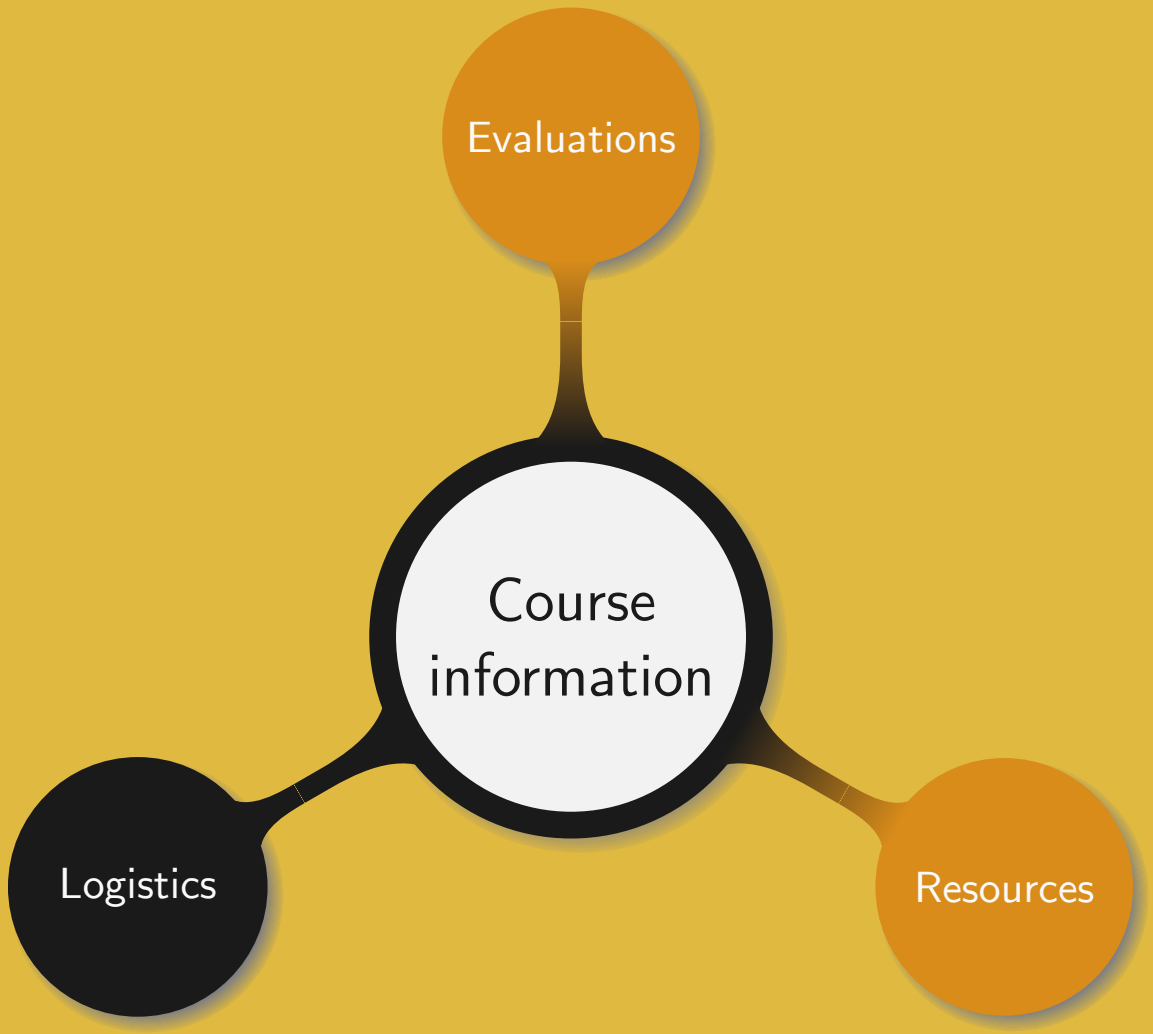
Introduction to Operating Systems

Manuel – Fall 2021

Table of contents

0. Course information

Chapter organisation



Basic information

Teaching team:

- Instructor: Manuel (charlem@sjtu.edu.cn)
- Teaching assistants:
  - Yiding (colossuschang@sjtu.edu.cn)
  - Yuchi (citrate@sjtu.edu.cn)
  - Boming (bomingzh@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 schedule

Course arrangements:

- Lectures:
  - Tuesday 10:00 – 11:40
  - Thursday 10:00 – 11:40
- Labs:
  - Tuesday 18:20 – 20:55
  - Friday 12:10 – 14:45
- Manuel's office hours: Appointment (TBD)
- TAs' office hours: TBA

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

Course workflow

Learning strategy:

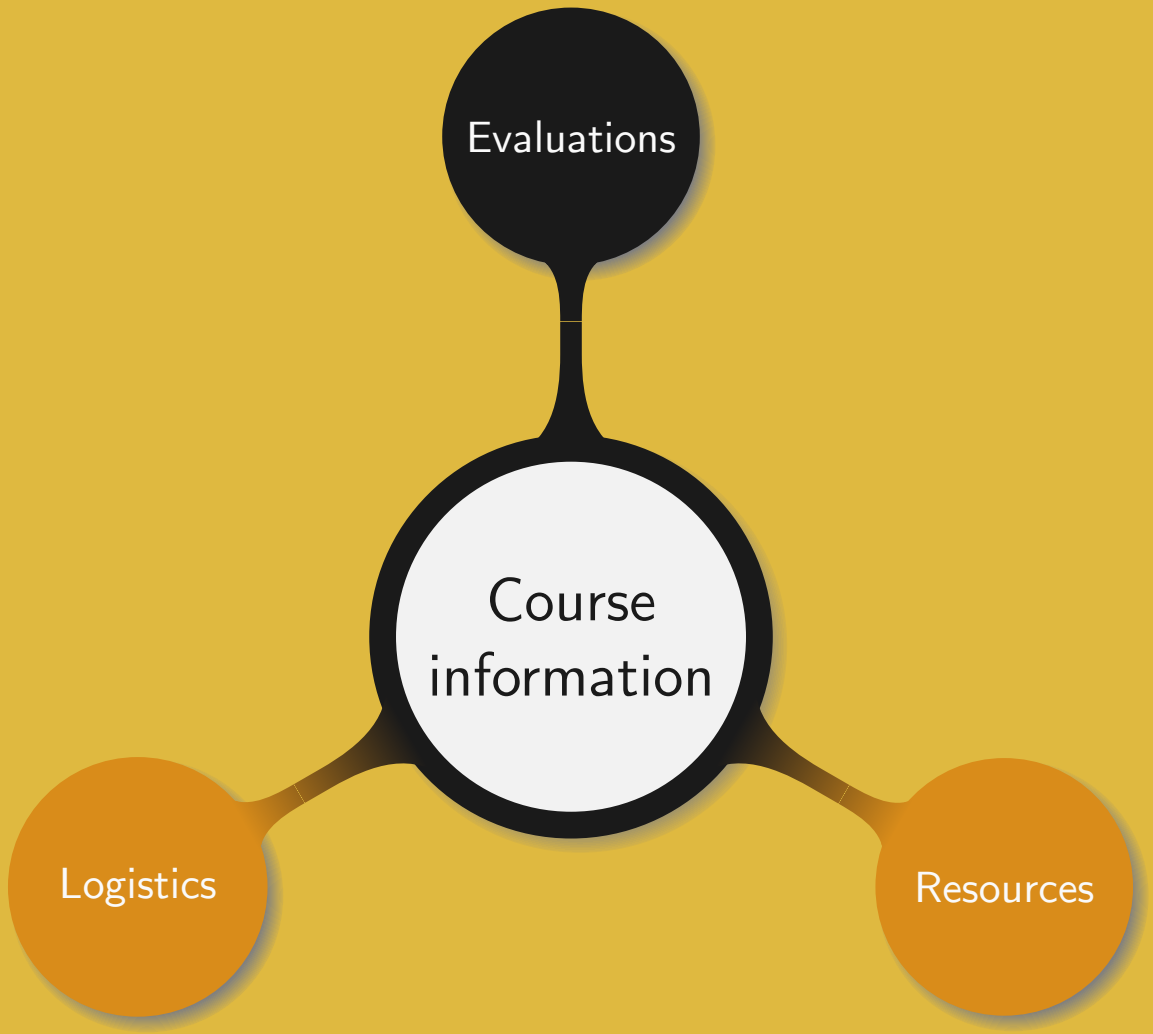
- Course side:
  1. Understand how to efficiently use the CPU
  2. Know how to handle Memory, Input/Output, and Filesystems
  3. Get a basic idea of security and distributed systems
- Personal side:
  1. Read and write code
  2. Relate known strategies to new problems
  3. Perform extra research

Course outcomes

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



Assignments

Homework:

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

Labs:

- Total: 9 + 3
- Content: improve programming skills, work on projects

Projects:

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

Extra: Linux kernel challenges

Grading policy

Grade weighting:

- Homework: 10%
- Projects: 40%
- Labs: 10%
- 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 $[\![B, B+]\!]$

Honor Code

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

Honor Code

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

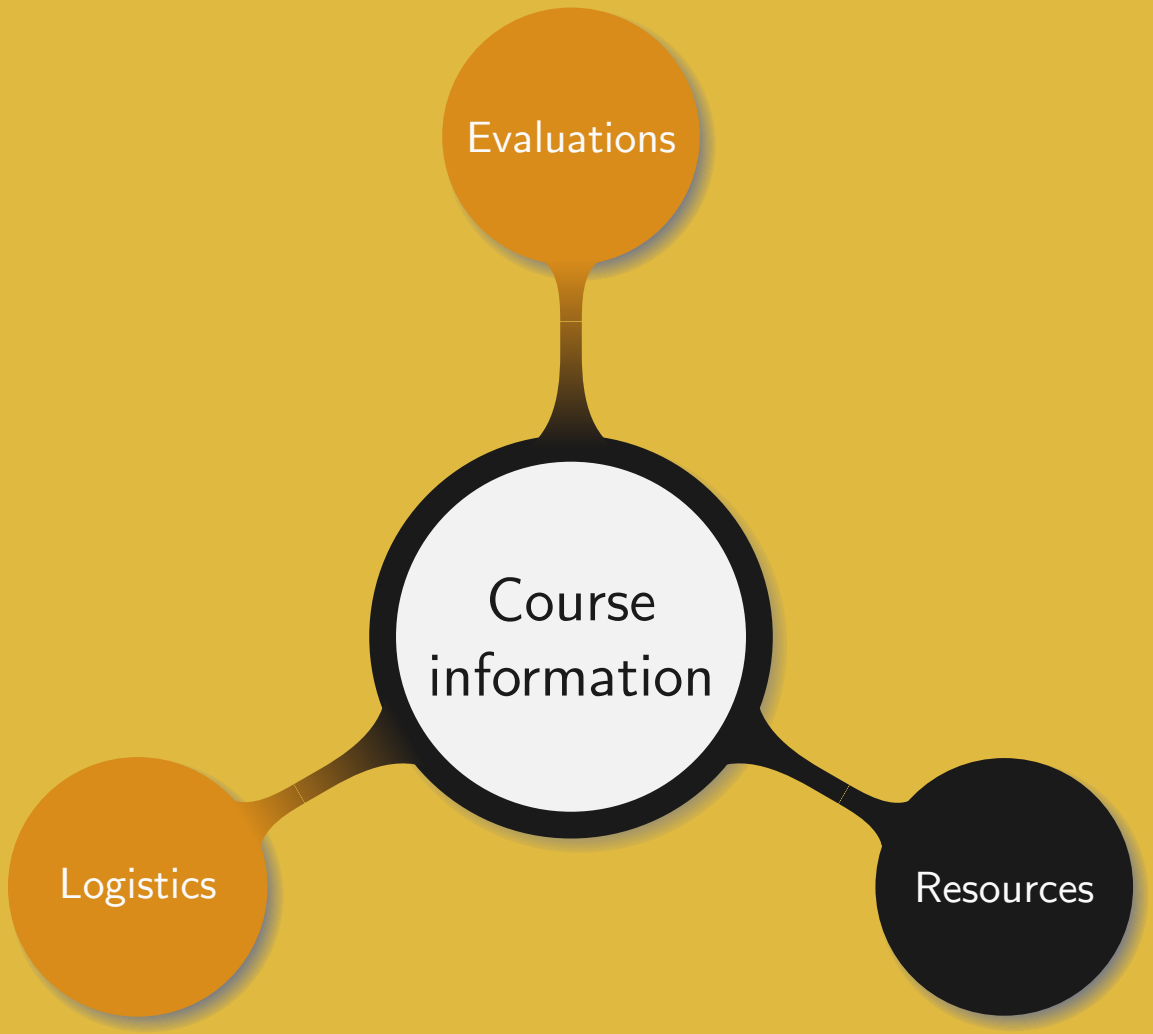
Special circumstances

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



Canvas

On Canvas platform:

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

References

Useful places where to find information:

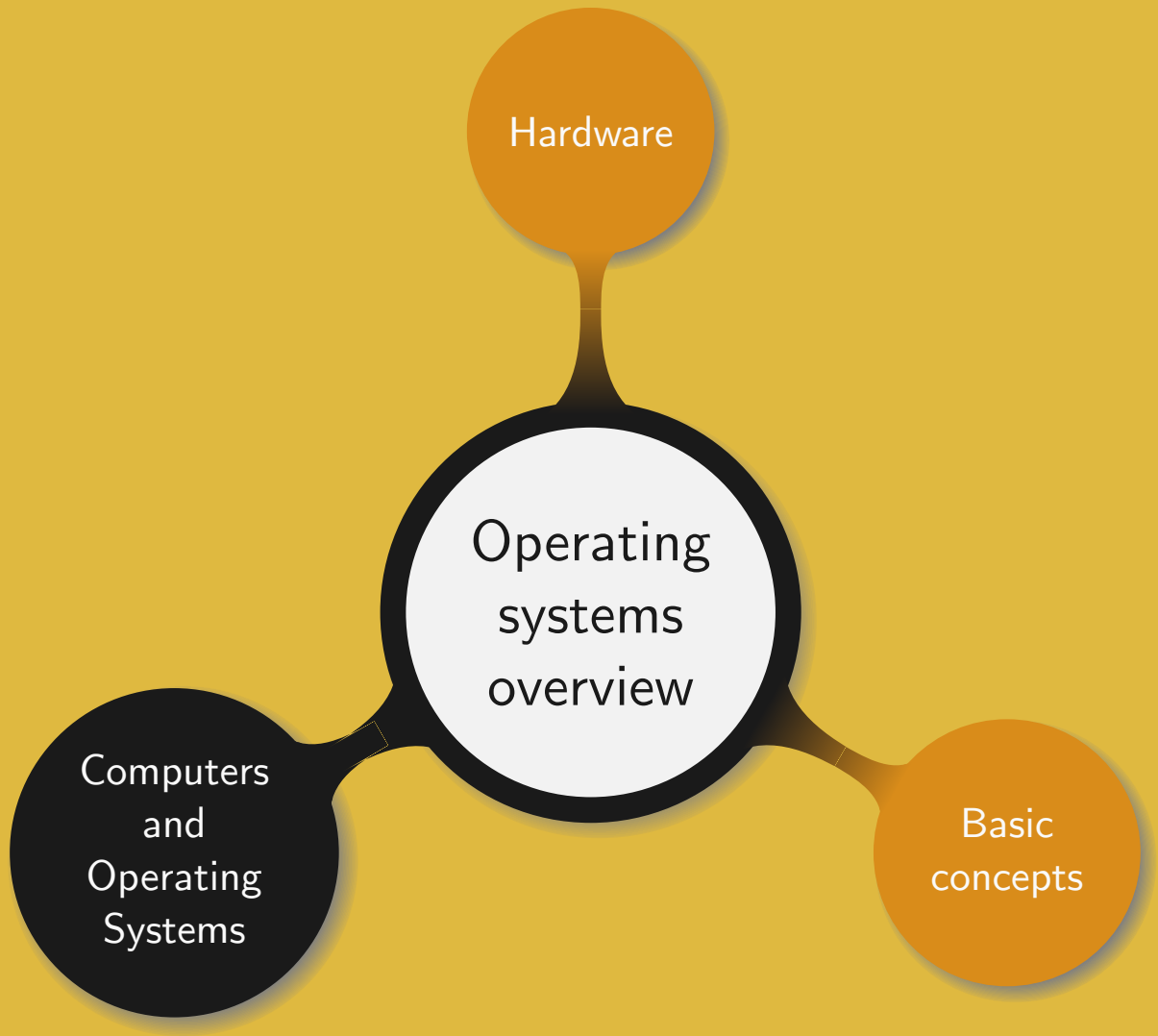
- *Modern Operating Systems*, A. Tanembaum
- *Operating System Concepts*, A. Silberschatz
- *Operating Systems: Three Easy Pieces*, R. and A. Arpcaci-Dusseau
- OS creation: `http://wiki.osdev.org/Main_Page`
- Piazza
- Search information online, i.e. $\{internet \setminus \{non\text{-}English\ websites\}\}$

Key points

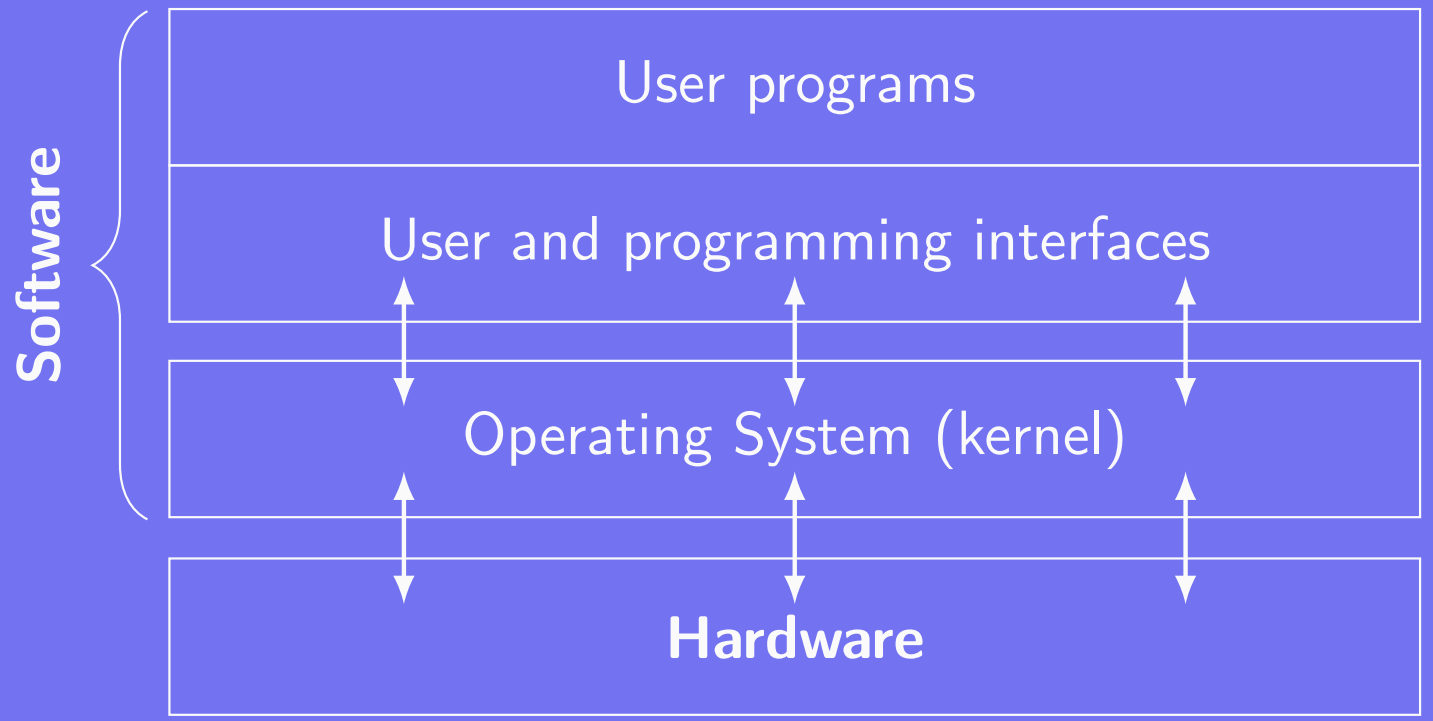
- Work regularly, do not wait the last minute
- Respect the Honor Code
- Go beyond what is taught
- Do not learn, understand
- Keep in touch with us
- Advice and suggestions are always much appreciated

1. Operating systems overview

Chapter organisation



Software and hardware layout



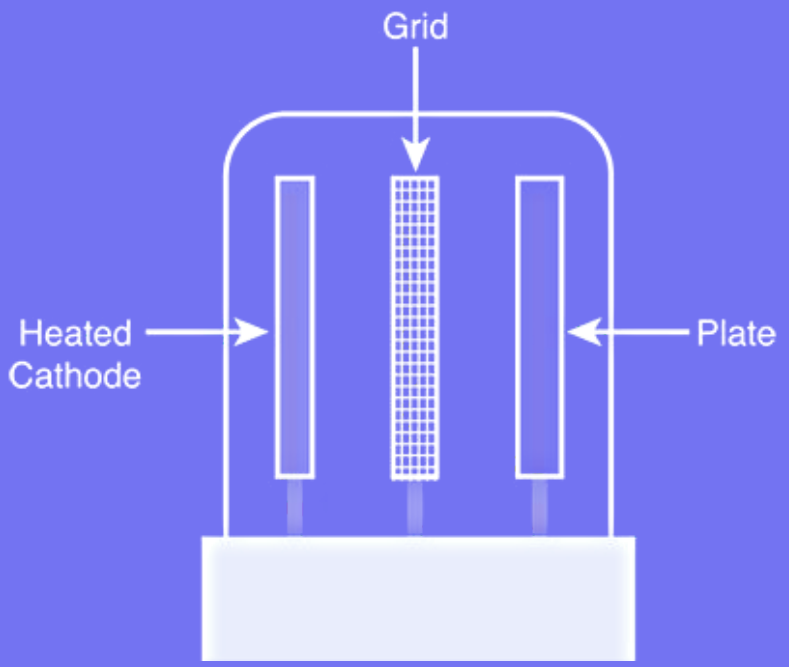
Job of an Operating System (OS):

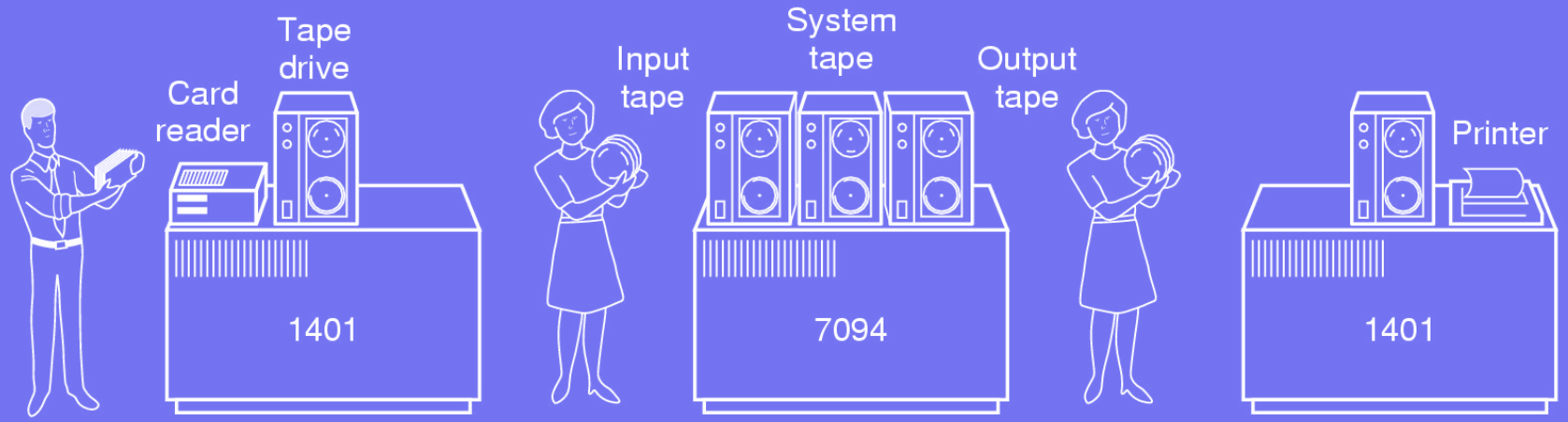
- Manage and assign the hardware resources
- Hide complicated details to the end user
- Provide abstractions to ease interactions with the hardware

A bit of history

The early days:

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





Remmington Rand 409

Using the device:

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

A bit of history

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

Modern OS

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...)

Common OS

Device and task oriented OS types:

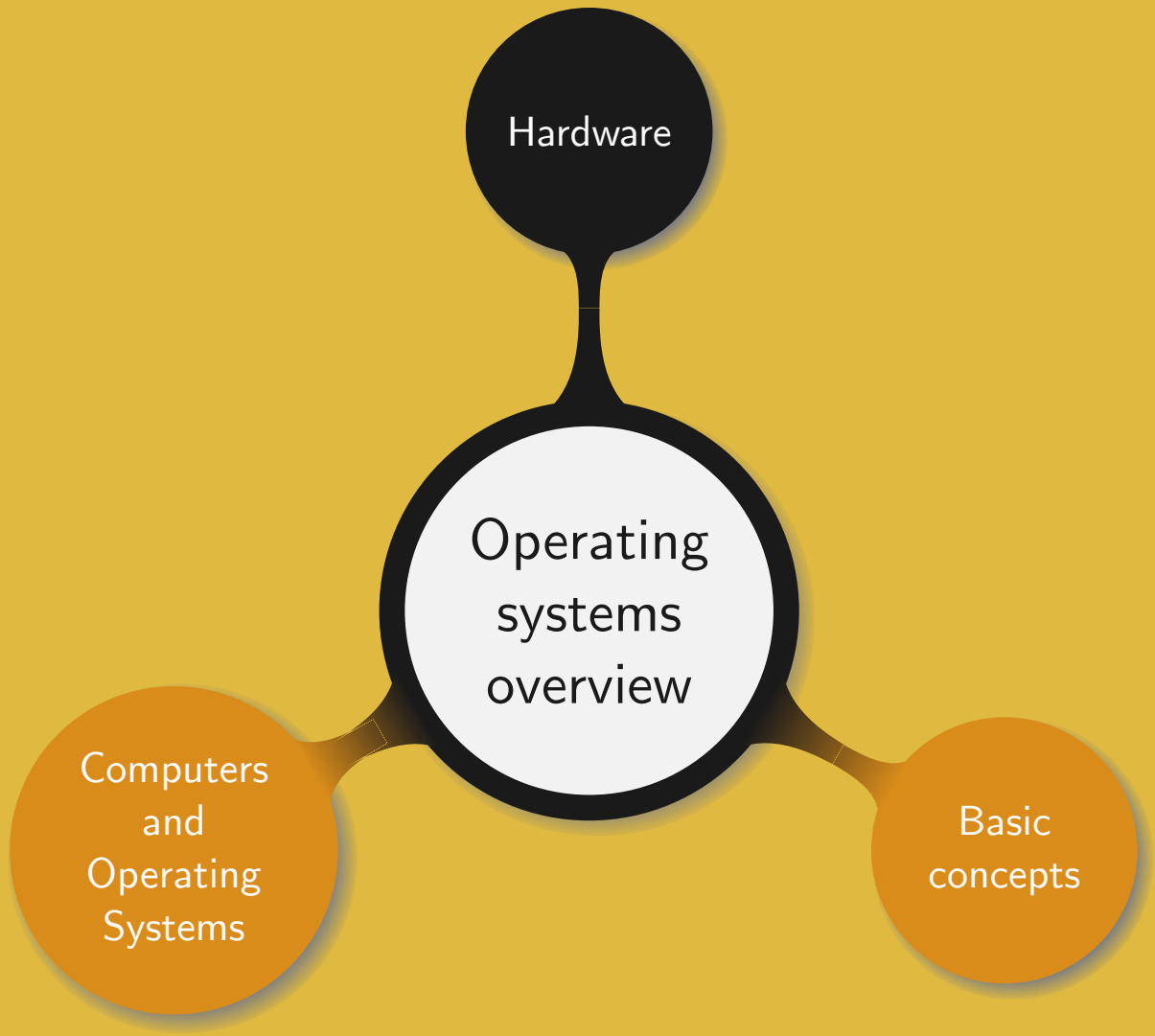
- Personal Computers (PC)
- Servers: serve users over a network (print, web, IM...) $\rightarrow$ Solaris, FreeBSD, Linux, Windows Server
- Multi processors: multiple CPU in a single system $\rightarrow$ Linux, Windows, OS X...
- Handheld computers: PDA, smartphone
- Embedded devices: TV, microwave,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

Less common OS

More device and task oriented OS types:

- Real-Time: time is key parameter (e.g. assembly line, army, avionics...) $\rightarrow$ overlap with embedded/handheld systems
- Mainframe: room-sized computers, data centers $\rightarrow$ 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 $\rightarrow$ simple OS design TinyOS
- Smart card: credit card size with a CPU chip, severe memory/processing constraints $\rightarrow$ smallest/primitive OS

Chapter organisation



Basic PC hardware

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?

CPU

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

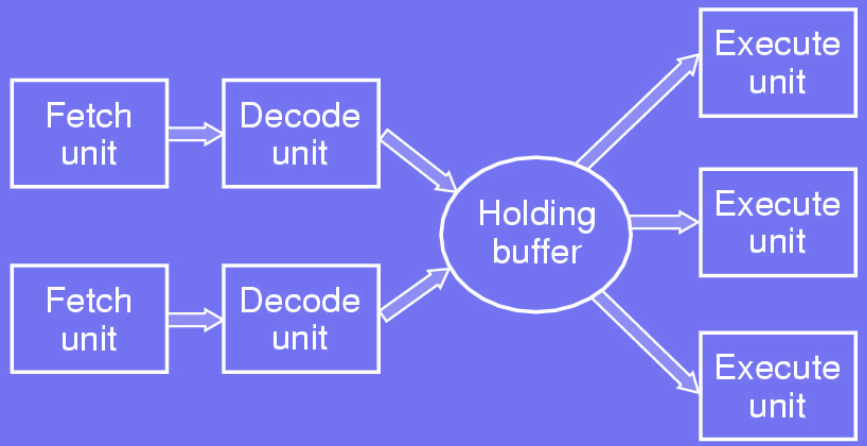
Pipeline vs. superscalar

Superscalar:

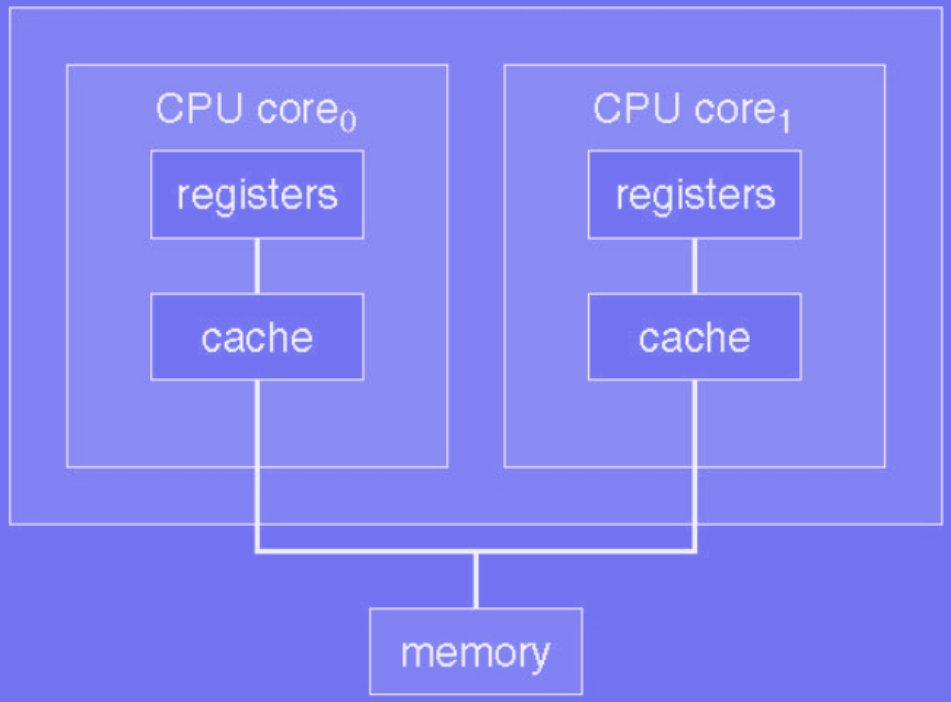
- Multiple execution units
  e.g. one for float, int, boolean
- Multiple instructions fetched and decoded at a time
- Instructions held in buffer to be executed
- Issue: no specific order to execute buffered instructions

Pipeline:

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



Multi-threading



Real multi-threading:

- Several physical CPU cores are available
- The OS sees several CPUs and can use them all *simultaneously*

Hyper-threading

Fake multi-threading:

- A physical CPU core is seen as two logical cores by the OS
- Some resources are duplicated in each physical core
- Hyper-threading allows a better utilisation of the CPU

Intel terminology:

- Socket: the physical computing component
- Core: number of independent CPUs on a socket
- Threads: *maximum* number of instructions that can be passed through or processed simultaneously by a single core
- Number of logical cores: number of cores times number of threads

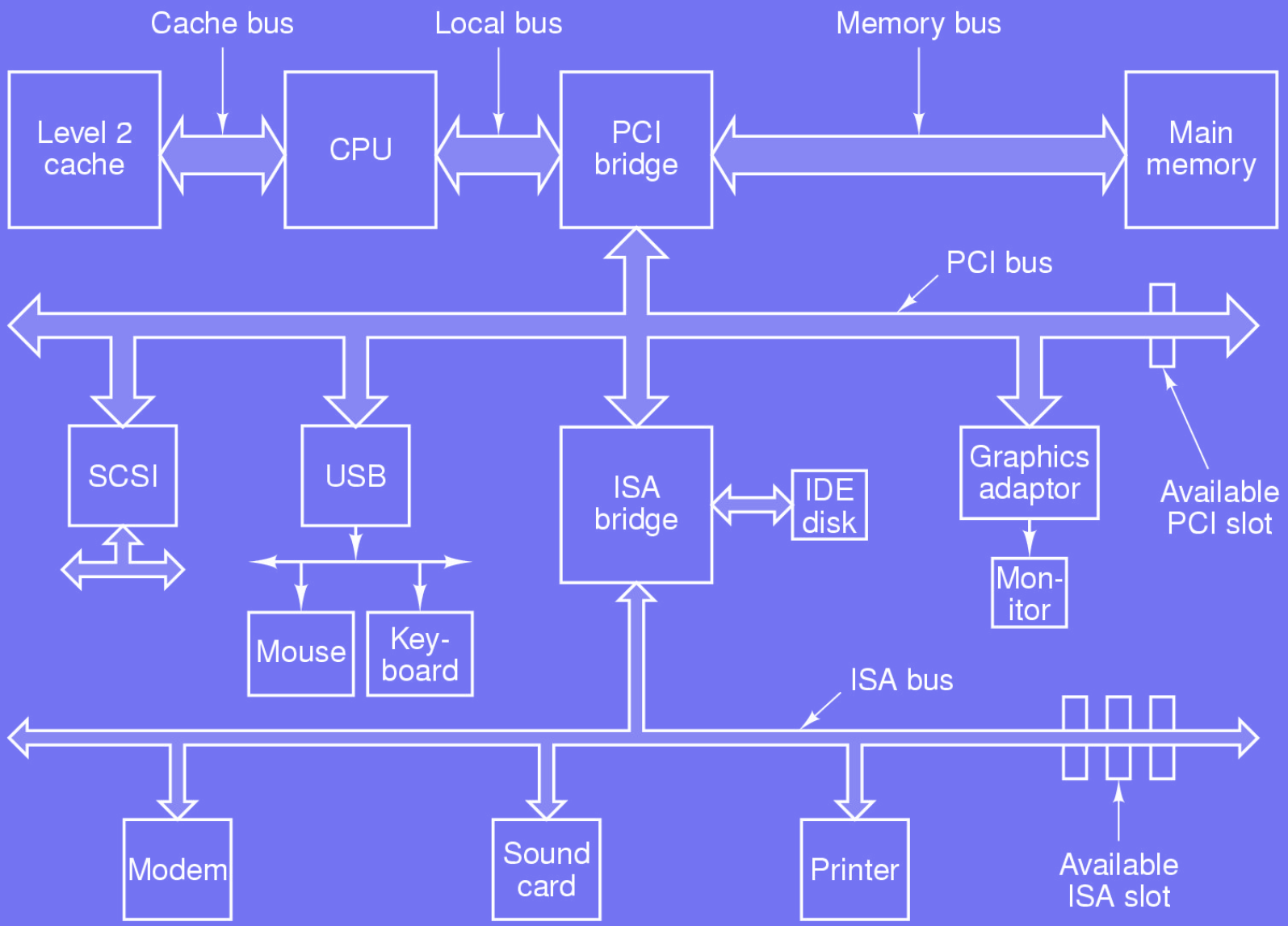
Memory

Access time		Capacity
1 ns	Registers	< 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

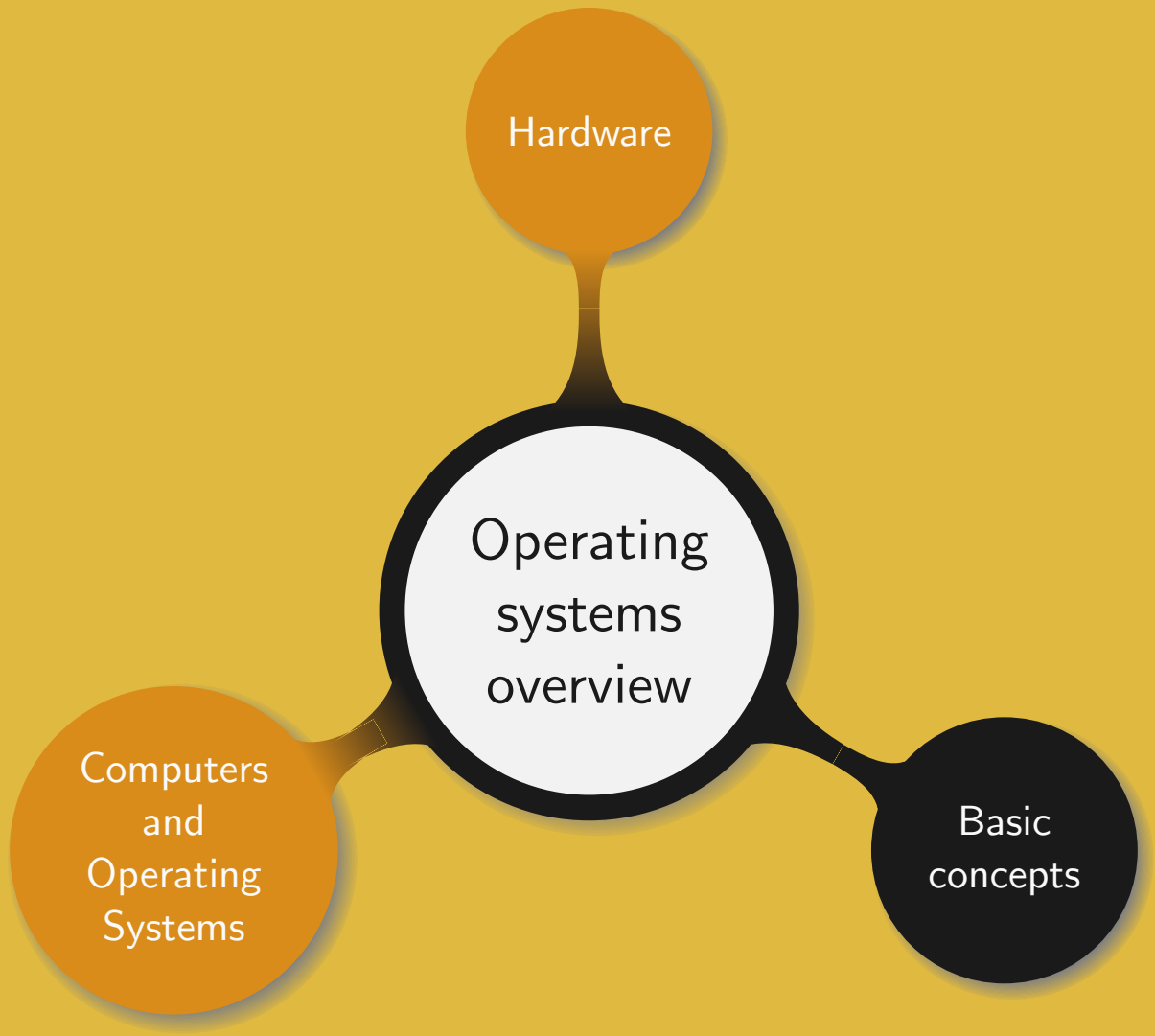
Memory types:

- 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

Bus


Cache bus
Local bus
Memory bus
Level 2 cache
CPU
PCI bridge
Main memory
PCI bus
SCSI
USB
ISA bridge
IDE disk
Graphics adaptor
Available PCI slot
Mon-itor
Mouse
Key-board
ISA bus
Modem
Sound card
Printer
Available ISA slot

Chapter organisation

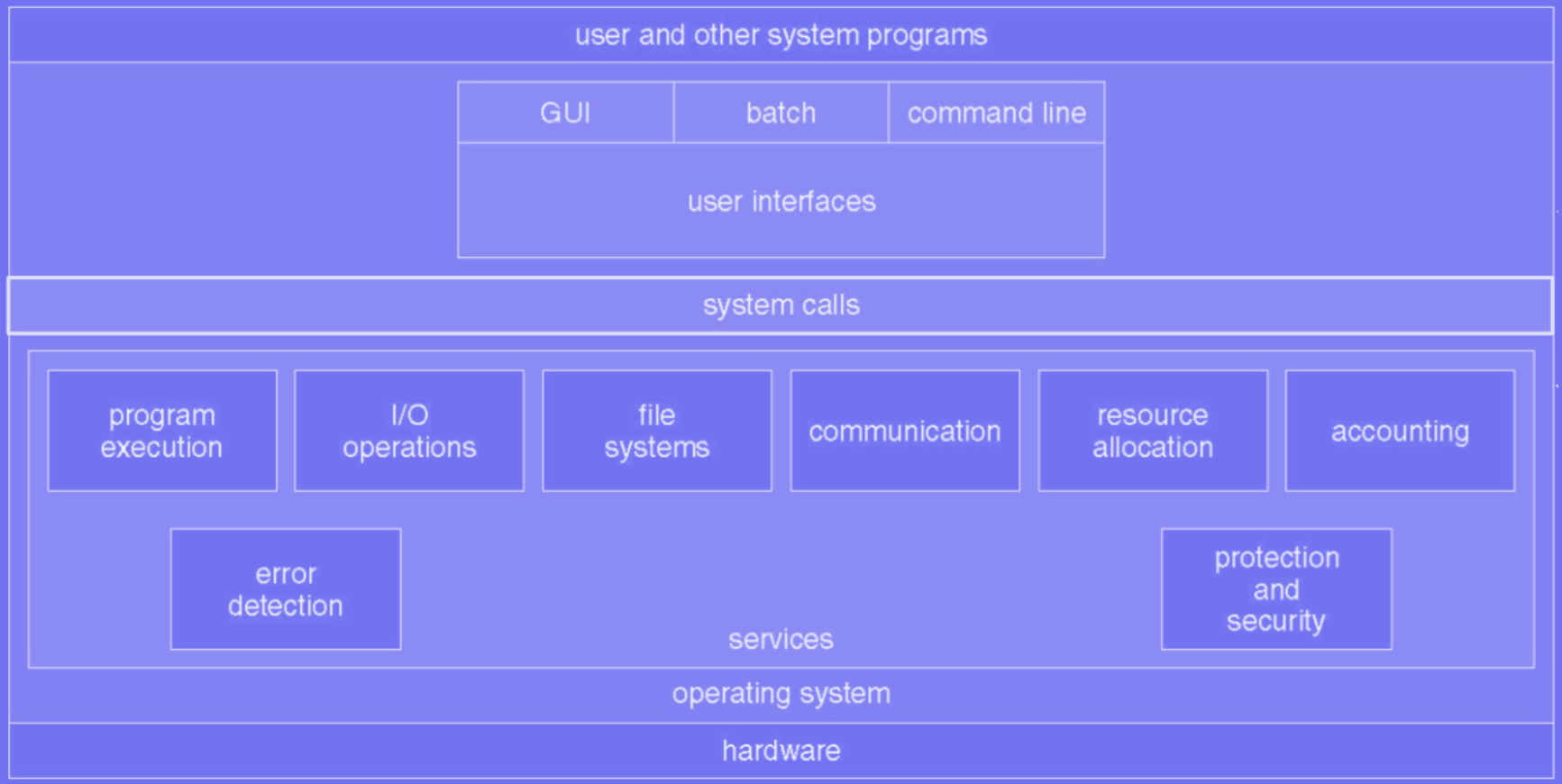


Main categories

Five major components of an OS:

- System calls: allows to interface with user-space
- Processes: defines everything needed to run programs
- File system: store persistent data
- Input-Output (IO): allows to interface with hardware
- Protection and Security: keep the system safe

System calls



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); mount(special,name,flags,types,args); umount(name); s=unlink(name); s=link(name1,name2);`
- Misc: `s=chdir(dirname); s=chmod(name,mode); sec=time(*t); s=kill(pid,signal);`

Processes

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

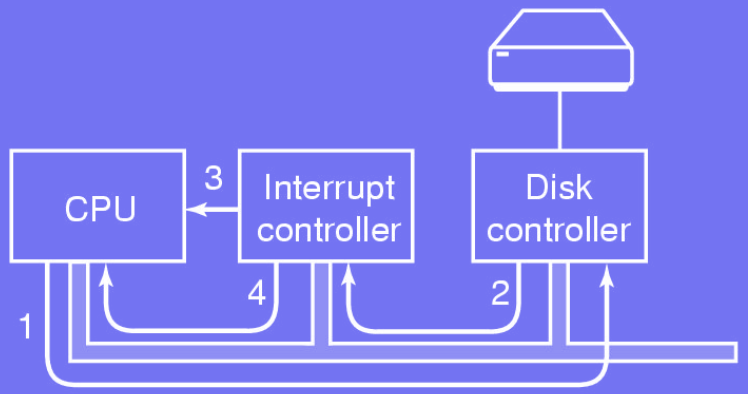
- Address space belonging to the process and containing:
  - Executable program
  - Program's data
  - Program's stack
- Set of resources:
  - Registers
  - List of open files
  - Alarms
  - List of related processes
  - Any other information required by the program

File system

The OS hides peculiarities of the disk and other IO devices

- Data stored in files grouped into directories
- The top directory is called *root* directory
- Any file can be specified using its path name
- Each process has a working directory
- Removable devices can be mounted onto the main tree
- Block files: for storage devices such as disks
- Character files: for devices accepting or outputting character streams
- Pipe: pseudo file used to connect two processes

Interrupts



Hardware interrupt:

1. Send instructions to the controller
2. The controller signals the end
3. Assert a pin to interrupt the CPU
4. Send extra information

Software interrupt:

- A call coming from userspace
- A software interrupt handler is invoked
- System call: switch to kernel mode to run privileged instruction

Operating systems are almost always interrupt driven

Busy waiting

Simplest method:

1. Call the driver
2. Start the input-output
3. Wait in a tight loop
4. Continuously poll the device to know its state

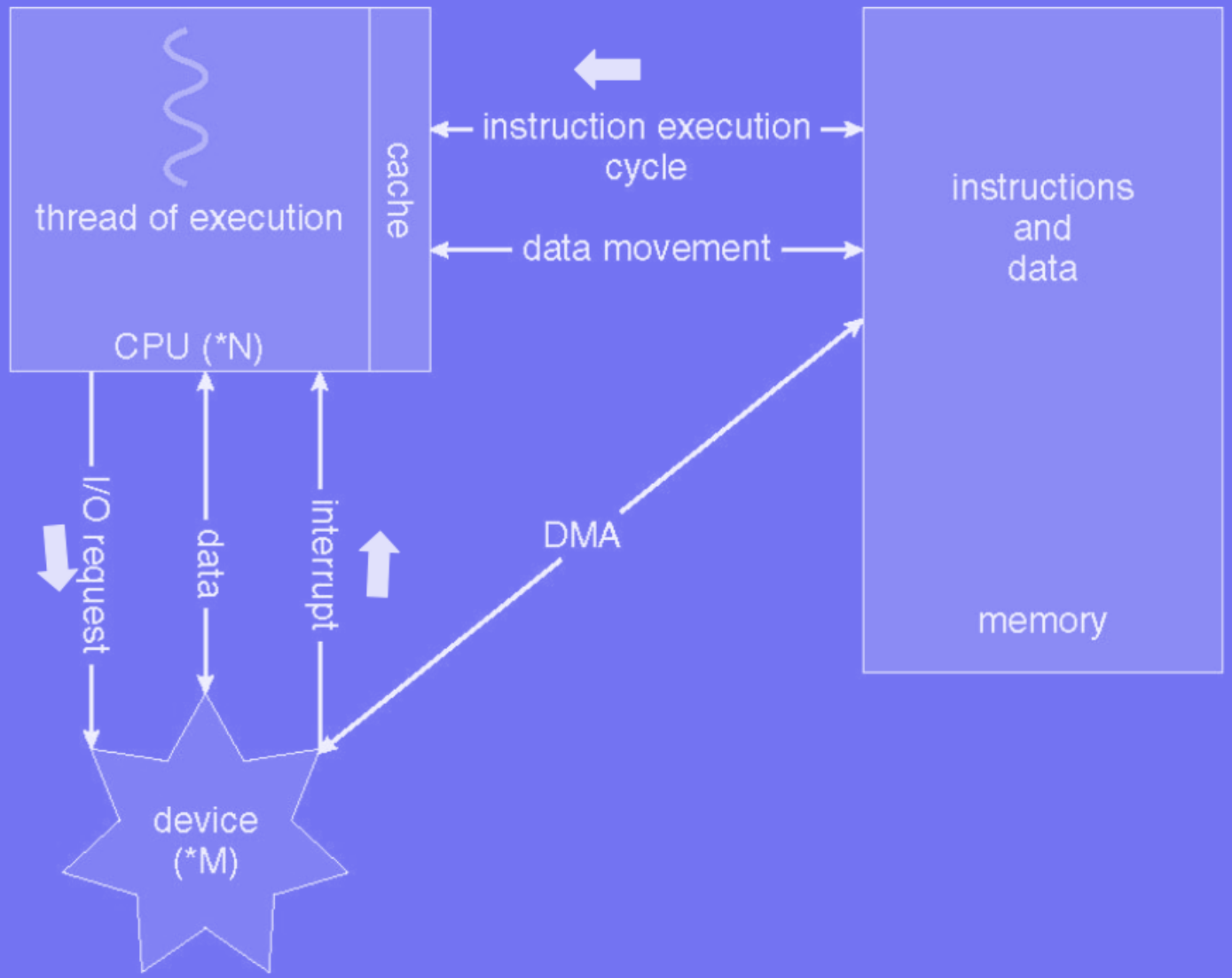
What is the drawback of this strategy?

Direct memory access

Direct Memory Access (DMA):

- Can transmit information close to memory speeds
- Directly transfer blocks of data from the controller to the RAM
- Only little needed from the CPU
- Issue a single interrupt per block, instead of one per byte

The three communication startegies



Protection and Security

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:

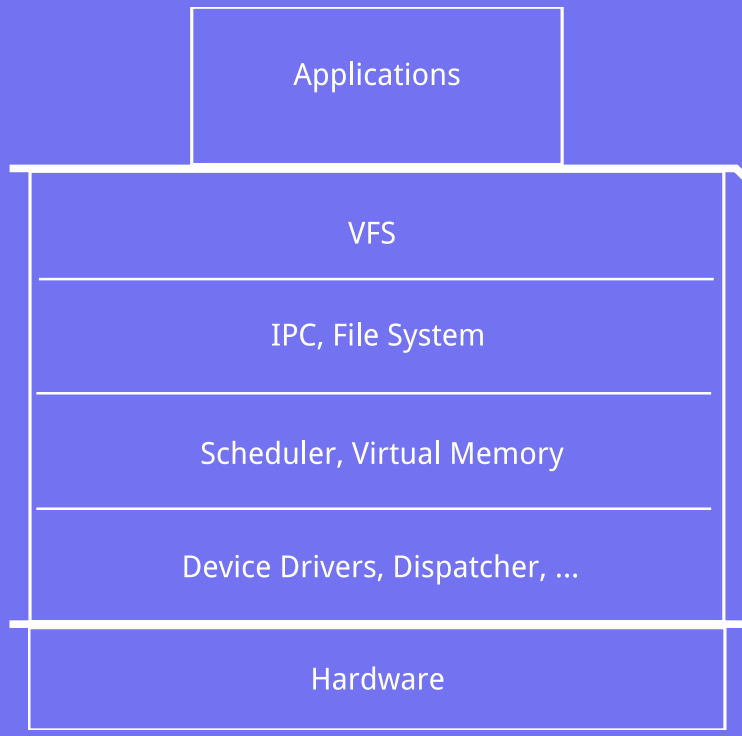
- Base and limit registers: holds the smallest legal physical memory address and the size of the range, respectively
- Memory outside the address space is protected

Input and Output:

- They are all privilege instructions
- The OS processes them to ensure their correctness and legality

Common operating system structures

Monolithic kernel



User mode

Kernel mode

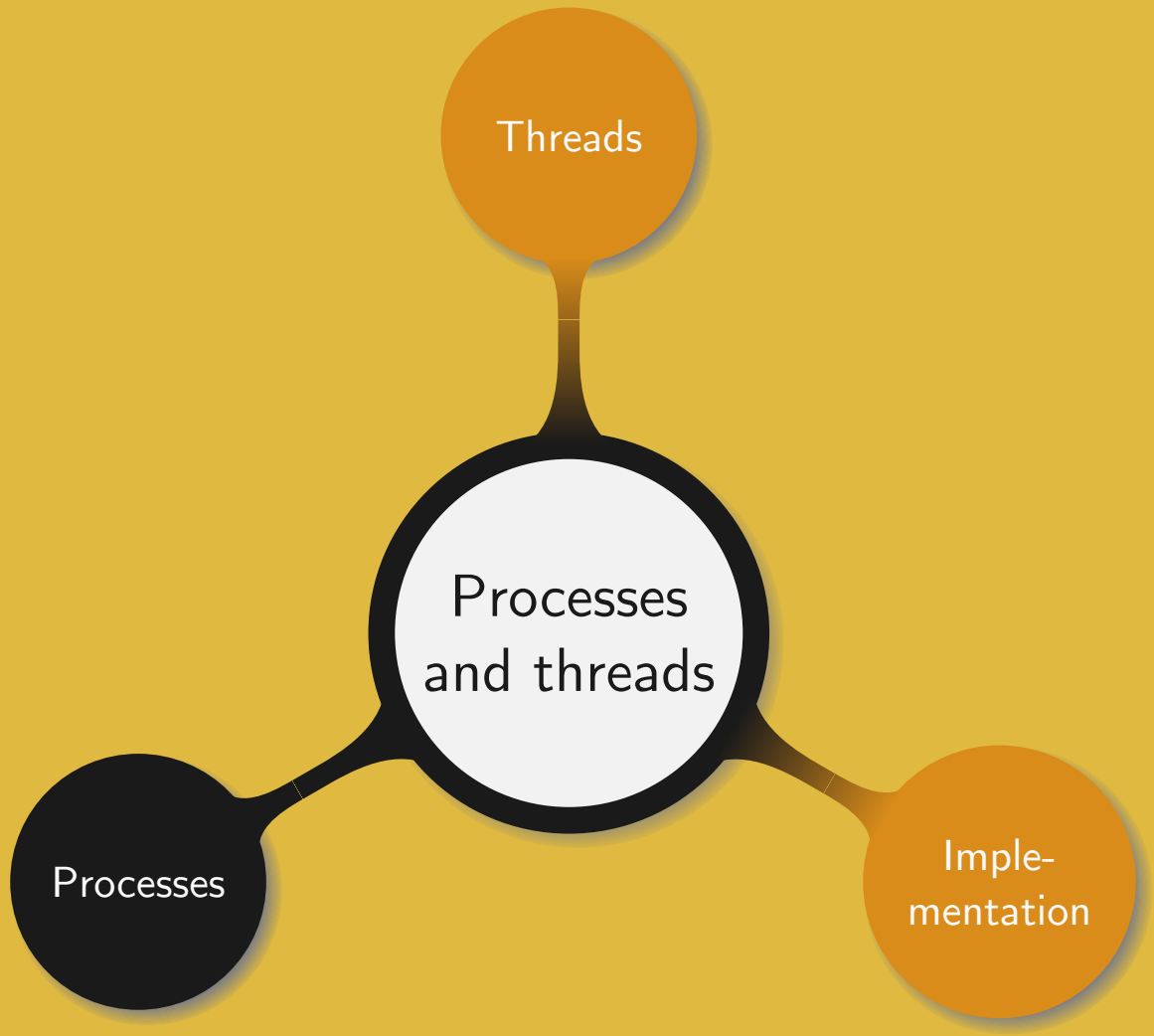
Micro kernel

Key points

- What is the main job of an OS?
- Why are there so many types of OS?
- Why is hardware important when writing an OS?
- What are the main components of an OS?
- What are system calls?

2. Processes and threads

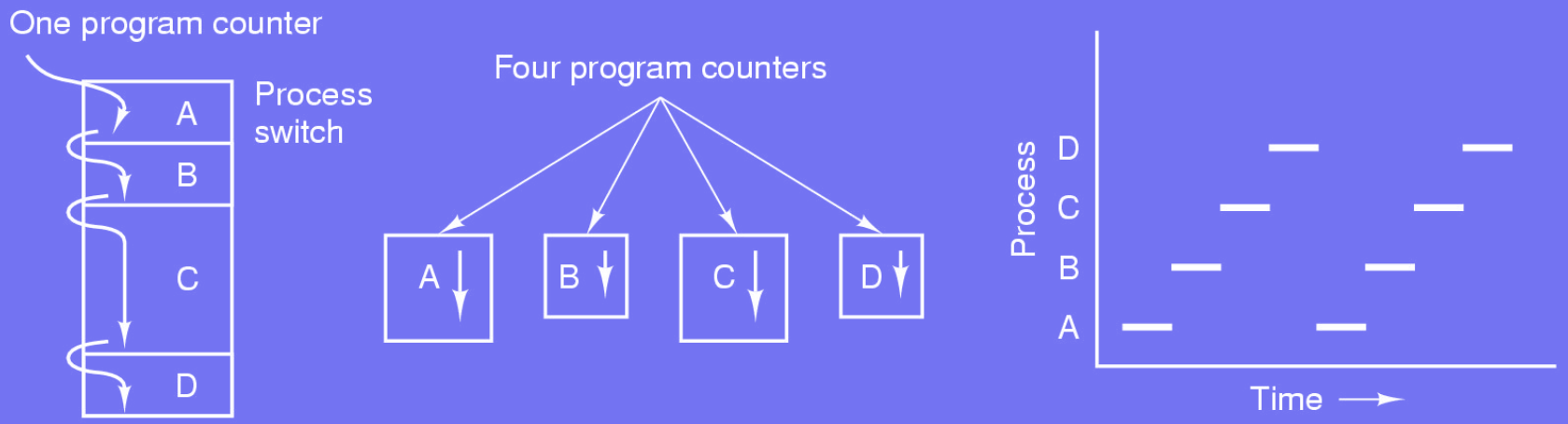
Chapter organisation



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



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

Describe the process of baking a cake when the phone rings

Process creation and termination

Times at which processes are created:

- System initialization
- Upon a user launching a new program
- Initialization of a batch job

Any created processes ends at some stage:

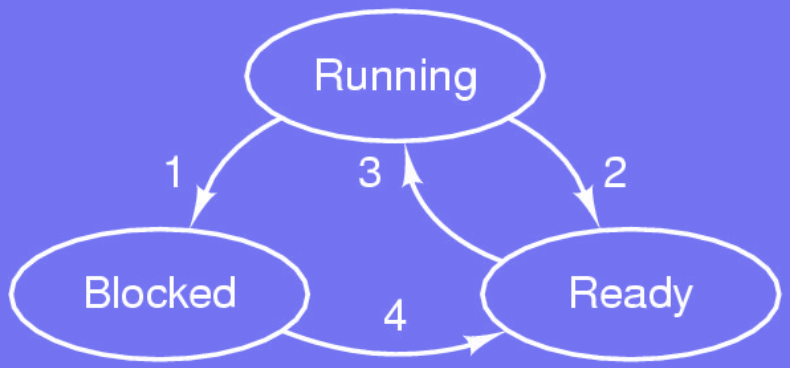
- Voluntarily:
  - The work is completed, issue a system call to inform the OS
  - An error is noticed, the process exits nicely
- Involuntary:
  - Fatal error, program crashes
  - Another process kills it

Process hierarchies

Two main approaches:

- UNIX-like systems:
  - A parent creates a child
  - A child can create its own child
  - The hierarchy is called *process group*
  - It is impossible to disinherit a child
- Windows system:
  - All processes are equal
  - A parent has a token to control its child
  - A 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

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

Modeling processes

A simple model for processes:

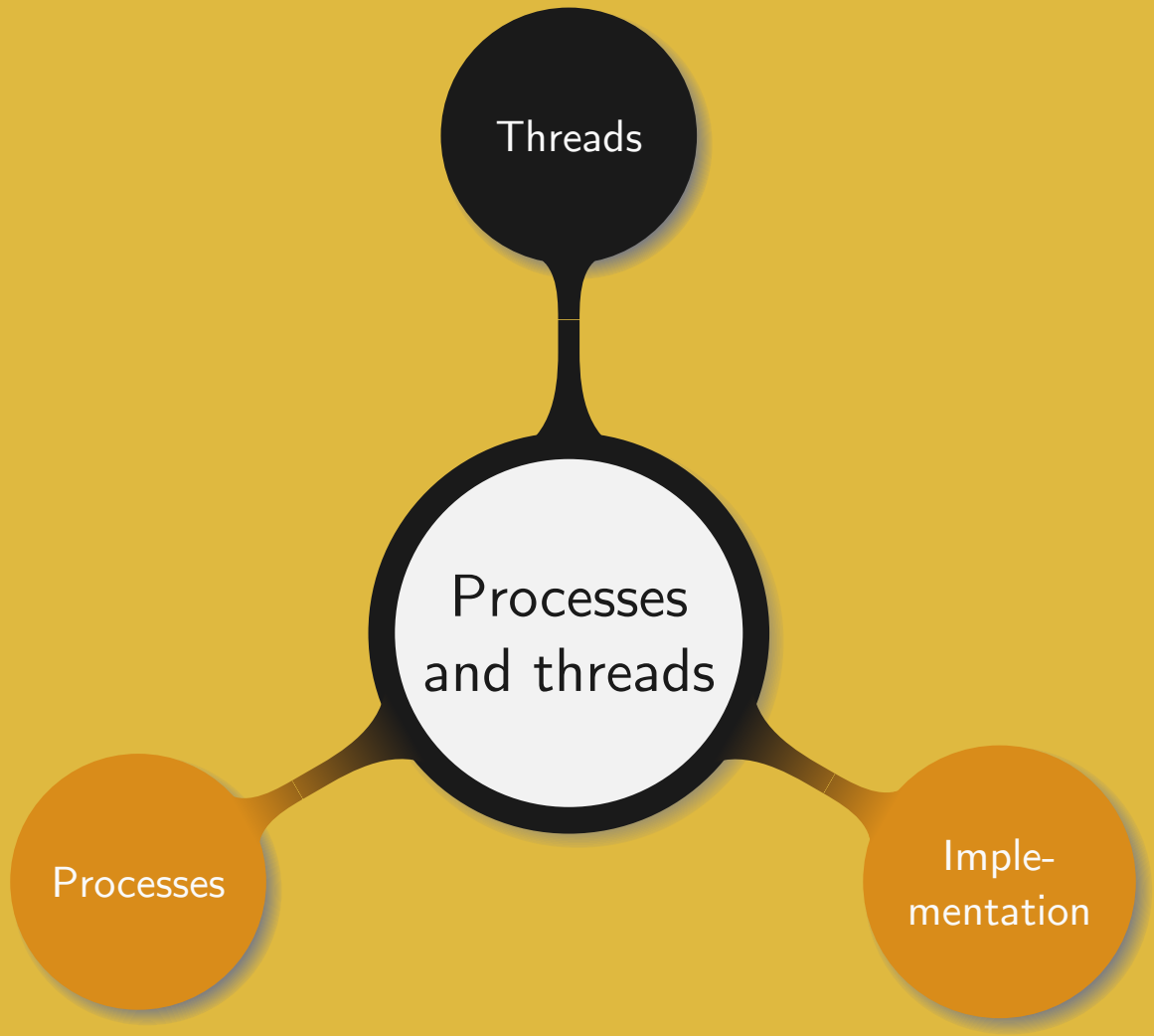
- A process is a data structure called *process control block*
- The structure contains important information such as:
  - State
  - Program counter
  - Stack pointer
  - Memory allocation
  - Open files
  - Scheduling information
- All the processes are stored in an array called *process table*

Interrupts and processes

Upon an interrupt the running process must be paused:

1. Push on the stack the user program counter, PSW, etc.
2. Load information from interrupt vector
3. Save registers (assembly)
4. Setup new stack (assembly)
5. Finish up the work for the interrupt
6. Decides which process to run next
7. Load the new process, i.e. memory map, registers, etc. (assembly)

Chapter organisation



Overview

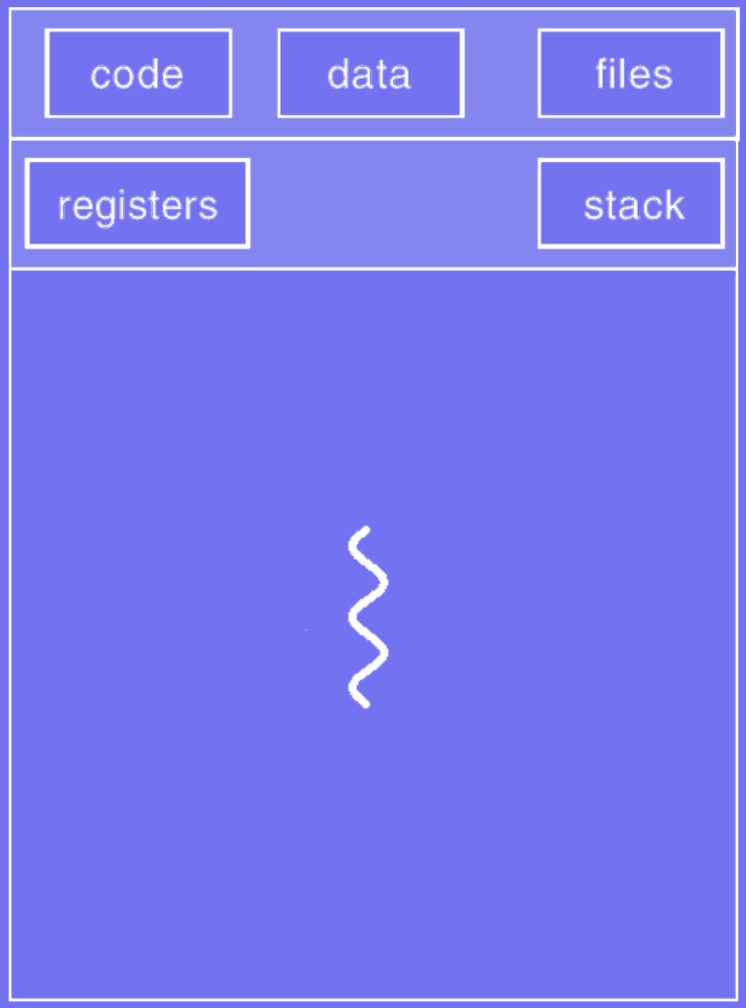
A thread is the basic unit of CPU utilisation consisting of:

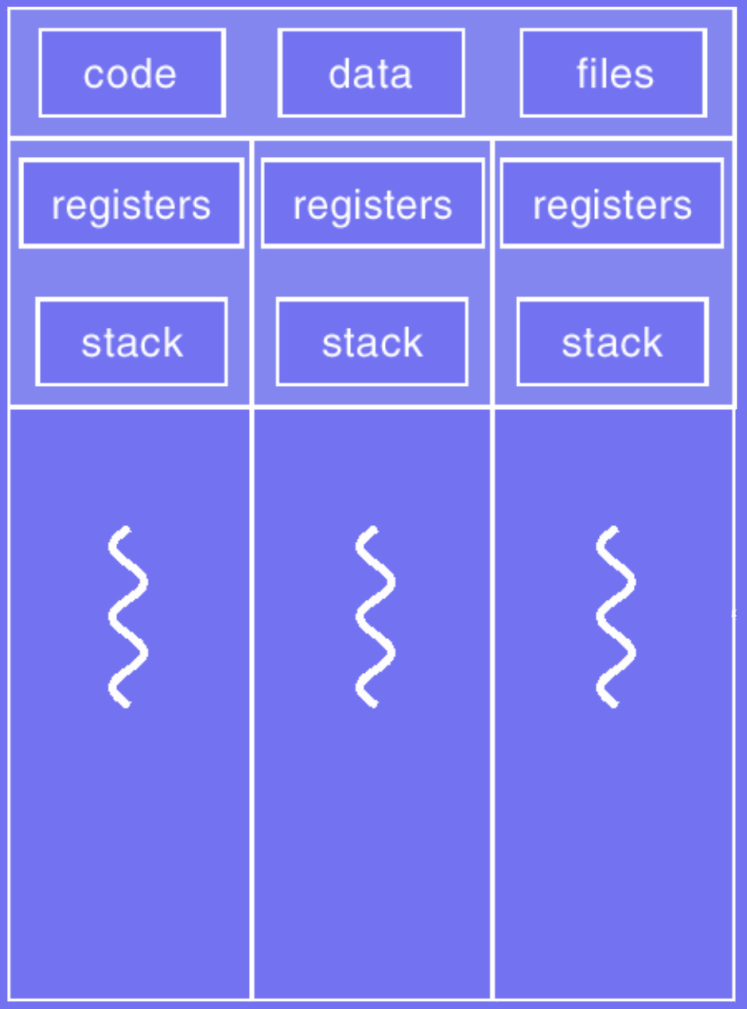
- A thread ID
- The program counter
- A register set
- A stack space

All the threads within a process share the same:

- Code section
- Data section
- Operating system resources

Single vs. multi-threaded



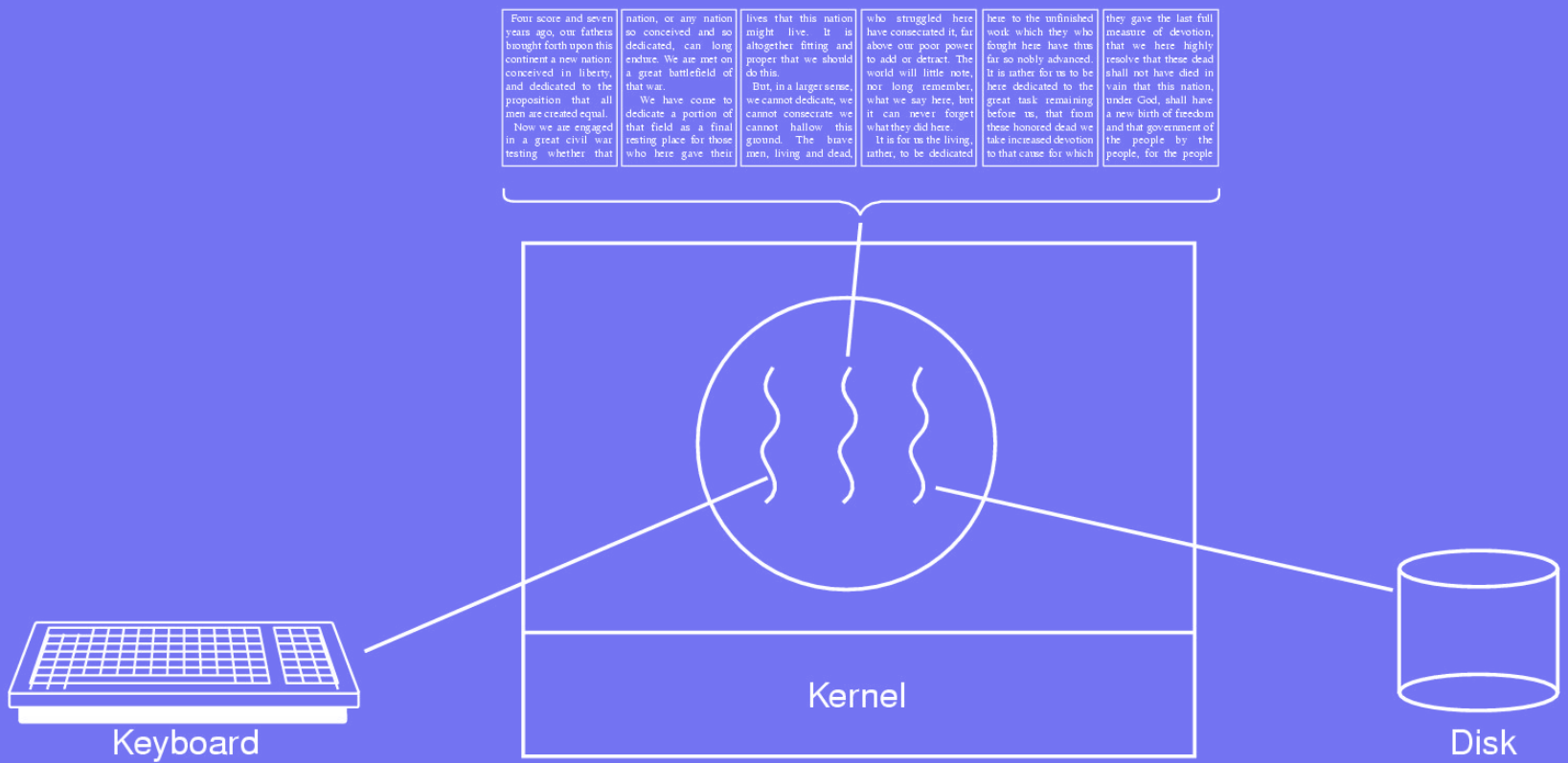


Notes and remarks

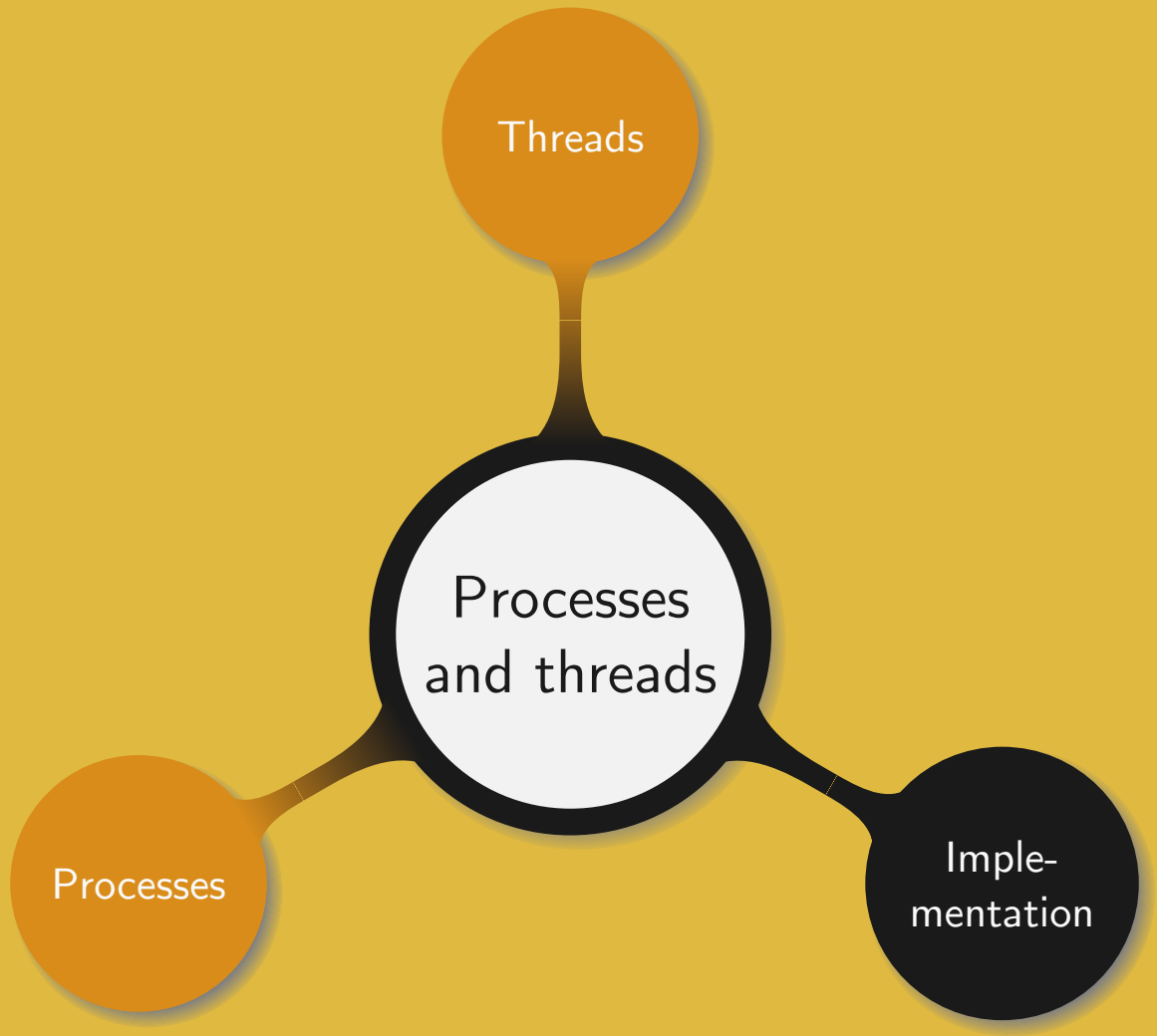
Processes and threads:

- 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 of a word processor



Chapter organisation



POSIX threads

The `pthread` library has over 60 function calls:

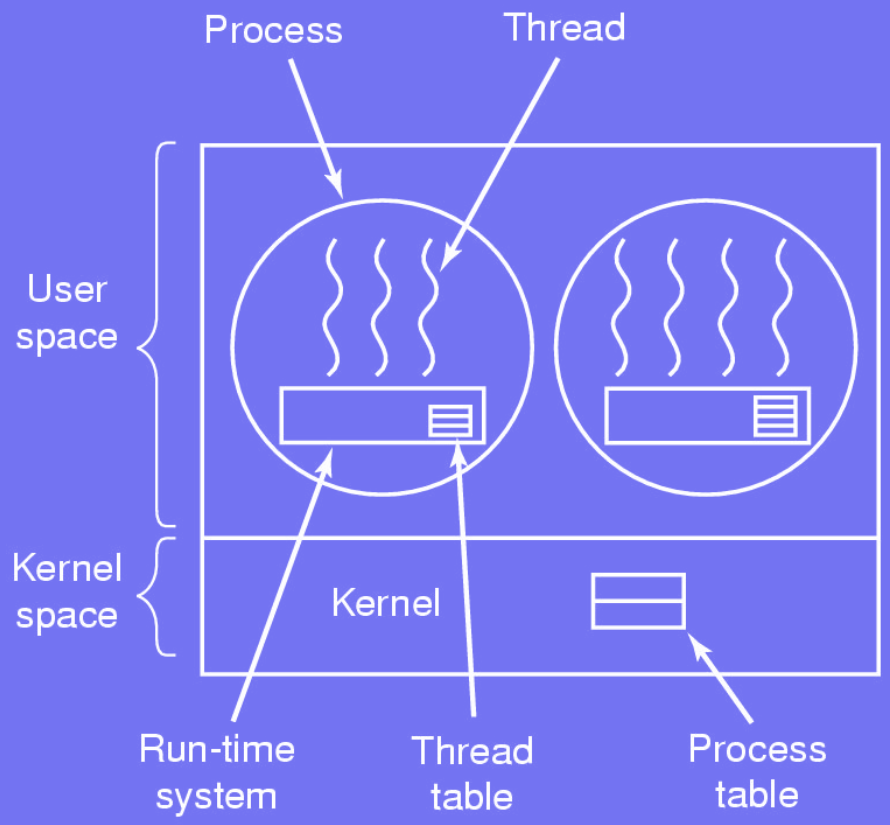
- Create a thread: `int pthread_create(pthread_t *thread, const pthread_attr_t *attr, void *(*start_routine) (void *), void *arg);`
- Terminate a thread: `void pthread_exit(void *retval);`
- Wait for a specific thread to end: `int pthread_join(pthread_t thread, void **retval);`
- Release CPU to let another 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);`

Creating ten threads and printing their ID

threads.c

```c
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#define THREADS 10
void *gm(void *tid) {
  printf("Good morning from thread %lu\n",*(unsigned long int*)tid);
  pthread_exit(NULL);
}
int main () {
  int status, i; pthread_t threads[THREADS];
  for(i=0;i< THREADS;i++) {
    printf("thread %d\n",i);
    status=pthread_create(&threads[i],NULL,gm,(void*)&(threads[i]));
    if(status!=0) {
      fprintf(stderr,"thread %d failed with error %d\n",i,status);
      exit(-1);
    }
  }
  exit(0);
}
```

Threads in user-space – N:1



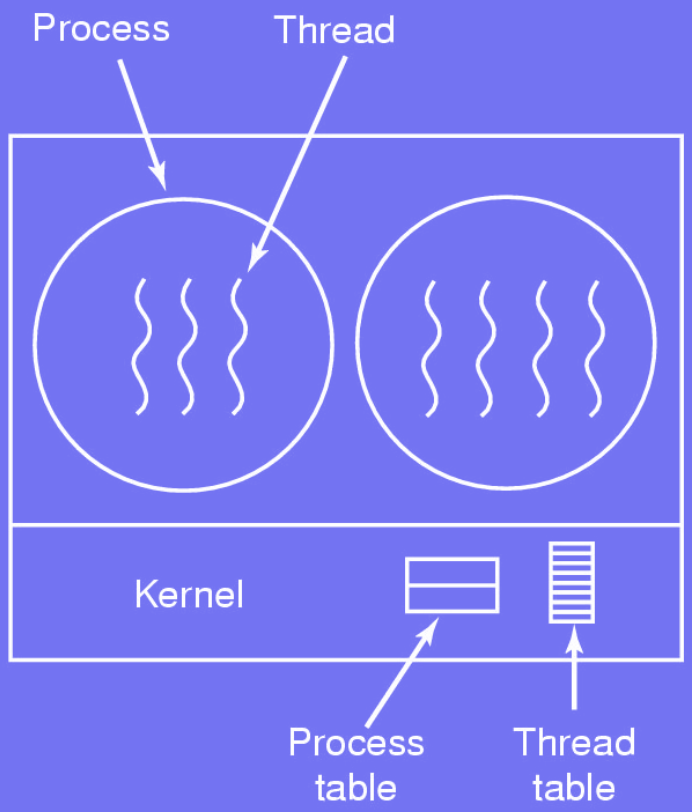
User-space threads:

- 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



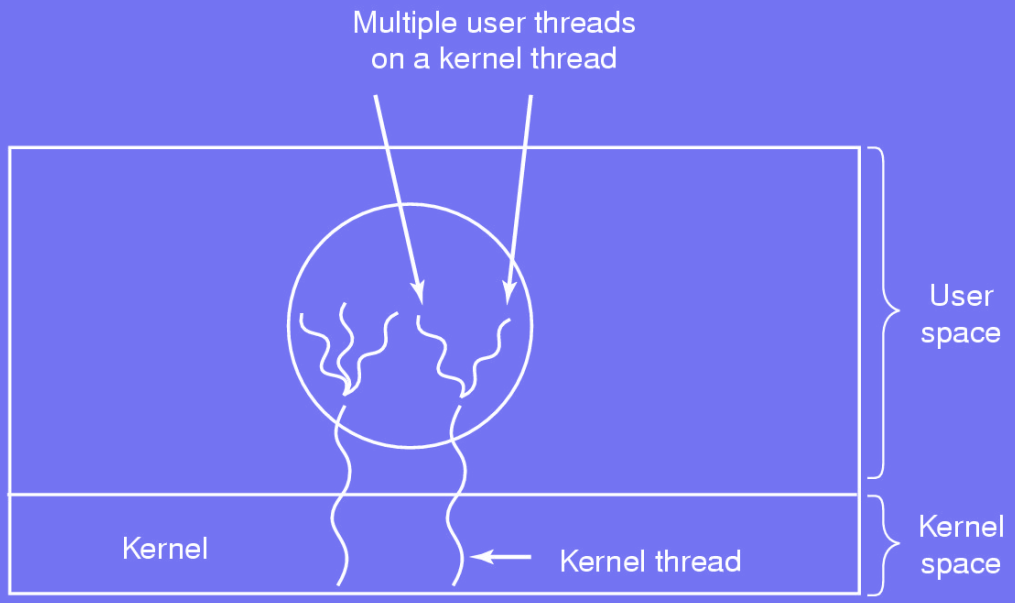
Kernel space thread:

- 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 received 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?

Notes and remarks

Best thread approach:

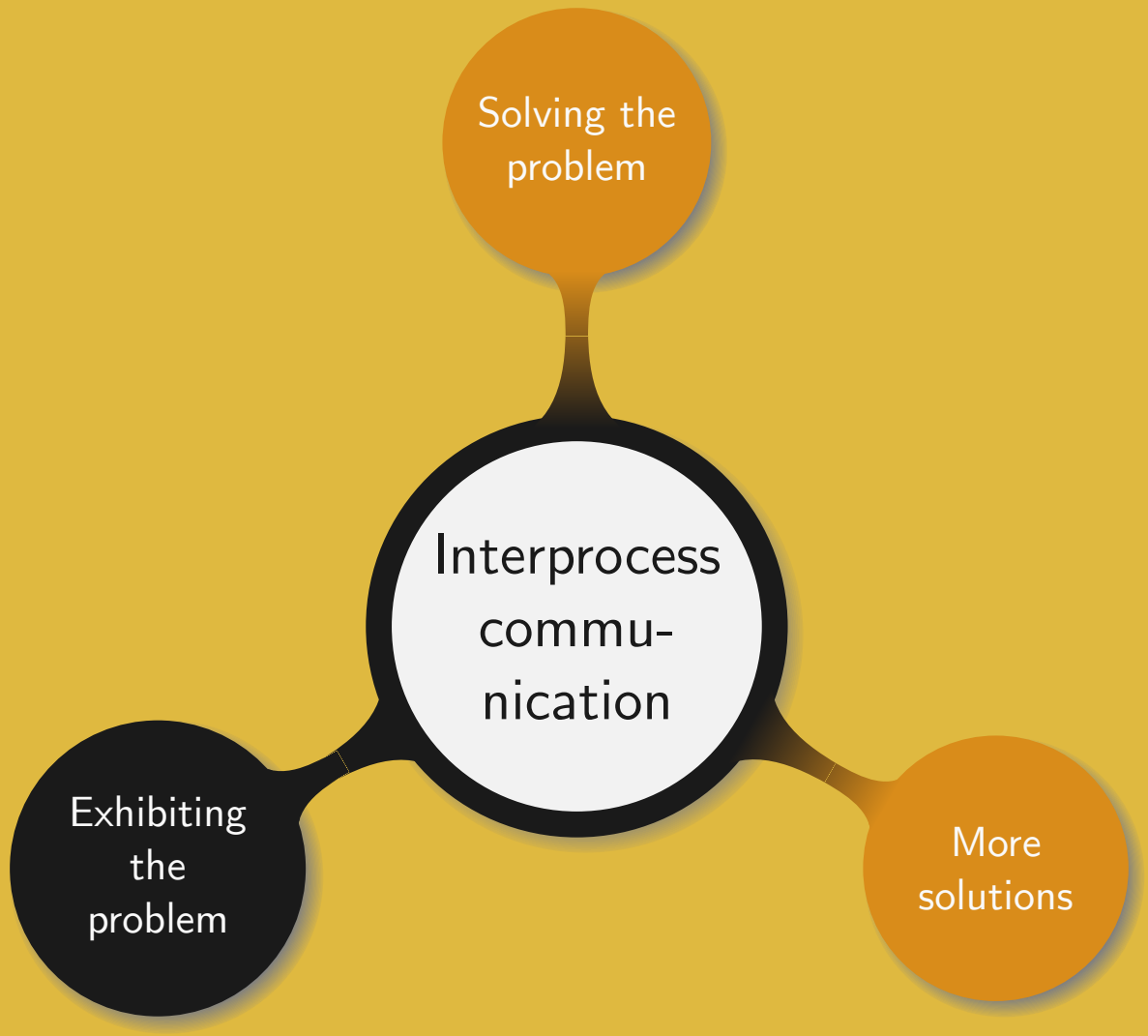
- Hybrid looked attractive
- Most systems are coming back to 1:1
- Different approaches exist on how to use threads
  e.g. thread bocks on “receive system call” vs. pop up threads
- Switching implementation from single thread to multiple thread is not easy task
- Requires redesigning the whole system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

Key points

- What is a process?
- How can processes be created and terminated?
- What are the possible states of a process?
- What is the difference between single thread and multi-threads?
- What approaches can be taken to handle threads?

3. Interprocess communication

Chapter organisation



Working with a single thread

In single tasking all threads are independent:

- They cannot affect or be affected by anything
- Their state is not shared with other threads
- The input state determines the output
- Everything is reproducible
- Stopping or resuming does not lead to any side effect

It suffices to run a thread to completion and start the next one

Working with multiple threads

Difficulties appear with multi-tasking:

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

Changes made by one thread can affect others

Thread and process cooperation

Setup for threads:

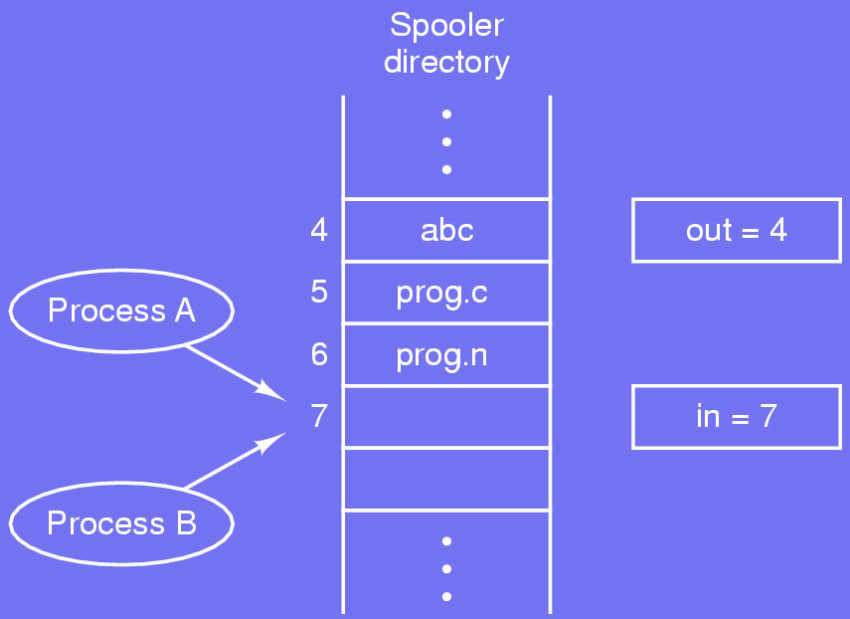
- Several threads share a common global variable
- The execution sequence impacts the global variable
- By default the behavior is random and irreproducible

Major problems:

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

All those thread issues within a process can be extended to processes within the operating system

Race conditions



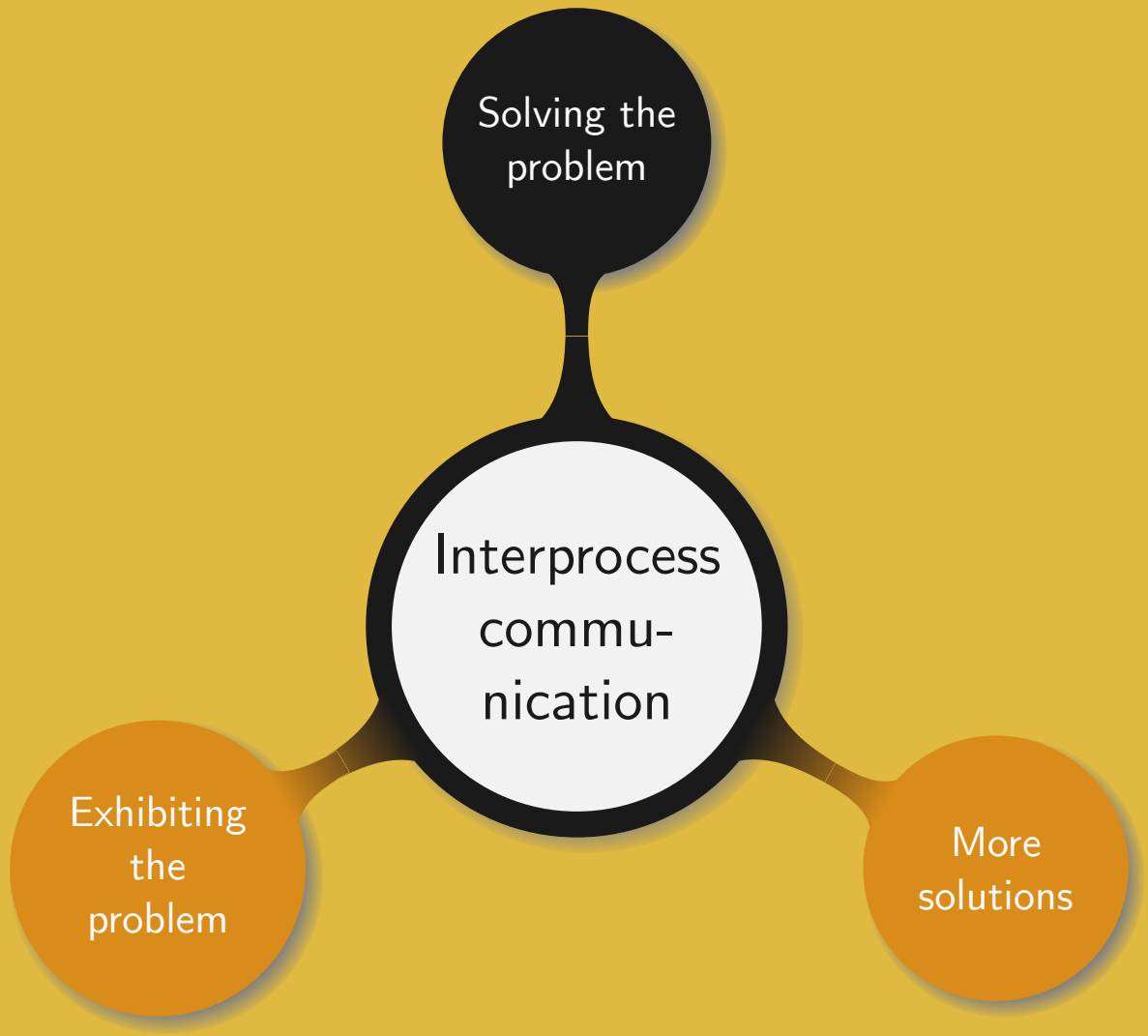
In the printing spool:

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

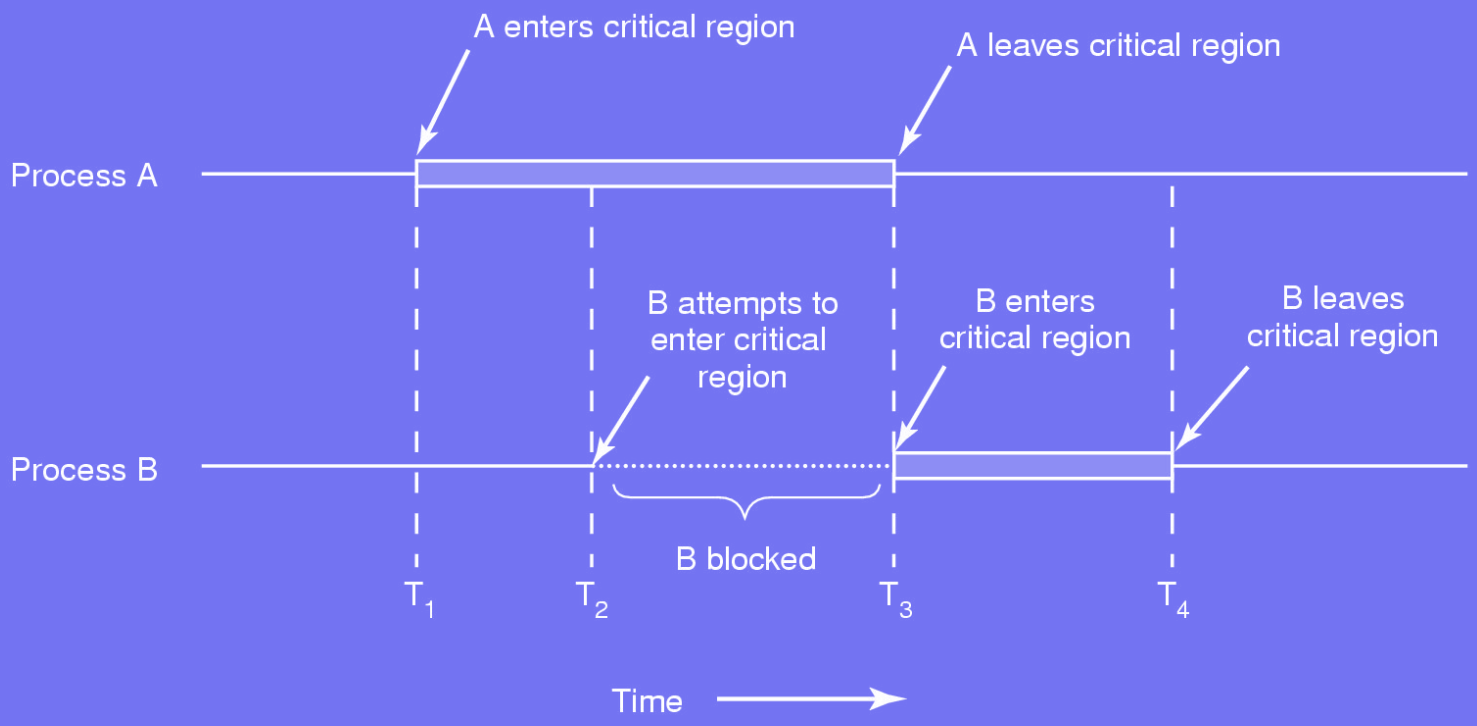
Too much milk

9:00 am 9:15 am 9:30 am 9:45 am

Chapter organisation



Critical region



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

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?

Peterson's idea

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?

Peterson's idea

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 }
```

Peterson's idea

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
```

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

Semaphore

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

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 IO 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

Mutexes

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    JZ ok
5    CALL thread_yield
6    JMP mutex-lock
7  ok: RET
8
9  mutex-unlock:
10   MOVE MUTEX,#0
11   RET
```

Questions:

- What differences were introduced compared to `enter_region` (3.88)?
- 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

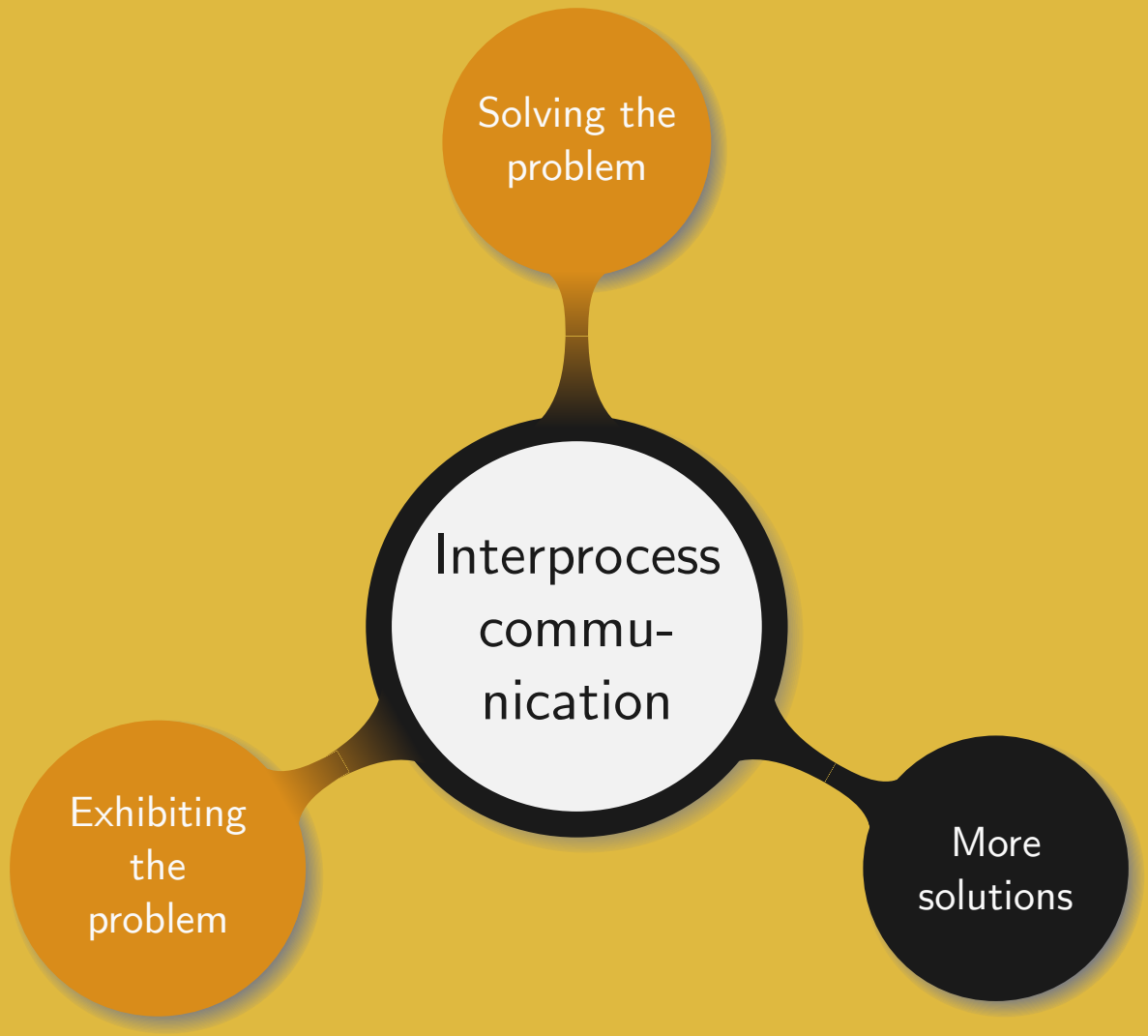
```c
#include <stdio.h>
#include <pthread.h>
#define MAX 1000
pthread_mutex_t m; pthread_cond_t cc, cp; int buf=0;
void *prod() {
  for(int i=1;i<MAX;i++) {
    pthread_mutex_lock(&m); while(buf!=0) pthread_cond_wait(&cp,&m);
    buf=1; pthread_cond_signal(&cc); pthread_mutex_unlock(&m);
  }
  pthread_exit(0);
}
void *cons() {
  for(int i=1;i<MAX;i++) {
    pthread_mutex_lock(&m);  while(buf==0) pthread_cond_wait(&cc,&m);
    buf=0; pthread_cond_signal(&cp); pthread_mutex_unlock(&m);
  }
  pthread_exit(0);
}
int main() {
  pthread_t p, c;
  pthread_mutex_init(&m,0);  pthread_cond_init(&cc,0); pthread_cond_init(&cp,0);
  pthread_create(&c,0,cons,0); pthread_create(&p,0,prod,0);
  pthread_join(p,0); pthread_join(c,0);
  pthread_cond_destroy(&cc); pthread_cond_destroy(&cp); pthread_mutex_destroy(&m);
}
```

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

Monitors are an attempt to merge synchronization with OOP

Basic idea behind monitors:

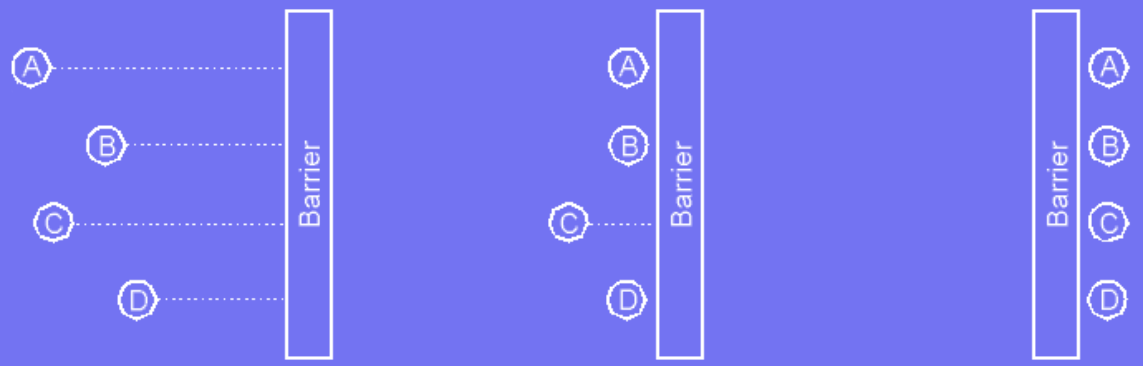
- Programming concept that must be known by the compiler
- The mutual exclusion is not handled by the programmer
- Locking occurs automatically
- Only one process can be active within a monitor at a time
- A monitor can be seen as a "special type of class"
- Processes can be blocked and awaken based 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 }
```

Barriers



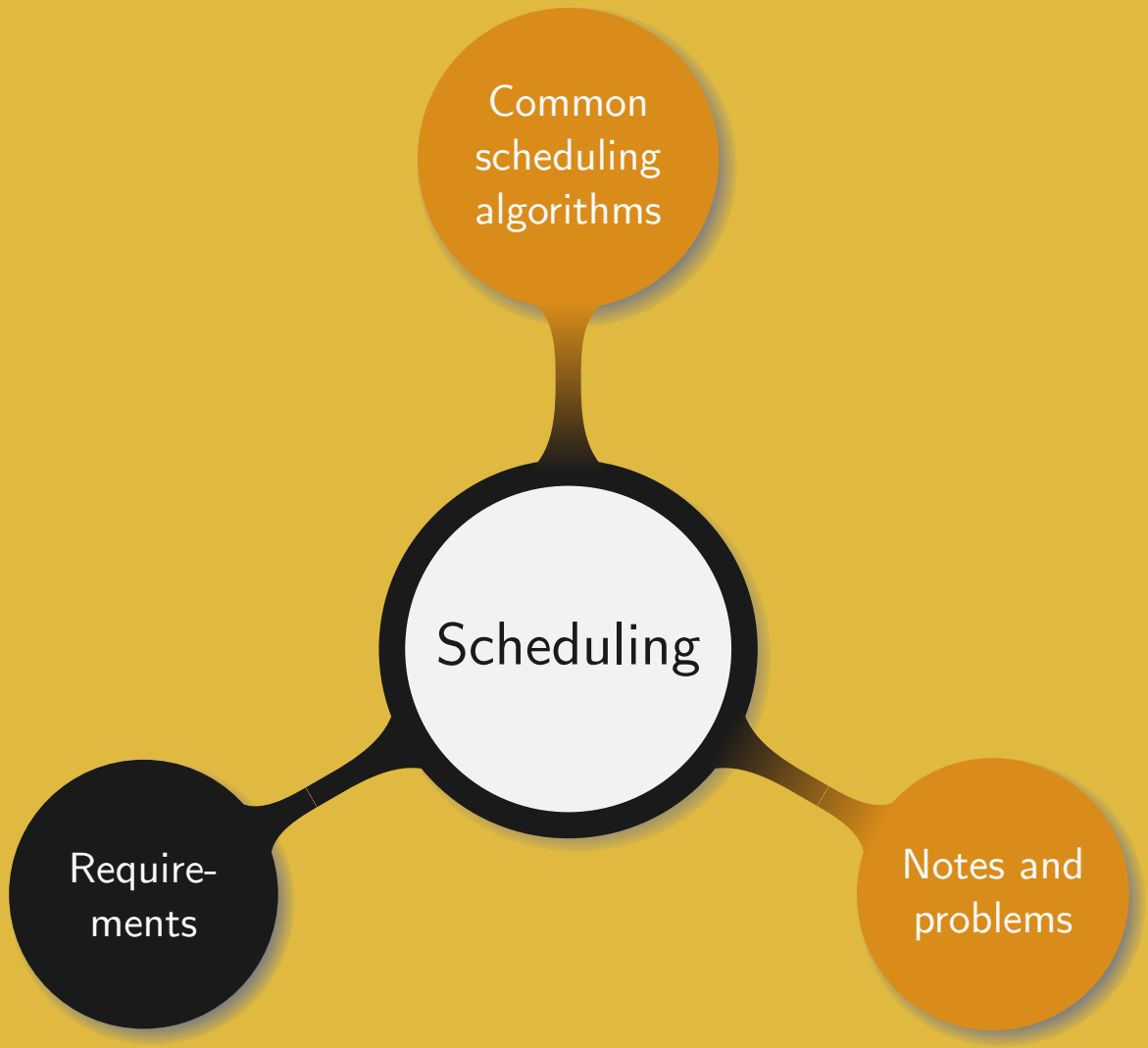
Useful when several processes must complete before the next phase

Key points

- Why is thread communication essential?
- What is a critical region?
- Do software solutions exist?
- What is an atomic operation?
- What are the two best and most common solutions?

4. Scheduling

Chapter organisation



Initial problem

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

When to decide what process to run next:

- A new process is created
- A process exits or blocks
- IO interrupt from a device that has completed its task

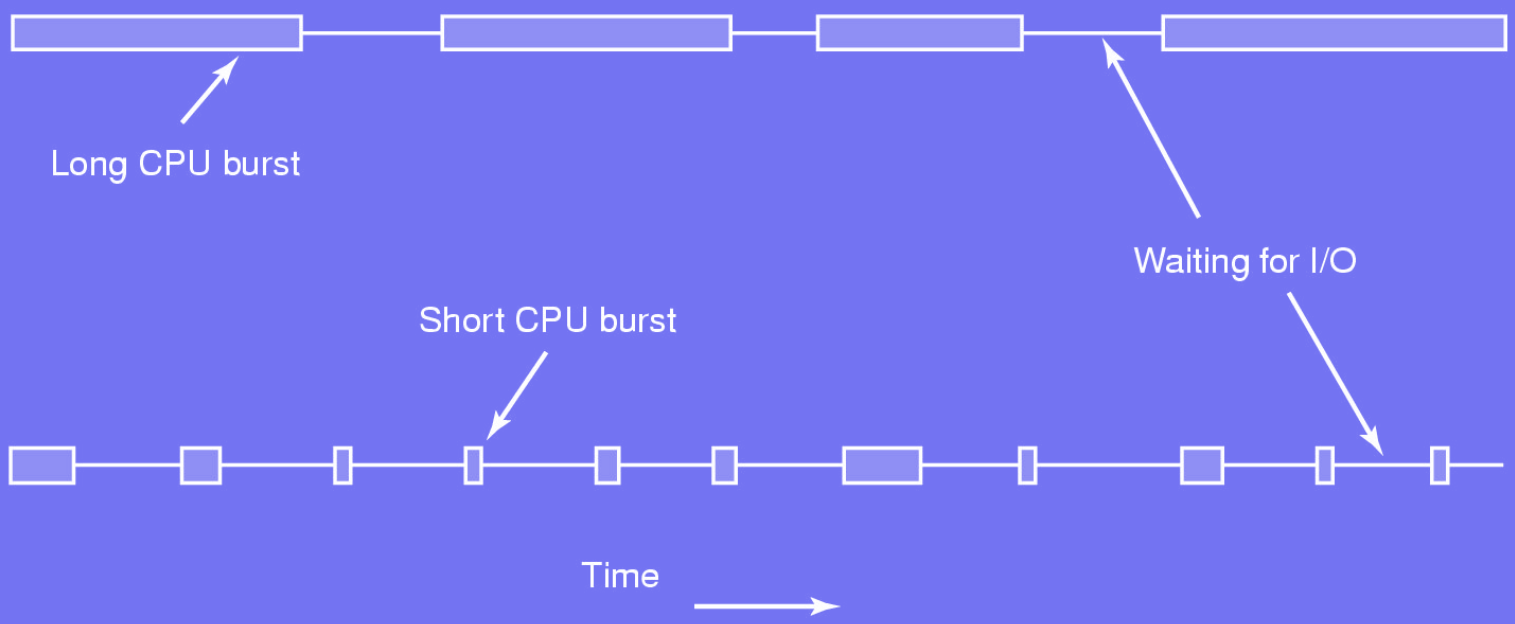
Scheduler's job

Switching process is expensive:

- Switch from user mode to kernel mode
- Save state of current process (save register, memory map, etc.)
- Run scheduling algorithm to select a new process
- Remap the memory address for the new process
- 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

Process behavior and scheduling times

Compute bound vs. input-output bound:

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

How to decide when it is best to run a process?

Preemptive vs. non-preemptive

Two main strategies for scheduling algorithms:

- Preemptive:
  - A process is run for at most n ms
  - If it is not completed by the end of the period then it is suspended
  - Another process is selected and run
- 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

Which strategy is best and what is needed to use it?

Goals when scheduling

All systems:

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

Interactive systems:

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

Goals when scheduling

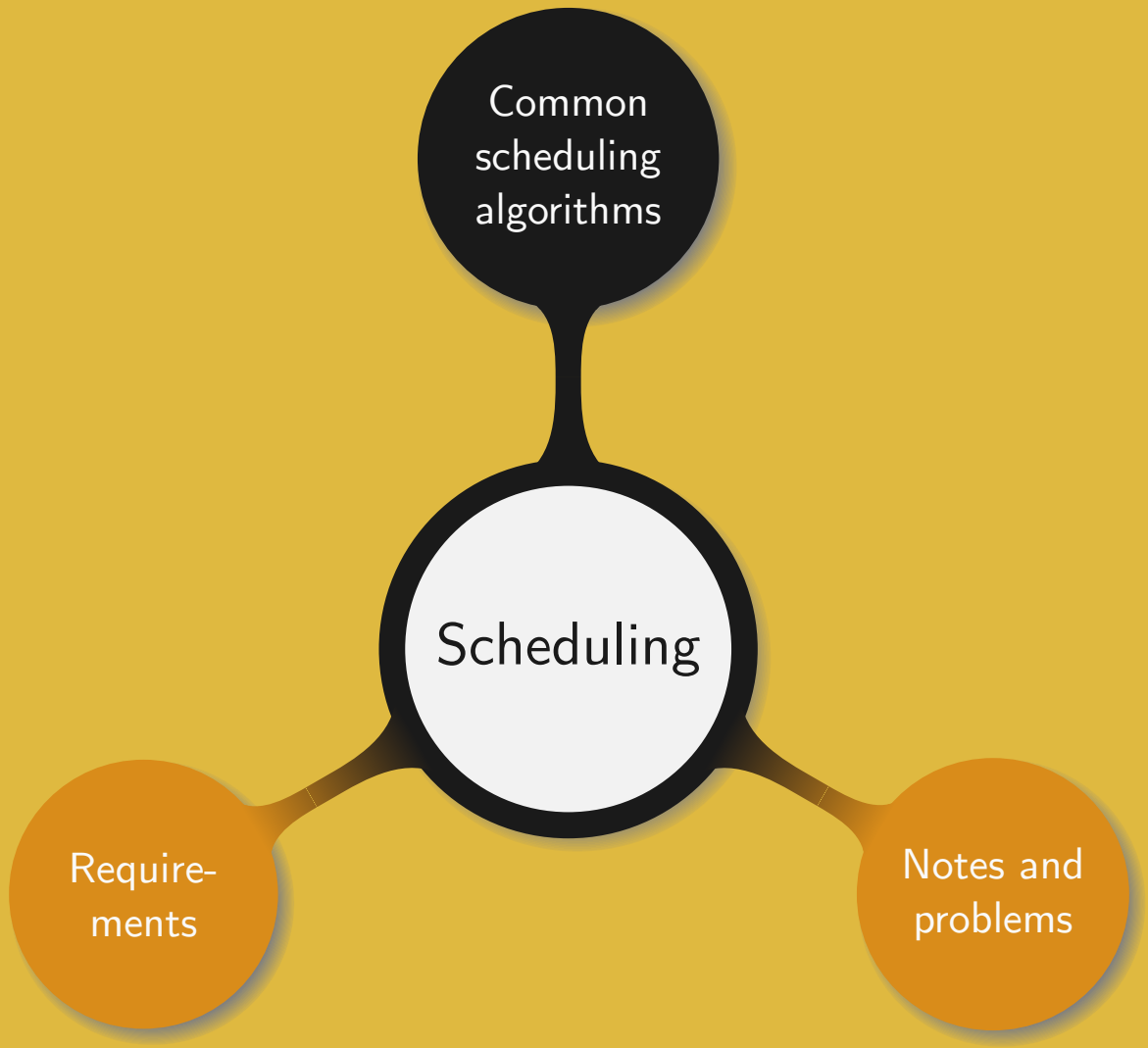
Batch systems:

- 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

Real-time systems:

- Meet deadlines: avoid any data loss
- Predictability: avoid quality degradation, e.g. for multimedia

Chapter organisation



First-come, first-served

Simplest algorithm but 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
- Any blocked process becoming ready is pushed to the queue

When is this algorithm appropriate and when should it be avoided?

Shortest job first

Non-preemptive algorithm with all run times known in advance:

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

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

When is this algorithm appropriate and when should it be avoided?

Round-Robin scheduling

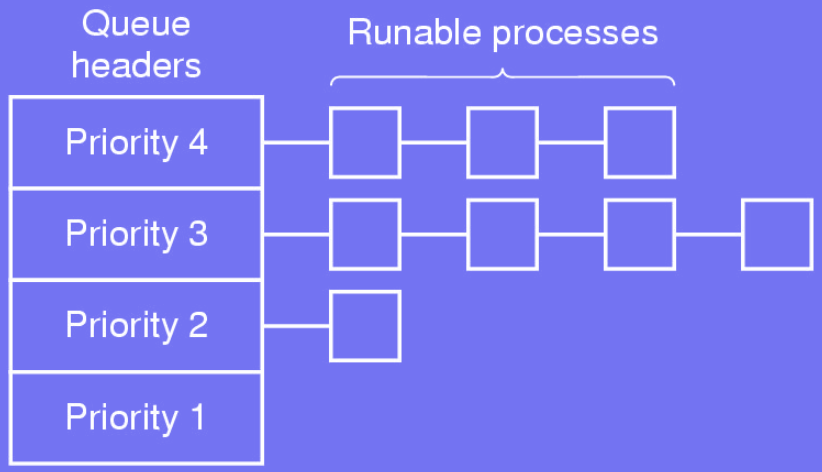
Preemptive, simple, fair, and most widely used algorithm:

- Each process is assigned a time interval called *quantum*
- A process runs until:
  - Getting blocked
  - being completed
  - Its quantum has elapsed
- A process switch occurs

When is this algorithm appropriate and when should it be avoided?

Priority scheduling

Preemptive algorithm allowing to define priorities based on who or what:



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

When is this algorithm appropriate and when should it be avoided?

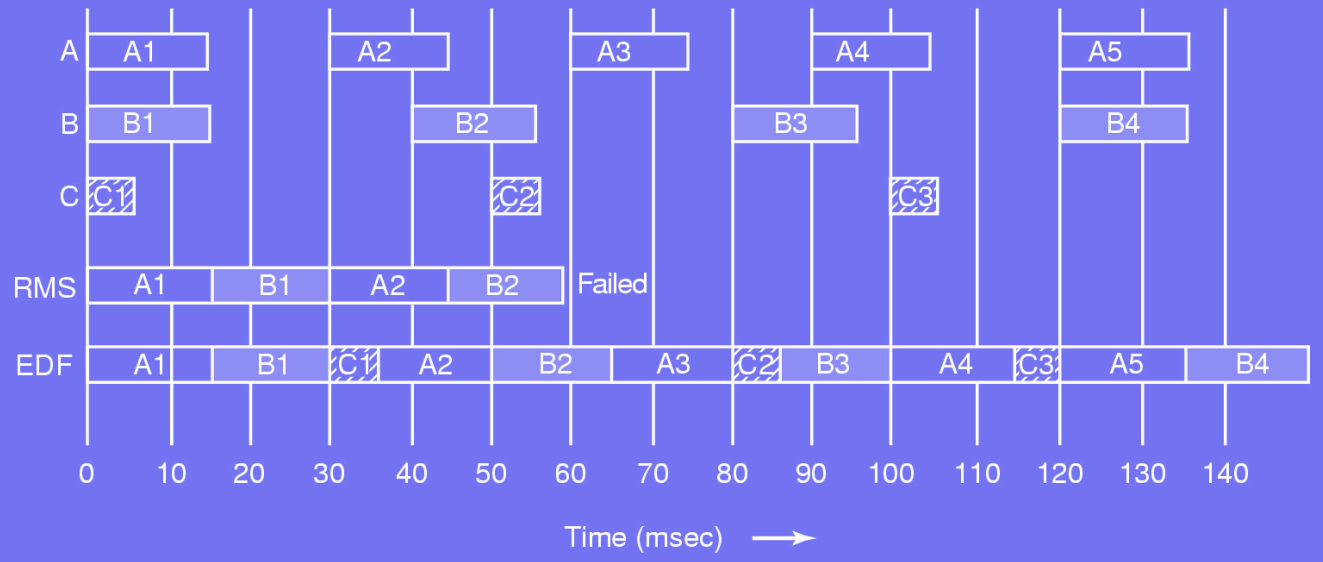
Lottery scheduling

Preemptive algorithm which can extend 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

When is this algorithm appropriate and when should it be avoided?

Earliest deadline first

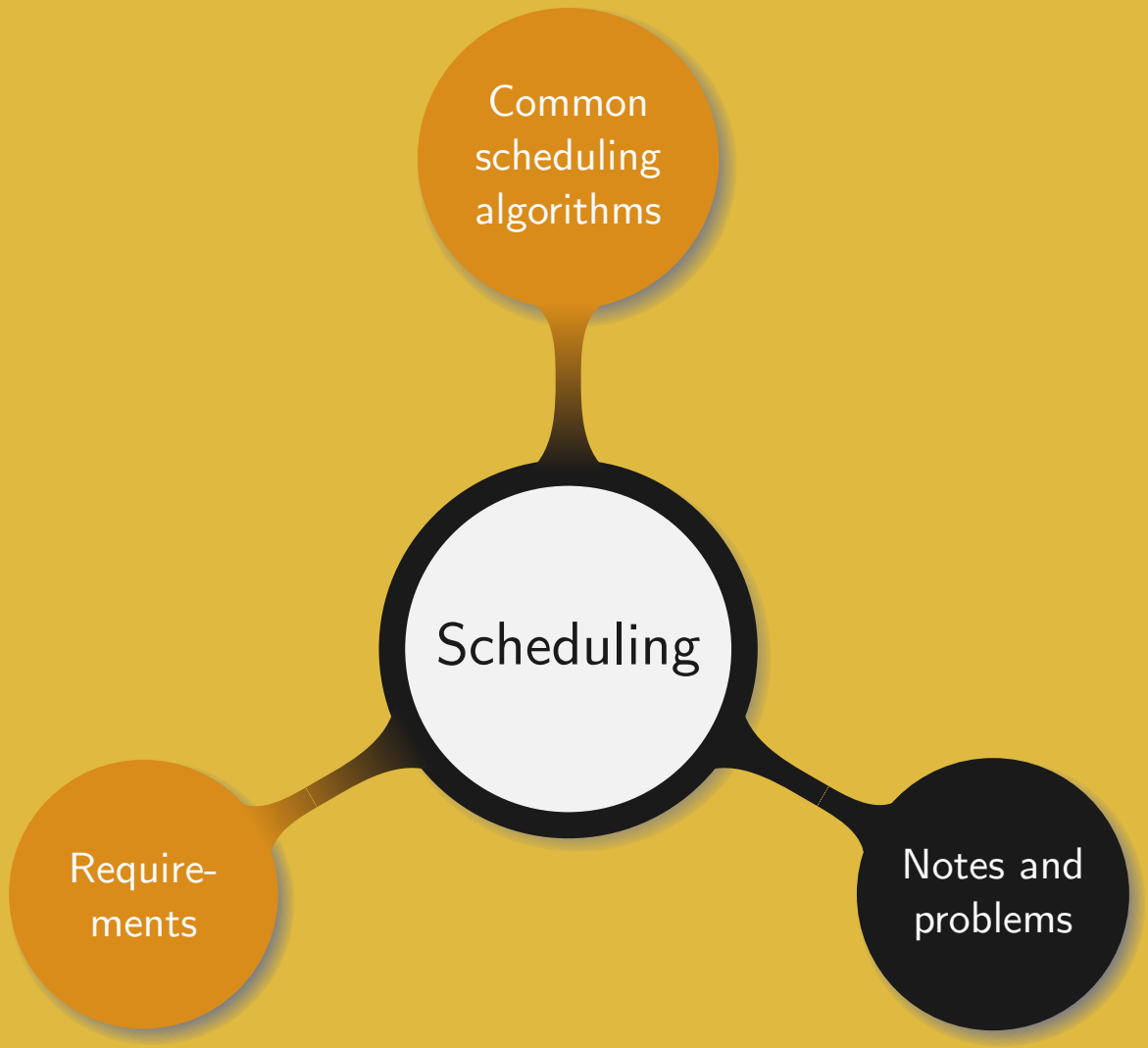


Priority based preemptive algorithm:

- 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

When is this algorithm appropriate and when should it be avoided?

Chapter organisation



Policy vs. mechanism

Limitations of the previous algorithms:

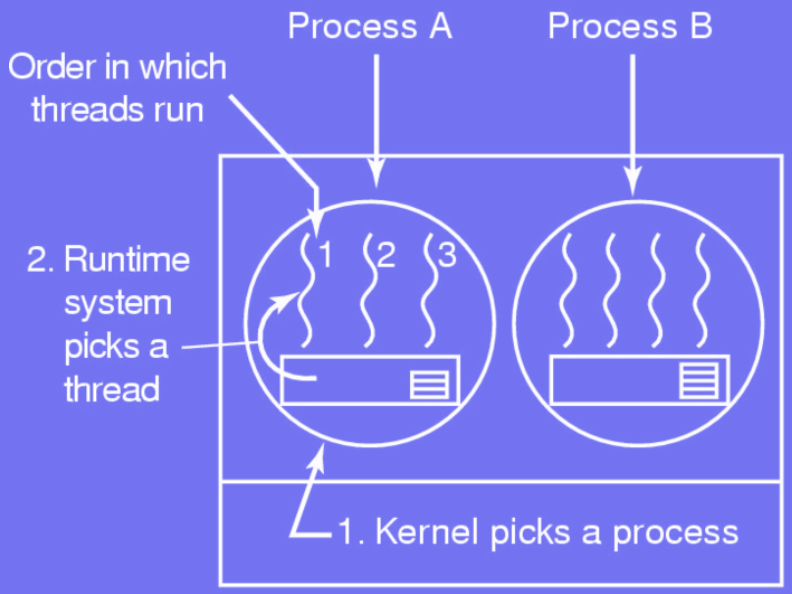
- They all assume that processes are competing
- Parent could know which of its children is most important

Separate the scheduling mechanism from the scheduling policy:

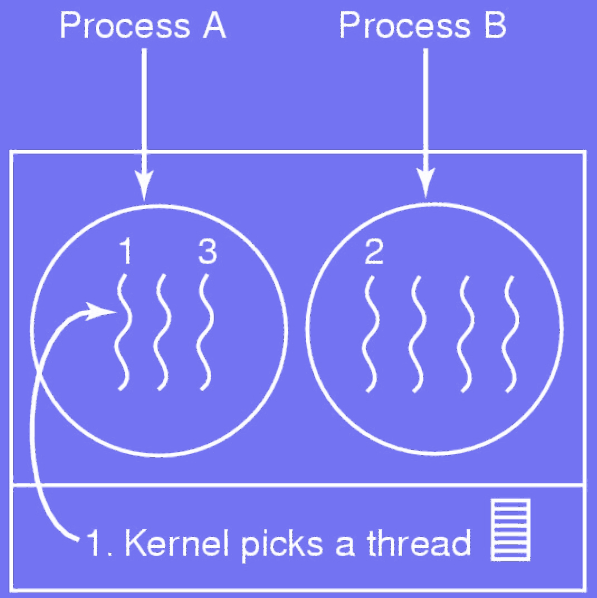
- Scheduling algorithm has parameters
- Parameters can be set by processes
- A parent can decide which of its children should have higher priority

Threads scheduling

Threads in user-space



Threads in kernel-space



In each case which of the following running orders are possible:

- A1, A2, A3, A1, A2, A3
- 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

What if they all take the left chop-stick at the same time?

Second solution:

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

What if they all start at the same time?

The dining philosophers problem

A solution using mutex:

- A philosopher thinks
- Locks mutex
- Acquires chop-sticks, eat, put them down
- Unlocks the mutex

How many philosophers can eat at the same time?

The dining philosophers problem

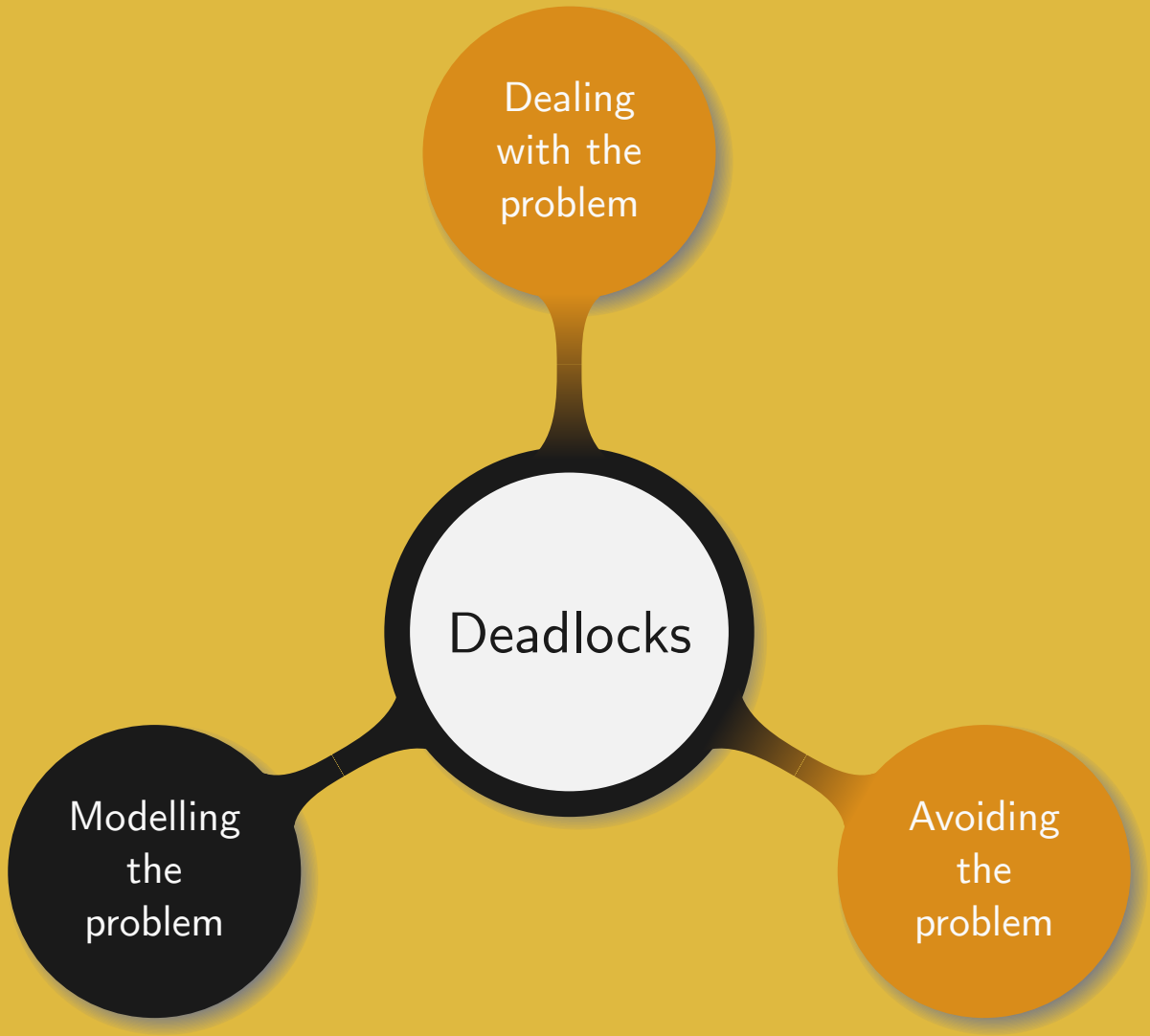
```
1  #define N 5
2  #define LEFT (i+N-1)%N
3  #define RIGHT (i+1)%N
4  enum { THINKING, HUNGRY, EATING };
5  int state[N]; mutex mut = 0 ; semaphore s[N];
6  void philosopher(int i) {while(TRUE) {think();take_cs(i);eat();put_cs(i);}}
7  void take_cs(int i) {
8    mutex-lock(&mut);
9    state[i] = HUNGRY; 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 }
```

Key points

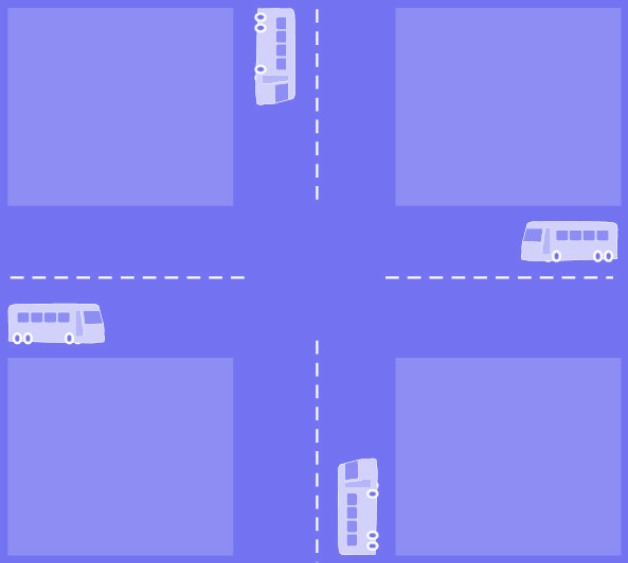
- Why is scheduling the lowest part of the OS?
- What are the two main types of algorithm?
- What are the two most common scheduling algorithms?
- Give an example of theoretical problem related to scheduling

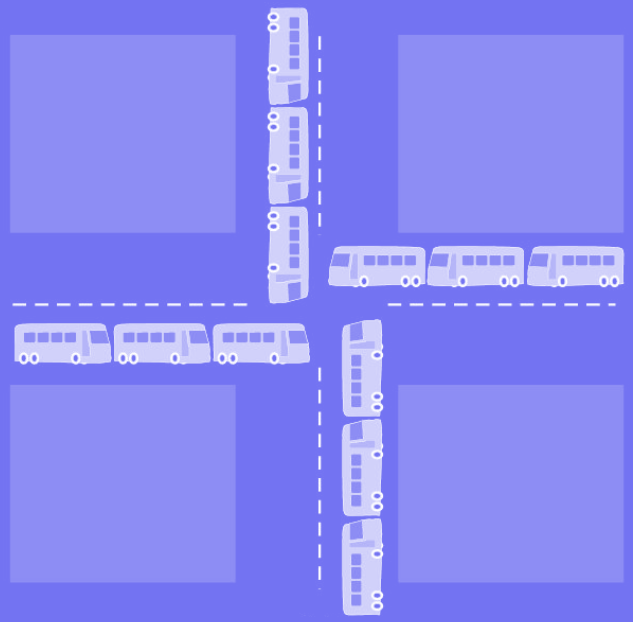
5. Deadlocks

Chapter organisation



Deadlocks in real life





Resource types

Two main types of resources:

- Preemptable: resources that can be safely taken away
- Non-preemptable: resources cannot be safely taken away

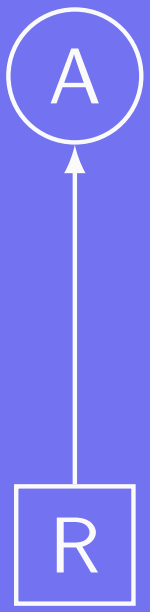
Examples.

- Preemptable:
  - Total RAM is 256MB, A and B are 256 MB each
  - A is loaded in RAM, acquires the printer, but exceeds its quantum
  - B is loaded in RAM but fails to acquire the memory
  - Memory can be taken away of B and given to A, which can complete
- Non-preemptable:
  - A is burning a DVD
  - B wants to access the DVD burner, but the drive is not accessible
  - The resource cannot be taken way from A without any damage

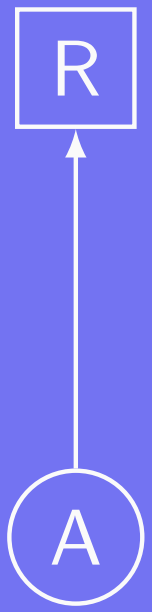
Representing deadlocks

Graphs to represent resource allocation

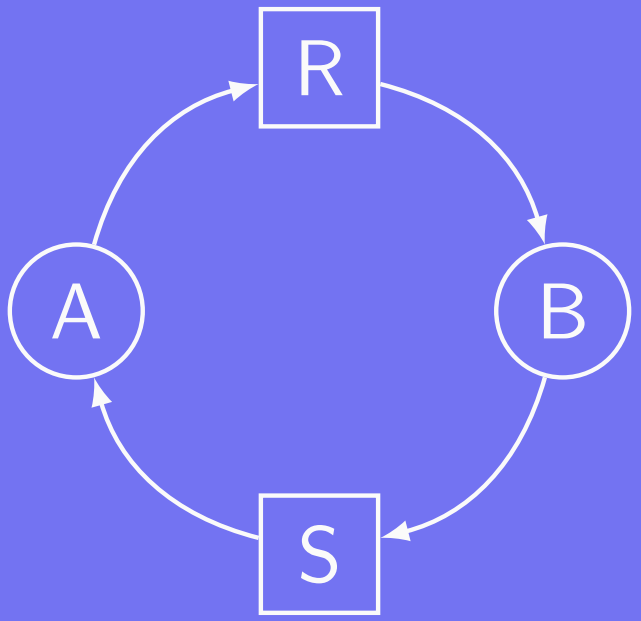
Resource *R* held by *A*



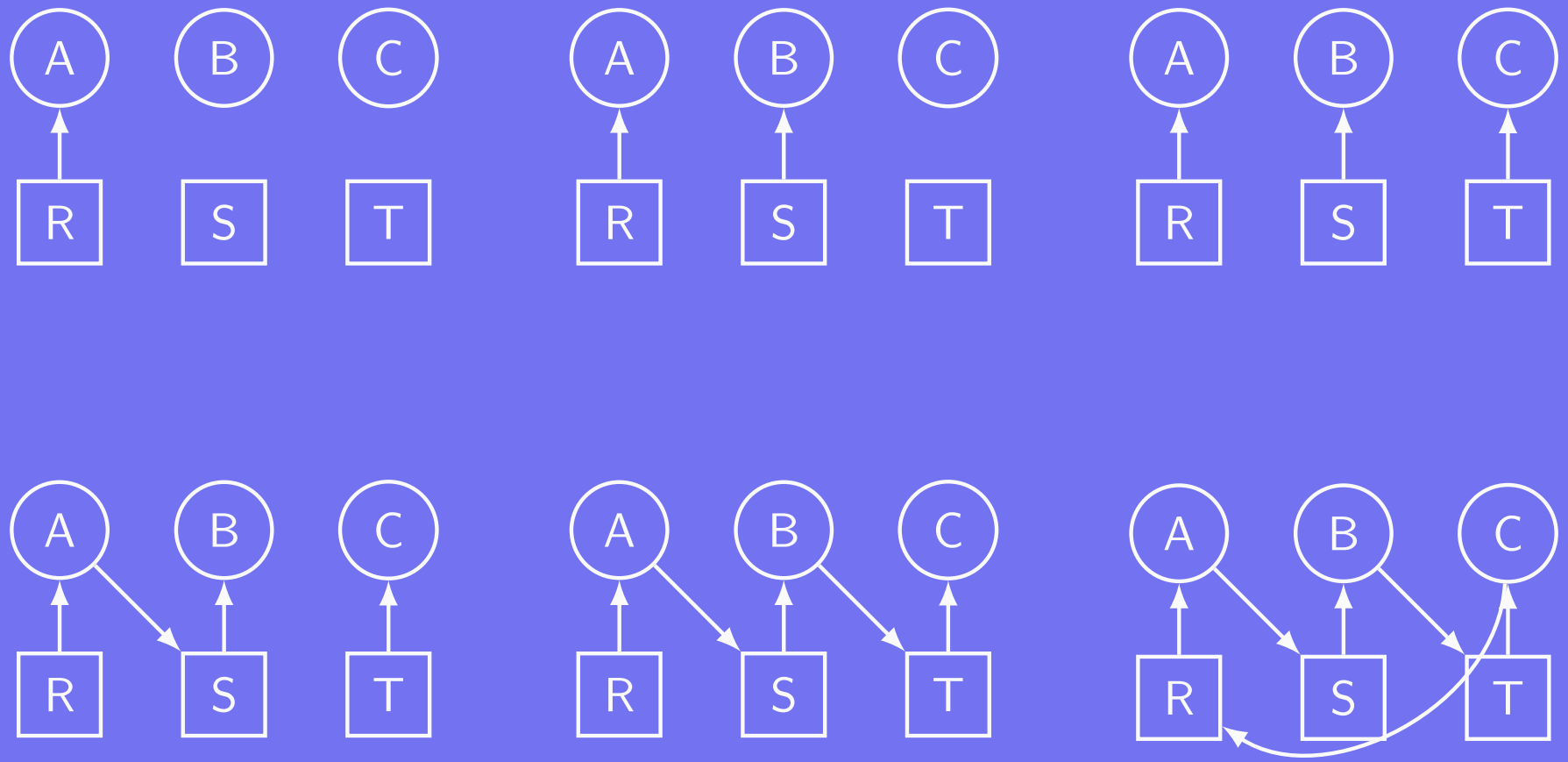
Resource *R* requested by *A*



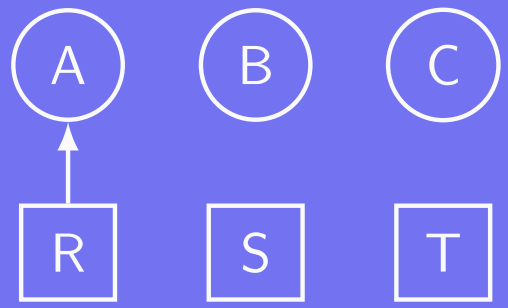
Deadlock

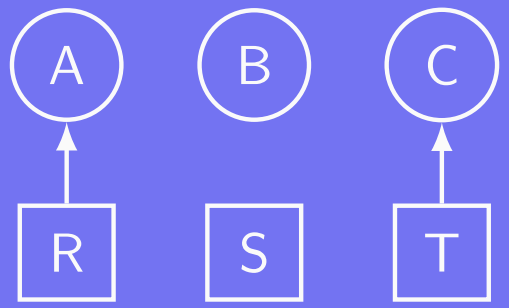


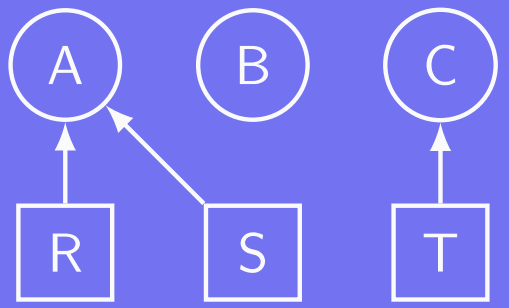
Example – Deadlock

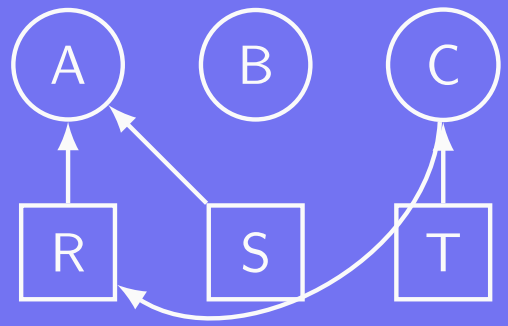


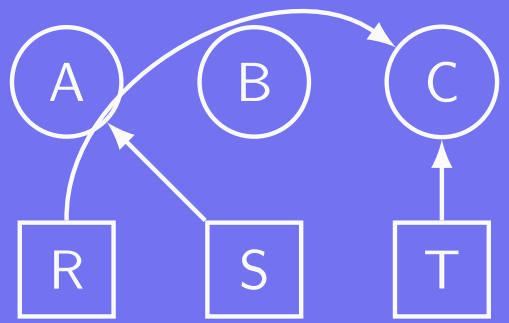
Example – No deadlock

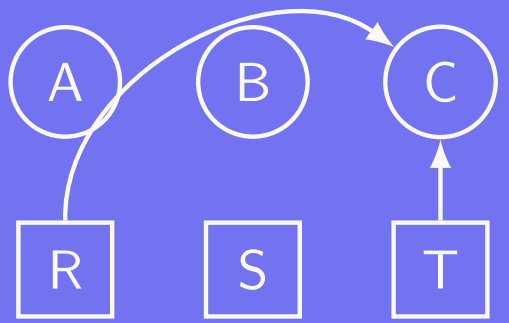




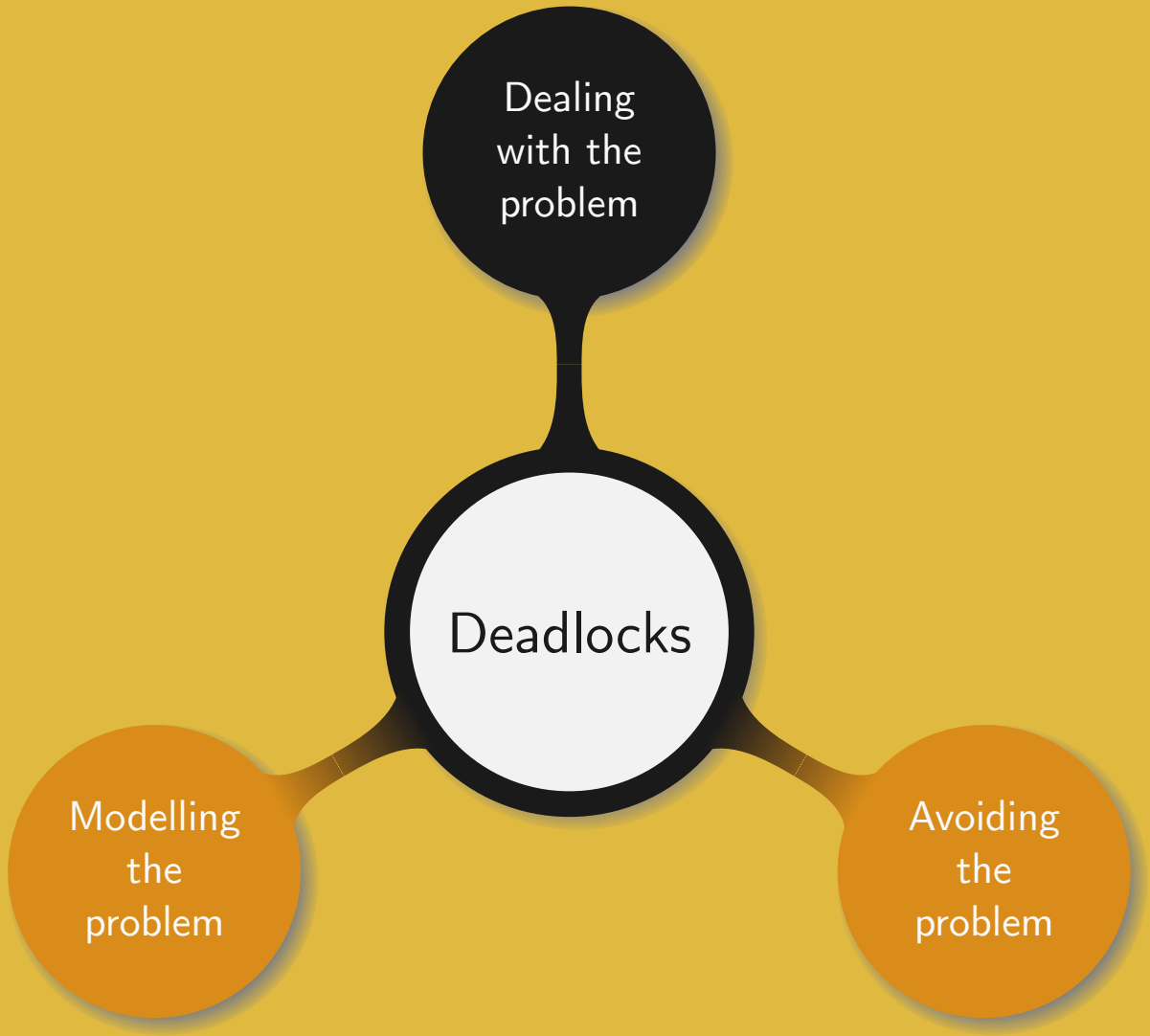








Chapter organisation



ALERT, ALERT

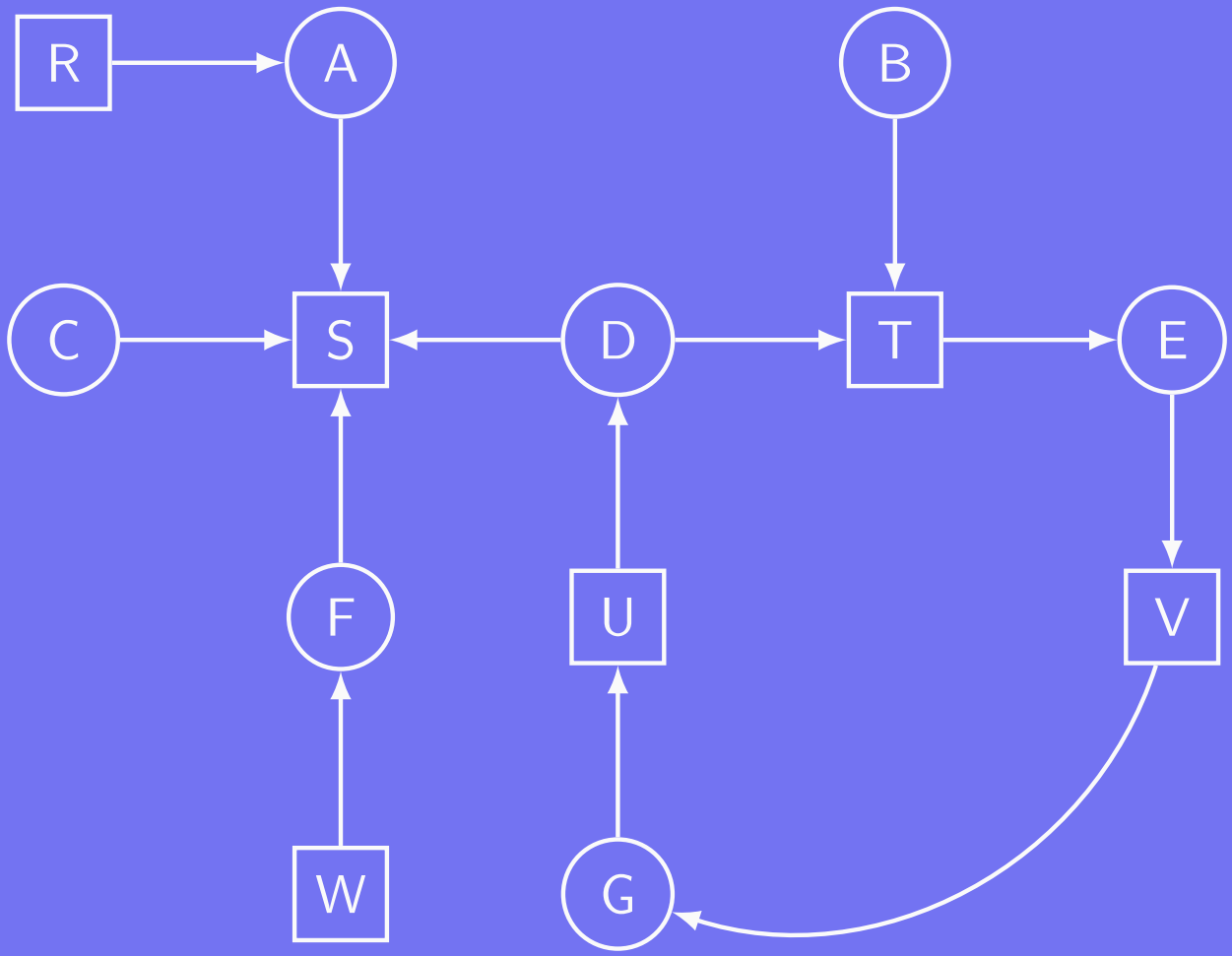
HIDE!!!!

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

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

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

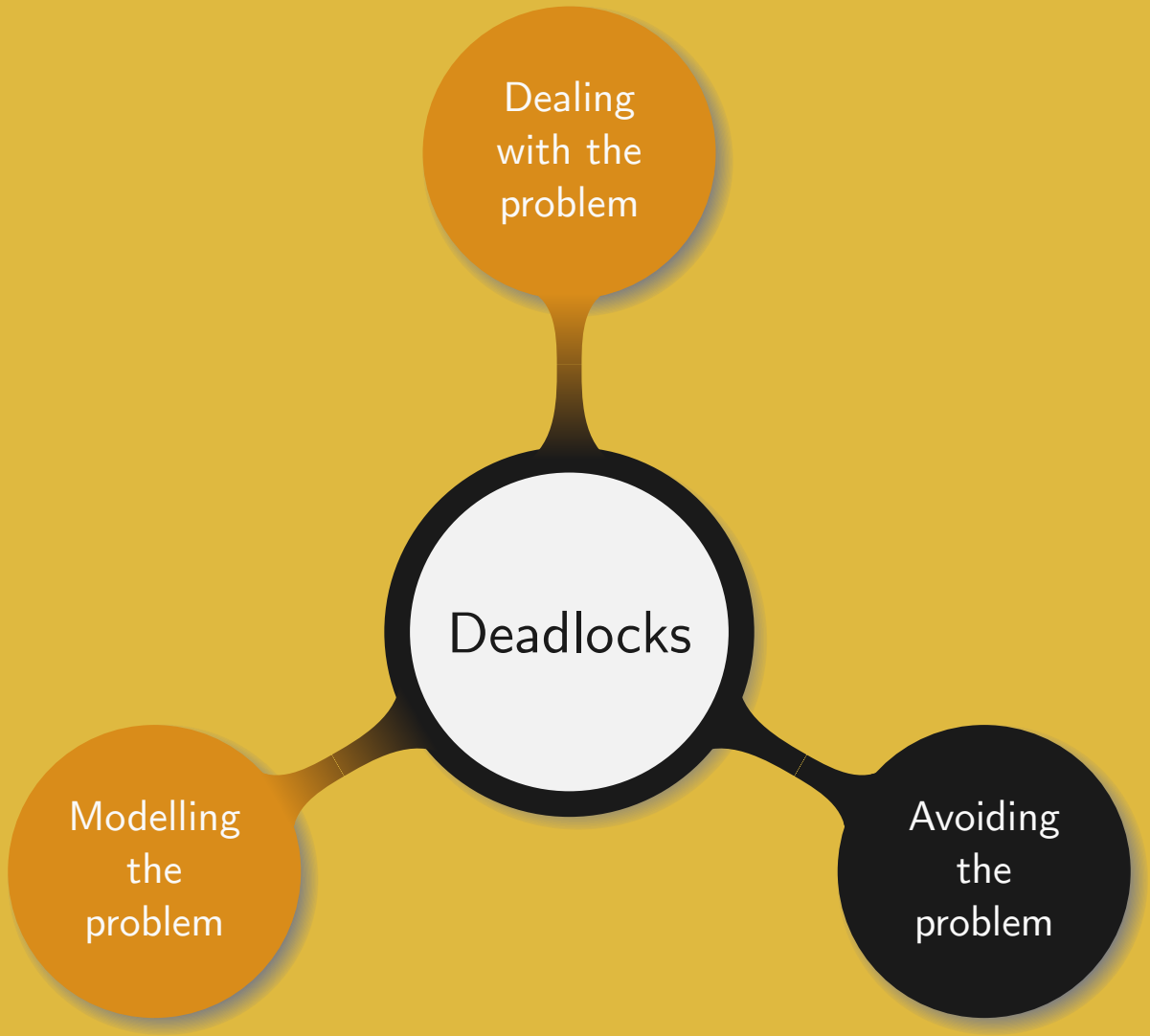
What if the second process requests two DVD burners?

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



Avoiding vs. preventing

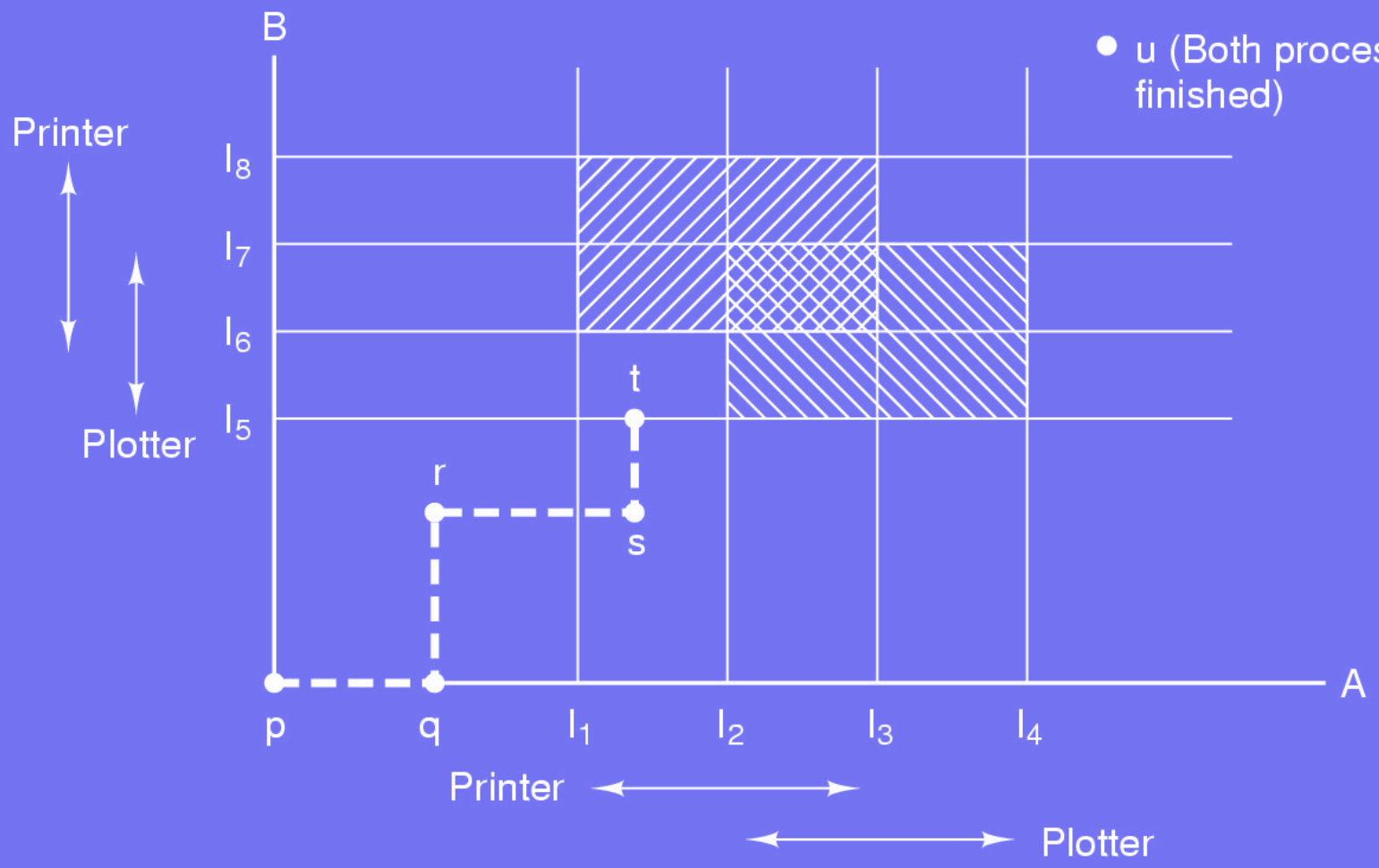
Avoiding deadlocks:

- Resources could be assigned and released one at a time
- Problems to solve
  - How to implement this idea?
  - How efficient would such a strategy be?

Preventing deadlocks:

- Model the circumstances under which a deadlock occurs
- Problems:
  - How would such a model be?
  - How much would this strategy slow down the system?

Resources trajectories



Defining states

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

- Safe state: there exists an order allowing all processes to complete, even if they request their maximum number of resources when scheduled. It 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 and deallocated. There is no way to predict whether or not all the processes will finish.

Remarks.

- 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.
- A process running alone will never encounter a deadlock. If it needs more resources than available it is assumed that they will not be all requested at once.

The banker's algorithm

General idea:

- Introduced by Dijkstra in 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 sate 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:

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

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

Deadlock prevention

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

Aside from carefully assigning resources not much can be done

Hold and wait condition

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

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)

Not a viable solution

Circular wait 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?

Best solution but it not always possible to use it in practice

A few more remarks

Deadlocks are not necessarily related to hardware resources:

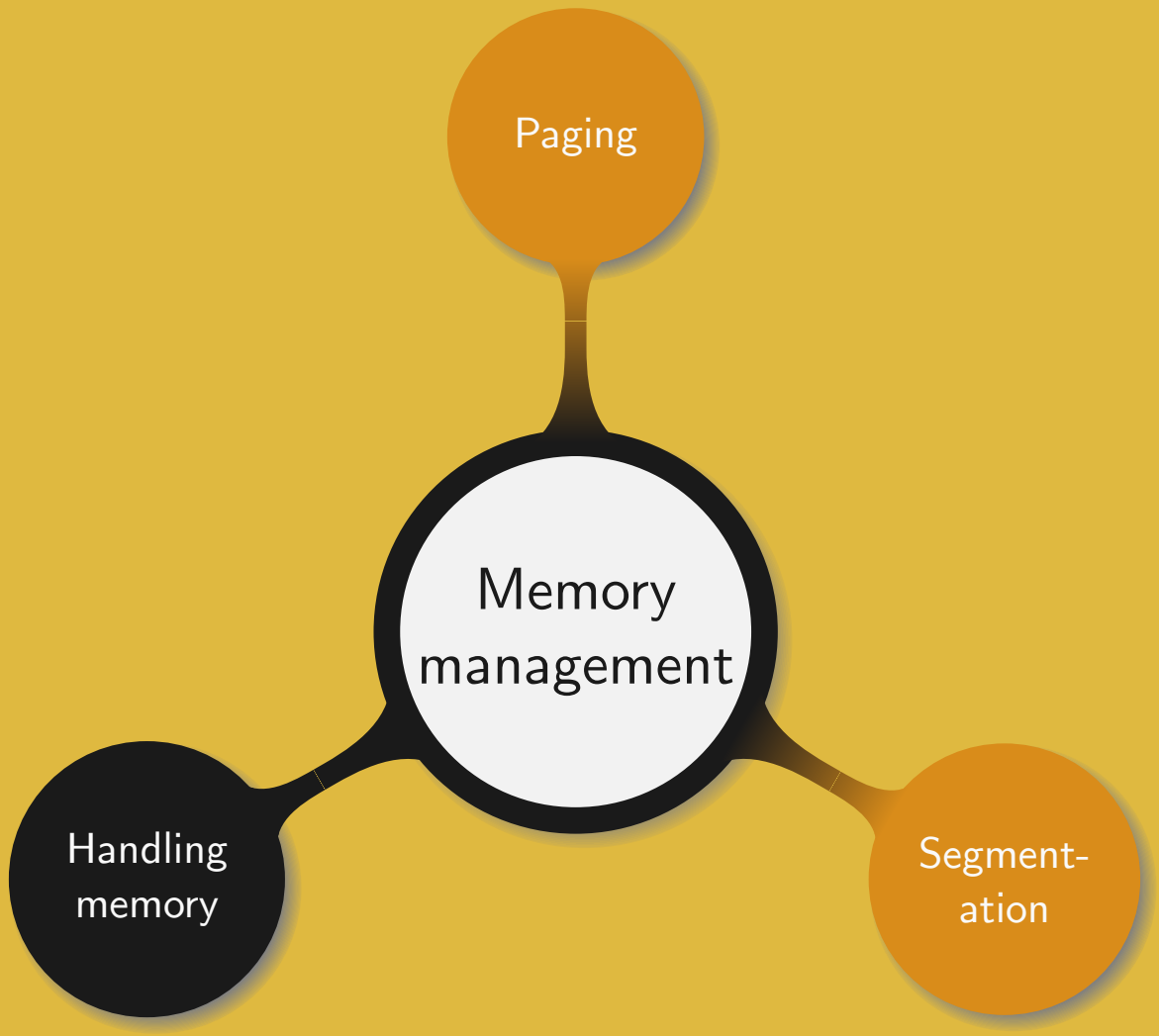
- Database records: two-phase locking solution; lock all the records in phase one; if one fails release all the locks and retry later, otherwise proceed with phase two and manipulate the records
- Communication deadlocks: mutex can lead to deadlocks; no hardware resource involved; could be due to the 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: a long process delayed to let shorter ones run, might never run...

Key points

- How to detect deadlocks?
- How to fix a deadlock in a clean way?
- What are the four conditions that characterise a deadlock?
- What is a common practice regarding deadlocks?

6. Memory management

Chapter organisation



Memory hierarchy

Access time		Capacity
1 ns	Registers	< 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?

Memory manager

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

Simplest model

No memory abstraction:

- Program sees the actual physical memory
- Programmers can 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 a time
  - More than one program is possible if using special hardware

Address space

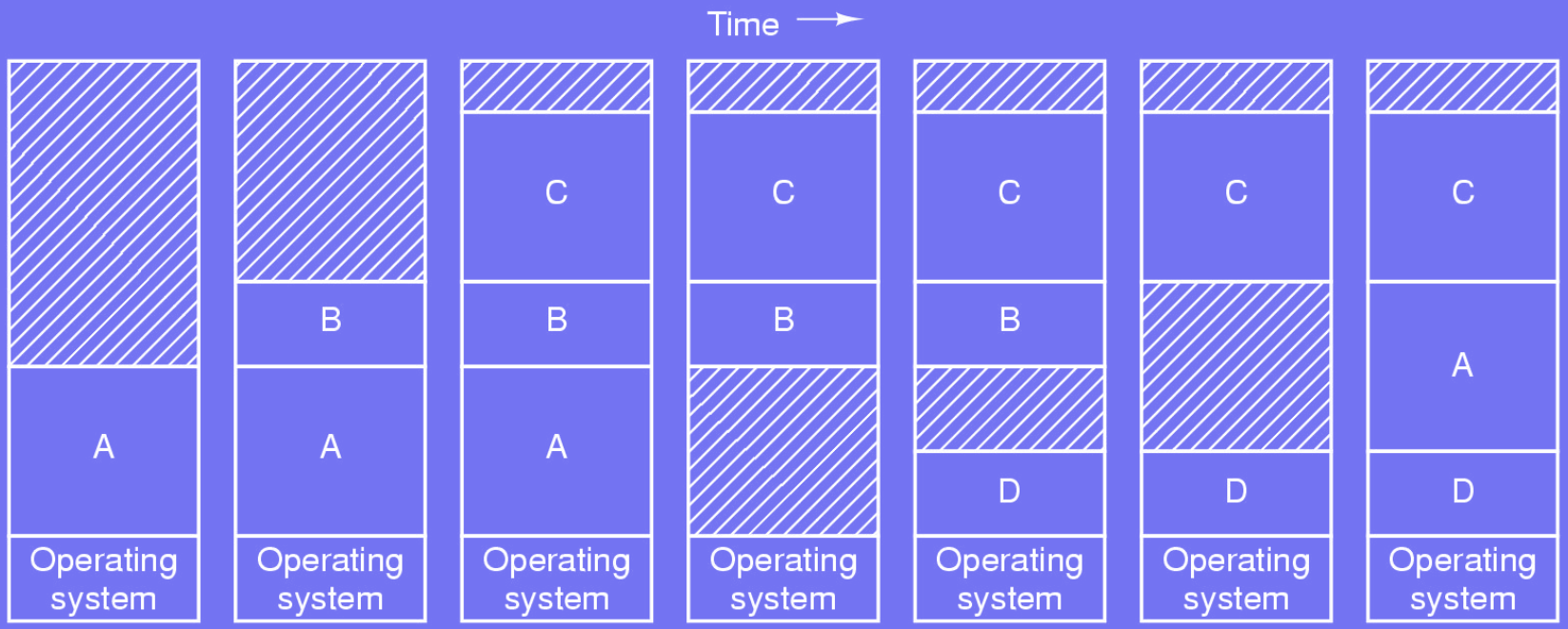
No abstraction leads to two main problems:

- 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

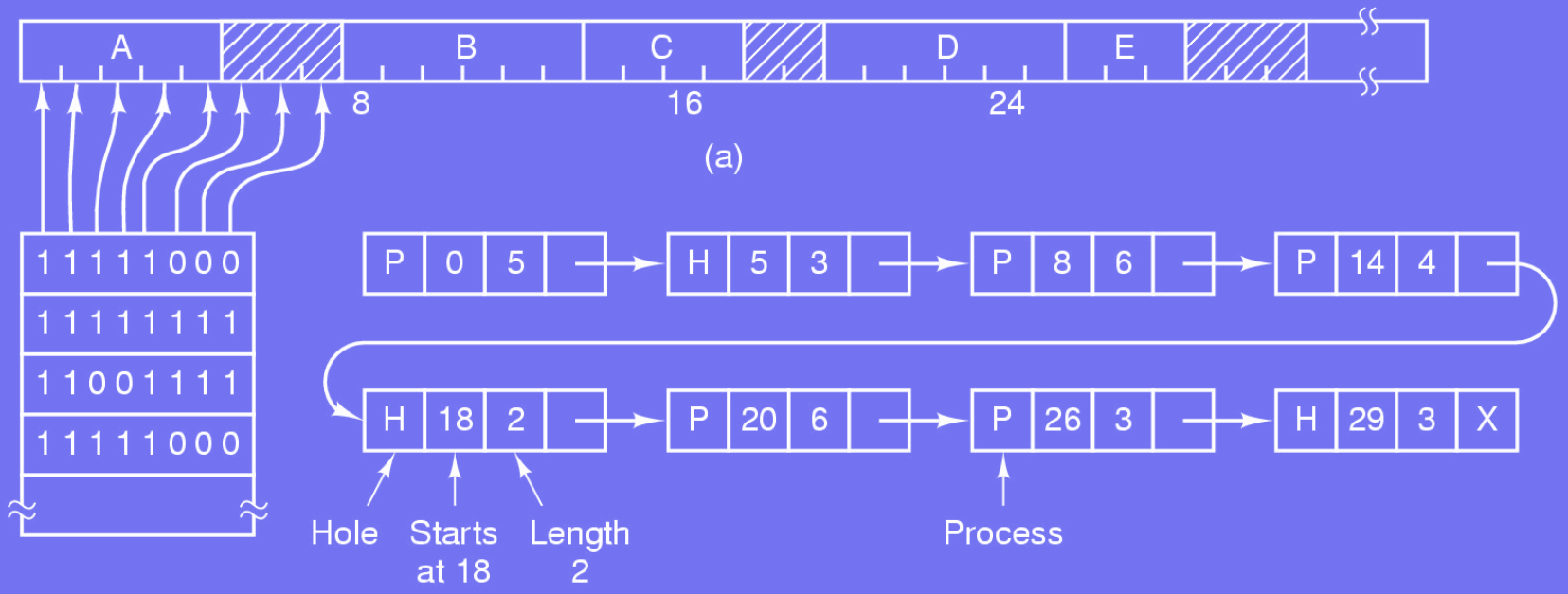
Memory size limitation



When booting many processes are started:

- As more programs are run more and more memory is needed
- More memory than available might be needed
- 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

Ways to assign memory to processes:

- 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 for common memory sizes, use the best

Characteristics:

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

Basics idea

Virtual memory:

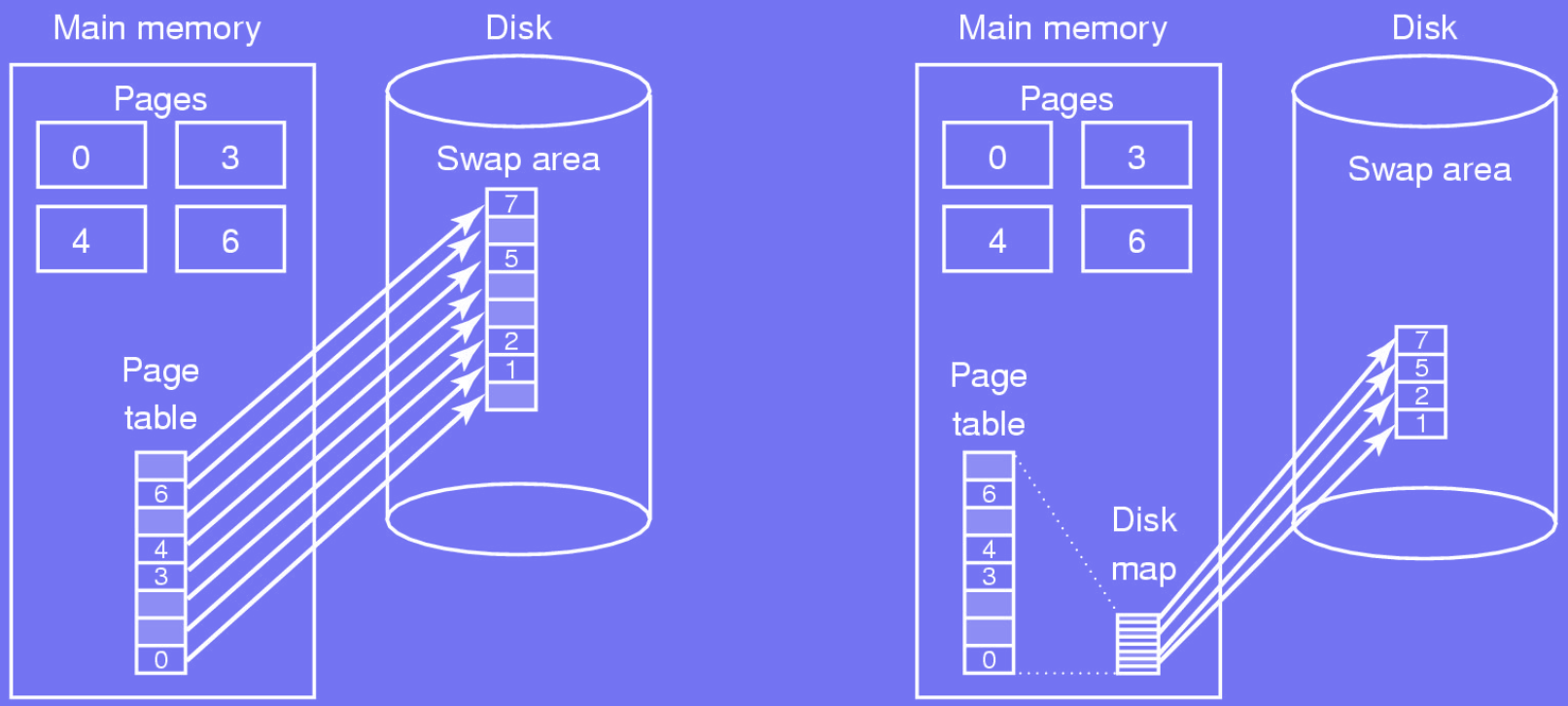
- Generalisation of the base and limit registers
- Each process has its own address space
- The address space is split into chunks called *pages*
- Each page corresponds to a range of addresses
- Pages are mapped onto physical memory
- Pages can be on different medium, e.g. RAM and swap

The swap area

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

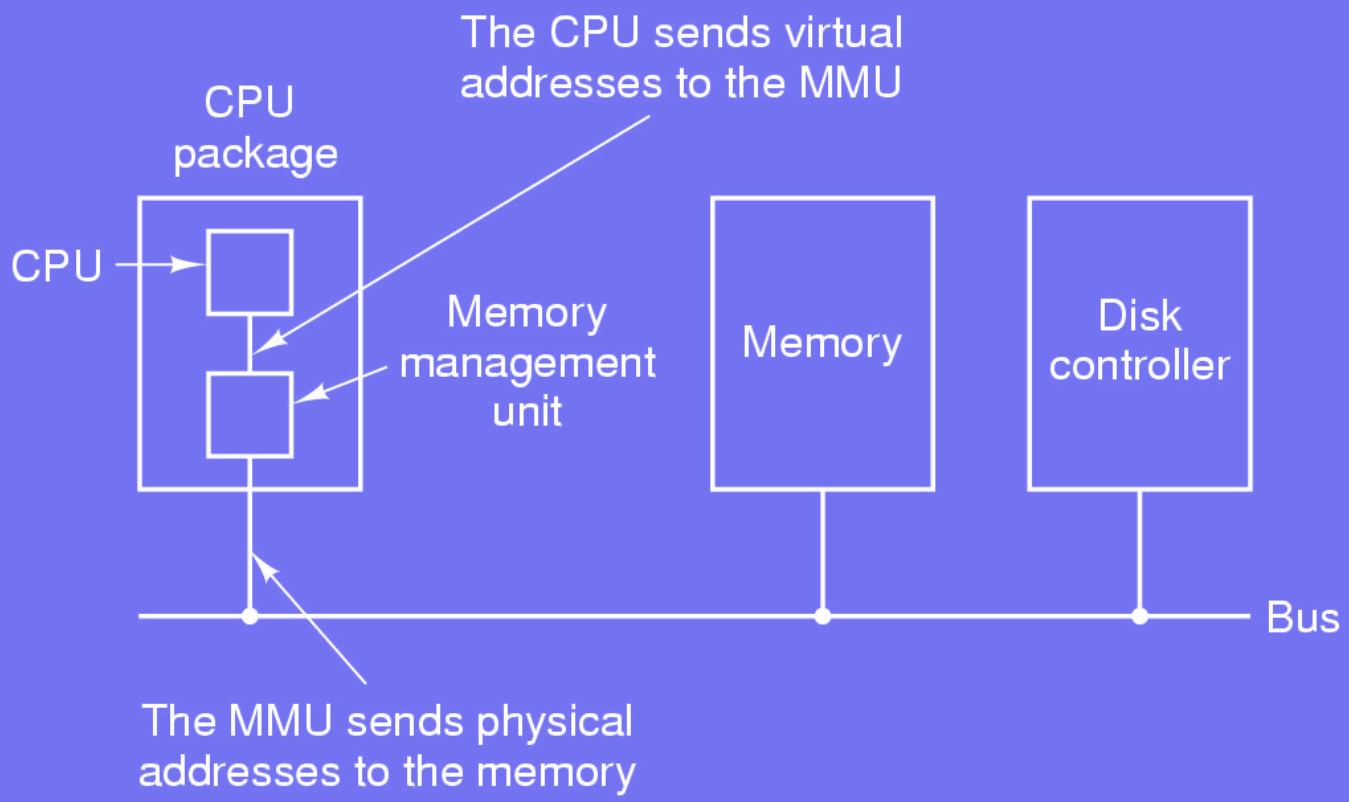
Initialising the swap area



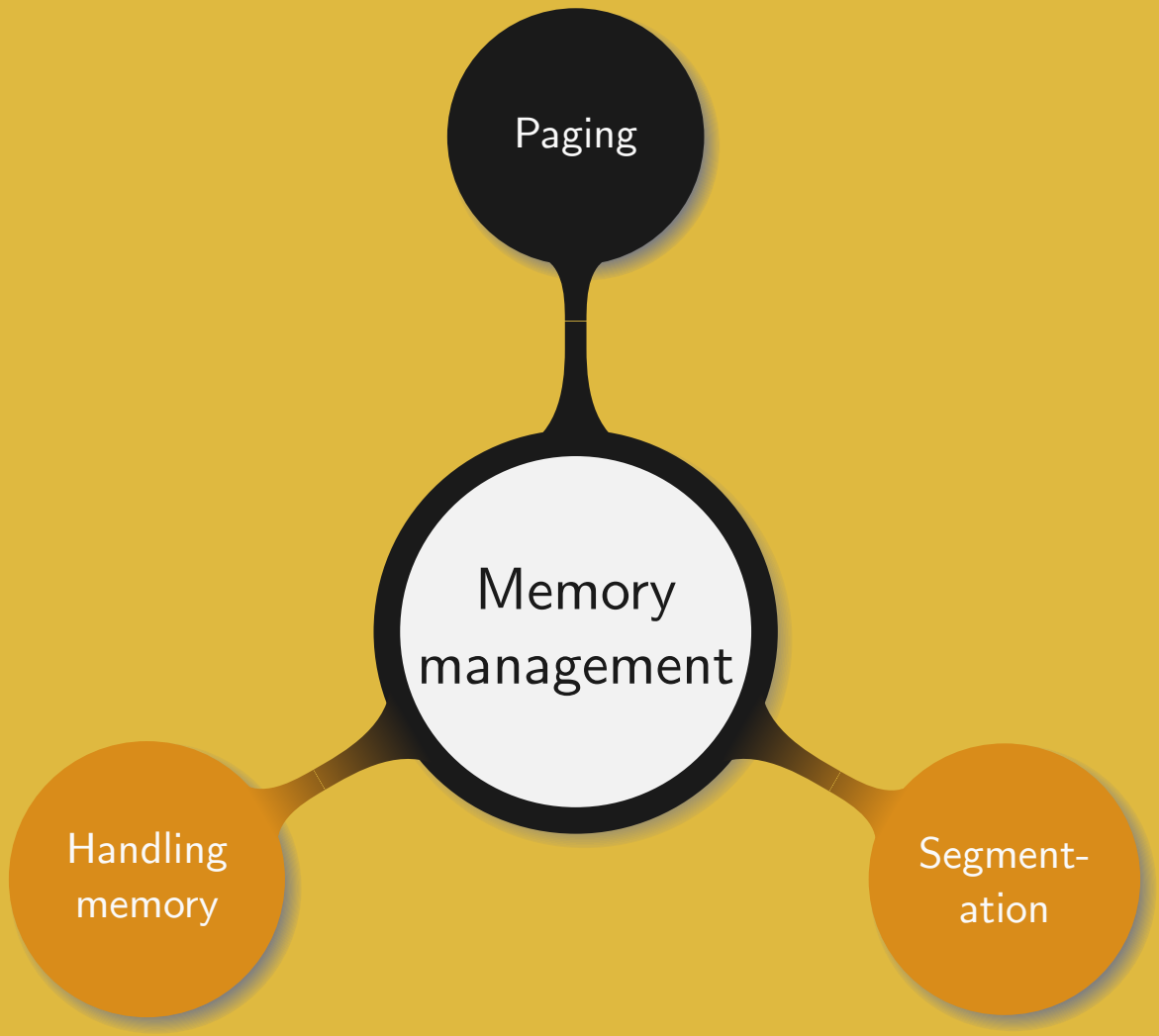
Two main strategies:

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

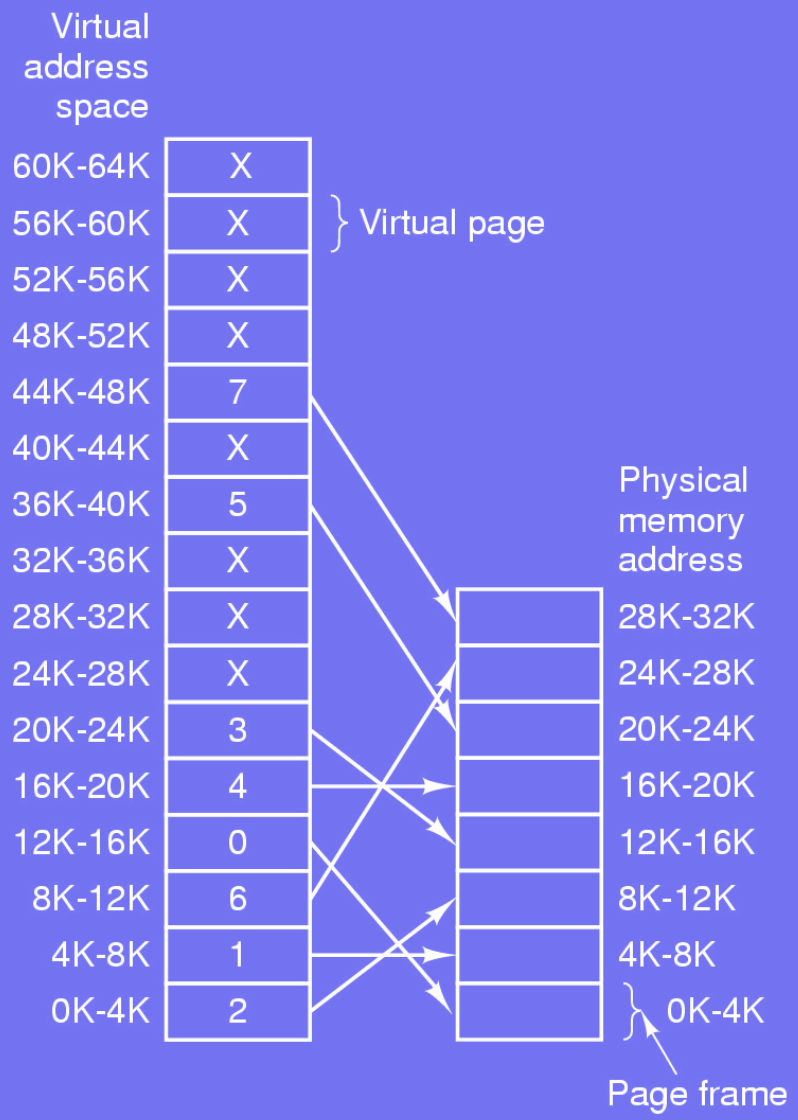
Memory Management Unit



Chapter organisation



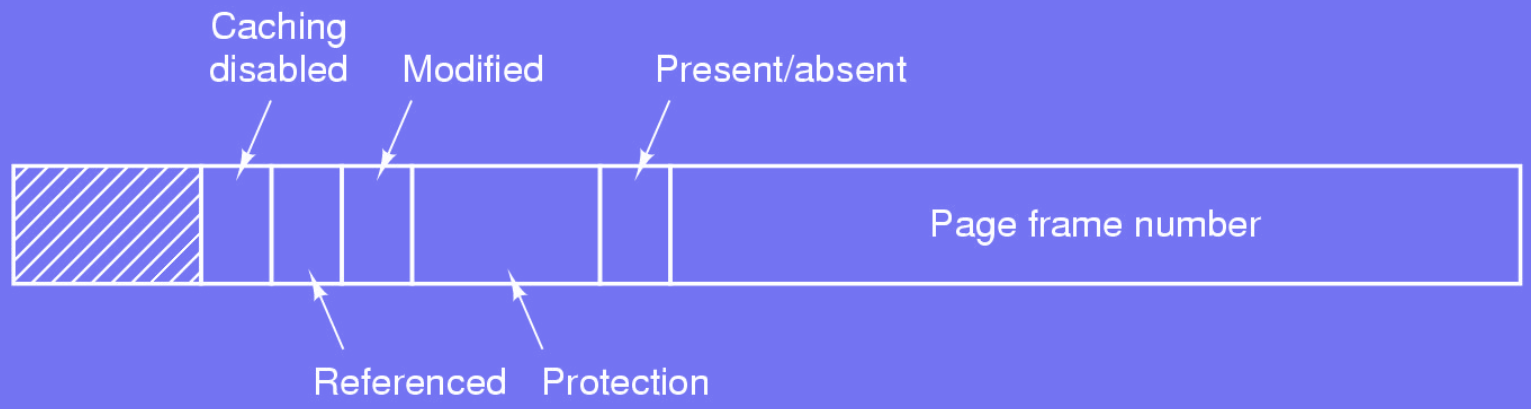
Virtual page and page frame



Organising the memory:

- Virtual address space divided into fixed-size units called *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

Page table



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

Paging

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

Page replacement

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

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

Can the information be known ahead of time?

Page replacement – LRU

Recently heavily used pages are very likely to be used again 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

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

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

The M and R bits

Use of the M and R bits for each page table entry:

- 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

Page replacement – Aging

Simulating 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. $n = 8$ with 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

Counter has a finite number of bits, a state is lost after $n \cdot t$

Definitions

Basic notions related to paging:

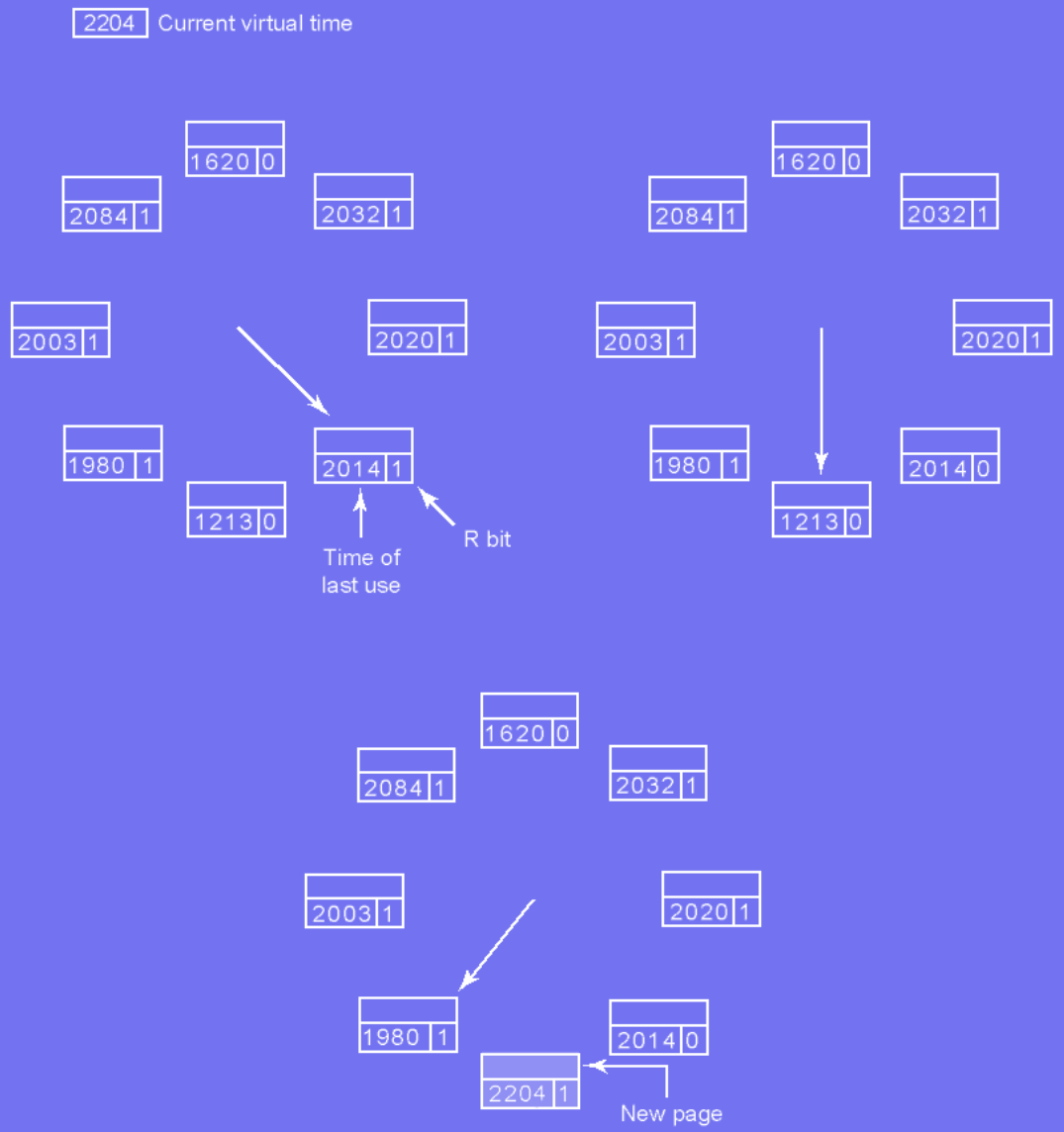
- Demand paging: pages are loaded on demand
- Locality reference: during an execution phase a process only access a small fraction of all its pages
- Working set: set of pages currently used by a process
- Thrashing: process causes many page fault due to a lack of memory
- Pre-paging: pages loaded in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

Page replacement – WSClock

Using a circular list of page frames for pages which have been inserted:

- Each entry is 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 $R = 0$, 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



Local vs. global allocation

Onto which set should the page replacement algorithm be applied:

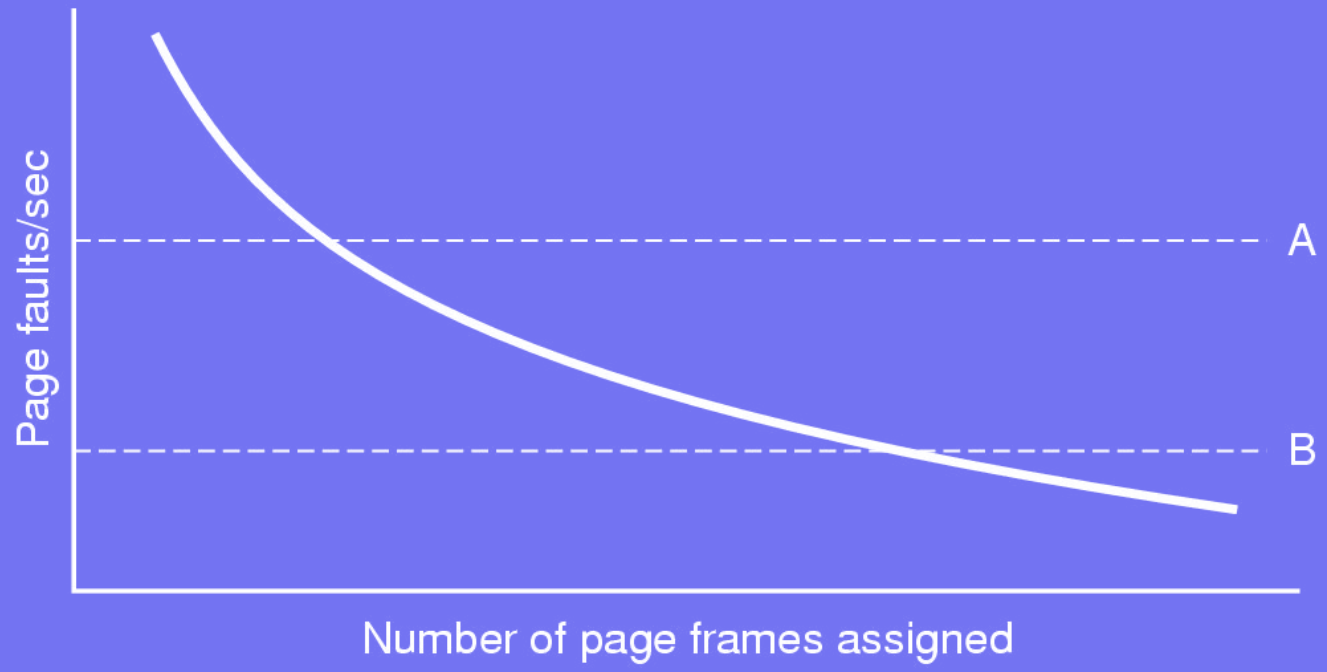
- Local, i.e. within the process:
  - Allocate a portion of the whole memory to a process
  - Only use the allocated portion
  - Number of page frames for a process remains constant
- Global, i.e. within the whole memory:
  - Dynamically allocate page frames to a processes
  - Number of page frames for 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



Page size

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:

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

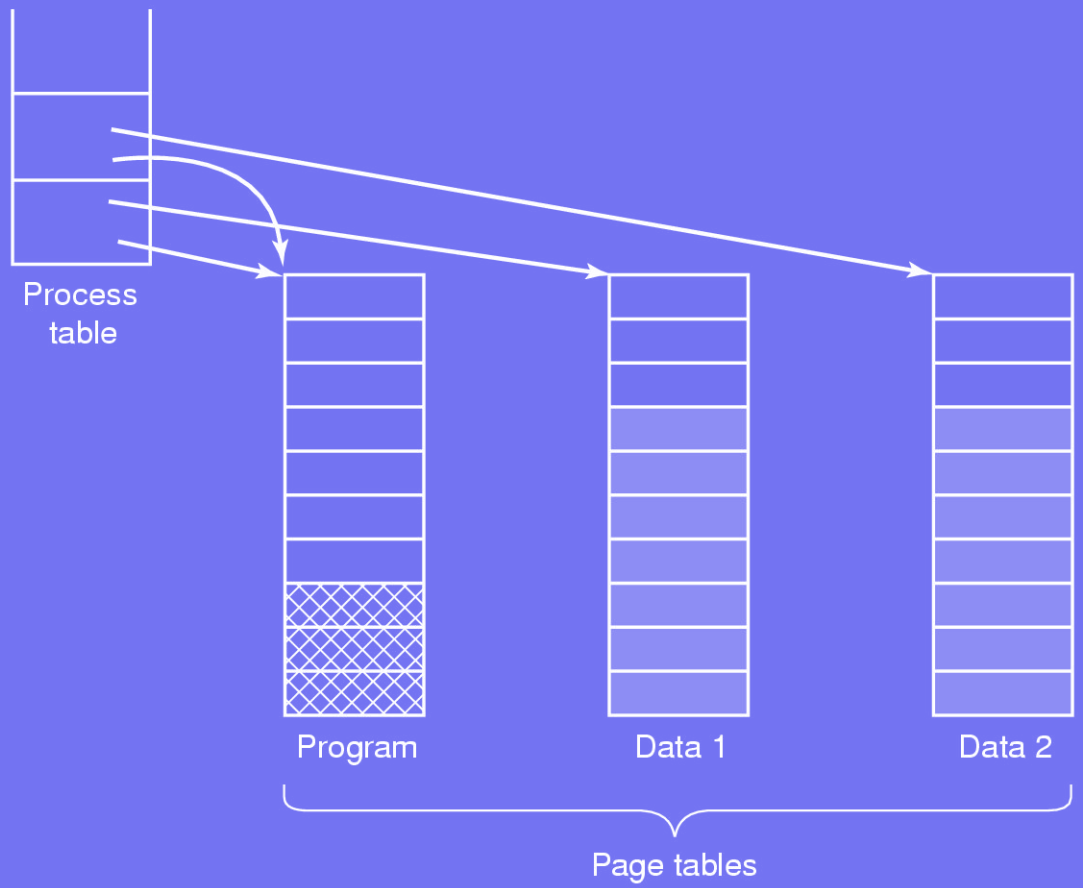
Optimal page size: $p = \sqrt{2se}$

Common page frame sizes: 4KB or 8KB

Page sharing

Decrease memory usage by sharing pages:

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



Page sharing

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?

Importance of paging in the OS

OS involved in paging related work on four occasions:

- Process creation:
  1. Determine process size
  2. Create process' page table (allocate and initialise memory)
  3. Initialise swap area
  4. Store information related to the swap area and page table in the process table
- Process execution:
  1. MMU resets for the new process
  2. Flush the TLB
  3. Make the new process' page table the current one

Importance of paging in the OS

OS involved in paging related work on four occasions:

- Page fault:
  1. Read hardware register to determine origin of page fault
  2. Compute which page is needed
  3. Locate the page on the disk
  4. Find an available page frame and replace its content
  5. Read the new page frame
  6. Rewind to the faulting instruction and re-execute it
- Process termination:
  1. Release page table entries, pages, and disk space
  2. Beware of any page that could be shared among several processes

Page fault handling

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 reserved such as not to be used by another process

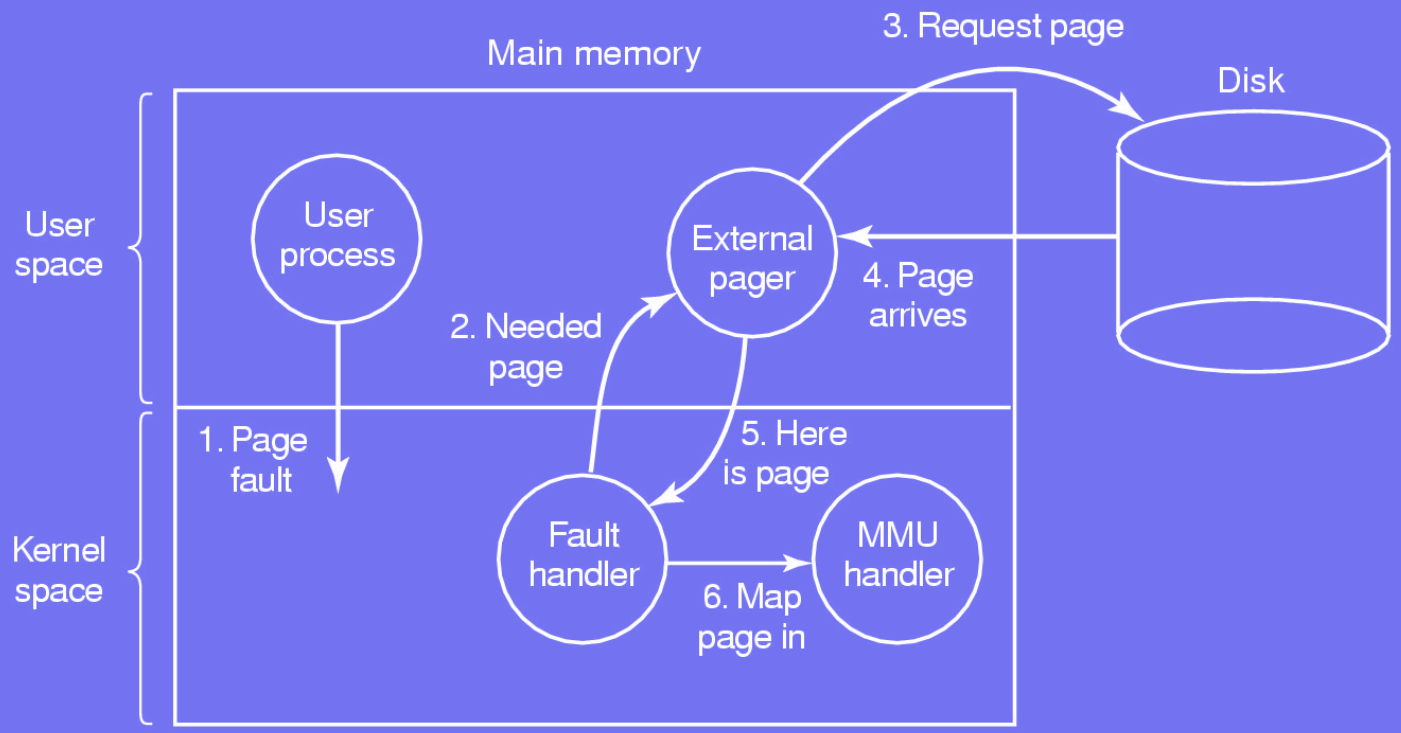
Page fault handling

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 showing how to dissociate policies from mechanisms:

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



Organisational issue

Where should the page replacement algorithm go?

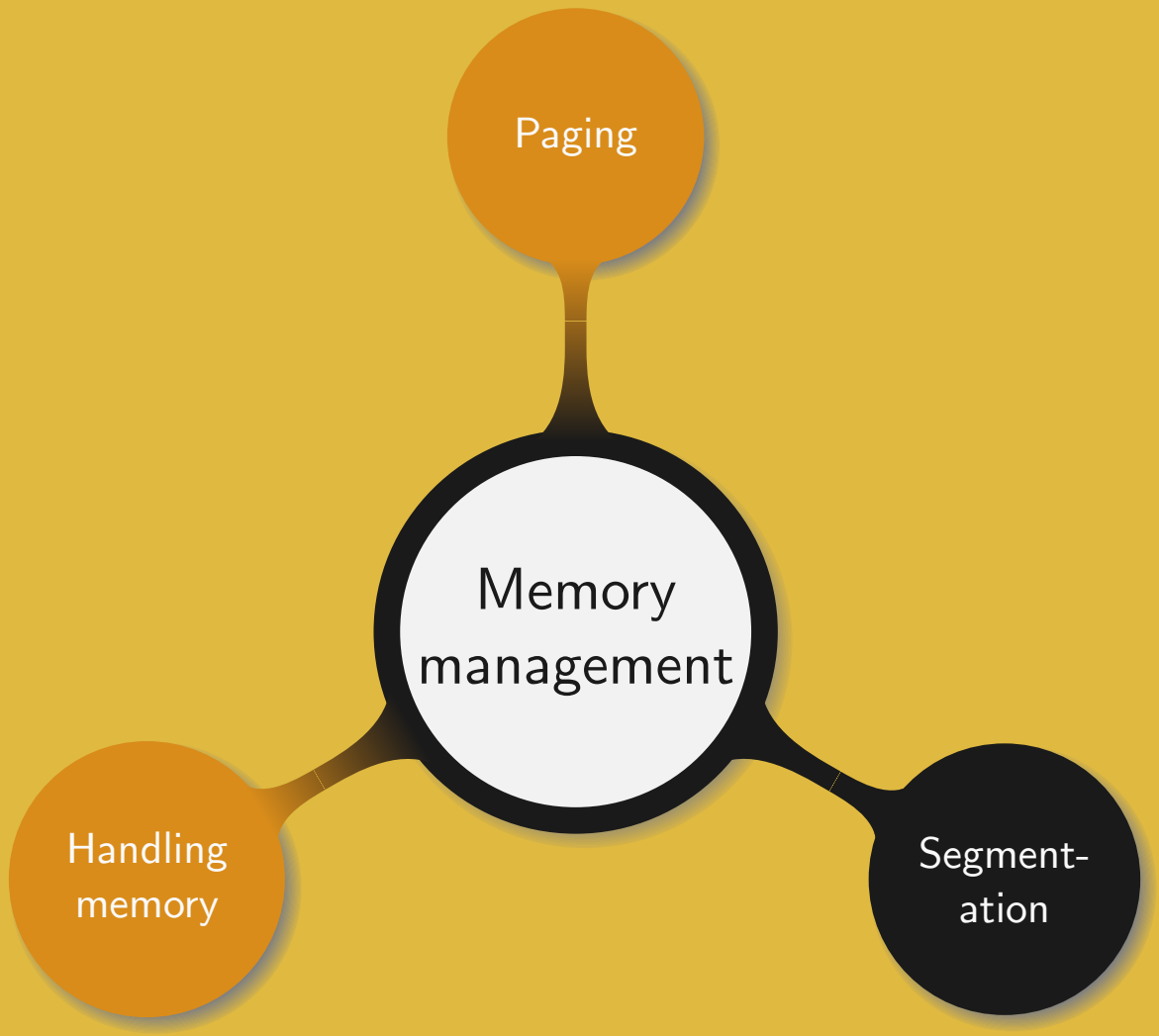
User space

- Use some mechanism to access the R and M bits
- Clean solution
- Overhead resulting from crossing user-kernel boundary several times
- Modular code, more flexibility

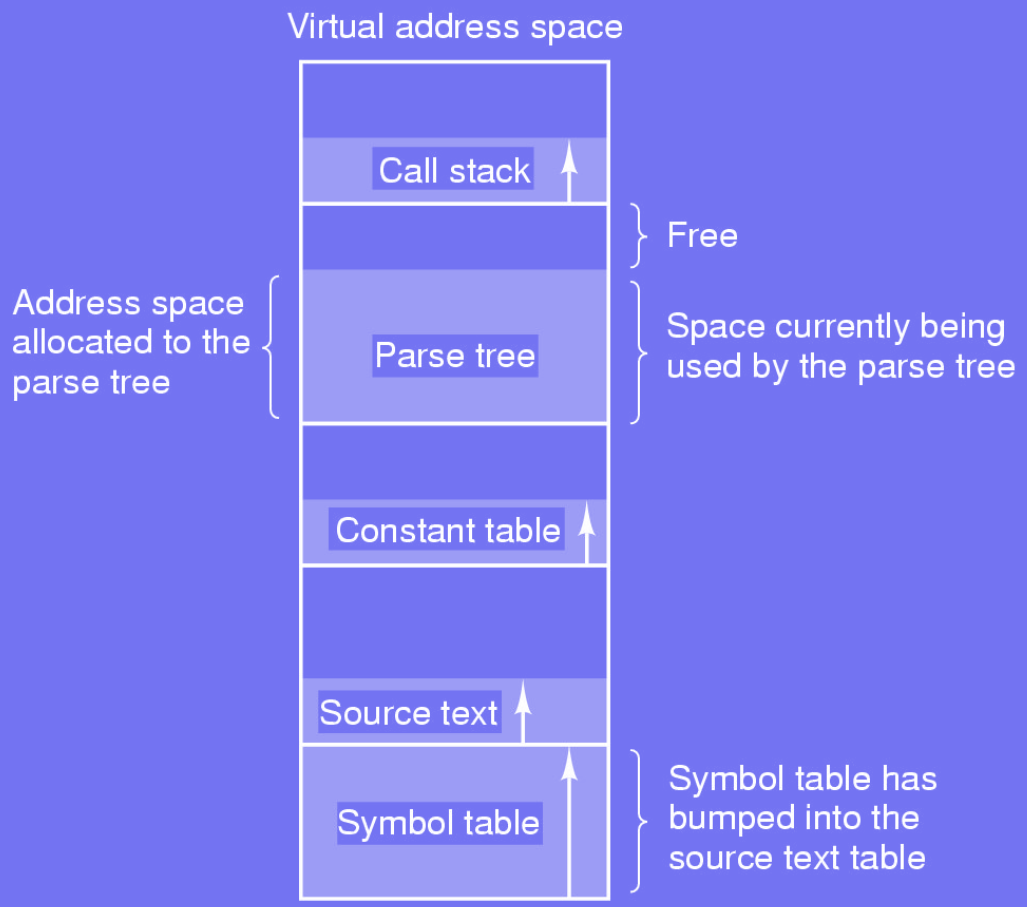
Kernel space

- 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



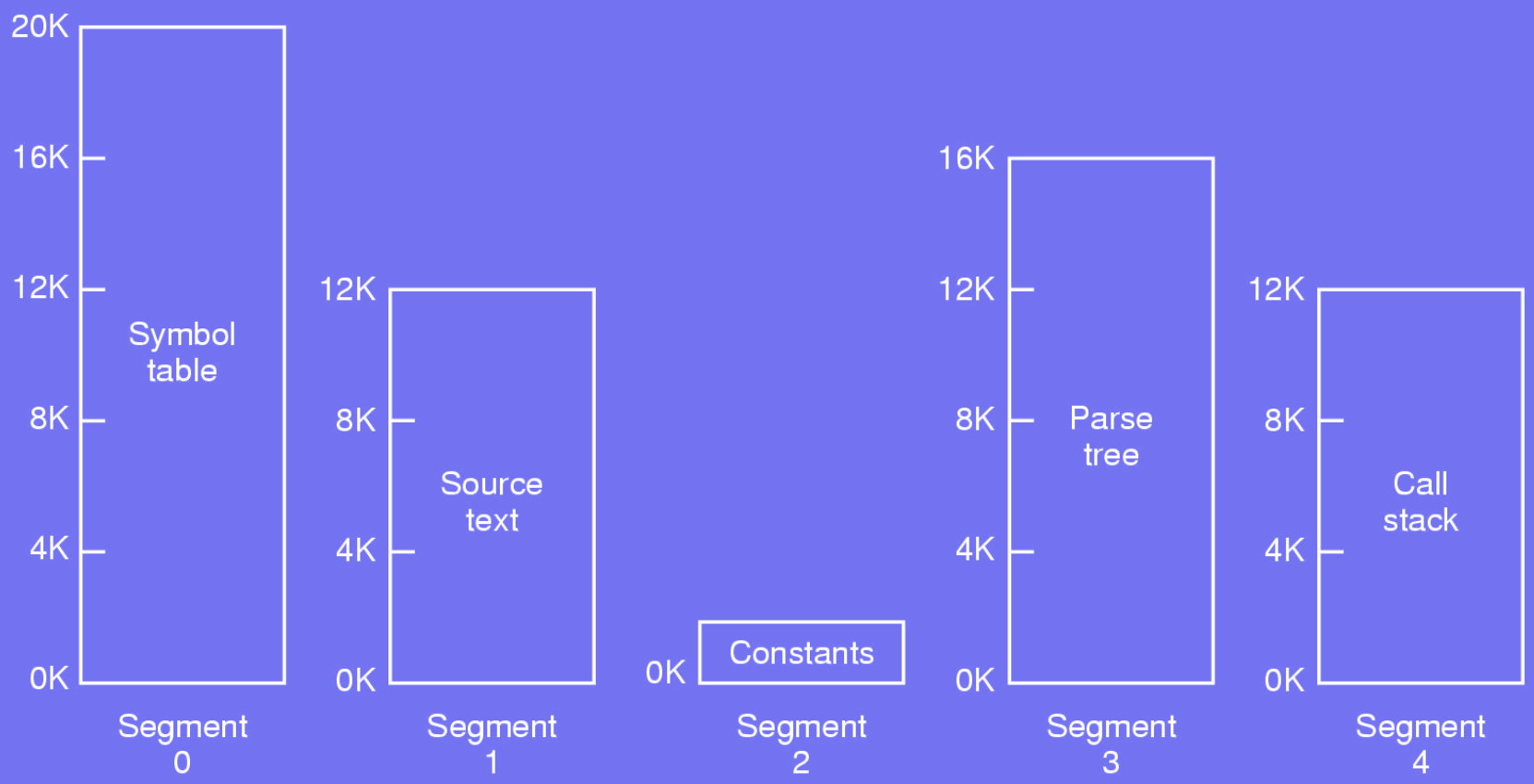
Paging



Basic paging summary:

- One dimension
- Starting address and a limit
- Example: compiler
- If many variables: symbol table expands on source text

Segmentation

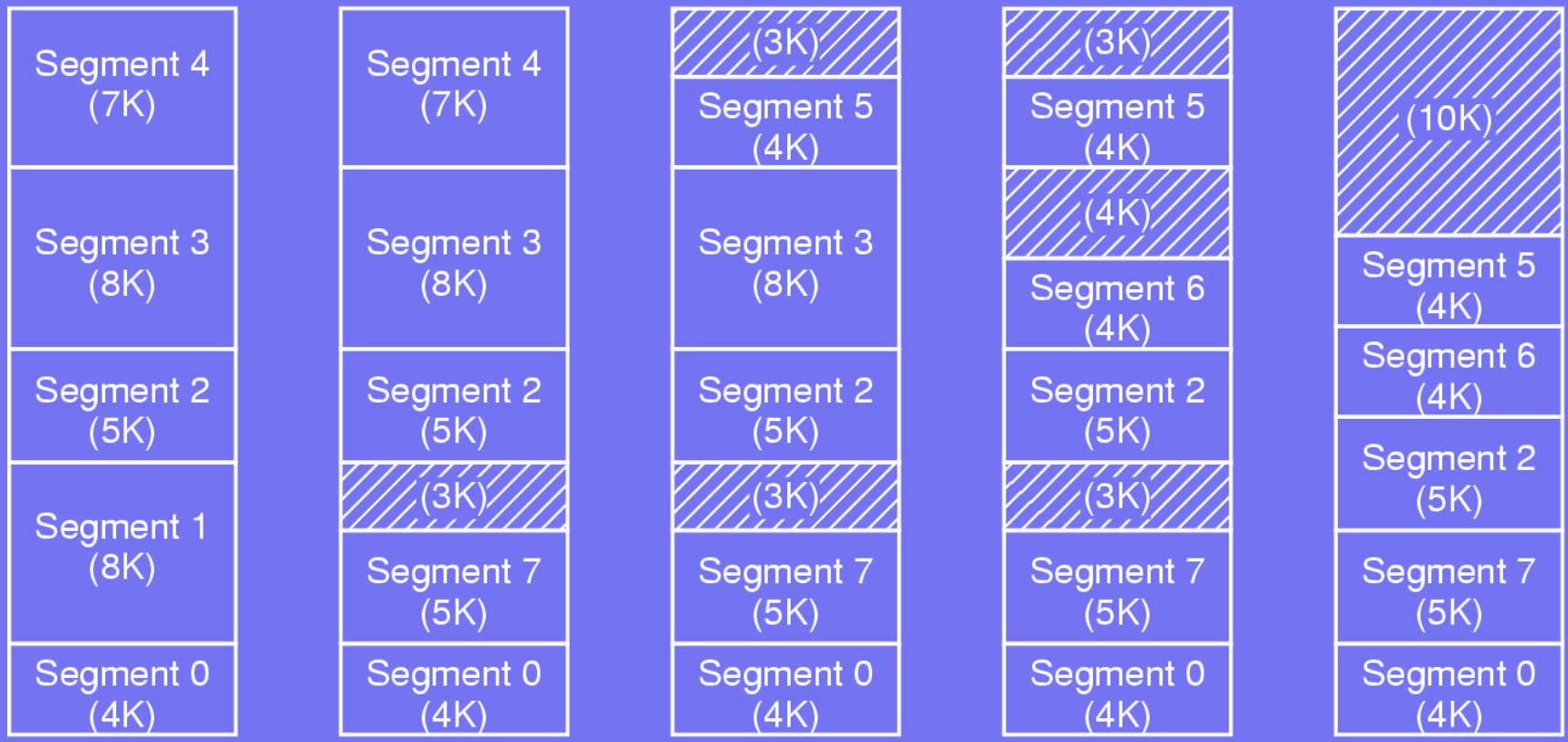


Address translation

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

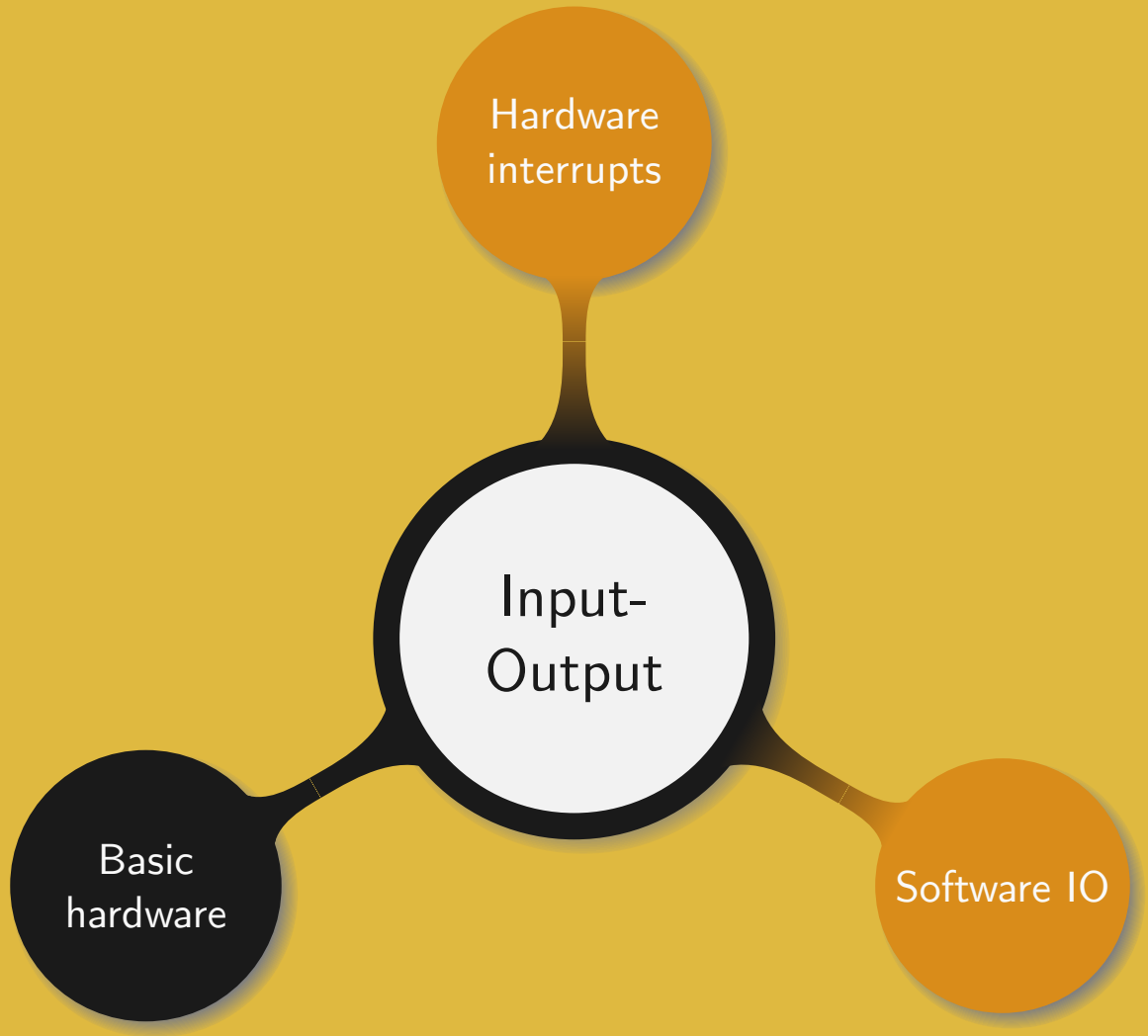
Considerations	Paging	Segmentation
Number of linear address space	1	many
Limited by the size of the RAM	no	no
Possible to separate and protect data and procedures	no	yes
Sharing procedures between users or programs	complex	easy

Key points

- What are the two main ways to model memory?
- What is the swap area?
- Cite two main page replacement algorithms
- Discuss the differences between paging and segmentation
- Explain external and internal fragmentation

7. Input-Output

Chapter organisation



OS and Input-Output

The OS controls all the IO:

- Send commands to the device
- Handle errors
- Catch interrupts

Two main device 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
- Clock: cause interrupts at some given interval

The OS provides a simple way to interface with hardware

Device controller

Most devices have two parts:

- Mechanical: the device itself
- Electronic: the part allowing to communicate 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 IO

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 IO port

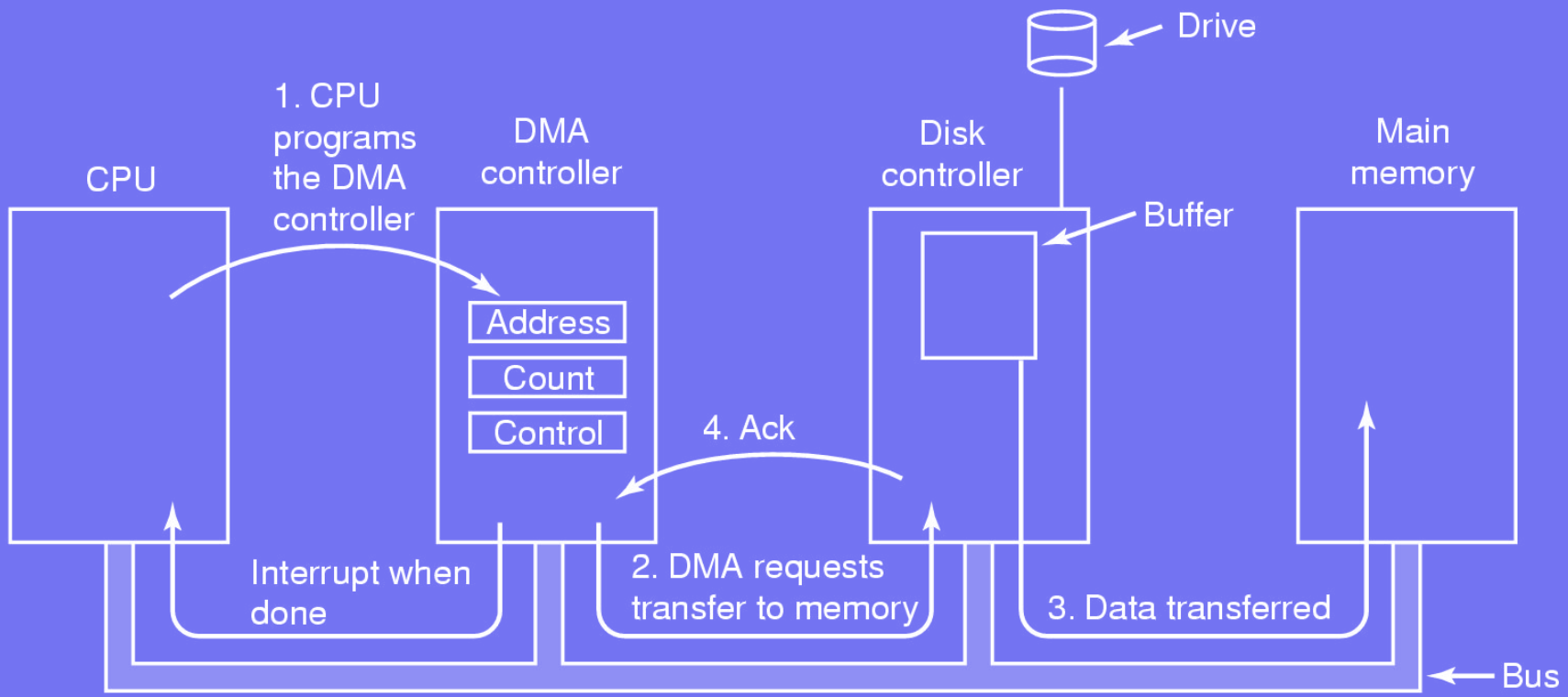
Memory-mapped IO

Strengths:

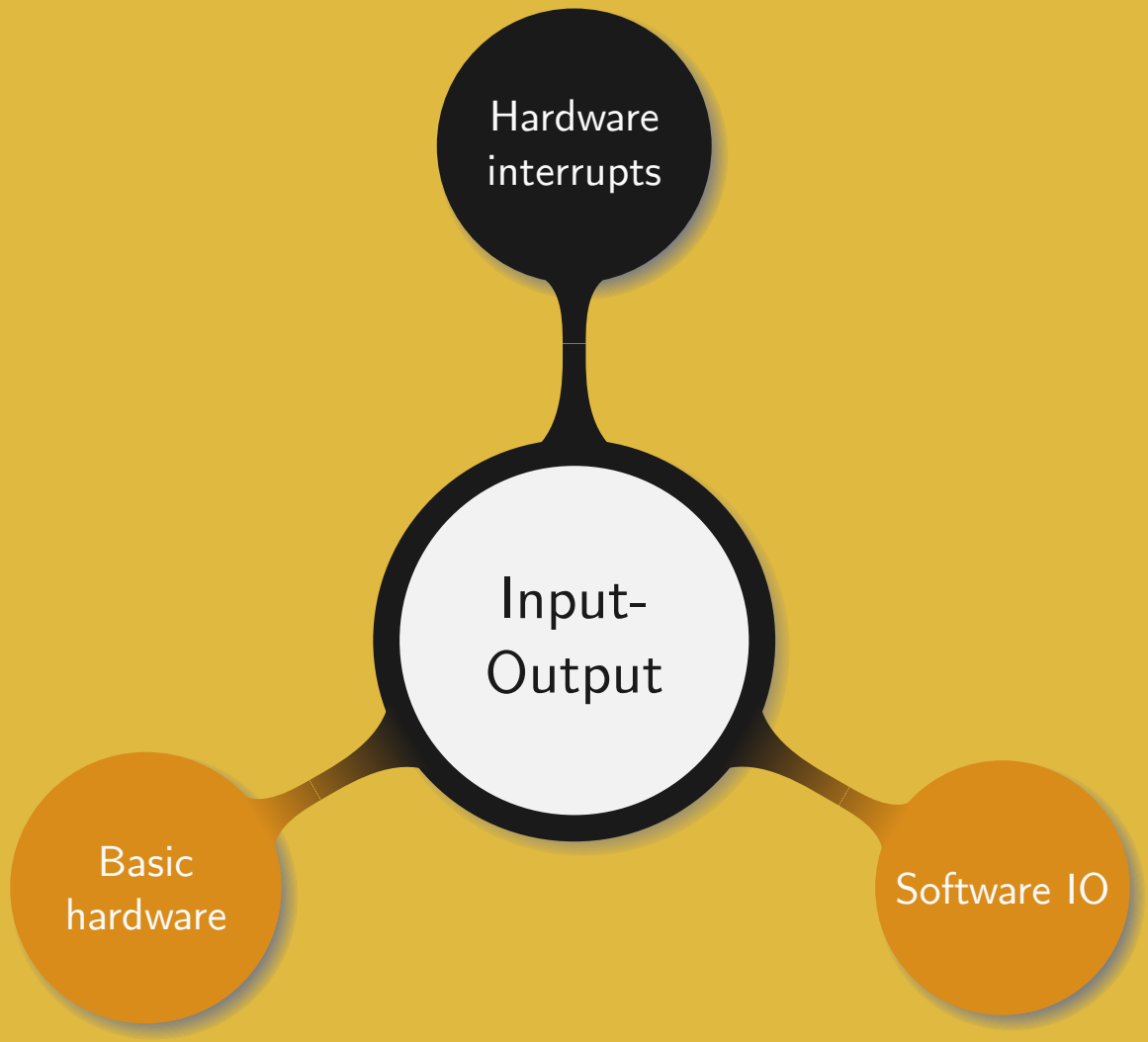
- Access memory not hardware: no need for assembly
- No special protection required: control register address space not included in the virtual address space
- Flexible: a specific user can be given access to a particular device
- Different drivers in different address spaces: reduces kernel size and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

Since memory words are cached what if the content of a control register is cached?

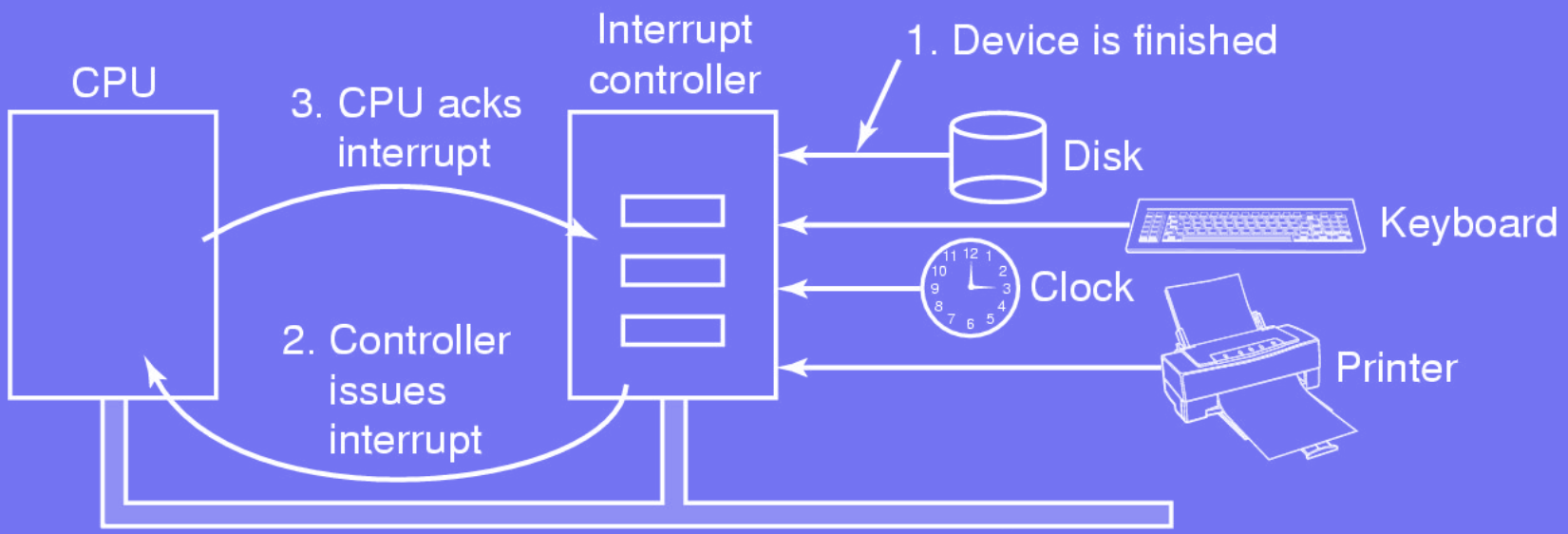
Direct Memory Access



Chapter organisation



Reminders


CPU
3. CPU acks interrupt
Interrupt controller
1. Device is finished
Disk
Keyboard
Clock
Printer
2. Controller issues interrupt

Precise interrupts

Interrupt handling on a basic setup:

- Push program counter (PC) and PSW on the stack
- Handle the interrupt
- Retrieve program counter and PSW from the stack
- Resume a process

Precise interrupt:

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

What are the consequences of more advanced architectures such as pipelined or superscalar CPU?

Imprecise interrupts

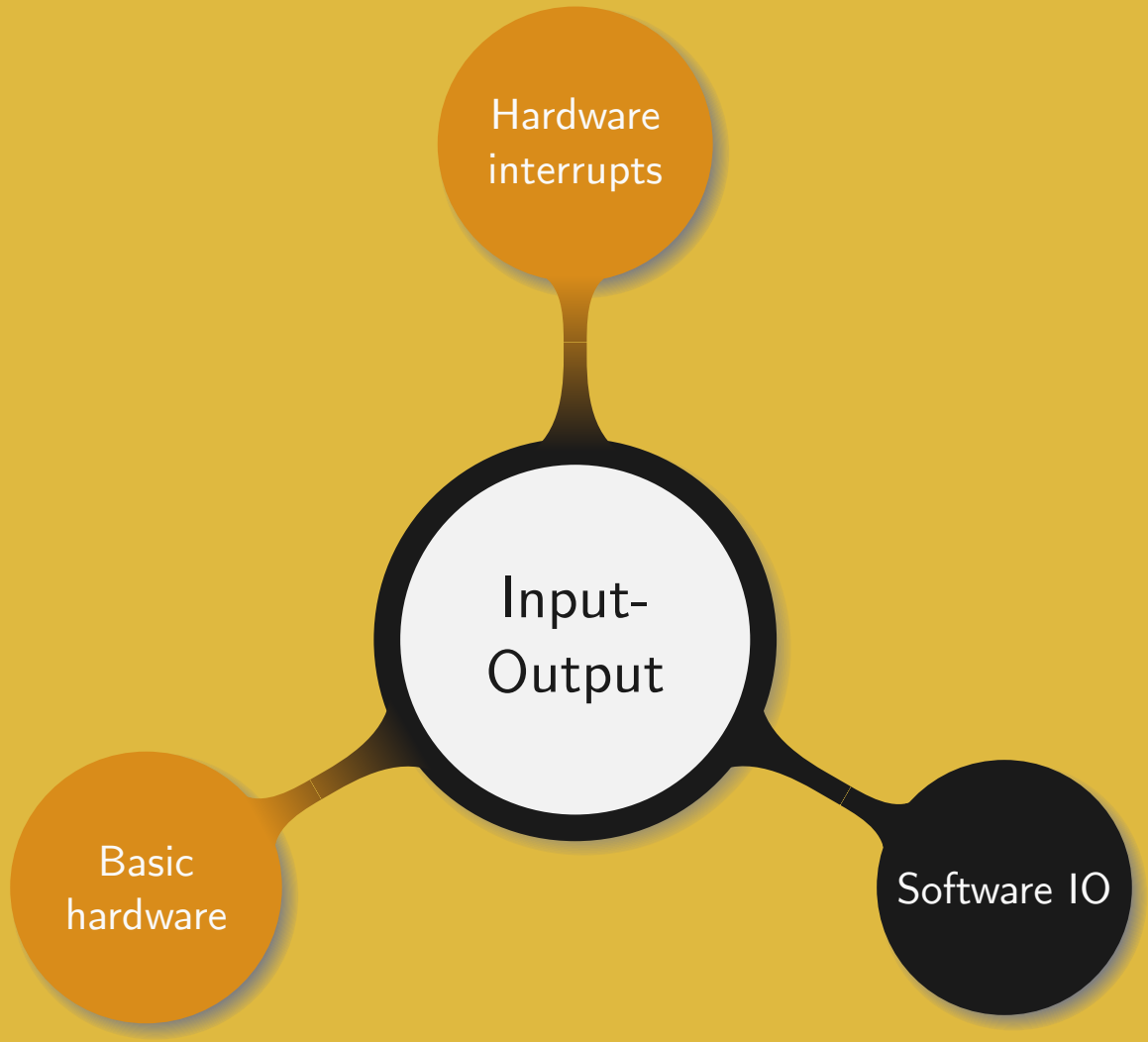
An interrupt which is not precise is called imprecise interrupt

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
- Code to resume process is complex
- The more details the more memory used

Imprecise interrupts are slow to process, what is an optimal strategy?

Chapter organisation



Communications strategies

Three communications strategies:

- Programmed IO:
  - Copy data into kernel space
  - Fill up device register and wait in tight loop until register is empty
  - Fill up, wait, fill up, wait, etc.
- Interrupt IO:
  - Copy data into kernel space, then fill up device register
  - The current process is blocked, the scheduler is called to let another process run
  - When the register is empty an interrupt is sent, the new current process is stopped
  - Filled up, block, resume, fill up, block, etc.
- DMA: similar to programmed IO, but DMA does all the work.

Key concepts

Main goals on the design of IO 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

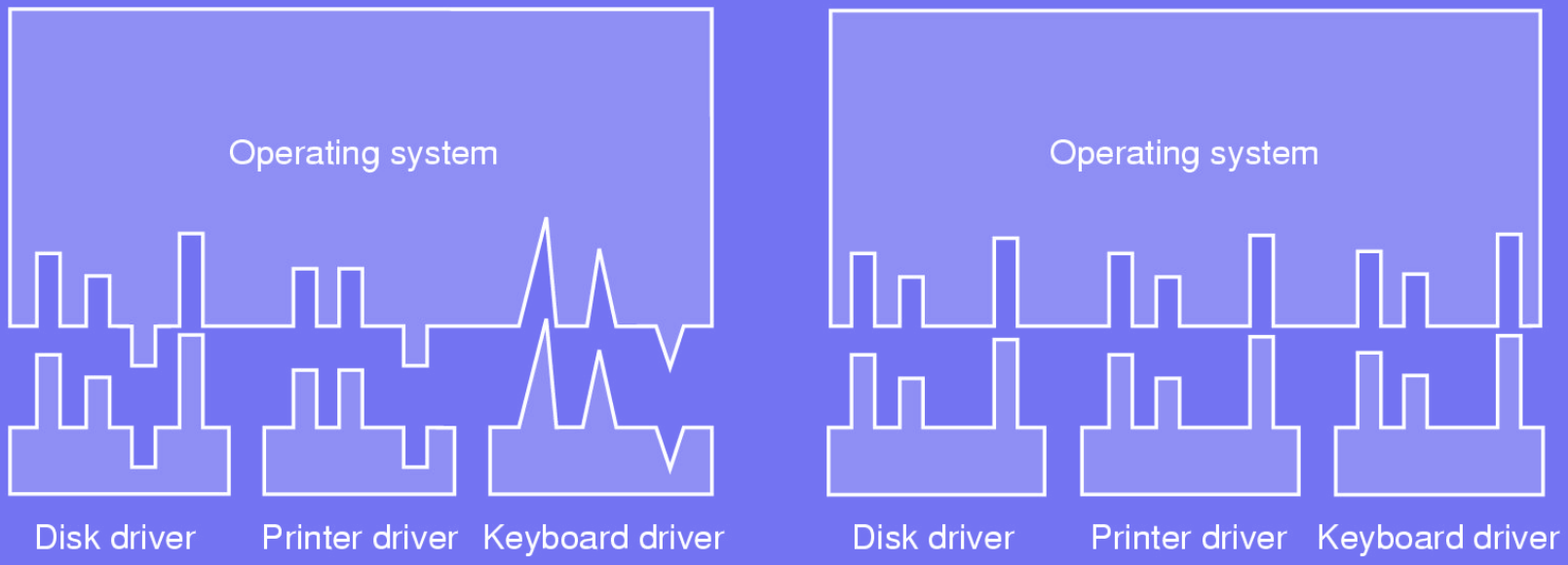
Basic layers in an OS

Interrupt handlers

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



Basic plugin design strategy:

- Similar devices have 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

Job of device drivers

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

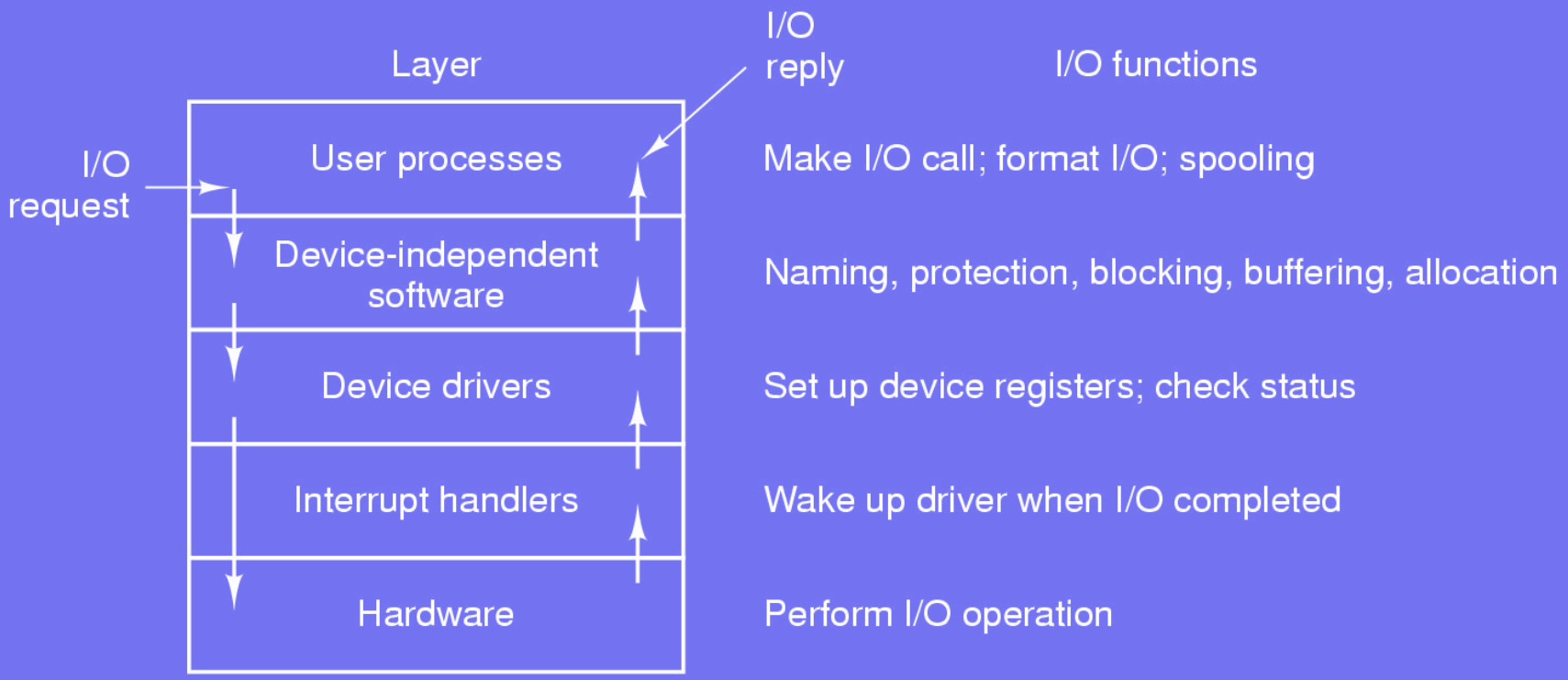
Device driver should react nicely even under special circumstances

Drivers in the OS

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, e.g. to manage MMU, set timers
- IO errors framework is device independent
- Clean and generic API such that it is easy to write new drivers

Software IO and OS

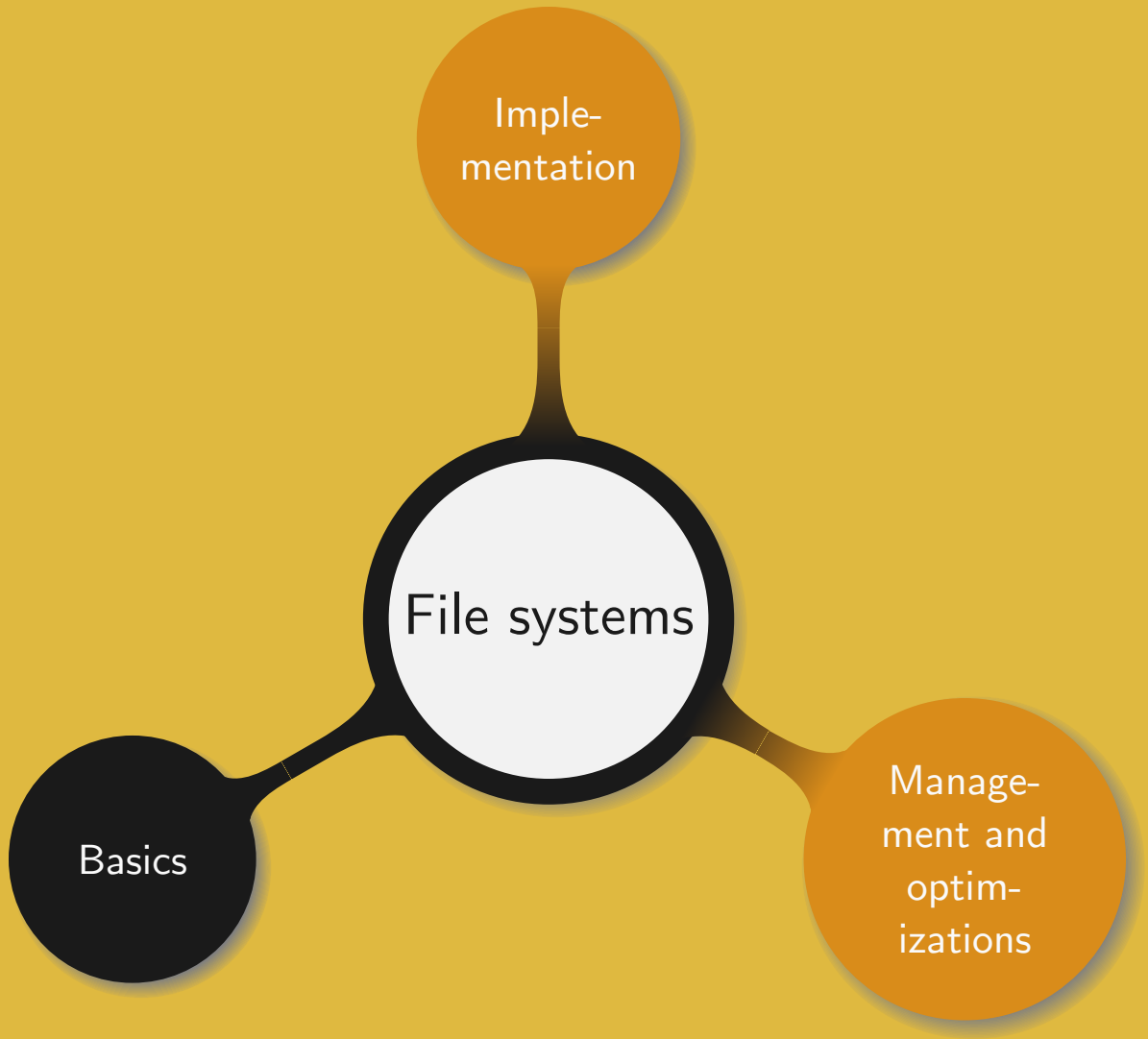


Key points

- What are the two main categories of devices?
- Explain the difference between precise and imprecise interrupts
- List three communication strategies
- Why should implementation be done using layers?
- What is a plugin architecture and why is it so important?

8. File systems

Chapter organisation



Goals

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

User's view

High level view of a file-system:

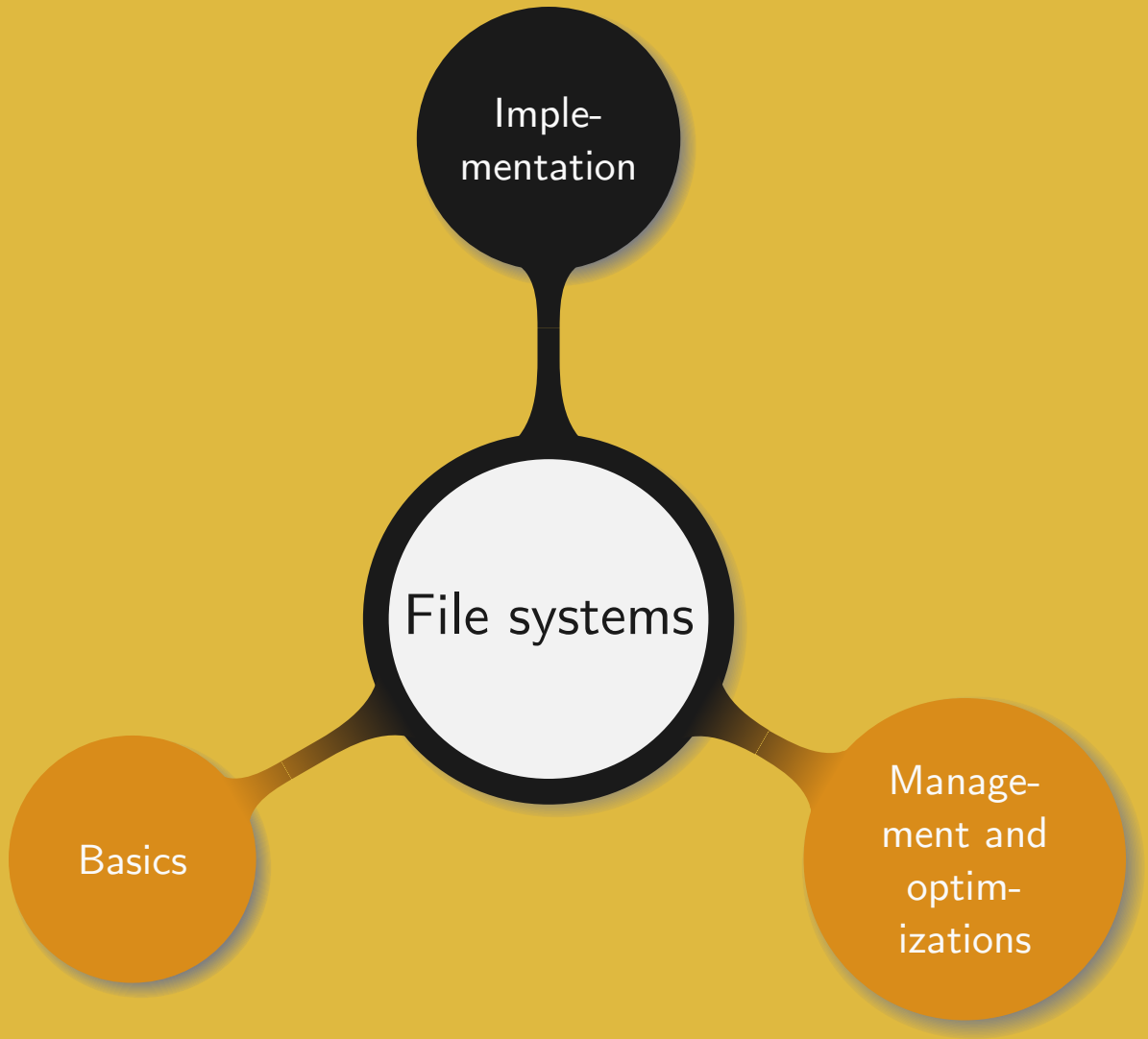
- Small part of the disk memory can be directly accessed using high level abstraction called a *file*
- File name can be case sensitive or insensitive
- File name is a string with (an optional) suffix
- Each file has some attributes containing special information

File organisation

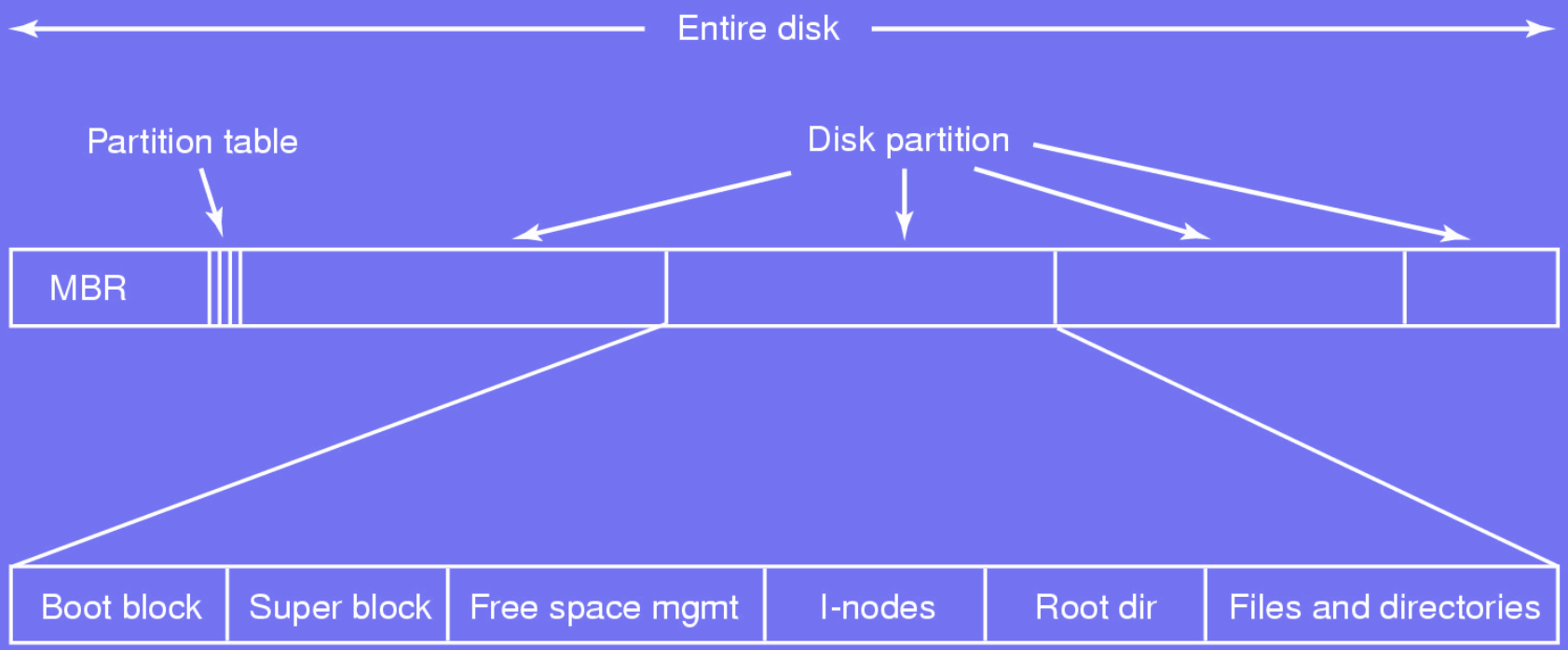
Structure content:

- 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

Chapter organisation

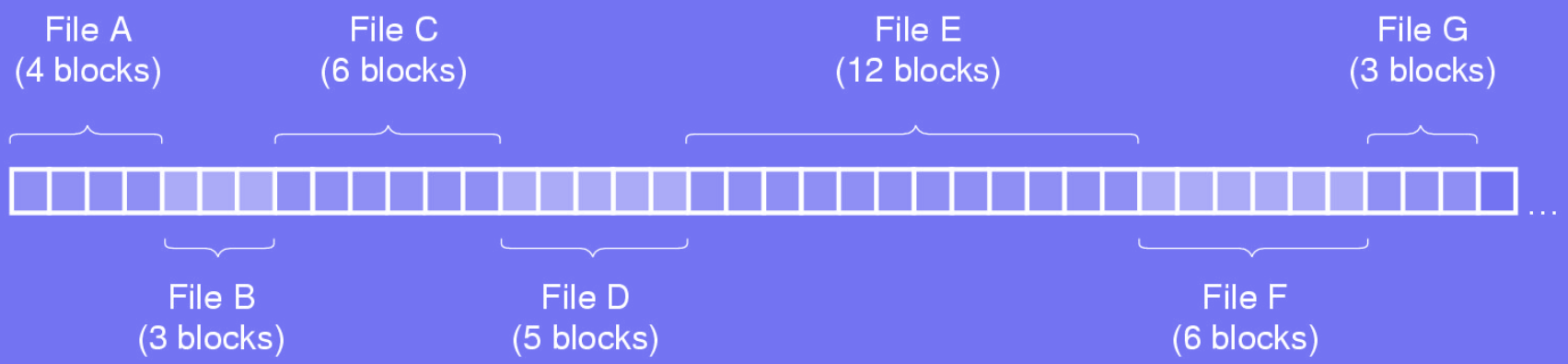


Basic disk layout



How to efficiently match disk blocks and files?

Contiguous allocation

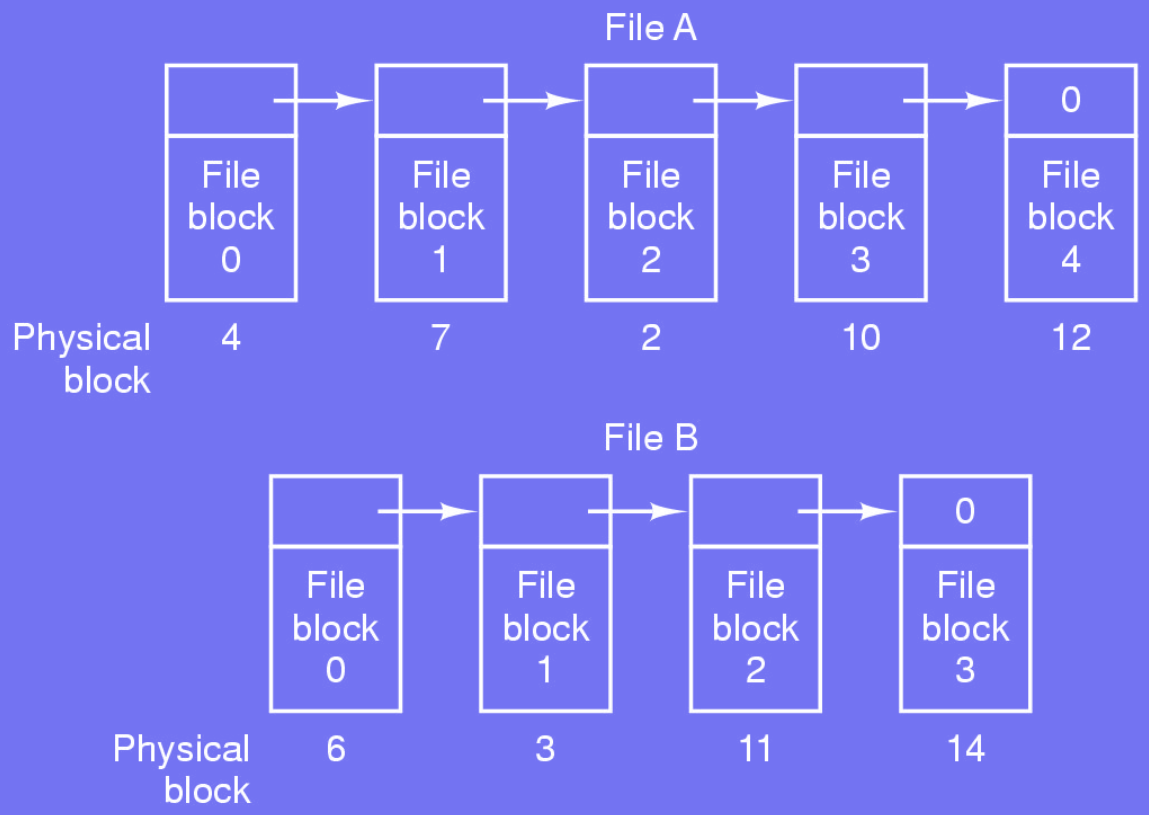


Advantages:

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

What if files D and F are deleted?

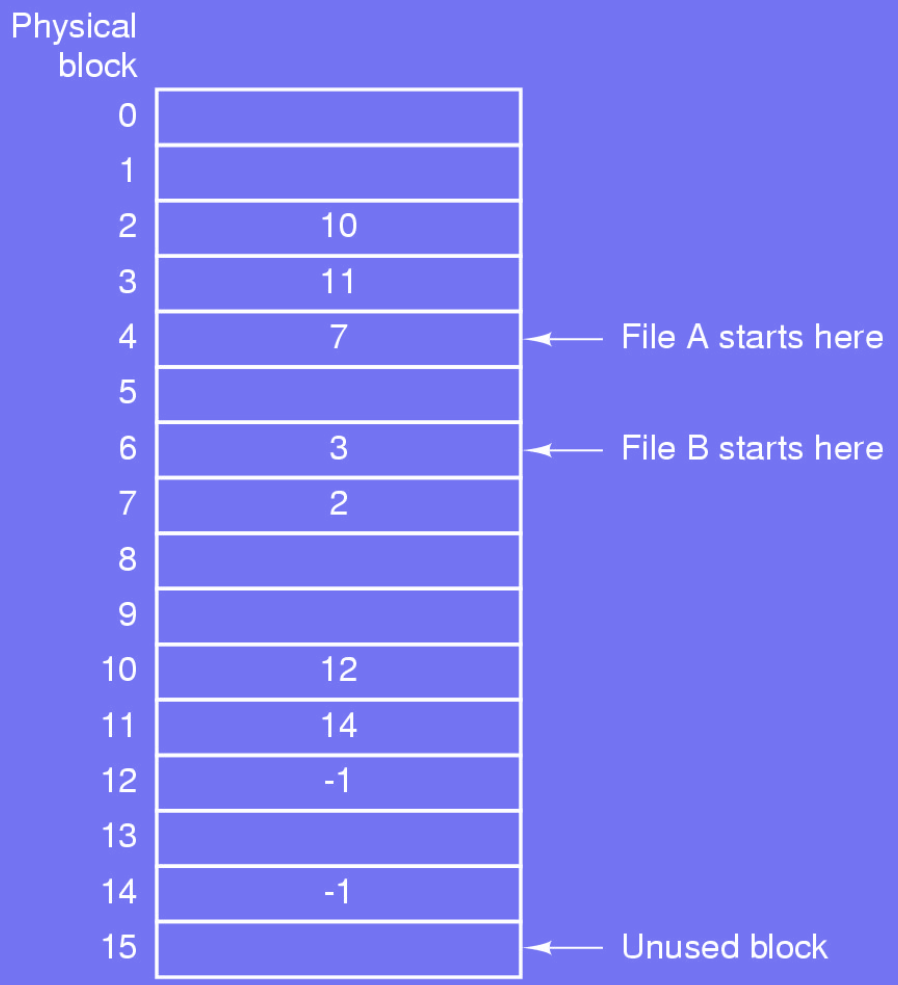
Linked list



Advantage: no fragmentation

How fast is random access?

File Allocation Table



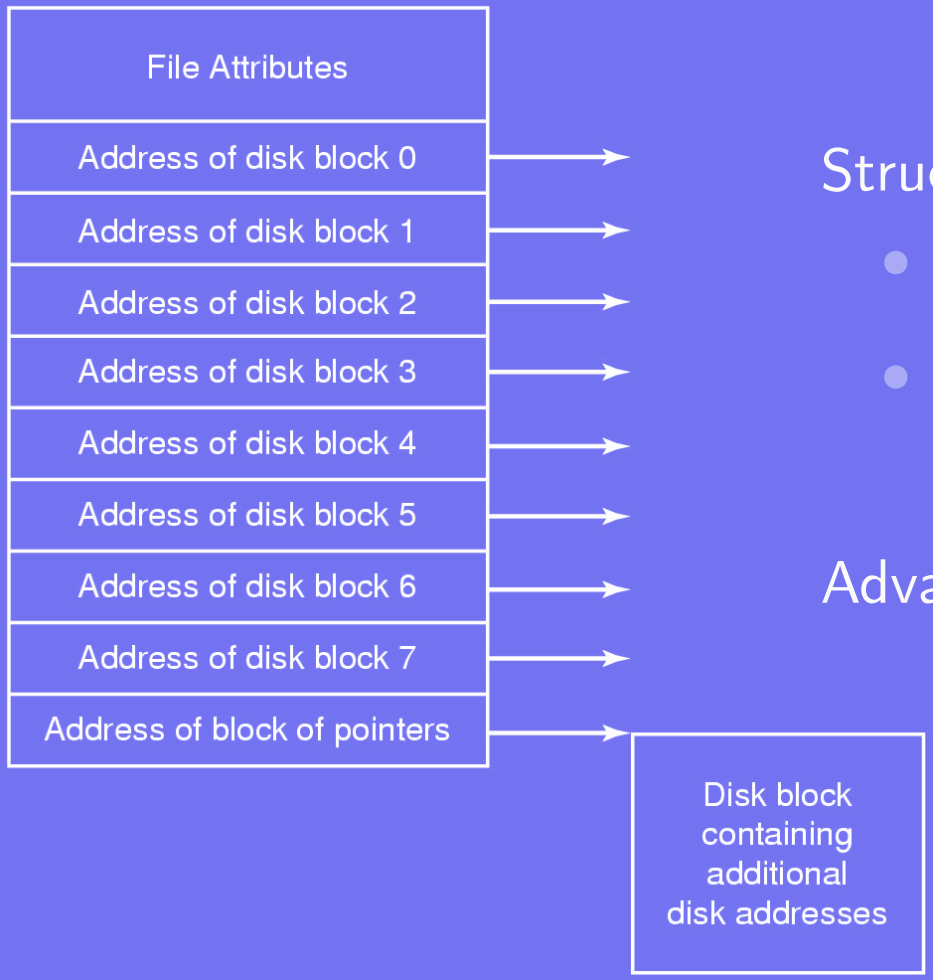
Basic idea:

- Have a pointer for each disk block
- Store all the pointers in a table
- Save the table in the RAM

Advantage: fast random access

What can be said about the memory usage?

Index node



Structure storing:

- The file attributes
- Pointers on the blocks where the file is written

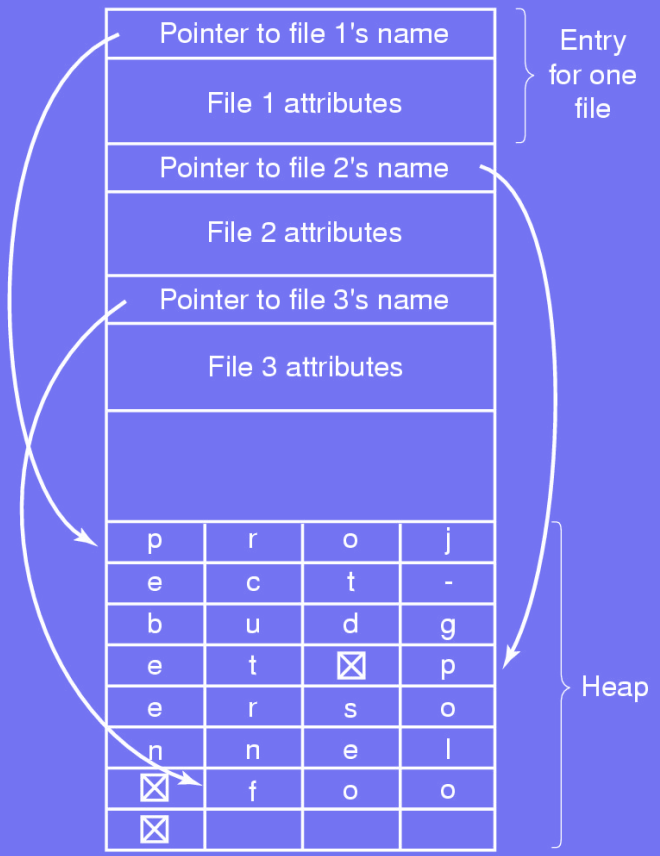
Advantage: fast, requires little memory

What if a large file needs more blocks that can fit in an inode?

Directories

What is the issue with the following strategies?

- Fixed filename size
- Arbitrary long filename size



Advantages of string pointers:

- No space wasted
- Space can be easily reused when a file is removed

How fast is this strategy?

Journaling FS

Keep a journal of the operations:

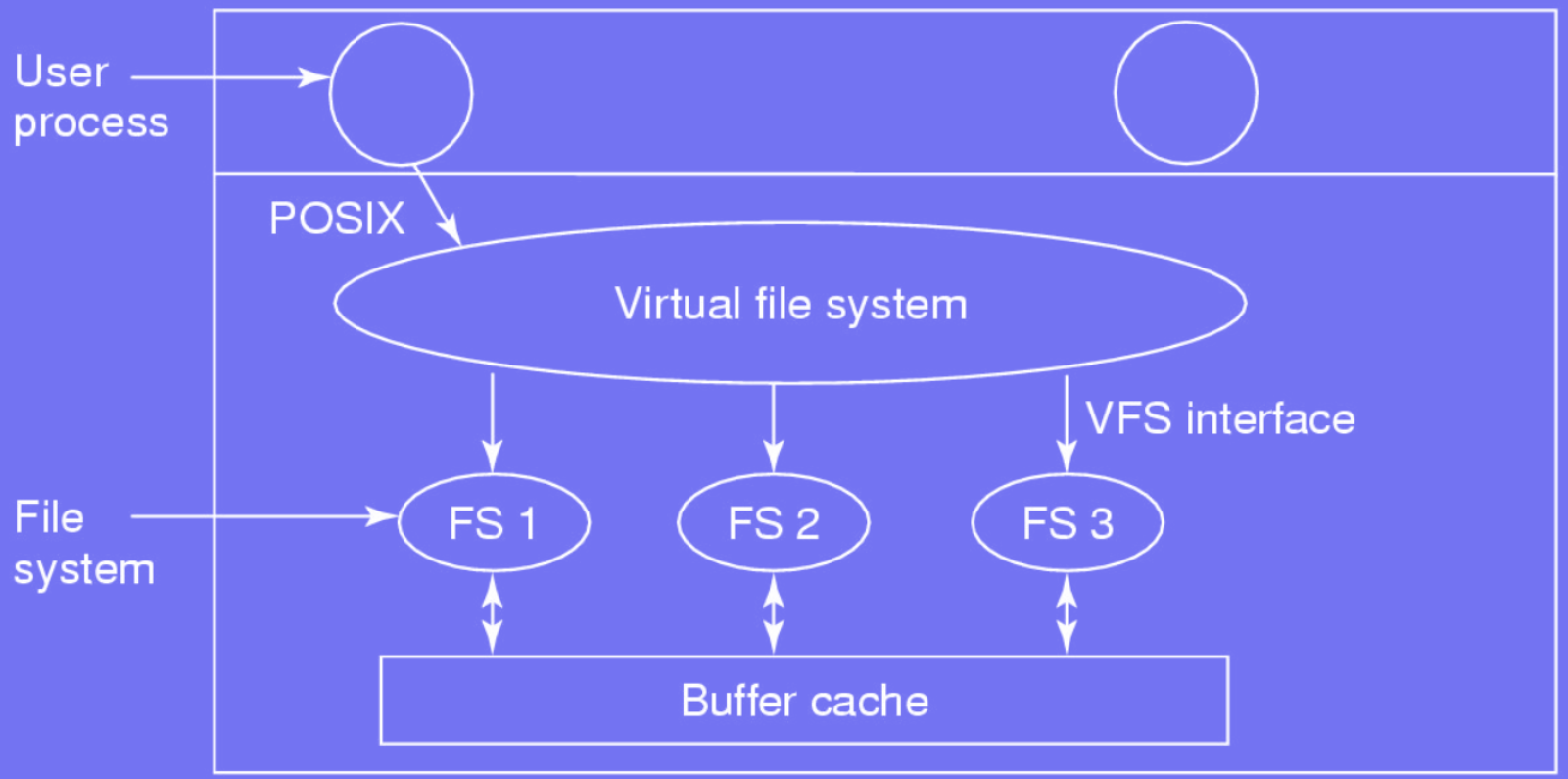
- Log the operation to be performed, run it, and erase the log
- If a crash interrupts an operation, re-run it on next boot

Can any operation be applied more than once?

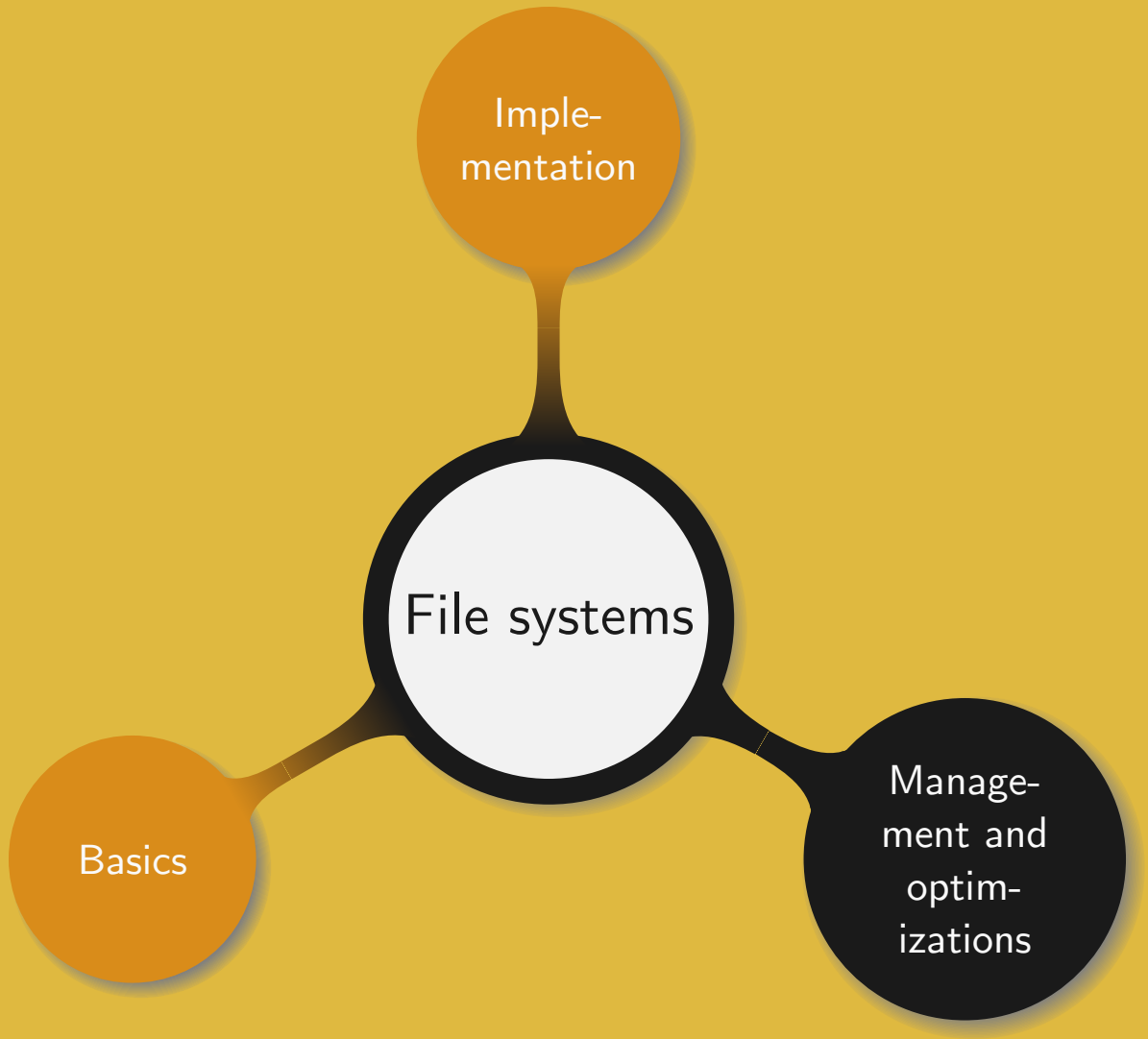
Example. Which of the following operations can be safely applied more than once?

- Remove a file from a directory
- Release a file inode
- Add a file disk blocks to the list of free blocks

Virtual FS


User process
POSIX
Virtual file system
VFS interface
File system
FS 1
FS 2
FS 3
Buffer cache

Chapter organisation



Block size

Using small blocks:

- Large files use many blocks
- Blocks are not contiguous

A small block size leads to a waste of time

Using large blocks:

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

A large block size leads to a waste of space

Free blocks

Keeping track of the 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?

File system consistency

Checking an inode based FS:

- Using the inodes: list all the blocks used by all the files and compare the complementary to the list of free blocks
- For every inode in every directory increment a counter by 1 and compare those numbers with the counts stored in the inodes

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 several files: copy block and add it to the files
- File related inconsistency:
  - Count in inode is higher: set link count to accurate value
  - Count in inode is lower: set link count to accurate value

Caching

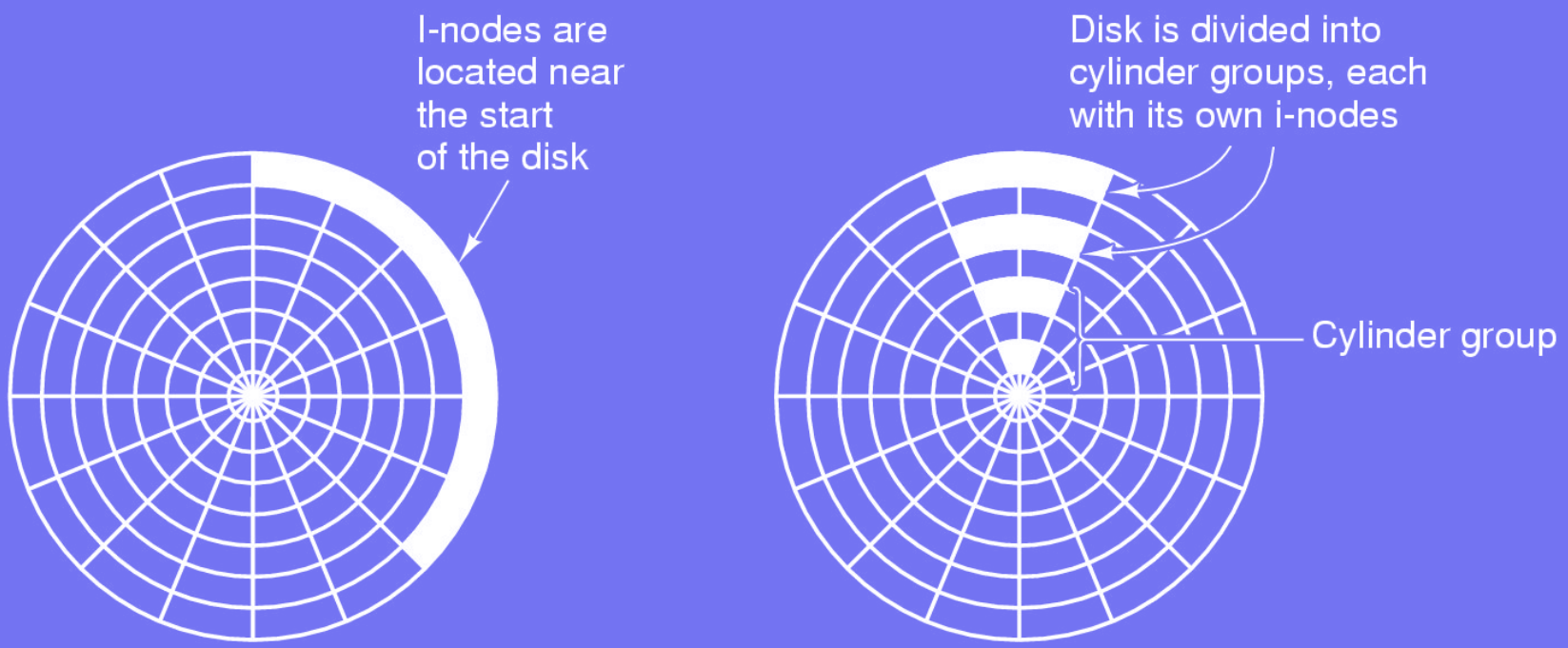
Keep in memory some disk blocks using the LRU algorithm:

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

Refined idea:

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

Arm motion


I-nodes are
located near
the start
of the disk
Disk is divided into
cylinder groups, each
with its own i-nodes
Cylinder group

Notes

A few extra remarks related to file systems:

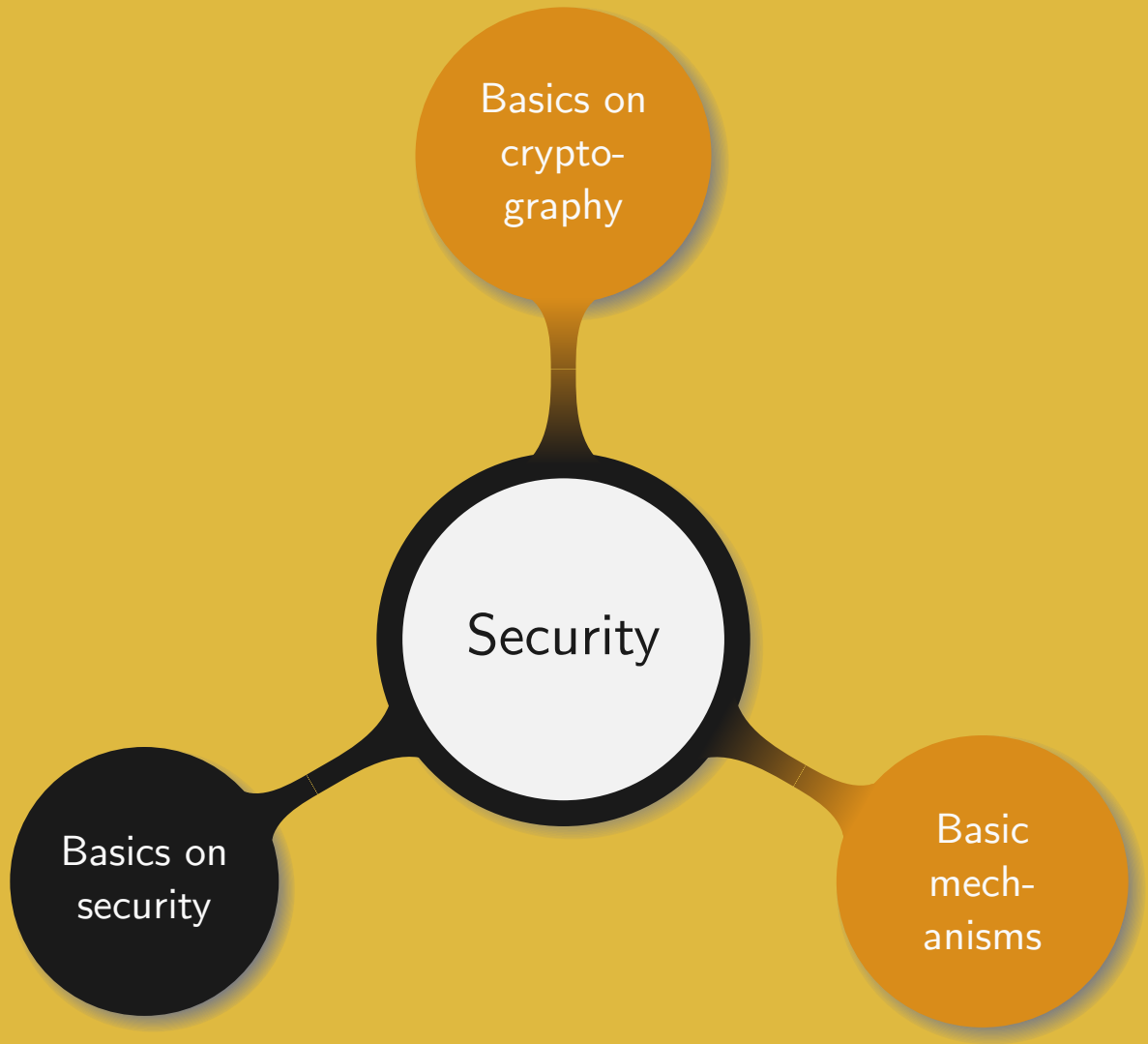
- Quotas: assign disk quotas to users
- Fragmentation: how useful is it to defragment a file system?
- Block read ahead: when reading block k assume $k + 1$ will soon be needed and ensure its presence in the cache
- Logical volumes: file system over several disks
- Backups: how to efficiently backup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

Key points

- What are the three main goals of a file system?
- Describe a basic disk layout
- Explain the structure of an inode
- Mention three challenges in the design of a file system

9. Security

Chapter organisation



Security setup and dangers

Simple reasoning:

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

To define the dangers, the setup must be known:

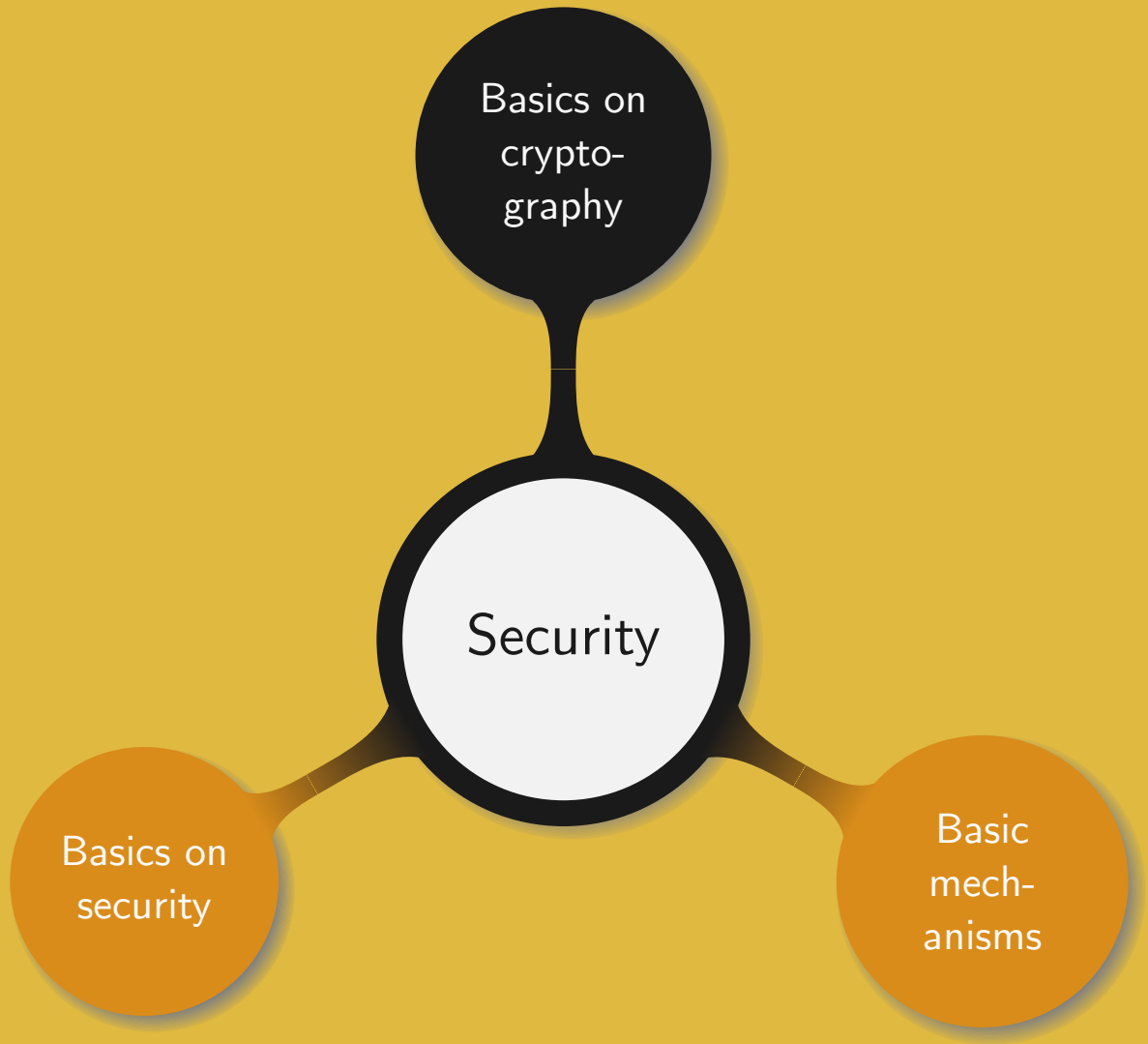
- General setup: operating system
- Processes: privileges
- Memory: sensitive information processed
- IO devices: intruders
- File system: sensitive data

Threats

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

Chapter organisation



Security and cryptography

Cryptography, the science of secret:

- Confidentiality
- Data integrity
- Authentication
- Non-repudiation

Two basic encryption strategies:

- Symmetric: same key used to encrypt an decrypt
- Asymmetric: many can encrypt but only one can decrypt

In an OS setup:

- Symmetric protocols best fit confidentiality
- Asymmetric protocols best fit authentication

Data integrity and authentication

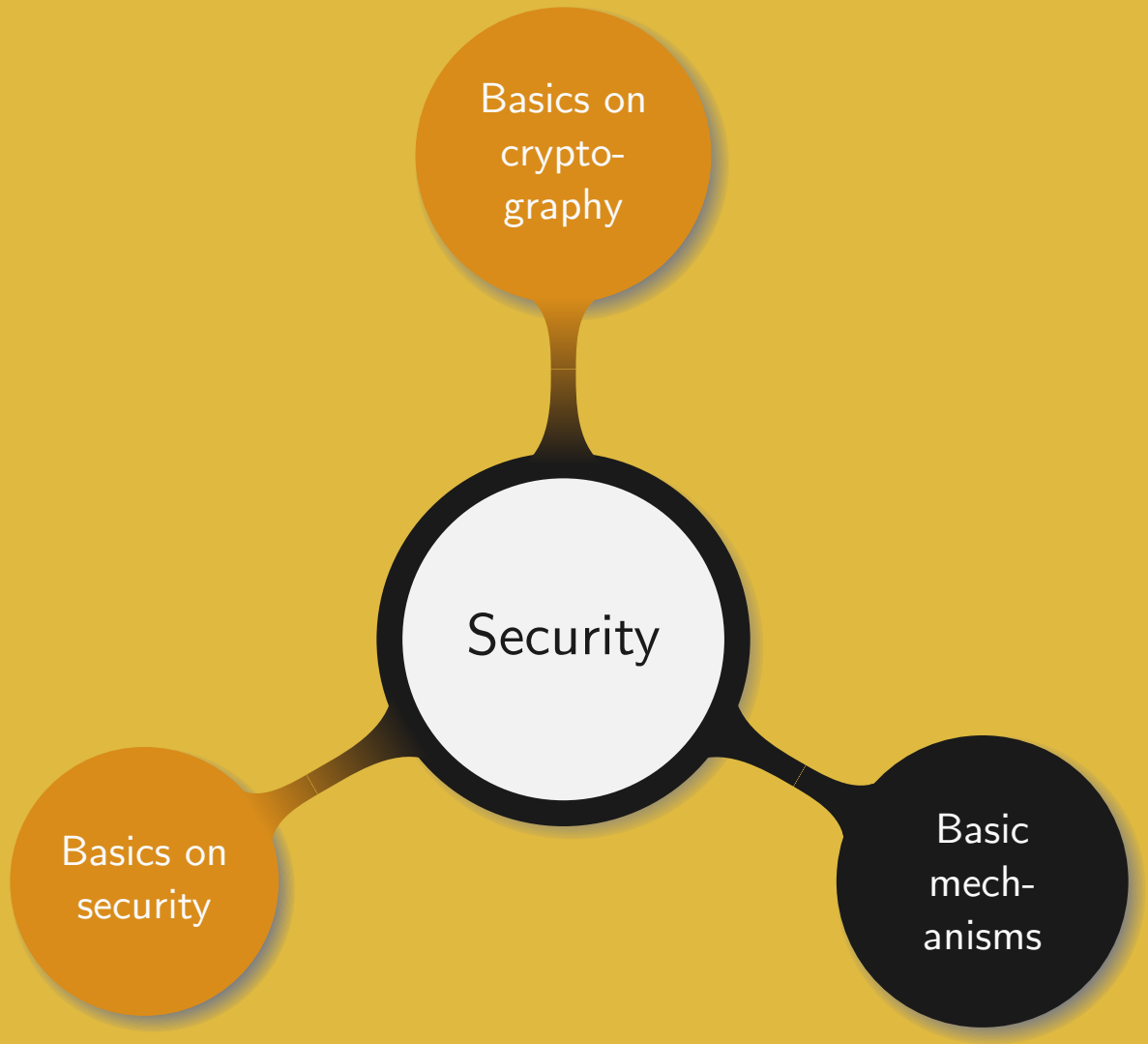
Ensure that data has not been altered using hash functions:

- 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



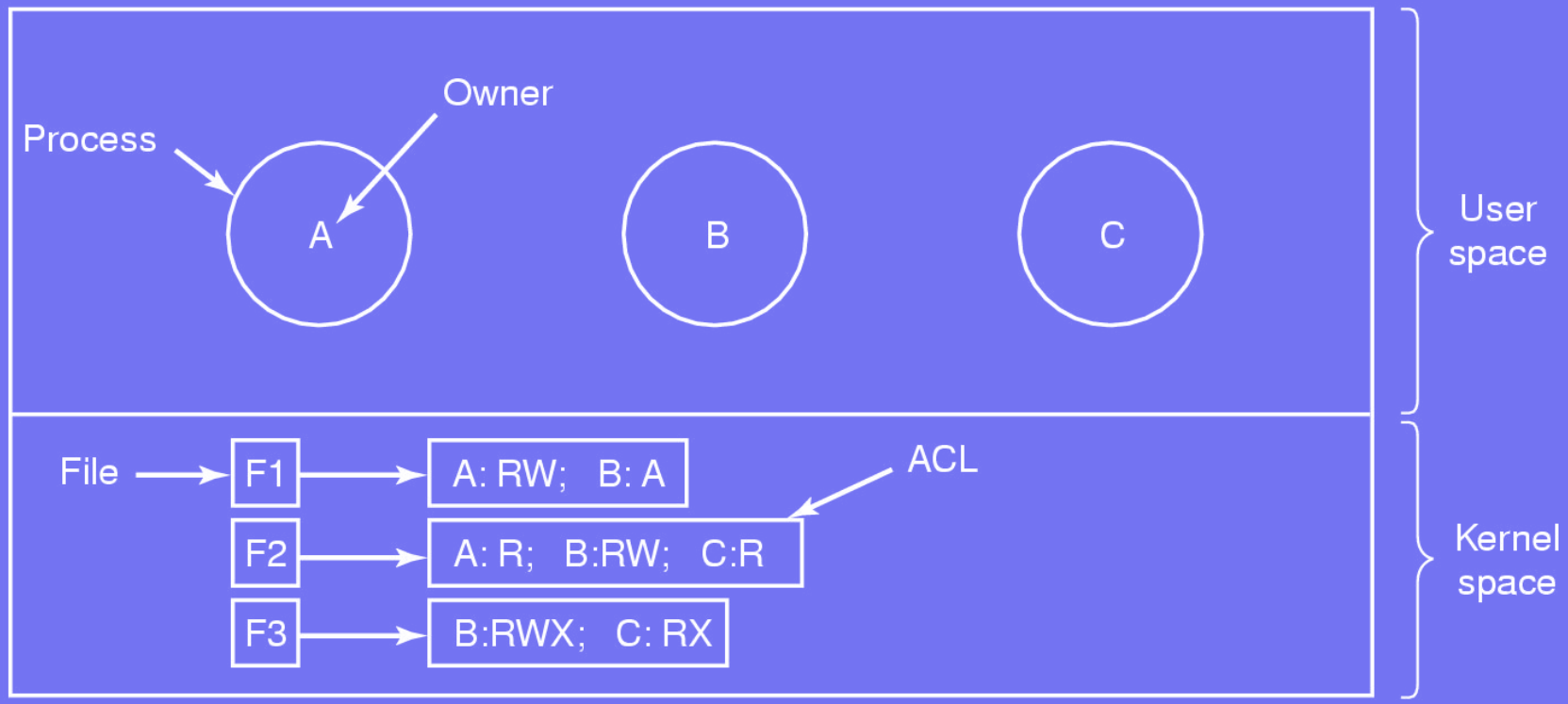
Authentication

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 of after the password input?
- What if the hard disk is mounted from another OS?

Solutions based on asymmetric cryptography are safer

Access Control Lists



Access control lists are used to give users different privileges:

- Administrator: root|admin
- Privileged users: belong to special groups
- Regular users cannot access IO devices

Keeping a system secure – Basic

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
- Physically block all the ports, no external device can be connected

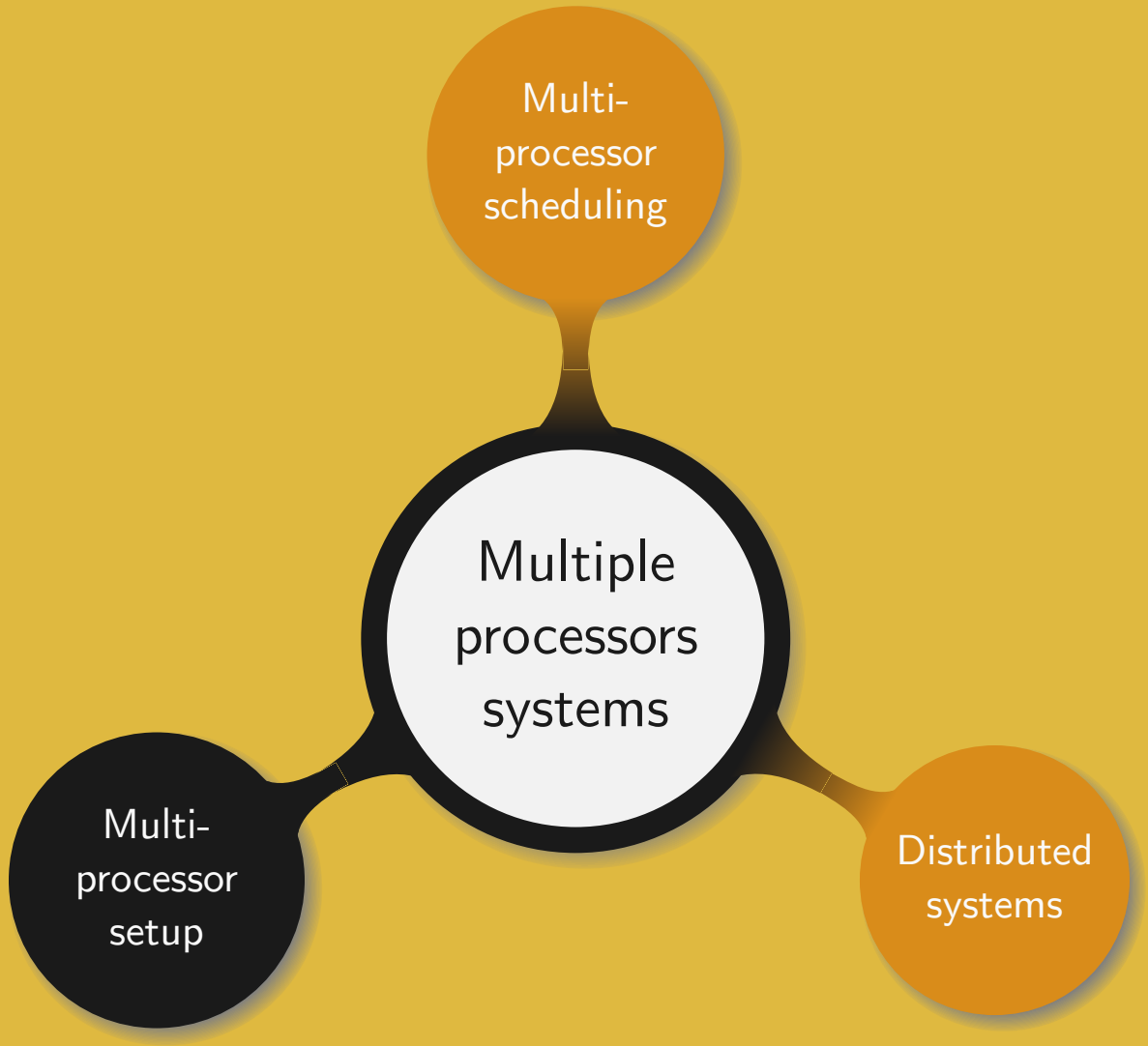
Can this setup be considered safe?

Key points

- What is security?
- What are the two main types of encryption?
- What are Access Control Lists?
- What is the safest authentication method?
- How safe can a computer be?

10. Multiple processors systems

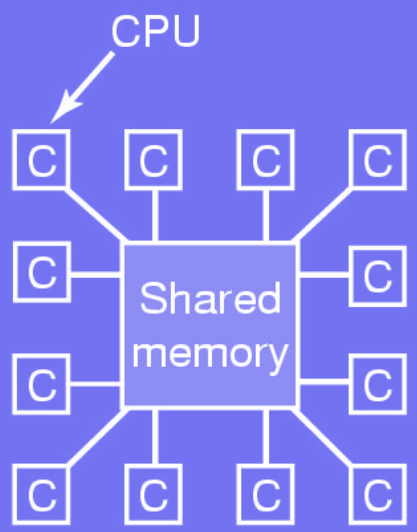
Chapter organisation



Shared memory model

Multiprocessors:

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



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: one OS shared by all CPU

SMP approach

Problems likely to occur:

- More than one CPUs run the same process
- A same free memory page is claimed at the same time

Basic idea of the solution:

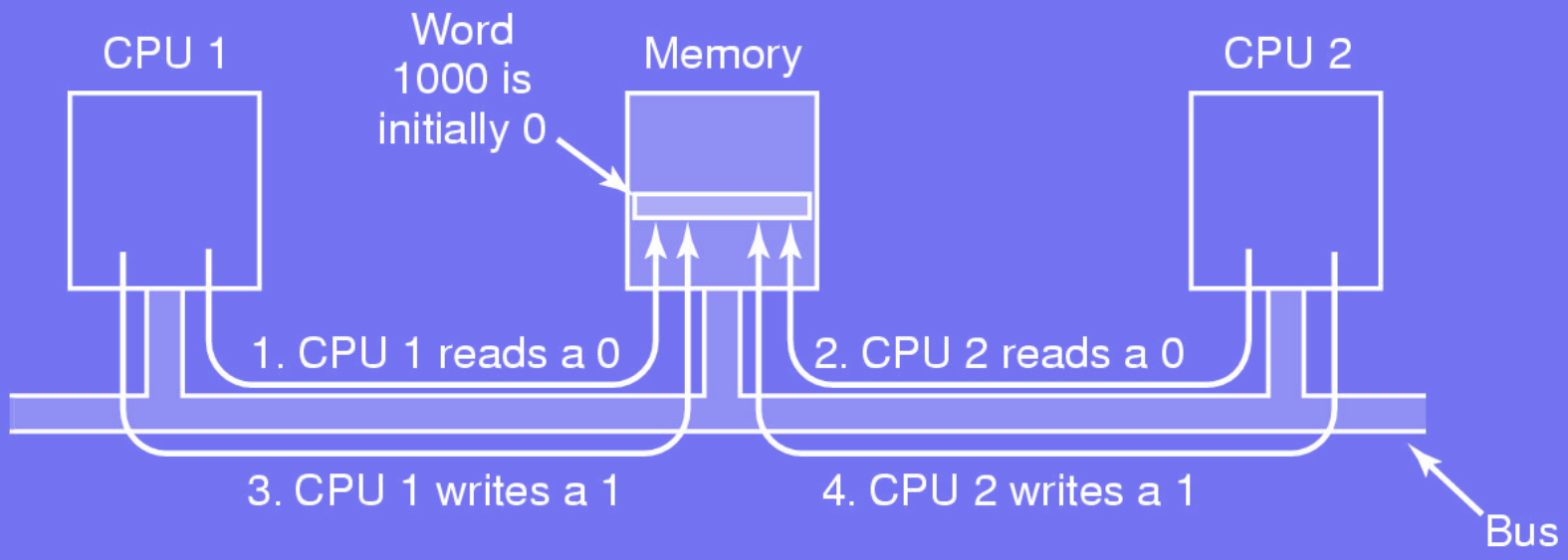
- Many parts of the OS are independent
- Split the OS into multiple critical regions
- Add a mutex when entering those regions
- Add mutex to all shared tables

The solution works but add complexity to the OS:

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

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



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

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

Read and TSL

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

More solutions

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:

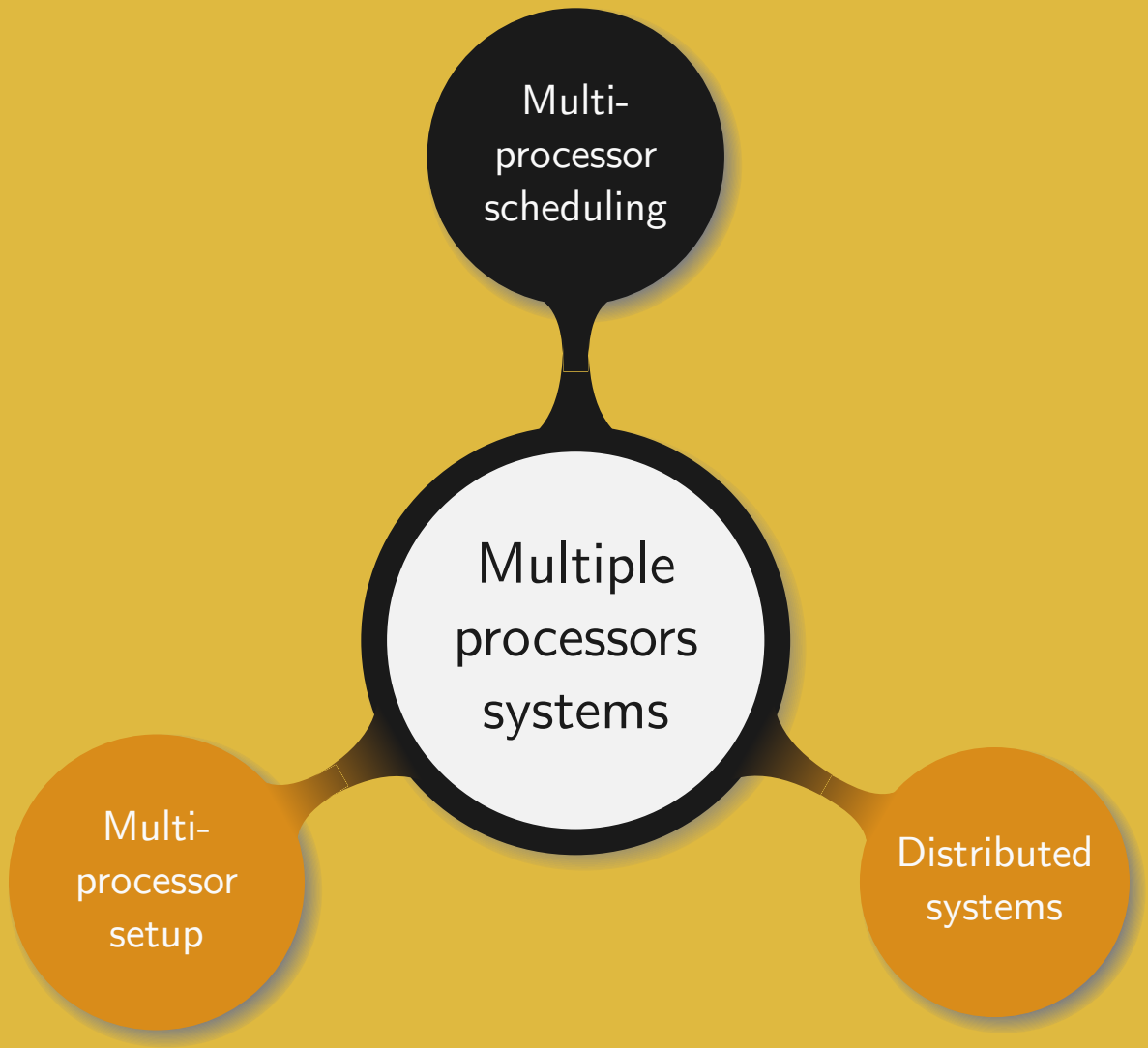
- On a uniprocessor: the time spent on waiting is wasted
- On a multiprocessor: one CPU is waiting while another works
- Switching is expensive but looping is a waste

There is no optimal solution:

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

Mix of waiting and switching with a (variable) threshold

Chapter organisation



Setup

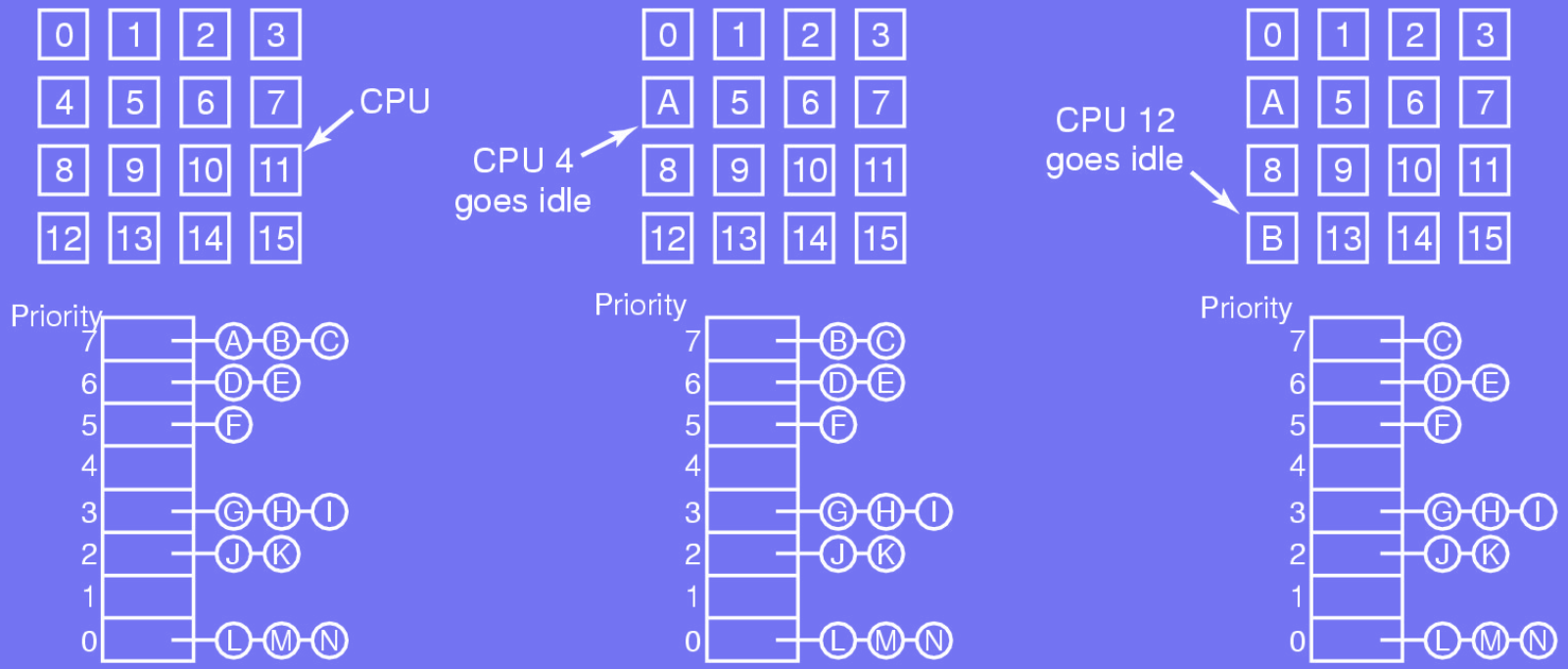
Single threaded process: only need to schedule the process

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 $\rightarrow$ no need to reload the whole process
- Threads of a process run in parallel $\rightarrow$ threads can cooperate more efficiently

Time-sharing



Advantages:

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

What if a thread holds a spin lock but reaches the end of its quantum?

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

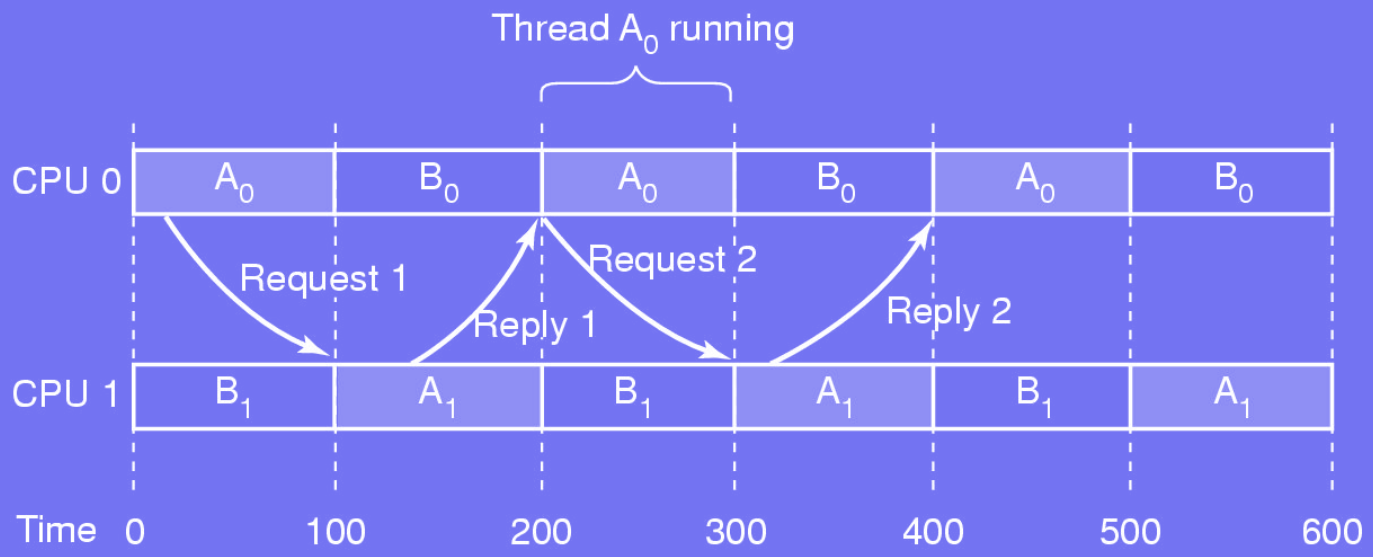
Space-sharing

When a process is created the scheduler checks if there are:

- More free CPU than threads: run a thread per CPU until completion
- More threads than free CPU: wait for more free CPU

What is the major drawback of this approach?

Gang scheduling

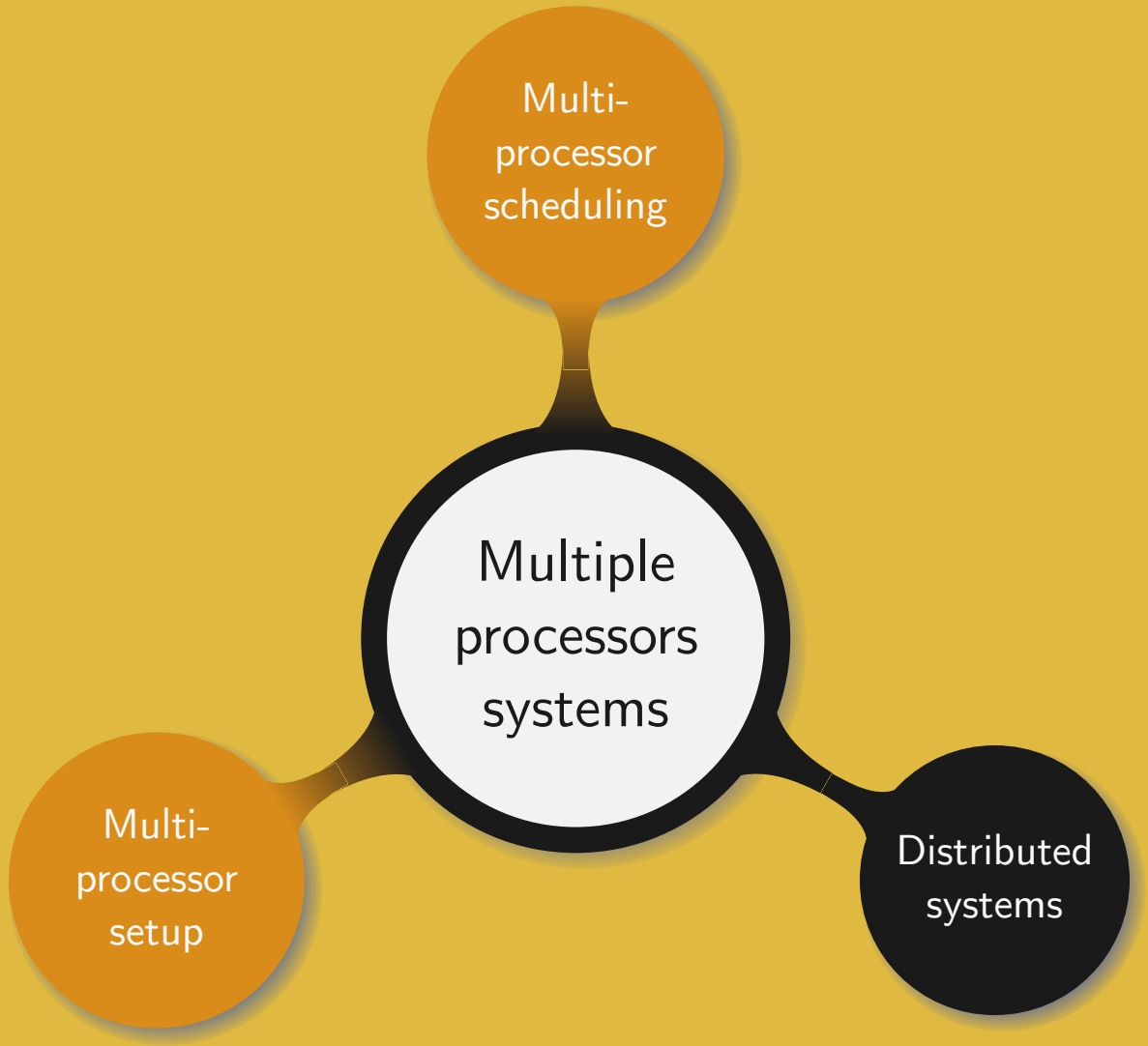


Gang scheduling schedules processes by group:

- 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

What if a member issues an IO request or finishes before others?

Chapter organisation

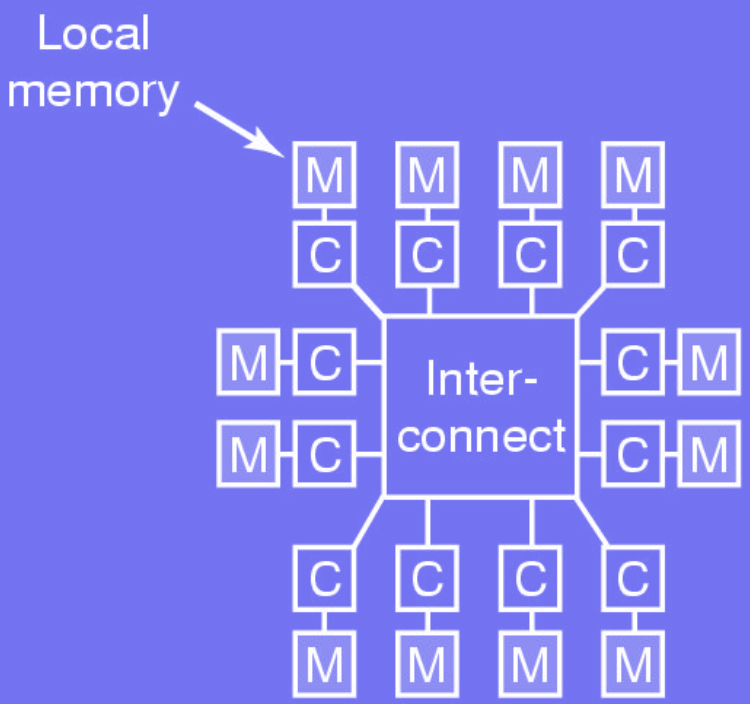


Main characteristics

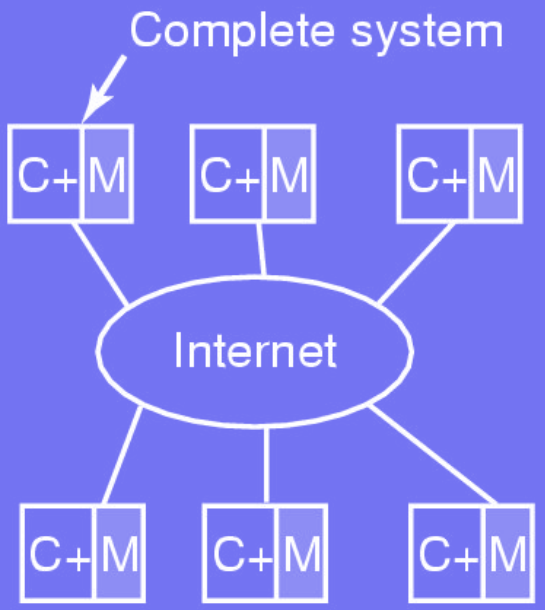
Characteristics of distributed systems:

- 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

Setup and challenges

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 handling the scheduling on slave nodes

Main challenges:

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

Interprocess communication

Over a distributed system semaphores and mutexes cannot be use

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

Example

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"

Key points

- What is the main difference between a distributed system and a multiprocessor system?
- Explain the SMP approach
- How is `TSL` working for multiprocessors?
- Cite three scheduling strategies targeting multiprocessors
- What is a middleware?

Thank you, enjoy the Winter break!