



Linux Pipes & Process Lifecycle: The Complete Guide

From Zero to Hero - Everything about Pipes, fork(), Zombies, Orphans & wait()



Table of Contents

Part 1: Foundation (Start Here!)

1. How Linux Really Works (The Big Picture)
2. What is a Process?
3. Memory: Why Parent & Child DON'T Share
4. File Descriptors: Your Gateway to Everything

Part 2: The Fork Story

5. What fork() Really Does
6. Copy-on-Write: The Memory Trick
7. Process States & The Kernel Scheduler

Part 3: Process Lifecycle - Zombies & Orphans

8. What Happens When a Process Dies?
9. Process Exit Status (The Final Message)
10. The Parent-Child Contract
11. Zombie Processes (The Undead!)
12. Orphan Processes (Home Alone)
13. How to Avoid Zombies

Part 4: The wait() Family

14. wait() - The Basic Version
15. waitpid() - More Control
16. WEXITSTATUS & Friends (Reading Exit Status)

Part 5: Enter Pipes

17. Why We Need IPC (Inter-Process Communication)

18. What is a Pipe? (Core Idea)
19. Creating a Pipe: pipe()
20. The Magic Connection: Fork + Pipe

Part 6: Deep Kernel Dive

21. How Pipes Work Inside the Kernel
22. VFS: Virtual File System
23. Inodes, Pages, and Buffers
24. The /proc Filesystem

Part 7: Making It Work

25. Why Close Unused Ends?
26. Reading & Writing
27. Blocking Behavior
28. Pipes + exec(): Real Pipelines

Part 8: Advanced Topics

29. Named Pipes (FIFOs)
 30. Buffer Internals
 31. Complete Code Walkthrough
-

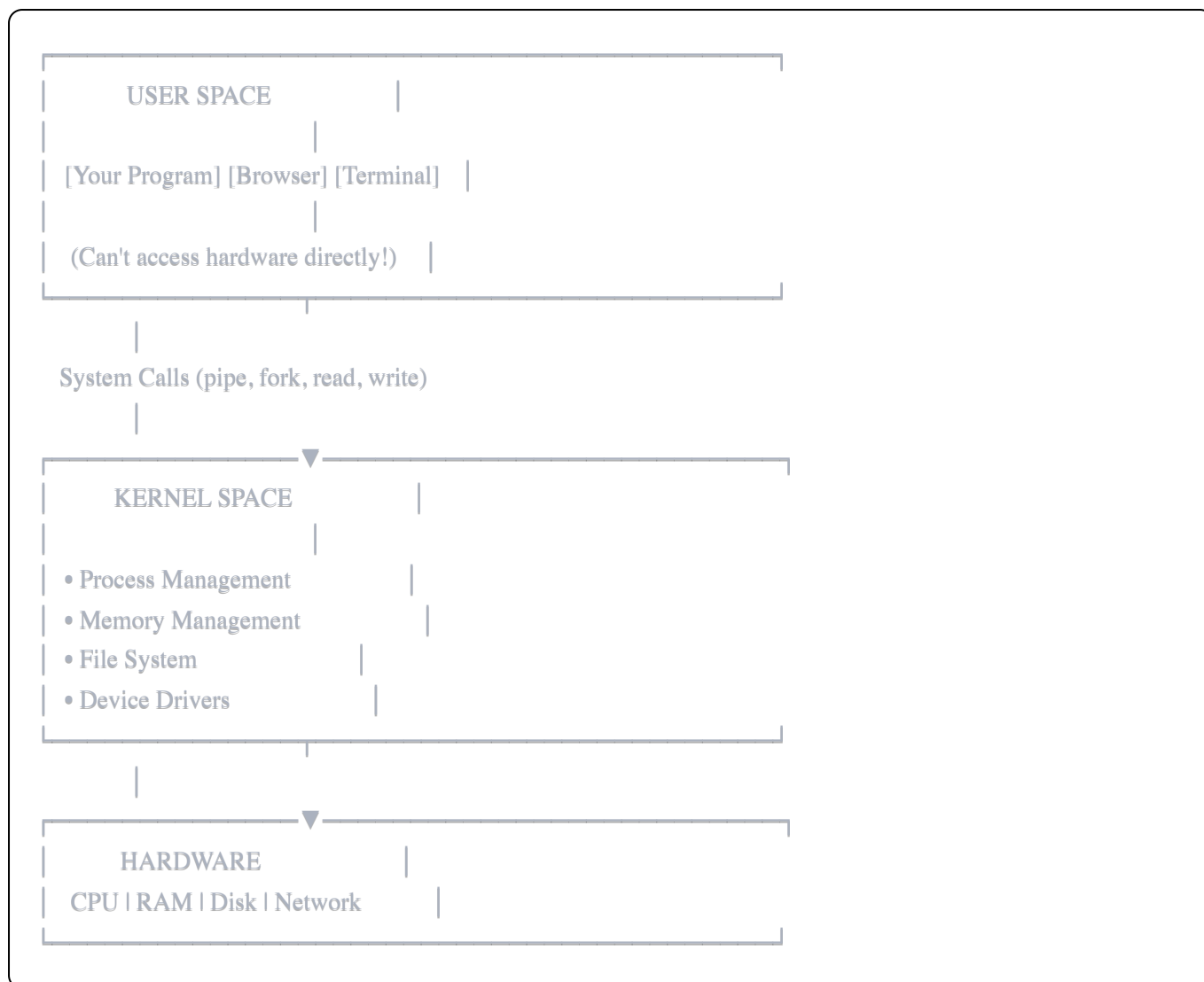
PART 1: FOUNDATION

1. 🖥️ How Linux Really Works (The Big Picture)

Before we talk about pipes and processes, you need to understand how Linux actually runs programs.

The Kernel vs User Space

Key Point: Your programs run in user space and can't touch hardware directly. They must ask the kernel through system calls.



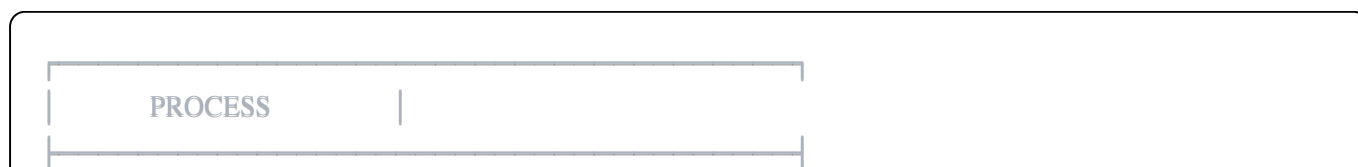
System calls are like asking the kernel for a favor:

- `open()` - "Hey kernel, open this file for me"
- `read()` - "Give me data from this file"
- `fork()` - "Make a copy of my process"
- `pipe()` - "Create a communication channel"
- `wait()` - "Tell me when my child process dies"

2. 🏃 What is a Process?

A process is a running program.

Every Process Has:



1. Process ID (PID)

Unique number: 1234

2. Memory Space

Stack

Heap

Data

Code (Text)

3. File Descriptor Table

[0] → stdin

[1] → stdout

[2] → stderr

[3] → opened files...

4. Process State

Running / Sleeping / Zombie

5. Parent Process ID (PPID)

Who created me?

Process Hierarchy

Every process (except PID 1) has a parent:

init (PID 1)

└─ bash (PID 500)

└─ vim (PID 1000)

└─ gcc (PID 1001)

└─ a.out (PID 1002)

Check your processes:

- `ps aux` - List all processes
- `pstree` - Show process tree
- `echo $$` - Print your shell's PID

3. 🧠 Memory: Why Parent & Child DON'T Share

🚩 CRITICAL FACT:

Parent and child do NOT share memory by default!

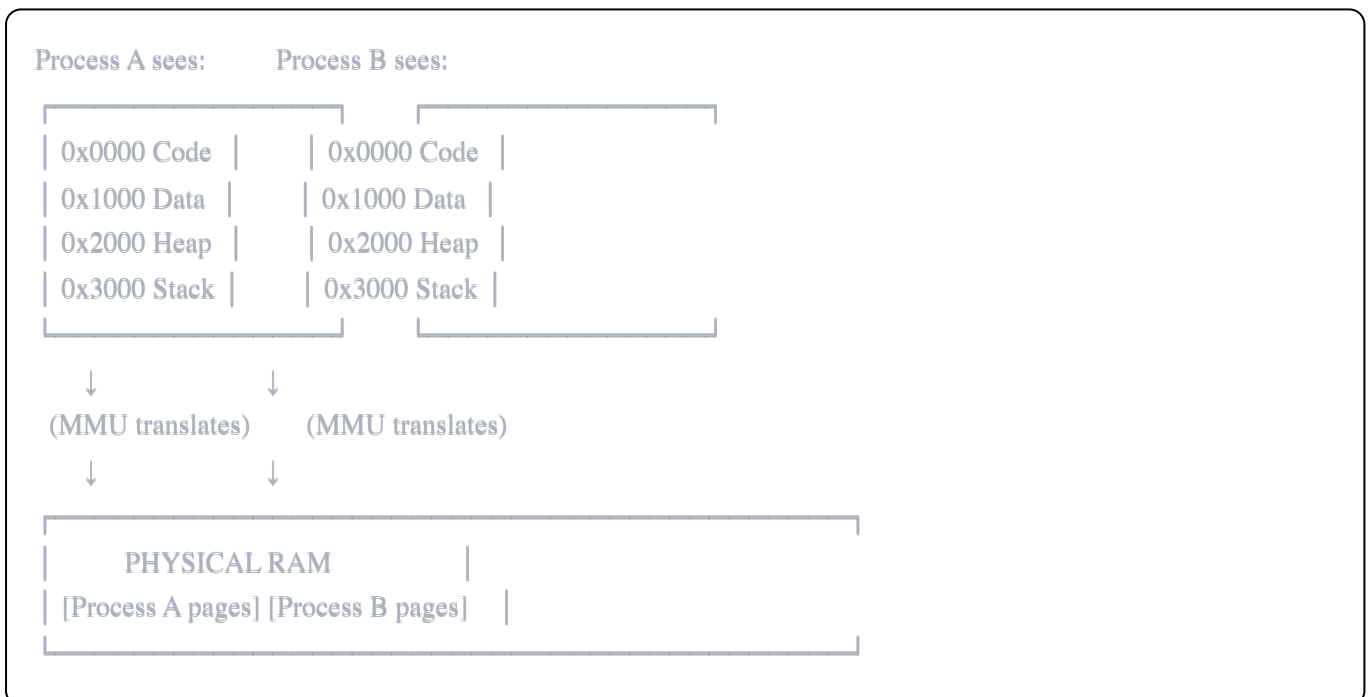


Variables are COPIED, not shared!

Example:

When child changes `x = 20`, parent's `x` stays `10`. Why? Each process has its own virtual memory space.

Virtual Memory (Simplified)



The MMU (Memory Management Unit) translates virtual addresses to physical addresses.

This is why we need IPC (Inter-Process Communication) like pipes!

4. 📁 File Descriptors: Your Gateway to Everything

What is a File Descriptor?

A file descriptor (fd) is a small integer that represents an open file (or pipe, or socket, or device).

Think of it like a ticket number at a restaurant:

- You get a number (file descriptor)
- You use that number to get your order (read/write data)
- The kitchen (kernel) keeps track of what each number means

Standard File Descriptors

Every process starts with three open file descriptors:

```
#define STDIN_FILENO 0 // Standard input (keyboard)
#define STDOUT_FILENO 1 // Standard output (screen)
#define STDERR_FILENO 2 // Standard error (screen)
```

Visual:

Process:

FD Table		
0	stdin	← Keyboard input
1	stdout	← Normal output
2	stderr	← Error messages

Opening Files Gives You New FDs

When you open a file, you get the next available file descriptor:

```
int fd = open("file.txt", O_RDONLY);
// fd is now 3 (first available)

int fd2 = open("data.txt", O_WRONLY);
// fd2 is now 4
```

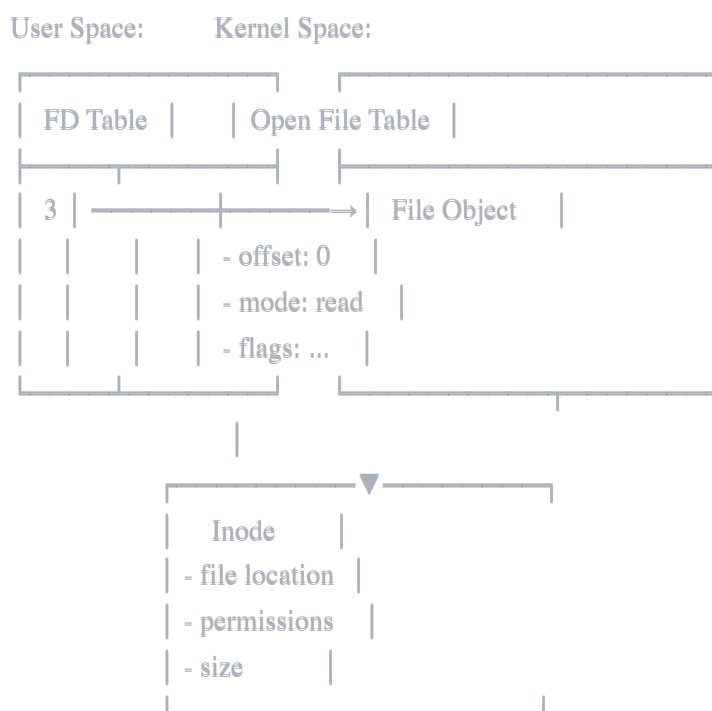
Process FD Table:

0	stdin	
1	stdout	
2	stderr	
3	file.txt	← Your file
4	data.txt	← Another file

File Descriptors Point to Kernel Structures

Three levels of indirection:

1. **FD** (integer in your program)
2. **File Object** (kernel structure with offset, mode)
3. **Inode** (actual file data on disk)



PART 2: THE FORK STORY

5. 🧐 What fork() Really Does

Function Signature

c

```
#include <unistd.h>
pid_t fork(void);
```

What Happens

`fork()` creates a new process by duplicating the calling process.

BEFORE `fork()`:

Parent Process	
PID: 1000	
int x = 5;	
int fd = 3;	

`pid = fork();` ← Magic happens here!

AFTER `fork()`:

Parent Process		Child Process	
PID: 1000		PID: 1001	
int x = 5;		int x = 5;	(copied!)
int fd = 3;		int fd = 3;	(copied!)
fork() = 1001		fork() = 0	(different return!)

Return Values

This is the clever part:

c


```
pid_t pid = fork();

if (pid < 0) {
    // ERROR: fork failed

} else if (pid == 0) {
    // CHILD PROCESS
    // fork() returned 0

} else {
    // PARENT PROCESS
    // fork() returned child's PID
}
```

Why different return values?

- Parent needs to know child's PID (to wait for it, send signals, etc.)
 - Child doesn't need to know its own PID (it can call `getpid()`)
 - This lets you run different code in each process!
-

6. 📄 Copy-on-Write: The Memory Trick

The Problem

If `fork()` copied ALL memory immediately:

- Large programs would be SLOW
- Waste of memory if child just calls `exec()`

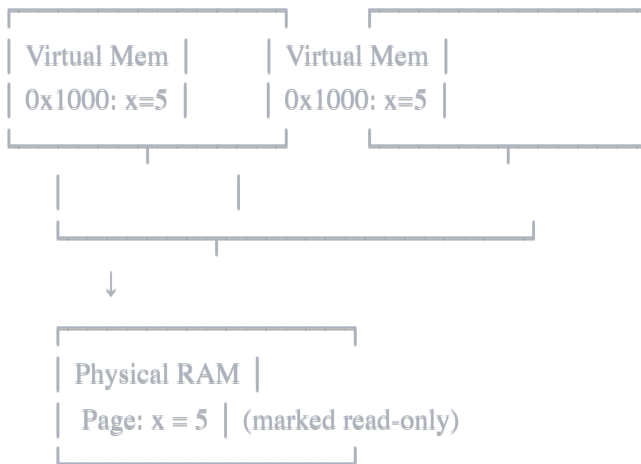
The Solution: Copy-on-Write (COW)

IMMEDIATELY AFTER `fork()`:

Parent & child SHARE the same physical pages (read-only)

Parent Process:

Child Process:

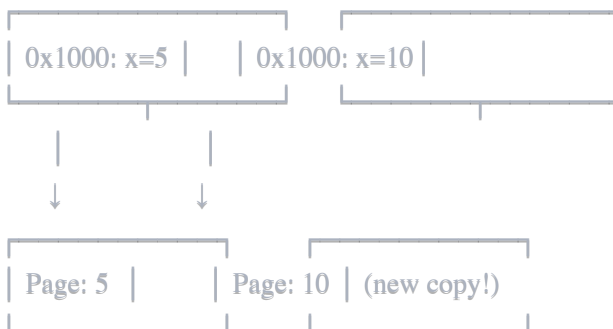


WHEN CHILD WRITES (x = 10):

Now the kernel copies the page!

Parent:

Child:

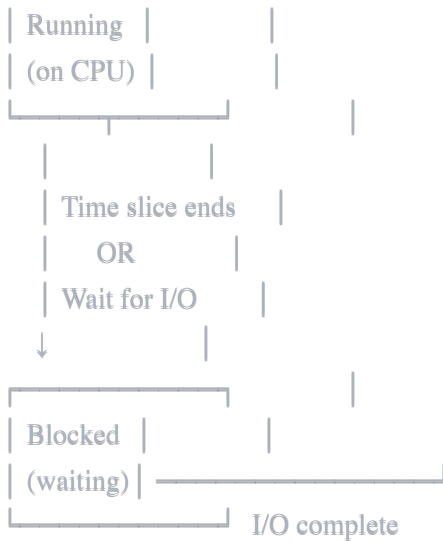


This makes fork() fast!

7. 🔄 Process States & The Kernel Scheduler

Process States





The Scheduler

The kernel's scheduler decides which process runs:

CPU Cores: [Core 0] [Core 1]

Ready Queue:



Every ~10ms (time slice):

- Scheduler picks next process
- Context switch happens
- Process gets to run

Context Switch

Context Switch = Save old process state, load new process state

What gets saved:

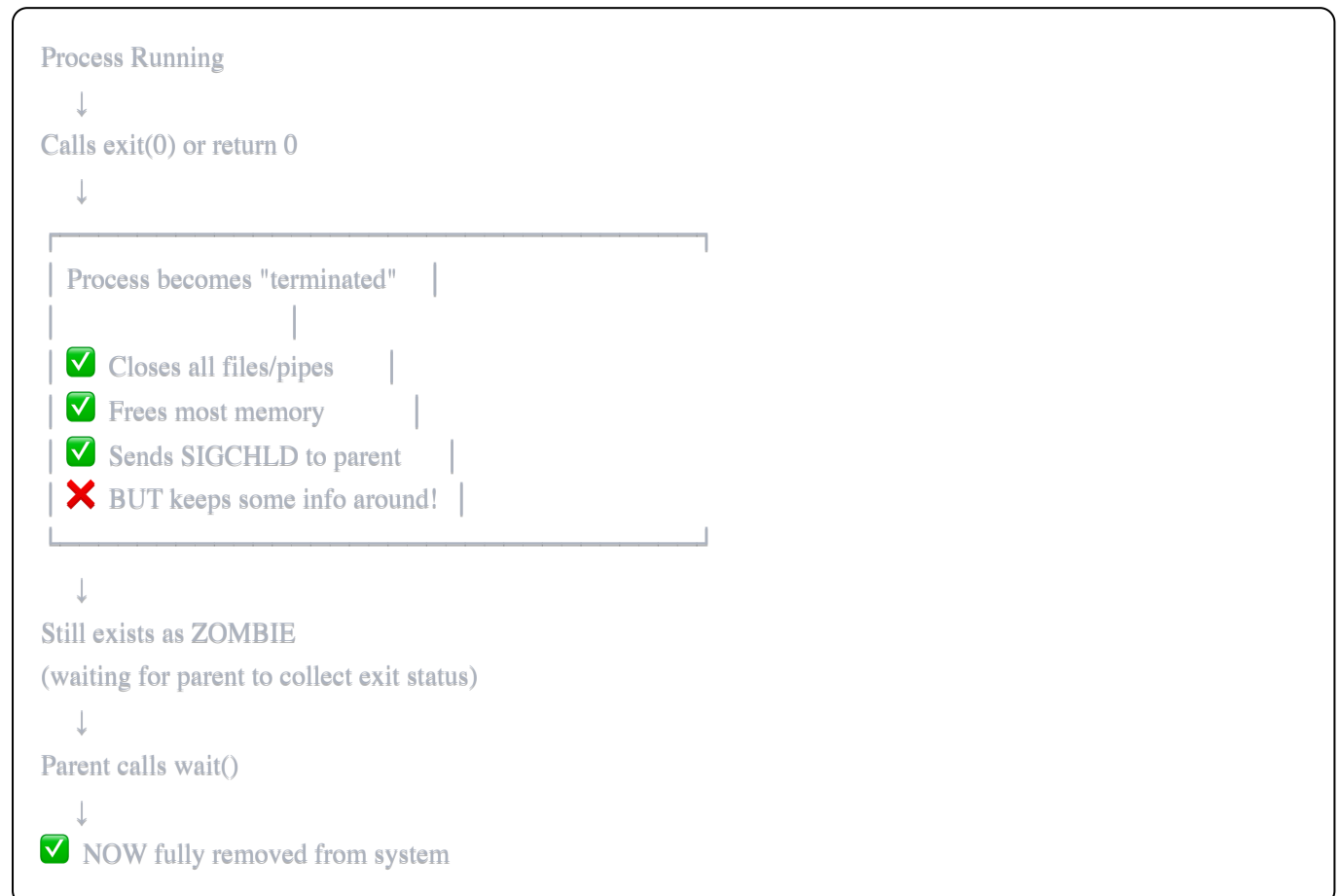
- CPU registers
- Program counter (where we are in code)
- Stack pointer
- Memory mappings

PART 3: PROCESS LIFECYCLE - ZOMBIES & ORPHANS

8. 🦴 What Happens When a Process Dies?

When a process finishes (returns from `main()` or calls `exit()`), it doesn't just disappear!

The Death Process



Key Point: A process can't completely die until its parent acknowledges its death!

9. 🇫🇷 Process Exit Status (The Final Message)

Every process has a "last will and testament" - its **exit status**.

What is Exit Status?

c

```
int main() {
    return 0; // ← This is the exit status!
}

// Same as:
exit(0);
```

Exit status = a number (0-255) that tells the parent how things went

Exit Status	Meaning
0	= Success! Everything's good
1-255	= Something went wrong (Each program defines what specific numbers mean)

Example:

```
c
if (fd == -1) {
    return 1; // Tell parent "I failed!"
}
return 0; // Tell parent "I succeeded!"
```

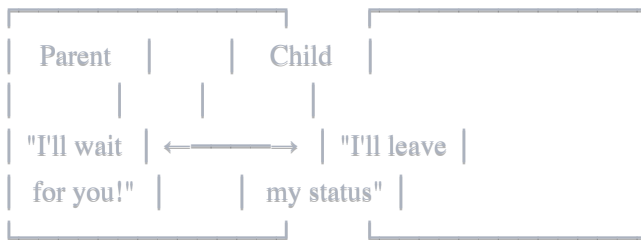
10. 🤝 The Parent-Child Contract

The Deal:

- **Child promises:** "When I die, I'll leave my exit status for you"
- **Parent promises:** "I'll collect your exit status with wait()"

If parent breaks promise → ZOMBIE! If parent dies first → ORPHAN!

Parent & Child Agreement:



11. 🧟 Zombie Processes (The Undead!)

What is a Zombie?

Zombie = A dead process that hasn't been "reaped" by its parent yet

Think of it like:

- You finish an exam and hand it in
- You can't leave until the teacher picks it up
- You're stuck in "zombie" state - done but not gone!

What's happening:

Time 0:

Parent (PID 100) - Running

Child (PID 101) - Running

Time 1s:

Parent (PID 100) - Running (sleeping)

Child (PID 101) - ZOMBIE 🧟

(done, but parent hasn't collected exit status)

Process Table:

PID	State	Parent	Exit Val	
100	Sleep	-	-	
101	ZOMBIE	100	0	← Taking up space!

How to Spot Zombies

```
bash
```

```
ps aux | grep Z
```

```
# or
```

```
ps aux | grep defunct
```

Output:

```
user 101 0.0 0.0 0 0 ? Z 10:00 0:00 [child] <defunct>
```

```
      ↑  
    ZOMBIE!
```

Why Zombies Are Bad

❌ **Waste process table slots** (limited resource!) ❌ **Can't be killed** (already dead!) ❌ **Stay until parent collects or parent dies**

The only cure: wait()!

12. 🏠 Orphan Processes (Home Alone)

What is an Orphan?

Orphan = A process whose parent died before it did

Like when your parent leaves the party before you're ready to go!

What happens:

Time 0:

Parent (PID 100) - Running

Child (PID 101) - Running (PPID = 100)

Time 1s:

Parent (PID 100) - DEAD 💀

Child (PID 101) - Running (PPID = 100... wait what?)

Time 1.1s:

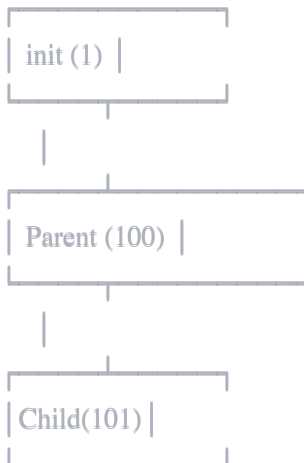
Kernel notices: "Child 101's parent is dead!"

Kernel assigns new parent: init (PID 1)

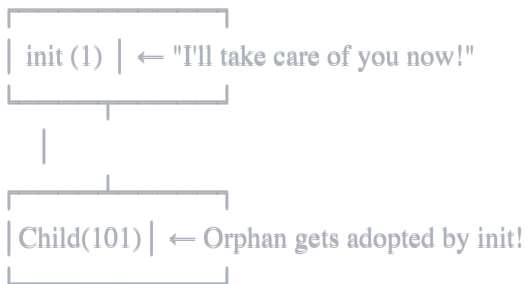
Child (PID 101) - Running (PPID = 1) ← Adopted!

The Adoption Process

Original Family:



After Parent Dies:



Why Orphans Are (Usually) OK

✅ **init/systemd automatically calls wait() on its adopted children** ✅ **No zombie risk** - they get cleaned up
✅ **Can continue running normally**

But: If you *expected* the parent to be around (to collect results, etc.), orphaning is a bug!

13. 🛡️ How to Avoid Zombies

Solution 1: Call wait() (Always!)

c


```

if (fork() == 0) {
    // Child does work
    exit(0);
} else {
    // Parent MUST call wait()
    wait(NULL); // ← Prevents zombie!
}

```

Solution 2: Ignore SIGCHLD

```

c

#include <signal.h>

signal(SIGCHLD, SIG_IGN); // "I don't care about children"
// Now children auto-cleanup, no wait() needed!

fork(); // Child will auto-reap when done

```

Solution 3: Handle SIGCHLD

```

c

#include <signal.h>

void sigchld_handler(int sig) {
    while (waitpid(-1, NULL, WNOHANG) > 0); // Reap all dead children
}

int main() {
    signal(SIGCHLD, sigchld_handler);
    // Now safe to fork multiple children!
}

```

PART 4: THE wait() FAMILY

14. 🕒 wait() - The Basic Version

Function Signature

```

c

```

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

```
pid_t wait(int *status);
```

What It Does

wait() does THREE things:

1. **Blocks** (sleeps) until one of your children dies
2. **Collects** the child's exit status
3. **Removes** the zombie from the process table

Return Value

```
c
pid_t child_pid = wait(&status);

if (child_pid == -1) {
    // ERROR (maybe no children?)
} else {
    // child_pid = PID of the child that died
}
```

Timeline:

Time 0s:

Parent: "Waiting for child..."

Parent: calls wait() → BLOCKS 🤔

Child: running...

Time 2s:

Child: exit(42) → becomes ZOMBIE 🧟

Time 2.0001s:

Kernel: "Hey parent, your child died!"

Parent: wakes up from wait()

Parent: collects status (42)

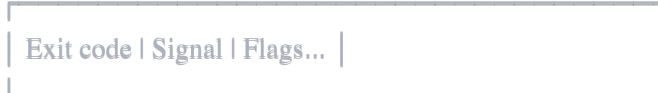
Child: FULLY DEAD ✅ (reaped)

The Status Argument

```
c
int status; // Storage for child's exit info
wait(&status); // Pass address so wait() can fill it
```

What's in `status`?

status is a 32-bit int with encoded information:



DON'T read it directly!

Use macros instead:

- WEXITSTATUS(status) → get exit code
- WIFEXITED(status) → did it exit normally?
- WIFSIGNALED(status) → was it killed by signal?

Ignoring Status

```
c
wait(NULL); // "I don't care about the exit status"
            // Just clean up the zombie!
```

15. 🎯 waitpid() - More Control

Function Signature

```
c
pid_t waitpid(pid_t pid, int *status, int options);
```

Why waitpid()?

wait() limitations:

- Can't choose WHICH child to wait for
- Always blocks (can't check without waiting)

waitpid() gives you control!

Arguments

```
c  
  
waitpid(pid, &status, options);  
  ↓      ↓      ↓  
which? where? how?
```

Argument 1: `pid` (which child?)

```
pid > 0 → Wait for that specific child  
pid = -1 → Wait for ANY child (like wait())  
pid = 0 → Wait for any child in same process group  
pid < -1 → Wait for any child in process group |pid|
```

Argument 2: `status` (same as wait())

```
c  
  
int status;  
waitpid(child_pid, &status, 0);  
// or  
waitpid(child_pid, NULL, 0); // Ignore status
```

Argument 3: `options` (blocking behavior)

```
0 → Block until child dies (default)  
WNOHANG → Don't block! Return immediately  
         (Returns 0 if no child dead yet)  
WUNTRACED → Also return if child stopped (not just dead)
```

Non-blocking check:

```
c
```

```
pid_t result = waitpid(-1, &status, WNOHANG);

if (result == 0) {
    printf("No child finished yet\n");
} else if (result > 0) {
    printf("Child %d finished!\n", result);
} else {
    perror("waitpid error");
}
```

Wait for all children:

```
c

int num_children = 5;
for (int i = 0; i < num_children; i++) {
    fork(); // Create children
}

// Parent waits for all
while (waitpid(-1, NULL, 0) > 0) {
    printf("A child finished\n");
}
```

16. 📖 WEXITSTATUS & Friends (Reading Exit Status)

The Macros

After calling `wait(&status)`, use these to decode `status`:

```
c
```

```

int status;
wait(&status);

// Did it exit normally?
if (WIFEXITED(status)) {
    int exit_code = WEXITSTATUS(status);
    printf("Exit code: %d\n", exit_code); // 0-255
}

// Was it killed by a signal?
if (WIFSIGNALED(status)) {
    int signal = WTERMSIG(status);
    printf("Killed by signal: %d\n", signal);
}

// Was it stopped (not dead)?
if (WIFSTOPPED(status)) {
    int signal = WSTOPSIG(status);
    printf("Stopped by signal: %d\n", signal);
}

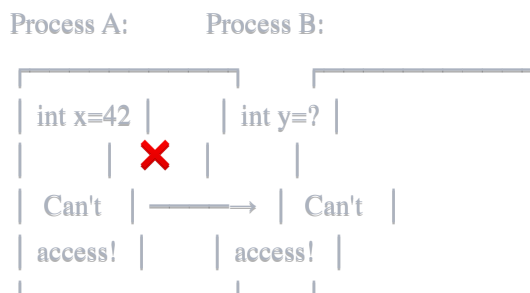
```

PART 5: ENTER PIPES

17. 🗨️ Why We Need IPC (Inter-Process Communication)

The Problem

Process A has data → Process B needs that data
 But they DON'T share memory!



Solutions (IPC Mechanisms)

Linux provides several ways for processes to communicate:

IPC Mechanisms in Linux



Pipes (Anonymous)

- One-way communication
- Parent ↔ Child only



FIFOs (Named Pipes)

- Like pipes, but any processes
- Has a name in filesystem



Shared Memory

- Fastest (no copying)
- Need synchronization



Message Queues

- Structured messages
- Persistent



Sockets

- Network communication
- Bi-directional



Signals

- Simple notifications
- Limited data

We focus on pipes.

18. 🔧 What is a Pipe? (Core Idea)

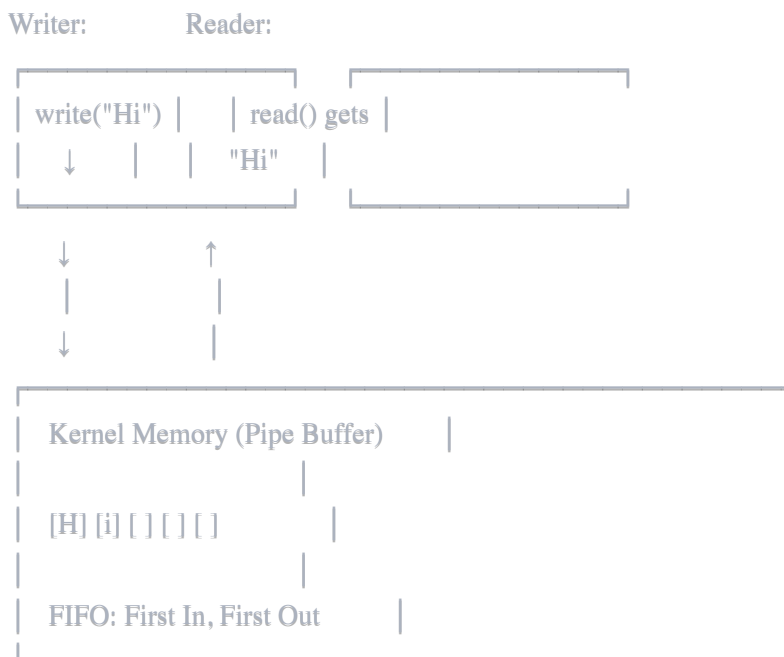
Definition

A **kernel-managed buffer** that allows one process to write bytes and another process to read those bytes, in FIFO order.

Visual

Think of it like:

[Writer Process] —→ [KERNEL PIPE BUFFER] —→ [Reader Process]



Properties

✅ **Unidirectional:** Data flows ONE way only (writer → reader) ✅ **FIFO:** First byte written is first byte read ✅ **Byte stream:** No message boundaries ✅ **Buffered:** Has kernel buffer (usually 64KB) ✅ **Blocking:** Read blocks if empty, write blocks if full ✅ **Lives in kernel memory:** Fast, no disk I/O

19. 📝 Creating a Pipe: pipe()

Function

```
c
#include <unistd.h>

int pipe(int pipefd[2]);
```

What It Does

```
c
```



```
int fd[2];  
pipe(fd);
```

// After this:

// fd[0] → read end

// fd[1] → write end

Kernel Creates Three Things:

1. A pipe buffer in kernel memory
2. Two file descriptors
3. Two file objects (one for reading, one for writing)

Visual:

User