CSE251: System Programming

22. Process (2)

Seongil Wi

Department of Computer Science and Engineering

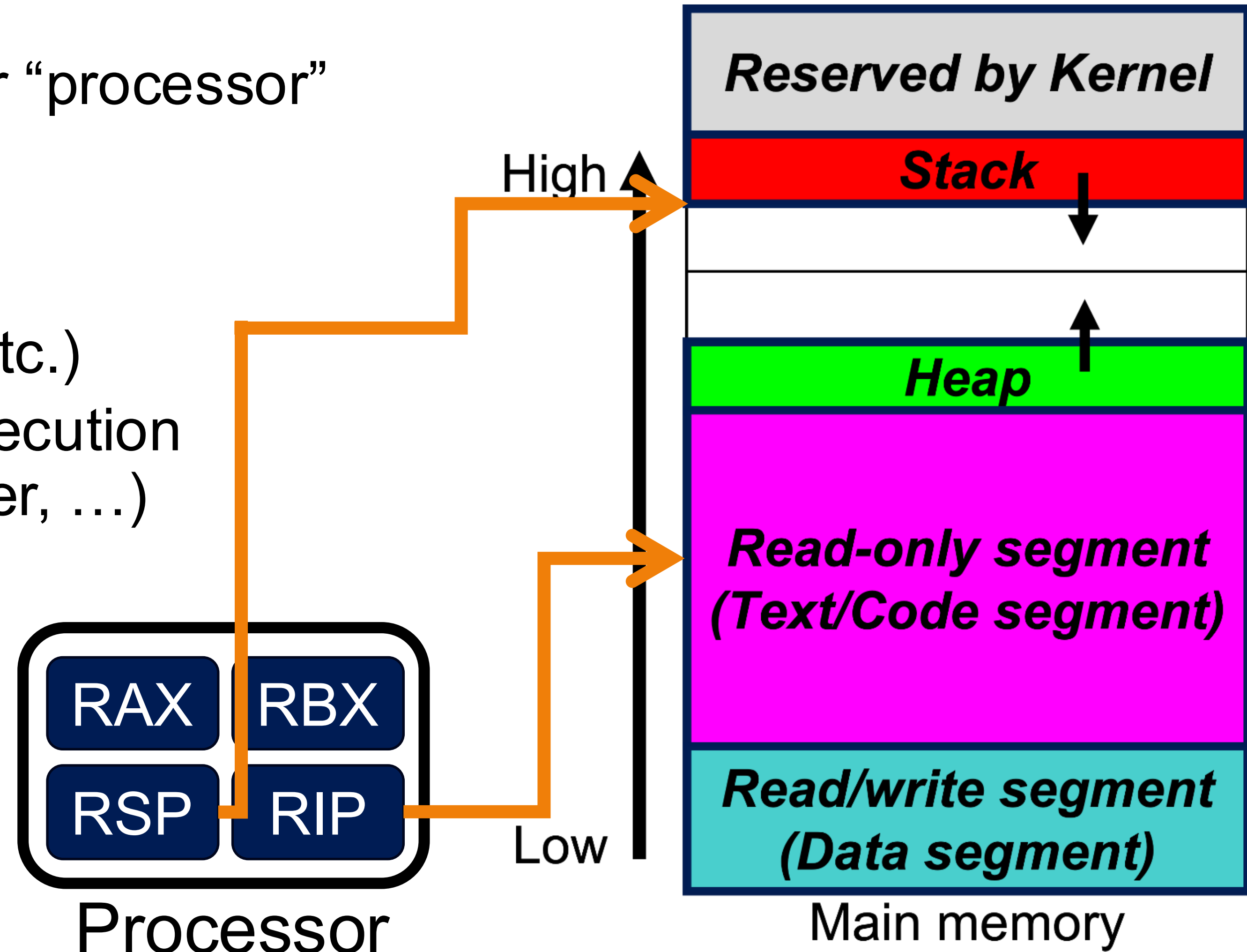
UNIST
ULSAN NATIONAL INSTITUTE OF SCIENCE AND TECHNOLOGY
2007

Recap: Process

An **instance of a program** that is being executed (or is ready for execution)

- Not the same as "program" or "processor"

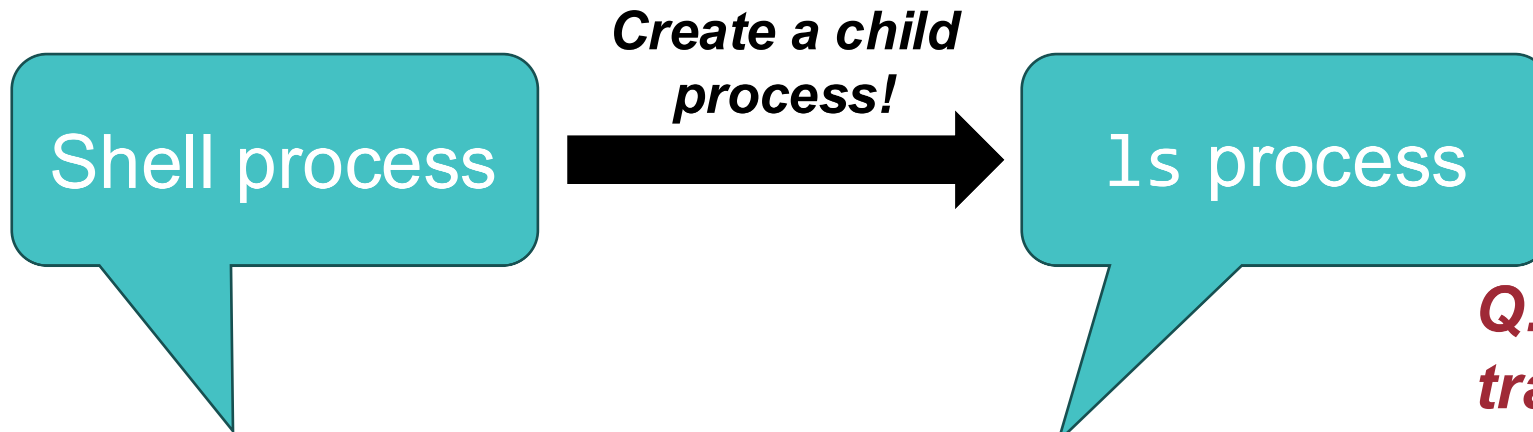
- Consists of:
  - Memory (code, heap, stack, etc.)
  - Registers used to manage execution (stack pointer, program counter, …)
  - Other resources



Question!

Q. How can a user program create, track, and terminate processes?

Example: A Shell



Q. How can our program create, track, and terminate processes?

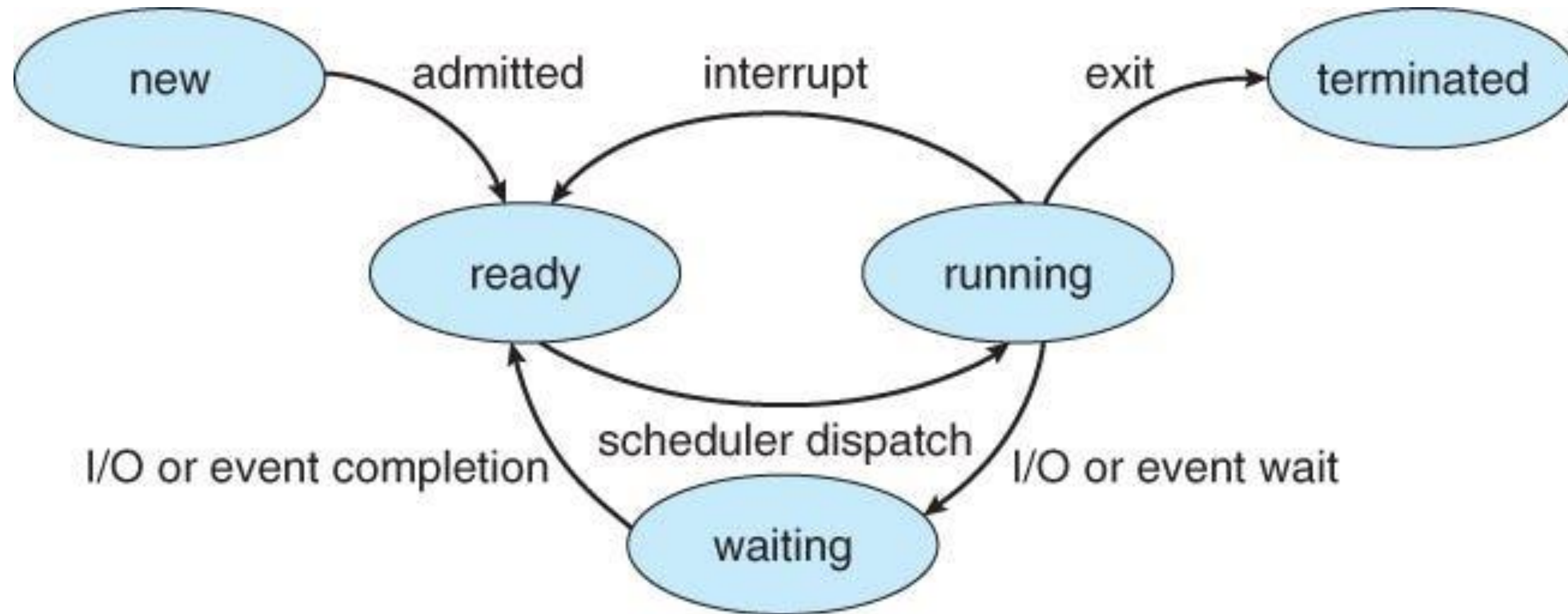
```
s20265006@3affb2a86ea8:~$ ls -al
total 1960
drwxrwxr-x 24 s20265006 s20265006    4096 Jun  2 03:00 .
drwxr-xr-x  1 root      root         4096 Mar  4 10:36 ..
-rw-------  1 s20265006 s20265006     406 May 26 04:21 .Xauthority
-rw-------  1 s20265006 s20265006   22604 Jun  2 03:00 .bash_history
-rw-r--r--  1 s20265006 s20265006     220 Mar  4 10:36 .bash_logout
-rw-r--r--  1 s20265006 s20265006    3855 Mar 28 07:37 .bashrc
drwx------  3 s20265006 s20265006    4096 Mar  4 08:28 .cache
```

Process Control

Lifecycle of a Process

- Ready
  - Process is ready to be scheduled by the kernel
- Running
  - Process is executing
- Waiting
  - Waiting for I/O or events to be completed
- Terminated
  - Process is stopped permanently

Lifecycle of a Process



Obtaining Process IDs (from Your Program)

```
pid_t getpid(void);
```

- Returns PID of current process

```
pid_t getppid(void);
```

- Returns PID of parent process

Obtaining Process IDs (from the Shell)

- Running "top" on your environment!

```
top - 10:57:12 up 42 days, 17:53,  1 user,  load average: 2.01, 2.02, 2.00
Tasks: 702 total,   3 running, 697 sleeping,   2 stopped,   0 zombie
%Cpu(s):  0.4 us,  5.8 sy,  0.0 ni, 93.7 id,  0.0 wa,  0.0 hi,  0.0 si,  0.0 st
MiB Mem : 515851.0 total,   1663.2 free,   4095.0 used, 510092.8 buff/cache
MiB Swap:  30518.0 total,  30512.0 free,      6.0 used. 508397.0 avail Mem

    PID USER      PR  NI    VIRT    RES    SHR S  %CPU  %MEM     TIME+ COMMAND
4056347 compu     20   0   44756  27036   3560 R 100.0   0.0  11764:27 lackey-amd64-li
4061301 compu     20   0   44756  27036   3560 R  99.7   0.0  11759:11 lackey-amd64-li
1550125 su3604    20   0   13812   4676   3496 R   0.7   0.0   0:00.07 top
1365471 su3604    20   0   14120   5856   4364 S   0.3   0.0   0:00.65 sshd
1542286 root      20   0       0      0      0 I   0.3   0.0   0:03.33 kworker/u64:0-edac-poller
1550032 root      20   0       0      0      0 I   0.3   0.0   0:00.01 kworker/24:1-events
      1 root      20   0  173920  18356   8528 S   0.0   0.0   1:01.53 systemd
      2 root      20   0       0      0      0 S   0.0   0.0   0:06.84 kthreadd
      3 root       0 -20       0      0      0 I   0.0   0.0   0:00.00 rcu_gp
      4 root       0 -20       0      0      0 I   0.0   0.0   0:00.00 rcu_par_gp
      6 root       0 -20       0      0      0 I   0.0   0.0   0:00.00 kworker/0:0H-events_highpri
     10 root       0 -20       0      0      0 I   0.0   0.0   0:00.00 mm_percpu_wq
     11 root      20   0       0      0      0 S   0.0   0.0   0:00.00 rcu_tasks_rude_
```

Terminating Processes

- Process becomes terminated for one of three reasons:
  - Receiving a signal whose default action is to terminate (next lecture!)
  - Returning from the **`main`** routine
  - Calling the **`exit`** function

- `void exit(int status)`
  - Terminates with an *exit status* of **`status`**
  - Convention: normal return status is 0, nonzero on error

- `exit` is called **once** but **never** returns

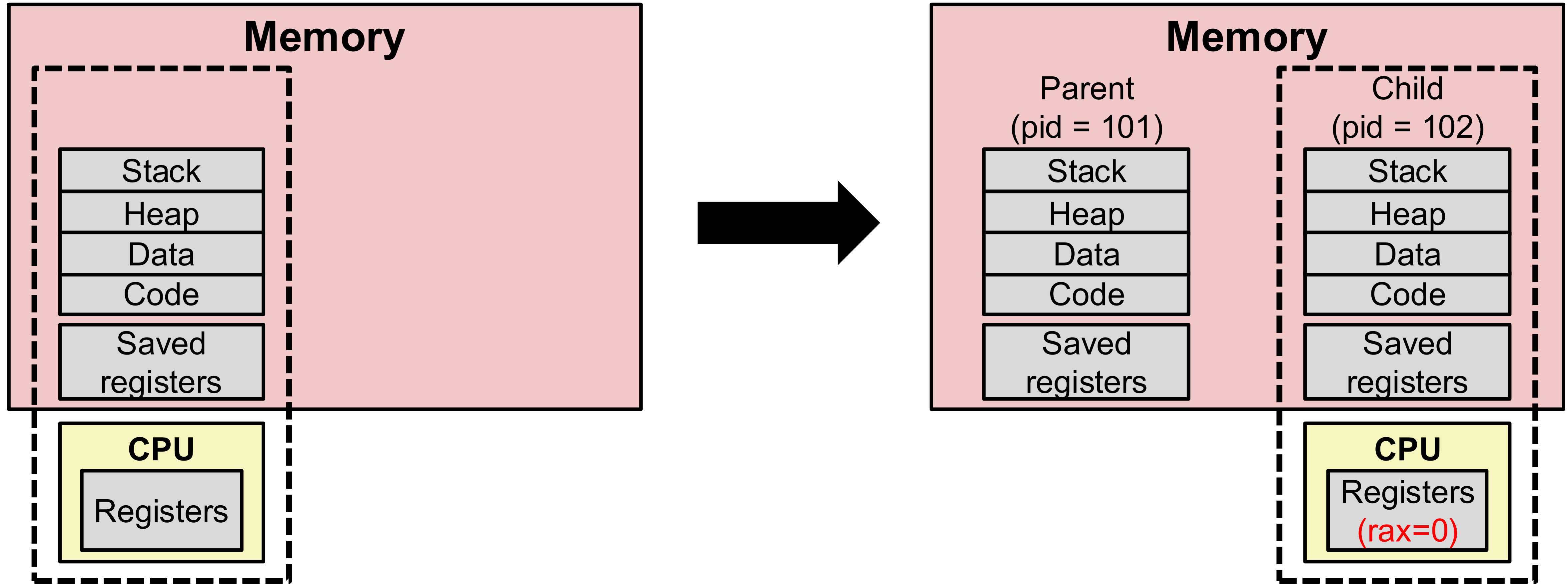
Crating Processes

- ***Parent process*** creates a new running ***child process*** by calling **`fork`**

- `int fork(void)`
  - Returns
    - 0 to the child process
    - child's PID to parent process
  - Child is almost identical to parent:
    - Childs get an **identical copy of the parent's virtual address space**
    - Child gets **identical copies of the parent's open file descriptors**
    - Child has a different PID than the parent

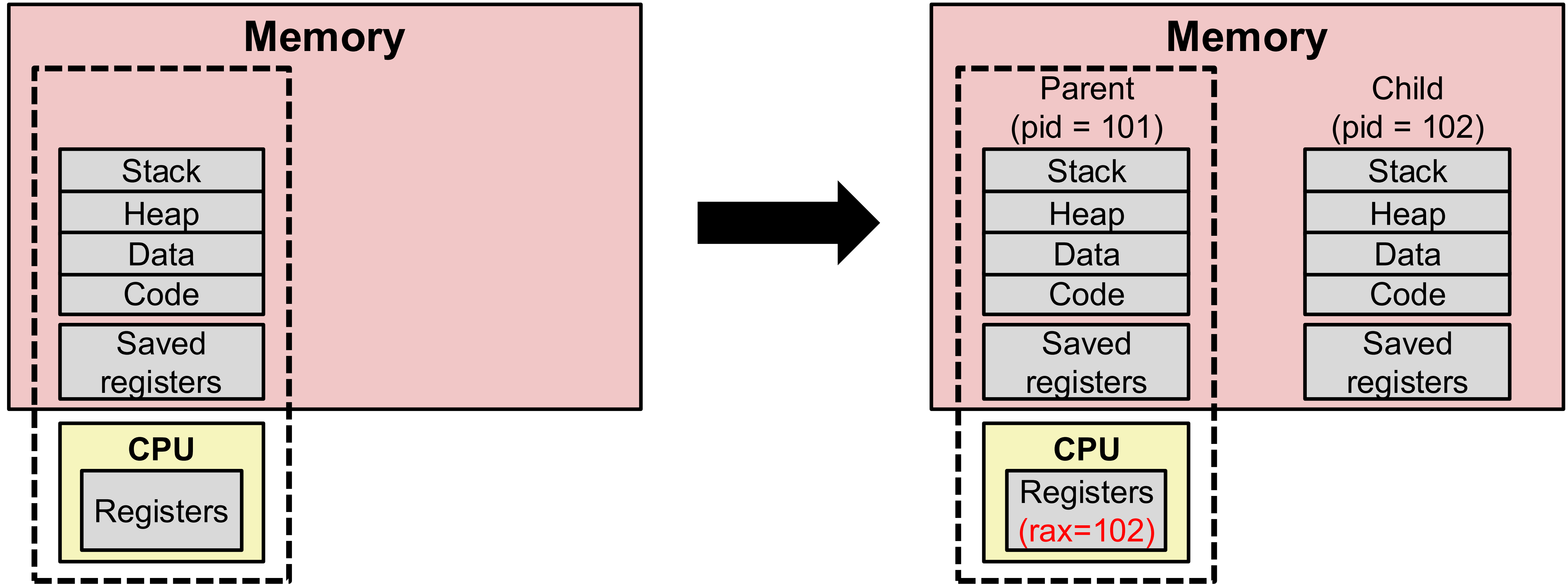
- **`fork`** is interesting (and often confusing) because it is called ***once*** but returns ***twice***

Conceptual View of fork()



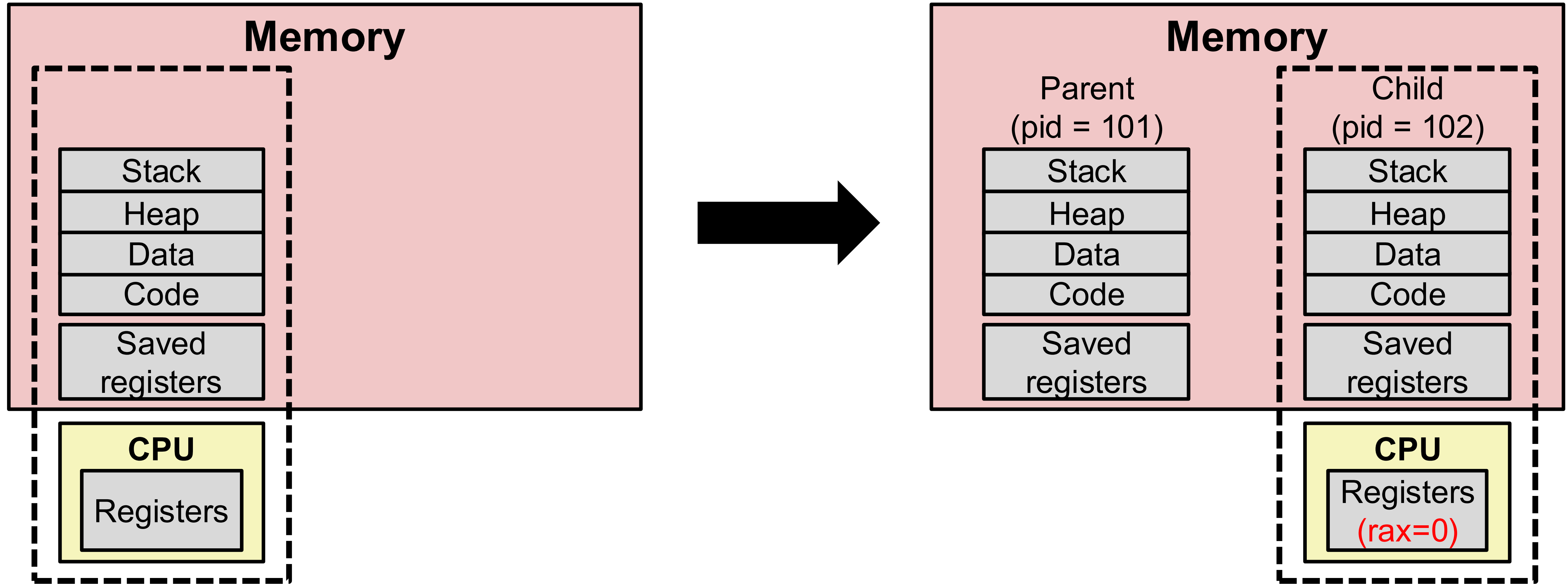
- Make complete copy of execution state
  - Designate one as parent and one as child
  - Resume execution of parent or child

Conceptual View of fork()



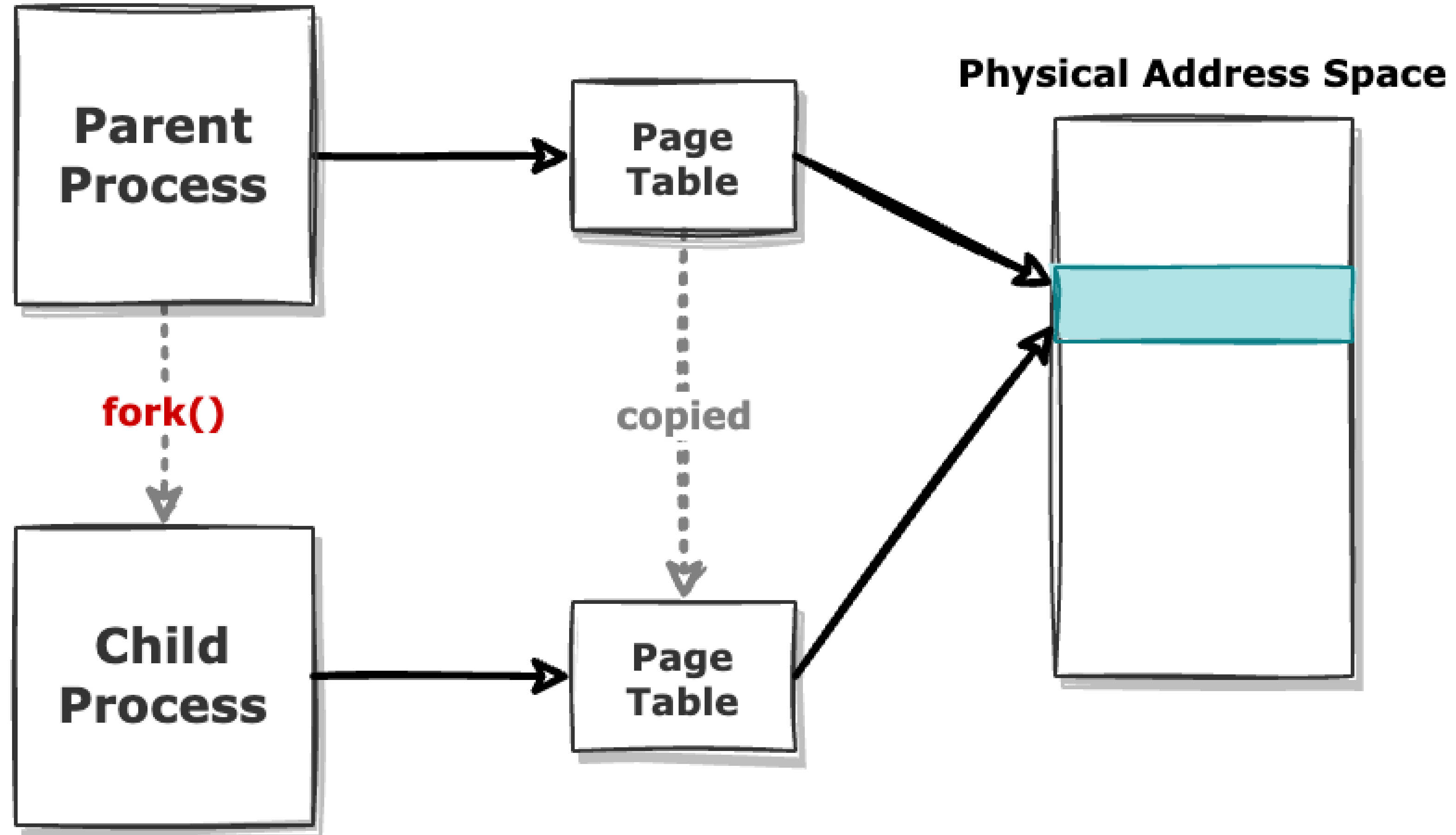
- Make complete copy of execution state
  - Designate one as parent and one as child
  - Resume execution of parent or child

Conceptual View of fork()



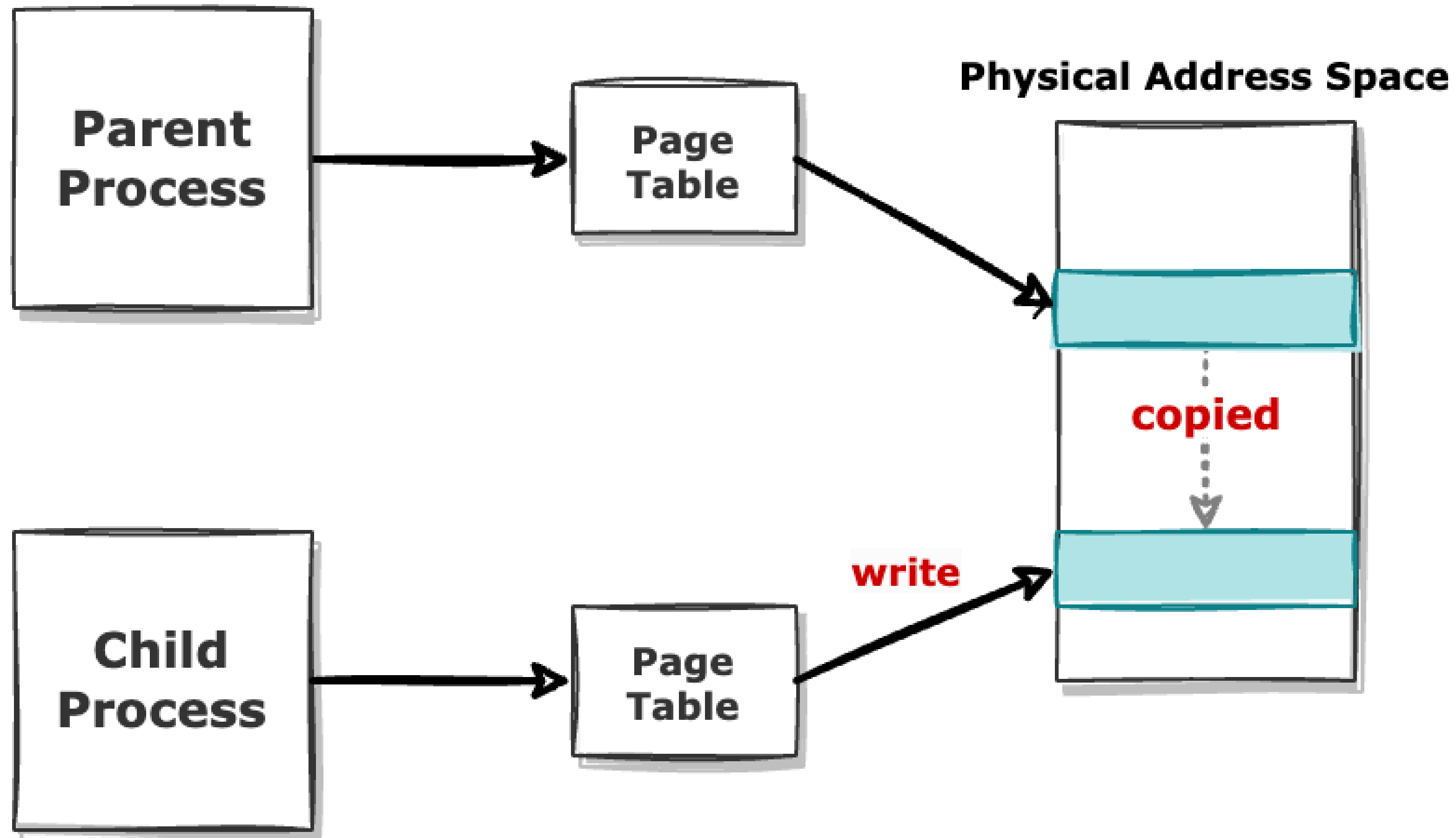
- Make complete copy of execution state
  - Designate one as parent and one as child
  - Resume execution of parent or child

Physical View of fork()



Physical View of fork()

- Optimization: use copy-on-write to avoid copying RAM



The fork() Function Summary

- Virtual memory is the key for **fork** to provide private address space for each process
- To create virtual address for new process:
  - Create ***exact copies*** of current page tables
- On return of fork(), two processes have the same virtual memory

- **Copy-on-write (COW)**
  - Should the kernel keep individual physical copies of memory of these two identical processes?
  - **Main idea:** Let's copy only when required
    - In details, any write creates new pages using copy-on-write (COW)

fork() Example


```
#include<unistd.h>
int main() {
  pid_t pid;
  int x = 1;

  pid = fork();
  if (pid == 0) { /* Child */
    printf("child: x=%d\n", ++x);
    return 0;
  } else {        /* Parent */
    printf("parent: x=%d\n", --x);
    return 0;
  }
}
```

```
$ ./fork
```

Q. What would be printed?

fork() Example


```
#include<unistd.h>
int main() {
  pid_t pid;
  int x = 1;

  pid = fork();
  if (pid == 0) { /* Child */
    printf("child: x=%d\n", ++x);
    return 0;
  } else {          /* Parent */
    printf("parent: x=%d\n", --x);
    return 0;
  }
}
```

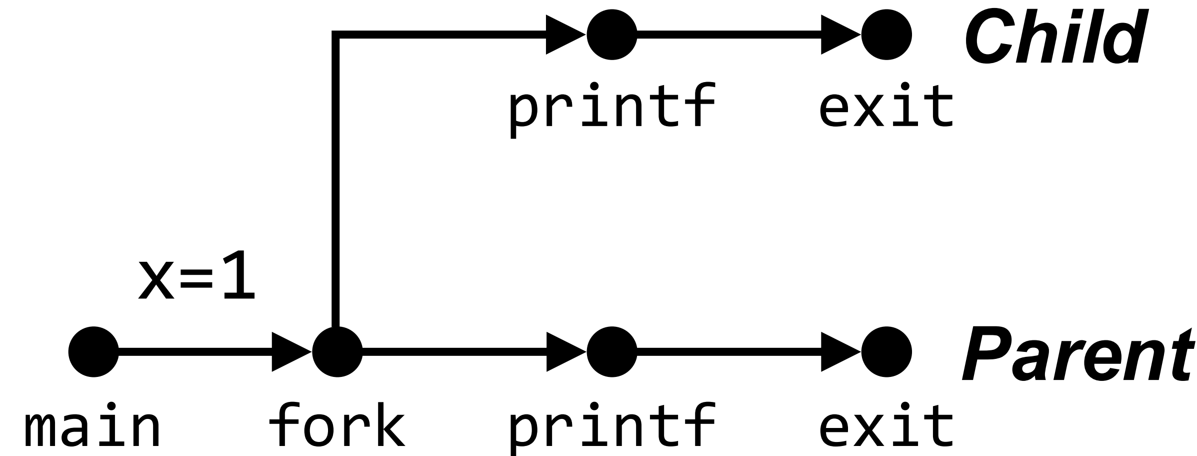
- Call once, **return twice**
- **Concurrent execution**
  - Can't predict **execution order of parent and child**
- **Duplicate but separate address space**
  - **x** has a value of 1 when fork returns in parent and child
  - Subsequent changes to x are independent
- **Shared open files**
  - `stdout` 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 (node) is the execution of a statement
  - a → b means ***a happens before b***
  - Each graph begins with a vertex with no in-edges
- Any ***topological sort (ordering)*** of the graph corresponds to a feasible total ordering
  - Total ordering of vertices where all edges point from left to right
  - This makes easier to understand
    - all feasible task ordering
    - non-feasible task ordering

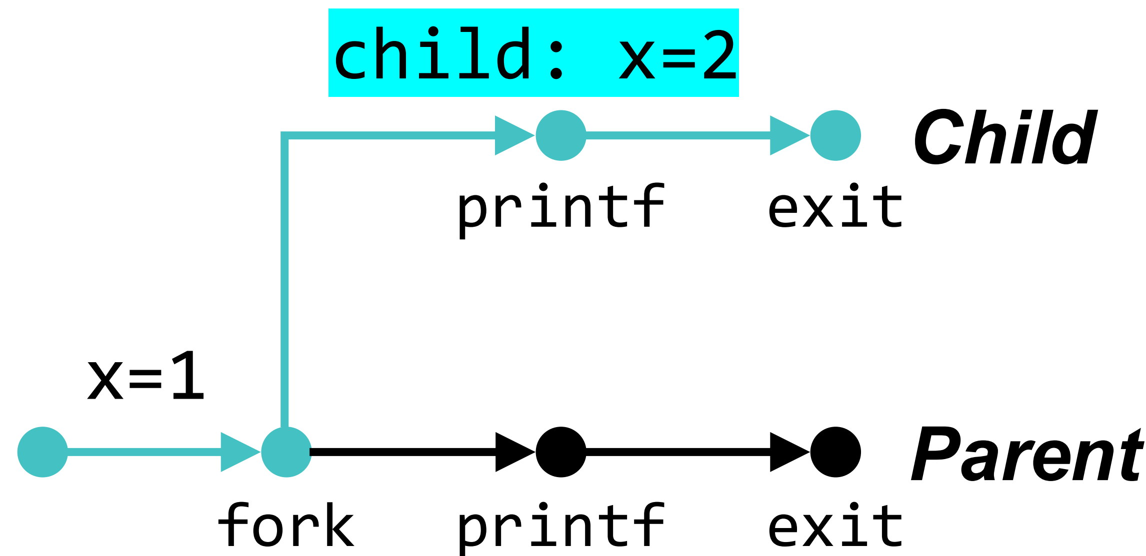
Process Graph Example


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

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

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



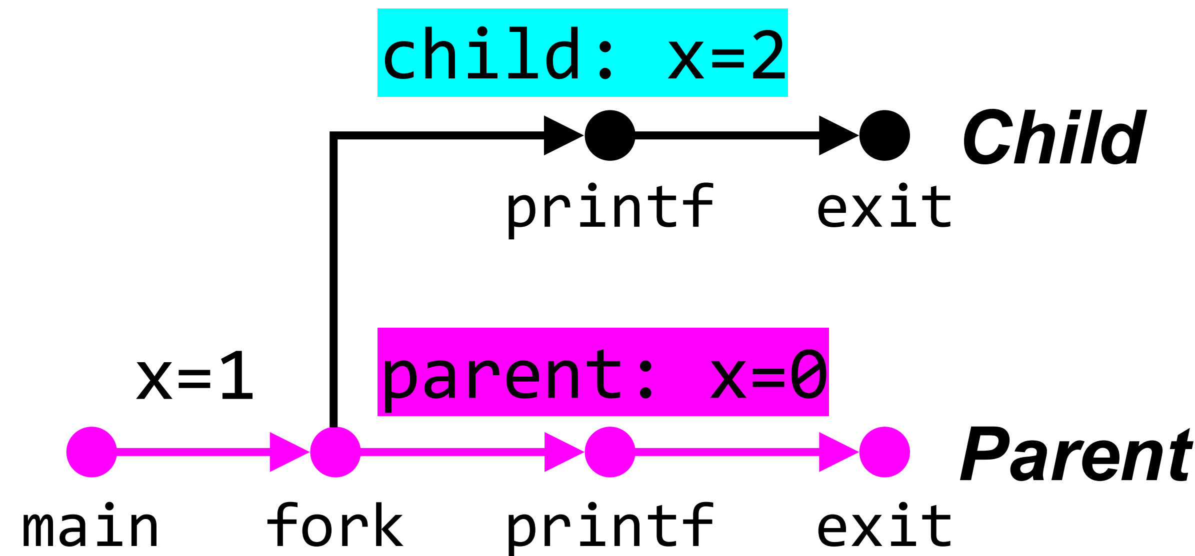
Process Graph Example


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

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

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

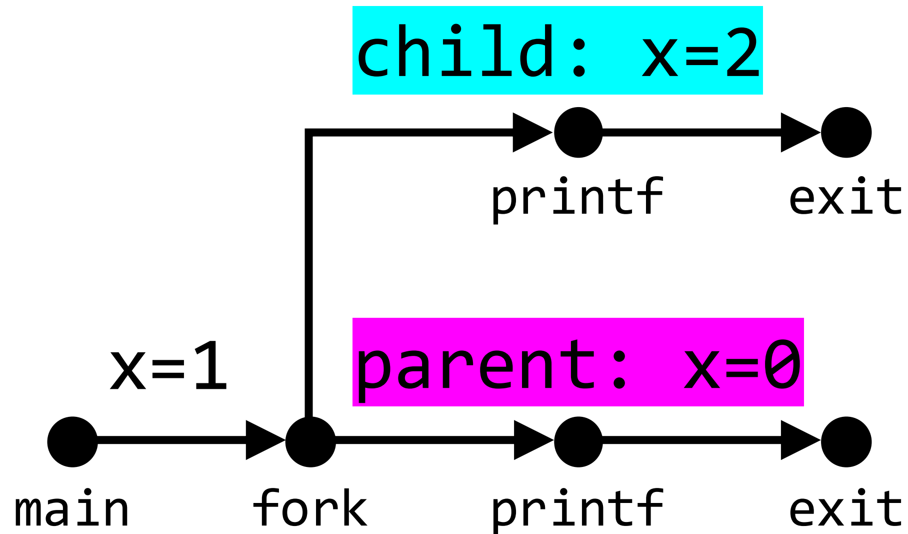


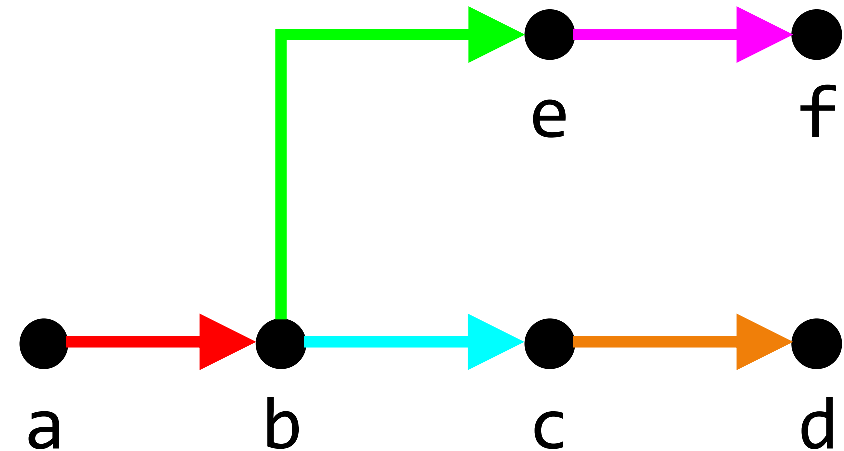
Process Graph Example

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

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

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

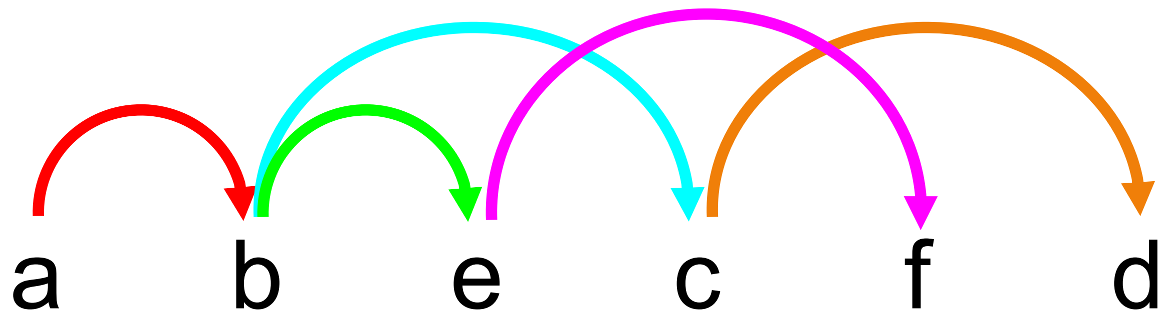


Interpreting Process Grpahs





Feasible total ordering:



Reaping Child Processes

- **Idea**
  - 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`**)
  - <u>Parent is given exit status information</u>
  - Kernel then deletes zombie child process

- **What if parent doesn't reap? (Problematic situation)**
  - Only need <u>explicit reaping in long-running processes</u>
    - e.g., shells and servers

Zombie Example

```
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
```

```
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

Zombie Example


```
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
```

```
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

ps shows child process as "defunct" (i.e., a zombie)

Zombie Example

```
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
```

```c
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

ps shows child process as "defunct" (i.e., a zombie)

Killing parent allows child to be reaped by **init** ☺

Non-terminating Child Example (Problem!)

```
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 6676

linux> ps
  PID TTY          TIME CMD
 6585 ttyp9    00:00:00 tcsh
 6678 ttyp9    00:00:00 ps
```

```
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);
  }
}
```

Child process still active even though parent has terminated

Must kill child explicitly, or else will keep running indefinitely ☹

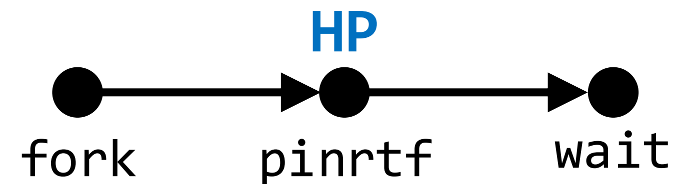
`wait`: Synchronizing with Children

- Parent reaps a child by calling the `wait` syscall

- `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 it will be set to a value indicating reason the child terminated and the exit status

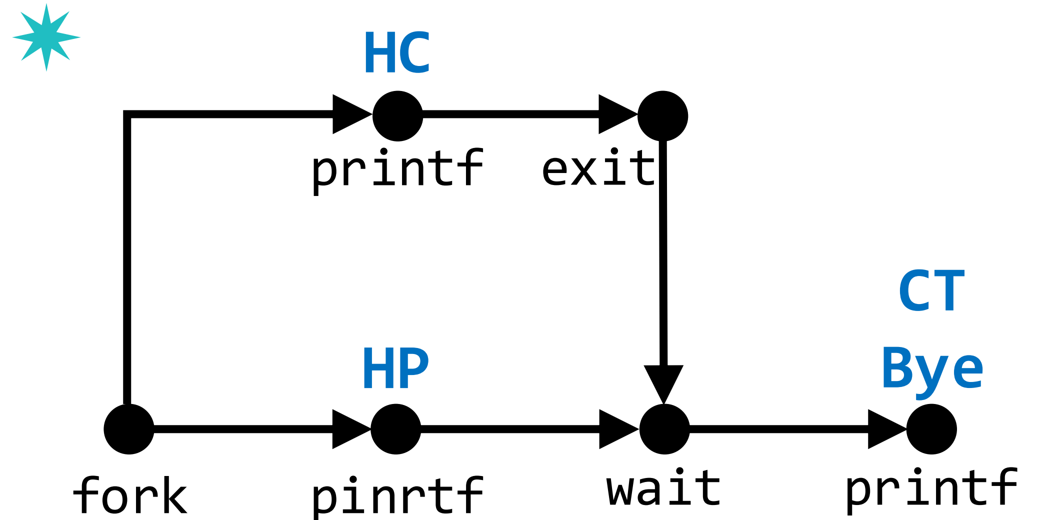
wait: Synchronizing with Children


```
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");
}
```



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");
}
```



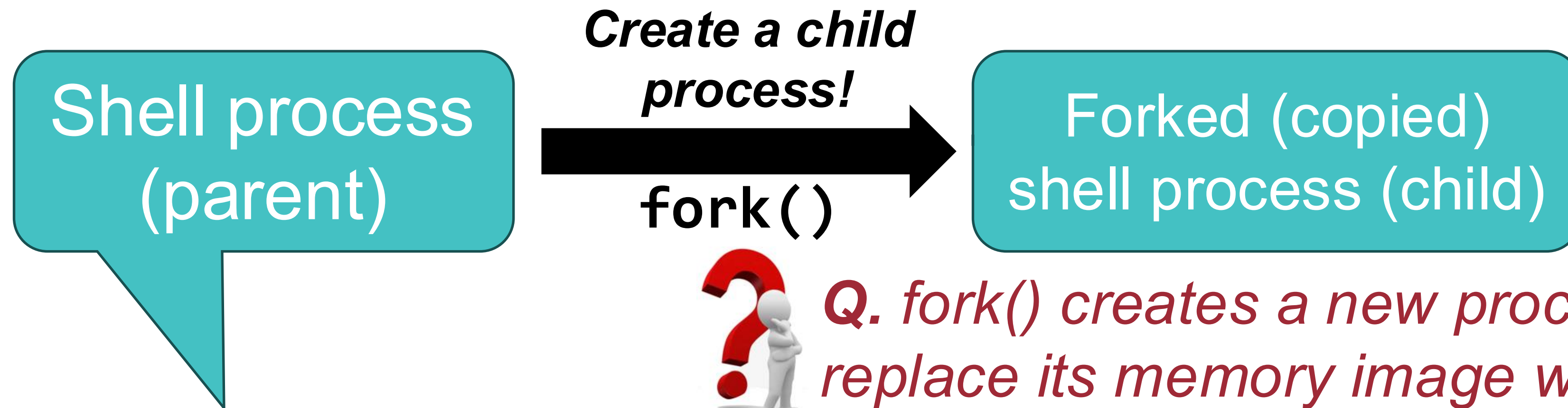
Feasible output(s):

HC	HP
HP	HC
CT	CT
Bye	Bye

Feasible or infeasible?

HP
CT
Bye
HC

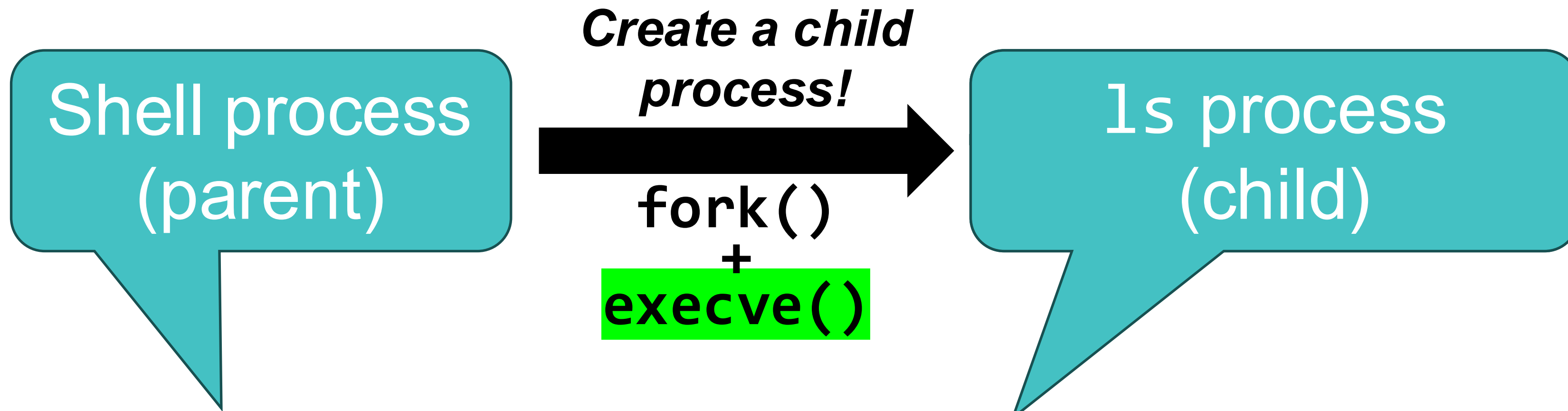
Recall: A Shell



***Q.** fork() creates a new process as a copy. How can we replace its memory image with a new program (e.g., `ls`)?*

```
s20265006@3affb2a86ea8:~$ ls -al
total 1960
drwxrwxr-x 24 s20265006 s20265006    4096 Jun  2 03:00 .
drwxr-xr-x  1 root      root         4096 Mar  4 10:36 ..
-rw-------  1 s20265006 s20265006     406 May 26 04:21 .Xauthority
-rw-------  1 s20265006 s20265006   22604 Jun  2 03:00 .bash_history
-rw-r--r--  1 s20265006 s20265006     220 Mar  4 10:36 .bash_logout
-rw-r--r--  1 s20265006 s20265006    3855 Mar 28 07:37 .bashrc
drwx------  3 s20265006 s20265006    4096 Mar  4 08:28 .cache
```

execve: Loading and Running Programs




```
s20265006@3affb2a86ea8:~$ ls -al
total 1960
drwxrwxr-x 24 s20265006 s20265006    4096 Jun  2 03:00 .
drwxr-xr-x  1 root      root         4096 Mar  4 10:36 ..
-rw-------  1 s20265006 s20265006     406 May 26 04:21 .Xauthority
-rw-------  1 s20265006 s20265006   22604 Jun  2 03:00 .bash_history
-rw-r--r--  1 s20265006 s20265006     220 Mar  4 10:36 .bash_logout
-rw-r--r--  1 s20265006 s20265006    3855 Mar 28 07:37 .bashrc
drwx------  3 s20265006 s20265006    4096 Mar  4 08:28 .cache
```

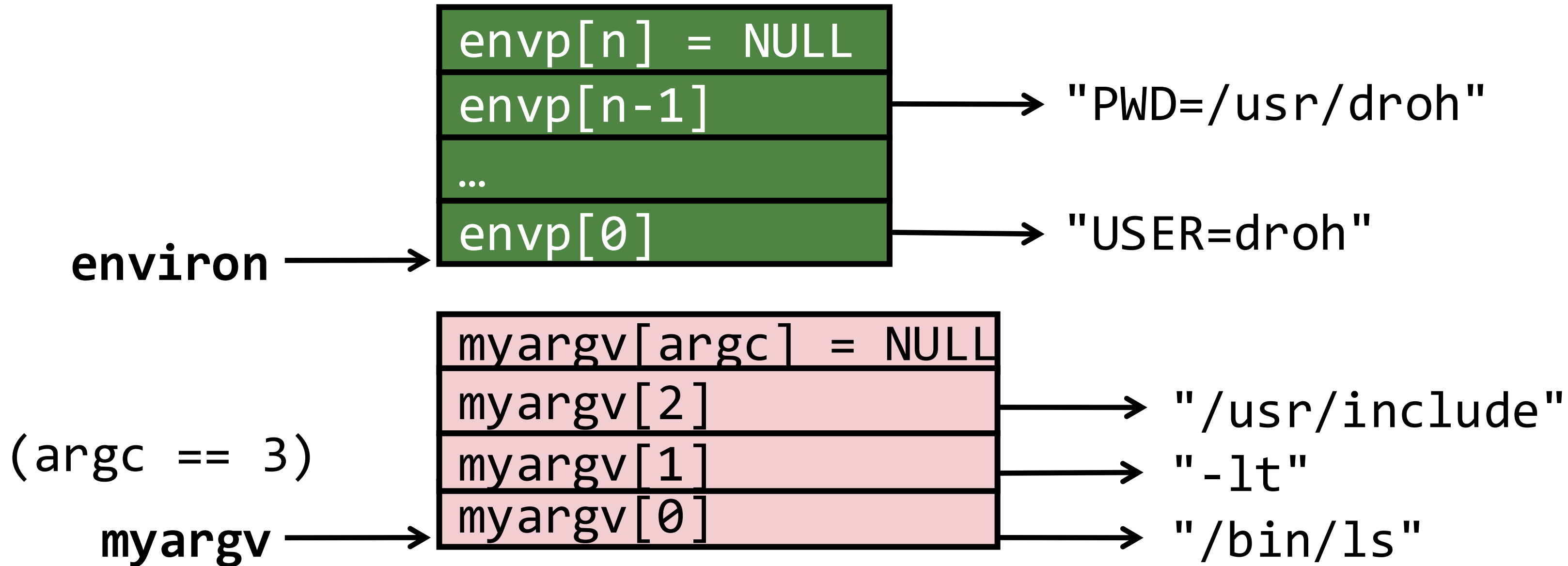
execve: Loading and Running Programs

```
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 (e.g., `/bin/ls`)
  - …with argument list **`argv`**
    - By convention **`argv[0]==filename`**
  - …and environment variable list **envp**
    - "name=value" strings (e.g., `USER=droh`)
    - `getenv`, `putenv`, `printenv`
- Overwrites code, data, and stack
  - Retains PID, open files and signal context
- Called once and never returns
  - …except if there is an error

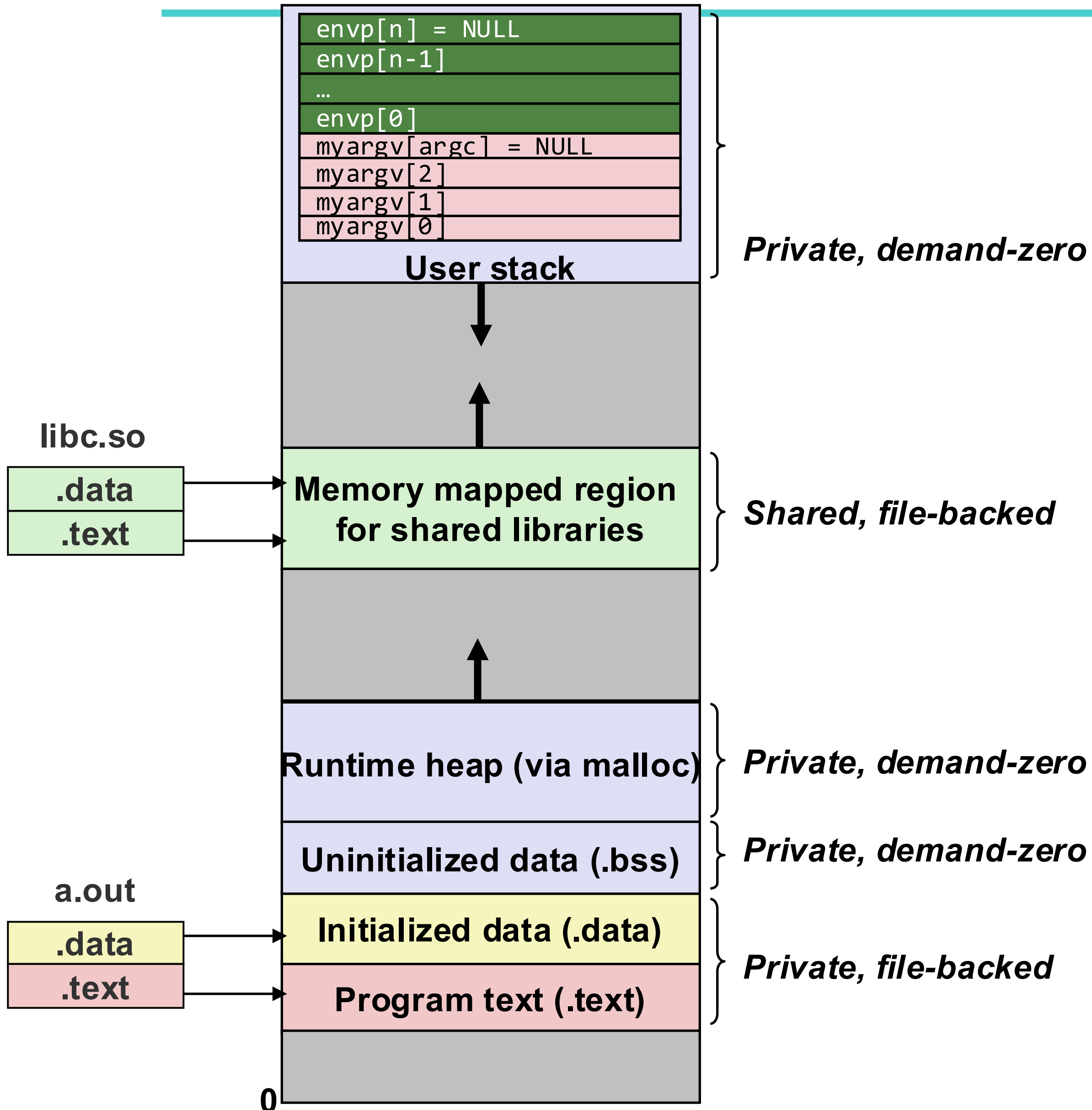
execve Example

Execute "`/bin/ls -lt /usr/include`" in child process:




```
if ((pid = fork()) == 0) {   /* Child runs program */
    if (execve(myargv[0], myargv, environ) < 0) {
        printf("%s: %s\n", myargv[0], strerror(errno));
        exit(1);
    }
}
```

The execve Function Revisited



- To load and run a new program **`/bin/ls`** in the current process using **`execve`**
- Setup memory layout
  - Initialize a page table according to the program headers
- Set PC to entry point in **`.text`**

Shells

Recall: A Shell

Shell process (parent)

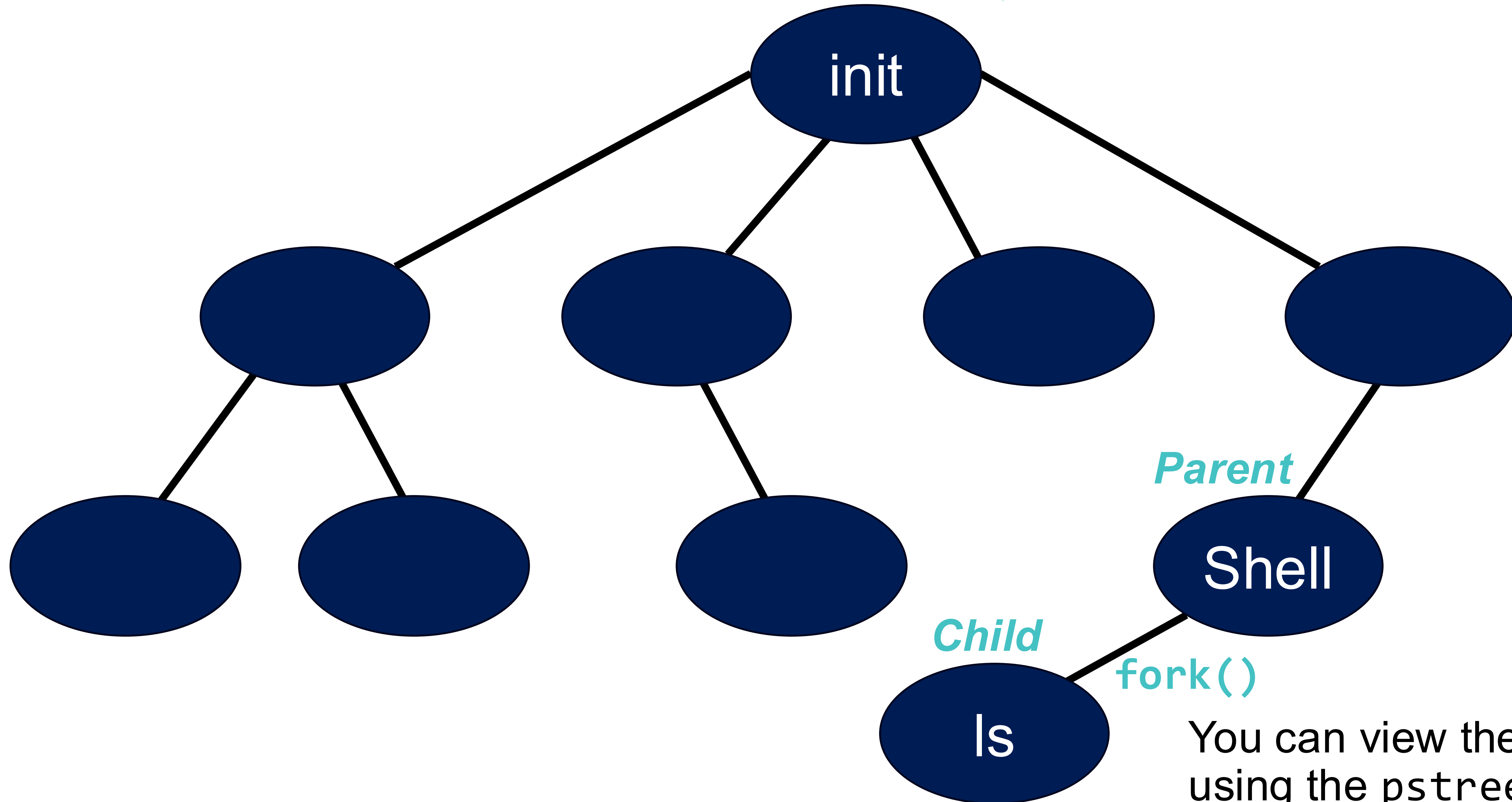
Create a child process!

`fork()` + `execve()`

`ls` process (child)

```
s20265006@3affb2a86ea8:~$ ls -al
total 1960
drwxrwxr-x 24 s20265006 s20265006    4096 Jun  2 03:00 .
drwxr-xr-x  1 root      root         4096 Mar  4 10:36 ..
-rw-------  1 s20265006 s20265006     406 May 26 04:21 .Xauthority
-rw-------  1 s20265006 s20265006   22604 Jun  2 03:00 .bash_history
-rw-r--r--  1 s20265006 s20265006     220 Mar  4 10:36 .bash_logout
-rw-r--r--  1 s20265006 s20265006    3855 Mar 28 07:37 .bashrc
drwx------  3 s20265006 s20265006    4096 Mar  4 08:28 .cache
```

Linux Process Hierarchy (Process Tree)



You can view the hierarchy using the `pstree` command

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)

shark.ics.cs.cmu.edu - PuTTY

```
zweinber@hammerheadshark ~
$ ls
15213         career   kilordle-strats  test       test.o    test.s
15410         gnu      OldFiles         test2.c    test.rs
15410-local   iclab    shark-audit.txt  test.c     test.rs~

zweinber@hammerheadshark ~
$ 
```

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)

- **Simple shell**
  - Described in the textbook, starting at Figure 8.23
  - Implementation of a very elementary shell
  - Purpose
    - Understand what happens when you type commands
    - Understand use and operation of process control operations

Simple Shell Example

```
linux> ./shellex
> /bin/ls -l csapp.c
-rw-r--r-- 1 bryant users 23053 Jun 15  2015 csapp.c
> /bin/ps
  PID TTY          TIME CMD
31542 pts/2    00:00:01 tcsh
32017 pts/2    00:00:00 shellex
32019 pts/2    00:00:00 ps
> /bin/sleep 10 &
32031 /bin/sleep 10 &
> /bin/ps
 PID TTY          TIME CMD
31542 pts/2    00:00:01 tcsh
32024 pts/2    00:00:00 emacs
32030 pts/2    00:00:00 shellex
32031 pts/2    00:00:00 sleep
32033 pts/2    00:00:00 ps
> quit
```

Must give full pathnames for programs

Run program in background

Sleep is running in background

Simple Shell Implementation

- **Basic loop**
  - Read line from command line
  - Execute the requested operation
    - Built-in command (only one implemented is **`quit`**)
    - Load and execute program from file

```
int main(int argc, char** argv)
{
    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);
```

parseline will
1) parse ‘buf’ into ‘argv’
2) return whether or not input line ended in ‘&’

shellex.c

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 */
```

Ignore empty lines

shellex.c

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 it is a ‘built in’ command, then handle it here in this program. Otherwise fork/exec the program specified in `argv[0]`

shellex.c

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 */
```

Create child

shellex.c

Simple Shell `eval` Function


```c
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 */
            execve(argv[0], argv, environ);
            // If we get here, execve failed.
            printf("%s: %s\n", argv[0], strerror(errno));
            exit(127);
        }
```

Start `argv[0]`.
Remember `execve` only returns on error.

shellex.c

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 */
            execve(argv[0], argv, environ);
            // If we get here, execve failed.
            printf("%s: %s\n", argv[0], strerror(errno));
            exit(127);
        }

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

If running child **in foreground**, wait until it is done.

shellex.c

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 */
            execve(argv[0], argv, environ);
            // If we get here, execve failed.
            printf("%s: %s\n", argv[0], strerror(errno));
            exit(127);
        }

        /* 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\n", pid, cmdline);
    }
    return;
}
```

If running child **in background**, print pid and continue doing other stuff

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 */
            execve(argv[0], argv, environ);
            // If we get here, execve failed.
            printf("%s: %s\n", argv[0], strerror(errno));
            exit(127);
        }

        /* 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\n", pid, cmdline);
    }
    return;
}
```

shellex.c

Oops. *There is a problem with this code*

Problem with Simple Shell Example

- Shell designed to run indefinitely
  - Should not accumulate unneeded resources
    - Memory
    - Child processes
    - File descriptors
- 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
- How do we solve this problem? Next class!

Summary

- **Processes**
  - At any given time, system has multiple active processes
  - Only one can execute at a time on any single core
  - Each process appears to have total control of processor + private memory space
- **Spawning processes**
  - Call fork
- **Process completion**
  - Call exit
- **Reaping and waiting for processes**
  - Call wait
- **Loading and running programs**
  - Call execve (or variant)

Question?