Exceptions

+ 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)

Control flow

Time ↓

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

+ Altering the Control Flow

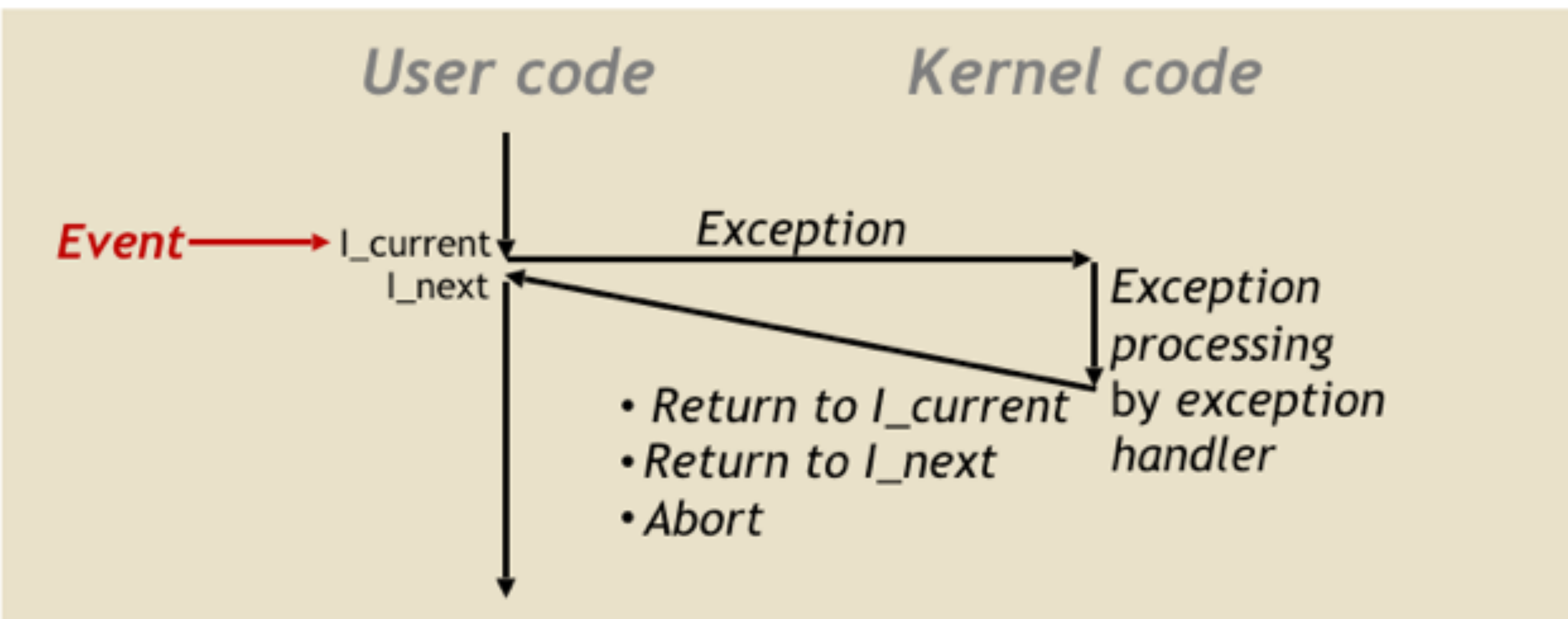
- **Up to now: two mechanisms for changing control flow:**
  - Jumps and branches
  - Call and return
  - Reactions to changes in *program state*
- **Insufficient for a useful system:**
  Difficult to react to changes in *system* state
  - Data arrives from a disk or a network adapter
  - Instruction divides by zero
  - System timer expires
- **System needs mechanisms for "exceptional control flow"**

+ Exceptional Control Flow

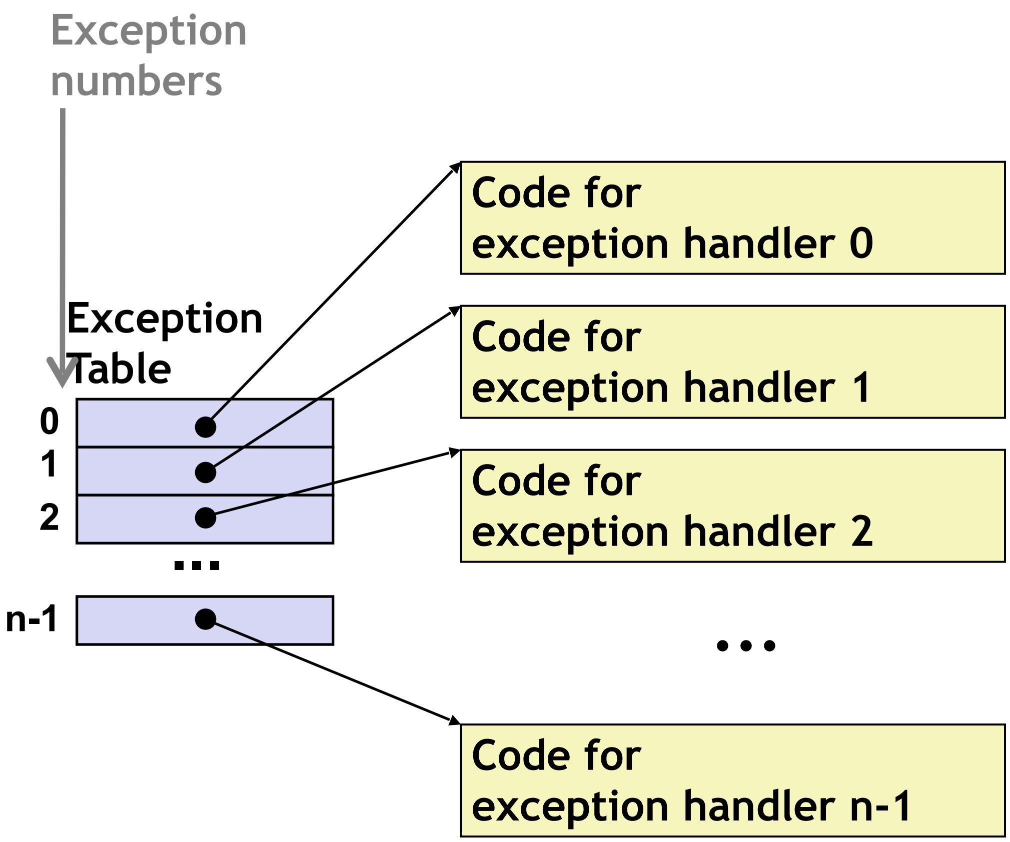
- **Exists at all levels of a computer system**
- **Low level mechanisms**
  - 1. *Exceptions*
    - Change in control flow in response to a system event (i.e., change in system state)
    - Implemented using combination of hardware and OS software
- **Higher level mechanisms**
  - 2. *Process context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

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, 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)
- 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**:
  - <u>*Timer interrupt*</u>
    - Every few ms, an external timer chip triggers an interrupt
    - Used by the kernel to take back control from user programs
  - <u>*I/O interrupt from external device*</u>
    - 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:**
  - *Traps*
    - Intentional
    - Example: *system calls*
    - Returns control to "next" instruction
  - *Faults*
    - Unintentional but possibly recoverable
    - Example: *page fault*
    - Either re-executes faulting ("current") instruction or aborts
  - *Aborts*
    - Unintentional and unrecoverable
    - Example: *illegal memory access*
    - Aborts current program

+ System Calls

Each x86-64 system call has a unique ID number

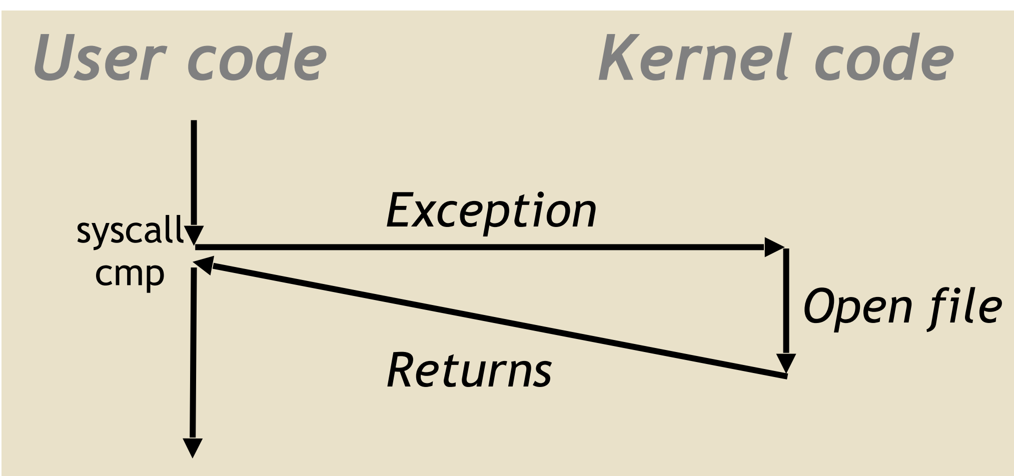
Examples:

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

System Call Example: Opening File

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

```
00000000000e5d70 <__open>:
...
e5d79:   b8 02 00 00 00       mov  $0x2,%rax   # 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
```



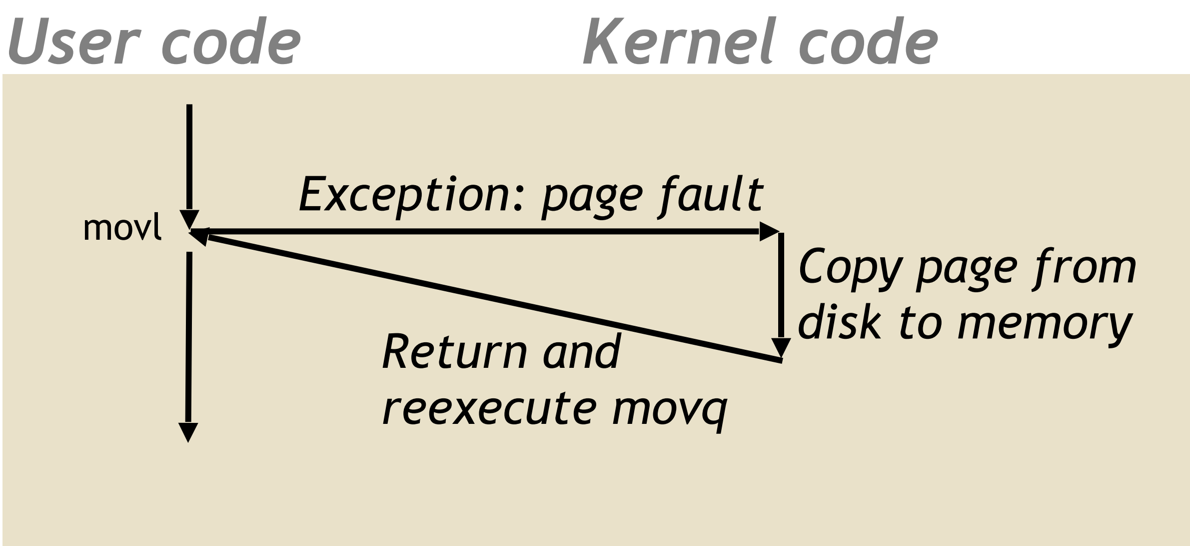
- `%rax` contains syscall number
- Other arguments in `%rdi`, `%rsi`, `%rdx`, `%r10`, `%r8`, `%r9`
- Return value in `%rax`
- Negative value is an error corresponding to negative errno

+ Fault Example: Page Fault

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

```
int a[10000];
main ()
{
    a[5000] = 13;
}
```

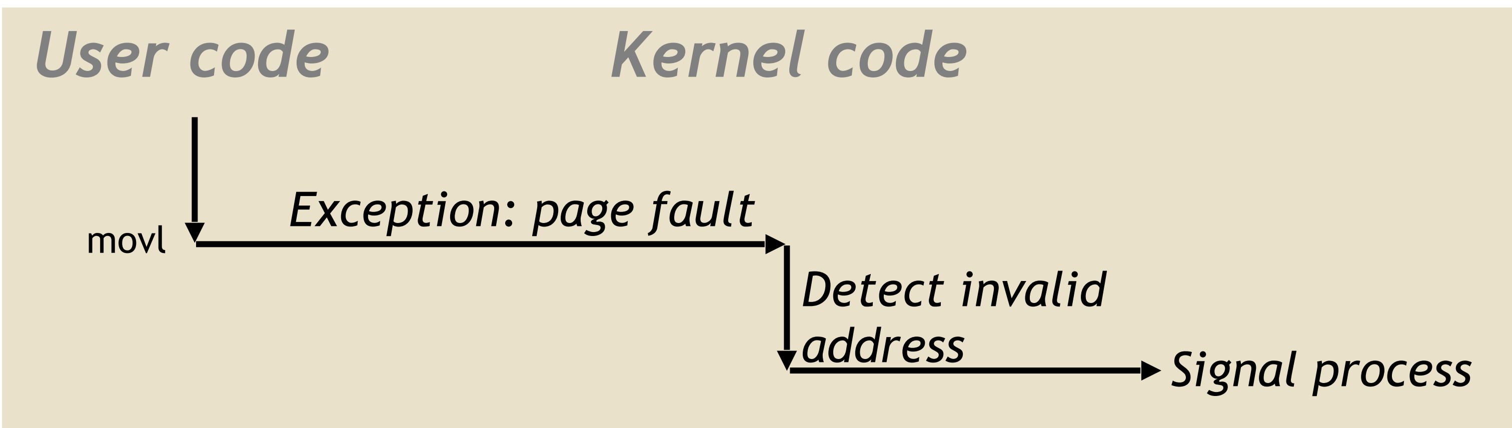
```
80483b7:      c7 05 10 9d 04 08 0d   movq   $0xd,0x8049d10
```



Fault Example: Invalid Memory Reference

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

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

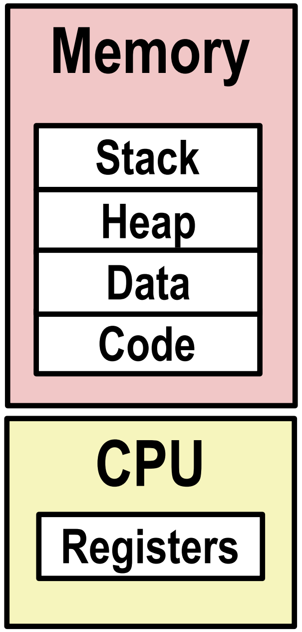


- Sends SIGSEGV signal to user process
- User process exits with “segmentation fault”

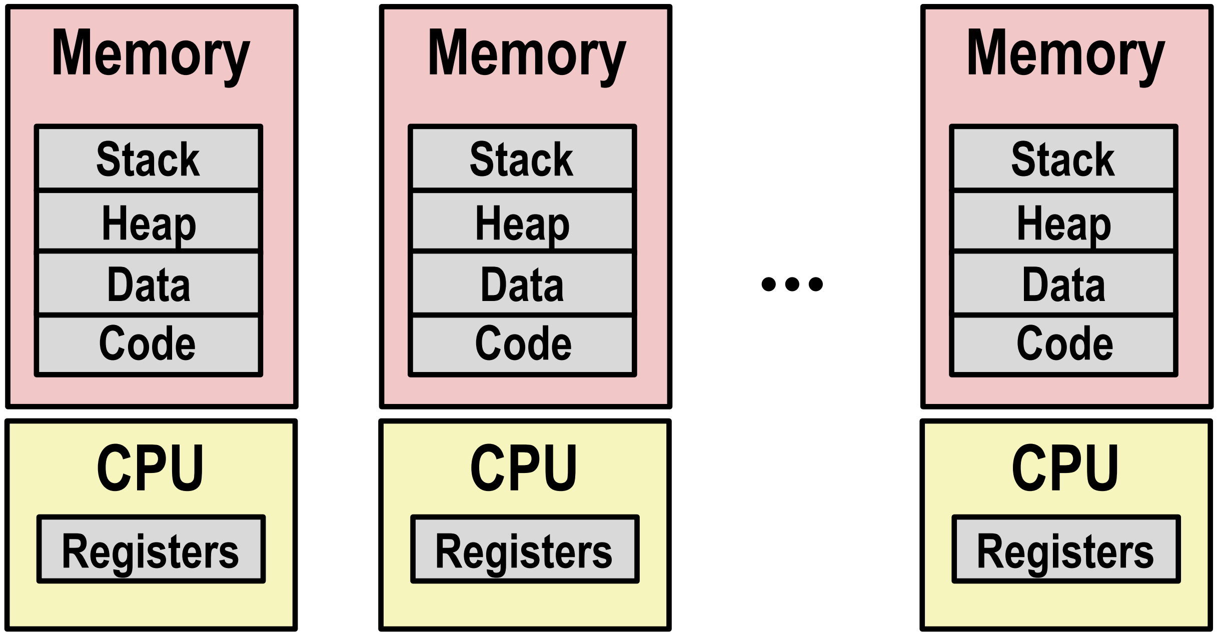
Processes

+ Processes

- **A *process* is an instance of a running program.**
  - One of the most successful ideas in computer science
  - Not the same as "program"
- **Process provided with two key abstractions by OS:**
  - *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*

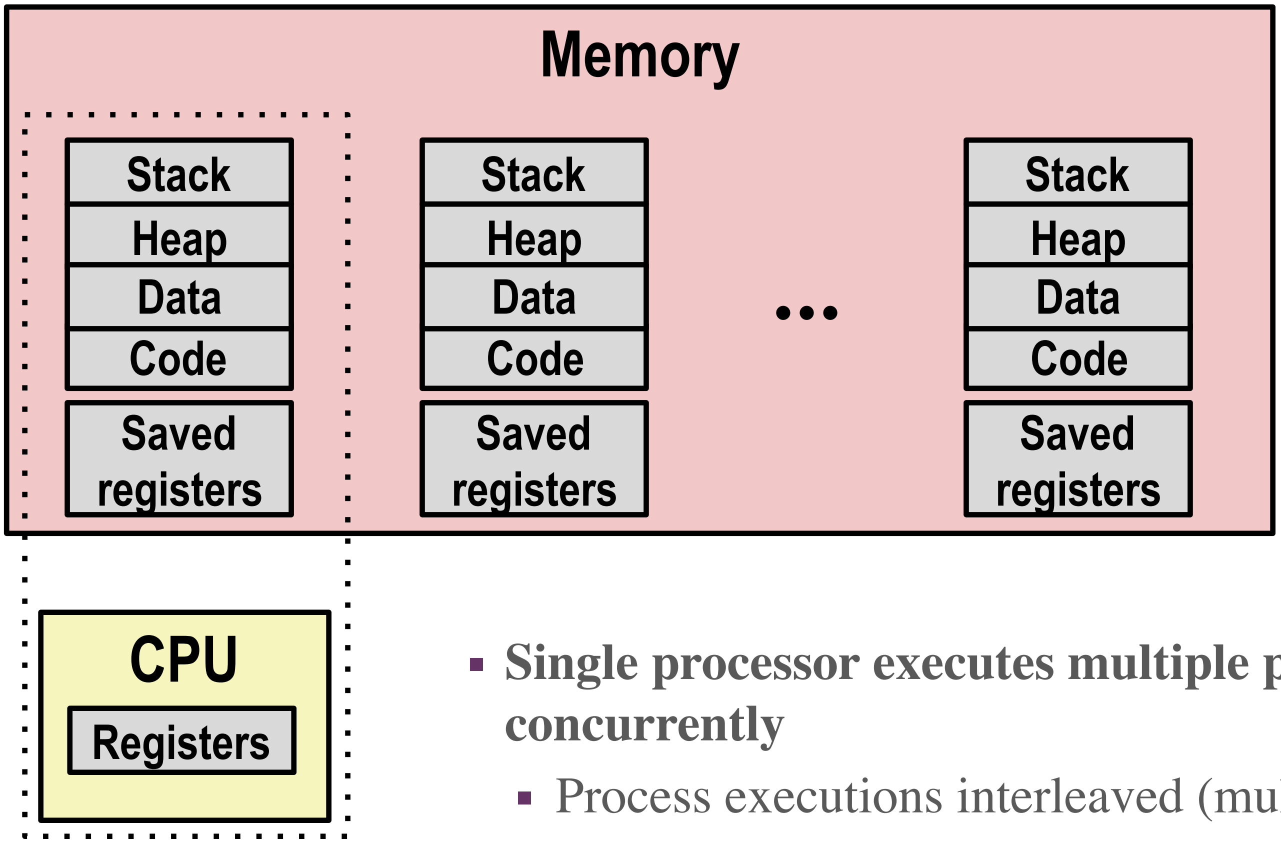


+ Multiprocessing: The Illusion



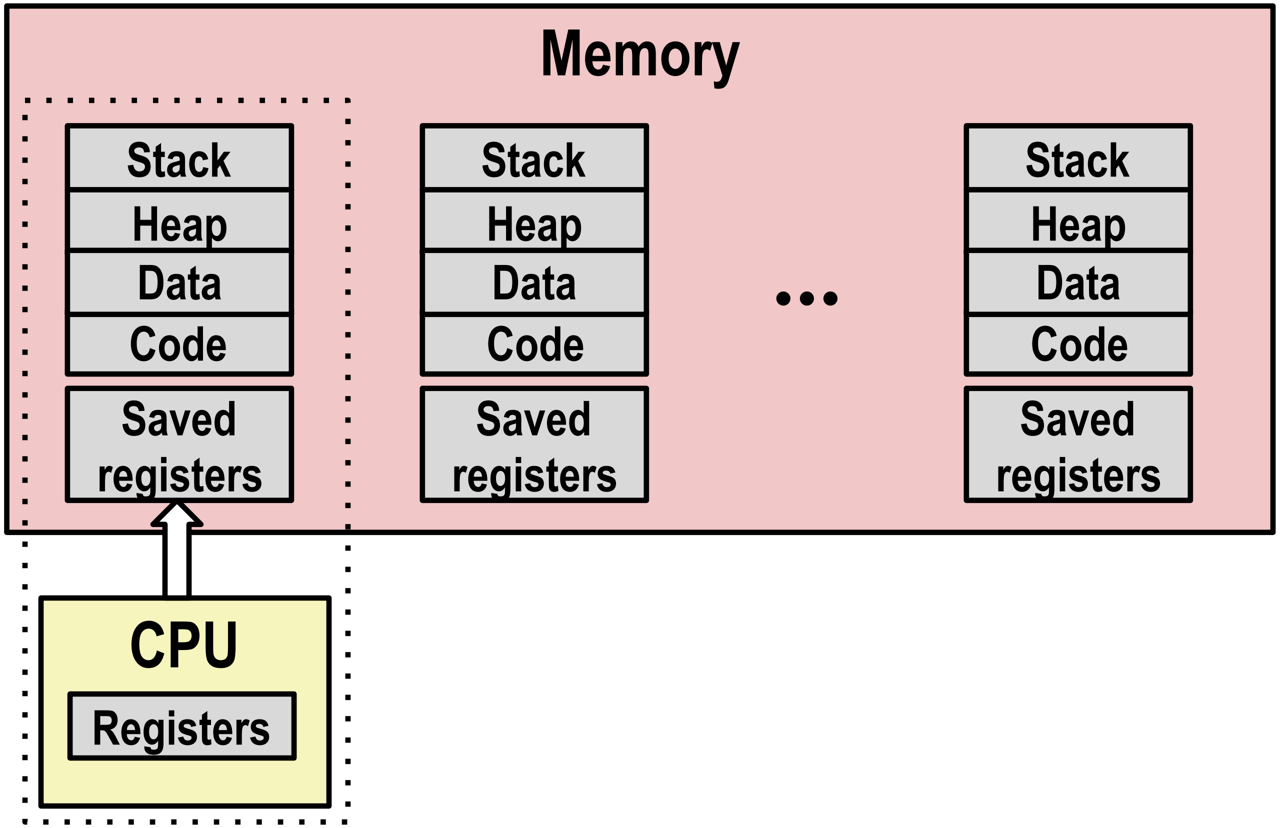
- **Computer runs many processes simultaneously**
  - Applications for one or more users
    - Web browsers, email clients, editors, …
  - Background tasks
    - Monitoring network & I/O devices

+ Multiprocessing: The (Traditional) Reality



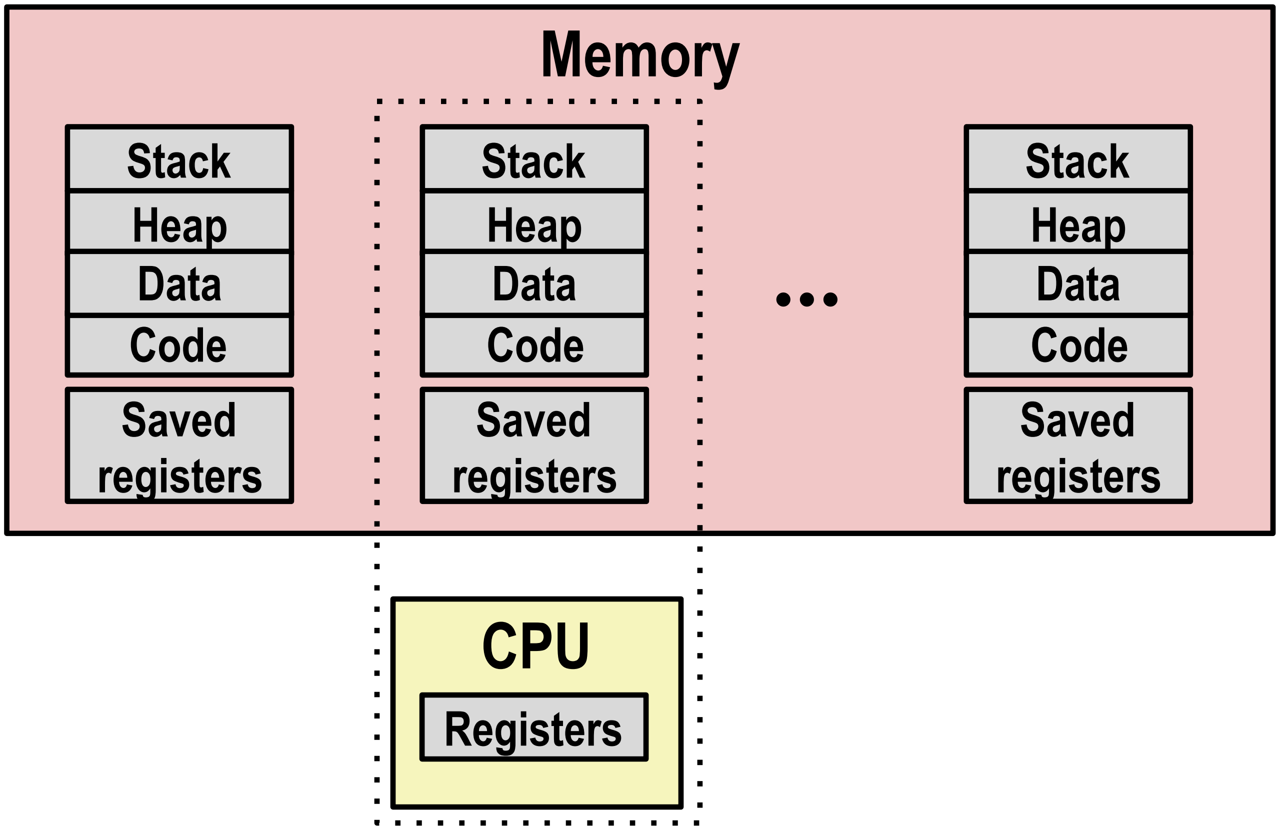
- **Single processor executes multiple processes concurrently**
  - Process executions interleaved (multitasking)
  - Address spaces managed by virtual memory system (later in course)
  - Register values for non-executing processes saved in memory (context)

Multiprocessing: The (Traditional) Reality



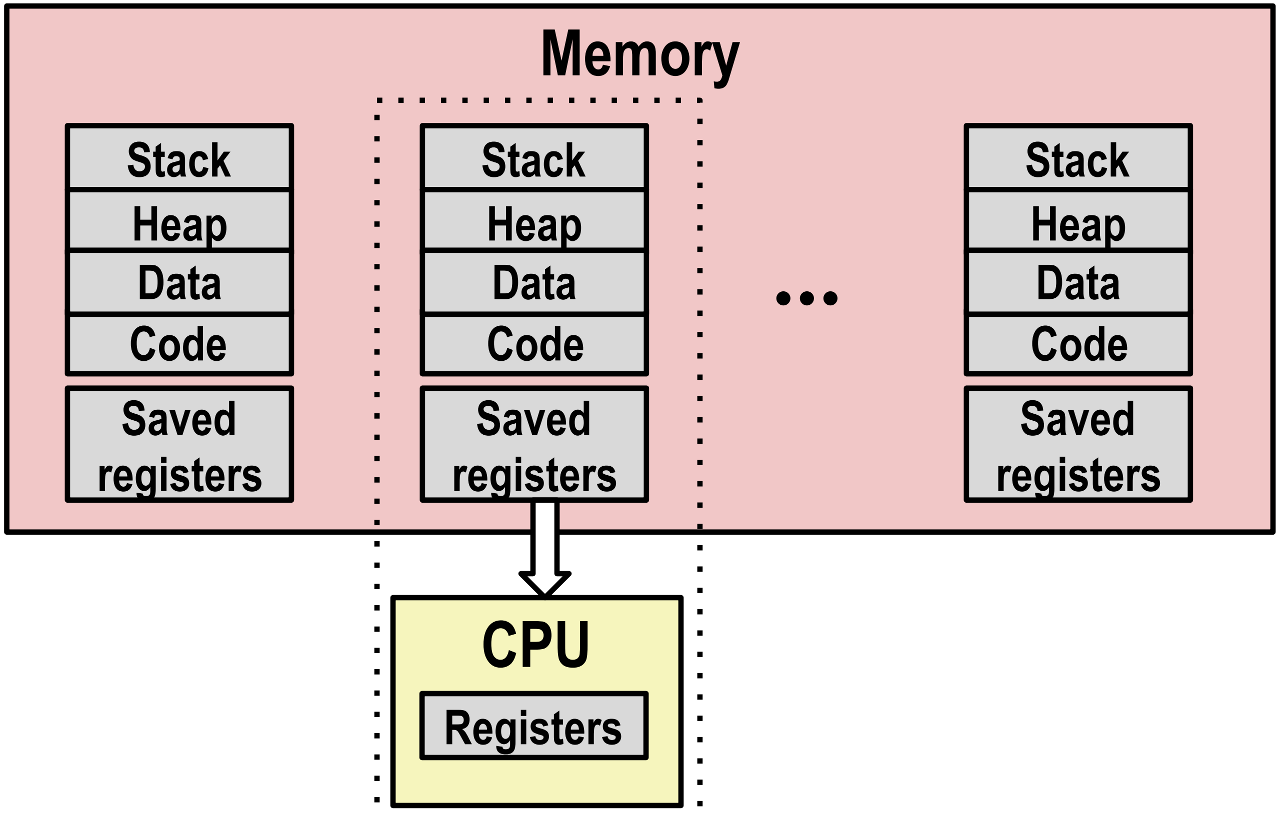
- Save current registers in memory

Multiprocessing: The (Traditional) Reality



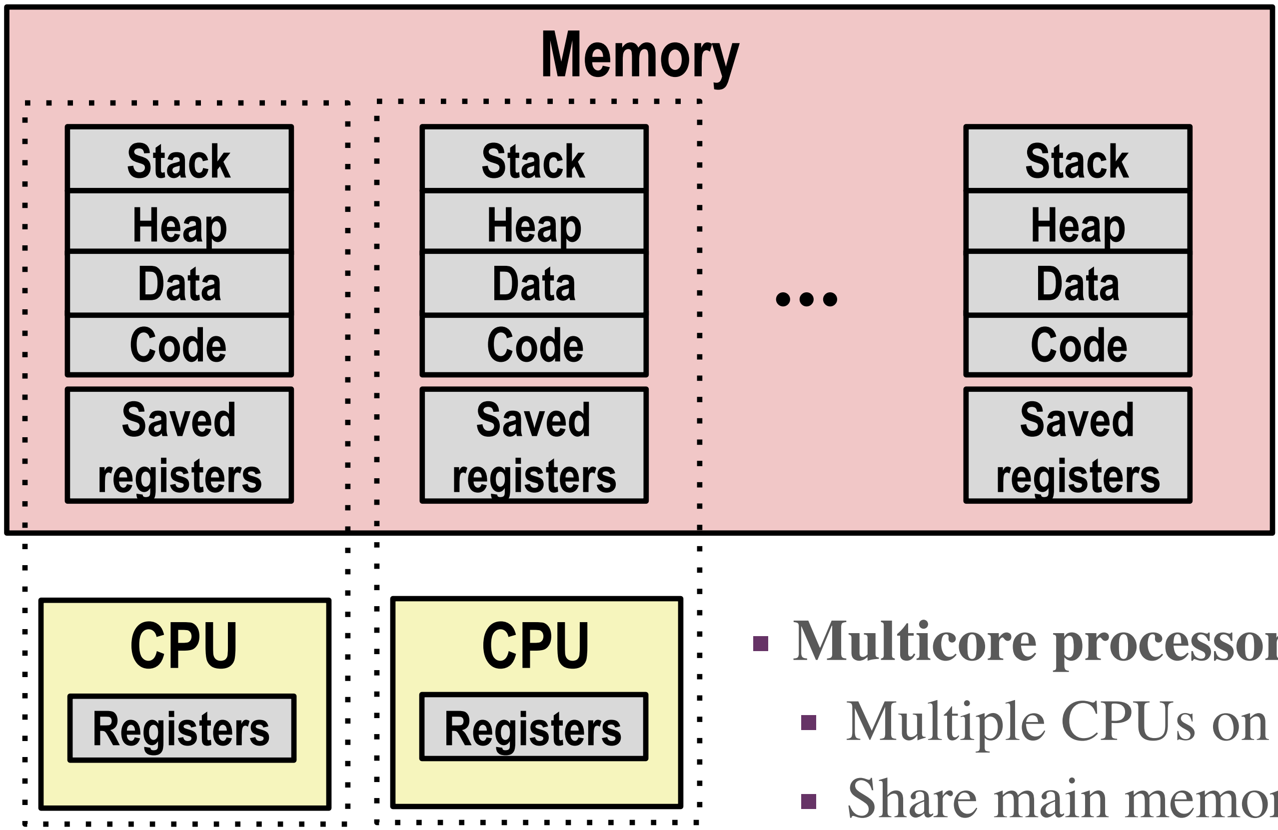
- Schedule next process for execution

+ Multiprocessing: The (Traditional) Reality



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

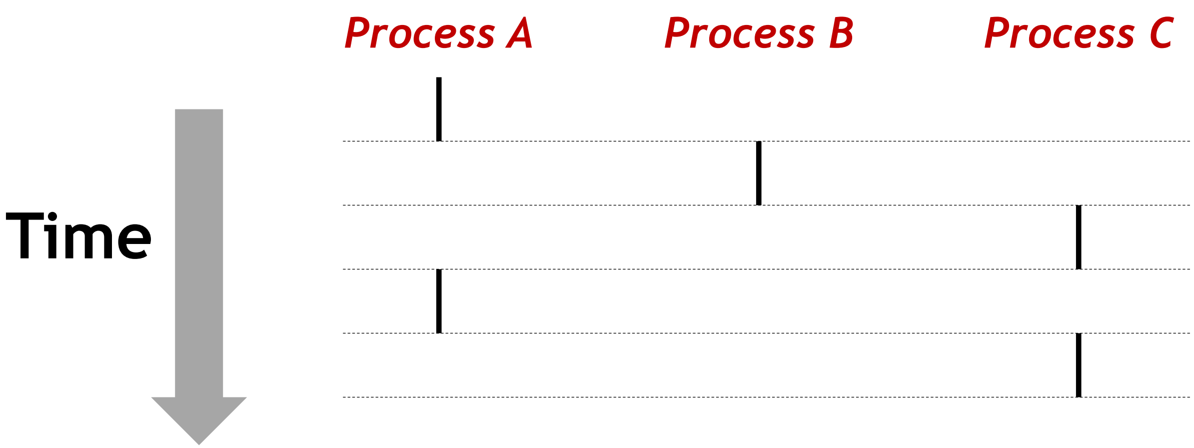
+ Multiprocessing: The (Modern) Reality



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

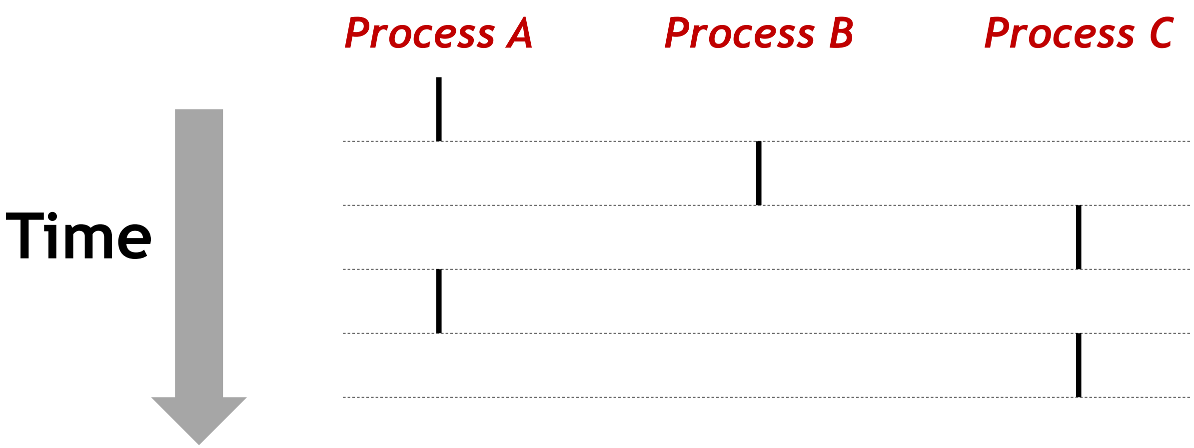
Concurrent Processes

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



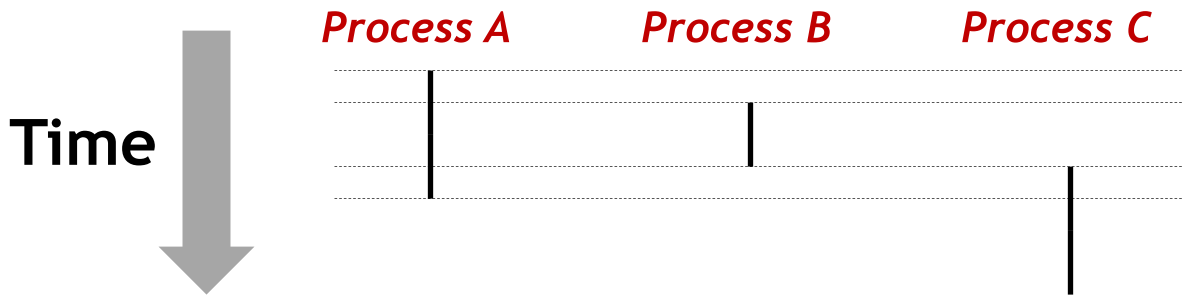
+ Concurrent Processes

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



+ Concurrent vs Parallel Processes

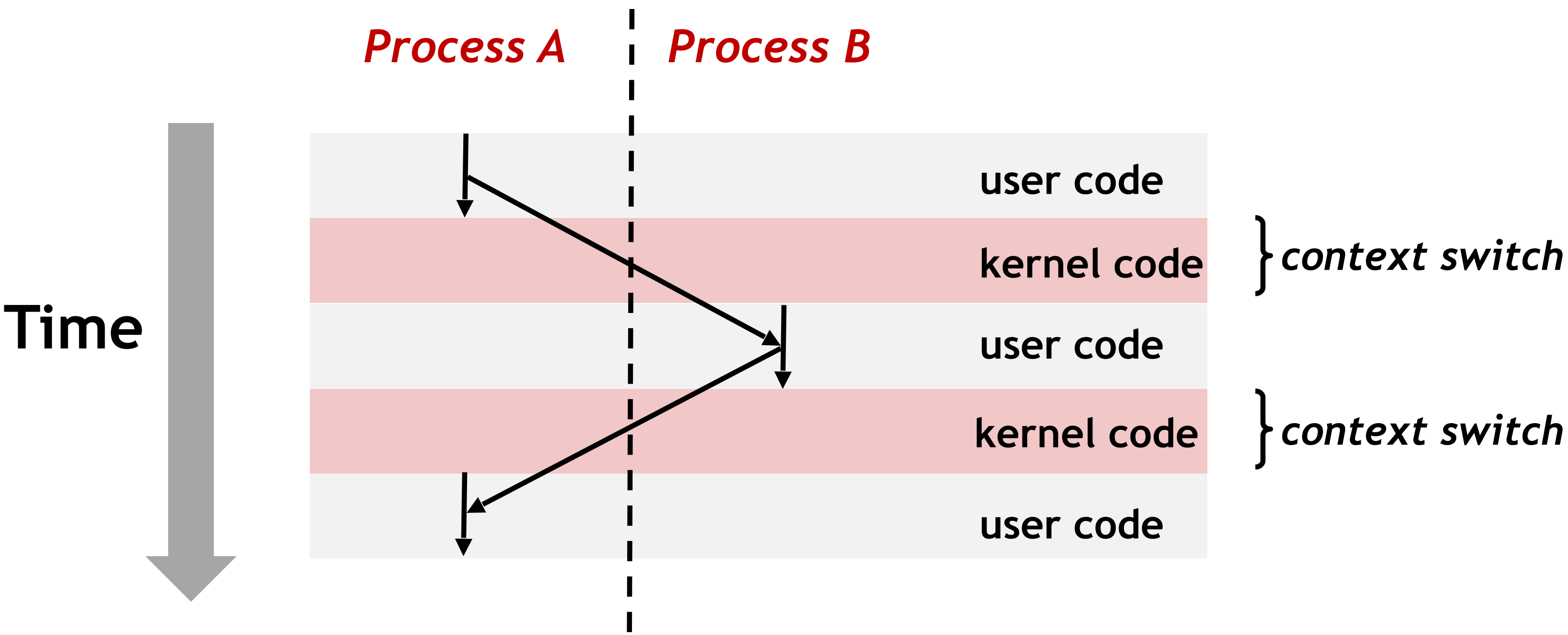
- **Control flows for concurrent processes are physically disjoint in time**
- **We can *think* of concurrent processes as running in parallel with each other, however, this is not necessarily the case.**
- **If two processes are executing simultaneously on different cores, these are true parallel processes.**



+ 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 all processes.

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



Process Control

+ 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
- **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

+ 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 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**

+ 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);
}
```

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

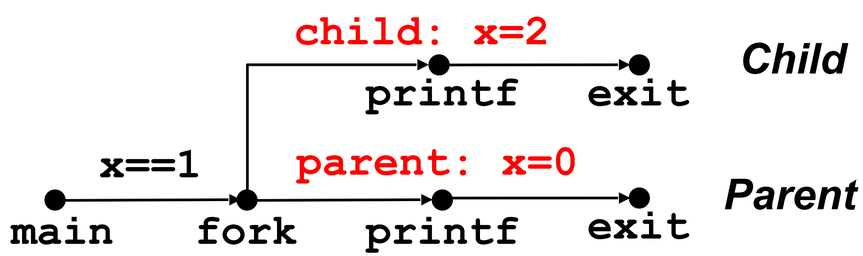
- **Call once, return twice**
- **Concurrent execution**
  - Can't predict execution order of parent and child
- **Duplicate but separate address space**
  - x has a value of 1 when fork returns in parent and child
  - Subsequent changes to x are independent

+ 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 in-edges
- **Any topological sort of the graph corresponds to a feasible total ordering.**
  - Total ordering of vertices where all edges point from left to right

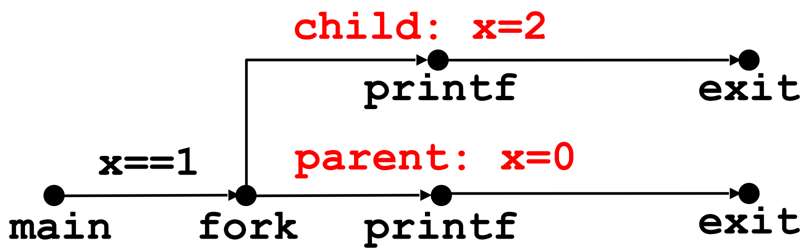
Process Graph Example

```
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);
}
```

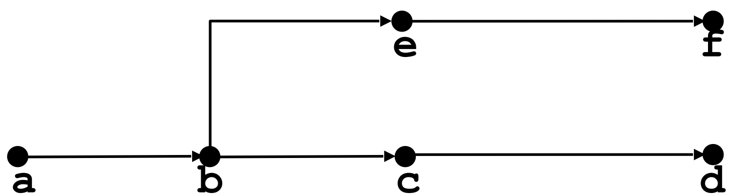


Interpreting Process Graphs

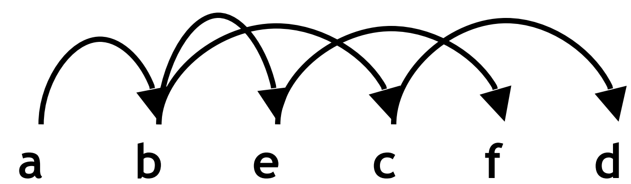
- **Original graph:**



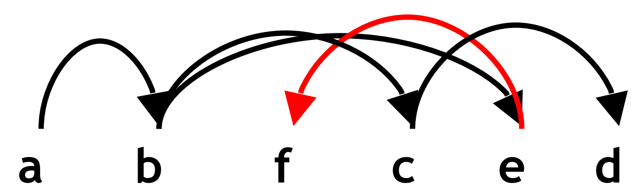
- **Re-labled graph:**



Feasible total ordering:

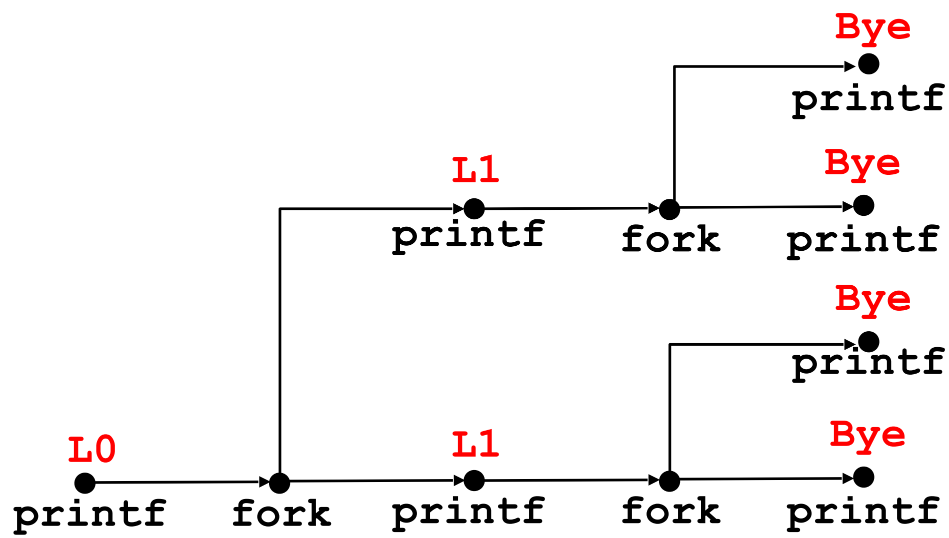


Infeasible total ordering:



+ fork Example: Two consecutive forks

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

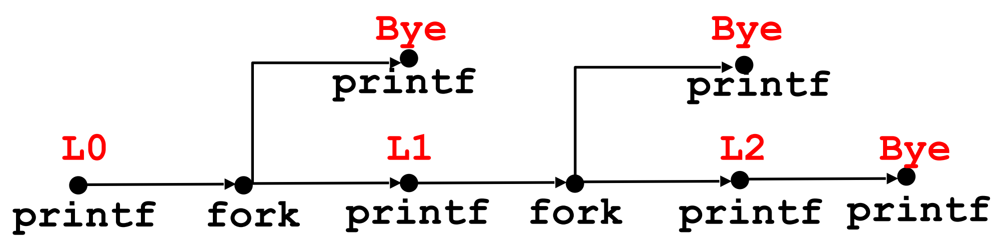


Feasible output:
L0
L1
Bye
Bye
L1
Bye
Bye

Infeasible output:
L0
Bye
L1
Bye
L1
Bye
Bye

+ `fork` Example: Nested `fork`s in parent

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



Feasible output:	Infeasible output:
L0	L0
L1	Bye
Bye	L1
Bye	Bye
L2	Bye
Bye	L2

+ Reaping Child Processes

- **Idea**
  - When process terminates, it still consumes system resources
  - Called a "**zombie**"
- **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 */
    }
}
```

```
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

```
void fork8()
{
    if (fork() == 0) {
        /* Child */
        printf("Child, PID = %d\n", getpid());
        while (1)
            ; /* Infinite loop */
    } else {
        printf("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 child explicitly, or else will keep running indefinitely