Fork, Wait, Exec

- ➢ System call error handling
- More on fork System Call and Process Graphs
- wait and waitpid system calls
- exec family system calls
- signals

System Call Error Handling

- **On error, Linux system-level functions typically return -1 and set global variable `errno` to indicate cause.**
- **Hard and fast rule:**
  - You must check the return status of every system-level function
  - Only exception is the handful of functions that return `void`
- **Example:**

```
if ((pid = fork()) < 0) {
    fprintf(stderr, "fork error: %s\n", strerror(errno));
    exit(0);
}
```

Error-reporting functions

- Can simplify somewhat using an *error-reporting function*:

```
void unix_error(char *msg) /* Unix-style error */
{
    fprintf(stderr, "%s: %s\n", msg, strerror(errno));
    exit(0);
}
```

```
if ((pid = fork()) < 0)
    unix_error("fork error");
```

Error-handling Wrappers

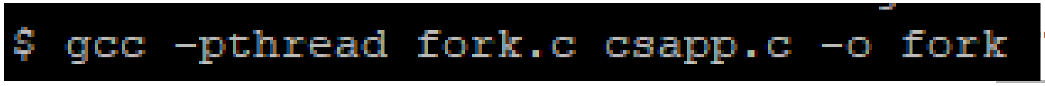
- Textbook simplifies the code further by using Stevens-style error-handling wrappers:

```
pid_t Fork(void)
{
    pid_t pid;

    if ((pid = fork()) < 0)
        unix_error("Fork error");
    return pid;
}
```

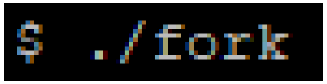
```
pid = Fork();
```

Note on sample programs

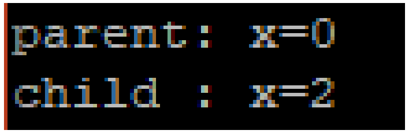
- Most code examples I use in this chapter are from text
- You can find them at : http://csapp.cs.cmu.edu/3e/code.html
    - Under efc
- There are some custom libraries book is using to get them work you need to include csapp.h Link with csapp.c and use –pthread option.



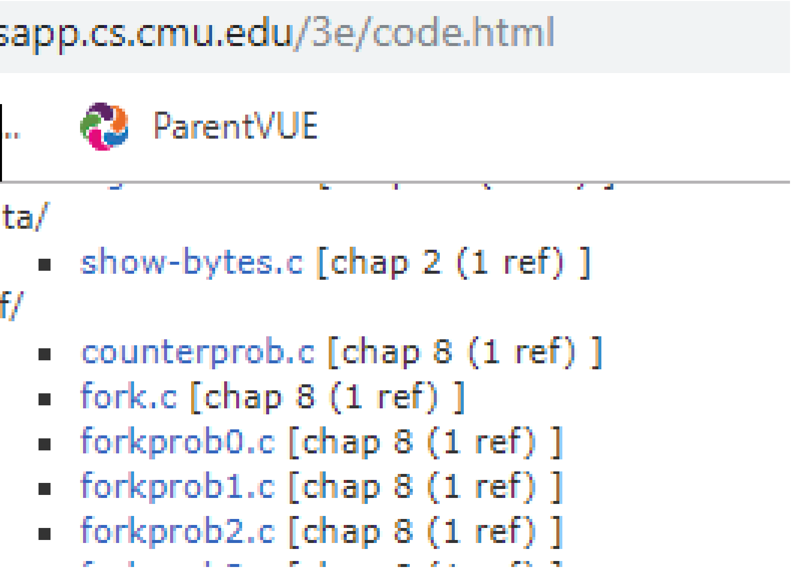
```
$ gcc -pthread fork.c csapp.c -o fork
```



```
$ ./fork
```



```
parent: x=0
child : x=2
```



data/

- show-bytes.c [chap 2 (1 ref)]

ecf/

- counterprob.c [chap 8 (1 ref)]
- fork.c [chap 8 (1 ref)]
- forkprob0.c [chap 8 (1 ref)]
- forkprob1.c [chap 8 (1 ref)]
- forkprob2.c [chap 8 (1 ref)]

Today

- System call error handling
- ➢ More on fork System Call and Process Graphs
- wait and waitpid system calls
- exec family system calls
- signals

`fork` facts

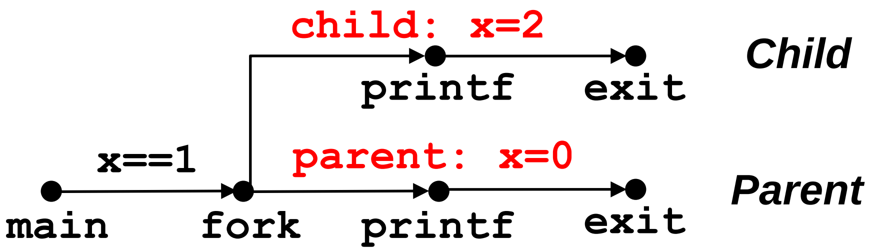
- **Concurrent execution**
  - Can’t predict execution order of parent and child
- **Duplicate but separate address space**
  - `x` has a value of 1 when fork returns in parent and child
  - Subsequent changes to `x` are independent
- **Shared open files**
  - `stdout` is the same in both parent and child

Modeling `fork` with Process Graphs

- **A *process graph* is a useful tool for capturing the partial ordering of statements in a concurrent program:**
  - Each vertex is the execution of a statement
  - a -> b means `a` happens before b
  - Edges can be labeled with current value of variables
  - `printf` vertices can be labeled with output
  - Each graph begins with a vertex with no inedges
- **Any *topological sort* of the graph corresponds to a feasible total ordering.**
  - Total ordering of vertices where all edges point from left to right

Process Graph Example

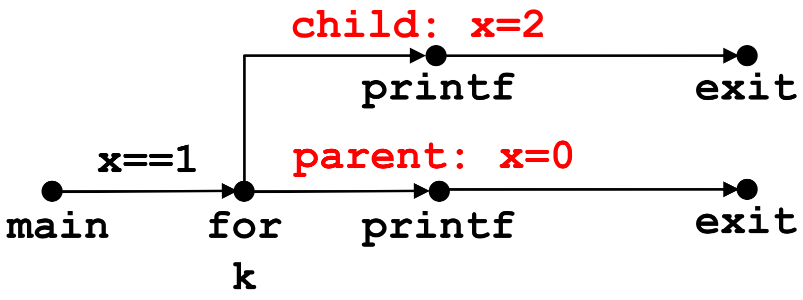
```
int main()
{
    pid_t pid;
    int x = 1;

    pid = Fork();
    if (pid == 0) {  /* Child */
        printf("child : x=%d\n", ++x);
        exit(0);
    }

    /* Parent */
    printf("parent: x=%d\n", --x);
    exit(0);
}
```



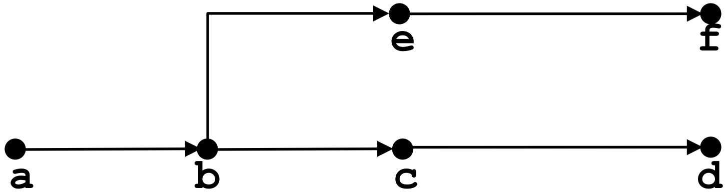
fork.c

Interpreting Process Graphs

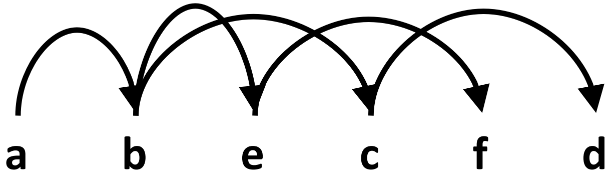
- **Original graph:**



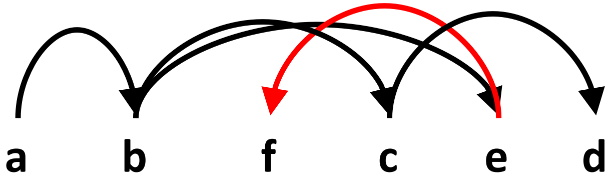
- **Re-labled graph:**



Feasible total ordering:

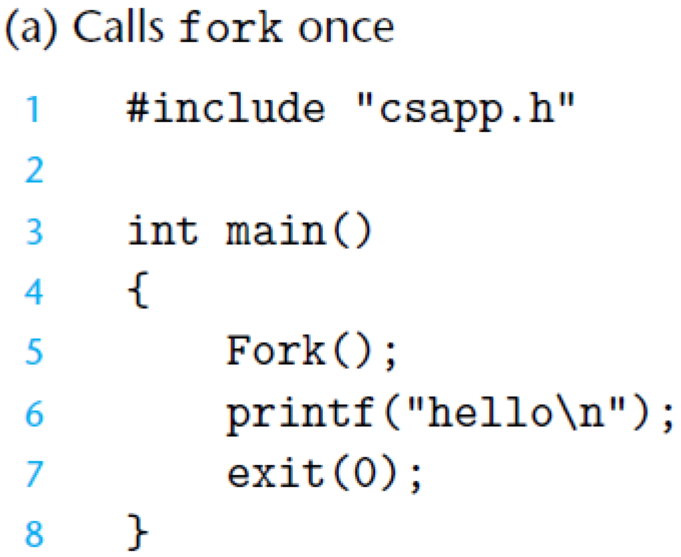


Infeasible total ordering:



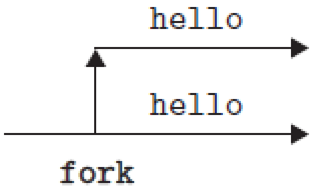
Practice

- **How many lines of output would the program generate**



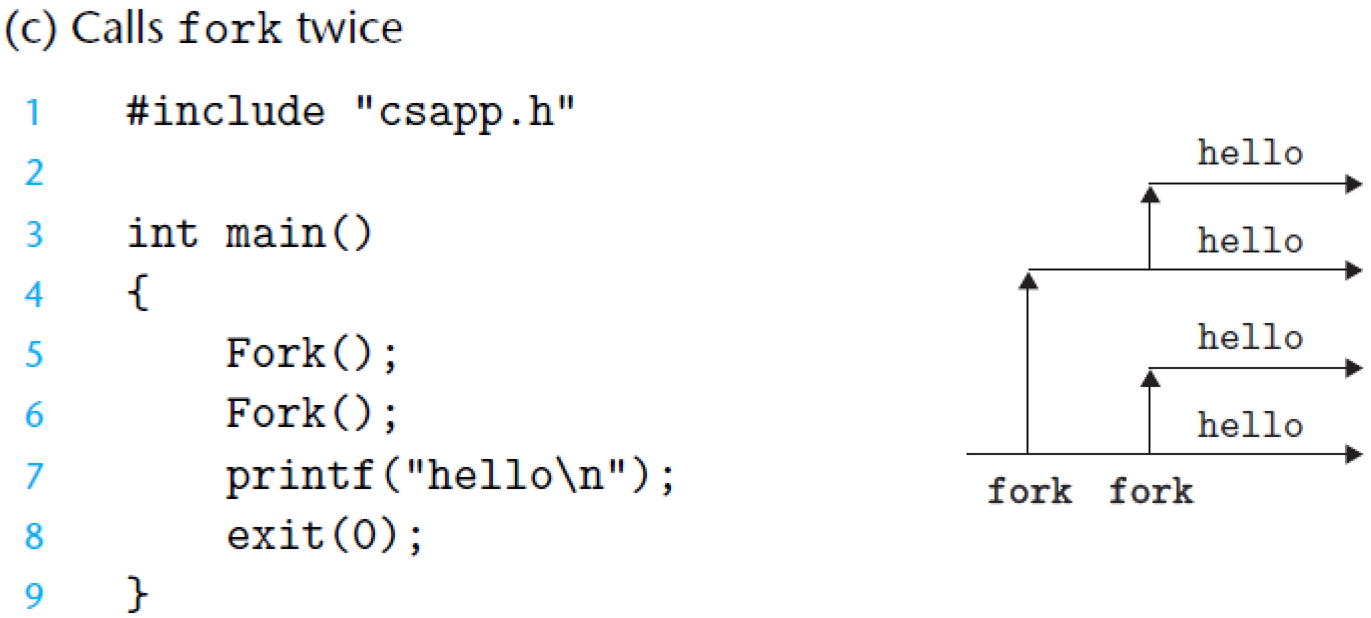
(a) Calls fork once

```
#include "csapp.h"

int main()
{
    Fork();
    printf("hello\n");
    exit(0);
}
```



Practice

- **How many lines of output would the program generate**



(c) Calls fork twice

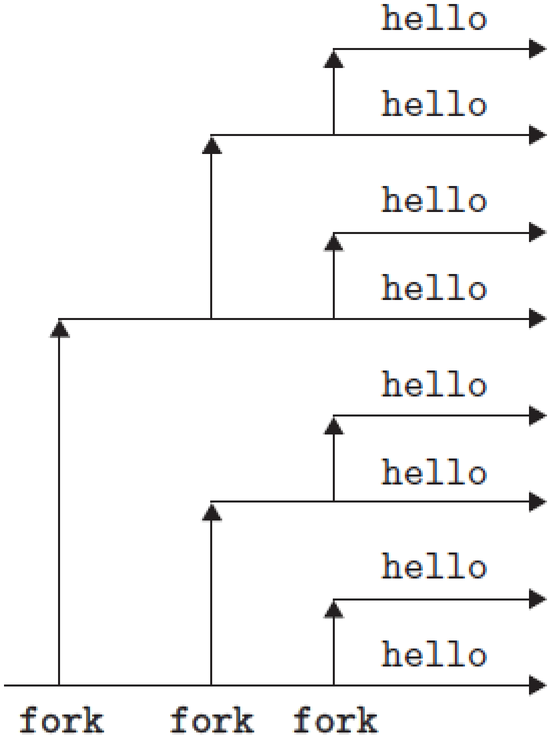
```
#include "csapp.h"

int main()
{
    Fork();
    Fork();
    printf("hello\n");
    exit(0);
}
```

Practice

- **How many lines of output would the program generate**

(e) Calls fork three times

```
#include "csapp.h"

int main()
{
    Fork();
    Fork();
    Fork();
    printf("hello\n");
    exit(0);
}
```

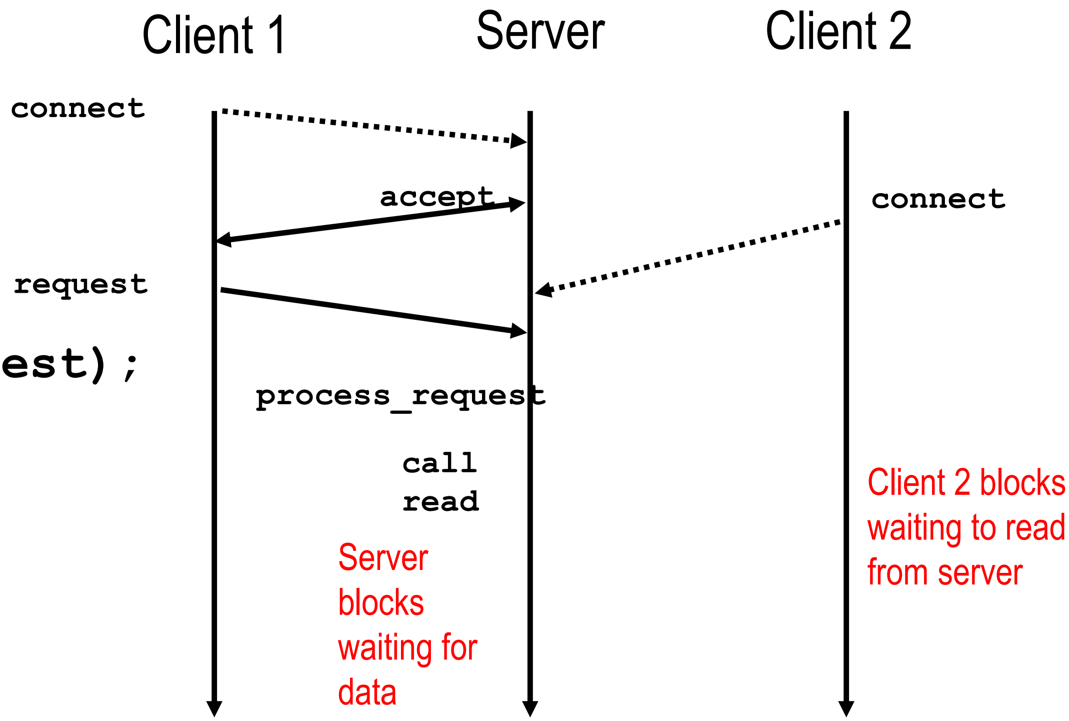


An application – Servers

- Most client requests involve disk accesses
  - File servers
  - Authentications servers
- When this happens, the server remains in the BLOCKED state if the server is iterative
- Cannot handle other customers’ requests
- Analogy
  - *A waitperson that would only be able to wait on one table at a time would be idle most of the time.*

Fundamental Flaw of Iterative Servers

```
for (;;) {
        receive(&client, request);
        process_request(...);
        send (client, reply);
} // for
```


Client 1
Server
Client 2
connect
accept
connect
request
process_request
call
read
Server blocks waiting for data
Client 2 blocks waiting to read from server

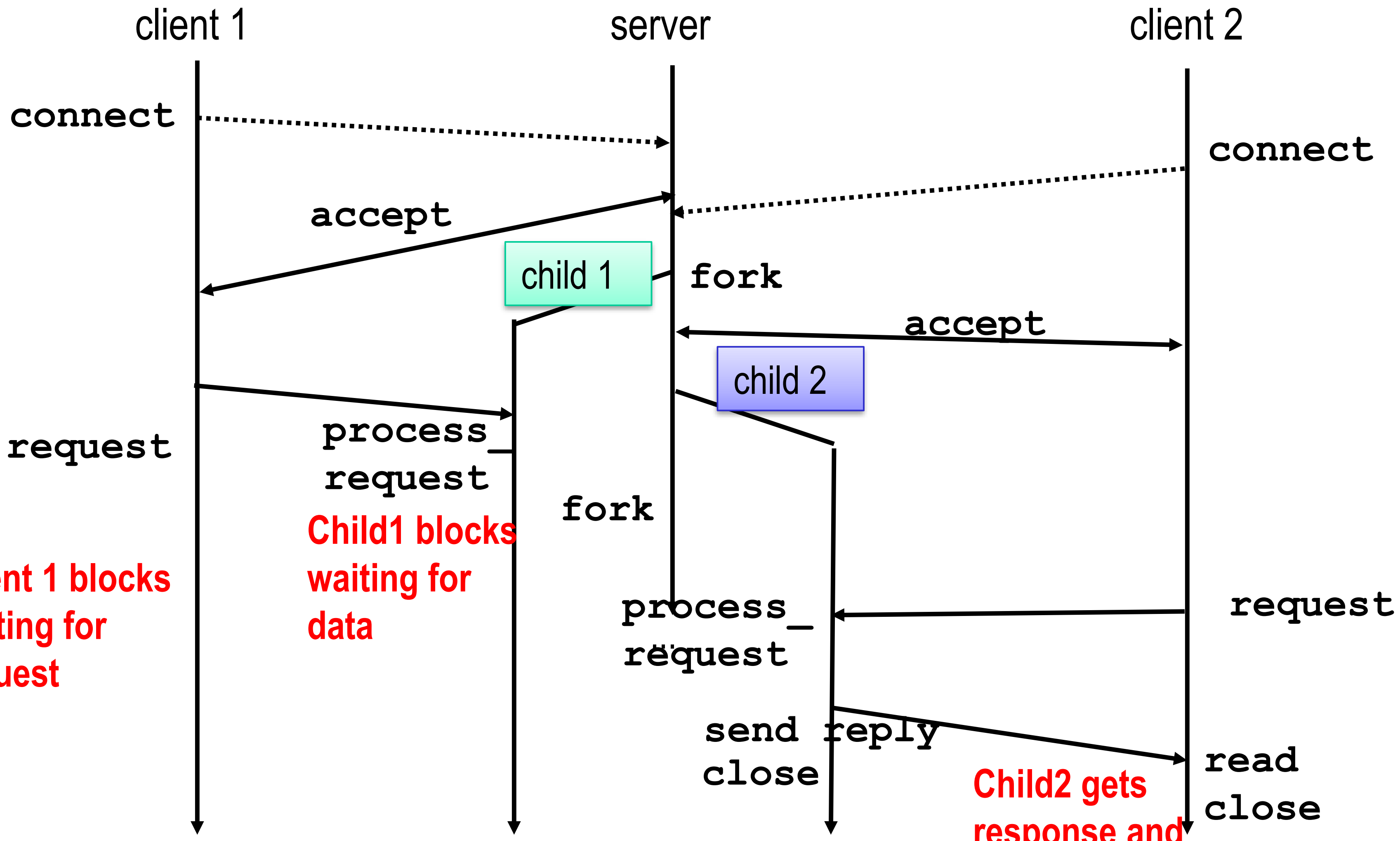
- **Solution: use *concurrent servers* instead**
  - Concurrent servers use multiple concurrent flows to serve multiple clients at the same time

The Simple Solution

```
int pid;
for (;;) {
     receive(&client, request);
     if ((pid = fork())== 0) {
          process_request(...);
          send (client, reply);
          _exit(0); // done
     } // if
} // for
```

The Simple Solution (con't)

- **Spawn separate process for each client**



Child1 blocks waiting for data

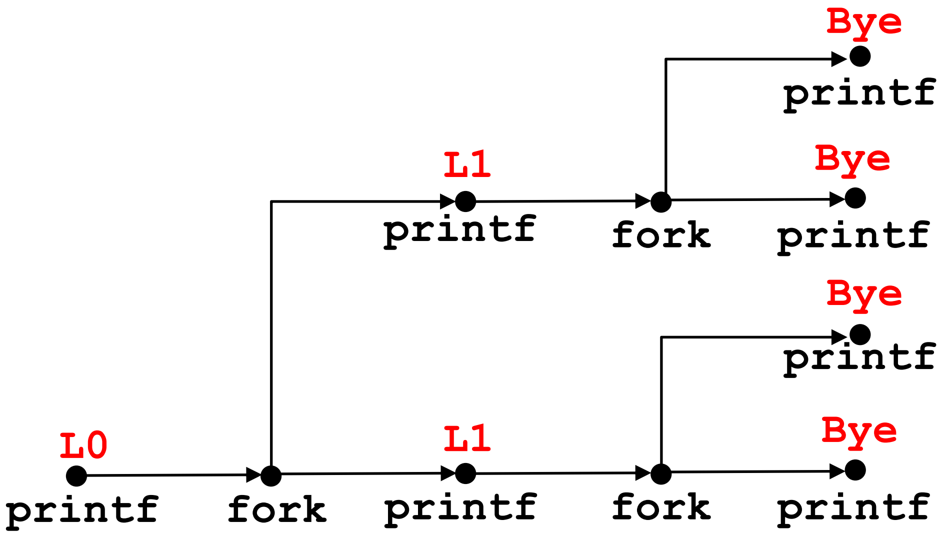
Client 1 blocks waiting for request

Child2 gets response and closes

fork Example: Two consecutive forks

```
void fork2()
{
    printf("L0\n");
    fork();
    printf("L1\n");
    fork();
    printf("Bye\n");
}
```

forks.c

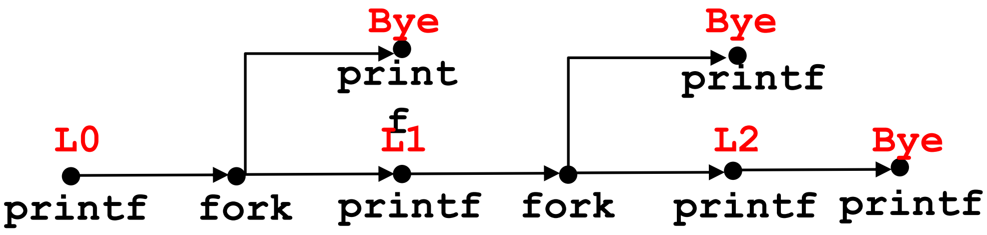


Feasible output:
L0
L1
Bye
Bye
L1
Bye
Bye

Infeasible output:
L0
Bye
L1
Bye
L1
Bye
Bye

`fork` Example: Nested `fork`s in parent

```
void fork4()
{
    printf("L0\n");
    if (fork() != 0) {
        printf("L1\n");
        if (fork() != 0) {
            printf("L2\n");
              }
    }
    printf("Bye\n");
}                                    forks.c
```

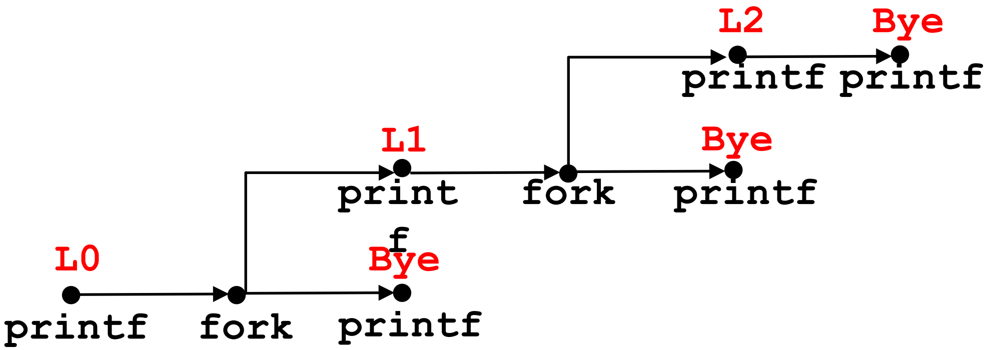


Feasible output:
L0
L1
Bye
Bye
L2
Bye

Infeasible output:
L0
Bye
L1
Bye
Bye
L2

fork Example: Nested forks in children

```
void fork5()
{
    printf("L0\n");
    if (fork() == 0) {
        printf("L1\n");
        if (fork() == 0) {
            printf("L2\n");
        }
    }
    printf("Bye\n");
}
                                    forks.c
```



Feasible output:
L0
Bye
L1
L2
Bye
Bye

Infeasible output:
L0
Bye
L1
Bye
Bye
L2

Today

- System call error handling
- More on fork System Call and Process Graphs
- wait and waitpid system calls
- exec family system calls
- signals

Reaping Child Processes

- **Problem**
  - When process terminates, it still consumes system resources
    - Examples: Exit status, various OS tables
  - Called a "zombie"
    - Living corpse, half alive and half dead

- **Reaping**
  - Performed by parent on terminated child (using `wait` or `waitpid`)
  - Parent is given exit status information
  - Kernel then deletes zombie child process
- **What if parent doesn't reap?**
  - If any parent terminates without reaping a child, then the orphaned child will be reaped by **`init`** process (pid == 1)
  - So, only need explicit reaping in long-running processes
    - e.g., shells and servers

Zombie Example

```
void fork7() {
    if (fork() == 0) {
        /* Child */
        printf("Terminating Child, PID = %d\n", getpid());
        exit(0);
    } else {
        printf("Running Parent, PID = %d\n", getpid());
        while (1); /* Infinite loop */
    }
}
```

forks.c

```
linux> ./forks 7 &
[1] 6639
Running Parent, PID = 6639
Terminating Child, PID = 6640
linux> ps
  PID TTY          TIME CMD
 6585 ttyp9    00:00:00 tcsh
 6639 ttyp9    00:00:03 forks
 6640 ttyp9    00:00:00 forks <defunct>
 6641 ttyp9    00:00:00 ps
linux> kill 6639
[1]    Terminated
linux> ps
  PID TTY          TIME CMD
 6585 ttyp9    00:00:00 tcsh
 6642 ttyp9    00:00:00 ps
```

- **`ps`** shows child process as "defunct" (i.e., a zombie)
- Killing parent allows child to be reaped by **`init`**

Zombie Example

- `ps` –t shows child process as "defunct"
- Killing parent allows child to be reaped

```
UNIX% ./example &
[1] 11299
Running Parent, PID = 11299
Terminating Child, PID = 11300
UNIX% ps x
   PID TTY         STAT   TIME  COMMAND
 11263 pts/7       Ss     0:00  -tcsh
 11299 pts/7       R      0:07  ./example
 11300 pts/7       Z      0:00  […] <defunct>
 11307 pts/7       R+     0:00  ps x
UNIX% kill 11299
[1]    Terminated
UNIX% ps x
   PID TTY         STAT   TIME  COMMAND
 11263 pts/7       Ss     0:00  -tcsh
 11314 pts/7       R+     0:00  ps x
```

First letter:

S: sleeping

T: stopped

R: running/ runnable

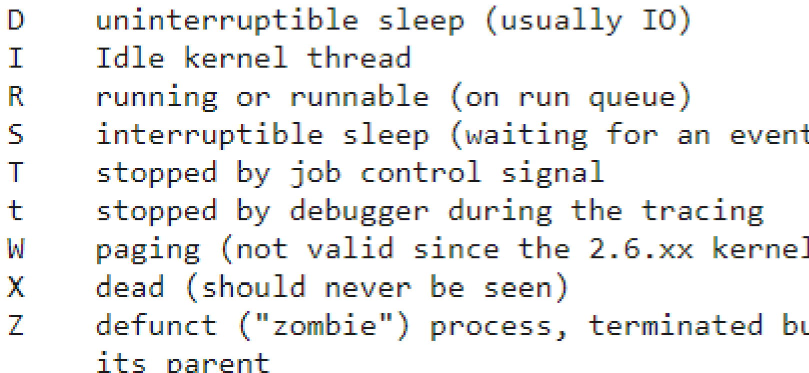
Z: Zombie

Second letter:

s: session leader

+: foreground proc group

Refer to man page for details



```
D    uninterruptible sleep (usually IO)
I    Idle kernel thread
R    running or runnable (on run queue)
S    interruptible sleep (waiting for an event
T    stopped by job control signal
t    stopped by debugger during the tracing
W    paging (not valid since the 2.6.xx kernel
X    dead (should never be seen)
Z    defunct ("zombie") process, terminated bu
     its parent
```

Non-terminating Child Example

```
void fork8()
{
   if (fork() == 0) {
      /* Child */
      printf("Running Child, PID = %d\n",getpid());
      while (1) ; /* Infinite loop */
   } else {
      printf("Terminating Parent, PID = %d\n", getpid());
      exit(0);
   }
}
```

forks.c

```
linux> ./forks 8
Terminating Parent, PID = 6675
Running Child, PID = 6676
linux> ps
  PID TTY          TIME CMD
 6585 ttyp9    00:00:00 tcsh
 6676 ttyp9    00:00:06 forks
 6677 ttyp9    00:00:00 ps
linux> kill -9 6676
linux> ps
  PID TTY          TIME CMD
 6585 ttyp9    00:00:00 tcsh
 6678 ttyp9    00:00:00 ps
```

- Child process still active even though parent has terminated
- Must kill child explicitly, or else will keep running indefinitely

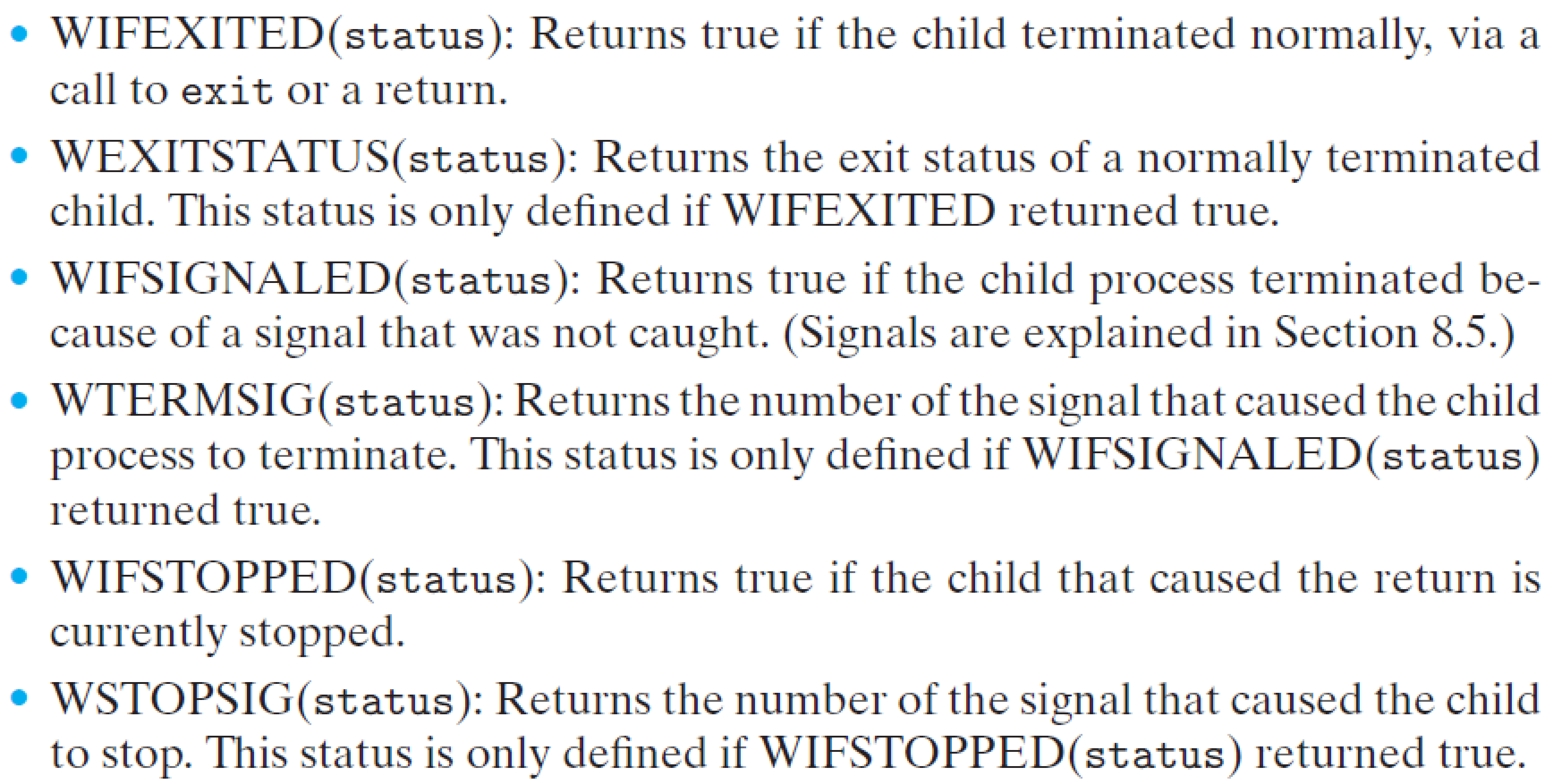
`wait`: Synchronizing with Children

- **Parent reaps a child by calling the `wait` function**

- **`int wait(int *child_status)`**
  - Suspends current process until one of its children terminates
  - Return value is the **`pid`** of the child process that terminated
  - If **`child_status != NULL`**, then the integer it points to will be set to a value that indicates reason the child terminated and the exit status:
    - Checked using macros defined in `wait.h`
      - `WIFEXITED`, `WEXITSTATUS`, `WIFSIGNALED`, `WTERMSIG`, `WIFSTOPPED`, `WSTOPSIG`, `WIFCONTINUED`
      - See textbook for details

`wait`: Synchronizing with Children

You can call them after calling wait to learn the reason of changing state



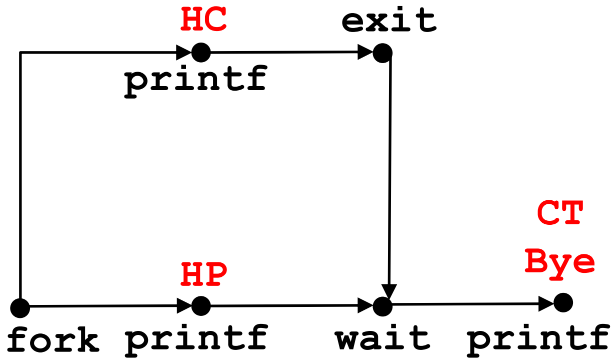
- WIFEXITED(`status`): Returns true if the child terminated normally, via a call to `exit` or a return.
- WEXITSTATUS(`status`): Returns the exit status of a normally terminated child. This status is only defined if WIFEXITED returned true.
- WIFSIGNALED(`status`): Returns true if the child process terminated because of a signal that was not caught. (Signals are explained in Section 8.5.)
- WTERMSIG(`status`): Returns the number of the signal that caused the child process to terminate. This status is only defined if WIFSIGNALED(`status`) returned true.
- WIFSTOPPED(`status`): Returns true if the child that caused the return is currently stopped.
- WSTOPSIG(`status`): Returns the number of the signal that caused the child to stop. This status is only defined if WIFSTOPPED(`status`) returned true.

wait: Synchronizing with Children

```
void fork9() {
    int child_status;

    if (fork() == 0) {
        printf("HC: hello from child\n");
        exit(0);
    } else {
        printf("HP: hello from parent\n");
        wait(&child_status);
        printf("CT: child has terminated\n");
    }
    printf("Bye\n");
}
```

forks.c

Process graph is a useful tool for capturing the partial ordering of statements in a concurrent program:



Feasible output-1:

HC
HP
CT
Bye

Feasible output-2:

HP
HC
CT
Bye

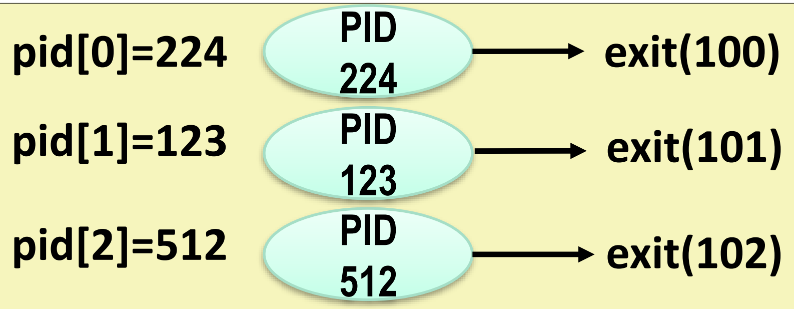
Infeasible output:

HP
CT
Bye
HC

Another wait Example

- If multiple children completed, will take in arbitrary order
- Can use macros WIFEXITED and WEXITSTATUS to get information about exit status

```
void fork10() {
   pid_t pid[N];
   int i, child_status;

   for (i = 0; i < N; i++)
      if ((pid[i] = fork()) == 0) {
         exit(100+i); /* Child */
      }
   for (i = 0; i < N; i++) { /* Parent */
      pid_t wpid = wait(&child_status);
      if (WIFEXITED(child_status))
         printf("Child %d terminated with exit status %d\n",
                wpid, WEXITSTATUS(child_status));
      else
         printf("Child %d terminate abnormally\n", wpid);
   }
}
```



Assume N = 3

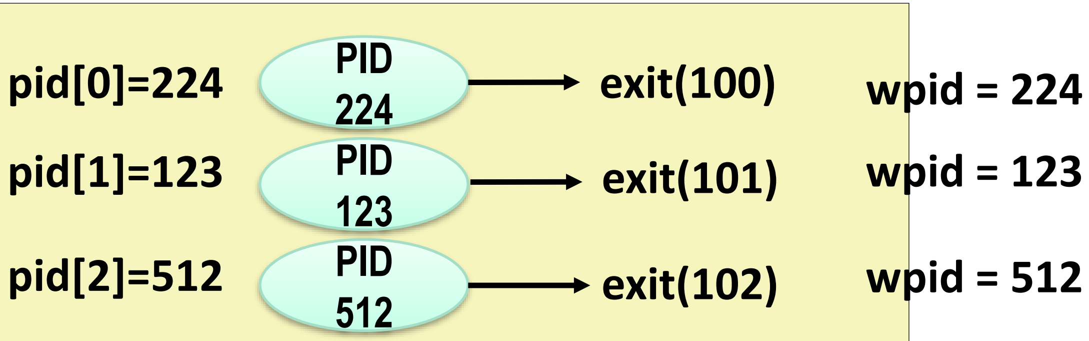
WIFEXITED(status**)**
returns true if the child terminated normally
WEXITSTATUS(status**)**
returns the exit status of the child.

Second for loop will run until all children are reaped in arbitrary order

Another wait Example

- If multiple children completed, will take in arbitrary order
- Can use macros WIFEXITED and WEXITSTATUS to get information about exit status

```
void fork10() {
   pid_t pid[N];
   int i, child_status;

   for (i = 0; i < N; i++)
      if ((pid[i] = fork()) == 0) {
         exit(100+i); /* Child */
      }
   for (i = 0; i < N; i++) { /* Parent */
      pid_t wpid = wait(&child_status);
      if (WIFEXITED(child_status))
         printf("Child %d terminated with exit status %d\n",
             wpid, WEXITSTATUS(child_status));
      else
         printf("Child %d terminate abnormally\n", wpid);
   }
}
```



pid[0]=224 PID 224 exit(100) wpid = 224

pid[1]=123 PID 123 exit(101) wpid = 123

pid[2]=512 PID 512 exit(102) wpid = 512

```
Child 224 is terminated with exit status 100
Child 512 is terminated with exit status 102
Child 123 is terminated with exit status 101
```

waitpid: Waiting for a Specific Process

- `pid_t waitpid(pid_t pid, int &status, int options)`
  - Suspends current process until specific process terminates
  - Various options available

```
void fork11() {
   pid_t pid[N];
   int i;
   int child_status;

   for (i = 0; i < N; i++)
      if ((pid[i] = fork()) == 0)
         exit(100+i); /* Child */
   for (i = N-1; i >= 0; i--) {
      pid_t wpid = waitpid(pid[i], &child_status, 0);
      if (WIFEXITED(child_status))
         printf("Child %d terminated with exit status %d\n",
              wpid, WEXITSTATUS(child_status));
      else
         printf("Child %d terminate abnormally\n", wpid);
   }
}
```

pid[0]=224 PID 224

pid[1]=123 PID 123

pid[2]=512 PID 512

Assume N = 3

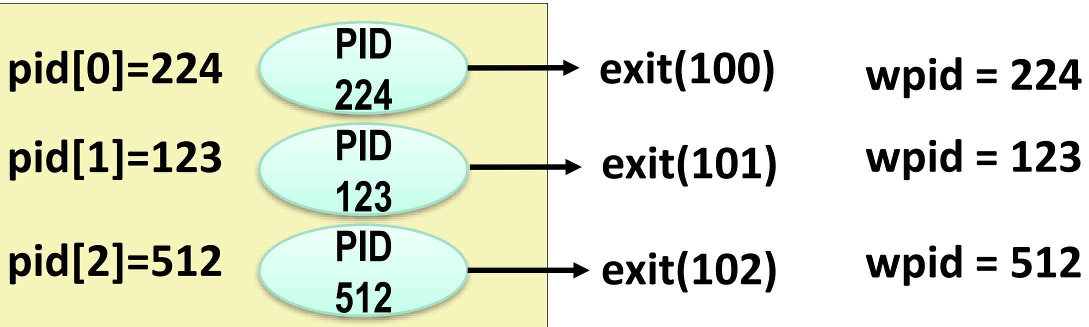
WIFEXITED(status**)**
returns true if the child terminated normally
WEXITSTATUS(status**)**
returns the exit status of the child.

Second for loop will run until all children are reaped in order from last to first

`waitpid`: Waiting for a Specific Process

- `pid_t waitpid(pid_t pid, int &status, int options)`
  - Suspends current process until specific process terminates
  - Various options available

```
void fork11() {
   pid_t pid[N];
   int i;
   int child_status;

   for (i = 0; i < N; i++)
      if ((pid[i] = fork()) == 0)
         exit(100+i); /* Child */
   for (i = N-1; i >= 0; i--) {
      pid_t wpid = waitpid(pid[i], &child_status, 0);
      if (WIFEXITED(child_status))
         printf("Child %d terminated with exit status %d\n",
              wpid, WEXITSTATUS(child_status));
      else
         printf("Child %d terminate abnormally\
   }
}
```



pid[0]=224 PID 224 → exit(100) wpid = 224

pid[1]=123 PID 123 → exit(101) wpid = 123

pid[2]=512 PID 512 → exit(102) wpid = 512

```
Child 512 is terminated with exit status 102
Child 123 is terminated with exit status 101
Child 224 is terminated with exit status 100
```

waitpid options

The value of *options* is an OR of zero or more of the following constants:

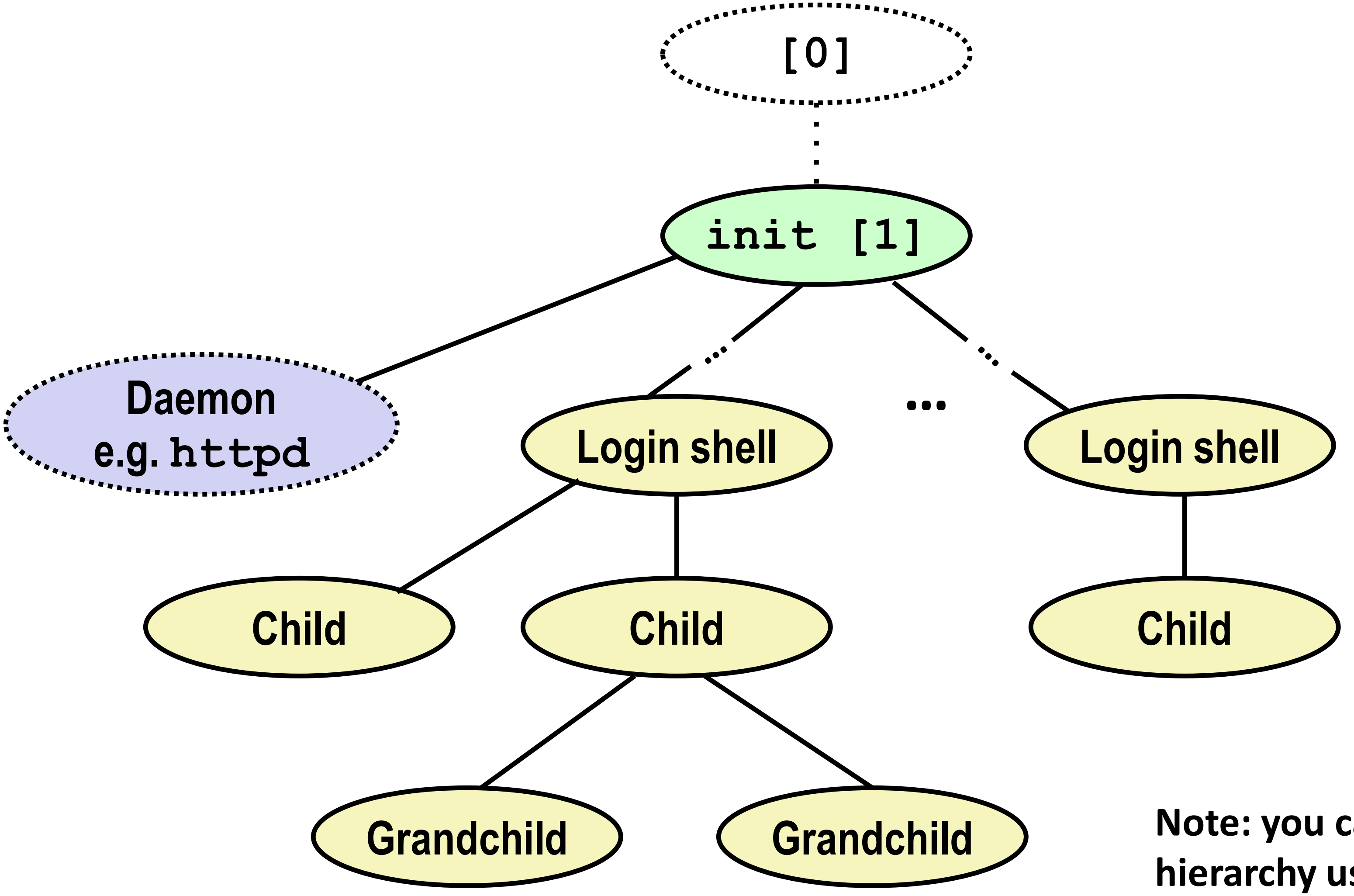
WNOHANG
return immediately if no child has exited.
WUNTRACED
also return if a child has stopped
WCONTINUED (since Linux 2.6.10)
also return if a stopped child has been resumed by delivery of **SIGCONT**.
(For Linux-only options, see below.)

Use man for details

Today

- System call error handling
- More on fork System Call and Process Graphs
- wait and waitpid system calls
- **exec family system calls**
- **signals**

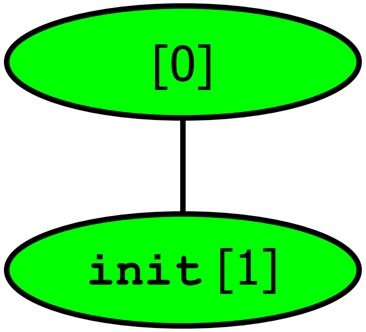
Linux Process Hierarchy



Note: you can view the hierarchy using the Linux `pstree` command

UNIX Startup: 1

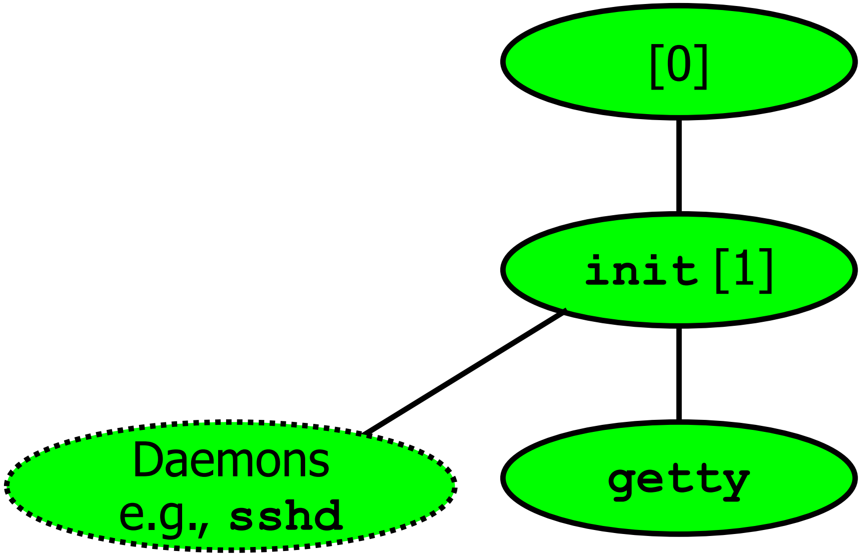
- Pushing reset button loads the PC with the address of a small bootstrap program
- Bootstrap program loads the boot block (disk block 0)
- Boot block program loads kernel from disk
- Boot block program passes control to kernel
- Kernel handcrafts the data structures for process 0



Process 0: handcrafted kernel process

Process 1: user mode process
`fork()` and **`exec(/sbin/init)`**

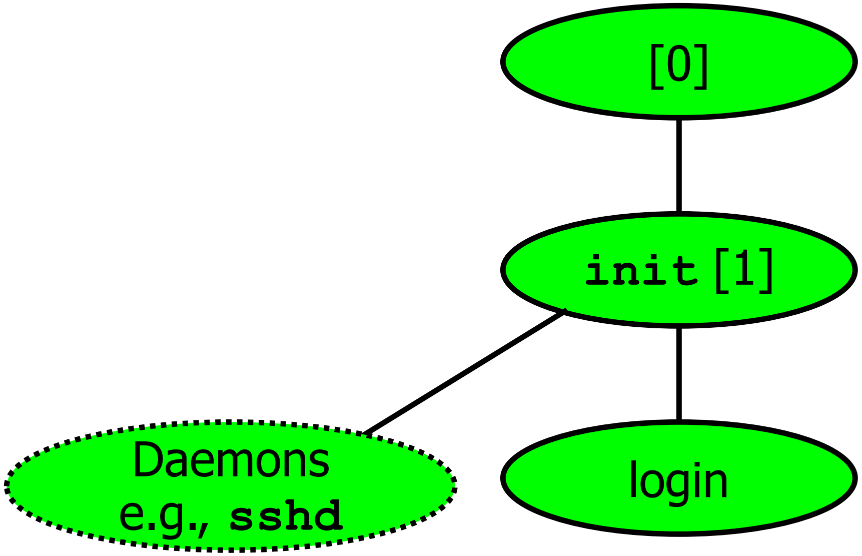
UNIX Startup: 2



init forks new processes as per the **/etc/inittab** file

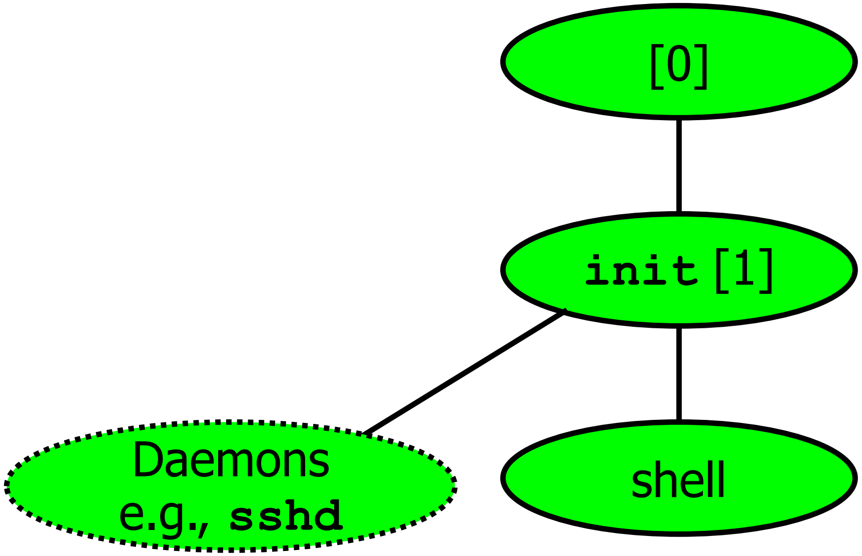
Forks **getty** (get tty or get terminal) for the console

UNIX Startup: 3



`getty` execs a login program

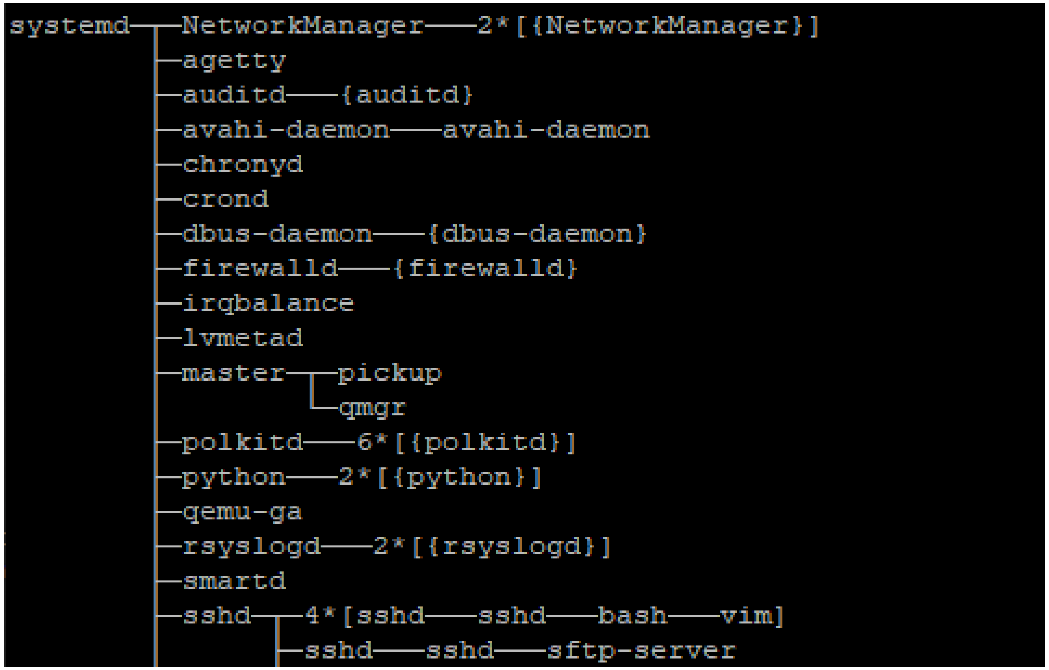
UNIX Startup: 4



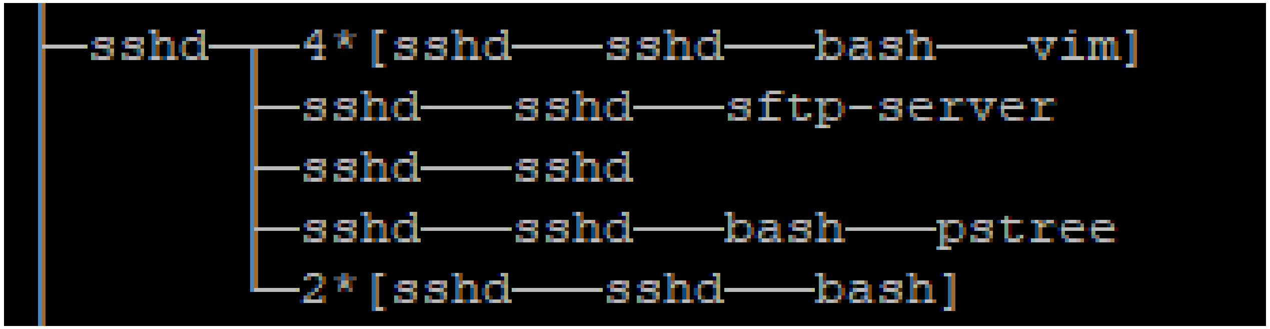
login gets user’s uid & password

- If OK, it execs appropriate shell
- If not OK, it execs **getty**

pstree



```
systemd─┬─NetworkManager───2*[{NetworkManager}]
        ├─agetty
        ├─auditd───{auditd}
        ├─avahi-daemon───avahi-daemon
        ├─chronyd
        ├─crond
        ├─dbus-daemon───{dbus-daemon}
        ├─firewalld───{firewalld}
        ├─irqbalance
        ├─lvmetad
        ├─master─┬─pickup
        │        └─qmgr
        ├─polkitd───6*[{polkitd}]
        ├─python───2*[{python}]
        ├─qemu-ga
        ├─rsyslogd───2*[{rsyslogd}]
        ├─smartd
        ├─sshd─┬─4*[sshd───sshd───bash───vim]
        │      ├─sshd───sshd───sftp-server
```



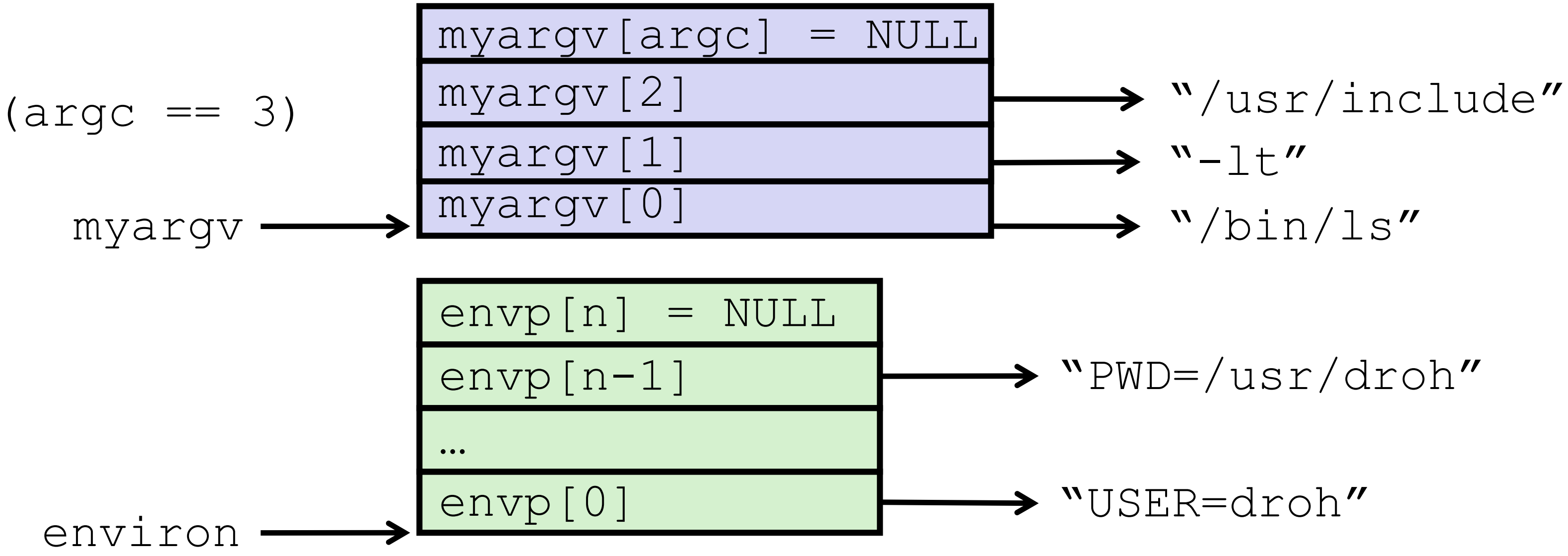
```
├─sshd─┬─4*[sshd───sshd───bash───vim]
│      ├─sshd───sshd───sftp-server
│      ├─sshd───sshd
│      ├─sshd───sshd───bash───pstree
│      └─2*[sshd───sshd───bash]
```

`execve`: Loading and Running Programs

- **`int execve(char *filename, char *argv[], char *envp[])`**
- **Loads and runs in the current process:**
  - Executable file **`filename`**
    - Can be object file or script file beginning with `#!interpreter` (e.g., `#!/bin/bash`)
  - …with argument list **`argv`**
    - By convention **`argv[0]==filename`**
  - …and environment variable list **`envp`**
    - "name=value" strings (e.g., `USER=droh`)
- **Overwrites code, data, and stack**
  - Retains PID, open files and signal context
- **Called once and never returns**
  - …except if there is an error

`execve` Example

- **Executes** `“/bin/ls -lt /usr/include”` **in child process using current environment:**



(argc == 3)

myargv

myargv[argc] = NULL

myargv[2]

myargv[1]

myargv[0]

“/usr/include”

“-lt”

“/bin/ls”

environ

envp[n] = NULL

envp[n-1]

…

envp[0]

“PWD=/usr/droh”

“USER=droh”

```
if ((pid = Fork()) == 0) {   /* Child runs program */
   if (execve(myargv[0], myargv, environ) < 0) {
      printf("%s: Command not found.\n", myargv[0]);
      exit(1);
   }
}
```

exec family function varieties

- **execl**
- **execle**
- **execlp**
- **execv**
- **execve**
- **execvp**

e(environment): It is an array of pointers that points to environment variables and is passed explicitly to the newly loaded process.
l: l is for the command line arguments passed a list to the function
p: p is the path environment variable which helps to find the file passed as an argument to be loaded into process.
v: v is for the command line arguments. These are passed as an array of pointers to the function.

Shell Programs

- A *shell* is an application program that runs programs on behalf of the user.
  - `sh` Original Unix shell (Stephen Bourne, AT&T Bell Labs, 1977)
  - `csh/tcsh` BSD Unix C shell
  - `bash` "Bourne-Again" Shell (default Linux shell)

```
int main()
{
   char cmdline[MAXLINE]; /* command line */

   while (1) {
      /* read */
      printf("> ");
      Fgets(cmdline, MAXLINE, stdin);
      if (feof(stdin))
         exit(0);

      /* evaluate */
      eval(cmdline);
   }
}
```

`shellex.c`

Execution is a sequence of read/evaluate steps

Simple Shell `eval` Function

```
void eval(char *cmdline)
{
    char *argv[MAXARGS]; /* Argument list execve() */
    char buf[MAXLINE];   /* Holds modified command line */
    int bg;              /* Should the job run in bg or fg? */
    pid_t pid;           /* Process id */

    strcpy(buf, cmdline);
    bg = parseline(buf, argv);
    if (argv[0] == NULL)
        return;   /* Ignore empty lines */

    if (!builtin_command(argv)) {
        if ((pid = Fork()) == 0) {   /* Child runs user job */
            if (execve(argv[0], argv, environ) < 0) {
                printf("%s: Command not found.\n", argv[0]);
                exit(0);
            }
        }

        /* Parent waits for foreground job to terminate */
    if (!bg) {
            int status;
            if (waitpid(pid, &status, 0) < 0)
                unix_error("waitfg: waitpid error");
        }
        else
            printf("%d %s", pid, cmdline);
    }
    return;
}
```

parseline,
builtin_command not
shown

parseline updates argv array and returns if process is background

builtin_command, check if one of the built in commands: quit

child is running the user job

Only reap non-background jobs

shellex.c

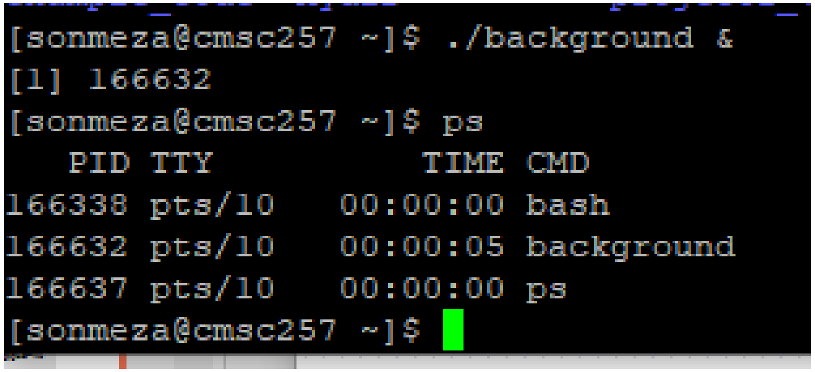
Problem with Simple Shell Example

- **Our example shell correctly waits for and reaps foreground jobs**

- **But what about background jobs?**
  - Will become zombies when they terminate
  - Will never be reaped because shell (typically) will not terminate
  - Will create a memory leak that could run the kernel out of memory

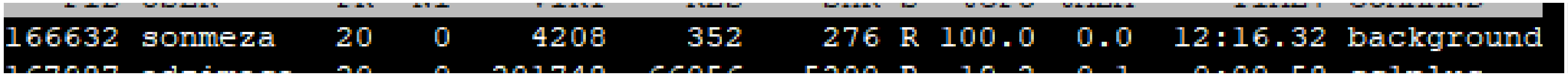
Running a job in background

- Add & to the end
- Or suspend a process running in foreground using CTRL+Z



```
[sonmeza@cmsc257 ~]$ ./background &
[1] 166632
[sonmeza@cmsc257 ~]$ ps
   PID TTY          TIME CMD
166338 pts/10   00:00:00 bash
166632 pts/10   00:00:05 background
166637 pts/10   00:00:00 ps
[sonmeza@cmsc257 ~]$
```

- When you exit and start another session, you will see background is still running.



```
166632 sonmeza   20   0    4208    352    276 R 100.0  0.0  12:16.32 background
```

ECF to the Rescue!

- **Solution: Exceptional control flow**
  - The kernel will interrupt regular processing to alert us when a background process completes
  - In Unix, the alert mechanism is called a ***signal***

Programmer's Model of Multitasking

- **Basic Functions**
  - `fork()` spawns new process
    - Called once, returns twice
  - `exit()` terminates own process
    - Called once, never returns
    - Puts process into "zombie" status
  - `wait()` and `waitpid()` wait for and reap terminated children
  - `execl()` and `execve()` run a new program in an existing process
    - Called once, (normally) never returns
- **Programming Challenge**
  - Understanding the nonstandard semantics of the functions
  - Avoiding improper use of system resources
    - E.g., "Fork bombs" can disable a system

Today

- System call error handling
- More on fork System Call and Process Graphs
- wait and waitpid system calls
- exec family system calls
- **Signals**
  - Will be covered next