



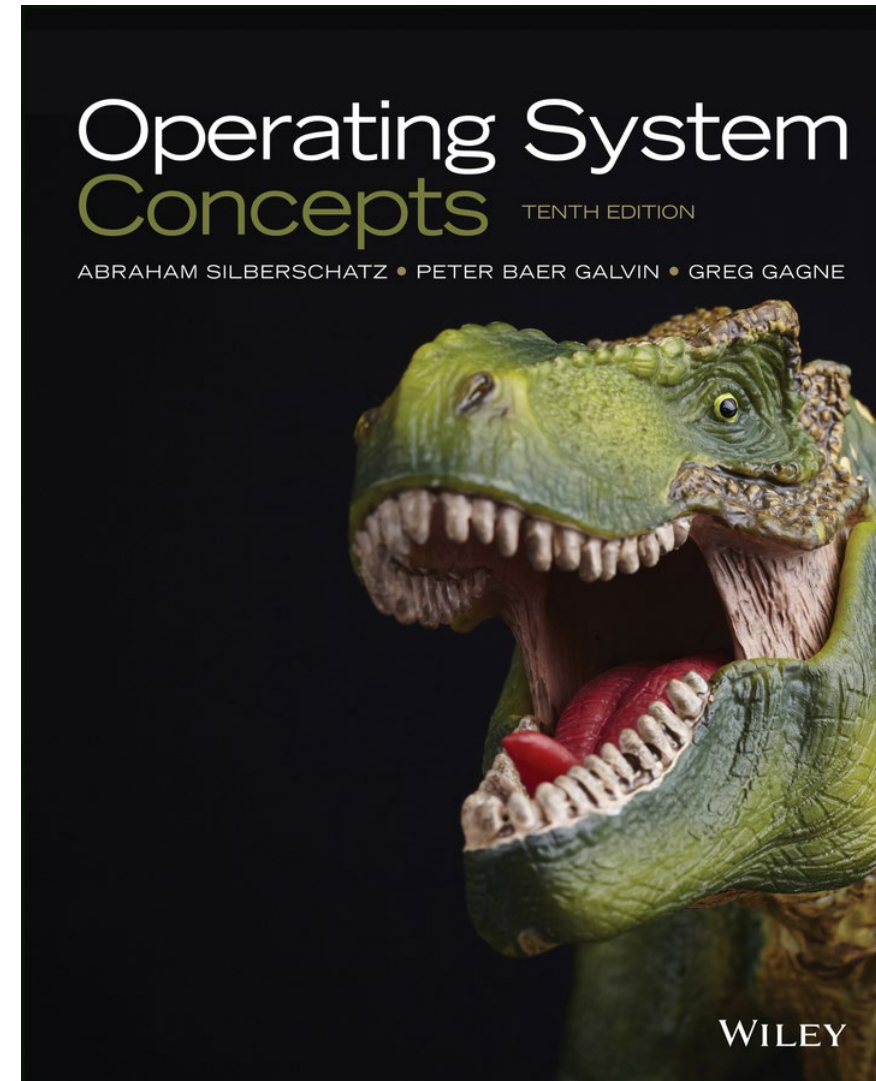
Informatik der Systeme – Chapter 10: Operating Systems

Prof. Dr. Michael Menth, Gabriel Paradzik

<http://kn.inf.uni-tuebingen.de>



- ▶ Adapted from Silberschatz, Abraham, et al.:
“Operating System Concepts”, 10th Edition,
© 2018, Wiley Global Education US





► Chapter 10

- OS Services and Kernel/User Space
- Process Management
- Process Synchronization
- File Systems



Informatik der Systeme – Chapter 10.1: Operating Systems: OS Services and Kernel/User Space

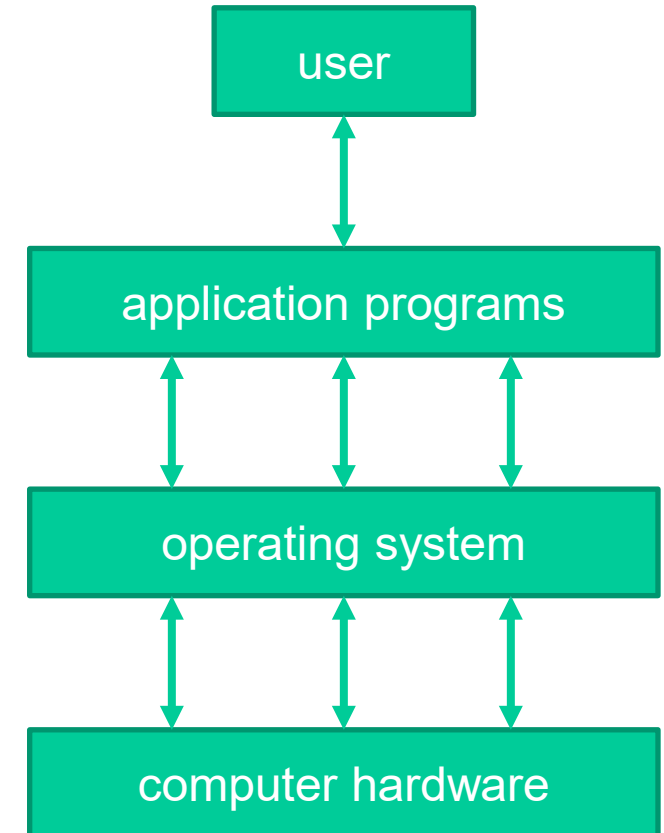
Prof. Dr. Michael Menth, Gabriel Paradzik

<http://kn.inf.uni-tuebingen.de>



What is an operating system (OS)?

- ▶ Acts as intermediary between the user and the computer hardware
- ▶ Provides an environment in which the user can execute programs conveniently
- ▶ Hardware
 - CPU, memory, I/O are the basic computing resources
- ▶ Programs
 - E.g., web browser, text editors, mail clients
 - OS defines the ways in which resources are used



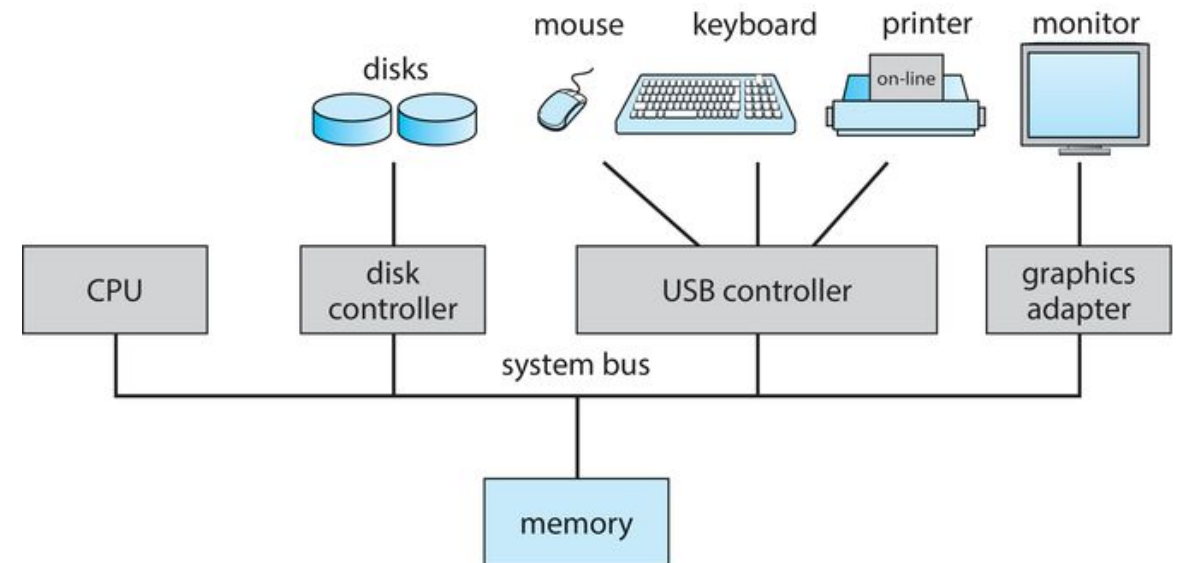


- ▶ Started with research
- ▶ Quickly moved to military use
 - E.g., trajectory computation and code breaking
- ▶ General purpose mainframes
 - Birth of operating systems
- ▶ Personal computers (PCs)
 - Operating systems needed to be user-friendly
- ▶ Laptops, smartphones and embedded devices



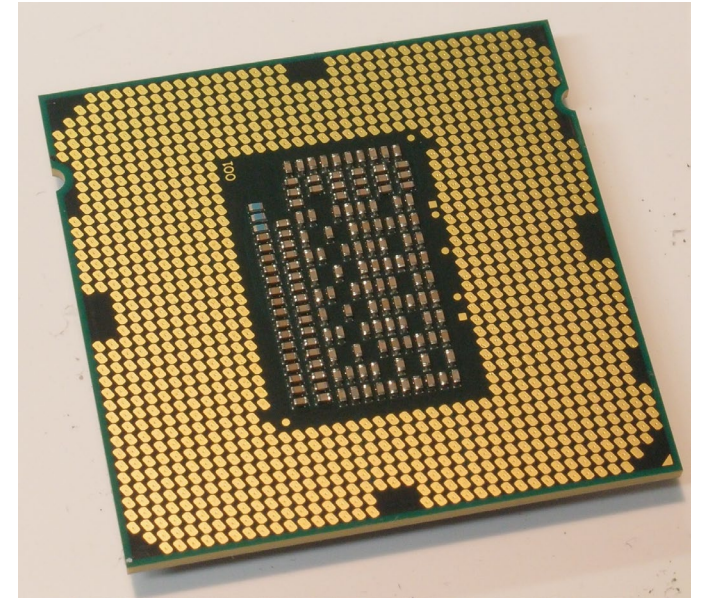
- ▶ A modern computer system consists of one or more CPUs and connected devices
 - Memory, Disks, USB devices, graphics card, ...
 - E.g., mouse needs to send position change to the CPU, so the operating system can calculate the new mouse position
 - OS wants display the new mouse position on the screen. It sends the information to the graphics card, which outputs the video signal

- ▶ Devices need to communicate with each other efficiently
 - They communicate through a bus
 - All devices share the same medium
 - Communication on this shared medium must be coordinated





- ▶ CPU chips communicate via pins with other devices
- ▶ Usage
 - Addresses
 - Data
 - Control
- ▶ Number of address pins n determine addressable memory
 - 2^n addresses
- ▶ Number of data pins m determine data transmission speed
 - Transmission of m bits in parallel
- ▶ Special control pins
 - Signals
 - Power
 - Clock
 - Bus control and arbitration



Pins on Core i7-2600 SR00B
[Source](#)



- ▶ Master: active role, initiates communication over bus
- ▶ Slave: passive role, waits to be contacted by master
- ▶ Devices can be both master and slave

Master	Slave	Example
CPU	Memory	Fetching instructions and data
CPU	I/O device	Initiating data transfer
CPU	Coprocessor	CPU handing instruction off to coprocessor
I/O	Memory	DMA (Direct Memory Access)
Coprocessor	CPU	Coprocessor fetching operands from CPU

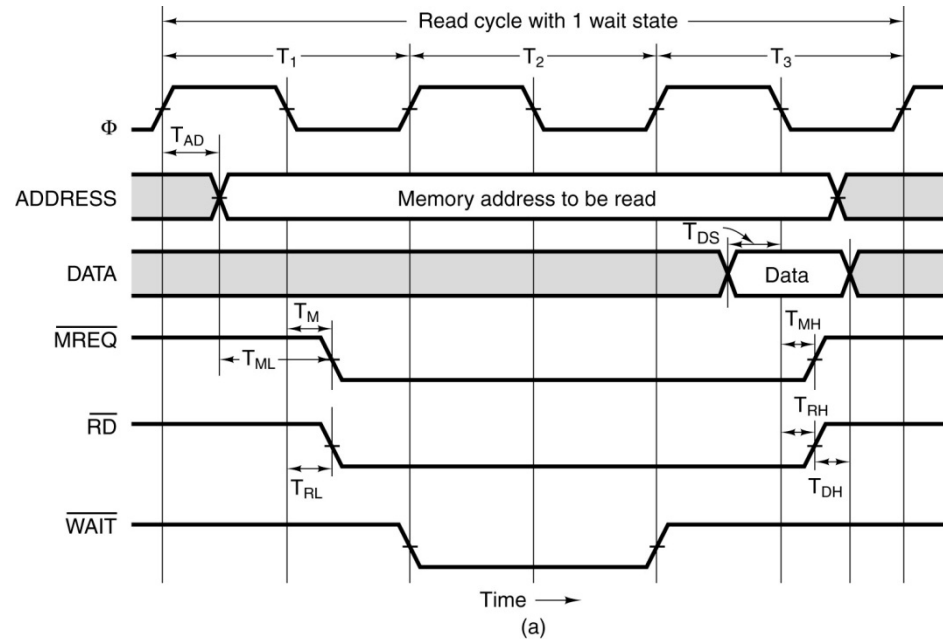
- ▶ Bus driver: amplifies digital signal for transmission (for master)
- ▶ Bus receiver: receives digital signals (for slave)
- ▶ Bus transceiver: for combined master and slave



- ▶ Synchronous bus (e.g., processor-memory buses)
 - Includes a clock in the control lines and has a fixed protocol for communication that is relative to the clock
 - Advantage: involves very little logic and can run very fast
 - Disadvantages
 - Every device communicating on the bus must use same clock rate
 - To avoid clock skew, synchronous busses cannot be long if they are fast
- ▶ Asynchronous bus (e.g., I/O buses)
 - Not clocked, so it requires a handshaking protocol and additional control lines (ReadReq, Ack, DataRdy)
 - Advantages
 - Can accommodate a wide range of devices and device speeds
 - Can be lengthened without worrying about clock skew or synchronization problems
 - Disadvantage: slow(er)

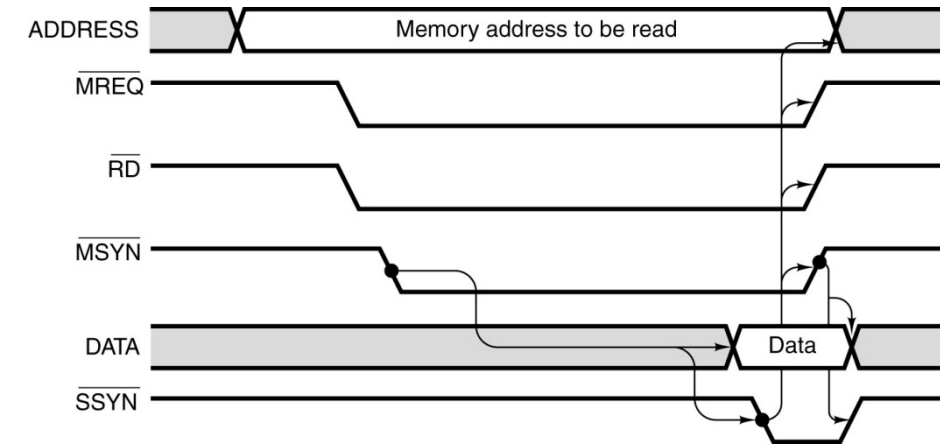


► Synchronous bus



- Transmission speed depends on clock
- Performance determined by protocol

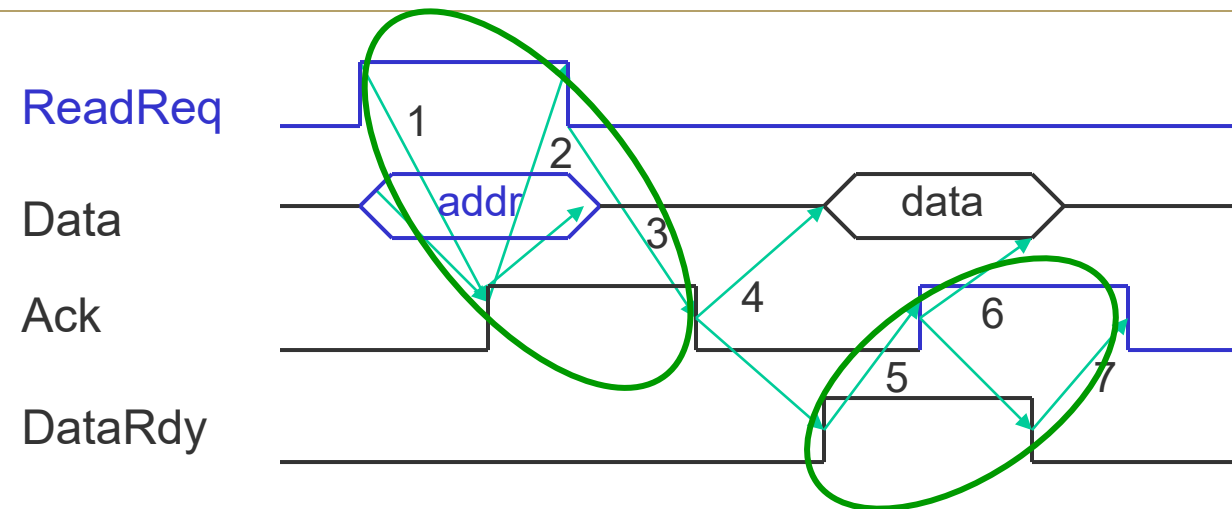
► Asynchronous bus



- Transmission speed depends on how fast master and slave react
- Performance can be improved by new devices



Asynchronous Bus Handshaking Protocol



► Output (read) data from memory to an I/O device

1. I/O device signals a request by raising ReadReq and putting the addr on the data lines
2. Memory sees ReadReq, reads addr from data lines, and raises Ack
3. I/O device sees Ack and releases the ReadReq and data lines
4. Memory sees ReadReq go low and drops Ack
5. When memory has data ready, it places it on data lines and raises DataRdy
6. I/O device sees DataRdy, reads the data from data lines, and raises Ack
7. Memory sees Ack, releases the data lines, and drops DataRdy
8. I/O device sees DataRdy go low and drops Ack



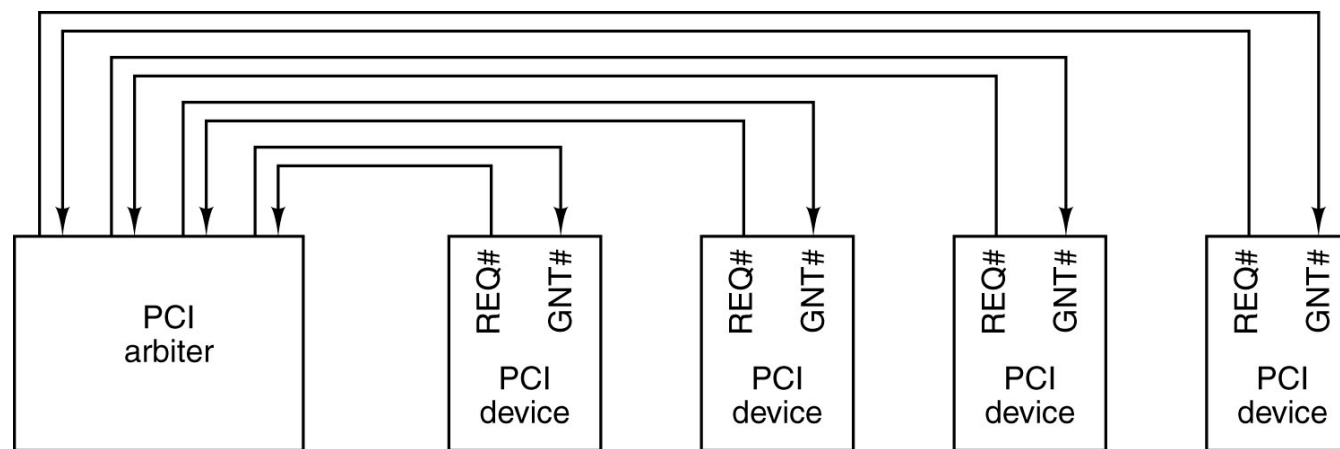
- ▶ Companies have transitioned from synchronous, parallel wide buses to asynchronous narrow buses
 - Reflection on wires and clock skew makes it difficult to use 16 to 64 parallel wires running at a high clock rate (e.g., ~400 MHz) so companies have moved to buses with a few one-way wires running at a very high “clock” rate (~2 GHz)



- Serial ATA cables (red) are much thinner than parallel ATA cables (green)



Centralized Bus Arbitration with PCI

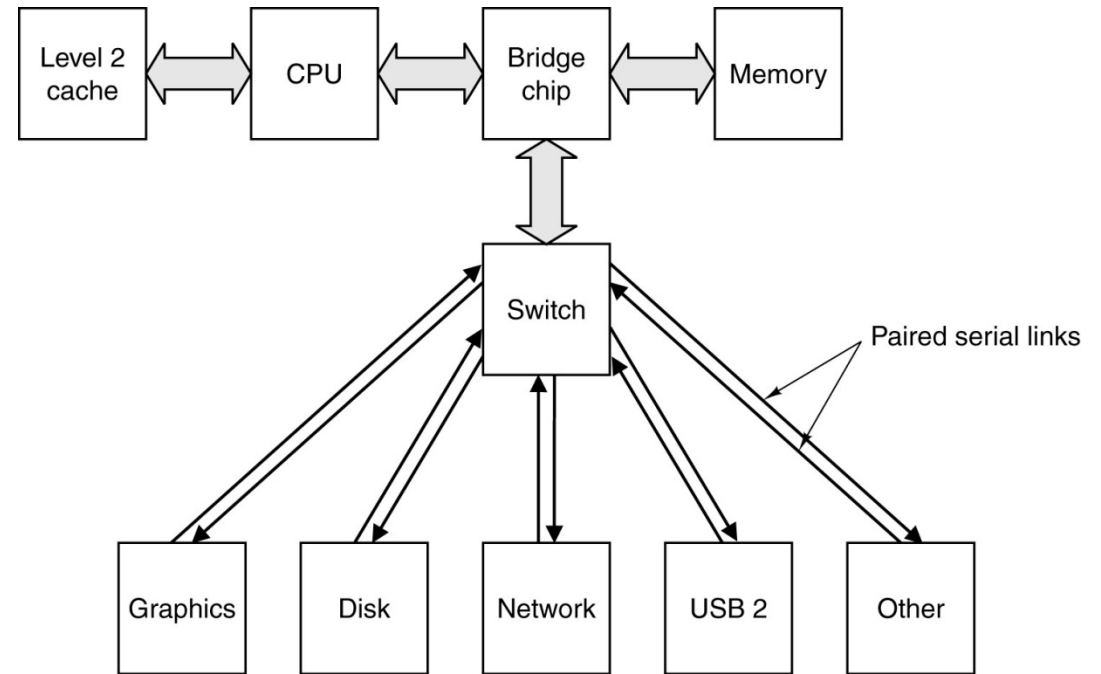


- ▶ Peripheral Component Interconnect (PCI) Bus (since 1993, v3.0 in 2004)
- ▶ Multiplex bus (data and control multiplexed over 32 or 64 bit signals)
- ▶ Synchronous (33 or 66 MHz, 1066 up to 4266 Mbit/s)
- ▶ PCI device sends request (REQ) to get bus
- ▶ Centralized arbiter sends grant (GNT) „Master-slave principle“
 - Valid for single transaction
 - Can be very long, arbiter can remove GNT
- ▶ Scheduling not defined: round robin, priority schemes, implementation of fairness
- ▶ Devices „hot pluggable“

- Old bus technology for PCs between 1991 and 2004
- PCI uses point-to-point lines for communication between masters and arbiter.
- Data are exchanged among devices via shared lines (not visible in figure).



- ▶ Since 2003, v4.0 in 2015
- ▶ Central switch, serial links
- ▶ Completely different from conventional busses
- ▶ More similar to switched Ethernet (local network)
- ▶ Packets with control headers instead of control lines
- ▶ Packets equipped with error correction code yield more reliability
- ▶ Links up to 50 cm allow more flexible system design
- ▶ Device can be other switch allowing a tree structure
- ▶ Devices „hot-pluggable“
- ▶ Smaller plugs allow for smaller devices and computers
- ▶ Serial point-to-point connections, any device can talk to any other
- ▶ 250 – 1969 MB/s per lane (x1 lane in desktops, up to x32 lanes possible, other plugs)





► The operating system (OS)

- Acts as the interface between the I/O hardware and the program requesting I/O since
 - Multiple programs using the processor share the I/O system
 - I/O systems usually use interrupts which are handled by the OS like exceptions
 - Low-level control of an I/O device is complex and detailed

► Thus OS must

- Handle interrupts generated by I/O devices
- Supply routines for low-level I/O device operations
- Provide equitable access to the shared I/O resources
- Protect those I/O devices/activities to which a user program doesn't have access
- Schedule I/O requests to enhance system throughput

► Required communication

- OS must be able to give commands to the I/O devices
- I/O device must be able to notify the OS about its status
- Data must be able to be transferred between the memory and the I/O device



- ▶ Software running in an embedded context, e.g., on a device
 - Provides low level control of (complex) hardware to higher level software, e.g., the operating system
 - Used to abstract hardware (setting voltages, and reading analog signals)
 - Stored on non-volatile memory, e.g., flash memory

- ▶ Examples of computer firmware and one task
 - BIOS (Basic Input/Output System, old), UEFI (Unified Extensible Firmware Interface, new)
 - Initializes components on boot (RAM, drives, etc.)
 - HDD
 - Controls motor and needle position
 - SSD/USB-Sticks
 - Distributes write requests among flash chips

- ▶ Other examples: automobiles, washing machines, TVs, network switches, etc.



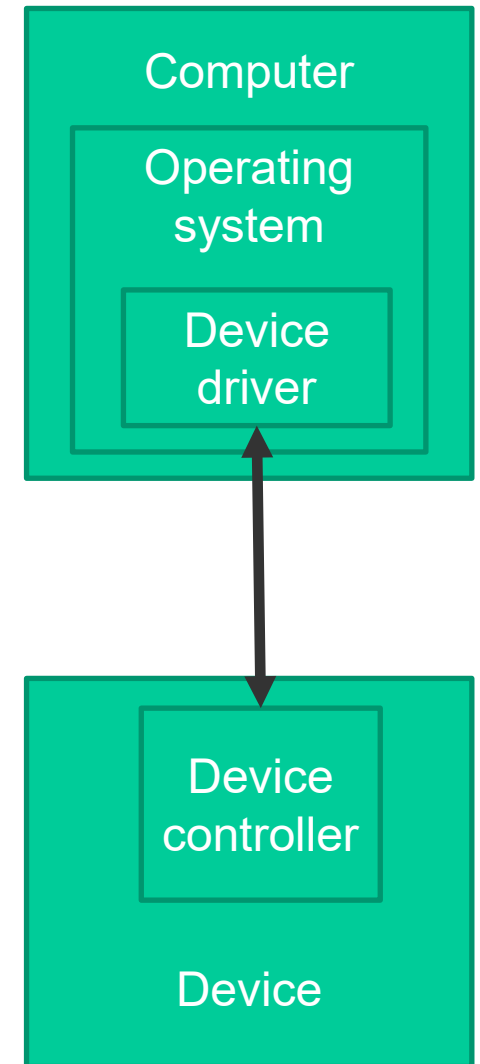
Device Driver and Device Controller

► Device controller

- Hardware or software on device
- Receives and understands signals sent by device driver
- Requires specific device driver on operating system to work properly

► Device driver

- Software within the OS
- Communicates with device controller on device
- May be accessed only in Kernel mode
- Provides rest of the operating system with a uniform interface





- ▶ I/O devices are managed by I/O controller hardware
 - Transfers data to/from device
 - Synchronizes operations with software

- ▶ Command registers
 - Cause device to do something

- ▶ Status registers
 - Indicate what the device is doing and occurrence of errors

- ▶ Data registers
 - Write: transfer data to a device
 - Read: transfer data from a device



Memory-mapped I/O

- ▶ Portions of address space assigned to an I/O device
- ▶ Reads and writes to those addresses interpreted as commands to the I/O device
- ▶ Access to this address space only through OS
- ▶ Read/write commands on bus ignored by memory but interpreted by device controller
- ▶ Address determines both device identity and type of transmission (read/write) between processor and device

Special I/O instructions

- ▶ Specify both device number and command word (or location thereof in memory)
- ▶ Can be used only in kernel mode
- ▶ Used by Intel x86 and IBM 370 computers



- ▶ Process of periodically checking the status of an I/O device
 - To determine the need to service the device
- ▶ I/O device provides information for polling in a status register
- ▶ Advantage
 - Predictable overhead, used in real-time embedded systems
- ▶ Disadvantage
 - High overhead since polling needed also in the absence of required actions



- ▶ I/O scheme that employs interrupts to indicate to the processor that an I/O device needs attention
- ▶ Used by almost all systems for at least some devices
- ▶ Interrupts CPU like an exception
 - But is asynchronous: not associated with any instruction
 - Doesn't prevent any instruction from completing
- ▶ Identification of interrupting device needed
 - Either „**cause register**“ – device places status field in cause register
 - Or „**vectored interrupt**“ – device sends vector address pointing to interrupt service routine (ISR), more efficient
- ▶ Advantages
 - Relieves the processor from having to continuously poll for an I/O event
 - User program progress is only suspended during the actual transfer of I/O data to/from user memory space
- ▶ Disadvantage
 - Special hardware needed
 - To indicate I/O device causing the interrupt and
 - To save the necessary information prior to servicing the interrupt and
 - To resume normal processing after servicing the interrupt



- ▶ I/O devices have different priorities
 - High-speed devices have higher priorities than low-speed devices
- ▶ Interrupt mechanisms provide interrupt priority levels
 - Example: UNIX has 4-6 of them
 - Indicates order in which interrupts need to be served



► Polling

- Processor loads data from I/O device registers and stores them into memory
- Processor reads data from memory and writes them to I/O device registers

► Interrupt-driven I/O

- OS transfers data in small pieces from or to a device
- OS works on other tasks during data transfer
- OS recognizes interrupt from that device, checks for errors, continues read/write
- OS informs program when data transfer is complete

► Advantage

- Allows for low-cost device controller and interfaces

► Disadvantage

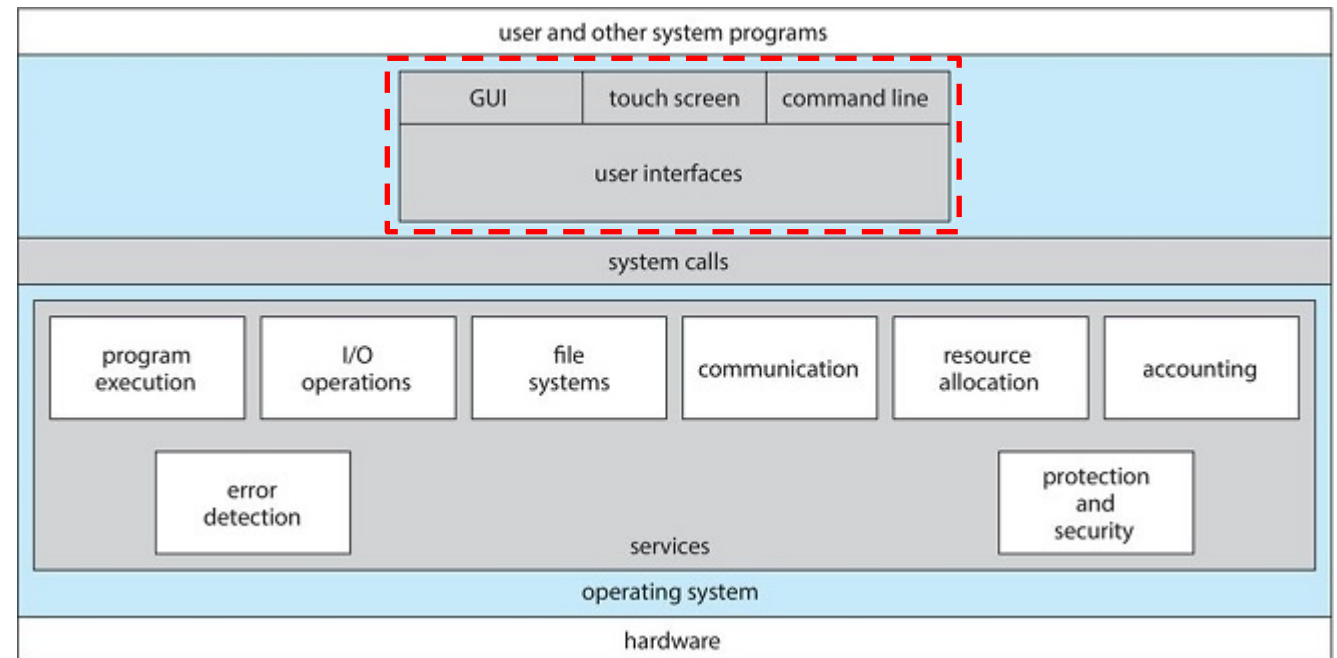
- Works well only with low-bandwidth devices



- ▶ For high-bandwidth devices (like disks)
 - Interrupt-driven I/O would consume a *lot* of processor cycles
- ▶ With DMA, the DMA controller has the ability to transfer large blocks of data **directly** to/from the memory without involving the processor
 1. The processor initiates the DMA transfer by supplying the I/O device address, the operation to be performed, the memory address destination/source, the number of bytes to transfer
 2. The DMA controller manages the entire transfer (possibly thousand of bytes in length), arbitrating for the bus
 3. When the DMA transfer is complete, the DMA controller interrupts the processor to let it know that the transfer is complete
- ▶ There may be multiple DMA devices in one system
 - Processor and DMA controllers contend for bus cycles and for memory

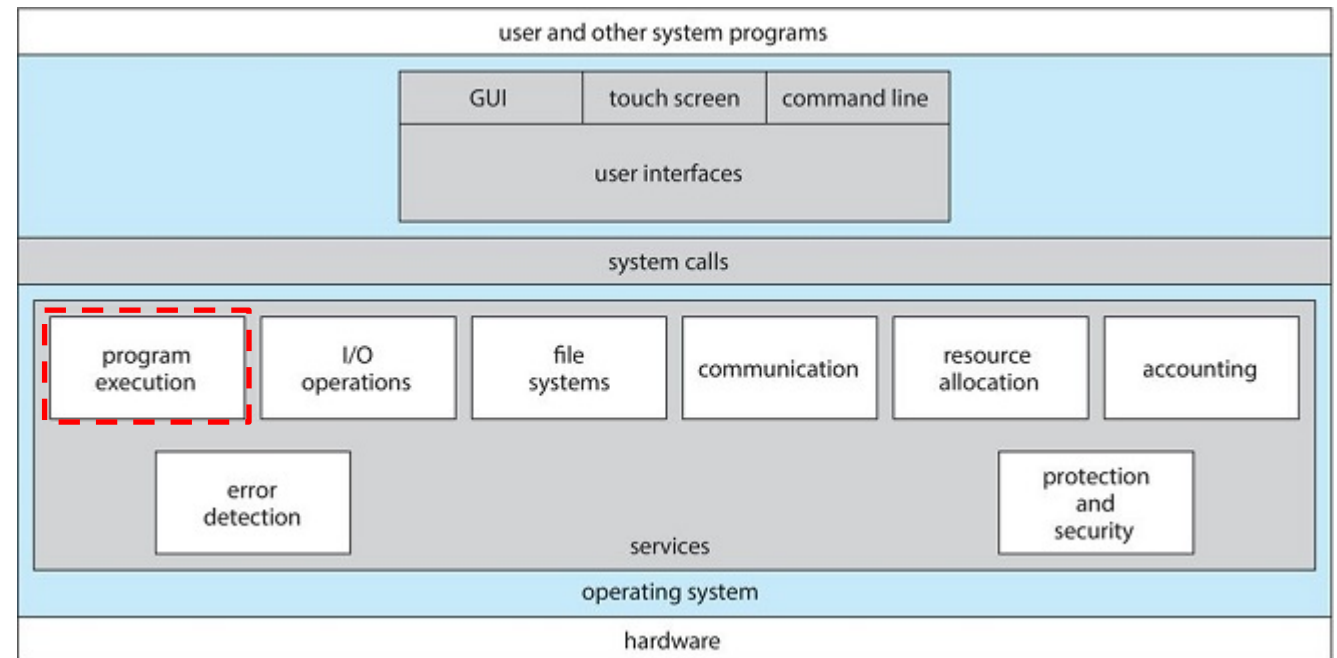


- ▶ Operating systems make certain services available to programs
 - Services may differ depending on OS
- ▶ User interface
 - Almost all OSs have a user interface (UI)
 - Most commonly graphical user interface (GUI)
 - Or command line interface (CLI)



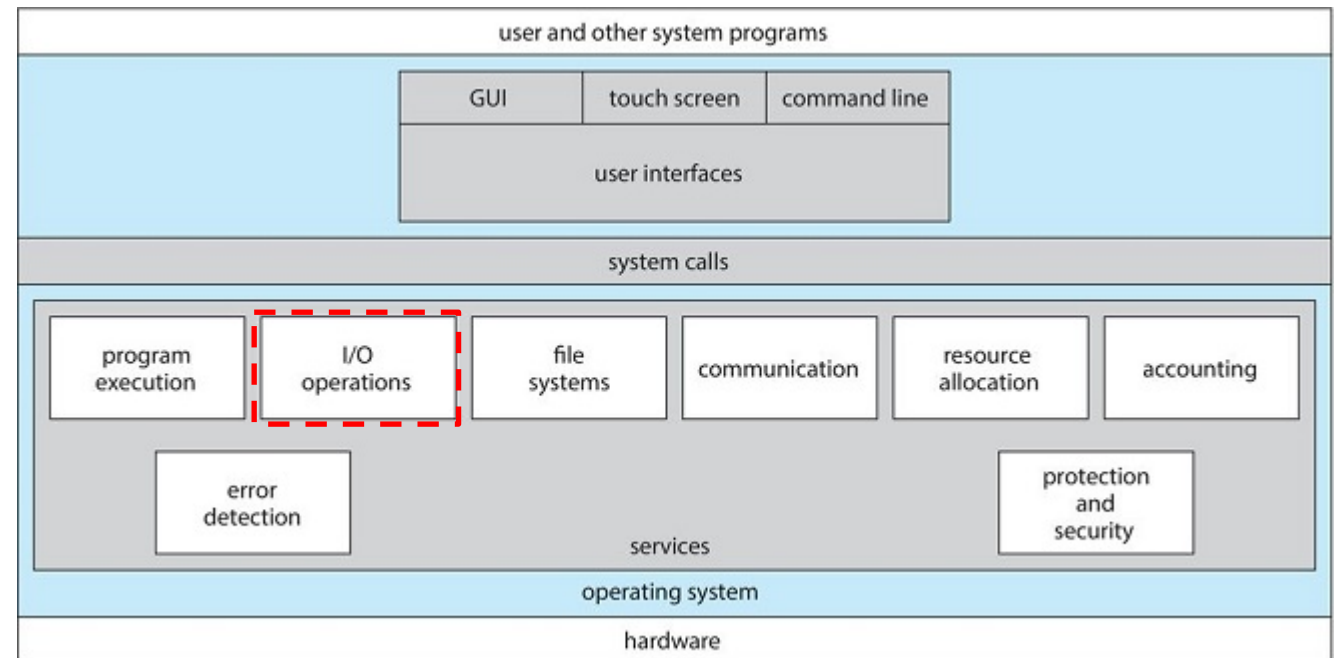


- ▶ Operating systems make certain services available to programs
 - Services may differ depending on OS
- ▶ Program execution
 - OS must be able to load a program into memory and run it
 - Program can signal normal or abnormal (error) termination



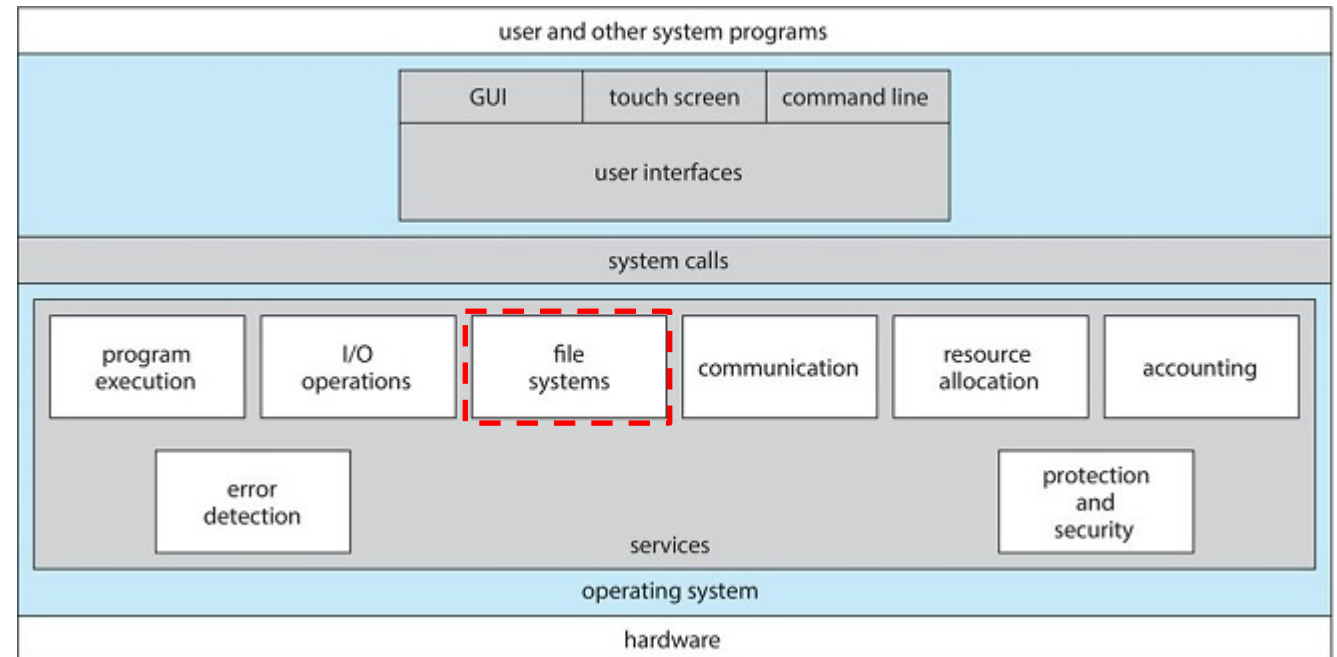


- ▶ Operating systems make certain services available to programs
 - Services may differ depending on OS
- ▶ I/O operations
 - Running program may require I/O
 - For efficiency and protection, user programs usually cannot control I/O devices directly
 - OS needs to provide an interface to do I/O



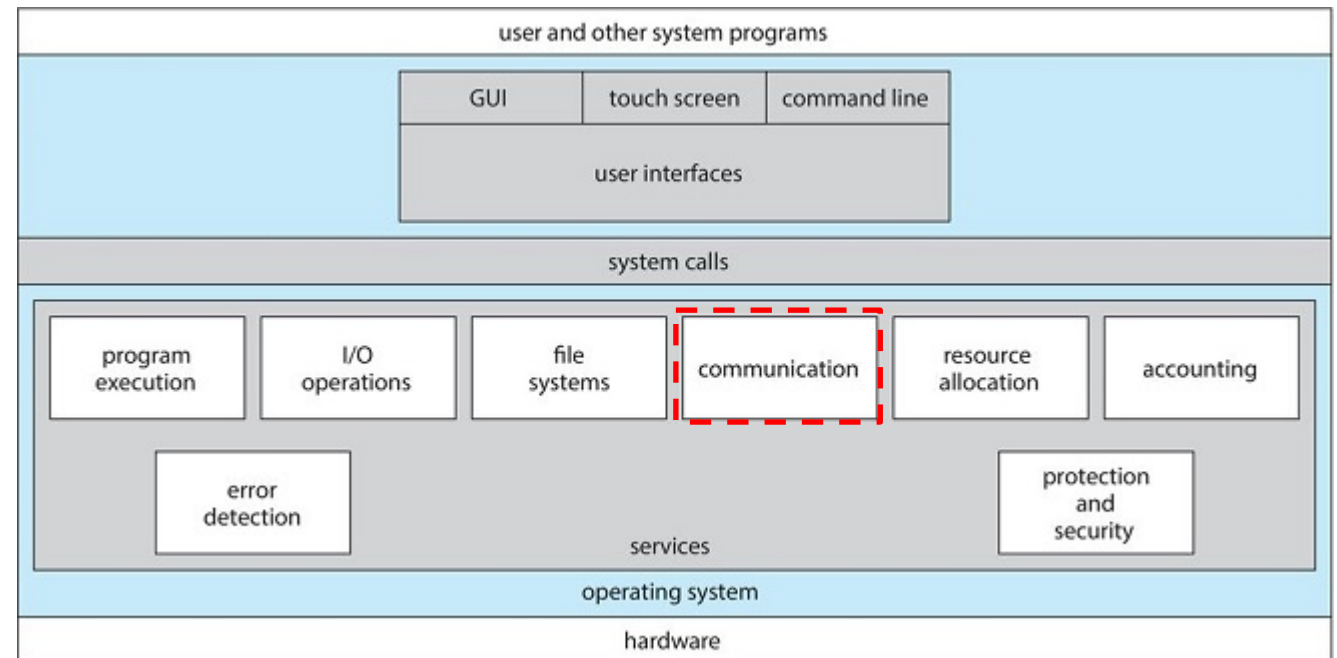


- ▶ Operating systems make certain services available to programs
 - Services may differ depending on OS
- ▶ File system manipulation
 - Programs need to read, write, create, and delete files and directories
 - Implement access control based on file ownership
 - Some OSs provide multiple file systems
 - User can choose based on features or performance characteristics



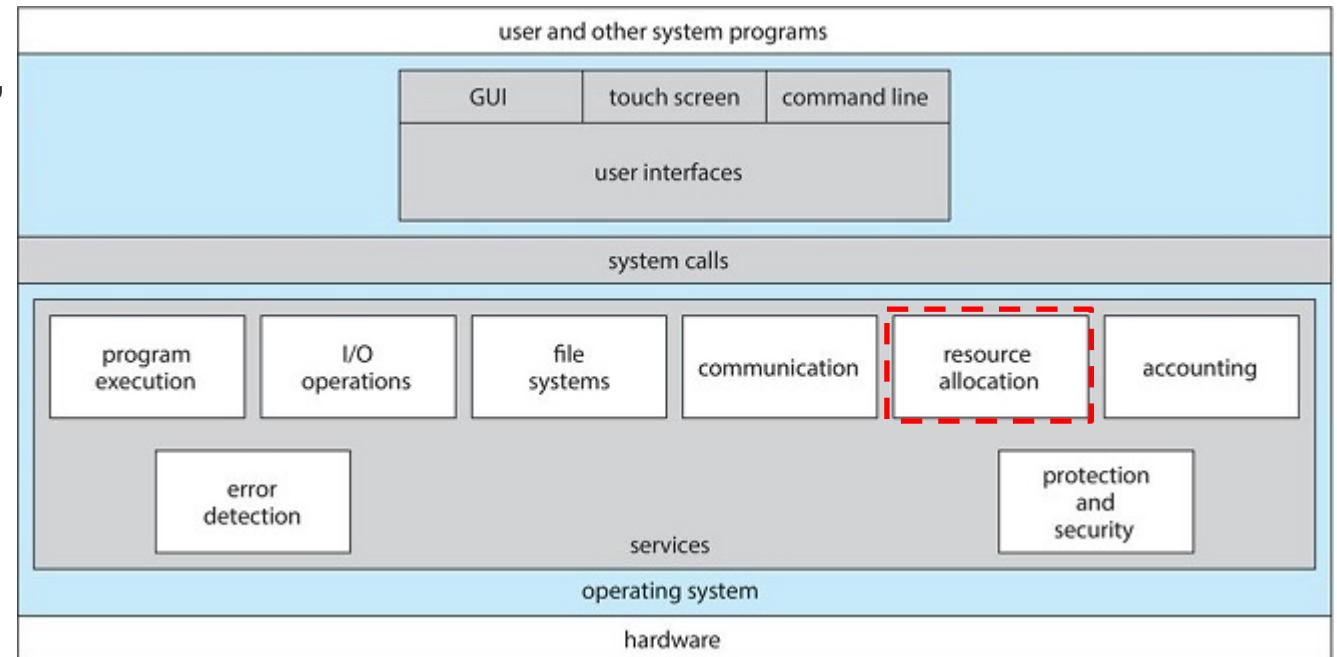


- ▶ Operating systems make certain services available to programs
 - Services may differ depending on OS
- ▶ Communication
 - Processes need to communicate with each other
 - May be implemented with shared memory in which two or more processes read/write to a shared section of the memory
 - Or implemented by message passing in which packets of information in predefined formats are moved between processes by the OS



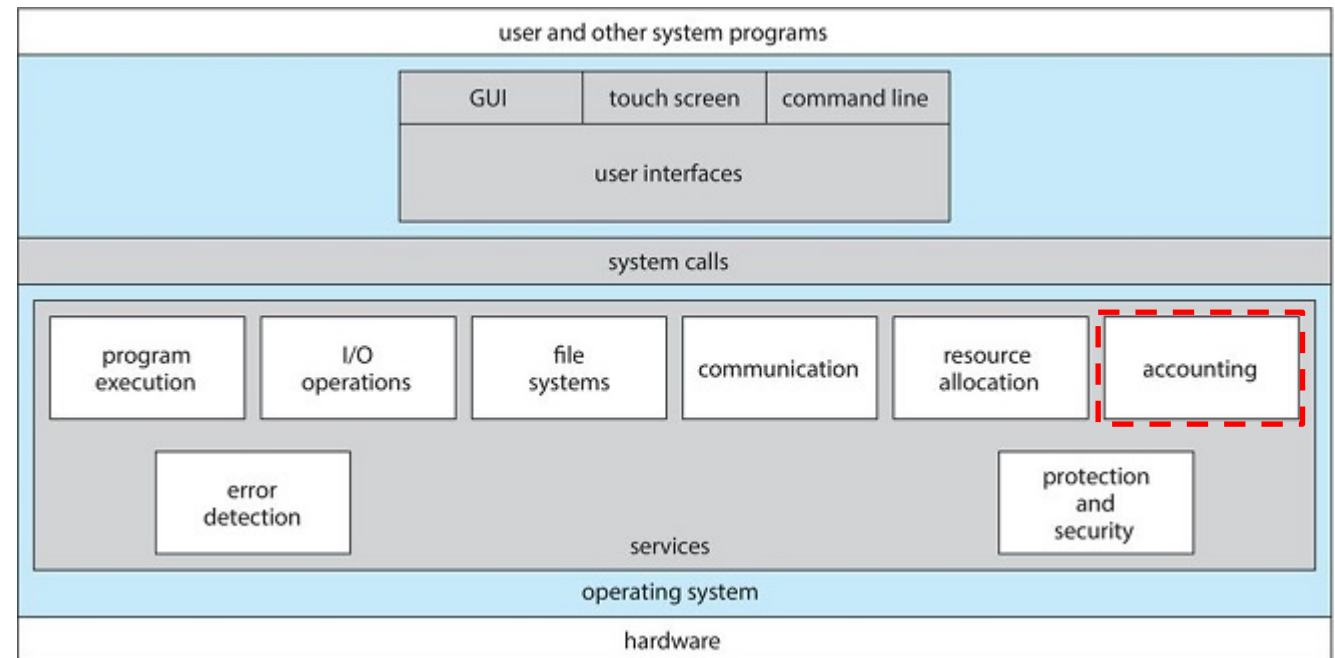


- ▶ Operating systems make certain services available to programs
 - Services may differ depending on OS
- ▶ Resource allocation
 - When multiple processes are running at the same time, resources need to be allocated to each of them
 - Resources to manage: CPU cycles, main memory, file storage, etc.)



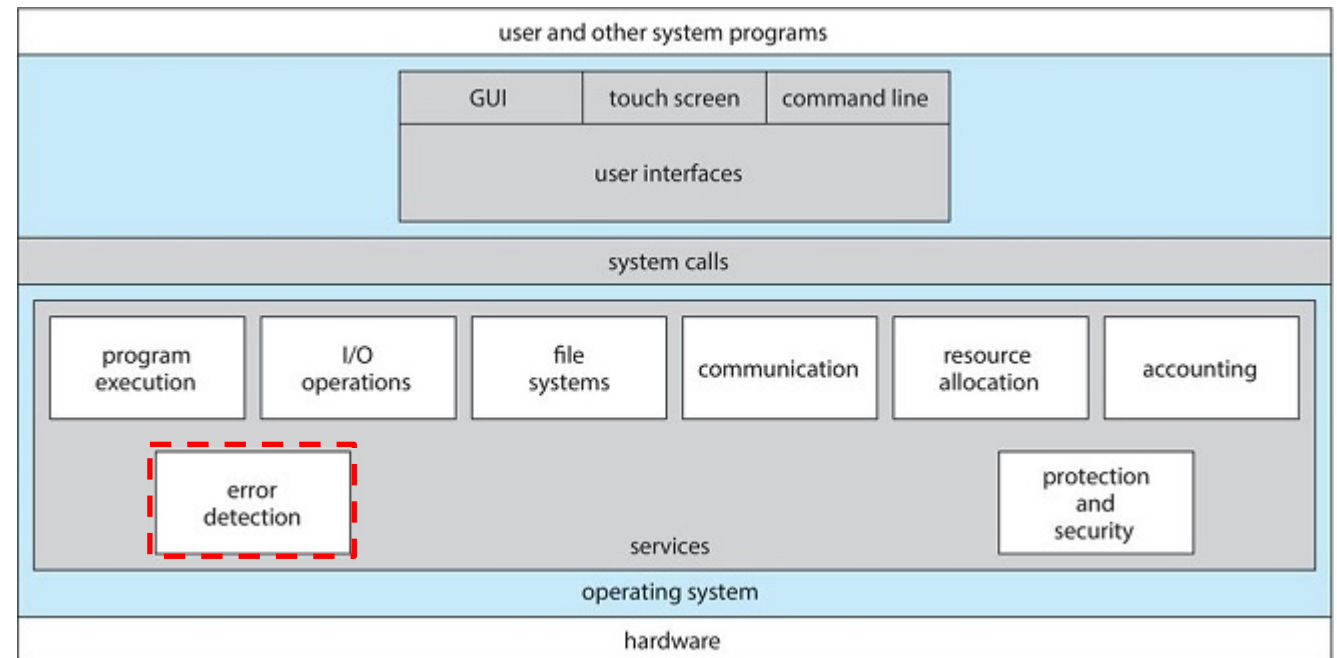


- ▶ Operating systems make certain services available to programs
 - Services may differ depending on OS
- ▶ Logging
 - Keep track of which programs use how much and what kinds of resources
 - Can be used for usage statistics or accounting



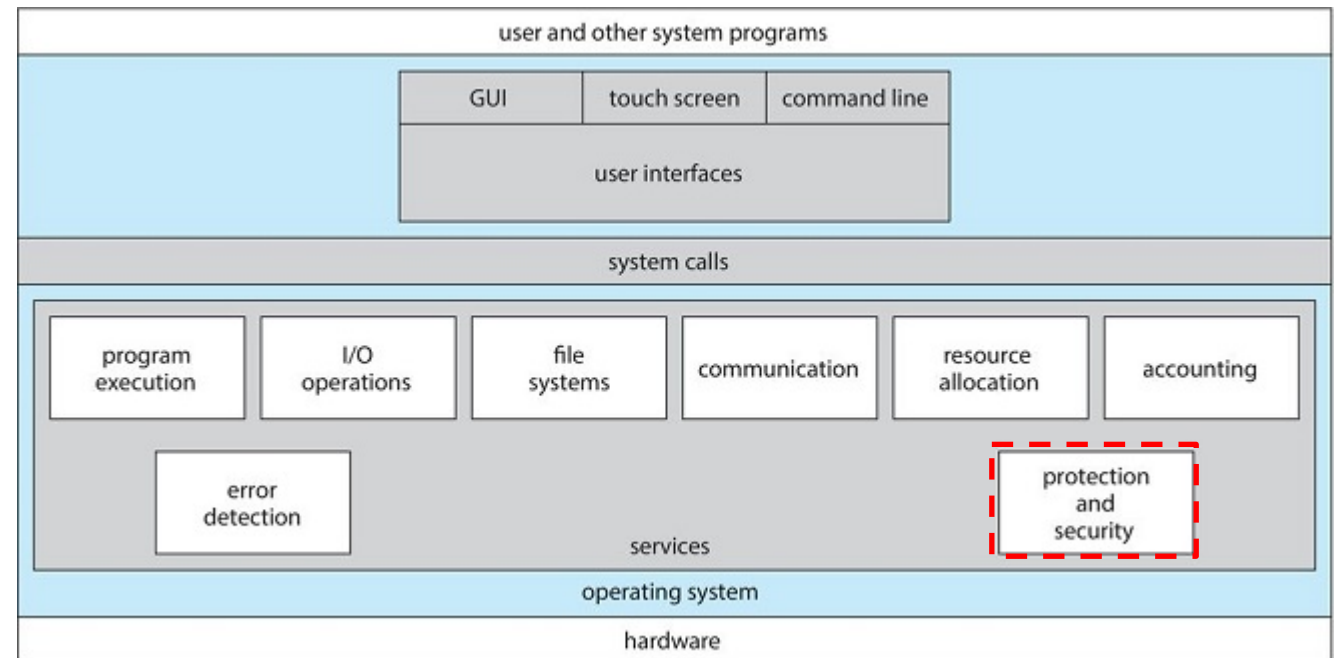


- ▶ Operating systems make certain services available to programs
 - Services may differ depending on OS
- ▶ Error detection
 - Many errors can occur (CPU or memory hardware, I/O devices, network, user program, etc.)
 - OS should take appropriate action to ensure correct computing



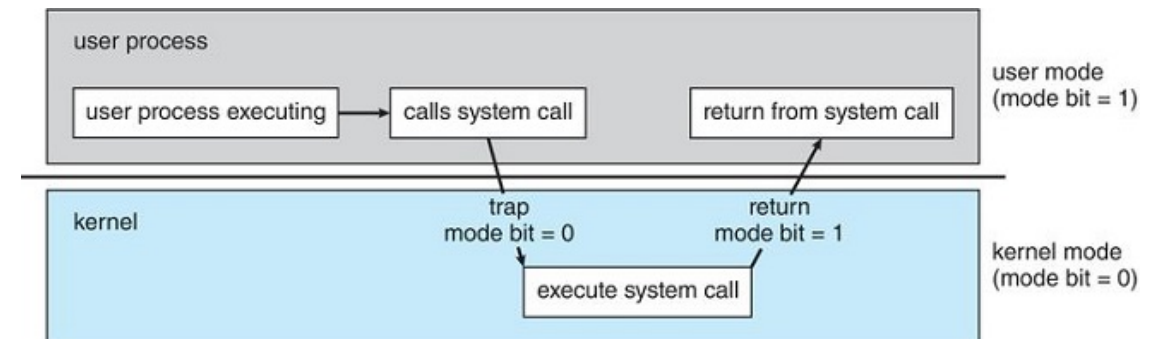


- ▶ Operating systems make certain services available to programs
 - Services may differ depending on OS
- ▶ Protection and security
 - Processed data should be secured from other users on a multi-user or networked computer system
 - Users should not interfere with each other
 - Security from outsiders
 - Authenticate actions on an OS (often with a password)
 - External I/O devices, network, etc.





- ▶ A CPU core must execute operating system code as well as user program code
- ▶ An OS must ensure that a misbehaving or malicious program cannot cause the OS or other programs to execute incorrectly
- ▶ Two operation modes
 - User mode
 - Kernel mode
- ▶ Hardware (CPU) keeps track which mode is currently active
- ▶ User mode is restricted in accessing resources
 - If a user program needs to access, e.g., the disk, it needs to do that via the kernel using a syscall
 - If a syscall is executed, the CPU switches into kernel mode and the OS is issuing an I/O request on behalf of the user program

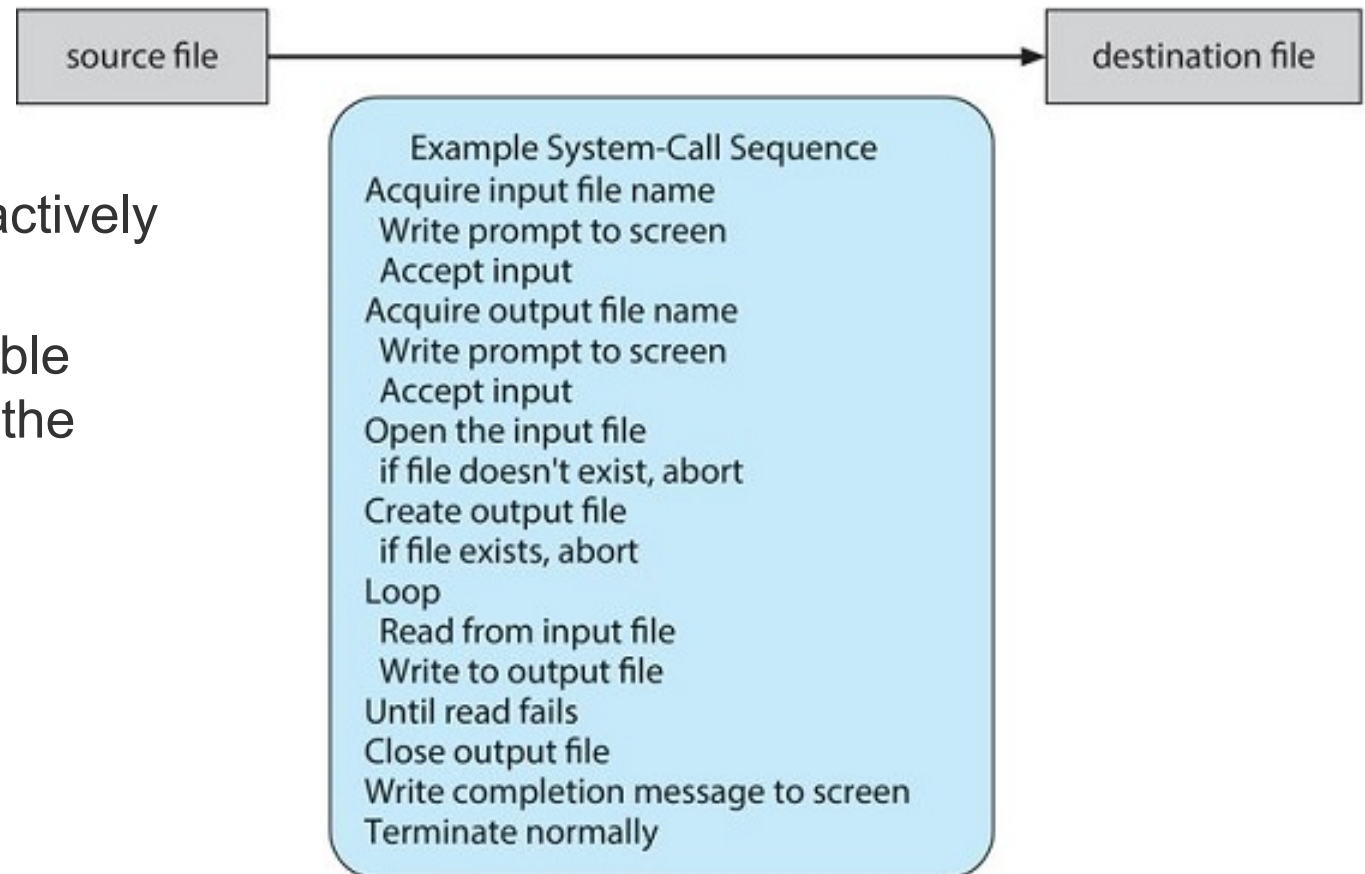




- ▶ System calls provide an interface to the operating system services

- ▶ Example: copy a file

- Copy program asks the user interactively for two input file names
- Figure on right side shows a possible sequence of syscalls executed by the program





- Unix has a open()/read() syscall which opens/reads data from a file

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

```
int          open (const char *pathname, int flags)
ssize_t      read  (int fd, void *buf, size_t count)
```

return value

function name

parameters

- Syscall open() takes a path and returns a file descriptor.
 - File descriptor is a unique integer which points to an opened file
- Syscall read() reads count bytes from a file descriptor fd into the buffer *buf
 - Return value signals how many bytes were read into the buffer



Informatik der Systeme – Chapter 10.2: Operating Systems: Process Management

Prof. Dr. Michael Menth, Gabriel Paradzik

<http://kn.inf.uni-tuebingen.de>



- ▶ Early computers allowed only one program to be executed at a time
 - Full control over system resources was given to the running program
 - Job-based: next program starts after the previous one finished its execution and terminated

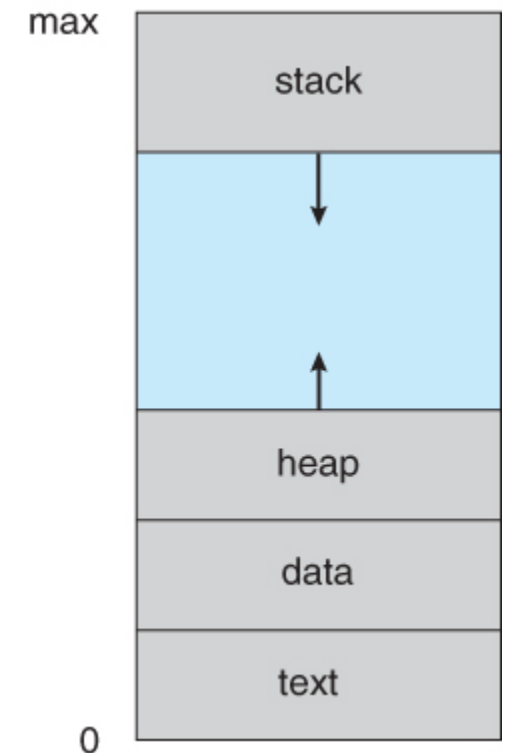
- ▶ Modern computers execute multiple programs concurrently
 - System resources need to be shared between multiple running programs

- ▶ To achieve this goal, the notion of a process was introduced

- ▶ Program vs. process
 - Program is just the executable code
 - Process is a program with its execution state (registers, memory, program counter, etc.)

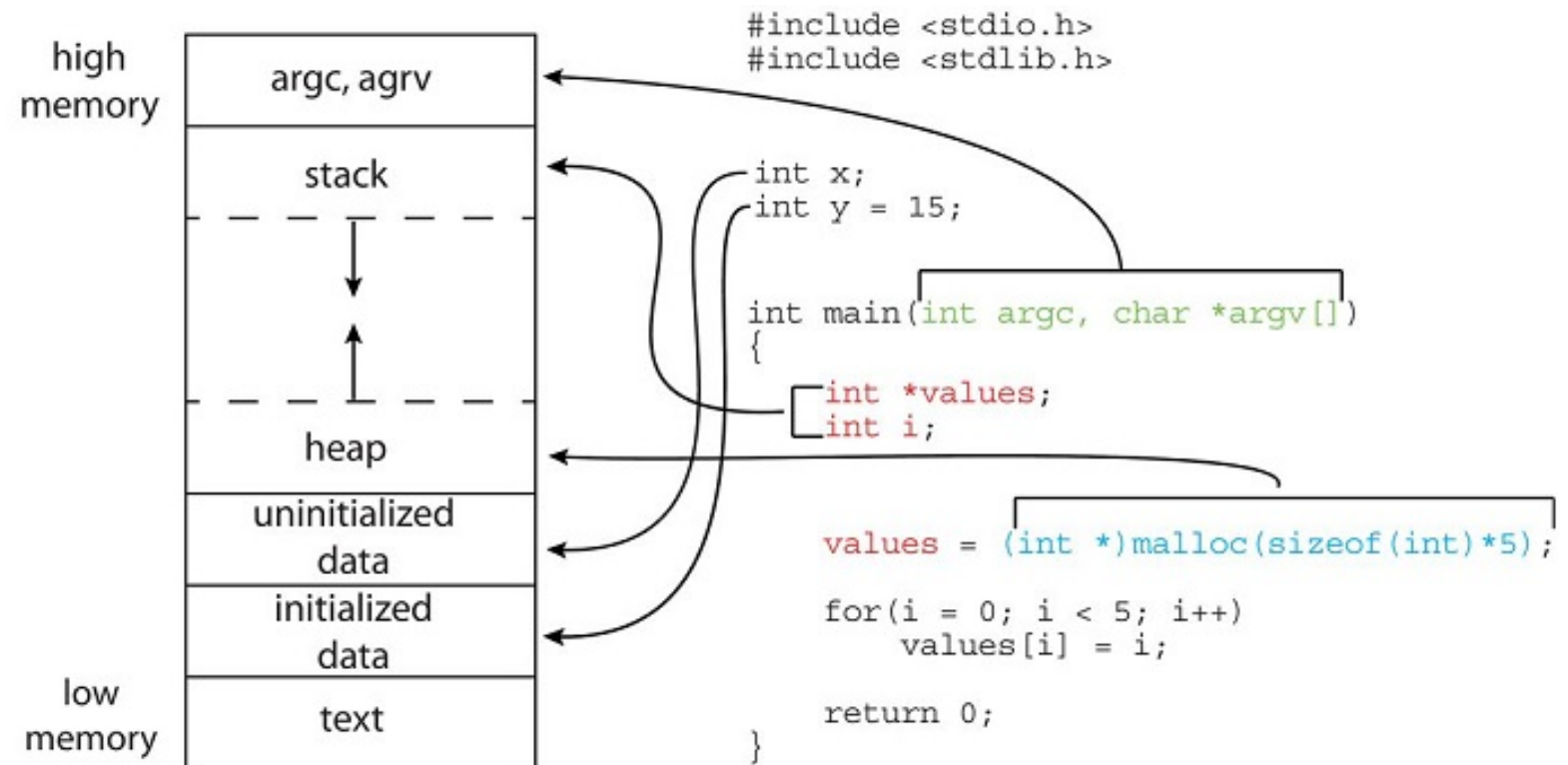


- ▶ A process is a program in execution
 - It needs resources (CPU, memory, files, I/O, etc.) to accomplish its task
 - It is the unit of work in an operating system
 - Modern operating systems allow multiple processes to run concurrently
- ▶ Memory of process consists of
 - **Stack:** temporary data storage when invoking functions
 - **Heap:** dynamic memory
 - **Data:** global variables
 - **Text:** executable code
- ▶ Data and text have a fixed size while stack and heap can grow/shrink dynamically





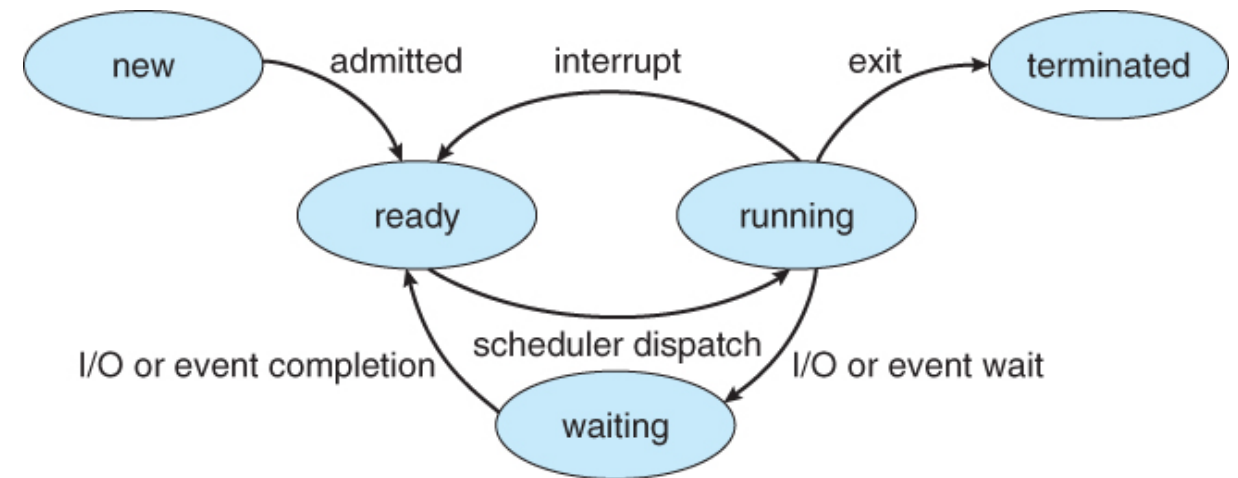
- ▶ Global variables are saved to the data section
 - This figure makes a difference between uninitialized data und initialized data
- ▶ Arguments of the main function are saved to the stack
- ▶ malloc() allocates memory on the heap





- ▶ Processes can be in different process states
 - Is it running?
 - Or waiting for I/O?
- ▶ OS needs to track process states
 - As a process executes, its state changes

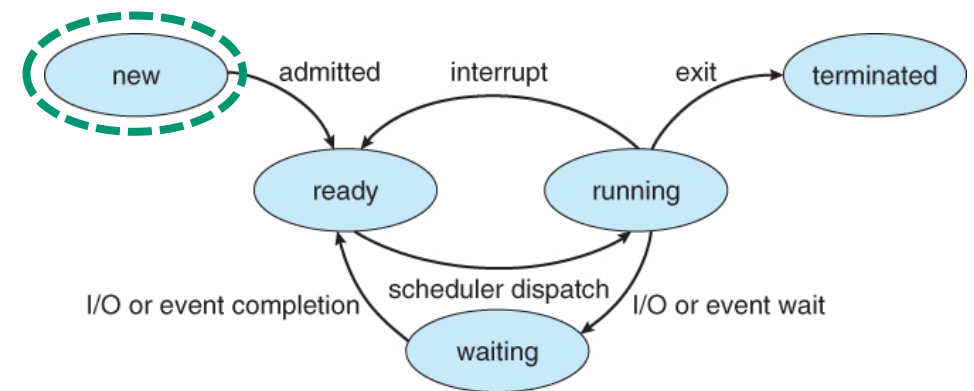
- ▶ New: process is being created
- ▶ Ready: process is waiting to be assigned to a processor
- ▶ Running: instructions are being executed
- ▶ Waiting: process is waiting for some event to occur (I/O, signal, etc.)
- ▶ Terminated: process has finished execution





Example: Directory Listing

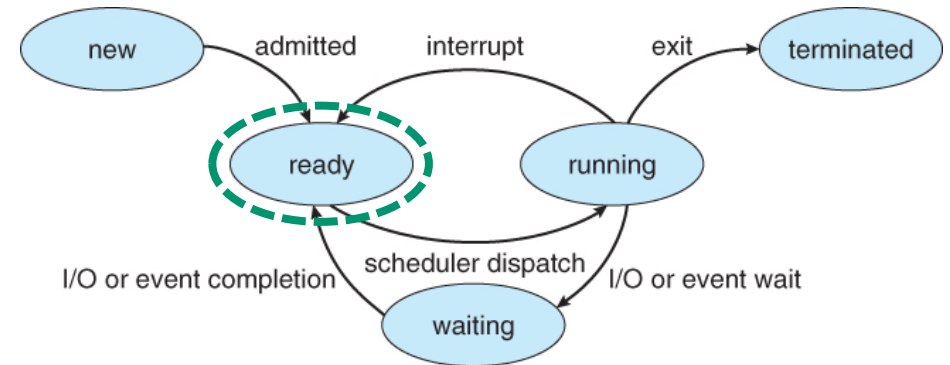
- ▶ Take a look at process state transitions
- ▶ User enters `ls` command, which lists all files and directories, into shell
- ▶ After hitting ENTER, the shell interprets this command and creates an `ls` process
 - Process is initially in new state





Example: Directory Listing

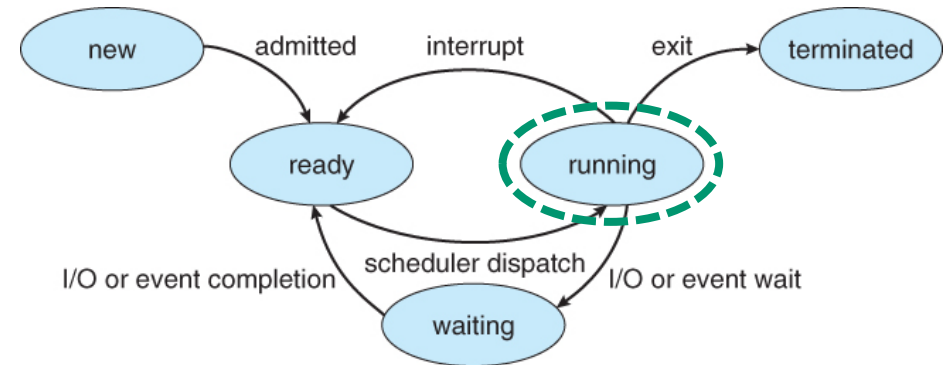
- ▶ Take a look at process state transitions
- ▶ User enters `ls` command, which lists all files and directories, into shell
- ▶ After hitting ENTER, the shell interprets this command and creates an `ls` process
 - Process is initially in new state
 - The kernel loads the program into main memory and transitions into the ready state





Example: Directory Listing

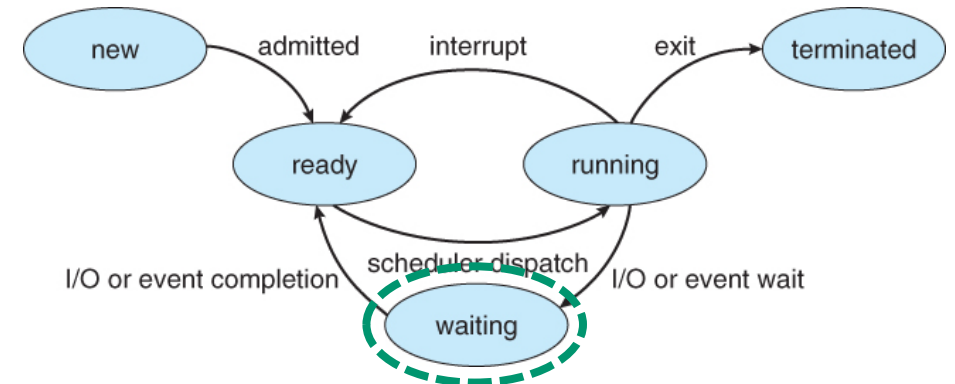
- ▶ Take a look at process state transitions
- ▶ User enters `ls` command, which lists all files and directories, into shell
- ▶ After hitting ENTER, the shell interprets this command and creates an `ls` process
 - Process is initially in new state
 - The kernel loads the program into main memory and transitions into the ready state
 - The scheduler selects the process to run





Example: Directory Listing

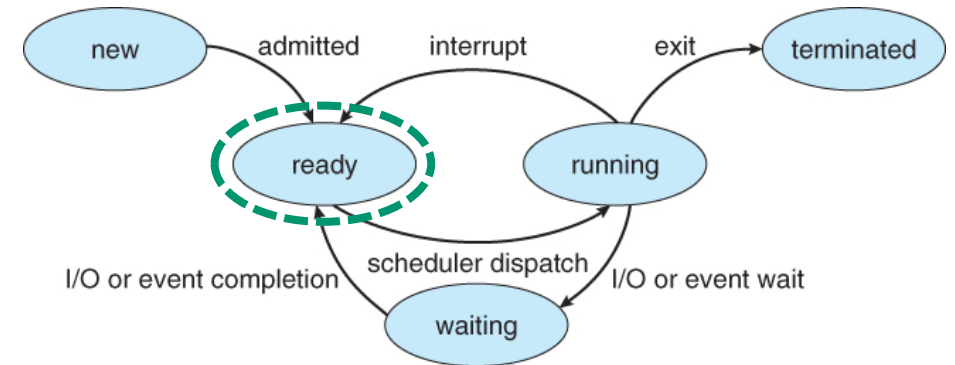
- ▶ Take a look at process state transitions
- ▶ User enters `ls` command, which lists all files and directories, into shell
- ▶ After hitting ENTER, the shell interprets this command and creates an `ls` process
 - Process is initially in new state
 - The kernel loads the program into main memory and transitions into the ready state
 - The scheduler selects the process to run
 - The `ls` process sends an I/O request to the disk and transitions into the waiting state





Example: Directory Listing

- ▶ Take a look at process state transitions
- ▶ User enters `ls` command, which lists all files and directories, into shell
- ▶ After hitting ENTER, the shell interprets this command and creates an `ls` process
 - Process is initially in new state
 - The kernel loads the program into main memory and transitions into the ready state
 - The scheduler selects the process to run
 - The `ls` process sends an I/O request to the disk and transitions into the waiting state
 - After I/O completion the process moves back into the ready state

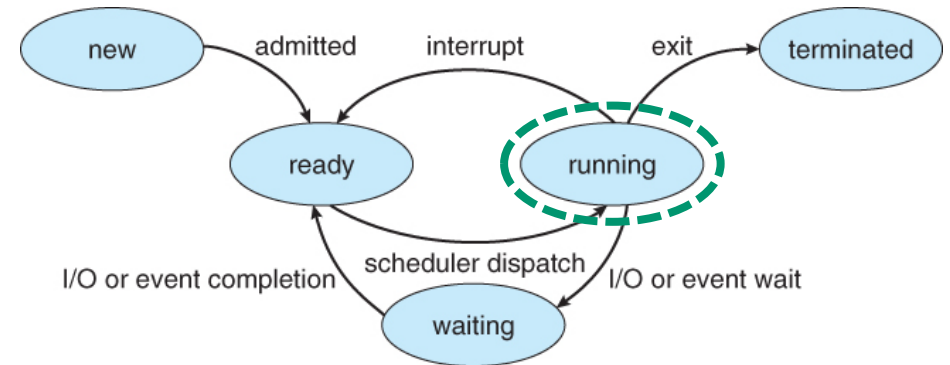




Example: Directory Listing

- ▶ Take a look at process state transitions
- ▶ User enters `ls` command, which lists all files and directories, into shell
- ▶ After hitting ENTER, the shell interprets this command and creates an `ls` process
 - Process is initially in new state
 - The kernel loads the program into main memory and transitions into the ready state
 - The scheduler selects the process to run
 - The `ls` process sends an I/O request to the disk and transitions into the waiting state
 - After I/O completion the process moves back into the ready state
 - Scheduler runs process which prints out results

```
$ ls
bin    lib    proc   sys
boot  lib64  root   tmp
dev    media  run    usr
etc    mnt    sbin   var
home   opt    srv
$
```

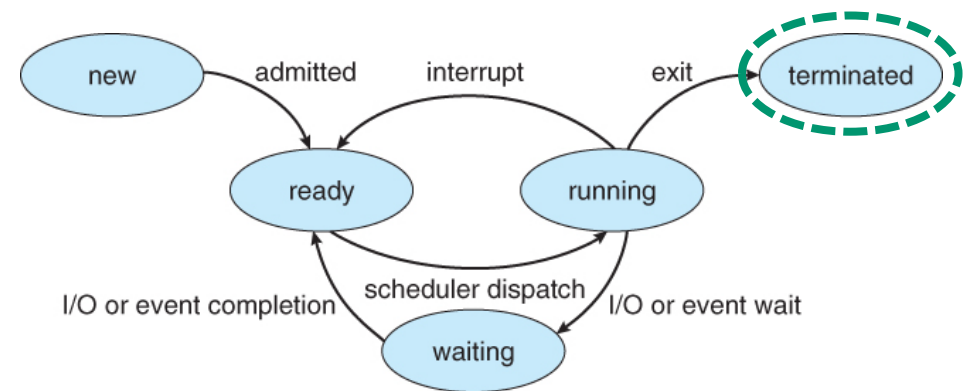




Example: Directory Listing

- ▶ Take a look at process state transitions
- ▶ User enters `ls` command, which lists all files and directories, into shell
- ▶ After hitting ENTER, the shell interprets this command and creates an `ls` process
 - Process is initially in new state
 - The kernel loads the program into main memory and transitions into the ready state
 - The scheduler selects the process to run
 - The `ls` process sends an I/O request to the disk and transitions into the waiting state
 - After I/O completion the process moves back into the ready state
 - Scheduler runs process which prints out results
 - Process finally terminates

```
$ ls
bin    lib    proc   sys
boot   lib64  root   tmp
dev    media  run    usr
etc    mnt    sbin   var
home   opt    srv
$
```



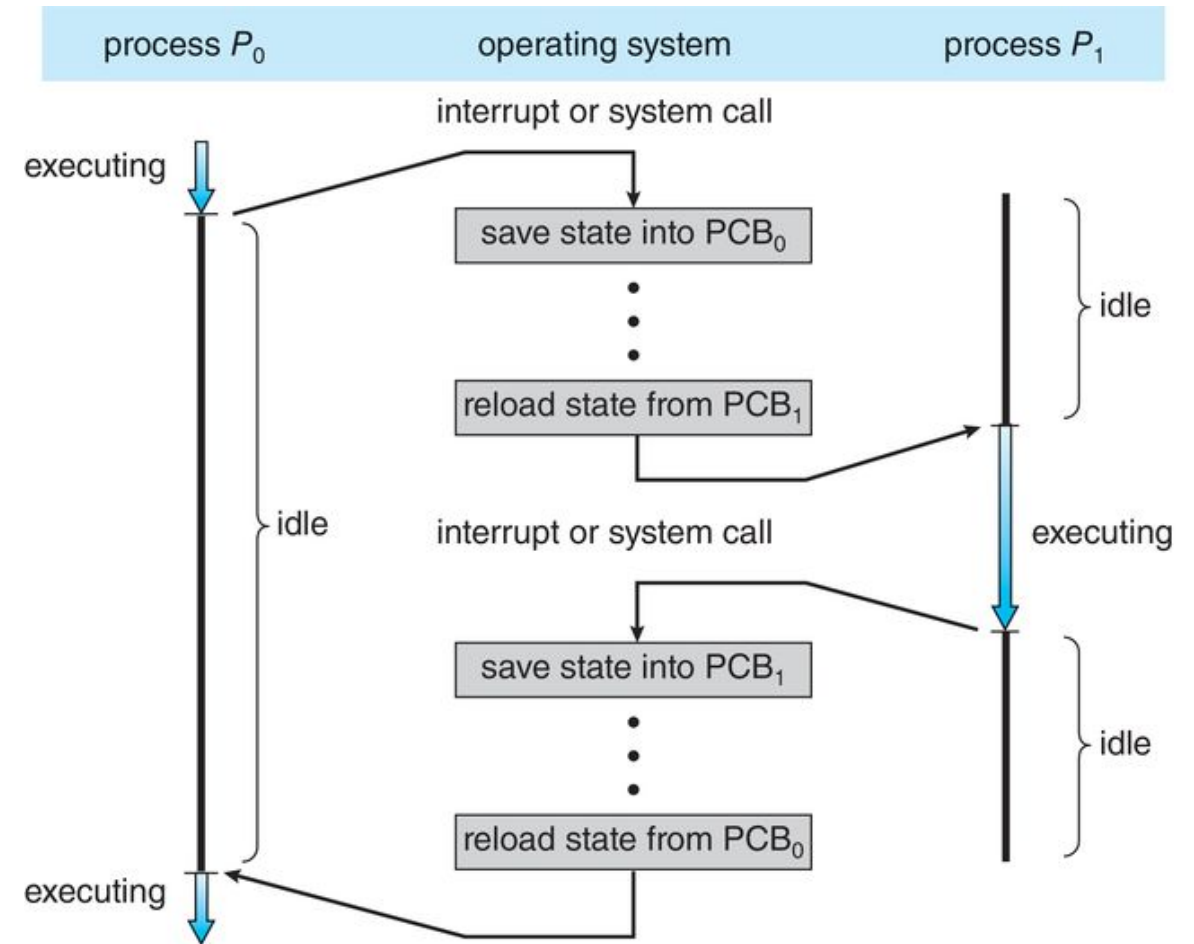


- ▶ Process is represented in an OS by a Process Control Block (PCB)
- ▶ PCB contains execution information of a process
 - Process state: New, Ready, etc.
 - Process number: unique number to identify the process
 - Program counter: pointer to next address to be executed
 - Registers: contains the values of all registers
 - Memory limits: memory management information
 - List of open files
 - Etc. (e.g., accounting information, I/O status information)



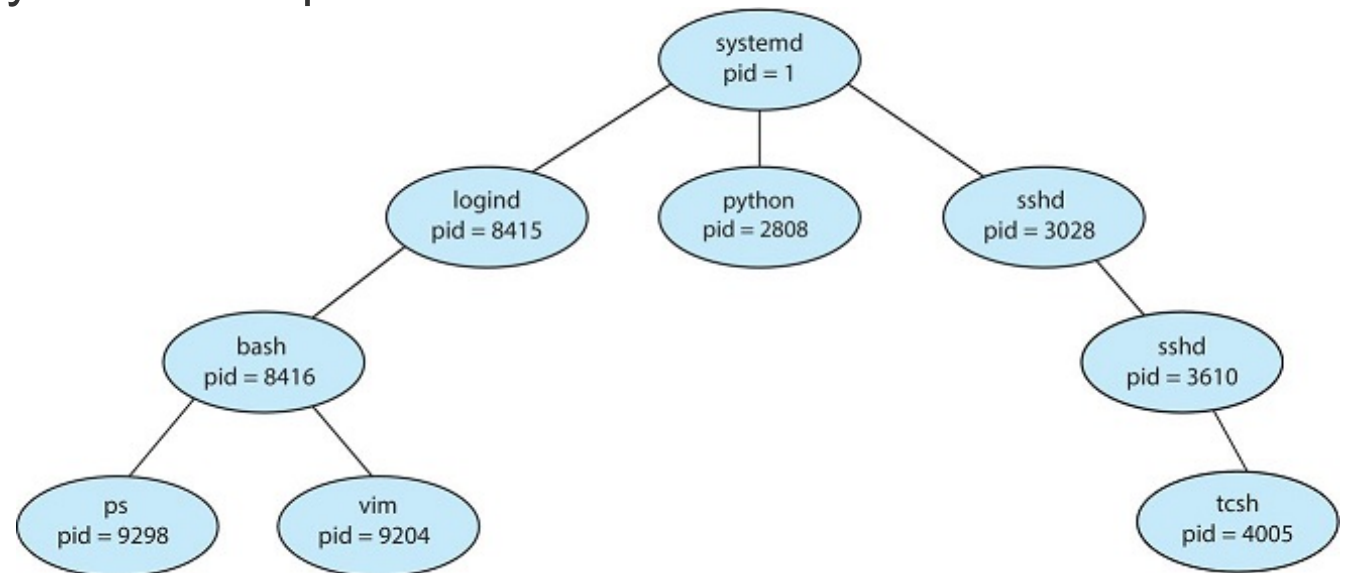


- ▶ When switching the control from a running process back to the operating system, the contents of all registers, the program counter, etc. must be saved into the PCB
- ▶ Operating system picks next process
- ▶ PCB of next process is loaded into registers, etc.
- ▶ Context switches are pure overhead because no meaningful work is done
 - Number of context switches should be kept minimal





- ▶ Processes are structured hierarchically in a tree
- ▶ A process P can create a new process Q
 - P is now the parent process of Q
 - Q is the child process of P
- ▶ Example tree on Linux
 - Parent process of all processes is systemd with pid=1





- ▶ When a child process is spawned, it needs resources (CPU time, memory)
 - Child may be able to obtain its resources directly from the OS
 - But parent process can limit resources of the child process to a subset of its own

- ▶ A parent process may wait for the termination of its child processes or execute concurrently
 - Syscall `wait()` waits for a child process to complete



- ▶ Process calls `fork()`
 - Program pauses and is duplicated
- ▶ Two copies of the program continue to run
 - But one copy has the process identifier of its child in variable `pid`
 - The other copy has the value zero in `pid`
 - In case of an error, `pid` has a negative value
- ▶ The child process executes the `ls` command which lists the contents of a directory
- ▶ The syscall `wait(NULL)` is used to wait until the child terminates
- ▶ Windows handles child creation differently

```
int main() {
    pid_t pid;

    /* fork a child process */
    pid = fork();

    if (pid < 0) { /* error occurred */
        fprintf(stderr, "Fork Failed");
        return 1;
    }
    else if (pid == 0) { /* child process */
        execvp("/bin/ls", "ls", NULL);
    }
    else { /* parent process */
        /* parent will wait for the child to complete */
        wait(NULL);
        printf("Child Complete");
    }

    return 0;
}
```



- ▶ Process model assumes only one thread of execution
- ▶ One program may want multiple threads of execution
 - E.g., word processor
 - Main loop handles keyboard input while the spell checker runs in a separate thread
 - Web browser
 - Render content while running network requests
- ▶ Solution: run multiple threads inside a process
- ▶ Why not just run multiple processes?
 - Threads share the same memory and can access each other's data

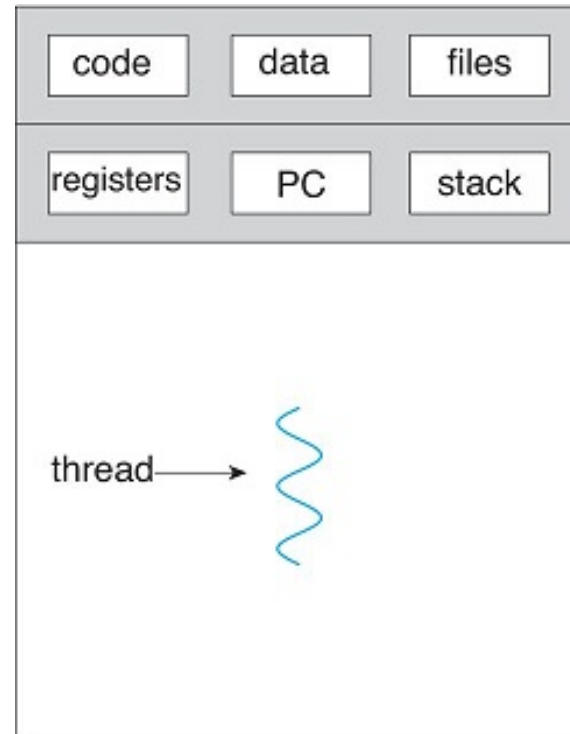


► Threads share

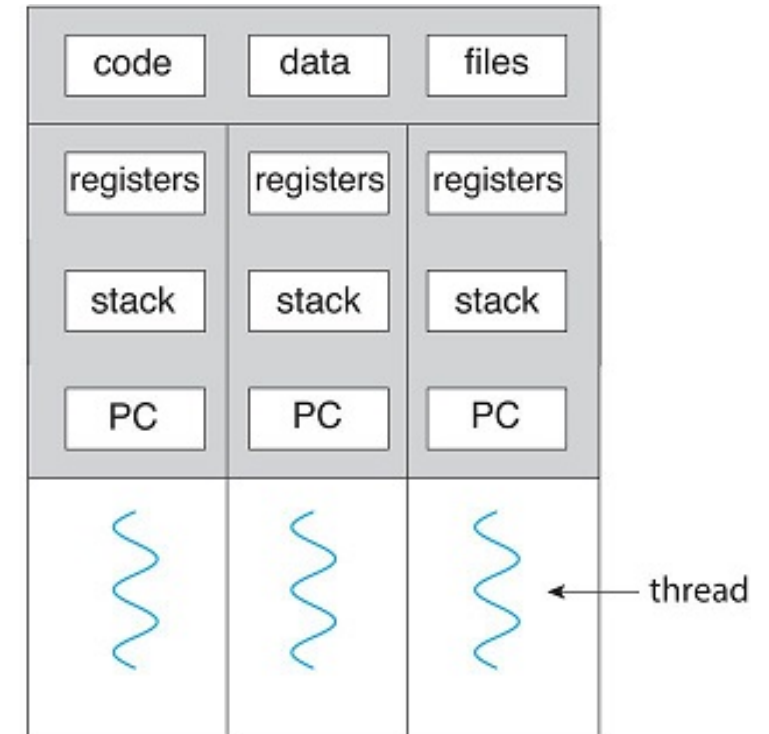
- Code
- Memory
- Open files

► But not

- Registers
- Stack
- Program counter



single-threaded process

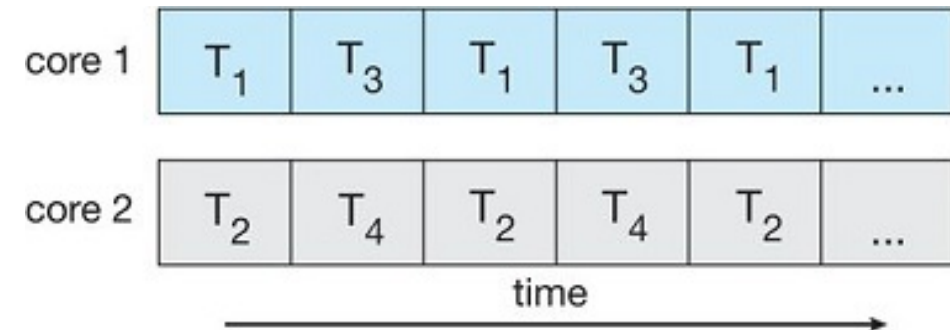
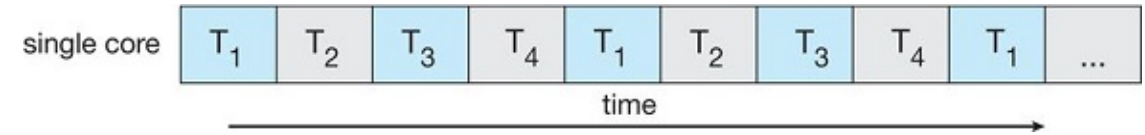


multithreaded process



Concurrency vs. Parallelism

- Concurrent execution switches between tasks
 - Allows all tasks to make progress by switching between them
 - Illusion of parallelism if switches happen fast
- Parallel execution uses multiple cores to run multiple threads





Preemptive vs. Non-Preemptive Scheduling

- ▶ A scheduling decision may take place under the following circumstances
 1. Process switches from running to the waiting state (e.g., as the result of an I/O request)
 2. Process switches from running to the ready state (e.g., when an interrupt occurs)
 3. Process switches from the waiting state to the ready state (e.g., at completion of I/O)
 4. Process terminates
- ▶ A scheduling scheme is non-preemptive (or cooperative) if scheduling takes place only under circumstances 1 and 4
 - A process in a non-preemptive scheduler runs until it gives up control by switching into the waiting state or terminates
- ▶ Otherwise a scheduling scheme is preemptive
 - Compute time of a preemptive system consists of time slices, which are allocated to processes
- ▶ A kernel must be explicitly designed to be preemptive or non-preemptive
 - Nearly all modern operating systems including Windows, Linux and Mac are preemptive



► Objectives

- Maximize CPU usage → have some process running at all times
- Time sharing → switch between processes frequently so that the user can interact between them

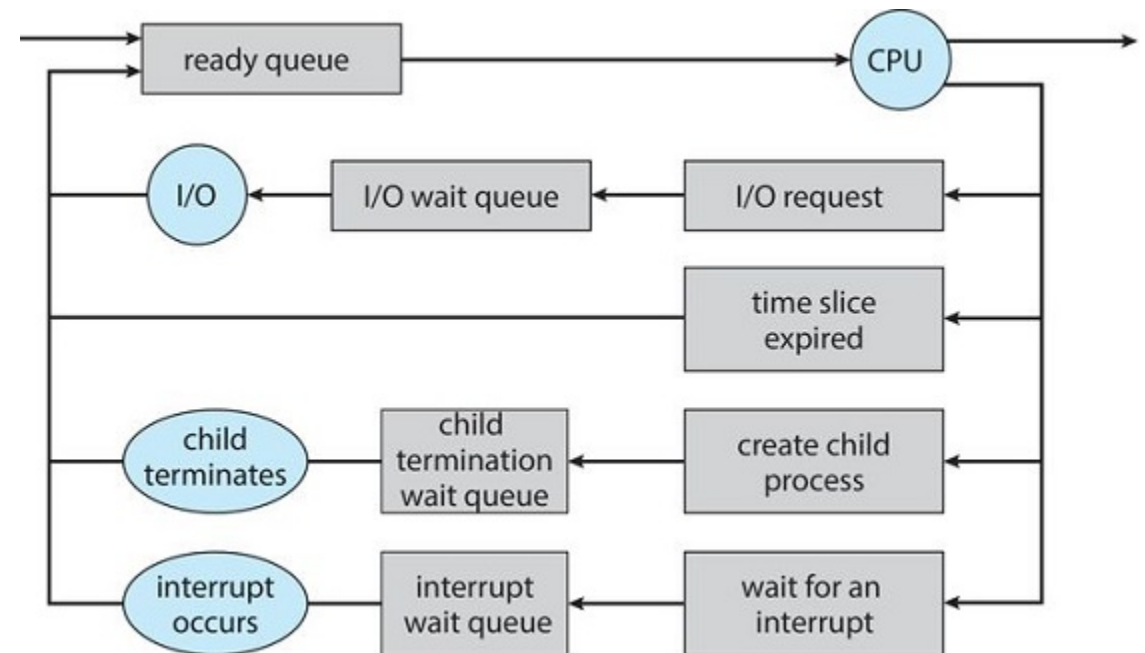
► But each CPU core can only run one process at a time

► The operating system needs to run a **process scheduler**

- Picks one process/thread to run on a CPU core from a list of available processes/threads



- Processes are initially put in a ready queue
- After some time, they are scheduled and run on a CPU
 - If the process starts a I/O request, it is put into a I/O wait queue because I/O is generally slower than a CPU
 - If the time slice expires, the process is put back into the ready queue so that another process can start its work
 - Process can wait for its child to terminate
 - Process can wait for an interrupt to happen





- ▶ CPU utilization
 - CPU should be as busy as possible
- ▶ Throughput
 - Number of processes that are completed per second
- ▶ Turnaround time
 - Time a process needs to finish, including time waited in scheduler queues or for I/O completion
- ▶ Waiting time
 - Time a process is waiting in the ready queue (without I/O)



- ▶ Scheduling algorithms choose a process from the ready queue to run on a CPU core
- ▶ We look at the following scheduling algorithms
 - First-Come First-Serve Scheduling
 - Shortest-Job-First Scheduling
 - Round-Robin Scheduling
 - Priority Scheduling
- ▶ Burst time
 - Time a process will run
 - Used in the examples
 - But the CPU does not know how long a process will run on a CPU
 - CPUs try to predict the burst time from past executions



First-Come First-Serve Scheduling

► The process which enters the ready queue first is dequeued first

► Advantages

- Simple to implement

► Disadvantages

- Average waiting time is long

– Consider:

Process	Burst time
P1	24
P2	3
P3	3

- Avg. wait time for order P1, P2, P3: 17
- If we let P2 and P3 run first and P1 last, the avg. wait time would be 3

▪ Non-preemptive algorithm

- Not a good choice for interactive systems

Process	Burst time
P1	7
P2	3
P3	12
P4	5
P5	9

Order
of
arrival



Order of execution

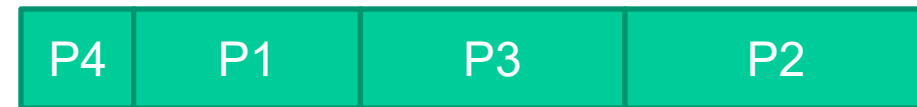


Shortest-Job-First Scheduling

- ▶ The process with lowest burst time is dequeued first
- ▶ Advantage
 - Avg. wait time is optimal
- ▶ Disadvantage
 - Cannot be implemented as is because the length of the bursts is not predictable
 - Algorithm must be approximated with empirical data

Process	Burst time
P1	6
P2	8
P3	7
P4	3

Order
of
arrival



Order of execution



Round-Robin Scheduling

- ▶ Similar to First-Come First-Serve, but with preemption
 - Defines time slice after which next process is selected
- ▶ Advantage
 - All processes continuously proceed, short process have short turnaround time in spite of concurrent longer processes
- ▶ Disadvantage
 - Performance depends on time slice
 - Too big: the algorithm is the same as First-Come First-Serve
 - Too small: can result in lots of context switches

Process	Burst time
P1	24
P2	3
P3	3

Order
of
arrival

Time slice = 4



Order of execution



- ▶ Each process is assigned a priority
 - Process with highest priority is next to run
 - If two processes have the same priority, First-Come First-Serve is used
 - In this example a lower number means higher priority

Process	Burst Time	Priority
P1	10	3
P2	3	1
P3	4	4
P4	3	5
P5	5	2

Order
of
arrival
↓



Order of execution →



Informatik der Systeme – Chapter 10.3: Operating Systems: Process Synchronization

Prof. Dr. Michael Menth, Gabriel Paradzik

<http://kn.inf.uni-tuebingen.de>



- ▶ System consists of several processes and threads, running concurrently or in parallel
- ▶ Processes/threads need to **share data**
- ▶ Uncontrolled access to **shared data** can lead to **race conditions**
 - Deutsch: Wettlaufbedingung, Wettlauf-Situation, Wettlauf um Zugriff auf Ressourcen
- ▶ Process synchronization tries to control access to **shared data** and avoid race conditions



- ▶ Let's look at an example to explain race conditions
 - Bounded-Buffer Problem
- ▶ Two parallel processes
 - Producer creates objects (`produce()`) and inserts them into the buffer
 - Objects are a placeholder for arbitrary data
 - Consumer waits until objects in buffer are available and consumes them
 - Shared variables
 - `B` (constant value, buffer size)
 - `count` (integer, buffer occupation)
- ▶ Running both processes in parallel can lead to race conditions
 - Take a closer look at the implementation of `count--` and `count++`

```
while (true) {                                     Producer
    item = produce();
    while (count == B)
        do_nothing();
    buffer[in] = item;
    in = (in + 1) % B;
    count++;
}
```

```
while (true) {                                     Consumer
    while (count == 0)
        do_nothing();
    item = buffer[out];
    out = (out + 1) % B;
    count--;
    consume(item);
}
```



► Implementation of count++/count-- at register level

```

$t0 = count      (load)
$t0 = $t0 ± 1    (in-/decrement)
count = $t0      (store)

```

► Look at the concurrent execution of count++ und count-- (initial value: 5)

- Instructions can execute in any order
- The following order leads to a race condition

Time	Producer Instruction	Consumer Instruction	\$t0 (producer)	\$t0 (consumer)	Value of count
0	\$t0 = count		5		5
1	\$t0 = \$t0 + 1		6		5
2		\$t0 = count	6	5	5
3		\$t0 = \$t0 - 1	6	4	5
4	count = \$t0		6	4	6
5		count = \$t0	6	4	4

→ Final value of count is 4, but it should be 5



- ▶ Instructions could have been reordered so that the final value of count is 6 instead of 4
- ▶ A situation in which the outcome of a computation depends on the order of instructions is called race condition
- ▶ To resolve the presented race condition, we need to ensure that only one process at a time can modify count
 - Processes need to be synchronized in some way
 - In the following some synchronization mechanisms will be presented



- ▶ We have multiple processes
 - Each process contains a critical section in which it is updating shared data
 - We want to guarantee that no two processes are running their critical section at the same time

- ▶ We want to design mechanisms to solve the critical section problem so that processes can cooperatively share data
 - Each process must request the entry of its critical section
 - If another process is actively running its critical section, the requesting process must wait

```
while (true) {
    enter critical section

    critical section code...

    exit critical section

    rest of code...
}
```



- ▶ Pure software solution for two processes i and j
- ▶ Two shared variables between two processes
 - `bool[2] flag, int turn`
- ▶ Every process sets its flag to true
 - Signals that it wants to enter the critical section
- ▶ Both processes set the turn to the other process
 - Only one write will be visible at the end
- ▶ Process i waits if process j wants to enter its critical section and turn is equal to j
 - Reflected in while condition
- ▶ As soon as a process exits its critical section, the flag is set to false
 - The other process can now enter its critical section

Code for process i

```
while (true) {  
    flag[i] = true  
    turn = j  
    while (flag[j] && turn == j)  
        ; // wait  
  
    // critical section...  
  
    flag[i] = false;  
  
    // rest of code...  
}
```



- ▶ Operating system provides synchronization primitives

- ▶ Mutex simplest synchronization primitive

- Mutex stands for Mutual exclusion

- ▶ Offers two functions: `acquire()` and `release()`

- Only one process can acquire a mutex
 - Mutex can be released from the same or from a different process

Shared: Mutex `m`

```
while (true) {
```

```
    acquire(m)
```

```
    // critical section
```

```
    release(m)
```

```
    // remainder section
```

```
}
```



- ▶ Each mutex has a variable “available” which is initially true
 - When acquire() is called, the mutex waits until available is true and then sets it to false
 - When release() is called, this variable is set back to true

- ▶ Disadvantage: busy waiting
 - Also called spinlock
 - Under some circumstances more efficient than a context switch
 - If lock time is shorter than two context switches

```
acquire() {
    while (!available)
        ; // busy waiting

    available = false;
}
```

```
release() {
    available = true;
}
```



- ▶ Java does not offer a “true” Mutex class
- ▶ ReentrantLock is similar to mutex
 - But it can only be unlocked from the thread which locked it
 - Recursive locks are also possible with ReentrantLocks, which is not possible with a true mutex

```
class X {
    ReentrantLock lock
        = new ReentrantLock();

    public void m() {
        lock.lock();

        // critical section

        lock.unlock()
    }
}
```

Disclaimer: In real world java, you need additional error handling
[ReentrantLock Java documentation](#)



- ▶ Mutex locks are the simplest synchronization tool
 - Semaphores allow more sophisticated use cases
- ▶ Two function modify integer s
 - All modification to the integer s happen atomically
 - Semaphore guarantees atomicity wrt the value of s
 - Code on the right is oversimplified
 - `wait(s)` waits until s is positive, then s is decremented
 - `signal(s)` increments s
- ▶ Code on the right is only for demonstrating a semaphore
 - Semaphores do not cause busy waiting as the code on the right suggests
 - Next slide shows a more realistic implementation without busy waiting
- ▶ Examples for semaphores will be shown later

```
wait(s) {
    while (s <= 0)
        ; // wait
    s--;
}
```

```
signal(s) {
    s++;
}
```



- ▶ Semaphores used to synchronize use of multiple resources among multiple processes
 - Advantage: no busy waiting, synchronizes multiple identical resources
- ▶ Data structure of semaphore s
 - Integer counter
 - $\text{counter} \geq 0$: counts available resources
 - $\text{counter} < 0$: counts waiting processes
 - List of waiting processes
- ▶ `wait(s)` (atomic function)
 - `counter--`
 - If $\text{counter} < 0$, the calling process must wait: add process to list and call `sleep()`
 - Otherwise, execution continues
- ▶ `signal(s)` (atomic function)
 - `counter++`
 - If $\text{counter} \leq 0$, wake up a waiting process

```

wait() {
    counter--;
    if (counter < 0) {
        add this process to list
of processes;
        sleep();
    }
}

signal() {
    counter++;
    if (counter ≤ 0) {
        remove a process P from
list of processes;
        wakeup(P);
    }
}
    
```



- ▶ Java offers a semaphore class in the standard library
- ▶ Example: semaphore is initialized with the value $s=5$
- ▶ Methods have different names
 - `wait(s)` is called
 - `s.acquire()` ($s--$)
 - `signal(s)` is called
 - `s.release()` ($s++$)

```
class X {
    Semaphore s
        = new Semaphore(5);

    public void m() {
        s.acquire();

        // critical section

        s.release()
    }
}
```

Disclaimer: In real world java, you need additional error handling
[Semaphore Java documentation](#)



- ▶ Mutex and semaphores offer a convenient mechanism for process synchronization
- ▶ But it is easy to use them incorrectly, which can result in timing errors
 - Timing errors are hard to detect because they happen only in particular execution sequences
- ▶ If a single process from a set of synchronized processes misbehaves (due to a bug or an uncooperative programmer), all processes may misbehave
- ▶ Look at a program with a critical section which is protected by a semaphore with an initial value 1
 - Under normal circumstances a process must call `wait(s)` before entering its critical section and `signal(s)` after leaving
 - If `wait(s)` and `signal(s)` are switched, multiple processes can enter their critical section simultaneously
 - If a program calls `wait(s)` two times, the semaphore is blocked indefinitely for all processes
- ▶ For these reasons, most programming languages offer higher-level synchronization primitives



Higher-Level Synchronization (2)

- ▶ Java offers a built-in synchronized keyword
 - It takes an object as argument
 - Only one thread can enter its critical section, synchronizing on the same object
 - But two synchronized blocks with different objects can execute simultaneously
- ▶ Synchronized keyword also exists for method signatures
 - Only one thread can execute the method per instance
 - But two synchronized methods can run concurrently if they belong to different instances
- ▶ Java also offers a large range of synchronization primitives in its standard library
 - Synchronized lists, maps, etc.

```
// ...

synchronized (o) {
    // critical section
}

// ...
```

```
public class X {
    synchronized void f() {
        // critical section
    }
}
```



- ▶ The use of synchronization can lead to deadlocks (Sackgasse)
- ▶ Classic example
 - Suppose there are two resources A and B and two processes P and Q
 - P acquires resource A
 - Q acquires resource B
 - P wants to acquire resource B, but it is held by Q
 - Q wants to acquire resource A, but it is held by P
- ▶ Both processes cannot continue with their work and are waiting indefinitely
 - They are in a deadlocked state



- ▶ We look at some classes of synchronization problems
 - Commonly used for testing newly proposed synchronization schemes
 - In the following, these synchronization problems are solved with semaphores

- ▶ Recall
 - Semaphore s has resource counter
 - `wait(s)`
 - Decrements counter
 - Calling process waits if counter < 0
 - `signal(s)`
 - Increments counter
 - Calling process wakes up other sleeping process if counter ≤ 0



Bounded-Buffer Problem (1)

► Already introduced before

- If instructions are executed concurrently, the value of count can be incorrectly saved

```
while (true) {                                     Producer
    // produce an item
    while (count == B)
        do_nothing();
    buffer[in] = next_produced;
    in = (in + 1) % B;
    count++;
}
```

```
while (true) {                                     Consumer
    while (count == 0)
        do_nothing();
    next_consumed = buffer[out];
    out = out + 1 % B;
    count--;
    // consume an item
}
```



Bounded-Buffer Problem (2)

- ▶ Solution uses following data structures
 - Buffer with size n
 - Semaphore `buffer_access = 1`
 - For synchronizing buffer access to add/remove item
 - Semaphore `free_slots = n`
 - Value indicates free slots in buffer
 - Semaphore `available_items = 0`
 - Value indicates available items in buffer
- ▶ Use the semaphore `buffer_access` when manipulating the buffer (critical section)
- ▶ Use semaphores `free_slots` and `available_items` to indicate buffer usage
 - Produce an item when buffer has free slots
 - Consume an item when available in buffer

```
while (true) {                                     Producer
    // produce an item
    wait(free_slots);
    wait(buffer_access);
    // add item to buffer
    signal(buffer_access);
    signal(available_items);
}
```

```
while (true) {                                     Consumer
    wait(available_items);
    wait(buffer_access);
    // remove item from buffer
    signal(buffer_access);
    signal(free_slots);
    // consume an item
}
```



- ▶ E.g., database is shared among multiple processes
- ▶ Some processes want to write to the DB
- ▶ But some only want to read data
- ▶ If two readers access the database, nothing bad will happen if no synchronization is applied
- ▶ But a writer and another (reader or writer) process need synchronization



Reader-Writers Problem (3)

► Solution uses following data structures

- Semaphore `write_access = 1`
- Integer `read_count = 0`
- Semaphore `read_count_access = 1`

► Writer

- Uses `write_access` to protect critical section for writing

► Reader

- Uses `read_count_access` to modify or read `read_count`
- Uses `write_access` to prevent writing while reading
 - Acquire lock before first reader starts
 - Release lock after last reader stops

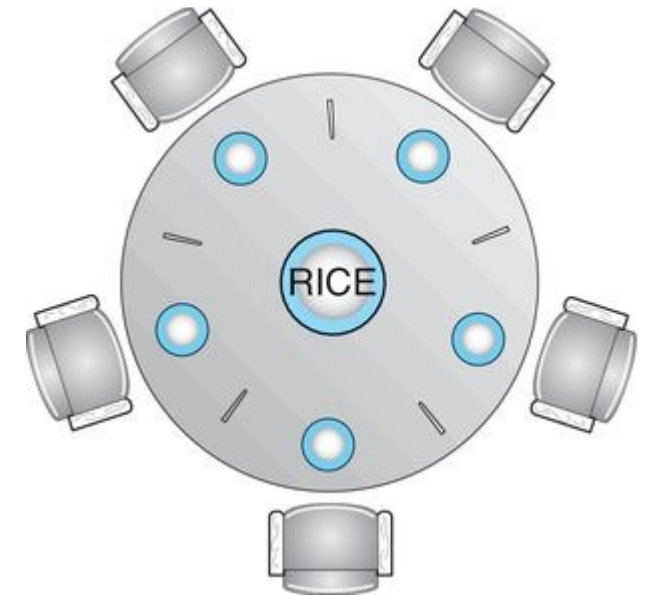
```
while (true) {                                     Writer
    wait(write_access);
    // writing is performed
    signal(write_access);
}
```

```
while (true) {                                     Reader
    wait(read_count_access);
    read_count++;
    if (read_count == 1)
        wait(write_access);
    signal(read_count_access);
    // reading is performed
    wait(read_count_access);
    read_count--;
    if (read_count == 0)
        signal(write_access);
    signal(read_count_access);
}
```



Dining-Philosophers Problem (1)

- ▶ Suppose there are five philosophers who want to eat a bowl of rice
- ▶ Between the five bowls are five chopsticks
- ▶ To eat the rice, one needs two chopsticks
 - A philosopher can only eat if no neighbor is currently eating
- ▶ Fun fact
 - Between waiting/eating, philosophers are thinking for some time
- ▶ Classic problem not because of its practical importance, but because it is an example of a large class of concurrency-control problems
 - Simple representation of the need to allocate several resources among several processes in a deadlock-free and starvation-free manner

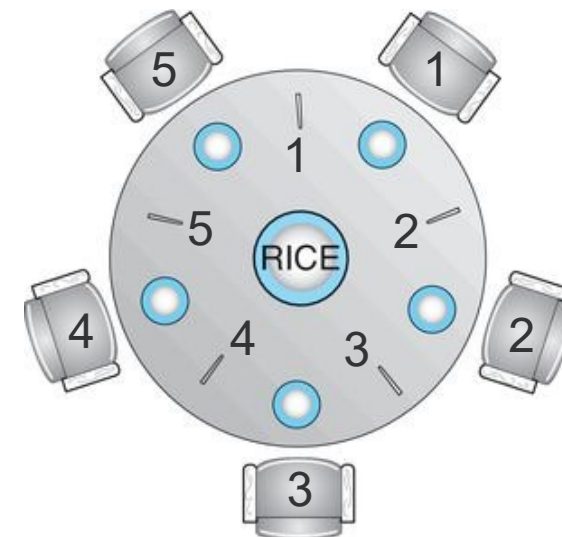




Dining-Philosophers Problem (2)

- ▶ Solution uses following data structures
 - Semaphore chopstick[5]
 - List of semaphores, one for each chopstick
- ▶ When a philosopher wants to eat, the right chopstick is held, then the left one
- ▶ Simple solution, but it is not deadlock-free
 - If everyone picks up the right chopstick, no one can pick up the left one → deadlock
- ▶ Solutions
 - Allow only four philosophers on the table
 - Only pick up the first chopstick if both are available (check in a critical section)
 - Asymmetric solution: odd-numbered philosopher picks up first the left chopstick. Even-numbered philosopher picks up first the right one
 - Semaphore solution is not presented here

```
while (true) {                                     Philosopher
    wait(chopstick[i]);
    wait(chopstick[(i+1) % 5]);
    // eat rice
    signal(chopstick[i]);
    signal(chopstick[(i+1) % 5]);
    // think again
}
```





Informatik der Systeme – Chapter 10.4: Operating Systems: File Systems

Prof. Dr. Michael Menth, Gabriel Paradzik

<http://kn.inf.uni-tuebingen.de>



► File system

- The most visible part of an operating system for most users
- Organized storage for all kinds of data
- Consists of two parts
 - Collection of files, each storing related data
 - Directory structure which organizes files in the system
- Lives on some storage device



- ▶ Computer can store information on different storage devices
 - NVMe devices, HDDs, magnetic tapes
- ▶ Operating system should provide a logical view of stored information
 - Abstract physical properties
- ▶ A file is
 - A named collection of related information on secondary storage
 - The smallest unit of logical data for a user
 - Data cannot be written to secondary storage unless it is in a file
- ▶ A file in a modern system is a sequence of bytes



- ▶ File is named for human convenience
 - Some systems differentiate between uppercase and lowercase characters while others don't

- ▶ File attributes
 - Name: symbolic file name in human-readable form
 - Identifier: unique tag for identification within the file system (not human-readable)
 - Type: if the system supports different file types
 - Location: pointer to the device and to the location
 - Size: size of file in bytes, words, or blocks.
 - Protection: access control information (who may read, write, etc.)
 - Timestamps and user identification: file creation/modification time and information
- ▶ Information about all files is kept in the directory structure



- ▶ These following seven basic operations comprise the minimal set of required file operations
- ▶ Creating a file
 - First, find space in the file system
 - Second, make an entry for the new file in a directory
- ▶ Opening a file
 - Opening a file takes a file path and returns a file handle (or file descriptor)
 - File handle is used in all other file operations except creation and deletion
 - Other file operations do not need the file path, only the file handle
 - File path evaluation and access control happens only once at open()



- ▶ Writing a file
 - Make a system call with file handle and the information to be written
 - System keeps a write pointer to the current location in the file
- ▶ Reading a file
 - Make a system call with file handle and pointer to memory
 - System keeps a read pointer to the current location in the file
- ▶ Repositioning within a file
 - System repositions current-file-position pointer to a given value
 - Does not necessarily cause actual I/O
 - Also known as seek operation
- ▶ Deleting a file
 - Release all file space
 - Remove entry from directory
- ▶ Truncating a file
 - Erase file contents, but keep all file attributes



- ▶ File types define how the contents of a file should be treated by the OS or user programs
 - Common technique is to include the file type in the file name by an extension

File Type	Usual extension	Function
Executable	exe, com, bin or none	Ready-to-run machine language program
Object	obj, o	Compiled, machine language, not linked
Source code	c, cc, java, hs, asm	Source code in various languages
Batch	bat, sh	Commands to the command interpreter
Markup	xml, html, tex	Textual data, documents
Word processor	doc, docx, rtf	Various word-processor formats
Library	lib, a, so, dll	Libraries of routines for programmers
Print or view	gif, pdf, jpg	ASCII or binary file in a format for printing or viewing
Archive	rar, zip, tar	Related files grouped into one file, sometimes compressed
Multimedia	mpeg, mov, mp3, mp4, avi	Binary file containing audio or A/V information



- ▶ System uses extension to indicate operation that can be done
 - E.g., only .exe, .com, etc. can be executed
 - User programs may use file extensions to indicate file types in which they are interested
 - Can be considered as hints for the applications that operate on them
 - Because operating systems at a low level do not differentiate between the file name and file extension
 - But a file extension puts no constraints on the file content

- ▶ UNIX uses a magic number
 - Stored at the beginning of some binary files to indicate the type of data in the file
 - E.g., format of an image file
 - GIFs always start with the sequence: GIF89a
 - Unix scripts always start with: #! (shebang)
 - Not all files have magic numbers



- ▶ We look at some file systems which are used in modern computers
 - ext4
 - NTFS
 - APFS
 - Linux proc file system



► “Fourth extended file system”

- Released 2008
- Ext4 is the default file system for most Linux distributions
 - Some modern Linux distributions use Btrfs
- Supports encryption since kernel version 4.6
- Supports journaling
 - Modification to the file systems are written sequentially to a journal
 - A set of operations that performs a specific task is a transaction
 - Once a transaction is written to the journal, it is considered to be committed
 - Now the transaction is applied to the actual file system structures
 - When an entire committed transaction is completed, it is removed from the journal
 - If the system crashes, some transactions may remain in the journal
 - Transactions can be executed until work is complete and file system structures remain consistent



► “New Technology File System”

- Proprietary file system from Microsoft
- Released 1993
- Default in all windows systems
- A file is not a simple byte stream as in UNIX
 - File is structured object consisting of typed attributes
 - File name, creation time, security descriptor
- Volume (deutsch: Band in Serie von Büchern) is a fundamental entity in NTFS
 - Volume may occupy portion of a device, or entire device, or may span several devices
 - Volumes may be protected with various levels of RAID
- Supports error recovery on power loss



► “Apple File System”

- Proprietary file system from Apple
- Released 2017
- Instant file cloning (copy-on-write)
 - When cloned, only the directory entry is duplicated
 - When a file is written, the block is copied
- Snapshots are consistent points-in-time, read-only instances of the file system
- Encryption is natively supported
- Supports error recovery on power loss



► FAT12, FAT16, FAT32, exFAT

- Originally for floppy disks (FAT12)
 - FAT: File Allocation Table
 - 12: Number of bits in a FAT entry
- Commonly found on portable data storage

► Unix/Linux file systems

- Btrfs (B-tree FS, “Butter-FS”)
 - Copy-on-write
 - Snapshots
 - Current state of the file system can be frozen atomically
- ZFS
 - Often used on NAS devices (network attached storage) or servers
 - Supports RAID



► Example of a virtual file system

- It does not store any persistent data
 - But user can interact with it, as with any other “real” file system
- Files are used to interact with the Linux kernel in various ways

► Acts as interface to kernel functionalities.

- Some examples
 - General information about the running kernel is stored
 - Transmitted bytes over the network, number of opened files, ...
 - proc file system contains a directory for every running process
 - Directory contains information about running process
 - Used memory, CPU usage, ...

► Proc file system provides a way for programs to access this information as plain text files



Linux Proc File System – Examples

- ▶ Example: Content of file /proc/cpuinfo
 - Contains information about installed CPUs

```
Processor      : 0
vendor_id     : AuthenticAMD
cpu family    : 23
model         : 96
model name    : AMD Ryzen 7 PRO
4750U with Radeon Graphics
stepping      : 1
microcode     : 0x8600106
cpu MHz       : 3962.193
cache size    : 512 KB
...
```

- ▶ Example: Content of file /proc/meminfo
 - Contains information about installed memory and its current usage

```
MemTotal:      32064840 kB
MemFree:       2087616 kB
MemAvailable:  13439912 kB
Buffers:       213828 kB
Cached:        19359204 kB
...
```