UNIX Processes

COS 417: Operating Systems

Spring 2025, Princeton University

Core of the UNIX Process API

Core of the UNIX Process API

`fork()`

Duplicate the memory, file descriptors, user, group, etc... of the current process

Core of the UNIX Process API

`fork()`

Duplicate the memory, file descriptors, user, group, etc… of the current process

- What's different?

Core of the UNIX Process API

`fork()`

Duplicate the memory, file descriptors, user, group, etc… of the current process

- What's different?

`wait(int pid)`

Wait for the process with process id `pid` to exit.

Core of the UNIX Process API

`fork()`

Duplicate the memory, file descriptors, user, group, etc… of the current process

- What's different?

`wait(int pid)`

Wait for the process with process id `pid` to exit.

`exec??(char *pathname, ...)`

Replace current program for process with program at `pathname`.

Flexible: Concurrency

```
def divide_and_concur():
  cpus = get_num_cpus()
  big_img = readfile("some_large_image.raw")
  step = big_img.pixels() / cpus
  for i in 0..cpus:
    pid = fork()
    if pid == 0:
      result = apply_filter(big_img[(step * i)..][..step])
      write_to_file("some_large_image_bw.raw", step * i, step)
```

Flexible: Don’t repeat yourself

```
def fancy_web_server(port):
  conf = read_config_file()
  very_expensive_initialization(conf)
  listen_socket = listen(port)
  for i in 0..10:
    pid = fork()
    if pid == 0:
      for connection in listen_socket:
        while req = read_request(connection):
          handle_req(conf, req)
```

Flexible: Run other programs concurrently

```
def simple_shell():
  cmd = input("What do you want to run? ")
  pid = fork()
  if pid == 0:
    exec(cmd)
  else:
    print("I'm still here, just waiting, k?")
    wait(pid)
```

What the `fork()`?

Concurrency

Code re-use

Multi-processing (e.g. a shell)

What the `fork()`?

Concurrency

Threads, e.g. `p-threads`, share memory thus almost always better for parallelism, events better for single-threaded concurrency…

Code re-use

Multi-processing (e.g. a shell)

What the `fork()`?

Concurrency

Threads, e.g. `p-threads`, share memory thus almost always better for parallelism, events better for single-threaded concurrency…

Code re-use

Used to be important because many libraries were not thread safe, so shared libraries not a good option. Mostly fixed now.

Multi-processing (e.g. a shell)

What the `fork()`?

Concurrency

Threads, e.g. `p-threads`, share memory thus almost always better for parallelism, events better for single-threaded concurrency...

Code re-use

Used to be important because many libraries were not thread safe, so shared libraries not a good option. Mostly fixed now.

Multi-processing (e.g. a shell)

Disaster!

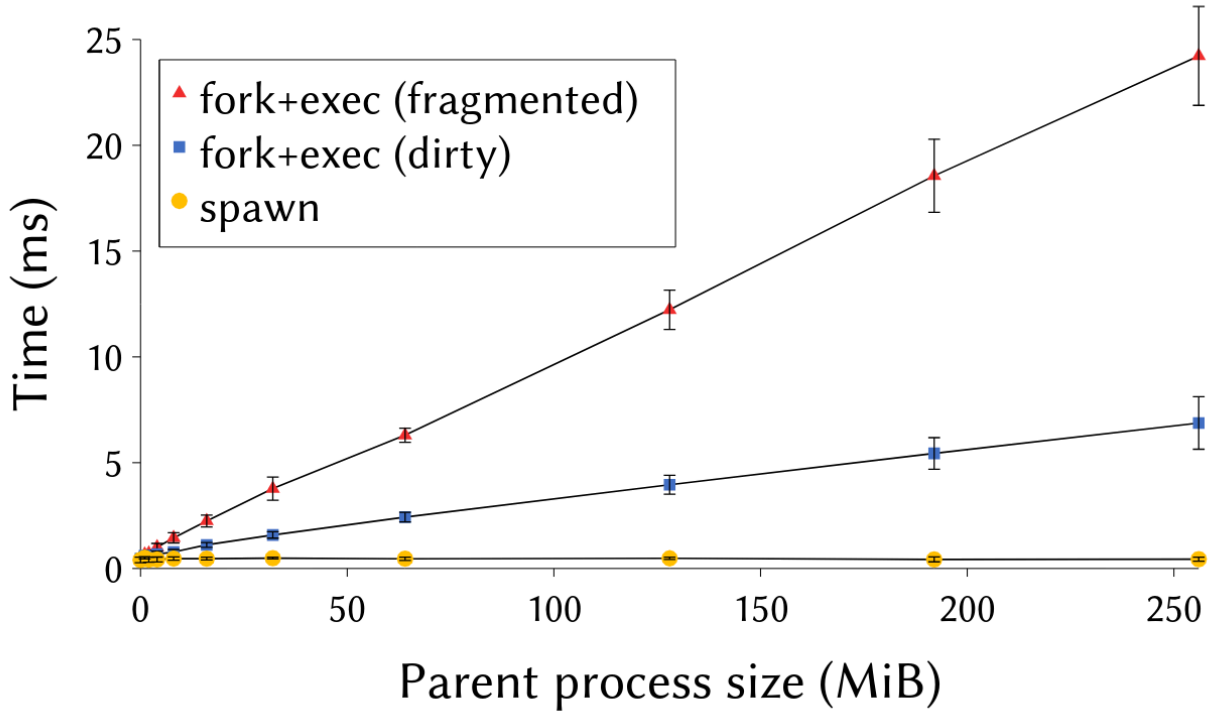


Figure 1: Time to run `fork()` + `exec()` vs. `spawn()`[1]

[1] A `fork()` in the road, Bauman et al.

Why is `fork()` so slow?

It seems reasonable to suppose that it exists in Unix mainly because of the ease with which fork could be implemented without changing much else.

— Dennis Ritchie (1984)

`fork()` on the original UNIX for PDP-7

- Couldn't have two processes in memory
  - No memory translation HW
- Context switch to a different process?
  1. Pause current process
  2. Copy main memory to storage
  3. Overwrite memory with next process's stored state
- For `fork()`, just skip the last step!

`fork()` on the original UNIX for PDP-7

- Couldn't have two processes in memory
  - No memory translation HW
- Context switch to a different process?
  1. Pause current process
  2. Copy main memory to storage
  3. Overwrite memory with next process's stored state
- For `fork()`, just skip the last step!

Simple to implement, fast (relatively)!

`fork()` on the original UNIX for PDP-7

- Memory is small!
  - <= 144 KB
- Memory access is (relatively) fast!
  - 1/2 CPU cycle

`fork()` today

- Memory is enourmous!
  - 1GB for a process is typical
- Memory access is (relatively) *slooooooooowwwww*!
  - 100s of CPU cycles

Two options for implementing:

1. Copy all of memory upfront.
2. Copy memory lazily on write (copy-on-write)
   - But each copy-on-write is *very slooooooooowwwww*

`fork()` + `exec()` today

`fork()`

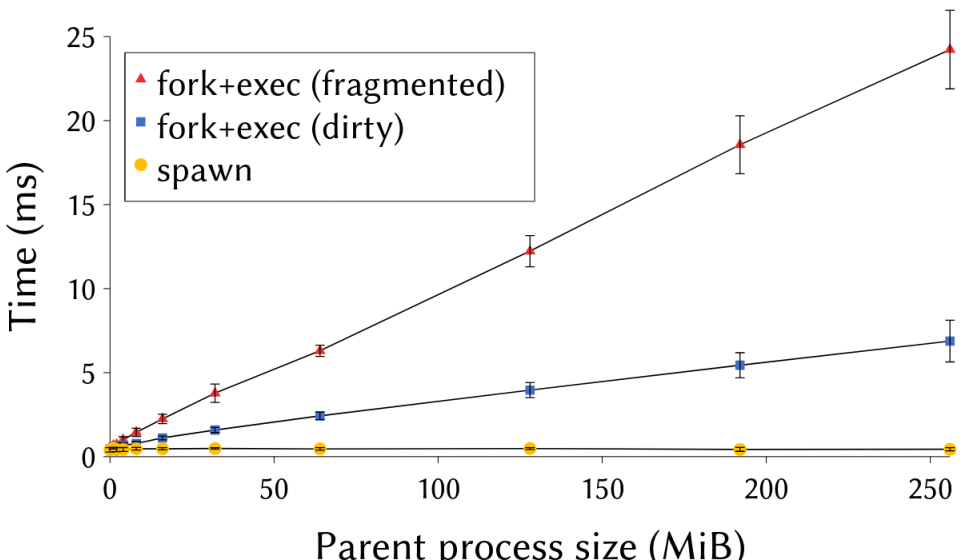
New process points to same memory, marked CoW

`exec()`

In new process immediately overwrites all memory

For each page in memory: incur a page fault; allocate new memory; copy memory; modify page tables

`fork()` + `exec()` vs. `spawn()`



`spawn()`

Create a new, empty process and loads new program into it.

No unnecessary copying, no page faults.

So why the `fork()` is `fork()` still around?

1. 50 years of legacy.
2. No other access to copy-on-write semantics in most operating systems
   - Snapshots & memoization (e.g. Android Zygotes)
   - Asynchronous persistence (e.g. Redis)

So why the `fork()` is `fork()` still around?

1. 50 years of legacy.
2. No other access to copy-on-write semantics in most operating systems
   - Snapshots & memoization (e.g. Android Zygotes)
   - Asynchronous persistence (e.g. Redis)

Consider not using `fork()` in the future.

Your favorite language *probably* doesn't.

More to cover

- file descriptors
- stdout, stderr, stdin
- pipes, how they allow inter process communication
- Linux vs. UNIX philosophy difference