Theory of Operating Systems

Week 2: Process Management

Faye (Chi Zhang)

CUNY Hunter

February 4, 2026

Today's Outline

1. What is a Process?
2. Process vs Program
3. Process Memory Layout
4. Process Lifecycle and States
5. Process Control Block (PCB)
6. Process Creation: fork() and exec()
7. Process Termination and wait()
8. Context Switching

Duration: 2 hours (with 10-minute break)

What is a Process?

Definition:

- A **program in execution**
- An instance of a running program
- The basic unit of work in an OS

Key Perspective:

- A process is a **state machine**
- Each instruction changes state
- OS manages state transitions

A Process Includes:

- Program code (text section)
- Current activity (PC, registers)
- Stack (temporary data)
- Data section (global variables)
- Heap (dynamically allocated memory)

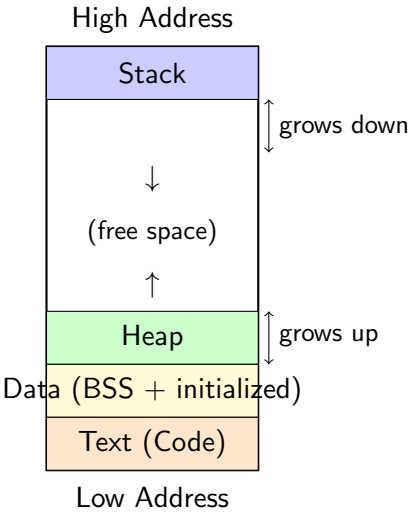
Process vs Program

Program	Process
Static entity (file on disk)	Dynamic entity (in memory)
Contains instructions	Executing instructions
No resources allocated	Resources allocated (CPU, memory)
Passive	Active
Can exist without process	Requires program to exist
Single copy on disk	Multiple instances possible

Analogy

A program is like a recipe. A process is like actually cooking the dish. You can cook multiple dishes from the same recipe simultaneously.

Process Memory Layout



Memory Sections:

- **Text**: Program instructions (read-only)
- **Data**: Global/static variables
  - Initialized data
  - BSS (uninitialized, zero-filled)
- **Heap**: Dynamic allocation (malloc, new)
- **Stack**: Function calls, local variables

Note: Stack and heap grow toward each other!

Understanding Memory Sections

```c
int global_var = 10;          // Data section (initialized)
int uninitialized_var;        // BSS section

int main() {
    int local_var = 5;        // Stack
    static int static_var;    // Data section

    int *ptr = malloc(100); // Heap allocation

    return 0;                 // Code in Text section
}
```

Key Points

- Text section is typically read-only (protection)
- Stack automatically manages function call frames

Multiple Processes from One Program

Same program, multiple processes:

- Each process has its own:
  - Address space (isolated memory)
  - Process ID (PID)
  - CPU state (registers, PC)
  - Open files and resources

Example:

- Open 3 terminal windows
- Each runs /bin/bash
- Each is a separate process
- Different PIDs
- Independent execution
- Don't share memory

Isolation

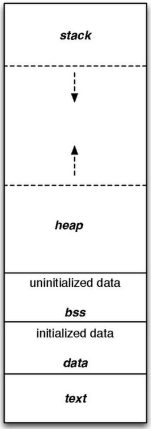
Process isolation is fundamental to OS security and stability. One process cannot corrupt another's memory.

Process Memory Layout (Detailed)

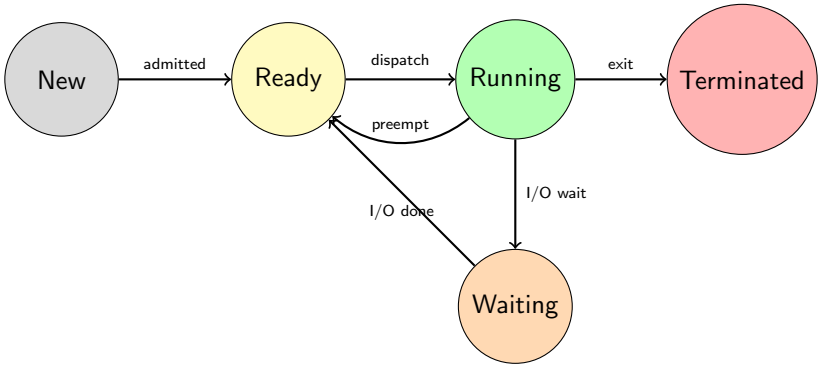
Typical Program Memory:

- Text: Executable code
- Data: Initialized globals
- BSS: Uninitialized globals
- Heap: Dynamic memory
- Stack: Function frames
- Environment variables
- Command-line arguments

Source: Wikimedia Commons



Process States



Process State Descriptions

New Process is being created

- Memory being allocated, PCB initialized

Ready Process waiting to be assigned to CPU

- Has all resources except CPU
- Sitting in the ready queue

Running Instructions are being executed

- Only one process per CPU core at a time

Waiting/Blocked Process waiting for some event

- I/O completion, signal, resource availability

Terminated Process has finished execution

- Resources being deallocated

Process State Diagram

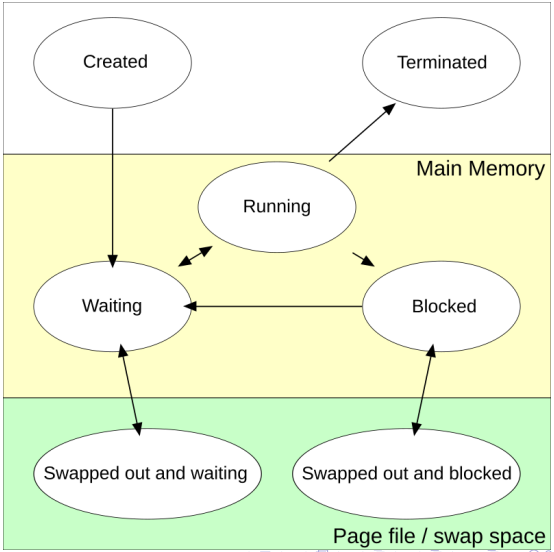
State Machine View:

- Created: Process initialization
- Ready: Waiting for CPU
- Running: Executing instructions
- Blocked: Waiting for I/O
- Terminated: Finished execution

Key Transitions:

- Scheduler dispatch
- Timer interrupt
- I/O request/completion

Source: Wikimedia Commons



State Transitions in Detail

Transition	Cause	Who Initiates
New → Ready	Admitted to system	OS
Ready → Running	Scheduled by scheduler	OS
Running → Ready	Timer interrupt (preemption)	OS
Running → Waiting	I/O request, wait for event	Process
Waiting → Ready	I/O complete, event occurs	OS
Running → Terminated	exit() call or error	Process/OS

Important

A process cannot move directly from Waiting to Running. It must go through Ready first—the scheduler decides who runs.

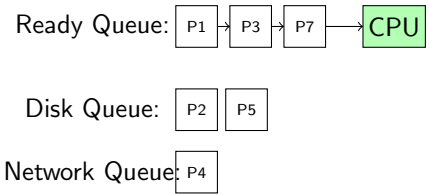
Process Queues

Ready Queue:

- All processes in Ready state
- Waiting for CPU time
- Scheduler picks next process
- Various scheduling algorithms

Wait Queues:

- One queue per device/event
- Processes waiting for I/O
- Move to Ready when done



Process Control Block (PCB)

Definition: Data structure containing all information about a process

Also Called:

- Task Control Block (TCB)
- Process Descriptor
- In Linux: `task_struct`

Purpose:

- OS's complete knowledge of process
- Enables context switching
- Stored in kernel memory

PCB
Process ID
Process State
Program Counter
CPU Registers
Memory Info
I/O Status
Scheduling Info

PCB Contents in Detail

Identification:

- Process ID (PID)
- Parent process ID (PPID)
- User ID (UID), Group ID (GID)

CPU State:

- Program counter
- Stack pointer
- General-purpose registers
- Status/flags register

Memory Management:

- Page table pointer
- Memory limits
- Segment registers

Scheduling:

- Process state
- Priority
- CPU time used

I/O:

- Open file descriptors
- Current working directory

Linux: task_struct

```
struct task_struct {
    volatile long state;        // Process state
    pid_t pid;                  // Process ID
    pid_t tgid;                 // Thread group ID

    struct mm_struct *mm;       // Memory descriptor
    struct fs_struct *fs;       // Filesystem info
    struct files_struct *files; // Open files

    struct task_struct *parent; // Parent process
    struct list_head children;  // Child processes

    unsigned int policy;        // Scheduling policy
    int prio;                   // Priority
    // ... hundreds more fields (several KB total)
};
```

Viewing Process Information

```
# List all processes
$ ps aux
USER   PID %CPU %MEM    VSZ   RSS TTY   STAT START   TIME COMMAND
root     1  0.0  0.1 169436 13140 ?     Ss   10:00   0:01 /sbin/init
faye  1234  0.5  1.2 284512 98304 pts/0 Sl   10:05   0:15 vim file.c

# Process tree
$ pstree -p
systemd(1)---bash(1000)---vim(1234)

# Detailed process info
$ cat /proc/1234/status
Name:   vim
State:  S (sleeping)
Pid:    1234
PPid:   1000
```

10-Minute Break

We'll continue with Process Creation

How Are Processes Created?

Events that create processes:

1. **System boot**: Init/systemd creates initial processes
2. **User request**: Double-click, command line
3. **Process spawn**: Running process creates child
4. **Batch job**: Scheduled task execution

In UNIX/Linux:

- All processes created by existing process
- First process: init (PID 1) or systemd
- Process tree: parent-child relationships
- `fork()` creates new process

The fork() System Call

fork() creates an exact copy of the calling process

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

int main() {
    printf("Before fork\n");
    pid_t pid = fork();

    if (pid < 0) {
        perror("fork failed");
    } else if (pid == 0) {
        printf("Child: my PID = %d\n", getpid());
    } else {
        printf("Parent: child PID = %d\n", pid);
    }
    return 0;
}
```

Understanding fork()

What fork() does:

- Creates new process (child)
- Child is copy of parent
- Both continue from fork()

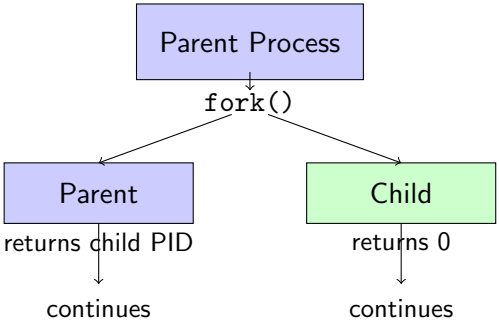
What's copied:

- Address space
- Open file descriptors
- Environment variables
- Current working directory

What's different:

- PID (child gets new ID)
- Return value of fork()
  - Parent: child's PID
  - Child: 0
- Parent PID
- Resource usage counters

fork() Execution Flow



Copy-on-Write (COW) Optimization

Problem: Copying entire address space is expensive

Solution: Copy-on-Write

- Don't copy memory immediately
- Parent and child share same physical pages
- Pages marked as read-only
- When either writes $\rightarrow$ page fault
- Only then is the page copied

Benefit

If child immediately calls exec(), no memory copying needed at all! This is the common pattern (fork + exec).

The exec() Family

exec() replaces current process image with new program

```c
#include <unistd.h>

int main() {
    pid_t pid = fork();

    if (pid == 0) {
        // Child: replace with "ls" program
        execlp("ls", "ls", "-la", NULL);
        // If exec returns, it failed
        perror("exec failed");
        exit(1);
    }
    // Parent continues here
    wait(NULL);  // Wait for child
    return 0;
}
```

exec() Family Variants

Function	Description
`execl`	Arguments as list
`execv`	Arguments as vector (array)
`execlp`	List + search PATH
`execvp`	Vector + search PATH
`execle`	List + custom environment
`execve`	Vector + environment (base syscall)

Naming convention:

- `l`: Arguments as list (comma-separated)
- `v`: Arguments as vector/array
- `p`: Search PATH for executable
- `e`: Specify environment variables

The fork() + exec() Pattern

Standard pattern for running new programs:

1. Parent calls `fork()` to create child
2. Child calls `exec()` to load new program
3. Parent optionally calls `wait()` for child

Why separate fork and exec?

- Flexibility between fork and exec
- Child can modify itself before exec:
  - Redirect I/O (stdin, stdout, stderr)
  - Change working directory
  - Set environment variables
  - Close unwanted file descriptors
  - Set up pipes for communication

How a Shell Works

```
while (1) {
    printf("$ ");
    char *cmd = read_command();

    pid_t pid = fork();
    if (pid == 0) {
        // Child: execute command
        exec(cmd);
        perror("exec failed");
        exit(1);
    } else {
        // Parent: wait for child
        int status;
        waitpid(pid, &status, 0);
    }
}
```

This is the core loop of any UNIX shell (bash, zsh, etc.)!

Process Termination

Normal termination:

- Return from main()
- Call `exit(status)`
- Call `_exit(status)` - immediate, no cleanup

Abnormal termination:

- Signal received (SIGKILL, SIGSEGV, etc.)
- Unhandled exception
- Killed by another process (`kill` command)

What happens:

- Resources deallocated (memory, files)
- Exit status saved for parent
- Process becomes "zombie" until parent calls wait()

Waiting for Child: wait()

```c
#include <sys/wait.h>

int main() {
    pid_t pid = fork();

    if (pid == 0) {
        sleep(2);
        exit(42);  // Exit with status 42
    } else {
        int status;
        pid_t child = wait(&status);

        if (WIFEXITED(status)) {
            printf("Child %d exited with %d\n",
                   child, WEXITSTATUS(status));
        }
    }
```

wait() Variants and Status Macros

Wait functions:

- `wait(&status)`: Wait for any child
- `waitpid(pid, &status, options)`: Wait for specific child
- `waitid()`: More flexible waiting

Status macros:

- `WIFEXITED(status)`: True if normal exit
- `WEXITSTATUS(status)`: Get exit code
- `WIFSIGNALED(status)`: True if killed by signal
- `WTERMSIG(status)`: Get signal number
- `WIFSTOPPED(status)`: True if stopped

Zombie and Orphan Processes

Zombie Process:

- Child has terminated
- Parent hasn't called wait()
- PCB still exists (holds exit status)
- Shows as 'Z' in ps
- Wastes system resources
- Parent must reap it!

Orphan Process:

- Parent terminated first
- Child still running
- Adopted by init (PID 1)
- init will reap when done
- Not a problem

Warning

Too many zombies exhaust PID space and system resources. Always reap your children!

What is a Context Switch?

Definition: Switching the CPU from one process to another

1. Save state of running process to its PCB
2. Load state of next process from its PCB
3. Jump to saved program counter of new process

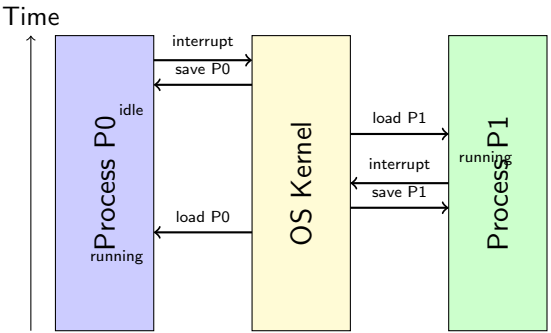
When does it happen?

- Timer interrupt (time slice expired)
- Process blocks (I/O, wait, etc.)
- Higher priority process becomes ready
- Process terminates
- System call that causes reschedule

Context Switch: Step by Step

1. **Interrupt/Trap occurs**: Timer, syscall, or I/O
2. **Save CPU state**: Push registers to kernel stack
3. **Save to PCB**: Copy state to process's PCB
4. **Update state**: Mark old process as Ready/Waiting
5. **Select next**: Scheduler chooses next process
6. **Update state**: Mark new process as Running
7. **Load from PCB**: Restore registers from new PCB
8. **Switch address space**: Load new page table
9. **Jump**: Set PC to saved value

Context Switch Visualization



The Cost of Context Switching

Direct costs:

- Save/restore registers: microseconds
- Switch page tables
- Flush TLB (Translation Lookaside Buffer)
- Kernel code execution

Indirect costs (often larger):

- **Cache pollution**: New process's data not cached
- **TLB misses**: Address translations not cached
- **Pipeline flush**: CPU pipeline restarts
- **Branch predictor**: Predictions invalidated

Typical total cost: 1-100 microseconds

Key Takeaways

1. **Process**: Program in execution with code, data, stack, heap
2. **States**: New $\rightarrow$ Ready $\rightarrow$ Running $\rightarrow$ Waiting $\rightarrow$ Terminated
3. **PCB**: All information OS needs about a process
4. **fork()**: Creates child process (copy of parent)
5. **exec()**: Replaces process image with new program
6. **wait()**: Parent waits for child termination (reaps zombie)
7. **Context switch**: Save old state, load new state, expensive!

This Week's Tasks

- **Quiz 2**: Process management concepts
- **Reading**: Textbook Chapters 3-5
- **Assignment 1**: Process Scheduling Simulator (Released!)
  - Implement process state transitions
  - Due: February 18
- **Hands-on**: Write programs using fork(), exec(), wait()
- **Next Week**: Process Scheduling Algorithms
  - FCFS, SJF, Round Robin, Priority
  - Turnaround time, response time, throughput

Questions?