# Operating Systems
# CMPSCI 377
# Spring 2020

## Clicker Question

How many peaches do we need to make everyone happy?

(A) 0

(B) 1

(C) 2^n

(D) Infinite

Answer on Next Slide

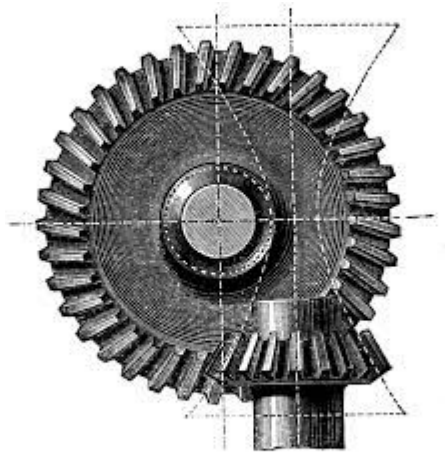
## Today's Class

- The Abstraction: The Process
- Process API

## The Crux of the Problem: How To Provide The Illusion Of Many CPUs?

Although there are only a few physical CPUs available, how can the OS provide the illusion of a nearly-endless supply of said CPUs?

# Virtualizing the CPU

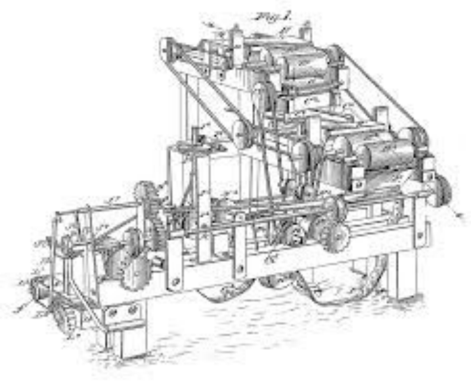


- Time Sharing
  - The OS can promote the illusion that many virtual CPUs exist.
  - The OS runs one process, stops it, then runs another, and so on.

- Implementing Virtualization
  - Low-level machinery and high-level intelligence.
  - **Mechanisms:** low-level methods implementing a needed piece of machinery.
    - Context Switch
  - **Policies:** algorithms for making decisions within the OS
    - Which program should the OS run?
    - Scheduling Policies
    - Historical information, workload knowledge, and performance metrics

## The Abstraction: A Process

- A Process
  - **An instance of a running program.**
    At any moment in time, we can summarize a process by taking an inventory of the different pieces of the system it <u>accesses</u> or <u>effects</u> *dur* *execution*.



- Machine State
  - What parts of the machine are important to the execution of the program.
  - What can a program read or update when it is running?
    - **Memory:** data & instructions, also known as its <u>address space</u>.
    - **Registers:** program counter (PC), stack pointer, frame pointer, etc.
    - **Persistent Storage:** I/O information such as open files.

## Process API

- Create
- Destroy
- Wait
- Miscellaneous Control
- Status

## Process API

- **Create:** An operating system must include some method to create new processes. When you type a command into the shell, or double-click on an application icon, the OS is invoked to create a new process to run the program you have indicated.
- Destroy
- Wait
- Miscellaneous Control
- Status

## Process API

- Create
- **Destroy:** As there is an interface for process creation, systems also provide an interface to destroy processes forcefully. Of course, many processes will run and just exit by themselves when complete; when they don’t, however, the user may wish to kill them, and thus an interface to halt a runaway process is quite useful.
- Wait
- Miscellaneous Control
- Status

## Process API

- Create
- Destroy
- **Wait:** Sometimes it is useful to wait for a process to stop running; thus some kind of waiting interface is often provided.
- Miscellaneous Control
- Status

## Process API

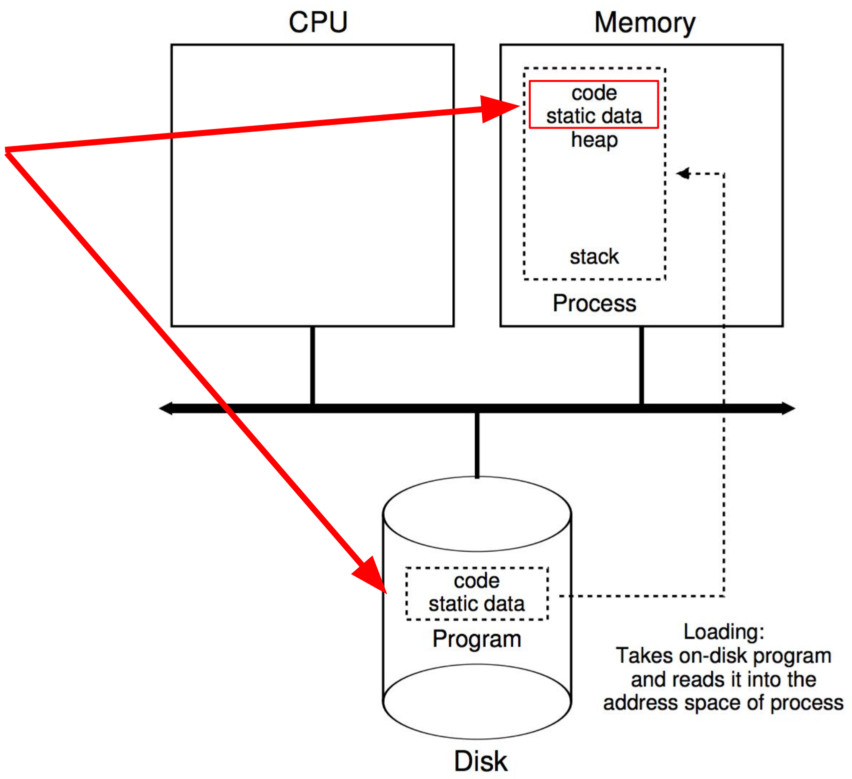
- Create
- Destroy
- Wait
- **Miscellaneous Control:** Other than killing or waiting for a process, there are sometimes other controls that are possible. For example, most operating systems provide some kind of method to suspend a process (stop it from running for a while) and then resume it (continue it running).
- Status

## Process API

- Create
- Destroy
- Wait
- Miscellaneous Control
- **Status:** There are usually interfaces to get some status information about a process as well, such as how long it has run for, or what state it is in.

## Process Creation: Loading

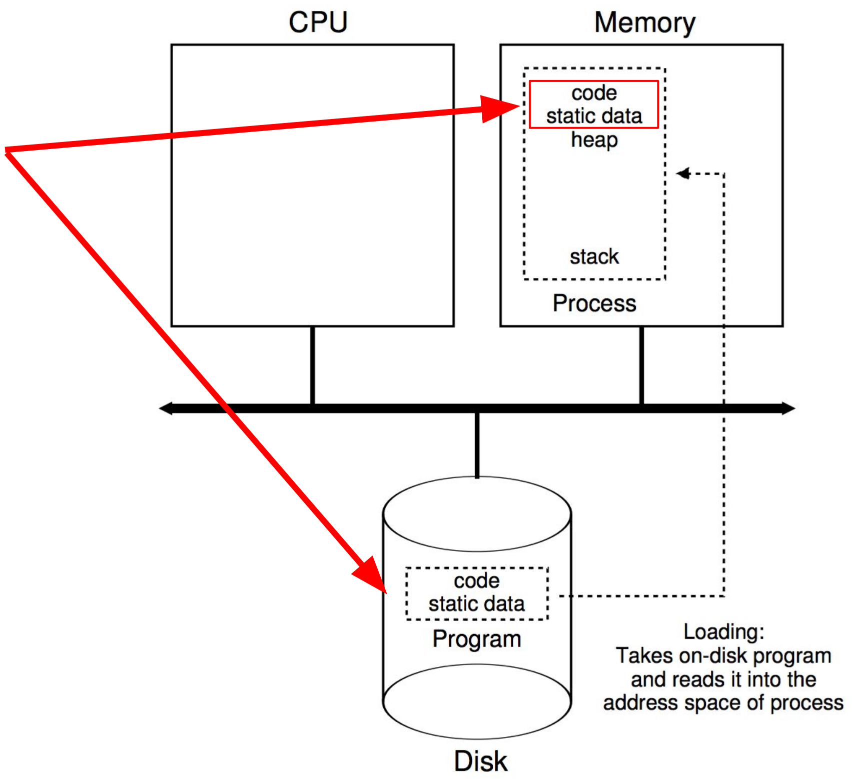
The first thing the OS must do is load its code and any static data into memory - the address space of the process.



# Process Creation: Loading

The first thing the OS must do is load its code and any static data into memory - the address space of the process.

Early operating systems performed this task eagerly, whole program.

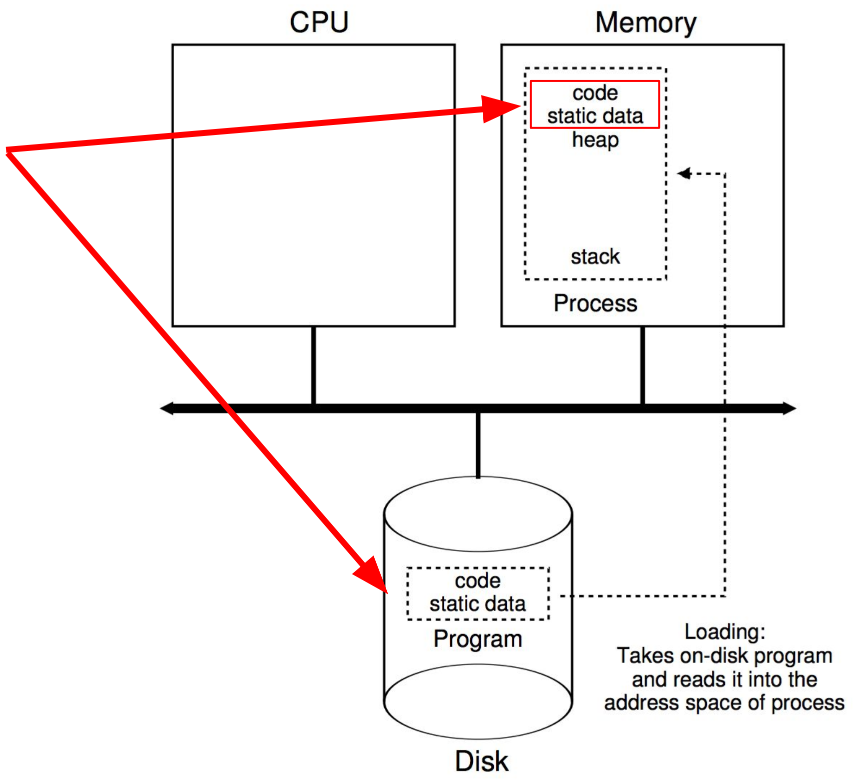


## Process Creation: Loading

The first thing the OS must do is load its code and any static data into memory - the address space of the process.

Early operating systems performed this task eagerly, whole program.

A modern OS performs this lazily, by loading pieces of code and data as they are needed.

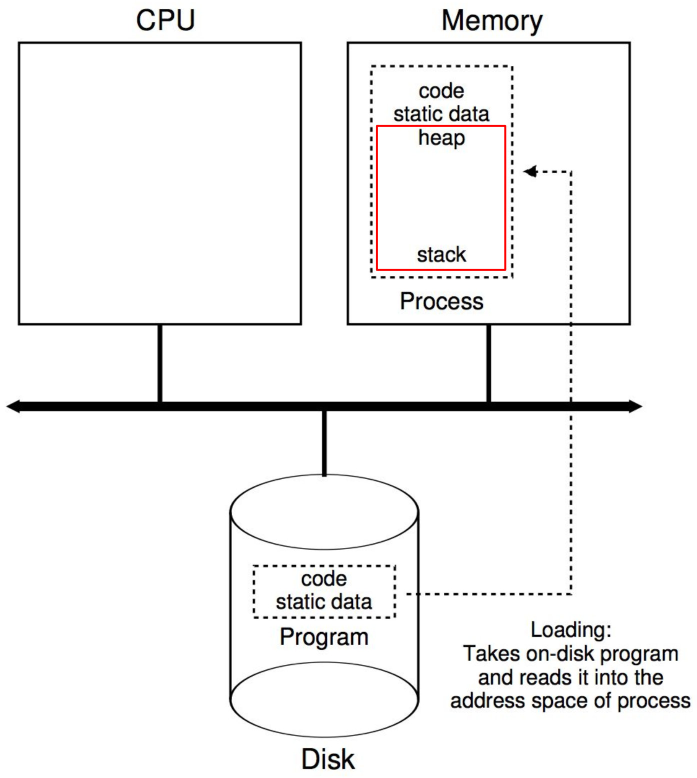


# Process Creation: Memory Allocation

The next step is to allocate important memory regions for the process.

**Stack:** for storing local variables and function parameters and return values.

**Heap:** for dynamically allocated data.

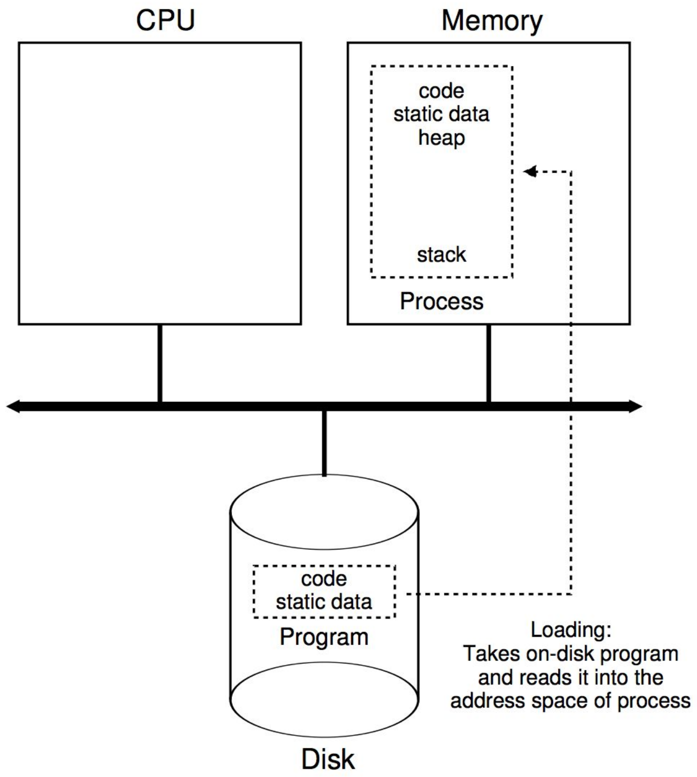


# Process Creation: Initialization

The OS will also do some other initialization tasks, particularly as related to input/output (I/O).

For example, in UNIX systems, each process by default has three open file descriptors, for standard input, output, and error

these descriptors let programs easily read input from the terminal as well as print output to the screen.

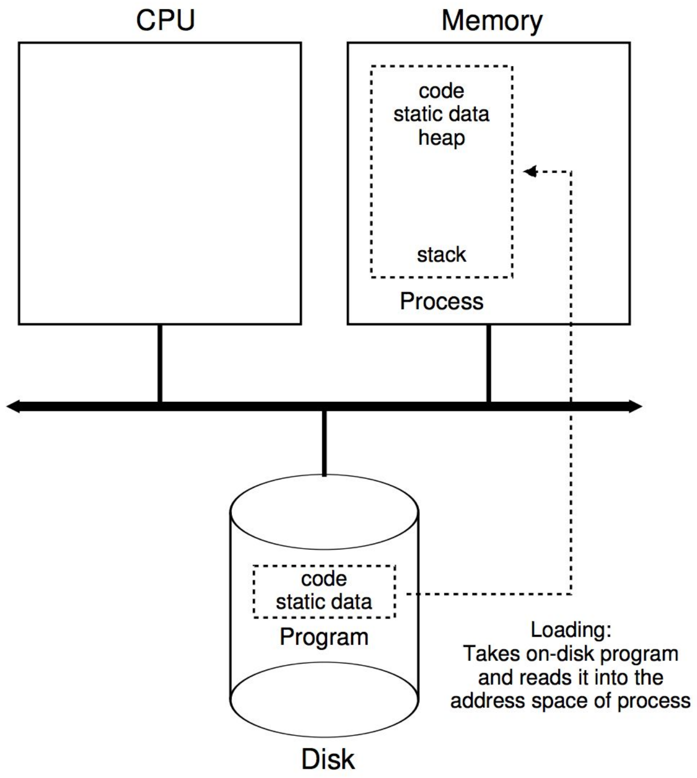


## Process Ready

The OS has now set the stage for program execution. It thus has one last task:
to start the program running at the entry point.

Historically, a special function called main().

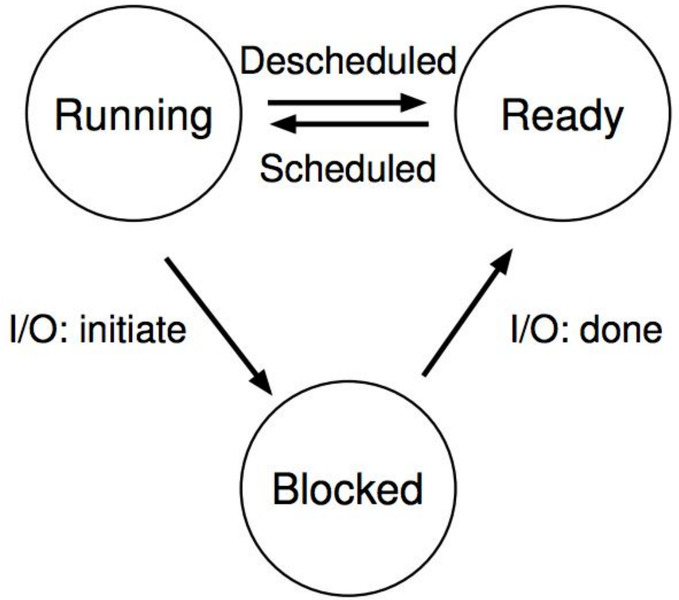
By "jumping" to the main() routine, the OS transfers control of the CPU to the newly-created process, and thus the program begins its execution.



## Process States

In a simplified view a process can be in one of three states:

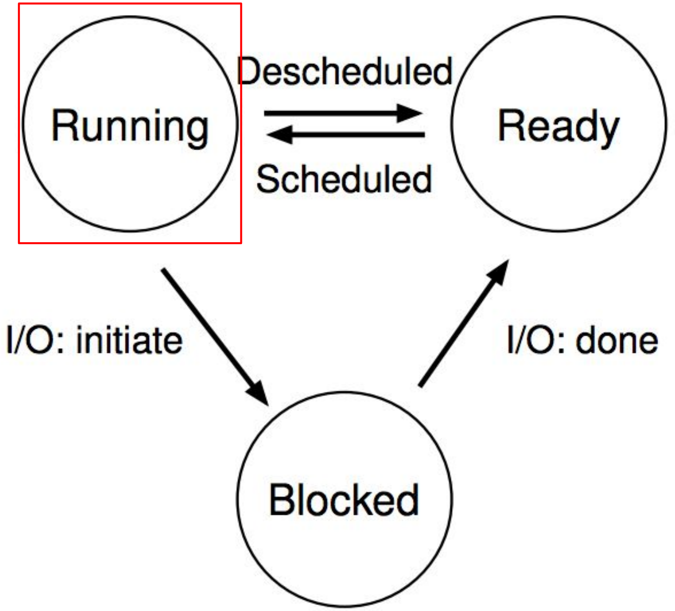
- Running
- Ready
- Blocked



## Process States

In a simplified view a process can be in one of three states:

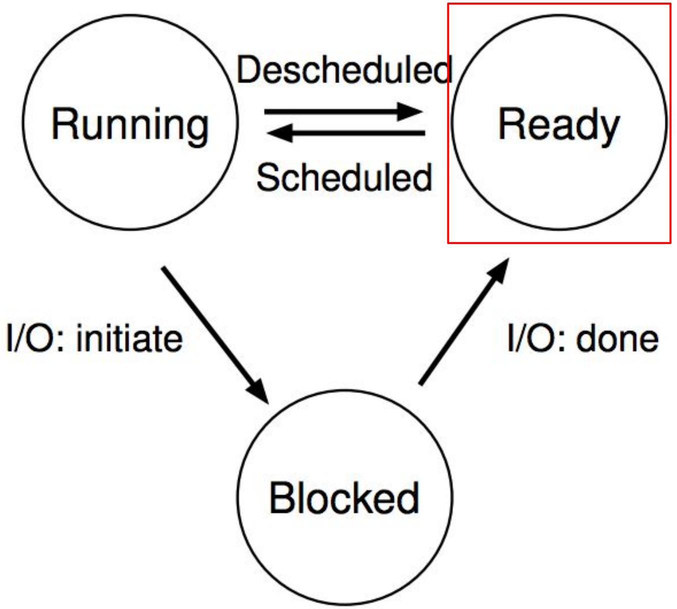
- **Running:** In the running state, a process is running on a processor. This means it is executing instructions.
- Ready
- Blocked



## Process States

In a simplified view a process can be in one of three states:

- Running
- **Ready:** In the ready state, a process is ready to run but for some reason the OS has chosen not to run it at this given moment.
- Blocked

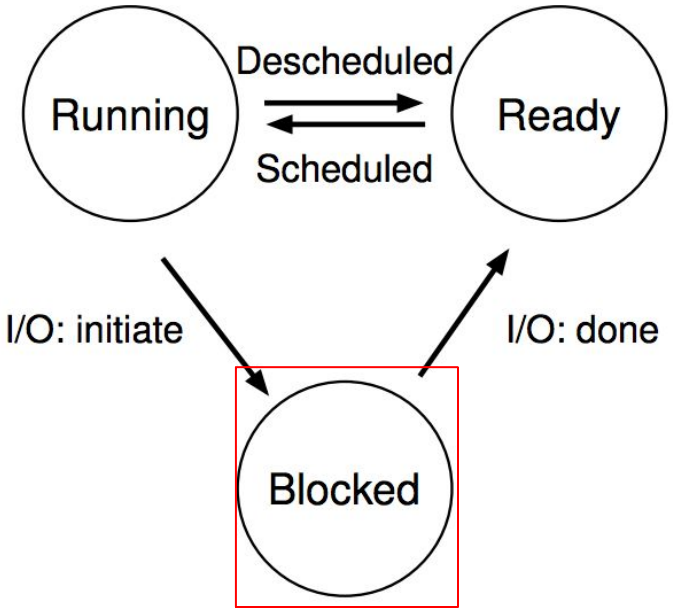


# Process States

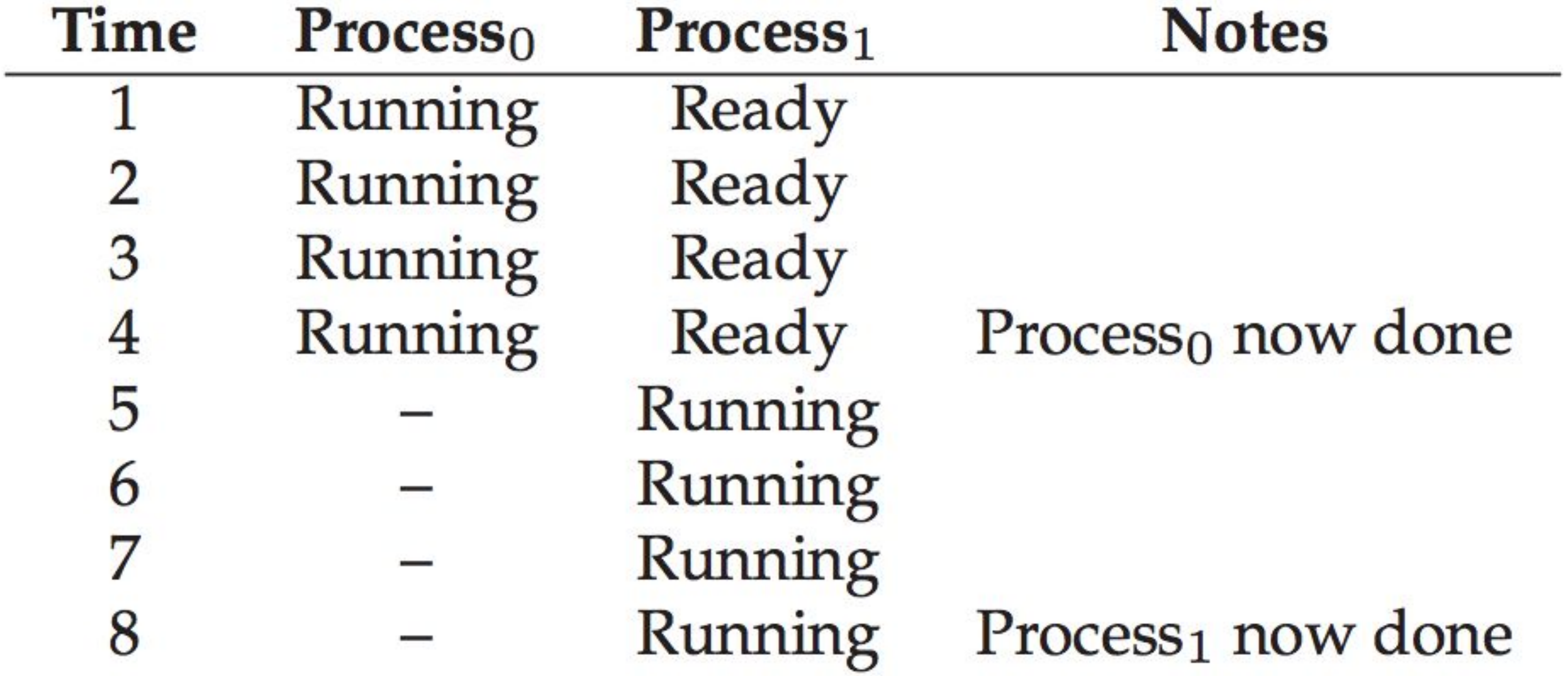
In a simplified view a process can be in one of three states:

- Running
- Ready
- **Blocked:** In the blocked state, a process has performed some kind of operation that makes it not ready to run until some other event takes place.

  Example: when a process initiates an I/O request to a disk, it becomes blocked and thus some other process can use the processor.



## Tracing Process State: CPU Only



| Time | $\text{Process}_0$ | $\text{Process}_1$ | Notes |
|---|---|---|---|
| 1 | Running | Ready | |
| 2 | Running | Ready | |
| 3 | Running | Ready | |
| 4 | Running | Ready | $\text{Process}_0$ now done |
| 5 | – | Running | |
| 6 | – | Running | |
| 7 | – | Running | |
| 8 | – | Running | $\text{Process}_1$ now done |

## Tracing Process State: CPU and I/O

| Time | $\text{Process}_0$ | $\text{Process}_1$ | Notes |
|---|---|---|---|
| 1 | Running | Ready | |
| 2 | Running | Ready | |
| 3 | Running | Ready | $\text{Process}_0$ initiates I/O |
| 4 | Blocked | Running | $\text{Process}_0$ is blocked, |
| 5 | Blocked | Running | so $\text{Process}_1$ runs |
| 6 | Blocked | Running | |
| 7 | Ready | Running | I/O done |
| 8 | Ready | Running | $\text{Process}_1$ now done |
| 9 | Running | – | |
| 10 | Running | – | $\text{Process}_0$ now done |

## Data Structures

Operating Systems are rich in data structures!

To provide an abstraction for a processes we require a data structure!

**Process Control Block:** C structure that contains information about each process.

**Process List:** Keeps track of all processes (PCBs) in the running system.

# Example: Xv6* OS Proc Structure

```
// the registers xv6 will save and restore
// to stop and subsequently restart a process
struct context {
  int eip;
  int esp;
  int ebx;
  int ecx;
  int edx;
  int esi;
  int edi;
  int ebp;
};

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

* Xv6 is a UNIX-like OS from MIT.

## Example: Xv6 OS Proc Structure

```c
// 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
};
```

**CRUX: HOW TO CREATE AND CONTROL PROCESSES**

What interfaces should the OS present for process creation and control? How should these interfaces be designed to enable ease of use as well as utility?

# fork()

```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;
}
```

## wait()

```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;
}
```

## Clicker Question

```
int main(int argc, char *argv[]){
    printf("hello world (pid:%d)\n",
           (int) getpid());
    int rc = fork();
    if (rc < 0) {
        fprintf(stderr, "fork failed\n");
        exit(1);
    } else if (rc == 0) {
        printf("child\n");
    } else {
        printf("parent\n");
        wait(NULL);
    }
    return 0;
}
```

I modified the code and flipped the two lines in red. What order does the print occur in now?

(A) child parent

(B) parent child

(C) child child

(D) Don't know

Answer on Next Slide

## Clicker Question

```
int main(int argc, char *argv[]){
    printf("hello world (pid:%d)\n",
            (int) getpid());
    int rc = fork();
    rc = fork();
    if (rc < 0) {
        fprintf(stderr, "fork failed\n");
        exit(1);
    } else if (rc == 0) {
        printf("child\n");
    } else {
        wait(NULL);
        printf("parent\n");
    }
    return 0;
}
```

I added one line of code in red, how many parents and children do you get (in some order)?

(A) One of each

(B) Two parents and Two children

(C) One Parent and Three Children

(D) Don’t know

Answer on Next Slide

# exec()

```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
        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;
}
```

## Why? Motivating the API

Of course, one big question you might have:

Why would we build such an odd interface to what should be the simple act of creating a new process?

## Why? Motivating the API

Of course, one big question you might have:

Why would we build such an odd interface to what should be the simple act of creating a new process?

Well, as it turns out, the separation of **`fork()`** and **`exec()`** is essential in building a UNIX shell.

## Why? Motivating the API

Of course, one big question you might have:

Why would we build such an odd interface to what should be the simple act of creating a new process?

Well, as it turns out, the separation of **`fork()`** and **`exec()`** is essential in building a UNIX shell.

It lets the shell run code after the call to **`fork()`** but before the call to **`exec()`**
*this code can alter the environment of the about-to-be-run program, and thus enables a variety of interesting features to be readily built.*

## Redirecting Output

```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);

        // 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 rc_wait = wait(NULL);
    }
    return 0;
}
```

## Clicker Question

```
close(STDOUT_FILENO);
open("/dev/null", O_CREAT|O_WRONLY|O_TRUNC, S_IRWXU);
int fd = open("./p4.output", O_CREAT|O_WRONLY|O_TRUNC, S_IRWXU);
```

To demonstrate this caveat, I change the code in the exec example to the above.  Where does my output go now?

(A) Don’t know

(B) The almighty bitbucket

(C) p4.output

Answer on Next Slide

## Next Time...

Chapter 6

Make sure you read!

Visit Piazza to review the course syllabus and schedule!