Chapter 3b: xv6 System Call Flow: fork()

Gene Cooperman

Copyright © 2026 Gene Cooperman, gene@ccs.neu.edu
This text may be copied as long as the copyright notice remains and no text is modified.

Contents

3b xv6 System Call Flow: fork() 1
3b.1 User-space `fork()` calls `sys_fork()` in the kernel . . . 1
3b.2 What `sys_fork()` does in the kernel – the real work . . . 3
3b.3 Two returns from one call . . . 4
3b.4 Visual Summary . . . 4
3b.5 Key Takeaways . . . 5
3b.6 Appendix: REVIEW: Process States in `proc.h` . . . 5

3b xv6 System Call Flow: fork()

This is a walkthrough of what happens when a user program calls `fork()` in xv6, tracing from user space down into the kernel and back.

3b.1 User-space `fork()` calls `sys_fork()` in the kernel

Most of the action here takes place in `usys.S` (to enter the kernel), `trap.c` (with the help of `trapasm.S`), and `syscall.c`, with the help of `syscall.h` for `SYS_fork`), to finally calls `sys_fork()` in `sysproc.c` (system calls for the process table).

a. User program calls `fork()`

A user program (like `sh.c`) calls `fork("file", O_RDONLY)`. But there's no C implementation of `fork()` in user space – it's just a thin wrapper that triggers a trap into the kernel.

b. `usys.S` – the user-space stub (same as libc.so in modern Linux)

The file `usys.S` (generated from macros) provides the actual `fork` symbol. It does something like:

```
fork:
  movl $SYS_fork, %eax     # load syscall number 1 into %eax
  int $T_SYSCALL           # trap instruction (int 64)
  ret
```

In RISC-V, this would be:

```
fork:
  mv SYS_fork, a0     # load syscall number 15 into argument register a0
  ecall $T_SYSCALL    # trap instruction (int 64)
  ret                 # As usual, the return value is also placed in a0.
```

From `syscall.h`, **`SYS_fork` is a macro, defined as 1**. The `ecall` instruction causes a privilege-level switch from user mode to kernel mode.

In an Intel CPU (not RISC-V), the processor looks up interrupt vector 64 (`T_SYSCALL`) in the IDT, which points to code in `trapasm.S`. This code:

- Pushes registers onto the stack to build a **trapframe** (saving the user's state)
- Calls `trap()` in `trap.c`

The CPU is now in kernel mode, instead of user mode. From inside the operating system kernel, the call to `trap()` inspects `tf->trapno`. Since it's `T_SYSCALL`, it calls **`syscall()`**.

c. `trapasm.S` -> `trap.c` – entering the kernel

The processor looks up interrupt vector 64 (`T_SYSCALL`) in the IDT, which points to code in `trapasm.S`. This code:

- Pushes registers onto the user's stack to build a **trapframe** (saving the user's state)
- Calls `trap()` in `trap.c`

`trap()` inspects `tf->trapno`. Since it's `T_SYSCALL`, it calls **`syscall()`**.

d. `syscall.c` – dispatching the call

The `syscall()` function (defined at line 127 of `syscall.c`) does the following:

- Reads the syscall number from `proc->tf->eax` (which the user put there – value 1)
- Uses it as an index into a function pointer table:

```
static int (*syscalls[])(void) = {
  ...
  [SYS_fork] sys_fork,
  ...
};
```

- Calls **`sys_fork()`**
- Stores the return value back into `proc->tf->eax` (so the user program receives it after returning)

NOTE: A note on the `[SYS_fork]` syntax:

This is a **C99 designated initializer**. In a normal array initializer, elements are assigned in order – index 0, 1, 2, etc. With a designated initializer, you can explicitly specify *which index* to assign. Since `syscall.h` defines `SYS_fork` as 15, writing `[SYS_fork] = sys_fork` means "put the function pointer `sys_fork` at index 15 of the array." All unspecified entries default to `NULL` (0).

This is a clean way to build a dispatch table where the array index matches the syscall number, without needing to manually count positions or leave explicit placeholder entries. It also means the table stays correct even if the syscall numbers are reordered or have gaps. At runtime, the dispatch is just:

```
syscalls[num]()    // e.g., syscalls[15]() calls sys_fork
```

If a user passes a bad syscall number, the entry will be `NULL`, which the kernel checks for before calling.

e. Extracting user-space arguments

We are lucky that `fork()` does not take any arguments. But if we were tracing another system call, like `sys_open()` in `sysfile.c`, then it would need to retrieve the arguments the user passed. It can't just read them directly – they're on the user stack, and the CPU is in kernel mode, not user mode. So it would use the helper functions defined in `syscall.c`:

- **`argstr(0, &path)`** – fetches the first argument (the filename string). Internally this calls `argint` -> `fetchint` to get the pointer, then `fetchstr` to safely copy the string from user memory, checking that it's within the process's address space.
- **`argint(1, &omode)`** – fetches the second argument (the open mode flags).

f. Summary of the Action So Far

To review what we've seen up to now, the user calls `fork()`. The `usys.S` stub loads `SYS_fork` (value 1) into `%eax` and traps into the kernel. The `syscall()` dispatcher calls **`sys_fork()`** (in `sysproc.c`, line 11), which simply calls **`fork()`** (in `proc.c`, line 129) – where the real work happens.

3b.2 What `sys_fork()` does in the kernel – the real work

`fork()` creates a near-exact copy of the calling process. It proceeds in several steps:

a. Allocate a new process

`fork()` calls `allocproc()`, which scans the `ptable` for a slot with state `UNUSED`. When it finds one, it:

- Sets the state to `EMBRYO` and assigns a new `pid` (from the global `nextpid`)
- Allocates a new **kernel stack** (`KSTACKSIZE` bytes)
- Sets up the kernel stack so that when the scheduler first switches to this process, it will enter `forkret()` and then `trapret`, which will pop the trapframe and "return" to user space

b. Copy the address space

`fork()` calls `copyuvm()` to create a **complete copy** of the parent's page table and all of its user memory. The child gets its own independent physical pages with identical contents. After this point, parent and child have the same data in memory, but modifications by one will not affect the other.

c. Copy the trapframe

The child's trapframe is copied from the parent's:

```
*np->tf = *proc->tf;
```

This is what makes the child resume execution at the same point as the parent – same instruction pointer, same stack pointer, same register values. But one register is changed:

```
np->tf->eax = 0;
```

This is the key trick: **the child's return value is set to 0**, while the parent's return value (set later by `syscall()`) will be the child's pid. This is how the caller distinguishes parent from child:

```
int pid = fork();
if (pid == 0) {
    // child: fork() returned 0
} else {
```

```
    // parent: fork() returned child's pid
}
```

d. Copy the open file table

`fork()` duplicates every open file descriptor:

```
for (i = 0; i < NOFILE; i++)
  if (proc->ofile[i])
    np->ofile[i] = filedup(proc->ofile[i]);
```

`filedup()` does **not** create a new `struct file`. It increments the `ref` count on the existing one. Parent and child now share the same `struct file` entries, which means they share file offsets (as described in the file abstractions document). The child's current working directory (`cwd`) is also shared via `idup()`.

e. Set state to RUNNABLE

Finally, `fork()` sets the child's state to `RUNNABLE` and returns the child's pid to the parent:

```
np->state = RUNNABLE;
return np->pid;
```

The child is now in the `ptable` and eligible to be picked up by the scheduler.

3b.3 Two returns from one call

This is the most unusual aspect of `fork()`: **one call produces two returns**.

- **In the parent**: `syscall()` stores the return value of `fork()` (the child's pid) into `proc->tf->eax` via the normal mechanism. The parent returns to user space through `trapret`, and sees a positive pid in the argument register, `eax`.
- **In the child**: The child has never actually called `syscall()`. Its kernel stack was set up by `allocproc()` to jump into `forkret()` -> `trapret`. The trapframe (copied from the parent) has set the argument register (return value) to `eax = 0`. So when `trapret` pops the trapframe and executes `iret`, the child resumes in user space at the instruction right after `int $T_SYSCALL` – with a return value of 0.

From user space, it looks as if `fork()` was called once and returned twice – once in each process, with different return values.

3b.4 Visual Summary

```
Parent process                          Child process
--------------                          -------------

fork() called
  |
  v
trap into kernel
  |
  v
```

```
proc.c: fork()
  |-- allocproc()         ---------->  new proc: state = EMBRYO
  |-- copyuvm()           ---------->  copy of parent's memory
  |-- copy trapframe      ---------->  *np->tf = *proc->tf
  |                                    np->tf->eax = 0
  |-- filedup() for each fd -------->  shared struct file entries (ref++)
  |-- np->state = RUNNABLE --------->  child is now schedulable
  |
  v
returns child pid
  |
  v
syscall(): tf->eax = pid                  (later, scheduler picks child)
  |                                             |
  v                                             v
trapret -> iret                            forkret -> trapret -> iret
  |                                             |
  v                                             v
user space: fork() == pid (positive)       user space: fork() == 0
```

3b.5 Key Takeaways

- **`fork()` creates a copy, not a new program.** The child is a near-clone of the parent: same code, same data, same open files, same position in the program. To run a different program, the child must call `exec()`.
- **The return value is the only difference** the caller can observe immediately. Parent gets the child's pid; child gets 0.
- **File descriptors are shared, not copied.** After `fork()`, parent and child point to the same `struct file` entries (with incremented `ref` counts). They share offsets. This is a deliberate design choice that makes shell pipelines and redirection work correctly.
- **Memory is copied, not shared.** Unlike file descriptors, the address space is fully duplicated. Changes to variables in one process are invisible to the other. (Modern UNIX systems optimize this with copy-on-write, but xv6 does a full copy for simplicity.)
- **The child never executed `fork()` in the kernel.** Its kernel stack is synthetically constructed by `allocproc()` so that the scheduler's first context switch into the child lands in `forkret()` -> `trapret`, which pops the (copied and modified) trapframe and enters user space as if `fork()` had returned 0.

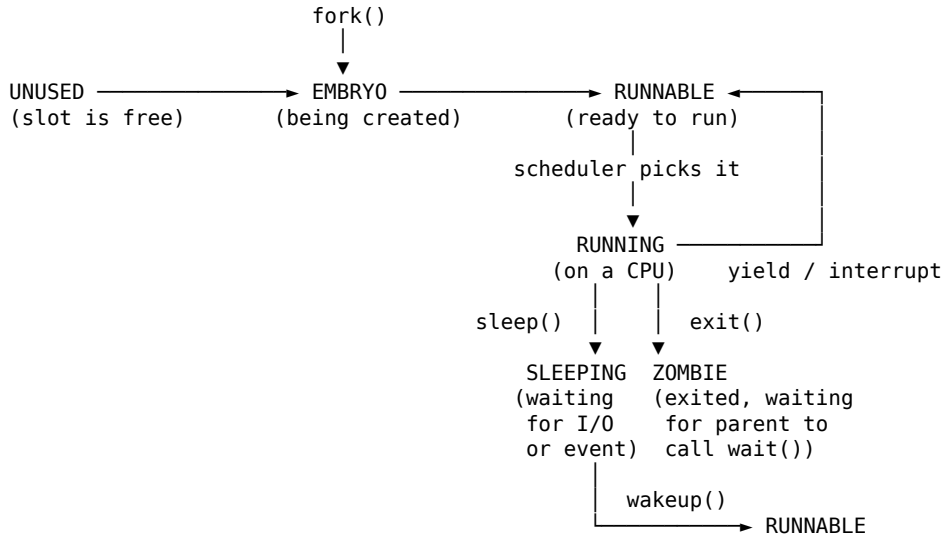
3b.6 Appendix: REVIEW: Process States in `proc.h`

In the original description, we saw the state of the child process transition from `UNUSED` to `EMBRYO` to `RUNNABLE`. In Chapter 3a (3a-xv6-proc-internals.pdf), we saw a section:

`enum procstate` – process life cycle states.

Here is a quick review of those process states in the lifetime of a process, specialized to an example for `fork()`.

The `procstate` enum in `proc.h` defines six states that a process moves through during its lifetime:



State	Purpose
UNUSED	The `ptable` slot is empty and available. `allocproc()` scans for these when creating a new process.
EMBRYO	The process is being set up by `allocproc()` – a pid has been assigned and a kernel stack allocated, but the process is not yet ready to run. This state prevents the scheduler from trying to run a half-initialized process.
RUNNABLE	The process is ready to execute but is not currently on a CPU. It sits in the `ptable` waiting for the `scheduler()` to pick it. A process enters this state when first created (end of `fork()`), when it yields the CPU, or when it is woken up from sleep.
RUNNING	The process is actively executing on a CPU. Only one process per CPU can be in this state. The `scheduler()` sets this state just before calling `swtch()` to transfer control to the process.
SLEEPING	The process is blocked waiting for some event – typically I/O completion (e.g., disk read), a pipe becoming readable/writable, or `wait()`ing for a child to exit. The `sleep()` function sets this state. The process cannot be scheduled until `wakeup()` moves it back to `RUNNABLE`.
ZOMBIE	The process has called `exit()` and is dead, but its entry in the `ptable` has not yet been cleaned up. It remains in this state until its parent calls `wait()`, which reads the exit status, frees the kernel stack and page table, and sets the slot back to `UNUSED`. If the parent never calls `wait()`, the zombie persists – this is a resource leak.

3b.6.1 The life cycle in one sentence

A process is born as `EMBRYO`, becomes `RUNNABLE`, alternates between `RUNNING` and `RUNNABLE` (and possibly `SLEEPING`) as it executes, becomes a `ZOMBIE` when it exits, and finally returns to `UNUSED` when its parent reaps it with `wait()`.

Based on xv6 rev8 (September 1, 2015) – MIT 6.828 teaching operating system.