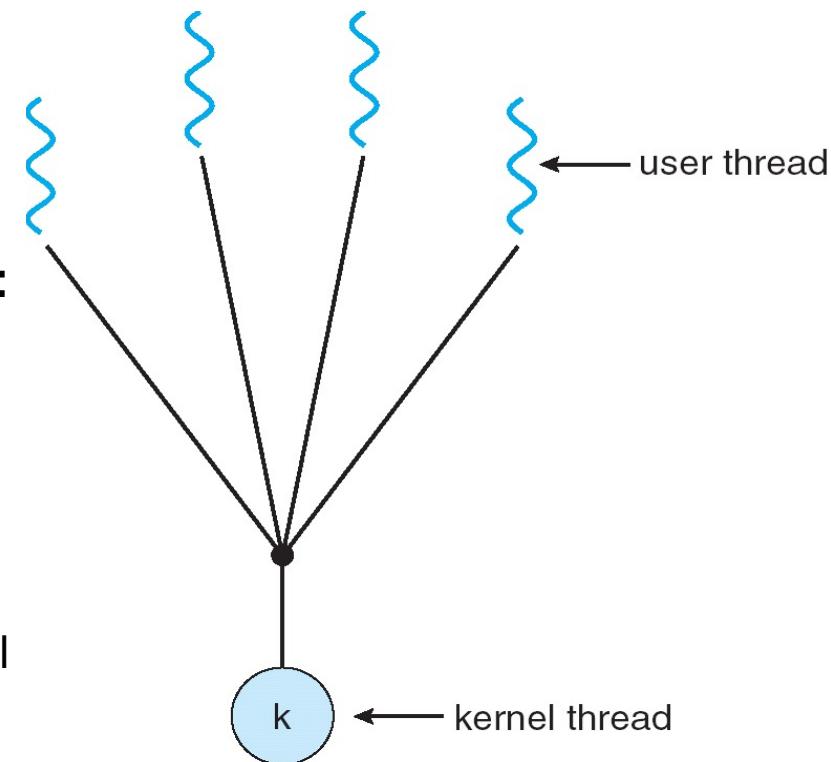
Helmut Neukirchen: Operating Systems/updated

Multithreading Models

- Even if kernel-level threads are supported by an OS, user-level threads may be used on top of it.
- Depending on how user-level threads are mapped on kernel-level threads, different multithreading models result:
 - Many-to-One,
 - One-to-One,
 - Many-to-Many.
 - (We will have a closer look on each of them on the following slides...)

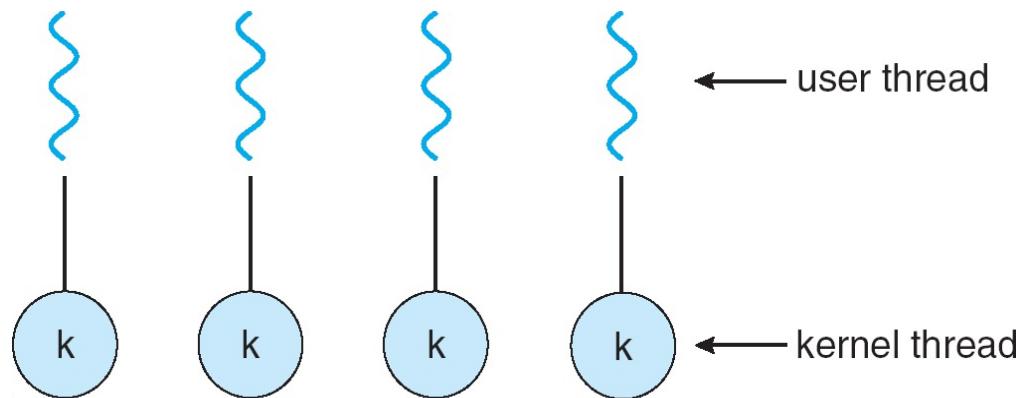
Many-to-One Multithreading Model

- One kernel-level thread per process.
- Multiple threads within a process achieved using user-level threads (less overhead).
 - OS is not aware of user-level threads:
 - If one of the user-level threads makes a blocking system call, all other user-level threads of that process are blocked as well.
 - Cannot make use of multiprocessor-/core architecture to run multiple threads of the same process in parallel (only the OS can distribute threads on different processors/cores).
- Examples:
 - Solaris *Green Threads*,
 - GNU *Portable Threads*.



One-to-One Multithreading Model

- Each user-level thread maps to kernel thread



Many-to-Many Multithreading Model

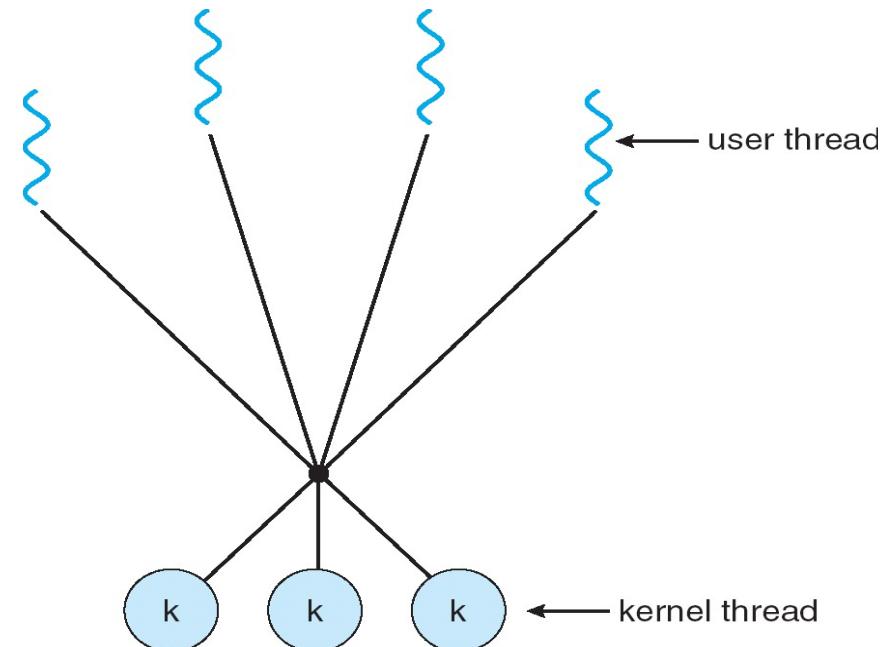
- Many user-level threads mapped to a smaller (or equal) number of kernel threads.

- Best of both worlds:

- Application may use efficient user-level threads.
 - However, when thread is blocked, a new kernel level thread is created that may execute one of the remaining non-blocked user-level threads.

- Examples:

- Solaris prior to version 9,
 - Microsoft Windows \geq NT < 7 with the *ThreadFiber* package.
 - Microsoft Windows \geq 7 with the *user-mode scheduling (UMS)* threads.



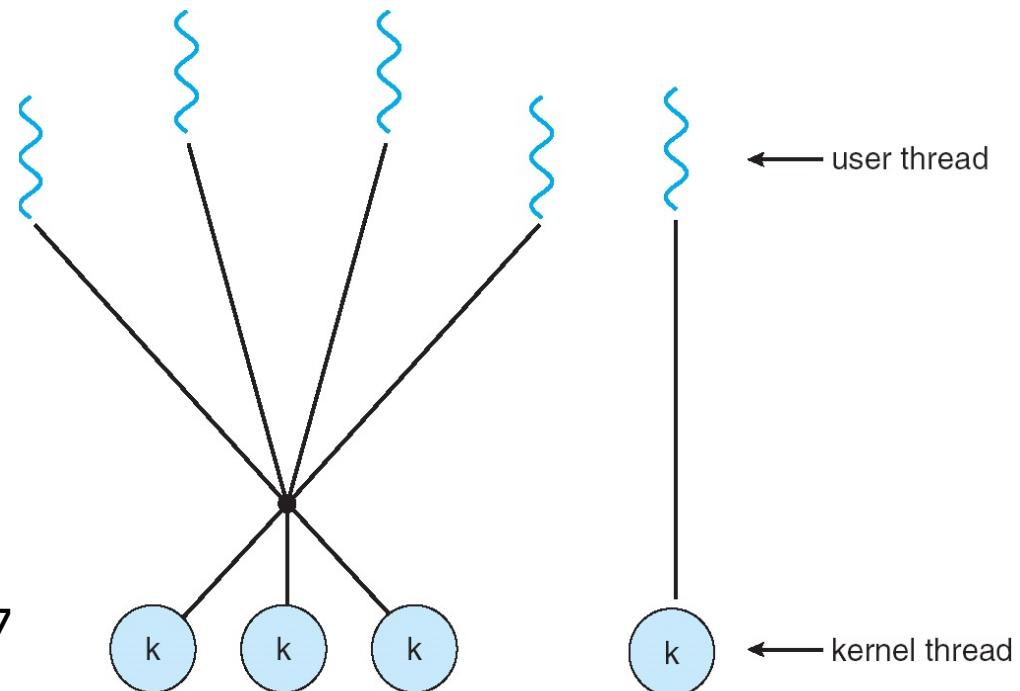
Two-level Multithreading Model

- Similar to Many-to-many model, except that it allows additionally to bind a user thread to a kernel thread.

- Allows an application to tell the OS which user-level threads needs always real concurrency.

- Examples:

- HP-UX,
 - Solaris prior to version 9.
 - Microsoft Windows \geq NT < 7 the *ThreadFiber* package.
 - Microsoft Windows \geq 7 with the *user-mode scheduling (UMS)* threads.



4.3 Thread Libraries

- Thread libraries provide programmers an API for thread handling.
 - May be implemented in user space (i.e. no system calls, no context switch – instead: simple local function calls between threads and thread library are used) or in kernel space (i.e. system calls to use kernel-level threads).
- Three main APIs for threads:
 - [POSIX Pthreads](#) (Portable library. May be a kernel- or user-level library.)
 - [Win32 threads](#) (MS Windows specific kernel-level library.)
 - [Java threads](#) (Java library. On POSIX systems implemented using Pthreads, on MS Windows using Win32 threads.)
 - (In the following, we will discuss Pthreads and Java threads...)

POSIX threads (Pthreads)

- A POSIX standard (IEEE 1003.1c) API for thread creation and synchronisation. (C language)
- API specifies interface and behaviour of the thread library.
 - It is the choice of the Pthread library developer how to implement POSIX Pthread API:
 - May be a kernel- or user-level library (i.e. various multithreading models).
 - (Pthread API is flexible enough to support both styles of threads. But: if Pthread library implementation is user-level, switching between threads can only occur when calls to the Pthread library are made – otherwise the thread library does not get control and cannot switch between threads.)
- Common in UNIX operating systems (Solaris, Linux, Mac OS X), but not Microsoft Windows.

Selected Pthread functions

- Create new thread and return its thread Id:

```
int pthread_create (pthread_t *thread,  
                    const pthread_attr_t *attr,  
                    void* (*start_routine) (void), void *arg)
```

Predefined data structure for thread Id
Desired thread attributes
Parameter of C function
Pointer to C function that will be executed in its own thread.

- Thread hands over control to thread library:

```
int pthread_yield () or  
int sched_yield ()
```

Must be called to switch to another thread if Pthreads are implemented as user-level threads.

- Thread terminates itself:

```
void pthread_exit (void *retval)  
(Alternatively: start_routine finishes)
```

Return value of Thread

- Wait for termination of a thread:

```
int pthread_join (pthread_t thread,  
                  void **thread_return)
```

ID of thread to wait for.
Pointer to memory location where return value of thread will be stored.

4.4 Java Thread API

- Threads provided by Java Virtual Machine and Java Thread API library.
 - Threads are part of the language: no import of Java API packages required.
- Each Java program comprises at least one thread for running the `main()` method.
- Additional Java threads may be defined by:
 - either extending class `Thread`.(Examples on next slides...)
 - or implementing interface `Runnable`.
 - However, this does not actually create or start a new thread:
 - Only by calling the method `start()` (provided by class `Thread`), a new thread is created and executes instructions contained in method `run()`.

Java Thread Creation (1)

Approach extending class **Thread**

- Example extending class **Thread**:

```
class MyThread extends Thread {  
    public void run() {  
        System.out.println("I Am a Further Thread");  
    }  
}  
  
class MainThread {  
    public static void main(String args[]) {  
        Thread runner = new MyThread();  
        runner.start();  
        System.out.println("I Am The Main Thread");  
    }  
}
```

Create instance of class for executing further thread.

Class must extend class **Thread**.

Code that shall be executed within a thread.

Start new thread.
Note: it is not allowed to call **start()** a second time! (For starting multiple threads, each thread must be started in its own instance of the **Thread** class.)

Each instance of class **Thread** provides just a thread control block for one thread.

Java Thread Creation (2)

Approach using interface **Runnable**

- If your class already inherits from another class, you may want to implement **interface Runnable** instead:

```
class MyThread2 implements Runnable {  
    public void run() {  
        System.out.println("I Am a Worker Thread ");  
    }  
}
```

Class must implement Runnable.

Code that shall be
executed within a thread.

```
class MainThread2 {  
    public static void main(String args[]) {  
        Runnable runner = new MyThread2();  
        Thread thread = new Thread(runner);  
        thread.start();  
        System.out.println("I Am The Main Thread");  
    }  
}
```

Instance of class
containing
`run()` method.

Create instance
of class for
executing
further thread.

Pass an
implementation
of interface
Runnable as
parameter.

Start new thread.
Note: it is not
allowed to call
`start()` a
second time!

Joining Threads

- Method **void join() throws InterruptedException** may be called on any instance of a class Thread to wait for the termination of that thread.

```
class JoinableThread extends Thread {  
    public void run() {  
        System.out.println("Worker working");  
    }  
}  
  
class MainThread3 {  
    public static void main(String[] args) {  
        Thread runner = new JoinableThread();  
        runner.start();  
        try {  
            runner.join();  
        } catch (InterruptedException ie) {  
            System.out.println("Interrupted while waiting thread");  
        }  
        System.out.println("Worker done");  
    }  
}
```

Someone may want to interrupt us while waiting the join() is waiting: this will throw an InterruptedException that always needs to be caught.

Wait for thread to terminate (i.e. run() method finishes/returns)

Excursion: Running Lambda Expressions inside Java Threads

- For those familiar with Lambda expressions introduced in Java (1.)8:
 - You can use a Lambda expression (right example) in order to avoid having to create a class containing the thread (left example):

```
class MyThread4 implements Runnable {    ...
    public void run() {
        System.out.println("I am a
Thread");
    }
}
...
Thread thread = new Thread(task);
...
Thread thread = new Thread(MyThread4);    thread.start();

thread.start();
```

- In this course, we will use the old-fashion style from slides 4-24 and 4-25.

The JVM and the host OS

- JVM is running on top of an OS and some libraries.
- JVM may decide to map each Java thread on a kernel-thread (one-to-one model), to user-level threads (many-to-one model), or the many-to-many model in-between.
- The Java standard leaves this decision to the particular implementation of the JVM.
 - Typically, if underlying OS supports kernel-threads, one-to-one model is used by a JVM implementation for that specific OS.

4.5 Implicit Threading

- Designing multi-threaded applications not trivial, e.g.
 - When to start a thread,
 - When to terminate a thread,
 - How to exchange data between threads.
- Instead of needing to create threads
- **Implicit threading:** Instead of letting an application developer writing code for creation and management of threading, let compilers and run-time libraries do this!

Implicit Threading: Thread Pools

- As discussed on slides 4-9 & 4-10, instead of creating a new thread for each request of a multithreaded server, a pool of worker threads that await work can be created in advance.
- Advantages:
 - Usually **slightly faster** to service a request with an existing thread from the pool than to create a new thread for each request (and terminate it again).
 - Allows to **restrict the number of parallel threads** in the server.
 - Otherwise, system resources (memory, CPU) might get exhausted and affect also other processes ("denial of service" attacks) if too many requests arrive.
- Disadvantage:
 - Management of thread pool more **difficult to implement**.
- Note: the Java package **java.util.concurrent** provides advanced threading support, including thread pools, i.e. **let that Java library do all the work needed** for thread pool management.
 - Not explained here, but if you need it: look in **java.util.concurrent**.

Issues mainly due to the fact that threads have been added late to Unix, when processes and process system calls did already exist.

4.6 Threading Issues

Semantics of Processes Management System Calls

- Consider the fork() system call that creates a copy of the calling process:
 - Calling process may have created multiple threads.
 - Design decision to make:
 - fork() copies all threads of that process,
 - fork() copies only thread that called fork().
 - Both variants of fork() semantics can be found in the different UNIX-based systems.
 - Sometimes, different system calls are offered to allow the application to decide on the semantics to be used.
 - E.g. if anyway exec() is called immediately after the fork to replace the instructions (& threads) of the child process, then it would make no sense to create copies of all the threads.

Threading Issues: Thread Cancellation

- Either a thread terminates itself, or it may be cancelled by another thread before it has finished.
 - Typically, only threads within the same process may cancel each other.
- Two general approaches when thread cancellation takes places:
 - **Asynchronous cancellation** terminates the target thread immediately. This may leave resources/data in an undefined state.
 - Requires intervention of OS, thus it is typically only available in kernel-level threads.
 - **Deferred cancellation** allows the target thread to check if it should be cancelled; if yes, it may decide to terminate in an orderly fashion.
 - When threads are implemented by a user-level library outside the OS kernel, typically only deferred cancellation is supported.
- Note:
 - “Asynchronous” usually refers to a decoupling of events with respect to time. You might wonder, why immediate cancellation is asynchronous?! Here, the term asynchronous refers rather to the fact that the target thread and the thread that tries to cancel the target thread are different threads.

Thread Cancellation in Java

- Asynchronous cancellation possible by calling `stop()` method on an instance of class Thread.
 - However, asynchronous cancellation may leave resources of the thread in an inconsistent state. (Thread is immediately cancelled, hence no chance for threads to do some clean-up before.)
 - Hence, using the `stop()` method is discouraged ("deprecated" =feature may not be anymore available in future Java versions).
- Deferred cancellation possible by calling `interrupt()` method on an instance of class Thread.
 - Thread itself can (periodically) check using `isInterrupted()` method (or by catching `InterruptedException`) whether someone requests its cancellation. If yes, the thread may terminate in an orderly fashion by releasing all resources and then simply terminates by leaving it's `run()` method.

Excursion: Signal Handling

- Signals are used in POSIX systems to notify a process that a particular event has occurred. (MS Windows has similar thing: Asynchronous Procedure Calls.)
- Reminder ch. 3: `kill (pid, SIGKILL)`
sends signal `SIGKILL` to process `pid`, thus terminating it. (Not all signals send via `kill()` do necessarily kill a process. E.g.: `SIGSTOP` puts a process to sleep, `SIGCONT` awakes process again.)
- A signal is called **synchronous if source and target are the same**, e.g. when a process does a division by zero, it will receive a signal to indicate this.
- A signal is called **asynchronous if source and target are different**, e.g. when one processes kills another process (or sends some other signal).
- If a signal occurs, it is processed by a **signal handler**:
 - Either default signal handler provided by operating system (typically just terminates the process)
 - Or user-defined signal handler (may release resources in an orderly fashion before shutting down).
 - POSIX system call `signal(int signum, sighandler_t handler)` instructs OS to call function `handler` when signal of type `signum` is received.

Threading Issues: Signal Handling and Threads

- How to deal with cases where several threads of a process install their own signal handler: to which thread shall a signal that is sent to a process be delivered? (A signal can only be sent to a process, but not to an individual thread – POSIX signals are older than POSIX threads.)
 - Option 1: Only one signal handler per process allowed. (Last thread that installs a signal handler wins.)
 - Option 2: Deliver the signal to every thread in the process.
 - Option 3: Deliver the signal to the first thread found that has registered a handler for the signal.
 - Option 4: Assign a specific thread to receive all signals for the process.
 - Further option, only applicable in cases of synchronous signals: Deliver the signal to the thread that triggered the signal.
- POSIX does not standardise these parts of threading/signal handling:
 - Different implementations of POSIX may handle this differently.

Threading Issues: Thread-specific Data (“Thread-safe” programming)

- All threads of a process share the same memory.

- Nice when threads want to use shared memory for communication.

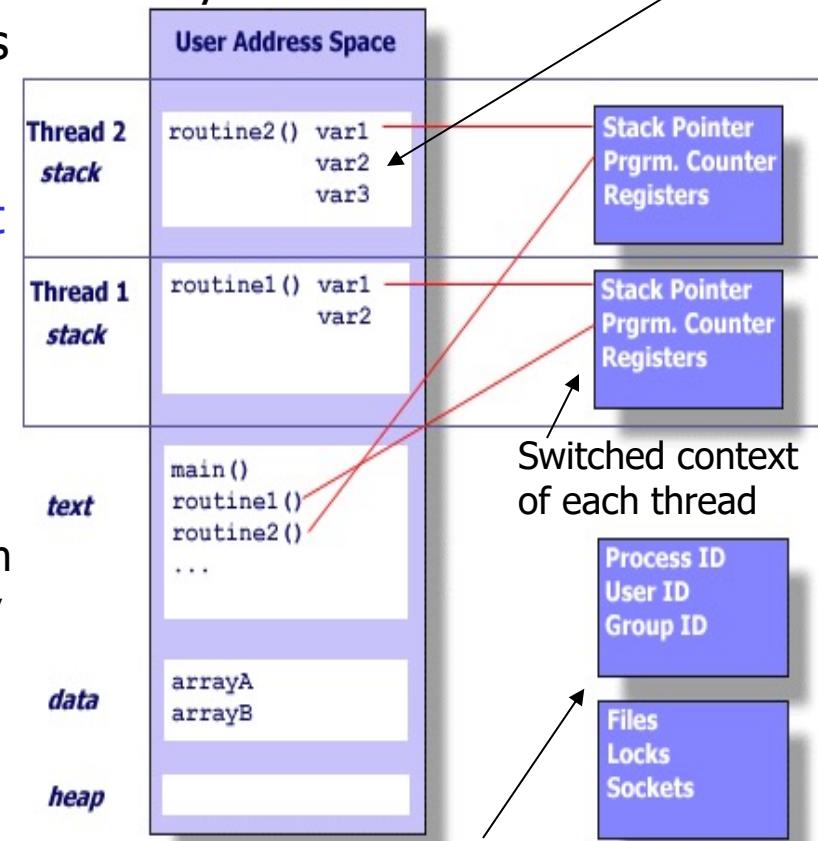
- However, when the same instructions are executed by several threads and each thread shall have its own data, some care is required to achieve that each thread uses its own data.

- Thread-safe programming using high-level programming languages:

- Use local variables instead of global variables: local variables are stored on the stack (each thread has its own stack), global variables are shared by all threads.

- If local variables are not sufficient, most thread-libraries offer calls for requesting a global memory area for thread-specific data (“thread-local storage”).

Stacks used to store, e.g., local variables of subroutines.



Threading Issues: Scheduler Activations

- Many-to-one multithreading model (user-level threads): thread must call a function of the user-level library to activate thread scheduler.
- One-to-one multithreading model: OS CPU scheduler is activated just as like for processes (i.e. by timer interrupt).
- Many-to-many and Two-level multithreading models:
 - Typically, all threads are user-level, but when a user-level thread is blocked, a further kernel-level thread needs to be created to execute one of the remaining non-blocked user-level threads:
 - Operating system informs user-level thread library when one of the threads is about to block ("upcall": from OS to process).
 - Upcall handler of user-level thread library then schedules/selects the next non-blocked user-level thread to be executed by a new kernel-level thread.

4.8 Summary

- Thread: flow of control within a process.
 - Multithreaded process: multiple flows of control within the same address space.
- User-level threads:
 - Provided by user-level library.
 - Only visible to programmer, unknown to the kernel.
 - Fast, but no real concurrency.
- Kernel-level threads:
 - Provided by kernel.
 - Slower, but full concurrency.
- Threading models:
 - Many-to-one: All user-level threads of a process are mapped to the single kernel-level thread of that process.
 - One-to-one: Each user-level thread maps to one kernel-level thread.
 - Many-to-many: Maps multiple user-level threads to a smaller or equal number of kernel-level threads.
- Multiple Thread APIs exist: POSIX Pthreads, Win32, Java Thread API.
- Implicit threading: describe the “what”, not the “how” (let library do the work).
- There may be issues, e.g. when using process-only concepts with threads.

Course
TÖL401G: Stýrikerfi /
Operating Systems
5. CPU Scheduling

Chapter Objectives

- Introduce CPU-scheduling, which is the basis for multiprogrammed operating systems.
- Introduce scheduling evaluation criteria.
- Describe various CPU-scheduling algorithms.
- Provide a glimpse into real-time scheduling and thread scheduling.
- Explain the issues related to multiprocessor/multicore scheduling and hyper-threading/hardware multithreading.
- Examine the scheduling algorithms of Microsoft Windows.

Contents

1. Basic Concepts
2. Scheduling Criteria
3. Scheduling Algorithms
4. Real-Time Scheduling
5. Thread Scheduling
6. Multiple-Processor Scheduling
7. Operating Systems Examples
8. Summary

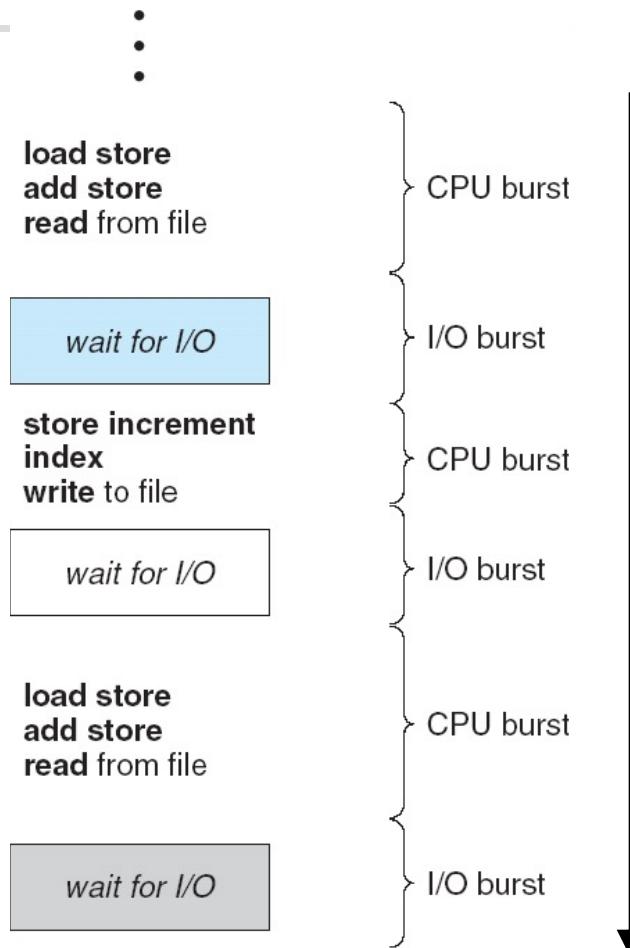
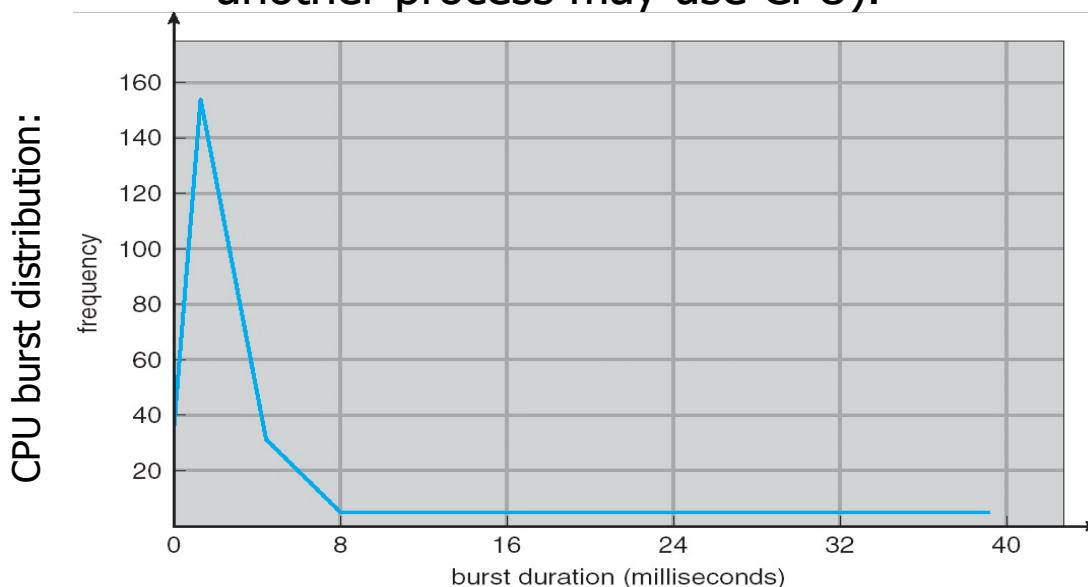
Note: The scheduling algorithms in Section 5.3 are presented slightly different than in the Silberschatz book. Furthermore, a section on real-time scheduling has been removed in newer book editions, but I kept one slide.

5.1 Basic Concepts of Scheduling: Motivation for scheduling

- Only one process/thread can run on a processor (or core) at a time.
 - All other processes have to wait until CPU (or a CPU core) is free and until they are scheduled to get CPU time.
- Two motivations for scheduling:
 - Multiprogramming (Batch) system: Maximise CPU utilisation and throughput of jobs.
 - While one process is blocked due to I/O, another process may use CPU.
 - Timesharing (Multitasking) system: Fast response time of all processes to allow users to work interactively.
 - While one process/thread is performing calculations, user can still interact with another process/thread because scheduler switches often between them.

Why Scheduling is Reasonable: CPU & IO Bursts

- Even though scheduling involves some overhead, it is effective to increase CPU utilisation:
 - CPU-I/O burst cycle:** Process execution typically consists of a cycle of CPU usage and subsequent I/O wait (during which another process may use CPU).



Definitions: Scheduler, Scheduling algorithm, Dispatcher

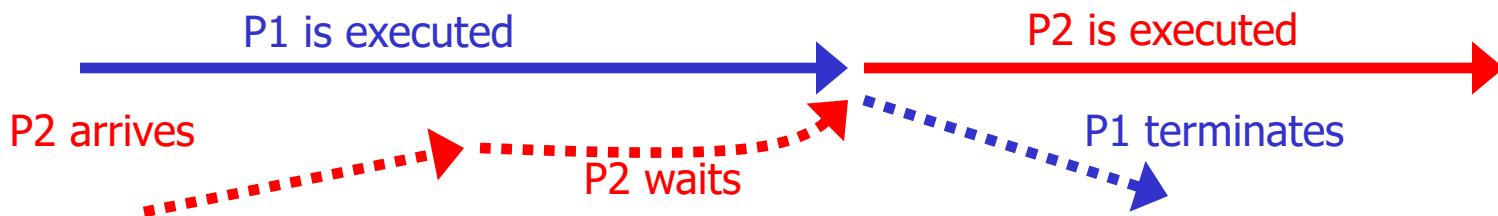
- **(CPU) Scheduler:**
 - Part of the OS kernel that assigns CPU time to processes/threads that are ready to execute.
- **Dispatcher:**
 - Part of the OS kernel that performs the actual context switch (restoring CPU registers of process, switching from kernel to user mode, resume execution of process)
 - **Dispatch latency:** time required for context switch (typically: $30\mu s$).
- **Scheduling algorithm:**
 - Algorithm that is used by scheduler to decide which process/thread gets the CPU for how long.

Some Remarks

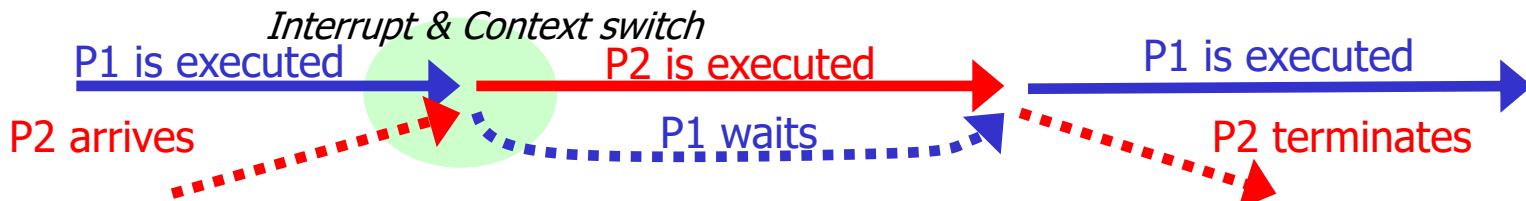
- Reminder from Section 2.6: Separate policy from mechanism!
 - Mechanism: Scheduler (& Dispatcher),
 - Policy: Scheduling algorithm.
- There is no “best” scheduling algorithm. It rather depends on the type of system (e.g. batch, multitasking, real-time) and scenarios.
- In Section 3.2 we also identified mid- and long-term schedulers: In this chapter, we will only consider short-term schedulers/CPU scheduling.
 - (Well, some can be used for long-term scheduling as well.)
- In operating systems with kernel-level threads, only threads (not processes) are scheduled. (Each process has at least one thread.)
 - In the following, we do not distinguish between processes and threads: “process scheduling” refers to threads as well.

Preemptive vs. Nonpreemptive Scheduling

- **Nonpreemptive:** CPU is allocated to process until it blocks or terminates.
 - Timesharing only possible if CPU-bound processes are **cooperative** and make calls to `yield()` system call (i.e. voluntary transition from running to ready state) to hand over control to scheduler. (Examples: MS Windows ≤3.x, Mac OS ≤9.x)



- **Preemptive:** Process may be interrupted, e.g. timesharing: time slice expired.
 - Problem: Process may be interrupted while updating data shared between processes: synchronisation between cooperating processes required to avoid inconsistencies.
 - Time slice timer may even expire while kernel code is executed: kernel must, e.g., disable interrupt processing while updating critical kernel data structures.



5.2 Scheduling Criteria (1)

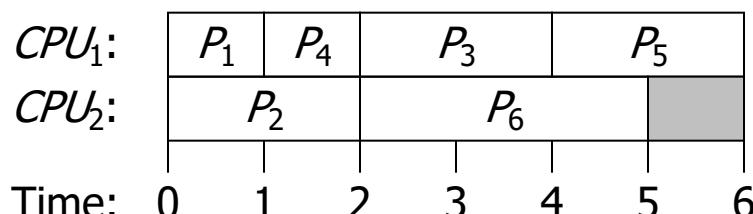
- Requirements on scheduling for **all types of systems**:
 - **Fairness**:
 - Each process gets CPU time (no starvation of processes).
 - **Enforcement of priorities**:
 - Processes with higher priorities are preferred.
 - **Balance**:
 - All the different resources of a system are reasonably utilised.
- Requirements on scheduling for multiprogramming (**Batch systems**):
 - **CPU utilization**:
 - Maximise CPU utilization: keep the CPU as busy as possible.
 - **Throughput**:
 - Maximise number of processes that complete their execution per time unit.
 - **Turnaround time**:
 - Minimise amount of time (from start to termination) to execute a particular process.

Scheduling Criteria (2)

- Requirements on scheduling for timesharing (multitasking) systems, i.e. **interactive systems**:
 - **Response time**:
 - Minimise amount of time it takes from when a request was submitted until the first response is produced.
- Requirements on scheduling for **real-time systems**:
 - **Meeting deadlines**:
 - Meet time constraints: processes (or events within) that must be started/finished until a certain point in time must be preferred.
 - **Predictability**:
 - As long as the system is not overloaded (beyond an allowed level), it can be predicted, when a certain process (or event within) is executed.

Definitions (1)

- Schedule:
 - A schedule S for a set of cores/processors $CPU = \{CPU_1, CPU_2, \dots, CPU_m\}$ and a set of processes (tasks) $P = \{P_1, P_2, \dots, P_n\}$ (additionally, there might be dependencies between P_1, P_2, \dots, P_m defined) with required service time d_1, d_2, \dots, d_m is a mapping of processes to processors (or cores) with respect to time.
 - Note: in the following, we consider only service time that is provided by the CPU and do not talk about time spent while waiting due to blocking I/O.
- Visualisation of schedules for $m = 2$ CPUs (cores) using a Gantt chart:

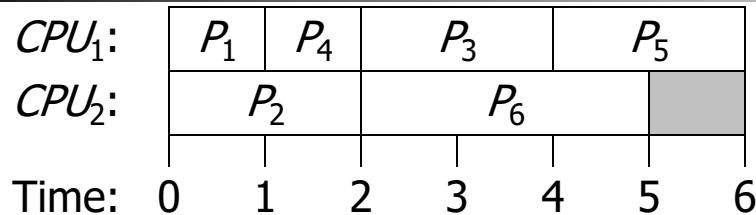


- The length $\lambda(S)$ of a schedule S is the time difference between the departure (finish time) of the final process and the arrival of the first process. (In the above example: $\lambda(S) = 6$).

Definitions (2)

- Different points in time and durations for a process P_i :
 - a_i : Arrival time (when process is created/enters system)
 - d_i : Service time (duration of process if it would have CPU exclusively to do its calculations)
 - s_i : Start time (when process gets CPU for the first time)
 - f_i : Finish time (when process is completed/leaves system)
 - $r_i = f_i - a_i = w_i + d_i$: Residence time (Turnaround time) (time in system)
 - $w_i = r_i - d_i$: Waiting time (how long process is in system without being serviced)
- $\tilde{W} = \frac{1}{n} * \sum_{i=1}^n w_i$: Average waiting time for n processes
- $\tilde{R} = \frac{1}{n} * \sum_{i=1}^n r_i$: Average residence time for n processes

Application of Definitions: Examples

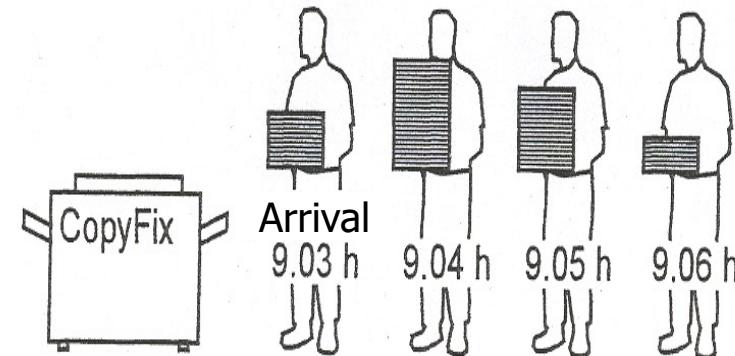


Process	Arrival time	Service time	Start time (= Waiting time, because nonpreemptive)	Residence time
P_1	$a_1 = 0$	$d_1 = 1$	$s_1 = w_1 = 0$	$r_1 = w_1 + d_1 = 1$
P_2	$a_2 = 0$	$d_2 = 2$	$s_2 = w_2 = 0$	$r_2 = w_2 + d_2 = 2$
P_3	$a_3 = 0$	$d_3 = 2$	$s_3 = w_3 = 2$	$r_3 = w_3 + d_3 = 4$
P_4	$a_4 = 0$	$d_4 = 1$	$s_4 = w_4 = 1$	$r_4 = w_4 + d_4 = 2$
P_5	$a_5 = 0$	$d_5 = 2$	$s_5 = w_5 = 4$	$r_5 = w_5 + d_5 = 6$
P_6	$a_6 = 0$	$d_6 = 3$	$s_6 = w_6 = 2$	$r_6 = w_6 + d_6 = 5$

- Average Waiting time: $\tilde{W} = \frac{1}{n} * \sum_{i=1}^n w_i = \frac{1}{6} * (0 + 0 + 2 + 1 + 4 + 2) = 1,5$
- Average Residence time: $\tilde{R} = \frac{1}{n} * \sum_{i=1}^n r_i = \frac{1}{6} * (1 + 2 + 4 + 2 + 6 + 5) = 3,3$

5.3 Scheduling Algorithms: First Come First Served (FCFS)

- Principle:
 - Processes get CPU allocated in order of their arrival.
 - Running processes are not interrupted.
- Properties:
 - Nonpreemptive,
 - Fair (Finally, each process gets CPU),
 - Easy to implement.
- Remark:
 - Waiting times of each process depends on order of arrival.
 - Average waiting time may differ quite a lot when comparing best case and worst case arrival scenarios.
- General comment applying to all of the following algorithms: none of them waits until all processes have arrived, but immediately start with their first scheduling decision once a process arrived.



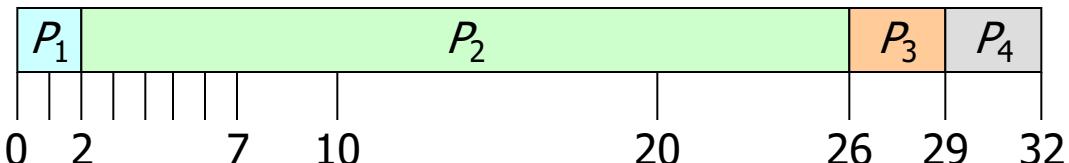
Scheduling Algorithms: First Come First Served (FCFS)

- Example scenario used on next slides
(also used to illustrate the further scheduling algorithms):
 - 4 processes,
 - 1 CPU (core),
 - No dependencies between processes.

Process	Arrival time	Service time
P_1	$a_1 = 0$	$d_1 = 2$
P_2	$a_2 = 1$	$d_2 = 24$
P_3	$a_3 = 2$	$d_3 = 3$
P_4	$a_4 = 7$	$d_4 = 3$

Scheduling Algorithms: First Come First Served (FCFS)

- Schedule with FCFS:



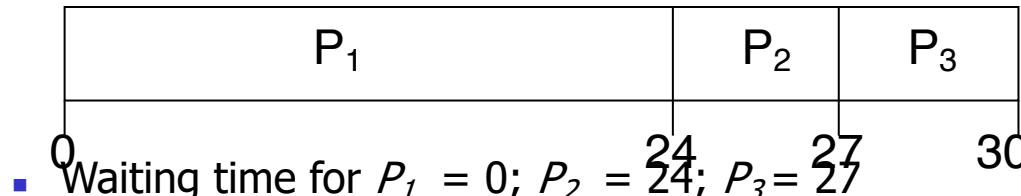
- As table:

Process	Arrival time	Service time	Start time	Waiting time	Residence time
P_1	$a_1 = 0$	$d_1 = 2$	$s_1 = 0$	$w_1 = s_1 - a_1 = 0$	$r_1 = w_1 + d_1 = 2$
P_2	$a_2 = 1$	$d_2 = 24$	$s_2 = 2$	$w_2 = s_2 - a_2 = 1$	$r_2 = w_2 + d_2 = 25$
P_3	$a_3 = 2$	$d_3 = 3$	$s_3 = 26$	$w_3 = s_3 - a_3 = 24$	$r_3 = w_3 + d_3 = 27$
P_4	$a_4 = 7$	$d_4 = 3$	$s_4 = 29$	$w_4 = s_4 - a_4 = 22$	$r_4 = w_4 + d_4 = 25$

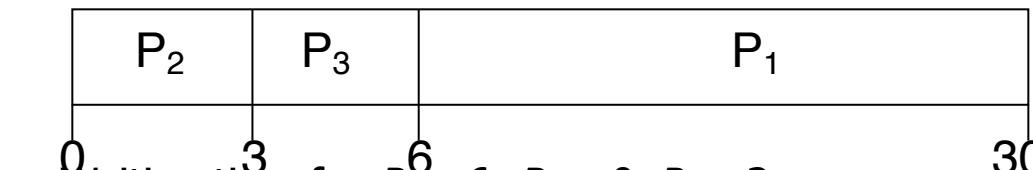
- Average waiting time $\tilde{W} = \frac{1}{n} * \sum_{i=1}^n w_i = \frac{1}{4} * (0 + 1 + 24 + 22) = 11,75$
- Average residence time $\tilde{R} = \frac{1}{n} * \sum_{i=1}^n r_i = \frac{1}{4} * (2 + 25 + 27 + 25) = 19,75$

Scheduling Algorithms: First Come First Served (FCFS)

- Waiting times (and thus also residence times) depend very much on order of process arrival!
- Example: $d_1=24$, $d_2=3$, $d_3=3$, two different orders of process arrival:
 - Processes arrive at $a_i=0$ in the order: P_1, P_2, P_3



- Average waiting time: $(0 + 24 + 27)/3 = 17$
- Processes arrive at $a_i=0$ in the order: P_2, P_3, P_1

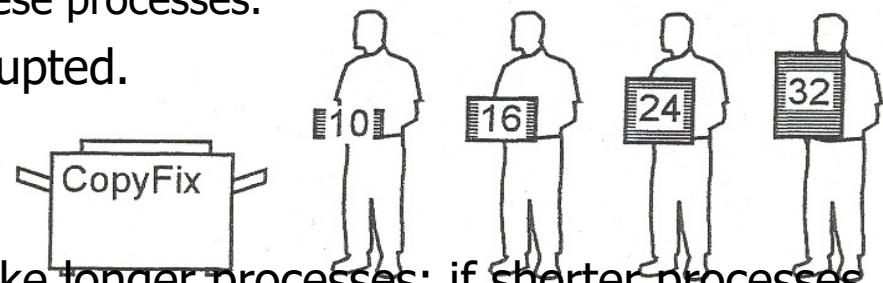


- Waiting time for $P_1 = 6$; $P_2 = 0$, $P_3 = 3$
- Average waiting time: $(6 + 0 + 3)/3 = 3$

This insight leads us to the scheduling algorithm on the next slide...

Scheduling Algorithms: Shortest Job First (SJF)

- **Principle:**
 - Considers service time of process: process with shortest service time gets CPU first.
 - Same service times: FCFS for these processes.
 - Running processes are not interrupted.
- **Properties:**
 - Nonpreemptive.
 - Unfair (short process may overtake longer processes: if shorter processes keep arriving all the time, longer process will suffer from starvation, i.e. will never get serviced.)
- **Remark:**
 - Optimises (provable) average waiting time of processes (nonpreemptive).
 - However, problems remain:
 - Unfair,
 - Service time must be known in advance.

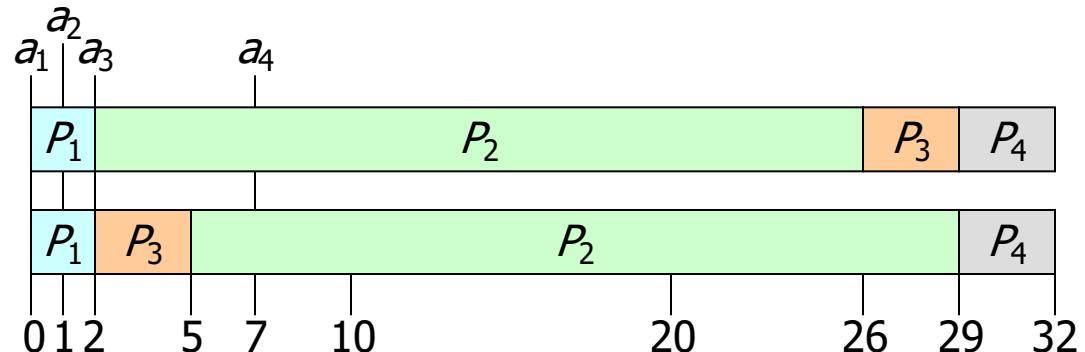


Excursion: How to Know Service Time of a Process in Advance?

- Typically, service time of a process is not known in advance.
 - At least, a scheduler cannot predict.
- However, **in batch systems** (e.g. supercomputers running jobs over night), human submitters of batch jobs often need to specify a time limit for their job.
 - **SJF can be used** (and is often used) **for long-term scheduling** (job scheduling) in batch systems.
 - User's motivation for estimating time limit as good as possible:
 - If specified time limit is too low, job might exceed time limit and gets therefore terminated (to detect hanging jobs).
 - If specified time limit is too high, job will have to wait long until SJF will give it CPU time.
- Later on, we will see an example how to predict the length of CPU bursts to use this for SJF-like short-term scheduling.

Scheduling Algorithms: Shortest Job First (SJF)

- For comparison:
Schedule with FCFS:
- Schedule with SJF:
- Table:

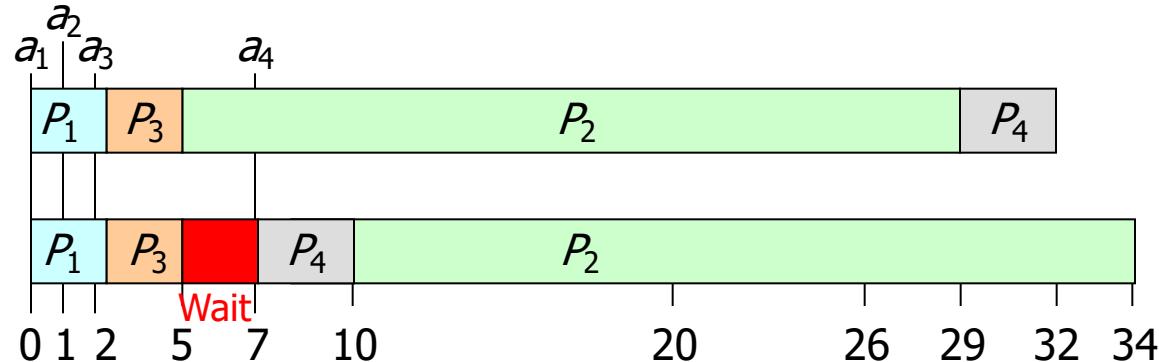


Process	Arrival time	Service time	Start time	Waiting time	Residence time
P_1	$a_1 = 0$	$d_1 = 2$	$s_1 = 0$	$w_1 = s_1 - a_1 = 0$	$r_1 = w_1 + d_1 = 2$
P_2	$a_2 = 1$	$d_2 = 24$	$s_2 = 5$	$w_2 = s_2 - a_2 = 4$	$r_2 = w_2 + d_2 = 28$
P_3	$a_3 = 2$	$d_3 = 3$	$s_3 = 2$	$w_3 = s_3 - a_3 = 0$	$r_3 = w_3 + d_3 = 3$
P_4	$a_4 = 7$	$d_4 = 3$	$s_4 = 29$	$w_4 = s_4 - a_4 = 22$	$r_4 = w_4 + d_4 = 25$

- Average waiting time $\widetilde{W} = \frac{1}{n} * \sum_{i=1}^n w_i = \frac{1}{4} * (0 + 4 + 0 + 22) = 6,5$
- Average residence time $\widetilde{R} = \frac{1}{n} * \sum_{i=1}^n r_i = \frac{1}{4} * (2 + 28 + 3 + 25) = 14,5$

Scheduling Algorithms: Shortest Job First (SJF) with Waiting

- Paradox, if SJF decides to do nothing (waiting):
- Schedule with SJF:
- Schedule with SJF & waiting:
- Table:



Process	Arrival time	Service time	Start time	Waiting time	Residence time
P_1	$a_1 = 0$	$d_1 = 2$	$s_1 = 0$	$w_1 = s_1 - a_1 = 0$	$r_1 = w_1 + d_1 = 2$
P_2	$a_2 = 1$	$d_2 = 24$	$s_2 = 10$	$w_2 = s_2 - a_2 = 9$	$r_2 = w_2 + d_2 = 33$
P_3	$a_3 = 2$	$d_3 = 3$	$s_3 = 2$	$w_3 = s_3 - a_3 = 0$	$r_3 = w_3 + d_3 = 3$
P_4	$a_4 = 7$	$d_4 = 3$	$s_4 = 7$	$w_4 = s_4 - a_4 = 0$	$r_4 = w_4 + d_4 = 3$

- Average waiting time
- Average residence time

$$\tilde{W} = \frac{1}{n} * \sum_{i=1}^n w_i = \frac{1}{4} * (0 + 9 + 0 + 0) = 2,25$$

$$\tilde{R} = \frac{1}{n} * \sum_{i=1}^n r_i = \frac{1}{4} * (2 + 33 + 3 + 3) = 10,25$$

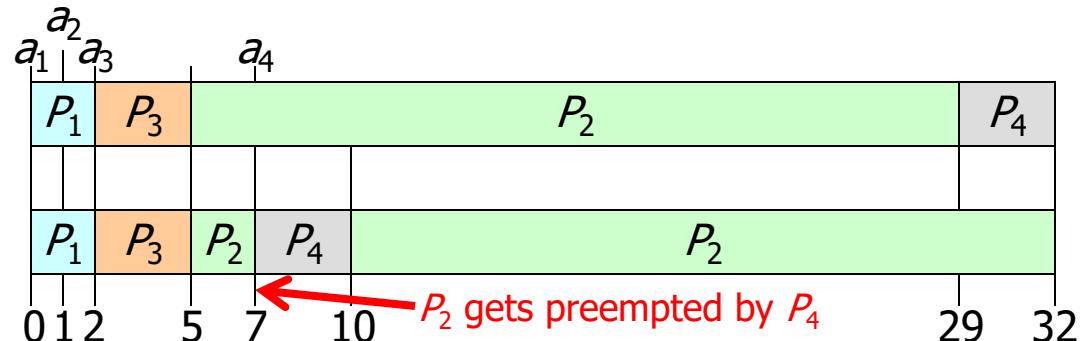
Got shorter!
(However,
requires
global
knowledge
to know
future.)

Scheduling Algorithms: Shortest Remaining Time First (SRTF)

- Principle:
 - At any time, the process with the shortest remaining time gets the CPU.
 - Preemptive variant of SJF (based on insight from the paradox on previous slide): When a new process arrives and service time of that process is shorter than the remaining service time of the running process, the running process will be preempted and the just arrived process gets the CPU.
 - Running processes may be interrupted.
- Properties:
 - Preemptive.
 - Unfair (short process may overtake longer processes: if shorter processes keep arriving all the time, longer process will suffer from starvation.)
- Remark:
 - Optimises (provable) average waiting time of processes (preemptive).
 - However, problems remain:
 - Unfair,
 - Service time must be known in advance.

Scheduling Algorithms: Shortest Remaining Time First (SRTF)

- For comparison:
Schedule with SJF
(without waiting):
- Schedule with SRTF
- Table:



Process	Arrival time	Service time	Start time	Waiting time	Residence time
P ₁	a ₁ = 0	d ₁ = 2	s ₁ = 0	w ₁ = 0	r ₁ = 2
P ₂	a ₂ = 1	d ₂ = 24	s ₂ = 5	w ₂ = 4 + 3 = 7 (preempted by P ₄)	r ₂ = 31
P ₃	a ₃ = 2	d ₃ = 3	s ₃ = 2	w ₃ = 0	r ₃ = 3
P ₄	a ₄ = 7	d ₄ = 3	s ₄ = 7	w ₄ = 0	r ₄ = 3

- Average waiting time
- Average residence time

$$\tilde{W} = \frac{1}{n} * \sum_{i=1}^n w_i = \frac{1}{4} * (0 + 7 + 0 + 0) = 1,75$$

$$\tilde{R} = \frac{1}{n} * \sum_{i=1}^n r_i = \frac{1}{4} * (2 + 31 + 3 + 3) = 9,75$$

Scheduling Algorithms: Interactive Timesharing Systems

- The previously discussed scheduling algorithms are mainly applicable for batch systems:
 - Processes run from start to completion and are typically not interrupted (except for I/O or – in case of SRTF – when a new process arrives.)
- To support **interactive systems** that give the user the impression that multiple processes are running in parallel, **preemptive scheduling algorithms** are required:
 - CPU time is divided into slices that are periodically allocated to processes based on criteria that are specific to each scheduling algorithm (→following slides).
 - A keypress will be delivered after a couple of time slices to a process waiting for it even if all other processes are only using the CPU.

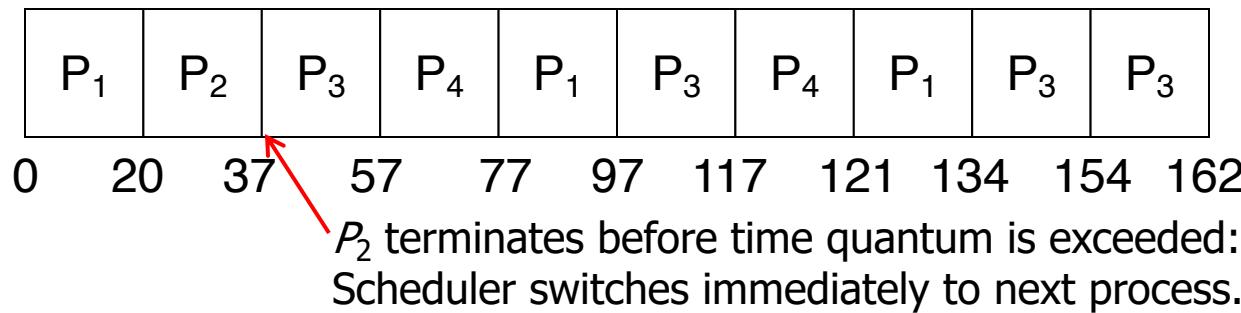
Scheduling Algorithms: Round-Robin (RR) Scheduling

- Principle:
 - Service time is divided into **time slices** (with a maximum duration of a **time quantum** – actual slice may be shorter, e.g. if process does I/O → 5-5).
 - Ready queue is a circular FIFO buffer (like FCFS, but circular): scheduler assigns time slice to process at head of ready queue.
 - Either: process makes blocking I/O before time slice expires: process is put at the end of the waiting queue.
 - Or: CPU burst of process is longer than time slice (timer interrupt preempts process): process is put at the end of the ready queue.
 - Newly arriving processes are put at the end of the ready queue.
- Properties:
 - Preemptive.
 - CPU time is equally distributed to **all processes (fair)**.

Otherwise, old process would suffer from starvation
if constantly new processes arrive.
- Remark:
 - Variations possible with respect to time quantum:
 - Long time quantum (> 100ms): slow interactions. (E.g.: 10 processes or users, and 100ms time slices: User has to wait 1s for next interaction with process.)
 - Short time quantum (< 10ms): large overhead due to frequent context switches.

Scheduling Algorithms: Round-Robin (RR) Scheduling

- Example (different scenario than before):
 - Service times: $d_1=53$, $d_2=17$, $d_3=68$, $d_4=24$
 - Processes arrive all at $t=0$ in the order: P_1, P_2, P_3, P_4
 - Schedule with RR and a time quantum of 20:

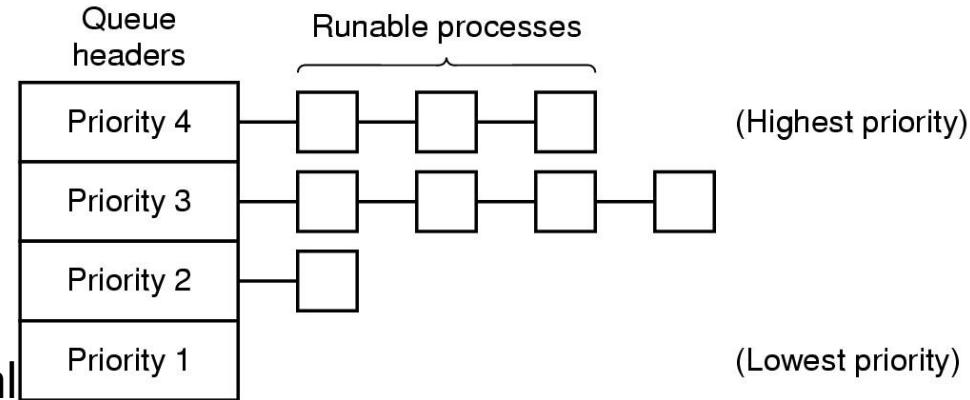


Process	Arrival time	Service time	Start time	Waiting time	Residence time
P_1	$a_1 = 0$	$d_1 = 53$	$s_1 = 0$	$w_1 = 0 + 57 + 24 = 81$	$r_1 = w_1 + d_1 = 134$
P_2	$a_2 = 0$	$d_2 = 17$	$s_2 = 20$	$w_2 = 20$	$r_2 = w_2 + d_2 = 37$
P_3	$a_3 = 0$	$d_3 = 68$	$s_3 = 37$	$w_3 = 37 + 40 + 17 = 94$	$r_3 = w_3 + d_3 = 162$
P_4	$a_4 = 0$	$d_4 = 24$	$s_4 = 57$	$w_4 = 57 + 40 = 97$	$r_4 = w_4 + d_4 = 121$

Scheduling Algorithms: Round-Robin with Priorities

■ Principle:

- Processes have priorities and each priority has its own ready queue.
- First, all processes from ready queue with highest priority are scheduled using Round Robin al
- Only if queue is empty (because all processes terminated or are blocked), the ready queue with the next lower priority is served and so on.
 - However, even if a higher priority process arrives, a currently running lower priority process gets not preempted, until its time slice expires anyway.

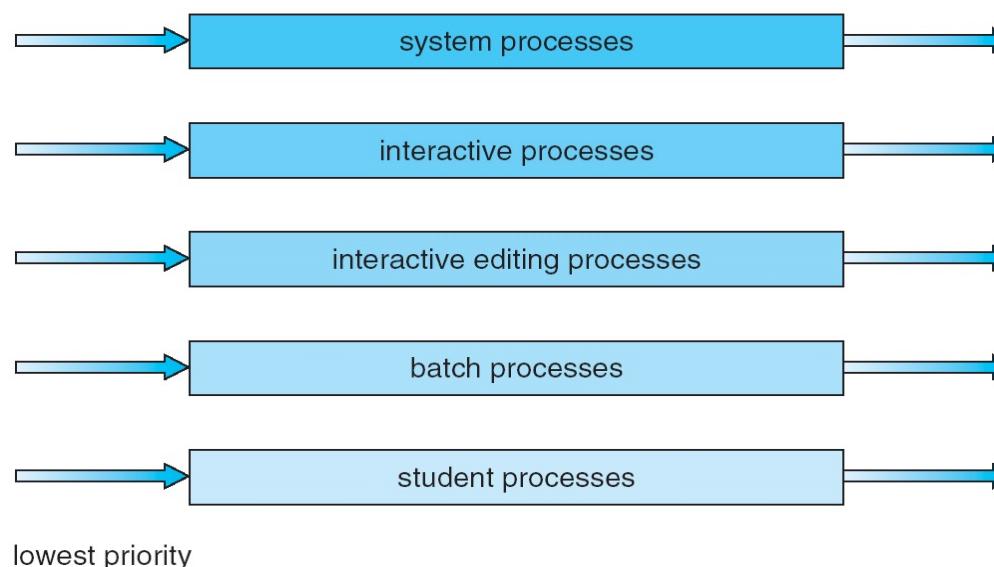


■ Properties:

- Low-priority processes may suffer from **starvation**: if new high priority processes are arriving steadily (or do never block or terminate), then low priority processes will never get executed.
- Solution: **Aging**: priorities of processes are dynamically adjusted:
 - Priority decreases with increasing residence time.
 - Priority increases with increasing waiting time.

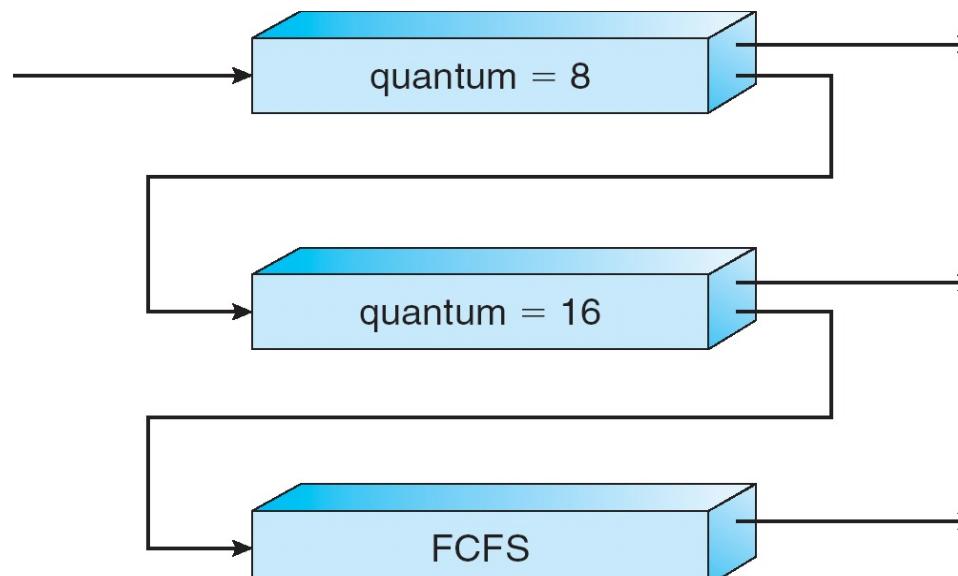
Scheduling Algorithms: Multilevel Queue Scheduling

- Principle:
 - Different queues have different scheduling algorithms, e.g.
 - queues for interactive foreground processes using RR scheduler.
 - queues for batch background processes using FCFS scheduler.
- Scheduler between queues required: e.g. foreground queue has absolute priority: no background process will run unless other queues are empty.



Scheduling Algorithms: Multilevel Feedback Queue Scheduling

- Principle:
 - Like Multilevel Queue Scheduling, however processes can move between the various queues.
 - Aging can be implemented this way:
 - if a process uses too much CPU time, it will move to a lower-priority queue.
 - if a process waits for long time, it will move to a higher-priority queue.



5.5 Thread Scheduling

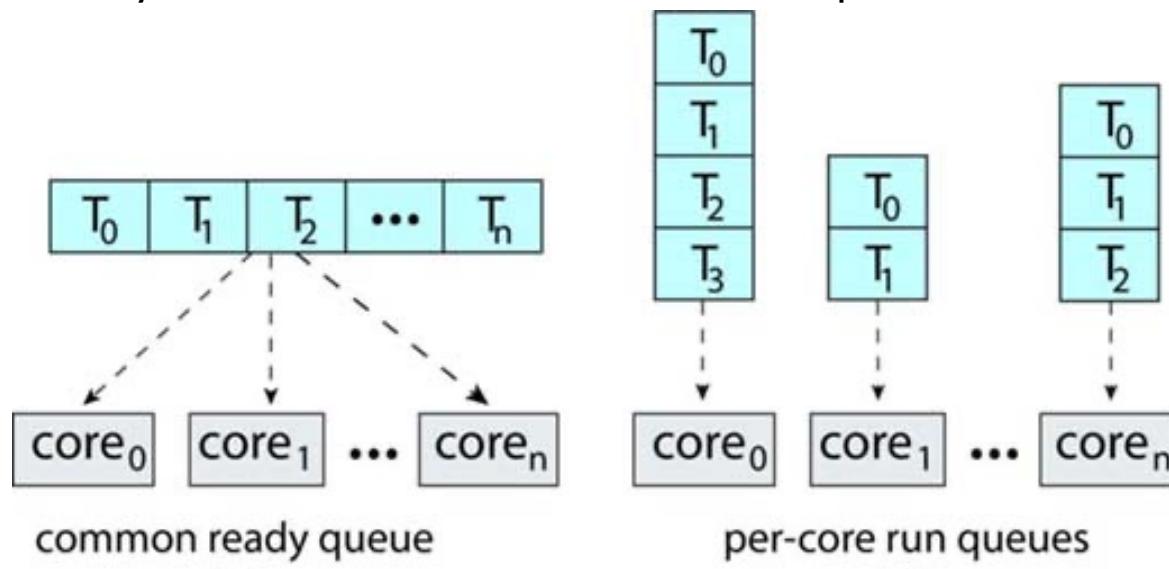
- If only **user-level threads** are used, the OS kernel is not aware of these threads, but simply schedules the processes.
 - Scheduling of user-level threads takes place within user-level thread library.
 - Threads of the same process are competing for CPU time that has been allocated by the OS scheduler to that process:
process-contention scope (PCS).
- However, if **kernel-level threads** are used, the OS kernel basically schedules threads, not processes.
 - Typically, OS scheduler does not care to which process all the threads belong. Instead, all threads of the system compete equally for CPU time:
system-contention scope (SCS).
 - Most OS that use the one-to-one multithreading model, use SCS.

5.6 Multiple-Processor Scheduling/ Multi-core Scheduling

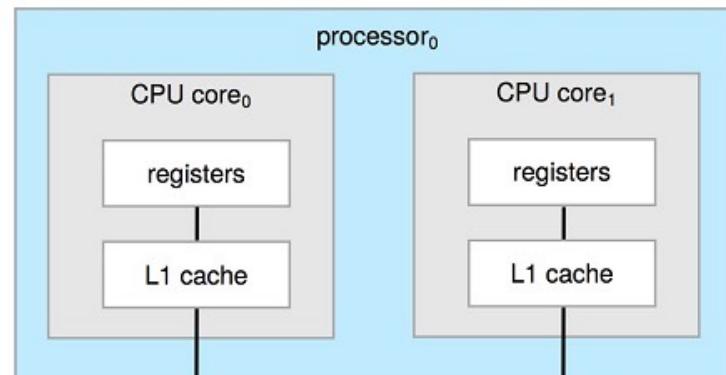
- CPU scheduling more complex when multiple CPUs/cores that share the physical memory are available.
 - Each CPU/core must be managed by the operating system.
- **Asymmetric multiprocessing:** only one master processor/core accesses the kernel data structures (e.g. scheduler queues) and makes scheduling decisions, the others are waiting to get work assigned.
 - Master processor runs fully-fledged kernel,
 - Slave processors only minimal kernel.
- **Symmetric multiprocessing (SMP):** all processors/cores are running the same kernel. I.e. each processor/core executes scheduler instructions and selects ready processes to be executed.
 - Nowadays, SMP used by all major operating systems.

Symmetric Multiprocessing and Scheduler Queues

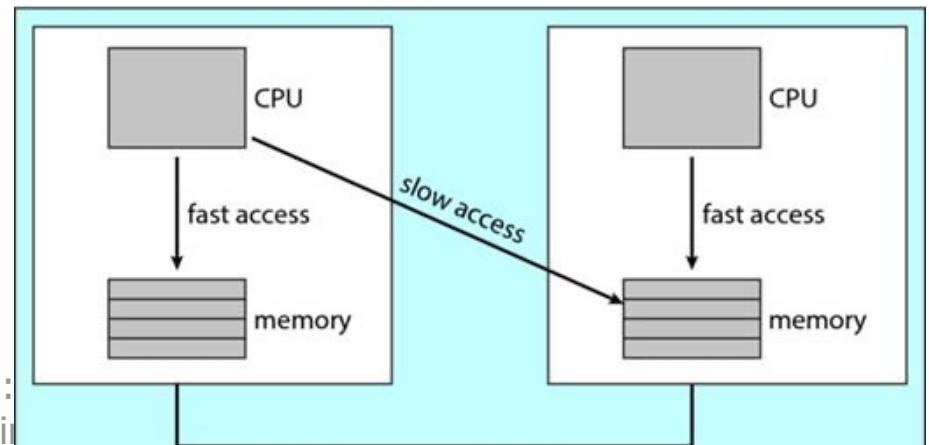
- Two ways of organising scheduler queues in SMP:
 - One common ready queue for all processors/cores:
 - A process from the ready queue may be run by any of the available CPUs/cores.
 - All processors/cores maintain concurrently the shared kernel data structures.
 - Synchronisation (\rightarrow ch. 6) required to avoid inconsistencies of kernel data structures.
 - Each processor/core has its own ready queue:
 - Queues are “private” to each CPU/core.
 - No synchronisation needed for access to queues needed.



Processor Affinity



- Each CPU/core has its own cache containing recently used instructions/data.
 - If the scheduler assigns a process to a CPU/core that has previously executed a different process, that chosen CPU/core will not have the instructions and data of the newly assigned process in its cache.
 - \Rightarrow No benefit (=no speedup) from caching if a process is migrated to a different processor (core) at each scheduling decision.
 - **Processor affinity**: scheduler tries to keep a process on the same processor/core.
- Even worse in **Non-Uniform Memory Access (NUMA)** multiprocessor systems:
 - Each CPU of a multiprocessor computer has its own RAM: fast access.
 - Still a CPU can access the RAM of the other CPU, but slower.
 - NUMA used in supercomputing:
 - One motherboard contains two multicore CPUs with fast access to its RAM.
 - Shared memory communication possible by accessing RAM of other CPU (=slower than own RAM, but still faster than communication via network).

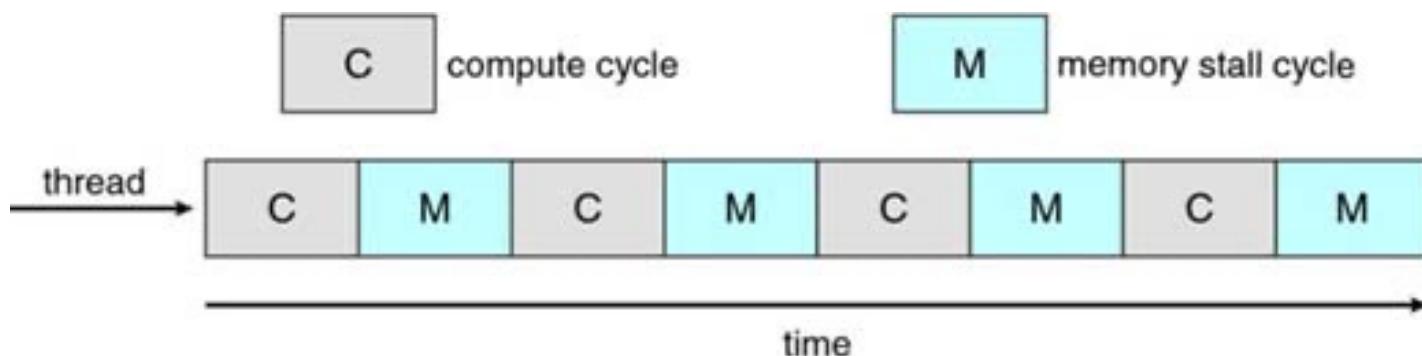


Processor Affinity vs. Load Balancing

- Hard processor affinity:
 - Process migrates never between CPUs/cores
 - E.g. when each CPU/core has its own, “private” ready queue.
 - Disadvantage: some CPUs/cores may be busy executing processes while others are idle even though ready processes are waiting in a queue.
- Load balancing attempts to distribute the workload between CPUs/cores.
 - Push migration checks periodically load on each processor/core and migrates (“pushes”) processes if necessary.
 - Pull migration is performed whenever a processor/core is idle (=queue becomes empty). Then, the idle processor/core pulls a process from another processor’s/core’s queue.
- Processor affinity and load balancing contradict each other.
 - Difficult to develop scheduling algorithms that achieve a good compromise.
 - Soft processor affinity: try to maintain affinity, but allow load balancing.

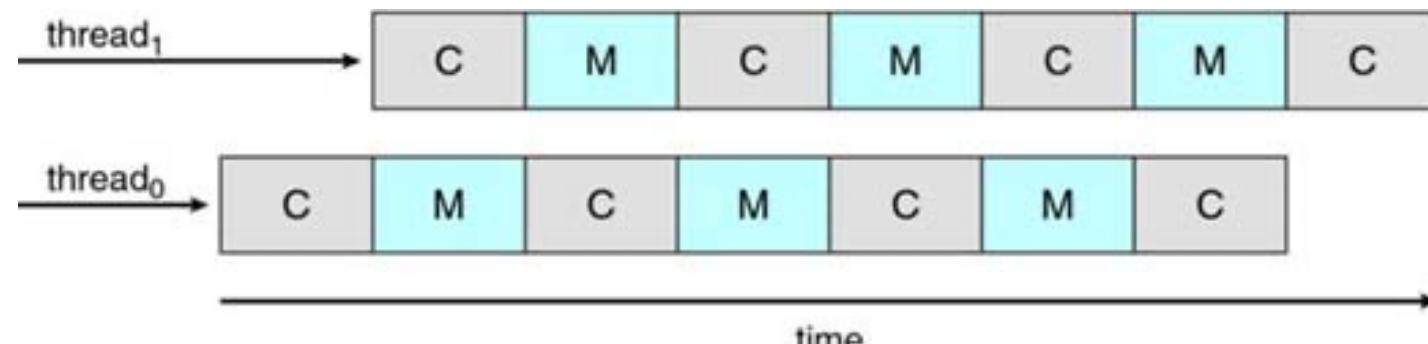
Memory stalls

- On OS-level, processes get blocked if they are waiting for I/O
 - Same may happen on CPU level:
 - Accessing main memory is slow (in comparison to a CPU cycle).
 - While a machine code instruction is waiting for RAM access ("memory stall"), the CPU is idle.
 - In practise, a CPU may be waiting approx. 50% of the time for RAM access.
 - Countermeasures:
 - Bigger caches inside CPU (=faster than RAM),
 - "Hyperthreading" →next slide.



“Hyper-Threading”/Hardware multithreading/ Simultaneous Multi Threading (SMT)/Chip multithreading (CMT)

- Just like OS-level, when a process is blocked because waiting for I/O and OS scheduler selects another ready process to execute instead
 - Do the same on CPU level:
- “Hyper-Threading” (Intel marketing term), also known as Hardware multithreading, Simultaneous Multi Threading (SMT), or Chip multithreading:
 - A CPU core gives the impression that it is actually two (or even more) CPU cores.
 - In fact, it is only one physical core that is just able to switch to another thread of instructions in case of a memory stall, so that one physical core actually executes two threads by pretending to be two logical cores.
 - For switching, CPU contains some sort of “scheduler” in the CPU hardware, e.g. save state (e.g. CPU registers) of hardware thread inside CPU hardware.



“Hyper-Threading” and Scheduling on Multicore Processors

- Problem: Even the OS may not be aware of hyper-threading and treats logical cores just like real physical cores.
 - Problem, e.g. dual core processor providing four logical cores, but only two processes are currently running:
 - OS CPU scheduler might accidentally choose to schedule these two processes on those two logical cores that belong to the same physical core.
 - That physical core would become extremely busy while the other physical core would be idle, leading to slower speed than is possible with better scheduling.
- ⇒ Make OS scheduler aware of hyper-threading!
 - So that the OS scheduler can do load balancing based on physical cores, not logical cores.

5.7 Operating System Examples

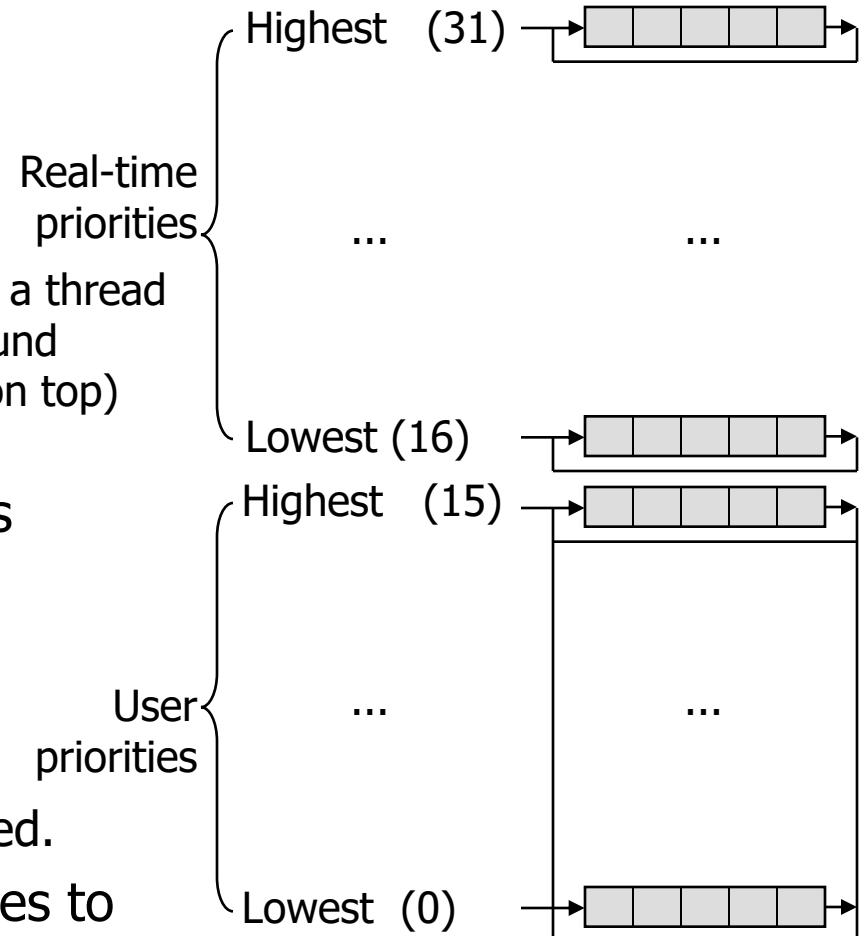
- In Unix-like systems (e.g. Linux), the scheduling algorithms are subject of frequent changes: often, each new kernel version has a new scheduler (However, typically just variants of RR with priorities).
 - ⇒ We do not cover Linux scheduling here as it is a moving target. (If you have a printed book that covers the Linux scheduler, you can be sure that it is outdated when you buy the book...)
- Scheduling algorithms of Microsoft Windows do not change that frequently.
 - ⇒ We will have a closer look on the Microsoft Windows scheduling algorithm (used since Windows XP and later) on the next slides.
 - Since Windows 7, user-level-like threads are supported natively (user-mode-scheduling, UMS) in the kernel and for them, own schedulers can be provided.

Microsoft Windows Scheduling

- Kernel-level threads, i.e. threads are scheduled with system-contention scope.
- Preemptive Multilevel Feedback Queue scheduling based on 32 different priorities:
 - 16 **real time (or system) priorities** (16-31):
 - For operating system threads or for threads to which the administrator assigned real-time priority.
 - Priorities in this class do not change.
 - 16 **user priorities** (0-15, 0 only used for low priority system threads):
 - Priorities may be set by a user (UI) or a process/thread (API).
 - Priorities in this class change over the time.
 - Soft-real time: If a higher priority real-time thread becomes ready (e.g. I/O completed) while another thread is running, that other is immediately preempted. (However, no guaranteed deadlines.)

Microsoft Windows Scheduler (“Dispatcher”)

- Round-Robin with priorities
 - Time quantum:
 - Windows Server: 120ms
 - Desktop editions: 20ms
 - Typically, the time quantum of a thread that is running in the fore-ground (i.e. window of application is on top) is multiplied by a factor of 3.
- Scheduler manages 32 FIFO queues containing ready threads of the respective priority.
 - Threads with lower priorities only get CPU allocated when all higher priority threads are blocked.
- Processor affinity: The scheduler tries to schedule a thread always on the same physical core.



Dynamic Priorities for Microsoft Windows User Processes

- Priorities of non real-time threads are dynamically adjusted:
 - Priority is increased (however, never >15 which would be real-time) when a blocked thread gets ready again.
 - ⇒ Utilisation of I/O devices increases (a thread that had to wait for I/O in the past, is likely to do I/O in the future again).
 - ⇒ Response time of interactive thread decreases (=gets faster).
 - Threads that have been blocked because of user interaction (waiting for keyboard, mouse) get a larger increase in priority than other I/O.
 - In addition, threads belonging to a window that is in the foreground get a priority boost (in addition to the boost of the time quantum).
 - After each time slice, priority of the expired thread decreases by one until initial priority is reached.
 - A thread that did not run for some amount of time (i.e. that might suffer from starvation), gets its priority set to 15 for 2 time slices.

5.9 Summary

- CPU scheduling: select a waiting process from the ready queue and allocate the CPU to it.
- First-come, first-served (FCFS): simple.
- Shortest-job-first (SJF): optimises average waiting time, but starvation possible (unfair).
 - However, length of job often not known in advance.
(Shortest Burst Next: Tries to predict length of next burst using exponential averaging.)
- Shortest-remaining-time-first (SRTF): preemptive variant of SJF.
- Round-Robin (RR): appropriate for interactive systems due to preemptive time slicing.
- Round-Robin with priorities: supports priorities, however starvation may occur (unfair).
 - Aging of process priorities may prevent starvation.
- Multilevel Queue algorithms use different algorithms for different classes of processes.
 - Feedback variant allows to move processes between queues.
- In practise, typically Round-Robin with priorities or Multilevel Feedback Queue Scheduling (often involving Round-Robin) is used (see e.g. Windows XP scheduling).
- Symmetric multiprocessing (SMP): each processor executes the same instructions of the kernel and access the same shared kernel data structures, e.g. scheduling queues.
- Advanced topic not treated in this course: Evaluation of scheduling algorithms
 - Analysis based on queueing-network theory, simulation, or implementation.

Course
TÖL401G: Stýrikerfi /
Operating Systems
6. Process Synchronisation

Chapter Objectives

- Illustrate a race condition and describe the critical-section problem.
- Present software, hardware and higher-level solutions to the critical-section problem:
 - Peterson's Algorithm,
 - Atomic instructions, including spinlocks,
 - Semaphores,
 - Monitors and condition variables,
 - Message passing.
 - Java API for semaphores and monitors with conditions.
- Present classical synchronisation problems.
- Have fun with the Deadlock Empire computer game.

Contents

1. Introduction: Shared Resources and Race Conditions
2. The Critical-Section Problem
3. Software Solutions
4. Hardware Solutions
5. Operating System-supported Solutions
6. Alternative Approaches
7. Classical Problems of Synchronisation
8. Summary

Note for users of the Silberschatz et al. book: For didactical reasons, I will present the material of this chapter slightly differently.

6.1 Introduction: Shared Resources(samnýttar auðlind)

- Processes/threads running in parallel, may share resources:
 - Devices: Printer, hard disk, etc.
 - Data: Files, memory shared between processes, global variables shared by threads of the same process, etc.
- If possible, shared resources shall be avoided (because they may lead to inconsistencies due to race conditions →next slide), but this is not always possible, e.g.
 - resource exists only once (because that hardware resource is expensive),
 - processes/threads need exactly to share data to work together and to communicate.

Race Conditions

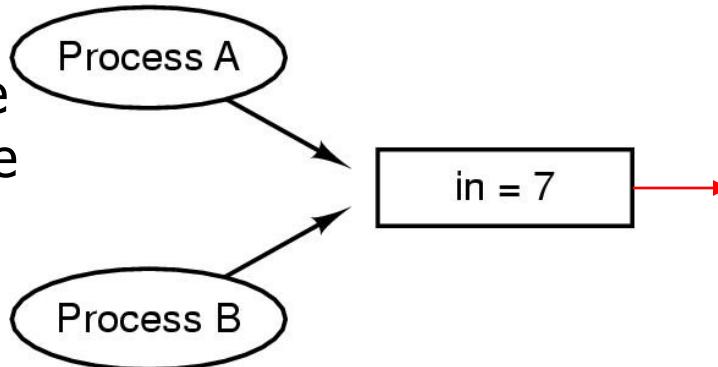
- Access to shared resource must be coordinated (**synchronised**).
 - Otherwise, produced results depend on the schedule (which may differ from run to run) of the concurrent processes instead of being deterministic (i.e. independent from the possible schedules): “race condition” (or “race”).
 - Race condition = **Result depends on timing**
i.e. order in which scheduler decides to execute concurrent processes.
 - Because a user does not see the underlying schedules, the result seems to be non-deterministic (óákveðinn): sometimes a software works, sometimes not.
 - Consequences of a race condition:
 - Data may get lost and even corrupted. If operating system data structures are subject of race conditions, the whole system might crash.
 - To avoid race conditions, the possible schedules of concurrent processes need to be restricted to those that do not lead to problems.
 - Essentially, this whole chapter 6 is about how to achieve this!

Example for Race Conditions (1)

- Two processes A and B submit concurrently their print job to a printer spooler:

- Print jobs are submitted to a buffer (e.g. RAM or a directory in the file system).
- The global (=shared) variable `in` is used to specify the next free slot in the shared buffer of the printer spooler where the processes shall submit their jobs.

- Each submitting process is responsible for updating the value of variable `in` after submitting a job:
 $in= in+1$.



Spooler directory	
4	:
5	abc
6	prog.c
7	prog.n
8	⋮

- If the processes try to submit print jobs concurrently, a race condition may occur (\rightarrow next slide).

Example for Race Conditions (2)

(Single processor/single core – concurrency by scheduler context switches)

■ Process A

- Reads value of variable `in` (= 7) into CPU register.
- Writes job into the slot that is specified by CPU register (= 7).
- Gets interrupted by scheduler timer interrupt before updating variable `in`.
 - Dispatcher saves CPU register in PCB of process A. Process B selected by scheduler as next process.

■ Process B

- Reads value of variable `in` (which is still 7) into CPU register.
- (Over-)Writes job into the slot that is specified by CPU register (= 7).
- Increments CPU register, updates variable `in` using value in CPU register (new value: 8).
- Terminates.

■ Process A

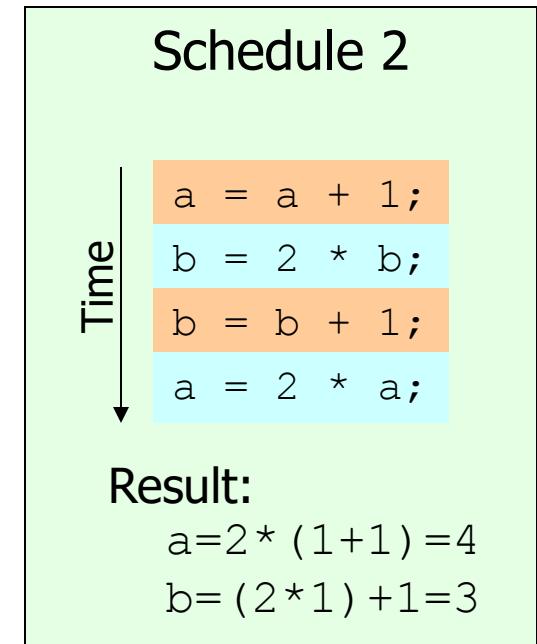
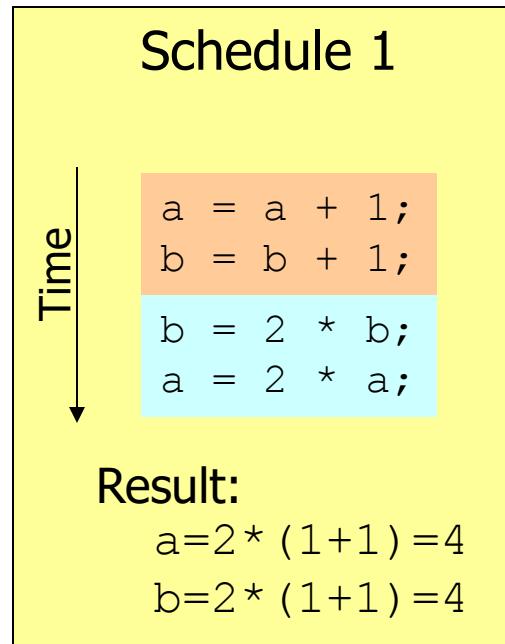
- CPU registers are restored by dispatcher, i.e. process A resumes execution where it was interrupted.
- Increments CPU register, updates variable `in` using value in CPU register (new value: 8).
- Terminates.

■ Job from process A gets lost, because access to variable `in` was not synchronised.

Example for Non-deterministic Results Due to Race Conditions

Pseudo code construct to specify execution of P1 and P2 as concurrent processes/threads.

```
int a = 1, b = 1;  
parallel {  
    P1 () {  
        a = a + 1;  
        b = b + 1;  
    }  
  
    P2 () {  
        b = 2*b;  
        a = 2*a;  
    }  
}
```



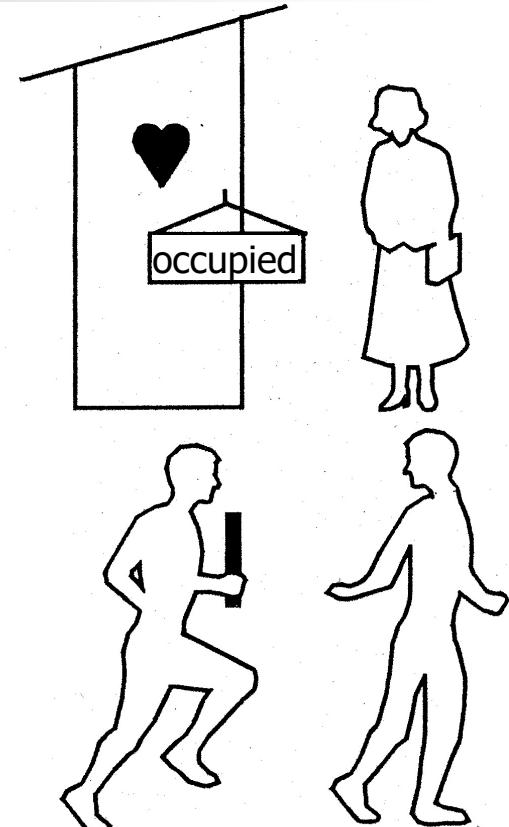
a and **b** are shared variables.

Note: These are just two possible schedules that the CPU scheduler of the operating system may create due to context switches while executing P1 and P2. In fact, even more different schedules exist. On multi-core/-processor systems, P1 and P2 may even run in parallel.

- Non-deterministic results due to race conditions!

Types of Synchronisation Between Interacting Processes

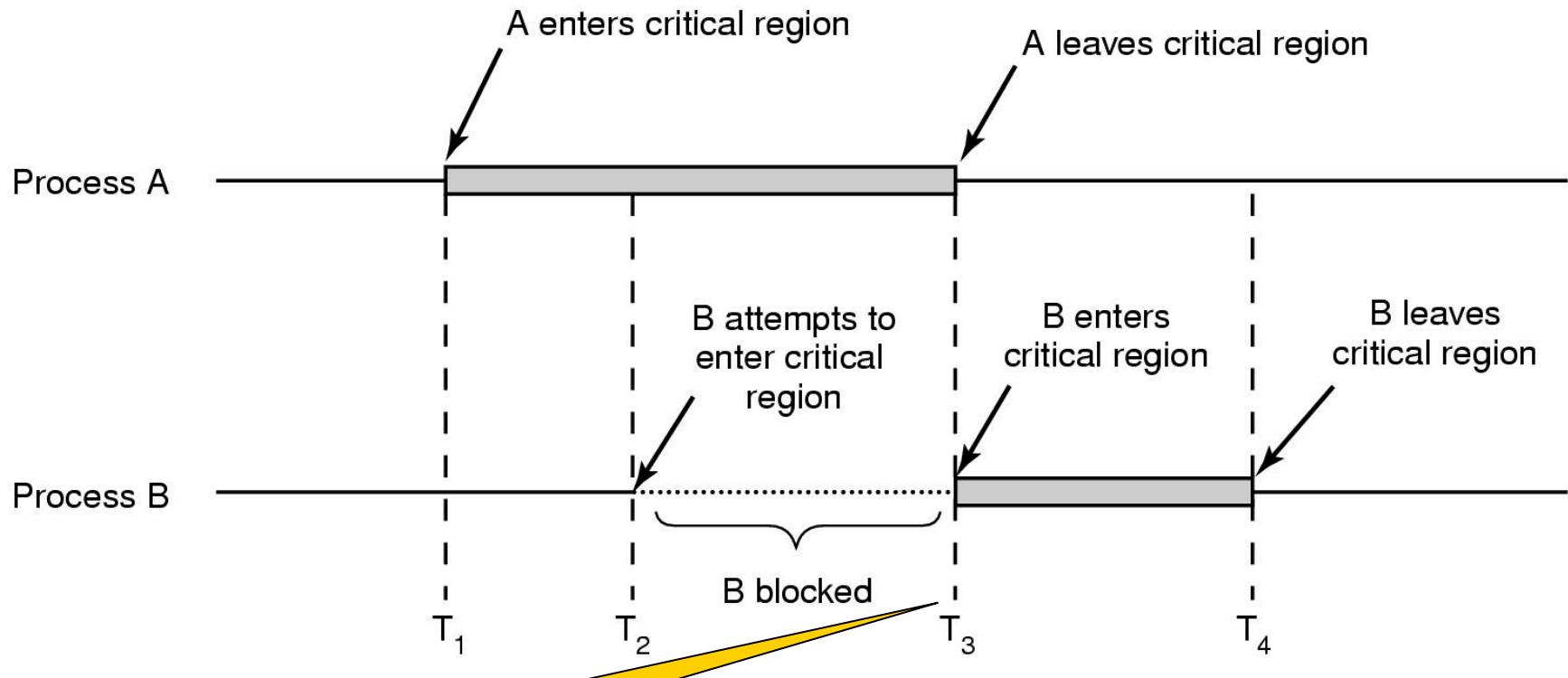
- Mutual Exclusion:
 - Prevent simultaneous access to shared resources.
 - (Order of access does not matter as long as it is mutual exclusive.)
 - Goal: Maintain consistency of data.
- Condition Synchronisation:
 - Wait for conditions/events.
 - Goal: Ordering of actions.
- Note: often, but not necessarily always, condition synchronisation includes mutual exclusion.
 - See, e.g. slide “Semaphores (10): Example Producer-Consumer Problem”:
 - There, conditions synchronisation does not include mutual inclusion – mutual exclusion rather needs to be added separately.



6.2 The Critical Section Problem

- A **critical section** (also called: critical region) of a process contains code that accesses shared resources.
 - Different sets of resources may be shared between processes. For each set of shared resources, a different critical section may exist.
 - E.g. critical section accessing resource A, another critical section accessing resource B: accessing A and B simultaneously is no problem.
- **Mutual Exclusion (Gagnkvæm útilokun):**
 - If a process is executing in its critical section, no other process can execute its critical section (concerning the same shared resources).
 - I.e. no process may enter its critical section, if another process is already in its critical section.
- By achieving mutual exclusion, the critical section problem (no accesses to the same shared resource at the same time) is solved.
 - A solution for orderly entering and leaving the critical section is required!

Entering & Leaving the Critical Section



To achieve mutual exclusion, B needs to be blocked until A leaves critical section

Requirements on Solutions for the Critical Section Problem

- No two processes may be in their critical section at the same time (**mutual exclusion**).
- No process that is outside its critical section may block other processes (**progress**).
- No process waits infinitely to enter its critical section (**bounded waiting**).
- Furthermore: No assumptions concerning the relative speed of processes and the number of CPUs/cores are made.

Atomicity/Atomic Instructions

- An operation or instruction (or a region of several instructions) is atomic, if it appears to the rest of the system (=other processes) to occur instantaneously, i.e. it cannot be divided or interrupted. E.g.:
 - The high-level programming language operation `a=a+1` is not atomic:
 - In fact, the compiler generates multiple machine code instructions:
 - Reading variable `a` from RAM into a CPU register,
 - Adding 1 to that CPU register,
 - Finally writing that CPU register back to variable `a` in RAM.
 - Each of these machine code instructions can be interrupted (e.g. by a timer interrupt that activates the scheduler which might switch to a different process) or on a multi-core/multi-processor system another core/processor may execute other instructions of other processes in parallel.
 - The parallel regions of P1 and P2 on slide 6-8 are not atomic:
 - A scheduler may interrupt P1 or P2 in-between and switch to the other process or on a multi-core/multi-processor system another core/processor may execute the other processes in parallel.
 - Atomicity is one possibility to achieve mutual exclusion.

Want to play a computer game? The Deadlock Empire!

- Browser game: <http://deadlockempire.github.io/>
 - While a real programmer has of course to avoid race conditions, you win the Dead-lock Empire games if you are able to provoke a race condition (or deadlock →ch. 7)
 - =Learning what can go wrong, teaches how avoid that things go wrong.
- Allows you to run step-by-step through various code examples using a simulator.
- You play being the scheduler, i.e. decide on which thread shall execute next its next instruction.
 - By this you can create “good” and “bad” schedules, e.g. circumventing race conditions or provoking deadlocks.
- Slay dragons, master concurrency!

The Deadlock Empire

User Interface of Simulator

Tutorial 2: Non-Atomic Instructions

Many statements are not atomic and are actually composed of several "minor" statements. Whenever such a statement is the active instruction, you can "expand" it to be able to step through with more precision. Follow the path:

First, click "Expand" to see that a simple single line statement such as an assignment as `a=a+1` is not atomic, but can expanded into sub-statements which more or less resemble the underlying machine code instructions generated by a compiler.

Undo Reset level Return to main menu

You will still need the old value of 'a' (zero) in the second thread! Click "Expand" it. Again, click "Step" to read the expression into a thread-local variable. This will ensure that both threads have access to the same critical section with both threads.

If you want to undo all your actions, click "Undo".

Thread 0

Step Expand

```
a = a + 1;  
if (a == 1) {  
    critical_section();  
}
```

Run each thread step-by-step.

```
int a = 0;
```

Current value of shared variable (subject to race condition).

Thread 1

Step Expand

```
// Expand the following instruction:  
a = a + 1;  
temp = a + 1;  
a = temp;  
if (a == 1) {  
    critical_section();  
}
```

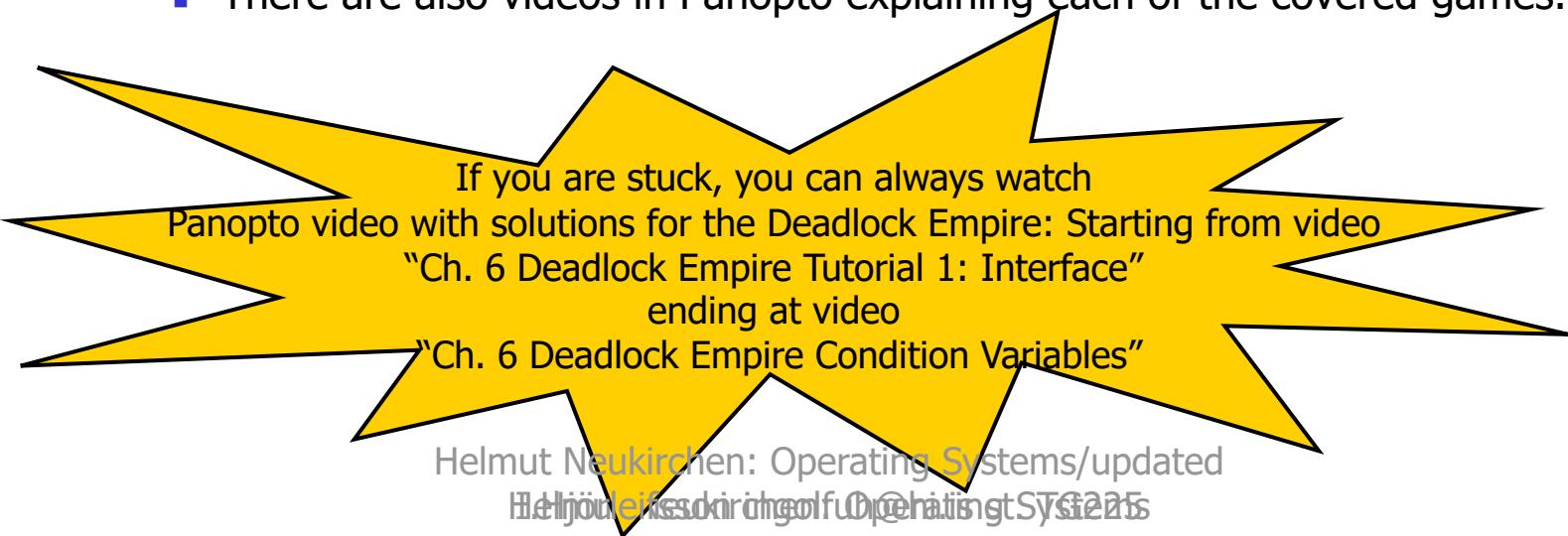
Yellow background line will always be executed next (after having clicked "Step").

Sometimes, there are white background lines, but these are in fact supposed to be transparent background lines.

The Deadlock Empire

Take a break: Play now!

- Now:
 - Open in your web browser: <http://deadlockempire.github.io/>
 - Play from the above web page the “Tutorial” consisting of two games:
 - Tutorial 1: Interface
 - Tutorial 2: Non-Atomic Instructions
 - Play this game not only once, but try all the possible schedules.
 - The further games refer to topics that you’ll learn later, so do not play them now, but later.
 - There are also videos in Panopto explaining each of the covered games.



If you are stuck, you can always watch
Panopto video with solutions for the Deadlock Empire: Starting from video
“Ch. 6 Deadlock Empire Tutorial 1: Interface”
ending at video
“Ch. 6 Deadlock Empire Condition Variables”

Types of Solutions for the Critical Section Problem

- **Software solutions:**
 - Solution on application/user level: No additional support by hardware or the operating system.
 - Busy waiting.
- **Hardware-supported solutions:**
 - Solution based on CPU support for special machine code instructions.
 - Atomic execution of instructions, Disable interrupts.
- **Operating system-supported solutions:**
 - Operating system provides special system calls.
 - Semaphore, Monitor, Message Passing, etc.
 - (OS may block process by putting into waiting state on scheduler level.)
- **(We will discuss these three types in the following sections...)**

6.3 Software Solutions

- Software solutions use **shared global variables** for synchronisation of processes.
- Software solutions use **Busy Waiting**:
 - Wait for a certain condition/event (e.g. global variable changes value) by continuously probing. E.g.

```
while (condition != true) { /* do nothing */ };
```

- Busy Waiting consumes CPU time for doing nothing: **inefficient!**
 - Use software solutions only if no hardware or OS-supported solution are available.
- In the following, we consider only a solution for two processes/threads.

Software Solutions: Structure Used in Examples

- In the following examples we use C/Java-like pseudocode to investigate potential software solutions.
All programs have the same structure:

```
while (true) {  
    Prologue critical section (Entry section)  
    critical section  
    Epilogue critical section (Exit section)  
    Remainder section  
};
```

- Prologue and epilogue of critical section are responsible for orderly entering and leaving critical section.
 - On the following slides, our main focus will be these prologues and epilogues of the critical sections. (It does not matter how the critical section looks like.)
 - For didactic reasons, we will first start with some (unsuccessful) tries that nearly solve the problem until we come up with the final solution.

Software Solutions: First Try (1)

- Global variable for synchronisation:

```
int turn = 1;
```

Specifies which process is allowed to enter the critical section.

- Two processes running in parallel:

Process P0:

```
while (true) {
    while (turn!=0) { /* wait */ };
    <critical section>
    turn = 1;
    /* Remainder */
};
```

Process P1:

```
while (true) {
    while (turn!=1) { /* wait */ };
    <critical section>
    turn = 0;
    /* Remainder */
};
```

Software Solutions: First Try (2)

- Discussion:
 - Advantage:
 - Mutual exclusion is achieved.
 - Disadvantage:
 - Processes may only enter critical section in an alternating manner.
 - Slower process dictates speed.
 - If one process terminates, the other process is blocked infinitely.
 - This violates the requirements on the solution
(→reminder on next slide).

Reminder: Requirements on Solutions for the Critical Section Problem

- No two processes may be in their critical section at the same time (**mutual exclusion**).
 - No process waits infinitely to enter its critical section (**bounded waiting**).
 - No process that is outside its critical section may block other processes (**progress**).
-
- Furthermore: No assumptions concerning the relative speed of processes and the number of CPUs are made.

Software Solutions: Second Try (1)

- Problem of first try:
 - Only that process whose ID is stored in the synchronisation variable, may enter the critical section next.
- Better:
 - Each process has its own “key” for the critical section to assure that
 - a process may still enter the critical section, even if the other process fails.
 - a process may enter the critical section independent from the other process that currently does not want to access the shared resource.

Software Solutions: Second Try (2)

- Global variables for synchronisation:

```
boolean flag[2] = {false, false};
```

flag[n]==true, if process n is
in critical section.

Process P0:

```
while (true) {
    while (flag[1]) { /* wait */ };
    flag[0] = true;
    <critical section>
    flag[0] = false;
    /* Remainder */
};
```

Process P1:

```
while (true) {
    while (flag[0]) { /* wait */ };
    flag[1] = true;
    <critical section>
    flag[1] = false;
    /* Remainder */
};
```

Software Solutions: Second Try (3)

- Discussion
 - Advantage:
 - If one of the processes fails (or terminates) outside its critical section, the other process may still enter its critical region.
 - Processes may enter their critical region in a non-alternating style independent from the speed of the other process.
 - Disadvantage:
 - Not even the requirement of mutual exclusion is achieved. Counter example:

Schedule ↓

Assumption: `flag[2] = {false, false}`

P0: executes `while (flag[1]);` and notices that `flag[1]` is `false`.

P1: executes `while (flag[0]);` and notices that `flag[0]` is `false`.

P0: executes `flag[0] = true;`

P1: executes `flag[1] = true;`

****** Both processes enter their critical section. ******

Reminder: Requirements on Solutions for the Critical Section Problem

- No two processes may be in their critical section at the same time (**mutual exclusion**).
- No process waits infinitely to enter its critical section (**bounded waiting**).
- No process that is outside its critical section may block other processes (**progress**).
- Furthermore: No assumptions concerning the relative speed of processes and the number of CPUs are made.

Software Solutions: Third Try (1)

- Problem of second try:
 - Process may change its flag **after** another process has probed that flag.
- Maybe, this problem can be solved by swapping the order of the two problematic instructions?
 - I.e. first change flag, then probe flag.

Software Solutions: Third Try (2)

- Global variables for synchronisation:

```
boolean      flag[2] = {false, false};
```

Process P0:

```
while (true) {
    flag[0] = true;
    while (flag[1]) { /* wait */ };
    <critical section>
    flag[0] = false;
    /* Remainder */
};
```

Process P1:

```
while (true) {
    flag[1] = true;
    while (flag[0]) { /* wait */ };
    <critical section>
    flag[1] = false;
    /* Remainder */
};
```

Software Solutions: Third Try (3)

- Is this third try the solution of the critical section problem?
 - Viewpoint of P0 – Possible cases:
 - P0 sets `flag[0]` to `true`, P1 is not in the critical section:
 - P1 may only enter critical section after P0 entered it (because P1 is blocked by `while (flag[0]);`) and left it again.
 - P0 sets `flag[0]` to `true`, P1 is in the critical section:
 - P0 is blocked by `while (flag[1]);` until P1 has left the critical section.
 - The same considerations are valid for P1, hence this third try achieves mutual exclusion.
 - **But:**

Software Solutions: Third Try (4)

- ... a deadlock (=each process is waiting for the other process → ch. 7) may occur:

Assumption: `flag[2] = {false, false}`

P0: executes `flag[0] = true;`

P1: executes `flag[1] = true;`

P0: executes `while (flag[1]);` and **waits** because `flag[1] == true.`

P1: executes `while (flag[0]);` and **waits** because `flag[0] == true.`

****** A deadlock occurs: Each process waits infinitely that the other process leaves the critical section. ******

- Of course, this violates one of our requirements...

Reminder: Requirements on Solutions for the Critical Section Problem

- No two processes may be in their critical section at the same time (**mutual exclusion**).
- No process waits infinitely to enter its critical section (**bounded waiting**).
- No process that is outside its critical section may block other processes (**progress**).
- Furthermore: No assumptions concerning the relative speed of processes and the number of CPUs are made.

Software Solutions: Excursion into History

- For long time (in the early days of computer science), researchers thought that there is no software solution for the critical section problem.
- In 1965, the Dutch mathematician Theodorus J. Dekker published a correct software solution for the critical section problem!
 - However, Dekker's algorithm is somewhat hard to understand and the correctness is hard to prove.
- In 1981, the American mathematician Gary L. Peterson published a more simple and more elegant solution of the critical section problem.
 - This algorithms uses both a `flag` array (to indicate the state of each process concerning the mutual exclusion) and a variable `turn` (to resolve problems of identical relative speed). (→next slide)

Software Solutions: Peterson's Algorithm (1)

- Global variables for synchronisation:

```
boolean      flag[2] = {false, false};  
int         turn;
```

In principle, the third try was good. Now, the `turn` variable is used to introduce some asymmetry so that not both processes wait for each other if they progress in parallel.

Process P0:

```
while (true) {  
    flag[0] = true;  
    turn = 1;  
    while (flag[1] && turn == 1)  
        { /* wait */ };  
<critical section>  
    flag[0] = false;  
    /* Remainder */  
};
```

Process P1:

```
while (true) {  
    flag[1] = true;  
    turn = 0;  
    while (flag[0] && turn == 0)  
        { /* wait */ };  
<critical section>  
    flag[1] = false;  
    /* Remainder */  
};
```

=Proofing that the four requirements on solutions for the critical section problem are not violated.

Software Solutions: Peterson's Algorithm (2) – Proof

- 1. No process waits infinitely to enter its critical section (bounded waiting):
 - Assumption (indirect proof, i.e. proof by contradiction): P0 is blocked infinitely in its inner **while** loop (=waits infinitely).
 - Distinguish three cases concerning what P1 might be doing:
 - Case 1:
 - P1 does not want to enter the critical section (=is outside).
 - Case 2:
 - P1 is waiting as well in its **while** loop to enter the critical section.
 - Case 3:
 - P1 enters the critical section over and over again, therefore infinitely blocking P0.
 - In the following: Proof by contradiction.

Software Solutions: Peterson's Algorithm (3) – Proof

- 1. No process waits infinitely to enter its critical section (bounded waiting):
 - Assumption (reminder):
 - P0 is blocked infinitely in its inner **while** loop.
 - This means:
 - flag[1] == **true** and turn == 1
 - Case 1:
 - P1 does not want to enter the critical section (=is outside).
 - This means, flag[1] == **false**, hence the condition for the blocking of P0 in its while loop is not fulfilled and thus, blocking of P0 is not possible.
⇒Contradiction to assumption.

Software Solutions: Peterson's Algorithm (4) – Proof

- 1. No process waits infinitely to enter its critical section (bounded waiting):
 - Assumption (reminder):
 - P0 is blocked infinitely in its inner **while** loop.
 - This means:
 - `flag[1] == true` and `turn == 1`
 - Case 2:
 - P1 is waiting as well in its **while** loop to enter the critical section.
 - Not possible, as P1 would be able to enter the critical section because `turn == 1`.
(`turn` is either 0 or 1, hence it is impossible that both processes are blocked at the same time.)

Software Solutions: Peterson's Algorithm (5) – Proof

- 1. No process waits infinitely to enter its critical section (bounded waiting):
 - Assumption (reminder):
 - P0 is blocked infinitely in its inner **while** loop.
 - This means:
 - flag[1] == **true** and turn == 1
 - Case 3:
 - P1 enters the critical section over and over again, therefore infinitely blocking P0.
 - Not possible as P1 sets **turn** to 0 each time it tries to enter the critical section, thus giving P0 the possibility to enter the critical section.
⇒Contradiction to assumption.

Software Solutions: Peterson's Algorithm (6) – Proof

- 2. No two processes may be in their critical section at the same time (mutual exclusion):

- Assumption:

Both, P0 and P1 want to enter the critical section.

- P0 sets `flag[0]` to `true`, P1 sets `flag[1]` to `true`.
 - `turn` is set by P0 as well as by P1. The one that sets it later, “wins”. `turn` can only have one value at a time, depending on the scheduling, two cases are possible:
 - `turn==0`:
 - P1 cannot enter critical section.
 - `turn==1`:
 - P0 cannot enter critical section.

⇒ Only one process in critical section.

Software Solutions: Peterson's Algorithm (7) – Proof

- 3. No process that is outside its critical section may block other processes (progress):
 - As soon as P1 leaves its critical section, it sets `flag[1]` to `false`, hence P0 may enter the critical section.
 - The same considerations are valid in the opposite direction:
As soon as P0 leaves its critical section, it sets `flag[0]` to `false`, hence P1 may enter the critical section.

Reminder: Requirements on Solutions for the Critical Section Problem

- No two processes may be in their critical section at the same time (mutual exclusion).
 - See proof 2. ✓
 - No process waits infinitely to enter its critical section (bounded waiting).
 - See proof 1. ✓

Well... if a process crashes inside its critical section, then the other process waits infinitely. (But this is a general problem that cannot be prevented – except using timeouts.)
 - No process that is outside its critical section may block other processes (progress).
 - See proof 3. ✓
 - Furthermore: No assumptions concerning the relative speed of processes and the number of CPUs are made.
 - No such assumptions were necessary. ✓

Well... there may always be unlucky schedules: priority inversion problem → 6-52
- Helmut Neukirchen: Operating Systems/up
I.Hjörleifsson ingolfuh@hi.is st. TG22

Remark: Running Software Solutions on Modern Systems: **volatile** keyword

- Modern **compilers** apply various **optimisations** to speed up program execution:
 - E.g. in the sequence "turn = 1; **while** (flag[1] && turn == 1)" a compiler could conclude that the check `turn == 1` is unnecessary (because `turn` has just been set to 1 in the statement before) and thus generate no code for it.
⇒ A software solution (that is based on that variable `turn`) is shared and may be set in parallel by another process) would not work anymore!
 - To avoid this, most programming languages allow to **declare a variable as "volatile"** to prevent such optimisations.
 - **volatile** will tell the compiler to always read the value from main memory.
 - Useful for shared variables modified by other processes/threads.
 - Useful if memory location may change for other reasons, e.g. an I/O controller directly writing data to that location.
 - However, even using the "**volatile**" keyword does not guarantee that software solutions **work in modern systems** > next slide.

Remark: Running Software Solutions on Modern Systems: Reordering by CPUs

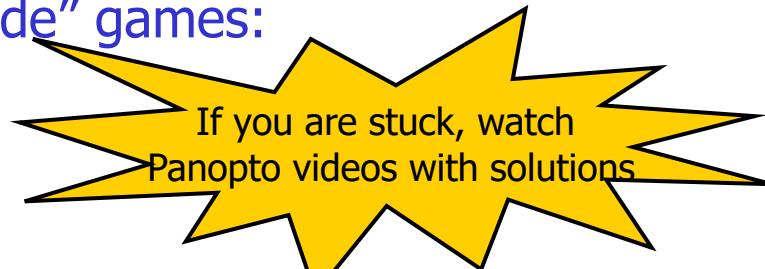
- Modern CPUs reorder internally instructions and their memory accesses to improve execution speed, e.g. to avoid memory stalls:
 - When running software solutions, the actual accesses to the synchronisation variables may occur in a different order than expected (only a problem with shared memory).
⇒ The software solution will not work anymore on any modern CPU!
- The memory model of a CPU describes what ordering is guaranteed and what ordering you cannot rely on, e.g.:
 - Strongly ordered: a memory modification on one processor/core is immediately visible to all other processors/cores.
 - Weakly ordered: modifications to memory on one processor /core may not be immediately visible to other processors/cores.

Remark: Running Software Solutions on Modern Systems: Memory barriers

- To enforce an ordering, CPUs have special machine code instructions that do not get reordered: “**memory barriers**”/“**memory fences**”:
 - All memory accesses of machine code instructions will be completed before the barrier machine code instruction continues execution and after continuing, all cores will see the result of these accesses performed before the barrier.
 - Also special atomic instruction (→Hardware solutions, next slides) have that property.
 - Compilers do not automatically generate these instructions, because these slow down execution and a compiler cannot detect those rare case when they are actually needed.
- Note: since version (1.)5 of Java, accesses to **volatile** variables use **memory barrier instructions** to prevent memory access reordering!
 - I.e. the memory model of the Java Virtual Machine changed!
- Other programming languages may not use barriers for **volatile**...

Want to play more computer games? The Deadlock Empire strikes back!

- Browser game: <http://deadlockempire.github.io/>
 - Reminder: in these games always the yellow background line will be executed next.
 - Sometimes, there are white background lines, but these are in fact supposed to be transparent background lines.
- Now, play the three “Unsynchronized Code” games:
 - Boolean Flags Are Enough For Everyone
 - Simple Counter
 - Confused Counter
 - “the failure instruction” refers to `Debug.Assert(false);`
 - C# is used: the `Interlocked.Increment` mentioned in one of the pop-ups refers to an atomic C# operation (comparable atomic Java operations exist as well, e.g. in class `java.util.concurrent.atomic.AtomicInteger`).
 - The further games refer to topics that you’ll learn later (or not at all).



6.4 Hardware Solutions

- Solving the critical section problem may use special hardware features of a CPU:
 - Either: Disabling interrupts
 - Or: [Atomic instructions](#).
- Atomic machine code instructions may be used even in user mode (however, typically a high-level programming language does not support them);
- However, disabling interrupts is typically a privileged machine code instruction that requires kernel mode and thus this is typically only used by the kernel itself.
 - Furthermore, not reasonable on today's multicore-processors.

Disabling Interrupts

- Simple approach: before entering the critical section, all hardware interrupts are disabled (and enabled again after leaving the critical section):

```
while (true) {  
    Disable interrupts  
    <critical section>  
    Enable interrupts  
    Remainder  
};
```

- Prevents that a timer interrupt is processed by the OS scheduler.
- ⇒ No context switch (preemption) occurs and thus no other processes get executed that might enter the critical section.
(well – see problems discussed on next slide...)

Disabling Interrupts: Problems

- Not reasonable in multi-core/multi-processor systems:
 - On multi-core/multi-processor systems, multiple processes can execute at the same time (one process on each core/processor) even without a scheduler and context switch, thus each of them might enter a critical section even if all interrupts are disabled.
- Process must not reside too long in critical section, otherwise
 - it may be too late to serve interrupt,
 - (Imagine the interrupt of a mass storage device that indicates that it wants to be served: data may get lost if it is not served fast enough.)
 - interrupts may get lost.
 - (Even worse than just handling interrupt too late.)
- If a process crashes (or is in a deadlock) while being in the critical section, the whole system gets blocked (because, interrupts never get enabled again!)
 - (The timer interrupt of the scheduler is disabled and thus the scheduler cannot interrupt crashed processes!)
- Realises only mutual exclusion, however not condition synchronisation.
 - A process cannot wait that a certain condition gets fulfilled by another process.

How to achieve atomicity?

- On **single core/single processor systems**, typically **interrupt disabling** is used to achieve mutual exclusion when accessing kernel data structures.
 - If care is taken that interrupts are disabled for a short section of code, this is an appropriate solution to achieve mutual exclusion within kernel code.
- However, for SMP (**symmetric multi core/multi processor**) systems, a different solution is required:
 - Typically, this is based **on atomic machine code instructions**:
 - The difficult part of the software solutions of the critical section problem was that between testing and setting a lock variable (variable “flag” in the previous examples) another process could set the lock variable as well.
 - Most CPUs provide special machine code instructions that allow to **test and set the value of a memory location without being interrupted in-between**.
 - No internal memory access reordering and even with multiple cores/CPUs, only one will be able to access memory at a time (“atomic instruction”).
 - Note: Instead of an atomic test-and-set, a CPU might offer an **atomic swap** or **atomic exchange instruction**: Sets value of memory location and returns previous value (that can later-on be tested).

Atomic Instructions: Test-And-Set Instruction

- If a Test-And-Set machine code instruction of a CPU would be implemented using Java syntax (in fact, a CISC CPU uses microcode to implement its machine code instructions), its implementation would look as follows:

```
boolean lock=false;  
  
boolean testAndSet (boolean newValue) {  
    boolean oldValue=lock;  
    lock=newValue;  
    return oldValue;  
}
```

Global variable used as "lock".

Atomic read/write access to memory location `lock` that cannot be interrupted and gets not reordered.

Return value can be used to check whether another process is already in critical section.

- Variant without `newValue` parameter:

```
boolean lock=false;  
  
boolean testAndSet () {  
    boolean oldValue=lock;  
    lock=true;  
    return oldValue;  
}
```

Global variable used as "lock".

Atomic read/write access to memory location `lock` that cannot be interrupted and gets not reordered.

Always set to true: If `lock` was already true, it remains true, if it was false, it becomes true.
Previous value will be returned and can be checked.

Atomic Instructions: Test-And-Set Instruction: Example

- Mutual exclusion of n processes using test-and-set instruction:

```
boolean lock=false;
void P() {
    while(true) {
        while( testAndSet() ) { /* busy waiting */ }

        /* critical section */

        lock = false;
        /* further program code */
    }
}

void main() {
    parallel { P() } { P() } ... { P() };
}
```

Wait until `lock` becomes false and set it to true. Due to atomicity, only one process will read a `false`.

Setting `lock` (without testing it) requires no atomic instruction.

- Easy to use.
- Applicable for mutual exclusion of n processes (not just 2).

Atomic Instructions: Discussion

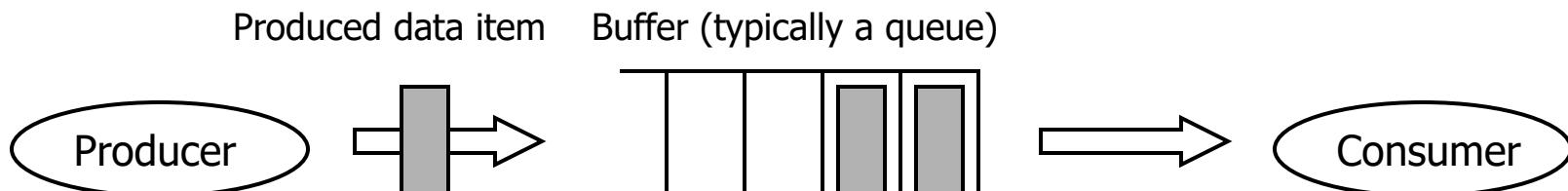
- Advantages:
 - Easy to use.
 - Applicable for an arbitrary number of processes.
 - Test-and-Set (or Exchange/Swap) allows (in contrast to disabling interrupts) control of different critical sections for different shared resources at the same time: just use a different lock variable for each shared resource.
- Disadvantages:
 - Busy waiting is used (wastes CPU time).
 - Starvation is possible:
 - A low priority process will never be able to enter critical section as long as there are higher priority processes competing for critical section using busy waiting.
 - Deadlocks are possible (“priority inversion problem” → next slide)

6.5 Operating System-supported Solutions

- Main problem of software and hardware solutions for the critical section problem is busy waiting.
- Wanted: synchronisation mechanisms that avoid busy waiting.
 - Can only be achieved on operating system-level, because only OS can block processes in a non-busy waiting style by putting processes into the scheduler's blocked queue/waiting state.
 - ⇒ Operating system-supported process synchronisation mechanisms.
 - Influence directly the scheduling by blocking one or more processes: scheduler assigns CPU only to processes that are in ready state.
 - ⇒ Set of possible schedules gets restricted.

Example: Producer-Consumer Problem

- In the following, we will use the producer-consumer problem as case study for applying operating system-supported process synchronisation mechanisms.
- Reminder producer-consumer problem: Concurrent producer and consumer processes share a common buffer.
 - Producer:**
 - Puts (inserts) data items into buffer until it is full.
 - Waits if buffer is full.
 - Resumes when buffer is not full any more.
 - Consumer:**
 - Consumes (removes) data items from buffer.
 - Waits if buffer is empty (no items available).
 - Resumes when new items are available.



Example: Producer-Consumer Problem

First try: Sleep and Wakeup (1)

- Assume that we have system calls **sleep** and **wakeup**:
 - **sleep()** system call:
 - The calling process volunteers to be put to sleep by the operating system until it is woken up by another process.
 - **wakeup(processID)** system call:
 - The calling process wakes up the sleeping process with PId **processID**.
 - If process with PId **processID** is not sleeping, wakeup signal is discarded.
- How would a solution of the producer-consumer problem look like when using **sleep** and **wakeup**?

Example: Producer-Consumer Problem

First try: Sleep and Wakeup (2)

■ Producer

Producer and consumer share buffer and this variable.

```
#define N 100          /* Size of buffer */
int count = 0;        /* Number of items currently in buffer */

void producer() {
    int item;

    while (TRUE) {           /* Endless loop */
        item = produce_item(); /* produce new item */
        if (count == N) {sleep();} /* sleep if buffer exceeded */
        insert_item(item);     /* put item into buffer */
        count = count + 1;     /* incr. number of items in buffer */
        if (count == 1) { wakeup(consumer); } /* if buffer was empty
before, consumer was sleeping to wait for new items */
    }
}
```

Example: Producer-Consumer Problem

First try: Sleep and Wakeup (3)

■ Consumer

```
#define N 100           /* Size of buffer */
int count = 0;          /* Number of items currently in buffer */

void consumer() {
    int item;
    while (TRUE) {           /* Endless loop */
        if (count == 0){sleep()};/* sleep if buffer empty */
        item = remove_item();   /* fetch new item from buffer */
        count = count - 1;     /* decr. number of items in buffer */
        if (count == N - 1){wakeup(producer)}; /* if buffer was full,
now one slot is available again: wakeup producer */
        consume_item(item);    /* consume fetched item*/
    }
}
```

Example: Producer-Consumer Problem

First try: Sleep and Wakeup (4)

- Solution looks good, however deadlock is possible. Consider the following scenario where both processes have just been started and the buffer is empty (i.e. `count=0`):

Consumer: checks `count` in if statement with condition `count == 0`

Scheduler: interrupts consumer (just before `sleep()`), and selects producer

Producer: produces item and inserts it into empty buffer:

```
insert_item(item);  
count = count + 1; /* count is now 1 */  
if (count == 1) wakeup(consumer);
```

wakeup signal is sent to consumer. However, consumer process does not sleep yet; hence, wakeup signal is discarded!

Scheduler: interrupts producer and selects consumer

Consumer: resumes interrupted execution:

- Executes second part of if statement (`sleep()`): puts itself to sleep

Scheduler: consumer is blocked, hence scheduler selects producer

Producer: resumes execution:

- produces new items and thus fills buffer until it is exceeded.
- Hence, puts itself to sleep to wait until consumer consumes item.

***** Deadlock, both processes are sleeping! *****

Example: Producer-Consumer Problem

First try: Sleep and Wakeup (5)

- The problem of **sleep** and **wakeup** in the previous example is that the wakeup signal gets discarded if the target process is not sleeping yet.
 - Problem would be solved, if wakeup signal gets not discarded but would be buffered in some queue for later consumption.
- Furthermore, an atomic “check and sleep” would have prevented the problem as well.
 - Besides this, another problem that might occur is that **count=count+1** in producer and **count=count-1** in consumer might be updated in parallel → race condition!
⇒ An atomic update of the counter is needed & the above queue.
 - This is how semaphores work!
 - So forget all about **sleep** and **wakeup**, instead find more on **semaphores** on the next slides.

Semaphores (1)

- Suggested 1965 by the Dutch computer scientist Edsger Wybe Dijkstra (1930-2002).
 - (Historically, the term “semaphore” refers to indicators whether a railway section is occupied or free – comparable to a traffic light.)
- A **semaphore** can be considered as a special type of variable (or as a special class in case of object-orientation). It consists of:
 - Integer value (“value” of the semaphore, used as counter),
 - Operations that can be applied to it:
 - Initialise (either pass a parameter or use 1 as default value): **init**
 - P (“Proberen”: Dutch for “try”): down/acquire/**wait**
 - V (“Verhogen”: Dutch for “increase”): up/release/**signal**
 - **wait** and **signal** are implemented as atomic operations.
 - In pseudocode, typically an object-oriented notation is used:
 - `mysemaphore.init(1), mysemaphore.wait() etc.`

In literature, all these different names can be found for these operations. From now on, only the names **wait** and **signal** will be used.



Semaphores (2): Implementation of init()

- Internal data structure used for each instance of a semaphore:

```
struct semaphore {  
    int count;  
    queueType queue;  
}
```

Queue for keeping track
of sleeping processes.

"Value" of semaphore: current value essentially indicates how much "space" (in terms of number of processes that are allowed to enter) is left in the critical section.

(There may be applications where it is reasonable to use a value higher than 1, in particular if the semaphore is not used to achieve mutual exclusion, but rather for condition synchronisation.)

- Init operation (Pseudocode implementation):

```
init(int init_value) {  
    count = init_value;  
    queue = empty;  
}
```

Semaphores (3): Implementation of wait()

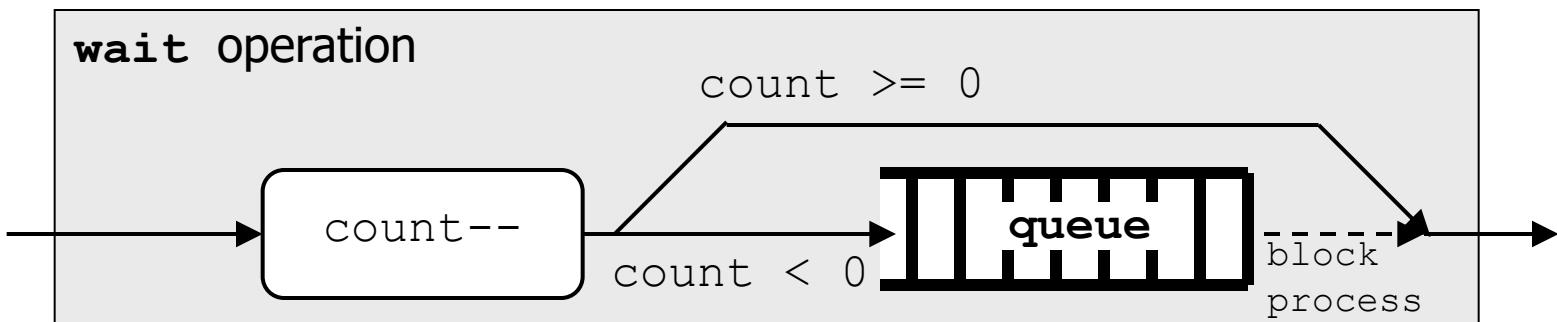
Atomic
(e.g. using hardware or software
solution of critical section problem)

- Atomic wait operation (pseudocode):

```
wait() {  
    count = count - 1;  
    if (count < 0) {  
        put this process in semaphore's queue;  
        block this process on OS level;  
        (=move process into waiting queue on OS level ⇒ process returns  
         only from this wait() once it gets woken up by signal())  
    }  
}
```

The same process that called the wait() operation will be put to sleep in this case!

- Schematic view:



Semaphores (4): Implementation of signal()

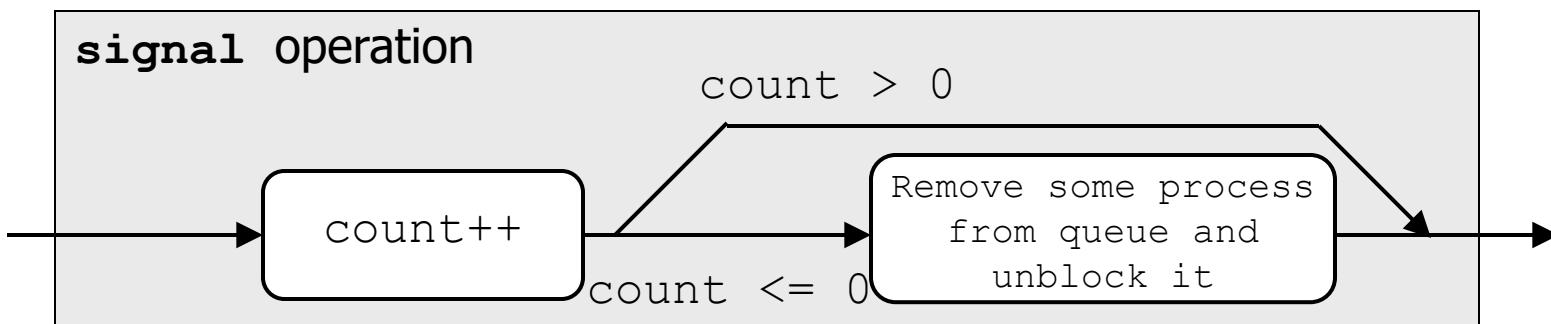
Atomic
(e.g. using hardware or software
solution of critical section problem)

- Atomic signal operation (pseudocode):

```
signal () {  
    count = count + 1;  
    if (count <= 0) {  
        Remove some process P from semaphore's queue;  
        Move process P to ready queue on OS level;  
        (=wakeup process P)  
    }  
}
```

Another process (one that called the wait() operation and was put to sleep) will get woken up in this case!

- Schematic view:



Semaphores (5): Typically Patterns of Using Semaphores

- Mutual exclusion:

```
sem_me.init(1)

Process A:
sem_me.wait()
<critical section>
sem_me.signal()

Process B:
sem_me.wait()
<critical section>
sem_me.signal()
```

- Condition synchronisation:

```
sem_condition.init(0)

Process A:
<condition occurred>
sem_condition.signal()

Process B:
sem_condition.wait()
<do something after
condition occurred>
```

- For more complex problems, multiple condition synchronisations may have to be used in parallel or mutual exclusion and condition synchronisation have to be combined. (In either case, this requires use of multiple independent semaphores.) Semaphores may have to be initialised with other values than 0 or 1 to use them as counters. → Examples on next slides...
- Take care that each `wait()` has somewhere a matching `signal()` for that semaphore, otherwise a waiting process gets never unblocked!
- A blocked process **cannot unblock itself** (it sleeps; only another process can signal).

Semaphores (6): Example Mutual Exclusion

- Mutual exclusion of n processes using a semaphore:

```
semaphore S_ME = S_ME.init(1); // S_ME=Semaphore for mutual excl.  
  
void P() {  
    while (true) {  
        S_ME.wait(); /* Enter critical section or wait */  
        /* critical section */  
        S_ME.signal(); /* Leave critical section */  
  
        /* Remainder */  
    }  
  
    void main() {  
        parallel { P() } { P() } ... { P() };  
    }  
}
```

Initial value of 1 achieves that at most 1 process may enter critical section. (If some problem would allow n processes in the critical section at the same time, initialise semaphore with n .)

Create n concurrent processes executing code of P()

Helmut Neukirchen: Operating Systems/updated
I.Hjörleifsson ingolfuh@hi.is st. TG225

Semaphores (7): Example Producer-Consumer Problem

- Solution for bounded producer-consumer problem (condition synchronisation) with a buffer size of 1:
 - Semaphores are not used to achieve mutual exclusion, but to put processes to sleep and wake them up according to conditions.

```
semaphore S_Data = S_Data.init(0); // >0: Producer produced item
semaphore S_NoData = S_NoData.init(1); // >0: Consumer requests item

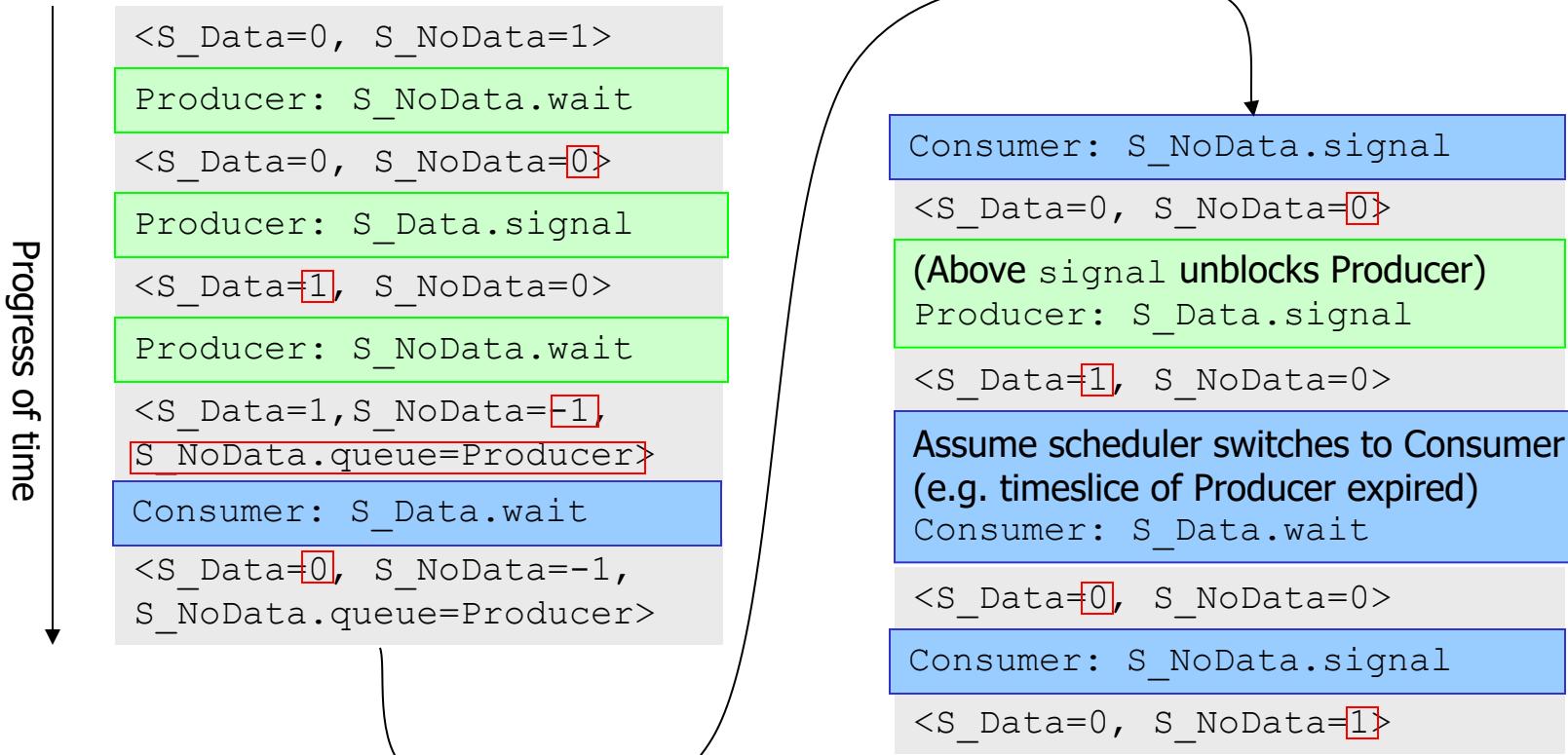
void Producer() {
    while (true) {
        S_NoData.wait();
        /* Produce item */
        S_Data.signal();
    }
}

void Consumer() {
    while (true) {
        S_Data.wait();
        /* Consume item */
        S_NoData.signal();
    }
}

void main() {
    parallel { Producer() } { Consumer() };
}
```

Semaphores (8): Example Producer-Consumer Problem

- Sample trace for the previous example:



Now, execution might repeat from the beginning of this trace (or many other traces possible as well).

Semaphores (9): Example Producer-Consumer Problem

- The following slide will provide a solution of the bounded producer-consumer problem with an arbitrary size of the buffer. For this, we need three different semaphores:
 - S_ME: Achieves mutual exclusion when accessing shared buffer that contains the produced items.
 - S_Free: Counts number of free slots in the buffer.
 - Used for condition synchronisation: wait if no free slots in the buffer.
 - S_Prod: Counts number of produced items in the buffer.
 - Used for condition synchronisation: wait if no items in the buffer.
 - (Values of S_Free and S_Prod will be used for condition synchronisation and will grow and shrink inverse to each other.
 - Two semaphores necessary because of two conditions: 1. buffer full,
2. buffer empty. For any buffer fill state in-between, producer and consumer can run concurrently and thus a third semaphore is needed to achieve mutual exclusive access to shared buffer.)

Semaphores (10): Example Producer-Consumer Problem

- Bounded producer-consumer problem with arbitrary buffer size:

```
semaphore S_Prod = S_Prod.init(0);           // no. of produced items
semaphore S_Free = S_Free.init(capacity);    // capacity: buffer size
semaphore S_ME = S_ME.init(1);                // mutual exclusion

void Producer() {
    while (true) {
        S_Free.wait();
        S_ME.wait();
        /* Produce item */
        S_ME.signal();
        S_Prod.signal();
    }
}

void Consumer() {
    while (true) {
        S_Prod.wait();
        S_ME.wait();
        /* Consume item */
        S_ME.signal();
        S_Free.signal();
    }
}

void main() {
    parallel { Producer(); } { Consumer(); }
}
```

- Sample trace for a buffer size of 2 will be shown on next slide...

Semaphores (11)

<S_Prod=0, S_Free=2, S_ME=1>	Consumer: S_Prod.wait	Assume: producer gets CPU now (e.g. timeslice of consumer expired now).
Producer: S_Free.wait	<S_Prod=1, S_Free=-1, S_ME=1, S_Free.queue=Producer>	Producer: S_ME.wait
<S_Prod=0, S_Free=1, S_ME=1>	Consumer: S_ME.wait	<S_Prod=0, S_Free=1, S_ME=0>
Producer: S_ME.wait	<S_Prod=1, S_Free=-1, S_ME=0, S_Free.queue=Producer>	Producer: S_ME.signal
<S_Prod=0, S_Free=1, S_ME=0>	Producer: S_ME.signal	<S_Prod=0, S_Free=1, S_ME=1>
Producer: S_Prod.signal	Consumer: S_ME.signal	Producer: S_Prod.signal
<S_Prod=0, S_Free=1, S_ME=1>	<S_Prod=1, S_Free=-1, S_ME=1, S_Free.queue=Producer>	<S_Prod=1, S_Free=1, S_ME=1>
Producer: S_Prod.signal	Consumer: S_Free.signal	Assume: consumer gets CPU.
<S_Prod=1, S_Free=1, S_ME=1>	<S_Prod=1, S_Free=0, S_ME=0>	Consumer: S_Prod.wait
Producer: S_Free.wait	Producer gets unblocked, but assume: producer does not get CPU, yet (e.g. timeslice of consumer not yet expired).	<S_Prod=0, S_Free=1, S_ME=1>
<S_Prod=1, S_Free=0, S_ME=1>	Consumer: S_Prod.wait	Consumer: S_ME.wait
Producer: S_ME.wait	<S_Prod=0, S_Free=0, S_ME=1>	<S_Prod=0, S_Free=1, S_ME=0>
<S_Prod=1, S_Free=0, S_ME=0>	Consumer: S_Prod.wait	Consumer: S_ME.signal
Producer: S_ME.signal	<S_Prod=0, S_Free=0, S_ME=1>	<S_Prod=0, S_Free=1, S_ME=1>
<S_Prod=1, S_Free=0, S_ME=1>	Consumer: S_ME.wait	Consumer: S_Free.signal
Producer: S_Prod.signal	<S_Prod=0, S_Free=0, S_ME=0>	<S_Prod=0, S_Free=2, S_ME=1>
<S_Prod=2, S_Free=0, S_ME=1>	Consumer: S_ME.signal	Now, execution might repeat from the beginning of this trace (or many other traces are possible as well).
Producer: S_Free.wait	<S_Prod=0, S_Free=0, S_ME=1>	dated
<S_Prod=2, S_Free=-1, S_ME=1, S_Free.queue=Producer>	Consumer: S_Free.signal	
Producer gets blocked	<S_Prod=0, S_Free=1, S_ME=1>	

Semaphores (12): Implementation of Atomicity

- wait and signal operations of a semaphore must be atomic.
 - Semaphores are implemented by the entity that manages the waiting and ready queues of the scheduler: typically, the OS kernel (or a user-level thread library).
 - However, even functions provided by the OS kernel are not atomic:
 - Kernel may be interrupted by hardware interrupts.
 - In multi-processor/multi-core systems, another processor/core may concurrently execute instructions and modify memory.
- ⇒ In single processor/core systems, the OS kernel typically **disables interrupts** (no busy waiting) to implement atomicity of wait and signal system call.
- ⇒ On multi-processor/multi-core systems, the OS kernel typically uses **atomic test-and-set operations** (together with busy waiting) to make wait and signal system calls mutual exclusive.

Semaphores (13): Further Types of Semaphores

- **Mutex/binary semaphore:**
 - A semaphore that has no generic counter variable, but that only controls mutual exclusion.
 - “Mutex” refers to “mutual exclusion”.
 - “Binary semaphore” refers to the fact that the semaphore has just two states and does not really need an integer counter:
 - count=1, if critical section is free,
 - count=0, if critical section is occupied (no negative values required, because it is only relevant whether critical section is free or not. However, there is still a queue of waiting process in case multiple processes do call wait operation while the critical section is occupied).
- **Counting semaphore:**
 - Term is used for generic semaphores that are used with values >1 .

Semaphores (14): Further Types of Semaphores

- **Weak semaphore:**
 - A semaphore where the order of processes in the semaphore queue has no influence on the order in which processes are woken up again (=no FIFO queue discipline).
 - Most OSes/semaphore APIs offer only weak semaphores!

- Note: sometimes, the term “weak semaphore” refers to the fact, that after being woken up, you need to re-check the condition you were waiting for (and if necessary: do a wait again).

- **Strong semaphore:**
 - A semaphore using FIFO queue discipline concerning the order in which processes are woken up again.

Semaphores vs. Spinlocks (1)

- Semaphores are nice, because they avoid busy waiting.
 - However, for performing the semaphore operations, system calls are required: System calls are “expensive” (slow, because they involve, e.g., context switch which is time consuming).
 - On single processor/single core system, there are not much alternatives: if wait operation would block, we would anyway need a context switch, because after blocking our process, the scheduler has to give the CPU to other (ready) processes that eventually unblock our waiting process.
 - However, on SMP systems, there may be situations where it is reasonable to avoid “expensive” semaphore operations and to use a busy waiting instead:
if the busy waiting is shorter than a system call and the other CPUs/cores may execute processes that eventually unblock our waiting process. ⇒ Spinlocks (→next slide)

Semaphores vs. Spinlocks (2)

- **Spinlock**: Mutual exclusion based on **busy waiting** (typically, using atomic test-and-set instruction →6-49...) that **avoids** an “expensive” system call.
 - Reasonable **only in SMP systems**, where one CPU/core executes a process that is in its critical section and the other CPU/core performs busy waiting (which is faster than a semaphore if the critical section(=busy waiting for it) is short).
 - Spinlocks are typically used by an SMP kernel itself when accessing its kernel data structures shared by the CPU cores.

The Java API for Semaphores

- Since Java 1.5 (or Java 5 respectively), Java supports semaphores that can be used by Java threads.
- Class **Semaphore** with corresponding methods **acquire()** and **release()** provided by Java API package **java.util.concurrent.Semaphore**
- Example:

```
import java.util.concurrent.Semaphore;  
...  
Semaphore sem = new Semaphore(1); // Initial semaphore value: 1  
...  
try {  
    sem.acquire();  
    // Start of critical section  
    ...  
    // End of critical section  
    sem.release();  
} catch (InterruptedException ie) { // Handle deferred cancellation,  
// i.e. someone called interrupt() while we were waiting in  
// sem.acquire() (see "Thread cancellation" in chapter 4)  
}
```

Semaphore to be shared by multiple threads

"wait" operation

"signal" operation

POSIX Pthreads API for Semaphores (for info)

- Initialise semaphore:
`int sem_init(sem_t *sem, int pshared, unsigned int value)`
Predefined data structure for semaphore to be initialised
- Signal on semaphore:
`int sem_post(sem_t *sem)`
Semaphore to signal on
- Wait on semaphore:
`int sem_wait(sem_t *sem)`
Semaphore to wait on
- Perform semaphore wait only if it will not block,
otherwise return immediately with EAGAIN errno:
`int sem_trywait(sem_t *sem)`
- Perform semaphore wait only until absolute time,
after that return with ETIMEDOUT errno:
`int sem_timedwait(sem_t *sem, timespec *abstime)`
- Get just value of semaphore counter:
`int sem_getvalue(sem_t *sem, int *sval)`
Pointer to memory location for storing the semaphores value
- Release all ressources having created by OS at `sem_init`:
`int sem_destroy(sem_t *sem)`
Semaphore to deallocate

Semaphore shared between threads of same process only or between different processes (provided `*sem` memory location needs then to be in shared memory).

Initial value

Pretty useless operation:
value is likely already outdated when you do something based on it. Better use atomic operations
`sem_signal/sem_wait/sem_trywait/ sem_timedwait`

Self-check questions: Semaphores

- After having read this chapter, you should be able to explain and apply semaphores for process synchronisation to create synchronised concurrent programs.
- In the following, some simple questions are given (and solutions on the following slides): they refer to using semaphores in pseudo code.
 - Try to answer these questions on your own (before looking at the solution).
 - If you are not able to answer: re-read again, check videos, make a comment on Piazza and ask at our weekly meetings.

Semaphores: Question

- How do you have to change (by adding semaphores) the following pseudo code to make the execution of P1 and P2 mutual exclusive?

```
int a = 1;           ← Shared variables
parallel {
    P1 () {
        a = a + 1;
    }

    P2 () {
        a = 2*a;
    }
}
```

Semaphores: Solution

- Introduce semaphore initialised with 1 (=one process may enter immediately),
- wait() in front of mutual exclusive section,
- signal() at end of mutual exclusive section.

```
int a = 1;
semaphore mutex = mutex.init(1);
parallel {
    P1 () {
        mutex.wait();
        a = a + 1;
        mutex.signal();
    }

    P2 () {
        mutex.wait();
        a = 2*a;
        mutex.signal();
    }
}
```

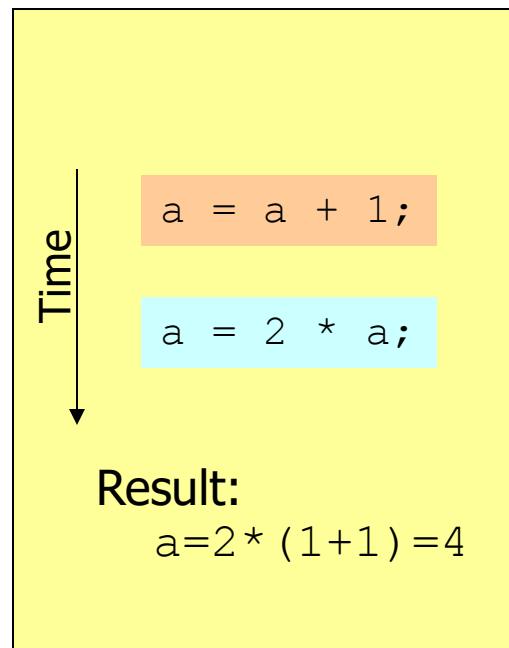
Semaphores: Question

- While mutual exclusion of P1 and P2 is nice, the result is still not deterministic (=there is still a race between P1 and P2 possible and depending who wins the race, different results are possible).
- How to you have to change the code (using semaphores) so that $a=2*a$ in P2 is always executed after $a=a+1$ in P1?

```
int a = 1;
```

```
parallel {
    P1 () {
        a = a + 1;
    }
}
```

```
P2 () {
    a = 2*a;
}
```



Semaphores: Solution

- Introduce semaphore initialised with 0 (=P2 process has to wait in case it wins the race), i.e. condition synchronisation for condition “P1 finished”.
- `wait()` in front of `a=2*a`,
- `signal()` at end of `a=a+1`.

```
int a = 1;
semaphore cond = cond.init(0);
parallel {
    P1() {
        a = a + 1;
        cond.signal();
    }

    P2() {
        cond.wait();
        a = 2*a;
    }
}
```

Semaphores: Question

- What is wrong with the following Semaphore-based code that is supposed to achieve mutual exclusion of P1 and P2?

```
int a = 1;
semaphore mutex = mutex.init(0);
parallel {
    P1() {
        mutex.wait();
        a = a + 1;
        mutex.signal();
    }

    P2() {
        mutex.wait();
        a = 2*a;
        mutex.signal();
    }
}
```

Semaphores: Solution

- The semaphore is initialised with 0 and thus both P1 and P2 will go to sleep and as there are no other processing signalling on the semaphore, they will sleep forever.

```
int a = 1;
semaphore mutex = mutex.init(0);
parallel {
    P1() {
        mutex.wait();
        a = a + 1;
        mutex.signal();
    }

    P2() {
        mutex.wait();
        a = 2*a;
        mutex.signal();
    }
}
```

Semaphores: Question

- What is wrong with the following semaphore-based code that is supposed to achieve condition synchronisation so that P2 waits for P1?

```
int a = 1;
semaphore cond = cond.init(1);
parallel {
    P1() {
        a = a + 1;
        cond.signal();
    }

    P2() {
        cond.wait();
        a = 2*a;
    }
}
```

Semaphores: Solution

- The semaphore is initialised with 1 and thus the wait() by P2 will not make P2 go to sleep and hence P2 will not wait for P1.

```
int a = 1;
semaphore cond = cond.init(1);
parallel {
    P1 () {
        a = a + 1;
        cond.signal();
    }

    P2 () {
        cond.wait();
        a = 2*a;
    }
}
```

Want to play even more computer games? The Return of the Deadlock Empire!

- Browser game: <http://deadlockempire.github.io/>
 - Reminder: in these games always the yellow background line will be executed next.
 - Sometimes, there are white background lines, but these are in fact supposed to be transparent background lines.
- Now, play the three “Semaphores” games:
 - Semaphores
 - Producer-Consumer
 - Producer-Consumer (variant)
 - Notes:
 - Initialisation of semaphores not shown there: default value is 0.
 - C# is used: `.Release()` operation is our semaphore “signal” operation.
 - `.Wait(relativeTime)` is a “wait” operation that blocks only for the given amount of milliseconds.
 - Dequeuing from an empty queue will raise an exception.
 - Do not forget to expand a statement where possible.



Problems of Semaphores (1)

- Disadvantage of semaphores:
 - Even though (or maybe just because) they are quite simple, **it is easy to create wrong solutions using them.**
 - Incorrect solutions may lead to a deadlock of all involved processes (see example on next slides).

Problems of Semaphores (2): Example

- Example for an error made when trying to solve the producer-consumer problem (in comparison to the correct solution on slide 6-68, just two statements are swapped):

```
semaphore S_Prod = S_Prod.init(0);           // no. of produced items
semaphore S_Free = S_Free.init(capacity);    // capacity: buffer size
semaphore S_ME = S_ME.init(1);                // mutual exclusion
```

```
void Producer() {
    while (true) {
        S_ME.wait();
        S_Free.wait();
        /* Produce item */
        S_ME.signal();
        S_Prod.signal();
    }
}
```

```
void main() {
    parallel { Producer() } { Consumer() };
}
```

```
void Consumer() {
    while (true) {
        S_Prod.wait();
        S_ME.wait();
        /* Consume item */
        S_ME.signal();
        S_Free.signal();
    }
}
```

Statements swapped

/* Produce item */

S_ME.signal();
S_Prod.signal();

}

}

■ Problematic example trace with buffer size of 2 shown on next slide...

Problems of Semaphores (3)

<S_Prod=0, S_Free=2, S_ME=1>

Producer: S_ME.wait

<S_Prod=0, S_Free=2, S_ME=0>

Producer: S_Free.wait

<S_Prod=0, S_Free=1, S_ME=0>

Producer: S_ME.signal

<S_Prod=0, S_Free=1, S_ME=1>

Producer: S_Prod.signal

<S_Prod=1, S_Free=1, S_ME=1>

Producer: S_ME.wait

<S_Prod=1, S_Free=1, S_ME=0>

Producer: S_Free.wait

<S_Prod=1, S_Free=0, S_ME=0>

Producer: S_ME.signal

<S_Prod=1, S_Free=0, S_ME=1>

Producer: S_Prod.signal

<S_Prod=2, S_Free=0, S_ME=1>

Producer: S_ME.wait

<S_Prod=2, S_Free=0, S_ME=0>

Producer: S_Free.wait

<S_Prod=2, S_Free=-1, S_ME=0,
S_Free.queue=Producer>

Producer sleeps

Consumer: S_Prod.wait

<S_Prod=1, S_Free=-1, S_ME=0,
S_Free.queue=Producer>

Consumer: S_ME.wait

<S_Prod=1, S_Free=-1, S_ME=-1,
S_Free.queue=Producer
S_ME.queue=Consumer>

Consumer sleeps

***Deadlock, both

processes are sleeping***

Monitors (1)

- Reason for possible problems when using semaphores:
 - Processes (or their developers) themselves are responsible for achieving the synchronisation.
- Hoare (1974) and Brinch Hansen (1975) suggested a higher level synchronisation mechanism:
 - Monitors (has nothing to do with a computer display):
 - A programmer only specifies **that** synchronisation is required, but **not how** to achieve it.
 - The implementation is provided by the higher level mechanism.

Monitors (2)

- Monitor construct is based on the encapsulation of resources (comparable to an object-oriented class).
 - A monitor is a **collection of variables** (=shared resources) and **operations to access these variables**.
 - Information hiding: resource cannot be directly accessed from outside.
 - Processes can access resource only via operations of the monitor.
 - Only one process at a time can be active in the monitor (i.e. mutual exclusion).
- Just like object-oriented classes are a programming language concept, **monitors are a programming language concept**.
 - I.e. the compiler (and not the programmer who is likely to introduce errors) is responsible for achieving the mutual exclusion.
 - E.g. by generating code that internally uses semaphores provided by OS.

Monitors (3): Mutual Exclusion Example

- Mutual exclusive access to a critical region that accesses a shared resource (e.g. a variable) contained in a monitor (pseudocode):

```
monitor myVarMonitor
    private int myResource;
    ...
    public void accessResource() {
        /* critical region
         * where variable myResource
         * is accessed
         */
    }
}
```

Process A:

```
while(true) {
    myVarMonitor.accessResource();
}
```

Process B:

```
while(true) {
    myVarMonitor.accessResource();
}
```

Call to this operation (and any other operation) of the monitor is mutual exclusive (only one process-call at a time can be active within the monitor).

Monitors (4): Condition Synchronisation

- Achieving mutual exclusion using monitors is really easy and convenient.
 - Compiler and run-time system of a programming language that supports monitors take care of the nasty and error-prone implementation details.
- However, does not allow to synchronise with respect to conditions.
 - I.e. one process wakes up another if a certain condition gets true.
 - Well, you could do this using a Boolean variable `condition` together with operations `setConditionToTrue()` and `isConditionTrue()`: Process B probes `isConditionTrue()` periodically in a busy waiting style until process A calls `setConditionToTrue()`. However, this would be busy waiting – and furthermore this would in fact not work inside the monitor, because process A would not be able to change the condition while process B is probing it (due to the mutual exclusion that a monitor provides!).
- Condition synchronisation provided by monitors:
 - Special **Condition variables** on which two special operations can be applied:
 - `.wait` to put the calling process to sleep until another process calls `.signal`.
 - `.signal` to wake up one process that has called `.wait` on the same condition variable.

Monitors (5): Condition Synchronisation Example

- Pseudocode example that defines a condition variable `myCondition` and uses `.wait` and `.signal`:

```
monitor example
    condition myCondition;
    ...
    public void op1() {
        ...
        myCondition.wait; // Blocks the calling process
        ...               // until condition is signalled
    }      // (During this, others can enter the monitor)
    public void op2() {
        ...
        myCondition.signal; // Unblocks the process that
        ...                 // did a myCondition.wait
    }
}
```

Even though method execution in a monitor is mutual exclusive, when this `wait` blocks, the method is put to sleep and others may then enter the monitor while this `wait` is blocked.

What to do after the `signal`? We definitely have to avoid that mutual exclusion is violated, i.e. the process that called `op1()` and the process that called `op2()` must not execute at the same time instructions within the monitor!

Monitors (6): Condition Synchronisation

- How to proceed after the signal operation has been issued?
- Three possible solutions:
 - “Signal-and-return” (suggested by Brinch Hansen):
 - Signal is the last statement of a monitor operation, i.e. signalling leaves the monitor.
 - “Signal-and-wait” (suggested by Hoare):
 - The signalling process is suspended until the unblocked formerly waiting process leaves the monitor.
 - “Signal-and-continue” (used by, e.g., Java):
 - The waiting process is only unblocked after the signalling process left the monitor.

Monitors (7): Condition Synchronisation

- **wait** and **signal** are to some extent like the **sleep** and **wakeup** operations on slide 6-56 and not that much like semaphores.
- Comparison to **sleep** and **wakeup**:
 - A monitor's **signal** for which no **wait** is performed, gets lost just like with **sleep** and **wakeup**.
 - Note: the problem in the **sleep** and **wakeup** example was not only that a **sleep** got lost, but that a "test-and-sleep" sequence was not atomic.
 - (Note: To motivate semaphores, slide 6-60 claimed for teaching purposes that the problem was a lost signal – but in fact it was also because of the lacking atomicity!)
 - Monitors provide mutual exclusion, hence a "test-and-wait" or "test-and-signal" sequence cannot get interrupted, but is rather atomic!
- Comparison to **semaphores**:
 - A monitor's **signal** for which no **wait** is performed, gets lost – in contrast to a semaphore's **signal** and **wait** operations.
 - Condition variables are not counters in contrast to semaphores – they can only be used for conveying a signal.

Monitors (8): Example Producer-Consumer Problem

- Solution for producer-consumer problem by implementing the shared buffer of size N using a monitor with condition variables:
 - Signal-and-continue semantics is assumed.
 - (Implementation of actual buffer data structure for items and corresponding insert/remove operations not shown.)
 - Note: solution quite similar to buggy sleep & wakeup solution (6-57 & 6-58). However, this time we cannot get interrupted during the **if** and the **wait** (because of the atomicity provided by a monitor)!

Pseudocode:

```
monitor ProducerConsumerBuffer {  
    condition full, empty;  
    private int count = 0;  
  
    public void insert(int item) {  
        if (count == N) full.wait;  
        insert_item(item);  
        count = count + 1;  
        if (count == 1) empty.signal;  
    }  
  
    public int remove() {  
        if (count == 0) empty.wait;  
        int item = remove_item();  
        count = count - 1;  
        if (count == N-1) full.signal;  
        return item;  
    }  
}
```

Monitors (9): Example Producer-Consumer Problem

- Producer and consumer processes that call monitor operations and thus produce and consume in a synchronised manner:

```
void producer() {  
    int item;  
    while (true) {  
        item = produce_item();  
        ProducerConsumerBuffer.insert(item);  
    }  
}  
  
void consumer() {  
    int item;  
    while (true) {  
        item=ProducerConsumerBuffer.remove();  
        consume_item(item);  
    }  
}
```

Monitors in Java (1)

- Monitors are supported by the Java language in order to synchronise Java threads:
 - Every Java **object** has internally a **lock** flag.
 - A method that is declared as **synchronized** can only be executed if that lock is not set.
 - Otherwise, the thread that wants to execute that method is blocked.
 - If a method that is declared as **synchronized** executes, the lock flag gets set and is reset after the method is left.
 - If there are blocked threads that are waiting for the object's lock to be reset, one of them becomes the new "owner" of the lock and may enter the **synchronised** method it was waiting for.
 - Mutual exclusion is achieved for all **synchronized** methods of the same object.
 - For all other (i.e. non **synchronised**) methods of an object, this monitor property does not apply: they may be entered concurrently.

Monitors in Java (2): Example

```
class SynchronizedCounter {
    private int c = 0;

    public synchronized void increment() {
        c++;
    }

    public synchronized void decrement() {
        c--;
    }

    public int value() {
        return c;
    }
}
```

- Typically, you would pass the same instance of such a class to different threads.
 - Due to the **synchronized** keyword, access to the counter by different threads will be mutual exclusive.
 - Note: **internal lock flag is object specific, not class specific.**
Hence, mutual exclusion is only guaranteed within the same instance of a class.
 - Well, for **static methods**, there is also a **class specific (**static**) internal lock flag.**

Monitors in Java (3): Condition Variables

- There is no special data type for condition variables. Instead, each object has automatically exactly one unnamed (anonymous) condition variable:
 - Predefined methods `this.wait()`, `this.notify()` and `this.notifyAll()` use this unnamed condition variable.
 - These methods can only be called from within **synchronized** methods.
 - The notify methods correspond to the previously described signal operations of a monitor. (More on them on the next slides.)
 - The **signal-and-continue approach** is used for the notify methods, i.e. a waiting method is only unblocked after a signalling method has finished.
 - To make thread cancellation (→chapter 4) possible while a thread is blocked in a `wait()` method, `wait()` will throw an exception when `interrupt()` is called on a thread while waiting.
 - `wait()` must be surrounded by
`try ... catch(InterruptedException)`.

Monitors in Java (4): Condition Variables

- When a thread calls `wait()`:
 - The thread is set to the blocked state.
 - The lock of the object is released.
 - So that other threads can use `notify()`/`notifyAll()` in their `synchronized` methods in order to wake up this waiting thread.
 - The `thread is placed in a wait set of threads`.
 - Like the lock flag, each instance/each object has its own wait set of threads that are waiting to be notified.
 - One condition variable per object \Rightarrow one wait set per object.

Monitors in Java (5): Condition Variables

- When a thread calls `notify()`:
 - One arbitrary thread t is picked and removed from the wait set of the object.
 - This thread t is set from the blocked state to the runnable state.
 - Once, the notifying thread finishes the current method, the lock is released.
 - Now, either the picked waiting thread t or any other further runnable thread that competes for the lock flag will become the owner of the lock and resume execution.
 - ⇒ Exactly one waiting thread will be woken up.
- When a thread calls `notifyAll()`:
 - All threads are removed from the wait set of the object and set from the blocked state to the runnable state.
 - Once, the notifying thread finishes the current method, the lock is released.
 - Now, either one of notified waiting threads or any other further runnable thread that competes for the lock flag will become the owner of the lock and resume execution.
 - ⇒ All waiting threads will be woken up. (However, as the wait occurs within a synchronized method, these threads will not get immediately executed, but only after another thread that just got the lock releases the lock again, one of these can run.)

Monitors in Java (6): Example Condition Variables

- A example of a monitor that can be used to inform a consumer that an item has been produced:

```
class MyMonitor {  
    int noOfItems = 0;  
    public synchronized void produce() {  
        noOfItems++;  
        notify();  
    }  
    public synchronized void consume() {  
        if (noOfItems == 0) {  
            try {  
                wait();  
            } catch (InterruptedException e) { }  
        }  
        noOfItems--;  
    }  
}
```

Note: only one condition variable for each object supported in Java
⇒ Using multiple conditions requires multiple objects and you can then call some of the synchronized methods of these object!

- Typically, you pass the same instance of such a class to different threads.

Monitors in Java (7): Example Condition Variables

- Java monitors uses the signal-and-continue approach \Rightarrow it may be the case that after the `notify()` has been made, the condition on which the `wait()`ing thread is waiting has been invalidated in the meantime (i.e. when being woken up, condition is invalid again).
 \Rightarrow Enclose the `wait()` in a conditional loop, i.e. if we are woken up from the `wait()`, we check first whether the condition to continue is still given, otherwise we `wait()` again.

```
class MyMonitor {  
    int noOfItems = 0;  
    public synchronized void produce() {  
        noOfItems++;  
        notify();  
    }  
    public synchronized void consume() {  
        while (noOfItems == 0) {  
            try {  
                wait();  
            } catch (InterruptedException e) {}  
        }  
        noOfItems--;  
    }  
}
```

- See also: <http://docs.oracle.com/javase/8/docs/api/java/lang/Object.html#wait-->
I.Hjörleifsson ingolfuh@hi.is st. TG225

Self-check questions: Monitors

- After having read this chapter, you should be able to explain and apply monitors for process synchronisation to create synchronised concurrent programs.
- In the following, some simple questions are given (and solutions on the following slides): they refer to usage of the Java monitor concept.
 - Try to answer these questions on your own (before looking at the solution).
 - If you are not able to answer: re-read again, check videos, make a comment on Piazza and ask at our weekly meetings.

Monitors with Java: Question

- How do you have to change the following Java class to make the method `advance()` mutual exclusive (so that no two threads that use the same instance of that class can execute that method at the same time)?

```
class Clock {  
    private int time = 0;  
  
    public void advance() {  
        time = time + 1;  
    }  
  
    public int getTime() {  
        return time;  
    }  
}
```

Solution on next slide

Monitors with Java: Solution

- Add keyword **synchronized** to the respective method:

```
class Clock {  
    private int time = 0;  
  
    public synchronized void advance() {  
        time = time + 1;  
    }  
  
    public int getTime() {  
        return time;  
    }  
}
```

Monitors with Java: Question

- How do you have to change the following Java class to make the caller of the method `goodNight()` go to sleep if field `time` has the ≥ 22 ?

```
class Clock {  
    private int time = 0;  
  
    public synchronized void advance() {  
        time = time + 1;  
    }  
  
    public void goodNight() {  
    }  
  
    public int getTime() {  
        return time;  
    }  
}
```

Solution on next slide

Monitors with Java: Solution

- Use `wait()` (which must be called from the context of synchronized method).

```
class Clock {  
    private int time = 0;  
  
    public synchronized void advance() {  
        time = time + 1;  
    }  
  
    public synchronized void goodNight() {  
        if (time >= 22) { // Even better would be while instead of if  
            try {  
                wait();  
            } catch (InterruptedException e) {}  
        }  
    }  
  
    public int getTime() {  
        return time;  
    }  
}
```

Monitors with Java: Question

- How do you have to change the following Java method `wakeUpOneSleeper()` to wake up just one thread that is sleeping due a `goodNight()` method call that has been made on the same instance of this class?

```
class Clock {  
    ...  
    public synchronized void goodNight() {  
        if (time >= 22) { // Even better would be while instead of if  
            try {  
                wait();  
            } catch (InterruptedException e) {}  
        }  
    }  
  
    public void wakeUpOneSleeper() {  
        ...  
    }  
}
```

Solution on next slide

Monitors with Java: Solution

- Use `notify()` (which must be called from the context of synchronized method).

```
class Clock {  
    ...  
    public synchronized void goodNight() {  
        if (time >= 22) { // Even better would be while instead of if  
            try {  
                wait();  
            } catch (InterruptedException e) {}  
        }  
    }  
  
    public synchronized void wakeUpOneSleeper() {  
        notify();  
    }  
}
```

Monitors with Java: Question

- How do you have to change the following Java method `wakeUpAllSleepers()` to wake up all threads that are sleeping due a `goodNight()` method call that has been made on the same instance of this class?

```
class Clock {  
    ...  
    public synchronized void goodNight() {  
        if (time >= 22) { // Even better would be while instead of if  
            try {  
                wait();  
            } catch (InterruptedException e) {}  
        }  
    }  
  
    public synchronized void wakeUpOneSleeper() {  
        notify();  
    }  
    public void wakeUpAllSleepers() {  
        ...  
    }  
}
```

Solution on next slide

Monitors with Java: Solution

- Use `notifyAll()` (which must be called from the context of synchronized method).

```
class Clock {  
    ...  
    public synchronized void goodNight() {  
        if (time >= 22) { // Even better would be while instead of if  
            try {  
                wait();  
            } catch (InterruptedException e) {}  
        }  
    }  
  
    public synchronized void wakeUpOneSleeper() {  
        notify();  
    }  
    public synchronized void wakeUpAllSleepers() {  
        notifyAll();  
    }  
}
```

Want to play more computer games? The Deadlock Empire awakens! (1)

- Browser game: <http://deadlockempire.github.io/>
 - Reminder: in these games always the yellow background line will be executed next.
 - Sometimes, there are white background lines, but these are in fact supposed to be transparent background lines.
- Note that some games use the C# API for synchronisation (which we did not cover):
 - Hence, you probably want to skip “High-Level Synchronization Primitives” (and “The Final Stretch”),
 - but you should still be able to play the games described on the two next slides!



If you are stuck in the games on the next two slides,
watch Panopto videos with solutions

Want to play more computer games? The Deadlock Empire awakens! (2)

- Now play the “Locks” games which are somewhat similar to monitors without condition variables, but you have to explicitly write `Monitor.Enter(mutex)` and `Monitor.Exit(mutex)` using an object such as `mutex` that you pass in and use as lock.
- Notes:
 - Even though all games have a critical section, in none of the games it will be possible to be in the critical sections at the same time (they are well protected by the locks). Instead the goal in the respective game is:
 - In “Insufficient Lock” you win by reaching line `Debug.Assert(false)`;
 - In “Deadlock” two lock objects are used (which you do not have in Monitors: there each monitor has exactly one implicit lock that is automatically used) and you win by provoking a deadlock, i.e. each process waiting on the other.
 - In “A More Complex Thread”, you win as well if manage to provoke a deadlock, i.e. achieving that none of the processes can progress.

Want to play more computer games? The Deadlock Empire awakens! (3)

- Now play the “Condition Variables” game which is in fact about monitors with condition variables:
 - `Monitor.Enter(mutex)` and `Monitor.Exit(mutex)` use the lock of the `mutex` object are just like entering and leaving a `synchronized` method in Java.
 - `Monitor.Wait(mutex)` waits on the condition variable of the `mutex` object and is thus like the `wait()` in Java.
 - In particular (just like in Java), a `wait()` that blocks does unlock the lock while waiting.
 - `Monitor.PulseAll(mutex)` is like the `notifyAll()` in Java.
- Note:
 - In the only existing game “Condition Variables” you win, if you manage to `queue.Dequeue()` ; on an empty queue.

6.6 Alternative Approaches

- Implicit synchronisation (\approx mutual exclusion with monitors) in [OpenMP](#) (\rightarrow 4.5):

```
void update(int value) {  
    ...  
    #pragma omp critical  
    {  
        count += value  
    }  
    ...  
}
```

Following {...} block contains critical section for which mutual exclusion is achieved.

- [Functional programming](#) languages (e.g. Scala, Erlang) offer a different paradigm than imperative (or procedural) languages in that they do not allow shared global variables with changing state (leading to race conditions).
 - There is increasing interest in functional languages such as Erlang and Scala: support threads/concurrency extremely well!
 - Might be the solution to multicore programming, but functional language paradigm hard to learn once you have been spoiled with imperative programming.
- Further alternative approach: Message passing \rightarrow next slides.
 - While semaphores and monitors work only locally, works also across network.

6.7 Classical Problems of Synchronisation

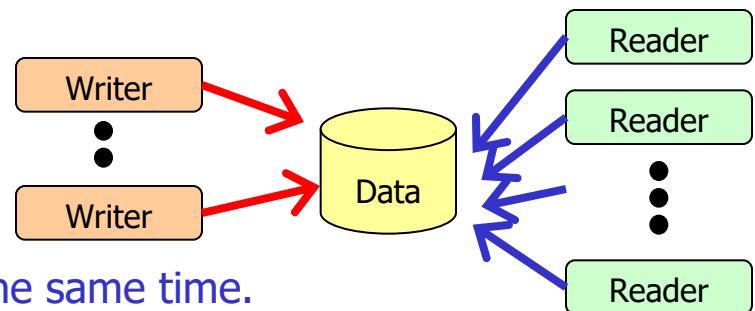
- Like the Producer-Consumer problem, there are further classical synchronisation problems:
 - Bounded-Buffer problem,
 - Readers-Writers problem,
 - Dining-Philosophers problem,
 - Sleeping-Barber problem.
- At first glance, not all of these problems are relevant for real-life SW development. Nevertheless, they are –like the producer-consumer problem– used to study synchronisation problems and solutions.
 - Every computer scientist/software engineer should have heard about them, hence they are presented in the following.
(However, without a solution.)

Bounded-Buffer Problem

- In fact, more or less the Producer-Consumer problem.
 - As the name implies, refers to that variant of the [Producer-Consumer problem](#) that uses a bounded buffer, i.e. buffer size is neither ∞ nor zero.
 - However, the Bounded-Buffer problem itself does not involve the actual production and consumption of items, but only that part of the solution of the Producer-Consumer problem that implements the actual buffer data structure and the synchronised access to it.
 - (=condition synchronisation and mutual exclusion).
 - Solution: See previous slides on semaphores, monitors, message passing.

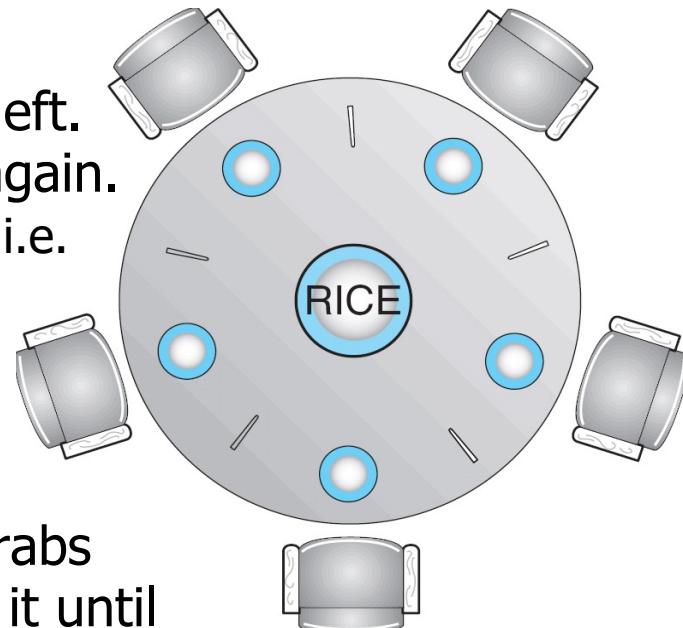
Readers-Writers Problem

- A data set (e.g. a file in a file system or a table in a data base) is shared among a number of concurrent processes:
 - Readers processes: only read the data set; they do **not** perform any updates.
 - Writers processes: can write (update) the data set; during write, the data is in an inconsistent state.
- Readers-Writers problem:
 - Avoid that multiple writers are writing at the same time.
 - Allow multiple readers to read at the same time.
 - Avoid that readers are reading and a writer is writing at the same time.
- Variants of the Readers-Writers problem (starvation possible):
 - “First Readers-Writers” problem: readers have priority, i.e. even if a writer is already waiting for readers to finish reading, new readers may join reading.
 - “Second Readers-Writers” problem: writers have priority, i.e. if a writer is already waiting for readers to finish reading, new readers may not join reading.



Dining-Philosophers Problem

- Five independent philosophers share a table with five shared chopsticks: each philosopher thinks for a while, then decides to eat for a while, think, eat, and so on.
- For eating, a philosopher has to grab one chopstick from the right and one from the left. After eating, the chopsticks are put down again.
 - Grabbing a pair of chopsticks is not atomic, i.e. while grabbing the chopstick from the right, the philosopher to the left may grab the second chopstick.
- A solution must deal with:
 - **Deadlocks** (imagine each philosopher grabs its right chopstick and does not release it until (s)he gets the chopstick to the left as well),
 - **Starvation / fairness**: (a philosopher that wants to eat, finally gets access to the left and right chopstick).



6.8 Summary

- A **critical section** of code accesses shared resources or data.
- Interacting processes need to be synchronised to avoid race conditions: **mutual exclusion** or **condition synchronisation**.
- **Software-based** solutions (Peterson algorithm), **Hardware-based** solutions (disabling interrupts, atomic instructions), or **Operating System-based** (Semaphores) / **Programming Language-based** solutions (Monitors) possible.
 - Higher-level operating system/programming language-based solutions are implemented using hardware solutions.
 - Operating system/programming language-based solutions restrict the possible schedules (by blocking some processes which are then not considered anymore by the scheduler unless they get unblocked).
 - Unless you are using Java, semaphores are used in practise instead of monitors (because not many languages support monitors):
 - E.g. using the semaphore API provided by the POSIX Pthreads library (POSIX semaphore calls not presented in this course).
 - Since Java (1.5), the Java API supports also semaphores.

Not discussed in this course (but in the Silberschatz book)

- Atomic transactions:
 - A set of operations (e.g. updating data) shall be either **committed** successfully or when it was (for whatever reason) unsuccessful, it is not simply aborted, but all the changes that have already applied as part of this set of operations are **rolled back**. (I.e. either it is executed completely or it is not executed at all.)
- Atomic transactions are typically discussed in a course on database systems.
 - However, we will in a later chapter on filesystems cover an example of some sort of atomic transactions when talking about log-based/journaling file systems.

Course
TÖL401G: Stýrikerfi /
Operating Systems
7. Deadlocks

Chapter Objectives

- Illustrate how deadlock can occur.
- Define the four necessary conditions that characterize deadlock.
- Detect a deadlock situation in a resource allocation graph.
- Detect a deadlock situation using the matrix-based deadlock detection algorithm
- Evaluate the four different approaches for handling deadlocks.
- Apply the Safety algorithm to obtain safe schedules (if they exist).
- Apply the Banker's algorithm for deadlock avoidance.
- Evaluate approaches for recovering from deadlock.

Contents

1. Introduction
2. Deadlock Characterisation
3. Methods for Handling Deadlocks
4. Ignoring Deadlocks
5. Deadlock Detection & Recovery from Deadlocks
6. Deadlock Avoidance
7. Deadlock Prevention
8. Summary

Note for users of the Silberschatz et al. book: for didactical reasons, we have re-arranged the order of sections 7.4 to 7.7

7.1 Introduction: Classes of Resources

- **Sharable resource**: can be used by many at the same time.
- **Non-sharable resource**: can only be used by one process (or thread) at the same time. Can be further classified into:
 - **Preemptable resources**: Non-sharable resources used by a process that can be revoked from the process without having a permanently negative influence on the process.
 - E.g. processor: can be preempted by scheduler. Because registers are saved, context switch is no problem even though CPU is a non-shareable resource.
 - Main memory: processes can be swapped out from memory and swapped in again as long as contents is saved on/restored from hard disk.
 - **Non-preemptable resources**: Non-sharable resources used by a process where it will have a permanently negative influence if it would be revoked from the process.
 - E.g. CD burner: would end up with a burned CD where the first part of a song is from process A, the second part from process B.
 - Printer: would end up with a printed page where some characters are from process A, some characters are from process B.

Introduction: Usage Pattern for Resources

- Deadlocks always involve **non-preemptable** resources.
 - (Multiple processes waiting on each other's resource; if it is non-preemptable, the processes may deadlock. More on conditions necessary for a deadlock: later slides...)
- As any non-sharable resource may only be used by one process at the same time (mutual exclusion), processes need to request their usage before using it.

Three step sequence:

1. Request resource,

If resource is available:

2. Use resource for a finite amount of time,

3. Release resource!

Three step sequence looks trivial, but the underlying assumption is really important: **if we give a process the requested resources, the process will finally give all resources back.**

Introduction: Multiple Instances For Each Type Of Resource

- Process may depend on different types of resources at the same time.
 - E.g. a CD cloning process may require a CD reader and a CD burner at the same time.
- For each type of resource, there may be multiple instances available.
 - E.g. two CD reader devices – then, any of the two devices would satisfy the request of a process for a CD reader.
- More formally:
 - Resource types R_1, R_2, \dots, R_m
 - Each resource type R_i has W_i instances.
 - E.g.:
Two CD readers, one CD writer:
 $R_1 = \text{CD reader}, \quad R_2 = \text{CD writer},$
 $W_1 = 2, \quad W_2 = 1.$

Introduction: Managing Access to Resources

- Resource may be managed by the operating system: requesting, using & releasing is performed by system calls.
 - E.g. a file in the file system:
 - Before using it, its usage must be requested: “open” system call.
 - OS could block system call if file has already been locked by another process.
 - Then, it can be used: e.g. writing to it using a “write” system call
 - After usage, it must be released: “close” system call.
 - Resource may be managed by processes (threads): processes themselves are responsible for granting access to resource.
 - E.g. a shared data structure in shared memory:
 - Ensure mutual exclusion by guarding usage of data structure with, e.g., a semaphore (which lead to system calls that may block).

Introduction:

Example (1)

- Two types of resources (R_1, R_2 with $W_1=1, W_2=1$) used by two processes P_A, P_B .
- Processes manage mutual exclusive access to resources using a semaphore for each type of resource:

```
semaphore resourceOne = resourceOne.init(1);  
semaphore resourceTwo = resourceTwo.init(1);
```

```
void process_A() {  
    resourceOne.wait();  
    resourceTwo.wait();  
  
    use_both_Resources();  
  
    resourceTwo.signal();  
    resourceOne.signal();  
}
```

```
void process_B() {  
    resourceOne.wait();  
    resourceTwo.wait();  
  
    use_both_Resources();  
  
    resourceTwo.signal();  
    resourceOne.signal();  
}
```

Introduction:

Example (2)

- Code is free from deadlocks. (No proof given.)
- Two sample schedules (single processor/single core):

1. schedule:

```
Process_A: resourceOne.wait();  
           resourceTwo.wait();  
           use_both_Resources();  
           resourceTwo.signal();  
           resourceOne.signal();
```

```
Process_B: resourceOne.wait();  
           resourceTwo.wait();  
           use_both_Resources();  
           resourceTwo.signal();  
           resourceOne.signal();
```

2. schedule:

```
Process_A: resourceOne.wait();
```

Time slice expired: context switch

```
Process_B: resourceOne.wait();
```

Process B is put to sleep: context switch

```
           resourceTwo.wait();  
           use_both_Resources();  
           resourceTwo.signal();  
           resourceOne.signal();
```

Process B is waked up again: context switch

```
           resourceTwo.wait();  
           use_both_Resources();  
           resourceTwo.signal();  
           resourceOne.signal();
```

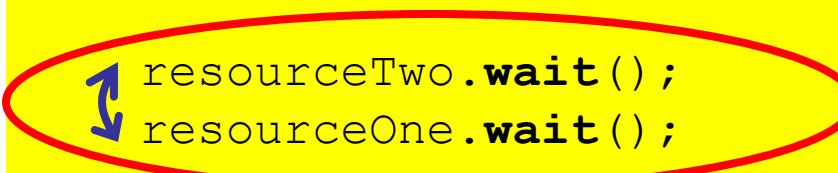
Introduction:

Example (3)

- Same example, but two lines swapped in resource request of process B:

```
semaphore resourceOne = resourceOne.init(1);  
semaphore resourceTwo = resourceTwo.init(1);
```

```
void process_A() {  
    resourceOne.wait();  
    resourceTwo.wait();  
  
    use_both_Resources();  
  
    resourceTwo.signal();  
    resourceOne.signal();  
}
```

```
void process_B() {  
    resourceTwo.wait();  
    resourceOne.wait();  
  
    use_both_Resources();  
  
    resourceOne.signal();  
    resourceTwo.signal();  
}
```

Introduction:

Example (4)

- Deadlocks may occur. Two sample schedules (single proc./single core):

1. Schedule (no deadlock):

```
Process_A: resourceOne.wait();  
           resourceTwo.wait();  
           use_both_Resources();  
           resourceTwo.signal();  
           resourceOne.signal();
```

```
Process_B: resourceTwo.wait();  
           resourceOne.wait();  
           use_both_Resources();  
           resourceOne.signal();  
           resourceTwo.signal();
```

2. schedule:

```
Process_A: resourceOne.wait();
```

Time slice expired: context switch

```
Process_B: resourceTwo.wait();  
           resourceOne.wait();
```

Process B is put to sleep: context switch

```
           resourceTwo.wait();
```

Process A is put to sleep

Each process waits for the other:

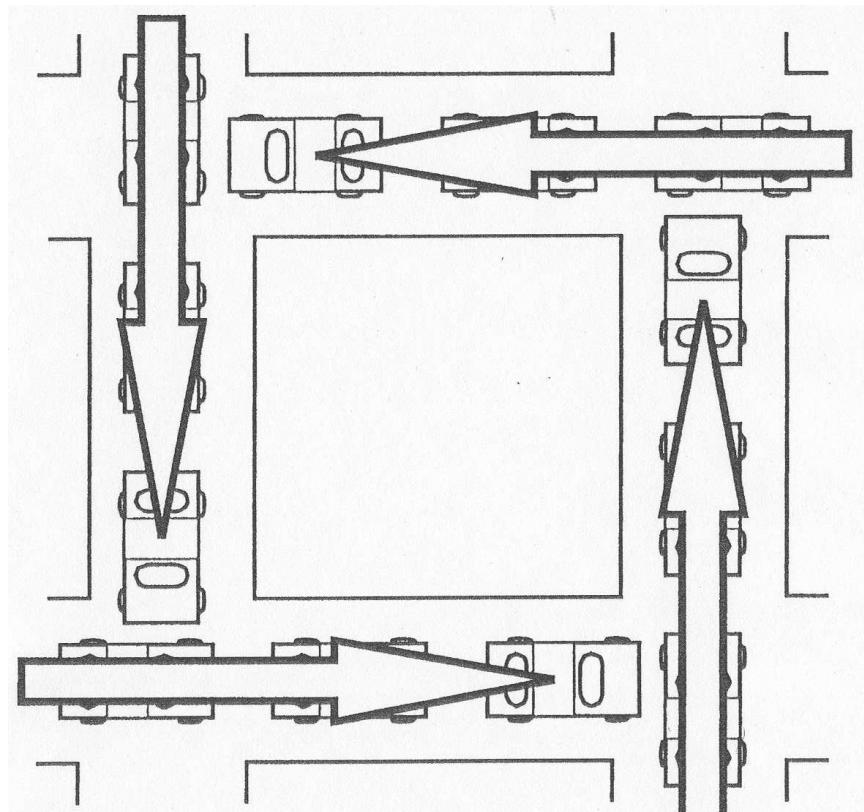
Process A waits for resourceTwo (however that will only be released by process B if it got resourceOne from process A).

Process B waits for resourceOne (however that will only be released by process A if it got resourceTwo from process B).

***** DEADLOCK *****

7.2 Deadlock Characterisation

- Law passed Kansas legislature early 20th century (according to Silberschatz et al.):
 - “When two trains approach each other at a crossing, both shall come to a full stop and neither shall start up again until the other has gone.”
- Informal definition of a deadlock:
 - „A set of processes is in a deadlock state, if each process from this set is waiting for an event that can only be triggered by another process from that set.“



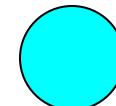
Necessary Conditions For A Deadlock

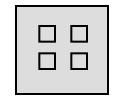
- According to Coffman et al. (1971), four conditions must hold simultaneously for a deadlock state:
 - Mutual exclusion:** only one process at a time can use a resource (non-sharable resource).
 - Hold and wait:** a process holding at least one resource is waiting to acquire additional resources held by other processes.
 - No preemption:** a resource can be released only voluntarily by the process holding it, after that process has completed its task.
 - Circular wait:** there exists a set $\{P_0, P_1, \dots, P_n\}$ of waiting processes such that P_0 is waiting for a resource that is held by P_1 , P_1 is waiting for a resource that is held by P_2 , ..., P_{n-1} is waiting for a resource that is held by P_n , and P_n is waiting for a resource that is held by P_0 .
- (Still general assumption applies: if we give a process all the requested resources, the process will finally give all resources back.)

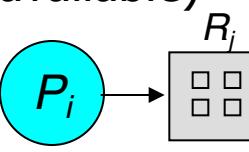
Resource-Allocation Graph

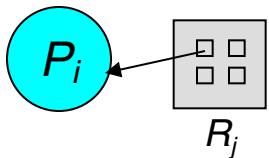
- For investigating deadlocks (and processes using resources) more formally, a **resource-allocation graph** can be used:

- Process node** represent a process:



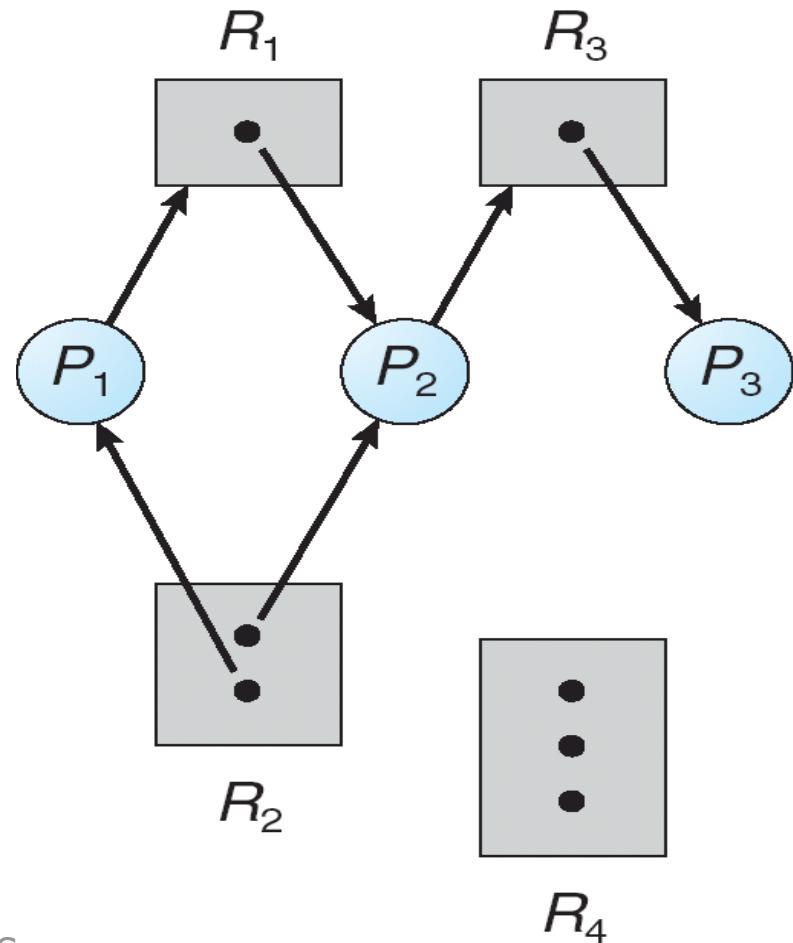
- Resource node** represents a resource type:  (4 instances of this type of resource are available)

- Request edge:**  (process P_i requests 1 resource of type R_j . If request can be fulfilled, edge becomes instantaneously an assignment edge.)

- Assignment edge:**  (1 instance of a resource type R_j is allocated to process P_i)

Resource-Allocation Graph: Example

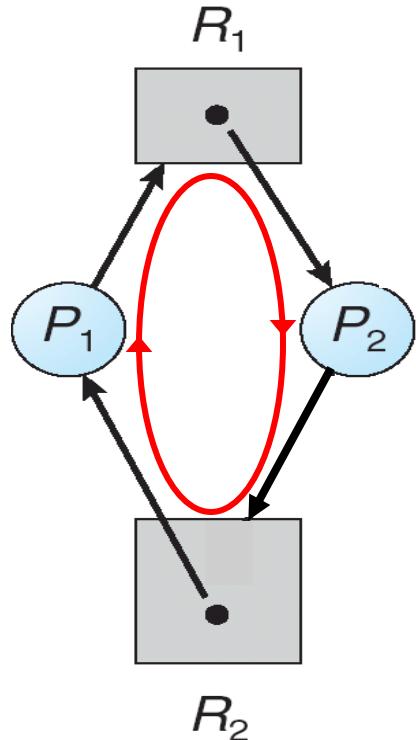
- Resources:
 - R_1 : 1 instance,
 - R_2 : 2 instances,
 - R_3 : 1 instance,
 - R_4 : 3 instances.
- Processes:
 - P_1 : waits for one instance of R_1 , one instance of R_2 is allocated to it.
 - P_2 : waits for one instance of R_3 , one instance of R_1 and one instance of R_2 is allocated to it.
 - P_3 : one instance of R_3 is allocated to it.



Resource-Allocation Graphs & Deadlocks

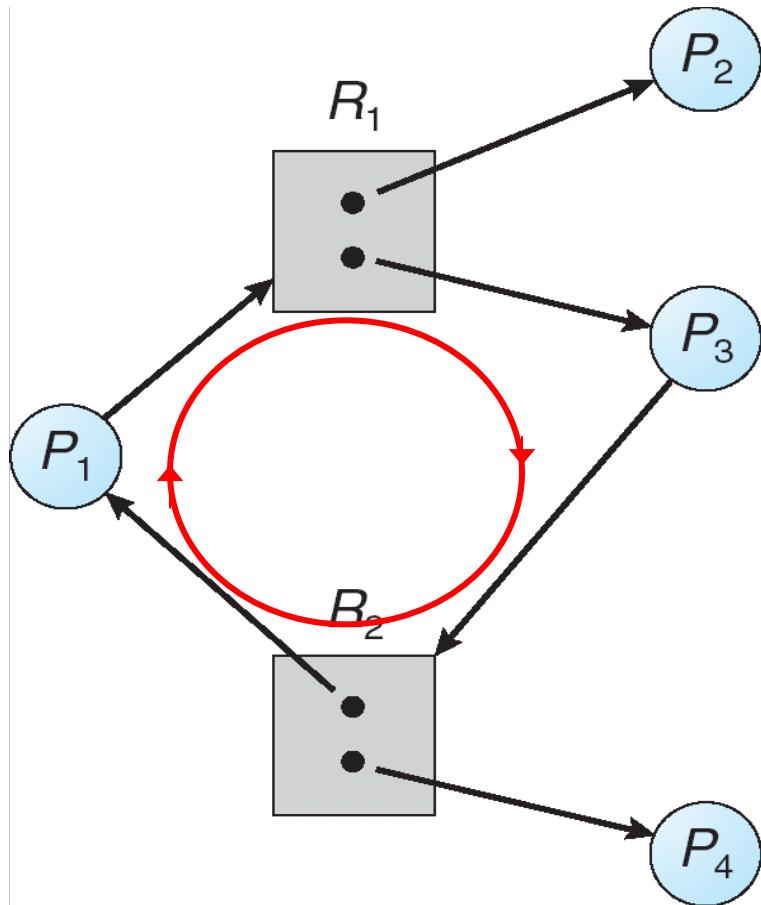
- Resource-allocation graph can be used to analyse Circular wait (& Hold and wait) condition:
 - If directed graph contains no cycles
⇒ no deadlock.
 - If directed graph contains a cycle ⇒
 - if only one instance per resource type, then deadlock.
 - (In this case, a cycle is both a necessary and sufficient condition.)
 - if several instances per resource type, possibility of deadlock.
 - (In this case, a cycle is a necessary, but not a sufficient condition: further investigation required to decide finally.)

Resource-Allocation Graphs & Deadlocks: Examples (1)



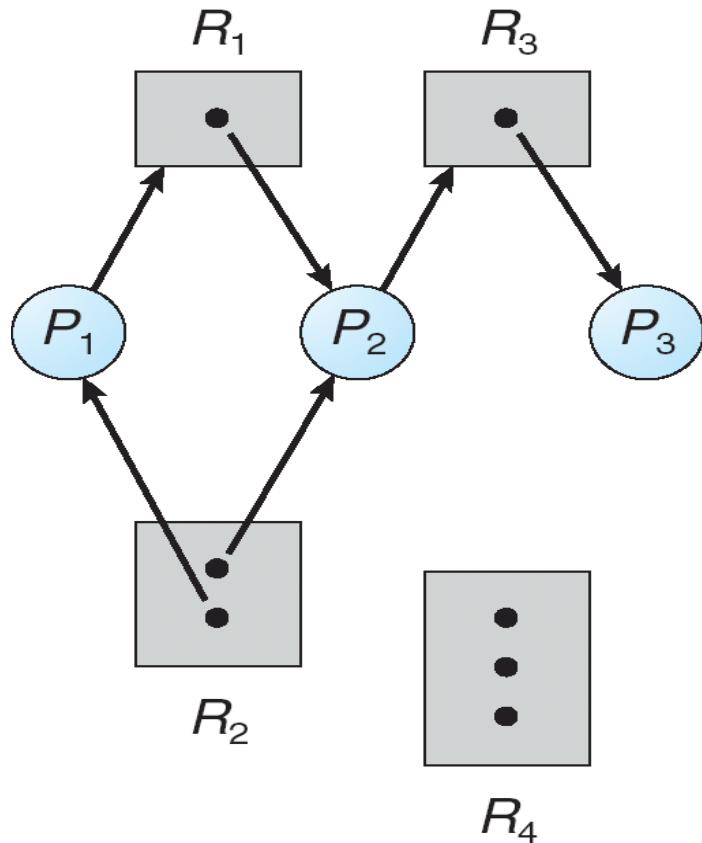
- Cycle & only one instance per resource type
⇒ deadlock!
 - (P_1 holds R_2 and waits for R_1 that is held by P_2 .
 P_2 holds R_1 and waits for R_2 that is held by P_1 . I.e. cycle $P_1 \rightarrow R_1 \rightarrow P_2 \rightarrow R_2 \rightarrow P_1$.)
- Note: all 4 deadlock conditions are fulfilled:
 - In particular: circular wait, hold and wait.
 - (mutual exclusion, no preemption are assumed as given anyway.)

Resource-Allocation Graphs & Deadlocks: Examples (2)



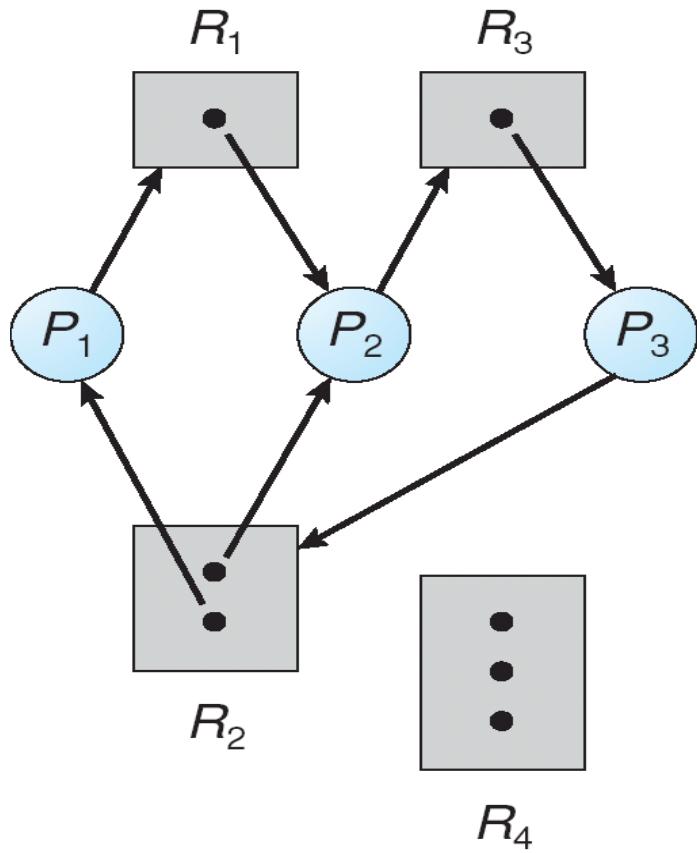
- Cycle & several instances per resource type \Rightarrow further analysis required:
- Even though we have a cycle $P_1 \rightarrow R_1 \rightarrow P_3 \rightarrow R_2 \rightarrow P_1$ we have no deadlock:
 - While P_1 and P_3 currently block each other, P_2 and P_4 are not blocked and thus still able to run. As they are able to run, they will eventually release their allocated resource, thus enabling also P_1 and P_3 to continue.

Resource-Allocation Graphs & Deadlocks: Examples (3)



- No cycle (note direction of edges!)
⇒ no deadlock!

Resource-Allocation Graphs & Deadlocks: Examples (4)



- Starting from example (3), P_3 requests R_2 .
- Cycle & several instances per resource type \Rightarrow further analysis required:
 - We have a small cycle $P_3 \rightarrow R_2 \rightarrow P_2 \rightarrow R_3 \rightarrow P_3$. For this, one might be tempted to argue that maybe P_1 eventually releases the instance of R_2 it holds, thus enabling P_3 to continue. But as we assume hold-and-wait, P_1 does rather not release the instance of R_2 it holds as it is also waiting for R_1 . So already this is a deadlock. Even if this would not be a deadlock, having a closer look reveals that there is an even further, larger circle into which also P_1 is involved:
 - Cycle $P_1 \rightarrow R_1 \rightarrow P_2 \rightarrow R_3 \rightarrow P_3 \rightarrow R_2 \rightarrow P_1$. In this cycle, all processes are waiting for a resource that is held by one of the other processes \Rightarrow deadlock of P_1 , P_2 , and P_3

7.3 Methods for Handling Deadlocks

- 4 different strategies to deal with deadlocks:
- **Ignore** the problem!
 - Not really a solution of the problem, but at least easy to implement.
(→section 7.4)
- **Detect** deadlocks **and recover** from them.
 - Allow system to enter deadlock state, detect this and recover
(→section 7.5).
- **Avoid** deadlocks by careful dynamic resource management.
 - Reject resource requests as soon as processes are endangered of creating a deadlock due to that resource request (→section 7.6).
- **Prevent** that deadlocks may occur at all.
 - Ensure that at least one of the four conditions necessary for a deadlock cannot hold (→section 7.7).
- Note: while the terms “avoidance” and “prevention” have a similar meaning, they refer to different solution **strategies**.

Handling deadlock: OS vs. applications

- All major operating system ignore deadlocks, i.e. deadlocks may occur concerning the resource managed by an OS (e.g. devices).
- While OSes are too generic resources to handle in a reasonable way concerning deadlocks, applications know their resources better.
- Example: **Database Management Systems** (DBMSes) have only one resource: data (e.g. the tables of an SQS database).
 - DBMS implementations use heavily **deadlock detection & recovery** to avoid deadlocks in concurrent atomic transactions.
 - Atomic transaction: either apply all changes or no changes, i.e. do not abort transaction with halfway applied changes.
 - Transactions may involve shared resources, i.e. same data accessed by different transactions in parallel: Deadlocks possible!
 - DBMS will detect deadlocks and then one (or more) transaction gets aborted (=none of their changes applied) while the others succeed (=all of their changes applied).

7.4 Ignoring Deadlocks

- Ignore the deadlock problem altogether.
 - Deadlocks may occur. If a deadlock occurs: so what?! – We have a multi-tasking operating system, thus we are still able to work with the remainder of the system that is not deadlocked. OK, maybe some resource (e.g. the printer) is blocked, but as long as no one wants to print, no one cares. At some point in time, hopefully someone restarts the system, thus resolving the deadlock.
- Advantage:
 - Easy to implement (nothing has to be implemented).
 - No management overhead for avoiding or detecting deadlocks.
- Disadvantage:
 - Processes are blocked in deadlock state: important data may not have been saved yet and thus get lost when process is killed or system is restarted.
- Reasonable approach
 - if deadlocks do not occur very often.
 - if costs of the other more advanced strategies are too high.
- All major operating system follow this approach: Unix, MS Windows (& Java).

Helmut Neukirchen: Operating Systems/updated

I.Hjörleifsson ingolfuh@hi.is st. TG225

7.5 Deadlock Detection & Recovery from Deadlock

- Allow system to enter deadlock state: processes may request resources as they like. This may lead to allocations of resources that result in a deadlock.
- In contrast to the “ignore deadlocks” approach, we want the operating system to detect that a deadlock occurred. If such a deadlock situation has been detected, the operating system should apply some recovery scheme to remove the deadlock (→later slides).
- Deadlock detection algorithms:
 - If all resource types have only a single instance: search for cycles in the resource allocation graph (slide 7-16). Complexity of graph algorithms for detecting cycles: $\mathcal{O}(n+m)^2$, where n = number of processes, m = number of different resource types, i.e. square of number of nodes in the graph.
 - In case of several instances per resource type: more advanced, matrix-based algorithm required (→next slides). Complexity: $\mathcal{O}(m \cdot n^2)$, where n is the number of processes, m the number of different resource types.

Matrix-based Deadlock Detection Algorithm for Several Instances of a Resource Type (1)

- Deadlock detection algorithm uses vectors and matrices as data structures:

Existing resources vector E

(describes how many instances of each resource exist
– not really needed by algorithm, just for the record):

$$(E_1, E_2, E_3, \dots, E_m)$$

Allocation matrix C

(describes how many instances of each resource are currently allocated ("hold") to each process):

$$\begin{bmatrix} C_{11} & C_{12} & C_{13} & \cdots & C_{1m} \\ C_{21} & C_{22} & C_{23} & \cdots & C_{2m} \\ \vdots & \vdots & \vdots & & \vdots \\ C_{n1} & C_{n2} & C_{n3} & \cdots & C_{nm} \end{bmatrix}$$

Row n describes current resource allocation of process n

Available resources vector A

(describes how many instances of each resource are currently available/free):

$$(A_1, A_2, A_3, \dots, A_m)$$

Request matrix R

(describes how many instances of each resource are currently requested ("wait") by each process):

$$\begin{bmatrix} R_{11} & R_{12} & R_{13} & \cdots & R_{1m} \\ R_{21} & R_{22} & R_{23} & \cdots & R_{2m} \\ \vdots & \vdots & \vdots & & \vdots \\ R_{n1} & R_{n2} & R_{n3} & \cdots & R_{nm} \end{bmatrix}$$

Row n describes which resources are currently requested by process n

Matrix-based Deadlock Detection Algorithm for Several Instances of a Resource Type (2)

- Algorithm:

- Initialise the available resources vector A and the allocation matrix C according to the current state of the system and the request matrix R according to the current requests of the processes. Be prepared to be able to mark later-on processes as finished: thus, unmark now all processes.
- Find an unmarked process P_i whose row i in the request matrix R is smaller (or equal) than the available resources vector A.
If no such process exists, go to step 4.
- Add row i of the allocation matrix C to the available resources vector A. Mark process P_i as finished.
Go to step 2.
- If there are unmarked processes: these processes are in a deadlock state.

If all processes are marked; system is deadlock free.

I.e. find a process for which it is possible to satisfy its current request.

I.e. pretend that we grant the request to that process and hence, finally, this process releases its resources making them available to the other processes.

Matrix-based Deadlock Detection Algorithm: Example (1)

- Example: Currently the system is in the state described by (E,) A, C (= "hold" part of "hold & wait") and R (= "wait" part of "hold & wait"):

Existing resources vector E

(describes how many instances of each resource exist
– not really needed by algorithm, just for the record):

$$E = (\begin{matrix} 4 & 2 & 3 & 1 \end{matrix})$$

Hard disk CD drive Printer Scanner

Available resources vector A

(describes how many instances of each resource are currently available/free):

$$A = (\begin{matrix} 2 & 1 & 0 & 0 \end{matrix})$$

Allocation matrix C

(describes how many instances of each resource are currently allocated to each process):

Processes:

$$\begin{array}{l} P1 \\ P2 \\ P3 \end{array} \quad C = \begin{pmatrix} 0 & 0 & 1 & 0 \\ 2 & 0 & 0 & 1 \\ 0 & 1 & 2 & 0 \end{pmatrix}$$

Request matrix R

(describes how many instances of each resource are currently requested by each process):

$$R = \begin{pmatrix} 2 & 0 & 0 & 1 \\ 1 & 0 & 1 & 0 \\ 2 & 1 & 0 & 0 \end{pmatrix}$$

- Now, we want to know: is this a deadlock state or not?
→ Apply matrix-based deadlock detection algorithm! (Use this state as step 1.)

Matrix-based Deadlock Detection Algorithm: Example (2)

- First iteration, step 2: Find an unmarked process P_i whose row i in the request matrix R is smaller/equal than the available resources vector A .

Existing resources vector E

(describes how many instances of each resource exist
– not really needed by algorithm, just for the record):

$$E = (\begin{matrix} 4 & 2 & 3 & 1 \end{matrix})$$

Hard disk CD drive Printer Scanner

Available resources vector A

(describes how many instances of each resource are currently available/free):

$$A = (\begin{matrix} 2 & 1 & 0 & 0 \end{matrix})$$

Allocation matrix C

(describes how many instances of each resource are currently allocated to each process):

Processes:

$$\begin{array}{l} P1 \\ P2 \\ P3 \end{array} \quad C = \begin{pmatrix} 0 & 0 & 1 & 0 \\ 2 & 0 & 0 & 1 \\ 0 & 1 & 2 & 0 \end{pmatrix}$$

Request matrix R

(describes how many instances of each resource are currently requested by each process):

$$R = \begin{pmatrix} 2 & 0 & 0 & 1 \\ 1 & 0 & 1 & 0 \\ 2 & 1 & 0 & 0 \end{pmatrix}$$

The last row (2, 1, 0, 0) is circled in red.

Matrix-based Deadlock Detection Algorithm: Example (3)

- First iteration, step 3: Add row i of the allocation matrix C to the available resources vector A . Mark process P_i as finished.

Existing resources vector E

(describes how many instances of each resource exist
– not really needed by algorithm, just for the record):

$$E = (\begin{matrix} 4 & 2 & 3 & 1 \\ \text{Hard disk} & \text{CD drive} & \text{Printer} & \text{Scanner} \end{matrix})$$

Available resources vector A

(describes how many instances of each resource are currently available/free):

$$\begin{array}{l} A = (\begin{matrix} 2 & 1 & 0 & 0 \\ 2 & 2 & 2 & 0 \end{matrix}) \\ + \Rightarrow (\begin{matrix} 2 & 2 & 2 & 0 \end{matrix}) \end{array}$$

Allocation matrix C

(describes how many instances of each resource are currently allocated to each process):

Processes:

P1

P2

P3✓

$$C = \left(\begin{matrix} 0 & 0 & 1 & 0 \\ 2 & 0 & 0 & 1 \\ 0 & 1 & 2 & 0 \end{matrix} \right)$$

Request matrix R

(describes how many instances of each resource are currently requested by each process):

$$R = \left(\begin{matrix} 2 & 0 & 0 & 1 \\ 1 & 0 & 1 & 0 \\ 2 & 1 & 0 & 0 \end{matrix} \right)$$

Matrix-based Deadlock Detection Algorithm: Example (4)

- Second iteration, step 2: Find an unmarked process P_i whose row i in the request matrix R is \leq than the available resources vector A .

Existing resources vector E

(describes how many instances of each resource exist
– not really needed by algorithm, just for the record):

$$E = (\begin{matrix} 4 & 2 & 3 & 1 \end{matrix})$$

Hard disk CD drive Printer Scanner

Available resources vector A

(describes how many instances of each resource are currently available/free):

$$A = (\begin{matrix} 2 & 2 & 2 & 0 \end{matrix})$$

Allocation matrix C

(describes how many instances of each resource are currently allocated to each process):

Processes:

$$C = \begin{pmatrix} 0 & 0 & 1 & 0 \\ 2 & 0 & 0 & 1 \\ 0 & 1 & 2 & 0 \end{pmatrix}$$

Request matrix R

(describes how many instances of each resource are currently requested by each process):

$$R = \begin{pmatrix} 2 & 0 & 0 & 1 \\ 1 & 0 & 1 & 0 \\ 2 & 1 & 0 & 0 \end{pmatrix}$$

The last two rows (P2 and P3) are circled in red.

Matrix-based Deadlock Detection Algorithm: Example (5)

- Second iteration, step 3: Add row i of the allocation matrix C to the available resources vector A . Mark process P_i as finished.

Existing resources vector E

(describes how many instances of each resource exist
– not really needed by algorithm, just for the record):

$$E = (\begin{matrix} 4 & 2 & 3 & 1 \end{matrix})$$

Hard disk CD drive Printer Scanner

Available resources vector A

(describes how many instances of each resource are currently available/free):

$$\begin{array}{l} A = (\begin{matrix} 2 & 2 & 2 & 0 \end{matrix}) \\ + \Rightarrow (\begin{matrix} 4 & 2 & 2 & 1 \end{matrix}) \end{array}$$

Allocation matrix C

(describes how many instances of each resource are currently allocated to each process):

Processes:

P1

P2✓

P3✓

$$C = \left(\begin{matrix} 0 & 0 & 1 & 0 \\ 2 & 0 & 0 & 1 \\ 0 & 1 & 2 & 0 \end{matrix} \right)$$

Request matrix R

(describes how many instances of each resource are currently requested by each process):

$$R = \left(\begin{matrix} 2 & 0 & 0 & 1 \\ 1 & 0 & 1 & 0 \\ 2 & 1 & 0 & 0 \end{matrix} \right)$$

Matrix-based Deadlock Detection Algorithm: Example (6)

- Third iteration, step 2: Find an unmarked process P_i whose row i in the request matrix R is \leq than the available resources vector A .

Existing resources vector E

(describes how many instances of each resource exist
– not really needed by algorithm, just for the record):

$$E = (\begin{matrix} 4 & 2 & 3 & 1 \\ \text{Hard disk} & \text{CD drive} & \text{Printer} & \text{Scanner} \end{matrix})$$

Available resources vector A

(describes how many instances of each resource are currently available/free):

$$A = (\begin{matrix} 4 & 2 & 2 & 1 \end{matrix})$$

Allocation matrix C

(describes how many instances of each resource are currently allocated to each process):

Processes:

P1

P2✓

P3✓

$$C = \begin{pmatrix} 0 & 0 & 1 & 0 \\ 2 & 0 & 0 & 1 \\ 0 & 1 & 2 & 0 \end{pmatrix}$$

Request matrix R

(describes how many instances of each resource are currently requested by each process):

$$R = \begin{pmatrix} 2 & 0 & 0 & 1 \\ 1 & 0 & 1 & 0 \\ 2 & 1 & 0 & 0 \end{pmatrix}$$

Matrix-based Deadlock Detection Algorithm: Example (7)

- Third iteration, step 3: Add row i of the allocation matrix C to the available resources vector A . Mark process P_i as finished.

Existing resources vector E

(describes how many instances of each resource exist
– not really needed by algorithm, just for the record):

$$E = (\begin{matrix} 4 & 2 & 3 & 1 \end{matrix})$$

Hard disk CD drive Printer Scanner

Available resources vector A

(describes how many instances of each resource are currently available/free):

$$\begin{array}{l} A = (\begin{matrix} 4 & 2 & 2 & 1 \end{matrix}) \\ + \Rightarrow (\begin{matrix} 4 & 2 & 3 & 1 \end{matrix}) \end{array}$$

Allocation matrix C

(describes how many instances of each resource are currently allocated to each process):

Processes:

P1 ✓
P2 ✓
P3 ✓

$$C = \left(\begin{matrix} 0 & 0 & 1 & 0 \\ 2 & 0 & 0 & 1 \\ 0 & 1 & 2 & 0 \end{matrix} \right)$$

Request matrix R

(describes how many instances of each resource are currently requested by each process):

$$R = \left(\begin{matrix} 2 & 0 & 0 & 1 \\ 1 & 0 & 1 & 0 \\ 2 & 1 & 0 & 0 \end{matrix} \right)$$

Matrix-based Deadlock Detection Algorithm: Example (8)

- Fourth iteration, step 2: Find an unmarked process P_i whose row i in the request matrix R is \leq than the available resources vector A . If no such process exists, go to step 4.

Existing resources vector E

(describes how many instances of each resource exist
– not really needed by algorithm, just for the record):

$$E = (\begin{matrix} 4 & 2 & 3 & 1 \end{matrix})$$

Hard disk CD drive Printer Scanner

Available resources vector A

(describes how many instances of each resource are currently available/free):

$$A = (\begin{matrix} 4 & 2 & 3 & 1 \end{matrix})$$

Allocation matrix C

(describes how many instances of each resource are currently allocated to each process):

Processes:

P1 ✓

P2 ✓

P3 ✓

$$C = \begin{pmatrix} 0 & 0 & 1 & 0 \\ 2 & 0 & 0 & 1 \\ 0 & 1 & 2 & 0 \end{pmatrix}$$

Request matrix R

(describes how many instances of each resource are currently requested by each process):

$$R = \begin{pmatrix} 2 & 0 & 0 & 1 \\ 1 & 0 & 1 & 0 \\ 2 & 1 & 0 & 0 \end{pmatrix}$$

- Fourth iteration, step 4: If there are unmarked processes: these processes are in a deadlock state. **If all processes are marked, system is deadlock free.** \Rightarrow no deadlock!

Matrix-based Deadlock Detection Algorithm: Example (9)

- Modified example:

Currently the system is in the state described by (E,) A, C and R:

Existing resources vector E

(describes how many instances of each resource exist
– not really needed by algorithm, just for the record):

$$E = (\begin{matrix} 4 & 2 & 3 & 1 \end{matrix})$$

Hard disk CD drive Printer Scanner

Available resources vector A

(describes how many instances of each resource are currently available/free):

$$A = (\begin{matrix} 2 & 0 & 0 & 0 \end{matrix})$$

Allocation matrix C

(describes how many instances of each resource are currently allocated to each process):

Processes:

$$\begin{array}{l} P1 \\ P2 \\ P3 \end{array} \quad C = \begin{pmatrix} 0 & 0 & 1 & 0 \\ 2 & 1 & 0 & 1 \\ 0 & 1 & 2 & 0 \end{pmatrix}$$

Request matrix R

(describes how many instances of each resource are currently requested by each process):

$$R = \begin{pmatrix} 2 & 0 & 0 & 1 \\ 1 & 0 & 1 & 0 \\ 2 & 1 & 0 & 0 \end{pmatrix}$$

- Now, step 2 (Find an unmarked process P_i whose row i in the request matrix R is \leq than the available resources vector A .) fails and we go to step 4. As no processes are marked, step 4 tells us that processes P1, P2, P3 are in a **deadlock state** (part of deadlock cycle)!

Detection Algorithm Usage

- Now, we have a nice algorithm to detect whether processes are in a deadlock state or not.
- When, and how often, to invoke a deadlock detection algorithm?
- Possibilities:
 - After each resource request that cannot be granted.
 - Disadvantage: computation overhead added to unsuccessful resource request operations.
 - Periodically.
 - Disadvantage: what is the appropriate period? Period too short: too much overhead. Period too long: deadlock remains undetected for some time during which the involved processes are blocked and even more processes might get involved into the deadlock.
 - When CPU utilisation is low.
 - Then, computation overhead does not harm. Furthermore, low CPU utilisation may be considered as a (necessary, but not sufficient) indicator of a deadlock; during a deadlock, processes are blocked (assuming no busy wait).

Recovery from Deadlock

- Once a deadlock detection algorithm has detected a deadlock, this situation has to be handled:
 - Either a human system operator is informed who has to deal with the deadlock **manually**.
 - Typically, the operator kills one of the deadlocked processes.
 - (This leads typically to a release of all resources held by that process, making these resources available for other processes that wait for them.)
 - Or **automatic recovery** is initiated by the system:
 - Either process termination (→next 2 slides)
 - or resource preemption. (→slide after the next 2 slides)

Process Termination (1)

- Terminate (& restart) at least one process that is part of the deadlock cycle.
 - “Brutal”, but simple approach.
 - When a process is terminated, the operating system releases typically all resources allocated to that process. Hence, other processes that were waiting for that resources may then use them.
 - However, this may leave resource in an inconsistent state.
⇒ Resource needs to be reset.
- Two alternative approaches of process termination:
 - Abort all deadlocked processes.
 - Definitely resolves the deadlock cycle. However, more processes than necessary may get killed.
 - Abort one process at a time until the deadlock cycle is eliminated.
 - As we have seen on slide 7-20, multiple deadlock cycles may exist at the same time. Hence, run deadlock detection algorithm after each killing of a process to decide whether this was sufficient to resolve the deadlock cycle.

Process Termination (2)

- When aborting one process at a time: how to select process to abort?
 - If deadlock detection algorithm runs frequently, it may be able to identify one single process that closed the deadlock cycle. This would be an obvious candidate.
 - Otherwise, we only know the set of deadlocked processes. Then selection could be based on many factors, e.g.
 - Priority of the process (scheduling priority):
 - abort process with lowest priority.
 - How long has process computed and how many resources has it used?
 - Process that has been used a lot of resources (incl. CPU time) is more likely to be finished soon, hence we should not abort that one.
 - Is process interactive or batch?
 - Aborting an interactive process is more annoying for the user than aborting a batch process that can be restarted without requiring user input.

Resource Preemption

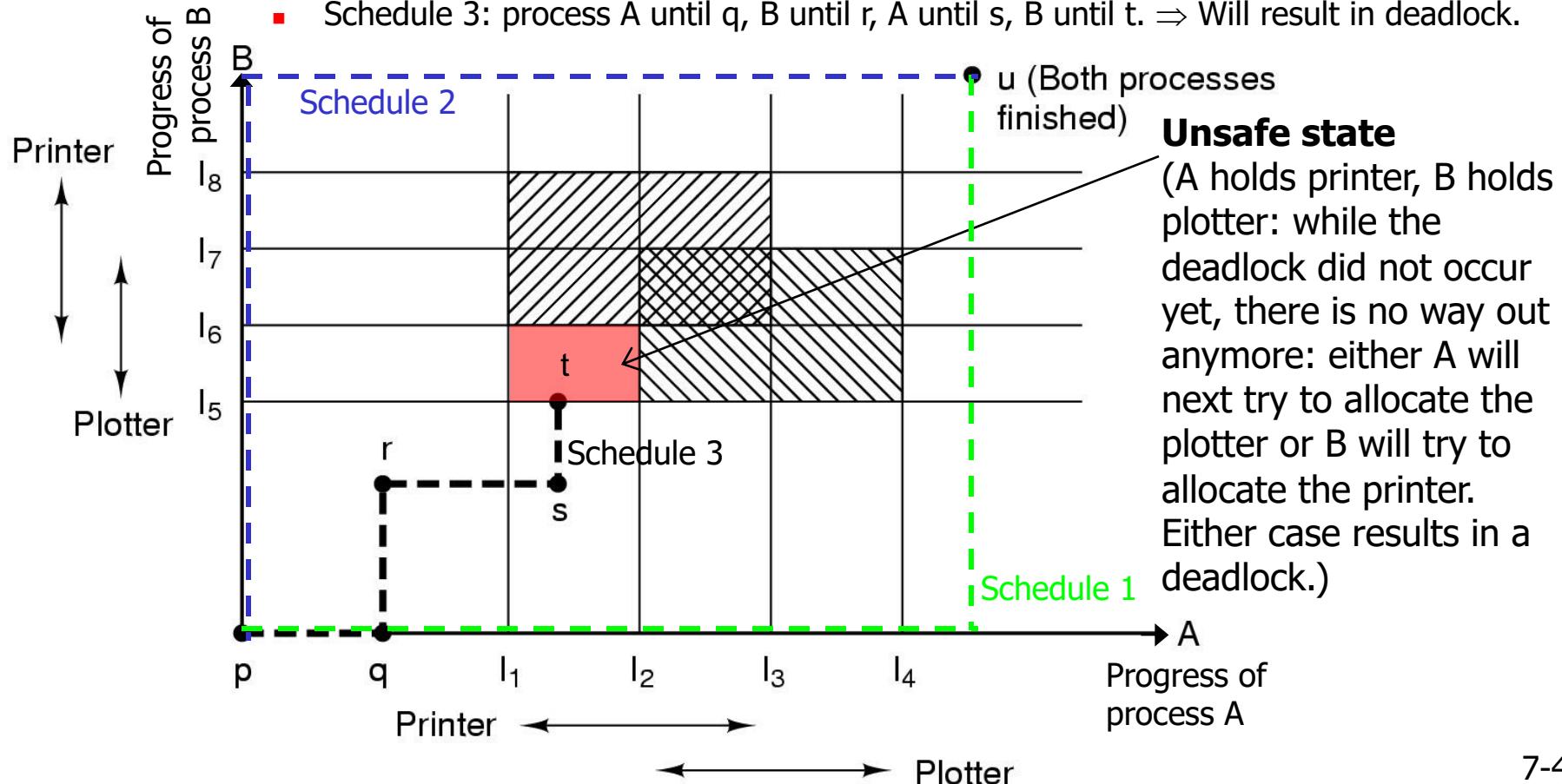
- Alternative to process termination: Preempt resource from one process and give it to a process that is requesting it.
 - Process and resource to be preempted need to be selected using some criteria (based on similar factors as on previous slide).
 - Resource preemption not easy to implement: process cannot simply continue execution after a resource has been preempted.
 - Solution: processes have to save periodically their current state ("checkpoint"). Then, a rollback can be performed back to a checkpoint where the preempted resource has not been requested yet.
 - Avoid starvation: it is likely that the preempted process tries to request the preempted resource again. Depending on the progression of the other processes this may result in a deadlock again. If the same factors for selecting a resource and process as last time are used, it is likely that the same resource gets preempted again from the same process
⇒ starvation.
 - Can be avoided by including number of preemptions into selection factors.

7.6 Deadlock Avoidance

- On slide 7-11, we have seen a schedule for two processes that leads to a deadlock and another schedule that leads to no deadlock for the same two processes.
 - If we were able to take only those schedules that lead to no deadlock, we would be able to avoid deadlocks.
- **Deadlock avoidance:** order processes and their resource requests (that potentially might lead to a deadlock) in a way that deadlocks do not occur (by using a clever schedule).
 - If processes announce their future resource requests in advance, the scheduler can take this into account and select processes to be executed based on this information to create schedules that avoid deadlocks.
 - I.e. as a result, a resource request of a process would not be granted, but the process would be blocked during resource request (**before** giving it the resource!) until another process that involves the same resources terminates.

Example (single core system): Trace of Three Different Schedules

- Process A will allocate printer at I_1 , allocate plotter at I_2 , release printer at I_3 , release plotter at I_4 .
- Process B will allocate plotter at I_5 , allocate printer at I_6 , release plotter at I_7 , release printer at I_8 .
 - Schedule 1: first process A, then process B. \Rightarrow No deadlock.
 - Schedule 2: first process B, then process A. \Rightarrow No deadlock.
 - Schedule 3: process A until q , B until r , A until s , B until t . \Rightarrow Will result in deadlock.



Safe States and Unsafe States

- Definition **safe state**:
 - A system is in a safe state if there exists an order of resource allocations (including release of resources once a process finishes) so that all requests of all processes can be satisfied.
 - Note: it is *not* necessary, that *all* different schedules fulfil this property. For a safe state, it is sufficient that at least *one* schedule exists that is able to fulfil all allocation requests.
- Definition **unsafe state**:
 - A system is in an unsafe state if there exists no order of resource allocations (including release of resources once a process finishes) so that all requests of all processes can be satisfied.
 - Note: In an unsafe state, a system has not deadlocked yet, however all possible schedules will unavoidably lead to a deadlock state.
(Because processes will not go backwards.)

Safe States and Unsafe States: Examples (1)

- To be able to select schedules so that unsafe states are avoided, the **maximum amount of requested resources** of the processes **needs to be known in advance** (for each resource type).
- **Example:** Process A will request a maximum of 9 instances of a resource, B a maximum of 4, C a maximum 7 instances of the resource.
 - Processes need to use system calls to specify in advance the maximum number of a requested resource, e.g. amount of needed memory:
 - A: max_memory(9); B: max_memory(4); C: max_memory(7);
 - Then, processes may use at any time system calls to allocate/release, e.g.:
 - A: mem_alloc(3), mem_alloc(6), mem_release();
 - B: mem_alloc(2), mem_alloc(2), mem_release();
 - C: mem_alloc(2), mem_alloc(5), mem_release();
 - OS may now decide for each resource request whether to grant it immediately or rather block the calling process on scheduler level (and run another process instead) until a request would be safe (because another process releases some resources).

Safe States and Unsafe States: Examples (2)

- Example: 10 instances of the same resource.
 - Process A will request a maximum of 9 instances of the resource, B a maximum of 4, C a maximum 7 instances of the resource.
 - Currently (step 1. shown below), 3 instances of the resource are free and A has 3, B has 2, C has 2 instances of the resource, i.e. in the worst case, A will request 6 further, B 2 further, C 5 further instances of the resource.
 - A possible schedule that allows to satisfy all future request is: grant request of B for 2 further instances (step 2a.) allowing B to finish and release its resources (step 2b.), then grant request of C for 5 further instances (step 3a.) allowing C to finish and release its resources (step 3b.), now A could request its 6 further instances of the resource. ⇒ States 1.) to 3b.) are safe states!

	Has	Max
A	3	9
B	2	4
C	2	7

Free: 3

	Has	Max
A	3	9
B	4	4
C	2	7

Free: 1

	Has	Max
A	3	9
B	0	-
C	2	7

Free: 5

	Has	Max
A	3	9
B	0	-
C	7	7

Free: 0

	Has	Max
A	3	9
B	0	-
C	0	-

Free: 7

Safe States and Unsafe States: Examples (3)

- Now, imagine the same initial situation (step 1.) as on the slide before, but where we would have chosen a schedule where we allowed process A to request first 1 further instance of the resource (step 2.).
- From step 2.) on, the best possible schedule would be to grant request of B for 2 further instances (step 3a.) allowing B to finish and release its resources (step 3b.) However, now we would not be anymore able to satisfy a request from A for 5 further instance nor from C for 5 further instances because only 4 instances are left.
⇒ While state 1.) is still a safe state, the inappropriate decision to grant process A one further instance of the resource leads to steps 2.) and beyond being unsafe states.
 - This would not have been happened, if we would have first scheduled process B and then process C as on the previous slide!

	Has	Max
A	3	9
B	2	4
C	2	7

Free: 3

	Has	Max
A	4	9
B	2	4
C	2	7

Free: 2

	Has	Max
A	4	9
B	4	4

Free: 0

	Has	Max
A	4	9
B	—	—

Free: 4

Safety Algorithm: Idea

- How to determine a schedule order that keeps processes in a safe state?
 - The example on slide 7-45 already demonstrates how to proceed:
 - If the currently available resources are sufficient to satisfy the maximum resource request of a process \Rightarrow give resources to that process and this process will eventually finish thus releasing even more resources that can then be used to satisfy further, even larger resource requests.

Safety Algorithm

- In fact, this idea is exactly the approach of the matrix-based deadlock detection algorithm from slides 7-25 to 7-35:
 - Just replace the Request matrix R by the Need matrix N as we do not consider current requests, but the maximum further needed instances of a resource:
 - Need matrix $N := \text{Maximum matrix } M - \text{Allocation matrix } C$,
 - where the Maximum matrix M describes how many instances of each resource type will at most be used by each process.
 - Then apply, matrix based deadlock detection algorithm.
 - Resulting order of process marking by that algorithm describes a schedule that is safe.
 - In unsafe state, if at least one process remains unmarked.

Safety Algorithm: Example (based on slide 7-27)

- Example: Currently the system is in the state described below, e.g.:
P1 holds 1 printer and may in future need up to 2 hard disks and 1 scanner.

Existing resources vector E

(describes how many instances of each resource exist
– not really needed by algorithm, just for the record):

$$E = (\begin{matrix} 4 & 2 & 3 & 1 \end{matrix})$$

Hard disk CD drive Printer Scanner

Available resources vector A

(describes how many instances of each resource are currently available/free):

$$A = (\begin{matrix} 2 & 1 & 0 & 0 \end{matrix})$$

Allocation matrix C

(describes how many instances of each resource are currently allocated to each process):

Need matrix N

(describes how many instances of each resource each process may need in addition):

Processes:

$$\begin{array}{l} P1 \\ P2 \\ P3 \end{array} \quad C = \begin{pmatrix} 0 & 0 & 1 & 0 \\ 2 & 0 & 0 & 1 \\ 0 & 1 & 2 & 0 \end{pmatrix}$$

$$N = \begin{pmatrix} 2 & 0 & 0 & 1 \\ 1 & 0 & 1 & 0 \\ 2 & 1 & 0 & 0 \end{pmatrix}$$

- Now, we want to know: is a safe schedule possible to satisfy all future requests of P1-P3 and, if yes, how does this safe schedule look like? \Rightarrow Apply algorithm from 7-26, however request matrix R is now called need matrix N. **Safe schedule exists, if all processes are marked: schedule is the order of marking** (for above example, safe schedule is P3,P2,P1).

Banker's Algorithm: Introduction

- The matrix-based safety algorithm yields only one possible order of a safe schedule of resource requests.
- However, in practise, processes may make resource requests in a different order and in different chunks.
- How to decide whether to grant a current request or whether to block a request until some other process has released further resources?
⇒ **Banker's algorithm!** (Dijkstra, 1965)
 - (Name comes from the idea that a banker should only grant a request for a loan if enough money remains to satisfy further future requests of that client who might finally be able to pay back only if all loans are granted.)

Banker's Algorithm

- Having the matrix-based safety algorithm, the banker's algorithm is simple:
 - 1. Pretend that the currently pending request has been granted, i.e. update
 - Available resources vector A,
 - Allocation matrix C, and
 - Need matrix N accordingly.
 - 2. Apply the matrix-based safety algorithm to decide if the resulting state is safe.
 - If yes: grant request (and keep A, C, N as modified in step 1 above.).
 - If no: reject request (i.e. put process to sleep without assigning resource to it even though resource would be available) and restore matrices A, C, N.
 - Once resources are released by another process: re-run banker's algorithm to see if it would now be safe to grant the previously rejected request and wake up process that was put to sleep.

Banker's Algorithm: Example

- Example: Currently the system is in the state described below, e.g.: P1 holds currently no resources, but P2 and P3 do.

Existing resources vector E

(describes how many instances of each resource exist – not really needed by algorithm, just for the record):

$$E = (\begin{matrix} 4 & 2 & 3 & 1 \end{matrix})$$

Hard disk CD drive Printer Scanner

Available resources vector A

(describes how many instances of each resource are currently available/free):

$$A = (\begin{matrix} 2 & 1 & 1 & 0 \end{matrix})$$

Allocation matrix C

(describes how many instances of each resource are currently allocated to each process):

Need matrix N

(describes how many instances of each resource each process may need in addition):

Processes:

$$\begin{array}{l} P1 \\ P2 \\ P3 \end{array} \quad C = \begin{pmatrix} 0 & 0 & 0 & 0 \\ 2 & 0 & 0 & 1 \\ 0 & 1 & 2 & 0 \end{pmatrix}$$

$$N = \begin{pmatrix} 2 & 0 & 1 & 1 \\ 1 & 0 & 1 & 0 \\ 2 & 1 & 0 & 0 \end{pmatrix}$$

- Now, let's assume P1 requests 1 printer (and may in future need in addition 2 hard disks and 1 scanner): The OS now, needs to decide whether to grant this request or not.
- ⇒ Apply steps of Banker's algorithm →next slide.

Banker's Algorithm: Example (based on slide 7-27)

- Step 1: Pretend that the currently pending request (=1 printer) has been granted, i.e. update Available resources vector A, Allocation matrix C, and Need matrix N accordingly.

Existing resources vector E

(describes how many instances of each resource exist – not really needed by algorithm, just for the record):

$$E = (4 \text{ Hard disk}, 2 \text{ CD drive}, 3 \text{ Printer}, 1 \text{ Scanner})$$

Available resources vector A

(describes how many instances of each resource are currently available/free):

$$A = (2 \quad 1 \quad 0 \quad 0)$$

Allocation matrix C

(describes how many instances of each resource are currently allocated to each process):

Processes:

$$C = \begin{pmatrix} 0 & 0 & 1 & 0 \\ 2 & 0 & 0 & 1 \\ 0 & 1 & 2 & 0 \end{pmatrix}$$

Need matrix N

(describes how many instances of each resource each process may need in addition):

$$N = \begin{pmatrix} 2 & 0 & 0 & 1 \\ 1 & 0 & 1 & 0 \\ 2 & 1 & 0 & 0 \end{pmatrix}$$

- Step 2: Apply the matrix-based safety algorithm (from 7-48) to decide if the resulting state is safe. If yes: grant request (and keep A, C, N as modified in step 1.). If no: reject request (i.e. put process to sleep without assigning resource to it even though resource would be available) and restore matrices A, C, N. (for above example: grant request!)

7.7 Deadlock Prevention

- By ensuring that one of the four conditions necessary for deadlocks (slide 7-13) does not hold, deadlocks can be prevented. Reminder of four conditions:
 - **Mutual exclusion:** only one process at a time can use a resource (non-sharable resource).
 - **Hold and wait:** a process holding at least one resource is waiting to acquire additional resources held by other processes.
 - **No preemption:** a resource can be released only voluntarily by the process holding it, after that process has completed its task.
 - **Circular wait:** there exists a set $\{P_0, P_1, \dots, P_n\}$ of waiting processes such that P_0 is waiting for a resource that is held by P_1 , P_1 is waiting for a resource that is held by P_2, \dots, P_{n-1} is waiting for a resource that is held by P_n , and P_n is waiting for a resource that is held by P_0 .

Deadlock Prevention: Mutual Exclusion

- By definition, non-shareable resources must be accessed in a mutual exclusive way.
 - However, take care to **consider a resource only as non-sharable where necessary**: e.g. a read-only file can be shared.
 - **Spooling** may be used for some output devices (e.g. printers).
 - Instead of having to wait for a mutual exclusive access to a printer, send print job to a spooler system process that buffers print jobs.
 - Only spooler process accesses printer in an exclusive way (spooler process can be accessed in a shared manner) \Rightarrow deadlock prevented!
 - Spooling is not applicable for all resources. However, the underlying idea is good:
 - Access resource directly (=mutual exclusive) only where necessary.
As often as possible, use some abstraction layer that is, e.g., able to re-order accesses.

Deadlock Prevention: Hold and Wait

- Prevent that processes keep resources while they are waiting for further resources.
- Possible approach:
 - Processes are required to request all needed resources at the beginning of their execution in one atomic step and are not allowed to request (in a hold & wait style) resources in-between.
 - Problem: in particular programs that are supposed to adapt dynamically to growing needs do not know in advance how many resources they need.
 - Solution: Processes are allowed to request resources also in-between, however, they are required to release all of their resources before requesting the new resources (again, the following request of the old and the new resources must be atomic).
 - Works nice as long as it is reasonable to release a resource during usage!

Deadlock Prevention: No Preemption

- In section 7.5 (Deadlock Detection and Recovery), we have seen that **preemption of resources is difficult** (in practise only possible if applications are designed to use checkpointing and rollback) and should therefore only be done as a last resort when a deadlock has already occurred.
 - Resource preemption as a method for deadlock prevention means to be able preempt resources at each resource request (if necessary).
 - “Polite” approach: do not preempt resources from processes that are busy using these resource, but only preempt resources from processes that are waiting anyway. E.g.:
 - Whenever a process has anyway to wait for some other resource, the resources of this waiting process are preempted.

Deadlock Prevention: Circular Wait

- Circular wait can be prevented if all processes have to request resources in the same order:
 - Impose a total ordering on all resource types and require that each process requests resources only in an increasing order of enumeration.
 - Example: 1. Tape drive, 2. disk drive, 3. printer.
 - Processes that want to allocate a tape drive and a printer, always have to allocate them in the order: first tape drive, then printer.
 - While processes may still have to wait for a resource, **they will never wait it in a circular way!**
 - Problem: while it may be easy to agree on an ordering for all standard resources in a system, each process might use a different ordering for “exotic” custom resources that were not known when the ordering of the standard resources has been defined.

7.8 Summary

- Deadlocks arise whenever non-shareable (**mutual exclusive**) and **non-preemptable** resources are used in a **hold-and-wait** manner and processes are **waiting circular** for resources held by other processes.
 - Assumption: processes do not crash, give resources back after using them.
- Methods to deal with deadlocks:
 - **Ignore**: do not waste efforts with advanced methods.
 - **Detect & recover**: allow deadlocks to occur, try to recover from them once they have been detected (e.g. using **matrix-based detection algorithm**).
 - **Avoid**: only grant resource requests in a way that only safe orderings of requests occur (**Banker's algorithm**), i.e. put processes to sleep if they are about to make unsafe requests. May wake them later up again when resources are released.
 - **Prevent**: break one of the four deadlock conditions.

Course
TÖL401G: Stýrikerfi /
Operating Systems

8. Memory Management Strategies

Chapter Objectives

- Explain the difference between a logical and a physical address and the role of the memory management unit (MMU) in translating addresses.
- Explain memory organisation and address binding when loading processes and how they relate to different types of MMUs.
- Explain Swapping.
- Apply first-, next-, best-, and worst-fit strategies for allocating memory contiguously.
- Explain the distinction between internal and external fragmentation.
- Understand Paged MMUs (PMMUs) with page tables and translation lookaside buffers for speeding up lookups.
- Translate logical to physical addresses in a paging system using PMMU.
- Describe hierarchical paging/multi-level paging.
- Describe applications of paging: memory protection, circumvent external fragmentation, shared memory.

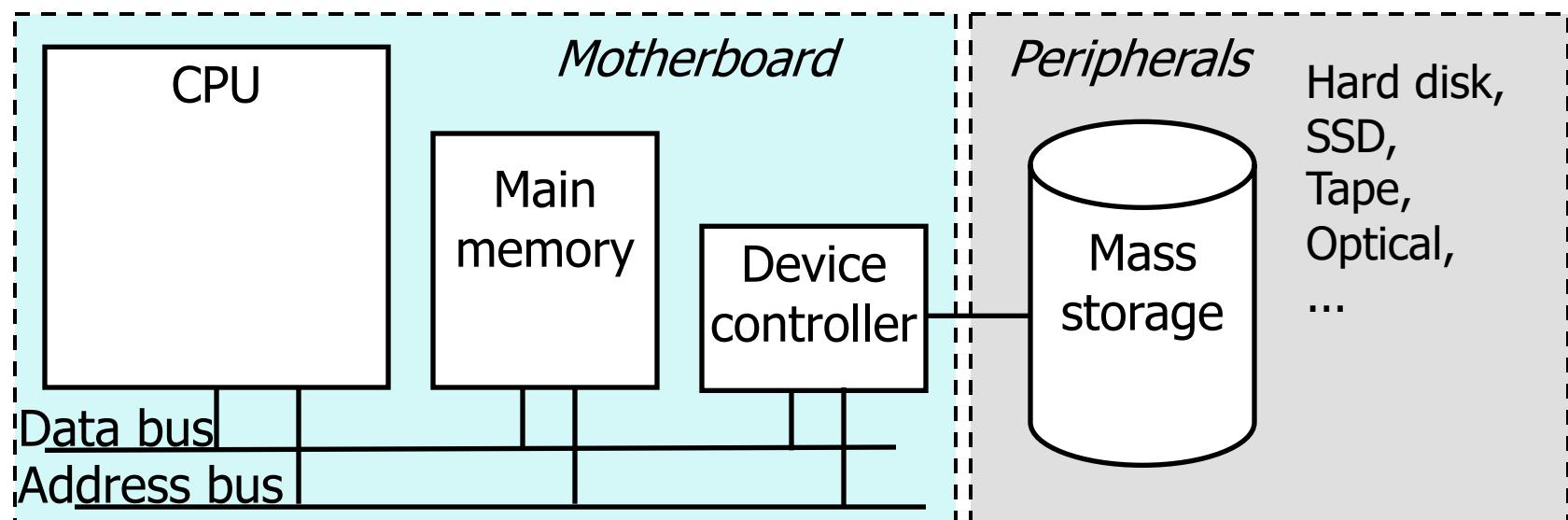
Contents

1. Introduction
2. Swapping
3. Contiguous Memory Allocation
4. Paging
5. Summary

Note for users of the Silberschatz et al. book: in the beginning, my slides are more extensive than in the book (whereas towards the end, I omit some topics in comparison to the book).

8.1 Introduction

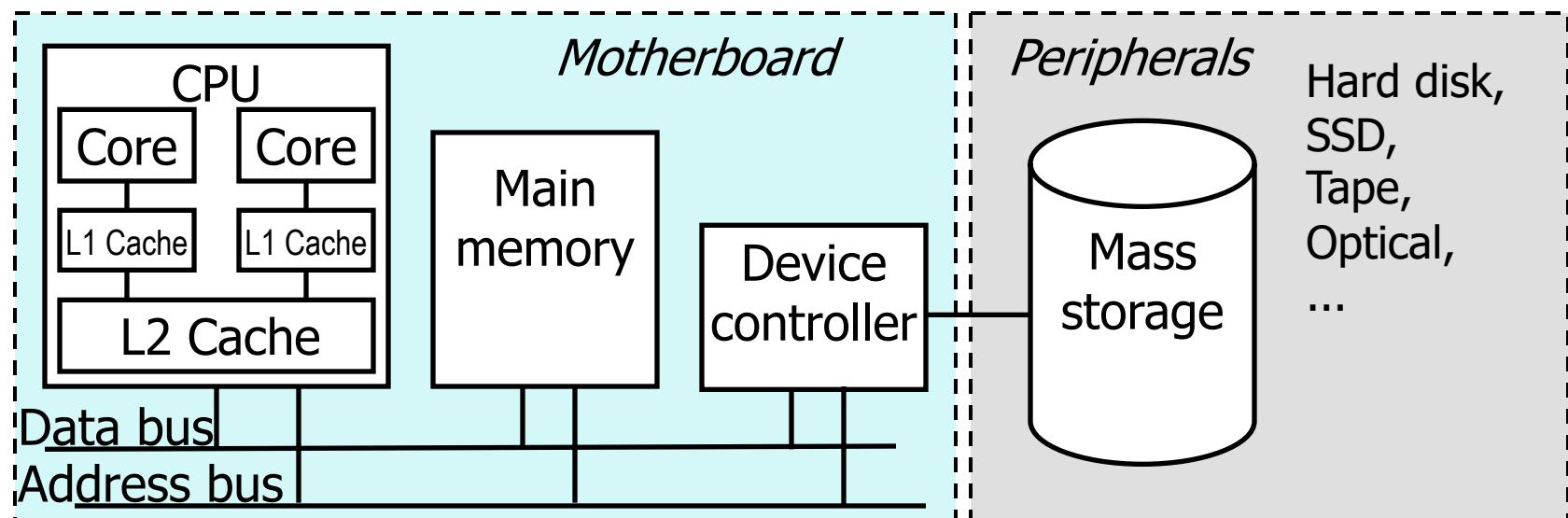
- Different types of memory in a multi-core CPU computer:



- CPU can only access locations (& execute instructions) in main memory (& read, store, and process data in CPU registers).
 - Mass storage only indirectly accessible via device controller.
 - Program instructions must be loaded from mass storage to main memory.

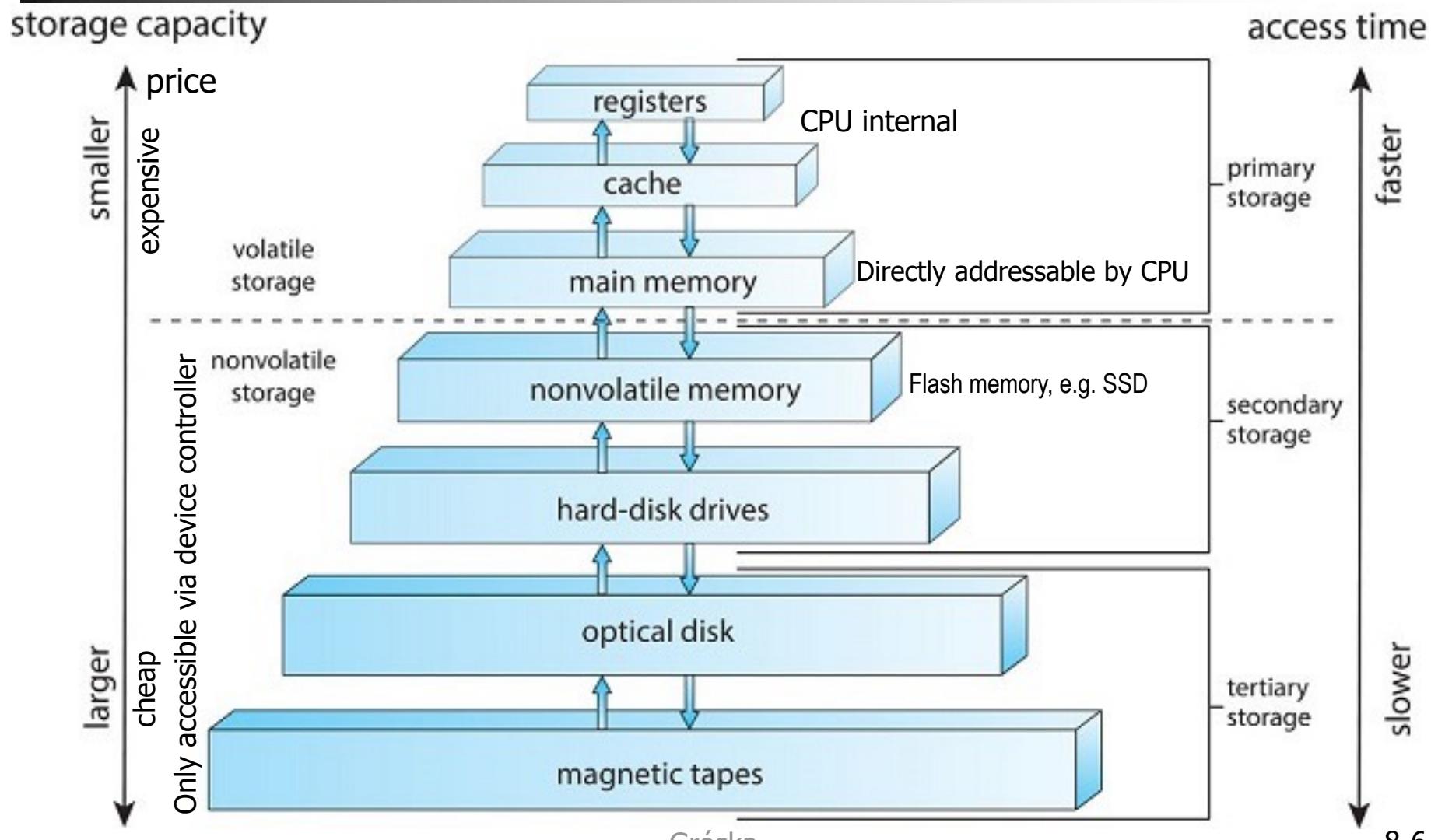
CPU Caches

- Access to main memory is slow (in comparison to CPU registers).



- Cache stores values that have been read or written before.
 - Fast cache RAM (reduces memory stalls), but limited in size.
- Multi-level caching (note: sometimes even Level 3 caches used):
 - Level 1 (L1) cache uses faster (& more expensive) RAM: small size.
 - Level 2 (L2) cache uses slower (& less expensive) RAM: larger size.

Reminder Chapter 1: Storage-Device Hierarchy



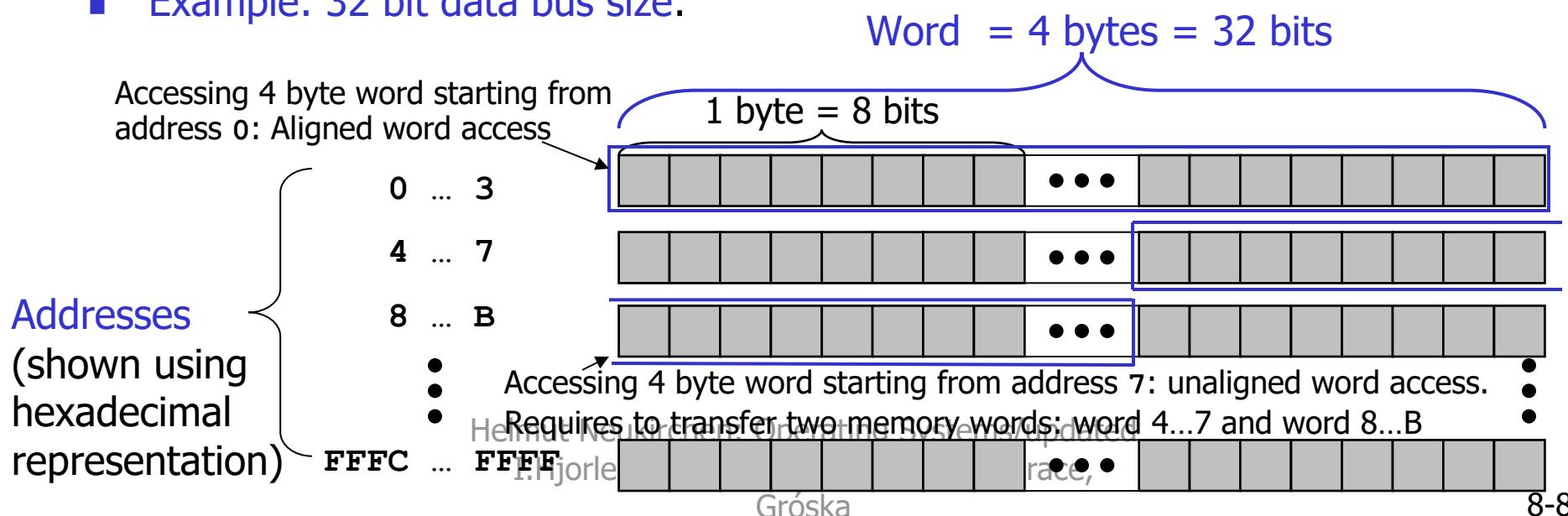
Reminder Chapter 1: Performance of Various Levels of Storage

Level	1	2	3	4	5
Name	registers	cache	main memory	solid state disk	magnetic disk
Typical size	< 1 KB	< 16MB	< 64GB	< 1 TB	< 10 TB
Implementation technology	custom memory with multiple ports CMOS	on-chip or off-chip CMOS SRAM	CMOS DRAM	flash memory	magnetic disk
Access time (ns)	0.25 - 0.5	0.5 - 25	80 - 250	25,000 - 50,000	5,000,000
Bandwidth (MB/sec)	20,000 - 100,000	5,000 - 10,000	1,000 - 5,000	500	20 - 150
Managed by	compiler	hardware	operating system	operating system	operating system
Backed by	cache	main memory	disk	disk	disk or tape

- In the past, cost was the main driver that prevented to use larger main memory.
- Then, the maximum memory addressable by a 32 Bit CPU became a restriction:
 - 32 Bit CPU uses 32 Bit to address memory: maximum memory addressable with 32 Bit: 4GB
- Today's 64 Bit CPUs allows to address larger memory.

Addressing Main Memory: Data

- Data bus size (e.g. 32 bit “words” or 64 bit “words”) determines how many bits are transferred during a memory access cycle.
 - RAM address to be accessed is put on the address bus by the CPU,
 - Then, data bus is used to transfer words between RAM and CPU.
 - Even if CPU wants to process only single bits or bytes, always the whole word is transferred.
 - If a word shall be accessed that is not aligned (i.e. crosses boundary of two physical words in memory), two memory access cycles are required. (\Rightarrow Aligned access faster.)
 - Example: 32 bit data bus size:



Addressing Main Memory: Addresses

- Address bus size and CPU address register size restrict memory size.
 - Size (in bits) of CPU address registers restricts maximum number of bytes addressable by programs (logical addresses).
 - Machine code uses CPU address register to address and access main memory.
 - (Program counter (PC) is a special address register to address the instruction in main memory to be executed.)
 - Size of address bus, i.e. number of CPU pins and number of address bus tracks found on motherboard, restricts size of accessible physical memory (physical addresses).
 - It makes no sense to plug-in more RAM in your computer than the address bus can address.
 - As memory is always accessed word-wise using word-aligned memory addresses, the address bus lacks the lowest significant bits.
 - E.g. with 64 bit words=8 Byte words, the lowest 3 Bits lacking on the address bus. (e.g. bytes 0..7 are read as one word over the databus).
- ⇒It is possible to logically address more locations than physically accessible: in chapter 9 on virtual memory, we will learn more.

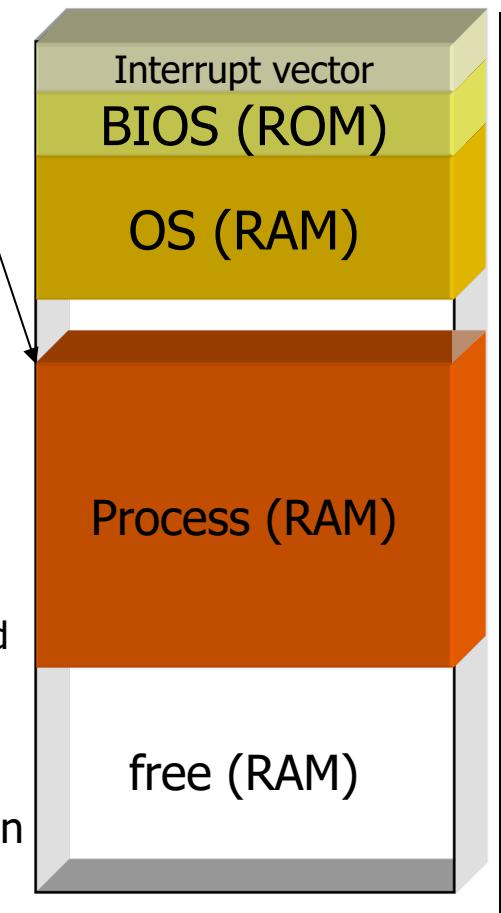
Addressing Main Memory: Addresses: 32 bit / 64 bit examples

- Example: 32 bit system:
 - 32 bit CPU address registers: allow to address 4GB.
 - 32 bit data bus: allows to plug-in 4GB of RAM.
- Example: 64 bit system:
 - 64 bit CPU address registers: allow to address 16 exabytes (=16.8 million TB).
 - As today, no-one needs 16 exabytes of RAM modules in a single computer, today's 64 bit CPUs only have – depending on CPU– a 36, 40 or 44 bit address bus : allows to access at least 64 GB of RAM (if motherboard allows to plug in as much).

Organisation of Memory: Simple Monoprogramming

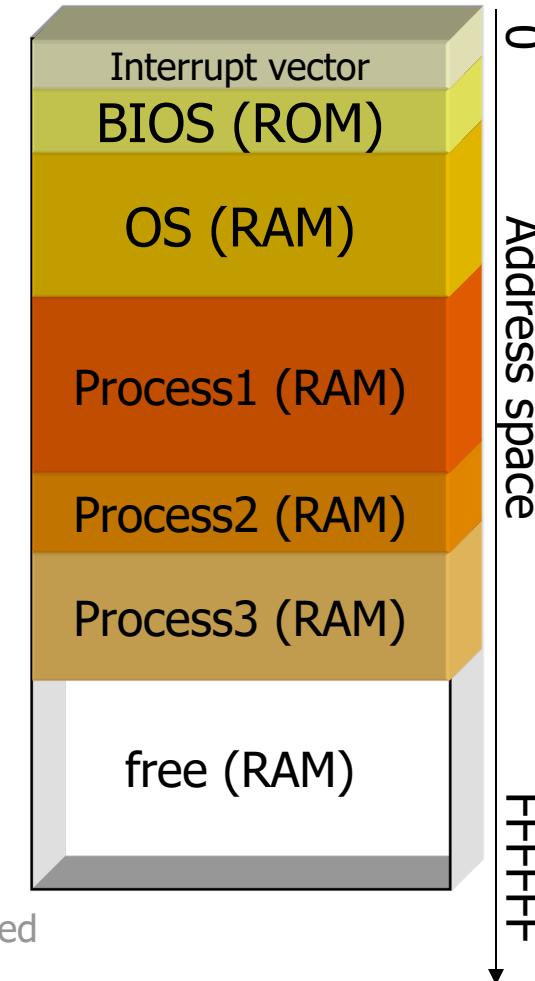
- Monoprogramming (e.g. MS-DOS):
 - BIOS in ROM, OS in RAM,
 - OS loads & executes only **one process** at a time.
 - ⇒ (Apart from BIOS & OS) there are only the instructions of the current process found in the address space:
 - Program code is created for running at a fixed start address in memory (**absolute code**).
 - E.g.: All programs are loaded at the start address 1000. If a program wants to access some data stored at the 8th byte of the program, the programmer can rely on that this byte will be at address 1008. (I.e. “binding” of address occurs at **compile time**.)
 - No **memory protection**: the whole address space can be accessed by the current process: **process might overwrite OS and thus crash the system**.

Same fixed start address for each program



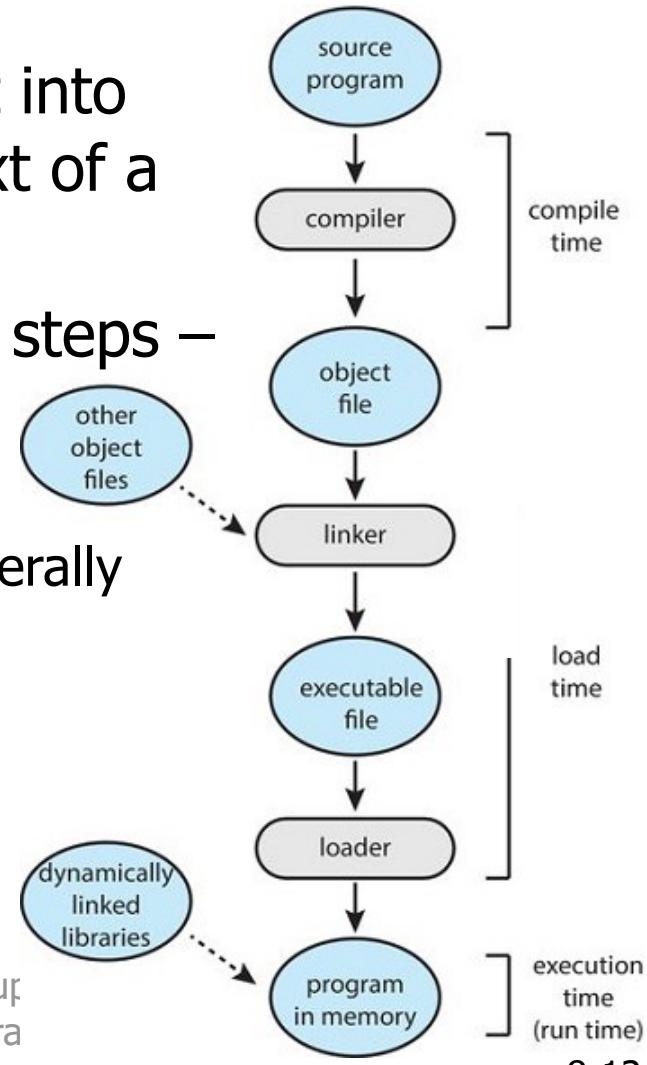
Organisation of Memory: Simple Multiprogramming

- Simple multiprogramming, i.e. still shared address space (e.g. Windows 3.11, 95, 98, ME):
 - BIOS in ROM, OS in RAM,
 - **Multiple processes** sharing the available RAM.
 - Start address is different for each process.
 - A program cannot rely anymore at which absolute address it is executing. (It rather depends on which start address it has been loaded to by the OS.)
 - Program code must be **relocatable**: use only machine code instructions based on relative addressing (if available) or adjust all absolute addresses during loading (**load-time binding**). (More on relocation on next slides.)
 - **No memory protection**: the whole address space can be accessed by the current process: **process might overwrite OS and other processes and thus crash these**.



Excursion: Address Binding

- To run, the program must be brought into memory and placed within the context of a process.
- A user program goes through several steps – some of which may be optional – before being executed.
 - Addresses in the source program are generally **symbolic** (such as a variable **count**).
 - A compiler typically binds these symbolic addresses to **relocatable addresses** (such as “14 bytes from the beginning”).
 - The loader in turn binds the relocatable addresses to **absolute addresses** (such as 74014).



Excursion: Relocatable Code (Load-time Binding)

- Program code is created by compiler/assembler for start address 0 and a relocation table is added to the program file.
- Relocation table records each absolute address found in the program file:

Generated code	Absolute Addresses	Assembly language instructions	
00000000		1 ORG	\$0
00000000		2	
00000000	4279 00000018	3 START	CLR.W
00000006	3039 0000001A	4	MOVE.W
0000000C	D179 00000018	5 LOOP	ADD.W
00000012	5340	6	SUB.W
00000014	66F6	7	BNE
00000016	4848	8	BREAK
00000018		9	
00000018		10 SUM	DS.W
0000001A	0019	11 COUNT	DC.W
0000001C	0000000C	12 LOOP_ADDR	DC.L

Relative address

Generate Code for addr. 0

Clear variable SUM

Load COUNT value

ADD D0 to SUM

Decrement counter

Loop if counter not zero

Tell the simulator to BREAK

Reserve one data word for SUM

Initial value for COUNT data value

Data value containing absolute pointer

- ⇒ Relocation table contains address entries: 2, 8, E, 1C
- When loading a program, the OS program loader chooses a free memory location, loads the program code into that memory area, and adds the starting address of the chosen memory location to all locations containing **absolute addresses listed in the relocation table**.

Excursion: Static Linking (Compile-time Binding)

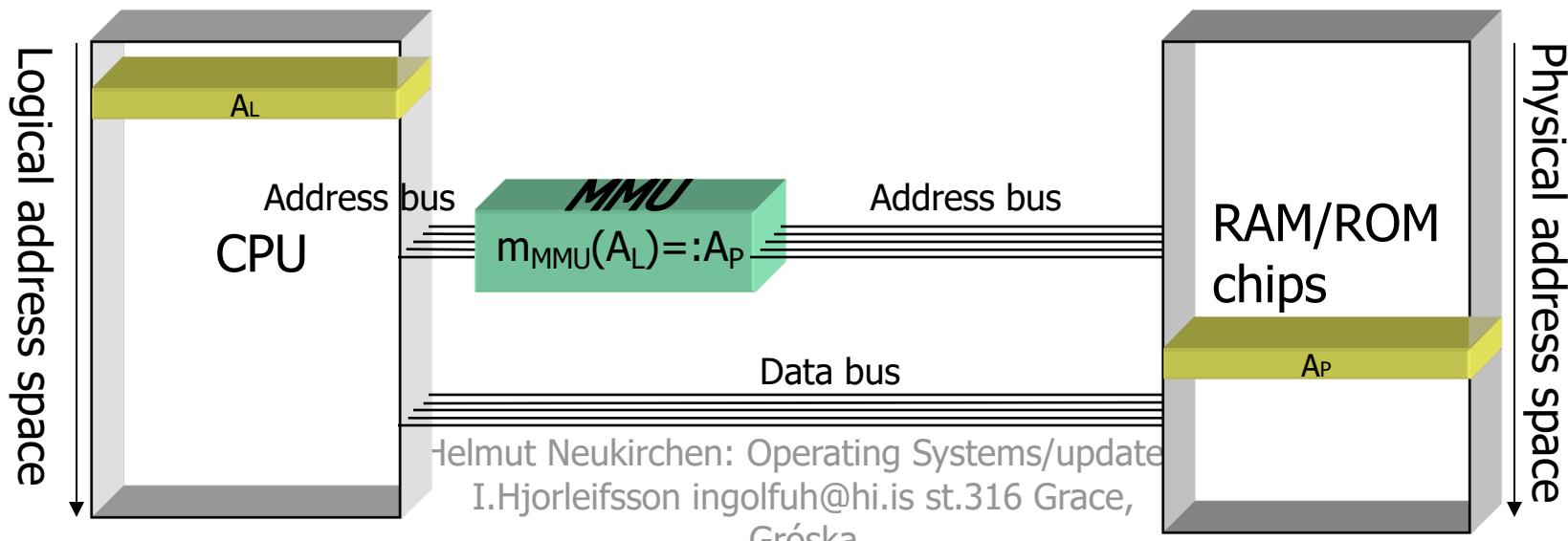
- In addition to its own instructions, a program typically calls instructions of a library.
 - E.g. the implementation of the C function `printf(...)` is contained in the C standard library (stdlib or libc).
- In the simplest case, the machine code instructions of all libraries needed by a program are linked to the program at compile time of that program (**static linking**).
 - Blows up size of executable program file on disk.
 - If multiple programs are in memory at the same time and these programs use the same library (which is likely for e.g. the C standard lib), **memory is wasted**, because all the programs have their own copy of the same library.
 - If an update of a library is published, the **programs that have already been linked, do still use the old library**.
 - Programs need to be **re-linked** using the new libraries.
 - (Re-linking requires the intermediate * .o object files of the program. I.e. re-linking not possible if just the executable program file is available.)

Excursion: Dynamic Linking / Dynamic-link Libraries (DLL)

- Dynamic linking of libraries avoids the problems of static libraries:
 - **Linking** occurs just **at execution time**:
 - At compile time (to be precise: at link time), only a small piece of code (a stub) that represents the library is linked to the program.
 - When a call to a library function is made by a program, the stub is called:
 - Stub asks OS whether library is already in memory: if not, OS loads it from a special library location from the file system (POSIX: /usr/lib) into a free memory location (to enable a library to run at any location, it must be relocatable).
 - Once library is in memory, stub locates memory location of the desired function in the dynamic library (using a symbol table) and calls that function.
 - To speed up further look-ups of the same library function, the memory location returned by the first lookup may be used next time to jump directly to the memory location of the desired function.
- ⇒ Size of executable program file is reduced (library exists only once in the file system); programs do not need to be re-linked to use a new library.
- ⇒ By sharing loaded libraries between all the running processes (shared library), also the main memory footprint can be reduced.

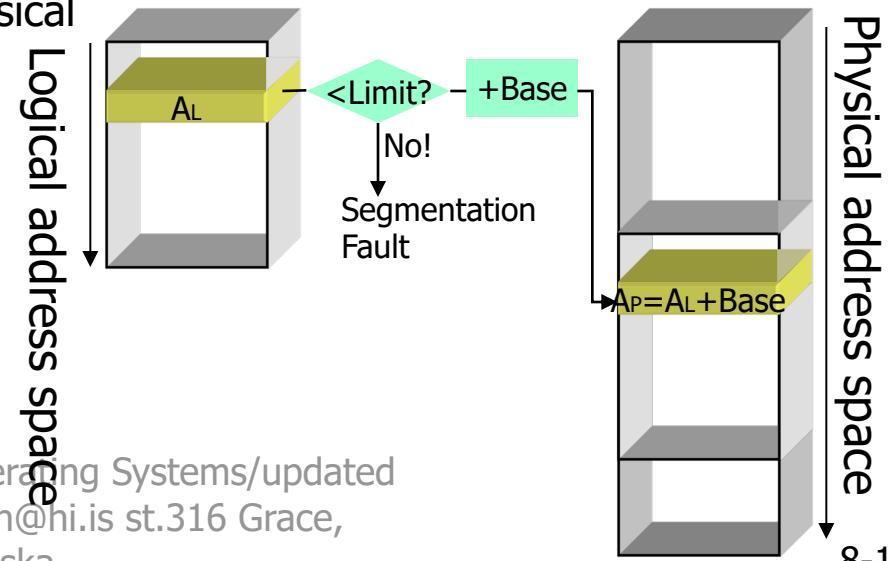
Organisation of Memory: MMUs & Logical and Physical Addresses

- A programmable **Memory Management Unit** (MMU) may be used to translate CPU addresses into addresses of the physical main memory:
 - MMU may be part of the CPU (e.g. in today's PCs) or a separate chip, or it may be completely absent (e.g. on cheap embedded systems).
 - Mapping from logical to physical addresses: $m_{\text{MMU}}: A_L \rightarrow A_P$
 - **physical address (A_P)**: real address in physical memory (view of MMU).
 - **logical address (virtual address) (A_L)**: address used by a program (view of CPU).



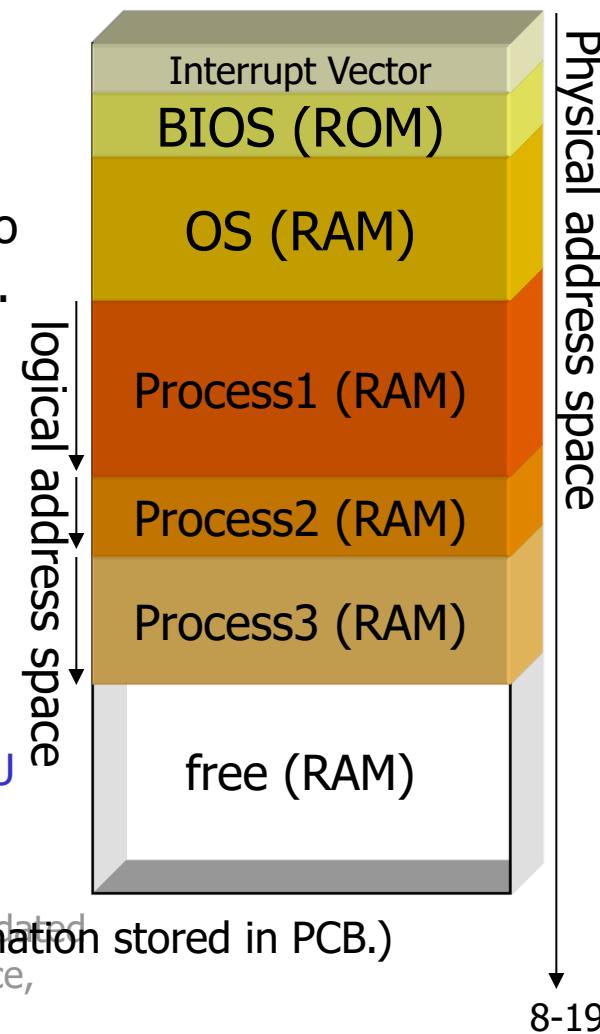
Organisation of Memory: Basic MMU Hardware

- MMU translates every logical address accessed by the CPU to a physical address.
- Most basic type of a programmable MMU: **Segmenting MMU** (obsolete, but Intel CPUs still support it).
 - Can be programmed using two registers:
 - **Base register**: physical address of logical address 0,
 - **Limit register**: maximum allowed logical address.
 - Segmenting MMU generates physical address by adding base register to logical address as long as it is within the limit:
 $A_P = A_L + \text{Base}$
 - If logical address is outside limit: Segmentation Fault interrupt is raised by MMU.
 - OS interrupt handler then typically terminates process that has gone wild.



Organisation of Memory: MMU-supported Multiprogramming

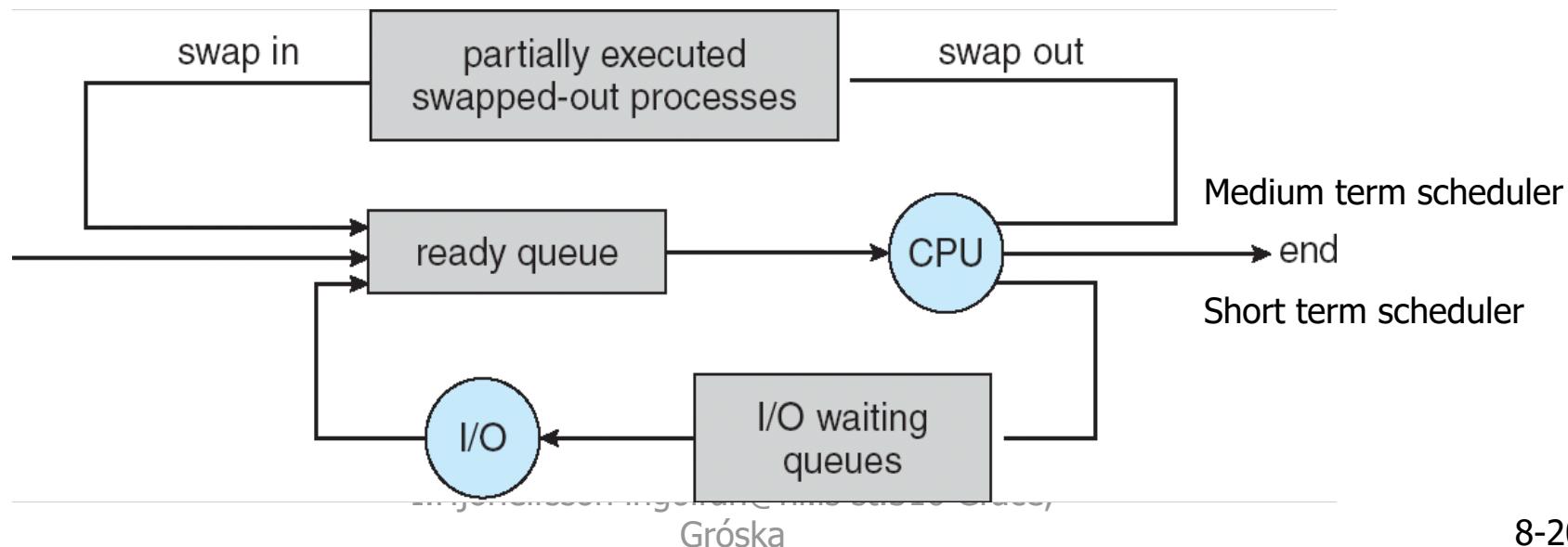
- Multiple processes, each having its own logical address space.
(e.g. POSIX, MS Windows since Windows NT)
 - MMU is reprogrammed at each context switch to give each process its own logical address space.
 - No relocation required, as each process has its own address space (each starting with logical address 0); hence all programs can be compiled for the same absolute memory location.
 - Memory protection: memory of other processes or of the OS not accessible. (When trying to access logical memory that is not allocated to current process: Segmentation Fault interrupt generated by MMU.) At each context switch, MMU is reprogrammed according to physical memory owned by current process!



8.2 Swapping: Reminder Chapter 3

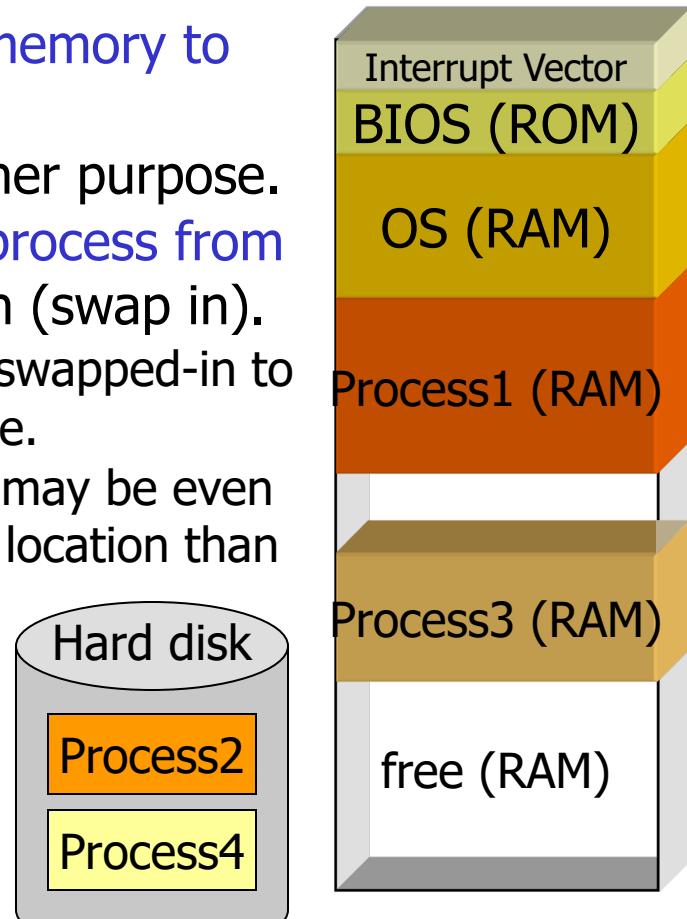
Medium Term Scheduling

- Sometimes, degree of multiprogramming needs to be temporarily reduced.
 - E.g. memory is filled up by processes and now, one of these processes requires further memory. The only solution is to (temporarily) remove one of the processes from memory.
 - A **medium term scheduler** may be used to identify processes that should not be considered by short-term scheduler for some time:
 - Processes are removed from main memory to hard disk (**swapped out**) and later-on, e.g. if another process terminates, put into main memory again (**swapped in**).



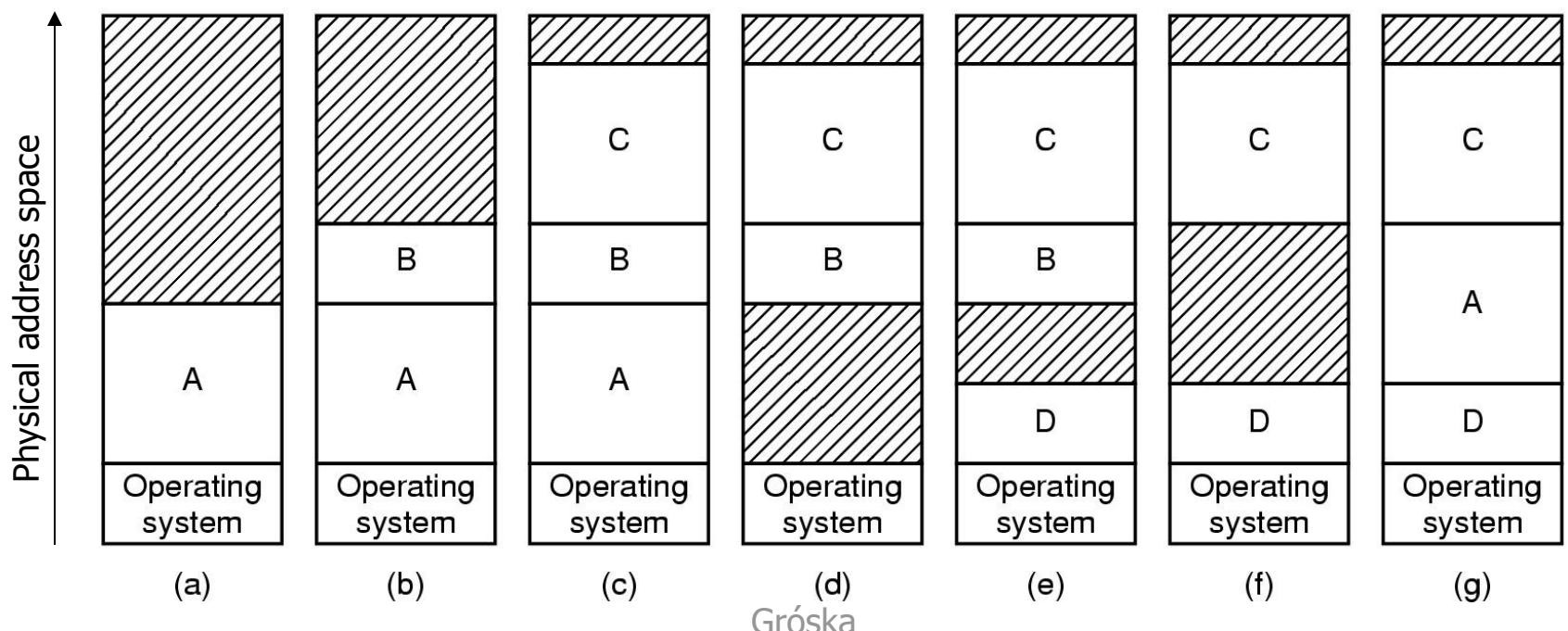
Swapping

- Transfer a process as a whole from main memory to hard disk (swap out).
- Use the gained main memory for some other purpose.
- Later, re-transfer the swapped-out whole process from hard disk to memory to continue execution (swap in).
 - If no MMU is available: processes must be swapped-in to exactly the same memory location as before.
 - If segmenting MMU is available: processes may be even swapped-in to a different physical memory location than before:
 - By reprogramming the base register, the MMU adapts the logical addresses to the new physical addresses, hence a process may just continue execution using the unchanged logical addresses.
- Today, swapping is not used very much, as it is inefficient to swap out whole processes. (Instead: Paging → 8.4)



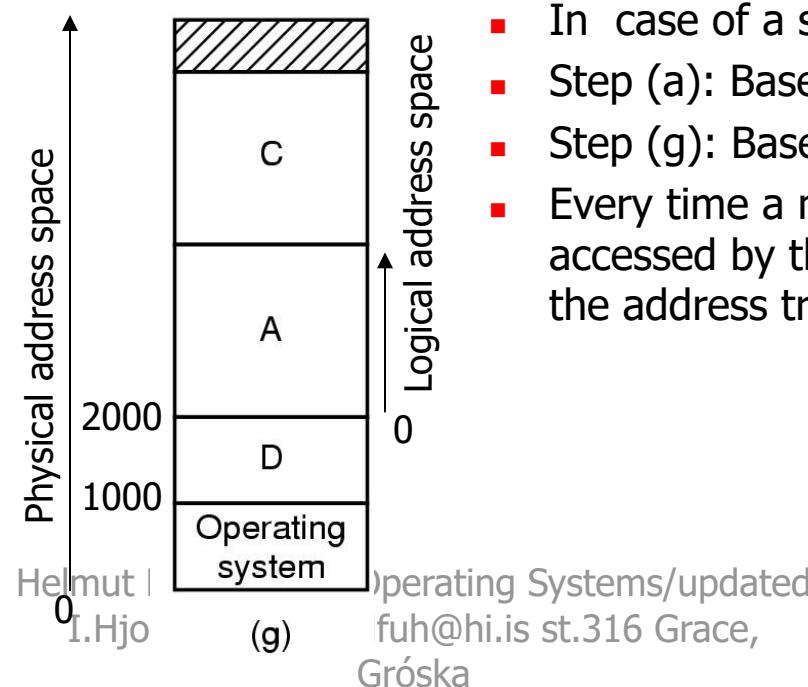
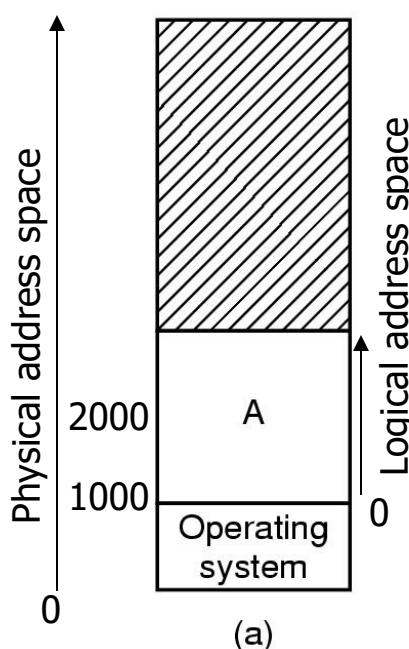
Swapping: Example using MMU

- First (a), process A is started. Then (b), process B is started and (c) process C is started.
- Next, process D shall be started. However, no sufficient space is left in the physical memory, hence e.g. process A is selected to get swapped-out (d). After that, process D can be started (e).
- Next, process A shall be run again. To make space for process A, e.g. process B is selected to be swapped-out (f). After that, process A can be swapped-in again (g).



Swapping: Example using MMU Logical vs. physical addresses

- Assume, before swapping out (a), logical address 0 of A was at physical address 1000.
- After swapping in again (g), logical address 0 of process A might then be at physical address 2000.
- A program knows only about logical addresses, hence the MMU needs to achieve that in step (a) logical address 0 is mapped to physical address 1000, but in step (g) logical address 0 is mapped to physical address 2000.



- In case of a segmenting MMU:
- Step (a): Base=1000
- Step (g): Base=2000
- Every time a memory location is accessed by the CPU, the MMU does the address translation.

8.3 Contiguous-Memory Allocation

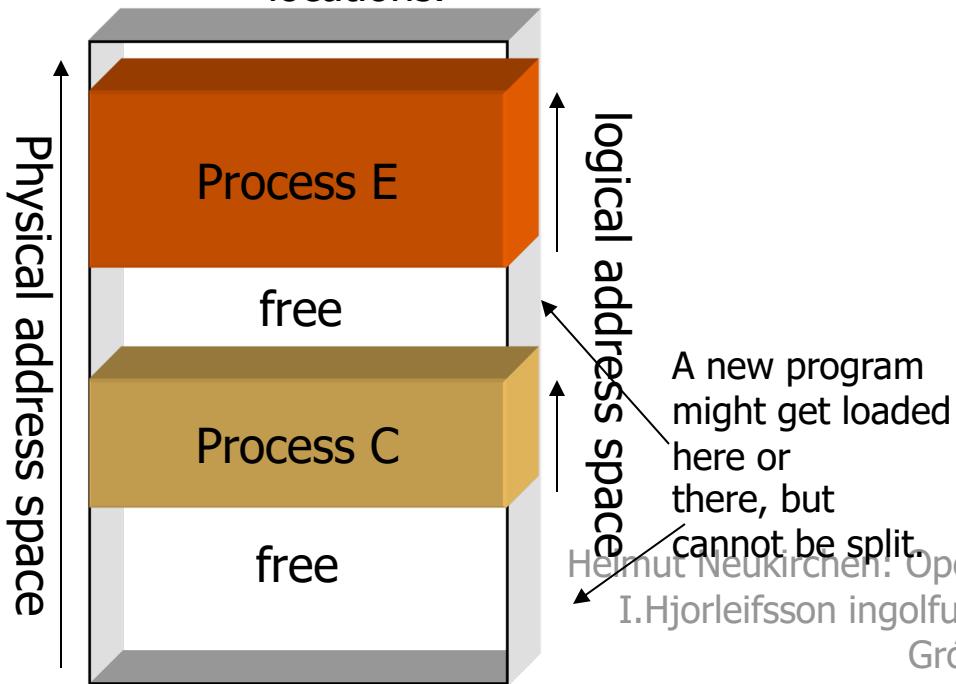
- As we have seen on the previous slides, there may be “holes” in the memory, i.e. free memory blocks next to allocated memory blocks.
 - As instructions of a program need to be in **consecutive memory locations** (the compiler keeps the code as compact as possible, i.e. consecutive instructions), it is not possible to gather multiple scattered smaller free memory locations to satisfy a request for a larger block of memory.
 - (At least, not with a segmenting MMU. – We will later-on learn more about paged MMUs that are more flexible.)

Contiguous-Memory Allocation and Release

- OS manages:
 - 1) Memory for whole processes,
 - 2) Memory within a process.
- Situations where OS has to **allocate free contiguous memory holes**:
 - Physical memory (simple multi-programming only): When a new process is started or when a swapped-out process is swapped in.
 - Logical memory: During run-time of a process: process may dynamically request memory (e.g. in Java or C++ using `new`, in C using the `malloc` C library call) from the “heap” that has been reserved for the process.
- Situations when allocated memory is **released**:
 - Physical memory: Processes termination. When a process is swapped out.
 - Logical memory: Release of heap memory (C++: `delete`, C/Unix: `free`).

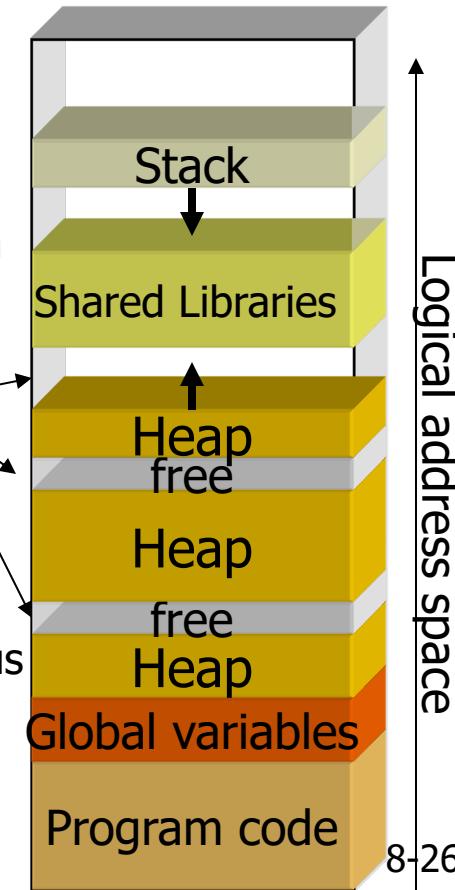
Contiguous-Memory Allocation Visualised

- Programs need to be loaded to contiguous logical memory locations.
 - At least with no MMU or with segmenting MMU this means: contiguous physical memory locations.



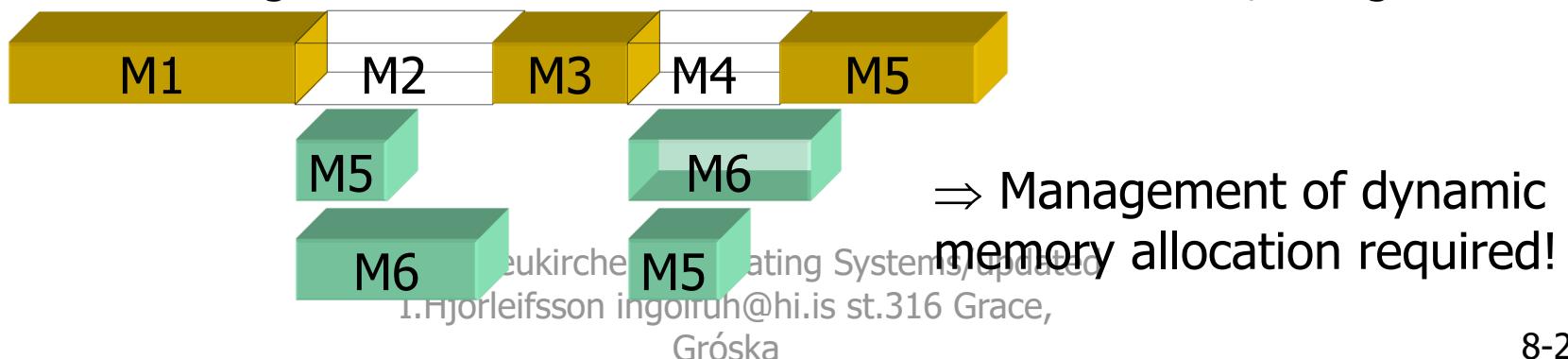
- Process may during runtime request further memory from the heap of their logical address space.

- After a couple of requesting and releasing memory from the heap, the heap may look like a Swiss cheese: a lot of holes.
- From these holes, a contiguous logical memory area will be needed to satisfy the next request of contiguous memory from the heap.



Problem of Contiguous-Memory Allocation

- When multiple holes are available to satisfy a memory allocation request, OS has to decide on which hole to use.
- Example:
 - Memory locations M2 and M4 have been released.
 - Now, requests for M5 and M6 are pending: which hole to chose for M5, which one for M6? ("Dynamic storage-allocation problem")
 - If we would put M5 in the hole of M2, this would satisfy M5. However, M6 cannot be satisfied anymore!
 - Putting M5 into hole of M4 and M6 into M2 would work, though.



Problem of Fragmentation

- Typically, memory allocation requests do not fill exactly a memory hole: some part of the hole is used for the allocation request, the remainder becomes a smaller hole again:



- ⇒ As a result, memory contains after some time many small holes:
- **External fragmentation:** Free memory that is, however, fragmented into many non-contiguous holes.
 - Even if the sum of the holes would satisfy a memory request, the memory is non-contiguous and can thus not satisfy the request for contiguous memory. (Solution if MMU is available to relocate whole processes in physical memory at run-time: shift them so that one large block free remains (**compaction**).)
 - **Internal fragmentation:** Free memory within the memory allocated to a process, that cannot be used to satisfy requests of other processes.
 - The MMU enforces memory protection, hence internally fragmented memory cannot be accessed by (and thus not allocated to) other processes.

⇒ Dynamic memory allocation strategies should minimise fragmentation.

Dynamic Memory Allocation Strategies

- Many strategies possible, e.g.:
 - **Best Fit:** Allocate the smallest hole that is big enough.
 - Produces the smallest leftover holes: sounds nice in theory. However, practise shows that the resulting leftover holes are so small that they can typically not be used to satisfy any future requests (fragmentation).
 - **Worst Fit:** Allocate the largest hole.
 - Produces the biggest leftover holes: sounds like a remedy for the problem of Best Fit. However, practise shows that huge holes are quickly degraded into smaller ones that cannot satisfy average sized future requests (fragmentation).
 - **First Fit:** Always search from the beginning of the free holes list to allocate the **first** hole that is big enough.
 - Small holes tend to accumulate at the beginning of the free holes list, making the search for bigger holes farther and farther (=slower) each time.
 - **Next Fit:** Like First Fit, however search for memory hole starts where last request has been satisfied.
 - (I.e. if hole used to satisfy last request has not been completely occupied by last request, that resulting smaller hole is used as starting point for next request.)
 - Eliminates the problem of First Fit.

Dynamic Memory Allocation Strategies: Example

- Holes of free memory are available in the following order: 10 KB, 5 KB, 14 KB, 9 KB, 33 KB and 25 KB. Now, contiguous-memory m_i of size $m_1=13$ KB, $m_2=6$ KB, $m_3=2$ KB, $m_4=17$ KB and $m_5=18$ KB is requested in consecutive order.

Holes:	10 KB	5KB	14 KB	9 KB	33 KB	25 KB
First Fit	m2=6 KB		m1=13 KB		m4=17 KB	m5=18 KB
	m3=2 KB		1 KB		16 KB	7 KB
	2 KB					
Next Fit			m1=13 KB	m2=6 KB	m4=17 KB	m5=18 KB
			1 KB	m3=2 KB	16 KB	7 KB
				1 KB		
Best Fit			m1=13 KB	m2=6 KB	m5=18 KB	m4=17 KB
			1 KB	m3=2 KB	15 KB	8 KB
				1 KB		
Worst Fit					m1=13 KB	m2=6 KB
					m3=2 KB	m4=17 KB
					m5=18 KB	2 KB

8.4 Paging

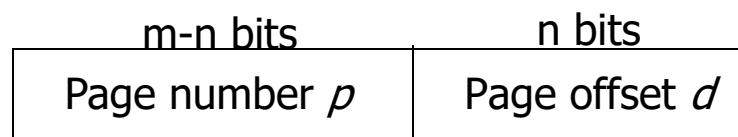
- While the basic segmenting MMU supports relocation of processes and memory protection, only one contiguous block of physical addresses can be mapped by a segmenting MMU to a process.
 - ⇒ Requests for process memory must be satisfied using contiguous physical memory.
 - ⇒ Only whole processes (not parts of it) may be swapped out and in.
- ⇒ Segmenting MMUs have been superseded by Paged MMUs.

Pages and Frames

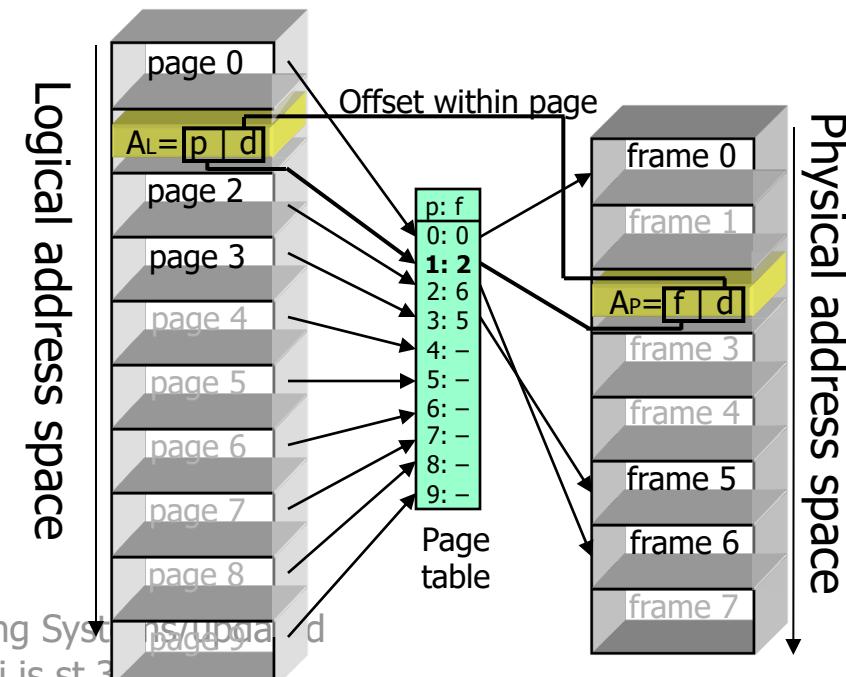
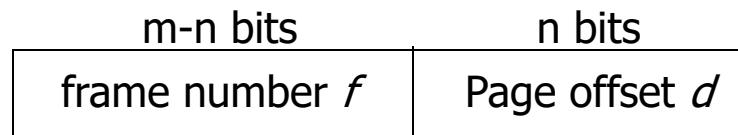
- Instead of mapping contiguous blocks of logical and physical memory, Paged MMUs divide the logical memory into multiple fixed-sized **pages** that can be mapped onto physical **frames** of the same size.
 - While the memory within each page and frame must be contiguous, the consecutive pages of a process may be mapped on frames scattered in the physical memory.
 - ⇒No external fragmentation.
 - However still internal fragmentation due to fixed size of pages/frames.
 - ⇒ Even parts of a process may be stored to and retrieved from a hard disk.

Advanced MMU Hardware: Paged MMUs

- More flexible: Paged MMU (PMMU).
 - Logical address space is divided into **pages** of fixed size.
 - Pages are mapped onto **frames** in the physical memory using a **page table**.
 - Pages/frames have same size that is a power of 2, i.e. page size= 2^n .
 - Logical address A_L (having m bits) generated by CPU is divided into:

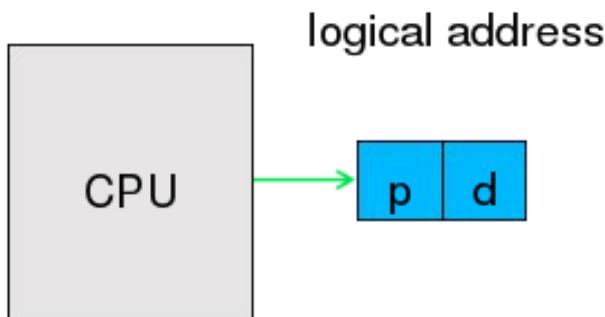


- By looking up the frame f onto which page p is mapped according to the page table, the resulting physical address A_P (m bits) is:



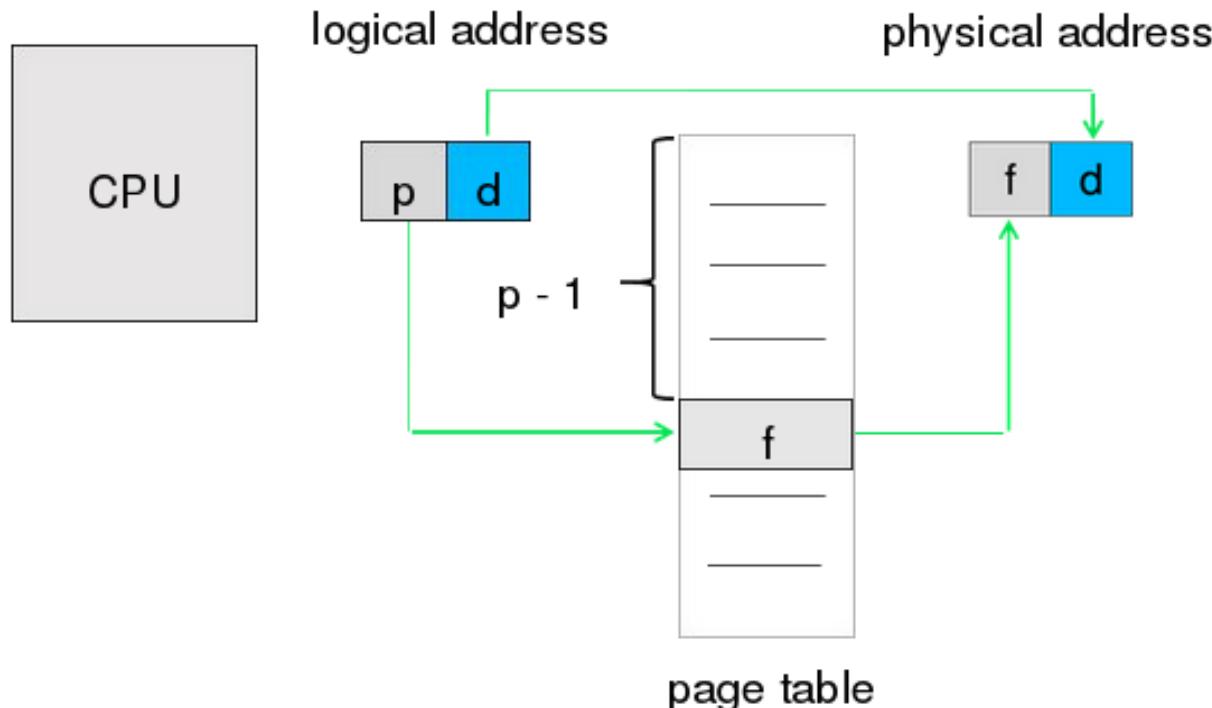
PMMU Address Translation Step 1

1. the logical address generated by the CPU is sent to the MMU where it is divided into a page number (p) and an offset (d)
 - the number of bits in each part depends on the page size



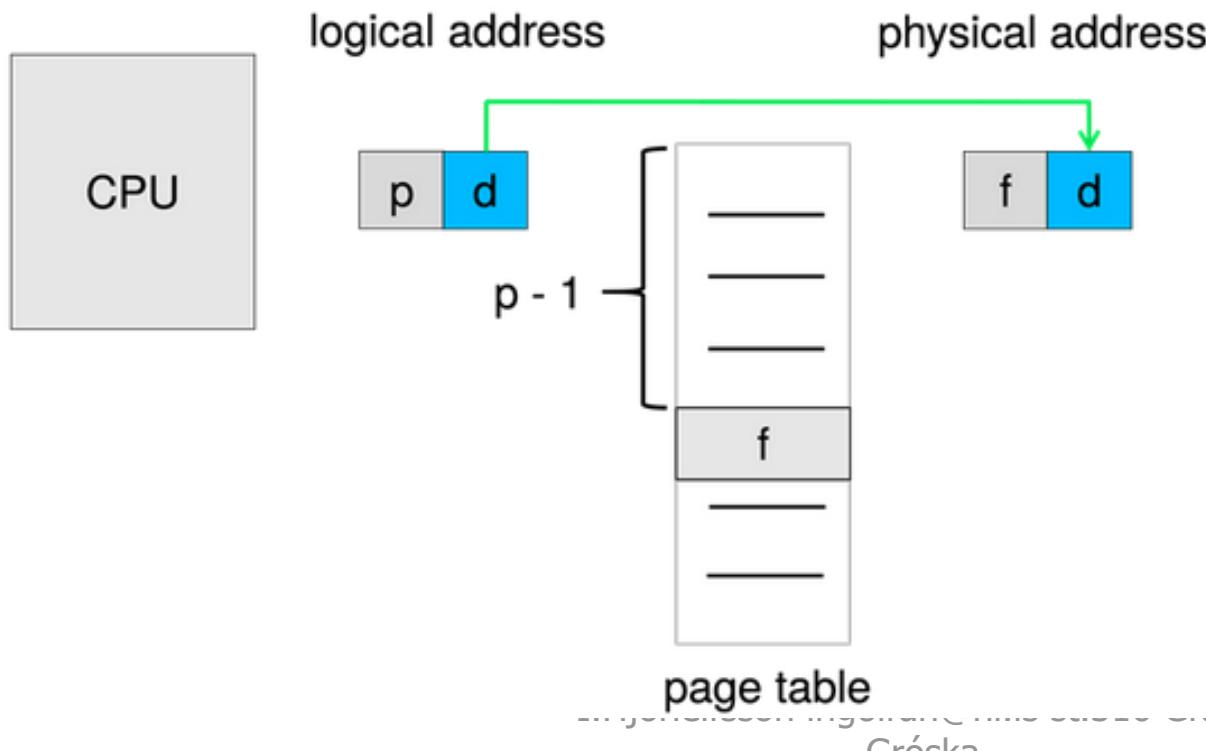
PMMU Address Translation Step 2

2. the page number is used as an index into the page table
 - the entry in the page table at that index is the number of the frame of physical memory containing the page
 - that frame number is the first part of the physical memory address



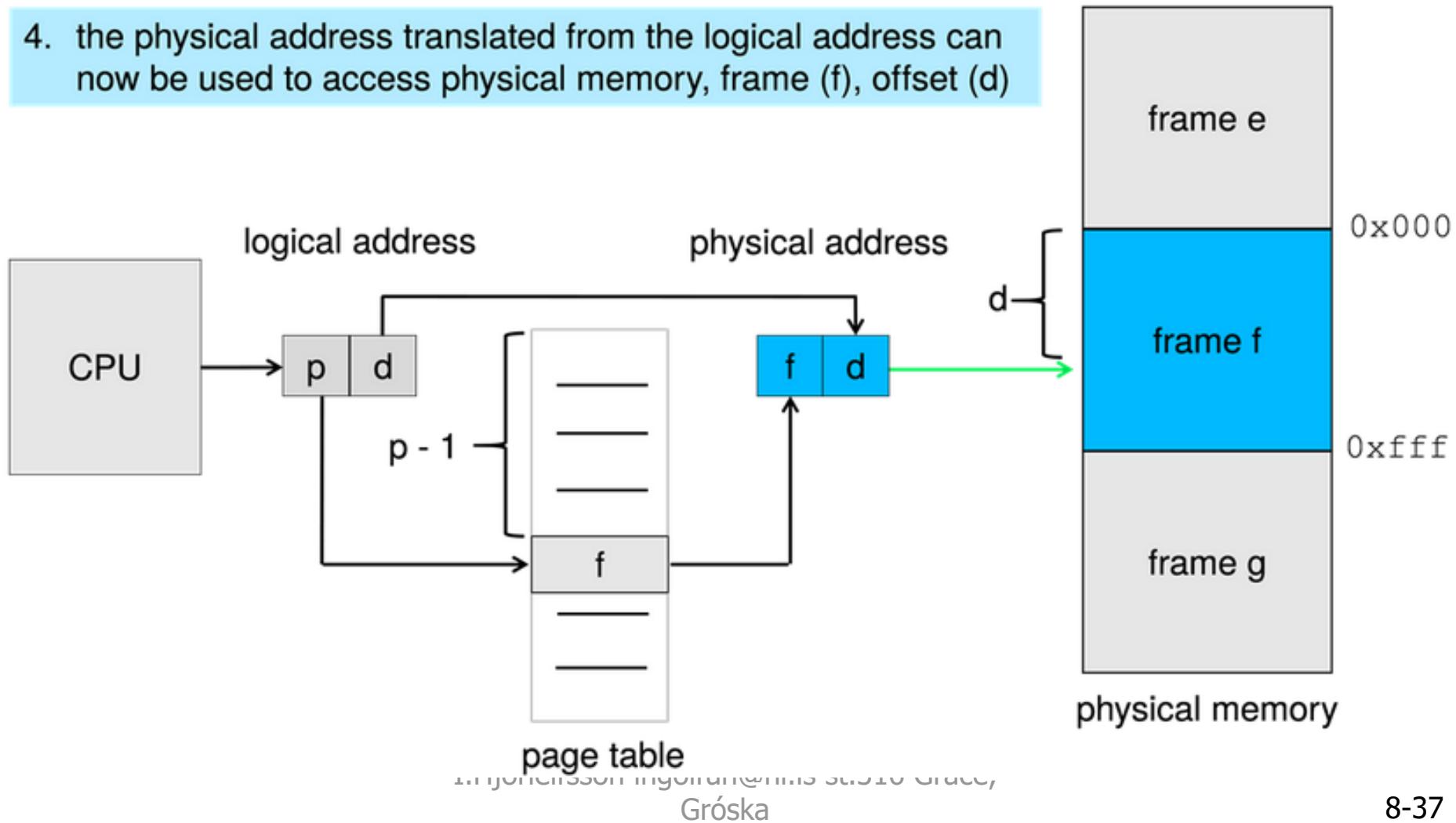
PMMU Address Translation Step 3

- the offset (d) within the page is the same as the offset within the frame, so it is retained, together with the frame number forming the physical address



PMMU Address Translation Step 4

- the physical address translated from the logical address can now be used to access physical memory, frame (f), offset (d)

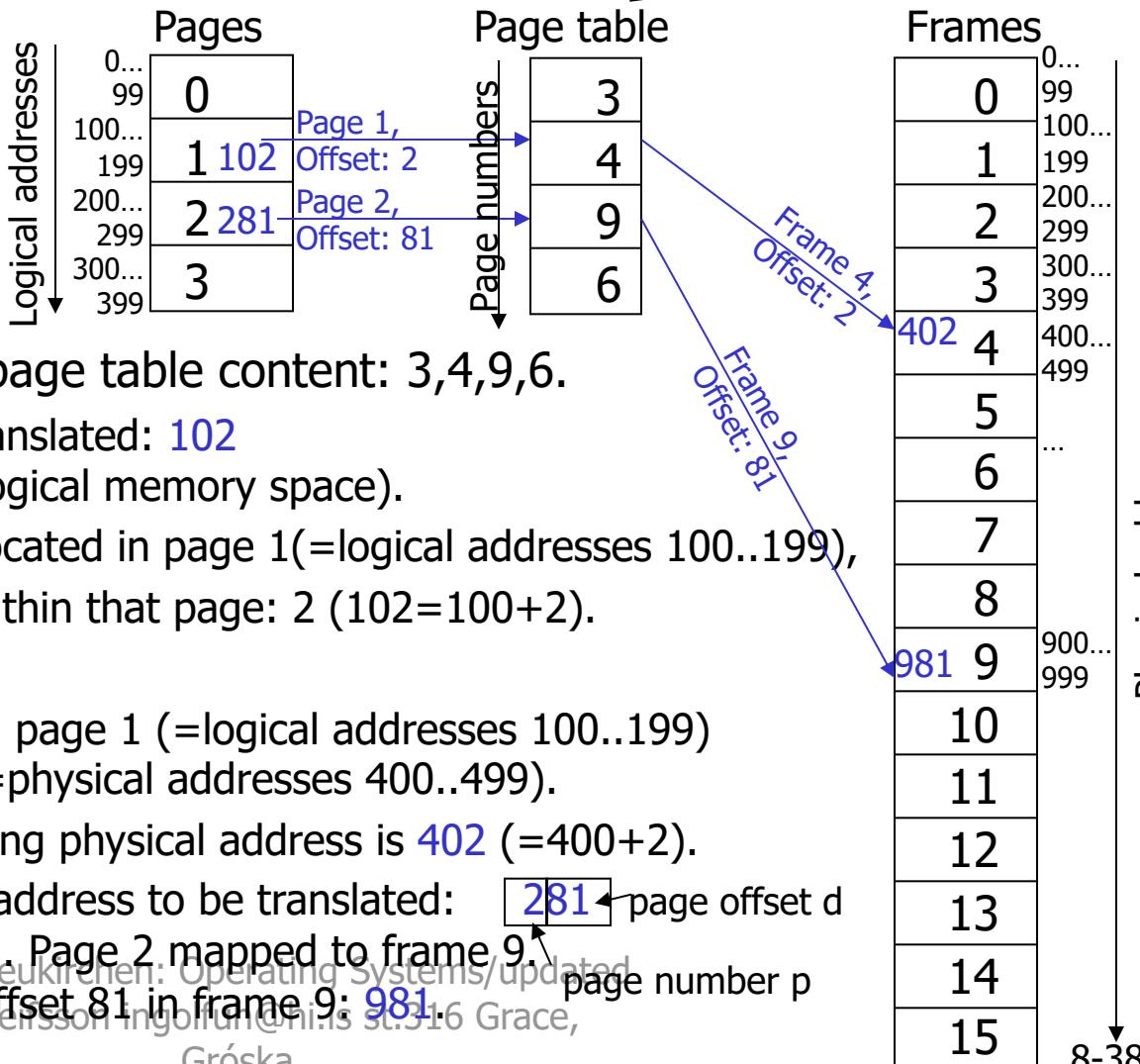


Example address translation by PMMU

Each process has its own page table (stored in PCB; PMMU's pointer to page table location is re-programmed as part of context switch).

- For simplicity, assume page sizes that are a power of 10 (not 2 as in real hardware).
- Example: page size =frame size=100 bytes, page table content: 3,4,9,6.

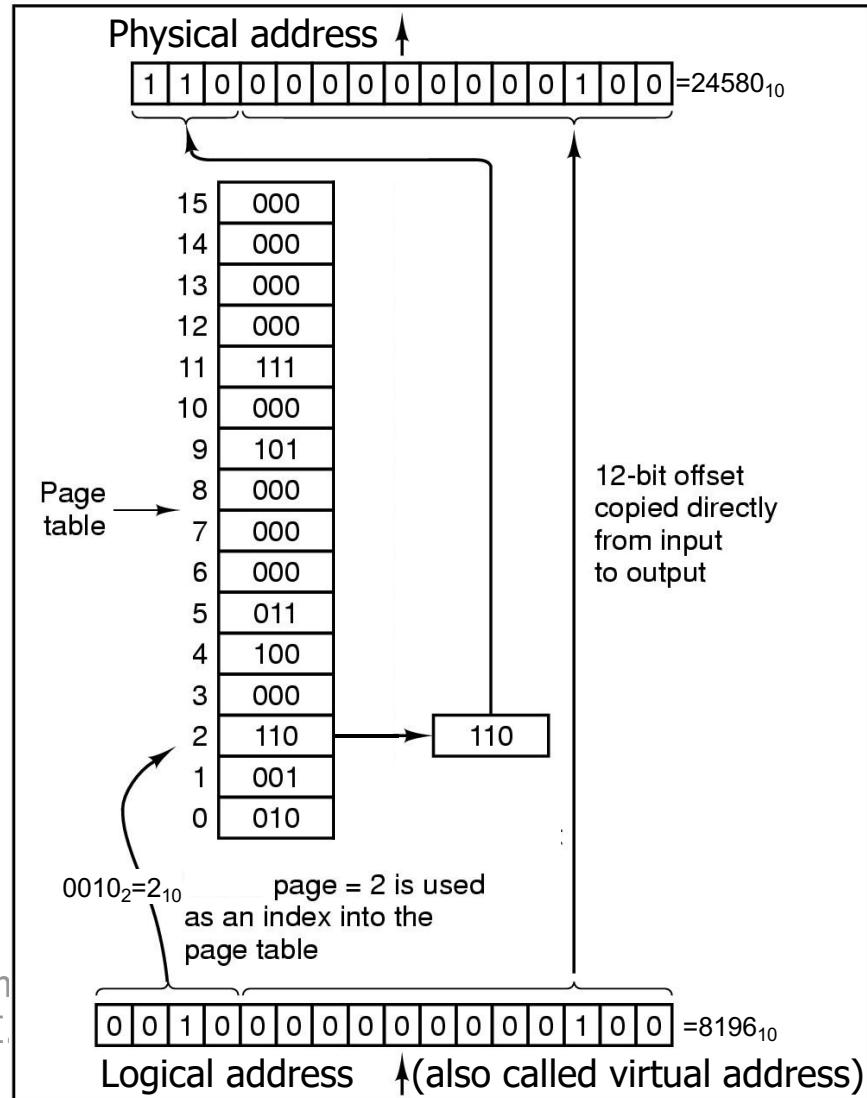
- Logical address to be translated: 102 (=byte number 102 in logical memory space).
- Logical address 102 is located in page 1(=logical addresses 100..199),
- Offset of address 102 within that page: 2 (102=100+2).



- According to page table, page 1 (=logical addresses 100..199) is mapped to frame 4 (=physical addresses 400..499).
- With offset 2, the resulting physical address is 402 (=400+2).
- Other example: Logical address to be translated: 281 → page 2, offset 81. Page 2 mapped to frame 9. → Physical address of offset 81 in frame 9: 981.

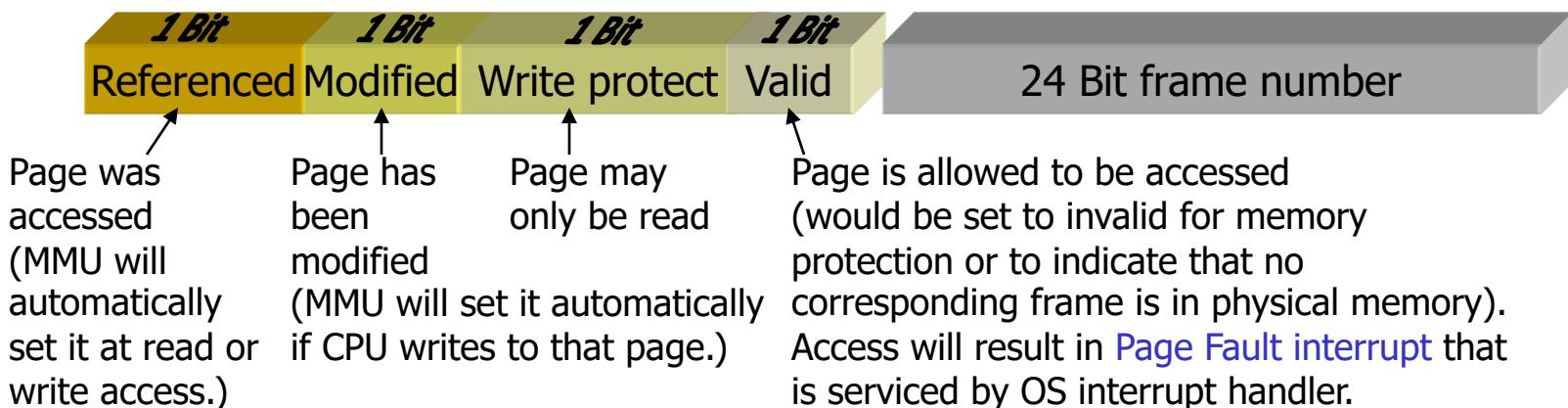
Advanced MMU hardware: Inner life of a Paged MMU

- Upper $m-n$ bits (=page number p) of logical address used as index into page table.
- Entry of page table (=frame number f) replaces upper $m-b$ bits of address.
- Lower n bits (=page offset d) of address copied without modification.
- Resulting address is used as physical address.



PMMU: Page Tables

- Today's PMMUs support page sizes between 256 B and 2 GB.
 - Most operating systems use 4 KB or 8 KB page size (Solaris even 4 MB).
- Structure of a page table entry: 4 Byte on a 32 bit system (assuming a 32 Bit address space and 256 Byte page size):



- On 64 bit systems, the physical address space is bigger and thus more than 2^{24} frames may exist.
 - Pages table entry is 8 Byte allowing more bits for the frame numbers.

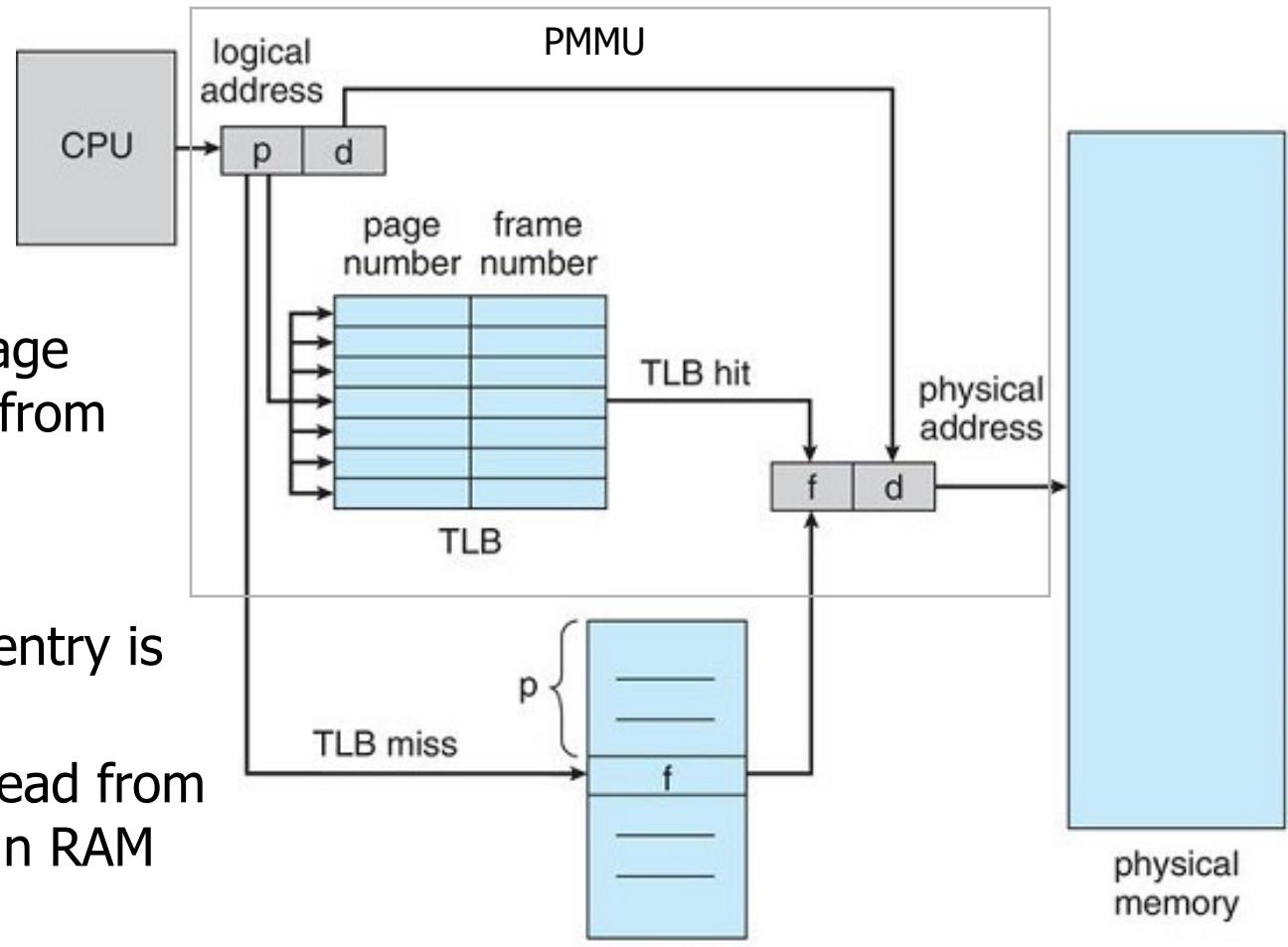
PMMU: Translation Lookaside Buffer TLB

- Page table can be huge and thus cannot be kept in PMMU-internal registers, but rather in the RAM.
 - The MMU is anyway in charge of managing the RAM, so no problem to access RAM in order to read page table.
 - Memory access slowed down by factor of 2 due to access to page table:
 1. Access to page table that is located in RAM,
 2. Access to final physical address.
- ⇒ Cache for page table entries (**Translation Lookaside Buffer TLB**):
- If page table entry is in TLB: only 1 RAM access needed (+ fast TLB access).
 - Memory access with PMMU and TLB almost as fast as without any PMMU.
 - Problem: Context switch.
 - Each process has its own page with completely different page table entries.
 - TLB/PMMU need to keep page table entries from each process apart.
 - Each process has an address-space identifier that is stored in each TLB entry.

PMMU with TLB

- **TLB hit:**

- Page table entry is in TLB.
- Can read page table entry from fast TLB.



PMMU: Size of Page Tables

- Example: 4KB ($=2^{12}$ B) page size, 32 Bit logical address space:
 - Address spaces consists of $2^{32}/2^{12} = 2^{20} \approx 1$ million pages.
Hence, page table needs 1 million entries, too.
 - Each page table entry is 4 Byte $\Rightarrow \approx 4$ MB for page table!
 - A typical translation lookaside buffer (TLB) has place for caching between 64 and 1536 entries.
 \Rightarrow Only a small fraction of the page table would be cacheable by the PMMU!
(Even worse with 64 Bit logical address space.)
- Luckily, the logical address space is almost never used completely; instead, there are many contiguous unused areas for which the valid bits of their corresponding page table entries would be set invalid.
 \Rightarrow Divide the page table into sections representing a contiguous set of pages: represent a section that contains only invalid pages by only one page table entry that indicates that all pages represented by this entry are invalid.

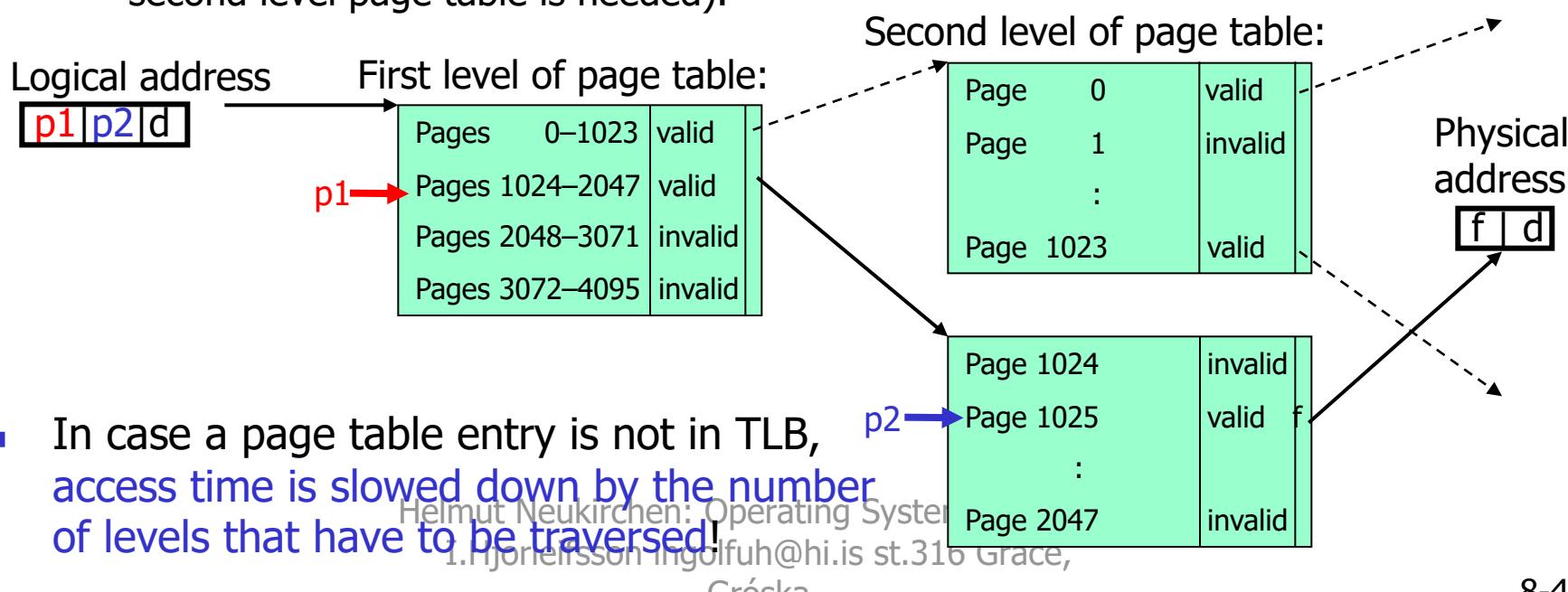
Multiply by 2^{32} for
a 64 Bit system!

\Rightarrow Multi-level page tables (\rightarrow next slide).

Without the kitchen Optimizing Systems/updated
I.Hjorleifsson ingolfuh@hi.is st.316 Grace,
Gróska

Hierarchical Paging: Multi-level Page Tables

- Example: Two-level page table.
 - Frame number of a first level page table entry points to a frame containing the second level page table for all the pages represented by the first level page table entry or
 - First level page table entry is set to invalid to indicate that all the pages represented by the first level page table entry are invalid (in this case, no frame containing second level page table is needed).



- In case a page table entry is not in TLB, access time is slowed down by the number of levels that have to be traversed!

Paging on 64 bit systems

- With modern 64 bit CPUs, it is even more likely that huge areas of the logical address space are unused (=set to invalid in the page tables entries).
- Typically, at least 4 levels multi-level page tables used by 64 bit OSes (and supported by PMMUs of 64 bit CPUs).
 - May lead to a 4 times slowdown of access times if page table entry is not in TLB (or more if more than 4 levels are used).
- Trends for PMMUs in 64 bit non-Intel CPUs:
 - **Hashed page tables, Inverted page tables.**
 - Avoid huge page tables and slowdown due multi-level lookup times.
 - Hashing techniques used instead.
 - Details not covered here.

Applications of Paging: Solve Fragmentation, Memory Protection

- A PMMU can be used to implement:
 - Getting rid of external fragmentation:
 - A free frame can be allocated to the contiguous logical address space of any process. (However, internal fragmentation due to fixed page size remains.)
 - Memory protection using write-protect & (in)valid flag of each page:
 - Invalid: To indicate memory that belongs to another process or is not available at all,
 - Write protect: Read-only memory,
 - Any other ("normal") page: Read and writable.
 - Switch PMMU page table pointer at each context switch.
 - Page table of each process ("address space") stores process' PCB.
 - Virtual memory: see chapter 9...
 - Shared memory: see next slides.

Applications of Paging: Shared Memory

- Shared memory=Map pages of different processes to the same frame:
 - Shared libraries:
 - Dynamic link library is loaded into physical memory only once and mapped (with read-only protection) into logical address space of all processes. ⇒ Reduces physical memory footprint of processes using this library.
 - Starting the same program multiple times:
 - OS detects that a program file has already been loaded to physical memory, hence frames containing program's instructions are mapped (with read-only protection) into logical address space of new processes.
 - Process communication using shared memory (→chapter 3):
 - Map shared memory frame into logical address space of all processes that want to share memory.
 - Reminder from slide 3-32: POSIX system calls for shared memory, e.g.:
 - `int shmget (long key, int size, int flag)`: create a new shared memory area (=frame) or retrieve shared memory handle (will be the return value) using a key (on which applications have to agree on).
 - `void* shmat (int id, char* addr, int flag)`: map shared memory area to page with `addr` using handle `id` to identify previously created area/frame.

8.5 Summary

- Different memory-management hardware (no MMU, segmenting MMU, paged MMU) support different memory organisation (from monoprogramming to advanced multiprogramming).
 - The resulting memory-management strategies can be compared based on:
 - Required **hardware support**: PMMU allows more sophisticated strategies than segmenting MMUs that are still better than no MMU.
 - **Performance**: MMUs add a performance penalty to each memory access. While segmenting MMUs are fast, PMMUs are slow; however, a TLB reduces this problem.
 - **Fragmentation**: Fixed-size allocation units (pages) suffer from internal fragmentation, one variable-size allocation unit (segmenting) suffers from external fragmentation.
 - **Relocation**: MMUs support relocating code, thus enabling compaction to remove external fragmentation (not an issue with PMMU).
 - **Swapping**: Whole processes are copied to hard disk and back to allow running more processes than can be fit into memory at one time.
 - **Sharing**: Sharing of pages between processes allows to fit more processes into memory, thus increasing the multiprogramming degree.
 - **Protection**: PMMUs support pages to be invalid, read-only, read-write; thus, enabling memory protection and sophisticated strategies, e.g. copy-on-write (→next chapter).

Course
TÖL401G: Stýrikerfi /
Operating Systems
9. Virtual-Memory Management

Chapter Objectives

- Define virtual memory and describe its benefits.
- Illustrate how pages are loaded into memory using demand paging.
- Apply the optimal, FIFO, LRU and Second-Chance page-replacement algorithms.
- Describe thrashing the working set of a process, and explain how it is related to program locality.
- Explain memory compression as alternative to paging out to storage devices
- Describe advanced applications of virtual memory, e.g. copy-on-write or demand loading.
- Explain management of kernel-internal memory.

Contents

1. Introduction
2. Virtual Memory Using Demand Paging
3. Page Replacement Algorithms
4. Allocation of Frames,
5. Thrashing & Working-Set Models
6. Memory Compression Instead of Paging Out to Storage Device
7. Advanced Applications of Virtual Memory
8. Allocating Kernel Memory
9. Summary

Note for users of the Silberschatz et al. book: newer editions have the material ordered differently than on my slides. (In particular, memory mapped files are covered there in this chapter, while I will cover them in the next chapter.)

Helmut Neukirchen: Operating Systems/updated
I. Hjorleifsson email: Ingolfur@hi.is. St. 510

9.1 Introduction

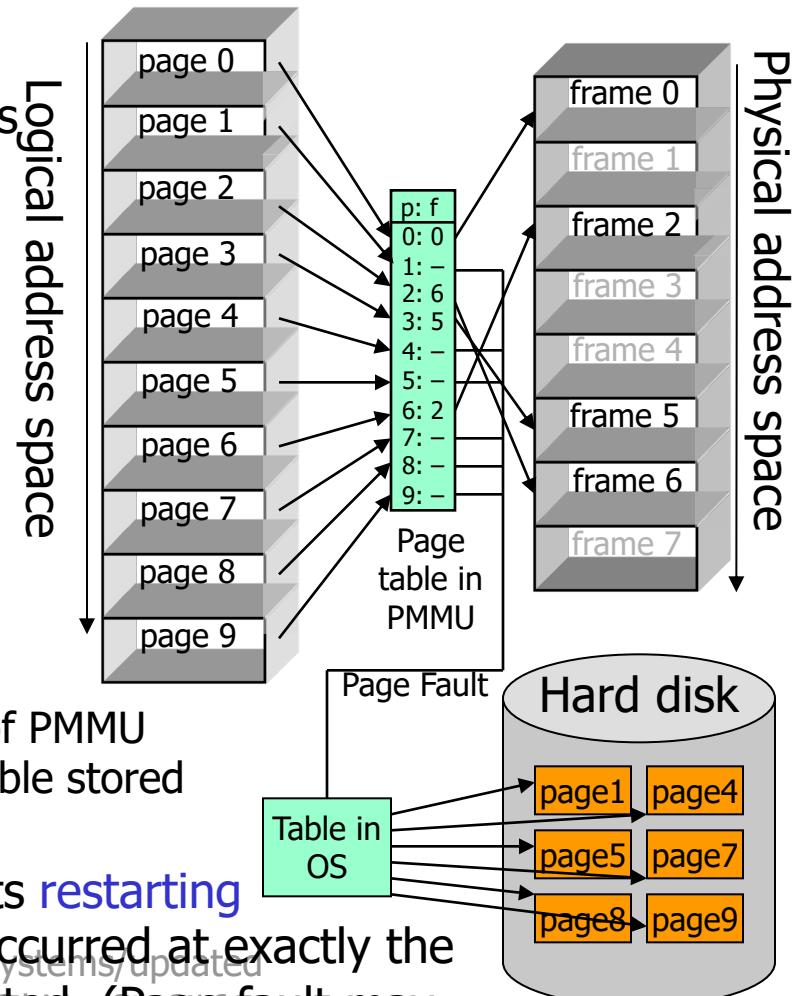
- Even though the previous chapter presented already some advanced usages of an MMU, it was still necessary that the complete logical address space used by a process is kept in physical memory while the process is executed.
 - E.g. swapping always swaps out and in the used logical address space of a process as a whole. It is not possible, that only a part of a process is swapped in or out.
- However, in practise, a process rarely accesses all of its logical address space or at least not all parts of the logical address space need not to be in physical memory at the same time.

Virtual Memory

- Virtual memory: separation of logical memory from physical memory, i.e. logical address space does not need to map 1:1 on existing physical memory anymore.
 - Only part of a program needs to be in memory for execution.
 - Instead of swapping whole processes in and out, only parts need to be stored and reloaded from hard disk (less I/O \Rightarrow significant speed up).
 - More programs can be kept in physical memory (increased degree of multiprogramming) if only that part of each program's logical address space is kept in physical memory that is actually currently needed.
 - Logical address space can be much larger than physical address space.
 - Programmers (& users) do not need to worry about the amount of physically available memory.
- Virtual memory typically implemented via demand paging.
 - Efficient: used by all of today's major operating systems.

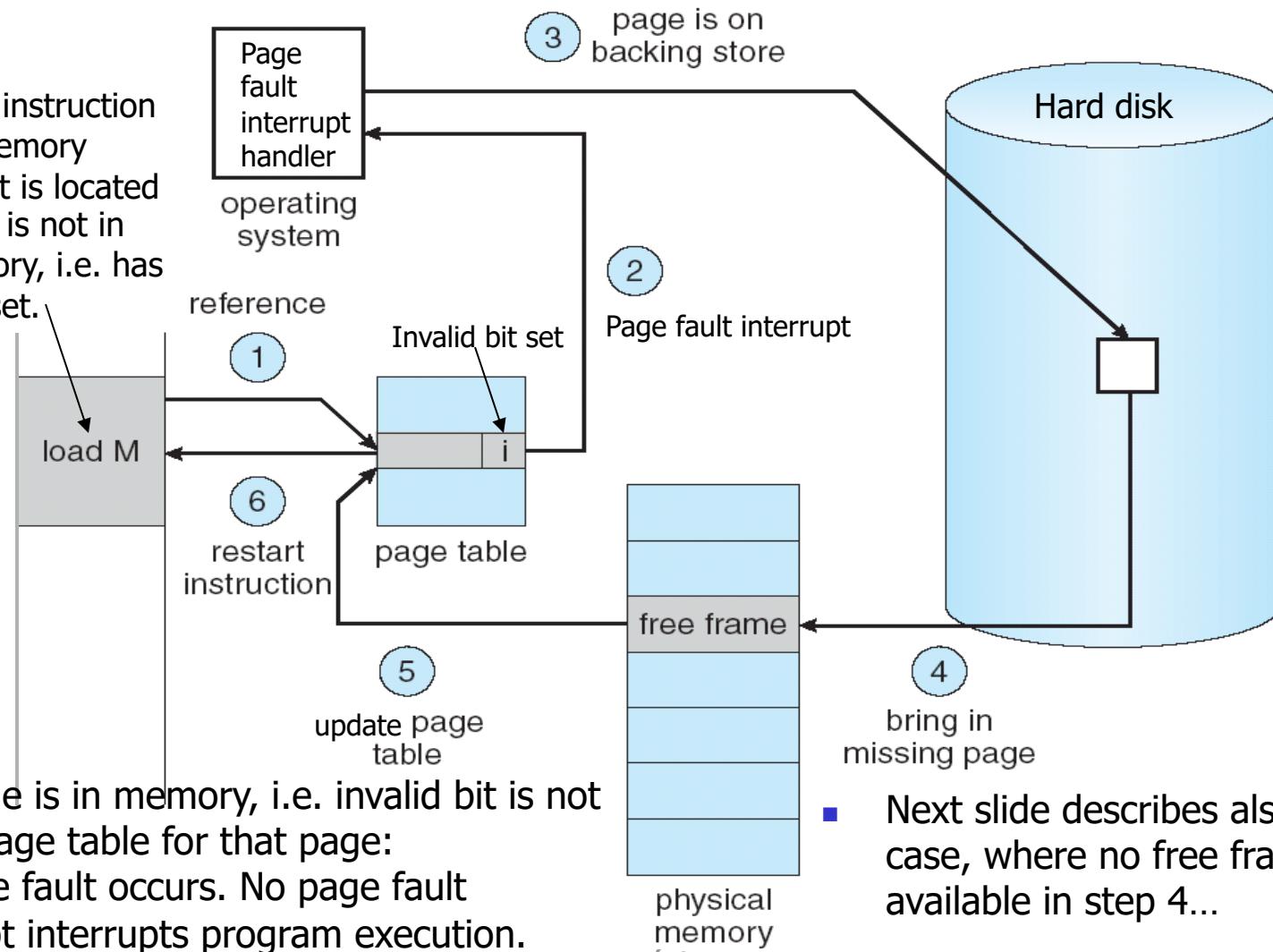
9.2 Virtual Memory Using Demand Paging

- Roughly comparable to idea of swapping; however, instead of whole processes, pages are used:
 - Bringing in/out **single pages**.
 - Do not bring in page unless it is accessed ("lazy paging").
 - Page currently not in physical memory will have in its page table entry the **valid bit set to invalid**: access to page triggers page Fault Interrupt that is serviced by OS and will call pager routine to bring in page.
 - Bring out a page only when physical memory is needed for another new page.
 - At each context switch, page table pointer of PMMU needs to be updated to point to the page table stored in the PCB of the particular process.
- Requires Paged MMU and CPU that supports **restarting** an instruction after a page fault interrupt occurred at exactly the same place and state **where it was interrupted**. (Page fault may occur at any memory access.) → Next slide.



Procedure for Handling a Page Fault (Graphical description)

Machine code instruction referencing memory location M that is located in a page that is not in physical memory, i.e. has its invalid bit set.



- If a page is in memory, i.e. invalid bit is not set in page table for that page:
No page fault occurs. No page fault interrupt interrupts program execution.

- Next slide describes also the case, where no free frame is available in step 4...

Procedure for Handling a Page Fault (Textual description)

- Access to page that is not in physical memory (valid bit of page is set to "invalid"). \Rightarrow Page fault interrupt generated by PMMU.
- Page fault interrupt handler of OS checks internal table (stored in PCB of current process) to determine whether page is invalid because it
 - refers to non allocated memory (illegal access): \Rightarrow process has gone wild: terminate process.
 - refers to a page that is currently on hard disk \Rightarrow needs to be brought in:
 - If a free frame is available in the physical memory:
read page from disk into free frame.
 - If no free frame is available in the physical memory :
 - Select a victim frame that will be replaced with contents of the new page.
 - Was victim frame modified due to a write access (modified bit of page table entry set)?
 - Yes: Save victim frame contents on disk.
 - Read required page from disk into victim frame.
 - Update page table: page is valid now, reset modified bit.
 - Inform scheduler that process is ready and interrupted instruction may be restarted.

Performance of Demand Paging

- Effective Access Time (EAT), if page
 - is in memory: EAT := time of memory access cycle (e.g. 160ns).
 - (Assuming that page table entry is in TLB.)
 - is not in memory: EAT := service page fault interrupt (e.g. 100μs)
 - + write page to disk (if modified) (e.g. 8ms)
 - + read page in from disk (e.g. 8ms)
 - + restart process (e.g. 100μs)
 - + time of memory access cycle (e.g. 160ns)($\Sigma \approx$ e.g. 16ms)
- EAT of page fault is about 100 000 times slower than EAT of a page hit.
 - In practise, the problem is not as severe, because while the process with the page fault is blocked for 16ms, other processes may be ready and execute during the disk access of the pager.
 - But still, good page replacement algorithms needed that take care that only those pages are removed from physical memory (and replaced by other pages) that are unlikely to be used.

9.3 Page Replacement Algorithms

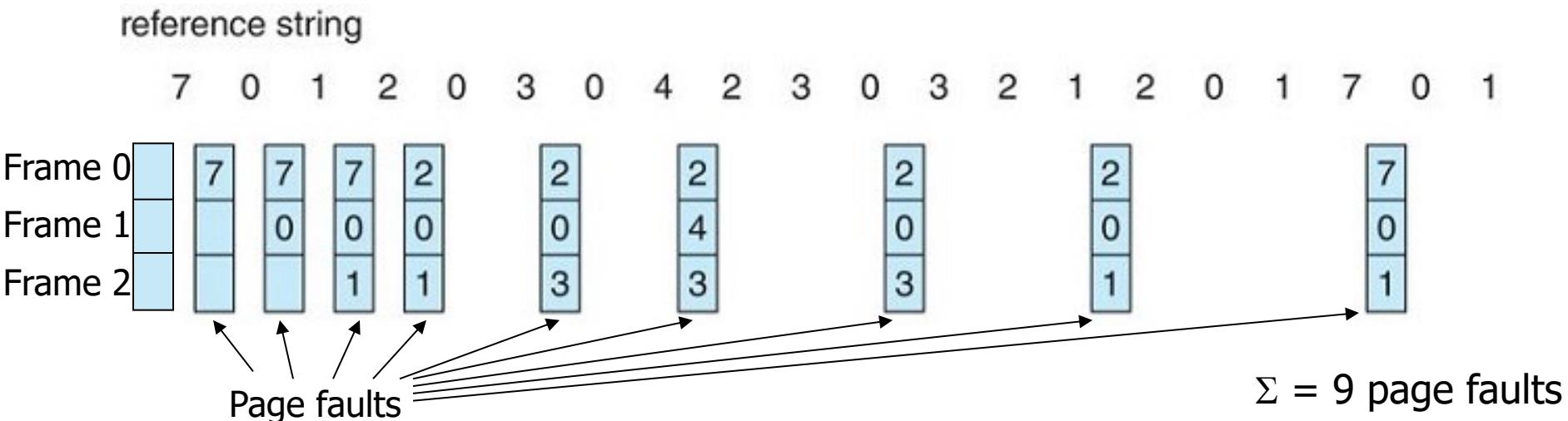
- If no frames are free at a page fault, the OS needs to select a victim frame whose content will be replaced by the content of the requested page.
 - Page replacement algorithms desired that lead to as few page faults (=slow hard disk accesses) as possible.
 - Note: The same algorithms are required in many other areas of computer science where caching is involved, e.g. caches in CPU hardware, caches in software, for storing values that have already been processes before:
 - Updating caches: which cache entry to replace after a cache miss?
- ⇒ Page replacement algorithms relevant for areas outside of operating systems.

Page Replacement Algorithms: Optimal Policy (OPT)

- Algorithm: Replace that frame containing the page that will not be used for the longest period of time in future.
- Guarantees the lowest possible page fault rate!
 - Unfortunately, not implementable: algorithm must be able to predict the future!
 - However, due to undecidability, it cannot be predicted which instructions will be executed in future.
 - Nevertheless, the optimal strategy can be used for comparing and evaluating other page replacement algorithms.
 - Comparison only reasonable when comparing performance for the same fixed string of page references and same number of frames.

Page Replacement Algorithms: Optimal Policy (OPT)

- Example:
 - 3 frames (initially empty) with the following string of page references:



Applies to all page replacement algorithms: the frames are empty in the beginning and as long as empty frames are available, these are taken (and this counts as a page fault because the respective page is not in any of the frames).

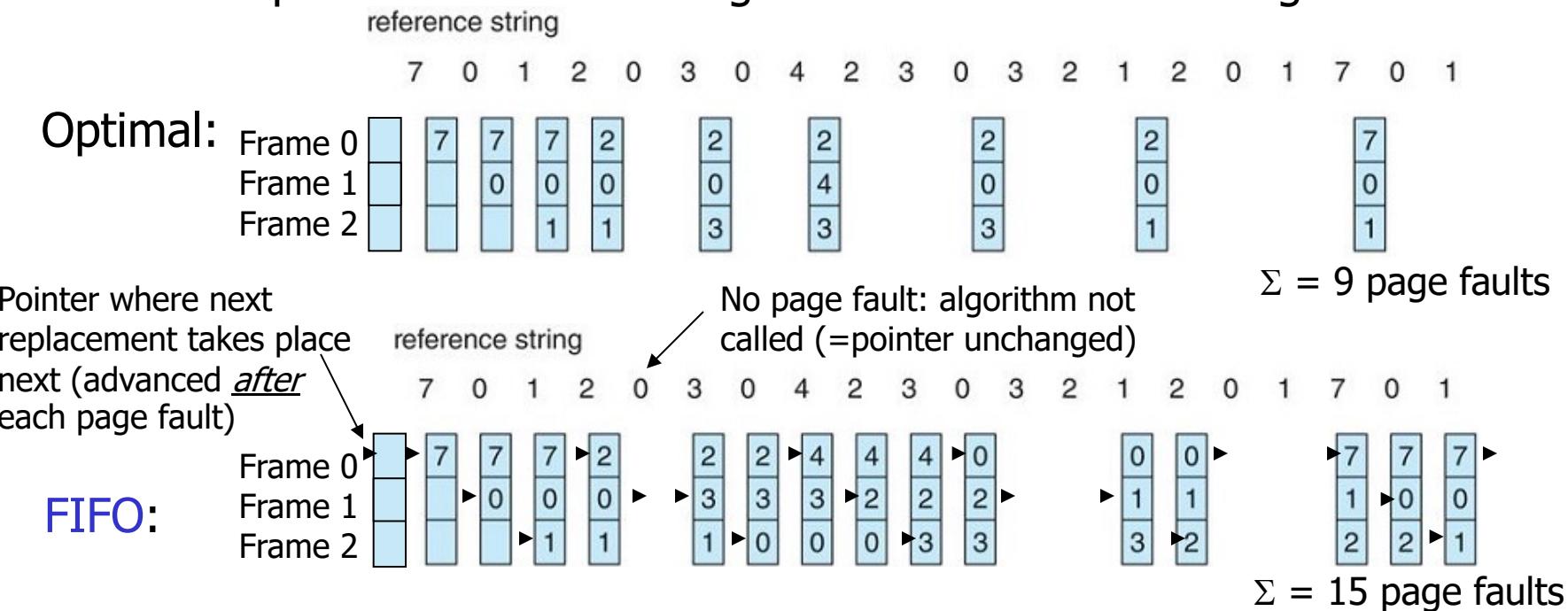
The actual page replacement algorithm is only applied if a page is accessed, but not in memory (i.e. due to a page fault interrupt). Then, a frame needs to be found for that page that is to be brought in.)

Page Replacement Algorithms: First-in, First-out Policy (FIFO)

- As we cannot look into future as required for OPT, maybe start with a simple algorithm that does not require this: FIFO.
- Algorithm: When a page must be replaced, chose that frame containing the oldest page.
 - I.e. replace page that resides for longest time in the set of frames (i.e. page that is “oldest” in terms of residence in memory).
- Advantage:
 - Easy to implement using FIFO queue (size of queue = number of frames). Page at head of queue (=page that is oldest in queue) will be removed to make space for new page.
 - Well, in practise, to consider all the used frames being the FIFO queue would mean to move each time for each and every frame the content of that frame from to the next frame = copying gigabytes of RAM which would be very slow.
 - In practise, a circular buffer as used where frame contents stays in each frame, but rather a pointer to head of queue advances like a clock hand after each page fault.
- Disadvantage:
 - Even though a page is old, it might be the page that is most frequently used, i.e. in this case it would not be wise to replace that page...

Page Replacement Algorithms: First-in, First-out Policy (FIFO)

- Example:
 - Comparison with OPT using the same reference string as before.



Implemented using circular buffer (see also Clock algorithm):

Pointer to head of queue advances like a clock hand after each page fault. I.e. pages stay in their frame instead of shifting them. If pointer reaches last frame, it starts again at first frame (like a clock).

I.Hjörleifsson email: ingolfuh@hi.is. St: 316

Grace, Gróska

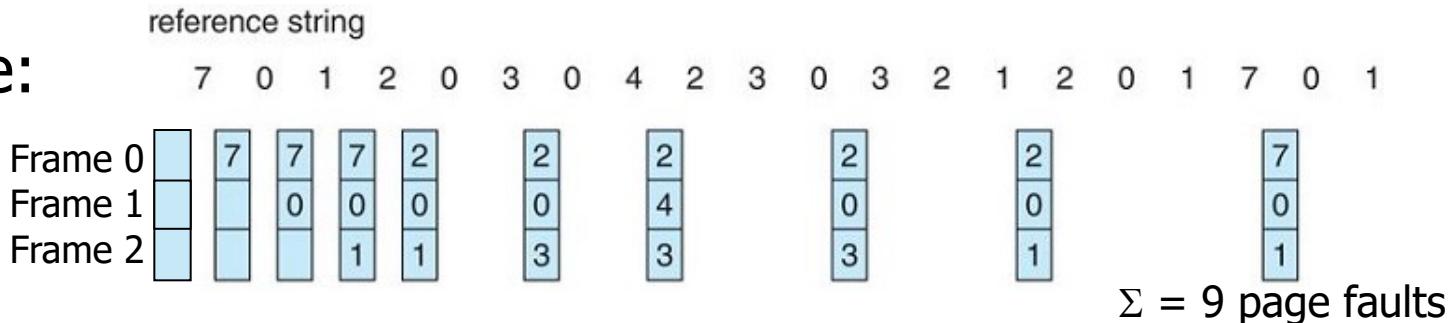
Page Replacement Algorithms: Least-Recently-Used Policy (LRU)

- FIFO is too simple as it does just look at the time when a page was brought in memory instead of considering the time when a page was recently used. We can try to approximate OPT based on the assumption that a page that has been recently used in the past will also be used soon in the future ("locality of execution"): LRU.
- Algorithm: Replace the page that has not been used for the longest period of time (i.e. page that is "oldest" in terms of being referenced).
 - At each reference to a page, the page needs to be timestamped to able to identify the page that is least recently used.
- Advantage:
 - Good approximation of OPT.
- Disadvantage:
 - No PMMU found in a modern CPU supports timestamping a page at each access! I.e. **only implementable with special extra hardware**.

Page Replacement Algorithms: Least-Recently-Used Policy (LRU)

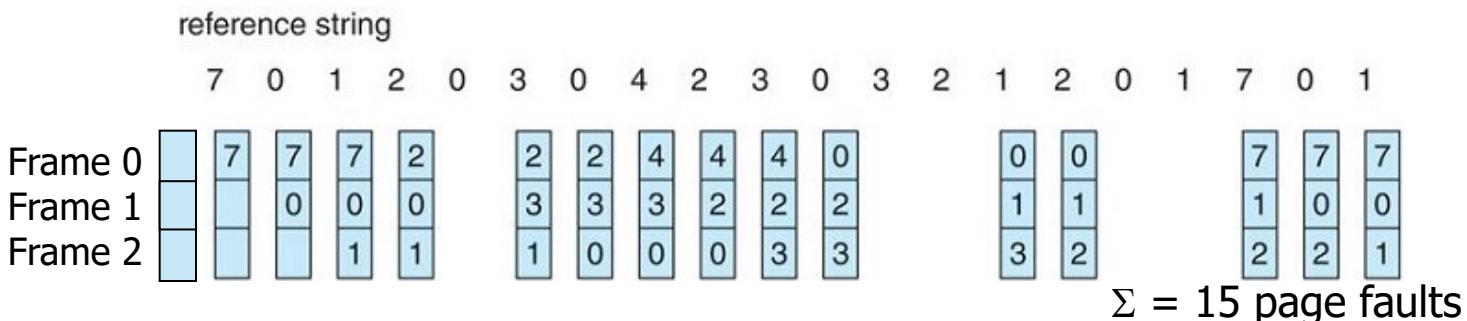
■ Example:

Optimal:



FIFO:

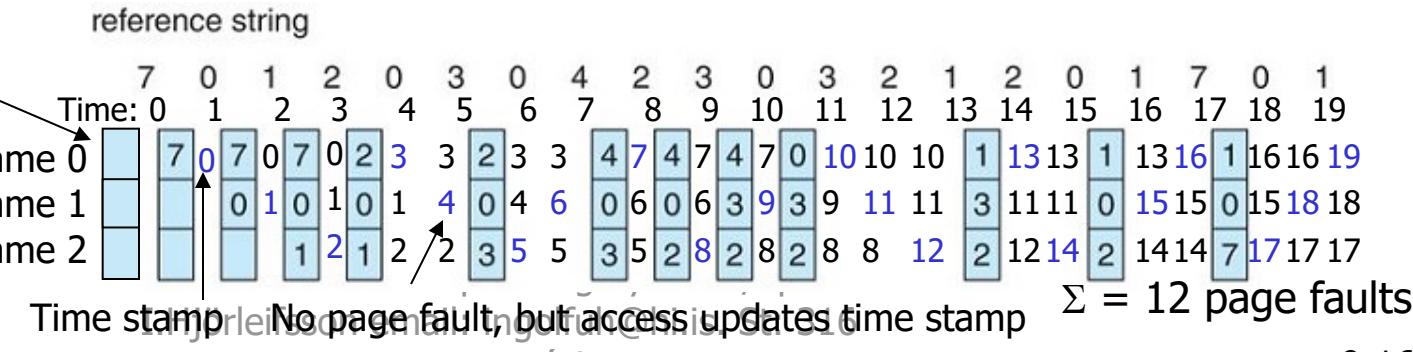
(Implemented
using circular
buffer)



Initially, all frames empty:
any frame can be chosen.

Let's assume first frame
chosen first.

LRU:

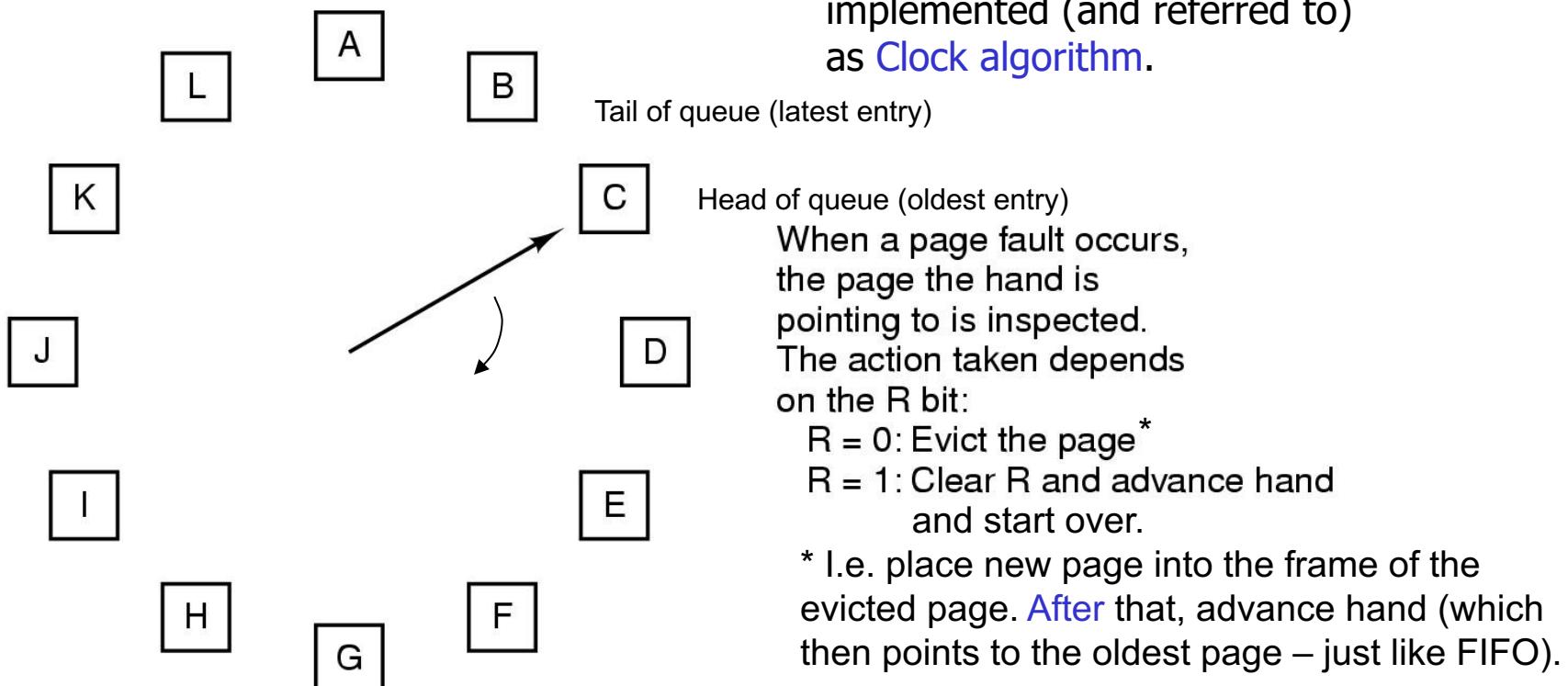


Page Replacement Algorithms: Second-Chance Policy

- LRU seems to be better than FIFO; However, even though not impossible to implement, LRU requires additional hardware. Hence, let's try a mixture of LRU and FIFO, where the oldest page in memory is only replaced if it has not been referenced: Second-Chance policy.
- Algorithm: Inspect (in a FIFO style) the page-table entry of the oldest page (in terms of residence in memory) in memory.
 - If the page table entry's reference bit (→ch. 8: gets set at each read or write access to an address inside that page) is not set: replace this page.
 - (I.e. page is both old and not recently used.)
 - If the reference bit is set: move page from head of queue to tail of queue and reset reference bit of that page. Start over from new head of queue.
 - (I.e. even though page is old, it has been recently used: give it second chance by treating it as new page.)

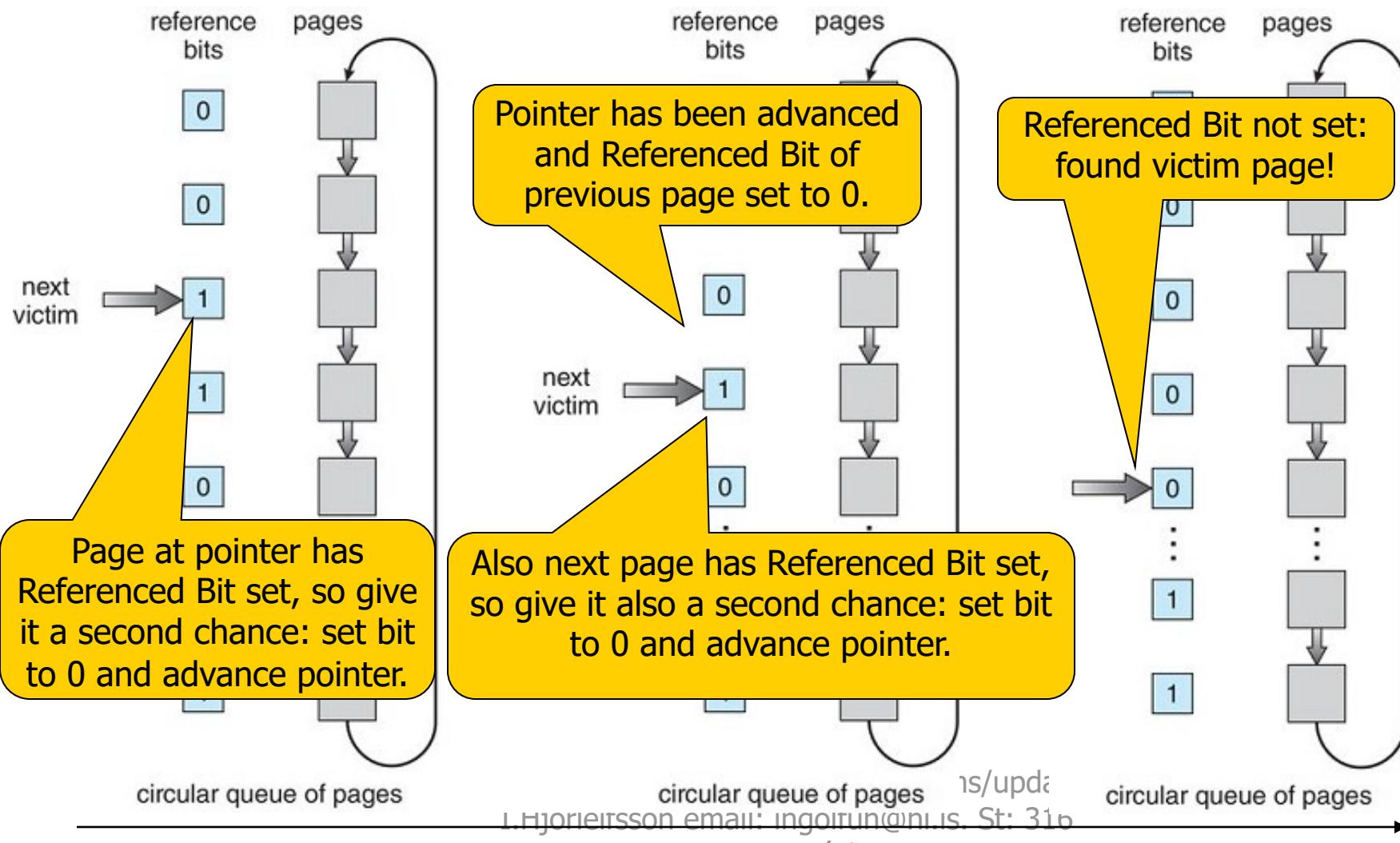
Page Replacement Algorithms: Second-Chance Policy Implemented Using Clock Algorithm

- As we have already seen in the example for FIFO, FIFO-style queues are typically implemented using a circular buffer. When representing a circular buffer graphically, it looks like a clock: \Rightarrow Second-Chance policy is often implemented (and referred to) as **Clock algorithm**.



Note: hand is only advanced in case of page fault: only in this case, interrupt is raised and only then, routine that advances hand can get called by OS interrupt handler.

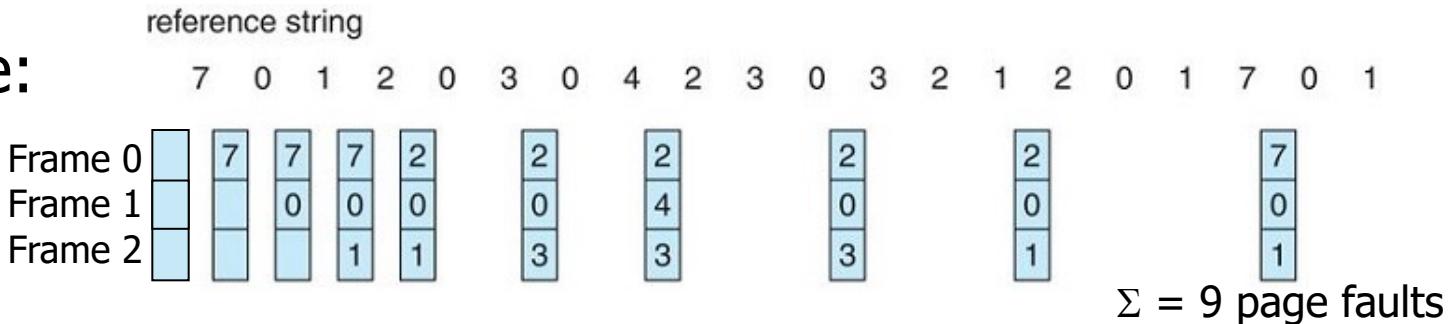
Page Replacement Algorithms: Second-Chance Policy Implemented Using Clock Algorithm: Searching a page to evict



Page Replacement Algorithms: Second-Chance/Clock

■ Example:

Optimal:



FIFO:

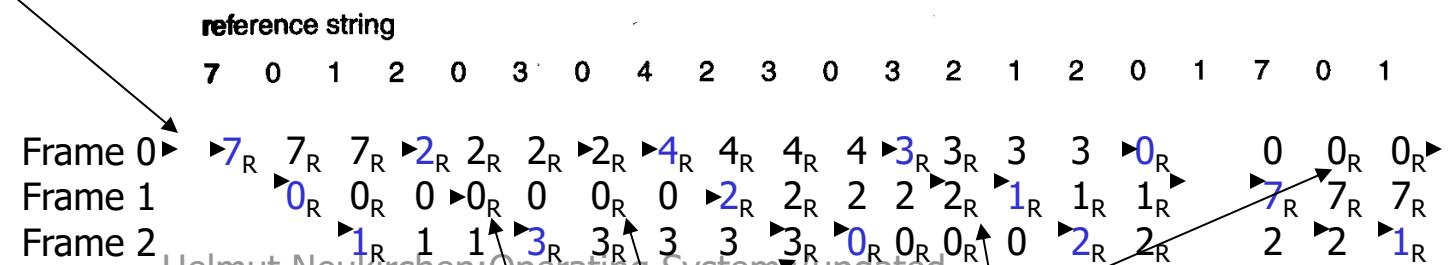
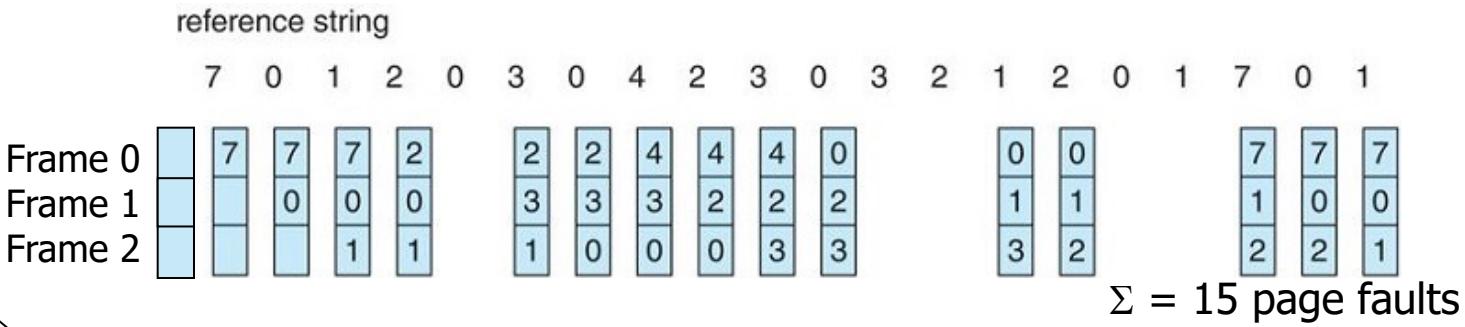
(Implemented
using circular
buffer)

Pointer where *next*
replacement will take
place

Second-
Chance/

Clock:

(_R means:
R bit is set)



Helmut Neukirchen: Operating Systems/updated

No page fault, however R bit of page gets set again by PMMU due to reference.

Page Replacement Algorithms: Second-Chance/Clock

- Advantage:
 - Easy to implement, in particular requires just a feature (reference bit that is set each time a page is either read or modified) that is provided by all modern PMMUs.
 - In fact, the Clock implementation of Second-Chance is used by all major operating systems (or some variants of it)!
- Disadvantage:
 - Just a rough approximation of LRU: Simply “referenced or not referenced” is used as least-recently used criterion. I.e. pages are divided into just two different classes (plus the age information due to the order in the FIFO queue).
- Note: Second-Chance degenerates to FIFO if all pages have their reference bit set.
 - At least, it terminates even in this case, because reference bits are step-by-step reset and finally, the algorithm will inspect the first page again which has now the reference bit reset and will thus be selected as victim.

Page Replacement Algorithms: Enhanced Second-Chance Policy

- Let's try an enhanced Second-Chance policy where at least four different classes of pages are distinguished. Sometimes, this Enhanced Second-Chance policy is also called: Not-Recently-Used (NRU).
- Algorithm: Use a classification of pages based on the reference bit and the modified bit of the page's entry in the page table (R, M):

- (0, 0) not referenced, not modified: best page to replace!
 - (0, 1) not referenced, but modified: not quite as good as page needs to be written to hard disk before replacement.
 - (1, 0) referenced, but not modified: likely to be used in future again.
 - (1, 1) referenced and modified: likely to be used in future again and page would need to be written to hard disk before replacement.
- When having to select a page to replace, go in a clock-style through the circular FIFO buffer and look for the first page that is from class 1: if such a page is found, replace that page. If it is not found (i.e. all the pages of the buffer have been investigated without success), proceed with the next class and start over.
 - In addition, to prevent that after a while all pages have their referenced bit set, the OS does periodically reset the referenced bit of all page table entries. As a result, class 2 may occur (in general, class 2 would not be possible, because when modifying a page, the referenced bit would also be set by the PMMU).

Advance hand like in Second-Chance/Clock, i.e. while searching and after replacing a frame. But R bit gets not reset during search!

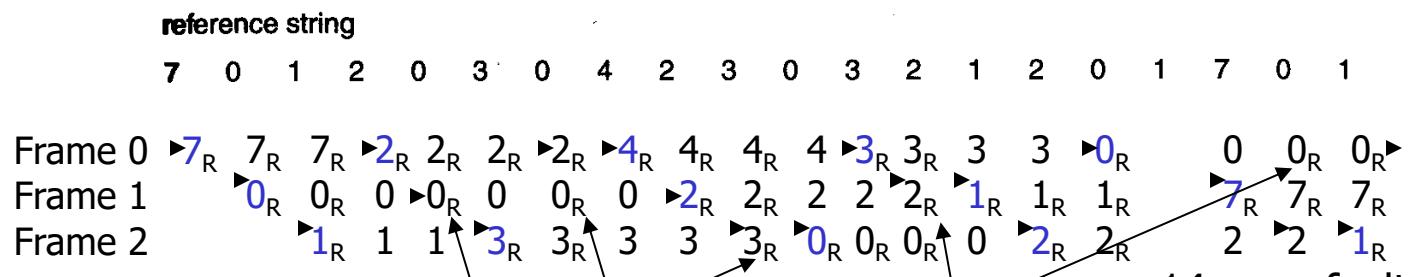
Page Replacement Algorithms: Enhanced Second-Chance Policy

- Example:

Second-Chance/

Clock:

(_R means:
R bit is set)

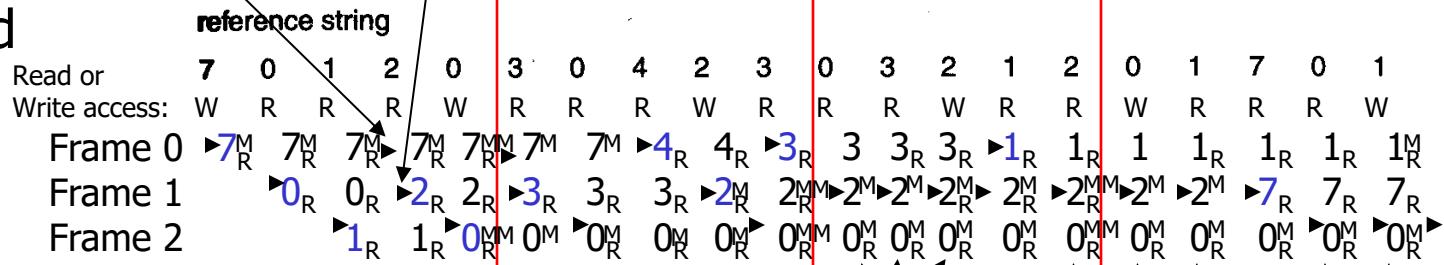


No page fault, however R bit of page gets set again by PMMU due to reference. $\Sigma = 14$ page faults

Search for frame for page 2 starts here, ends there. Reset R bit after, e.g., every 5th step.

Enhanced
Second-Chance

(_R means:
R bit is set,
^M means:
M bit is set
in page table
entry)



No page fault, however R bit of page gets set again by PMMU due to reference. $\Sigma = 11$ page faults

Page Replacement Algorithms: Enhanced Second-Chance Policy

- Advantage:
 - Having four different classes is already a better approximation of LRU than just the two of the non-enhanced Second-Chance policy.
 - Furthermore, it is implementable using just the features (reference bit and modified bit) that are provided by all modern PMMUs.
- Disadvantage:
 - Periodic resetting the reference bits of all page table entries may be too time consuming (in practise, page tables may be pretty huge).
 - This is probably the reason why major OS use rather Second-Chance.
 - Distinguishing four different classes may still not be sufficient for approximating LRU as much as possible. (Improvement: next slide)

Belady's Anomaly

- Paradox: It may be the case that FIFO results in more page faults when you **increase** the number of available frames!
- Example:
 - 3 frames:

	0	1	2	3	0	1	4	4	4	2	3	3
Youngest page	0	1	2	3	0	1	4	4	4	2	3	3
Oldest page	0	1	2	3	0	1	1	1	1	4	2	2
P	P	P	P	P	P	P	P	P	P	P	P	P

9 Page faults

(a)

	0	1	2	3	0	1	4	0	1	2	3	4
Youngest page	0	1	2	3	3	3	4	0	1	2	3	4
Oldest page	0	1	2	2	2	2	3	4	0	1	2	3
P	P	P	P	P	P	P	P	P	P	P	P	P

10 Page faults

(b)

- So called **stack algorithms** do not suffer from Belady's anomaly.
 - Stack algorithm: set of pages in memory for n frames is always a subset of the set of pages for $n+1$ frames.
 - (OPT and LRU are **stack algorithms**.)

9.6 Memory Compression Instead of Paging Out to Storage Device

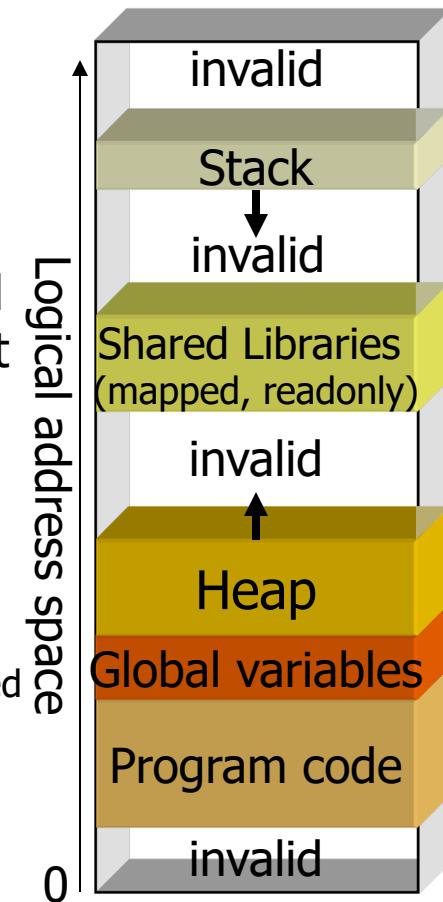
- Mobile devices typically do not page out pages to a storage device.
 - Still, they may run out of physical memory. Approaches to deal with:
 - **Terminate processes** (but allow a process to store its status, so that it can later be re-started).
 - **Compress memory:**
 - For those pages that would be candidates for paging out to storage device:
 - Compress contents of these pages using compression algorithm.
 - Similar to ZIP, for file compression. E.g. Microsoft's Xpress and Apple's WKdm: reduction to 30%-50% of original page size.
 - In average: 2-3 compressed pages fit into one frame, making thus 1-2 frames available.
 - While compression needs CPU time, typically still faster than paging out to SSD storage.
 - Even non-mobile OSes use nowadays memory compression.
 - Only if memory compression is not enough, page out to storage device.

9.7 Advanced Applications of Virtual Memory: Demand Loading vs. Prepaging

- As the infrastructure of demand paging is anyway available, **demand loading** becomes possible:
 - When a process is started, do not load whole binary file containing instructions into memory, but just mark initially all pages as invalid.
 - At a page fault, load the according instructions for that page into memory.
 - Advantage: no unnecessary loading of instructions that might never get executed.
 - Disadvantage: resulting page faults lead to an overhead.
- **Prepaging** (just the opposite of demand loading):
 - Load all instructions into memory to avoid page faults.
 - Advantage: reduced number of page faults.
 - Disadvantage: unnecessary loading of instructions may occur.
- In practise, a compromise is used, i.e. the first n pages are prepaged and at a page fault, **multiple consecutive pages** (e.g. current working set size) are loaded

Advanced Applications of Virtual Memory: Growing Heap & Stack

- Without virtual memory, reserving the right amount of memory for stack and heap is difficult:
 - Too small: stack or heap overflow possible,
 - Too huge: memory is wasted.
- With virtual memory, we can just reserve the whole logical address space for a process and reserve a big amount of it for stack and heap.
 - While sufficient space for stack and heap is reserved in the logical address space, use only as much physical frames as currently required.
 - As stack and heap increase, just more pages are actually used.
 - Overwriting shared libraries by stack or heap impossible as shared libraries serving as a sentinel or buffer in-between are read-only.
- Management of heap space:
 - While paging avoids external fragmentation, internal fragmentation may still occur within the heap of a process: when releasing allocated heap space, holes in the logical address space of the heap occur.



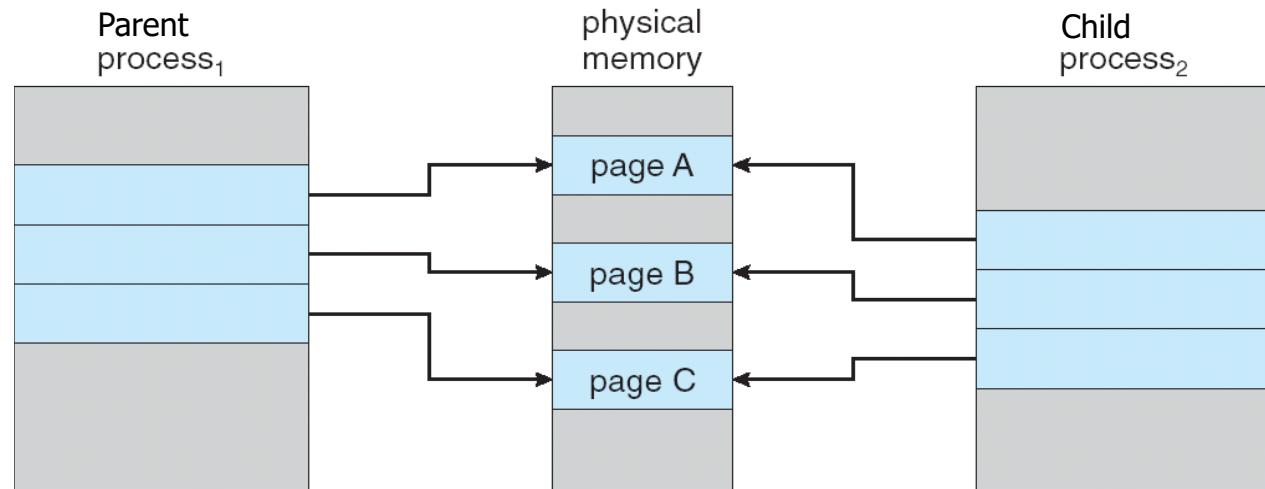
Advanced Applications of Virtual Memory: Fork Using Copy-on-Write/Lazy-Copy

- Reminder: at **fork**, the child process gets an exact copy of the address space of the parent.
 - Side note: This becomes only possible using a programmable MMU:
 - The child's copy will be located at a different physical address. However, the child will also get a copy of all the parent's address references. These remain only valid, if a programmable MMU can be used to map the different physical addresses of the copy for the child to the same logical addresses that the parent process used.
- However, copying the physical memory of the parent is slow.
- ⇒ Faster: just map frames (instead of copying) of parent containing instructions and data into address space of child (=shared memory).
 - However, when parent or child modifies its address space, the copy of the other party must not be modified!
 - ⇒ Mark shared pages as write-protected in page table entry: as soon as parent or child modify data, page fault interrupt occurs. Only then, just these frames are physically copied. ("Copy-on-Write"/"Lazy-Copy")

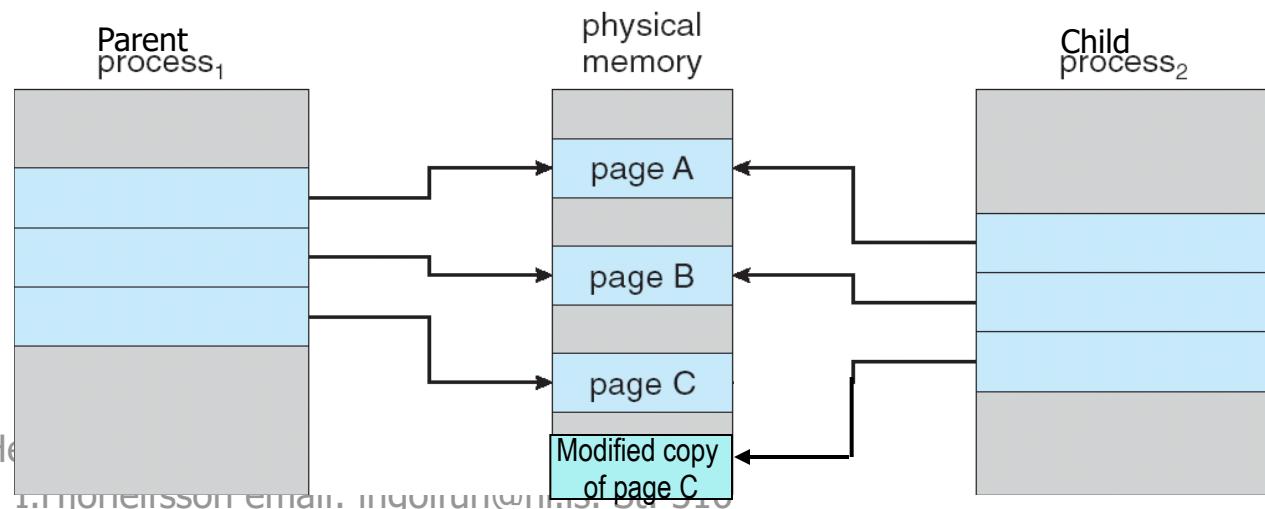
Advanced Applications of Virtual Memory: Fork Using Copy-on-Write/Lazy-Copy

■ Example:

- Initially, pages are shared between child and parent process.



- After child (or parent) tries to modify page C, a copy of page C is created and the modifying CPU instruction is restarted.



9.9 Summary

- Virtual memory allows to execute processes whose logical address space is larger than physically available memory.
 - Allows to run extremely large processes and to increase degree of multiprogramming beyond physically available memory.
 - Typically implemented using demand paging.
 - Single pages are transferred between physical memory and hard disk.
 - Significantly faster than swapping of whole processes.
 - Still system may be busy doing disk transfers when thrashing occurs.
 - Different page replacement algorithms perform differently.
 - While the Optimal policy cannot be implemented and others involve too much overhead, Second-Chance is used in today's operating systems.
 - Instead of paging out to storage device: try to compress pages.
 - From speed point of view: have enough physical RAM to avoid paging!
 - Virtual memory enables other advanced applications, e.g. fast forking using copy-on-write/lazy-copy.
- Kernel memory managed separately: buddy system, slab allocation.

Course
TÖL401G: Stýrikerfi /
Operating Systems
10. File-System Interface

Chapter Objectives

- Explain the function of file systems.
- Describe the interfaces to file systems.
- Understand memory-mapped files.
- Discuss file-system design tradeoffs, including access methods, file sharing, file locking, and directory structures.
- Explore briefly file-system protection.

Contents

1. Introduction: File Concept
2. Operations for Accessing Files
3. Concept & Structure of Directories
4. File-System Mounting
5. Shared File Access and Protection
6. Summary

Note for users of the Silberschatz et al. book: newer editions have the material ordered differently than on my slides.

Helmut Neukirchen: Operating Systems/updated
I.Hörleifsson email:ingolfuh@hi.is st.316 Grace,
Gróska

10.1 Introduction: File Concept

- For storing data, an application program or its developer should not have to deal with physical details of a storage device.
 - Chaos would occur if all the applications would write directly to a storage device: application A would not be aware of application B's data and might just overwrite it.
 - Organised access to data on a storage device required.

⇒ Operating systems offer support for **files**.

- Store and access of data using **symbolic names**.
 - Application needs not to know about internal physical organisation of files on storage device – just use the file name.

Helmut Neukirchen: Operating Systems/updated

I.Hörleifsson email:ingolfuh@hi.is st.316 Grace,
Gróska

File Concept: File Attributes

- In addition to the actual file contents, **file attributes** are required for managing files:
 - Name (Many Unixes: 255 character maximum length),
 - Location where file contents is stored on storage device,
 - Size (in bytes)
- Additional (typically used, but not necessarily required) file attributes:
 - Date (e.g. file creation; last write, last read access),
 - Owning user, owning group of users,
 - Protection information.

10.2 Operations for Accessing Files

- POSIX operating systems:

- Open an existing file:

```
int open(const char *pathname, int flags)
```

- Create new file/Overwrite an existing file and open it in write mode:

```
int creat(const char *pathname, mode_t mode)
```

- Return value: File descriptor, -1=error
 - pathname: Name of file to open
 - flags: e.g.: O_RDONLY/O_WRONLY/O_RDWR=for read only/write only/read and write access, O_APPEND=for appending at the file end.
 - mode: Access rights for the file to be created

- Close opened file:

```
int close(int fd)
```

- Return value: -1=error
 - fd:

Helmut Hölzl, Operating System Update

I. Hörlifsson email:ingolfuh@hi.is st.316 Grace,

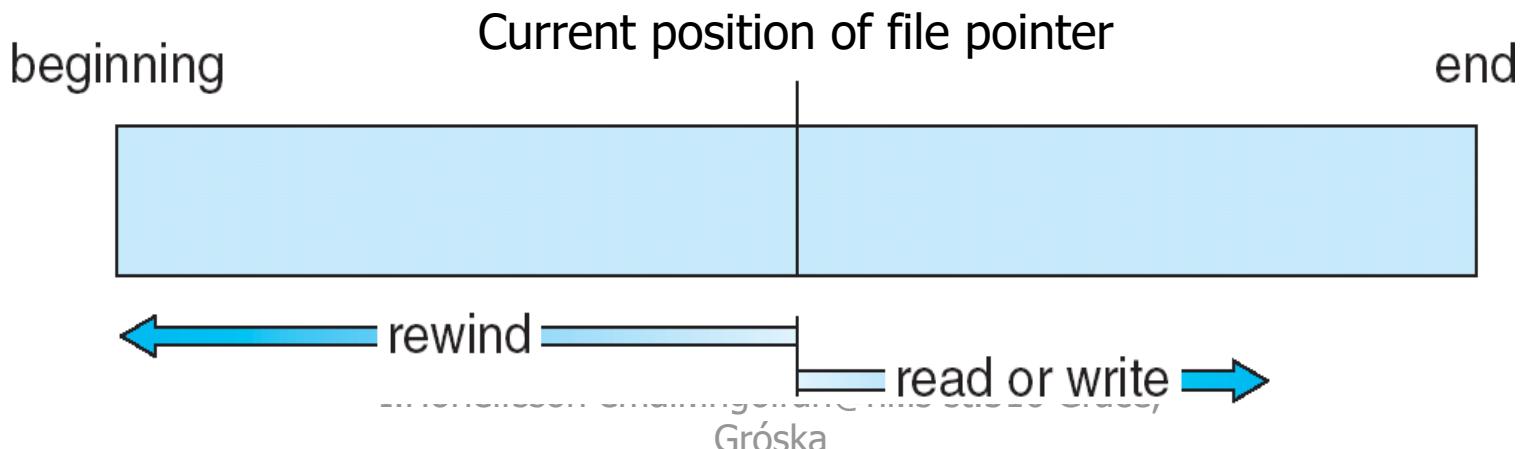
Gróska

Excursion: File Descriptor

- A **file descriptor (FD)** is a **number that uniquely identifies an opened file**. The FD is generated by the OS when opening/creating a file.
 - When opening/creating file, it is checked whether file exists, where it is stored, who is permitted to access, etc. If everything is OK, FD is generated.
- Most file operations require a valid file descriptor.
 - If no FD would be used, but instead a file name would be passed to each file operation, the OS would have to check at each file operation, whether the file exists, where it is located in the file system, ...
 - Using the FD approach, the OS just has to check this once when opening/creating a file.

File Pointer

- When reading/writing bytes from/to a file, the bytes are read/written with respect to an offset in the file: the **file pointer**.
- When a file is opened, file pointer is set to 0.
 - (When opened with as `o_APPEND`, file pointer is set to end of file.)
- Reading/writing advances file pointer (sequential access).**
- In addition, “rewind”/“fast forward” is possible to move file pointer without reading/writing (**direct access/random access**).
- Each process has it's own file pointer for each opened file.



Operations for Accessing Files (2)

- POSIX operating systems:
 - Read data starting from current file pointer position:
`ssize_t read(int fd, void *buf, size_t count)`
 - Write data starting from current file pointer position:
`ssize_t write(int fd, const void *buf, size_t count)`
 - Return value: Number of actually transferred bytes, -1=error
 - fd: File descriptor
 - buf: Address in main memory where bytes should read into/write from
 - count: Number of bytes to be transferred
 - Move file pointer: `off_t lseek(int fd, off_t offset, int whence)`
 - Return value: New (absolute) file pointer position, -1=error
 - fd: File descriptor
 - offset: New position for file pointer (relative or absolute according to whence)
 - whence: SEEK_SET=absolute, SEEK_CUR=relative with respect to current file pointer, SEEK_END=relative with respect to end of file (i.e.using negative offset)
 - Further operations: `truncate`, `rename`, `unlink` (delete file or directory).

Memory-mapped Files

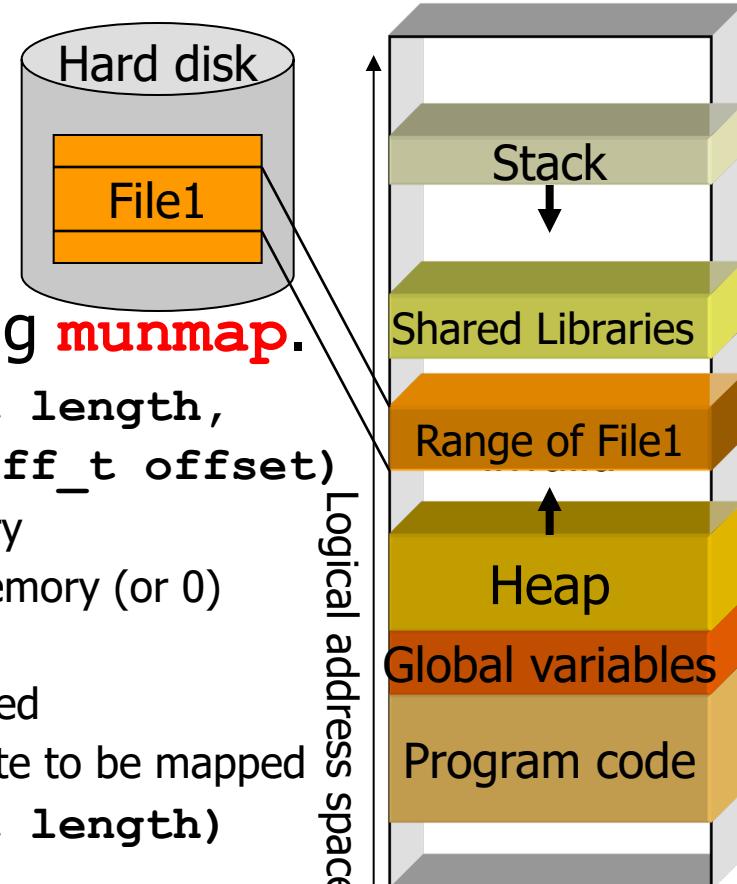
- If a huge file shall be accessed, it may be more convenient and faster to map the file contents into memory:
 - File contents can be accessed via ordinary memory accesses (e.g. using pointers).
 - No need to use `read()`/`write()` system calls (system call=slow).
 - Demand paging/lazy loading is internally used: only if a block of a file (corresponding to a page in memory) is actually accessed, it is loaded from the file system into memory.
 - Initially, a page corresponding to a file block is marked as invalid,
 - An access to that page will trigger a page fault interrupt,
 - Interrupt service routine will load block into frame,
 - Page table is updated: page marked as valid and points to frame,
 - Now, access to page is possible.

Memory-mapped Files (POSIX)

1. Map a range of file into the logical address space (**mmap**).
2. Modify memory (or just read).
3. Write modified memory back using **munmap**.

- `void* mmap(void *start, size_t length,
int prot, int flags, int fd, off_t offset)`
 - **return value:** Address of mapped memory
 - **start:** Desired start address in main memory (or 0)
 - **length:** Size of range to be mapped
 - **fd:** File descriptor of file to be mapped
 - **offset:** Position in file containing first byte to be mapped
- `int munmap(void *start, size_t length)`
 - **return value:** 0=OK, -1=error
 - **start:** Start address of range to be written back
 - **length:** Size of range to be written back

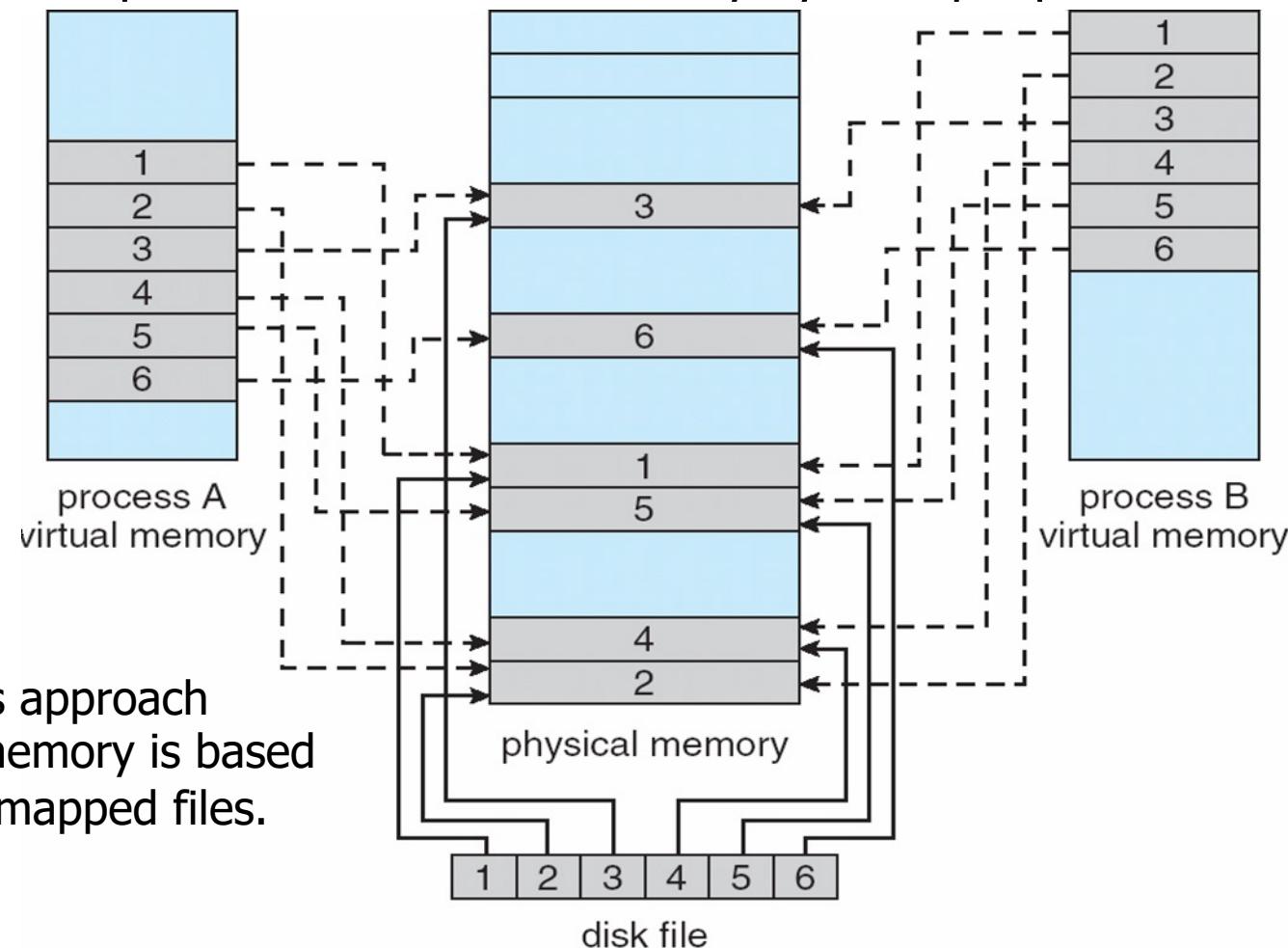
Memory-mapped files also supported by Java API!



Memory-mapped Files and Shared Memory

- Even possible to map the same file concurrently by multiple processes

⇒ shared
memory
between
processes.



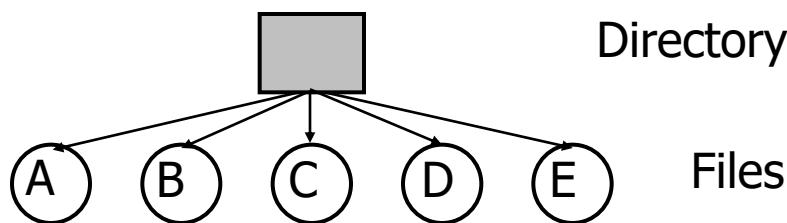
- MS Windows approach for shared memory is based on memory-mapped files.

Memory-mapped Files: Discussion

- You can of course as well load a huge file into memory using `read()`, modify it, and write it back using `write()`.
- However, memory-mapped files allow the OS to **load only those pages from a file that are actually accessed.**
 - ➔ Suitable for processing huge files where you do not know in advance which locations you actually access.
 - Only the actually accessed locations need to be loaded from the (slow) mass-storage device,
 - Only the modified (using PMMU modified bit) locations need to be written to the (slow) mass-storage device.
 - In fact, an operating system typically does demand loading of executable binary files (→9-36) by using internally the memory-mapped file concept.

10.3 Concept & Structure of Directories

- Files are managed in directories:
 - Flat (=single level) directory structure (outdated): Directory contains only files.

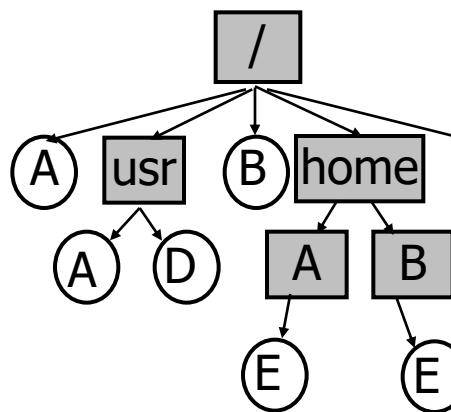


- Disadvantage: Directory gets cluttered in presence of many files
⇒ Hierarchical tree-like directory structure!

Hierarchical Directory Structure

- Directory contains files and sub-directories.
 - Directory tree starting from root directory:
 - Sub-directories can be used for structuring.
 - Files in different directories are independent from each other and may even have the same name.
 - Path navigates to file:
 - “/” as separator of directory levels.

Root directory



- “/” at the beginning of path refers to the root directory, i.e. a path beginning with “/” is an absolute path.
- If no “/” is at the beginning of path, this is a relative path, that refers to the current directory of a process.
(Each process has a current directory as part of its context, i.e. this information is stored in the PCB.)

Helmut Neukirchen: Operating Systems/updated

Tobias Hüttemann email:ingolfuh@hi.is st.316 Grace,

Gróska

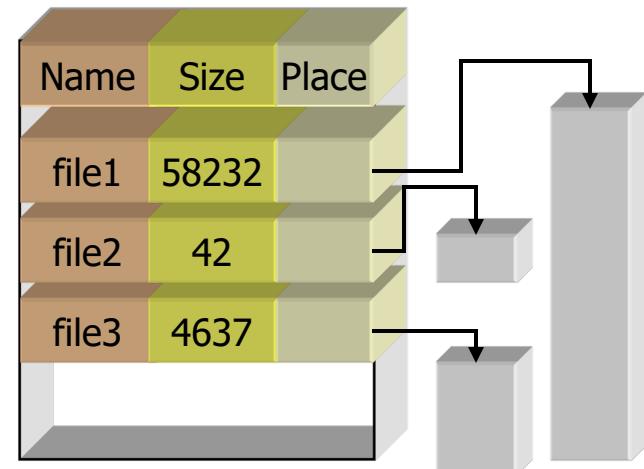
Directories are just Files

- Using hierarchical directory structures, a (sub-)directory is just another file that is contained in a directory.
- However, what is the contents of such a directory file?

→ File attributes of all files stored in that directory!

- At least stored in directory entry:
 - Name of file,
 - Location of file in file system (e.g. sector or block number on storage device).

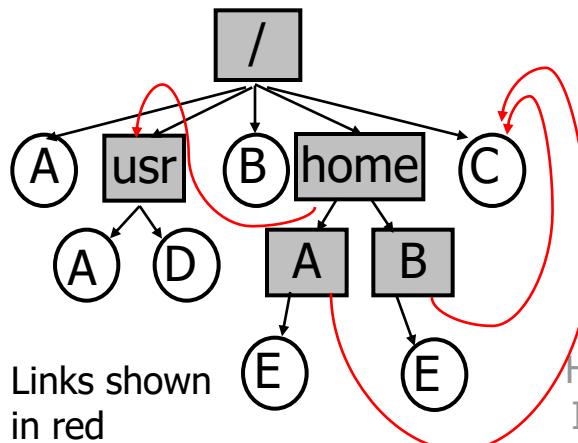
- Further attributes (e.g. file size or information that a file is a directory):
 - Either stored as well in directory file
 - or stored at location of file in file system (enables hard links → 10-17).



Directory with Links

- Adding **links** within directory tree:
acyclic tree → acyclic graph (or even cyclic graph):
 - One file may be referenced in different directories or even using different file names.
 - **Special directory entries** that are in fact links:
 - “.” references the directory itself, i.e.
“.” is a link to the file containing the respective directory.
 - “..” references the parent directory, i.e.
“..” is a link to the file containing the parent directory.

Root directory



The contents of file /C can also be accessed as /home/A/C and /home/B/C; the directory /usr also as /home/usr.

Example usage of links: keep different versions of a file and switch between them by letting the link point to the file that shall be used; see example on slide 10-20.

Helmut Neukirchen: Operating Systems/updated
I.Horlersson email:inguru@hi.is st.310 Grace,
Gróska

Hard Links

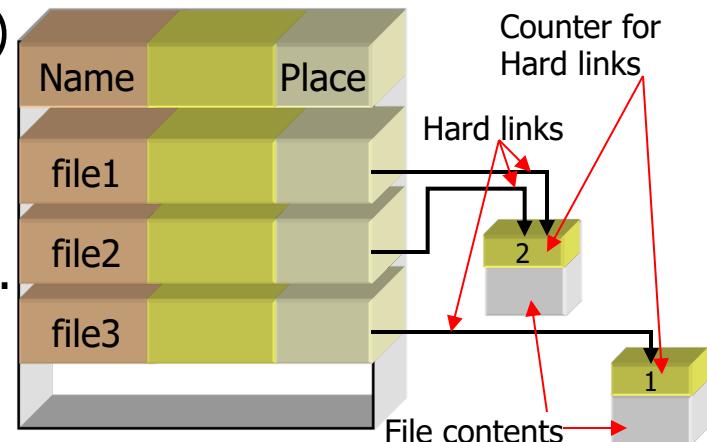
- **Hard link:** Link target is original file, i.e. a pointer to the original file's location (=block number). ⇒ It cannot be distinguished which of the directory entries contains the primary entry and which one the hard link. (In fact, both entries are hard links.)

- Calling a delete command on a hard link would delete the original file (& the hard link entry in the directory) ⇒ all further hard links would point to non-existing file.

■ Solution:

This is the reason, why the POSIX system call for deleting a file has as name **unlink**

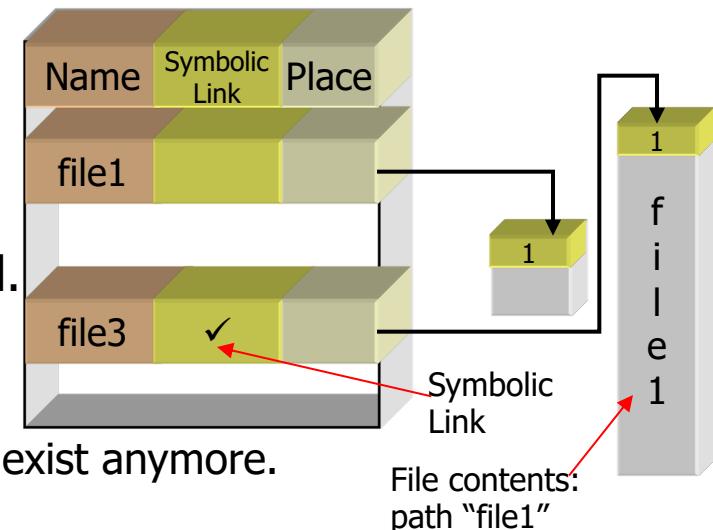
- Each file has a counter: how many hard links are currently pointing to that file?
- When deleting, decrement hard link counter and remove directory entry, but do not remove file itself.
- Only after last reference to a file has been deleted, the file itself is deleted.
- Disadvantage: file location pointer only possible within same file system; file attributes other than name & place must be stored at file location.



Also known as
"Shortcut" on
MS Windows.

Symbolic Links

- **Symbolic link:** Link target is a special file that contains a path name pointing to the original file. (The fact that the file is a special file, i.e. containing a path must be flagged by a special file attribute.) \Rightarrow It can be distinguished which directory entry is the primary entry for the file, and which directory entry is just the link.
 - Calling a delete command on a symbolic link just deletes the special file containing the path name (and the symbolic link entry in the directory). Original file is not affected.
 - Disadvantage:
 - Deleting the original file, leaves a dangling symbolic link pointing to a file that does not exist anymore.
 - Advantage:
 - Paths possible that link to files that are located on a different file systems.
 - File attributes (e.g. size) may be stored in directory entry (instead of storing file attributes at file location like the hard link counter).



Hard links/Symlinks on POSIX

■ Example: Output of `ls -laF /boot`

"File contains directory entries" flag

Hard link pointing to current directory

drwxr-xr-x	3	root	root	4096	Nov 21	16:05	.	Hard-Link pointing to parent directory
drwxr-xr-x	24	root	root	4096	Nov 13	10:10	..	Hard-Link pointing to parent directory
lrwxrwxrwx	1	root	root	21	Sep 8	18:09	System.map -> System.map-3.4.22	
-rw-r--r--	1	root	root	516858	Mar 26	2018	System.map-3.4.20	
-rw-r--r--	1	root	root	544356	Jun 25	2018	System.map-3.4.21	
-rw-r--r--	1	root	root	591812	Nov 21	16:05	System.map-3.4.22	
-rw-r--r--	1	root	root	512	Jun 12	2018	boot.0300	
-rw-r--r--	1	root	root	36044	Mar 26	2018	config-3.4.20	
-rw-r--r--	1	root	root	34896	Jun 25	2018	config-3.4.21	
-rw-r--r--	1	root	root	27784	Nov 21	16:04	config-3.4.22	
drwxr-xr-x	2	root	root	4096	Nov 21	16:46	grub/	
-rw-r--r--	1	root	root	147086	Jun 12	2018	initrd.gz	
-rw-----	1	root	root	54272	Sep 8	19:23	map	
lrwxrwxrwx	1	root	root	18	Nov 21	16:05	vmlinuz -> vmlinuz-3.4.22	Target of symlink
-rw-r--r--	1	root	root	925720	Mar 26	2018	vmlinuz-3.4.20	
-rw-r--r--	1	root	root	941965	Jun 25	2018	vmlinuz-3.4.21	
-rw-r--r--	1	root	root	1220912	Nov 21	16:05	vmlinuz-3.4.22	

"File contains
symlink path" flag

Hard link counter

Size of directory file always a multiple of the block size (typically 4096)

I.Hörleifsson email:ingolfuh@hi.is st.316 Grace,

Gróska

Directory Operations

- Unix-based operating systems:
 - Change current working directory of process (used by relative pathnames):
 - `int chdir(const char *pathname)`
 - Create/remove directory:
 - `int mkdir(const char *pathname, mode_t mode)`
 - `int rmdir(const char *pathname)`
 - Create hard link/symbolic link, dereference symbolic link:
 - `int link(const char *oldpath, const char *newpath)`
 - `int symlink(const char *oldpath, const char *newpath)`
 - `int readlink(const char *path, char *buf, size_t bufsiz)`
 - Read directory entries (`DIR`=directory descriptor incl. pointer to current dir. entry):
 - `DIR *opendir(const char *pathname)` Open directory
 - `int closedir(DIR *dir)` Close directory
 - `struct dirent *readdir(DIR *dir)` Retrieve current directory entry and advance pointer to next entry
 - `void rewinddir(DIR *dir)` Reset pointer to current dir. entry
 - ...

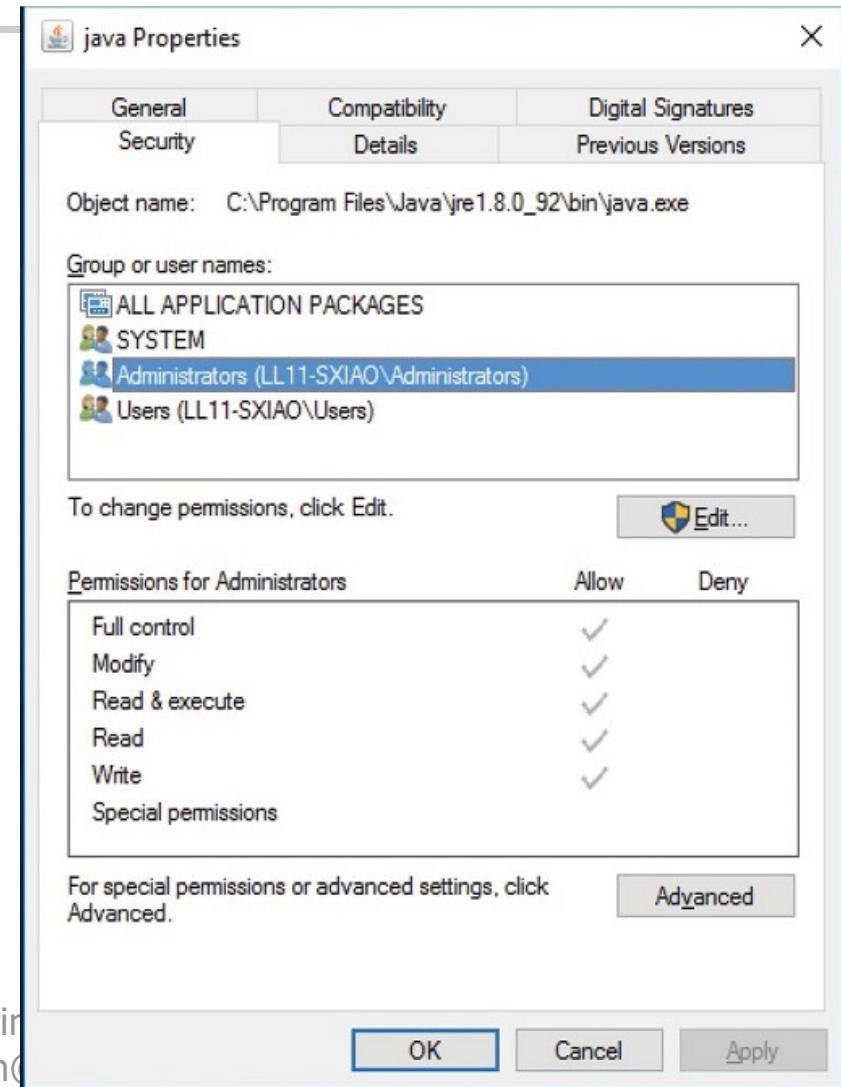
10.5 Shared File Access and Protection

- An OS has to support users to **share files**, e.g.
 - System/program files (to avoid duplicating them for each and every user),
 - Application data (to allow users to collaborate).
- but also to **protect files** from being accessed by unauthorised users.
- To manage sharing/protection on POSIX systems, for each file, the access rights
 - **read/write/execute** can be granted to
 - **owner/user** (user that created the file),
 - **group** (each file has also a group associated: by default, the group to which the owner belongs. Groups are defined by the system administrator),
 - **all** (any user on that system).
- **chown** and **chgrp** commands can be used to change owner/group.
- **chmod** command can be used to change access rights, e.g.
 - **chmod g+w myfile.txt** to give write permission to the group.

-rw-r--r-- helmut hi myfile.txt
All may read.
Members of group **hi** may read.
Owner **helmut** may read/write.

Access Control Lists

- MS Windows NTFS file system and some advanced Unix file systems use **Access Control Lists (ACL)** to allow to specify access rights individually to different users (or groups).
 - No need to have the system administrator define fixed groups for you.
 - Instead, the owner of a file can just add new users to the ACL and set permissions accordingly.
 - Incompatible with POSIX tools that assume the **rwx** user/group/all approach.



10.6 Summary

- Operating system provides file concept.
 - Applications just use file name and access operations to read/write data.
 - Applications do not need to deal with file system details.
 - Contents of file depends on application.
- Directories contain a mapping from file name to location in file system.
 - Nowadays used: hierarchical directories with at least symbolic links.
- In this chapter, only the interface for accessing files and directories have been presented.
 - Possible implementations of file systems are discussed in the next chapter.

Course
TÖL401G: Stýrikerfi /
Operating Systems
11. File-System Implementation

Chapter Objectives

- Describe the details of implementing local file systems and directory structures.
- Discuss block allocation methods and free-space management.
- Explore file system efficiency and performance issues.
- Look at recovery from file system failures.

Contents

1. Introduction: File-System Structure
2. File-System Implementation
3. Allocation Methods
4. Free-Space Management
5. Directory Implementation
6. Efficiency and Performance
7. Recovery
8. Summary

Note for users of the Silberschatz et al. book:
these slides differ slightly from their chapter 11

Helmut Neukirchen: Operating Systems/updated
I.Hjörleifsson email:ingolfuh@hi.is st. 316 Grace,
Gróska

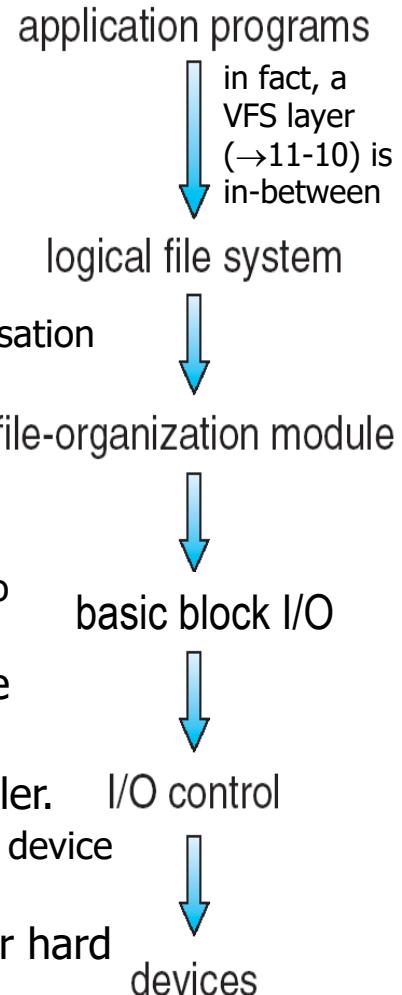
11.1 Introduction: File-System Structure

- File system store files (incl. directories) on secondary storage devices:
 - Mainly: Hard disk & flash memory/solid state drive (SSD), etc.
- Storage area of file system is divided into logical storage units:
 - Unix: **blocks** (typical size: 4KB), MS Windows: **clusters** (different sizes used).
 - In the following, the term "block" may also refer to "clusters".
 - If file is longer than a block, multiple of them are needed for storing the file contents.
 - If file size is not a multiple of block size, the last block is only partially used (**internal fragmentation**).
- In addition, some **metadata** (= data in addition to actual file contents) is required and stored as part of the file system on the storage device: information about
 - **which blocks on the storage device are allocated/free** (→free space management).
 - **in which blocks is the contents of each file stored** (→allocation methods).



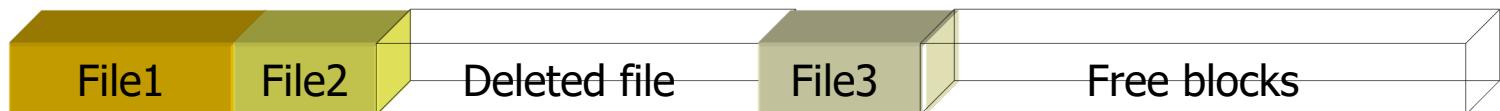
File-System Layers

- **Logical file system:** manages file pointer, metadata (e.g. owner, permissions, directory contents), symbolic links.
 - As directories are just some special kind of file, the underlying file-organisation module can be used to read/write content of a directory as a file.
- **File-organisation module:** manages in which blocks of the device the file contents is stored (→allocation methods) and where free blocks can be found on the device (→free space management).
 - Translates request from logical file system ("read first block of file x") into requests for basic block I/O and I/O control ("load block 123 from disk").
- **Basic block I/O:** e.g. handles buffers for blocks that are waiting to be transferred to/from storage device, caches blocks.
- **I/O control:** device driver that is specific to underlying device controller.
 - Translates command from block I/O layer ("load block number 123") into device specific hardware commands.
- **Device:** actual I/O device and controller (e.g. Serial ATA controller for hard disk, USB controller for flash memory key).
- Well defined interfaces between the layers allow to easily exchange, e.g., Logical file system, I/O device.
 - E.g. the same file-system may be used on different devices or different file-systems may be used on the same type of device.



11.3 Allocation Methods: Contiguous Allocation

- How to store file contents in the blocks of a file system?
- Naïve approach: allocate file contents in contiguous blocks.



- Problem: If file needs to grow (e.g. File1 in the above example), subsequent blocks may already be allocated by other files.
- Possible solution: extents (→next slide).

Fragmented Allocation: Extents

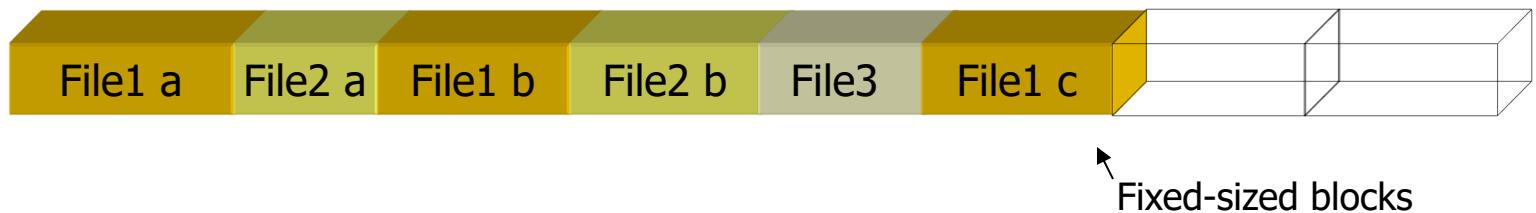
- Modification of contiguous allocation scheme:
 - If initial chunk of contiguous blocks is full, use additional contiguous chunks (an **extent**) that start at a new location and can be added to the already existing extents of a file.
 - Start and length of current extent needs to be stored together with information about where to find information about the following extent.
 - Such additional file allocation information is called (allocation-)**metadata**.



- Extents have an arbitrary size (in contrast to the one-block-per-allocation linked & indexed approaches covered in the remainder)
 - But still, size of an extent is a multiple of a block.
- As with main memory management, using arbitrary-sized extents may result in different sized holes of free storage areas.
 - Having many small free holes leads to a significant fragmentation.

Fragmented Allocation: Linked and Indexed Allocation

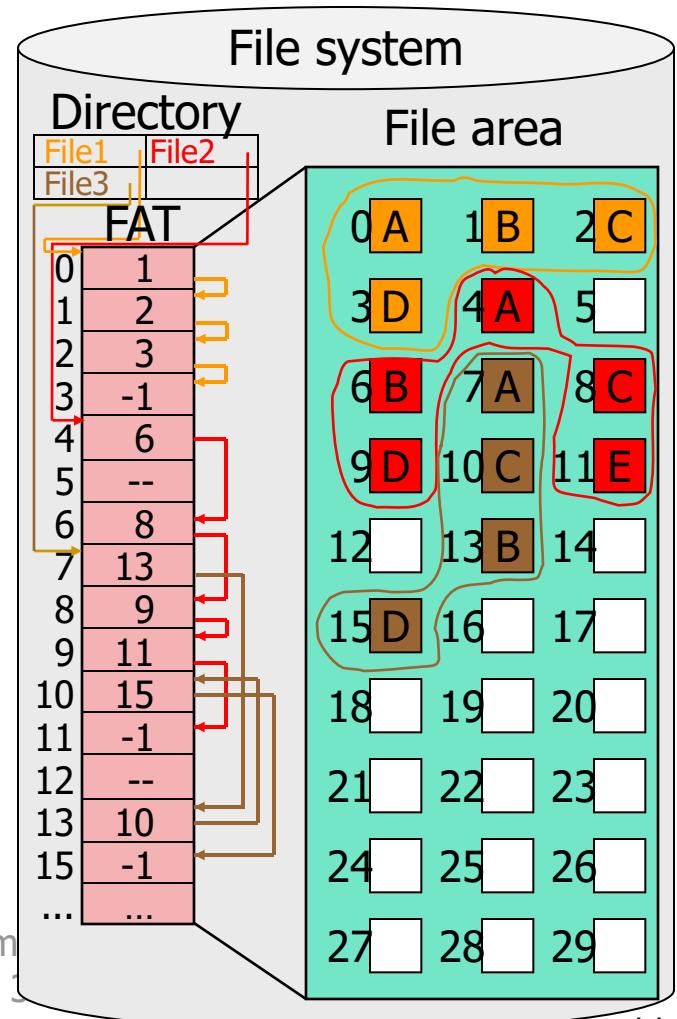
- Approaches that support fragmented storage by design (i.e. storing in a fragmented way does not require more allocation metadata than storing in a contiguous way):
 - For each individual **fixed-sized block** of a file, it is always (even if these blocks are contiguous) stored in the metadata where it is located in the file system.



- Typically used approaches:
 - **Linked allocation:** File Allocation Table (**FAT**) or
 - **Indexed allocation:** Index table (**I-Nodes**).
 - Also, Extents approach becomes popular again.

Linked Allocation: File Allocation Table (FAT)

- FAT=File Allocation Table (e.g. MS-DOS, USB keys, SD card).
- FAT is sorted with respect to blocks of file system:
 - FAT is a map of all blocks of the file system, i.e. for each block, one entry in the FAT exists.
 - The FAT entry of a block contains the number of the succeeding block of a file: can both be used to load that block and to identify next FAT entry (comparable to the pointer of a linked list).
 - -1 entry specifies the end of the linked list.
 - Directory entry points to first block of a file.
 - Depending on the number of blocks of the file system, a FAT entry has 12, 16 or 32 Bit (FAT12, FAT16, FAT32).

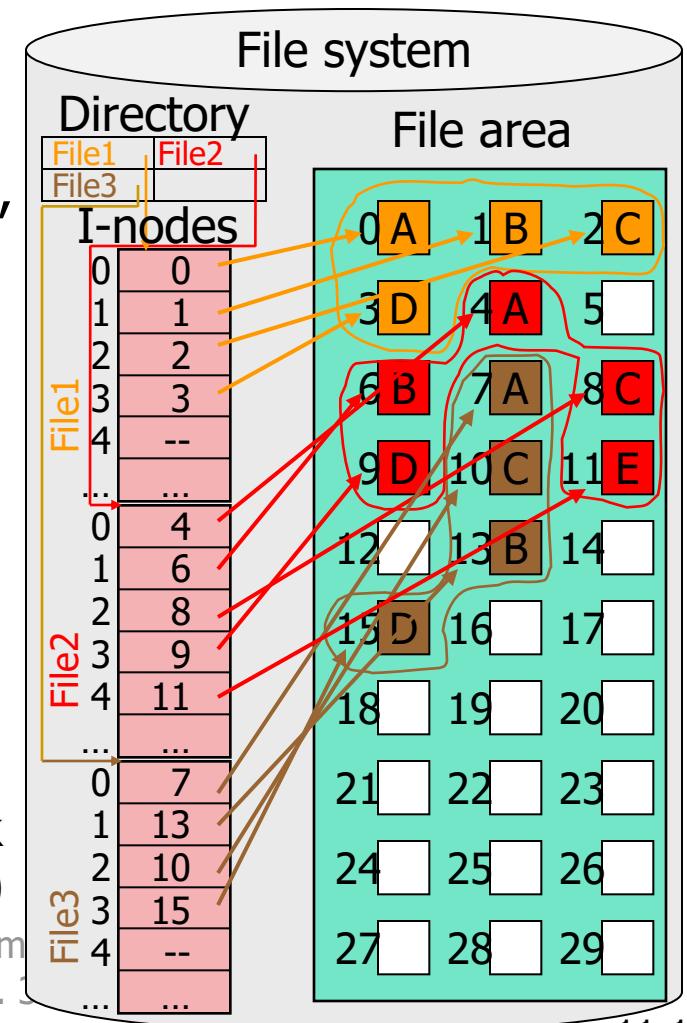


File Allocation Table: Disadvantages

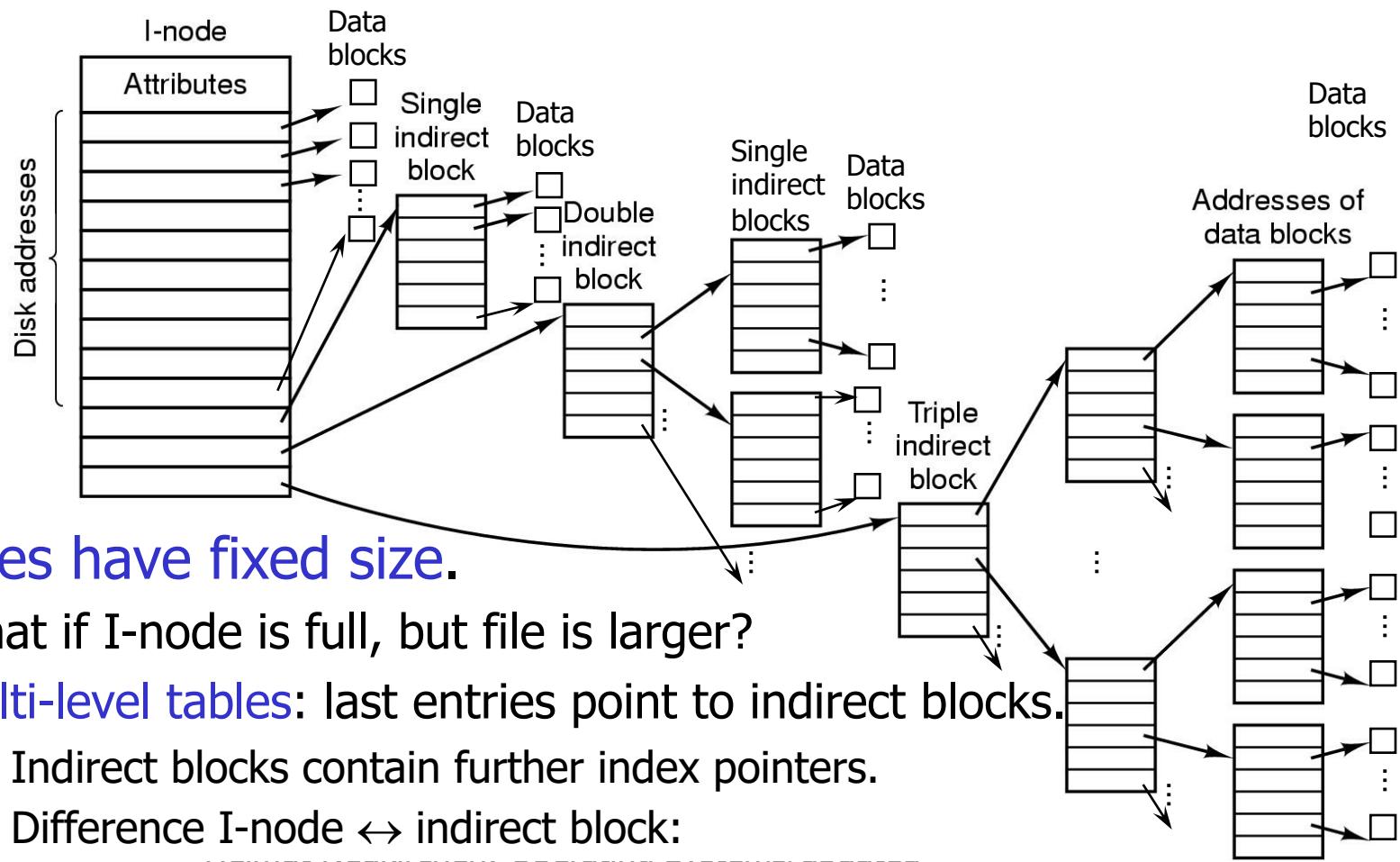
- If a file is stored fragmented, the corresponding FAT entries are as well scattered in the same way in the FAT.
 - In the worst case, for reading a file from start to end, huge parts of the FAT have to be traversed, i.e. the blocks of the FAT need to be read from the secondary storage device.
 - To speed up traversal: **keep complete FAT in main memory.**
 - Size of a FAT: e.g. 200 GB file system, 512 Byte block size
 $\Rightarrow 4 \cdot 10^8$ FAT entries, 4 Byte per entry $\Rightarrow 1.6$ GB for FAT :-(
 - ⇒ Approach to reduce size of FAT: **User clusters of blocks (up to 32KB clusters)**, i.e. FAT entries refer to cluster numbers.
 - Disadvantage of huge cluster sizes: **larger internal fragmentation!**
- To access the n^{th} cluster of a file (i.e. to find out in which cluster it is stored), $n-1$ FAT entries need to be traversed.
 - **Complexity to determine n^{th} block/cluster:** $\alpha(n)$

Indexed Allocation: I-Nodes

- I(ndex)-nodes (most Unix file systems).
 - Alternative spelling: inode.
- Instead of having a file system-wide table, each file has its own table: the I-node.
 - For speeding up file access, it is sufficient to keep just the I-nodes of the currently opened files in main memory.
 - Directory entry points to I-node of a file.
- Each I-node is sorted with respect to blocks of a file:
 - I-node of a file contains for each block of that file a pointer to the block in the file system. (E.g.: storage location of 2nd block of file can be found in 2nd entry of I-node.)
 - Direct access to nth block possible: α_1



Indexed Allocation: Multi-level Index



- I-nodes have fixed size.
 - What if I-node is full, but file is larger?
⇒ Multi-level tables: last entries point to indirect blocks.

- Indirect blocks contain further index pointers.
- Difference I-node \leftrightarrow indirect block:

I-node contains file attributes (except file name: stored in directory).

11.4 Free-Space Management

- To create a new file/extend an existing file, OS needs to know which blocks/clusters are free:
 - FAT: **Special cluster number** (typically -2) in FAT marks free cluster.
 - Linked list FAT data structure serves at the same time free-space management:
 - To find a free cluster: just search in FAT for a cluster with that special number. Complexity: $\mathcal{O}(n)$ (n =number of clusters managed by FAT).
 - I-node: only deals with existing files, i.e. keeps only track of allocated blocks, not of free blocks :-(
 - Use an **additional free space-bitmap** where one bit indicates whether the corresponding block is allocated or free.
 - To find a free block: just search bitmap for a bit that indicates a free block. Complexity: $\mathcal{O}(n)$ (n =number of bits in bitmap)
 - Size of free-space bitmap: as many bits as the file system has blocks.

Helmut Neukirchen: Operating Systems/updated
I.Hjörleifsson email:ingolfuh@hi.is st. 316 Grace,
Gróska

Free-Space Management: Counting

- Problem with free-space bitmaps needed by I-node file-system: For a 1 TB file system, a bitmap where each 4 KB block is represented by 1 bit has a size of 32 MB that needs to be searched/updated (=slow).
 - Solution (used by more recent I-node file-systems): As free blocks are often contiguous, **count** number of free blocks and **just store** for **each range of contiguous free blocks**: *[start of contiguous range of free blocks, number of contiguous free blocks in that range]*.
 - To find a free block: just search list for first entry (=first fit).
 - Complexity to find a free block in such a list:
 $\mathcal{O}(n)$ (n =number of entries in list).

Free-Space Management: Choosing free blocks

- Which free block/cluster to choose?
 - Same memory allocation strategies as for main memory: e.g. First Fit.
 - When extending an existing file: chose location near to existing location (e.g. First Fit starting at current end of file) to minimise mechanical head movements of hard disk.
 - (Not relevant for electronic disks (SSDs) that have no moving mechanical parts.)

Free-Space Management: SSD TRIM

- Problem of electronic disks (SSDs):
 - Each write to a flash-memory cell wears that cell out (=after a finite number of writes, cell “dies”).
 - An SSD is not aware of the file system used by the OS, hence it does not know that a free-space block of the file-system stored on the SSD is in fact currently not used for storing data.
 - A file-system implementation can send the SSD a so-called **TRIM** command to tell the SSD that a block is not used.
 - ⇒ SSD can then internally map heavily used file-system blocks to unused blocks and thus achieve **wear leveling** of all flash-memory cells.
 - (See also forthcoming ch. 12.)

Free-Space Management (De-)Fragmentation/Compaction

- If a file system has been in use for some time and some files have been deleted & many files added, it may be possible that no contiguous blocks can be found to store a new file or extend and existing file: external fragmentation.
 - Contents of a file is stored at various locations scattered throughout the file system.
 - On a mechanical hard disk (→ch. 12), this slows down sequential reading of a file.
- Defragmentation/Compaction: Move blocks/cluster (by copying them)
 - to remove external fragmentation of files (i.e. store blocks/cluster of a file contiguous) and
 - to create contiguous free space for future files.

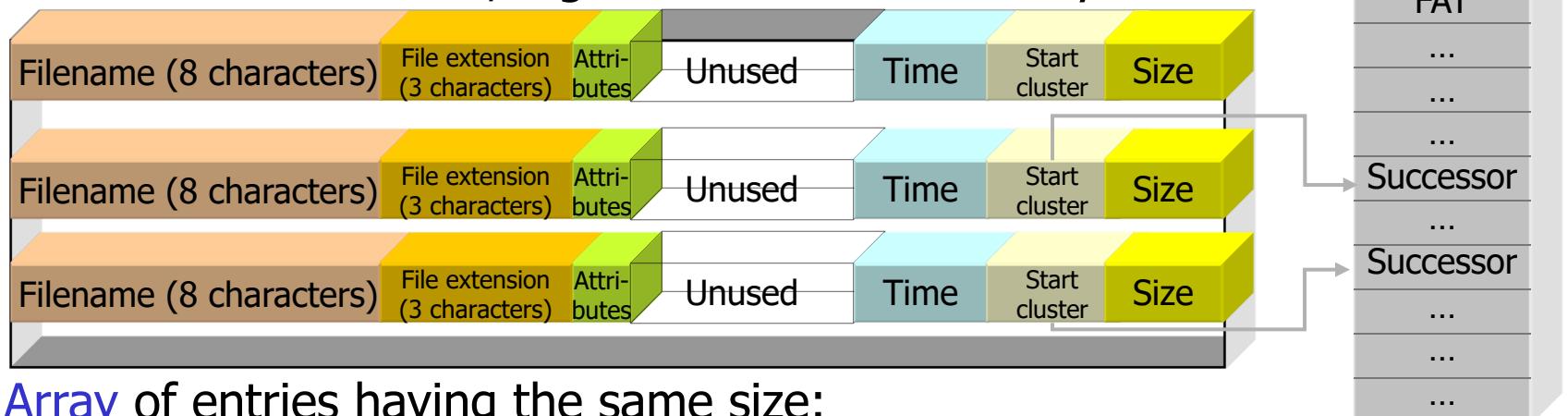
Free-Space Management

Fragmentation on I-node systems

- Note: It is often claimed that I-node based file systems do not suffer from fragmentation: this is not true!
- Fragmentation occurs, but maybe less severe:
 - Most I-node based file systems typically divide the file system into sections, each of them having their own “pre-allocated” set of I-nodes. New blocks are preferably chosen from the same section as the corresponding I-node (=less far-away scattered).
 - ⇒ More likely that all blocks of a file are located in the same section, less likely that a file “steals” free blocks from other sections (only if current section is full), thus leaving free blocks of another section for I-nodes of that other section.
 - ⇒ Blocks of a file may still be fragmented, but are typically closer together (=access with a mechanical hard disk not slowed down that heavily).

11.5 Directory Implementation: Entries with fixed size

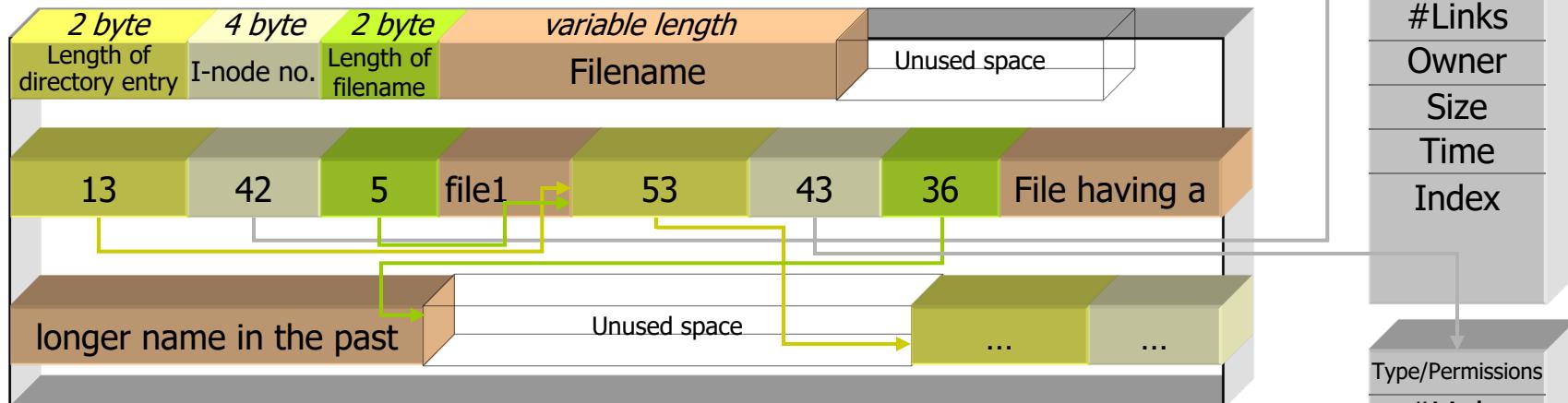
- File name of **fixed size**, e.g. MS-DOS FAT 16 file system:



- Array of entries having the same size:
 - Deleting an entry: Mark entry as empty (set first character to 0xE5).
 - Creating new entry: Find first empty entry.
- Entries unsorted in order of creation \Rightarrow Searching an entry: $O(n)$.
- All file attributes in directory entry \Rightarrow Hard links not possible.
- Longer file names (since Win95, backwards compatible to MS-DOS):
 - Entry with invalid combination of attributes contains in further elements of the entry the long name that refers to the subsequent entry with 8+3 name.

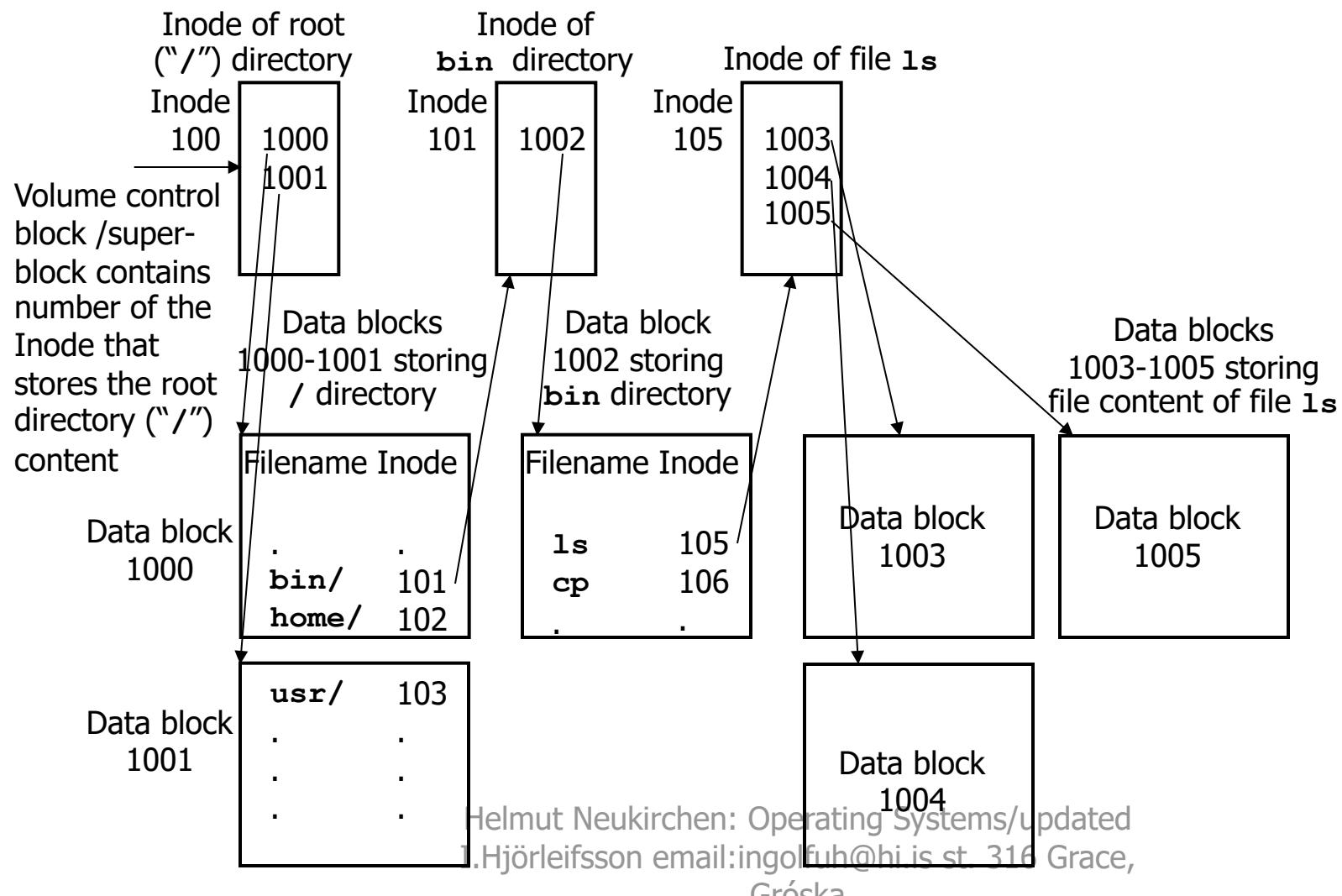
Directory Implementation: Entries with variable size

- File name with **variable size**, e.g. most Unix file systems:



- Linked list of entries with variable size:
 - Renaming, name shorter: remaining space unused.
 - Renaming, name longer: create new entry, delete old entry.
 - Delete entry: Unused space of previous entry extends.
 - Compaction as required (e.g. if no space left in directory block).
- Entries unsorted in order of creation \Rightarrow Searching an entry: $O(n)$.
- Attributes (except name) in I-node \Rightarrow Hard links possible.**

Example Inode file system with directory and file (Inode 100 is entry point into file system)



Directory Implementation: Hash Tables/Sorted Trees

- The fixed and variable sized directory implementation assume a linear list-based directory data structure.
 - The associated complexity of $\mathcal{O}(n)$ of searching an entry in a linear list-based directory data structure with n entries becomes a problem if a directory contains thousands of files.
 - Solution: Faster (but less easy to implement) data structure for directory entries.
 - **Sorted trees**: search complexity $\mathcal{O}(\log n)$,
 - **Hash tables** (í. tætitafla): search complexity $\mathcal{O}(1)$.
 - File name mapped on a single number (using a hash function, e.g. one that calculates a checksum). Number used as index into a table that contains directory entries (collisions possible if different file names map on the same hash value: hash algorithm needs to deal with this).

Sparse Files and Short Files

- **Sparse files:** if a file contains blocks that consist of only entries with value 0:
 - Do not waste a data block for storing these zeros, but just use a special marker (e.g. -1) in the I-node entry for that block.
 - Should later that block not be anymore 0 only, just start using a normal data block and let I-node point to it.
- **Short files:** If a file has a size of just a few bytes (typical example: a symlink containing a short path name):
 - Do not waste a data block for storing short file, rather use the I-nodes themselves (i.e. the areas otherwise containing the block pointers) for storing the short file contents.
 - Should later the file grow in size, still possible to use normal data block and let I-node point to it.

11.7 Recovery

- Files store valuable data, so being able to recover from file system problems is important.
 - Hard disk crashes, bugs in file system implementations, but also OS crashes or power outage during file system updates may lead to data loss.
- Example: Steps when appending to a file (I-node based scenario):
 - Select block locations to be used based on bitmap of free blocks,
 - Mark blocks as allocated in bitmap of free blocks,
 - Add additional block pointers to indirect blocks (if involved) & I-node,
 - Write actual file contents to selected blocks locations.
 - Update file size attribute stored in I-node.
- As soon as any of this steps (except the first) fails (power outage/system crash), the **file system becomes inconsistent**, e.g.:
 - Bitmap of free blocks and actual block allocation do not match,
 - Indexes in I-nodes/indirect blocks point to wrong locations, ...
- (The probability increases when write accesses to disk are cached.)

Consistency Checking

- Regularly check file system consistency to prevent that a possible corruption affects even further data.
 - Often done at the next system boot after a system crashed (=has not been properly shut down, i.e. a flag in the file system on disk is set by OS when mounting, but has not been reset by proper unmounting).
- Tools for checking the consistency of file systems:
fsck on Unix, **chkdsk** on MS Windows.
 - Scan a whole file system (all the metadata) and collect information about file systems.
 - Compare whether this information fits together.
 - If not: report inconsistency.
 - Tools may even be able to repair some inconsistencies.
 - Typically, lost data cannot be reconstructed, but the metadata can be repaired.

Journaling/Log-based Transaction-oriented File Systems

- Consistency checks and file system repairs are performed after an inconsistency has occurred.
- Journaling (or log-based transaction-oriented) file systems can avoid inconsistencies to occur at all.
 - (At least, inconsistencies due to power outage/system crash while writing data can be prevented – but still data that has not been written yet, can get lost).
 - Journaling file systems are state of the art in all modern file systems e.g. (Microsoft NTFS, all modern POSIX/Unix-like file systems, but not FAT).
- Based on the concept of transactions: Either write all data or no data! (I.e. either old file system state or new file system state.)
 - Three step approach: announce action, commit action, acknowledge action. (→next slide...)

Helmut Neukirchen: Operating Systems/updated
I.Hjörleifsson email:ingolfuh@hi.is st. 316 Grace,
Gróska

Journaling/Log-Based File Systems: Approach

1. Write information about intended changes to an intermediate buffer („Journal“ or „Log“) on file system. Format: [header, changes, trailer].
 - If crash during this step: actual storage location has not been changed, yet. After restart, journal entry will get discarded due to missing trailer.
2. If journal full: Write changes logged in journal to actual storage locations in file system.
 - If crash (i.e. changes only partially written, journal entry not deleted yet): After restart, write again changes still stored in journal (steps 2 & 3).
3. Delete entry from journal.
 - If crash during this step: Restart with step 2 (if journal entry not yet deleted) or do nothing (journal entry deleted).

Journaling/Log-Based File Systems: Discussion

- Write accesses get slower, as all data is written twice: first to journal, then to actual storage location.
 - But: journal uses contiguous blocks, thus write (typically using synchronous write) is sequential (=fast)
 - If no journaling would be used, writes would be made to different locations (I-nodes, free space-bitmap, directory, data blocks) in a random-access style = slow!
 - Writing the changes logged in journal to actual storage locations in file system is done in the background using asynchronous write, i.e. may occur when the hard disk is anyway idle.
 - In the optimal case, journaling may even be faster than no journaling.
- Faster variant: use journaling only for metadata (data structures, no file content).
 - File content may get inconsistent, file system structure will be still consistent.

11.8 Summary

- Today's operating systems support fragmented storage of files.
 - Information about location of file contents:
 - Link-based (FAT),
 - Index-based (I-nodes).
 - Recent file-system take up again the extents-based approach for speed reasons.
 - Information about free space: like management of free main memory.
- Directories may support fixed or variable size file names.
 - Storing all file attributes (other than file name) in I-node enables hard-links.
- Journaling file systems guarantee consistency of file system data structures.
 - Still important: backup!
- Only basic concepts presented. Not covered:
 - Network file systems (covered in TÖL503M Distributed Systems course).
 - Microsoft NTFS file system, recent Linux and Mac OS file systems.

Still to discuss other operating systems/updated

I.Hjörleifsson email:ingolfuh@hi.is st. 316 Grace,
Gróska

Course TÖL401G: Stýrikerfi / Operating Systems 13. I/O Systems

Mainly based on slides and figures
copyright Silberschatz, Galvin and Gagne

Chapter Objectives

- Explore the structure of an operating system's I/O subsystem.
- Discuss the principles and complexities of I/O hardware.
- Explain the performance aspects of I/O hardware and software.

Contents

1. Overview I/O Systems
2. I/O Hardware
3. *Application I/O Interface*
4. *Kernel I/O Subsystem*
5. *Transforming I/O Requests to Hardware Operations*
6. *Performance*
7. Summary

Helmut Neukirchen: Operating Systems/updated
I.Hjörleifsson email: ingolfuh@hi.is st. 316
Grace, Gróska

13.1 Overview Input/Output Systems

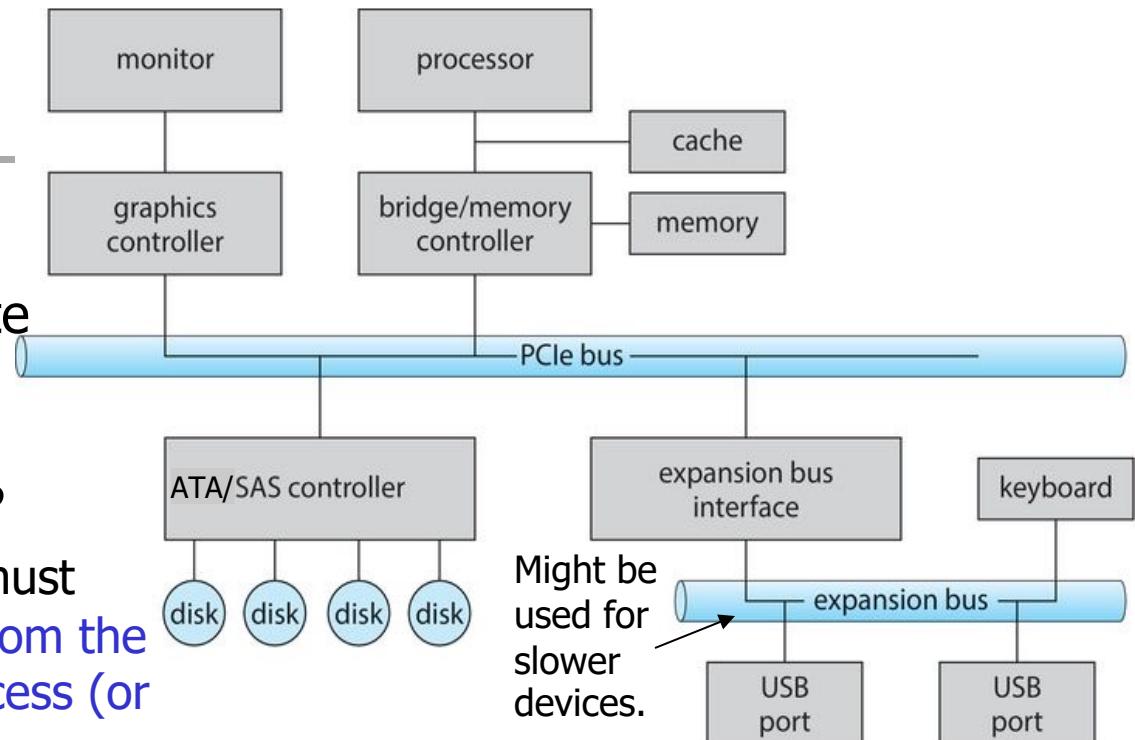
- **Magnitude of I/O devices** (storage, transmission, human-interface):
 - Printer
 - Scanner
 - Keyboard
 - Mouse
 - Game controller
 - Digitiser touch screen/tablet
 - Hard disk
 - Flash memory SSD disk
 - CD/DVD/Blu-ray Reader & Writer
 - Tape drive
 - Graphic card
 - Sound card
 - GPS
 - Accelerometer
 - Wired & Wireless network
 - ...
- **Magnitude of hardware interfaces:**
 - USB
 - (Serial) ATA
 - NVM Express (NVMe)
 - Serial-attached SCSI (SAS)
 - Thunderbolt,
 - PCI Express (PCIe),
 - Legacy interfaces (parallel, serial),
 - Ethernet, ...

I/O and the Operating System

- Responsibility of the OS concerning I/O devices:
 - Manage I/O devices (using **device drivers**):
 - Provide processes/programmer a simple, but controlled access.
 - In multiprogramming systems:
 - Provide a fair access to I/O device used by multiple processes.
 - Minimise I/O performance bottleneck using suitable scheduling.
 - I/O devices are much much slower than CPU and main memory.
 - Avoid that I/O-intensive processes slow down CPU-intensive processes!
 - Cope with magnitude of different devices and interfaces:
 - Application developer should not have to deal with magnitude.
⇒Provide a standard interface for unified access to devices with comparable characteristics.

13.2 I/O Hardware

- How do device drivers (=software) communicate with the actual I/O devices or their device controllers (=hardware)?
 - OS and I/O hardware must support sending data from the address space of a process (or of the OS performing the respective I/O system call) to an I/O device and vice-versa.
 - Devices are either directly connected to the CPU via the data and address bus, or via some peripheral bus, e.g. the PCIe bus (having multiple PCIe "lanes").



- Note: Nowadays, many controllers are not plugged into a PCIe slot, but integrated into the bridge/memory controller (it's "southbridge" part).
- In fact, since 2011, the bridge/memory controller (former "northbridge") is part of the CPU chip itself, and the device controllers (incl. integrated graphics) are located in the Platform Controller Hub (PCH) chip.

Helmut Neukirchen: Operating Systems/2014
1.Hjorleifsson email: ingsfuh@chis.tu-316
Grace, Gróska

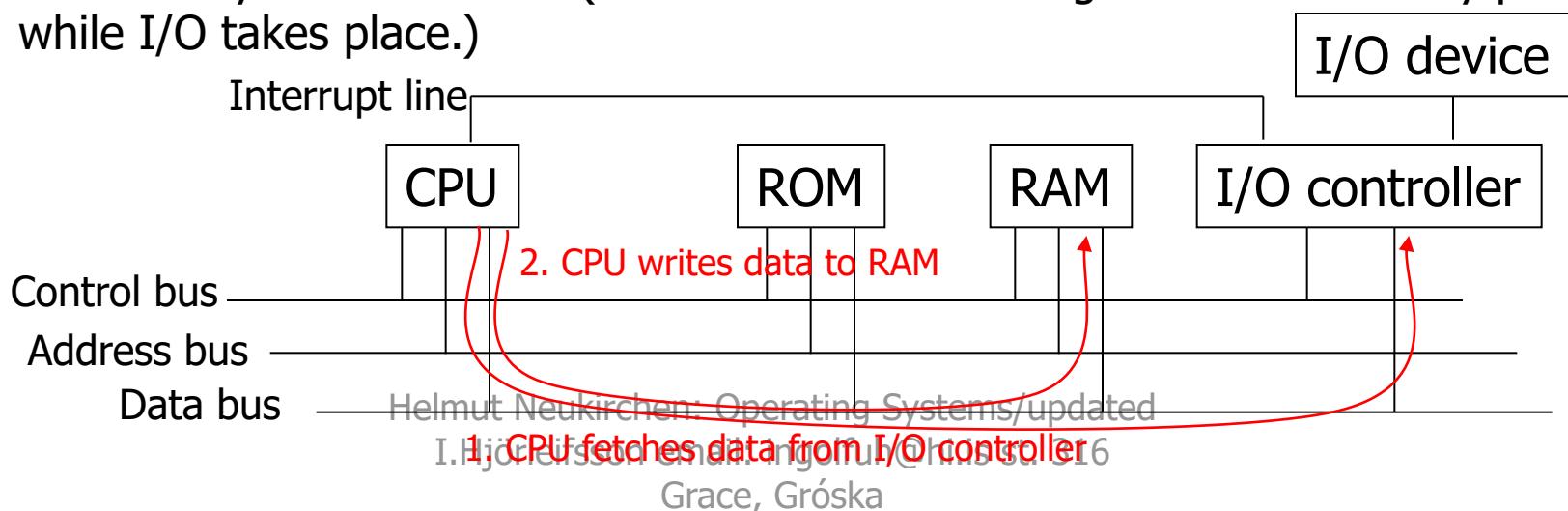
Starting from 2014 also the PCH is sometimes part of the CPU chip itself.

I/O Hardware

- Possible ways for connecting an I/O controller to a CPU or main memory (more on next slides):
 - Programmed I/O (PIO),
 - Memory-mapped I/O,
 - Direct Memory Access (DMA).
- Knowing the different characteristics of these hardware connections may help minimising the I/O performance bottleneck.
 - Optimal: While I/O device is working (corresponding process is blocked), CPU can be assigned to other (ready) process.

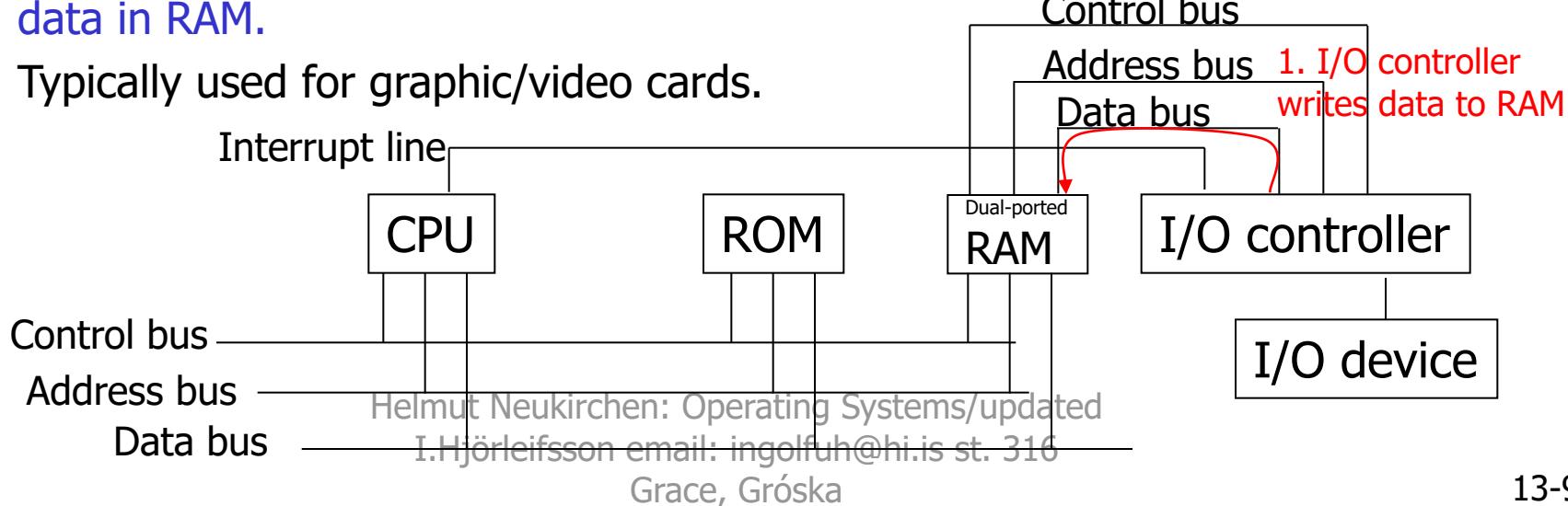
Programmed I/O (PIO)

- CPU is connected to a data register ("I/O port") of the controller, i.e. device driver or CPU respectively have to read/write data byte-wise from/to controller.
 - **Polling**: Device driver (=CPU) has to read periodically status register of controller whether data needs to be transferred from/to controller. (Disadvantage: **busy waiting**)
 - **Interrupt**: Controller raises interrupt if data must be transferred. Interrupt handler dispatches to device driver that transfers actual data from/to controller. (Disadvantage: interrupt handling is slow due to involved **context switch**.)
- Disadvantage of PIO: not suitable for multiprogramming, as byte-wise transfer of data is very **CPU intensive!** (I.e. CPU cannot be assigned to some ready process while I/O takes place.)



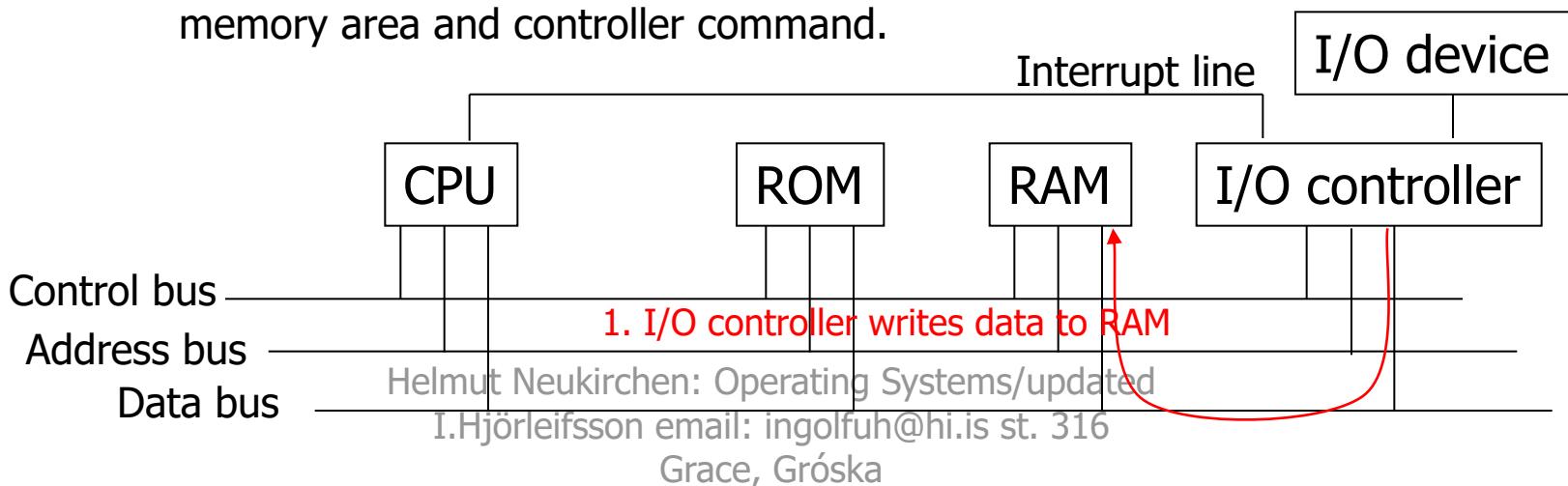
Memory-Mapped I/O

- Controller (e.g. graphic card) has its own RAM memory that is mapped into the physical address space of the CPU.
- Disadvantage: RAM memory that can be used by two parties at the same time (i.e. by controller & CPU), is either **more expensive** (dual-ported RAM) or **slower** (shared Memory: I/O controller steals bus memory cycles from CPU to access ordinary (=single ported) RAM – however no problem if CPU cache contains all data needed by CPU) than ordinary RAM.
- Advantage: Less CPU intensive. **CPU can work while controller is accessing data in RAM.**
- Typically used for graphic/video cards.



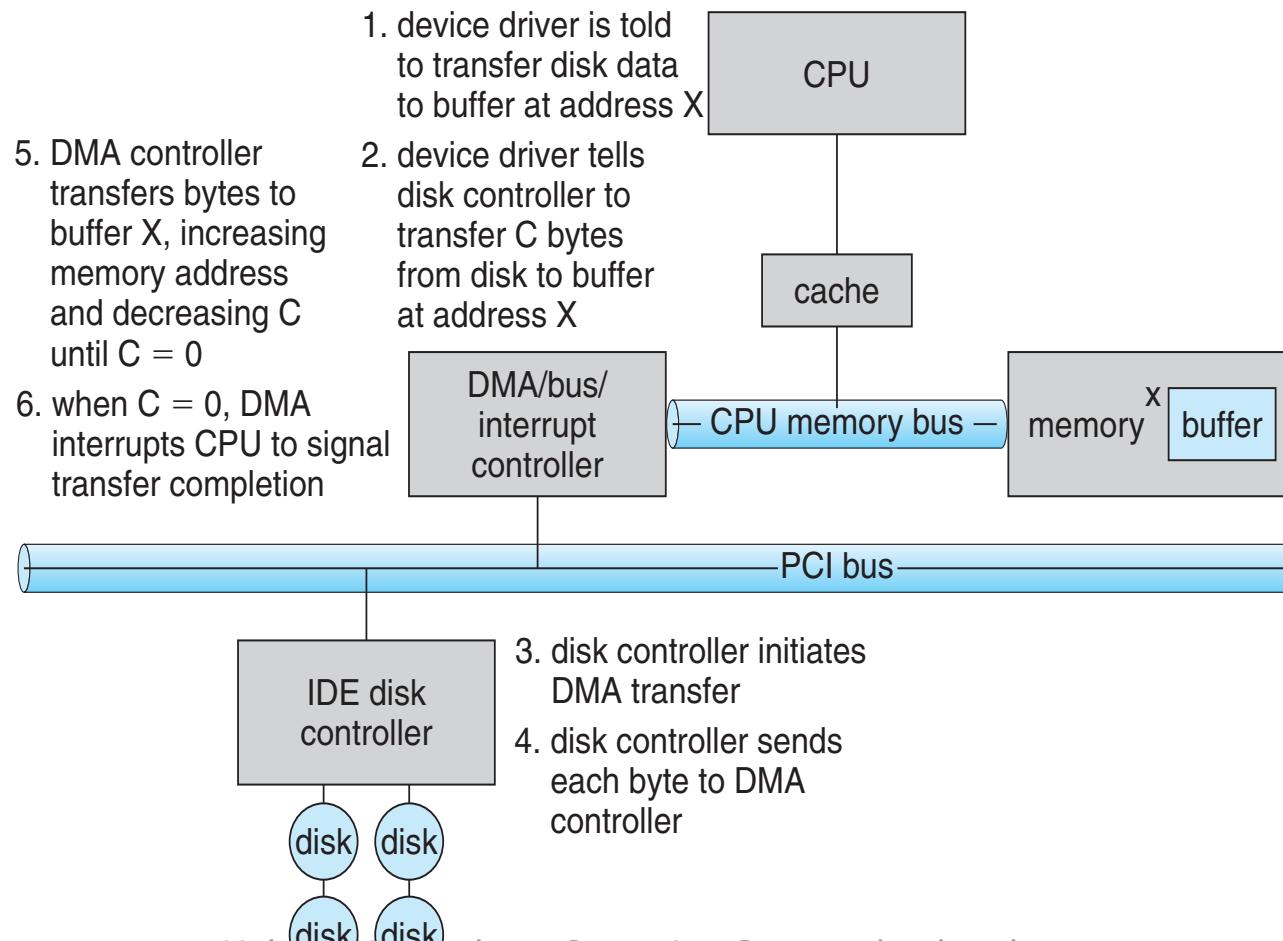
Direct Memory Access (DMA) (1)

- Controller (e.g. disk controller) is able to write from/to arbitrary addresses of the main memory without intervention of the CPU.
 - CPU just configures DMA controller once (using PIO) for each transfer:
 1. command to be executed by controller (e.g. sector to be read from disk & number of consecutive sectors to be read),
 2. start of memory address to read from/write to via DMA.
 - DMA controller raises interrupt when DMA transfer is finished.
 - CPU can then, e.g., issue another command.
 - Each DMA transfer comprises exactly one consecutive memory area and controller command.



Direct Memory Access (DMA) (2)

Detailed Steps



Direct Memory Access (DMA) (3)

Discussion

- Advantage: Less CPU intensive. CPU can work while controller accesses data in RAM.
 - Ideal for virtual memory using paging: While DMA controller is loading pages, CPU may still execute those processes that are ready and in physical memory.
 - However, frames that are currently subject of DMA transfer must not be chosen as victim for replacement while DMA transfer takes place!
 - Still, DMA transfer slows down access of CPU to RAM ("memory cycle stealing" by DMA controller to access RAM in parallel to CPU just like shared-memory approach of memory-mapped I/O).
 - But again: no slow down if data needed by CPU is in CPU cache.
- All major device controllers that transfer huge amounts of data are nowadays DMA-enabled.
 - E.g. hard disk/SSD controller, USB controller

13.7 Summary

- DMA attachment of mass storage devices allows CPU to compute in parallel to I/O.
- OS provides a unified interface for accessing I/O devices.
- Layered approach allows to exchange low-level device drivers while keeping higher-level device-independent drivers and file system implementations.
- Kernel I/O subsystem may help to reduce I/O bottleneck.

Course TÖL401G: Stýrikerfi / Operating Systems

14. Protection & Security

Based on slides copyright Silberschatz, Galvin and Gagne, 2013

Contents

1. Introduction
2. Principles of Protection
3. Domain of Protection
4. Protection Access Matrix
5. Implementation of the Access Matrix
6. The Security Problem
7. Programs as Security Threats
8. System and Network Threats
9. Cryptography as a Security Tool
10. User Authentication
11. Summary

Protection Security



Note for users of the Silberschatz et al. book: this is a shortened and combined version of chapters 14 and 15.

14.1 Introduction: Protection vs. Security

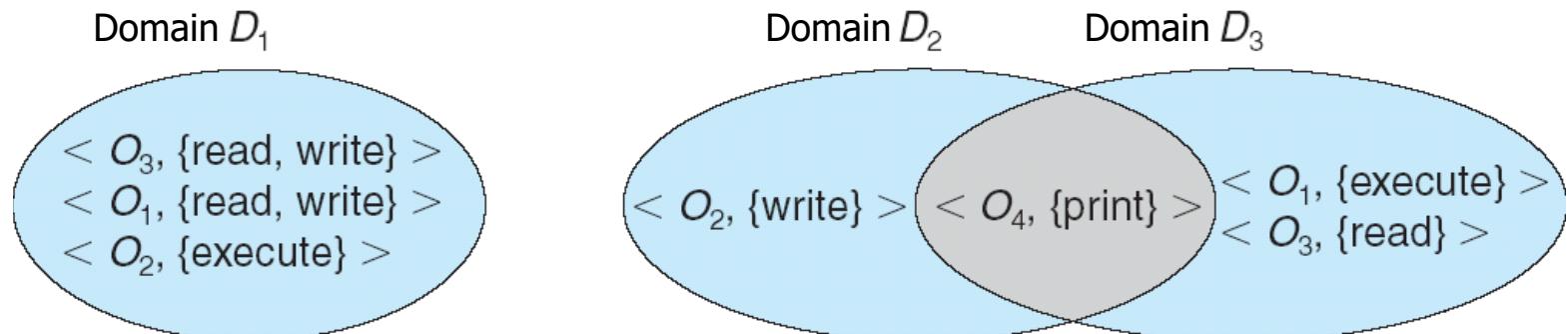
- **Protection:** Mechanism to control who may access which resources in a computer system.
 - Essentially: who of the known, internal authenticated users and processes may do what.
 - Addressed by, e.g., policies based on file permissions and other access rights.
- **Security:** Preserve confidentiality and integrity of system.
 - Essentially: prevent that any unknown, external user (or processes) may perform an intentional or accidental malicious action.
 - E.g. external user may try to get access rights of an internal user.
 - Addressed by, e.g., virus scanners, firewalls, encryption, secure coding.
- **Authentication:** Checking identity – Is this really user x ?
- **Authorisation:** What is authenticated user x allowed to do?
- Protection & Security work together:
 - Authorisation (i.e. protection) useless if already authentication (i.e. security) is broken.

14.2 Principles of Protection

- Principle of least privilege:
 - Programs, users and systems should be given just enough (but not more) privileges to perform their tasks.
 - E.g.:
 - It would be stupid to give super user privileges to an ordinary user.
 - Even if a user needs access rights for some privileged resource, do not grant super user privileges, but just the additional privilege needed for that particular privileged resource.
- Need-to-know principle:
 - Programs, users and systems should just know enough (but not more) to perform their tasks.
 - (As knowing something is comparable to have privilege to access something, these principle are quite similar.)

14.3 Domain of Protection

- To facilitate these principles, protection domains are used:
 - A process or user belongs to a particular protection domain.
 - Each protection domain has a different set of access rights for the different resources.
 - (In accordance to the object-oriented paradigm, the term object is often used for a resource and its access operations.)
 - Access-right = $\langle \text{object-name}, \text{rights-set} \rangle$
 - where rights-set is a subset of all valid operations that can be performed on the object.
 - Domain = set of access-rights (i.e. objects O_i and their rights set).



Domain Implementation in POSIX

- Domain = user ID/group ID. slide 10-22:
- Access rights for files.
 - Owner of file may change access rights (e.g. `chmod` command).
- As devices are also accessed as files, low-level access to devices can be protected just like files (→slide 13-15).
 - Access rights of device files prevent direct device access by ordinary users.
- Some (=privileged) system calls only callable by the super user `root`.
- To support protection principles, dynamic domain switching allows ordinary users to execute system programs that use privileged system calls or devices: setuid ("set user ID") file permission:
 - If an executable program file has setuid bit set (by owner using `chmod`), the user ID of the process is set to the user ID of the file owner while that program file is executed. When execution completes, user ID is reset.
 - E.g. `ping` system program: ~~-rwsr-xr-x root root /bin/ping~~

Reminder from

`-rw-r--r-- helmut hi myfile.txt`

All may read.
Members of group `hi` may read.
Owner `helmut` may read/write.

setuid bit set Set user ID to `root` All may execute ⇒ `ping` is executed by all with `root` user ID

14.4 Protection Access Matrix

- Conceptually, different access rights for different domains can be viewed as matrix (**access matrix**):
 - Lines represent domains (e.g. one domain per user or group of users).
 - Columns represent objects (e.g. files F_i).
 - $access(i, j)$ is the set of operations that a process executing in Domain_i can invoke on Object_j,
 - e.g.: read, write, execute operations for a file F_n ; print operation for a printer object.

object domain	F_1	F_2	F_3	printer
D_1	read		read	
D_2				print
D_3		read	execute	
D_4	read write		read write	

Protection Access Matrix

Domain Switching

- To support domain switching, a special “switch” operation/access right is used:
 - By adding domains as well as objects (=as additional columns) and defining a “switch” operation, it may be specified for each domain to which other domain a switch is allowed.
 - E.g.: $\text{switch} \in \text{access}(\text{Domain } D_2, \text{ Object } D_3)$
⇒ A user/process from domain D_2 can switch to domain D_3 .

object domain	F_1	F_2	F_3	laser printer	D_1	D_2	D_3	D_4
D_1	read		read			switch		
D_2				print			switch	switch
D_3		read	execute					
D_4	read write		read write		switch			

Changing the Access Matrix

- It must be defined, who is allowed to change the access matrix.
 - ⇒ **Make the access matrix itself an object**: enables to define who is allowed to change the access matrix.
 - Typically, each column (=object) of the access matrix needs to be considered as a different object (i.e. add “change” operation).
 - E.g. in UNIX: no access matrix, but file-based access rights:
 - only the owner of a file may call **chmod** to change access rights of the file,
 - only the owner of a file may call **chgrp** to change the group to which the file belongs,
 - only the superuser of a file may call **chown** to change the owner of the file (not even the current file owner can do this).
 - Shell commands invoke corresponding system calls: OS kernel checks domain of calling user (and file access rights).

14.5 Implementing the Access Matrix

- An access matrix tends to get very very large! (Imagine, one column for each of the thousands files in a file system.)
 - In practise, not a real matrix is used. Instead, e.g.:
 - **Global table:** A list of triples <domain, object, access rights set>.
 - Whenever an operation is to be executed from within domain D_i on object O_j , the global table is searched for an entry < D_i , O_j , access rights set> and then checked whether the access rights set allows the intended operation.
 - Global table may still be very large (cannot be kept in main memory).
 - **Access list:** Each object stores access rights for those domains that have access rights defined on it using a pair <domain, access rights set>.
 - Whenever an operation is to be executed from within domain D_i on object O_j , the access list of object O_j is searched for a pair < D_i , access rights set> for domain D_i and then checked whether the access rights set allows the intended operation.
 - Example UNIX: each file stores its file attributes access rights (`r,w,x`, etc.) for owner, group, others.
 - Only to be kept in memory for current object. (POSIX: when opening file.)

14.6 The Security Problem

- A protection approach is ineffective, if already user authentication is compromised or programs can be run by unauthorised users.
 - **Security** must consider **external environment** (but of course also the internal environment) of the system!
- **Intruders** attempt to breach security.
- **Threat** is the potential for a security violation.
- **Attack** is an attempt to break security.
- Attack can be accidental or malicious.

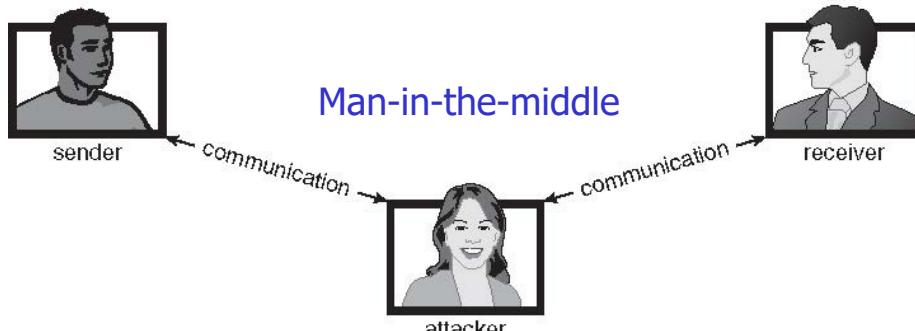
Security Violations

- Categories:
 - Breach of confidentiality: unauthorised reading of data.
 - Breach of integrity: unauthorised modification of data.
 - Breach of availability: unauthorised destruction of data.
 - Theft of service: unauthorised use of resources.
 - E.g. someone uses your CPU to do calculations or someone breaks into your cloud service account and you get then charged for the cloud services used.
 - Denial-of-Service (DoS): preventing legitimate use of a system.

Most Common Methods to Violate Security

- Most common:

- **Masquerading** (breach of authentication): pretend to be someone else.
- **Replay attack**: malicious repeat of a captured valid data exchange.
 - Often involves message modification.
- **Man-in-the-middle attack**: intruder sits in the chain of communication, masquerading as receiver to the sender and as sender to the receiver, thus being able to capture and to modify data exchange.
- **Exploitation of vulnerabilities**: use bugs in software to escalate privileges.



Security Measure Levels

- Security measures are necessary at all levels:
 - **Physical**: secure rooms where computers are located.
 - **Human**: users need to be aware of security.
 - E.g. use of complex passwords, recognise phishing attack.
 - **Operating System**: protect itself from security breaches.
 - E.g. avoid typical vulnerabilities.
 - **Network**: prevent replay-, man-in-the-middle- and DoS-attacks.
 - E.g. use of cryptography!
 - (DoS attacks cannot be prevented by cryptography.)
- Security is as weak as the weakest link of chain!

14.7 Programs as Security Threats

- Programs are (together with the OS kernel code itself) the entities that are executed. Hence, it is common to use programs to break security:
 - [Trojan Horse](#),
 - [Virus](#),
 - [Back Door \(or Trap Door\)](#),
 - [Logic Bomb](#),
 - [Stack and Buffer Overflow](#).
- More on these on the following slides.
 - Note: assuming that your OS comes from a trusted source, the OS contains rarely a Trojan horse, virus, back door or logic bombs.
 - But OS may contain vulnerabilities due to programming errors such as possible stack and buffer overflows.

Trojan Horse

- A program that misuses its environment:
 - Program that is intended to be executed in a domain of some other user (by pretending to be some other program), thus being able to perform malicious actions with the access rights of that user.
- Examples:
 - A program sent in the attachment of an e-mail claiming to be some important PDF document that however contains executable malicious code.
 - Software that fulfills some usable purpose (and therefore people download and install it by intention), but does in addition do malicious things (Spyware).
 - Installed while having superuser privileges \Rightarrow spyware has superuser privileges as well.
 - Badly configured executable search path (environment variable `PATH`) that contains in addition to, e.g., `/bin` the current working directory (".".): if a user works in a directory where another user has write access to (e.g. `/tmp`) it is possible to place a trojan horse into that directory (e.g. `1s` in `/tmp` gets executed, not `/bin/1s`).
 - Fake login program: a malicious user leaves a program running that looks like the normal login prompt; the next user will enter username and password which gets recorded by the fake login program.
 - This is why MS Windows uses ctrl-alt-del to get the login prompt: this would reveal a fake login program: ctrl-alt-del screen would then appear instead of real login prompt.

Virus

- Like a Trojan horse, but malicious code fragment is embedded in legitimate program and thus gets executed in a domain of a user.
- In contrast to trojan horse: Self-replicating by infecting other programs or application files (e.g. MS Office documents).
 - Infected files are then typically handed over (without knowing that these are infected) to other users (via e-mail attachment, USB key).
⇒ machines of other users get infected as well.
- Consists of a part for self-replication and a part containing the actual malicious code.
 - Malicious code may be everything, e.g.:
 - Deleting user files/encrypting them and request money in order to decrypt,
 - Sending user data over the internet,
 - Theft of service: Distributing spam e-mails or participating in a distributed denial of service attack (DDoS),
 - Waiting for a command/downloading command code from a command server in the internet.

Sample Malicious Code

- Very specific to operating system / applications.
 - E.g.: virus contained in a macro of some document; macro gets executed when document is opened by application.
 - Visual Basic Macro in MS Office document to format hard drive:

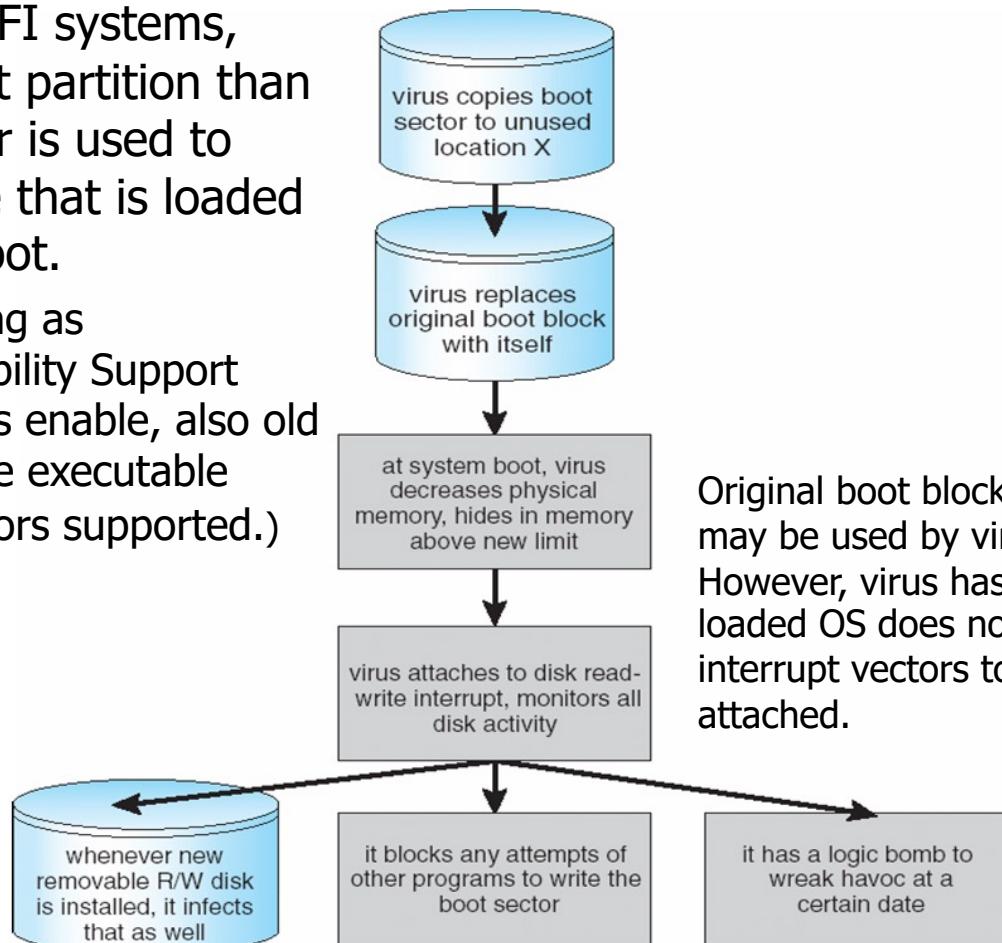
```
Sub AutoOpen()
    Dim oFS
    Set oFS =
        CreateObject("Scripting.FileSystemObject")
    vs = Shell("c:\command.com /k format c:", vbHide)
End Sub
```

Virus: Categories

- **File virus:** appends itself to executable binary file, changes part of program so that appended virus code gets additionally executed.
- **Boot sector virus:** infects the boot sector of a system. Executed at every boot: watches for other attached bootable media (e.g. USB key) and infects their boot sector as well.
 - Example on next slide.
- **Macro virus:** infects application files that support executing macros.
- **Source code virus:** looks for source code and inserts itself into the source code.
- **Firmware virus:** infects firmware code (BIOS/UEFI).

Virus: A Boot-sector Computer Virus

- (On pure UEFI systems, rather a boot partition than a boot-sector is used to contain code that is loaded at system boot.
 - But as long as “Compatibility Support Module” is enable, also old BIOS-style executable boot-sectors supported.)



Original boot block stored at location X may be used by virus to load the OS. However, virus has to take care that loaded OS does not overwrite the interrupt vectors to which the virus attached.

Virus: Detecting / Hiding from Detection

- Virus scanner software searches files/programs for the signature of a virus.
 - Fixed signatures (e.g. series of instructions) of known viruses, generic signatures of suspicious typical generic virus code.
- Virus shield protection software tries to recognise typical virus actions.
 - E.g. a boot sector virus using `write()` system call to change the boot sector.
- Viruses try to hide from virus scanners/protection shields:
 - Polymorphic virus: changes each time to avoid having a fixed virus signature.
 - Encrypted virus: virus code is encrypted to avoid exposing generic virus signatures (decrypts itself before each execution).
 - Stealth virus: avoids detection by modifying OS, e.g. modify `read()` system call so that if virus detection software uses it to read a file, the original file is returned.
 - Might even provide a VM hypervisor to run the actual OS (and virus scanner). Virus hides inside VM hypervisor. (Also firmware viruses may hide very well!)
 - Tunneling virus: circumvents virus protection shields by hiding system calls from them so that virus actions cannot get recognised by virus protection shield.
 - Multipartite virus: uses combination of, e.g., boot sector and file virus.
 - Armored virus: makes it hard to be unravelled (e.g. changes its behaviour if ran in debugger).

Virus: The Threat Continues

- Computer viruses still common, still occurring.
- Attacks moved over time from science experiments to tools of organized crime.
 - Targeting specific companies/organisations.
 - Creating **botnets** to use as tool for spam and DDoS delivery.
 - **Keystroke logger** to grab passwords, credit card numbers.
 - Also hardware keyloggers (implanted in keyboards or USB cable) used, but requires physical access (=not a virus approach).
- Why is MS Windows the target for most attacks?
 - Everyone is an administrator (in some versions) & Most common.
 - Linux or Mac OS X are not necessarily safer: just less common
⇒ less worthwhile to attack.
 - On Mobile phones (Android) and many WLAN routers Linux is most common:
⇒ worthwhile to attack.
 - **Monoculture** considered harmful.

Trap Door/Back Door and Logic Bomb

- **Back Door (or Trap Door):**
 - Programmer leaves a security hole in a program that can be later exploited by him/her (e.g. by entering a special password) or someone else.
 - Nicer variant: Easter egg, e.g. a harmless game hidden in some office program that is activated via, e.g., some key combination.
- **Logic Bomb:**
 - Programmer leaves code that automatically initiates a security incident under certain circumstances (e.g. programmer not anymore listed as employee in the company's data base).
- Both are hard to detect in an executable program.
 - Easier to detect if source code is available, but still whole code needs to be searched for.

Stack and Buffer Overflow Attacks

- A Trojan horse, Virus, Back Door or Logic Bomb require access to a system to be able to place and start that program on the machine.
 - E.g. by sending the program as an e-mail attachment and requesting the recipient to open it.
- Overflow attacks target already running processes (or the kernel) to make them execute malicious code with the privileges of that process.
 - Typical target: a server process running in a super user domain.
 - If the targeted server process is listening for network connections, an exploit may even be possible from outside the system via the network.
 - Alternative target: a web browser that is used to visit a web site that exploits an overflow vulnerability of the web browser/browser plug-in (e.g. Adobe Flash).
- General idea of overflow attack:
 - Provide a larger input than expected (overflow of input field/input buffer, command-line argument, or field in some file).
 - Process will copy larger input to some internal data structure, thus overwriting data located behind that data structure.
 - By designing the content of the larger input appropriately, the behaviour of the attacked process will change and can be used maliciously.

Stack Overflow Attack

- Makes use of the fact that
 - local variables (as generated by a compiler) are stored on the stack.
 - the return address of a function is stored (by CPU) on the stack just following the local variables (→stack layout on next slide).
 - due to virtual memory, the address space of every process starts at the same logical address and thus, the whole memory layout of a process (incl. stack area) is quite predictable.
 - many programs are vulnerable: buffers (=local variables) of fixed size are used for processing input data and when copying the variable size input data into the fixed size buffer, e.g. the C `strcpy` function is used (instead of `strncpy` that would only copy a limited number of bytes):
 - `strcpy` copies as many bytes as the input data has, thus in the worst case overwriting the return address on the stack.
 - By preparing the input data in a way that the overwritten return address points to own code (typically code contained in the passed input data as well), malicious code can be executed.
 - Due to predictable memory layout, address where passed malicious code will be copied to on the stack is easy to predict.

Stack Overflow Attack: Example

- Vulnerable C code:

```
#include <string.h>

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

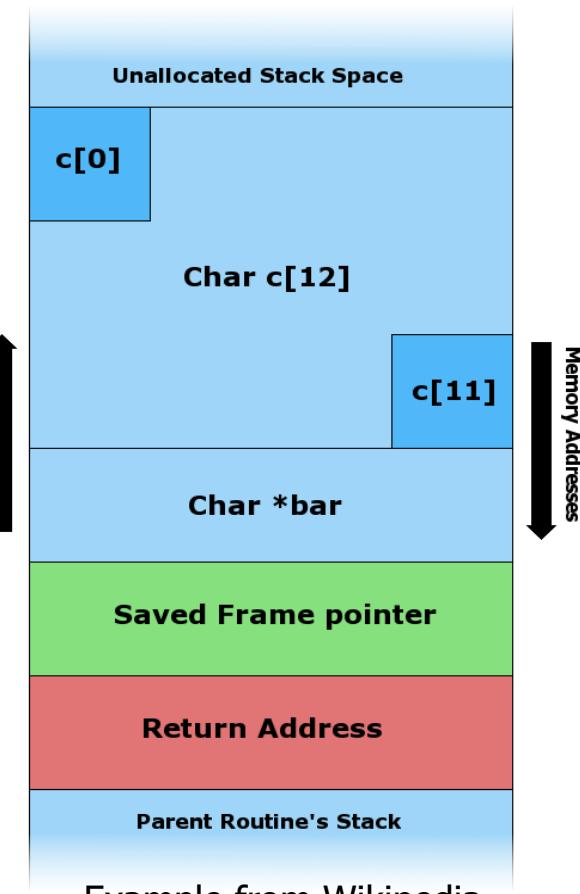
int main (int argc, char **argv)
{
    foo(argv[1]);
}

Pass pointer to first command line
parameter string (which may have
arbitrary size) to function foo.
```

Local variable buffer of size 12 (located on the stack) to store passed parameter for later use.

String copy: if the passed string `bar` contains more than 12 characters, `strcpy` will overflow buffer `c`.

- Stack layout:

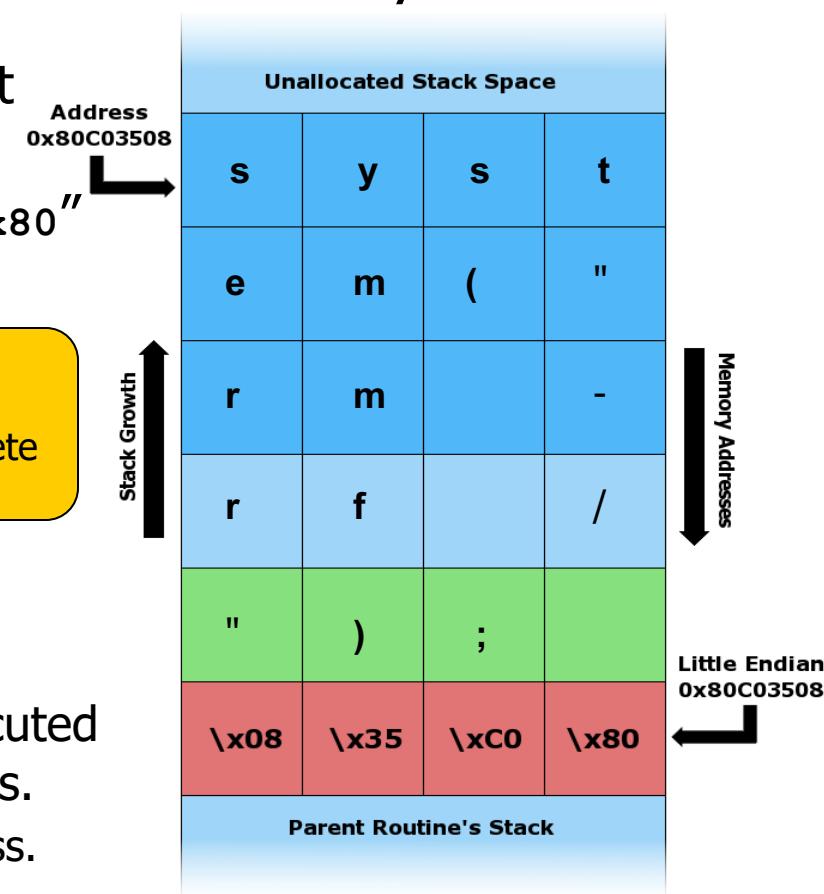


Stack Overflow Attack: Exploit

- An intruder who learned from running the targeted process that the array `c[12]` will be, e.g., typically located at memory address `0x80C03508`, passes "`system("rm -rf /"); \x08\x35\xc0\x80`" as the first command line argument.
- Stack layout:

In practise, not the C source code `system("rm -rf /");` needs to be used, but the corresponding machine code instructions. (Also `rm` command will in practise refuse to delete the root directory.)

- As a result, the return address will be overwritten with value `0x80C03508`.
 - The code at `0x80C03508` will get executed as soon as the current function returns.
 - Code will run in user domain of process.



Defeating Stack Overflow Attacks

- As an application programmer (C/C++, Java not affected):
 - Always use `strncpy` instead of `strcpy` for copying input values of variable size! (Problem: string termination null byte may not get copied.)
- As an operating system designer:
 - Address space layout randomisation (ASLR): make address space less predictable (e.g. use different start addresses when loading each program).
- CPU designer together with operating system designer:
 - Nonexecutable stack / Data Execution Prevention (DEP): data contained in pages marked as nonexecutable will not get executed as instructions.
 - Recent CPUs provide a special NX (No eXecute) bit in PMMU page table entries.
 - Operating System configures PMMU page table entries to set NX bit for stack, heap and global data buffers of a process.
 - Still, there are enough pages that contain legal executable code where the NX bit is not set. E.g. the libc mapped into the address space of a process ([Return-to-libc attack](#)): Overwrite return address on stack just by address of some libc instruction, e.g. `system()` function to execute an arbitrary program. Or, e.g. let a Just-in-Time compiler generate code ([JIT Spraying](#)).

14.8 System and Network Threats

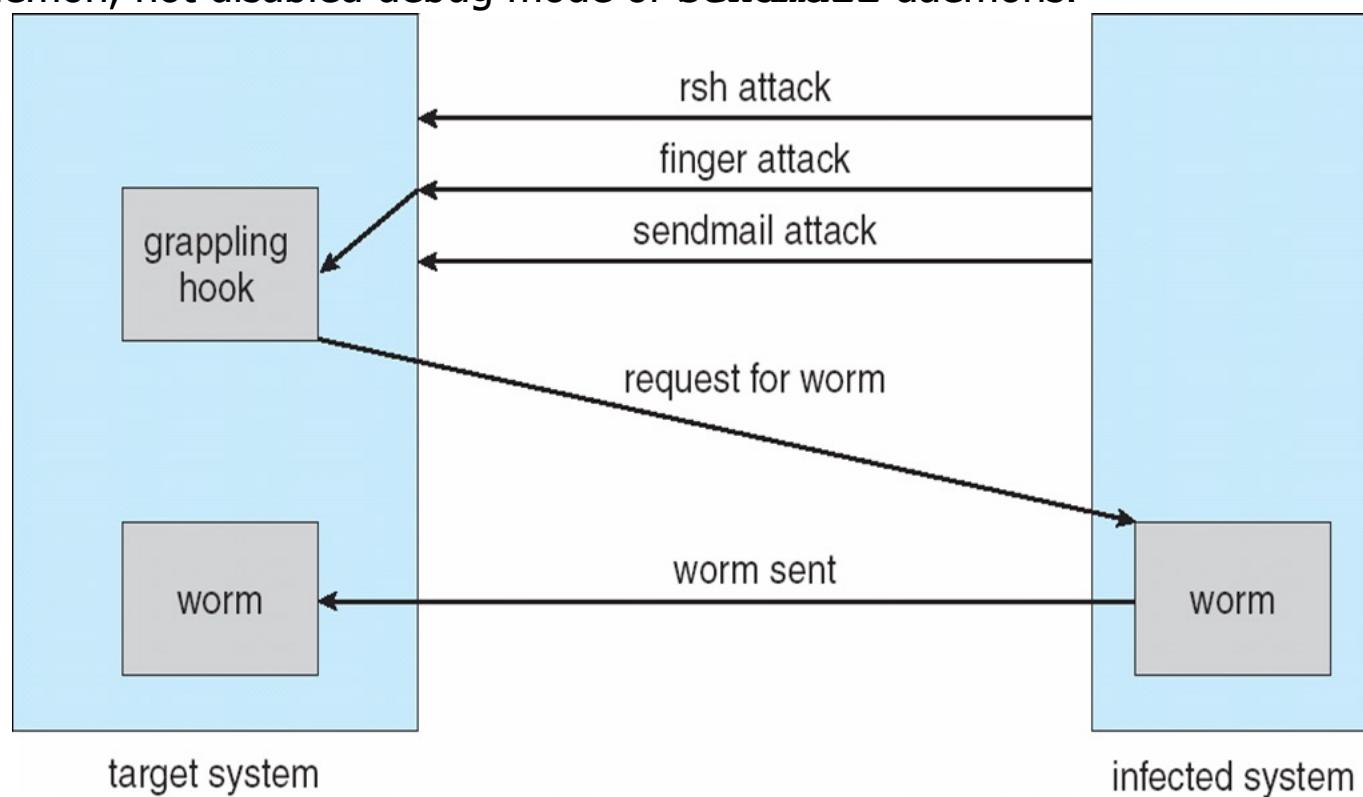
- System and network threats abuse operating system services and network resources.
 - In the past, OSes started by default a lot of services (e.g. FTP daemon for file transfer, telnet daemon for remote login etc.) to make systems easy to use out of the box.
 - These service could be target of an attack.
 - To reduce the **attack surface**, nowadays such services are disabled by default ("secure by default") and need specifically be enabled.

System and Network Threats: Worms

- **Worms:** While computer viruses infect only local resources (e.g. files, bootsector) and require that a human user “carries” them (e.g. by exchanging infected files via a USB key or by sending them as an e-mail attachment), a worm makes itself use of the network to replicate itself to remote machines.
 - Typically, some other vulnerability (e.g. buffer overflow) is used to enter a remote machine (“bootstrapping” or “grappling hook”). From there, it downloads the remainder of the worm from the source machine to the hooked system.

System and Network Threats: Worms

- The first severe worm on the Internet (1988) by Robert Morris.
 - No code to damage system, just used up resources.
 - Used remote shell (**rsh**) setups of users that had allowed login to other machines without having to provide password, stack-overflow vulnerability of **finger** daemon, not disabled debug mode of **sendmail** daemons.



System and Network Threats: Port Scanning

- Port scanning: Not an attack on its own, but part of preparation to detect vulnerabilities to attack: A port scanner tries to connect to well-known port numbers of a remote machine.
 - Either no service is listening on that port of the remote machine (then, simply the next port number is probed) or a service answers.
 - Typically, the answer contains information about the service implementation (e.g. name and version of service implementation) from which conclusions can be drawn, e.g.: is this a vulnerable service implementation? which OS is used?
 - If a vulnerable service implementation is found, it could get attacked in a next step.
 - Permanently running local tools may detect port scans (because, a lot of port numbers are contacted within a short period of time) and decide not to answer anymore to requests from the originating machine.
 - Often “zombie systems” are used to originate a port scan: systems that were vulnerable and are thus infected. If this machine is recognised by a port scan detection tool, simply the next zombie system is used to continue port scan.

System and Network Threats: Denial of Service (1)

- Denial-of-Service (DoS) attack: not aimed at gaining information or stealing resources, but rather at preventing legitimate use of a system. (E.g. to achieve that a company cannot gain any revenues by legitimate users that are not able to use the services of that company anymore.)
 - No vulnerability is used, but simply a lot of requests are made to a service: these lead to usage of a lot of computing resources or network resources. In either case, the service originally provided becomes unreachable.
 - Typically, the requests are designed in a way that only few computing and network resources are required by the originator but lead to huge resource consumption at the target. (E.g. small request leading to huge answer.)

System and Network Threats: Denial of Service (2)

- In the beginning it may not be possible to distinguish whether a service just became very popular or a DoS attack has started.
 - Imagine a link to the web server of your small company is published in some popular newspaper (or web magazine such as slashdot.org): a lot of people will follow that web link and thus overload your small web server.
- Many other ways of DoS than exhausting resources, e.g.:
 - Issue several login requests for users using wrong password
⇒ future login attempts get blocked because an attack was detected
⇒ user cannot login any more.
- **Distributed Denial-of-Service attack (DDoS):** many different machines (e.g. zombie systems) take part in a co-ordinated DoS.
 - While a DoS can be simply defeated by blocking the single originator, a DDoS is almost impossible to defeat.

14.9 Cryptography as a Security Tool

- Within a local computer, the OS can reliably determine sender and receiver of interprocess communication (=the IDs of local processes).
- In a networked system, where many other untrusted machines are involved in forwarding a message.
 - Sender and receiver cannot be reliably determined by the OS of receiver and sender processes respectively. (Protection schemes based on simple user IDs as usable by OS within local computer not possible.)
 - Transmitted data may be eavesdropped or modified.
- Cryptography: broadest security tool available!
 - Source, destination, and contents of messages cannot be trusted without cryptography.
 - Means to constrain potential senders (sources) and / or receivers (destinations) of messages.
- Based on secrets (keys).

Encryption

- Constrains the set of possible receivers of a message.
- Encryption algorithm consists of:
 - Set K of keys,
 - Set M of messages,
 - Set C of ciphertexts (encrypted messages).
 - Encryption function $E : K \rightarrow (M \rightarrow C)$. That is, for each key $k \in K$, E_k is a function for generating ciphertexts C from messages M .
 - Both E and E_k for any k should be efficiently computable functions.
 - Otherwise, encrypting huge files would take too long time or encryption of real-time communication too slow.
 - Decryption function $D : K \rightarrow (C \rightarrow M)$. That is, for each key $k \in K$, D_k is a function for generating messages M from ciphertexts C .
 - Both D and D_k for any k should be efficiently computable functions.
 - Same reasons as for encryption.

Encryption (continued)

- An encryption algorithm must provide an essential property:
Given a ciphertext $c \in C$, a computer can compute the message m such that $E_k(m) = c$ only if it possesses the key k .
 - In other words: a computer holding the key k can decrypt ciphertexts to the plaintexts used to produce them, but a computer not holding the key k cannot decrypt ciphertexts.
 - Since ciphertexts are generally exposed (for example, sent via the network), it is important that it is **not possible to derive the key k from the ciphertexts**.

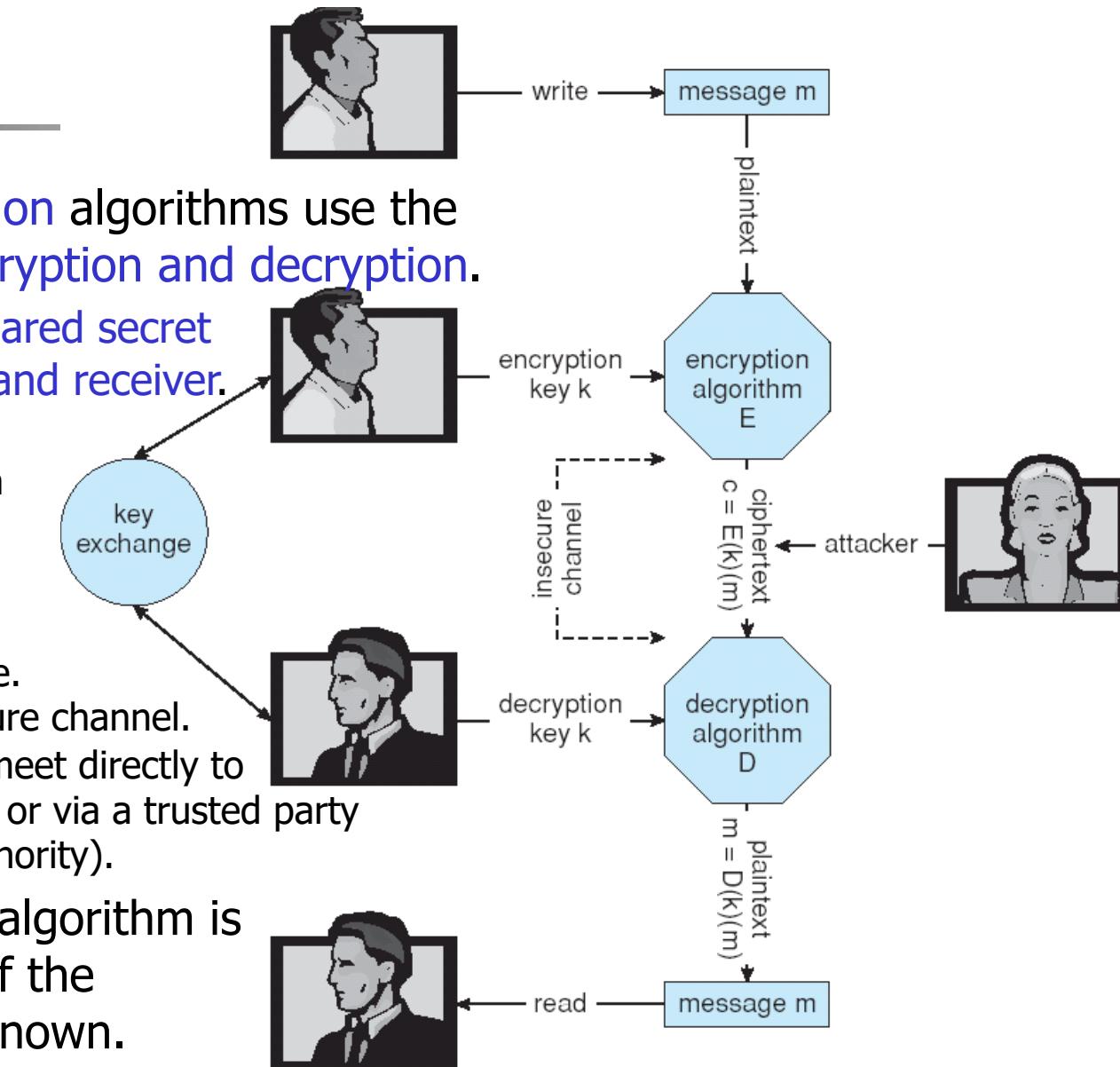
Symmetric Encryption

- Symmetric encryption algorithms use the same key k for encryption and decryption.

- Key must be a shared secret between sender and receiver.

- Allows secure communication over insecure channel.
 - Key exchange out-of-band, i.e. not over insecure channel. Either people meet directly to exchange keys or via a trusted party (certificate authority).

- A good encryption algorithm is unbreakable even if the algorithm itself is known.



Symmetric Encryption

DES

- DES (Data Encryption Standard) was most commonly used symmetric block-encryption algorithm (block cipher):
 - Encrypts a block of data (64 bits) at a time,
 - Created by US National Institute of Standards and Technology (NIST):
 - National Security Agency (NSA) took care that initial key size of 64 bit was reduced to 58 bit, thus making decryption by brute-force attacks (=simply try all possible keys) easier (e.g. brute-force attack by NSA).
 - Now considered **insecure** due to too short key size.
⇒ **Triple-DES considered more secure:**
 - DES algorithm is simply used 3 times (3 times more computation time) using 2 or 3 different keys (key space grows by 2^{58} or 2^{116})
 - For example: $c = E_{k3}(D_{k2}(E_{k1}(m)))$

Symmetric Encryption

AES

- 2001 NIST adopted new block cipher: Advanced Encryption Standard ([AES](#))
 - considered secure!
 - Keys of 128, 192, or 256 bits size (256 bits recommended). Works on 128 bit blocks.
 - Problems if message m does not fit exactly block size:
 - Longer than block size: using the same key for the next block may make block ciphers less secure.
 - Shorter than block size: message must be padded to match block size: padding all with a known value (e.g. just 0) may make block ciphers less secure.
- ⇒ [Modes of operation](#): Specify how to use a block cipher for messages that do not exactly fit block size.
- Typically based on a unique binary sequence ([initialization vector](#)) for each encryption operation. The initialization vector has to be non-repeating (sometimes random as well). The initialization vector is used to ensure distinct ciphertexts are produced even when the same plaintext is encrypted multiple times independently with the same key.
 - An initialization vector must never be reused with the same key.

Symmetric Encryption

RC4

- Stream ciphers handle an arbitrary number of bits (in contrast to the block ciphers based on a fixed number of bits per block).
- RC4 is most common symmetric stream cipher, but known to have vulnerabilities.
 - Encrypts/decrypts a stream of bytes.
 - E.g., used in wireless transmission WEP WiFi security standard or in the Internet Protocol Secure Sockets Layer (SSL) successor Transport Layer Security (TLS)).
 - A key k is used as input to a pseudo-random-bit generator:
 - Generates an infinite keystream bit sequence: used to XOR with message m to generate ciphertext c .

Asymmetric Encryption

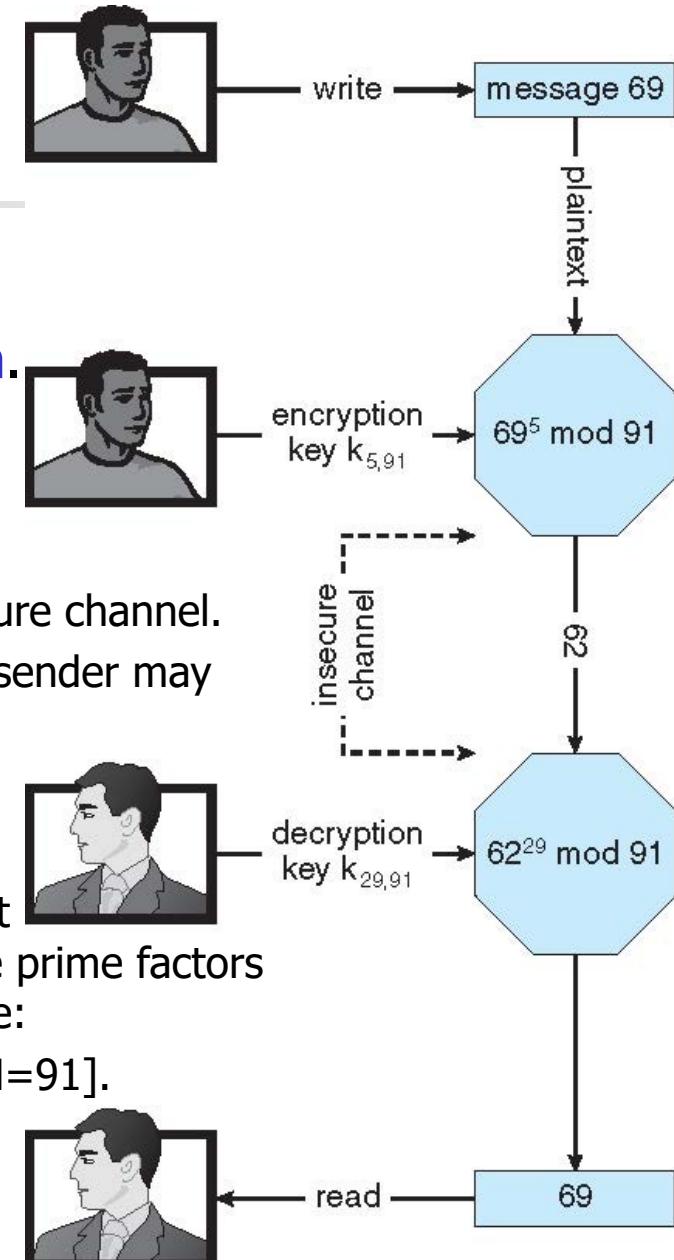
- Asymmetric encryption algorithms use different keys for encryption and decryption.

- Key k_e for encryption may be public, only key for decryption k_d needs to be private: "public key encryption".

- Public key can be exchanged even over insecure channel.
 - Every receiver has its own private key, every sender may use the public key of the receiver.

- Most common asymmetric encryption algorithm: RSA by Rivest, Shamir, Adleman.

- Based on prime numbers (security due to that no efficient algorithm is known for finding the prime factors of a number) and modulo arithmetic. Example:
 - Public key [$k_e=5, N=91$], private key [$k_d=29, N=91$].
 - Encryption: $c=E_{5,91}(m):=m^5 \bmod 91$
 - Decryption: $m=D_{29,91}(c):=c^{29} \bmod 91$



Asymmetric Encryption

RSA

- k_e is the **public key** used for encryption (together with public N),
- k_d is the **private key** used for decryption (together with public N),
- N is the **product** of two large, randomly chosen **prime numbers** p and q .
- Encryption algorithm is: $E_{k_e, N}(m) = m^{k_e} \bmod N$,
 - where k_e and k_d satisfy $k_e k_d \bmod (p-1)(q-1) = 1$
- The **decryption** algorithm is then: $D_{k_d, N}(c) = c^{k_d} \bmod N$.
- Example (see previous slide):
 - Choose prime numbers: $p = 7$, $q = 13$ (in practise: large, e.g. 512 bits each)
 - We then calculate $N := p q = 7 * 13 = 91$ and $\varphi(N) := (p-1)(q-1) = 72$.
 - We next select k_e relatively prime to 72 (i.e.: $\gcd(k_e, 72) = 1$) and $k_e < 72 \Rightarrow k_e = 5$.
 - Finally, we calculate k_d such that $k_e k_d \bmod 72 = 1 \Rightarrow k_d = 29$.
 - We now have our keys: Public key $k_{e,N} = [5, 91]$; Private key $k_{d,N} = [29, 91]$.
 - Computationally infeasible to derive the private key (derived from p and q) from the public key (even though $N (= p q)$ is shared between private and public key.)
⇒ Public key can be widely disseminated: everyone can encode a message, but only private key owner can decode.

Asymmetric Encryption Applications

- The open source tool PGP/GnuPG uses asymmetric encryption!
 - Asymmetric encryption is much more compute intensive (=slower) than symmetric encryption (=RSA & PGP/GnuPG considered secure).
 - E.g. RSA: calculate powers such as $c = E_{5,91}(m) := m^5 \text{ mod } 91$ and $m = D_{29,91}(c) := c^{29} \text{ mod } 91$. (≥ 1024 bit numbers involved, better ≥ 2048 bit.)
 - – in contrast to the fast, simple operations (often simple mappings/ permutations) involved in symmetric encryption.
- ⇒ Asymmetric encryption is typically not used for bulk data encryption. But rather to encrypt small amounts of data, e.g.:
- Keys of fast symmetric encryption algorithms: can then be send via asymmetric encryption over an insecure channel.
 - Checksums (hashes) of messages: can be encrypted via asymmetric encryption to be able to verify that a message has not been changed and to be sure about the sender of that message.
 - Nice RSA property: If private key is used for encryption (e.g. of the message hash), everyone having the public key can decrypt.

Authentication of Messages

Hash functions

- Authentication of Messages based on hash functions: creates a number (message digest/hash value) from a message m , i.e. some sort of checksum.
 - Long message mapped to a short number.
- Hash function H must be collision resistant on m :
 - Must be infeasible to find an $m' \neq m$ such that $H(m) = H(m')$, i.e. an attacker must not be able to modify a message m to m' in a way that the hash value stays the same. (While there will be cases where $m' \neq m$ with $H(m) = H(m')$, it is infeasible to design such an m' intentionally.)
- Common message-digest functions include MD5, which produces a 128-bit hash, and SHA-1, which outputs a 160-bit hash.
 - However, security flaws in MD5 and SHA-1 have been found: SHA-2 or SHA-3 are now recommended!

Authentication of Messages

Digital Signatures

- Authentication of messages:
 - A sender sending m calculates $h := H(m)$ and sends $[m, h]$.
 - A receiver receiving $[m, h]$ calculates $H(m)$ again:
if $h = H(m)$, then m has not been modified!
- Of course, an attacker that modifies m in $[m, h]$ to m' may also recalculate $h' := H(m')$ and then send $[m', h']$.
- ⇒ A sender uses RSA asymmetric encryption to encrypt h using his/her private key (i.e. no one else can do this).
 - A receiver uses the public key of the sender to decrypt the encrypted h (by design of RSA possible if private and public key match).
- ⇒ An attacker will not be able to create an encrypted h' .
- ⇒ Sender of the message must be the one with the private key matching the public key.
- ⇒ **Digital signature:** provable who signed the message + message content not modified.
- Message m may either be encrypted as well or non-encrypted (i.e. everyone can read the message content and use h to verify who is the sender of the message (by using the public key of the sender) and to verify that the message was not modified (by calculating $H(m)$ again and comparing it to the h from the digital signature)).

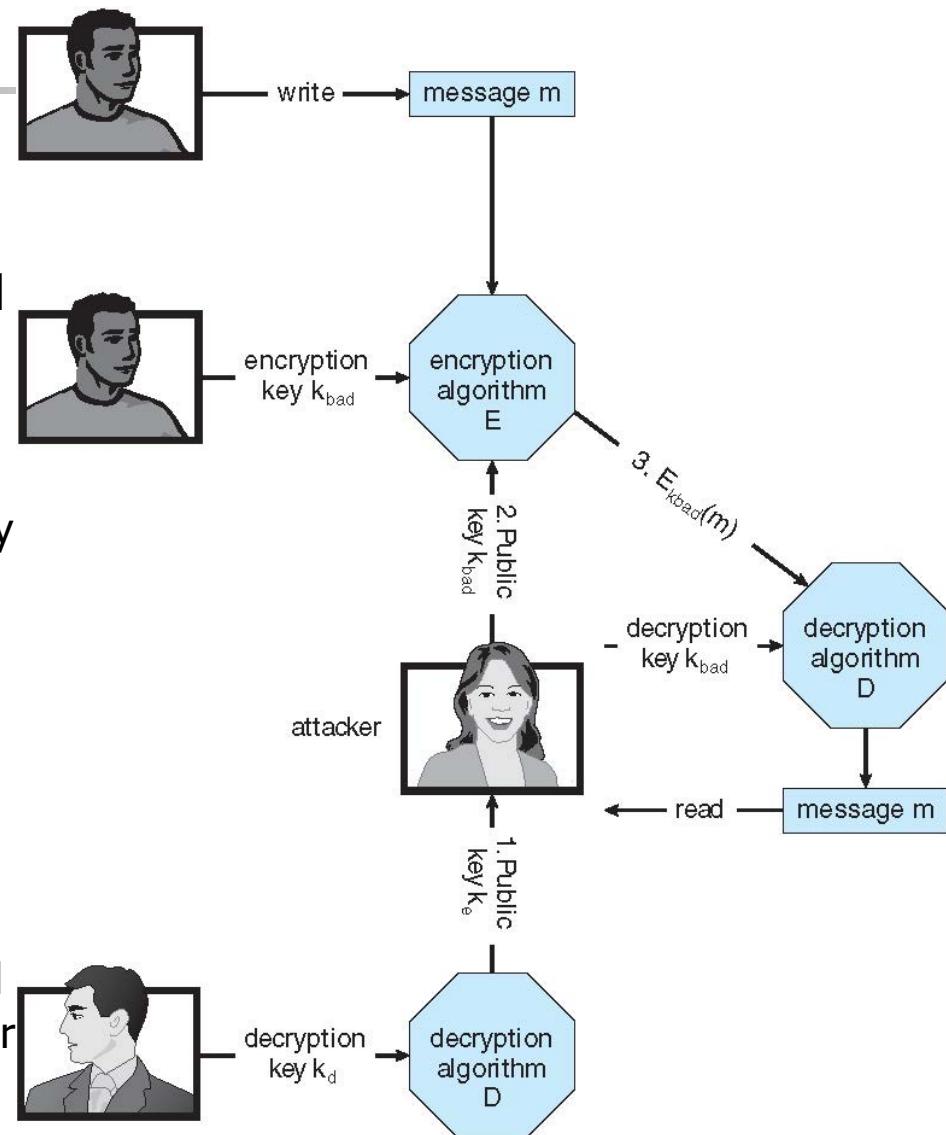
Key Distribution

- Symmetric keys must be exchanged between sender and receiver out-of-band (e.g. via a personal meeting):
 - A different key for each pair of users,
 - For more security, change keys frequently.

⇒ Can be annoying.
- Using asymmetric encryption, symmetric keys can be exchanged via an insecure channel.
 - Use public key of a user to encrypt and send the symmetric key.
 - Collection of all the public keys you know: “key ring”.
 - Still, a man-in-the-middle attack is possible (→next slide):

Man-in-the-middle Attack on Asymmetric Cryptography

- An attacker creates a public key for another person (thus the attacker owns the matching private key) and distributes this fake public key and thus makes others use that public key.
 - E.g. an Internet router might modify (step 2.) public keys distributed via the Internet (step 1.)
- An attacker may now intercept and decrypt messages encrypted with the fake public key (step 3.)
 - Even possible to change the message, encrypt it with the real public key of a receiver and forward it to the receiver so that the receiver does not even notice the attack.



Digital Certificates/ Certificate Authorities

- The man-in-the-middle-attack would not be possible if the distributed public key would have been digitally signed.
 - Then, any tampering with the public key data would be detectable.
 - Chicken-and-egg problem: which key to use to verify the digital signature?
 - Digital certificates: a public key signed by a trusted party.
 - Everyone has to know the public key of that trusted party (needs to be pre-installed on your system).
 - Certificate authorities sign public keys of others using their own trusted public key.
 - Public keys of certificate authorities pre-installed in web browsers.
 - But: web browser can also be tampered...
 - If you do not trust central certificate authorities (e.g., because some secret service of a Government might be involved):
 - Meet personally and exchange your public keys (or hashes of them): Physically attend a PGP key signing party.

Implementation of Cryptography

- Main insecure channel: the Internet.
 - Network protocols based on layers (OSI model).
- Encryption can be added on various layers, typically:
 - Transport Layer Security (TLS) and its predecessor Secure Socket Layer (SSL) at the Presentation or Transport layer.
 - E.g. https (=http over SSL/TLS) in your Web browser.
 - Virtual Private Networks (VPNs) / IPsec at the Network layer.
 - Everything transmitted in IP packets (typically TCP & UDP) gets encrypted.
 - Involves digital certificates.
 - E.g. web browser gives you a warning if certificate is not from a known certificate authority, but self-signed.

Source:

http://en.wikipedia.org/wiki/OSI_model

OSI model

7. Application Layer

NNTP · SIP · SSI · DNS · FTP ·
Gopher · HTTP · NFS · NTP · SMPP ·
SMTP · SNMP · Telnet · Netconf ·
(more)

6. Presentation Layer

MIME · XDR · TLS · SSL

5. Session Layer

Named Pipes · NetBIOS · SAP · L2TP ·
PPTP · SPDY

4. Transport Layer

TCP · UDP · SCTP · DCCP · SPX

3. Network Layer

IP (IPv4, IPv6) · ICMP · IPsec · IGMP ·
IPX · AppleTalk

2. Data Link Layer

ATM · SDLC · HDLC · ARP · CSLIP ·
SLIP · GFP · PLIP · IEEE 802.3 ·
Frame Relay · ITU-T G.hn DLL · PPP ·
X.25 · Network Switch · DHCP

1. Physical Layer

EIA/TIA-232 · EIA/TIA-449 ·
ITU-T V-Series · I.430 · I.431 · POTS ·
PDH · SONET/SDH · PON · OTN ·
DSL · IEEE 802.3 · IEEE 802.11 ·
IEEE 802.15 · IEEE 802.16 · IEEE 1394
· ITU-T G.hn PHY · USB · Bluetooth ·
Hubs

This box: [view](#) · [talk](#) · [edit](#)

14.10 User Authentication

- Already within a local computer, it is crucial for an OS to identify users correctly, as protection systems depend on user ID.
- User identity most often established through **passwords** – can be considered a special case of a secret key.
 - Also can include something user has
 - (e.g. phone SIM card: additional code sent via SMS or Rafræn skilríki)
 - and/or a biometric user attribute
 - (e.g. fingerprint, retina pattern)
- ⇒ multifactor authentication
- Passwords may also be one-time passwords, e.g.:
 - Generated by a token generator (e.g. Auðkenni), from a printed list (e.g. Transaction Numbers TAN).

Passwords

- Passwords must be kept secret.
 - Frequent change of passwords.
 - Use of “non-guessable” passwords.
 - E.g. use random generator to generate password.
 - As this will be hard to memorise: set screensaver timeout to 1 minute and require to enter password to unlock. ⇒ You'll learn it!
- Do not re-use password for different services.
 - If you have problems memorising password for different services: use the same password, but modify it by involving some characters non-obviously derived from the service name.
 - Example (just to get an idea, invent your own): second and last character of service name and a digit derived from length of the service name minus 2.

Storing and Attacking Passwords (1)

Storing passwords encrypted

- The OS compares entered password against stored password.
 - Only if these match, user authentication is successful.
- Password should always be stored encrypted with no possibility to decrypt it!
 - I.e. encrypt entered password x using a function $f(x)$ for which no inverse function f^{-1} exists. Store this encrypted password in some password database.
 - At each login: encrypt entered password x' using $f(x')$ and compare to stored encrypted password $f(x)$.
 - ⇒ Password x itself is nowhere stored, only the encrypted $f(x)$.
 - ⇒ Even if someone breaks into a system and is able to steal the password database, the plaintext password cannot be determined.

Storing and Attacking Passwords (2)

Brute force attack on encrypted passwords

- However, if the encrypted password database has been stolen, it is still possible to use a **brute force attack** if function f is known:
 - In a loop, generate all possible combinations of character strings s_i , feed each into function f , and check whether the outcome matches the stored encrypted password $f(x)$. If yes, the password was s_i .
 - In practise, function f is known, because it is difficult to define good functions f , therefore not many different functions are used in practise, e.g. MD5.
 - MD5 is a hash function that maps any input to a 128 bit hash value.
 - However, security flaws in MD5 have been found: SHA-2 or SHA-3 are better but still rather fast to compute (brute force attacks possible →next slide)!

TODO

- Update MD5 times on next slides: using CPU & GPU, e.g.
- <http://calc.opensecurityresearch.com/>

Storing and Attacking Passwords (3)

Brute force and Rainbow tables

- Trying out all passwords using brute force takes time, e.g. in 2011:

■ Length of password	Number of possibilities	CPU time needed (MD5)
■ 4 characters	35,153,041	3 minutes
■ 5 characters	2,706,784,157	3.75 hours
■ 6 characters	208,422,380,089	12 days
■ 7 characters	16,048,523,266,853	2.5 years
■ 8 characters	1,235,736,291,547,681	195 years

Or: 100 quadcore CPUs in ½ year!
(CPU speed from 2011 – today's CPUs even faster: few days)!
- Attackers that want to avoid trying over and over again all possible combinations s_i can do this once and store for each s_i the resulting $f(s_i)$ and the plaintext password s_i in a file. Such a file is called **rainbow table**:

■ Length of password	Number of possibilities	Size of rainbow table (MD5)
■ 4 characters	35,153,041	913 MB
■ 5 characters	2,706,784,157	70 GB
■ 6 characters	208,422,380,089	5.4 TB
■ 7 characters	16,048,523,266,853	417 TB
■ 8 characters	1,235,736,291,547,681	32 PB

■ In practise, smaller rainbow tables are used: http://en.wikipedia.org/wiki/Rainbow_table

Storing and Attacking Passwords (4)

Salts

- Note: on the previous slide, it was assumed that all 77 printable ASCII characters are tried. In practise, users tend to use only a subset of them. Furthermore, not all use 7 or 8 character passwords.
- To increase the number of possible passwords (and therefore the size of the needed rainbow table), nowadays passwords chosen by a user are extended by a random number ("salt"):
 - E.g. in early Unix, each password (up to 8 significant characters) was prefixed by a random 12 bit salt before it was used as input parameter for function f . \Rightarrow Number of possible encrypted passwords (size of rainbow table) increases by a factor of $2^{12}=4096$.
 - The encrypted password (which was prefixed with the salt) is then stored together with the plaintext(!) salt in the password database.
 - If the salt would not be stored as plaintext, the OS could not check whether the entered password matches the stored one: the entered password needs again be prefixed by the same salt before it is encrypted using f and then compared to stored, encrypted & salted password.

Storing and Attacking Passwords (5)

GPUs and Brute force

- While you should Use long salts (48 bit (=+6 chars.) or 128 bits (=+16 chars.) are now common), salting defends only against rainbow table, not against brute force:
 - As the salt needs to be stored in plaintext in the password database, a brute force attack will know the salt to be used for the attack on a stolen password database.
- Even worse: Graphical Processing Units (GPUs) found on modern graphic cards can perform brute force passwords attacks much faster than CPUs.

⇒ Use functions f that take even on GPUs longer time to encrypt passwords.

 - Consider MD5, SHA-2/SHA-3 as broken with respect to password hashing!
 - Good GPU resistant functions still subject of research..
 - Good starters seem to be, e.g., bcrypt or Argon2.
- Note that rainbow tables and GPUs are only usable to attack stolen encrypted password databases – not at login prompt!

Storing and Attacking Passwords (6)

Shoulder surfing, sniffing

- Also possible: simply try passwords at login prompt.
 - After unsuccessful login, prompt shall wait a few seconds to prevent trying out many passwords in short time.
- Additional attacks to obtain password:
 - **Shoulder surfing**: look over the shoulder (or use a hidden camera) when a user enters a password.
 - Polite while someone enters a password: look away.
 - **Sniffing**: install a key logger software or hardware that records any key typed on a keyboard.
 - Key logger software acts as a keyboard device driver.
 - Requires superuser privileges – e.g. by exploiting a vulnerability.
 - Key logger hardware is soldered into a keyboard or plugged between USB keyboard cable and the computer's USB port.
 - Requires just physical access to the computer.

14.11 Summary

- **Protection:** Mechanism to control who may access which resources in a computer system (internal view).
 - Access right: permission to perform operation on an object.
 - Domain: set of access rights.
 - Processes execute in domains.
 - Domain switch possible during run-time.
 - Access matrix: conceptual model of protection. In practise realised as access control list (ACL).
- **Security:** Preserve integrity of system considering the external environment.
 - Prevent unauthorised access, destruction, and alteration of data.
 - Most common security attacks via/against programs to escalate privilege to that of the program executer.
 - While many program-based threats require local access, some of them even work remotely, e.g. overflow attacks via network ⇒ make your C/C++ code resistant!
 - Cryptography limits domain of senders and receivers and provides confidentiality.
 - User authentication typically based on passwords ⇒ store encrypted and salted!
 - Not covered: intrusion detection, virus scanners, firewalls.