Chapter 8: Exceptional Control Flow and Shell Programs

Chapter 8 Topics:

- Exceptions
- Processes
- Signals

Announcements

Performance Lab grading this week

Midterm 2 next Tuesday April 18 in class

- See email/lecture announcements. Bring your laptops.
- Students needing extended time should contact profs/TAs
- Monday midterm review session
- Practice midterm #2 with solutions released

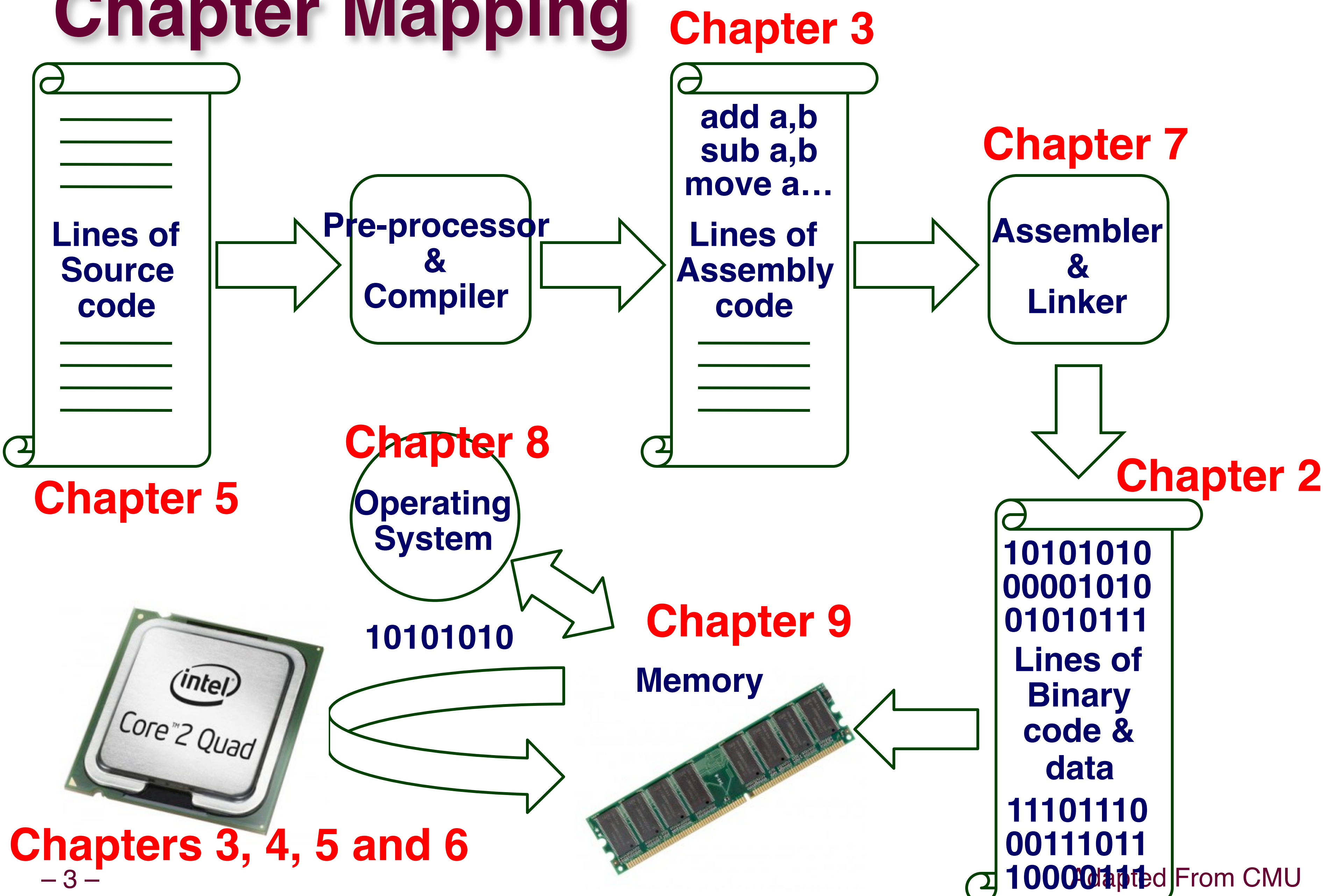
Shell Lab released – material from Chapter 8

- Recitations introduce it next week.

Reading

- Skip Chapter 7, go to Ch 8, read all sections (except 8.6), return to Ch 7 at end

Chapter Mapping



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*).**

Physical control flow

Time

<startup>
inst_1
inst_2
inst_3
…
inst_n
<shutdown>

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

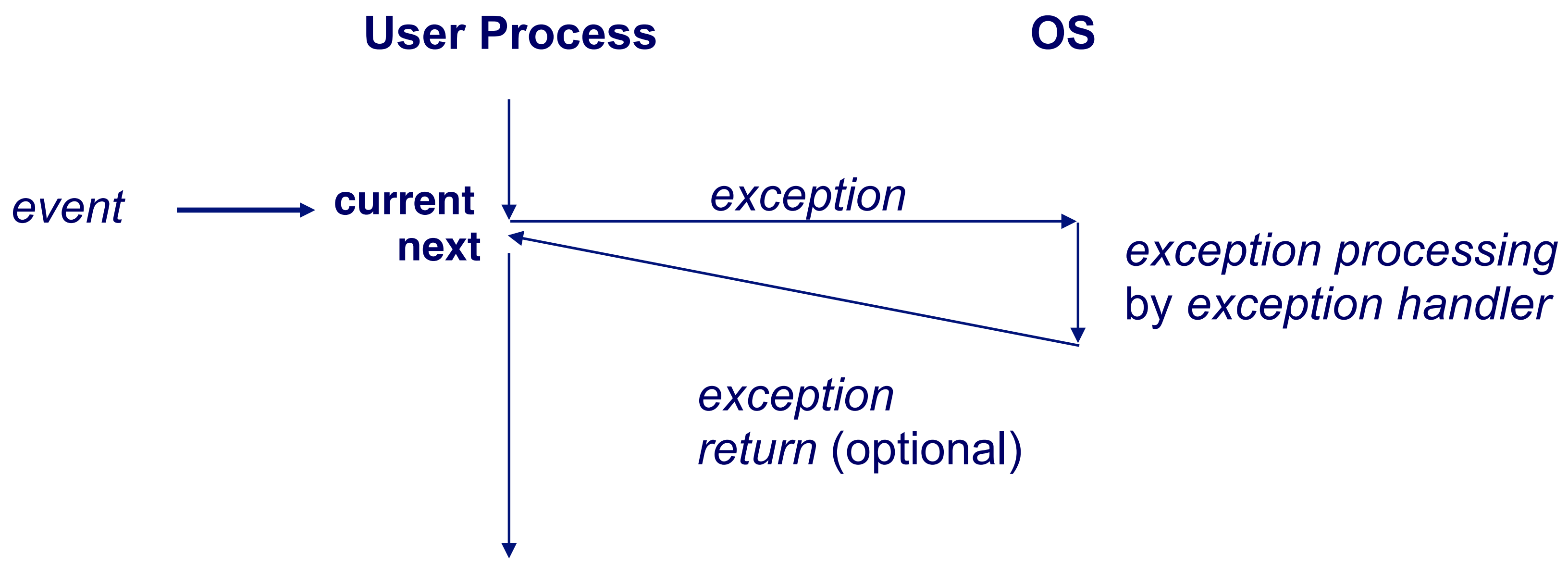
- exceptions
  - change in control flow in response to a system event (i.e., change in system state)
- Combination of hardware and OS software

Higher Level Mechanisms

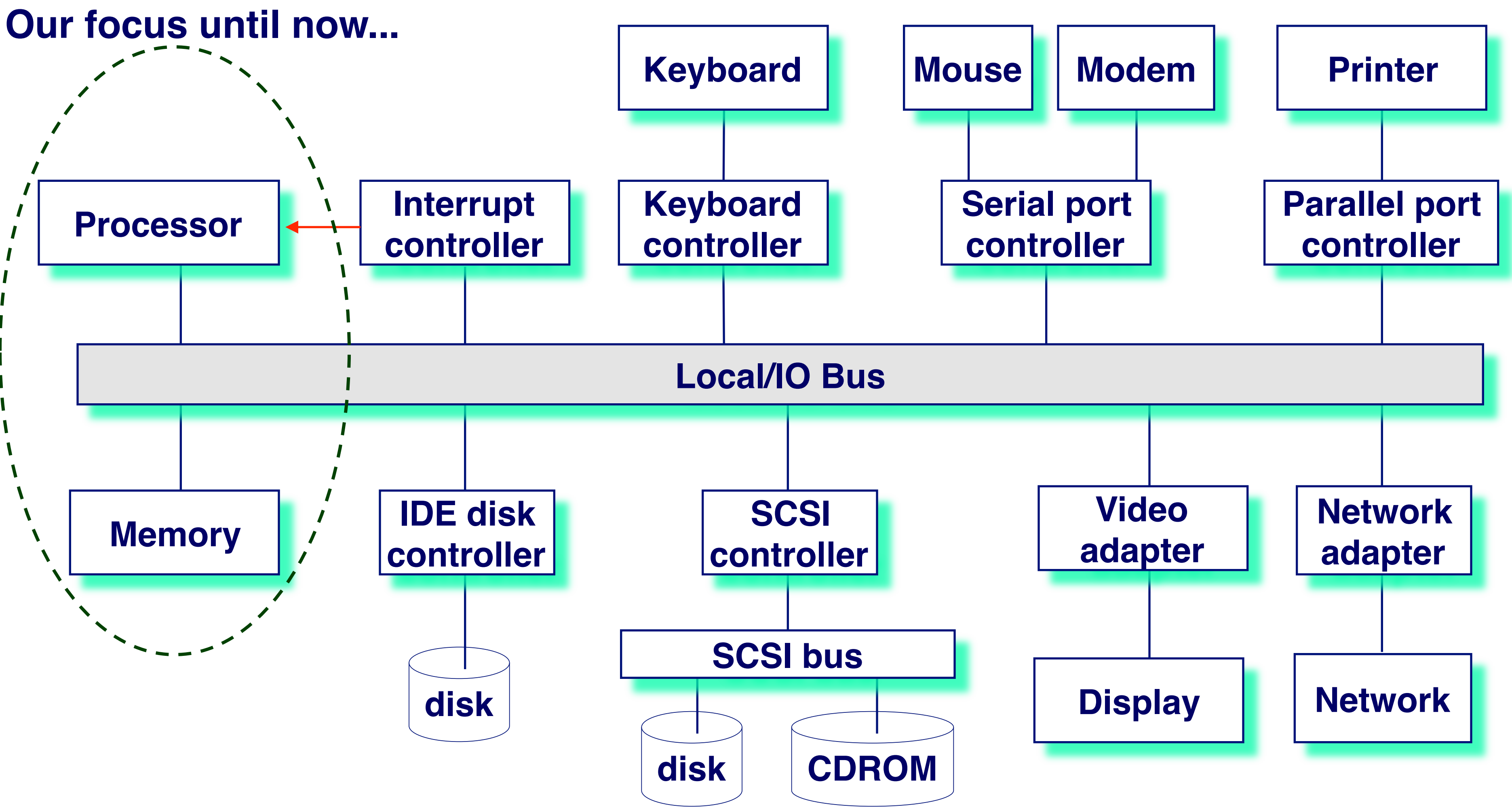
- Process context switch
- Signals
- Nonlocal jumps (setjmp/longjmp), try / except blocks
- Implemented by either:
  - OS software (context switch and signals).
  - C language runtime library: nonlocal jumps.

Exceptions

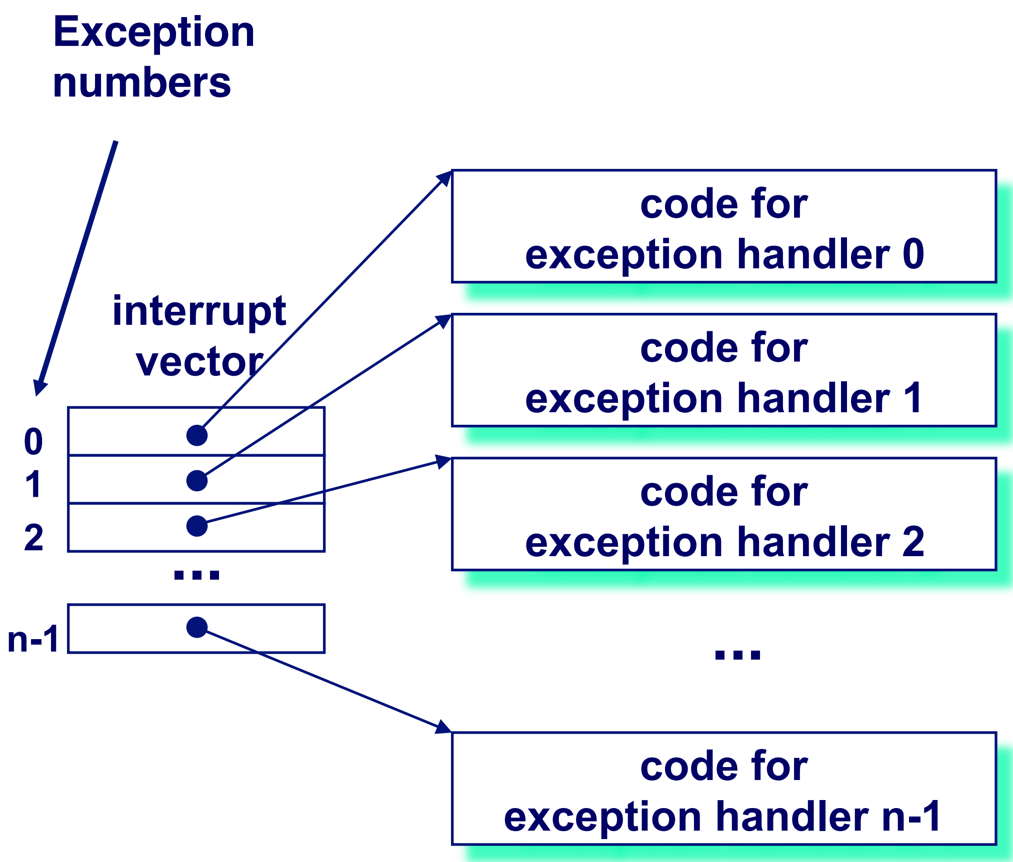
An *exception* is a transfer of control to the OS in response to some *event* (i.e., change in processor state)



System context for exceptions



Interrupt Vectors



- 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

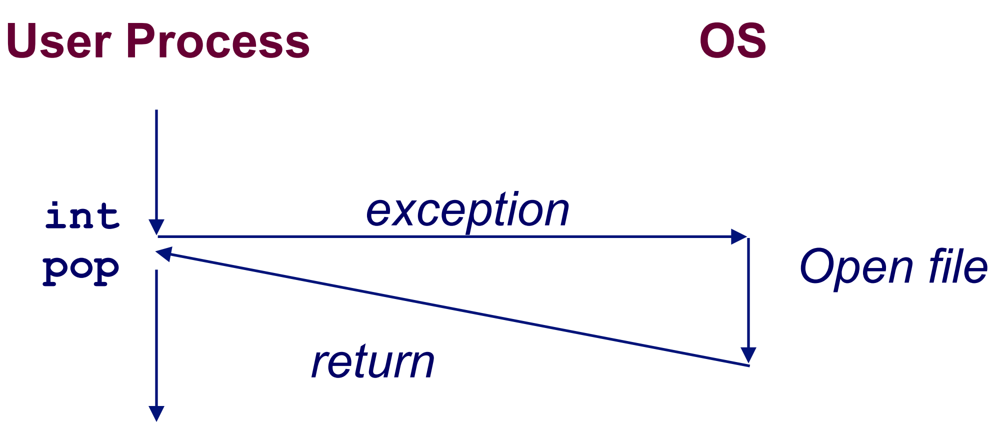
Trap Example

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

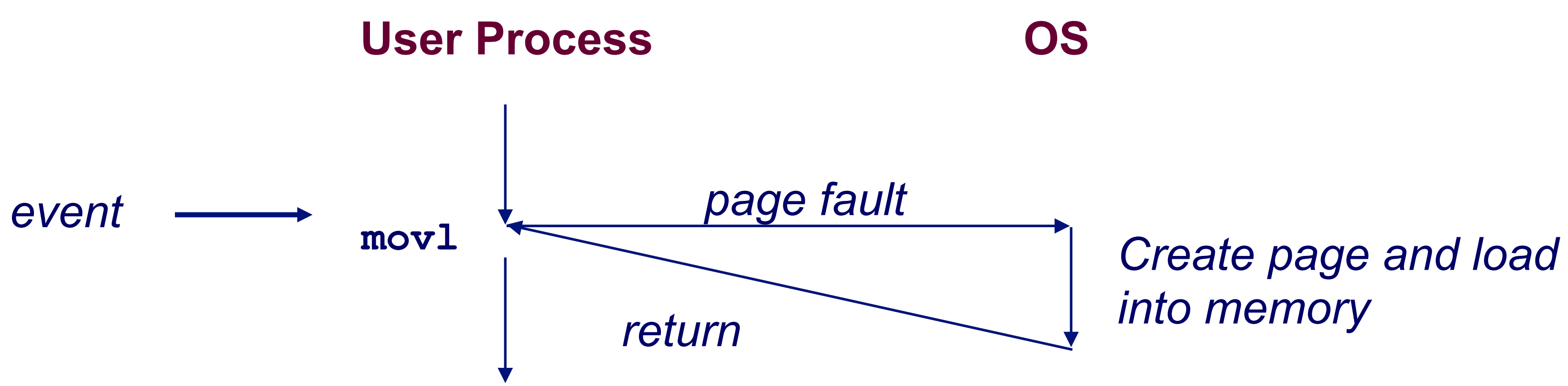
```
int a[1000];
main ()
{
    a[500] = 13;
}
```

Memory Reference

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

```
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

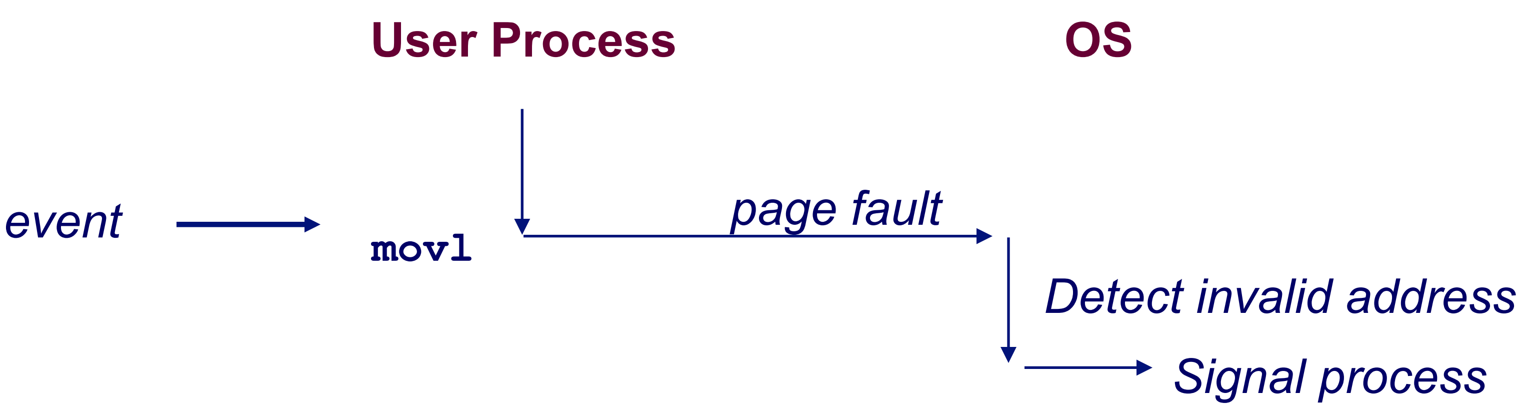
```
int a[1000];
main ()
{
    a[5000] = 13;
}
```

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”



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:

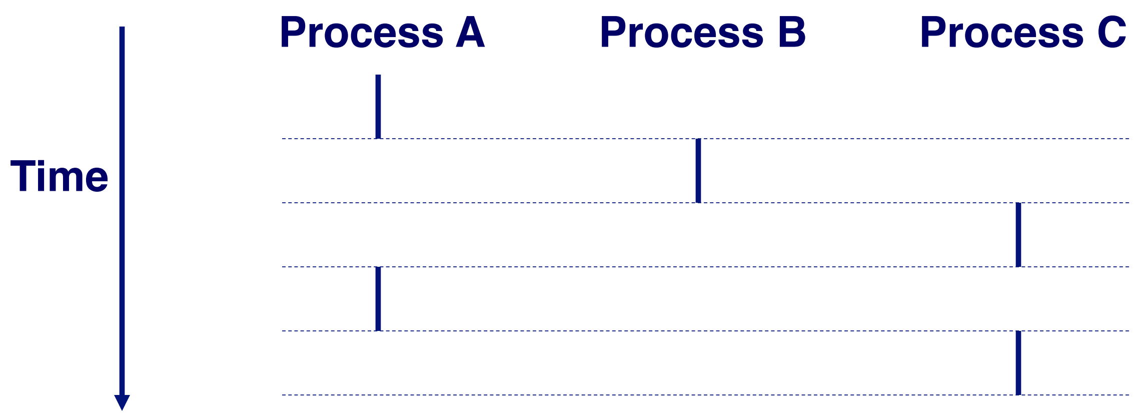
- Logical control flow
  - Each program seems to have exclusive use of the CPU.
- Private address space
  - Each program seems to have exclusive use of main memory.

How are these Illusions maintained?

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

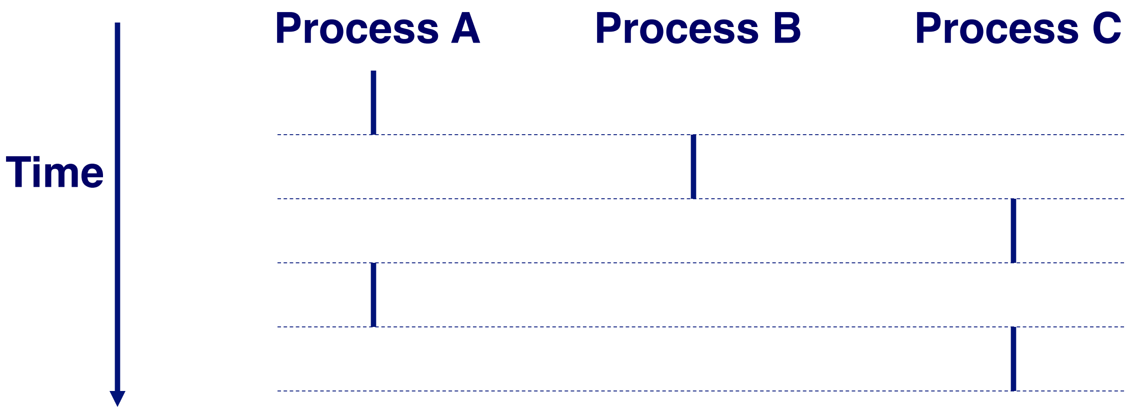
Logical Control Flows

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*.



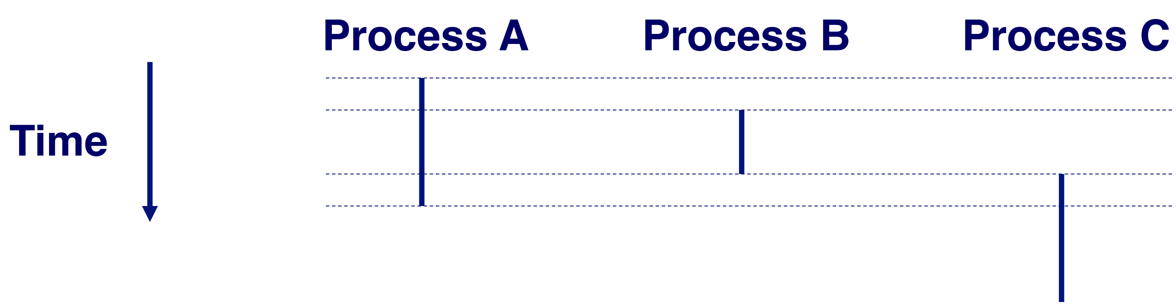
Examples:

- 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.

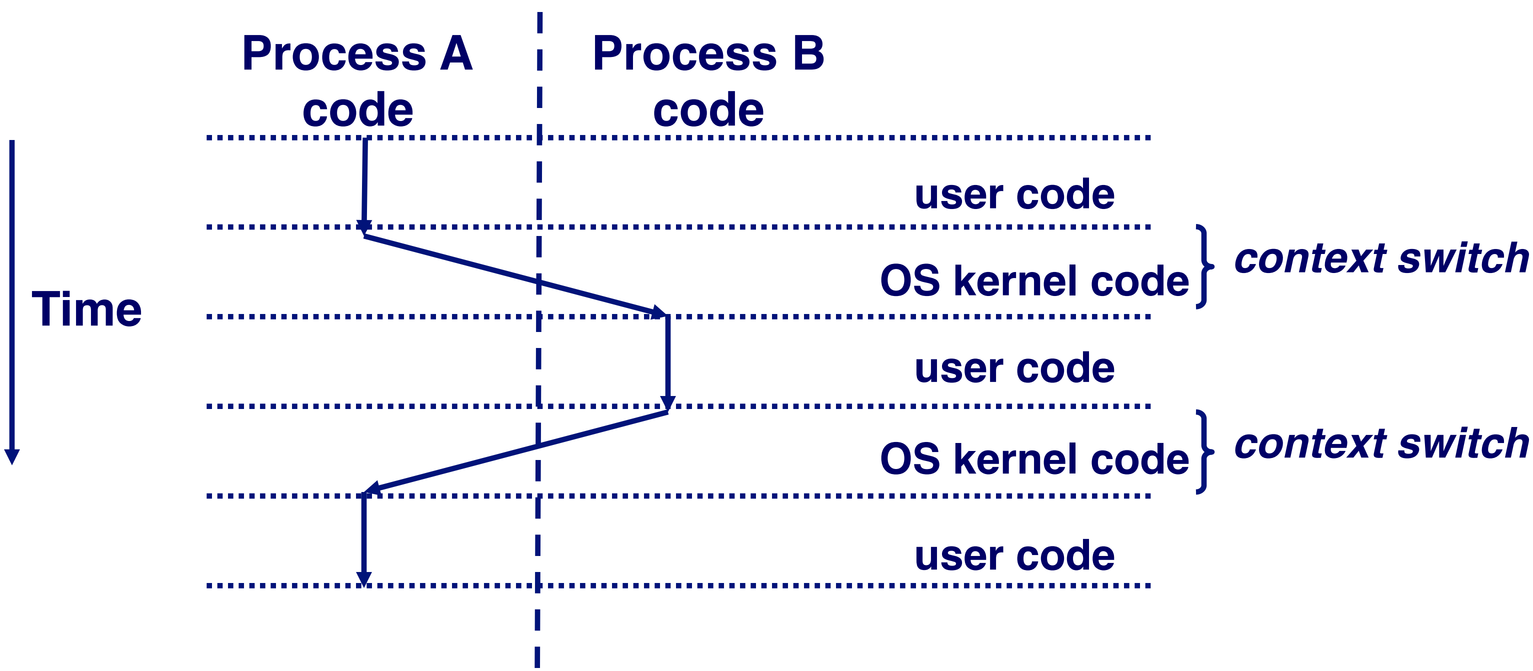


Context Switching

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

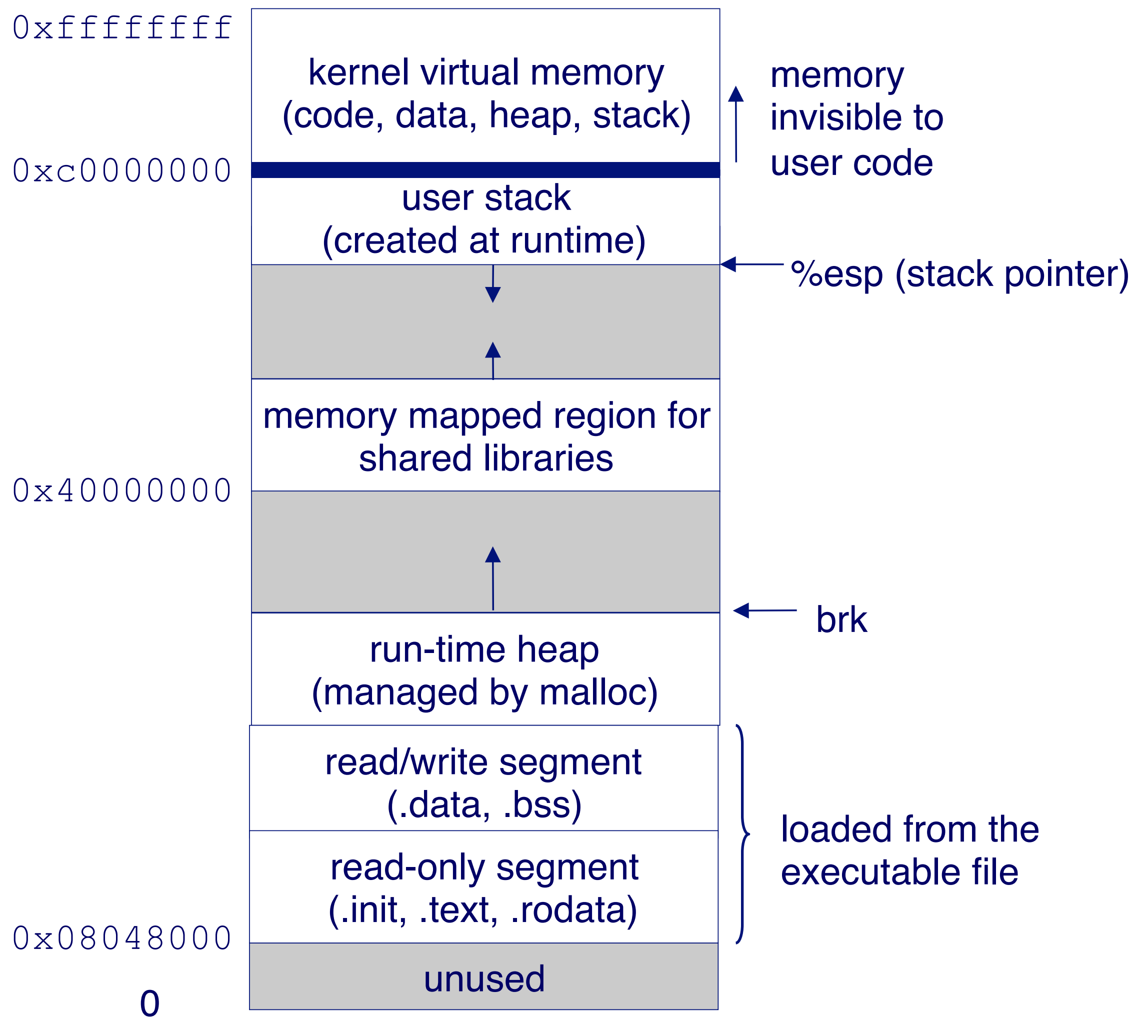
- Important: the kernel is not a separate process, but rather runs as part of some user process

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



Private Address Spaces

Each process has its own private address space.

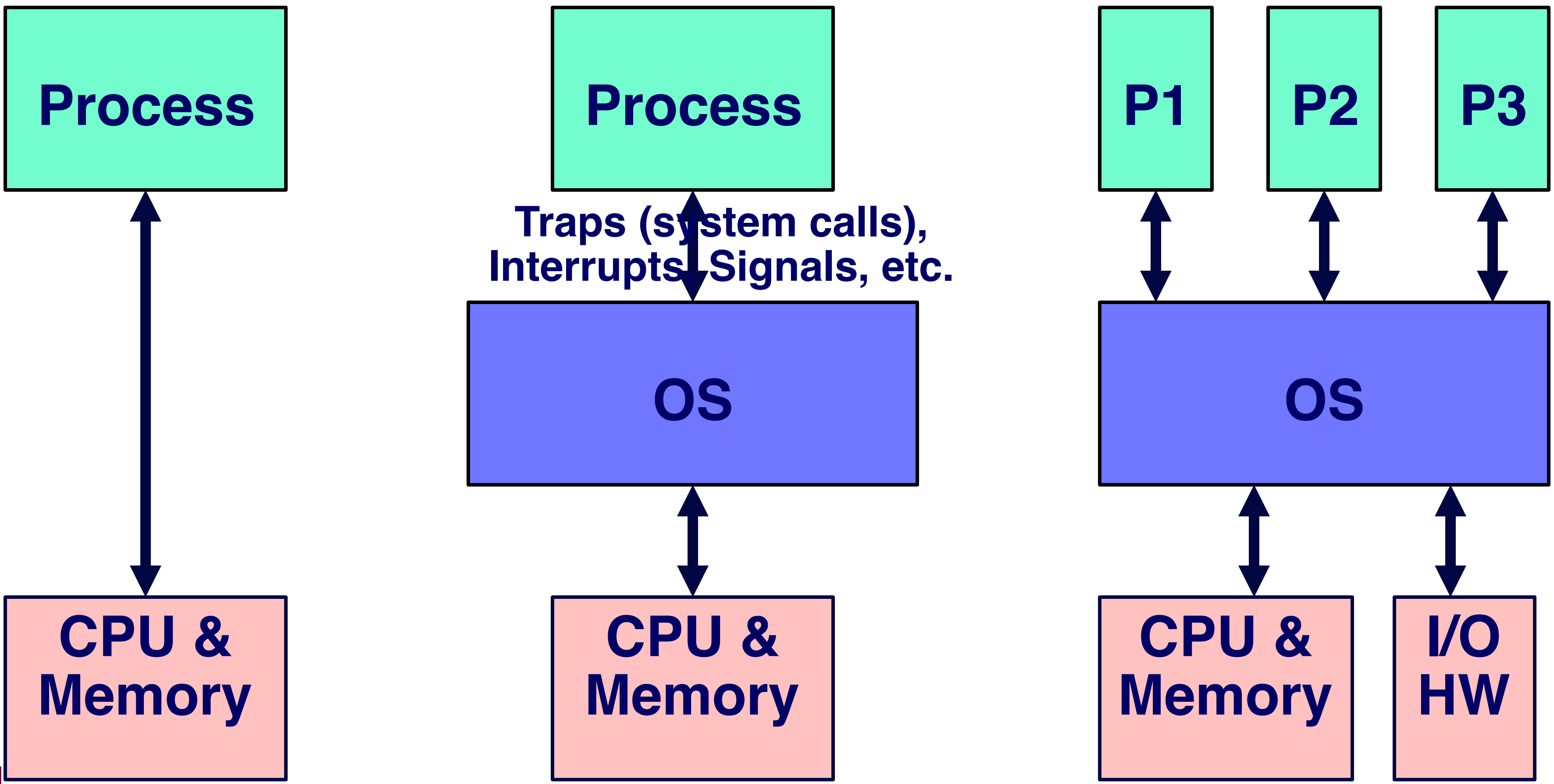


Process & OS Conceptual View

Original Concept: 1 Isolated Process

Revised Concept: 1 Process + OS

Overall Concept: Processes + OS

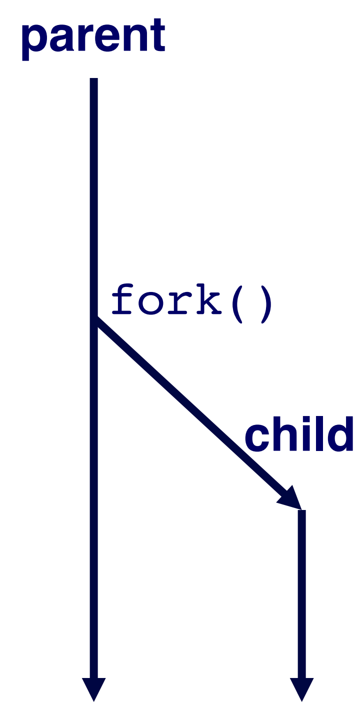


`fork`: Creating new processes

`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*

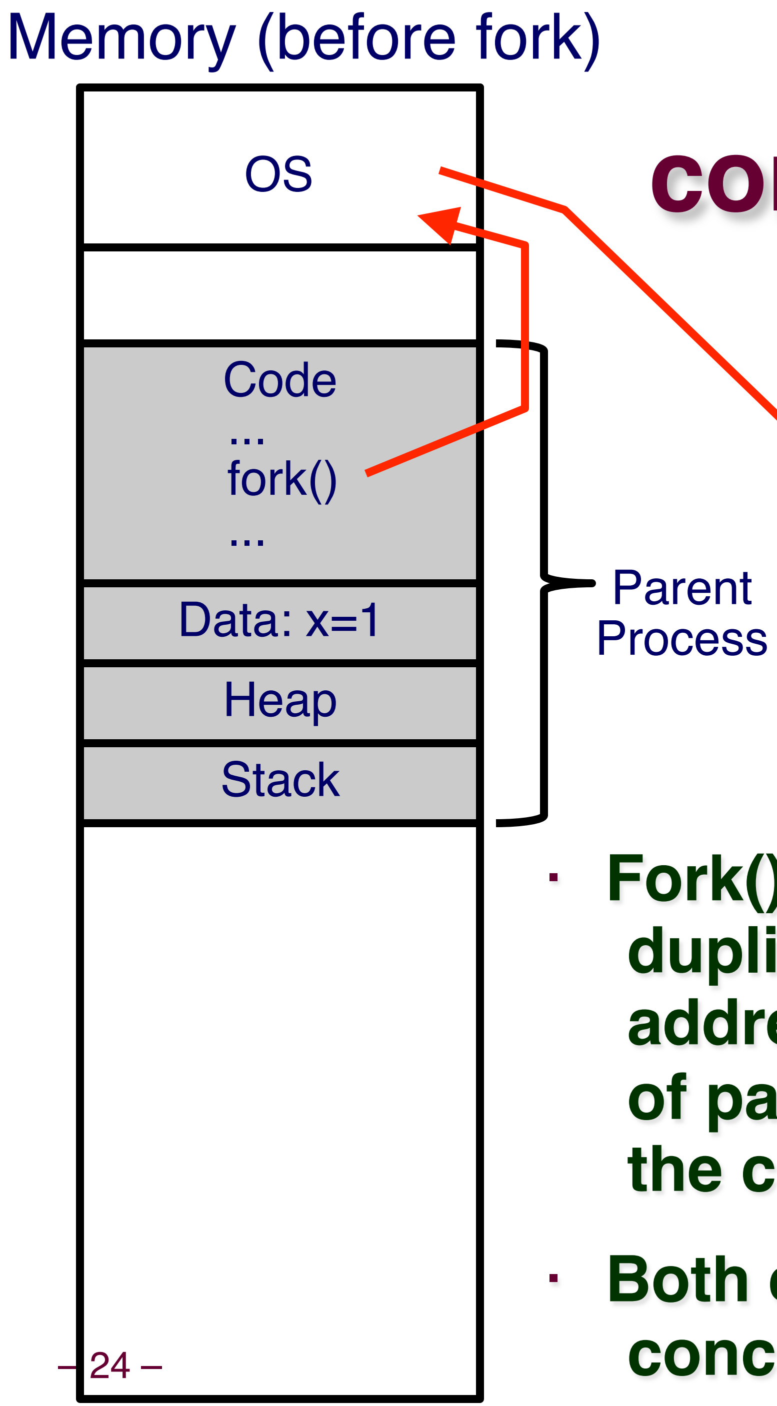
Fork Example #1

```
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);
}
```

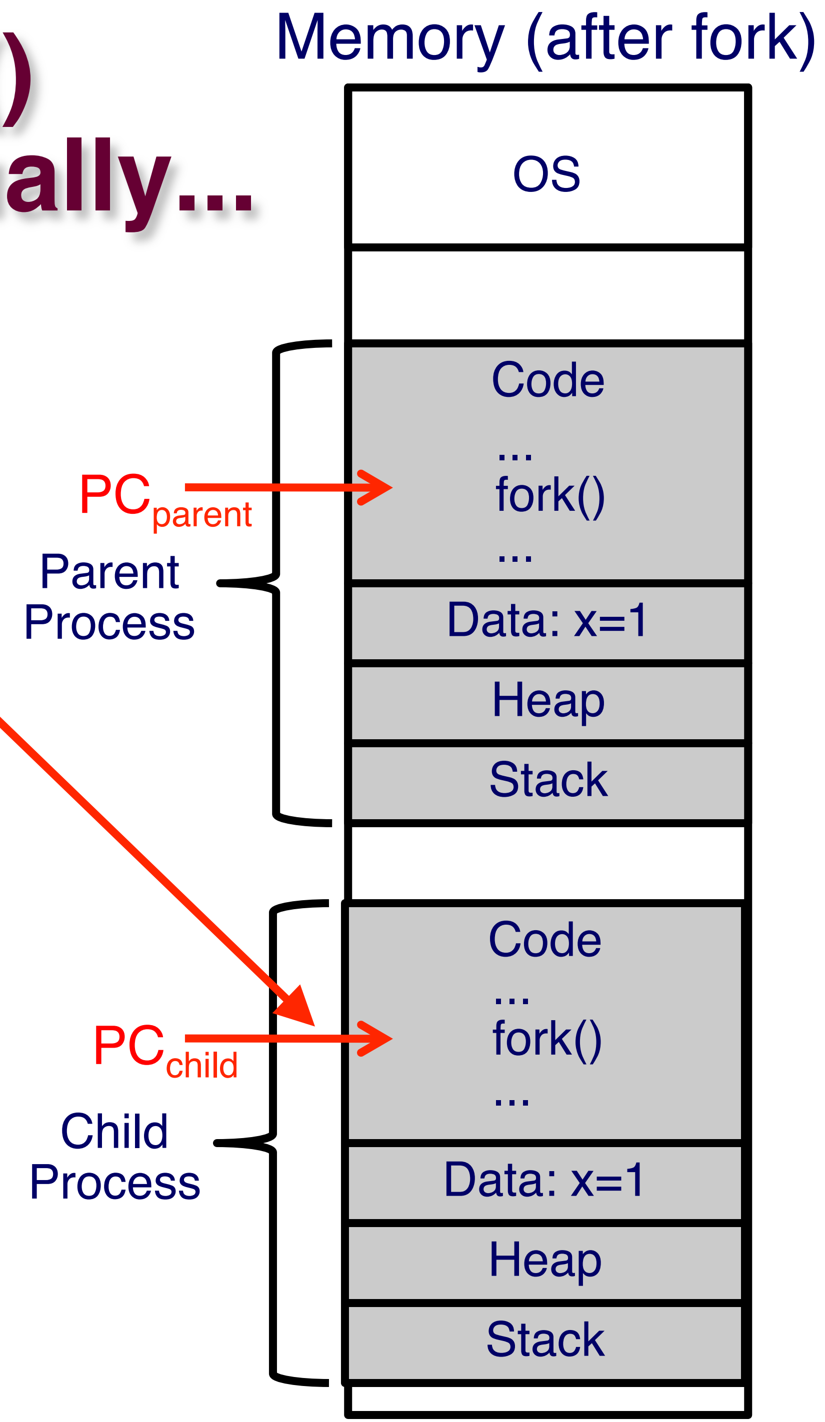
Key Points

- 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

Fork() conceptually...



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

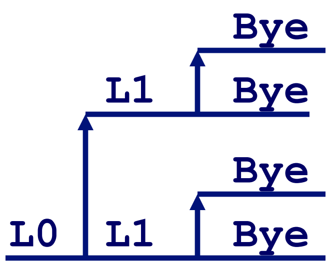


Fork Example #2

Key Points

- Both parent and child can continue forking

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

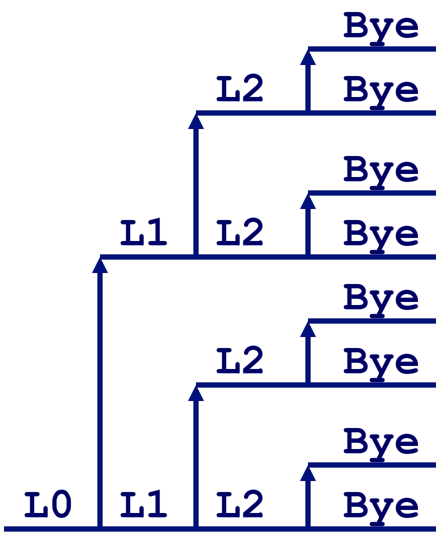


Fork Example #3

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");
}
```

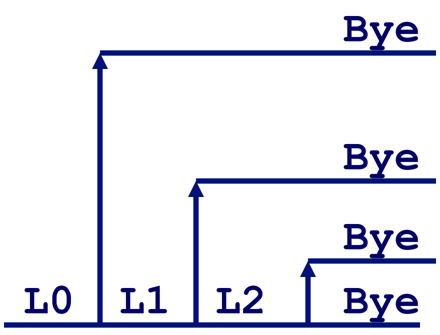


Fork Example #4

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");
}
```

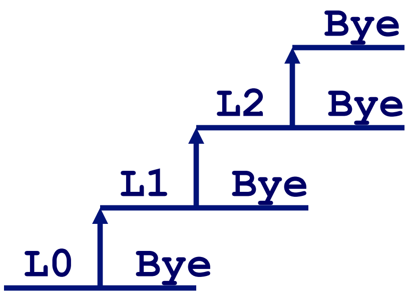


Fork Example #5

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