Exceptional Control Flow: Exceptions and Processes

B&O Readings: 8.1-8.4
CSE 361: Introduction to Systems Software

Instructor:
I-Ting Angelina Lee

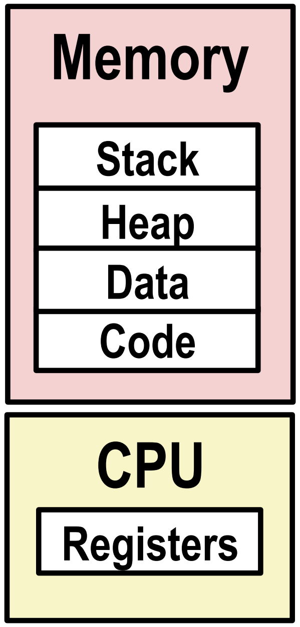
Today

- **Processes**
- Exceptions and Exceptional Control Flow
- Process Control

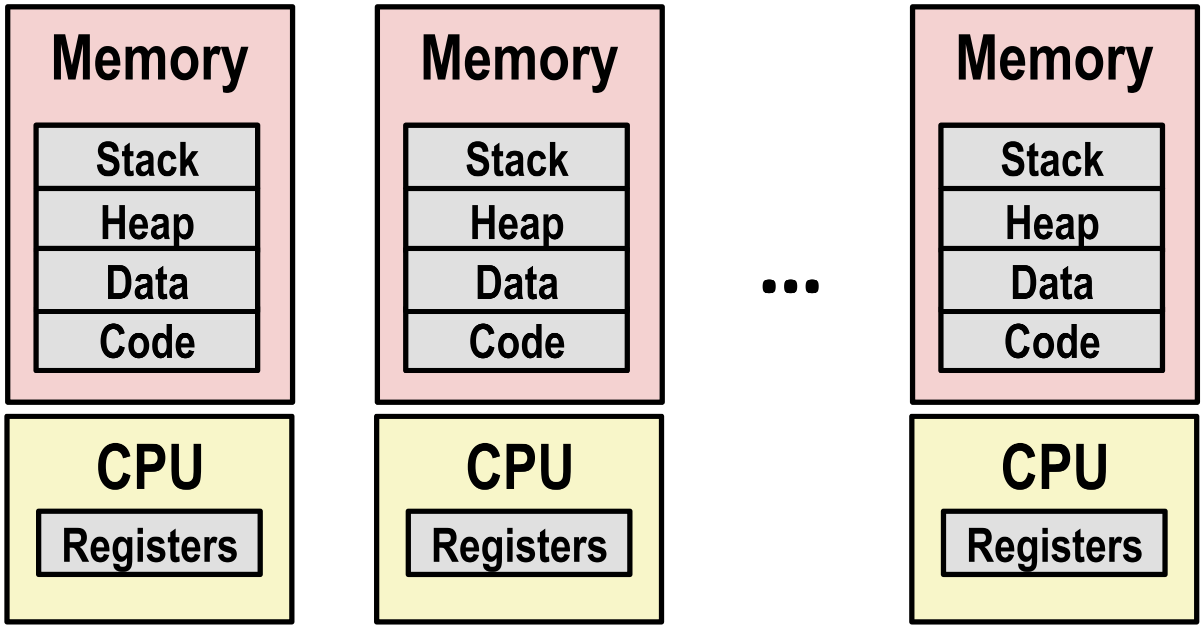
Processes

- **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
    - Provided by kernel mechanism called *context switching*
  - *Private address space*
    - Each program seems to have exclusive use of main memory.
    - Provided by 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

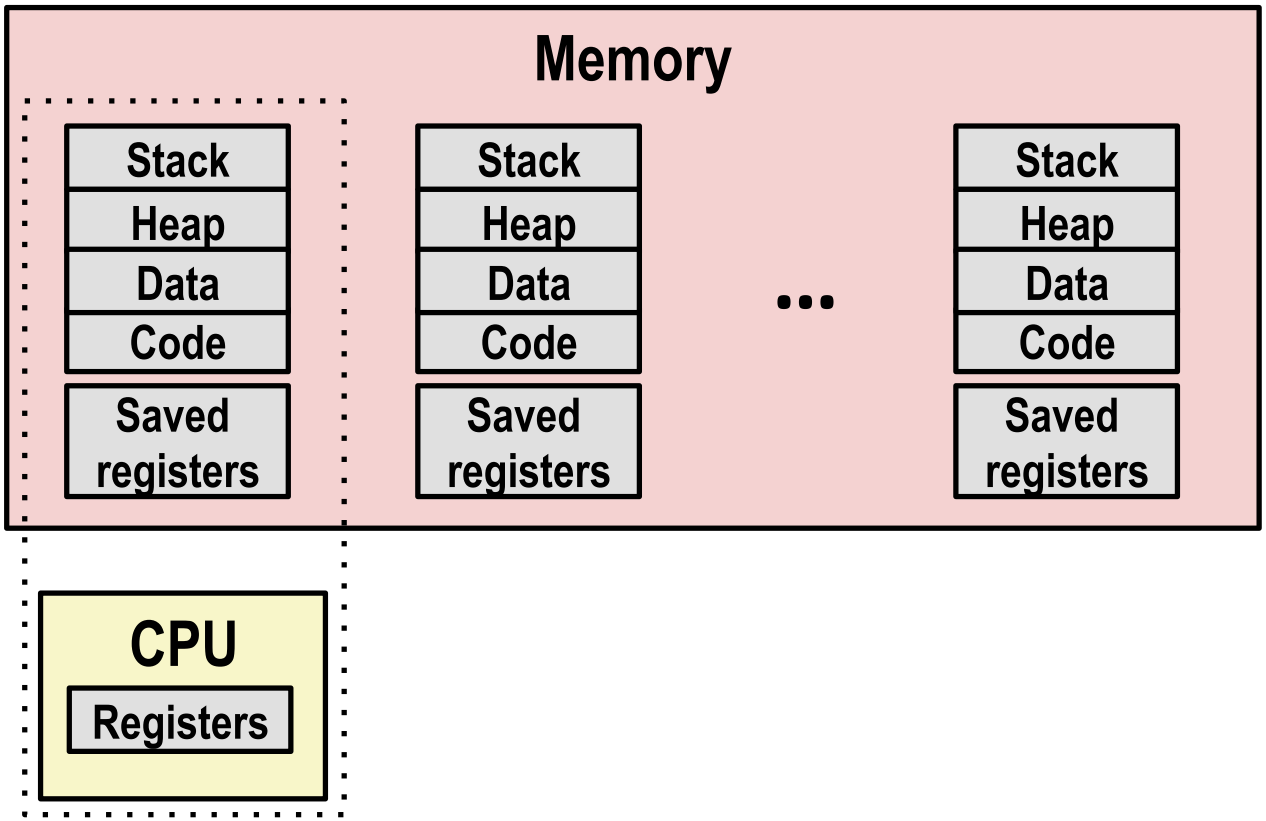
Multiprocessing Example

lectures — angelinas-air:fall17/lectures — top — 80×95

```
Processes: 302 total, 2 running, 18 stuck, 282 sleeping, 1734 threads  15:33:17
Load Avg: 1.51, 1.68, 1.66  CPU usage: 3.81% user, 4.29% sys, 91.88% idle
SharedLibs: 84M resident, 3824K data, 9968K linkedit.
MemRegions: 95589 total, 3714M resident, 72M private, 473M shared.
PhysMem: 8093M used (1873M wired), 96M unused.
VM: 828G vsize, 535M framework vsize, 12747121(0) swapins, 13411940(0) swapouts.
Networks: packets: 3970425/2599M in, 2213720/592M out.
Disks: 2478624/97G read, 1443105/92G written.

PID    COMMAND      %CPU TIME     #TH    #WQ   #PORT MEM    PURG   CMPRS  PGRP
20197  screencaptur 0.3  00:00.25 2      0     52    3436K  20K    0B     340
20194- CVMCompiler  0.0  00:00.23 2      1     27    9840K  0B     0B     20194
20192  VTDecoderXPC 0.0  00:00.16 2      0     39    6176K  0B     0B     20192
20191  mdworker     0.0  00:00.13 3      0     59    4980K  0B     0B     20191
20189  Google Chrom 0.0  00:02.37 13     0     106   65M    0B     0B     329
20188  top          3.5  00:05.17 1/1    0     26    4020K  0B     0B     20188
20174  cupsd        0.0  00:00.30 3      0     43    9244K  0B     0B     20174
20169- mdworker32   0.0  00:01.07 3      0     54    7052K  0B     0B     20169
20168  QuickLookSat 0.0  00:01.09 7      1     93    21M    496K   0B     20168
20167  quicklookd   0.0  00:00.59 4      0     93    7900K  0B     2264K  20167
20150  com.apple.Co 0.0  00:00.03 2      1     23    52K    0B     1936K  20150
```

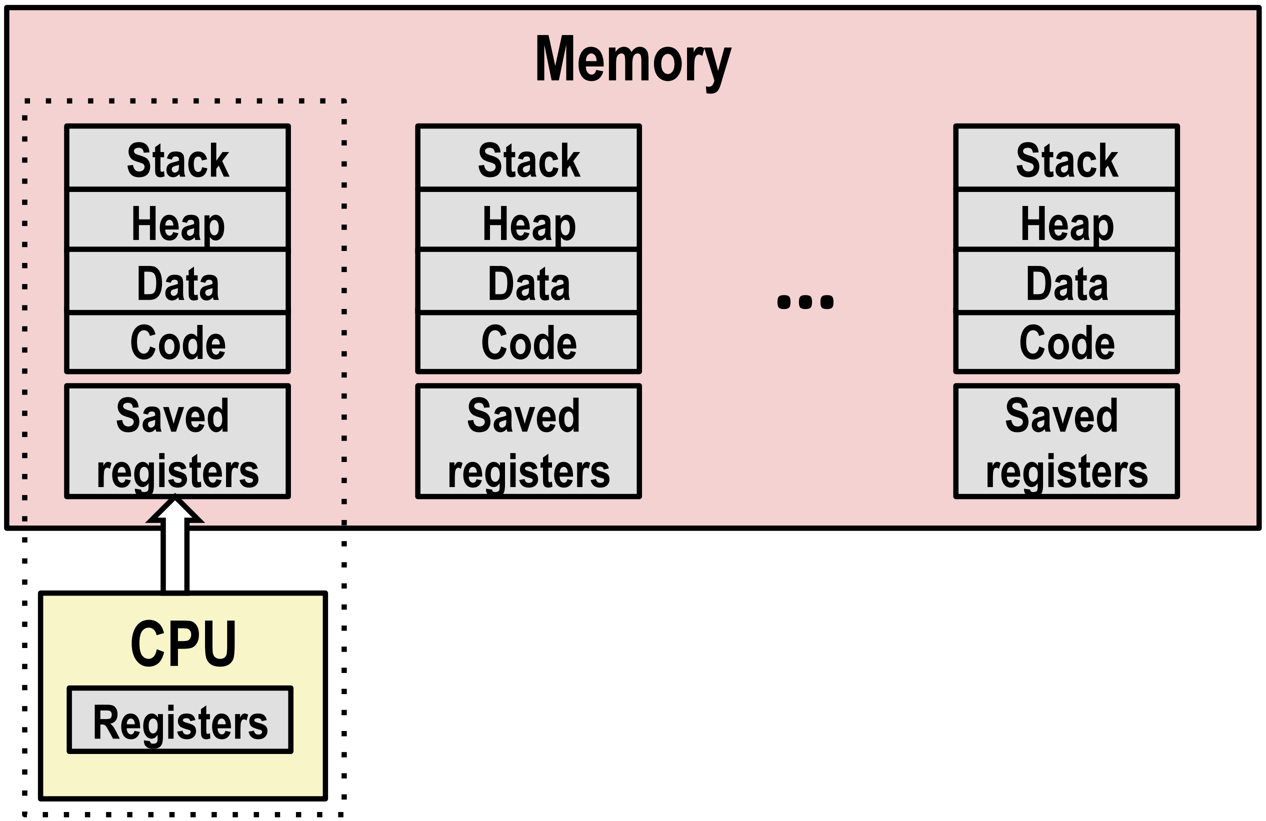
- **Running program “top” on Mac**
  - System has 302 processes, 2 of which are active
  - Identified by Process ID (PID)

Multiprocessing: The (Traditional) Reality



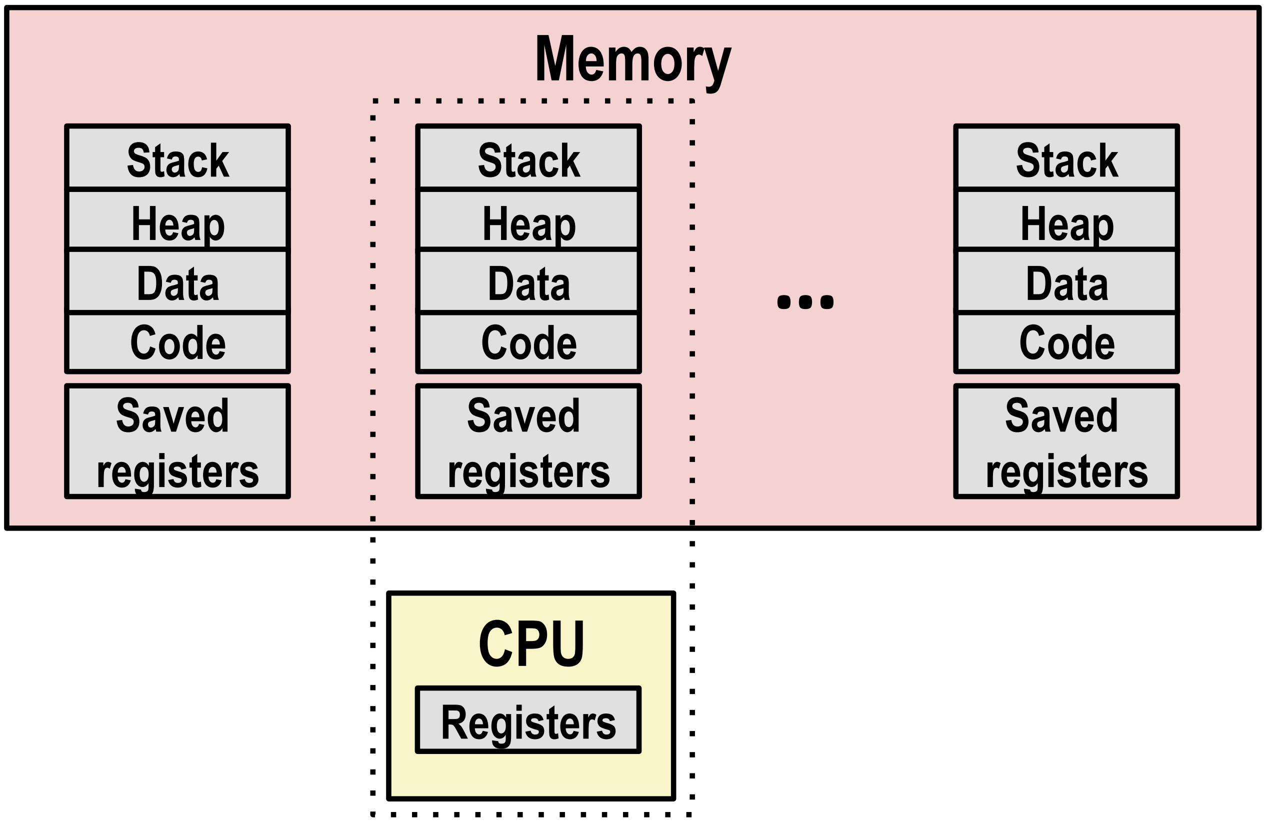
- **Single processor executes multiple processes concurrently**
  - Process executions interleaved (multitasking)
  - Address spaces managed by virtual memory system
  - Register values for nonexecuting processes saved in memory

Multiprocessing: The (Traditional) Reality



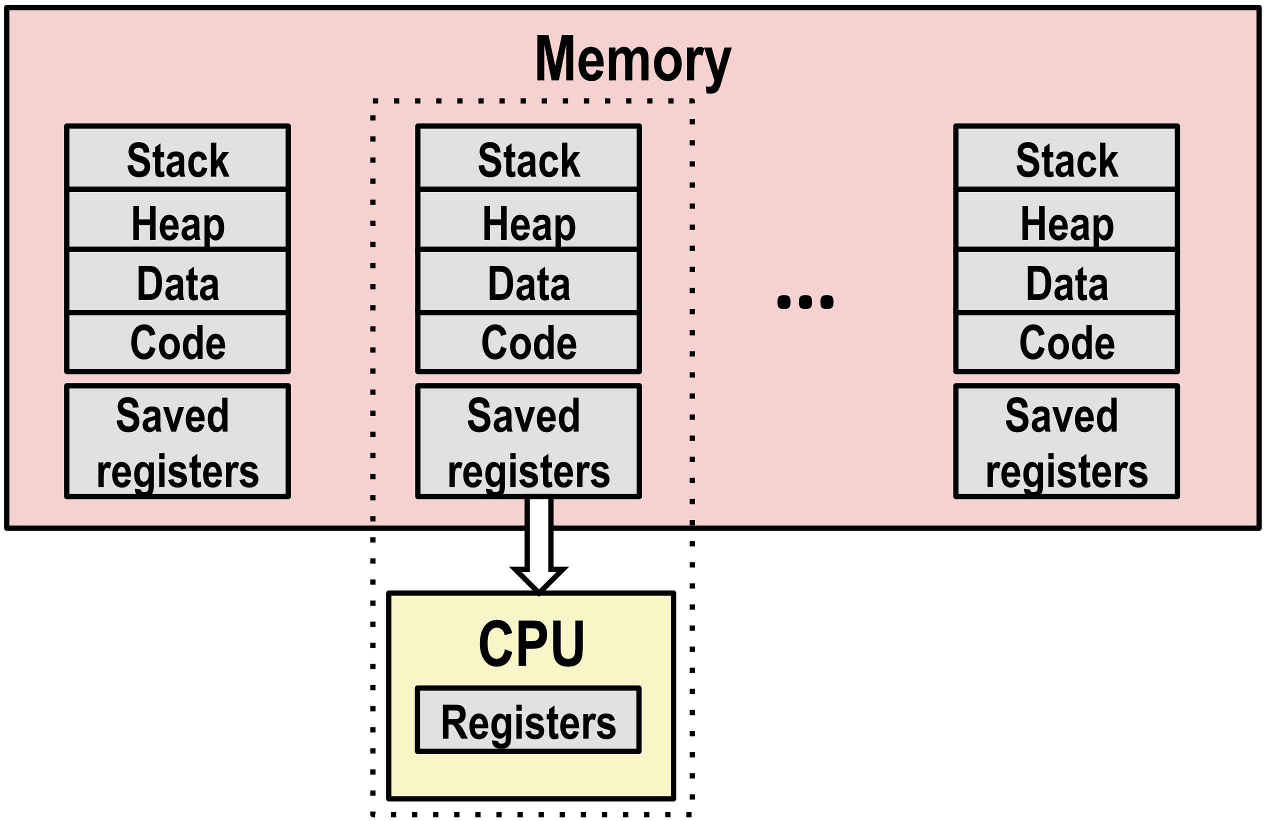
- Save current registers in memory

Multiprocessing: The (Traditional) Reality



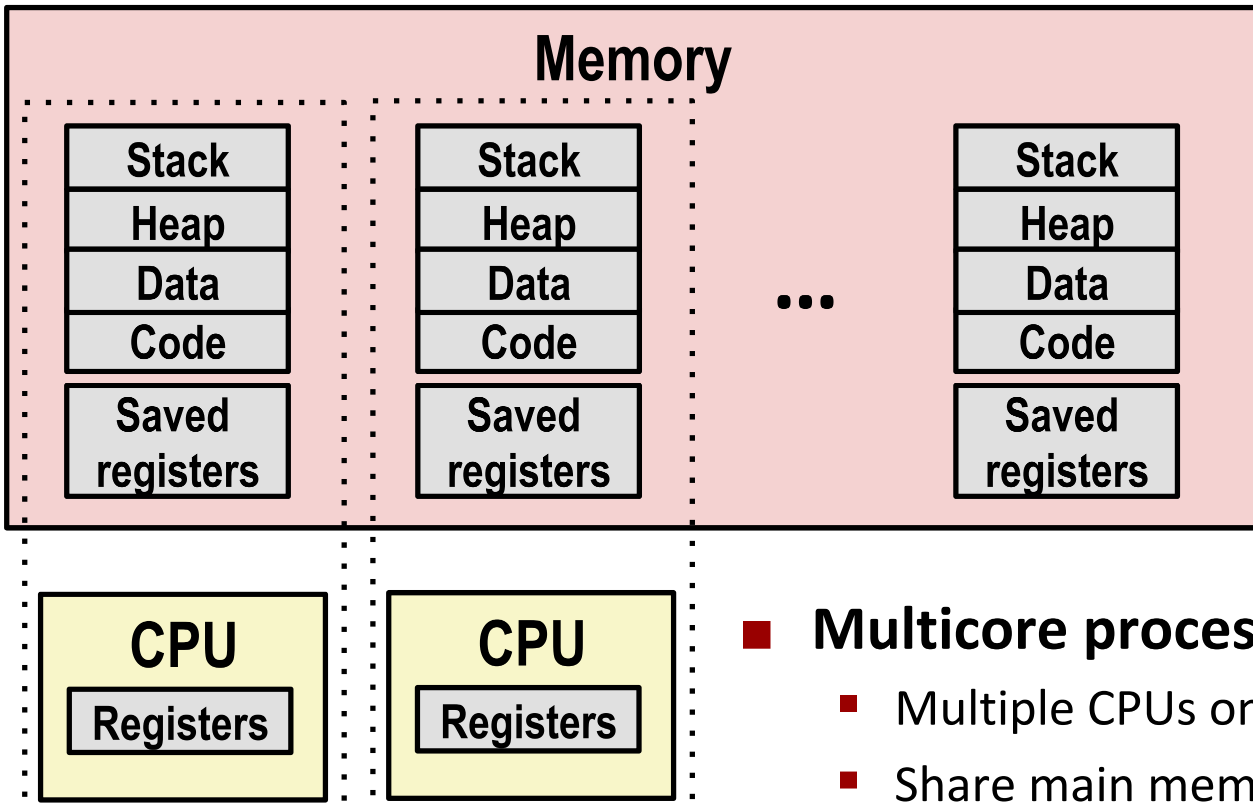
- Schedule next process for execution

Multiprocessing: The (Traditional) Reality



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

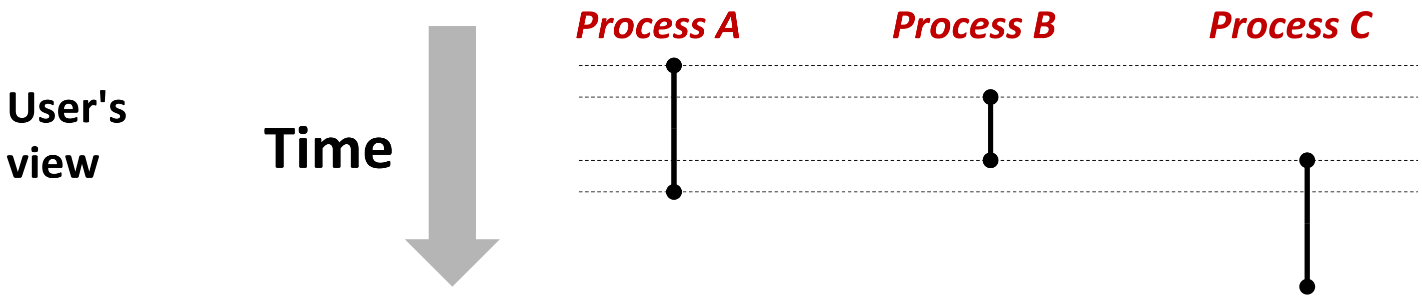
Multiprocessing: The (Modern) Reality



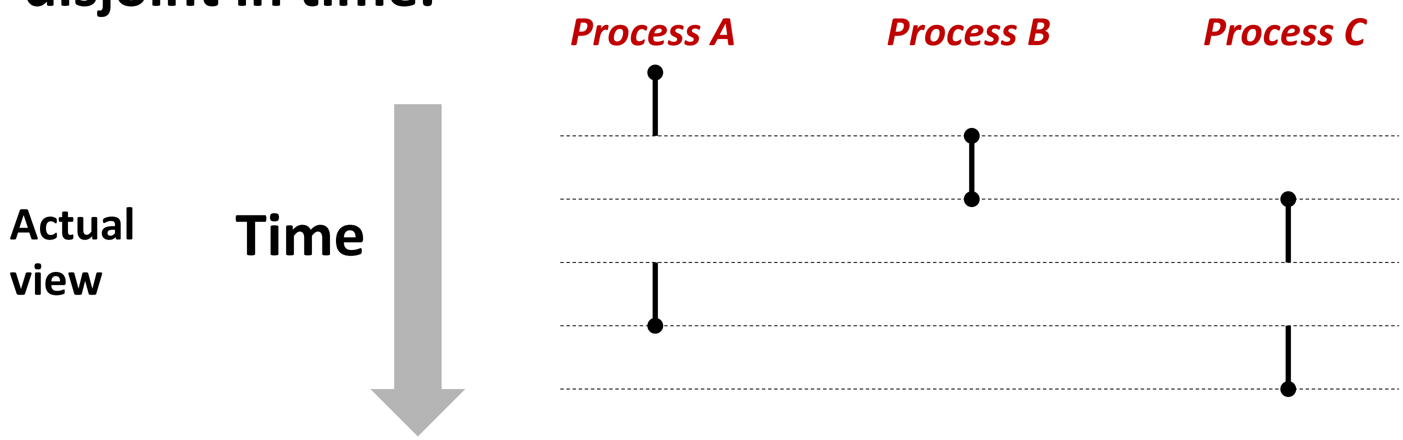
- **Multicore processors**
  - Multiple CPUs on single chip
  - Share main memory (and some of the caches)
  - Each can execute a separate process
    - Scheduling of processors onto cores done by kernel

User View of Concurrent Processes

- We can think of concurrent processes as running simultaneously with each other

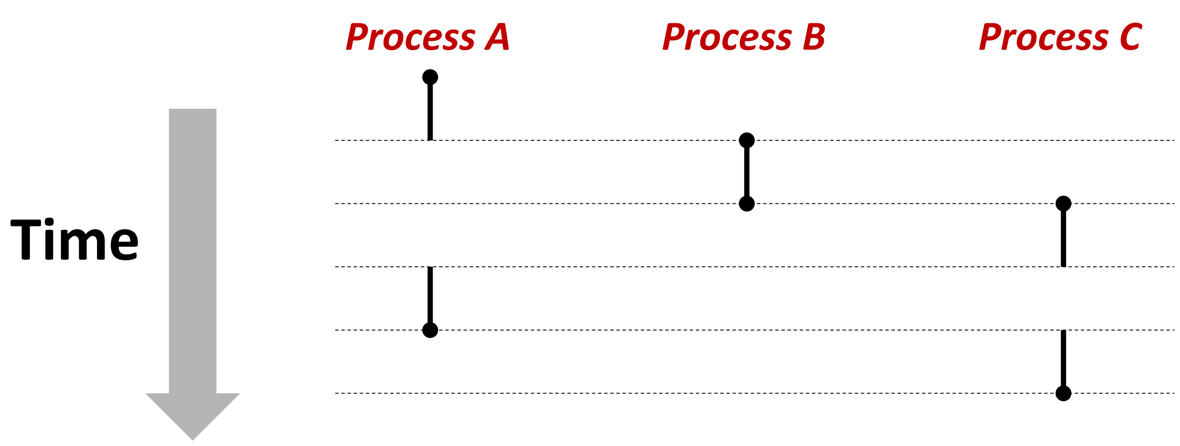


- Control flows for concurrent processes can be physically disjoint in time.



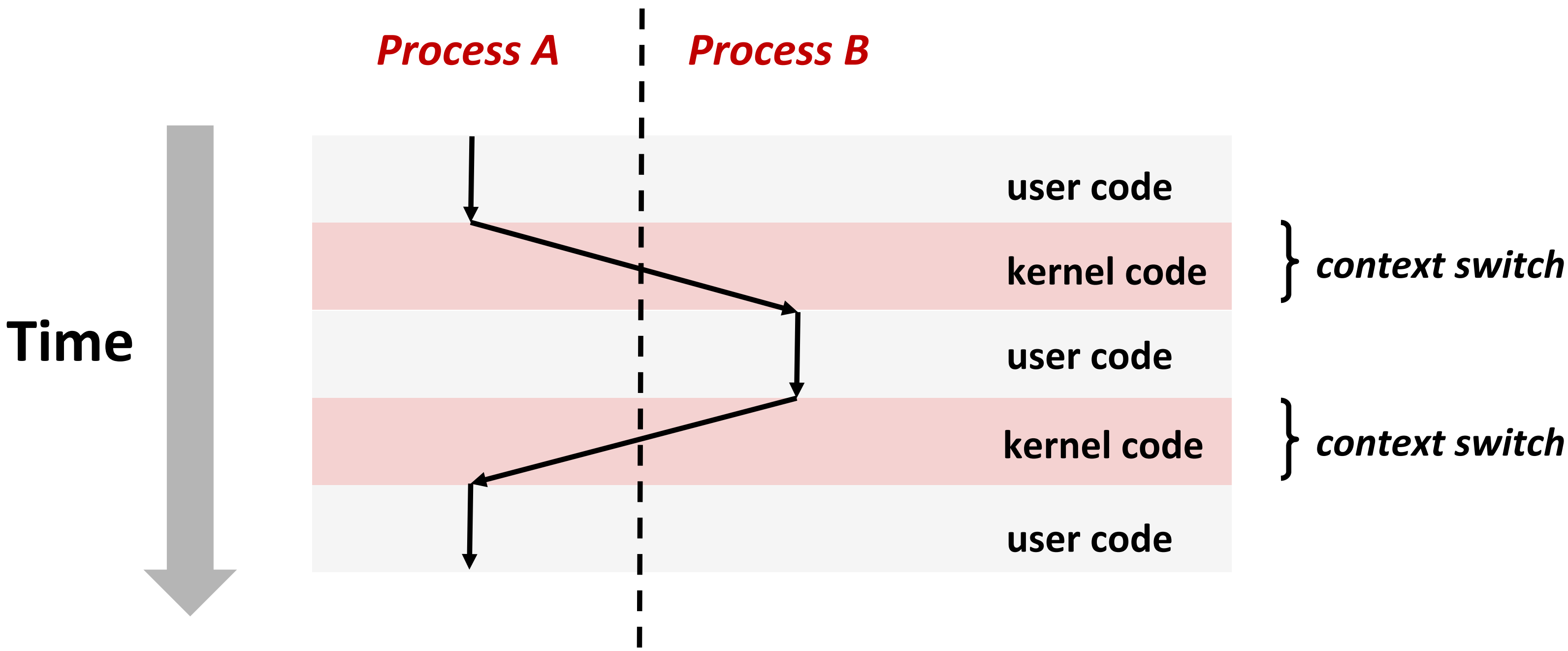
Concurrent Processes

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



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



Today

- Processes
- **Exceptions and Exceptional Control Flow**
- Process Control

Control Flow

- **Processors do only one thing:**
  - From startup to shutdown, a CPU simply reads and executes (interpreted) 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>

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***

- **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: may need to context switch

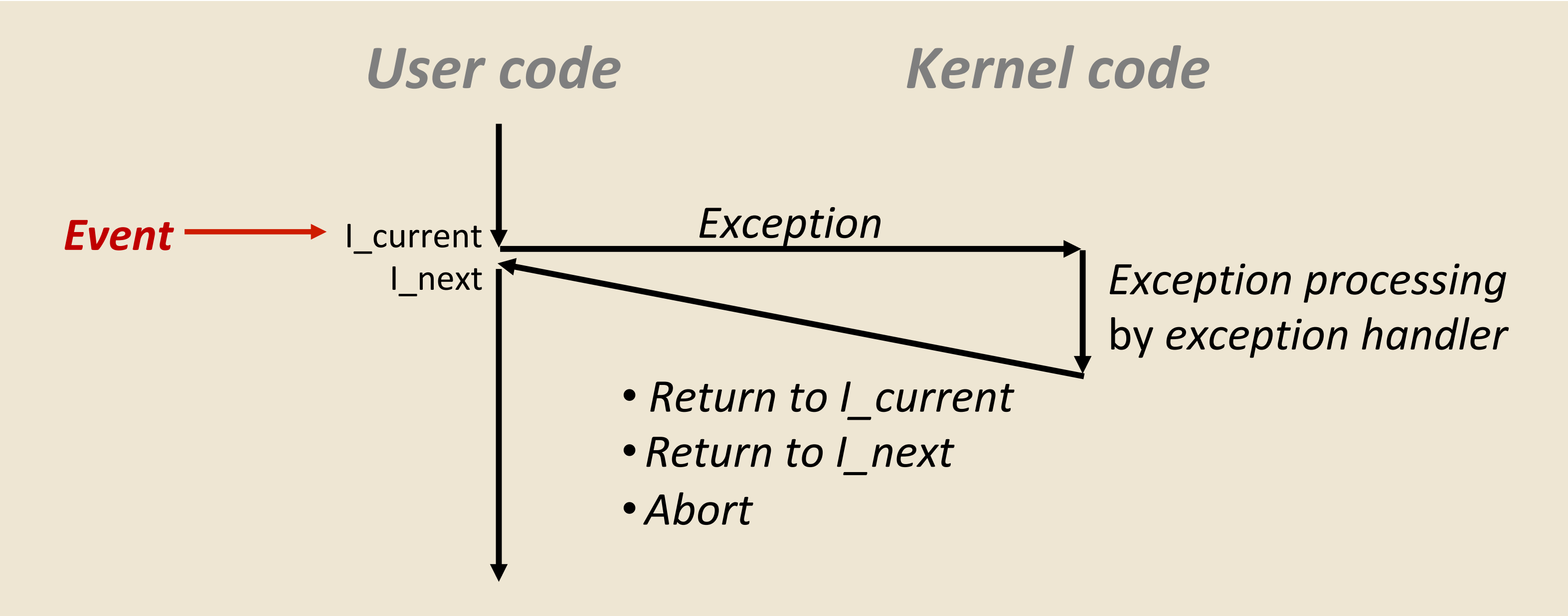
- **Exceptional Control Flow:**
  - System reacts to these significant changes in system state by making abrupt changes in the control flow

Exceptional Control Flow

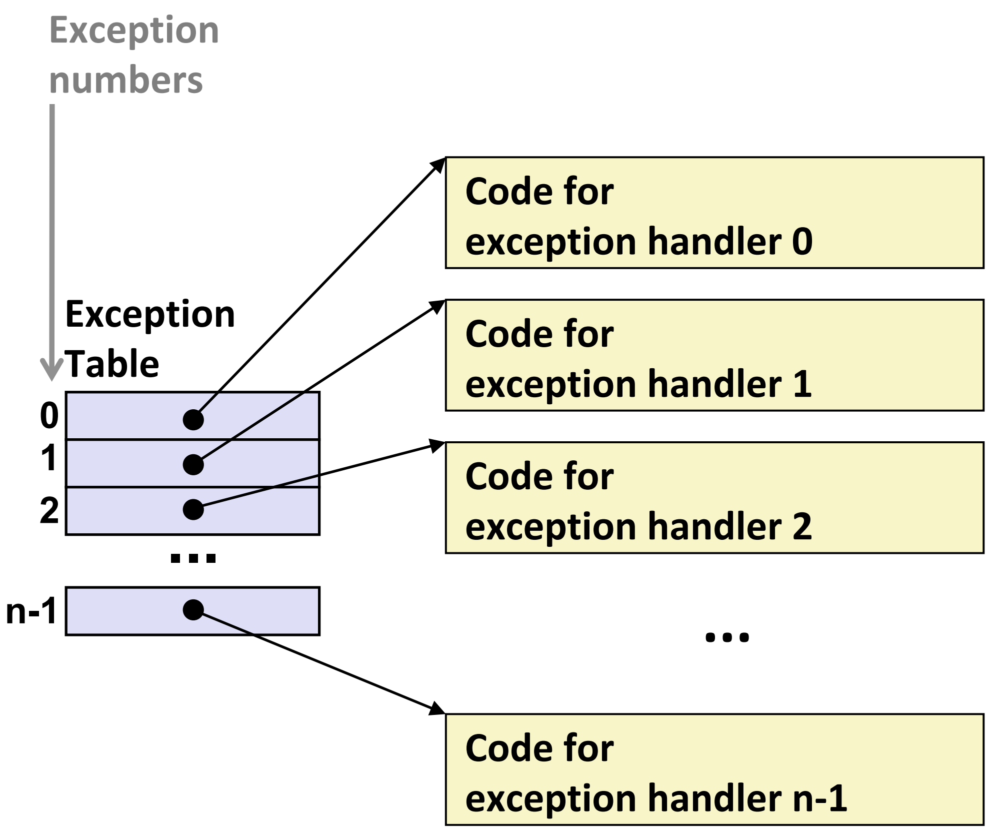
- **Exists at all levels of a computer system**
- **Low level mechanisms**
  - 1. **Exceptions**
    - Change in control flow in response to a system event (i.e., change in system state)
    - Implemented using combination of hardware and OS software
- **Higher level mechanisms**
  - 2. **Process context switch**
    - Implemented by OS software and hardware timer
  - 3. **Signals**
    - Implemented by OS software
  - 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, arithmetic overflow, page fault, I/O request completes, typing Ctrl-C

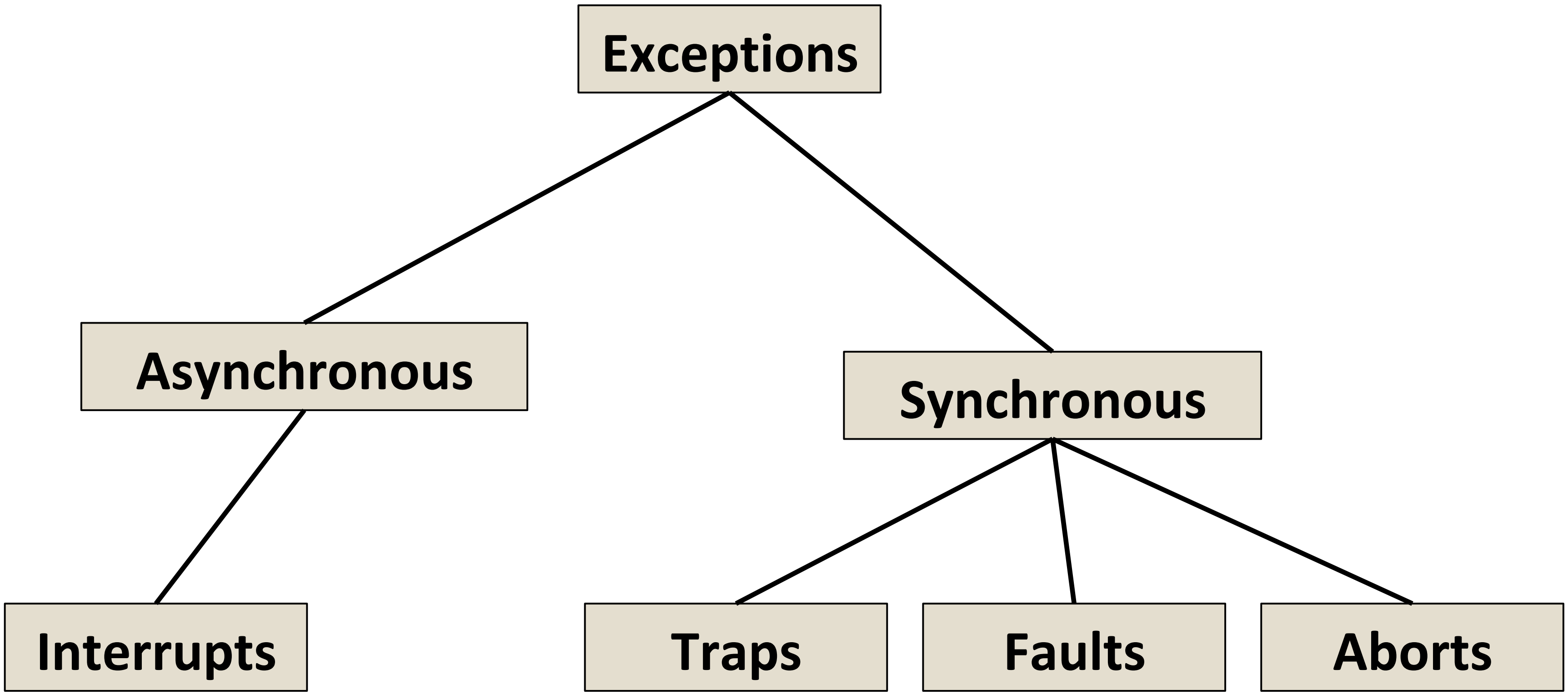


Exception Tables



- Each type of event has a unique exception number k
- k = index into exception table (a.k.a. interrupt vector)
- Handler k is called each time exception k occurs
- Handler is run in kernel mode, so it has access to all kernel resources.

Taxonomy


Exceptions
Asynchronous
Synchronous
Interrupts
Traps
Faults
Aborts

Asynchronous Exceptions (Interrupts)

- **Caused by events external to the processor**
  - Indicated by setting the processor's *interrupt pin*
  - 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 (e.g. DMA)

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), floating point exceptions
    - Either re-executes faulting (“current”) instruction or aborts
  - ***Aborts***
    - Unintentional and unrecoverable
    - Examples: illegal instruction, hardware errors (e.g. parity error)
    - Aborts current program

System Calls

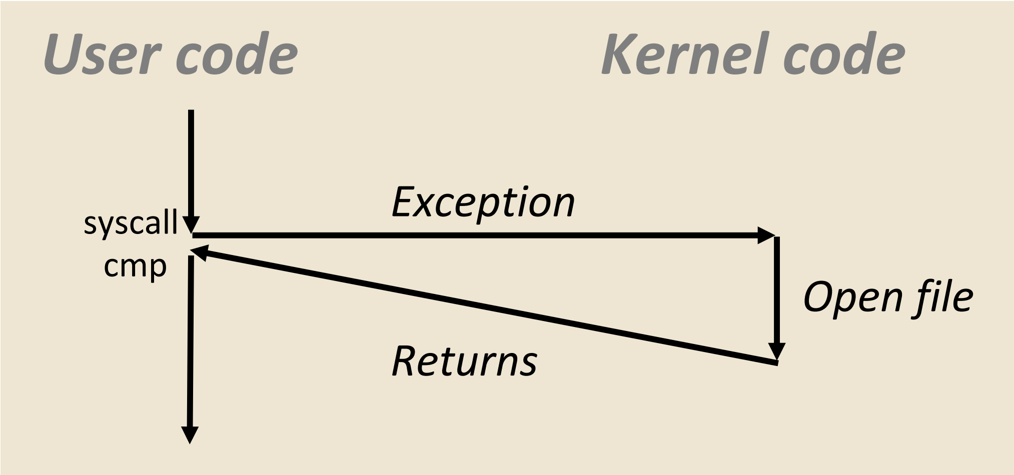
- **Procedure-like interface to request services from the kernel**
- **Each x86-64 system call has a unique ID number**
- **Examples:**

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,%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`**

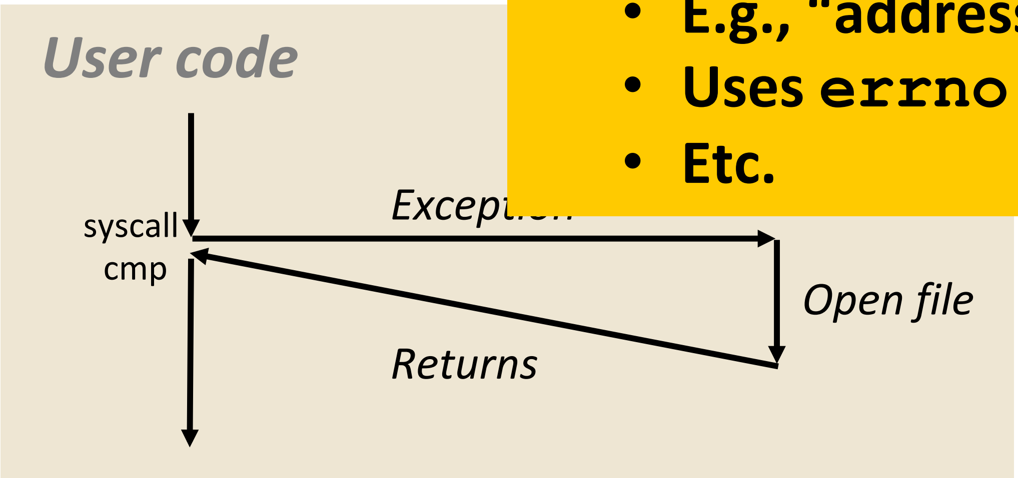
System Call I

- User calls: **open(f**
- Calls **__open** functi

```
00000000000e5d70 <
...
e5d79:   b8 02 00
e5d7e:   0f 05
e5d80:   48 3d 01
...
e5dfa:   c3
```

Almost like a function call
- **Transfer of control**
- **On return, executes next instruction**
- **Passes arguments using calling convention**
- **Gets result in `%rax`**

One Important exception!
- **Executed by Kernel**
- **Different set of privileges**
- **And other differences:**
  - **E.g., "address" of "function" is in `%rax`**
  - **Uses `errno`**
  - **Etc.**



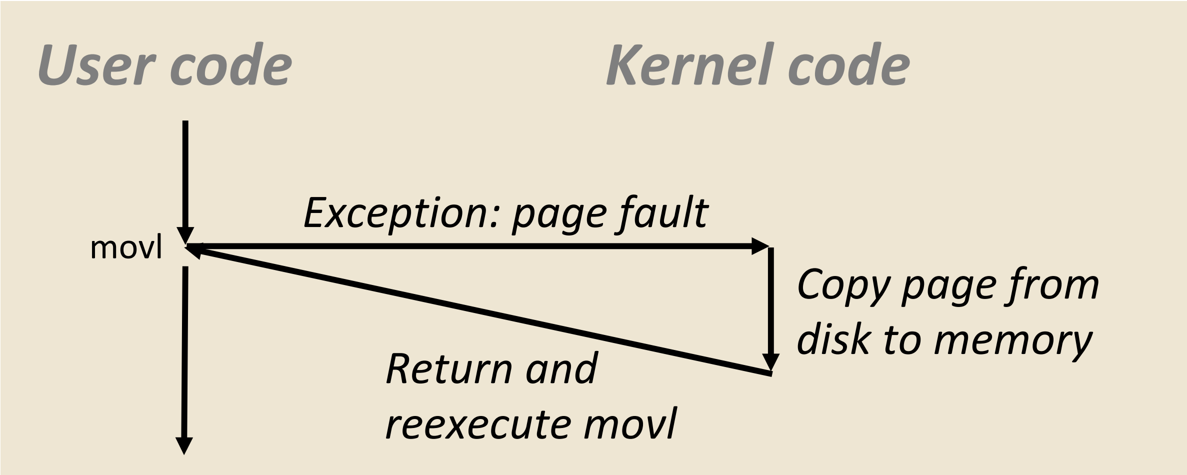
- Return value in `%rax`
- Negative value is an error corresponding to negative `errno`

Fault Example: Page Fault

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

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

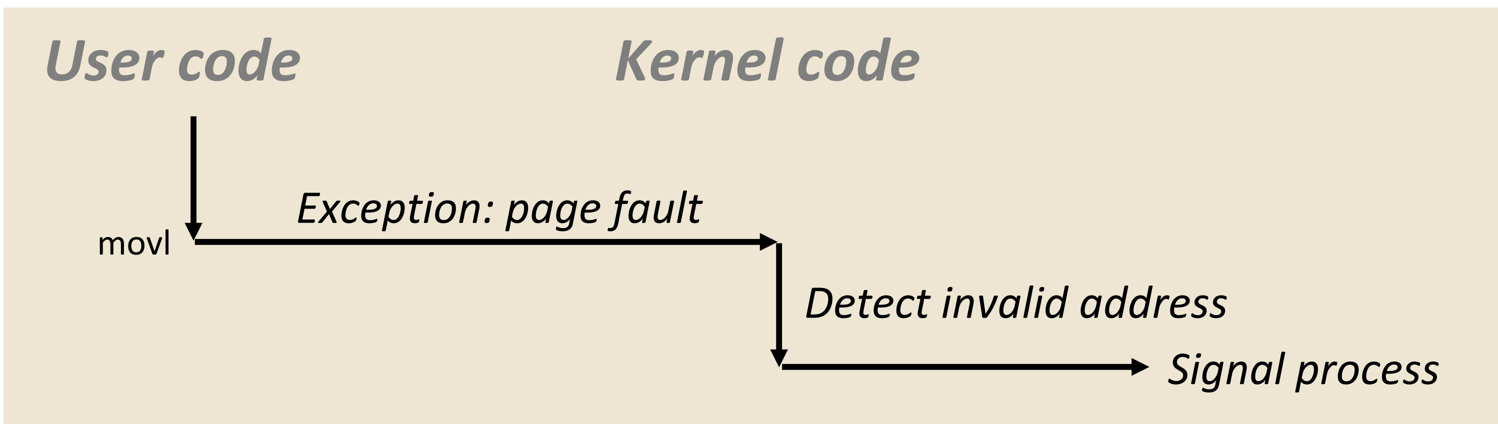
```
80483b7:       c7 05 10 9d 04 08 0d    movl    $0xd,0x8049d10
```



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
- User process exits with “segmentation fault”

Today

- Processes
- Exceptions and Exceptional Control Flow
- **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 (next lecture when we study signals)
- **Terminated**
  - Process is stopped permanently

Terminating Processes

- **Process becomes terminated for one of three reasons:**
  - Returning from the `main` routine
  - Calling the `exit` function
  - Receiving a signal whose default action is to terminate (next lecture)

- **`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

- **`exit` 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 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

```c
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);
}
```

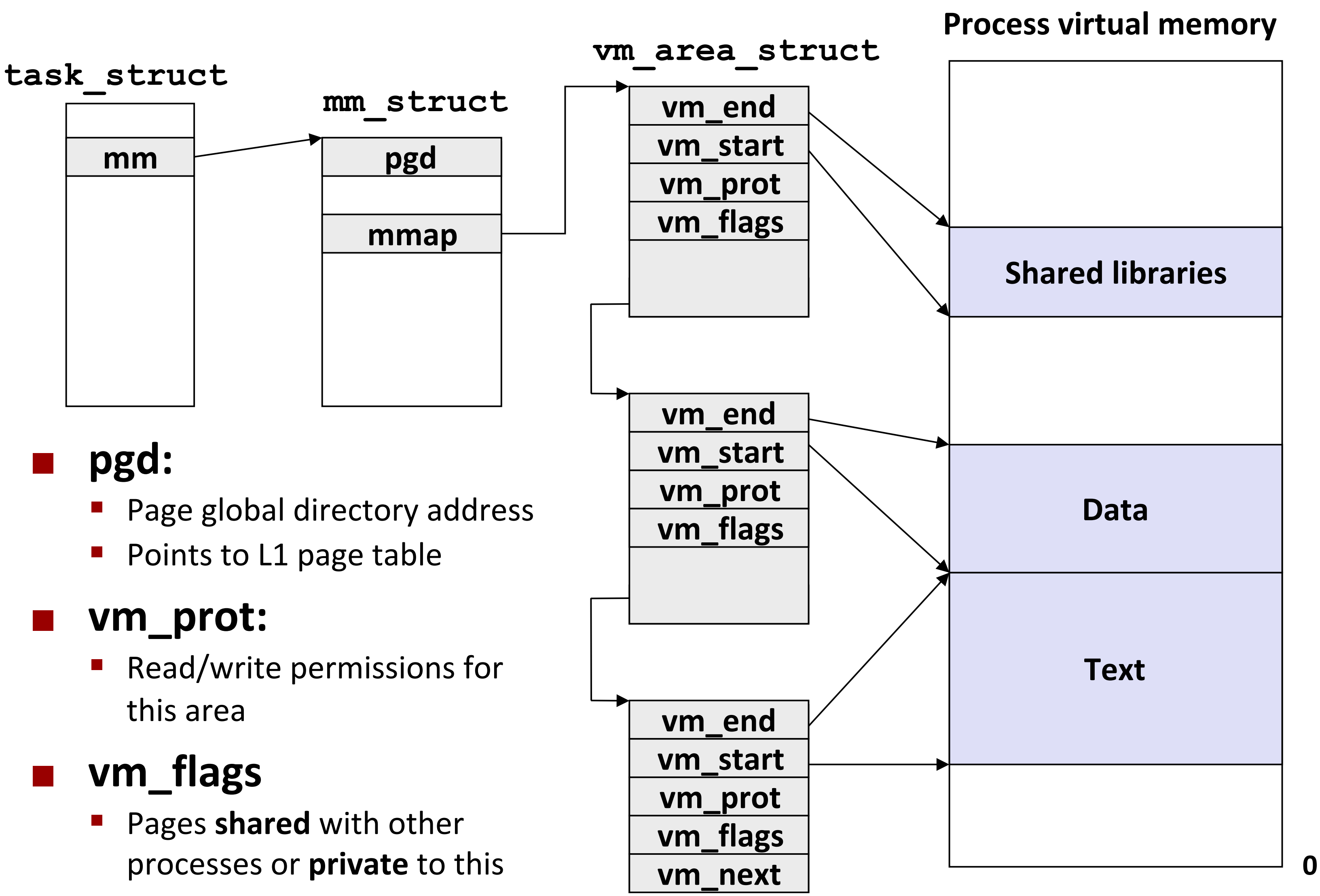
fork.c

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

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

- 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 (copy on write!)
- Shared open files
  - `stdout` is the same in both parent and child

Linux Organizes VM as Collection of "Areas"



- **pgd:**
  - Page global directory address
  - Points to L1 page table
- **vm_prot:**
  - Read/write permissions for this area
- **vm_flags**
  - Pages **shared** with other processes or **private** to this process

The `fork` Function

- **VM and memory mapping explain how `fork` provides private address space for each process.**
- **To create virtual address for new new process**
  - Create exact copies of current `mm_struct`, `vm_area_struct`, and page tables.
  - Flag each page in both processes as read-only
  - Flag each `vm_area_struct` in both processes as private COW
- **On return, each process has exact copy of virtual memory**
- **Subsequent writes create new pages using COW mechanism.**

System Call Error Handling

- **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:**

```
if ((pid = fork()) < 0) {
    fprintf(stderr, "fork error: %s\n", strerror(errno));
    exit(1);
}
```

Error-reporting functions

- 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(1);
}
```

```
 if ((pid = fork()) < 0)
    unix_error("fork error");
```

Error-handling Wrappers

- We simplify the code we present 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;
}
```

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

`fork` Example Fixed

```
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
```

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

- 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

Obtaining Process IDs

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

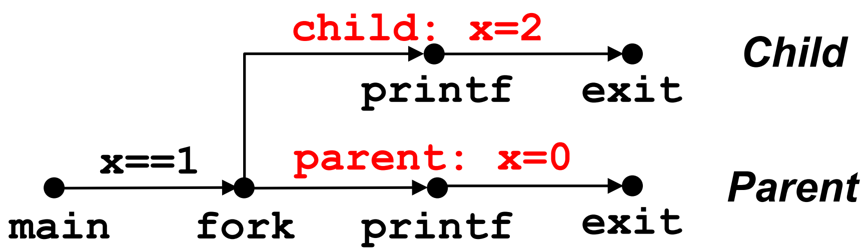
Modeling `fork` with Process Graphs

- **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 inedges
- **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

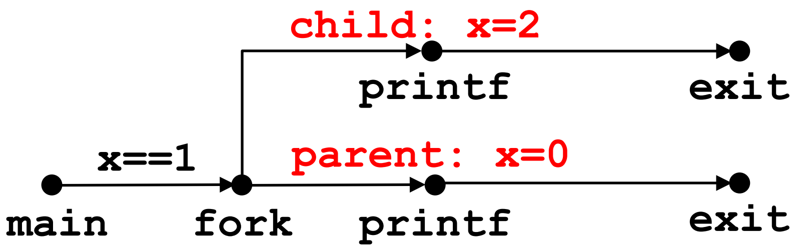
```c
int main()
{
    pid_t pid;
    int x = 1;

    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

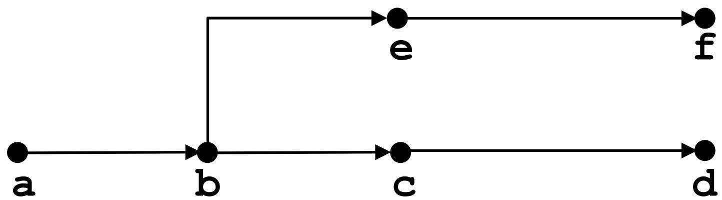


Interpreting Process Graphs

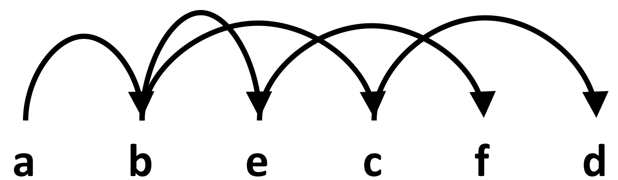
- **Original graph:**



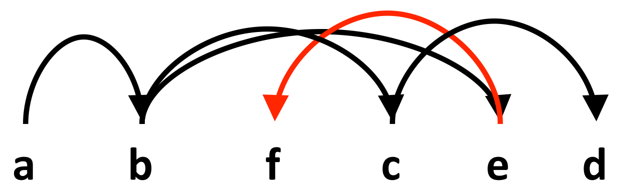
- **Relabled graph:**



Feasible total ordering:



Infeasible total ordering:



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

```c
void fork2()
{
    printf("L0\n");
    if (Fork() != 0) {
        printf("L1\n");
        if (Fork() != 0) {
            printf("L2\n");
        }
    }
    printf("Bye\n");
}
```

forks.c

Feasible output?

L0
L1
Bye
Bye
L2
Bye

Feasible output?

L0
Bye
L1
Bye
Bye
L2