Operating Systems

Processes

Yanyan Zhuang

Department of Computer Science

http://www.cs.uccs.edu/~yzhuang

Recap of the Last Class

- Computer hardware
  - Time-multiplexed
  - Space-multiplexed
- OS components
  - Process management
  - Memory management
  - File and storage management

Process

- Process == Program??

Process

- Definition
  - An instance of a program running on a computer
    - Process == Program in execution
  - An abstraction that supports running programs

Process

- Definition
  - An instance of a program running on a computer
    - Process == Program in execution
  - An abstraction that supports running programs

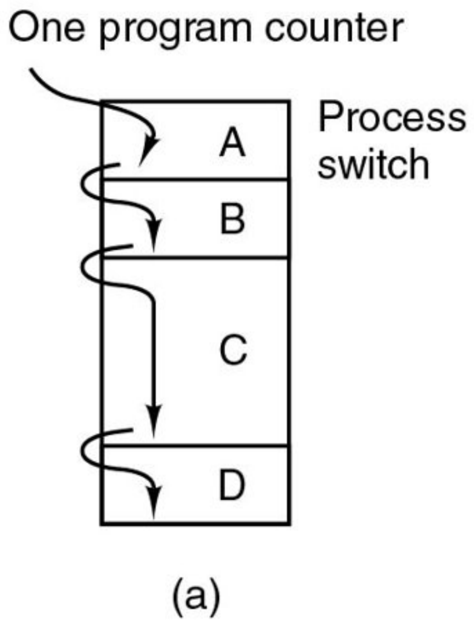
- Allows the OS to simplify
  - Resource allocation/management
  - OS manages resources and internal state of every process
    - Dynamic process state
      - Registers: PC, SP,…
      - Memory: address space, code, stack, heap …
      - I/O status: opened files …

Program v.s Process

- Program != Process
  - Program = static code + data
  - Process = dynamic instantiation of code + data + files …
- No 1:1 mapping
  - A program can invoke many processes
    - Running the same program twice
    - A program contains fork()

The Process Model

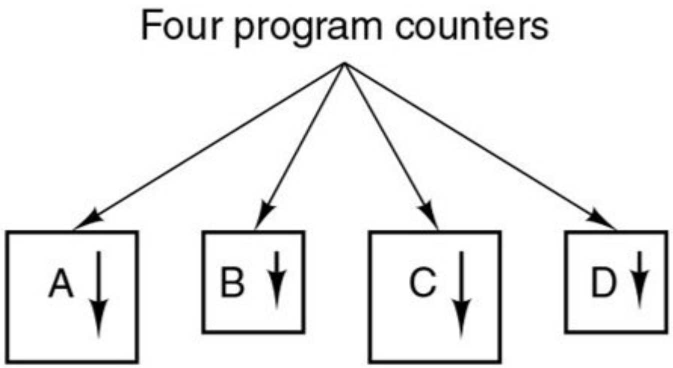
- Assume a single program counter
  - Each process in unique memory location
  - CPU switches back and forth from process to process



Multiprogramming of four programs

The Process Model

- Each process has their own flow of control (own logical program counter)
- Each time we switch processes, save the program counter of first process and restore the program counter of the second
  - Hiding the effects of interrupts and blocking system calls

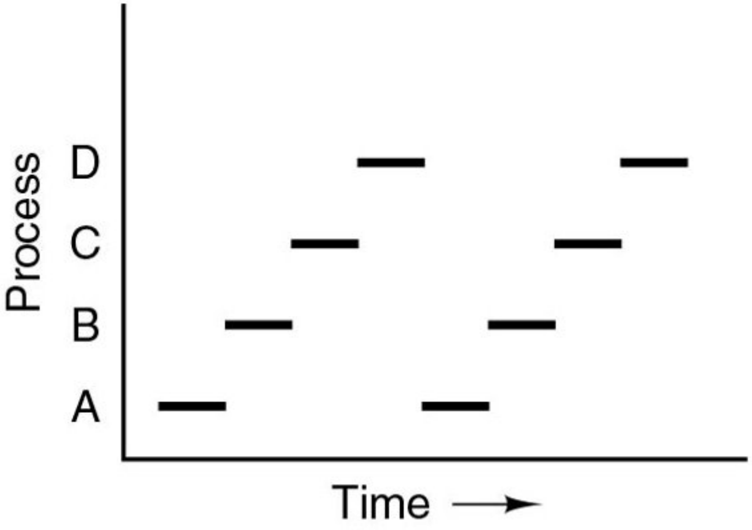


Conceptual model of four independent, sequential processes

(b)

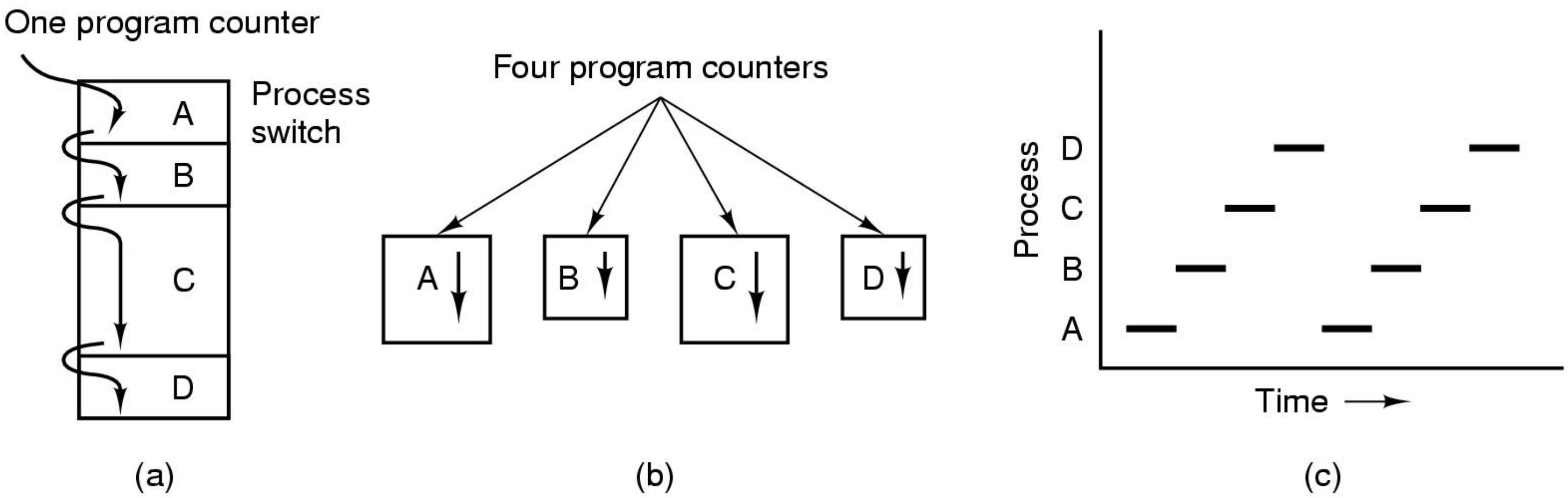
The Process Model

- Assume a single program counter
  - All processes make progress, but only one is active at any given time



(c)

The Process Model



- CPU time can be allocated to different processes
  - ✓ OS normally offers **no timing or ordering** guarantees

Process Creation

- Principal events that cause process creation
  - System initialization
  - Execution of a process creation system call by an existing process
  - User request to create a new process
  - Initiation of a batch job

Process Creation

- Principal events that cause process creation
  - System initialization
  - Execution of a process creation system call by an existing process
  - User request to create a new process
  - Initiation of a batch job
- UNIX example
  - ***fork*** system call creates an **exact copy** of calling process
    - Same memory image, environment settings, and opened files
    - After fork, caller is parent, newly-created process is child

Process Creation

- Principal events that cause process creation
  - System initialization
  - Execution of a process creation system call by an existing process
  - User request to create a new process
  - Initiation of a batch job
- UNIX example
  - ***fork*** system call creates an **exact copy** of calling process
    - Same memory image, environment settings, and open files
    - After fork, caller is parent, newly-created process is child
  - Child process calls ***execve*** to change its memory image and run a new program

After fork()

- The child process returns executing at the exact same point after its parent called fork()
  - fork() returns **twice**: the new PID to the parent, and 0 to the child

```
pid= fork();
if (pid == 0) {
    /* I am the child (0: invalid PID) */
} else {
   /* I am the parent */
}
```

Two processes execute the code! (parent/child share same text)

  - All memory contents of parent/child are identical
  - Both have the same files open at the same position (point to the same file objects)

Putting it Together

```
/* now create new process */
pid = fork();
char *const parmList[] = {”./Helloworld", NULL};
if (pid == 0) /* fork() returns 0 to the child process */
    {
        sleep(1);
        printf("CHILD: My parent's PID: %d\n", getppid());
        execve(“./Helloworld”, parmList);
        printf("retval of Helloworld: %d\n", retval);
        exit(retval);
    }
else /* fork() returns new pid to the parent process */
    {
        printf("PARENT: my child PID: %d\n", pid);
        wait(&status);
        printf("PARENT: Child's exit code is: %d\n”, WEXITSTATUS(status));
        exit(0);
    }
```

Process Management System Calls

- fork: create a new process
  - Child is a **clone** of the parent
  - Shares **some** resources with the parent
- exec: execute a new process image
  - Used in combination with fork
  - Has a family of calls, e.g., execve
- exit: cause voluntary process termination
  - Exit status returned to the parent
- wait: parent wait for child to finish
  - wait(&status): a child's exit status will be stored in status **upon return**
- kill: send a signal to a process (or group)
  - Can cause involuntary process termination

Many of them have C lib call or command line wrappers

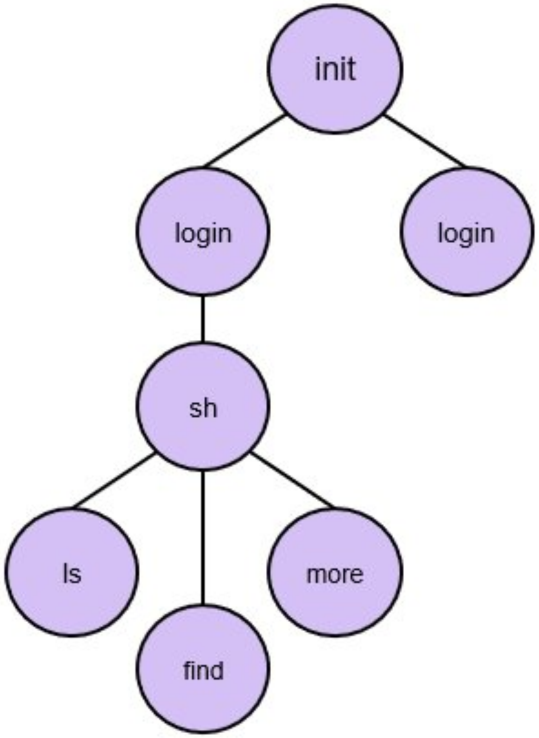
Process Termination

- Conditions which terminate processes
  - Normal exit (voluntary)
  - Error exit (voluntary)
  - Killed by another process (involuntary)

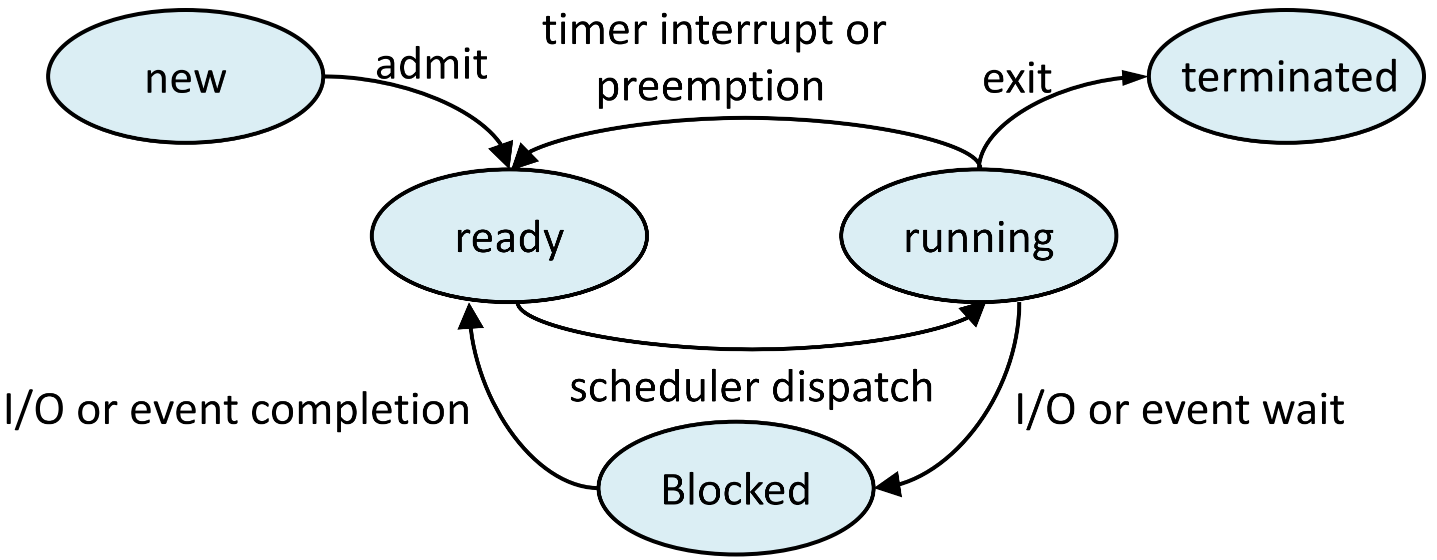
Process Hierarchies (Trees)

- Parent creates a child process, child processes can create its own process

- Forms a hierarchy
  - UNIX: a process and all its children and further descendants form a "process group“
  - ***init***, a special process present in the boot image
- Try: **`pstree -h`**



Process Life Cycle



To allocate CPU time, the OS needs to track **process states**:

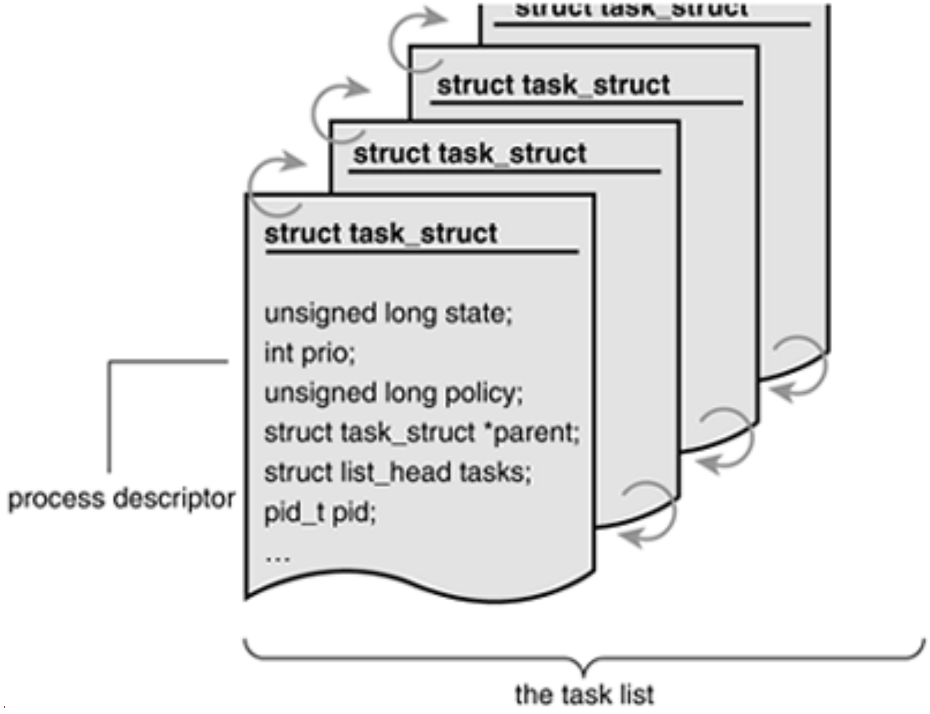
- **Running**: process is currently executed by the CPU
- **Blocked**: process is waiting for some event
- **Ready**: process is waiting to be selected to run

Implementation of Processes

- Process table
  - One entry per process
  - Each entry is called a process control block (PCB)
- Process control block (PCB)
  - OS data structure containing data associated with processes
    - ID
    - Memory address space
    - Hardware registers (e.g., program counter)
    - Opened files
    - ……

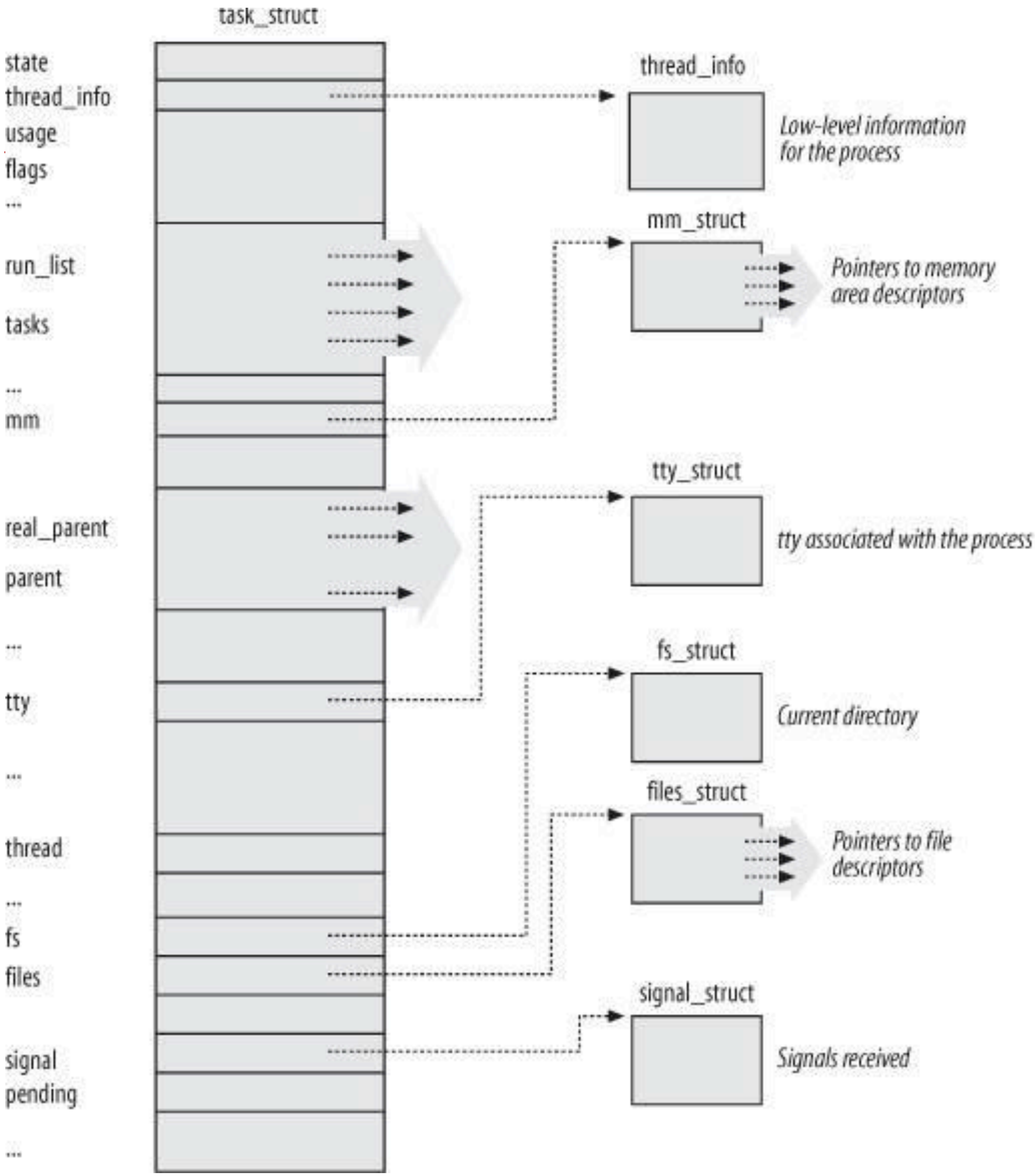
Linux Processes

- Process table: implemented as a linked list/ hashtable
  - Each element: a process descriptor of type **task_struct**
  - Dynamically allocated for each process
  - Contains all info about a specific process



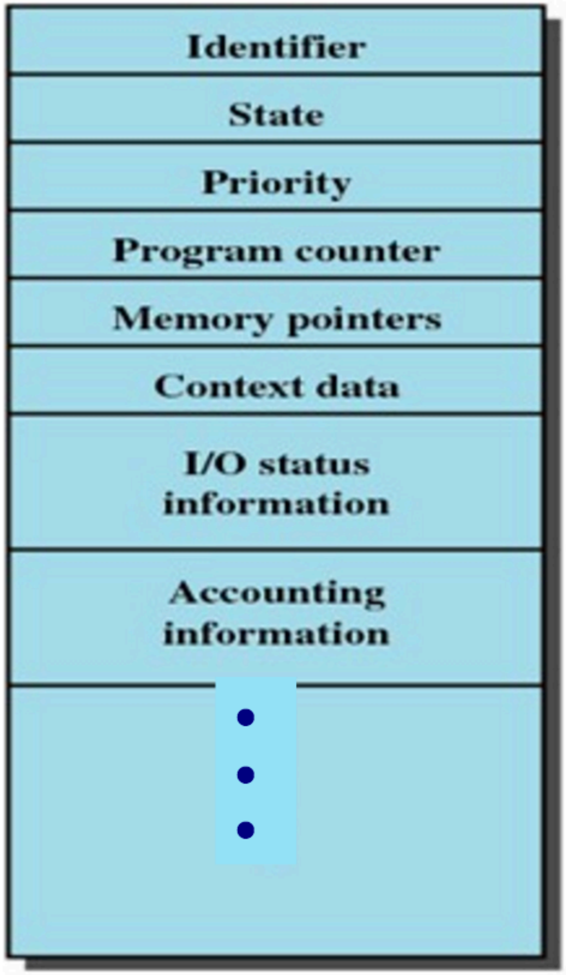
Linux Processes

- Process descriptor (PCB)
  - State
  - Identifiers
  - Scheduling info
  - File system
  - Virtual memory
  - …



Linux Process Descriptor

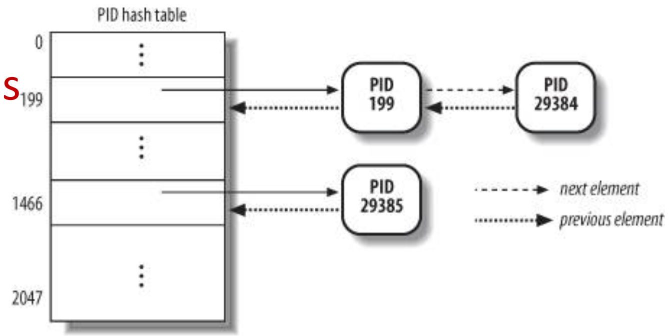
- State
  - TASK_RUNNING
    - Running
  - TASK_INTERRUPTABLE
    - Blocked
  - EXIT_ZOMBIE
    - Terminated by not deallocated
  - EXIT_DEAD
    - Completely terminated



Linux Process Descriptor (cont')

- Identifiers
  - pid: PID of the process/thread
  - ….
- How to get the pointer to a specific process?
  - The **`current`** macro
  - The **`init_task`** macro
  - find_task_by_vpid(pid_t pid)

For projects, don't use this



Kernel modules can only do this to get task_struct:
pid* pid_struct = find_get_pid(int pid);
task_struct* task = pid_task(pid_struct,PIDTYPE_PID);

Use this

Linux Process Descriptor (cont')

- Files
  - fs_struct
    - file system information: root directory, current directory
  - files_struct
    - Information on opened files

Summary

- What is a process?
  - An instantiation of a program
- Program life cycle
  - Ready, running, blocked, new, terminated
- Process implementation
  - Process table, PCB
- Additional practice
  - Download Linux kernel source to your VM, find the following fields in structure task_struct (PCB) in LINUX_SRC_FOLDER/include/linux/sched.h
    - Program counter (try to google)
    - Stack pointer
    - Process ID
    - Opened file descriptors