Process Termination

1- How is the process terminated?
2- What happens after termination?

The process is terminated in one of the following scenarios:

- ↳ executes all its instructions (return) } normal termination
- ↳ exits based on a given situation } normal termination
- ↳ killed by another priviliged process } abnormal termination

Normal Termination

```
<sys/wait.h>
int exit (int status);
```

→ return ≡ exit(0).

status → used to indicate wether the program executed successfully or if there were any errors

Status: a number (1 byte) (0 → 255) used by the process to tell its parent about its exit(status).

N.B: each process running in the system has a PCB. when the process exits, it will stay on the system until his parent gets the info sent by the exit (status)

How the parent gets info about the death of the child?

↳ <sys/wait.h>

```
int wait(int *st_ptr);
pid = wait(&str_ptr);
```

Wait Statement:

the wait statement is a blocking statement, this means that the parent will wait for the child to exit( ).

little demo:

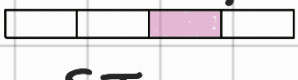
parent	child
I_1	I_1
I_2	I_2
I_3 pid=wait(&st)	⋮
I_4	I_{100}
⋮	I_{101} exit(status)

↳ the wait function returns the pid of the exited child.

↳ the wait function is used for the first child that exits.

↳ if the parent has no children, wait returns -1

* while(wait(&str_pt)!=-1);
  this means that the parent will wait for all the children to exit.
* the parameter of the wait function is a pointer to int used for receiving the value sent by the child using the function exit(status).
* wait(null);
  ignores the return value sent by the child

Note: the returned status is put in the second byte of the integer  so the returned number is stored in ST as a different int, to solve this problem, we use:

wait(&st)

ST = ST << 8 or ST = WEXITSTATUS(ST)

(these are used to shift 8 bits to the right).

exit & wait: using exit and wait, parent and its child can make some naïve communication _ some kind of synchronization

Example:

array of integers (0 and 255)

int tab[10] = {12, 5, 3, 25, 16, 70, 112, 200, 240, 254}

* determine the max of this vector

parent	child
0 → 4	5 → 9

display the max on screen.

```
* #include <unistd.h>
  #include <stdlib.h>
  #include <sys/wait.h>

  int main(){
  int tab[10]={12,5,3,25,16,70,112,200,240,254}, j;
  if (pid=fork()){
     int max =0;
     for(int i=0; i<5; i++){
        if(A[i]>max)
           max= A[i];
     } wait (&j);    // j is max1.
      j = j << 8;
     if (max>j)
        printf("%d", max);
     else
        printf("%d", j);
  } else{
     int max1=0;
     for(int i=5; i<10; i++){
        if (A[i]>max1)
           max1=A[i];} exit(max1);
  }
```

int sleep(int nsec)

the process calling this function suspends its execution for nsec.

Zombie Process:

- a child process that has completed its execution but still has an entry in the process table (process table entry contains process ID, exit status, etc).
- this occurs when the parent process hasn't yet collected the exit status of its child process. The operating system retains the zombie process entry in the process table until the 'wait' system call or similar mech.
- once the parent process collects the exit status of the child process to acknowledge their termination.

Orphan Process:

- a child process whose parent process has terminated or been killed before the child process has completed its execution.
- orphan processes are typically adopted by the init process (process with pid=1). the init process becomes the new parent of the orphaned child process.
- orphan processes continue their execution, and when they finish, they become zombie processes until the init process collects their exit status.

↳ orphan processes are usually the result of a parent process creating a child process and then exiting unexpectedly before the child has completed its tasks.

Ex:

```
void main() {
  int pid;
  printf("I'm the main process with pid=%d\n",
          get pid());
  pid= fork();
  if (!pid){
    printf("I'm the child process with pid=
            %d\n", get pid());
    exit(2);
  }
  else {
    sleep(100);
    wait(null);
  }
}
```

Output

```
child : pid=100
parent: pid=90
```