

# CPU Virtualization: Processes

## CS 537: Introduction to Operating Systems

Louis Oliphant

University of Wisconsin - Madison

Fall 2024

# Administrivia

- Check that you have a `~cs537-1/handin/<username>/P1/` directory and that you can write to it. This is where you should turn in your project 1 solution.
- Want to learn the GNU/Linux Command Line? Read the online book at <https://linuxcommand.org>

# Agenda

## Today

- What is a process and what is its lifecycle? (abstraction)
- How does an OS manage processes? (mechanism)
- How can you create and work with processes? (API)

## Next Time

- How should the OS decide which process gets to execute and for how long (policy)

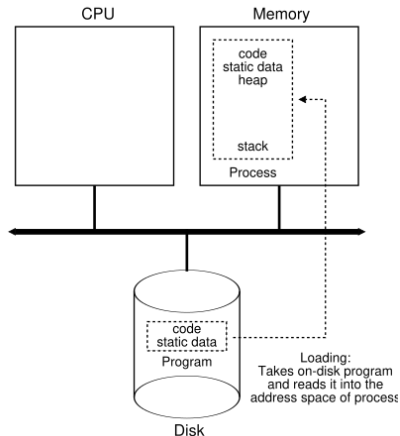
## Aside – CS Terms

- Abstraction** a concept-object that mirrors common features or attributes of non-abstract objects.
- Mechanism** Low-level machinery (methods or protocols) that implement a needed piece of functionality.
- Policy** An algorithm for making some decision within the OS.
- API** Application Program Interface is a type of public interface a program offers as a service to other programs.

# Process

A program is a passive collection of instructions (typically on disk).

A **process** is the abstraction provided by the OS of a **running program**.



# Machine State of a Process

The **machine state**: What a program can read or change when it is running.

- **Registers** (general purpose, stack pointer, program counter, frame pointer, etc.)
- **Address space** (heap, stack, etc.)
- **Open files**

OS will need to save this state to context switch between processes

# OS Control of Processes

- **Create** – When you type a command (or click on an application icon), the OS is invoked to create a new process.
- **Destroy** – OS provides a way to forcefully destroy a process.
- **Wait** – It is useful to be able to wait for a process to stop running.
- **Miscellaneous Control** – e.g. suspend (temporarily stop) a process and resume it again.
- **Status** – Get information about a process (e.g. how long has it run for?)

# Creation of A Process by OS

- Load data from disk to memory
- Allocate space for the run-time **stack** and initialize the stack with arguments (i.e. fill in the parameters for `argc` and `argv`)
  - \* Allocate memory for program's **heap**. Initially small, but OS may grow the heap as needed.
- Setup initial **file descriptors** (`stdin`, `stdout`, `stderr`).
- Transfer control of the CPU to the newly-created process (i.e. `main()`).

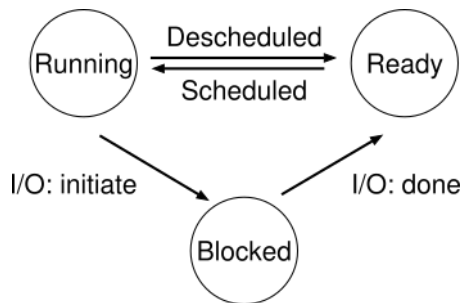


## Aside – OSTEP Homeworks

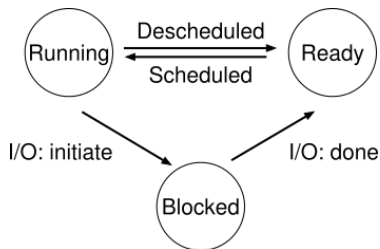
- Optional homeworks corresponding to chapters in book
- Little simulators to help you understand
- Can generate problems and solutions

<https://github.com/remzi-arpacidusseau/ostep-homework>

# Process Life Cycle



# Process Scheduling



```
./process-run.py -l 3:100,3:50
```

Time	PID: 0	PID: 1	CPU	I/Os
1	RUN:cpu	READY	1	
2	RUN:cpu	READY	1	
3	RUN:cpu	READY	1	
4	DONE	RUN:cpu	1	
5	DONE	RUN:io	1	
6	DONE	BLOCKED		1
7	DONE	BLOCKED		1
8	DONE	BLOCKED		1
9	DONE	BLOCKED		1
10	DONE	BLOCKED		1
11*	DONE	RUN:io_done	1	
12	DONE	RUN:cpu	1	

All IO takes 5 time slices

# Direct Execution

For efficiency, we want processes to run directly on hardware

## Problems

- ❶ Process could do something illegal  
e.g., read/write other processes' memory
- ❷ Process could run forever  
OS needs to be able to switch between processes
- ❸ Process could do something slow  
OS wants to use resources efficiently

## Solution

**LIMITED DIRECT EXECUTION** – OS and hardware maintain some control

## Limited Direct Execution Prob #1 – Restricted Ops

How can we ensure user process can't harm others?

**Solution – Privilege Levels Supported by Hardware (bit of status)**

- User processes run in user mode (restricted mode)
- OS runs in kernel mode (not restricted)
  - Instructions for interacting with devices
  - Could have many privilege levels (advanced topic)

**How can process perform restricted instruction?**

- Ask the OS to do it through a system call
- Change privilege level as system call is made (trap)

# System Call



Figure 1: System Call

- P can only see its own memory because it runs in **user mode**.
- P wants to call `read()` but no way to call it directly.

## xv6 Traps and System Calls

### trap.h

```
#define T_ALIGN      17      // alignment ch
#define T_MCHK       18      // machine che
#define T_SIMDERR    19      // SIMD floati

// These are arbitrarily chosen, but with care
// processor defined exceptions or interrupt v
#define T_SYSCALL     64      // system call
#define T_DEFAULT    500     // catchall
```

### syscall.h

```
#define SYS_exit      2
#define SYS_wait      3
#define SYS_pipe      4
#define SYS_read      5
#define SYS_kill      6
#define SYS_exec      7
#define SYS_fstat     8
#define SYS_chdir     9
#define SYS_dup      10
```

# System Call

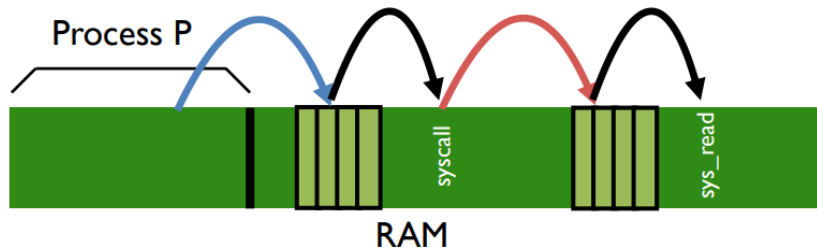


Figure 2: System Call

```
movl $5, %eax;
```

```
int $64
```



# System Call

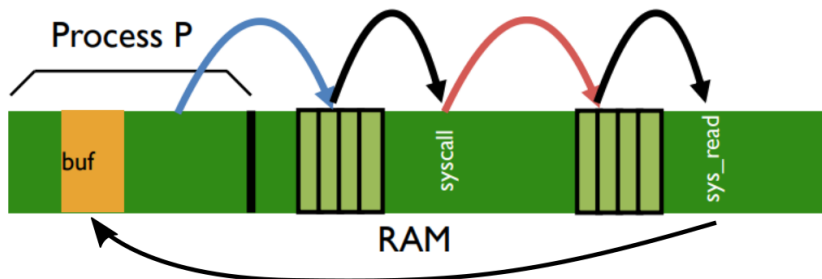


Figure 3: System Call

- Kernel can access user memory to fill in user buffer
- return-from-trap at end to return to Process P

OS @ run (kernel mode)	Hardware	Program (user mode)
Create entry for process list		
Allocate memory for program		
Load program into memory		
Setup user stack with argv		
Fill kernel stack with reg/PC		
<b>return-from-trap</b>	restore regs (from kernel stack) move to user mode jump to main	Run main() ... Call system call <b>trap</b> into OS
	save regs (to kernel stack) move to kernel mode jump to trap handler	
Handle trap		
Do work of syscall		
<b>return-from-trap</b>	restore regs (from kernel stack) move to user mode jump to PC after trap	... return from main <b>trap</b> (via <code>exit()</code> )
Free memory of process		
Remove from process list		

## Limited Direct Execution Prob #2 CPU Sharing

- **Cooperative Approach:** Could wait for current process to yield the CPU
- **True multi-tasking:** Could interrupt current process to regain control
  - Guarantee OS can obtain control periodically
  - Hardware generates timer interrupt, allowing OS to *context switch*

# Context Switch

OS @ run (kernel mode)	Hardware	Program (user mode)
		Process A
		...
	<b>timer interrupt</b> save regs(A) $\rightarrow$ k-stack(A) move to kernel mode jump to trap handler	
Handle the trap Call <code>switch()</code> routine save regs(A) $\rightarrow$ proc_t(A) restore regs(B) $\leftarrow$ proc_t(B) switch to k-stack(B) <b>return-from-trap (into B)</b>		
	restore regs(B) $\leftarrow$ k-stack(B) move to user mode jump to B's PC	
		Process B
		...

# Intialize Trap Table and Start Timer

**OS @ boot  
(kernel mode)**

**initialize trap table**

**start interrupt timer**

**Hardware**

remember addresses of...  
syscall handler  
timer handler

start timer  
interrupt CPU in X ms

# OS Data Structures for Managing Processes

## Process control block (PCB) and Process list

```
// Per-process state
struct proc {
    uint sz;                // Size of process memory (bytes)
    pde_t* pgdir;           // Page table
    char *kstack;           // Bottom of kernel stack for this process
    enum procstate state;   // Process state
    int pid;                // Process ID
    struct proc *parent;    // Parent process
    struct trapframe *tf;   // Trap frame for current syscall
    struct context *context; // swtch() here to run process
    void *chan;             // If non-zero, sleeping on chan
    int killed;             // If non-zero, have been killed
    struct file *ofile[NOFILE]; // Open files
    struct inode *cwd;      // Current directory
    char name[16];          // Process name (debugging)
};
```

# Quiz 1 - Processes

## Processes

**You must use your UW-Madison account to access.**

<https://tinyurl.com/cs537-fa24-q1>



# Linux API for Processes

- `fork()` – Used to create a new process
- `exec()` – Replaces the current process image with a new process image (whole family of functions: `execl()`, `execvp()`, `execle()`, `execv()`, `execvp()`, `execvpe()`)
- `wait()` – Waits for a child process to stop or terminate

## Demo

Run chapter 5's demo code from [cpu-api](#) (the programs p1, p2, p3, and p4) to see how these three system calls work.