Introduction to Operating Systems

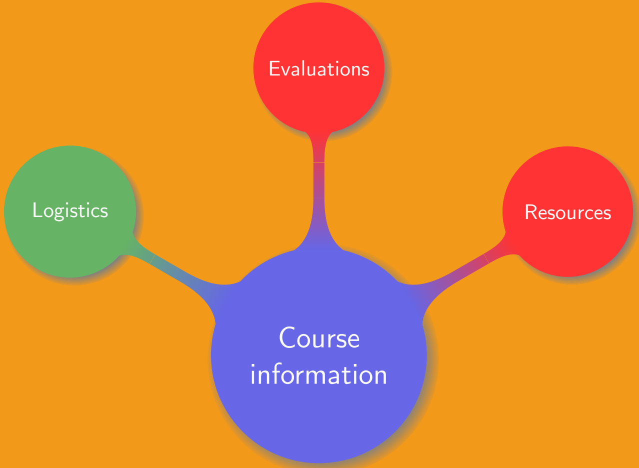
Manuel – Fall 2019

Table of contents

Chapter 0: Course information
Chapter 1: Operating systems overview
Chapter 2: Processes and threads
Chapter 3: Interprocess communication
Chapter 4: Scheduling
Chapter 5: Deadlocks
Chapter 6: Memory management
Chapter 7: Input/Output
Chapter 8: Filesystems
Chapter 9: Security
Chapter 10: Multiple processors systems

0. Course information

Chapter organisation



Basic information

Teaching team:

- Instructor: Manuel (charlem@sjtu.edu.cn)
- Teaching assistants:
  - Jiayi (jane_chen@sjtu.edu.cn)
  - Minhao (jinminhao@sjtu.edu.cn)

Basic information

Teaching team:

- Instructor: Manuel (charlem@sjtu.edu.cn)
- Teaching assistants:
  - Jiayi (jane_chen@sjtu.edu.cn)
  - Minhao (jinminhao@sjtu.edu.cn)

Important rules:

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

Basic information

Teaching team:

- Instructor: Manuel (charlem@sjtu.edu.cn)
- Teaching assistants:
  - Jiayi (jane_chen@sjtu.edu.cn)
  - Minhao (jinminhao@sjtu.edu.cn)

Important rules:

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

Never send large files by email

Course schedule

Course arrangements:

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

Appointments outside of the office hours can be taken by email

Course objectives

Main goals of this course:

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

Course objectives

Main goals of this course:

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

Be able to share in the development of an operating system

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

Course workflow

Learning strategy:

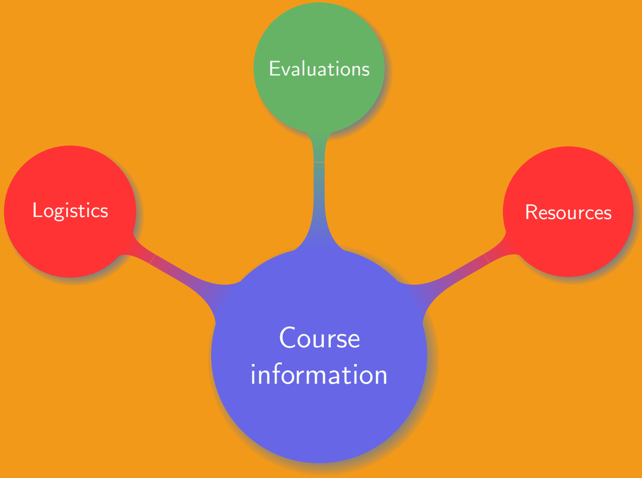
- Course side:
  1. Understand how to efficiently use the CPU
  2. Know how to handle Memory, Input/Output, and Filesystems
  3. Get a basic idea of security and distributed systems
- Personal side:
  1. Read and write code
  2. Relate known strategies to new problems
  3. Perform extra research

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: 8
- Content: improve programming skills

Projects:

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

Extra: Linux kernel challenges

Grading policy

Grade weighting:

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

Grading policy

Grade weighting:

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

Assignment submissions:

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

Grades will be curved with the median in the range $[\![B, B+]\!]$

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

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

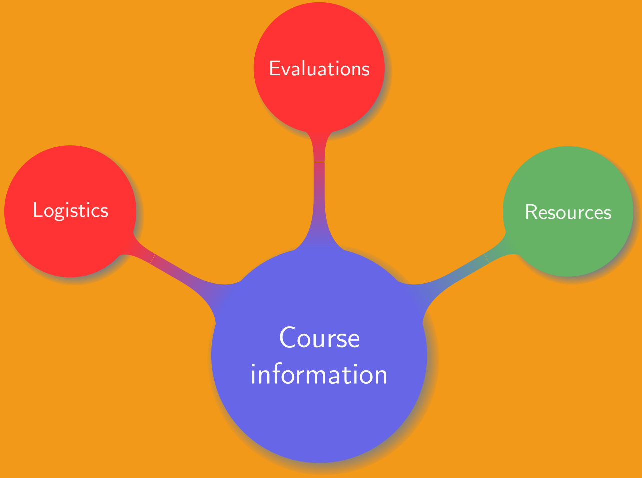
Special circumstances

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
  - Polls

References

Useful places where to find information:

- *Modern Operating Systems*, A. Tanembaum
- *Operating System Concepts*, A. Silberschatz
- OS creation: `http://wiki.osdev.org/Main_Page`
- Piazza
- Search information online, i.e. $\{\textit{websites} \setminus \{\textit{local Chinese network}\}\}$

References

Useful places where to find information:

- *Modern Operating Systems*, A. Tanembaum
- *Operating System Concepts*, A. Silberschatz
- OS creation: http://wiki.osdev.org/Main_Page
- Piazza
- Search information online, i.e. $\{\textit{websites} \setminus \{\textit{local Chinese network}\}\}$

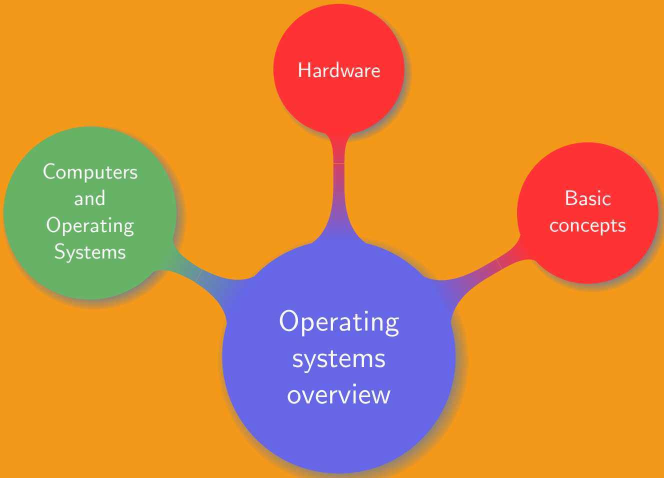
Never use Baidu in any course

Key points

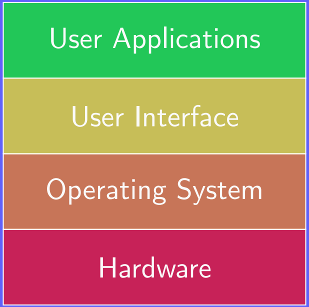
- Work regularly, do not wait the last minute/day
- Respect the Honor Code
- Go beyond what is taught
- Do not learn, understand
- Keep in touch with us
- Any advice/suggestions will be much appreciated

1. Operating systems overview

Chapter organisation



Hardware and software



A computer consists of:

- Hardware
- Software
  - Kernel mode
  - User mode

Operating System

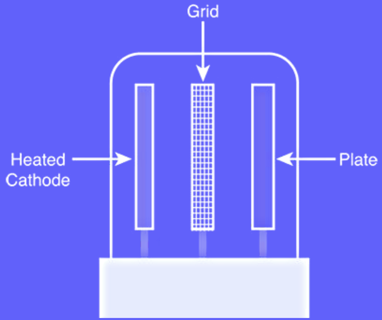
Operating System (OS)

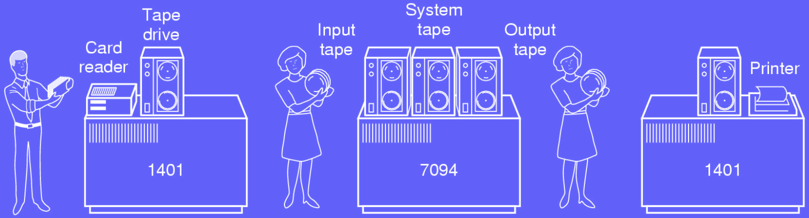
- Hardware is complicated to handle
- OS hides all the messy details
- OS manages resources for each program (time and space)
- Renders computer much easier to use

A bit of history

The first days:

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





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

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:

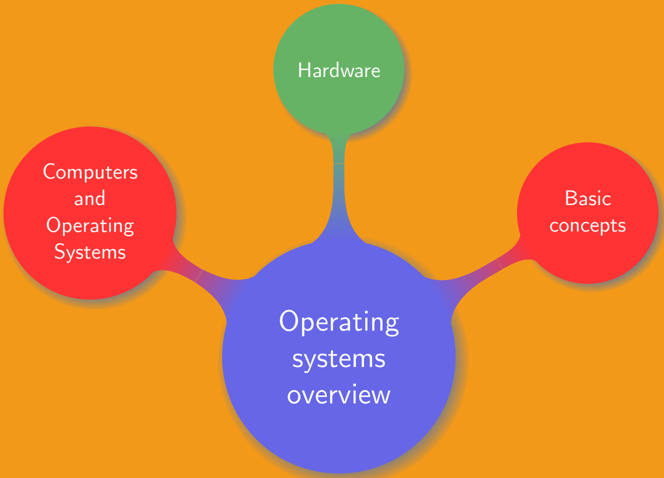
- Personal Computers (PC)
- Servers: serve users over a network (print, web, IM...) → Solaris, FreeBSD, Linux, Windows Server
- Multi processors: multiple CPU in a single system → Linux, Windows, OS X...
- Handheld computers: PDA, smartphone
- Embedded devices: TV, microwave, DVD player, mp3 player,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

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

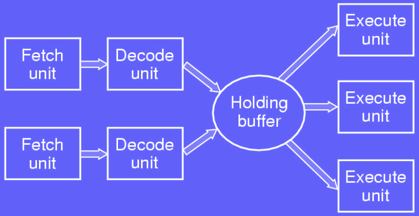
Pipeline vs. superscalar

Superscalar:

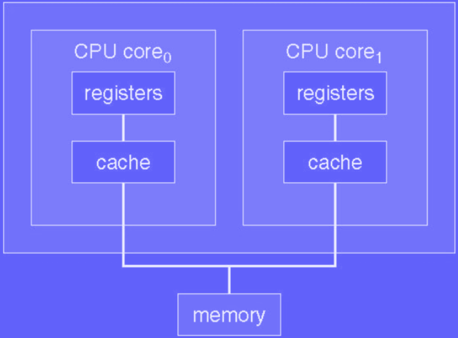
- Multiple execution units
  e.g. one for float, int, boolean
- Multiple instructions fetched and decoded at a time
- Instructions held in buffer to be executed
- Issue: no specific order to execute buffered instructions

Pipeline:

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



Multi-threading



CPUs and cores:

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

Clarification

Modern terminology[1]:

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

[1]source: ARK

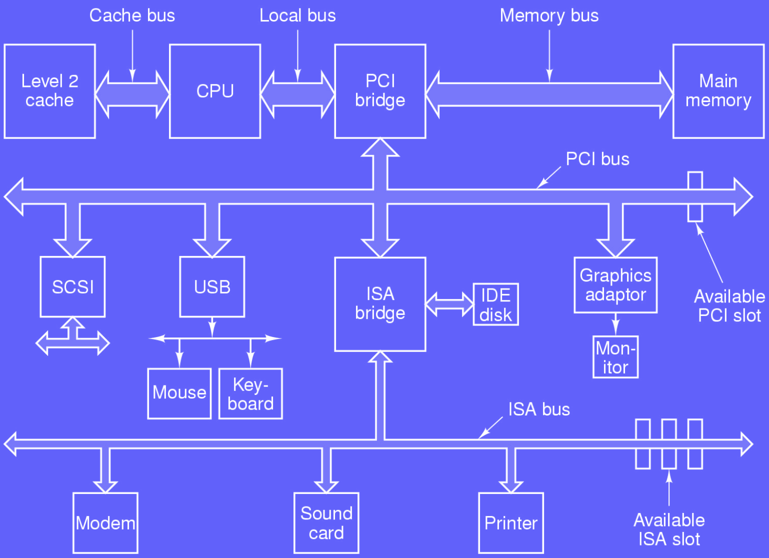
Memory

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:

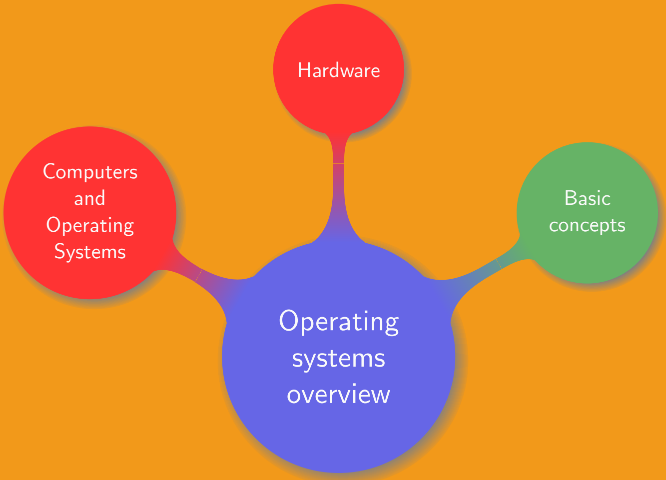
- Random Access Memory (RAM): volatile
- Read Only Memory (ROM)
- Electrically Erasable PROM (EEPROM) and flash memory: slower than RAM, non volatile.
- CMOS: save time and date , BIOS parameters
- HDD: divided into cylinder, track and sector

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
Mon-itor
Available PCI slot
Mouse
Key-board
ISA bus
Modem
Sound card
Printer
Available ISA slot

Chapter organisation



Main categories

Five major components of an OS:

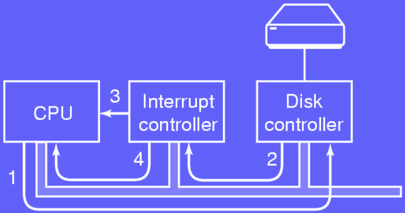
- Input/Output
- Protection/Security
- Processes
- File system
- System calls

First communication strategy

Using the device controller:

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

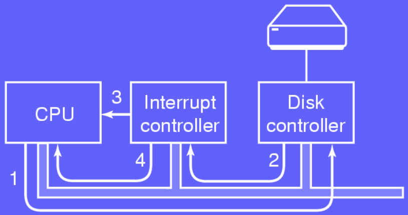
Interrupt



Starting an I/O device and getting an interrupt:

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

Interrupt



Starting an I/O device and getting an interrupt:

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

Processing an interrupt:

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

Interrupts

Remarks of interrupts:

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

Second communication strategy

Simplest method:

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

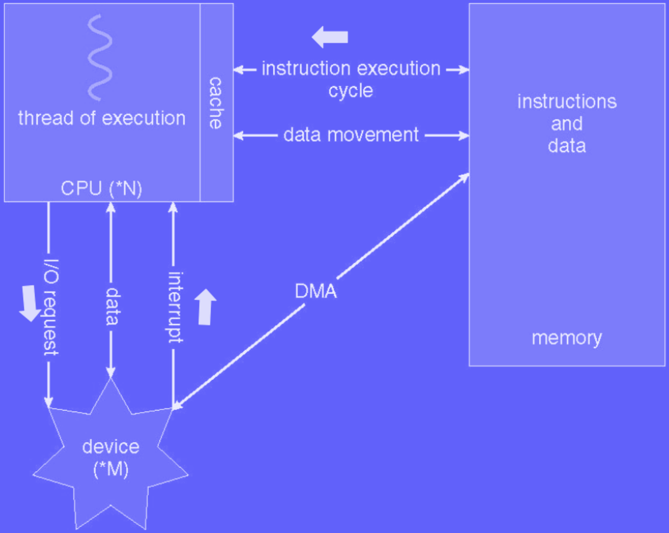
Disadvantage: CPU *busy waiting* until device is finished

Third communication strategy

Direct Memory Access (DMA):

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

Summary


thread of execution
cache
CPU (*N)
instruction execution cycle
data movement
instructions
and
data
memory
I/O request
data
interrupt
DMA
device
(*M)

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 smallest legal physical memory address and size of range, respectively
- Memory outside the address space is protected

Input/Output:

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

Processes

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

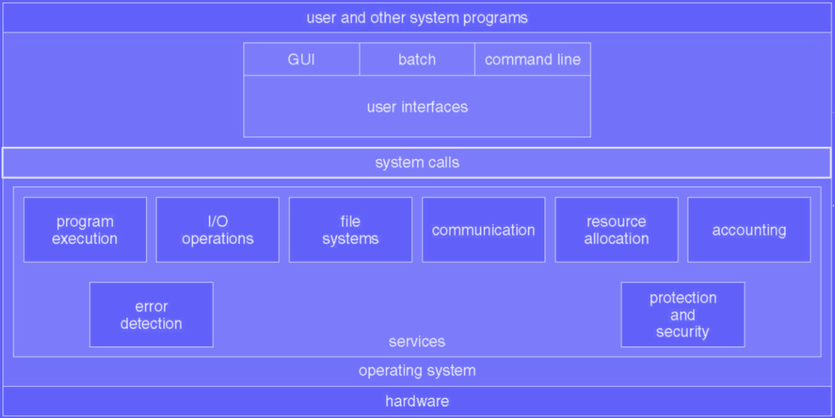
- Address space belonging to the process and containing:
  - Executable program
  - Program's data
  - Program's stack
- Set of resources:
  - Registers
  - List of open files
  - Alarms
  - List of related processes
  - Any other information required by the program

File system

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

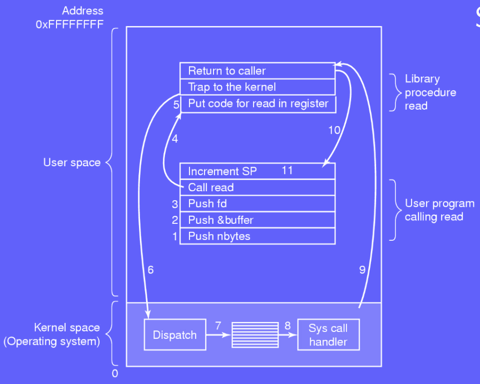
- Data stored in files
- Files are grouped inside directories
- Top directory is called root directory
- Any file can be specified using its path name
- Each process has a working directory
- Before reading/writing in a file permissions are checked
- If access is granted a file descriptor is returned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

System calls



Simple example

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



Steps for a simple `read`:

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

Common system calls

Partial list of common Unix system calls:

- Processes:

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

- Files:

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

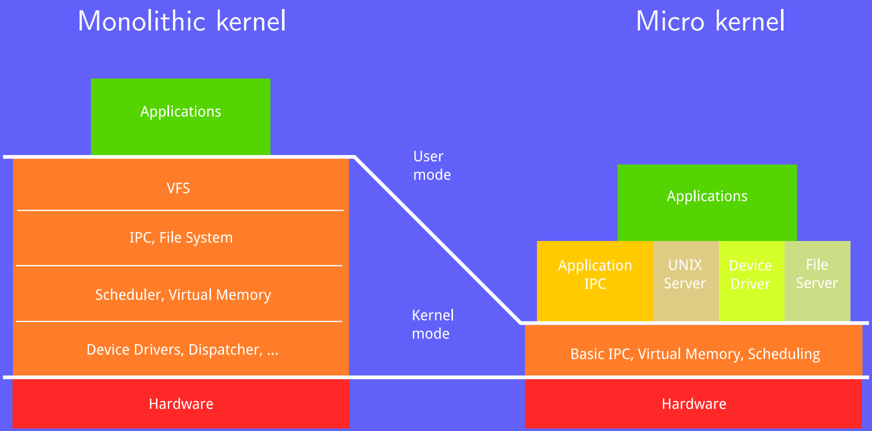
- Directory and file system:

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

- Misc:

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

OS structure

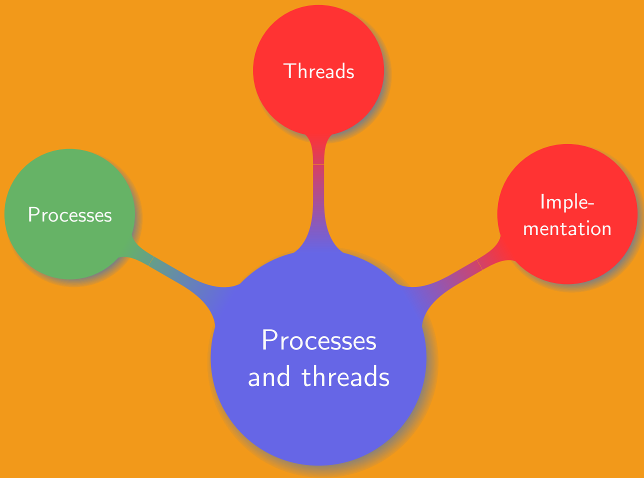


Key points

- 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

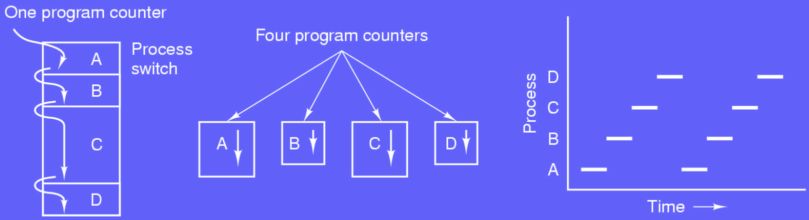


Basics

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

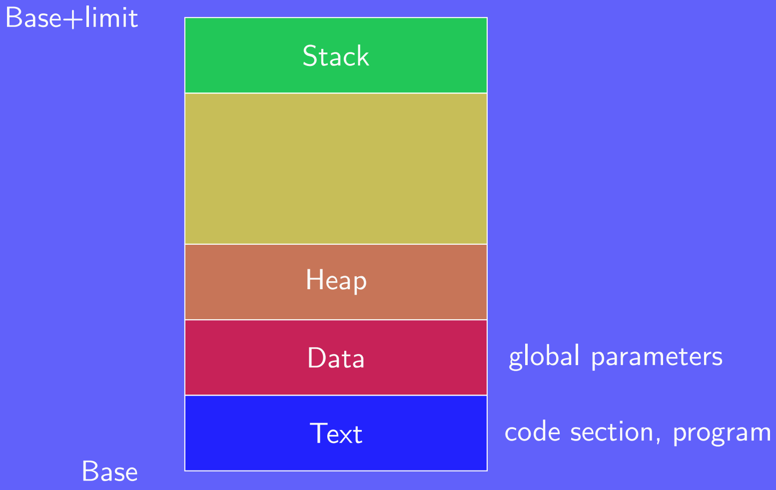
Program vs. process

Differences between programs and processes:

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

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

Process in memory



Process creation

Four main events causing process to be created:

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

Example

Unix like systems:

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

Windows system:

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

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

Process termination

Any created processes ends at some stage:

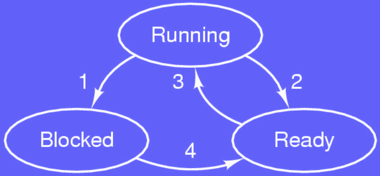
- Normal exit (voluntary)
  work is done, execute a system call to tell OS it is finished
  `exit`, `ExitProcess`
- Error exit (voluntary)
  e.g. requested file does not exist
- Fatal error (involuntary)
  e.g. referencing non existent memory, dividing by 0
- Killed by another process (involuntary)
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

Process hierarchies

Two main approaches:

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

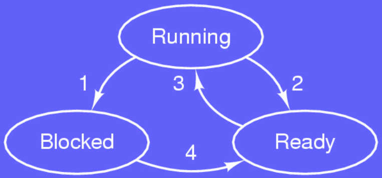
Process states



Possible states:

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

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:

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

Modeling processes

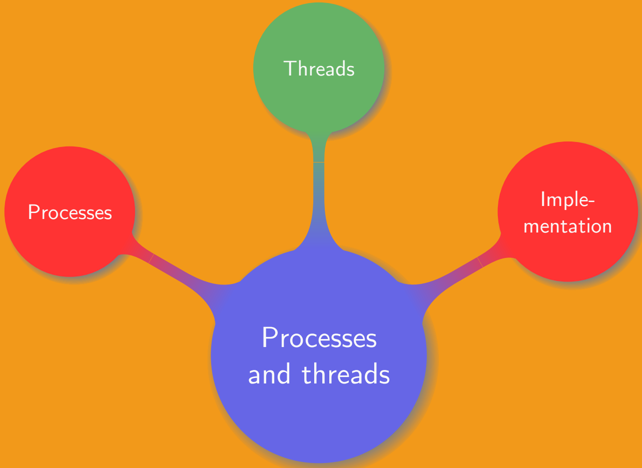
Process management	Memory management	File management
registers	pointer to text segment info	root directory
program counter	pointer to data segment info	working directory
program status word	pointer to stack segment info	file descriptors
stack pointer		user ID
process state		group ID
priority		
scheduling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

Interrupts and processes

Lowest OS level:

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

Chapter organisation



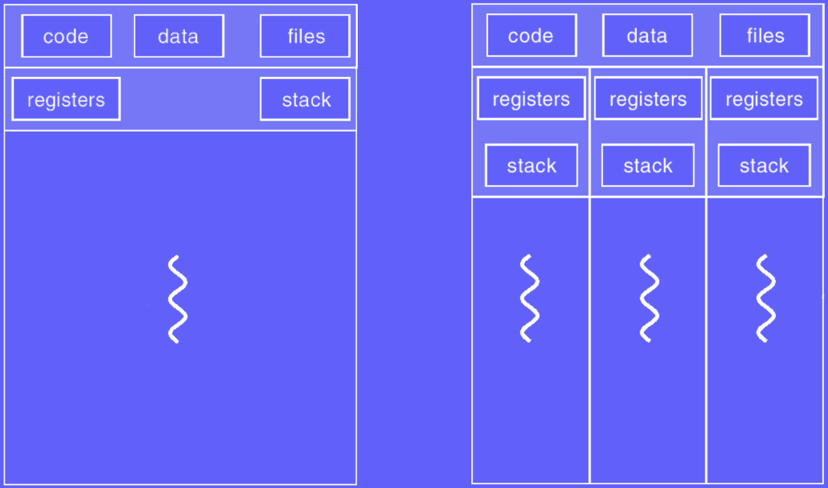
A thread is the basic unit of CPU utilisation consisting of:

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

All the threads within a process share:

- Code section
- Data section
- OS resources

Single vs. multi-threaded

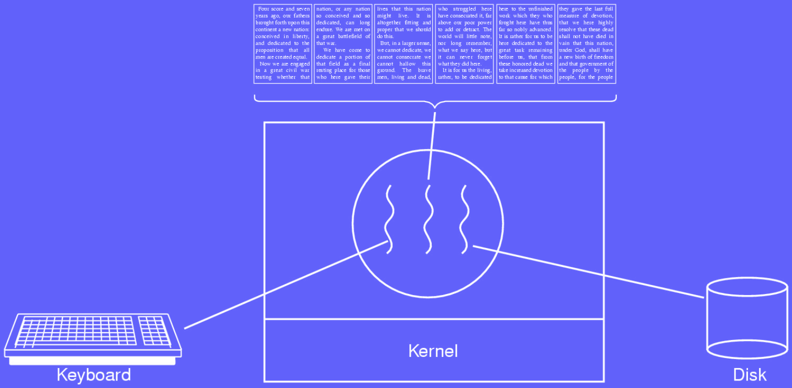


Notes and remarks

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 process
- 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


Four score and seven years ago, our fathers brought forth upon this continent a new nation: conceived in liberty, and dedicated to the proposition that all men are created equal.
Now we are engaged in a great civil war testing whether that
nation, or any nation so conceived and so dedicated, can long endure. We are met on a great battlefield of that war.
We have come to dedicate a portion of that field as a final resting place for those who here gave their
lives that this nation might live. It is altogether fitting and proper that we should do this.
But, in a larger sense, we cannot dedicate, we cannot consecrate we cannot hallow this ground. The brave men, living and dead,
who struggled here have consecrated it, far above our poor power to add or detract. The world will little note, nor long remember, what we say here, but it can never forget what they did here.
It is for us the living, rather, to be dedicated
here to the unfinished work which they who fought here have thus far so nobly advanced. It is rather for us to be here dedicated to the great task remaining before us, that from these honored dead we take increased devotion to that cause for which
they gave the last full measure of devotion, that we here highly resolve that these dead shall not have died in vain that this nation, under God, shall have a new birth of freedom and that government of the people by the people, for the people
Kernel
Keyboard
Disk

Multi-threading problems

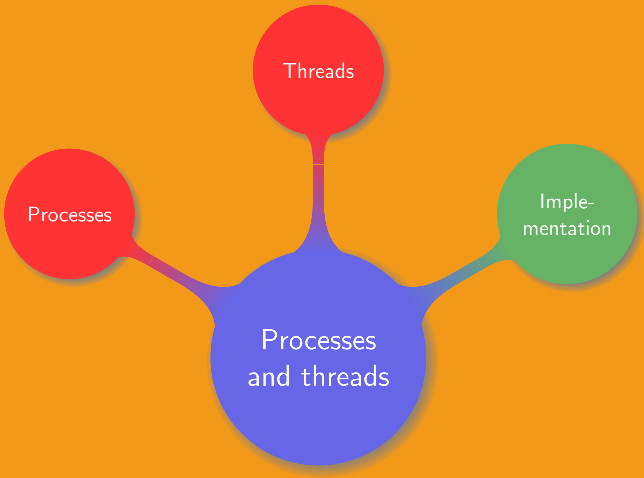
Thread share many data structure:

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

Should a child inherit all the threads form its parents?

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

Chapter organisation



POSIX threads

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

- Create a thread:
  `int pthread_create(pthread_t *thread, const pthread_attr_t *attr,void *(*start_routine) (void *), void *arg);`
- Terminate a thread:
  `void pthread_exit(void *retval);`
- Wait for a specific thread to end:
  `int pthread_join(pthread_t thread, void **retval);`
- Release CPU to let another thread run:
  `int pthread_yield(void);`
- Create and initialise a thread attribute structure:
  `int pthread_attr_init(pthread_attr_t *attr);`
- Delete a thread attribute object:
  `int pthread_attr_destroy(pthread_attr_t *attr);`

Exercise

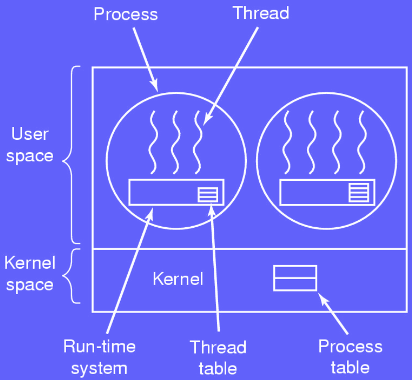
Write a program that creates 10 threads, and prints their ID.

Solution

threads.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



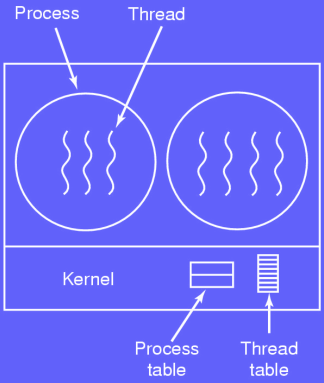
User-space threads:

- 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



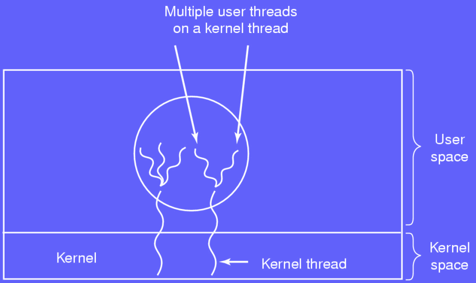
Kernel space thread:

- 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:

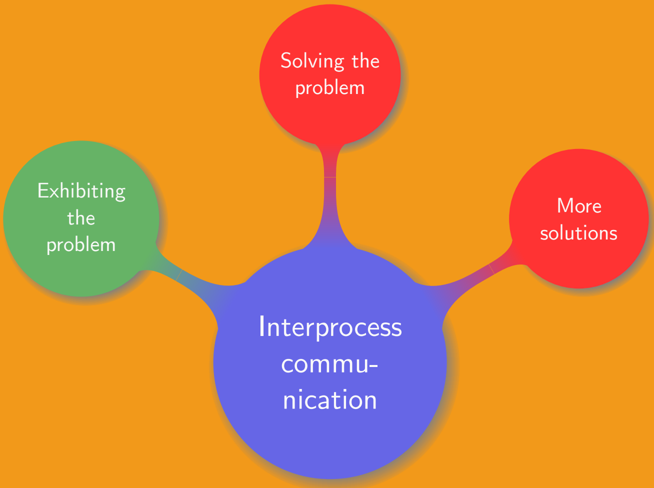
- Hybrid looked attractive
- Most systems are coming back to 1:1
- Different approaches exist on how to use threads
  e.g. thread bocks on "receive system call" vs. pop up threads
- Switching implementation from single thread to multiple thread is not easy task
- Requires redesigning the whole system
- 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



Easy setup

Independent threads:

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

Executing threads

Where difficulties start:

- Single-tasking: run a thread to completion and start next one
- Multi-tasking:
  - One core shared among several threads
  - Several cores run several threads in parallel
- A thread runs on one core at a time
- A thread can run on different cores at different times
- For threads no difference between one or more cores

Cooperating threads

Setup:

- Threads sharing a state
- Behavior depends on the execution sequence
- Behavior may seem random/irreproducible

Cooperating threads

Setup:

- Threads sharing a state
- Behavior depends on the execution sequence
- Behavior may seem random/irreproducible

Major problems:

1. How can threads/processes share information?
2. How to prevent them for getting in each other's way?
3. How to handle sequencing?

Atomic operation

Operation that either happens in its entirety or not at all:

- Atomic operation cannot be created
- Atomic operations are hardware level
- Central to the question of parallel programming

Atomic operation

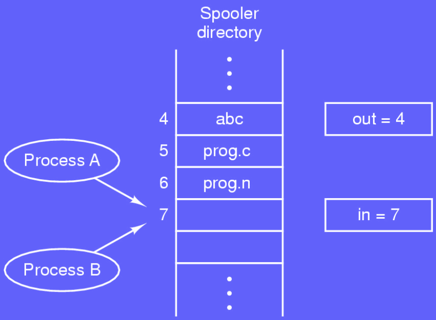
Operation that either happens in its entirety or not at all:

- Atomic operation cannot be created
- Atomic operations are hardware level
- Central to the question of parallel programming

Example.
Most basic atomic operation, `A=B`:

- Read a clean value for B
- Set a clean value for A

Race conditions



Practical example:

- Process A wants to queue a file, reads next_free_slot=7
- Interrupt occurs, Process B reads next_free_slot=7
- Process B queues its file in slot 7, and update next_free_slot=8
- Process A resumes using next_free_slot=7
- Game over

Too much milk

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

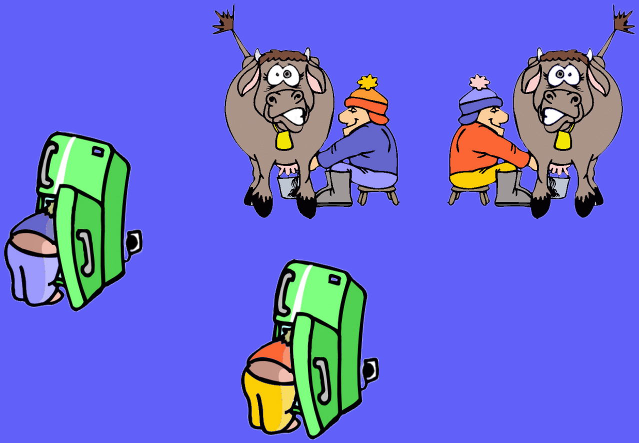
Too much milk

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

Too much milk

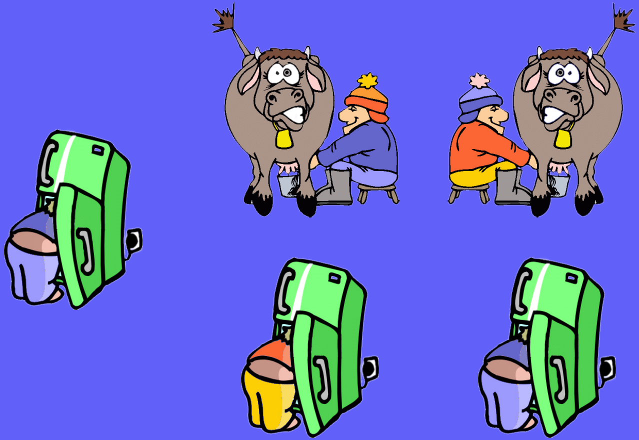
9:00 am	9:15 am	9:30 am	9:45 am

Too much milk



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

Too much milk

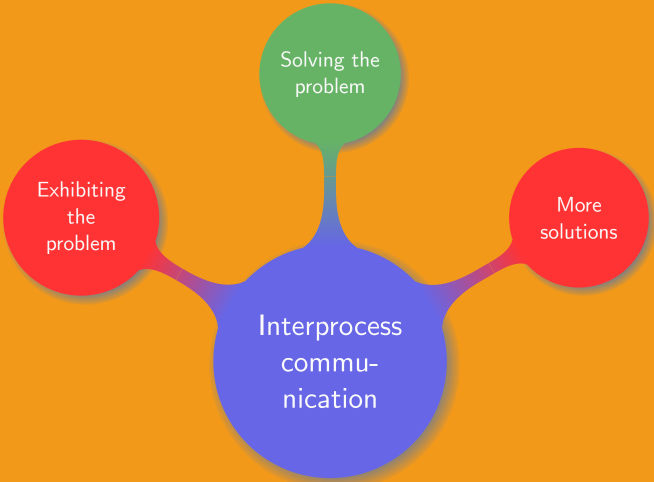


9:00 am	9:15 am	9:30 am	9:45 am

Too much milk

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

Chapter organisation

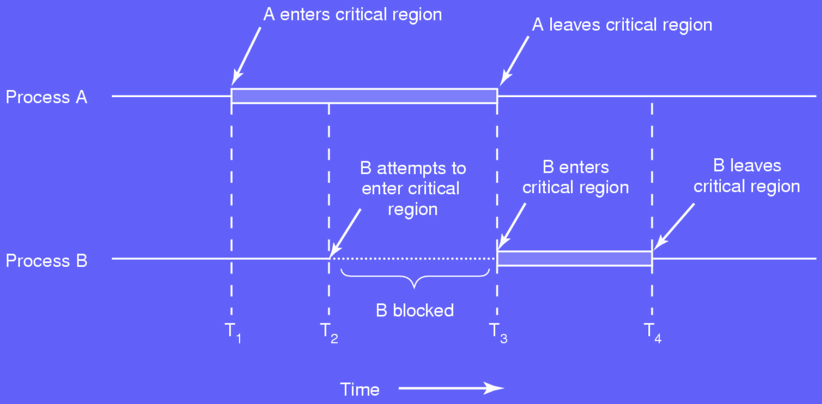


Critical region

Part of the program where shared memory is accessed:

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

Mutual exclusion



Too much milk

Frank

```
if(no milk && no note) {
  leave note;
  milk the cow;
  remove note;
}
```

John

```
if(no milk && no note) {
  leave note;
  milk the cow;
  remove note;
}
```

Too much milk

Frank

```
if(no milk && no note) {
  leave note;
  milk the cow;
  remove note;
}
```

John

```
if(no milk && no note) {
  leave note;
  milk the cow;
  remove note;
}
```

Result: too much milk

Too much milk

Frank

```
leave note Frank;
if(no note John) {
  if(no milk){
    milk the cow;
  }
}
remove note Frank;
```

John

```
leave note John;
if(no note Frank) {
  if(no milk) {
    milk the cow;
  }
}
remove note John;
```

Too much milk

Frank

```
leave note Frank;
if(no note John) {
  if(no milk){
    milk the cow;
  }
}
remove note Frank;
```

John

```
leave note John;
if(no note Frank) {
  if(no milk) {
    milk the cow;
  }
}
remove note John;
```

Result: no milk

Too much milk

Frank

```
leave note Frank;
while(note John) {
  nothing;
}
if(no milk) {
  milk the cow;
}
remove note Frank;
```

John

```
leave note John;
if(no note Frank) {
  if(no milk) {
    milk the cow;
  }
}
remove note John;
```

Too much milk

Frank

```
leave note Frank;
while(note John) {
  nothing;
}
if(no milk) {
  milk the cow;
}
remove note Frank;
```

John

```
leave note John;
if(no note Frank) {
  if(no milk) {
    milk the cow;
  }
}
remove note John;
```

Result: just enough milk

Too much milk

Comments on the solution:

- Solution is correct
- Complicated
- Asymmetrical
- Busy wait, CPU time wasted

Peterson's idea

Simple strategy:

- Assume two processes
- Two functions:
  - `enter_region`: show process is interested and wait for its turn
  - `leave_region`: indicates departure from critical region
- Solution is correct
- Drawback: busy wait

Peterson's idea

Simple strategy:

- Assume two processes
- Two functions:
  - `enter_region`: show process is interested and wait for its turn
  - `leave_region`: indicates departure from critical region
- Solution is correct
- Drawback: busy wait

Exercise.
Write the pseudo C code for Peterson's idea using two processes represented by the integers 0 and 1

Peterson's idea

```c
#define TRUE 1
#define FALSE 0
int turn;
int interested[2];
void enter_region(int p) {
  int other;
  other=1-p;
  interested[p]=TRUE;
  turn=p;
  while(turn==p && interested[other]==TRUE)
}
void leave_region(int p) {
  interested[p]=FALSE;
}
```

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

Software vs. hardware?

Disabling interrupts, good or bad:

Software vs. hardware?

Disabling interrupts, good or bad:

- Case 1: block interrupts for the whole computer
  - Interrupts could be disabled for a while (too long?)
  - This gives a lot of power to the programmer
- Case 2: block interrupts for only one CPU
  - No effect on other processors
  - Another CPU can access the variable between read and write

The TSL instruction

A simple atomic operation:

- Test and Set Lock (TSL)
- Hardware level, requires assembly
- Task: copy *lock* to register and set *lock* to 1
- Atomic operation

The TSL instruction

A simple atomic operation:

- Test and Set Lock (TSL)
- Hardware level, requires assembly
- Task: copy *lock* to register and set *lock* to 1
- Atomic operation

```
enter_region:
  TSL REGISTER,LOCK
  CMP REGISTER,#0
  JNE enter_region
  RET

leave_region:
  MOVE LOCK,#0
  RET
```

The TSL instruction

A simple atomic operation:

- Test and Set Lock (TSL)
- Hardware level, requires assembly
- Task: copy *lock* to register and set *lock* to 1
- Atomic operation

```
enter_region:
  TSL REGISTER,LOCK
  CMP REGISTER,#0
  JNE enter_region
  RET

leave_region:
  MOVE LOCK,#0
  RET
```

Note: TSL displays the same side-effect as Peterson's solution when dealing with processes having different priorities

Blocking strategy

```c
#define N 100
int count=0;
void producer() {
  int item;
  while(1) {
    item=produce_item();
    if(count==N) sleep();
    insert_item(item); count++;
    if(count==1) wakeup(consumer);
  }
}
void consumer() {
  int item;
  while(1) {
    if(count==0) sleep();
    item=remove_item(); count--;
    if(count==N-1) wakeup(producer);
    consume_item(item);
  }
}
```

Race condition

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:

- 1965, Edseger Dijkstra
- Simple hardware based solution
- Basis of all contemporary OS synchronization mechanisms

Semaphore

Basics:

- 1965, Edseger Dijkstra
- Simple hardware based solution
- Basis of all contemporary OS synchronization mechanisms

A semaphore *s* in a non-negative variable that can only be changed or tested by two atomic actions:

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

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

Semaphore and processes

The down operation

Check if the value is > 0

- True: decrement the value and continues
- False: sleep, do not complete the down

Only one single atomic action

The up operation

- Increment the value of the semaphore
- If one or more processes were asleep, randomly choose one to wakeup (complete its down)

Only one single atomic action

Semaphore implementation

Semaphores MUST be implemented in an indivisible way:

- Up and down are implemented using system calls
- OS disables all interrupts while testing, updating the semaphore and putting process to sleep
- When dealing with more than one CPU, semaphores are protected using the TSL instruction

Note: a semaphore operation only takes a few microseconds

Semaphore and interrupts

Hiding interrupts using semaphores:

- Each I/O device gets a semaphore initialised to 0
- Managing process applies a `down` when starting an I/O device
- Process is blocked
- Interrupt handler applies an `up` when receiving an interrupt
- Process is ready to run again

Mutex

MUTual EXclusion:

- Simplified semaphore
- Takes values 0 (unlocked) or 1 (locked)

On a mutex-lock request:

- If mutex is currently unlocked, lock it; thread can enter in critical region
- If mutex is currently locked, block the calling thread until thread in critical regions is done and calls mutex-unlock
- Randomly chose which thread acquires the lock if more than one are waiting

Mutex implementation

Mutexes can be implemented in user-space using TSL

```
mutex-lock:
  TSL REGISTER,MUTEX
  CMP REGISTER,#0
  JNE ok
  CALL thread_yield
  JMP mutex-lock
ok: RET

mutex-unlock:
  MOVE MUTEX,#0
  RET
```

Questions.

- What differences were introduced compared to `enter_region` (3.102)?
- In user-space what happens if a thread tries to acquire lock through busy-waiting?
- Why is `thread_yield` used?

Example

consumer_producer.c

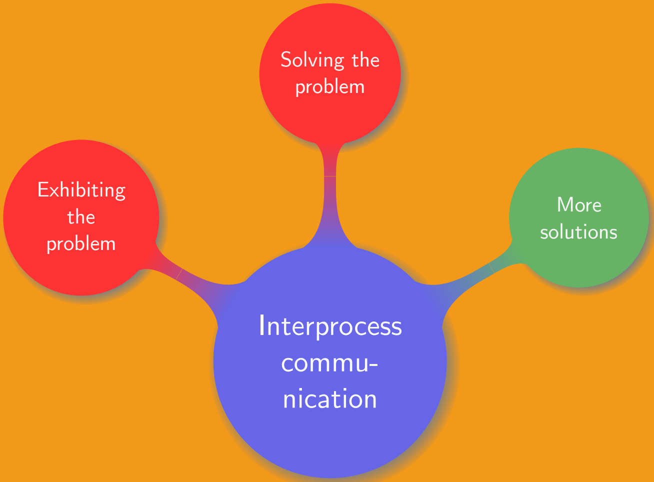
```c
#include <stdio.h>
#include <pthread.h>
#define MAX 1000
pthread_mutex_t m; pthread_cond_t cc, cp; int buf=0;
void *prod() {
  for(int i=1;i<MAX;i++) {
    pthread_mutex_lock(&m); while(buf!=0) pthread_cond_wait(&cp,&m);
    buf=1; pthread_cond_signal(&cc); pthread_mutex_unlock(&m);
  }
  pthread_exit(0);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

Example

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



Dangers...

```c
mutex mut = 0; semaphore empty = 100; semaphore full = 0;
void producer() {
  while(TRUE) {
    item = produce_item();
    mutex-lock(&mut);
    down(&empty);
    insert_item(item);
    mutex-unlock(&mut);
    up(&full);
  }
}
void consumer()  {
  while(TRUE) {
    down(&full);
    mutex-lock(&mut);
    item = remove_item();
    mutex-unlock(&mut);
    up(&empty); consume_item(item);
  }
}
```

Questions

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

As semaphores and mutexes are sometimes risky *monitors* were introduced:

- Monitors are higher level, cleaner and less risky solution
- A monitor is composed of
  - Procedures/functions
  - Structures
- Only one process can be active within a monitor at a time
- Monitors are a programming concept, compiler must know them
- Mutual exclusion handled by the compiler not the programmer

Monitors

Basics on monitors:

- Monitors are easy to use for mutual exclusion
- Offer a condition to block “properly” when the buffer is full
- A signal on the condition variable can wake up a blocked process
- Only one process can be active in the monitor
- As soon as the signal on the condition is sent, exit the monitor
- Other process can resume

Example

```
monitor ProducerConsumer {
  condition full, empty;
  int count;
  void insert(item) {
    if (count == N) wait(full);
    insert_item(item);
    count++;
    if (count==1) signal(empty);
  }
  void remove() {
    if (count==0) wait(empty);
    removed = remove_item;
    count--;
    if (count==N-1) signal(full);
  }
  count:= 0;
}
```

```
void ProducerConsumer::producer() {
  while (TRUE) {
    item = produce_item();
    ProducerConsumer.insert(item);
  }
}
void ProducerConsumer::consumer() {
  while (TRUE) {
    item=ProducerConsumer.remove();
    consume_item(item)
  }
}
```

Message passing

Limitation of semaphores and monitors: processes need to share some part of the memory. Distributed systems over a network consist of multiple CPU each with its own private memory

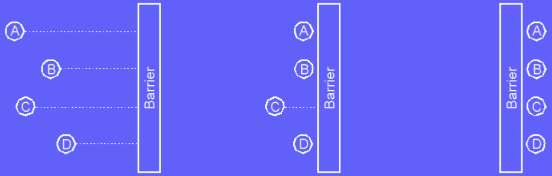
Message passing:

- `send(destination,&message)`
- `receive(source,&message)`, can either block or exit if nothing is received

Potential issues:

- Message lost (sending/acknowledging reception)
- No possible confusion on process names
- Security (authentication, traffic)
- Performance

Barriers



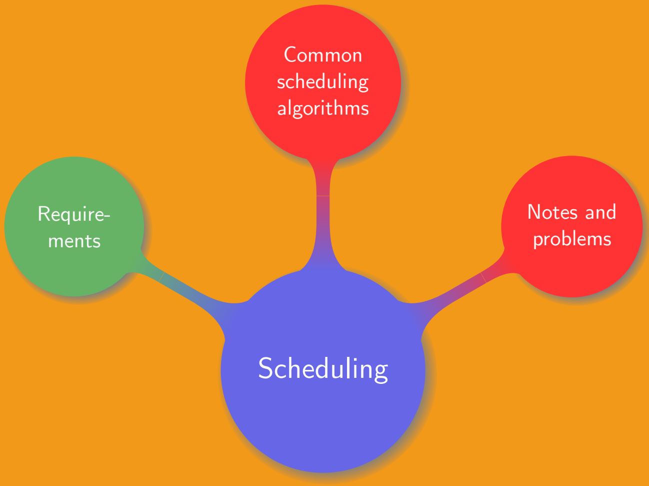
Useful for problems where several processes must complete before the next phase can start

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

Scheduler's job

Switching process is expensive:

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

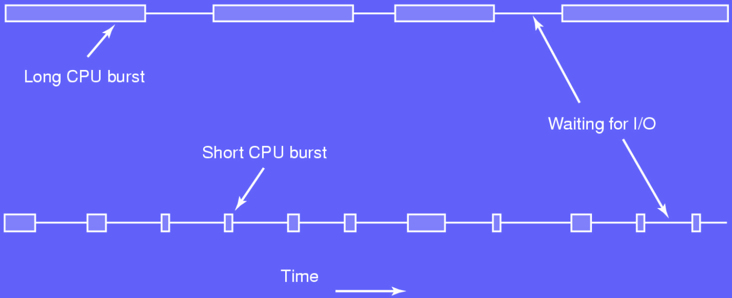
Scheduler's job

Switching process is expensive:

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

Remark.
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

Compute bound vs. I/O bound:

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

Process behavior

Compute bound vs. I/O bound:

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

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

When to schedule?

Basic scheduling issues:

- Creation of a new process: both the parent and the child are in ready state
- A process exits: which process to run next?
  An idle process is run if no process is ready
- A process blocks (semaphore, I/O...), what process to run?
  Note: a blocked process could be waiting for another process, how to run this specific process?
- I/O interrupt from a device that has completed its task: run the newly ready process, or another one?

Preemptive vs. non-preemptive

Two main strategies for scheduling algorithms:

1. Preemptive: a process is run for at most n ms; if it is not done at the end of the period then it is suspended and another process is selected and run: require a clock interrupt

2. Non-preemptive: a process runs until it blocks or voluntarily releases the CPU; it is resumed after an interrupt unless another process with higher priority is in the queue

What is the goal?

All systems:

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

What is the goal?

All systems:

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

Batch systems:

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

What is the goal?

Interactive systems:

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

What is the goal?

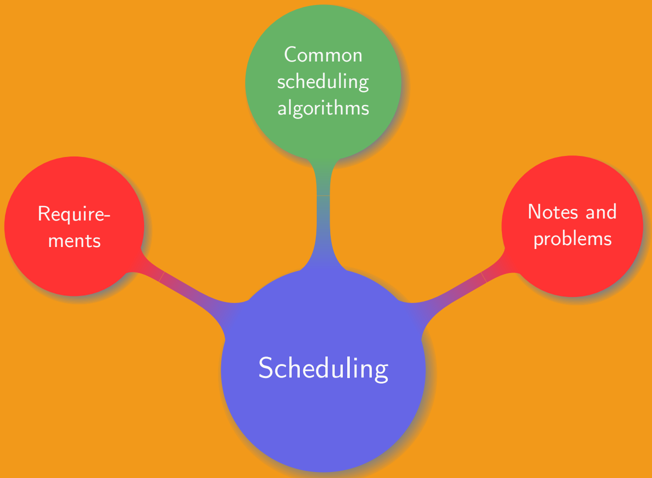
Interactive systems:

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

Real-time systems:

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

Chapter organisation



First-come, first-served

Basics: simplest algorithm, non-preemptive

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

First-come, first-served

Basics: simplest algorithm, non-preemptive

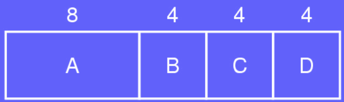
- CPU is assigned in the order it is requested
- Processes are not interrupted, they can run a long as they want
- New jobs are put at the end of the queue
- When a process blocks the next in line is run
- When a blocked process becomes ready it is put at the end of the queue

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

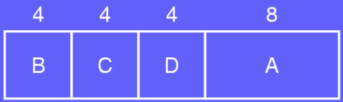
Drawback: what happens when there is one compute-bound process and many I/O-bound ones?

Shortest job first

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



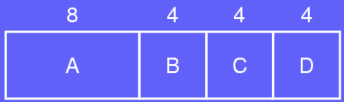
- Run time: A: 8 min, B: 4 min, C: 4 min, D: 4 min
- Turnaround time: $\frac{8+12+16+20}{4} = 14$ min

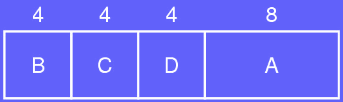


- Run time: B: 4 min, C: 4 min, D: 4 min, A: 8 min
- Turnaround time: $\frac{4+8+12+20}{4} = 11$ min

Shortest job first

Basics: Non-preemptive, run times are 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

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

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

Round-Robin scheduling

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

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

Round-Robin scheduling

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

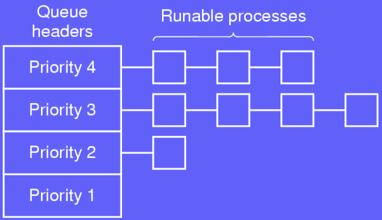
- Each process is assigned a time interval called *quantum*
- A process runs until (i) it blocks, (ii) it is finished or (iii) its quantum is over
- Switching process is then done

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

Drawback: how long should a quantum be?

Priority scheduling

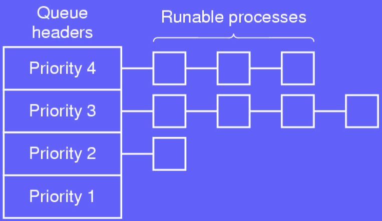
Basics: preemptive, priority depending on who or what runs



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

Priority scheduling

Basics: preemptive, priority depending on who or what runs



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

Characteristics: good for interactive systems, flexible, adjustable

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

Lottery scheduling

Basics: preemptive, extends priority scheduling

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

Lottery scheduling

Basics: preemptive, extends priority scheduling

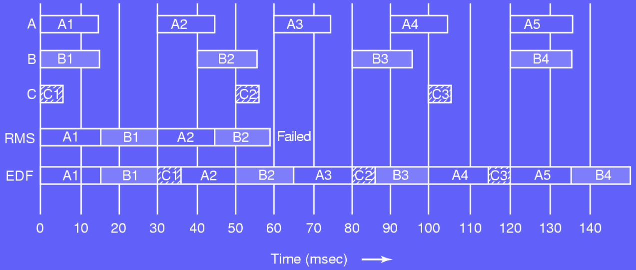
- Processes get "lottery tickets"
- When a scheduling decision is made a random ticket is chosen
- Price for the winner is to access resources
- High priority processes get more tickets (higher probability of winning)

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

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

Earliest deadline first

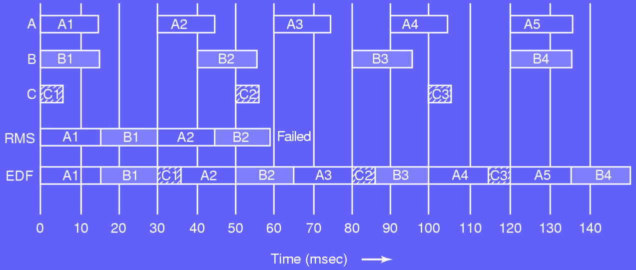
Basics: preemptive, priority based



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

Earliest deadline first

Basics: preemptive, priority based

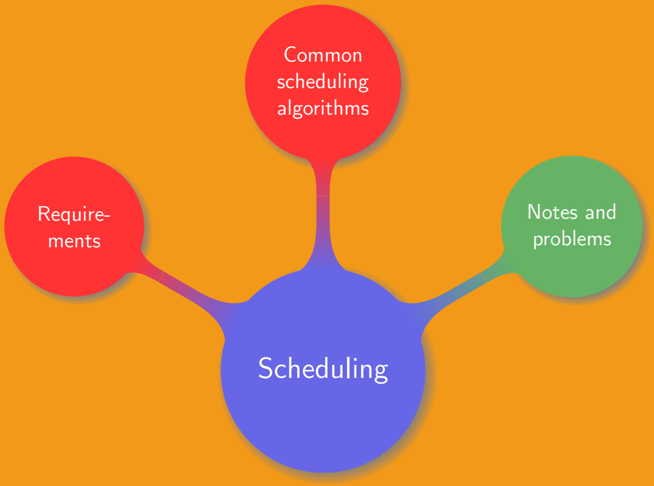


- Process needs to announce (i) its presence and (ii) its deadline
- Scheduler orders processes with respect to their deadline
- First process in the list (earliest deadline) is run

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

Drawback: complex to implement

Chapter organisation



Policy vs. mechanism

Limitations of the previous algorithms:

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

Policy vs. mechanism

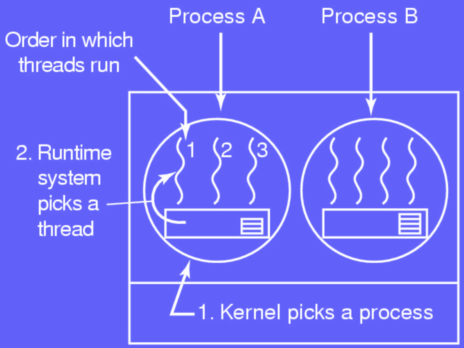
Limitations of the previous algorithms:

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

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

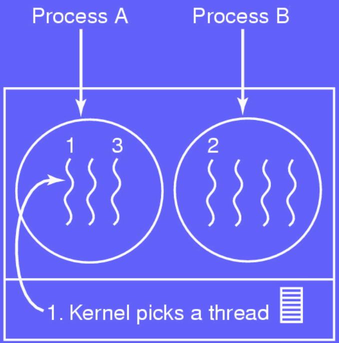
- Scheduling algorithm has parameters
- Parameters can be set by processes
- A parent can decide which of its children should have higher priority

Threads scheduling – User-level



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

Threads scheduling – Kernel-level



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

The dining philosophers problem

Synchronisation problem:

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

The dining philosophers problem

First obvious solution:

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

The dining philosophers problem

First obvious solution:

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

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

The dining philosophers problem

First obvious solution:

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

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

Second solution:

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

The dining philosophers problem

First obvious solution:

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

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

Second solution:

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

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

The dining philosophers problem

What about using mutex?

The dining philosophers problem

What about using mutex?

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

The dining philosophers problem

What about using mutex?

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

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

The dining philosophers problem

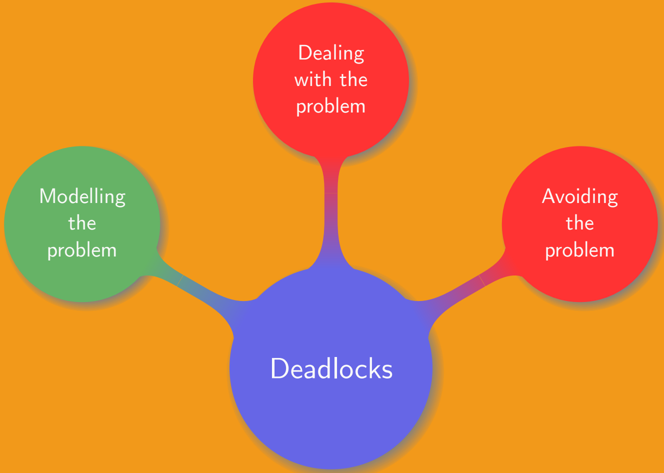
```c
#define N 5
#define LEFT (i+N-1)%N
#define RIGHT (i+1)%N
enum { THINKING, HUNGRY, EATING };
int state[N]; mutex mut = 0 ; semaphore s[N];
void philosopher(int i) {while(TRUE) {think();take_cs(i);eat();put_cs(i);}}
void take_cs(int i) {
  mutex-lock(&mut);
  state[i] = HUNGRY; test(i);
  mutex-unlock(&mut); down(&s[i]);
}
void put_cs(int i) {
  mutex-lock(&mut);
  state[i] = THINKING; test(LEFT); test(RIGHT);
  mutex-unlock(&mut);
}
void test(int i) {
  if(state[i]==HUNGRY && state[LEFT]!=EATING && state[RIGHT]!=EATING;) {
    state[i]=EATING; up(&s[i]); }
}
```

Key points

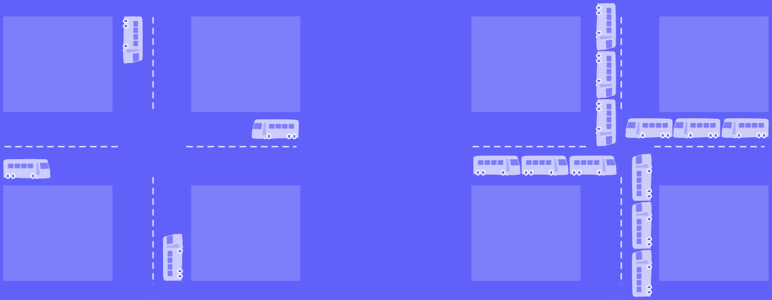
- Why is scheduling the lowest part of the OS?
- What are the two main types of algorithm?
- What are the two most common scheduling algorithms?
- Give an example of theoretical problem related to scheduling

5. Deadlocks

Chapter organisation



What is a deadlock?



A typical deadlock

Simple example:

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

A typical deadlock

Simple example:

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

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

Types of resources

Preemptable

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

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

Non-preemptable

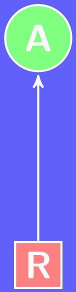
(Resources that cannot be safely taken away from a process)

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

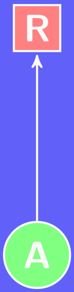
Representing deadlocks

Resource allocation represented using graphs:

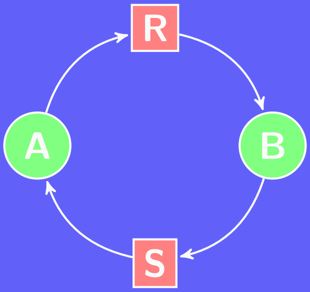
Resource R held by A



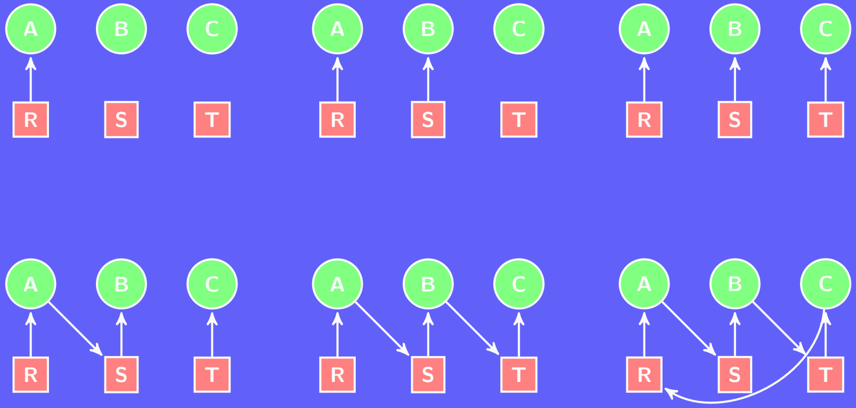
Resource R requested by A



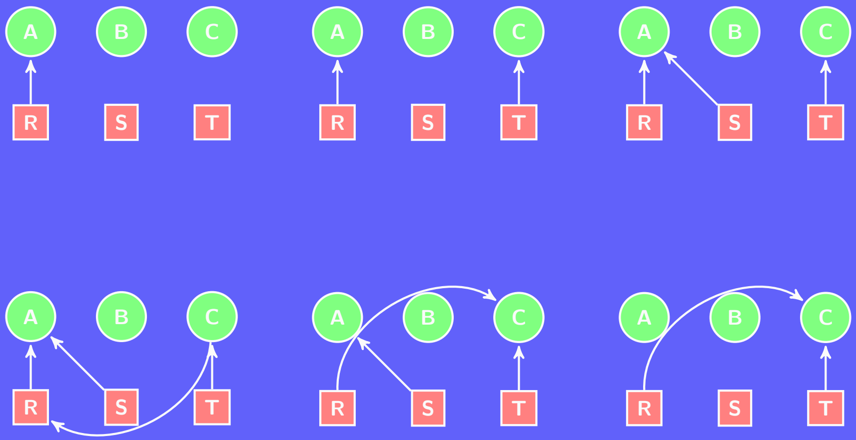
Deadlock



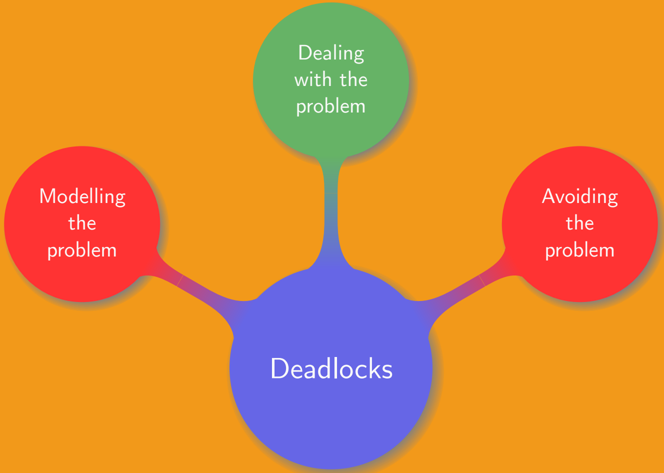
Example – Deadlock



Example – No deadlock



Chapter organisation



ALERT, ALERT

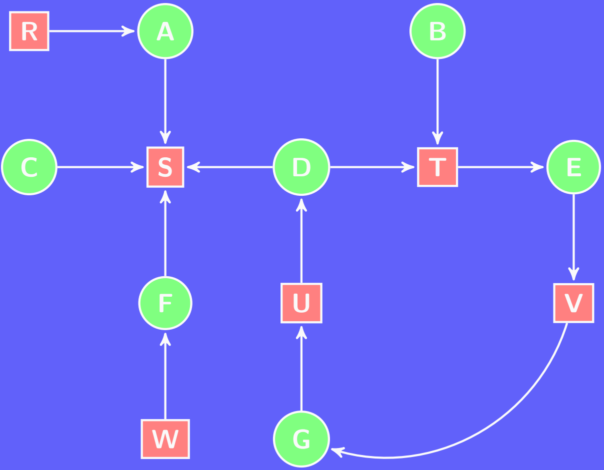
HIDE!!!!

Deadlock detection – Single resource

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

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

Deadlock detection – Single resource



Deadlock detection – Multiple resources

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

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

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

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

Deadlock detection – Multiple resources

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

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

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

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

Deadlock detection – Multiple resources

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

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

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

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

Deadlock detection – Multiple resources

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

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

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

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

Deadlock detection – Multiple resources

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

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

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

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

Deadlock detection – Multiple resources

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

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

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

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

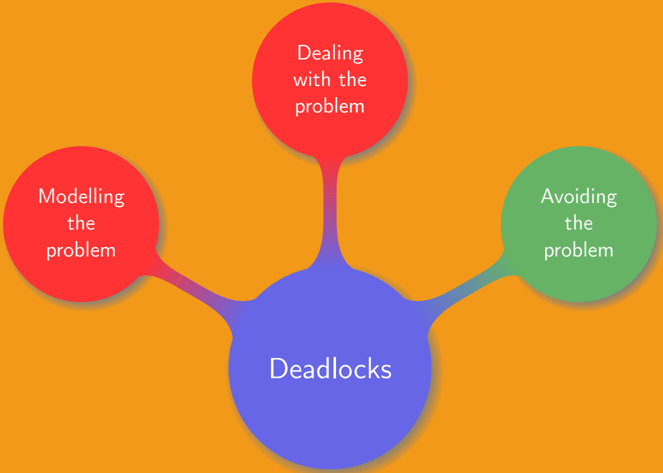
What if the second process requests two DVD burners?

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



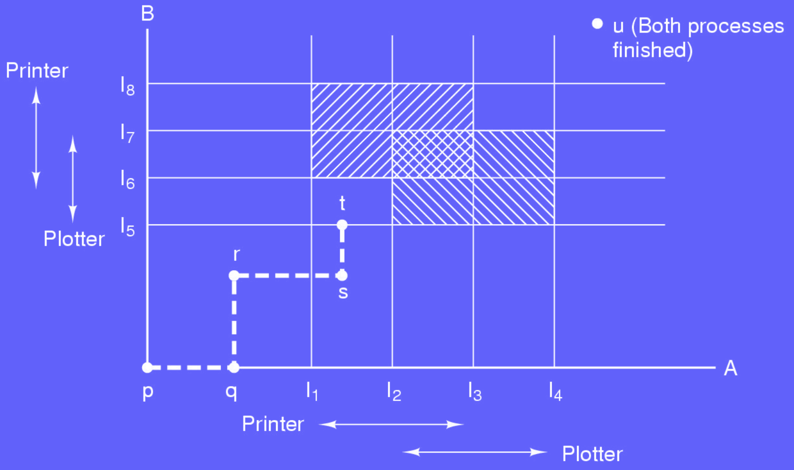
Avoidance vs. prevention

Two main strategies against deadlocks:

- Avoidance: resources are assigned and released one at a time.
  Is there any algorithm that can perform the right choice to avoid deadlocks?

- Prevention: when is a deadlock occurring?
  If it is possible to describe the circumstances under which deadlocks occur it might be possible to prevent them

Resources trajectories



Defining states

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

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

Defining states

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

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

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

The banker's algorithm

General idea:

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

The banker's algorithm

General idea:

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

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

The banker's algorithm

Deciding whether a state is safe or not:

1. Select a row in R whose resource request can be met. If no such row exists there is a possibility for a deadlock
2. When the process terminates it releases all its resource, and they can be added to the vector A
3. If all the processes terminate when repeating steps 1. and 2. then the 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:

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

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

The banker's algorithm

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

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

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

The banker's algorithm

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

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

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

The banker's algorithm

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

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

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

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

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

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

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

Conclusion: possible, but far from optimal

No preemption condition

Preventing deadlocks:

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

Conclusion: not viable to break this condition

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?

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

A few more remarks

Deadlocks are not necessarily related to hardware resources:

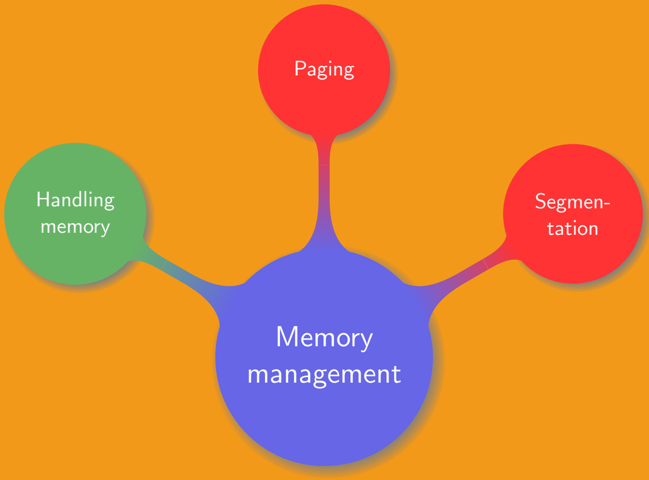
- Database records: two-phase locking solution; lock all the records in phase one; if one fails release all the locks and retry later, otherwise proceed with phase two and manipulate the records
- Communication deadlocks: mutex can lead to deadlocks; no hardware resource involved; could be due to the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

Key points

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

Address space

No abstraction leads to two main problems:

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

Address space

No abstraction leads to two main problems:

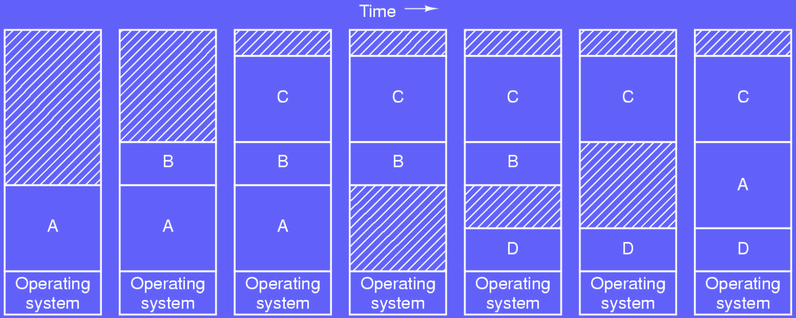
- Protection: prevent program from accessing other's memory
- Relocation: rewrite address to allocate personal memory

A solution is to set an address space:

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

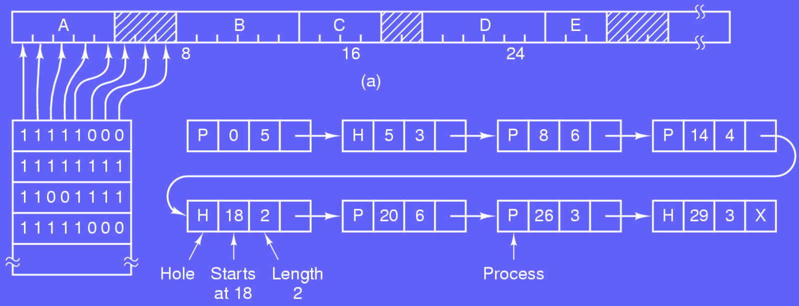
Memory size limitation

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



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

Bitmap and linked lists



Simple idea:

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

Allocating memory

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

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

Allocating memory

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

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

Characteristics:

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

Basics idea

Virtual memory:

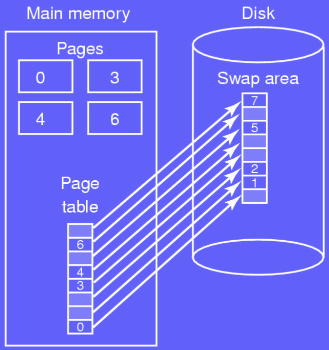
- Generalisation of the base and limit registers
- Each process has its own address space
- The Address space is split into chunks called **pages**
- Each page corresponds to a range of addresses
- Pages are mapped onto physical memory
- Pages can be on different medium (e.g. RAM and swap)

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

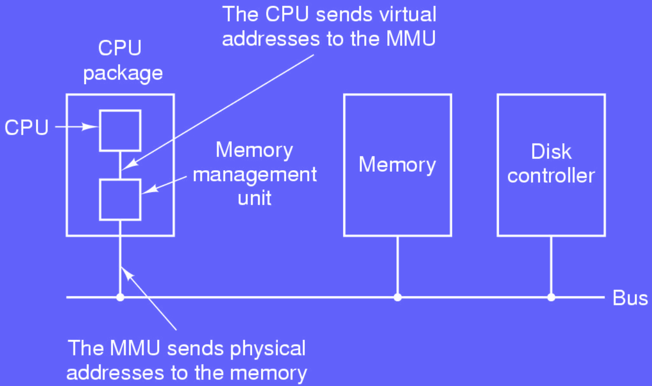
Initialising the swap area



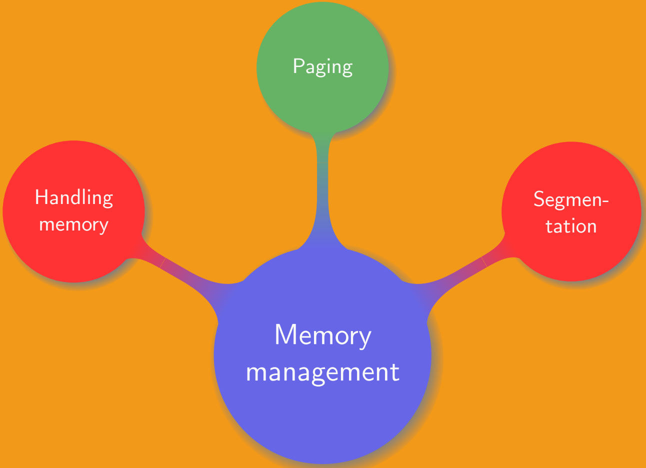
Two main strategies:

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

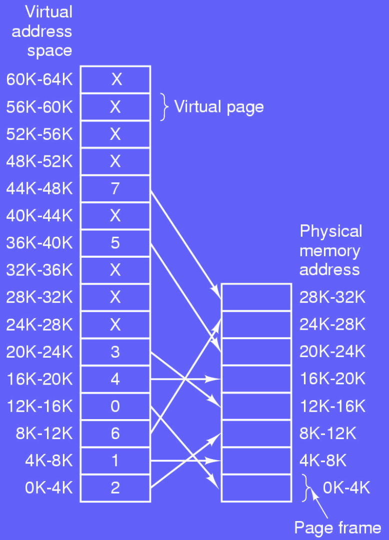
Memory Management Unit



Chapter organisation

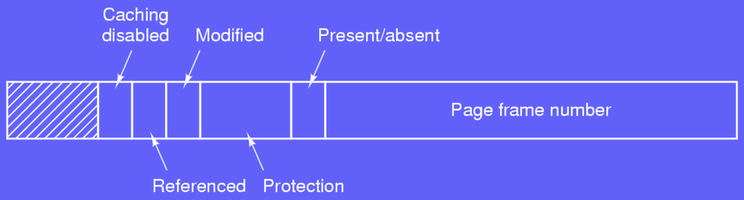


Virtual page and page frame



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

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

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

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

Page replacement – Optimal solution‘

Determining which page to remove when a page fault occurs:

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

Page replacement – Optimal solution‘

Determining which page to remove when a page fault occurs:

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

Problem: can the information be known ahead of time?

Page replacement – LRU

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

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

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

Page replacement – LRU

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

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

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

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

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

The *M* and *R* bits

M and *R* bits contained in the page table

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

Page replacement – Aging

Goal: simulate the LRU in software

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

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

Page replacement – Aging

Goal: simulate the LRU in software

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

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

t	t_0	t_1	t_2	t_3
R	$\begin{bmatrix} 1 & 0 & 1 & 0 \end{bmatrix}$	$\begin{bmatrix} 1 & 1 & 0 & 0 \end{bmatrix}$	$\begin{bmatrix} 1 & 1 & 0 & 1 \end{bmatrix}$	$\begin{bmatrix} 1 & 0 & 0 & 0 \end{bmatrix}$
$p1$	10000000	11000000	11100000	11110000
$p2$	00000000	10000000	11000000	01100000
$p3$	10000000	01000000	00100000	00010000
$p4$	00000000	00000000	10000000	01000000

Page replacement – Aging

Goal: simulate the LRU in software

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

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

t	t_0	t_1	t_2	t_3
R	$\begin{bmatrix} 1 & 0 & 1 & 0 \end{bmatrix}$	$\begin{bmatrix} 1 & 1 & 0 & 0 \end{bmatrix}$	$\begin{bmatrix} 1 & 1 & 0 & 1 \end{bmatrix}$	$\begin{bmatrix} 1 & 0 & 0 & 0 \end{bmatrix}$
$p1$	10000000	11000000	11100000	11110000
$p2$	00000000	10000000	11000000	01100000
$p3$	10000000	01000000	00100000	00010000
$p4$	00000000	00000000	10000000	01000000

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

Definitions

Basic notions related to paging:

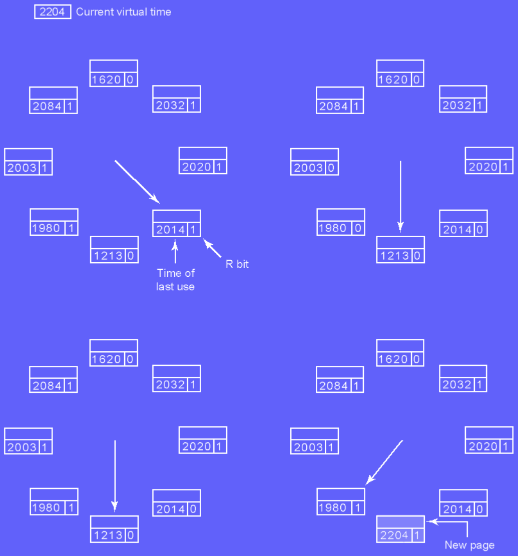
- Demand paging: pages are loaded on demand
- Locality reference: during an execution phase a process only access a small fraction of all its pages
- Working set: set of pages currently used by a process
- Thrashing: process causes many page fault due to a lack of memory
- Pre-paging: pages loaded in 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

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

- On a page fault examine the pages the hand points to
- If $R = 1$, bad candidate: set R to 0 and advance hand
- If $R = 0$, 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: within the process $\Rightarrow$ allocate a portion of the whole memory to a process and only use this portion; number of page frames for a process remains constant
- Global: within the whole memory $\Rightarrow$ dynamically allocate page frames to a process; number of page frames for a process varies over time

Local vs. global allocation

Onto which set should the page replacement algorithm be applied:

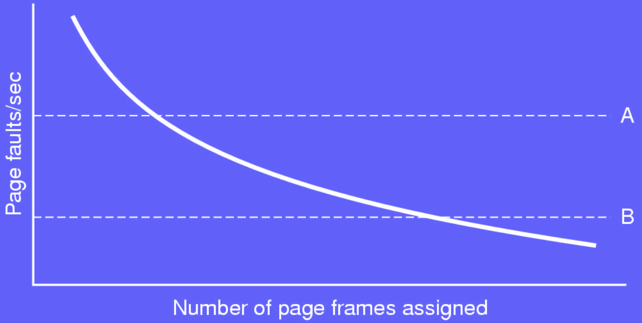
- Local: within the process $\Rightarrow$ allocate a portion of the whole memory to a process and only use this portion; number of page frames for a process remains constant
- Global: within the whole memory $\Rightarrow$ dynamically allocate page frames to a process; number of page frames for a process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

Which approach is best?

Page fault frequency

Adjusting the number of pages:

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



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

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

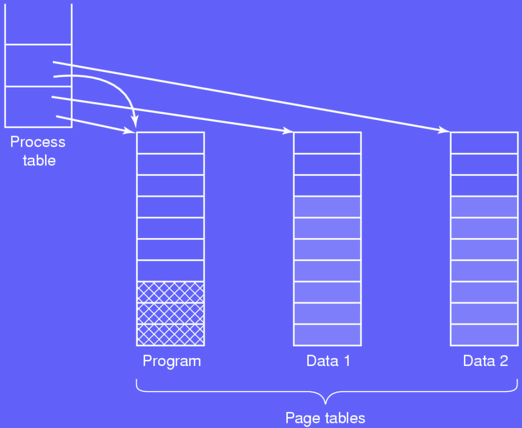
When a same program is run by different users at the same time, then sharing pages decreases memory use:

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

Page sharing

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

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



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?

When to use paging?

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

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

When to use paging?

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

- **Page fault:** (i) read hardware register to determine origin of page fault; (ii) compute which page is needed; (iii) locate the page on the disk; (iv) find an available page frame and replace its content; (v) read the new page frame; (vi) rewind to the faulting instruction and re-execute it

- **Process termination:** (i) release page table, pages and disk space; (ii) beware of any page that could be shared among several processes

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 used such as not to be used by another process

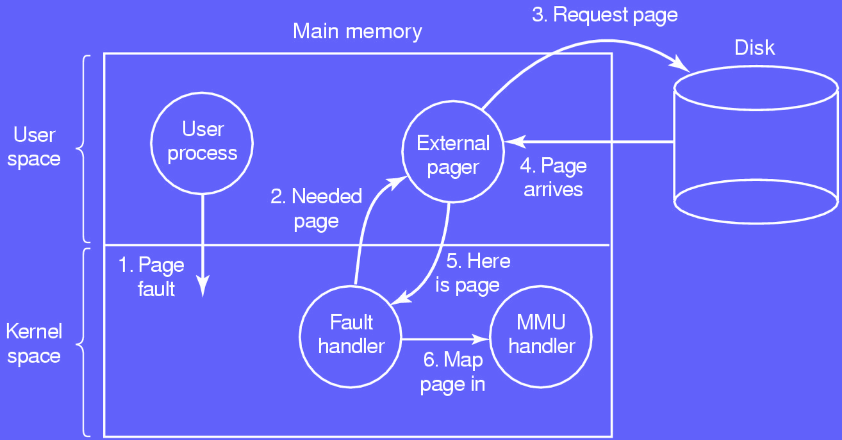
Page fault handling

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:

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



Organisational issue

Where should the page replacement algorithm go:

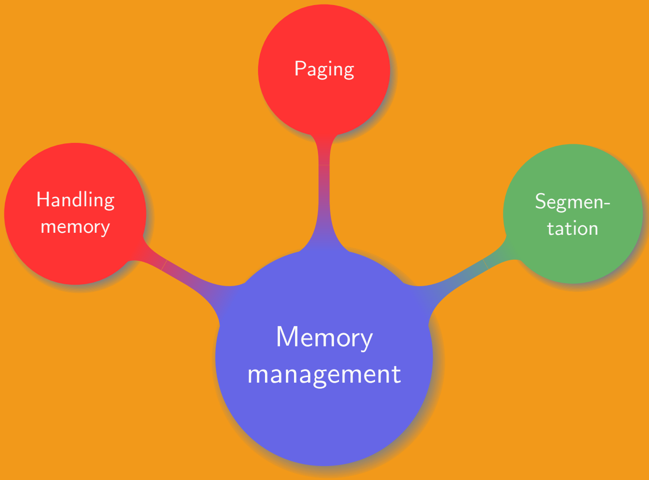
User space

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

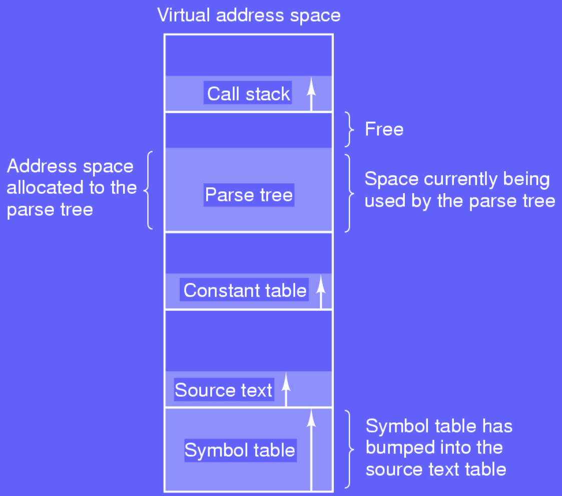
Kernel space

- 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

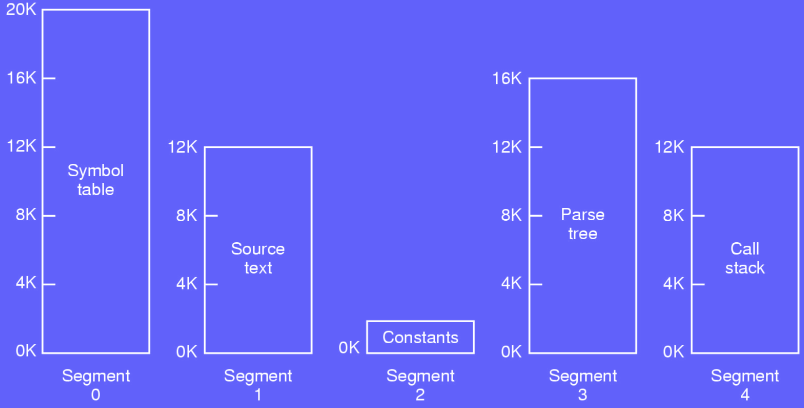


Paging



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

Segmentation

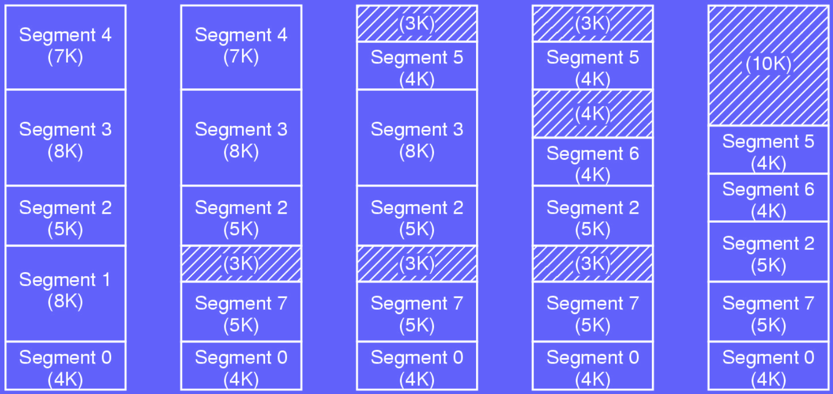


Address translation

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

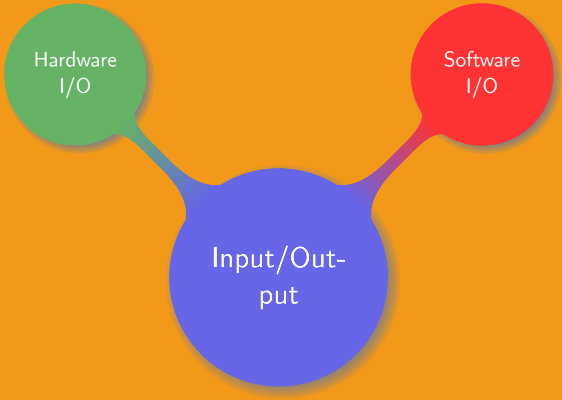
Considerations	Paging	Segmentation
Number of linear address space	1	many
Limited by the size of the physical memory	no	no
Possible to separate and protect data and procedures	no	yes
Sharing procedures between users or programs	complex	easy

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 I/O:

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

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

Representing devices

Two main categories:

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

Representing devices

Two main categories:

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

Device controller

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

The electronic part is called the device controller:

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

Memory-mapped I/O

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

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

Modern approach:

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

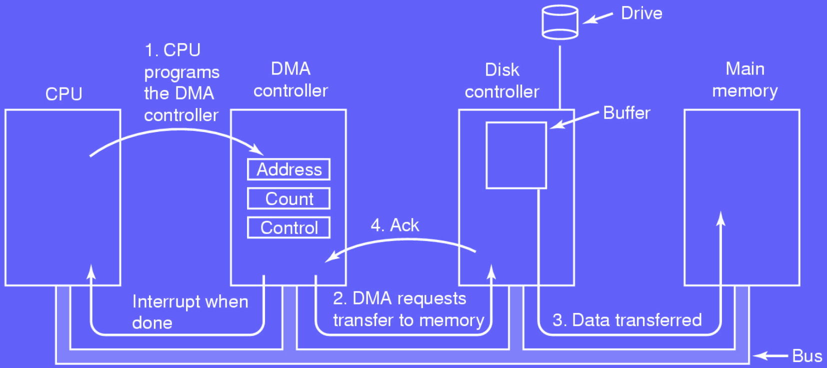
Memory-mapped I/O – Pros and cons

Strengths:

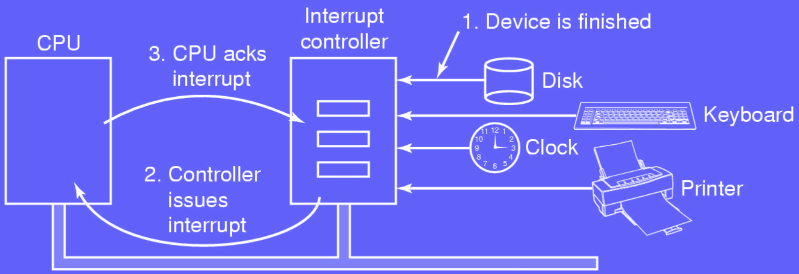
- Access memory not hardware $\rightarrow$ no need for assembly
- No special protection required $\rightarrow$ control register address space not included in the virtual address space
- Flexible $\rightarrow$ a specific/privileged user can be given access to a particular device
- Different drivers in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

Weakness: memory words are cached $\rightarrow$ what if the content of control register is cached?

Direct Memory Access



Interrupts


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

Precise vs. imprecise interrupts

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

New setup: pipelined or superscalar CPU. What consequences?

Precise vs. imprecise interrupts

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

New setup: pipelined or superscalar CPU. What consequences?

Precise interrupt:

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

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

Dealing with imprecise interrupts

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

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

Conclusion: slow interrupts

Dealing with imprecise interrupts

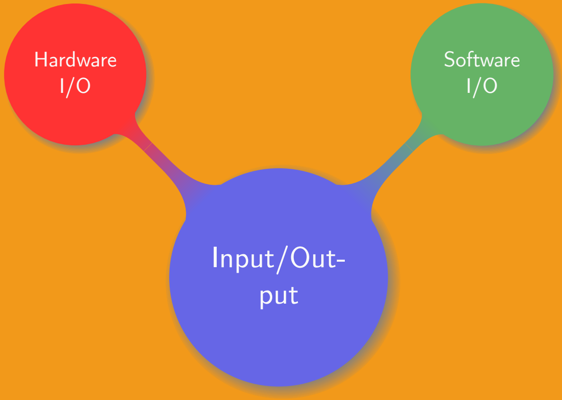
Difficult to figure out what happened and what has to happen:

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

Conclusion: slow interrupts

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

Chapter organisation



Key concepts

Main goals on the design of I/O software:

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

Communications strategies

Three communications strategies:

Communications strategies

Three communications strategies:

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

Layers

User Level I/O Software

Device Independent OS Software

Device Drivers

Interrupt Handlers

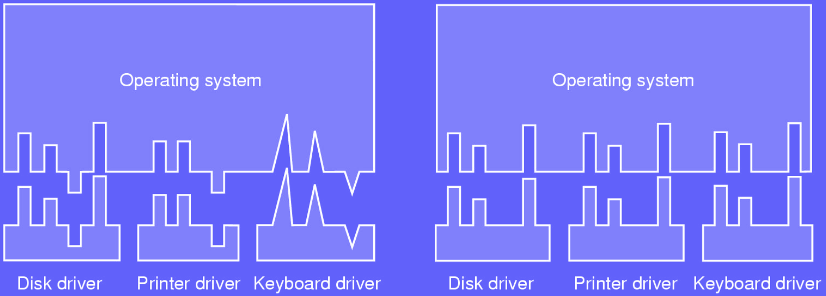
Hardware

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



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

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

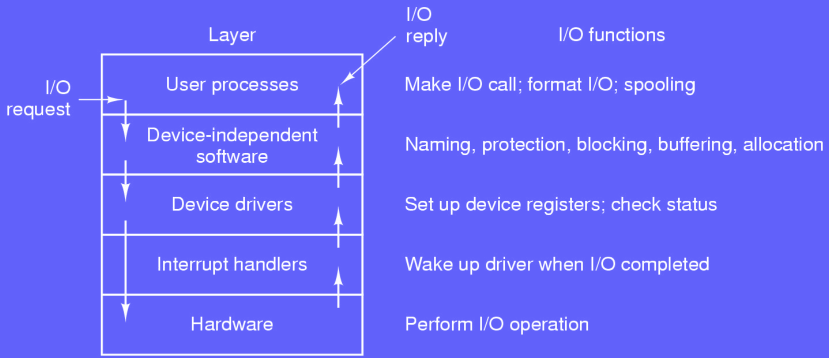
Device driver should react nicely even under special circumstances

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 (manage MMU, timers, DMA etc...)
- I/O errors framework is device independent
- Clean and generic API such that it is easy to write new drivers

Software I/O and OS

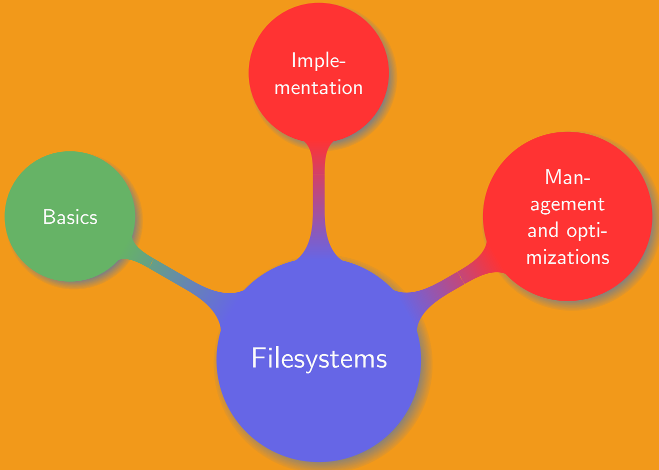


Key points

- 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?

8. Filesystems

Chapter organisation



Goals

Limitations of virtual memory:

- Small
- Volatile
- Process dependent

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 operations

Common system calls related to files (Unix):

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

File organisation

Structure content:

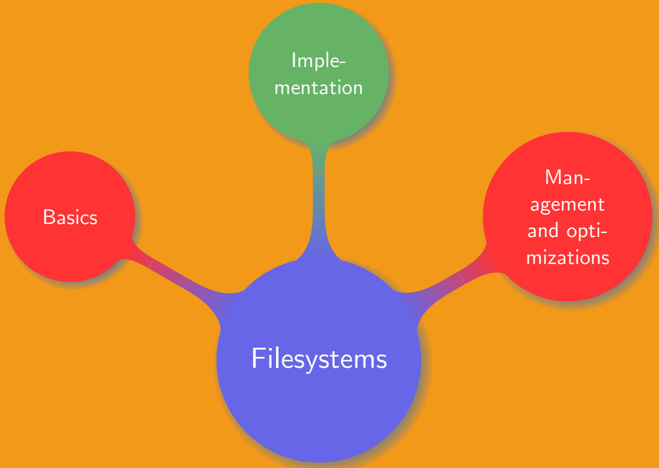
- Files are grouped inside a *directory*
- Directories are organised in a *tree*
- Each file has an *absolute path* from the root of the tree
- Each file has an *relative path* from the current location in the tree

Directory operations

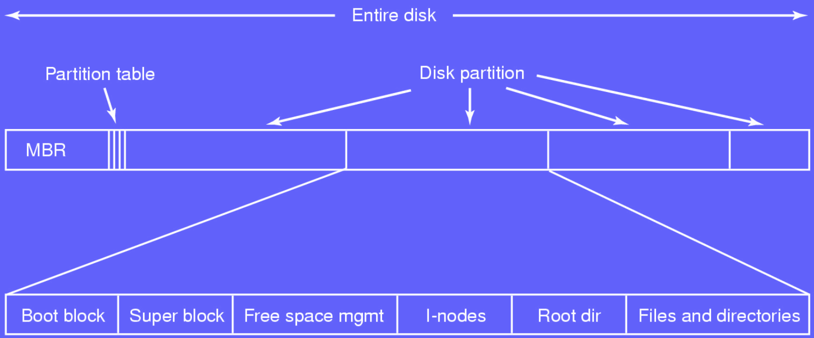
Common system calls related to directories (Unix):

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

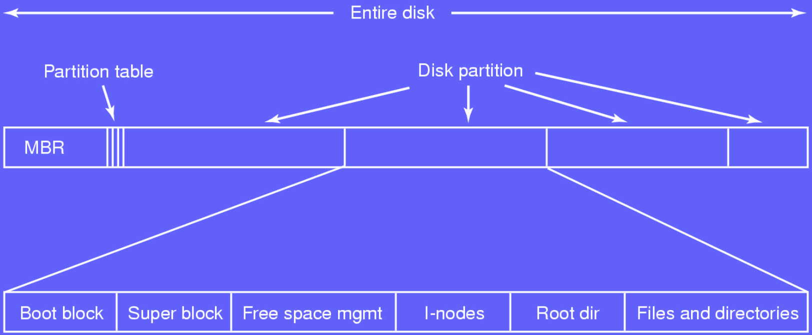
Chapter organisation



Basic disk layout

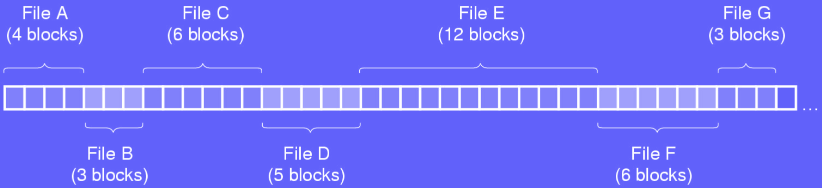


Basic disk layout



How to efficiently match disk blocks and files?

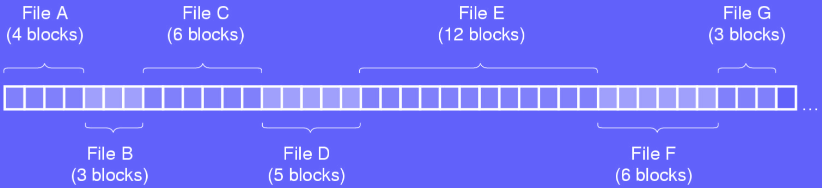
Contiguous allocation



Advantages:

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

Contiguous allocation

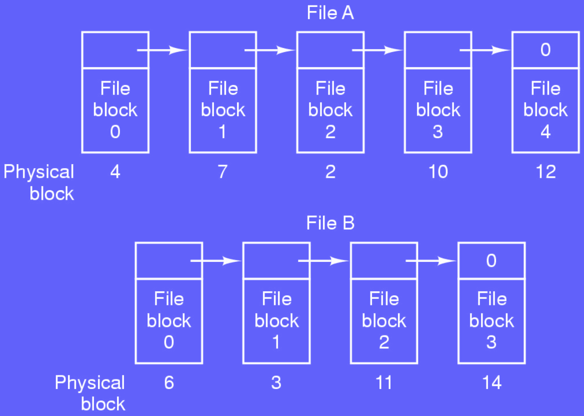


Advantages:

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

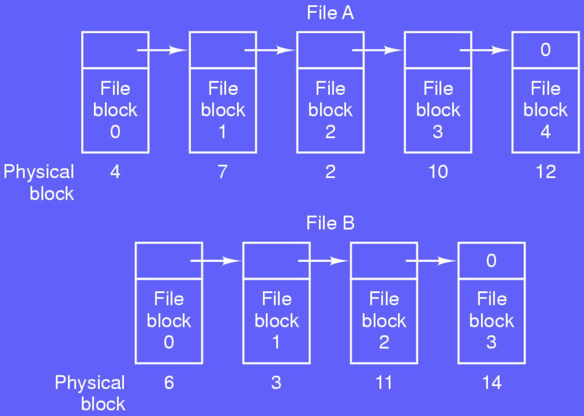
Drawback: what if files *D* and *F* are deleted?

Linked list



Advantage: no fragmentation

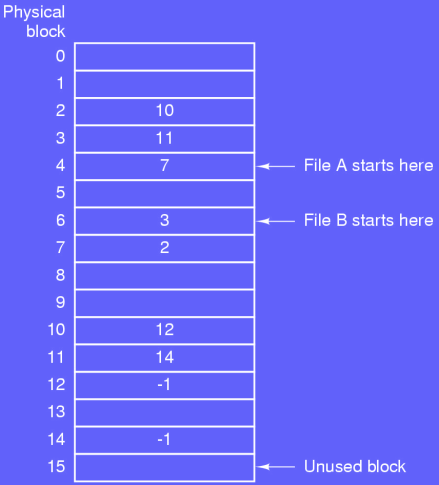
Linked list



Advantage: no fragmentation

Drawback: slow random access

File Allocation Table



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

Advantage: fast random access

File Allocation Table

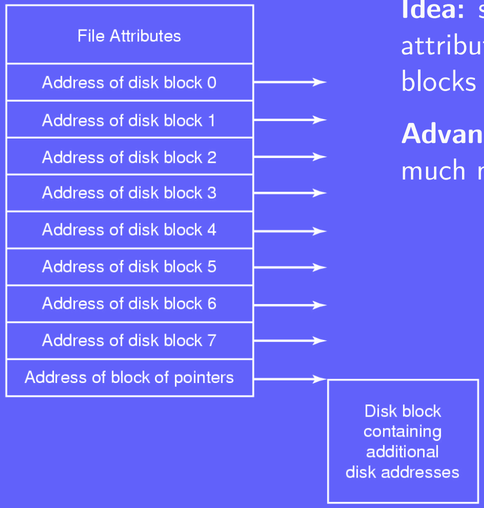
Physical block		
0		
1		
2	10	
3	11	
4	7	← File A starts here
5		
6	3	← File B starts here
7	2	
8		
9		
10	12	
11	14	
12	-1	
13		
14	-1	
15		← Unused block

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

Advantage: fast random access

Drawback: memory usage

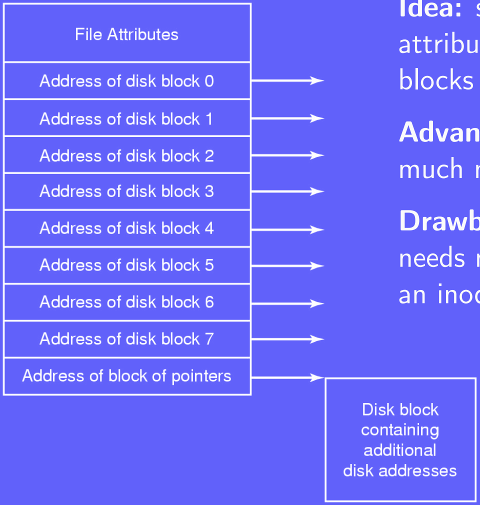
Index node



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

Advantage: fast, do not require much memory

Index node



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

Advantage: fast, do not require much memory

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

Directories

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

Directories

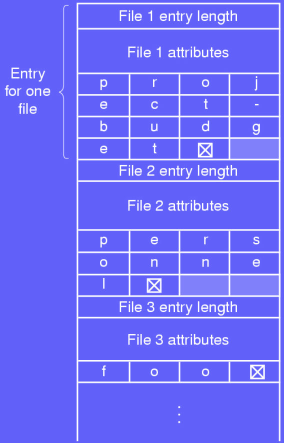
Simple design: fixed size entry (filename, attributes, disk address)

Drawback: how to handled long filenames?

Directories

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

Drawback: how to handled long filenames?



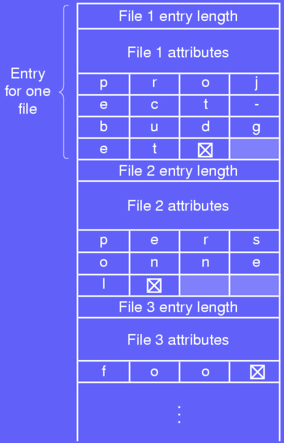
Idea: filename length not fixed

Advantage: can fit filename of arbitrary length

Directories

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

Drawback: how to handled long filenames?

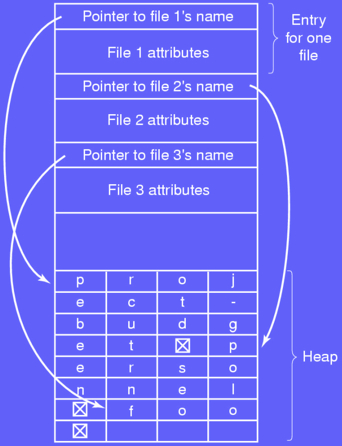


Idea: filename length not fixed

Advantage: can fit filename of arbitrary length

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

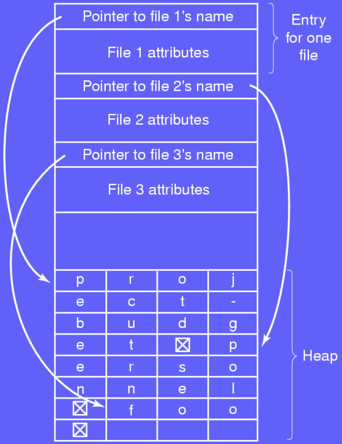
Directories



Idea: pointer to the filename

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

Directories



Idea: pointer to the filename

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

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

Journaling FS

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

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

Journaling FS

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

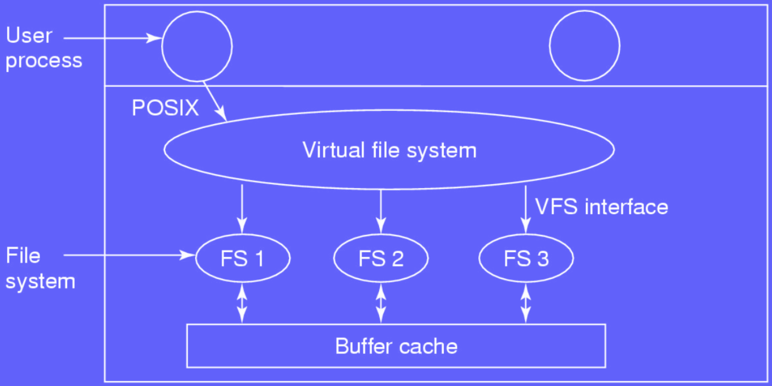
Strategy: if a crash interrupts an operation, re-run it on next boot

Problem: can any operation be applied more than once?

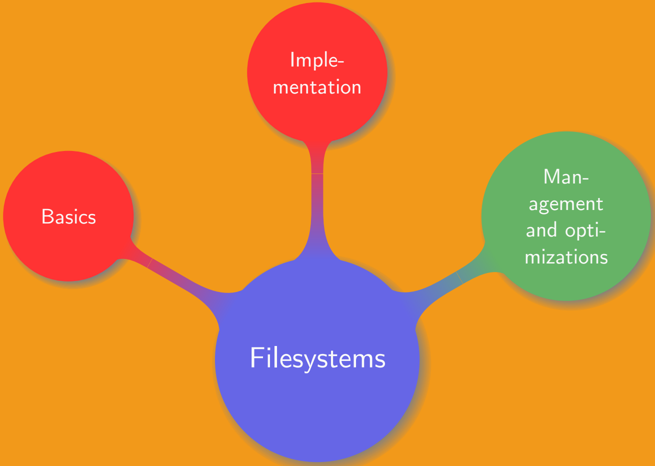
Example.
File deletion:

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

Virtual FS


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

Chapter organisation



Block size

Problem: how big should a block be?

Using small blocks:

- Large files use many blocks
- Blocks are not contiguous

Conclusion: time wasted

Block size

Problem: how big should a block be?

Using small blocks:

- Large files use many blocks
- Blocks are not contiguous

Conclusion: time wasted

Using large blocks:

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

Conclusion: space wasted

Free blocks

Problem: how to keep track of free blocks?

Free blocks

Problem: how to keep track of free blocks?

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

Which strategy is best?

FS consistency

Checking the FS:

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

FS consistency

Checking the FS:

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

Common problems and solutions:

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

Caching

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

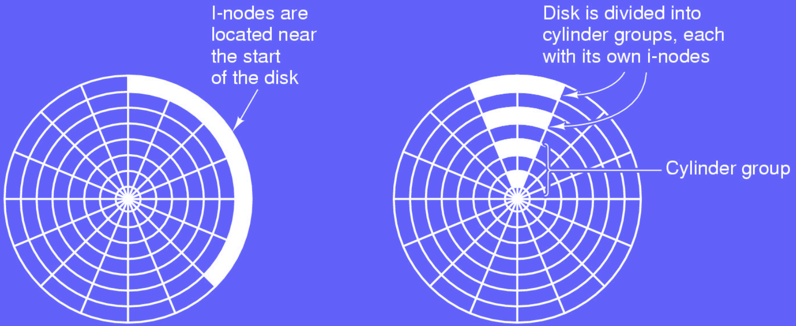
Questions:

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

Modified idea:

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

Arm motion


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:

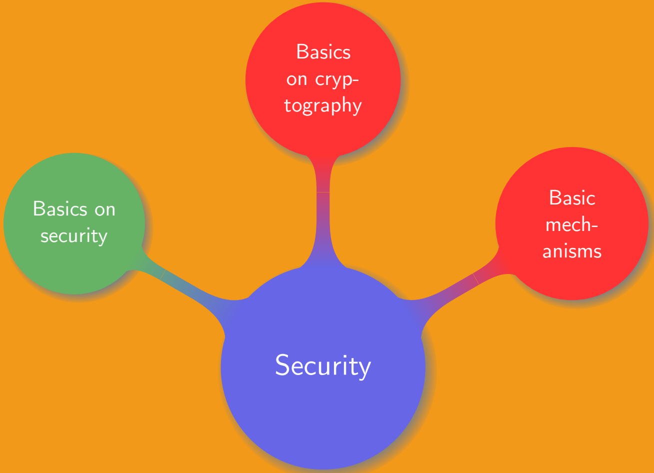
- Quotas: assign disk quotas to users
- Fragmentation: how useful is it to defragment a file system?
- Block read ahead: when reading block k assume $k + 1$ will soon be needed and ensure its presence in the cache
- Logical volumes: file system over several disks
- Backups: how to efficiently backup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

9. Security

Chapter organisation



What is security?

Simple reasoning:

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

What is security?

Simple reasoning:

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

What are the dangers?

Setup and dangers

To define the dangers, the setup must be known:

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

Threats

In an OS threats can be divided into four categories:

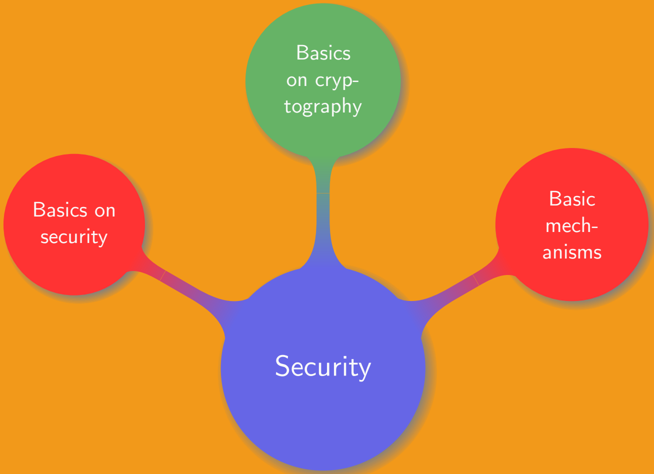
- Data stolen: confidentiality
- Data changed: integrity
- Intrusion: exclusion of outsiders
- Denial of service: system availability

Who?

Knowing who is likely to attack is of a major important:

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

Chapter organisation



Security and cryptography

Cryptography, the science of secret:

- Confidentiality
- Data integrity
- Authentication

Two basic strategies:

- Symmetric
- Asymmetric

Confidentiality

Encrypted data will remain confidential. How to encrypt data?

Symmetric:

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

Question: are those strategies secure?

Confidentiality

Encrypted data will remain confidential. How to encrypt data?

Symmetric:

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

Question: are those strategies secure?

Asymmetric:

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

Symmetric protocols better fit the OS setup

Data integrity

Ensure that data has not been altered using hash functions:

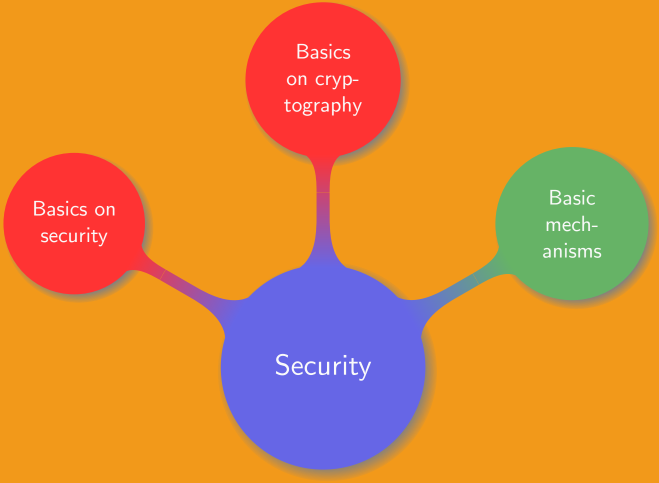
- Easy to compute
- Infeasible to generate a message with a given hash
- Infeasible to modify a message without modifying the hash
- Infeasible to find two different messages with same hash

Authentication

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?

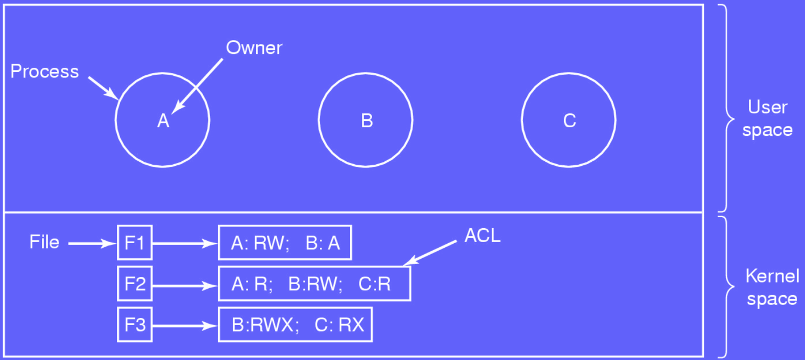
Authentication

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?

Other common strategy: use a key

Access Control Lists



Security layers

ACL are used to give users different privileges:

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

Keeping a system secure – Basic

Remark.
An OS cannot be kept 100% secure

Keeping a system secure – Basic

Remark.
An OS cannot be kept 100% secure

Basic strategy:

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

Keeping a system secure – Advanced

Advanced strategy:

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

Keeping a system secure – Paranoiac

Paranoiac strategy:

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

Keeping a system secure – Paranoiac

Paranoiac strategy:

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

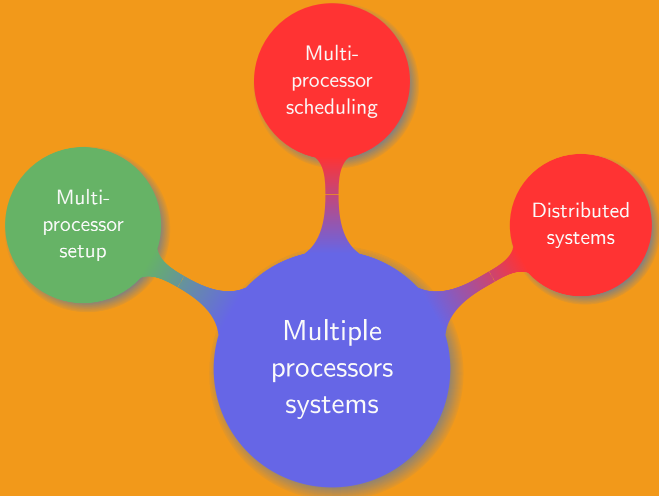
Is it now safe?

Key points

- What is security?
- What is the difference between symmetric and asymmetric cryptography?
- What are Access Control Lists?
- How safe can a computer be?

10. Multiple processors systems

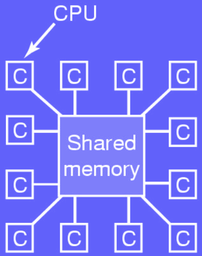
Chapter organisation



Shared memory model

Multiprocessors:

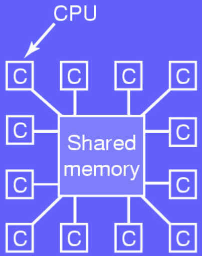
- CPUs communicate through the shared memory
- Every CPU has equal access to entire physical memory
- Access time: 2-10 ns



Shared memory model

Multiprocessors:

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



Three main approaches:

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

SMP approach

One copy of the OS that can be run by any of the CPUs

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

SMP approach

One copy of the OS that can be run by any of the CPUs

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

Solution:

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

SMP approach

One copy of the OS that can be run by any of the CPUs

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

Solution:

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

Challenges:

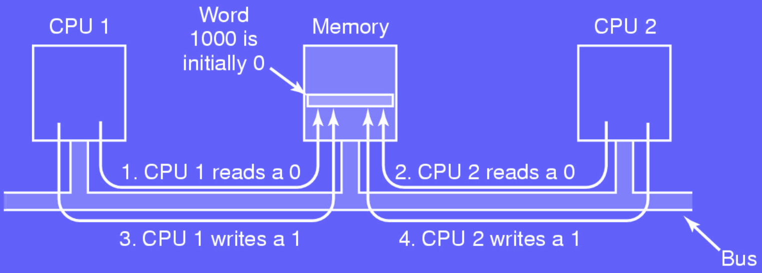
- How to divide up the OS
- Easy to run into deadlock with the shared tables
- Hard to keep consistency between programmers

Synchronisation – Problem

Synchronisation strategy with a single CPU:

Synchronisation – Problem

Synchronisation strategy with a single CPU: TSL instruction



Synchronisation – Solution

The `TSL` instruction should:

1. Lock the bus by asserting a special line on the bus
2. Test and Set the Lock
3. Unlock the bus

Synchronisation – Solution

The `TSL` instruction should:

1. Lock the bus by asserting a special line on the bus
2. Test and Set the Lock
3. Unlock the bus

New multiprocessor issue: slows down the whole system

Solution: use a local cache such as not to block the bus to often

Cache with multiprocessors

Multiprocessors cache implementation:

- Requesting CPU reads the lock and gets a copy in its cache
- Polling is done using the value in cache
- When the lock is remove the cache protocol invalidates all the remote copies
- Updated value is to be fetched

Cache with multiprocessors

Multiprocessors cache implementation:

- Requesting CPU reads the lock and gets a copy in its cache
- Polling is done using the value in cache
- When the lock is remove the cache protocol invalidates all the remote copies
- Updated value is to be fetched

Problem:

- Mutex is 1 bit, but a whole block is copied
- `TSL` require write access
- Cached block that is modified is invalidated
- Cache need to be recopied

Read and TSL

New approach:

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

Read and TSL

New approach:

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

Problem: what if two CPUs see the lock being free and apply the `TSL` instruction? Does it lead to a race condition?

Read and TSL

New approach:

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

Problem: what if two CPUs see the lock being free and apply the `TSL` instruction? Does it lead to a race condition?

- The value returned by the read is only a hint
- Only one CPU gets the lock
- The `TSL` instruction prevent 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

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
- Next CPU on the list can start its work

Spinning or switching?

New perspective:

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

Dilemma: switching is expensive but looping is a waste

Best choice:

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

Spinning or switching?

New perspective:

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

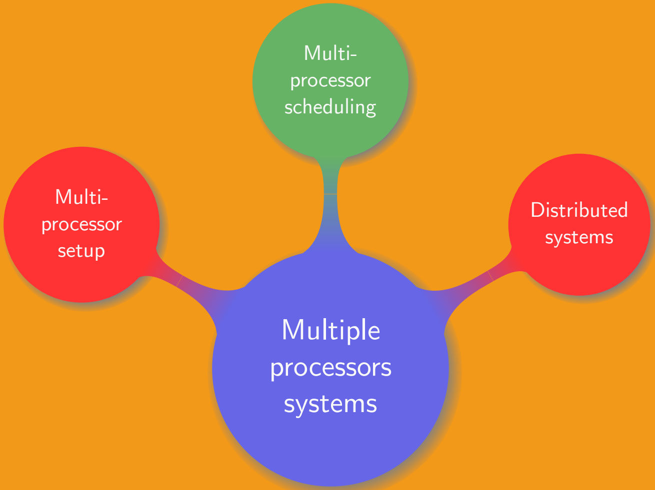
Dilemma: switching is expensive but looping is a waste

Best choice:

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

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

Chapter organisation



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?

Setup

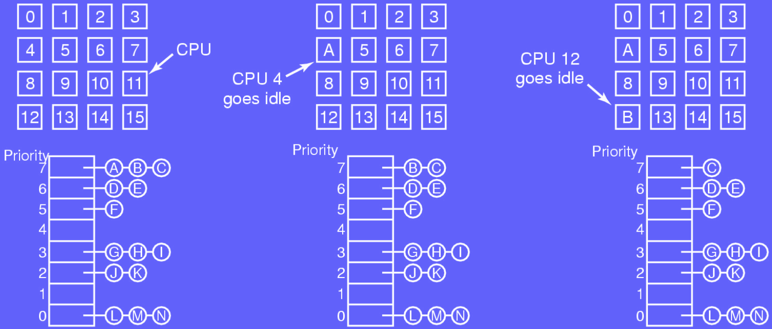
Single threaded process: only need to schedule the process

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

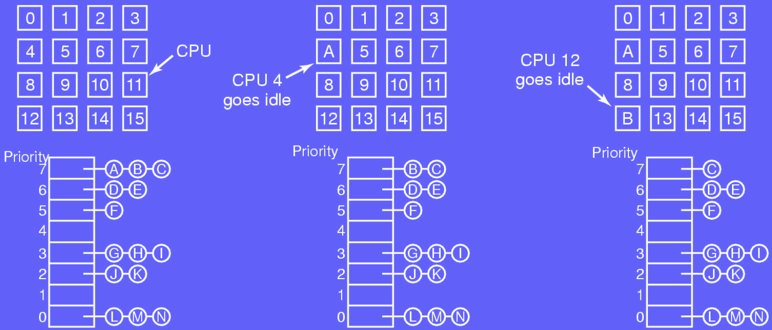
Time-sharing



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

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

Time-sharing



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

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

Improvements on time-sharing

Smart scheduling:

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

Affinity scheduling:

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

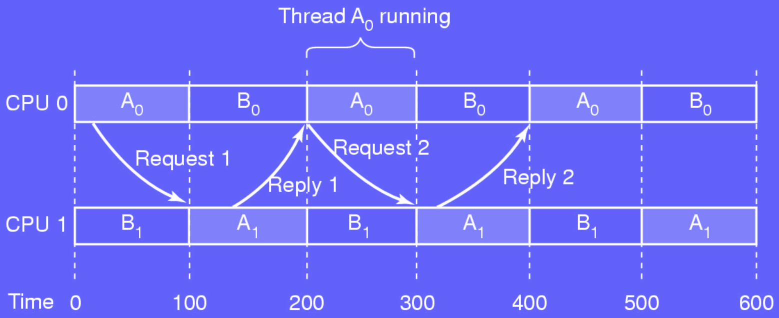
Space-sharing

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

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

Optimization: try to apply shortest job first $\rightarrow$ in practice FIFO better

The communication problem



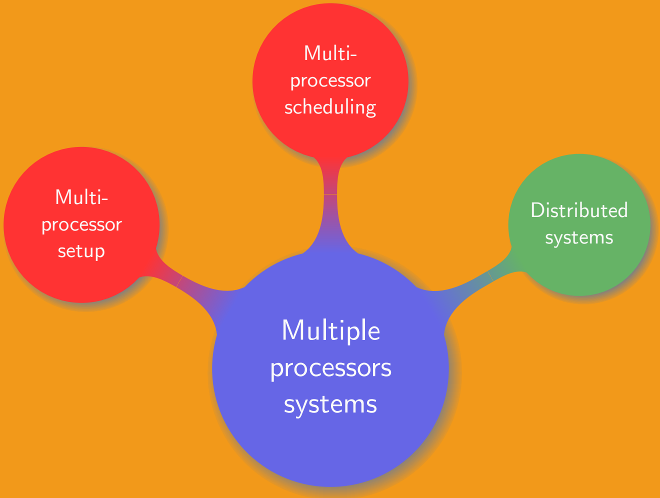
Gang scheduling

Simple idea: schedule processes by group:

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

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

Chapter organisation



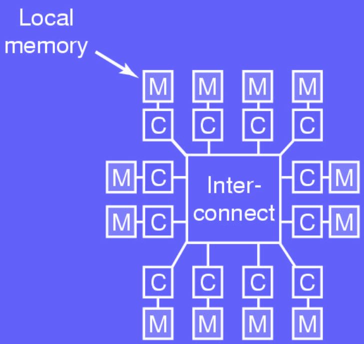
Main characteristics

Characteristics of distributed systems:

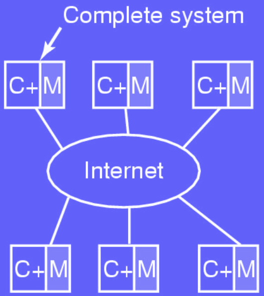
- Composed of autonomous entities, each with its own memory
- Communication is done over a network using message passing
- The system must tolerate node failures
- All the nodes perform the same task

Cluster: set of connected computers working together

Structure



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 handles the management of the scheduling and slave nodes

Main challenges:

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

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!