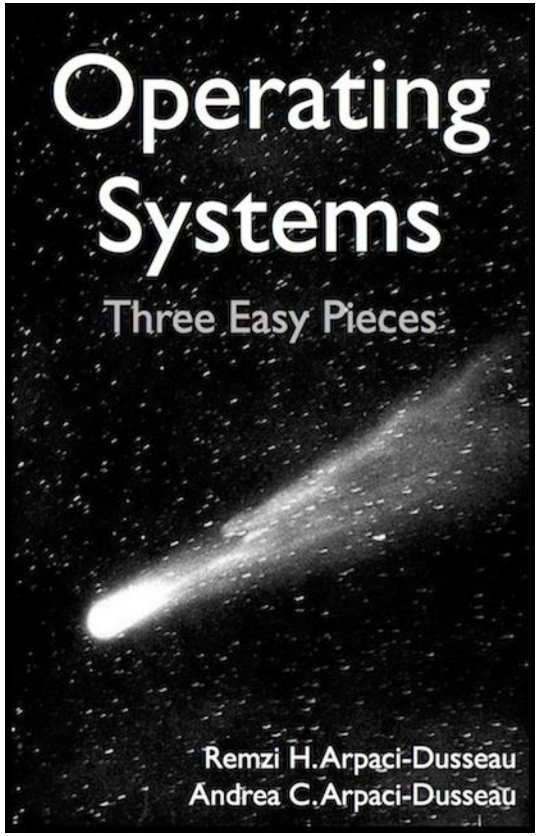
CMSC 125: Operating Systems

- Instructor: **Joseph Anthony C. Hermocilla**
- Email: jchermocilla@up.edu.ph
- Web: https://jachermocilla.org

Resources

Book: https://pages.cs.wisc.edu/~remzi/OSTEP/

Slides Template:
https://pages.cs.wisc.edu/~remzi/OSTEP/Educators-Slides/Youjip/



Acknowledgement

- This lecture slide set was initially developed for Operating System course in Computer Science Dept. at Hanyang University. This lecture slide set is for OSTEP book written by Remzi and Andrea at University of Wisconsin.

4. The Abstraction: The Process

Operating System: Three Easy Pieces

How to provide the illusion of many CPUs?

- Users want to run many programs but we only have a limited of CPUs (ex. 8 cores)
- CPU virtualization
  - The OS can promote the illusion that many virtual CPUs exist
    - How? By juggling programs
  - **Time sharing**: Running one process, then stopping it and running another
    - The potential cost is performance.
    - Via a **mechanism** called **context-switch**

http://clipart-library.com/clipart/203581.htm

Mechanisms and Policies.. Revisited

- **Mechanism**
  - Low-level methods or protocols that implement a functionality
  - “how?”
  - Example: context-switch
- **Policy**
  - Algorithms for making some kinds of decision within the OS
  - “which?”
  - Example: scheduling policy - deciding which program to run first?next?

A Process

A process is a running program.

- A **program** itself is "lifeless" – just sits on the disk as bunch of instructions (and some data) aka **program image**
- We can **characterize** a process by looking at the different data structures and systems resources it uses or accesses
- What comprises a process?
  - Memory (address space – range of addresses that a process can access)
    - Instructions/Code
    - Data
  - Machine state (via Registers)
    - Program counter (Instruction pointer)
    - Stack pointer
    - Frame pointer
  - I/O information – list of open files

Process API

- These APIs are available on any modern OS
  - **Create**
    - Create a new process to run a program – `fork(), exec()`
  - **Destroy**
    - Halt a runaway process – `kill(pid, SIGTERM)`
  - **Wait**
    - Wait for a process to stop running – `wait(pid)`
  - **Miscellaneous Control**
    - Some kind of method to suspend a process and then resume it – `kill(pid, SIGSTOP), kill(pid, SIGCONT)`
  - **Status**
    - Get some status info about a process – `cat /proc/pid/status`

Process Creation

1. **Load** a program code(and statc data) into <u>memory</u>, into the address space of the process.
    - Programs initially reside on disk in *executable format*. (ELF, PE)
    - OS perform the loading process lazily.
        - Loading pieces of code or data only as they are needed during program execution. (some programs a big)

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

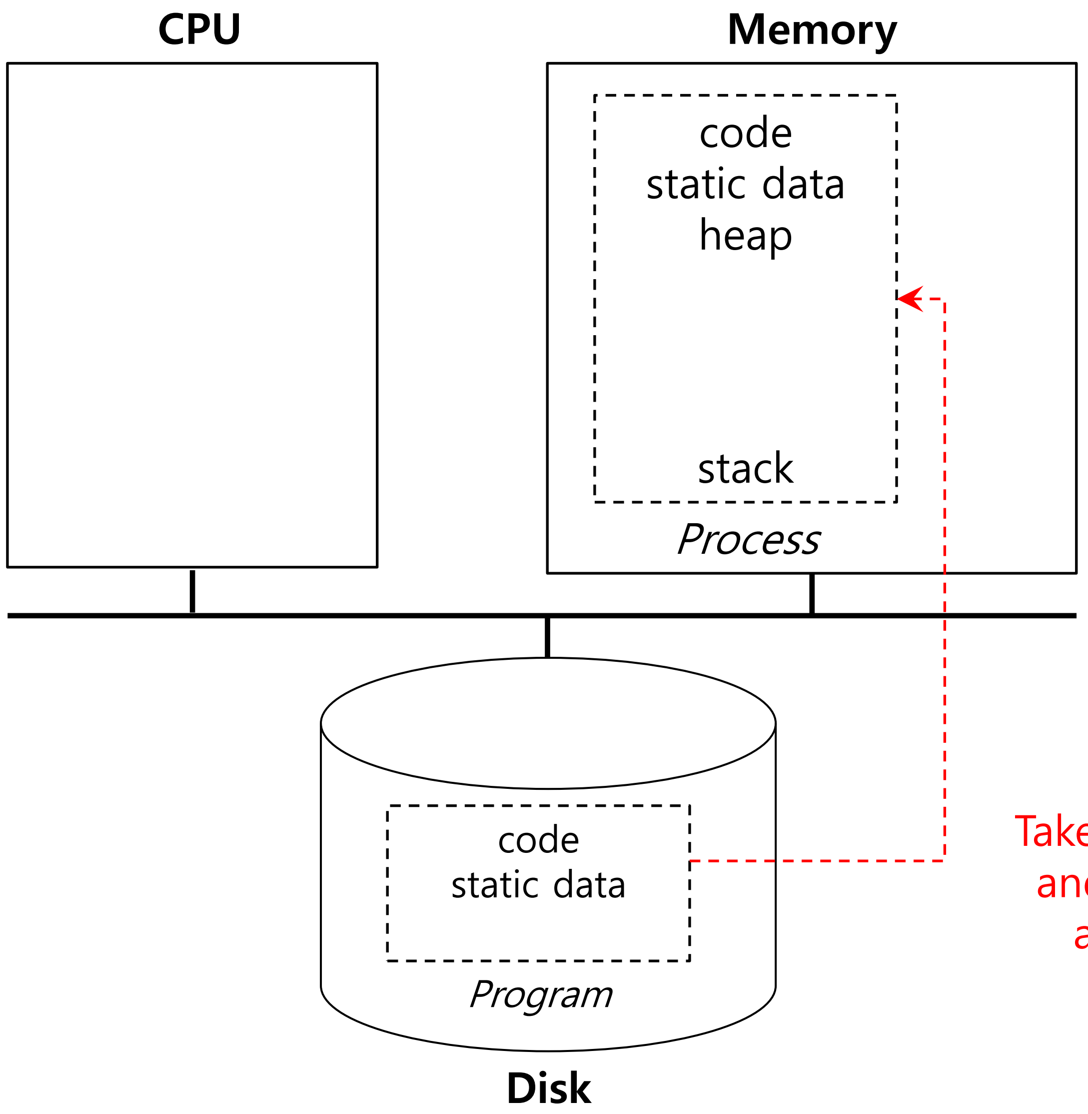
Process Creation (Cont.)

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 (STDIN), standard output(STDOUT), and standard error(STDERR)

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

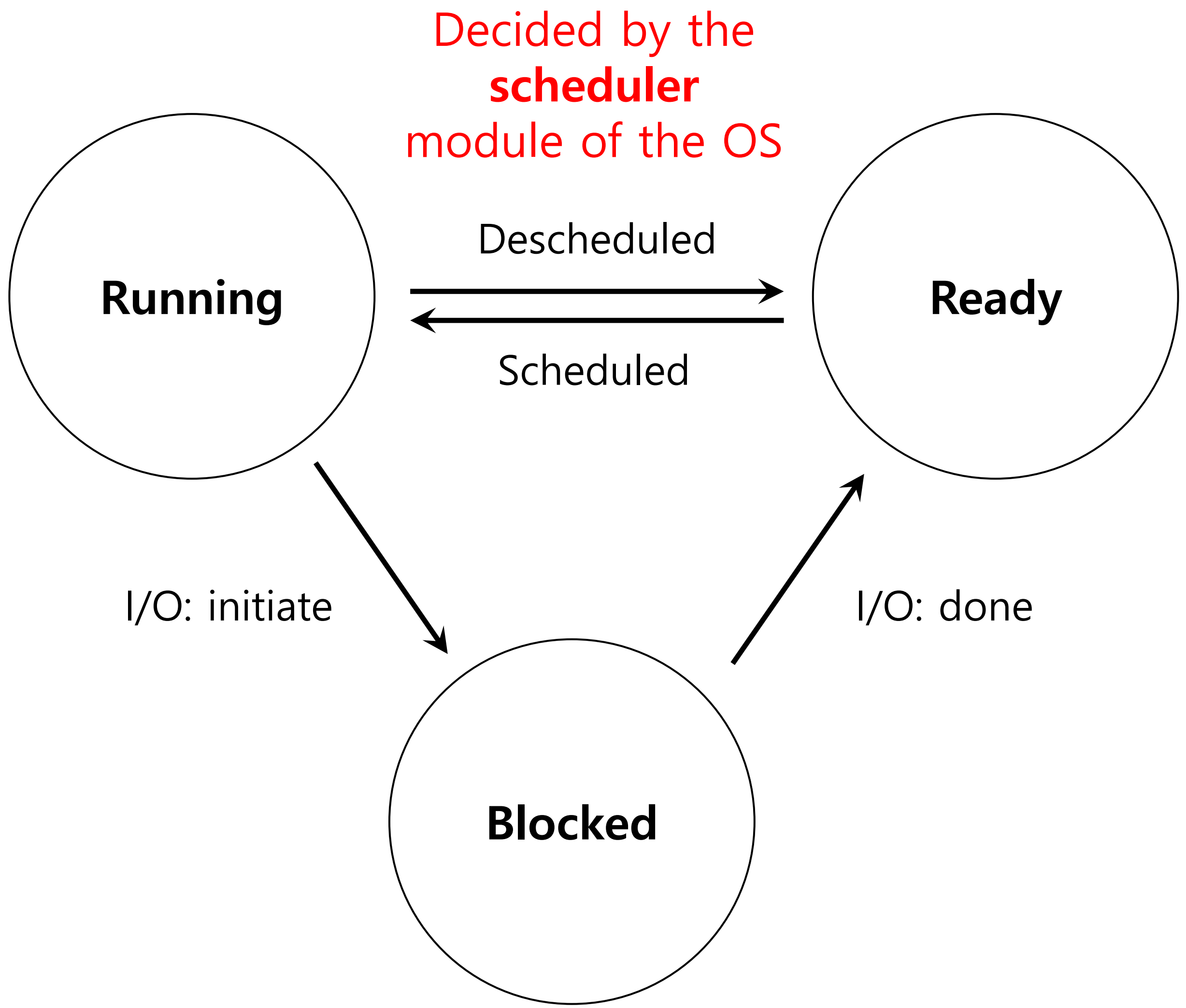


Loading:
Takes on-disk program and reads it into the address space of process

Process States

- A process can be in one of three states.
  - **Running**
    - A process is running on a processor. Instructions are being executed
  - **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 and is waiting for the result
    - Example: When a process initiates an I/O request to a disk, it becomes blocked and thus some other process can use the processor
- Additional states may be present depending on the OS
  - **Initial**
  - **Final/Terminated**

Process State Transition



Tracing Process States: CPU Only

- Assumes a single processor is available

Time	$Process_0$	$Process_1$	Notes
1	Running	Ready	
2	Running	Ready	
3	Running	Ready	
4	Running	Ready	$Process_0$ now done
5	–	Running	
6	–	Running	
7	–	Running	
8	–	Running	$Process_1$ now done

Tracing Process States: CPU and I/O

- Assumes a single processor is available

Time	$Process_0$	$Process_1$	Notes
1	Running	Ready	
2	Running	Ready	
3	Running	Ready	$Process_0$ initiates I/O
4	Blocked	Running	$Process_0$ is blocked,
5	Blocked	Running	so $Process_1$ runs
6	Blocked	Running	
7	Ready	Running	I/O done
8	Ready	Running	$Process_1$ now done
9	Running	–	
10	Running	–	$Process_0$ now done

Data Structures

- The OS has some key data structures that track various relevant pieces of information.
  - **Process/Tasks lists**
    - Ready processes
    - Blocked processes
    - Current running process
  - **Register context**
    - Will hold the values of the registers(in memory) when a process is stopped to be able to resume later
    - This is done by copying these values back to the actual registers during a **context-switch**
- PCB(Process Control Block)
  - A C-structure that contains information about a process.

Example: The xv6 kernel register context structure and process states definitions

```
// the registers xv6 will save and restore
// to stop and subsequently restart a process
struct context {
    int eip;    // Index pointer register
    int esp;    // Stack pointer register
    int ebx;    // Called the base register
    int ecx;    // Called the counter register
    int edx;    // Called the data register
    int esi;    // Source index register
    int edi;    // Destination index register
    int ebp;    // Stack base pointer register
};

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

Example: The xv6 kernel proc structure definition

```
// 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
};
```