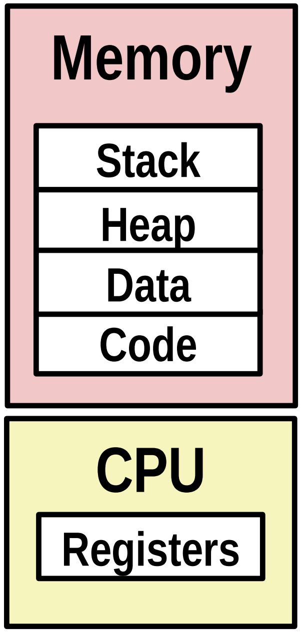
Today

- **Processes**
- Process Control

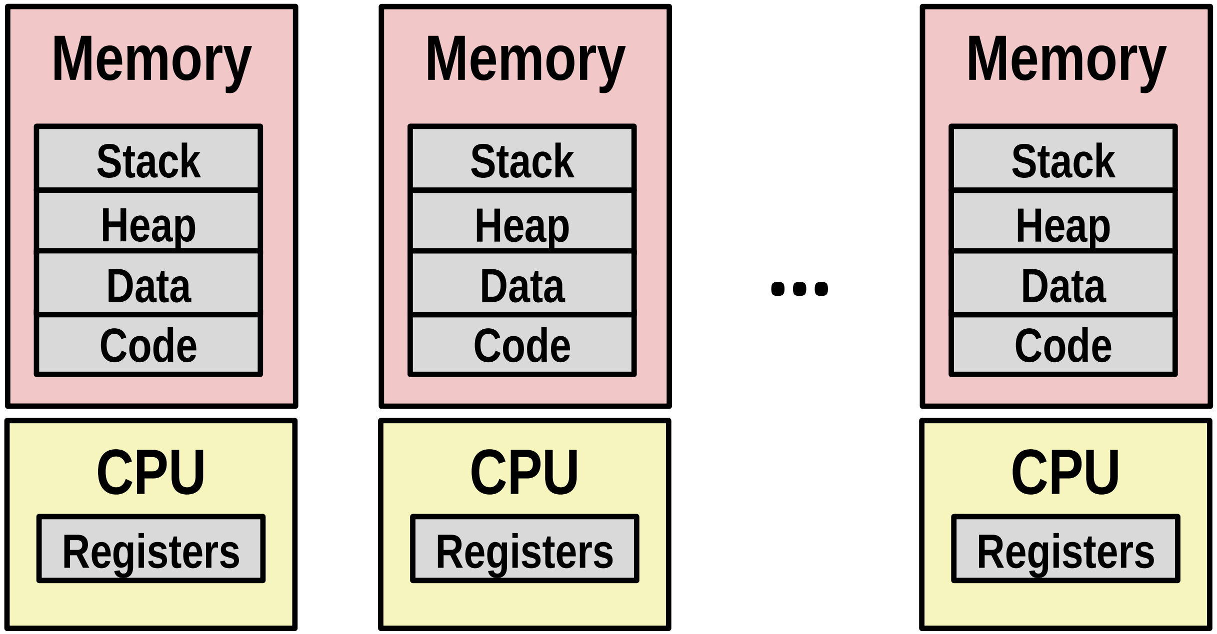
Processes

- **Definition: A *process* is an instance of a running program.**
  - One of the most profound ideas in computer science
  - Not the same as “program” or “processor” or “application”

- **Process provides each program with two key abstractions:**
  - ***Logical control flow***
    - Each program seems to have exclusive use of the CPU
    - Provided by kernel mechanism called *context switching*
  - ***Private address space***
    - Each program seems to have exclusive use of main memory.
    - Provided by kernel mechanism called *virtual memory*



Multiprocessing: The Illusion



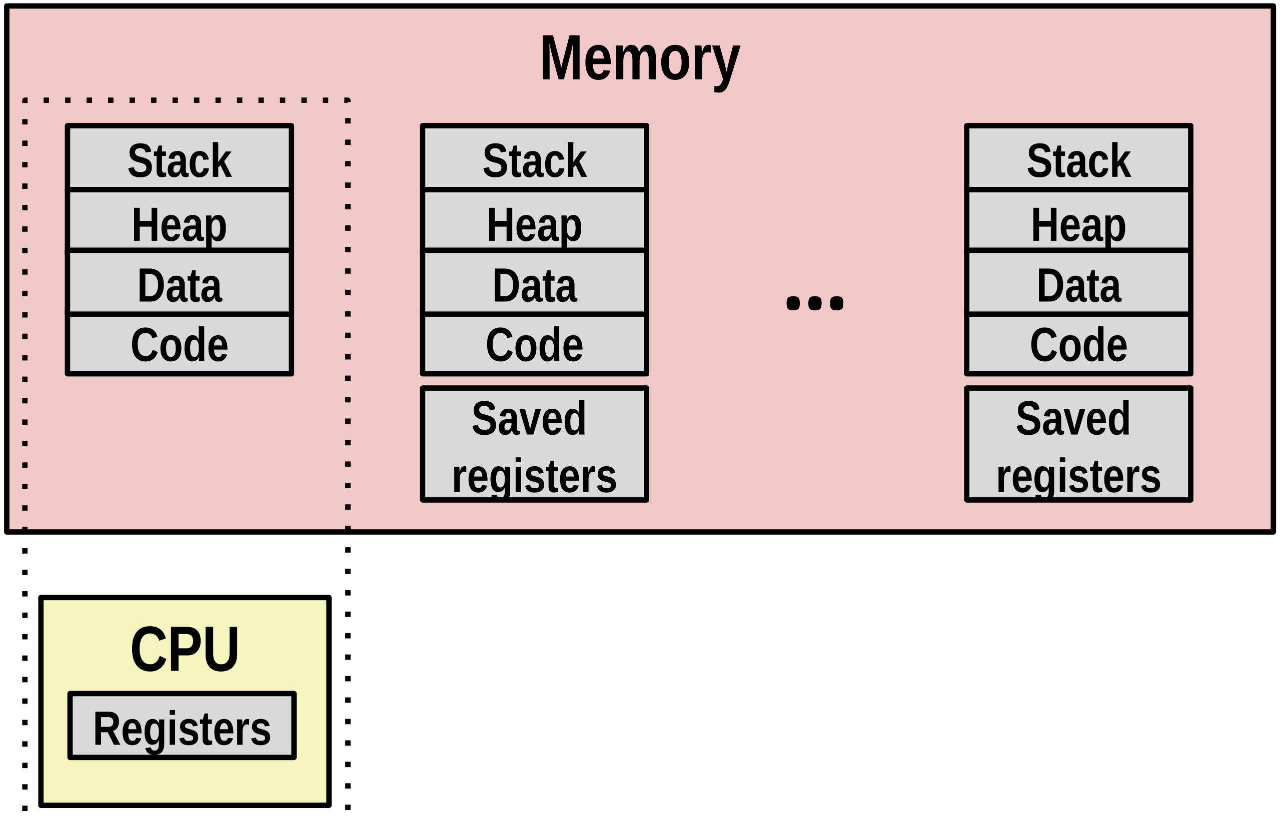
- **Computer runs many processes simultaneously**
  - Applications for one or more users
    - Web browsers, email clients, editors, …
  - Background tasks
    - Monitoring network & I/O devices

Multiprocessing Example

```
last pid: 24022;  load averages:  0.22,  0.21,  0.22       up 1+04:05:07  10:49:08
49 processes:  1 running, 48 sleeping
CPU:  0.9% user,  0.0% nice,  0.2% system,  0.0% interrupt, 98.9% idle
Mem: 350M Active, 5215M Inact, 311M Laundry, 1392M Wired, 758M Buf, 575M Free
Swap: 4096M Total, 4096M Free

  PID USERNAME    THR PRI NICE   SIZE    RES STATE    C   TIME    WCPU COMMAND
 1335 cahir         3  20    0    35M    20M select   2  22:30   1.43% python3.6
 1163 root          3  20    0    77M    45M select   2  15:44   0.85% Xorg
 1334 cahir         1  20    0    29M    15M kqread   0   9:14   0.61% i3bar
 1337 root          3  20    0    24M  7240K select   1   7:05   0.43% upowerd
 1069 messagebus    1  20    0    13M  3608K select   0   4:16   0.27% dbus-daemon
 1330 cahir         1  20    0    18M  7844K select   0   1:30   0.08% compton
24021 cahir         1  20    0    13M  3708K CPU2     2   0:00   0.04% top
 1141 root          1  20    0    11M  2204K select   1   0:16   0.03% powerd
 1267 haldaemon     2  20    0    22M  8756K select   2   0:21   0.02% hald
 2563 cahir        21  20    0   613M   360M select   3   2:46   0.01% chrome
23991 cahir         1  20    0    20M  9808K select   3   0:00   0.01% sshd
 1138 ntpd          1  20    0    19M    19M select   1   0:06   0.01% ntpd
 2565 cahir         8  20    0   312M   113M select   1   0:35   0.00% chrome
 2566 cahir         4  20    0   326M   102M uwait    1   0:16   0.00% chrome
 2853 cahir         9  20    0    11G   203M uwait    0   0:13   0.00% chrome
 2843 cahir         9  20    0   462M   136M uwait    2   0:09   0.00% chrome
10394 cahir         1  52    0    19M  8872K ttyin    2   0:02   0.00% zsh
```

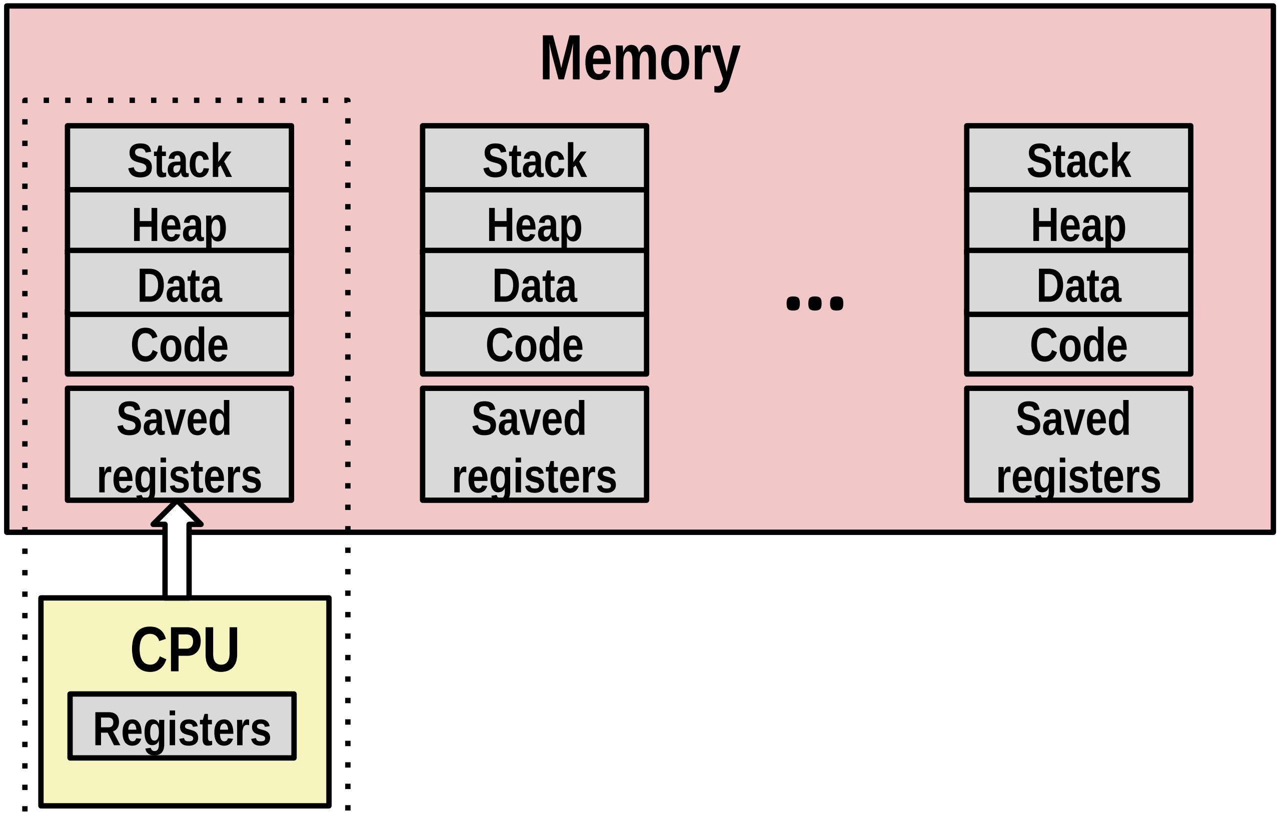
- **Running program "top" on FreeBSD**
  - System has 49 processes, only one is active
  - Identified by Process ID (PID)

Multiprocessing: The (Traditional) Reality



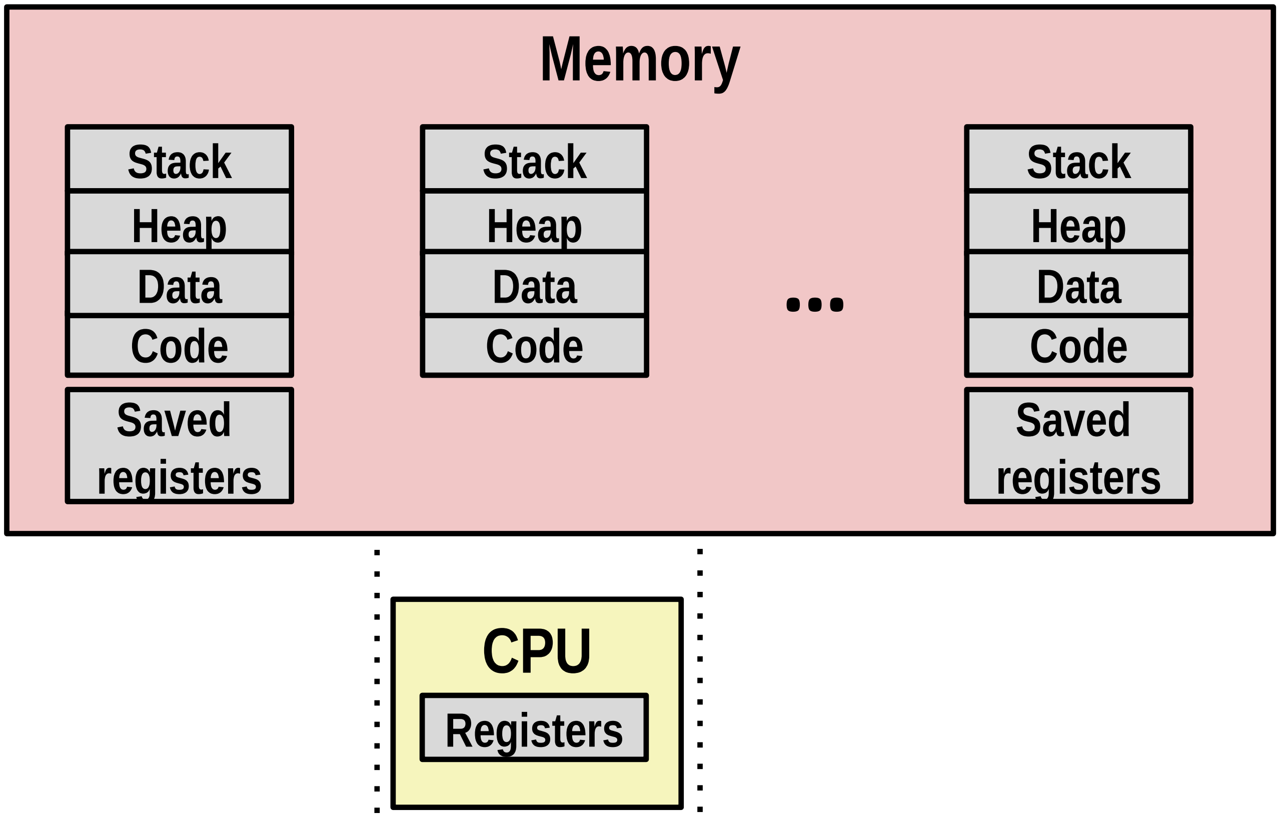
- **Single processor executes multiple processes concurrently**
  - Process executions interleaved (multitasking)
  - Address spaces managed by virtual memory system (later in course)
  - Register values for inactive (nonexecuting) processes saved in memory

Multiprocessing: The (Traditional) Reality



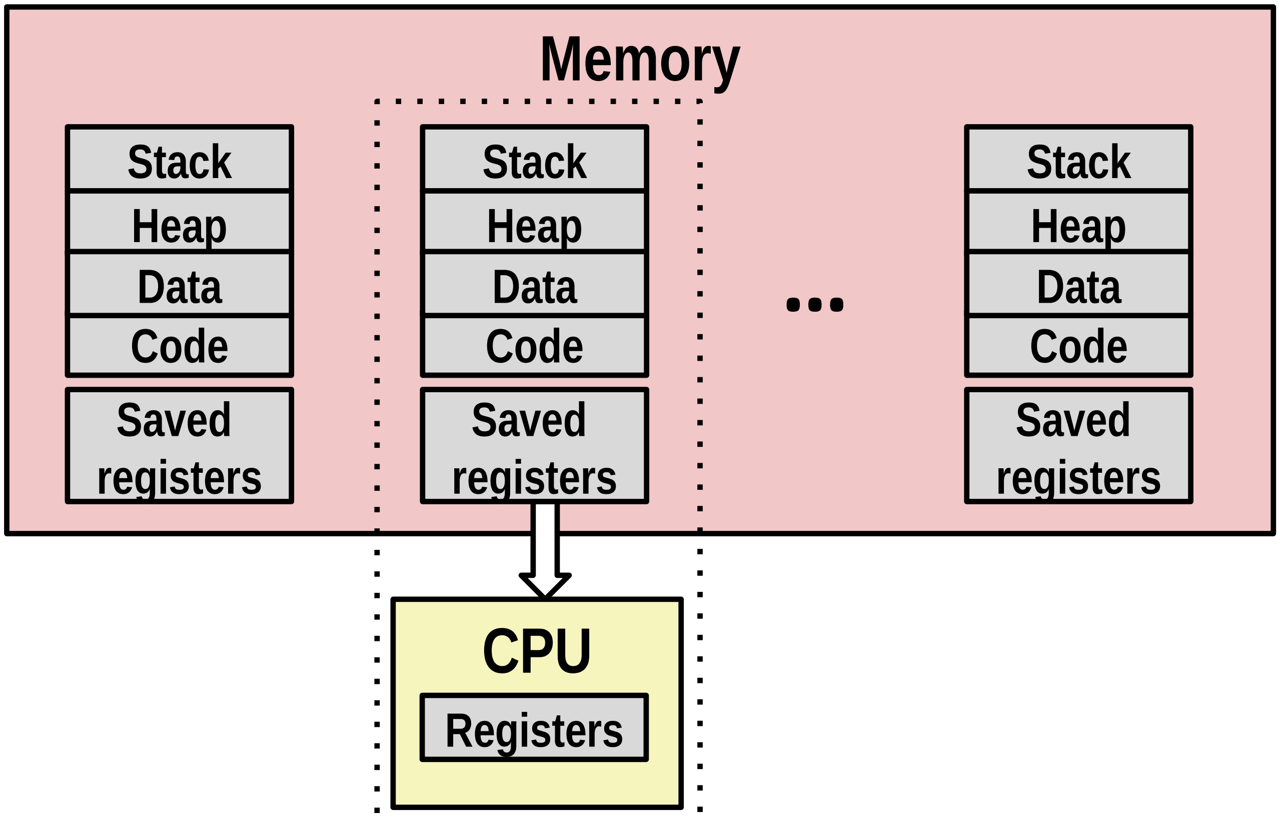
- Save current registers in memory

Multiprocessing: The (Traditional) Reality



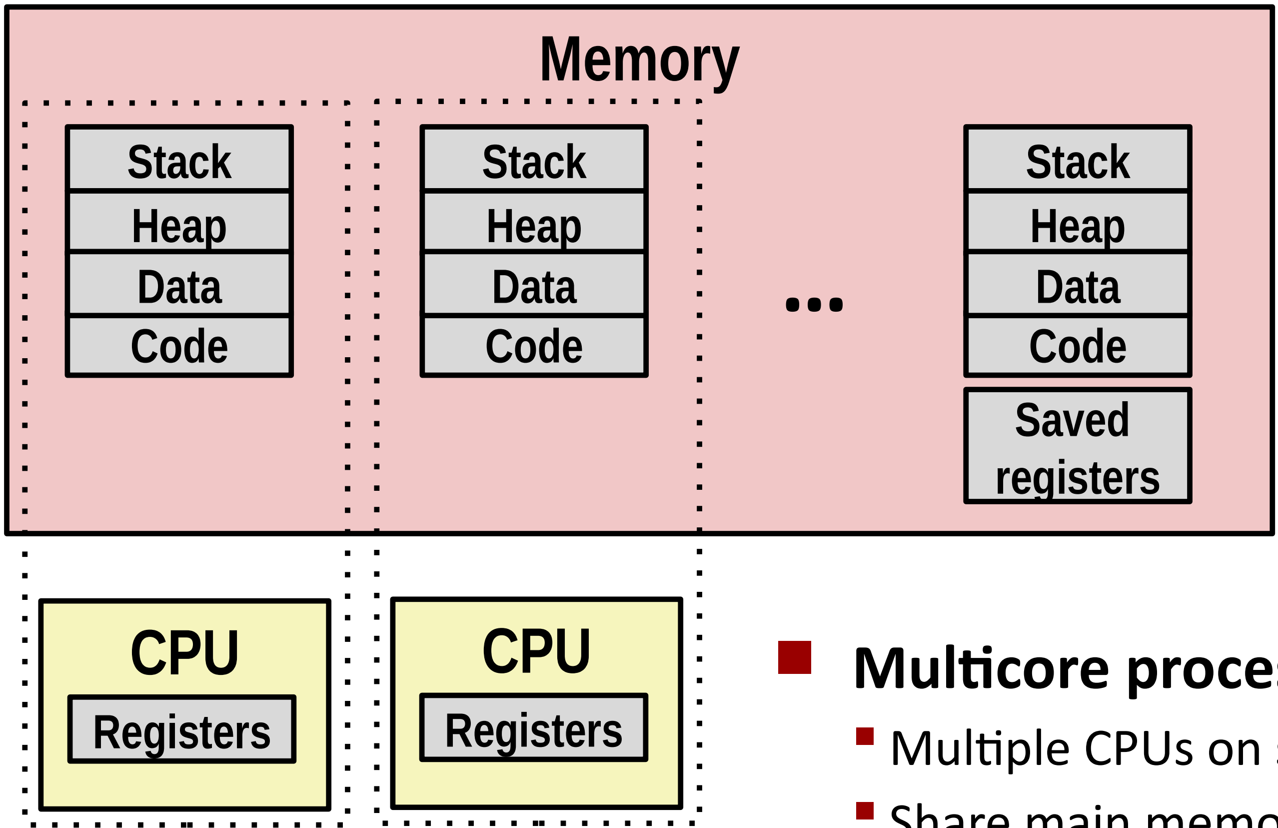
- Schedule next process for execution

Multiprocessing: The (Traditional) Reality



- Load saved registers and switch address space (context switch)

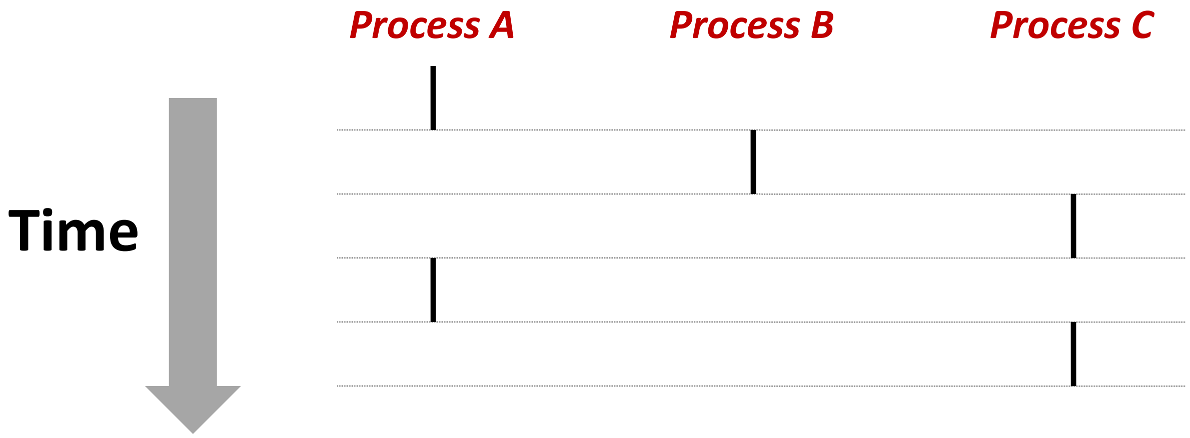
Multiprocessing: The (Modern) Reality



- **Multicore processors**
  - Multiple CPUs on single chip
  - Share main memory (and some caches)
  - Each can execute a separate process
    - Scheduling of processors onto cores done by kernel

Concurrent Processes

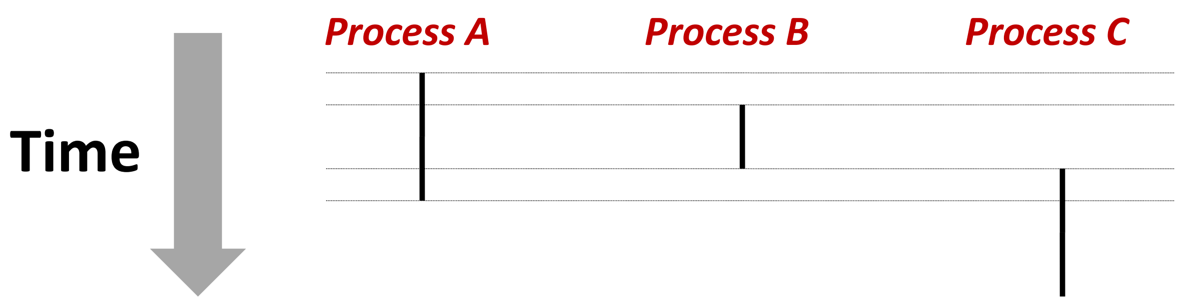
- **Each process is a logical control flow.**
- **Two processes *run concurrently* (*are concurrent)* if their flows overlap in time**
- **Otherwise, they are *sequential***
- **Examples (running on single core):**
  - Concurrent: A & B, A & C
  - Sequential: B & C



User View of Concurrent Processes

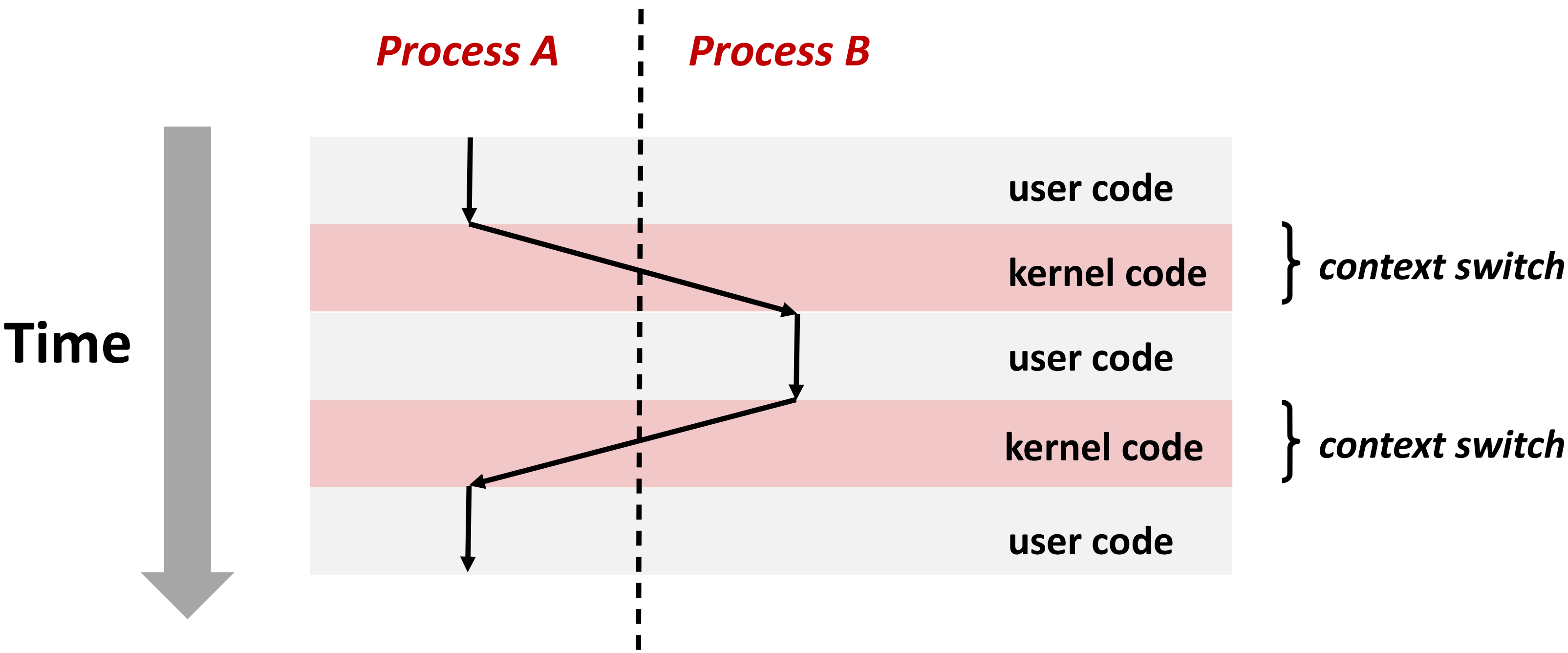
- **Control flows for concurrent processes are physically disjoint in time**

- **However, we can think of concurrent processes as running in parallel with each other**



Context Switching

- **Processes are managed by a shared chunk of memory-resident OS code called the *kernel***
  - Important: the kernel is not a separate process, but rather runs as part of some existing process.
- **Control flow passes from one process to another via a *context switch***



Today

- Exceptional Control Flow
- Exceptions
- Processes
- **Process Control**

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(-1);
}
```

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(-1);
}
```

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

- But, must think about application. Not always appropriate to exit when something goes wrong.

Error-handling Wrappers

- We simplify the code we present to you even 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();
```

- NOT what you generally want to do in a real application

Obtaining Process IDs

- `pid_t getpid(void)`
  - Returns PID of current process

- `pid_t getppid(void)`
  - Returns PID of parent process

Creating and Terminating Processes

From a programmer's perspective, we can think of a process as being in one of three states

- **Running**
  - Process is either executing, or waiting to be executed and will eventually be *scheduled* (i.e., chosen to execute) by the kernel
- **Stopped**
  - Process execution is *suspended* and will not be scheduled until further notice (next lecture when we study signals)
- **Terminated**
  - Process is stopped permanently

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
  - Another way to explicitly set the exit status is to return an integer value from the main routine

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

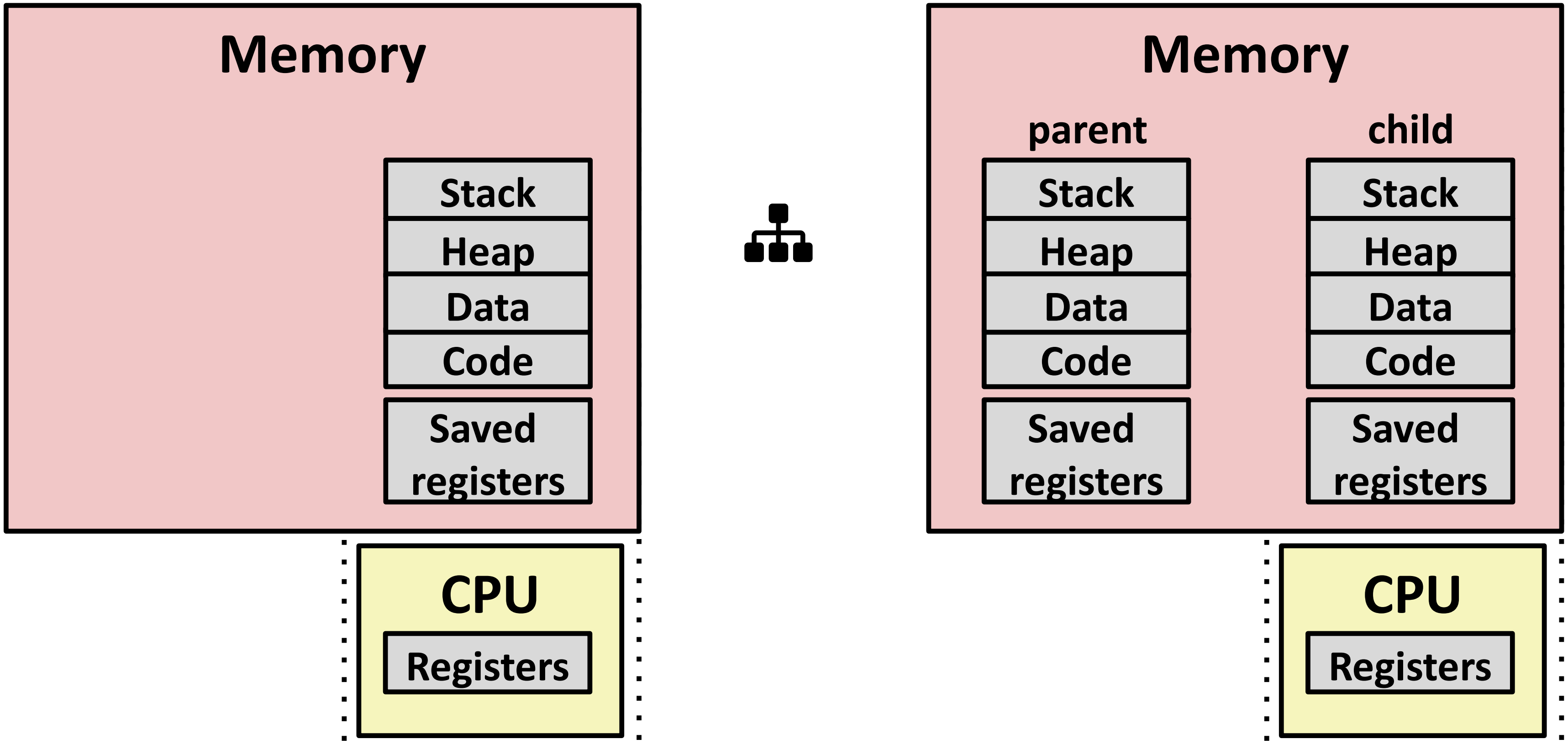
Creating Processes

- ***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:
    - Child get an identical (but separate) 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

`fork` Example

```
int main(int argc, char** argv) {
  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;
}                                fork.c
```

- **Call once, return twice**
- **Concurrent execution**
  - **Can't predict execution order of parent and child**

```
linux> ./fork
parent: x=0
child : x=2
```

```
linux> ./fork
child : x=2
parent: x=0
```

```
linux> ./fork
parent: x=0
child : x=2
```

```
linux> ./fork
parent: x=0
child : x=2
```

Making `fork` More Nondeterministic

- **Problem**
  - Linux scheduler does not create much run-to-run variance
  - Hides potential race conditions in nondeterministic programs
    - E.g., does `fork` return to child first, or to parent?
- **Solution**
  - Create custom version of library routine that inserts random delays along different branches
    - E.g., for parent and child in `fork`
  - Use runtime interpositioning to have program use special version of library code

Variable delay `fork`

```c
/* fork wrapper function */
pid_t fork(void) {
    initialize();
    int parent_delay = choose_delay();
    int child_delay = choose_delay();
    pid_t parent_pid = getpid();
    pid_t child_pid_or_zero = real_fork();
    if (child_pid_or_zero > 0) {
        /* Parent */
        if (verbose) {
            printf("Fork.  Child pid=%d, delay = %dms."
                   "Parent pid=%d, delay = %dms\n",
                   child_pid_or_zero, child_delay,
                   parent_pid, parent_delay);
            fflush(stdout);
        }
        ms_sleep(parent_delay);
    } else {
        /* Child */
        ms_sleep(child_delay);
    }
    return child_pid_or_zero;
}
```

myfork.c

`forkx2` Example

```
int main(int argc, char** argv) {
  pid_t pid;
  int x = 1;

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

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

```
linux> ./fork2
parent: x=0
parent: x=-1
child : x=2
child : x=3
```

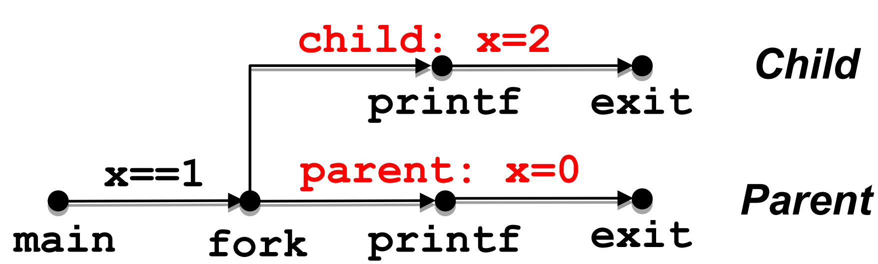
- Call once, return twice
- Concurrent execution
  - Can't predict execution order of parent and child
- Duplicate but separate address space
  - `x` has a value of 1 when fork returns in parent and child
  - Subsequent changes to `x` are independent
- Shared open files
  - `stdout` is the same in both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

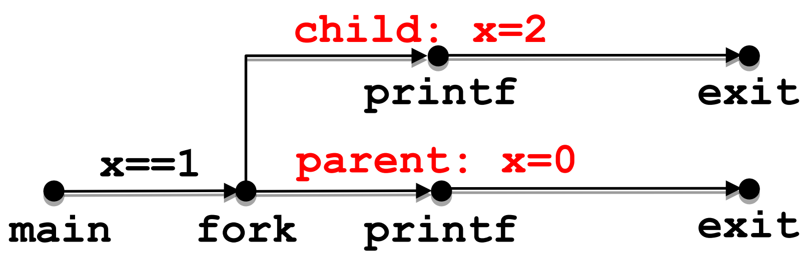
Process Graph Example

```
int main(int argc, char** argv) {
  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;
}                                   fork.c
```

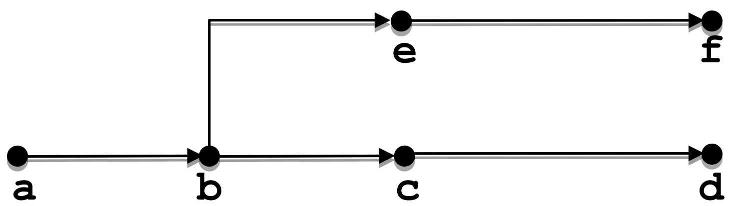


Interpreting Process Graphs

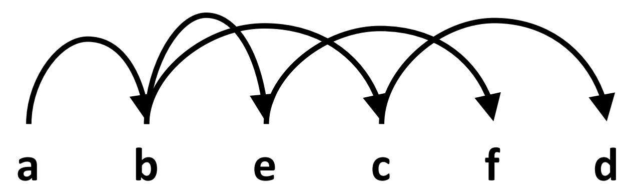
- **Original graph:**



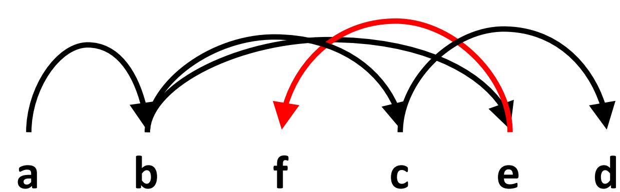
- **Relabeled graph:**



Feasible total ordering:



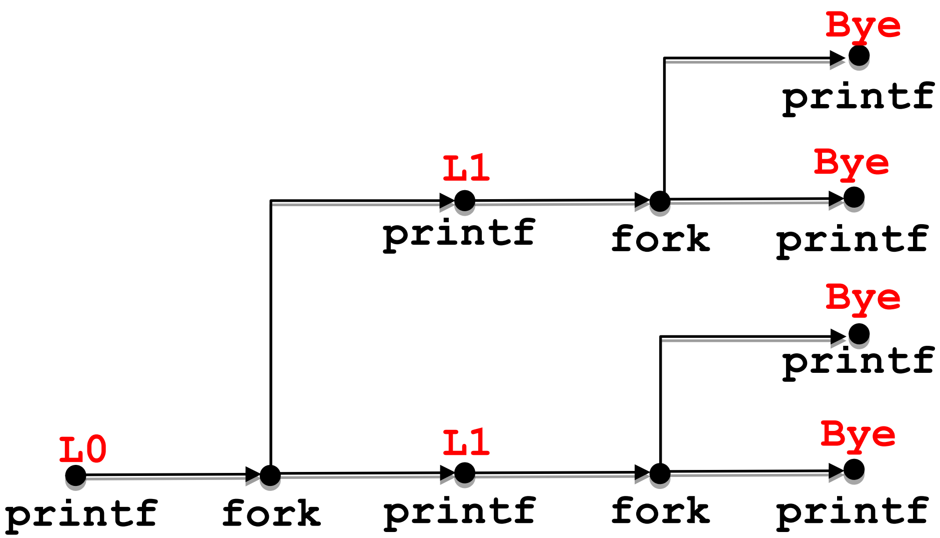
Infeasible total ordering:



fork Example: Two consecutive forks

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

forks.c



Feasible output:

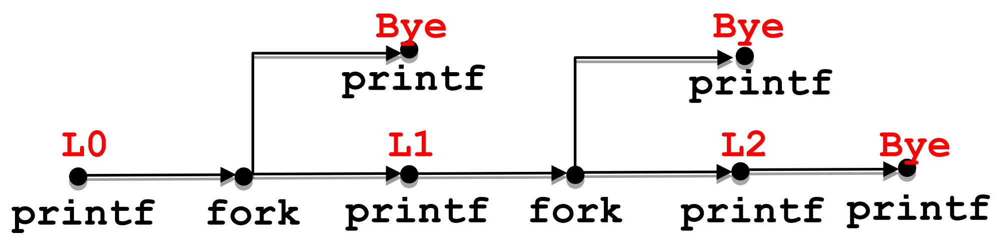
L0
L1
Bye
Bye
L1
Bye
Bye

Infeasible output:

L0
Bye
L1
Bye
L1
Bye
Bye

fork Example: Nested forks in parent

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



Feasible output:

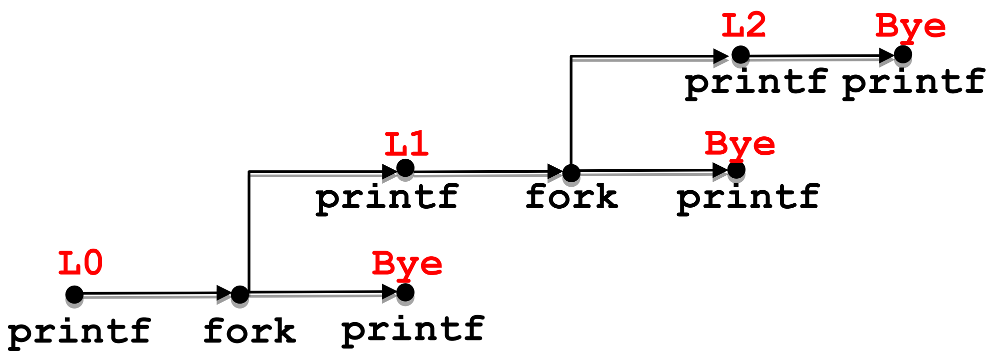
L0
L1
Bye
Bye
L2
Bye

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:	Infeasible output:
L0	L0
Bye	Bye
L1	L1
L2	Bye
Bye	Bye
Bye	L2

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`)
  - 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)
      continue; /* 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**

Non-terminating Child Example

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

forks.c

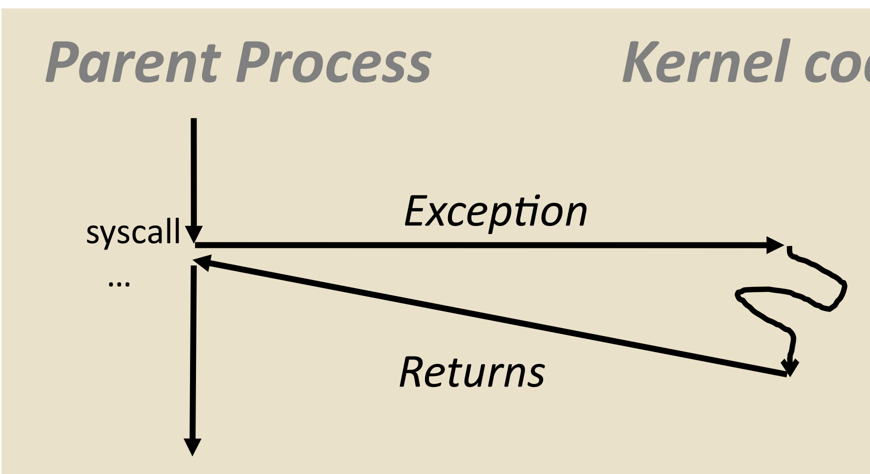
```
linux> ./forks 8
Terminating Parent, PID = 6675
Running Child, PID = 6676
linux> ps
  PID TTY          TIME CMD
 6585 ttyp9    00:00:00 tcsh
 6676 ttyp9    00:00:06 forks
 6677 ttyp9    00:00:00 ps
linux> kill 6676
linux> ps
  PID TTY          TIME CMD
 6585 ttyp9    00:00:00 tcsh
 6678 ttyp9   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



And, potentially other user processes, including a child of parent

`wait`: Synchronizing with Children

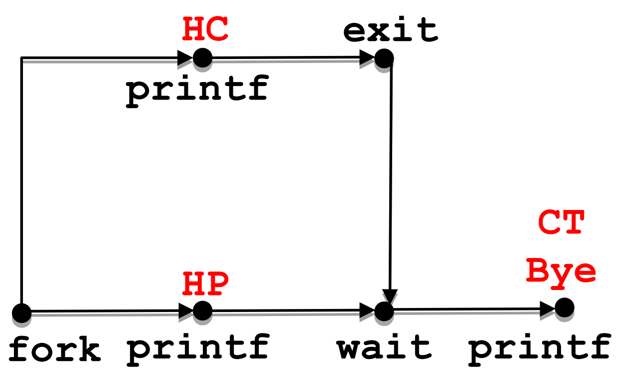
- **Parent reaps a child by calling the `wait` function**

- **`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

```c
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



Feasible output(s):

HC	HP
HP	HC
CT	CT
Bye	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);
  }
}
```

forks.c

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 (see textbook)

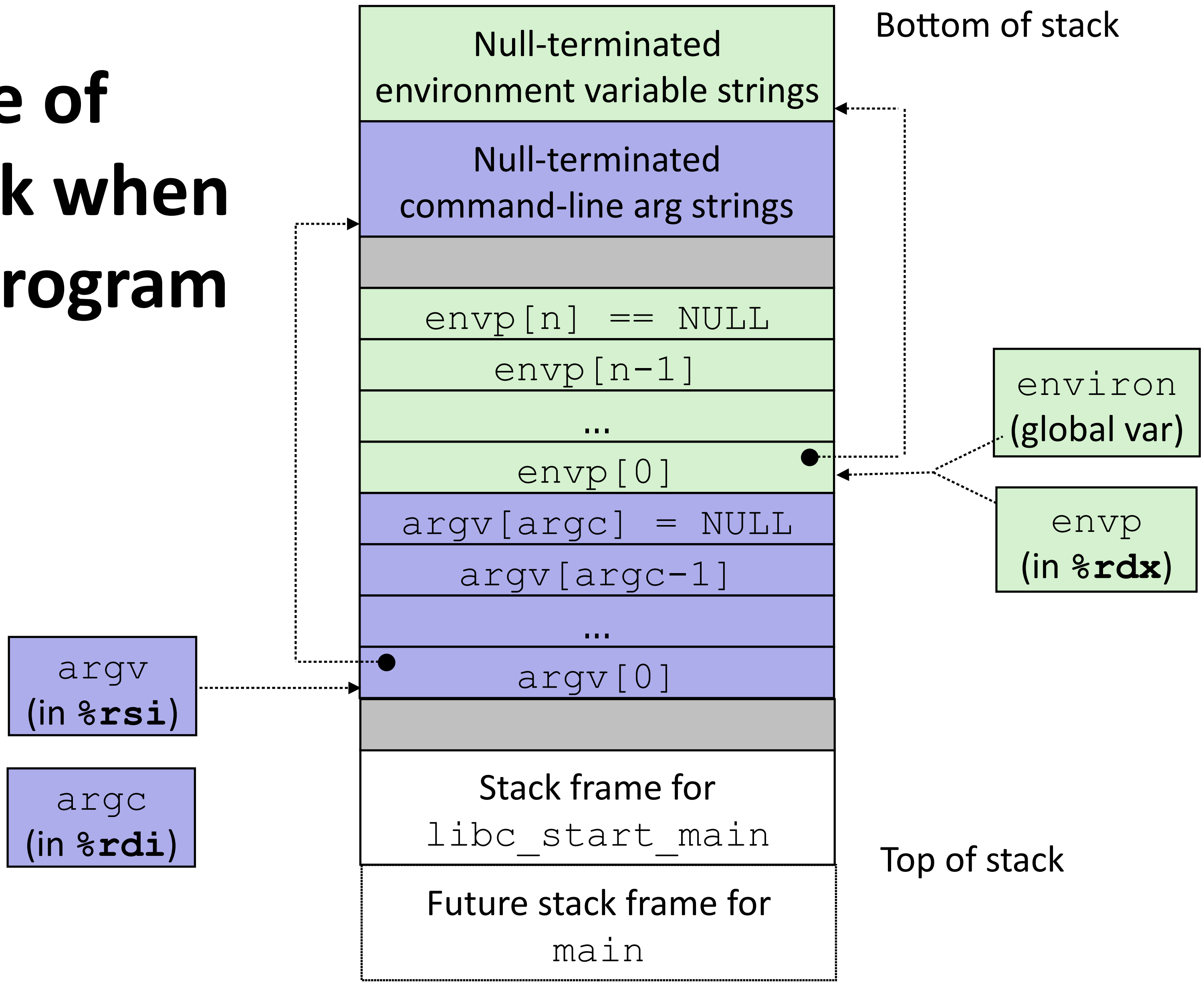
```
void fork11() {
  pid_t pid[N];
  int i;
  int child_status;

  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);
  }
}
```

forks.c

`execve`: Loading and Running Programs

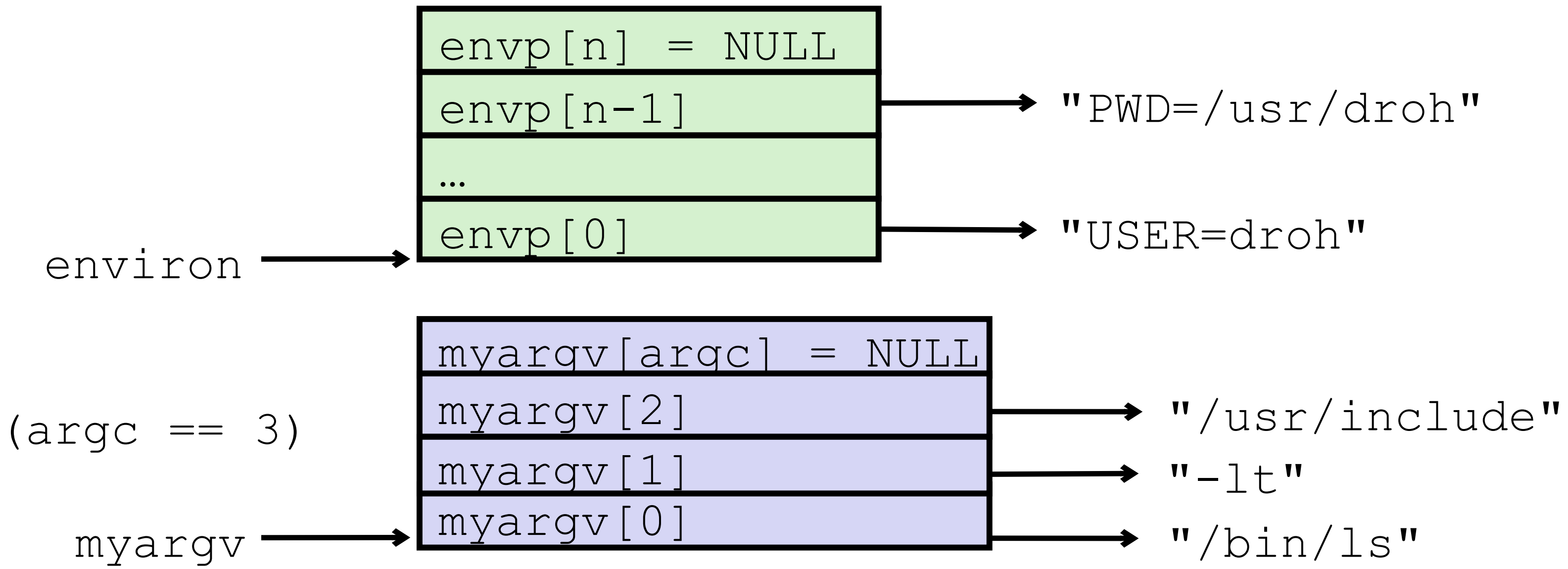
- `int execve(char *filename, char *argv[], char *envp[])`
- **Loads and runs in the current process:**
  - Executable file **`filename`**
    - Can be object file or script file beginning with `#!interpreter` (e.g., `#!/bin/bash`)
  - …with argument list **`argv`**
    - By convention **`argv[0]==filename`**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

Structure of the stack when a new program starts



execve Example

- **Execute "/bin/ls -lt /usr/include" in child process using current environment:**




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

Summary

- **Exceptions**
  - Events that require nonstandard control flow
  - Generated externally (interrupts) or internally (traps and faults)

- **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

Summary (cont.)

- **Spawning processes**
  - Call `fork`
  - One call, two returns
- **Process completion**
  - Call `exit`
  - One call, no return
- **Reaping and waiting for processes**
  - Call `wait` or `waitpid`
- **Loading and running programs**
  - Call `execve` (or variant)
  - One call, (normally) no return