Processes

11/3/16

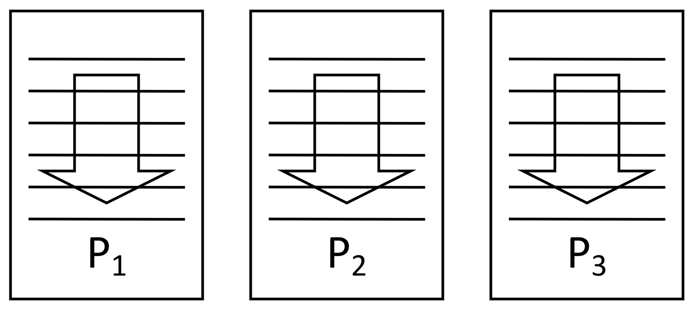
Recall: the kernel's job

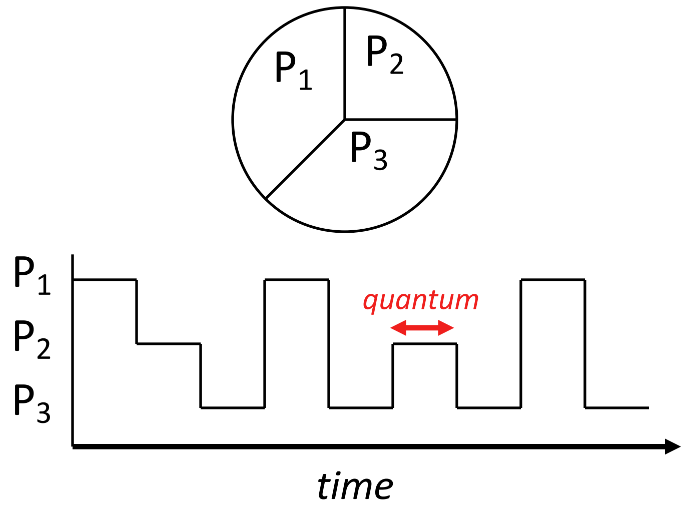
Ensure that all running processes have reasonable access to system resources:

- CPU
- Memory
- IO devices

Provide a **lone view** abstraction: each process should run as if it were the only one on the system.

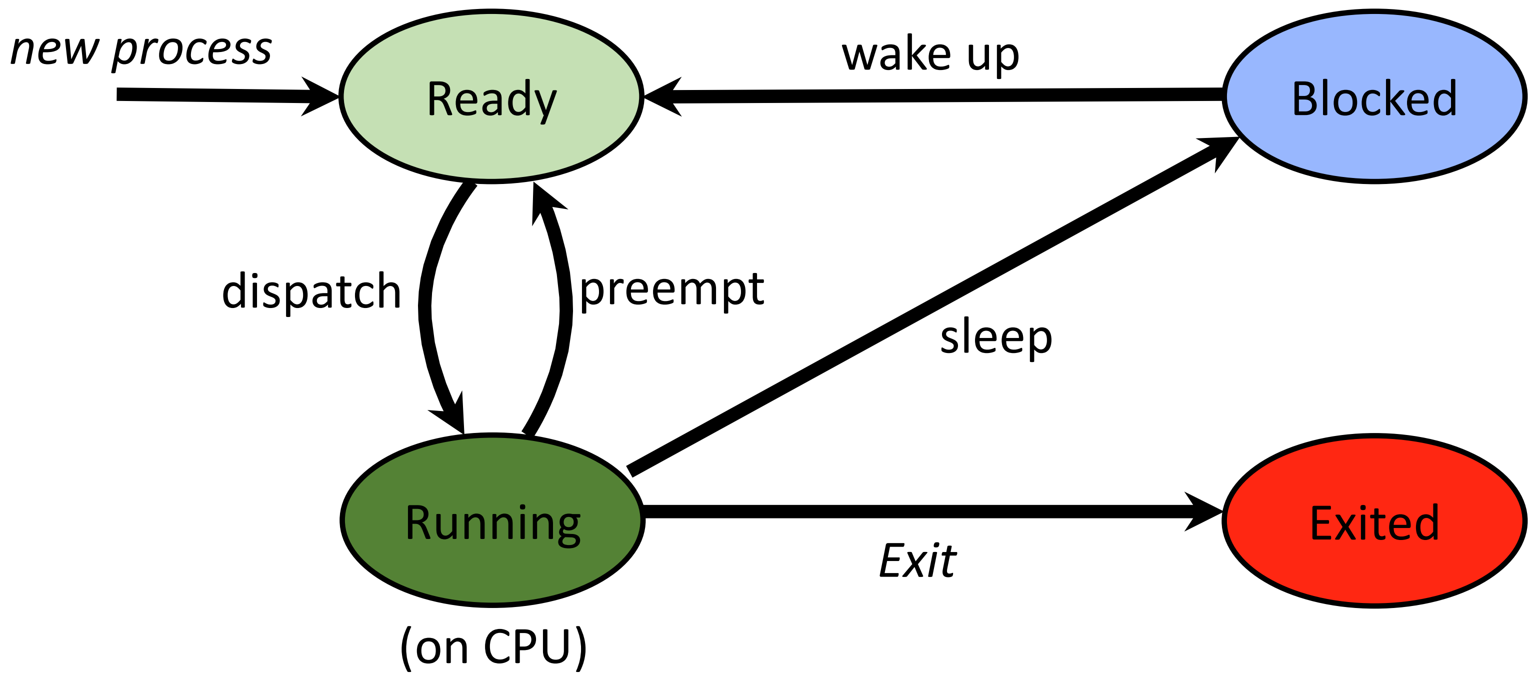
Managing CPU cycles: timesharing





- Multiple processes per CPU
- Each process makes progress over time
- Illusion of parallel progress by rapidly switching CPU
- When scheduled, a process runs for a few milliseconds ($1\text{ms} = 10^{-3}$ seconds) before getting preempted.

Process State



- 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

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).

How is Timesharing Implemented?

- Kernel tracks the state of each process:
  - 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
- A process runs until stopped:
  - Can stop itself by making a system call (TRAP)
  - Can be stopped by the kernel (interrupt)
- Kernel runs to perform a context switch:
  - Save the state of the current process
  - Selects another ready process and load its state

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
  - CPU context (register values)
  - areas of memory being used
  - other information

How a Context Switch Occurs

- Process makes system call (TRAP) or is interrupted
  - 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/trap handler
- In software (in the kernel code)
  - Save context of last-running process
  - Select new process from those that are ready
  - Restore context of selected process
  - OS returns control to a process from interrupt/trap

Why should the kernel (not each process) control context switching?

A. It would cause too much overhead.

B. They could refuse to give up the CPU.

C. They don’t have enough information about other processes.

D. Some other reason(s).

Policy question: how should the kernel pick the next process to run?

There are lots of reasonable options:

- Keep running the same process when possible.
  - maximize throughput
- Run the process that has been ready longest.
  - ensure fairness
- Prioritize some processes.
  - Should user(s) set the priorities?
  - Should priority be determined automatically?

Linux’s Policy
(You’re not responsible for this.)

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

- 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.

Where do processes come from?

The `fork()` system call creates a new process.

On boot, the kernel spawns the “init” process.

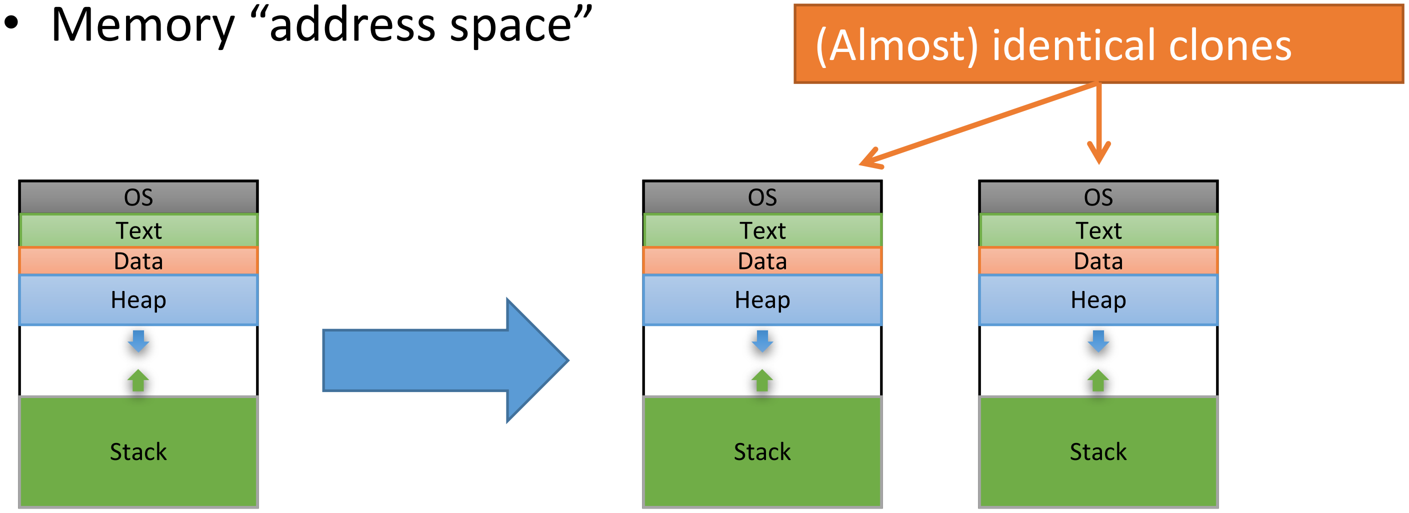
Init calls `fork()` to create child processes.

Those processes can also create children with `fork`.

`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”);
```

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

After `fork()`, call `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

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

Common `fork()` usage: Shell

1. `fork()` child process.

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

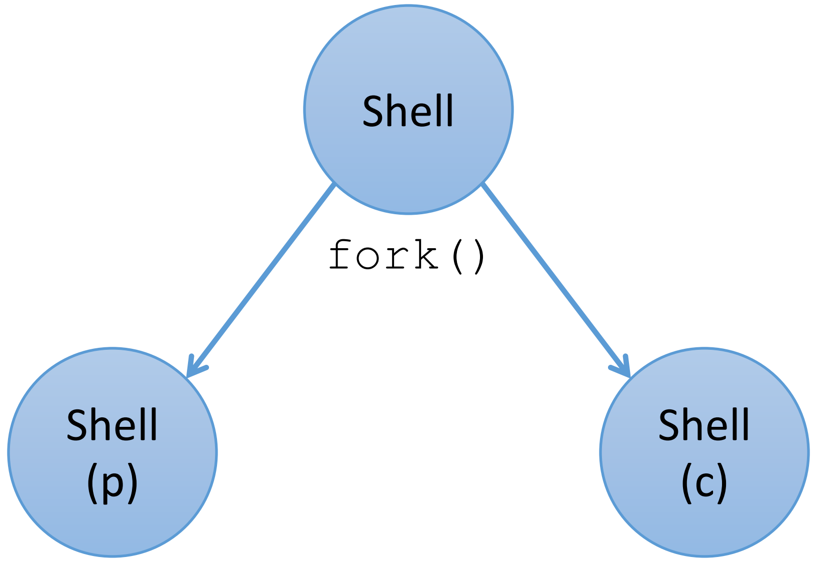
2. `wait()` for child process to terminate.

3. repeat…

The parent and child each do something different next.

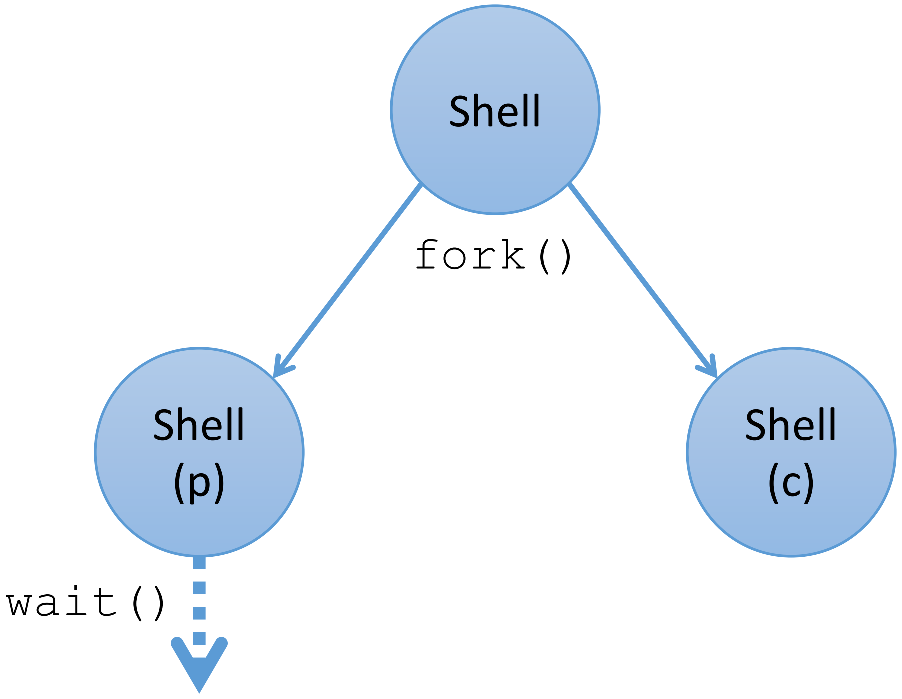
Common `fork()` usage: Shell

1. `fork()` child process.



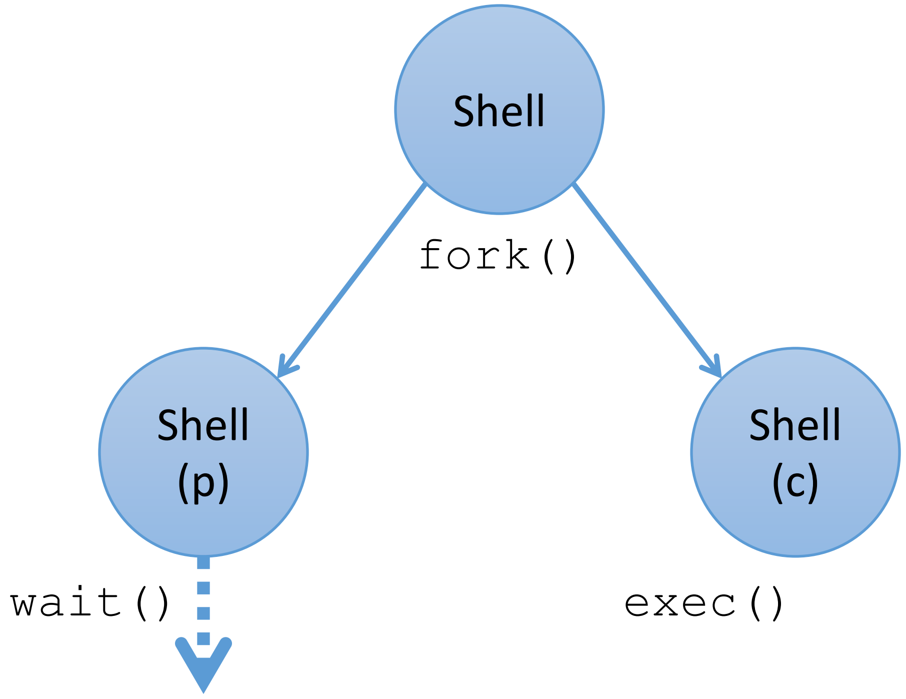
Common `fork()` usage: Shell

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



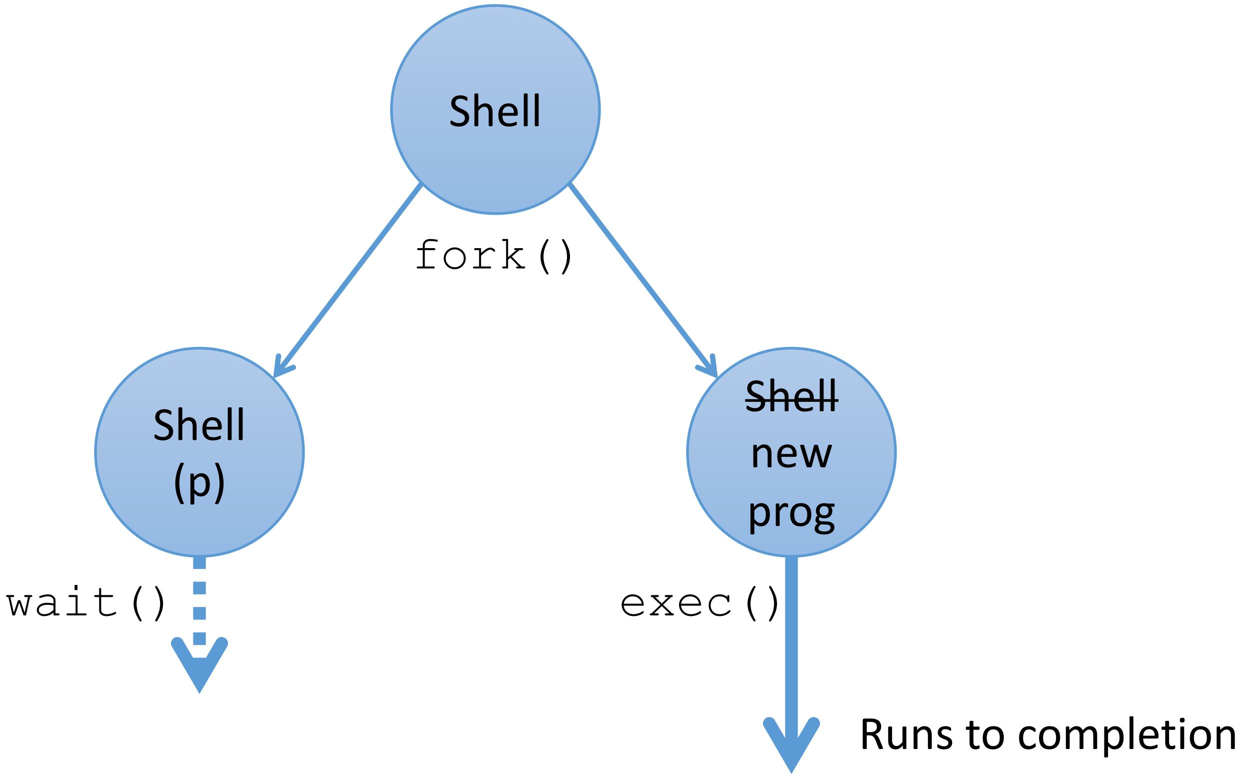
Common `fork()` usage: Shell

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



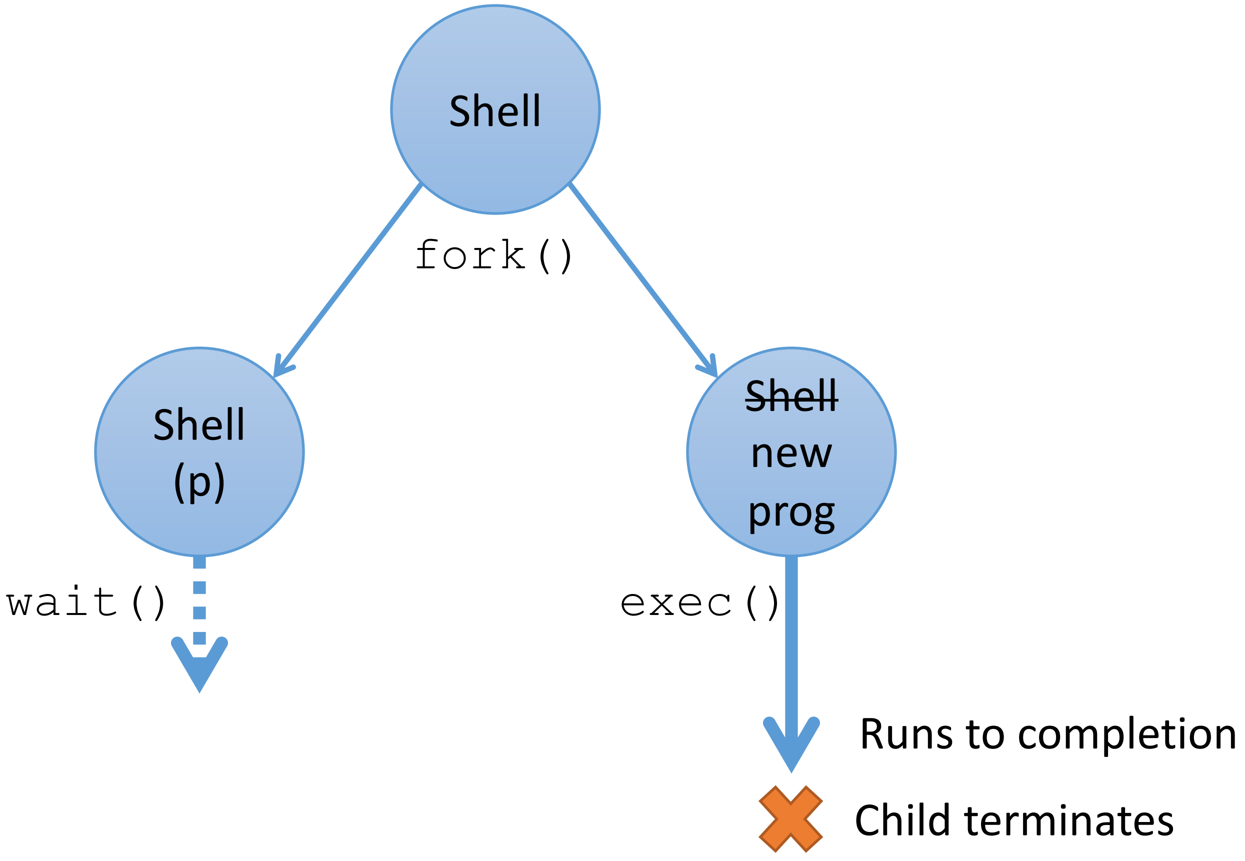
Common `fork()` usage: Shell

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



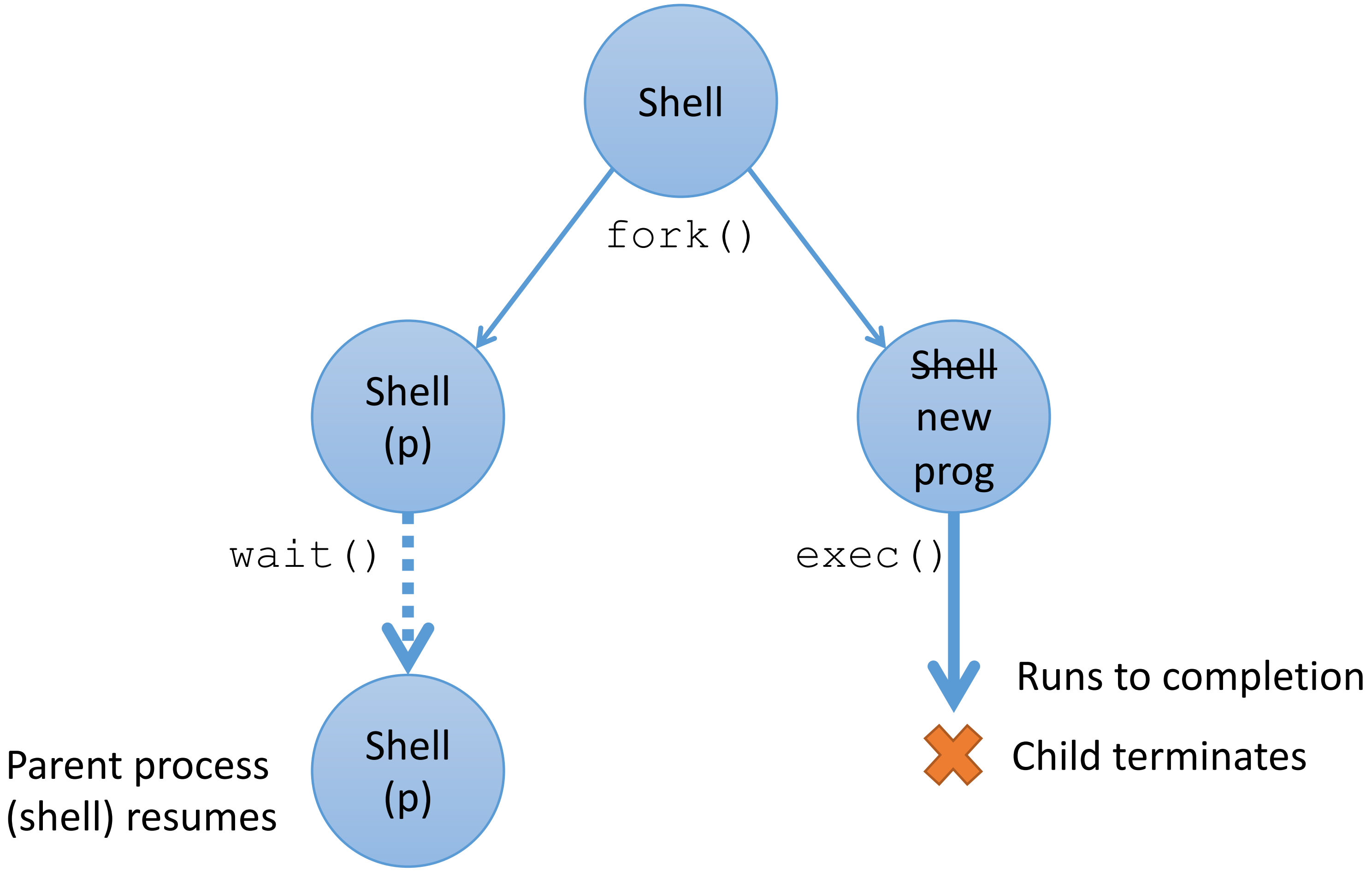
Common `fork()` usage: Shell

3. child program terminates, cycle repeats



Common `fork()` usage: Shell

3. child program terminates, cycle repeats



Process Termination

A process terminates when:

- `main` returns
- It calls `exit()`
- It receives a termination signal (from the OS or another process)

- In all cases, it process *produces status information*.
- The parent process receives this information.

What Happens when a process exits?

It becomes a zombie process until its parent reaps it.

Zombie process:

- exited, mostly dead, not runnable anymore
- waiting for parent to completely remove all of its state from the system

Reaping zombies

The parent process is responsible for cleaning up child process state. Two options:

1. Parent explicitly waits for child to exit by calling a wait function:
   - `wait`: wait for any child to exit (pid returned).
   - `waitpid`: wait for a specific child to exit.

2. Parent receives a SIGCHILD signal, and the signal handler code calls `wait` to reap the child.

Summary: system calls for processes

- `fork`: spawns new process.
  - Called once, Returns twice (in parent and child process).
- `exit`: terminates own process.
  - Called once, never returns.
  - Puts it into “zombie” status.
- `wait` or `waitpid`: reap terminated children.
- `exec` family: runs new program in existing process.
  - Called once, (normally) never returns.

Signals

Signal: a software interrupt: a small message to tell a process that some event has happened.

- OS <u>sends</u> a signal to a process
  - On behalf of another process that called the `kill` syscall
  - As the result of some event (NULL pointer dereference)

- A process <u>receives</u> a signal

  Asynchronous: signalee doesn't know when it will get one

  A signal is <u>pending</u> before a process receives it

- A signal <u>interrupts</u> the receiving process, which then runs <u>signal handler</u> code
  - default handlers for each signal type in OS
  - programmer can also add signal handler code

Signals

OS identifies specific signal by its number, examples:

ID	*Name*	*Default Action*	*Corresponding Event*
2	SIGINT	Terminate	Interrupt (e.g., ctl-c from keyboard)
9	SIGKILL	Terminate	Kill program (cannot override or ignore)
11	SIGSEGV	Terminate	Invalid memory reference (e.g. NULL ptr)
14	SIGALRM	Terminate	Timer signal
17	SIGCHLD	Ignore	Child stopped or terminated

Sending Signals:

Unix command:

`$ kill -9 1234` # send `SIGKILL` signal to process 1234

System call:

`kill(1234, SIGKILL);` // send `SIGKILL` to process 1234

Implicitly sent: side-effect of program doing something
(NULL ptr dereference causes `SIGSEGV`)

Receiving a Signal

- A destination process *receives* a signal when it is forced by the kernel to react in some way to the delivery of the signal.

- Three possible ways to react:
  - *Ignore* the signal (do nothing)
    not all signals can be ignored (e.g. SIGKILL)
  - *Terminate* the process on receipt of signal
  - *Catch* the signal by executing a user-level function called *signal handler*

Installing Signal Handlers

```
signal(int signum, handler_t *handler);
```

- Modifies the default action associated with the receipt of a particular signal.

- `handler` is a *signal handler* function
    - When program receives signal, it jumps to start executing the `handler` function.
    - When the `handler` done executing, control passes back to instruction in the control flow of the process that was interrupted by receipt of the signal.

Summary

- Processes are cycled off and on CPU rapidly.
  - Mechanism: context switch
  - Policy: CPU scheduling

- Processes are created by `fork()`.

- Other functions to manage processes:
  - `exec()`: replace address space with new program
  - `exit()`: terminate process
  - `wait()`: reap child process, get status info

- Signals communicate events to processes.