UNM SCHOOL of ENGINEERING
Department of Computer Science

# Lecture #23:
# Chap. 8
# Exceptional Control Flow (ECF)

Prof. Soraya Abad-Mota, PhD

# Topics

- Intro. to Exceptional Control Flow (ECF)
- Exceptions
- The Process Abstraction
- Process Control

# 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*)

*Physical control flow*

Time

<startup>
$inst_1$
$inst_2$
$inst_3$
…
$inst_n$
<shutdown>

Control transfer: transition from $inst_i$ to $inst_j$ (may not be adjacent)

Flow of control: sequence of control transfers

# 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,*** provided by the program variables

- 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

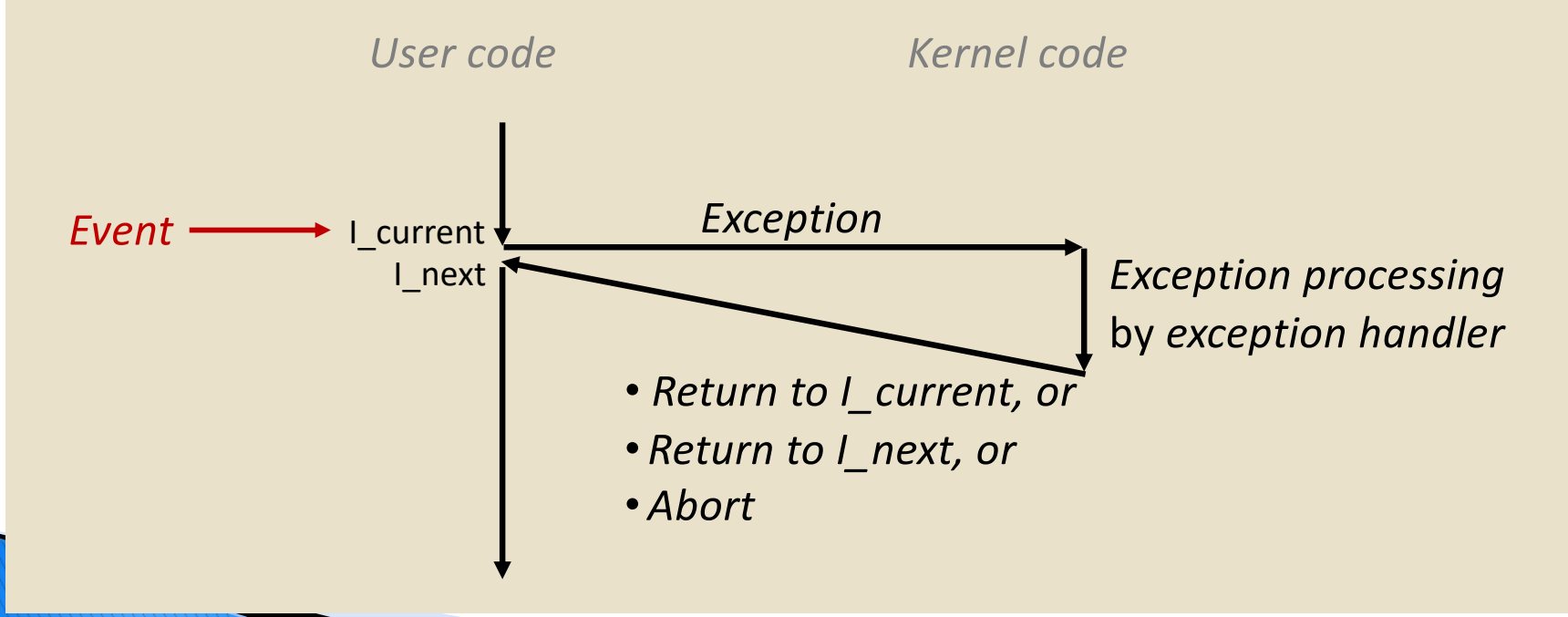
- Exists at all levels of a computer system
- Low level mechanisms
  - 1. **Exceptions**
    - Change in control flow in response to a system event (i.e., change in system state)
    - Implemented using combination of hardware and OS software
- Higher level mechanisms
  - 2. **Process context switch**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

# Topics

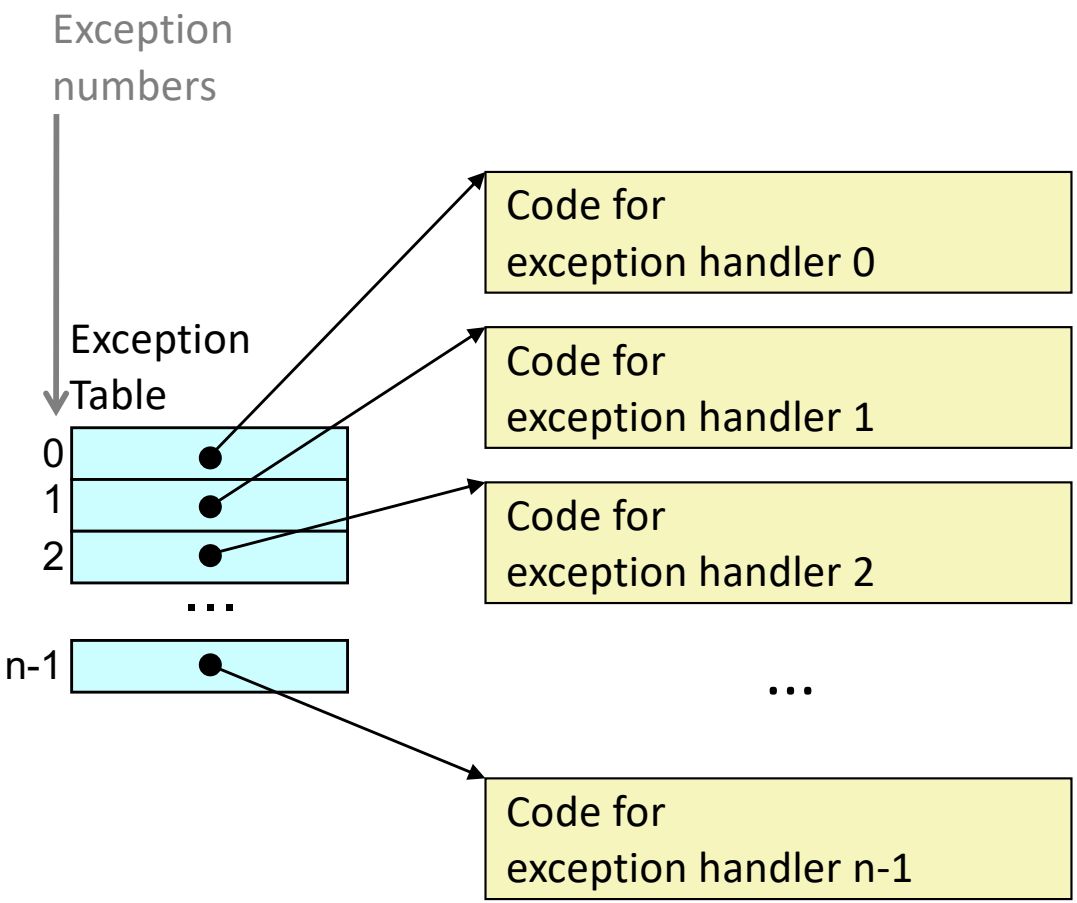
- Intro. to Exceptional Control Flow (ECF)
- Exceptions
- The Process Abstraction
- Process Control

# Exceptions

- An *exception* is a transfer of control to the OS *kernel* in response to some *event* (i.e., change in processor state)
  - Kernel is the memory-resident part of the OS
  - Examples of events: Divide by 0, arithmetic overflow, page fault, I/O request completes, typing Ctrl-C



# Exception Tables




- Each type of event has a unique exception number k
- k = index into exception table (a.k.a. *interrupt vector*, similar to a jump table)
- Handler k is called each time exception k occurs
- Handlers run in kernel mode

exception table base register: contains the starting address of the exception table

# Asynchronous Exceptions (Interrupts)

- Caused by events external to the processor
  - Indicated by setting the processor's *interrupt pin*
  - Places the exception number on the system bus
  - (Interrupt) Handler returns to "next" instruction
- Examples:
  - Timer interrupt
    - Every few ms, an external timer chip triggers an interrupt
    - Used by the kernel to take back control from user programs
  - I/O interrupt from external device
    - Hitting Ctrl-C at the keyboard
    - Arrival of a packet from a network
    - Arrival of data from a disk

# Synchronous Exceptions

- Caused by events that occur as a result of executing an instruction: (the faulting instruction)
  - ***Traps***
    - Intentional. Procedure-like interface user program & kernel
    - Examples: ***system calls***, breakpoint traps, special instructions
    - Returns control to "next" instruction
  - ***Faults***
    - Unintentional but possibly recoverable
    - Examples: page faults (recoverable), protection faults (unrecoverable), floating point exceptions
    - Either re-executes faulting ("current") instruction or aborts
  - ***Aborts***
    - Unintentional and unrecoverable
    - Examples: illegal instruction, parity error, machine check
    - Aborts current program

# 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 |

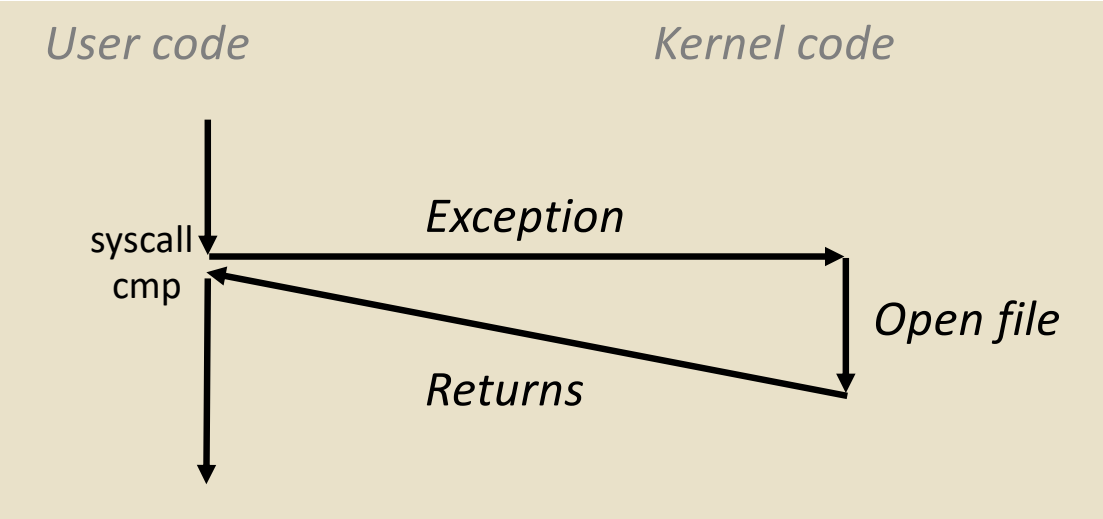
# Exceptions in x86-64 Systems

- From 0 to 31 defined by the Intel architects
- The rest defined by the operating system
- There are 256 different types

# System Call Example: Opening File

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

```
00000000000e5d70 <__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
```



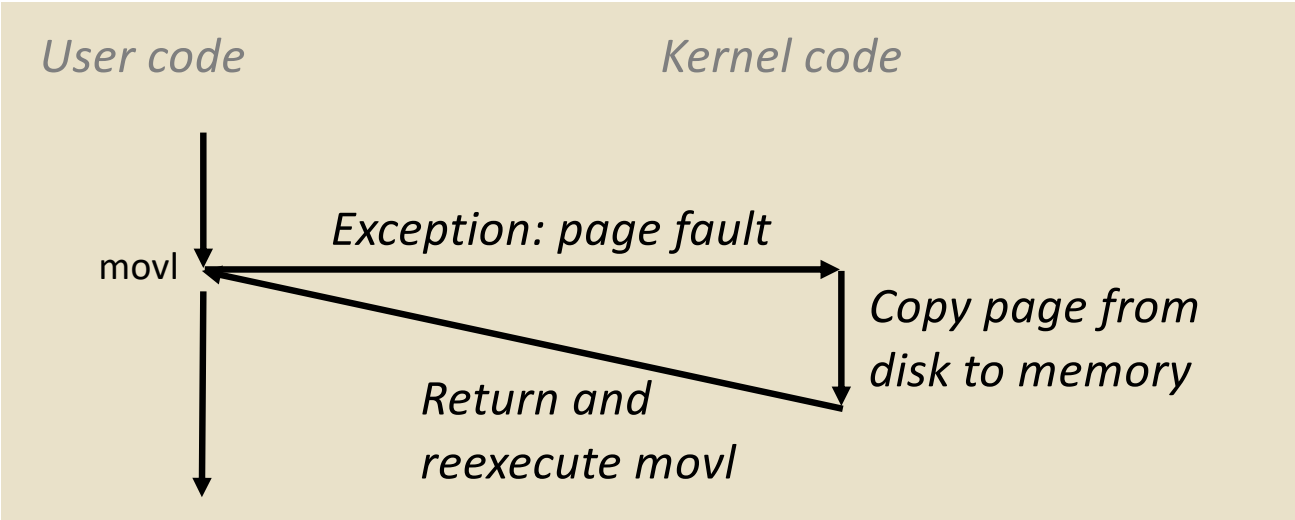

- `%rax` contains syscall number
- Other arguments in `%rdi`, `%rsi`, `%rdx`, `%r10`, `%r8`, `%r9`
- Return value in `%rax`
- Negative value is an error corresponding to negative `errno`

# Fault Example: Page Fault

- User writes to memory location
- That portion (page) of user's memory is currently on disk (Details covered in Virtual Memory)

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

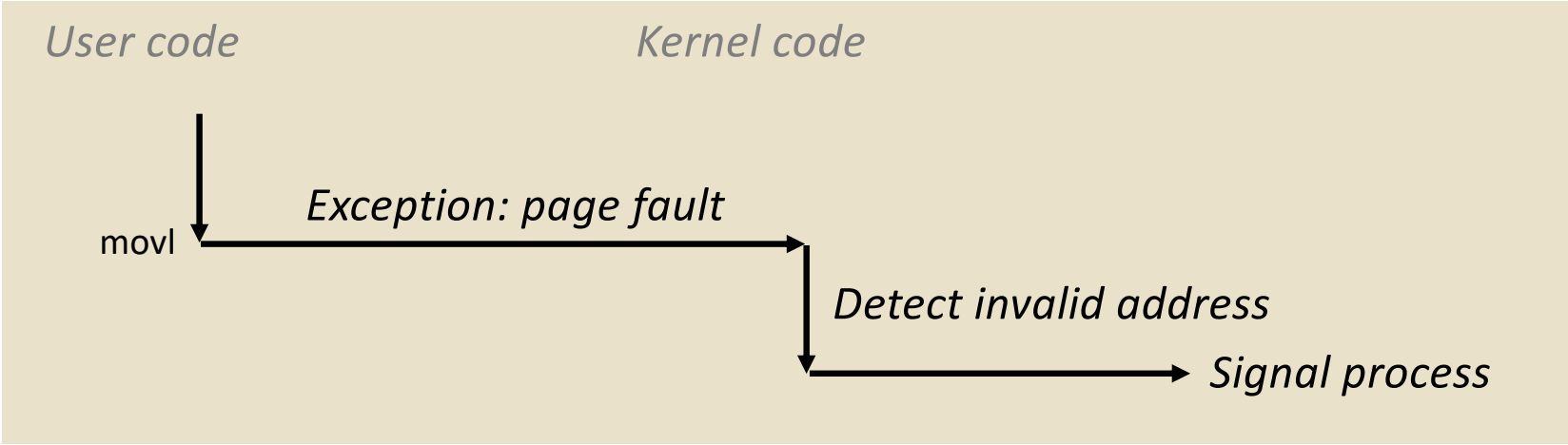
```
80483b7:   c7 05 10 9d 04 08 0d   movl    $0xd,0x8049d10
```



# Fault Example: 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 (Covered later with Signals)
- User process exits with "segmentation fault"

# Understanding ECF

1. ECF implements I/O, processes, and virtual memory
2. Interaction between applications and OS
3. Write interesting new application programs
4. Understand concurrency
5. Understand (high-level) software exceptions [Java and C++ try, catch, throw. C has non-local jumps] (later)

Understand interaction with the Operating System

# Review questions about Exceptions

- Exceptions
  - there is one type of *asynchronous exception*, what is it called? provide one example
  - three types of *synchronous exceptions*, what are their names?
  - do you remember the description of one of them?

Write answers to all of the above in your index card. Take 5 minutes to do this

# Topics

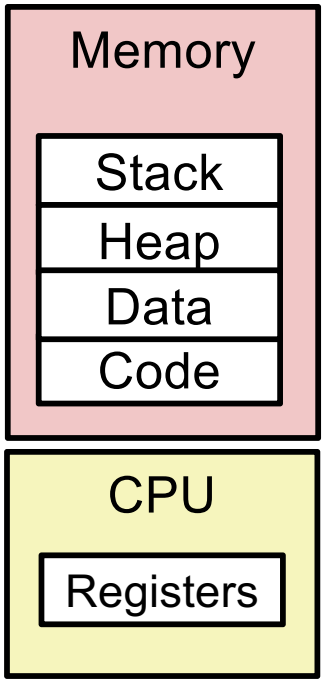
Exceptional Control Flow

- Exceptions
- The Process Abstraction
- Process Control

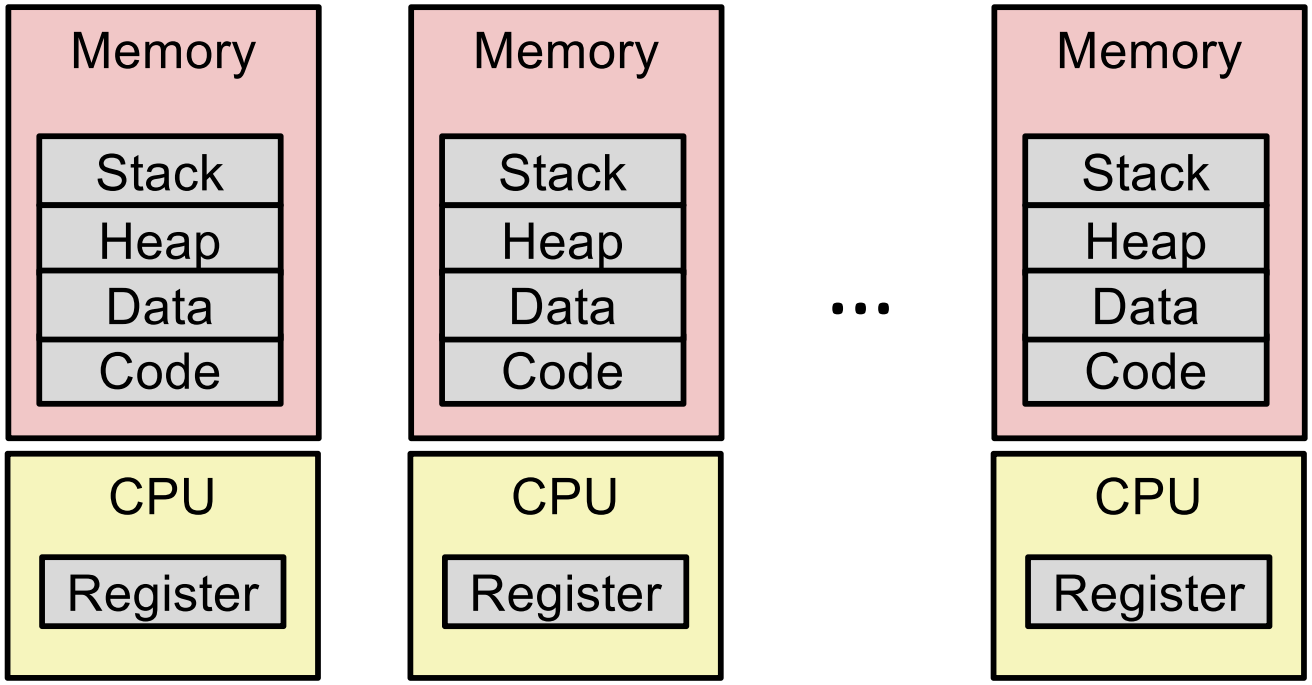
# Processes

- Definition: A *process* is "an instance of a program in execution."
  - 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*** (multiprocessing)
    - 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 (private address space)*



# 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

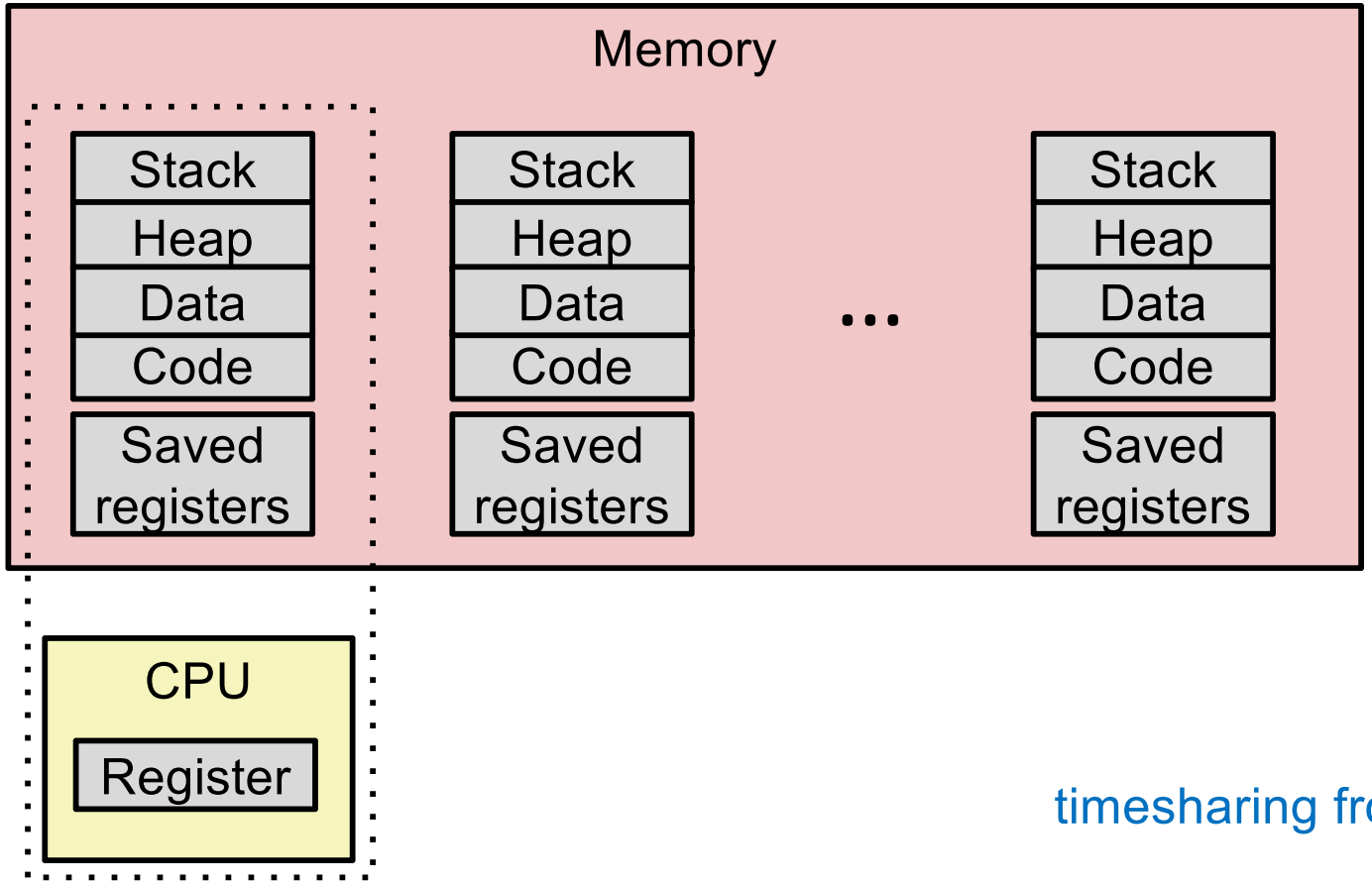
xterm

```
Processes: 123 total, 5 running, 9 stuck, 109 sleeping, 611 threads                11:47:07
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  ocspd        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  00:11.17 3     1    53    91    2464K  6148K  9976K  44M    2434M
50410  xterm        0.0  00:00.13 1     0    32    73    280K   872K   532K   9700K  2382M
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 **Process ID (PID)**

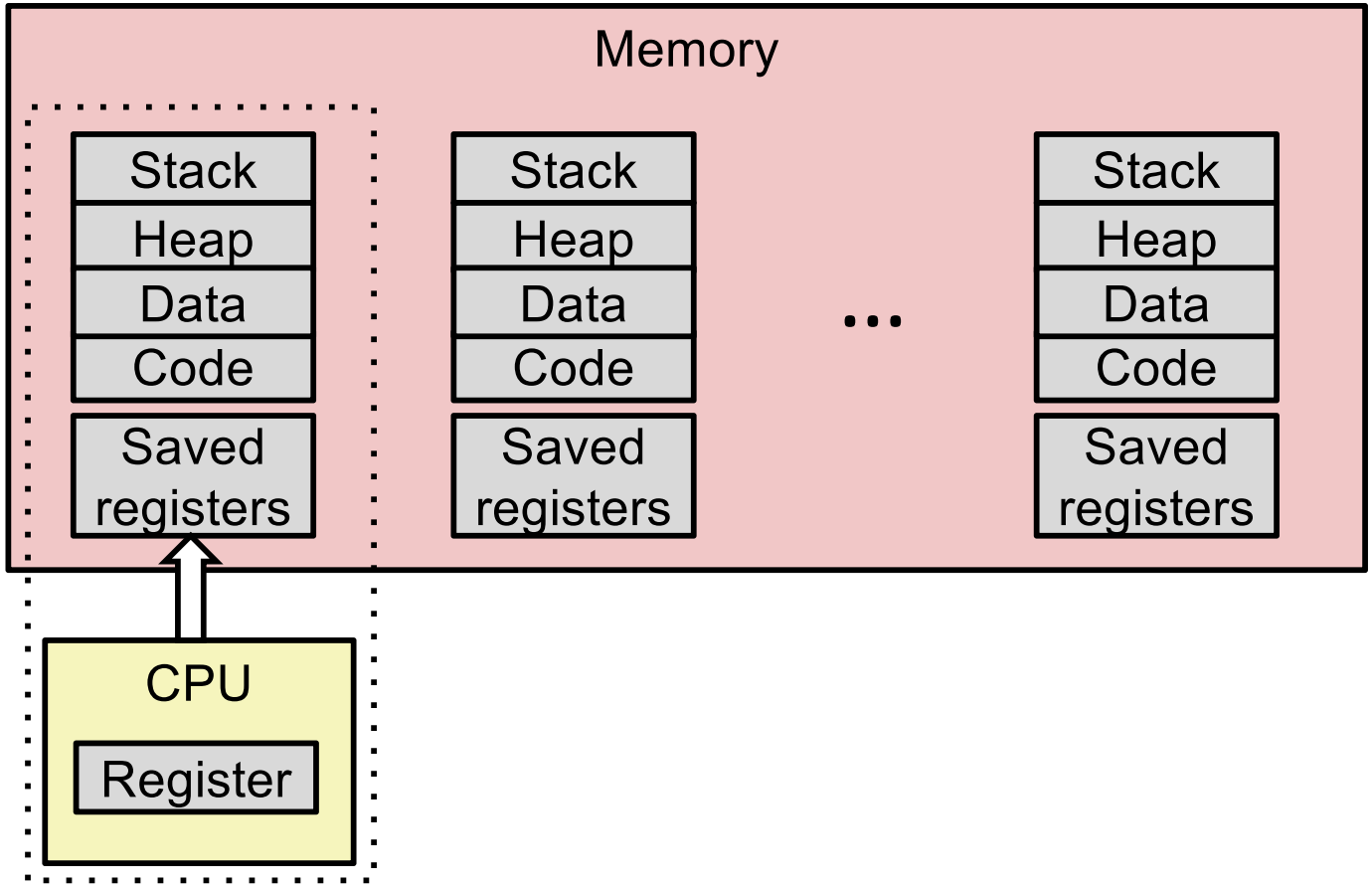
# Multiprocessing: The (Traditional) Reality




timesharing from the 70's

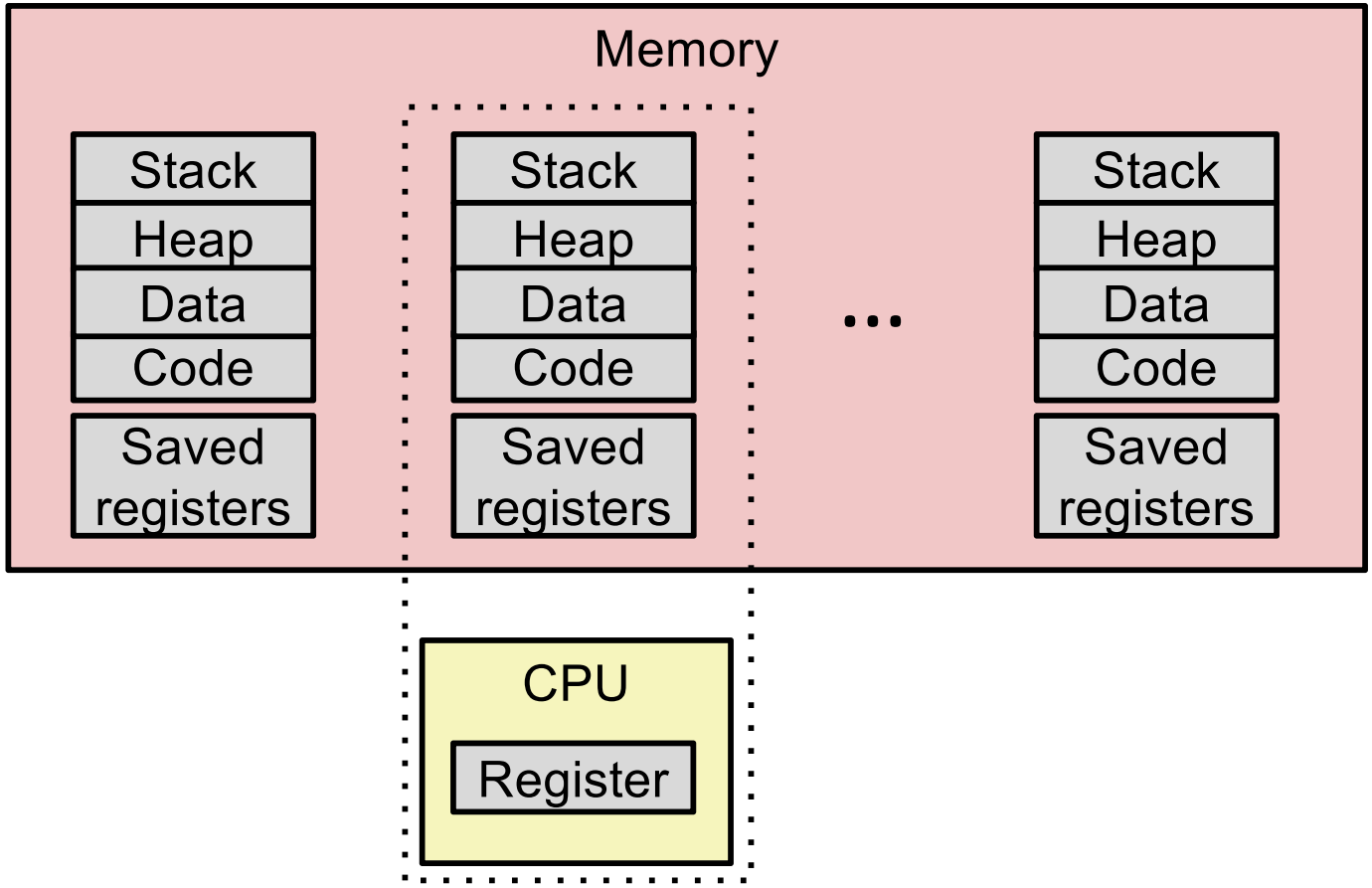
- Single processor executes multiple processes *concurrently* (not simultaneous)
  - Process executions interleaved (multitasking, time slices for each process)
  - Address spaces managed by virtual memory system (next in the course)
  - Register values for non-executing processes saved in memory

# Multiprocessing: The (Traditional) Reality



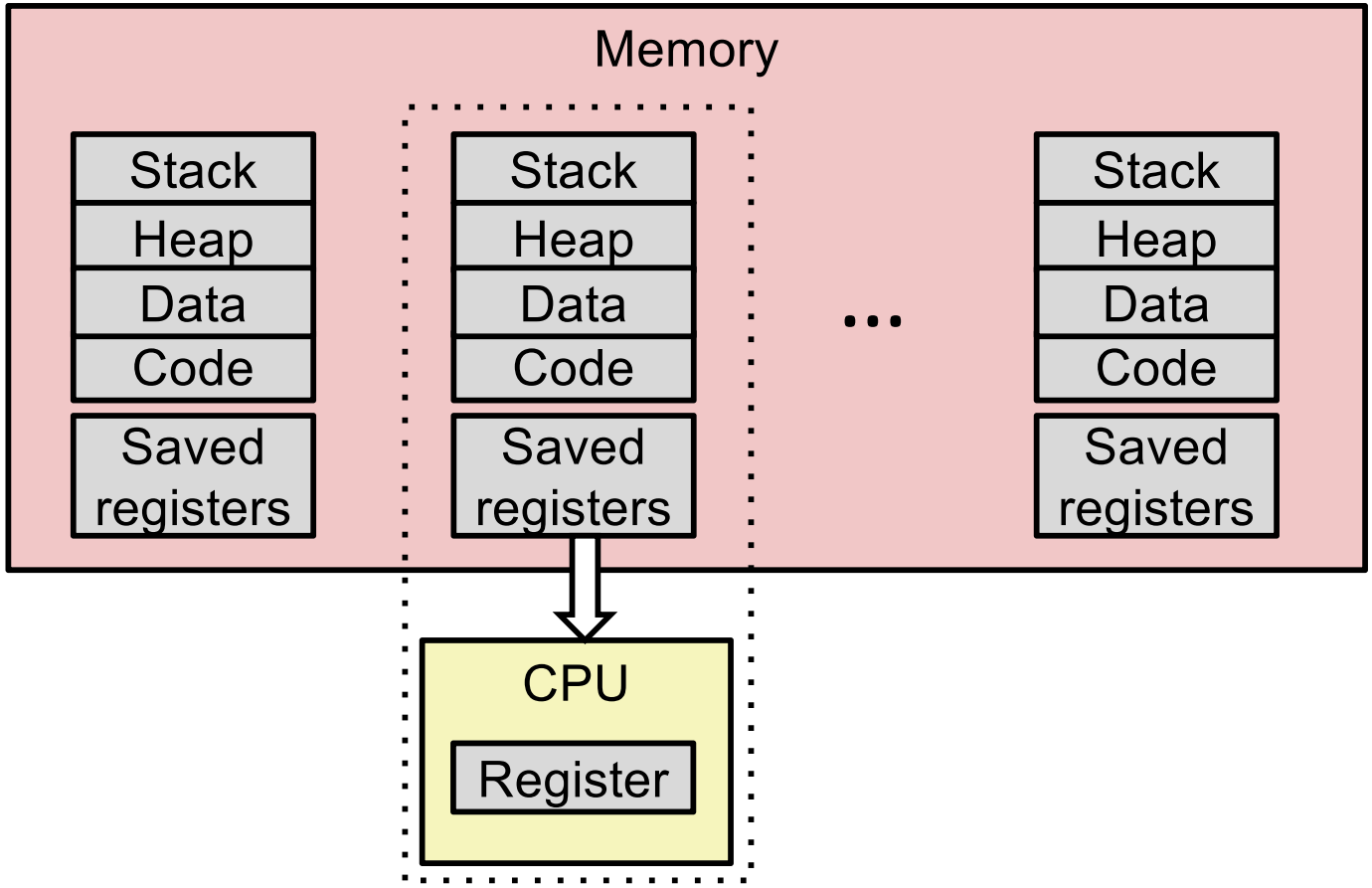

- Save current registers in memory

# Multiprocessing: The (Traditional) Reality



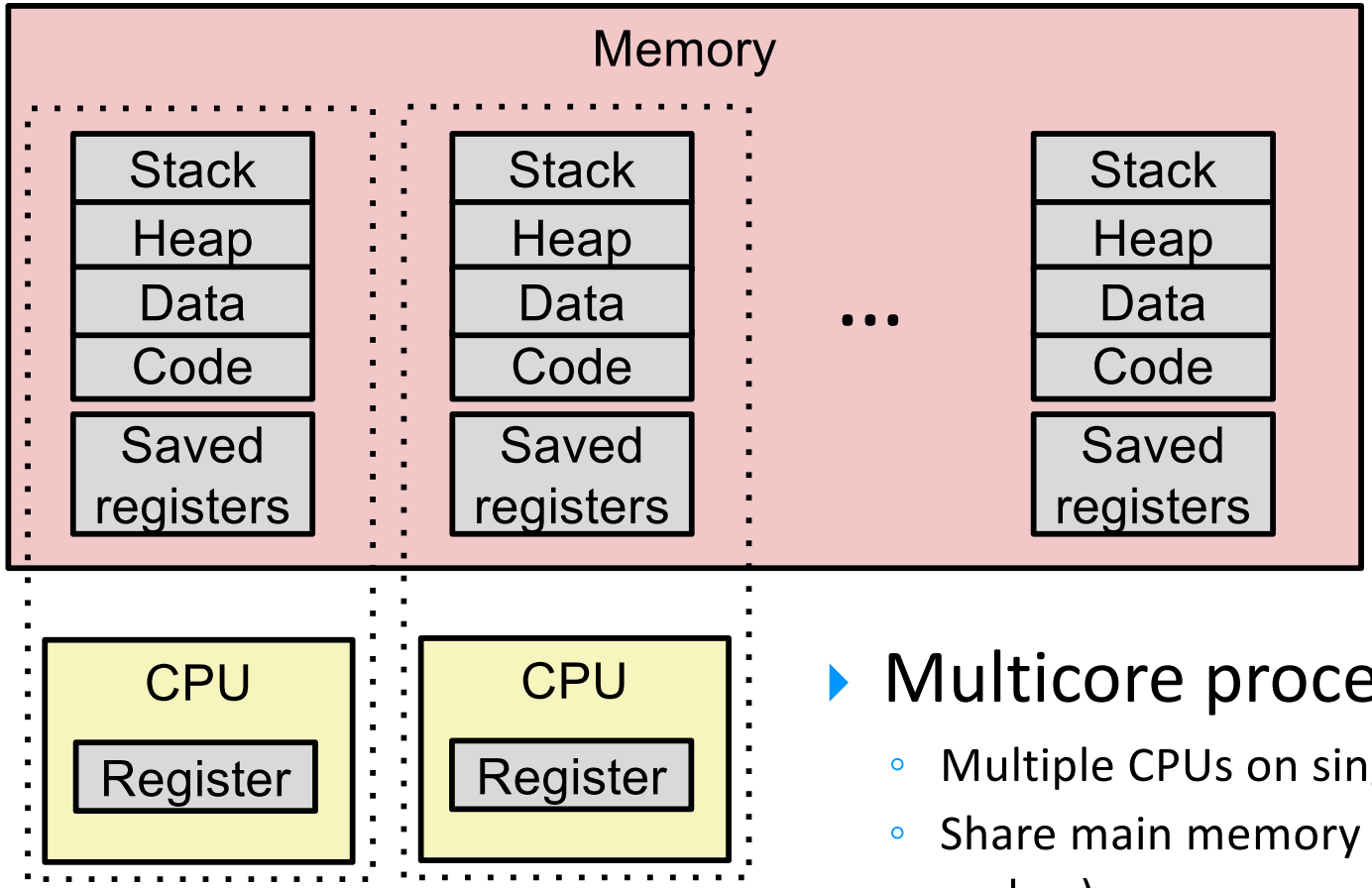

- Schedule next process for execution

# Multiprocessing: The (Traditional) Reality




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

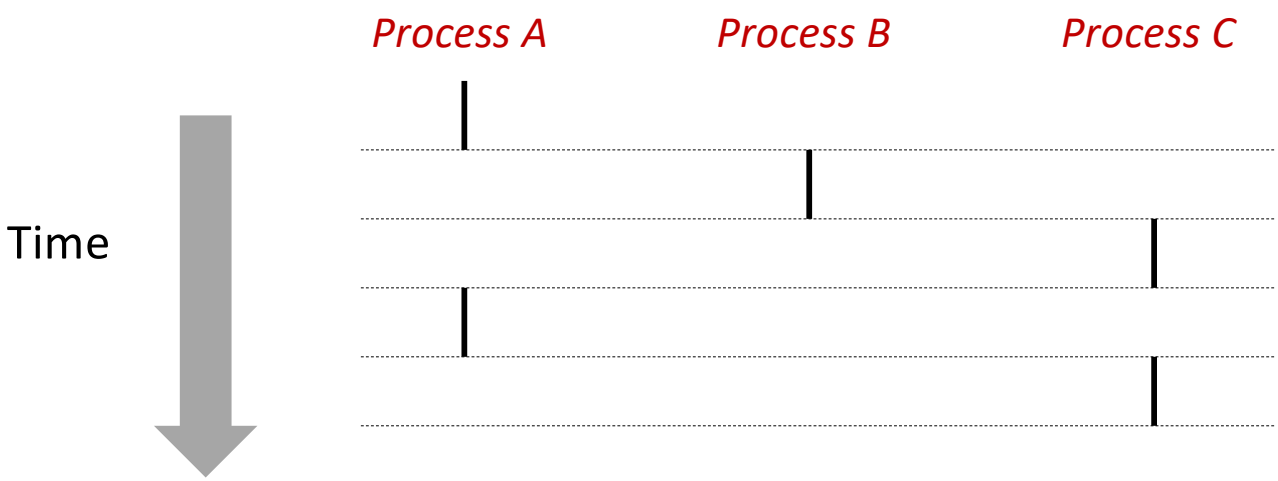
# Multiprocessing: The (Modern) Reality




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

# Concurrent Processes

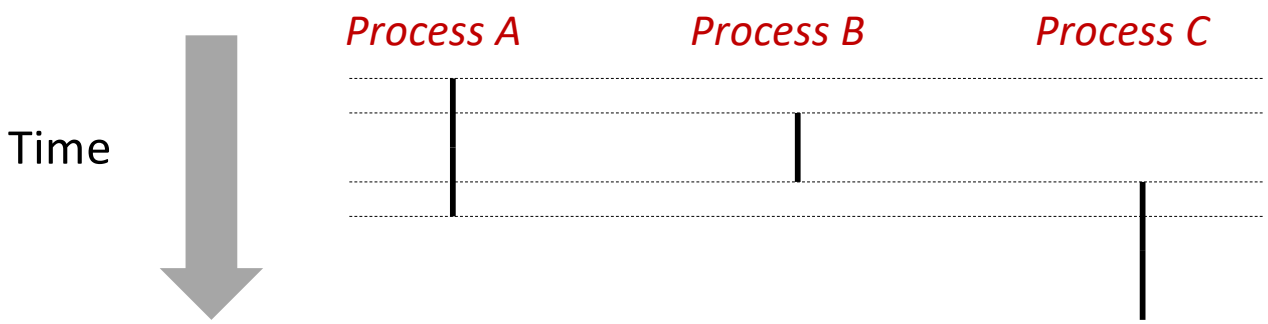
- Each process is a logical control flow.
- Two processes *run concurrently* (*are concurrent)* if their flows overlap in time
- Otherwise, they are *sequential (one after the other)*
- Examples (running on single core):
  - Concurrent: A & B, A & C
  - Sequential: B & C



# User View of Concurrent Processes

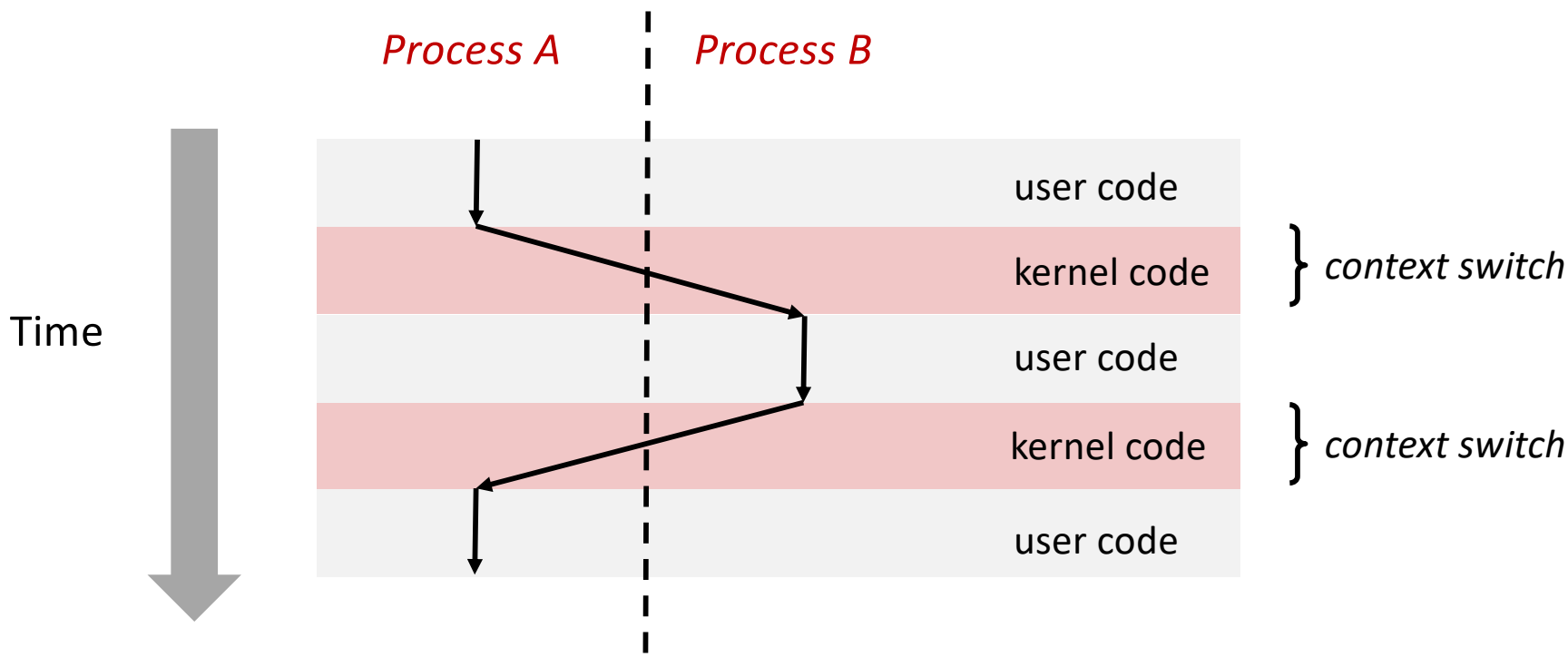
- Control flows for concurrent processes are physically disjoint in time

- However, we can think of concurrent processes as running in parallel with each other



# Context Switching (sec. 8.2.5)

- 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*



# The mechanism of context switching

1. save context of current process
2. restore saved context of previously preempted process
3. pass control to this newly restored process

Who does this?

# Topics

Exceptional Control Flow

- Exceptions
- The Process abstraction
- Process Control
  - error handling
  - system calls for manipulating processes:
    - creation and termination
    - destruction and reaping
    - loading and running programs
    - process commands and shells

# System Call Error Handling (sec. 8.3)

- 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: (with the fork system function)

```
if ((pid = fork()) < 0) {
    fprintf(stderr, "fork error: %s\n", strerror(errno));
    exit(0);
}
```

# Error-reporting functions

- Since some skip it because it makes the code hard to read
- 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 Wrappers

- Simplify the code presented 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;
}
```

Notice that the name of the wrapper is the same as the function but with capital F

```
  pid = Fork();
```

# Topics

## Exceptional Control Flow

- Exceptions
- The Process abstraction
- Process Control
  - error handling
  - system calls for manipulating processes:
    (provided by Unix to C programs)
    - creation and termination (Sec. 8.4.2)
    - destruction and reaping
    - loading and running programs
    - process commands and shells

# Obtaining Process IDs (sec. 8.4.1)

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

- `pid_t getppid(void)`
  - Returns PID of parent process (i.e. the process that created the calling process)

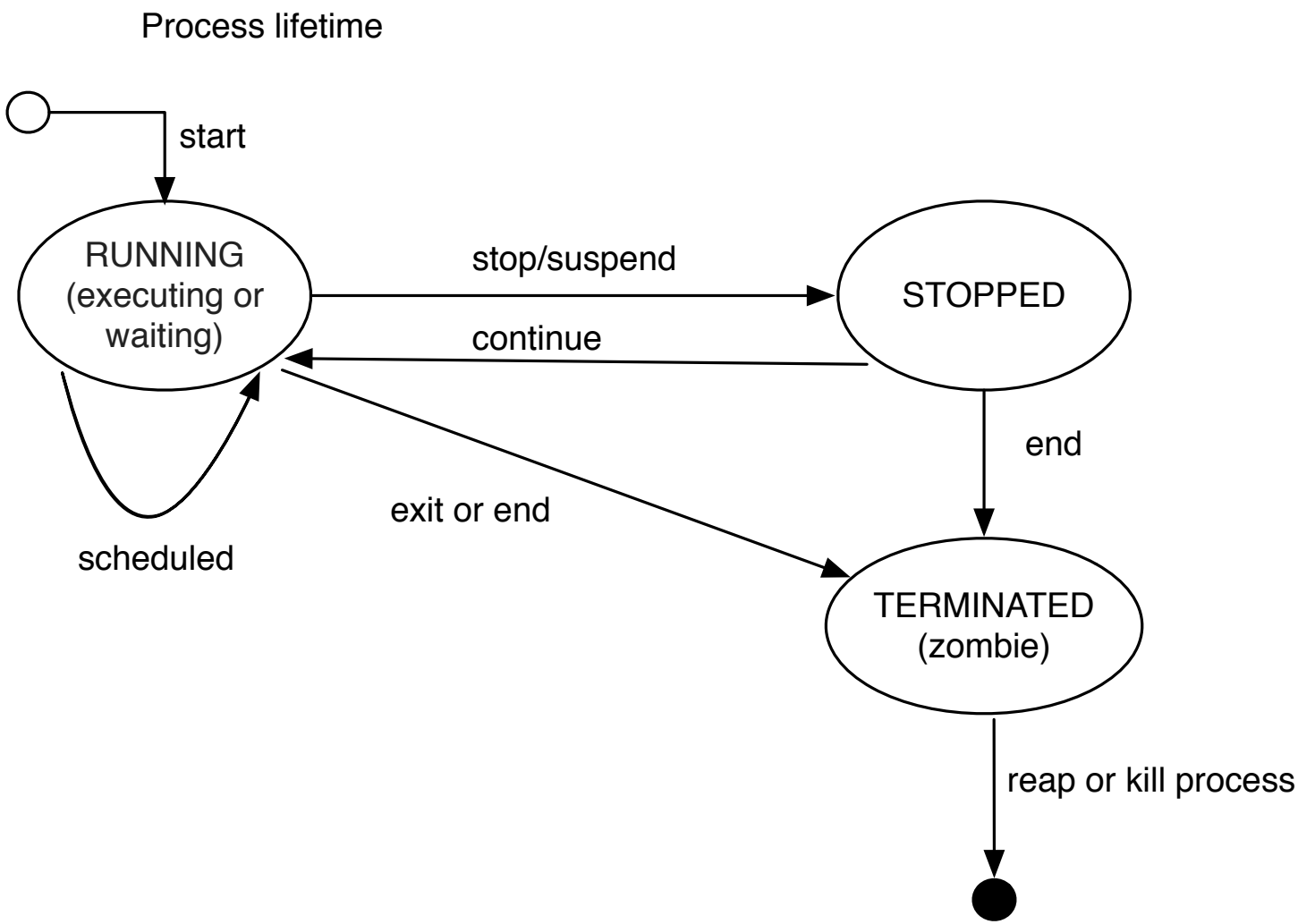
# Creating and Terminating Processes

## (section 8.4.2)

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 (we study signals to understand this better).
- Terminated
  - Process is stopped permanently . Signal to terminate, returning from main, or call exit(). (Next slide)

# Dynamic diagram of a *process*



# Terminating Processes

- Process becomes terminated for one of three reasons:
  - Receiving a signal whose default action is to terminate (signals 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 and nonzero on error
  - Another way to explicitly set the exit status is to return an integer value from the main routine

- `exit` function 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
  - Child is *almost* identical to parent:
    - Child gets an identical (but separate) copy of the parent's virtual address space.
    - Child gets identical copies of the parent's open file descriptors (Ch. 10: I/O)
    - Child has a different PID than the parent

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

# fork Example

```
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

# Practice Problem 8.2 (p. 743)

```
int main()
{
    int x = 1;
    if (Fork() == 0)
        printf("p1: x=%d\n",++x);
    printf("p2: x=%d\n",--x);
    exit(0);
}
```

Output of child process?

Output of parent process?

Process graph?

starts on slide 45