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).

↳ 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**?

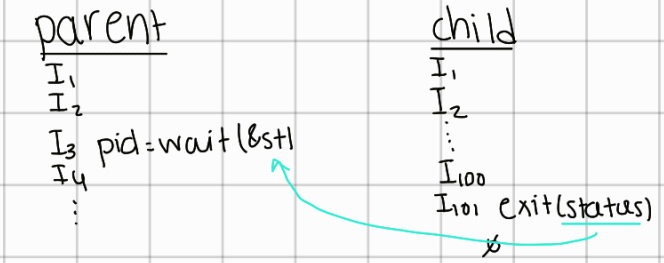
↳ <sys/wait.h>

```
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:

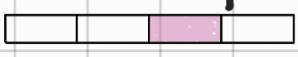


↳ the wait function returns the pid of the exited child.

↳ 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:
ST = ST << 8 or **ST = WEXITSTATUS(ST)** ©
(these are used to shift 8 bits to the right).

wait(&st)

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: 0 → 4 ; child: 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 uexpectedly before the child has completed its tasks.

Ex:

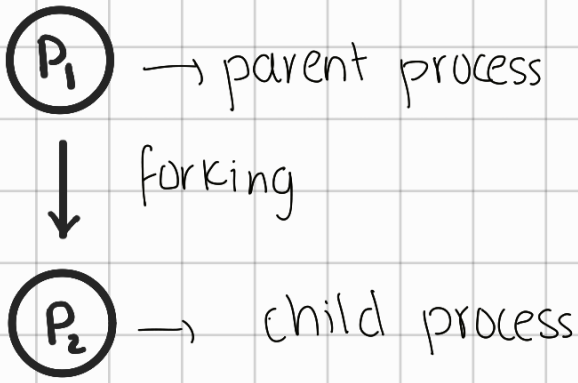
```
void main() {
   int pid;
   printf("I'm the main process with pid=%d\n",
          getpid());
   pid= fork();
   if (!pid){
      printf("I'm the child process with pid=
             %d\n", getpid());
      exit(2);
   }
   else {
      sleep(100);
      wait(null);
   }
}
```

Output

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

lecture 3: Process Creation

How the process is created? Unix.



the **main process** in the unix is **Init → pid=1**

- Initially when the **system boots**, a process **(init) with pid = 1 is started.**

Je n'appelle Jiled

- In unix, the **process are created by cloning/ duplication** => one process duplicated

P_1 becomes parent.
P_2 child

- int fork();

```
#include <unistd.h>
int fork();
```

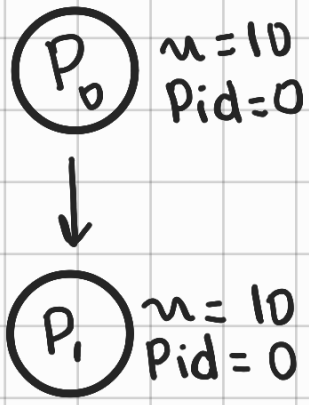

```
int fork()
```

- -1 → failure
- 0 → (child)
- <> 0 → (parent) pid of child.

Ex:

```
void main(){
  int x, pid;
  x = 10;
  if (pid = fork())
    x = x+5;
  printf("x = %d /n", x);
}
```

parent



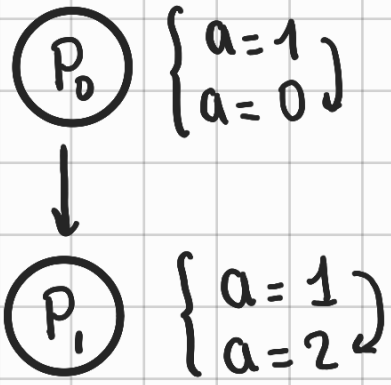

```
if (Pid = fork())
  ① pid = fork();
  ② if (pid) = if(pid!=0)
```

```
void main(){
  int a = 1;
  if(!fork()){
    a++;
    printf("%d\n", a);
  } else
    a++;
  printf("%d\n", a);
}
```

child.

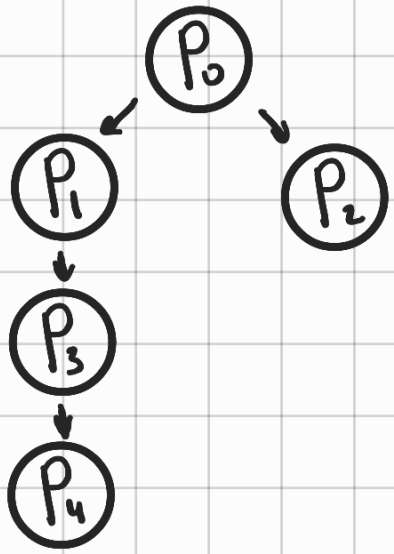
output:		
2	0	2
0	2	2
2	2	0

```
if (!fork())
  → pid = fork()
  → if (pid == 0)
```

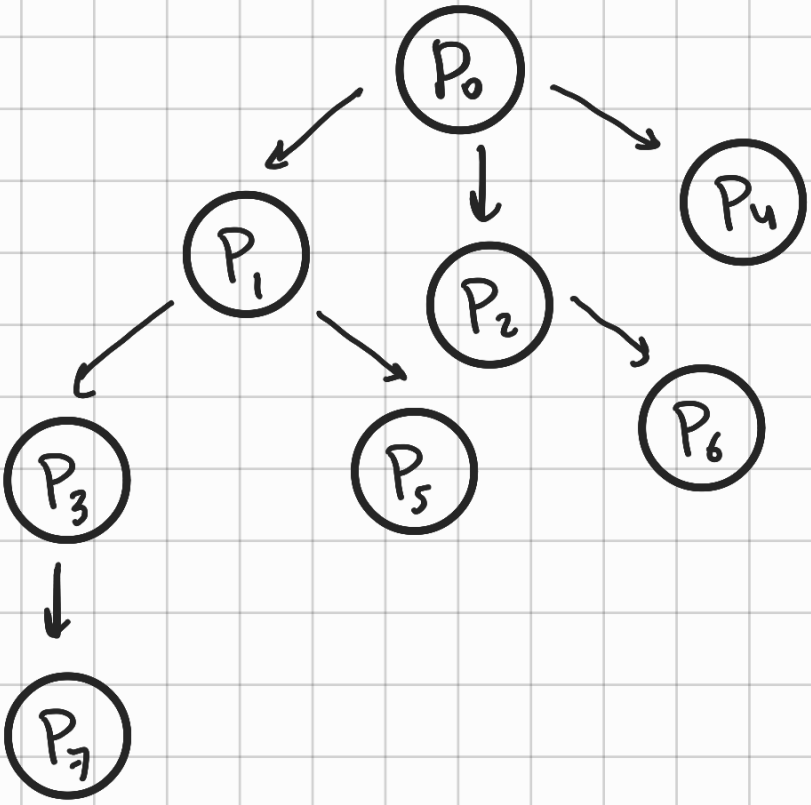


Ex:
```
void main(){
    if(fork())
        fork();
    else if (!fork())
        fork();
}
```

if (fork() => Pid = fork() => P_1
if (Pid <> 0) => P_0.



Ex:
```
void main(){
    fork();
    fork();
    fork();
}
```



Process Identification

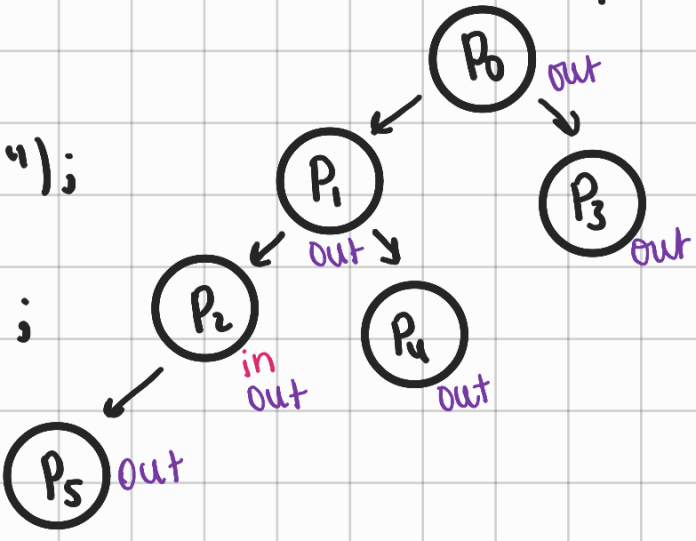
- int getpid()
  returns the pid of the calling process.

- int get ppid()
  returns the pid of the parent of the calling proces.

Extra Exercises:

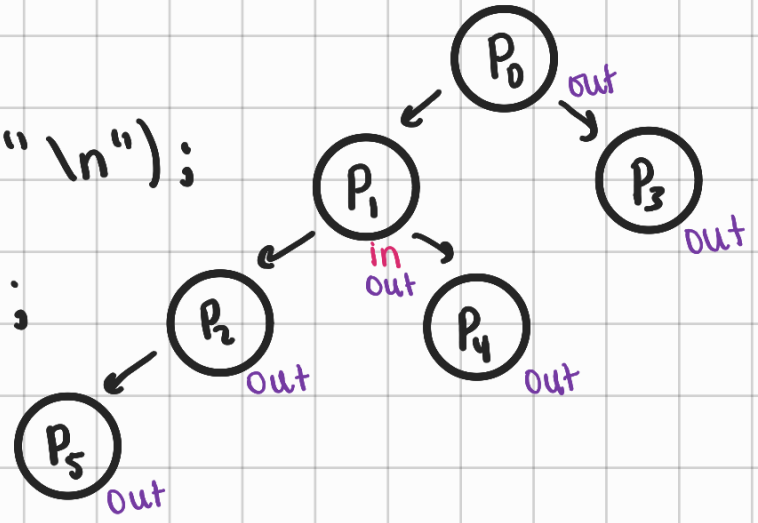
Ex:
```
void main() {
    if (!fork())
        if (!fork())
            printf("I'm in \n");
    fork();
    printf("I'm out \n");
}
```

draw tree + output



-
```
void main(){
    if (! fork())
        if (fork())
            printf ("I'm in" \n");
    fork();
    printf("I'm out \n");
}
```

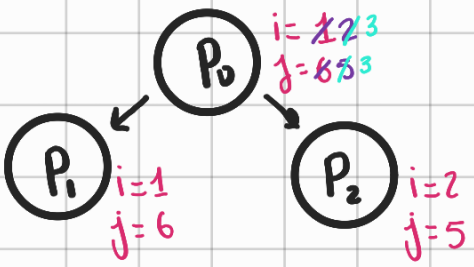
draw tree + output



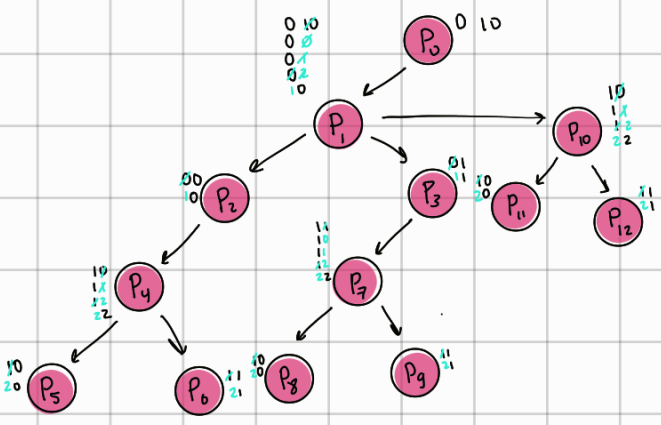
N.B: Break

-
```
void main(){
    int i,j=6;
    for(i=1; i<j; i++){
        if (!fork())
            break;
        else
            j=-i;
    }
    printf("i=%d, j=%d\n", i, j);
}
```

tree of process + output



Ex:
```
void main(){
    int i=10, j=10;
    for(i=0, i<2, i++){
        if (fork()) · if parent break
            break;
        for(j=0, j<2, j++){
            if (!fork()) if child break
                break;
        }
    }
    printf("%d, %d\n", i, j);
}
```



output: 0:0 10 4:22 8:20 12:21

1:12 5:20 9:21

2:10 6:21 10:22

3:11 7:22 11:20

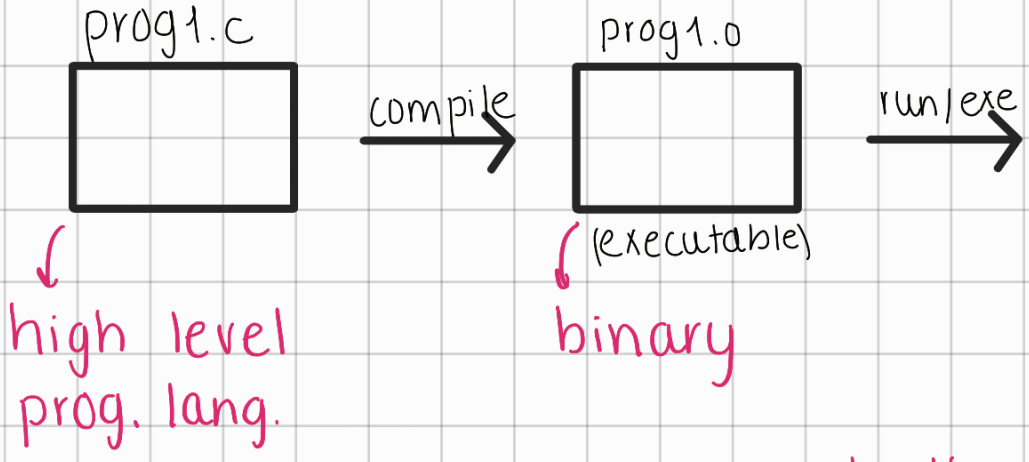
Lecture 2- Process Management.

Resource management
↳ process management
↳ memory management
↳ file system management
↳ I/O devices management
↳ security and protection

} OS responsible for managing these resources.

I- What is a process?

- the process is a **program** in execution.
↳ **program**: set of instructions in machine language



terminal: `>./prog1` → execute the program ↓ process.

Example
program: steps of making a cake.
process: CPU is the baker, he uses the ingredients (data), and follows the recipe (instructions) to make the cake

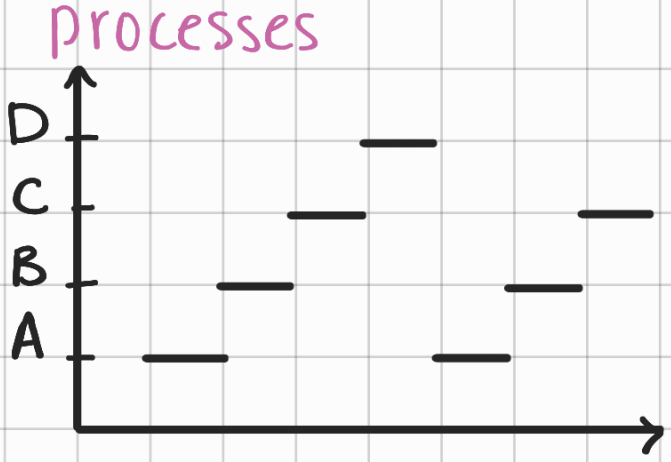
Actual Process Simplified

- When a **program** is run in a computer, its **instructions** are loaded into the **RAM**.
- The **CPU** **fetches** these **instructions** from **memory**, one at a time, **decodes** them, and then **executes** the **instructions**. (process)

Program → static
Process → dynamic

Multi programming mode (time-sharing model)

- execute several processes at the same time
- only one **CPU** → shared resource



(the time is very little, hence unnoticeable)

↳ it seems that 4 independent processes are happening in parallel (**pseudo-parallelism**)

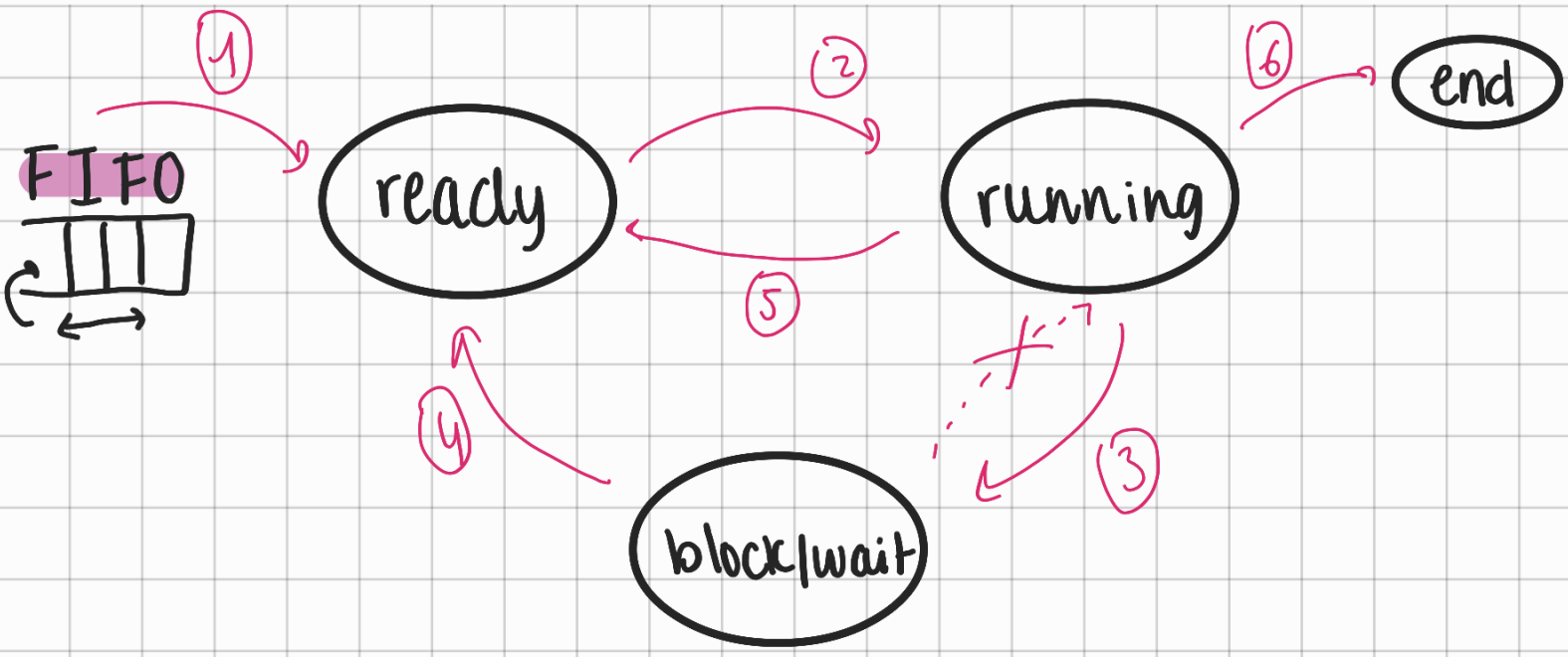
- each process has a programming instruction, input, output, state, set of registers.
- a single processor (CPU) may be shared among several processes
- some scheduling algorithim is used to determin when to terminate a process and serve anotherone

Each process has several attributes:

1- process id → pid (unique identifier)
2- state → to describe
3- priority
4- general purpose registers
   ↳ PC (program counter)
   ↳ IR for decode (instruction register)
   ↳ accumulator
5- list of opened files
6- list of opened devices (I/O devices)
7- protection

State of the process

- refers to the condition or status of a running program / process.
- each process can be in one of several states:
  ↳ **running**: the process is currently being executed by the CPU
  ↳ **ready**: the process was loaded into memory and is prepared to run but is waiting for its turn to execute on the CPU.
  ↳ **blocked / waiting**: the process is in a state of waiting, often because it's waiting for some event, such as user input from an I/O device.
  ↳ **terminated / exit**: the process has completed its execution and has been terminated. It may be waiting for the operating system to clean up resources (freeing memory, closing open files...) or for the parent process to collect exit status (determine out come of the execution



1 the process is created
2 _ the process gets to the CPU
_ the dispatcher elects the process to execute
3 the process makes I/O request so its blocked till the availability of the request
4 I/O request are now available
5 the process is not ended but its quantum time (slice of time) is expired or another process with higher priority arrives to ready queue (context switching)
6 the process is ended

FIFO: "First input, first out put" refers to a data processing principle in which the earliest data or request received is the first to be processed or served.

Types of process

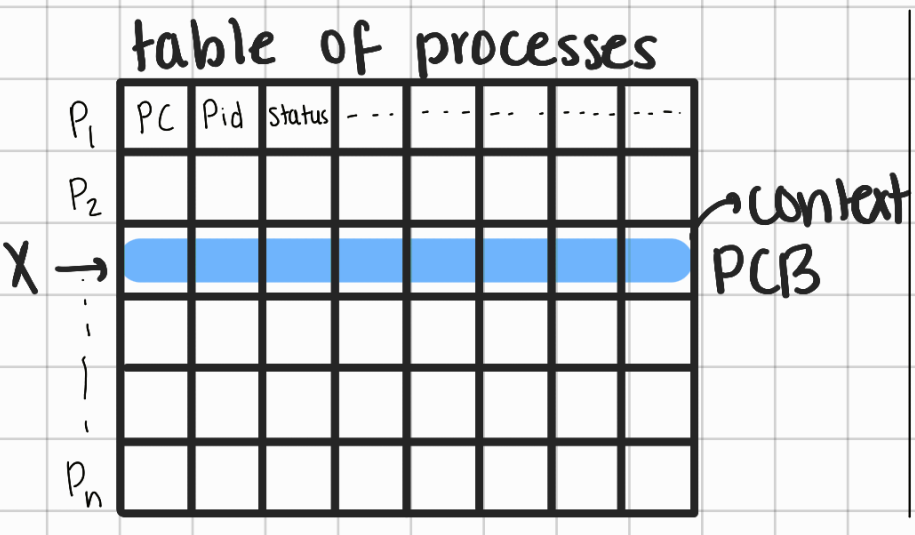
↳ CPU bound → it needs long time CPU (mathematical simulations)

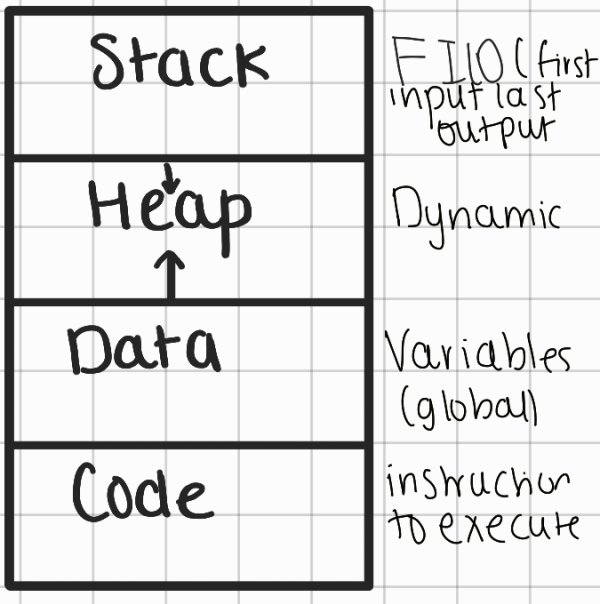
↳ I/O bound → it needs long time I/O (reading files)

Process Creation

PCB: Process Control Block, contains the attributes of each process (pid, state, priority, ...)

in simpler terms: PCB is like a file that the computer uses to keep track of everything it's doing for each running program, such as what it's working on, where it's saved, and how much time it has used so far.





- When a process is created, it is assigned a process control block (PCB), which is stored in the the table of processes located in the operating system's memory. The PCB contains essential info. about the process, including its process identifier (PID), status, priority, and pointers to the memory segments associated with this process, (Stack, Heap, Data, Code)

How Process is Created

1- In UNIX, processes are created using a mechanism called "fork"

2- A process (referred to as the "parent") can create a new process (the "child") by duplicating itself. This duplication includes copying the current state of the parent process such as its code, data, and file descriptors.

3- When you start a UNIX-based system, the first process created has a process identifier (PID) of 1 and is known as "init". Init is responsible for initializing the system and starting other processes.

- The "fork" function is a system call provided by the "unistd.h" library in UNIX. When you call "fork()" in a program, it creates a new child process that is essentially a copy of the current (parent) process. The child process starts executing from the same point in the code as the parent process.