

Processes

Ryan Baker

October 15, 2026

Contents

1	Programs and Processes	3
1.1	Looking at Processes With <code>ps</code>	3
2	What a Process Keeps	5
2.1	Execution State	5
2.2	Existing Is Not Executing	6
2.3	An Address Space of Its Own	7
3	Parents and Children	7
3.1	The Process Tree	8
3.2	A Starting Context	9
4	Creating a Process, Replacing a Program	9
4.1	One Call, Two Continuations	10
4.2	What Was Copied?	11
4.3	A New Program, the Same Process	12
5	Termination and Waiting	13
5.1	Collecting the Result	14
5.2	Zombies	14
6	Exit Status in the Shell	15
6.1	A Result, Not a History	15
6.2	What Nonzero Means	16
6.3	Using the Result to Choose What Happens Next	17
7	Foreground and Background	18
7.1	Waiting Later	19
7.2	Stopping Is Different from Backgrounding	19
	Try It Yourself	21
A	Optional: Launching a Child in C	22
B	Answers and Further Reading	25

1 Programs and Processes

A program can sit on disk indefinitely without ever doing anything. Such a program can rightly be considered a waste of bits, but it's a perfectly valid arrangement nonetheless.

An executable file contains instructions and data needed to run a program. However, it cannot tell us anything about a particular execution of said program: Which instruction is running right now? What values are currently stored in its variables? Which files has it opened?

A **process** is a program in execution, together with the state and resources that belong to that execution. A process has a place in the program, values in memory, and resources such as open files. The program describes what to do; the process is a particular instance of doing.

1.1 Looking at Processes With `ps`

To see which processes are present, we use the `ps` command, short for *process status*:

Terminal

```
> ps
  PID TTY          TIME CMD
 6172 ttys001    0:00.05 zsh
 6191 ttys001    0:00.00 ps
```

Each row of output (below the header) represents one process. The first is our shell, `zsh`. The second is `ps` itself: the command we ran to list processes is itself a process, so it appears in its own output. By the time the shell displays its next prompt, the process running `ps` has finished.

The columns describe different aspects of each process.

The `PID` column shows its **process ID**: a number that distinguishes it from every other process currently on the system. Running `ps` again will typically show a different `PID` for `ps`, since each invocation is a separate process, while the shell's `PID` remains unchanged:

Terminal

```
> ps
  PID TTY          TIME CMD
 6172 ttys001    0:00.05 zsh
 6199 ttys001    0:00.00 ps
```

After a process is gone, its PID may be reused by another process.

The `CMD` (command) column helps us recognize what each process is doing. It typically shows the program's name and command-line arguments.

Aside

Command-line arguments can appear in process listings, so they are a poor place to put passwords or other sensitive information. Some programs overwrite their argument strings to hide these values from subsequent listings, but the original values may still be visible before the overwrite.

The `TTY` column identifies the terminal associated with a process, if it has one. The `TIME` column reports accumulated CPU time, rather than the elapsed time since the process started.

By default, `ps` shows only a selection of processes. To include all processes visible to `ps`, we can use the `-e` option:

Terminal

```
> ps -e
```

We can also choose which columns to display using `-o`:

Terminal

```
> ps -e -o pid,comm
```

Here, we have requested to see only the process ID and command name, in that order. The `comm` column shows the command name without its arguments.

The options accepted by `ps` vary between systems. For details about your system's version, consult `man ps`.

2 What a Process Keeps

Calling a process a “running program” is useful, provided we do not take *running* too literally. A process can exist while doing no work at all. Is such a process *running*?

Suppose two processes are sharing a CPU, each counting from one to a million. Because they’re sharing the CPU, one counts for a while, then the other gets a turn, and so on. When the first resumes, it should continue where it left off, rather than start over at one.

For this to work, the operating system must preserve each process’s progress, including its current count and the instruction it should execute next. That requirement gives us a useful way to understand what belongs to a process.

2.1 Execution State

First, we need a position in the instruction stream. The CPU tracks this using a register commonly called the **program counter** or **instruction pointer**. A **register** is a small storage location built into the CPU.

Remembering the instruction address is necessary, but not sufficient. Other registers may hold intermediate results, memory contains variables and allocated objects, and a stack records information needed for active function calls.

To switch execution to something else, the operating system saves the relevant CPU state and later restores it. This is part of a **context switch**. The process’s memory remains associated with it while it is away from the CPU; the system does not copy the entire memory contents into a little record on every switch.

The kernel maintains bookkeeping for each process. Textbooks often call this a **process control block**, or **PCB**. Think of it as the kernel’s record of a process and its connections to other structures, rather than one box containing every byte the process owns.

Information	What it lets the system track
Identity	Which process this is and who its parent is.
Execution context	Where execution can continue and which register values it needs.
Memory mappings	Which address space belongs to the process.
Resources	Open files, a working directory, and other operating-system state.
Scheduling and accounting	Whether execution is eligible to proceed and how resources have been used.

Real kernels divide this information differently. There is no portable PCB layout to memorize. The important relationship is that an execution has an identity, and the system can find the state associated with that identity.

2.2 Existing Is Not Executing

A process may be ready to continue but waiting for CPU time. Alternatively, it may be unable to continue until something happens: input arrives, a timer expires, or a child process finishes.

These are different situations. The first is **runnable**; the second is **blocked**, often described as *sleeping*. A scheduler chooses among runnable work. It does not make an unavailable input arrive sooner by repeatedly giving the waiting process a turn.

`sleep 3` asks the system to wait for a timer. It does not normally spend three seconds looping and checking a clock. Similarly, a shell waiting for an external command can block while the command makes progress.

With one available execution slot, different processes can make progress by taking turns: **concurrency**. With several execution slots, work can also proceed at the same instant: **parallelism**. Neither guarantees that two independently executing processes will reach particular instructions in a predictable order.

There is one refinement worth making here. A process can contain multiple **threads**: separate streams of execution sharing the process's address space and many of its resources. Each thread needs its own execution context and stack. On modern systems, the scheduler ordinarily schedules threads. For the rest of this chapter, imagine a process with one thread unless stated

otherwise. That lets us discuss the lifecycle without prematurely writing a second chapter inside this one.

2.3 An Address Space of Its Own

Saving an instruction address would be useless if, while we were away, another process could casually replace the instruction at that address.

On the systems considered here, a process ordinarily uses **virtual addresses**. The system translates those addresses using mappings associated with the process. Its **virtual address space** is the range of addresses it can name; only some portions are mapped for valid use.

The same numeric address can therefore mean different things in different processes:

Process	Virtual address	May refer to
Editor	0x400000	Part of the editor's instructions.
Compiler	0x400000	Part of the compiler's instructions.

These are illustrative addresses, not locations you should expect to find. The point is that an address needs a context. The number alone does not identify a unique byte across all processes.

Memory mappings also carry access permissions. Together, translation and protection normally prevent one process from directly reading or overwriting another's private memory. An independent address space does not mean that every byte must occupy a unique physical memory location: read-only program code can be shared, and processes can explicitly arrange shared memory.

For now, keep the distinction between an independent *view* of memory and an entirely separate physical copy of everything. We will need it when we create a child process.

3 Parents and Children

Ordinary user processes are created by existing processes. The creator is the **parent**; the new process is its **child**. A child's **parent process ID**, or **PPID**, identifies its current parent.

Return to our shell and ask `ps` to include that relationship:

Terminal

```
> ps -o pid,ppid,comm
  PID  PPID  COMM
 6172  6140  zsh
 6230  6172  ps
```

The parent ID of `ps` matches the shell's PID. The shell created the process that is running `ps`. The shell also has a parent, which is not necessarily included in this selection.

You can inspect your shell explicitly. In an ordinary interactive shell, `$$` expands to its PID:

Terminal

```
> ps -p "$$" -o pid,ppid,comm
  PID  PPID  COMM
 6172  6140  zsh
```

The `-p` option selects a PID. To investigate the next generation up, use the PPID displayed by your own output as the argument to another `ps -p` command.

3.1 The Process Tree

Parent-child relationships give processes a tree structure. A shell might create a compiler, which in turn creates other tools. Being a child does not prevent a process from becoming a parent.

At the root of the ordinary user-process hierarchy is a special process with PID 1. Traditionally called `init`, it is started during system startup rather than by another ordinary user process. On many Linux systems its program is `systemd`; on macOS it is `launchd`. Inspect your own system with:

Terminal

```
> ps -p 1 -o pid,ppid,comm
```

A container may show a different program as its PID 1. Nor should you expect every desktop to have exactly the same chain from PID 1 to the terminal to the shell. Service managers and session machinery add their own structure.

The parent-child relationship also does not require the parent to remain alive forever. If a parent exits first, surviving children are reparented to a process responsible for adopting them. This is often PID 1; Linux can also use a designated *subreaper*. An **orphan** is a child whose original parent has ended. It need not have stopped working.

3.2 A Starting Context

A new process needs more than a program name. It needs a working directory, an environment, access to resources, and an identity under which to operate.

Much of that context can come from its parent. This is why commands launched from a shell usually begin in the shell's working directory and receive the environment the shell prepared. We do not need to tell every command which directory we are standing in.

Inheritance does not generally create a live connection between all of these values. If a child changes its own working directory, its parent stays where it was. If a child changes an environment variable, that change does not travel back up the process tree.

Other inherited resources *can* refer to shared objects. A child may inherit access to the same terminal or open file as its parent. We will study that distinction when discussing file descriptors. For now, “inherits” means that the child receives a starting context, not that every resource becomes a private duplicate.

4 Creating a Process, Replacing a Program

It would be reasonable to design a single operation that means “create a process running this program with these settings.” Unix provides an interface of that kind, `posix_spawn`. But its classic process-creation model separates two operations:

- `fork` creates a child process based on the calling process.

- `exec` replaces the program running in an existing process.

Separating these operations gives the child an opportunity to adjust its inherited context before starting a different program. To understand what happens when a shell launches a command, we will first follow this model. Actual shells may use other interfaces or optimizations.

4.1 One Call, Two Continuations

The C interface to `fork` looks almost suspiciously small:

Creating a child

```
pid_t result = fork();
```

`pid_t` is the type used for process IDs. On success, there are now two processes, each continuing just after this call. The child does not begin again at the top of the source file, and it does not start by running a different executable.

This is the peculiar part: one process calls `fork`, but two processes return from it.

Each process receives its own return value, which tells it what happened:

Value of <code>result</code>	Meaning in this process
Positive	We are the parent; the value is the new child's PID.
Zero	We are the child.
-1	Creation failed; we are still the parent, and no child was created.

Zero is a marker telling the child which branch to take. It is *not* the child's PID. The child can obtain its actual PID using `getpid()`.

Different work in each process

```
if (result == -1) {  
    // The original process handles the failure.  
} else if (result == 0) {  
    // The child continues here.  
} else {  
    // The parent continues here.  
}
```

The two branches execute in different processes. They are not two branches taken in succession by one execution of the `if` statement. Unless we arrange synchronization, we should not assume that the parent branch runs before the child branch, or vice versa.

4.2 What Was Copied?

The child's private memory initially has the same contents as the parent's. A variable that held `10` before the `fork` initially holds `10` in both processes. If the child assigns `20` to its copy, the parent's copy still holds `10`.

That behavior does not require eagerly copying every physical memory page. Modern Unix-like systems commonly use **copy-on-write**: parent and child initially share physical backing for private pages, with the system arranging a separate copy when a write requires it. The observable result is still independent private memory. Explicitly shared mappings are a different case.

Nor does “copy” mean that every field is identical. The child needs a new PID, its PPID points to the parent, and its resource accounting begins afresh. Its inherited open file descriptors can refer to the same underlying objects as the parent's descriptors. A `fork` creates another process; it does not duplicate the outside world.

Our examples use a single-threaded process. Forking a multithreaded process has additional restrictions: only the calling thread continues in the child, even though the address space contains state belonging to the other threads. That is an important systems-programming topic, but it is not needed to follow this lifecycle.

4.3 A New Program, the Same Process

After `fork`, we have two processes running the same program. To make the child run `sleep`, it can use a function from the `exec` family:

Replacing the child's program

```
execlp("sleep", "sleep", "3", (char *) NULL);
```

The first string identifies the executable to find. The second supplies the conventional program name in its argument list; `"3"` is the argument that asks it to wait three seconds. The final null pointer marks the end of the list. This particular function searches `PATH`, which accounts for the `p` in `execlp`.

On success, the process receives a new program image: new instructions, a newly initialized stack, and the other memory needed by the new program. Its old local variables and active function calls are gone. It continues at the new program's entry point, rather than at the next line of the caller.

Its PID remains the same. So does its parent relationship. Its working directory is preserved, and many other operating-system resources can survive the replacement. The arguments and environment for the new program are supplied through the execution interface; `execlp` uses the caller's environment.

A successful `exec` does not return. There is no old function call to return to. If the function returns with `-1`, the replacement failed, and the caller must handle the error. For example, the requested executable might not exist.

The word `exec` describes a family of interfaces, including `execlp`, `execvp`, and `execve`. Their arguments and lookup behavior differ, but the central operation is the same. Likewise, `exec` is useful without `fork`: a process can replace its own program whenever that is what it intends to do.

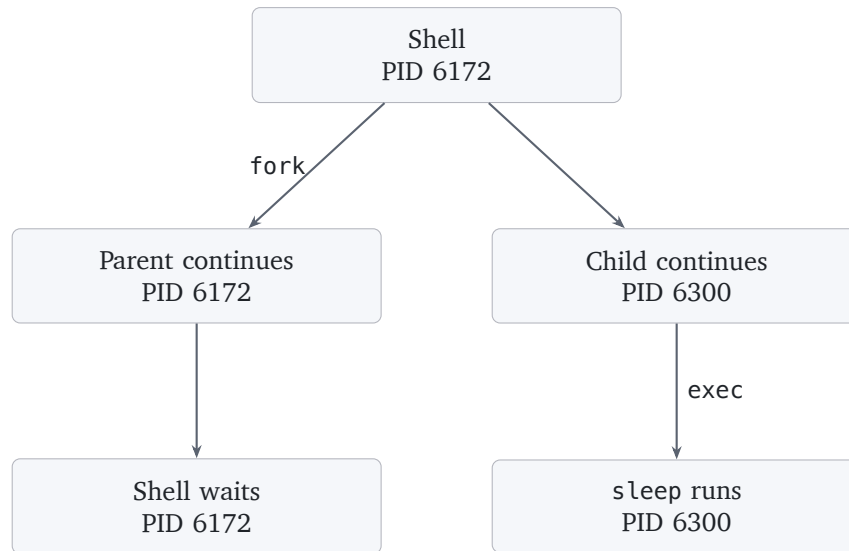


Figure 1: Fork adds a process. Exec changes the program in one process. The IDs are illustrative.

This also explains why a child changing its directory does not move the parent shell. The child has its own working-directory state. To change the directory used by later commands, `cd` must affect the current shell itself, which is why shells provide it as a built-in.

5 Termination and Waiting

Eventually, `sleep` finishes waiting. Its process asks the operating system to terminate it, reporting an **exit status** along the way.

An exit status of zero conventionally indicates success. A nonzero value indicates some other outcome, whose exact meaning belongs to the program. In a C program, returning `0` from `main` reports successful completion:

At the end of `main`

```
return 0;
```

This special relationship applies to returning from `main`. Returning from an ordinary helper function does not terminate the process; it just returns to that function's caller.

A process can also terminate because of a signal. For example, pressing `CTRL+C` in a normal terminal session requests an interrupt of the foreground job. The default response to that signal is termination, though programs can handle it differently. Signals deserve their own treatment; for now, distinguish an ordinary exit from termination caused by a signal.

5.1 Collecting the Result

The parent often needs to know when a child finishes and how it ended. The `wait` family of interfaces provides that information. For example, `waitpid` can wait for a particular child PID.

If the child has not finished, an ordinary blocking wait lets the parent sleep until a result becomes available. The child continues independently.

If the child has already finished, the parent can collect the result immediately. The result includes information distinguishing a normal exit from termination by a signal; it is more than just the small exit-code number. Appendix A shows how a C program decodes it.

For a simple foreground external command, this gives the shell its familiar behavior: launch the child, wait for it, remember the result, and display another prompt. A shell can also respond when a foreground job stops; it need not wait for termination in every case.

5.2 Zombies

There may be a gap between a child finishing and its parent collecting the result. The kernel must keep enough information to bridge that gap. Otherwise, a fast child could disappear before its parent learned what happened.

A terminated child whose completion record remains available for collection is called a **zombie**. Its program is no longer executing. Most resources, including its user address space, have been released, but it still occupies some kernel bookkeeping resources and retains its PID.

Collecting the completion information is called **reaping** the child. A brief zombie state is a normal part of this model; a growing collection of unreaped children indicates that something is failing to do the bookkeeping.

A zombie is not a process stubbornly refusing to stop. It has already stopped permanently. Sending it another termination signal does not make its par-

ent collect the result. The unfinished work belongs to the parent, or to an adopting reaper if the parent exits.

An orphan and a zombie therefore describe different things. An orphan has lost its original parent and may still be executing. A zombie has terminated and still has a completion record awaiting collection. Unix terminology does occasionally make a routine administrative task sound worse than it is.

6 Exit Status in the Shell

The shell exposes the status of the most recent foreground pipeline through the special parameter `$?`. For the single commands we are using, this is simply the most recent command's status.

Two convenient commands make the convention visible. `true` succeeds and `false` fails, without producing ordinary output:

Terminal

```
> true
> echo "$?"
0
> false
> echo "$?"
1
```

The words `true` and `false` name commands here. They are not special literal values that the shell prints or evaluates as an expression. Shells commonly implement them as built-ins, which also demonstrates that a command can report a status without creating a new external process.

Zero means success because the status convention reserves one value for success and leaves other values available to distinguish unsuccessful outcomes. This convention is separate from the use of nonzero values for Boolean truth in languages such as C.

6.1 A Result, Not a History

Inspect `$?` before running another command whose status would replace it:

Terminal

```
> false
> echo "$?"
1
> echo "$?"
0
```

The first echo prints the status from `false`. That echo succeeds, so the second prints zero. Nothing repaired the failed command. We merely asked about a different command the second time.

If you need the value later, save it immediately in a variable:

Terminal

```
> false
> saved_status=$?
> echo "Doing something else"
Doing something else
> echo "$saved_status"
1
```

The assignment itself also has a status, but the expansion on its right-hand side captures the previous value first. We use `saved_status` rather than `status`, which has a special meaning in Zsh.

6.2 What Nonzero Means

An exit status is a compact result intended for other programs. Text output is a different channel. A program can print nothing and fail, or print a long explanation and succeed. Reading the terminal is not how the shell determines the result.

Nor does every nonzero value mean that the program malfunctioned. A search command can use one value to mean “no match” and another to mean that the search itself could not be performed. The manual for that command defines the distinction.

There are also shell conventions: a command that cannot be found produces status 127; one that is found but cannot be executed produces 126. The shell can therefore report failure even when it never successfully starts the

requested program. Programs can use these numbers themselves too, so the number alone is not a complete diagnosis.

For ordinary Unix exit codes, the value available through the usual wait status is limited to eight bits, giving values from 0 to 255. Do not use an exit code to return an arbitrary integer result or a paragraph of data. Output streams are better suited to that work.

6.3 Using the Result to Choose What Happens Next

The `&&` operator runs its right-hand command only if its left-hand command succeeds:

Terminal

```
> true && echo "That worked"
That worked
> false && echo "That worked"
```

The `||` operator does the complementary test: it runs the right-hand command only if the left-hand command returns a nonzero status:

Terminal

```
> false || echo "That did not work"
That did not work
> true || echo "That did not work"
```

The shell waits for the left-hand command's result before deciding. This is **conditional execution**; the second command may never run. For example, `cd /tmp && pwd` prints the working directory only if the directory change succeeds.

`&&` and `||` have equal precedence in the shell and are evaluated from left to right. Avoid treating `a && b || c` as a general-purpose “if a, then b, otherwise c.” If a succeeds but b fails, c runs too. A later chapter on shell scripting will introduce explicit `if` statements for decisions that need more structure.

7 Foreground and Background

So far, the shell has waited for each external command before giving us another prompt. That is a choice about how it manages the command, not a requirement of process creation.

Add a single `&` to run a command asynchronously:

Terminal

```
> sleep 30 &  
[1] 6401  
> echo "The shell is available"  
The shell is available
```

The prompt returns while `sleep` is still waiting. The shell and its child now have overlapping lifetimes in which either can make progress. The ampersand did not make `sleep` complete faster. It changed whether the shell waits before accepting another command.

The bracketed number is a shell **job number**. The other number is a PID. They belong to different naming systems: job numbers identify jobs managed by this shell; PIDs identify processes in the operating system. A job can contain several processes, as happens with a pipeline. Our example contains one.

Use `jobs` to see the current shell's jobs:

Terminal

```
> jobs  
[1]+  Running                  sleep 30 &
```

The exact decoration varies between Bash and Zsh. Also, “Running” here means the job has not stopped or finished. Its process may currently be asleep, waiting for CPU time, or actually executing. Job-control terminology describes a different level of the system from the scheduler’s moment-to-moment state.

7.1 Waiting Later

For a simple background command, `#!` gives the PID associated with the most recently started asynchronous command. Save it if you intend to refer to that child after starting other work:

Terminal

```
> sleep 3 &
[2] 6410
> child_pid=$!
> wait "$child_pid"
> echo "$?"
0
```

The shell's `wait` built-in waits for that child and reports its status. If the child has already finished and the shell has retained its result, the built-in can return immediately. The shell may have already reaped the kernel's child record and cached the status itself.

By contrast, inspecting `$?` immediately after `sleep 3 &` does not tell you how `sleep` eventually ended. At that point, it may not have ended at all. Use `wait` when you need its completion result. This built-in works with children known to the shell; it is not a general tool for waiting on any PID from `ps`.

7.2 Stopping Is Different from Backgrounding

In an ordinary interactive session with job control, `CTRL+Z` requests that the foreground job stop. Try this with a fresh command:

Terminal

```
> sleep 60
```

Press `CTRL+Z` before the command finishes. The shell should report that the job stopped and display another prompt. Its process still exists, with enough state to continue later. It is not a zombie: its execution has been suspended, not terminated.

To resume the current job in the foreground, enter:

Terminal

```
> fg
```

Once it is in the foreground, press `CTRL+C` to end this `sleep` command if it has not finished. Time spent stopped may count toward the timer, so resuming it does not promise another full sixty-second wait.

Alternatively, `bg` resumes a stopped job in the background. A background job is allowed to proceed; a stopped job is not. It is easy to confuse these because both leave you looking at a shell prompt.

Foreground status also matters to terminal input. In the usual job-control arrangement, the foreground process group receives terminal-generated signals such as the interrupt from `CTRL+C`. Background processes do not automatically get ordinary keyboard input; a background job attempting to read from the terminal is normally stopped. Background output can still appear in the terminal, however. The `&` operator does not make a command quiet.

Following One Command

We can now account for the unremarkable-looking command `sleep 3`. The shell prepares its arguments and finds the executable. In the classic launch model, it creates a child, the child replaces its program with `sleep`, and the shell waits. The child's timer wait lets the system use CPU time elsewhere. When the child exits, the shell collects its result and becomes ready for another command.

The executable file is still on disk. The child process is gone. The parent shell remains, with an exit status recording how the command ended.

Try It Yourself

Predict the answers before checking the explanations in Appendix B.

1. Two terminal windows each run `sleep 30`. Are they using one process or two? Must they use two separate copies of the executable file on disk?
2. A shell has PID 7000. It forks a child with PID 7001, and the child successfully executes `sleep`. What are the child's PID and PPID after `exec`? Does the shell become `sleep` too?
3. Immediately before a fork, a local integer variable contains 4. The child changes its copy to 9. Assuming ordinary private memory, what value does the parent see? Does copy-on-write change that answer?
4. What does each echo print?

Terminal

```
> false
> saved_status=$?
> true
> echo "$?"
> echo "$saved_status"
```

5. Does the final command run? Explain which status causes the decision.

Terminal

```
> true && false || echo "This runs"
```

6. A process appears in `ps` as a zombie. Would resuming it with `fg` make it continue? Who still has work to do?
7. Start `sleep 3 &`, save `!` in `child_pid`, and wait for that PID. Explain why the status after `wait` is useful in a way that the status immediately after `&` is not.

A Optional: Launching a Child in C

This program performs the classic sequence directly. It creates one child, runs `sleep 1` there, and collects its completion status. It assumes a C compiler and the ordinary `sleep` command on `PATH`.

Save both parts of the following listing, in order, as one file named `launch.c`. The first part creates the child and handles program replacement:

launch.c: creation and replacement

```
#define _POSIX_C_SOURCE 200809L
#include <errno.h>
#include <stdio.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <unistd.h>

int main()
{
    pid_t child = fork();
    if (child == -1) {
        perror("fork");
        return 1;
    }

    if (child == 0) {
        execlp("sleep", "sleep", "1", (char *) NULL);
        const char message[] = "could not execute sleep\n";
        (void) write(STDERR_FILENO, message,
                     sizeof message - 1);
        _exit(127);
    }
}
```

The code after `execlp` runs only when replacement fails. The child writes a diagnostic and calls `_exit`, which terminates it directly without running inherited C-library exit handlers or flushing copied output buffers. This distinction matters after a fork. The small diagnostic uses `write`; a complete application would also consider failures while reporting an error.

The parent reaches the following continuation. It waits for the PID returned by fork, retrying if a caught signal interrupts the wait:

launch.c: waiting and reporting

```
int status;
pid_t waited;
do {
    waited = waitpid(child, &status, 0);
} while (waited == -1 && errno == EINTR);
if (waited == -1) {
    perror("waitpid");
    return 1;
}
if (WIFEXITED(status)) {
    int code = WEXITSTATUS(status);
    printf("child exited with status %d\n", code);
    return code;
}
if (WIFSIGNALED(status)) {
    printf("child terminated by signal %d\n",
          WTERMSIG(status));
}
return 1;
}
```

status is an encoded wait result, not a plain exit code. WIFEXITED checks whether the child exited normally; only then do we use WEXITSTATUS to retrieve its code. WIFSIGNALED and WTERMSIG handle the distinct case of termination by a signal. Here, “normally” describes the mechanism of termination, not whether the code indicates success.

Compile and run it:

Terminal

```
> cc -std=c11 -Wall -Wextra launch.c -o launch
> ./launch
child exited with status 0
> echo "$?"
0
```

Output appears after about a second. The parent returns the child's ordinary exit code so the outer shell can observe it too. Each process chooses its own result; it is not automatically passed through generations.

To test replacement failure, change the first string in `exec lp` to an absolute path you know does not exist, then rebuild. The child should exit with 127, which the parent reports and returns. The failed `exec` returned -1; our child chose 127.

B Answers and Further Reading

Exercise Answers

1. Two processes. Both can use the same executable file. A program file can support many executions without being duplicated on disk.
2. PID 7001 and PPID 7000, assuming the parent remains alive. Successful execution replaces the child's program while preserving its process identity. The shell continues in PID 7000.
3. The parent sees 4. Copy-on-write is an implementation technique that preserves the independent behavior of private memory; it does not turn the child's assignment into an assignment in the parent.
4. The first echo prints 0, the status from `true`. The second prints the saved value 1, captured immediately after `false`.
5. Yes. `true` succeeds, so `false` runs. Its nonzero result causes the command after `||` to run.
6. No. The process has terminated and cannot resume. Its parent, or an adopting reaper, must collect its completion information. A stopped live job is different.
7. `wait` reports the selected child's completion result. The status immediately after starting the asynchronous command does not report the child's eventual outcome.

Reading the Interfaces

Your machine's manual pages describe its implementation. Start with `man ps`, `man 2 fork`, `man 2 execve`, and `man 2 waitpid`. The 2 selects the system-call section, distinguishing these interfaces from similarly named shell commands. Bash documents job control and its `wait` built-in in its own manual; for Zsh, see `man zshbuiltins` and `man zshmisc`.

These primary references provide further detail. Linux-specific behavior need not apply to other Unix implementations.

- [POSIX.1-2024: fork](#) and [Linux fork\(2\)](#): return values, inheritance, and copy-on-write.
- [Linux execve\(2\)](#) and [exec\(3\)](#): process-image replacement and the C interfaces.
- [Linux posix_spawn\(3\)](#): an interface combining process creation with program execution.

- [Linux `wait\(2\)`](#): completion records, zombies, adoption, and status decoding.
- [Bash: Exit Status](#), [Lists of Commands](#), and [Job Control Basics](#): shell results and foreground/background behavior.
- [Linux `ps\(1\)`](#), [`sched\(7\)`](#), and [`pthread\(7\)`](#): process inspection, scheduling, and the thread refinement.
- [Linux kernel documentation: Memory Management Concepts](#): virtual addresses, mappings, and physical memory.