Écrit en vert = notes de la séance de révisions avec Gaby (à noter : si une réponse en vert suit une réponse en bleu, c'est la réponse en vert qu'il faut prendre en compte, puisque c'est celle donnée par Gaby. Parfois, les notes en vert sont un complément de la réponse en bleu)
lien vers les slides du cours :

- Intro
- Process & Scheduling
- Memory
- Synchronisation
- FileSystems

(Swapping : prendre mémoire virtuelle, mettre sur ram, démapper, envoyer à la tâche)

Q: Kernel, Services, Libraries, Application: define the 4 terms, and their roles.

A:

- **Kernel**:
  - Core of the OS. It is a critical part of the computer, therefore no errors are allowed inside. It is hardware independant and handles basic resources management. Connects software application to hardware.
- **Services**:
  - Provided by the OS, for the convenience of its users. Background process which performs system "chores"
  - Includes program execution (load into memory, execute, …), I/O operations (read/write), File System management, error detection, resource allocation
- **Libraries**:
  - Collection of resources used by computer programs, allowing to use basic functionalities without re-writting it each time we need it. (libs/APIs)
- **Application**:
  - Program designed to perform a specific function directly for the user or for another program.

Q: What are the different states for a task in an OS ?

A:

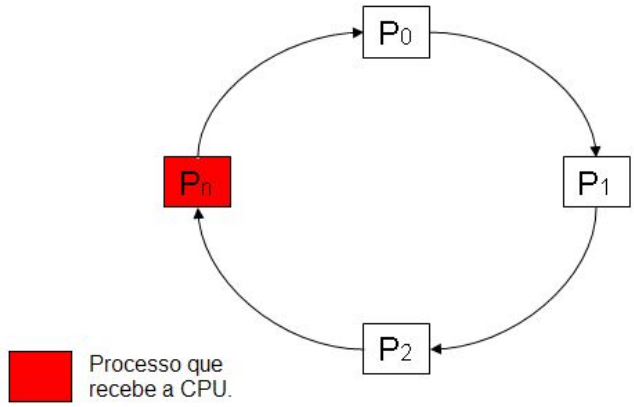
- Running
- Ready
- Blocked

Il faut expliquer le passage d'un état à un autre :
running -> **waiting/blocked :** syscall bloquant (read/write/wait)
ready -> **running** : quand le scheduler le décide (dépend donc du scheduler, cf question suivante)
running -> **ready** : le scheduler décide qu'un autre process doit tourner
waiting/blocked -> **ready** : blocking request finished

Q: Name and explain 3 scheduling algorithms

A:

À chaque fois, il faut décrire le passage des états d'une tâche

- **FCFS:** First come First served : nom à utiliser = **FIFO**
  - Fifo of ready process, first ready → first running until it stops, kernel doesn't interrupt.
  - Non préemptif (= ne peut pas stopper une tâche en cours, donc est moins réactif)
- **Round Robin:**
  - No priorities, processes are given a defined time to run (called quantum) in a circular order.
  - running → ready : quantum écoulé
  - ready → running : le process précédent a utilisé son quantum
  - Préemptif (peut stopper une tâche en cours, mais des problèmes peuvent survenir si le processus en cours d'exécution veut accéder à une ressource qu'un processus précédent n'a pas fini d'utiliser)
  - 

#spanish
- **Completely Fair scheduling:**
  - process are given the right of running accordingly to the time they have waited AND the priority they have. Processes are stored by "waiting time" in a red black tree, and higher priority processes have their time elapsing faster.
  - minimise le waiting time (le temps passé dans la queue de ready)
- **Ticket scheduler**

Q: What are signals in UNIX systems ? give 5 common signal names and explain them

A:

A signal is an asynchronous notification sent to a process or a specific thread in order to notify it of an event that occurred.

1. SIGKILL
   - sent to cause the **instant termination (kill)** of the process
2. SIGSEGV
   - sent when a process makes an invalid virtual memory reference/segmentation fault.
3. SIGSTP

    - request to **stop temporarily** the process (Ctrl-Z), sent by user input (=/= SIGSTOP: sent programatically, cannot be caught)
4. SIGINT
    - **interrupt** the process (Ctrl-C). Can be caught and/or ignored.
5. SIGTERM
    - request the **termination** of the process in a clean way (release resources/save state)

(**NB :** KILL = syscall qui envoie un signal)

Q: How can we execute custom behavior when receiving a signal ? Which signals can't have their default behavior overriden ?

A:
Some signals can be caught by a program; then, depending on which signal was caught, we can execute a custom behavior.
Signals which can't be overriden are:

- SIGKILL
- SIGSTOP

Q: Describe a mechanism for enforcing memory protection in order to prevent a program from modifying the memory associated with other programs.

A:
locks: When entering a critical part of a program, put a lock which forbid interruptions while going through the critical part. When the lock is released, normal scheduling can be applied again. **FAUX**

Virtualisation de la mémoire (et donc des pages) pour qu'une tâche se croit seule sur un espace mémoire continu et qu'elle n'interagisse pas avec d'autres tâches, sauf si l'OS le veut. (NB: en vrai, dans l'espace mémoire physique l'espace est fragmenté, c'est le principe des pages)
⇒ niveau processeur

Q: Some computer systems do not provide a privileged mode of operation in hardware. Is it possible to construct a secure operating system for these computer systems? Give arguments both that it is and that it is not possible.

A:
Not possibru: Hardware address protection is necessary to avoid having a program access any resource it wants, such as memory or I/O-related resources.

Possible: In an entirely software-oriented OS, you can restrict which programs are run, where they come from and how they are created. The compiler would have to check any memory access. This means that all code would have to be created on such a system, as imported code cannot be verified, and some tools forbidden (no assembler or external compilers).

Q: Which of the following instructions should be privileged?

a. Set value of a timer.
b. Read the clock.
c. Clear memory.
d. Issue a trap instruction.
e. Turn off interrupts.
f. Modify entries in device-status table.
g. Switch from user to kernel mode.
h. Access IO device.

A:

Set value of a timer, clear memory, turn off interrupts, modify entries in device-status table, access IO device
Everything else can be done while in user mode (g. is done by syscalls to read, open, write)

Q: What is the purpose of interrupts? What are the differences between a trap and an interrupt? Can traps be generated intentionally by a user program? If so, for what purpose?

A:

Interrupts: It's a kind of notification, thrown generally by hardware to the CPU, to inform it that some devices need attention (keyboard when a key is pressed, mouse, disk drives, …). The CPU will stop whatever it is doing to provide the service required by the device.

Interrupts are hardware interrupts, while traps are software-invoked interrupts.

Traps can be generated intentionally by a user program, they are a kind of Exception. They are used as a transfer of control: they usually invoke a kernel routine, because they run at a higher priority.

Q: What system calls have to be executed by a command interpreter or shell in order to start a new process?

A:

fork + execve

Q: What are the advantages and disadvantages of using the same system-call interface for manipulating both files and devices?

A:

Advantages:

- device can be accessed as a file in the file system
- Easy to add a new driver to deal with this particular file (which is a device)

Disadvantages:

- Can be difficult to capture the functionality of certain device with file access API
    - loss of functionality
    - loss of performance

Q: Describe the actions taken by a kernel to context-switch between processes.

A:

A context switch consists of saving the state of the current process and restoring the state of the process. The process state of the current process is changed to **ready** (as in, ready to be continued) and the next process' state is changed to **running** (to indicate is currently being executed). This action is pure overhead as no useful work is actually performed.

Q: It is said that fork(2) uses a copy on write mechanism. Explain fork semantics, and how it works in practice

A:

Fork is a syscall which creates a copy of the current process. It implies a separate address space for the child. For some performance reasons, fork uses copy on write mechanism; it means that even if a separate address space is created for the child in virtual memory, the physical memory remains the same until one of the processes writes on it. Then, a new part of physical memory will be mapped and copied from the original process.

Q: Give an example of a situation in which ordinary pipes are more suitable than named pipes and an example of a situation in which named pipes are more suitable that ordinary pipes.

A:

- Ordinary > named:
    - Simple communication works well with ordinary pipes. For example, assume we have a process that counts characters in a file. An ordinary pipe can be used where the producer writes the file to the pipe and the consumer reads the files and counts the number of characters in the file.
- Named > ordinary:
    - For an example where named pipes are more suitable, consider the situation where several processes may write messages to a log. When processes wish to write a message to the log, they write it to the named pipe. A server reads the messages from the named pipe and writes them to the log file.

Q: Describe the differences among short-term, medium-term and long-term scheduling.

A:

- long term: plan for tasks in the future
- short term: plan for next task based on dynamic informations
- middle term: based on current load, plan for actions

Q:Using the following program, identify the values of pid at line A, B, C, and D, assuming that the actual pids of the parent and child are 2600 and 2603.

```
#include <err.h>
#include <stdio.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <unistd.h>

int main()
{
	pid_t pid, pid1;

	pid = fork();

	if (pid < 0) {
		err(1, "fork failed");
	} else if (pid == 0) {
		pid1 = getpid();
		printf("pid = %u\n", pid); /* A */
		printf("pid1 = %u\n", pid1); /* B */
	} else {
		pid1 = getpid();
		printf("pid = %u\n", pid); /* C */
		printf("pid1 = %u\n", pid1); /* D */

		wait(0);
	}

	return 0;
}
```

A:

A: 0 (if condition)
B: 2603 (dans le child process, fork retourne 0, donc on print le pid du child)
C: 2603 (dans le process parent, fork retourne le pid du child process)
D: 2600

Q: Using the following program, explain what the output will be at line A.

```
#include <stdio.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <unistd.h>
```

```
int value = 5;

int main()
{
        pid_t pid = fork();

        if (pid == 0) {
                value += 15;
                return 0;
        } else if (pid > 0) {
                wait(0);
                printf("value = %d\n", value); /* A */
                return 0;
        }
}
```

A:

value = 5

Reason :

Fork uses Copy on write. This means that initially, all variables are shared, but as soon as a forked proccess tries to write on it, a copy is created for the process to write on. Basically, the child has its own copy of value, and modifying it in its process does not modify it in the parent process.

Q: Which of the following components of program state are shared across threads in a multithreaded process ?

a. Register values
b. Heap memory
c. Global variables
d. Stack memory

A:

b and c

Q: What are two differences between user-level threads and kernel-level threads? Under what circumstances is one type better than the other?

A:

User-level threads are threads that the kernel is not aware of. They are entirely within a process, and are scheduled to run within that process, while kernel is aware of kernel-level threads, and aren't linked to any process.

User-level threads are much faster to switch between since there is no context switch, while kernel-level threads are scheduled by the kernel, which allow it to have its own timeslice in the scheduling algorithm (since kernel-level threads aren't linked to a process).

Q: Can a multithreaded solution using multiple user-level threads achieve better performance on a multiprocessor system than on a single-processor system? Explain.

A:

A multithreaded system comprising of multiple user level threads cannot make use of the different processors in a multiprocessor system simultaneously. The operating system sees only a single process and will not schedule the different threads of the process on separate processors. Consequently, there is no performance benefit associated with executing multiple user-level threads on a multiprocessor system.

Q: Under what circumstances does a multithreaded solution using multiple kernel threads provide better performance than a single threaded solution on a single processor system?

A:

When a kernel thread suffers a page fault, another kernel thread can be switched in to use the interleaving time in a useful manner. A single-threaded process, on the other hand, will not be capable of performing useful work when a page fault takes place. Therefore, in scenarios where a program might suffer from frequent page faults or has to wait for other system events, a multithreaded solution would perform better even on a single-processor system.

Q: Describe the actions taken by a thread library to context-switch between user-level threads.

A:

Context switching between user threads is quite similar to switching between kernel threads, although it is dependent on the threads library and how it maps user threads to kernel threads. In general, context switching between user threads involves taking a user thread of its LWP and replacing it with another thread. This act typically involves saving and restoring the state of the registers

Q: Why is it important for the scheduler to distinguish IO-bound programs from CPU-bound programs?

A:

Because IO-bound programs can be blocked, waiting for exemple to receive an entry, while CPU-bound haven't such disadvantage.

IO-bound : Tâche qui va faire bcp de syscalls bloquants, donc elle va passer bcp de temps dans la chaine de ready. Ex : réseau, server http, navigateur ...

CPU-bound : Fait peu de syscalls bloquants et qui va faire des calculs et opérations. Ex : compilateur

Peut changer en fonction de ce que la tâche doit faire.

> make : passe son temps à exec et wait des programmes, donc IO-bound

C'est important de les distinguer car si une tâche io bound n'utilise pas tout son quantum de temps (intervalle de temps sur round robin par ex) il ne faut pas perdre de temps et lancer la prochaine tâche.

CFS (completely fair scheduling) : pour les tâches IO bound : minimiser le waiting time -> leur donner un plus petit quantum, mais beaucoup plus fréquemment. Tâche cpu bound : peut avoir besoin d'un + grand quantum, pour finir ses calculs

Q: Explain why interrupts are not appropriate for implementing synchronization primitives in multiprocessor systems.

A:

Interrupts are not sufficient in multiprocessor systems since disabling interrupts only prevents other processes from executing on the processor in which interrupts were disabled; there are no limitations on what processes could be executing on other processors and therefore the process disabling interrupts cannot guarantee mutually exclusive access to program state.

Q: Servers can be designed to limit the number of open connections. For example, a server may wish to have only N socket connections at any point in time. As soon as N connections are made, the server will not accept another incomming connection until an existing connection is released. Explain how semaphores can be used by a server to limit the number of concurrent connections.

A:

A semaphore is initialized to the number of allowable open socket connections. When a connection is accepted, the **acquire()** method is called, when a connection is released, the **release()** method is called. If the system reaches the number of allowable socket connections, subsequent calls to **acquire()** will block until an existing connection is terminated and the release method is invoked.

Q: Explain why implementing synchronization primitives by disabling interrupts is not appropriate in a single-processor system if the synchronization primitives are to be used in user-level programs.

A:

If a user-level program is given the ability to disable interrupts, then it can disable the timer interrupt and prevent context switching from taking place, thereby allowing it to use the processor without letting other processes to execute.

Q: What is a deadlock?

A:

It is a situation in which two or more competing actions are each waiting for the other to finish, and so, neither of them ever finish.

Exemple: two processes try to fill an array, process A modify row 1 then row 2, while process B modify row 2 then row 1. Afterr A modified 1 and B modified 2, A waits B to modify 1 while B waits A to modify 2 → deadlock.

on utilise lock pour éviter cette situation

Q: On a system with paging, a process cannot access memory that it does not own. Why? How could the operating system allow access to other memory? Why should it or should it not?

A:

An address on a paging system is a logical page number and an offset. The physical page is found by searching a table based on the logical page number to produce a physical page number. Because the operating system controls the contents of this table, it can limit a process to accessing only those physical pages allocated to the process. There is no way for a process to refer to a page it does not own because the page will not be in the page table. To allow such access, an operating system simply needs to allow entries for non-process memory to be added to the process' page table. This is useful when two or more processes need to exchange data they just read and write to the same physi-cal addresses (which may be at varying logical addresses). This makes for very efficient interprocess communication.

Q: Under what circumstances do page faults occur? Describe the actions taken by the operating system when a page fault occurs.

A:

A page fault occurs when an access to a page that has not been brought into main memory takes place. The operating system verifies the memory access, aborting the program if it is invalid. If it is valid, a free frame is located and I/O is requested to read the needed page into the free frame. Upon completion of I/O, the process table and page table are updated and the instruction is restarted.

2 raisons :

- Mémoire virtuelle non mappée dans mémoire physique
- Mauvais droits

2 @ :

- user : kernel essaye d'y accéder = SIGSEV à envoyer
- kernel : si kernel page fault sur @ kernel = il y a un bug, il faut fix

- erreur
- paging-on-demand
- swap (prendre mém virt, mettre sur ram, démapper, envoyer à task)
- fork : COW (copy on write, cf question suivante): écrire dans une zone read-only (mauvais droits) (on copie alors les zones dans la ram pour pouvoir écrire dessus)

Q: What is the copy-on-write feature, and under what circumstances is it beneficial to use this feature?

A:

Copy on Write allows processes to share pages rather than each having a separate copy of the pages. However, when one process tried to write to a shared page, then a trap is generated and the OS makes a separate copy of the page for each process.

This is commonly used in a fork() operation where the child is supposed to have a complete copy of the parent address space. Rather than create a separate copy, the OS allows the parent and child to share the parent's pages. However, since each is supposed to have its own private copy of the pages, the pages are copied when one of them attemps a write.

Q: Why do some systems keep track of the type of a file, while others leave it to the user and other simply do not implement multiple file types? Which system is "better"?

A:

Some systems allow different file operations based on the type of the file (for instance, an ascii file can be read as a stream while a database file can be read via an index to a block). Other systems leave such interpretation of a file's data to the process and provide no help in accessing the data. The method that is "better" depends on the needs of the processes on the system, and the demands the users place on the operating system. If a system runs mostly database applications, it may be more efficient for the operating system to implement a database-type file and provide operations, rather than making each program implement the same thing (possibly in different ways). For general-purpose systems it may be better to only implement basic file types to keep the operating system size smaller and allow maximum freedom to the processes on the system.

Q: If the operating system knew that a certain application was going to access file data in a sequential manner, how could it exploit this information to improve performance?

A:

When a block is accessed, the file system could prefetch the subsequent blocks in anticipation of future requests to these blocks. This prefetching optimization would reduce the waiting time experienced by the process for future requests.

Q: Consider a file system that uses inodes to represent files. Disk blocks are 4kb in size, and a pointer to a disk block requires 8bytes. This file system has 6 direct blocks, as well as

single, double and triple indirect disk blocks. What is the maximum size of a file that can be stored in this file system?

A:

(12 * 8 KB) + (2048 * 8 KB) + (2048 * 2048 * 8 KB) + (2048 * 2048 * 2048 * 8 KB) = 64 terabytes

k = puissance de 2, b = bytes (on parle de blocs, on va pas compter en bits…)
4kb = 4196 bytes
Lba = 8 bytes (pointeur vers un bloc)

Q: Why must the bit map for file allocation be kept on mass storage, rather than in main memory?

A:

In case of system crash (memory failure) the free-space list would not be lost as it would be if the bit map had been stored in main memory.

Q: What is ioctl(2)? What are the issues solved by this system call? What are the problems caused by it? Propose a mechanism to solve its problems.

A:

ioctl is a syscall which takes care of the I/O of various devices which can't be executed by a regular syscall. It takes a request code as parameter, and its effect completely depends on this request code. More often, the request code is device-specific. Problem? Solution?

Q: Describe how you could obtain a statistical profile of the amount of time spent by a program executing different sections of its code. Discuss the importance of obtaining such a statistical profile.

A:

One could issue periodic timer interrupts and monitor what instructions or what sections of code are currently executing when the interrupts are delivered. A statistical profile of which pieces of code were active should be consistent with the time spent by the program in different sections of its code. Once such a statistical profile has been obtained, the programmer could optimize those sections of code that are consuming more of the CPU resources.