CSE4509 Operating Systems

Virtual CPU (Processes and Threads)

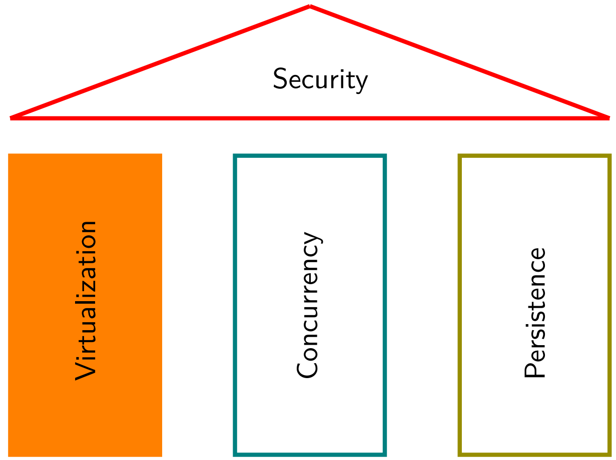
Salman Shamil

United International University (UIU)

Summer 2025

Original slides by Mathias Payer and Sanidhya Kashyap [EPFL]

Virtualization



Lecture Topics

- The **process** abstraction
- A notion on **address spaces**
- **How** processes are **created**
- **Interaction** between processes and the **OS**

This slide deck covers chapters 4–6 in OSTEP.

What is a Process?

Figure 1: Processes are controlled by the Operating System

Think of Process as a **running program**, initiated and maintained by the operating system.

CPU, Memory, and Disk: Limitations

Status quo[1]:

- CPUs execute an endless stream of instructions (in memory)
- All system memory is in a contiguous physical address space
- The disk is a finite set of blocks
- All instructions execute in privileged mode

Is it good enough to run multiple programs simultaneously?

To handle concurrent programs, the OS must *separate* the execution of different programs, providing the *illusion* to programs that each program is ***the only running program***.

The ***virtual* process abstraction** provides this illusion.

[1]Some simplifying assumptions apply to make our life easier.

Process Abstraction

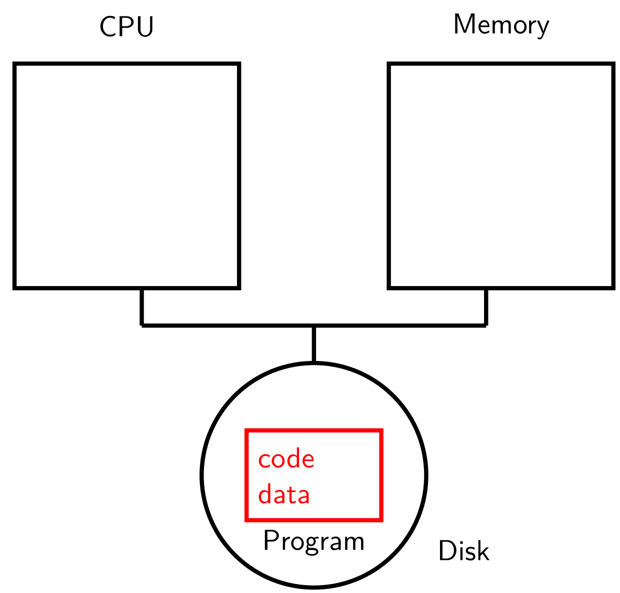
- A ***program*** consists of static code and data, e.g., on the disk.
- A ***process*** is an instance of a program (at any time there may be 0 or more instances of a program running, e.g., a user may run multiple concurrent shells).

Process Definition

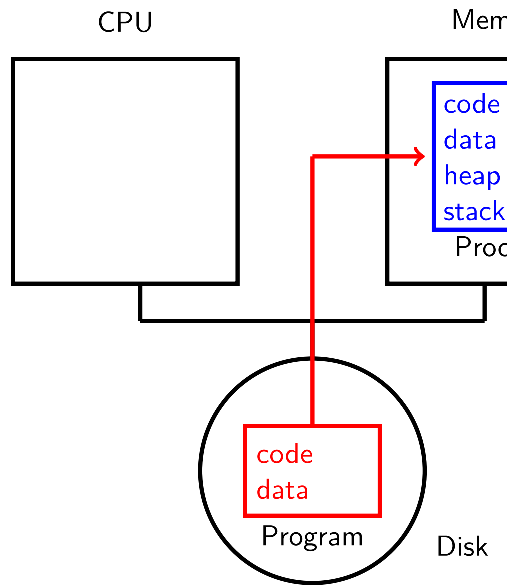
*A **process** is an **execution stream** in the context of a **process state**. The execution stream is the sequence of executing instructions (i.e., the "thread of control"). The process state encompasses everything that executing instructions can affect or are affected by (e.g., registers, address space, persistent state such as files).*

Note: state has two sides, the process view and the OS view. The OS keeps track of the address space and persistence.

Process Creation (1/2)



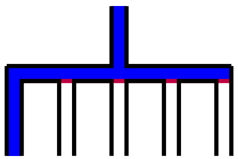
Process Creation (2/2)



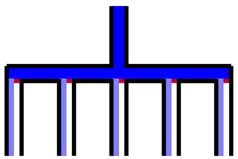
Comparison of Terms:

- A ***program*** is on-disk application, consisting of code and data; programs become a process when they are executed
- A ***process*** is a running instance of a program. A process starts with a single thread of execution and an address space.
- A process can launch multiple ***threads*** of execution in the same address space. Each thread receives its own stack but they share global data, code, and heap.

Sharing Resources: Two Forms



Time sharing (one at a time)



Space sharing (all a little)

- Shared in time (I get to use the toolbox exclusively)
- Shared in space (I get to pick the two screwdrivers I need)

Virtualizing the CPU

- Goal: give each process the illusion of exclusive CPU access
- Reality: the CPU is a shared resource among all processes
- Different strategies for CPU, memory, and disk
  - ***CPU:*** time sharing, alternate between tasks
  - ***Memory:*** space sharing (more later)
  - ***Disk:*** space sharing (more later)

OS provides Process Abstraction

- When the user executes a program, the OS creates a process.
- OS time-shares CPU across multiple processes.
- OS scheduler picks ***one*** of the executable processes to run.
  - Scheduler must keep a list of processes
  - Scheduler must keep metadata for policy

Difference between Policy and Mechanism

- ***Policy:*** High-level decision. e.g., which process to run?
- ***Mechanism:*** Low-level implementation. e.g., how to switch from one process to another?

Distinction between policy and mechanism enables modularity. The scheduling policy is independent of the context switch functionality.

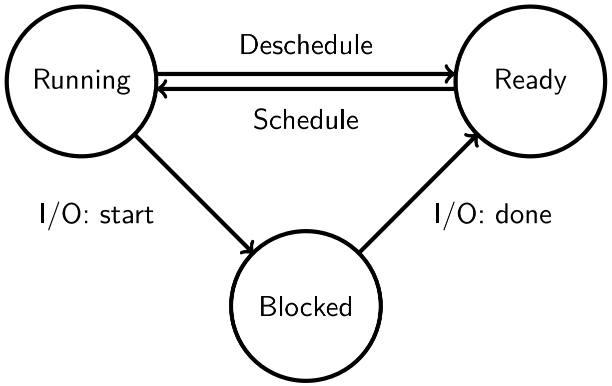
Process Creation

- OS allocates internal data structures
- OS allocates an address space
  - Loads code, data from disk
  - Creates runtime stack, heap
- OS opens basic files (STDIN, STDOUT, STDERR)
- OS initializes CPU registers

Process States

- ***Running***: this process is currently executing
- ***Ready***: this process is ready to execute (and will be scheduled when the policy decides so)
- ***Blocked***: this process is suspended (e.g., waiting for some action; OS will unblock it when that action is complete)
- ***New***: this process is being created (to ensure it will not be scheduled)
- ***Dead***: this process has terminated (e.g., if the parent process has not read out the return value yet)

Process State Transitions



Example: Process State Transitions

Time	Process 0	Process 1	Notes
1	Running	Ready	
2	Running	Ready	
3	Running	Ready	P0 initiates I/O
4	Blocked	Running	P0 is blocked, P1 runs
5	Blocked	Running	
6	Blocked	Running	
7	Blocked	Running	I/O completes
8	Ready	Running	P1 is complete/exits
9	Running	-	

OS Data Structures

- OS maintains data structure (array/list) of active processes.
- Information for each process is stored in a process control block (on Linux, this is called `task struct`) that contains:
  - Process identifier (PID)
  - Process state (e.g., ready)
  - Pointer to parent process (`cat /proc/self/status`)
  - CPU context (if process is not running)
  - Pointer to address space (`cat /proc/self/maps`)
  - Pointer to list of open files (file descriptors, `cat /proc/self/fdinfo/*`)

Distinction: Program / Process / Thread

- ***Program:*** consists of an executable on disk. Contains all information to boostrap a process
- ***Process:*** a running instance of a program; has data section and stack initialized
- ***Thread:*** a process can have multiple threads in the same address space (computing on the same data)

Processes vs Threads

- A thread is a "lightweight process" (LWP)
  - A thread consists of a stack and register state (stack pointer, code pointer, other registers).
  - Each process has one or more threads.

For example, two processes reading address `0xc0f3` may read different values. While two threads in the same process will read the same value.

Requesting OS Services

- Processes can request services through the system call API (Application Programming Interface).
- System calls transfer execution to the OS (the OS generally runs at higher privileges, enabling privileged operations).
- Sensitive operations (e.g., hardware access, raw memory access) require (execution) privileges.
- Some system calls (e.g., `read`, `write`) may cause the process to block, allowing the OS to schedule other processes.
- Libraries (the libc) hide system call complexity, export OS functionality as regular function calls.

Process API

The process API enables a process to control itself and other processes through a set of system calls:

- `fork()` creates a new child process (a copy of the process)
- `exec()` executes a new program
- `exit()` terminates the current process
- `wait()` blocks the parent until the child terminates
- This is a small subset of the complex process API (more later)

Process API: `fork()`, creating a new process

- The OS allocates data structures for the new process (child).
- The OS makes a copy of the caller's (parent's) address space.
- The child is made ready and added to the list of processes.
- `fork()` returns different values for parent/child.
- Parent and child continue execution in ***their own separate copy*** of their address space.

Process API: fork() demo!

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

int main(int argc, char* argv[]) {
  printf("Hello, I'm PID %d (%d, %s)\n", (int)getpid(),
         argc, argv[0]);
  int pid = fork();
  if (pid < 0) exit(-1);  // fork failed
  if (pid == 0) {
    printf("o/ I'm PID %d\n", (int)getpid());
  } else {
    printf("\\o, my child is PID %d\n", pid);
  }
  return 0;
}
```


Process API: wait(), waiting for a child

- Child processes are tied to their parent.
- `exit(int retval)` takes a return value argument.
- Parent can `wait()` for termination of child and read child's return value.

Process API: exec(), executing a (new) program

- Always executing the same program is boring (we would need one massive program with all functionality, e.g., `emacs`).
- `exec()` replaces address space, loads new program from disk.
- Program can pass command line arguments and environment.
- Old address space/state is destroyed except for STDIN, STDOUT, STDERR which are kept, allowing the parent to redirect/rewire child's output!

Why do we need fork() and exec()?

Assume a user wants to start a different program. For that, the operating system needs to create a new process and create a new address space to load the program.

Let's use divide and conquer:

- `fork()` creates a new process with a copy of this address space
- `exec()` creates a new address space for a program

fork() and exec() in action

```c
int main(int argc, char *argv[])
{
    int rc = fork();
    if (rc < 0) {
        // fork failed; exit
        fprintf(stderr, "fork failed\n");
        exit(1);
    } else if (rc == 0) {
        // child: redirect standard output to a file
        close(STDOUT_FILENO);
        open("./p4.output", O_CREAT|O_WRONLY|O_TRUNC, S_IRWXU);

        // now exec "wc"...
        char *myargs[3];
        myargs[0] = strdup("wc");   // program: "wc" (word count)
        myargs[1] = strdup("p4.c"); // argument: file to count
        myargs[2] = NULL;           // marks end of array
        execvp(myargs[0], myargs);  // runs word count
    } else {
        // parent goes down this path (original process)
        int wc = wait(NULL);
        assert(wc >= 0);
    }
    return 0;
}
```

Libraries to include...

```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>
#include <fcntl.h>
#include <assert.h>
#include <sys/wait.h>
```


A Tree of Processes

- Each process has a parent process
- A process can have many child process
- Each process again can have child processes

```
 3621  ?      Ss    \_ tmux
 3645  pts/2  Ss+   |   \_ -zsh
 3673  pts/3  Ss+   |   \_ -zsh
 4455  pts/4  Ss+   |   \_ -zsh
27124  pts/1  Ss+   |   \_ -zsh
21093  pts/5  Ss    |   \_ -zsh
10589  pts/5  T     |   |   \_ vim 02-review.md
10882  pts/5  R+    |   |   \_ ps -auxwf
10883  pts/5  S+    |   |   \_ less
21264  pts/7  Ss    |   \_ -zsh
 1382  pts/7  T     |   |   \_ vim /home/gannimo/notes.txt
14368  pts/9  Ss    |   \_ -zsh
29963  pts/9  S+    |       \_ python
```


Limited Direct Execution

A process executes instructions directly on the CPU.

Issues with running directly on hardware:

- Process could do something illegal (read/write to memory that does not belong to the process, access hardware directly)
- Process could run forever (OS must stay in control)
- Process could do something slow, e.g., I/O (OS may want to switch to another process)

Solution: OS maintains some control with help from hardware. For example, the OS maintains timers to intercept the execution at regular intervals and the process may not execute privileged instructions that access the hardware directly (*user mode vs kernel mode*).

Summary

- Processes are a purely ***virtual concept***
- Separating policies and mechanisms enables modularity
- OS is a server, ***reacts to requests*** from hardware and processes
- Processes are ***isolated*** from the OS/other processes
  - Processes have no direct access to devices
  - Processes run in virtual memory
  - OS provides functionality through system calls
- A process consists of an address space, associated kernel state (e.g., open files, network channels), and one or more threads of execution