Computer Systems

Lecture 4: Processes (Part III)

Huaicheng Li

February 4, 2026



CS 3214
Spring 2026

Quick Review: What Did We Learn?

Parts 1 & 2 Recap

- Process = illusion of dedicated machine (Part 1)
- Dual-mode operation, context switching (Part 1)
- Process states: RUNNING, READY, BLOCKED (Part 2)
- Job control and the T state (Part 2)

Today's Focus: Process Creation APIs

- How does the shell create a new process to run `ls`?
- What are `fork()`, `exec()`, and `posix_spawn()`?
- How does a parent wait for its child?
- What are zombies and orphans?

Process Management (Windows)

- OS provide APIs (system calls) to manage processes
- Example: CreateProcessA in Windows

```
BOOL CreateProcessA(
  LPCSTR                lpApplicationName,
  LPSTR                 lpCommandLine,
  LPSECURITY_ATTRIBUTES lpProcessAttributes,
  LPSECURITY_ATTRIBUTES lpThreadAttributes,
  BOOL                  bInheritHandles,
  DWORD                 dwCreationFlags,
  LPVOID                lpEnvironment,
  LPCSTR                lpCurrentDirectory,
  LPSTARTUPINFOA        lpStartupInfo,
  LPPROCESS_INFORMATION lpProcessInformation
);
```

- Creates ("spawns") a new process, and instruct it to run a new program with arguments and attributes

Process Management (Unix)

- Unix separates process creation from loading a new program
- The `fork()` system call creates a new process, but does not load a new program
- The newly created process is called a child process (the creating process is referred to as parent)
  - Corollary: Unix processes form a tree-like hierarchy
  - Child processes may inherit parts of their environment from their parents, but are otherwise distinct entities
- The child process then may change/set up the environment and, when ready, load a new program that replaces the current program but retains certain aspects of the environment (`exec()`)
- The parent has the option of waiting (via `wait()`) for the child process to terminate, which is also called "joining" the child process
  - Parent can also learn how the child process terminated, e.g. the code that the child passed to `exit()`

Comparison of `fork()` and `exec()`

- `fork()`
  - Keeps program and process, but also creates a new process
  - New process is a clone of the parent; child state is a (now separate) copy of parent's state, including everything: heap, stack, file descriptors
  - Called once, returns twice (once in parent, once in child)
- `exec()`
  - Keeps process, but discards old program and loads a new program
  - Reinitializes process state (clears heap + stack, starts at new program's `main()`); except it retains file descriptors
  - If successful, is called once but does not return
  - includes multiple variants (`execvp()`, etc.)

fork(): Called Once, Returns Twice

The Weirdest Function You'll Ever See

`fork()` is called once but returns **twice** – once in each process!

In the parent:

- Returns the child's PID (> 0)
- Continues executing after fork
- Can use PID to wait for child

In the child:

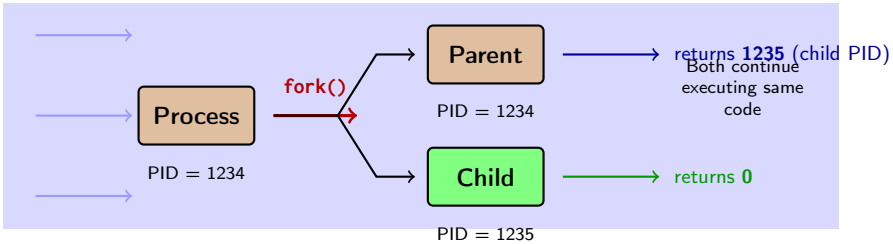
- Returns 0
- Also continues after fork
- Has same code, separate memory

Key Insight

After `fork()`, parent and child have **identical but separate** copies of all memory. Changes in one don't affect the other.

The child is a "clone" at the moment of fork, but then they diverge.

Visualizing fork(): The Raft in the River



Key Insight (Godmar's Analogy)

At the moment of `fork()`, both rafts are **identical** – same cargo, same position in the code. They then float downstream independently, potentially taking different branches.

fork() Basic Example

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

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

    if (pid < 0) {
        perror("fork failed");
    } else if (pid == 0) {
        // This is the CHILD process
        printf("Child: my PID is %d\n", getpid());
    } else {
        // This is the PARENT process
        printf("Parent: my child's PID is %d\n", pid);
    }
    return 0;
}
```

fork() Subtleties and Copy-on-Write

What IS copied:

- Memory
- File descriptors
- Signal handlers
- Current working directory
- Environment variables

What is NOT copied:

- Other threads (only calling thread!)
- Pending async I/O
- File locks
- Memory locks (mlock)

Copy-on-Write: Why fork() Isn't Wasteful

"fork() copies all memory, then exec() throws it away. Wasteful?"
No! Modern systems use **copy-on-write (COW)**: pages are shared read-only until one process writes, then only that page is copied. fork() is nearly free!

Tricky: File Descriptors are Shared!

Parent and child have **separate** fd tables, but they point to the **same** kernel file objects. They **share the file offset**!

fork() Pitfall: Buffered Output

Classic Interview Question

How many lines does this print?

```c
int main() {
    printf("hello");  // No newline!
    fork();
    printf(" world\n");
    return 0;
}
```

Answer: It prints "hello world" **twice**!

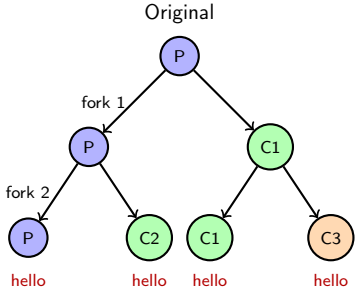
Why? The "hello" sits in the stdio buffer (not flushed). After fork(), both parent and child have a copy of this buffer. Both then print "hello world\n" when the buffer flushes.

Use `fflush(stdout)` before fork, or use unbuffered I/O.

Solving Fork Puzzles: Draw the Process Tree

Strategy: Trace the Execution

For each `fork()`, the process tree doubles. Draw it out!




```
fork();
fork();
printf("hello\n");
```

General Rule

n calls to `fork()` create 2^n processes total.

Each process executes code after the last fork.

Fork Puzzle 1: Variable Copies

```c
int main() {
    int x = 1;
    if (fork() == 0)
        printf("p1: x=%d\n", ++x);
    printf("p2: x=%d\n", --x);
    exit(0);
}
```

Fork Puzzle 2: Multiple Forks (The 2^n Rule)

```c
int main() {
    int i;
    for (i = 0; i < 2; i++)
        fork();
    printf("hello\n");
    exit(0);
}
```

How many “hello” lines?

Fork Puzzle 3: Output Ordering with wait()

```c
int main() {
    if (fork() == 0) {
        printf("a"); fflush(stdout);
    } else {
        printf("b"); fflush(stdout);
        waitpid(-1, NULL, 0);
    }
    printf("c"); fflush(stdout);
    exit(0);
}
```

Fork Puzzle 4: Counter with Wait

```c
int counter = 1;

int main() {
    if (fork() == 0) {
        counter--;
        exit(0);
    } else {
        wait(NULL);
        printf("counter = %d\n",
               ++counter);
    }
    exit(0);
}
```

The exec() Family: Quick Reference

Six Variants – Know the Naming Convention

Function	Args	PATH?	Env
`execl(path, arg0, ...)`	list	No	inherit
`execv(path, argv[])`	vector	No	inherit
`execlp(file, arg0, ...)`	list	Yes	inherit
`execvp(file, argv[])`	vector	Yes	inherit
`execle(path, ..., envp[])`	list	No	explicit
`execve(path, argv[], envp[])`	vector	No	explicit

Naming mnemonic:

- `l` = **l**ist (args as separate parameters, NULL-terminated)
- `v` = **v**ector (args as `char *argv[]` array)
- `p` = searches **P**ATH environment variable
- `e` = explicit **e**nvironment array

`execve()` is the actual system call; others are library wrappers.

What Survives exec()? (VM Connection)

Preserved

- Process ID (PID)
- Parent PID (PPID)
- Process group ID (PGID)
- Session ID (SID)
- Open file descriptors*
- Current working directory
- Signal mask, pending signals
- Resource limits

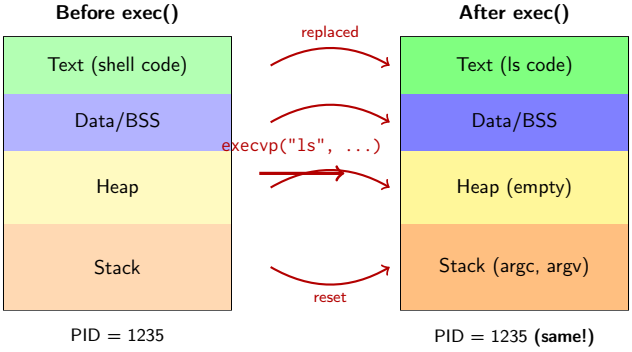
Replaced/Reset

- Text (code) segment
- Data, BSS, Heap, Stack
- Signal handlers (reset)
- Memory mappings
- Thread state

Under the hood: Kernel parses ELF, destroys old page tables, creates new ones, uses **demand paging** (pages loaded on first access).

*Unless `O_CLOEXEC` / `FD_CLOEXEC` flag is set.

exec() Replaces the Address Space



What exec() Preserves

Same process: PID, PPID, process group, session, open file descriptors*, cwd
New program: Text, data, heap, stack all replaced; execution starts at `_start`

Why fork() and exec() Are Separate

Key Question

Why didn't Unix provide a single "create process and run program" call?

Answer: The gap between fork and exec is where you set up the child's environment:

- **I/O redirection**: Close/open file descriptors before exec (e.g., `ls > out.txt`)
- **Pipes**: Set up pipe endpoints before exec (e.g., `cat | grep`)
- **Env changes**: Change directory, set env vars, drop privileges

Shell example for `ls > output.txt`:

1. fork() – create child
2. Child: close(1) – close stdout
3. Child: open("output.txt", ...) – gets fd 1 (lowest available!)
4. Child: exec("ls") – ls writes to fd 1, which is now the file

posix_spawn: The Modern Alternative

Combines fork+exec into one call. Declarative setup via file_actions. **Required for P1!**

Waiting for Children: waitpid()

```c
pid_t waitpid(
    pid_t pid,    // Which child? -1 = any, >0 = specific PID
    int *status,  // OUTPUT: termination status
    int options   // Flags: WNOHANG, WUNTRACED, etc.
);
```

Returns: PID of child that changed state, 0 if WNOHANG and no child ready, -1 on error

Common patterns for P1:

- `waitpid(pid, &status, 0)` – Block until specific child exits
- `waitpid(-1, &status, 0)` – Block until **any** child exits
- `waitpid(-1, &status, WNOHANG)` – Check without blocking
- `waitpid(-1, &status, WUNTRACED)` – Also report stopped children

P1 Connection

Your shell uses `waitpid()` to wait for foreground jobs and reap children.

Interpreting Exit Status: The WIF* Macros

After waitpid() returns, what happened to the child?

```c
int status;
waitpid(pid, &status, 0);

if (WIFEXITED(status)) {
    // Child called exit() or returned from main()
    int exit_code = WEXITSTATUS(status);  // 0-255
    printf("Exited with code %d\n", exit_code);
}
else if (WIFSIGNALED(status)) {
    // Child was killed by a signal (e.g., SIGKILL, SIGSEGV)
    int sig = WTERMSIG(status);
    printf("Killed by signal %d\n", sig);
}
else if (WIFSTOPPED(status)) {
    // Child was stopped (Ctrl-Z or SIGSTOP)
```

Some Unix Jargon for various scenarios

Zombies

Processes that have exited, but whose parent is still alive **and** has not (yet) waited for them. They will exist until either their parent waits for them ("reaps them") or their parent exits.

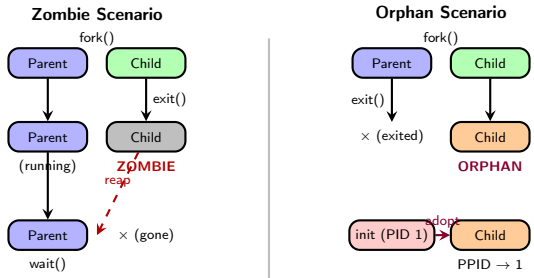
Orphans & Daemons

Processes that are alive, but whose parent exited without waiting for them. They are reassigned to the `init` process (pid 1). Usually unintended. If intended, the orphan may also be called a daemon.

Run-Aways

Processes that are alive, have not exited, are always READY/RUNNING and thus, if scheduled, use up 100% of a CPU without performing useful work.

Zombie vs Orphan: Lifecycle Comparison



Key Difference

Zombie: Child exits first, waits to be reaped ("undead")
Orphan: Parent exits first, child is adopted by init

The "Message in a Bottle" Metaphor

Key Insight from Godmar's Lecture

Think of the exit status as a "message in a bottle" from child to parent.

When child exits first (zombie):

- Child puts message in bottle, leaves it at the "gate"
- Child becomes a zombie (dead but message still there)
- When parent calls `wait()`, it picks up the message
- Zombie is then reaped (fully deleted)

When parent waits first:

- Parent blocks at the gate, waiting for message
- When child exits, message is delivered immediately
- No zombie needed (or very brief)

The Guarantee

The OS guarantees the message is **never lost**, regardless of timing.

Creating and Observing a Zombie

Demo: Create a Zombie Process

```c
int main() {
    pid_t pid = fork();
    if (pid == 0) {
        // Child: exit immediately
        exit(0);
    }
    // Parent: sleep without calling wait()
    printf("Child PID: %d\n", pid);
    printf("Check: ps aux | grep %d\n", pid);
    sleep(60);  // Zombie exists during this time
    // After 60 seconds, we exit and zombie is adopted
    return 0;
}
```

Observation

Orphans: When Parents Die First

The Orphan Problem:

- Parent exits while children still running
- Who will reap the children when they exit?
- Can't have permanent zombies!

The Solution: Adoption

- Orphans are "adopted" by init (PID 1)
- init periodically calls `wait()`
- Ensures all zombies are eventually reaped

Demo: Watch Orphan Adoption

`ps -ef | grep your_program`
Before parent dies: PPID = parent's PID
After parent dies: PPID = 1 (init)

P1 Rule

Always reap your children! For foreground jobs, call `waitpid()`.
For background jobs, use a SIGCHLD handler.

EXAM FOCUS: Process Creation

Common Question Types

1. **fork() output:**
   - "How many times does this print X?" (Draw process tree!)
   - "What are all possible outputs?"
2. **fork() return values:**
   - "What does fork() return in parent?" (Child's PID)
   - "What does fork() return in child?" (0)
3. **waitpid() and status:**
   - "Given status = 0x000B, what happened?" (Use WIFSIGNALED, WTERMSIG)
   - "What's the difference between wait() and waitpid()?"
4. **Zombies:**
   - "Does this code create a zombie?" (Yes if no wait before parent continues)

Practice: CS:APP3e Chapter 8, Problems 8.6-8.12

Summary: Key Takeaways

1. **fork() creates a clone**
   - Called once, returns twice
   - Child gets 0, parent gets child's PID
2. **exec() replaces the program**
   - Called once, doesn't return (if successful)
   - Preserves PID, file descriptors, cwd
3. **wait()/waitpid() reaps children**
   - Gets exit status via WIF* macros
   - Prevents zombies

Reading: CS:APP Chapter 8.4 (Process Control)
Next lecture: File descriptors and pipes

References

- Computer Systems: A Programmer's Perspective, 3rd Ed. – Bryant & O'Hallaron
- Operating Systems: Three Easy Pieces – Arpaci-Dusseau (free online)
- The Linux Programming Interface – Michael Kerrisk
- Advanced Programming in the UNIX Environment – Stevens & Rago