

Chapter 8: Exceptional Control Flow or

How a (shell) program interacts with the OS and I/O hardware

Useful for Shell Lab

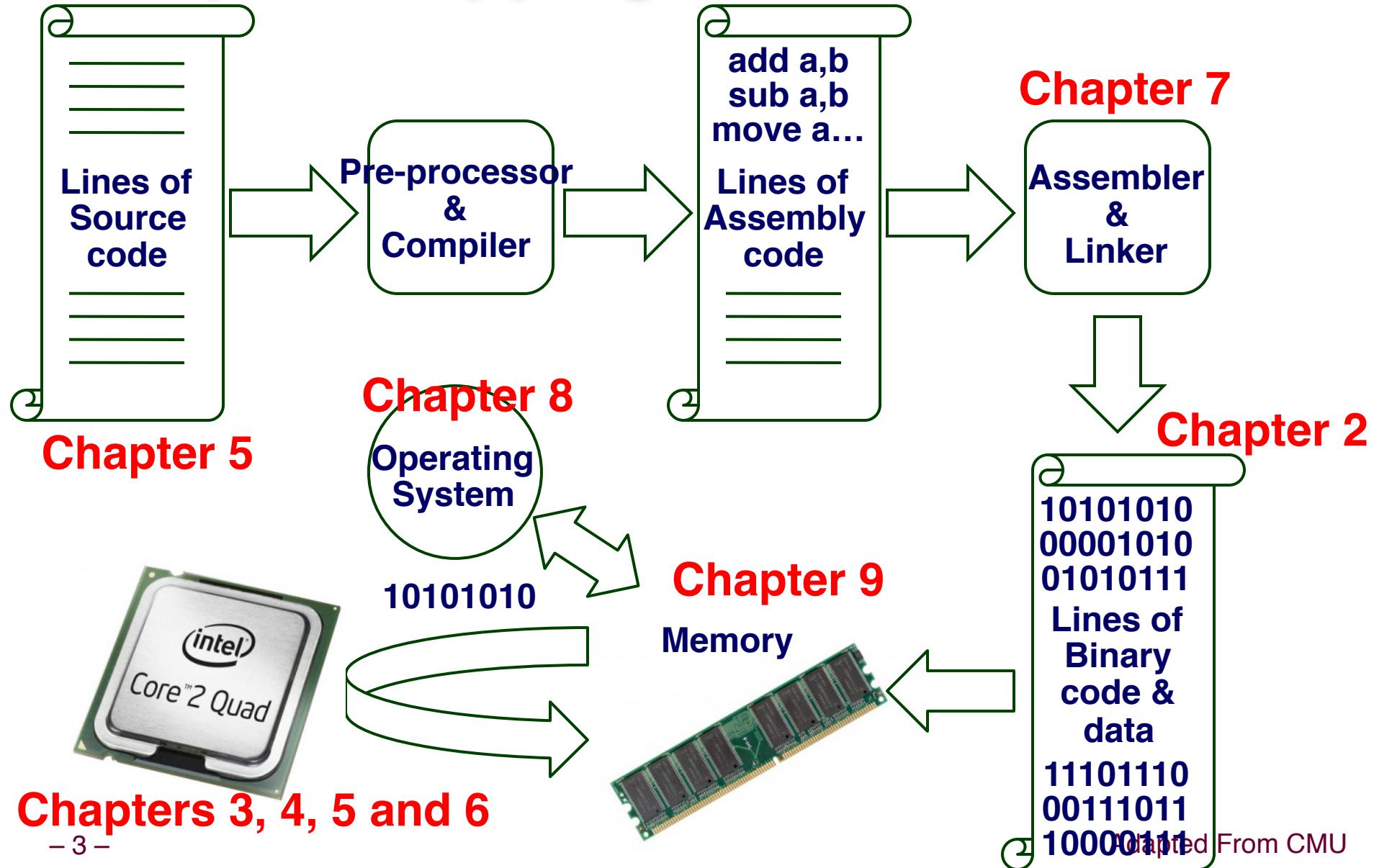
Topics

- Exceptions
- Creating and destroying processes

Announcements

- **Performance Lab grading this week**
 - Sign up for time slots
- **Shell Lab due after Fall Break, Monday Dec 8 by 8 am**
 - Introduced in recitation this week
- **Midterm #2 returned in TA office hours this Thursday & Friday**
 - Also handed back in recitation after Fall Break
- **Essential that you read the textbook in detail & do the practice problems**
 - Skip Chapter 7 for now, read Chapter 8 except 8.6 (no nonlocal jumps)

Chapter Mapping

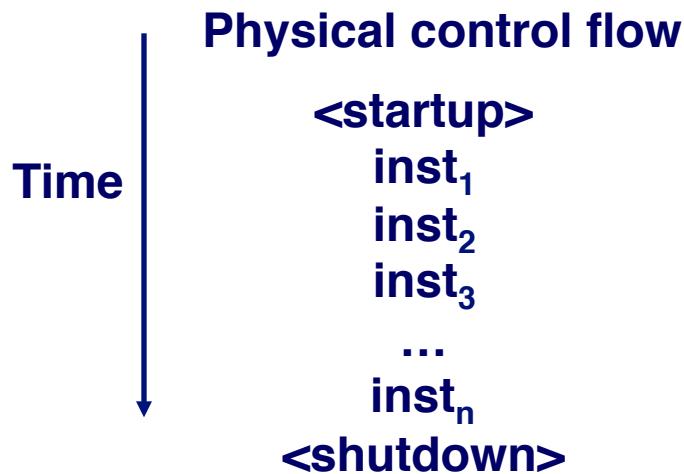


Chapters 3, 4, 5 and 6

Control Flow

Computers 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 system's physical *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 using the stack discipline.
- Both react to changes in program state.

Insufficient for a useful system

- Difficult for the CPU to react to changes in system state.
 - data arrives from a disk or a network adapter.
 - Instruction divides by zero
 - User hits `ctl-c` at the keyboard
 - System timer expires

System needs mechanisms for “exceptional control flow”

Exceptional Control Flow

- Mechanisms for exceptional control flow exists at all levels of a computer system.

Low level Mechanism

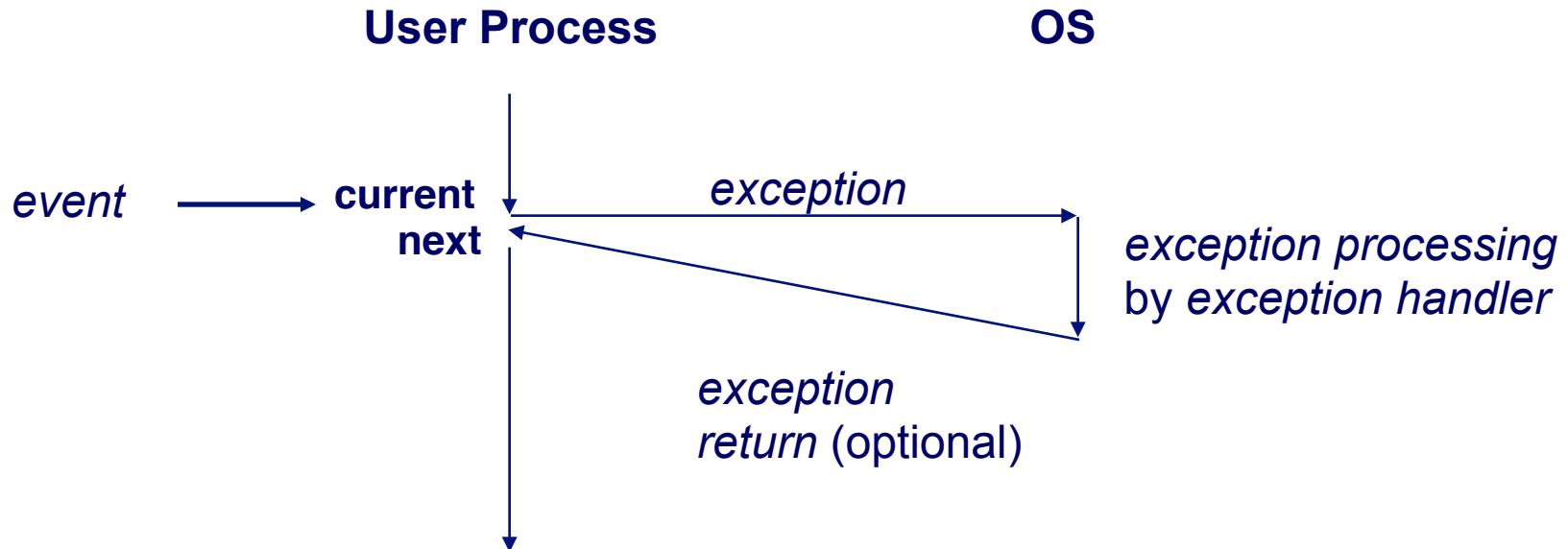
- exceptions
 - change in control flow in response to a system event (i.e., change in system state)
- Combination of hardware and OS software

Higher Level Mechanisms

- Process context switch
- Signals
- Nonlocal jumps (setjmp/longjmp), try / except blocks
- Implemented by either:
 - OS software (context switch and signals).
 - C language runtime library: nonlocal jumps.

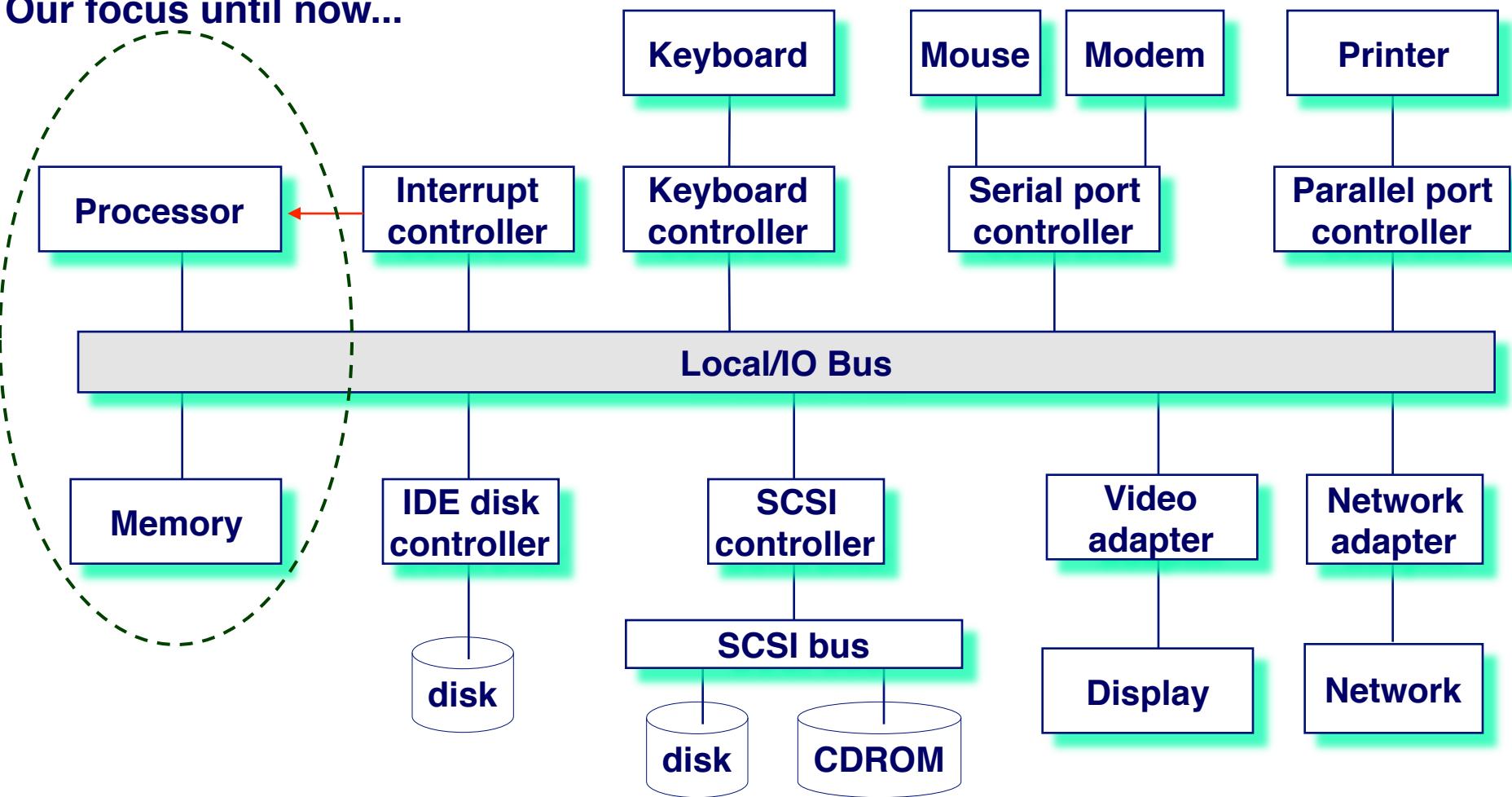
Exceptions

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



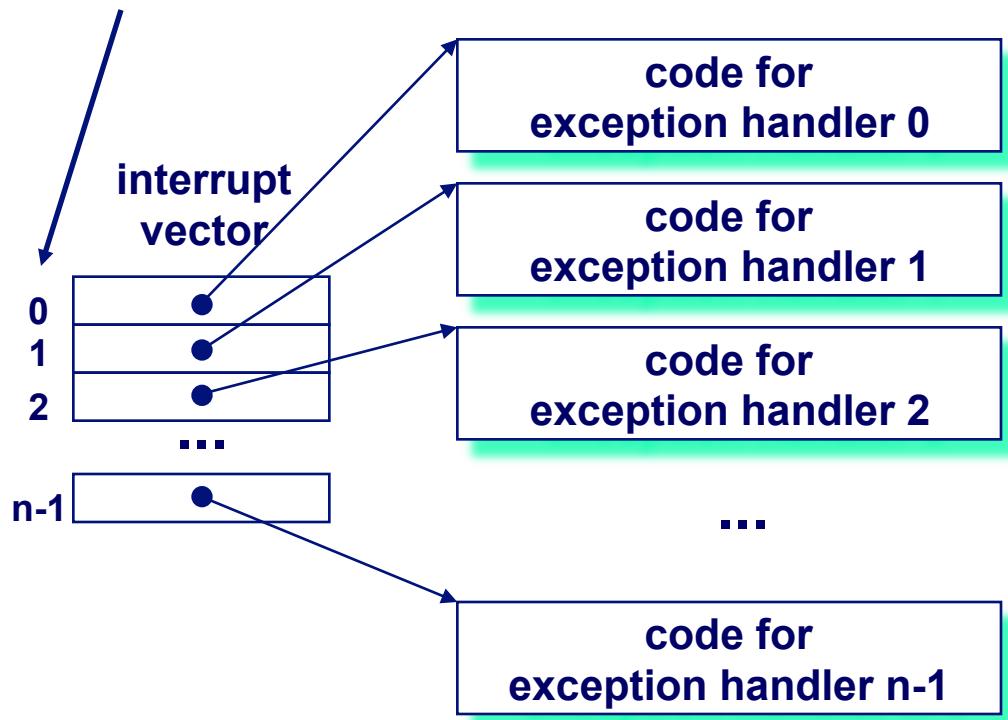
System context for exceptions

Our focus until now...



Interrupt Vectors

Exception numbers



- Each type of event has a unique exception number k
- Index into jump table (a.k.a., interrupt vector)
- Jump table entry k points to a function (exception handler).
- 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 **ctl-c** at the keyboard
 - arrival of a packet from a network
 - arrival of a data sector from a disk
- Hard reset interrupt
 - hitting the **reset button**
- Soft reset interrupt
 - hitting **ctl-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 (unrecoverable).
 - Either re-executes faulting (“current”) instruction or aborts.
- **Aborts**
 - unintentional and unrecoverable
 - Examples: parity error, machine check.
 - Aborts current program

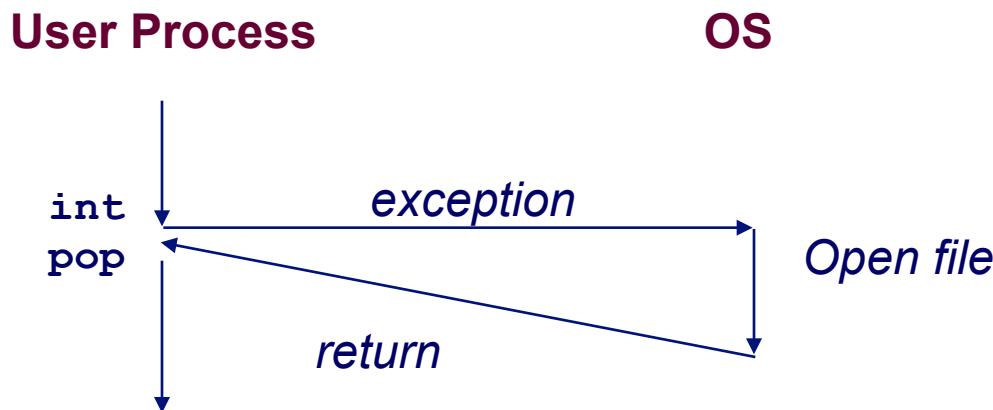
Trap Example

Opening a File

- User calls `open(filename, options)`

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

- Function `open` executes system call instruction `int`
- OS must find or create file, get it ready for reading or writing
- Returns integer file descriptor



Fault Example #1

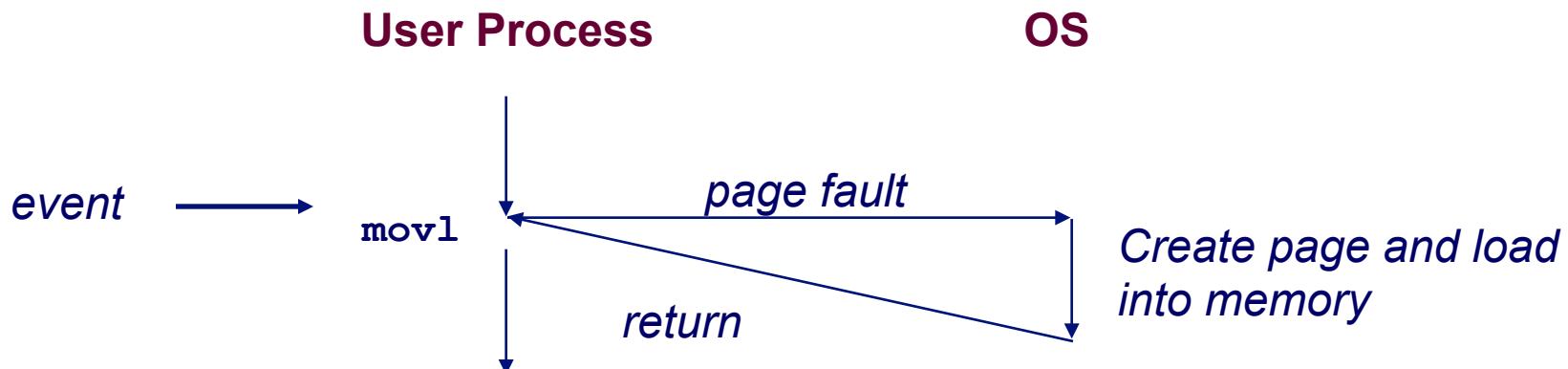
Memory Reference

- 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 #2

Memory Reference

- User writes to memory location
- Address is not valid

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

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



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

Processes

Def: 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:

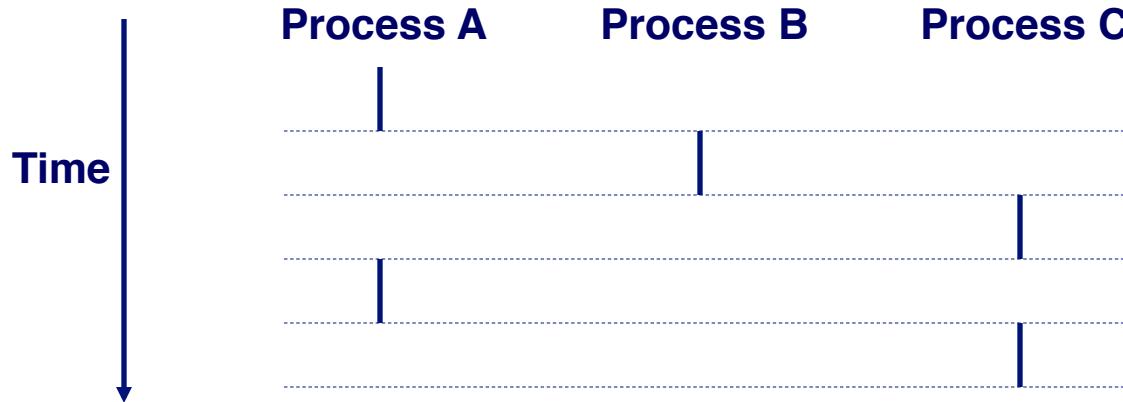
- Logical control flow
 - Each program seems to have exclusive use of the CPU.
- Private address space
 - Each program seems to have exclusive use of main memory.

How are these Illusions maintained?

- Process executions interleaved (multitasking)
- Address spaces managed by virtual memory system

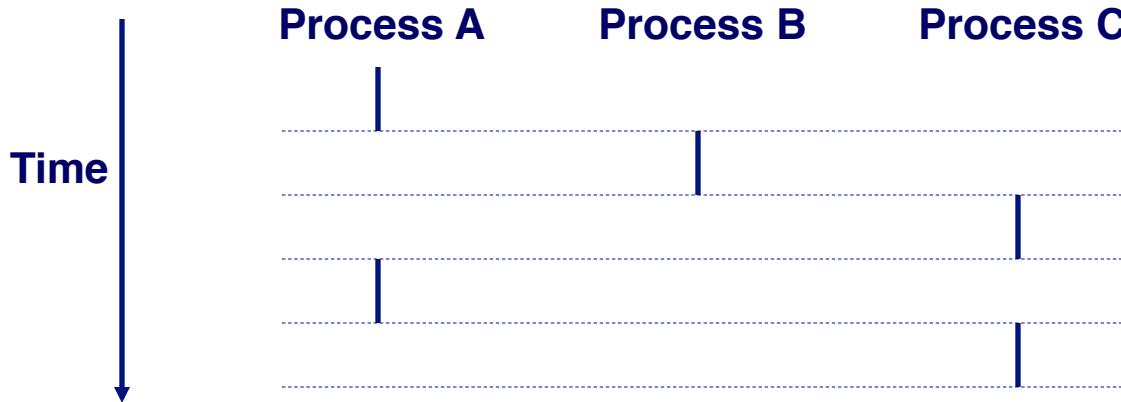
Logical Control Flows

Each process has its own logical control flow



Concurrent Processes

Two processes *run concurrently (are concurrent)* if their flows overlap in time. Otherwise, they are *sequential*.



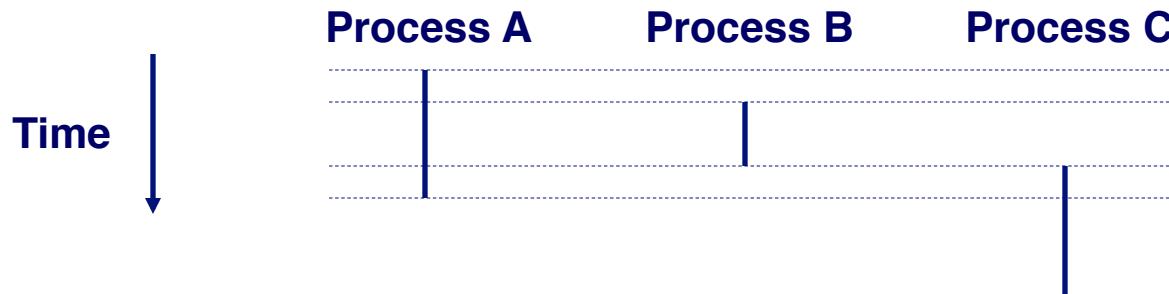
Examples:

- 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 are running in parallel with each other.

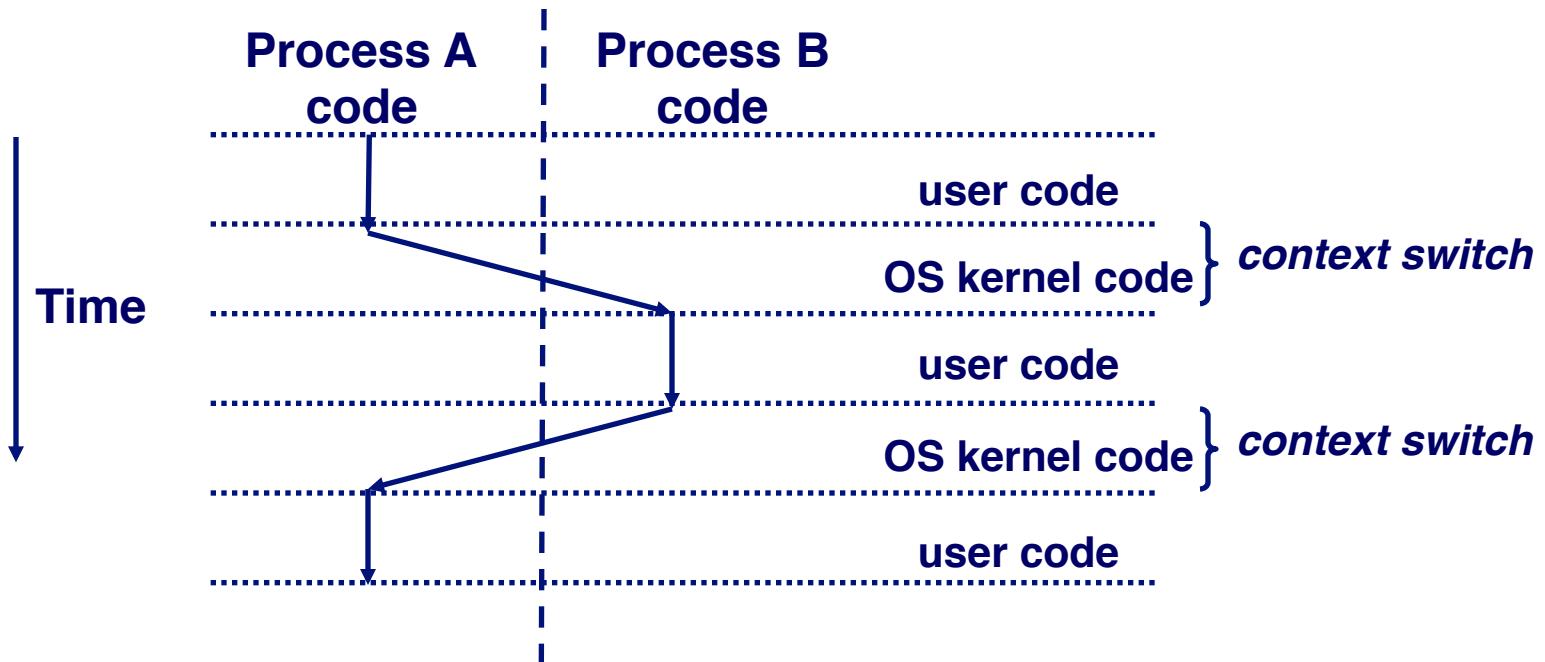


Context Switching

Processes are managed by a shared chunk of OS code called the *kernel*

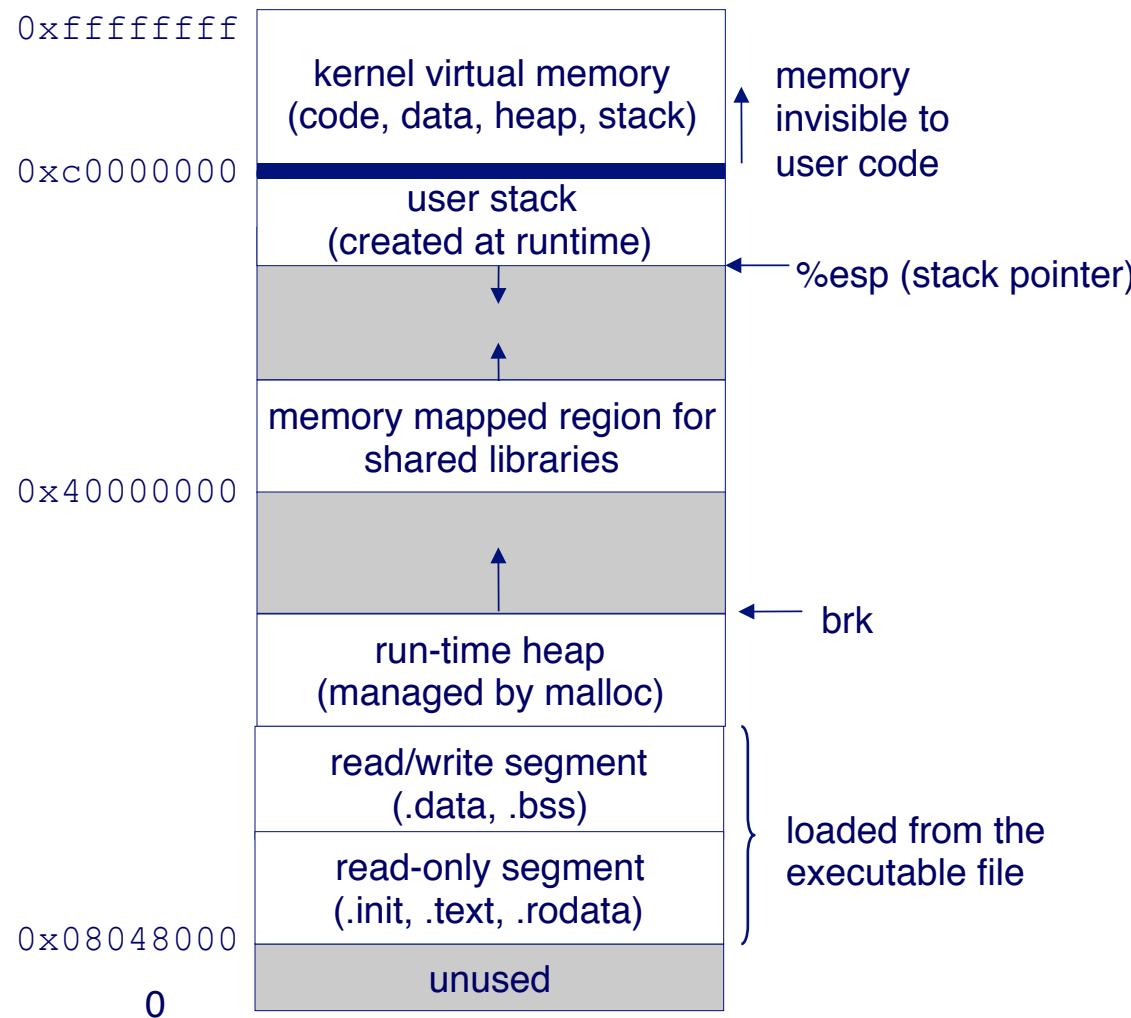
- Important: the kernel is not a separate process, but rather runs as part of some user process

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



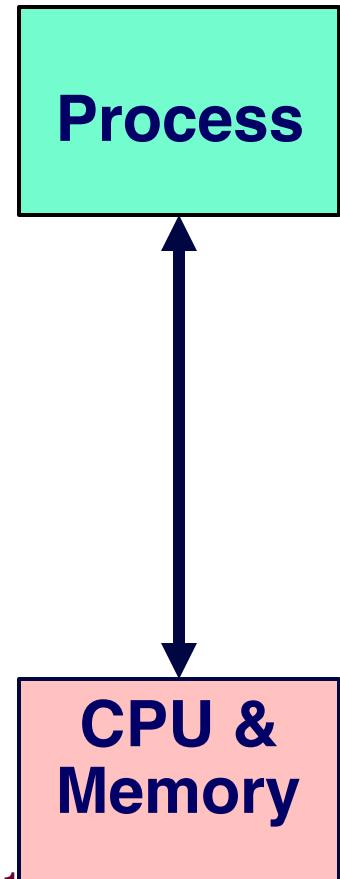
Private Address Spaces

Each process has its own private address space.



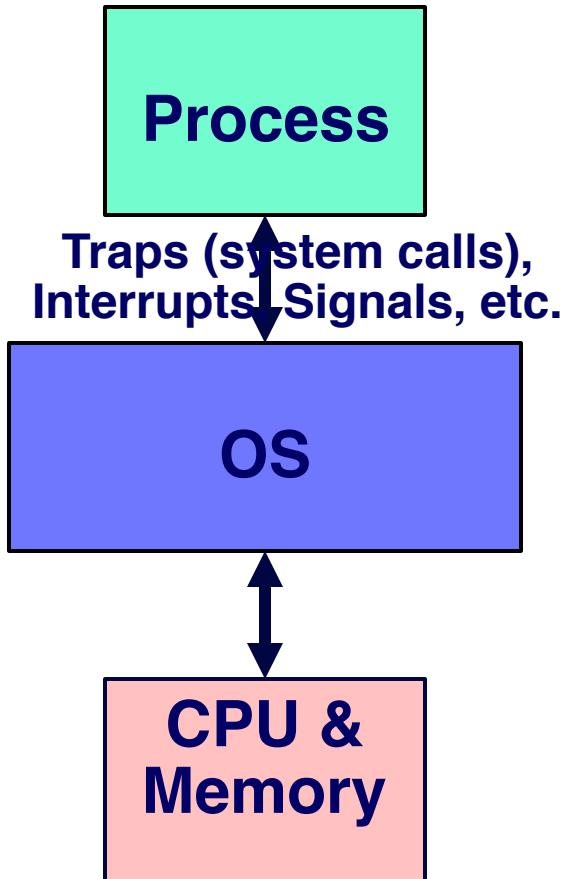
Process & OS Conceptual View

Original Concept:



1 Isolated Process

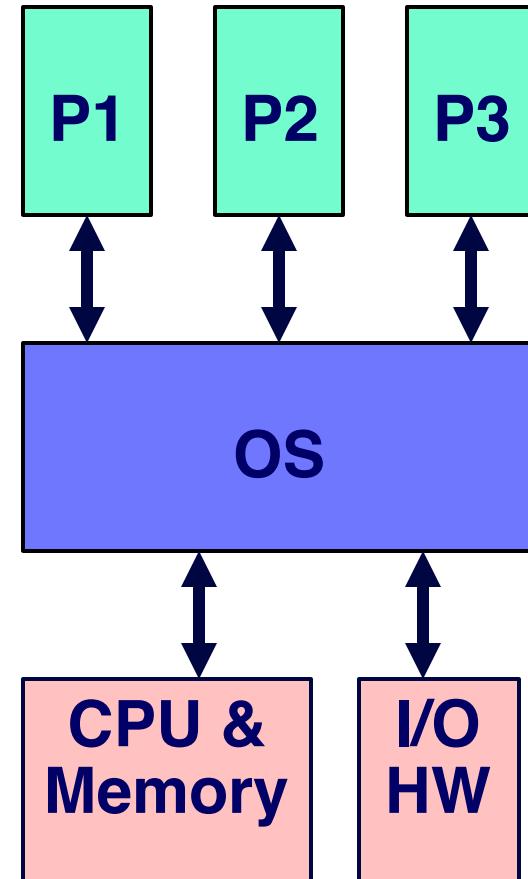
Revised Concept:



1 Process + OS

Overall Concept:

Processes + OS



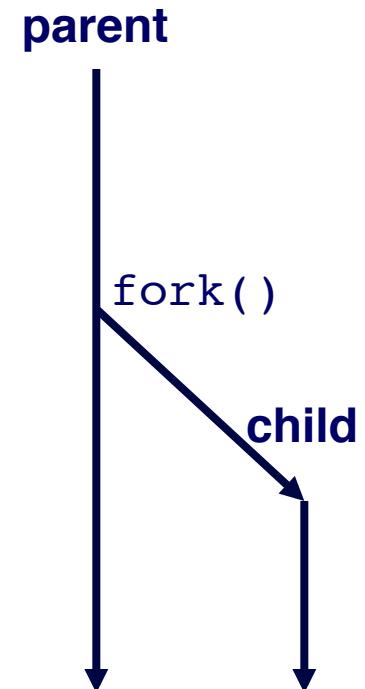
fork: Creating new processes

```
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 to the parent process

```
if (fork() == 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*



Fork Example #1

```
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);
}
```

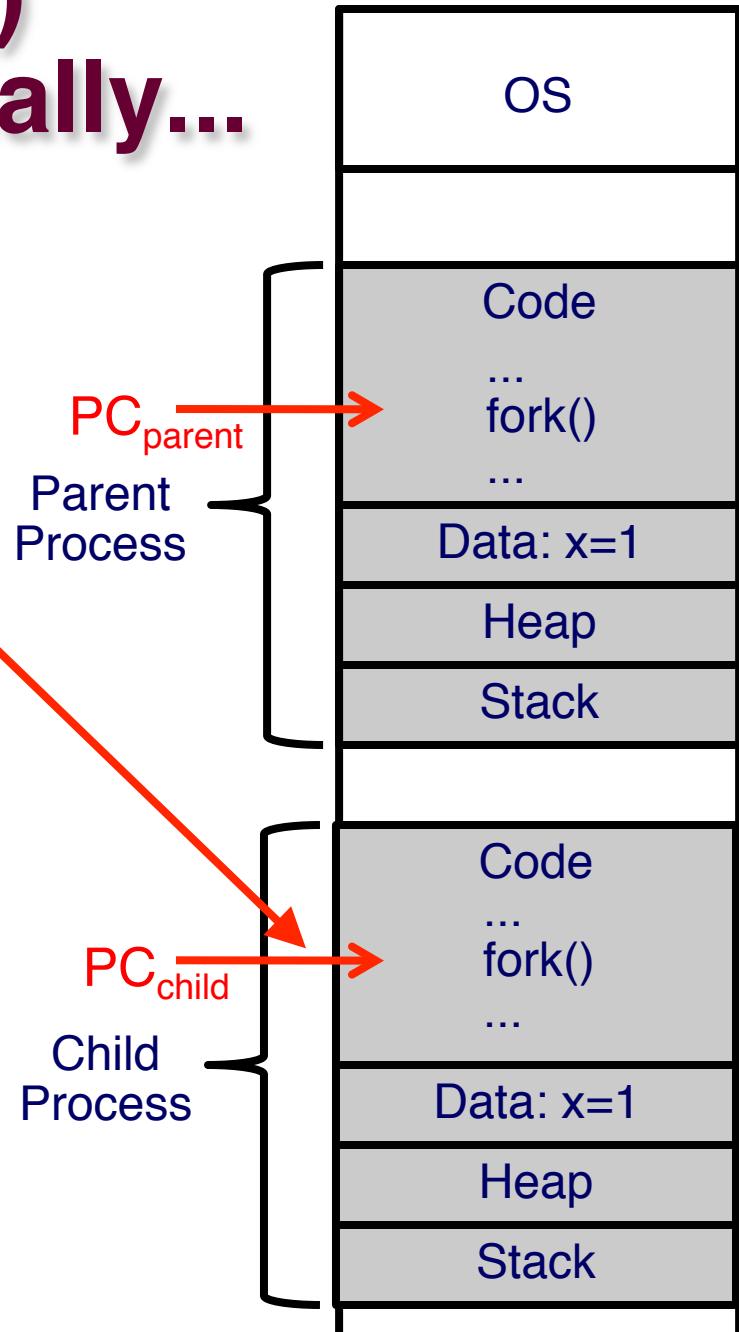
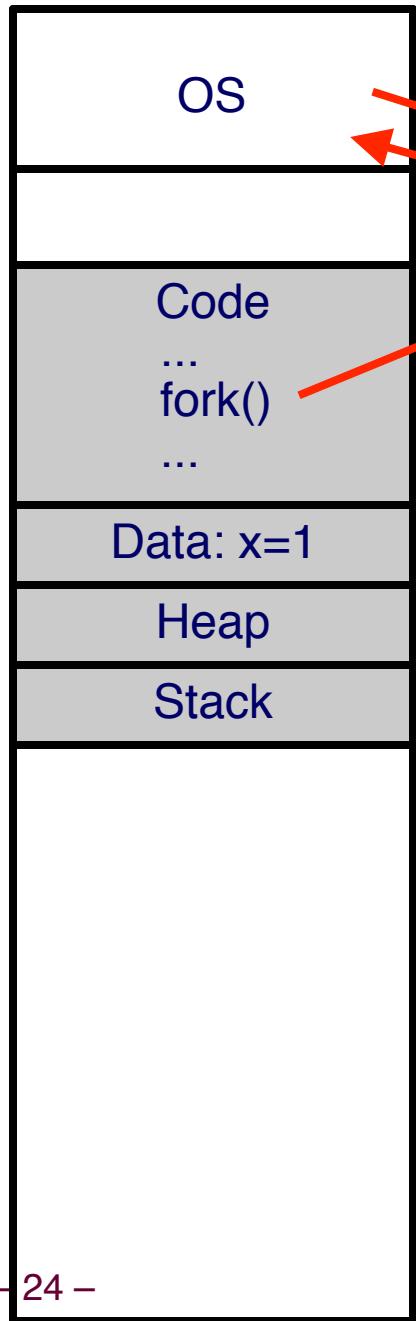
Key Points

- Parent and child both run same code, i.e. they start as ***twins!***
 - Except parent differs 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

Memory (before fork)

Fork() conceptually...

Memory (after fork)



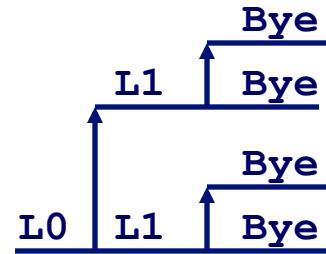
- **Fork() duplicates address space of parent in the child**
- **Both execute concurrently**

Fork Example #2

Key Points

- Both parent and child can continue forking

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

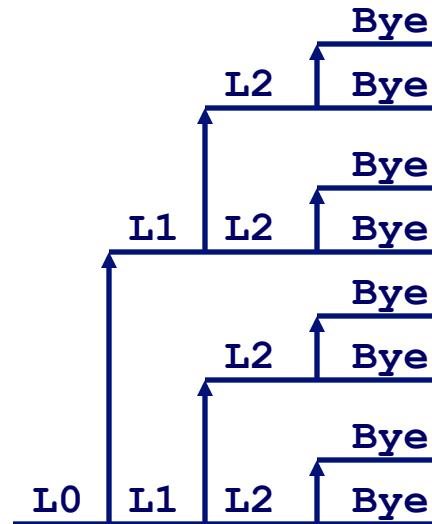


Fork Example #3

Key Points

- Both parent and child can continue forking

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

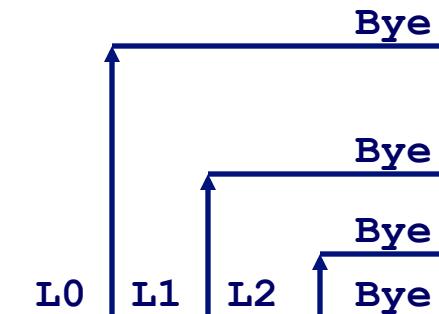


Fork Example #4

Key Points

- Both parent and child can continue forking

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

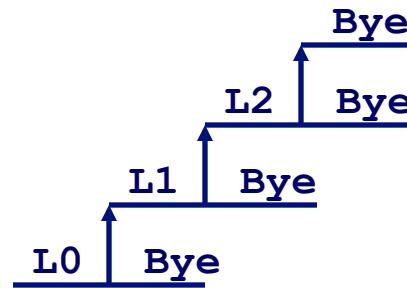


Fork Example #5

Key Points

- Both parent and child can continue forking

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



Note: avoid fork “bombs”, i.e. uncontrolled repeated forking, which can disable a system

exit: Destroying Process

`void exit(int status)`

- exits a process

- Normally return with status 0
- A non-zero return status can be passed to the `wait()` statement to identify the exiting process to the waiting process (see later slides)

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

- `atexit()` registers functions to be executed upon exit

Zombies

Idea

- When process terminates, still consumes system resources
 - Various tables maintained by OS
- Called a “zombie”
 - Living corpse, half alive and half dead

Reaping

- Performed by parent on terminated child
- Parent is given exit status information
- Kernel discards process

What if Parent Doesn't Reap?

- If any parent terminates without reaping a child, then if a child later terminates, it will be reaped by the `init` process
- If a child is a long-running process, its resources can be freed only by explicitly terminating it
 - 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**

Nonterminating Child Example

```
linux> ./forks 8
Terminating Parent, PID = 6675
Running Child, PID = 6676
linux> ps
  PID TTY          TIME CMD
 6585 tttyp9      00:00:00 tcsh
 6676 tttyp9      00:00:06 forks
 6677 tttyp9      00:00:00 ps
linux> kill 6676
linux> ps
  PID TTY          TIME CMD
 6585 tttyp9      00:00:00 tcsh
 6678 tttyp9      00:00:00 ps
```

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

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

wait: Synchronizing with children

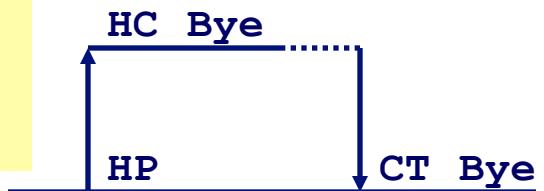
```
int wait(int *child_status)
```

- 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
- This then triggers the OS to reap/deallocate the memory of the child process

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 with Multiple Children

- 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 immediately exits */
    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);
    }
}
```

Wait Example Outputs

Using `wait` (`fork10`)

```
Child 3565 terminated with exit status 103
Child 3564 terminated with exit status 102
Child 3563 terminated with exit status 101
Child 3562 terminated with exit status 100
Child 3566 terminated with exit status 104
```

Waitpid

■ `waitpid(pid, &status, options)`

- Can wait for specific process
- Various options

```
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 = 0; i < N; 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);
    }
}
```

Wait/Waitpid Example Outputs

Using wait (fork10)

```
Child 3565 terminated with exit status 103
Child 3564 terminated with exit status 102
Child 3563 terminated with exit status 101
Child 3562 terminated with exit status 100
Child 3566 terminated with exit status 104
```

Using waitpid (fork11)

```
Child 3568 terminated with exit status 100
Child 3569 terminated with exit status 101
Child 3570 terminated with exit status 102
Child 3571 terminated with exit status 103
Child 3572 terminated with exit status 104
```

Terminated
in order

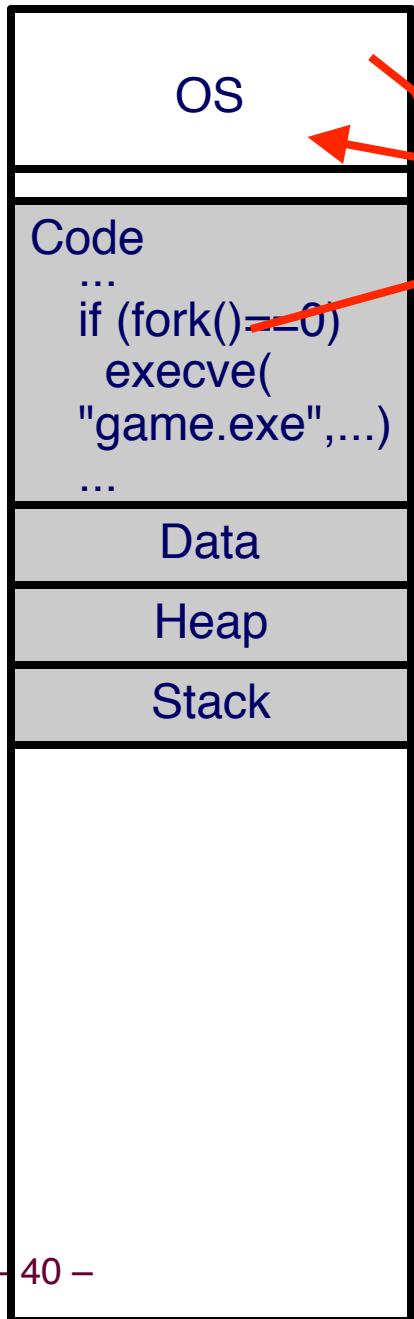
exec: Running new programs

```
int execl(char *path, char *arg0, char *arg1, ..., 0)
```

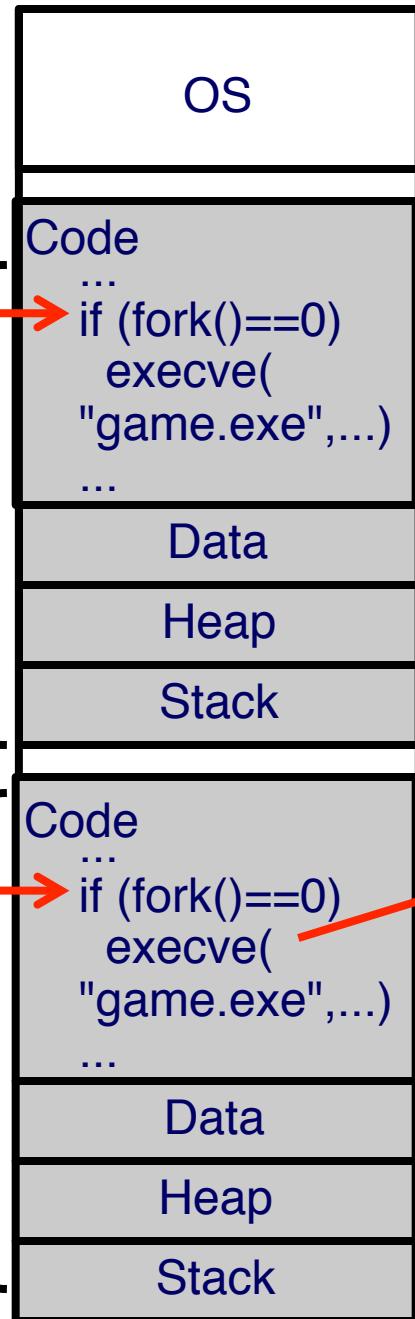
- loads and runs executable at path with args arg0, arg1, ...
 - path is the complete path of an executable
 - arg0 becomes the name of the process
 - » typically arg0 is either identical to path, or else it contains only the executable filename from path
 - “real” arguments to the executable start with arg1, etc.
 - list of args is terminated by a (char *) 0 argument
- returns -1 if error, otherwise doesn't return!

```
main() {  
    if (fork() == 0) {  
        execl("/usr/bin/cp", "cp", "foo", "bar", 0);  
    }  
    wait(NULL);  
    printf("copy completed\n");  
    exit();  
}
```

Memory (before fork)



Memory (after fork)



Memory (after execve)

