Roadmap

C:

```
car *c = malloc(sizeof(car));
c->miles = 100;
c->gals = 17;
float mpg = get_mpg(c);
free(c);
```

Java:

```
Car c = new Car();
c.setMiles(100);
c.setGals(17);
float mpg =
    c.getMPG();
```

Assembly language:

```
get_mpg:
    pushq   %rbp
    movq    %rsp, %rbp
    ...
    popq    %rbp
    ret
```

Machine code:

```
0111010000011000
100011010000010000000010
1000100111000010
11000001111101000011111
```

OS:

Computer system:

Data & addressing
Integers & floats
Machine code & C
x86 assembly
Procedures & stacks
Arrays & structs
Memory & caches
Processes
Virtual memory
Memory allocation
Java vs. C

Processes – another important abstraction

- **First some preliminaries**
  - Control flow
  - Exceptional control flow
  - Asynchronous exceptions (interrupts)
  - Synchronous exceptions (traps & faults)
- **Processes**
  - Creating new processes
  - Fork and wait
  - Zombies

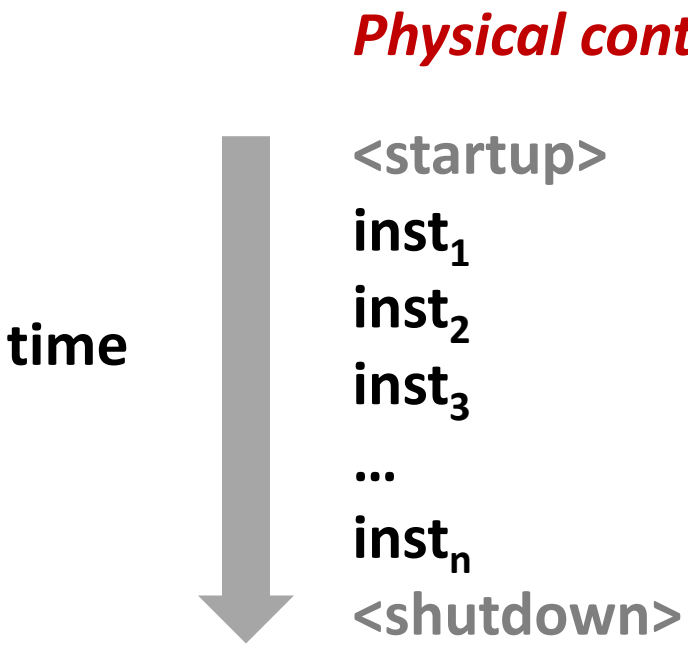
Control Flow

- So far, we've seen how the flow of control changes as a **single program** executes
- But a single CPU executes more than one program at a time – we also need to understand how control flows across the many components of the system
- For now we will assume there is only ONE CPU

- ***Exceptional* control flow is the basic mechanism used for:**
  - Transferring control between processes and OS
  - Handling I/O and virtual memory within the OS
  - Implementing multi-process applications like shells and web servers
  - Implementing concurrency

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*)



Altering the Control Flow

- **Up to now: two ways to change control flow:**
  - Jumps (conditional and unconditional)
  - Call and return

  Both react to changes in ***program state***

- **Processor also needs to react to changes in *system state***
  - user hits "Ctrl-C" at the keyboard
  - user clicks on a different application's window on the screen
  - data arrives from a disk or a network adapter
  - instruction divides by zero
  - system timer expires

Can jumps and procedure calls achieve this?

- Jumps and calls are not sufficient – the system needs mechanisms for *"exceptional"* control flow!

Exceptional Control Flow

- **Exists at all levels of a computer system**
- **Low level mechanisms**
  - Exceptions
    - change in processor's control flow in response to a system event (i.e., change in system state, user-generated interrupt)
    - Implemented using a combination of hardware and OS software
- **Higher level mechanisms**
  - Process context switch
    - Implemented by OS software and hardware timer
  - Signals – you'll hear about these in CSE451 and CSE/EE ~~466~~ 474
    - Implemented by OS software

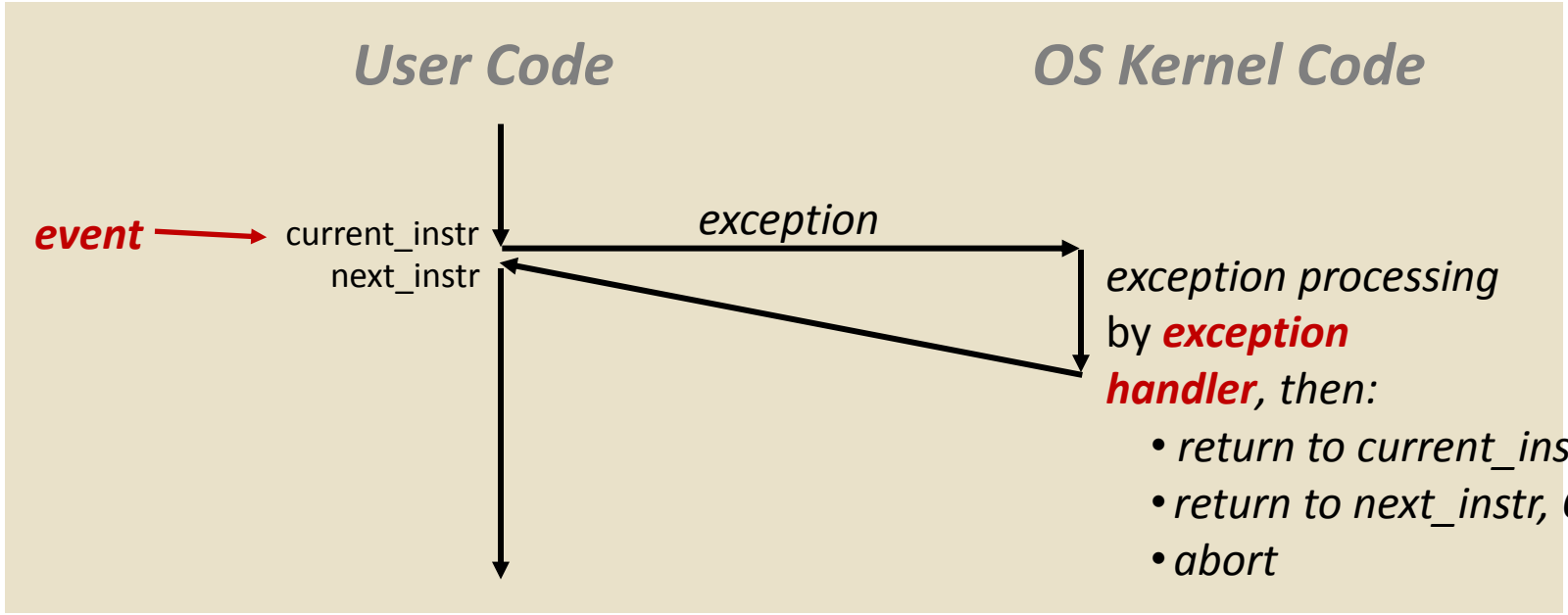
We'll talk about exceptions and process context switch

Processes

- **First some preliminaries**
  - ~~Control flow~~
  - Exceptional control flow
  - Asynchronous exceptions (interrupts)
  - Synchronous exceptions (traps & faults)
- **Processes**
  - Creating new processes
  - Fork and wait
  - Zombies

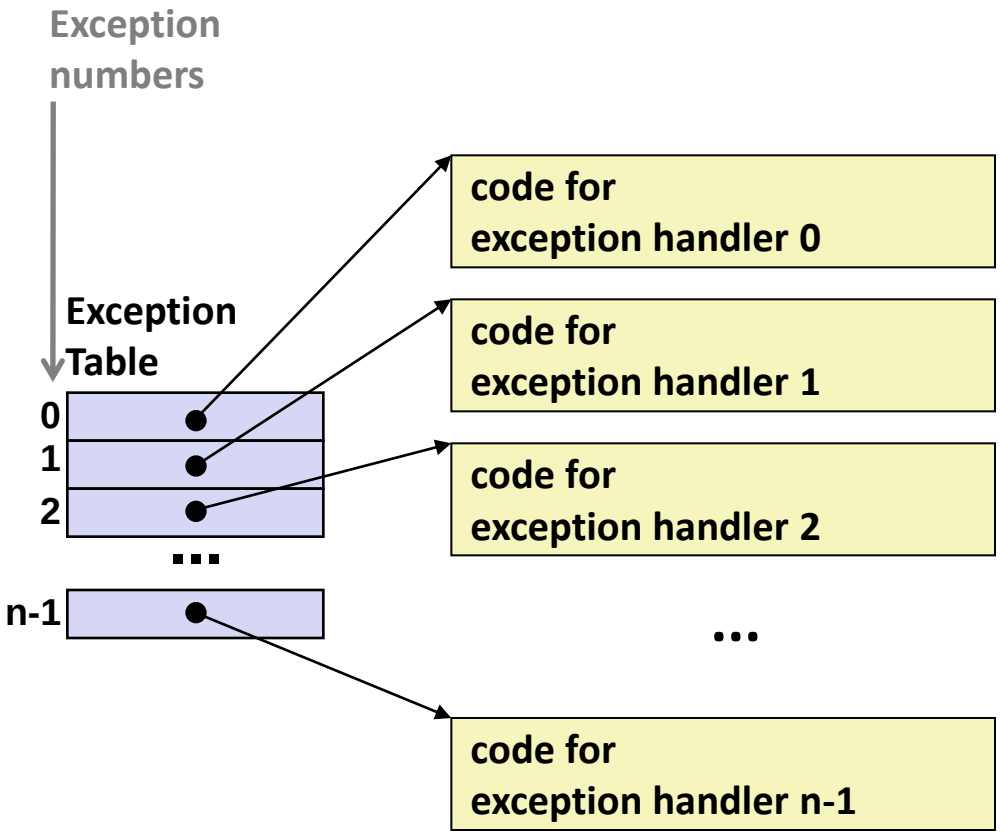
Exceptions

- **An *exception* is transfer of control to the operating system (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:
    div by 0, page fault, I/O request completes, Ctrl-C



- *How does the system know where to jump to in the OS?*

Exception Table: a jump table for exceptions



- 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

Also called: Interrupt Vector Table

Exception Table (Excerpt)

Exception Number	*Description*	*Exception Class*
0	Divide error	Fault
13	General protection fault	Fault
14	Page fault	Fault
18	Machine check	Abort
32-255	OS-defined	Interrupt or trap

Types of Exceptions

- **Asynchronous Exceptions (Interrupts**) - Caused by events external to the processor

- **Synchronous Exceptions** – Caused by events that occur as a result of executing an instruction
  - **Traps** - Intentional
  - **Faults** - Unintentional
  - **Aborts** - Unintentional

Processes

- **First some preliminaries**
  - ~~Control flow~~
  - ~~Exceptional control flow~~
  - Asynchronous exceptions (interrupts)
  - Synchronous exceptions (traps & faults)
- **Processes**
  - Creating new processes
  - Fork and wait
  - Zombies

Asynchronous Exceptions (Interrupts)

- **Caused by events external to the processor**
  - Indicated by setting the processor's interrupt pin(s) (wire into CPU)
  - After interrupt handler runs, the handler returns to "next" instruction
- **Examples:**
  - I/O interrupts
    - hitting Ctrl-C on the keyboard
    - clicking a mouse button or tapping a touchscreen
    - arrival of a packet from a network
    - arrival of data from a disk
  - Timer interrupt
    - Every few ms, an external timer chip triggers an interrupt
    - Used by the OS kernel to take back control from user programs

Synchronous Exceptions

- **Caused by events that occur as a result of executing an instruction:**
  - ***Traps***
    - **Intentional**: transfer control to OS to perform some function
    - Examples: ***system calls***, breakpoint traps, special instructions
    - Returns control to "next" instruction
  - ***Faults***
    - **Unintentional** but possibly recoverable
    - Examples: ***page faults*** (recoverable), segment protection faults (unrecoverable), integer divide-by-zero exceptions (unrecoverable)
    - Either re-executes faulting ("current") instruction or aborts
  - ***Aborts***
    - **Unintentional** and unrecoverable
    - Examples: parity error, machine check (hardware failure detected)
    - Aborts current program

System Calls

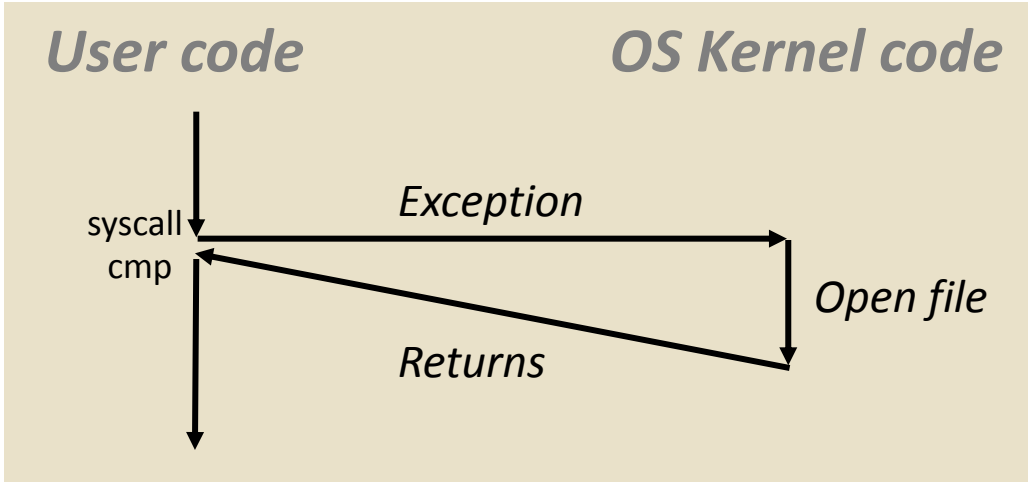
- Each x86-64 system call has a unique ID number
- Examples:

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

Traps: System Call Example: Opening File

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

```
0000000000e5d70 <__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
```



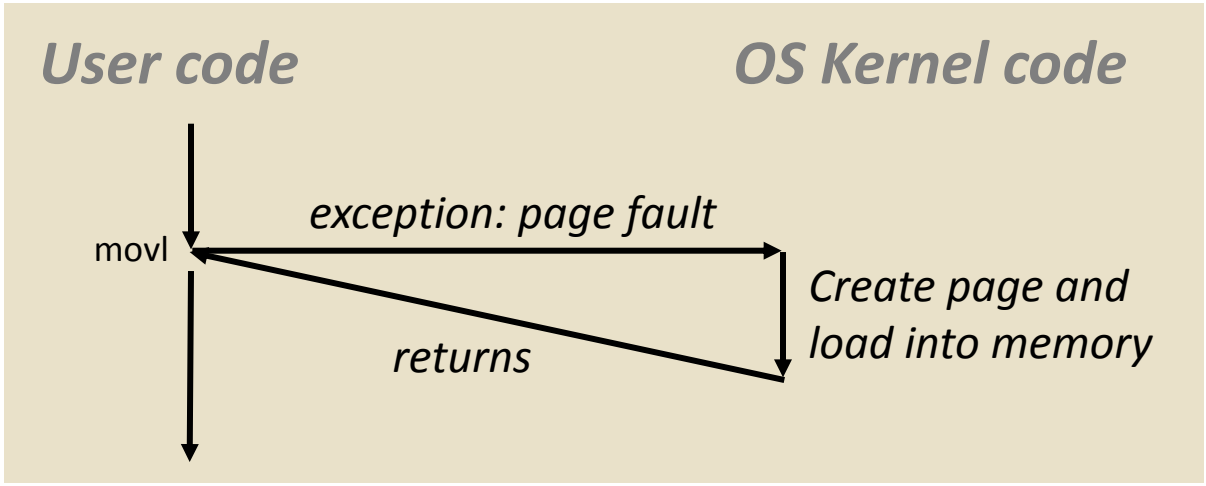
- `%rax` contains syscall number
- Other arguments in `%rdi`, `%rsi`, `%rdx`, `%r10`, `%r8`, `%r9`
- Return value in `%rax`
- Negative value is an error corresponding to negative `errno`

Fault Example: Page Fault

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

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

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

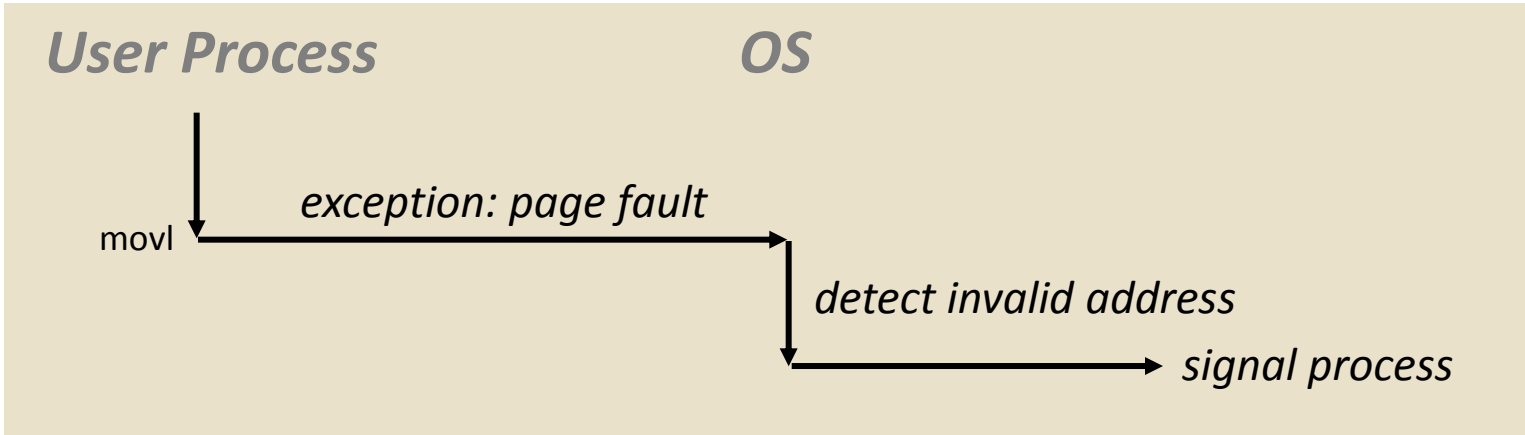


- Page fault handler must load page into physical memory
- Returns to faulting instruction: **`mov`** is executed again!
- Successful on second try

Fault Example: Invalid Memory Reference

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

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



- Page fault handler detects invalid address
- Sends **`SIGSEGV`** signal to user process
- User process exits with "segmentation fault"

Summary

- **Exceptions**
  - Events that require non-standard control flow
  - Generated externally (interrupts) or internally (traps and faults)
  - After an exception is handled, one of three things may happen:
    - Re-execute the current instruction
    - Resume execution with the next instruction
    - Abort the process that caused the exception

Processes

- **First some preliminaries**
  - ~~Control flow~~
  - ~~Exceptional control flow~~
  - ~~Asynchronous exceptions (interrupts)~~
  - ~~Synchronous exceptions (traps & faults)~~
- **Processes**
  - Creating new processes
  - Fork and wait
  - Zombies

What is a process?

- **Processes** are another *abstraction* in our computer system
  - provided by the OS
  - OS uses a data structure to represent each process
  - provides an *interface* between the program and the underlying hardware (CPU + memory)

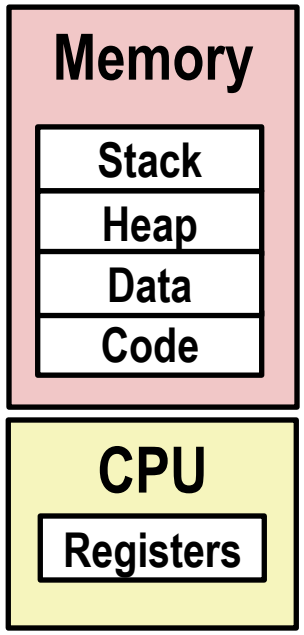
- What do processes have to do with ***exceptional control flow***?
  - Exceptional control flow is the mechanism that the OS uses to enable **multiple processes** to run on the same system.

- **What is the difference between:**
  - a processor? a program? a process?

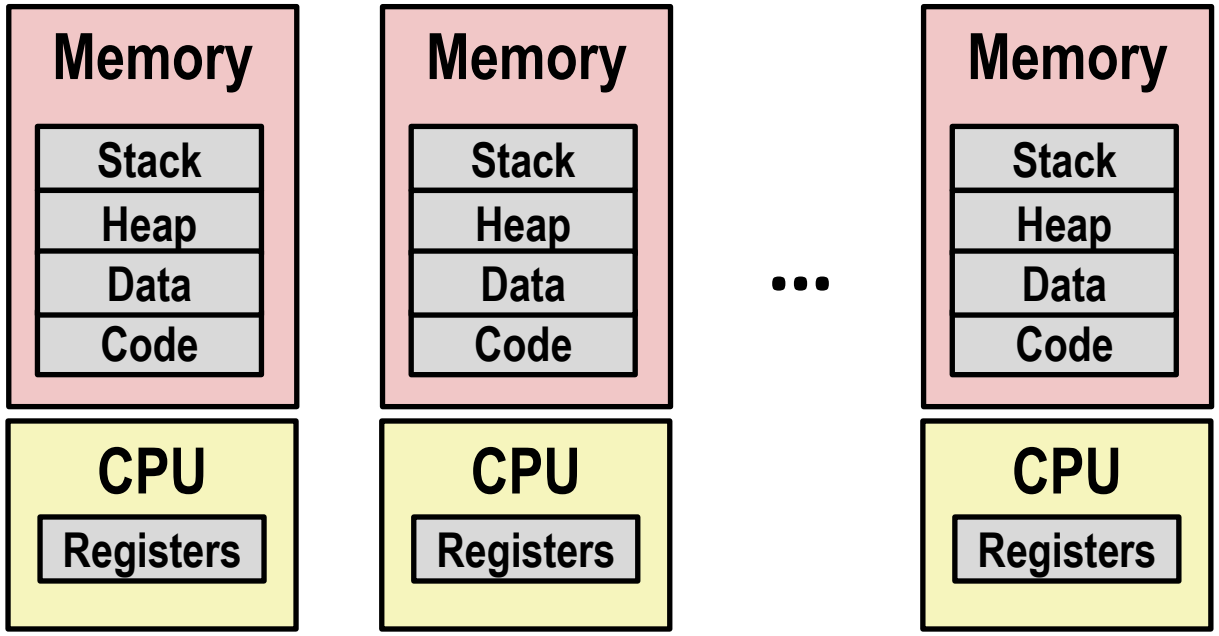
Processes

- **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 kernel mechanism called *virtual memory*

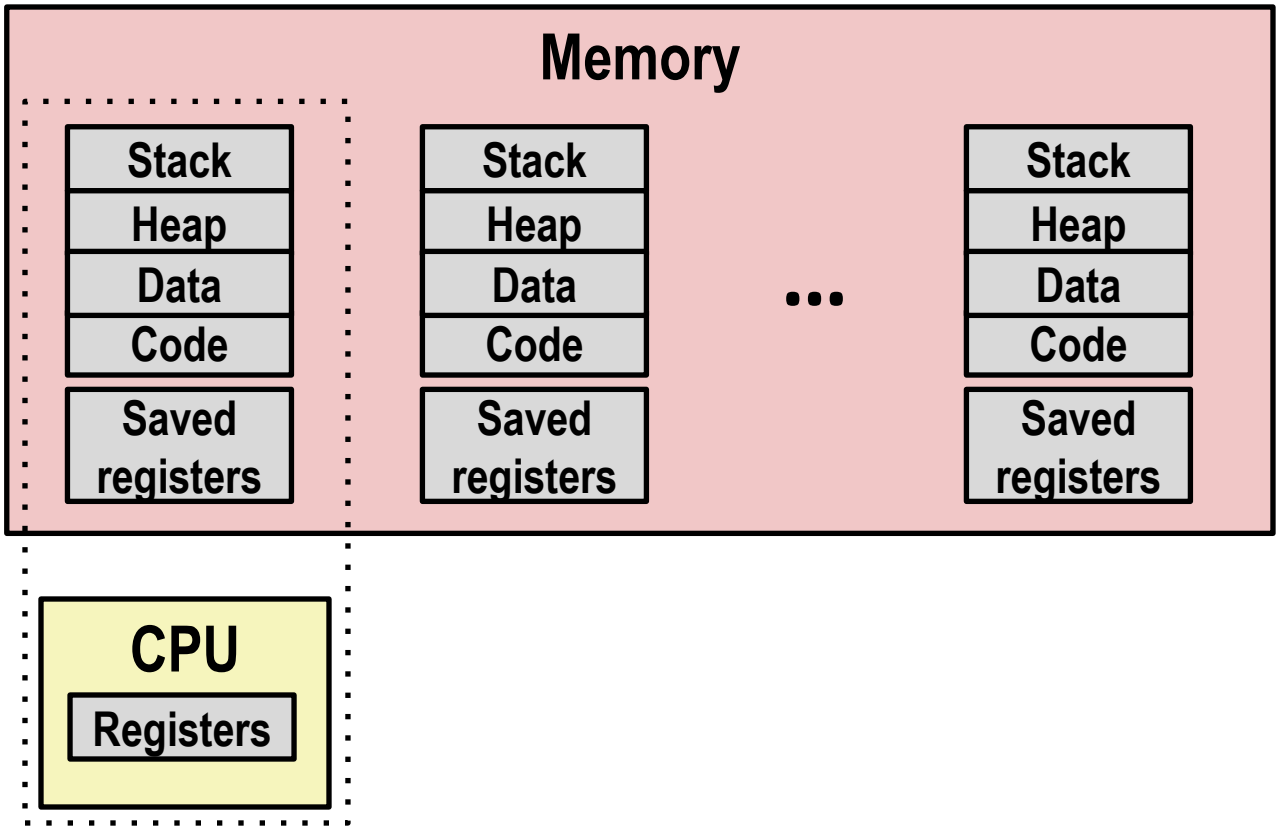


Multiprocessing: The Illusion



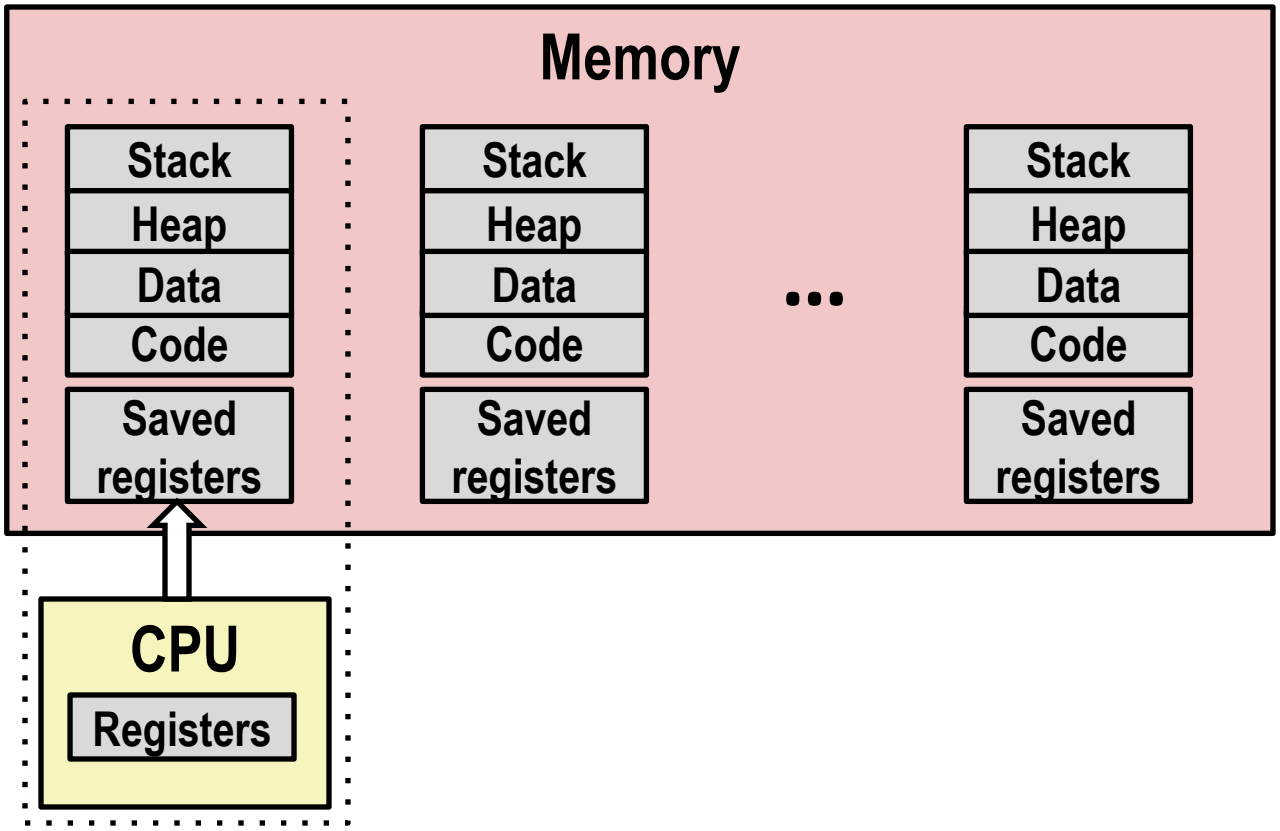
- **Computer runs many processes simultaneously**
  - Applications for one or more users
    - Web browsers, email clients, editors, …
  - Background tasks
    - Monitoring network & I/O devices

Multiprocessing: The (Traditional) Reality



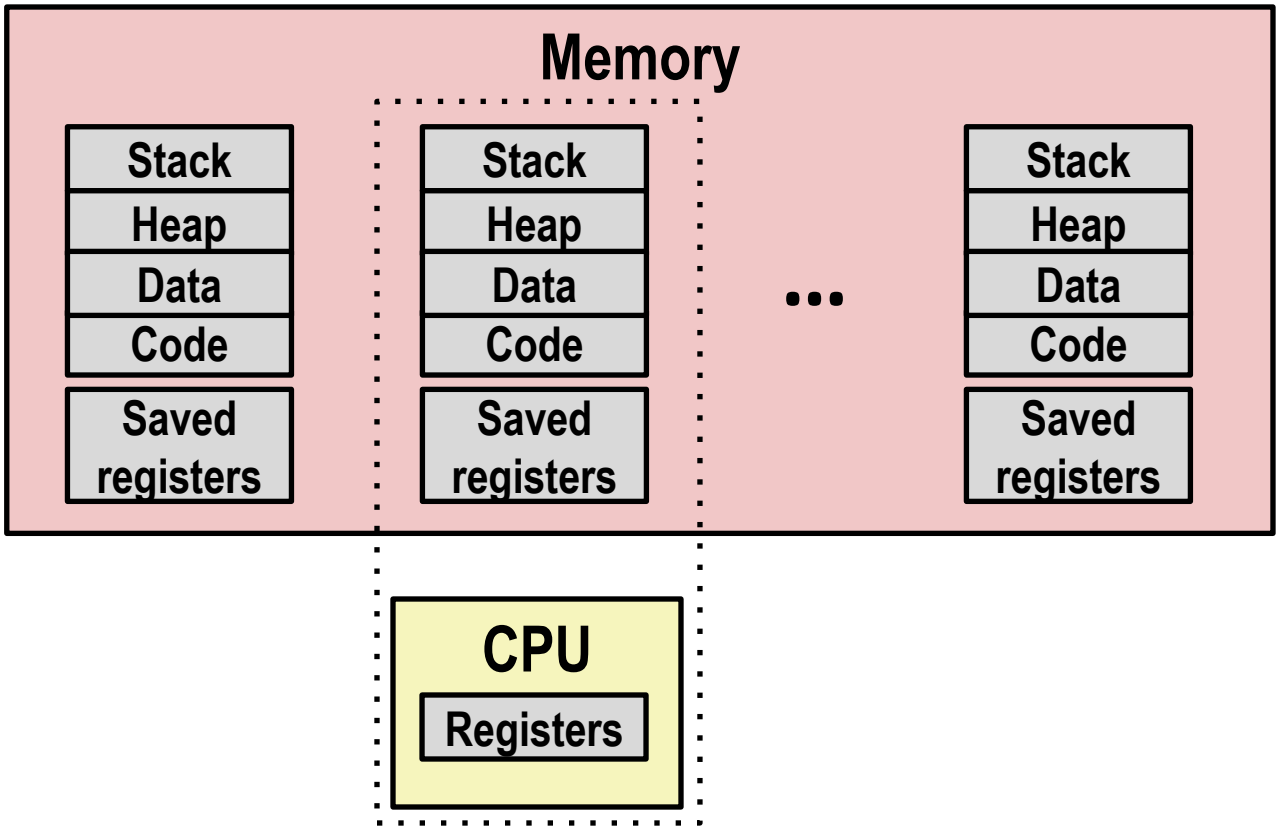
- **Single processor executes multiple processes concurrently**
  - Process executions interleaved (multitasking)
  - Address spaces managed by virtual memory system (later in course)
  - Register values for nonexecuting processes saved in memory

Multiprocessing: The (Traditional) Reality



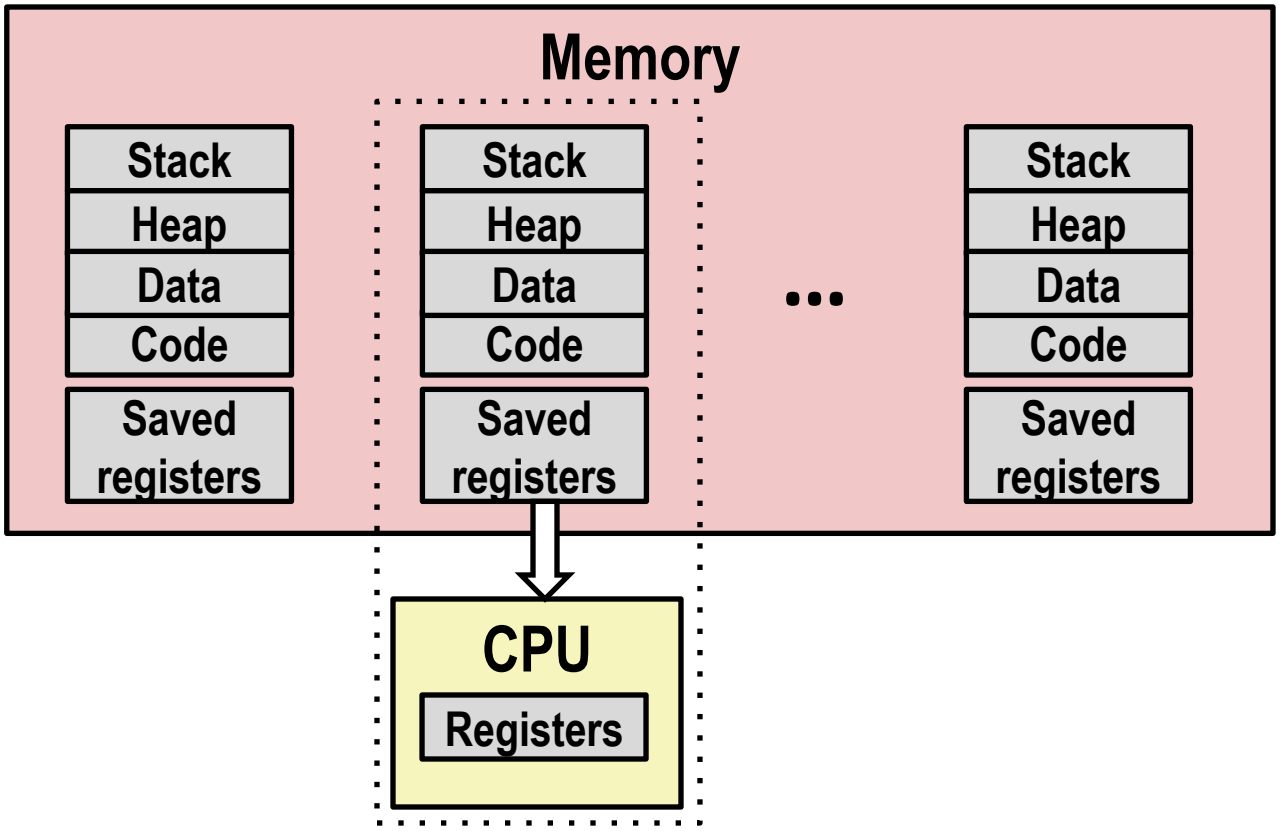
- Save current registers in memory

Multiprocessing: The (Traditional) Reality



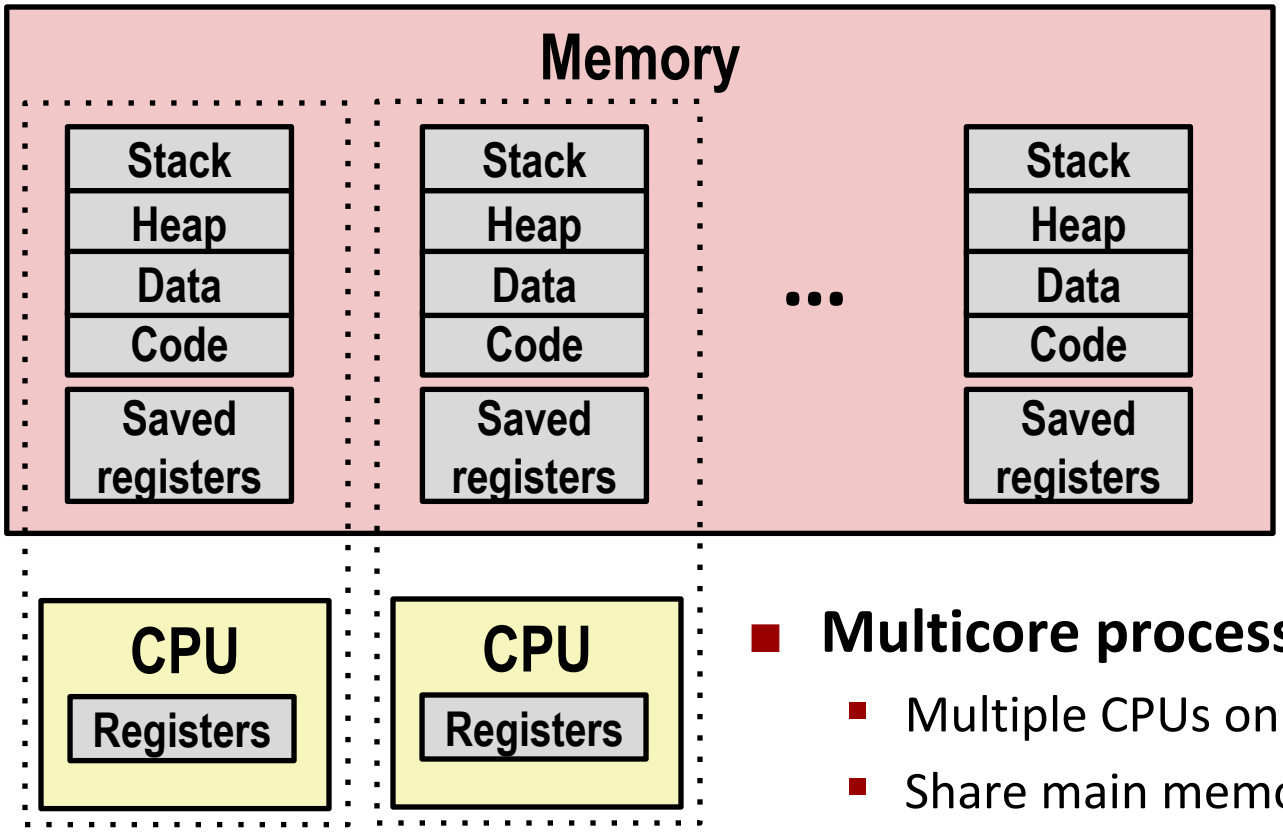
- Schedule next process for execution

Multiprocessing: The (Traditional) Reality



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

Multiprocessing: The (Modern) Reality

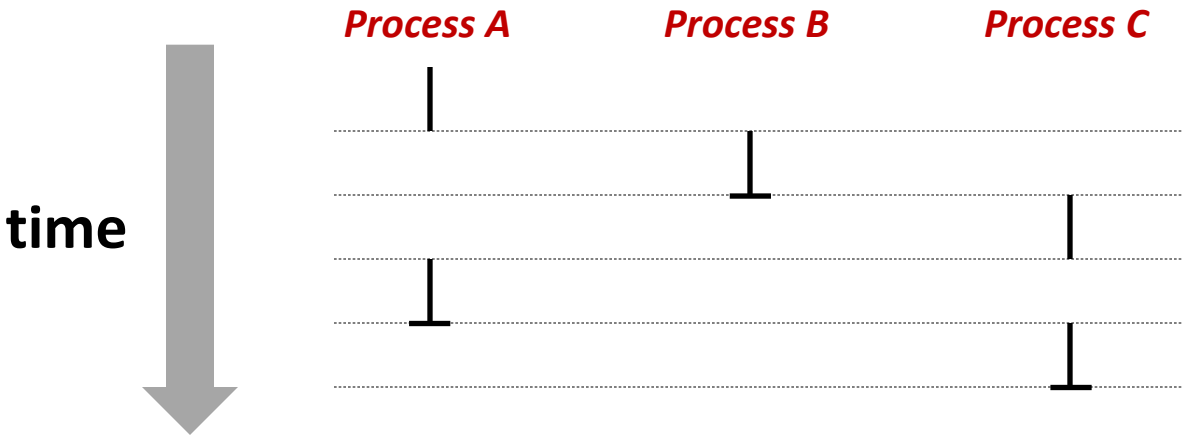


- **Multicore processors**
  - Multiple CPUs on single chip
  - Share main memory (and some of the caches)
  - Each can execute a separate process
    - Scheduling of processors onto cores done by kernel

Assume only **one** CPU

Concurrent Processes

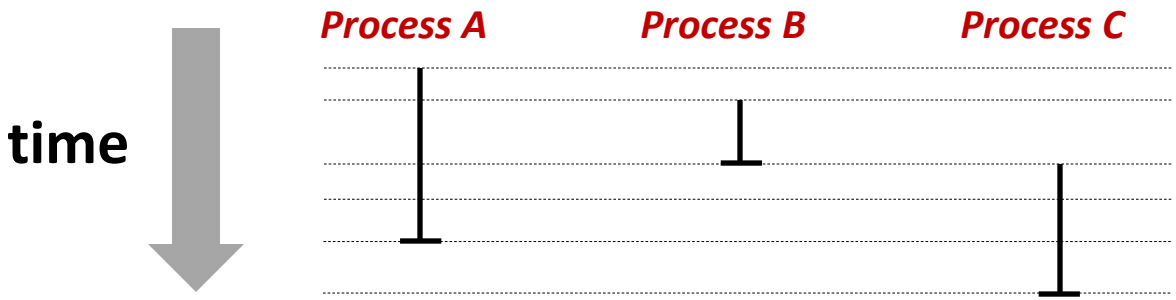
- **Each process is a logical control flow.**
- **Two processes *run concurrently* (are concurrent) if their instruction executions (flows) overlap in time**
- **Otherwise, they are *sequential***
- **Examples (running on single core):**
  - Concurrent: A & B, A & C
  - Sequential: B & C (B ends before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



Assume only **one** CPU

User's View of Concurrent Processes

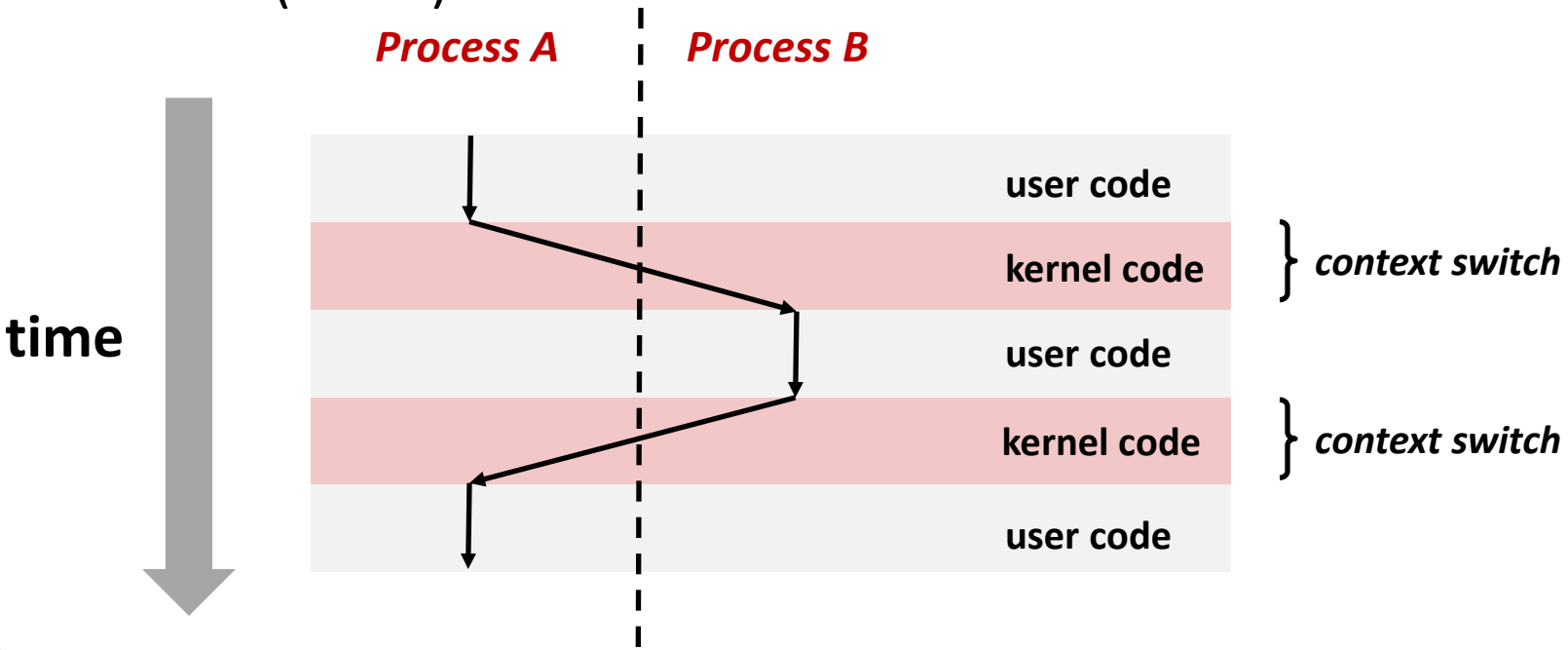
- **Control flows for concurrent processes are physically disjoint in time**
  - CPU only executes instructions for one process at a time
- **However, the user can *think of* concurrent processes as executing at the same time, in *parallel***



Assume only **one** CPU

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 a user process
- **Control flow passes from one process to another via a *context switch…*** (how?)



Processes

- **First some preliminaries**
  - ~~Control flow~~
  - ~~Exceptional control flow~~
  - ~~Asynchronous exceptions (interrupts)~~
  - ~~Synchronous exceptions (traps & faults)~~
- **Processes**
  - Creating new processes
  - Fork and wait
  - Zombies

Creating New Processes & Programs

- **fork-exec model:**
  - **`fork()`** creates a copy of the current process
  - **`execve()`** replaces the current process' code & address space with the code for a different program

- **`fork()` and `execve()` are *system calls***

- **Other system calls for process management:**
  - **`getpid()`**
  - **`exit()`**
  - **`wait()` / `waitpid()`**

`fork`: Creating New Processes

`pid_t fork(void)`

- creates a new process (child process) that is identical to the calling process (parent process), including all state (memory, registers, etc.)
  - returns 0 to the **child** process
  - returns child's process ID (`pid`) to the **parent** process
- Child is *almost* identical to parent:
  - Child gets an identical (but separate) copy of the parent's virtual address space.
  - Child has a different PID than the parent

```
pid_t pid = fork();
if (pid == 0) {
   printf("hello from child\n");
} else {
   printf("hello from parent\n");
}
```

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

Understanding fork

Process x (parent)

```
pid_t pid = fork();
if (pid == 0) {
   printf("hello from child\n");
} else {
   printf("hello from parent\n");
}
```

Understanding fork

Process x (parent)

```
pid_t pid = fork();
if (pid == 0) {
   printf("hello from child\n");
} else {
   printf("hello from parent\n");
}
```

Process y (child)

```
pid_t pid = fork();
if (pid == 0) {
   printf("hello from child\n");
} else {
   printf("hello from parent\n");
}
```

Understanding fork

Process x (parent)

```
pid_t pid = fork();
if (pid == 0) {
   printf("hello from child\n");
} else {
   printf("hello from parent\n");
}
```

pid = y

```
pid_t pid = fork();
if (pid == 0) {
   printf("hello from child\n");
} else {
   printf("hello from parent\n");
}
```

Process y (child)

```
pid_t pid = fork();
if (pid == 0) {
   printf("hello from child\n");
} else {
   printf("hello from parent\n");
}
```

pid = 0

```
pid_t pid = fork();
if (pid == 0) {
   printf("hello from child\n");
} else {
   printf("hello from parent\n");
}
```

Understanding fork

Process x (parent)

Process y (child)

```
pid_t pid = fork();
if (pid == 0) {
   printf("hello from child\n");
} else {
   printf("hello from parent\n");
}
```

```
pid_t pid = fork();
if (pid == 0) {
   printf("hello from child\n");
} else {
   printf("hello from parent\n");
}
```

pid = y

```
pid_t pid = fork();
if (pid == 0) {
   printf("hello from child\n");
} else {
   printf("hello from parent\n");
}
```

pid = 0

```
pid_t pid = fork();
if (pid == 0) {
   printf("hello from child\n");
} else {
   printf("hello from parent\n");
}
```

```
pid_t pid = fork();
if (pid == 0) {
   printf("hello from child\n");
} else {
   printf("hello from parent\n");
}
```

```
pid_t pid = fork();
if (pid == 0) {
   printf("hello from child\n");
} else {
   printf("hello from parent\n");
}
```

`hello from parent`

Which one is first?

`hello from child`

Fork Example

- **Parent and child both run the same code**
  - Distinguish parent from child by return value from **`fork()`**
  - Which process runs first after the **`fork()`** is undefined
    - Could be parent, could be child!
- **Start with same state, but each has a *private* copy**
  - Same variables, same call stack, same file descriptors, same register contents, **same program counter**…

```
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);
}
```

Fork Example

```
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);
}
```

Fork Example

```
  void fork1()
  {
1.     int x = 1;
2.     pid_t pid = fork();
3.     if (pid == 0) {
4.      printf("Child has x = %d\n", ++x);
5.     } else {
6.      printf("Parent has x = %d\n", --x);
7.     }
8.     printf("Bye from process %d with x = %d\n", getpid(), x);
  }
```

- **Both processes continue/start execution at line 3**
  - Child does not call fork! The instruction pointer points to the instruction after the call to fork, the instruction pointer is part of the state that is copied to the child
  - Child has variable pid = 0, parent has pid = process ID of child
- **Concurrent execution:** Can't predict execution order of parent and child
- **Duplicate but separate address space:** both processes have a copy of x
  - `x` has a value of 1 at line 3 in both parent and child
  - Subsequent changes to `x` are independent
- **Shared open files:** `stdout` is the same in both parent and child

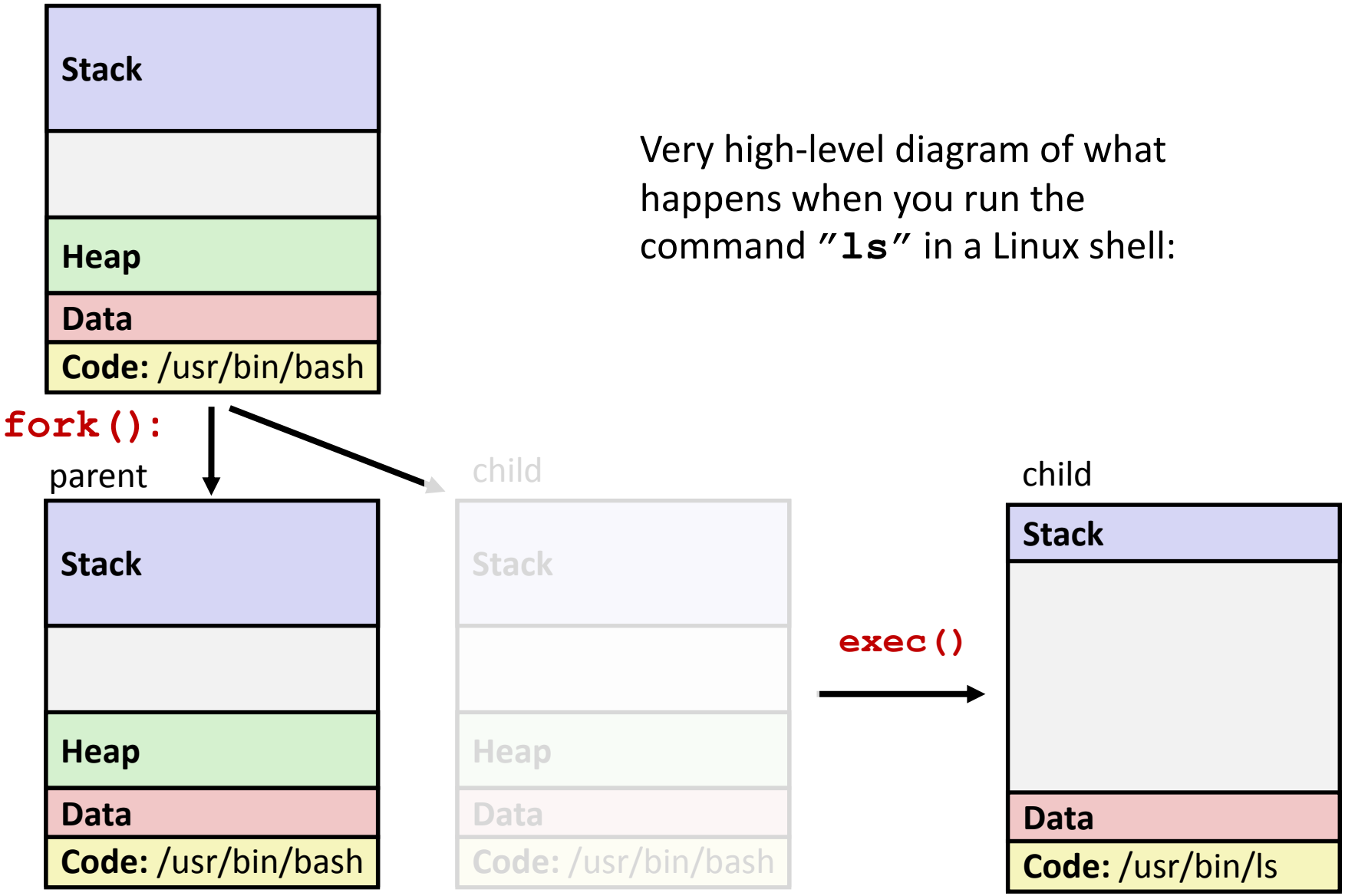
Fork-Exec

Note: the return values of **fork** & **execv** should be checked for errors.

- **fork-exec model:**
  - **fork()** creates a copy of the current process
  - **execve()** replaces the current process' code & address space with the code for a different program
    - There is a whole family of **exec** calls – see **exec(3)** and **execve(2)**

```
// Example arguments: path="/usr/bin/ls",
//     argv[0]="/usr/bin/ls", argv[1]="-ahl", argv[2]=NULL
void fork_exec(char *path, char *argv[])
{
    pid_t pid = fork();
    if (pid != 0) {
        printf("Parent: created a child %d\n", pid);
    } else {
        printf("Child: exec-ing new program now\n");
        execv(path, argv);
    }
    printf("This line printed by parent only!\n");
}
```

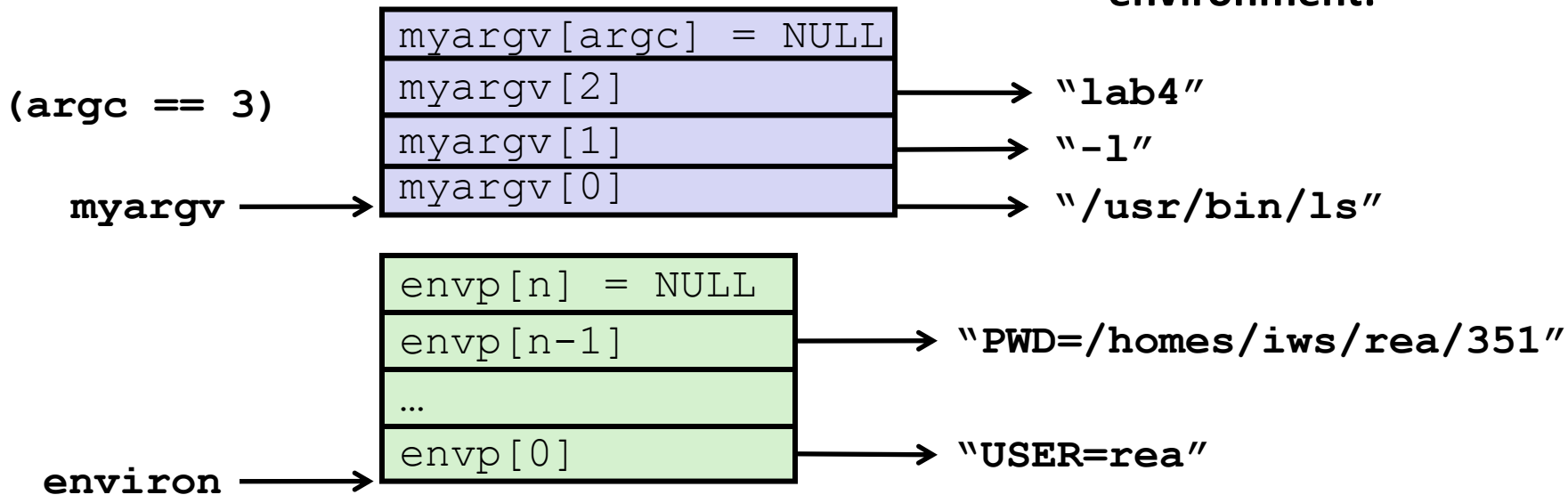
Exec-ing a new program



Very high-level diagram of what happens when you run the command "`ls`" in a Linux shell:

execve Example

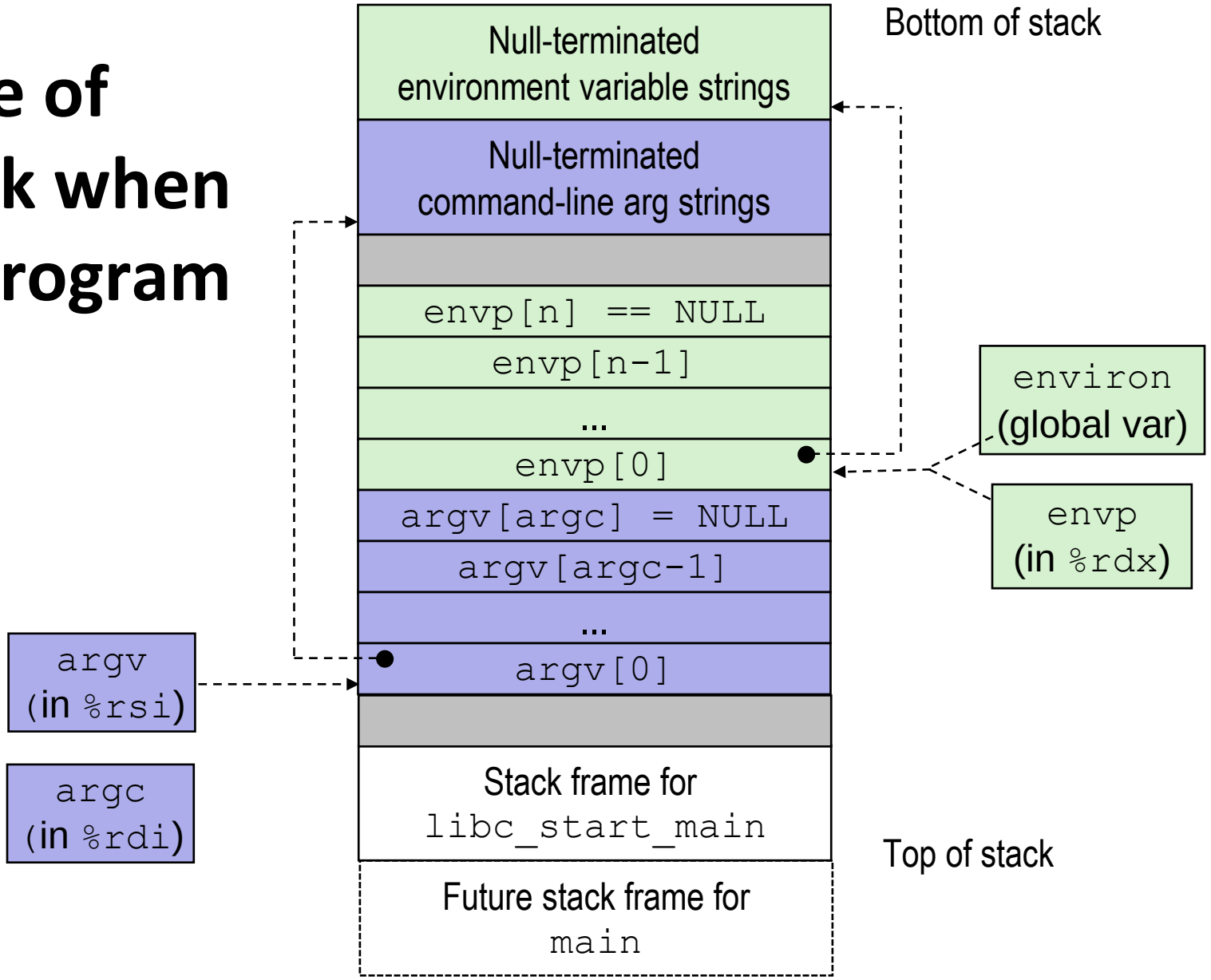
Execute "/usr/bin/ls -l lab4" in child process using current environment:




```
if ((pid = Fork()) == 0) {    /* Child runs program */
    if (execve(myargv[0], myargv, environ) < 0) {
        printf("%s: Command not found.\n", myargv[0]);
        exit(1);
    }
}
```

Run the **printenv** command in a linux shell to see your own environment variables.

Structure of the stack when a new program starts



`exit`: Ending a process

- **`void exit(int status)`**
  - Exits a process
    - Status code: 0 is used for a normal exit, nonzero for abnormal exit
  - **`atexit()`** registers functions to be executed upon exit

```
void cleanup(void) {
    printf("cleaning up\n");
}

void fork6() {
    atexit(cleanup);
    fork();
    exit(0);
}
```

"cleanup" is a function pointer

Processes

- **First some preliminaries**
  - ~~Control flow~~
  - ~~Exceptional control flow~~
  - ~~Asynchronous exceptions (interrupts)~~
  - ~~Synchronous exceptions (traps & faults)~~
- **Processes**
  - ~~Creating new processes~~
  - Fork and wait
  - Zombies

Zombies

- **Idea**
  - When process terminates, it still consumes system resources
    - Various tables maintained by OS
  - Called a "zombie"
    - A living corpse, half alive and half dead
- **Reaping**
  - Performed by parent on terminated child
  - 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)
  - But in long-running processes we need *explicit* reaping
    - e.g., shells and servers

On more recent Linux systems, *init* has been renamed as "*systemd*".

`wait`: Synchronizing with Children

- **Parent reaps a child by calling the `wait` function**
- **`int wait(int *child_status)`**
  - Suspends current process (i.e. the parent) until one of its children terminates
  - Return value is the **`pid`** of the child process that terminated
    - On successful return, the child process is reaped
  - If **`child_status != NULL`**, then the **`int`** that it points to will be set to a value indicating why the child process terminated
    - NULL is a macro for address 0, the null pointer
    - There are special macros for interpreting this status – see **man wait(2)**
- If parent process has multiple children, **`wait()`** will return when *any* of the children terminates
  - **`waitpid()`** can be used to wait on a specific child process

Modeling `fork` with Process Graphs

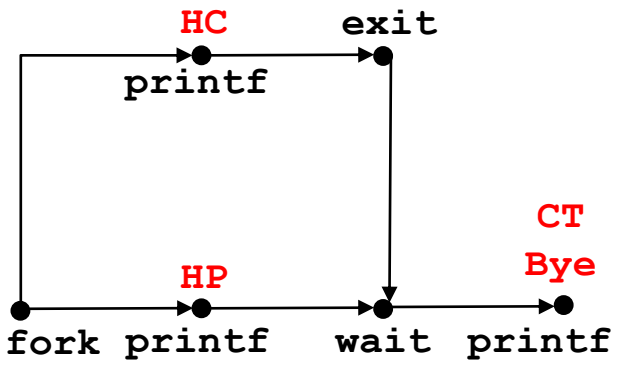
- **A *process graph* is a useful tool for capturing the partial ordering of statements in a concurrent program:**
  - Each vertex is the execution of a statement
  - a -> b means `a` happens before b
  - Edges can be labeled with current value of variables
  - `printf` vertices can be labeled with output
  - 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

`wait`: Synchronizing with Children

```
void fork_wait() {
    int child_status;

    if (fork() == 0) {
        printf("HC: hello from child\n");
        exit(0);
    } else {
        printf("HP: hello from parent\n");
        wait(&child_status);
        printf("CT: child has terminated\n");
    }
    printf("Bye\n");
}
```

forks.c



Feasible output:

HC
HP
CT
Bye

Infeasible output:

HP
CT
Bye
HC

`wait` Example #2

```
void fork_wait2() {
   int child_status;
   pid_t child_pid;

   if (fork() == 0) {
      printf("child!\n");
   } else {
      printf("parent!\n");
      child_pid = wait(&child_status);
}
   printf("Bye\n");
   exit(0);
}
```

Process management summary

- **`fork` gets us two copies of the same process (but `fork()` returns different values to the two processes)**
- **`execve` has a new process substitute itself for the one that called it**
  - Two-process program:
    - First **`fork()`**
    - if (pid == 0) { /* child code */ } else { /* parent code */ }
  - Two different programs:
    - First **`fork()`**
    - if (pid == 0) { **`execve()`** } else { /* parent code */ }
    - Now running two completely different programs
- **`wait` / `waitpid` used to synchronize parent/child execution and to reap child process**

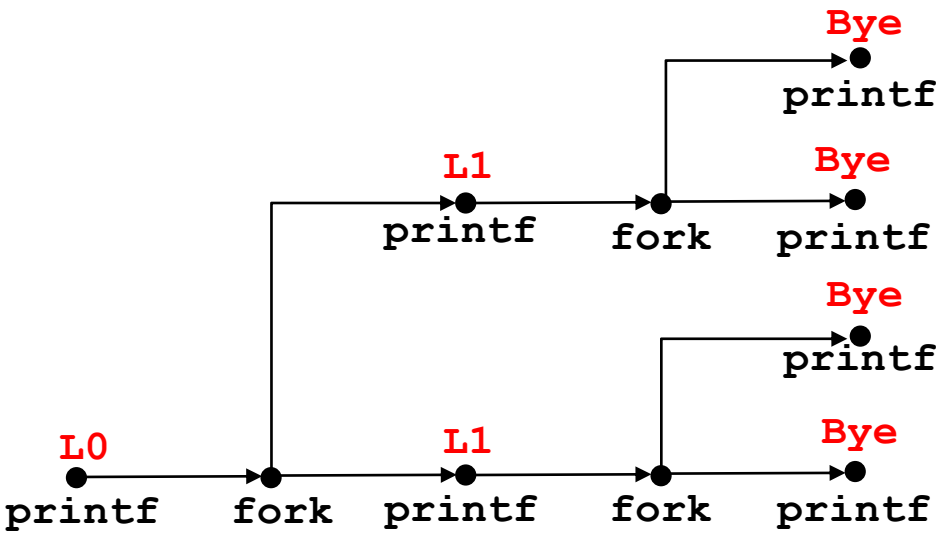
Summary

- **Processes**
  - At any given time, system has multiple active processes
  - On a one-CPU system, only one can execute at a time, but each process appears to have total control of the processor
  - OS periodically "context switches" between active processes
    - Implemented using *exceptional control flow*
- **Process management**
  - **`fork`**: one call, two returns
  - **`exec`**: one call, usually no return
  - **`wait`** or **`waitpid`**: synchronization
  - **`exit`**: one call, no return

Detailed examples

`fork` Example: Two consecutive `fork`s

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



Feasible output:

L0
L1
Bye
Bye
L1
Bye
Bye

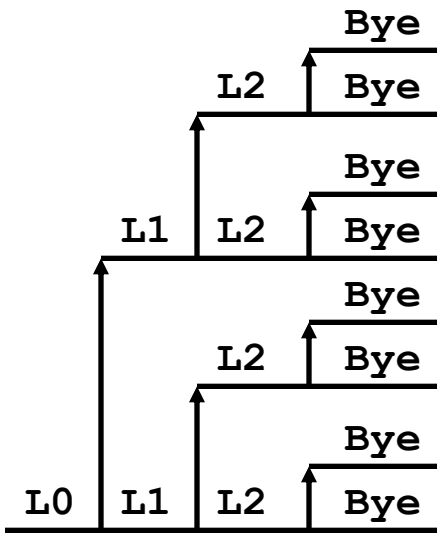
Infeasible output:

L0
Bye
L1
Bye
L1
Bye
Bye

`fork` Example: Three consecutive `fork`s

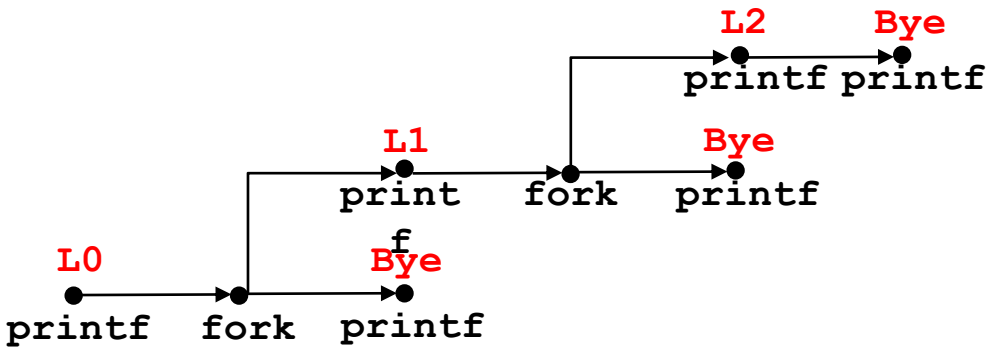
- Both parent and child can continue forking

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



fork Example: Nested forks in children

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



Feasible output:

L0
Bye
L1
L2
Bye
Bye

Infeasible output:

L0
Bye
L1
Bye
Bye
L2

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 */
    }
}
```

forks.c

```
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"
- Killing parent allows child to be reaped by **init**

Non-terminating Child Example

```
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);
    }
}
```

forks.c

```
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 explicitly, or else will keep running indefinitely

wait() Example

- 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 */
    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 terminated abnormally\n", wpid);
    }
}
```

`waitpid()`: Waiting for a Specific Process

`pid_t waitpid(pid_t pid, int &status, int options)`

- suspends current process until specific process terminates
- various options (that we won't talk about)

```
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);
    }
```