A **process** consists of :

- an executing (running) program
- its current values
- state information
- the resources used by the operating system to manage the execution

`ps` displays a list of processes executed in current shell

```
% ps
PID   TTY STAT TIME COMMAND
14429  p4 S    0:00 -bash
14431  p4 R    0:00 ps
%
```

terminal status time execut. command execution

`ps -l` shows full information (long format):

```
FLAGS   UID   PID  PPID PRI  NI   SIZE   RSS WCHAN       STA TTY TIME  COMMAND
   100  1002   379   377   0   0   2020   684 c0192be3   S   p0  0:01  -bash
   100  1002  3589  3588   0   0   1924   836 c0192be3   S   p2  0:00  -bash
   100  1002 14429 14427  10   0   1908  1224 c0118060   S   p4  0:00  -bash
100000  1002 14611 14429  11   0    904   516 0          R   p4  0:00  ps -l
%
```

owner; parent process PID; priority; size of text+data+stack; size in mem.; event for which the process is sleeping; status; terminal; exec. time; execution command

`ps -ax` information about all processes running currently in the system (a – show processes of other users too, x – show processes without controlling terminal)

Every process can create other process :

- the initiating process is called ***parent***
- a newly created process is called ***child***.

Processes create a hierarchical structure, the root is **init** process (id = 1)

`pstree`

`top`

```
kill [-name_or_signal_number] process_identifier
```

By convention, if the signal is not specified, a process with a given PID is terminated with `SIGTERM` signal (signal number 15)

```
kill 14285
killall vi
```

kill –l shows names of all signals defined in the system

striking ^C key from terminal keyboard - the active shell will send immediately the `SIGINT` signal to all active child processes.

Not all processes can be stopped this way. Some special processes (as shell process) can be killed only by the `SIGKILL` signal (signal number 9)

```
% kill -9 14280
% kill -KILL 14280
```

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

…

}
```

```
cc prog1.c     → ./a.out
cc prog1.c –o prog1 →./prog1
```

1 Creating a process

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

pid_t fork();
```

RETURNS: success :

- process identifier of the child process (a value greater than 0) to the parent process
- value of 0 to the child process

error: -1

The `fork` system call creates a child process - is called once but returns twice
The return value allows the process to determine if it is the parent or the child.

Example 1 Creating a new process

```
void main() {
   printf("Start\n");
   if (fork()==0)
      printf("Hello from child\n");
   else
      printf("Hello from parent\n");
   printf("Stop\n");
}
```

A process terminates either by calling `exit` (normal termination) or due to receiving an uncaught signal

```
#include <unistd.h>

void exit(int status);
```

The argument status is an exit code, which indicates the reason of termination, and can be obtained by the parent process.

`exit(0)` – process ends correctly
`exit(value<>0)` failure occured

```
main(){
    fork()
    fork()
    if (fork()==0)
      fork();
    fork();
    }
```

```
main(){
   fork()
   fork()
   if (fork()==0)
     exit(0);
   fork();
   }
```

```
main(){
  if (fork()!=0){
   if (fork()==0);
     fork();
   }
   else
     exit(0);
   fork();
   }
```

Processes are concurrent

Each process is identified by a unique value:PID

```
#include <unistd.h>

int getpid();
int getppid();
```

```
RETURNS: success : PID or PPID.
         error: -1
```

Waiting for a process to terminate

To wait for an immediate child to terminate, a `wait` system call is used.

```
#include <sys/wait.h>

int wait(int *statusp)
```

```
RETURNS:  success  :  PID  of  child  process  which
                      stopped or died
          error: -1
```

The location pointed to by `statusp`. It indicates the cause of termination and other information about the terminated process in the following manner:

- If the child process terminated normally, the low-order byte will be 0 and the high-order byte will contain the exit code passed by the child as the argument to the `exit` system call.

- If the child process terminated due to an uncaught signal, the low-order byte will contain the signal number, and the high-order byte will be 0.

```
sleep (time)
```

Example 3 wait and correct ending of the process

```
#include <stdio.h>

main(){
int pid1, pid2, status;

pid1 = fork();
if (pid1 == 0)
  exit(7);

printf("Child id : %d\n", pid1);
pid2 = wait(&status);
 printf("Process ending status %d: %x\n", pid2, status);
}
```

Example 4 wait and killing the process

```
#include <stdio.h>
main(){
int pid1, pid2, status;


pid1 = fork();
if (pid1 == 0){
sleep(10);
exit(7);
}

printf("Child id : %d\n", pid1);
kill(pid1, 9);
pid2 = wait(&status);
printf("Ending status %d: %x\n", pid2, status);
}
```

Starting a new code

To start the execution of a new code an `exec` system call is used.

The system call replaces the current process image (i.e. the code, data and stack segments) with a new one contained in the file the location of which is passed as the first argument. It does not influence other parameters of the process e.g. PID, parent PID, open files table.

There is no return from a successful `exec call` because the calling process image is overlaid by the new process image

```
#include <unistd.h>

int execl(const char *path, const char *arg0, ...,
          const char *argn, char * /*NULL*/);

int execv(const char *path, char *const argv[]);

int execle(const char *path,char *const arg0[], ... ,
          const char *argn, char * /*NULL*/, char *const
          envp[]);

int execve(const char *path, char *const argv[],
          char *const envp[]);

int execlp(const char *file, const char *arg0, ...,
          const char *argn, char * /*NULL*/);

int execvp (const char *file, char *const argv[]);
```

```
RETURNS: success : execution of program passed as
                   the argument
         error: -1
```

`path` a pointer to the path name of the file containing the program to be executed.

`arg0`, ..., `argn` list of arguments available to the new program.

By convention, `arg0` points to a string that is the same as `path` (or the last component of `path`). The list of argument strings is terminated by a `(char*)0` argument.

`argv` an array of character pointers to null-terminated strings that constitute the argument list available to the new process image.

By convention, `argv` must have at least one member, and it should point to a string that is the same as `path` (or its last component). `argv` is terminated by a null pointer.

This form of `exec` system call is useful when the list of arguments is known at the time of writing the program.
The `execv` version is useful when the number of arguments is not known in advance.
The `execle` and `execve` system calls allow passing an environment to a process

```
execl(„/bin/ls”, „ls”, „-l”,null)
execlp(„ls”, „ls”, „-l”,null)

char* const av[]={„ls”, „-l”, null}
execv(„/bin/ls”, av)

char* const av[]={„ls”, „-l”, null}
execvp(„ls”, av)
```

Example:

```
#include <stdio.h>

main(){
    printf("Poczatek\n");
    execlp("ls", "ls", "-a", NULL);
    printf("Koniec\n");
}
```

Excersices:

1. Write a program which prints the list of files from the current directory. The file list should be preceded by the word „*Beginning*”, and ended with the word „*End*”

2. Write a program that creates three zombie processes