Processes

Real-Time Operating Systems Programming (RTOSP)
Master in Critical Computing Systems Engineering (MCCSE)

2022/23

Paulo Baltarejo Sousa and Cláudio Maia
{pbs,crr}@isep.ipp.pt

Disclaimer

Material and Slides

Some of the material/slides are adapted from various:

- Presentations found on the internet;
- Books;
- Web sites;
- ...

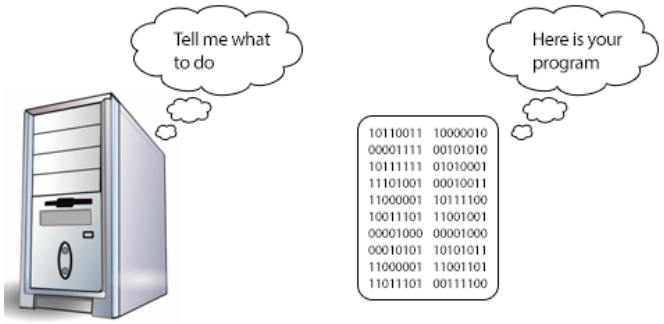
Outline

1. Processes
2. Application Programming Interface

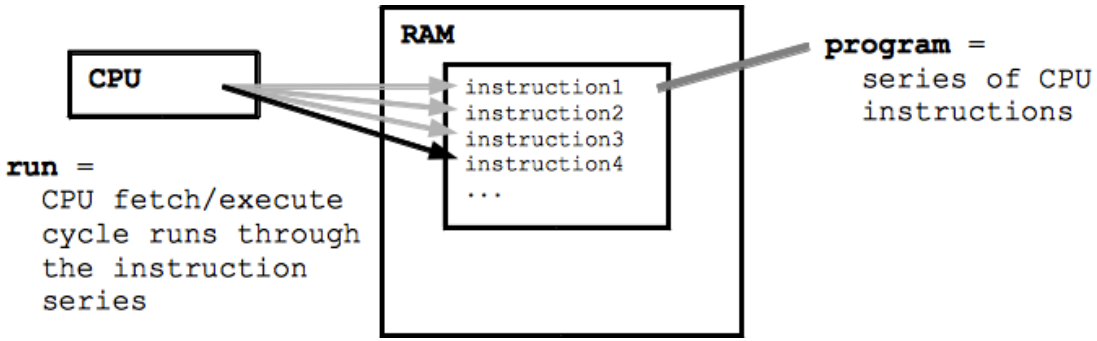
Processes

Computer Application (I)

- A computer **program is a collection of instructions** that performs a specific task when executed by a computer.



- The program is loaded into Main memory (RAM).
- The CPU must look into memory and fetch in the instructions and data and act upon them according to what the instructions are.



What is a Process?

- An instance of a program in execution is called a **Process**.
  - **Process is the execution of a program** that performs the actions specified in that program.
- The OS helps you to create, schedule, and terminates the processes which is used by CPU.
- A process created by the a process is called a **child** process, so, the creator is the **parent** process.

Process Status Linux command (I)

- `ps` command is used to list the currently running processes.
- `> ps`

```
pbs@hn3:~$ ps
    PID TTY          TIME CMD
  49414 pts/0    00:00:00 bash
  50618 pts/0    00:00:00 ps
```

  - `PID`: Process ID
  - `TTY`: The terminal linked to the process
  - The cumulated `CPU TIME` in `hh:mm:ss` format
  - `CMD`: The executable name

Process Status Linux command (II)

- `> ps -aux`

```
pbs@hn3:~$ ps -aux
USER         PID %CPU %MEM    VSZ   RSS TTY      STAT START   TIME COMMAND
root           1  0.0  0.0 168552 11916 ?        Ss   fev20   0:02 /sbin/init sp
root           2  0.0  0.0      0     0 ?        S    fev20   0:00 [kthreadd]
root           3  0.0  0.0      0     0 ?        I<   fev20   0:00 [rcu_gp]
```

  - `a` represents all users
  - `u` represents the user/owner
  - `x` displays processes executed outside of the terminal
    - `USER` represents the user that initiated the process.
    - `%CPU` is the CPU utilization of the process.
    - `%MEM` shows the memory usage.
    - `VSZ` means the virtual memory size of the process in KiB.
    - `RSS` is the non-swapped physical memory that a process has used (in kilobytes).
    - `STAT` shows the process state.
    - `START` represents the time at which the command started.

Terminating a process (I)

- `kill` command is used to terminate processes manually.
  - `kill` command sends a signal to a process which terminates the process.
- `kill -l` :To display all the available signals you can use below command option:

```
pbs@hn3:~$ kill -l
 1) SIGHUP       2) SIGINT       3) SIGQUIT      4) SIGILL       5) SIGTRAP
 6) SIGABRT      7) SIGBUS       8) SIGFPE       9) SIGKILL     10) SIGUSR1
11) SIGSEGV     12) SIGUSR2     13) SIGPIPE     14) SIGALRM     15) SIGTERM
16) SIGSTKFLT   17) SIGCHLD     18) SIGCONT     19) SIGSTOP     20) SIGTSTP
21) SIGTTIN     22) SIGTTOU     23) SIGURG      24) SIGXCPU     25) SIGXFSZ
26) SIGVTALRM   27) SIGPROF     28) SIGWINCH    29) SIGIO       30) SIGPWR
31) SIGSYS      34) SIGRTMIN    35) SIGRTMIN+1  36) SIGRTMIN+2  37) SIGRTMIN+3
38) SIGRTMIN+4  39) SIGRTMIN+5  40) SIGRTMIN+6  41) SIGRTMIN+7  42) SIGRTMIN+8
43) SIGRTMIN+9  44) SIGRTMIN+10 45) SIGRTMIN+11 46) SIGRTMIN+12 47) SIGRTMIN+13
48) SIGRTMIN+14 49) SIGRTMIN+15 50) SIGRTMAX-14 51) SIGRTMAX-13 52) SIGRTMAX-12
53) SIGRTMAX-11 54) SIGRTMAX-10 55) SIGRTMAX-9  56) SIGRTMAX-8  57) SIGRTMAX-7
58) SIGRTMAX-6  59) SIGRTMAX-5  60) SIGRTMAX-4  61) SIGRTMAX-3  62) SIGRTMAX-2
63) SIGRTMAX-1  64) SIGRTMAX
```

  - If the user does not specify any signal which is to be sent along with kill command then default `SIGTERM` signal is sent that terminates the process.

Terminating a process (II)

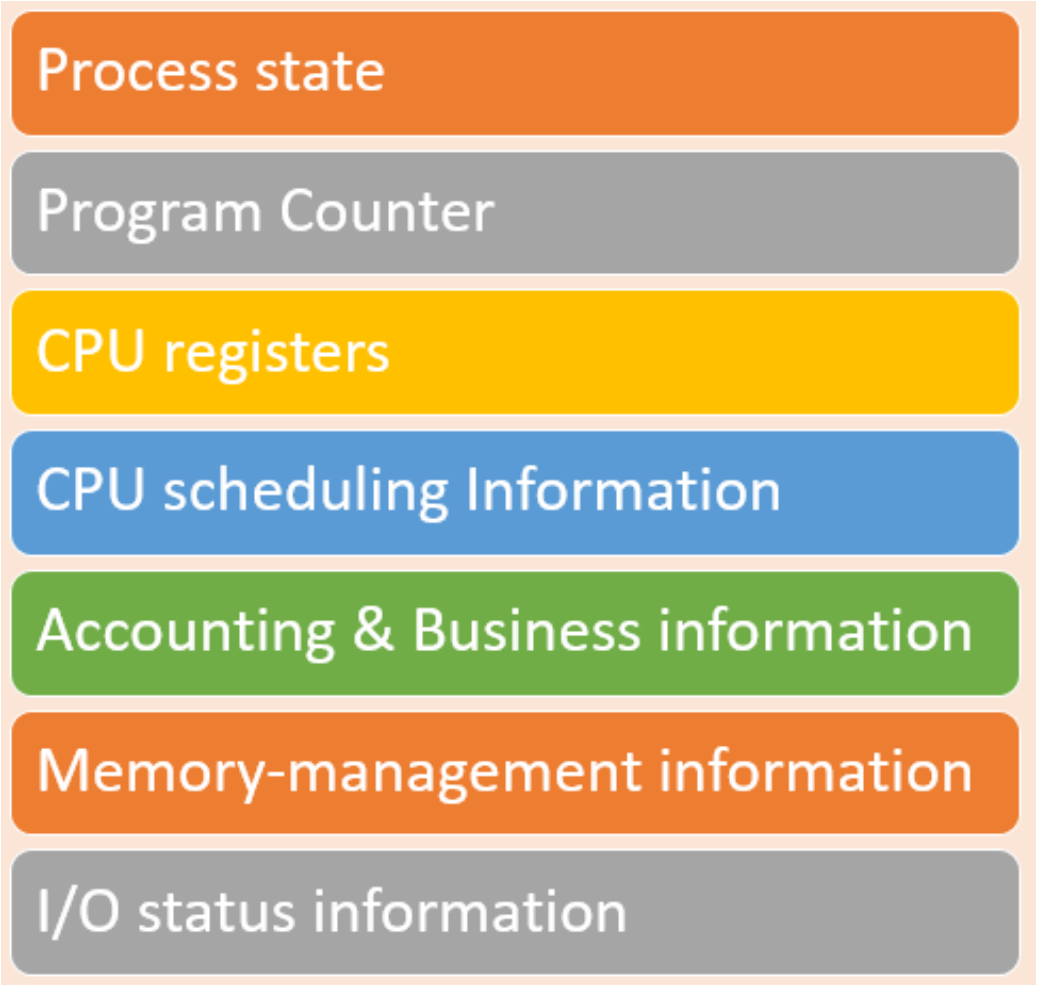
- `> kill $PID`

- `> kill -$signal $PID`

  - Send a `SIGKILL` signal.

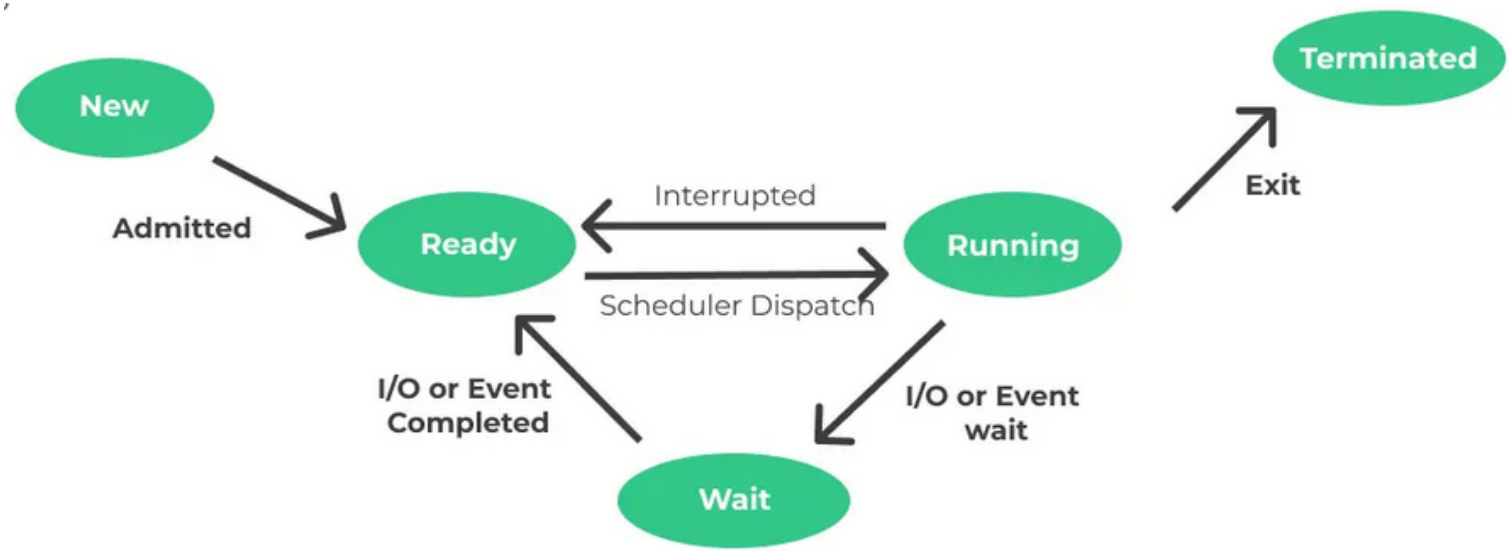
Process Control Block (PCB)

- PCB is a data structure that contains information of the process related to it.



Process States

- **New**: New Process Created
- **Ready**: Process Ready for Processor/computing power allocation
- **Running**: Process getting executing
- **Wait**: Process waiting for signal
- **Terminated**: Process execution completed

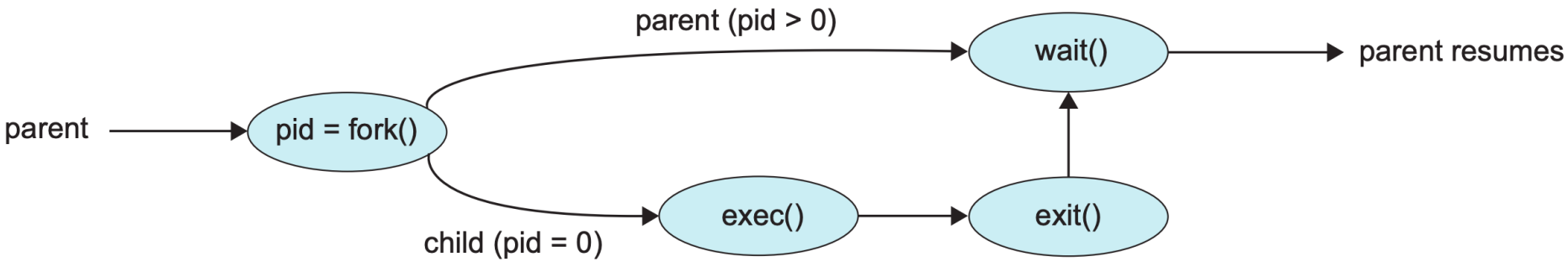


Lifecycle (I)

- A process begins its life when it is **created**.
- A process **goes through different states before it gets terminated**.
- The first state that any process goes through is the creation of itself.
  - Process creation happens through the use of `fork` system call, which creates a new process(child process) by duplicating an existing one(parent process).
  - The process that calls `fork` is the **parent**, whereas the new process is the **child**.
- In most cases, we may want to execute a different program in child process than the parent.
- The `exec` family of function calls creates a new address space and **loads a new program (machine code) into it**.
- Finally, a process exits or terminates using the `exit` system call.

Lifecycle (II)

- A parent process can enquire about the status of a terminated child using `wait` system call.
  - When the parent process uses `wait` system call, the parent process is blocked till the child on which it is waiting terminates.



Application Programming Interface

`fork` System Call

- When a parent process uses `fork()`, it creates a duplicate copy of itself and this duplicates becomes the child of the process.
  - A non-zero value(Process ID of child) is returned to the parent.
  - A value of zero is returned to the child.
  - In case the child is not created successfully due to any issues like low memory, -1 is returned.

```
pid = fork();
// Both child and parent will now start execution from here.
if(pid < 0) {
  //child was not created successfully
  return 1;
}
else if(pid == 0) {
  // This is the child process and child process code goes here
}
else {
  // Parent process code goes here
}
printf("This is code common to parent and child");
```

Check: `https://man7.org/linux/man-pages/man2/fork.2.html`

exec Set of System Calls

- The exec family of functions replaces the current running program(executable) with a new executable.
- This is very useful when you want the child process to run a different program than the parent.

```
/* fork a child process */
pid = fork();
if (pid < 0) {
  /* error occurred */
  return 1;
} else if (pid == 0) {
  /* child process */
  execlp("/bin/ls", "ls", NULL); // A new program(ls executable is loaded into memory
        and executed
} else {
  /* parent process */
  /* parent will wait for the child to complete */
  wait(NULL);
  printf("Child Complete");
}
return 0;
```

Check: https://man7.org/linux/man-pages/man3/exec.3.html

`wait` System Call

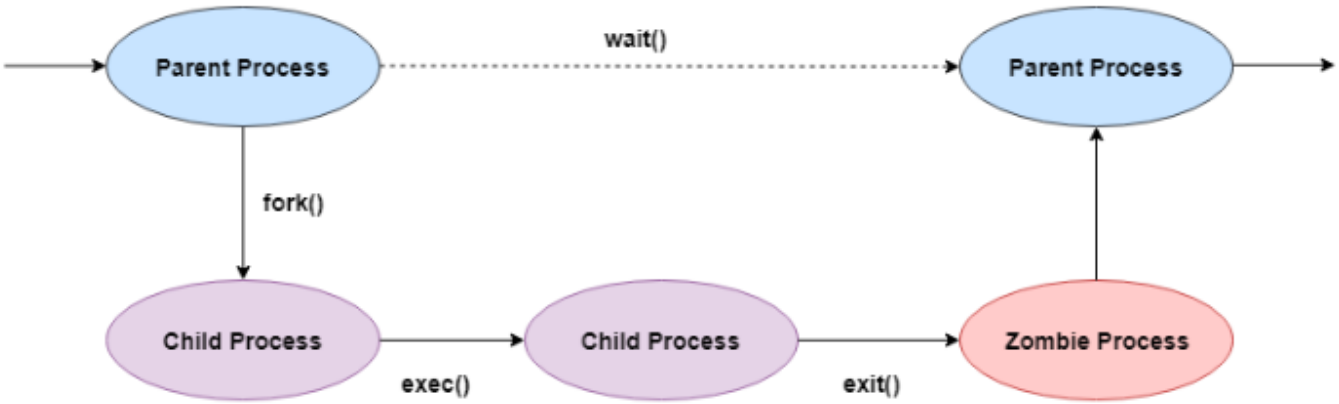
- The parent process may invoke a `wait` system call, which suspends the execution of the parent process while the child executes.
- When the child process terminates, it returns an `exit` status to the operating system, which is then returned to the waiting parent process.
  - Information can be retrieve from returned value using macros like `WIFEXITED`, `WEXITSTATUS`, ...

```
 int stat;
// This status 1 is reported by WEXITSTATUS
if (fork() == 0){
  exit(1);
}else{
  wait(&stat);
  if (WIFEXITED(stat)){
    printf("Exit status: %d\n", WEXITSTATUS(stat));
  }
}
```

Check: `https://man7.org/linux/man-pages/man2/wait.2.html`

Why does parent waits for a child process?

- The parent can assign a task to its child and wait till it completes its task.
- Once the child terminates, all the resources associated with child are freed except for the PCB.
- Now, the child is in zombie state. Using `wait()`, parent can inquire about the status of child and then ask the kernel to free the PCB.
- In case parent does not uses wait, the child will remain in the zombie state



`exit` System Call

- A computer process terminates its execution by invoking the `exit` system call.
- When the child process terminates ("dies"), either normally by calling `exit`, or abnormally due to a fatal exception or signal (e.g., `SIGTERM`, `SIGINT`, `SIGKILL`), an exit status is returned to the operating system and a `SIGCHLD` signal is sent to the parent process.
- The exit status can then be retrieved by the parent process via the wait system call.
- `exit` system call is not always implicit in a program.
  - A process can also terminate/return if control reaches the end of the function.

`getpid` System Call

- `getpid` returns the process ID of the calling process.
- `getppid` returns the process ID of the parent of the calling process.
  - If the calling process was created by the `fork` function and the parent process still exists at the time of the `getppid` function call, this function returns the process ID of the parent process.
  - Otherwise, this function returns a value of 1 which is the process id for init process.

```
int pid;
pid = fork();
if (pid == 0){
  printf("(Child) my pid is: %d\n", getpid());
  printf("(Child) my parent pid : %d\n", getppid());
}
return 0;
```

Check: `https://man7.org/linux/man-pages/man2/getpid.2.html`