CS-323 Operating Systems

Processes (and threads)

Mathias Payer

EPFL, Fall 2019

Topics covered in this lecture

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

What is a Process?

Figure 1: Processes are controlled by the Operating System

CPU, memory, and disk: limitations

Status quo:

- The CPU executes an endless stream of instructions
- All system memory is in a single physical address space
- The disk is a finite set of blocks

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

CPU, memory, and disk: limitations

Status quo:

- The CPU executes an endless stream of instructions
- All system memory is in a single physical address space
- The disk is a finite set of blocks

To handle concurrent programs, the OS must *separate* the execution of different programs, providing the *illusion* to programs that they are the only running program.
The ***virtual*** **process abstraction** provides this illusion.

Process abstraction

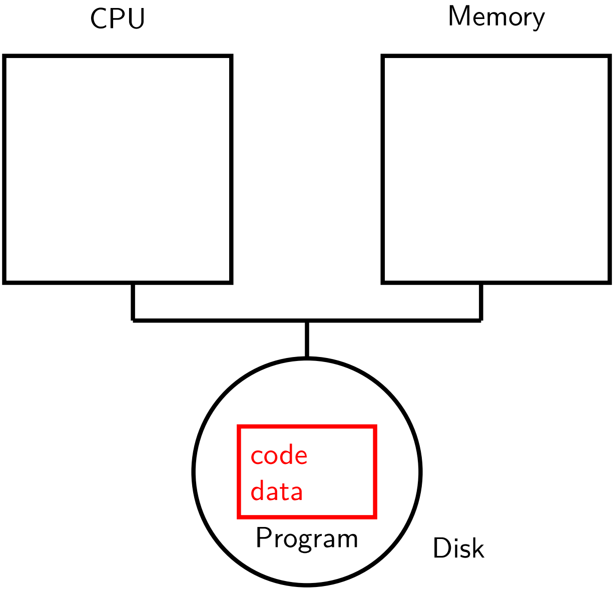
- A program consists of static code and data, e.g., on the disk.
- A process is an instance of a program (at any time there may be 0 or more instances of a program running, e.g., a user may run multiple concurrent shells).

Process definition

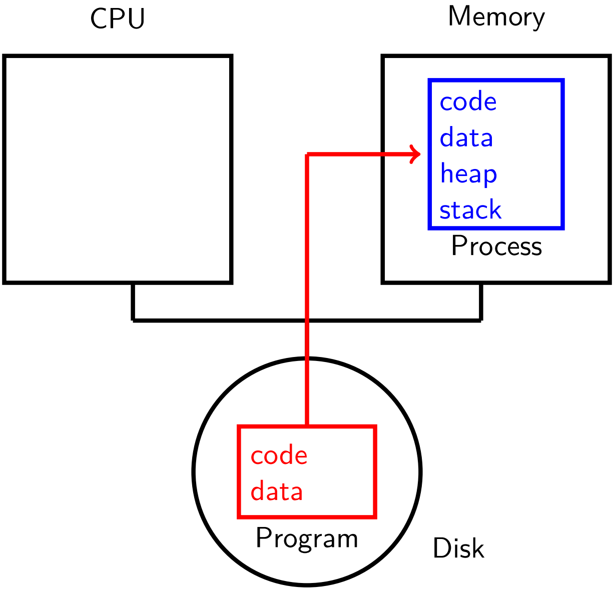
> *A process is an execution stream in the context of a process sate. The execution stream is the sequence of executing instructions (i.e., the "thread of control"). The process state captures everything executing instructions can affect or are affected by (e.g., registers, address space, persistent 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 process (1/2)



Process creation process (2/2)



Virtualizing the CPU

- Goal: give each process the illusion of exclusive CPU access
- Reality: the CPU is a shared resource among all processes
- Two approaches: time sharing or space sharing
  - CPU: time sharing, alternate between tasks
  - Memory: space sharing (more later)
  - Disk: space sharing (more later)

OS provides process abstraction

- When you execute a program, the OS creates a process.
- OS time shares CPU across multiple processes.
- OS scheduler picks which 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: which process to run
- Mechanism: 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.

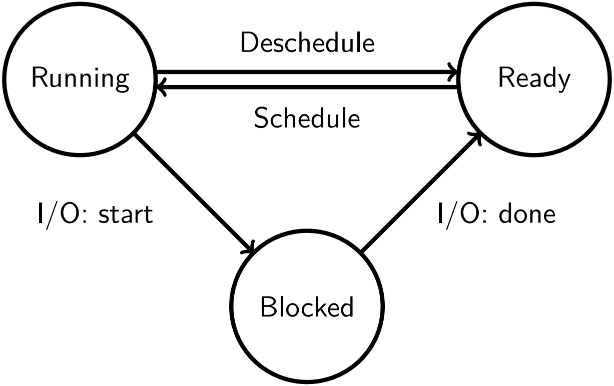
Process creation

- 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 (or `task struct` on Linux), e.g.,
  - Process identifier (PID)
  - Process state (e.g., runnable)
  - 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 between processes and 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 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 (next week: how can we efficiently handle the copy of address spaces?)

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

- Always copying 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!

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: 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;
}
```

Ensuring efficient execution

Process executes instructions directly on the CPU.

Ensuring efficient execution

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 and hardware maintain some control with help from hardware

Process isolation policy

- On most operating systems, processes are:
  - Isolated from each other
  - Isolated from the OS
- Isolation is a core requirement for security:
  - Constraints bugs to the process
  - Enables privilege isolation
  - Enables compartmentalization (breaking complex systems into independent fault domains)

What mechanism allows process isolation?

Process isolation mechanism

- Virtual memory: one (virtual) address space per process
- Different execution modes: OS executes at higher privileges
  - Process executes in user mode (ring 3 on x86)
  - OS executes in super mode (ring 0 on x86)

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 hardware access
  - 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

Don't forget to get your learning feedback through the Moodle quiz!