



# Introduction to Operating Systems

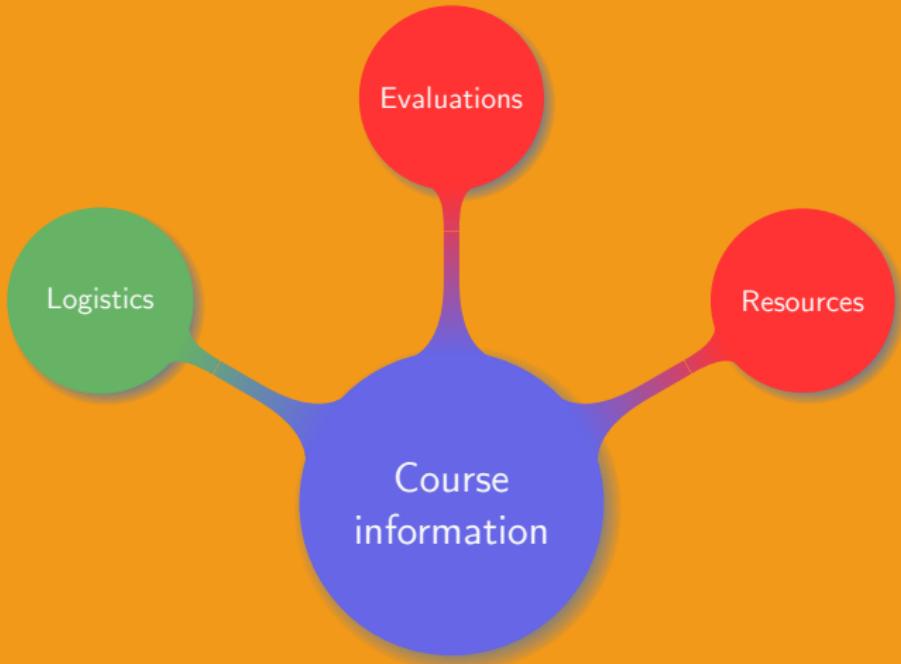
Manuel – Fall 2019



## 0. Course information

---

# Chapter organisation



## Teaching team:

- Instructor: Manuel ([charlem@sjtu.edu.cn](mailto:charlem@sjtu.edu.cn))
- Teaching assistants:
  - Jiayi ([jane\\_chen@sjtu.edu.cn](mailto:jane_chen@sjtu.edu.cn))
  - Minhao ([jinminhao@sjtu.edu.cn](mailto:jinminhao@sjtu.edu.cn))

## Teaching team:

- Instructor: Manuel ([charlem@sjtu.edu.cn](mailto:charlem@sjtu.edu.cn))
- Teaching assistants:
  - Jiayi ([jane\\_chen@sjtu.edu.cn](mailto:jane_chen@sjtu.edu.cn))
  - Minhao ([jinminhao@sjtu.edu.cn](mailto:jinminhao@sjtu.edu.cn))

## Important rules:

- When contacting a TA for an important matter, CC the instructor
- Prepend [VE482] to the subject, e.g. Subject: [VE482] Grades
- Use SJTU jBox service to share large files (> 2 MB)

## Teaching team:

- Instructor: Manuel ([charlem@sjtu.edu.cn](mailto:charlem@sjtu.edu.cn))
- Teaching assistants:
  - Jiayi ([jane\\_chen@sjtu.edu.cn](mailto:jane_chen@sjtu.edu.cn))
  - Minhao ([jinminhao@sjtu.edu.cn](mailto:jinminhao@sjtu.edu.cn))

## Important rules:

- When contacting a TA for an important matter, CC the instructor
- Prepend [VE482] to the subject, e.g. Subject: [VE482] Grades
- Use SJTU jBox service to share large files (> 2 MB)

Never send large files by email

## Course arrangements:

- Lectures:
  - Tuesday 12:10 – 13:50
  - Thursday 12:10 – 13:50
  - Friday 8:00 – 9:40 (even weeks)
- Office hours:
  - Tuesday 9:40 – 11:20
  - Thursday 9:40 – 11:20

*Appointments outside of the office hours can be taken by email*

# Course objectives

Main goals of this course:

- Understand the functioning of operating systems
- Become familiar with the internal structure of operating systems
- Be able to perform basic operating system coding

# Course objectives

Main goals of this course:

- Understand the functioning of operating systems
- Become familiar with the internal structure of operating systems
- Be able to perform basic operating system coding

*Be able to share in the development of an operating system*

## Learning strategy:

- Course side:
  - ① Understand how to efficiently use the CPU
  - ② Know how to handle Memory, Input/Output, and Filesystems
  - ③ Get a basic idea of security and distributed systems

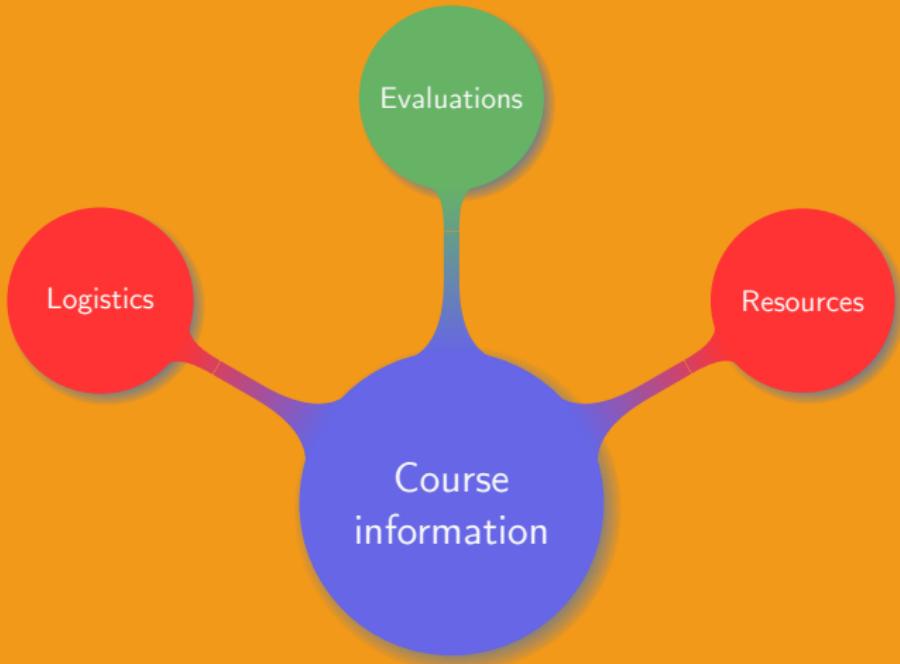
## Learning strategy:

- Course side:
  - ① Understand how to efficiently use the CPU
  - ② Know how to handle Memory, Input/Output, and Filesystems
  - ③ Get a basic idea of security and distributed systems
- Personal side:
  - ① Read and write code
  - ② Relate known strategies to new problems
  - ③ Perform extra research

## Detailed goals:

- Understand the general organisation of an OS
- Understand the hardware organisation
- Be familiar with the concept of process and threads
- Be able to solve common problems related to inter-process communication
- Be able to implement the most common scheduling algorithms
- Be able to analyse, prevent or solve deadlock issues
- Be familiar with the memory management and filesystems
- Be proficient at using Unix systems, spot particular parts of the kernel code, and write clean and well shaped code
- Understand the concept of security in an OS

# Chapter organisation



## Homework:

- Total: 8
- Content: basic concepts, programming, scripting

## Labs:

- Total: 8
- Content: improve programming skills

## Projects:

- Total: 3
- Content: shell, thread communication, scheduling

Extra: Linux kernel challenges

### Grade weighting:

- Assignments: 12.5%
- Projects: 40%
- Labs: 7.5%
- Midterm exam: 20%
- Final exam: 20%

### Grade weighting:

- Assignments: 12.5%
- Projects: 40%
- Labs: 7.5%
- Midterm exam: 20%
- Final exam: 20%

### Assignment submissions:

- Late submission: -10% per day, not accepted after three days
- Dirty or hard to decipher: up to -10%

*Grades will be curved with the median in the range  $\llbracket B, B+ \rrbracket$*

## General rules:

- Not allowed:
  - Reuse the code or work from other students
  - Reuse the code or work from the internet
  - Give too many details on how to solve an exercise

## General rules:

- Not allowed:
  - Reuse the code or work from other students
  - Reuse the code or work from the internet
  - Give too many details on how to solve an exercise
- Allowed:
  - Share ideas and understandings on the course
  - Provide general directions on where or how to find information

Documents allowed during the exams: none

Group works:

- Every student in a group is responsible for his group submission
- If a student breaks the Honor Code, the whole group is sent to Honour Council

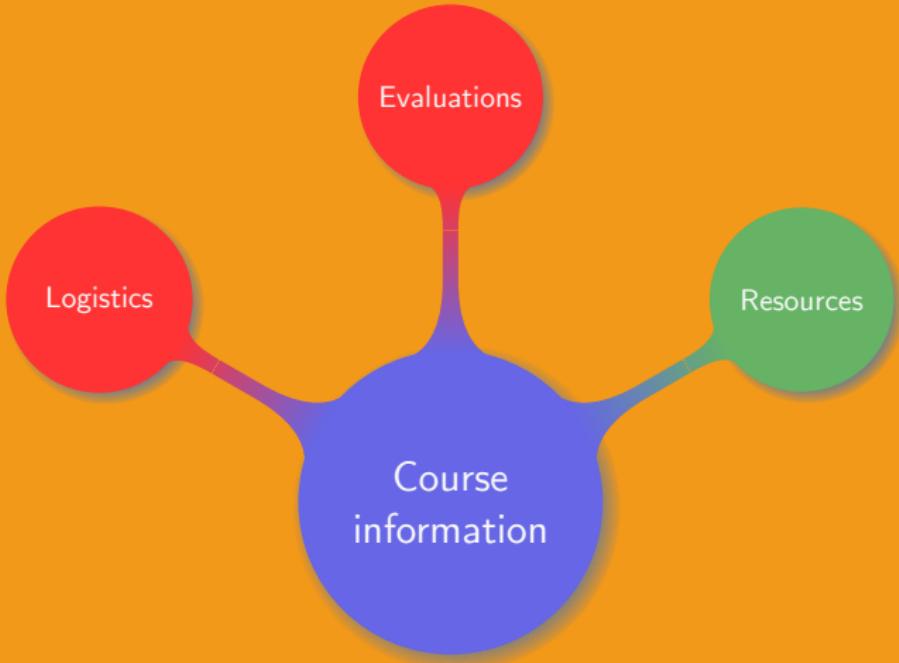
## Special circumstances

Contact us as early as possible when:

- Facing special circumstances (e.g. full time work, illness...)
- Feeling late in the course
- Feeling to work hard without any result

Any late request will be rejected

# Chapter organisation



On **Canvas** platform:

- Course materials:
  - Syllabus
  - Lecture slides
  - Homework
  - Labs
  - Projects
  - Challenges
- Course information:
  - Announcements
  - Notifications
  - Grades
  - Polls

Useful places where to find information:

- *Modern Operating Systems*, A. Tanenbaum
- *Operating System Concepts*, A. Silberschatz
- OS creation: [http://wiki.osdev.org/Main\\_Page](http://wiki.osdev.org/Main_Page)
- Piazza
- Search information online, i.e. {websites \ {local Chinese network\}}

Useful places where to find information:

- *Modern Operating Systems*, A. Tanenbaum
- *Operating System Concepts*, A. Silberschatz
- OS creation: [http://wiki.osdev.org/Main\\_Page](http://wiki.osdev.org/Main_Page)
- Piazza
- Search information online, i.e. {websites \ {local Chinese network}}

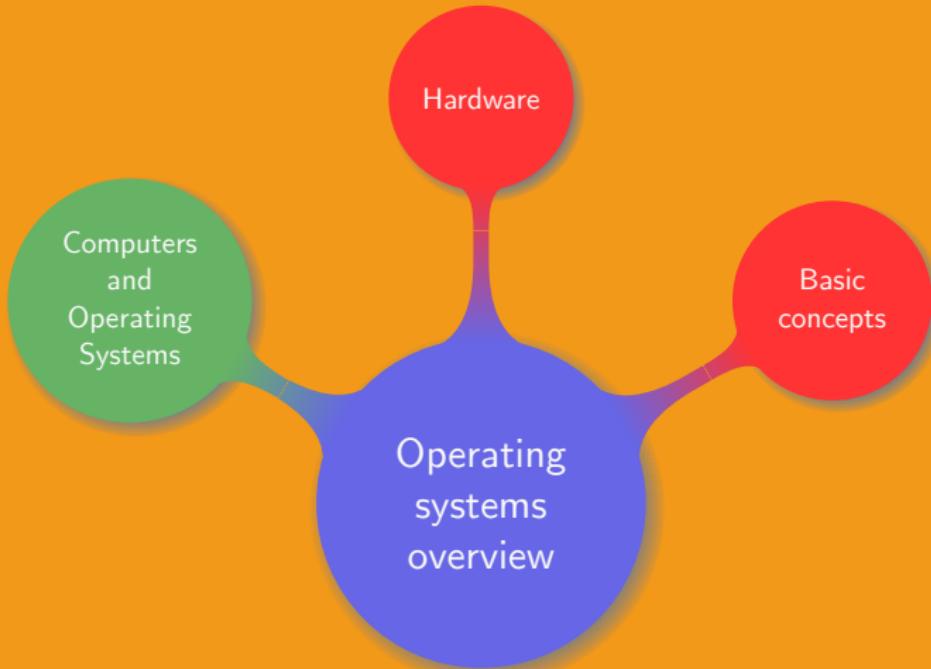
Never use Baidu in any course



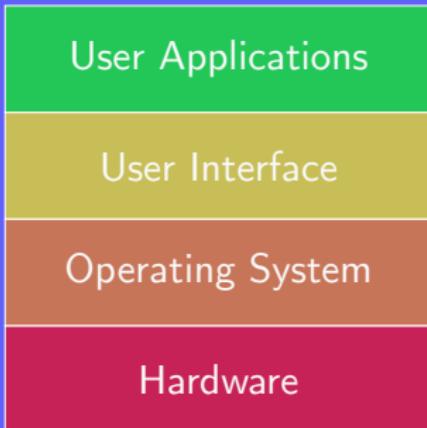
# 1. Operating systems overview

---

# Chapter organisation



# Hardware and software



A computer consists of:

- Hardware
- Software
  - Kernel mode
  - User mode

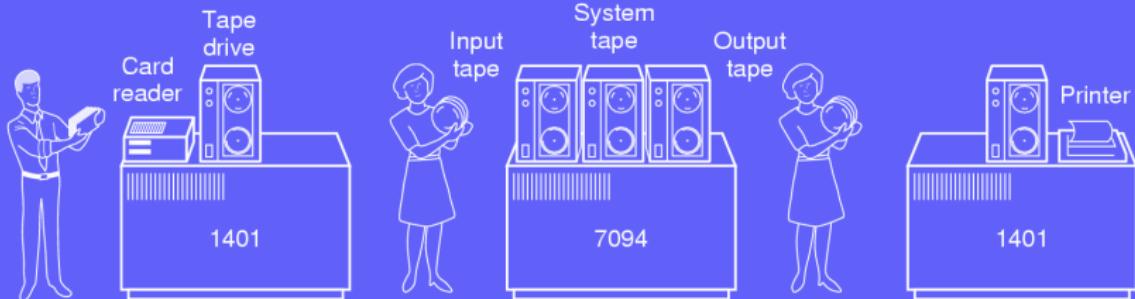
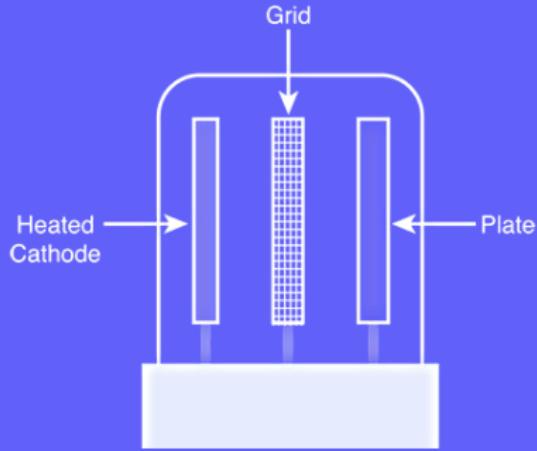
## Operating System (OS)

- Hardware is complicated to handle
- OS hides all the messy details
- OS manages resources for each program (time and space)
- Renders computer much easier to use

# A bit of history

The first days:

- Birth of modern computing:  
19th century (Babbage)
- Vacuum tube: 1945–1955  
(1st generation)
- Transistor: 1955–1965 (2nd  
generation)



# Remington Rand 409



Using the device:

- Program at most 40 steps
- Wire them on a plugboard
- Read input from cardboards
- Punch output on cardboards

## Multiprogramming: 1965–1980 (3rd generation)

- Multiple jobs kept in memory at the same time
- CPU multiplexed among them

Multiprogramming: 1965–1980 (3rd generation)

- Multiple jobs kept in memory at the same time
- CPU multiplexed among them

Multiprogramming requires:

- Memory management: allocate memory to several jobs
- CPU scheduling: choose a job to be run
- Simultaneous Peripheral Operation On Line (SPOOL): load a new job from disk, run it, output it on disk

Most famous OS:

- Disk Operating System (DOS)
- DOS/Basic package sold to computer companies
- MS-DOS, including many features from UNIX
- GUI invented in the 1960s, then copied by Apple
- Microsoft copied Apple (Windows working on top of MS-DOS)
- Many OS derived from UNIX (MINIX, LINUX, BSD...)

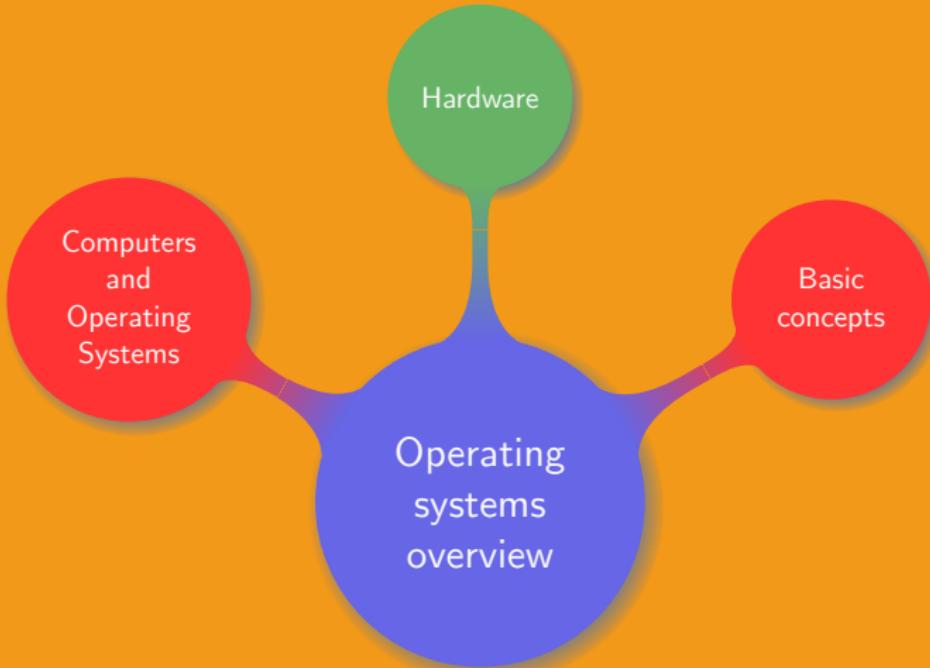
Device and task oriented OS types:

- Personal Computers (PC)
- Servers: serve users over a network (print, web, IM...) → Solaris, FreeBSD, Linux, Windows Server
- Multi processors: multiple CPU in a single system → Linux, Windows, OS X...
- Handheld computers: PDA, smartphone
- Embedded devices: TV, microwave, DVD player, mp3 player, old cell phones → everything stored in ROM, much more simple OS

More device and task oriented OS types:

- Real-Time: time is key parameter (e.g. assembly line, army, avionics...) → overlap with embedded/handheld systems
- Mainframe: room-sized computers, data centers → OS oriented toward processing many jobs at once and efficient I/O
- Sensor node: tiny computers communicating between each other and a base station (guard border, intrusion/fire detection etc...). Composed of CPU RAM ROM (+other sensors), small battery → simple OS design TinyOS
- Smart card: credit card size with a CPU chip, severe memory/processing constraints → smallest/primitive OS

# Chapter organisation



A computer is often composed of:

- CPU
- Memory
- Monitor + video controller
- Keyboard + keyboard controller
- Hard Disk Drive (HDD) + hard disk controller
- Bus

What are the controllers, and the bus?

## Basics:

- CPU is the “computer’s brain”
- CPU can only execute a specific set of instructions
- CPU fetches instructions from the memory and executes them

## Registers:

- General register: hold variables/temporary results  
e.g. program counter: address of next instruction to fetch
- Stack pointer: parameters/variables not kept in registers
- Program Status Word (PSW): control bits

# Pipeline vs. superscalar

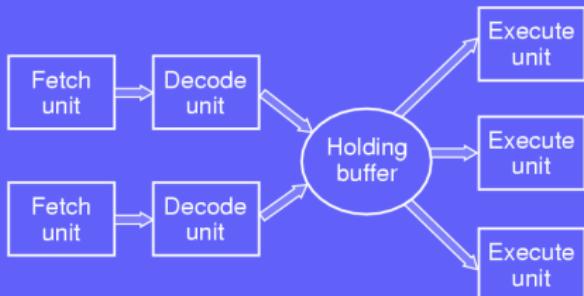


Superscalar:

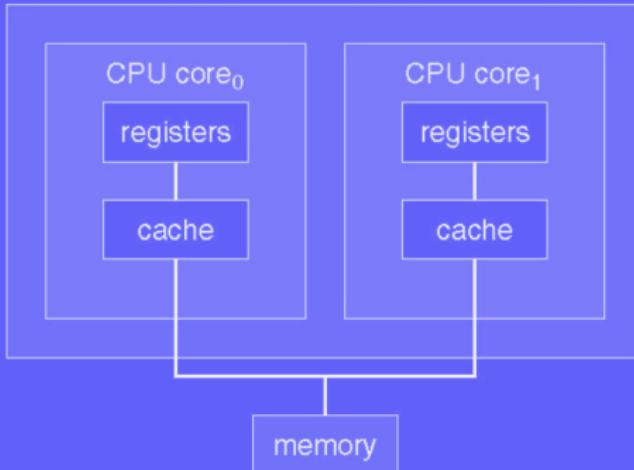
- Multiple execution units  
e.g. one for float, int, boolean
- Multiple instructions fetched and decoded at a time
- Instructions held in buffer to be executed
- Issue: no specific order to execute buffered instructions

Pipeline:

- Execute instruction  $n$ , decode  $n + 1$  and fetch  $n + 2$
- Any fetched instruction must be executed
- Issue: conditional statements



# Multi-threading



CPUs and cores:

- A CPU core can hold the state of more than one thread
- A core can switch between threads in a nanosecond time scale
- The OS sees several CPUs instead of one core
- Issue: what happens if there are more than two such cores?

Modern terminology<sup>1</sup>:

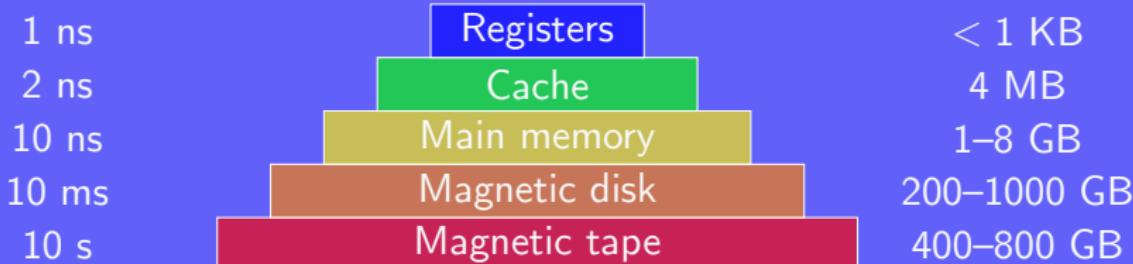
- CPU: computing component (the physical entity)
- Number of cores: number of independent CPUs in a computing component
- Number of threads: maximum number of instructions that can be passed through or processed by a single core
- Number of logical cores: number of cores times number of threads

---

<sup>1</sup>source: ARK

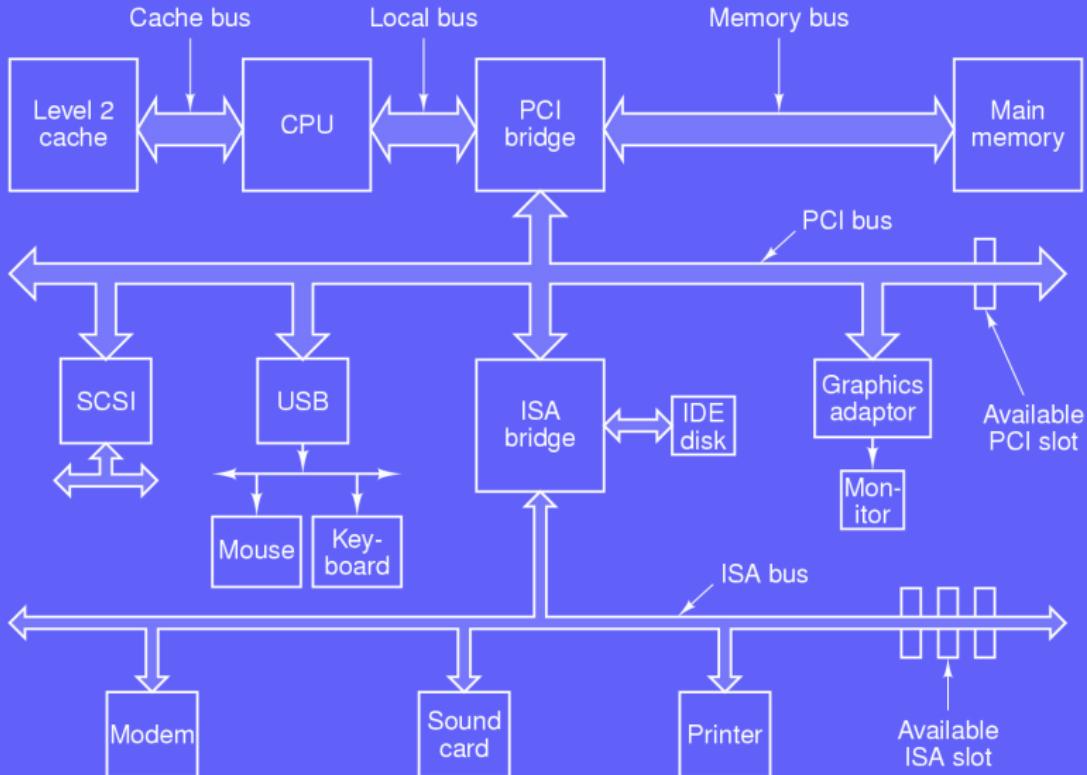
Access time

Capacity

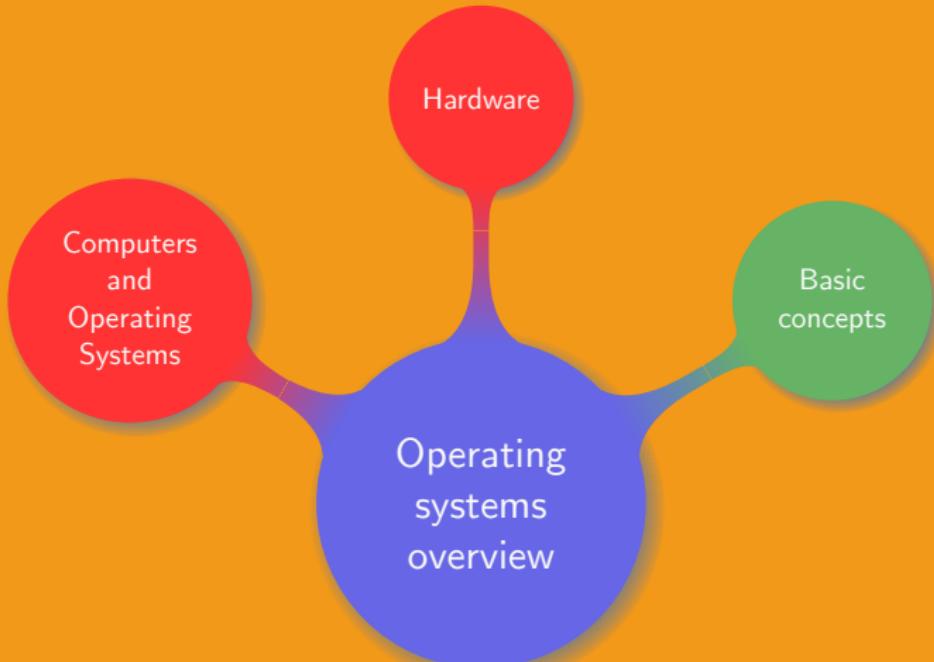


Memory types:

- Random Access Memory (RAM): volatile
- Read Only Memory (ROM)
- Electrically Erasable PROM (EEPROM) and flash memory: slower than RAM, non volatile.
- CMOS: save time and date , BIOS parameters
- HDD: divided into cylinder, track and sector



# Chapter organisation



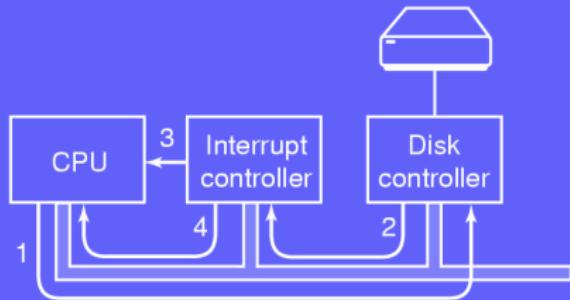
Five major components of an OS:

- Input/Output
- Protection/Security
- Processes
- File system
- System calls

## First communication strategy

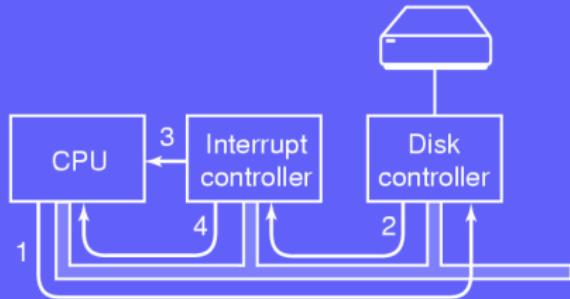
Using the device controller:

- I/O devices and CPU can execute concurrently
- Device controller in charge of a particular device
- Drive controllers have a buffer
- Buffer used to move data to/from buffer to/from memory
- Device controllers interacts with the CPU using interrupts



Starting an I/O device and getting an interrupt:

- ① Send instruction to controller
- ② Controller signals the end to interrupt controller
- ③ Assert pin to interrupt the CPU
- ④ Send extra information



Starting an I/O device and getting an interrupt:

- ① Send instruction to controller
- ② Controller signals the end to interrupt controller
- ③ Assert pin to interrupt the CPU
- ④ Send extra information

Processing an interrupt:

- ① Push user program counter and PSW onto the stack
- ② Switch to kernel mode
- ③ Find the device interrupt handler in the interrupt vector
- ④ Query the device for its status
- ⑤ Return to user program (1st instruction not yet executed)

Remarks of interrupts:

- Incoming interrupts are disabled during the process
- Software can generate interrupts: trap  
e.g. java exception, division by 0...
- An OS is almost always interrupt driven

## Second communication strategy

Simplest method:

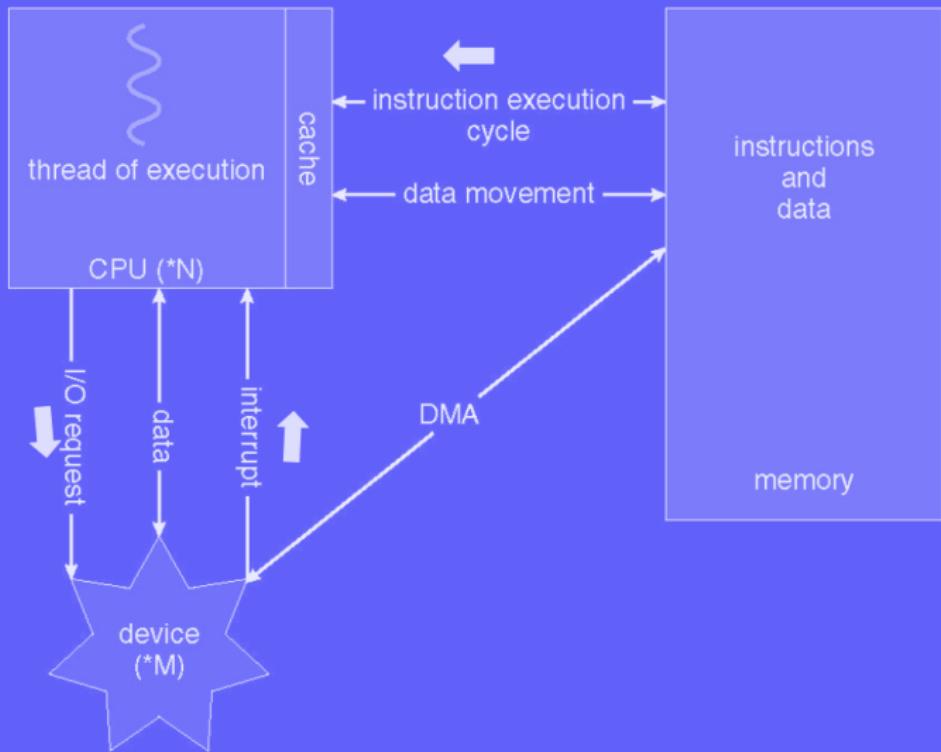
- ① Call the driver
- ② Start the I/O
- ③ Wait in a tight loop
- ④ Continuously poll the device to know its state

Disadvantage: CPU *busy waiting* until device is finished

## Third communication strategy

Direct Memory Access (DMA):

- High speed I/O devices
- Transmit information close to memory speeds
- Device controller directly transfers the blocks of data from the buffer to the main memory
- No help from the CPU
- One interrupt per block, instead of one per byte



## CPU:

- Kernel Mode:
  - Set using a bit in the PSW
  - Any CPU instruction and hardware feature are available
- User mode:
  - Only a subset of instructions/features is available
  - Setting PSW kernel mode bit forbidden

## Memory:

- Base and limit registers: holds smallest legal physical memory address and size of range, respectively
- Memory outside the address space is protected

## Input/Output:

- All I/O instructions are privilege instructions
- OS treats I/O operations to ensure correctness and legality

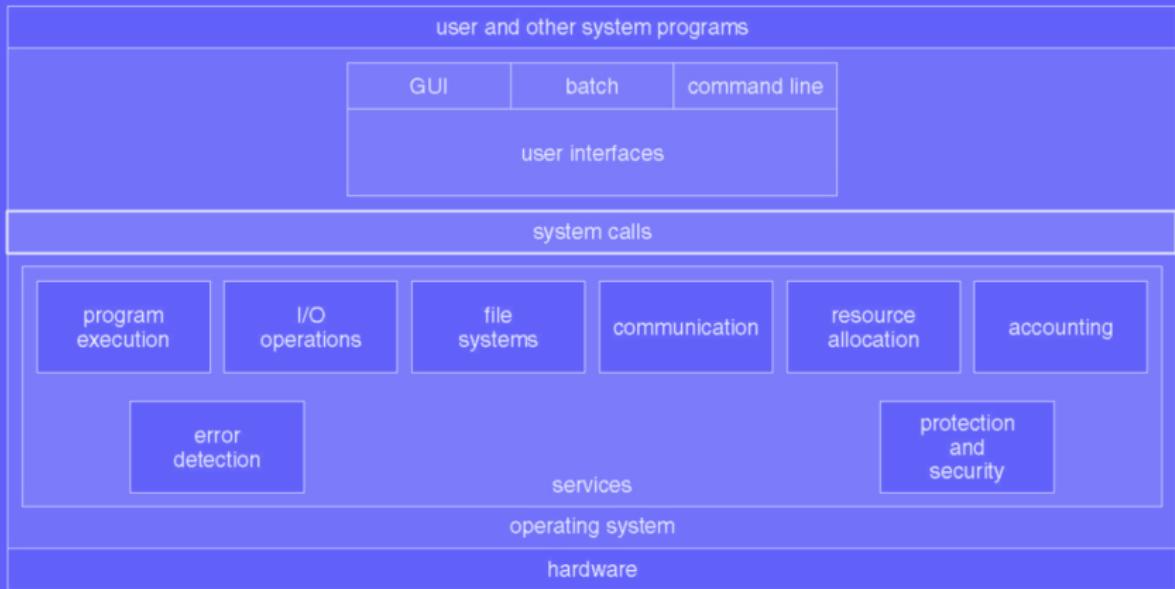
A process holds all the necessary information to run a program:

- Address space belonging to the process and containing:
  - Executable program
  - Program's data
  - Program's stack
- Set of resources:
  - Registers
  - List of open files
  - Alarms
  - List of related processes
  - Any other information required by the program

The OS hides peculiarities of the disk and other I/O devices

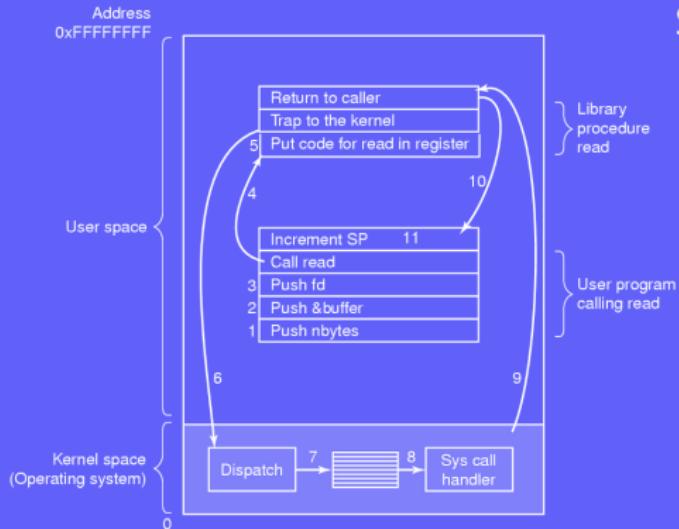
- Data stored in files
- Files are grouped inside directories
- Top directory is called root directory
- Any file can be specified using its path name
- Each process has a working directory
- Before reading/writing in a file permissions are checked
- If access is granted a file descriptor is returned
- Removable devices can be mounted on the main tree
- Block special files: represent devices such as disks
- Character special files: represent devices that accept or output character stream
- Pipe: pseudo file used to connect two processes

# System calls



# Simple example

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



Steps for a simple read:

- 1-3: push parameters on the stack
- 5-6: switch to kernel mode
- 7: use a table of pointers to system calls handler to dispatch to the correct one
- 11: increment stack pointer to remove parameters pushed (1-3)

# Common system calls

Partial list of common Unix system calls:

- Processes:

```
pid=fork(); pid=waitpid(pid, &statloc, options);  
s=execve(name, argv, environp); exit(status);
```

- Files:

```
fd=open(file,how,...); s=close(fd); s=stat(name,*buf);  
n=read(fd,buffer,nbytes); n=write(fd,buffer,nbytes);  
position=lseek(fd,offset,whence);
```

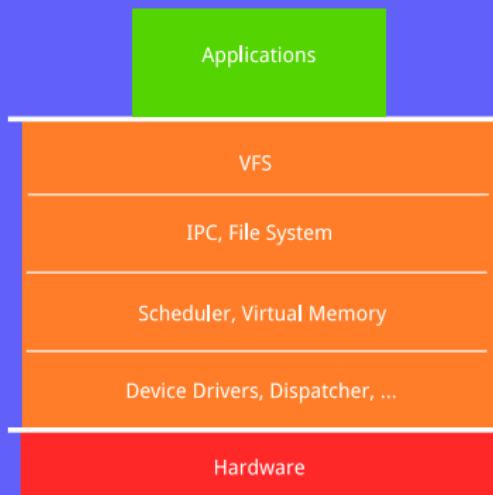
- Directory and file system:

```
s=mkdir(name,mode); s=rmdir(name); umount(name); s=unlink(name);  
mount(special,name,flags,types,args); s=link(name1,name2);
```

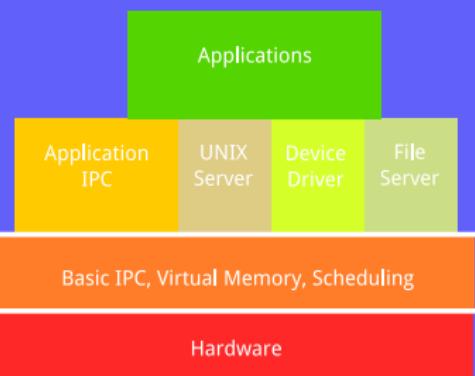
- Misc:

```
s=chdir(dirname); s=chmod(name,mode); sec=time(*t);  
s=kill(pid,signal);
```

Monolithic kernel



Micro kernel

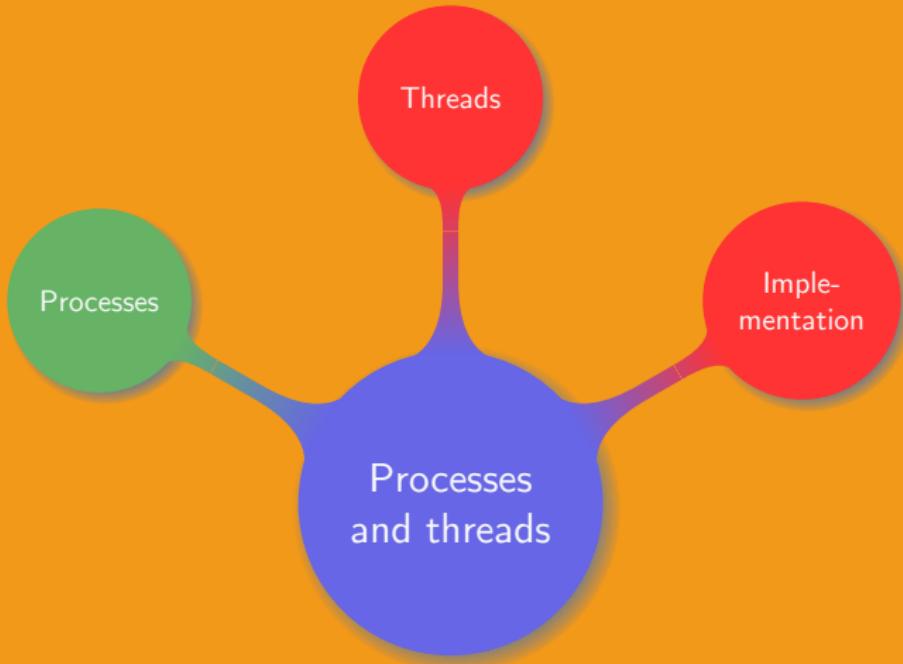




## 2. Processes and threads

---

# Chapter organisation

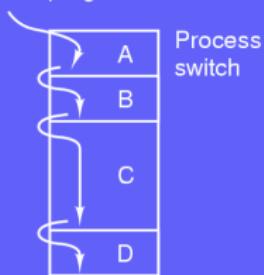


A *process* is an abstraction of a running program:

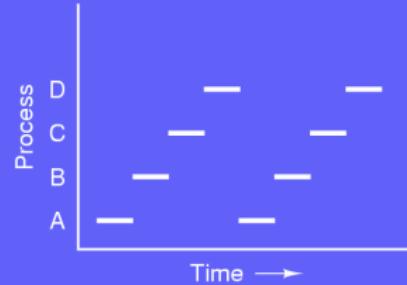
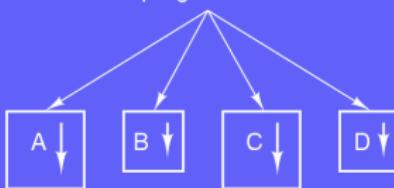
- At the core of the OS
- Process is the unit for resource management
- Oldest and most important concept
- Turn a single CPU into multiple virtual CPUs
- CPU quickly switches from process to process
- Each process run for 10-100 ms
- Processes hide the effect of interrupts

# Multiprogramming

One program counter



Four program counters



Multiprogramming strategies and issues:

- CPU switches rapidly back and forth among all the processes
- Rate of computation of a process is not uniform/reproducible
- Potential issue under time constraints; e.g.
  - Read from tape
  - Idle loop for tape to get up to speed
  - Switch to another process
  - Switch back... too late

# Program vs. process

Differences between programs and processes:

- Running twice a program generates two processes
- Program: sequence of operations to perform
- Process: program, input, output, state

# Program vs. process

Differences between programs and processes:

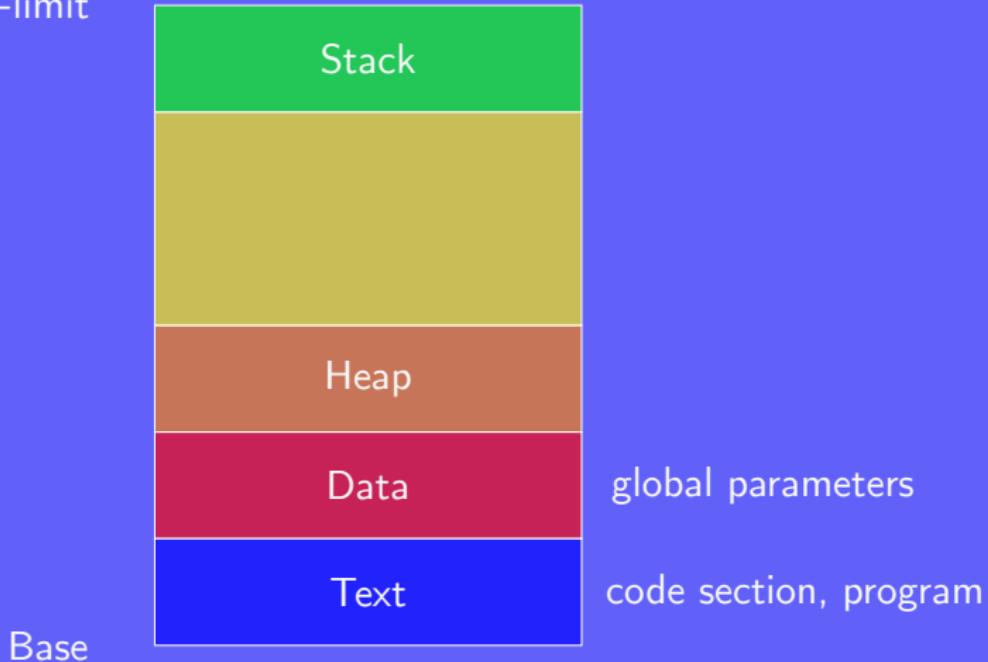
- Running twice a program generates two processes
- Program: sequence of operations to perform
- Process: program, input, output, state

Example.

Describe the process of baking a cake when the phone rings...

# Process in memory

Base+limit



Four main events causing process to be created:

- System initialization
- Execution of a “process creation” system call
- A user requests a new process
- Initiation of a batch job

Unix like systems:

- Creating a new process is done using one system call: `fork()`
- The call creates an exact clone of the calling process
- Child process executes `execve` to run a new program

Windows system:

- Function call `CreateProcess`, creates a new process and loads the program in the new process
- This call takes 10 parameters

Parent and child have their own address space and a change in one is invisible to the other

Any created processes ends at some stage:

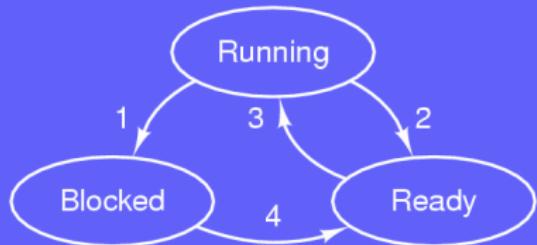
- Normal exit (voluntary)  
work is done, execute a system call to tell OS it is finished  
`exit`, `ExitProcess`
- Error exit (voluntary)  
e.g. requested file does not exist
- Fatal error (involuntary)  
e.g. referencing non existent memory, dividing by 0
- Killed by another process (involuntary)  
`kill`, `TerminateProcess`

Two main approaches:

- UNIX like systems:
  - Parent creates a child
  - Child can create its own child
  - The hierarchy is called *process group*
  - Impossible to disinherit a child
- Windows system:
  - All processes are equal
  - A parent has a token to control its child
  - Token can be given to another process

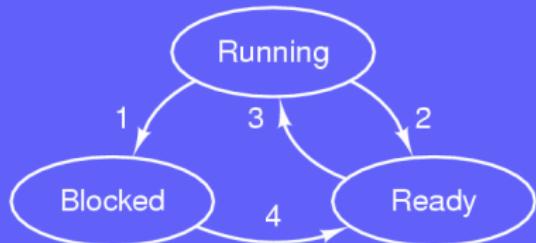
# Process states

Possible states:



- ① Waiting for some input
- ② Scheduler picks another process
- ③ Scheduler picks this process
- ④ Input becomes available

Possible states:



- ① Waiting for some input
- ② Scheduler picks another process
- ③ Scheduler picks this process
- ④ Input becomes available

Change of perspective on the inside of the OS:

- Do not think in terms of interrupt but of process
- Lowest level of the OS is the scheduler
- Interrupt handling, starting/stopping processes are hidden in the scheduler

A simple model for processes:

- Each process is represented using a structure called *process control block*
- The structure contains important information such as:
  - State
  - Program counter
  - Stack pointer
  - Memory allocation
  - Open files
  - Scheduling information
  - ...
- All the processes are stored in an array called *process table*

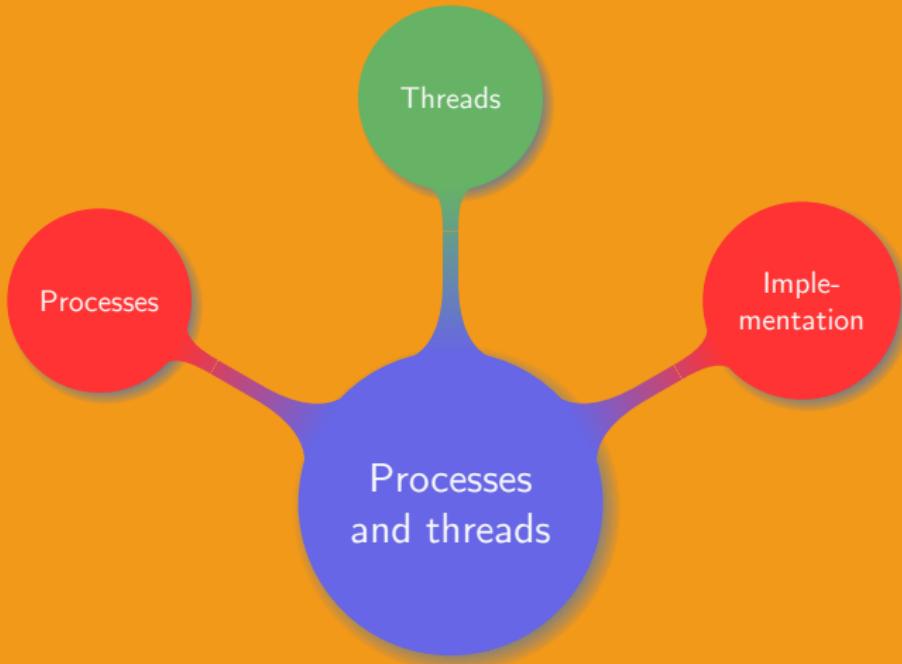
# Modeling processes

Process management	Memory management	File management
registers	pointer to text segment info	root directory
program counter	pointer to data segment info	working directory
program status word	pointer to stack segment info	file descriptors
stack pointer		user ID
process state		group ID
priority		
scheduling parameters		
process ID		
parent process		
process group		
signals		
starting time		
CPU time used		
children's CPU time		
next alarm		

Lowest OS level:

- ① Push user program counter, PSW... on stack
- ② Hardware loads information from interrupt vector
- ③ Save registers (assembly)
- ④ Setup new stack (assembly)
- ⑤ Finish up the work for the interrupt
- ⑥ Scheduler decides which process to run next
- ⑦ Load and run the “new current process”, i.e. memory map, registers... (assembly)

# Chapter organisation



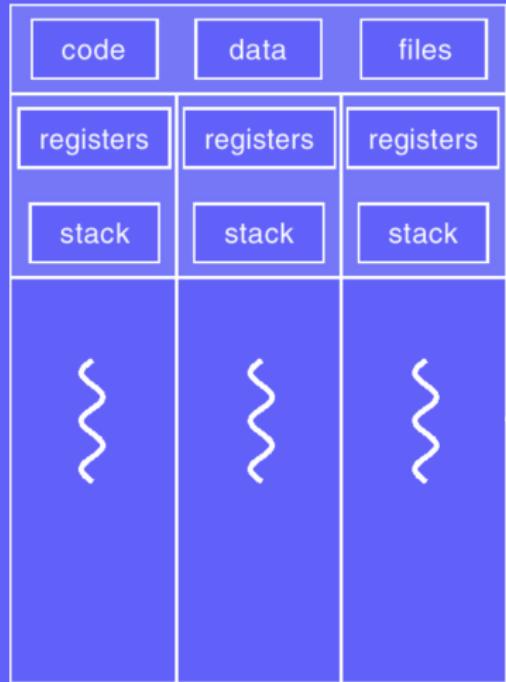
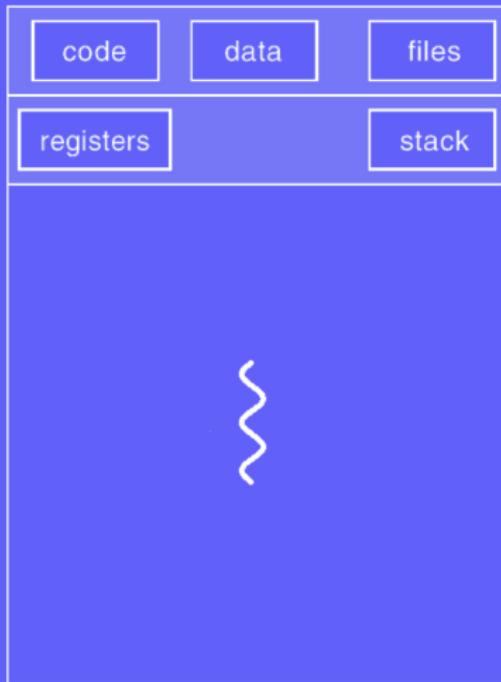
A thread is the basic unit of CPU utilisation consisting of:

- Thread ID
- Program counter
- Register set
- Stack space

All the threads within a process share:

- Code section
- Data section
- OS resources

# Single vs. multi-threaded



## Processes and threads:

- A thread has the same possible states as a process
- Transitions are similar to the case of a process
- Threads are sometimes called lightweight processes
- No protection is required for threads, compared to processes
- A process starts with one threads and can create more
- Processes want as much CPU as they can
- Threads can give up the CPU to let others using it

# Example

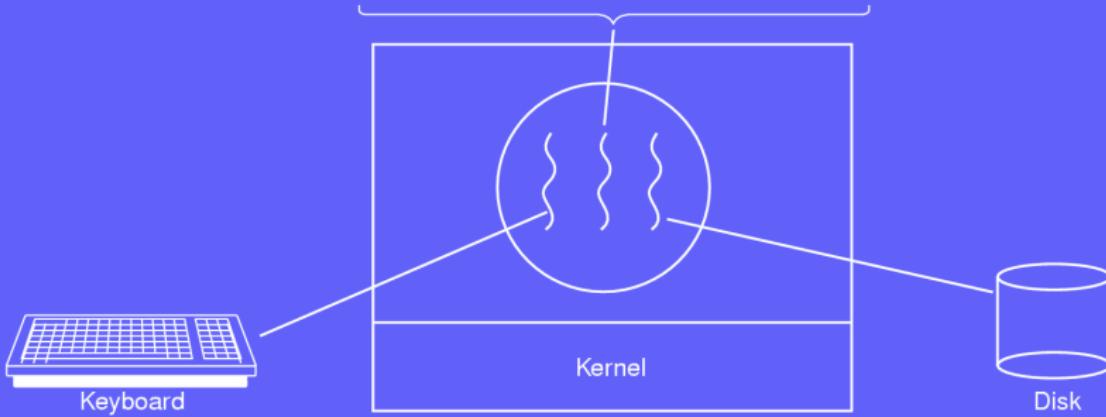
Patriots and slaves  
long ago, our fathers  
brought forth upon this  
continent a nation  
conceived in liberty,  
and dedicated to the proposition  
that all men are created equal.  
Now we are engaged  
in a great civil war,  
testing whether that

nation, or any nation  
so conceived and so  
dedicated, can long  
endure. We are met on  
a great battlefield of  
a nation, and we  
have come to dedicate  
a portion of that  
battlefield, as a final  
resting place for those  
who have given their

lives that this nation  
might live. It is  
for us, therefore, to  
be dedicated here  
to the unfinished  
business which they  
brought with them  
and which we  
cannot dedicate, we  
cannot consecrate, we  
cannot hallow. The brave  
men, living and dead,

who struggled here  
have consecrated it, far  
more than we shall ever  
be able to do. The world  
will little note  
what we say here, but  
it can never forget  
what they did here.  
It is for us the living  
hereafter to be dedicated  
to that cause for which

they gave the last full  
measure of devotion,  
that we here highly  
revere that these dead  
shall not have died in  
vain; that this nation,  
under God, shall have  
a new birth of freedom  
and that government of  
the people, by the  
people, for the people.



# Multi-threading problems

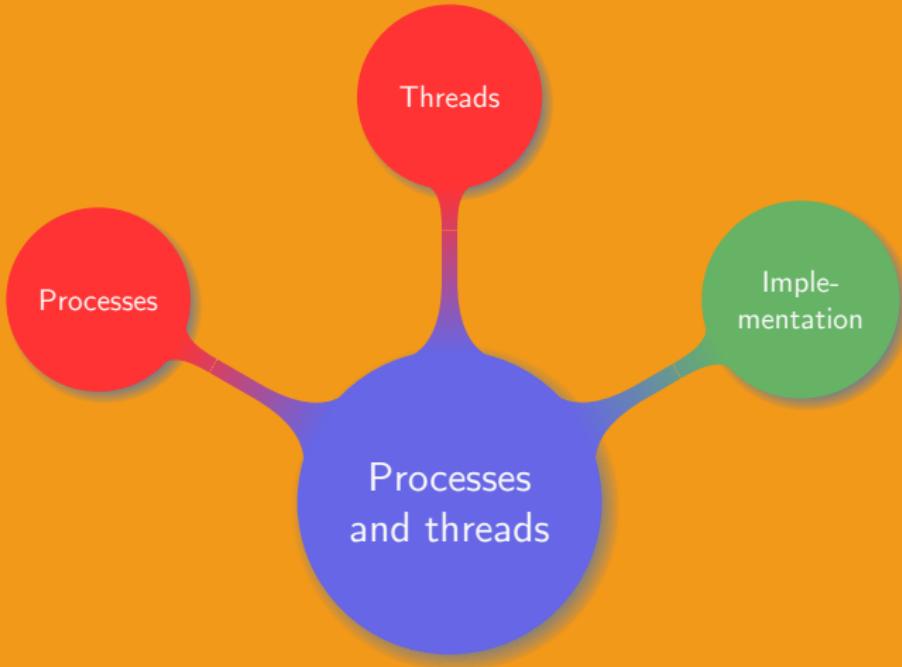
Threads share many data structures:

- A thread could close a file that another thread is reading
- A thread notices a lack of memory and allocates more. A thread switch occurs, the new threads also notices the lack of memory and also allocates some

Should a child inherit all the threads from its parents?

- No: the child might not function properly
- Yes: if a parent thread was waiting for some keyboard input, who gets it?

# Chapter organisation



The pthread library has over 60 function calls, important ones are:

- Create a thread:

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

- Terminate a thread:

```
void pthread_exit(void *retval);
```

- Wait for a specific thread to end:

```
int pthread_join(pthread_t thread, void **retval);
```

- Release CPU to let another thread run:

```
int pthread_yield(void);
```

- Create and initialise a thread attribute structure:

```
int pthread_attr_init(pthread_attr_t *attr);
```

- Delete a thread attribute object:

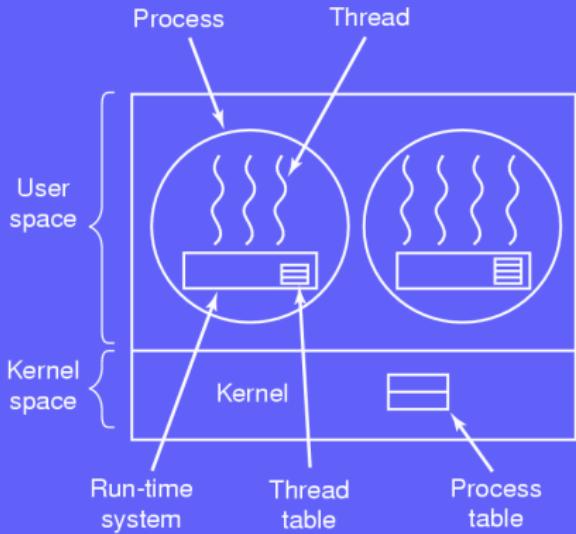
```
int pthread_attr_destroy(pthread_attr_t *attr);
```

Write a program that creates 10 threads, and prints their ID.

## threads.c

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <pthread.h>
4 #define THREADS 10
5 void *gm(void *tid) {
6     printf("Good morning from thread %lu\n",*(unsigned long int*)tid);
7     pthread_exit(NULL);
8 }
9 int main () {
10     int status, i; pthread_t threads[THREADS];
11     for(i=0;i< THREADS;i++) {
12         printf("thread %d\n",i);
13         status=pthread_create(&threads[i],NULL,gm,(void*)&(threads[i]));
14         if(status!=0) {
15             fprintf(stderr,"thread %d failed with error %d\n",i,status);
16             exit(-1);
17         }
18     }
19     exit(0);
20 }
```

# Threads in user-space – N:1



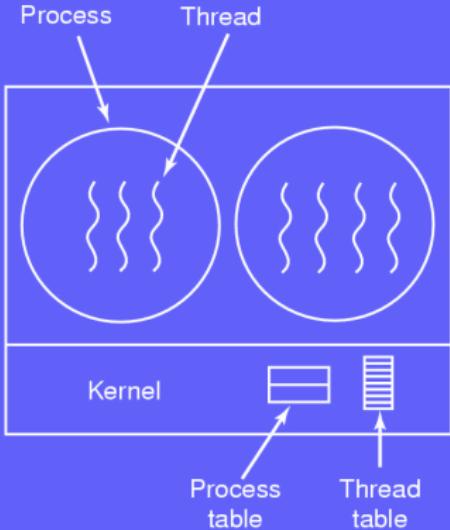
## User-space threads:

- Kernel thinks it manages single threaded processes
- Threads implemented in a library
- Thread table similar to process table, managed by run-time system
- Switching thread does not require to trap the kernel

## Questions.

- What if a thread issues a blocking system call?
- Threads within a process have to voluntarily give up the CPU

# Thread in the kernel – 1:1



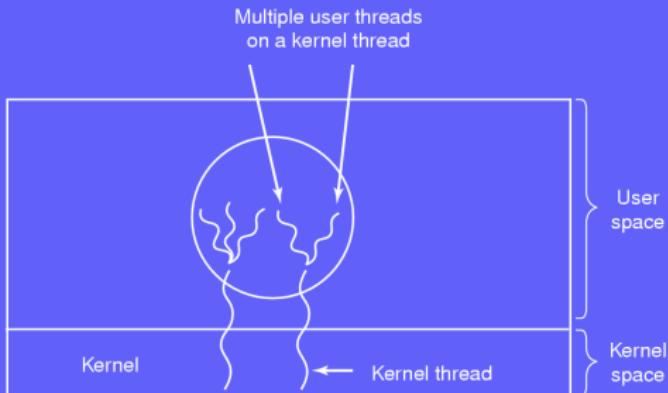
Kernel space thread:

- Kernel manages the thread table
- Kernel calls are issued to request a new thread
- Calls that might block a thread are implemented as system call
- Kernel can run another thread in the meantime

Questions.

- Why does it have a much higher cost than user space threads?
- Signals are sent to processes, which thread should receive it?

# Hybrid threads – M:N



## Hybrid threads:

- Compromise between user-level and kernel-level
- Threading library schedules user threads on available kernel threads

## Questions.

- How to implement hybrid threads?
- How to handle scheduling?

Best thread approach:

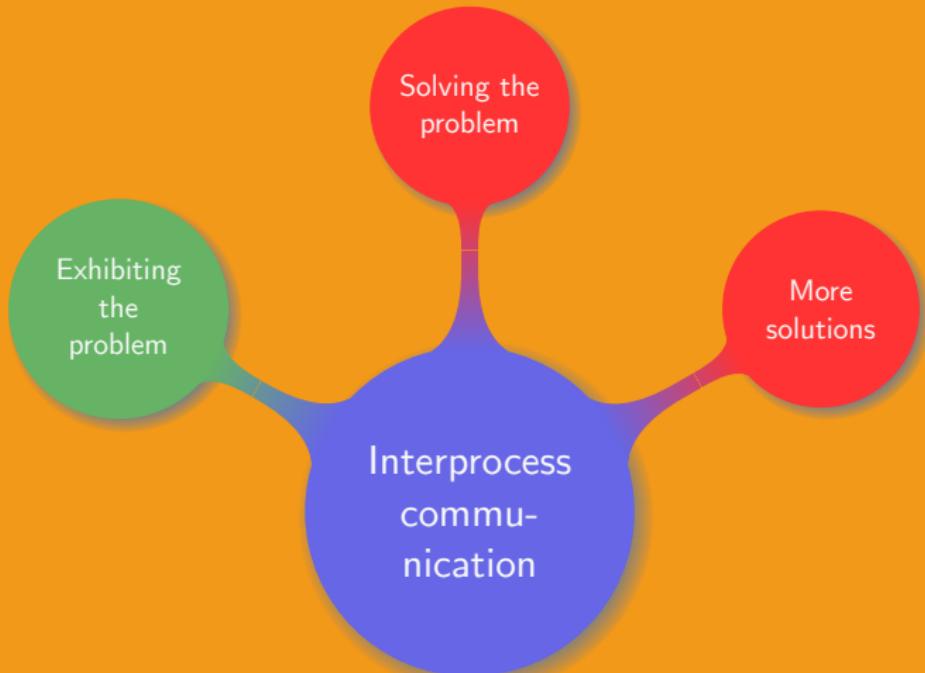
- Hybrid looked attractive
- Most systems are coming back to 1:1
- Different approaches exist on how to use threads
  - e.g. thread blocks on “receive system call” vs. pop up threads
- Switching implementation from single thread to multiple thread is not easy task
- Requires redesigning the whole system
- Backward compatibility must be preserved
- Research still going on to find better ways to handle threads



### 3. Interprocess communication

---

# Chapter organisation



Independent threads:

- Cannot affect/be affected by anything
- State not shared with other threads
- Input state determines the output
- Reproducible
- No side effect when stopping/resuming

Single-tasking: run a thread to completion and start the next one

Difficulties appear with multi-tasking:

- One core shared among several threads
- Several cores run several threads in parallel
- A thread runs on one core at a time
- A thread can run on different cores at different times
- The number of cores has no impact on the running of the threads

# Thread and process cooperation

Setup for threads:

- Several threads share a common global variable
- The execution sequence impacts the global variable
- By default the behavior is random and irreproducible

# Thread and process cooperation

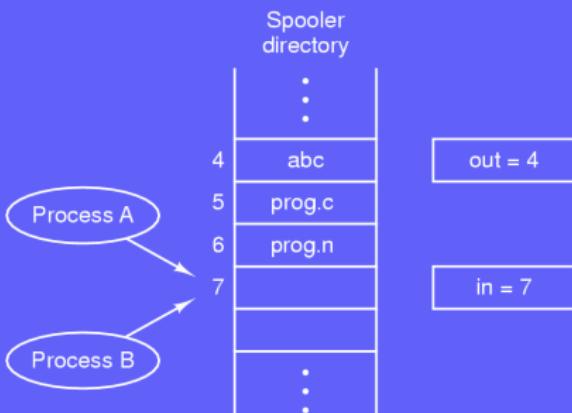
Setup for threads:

- Several threads share a common global variable
- The execution sequence impacts the global variable
- By default the behavior is random and irreproducible

Major problems:

- How can threads or processes share information?
- How to prevent them from getting on each other's way?
- How to ensure an acceptable running order?

# Race conditions



In the printing spool:

- ① A wants to queue a file: reads `next_free_slot=7`
- ② An interrupt occurs
- ③ B wants to queue a file: reads `next_free_slot=7`
- ④ B queues its file in slot 7, and updates `next_free_slot=8`
- ⑤ A queues its file in slot 7

# Too much milk



9:00 am

9:15 am

9:30 am

9:45 am

# Too much milk



9:00 am

9:15 am

9:30 am

9:45 am

# Too much milk



9:00 am

9:15 am

9:30 am

9:45 am

# Too much milk



9:00 am

9:15 am

9:30 am

9:45 am

# Too much milk



9:00 am

9:15 am

9:30 am

9:45 am

# Too much milk



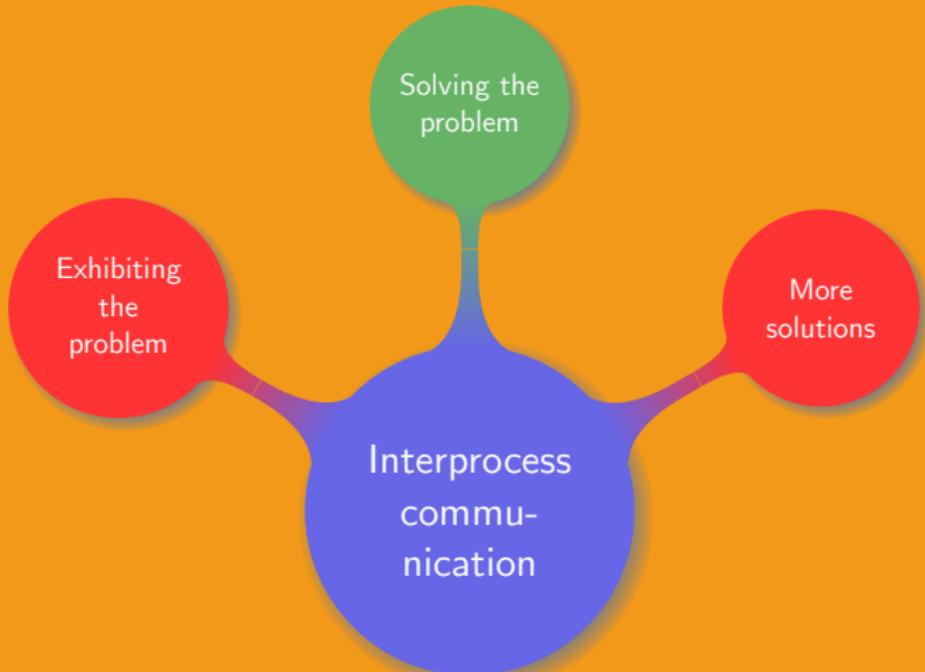
9:00 am

9:15 am

9:30 am

9:45 am

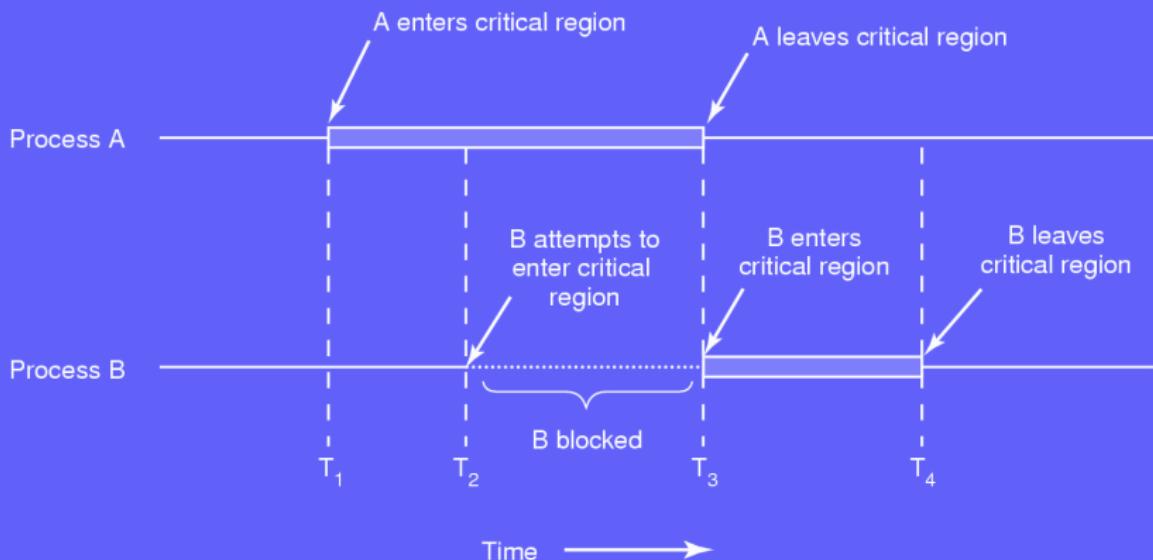
# Chapter organisation



Part of the program where shared memory is accessed:

- No two processes can be in a critical region at a same time
- No assumption on the speed or number of CPUs
- No process outside a critical region can block other processes
- No process waits forever to enter a critical region

# Critical region



# Too much milk

**Frank**

```
1 if(no milk && no note) {  
2     leave note;  
3     milk the cow;  
4     remove note;  
5 }
```

**John**

```
1 if(no milk && no note) {  
2     leave note;  
3     milk the cow;  
4     remove note;  
5 }
```

# Too much milk

Frank

```
1 if(no milk && no note) {  
2   leave note;  
3   milk the cow;  
4   remove note;  
5 }
```

John

```
1 if(no milk && no note) {  
2   leave note;  
3   milk the cow;  
4   remove note;  
5 }
```

```
1 leave note Frank;  
2 if(no note John) {  
3   if(no milk) milk the cow;  
4 }  
5 remove note Frank;
```

```
1 leave note John;  
2 if(no note Frank) {  
3   if(no milk) milk the cow;  
4 }  
5 remove note John;
```

*What is the issue with those two strategies?*

# Too much milk

Frank

```
1 leave note Frank;  
2 while(note John) {  
3     nothing;  
4 }  
5 if(no milk) {  
6     milk the cow;  
7 }  
8 remove note Frank;
```

John

```
1 leave note John;  
2 if(no note Frank) {  
3     if(no milk) {  
4         milk the cow;  
5     }  
6 }  
7 remove note John;
```

*How good is this strategy?*

Symmetric strategy for two processes:

- When wanting to enter a critical region a process:
  - Shows its interest for the critical region
  - If it is accessible it exits the function and accesses it
  - If it is not accessible it waits in a tight loop
- When a process has completed its work in the critical region it signals its departure

*What is the main drawback of this strategy?*

*Pseudo C code for two processes represented as 0 and 1*

```
1 #define TRUE 1
2 #define FALSE 0
3 int turn;
4 int interested[2];
5 void enter_region(int p) {
6     int other;
7     other=1-p;
8     interested[p]=TRUE;
9     turn=p;
10    while(turn==p && interested[other]==TRUE)
11    }
12 void leave_region(int p) {
13     interested[p]=FALSE;
14 }
```

Side effects of Peterson's idea:

- Two processes: L, low priority, and H, high priority
- L enters in a critical region
- H becomes ready
- H has higher priority so the scheduler switches to H
- L has lower priority so is not rescheduled as long as H is busy
- H loops forever

# Mutual exclusion at hardware level

Prevent the process in the critical region from being stopped:

- Disable interrupts:
  - Can be done within the kernel for a few instructions
  - Cannot be done by user processes
  - Only works when there is a single CPU
  - An interrupt on another CPU can still mess up the shared variable
- Use atomic operations:
  - Either happens in its entirety or not at all
  - Several operations can be performed at once, e.g.  $A = B$
  - Requires the CPU to support the atomic update of a memory space
  - Can be used to prevent other CPUs to access a shared memory

# Using the TSL instruction

A simple atomic operation:

- Test and Set Lock: TSL
- Copies LOCK to a register and set it to 1
- LOCK is used to coordinate the access to a shared memory
- Ensures LOCK remains unchanged while checking its value

```
1 enter_region:  
2   TSL REGISTER,LOCK  
3   CMP REGISTER,#0  
4   JNE enter_region  
5   RET  
6  
7 leave_region:  
8   MOVE LOCK,#0  
9   RET
```

# Using the TSL instruction

A simple atomic operation:

- Test and Set Lock: TSL
- Copies LOCK to a register and set it to 1
- LOCK is used to coordinate the access to a shared memory
- Ensures LOCK remains unchanged while checking its value

```
1 enter_region:  
2   TSL REGISTER,LOCK  
3   CMP REGISTER,#0  
4   JNE enter_region  
5   RET  
6  
7 leave_region:  
8   MOVE LOCK,#0  
9   RET
```

*Is this strategy better than Peterson's idea?*

# Consumer-producer problem

```
1 #define N 100
2 int count=0;
3 void producer() {
4     int item;
5     while(1) {
6         item=produce_item(); if(count==N) sleep();
7         insert_item(item); count++;
8         if(count==1) wakeup(consumer);
9     }
10 }
11 void consumer() {
12     int item;
13     while(1) {
14         if(count==0) sleep();
15         item=remove_item(); count--;
16         if(count==N-1) wakeup(producer); consume_item(item);
17     }
18 }
```

*Is this code exhibiting any problem?*

# Consumer-producer problem

Assume the buffer is empty:

- Consumer reads count == 0
- Scheduler stops the consumer and starts the producer
- Producer adds one item
- Producer wakes up the consumer
- Consumer not yet asleep, signal is lost
- Consumer goes asleep
- When the buffer is full the producer falls asleep
- Both consumer and producer sleep forever

## Basics:

- Introduced by Dijkstra in 1965
- Simple hardware based solution
- Basis of all modern OS synchronization mechanisms

## Basics:

- Introduced by Dijkstra in 1965
- Simple hardware based solution
- Basis of all modern OS synchronization mechanisms

A semaphore `sem` is:

- A positive integer variable
- Only changed or tested through two actions

```
1 down(sem) {  
2     while(sem==0) sleep();  
3     sem--;  
4 }
```

```
1 up(sem) {  
2     sem++;  
3 }
```

# Semaphores and processes

## The down operation

- If  $sem > 0$ , decrease it and continue
- If  $sem = 0$ , sleep and do not complete the down

## The up operation

- Increment the value of the semaphore
- An awaken sleeping process can complete its down

# Semaphores and processes

## The down operation

- If  $sem > 0$ , decrease it and continue
- If  $sem = 0$ , sleep and do not complete the down

## The up operation

- Increment the value of the semaphore
- An awaken sleeping process can complete its down

Checking or changing the value and sleeping are done atomically:

- Single CPU: disable interrupts
- Multiple CPUs: use TSL to ensure only one CPU accesses the semaphore

*Is disabling the interrupts to process the semaphore an issue?*

## Semaphores and interrupts

Using semaphores to hide interrupts:

- Each I/O device gets a semaphore initialised to 0
- A process accessing the device applies a down
- The process becomes blocked
- An interrupt is issued when the device has completed the work
- The interrupt handler processes the interrupt and applies an up
- The process becomes ready

*A mutex is a semaphore taking values 0 (unlocked) or 1 (locked)*

On a mutex-lock request:

- If the mutex is unlocked:
  - Lock the mutex
  - Enter the critical region
- If mutex is locked: put the calling thread asleep
- When the thread in the critical region exits:
  - Unlock the mutex
  - Allow a thread to acquire the lock and enter the critical region

# Mutex implementation

*Mutexes can be implemented in user-space using TSL*

```
1  mutex-lock:  
2      TSL REGISTER,MUTEX  
3      CMP REGISTER,#0  
4      JNE ok  
5      CALL thread_yield  
6      JMP mutex-lock  
7  ok: RET  
8  
9  mutex-unlock:  
10     MOVE MUXEX,#0  
11     RET
```

Questions:

- What differences were introduced compared to enter\_region (3.99)?
- In user-space what happens if a thread tries to acquire lock through busy-waiting?
- Why is `thread_yield` used?

# Consumer-producer problem

## consumer\_producer.c

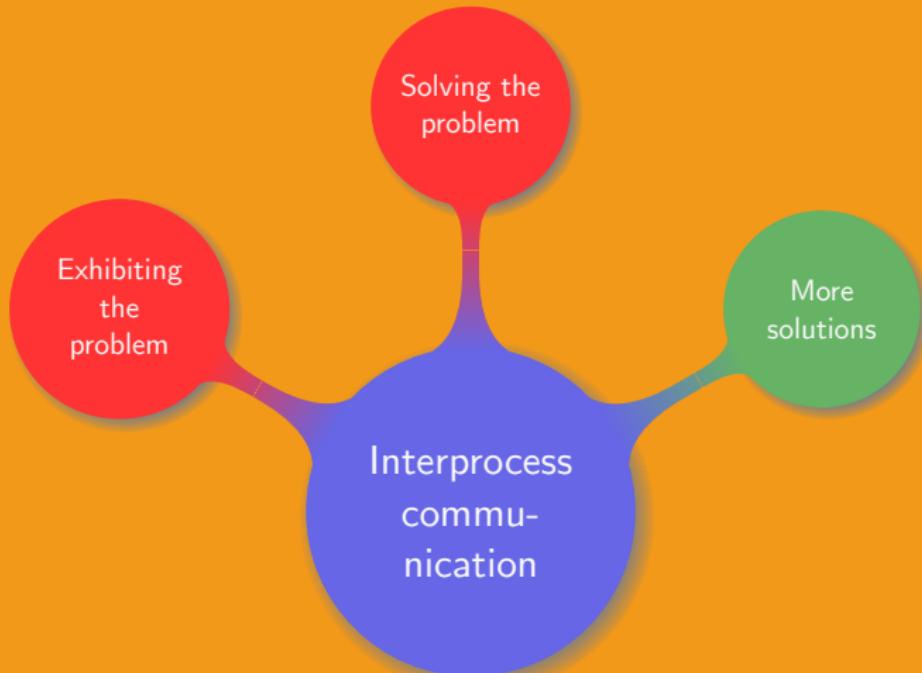
```
1 #include <stdio.h>
2 #include <pthread.h>
3 #define MAX 1000
4 pthread_mutex_t m; pthread_cond_t cc, cp; int buf=0;
5 void *prod() {
6     for(int i=1;i<MAX;i++) {
7         pthread_mutex_lock(&m); while(buf!=0) pthread_cond_wait(&cp,&m);
8         buf=1; pthread_cond_signal(&cc); pthread_mutex_unlock(&m);
9     }
10    pthread_exit(0);
11}
12 void *cons() {
13     for(int i=1;i<MAX;i++) {
14         pthread_mutex_lock(&m); while(buf==0) pthread_cond_wait(&cc,&m);
15         buf=0; pthread_cond_signal(&cp); pthread_mutex_unlock(&m);
16     }
17    pthread_exit(0);
18}
19 int main() {
20     pthread_t p, c;
21     pthread_mutex_init(&m,0); pthread_cond_init(&cc,0); pthread_cond_init(&cp,0);
22     pthread_create(&c,0,cons,0); pthread_create(&p,0,prod,0);
23     pthread_join(p,0); pthread_join(c,0);
24     pthread_cond_destroy(&cc); pthread_cond_destroy(&cp); pthread_mutex_destroy(&m);
25 }
```

## Consumer-producer problem

Alter the previous program such as:

- To display information on the consumer and producer
- To increase the buffers to 100
- To have two consumers and one producer. In this case also print which consumer is active.

# Chapter organisation



# Consumer-producer problem

```
1 mutex mut = 0; semaphore empty = 100; semaphore full = 0;
2 void producer() {
3     while(TRUE) {
4         item = produce_item();
5         mutex-lock(&mut);
6         down(&empty); insert_item(item);
7         mutex-unlock(&mut);
8         up(&full);
9     }
10 }
11 void consumer() {
12     while(TRUE) {
13         down(&full);
14         mutex-lock(&mut); item = remove_item(); mutex-unlock(&mut);
15         up(&empty); consume_item(item);
16     }
17 }
```

*Is this code working as expected?*

## Consumer-producer problem

In the previous code:

- What is the behavior of the producer when the buffer is full?
- What about the consumer?
- What is the final result for this program?
- How to fix it?

*Monitors are an attempt to merge synchronization with OOP*

Basic idea behind monitors:

- Programming concept that must be known by the compiler
- The mutual exclusion is not handled by the programmer
- Locking occurs automatically
- Only one process can be active within a monitor at a time
- A monitor can be seen as a “special type of class”
- Processes can be blocked and awaken based on condition variables and wait and signal functions

# Consumer-producer problem

```
1 monitor ProducerConsumer {  
2     condition full, empty;  
3     int count;  
4     void insert(item) {  
5         if (count == N) wait(full);  
6         insert_item(item);  
7         count++;  
8         if (count==1) signal(empty);  
9     }  
10    void remove() {  
11        if (count==0) wait(empty);  
12        removed = remove_item;  
13        count--;  
14        if (count==N-1) signal(full);  
15    }  
16    count:= 0;  
17 }
```

```
1 void ProducerConsumer::producer() {  
2     while (TRUE) {  
3         item = produce_item();  
4         ProducerConsumer.insert(item);  
5     }  
6 }  
7 void ProducerConsumer::consumer() {  
8     while (TRUE) {  
9         item=ProducerConsumer.remove();  
10        consume_item(item);  
11    }  
12 }
```

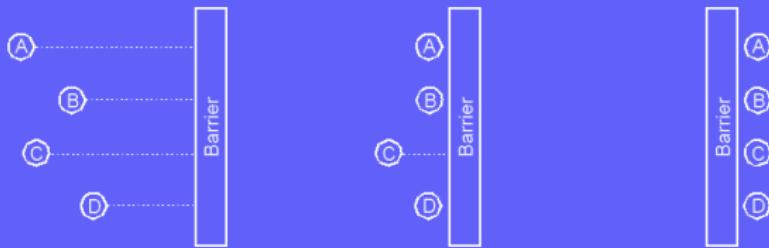
*Over a distributed system semaphores and mutexes cannot be used*

Message passing strategy:

- `send(destination,&message)`
- `receive(source,&message)`: blocks or exit if nothing is received

Potential issues:

- Messages can get lost, e.g. sending or acknowledging reception
- Confusion on the process names
- Security, e.g. authentication, traffic encryption
- Performance



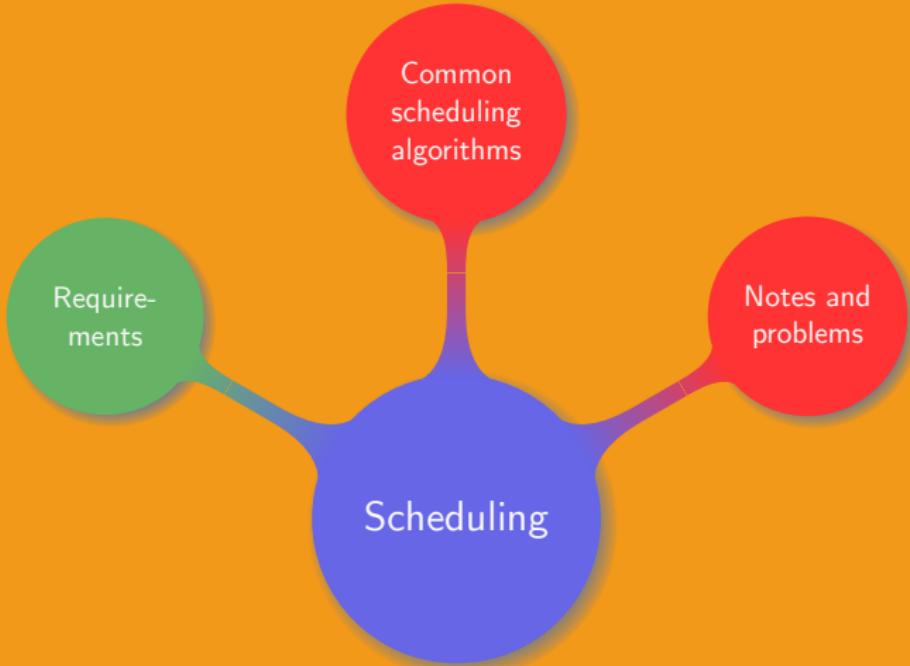
*Useful when several processes must complete before the next phase*



## 4. Scheduling

---

# Chapter organisation



Scheduler's job:

- Multiple processes competing for using the CPU
- More than one process in ready state
- Which one to select next?
- Key issue in terms of “perceived performance”
- Need “clever” and efficient scheduling algorithms

Switching process is expensive:

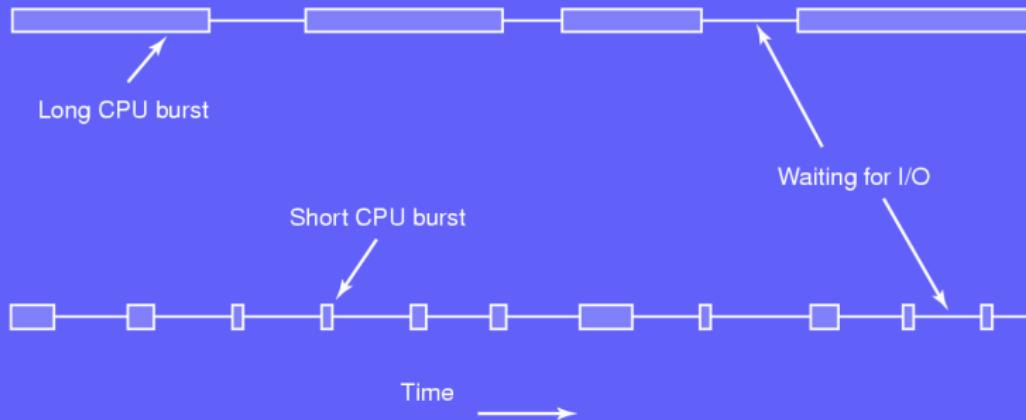
- Switch from user mode to kernel mode
- Save state of current process (save register, memory map...)
- Run scheduling algorithm to select a new process
- Remap the memory address for the new process (convert address generated by the program into physical memory address in RAM); done by the Memory Management Unit (MMU)
- Start new process

Switching process is expensive:

- Switch from user mode to kernel mode
- Save state of current process (save register, memory map...)
- Run scheduling algorithm to select a new process
- Remap the memory address for the new process (convert address generated by the program into physical memory address in RAM); done by the Memory Management Unit (MMU)
- Start new process

*Too many switches per second wastes much CPU*

## Process behavior



Typical behavior:

- Process runs for a while
- System call emitted to read/write from/in a file
- More general: process in blocked state until external device has completed its work

Compute bound vs. I/O bound:

- Most time spent computing vs. waiting for I/O
- Length of the CPU burst:
  - I/O time is constant
  - Processing data is not constant
- As CPUs get faster processes are more and more I/O bound

Compute bound vs. I/O bound:

- Most time spent computing vs. waiting for I/O
- Length of the CPU burst:
  - I/O time is constant
  - Processing data is not constant
- As CPUs get faster processes are more and more I/O bound

**Conclusion:** I/O bound processes should be run quickly, such as to issue their I/O request and keep the disk busy

### Basic scheduling issues:

- Creation of a new process: both the parent and the child are in ready state
- A process exits: which process to run next? i
- A process blocks (semaphore, I/O...), what process to run?
- I/O interrupt from a device that has completed its task: run the newly ready process, or another one?

## Preemptive vs. non-preemptive

Two main strategies for scheduling algorithms:

- ① Preemptive: a process is run for at most  $n$  ms; if it is not done at the end of the period then it is suspended and another process is selected and run: require a clock interrupt
  
- ② Non-preemptive: a process runs until it blocks or voluntarily releases the CPU; it is resumed after an interrupt unless another process with higher priority is in the queue

# What is the goal?

All systems:

- Fairness: fair share of the CPU for each process
- Policy enforcement: following the defined policy
- Balance: all parts of the system are busy

# What is the goal?

All systems:

- Fairness: fair share of the CPU for each process
- Policy enforcement: following the defined policy
- Balance: all parts of the system are busy

Batch systems:

- Throughput: maximise the number of jobs per hour
- Turnaround time: minimise the time between submission and termination of a job
- CPU utilisation: keep the CPU as busy as possible

# What is the goal?

Interactive systems:

- Response time: quickly respond to requests
- Proportionality: meet user's expectations

# What is the goal?

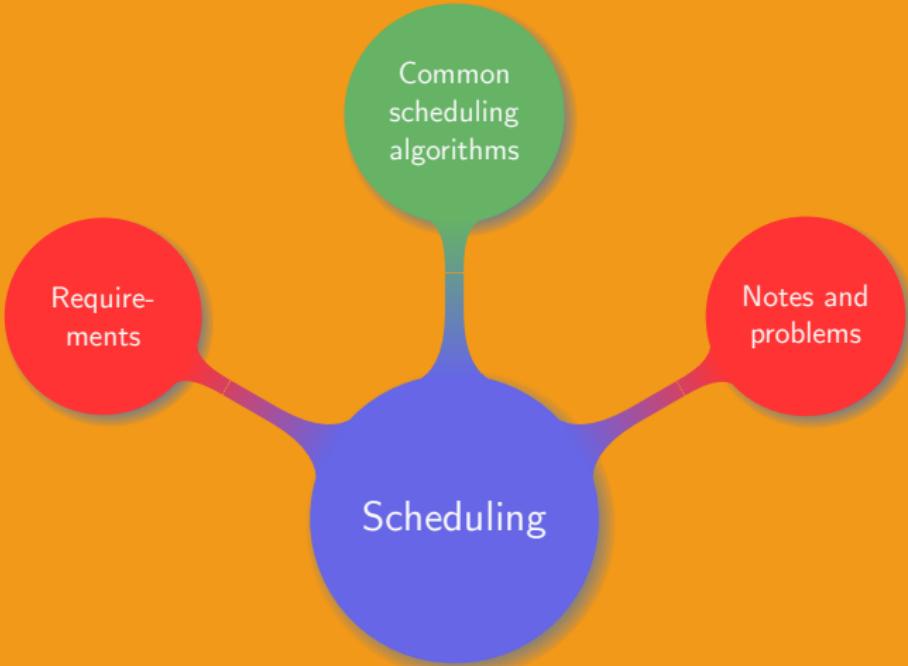
Interactive systems:

- Response time: quickly respond to requests
- Proportionality: meet user's expectations

Real-time systems:

- Meeting deadlines: avoid losing data
- Predictability: avoid quality degradation for multimedia

# Chapter organisation



**Basics:** simplest algorithm, non-preemptive

- CPU is assigned in the order it is requested
- Processes are not interrupted, they can run a long as they want
- New jobs are put at the end of the queue
- When a process blocks the next in line is run
- When a blocked process becomes ready it is put at the end of the queue

**Basics:** simplest algorithm, non-preemptive

- CPU is assigned in the order it is requested
- Processes are not interrupted, they can run a long as they want
- New jobs are put at the end of the queue
- When a process blocks the next in line is run
- When a blocked process becomes ready it is put at the end of the queue

**Characteristics:** good for batch systems, implemented using a single linked list

**Drawback:** what happens when there is one compute-bound process and many I/O-bound ones?

# Shortest job first

**Basics:** Non-preemptive, run times are known in advance

8	4	4	4
A	B	C	D

4	4	4	8
B	C	D	A

- Run time: A: 8 min, B: 4 min, C: 4 min, D: 4 min
- Turnaround time:  
$$\frac{8+12+16+20}{4} = 14 \text{ min}$$
- Run time: B: 4 min, C: 4 min, D: 4 min, A: 8 min
- Turnaround time:  
$$\frac{4+8+12+20}{4} = 11 \text{ min}$$

# Shortest job first

**Basics:** Non-preemptive, run times are known in advance

8	4	4	4
A	B	C	D

4	4	4	8
B	C	D	A

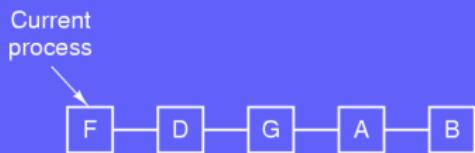
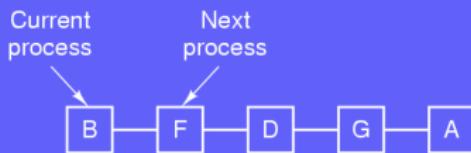
- Run time: A: 8 min, B: 4 min, C: 4 min, D: 4 min
- Turnaround time:  
$$\frac{8+12+16+20}{4} = 14 \text{ min}$$
- Run time: B: 4 min, C: 4 min, D: 4 min, A: 8 min
- Turnaround time:  
$$\frac{4+8+12+20}{4} = 11 \text{ min}$$

**Characteristics:** good for batch systems, very specific to a system

**Drawback:** what happens when the jobs are not all in a queue at the beginning?

# Round-Robin scheduling

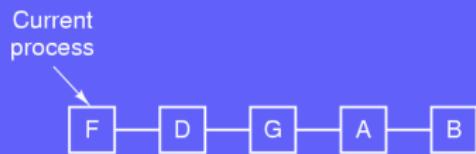
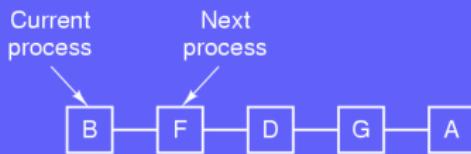
**Basics:** preemptive, old, simple, fair, most widely used



- Each process is assigned a time interval called *quantum*
- A process runs until (i) it blocks, (ii) it is finished or (iii) its quantum is over
- Switching process is then done

# Round-Robin scheduling

**Basics:** preemptive, old, simple, fair, most widely used



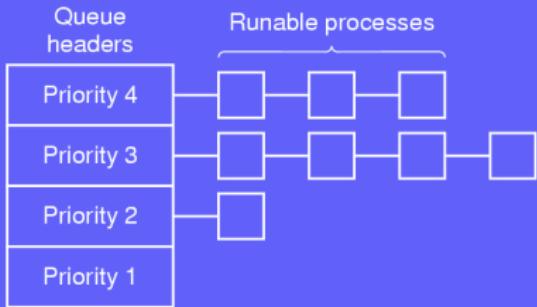
- Each process is assigned a time interval called *quantum*
- A process runs until (i) it blocks, (ii) it is finished or (iii) its quantum is over
- Switching process is then done

**Characteristics:** good for interactive systems, easy to implement (only needs to maintain a list of runnable processes)

**Drawback:** how long should a quantum be?

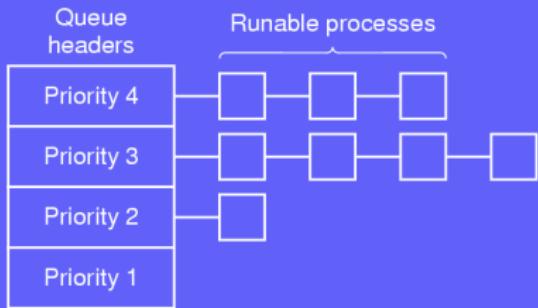
# Priority scheduling

**Basics:** preemptive, priority depending on who or what runs



- Processes are more or less important (e.g. printing...)
- Creates priority classes
- Use Round-Robin within a class
- Run higher priority processes first

**Basics:** preemptive, priority depending on who or what runs



- Processes are more or less important (e.g. printing...)
- Creates priority classes
- Use Round-Robin within a class
- Run higher priority processes first

**Characteristics:** good for interactive systems, flexible, adjustable

**Drawback:** what happens if many high-priority processes run for a long time?

**Basics:** preemptive, extends priority scheduling

- Processes get “lottery tickets”
- When a scheduling decision is made a random ticket is chosen
- Price for the winner is to access resources
- High priority processes get more tickets (higher probability of winning)

**Basics:** preemptive, extends priority scheduling

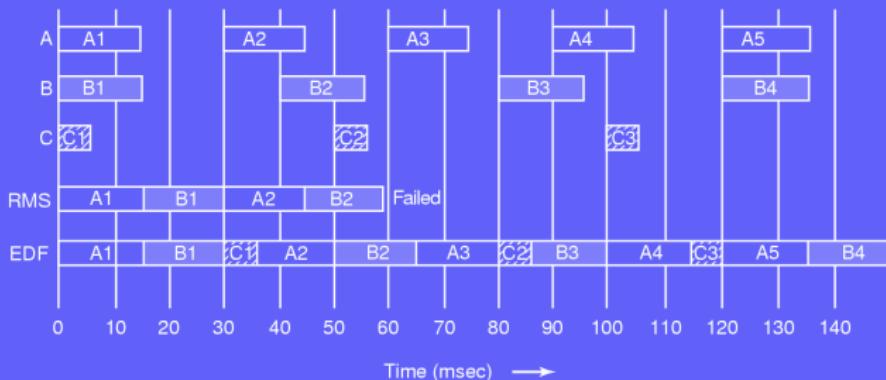
- Processes get “lottery tickets”
- When a scheduling decision is made a random ticket is chosen
- Price for the winner is to access resources
- High priority processes get more tickets (higher probability of winning)

**Characteristics:** good for interactive systems, highly responsive, possibility of cooperation between processes

**Drawback:** what if a “low level user” runs many small processes instead of a big one?

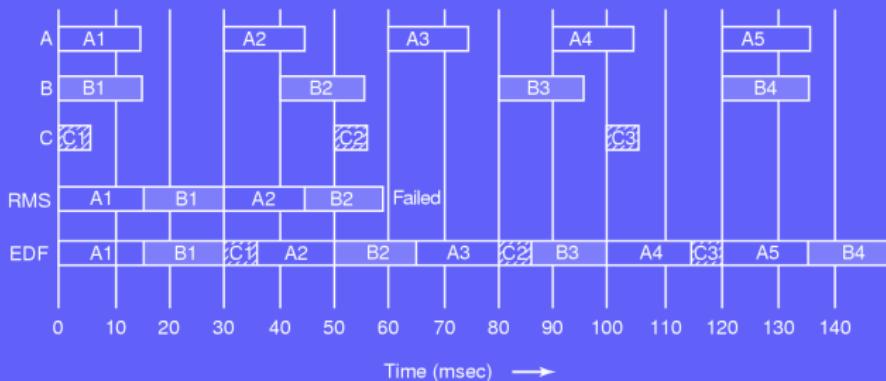
# Earliest deadline first

**Basics:** preemptive, priority based



- Process needs to announce (i) its presence and (ii) its deadline
- Scheduler orders processes with respect to their deadline
- First process in the list (earliest deadline) is run

**Basics:** preemptive, priority based

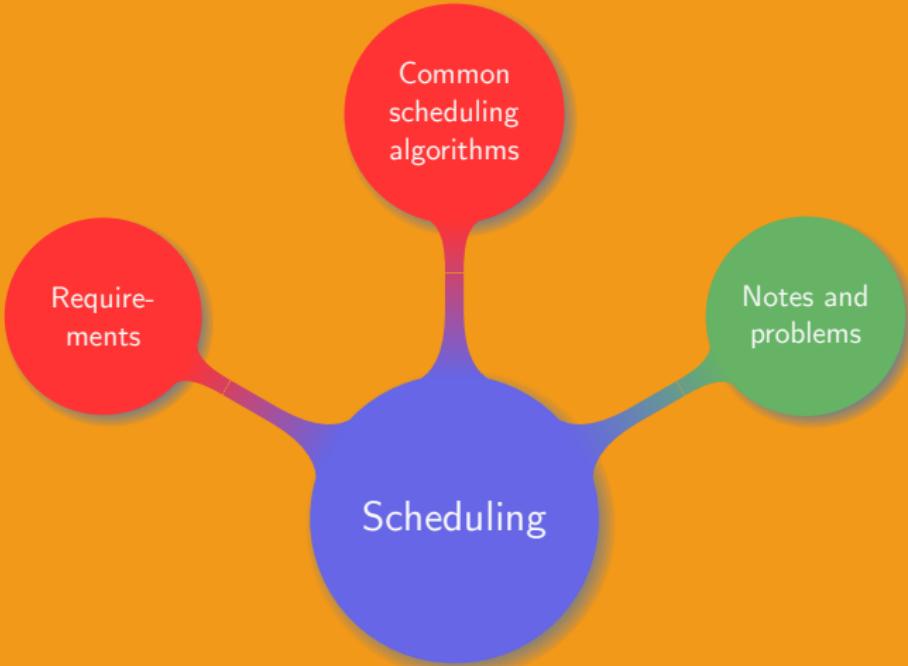


- Process needs to announce (i) its presence and (ii) its deadline
- Scheduler orders processes with respect to their deadline
- First process in the list (earliest deadline) is run

**Characteristics:** good for realtime systems, can fully use the CPU

**Drawback:** complex to implement

# Chapter organisation



## Policy vs. mechanism

Limitations of the previous algorithms:

- They all assume that processes are competing
- Parent could know which of its children is most important

## Policy vs. mechanism

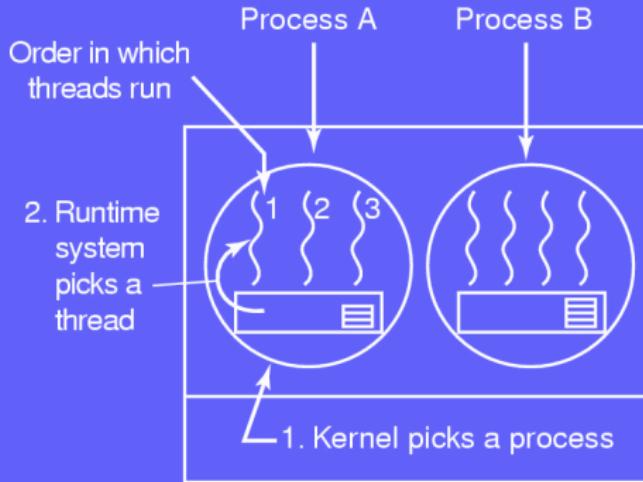
Limitations of the previous algorithms:

- They all assume that processes are competing
- Parent could know which of its children is most important

The solution consists in separating the scheduling mechanism from the scheduling policy:

- Scheduling algorithm has parameters
- Parameters can be set by processes
- A parent can decide which of its children should have higher priority

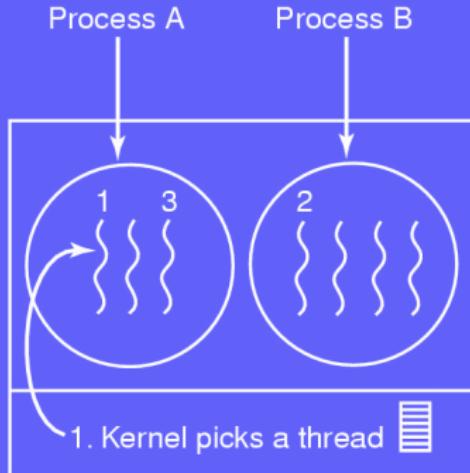
# Threads scheduling – User-level



Possible: A1, A2, A3, A1, A2, A3

Impossible: A1, B1, A2, B2, A3, B3

# Threads scheduling – Kernel-level



Possible: A1, A2, A3, A1, A2, A3

Also possible: A1, B1, A2, B2, A3, B3

# The dining philosophers problem



Synchronisation problem:

- A philosopher is either thinking or eating
- When he is hungry he takes:
  - ① His left chop-stick
  - ② His right chop-stick
- Eats
- Puts down his chop-sticks
- Thinks

# The dining philosophers problem

First obvious solution:

- Wait for a chop-stick to be available
- Seize it as soon as it becomes available

# The dining philosophers problem

First obvious solution:

- Wait for a chop-stick to be available
- Seize it as soon as it becomes available

**Problem:** they all take the left chop-stick at the same time and wait forever for the right one

# The dining philosophers problem

First obvious solution:

- Wait for a chop-stick to be available
- Seize it as soon as it becomes available

**Problem:** they all take the left chop-stick at the same time and wait forever for the right one

Second solution:

- Take left chop-stick
- If right chopstick not available put down the left one
- Wait for some time and repeat the process

# The dining philosophers problem

First obvious solution:

- Wait for a chop-stick to be available
- Seize it as soon as it becomes available

**Problem:** they all take the left chop-stick at the same time and wait forever for the right one

Second solution:

- Take left chop-stick
- If right chopstick not available put down the left one
- Wait for some time and repeat the process

**Problem:** all process start at the same time, then nobody ever eats

# The dining philosophers problem

What about using mutex?

# The dining philosophers problem

What about using mutex?

- Philosopher thinks
- Lock mutex
- Acquire chop-sticks, eat, put down chop-sticks
- Unlock mutex

# The dining philosophers problem

What about using mutex?

- Philosopher thinks
- Lock mutex
- Acquire chop-sticks, eat, put down chop-sticks
- Unlock mutex

**Problem:** how many philosophers can eat at the same time?

# The dining philosophers problem

```
1 #define N 5
2 #define LEFT (i+N-1)%N
3 #define RIGHT (i+1)%N
4 enum { THINKING, HUNGRY, EATING };
5 int state[N]; mutex mut = 0 ; semaphore s[N];
6 void philosopher(int i) {while(TRUE) {think();take_cs(i);eat();put_cs(i);}}
7 void take_cs(int i) {
8     mutex-lock(&mut);
9     state[i] = HUNGRY; test(i);
10    mutex-unlock(&mut); down(&s[i]);
11 }
12 void put_cs(int i) {
13     mutex-lock(&mut);
14     state[i] = THINKING; test(LEFT); test(RIGHT);
15     mutex-unlock(&mut);
16 }
17 void test(int i) {
18     if(state[i]==HUNGRY && state[LEFT]!=EATING && state[RIGHT]!=EATING;) {
19         state[i]=EATING; up(&s[i]); }
20 }
```



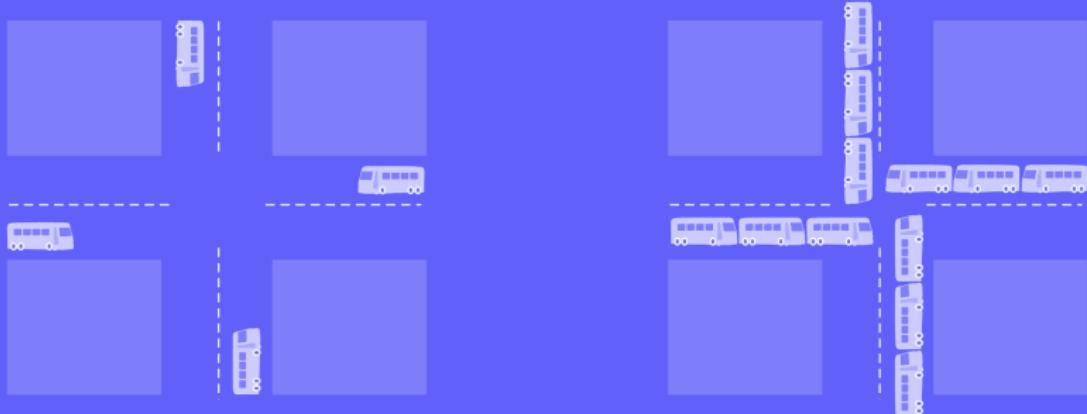
## 5. Deadlocks

---

# Chapter organisation



# What is a deadlock?



## A typical deadlock

Simple example:

- Process *A* needs to access resource *R*
- Process *B* needs to access resource *S*
- Process *A* wants to access resource *S*
- Process *B* wants to access resource *R*

## A typical deadlock

Simple example:

- Process *A* needs to access resource *R*
- Process *B* needs to access resource *S*
- Process *A* wants to access resource *S*
- Process *B* wants to access resource *R*

**Problem:** neither *A* nor *B* releases the resources, both *A* and *B* will wait indefinitely

# Types of resources

## Preemptable

(Resources that can be safely taken away from a process)

- Total memory: 256 MB
- Processes *A* and *B*, 256MB each
- *A* loaded in memory and acquires the printer
- *A* exceeds its quantum
- *B* loaded in memory and tries to acquire the printer
- *B* fails, but has the memory
- Memory given to *A*
- *A* finishes its printing

## Non-preemptable

(Resources that cannot be safely taken away from a process)

- *A* is burning a DVD
- *B* wants the DVD
- DVD drive is not accessible
- Resource cannot be taken away from *A*

# Representing deadlocks

Resource allocation represented using graphs:

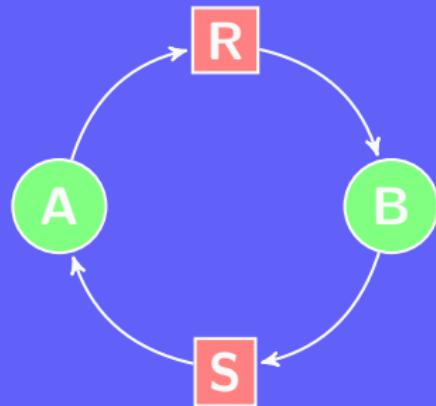
Resource  $R$   
held by  $A$



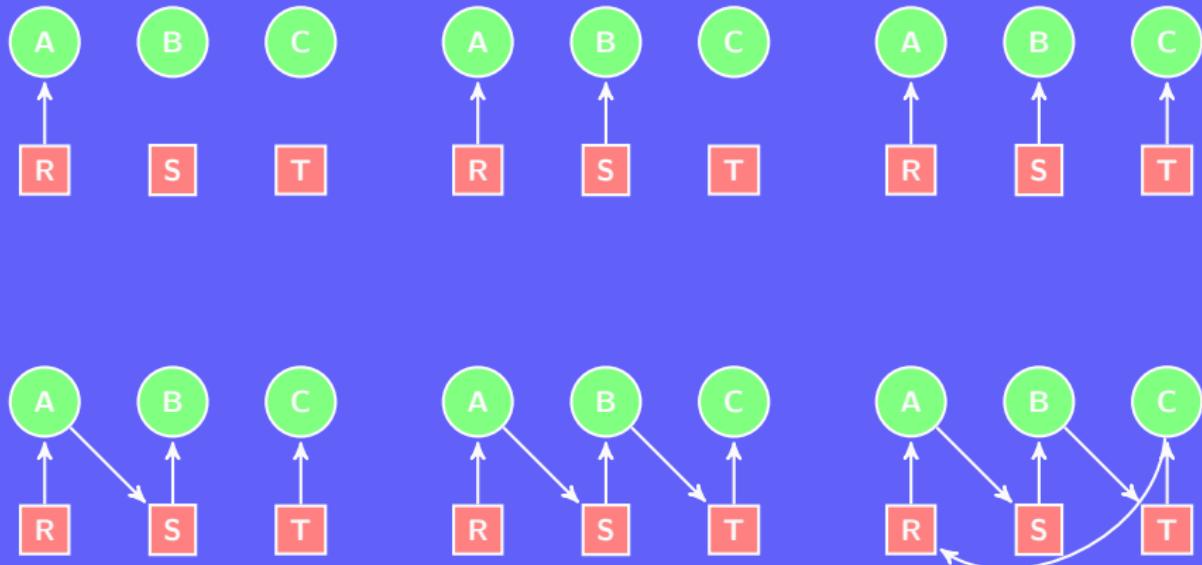
Resource  $R$   
requested by  $A$



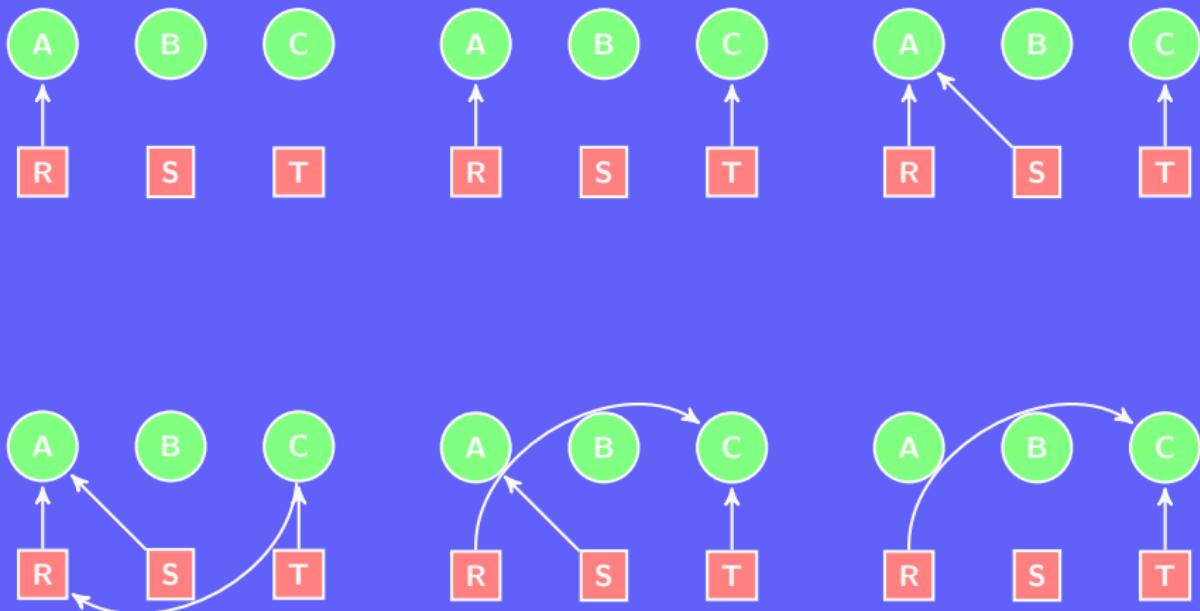
Deadlock



# Example – Deadlock



## Example – No deadlock



# Chapter organisation



**ALERT, ALERT**

**HIDE!!!!**

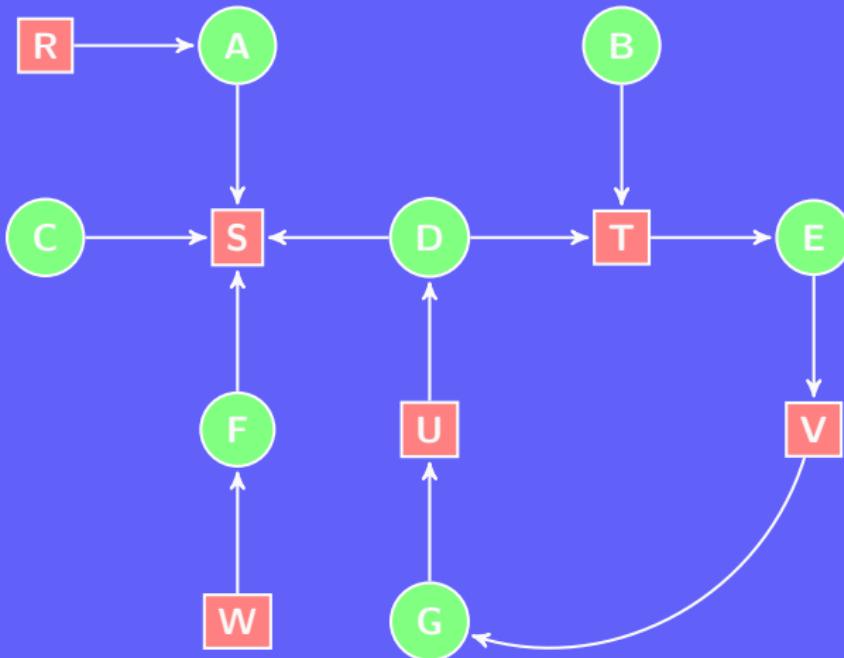
## Deadlock detection – Single resource

Example.

Is there any deadlock in the following system with seven processes ( $A-G$ ) and six resources ( $R-W$ )?

- Process  $A$  holds  $R$  and wants  $S$
- Process  $B$  wants  $T$
- Process  $C$  want  $S$
- Process  $D$  holds  $U$  and wants both  $S$  and  $T$
- Process  $E$  holds  $T$  and wants  $V$
- Process  $F$  holds  $W$  and wants  $S$
- Process  $G$  holds  $V$  and wants  $U$

## Deadlock detection – Single resource



## Deadlock detection – Multiple resources

Let  $E$  and  $A$  be two vectors representing the existing and the available resources respectively.  $C$  represents the current allocation matrix and  $R$  the request matrix.

Four resource types: Printer, Scanner, DVD burner and Plotter

$$E = ( \begin{array}{cccc} 4 & 2 & 3 & 1 \end{array} ) \quad A = ( \begin{array}{cccc} 2 & 1 & 0 & 0 \end{array} )$$

$$C = \left( \begin{array}{cccc} 0 & 0 & 1 & 0 \\ 2 & 0 & 1 & 1 \\ 0 & 1 & 1 & 0 \end{array} \right) \quad R = \left( \begin{array}{cccc} 2 & 0 & 0 & 1 \\ 1 & 0 & 1 & 0 \\ 2 & 1 & 0 & 0 \end{array} \right)$$

## Deadlock detection – Multiple resources

Let  $E$  and  $A$  be two vectors representing the existing and the available resources respectively.  $C$  represents the current allocation matrix and  $R$  the request matrix.

Four resource types: Printer, Scanner, DVD burner and Plotter

$$E = ( \begin{array}{cccc} 4 & 2 & 3 & 1 \end{array} ) \quad A = ( \begin{array}{cccc} 0 & 0 & 0 & 0 \end{array} )$$

$$C = \left( \begin{array}{cccc} 0 & 0 & 1 & 0 \\ 2 & 0 & 1 & 1 \\ 2 & 2 & 1 & 0 \end{array} \right) \quad R = \left( \begin{array}{cccc} 2 & 0 & 0 & 1 \\ 1 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 \end{array} \right)$$

## Deadlock detection – Multiple resources

Let  $E$  and  $A$  be two vectors representing the existing and the available resources respectively.  $C$  represents the current allocation matrix and  $R$  the request matrix.

Four resource types: Printer, Scanner, DVD burner and Plotter

$$E = ( \begin{array}{cccc} 4 & 2 & 3 & 1 \end{array} ) \quad A = ( \begin{array}{cccc} 2 & 2 & 1 & 0 \end{array} )$$

$$C = \left( \begin{array}{cccc} 0 & 0 & 1 & 0 \\ 2 & 0 & 1 & 1 \\ \cdot & \cdot & \cdot & \cdot \end{array} \right) \quad R = \left( \begin{array}{cccc} 2 & 0 & 0 & 1 \\ 1 & 0 & 1 & 0 \\ \cdot & \cdot & \cdot & \cdot \end{array} \right)$$

## Deadlock detection – Multiple resources

Let  $E$  and  $A$  be two vectors representing the existing and the available resources respectively.  $C$  represents the current allocation matrix and  $R$  the request matrix.

Four resource types: Printer, Scanner, DVD burner and Plotter

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

$$C = \begin{pmatrix} 0 & 0 & 1 & 0 \\ 3 & 0 & 2 & 1 \\ \cdot & \cdot & \cdot & \cdot \end{pmatrix} \quad R = \begin{pmatrix} 2 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 \\ \cdot & \cdot & \cdot & \cdot \end{pmatrix}$$

## Deadlock detection – Multiple resources

Let  $E$  and  $A$  be two vectors representing the existing and the available resources respectively.  $C$  represents the current allocation matrix and  $R$  the request matrix.

Four resource types: Printer, Scanner, DVD burner and Plotter

$$E = \begin{pmatrix} 4 & 2 & 3 & 1 \end{pmatrix} \quad A = \begin{pmatrix} 4 & 2 & 2 & 1 \end{pmatrix}$$

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

## Deadlock detection – Multiple resources

Let  $E$  and  $A$  be two vectors representing the existing and the available resources respectively.  $C$  represents the current allocation matrix and  $R$  the request matrix.

Four resource types: Printer, Scanner, DVD burner and Plotter

$$E = ( \begin{array}{cccc} 4 & 2 & 3 & 1 \end{array} ) \quad A = ( \begin{array}{cccc} 2 & 1 & 0 & 0 \end{array} )$$

$$C = \left( \begin{array}{cccc} 0 & 0 & 1 & 0 \\ 2 & 0 & 1 & 1 \\ 0 & 1 & 1 & 0 \end{array} \right) \quad R = \left( \begin{array}{cccc} 2 & 0 & 0 & 1 \\ 1 & 0 & 2 & 0 \\ 2 & 1 & 0 & 0 \end{array} \right)$$

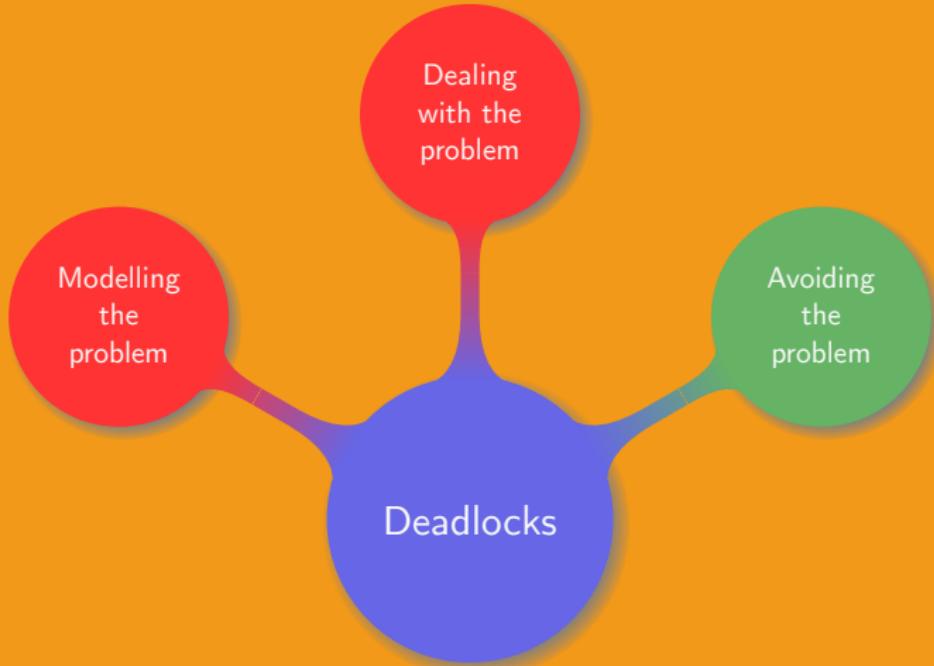
What if the second process requests two DVD burners?

# Recovering from a deadlock

Three main recovery strategies:

- Preemption:
  - Take a resource from another process
  - Might require manual intervention (e.g. collect papers from printer, pile them up and resume printing later)
- Rollback:
  - Set periodical checkpoints on processes
  - Save process state at the checkpoints
  - Restart process at a checkpoint (from before the deadlock)
- Killing:
  - Simplest strategy
  - Kill a process that uses resources related to the deadlock
  - Pick a process that can be re-run from the beginning

# Chapter organisation

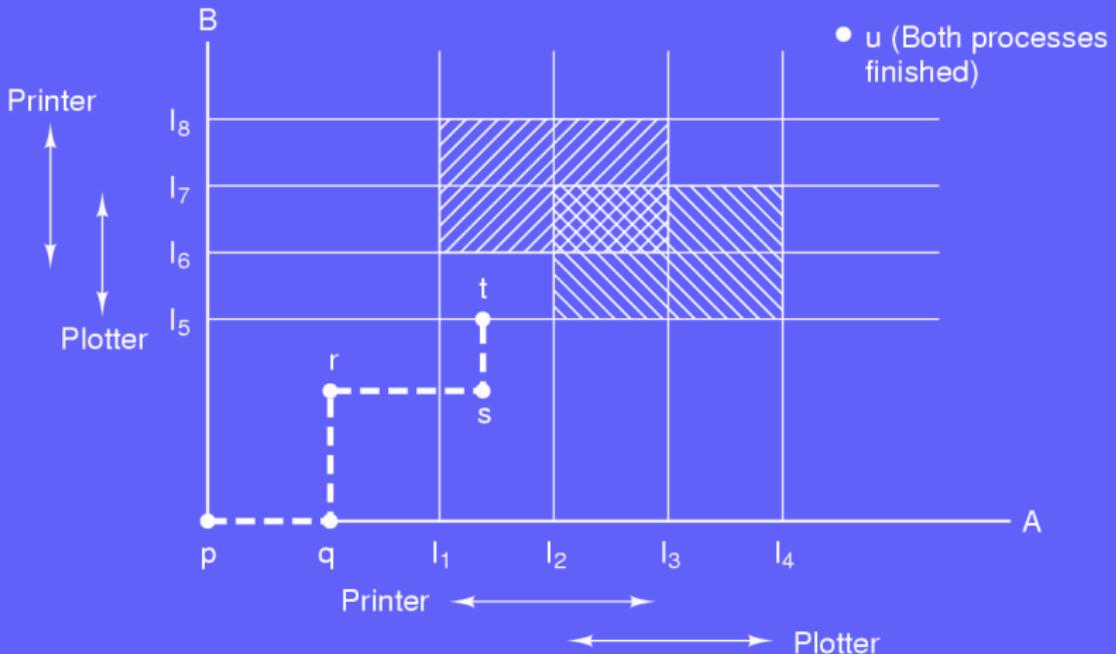


## Avoidance vs. prevention

Two main strategies against deadlocks:

- Avoidance: resources are assigned and released one at a time.  
Is there any algorithm that can perform the right choice to avoid deadlocks?
- Prevention: when is a deadlock occurring?  
If it is possible to describe the circumstances under which deadlocks occur it might be possible to prevent them

## Resources trajectories



Using the matrices  $E$ ,  $A$ ,  $C$  and  $R$ , define:

- Safe state: there exists a scheduling order allowing all processes to complete, even if they suddenly all request their maximum number of resources. The system can guarantee that all processes can finish
- Unsafe state: the ability of the system not to deadlock depends on the order the resources are allocated/deallocated. There is no way to predict whether or not all the processes will finish

Using the matrices  $E$ ,  $A$ ,  $C$  and  $R$ , define:

- Safe state: there exists a scheduling order allowing all processes to complete, even if they suddenly all request their maximum number of resources. The system can guarantee that all processes can finish
- Unsafe state: the ability of the system not to deadlock depends on the order the resources are allocated/deallocated. There is no way to predict whether or not all the processes will finish

Remark.

An unsafe state does not necessarily imply a deadlock; the system can still run for a while, or even complete all processes if some release their resources before requesting some more

# The banker's algorithm

General idea:

- Dijkstra (1965)
- Based on the detection algorithm
- Idea: avoid deadlocks by avoiding to run into an unsafe state
- Any request leading to an unsafe state is denied

# The banker's algorithm

General idea:

- Dijkstra (1965)
- Based on the detection algorithm
- Idea: avoid deadlocks by avoiding to run into an unsafe state
- Any request leading to an unsafe state is denied

Remark.

Mostly useless in practice since a process rarely knows the maximum resources it will need and the number of processes keeps varying. It also does not take into account hardware related issues (e.g. crashed printer)

## The banker's algorithm

Deciding whether a state is safe or not:

- ① Select a row in  $R$  whose resource request can be met. If no such row exists there is a possibility for a deadlock
- ② When the process terminates it releases all its resource, and they can be added to the vector  $A$
- ③ If all the processes terminate when repeating steps 1. and 2. then the state is safe. If step 1. fails at any stage (not all the processes being finished) then the state is unsafe and the request should be denied

# The banker's algorithm

Example.

Consider a system with 6 scanners, 3 plotters, 4 printers and 2 DVD drives:

$$E = \begin{pmatrix} 6 & 3 & 4 & 2 \end{pmatrix} \quad A = \begin{pmatrix} 1 & 1 & 2 & 0 \end{pmatrix}$$

$$C = \begin{pmatrix} 3 & 0 & 1 & 1 \\ 0 & 1 & 0 & 0 \\ 1 & 0 & 1 & 0 \\ 1 & 1 & 0 & 1 \\ 0 & 0 & 0 & 0 \end{pmatrix} \quad R = \begin{pmatrix} 2 & 1 & 0 & 0 \\ 0 & 1 & 1 & 1 \\ 3 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 3 & 1 & 1 & 0 \end{pmatrix}$$

# The banker's algorithm

Example.

Consider a system with 6 scanners, 3 plotters, 4 printers and 2 DVD drives:

$$E = \begin{pmatrix} 6 & 3 & 4 & 2 \end{pmatrix}$$

$$A = \begin{pmatrix} 2 & 2 & 2 & 1 \end{pmatrix}$$

$$C = \begin{pmatrix} 3 & 0 & 1 & 1 \\ 0 & 1 & 0 & 0 \\ 1 & 0 & 1 & 0 \\ \cdot & \cdot & \cdot & \cdot \\ 0 & 0 & 0 & 0 \end{pmatrix}$$

$$R = \begin{pmatrix} 2 & 1 & 0 & 0 \\ 0 & 1 & 1 & 1 \\ 3 & 1 & 0 & 0 \\ \cdot & \cdot & \cdot & \cdot \\ 3 & 1 & 1 & 0 \end{pmatrix}$$

# The banker's algorithm

Example.

Consider a system with 6 scanners, 3 plotters, 4 printers and 2 DVD drives:

$$E = \begin{pmatrix} 6 & 3 & 4 & 2 \end{pmatrix} \quad A = \begin{pmatrix} 5 & 2 & 3 & 2 \end{pmatrix}$$

$$C = \begin{pmatrix} \cdot & \cdot & \cdot & \cdot \\ 0 & 2 & 1 & 1 \\ 1 & 0 & 1 & 0 \\ \cdot & \cdot & \cdot & \cdot \\ 0 & 0 & 0 & 0 \end{pmatrix} \quad R = \begin{pmatrix} \cdot & \cdot & \cdot & \cdot \\ 0 & 1 & 1 & 1 \\ 3 & 1 & 0 & 0 \\ \cdot & \cdot & \cdot & \cdot \\ 3 & 1 & 1 & 0 \end{pmatrix}$$

# The banker's algorithm

Example.

Consider a system with 6 scanners, 3 plotters, 4 printers and 2 DVD drives:

$$E = \begin{pmatrix} 6 & 3 & 4 & 2 \end{pmatrix}$$

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

$$C = \begin{pmatrix} 3 & 0 & 1 & 1 \\ 0 & 1 & 0 & 0 \\ 1 & 0 & 1 & 0 \\ 1 & 1 & 0 & 1 \\ 0 & 0 & 0 & 0 \end{pmatrix}$$

$$R = \begin{pmatrix} 3 & 1 & 0 & 0 \\ 0 & 1 & 1 & 1 \\ 3 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 3 & 1 & 1 & 0 \end{pmatrix}$$

Assuming that the first process requests 3 more scanners instead of 2, is the state safe?

Resource deadlocks only occur under the following four conditions:

- Mutual exclusion: a resource can be assigned to at most one process at a time
- Hold and wait: a process currently holding some resources can request some more
- No preemption: resources must be released by the process itself, i.e. they cannot be taken away by another process
- Circular wait: there is a circular chain of processes each of them waiting from some resources held by another process

## Mutual exclusion condition

Preventing deadlocks:

- Not possible to remove it (e.g. two processes cannot print at the same time)
- Use daemon that handle specific output (e.g. printing daemon uses SPOOL)
- Deadlock can still happen (e.g. two processes fill up the SPOOL disk, without any of them being full)
- SPOOL cannot always be applied

**Conclusion:** not much can be done on this problem apart from carefully assigning resources

Preventing deadlocks:

- Require processes to claim all the resources at once
- Not realistic, a process does not always know what resources will be necessary
- What if computation last for hours, and then the result is burnt on a DVD?
- Resources are not handle in an optimal way
- Alternative strategy: process has to release its resources before getting new ones

**Conclusion:** possible, but far from optimal

## No preemption condition

Preventing deadlocks:

- Issue inherent to the hardware
- Often impossible to do anything (e.g. stop burning a DVD and resume later)
- Might require human intervention (e.g. stop a printer job, get the already printed pages, print another job and resume the first one)

**Conclusion:** not viable to break this condition

Preventing deadlocks:

- Order the resources
- Processes have to request resources in increasing order
- A process can only request a lower resource if it has released all the larger ones
- Is there an order satisfying everybody?

**Conclusion:** this is the best solution but it not always possible to use it in practice

Deadlocks are not necessarily related to hardware resources:

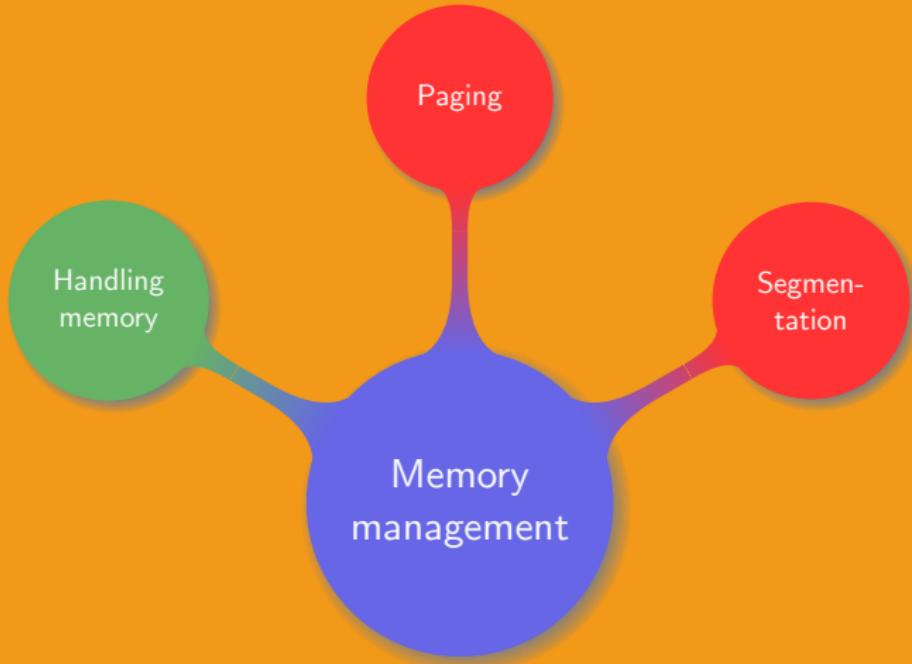
- Database records: two-phase locking solution; lock all the records in phase one; if one fails release all the locks and retry later, otherwise proceed with phase two and manipulate the records
- Communication deadlocks: mutex can lead to deadlocks; no hardware resource involved; could be due to the loss of a message
- Livelock: lack of resources, process can not keep going so seat in tight loop and keeps trying not knowing it is hopeless
- Starvation: one long process delayed to let shorter ones run; might never run...



## 6. Memory management

---

# Chapter organisation



Access time

1 ns

Registers

Capacity

&lt; 1 KB

2 ns

Cache

4 MB

10 ns

Main memory

1–8 GB

10 ms

Magnetic disk

200–1000 GB

10 s

Magnetic tape

400–800 GB

Problems related to memory:

- From expensive to cheap, fast to slow
- Job of the OS to handle the memory
- How to model the hierarchy?
- How to manage this abstraction?

Efficiently manage memory:

- Keep track of which part of the memory is used
- Allocate memory to processes when required
- Deallocate memory at the end of a process

Remark.

It is the job of the hardware to manage the lowest levels of cache memory

No memory abstraction:

- Program sees the actual physical memory
- Programmer could access the whole memory
- Limitations when running more than one program:
  - Have to copy the whole content of the memory into a file when switching program
  - No more than one program in the memory at once
  - More than one program possible with special hardware

No abstraction leads to two main problems:

- Protection: prevent program from accessing other's memory
- Relocation: rewrite address to allocate personal memory

No abstraction leads to two main problems:

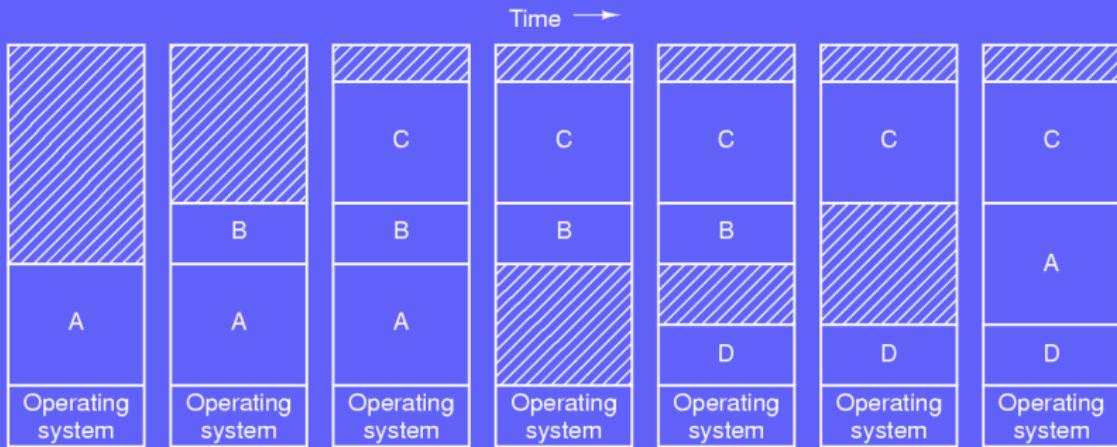
- Protection: prevent program from accessing other's memory
- Relocation: rewrite address to allocate personal memory

A solution is to set an address space:

- Set of addresses that a process can use
- Independent from other processes' memory

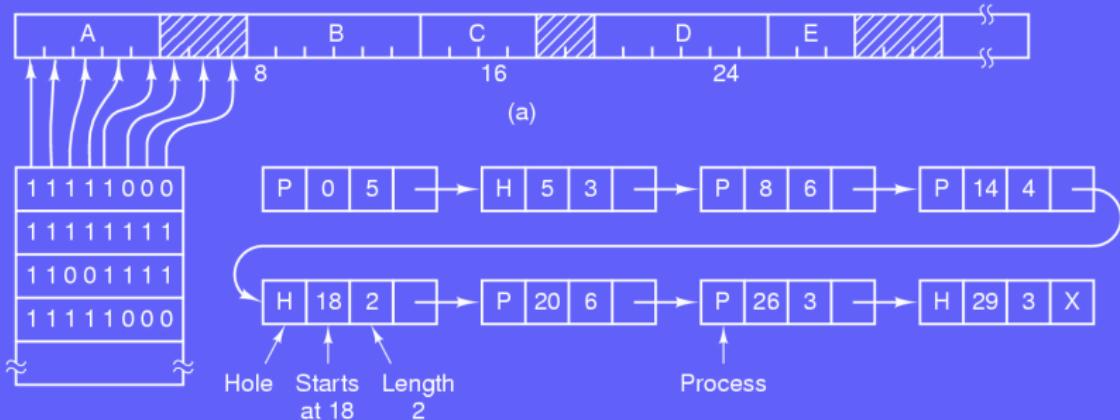
# Memory size limitation

When booting many processes are started, then more are run by the user: much memory is required, more than available in RAM.



- Processes are swapped in (out) from (to) the disk.
- OS has to manage dynamically assigned memory

# Bitmap and linked lists



Simple idea:

- Define some base size for an area  $s$
- Split up the whole memory into  $n$  chunks of size  $s$
- Keep track of the memory used in a bitmap or linked list

## Allocating memory

Assuming memory manager knows how much memory should be assigned, different strategies can be used:

- First fit: search for a hole big enough and use the first found
- Best fit: search whole list and use smallest, big enough hole
- Quick fit: maintain lists of common requested memory sizes, use the best one

Assuming memory manager knows how much memory should be assigned, different strategies can be used:

- First fit: search for a hole big enough and use the first found
- Best fit: search whole list and use smallest, big enough hole
- Quick fit: maintain lists of common requested memory sizes, use the best one

Characteristics:

- Speed: quick fit > first fit > best fit
- Locally optimal: quick fit = best fit > first fit
- Globally optimal: first fit > quick fit = best fit

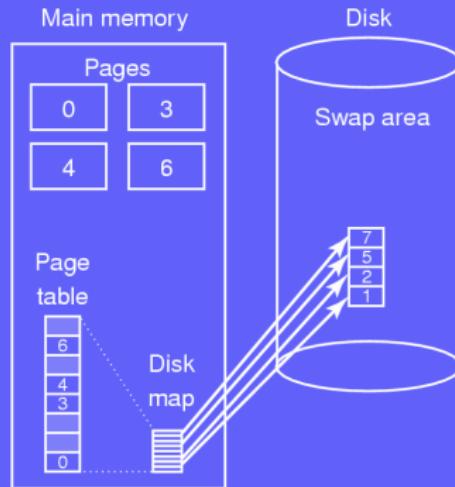
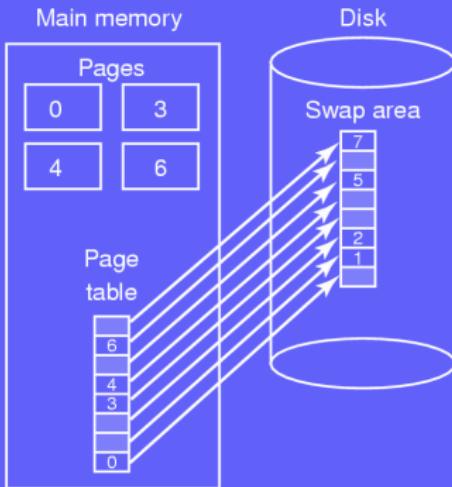
Virtual memory:

- Generalisation of the base and limit registers
- Each process has its own address space
- The Address space is split into chunks called **pages**
- Each page corresponds to a range of addresses
- Pages are mapped onto physical memory
- Pages can be on different medium (e.g. RAM and swap)

### Swap partition principles:

- Simple way to allocate page space on the disk
- OS boots, swap is empty and defined by two numbers: its origin and its size
- When a process is started, a chunk of the partition equal to the process' size is reserved
- The new “origin” is computed
- When a process terminates its swap area is freed
- The swap is handled as a list of free chunks
- When a process starts, its swap area is initialised

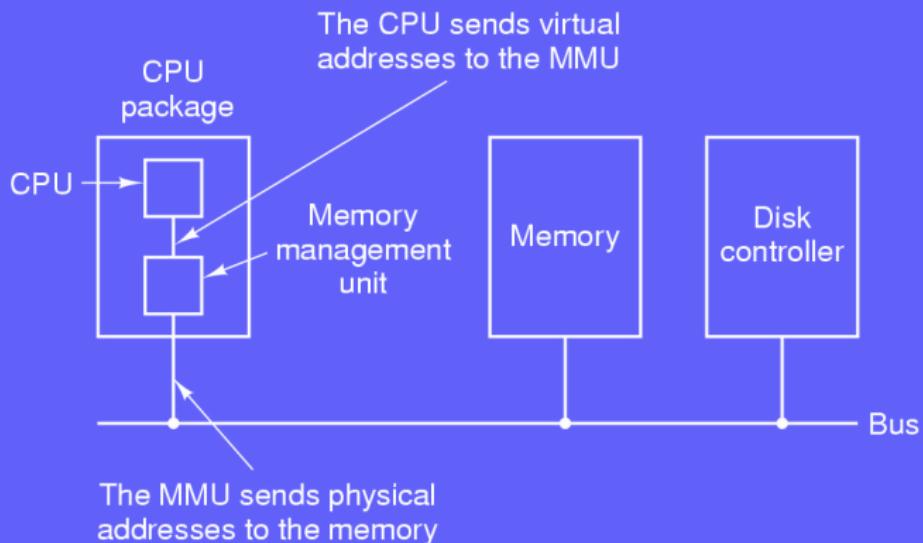
# Initialising the swap area



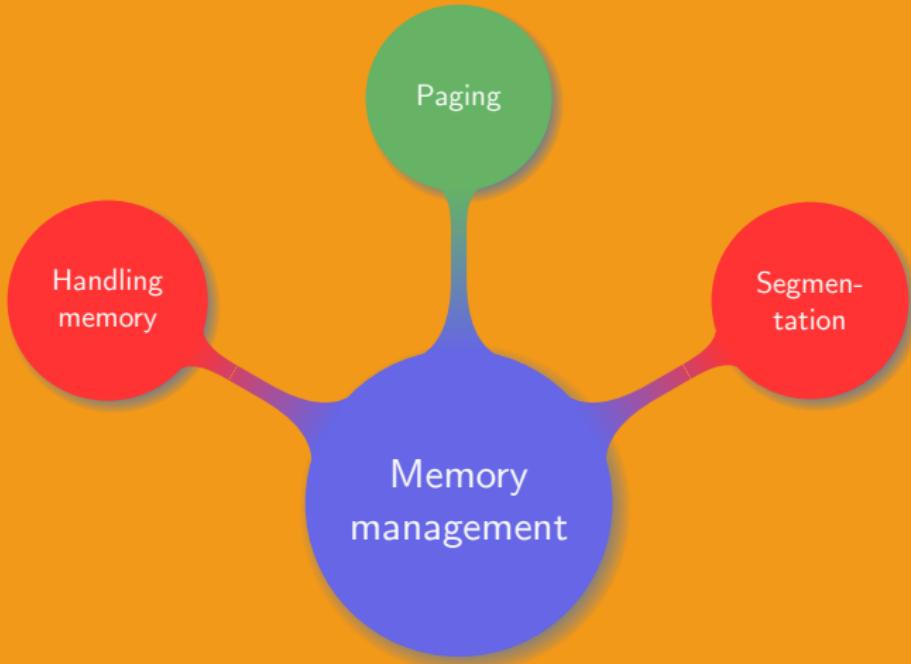
Two main strategies:

- Copy the whole process image to the swap area
- Allocate swap disk space on the fly

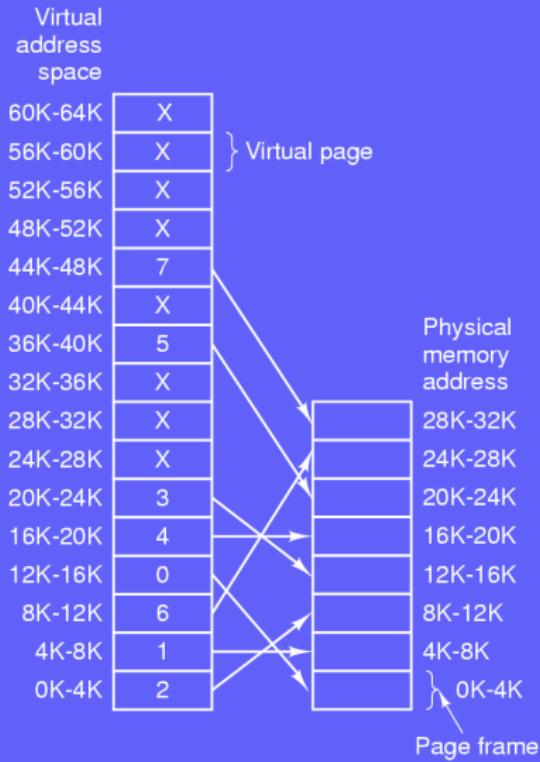
# Memory Management Unit



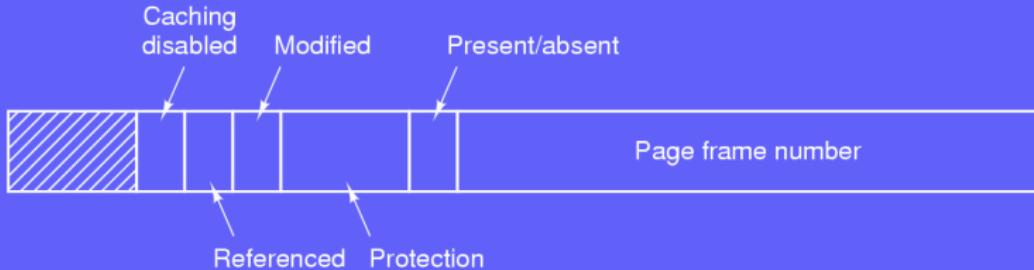
# Chapter organisation



# Virtual page and page frame



- Virtual address space divided into fixed-size units: pages
- Pages and page frames are usually of same size
- MMU maps virtual addresses to physical addresses
- MMU causes the CPU to trap on a page fault
- OS copies content of a little used page onto the disk
- Page frame loaded onto newly freed page



Structure of a page entry:

- Present/absent: 1/0; missing causes a page fault
- Protection: 1 to 3 bits: reading/writing/executing
- Modified: 1/0 = dirty/clean; page was modified and needs to be updated on the disk
- Referenced: bit used to keep track of most used pages; useful in case of a page fault
- Caching: important for pages that map to registers; do not want to use old copy so set caching to 0

Two main issue must be solved in a paging system:

- Mapping must be done efficiently
- A large virtual address space implies a large page table

Two main issue must be solved in a paging system:

- Mapping must be done efficiently
- A large virtual address space implies a large page table

Translation Lookaside Buffer (TLB):

- Hardware solution implemented inside the MMU
- Keeps track of few most used pages
- Features the same fields as for page table entries including the virtual page number and page frame

On a page fault the following operations are performed:

- Choose a page to remove from the memory
- If the page was modified while in the memory it needs to be rewritten on the disk; otherwise nothing needs to be done
- Overwrite the page with the new memory content

**Problem:** how to optimize the selection of the page to be evicted?

## Page replacement – Optimal solution'

Determining which page to remove when a page fault occurs:

- Label and order all the pages in memory
- The page with lower label is used first
- The page with larger label is swapped out of the memory

## Page replacement – Optimal solution'

Determining which page to remove when a page fault occurs:

- Label and order all the pages in memory
- The page with lower label is used first
- The page with larger label is swapped out of the memory

**Problem:** can the information be known ahead of time?

## Page replacement – LRU

**Basic idea:** recently heavily used pages will be used again and pages that haven't been accessed recently won't be used soon

Hardware solution, for  $n \times n$  page frames:

- Initialise a binary  $n \times n$  matrix to 0
- When frame  $k$  is used set row  $k$  to 1 and column  $k$  to 0
- Replace the page with the smallest value

# Page replacement – LRU

**Basic idea:** recently heavily used pages will be used again and pages that haven't been accessed recently won't be used soon

Hardware solution, for  $n \times n$  page frames:

- Initialise a binary  $n \times n$  matrix to 0
- When frame  $k$  is used set row  $k$  to 1 and column  $k$  to 0
- Replace the page with the smallest value

$$\begin{bmatrix} 0 & 1 & 1 & 1 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix} \quad \begin{bmatrix} 0 & 0 & 1 & 1 \\ 1 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix} \quad \begin{bmatrix} 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 1 \\ 1 & 1 & 0 & 1 \\ 0 & 0 & 0 & 0 \end{bmatrix} \quad \begin{bmatrix} 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 \\ 1 & 1 & 1 & 0 \end{bmatrix}$$
$$\begin{bmatrix} 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ 1 & 1 & 0 & 1 \\ 1 & 1 & 0 & 0 \end{bmatrix} \quad \begin{bmatrix} 0 & 0 & 0 & 0 \\ 1 & 0 & 1 & 1 \\ 1 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 \end{bmatrix} \quad \begin{bmatrix} 0 & 1 & 1 & 1 \\ 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 \end{bmatrix} \quad \begin{bmatrix} 0 & 1 & 1 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 0 \end{bmatrix}$$

### $M$ and $R$ bits contained in the page table

- Software solutions require some hardware information
- OS needs to collect information on page usage
- Process starts: none of its page table entries are in memory
- Page is referenced: set the  $R$  bit
- Page is written: set the  $M$  bit
- $M$  and  $R$  must be updated on every memory reference

**Goal:** simulate the LRU in software

- For each page initialise an  $n$ -bit software counter to 0
- At each clock interrupt the OS scans all the pages in memory
- Shift all the counters by 1 bit to the right
- Add  $2^{n-1} \cdot R$  to the counter

Example:  $n = 8$ , 4 pages over 4 clock interrupts

# Page replacement – Aging

**Goal:** simulate the LRU in software

- For each page initialise an  $n$ -bit software counter to 0
- At each clock interrupt the OS scans all the pages in memory
- Shift all the counters by 1 bit to the right
- Add  $2^{n-1} \cdot R$  to the counter

Example:  $n = 8$ , 4 pages over 4 clock interrupts

$t$	$t_0$	$t_1$	$t_2$	$t_3$
$R$	[ 1 0 1 0 ]	[ 1 1 0 0 ]	[ 1 1 0 1 ]	[ 1 0 0 0 ]
$p1$	10000000	11000000	11100000	11110000
$p2$	00000000	10000000	11000000	01100000
$p3$	10000000	01000000	00100000	00010000
$p4$	00000000	00000000	10000000	01000000

# Page replacement – Aging

**Goal:** simulate the LRU in software

- For each page initialise an  $n$ -bit software counter to 0
- At each clock interrupt the OS scans all the pages in memory
- Shift all the counters by 1 bit to the right
- Add  $2^{n-1} \cdot R$  to the counter

Example:  $n = 8$ , 4 pages over 4 clock interrupts

$t$	$t_0$	$t_1$	$t_2$	$t_3$
$R$	[ 1 0 1 0 ]	[ 1 1 0 0 ]	[ 1 1 0 1 ]	[ 1 0 0 0 ]
$p1$	10000000	11000000	11100000	11110000
$p2$	00000000	10000000	11000000	01100000
$p3$	10000000	01000000	00100000	00010000
$p4$	00000000	00000000	10000000	01000000

**Note:** counter has a finite number of bits, a state is lost after  $n \cdot t$ .

Basic notions related to paging:

- Demand paging: pages are loaded on demand
- Locality reference: during an execution phase a process only access a small fraction of all its pages
- Working set: set of pages currently used by a process
- Thrashing: process causes many page fault due to a lack of memory
- Pre-paging: pages loaded in memory before letting process run
- Current virtual time: amount of time during which a process has used the CPU
- $\tau$ : age of the working set

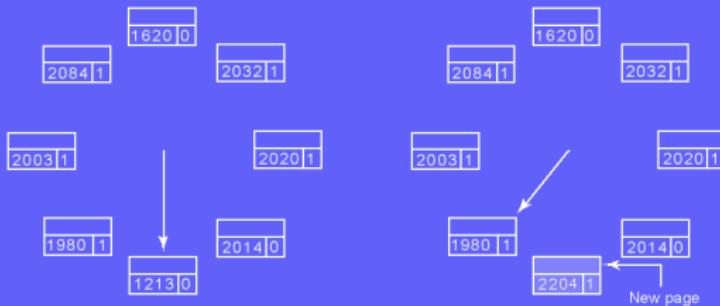
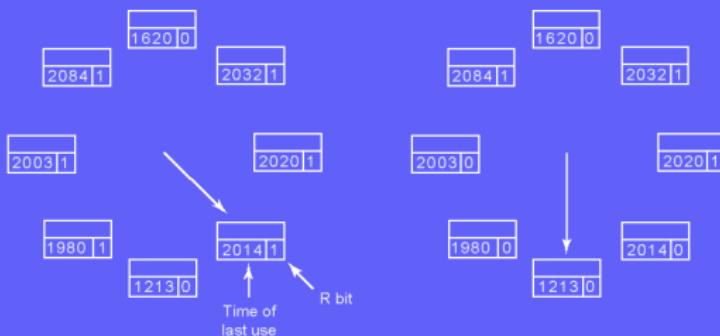
## Page replacement – WSClock

**Setup:** a circular list of page frames, pages have been inserted, each entry composed of time of last use,  $R$  and  $M$  bits

- On a page fault examine the pages the hand points to
- If  $R = 1$ , bad candidate: set  $R$  to 0 and advance hand
- If  $R = 0$ ,  $\text{age} > \tau$ 
  - If page is clean, then use page frame
  - Otherwise schedule write, move the hand repeat algorithm
- If hand has completed one cycle
  - If at least one write was scheduled, keep the hand moving until a write is completed and a page frame becomes available
  - Otherwise (i.e. all the pages are in the working set) take any page ensure it is clean (or write it to the disk) and use its corresponding page frame

# Page replacement – WSClock

2204 | Current virtual time



## Local vs. global allocation

Onto which set should the page replacement algorithm be applied:

- Local: within the process  $\Rightarrow$  allocate a portion of the whole memory to a process and only use this portion; number of page frames for a process remains constant
- Global: within the whole memory  $\Rightarrow$  dynamically allocate page frames to a process; number of page frames for a process varies over time

## Local vs. global allocation

Onto which set should the page replacement algorithm be applied:

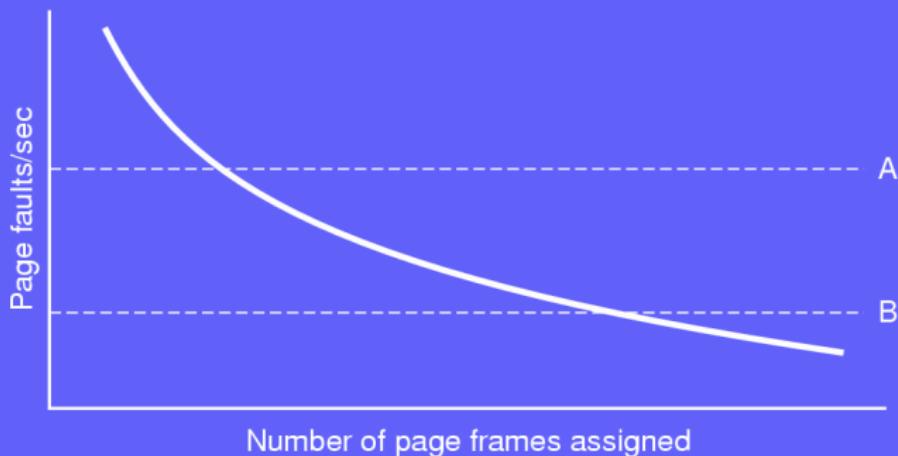
- Local: within the process  $\Rightarrow$  allocate a portion of the whole memory to a process and only use this portion; number of page frames for a process remains constant
- Global: within the whole memory  $\Rightarrow$  dynamically allocate page frames to a process; number of page frames for a process varies over time

Which approach is best?

# Page fault frequency

Adjusting the number of pages:

- Start process with a number of pages proportional to its size
- Adjust page allocation based on the page fault frequency
  - Count number of page fault per second
  - If larger than  $A$  then allocate more page frames
  - If below  $B$  then free some page frames



Finding optimal page size given a page frame size:

- In average half of the last page is used (internal fragmentation)
- The smaller the page size, the larger the page table

Finding optimal page size given a page frame size:

- In average half of the last page is used (internal fragmentation)
- The smaller the page size, the larger the page table

Page size  $p$ , process size  $s$  bytes, average size for page entry  $e$  and overhead  $o$ :

$$o = \frac{se}{p} + \frac{p}{2}$$

Differentiate with respect to  $p$  and equate to 0:

$$\frac{1}{2} = \frac{se}{p^2}$$

Optimal page size:  $p = \sqrt{2se}$

Common page frame sizes: 4KB or 8KB

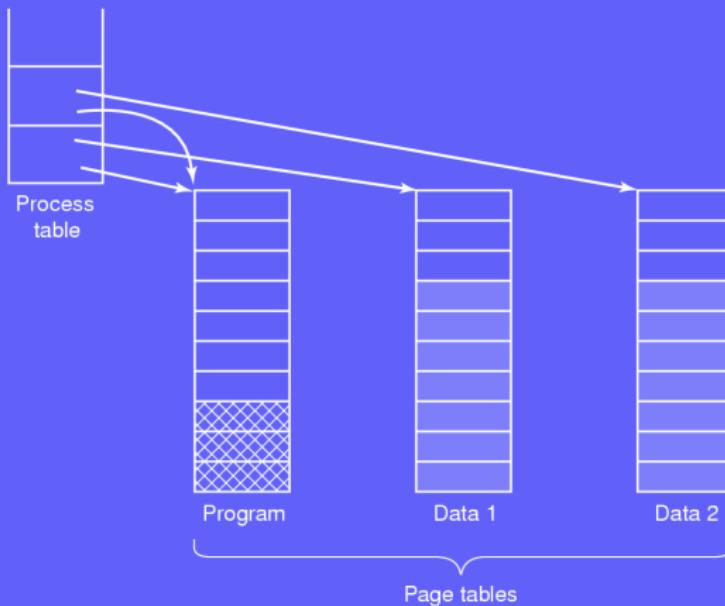
When a same program is run by different users at the same time, then sharing pages decreases memory use:

- Pages containing the program can be shared
- Personal data should not be shared

# Page sharing

When a same program is run by different users at the same time, then sharing pages decreases memory use:

- Pages containing the program can be shared
- Personal data should not be shared



Several basic problems arise:

- On a process switch do not remove all pages if required by another process: would generate many page fault
- When a process terminates do not free all the memory if it is required by another process: would generate a crash
- How to share data in read-write mode?

# When to use paging?

OS involved in paging related work on four occasions (1-2):

- **Process creation:** (i) determine process size; (ii) create process' page table (allocate and initialise memory); (iii) initialise swap area; (iv) store information related to the swap area and page table in the process table
- **Process execution:** (i) MMU reset for the new process; (ii) flush the TLB; (iii) make the new process' page table the current one

# When to use paging?

OS involved in paging related work on four occasions (3-4):

- **Page fault:** (i) read hardware register to determine origin of page fault; (ii) compute which page is needed; (iii) locate the page on the disk; (iv) find an available page frame and replace its content; (v) read the new page frame; (vi) rewind to the faulting instruction and re-execute it
- **Process termination:** (i) release page table, pages and disk space; (ii) beware of any page that could be shared among several processes

Process on a page fault:

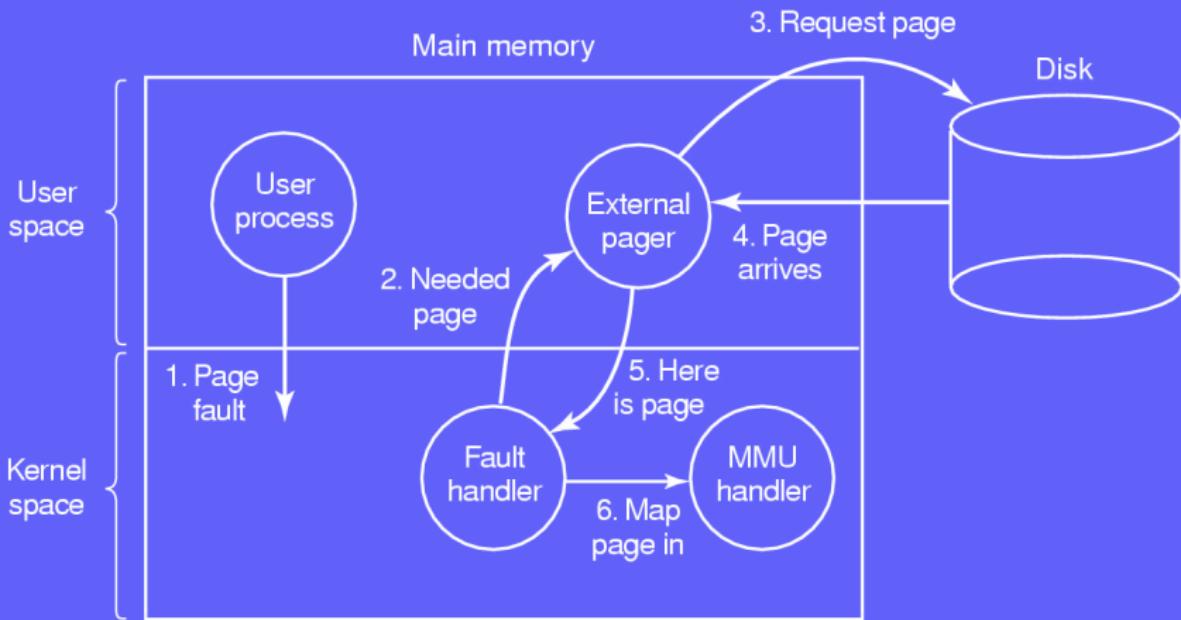
- ① Trap to the kernel is issued; program counter is saved in the stack; state of current instruction saved on some specific registers
- ② Assembly code routine started: save general registers and other volatile information
- ③ OS search which page is requested
- ④ Once the page is found: check if the address is valid and if process is allowed to access the page. If not kill the process; otherwise find a free page frame
- ⑤ If selected frame is dirty: have a context switch (faulting process is suspended) until disk transfer has completed. The page frame is marked as used such as not to be used by another process

- ⑥ When page frame is clean: schedule disk write to swap in the page. In the meantime the faulting process is suspended and other processes can be scheduled
- ⑦ When receiving a disk interrupt to indicate copy is done: page table is updated and frame is marked as being in a normal state
- ⑧ Rewind program to the faulting instruction, program counter reset to this value
- ⑨ Faulting process scheduled
- ⑩ Assembly code routine starts: reload registers and other volatile information
- ⑪ Process execution can continue

# Policy and mechanism

Example:

- Low level MMU handler: architecture dependent
- Page fault handler: kernel space
- External handler: user space



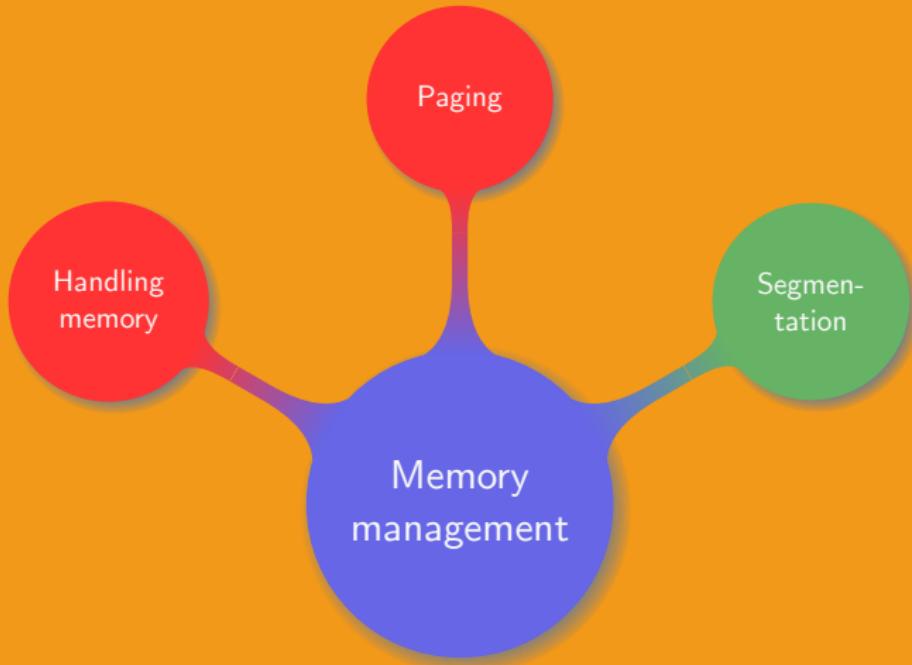
Where should the page replacement algorithm go:

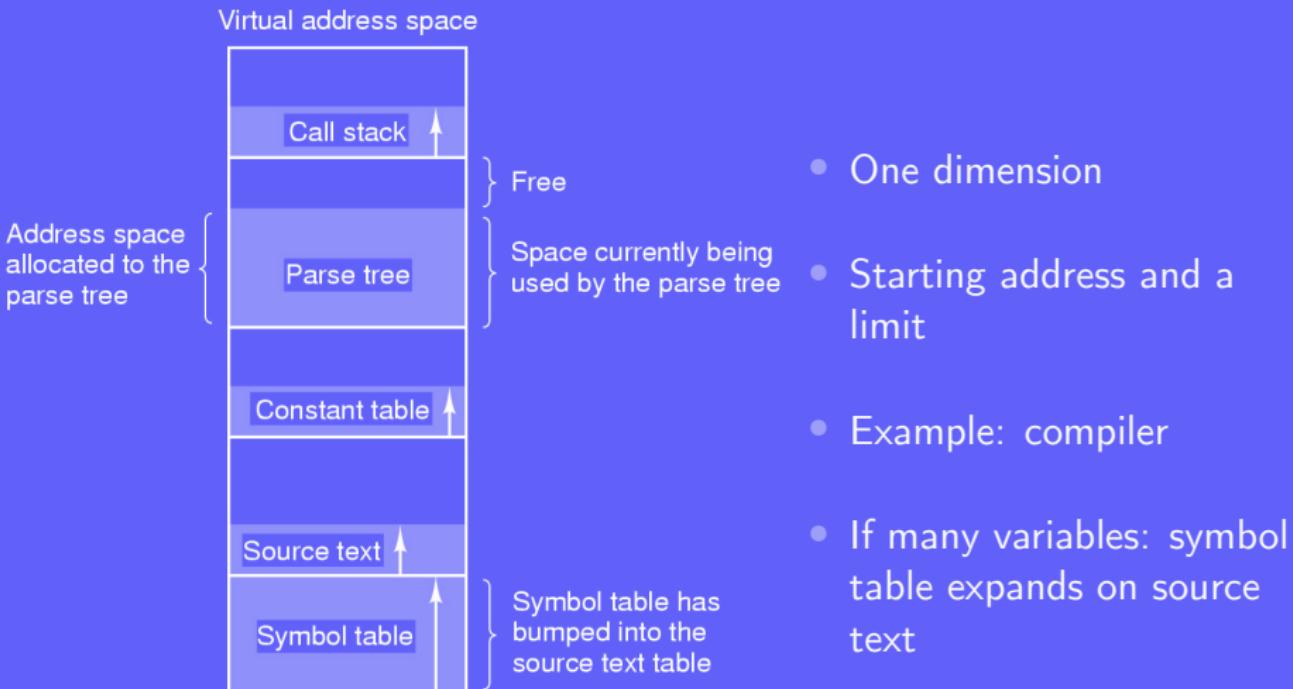
User space

Kernel space

- Use some mechanism to access the  $R$  and  $M$  bits
  - Clean solution
  - Overhead resulting from crossing user-kernel boundary several times
  - Modular code, better flexibility
- 
- Fault handler sends all information to external pager (which page was selected for removal)
  - External pager writes the page to the disk
  - No overhead, faster

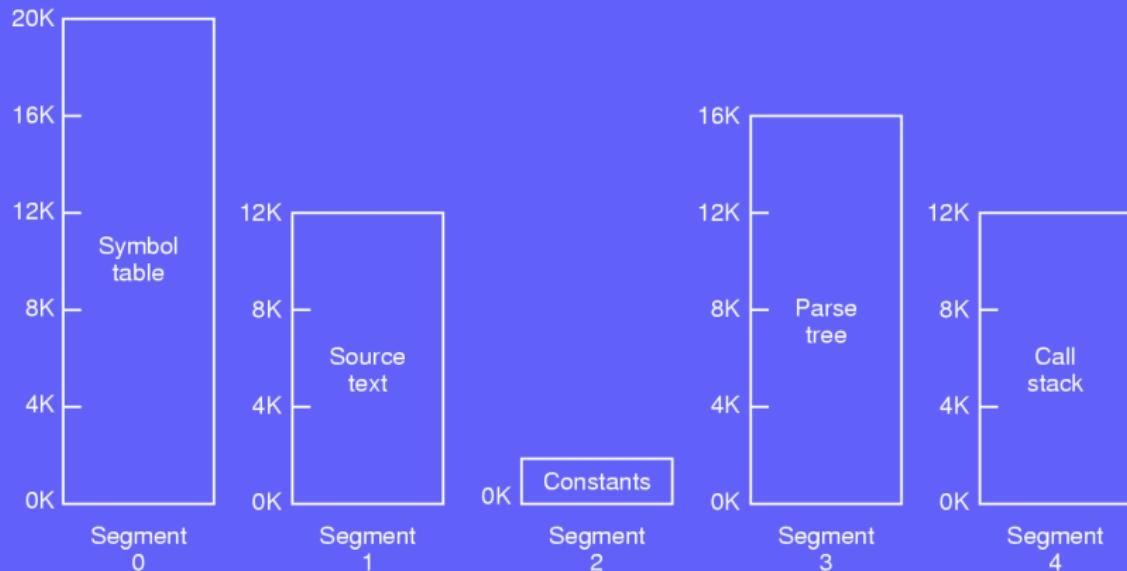
# Chapter organisation





# Segmentation

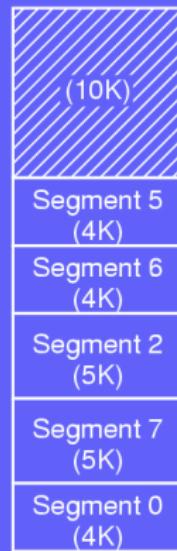
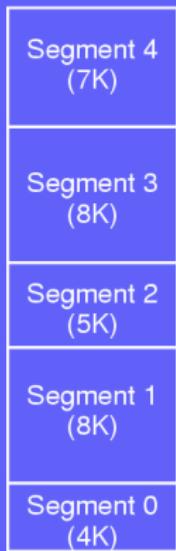
209



Handling segmentation in the OS:

- Each segment has a number and an offset
- Segment table: contains the starting physical address of each segment, the *base*, together with its size, the *limit*
- Segment table base register: points to the segment table
- Segment table length register: number of segments used in a program

# External fragmentation and compaction



# Paging vs. segmentation

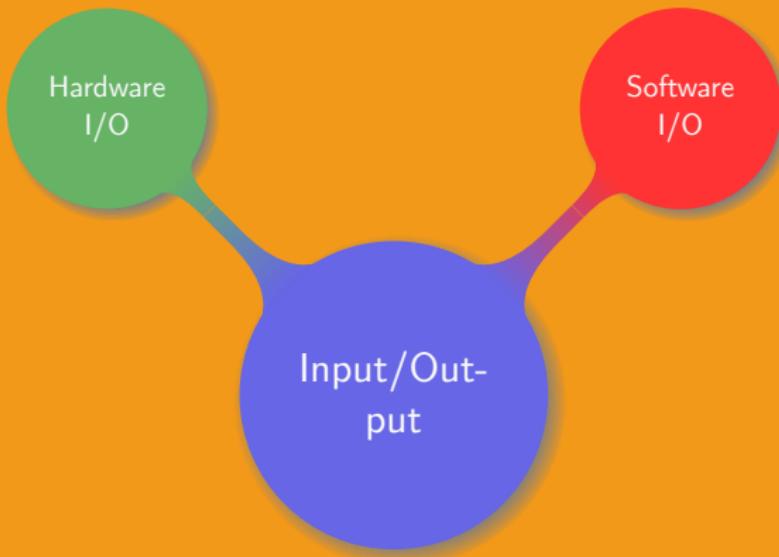
Considerations	Paging	Segmentation
Number of linear address space	1	many
Limited by the size of the physical memory	no	no
Possible to separate and protect data and procedures	no	yes
Sharing procedures between users or programs	complex	easy



## 7. Input/Output

---

# Chapter organisation



The OS controls all the I/O:

- Issue commands to the devices
- Catch interrupts
- Handle errors

*The OS provides a simple way to use interfaces for the rest the system*

Two main categories:

- Block devices:
  - Stores information in blocks of fixed size
  - Can directly access any block independently of other ones
- Character devices:
  - Delivers or accepts a stream of characters
  - Not addressable, no block structure

Two main categories:

- Block devices:
  - Stores information in blocks of fixed size
  - Can directly access any block independently of other ones
- Character devices:
  - Delivers or accepts a stream of characters
  - Not addressable, no block structure
- Others: ex. clock: cause interrupts at some given interval

Most devices have two parts: *mechanical*, the device itself and *electronic* that allows the communication with the device.

The electronic part is called the device controller:

- Allows to handle mechanical part in an easier way
- Performs error corrections for instance in the case of a disk
- Prepares and assemble blocks of bits in a buffer
- The blocks are then copied into the memory

# Memory-mapped I/O

The CPU communicates with the device using *control registers*:

- OS writes on registers to: send/accept data, switch device on/off
- OS reads from registers to: know device's state

Modern approach:

- Map the buffer to a memory address
- Map each register to a unique memory address or I/O port

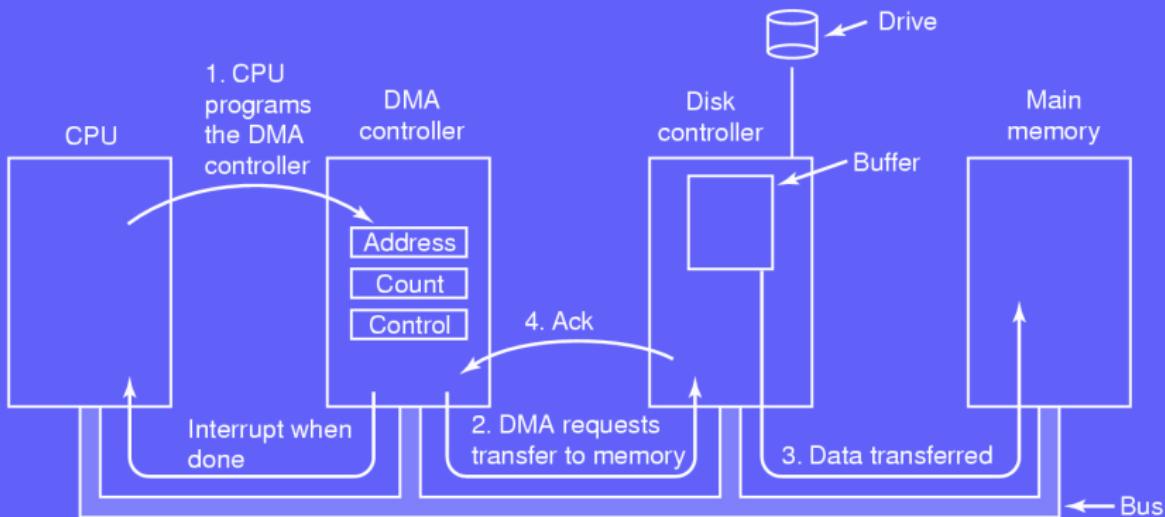
# Memory-mapped I/O – Pros and cons

## Strengths:

- Access memory not hardware → no need for assembly
- No special protection required → control register address space not included in the virtual address space
- Flexible → a specific/privileged user can be given access to a particular device
- Different drivers in different address spaces → reduces kernel size + no interference between drivers

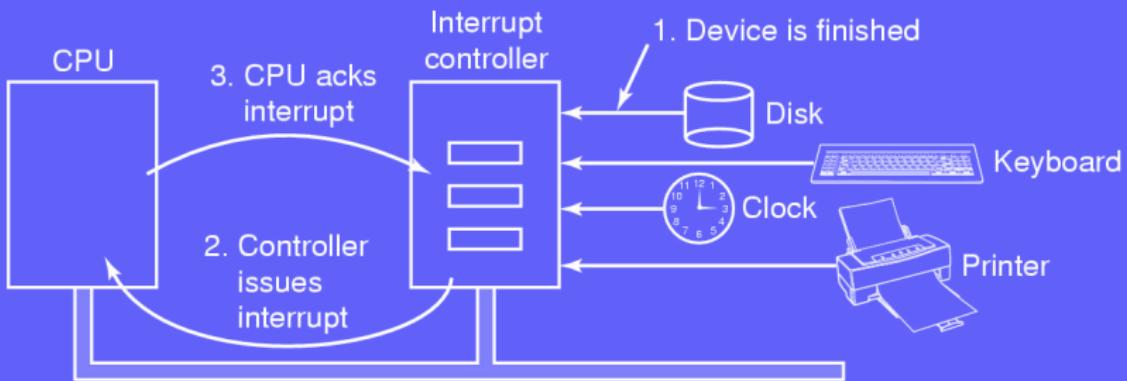
Weakness: memory words are cached → what if the content of control register is cached?

# Direct Memory Access



# Interrupts

222



## Precise vs. imprecise interrupts

**Initial setup:** on an interrupt push Program Counter (PC) and PSW on the stack, handle interrupt, retrieve program counter and PSW and resume process.

**New setup:** pipelined or superscalar CPU. What consequences?

# Precise vs. imprecise interrupts

**Initial setup:** on an interrupt push Program Counter (PC) and PSW on the stack, handle interrupt, retrieve program counter and PSW and resume process.

**New setup:** pipelined or superscalar CPU. What consequences?

**Precise interrupt:**

- PC saved in a known place
- All instructions before PC have been executed
- No instruction after the one pointed by PC has been executed
- Execution state of the instruction pointed by PC is known

An interrupt which is not precise is called **imprecise interrupt**.

## Dealing with imprecise interrupts

Difficult to figure out what happened and what has to happen:

- Instructions near PC are in different stages of completion
- General state can be recovered if given many details on the internal state
- Code to resume process is complex
- Many details implies much memory used

Conclusion: slow interrupts

## Dealing with imprecise interrupts

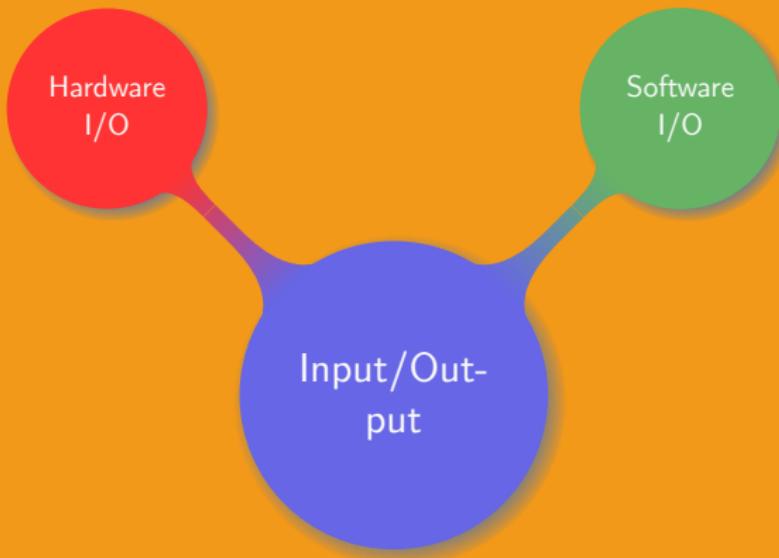
Difficult to figure out what happened and what has to happen:

- Instructions near PC are in different stages of completion
- General state can be recovered if given many details on the internal state
- Code to resume process is complex
- Many details implies much memory used

Conclusion: slow interrupts

Possible to get precise interrupts at the cost of complex interrupt logic within the CPU. CPU area used to get precise interrupts is wasted.

# Chapter organisation



Main goals on the design of I/O software:

- Device independence: whatever the support, files are handled the same way
- Uniform naming: devices organised by type with a name composed of string and number
- Error handling: fix error at lowest level possible
- Synchronous vs. asynchronous: OS decides if interrupt driven operations look blocking to user programs
- Buffer: need some temporary space to store data
- Shared vs. dedicated devices: more than/only one user can use a device at a time

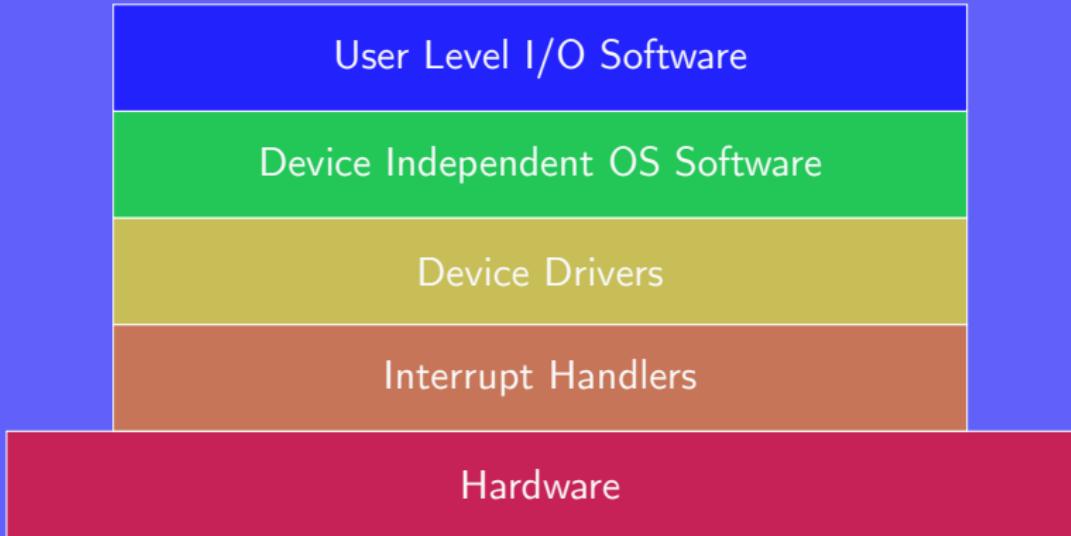
# Communications strategies

Three communications strategies:

# Communications strategies

Three communications strategies:

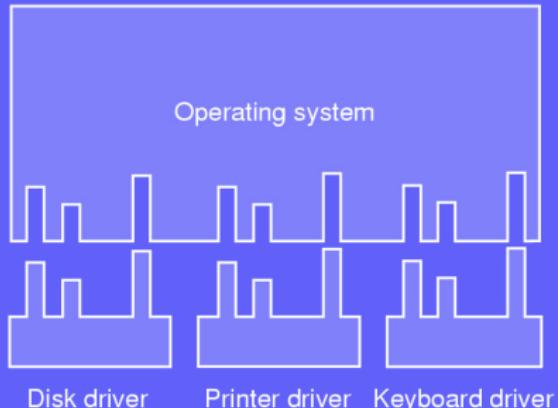
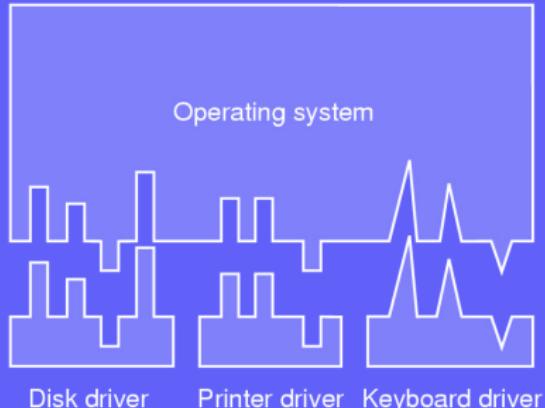
- ① **Programmed I/O:** copy data into kernel space, then fill up device register and wait in tight loop until device register is empty, fill it up...
- ② **Interrupt I/O:** copy data into kernel space, then fill up device register. The current process is blocked so the scheduler is called to let another process run. When the register is empty an interrupt is sent, the new current process is stopped and register is filled up...
- ③ **DMA:** similar to programmed I/O, but DMA does all the work.



Actions to performs on an interrupt:

- ① Save registers
- ② Setup a context for handling the interrupt
- ③ Setup a stack
- ④ Acknowledge interrupt controller + re-enable interrupts
- ⑤ Load registers
- ⑥ Extract information from interrupting device's controller
- ⑦ Choose a process to run next
- ⑧ Setup MMU and TLB for next process
- ⑨ Load new process registers
- ⑩ Run new process

# Device drivers interface



- Same class of device has a common basic set of functionalities
- OS defines which functionalities should be implemented
- Use a table of function pointers to interface device driver with the rest of the OS
- Uniform naming at user level

Basic functions of a driver:

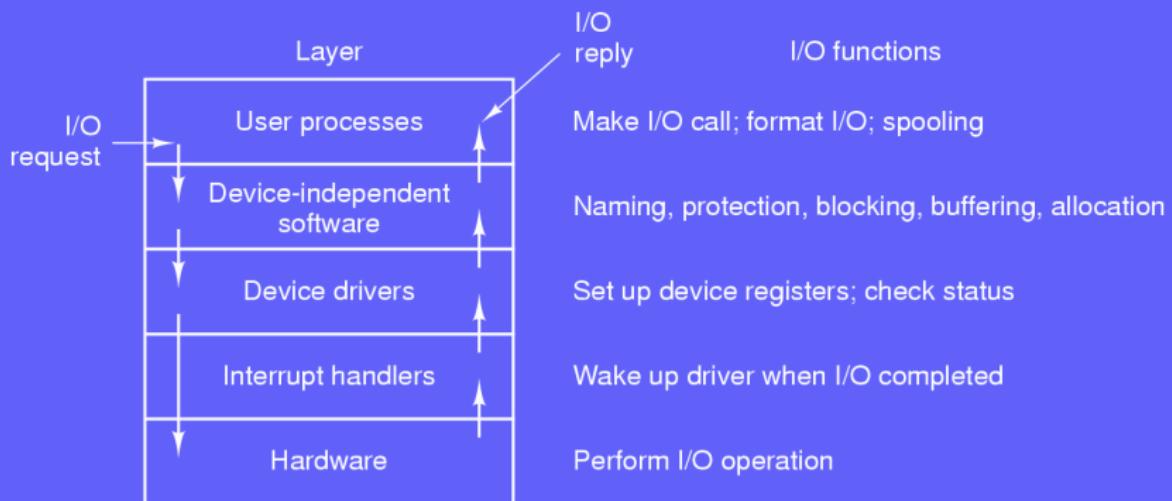
- Initialization
- Accept generic requests (e.g. read/write)
- Log events
- Retrieve device status
- Handle device specific errors
- Specific actions depending on the device

Device driver should react nicely even under special circumstances

General remarks on drivers:

- Location: user or kernel space
- Drivers can be compiled in kernel
- Drivers can be dynamically loaded at runtime
- Drivers can call certain kernel procedures (manage MMU, timers, DMA etc...)
- I/O errors framework is device independent
- Clean and generic API such that it is easy to write new drivers

# Software I/O and OS

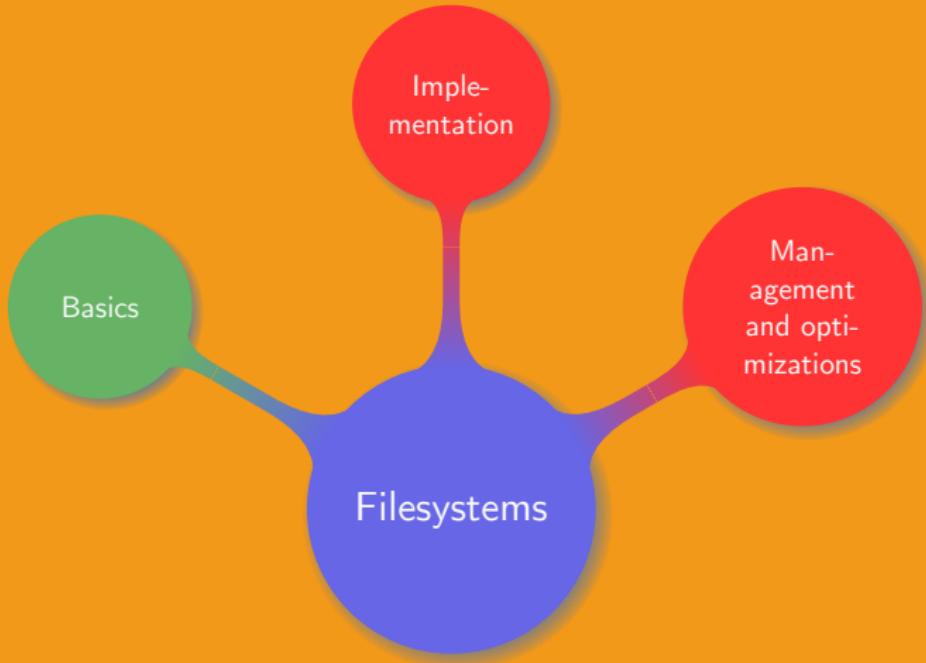




## 8. Filesystems

---

# Chapter organisation



Limitations of virtual memory:

- Small
- Volatile
- Process dependent

Limitations of virtual memory:

- Small
- Volatile
- Process dependent

Goals that need to be achieved:

- Store large amount of data
- Long term storage
- Information shared among multiple processes

High level view of a file-system:

- Small part of the disk memory can be directly accessed using high level abstraction called a *file*
- File name can be case sensitive or insensitive
- File name is a string with (an optional) suffix
- Each file has some attributes containing special information

Common system calls related to files (Unix):

- Create
- Write
- Delete
- Append
- Rename
- Seek
- Open
- Set attributes
- Close
- Get attributes
- Read

Structure content:

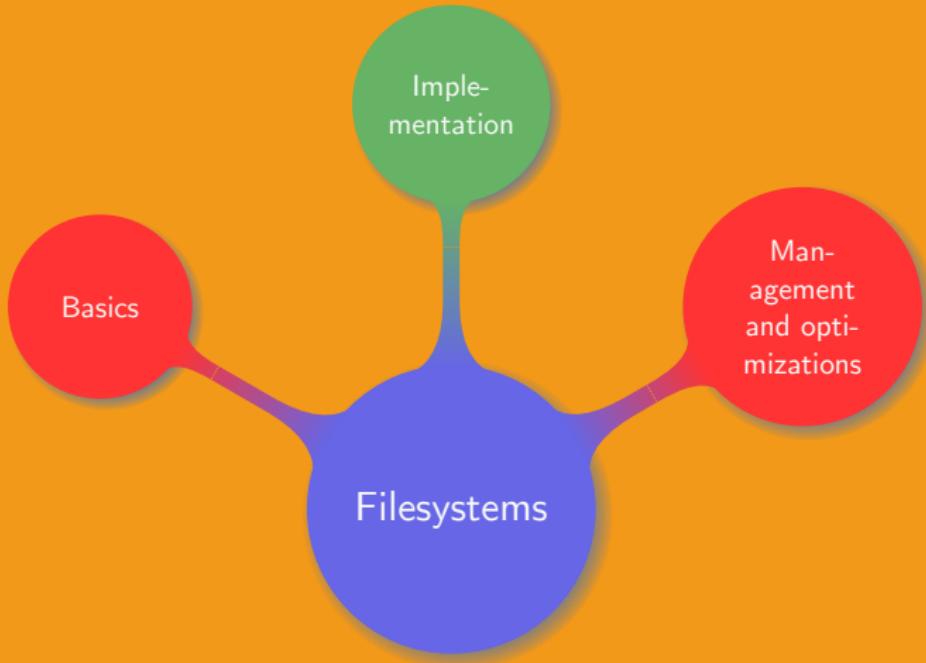
- Files are grouped inside a *directory*
- Directories are organised in a *tree*
- Each file has an *absolute path* from the root of the tree
- Each file has an *relative path* from the current location in the tree

# Directory operations

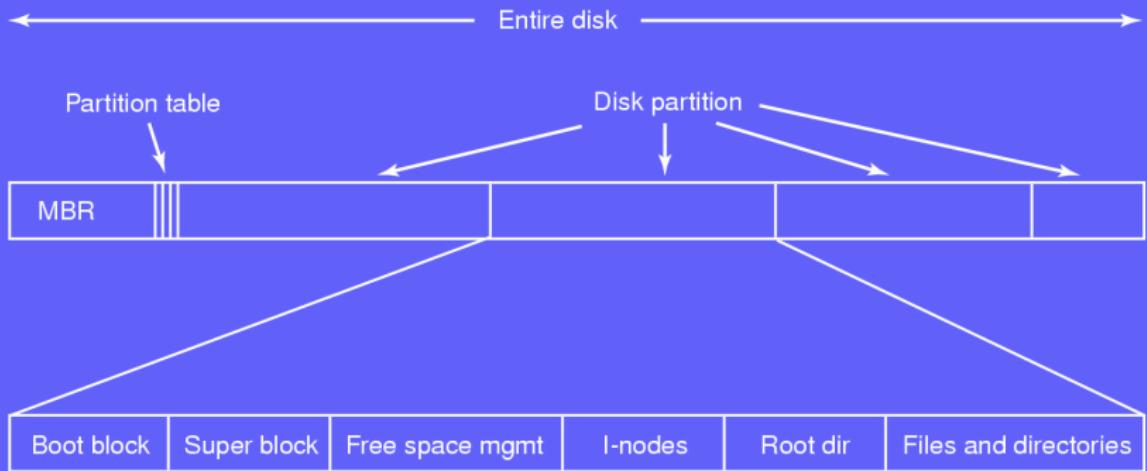
Common system calls related to directories (Unix):

- Create
- Delete
- Opendir
- Closedir
- Readdir
- Rename
- Link
- Unlink

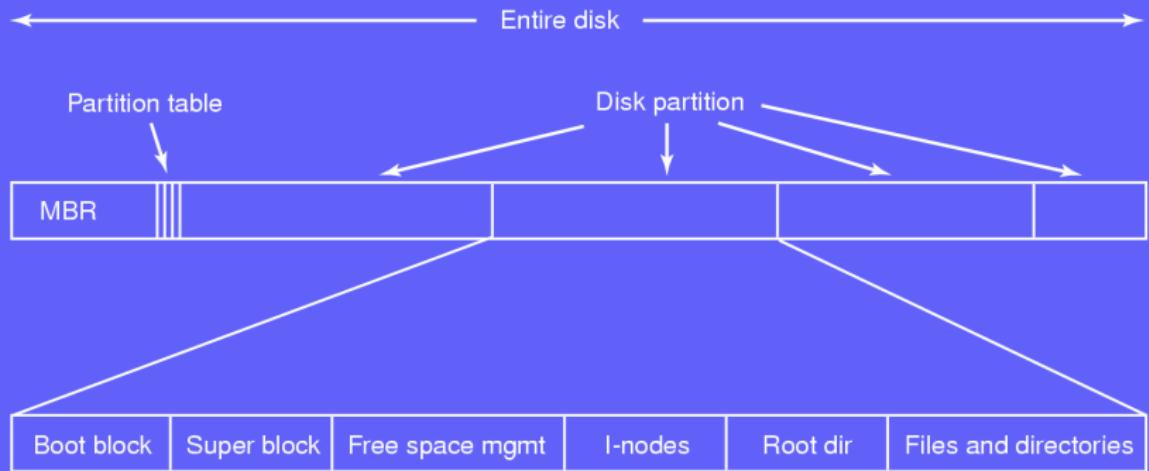
# Chapter organisation



# Basic disk layout

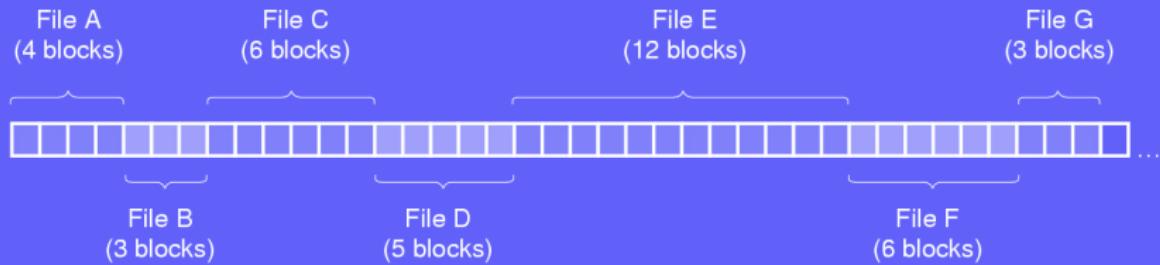


# Basic disk layout



*How to efficiently match disk blocks and files?*

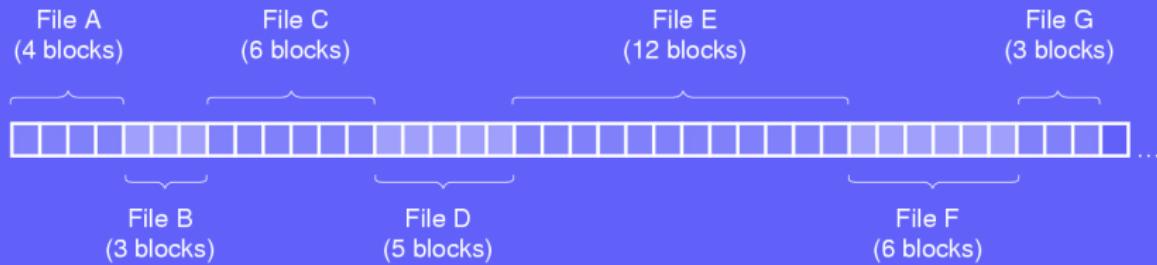
# Contiguous allocation



## Advantages:

- Simple to implement
- Fast: read a file using a single disk operation

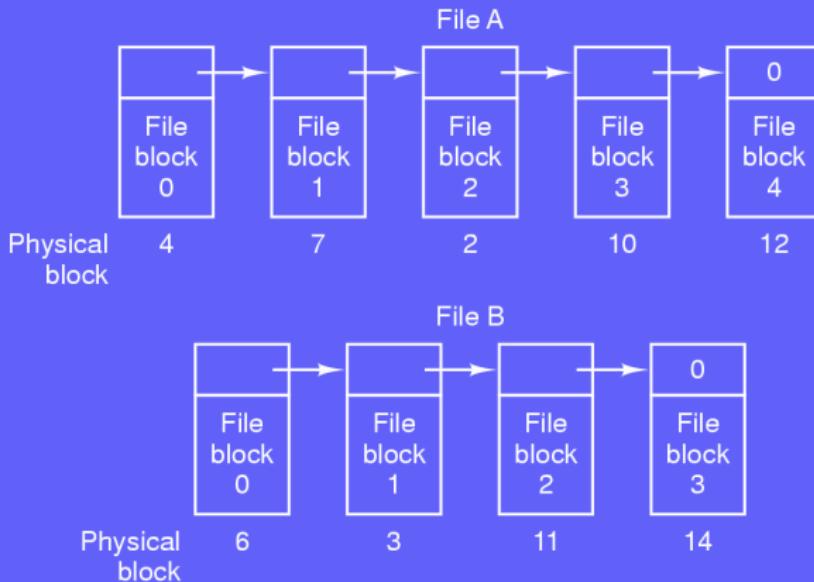
# Contiguous allocation



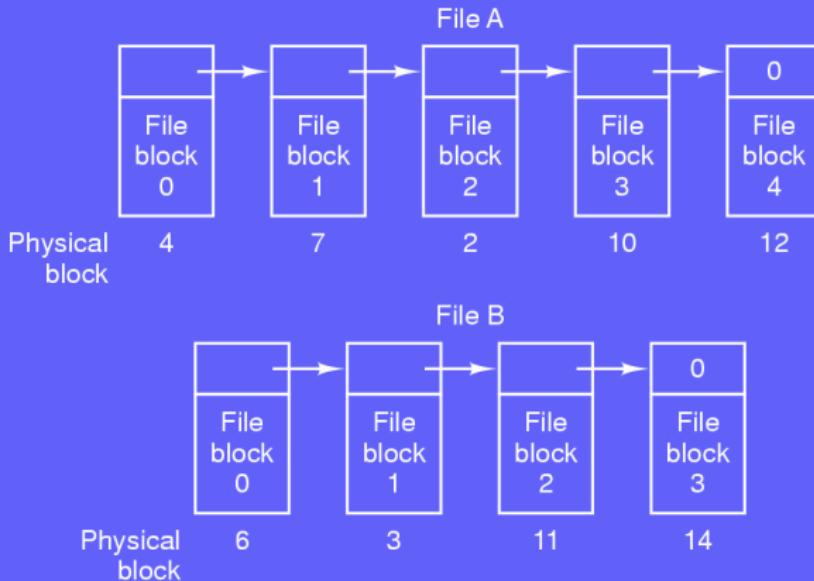
## Advantages:

- Simple to implement
- Fast: read a file using a single disk operation

**Drawback:** what if files *D* and *F* are deleted?



**Advantage:** no fragmentation



**Advantage:** no fragmentation

**Drawback:** slow random access

# File Allocation Table

Physical  
block

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

← File A starts here

← File B starts here

← Unused block

**Idea:** save the pointers on all the disk blocks inside a table in the main memory

**Advantage:** fast random access

# File Allocation Table

Physical  
block

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

← File A starts here

← File B starts here

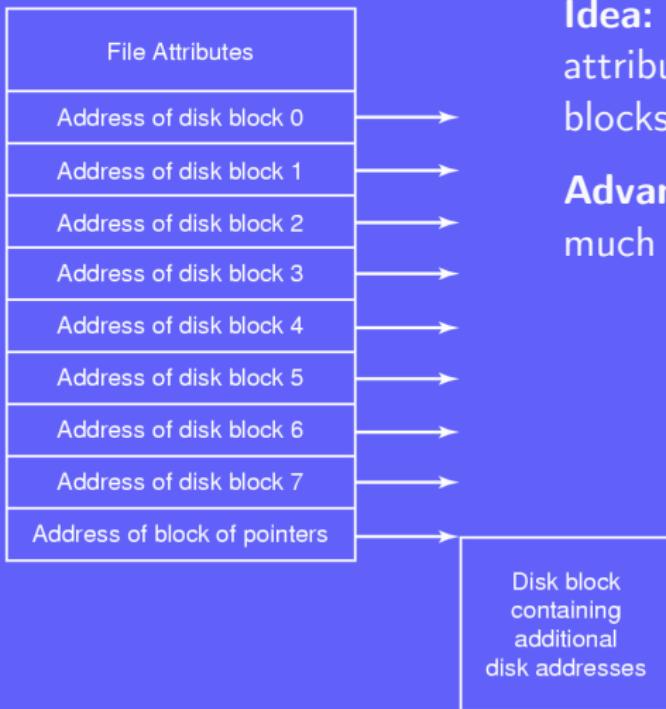
← Unused block

**Idea:** save the pointers on all the disk blocks inside a table in the main memory

**Advantage:** fast random access

**Drawback:** memory usage

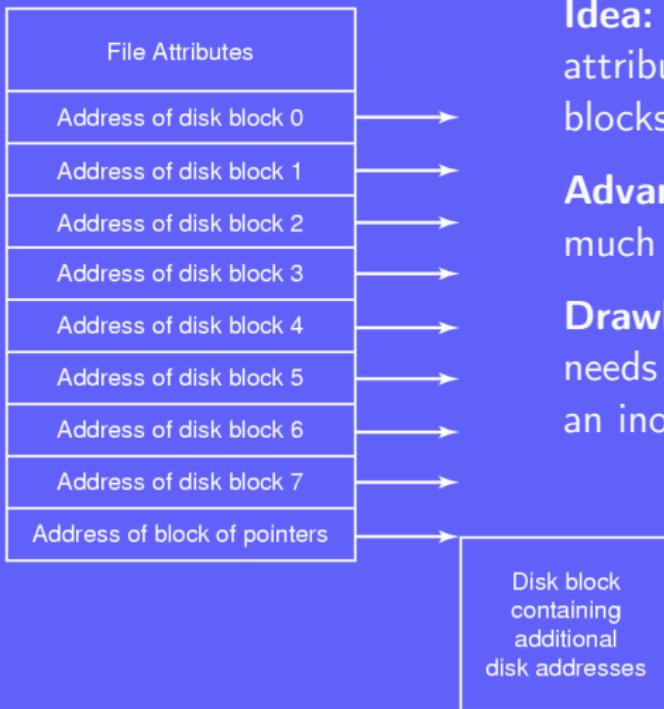
## Index node



**Idea:** structure containing the file attributes and pointers on the blocks where the file is written

**Advantage:** fast, do not require much memory

# Index node



**Idea:** structure containing the file attributes and pointers on the blocks where the file is written

**Advantage:** fast, do not require much memory

**Drawback:** what if a large file needs more blocks than can fit in an inode?

**Simple design:** fixed size entry (filename, attributes, disk address)

**Simple design:** fixed size entry (filename, attributes, disk address)

**Drawback:** how to handled long filenames?

**Simple design:** fixed size entry (filename, attributes, disk address)

**Drawback:** how to handle long filenames?

Entry for one file	File 1 entry length			
	File 1 attributes			
	p	r	o	j
	e	c	t	-
	b	u	d	g
	e	t	☒	
	File 2 entry length			
	File 2 attributes			
	p	e	r	s
	o	n	n	e
	l	☒		
File 3 entry length				
File 3 attributes				
f o o ☒				
⋮				

**Idea:** filename length not fixed

**Advantage:** can fit filename of arbitrary length

**Simple design:** fixed size entry (filename, attributes, disk address)

**Drawback:** how to handle long filenames?

Entry for one file

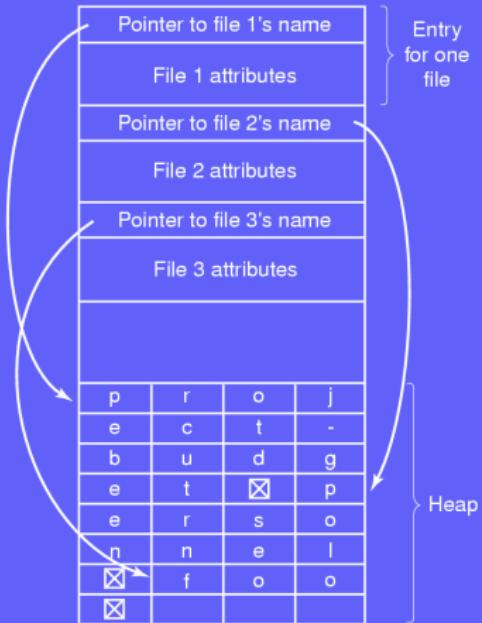
File 1 entry length			
File 1 attributes			
p	r	o	j
e	c	t	-
b	u	d	g
e	t	☒	
File 2 entry length			
File 2 attributes			
p	e	r	s
o	n	n	e
l	☒		
File 3 entry length			
File 3 attributes			
f	o	o	☒
⋮			

**Idea:** filename length not fixed

**Advantage:** can fit filename of arbitrary length

**Drawback:** space wasted, what if a directory entry spans multiple pages?

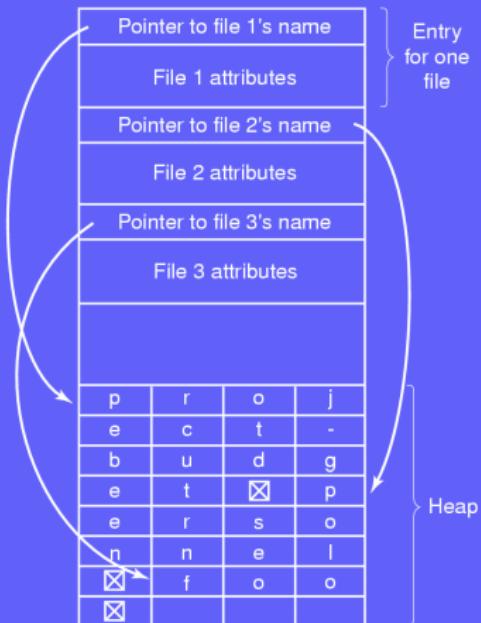
# Directories



**Idea:** pointer to the filename

**Advantage:** no waste of space,  
space can be easily reused when a  
file is removed

# Directories



**Idea:** pointer to the filename

**Advantage:** no waste of space,  
space can be easily reused when a  
file is removed

**Drawback:** as all the other  
strategies: slow on long directories

**Basic idea:** log the operation to be performed, run it, and erase the log

**Strategy:** if a crash interrupts an operation, re-run it on next boot

**Basic idea:** log the operation to be performed, run it, and erase the log

**Strategy:** if a crash interrupts an operation, re-run it on next boot

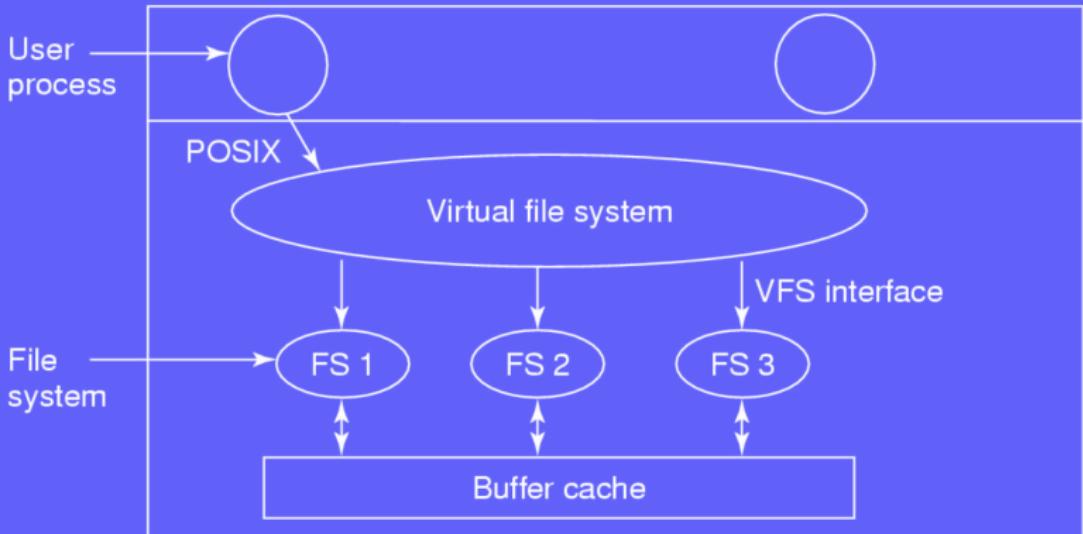
**Problem:** can any operation be applied more than once?

Example.

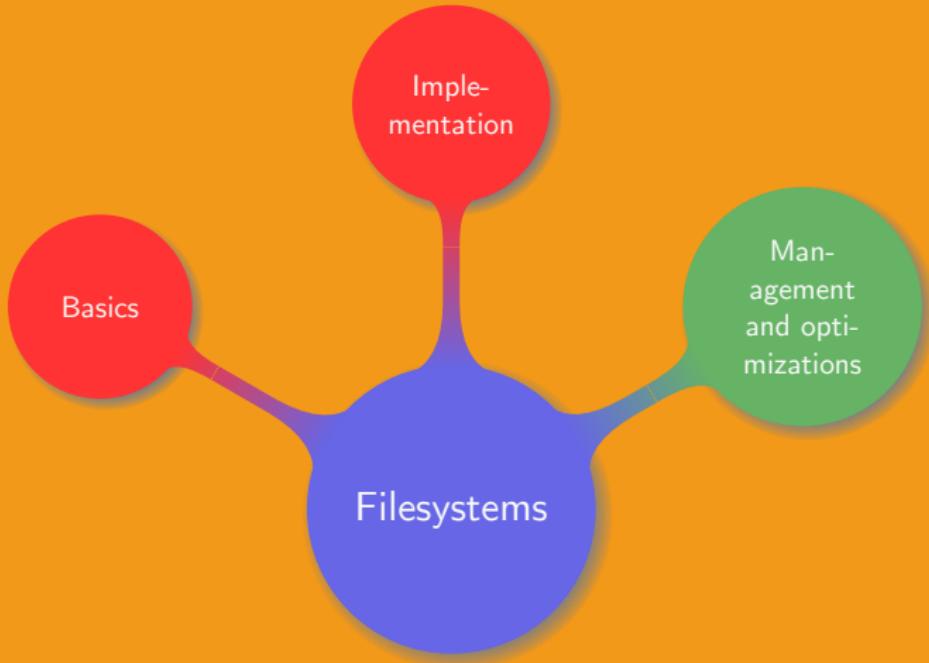
File deletion:

- (i) remove file from directory, (ii) release its i-node and (iii) add its disk blocks to the list of free blocks
- Operations (i) and (ii) can be repeated not (iii)

# Virtual FS



# Chapter organisation



**Problem:** how big should a block be?

Using small blocks:

- Large files use many blocks
- Blocks are not contiguous

**Conclusion:** time wasted

**Problem:** how big should a block be?

Using small blocks:

- Large files use many blocks
- Blocks are not contiguous

**Conclusion:** time wasted

Using large blocks:

- Small files do not fill up the blocks
- Many blocks partially empty

**Conclusion:** space wasted

**Problem:** how to keep track of free blocks?

**Problem:** how to keep track of free blocks?

- *Using a linked list:* free blocks addresses are stored in a block  
e.g. using 4KB blocks with 64 bits block address, how many free blocks addresses can be stored in a block?
- *Using a bitmap:* one bit corresponds to one free block
- *Using consecutive free blocks:* a starting block and the number of free block following it

*Which strategy is best?*

Checking the FS:

- Using the i-nodes, list in all the blocks used by all the files.  
Compare the complementary to the list of free blocks
- For every i-node in every directory increment a counter by 1.  
Compare those numbers with the counts stored in the i-nodes

## Checking the FS:

- Using the i-nodes, list in all the blocks used by all the files.  
Compare the complementary to the list of free blocks
- For every i-node in every directory increment a counter by 1.  
Compare those numbers with the counts stored in the i-nodes

## Common problems and solutions:

- Block related inconsistency:
  - List of free blocks is missing some blocks → add blocks to list
  - Free blocks appear more than once in list → remove duplicates
  - A block is present in more than one file → copy block and add it to the files
- File related inconsistency:
  - Count in i-node is higher → set link count to accurate value
  - Count in i-node is lower → set link count to accurate value

**Idea:** keep in memory some disk blocks using the LRU algorithm

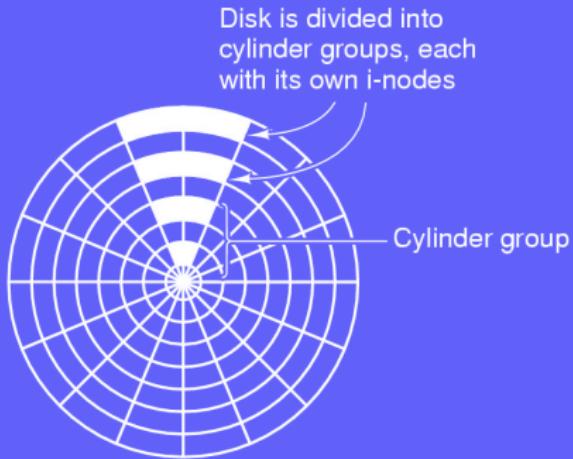
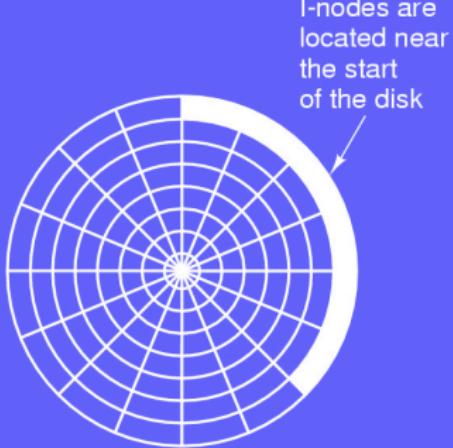
**Questions:**

- Is a block likely to be reused soon?
- What happens on a crash?

**Modified idea:**

- Useless to cache i-node blocks
- Dangerous to cache blocks essential to file system consistency
- Cache partially full blocks that are being written

# Arm motion



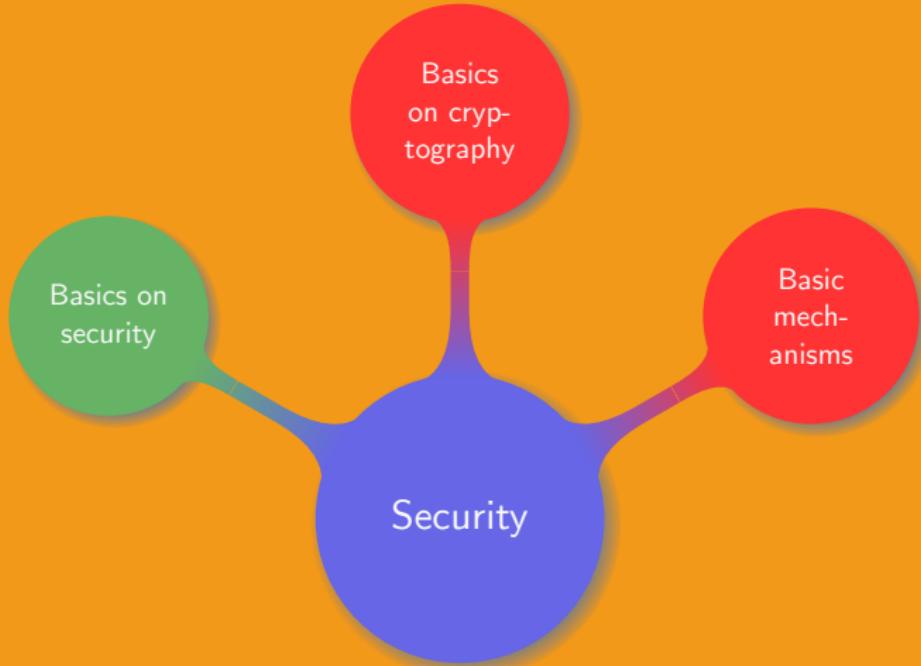
A few extra remarks related to file systems:

- Quotas: assign disk quotas to users
- Fragmentation: how useful is it to defragment a file system?
- Block read ahead: when reading block  $k$  assume  $k + 1$  will soon be needed and ensure its presence in the cache
- Logical volumes: file system over several disks
- Backups: how to efficiently backup a whole filesystem?
- RAID: Redundant Arrays of Inexpensive Disks



## 9. Security

# Chapter organisation



# What is security?

Simple reasoning:

- Security is needed to protect from some danger
- If the danger is unknown it is impossible to avoid it

# What is security?

Simple reasoning:

- Security is needed to protect from some danger
- If the danger is unknown it is impossible to avoid it

*What are the dangers?*

To define the dangers, the setup must be known:

- General setup: operating system
- Processes: privileges
- Memory: sensitive information processed
- I/O devices: intruders
- File system: sensitive data

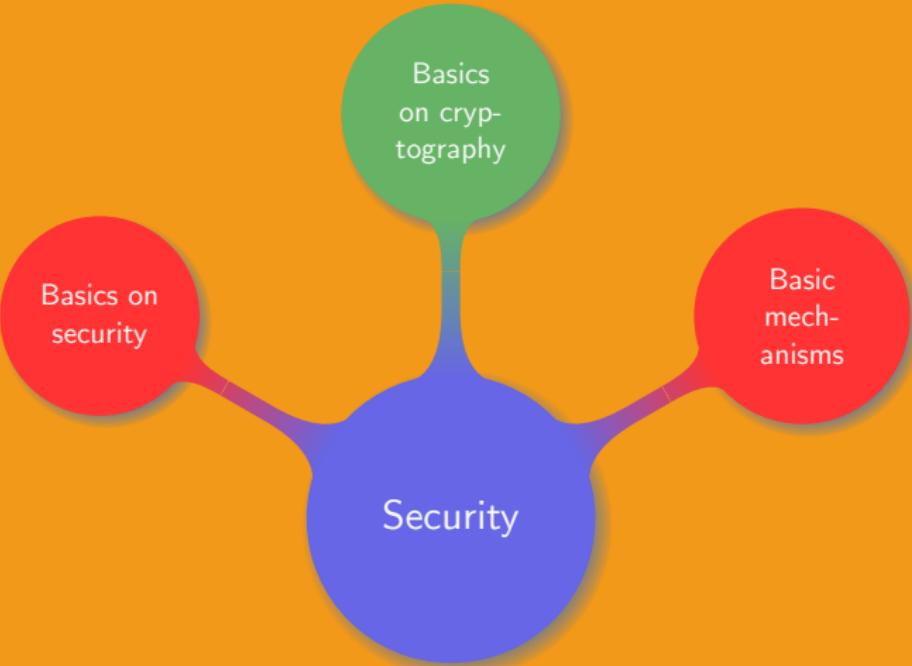
In an OS threats can be divided into four categories:

- Data stolen: confidentiality
- Data changed: integrity
- Intrusion: exclusion of outsiders
- Denial of service: system availability

Knowing who is likely to attack is of major importance:

- Local user reading other users files
- Regular other users on the same network
- Mafia
- Espionage
- Bad luck

# Chapter organisation



Cryptography, the science of secret:

- Confidentiality
- Data integrity
- Authentication

Two basic strategies:

- Symmetric
- Asymmetric

Encrypted data will remain confidential. How to encrypt data?

Symmetric:

- Shuffle all the letters of the alphabet and map the first one to A, the second one to B...
- One-time-pad: xor a message and a key of same length

**Question:** are those strategies secure?

Encrypted data will remain confidential. How to encrypt data?

Symmetric:

- Shuffle all the letters of the alphabet and map the first one to A, the second one to B...
- One-time-pad: xor a message and a key of same length

**Question:** are those strategies secure?

Asymmetric:

- Based on the concept of one-way-function
- Common examples: RSA, ELgamal

*Symmetric protocols better fit the OS setup*

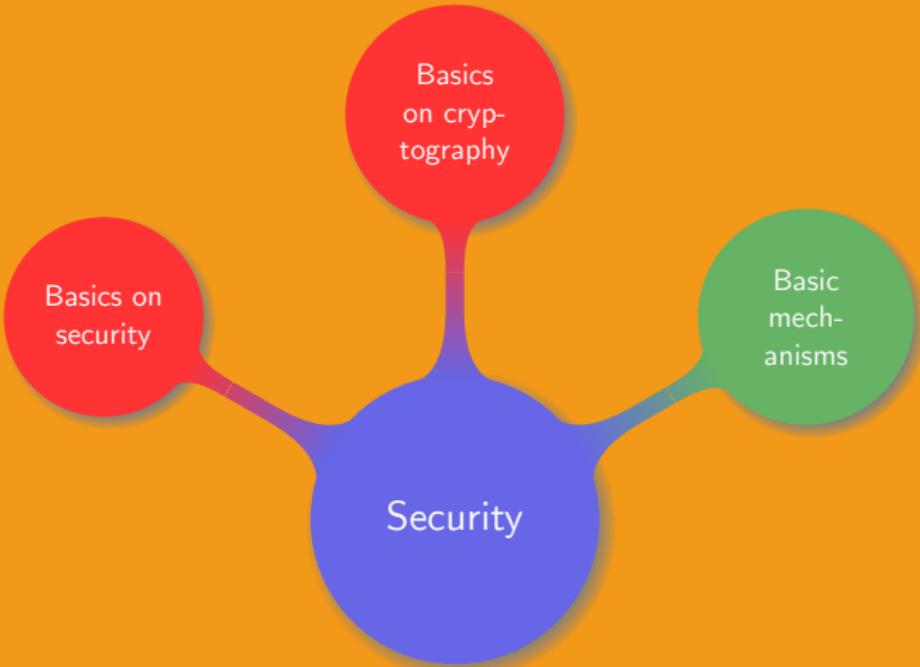
Ensure that data has not been altered using hash functions:

- Easy to compute
- Infeasible to generate a message with a given hash
- Infeasible to modify a message without modifying the hash
- Infeasible to find two different messages with same hash

Prove that a user is really who he pretends to be:

- Secret
- Challenge-response
- Token
- Biometrics

# Chapter organisation



Most obvious strategy is to setup a login and password:

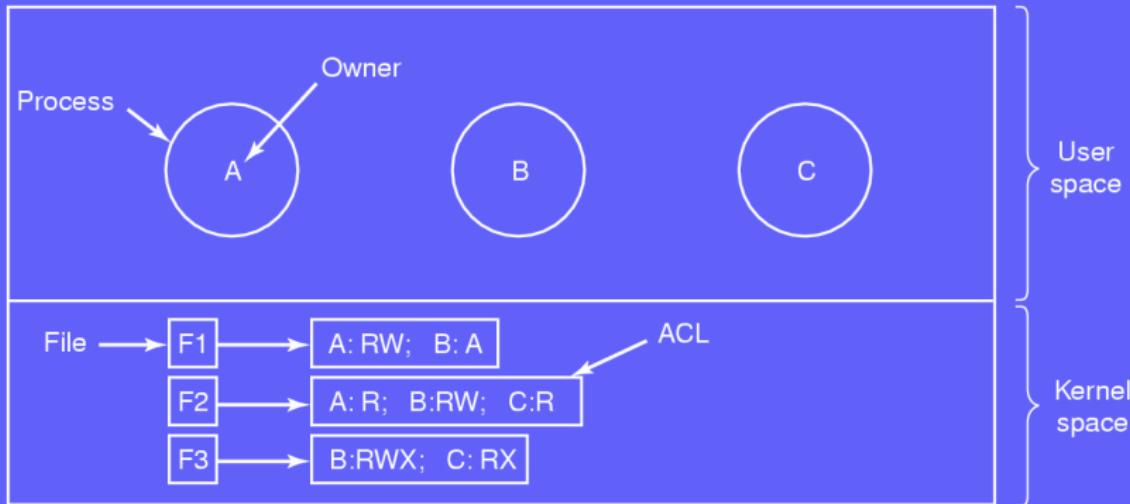
- Password should not be displayed when entered
- Should something be displayed when typing the password?
- When to reject a login: before or after the password input?
- What if the hard disk is mounted from another OS?

Most obvious strategy is to setup a login and password:

- Password should not be displayed when entered
- Should something be displayed when typing the password?
- When to reject a login: before or after the password input?
- What if the hard disk is mounted from another OS?

Other common strategy: use a key

# Access Control Lists



ACL are used to give users different privileges:

- Administrator: root/admin
- Privileged users: belong to special groups
- Regular user cannot access I/O devices

## Keeping a system secure – Basic

Remark.

An OS cannot be kept 100% secure

# Keeping a system secure – Basic

Remark.

An OS cannot be kept 100% secure

Basic strategy:

- Keep the system minimal
- No new software versions
- Regularly update the system
- Install software only from trusted parties
- Strong passwords or no password

## Keeping a system secure – Advanced

Advanced strategy:

- Apply the basic strategy
- Filter any outgoing network traffic
- Block any incoming new connection
- Keep a checksum of all the files
- Only use encrypted network traffic
- Use containers or virtual machines to run sensitive services
- Associate with each program a profile that restricts its capabilities

# Keeping a system secure – Paranoiac

Paranoiac strategy:

- Apply the advanced strategy
- Encrypt all the disk (including the swap)
- Isolate the computer (no network connection)
- Keep an encrypted checksum of all the files
- No extra device can be connected

# Keeping a system secure – Paranoiac

Paranoiac strategy:

- Apply the advanced strategy
- Encrypt all the disk (including the swap)
- Isolate the computer (no network connection)
- Keep an encrypted checksum of all the files
- No extra device can be connected

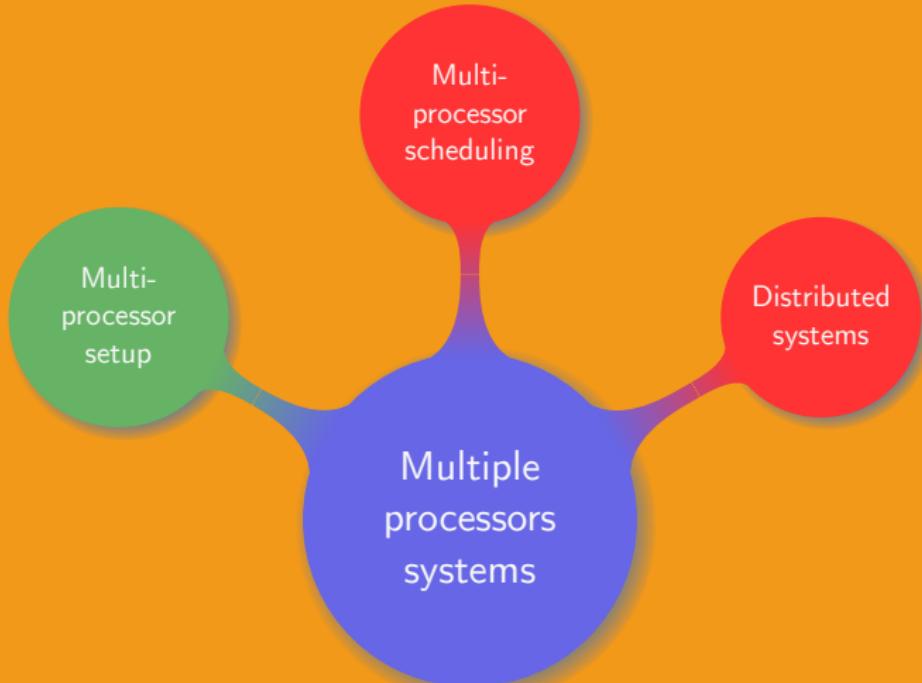
*Is it now safe?*



## 10. Multiple processors systems

---

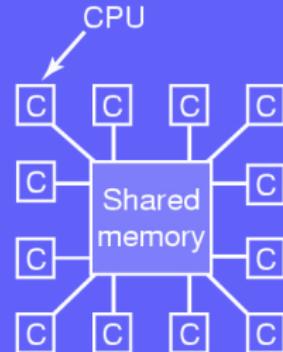
# Chapter organisation



# Shared memory model

Multiprocessors:

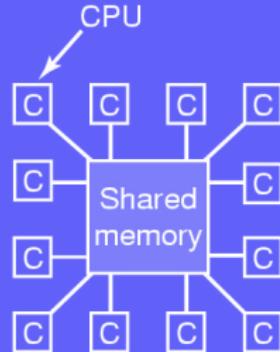
- CPUs communicate through the shared memory
- Every CPU has equal access to entire physical memory
- Access time: 2-10 ns



# Shared memory model

Multiprocessors:

- CPUs communicate through the shared memory
- Every CPU has equal access to entire physical memory
- Access time: 2-10 ns



Three main approaches:

- Each CPU has its own OS: no sharing, all independent
- Master-slave multiprocessors: one CPU handles all the requests
- Symmetric Multi-Processor: solution used in practice

## SMP approach

*A single copy of the OS that can be run by any of the CPUs*

**Problem:** what if two CPUs run the same process or claim the same free memory page at the same time?

*A single copy of the OS that can be run by any of the CPUs*

**Problem:** what if two CPUs run the same process or claim the same free memory page at the same time?

**Solution:**

- Many parts of the OS are independent
- Split the OS into multiple critical regions
- Add a mutex when entering those regions
- Add mutex to all shared tables

*A single copy of the OS that can be run by any of the CPUs*

**Problem:** what if two CPUs run the same process or claim the same free memory page at the same time?

**Solution:**

- Many parts of the OS are independent
- Split the OS into multiple critical regions
- Add a mutex when entering those regions
- Add mutex to all shared tables

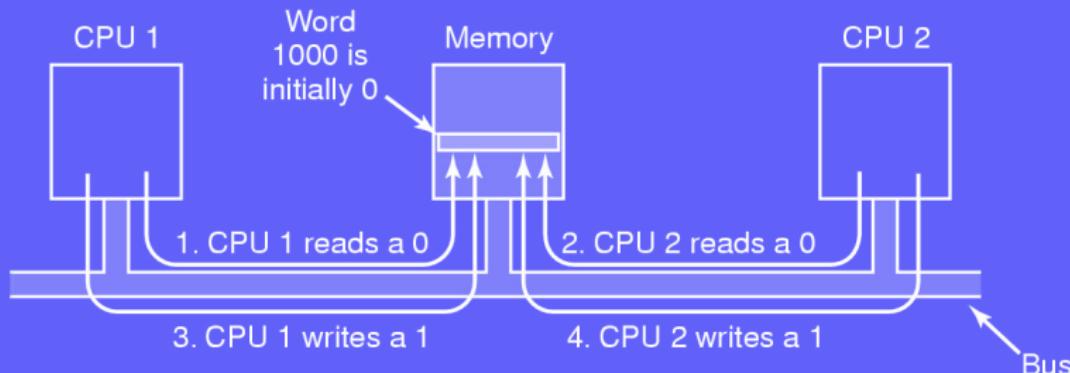
**Challenges:**

- How to divide up the OS
- Easy to run into deadlock with the shared tables
- Hard to keep consistency between programmers

# Synchronisation – Problem

## Synchronisation strategies:

- Single CPU:
  - Kernel space: disable interrupts
  - User space: take advantage of atomic operations, e.g. mutex
- Multiple CPUs:
  - Disabling interrupts on the current CPU is not enough
  - The memory bus needs to be disabled



## Synchronisation – Solution

A first idea:

- ① Lock the memory bus by asserting a special line on the bus
- ② The bus is only available to the processor which locked it
- ③ Perform the necessary memory accesses
- ④ Unlock the bus

# Synchronisation – Solution

A first idea:

- ① Lock the memory bus by asserting a special line on the bus
- ② The bus is only available to the processor which locked it
- ③ Perform the necessary memory accesses
- ④ Unlock the bus

**New multiprocessor issue:** slows down the whole system

**Solution:** do not lock the whole bus, but just the cache

## Cache with multiprocessors

Multiprocessors cache implementation:

- The requesting CPU reads the lock and copies it in its cache
- As long as the lock is unchanged the cached value can be used
- When the lock is released all the remote copies are invalidated
- All other CPUs must fetch the updated value

# Cache with multiprocessors

Multiprocessors cache implementation:

- The requesting CPU reads the lock and copies it in its cache
- As long as the lock is unchanged the cached value can be used
- When the lock is released all the remote copies are invalidated
- All other CPUs must fetch the updated value

**Problem:**

- The TSL instruction is used to acquire the lock
- The TSL instruction requires write access
- Any modified cached block must be invalidated
- A whole cache block needs to be recopied each time
- Much traffic is generated just to check the lock

Improved approach:

- Check if the lock is free using a read
- If the lock appears to be free apply the TSL instruction

*If two CPUs see the lock being free and apply the TSL instruction does it lead to a race condition?*

Improved approach:

- Check if the lock is free using a read
- If the lock appears to be free apply the TSL instruction

*If two CPUs see the lock being free and apply the TSL instruction does it lead to a race condition?*

A safe solution:

- The value returned by the read is only a hint
- Only one CPU can get the lock
- The TSL instruction prevents any race condition

Ethernet binary exponential back-off algorithm:

- Do not poll at regular intervals
- Add a loop where waiting time is doubled at each iteration
- Setup a maximum waiting time

Ethernet binary exponential back-off algorithm:

- Do not poll at regular intervals
- Add a loop where waiting time is doubled at each iteration
- Setup a maximum waiting time

Set a mutex for each CPU requesting the lock:

- When a CPU requests a lock it attaches itself at the end of the list of CPUs requesting the lock
- When the initial lock is released it frees the lock of the first CPU on the list
- The first CPU enters the critical region, does its work and releases the lock
- the following CPU on the list can start its work

# Spinning or switching?

New perspective:

- On a uniprocessor: the time spent on waiting is wasted
- On a multiprocessor: one CPU is waiting while another works

**Dilemma:** switching is expensive but looping is a waste

Best choice:

- Only know which solution was best after
- Impossible to have an always accurate optimal decision

# Spinning or switching?

New perspective:

- On a uniprocessor: the time spent on waiting is wasted
- On a multiprocessor: one CPU is waiting while another works

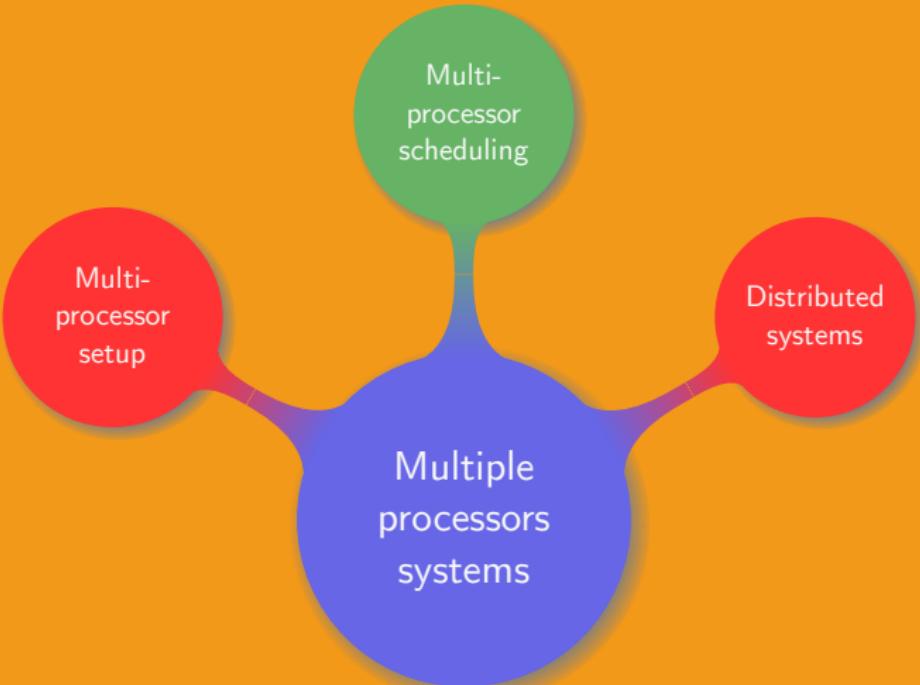
**Dilemma:** switching is expensive but looping is a waste

Best choice:

- Only know which solution was best after
- Impossible to have an always accurate optimal decision

**Solution:** mix of waiting and switching with a (variable) threshold

# Chapter organisation



Single threaded process: only need to schedule the process

Uniprocessor: which thread to run?

Multiprocessor:

- Which thread to run?
- Which CPU to run it on?

Single threaded process: only need to schedule the process

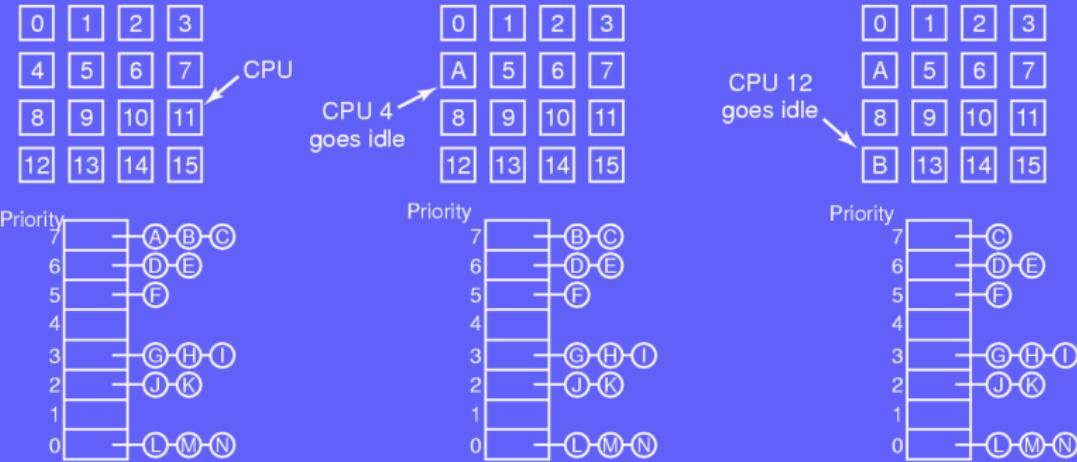
Uniprocessor: which thread to run?

Multiprocessor:

- Which thread to run?
- Which CPU to run it on?
- Threads of a process run on the same CPU → no need to reload the whole process
- Threads of a process run in parallel → threads can cooperate more efficiently

# Time-sharing

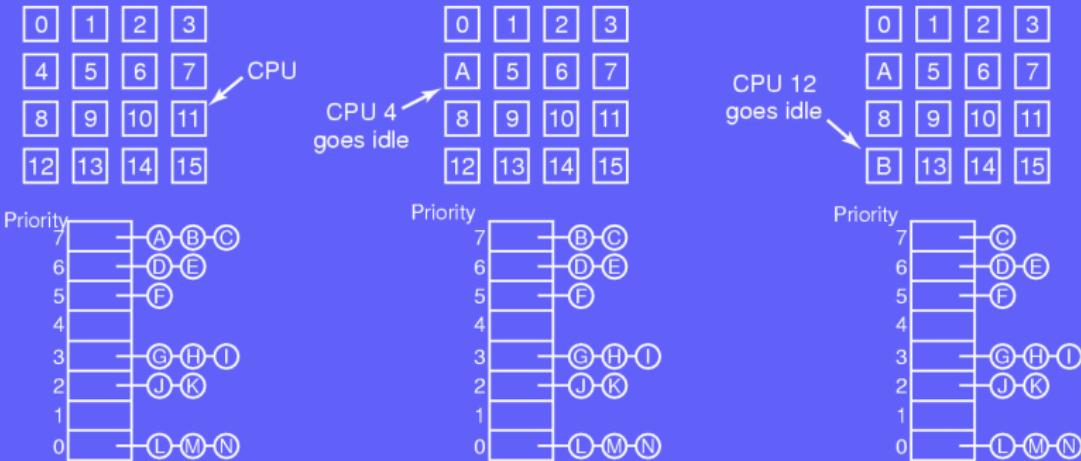
291



- Single data structure for ready processes
- Simple and efficient implementation

**Limitation:** a thread holds a spin lock but reaches the end of its quantum

# Time-sharing



- Single data structure for ready processes
- Simple and efficient implementation

**Limitation:** a thread holds a spin lock but reaches the end of its quantum → other CPUs spin waiting for lock to be released

# Improvements on time-sharing

## Smart scheduling:

- A thread holding a spin lock sets a flag
- Scheduler lets such thread run after the end of the quantum
- Clear the flag when lock is released

## Affinity scheduling:

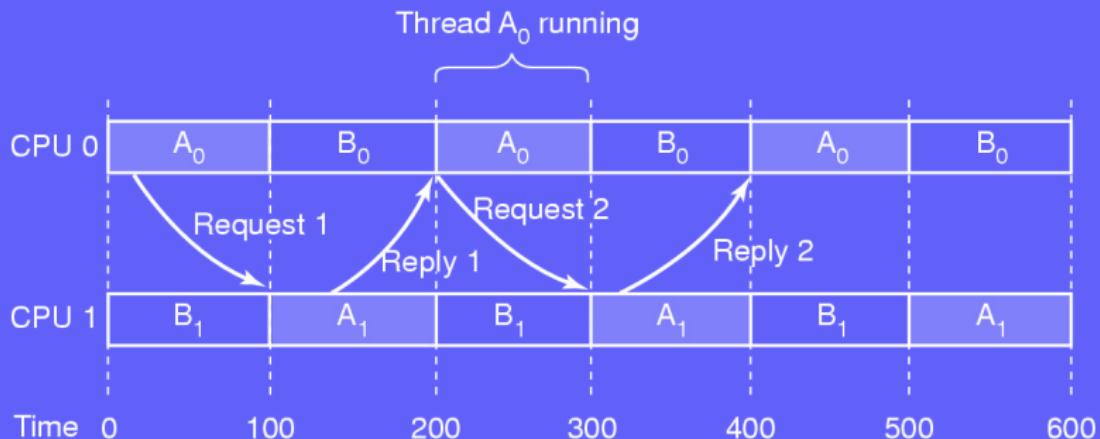
- Thread is assigned a CPU when it is created
- Try as much as possible to run a thread on the same CPU
- Cache affinity is maximized

When a process is created the scheduler checks if the number of free CPUs is larger then the number of threads:

- Yes: run one thread per CPU until completion
- No: wait for more free CPUs
- Keep the CPU even during I/O

**Optimization:** try to apply shortest job first → in practice FIFO better

# The communication problem

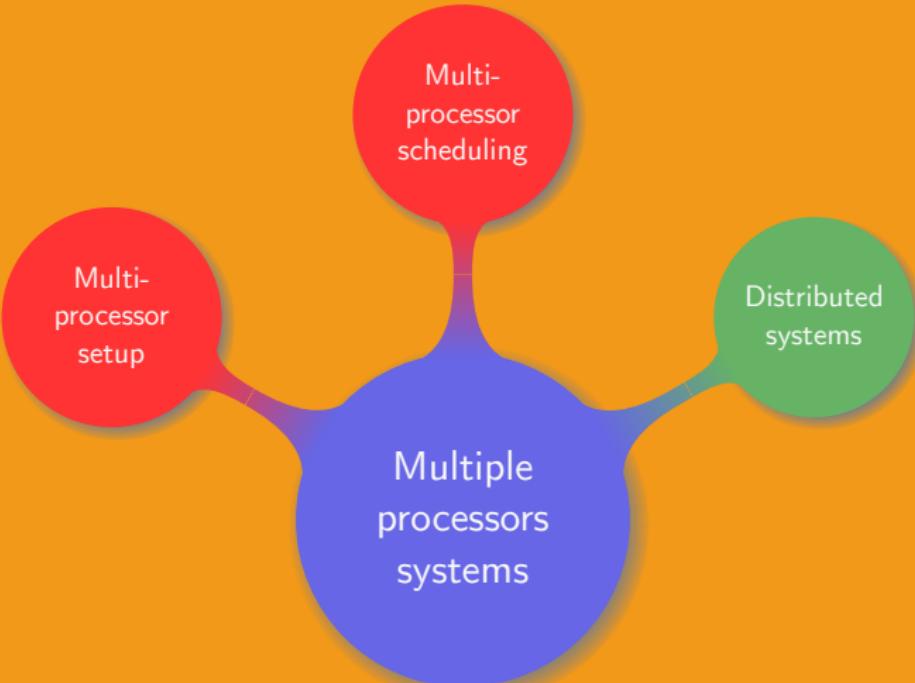


**Simple idea:** schedule processes by group:

- Group related threads into a gang
- All gang members run simultaneously on different CPUs
- All members start and end at the same time
- No intermediary scheduling decision is taken

**Limitations:** what if one gang member issue an I/O request or finishes earlier than others?

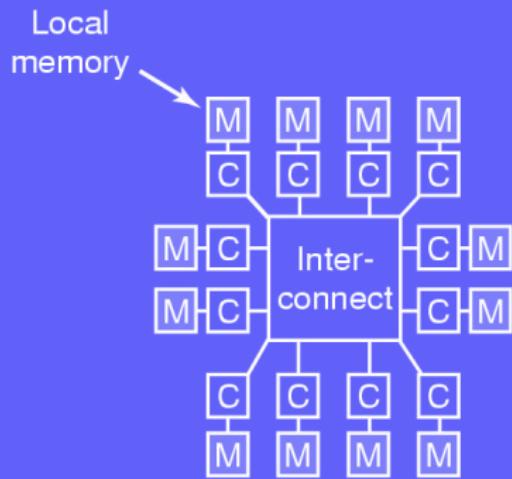
# Chapter organisation



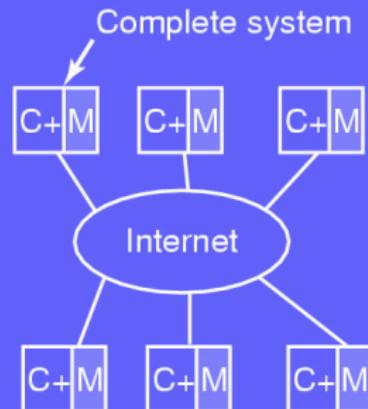
Characteristics of distributed systems:

- Composed of autonomous entities, each with its own memory
- Communication is done over a network using message passing
- The system must tolerate node failures
- All the nodes perform the same task

Cluster: set of connected computers working together



Multicomputer



Distributed system

General idea of how a cluster works:

- Computing nodes connected over a LAN
- A clustering middleware sits on the top of the node
- Users view a large computer

Example.

A single master node handles the management of the scheduling and slave nodes

Main challenges:

- Scheduling: where should a job be scheduled?
- Load balancing: should a job be rescheduled on another node?

## Apache Hadoop:

- Opensource framework for distributed file system
- Written in Java
- Very large files stored across multiple nodes
- Used and enhanced by Yahoo!, Facebook, Amazon, Microsoft, Google, IBM...

## More on distributed computing

Advances in network technologies lead to the development of:

- Volunteer computing: volunteer offer part of their computational power to some project
- Grids: collection of computer resources from multiple locations to reach a common goal
- Jungle computing: network not necessarily composed of “regular computers”





Thank you, enjoy the Winter break!