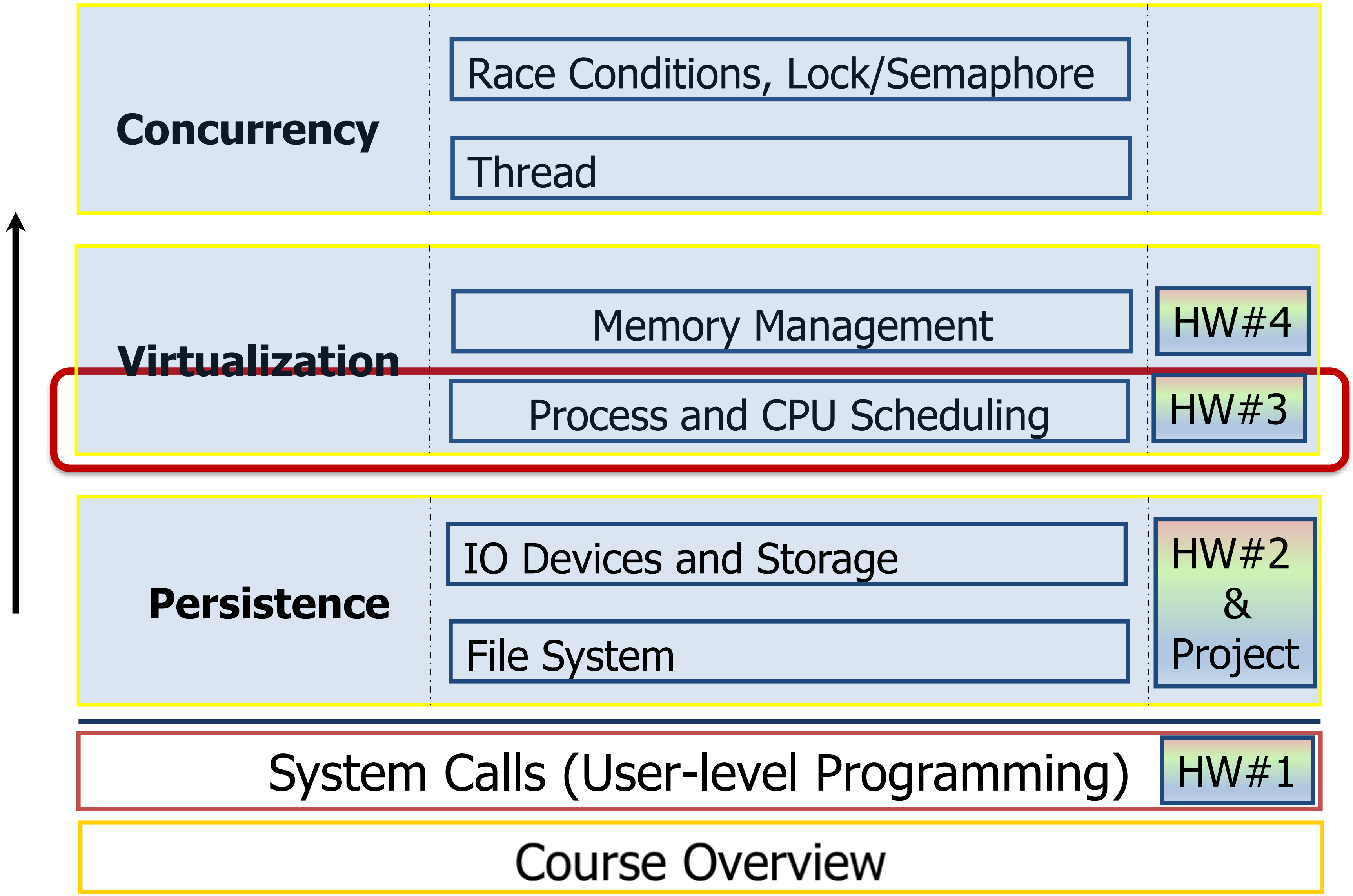
Lecture 7: Virtualizing CPU - Process

The Course Organization (Bottom-up)

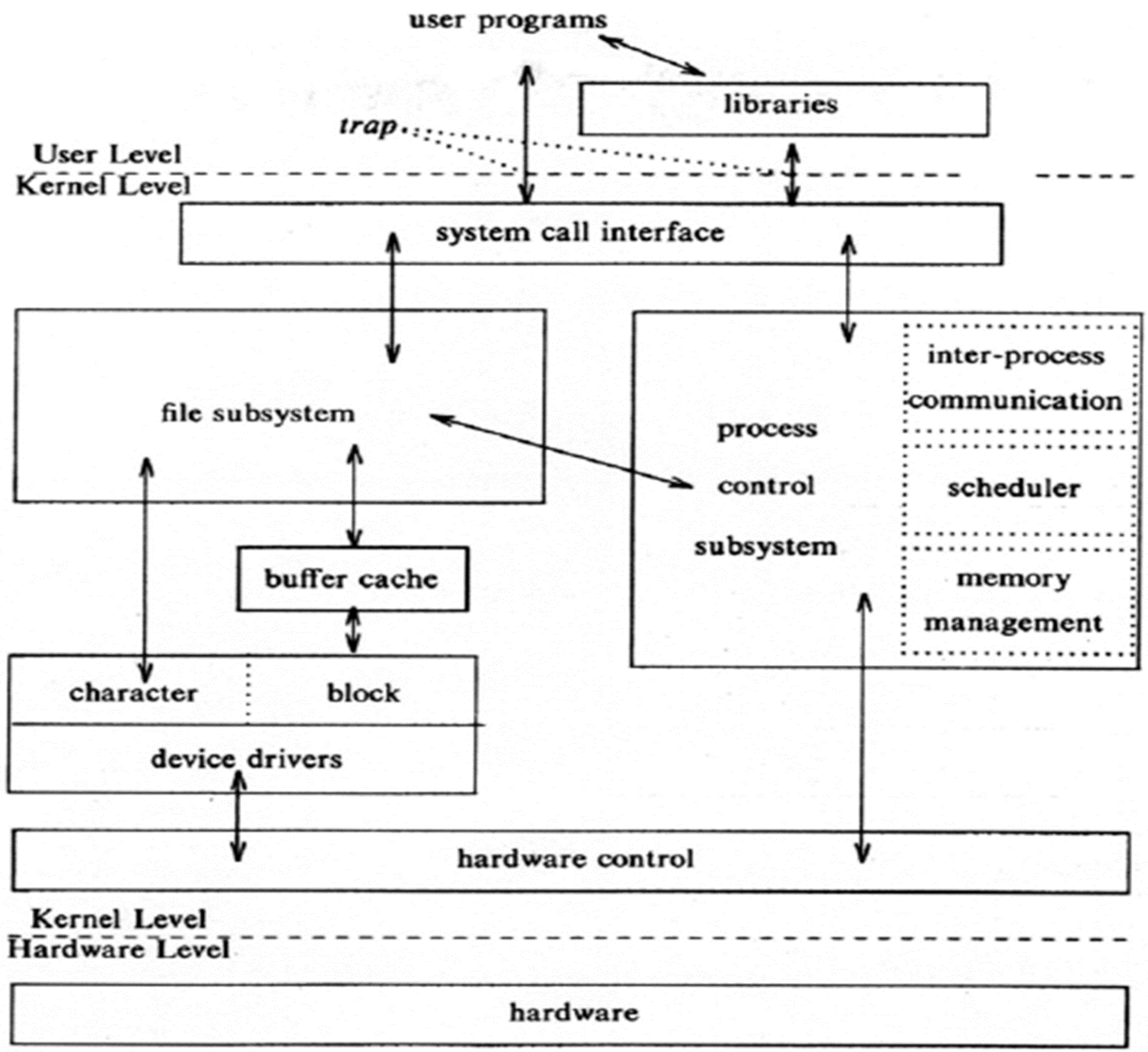


OS – Resource management via virtualization

OS provides services via **System Call** (typically a few hundred) to run **process**, access memory/devices/files, etc.

The OS **manages resources** such as *CPU*, *memory* and *disk* via **virtualization**.

- many programs to run (processes) → Sharing the CPU
- many processes to *concurrently* access their own instructions and data → Sharing memory
- many processes to access devices → Sharing disks



The Design Of The Unix Operating System (Maurice Bach, 1986)

Part I. The Abstraction: The Process

Virtualizing CPUs

- The OS can provide the illusion that many virtual CPUs exist.
- **Time sharing**: Running one process, then stopping it and running another
  - The potential cost is performance.

A Process

A process is a running program.

- Comprising of a process:
  - Memory (address space)
    - Instructions
    - Data section
  - Registers
    - Program counter
    - Stack pointer

Process API

- These APIs are available on any modern OS.
  - **Create**
    - Create a new process to run a program
  - **Destroy**
    - Halt a runaway process
  - **Wait**
    - Wait for a process to stop running
  - **Miscellaneous Control**
    - Some kind of method to suspend a process and then resume it
  - **Status**
    - Get some status info about a process

Process Creation

1. **Load** a program code into memory, into the address space of the process.
   - Programs initially reside on disk in *executable format*.
   - OS perform the loading process lazily.
     - Loading pieces of code or data only as they are needed during program execution.

2. The program's run-time **stack** is allocated.
   - Use the stack for *local variables*, *function parameters*, and *return address*.
   - Initialize the stack with arguments → `argc` and the `argv` array of `main()` function

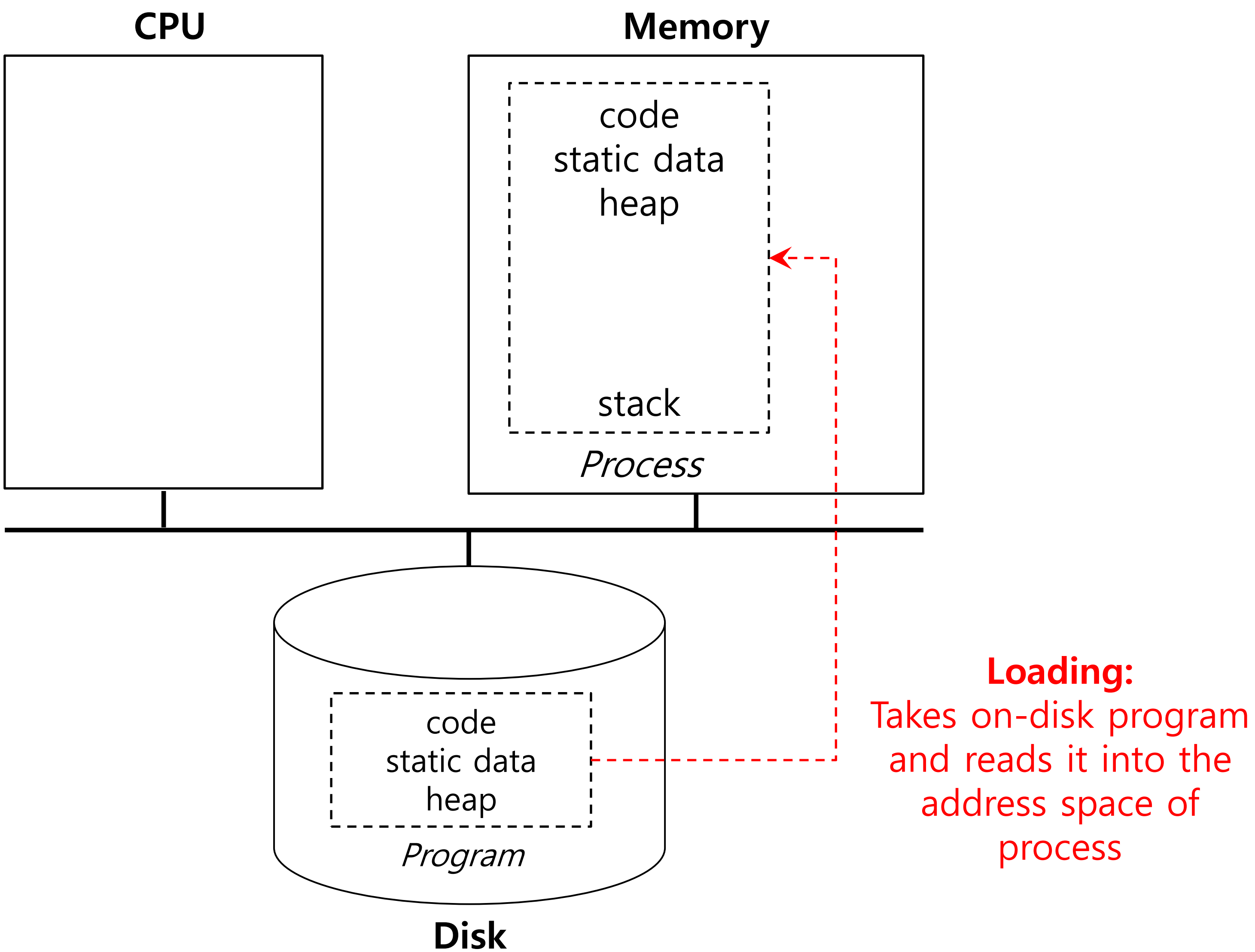
Process Creation (Cont.)

3. The program's **heap** is created.
   - Used for explicitly requested dynamically allocated data.
   - Program request such space by calling `malloc()` and free it by calling `free()`.

4. The OS do some other initialization tasks.
   - Input/output (I/O) setup
     - Each process by default has three open file descriptors.
     - Standard input, output and error

5. **Start the program** running at the entry point, namely `main()`.
   - The OS *transfers control* of the CPU to the newly-created process.

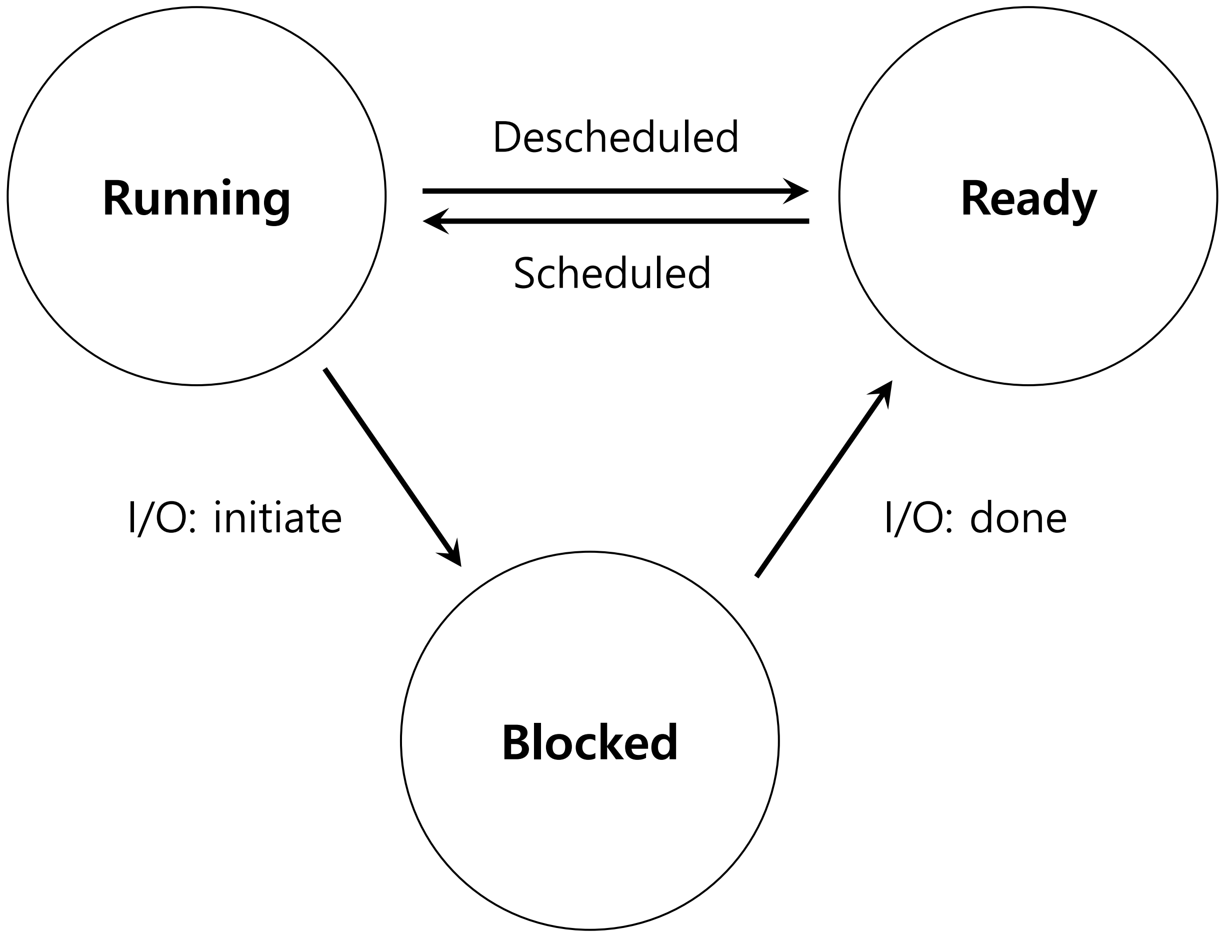
Loading: From Program To Process



Process States

- A process can be one of three states.
  - **Running**
    - A process is running on a processor.
  - **Ready**
    - A process is ready to run but for some reason the OS has chosen not to run it at this given moment.
  - **Blocked**
    - A process has performed some kind of operation.
    - When a process initiates an I/O request to a disk, it becomes blocked and thus some other process can use the processor.

Process State Transition



Data structures

- The OS has some key data structures that track various relevant pieces of information.
  - **Process list**
    - Ready processes
    - Blocked processes
    - Current running process
  - **Register context**

- PCB (Process Control Block)
  - An in-memory data structure that contains information about each process.

Example: The xv6 kernel Proc Structure (proc.h)

```c
// proc.h in xv6
// the registers xv6 will save and restore
// to stop and subsequently restart a process
struct context {
        uint edi;
        uint esi;
        uint ebx;
        uint ebp;
        uint eip;
};

// the different states a process can be in
enum proc_state { UNUSED, EMBRYO, SLEEPING,
                  RUNNABLE, RUNNING, ZOMBIE };
```

https://github.com/mit-pdos/xv6-public

xv6 is a re-implementation of **Dennis Ritchie's and Ken Thompson's Unix Version 6** (v6). xv6 loosely follows the structure and style of v6, but is implemented for a modern x86-based multiprocessor using ANSI C.

Example: The xv6 kernel Proc Structure (Cont.)

```
// the information xv6 tracks about each process
// including its register context and state
struct proc {
    char *mem;                      // Start of process memory
    uint sz;                        // Size of process memory
    char *kstack;                   // Bottom of kernel stack
                                    // for this process
    enum proc_state state;          // Process state
    int pid;                        // Process ID
    struct proc *parent;            // Parent 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
    struct context context;         // Switch here to run process
    struct trapframe *tf;           // Trap frame for the
                                    // current interrupt
};
```

Part II: Limited Direct Execution Mechanism

How to efficiently virtualize the CPU with control?

- The OS needs to share the physical CPU by time sharing.
- Issue
  - **Performance**: How can we implement virtualization without adding excessive overhead to the system?
  - **Control**: How can we run processes efficiently while retaining control over the CPU?

Direct Execution

- Just run the program directly on the CPU.

OS	Program
1. Create entry for process list 2. Allocate memory for program 3. Load program into memory 4. Set up stack with `argc` / `argv` 5. Clear registers 6. Execute call `main()`	
	7. Run `main()` 8. Execute `return` from `main()`
9. Free memory of process 10. Remove from process list	

Without *limits* on running programs,
the OS wouldn't be in control of anything and
thus would be "just a library"

Problem 1: Restricted Operation

- What if a process wishes to perform some kind of restricted operation such as …
  - Issuing an I/O request to a disk
  - Gaining access to more system resources such as CPU or memory

- **Solution**: Using protected control transfer
  - User mode: Applications do not have full access to hardware resources.
  - Kernel mode: The OS has access to the full resources of the machine

System Call

- Allow the kernel to carefully expose certain key pieces of functionality to user program, such as …
  - Accessing the file system
  - Creating and destroying processes
  - Communicating with other processes
  - Allocating more memory

System Call (Cont.)

- **Trap** instruction
  - Jump into the kernel
  - Raise the privilege level to kernel mode

- **Return-from-trap** instruction
  - Return into the calling user program
  - Reduce the privilege level back to user mode

Example: Use “strace” to trace system calls and signals

```
csci3150@csci3150-VirtualBox:/$
csci3150@csci3150-VirtualBox:/$
csci3150@csci3150-VirtualBox:/$ strace ls
execve("/bin/ls", ["ls"], 0xbfbd53f0 /* 48 vars */) = 0
brk(NULL)                               = 0x1806000
access("/etc/ld.so.nohwcap", F_OK)      = -1 ENOENT (No such file or directory)
mmap2(NULL, 8192, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0xb7ef6000
access("/etc/ld.so.preload", R_OK)      = -1 ENOENT (No such file or directory)
openat(AT_FDCWD, "/etc/ld.so.cache", O_RDONLY|O_CLOEXEC) = 3
fstat64(3, {st_mode=S_IFREG|0644, st_size=79757, ...}) = 0
mmap2(NULL, 79757, PROT_READ, MAP_PRIVATE, 3, 0) = 0xb7ee2000
close(3)                                = 0
access("/etc/ld.so.nohwcap", F_OK)      = -1 ENOENT (No such file or directory)
openat(AT_FDCWD, "/lib/i386-linux-gnu/libselinux.so.1", O_RDONLY|O_CLOEXEC) = 3
read(3, "\177ELF\1\1\1\0\0\0\0\0\0\0\0\0\3\0\3\0\1\0\0\0L\0\0004\0\0\0"..., 512) = 512
fstat64(3, {st_mode=S_IFREG|0644, st_size=169960, ...}) = 0
mmap2(NULL, 179612, PROT_READ|PROT_EXEC, MAP_PRIVATE|MAP_DENYWRITE, 3, 0) = 0xb7eb6000
mmap2(0xb7edf000, 8192, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_FIXED|MAP_DENYWRITE, 3, 0x28000) = 0xb7edf000
mmap2(0xb7ee1000, 3484, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_FIXED|MAP_ANONYMOUS, -1, 0) = 0xb7ee1000
close(3)                                = 0
access("/etc/ld.so.nohwcap", F_OK)      = -1 ENOENT (No such file or directory)
openat(AT_FDCWD, "/lib/i386-linux-gnu/libc.so.6", O_RDONLY|O_CLOEXEC) = 3
read(3, "\177ELF\1\1\1\3\0\0\0\0\0\0\0\0\3\0\3\0\1\0\0\0\20\220\1\0004\0\0\0"..., 512) = 512
fstat64(3, {st_mode=S_IFREG|0755, st_size=1942840, ...}) = 0
mmap2(NULL, 1948188, PROT_READ|PROT_EXEC, MAP_PRIVATE|MAP_DENYWRITE, 3, 0) = 0xb7cda000
mprotect(0xb7eaf000, 4096, PROT_NONE)   = 0
mmap2(0xb7eb0000, 12288, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_FIXED|MAP_DENYWRITE, 3, 0x1d5000) = 0xb7eb0000
mmap2(0xb7eb3000, 10780, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_FIXED|MAP_ANONYMOUS, -1, 0) = 0xb7eb3000
```

Example: Use command “ausyscall” to see the system call list

Terminal - csci3150@csci3150-VirtualBox: /

File Edit View Terminal Tabs Help

```
csci3150@csci3150-VirtualBox:/$ ausyscall --dump
Using i386 syscall table:
0       restart_syscall
1       exit
2       fork
3       read
4       write
5       open
6       close
7       waitpid
8       creat
9       link
10      unlink
11      execve
12      chdir
13      time
14      mknod
15      chmod
16      lchown
17      break
18      oldstat
19      lseek
20      getpid
21      mount
22      umount
23      setuid
24      getuid
25      stime
```

Limited Direction Execution Protocol

OS @ boot (kernel mode)	Hardware
initialize trap table	
	remember address of … syscall handler

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 **return-from -trap**		
	restore regs from kernel stack move to user mode jump to main	
		Run main() … Call system **trap** into OS

Limited Direction Execution Protocol (Cont.)

OS @ run (kernel mode)	Hardware	Program (user mode)
	(Cont.)	
	save regs to kernel stack move to kernel mode jump to trap handler	
Handle trap Do work of syscall **return-from-trap**		
	restore regs from kernel stack move to user mode jump to PC after trap	
		… return from main trap (via `exit()`)
Free memory of process Remove from process list		

Problem 2: Switching Between Processes

- How can the OS regain control of the CPU so that it can switch between *processes*?
  - A cooperative Approach: **Wait for system calls**
  - A Non-Cooperative Approach: **The OS takes control**

A cooperative Approach: Wait for system calls

- Processes periodically give up the CPU by making **system calls** such as `yield`.
  - The OS decides to run some other task.
  - Application also transfer control to the OS when they do something illegal.
    - Divide by zero
    - Try to access memory that it shouldn't be able to access
  - e.g Early versions of the Macintosh OS, The old Xerox Alto system

A process gets stuck in an infinite loop.
→ Reboot the machine

A Non-Cooperative Approach: OS Takes Control

- **A timer interrupt**
  - During the boot sequence, the OS start the timer.
  - The timer raises an interrupt every so many milliseconds.
  - When the interrupt is raised :
    - The currently running process is halted.
    - Save enough of the state of the program
    - A pre-configured interrupt handler in the OS runs.

A timer interrupt gives OS the ability to run again on a CPU.

Saving and Restoring Context

- Scheduler makes a decision:
  - Whether to continue running the **current process**, or switch to a **different one**.
  - If the decision is made to switch, the OS executes <u>context switch</u>.

Context Switch

- A low-level piece of assembly code
  - **Save the values of necessary registers** for the current process onto its kernel stack
    - General purpose registers
    - PC
    - kernel stack pointer
  - **Restore the register values** for the soon-to-be-executing process from its kernel stack
  - **Switch to the kernel stack** for the soon-to-be-executing process

Limited Direction Execution Protocol (Timer interrupt)

OS @ boot (kernel mode)	Hardware
initialize trap table	
	remember address of … syscall handler timer handler
start interrupt timer	
	start timer interrupt CPU in X ms

OS @ run (kernel mode)	Hardware	Program (user mode)
		Process A …
	timer interrupt save regs(A) to k-stack(A) move to kernel mode jump to trap handler	

Limited Direction Execution Protocol (Timer interrupt)

OS @ run (kernel mode)	**Hardware**	**Program (user mode)**
	(Cont.)	

Handle the trap
Call switch() routine
  save regs(A) to proc-struct(A)
  restore regs(B) from proc-struct(B)
  switch to k-stack(B)
return-from-trap (into B)		

	restore regs(B) from k-stack(B) move to user mode jump to B's PC	
		Process B …

The xv6 Context Switch Code (swtch.S)

```
# Context switch (swtch.S)
#
#   void swtch(struct context **old, struct context *new);
#
# Save the current registers on the stack, creating
# a struct context, and save its address in *old.
# Switch stacks to new and pop previously-saved registers.

.globl swtch
swtch:
  movl 4(%esp), %eax          // Set %eax to contain the old context
  movl 8(%esp), %edx          // Set %edx  to contain the new context

  # Save old callee-saved registers
  pushl %ebp                  // Save %ebp onto the old stack
  pushl %ebx                  // Save %ebx onto the old stack
  pushl %esi                  // Save %esi onto the old stack
  pushl %edi                  // Save %edi onto the old stack

  # Switch stacks
  movl %esp, (%eax)           // Copy %esp to the old context
  movl %edx, %esp             // Set the next context to %esp

  # Load new callee-saved registers
  popl %edi                   //  Set %edi with the new stack (pop)
  popl %esi                   //  Set %esi with the new stack (pop)
  popl %ebx                   //  Set %ebx with the new stack (pop)
  popl %ebp                   //  Set %ebp with the new stack (pop)
  ret                         //  Set %eip with the new stack (ret)
```

Concurrency Problems?

- What happens if, during interrupt or trap handling, another interrupt occurs?
- OS handles these situations:
  - **Disable interrupts** during interrupt processing
  - Use a number of sophisticate **locking** schemes to protect concurrent access to internal data structures.

Summary

- Virtualize CPU
  - The abstraction of process – Process in OS kernel
    - Process creation process, Process state, Process data structure in OS kernel
  - Limited Direct Execution
    - User/kernel mode, System call (the interface between user/kernel), System call working process, and Process switch
- Next: CPU Scheduling
  - Chapter 7 (Scheduling), Chapter 8 (Multi-level Feedback Queue), Chapter 9 (Proportional Share), Chapter 10 (Multi-CPU Scheduling)