CSCI2467: Systems Programming Concepts

Slideset 7: Exceptional Control Flow

Source: CS:APP Chapter 8, Bryant & O'Hallaron

Course Instructors:

Matthew Toups
Caitlin Boyce

Course Assistants:

Saroj Duwal
David McDonald

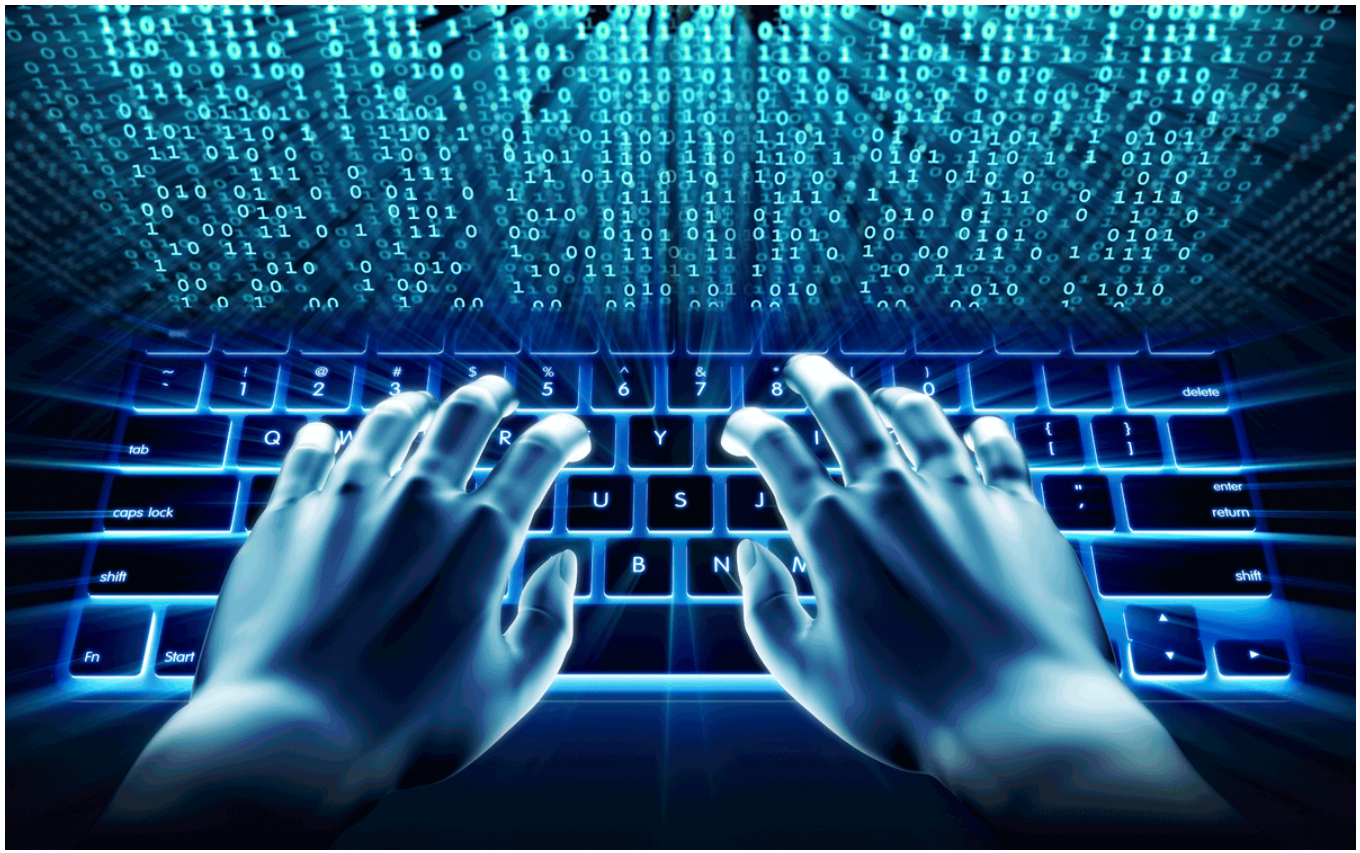
Spring 2020

THE UNIVERSITY of NEW ORLEANS

DEPARTMENT OF COMPUTER SCIENCE

Bash script extra credit

Your (first) shell script (using bash)



Bash script extra credit

Your (first) shell script (using bash)

- Due in Autolab by 11:59pm
- Should only be a few (6-10) lines
- Don't use absolute path to `grep.tar`
  - Autolab won't have your own home directory
  - just assume grep.tar is in the current directory where the script *starts*
  - either copy it into new directory, or use `../grep.tar`
- Don't forget comments!
  - for each step
  - no points if comments omitted

Today

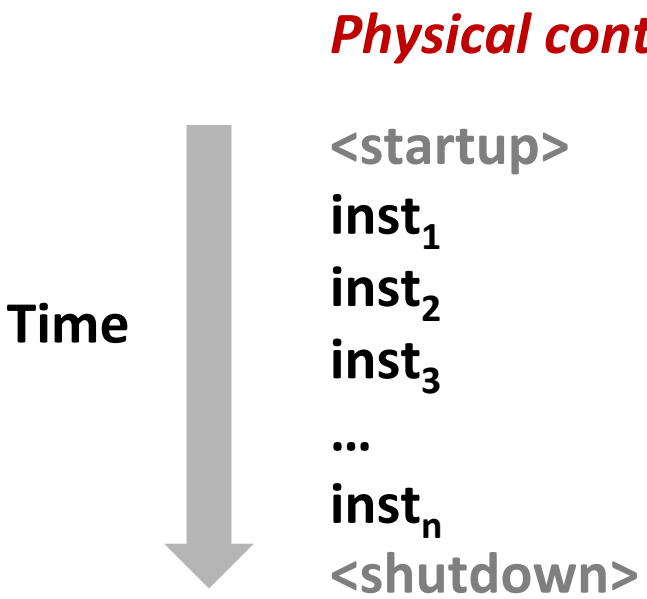
- Shell lab

1. Intro to processses
   - Control flow
   - Processes

2. Exceptional Control Flow
   - Process Control
   - Exceptions
   - Summary

Control Flow

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



Altering the Control Flow

- **Up to now: two mechanisms for changing control flow:**
  - Jumps and branches
  - Call and return

  React to changes in ***program state***

Altering the Control Flow

- **Up to now: two mechanisms for changing control flow:**
  - Jumps and branches
  - Call and return

  React to changes in ***program state***

- **Insufficient for a useful system:**
  Difficult to react to changes in *system state*
  - Data arrives from a disk or a network adapter
  - Instruction divides by zero
  - User hits Ctrl-C at the keyboard
  - System timer expires

- **System needs mechanisms for "exceptional control flow"**

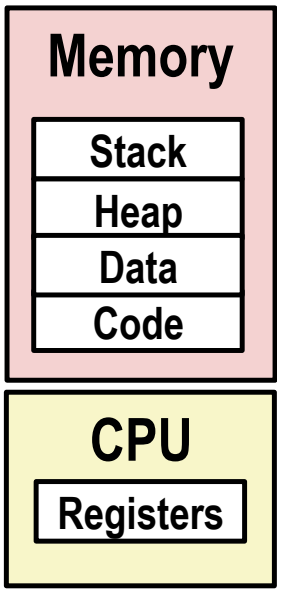
Exceptional Control Flow

- **Exists at all levels of a computer system**
- **Low level mechanisms**
  - 1. **Exceptions**
    - Change in control flow in response to a system event (i.e., change in system state)
    - Implemented using combination of hardware and OS software
- **Higher level mechanisms**
  - 2. **Process context switch**
    - Implemented by OS software and hardware timer
  - 3. **Signals**
    - Implemented by OS software
  - 4. **Nonlocal jumps**: `setjmp()` and `longjmp()`
    - Implemented by C runtime library

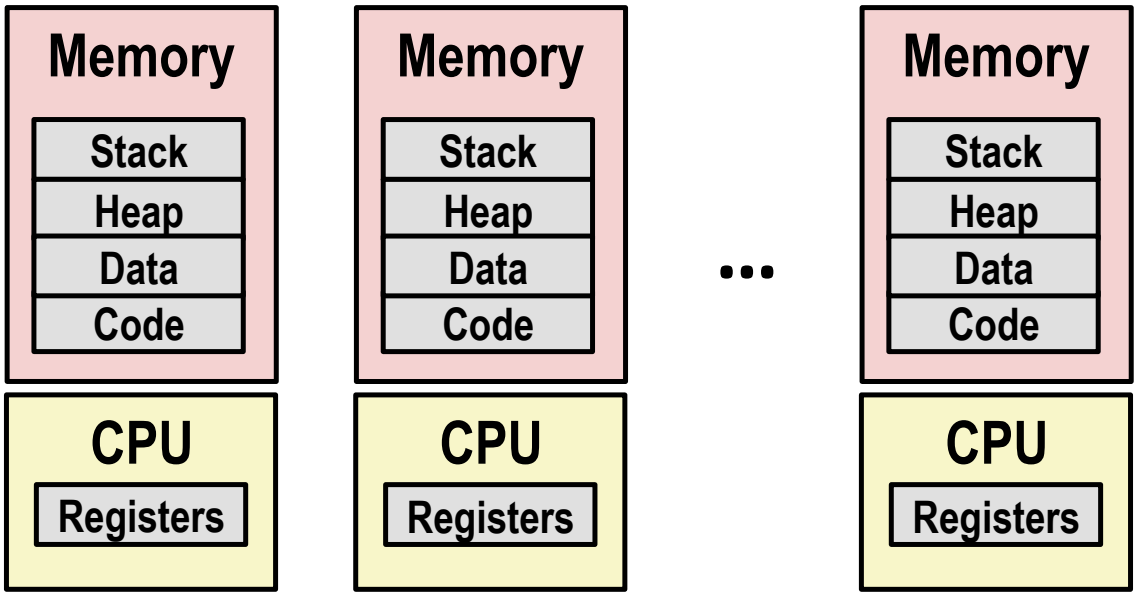
Processes

- **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”

- **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

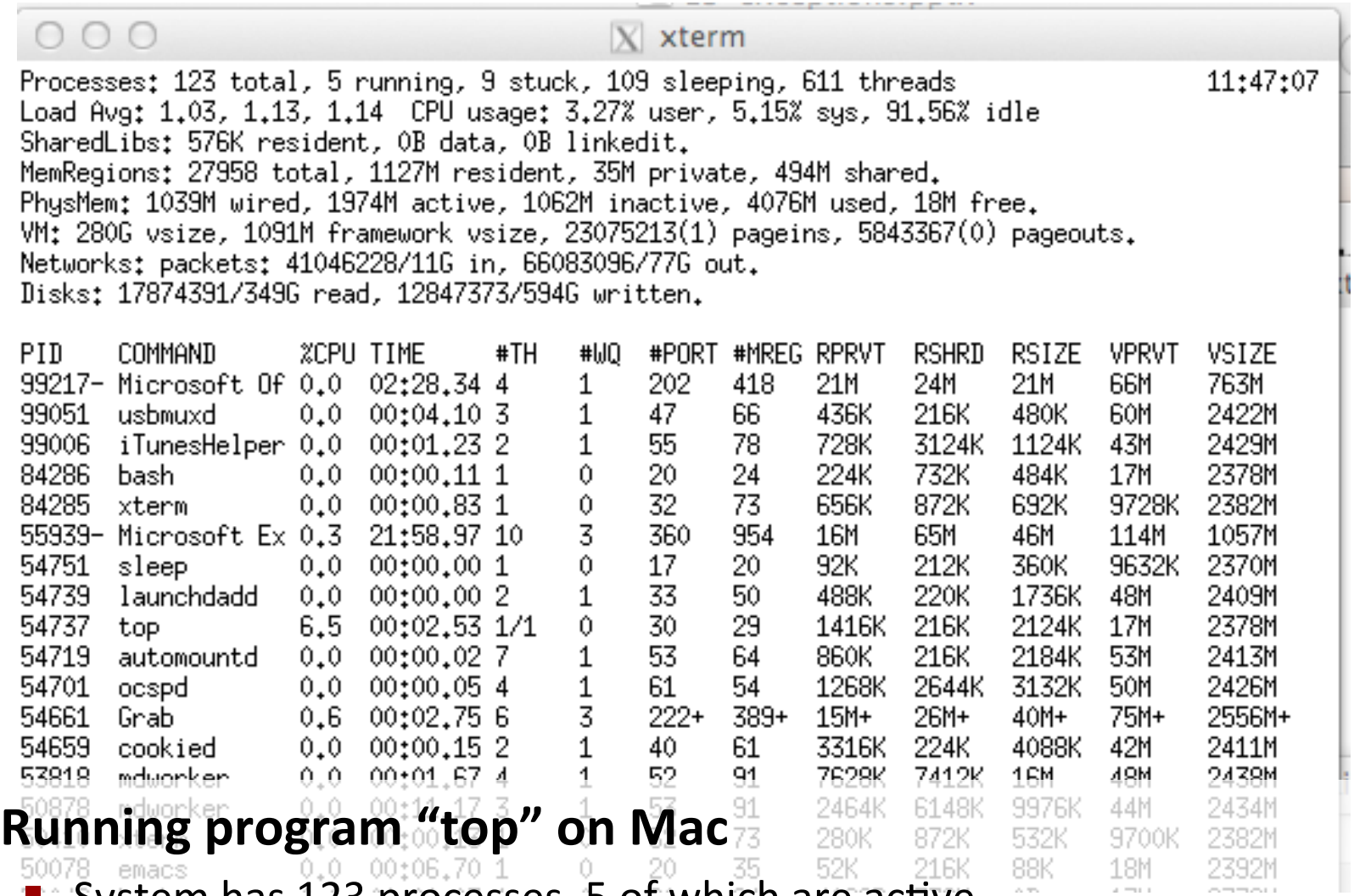
Multiprocessing example (Linux)

ps command

```
csadmin@math209:~$ ps
  PID TTY          TIME CMD
 1775 pts/0    00:00:00 bash
 5124 pts/0    00:00:00 ps
csadmin@math209:~$
```

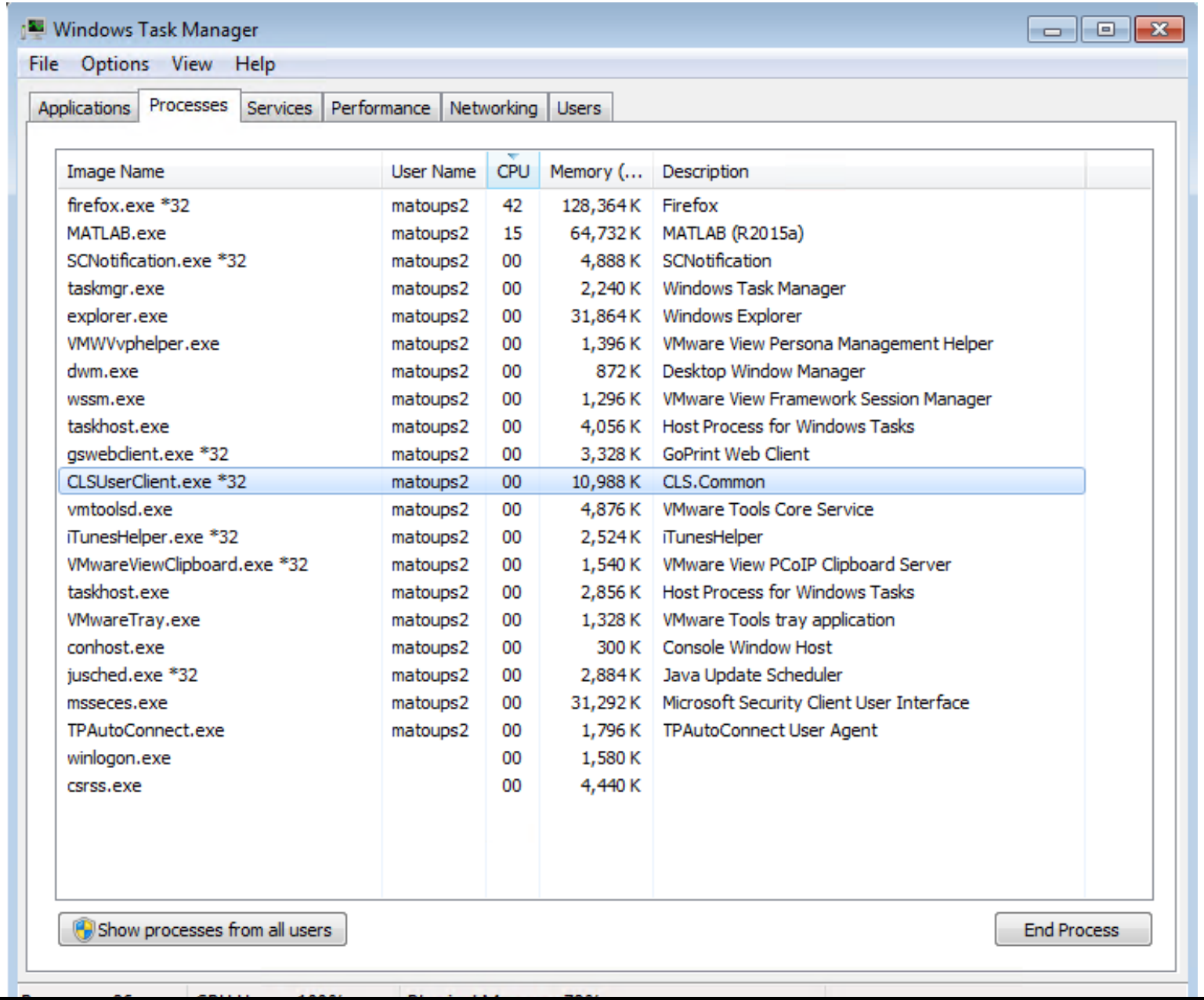
```
csadmin@math209:~/2467/shlab-handout$ ./myspin 10 &
[1] 5304
csadmin@math209:~/2467/shlab-handout$ ./myspin 20 &
[2] 5306
csadmin@math209:~/2467/shlab-handout$ ps
  PID TTY          TIME CMD
 1775 pts/0    00:00:00 bash
 5304 pts/0    00:00:00 myspin
 5306 pts/0    00:00:00 myspin
 5307 pts/0    00:00:00 ps
csadmin@math209:~/2467/shlab-handout$
```

Multiprocessing example (Mac OS)

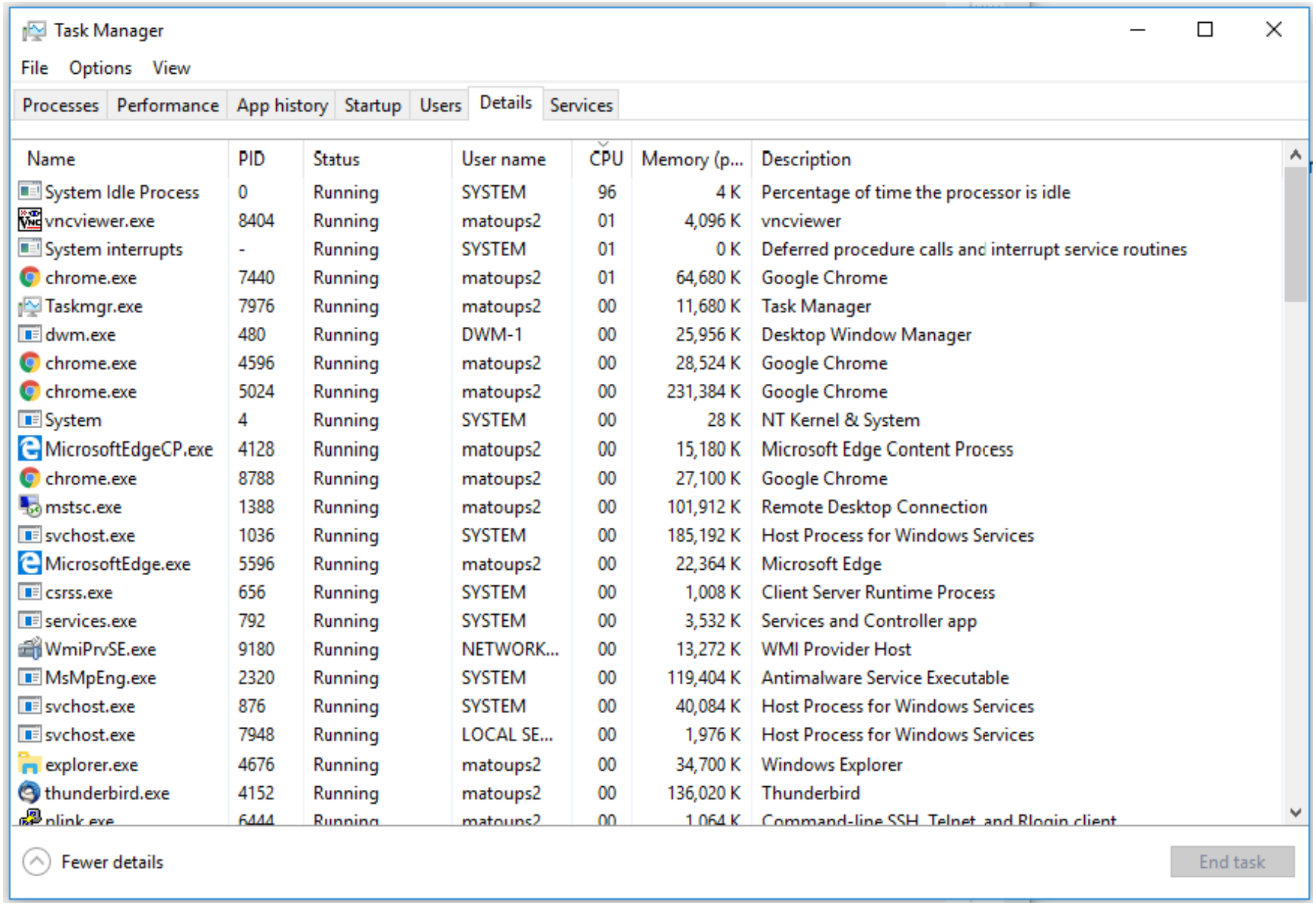


- **Running program "top" on Mac**
  - System has 123 processes, 5 of which are active
  - Identified by Process ID (PID)

Multiprocessing example (Windows 7)



Multiprocessing example (Windows 10)



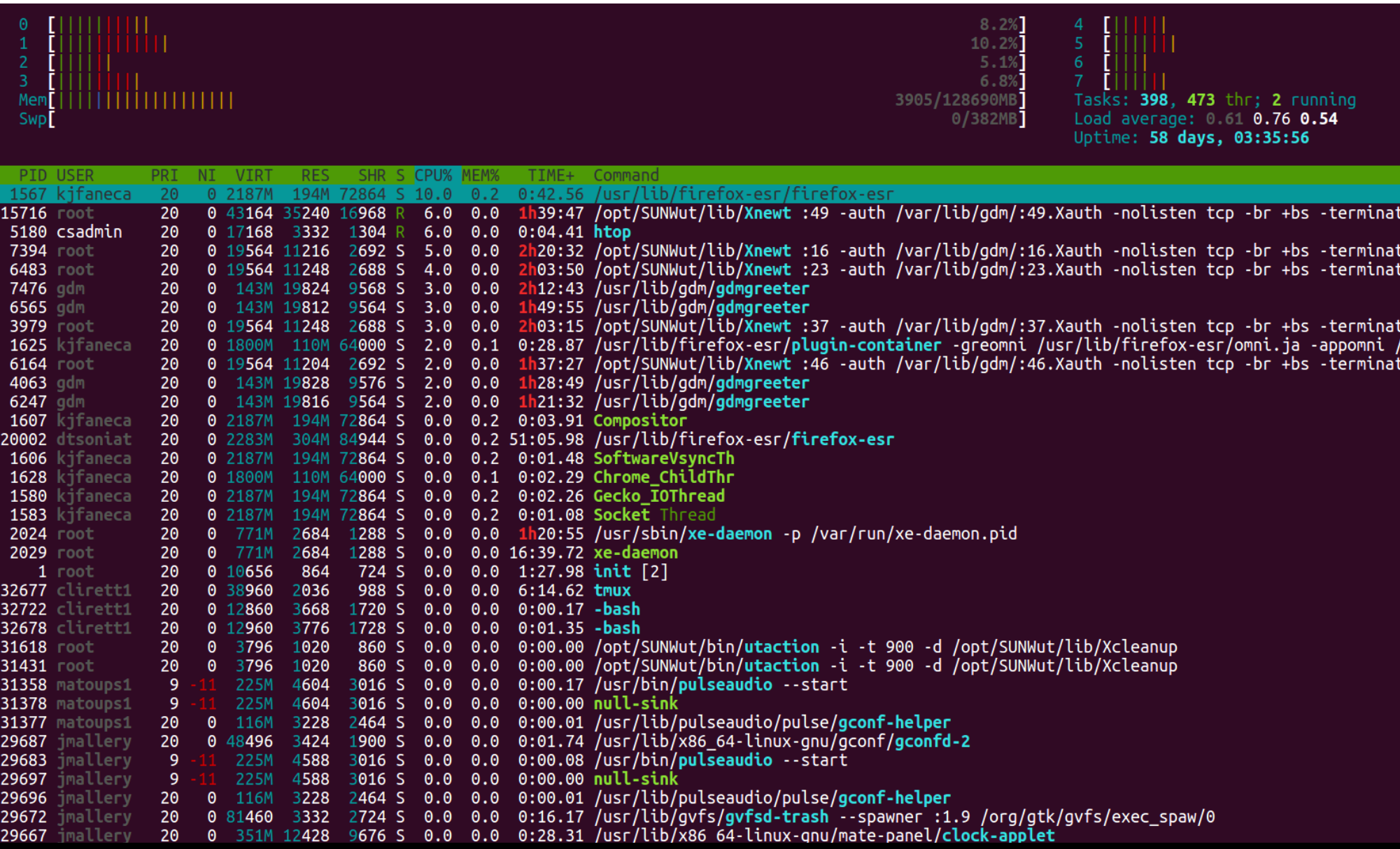
Multiprocessing example (Linux)

`ps af` command

```
  PID TTY      STAT   TIME COMMAND
32722 pts/10   Ss+    0:00 -bash
32678 pts/8    Ss+    0:01 -bash
28133 pts/4    Ss     0:00 sh -c /bin/bash
28136 pts/4    S+     0:00  \_ /bin/bash
25137 pts/5    Ss     0:00 -bash
30336 pts/5    S+     0:00  \_ /usr/bin/perl /usr/bin/dpkg-buildpackage -rfakeroot
30454 pts/5    S+     0:00      \_ /usr/bin/make -f debian/rules build
30455 pts/5    S+     0:00          \_ /usr/bin/perl -w /usr/bin/dh build --with python3 --buildsys
30459 pts/5    S+     0:00              \_ /usr/bin/perl -w /usr/bin/dh_auto_configure -O--buildsys
22407 pts/3    Ss     0:00 -bash
30468 pts/3    R+     0:00  \_ ps fa
20840 pts/1    Ss     0:00 -bash
20926 pts/1    S      0:00  \_ su - csadmin
20934 pts/1    S+     0:00      \_ -su
17962 pts/2    Ss+    0:00 -bash
 9922 pts/9    Ss     0:00 sh -c /bin/bash
 9926 pts/9    S+     0:00  \_ /bin/bash
 1129 pts/48   Ss     0:00 sh -c /bin/bash
 1132 pts/48   S+     0:00  \_ /bin/bash
25684 pts/5    S      0:00 sh -c ./sdriver.pl -t trace01.txt -s ./tsh -a '-p'
25685 pts/5    S      0:00  \_ /usr/bin/perl ./sdriver.pl -t trace01.txt -s ./tsh -a -p
25686 pts/5    S      0:00      \_ ./tsh -p
 3481 tty6     Ss+    0:00 /sbin/getty 38400 tty6
 3480 tty5     Ss+    0:00 /sbin/getty 38400 tty5
 3479 tty4     Ss+    0:00 /sbin/getty 38400 tty4
 3478 tty3     Ss+    0:00 /sbin/getty 38400 tty3
 3477 tty2     Ss+    0:00 /sbin/getty 38400 tty2
```

Multiprocessing example (Linux)

`htop` command

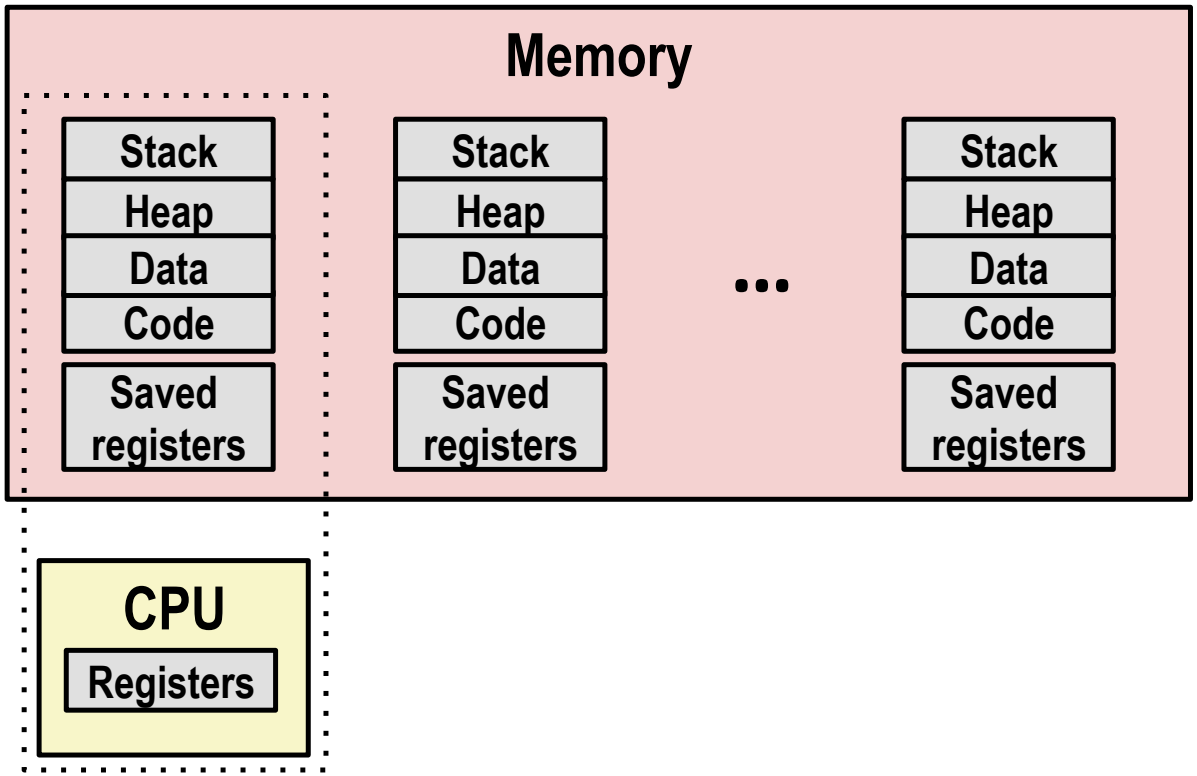


Multiprocessing example (Linux)

`glances` command

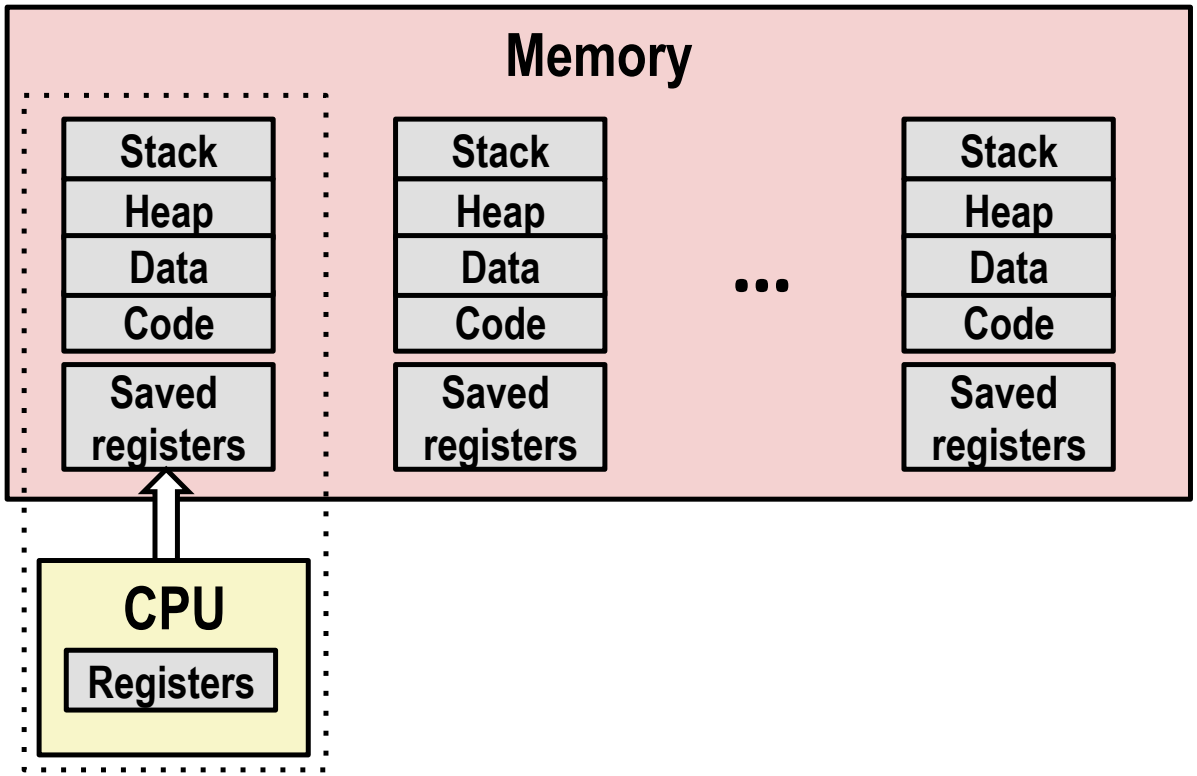
```
TASKS 259 (936 thr), 1 run, 258 slp, 0 oth sorted automatically by cpu_percent, tree view

  CPU%  MEM%  VIRT   RES   PID USER        NI S    TIME+ IOR/s IOW/s Command
   0.0   0.0  181M 5.06M     1 root         0 S  3:34.32     0     0 /sbin/init splash
   0.0   0.0  342M 3.90M  1222 root         0 S  0:02.93     0     0 ├─ /usr/sbin/lightdm
   0.0   0.0  225M 5.69M  1636 root         0 S  0:00.59     0     0 │  ├─ lightdm --session-child 12 19
   0.0   0.0 45.4M 3.14M  1992 mtoups       0 S  0:10.69     0     0 │  │  └─ /sbin/upstart --user
   0.0   5.1 3.62G  811M 18749 mtoups       0 S 58:49.58     0     0 │  │     ├─ /usr/lib/firefox/firefox -private
  10.9   4.7 3.07G  743M 11153 mtoups       0 S 11:58.84     0     0 │  │     │  ├─ /usr/lib/firefox/firefox -cont
   5.3  11.8 16.0G 1.84G 10769 mtoups       0 S 15:37.80     0     0 │  │     │  ├─ /usr/lib/firefox/firefox -cont
   0.3   3.0 2.86G  471M 18813 mtoups       0 S 21:40.65     0     0 │  │     │  ├─ /usr/lib/firefox/firefox -cont
   0.0   3.7 2.79G  595M 11186 mtoups       0 S 32:42.62     0     0 │  │     │  ├─ /usr/lib/firefox/firefox -cont
   0.0   0.6 1.84G 88.1M  5698 mtoups       0 S  1:47.94     0     0 │  │     │  └─ /usr/lib/firefox/firefox -cont
  12.5   4.9 3.65G  783M  3056 mtoups       0 S 18:46.47     0     0 │  │     ├─ compiz
   2.8   0.5  732M 81.3M  7687 mtoups       0 S  1:39.29     0     0 │  │     ├─ /usr/lib/gnome-terminal/gnome-ter
   0.0   0.0 23.1M 6.00M  2303 mtoups       0 S  0:00.40     0     0 │  │     │  ├─ bash
   6.2   0.2 93.2M 29.9M  4096 mtoups       0 R  1:04.82     0     0 │  │     │  │  └─ /usr/bin/python3 /usr/bin/g
   0.0   0.0 23.0M 6.12M   947 mtoups       0 S  0:00.21     0     0 │  │     │  ├─ bash
   0.3   0.0 47.9M 5.09M  2158 mtoups       0 S  0:00.12     0     0 │  │     │  │  └─ ssh csadmin@math209.cs.uno.
   0.0   0.0 22.9M 5.97M 27951 mtoups       0 S  0:00.60     0     0 │  │     │  ├─ bash
   0.0   0.2  192M 26.3M 27974 mtoups       0 S  0:00.75     0     0 │  │     │  │  └─ vim FAIL
   0.0   0.0 24.0M 7.07M 28066 mtoups       0 S  0:01.98     0     0 │  │     │  ├─ bash
   0.0   0.0 22.9M 5.98M 30713 mtoups       0 S  0:01.84     0     0 │  │     │  ├─ bash
   0.0   0.0 48.0M 5.80M 18229 mtoups       0 S  0:00.17     0     0 │  │     │  │  └─ ssh csadmin@autolab.cs.uno.
   0.0   0.0 22.6M 2.07M  9877 mtoups       0 S  0:00.36     0     0 │  │     │  ├─ bash
   0.0   0.0 23.4M 5.57M 21113 mtoups       0 S  0:01.22     0     0 │  │     │  ├─ bash
   0.0   0.0 23.4M 5.33M 20982 mtoups       0 S  0:00.85     0     0 │  │     │  ├─ bash
   0.0   0.0 22.5M 3.56M 26694 mtoups       0 S  0:00.30     0     0 │  │     │  ├─ bash
   0.0   0.0 22.9M 6.02M 24970 mtoups       0 S  0:00.80     0     0 │  │     │  ├─ bash
   0.0   0.2  192M 27.4M 28683 mtoups       0 S  0:00.78     0     0 │  │     │  │  └─ vim README.md
   0.0   0.0 22.9M 6.00M 16403 mtoups       0 S  0:00.25     0     0 │  │     │  ├─ bash
   0.0   0.0 23.0M 5.86M 15823 mtoups       0 S  0:00.40     0     0 │  │     │  ├─ bash
   0.0   0.0 22.9M 6.02M 28625 mtoups       0 S  0:00.90     0     0 │  │     │  ├─ bash
   0.0   0.0 23.0M 6.00M 12036 mtoups       0 S  0:00.15     0     0 │  │     │  ├─ bash
   0.0   0.7 1.26G  115M  2080 mtoups       0 S  0:12.25     0     0 │  │     │  │  └─ evince 07ecf.pdf
```

Multiprocessing: The (traditional) reality



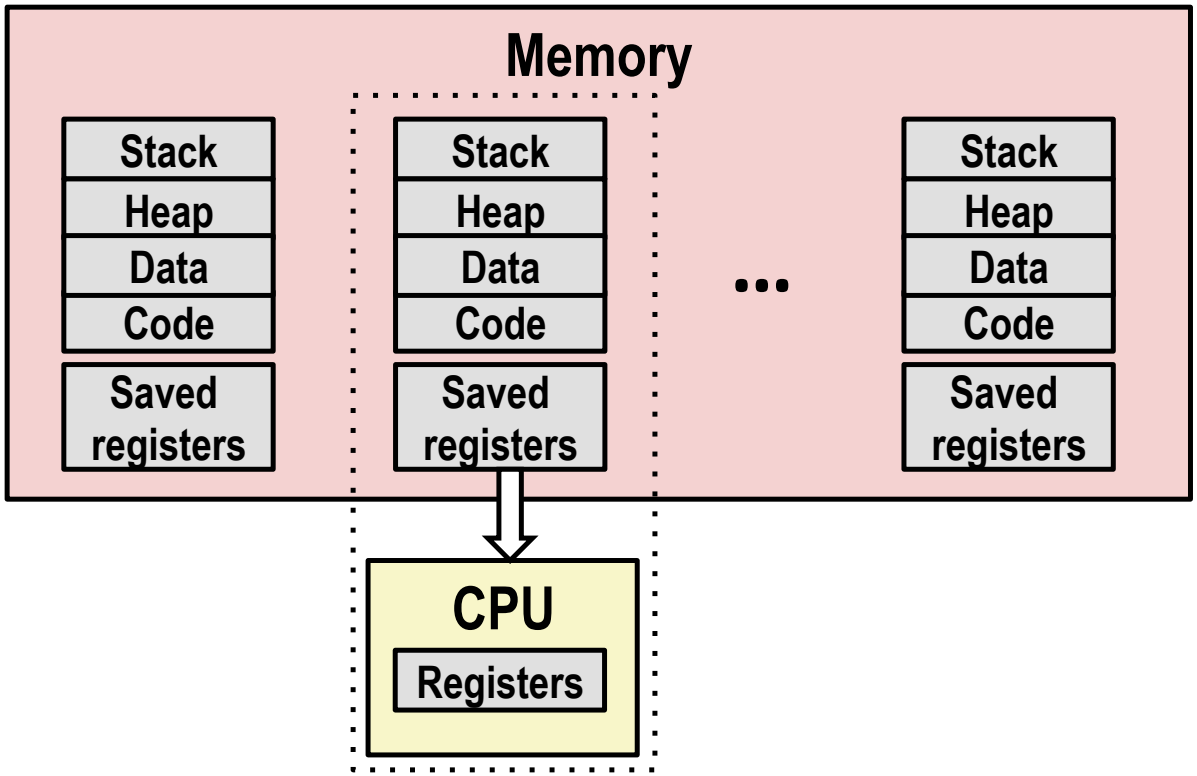
- **Single processor executes multiple processes concurrently**
  - Process executions interleaved (multitasking)
  - Address spaces managed by virtual memory system (later in course)
  - Register values for nonexecuting processes saved in memory

Multiprocessing: The (traditional) reality



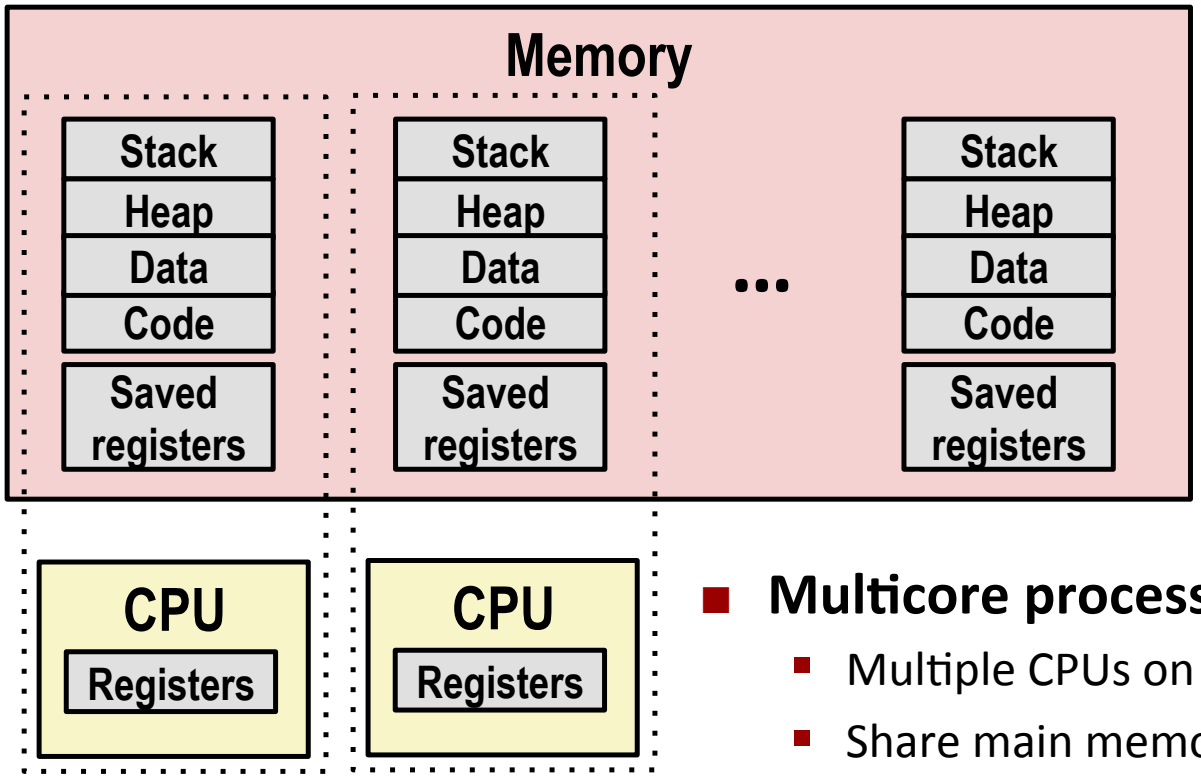
- **Save current registers in memory**

Multiprocessing: The (traditional) reality



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

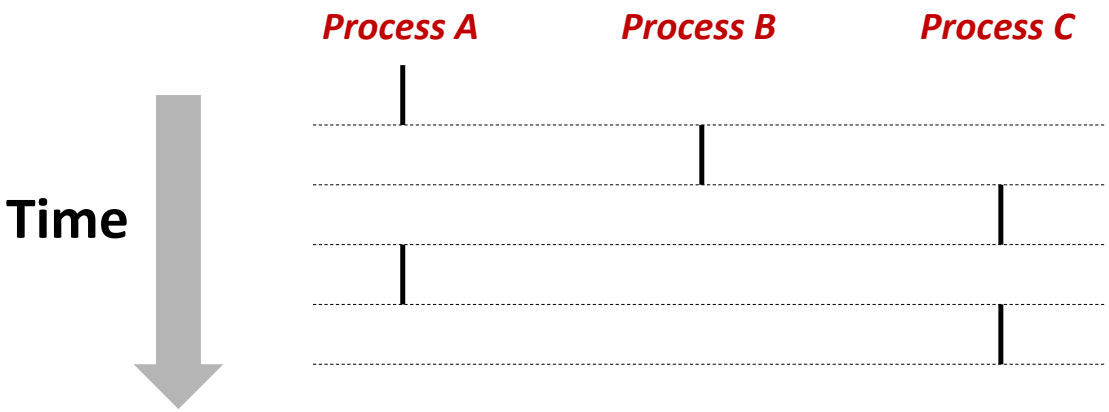
Multiprocessing: The (modern) reality



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

Concurrent processes

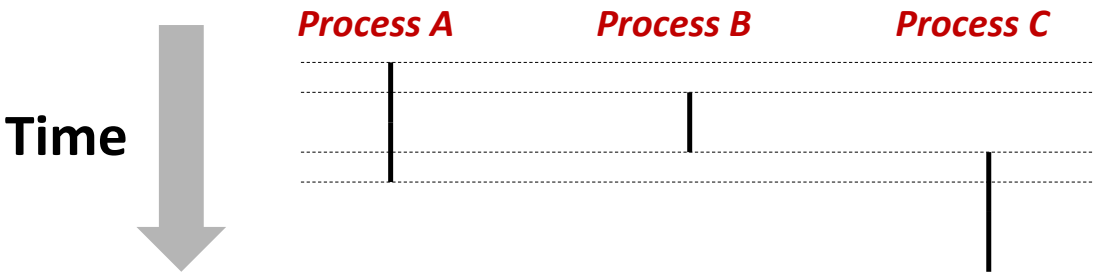
- **Each process is a logical control flow.**
- **Two processes *run concurrently* (*are concurrent)* if their flows overlap in time**
- **Otherwise, they are *sequential***
- **Examples (running on single core):**
  - Concurrent: A & B, A & C
  - Sequential: B & C



Concurrent processes

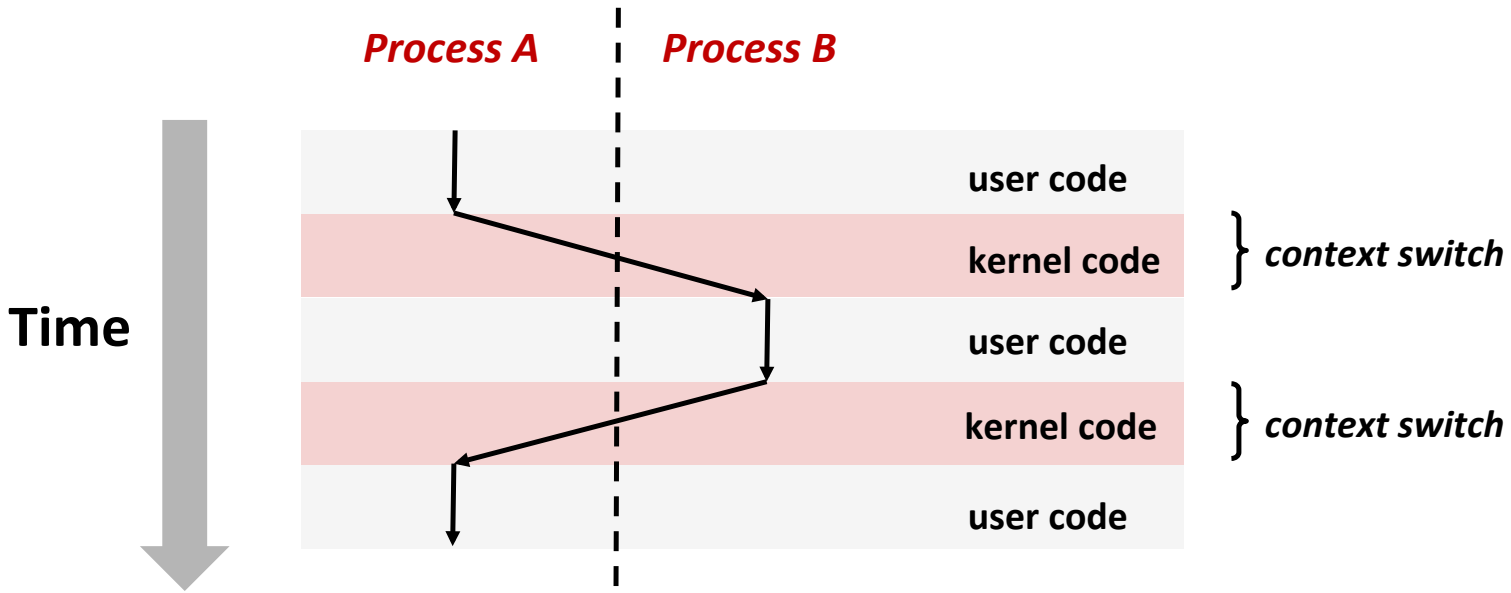
- **Control flows for concurrent processes are physically disjoint in time**

- **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

- Shell lab

1. Intro to processses
   - Control flow
   - Processes

2. Exceptional Control Flow
   - Process Control
   - Exceptions
   - Summary

Obtaining process IDs

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

Example from lab activity
(starting point)

```c
#include <sys/types.h>
#include <unistd.h>
#include <stdio.h>

void main()
{
        pid_t mypid;
        /* pid_t is defined as int (32-bit signed) */

        mypid = getpid();

        printf(" my PID is: %d\n", mypid);

}
```

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.**

Example of exit()

You can run this program on systems-lab:

```
$ /home/CSCI2467/labs/misc/ch08/exit
$ /home/CSCI2467/labs/misc/ch08/exit 4
```

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

    if (argc<2) { exit(7); }
    /* atoi converts char to int */
    value = atoi(argv[1]);

    printf(" value is %d\n", value);
    return value+1;
}
```

Notes:

- argc is **c**ount of arguments
- argv is **v**ector of arguments (array of strings)
- atoi() converts characters to int

Creating processes
The indispensable `fork()`

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

Creating processes
The indispensable `fork()`

- ***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

Creating processes
The indispensable `fork()`

- ***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***

Example from lab activity
(one line added)

```
void main()
{
        pid_t mypid;

        fork(); /* this line: crucial difference */
        mypid = getpid();

        printf(" my PID is: %d\n", mypid);
}
```

A more practical `fork()` use

- What's the point of doing this?

A more practical `fork()` use

- What's the point of doing this?
- Program does twice as much stuff

A more practical `fork()` use

- What's the point of doing this?
- Program does twice as much stuff
- But we want to be able to create a child process that can do more than just duplicate the parent

A more practical `fork()` use

- What's the point of doing this?
- Program does twice as much stuff
- But we want to be able to create a child process that can do more than just duplicate the parent
- Crucial observation: `fork()` has two *distinct* return values

A more practical `fork()` use

- What's the point of doing this?
- Program does twice as much stuff
- But we want to be able to create a child process that can do more than just duplicate the parent
- Crucial observation: `fork()` has two *distinct* return values
- Solution: we can use an `if( )` statement in order to have the parent and child do different things

Example from lab activity

```
void main()
{
  int mypid,forkresult;

  printf("program starts, pid is: %d\n", getpid());

  forkresult = fork();

  mypid = getpid();

  if (forkresult == 0) {
    printf("ZERO fork returned %d, my PID is: %d\n",
        forkresult, mypid);
  } else {
    printf("NONZERO fork returned %d, my PID is: %d\n"
        , forkresult, mypid);
  }
}
```

Running the fork example

When you ran this in class, your got a unique PID (each time)

```
program starts, pid is: 8805
NONZERO fork returned 8806, my PID is: 8805
ZERO fork returned 0, my PID is: 8806
```

`fork()` is essential!

- This is not an obscure detail
- This is what makes a multi-process system possible!
- All processes begin with `fork()`

  (called by `init`, `bash`, `sshd`, `firefox`, etc)

  ... also crucial to Shell Lab

fork example

See similar code on systems-lab: `/home/CSCI2467/labs/misc/forks.c`

```c
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

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

- **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**

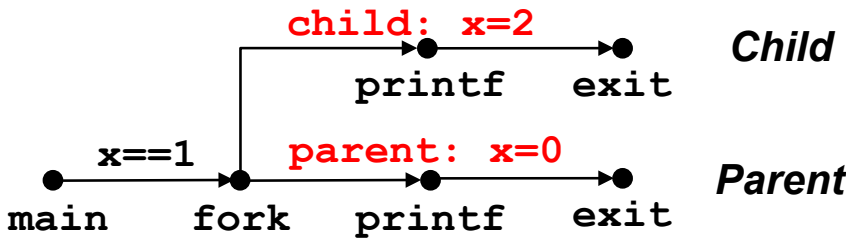
Modeling `fork` with process graphs

- **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

```c
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);
}
```

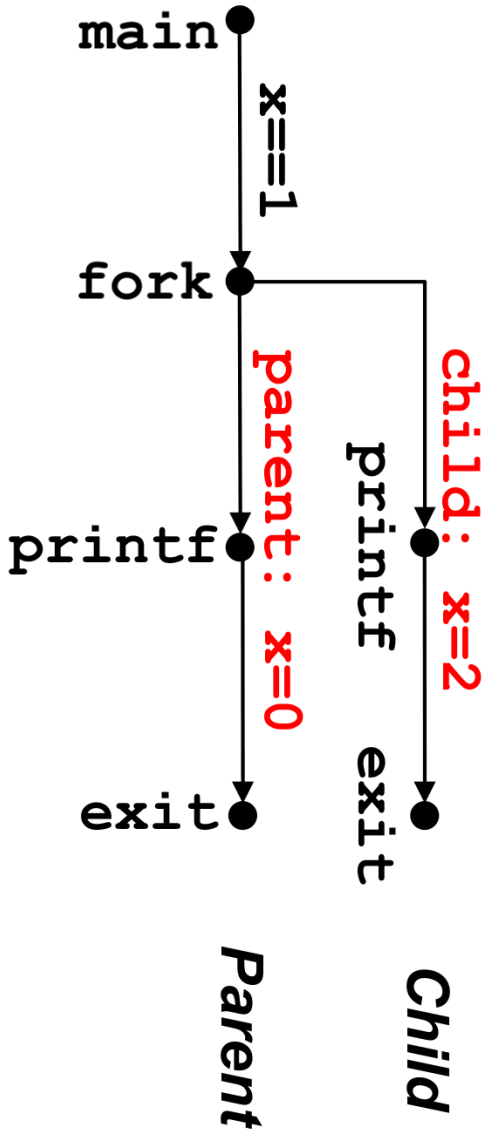
fork.c



Process graph example (rotated)

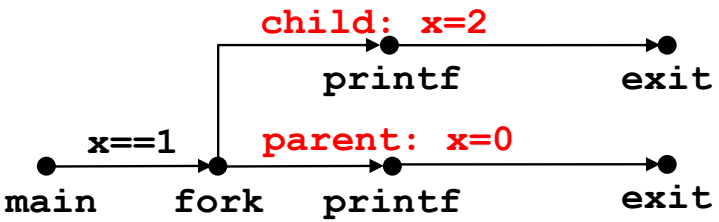
```c
int main()
{
    pid_t pid;
    int x = 1;

    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

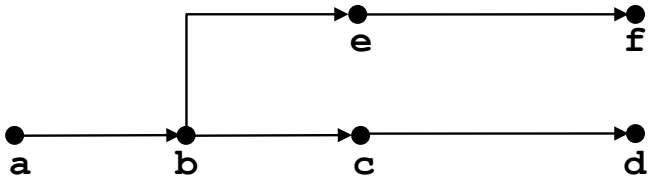


Interpreting process graphs

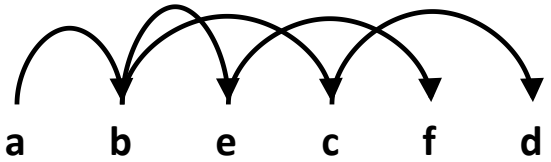
- **Original graph:**



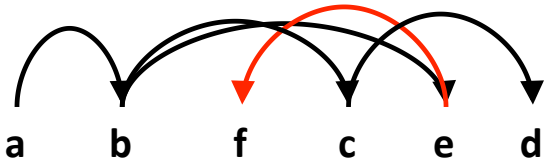
- **Relabled graph:**



Feasible total ordering:

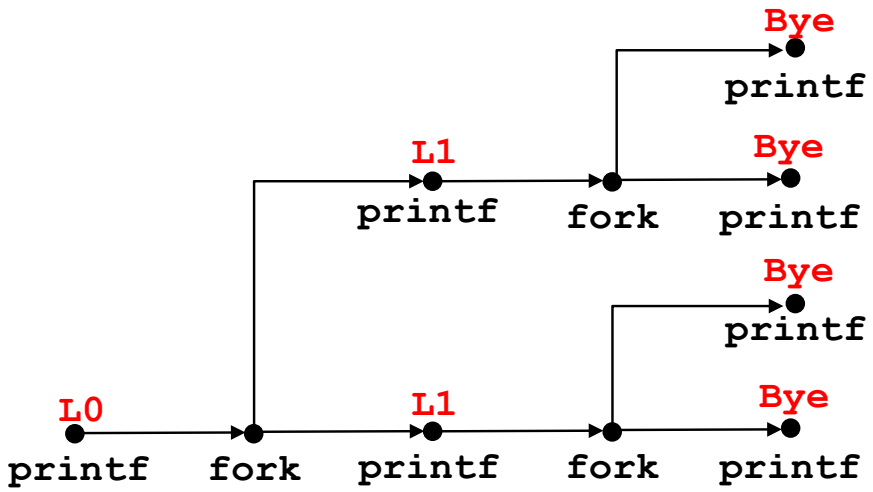


Infeasible total ordering:



fork() example: two consecutive fork()s

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



Feasible output:

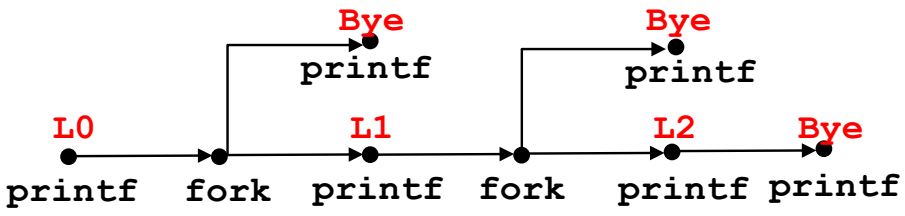
L0
L1
Bye
Bye
L1
Bye
Bye

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:

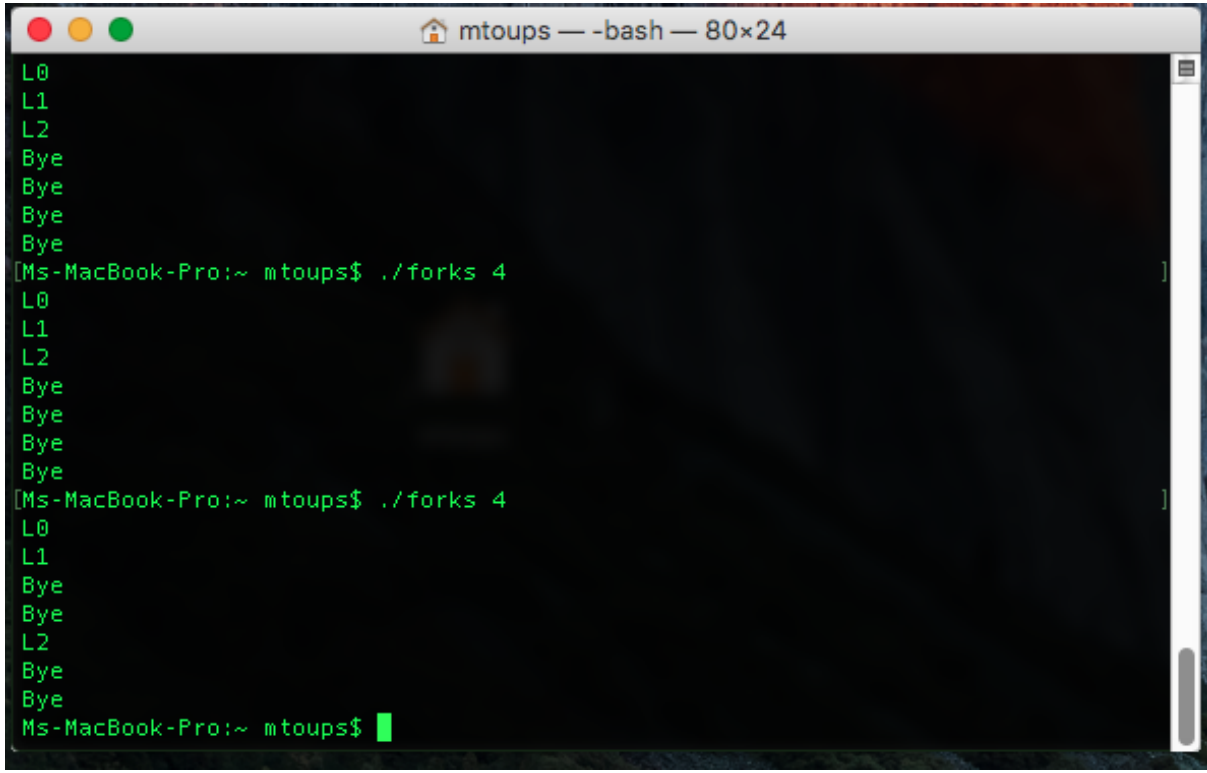
L0
L1
Bye
Bye
L2
Bye

Infeasible output:

L0
Bye
L1
Bye
Bye
L2

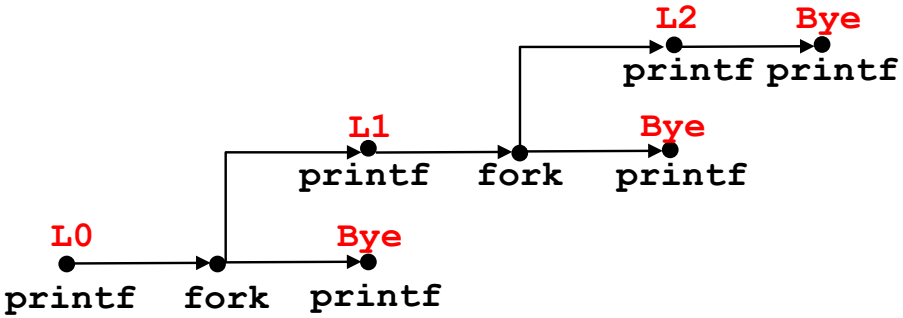
`fork()` example: `./forks 4` has multiple feasible orderings

```
mtoups@x1ubuntu:/tmp$ ./forks 4
L0
L1
L2
Bye
Bye
Bye
Bye
mtoups@x1ubuntu:/tmp$ ./forks 4
L0
L1
Bye
L2
Bye
Bye
Bye
mtoups@x1ubuntu:/tmp$
```



fork() example: nested fork()s in children

```c
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

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)
            ; /* 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

```c
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 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

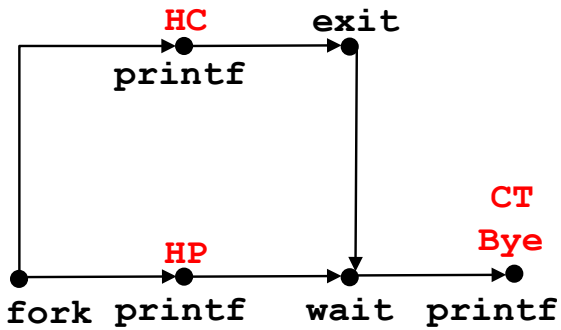
- **Parent reaps a child by calling the `wait` function**

- **`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

```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:

HC
HP
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

```c
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)

```c
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);
    }
}
```

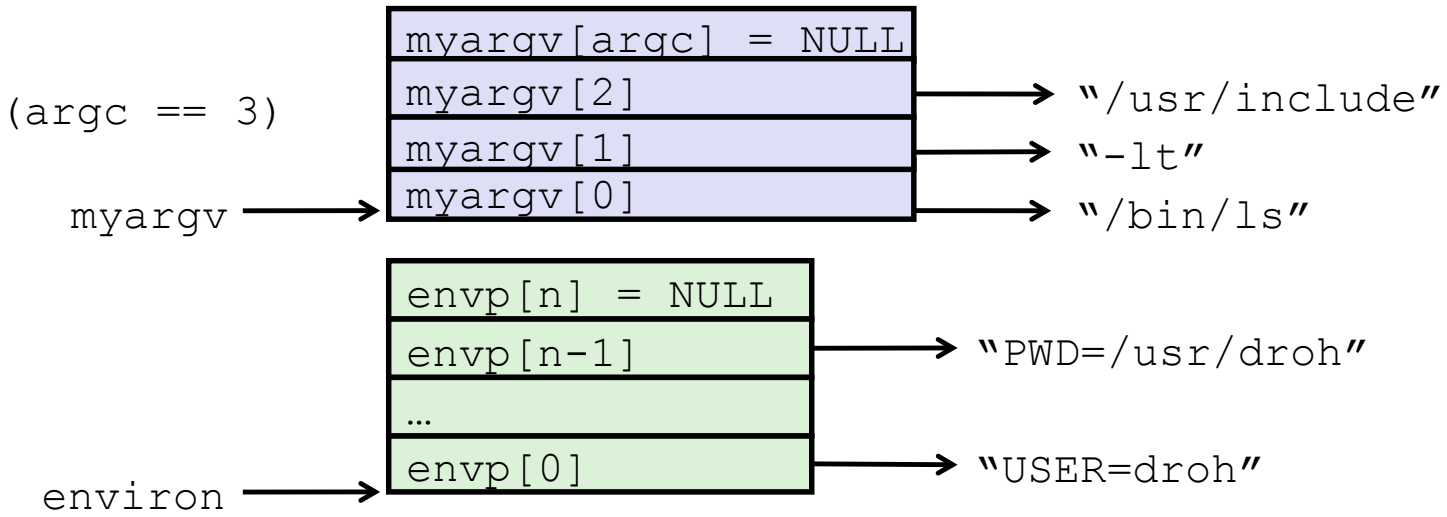
forks.c

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
- **Called once and never returns**
  - …except if there is an error

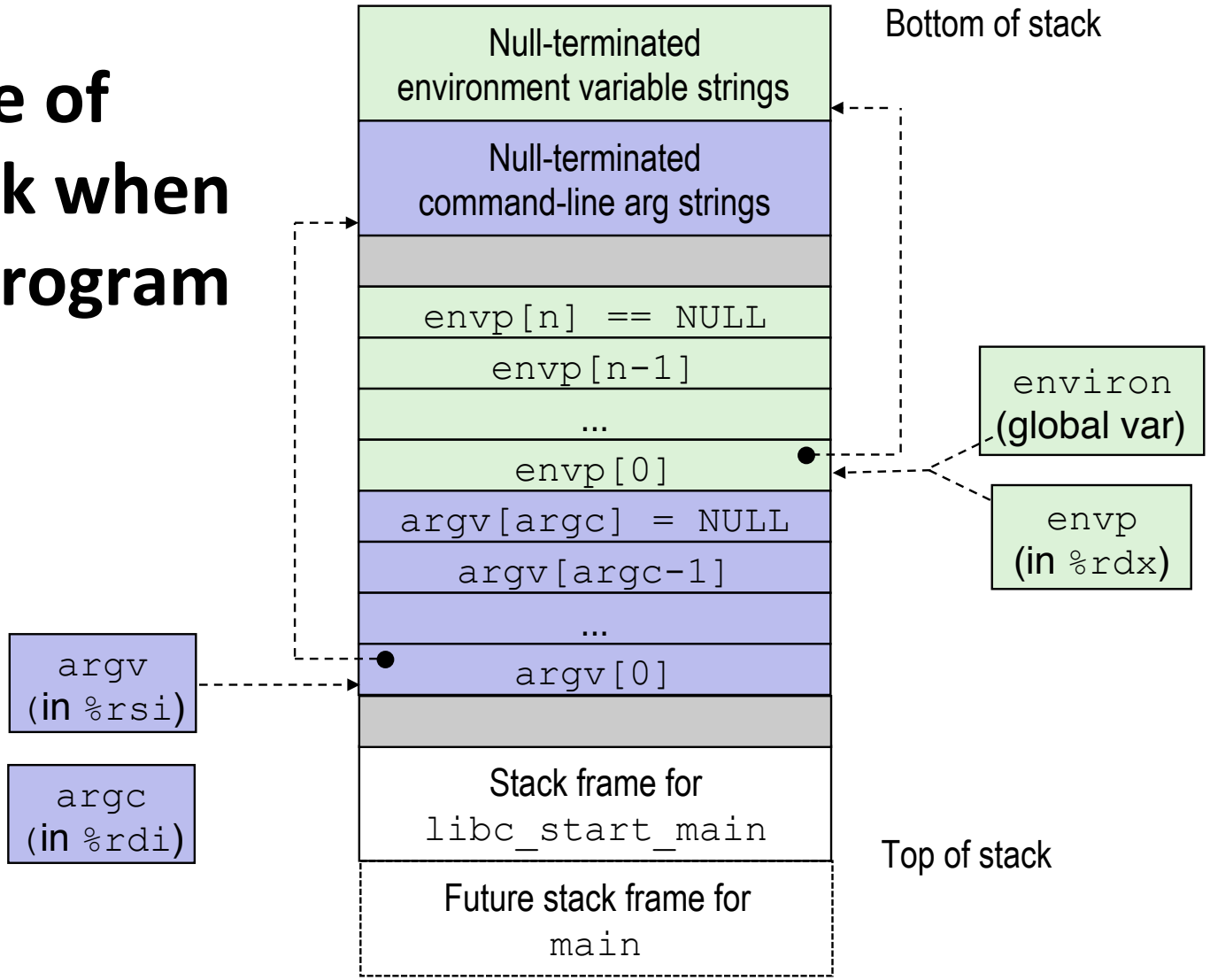
execve() example

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




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

Structure of the stack when a new program starts



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 wrapper functions

Info on the textbook's shorthand

- **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();
```

System Calls

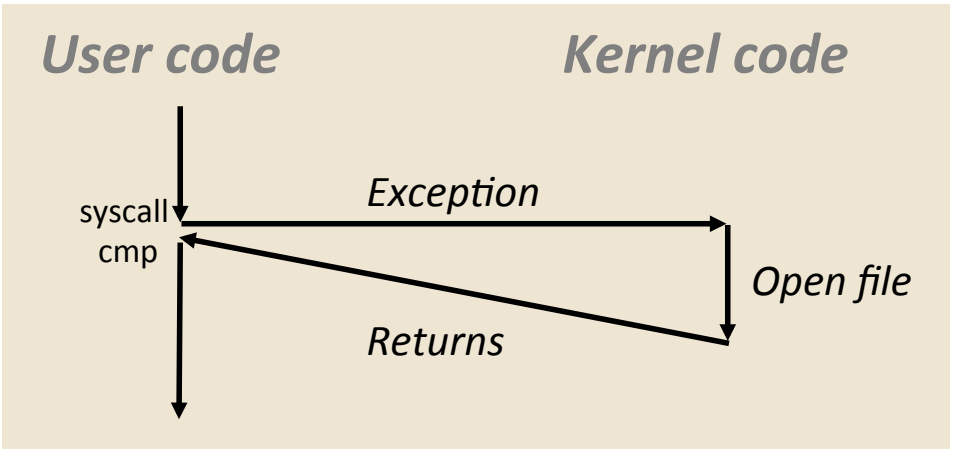
- **Each x86-64 system call has a unique ID number**
- **Examples:**

Number	*Name*	*Description*
0	`read`	Read file
1	`write`	Write file
2	`open`	Open file
3	`close`	Close file
4	`stat`	Get info about file
57	`fork`	Create process
59	`execve`	Execute a program
60	`_exit`	Terminate process
62	`kill`	Send signal to process

System Call example: opening file

- User calls: **`open(filename, options)`**
- Calls **`__open`** function, which invokes system call instruction **`syscall`**

```
0000000000e5d70 <__open>:
...
e5d79:   b8 02 00 00 00      mov  $0x2,%eax  # open is syscall #2
e5d7e:   0f 05               syscall         # Return value in %rax
e5d80:   48 3d 01 f0 ff ff   cmp  $0xfffffffffffff001,%rax
...
e5dfa:   c3                  retq
```



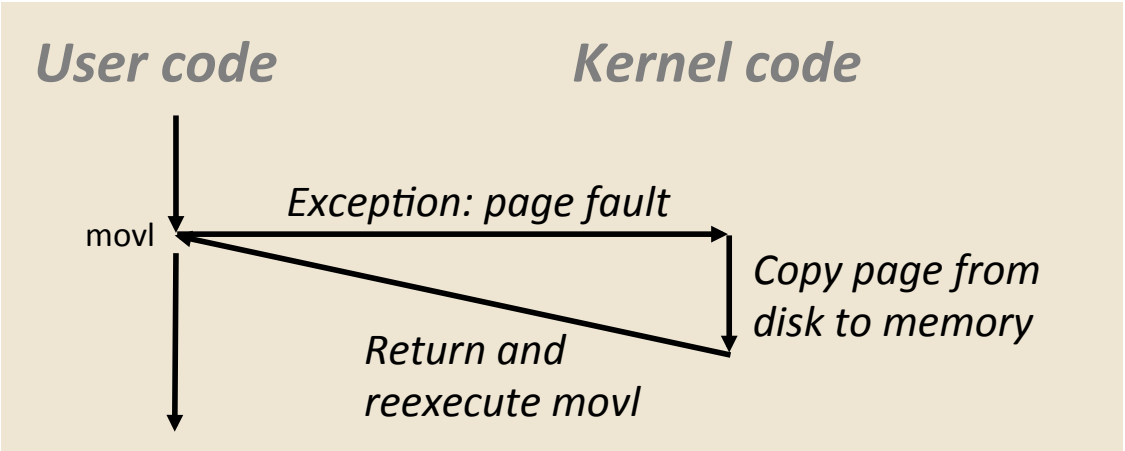
- `%rax` contains syscall number
- Other arguments in `%rdi`, `%rsi`, `%rdx`, `%r10`, `%r8`, `%r9`
- Return value in `%rax`
- Negative value is an error corresponding to negative `errno`

Fault example (recoverable): page fault

- User writes to memory location
- That portion (page) of user's memory is currently on disk

```
int a[1000];
main ()
{
    a[500] = 13;
}
```

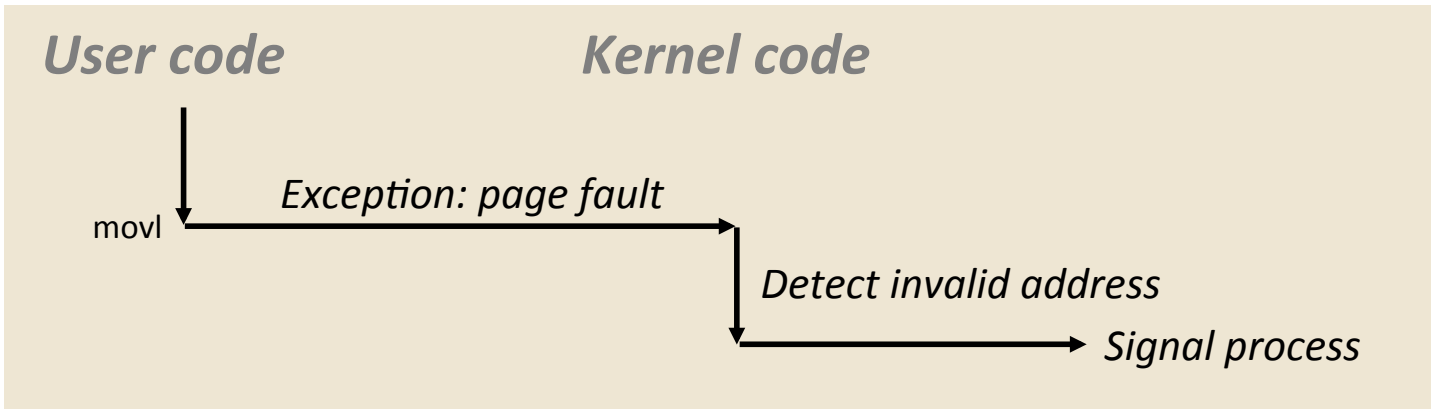
```
80483b7:      c7 05 10 9d 04 08 0d   movl   $0xd,0x8049d10
```



Fault example (unrecoverable): invalid memory reference

```
int a[1000];
main ()
{
    a[5000] = 13;
}
```

```
80483b7:      c7 05 60 e3 04 08 0d   movl   $0xd,0x804e360
```



- Sends **`SIGSEGV`** signal to user process
- User process exits with “segmentation fault”

Summary of exceptions

Class	Cause		Return behavior
Interrupt	Signal from I/O device	Async	Always returns to *next* instruction
Trap	Intentional exception	Sync	Always returns to *next* instruction
Fault	Potentially recoverable error	Sync	Might return to *current* instruction
Abort	Nonrecoverable error	Sync	Never returns

Figure 8.4 in CS:APP3e (p.726)

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 a single core, though
  - Each process appears to have total control of processor + private memory space

Summary (continued)

- **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