Engineering & Computer Science
Syracuse University

CIS 657 – Principles of Operating Systems

Topic: Process

Endadul Hoque

Acknowledgement

- Youjip Won (Hanyang University)
- OSTEP book – by Remzi and Andrea Arpaci-Dusseau (University of Wisconsin)

Revisit: An Operating System (OS)

- Responsible for
  - Making it easy to **run** programs
  - Allowing programs to **share** memory
  - Enabling programs to **interact** with devices

OS is in charge of making sure the system operates **correctly** and **efficiently**.

How does OS ensure this?

Revisit: Virtualization

- The OS takes a physical resource and transforms it into a virtual form of itself.
  - **Physical resource**: Processor, Memory, Disk …
- The virtual form is more general, powerful and easy-to-use.
- Sometimes, we refer to the OS as a **virtual machine**.

Revisit: Virtualizing the CPU

- The system has a very large number of virtual CPUs.
  - Turning a single CPU into a <u>seemingly infinite number</u> of CPUs.
  - Allowing many programs to <u>seemingly run at once</u> → **Virtualizing the CPU**

How to provide the illusion of many CPUs?

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

Space sharing: the counterpart (e.g., disk space)

How to implement virtualization of the CPU?

OS needs

- Some low-level machinery (**mechanisms**)
- Some high-level intelligence (**policies**)

Policy vs. Mechanism

- In many OSes, a common design paradigm
  - Separate *high-level policies* from *low-level mechanisms*
- **Policy**: provides answer to which/what question
  - E.g., which process should the OS run now?
  - E.g., only Alice should read file ***f***
- **Mechanism**: provides answer to how question
  - E.g., how does OS perform a context switch?
  - E.g., encrypt file ***f***
- A general software design principle: **modularity**

A Process

A process is a running program

- Comprising of a process (aka **machine state**)
  - Memory (address space)
    - Instructions
    - Data section
  - Registers
    - Program Counter (PC) or Instruction Pointer (IP)
    - Stack pointer w/ frame pointer
  - I/O information
    - E.g., open files

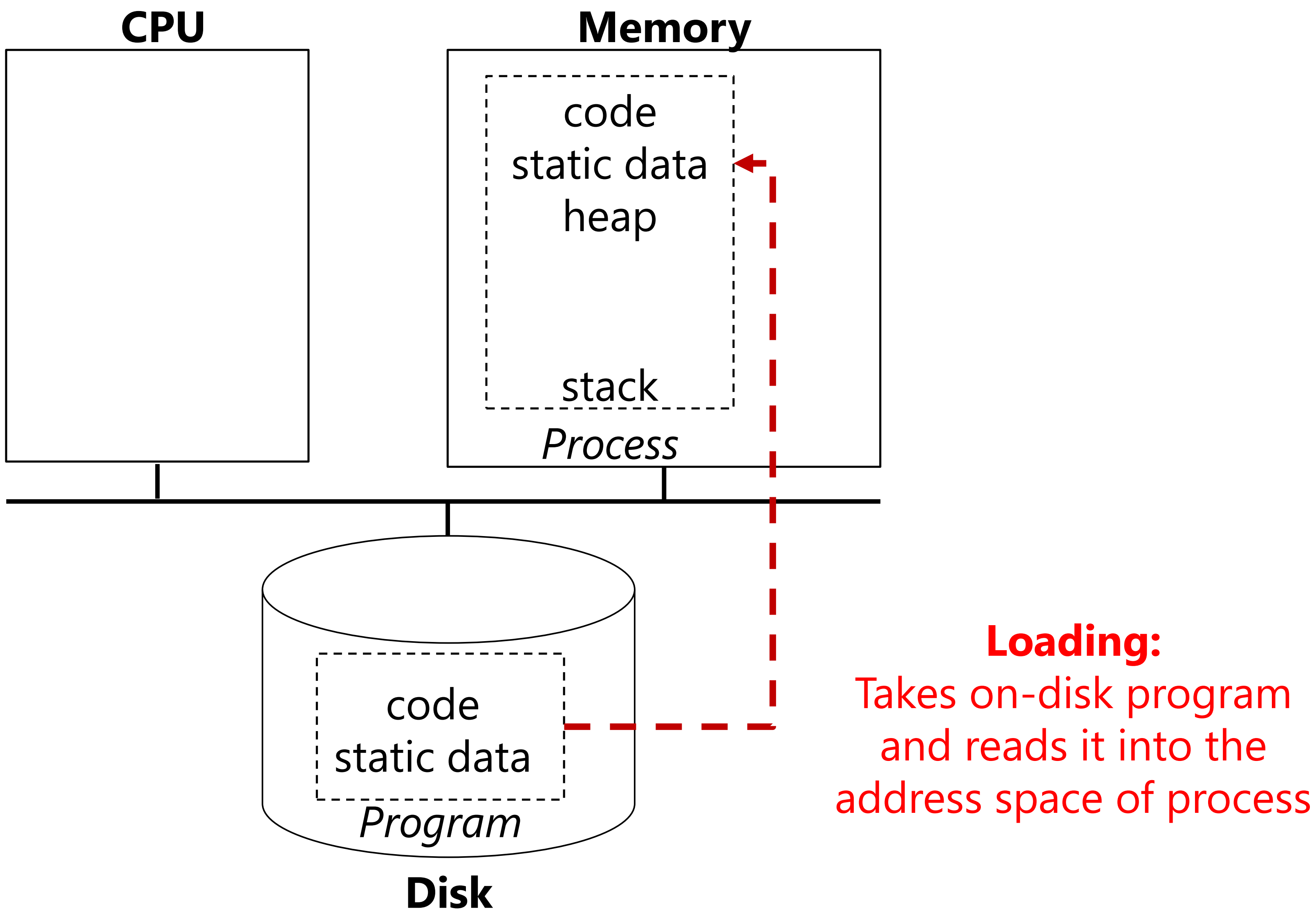
Analogy based on Java

Class $\mapsto$ Object
Program $\mapsto$ Process

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



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

3. The program's **heap** is created.
   - Used for explicitly requested dynamically allocated data.
   - Use `malloc()` for requesting space and `free()` to release it

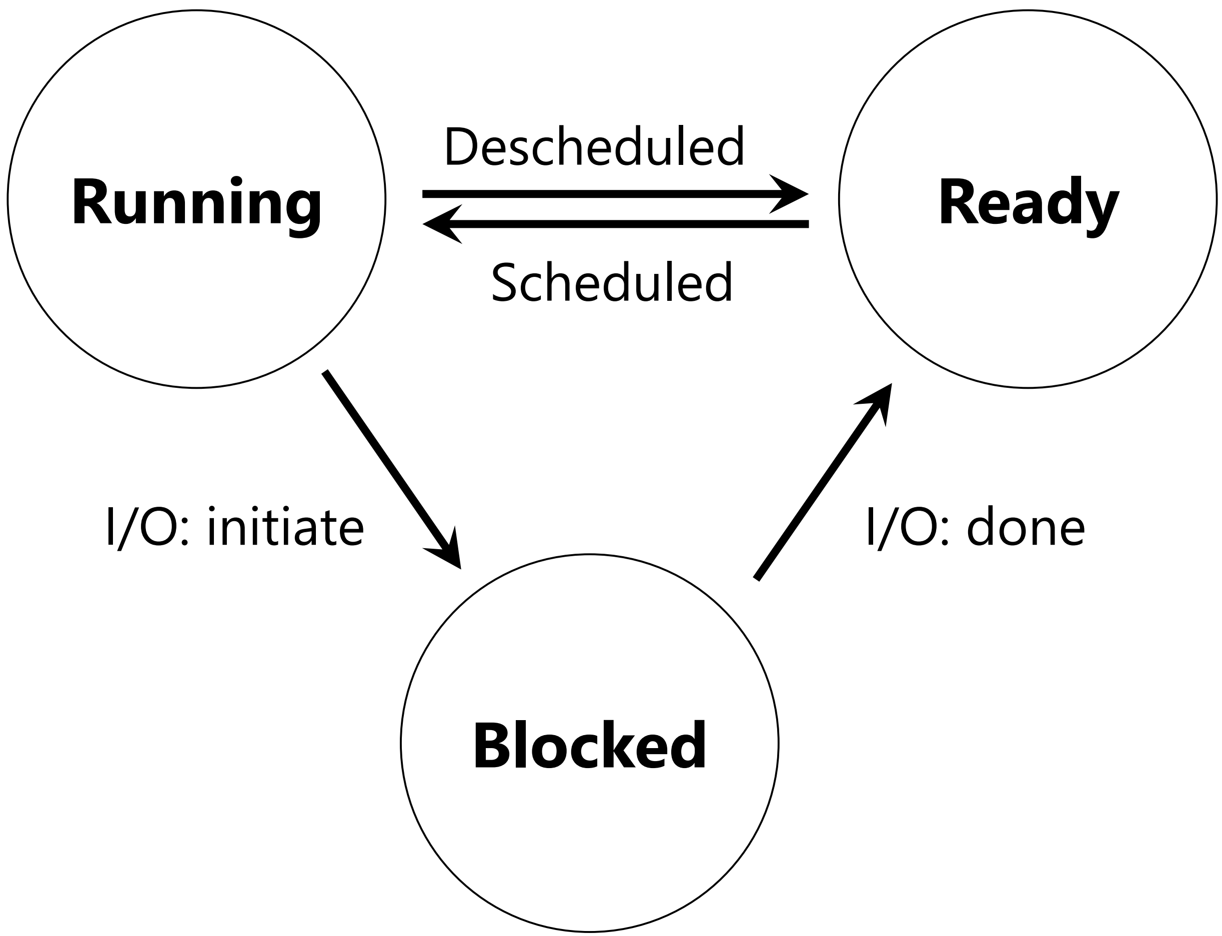
4. The OS does some other initialization tasks.
   - input/output (I/O) setup
   - Each process by default has **three** open file descriptors: Standard input (`stdin`), output (`stdout`) and error (`stderr`)

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

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



Tracing Process State

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

Figure 4.3: **Tracing Process State: CPU Only**

Tracing Process State

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

Figure 4.4: **Tracing Process State: CPU and I/O**

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)
  - A C-structure that contains information about each process.

Example: The xv6 kernel Proc Structure

```
// 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

```
// 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
};
```

UNIX/LINUX PROCESS API

Read The Man Pages

- **Manual** (Man) pages are the original form of documentation that exist on UNIX systems

- Spend some time reading man pages
  - A key step in becoming a systems programmer
  - There are tons of **useful tidbits** hidden there

- Reading them can <u>save you some embarrassment</u> (in professional settings) and <u>debugging</u> time

fork()

- Create a new process
  - The newly-created process has its own copy of the **address space**, **registers**, and **PC**.

p1.c

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

int main(int argc, char *argv[]){
    printf("hello world (pid:%d)\n", (int) getpid());
    int rc = fork();
    if (rc < 0) {       // fork failed; exit
        fprintf(stderr, "fork failed\n");
        exit(1);
    } else if (rc == 0) { // child (new process)
        printf("hello, I am child (pid:%d)\n", (int) getpid());
    } else {            // parent goes down this path (main)
        printf("hello, I am parent of %d (pid:%d)\n",
        rc, (int) getpid());
    }
    return 0;
}
```

fork()

Result (Not deterministic)

```
prompt> ./p1
hello world (pid:29146)
hello, I am parent of 29147 (pid:29146)
hello, I am child (pid:29147)
prompt>
```

OR

```
prompt> ./p1
hello world (pid:29146)
hello, I am child (pid:29147)
hello, I am parent of 29147 (pid:29146)
prompt>
```

wait()

- This system call won't return until the child has run and exited

p2.c

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/wait.h>
int main(int argc, char *argv[]){
    printf("hello world (pid:%d)\n", (int) getpid());
    int rc = fork();
    if (rc < 0) {        // fork failed; exit
        fprintf(stderr, "fork failed\n");
        exit(1);
    } else if (rc == 0) { // child (new process)
        printf("hello, I am child (pid:%d)\n", (int) getpid());
    } else {             // parent goes down this path (main)
        int wc = wait(NULL);
        printf("hello, I am parent of %d (wc:%d) (pid:%d)\n",
        rc, wc, (int) getpid());
    }
    return 0;
}
```

wait()

Result (Deterministic)

```
prompt> ./p2
hello world (pid:29266)
hello, I am child (pid:29267)
hello, I am parent of 29267 (wc:29267) (pid:29266)
prompt>
```

More on wait()

- `wait()` vs `waitpid()`
- When to use `waitpid()`?
- Some cases where `wait()` returns before the child exits
  - What is a process group? How is this related to `wait()`?
- Relationship between `wait()` and child processes running in background?

Read the man pages

exec()

- Run a program that is different from the calling program

p3.c

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

int main(int argc, char *argv[]){
    printf("hello world (pid:%d)\n", (int) getpid());
    int rc = fork();
    if (rc < 0) {              // fork failed; exit
        fprintf(stderr, "fork failed\n");
        exit(1);
    } else if (rc == 0) {      // child (new process)
        printf("hello, I am child (pid:%d)\n", (int) getpid());
        char *myargs[3];
        myargs[0] = strdup("wc"); // program: "wc" (word count)
        myargs[1] = strdup("p3.c"); // argument: file to count
        myargs[2] = NULL;           // marks end of array
        …
```

exec()

- Run a program that is different from the calling program

p3.c

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

```
…
        execvp(myargs[0], myargs); // runs word count
        printf("this shouldn't print out");
    } else {  // parent goes down this path (main)
        int wc = wait(NULL);
        printf("hello, I am parent of %d (wc:%d) (pid:%d)\n",
            rc, wc, (int) getpid());
    }
    return 0;
}
```

```
        char *myargs[3];
        myargs[0] = strdup("wc"); // program: "wc" (word count)
        myargs[1] = strdup("p3.c"); // argument: file to count
        myargs[2] = NULL;           // marks end of array
        …
```

exec()

```
        printf("hello, I am child (pid:%d)\n", (int) getpid());
        ... ...
        execvp(myargs[0], myargs); // runs word count
        printf("this shouldn't print out");
    } else {   // parent goes down this path (main)
        int wc = wait(NULL);
        printf("hello, I am parent of %d (wc:%d) (pid:%d)\n",
            rc, wc, (int) getpid());
    }
    return 0;
}
```

Excerpt from p3.c

Result

```
prompt> ./p3
hello world (pid:29383)
hello, I am child (pid:29384)
29 107 1030 p3.c
hello, I am parent of 29384 (wc:29384) (pid:29383)
prompt>
```

exec() family

- On Linux, there are six variants of `exec`()
  - `execl(), execlp(), execle(), execv(), execvp(), execvpe()`

Read the man pages

A pictorial presentation:
https://gist.github.com/fffaraz/8a250f896a2297db06c4

I/O Redirection

p4.c

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

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);
        …
```

I/O Redirection

p4.c

```
        …
        // 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 (main)
        int wc = wait(NULL);
    }
    return 0;
}
```

```
    } 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);
        …
```

I/O Redirection

p4.c

```
…
```

```
prompt> ./p4
prompt> cat p4.output
32 109 846 p4.c
prompt>
```

Result

```
        int wc = wait(NULL);
    }
    return 0;
}
```

```
} 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);
    …
```

Other Process API

- `pipe()`
  - E.g., `grep -o foo file | wc -l`
  - the output of one process (i.e., `grep`) is connected to an in-kernel pipe (i.e., queue), and the input of another process (i.e., `wc`) is connected to that same pipe
- `kill()`
  - send signals to a process (e.g., pause, die, terminate and so on)
  - Control-c sends a `SIGINT` (interrupt) to the process to normally terminate it
  - Control-z sends a `SIGTSTP` (stop) to the process to pause it

Can a user send signals (e.g., `SIGINT`) to any arbitrary process?

Read the man pages

Some Useful Shell Commands

- `ps`
  - Display which processes are running
- `top`
  - Display the processes of the system, the resources and their status
- `kill`/`killall`
  - Send different signals to processes

Read the man pages

Reading Material

- **Chapter 4, 5** of OSTEP book – by Remzi and Andrea Arpaci-Dusseau (University of Wisconsin)
  http://pages.cs.wisc.edu/~remzi/OSTEP/cpu-intro.pdf
  http://pages.cs.wisc.edu/~remzi/OSTEP/cpu-api.pdf

Questions?