# CHAPTER V: PROCESS CONTROL IN UNIX

B. Boufama

University of Windsor

# Unix Processes

An executing program → a *process*.

In particular, every process in Unix has the followings :

- A unique process ID (PID)
- Some code : instructions that are being executed
- Some data : variables
- A stack : a form of memory where it is possible to push and pop.
- An environment : registers' contents, tables of open files,...

Unix starts as a single process, called *init*. The PID of *init* is 1.
The only way to create a new process in Unix, is to duplicate an existing one.
→ the process *init* is the ancestor of all subsequent processes.
In particular, process *init* never dies.

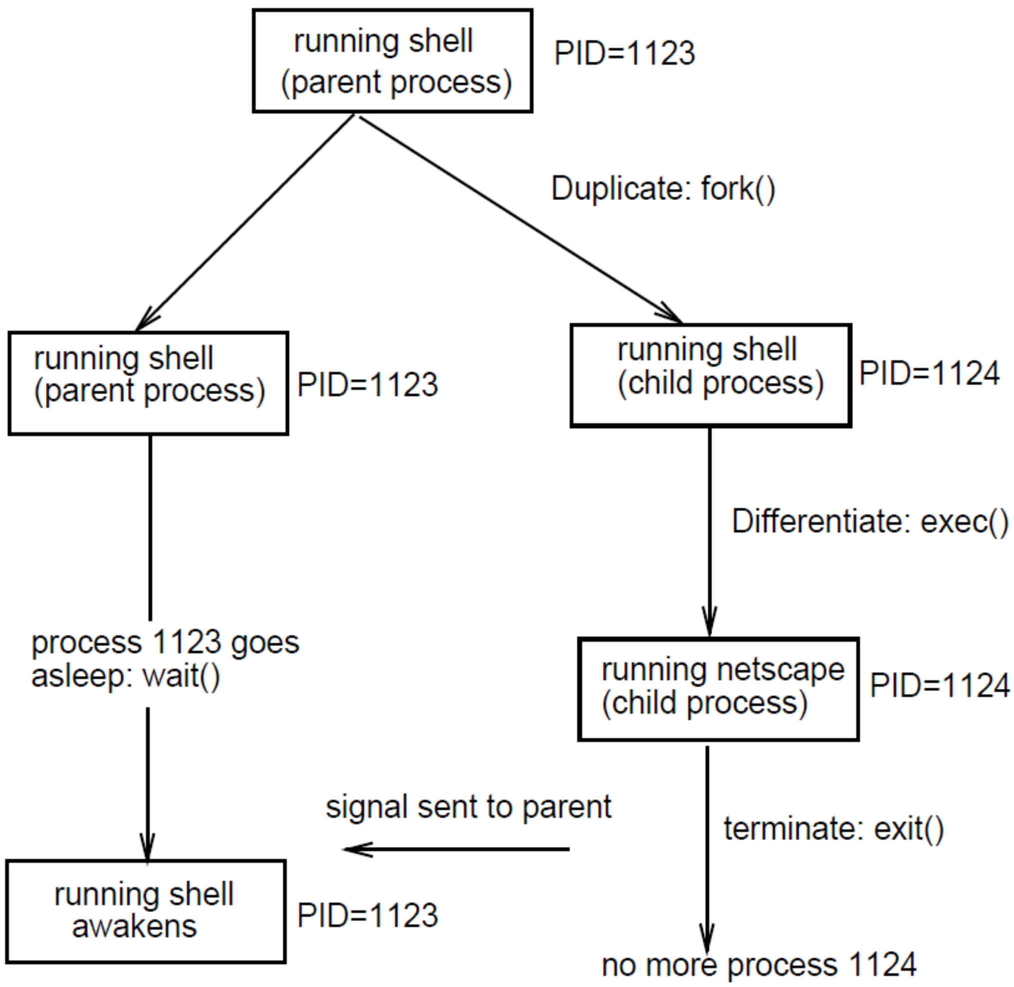
The creation or spawning of new processes is done with two system calls:

- *fork()*: duplicates the caller process
- *exec()*: replaces the caller process by a new one.

Example : Running a utility from a shell(*csh*).

The following steps are necessary

- *csh fork*s a copy of itself.
- the child process *exec*s the utility program, for example, *netscape*.
- the parent process waits for the termination(*exit*) of its child process by going asleep.
- when the child process terminates, a signal is sent to the parent process(the shell program *csh*). The latter wakes-up and becomes ready to accept the next command.


running shell
(parent process)
PID=1123
Duplicate: fork()
running shell
(parent process)
PID=1123
running shell
(child process)
PID=1124
Differentiate: exec()
process 1123 goes
asleep: wait()
running netscape
(child process)
PID=1124
signal sent to parent
terminate: exit()
running shell
awakens
PID=1123
no more process 1124

# Creating a new Process: fork()

Synopsis: **pid_t fork(void);**

when succesful, the *fork()* system call :

- creates a copy of the caller (parent) process.
- returns the *PID* of the newly created process to the parent
- returns 0 to the new process (the child).

If not succesful, the *fork()* returns -1.

*fork()* is a strange system call : called by a single process but returns twice, to two different processes.

In particular, a child process has :

- its own unique *PID*,
- a different *PPID*,
- its own copy of the parent's data segment and file descriptors

*fork()* is primarily used in two situations :

1. A process wants to execute another program (Shells).
2. A process has a main task and when necessary creat a child to handle an operation(Servers).

Here is a simple example :

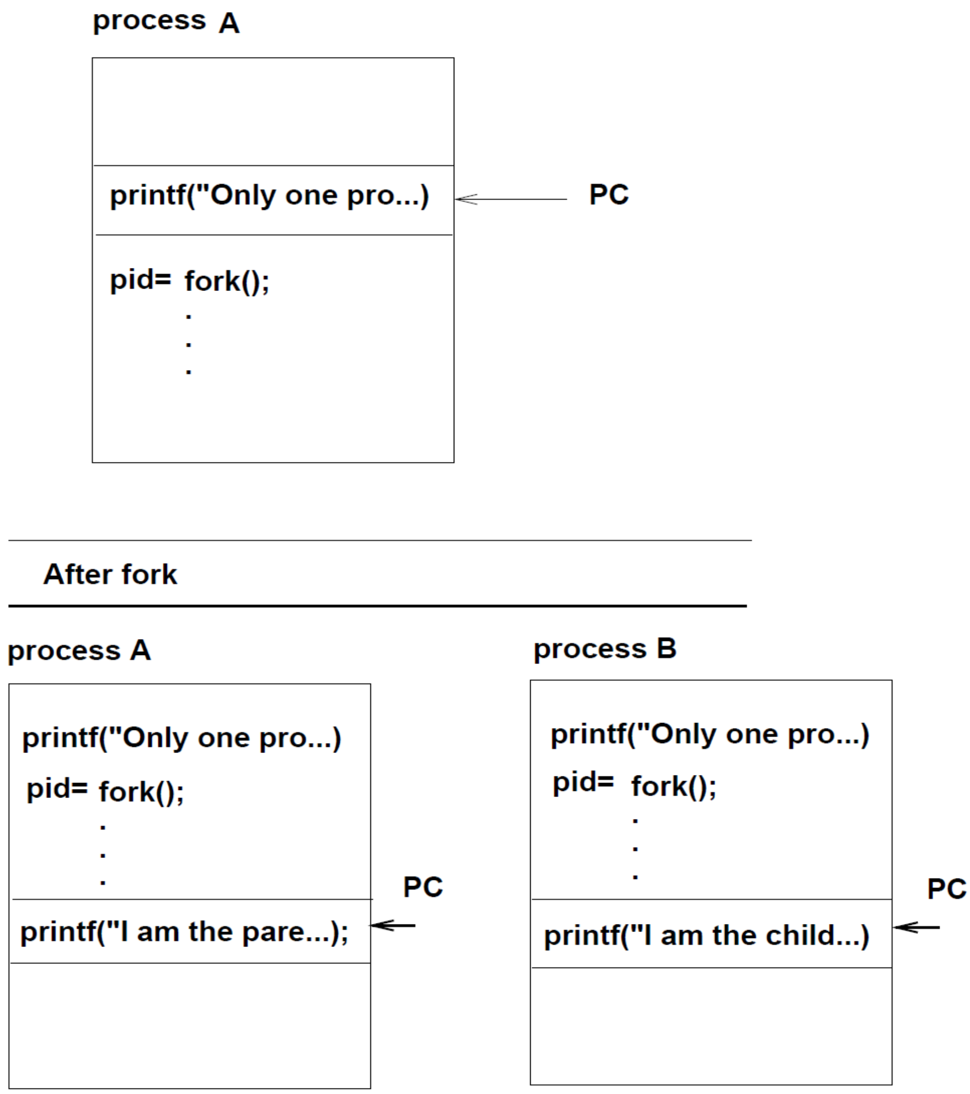
```
#include <unistd.h>

int main(int argc, char *argv[]){
  int pid;

  printf("Only one process\n");
  pid = fork();

  if(pid == -1){
    perror("impossible to fork");
    exit(1);
  }
  if(pid > 0)
    printf("I am the parent, pid=%d\n", getpid());
  else
    if(pid == 0)
      printf("I am the child, pid=%d\n", getpid());

  exit(0);
}
```


process A
printf("Only one pro...)
PC
pid= fork();
After fork
process A
printf("Only one pro...)
pid= fork();
PC
printf("I am the pare...);
process B
printf("Only one pro...)
pid= fork();
PC
printf("I am the child...)

# Terminating a process: exit()

Synopsis: **void exit(int status);**
This call terminates a process and never returns
The *status* value is available to the parent process through the *wait()* system call.

When invoked by a process, the *exit()* system call :

- closes all the process's file descriptors
- frees the memory used by its code, data and stack
- sends a **SIGCHLD** signal to its parent and waits for the parent to accept its return code.

## Waiting for a process: wait()

Synopsis: **pid_t wait(int *status);**

This call allows a parent process to wait for one of its children to terminate and to accept its child's termination code.

When called, *wait()* can

- block (suspend) the caller process, if all of its children are still running, or
- return immediately with a termination status of a child, if a child has terminated and is waiting for its termination to be accepted, or
- return immediately with an error(-1) if it does not have any child process.

# Waiting for a specific process: waitpid()

Synopsis: pid of a specific child not all

**pid_t waitpid(pid_t pid, int *status, int options);**

This call allows a parent process to wait for a specific child to terminate and to accept its child's termination code.

One interesting option is **WNOHANG** that causes the **waitpid()** to return immediately even if no child has exited.

**Note: wait(&status)** is equivalent to
**waitpid(-1, &status, 0)**
any child

When succesful, *wait()* returns the *pid* of the terminating child process.

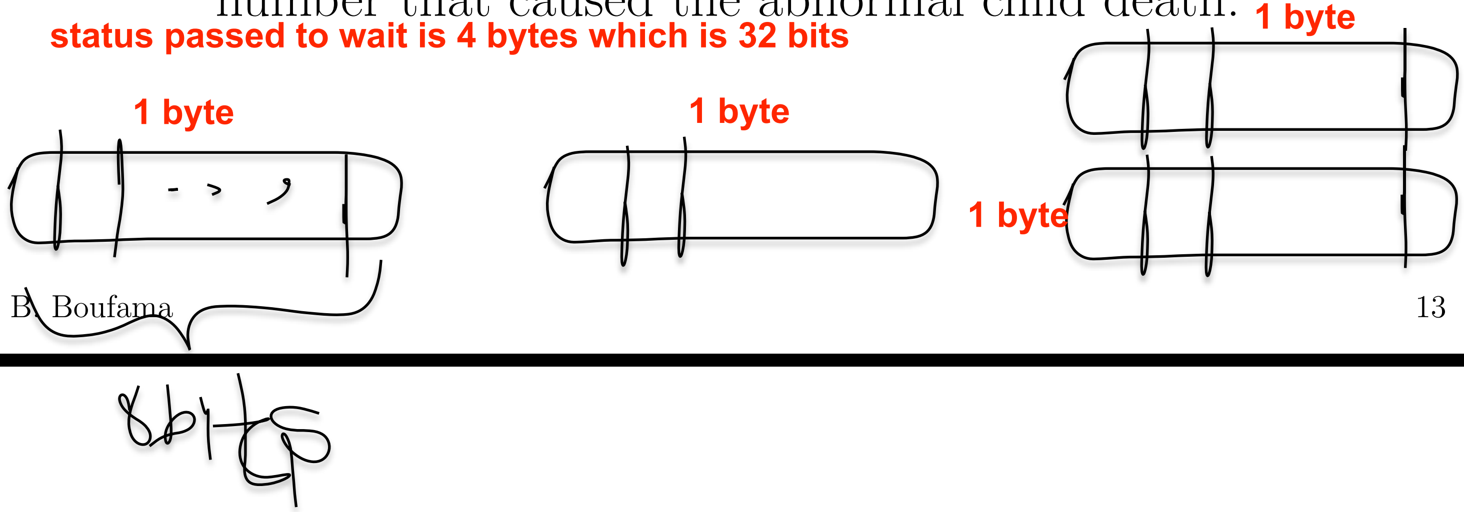
The value in *status* is encoded as follow:

- if the rightmost byte of *status* is zero, then the leftmost byte contains the status returned by the child: a value between 0 and 255.
  This represents a normal termination of the child process.
- if the rightmost byte of *status* is nonzero, then the rightmost 7 bits are equal to the *signal number*, that caused the process to terminate. The remaining bit of the rightmost byte is set to 1 if a core dump was produced by the child process.

Some bit-manipulation macros have been defined to deal with the value in the variable *status*(you need to include $< sys/wait.h >$).

- *WIFEXITED(status)*: true for normal child termination.
- *WEXITSTATUS(status)*: used only when WIFEXITED(status) is true, it returns the exit status as an integer(0-255).
- *WIFSIGNALED(status)*: true for abnormal child termination
- *WTERMSIG(status)*: used only when WIFSIGNALED(status) is true, it returns the signal number that caused the abnormal child death.

**status passed to wait is 4 bytes which is 32 bits**



# Orphan and zombie Processes

A process that terminates does not leave the system before its parent accepts its return.

There are 2 interesting situations, when :

1. a parent exits(for example, the parent has been killed prematurely) while its children are still alive.
   → the children become *orphans*.

   Because somebody must accept their return codes, the kerned simply changes their *PPID* to 1.

   → *orphan* processes are systematically adopted by the process *init* (PID of init is 1).
   In particular, *init* accepts all its children returns.

2. a live parent never makes the system call *wait()*.
   the children become *zombies* and remain in the system's process table waiting for the acceptance of their return. However, they loose their ressources (data, code, stack...).

   Because the system's process table has a fixed-size, too many zombie processes can require the intervebtion of the system administrator.

   Example :
   Below is a C program, called zombie.c, to create a zombie.

```
int main(int argc, char *argv[]){
 int pid;

 pid = fork();
 if (pid){  // means pid !=0
   printf("parent process, pid=%d\n", getpid());
   while(1)
    sleep(5);
 }
 printf("child process, pid=%d\n", getpid());
 exit(0);
}
```

1- run in background the program:> ./zombie &
On the screen you will get:
I am the child, pid=12357
I am the parent, pid=12356
2- look for the processes:> ps -ef | grep 12356
On the screen you will get:
boufama 12357 12356 0 0:00 $< defunct >$
boufama 12356 20142 0 20:01:44 pts/7 0:00 ./zombie

## Differentiating a process: exec()

The *exec()* family of system calls allowsa process to replace its current code, data and stack with those of another program.

**Synopsis :**

- int execl(const char *path, [const char *$arg_i$,]$^+$ NULL)
- int execlp(const char *path, [const char *$arg_i$,]$^+$ NULL)
- int execv(const char *path, const char *argv[])
- int execvp(const char *path, const char *argv[])

where $i = 0, \ldots, n$ and $^+$ means one or more times.
The difference between these 4 system calls has to do with syntax.
*execl()* and *execv()* require the whole pathname of the executable program to be supplied.
*execlp()* and *execvp()* use the variable $**PATH** to find the program.

In particular :

- A successful call to *exec()* never returns.
- *exec()* returns -1 if not successful.
- For both *execl()* and *execlp()* $arg_0$ must be the name of the program.
- For both *execv()* and *execvp()* $arg[0]$ must be the name of the program.

# Changing directories: chdir()

A child process inherents its current working directory from its parent.
Each process can change its working directory using *chdir()*.
**Synopsis :**
**int chdir(const char * pathName);**

*chdir()* returns 0 if successful -1 otherwise.
It fails if the specified path name does not exist or if the process does not have execute permission from the directory.