CS 31: Introduction to Computer Systems

20 OS Processes

04-10-2025

OS Big Picture Goals

- OS is a layer of code between user programs and hardware.
- Goal: Make life easier for users and programmers.
- How can the OS do that?

Key OS Responsibilities

1. Simplifying abstractions for programs
2. Resource allocation and/or sharing
3. Hardware gatekeeping and protection

Running multiple programs

More than 200 processes running on a typical desktop!

- Benefits: when I/O issued, CPU not needed
  - Allow another program to run
  - Requires yielding and sharing memory

- Challenges: what if one running program…
  - Monopolizes CPU, memory?
  - Reads/writes another's memory?
  - Uses I/O device being used by another?

OS: Turn undesirable into desirable

- Turn undesirable inconveniences: reality
  - Complexity of hardware
  - Single processor
  - Limited memory
- Into desirable conveniences: illusions
  - Simple, easy-to-use resources
  - Multiple/unlimited number of processors
  - Large/unlimited amount of memory

Virtualization

- Rather than exposing real hardware, introduce a “virtual”, abstract notion of the resource

- Multiple virtual processors
  - By rapidly switching CPU use
- Multiple virtual memories
  - By memory partitioning and re-addressing
- Virtualized devices
  - By simplifying interfaces, and using other resources to enhance function

We'll focus on the OS 'kernel'

- "Operating system" has many interpretations
  - E.g., all software on machine minus applications (user or even limited to 3rd party)

- Our focus is the *kernel*
  - What's necessary for everything else to work
  - Low-level resource control
  - Originally called the nucleus in the 60's

The Kernel

- All programs depend on it
  - Loads and runs them
  - Exports system calls to programs
- Works closely with hardware
  - Accesses devices
  - Responds to interrupts (hardware events)
- Allocates basic resources
  - CPU time, memory space
  - Controls I/O devices: display, keyboard, disk, network

Tron, 1982

Kernel provides common functions

- Some functions useful to many programs
  - I/O device control
  - Memory allocation
- Place these functions in central place (kernel)
  - Called by programs ("system calls")
  - Or accessed in response to hardware events
- What should functions be?
  - How many programs should benefit?
  - Might kernel get too big?

OS Kernel

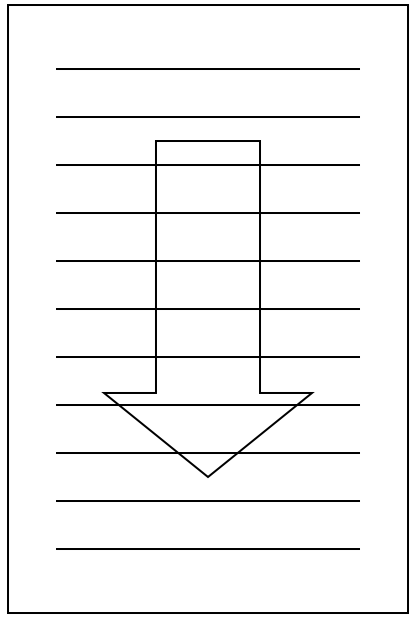
- Big Design Issue: How do we make the OS efficient, reliable, and extensible?

- General OS Philosophy: The design and implementation of an OS involves a constant tradeoff between simplicity and performance.

- As a general rule, strive for simplicity.
  - except when you have a strong reason to believe that you need to make a particular component complicated to achieve acceptable performance
  - (strong reason = simulation or evaluation study)

Main Abstraction: The Process

- Abstraction of a running program
  - "a program in execution"
- Dynamic
  - Has state, changes over time
  - Whereas a program is static
- Basic operations
  - Start/end
  - Suspend/resume



Basic Resources for Processes

- To run, process needs some basic resources:
  - CPU
    - Processing cycles (time)
    - To execute instructions
  - Memory
    - Bytes or words (space)
    - To maintain state
  - Other resources (e.g., I/O)
    - Network, disk, terminal, printer, etc.

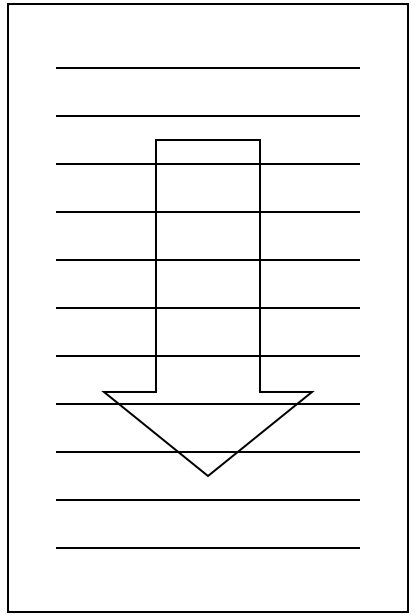
Machine State of a Process

- CPU or processor context
  - PC (program counter)
  - SP (stack pointer)
  - General purpose registers
- Memory
  - Code
  - Global Variables
  - Stack of activation records / frames
  - Other (registers, memory, kernel-related state)

Must keep track of these for every running process !

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)



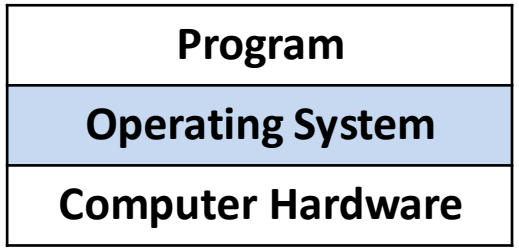
Managing Processes

- 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.

Recap



- OS sits between HW and User/Program
  - Manages HW & Makes system easier to use
  - Interrupt Driven: HW or User/Program interrupt it to do something on their behalf
    - HW interrupt, Program system call traps to OS
- OS implements the Process Abstraction
  - Process: a running program
    - Lone view & private virtual address space
  - Why: efficient use of system resources
    - Multiprogramming and Timesharing. (`ps -A`)
  - `fork()`: system call to create a new process

Managing Processes

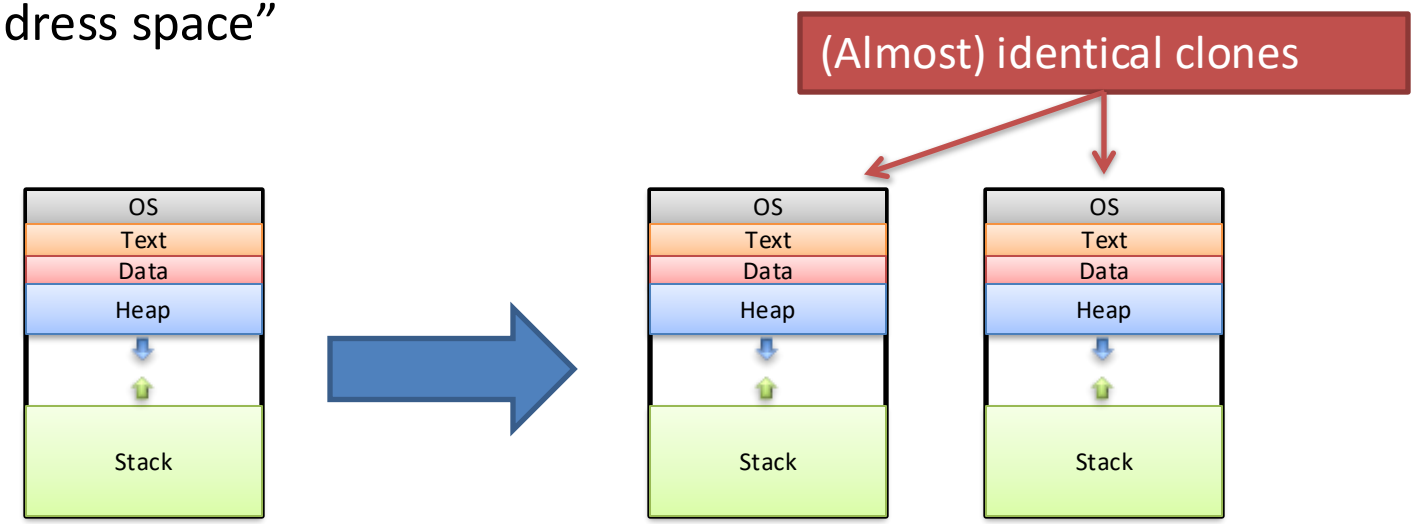
- Given a process, how do we make it execute the program we want?

- 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.

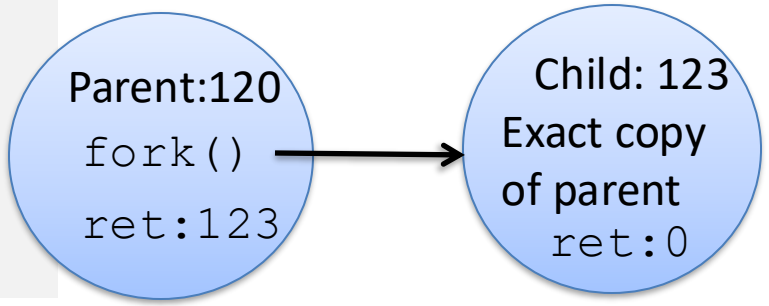
```
pid_t pid = fork(); // both continue after call
if (pid == 0) {
   printf("hello from child\n");
} else {
   printf("hello from parent\n");
}
```

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

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

What Happens after a fork?

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



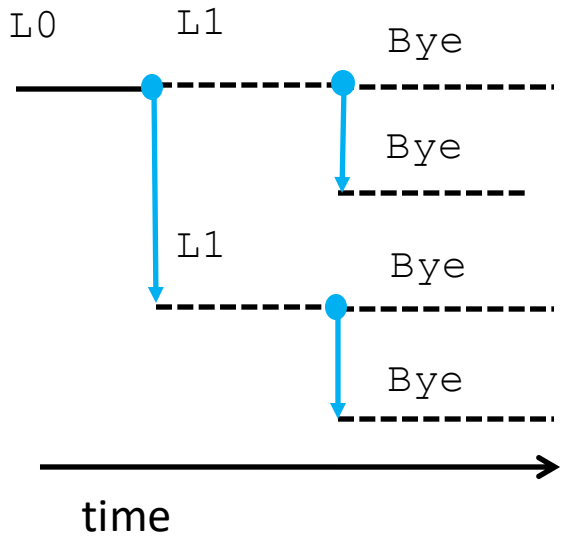
Parent & Child become concurrent processes

- Both assign return value to their copy of `ret` variable
- Both execute `if-else`
  - Child: if-cond is true
  - Parent: if-cond is false
- Who executes the `printf` statement first?
  - Depends on which gets scheduled on CPU first
  - Can vary every execution: no ordering of concurrent actions

fork example

- Both Parent and Child process can continue forking

```
void forky()
{
   printf("L0 ");
   fflush(stdout);  //  forces printf to output
                    //  (printing \n does too)

   fork();          //  parent & child cont .
   printf("L1 ");   //  both print
   fflush(stdout);

   fork();          //  both fork new child
   printf("Bye ");  //  all 4 processes print
   fflush(stdout);
}
```



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

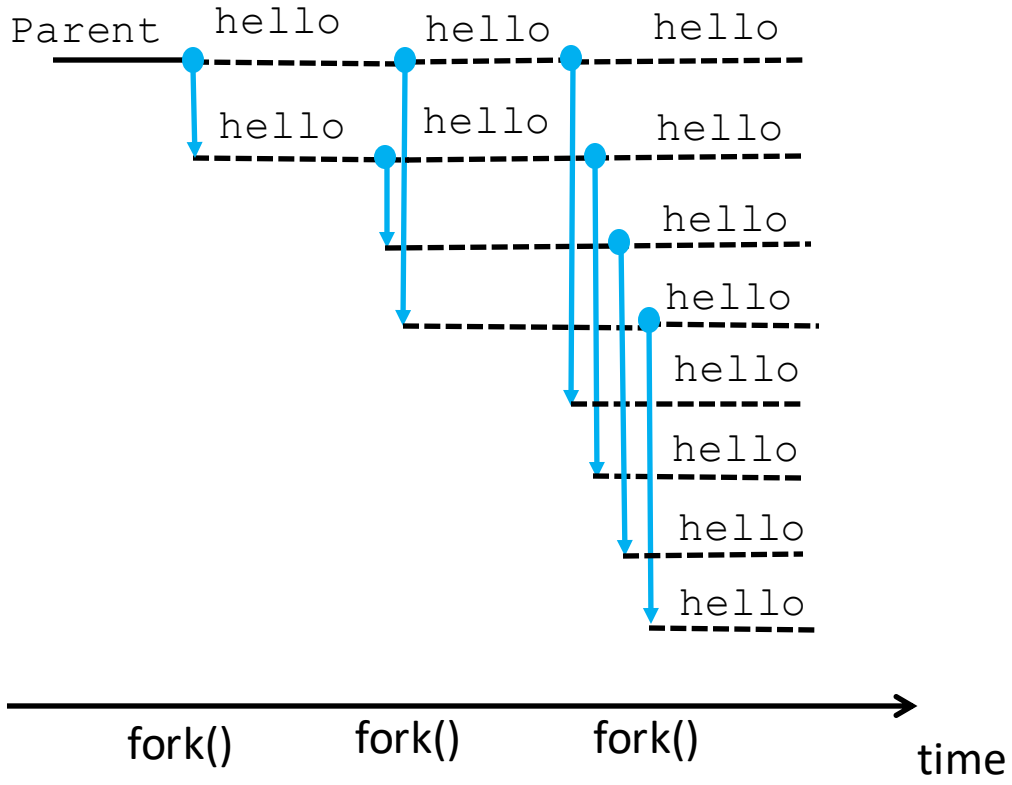
C.12

D.16

E.18

How many hello's will be printed?

```
fork();
printf("hello");
if (fork()) {
  printf("hello");
}
fork();
printf("hello");
```



8 processes and 12 "hello"s printed

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.

exec system calls

(ex) `int execvp(char *filename,char *argv[]);`

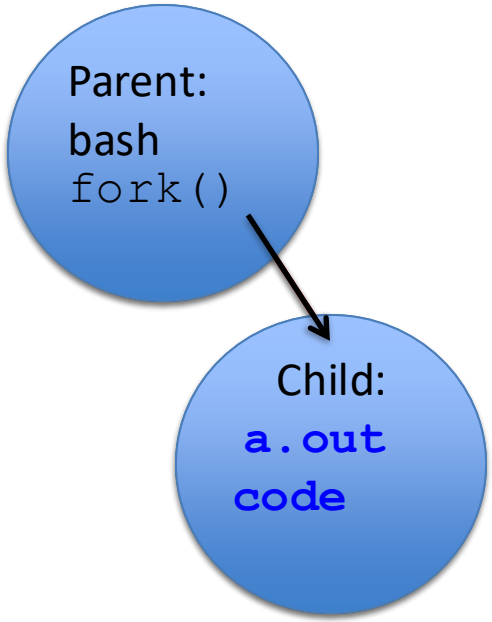
(there are different versions of exec, diff names and diff args)

1. Overlays the executable code from `filename` on the calling process's address space
2. Initializes other parts of memory space: stack, heap, data, ...
3. Sets up process to execute the first instruction in the `filename` binary (changes child's `%rip` value)
4. Passes in `argv` as command line arguments

`exec` system call only returns if it fails with an error.

Child Process exec's

```
pid = fork();
if (pid == 0) { /* child */
  if (execvp("a.out",argv)<0) {
     printf("%s: Command not found.\n", argv[0]);
     exit(0);
  }
}
```



execvp: child runs a.out code from its start rather than its copy of parent's code from after fork (which has been completely overwritten with a.out by execvp)

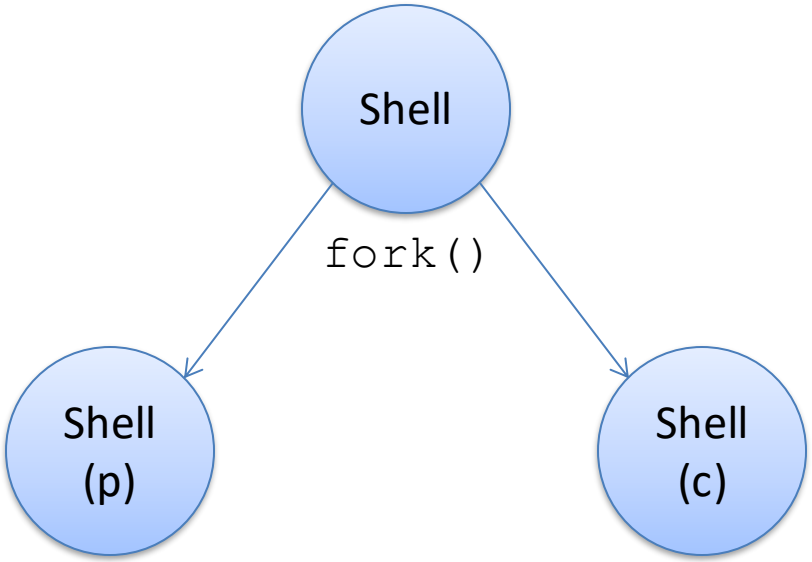
Common `fork()` usage: Shell

1. `fork()` child process.
2. `exec()/execvp()` 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.

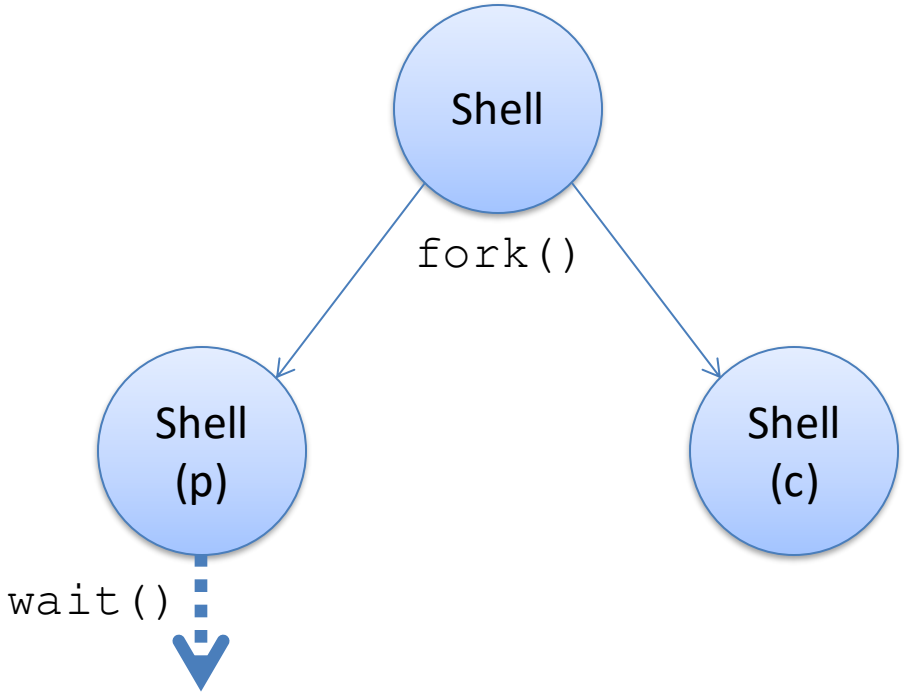
Common `fork()` usage: Shell

1. `fork()` child process.



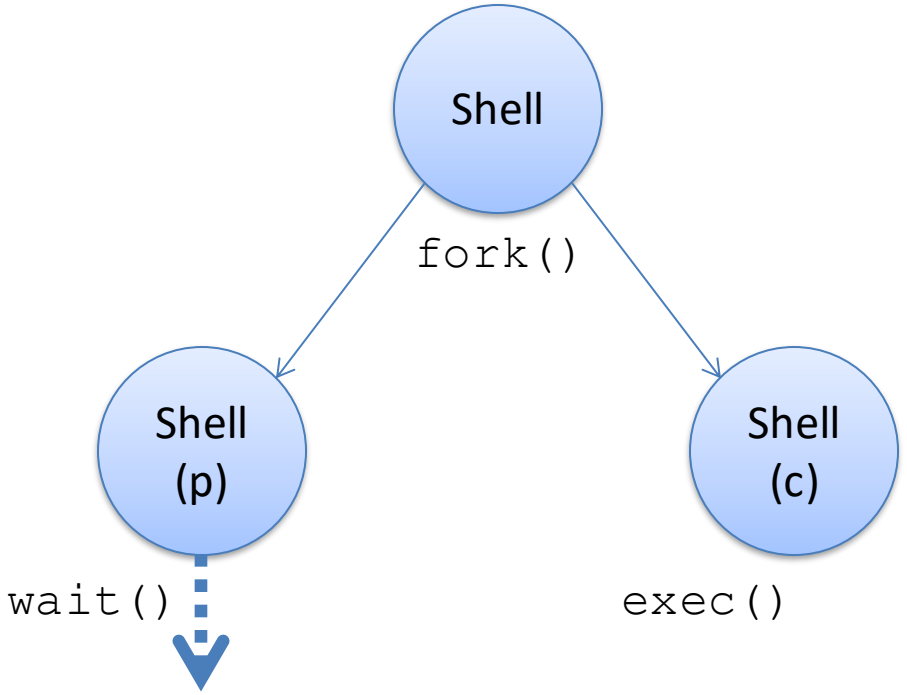
Common `fork()` usage: Shell

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



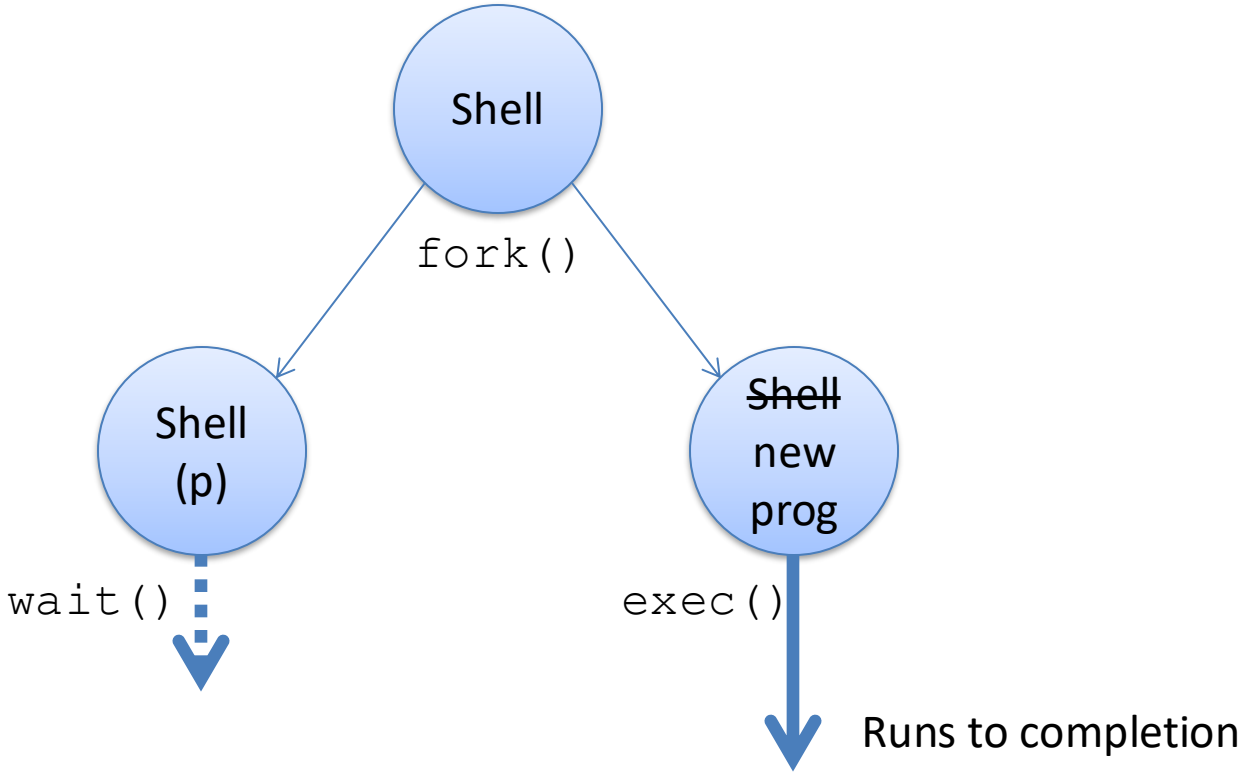
Common `fork()` usage: Shell

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



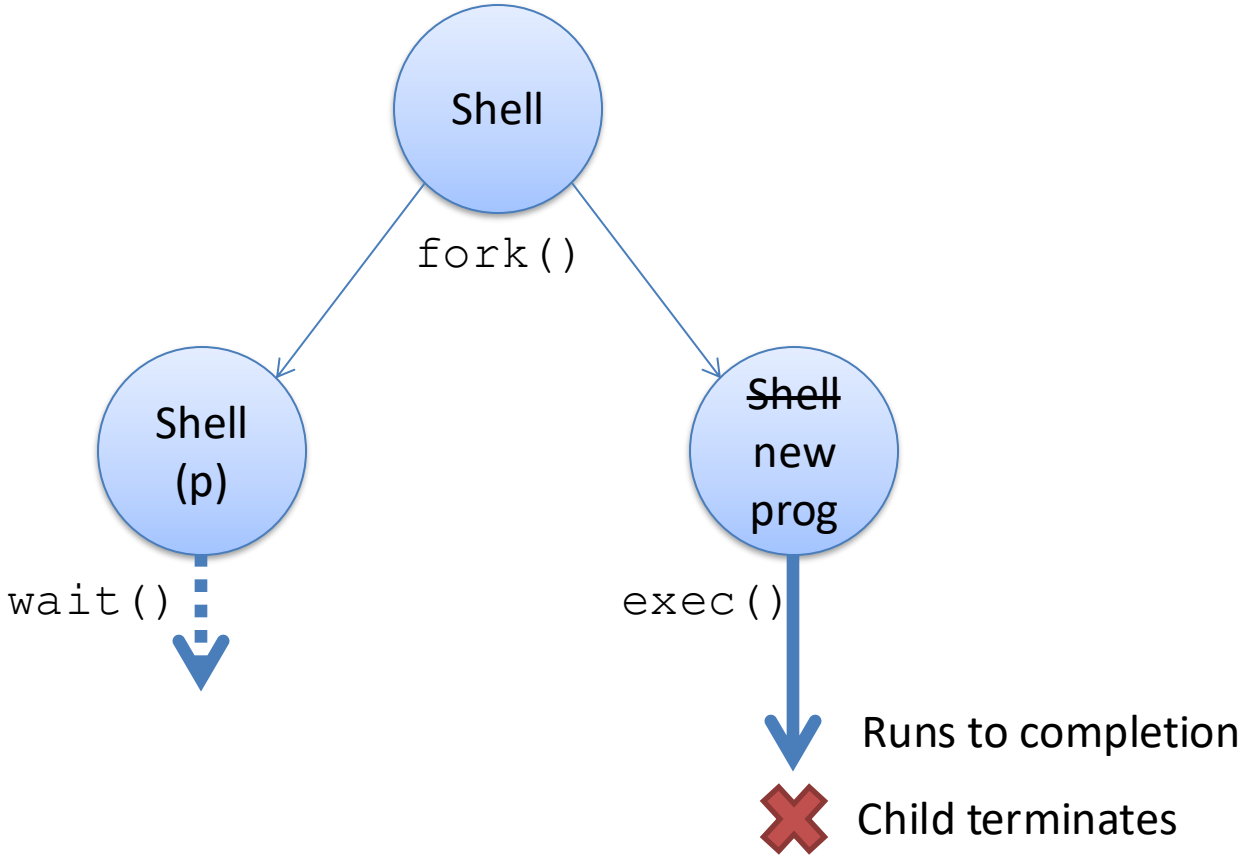
Common `fork()` usage: Shell

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



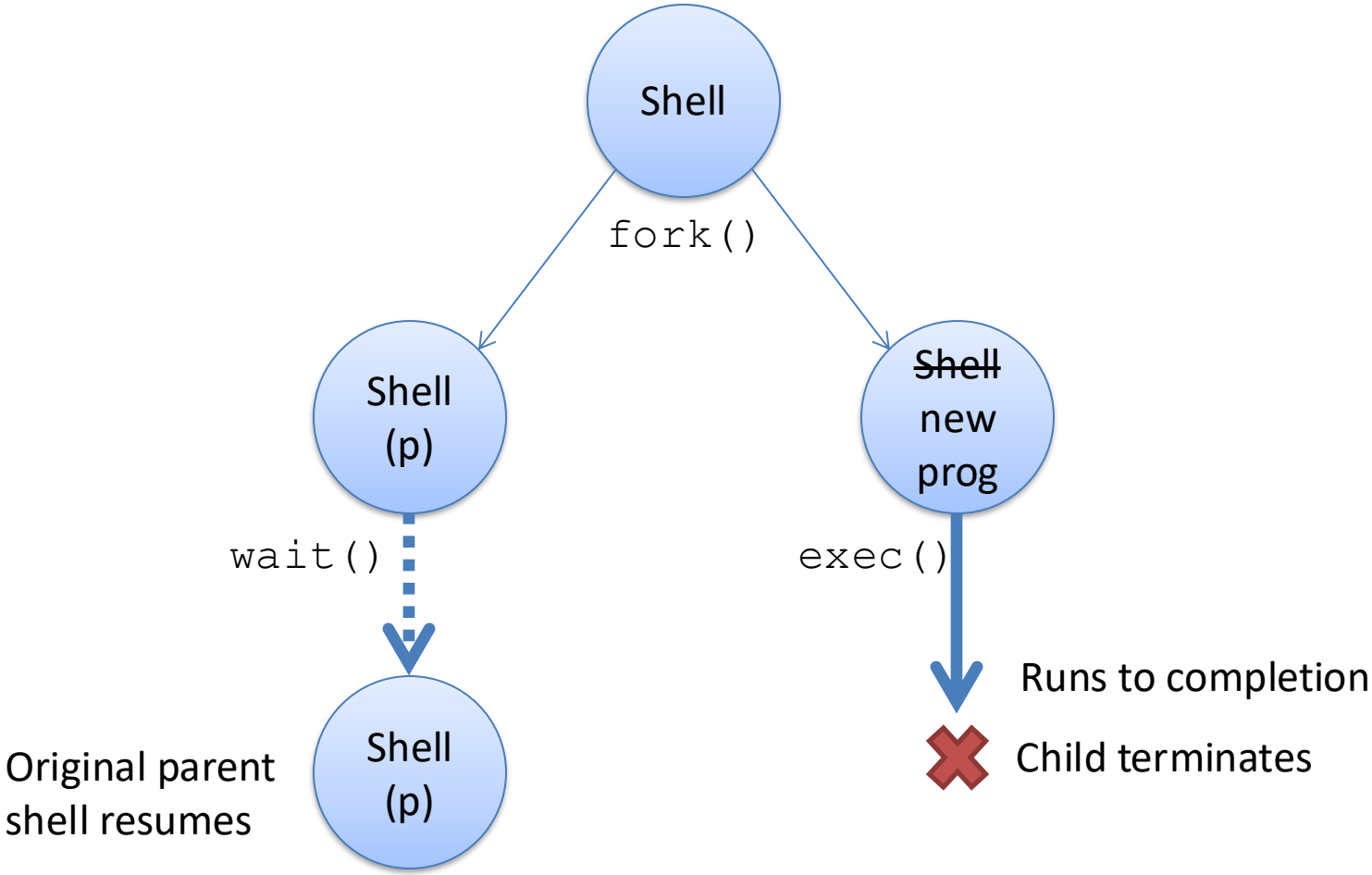
Common `fork()` usage: Shell

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

B. The OS should leave them in the process table

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.

Terminating a Process

```
void exit(int status);
```

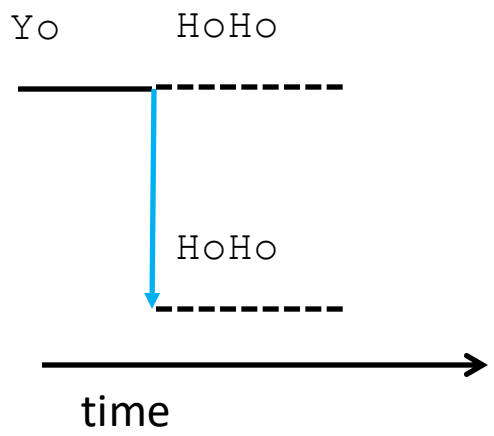
- A process calls `exit` to terminate:

```
exit(0);   // 0 means: exit without an error
exit(1);  // non-zero means: an error exit
```

To see program's exit value:

```
./a.out
```

```
fork_pirates() {
    printf("Yo");
    fflush(stdout);
    fork();
    printf("HoHo");
    fflush(stdout);
    exit(0);
}
```

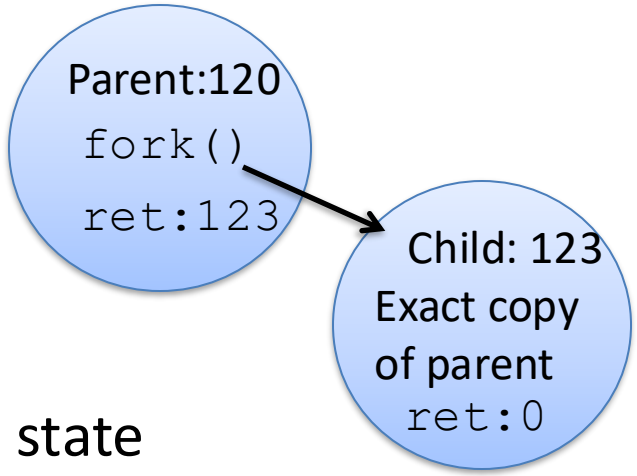


How does the call to exit happen?

1. Explicit to the C programmer:
   - include a call to `exit` in the program code
2. "Implicit" hidden from C programmer:
   - return from main

     code runs after main returns and it calls `exit`
3. In signal handler code:
   - Another process can send this process a kill signal telling it to call `exit` to terminate (CNTL-C)
   - This process does something irreversibly bad (SEGFAULT), triggers code that calls `exit`

Process
create & exit



`fork()` create a new child process

- gets exact copy of its parent execution state
- fork returns 2x: one in parent's execution context, one in child's
- returns 0 to child, child's pid to parent
- parent and child are **concurrent** processes after fork

`exit()` terminate the calling process

- process cleans up most of own process state from system by running `exit` system call
- then enters the EXITED state (can no longer run on CPU)

What Happens when a process exits?

It becomes a zombie process until its parent reaps it

Zombie process:

- exited, mostly dead, not runnable anymore

  (unlike real zombies they won't try to eat other processes)

- waiting for parent to completely remove all of its state from the system

FYI: How to see a zombie

```
void zombie(){
  if (fork() == 0) {// child
    printf("Child, PID = %d\n", getpid());
    exit(0);
  }
  else {//parent
    printf("Parent, PID = %d\n", getpid());
    while(1) {
       //Infinite loop
    }
  }
}
```

```
$ ./a.out &
Parent, PID = 6639
Child, PID = 6640

$ ps -S
  PID TTY          TIME CMD
 6585 ttyp9    00:00:00 bash
 6639 ttyp9    00:00:03 ./a.out
 6640 ttyp9    00:00:00 ./a.out <defunct>
 6641 ttyp9    00:00:00 ps

$ kill -9 6639
$ ps
```

- stop parent from exiting before we can see it (ex. infinite loop)
- `ps -S` lists processes started by shell

  `<defunct>`: zombie child process
- `kill -9 6639` kills parent process, which will reap its zombie children

How does parent reap zombie child?

1. Parent waits for child to exit by calling a `wait` system call:
    1. Blocks the parent until the child exits
    2. reaps the exited child and returns

2. Parent receives a SIGCHILD signal when child exits, and its signal handler code calls `wait` to reap the child:
    1. Reaps the exited child and returns

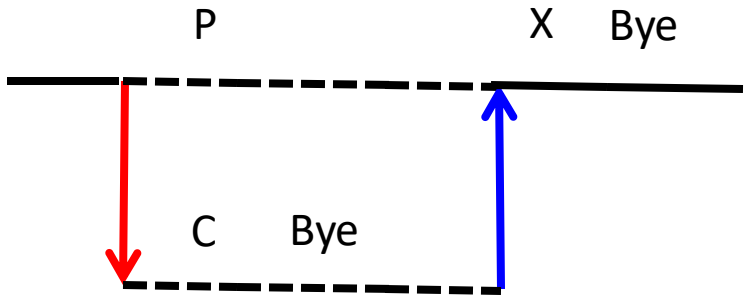
`wait` system calls

Remove all remaining parts of exited child process from the system

```
//  blocks caller (parent)  until child process exits,
//  returns pid of child process that terminated
pid_t wait(int *child_status);

// more configurable: wait for specific child, or any, or just
// check and see if a child exited (don't block or reap if not),
pid_t waitpid(pid_t pid, int *status, int options);
```

Wait Example

```
void fork_and_wait() {

   int child_status;
   pid_t pid;

   if (fork() == 0) {
      printf("C\n");
   }
   else {
      printf("P\n");
      pid = wait(&child_status);
      printf("X\n");
   }
   printf("Bye\n");
   exit(0);
}
```



Q: What are all possible orderings of program output?

Summary: Running a Program

Basic system calls:

- `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
- `execvp`: runs new program in existing process
  - Called once, (normally) never returns

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