Operating System; general class

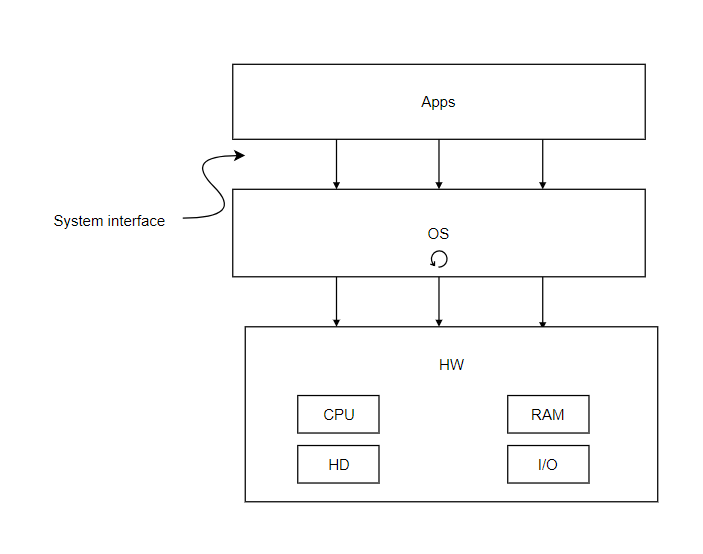
11.09.2018

# Introduction

### 1/ Presentation note :

* MP on campus
* Always 2 weeks to do lab TP

### 2/ Def



* Manage the resources
* It’s a software
* What’s happening inside the os? Algorithms
* Apps go through what’s inside the os
* How it runs cpu, stores data, manages files?

**Interface**

* Given by the OS in order to use resources we have.
* It gives you the permissions needed to use your programs.
* Lowest level -> System Programming

### 3/ Labs

* Small reports at each lab, explanations of why it happened
* Don’t care about results, if good explanations -> highest grade
* At least you tried
* Linux based, class on general OS

### 4/Grading (maybe)

* 50% labs
* 50% final quizz (QCM)
* Questions about labs

### 5/ Summary :

1/ Managing processes

2/ Managing memory (volatile)

3/ Managing persistent memory (filesystems)

4/ Managing I/O

# Managing Processes

*What it means to run an app?*

### 0/ Definition

*What is the difference between a program and a process?*

What is a **program(an executable)?**

* CPU instructions (binary code)
* “Some kind of data”

What is a **process?**

* A running program/executable

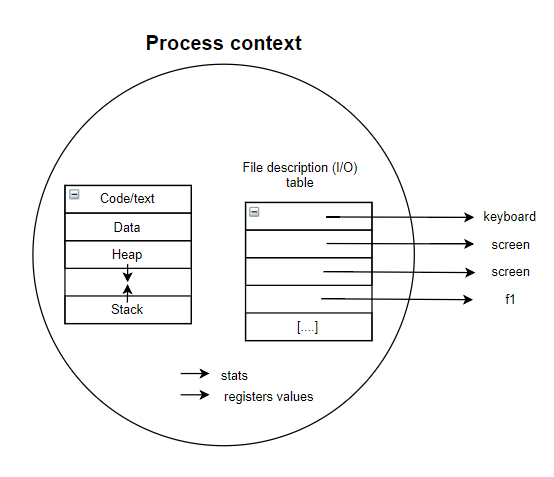
### 1/ Process specification

What does a process contain?

**Execution context / environnement**

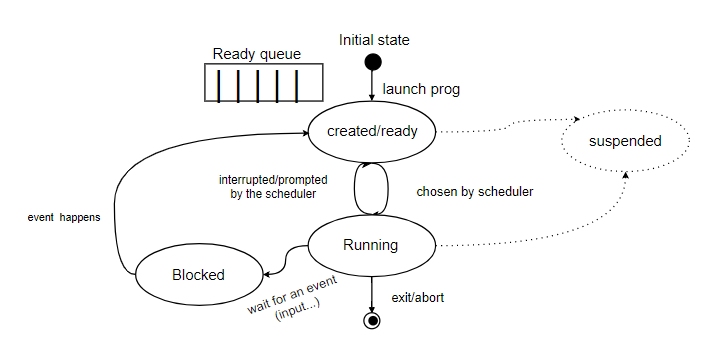
* CPU instructions (code) = **text segment**
* **Global data** **segment** : global/static variables, constants
* **Heap** : moving space. It contains dynamic allocations (malloc, new…)
  + You have to go to the heap to free the space (remove @ of your variable which is in your stack for eg) OR you stock the variable itself in the function frames, this way it is removed automatically when its execution is finished
* **Execution stack** (if it’s sequential, you don’t need one). Also a moving part
* **File Descriptor (I/O) table** : it keeps track on what is happening to your files, their interactions (which ones are opened for eg)
  + Standard Input (keyboard)
  + Standard Output (screen)
  + Standard Error(screen)
  + Files 1 … X
* Statistics : how much time the CPU was running? % ?
* Register values : you have to save this values

There is a lot more, but these are the fundamental ones.



One different context per process. In **multitasking systems** each context is totally **isolated** from the others for obvious security reasons.

### 2/ Lifecycle of a process



*How does an object changes from one state to another?*

**Create a process :** launch its context

* When it’s **created**, if you doesn’t want to run it, it goes to the ready queue.
* If the scheduler chose it then it’s the one process **running**. The running process can be put back in a ready state so the processes on the ready queue can also be executed.
* There is a **blocked** state since we don’t want our process to be running without doing anything. It’s waiting for something, it’s in the blocked queue. When the event happens, then the transition happens too, and the process is once again ready to be selected by the scheduler.
* The **suspended state** is not fundamental and can happen whenever you need it

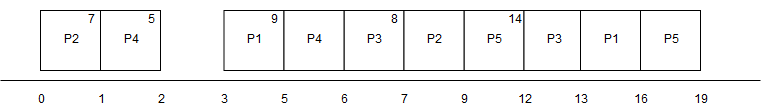
## Process Table examples :

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | Arrival Time | Running Time | Priority | I / O |
| P1 | 3 | 5 | 5 | 2 / 4 |
| P2 | 0 | 3 | 4 | 1 / 6 |
| P3 | 5 | 2 | 3 | 1 / 1 |
| P4 | 1 | 2 | 2 | 1 / 3 |
| P5 | 7 | 6 | 1 | 3 / 2 |

For P1 if it was running in the CPU for 2, then at 2 after start takes input and take 4 to output and then do the 3 more to finish it’s process (because 5)

## How to choose the process order ?

### 1/ First Come First Serve (FCFS)



Turnaround Time (Global Time) : end of process - creation / arrival time from table

P1 (16 - 3 = 13)

P2 (9)

P3 (8)

P4 (5)

P5 (12)

Waiting Time : Turnaround time - Running time

P1 (13 - 5 = 8)

P2 (6)

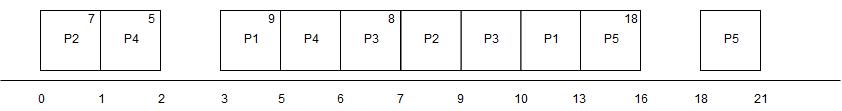
P3 (6)

P4 (3)

P5 (6)

### 2/ Shortest Job First

Don’t interrupt running process. On the ready queue when 2 processes at the same time, the one with the least time remaining is chosen.



Turnaround Time (Global Time) : end of process - creation / arrival time from table

P1 (13 - 3 = 10)

P2 (9)

P3 (5)

P4 (5)

P5 (14)

Waiting Time : Turnaround time - Running time

P1 (10 - 5 = 5)

P2 (6)

P3 (3)

P4 (3)

P5 (8)

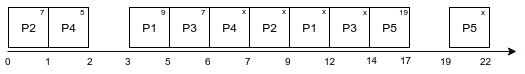
But problem with interactions, if long time to run in the middle of really short time it will never execute. You don’t know how much time it will take for a process to finish.

Starvation : starving for the CPU you’re not having, you want to run

To resolve problem -> Aging : Highering the priorities if you wait a long time

### 3/ Priority (preemptive) algorithm

Preemptive should be able to force a process to go from Running state to Ready state because of a lower priority



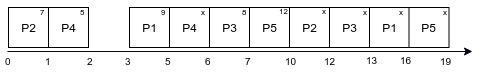
(not preempted here)

### 4/ Round Robin

**Quantum =** Load time for each task

* ~= 10ms (usually)
* Large quantums aren’t favorable for interaction ( = freezing)
* So tradeoff, small enough so all processes are running at the same time, no freezes

**Eg1 :** q = 3 ms



**Eg2 :**

With another set of processes :

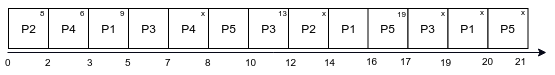
|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | Arrival Time | Running Time | Priority | I / O |
| P1 | 3 | 5 | 5 | 2 / 4 |
| P2 | 0 | 4 | 4 | 2 / 6 |
| P3 | 5 | 6 | 3 | 4 / 1 |
| P4 | 1 | 2 | 2 | 1 / 3 |
| P5 | 7 | 4 | 1 | 3 / 2 |

q = 2

Queue :

|  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| P5 | P1 | P3 | P5 | P1 | P2 | P3 | P5 | P4 | P3 | P1 | P4 | P2 |

Result :



### 5/ Switching context

*What happens when you switch context (process or thread) ?*

It is not instantaneous, it is memory management that takes time.

* If you choose a very small quantum vs a very big one. The larger the less context switches.
* The more you switch, the more you lose CPU time.
* Switching context in the same process (threads) takes less time than switching context between 2 processes because you have to replace the code, the data…

**Thread** = light weight process

### 6/ Multiple queues

* Usually, you have **multiple queues** during the scheduling.
* If the whole quantum is used by a process, it means that maybe it’s not interactive.
  + So, the system can switch it to another queue where the quantum is smaller. It’s **responsive**.
* There is a **priority based algorithm** between the different queues.

*What if your process is interactive, but just not in the beginning*?

* It doesn’t use its whole quantum in the new queue.
* The system pushes it back in its old queue. It has a strategy.
* Since it has a priority based algorithm, it is scheduled very quickly.

### 7/ Synchronisation

#### Sharing problem (race condition)

Eg :

i <- 5

**T1 T2**

I++ i--

Final value should be : 5.

But it can end up being 4 or 6.

The main problem is that we’re sharing the same data, and trying to write on it at the same time.

*Solution :*

* do it one by one: sequentially.
* **mutual exclusion :** if someone is using it, it’s going to exclude the others.
* You have to make an **atomic/indivisible (uninterruptible)** operation.

|  |  |  |
| --- | --- | --- |
| **T1** |  | **T2** |
| reg<-i  reg++  i<-reg | reg<-i  reg--  i<-reg |

**Problem :** doing things sequentially isn’t the most performant way. Things are going to be slower, and aren’t parallelized.

#### Solution (tool)

1- Whenever a user comes in -> add to waiting list

2- 4 computers -> 1st 4 on the list

3- When a user is done, give the computer to the 1st on the waiting list

Structure : counter (# of permissions/tokens/resources)

Waiting list.

|  |  |
| --- | --- |
| **Acquire** | **Release** |
| if(no resource is available)  Add task to waiting list  Else  Counter -- | Counter ++  If (waiting list is not empty)  Wakeup a task from the waiting list |

To apply mutual exclusion to the previous program : you apply and release semaphore to both sides.

|  |  |  |
| --- | --- | --- |
| **T1** |  | **T2** |
| Aquire(S)  reg++  Release(S) | Aquire(S)  reg--  Release(S) |

When you acquire/release you enforce mutual exclusion.

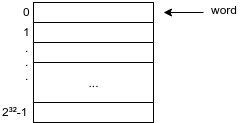
Semaphore & counter = shared variables => same problem that with i

# Memory management

**Memory is a set of slots**; one such slot is called a **word**.

Main operations: **reading** and **writing**, **allocating**, and **freeing** memory. Before reading or writing, you need to allocate space. Reading or writing is straightforward. There is no algorithm in these cases.

You need an **address**. The number of possible indexes (for addresses) depends on the address bus (for example, for 32bits you have 2^32 - 1 possible indexes).

**Start with allocating first!** (use **malloc**() function, returns a pointer)

Size : 2³² / 2³ = 2²⁹bytes = 512MB

Example of memory allocation function:

Int main() {

Int \*ptr = malloc(5) ; ← could return the list of slots allocated by the malloc() function (could also be a link list)

ptr[0] = 55;

ptr[4] = 77;

}

(but before running this code, system has already started allocating memory)

**What will happen in memory? Possible example (contiguous method):**

Allocation (nb) :

1. Find first + second available slots

* Then continue to the nth slot

2. Return the address of the 1st slot

*Is this* ***contiguous*** *or no*t? (Contiguous = next in sequence)

The **contiguous** method is easier to implement and is more efficient, because it is easier to determine which slots are used in memory. In this case, you just need to have the base address and the displacement (i.e the number of slots used) to determine all slot addresses that were used.

To allocate in a **direct access scheme** :

* we first find a number of contiguous slots
* return the base address of step 1

The **non-contiguous** method may take more space in memory, because the linked list created this way would take more space in memory, and take more time to access.

To keep track of whether a memory slot is used or not, we need a **register**. This register will contain a (possibly boolean) value which will determine each state (1 = used, 0 = unused for example). ← Not the most efficient way, because such a list would take up 512MB (2³² / 2³ = 2²⁹bytes = 512MB)

A more efficient way would be to make a linked list of empty slots (doesn’t take into account used slots, so less memory used by the list).

To release the memories, you have to flip the values (for example, for every used space, the boolean value would be 1; flip it to 0 to free that slot)

Instead of having access to the physical memory, which is shared by all the processes (it is not safe), we’re going to use **logical** memory (a memory that doesn’t really exist). The system maps the virtual memory to the physical one at the end.

Logical Memory is a clone of the Physical Memory (thus, same size contained in both) ; between both types of memories, there is a correspondence table.

Eg of allocation for a **logical access scheme** / virtual memory

1. Find nb of contiguous slots in logical memory (cause it’s easier to manage)
2. Find nb of **non-contiguous** slots in physical memory (it should be written somewhere) & update the mapping table for each used slot

* A problem that can occur, if you try and only use contiguous slots, is that you will have holes in your physical memory which will remain unused (because they will be too small to contain some contiguous slots). That’s why you have to use non-contiguous slots.

3. Return the base address inside logical memory of step 1

**Mapping table :** links logical address to its physical counterpart

Mapping tables are unique to each process running in a multitasking system; this keeps each process from running into conflict with another when it needs to use a memory slot.

Vocab: segmentation fault

|  |  |  |  |
| --- | --- | --- | --- |
|  | logical |  | physical |
| ... | ... | ... | ... |
| 2 | 7 | 721 | 7 |
|  |  |  |  |
| 22 | x | 1025 | x |

|  |  |
| --- | --- |
| Logical @ | Physical @ |
| 2 | 721 |
| ... | ... |

“Logical address 2 is inside physical slot 721”

Eg :

Int main()

{

int \*p = 2;

p[ϕ] = 78; //it is going to look in the mapping table for this process, but it won’t find

anything = segmentation fault

}

If you’re looking for something that is near the end of the mapping table, it can be really **time-consuming.** Especially compared to the direct access method.

The max size of the mapping table is nbOfEntry \* 2 \* 8 bytes (4 bytes on 32b systems, 8 bytes on 64b systems)

*How to make it more time efficient?*

One to one slot mapping table with implicit key, so has the whole table

If you’re looking for the physical address, you can use the logical address as an index in the mapping table, and you’ll get it.

|  |  |
| --- | --- |
| Same slot as the logical one | Physical nb  slot |
| ... | ... |
| 2 | 721 |
| ... | ... |
| 22 | 1025 |

**Time** : still slower (2 times) but better, all accesses cost the same now

**Space** : 2⁶⁴ \* 8 bytes which is huge

The problem is that you’re using a big mapping table per process, which means you need lot of space for each process you’re using.

*How to make it space efficient?*

If we want to make this work we need to make the table really smaller, by limiting the number of entries.

One to one group of slots mapping table

* Split logical memory into a group of slots
* Split physical memory into blocks of n slots

Make a corresponding between logical blocks and physical blocks

The bigger the block in slot the smaller the mapping table will be

|  |  |
| --- | --- |
| Logical block | Physical address (list) |
|  |  |
|  |  |

1 - Find nb of contiguous slots in logical mem

2 - For every logical block used find a free physical block of the same size & update the mapping table

3 - return back the address of step 1

Block size: 32 => 2⁶⁴ / 32 = 2⁵⁹

Smaller but not enough. You have to use bigger blocks.

Vocab:

Logical access scheme = paging

Logical block is a **page**

Physical block is a **frame**

Using blocks is like subdividing things logically.

The paging table contains the corresponding addresses of the logical slot as they are represented in physical memory.

By this method, the **relative offset** of the corresponding slot in physical memory is the same as in logical memory.

Page : 4k slots = 2¹² slots

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | Logical mem |  | Paging table |  | Physical mem |
|  |  |  | 576 |  |  |
|  |  | 978 |  |
|  |  |  |  |

2¹² - 2 <=> 576\*2¹² + (2¹² - 2)

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 0 | 2¹²-2 | <-> | 576 | 2¹²-2 |
| Page nb | offset |  | Frame nb | offset |

Int main() {

int \*ptr = malloc(5); // 2¹² - 2

ptr[0] = 7; // 2¹² - 2

ptr[4] = 66; // 2¹² + 2

}

2¹² + 2 <-> 978\*2¹² + 2

(1, 2) <-> (978, 2)

**Time** : same as before

**Space** : size = 2⁶⁴ / 2¹² \* 8 bytes = 2⁵⁵ bytes => still huge

Even if you increase the size of the page, it won’t make it enough smaller.

On a 32 bits systems

size = 2^32 / 2^12 \* 4 bits = 4 MB

It uses this size per program.

The only way to make it smaller is by grouping blocks.

Physical memory : Int mem[2¹⁶]

void myAllocate(int nb)

void myWrite(int addr, int val) //writes into address the value in param

Int myRead(int addr)

void myFree(int addr, int nb)

Int main() {

Int addr = myAllocate(5);

myWrite(addr, 55); //p[0] = 55

myWrite(addr + 4, 77); //p[4] = 77

…

}

**PAGING (cont.)**

We are going to divide the previous blocks into bigger blocks containing a number of previous smaller blocks

If we divide the bigger level of blocks of a size of 2^21 bits

We now have a 1st level table : block table

From 0 to 2^11 - 1 (because 2^32 / 2^21)

And now each block from the block table has a page table still from 0 to 2^12 - 1

And the physical memory is still mapped with frames

On a 32 bit system :

Size : 2^11 \* 4 + 2^9 \* 4 -> 10KB

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| ( 0 | 0 | 2^12 - 1 ) | <-> | ( 721 | 2^12 - 1 ) |
| block | page | offset |  | frame | offset |

Example :

Process users [0, 500 K slots] u [ 2^26 - 1, 2^26 + 1 ]

3-level paging scheme ( 9, 6, 7, 10 ) as ( block, subblock, page, offset ) so addressing over 32 bits

Calculate the size of the mapping tables

**For [ 0, 500K ] :**

Block on 9 bits so 2^9 different blocks

Size of a block : 2^32 / 2^9 = 2^23

500K slots < 2^23

So there is only 1 block used.

This block is divided in subblocks

Subblock on 6 bits so 2^6 different subblocks

Size of a subblock : 2^23 / 2^6 = 2^17

500K slots in [ 0 , 2^19 ]

500K / 2^17 = 3,9 so last subblock used is 3

So there is 4 subblocks used

Each subblock is divided in pages

Page is on 7 bits so 2^7 different pages

Size of a page = 2^17 / 2^7 = 2^10

**For [ 2^26 - 1, 2^26 + 1 ] :**

2^26 - 1 / 2^23 = 7

2^26 + 1 / 2^23 = 8

So uses 2 blocks, each of the blocks uses 1 subblock, each subblock uses 1 frame

**Size :**

Size block table : 2^9 \* 4 Bytes = 2 KB

Size subblock table : 2^6 \* 4 Bytes = 256 Bytes

Size page table : 2^7 \* 4 Bytes = 512 Bytes

Total : 3 blocks + 3 subblocks + 6 frames

3 \* 2 KB + 3 \* 256 Bytes + 6 \* 512 bytes = 10 KB