

Exceptional Control Flow: Exceptions and Processes

15-213 : Introduction to Computer Systems
13th Lecture, Jun 23, 2015

Instructors:

nwf and Greg Kesden

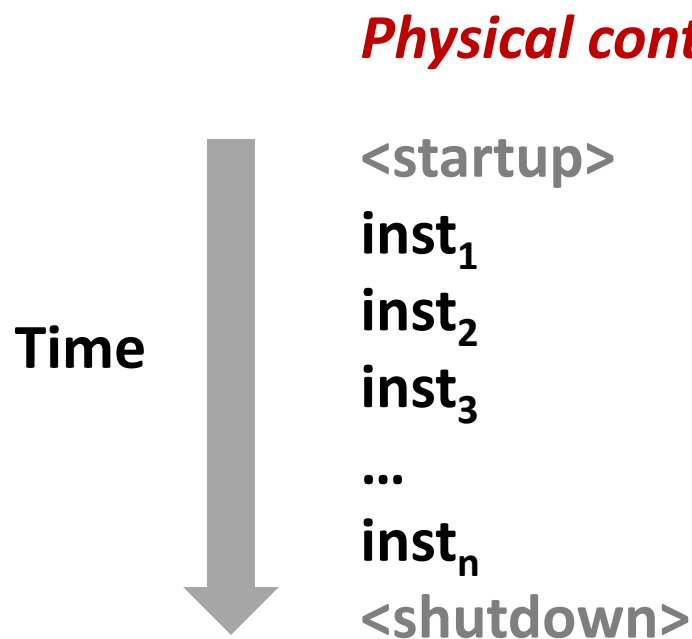
Today

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

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

Both 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

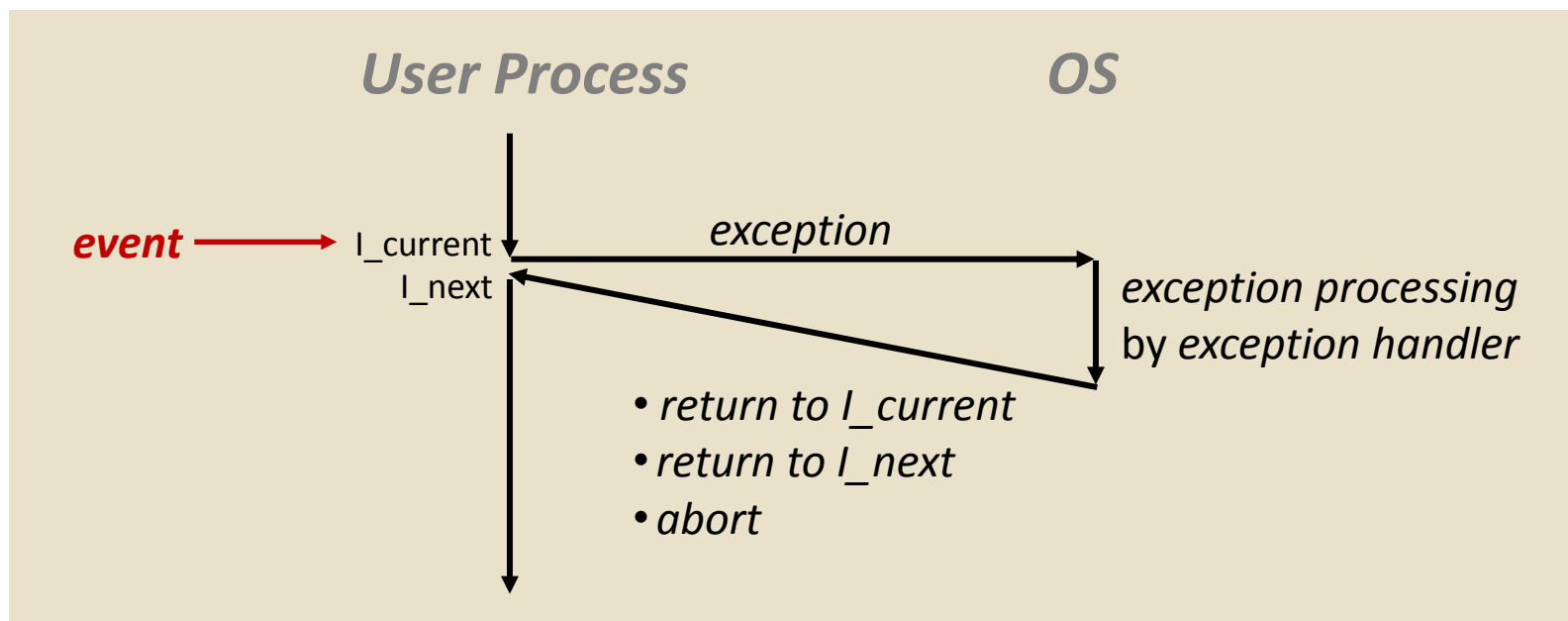
- **Exists at all levels of a computer system**
- **Low level mechanisms**
 - Exceptions (Intel taxonomy: traps, faults, interrupts, aborts)
 - change in control flow in response to a system event (i.e., change in system state)
 - Combination of hardware and OS software
- **Higher level mechanisms**
 - Nonlocal jumps: `setjmp()/longjmp()`
 - Process context switch
 - Signals
 - Implemented by either:
 - OS software (context switch and signals)
 - C language runtime library (nonlocal jumps)

Today

- Exceptional Control Flow
- Exceptions
- Processes

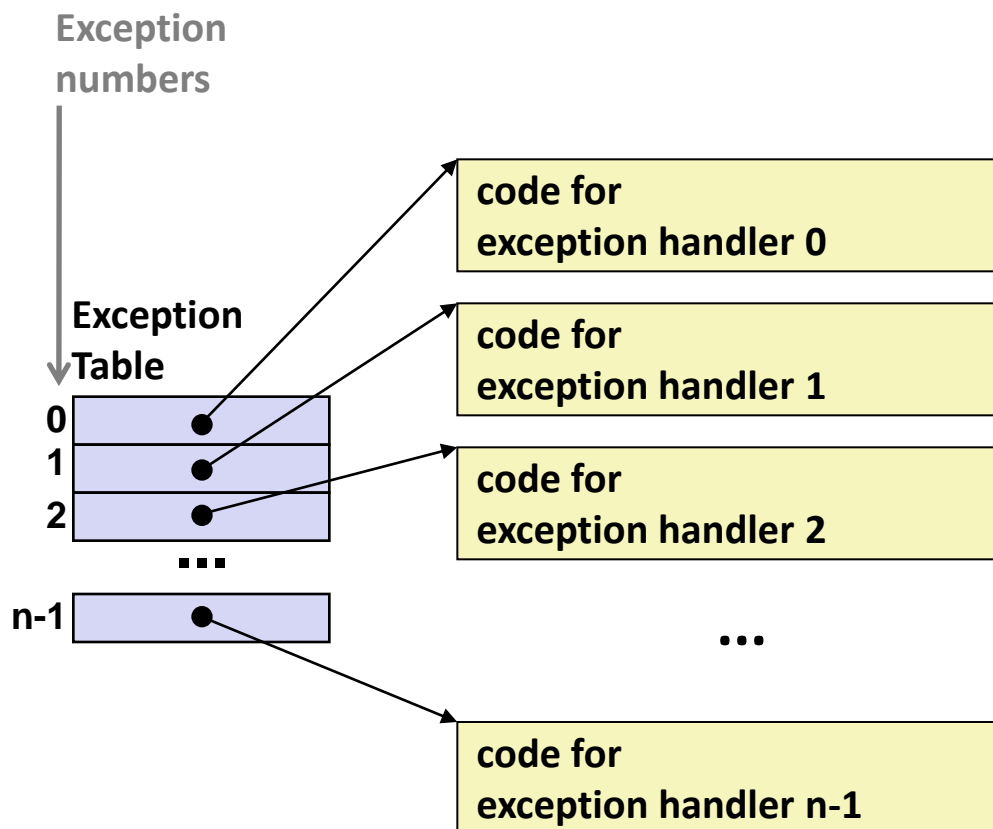
Exceptions

- An **exception** is a transfer of control to the OS kernel in response to some *event* (i.e., change in processor state)



- **Examples:**
div by 0, arithmetic overflow, page fault, I/O request completes, Ctrl-C

Exception Tables



- Each type of event has a unique exception number k
- k = index into exception table (a.k.a. interrupt vector)
- Handler k is called each time exception k occurs

Asynchronous Exceptions (**Interrupts**)

■ Caused by events external to the processor

- Indicated by setting the processor's interrupt pin
- Handler returns to "next" instruction

■ Examples:

- I/O interrupts
 - hitting Ctrl-C at the keyboard
 - arrival of a packet from a network
 - arrival of data from a disk
- Hard reset interrupt
 - hitting the power button
- Soft reset interrupt
 - hitting Ctrl-Alt-Delete on a PC

Synchronous Exceptions

- Caused by events that occur as a result of executing an instruction:

- **Traps**

- Intentional
- Examples: **system calls**, breakpoint traps, special instructions
- Returns control to “next” instruction

- **Faults**

- Unintentional but possibly recoverable
- Examples: page faults (recoverable), protection faults (usually unrecoverable), floating point exceptions
- Either re-executes faulting (“current”) instruction or aborts

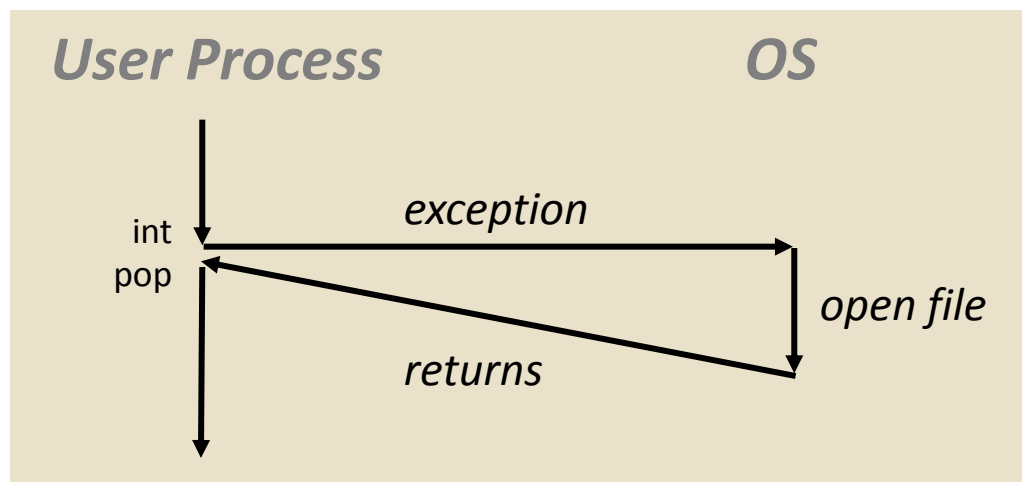
- **Aborts**

- unintentional and unrecoverable
- Examples: parity error, machine check
- Aborts current program

Trap Example: Opening File

- User calls: `open(filename, options)`
- Function `open` executes system call instruction `int`

```
0804d070 <__libc_open>:  
. . .  
804d082:      cd 80          int    $0x80  
804d084:      5b              pop    %ebx  
. . .
```



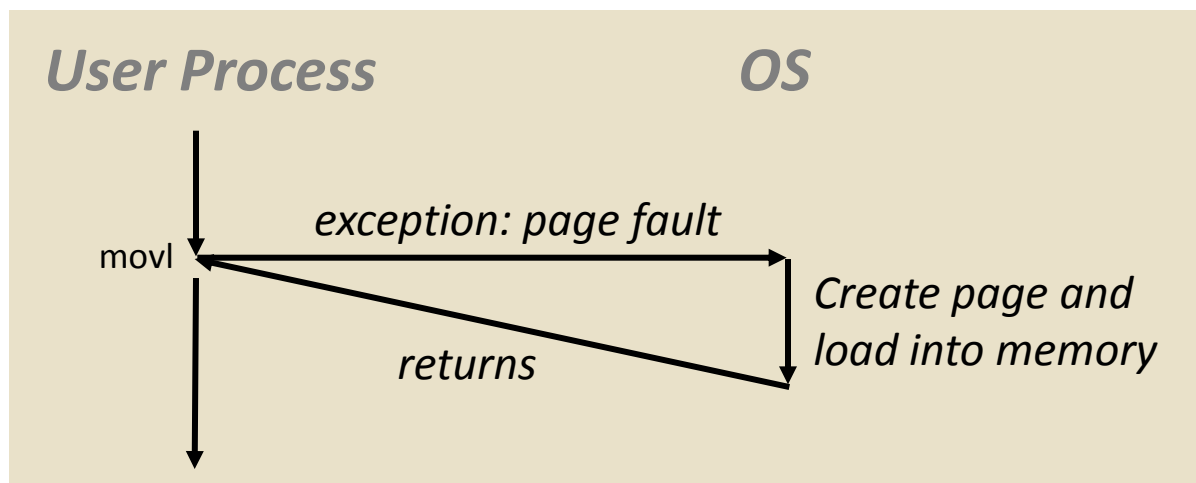
- OS must find or create file, get it ready for reading or writing
- Returns integer file descriptor

Fault Example: 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
----------	----------------------	------	-----------------

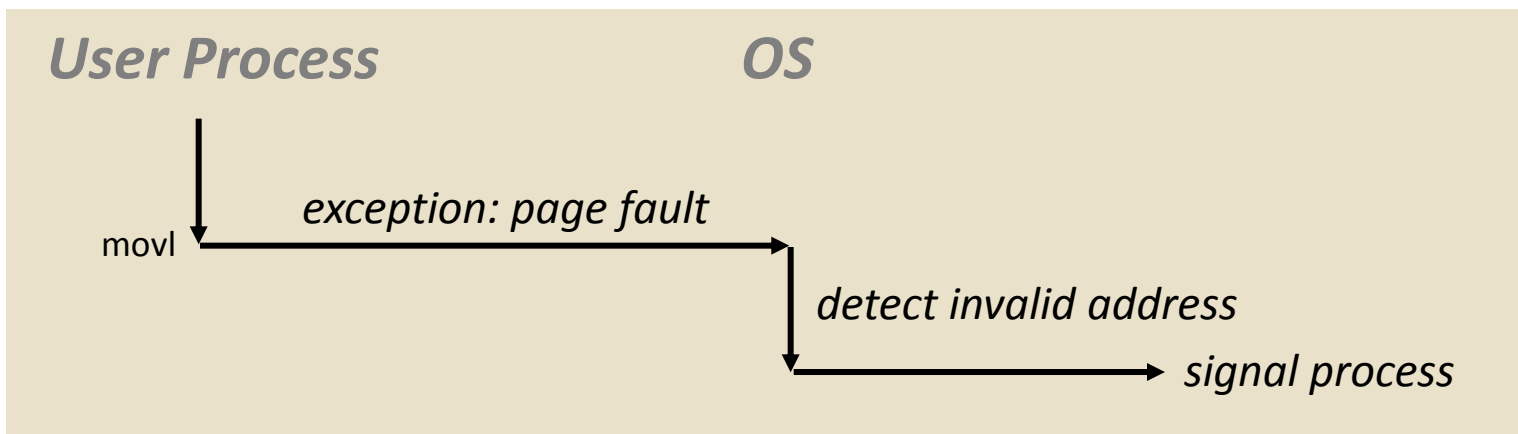


- Page handler must load page into physical memory
- Returns to faulting instruction
- Successful on second try

Fault Example: Invalid Memory Reference

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

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



- Page handler detects invalid address
- Sends **SIGSEGV** signal to user process
- User process exits with “segmentation fault”

Exception Table IA32 (Excerpt)

<i>Exception Number</i>	<i>Description</i>	<i>Exception Class</i>
0	Divide error	Fault
13	General protection fault	Fault
14	Page fault	Fault
18	Machine check	Abort
32-255	OS-defined	Interrupt or trap

Linux (IA32) uses vector 128 (0x80) for its system calls,
and used to map the 16 hardware interrupts to vectors 32-47.
(This story is more complicated now, sorry.)

Check Table 6-1:

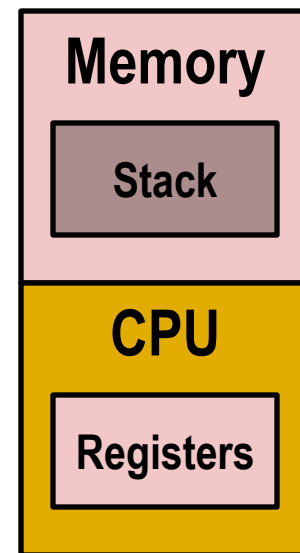
<http://download.intel.com/design/processor/manuals/253665.pdf>

Today

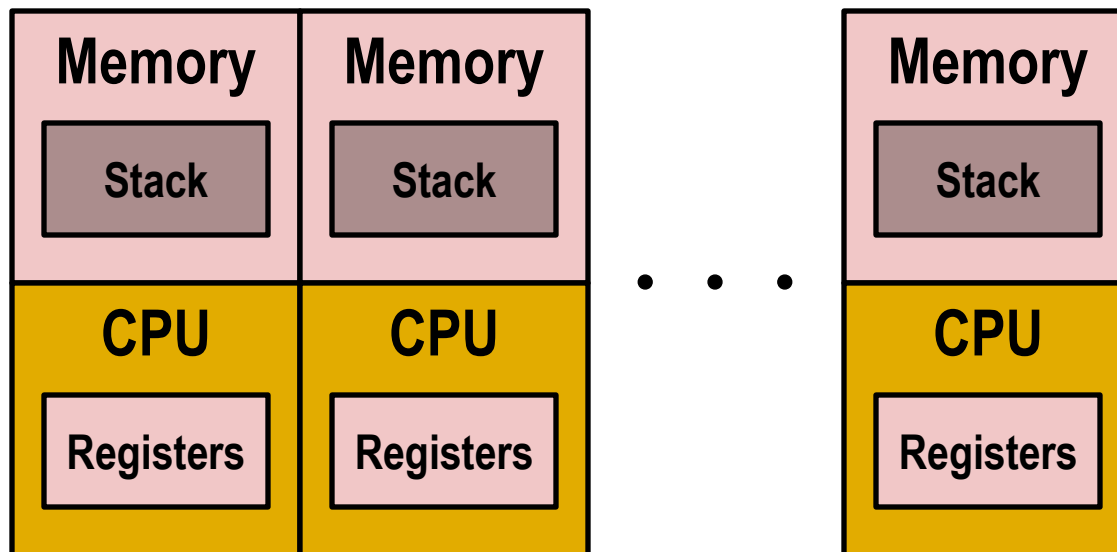
- Exceptional Control Flow
- Exceptions
- **Processes**

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
 - Private copy of program state
 - Register values (PC, stack pointer, general registers, condition codes)
 - Private virtual address space
 - Program has exclusive access to main memory
 - Including stack



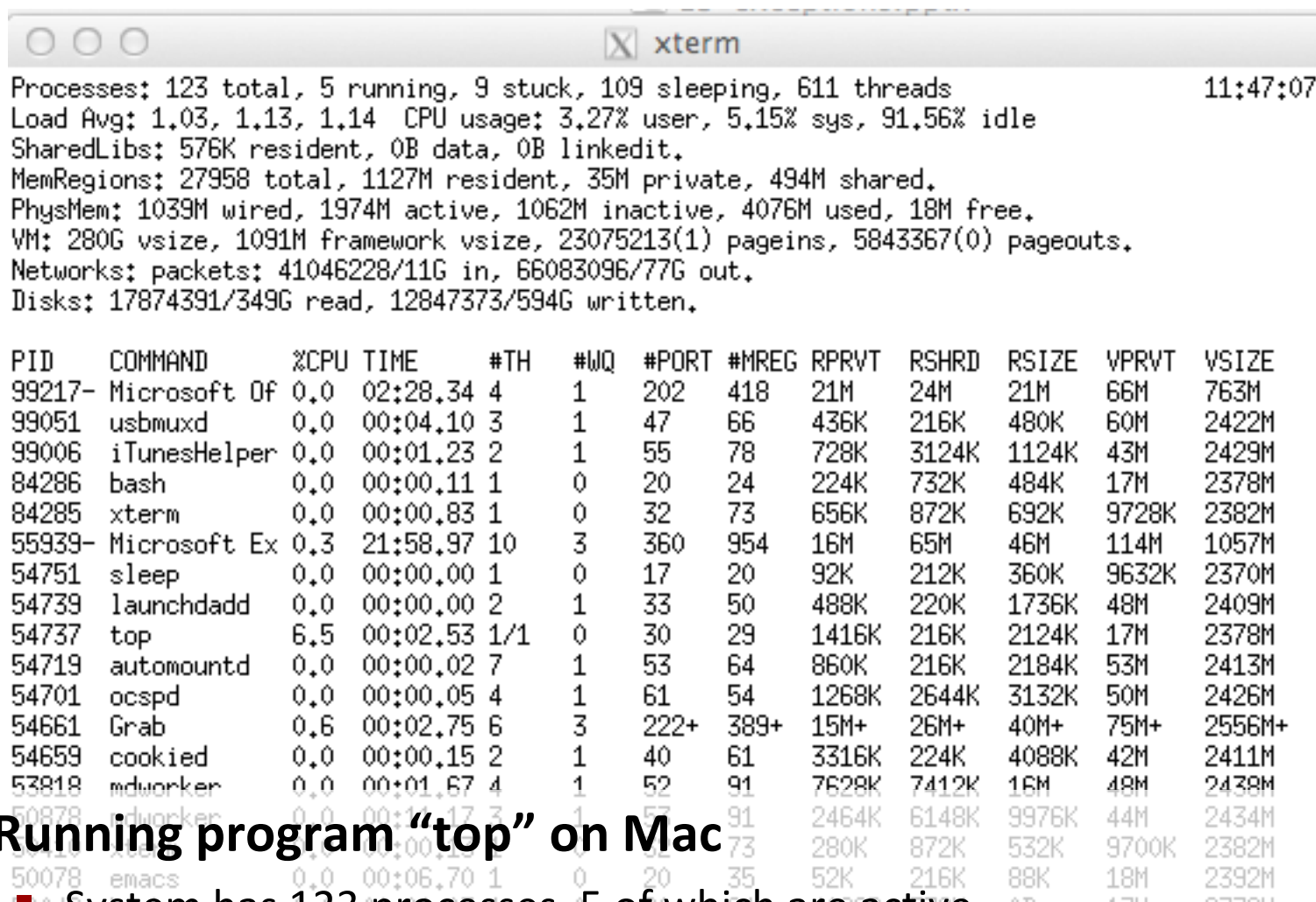
Multiprocessing: The Illusion



■ 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



```

Processes: 123 total, 5 running, 9 stuck, 109 sleeping, 611 threads
Load Avg: 1.03, 1.13, 1.14  CPU usage: 3.27% user, 5.15% sys, 91.56% idle
SharedLibs: 576K resident, 0B data, 0B linkedit.
MemRegions: 27958 total, 1127M resident, 35M private, 494M shared.
PhysMem: 1039M wired, 1974M active, 1062M inactive, 4076M used, 18M free.
VM: 280G vsize, 1091M framework vsize, 23075213(1) pageins, 5843367(0) pageouts.
Networks: packets: 41046228/11G in, 66083096/77G out.
Disks: 17874391/349G read, 12847373/594G written.

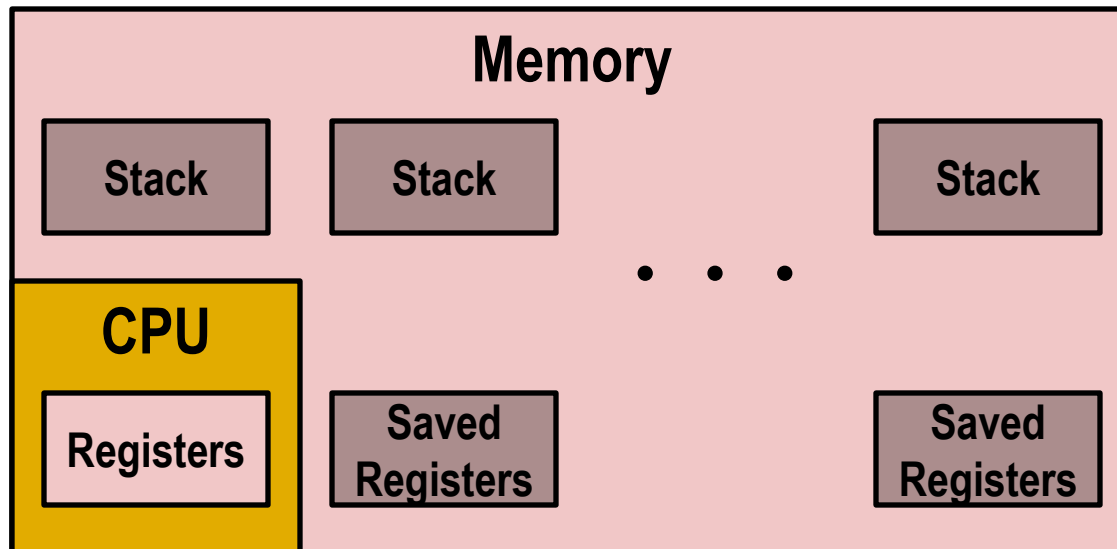
PID    COMMAND    %CPU TIME    #TH    #WQ    #PORT    #MREG    RPRVT    RSHRD    RSIZE    VPRVT    VSIZE
99217-  Microsoft Of 0.0 02:28.34 4      1      202     418     21M     24M     21M     66M     763M
99051  usbmuxd     0.0 00:04.10 3      1      47      66      436K    216K    480K    60M     2422M
99006  iTunesHelper 0.0 00:01.23 2      1      55      78      728K    3124K   1124K   43M     2429M
84286  bash        0.0 00:00.11 1      0      20      24      224K    732K    484K    17M     2378M
84285  xterm       0.0 00:00.83 1      0      32      73      656K    872K    692K    9728K   2382M
55939-  Microsoft Ex 0.3 21:58.97 10     3      360     954     16M     65M     46M     114M    1057M
54751  sleep       0.0 00:00.00 1      0      17      20      92K     212K    360K    9632K   2370M
54739  launchdadd  0.0 00:00.00 2      1      33      50      488K    220K    1736K   48M     2409M
54737  top         6.5 00:02.53 1/1    0      30      29      1416K   216K    2124K   17M     2378M
54719  automountd  0.0 00:00.02 7      1      53      64      860K    216K    2184K   53M     2413M
54701  ocsdpd      0.0 00:00.05 4      1      61      54      1268K   2644K   3132K   50M     2426M
54661  Grab        0.6 00:02.75 6      3      222+    389+    15M+    26M+    40M+    75M+    2556M+
54659  cookied     0.0 00:00.15 2      1      40      61      3316K   224K    4088K   42M     2411M
53818  mdworker    0.0 00:01.67 4      1      52      91      7628K   7412K   16M     48M     2438M
50878  mdworker    0.0 00:01.17 3      1      57      91      2464K   6148K   9976K   44M     2434M
50078  emacs       0.0 00:06.70 1      0      20      35      52K     216K    88K     18M     2392M

```

■ Running program “top” on Mac

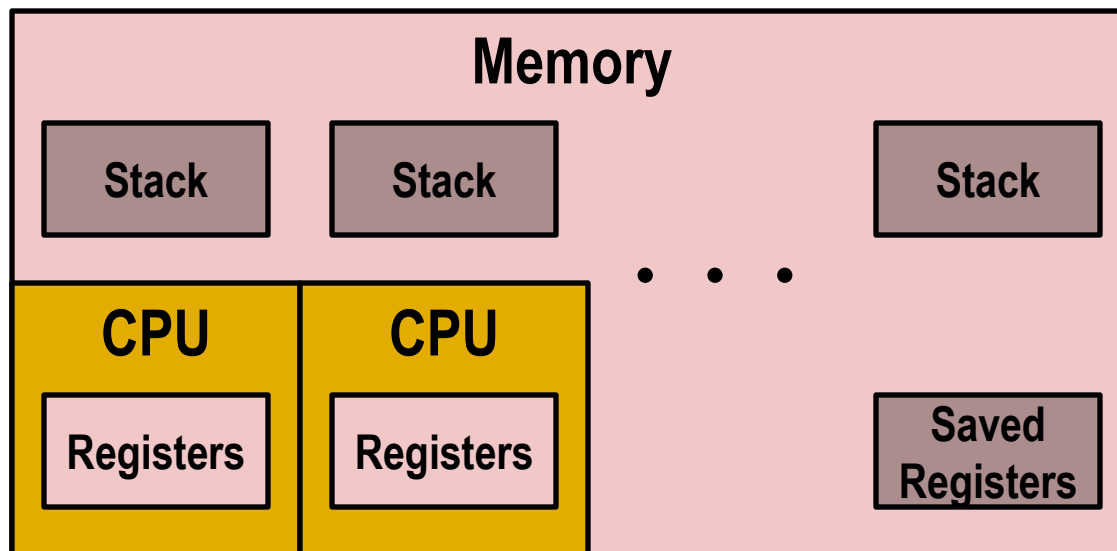
- System has 123 processes, 5 of which are active
- Identified by OS-managed Process ID (PID)

Multiprocessing: The Single-Core Reality



- **Single Processor Executes Multiple Processes Concurrently**
 - Process executions interleaved (multitasking)
 - Address spaces managed by virtual memory system
 - we'll talk about this in a couple of weeks
 - Register values for nonexecuting processes saved in memory

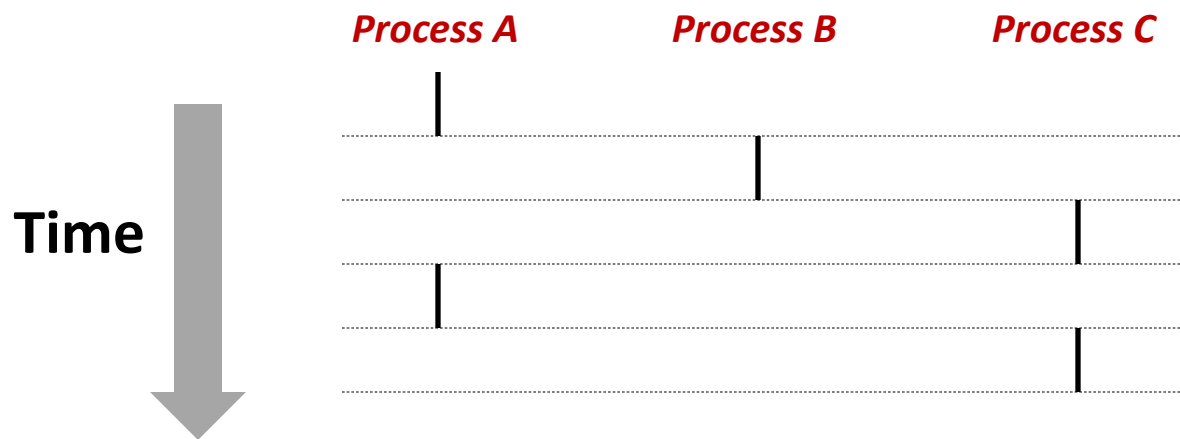
Multiprocessing: The Multiple CPU Reality



- **Systems can have multiple CPU sockets**
- **These days, have multiple CPU cores per socket**
 - Multiple CPUs on single chip
 - Share main memory (and some of the caches)
 - Each *core* can execute a separate process
 - Scheduling of processes onto sockets and cores done by OS

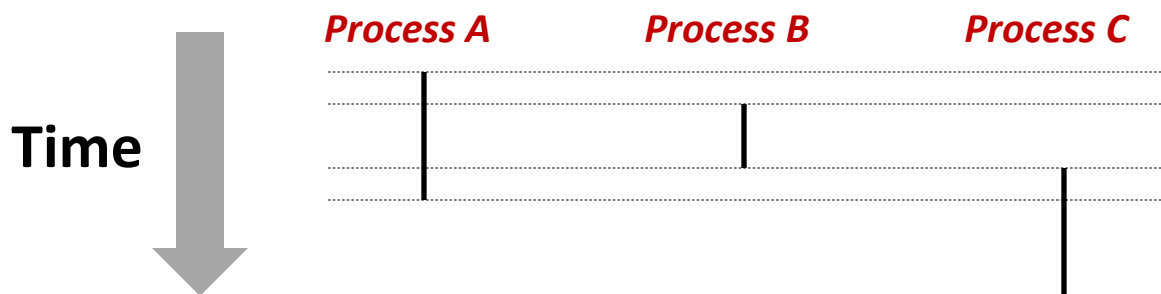
Concurrent Processes

- 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



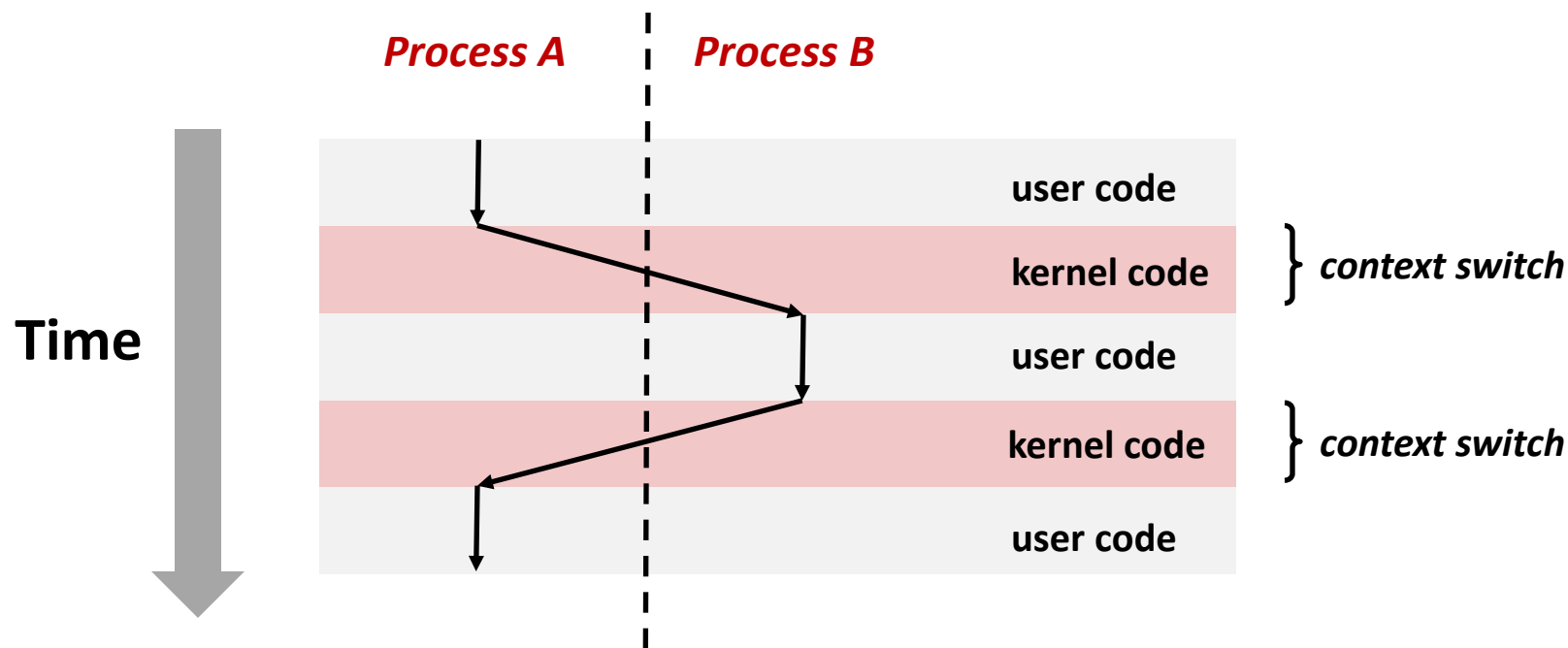
User View of Concurrent Processes

- 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 user process
- Control flow passes from one process to another via a *context switch*



Programmer's Model of Multitasking

■ Basic functions

- **fork** spawns new process
 - Called once, returns twice
- **execve** runs new program in existing process
 - Called once, (normally) never returns
- **exit** terminates own process
 - Called once, never returns
 - Puts it into “zombie” status
- **wait** and **waitpid** wait for and reap terminated children

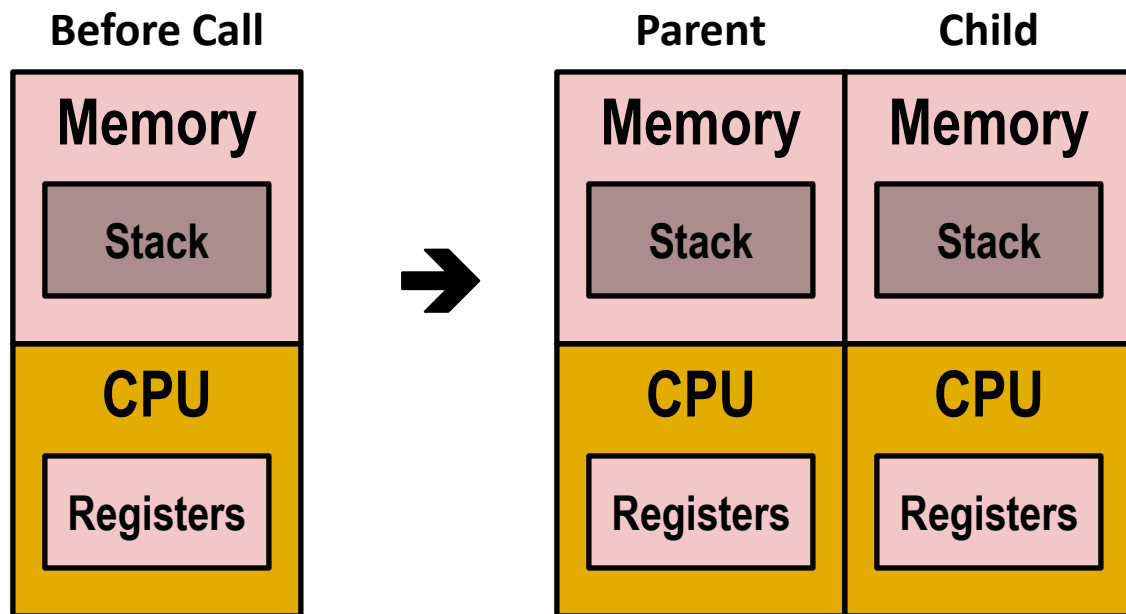
■ Programming challenge

- Understanding the nonstandard semantics of the functions
- Avoiding improper use of system resources
 - E.g. “Fork bombs” can disable a system

fork: Creating New Processes

■ `int fork(void)`

- creates a new process (child process) that is identical to the calling process (parent process)
- (Appears to) create complete new copy of program state
- Child & parent then execute as independent processes
 - Writes by one don't affect reads by other
 - But ... share any open files



fork Details

■ `int fork(void)`

- creates a new process (child process) that is identical to the calling process (parent process)
- returns 0 to the child process
- returns child's **pid** (process id) to the parent process

```
pid_t pid = fork();  
if (pid == 0) {  
    printf("hello from child\n");  
} else {  
    printf("hello from parent\n");  
}
```

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

Understanding fork

Process n



```
pid_t pid = fork();
if (pid == 0) {
    printf("hello from child\n");
} else {
    printf("hello from parent\n");
}
```



pid = m

```
pid_t pid = fork();
if (pid == 0) {
    printf("hello from child\n");
} else {
    printf("hello from parent\n");
}
```



```
pid_t pid = fork();
if (pid == 0) {
    printf("hello from child\n");
} else {
    printf("hello from parent\n");
}
```

Child Process m



```
pid_t pid = fork();
if (pid == 0) {
    printf("hello from child\n");
} else {
    printf("hello from parent\n");
}
```



pid = 0

```
pid_t pid = fork();
if (pid == 0) {
    printf("hello from child\n");
} else {
    printf("hello from parent\n");
}
```



```
pid_t pid = fork();
if (pid == 0) {
    printf("hello from child\n");
} else {
    printf("hello from parent\n");
}
```

hello from parent

Which one is first?

hello from child

Fork Example #1

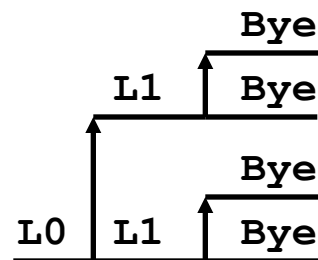
- **Parent and child both run same code**
 - Distinguish parent from child by return value from `fork`
- **Start with same state, but each has private copy**
 - Including shared output file descriptor
 - Relative ordering of their print statements undefined

```
void fork1()
{
    int x = 1;
    pid_t pid = fork();
    if (pid == 0) {
        printf("Child has x = %d\n", ++x);
    } else {
        printf("Parent has x = %d\n", --x);
    }
    printf("Bye from process %d with x = %d\n", getpid(), x);
}
```

Fork Example #2

■ Two consecutive forks

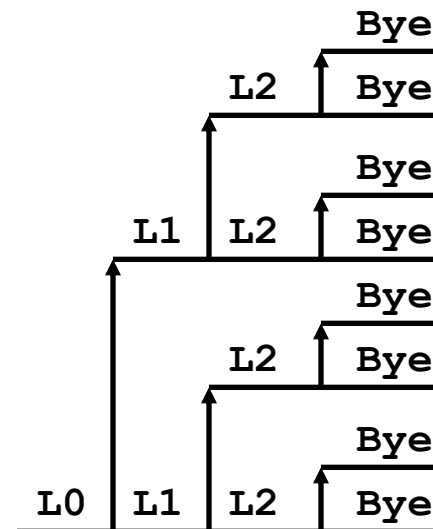
```
void fork2()  
{  
    printf("L0\n");  
    fork();  
    printf("L1\n");  
    fork();  
    printf("Bye\n");  
}
```



Fork Example #3

■ Three consecutive forks

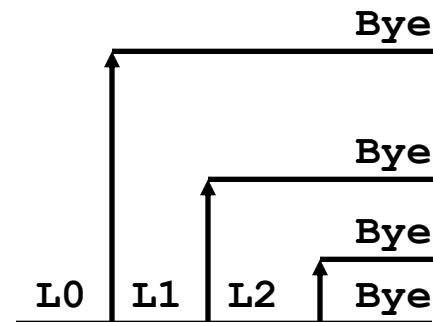
```
void fork3()
{
    printf("L0\n");
    fork();
    printf("L1\n");
    fork();
    printf("L2\n");
    fork();
    printf("Bye\n");
}
```



Fork Example #4

■ Nested forks in parent

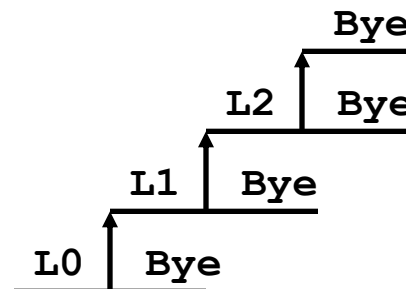
```
void fork4()  
{  
    printf("L0\n");  
    if (fork() != 0) {  
        printf("L1\n");  
        if (fork() != 0) {  
            printf("L2\n");  
            fork();  
        }  
    }  
    printf("Bye\n");  
}
```



Fork Example #5

■ Nested forks in children

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



exit: Ending a process

- `void exit(int status)`
 - exits a process
 - Normally return with status 0
 - `atexit()` registers functions to be executed upon exit

```
void cleanup(void) {  
    printf("cleaning up\n");  
}  
  
void fork6() {  
    atexit(cleanup);  
    fork();  
    exit(0);  
}
```

Zombies

■ Idea

- When process terminates, it still consumes system resources
 - 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 discards process

■ What if parent doesn't reap?

- If any parent terminates without reaping a child, then 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

```
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
```

```
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 */
    }
}
```

- **ps** shows child process as “defunct”
- Killing parent allows child to be reaped by **init**

Nonterminating Child Example

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

```
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

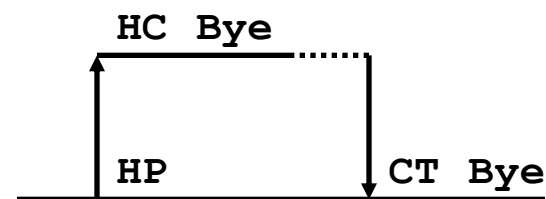
- Child process still active even though parent has terminated
- Must kill explicitly, or else will keep running indefinitely

`wait`: Synchronizing with Children

- Parent reaps 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 object it points to will be set to a status indicating why the child process terminated

wait: Synchronizing with Children

```
void fork9() {  
    int child_status;  
  
    if (fork() == 0) {  
        printf("HC: hello from child\n");  
    }  
    else {  
        printf("HP: hello from parent\n");  
        wait(&child_status);  
        printf("CT: child has terminated\n");  
    }  
    printf("Bye\n");  
    exit();  
}
```



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;  
    int child_status;  
    for (i = 0; i < N; i++)  
        if ((pid[i] = fork()) == 0)  
            exit(100+i); /* Child */  
    for (i = 0; i < N; i++) {  
        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);  
    }  
}
```

waitpid() : Waiting for a Specific Process

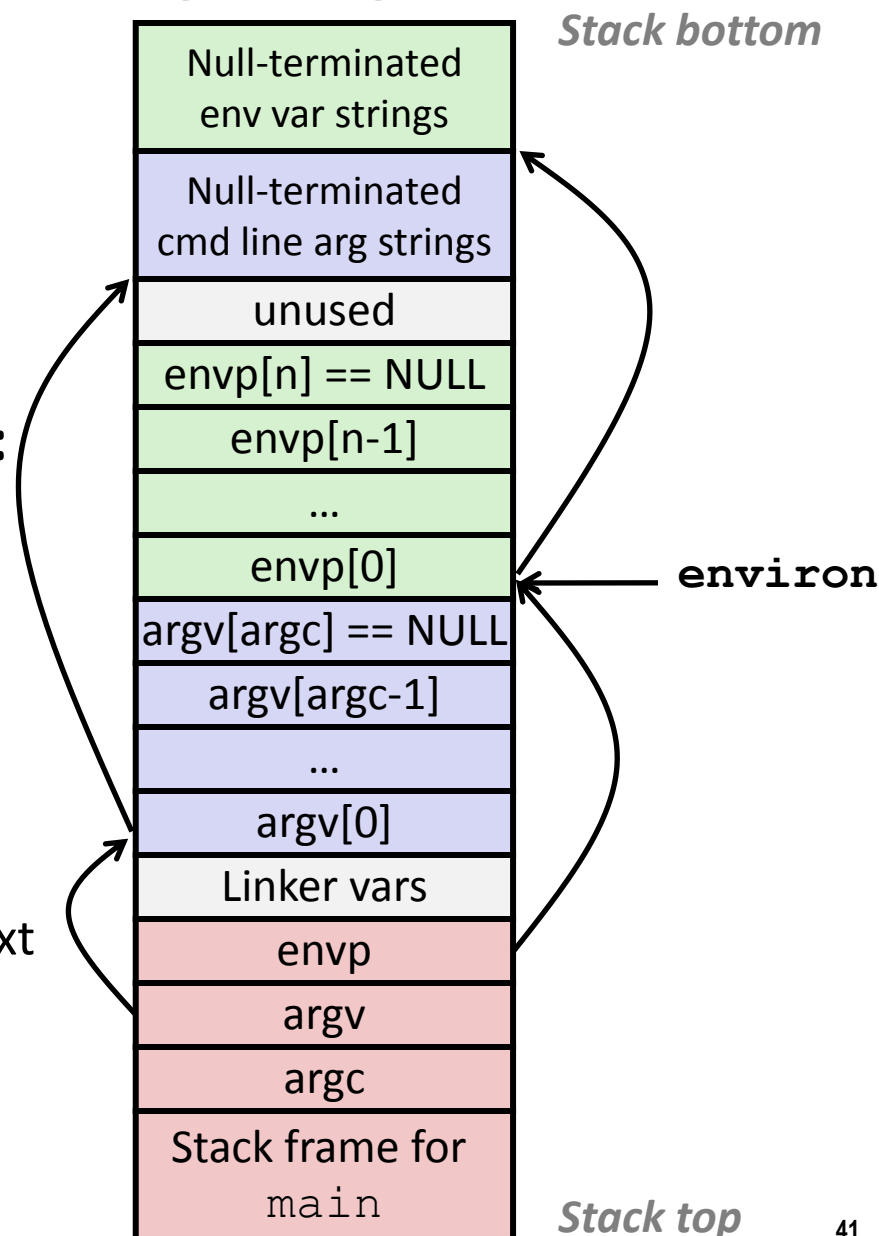
■ waitpid(pid, &status, options)

- 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 terminated abnormally\n", wpid);
    }
}
```

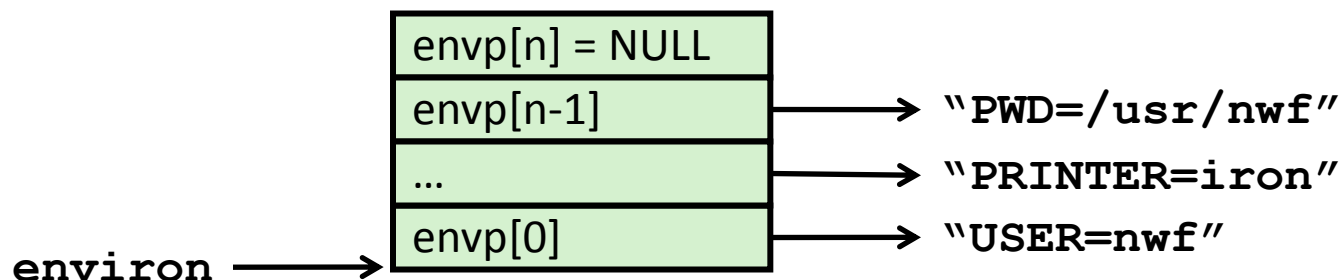
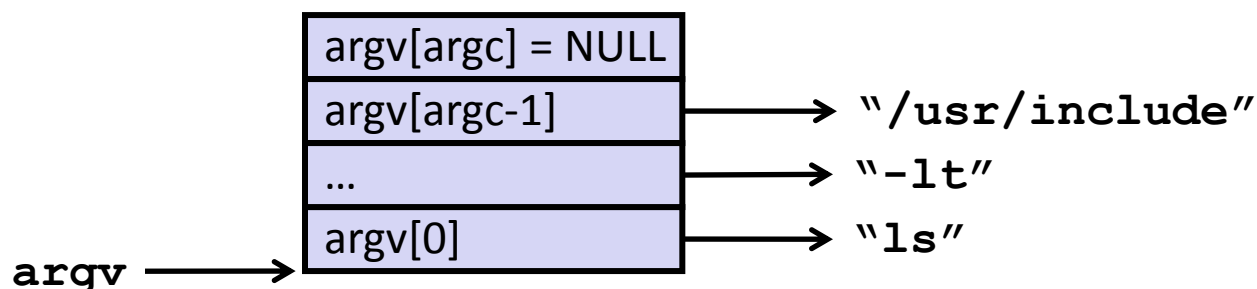

execve: Loading and Running Programs

- `int execve(`
 `char *filename,`
 `char *argv[],`
 `char *envp[]`
 `)`
- **Loads and runs in current process:**
 - Executable `filename`
 - With argument list `argv`
 - And environment variable list `envp`
- **Does not return (unless error)**
- **Overwrites code, data, and stack**
 - keeps pid, open files and signal context
- **Environment variables:**
 - “name=value” strings
 - `getenv` and `putenv`



execve Example

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



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 (cont.)

■ Spawning processes

- Call `fork`
- One call, two returns

■ Process completion

- Call `exit`
- One call, no return

■ Reaping and waiting for processes

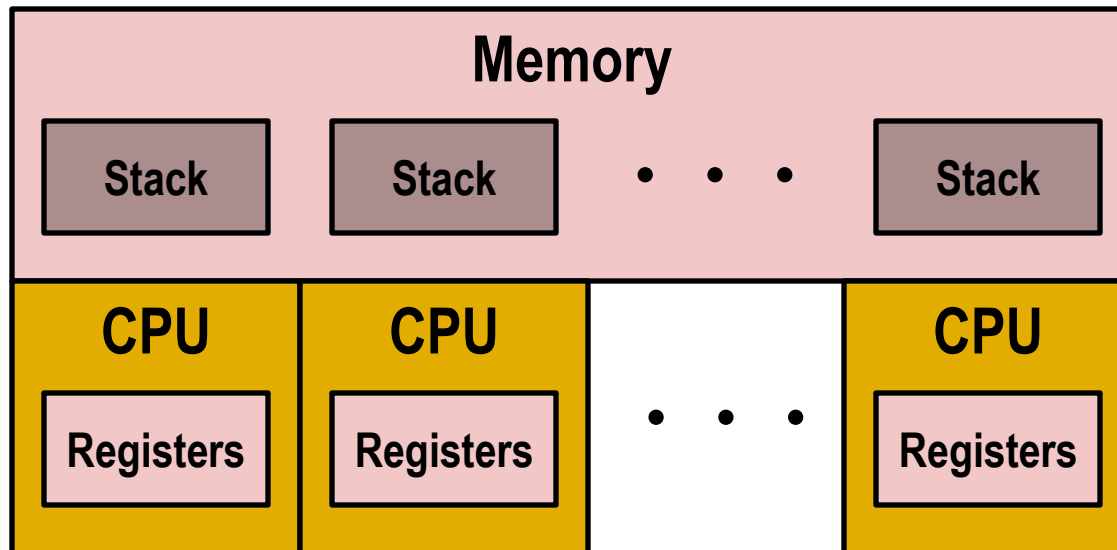
- Call `wait` or `waitpid`

■ Loading and running programs

- Call `execve` (or variant)
- One call, (normally) no return

Additional slides

Multithreading: The Illusion



- Single process runs multiple *threads* concurrently
- Each has own control flow and runtime state
 - But view memory as shared among all threads
 - One thread can read/write the state of another
- We will talk about this later in the term
 - For today, just consider one thread / process executing on single core