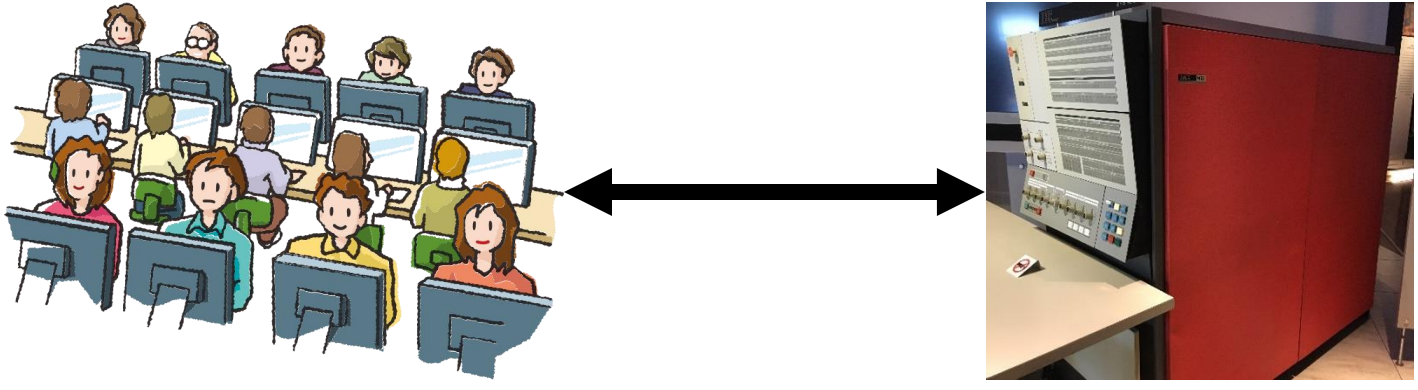
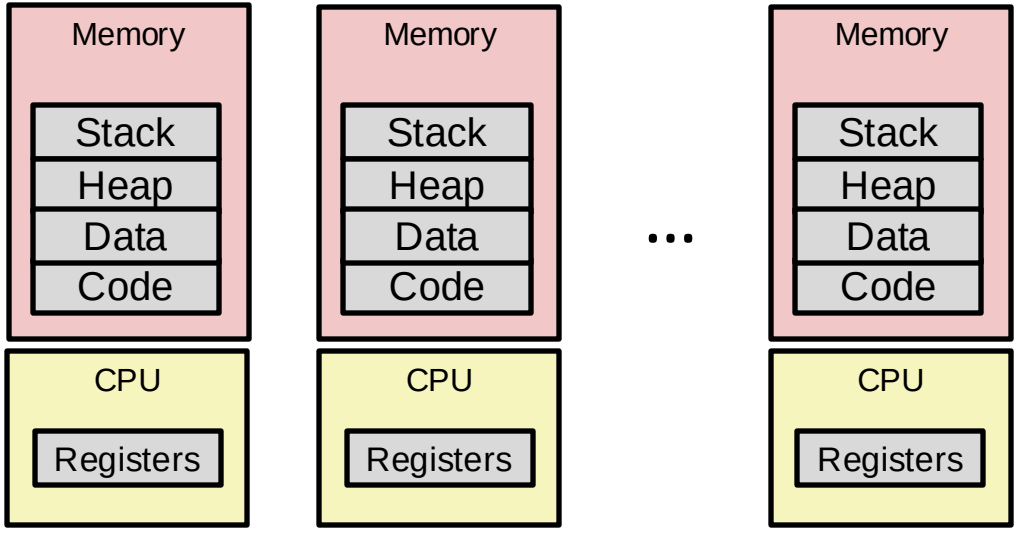
Processes

How can many people share one computer efficiently?

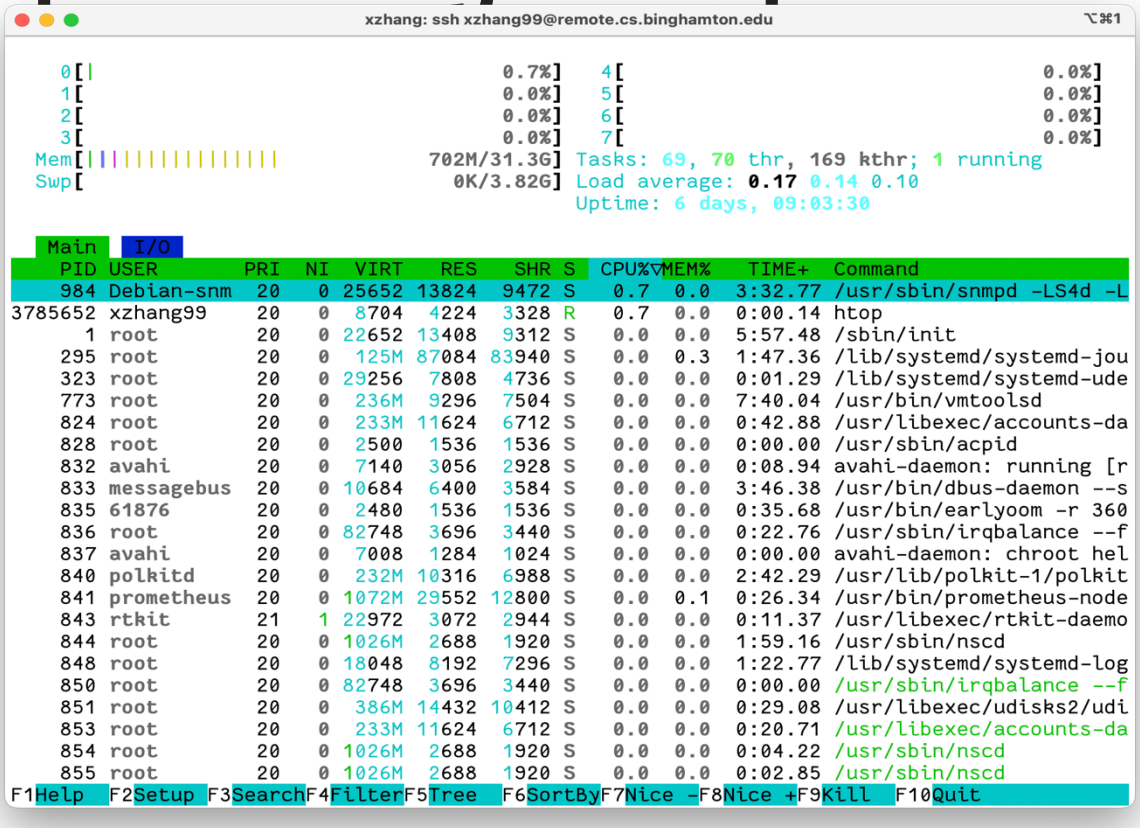


Multiprocessing



- **Sharing through hardware abstraction**
  - Software(OS) managed hardware(CPU, memory, devices)
  - Providing **consistent abstraction** view to every user program
- Computer runs many processes simultaneously
  - Applications for one or more users
    - Web browsers, editors, …
  - Background tasks
    - Monitoring network & I/O devices

Multiprocessing Example

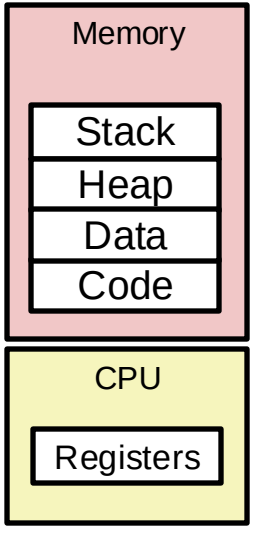


- Running program "`htop`" on department machine
  - System has 69 "tasks", 1 of which is running
  - Identified by Process ID (PID), user account, command name

Processes

- Definition: A *process* is an instance of a running program.
  - Not the same as “program” or “processor”

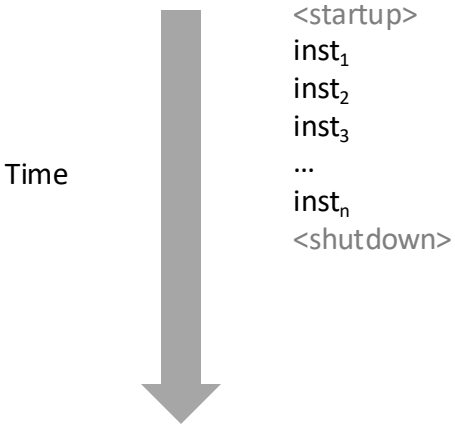
- Process provides each program with two key abstractions:
  - ***Private address space***
    - Each program seems to have exclusive use of main memory.
    - Provided by kernel mechanism called *virtual memory*
  - ***Logical control flow***
    - Each program seems to have exclusive use of the CPU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



Control Flow

- Processors do only one thing:
  - From startup to shutdown, each CPU core simply reads and executes a sequence of machine instructions, one at a time *
  - This sequence is the CPU's *control flow* (or *flow of control*)

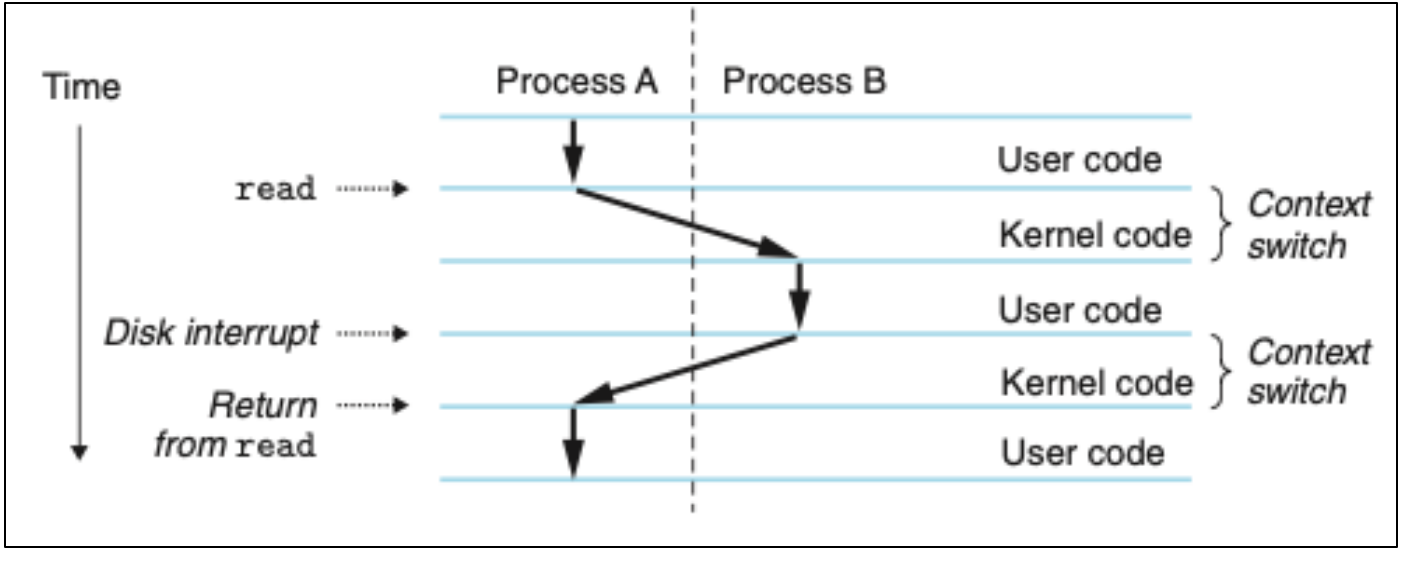
Physical control flow



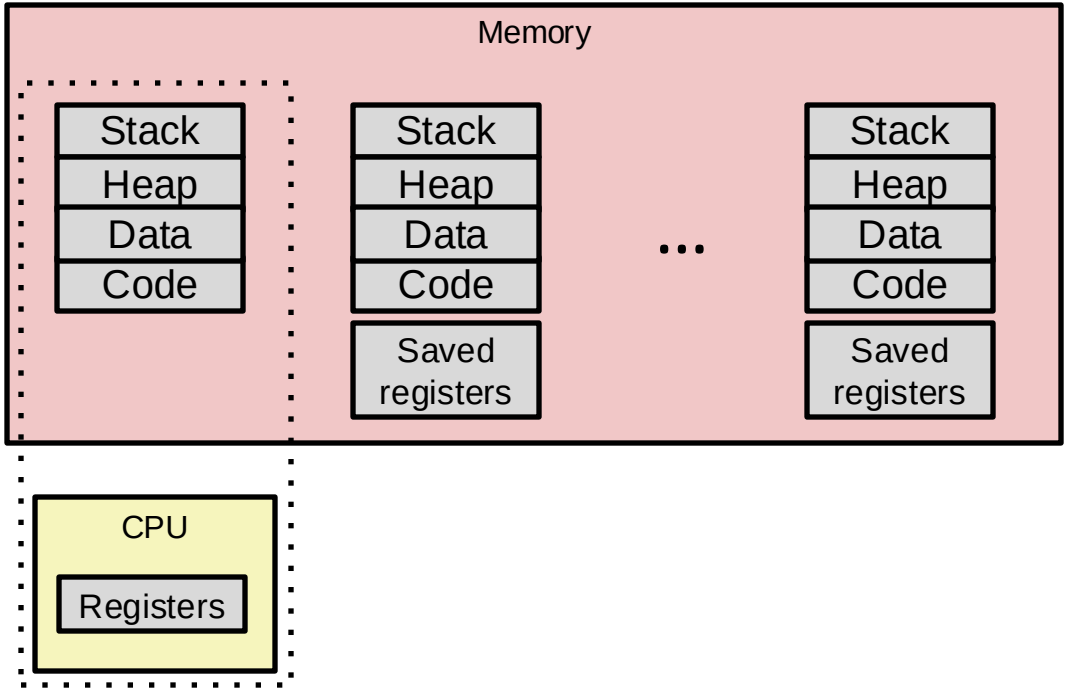
* many modern CPUs execute several instructions at once and/or out of program order, but this is invisible to the programmer

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*

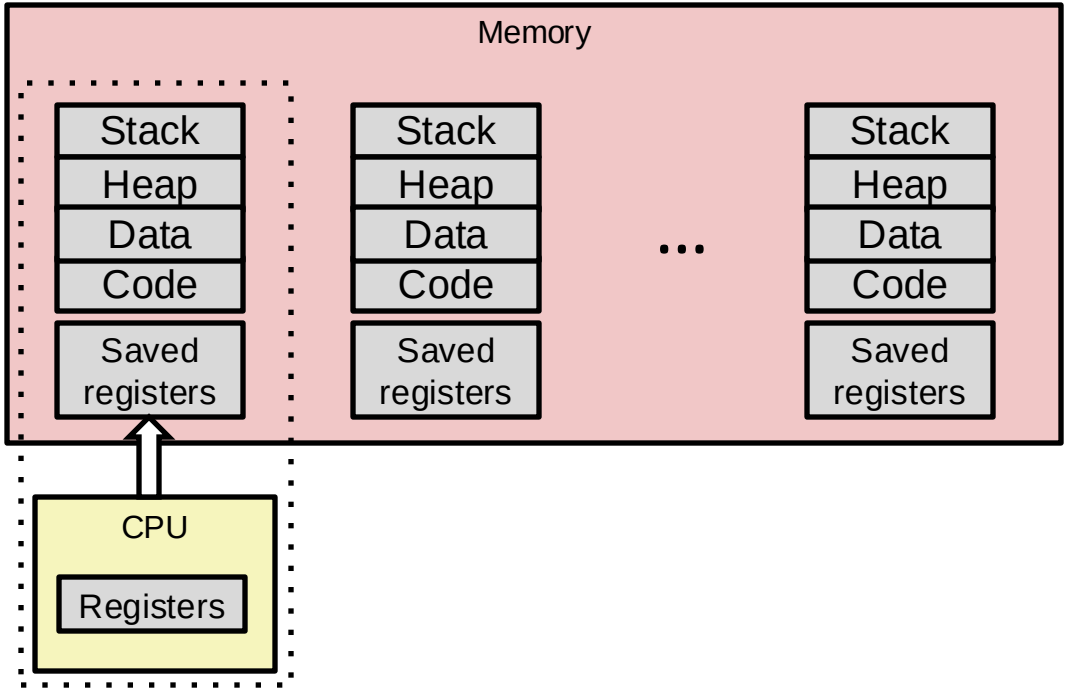


Context Switching (Uniprocessor)



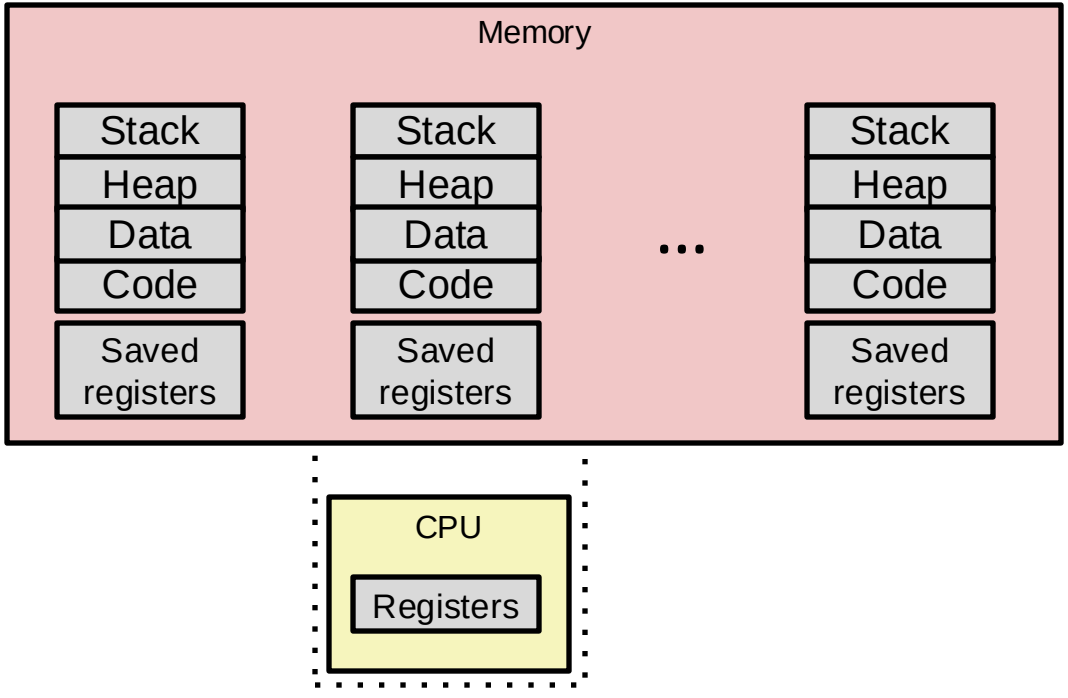
- Single processor executes multiple processes concurrently
  - Process executions **interleaved** (multitasking)
  - Address spaces managed by virtual memory system
  - Register values for nonexecuting processes saved in memory

Context Switching (Uniprocessor)



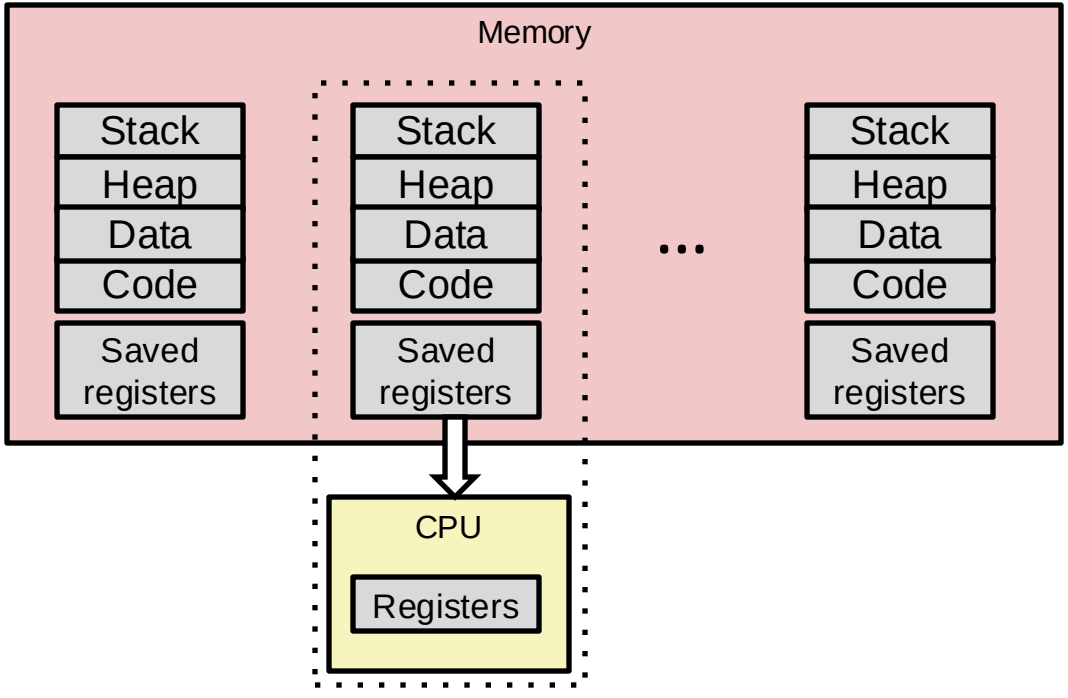
- Save current registers in memory

Context Switching (Uniprocessor)



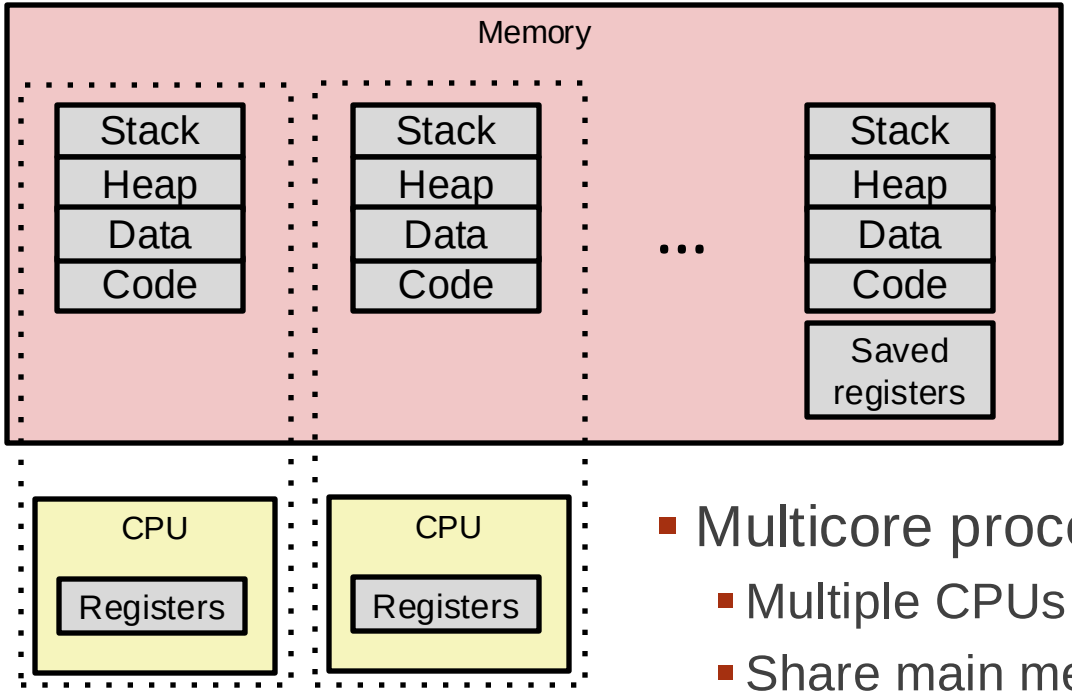
- Schedule next process for execution

Context Switching (Uniprocessor)



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

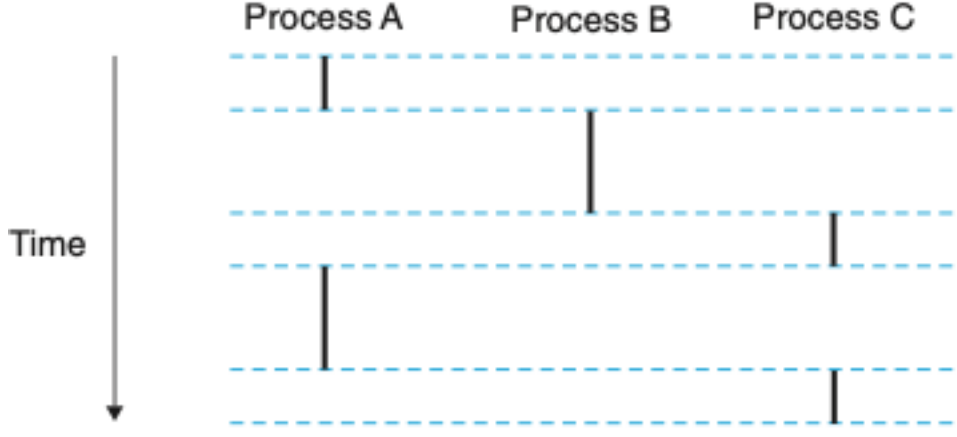
Context Switching (Multicore)



- 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

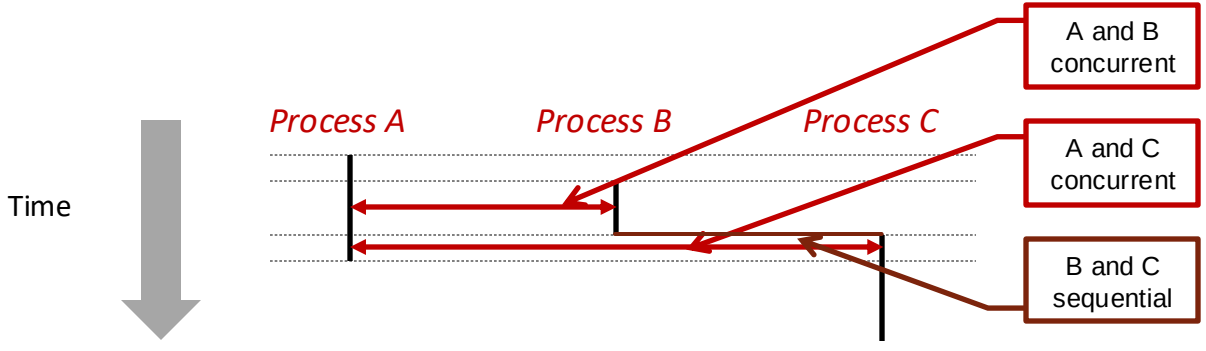
Uniprocessor Logical control flows

- Only one process runs at a time
- A and B execution is *interleaved*, not truly concurrent
- Similarly for A and C
- Still possible for A and B / A and C to interfere with each other



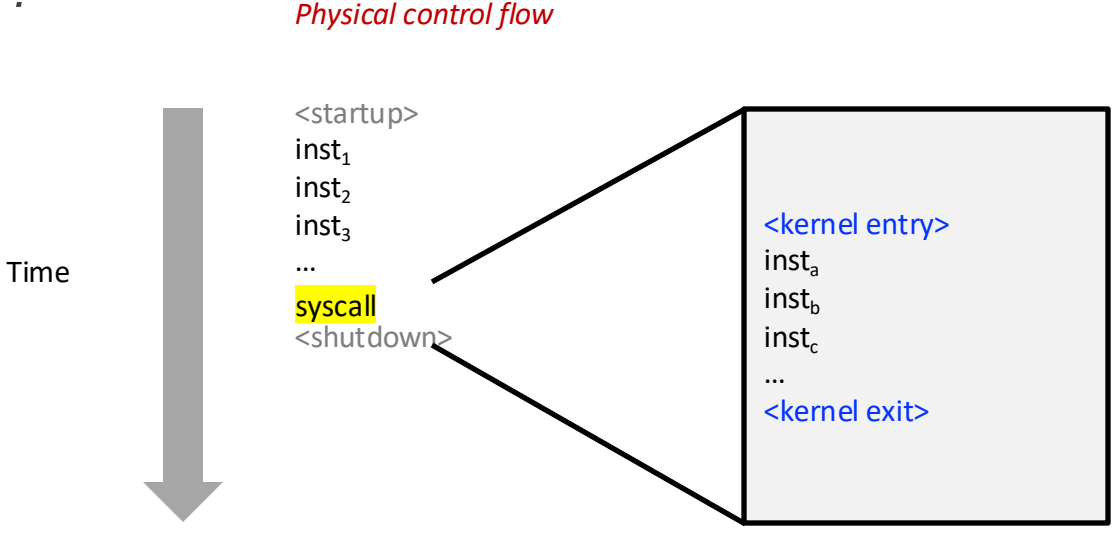
Concurrent flows

- Two processes *run concurrently* (*are concurrent*) if their execution overlaps in time
- Otherwise, they are *sequential*
- Appears as if concurrent processes run in parallel with each other
  - This means they can interfere with each other



How does the kernel take control?

- The CPU executes instructions in sequence
- We don't write "now run kernel code" in our programs…
  - Timer interrupts: generate by hardware timer
  - *Or do we??*



System Calls

- Whenever a program wants to cause an effect outside its own process, it must ask the kernel for help

- Examples:
  - Read/write files
  - Get current time
  - Allocate RAM (sbrk)
  - Create new processes

```c
// fopen.c
FILE *fopen(const char *fname, const char *mode)
{
  int flags = mode2flags(mode);
  if (!flags) return NULL;
  int fd = open(fname, flags, DEFPERMS);
  if (fd == -1) return NULL;
  return fdopen(fd, mode);
}

// open.S
    .global open
open:
    mov $SYS_open, %eax
    syscall
    cmp $SYS_error_thresh, %rax
    ja  __syscall_error
    ret
```

System Call Error Handling

- Almost all system-level operations can fail
  - Only exception is the handful of functions that return `void`
  - You must explicitly check for failure

- On error, most system-level functions return −1 and set global variable `errno` to indicate cause.
- Example:

```
1 pid_t pid = fork();
2 if (pid == -1) {
3   fprintf(stderr, "fork error: %s\n", strerror(errno));
4   exit(1);
5 }
```

Obtaining Process IDs

- Every process has a process ID, a code number that identifies it
- PID does not change during the process lifetime
- Every process has a parent process
  - The process that started it
  - Except the init process(pid 1)

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

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

Process States

At any time, each process is either:

- Running
  - Process is either executing instructions, or it *could be* executing instructions if there were enough CPU cores.
- Blocked / Sleeping
  - Process cannot execute any more instructions until some external event happens (usually I/O).
- Stopped
  - Process has been prevented from executing by user action (control-Z).
- Terminated / Zombie
  - Process is finished. Parent process has not yet been notified.

Terminating Processes

- Process becomes terminated for one of three reasons:
  - Receiving a signal whose default action is to terminate
  - 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
    - The way to distinguish the parent process and the child 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

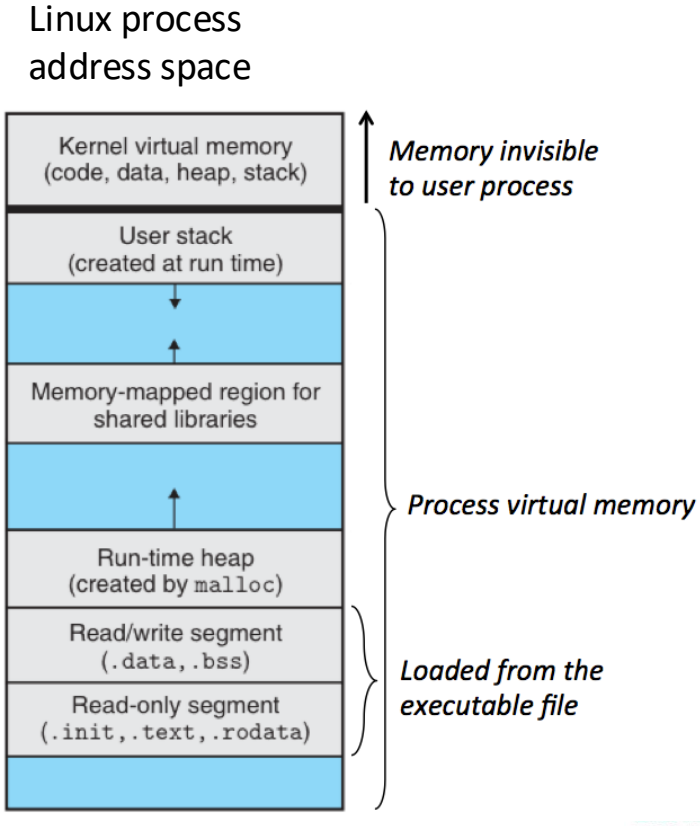
- `fork` is interesting (and often confusing) because it is called *once* but returns *twice*

Differences between parent and child

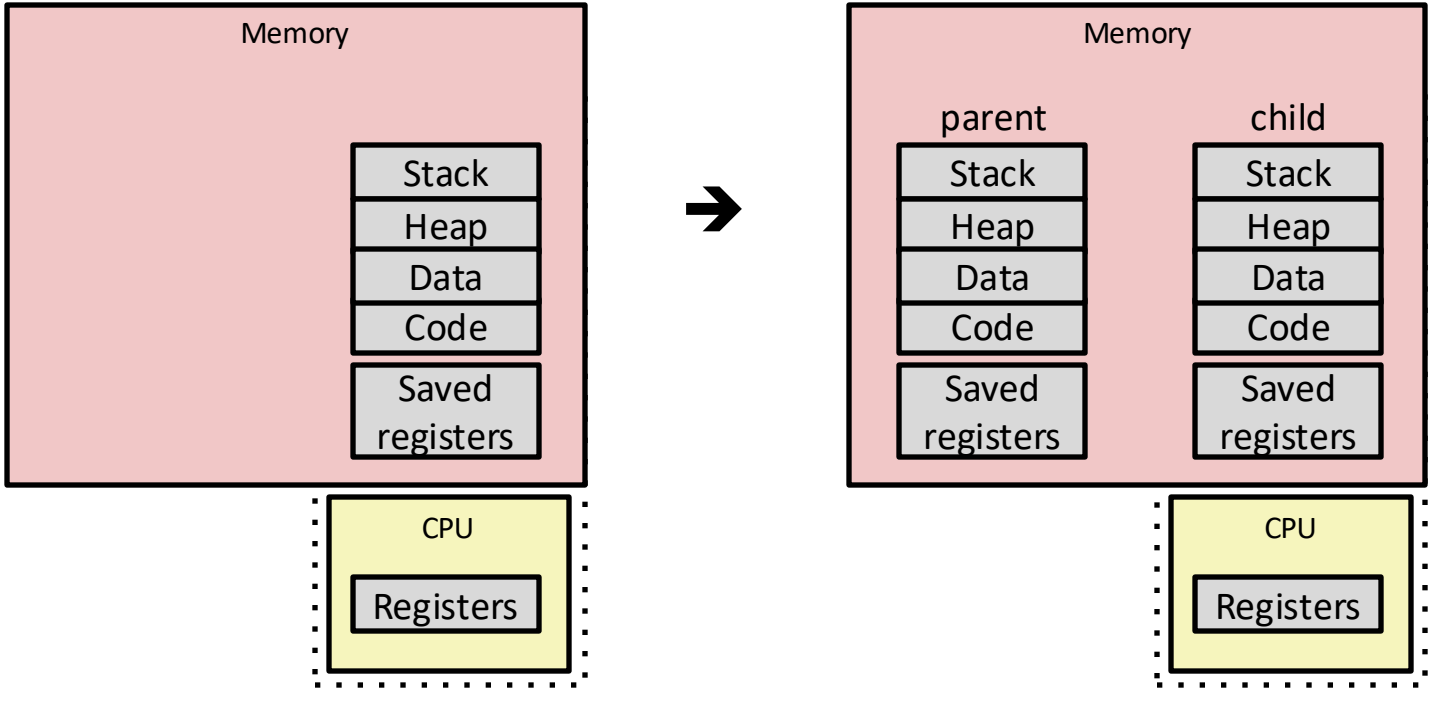
- The return values from fork are different.
- The process IDs are different (`getpid`() to obtain process ID).
- The two processes have different parent process IDs (`getppid`() to obtain parent process ID).
- File locks set by the parent are not inherited by the child.
- Pending alarms are cleared for the child.
- The set of pending signals for the child is set to the empty set.

Parent process and child process almost identical

- The two processes have
  - the same program text.
  - but separate copies of the data, stack, and heap segments.
- The child's data, stack, and heap segments are **initially** exact duplicates of the corresponding parts the parent's memory.
- After the `fork`(), each process can modify the variables in its data, stack, and heap segments without affecting the other process.



Conceptual View of `fork`



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

Memory semantics of `fork()`

- **Problems** of **direct copying** parent's (virtual) memory pages when forking.
  - **Slow process creation** - the child process won't be ready until all the memory copying is done.
  - **Waste of resources and time**
    - Many resources could be **shared** between parent and child.
    - fork() is usually immediately followed by an `exec` function to run a different program
      - Causes a *replacement* of the child process text segment, and *reinitialization* of the data, stack and heap segments, making the direct copying when forking a **waste**

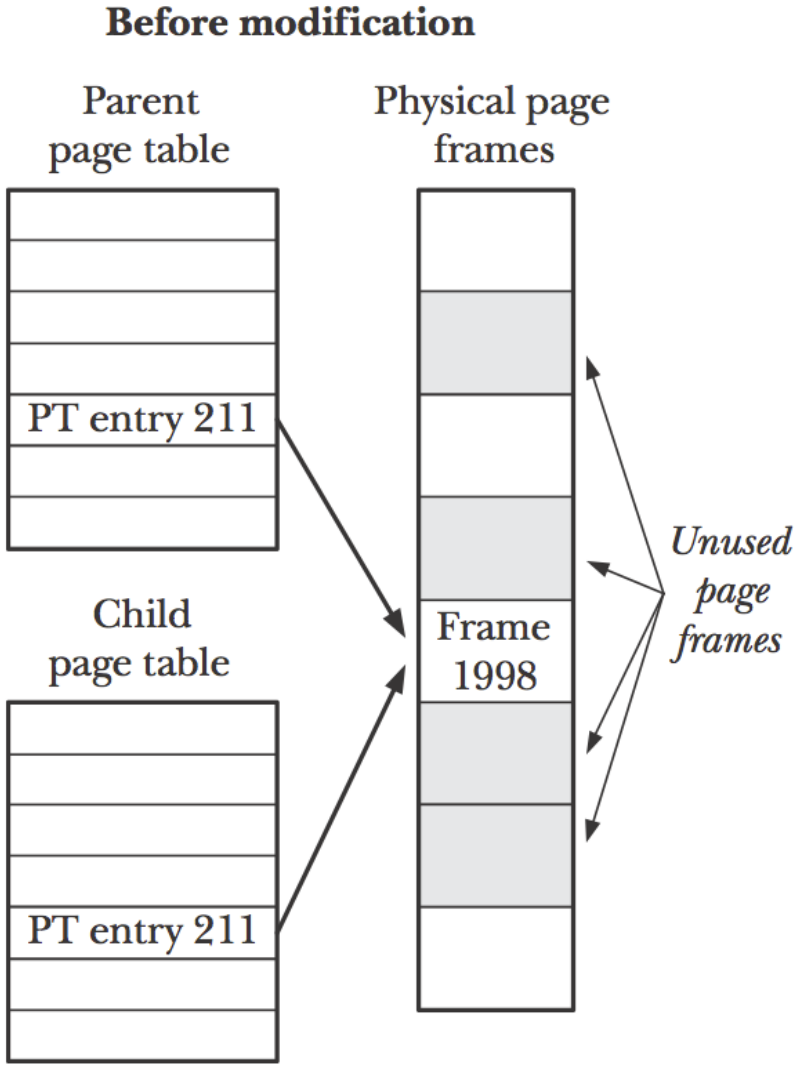
Memory semantics of `fork()`

Most modern Unix implementations use two techniques to avoid the wasteful copying:

- For the **text segment** of each process, the kernel marks it as **read-only**.
  - A process cannot modify its own code.
  - When fork() creates text segment of the child, it builds the page table entries for the text segment that have **the same virtual/physical mappings** as the parent.
- Applying the **copy-on-write** technique for data, stack, and heap segments

Copy-on-write for `fork()`

- After fork(), the child and parent **share the same memory mappings**
- Memory pages (data, stack, heap) are **marked as read-only** for both processes
- When either process tries to **modify a page**, the kernel
  - Detects the write attempt
  - Creates a **duplicate copy** of the page for the modifying process
  - Updates the process's page table with the new copy
- **Both processes now have separate writable copies** of the modified pages
- **This avoids unnecessary duplication** and improves performance



`fork` Example

```
1  int main(int argc, char** argv)
2  {
3      pid_t pid;
4      int x = 1;
5
6      pid = fork();
7      if (pid == 0) {  /* Child */
8          printf("child : x=%d\n", ++x);
9       return 0;
10     }
11
12     /* Parent */
13     printf("parent: x=%d\n", --x);
14     return 0;
15 }
```

- **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
```

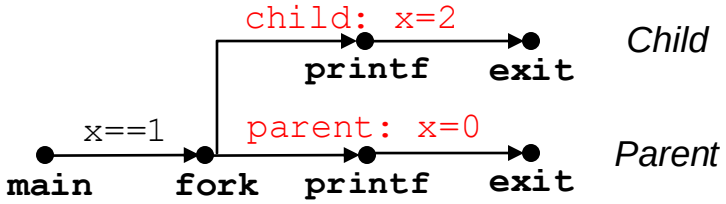
```
linux> ./fork
parent: x=0
child : x=2
```

Modeling `fork` with Process Graphs

- Represents partial ordering of statements in a concurrent program
- Vertices = Execution of statements
- Edges ($a \rightarrow b$) = a happens before b
- Labels on edges may show variable values.
- printf vertices indicate output
- Starts with a vertex having no incoming edges
- Topological sort gives a valid total execution order

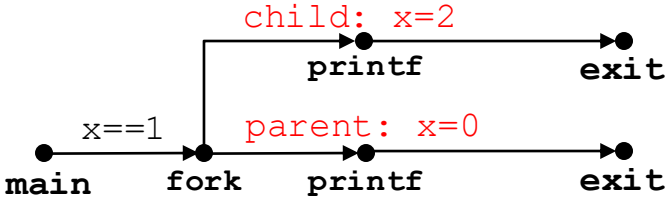
Process Graph Example

```c
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;
}
```

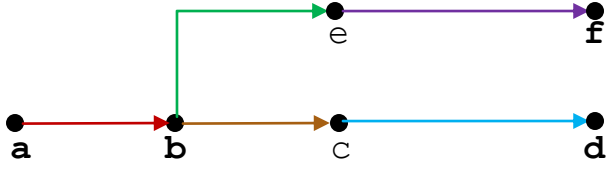


Interpreting Process Graphs

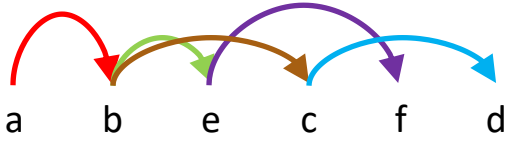
- Original graph:



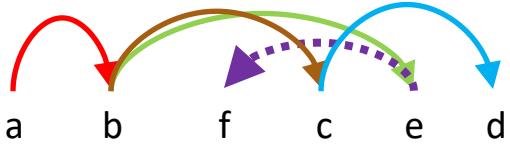
- Relabled graph:



Feasible total ordering:



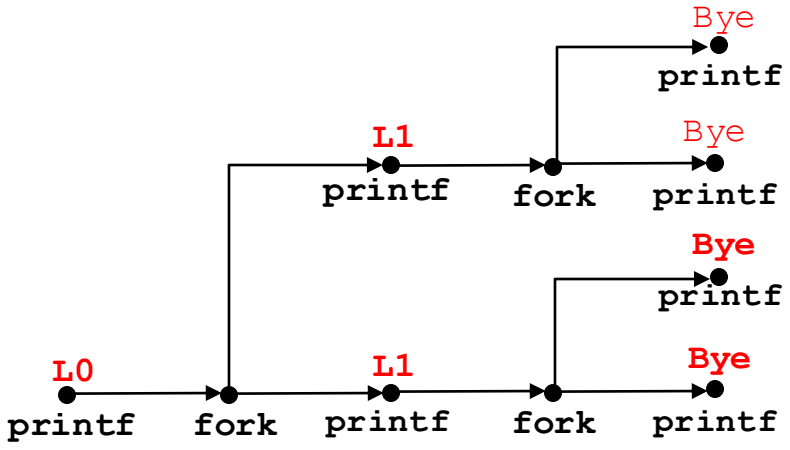
Feasible or Infeasible?



Infeasible: not a topological sort

`fork` Example: Two consecutive `forks`

```
1 void fork2()
2 {
3     printf("L0\n");
4     fork();
5     printf("L1\n");
6     fork();
7     printf("Bye\n");
8 }
```



Feasible

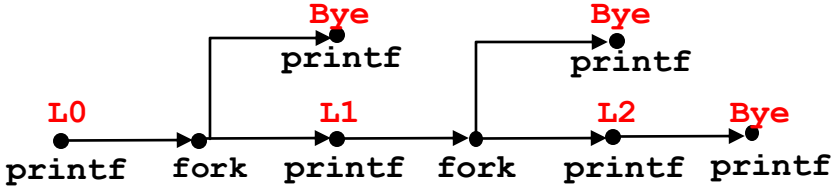
```
L0
L1
Bye
Bye
L1
Bye
Bye
```

Infeasible

```
L0
Bye
L1
Bye
L1
Bye
Bye
```

`fork` Example: Nested `forks` in parent

```
1  void fork4()
2  {
3      printf("L0\n");
4      if (fork() != 0) {
5          printf("L1\n");
6          if (fork() != 0) {
7              printf("L2\n");
8      }
9      }
10     printf("Bye\n");
11 }
```



Feasible or Infeasible?

L0
Bye
L1
Bye
Bye
L2

Infeasible

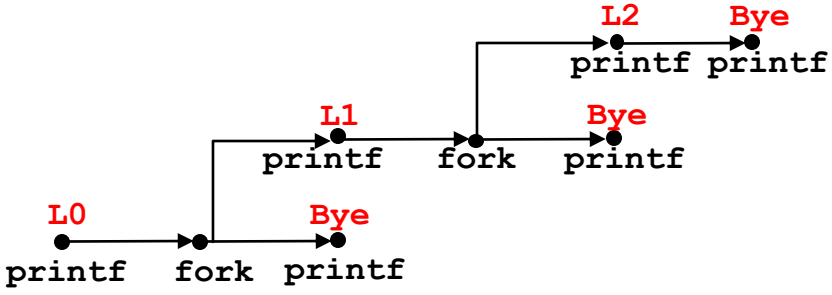
Feasible or Infeasible?

L0
L1
Bye
Bye
L2
Bye

Feasible

`fork` Example: Nested `forks` in children

```
1  void fork5()
2  {
3      printf("L0\n");
4      if (fork() == 0) {
5          printf("L1\n");
6          if (fork() == 0) {
7              printf("L2\n");
8          }
9      }
10     printf("Bye\n");
11 }
```



Feasible or Infeasible?

L0
Bye
L1
Bye
Bye
L2

Infeasible

Feasible or Infeasible?

L0
Bye
L1
L2
Bye
Bye

Feasible

Quiz

After a program executes the following series of fork() calls, how many new processes will result (assuming that none of the calls fails)?

```
fork();
fork();
fork();
```

The `vfork()` function

```
#include <unistd.h>

pid_t vfork(void);
```

In parent: returns process ID of child on success, or −1 on error;
in successfully created child: always returns 0

- `vfork()` function is used in the same way as `fork()`
- `vfork()` is generally designed to be used in programs where the child performs an immediate `exec` call

The `vfork()` function

- Difference between `vfork()` from `fork()`, `vfork()` has the following features
  - **No duplication** of parent process address space.
  - The child **shares** the parent's memory until it either performs a successful `execve()` or calls `_exit()` to terminate.
  - Execution of the parent process is **suspended** until the child has performed an `exec()` or `_exit()`
  - The child process is **always** scheduled to run **before** the parent process with `vfork()` (but not true for `fork()`)
  - `vfork()` is more **efficient** since it does not make a copy of parent's address space, however, be careful to avoid modification of parent's address space

The `vfork()` function

- The `vfork()` function originated with early BSD Unix, whose implementation of fork() is performing direct copy after forking (i.e., **without COW**)
- BSD introduced `vfork()` to solve the inefficiency problem of `fork()`
- Now modern Unix implementations have COW for `fork()`, thus largely eliminating the need for `vfork()`
- Linux (and many other UNIX implementations) provide a `vfork()` system call with BSD semantics for programs that **require the fastest possible fork**
  - Except where speed is absolutely critical, new programs should avoid the use of `vfork()` in favor of `fork()`

Reaping Child Processes

- Idea
  - When process terminates, it **still consumes** system resources, kept around in a terminated state
    - Examples: Exit status, various OS tables
  - Called a "zombie" process
    - Terminated but not yet reaped
- **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 should be reaped by **`init`** process (pid == 1)
    - Unless it was **`init`** that terminated! Then need to reboot…
  - Only need explicit reaping in long-running processes
    - e.g., shells and servers

Zombie Example

```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 */
    }
}
```

```
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

```
1  void fork8()
2  {
3      if (fork() == 0) {
4          /* Child */
5          printf("Running Child, PID = %d\n",
6                 getpid());
7          while (1)
8              ; /* Infinite loop */
9      } else {
10         printf("Terminating Parent, PID = %d\n",
11                getpid());
12         exit(0);
13     }
14 }
```

```
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
```

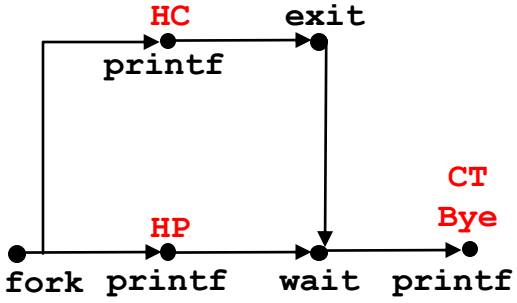
- Child process still active even though parent has terminated
- Must kill child explicitly, or else will keep running indefinitely

`wait`: Synchronizing with Children

- Parent **reaps** a child process with one of these system calls:
- `pid_t wait(int *status)`
  - Suspends current process until one of its children terminates
  - Returns PID of child, records exit status in **`status`**
- `pid_t waitpid(pid_t pid, int *status, int options)`
  - More flexible version of **`wait`**
  - Can wait for a specific child or group of children
  - Can be told to return immediately if there are no children to reap

`wait`: Synchronizing with Children

```
1  void fork9() {
2      int child_status;
3
4      if (fork() == 0) {
5          printf("HC: hello from child\n");
6           exit(0);
7      } else {
8          printf("HP: hello from parent\n");
9          wait(&child_status);
10         printf("CT: child has terminated\n");
11     }
12     printf("Bye\n");
13 }
```



Feasible output(s):

HC	HP
HP	HC
CT	CT
Bye	Bye

Infeasible output:

```
HP
CT
Bye
HC
```

wait: Status codes

- Return value of `wait` is the pid of the child process that terminated

- If `status != NULL`, then the integer it points to will be set to a value that indicates the exit status
  - More information than the value passed to **`exit`**
  - Must be decoded, using macros defined in sys/wait.h
    - `WIFEXITED, WEXITSTATUS, WIFSIGNALED, WTERMSIG, WIFSTOPPED, WSTOPSIG, WIFCONTINUED`

Another `wait` Example

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

```
1  void fork10() {
2      pid_t pid[N];
3      int i, child_status;
4
5      for (i = 0; i < N; i++)
6          if ((pid[i] = fork()) == 0) {
7              exit(100+i); /* Child */
8          }
9      for (i = 0; i < N; i++) { /* Parent */
10         pid_t wpid = wait(&child_status);
11         if (WIFEXITED(child_status))
12             printf("Child %d terminated with exit status %d\n",
13                    wpid, WEXITSTATUS(child_status));
14         else
15             printf("Child %d terminate abnormally\n", wpid);
16     }
17 }
```

`waitpid`: Waiting for a Specific Process

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

```
1  void fork11() {
2      pid_t pid[N];
3      int i;
4      int child_status;
5
6      for (i = 0; i < N; i++)
7          if ((pid[i] = fork()) == 0)
8              exit(100+i); /* Child */
9      for (i = N-1; i >= 0; i--) {
10         pid_t wpid = waitpid(pid[i], &child_status, 0);
11         if (WIFEXITED(child_status))
12             printf("Child %d terminated with exit status %d\n",
13                    wpid, WEXITSTATUS(child_status));
14         else
15             printf("Child %d terminate abnormally\n", wpid);
16     }
17 }
```

`sleep`: putting processes to sleep

- `unsigned int sleep(unsigned int secs);`
  - Put process into sleep state
  - returns 0 if the requested amount of time has elapsed
  - Or number of seconds still left to sleep

- `int pause(void);`
  - Puts the calling process to sleep until a signal is received by the process

`execve`: Loading and Running Programs

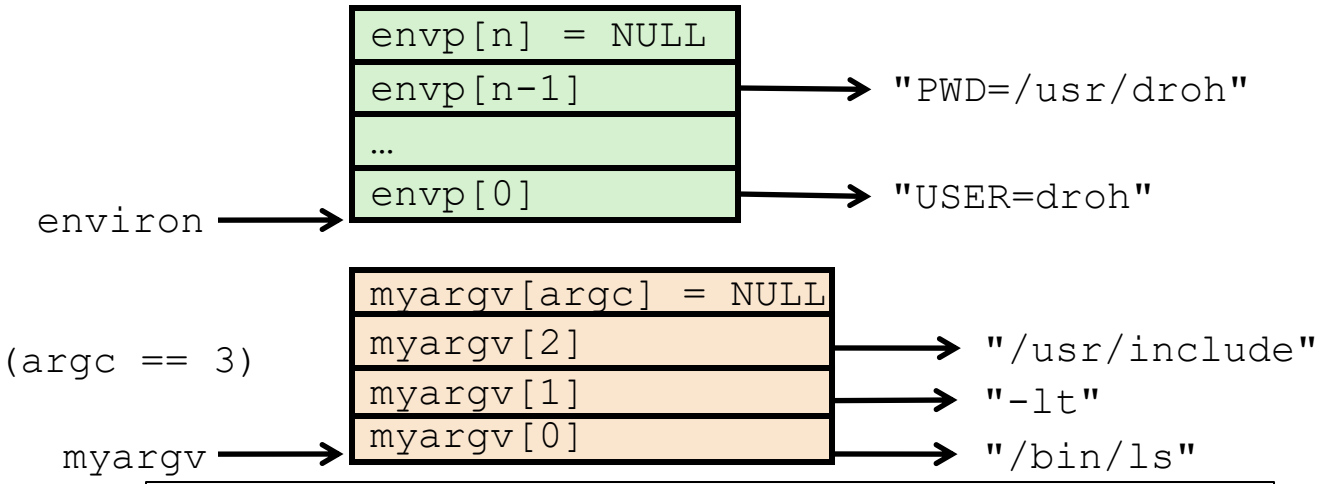
- `int execve(char *filename, char *argv[], char *envp[])`

- Loads and runs in the current process:
  - Executable file **`filename`**
    - (e.g., `#!/bin/bash`) Can be object file or script file beginning with `#!interpreter`
  - …with argument list **`argv`**
    - By convention **`argv[0]==filename`**
  - …and environment variable list **`envp`**
    - "name=value" strings

- **Overwrites** code, data, heap, and stack segments
  - Retains PID, open files and signal context
- Called once and never returns
  - …except if there is an error

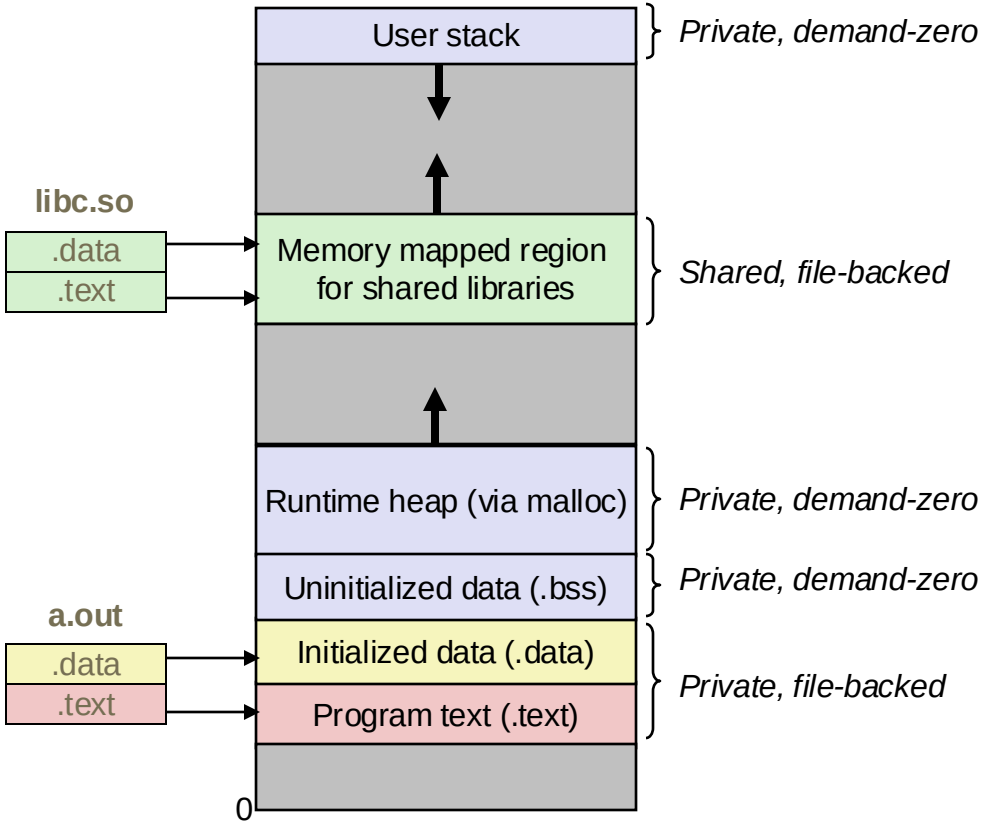
execve Example

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




```
1 if ((pid = fork()) == 0) {   /* Child runs program */
2   if (execve(myargv[0], myargv, environ) < 0) {
3     printf("%s: %s\n", myargv[0], strerror(errno));
4     exit(1);
5     }
6 }
```

`execve` and process memory layout



- To load and run a new program `a.out` in the current process using `execve`
- Free `vm_area_struct`'s and page tables for old areas
- Create `vm_area_struct`'s and page tables for new areas
  - Programs and initialized data backed by object files.
- Set PC to entry point in `.text`
  - Linux will fault in code and data pages as needed

fork and execve to run programs

- Programs like shells and Web servers make heavy use of `fork` and `execve`
  - Traditional web server for handling connections, or to show dynamic contents
  - Use shell to run commands or program

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

```
1  int main(int argc, char** argv)
2  {
3      char cmdline[MAXLINE]; /* command line */
4
5      while (1) {
6          /* read */
7          printf("> ");
8          fgets(cmdline, MAXLINE, stdin);
9          if (feof(stdin))
10             exit(0);
11
12         /* evaluate */
13         eval(cmdline);
14     }
15     ...
```

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 */
```

`parseline` will parse 'buf' into 'argv' and 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

```
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]`.
`execve` only returns on error.
127 used for command not found

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.

shellex.c

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
  - Could run the entire computer out of memory
    - Moreover, run out of PIDs

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

Summary (cont.)

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