CS 31: Intro to Systems
Processes

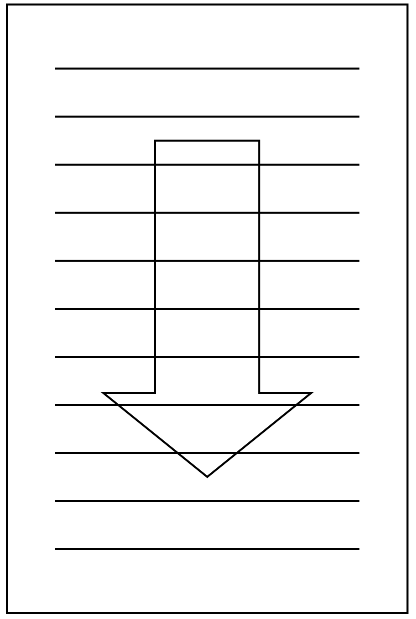
Vasanta Chaganti & Kevin Webb

Swarthmore College

November 16, 2023

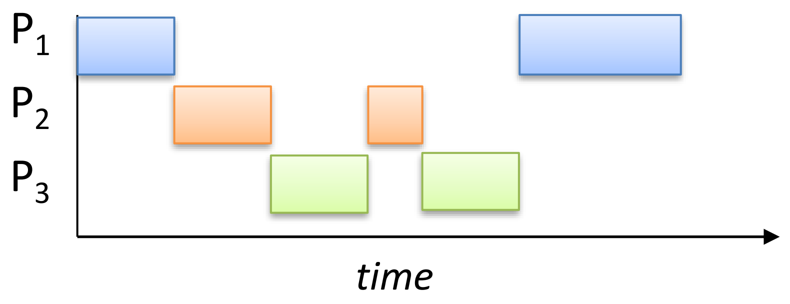
Anatomy of a Process

- Abstraction of a running program
  - a dynamic “program in execution”
- OS keeps track of process state
  - What each process is doing
  - Which one gets to run next
- Basic operations
  - Suspend/resume (context switch)
  - Start (spawn), terminate (kill)

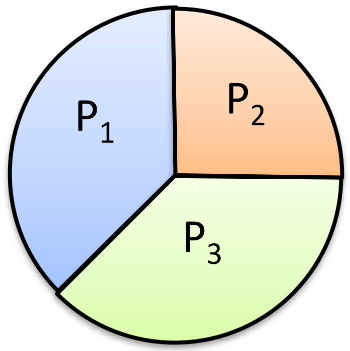


Resource Sharing

CPU: Time



Memory: Space



Reality

- Multiple processes
- Small number of CPUs
- Finite memory

Abstraction

- Process is all alone
- Process is always running
- Process has all the memory

Timesharing: Sharing the CPUs

- Abstraction goal: make every process think it's running on the CPU all the time.
  - Alternatively: If a process was removed from the CPU and then given it back, it shouldn't be able to tell

- Reality: put a process on CPU, let it run for a short time (~10 ms), switch to another, … (<u>context switching</u>)

How is Timesharing Implemented?

- Kernel keeps track of progress of each process
- Characterizes <u>state</u> of process's progress
  - Running: actually making progress, using CPU
  - Ready: able to make progress, but not using CPU
  - Blocked: not able to make progress, can't use CPU
- Kernel selects a ready process, lets it run
  - Eventually, the kernel gets back control
  - Selects another ready process to run, …

Why might a process be blocked (unable to make progress / use CPU)?

A. It’s waiting for another process to do something.

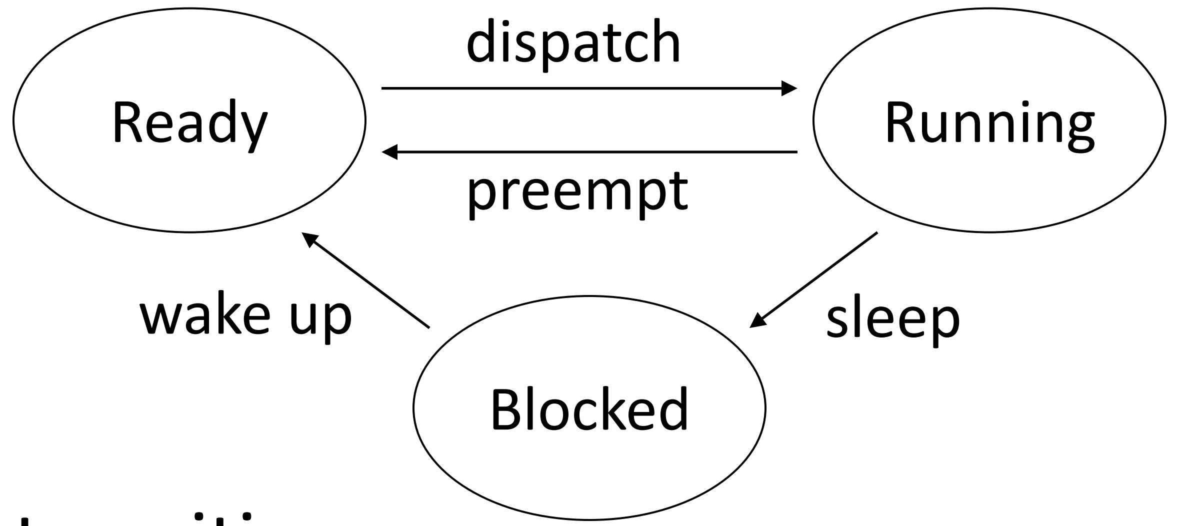
B. It’s waiting for memory to find and return a value.

C. It’s waiting for an I/O device to do something.

D. More than one of the above. (Which ones?)

E. Some other reason(s).

Process State Diagram



- State transitions
  - Dispatch: allocate the CPU to a process
  - Preempt: take away CPU from process
  - Sleep: process gives up CPU to wait for event
  - Wakeup: event occurred, make process ready

Kernel Maintains Process Table

Process ID (PID)	State	Other info
1534	Ready	Saved context, …
34	Running	Memory areas used, …
487	Ready	Saved context, …
9	Blocked	Condition to unblock, …

- List of processes and their states
  - Also sometimes called "process control block (PCB)"
- Other state info includes
  - contents of CPU context
  - areas of memory being used
  - lots of other bookkeeping information

Values of registers in use by process

Multiprogramming

- Given a running process
  - At some point, it needs a resource, e.g., I/O device
  - If resource is busy (or slow), process can’t proceed
  - “Voluntarily” gives up CPU to another process

- Mechanism: Context switching

Context Switching

- Allocating CPU from one process to another
  - First, save context of currently running process
  - Next, load context of next process to run

Context Switching

- Allocating CPU from one process to another
  - First, save context of currently running process
  - Next, load context of next process to run
- Loading the context
  - Load general registers, stack pointer, etc.
  - Load program counter (must be last instruction!)

How a Context Switch Occurs

- Process makes system call (TRAP) or is interrupted by device HW
  - These are the only ways of entering the kernel
- In hardware
  - Switch from user to kernel mode: amplifies power
  - Go to fixed kernel location: interrupt/syscall handler
- In software (in the kernel code)
  - Save context of last-running process
  - Conditionally
    - Select new process from those that are ready
    - Restore context of selected process
  - OS returns control to a process from interrupt/syscall

Time Sharing / Multiprogramming

- Given a running process
  - At some point, it needs a resource, e.g., I/O device
  - If resource is busy (or slow), process can't proceed
  - "Voluntarily" gives up CPU to another process

- Mechanism: Context switching
- Policy: CPU scheduling

The CPU Scheduling Problem

- Given multiple processes, but only one CPU
- How much CPU time does each process get?
- Which process do we run next?

- Possibilities
  - Process keeps CPU till done
  - Each process uses CPU a bit and passes it on
  - Each process gets proportional to what they pay

Which CPU scheduling policy is the best?

A. Processes keep CPU until done (maximize throughput)

B. Processes use a fraction of CPU and pass it on (ensure fairness)

C. Processes receive CPU in proportion to their priority or what they pay (prioritize importance)

D. Other (explain)

There is No Single Best Policy

- Depends on the goals of the system

- Different for…
  - Your personal computer
  - Large time-shared (super) computer
  - Computer controlling a nuclear power plant

- Often have multiple (conflicting) goals

Common Policies

- Details beyond scope of this course (Take OS)

- Different classes of processes
  - Those blessed by administrator (high/low priority)
  - Everything else

Common Policies

- Special class gets special treatment (varies)

- Everything else: *roughly* equal time quantum
  - "Round robin"
  - Give priority boost to processes that frequently perform I/O
  - Why?

- "I/O bound" processes frequently block.
  - If we want them to get equal CPU time, we need to give them the CPU more often.

Linux’s Policy

(You’re not responsible for this.)

- Special “real time” process classes (high prio)

- Other processes:
  - Keep red-black BST of process, organized by how much CPU time they’ve received.
  - Pick the ready with process that has run for the shortest time thus far.
  - Run it, update it’s CPU usage time, add to tree.

- Interactive processes: Usually blocked, low total run time, high priority.

Managing Processes

- Processes created by calling `fork()`
  - "Spawning" a new process

- "Parent" process spawns "Child" process
  - Brutal relationship involving "zombies", "killing" and "reaping". (I'm not making this up!)

- Processes interact with one another by sending signals.

Managing Processes

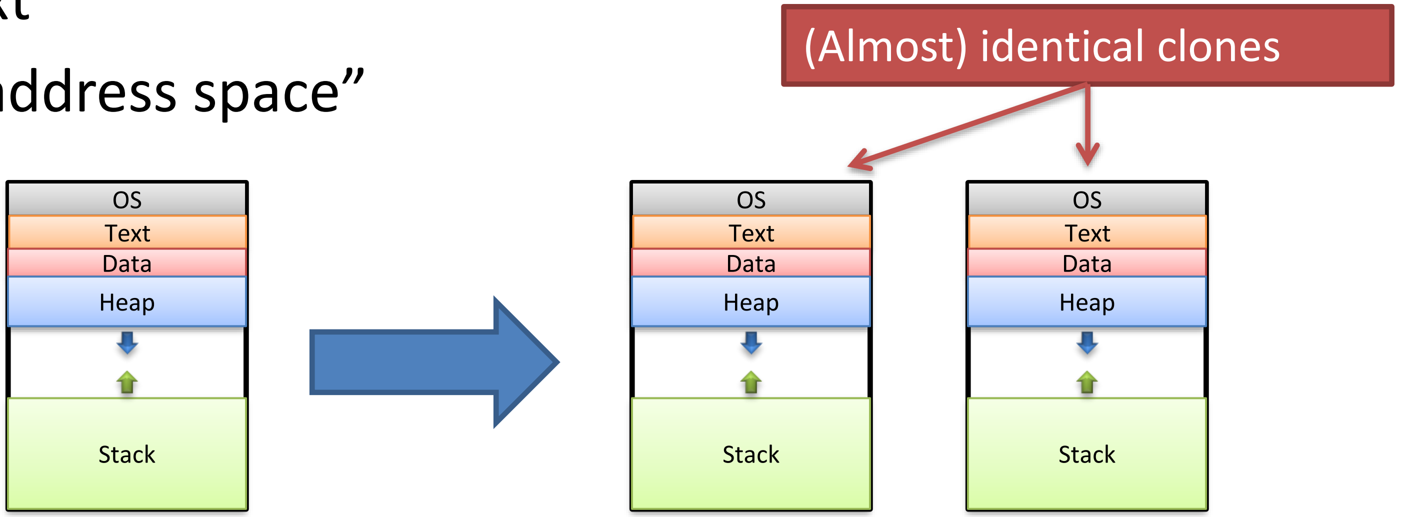
- Given a process, how do we make it execute the program we want?

- Model: fork() a new process, execute program

`fork()`

- System call (function provided by OS kernel)

- Creates a duplicate of the requesting process
  - Process is cloning itself:
    - CPU context
    - Memory "address space"



`fork()` return value

- The two processes are identical in every way, except for the return value of `fork()`.
  - The child gets a return value of 0.
  - The parent gets a return value of child's PID.

```
pid_t pid = fork(); // both continue after call
if (pid == 0) {
   printf("hello from child\n");
} else {
   printf("hello from parent\n");
}
```

Which process executes next? Child? Parent? Some other process?

Up to OS to decide. No guarantees. Don't rely on particular behavior!

How many hello’s will be printed?

```
fork();
printf(“hello”);
if (fork()) {
  printf(“hello”);
}
fork();
printf(“hello”);
```

How many hello’s will be printed?

```
fork();
printf(“hello”);
if (fork()) {
  printf(“hello”);
}
fork();
printf(“hello”);
```

A. 6
B. 8
C. 12
D. 16
E. 18

Common `fork()` usage: Shell

- A “shell” is the program controlling your terminal (e.g., bash).
- It `fork()`s to create new processes, but we don’t want a clone (another shell).
- We want the child to execute some other program: `exec()` family of functions.

`exec()`

- Family of functions (execl, execlp, execv, …).
- Replace the current process with a new one.
- Loads program from disk:
  - Old process is overwritten in memory.
  - Does not return unless error.

Common `fork()` usage: Shell

1. `fork()` child process.

2. `exec()` desired program to replace child's address space.

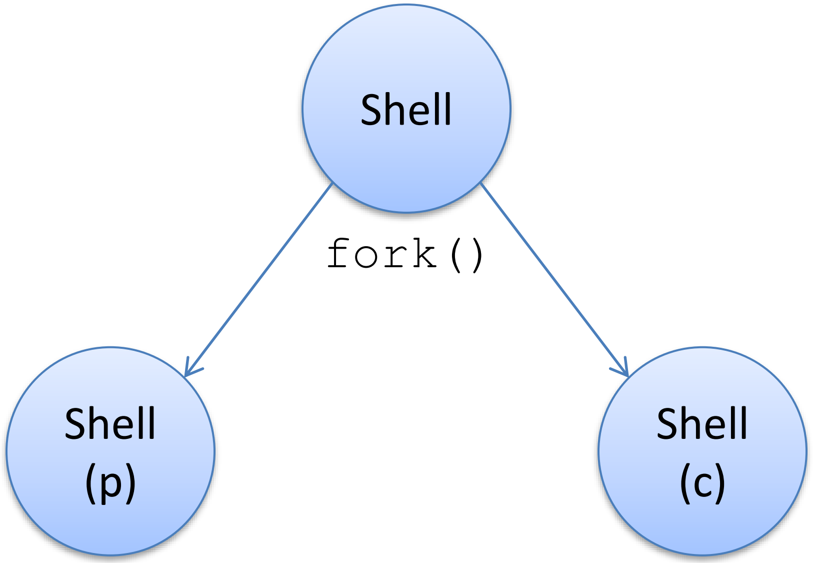
2. `wait()` for child process to terminate.

3. repeat…

The parent and child each do something different next.

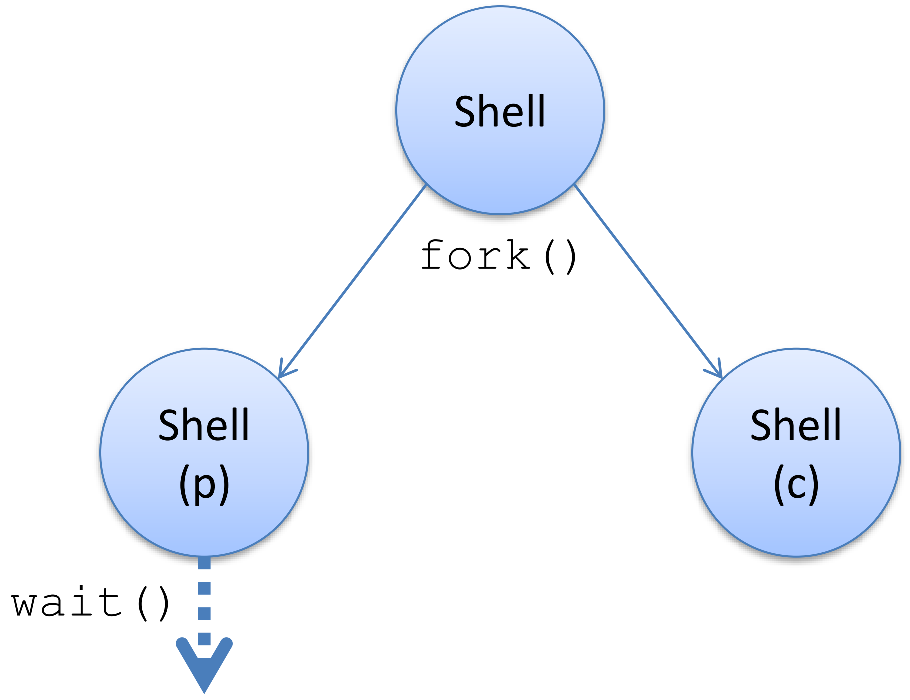
Common `fork()` usage: Shell

`1.` `fork()` child process.



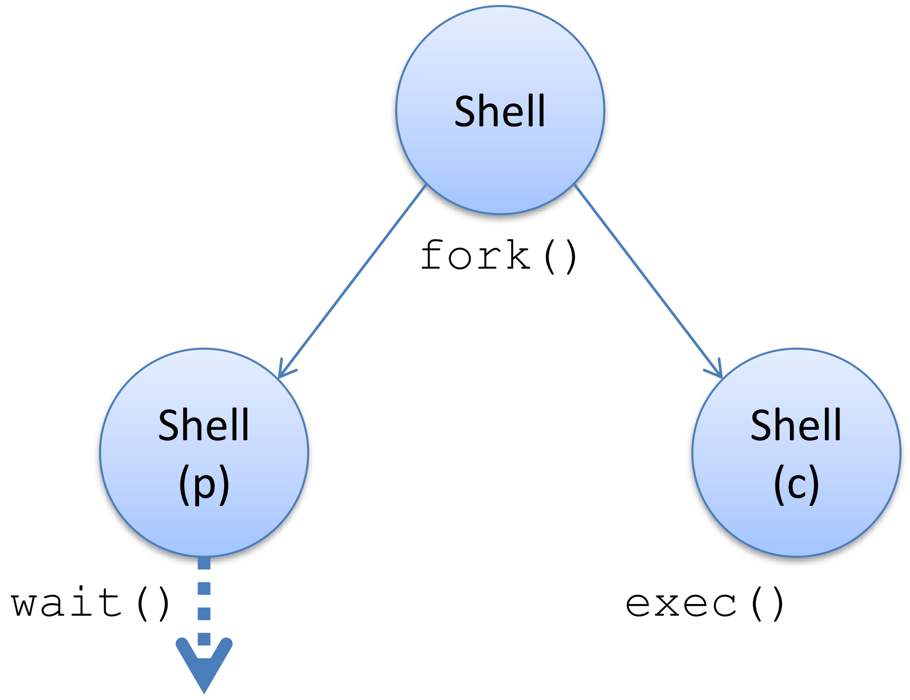
Common `fork()` usage: Shell

2. parent: `wait()` for child to finish



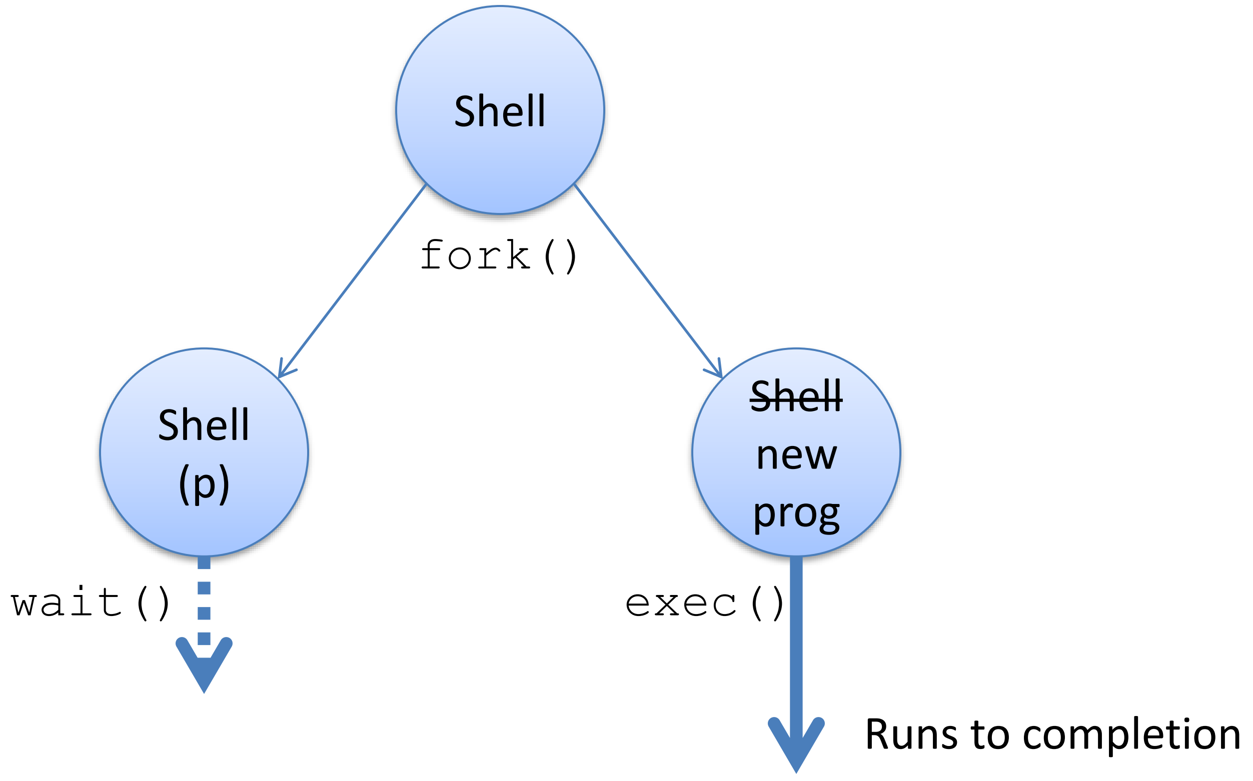
Common `fork()` usage: Shell

2. child: `exec()` user-requested program



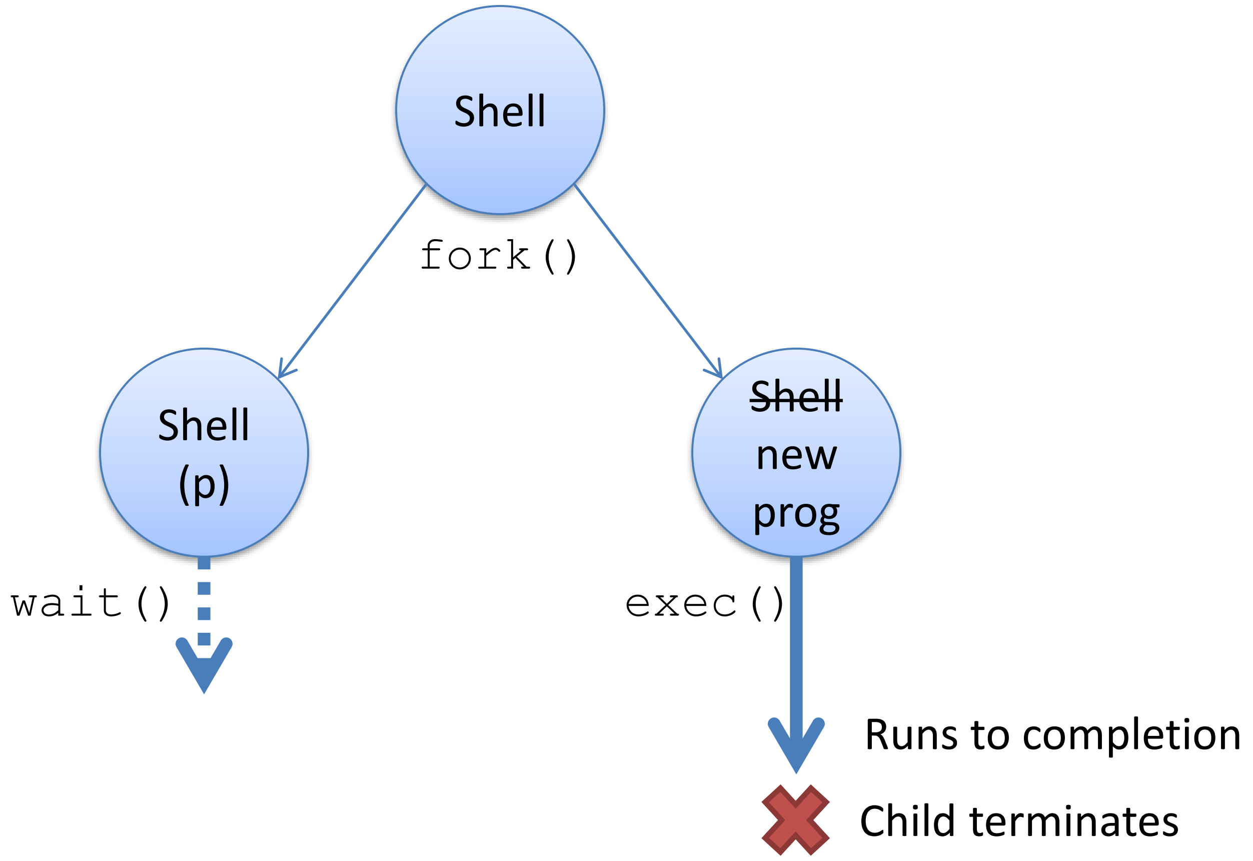
Common `fork()` usage: Shell

2. child: `exec()` user-requested program



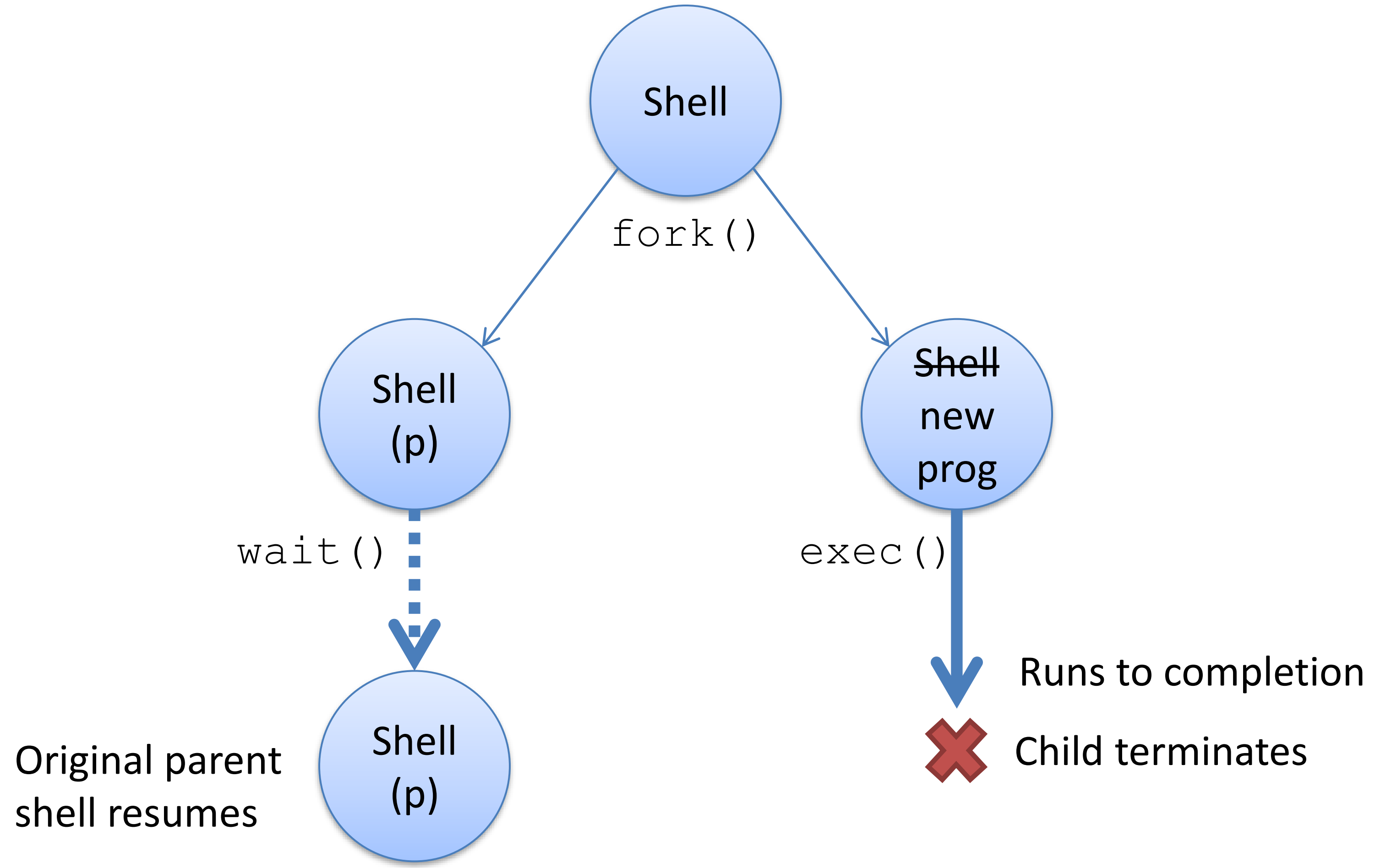
Common `fork()` usage: Shell

3. child program terminates, cycle repeats



Common `fork()` usage: Shell

3. child program terminates, cycle repeats



Process Termination

- When does a process die?
  - It calls `exit(int status);`
  - It `returns` (an int) from main
  - It receives a termination signal (from the OS or another process)

- Key observation: the dying process *produces status information*.

- Who looks at this?
- The parent process!

Reaping Children

(Bet you didn't expect to see THAT title on a slide when you signed up for CS 31?)

- `wait()`: parents reap their dead child processes
  - Given info about why child terminated, exit status, etc.

- Two variants:
  - wait(): wait for and reap next child to exit
  - waitpid(): wait for and reap specific child

- This is how the shell determines whether or not the program you executed succeeded.

Common `fork()` usage: Shell

`fork()` child process.

(child) `exec()` desired program to replace child's address space.

(parent) `wait()` for child process to terminate.

- Check child's result, notify user of errors.

repeat…

What should happen if dead child processes are never reaped? (That is, the parent has not `wait`()ed on them?)

A. The OS should remove them from the process table (process control block / PCB).

B. The OS should leave them in the process table (process control block / PCB).

C. The neglected processes seek revenge as undead in the afterlife.

“Zombie” Processes

- Zombie: A process that has terminated but not been reaped by parent. (AKA defunct process)
- Does not respond to signals (can’t be killed)
- OS keeps their entry in process table:
  - Parent may still reap them, want to know status
  - Don’t want to re-use the process ID yet

Basically, they’re kept around for bookkeeping purposes, but that’s much less exciting...

Signals

- How does a parent process know that a child has exited (and that it needs to call wait)?

- Signals: inter-process notification mechanism
  - Info that a process (or OS) can send to a process.
    - Please terminate yourself (SIGTERM)
    - Stop NOW (SIGKILL)
    - Your child has exited (SIGCHLD)
    - You’ve accessed an invalid memory address (SIGSEGV)
    - Many more (SIGWINCH, SIGUSR1, SIGPIPE, …)

Signal Handlers

- By default, processes react to signals according to the signal type:
  - SIGKILL, SIGSEGV, (others): process terminates
  - SIGCHLD, SIGUSR1: process ignores signal
- You can define "signal handler" functions that execute upon receiving a signal.
  - Drop what program was doing, execute handler, go back to what it was doing.
  - Example: got a SIGCHLD? Enter handler, call `wait()`
  - Example: got a SIGUSR1? Reopen log files.
- Some signals (e.g., SIGKILL) cannot be handled.

Summary

- Processes cycled off and on CPU rapidly
  - Mechanism: context switch
  - Policy: CPU scheduling
- Processes created by `fork()` ing
- Other functions to manage processes:
  - exec(): replace address space with new program
  - exit(): terminate process
  - wait(): reap child process, get status info
- Signals one mechanism to notify a process of something