# Program and Process

# The fork() System Call

- 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;
}
```

# Calling fork() example (Cont.)

**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>
```

# The wait() System Call

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

p2.c

```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;
}
```

# The wait() System Call (Cont.)

**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>
```

# The exec() System Call

- Run a program that is different from the parent

**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
        …
```

# The exec() System Call (Cont.)

**p3.c (Cont.)**

```
...
        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;
}
```

**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>
```

# 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

# A Process

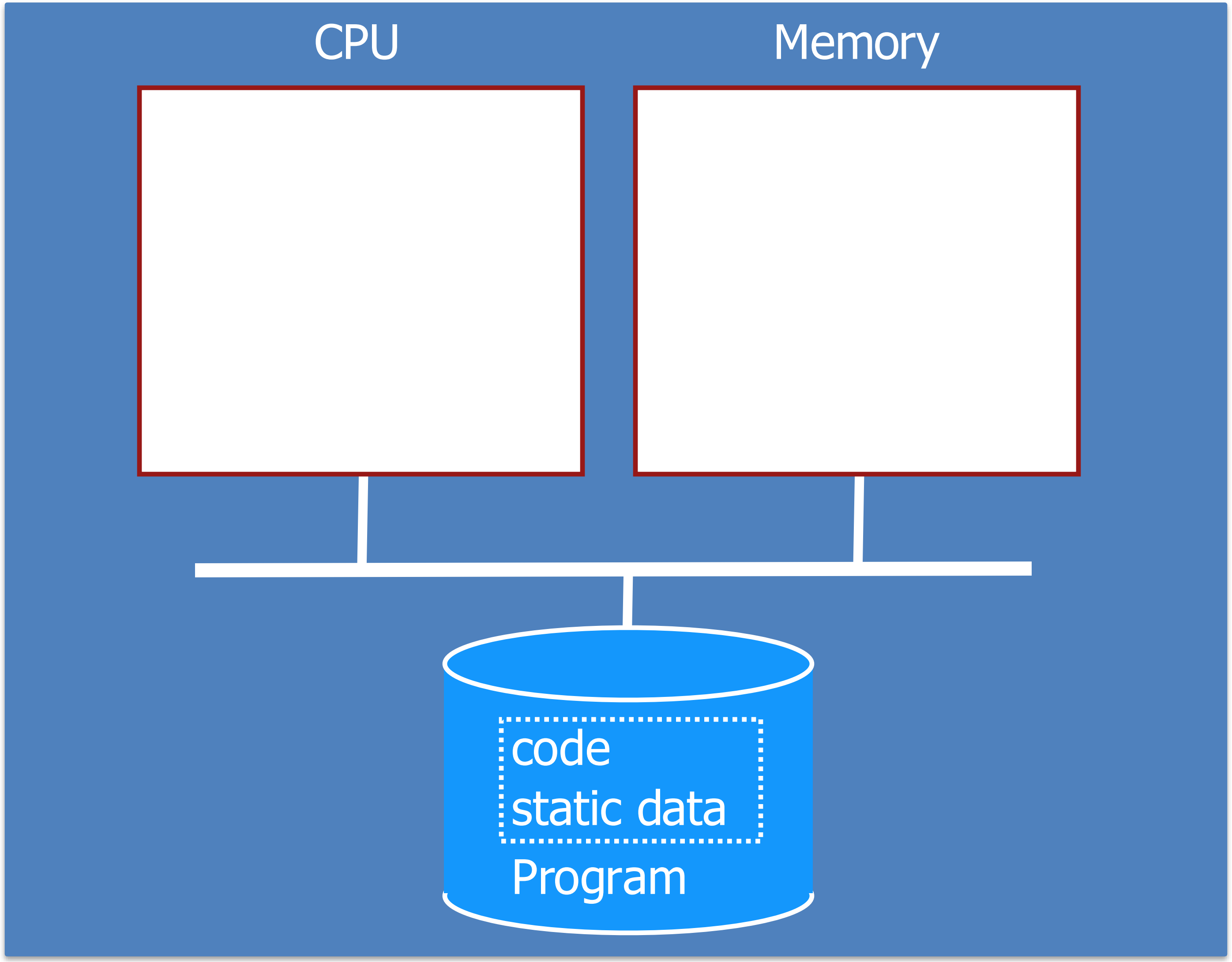
**A process is a program in execution**

- Comprising of a process:
  - Memory (address space)
    - Instructions
    - Data section
  - Registers
    - Program counter
    - Stack pointer
- A process is different than a program
  - Program: Static code and static data
  - Process: Dynamic instance of code and data
- Can have multiple process instances of same program
  - Can have multiple processes of the same program
  - Example: many users can run “ls” at the same time

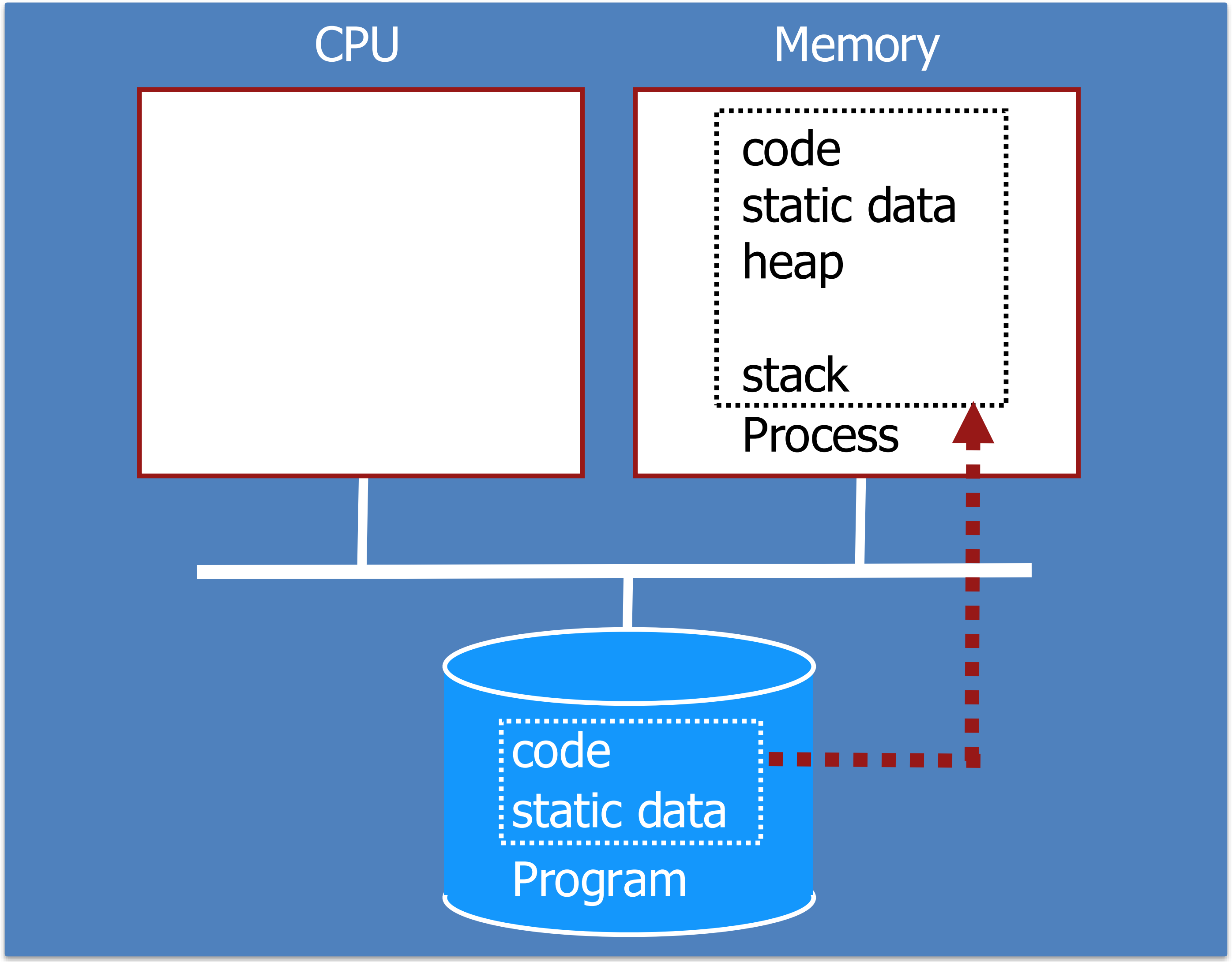
# Process API

- 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



# 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

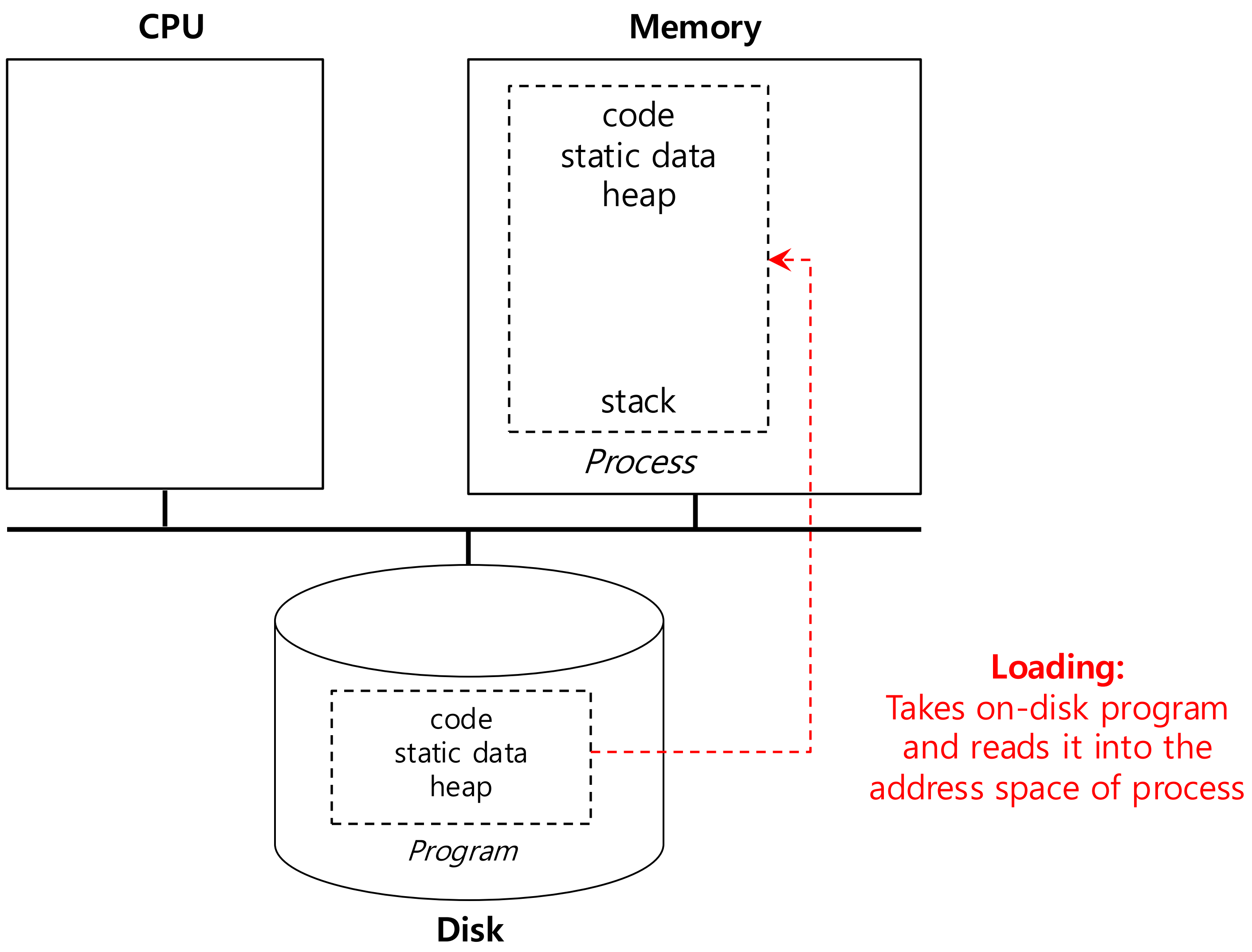
# Process Creation (Cont.)

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

4. The OS does 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

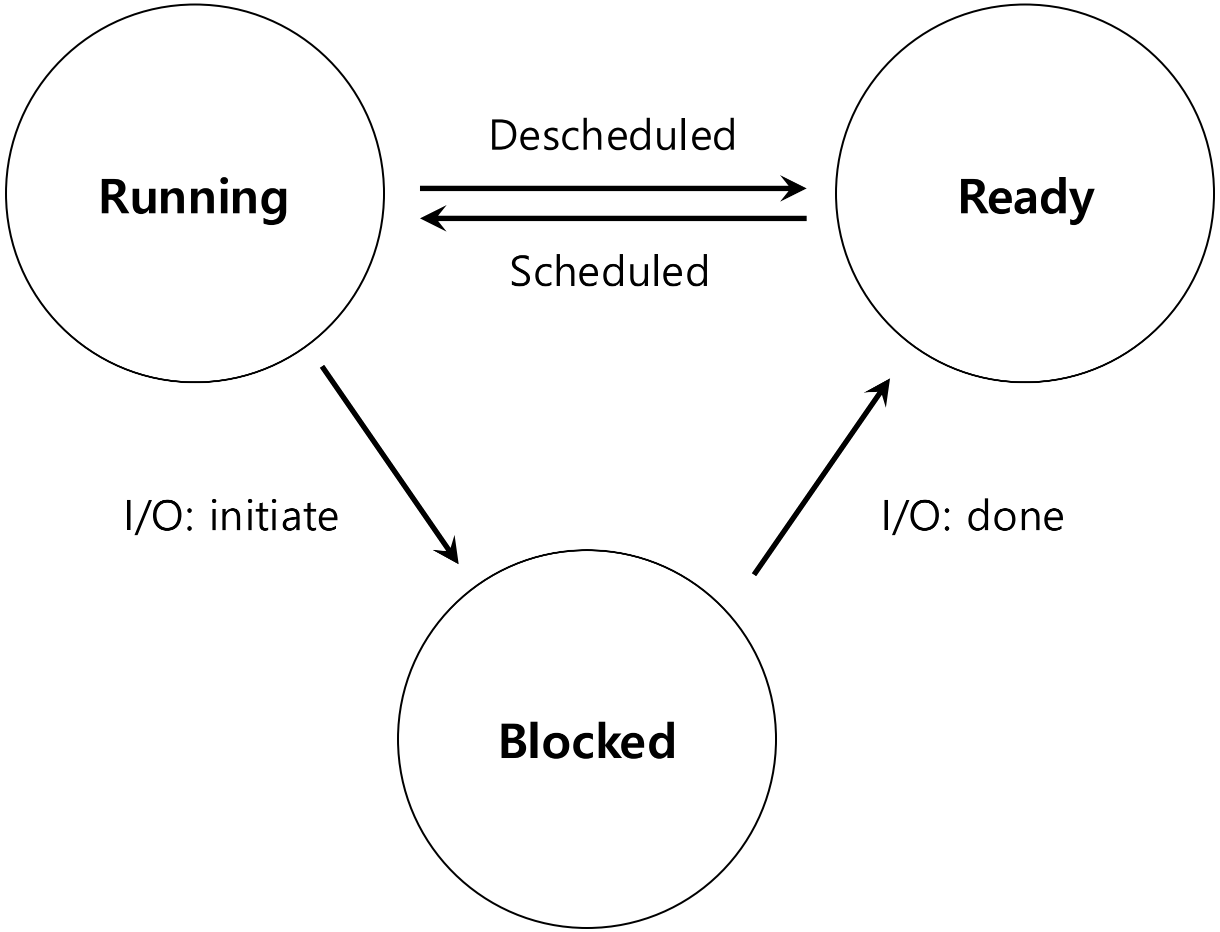
# Loading: From Program To Process



# 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)
  - 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
};
```

# 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<br>2. Allocate memory for program<br>3. Load program into memory<br>4. Set up stack with `argc` / `argv`<br>5. Clear registers<br>6. Execute call `main()` | |
| | 7. Run `main()`<br>8. Execute `return` from `main()` |
| 9. Free memory of process<br>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

# 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<br>Allocate memory for program<br>Load program into memory<br>Setup user stack with argv<br>Fill kernel stack with reg/PC<br>**return-from-trap** | | |
| | restore regs from kernel stack<br>move to user mode<br>jump to main | |
| | | Run main()<br>…<br>Call system<br>**trap** into OS |

# Limited Direction Execution Protocol

| OS @ run (kernel mode) | Hardware | Program (user mode) |
|---|---|---|
| | *(Cont.)* | |
| | save regs to kernel stack<br>move to kernel mode<br>jump to trap handler | |
| Handle trap<br>Do work of syscall<br>**return-from-trap** | | |
| | restore regs from kernel stack<br>move to user mode<br>jump to PC after trap | |
| | | …<br>return from main<br>trap (via `exit()`) |
| Free memory of process<br>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
  - Ex) 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 raise 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 a few register values** for the current process onto its kernel stack
    - General purpose registers
    - PC
    - kernel stack pointer
  - **Restore a few** 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 …<br>syscall handler<br>timer handler |
| **start interrupt timer** | |
| | start timer<br>interrupt CPU in X ms |

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

# Limited Direction Execution Protocol (Timer interrupt)

| OS @ run (kernel mode) | Hardware | Program (user mode) |
|---|---|---|
| | *(Cont.)* | |
| Handle the trap<br>Call switch() routine<br>  save regs(A) to proc-struct(A)<br>  restore regs(B) from proc-struct(B)<br>  switch to k-stack(B)<br>**return-from-trap (into B)** | | |
| | restore regs(B) from k-stack(B)<br>move to user mode<br>jump to B's PC | |
| | | Process B<br>… |

# The xv6 Context Switch Code

```
# void swtch(struct context *old, struct context *new);
#
# Save current register context in old
# and then load register context from new.
.globl swtch
swtch:
        # Save old registers
        movl 4(%esp), %eax          # put old ptr into eax
        popl 0(%eax)                # save the old IP
        movl %esp, 4(%eax)          # and stack
        movl %ebx, 8(%eax)          # and other registers
        movl %ecx, 12(%eax)
        movl %edx, 16(%eax)
        movl %esi, 20(%eax)
        movl %edi, 24(%eax)
        movl %ebp, 28(%eax)

        # Load new registers
        movl 4(%esp), %eax          # put new ptr into eax
        movl 28(%eax), %ebp         # restore other registers
        movl 24(%eax), %edi
        movl 20(%eax), %esi
        movl 16(%eax), %edx
        movl 12(%eax), %ecx
        movl 8(%eax), %ebx
        movl 4(%eax), %esp          # stack is switched here
        pushl 0(%eax)               # return addr put in place
        ret                         # finally return into new ctxt
```

# Worried About Concurrency?

- 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