# Process

A process is an instance of a program just like an object is an instance of a class.

An OS exists solely to serve processes; without processes there would be no need for an OS.

The OS is responsible for:

- Process creation (at the instigation of some other process).
- Allocation of resources to a process.
- Sharing of resources (process management).
- Process termination.
- Inter–process communication (to some extent).

# History

The earliest computers had no notion of processes. The “system” would be controlled by a simplistic operating system which would allow only one program to execute.

The executing program could use all the computer resources, although painful experience quickly placed limits to what a user program could do.

The single user model led to very poor utilisation of computer resources. No application needs all the the resources of a computer system. Even if one came up with an application that uses all the resources, most of them would be used only some of the time and idle the rest of the time.

In the single–user model most resources were sitting idle most of the time. Considering the outlandish prices of computer equipment in those days, it became clear that the only economically practical way is to **share** resources.

## Time sharing

The concept originated in the tourism industry; it has almost exactly the same meaning as in Operating Systems.

Take a resource (hotel room, CPU) and a period of time. Divide the period of time into $n$ slices and give one slice to every customer.

In tourism, the period of time (1 year) is divided into 52 weeks; each timeshare owner buys one week (a slice).

In OS, the time period is not fixed (because of a variable pool of clients) but the slice is: some arbitrary small time interval such as 200 ms (the duration of a slice used to be important but is not anymore).

Timesharing works for resources that cannot be divided; otherwise the simpler method of dividing the resource into fragments is more practical (for main memory, disk space, access to concerts, etc.).

## Improvements

If one process per computer did not work, something had to be changed. Two different solutions were used:

- Multiply the machines: create several machines out of one computer and give each machine to a process. the machines so created do not exist physically, so they are virtual machines.
- Multiply the processes: allow more than one process in execution concurrently.

## Virtual machines

The basic idea is as follows: the real OS carves out of the computer system a subset of its resources, time–shares the other resources, calls the whole thing a virtual machine and gives it to user process.

It does not work, because the user process needs some OS to rely upon; hence the VM was given not to a user process but to a guest OS which was then used by the user to run the user process.

This approach has tremendous appeal to the expert: it allows to do operating system programming in a safe way and permits the co–existence of many different operating systems on the same machine at the same time.

## Moreover

If I have a real OS (let it be called CP) that creates virtual machines, I can put another copy of CP on one of the virtual machines and let that copy create more virtual machines, etc. Now I can create any number of computers out of one real computer.

Another option is to combine virtual machines with the other solution (many processes per system).

Yet another application: I want students to write a program that executes in kernel mode. It is impractical to each student a real computer to play with (note that these computers cannot be shared by more than one student); but I could give each a virtual machine (perhaps even many virtual machines).

## Multiplying processes

Today's computer are based on the concept of multiprogramming, when several processes are in execution "at the same time" (or rather seem to be). If a system has only one CPU, it can execute only one instruction at a time and thus serve at most one process at a time, so it is clear that multiprogramming is an illusion, a virtual reality.

On a single–processor machine several devices operate in parallel with the CPU (this is true parallelism) but no two processes can use the same CPU in parallel. Processes share the CPU (and many other resources) by getting access to them for only part of the time; this is called concurrent, as opposed to parallel, access.

# Concurrent vs. parallel

In the human world there is no need to distinguish between the terms concurrent and parallel (or its variant, simultaneous).

The distinction is of great importance in any virtual reality simulated within a real world.

Consider the execution of machine instructions by two processes:

**Parallel** execution takes place when each process uses a different CPU; hence both can execute their code simultaneously (at the same instant of universal time). the other.

**Concurrent** execution takes place when the lifespans of the two processes overlap, i.e. there is a moment in time when both processes are in progress (each having already executed some, but not all, of its code).

## Concurrent vs. parallel

Concurrent is weaker then parallel because everything that is parallel is also concurrent but not the other way around.

Consider two courses taught at a sane university in the Winter 2009 term. They are taught concurrently (from January to April). Are they taught in parallel?

**Yes** if they both are taught MWF 11:30–12:20. Clearly that implies they are taught in different classrooms (sane university).

**No** if one is taught MWF 11:30–12:20 and the other MWF 1:30–2:20. Note that they use the same classroom but being active at different times prevents them from being parallel.

## Sharing

The OS allows several processes to execute concurrently. It makes the OS responsible for letting these processes share peacefully the computer resources. The resources are:

- CPU.
- Main memory.
- File system.
- Network(s).
- Peripherals, including two special ones: standard input (keyboard?) and standard output (monitor?).


The CPU can be shared in only one way: a process is given the right to use the CPU for a preset amount of time; the CPU is then taken away from that process and given to the next process. The action of taking the CPU away is performed by the OS as a result of a timer interrupt.

It is obvious that manipulating the timers must be a privileged activity or an process may hijack the CPU forever.

# Main memory

Originally, memory was simply divided into blocks called partitions, each block given to a different process.

Many improvements were made until virtual memory solved the problem:

- Every process receives the maximum possible address space (on traditional hardware, $2^{32}$ bytes). This address space does not exist anywhere as an entity. It is divides into small blocks called pages; pages have a fixed size (typically 2048 bytes with 1024, 4096 also common).
- The virtual memory is simulated by real memory which is divided into blocks of the same size as pages (called frames).
- All the pages reside on a secondary storage device (disk) if they exist at all. Copies of them are kept in main memory when needed.
- When a process needs a page that is not in main memory, a frame is found to hold the page.

This is a very simplified description; a more adequate

presentation will follow in due time.

## File systems

This is totally effortless–all that needed to be done was to add the concept of file ownership and access rights.

## Networks

Fortunately, networking software was introduced much later than other OS code. Right from the start it was well–structured (divided into the famed 7 layers). A process, whether on a single–process or a multi–process machine, interacts only with the top levels of networking software (TCP/UDP) which give the impression that the process is the one and only process around.

## So, what is a process?

The definition (a program in execution) means nothing to an OS. Inside the kernel, a process is a data structure called Process Control Block pointed to by a Process IDentifier. A PCB is unique in the system and it represents the identity of the process.

There are two ways to store PCBs, giving way to two sets of terms:

- A system–wide **Process Table** is made of Process Table Entries each of them containing one PCB.
- Each process has a block of memory called the u area associated with it. The u area contains the PCB and other system information describing the process and its environment.

Both are functionally identical.

## The u area

This area contains all the information about the context of a process. It is made of:

**Identity:** pid, ppid, gid.

**Context switch information:** registers, PSW, pointer to segment (or page) table.

**Scheduling information:** priority, usage statistics (CPU, memory faults, etc.)

**File system information:** open files, current directory, etc.

**Signal information:** vector of pending signals, vector of signal handler addresses, signal mask.

**Context information:** shared segments, semaphores, etc.

**Per process kernel stack:** sometimes the kernel executes in the context of a process (i.e. does not perform a full context switch before proceeding). Then it uses this area to accept interrupts.

## Moral

The kernel thinks that a process is a PCB.

Note that varies drastically from what a process really is because in reality a process is a collection of memory areas including the code of the process, etc.

## Process creation

A new process is created by an existing process through a system call:

- In **Windows** a single system call CreateProcess creates a new PCB and starts the process with code taken from a file. The child never shares any memory (code or data) with the parent.
- In **Unix** process creation is divided into two separate steps: creation of a PCB (fork()) and loading the child's code from a file (exec()) which is not a required part of process creation.

## Process creation in Unix

The system call exec does not create a new process; it replaces the code of an existing process with code taken from an executable file (traditional OSs call this loading).

Consider the code:

```
printf( "A...\n" );
execve( "/misc/starsandstripes", ... );
printf( "B...\n" );
```

This code will display A and then start playing some strange music. B will never be displayed.

# Process creation in Unix

Unix has one system call to create a process: fork() (it exists in two versions, the other being vfork()).

When invoked, fork() makes an exact copy of u area of the current process. This gives the new process a new identity but retains all the other attributes of the original process ("parent").

One aspect of fork that is somewhat tricky is its handling of the address space: the space is not copied, only the segment/page table is. Both the original and the copy of the table has all the entries marked Copy–on–Write which will be discussed in due time (Memory Management).

The alternative form vfork does not make a copy of the segment table and puts in the new PCB a pointer to the parent's table (this way both the parent and the child share the same physical memory). Unix expects the child to replace immediately its memory image using exec and guarantees bad results if that is not done.

## Process hierarchy

Every process, except process 0, is created by the fork() system call.

When the system is booted, a simple process image is "manually" created and is given the pid of 0.

This process contains two instructions: a fork() followed by an exec().

The fork creates process 1 (**init**) which becomes the ancestor of all other processes. The exec that follows changes the code of 0 into the **swapper** (see lyrics for a musical explanation).

init goes through all the list of all devices of type tty and forks a *dæmon* process for each of them called getty.

## Shells

A terminal *dæmon* waits for someone to log in. when that happens, getty forks a process that immediately execs the file given as default shell in the /etc/passwd file starting an interactive interpreter that accepts input form the keyboard (this input is in a special programming language, such as bash) or a GUI.

## Process termination

Whether a process terminated willingly or not, chances are good that the process did not release all the resources it held.

These resources are released by the OS with the use of the u area. some resources cannot be released automatically (e.g. file locks) and may eventually create deadlocks.