Architecture et Systèmes

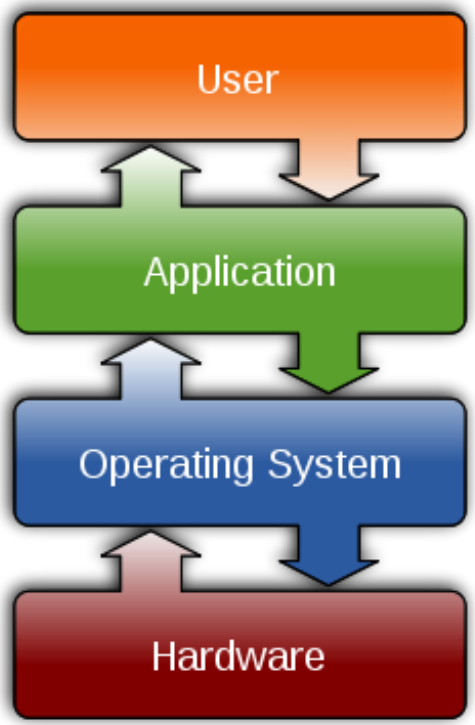
Stefan Schwoon

Cours L3, 2014/15, ENS Cachan

What is an operating system?

Characterisation according to Tanenbaum:

> *Software consists of two categories: the system programs, which enable the operation of the computer, and the application programs, which resolve the user's problems. The operating system is the most important of the system programs. It controls the resources of a computer and provides a basis for the application programs.*



Thus, the operating system intervenes between the hardware and the actual applications that the user sees.

Tasks of an operating system

Enable good things:

Provide functionality to applications, making them easier to write.

Abstract from hardware details, enable same software to run on different computers.

Provide useful abstractions: files, processes, windows, . . .

Smooth handling of concurrency; every application may run as if it was alone.

Tasks of an operating system

Prevent bad things:

Keep different users from interfering with one another, maliciously or inadvertently.

Ditto for different processes of the same user.

Provide for a fair distribution of resources.

Do not allow malicious processes (e.g., viruses) to corrupt the system.

Organisation of the operating system

Within an operating system, we can distinguish the following components:

The kernel: set of low-level *system calls*; always resides in memory; critical code allowed to access 'real' mode.

The library functions providing a more user-friendly interface to the kernel.

Low-level applications like the shell etc.

System and library functions

A programmer often does not interact with system calls directly, but through the libraries.

Example: `brk` (system call that sets boundary of heap);
`malloc` (library call to allocate some part of the heap)

Another example: `write` (system call) / `printf` (library function)

Calls to library functions and system calls can be observed with the shell commands `ltrace` and `strace`.

Low-level applications

Set of programs that allow the user to do anything useful at all with the system (without actually resorting to programming).

Examples: shell and associated programs (`ls`, `cat`, `...`)

In today's computers: graphical-user interface

Line between OS programs and application programs not always precise.

Domains of an OS

Processes, Threads, Signals, Scheduling

File system, Networking, I/O in general

Memory management

User management

(graphical user interface)

...

Unix and Posix

Unix was the name of a first vendor-independent OS created in 1970 by Kernighan and Ritchie.

The same authors developed the C language, notably as a base for programming in Unix.

Incompatibilities between Unix and various clones lead to the creation of the Posix standard in the 1980s.

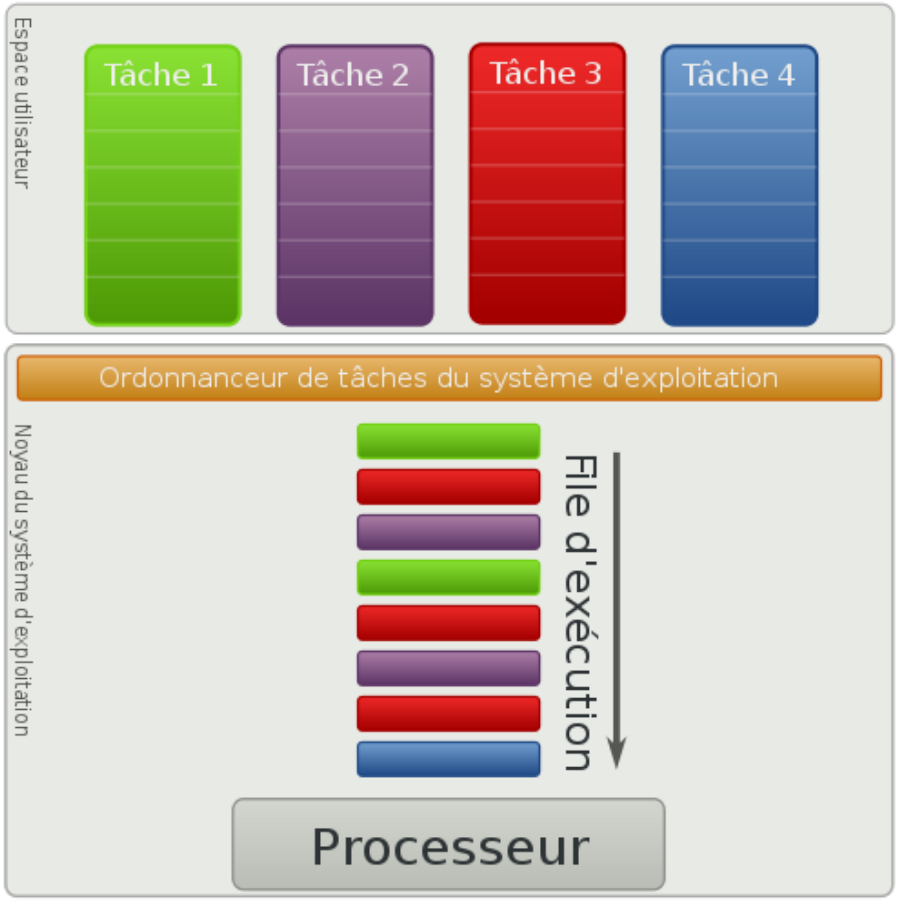
Posix defines a large set of system/library calls and their behaviours; also (largely) adhered to by MacOS / Windows.

This course: based on Posix

Processes

CPU: (traditionally) sequential machine, treats one instruction after another

Process: data structure within the OS representing a task or “unit of execution”; OS assures fair time-sharing and non-interference



What is a process?

Processes can be seen as a data structure representing a sequential “unit of execution” in a multi-tasking system.

A process (or: task) is a sequential activity pursued by the computer; it executes some code, has its own data and other attributes.

Not to be confused with program. Analogy with cooking: If a program is a recipe, the act of cooking is the analogue of a process.

Examples: the “init” process, service programs (daemons), shell, programs invoked by the shell, editor, . . .

Each process has a numerical identifier (pid). A list of current processes can be viewed with shell commands like `ps` or `top`.

Why processes?

Facilitates programming: each program can be written in sequential style, without taking care of other processes (unless interaction is explicitly desired), operating system takes care of the concurrency-related aspects.

Processes are isolated from one another: cannot spy on/modify other processes, failure in one process won't harm the others.

Processes can interact only in carefully defined ways.

Multitasking on a sequential CPU

On a (single-core) only one process can execute at the same time.

The OS puts each process into its own “virtual machine” and switches between active processes, giving each one small timeslot at a time.

User perspective: illusion of concurrency
Process perspective: sequential execution

We will look at the precise mechanics of scheduling later in the course.

The Unix command-line shell

Provides a rich, text-based interface for managing processes.

Easy access to other systems programs that implement OS functionalities, in particular operations on files.

User types commands at the prompt.

Shell executes the command (typically launching one or more processes) and waits for its completion before accepting the next command.

Simple shell commands

Standard form of a shell command:

```
program_name  arg1  arg2  ...
```

Spaces separate the program name and the arguments.

Example: `echo Bonjour`

Command: `echo` (simply outputs its arguments)

Example: `cat foo.txt`

Command: `cat` (shows contents of file)

Example: `ll`

Command: `ls` (`ll` is an alias for `ls -l`)

If an argument itself contains a space, enclose it in quotes:

```
cat "my file"
```

What happens?

The shell will start looking for a program with the given name. For this, it will use the PATH variable.

If the program is found in on of the directories named by PATH, a new process is created.

This process then executes the program and passes the parameters to it.
In C:

```
int main (int argc, char **argv)
```

Process hierarchy and creation

Each process may create children by issuing the `fork` system call.

> `fork` duplicates the calling (“parent”) process, creating a copy called the “child”.
>
> If successful, parent receives the process id of the child, the child receives 0. In case of failure, returns -1.
>
> Details: `man fork`
>
> The child may obtain the process id of its parent by calling `getppid`, and any process may obtain its own id using `getpid`.

In Unix, processes are organized as a tree, according to their creation, with the special “init” process as the root (command: `pstree`).

fork and memory

After the call to `fork` we have two processes that are identical, except for two things:

- their process identifier (pid)
- the value that `fork` just returned

All memory contents will be identical at the beginning.

However, parent and child now have separate memories, any changes in one process will not affect the other.

Attributes of a process

Process id (numeric, PID), parent id

Context (program counter, CPU registers)

Memory: code, data, stack

State (active, waiting, blocked, . . .)

many others, we will see some. . .

Life and death of a process

A process may *terminate* by calling `exit` or `return`ing from the main function.

In doing so, it may return a value called the exit code of the process.

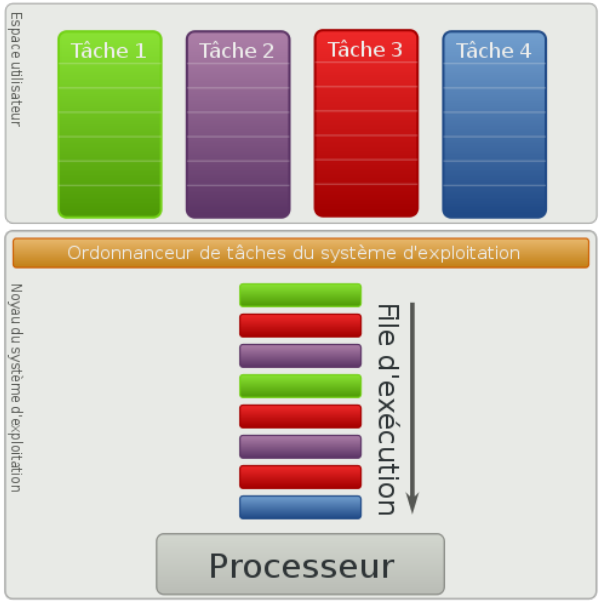
A parent process may `wait` for one of its children to terminate. (This is what the shell does before it accepts another command from the user.)

`wait` also returns information about the child (whether it terminated normally, its exit code, etc).

Notably, the parent can obtain the exit code of the child from the return value of `wait` by using the `WEXITSTATUS` macro.

Process states

Scheduling means that processes alternate between two basic states:



Active: the process is currently running, i.e. it has been selected by the scheduler.

Suspended: the process is not currently executing (another process is).

Looking closer at the suspended states, we find that some processes cannot continue working for logical reasons,

e.g., it is waiting for input; or for a child;

or it has been put to `sleep`, etc

To avoid uselessly scheduling such processes, the suspended processes are further subdivided:

Waiting: the process is willing to execute but is currently suspended by the scheduler.

Blocked: the process is waiting for something, will not be scheduled until it is unblocked.

The scheduler only considers the waiting processes.

Special process states

Unix knows some other process states, for example:

Created: entry created in the process table, but process is still being set up.

Zombie: process has finished execution but not been purged from memory. Will disappear when the parent calls `wait`.

Note (1): when a program terminates, its children are attached to another process (usually the init process, but depends on OS). The shell uses this opportunity to clean up zombies.

Note (2): Unix has some other special states for processes, these are only the most important ones.

Exec

The `exec` family of functions allows a process to replace itself with another program. The system will then load the specified program into memory and start executing it under the id of the process that called the function.

Example (launching a command in the shell):

The user types a command at the prompt.

The shell first `fork`s.

The child calls, e.g., `execvp` to launch the desired program.

The parent waits for the child to terminate, then returns to the prompt.

Combining multiple commands in the shell

`cmd1 ; cmd2`: execute first cmd1, then cmd2

shell launches one child, waits for its termination, then launches another child

`cmd1 || cmd2`: execute cmd1, and if return code non-null, execute cmd2

`cmd1 && cmd2`: execute cmd1, and if return code null, execute cmd2

same principle, but query the code returned by `wait` before eventually launching the second child