Process Management

Advanced Operating Systems — Week 2

Hunter College, CUNY

Faye

czhang2@gradcenter.cuny.edu

February 2026

Contents

1 What is a Process?

A **process** is an instance of a program in execution. It is the OS's fundamental unit of work — the abstraction through which the OS virtualizes the CPU.

1.1 Program vs Process

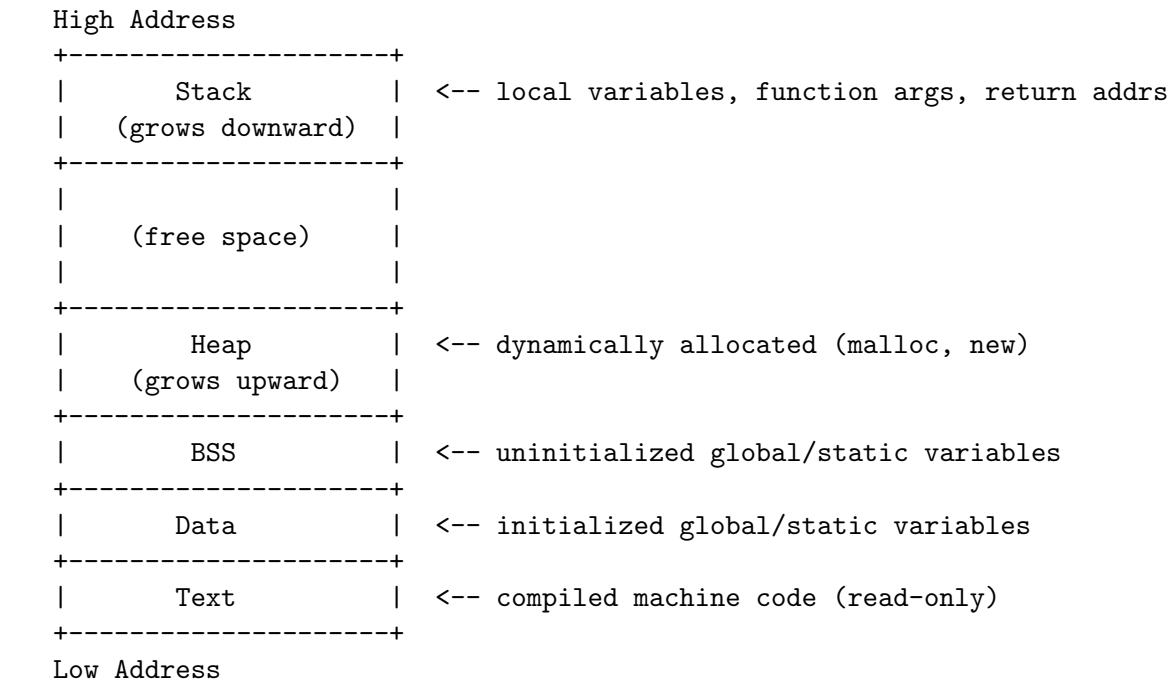
Aspect	Program	Process
Nature	Passive (file on disk)	Active (executing in memory)
Lifetime	Permanent (until deleted)	Temporary (created, runs, terminates)
Resources	None	CPU time, memory, open files, sockets
State	No state	Has state (running, ready, waiting)
Instances	One copy on disk	Multiple processes from same program
Identity	File path / inode	PID (Process ID)

Analogy: Recipe vs Cooking

A **program** is like a *recipe* written in a cookbook — it is a static set of instructions sitting on a shelf. A **process** is like the act of *cooking* — you follow the recipe, use ingredients (resources), and the state changes over time (chopping, mixing, baking). Multiple chefs (CPUs) can follow the same recipe simultaneously, creating multiple independent cooking sessions (processes).

2 Process Memory Layout

Every process has a virtual address space organized into well-defined segments:



2.1 Where Variables Live

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

```c
int global_init = 42;        // Data segment (initialized)
int global_uninit;           // BSS segment (uninitialized, zeroed)
static int static_var = 10;  // Data segment (initialized static)

int main(void) {
    int local = 5;           // Stack
    static int s_local = 7;  // Data segment (static, persists)
    int *p = malloc(100);    // p is on Stack; *p is on Heap
    char *str = "hello";     // str on Stack; "hello" in Text (read-
        only)

    printf("Text:  %p (main)\n", (void *)main);
    printf("Data:  %p (global_init)\n", (void *)&global_init);
    printf("BSS:   %p (global_uninit)\n", (void *)&global_uninit);
    printf("Heap:  %p (malloc'd)\n", (void *)p);
    printf("Stack: %p (local)\n", (void *)&local);

    free(p);
    return 0;
}
```

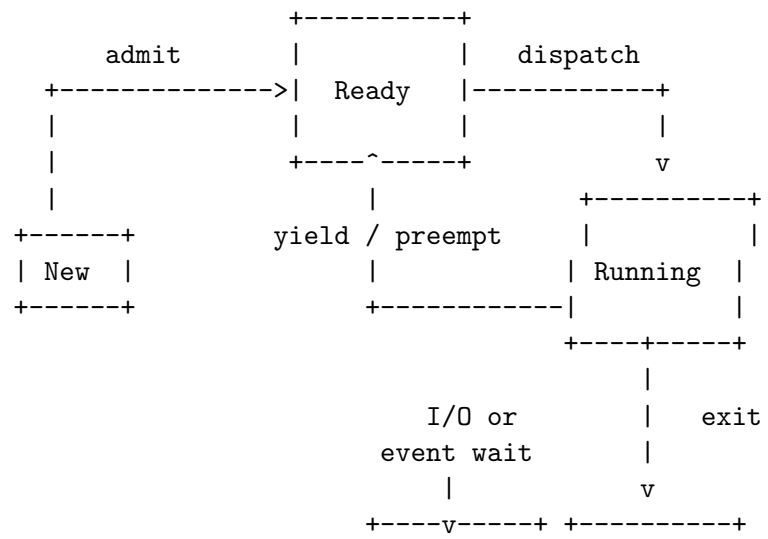
Understanding `p` vs `*p`

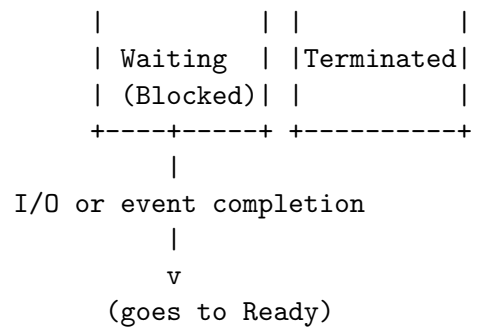
When we write `int *p = malloc(100);`:

- `p` itself is a local variable — it lives on the **Stack**. It holds an address (8 bytes on a 64-bit system).
- `*p` (the memory `p` points to) is the block returned by `malloc` — it lives on the **Heap** (100 bytes in this case).
- If `p` goes out of scope without `free(p)`, the heap memory is **leaked** — still allocated but unreachable.

3 Process States

A process transitions through several states during its lifetime:





3.1 State Transition Table

From	To	Cause	Initiator
New	Ready	Process admitted to ready queue	OS (long-term scheduler)
Ready	Running	CPU dispatched to process	OS (short-term scheduler)
Running	Ready	Time quantum expired or preempted	OS (timer interrupt)
Running	Waiting	Process requests I/O or waits on event	Process (system call)
Waiting	Ready	I/O complete or event occurred	Hardware (interrupt)
Running	Terminated	Process calls `exit()` or receives fatal signal	Process / OS

Important: No Waiting -> Running Transition

A process in the **Waiting** (Blocked) state *cannot* go directly to the **Running** state. When its I/O completes, it moves to the **Ready** queue and must wait for the scheduler to dispatch it. This ensures fairness — a process that just finished I/O does not automatically preempt the currently running process.

4 Process Control Block (PCB)

The OS maintains a **Process Control Block (PCB)** for every process. It stores all the information the OS needs to manage the process:

- **Process identification** — PID, parent PID (PPID), user ID (UID), group ID (GID).
- **Process state** — running, ready, waiting, etc.
- **CPU registers** — program counter (PC/RIP), stack pointer (SP/RSP), general-purpose registers, flags register.
- **CPU scheduling info** — priority, scheduling queue pointers, time slice remaining.
- **Memory management info** — page table base register (CR3), segment table, memory limits, VM areas.
- **I/O status** — list of open files (file descriptor table), pending I/O requests, assigned devices.
- **Accounting info** — CPU time used, wall clock time, resource usage limits.
- **Signal handling** — signal mask, pending signals, signal handlers.

4.1 Linux `task_struct`

In Linux, the PCB is the `task_struct` (defined in `include/linux/sched.h`), one of the largest structures in the kernel (~6–8 KB):

```
// Simplified view of task_struct
struct task_struct {
    volatile long state;          // process state (-1 unrunnable, 0
       runnable, >0 stopped)
    pid_t pid;                    // process ID
    pid_t tgid;                   // thread group ID (== pid for main
        thread)
    struct task_struct *parent;   // pointer to parent process
    struct list_head children;    // list of child processes
    struct mm_struct *mm;         // memory descriptor (page tables,
       VMAs)
    struct files_struct *files;   // open file descriptors
    struct thread_struct thread; // CPU-specific state (registers)
    int prio;                     // dynamic priority
    unsigned int policy;          // scheduling policy (SCHED_NORMAL,
        SCHED_FIFO, ...)
    u64 utime, stime;            // user and system CPU time
    // ... hundreds more fields
};
```

4.2 The `/proc` Filesystem

Linux exposes PCB information through `/proc`:

```
# View process status
cat /proc/<pid>/status

# View memory map
cat /proc/<pid>/maps

# View open file descriptors
ls -la /proc/<pid>/fd/

# View command line arguments
cat /proc/<pid>/cmdline

# View CPU and memory usage summary
cat /proc/<pid>/stat
```

5 Process Creation: `fork()`

`fork()` is the **only** way to create a new process in UNIX/Linux (aside from the initial `init`/`systemd` process created at boot).

What `fork()` does:

1. Creates a **new process** (child) that is an almost exact copy of the calling process (parent).
2. The child gets a **new PID** but inherits the parent's memory, open files, environment, and signal handlers.
3. Both parent and child continue execution from the instruction *after* the `fork()` call.

Return values:

- In the **parent**: returns the child's PID (a positive integer).
- In the **child**: returns `0`.
- On **failure**: returns `-1` (no child created; `errno` set).

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

int main(void) {
    printf("Parent PID: %d\n", getpid());

    pid_t pid = fork();

    if (pid < 0) {
        perror("fork failed");
        return 1;
    } else if (pid == 0) {
        // Child process
        printf("Child here! PID=%d, Parent PID=%d\n", getpid(),
            getppid());
    } else {
        // Parent process
        printf("Parent here! Child PID=%d\n", pid);
        wait(NULL);  // wait for child to finish
        printf("Child has terminated.\n");
    }
    return 0;
}
```

Copy-on-Write (COW)

Modern kernels do **not** physically copy the parent's entire address space on `fork()`. Instead, parent and child share the same physical pages, marked **read-only**. When either process tries to *write* to a shared page, a page fault occurs, and the kernel creates a private copy of that page for the writer. This makes `fork()` very fast — only the page tables and PCB are copied initially, not the actual memory contents.

5.1 Fork Counting Challenge

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

int main(void) {
    fork();   // Line 1: 1 process -> 2 processes
    fork();   // Line 2: 2 processes -> 4 processes
    fork();   // Line 3: 4 processes -> 8 processes
    printf("Hello!\n");
    return 0;
}
// Output: "Hello!" is printed 8 times (2^3 = 8 processes)
```

General rule: n consecutive `fork()` calls (with no conditionals) create 2^n total processes.

6 The `exec()` Family

While `fork()` creates a copy, `exec()` **replaces** the current process's memory image with a new program. After `exec()`, the process has the same PID but runs entirely different code.

6.1 Variants

Function	Signature	Description
`execl`	`execl(path, arg0, ..., NULL)`	List of args, absolute path
`execv`	`execv(path, argv[])`	Vector (array) of args, absolute path
`execlp`	`execlp(file, arg0, ..., NULL)`	List of args, searches `PATH`
`execvp`	`execvp(file, argv[])`	Vector of args, searches `PATH`
`execle`	`execle(path, arg0, ..., NULL, envp[])`	List of args, custom environment
`execve`	`execve(path, argv[], envp[])`	Vector of args, custom env (raw syscall)

6.2 Naming Convention

- **l** (list) — arguments passed as a comma-separated list, terminated by `NULL`.
- **v** (vector) — arguments passed as a `char *argv[]` array.
- **p** (path) — searches the `PATH` environment variable for the executable.
- **e** (environment) — allows passing a custom environment (`char *envp[]`).

Note: Only `execve()` is a true system call. All others are C library wrappers that ultimately call `execve()`.

6.3 The `fork()` + `exec()` Pattern

This is the standard UNIX process creation pattern:

1. Parent calls `fork()` to create a child process.
2. Child calls `exec()` to replace itself with a new program.
3. Parent calls `wait()` to wait for the child to finish.

Why separate `fork()` and `exec()`? The gap between `fork()` and `exec()` allows the child to set up its environment before running the new program: redirect I/O (`dup2`), close unnecessary file descriptors, change directory, set signal handlers, drop privileges, etc.

6.4 Shell Loop Example

```
// Simplified shell loop
while (1) {
    char *cmd = read_command();  // read user input
    char **args = parse(cmd);    // parse into argv[]

    pid_t pid = fork();
    if (pid == 0) {
        // Child: set up redirections, pipes, etc.
        // Then replace with the requested program
```

```
        execvp(args[0], args);
        perror("exec failed");  // only reached if exec fails
        exit(1);
    } else {
        // Parent (shell): wait for child
        int status;
        waitpid(pid, &status, 0);
    }
}
```

7 Process Termination

7.1 Normal Termination

- `return` from `main()` — the C runtime calls `exit()`.
- `exit(status)` — flushes stdio buffers, calls `atexit` handlers, then calls `_exit()`.
- `_exit(status)` / `_Exit(status)` — immediate termination; no cleanup.

7.2 Abnormal Termination

- Unhandled signal: `SIGSEGV` (segfault), `SIGFPE` (divide by zero), `SIGKILL`, `SIGABRT`.
- Calling `abort()` — raises `SIGABRT`.
- Killed by another process: `kill(pid, SIGKILL)`.

Zombie vs Orphan Processes

Zombie Process:

- A process that has **terminated** but whose parent has **not yet called** `wait()`.
- It retains its PID and exit status in the process table (minimal resources, but wastes a PID slot).
- Shown as `Z` (defunct) in `ps` output.
- Fix: parent must call `wait()`/`waitpid()`, or handle `SIGCHLD`.
- Danger: too many zombies can exhaust the PID space.

Orphan Process:

- A process whose **parent has terminated** before it.
- The orphan is **re-parented** to `init` (PID 1) or a subreaper process.
- `init` periodically calls `wait()` to clean up orphans — so orphans are generally harmless.
- Orphans continue running normally; they are not killed.

7.3 Waiting for Children: `wait()` and `waitpid()`

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

pid_t wait(int *status);                    // wait for ANY child
pid_t waitpid(pid_t pid, int *status, int options);  // wait for
    specific child
```

Status examination macros:

- `WIFEXITED(status)` — true if child terminated normally.
- `WEXITSTATUS(status)` — exit code (0–255) if `WIFEXITED` is true.
- `WIFSIGNALED(status)` — true if child was killed by a signal.
- `WTERMSIG(status)` — signal number that killed the child.
- `WIFSTOPPED(status)` — true if child is currently stopped.
- `WSTOPSIG(status)` — signal that stopped the child.

8 Context Switching

A **context switch** occurs when the OS saves the state of the currently running process and restores the state of the next process to run.

8.1 Steps in a Context Switch

1. An interrupt, trap, or system call transfers control to the OS.
2. The OS saves the current process's CPU state (registers, PC, SP, flags) into its PCB.
3. The OS updates the current process's state (Running → Ready or Waiting).
4. The scheduler selects the next process to run from the ready queue.
5. The OS loads the new process's CPU state from its PCB.
6. The OS updates the page table base register (CR3) to the new process's page tables.
7. The OS returns to user mode; the new process resumes execution.

8.2 Costs of Context Switching

Direct costs:

- Saving and restoring registers (∼1–2 µs).
- Switching page tables and flushing TLB entries.
- Kernel overhead for scheduler decisions.

Indirect costs (often more significant):

- **Cache pollution** — the new process's working set replaces the old process's cache lines (L1, L2, L3).
- **TLB misses** — virtual-to-physical translations must be re-fetched.
- **Pipeline flush** — CPU branch predictor state is invalidated.
- **Cache warm-up** — may take thousands of cycles before the new process reaches full speed.

8.3 Context Switch vs Process Swapping

Aspect	Context Switch	Swapping
What moves	CPU registers, PC, SP	Entire process image (memory pages)
Where	Between CPU and PCB (RAM)	Between RAM and disk (swap space)
Speed	Fast (∼1–10 µs)	Very slow (∼milliseconds, disk I/O)
When	Every scheduling decision	Only under severe memory pressure
Frequency	Very frequent (100s/sec)	Rare (ideally never)
Initiated by	Short-term scheduler	Medium-term scheduler (swapper)

9 Key Takeaways

Week 2 Summary

1. A **process** is a program in execution — the OS's fundamental abstraction for CPU virtualization. A program is passive; a process is active.
2. The process memory layout has five segments: **Text** (code), **Data** (initialized globals), **BSS** (uninitialized globals), **Heap** (dynamic allocation, grows up), and **Stack** (local variables, grows down).
3. Processes transition through five states: **New** → **Ready** ↔ **Running** → **Terminated**, with a **Waiting** state for I/O. A blocked process must return to Ready before Running.
4. The **PCB** (`task_struct` in Linux) stores all process metadata. The `/proc` filesystem exposes this information.
5. `fork()` creates a child process via **Copy-on-Write**; `exec()` replaces the process image. Together they form the UNIX process creation pattern.
6. **Zombies** arise when a parent does not call `wait()`; **orphans** are re-parented to `init`. Always reap children.
7. **Context switches** are fast (∼µs) but have significant indirect costs from cache and TLB pollution. Minimizing unnecessary context switches is key to system performance.