Exceptional Control Flow: Exceptions and Processes

Entire system

CS230 System Programming
10th Lecture

Instructors:

Jongse Park

hit keyboard, outside interruption, ...

Today

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

Control Flow

- **Processors do only one thing:** at a time
  - 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*) trace of instructions

Physical control flow

Time

<startup>
inst_1 they are data
inst_2
inst_3
…
inst_n
<shutdown>

Altering the Control Flow

- **Up to now: two mechanisms for changing control flow:**
  - Jumps and branches within program
  - Call and return ~> more restricted control flow

  React to changes in ***program state*** condition codes, flags, ...

- **Insufficient for a useful system:**
  Difficult to react to changes in *system state*
  - Data arrives from a disk or a network adapter ~> disk should notify processor should react
  - Instruction divides by zero ~> possible at runtime
  - User hits Ctrl-C at the keyboard
  - System timer expires ~> at certain period of time

  kill infinite loop

  create event → time is up!

- **System needs mechanisms for "exceptional control flow"**

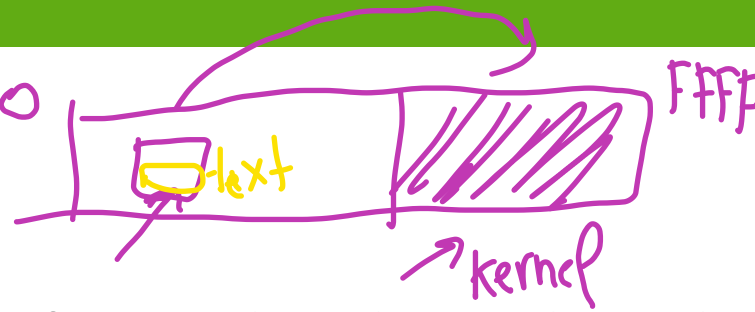
Exceptional Control Flow

- **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** → events created by software
    - Implemented by OS software
  - 4. **Nonlocal jumps**: `setjmp()` and `longjmp()` → jumping outside of memory space
    - Implemented by C runtime library

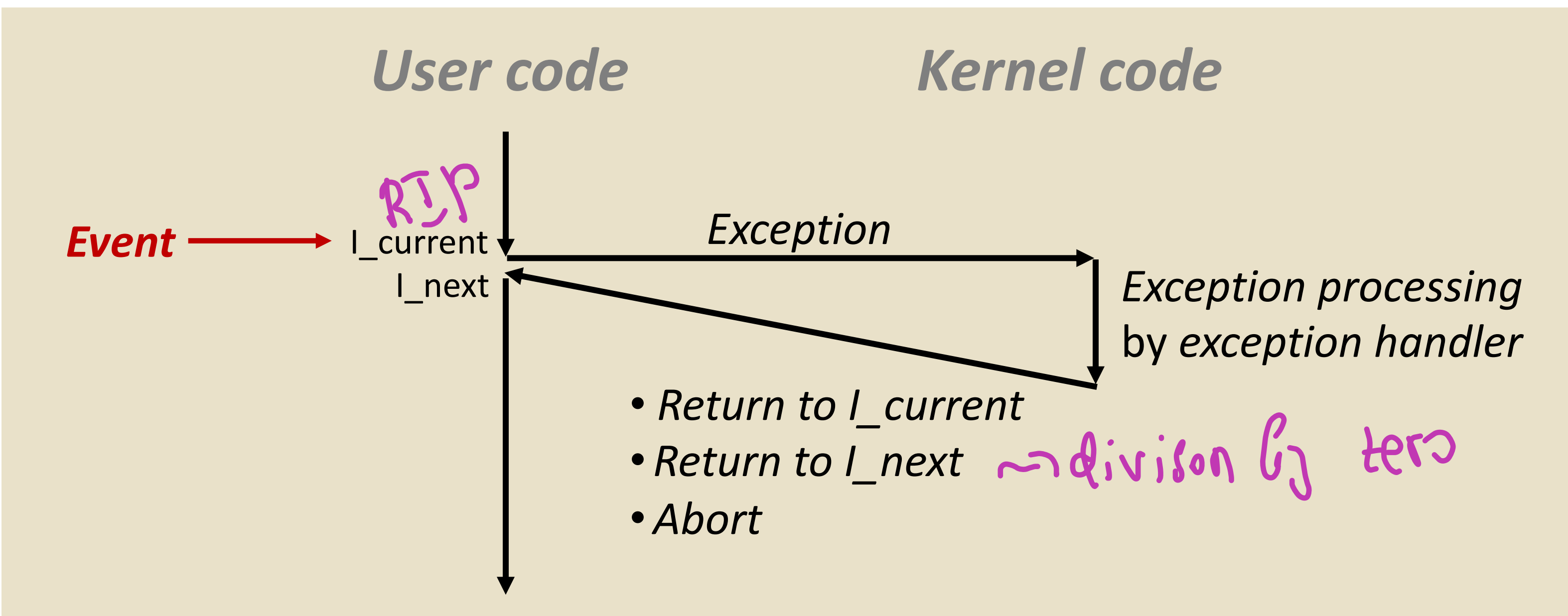
Today

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

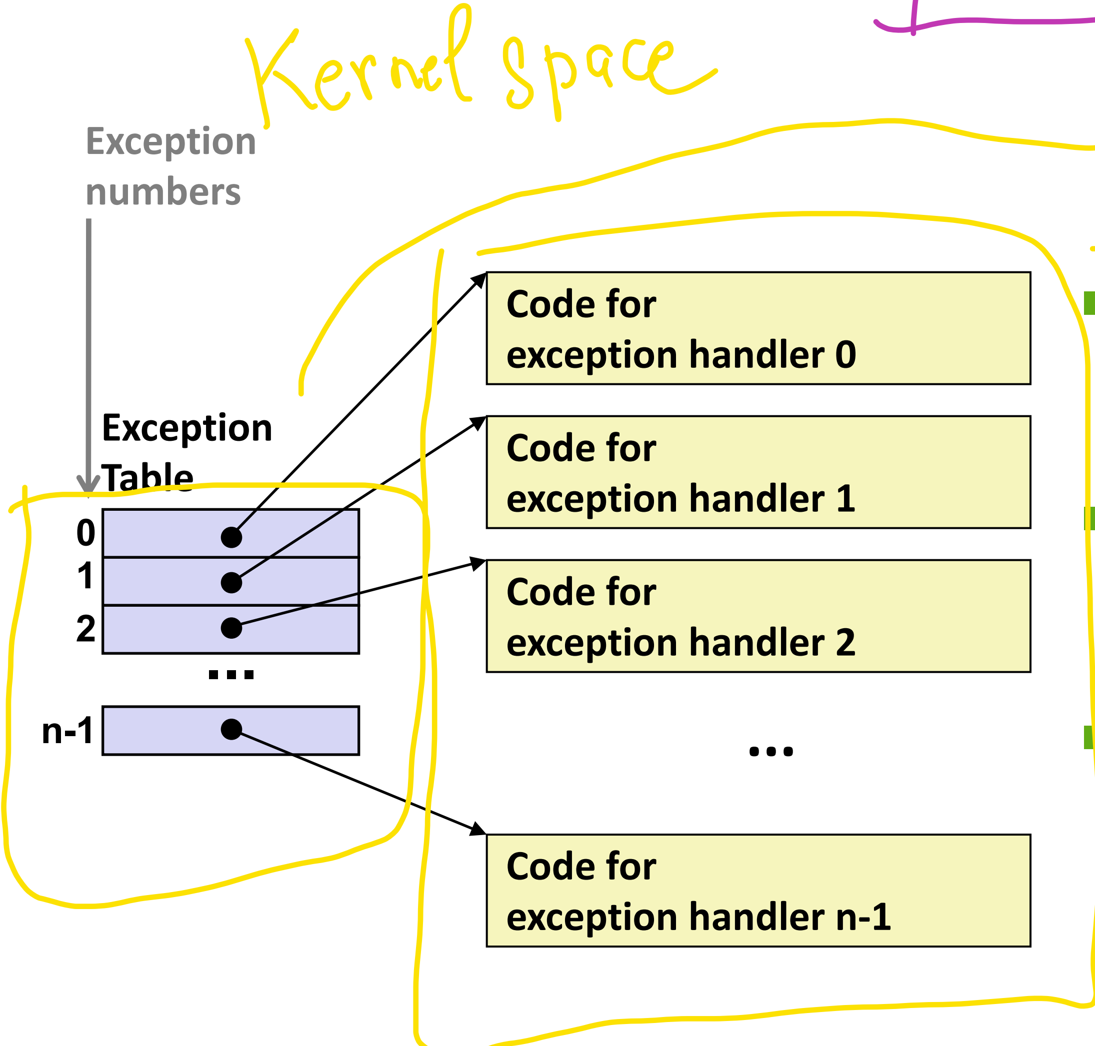


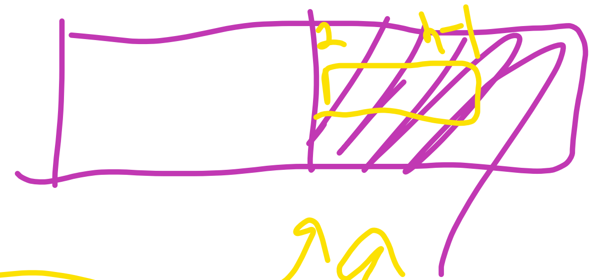
Exceptions

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

Asynchronous Exceptions (Interrupts)

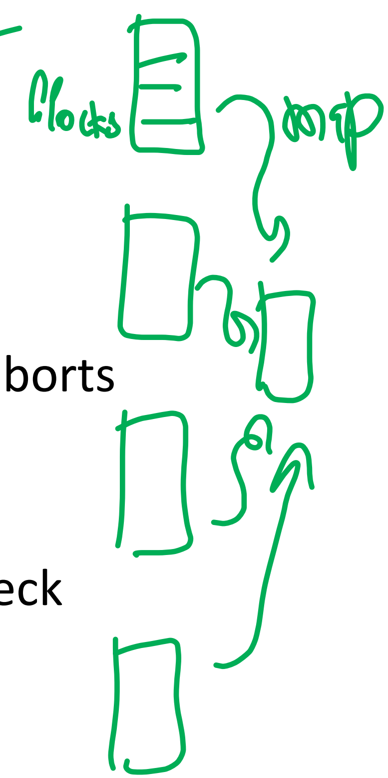
- **Caused by events external to the processor**
  - Indicated by setting the processor's *interrupt pin*
  - Handler returns to "next" instruction

- **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

not controlled by processor

Synchronous Exceptions

- **Caused by events that occur as a result of executing an instruction:**
  - ***Traps***
    - Intentional communicate with kernel (printf)
    - Examples: ***system calls***, breakpoint traps, special instructions
    - Returns control to "next" instruction bring from disk
  - ***Faults*** mov M[ ] → res
    - 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 never come back



System Calls

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

system → for calling, you go to kernel space
lib

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

↳ number as identifier (for kernel)

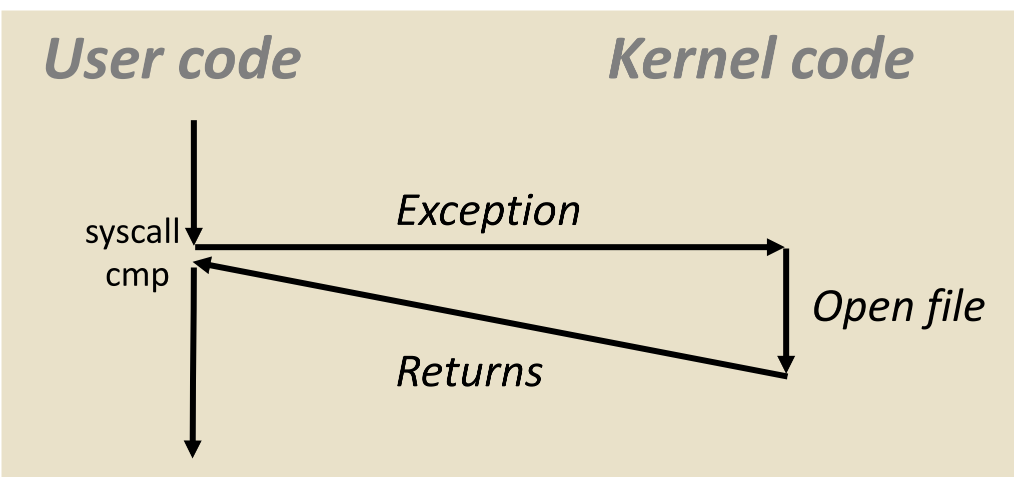
System Call Example: Opening File

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

as programmer, never initiate directly

```
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
```

goes to kernel



- `%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`

failed

Fault Example: Page Fault

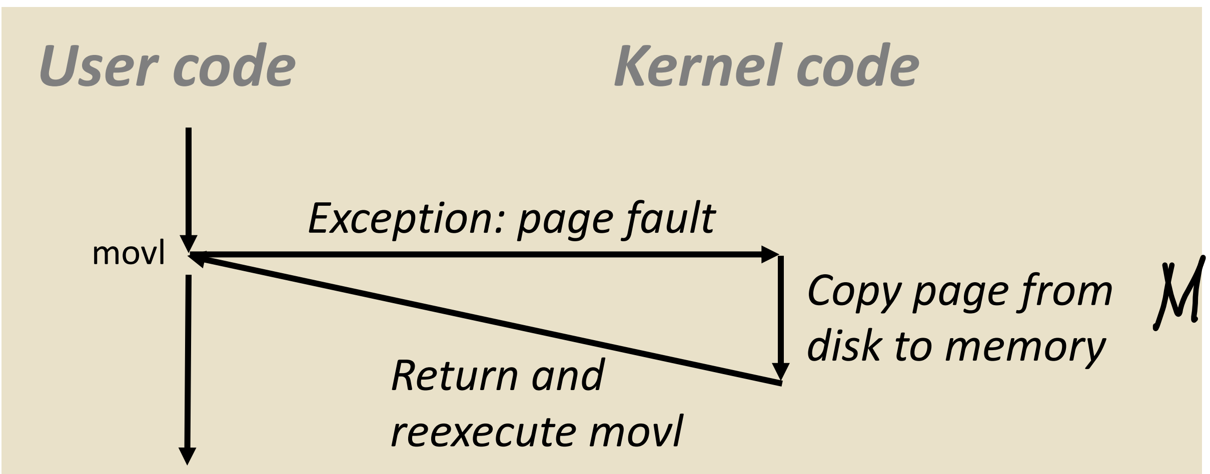
- User writes to memory location
- That portion (page) of user's memory is currently on disk

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

properly allocated

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

no guarantee a[500] physical location



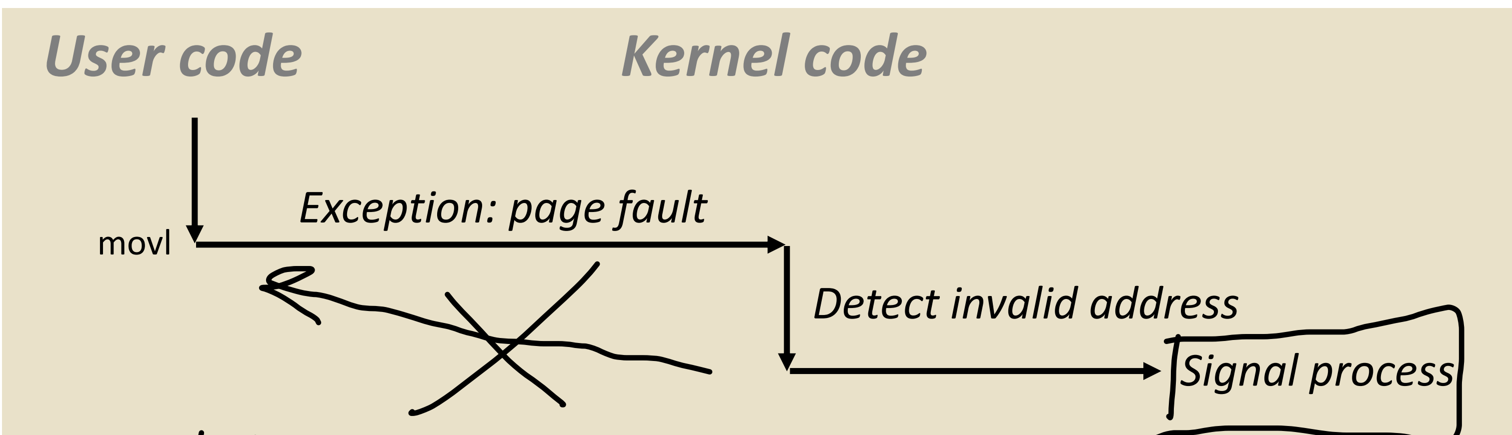
Memory controller

Fault Example: Invalid Memory Reference

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

not in a+4x1000 memory

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



segmentation fault   termination   kernel kills user program

- Sends **SIGSEGV** signal to user ~~process~~
- User process exits with "segmentation fault"

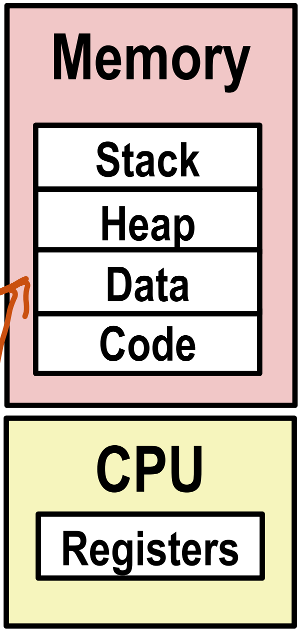
not always recoverable

Today

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

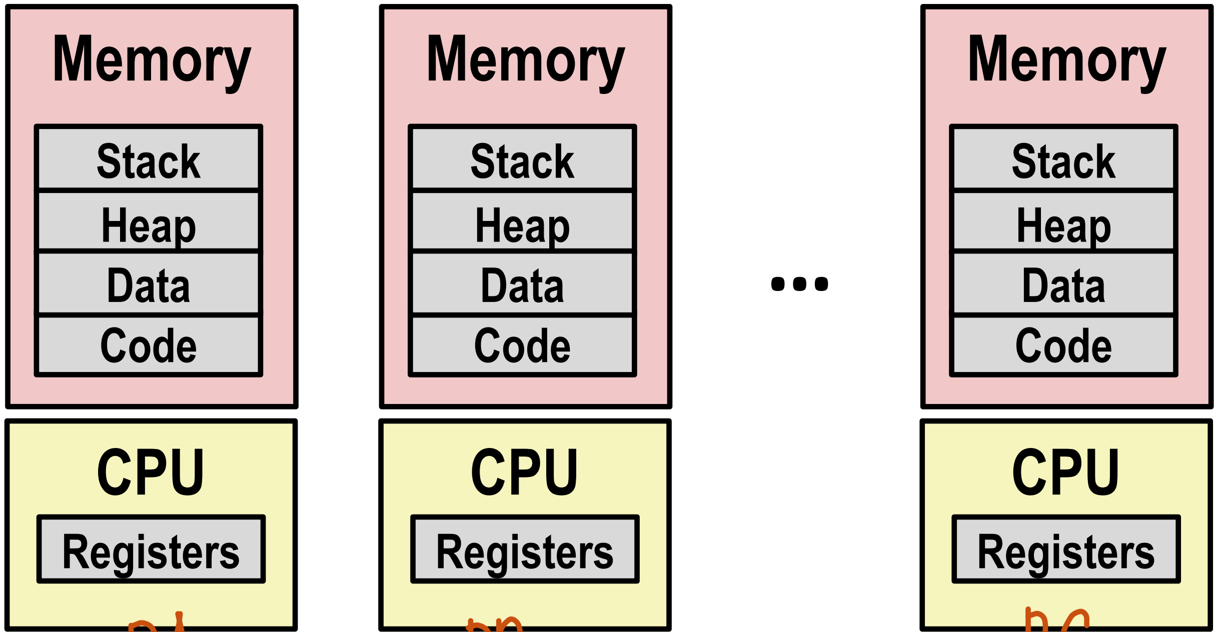
Processes

- **Definition: A *process* is an instance of a running program.** virtual entity
  - One of the most profound ideas in computer science
  - Not the same as "program" or "processor" ./prog
    ↳ permanent physical entity ./prog multiple processes
- **Process provides each program with two key abstractions:** time sharing
  conceptually
  - ***Logical control flow*** contiguously
    - Each program seems to have exclusive use of the CPU
      physically support multi-processes
    - 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

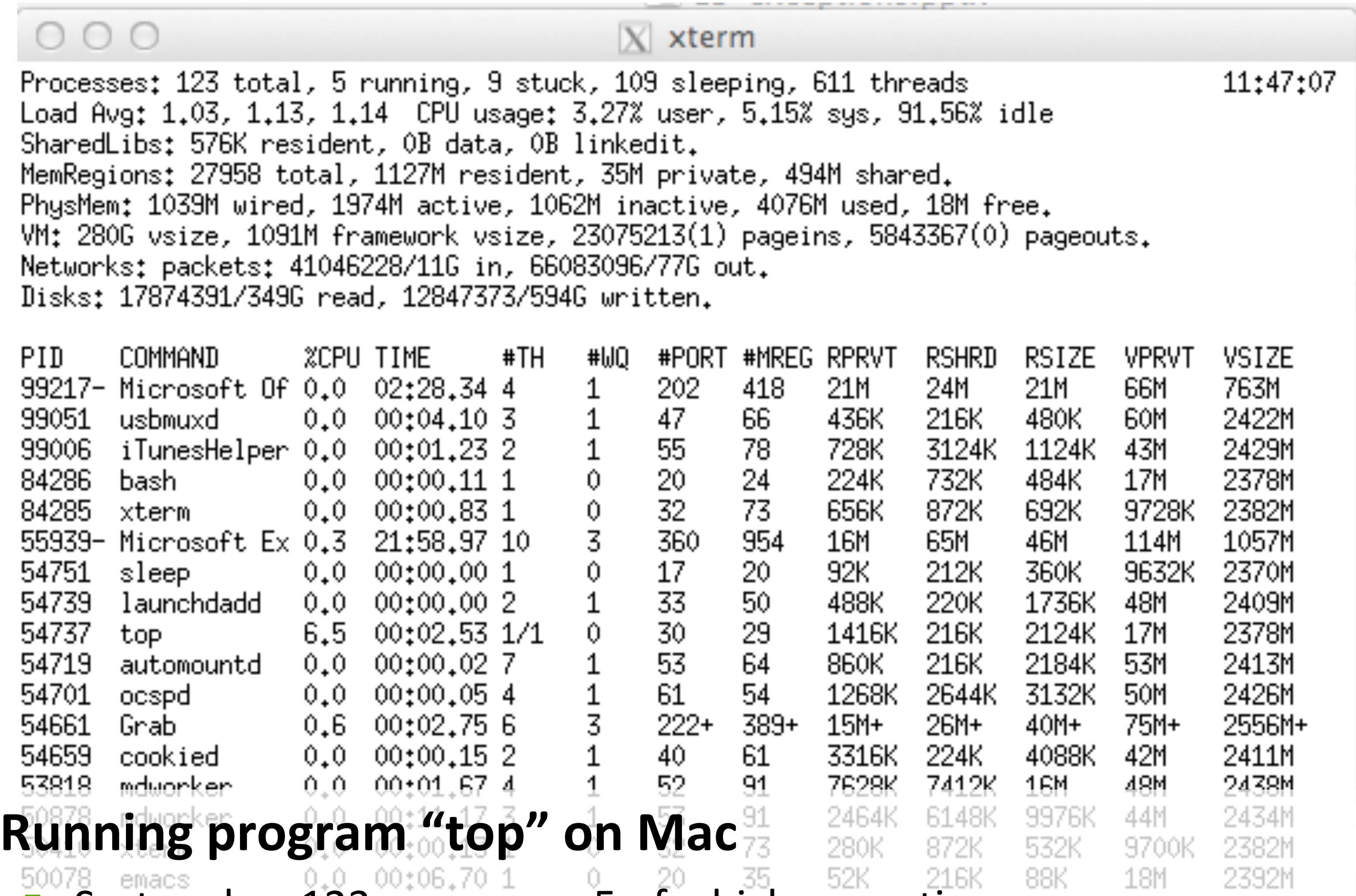
Pi → I've my own CPU, Memory



PA PB PC

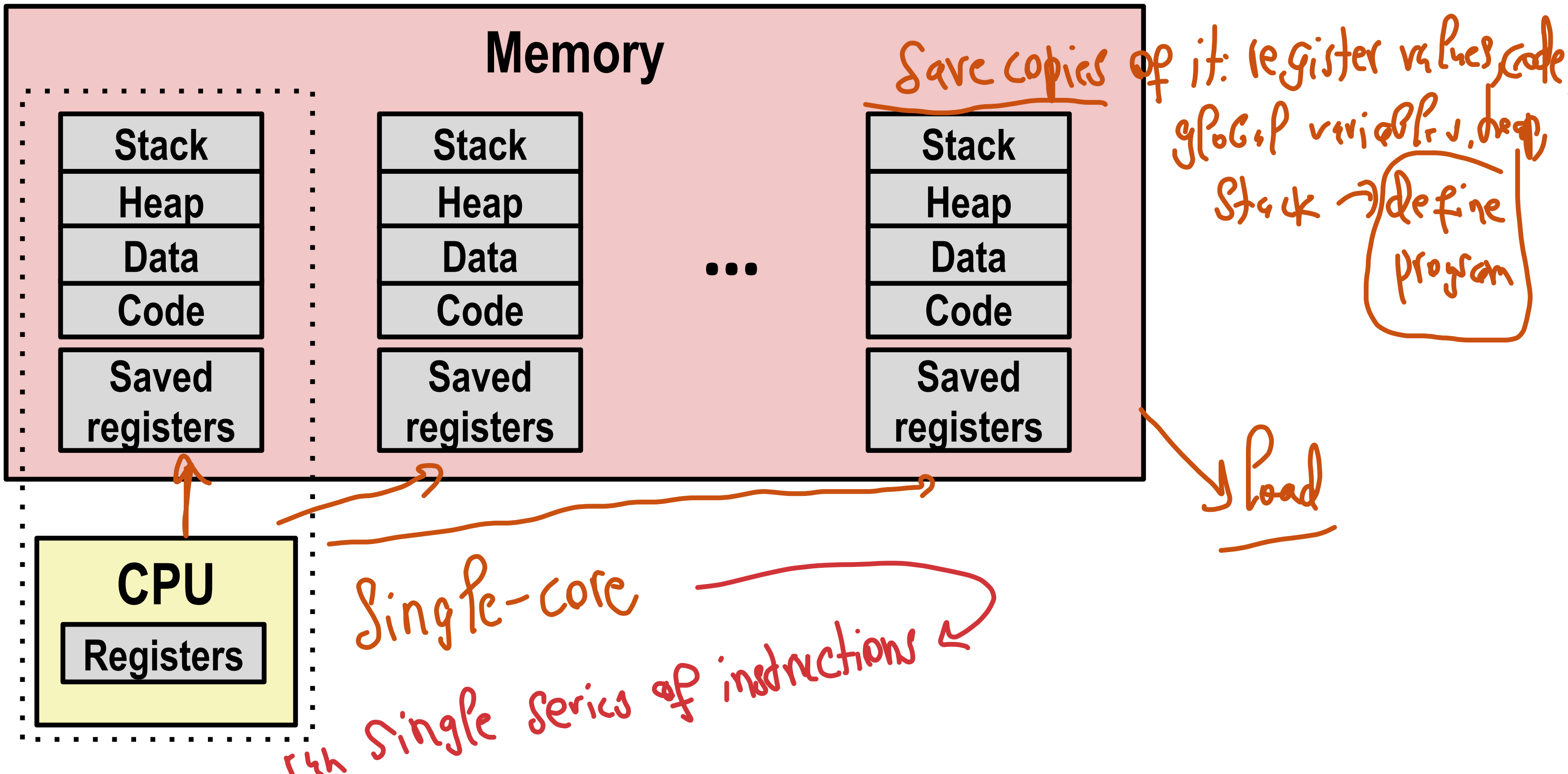
- **Computer runs many processes simultaneously** (together)
  time-sharing manner
  - Applications for one or more users
    - Web browsers, email clients, editors, …
  - Background tasks
    - Monitoring network & I/O devices

Multiprocessing Example



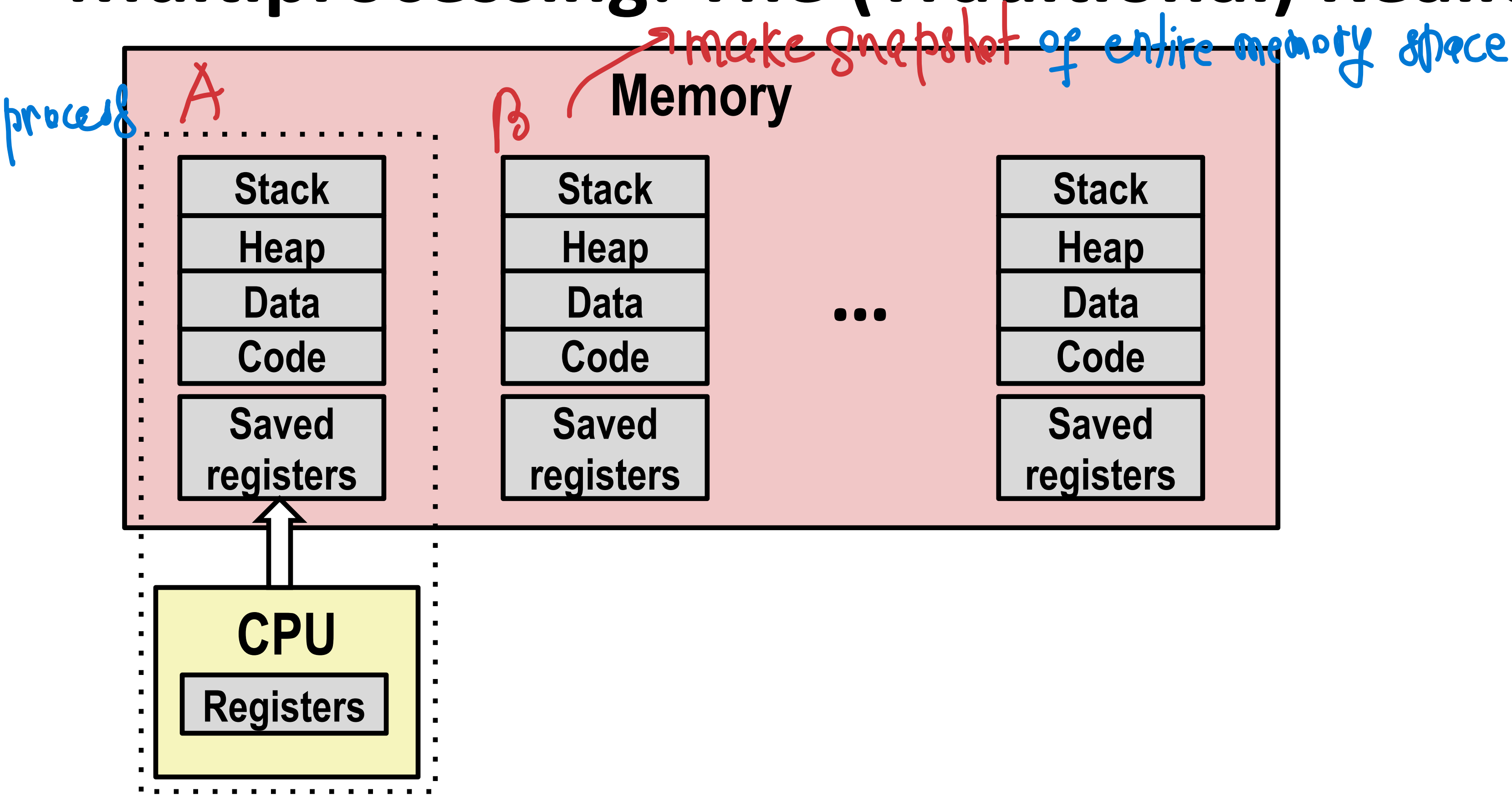
- **Running program "top" on Mac**
  - System has 123 processes, 5 of which are active
  - Identified by Process ID (PID)

Multiprocessing: The (Traditional) Reality



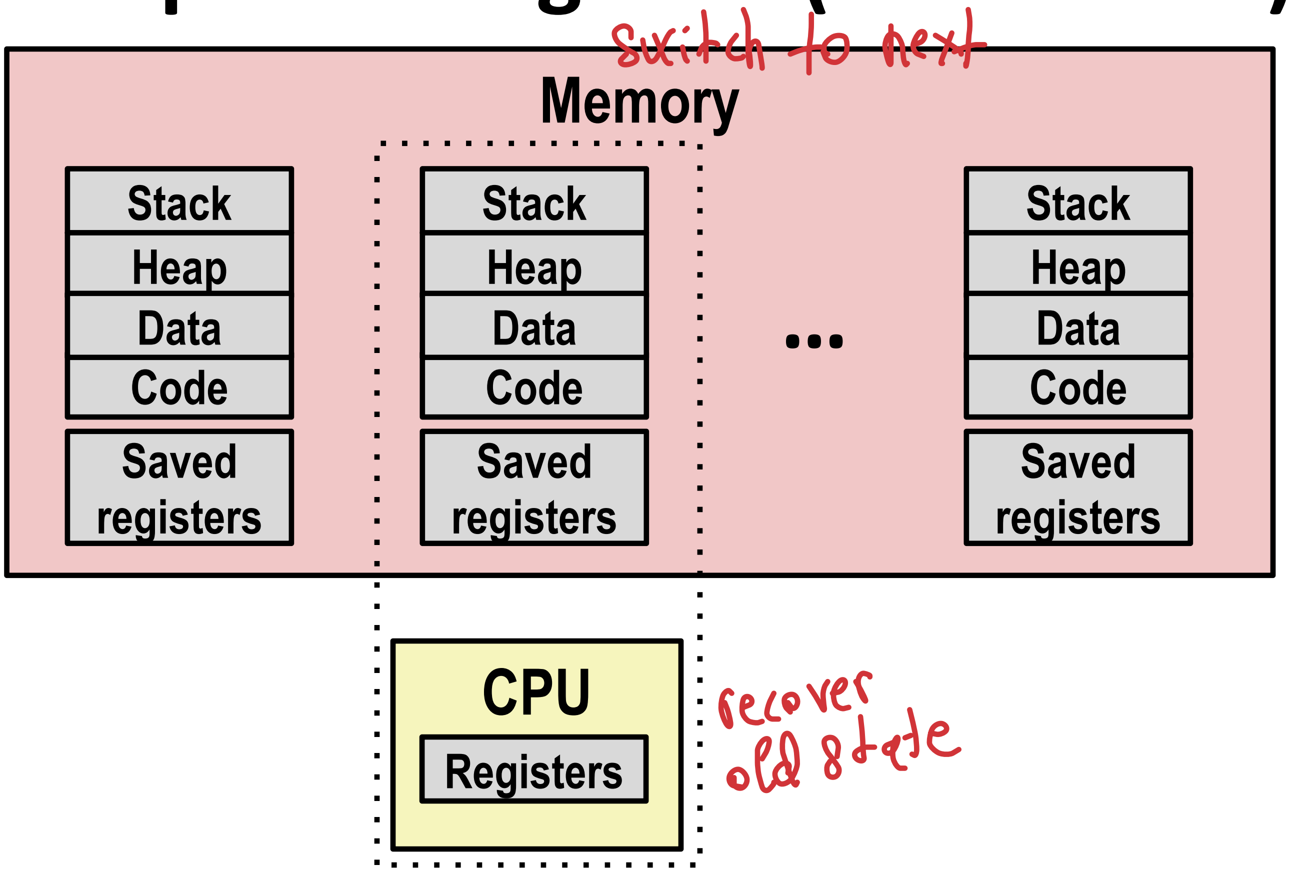
- **Single processor executes multiple processes concurrently**
  - Process executions interleaved (multitasking)
  - Address spaces managed by virtual memory system (later in course)
  - Register values for nonexecuting processes saved in memory

Multiprocessing: The (Traditional) Reality



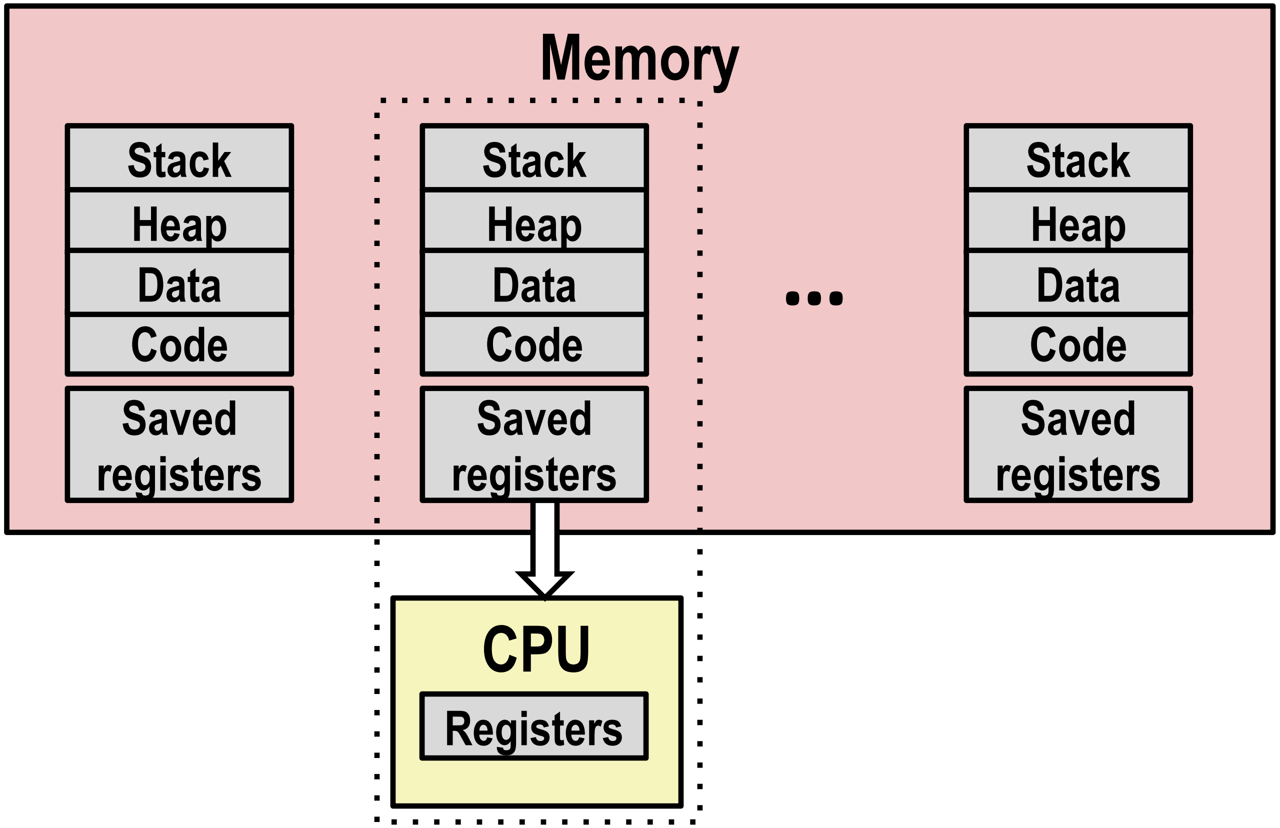
- Save current registers in memory

Multiprocessing: The (Traditional) Reality



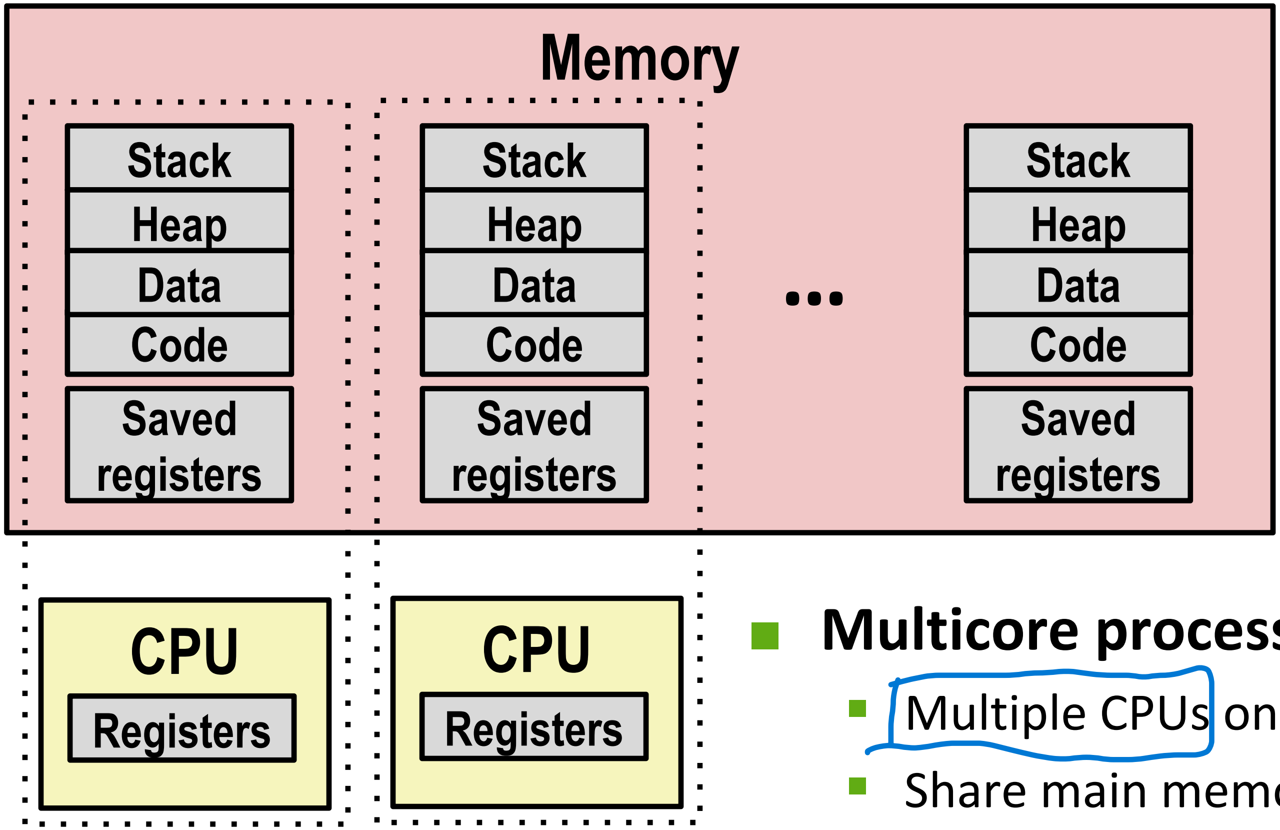
- Schedule next process for execution

Multiprocessing: The (Traditional) Reality



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

Multiprocessing: The (Modern) Reality



Context switching

run multiple instructions in parallel
≠ time-sharing

- **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

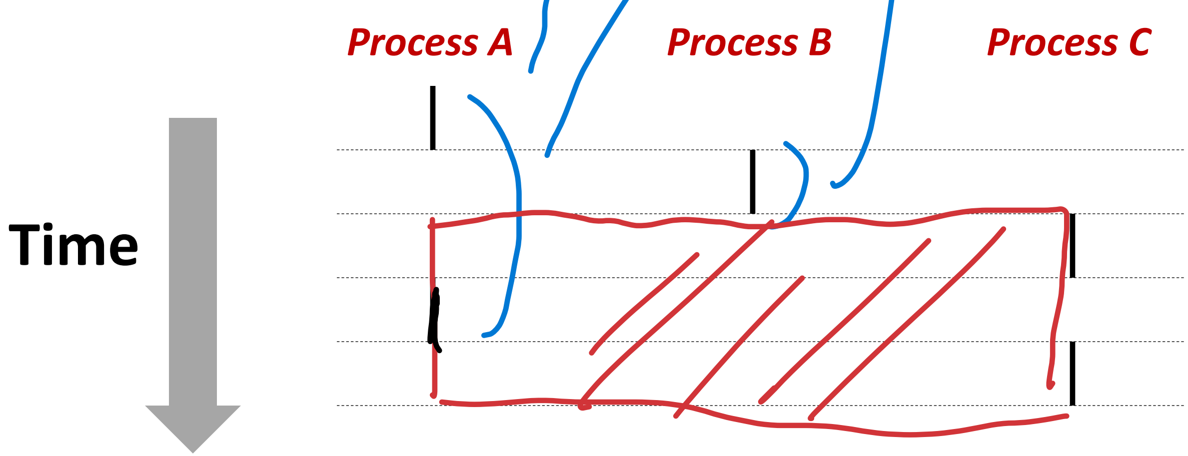
Concurrent Processes

≠ parallelite

- 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

Concurrent running

logical control flow of A

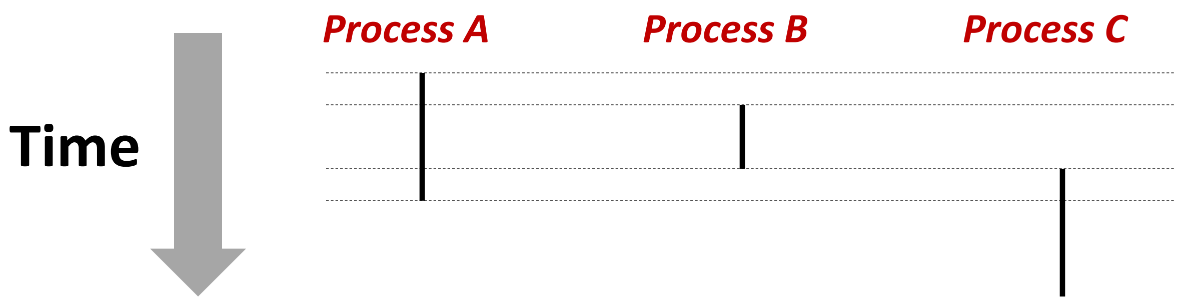


Running on same core

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**



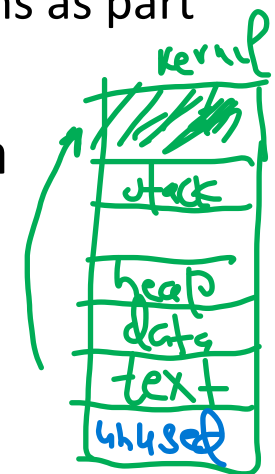
Context Switching

Loaded in DRAM

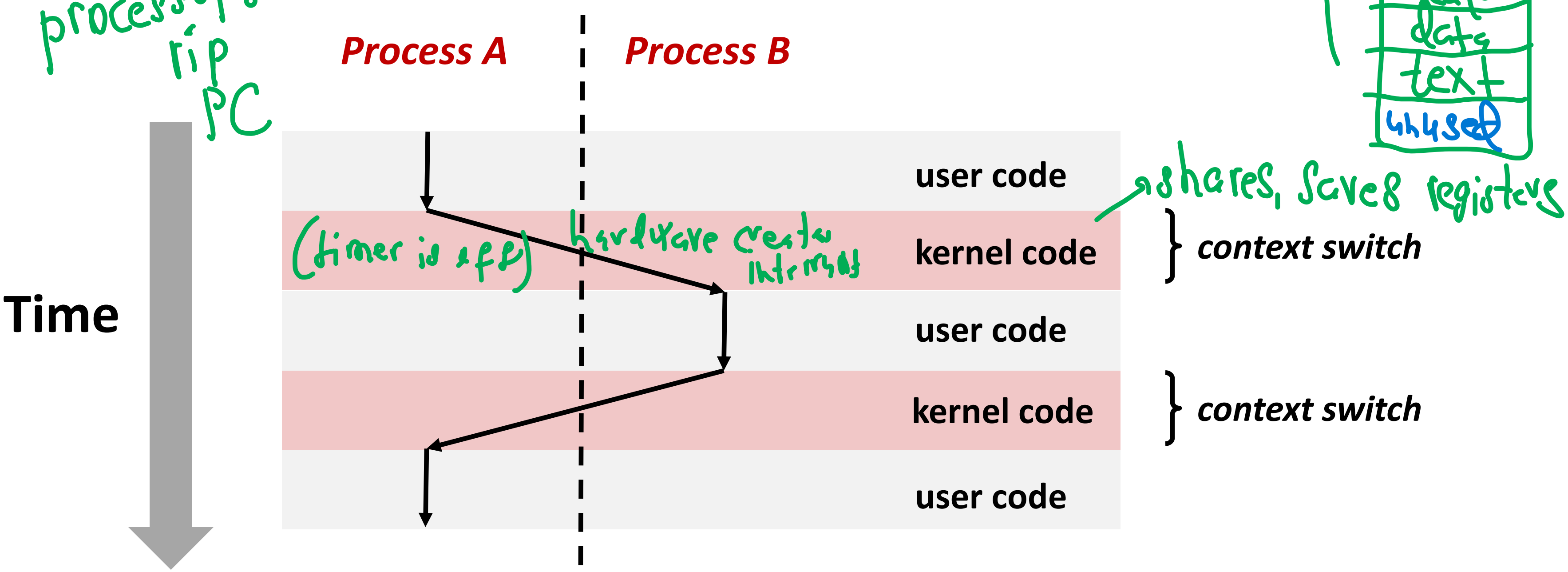
controls the processes

- **Processes are managed by a shared chunk of memory-resident OS code called the *kernel*** → core of OS
  - Important: the kernel is not a separate process, but rather runs as part of some existing process. → they are all shared
- **Control flow passes from one process to another via a *context switch***

processes have region for kernel



processor's rip PC



Today

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

System Call Error Handling

fail here no problem, but HANDLE IT!

- **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:**

wrapper for error-check

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

app

kernel

passing control to kernel

syscall

Error-reporting functions

when you call system-call

- **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

- 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;
}
```

fork() → system call

create process

getpid: get process ID

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

~→ simple, protective

Obtaining Process IDs

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

processes created by other processes
there's always parent process

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) Ctrl + Z does not schedule
- **Terminated**
  - Process is stopped permanently kill the process

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 → I'm done
  - Calling the `exit` function → system-call
    sending kernel: stop
- **`void exit(int status)`** → How I terminate?
  - 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

```
int main() {
    return 0;
}
```

- **`exit` is called once but never returns.**

Creating Processes

- ***Parent process*** **creates a new running** ***child process*** **by calling `fork`**

OS creates shell for you (process for the first)

Server →

function call

shell

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. (same global data, stack)
    - 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***

process { parent → child's PID ; child → 0 } return value

fork Example

x
0
x

```
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);
}
```

2 processes running

different in VM

fork.c

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

- Call once, return twice
- Concurrent execution
  Context switching
  - 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
    Seperate copies
  - Subsequent changes to **x** are independent
- Shared open files
  - `stdout` is the same in both parent and child

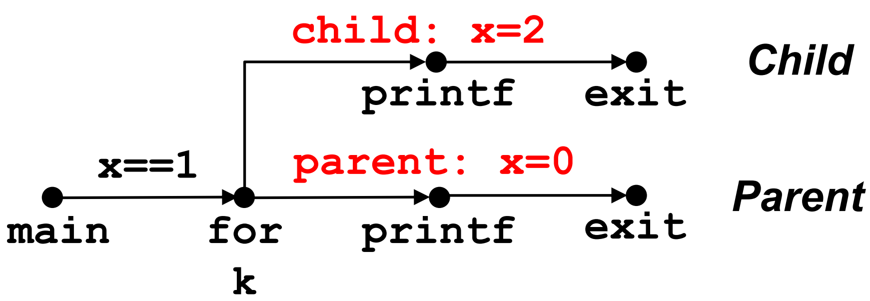
Modeling `fork` with Process Graphs

processes running concurrently

- **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 incoming 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

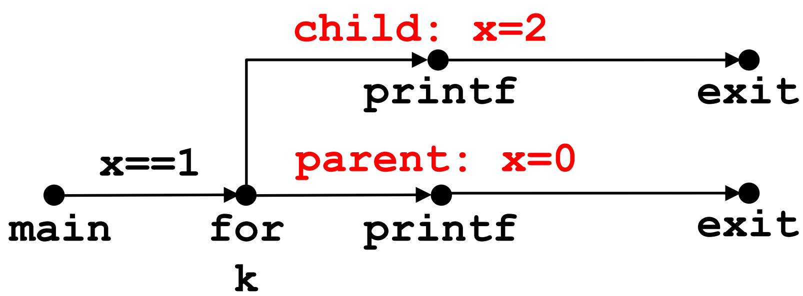
Process Graph Example

```
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);
}                                      fork.c
```

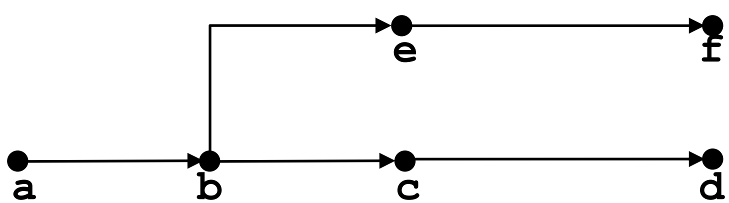


Interpreting Process Graphs

- Original graph:

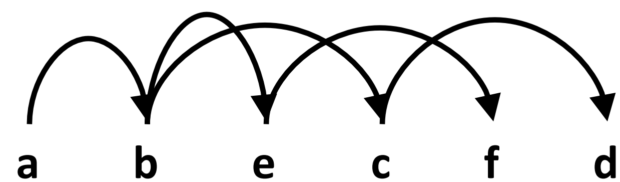


- Relabled graph:



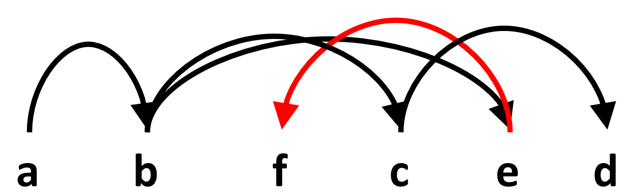
left-to-right ✓ correct

Feasible total ordering:



wrong

Infeasible total ordering:

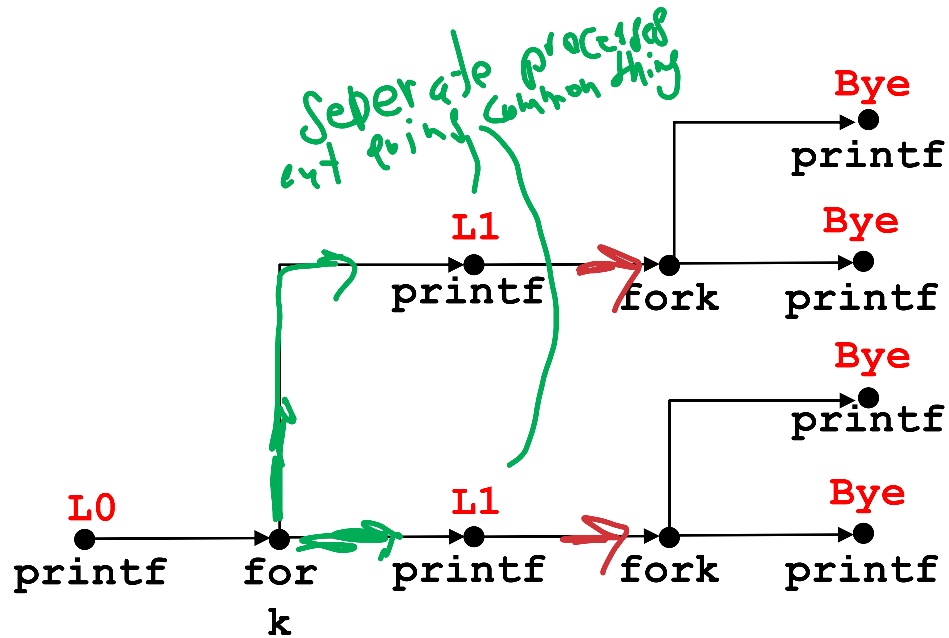


fork Example: Two consecutive forks

System call

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

forks.c



Feasible output:

L0
L1
Bye
Bye
L1
Bye
Bye

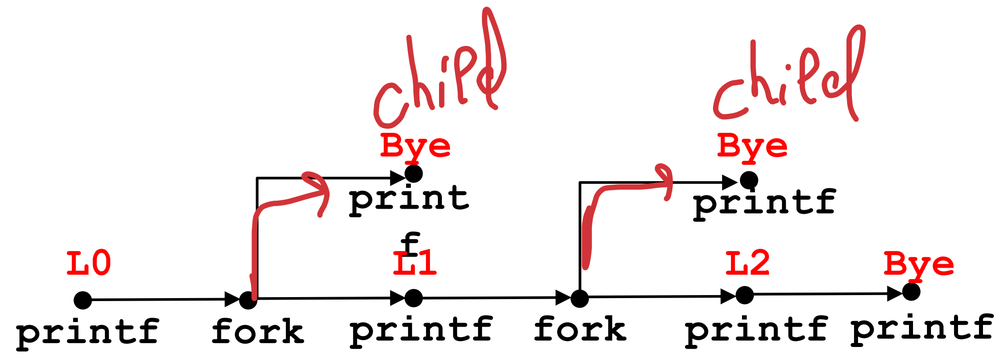
Infeasible output:

L0
Bye
L1
Bye
L1
Bye
Bye

Bye happens after L1

`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");
}                                  forks.c
```



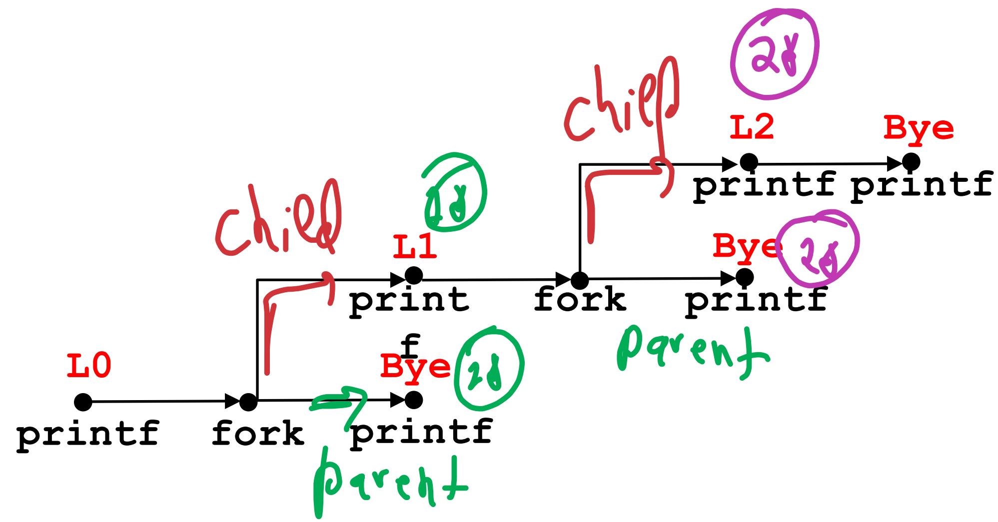
Feasible output:
L0
L1
Bye
Bye
L2
Bye

Infeasible output:
L0
Bye
L1
Bye
Bye
L2

`fork` Example: Nested `fork`s in children

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

forks.c



Feasible output:

L0
Bye
L1
L2
Bye
Bye

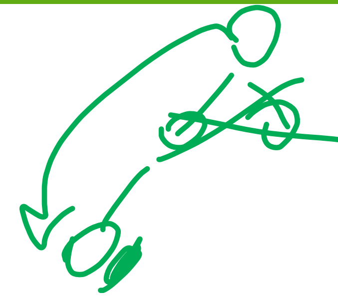
order can be different

Infeasible output:

L0
Bye
L1
Bye
Bye
L2

L2 should be present

Reaping Child Processes



- **Idea** child
  - When process terminates, it still consumes system resources
    - Examples: Exit status, various OS tables ← use these info
  - Called a "zombie" → some info in OS
    - Living corpse, half alive and half dead

system call

- **Reaping** → only by parent
  - Performed by parent on terminated child (using `wait` or `waitpid`)
  - Parent is given exit status information exit(0) parent waits for a certain process to be done
  - Kernel then deletes zombie child process
    cleans up ← then

- **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) → running in the system all the time
    → ultimate genesis process
  - 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 {   parent
        printf("Running Parent, PID = %d\n", getpid());
        while (1)
            ; /* Infinite loop */   parent infinite loop
    }                               never terminate
}                                   not zombie
                                                       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   manually kill parent process
[1]    Terminated   → parent process
linux> ps
  PID TTY          TIME CMD
 6585 ttyp9    00:00:00 tcsh
 6642 ttyp9    00:00:00 ps
→ not reaping child process (done by init)
```

- **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("Running Child, PID = %d\n",
               getpid());
        while (1)
            ; /* Infinite loop */
    } else {
        printf("Terminating Parent, PID = %d\n",
               getpid());
        exit(0);
    }
}
```

forks.c

child does not terminate

parent is gone

but child is remaining (running)

nobody kills running child process

```
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

Such inaccuracies can kill your system

parent process → zombie process

`wait`: Synchronizing with Children

explicitly

- Parent reaps a child by calling the `wait` function

pointer → status info

- `int wait(int *child_status)` Return → PID of child that terminated
  - Suspends current process until one of its children terminates
  - Return value is the **`pid`** of the child process that terminated
  - If **`child_status != NULL`**, then the integer it points to will be set to a value that indicates reason the child terminated and the exit status:
    - Checked using macros defined in `wait.h`
      - `WIFEXITED, WEXITSTATUS, WIFSIGNALED, WTERMSIG, WIFSTOPPED, WSTOPSIG, WIFCONTINUED` ?
      - See textbook for details

aligning steps

wait: Synchronizing with Children

matching behavior

```
void fork9() {
    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");
}
```

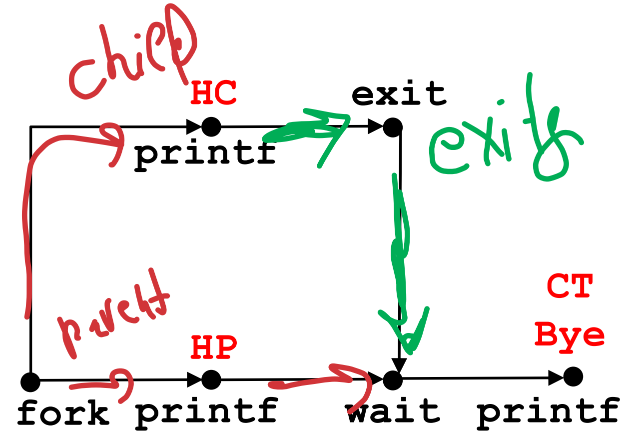
forks.c

child

parent

sleep

child terminates then you wake up



waiting for child process to terminate

shell

ls

> ls -al &

not waiting

shell will wait until ls finishes its job

Feasible output:
HC
HP
CT
Bye

Infeasible output:
HP
CT
Bye
HC

Another wait Example

for only 1 child to be completed

- 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, child_status;

    for (i = 0; i < N; i++)
        if ((pid[i] = fork()) == 0) {
            exit(100+i); /* Child */
        }
    for (i = 0; i < N; i++) { /* Parent */
        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 terminate abnormally\n", wpid);
    }
}
```

forks.c

parent does not get into loop body

obtain exit status from child status

PID of child

normal exit status

actual status

waitpid: Waiting for a Specific Process terminated

pass process ID to wait for specific child process to be

- `pid_t waitpid(pid_t pid, int &status, int options)`
  - Suspends current process until specific process terminates
  - Various options (see textbook)

backwards

Partly created child process to be terminated

```
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 = N-1; i >= 0; 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 terminate abnormally\n", wpid);
    }
}
```

forks.c

system call

execve: Loading and Running Programs

fork → creates copy (if/else)

Want to create process of ur own executable?

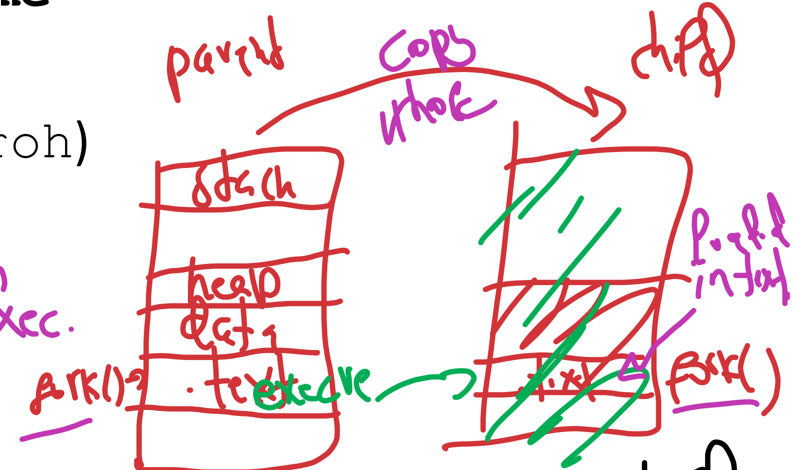
- `int execve(char *filename, char *argv[], char *envp[])`

  executable — environment variable

- **Loads and runs in the current process:**
  - Executable file **`filename`** → _ .sh
    - Can be object file or script file beginning with `#!interpreter` (e.g., `#!/bin/bash`) → treat as executable
  - …with argument list **`argv`**
    - By convention **`argv[0]==filename`**
  - …and environment variable list **`envp`**
    - "name=value" strings (e.g., `USER=droh`)
    - `getenv, putenv, printenv`
- **Overwrites code, data, and stack** with your exec.
  - Retains PID, open files and signal context

  running your exec from shell
- **Called once and never returns**
  - …except if there is an error

does not know about execve



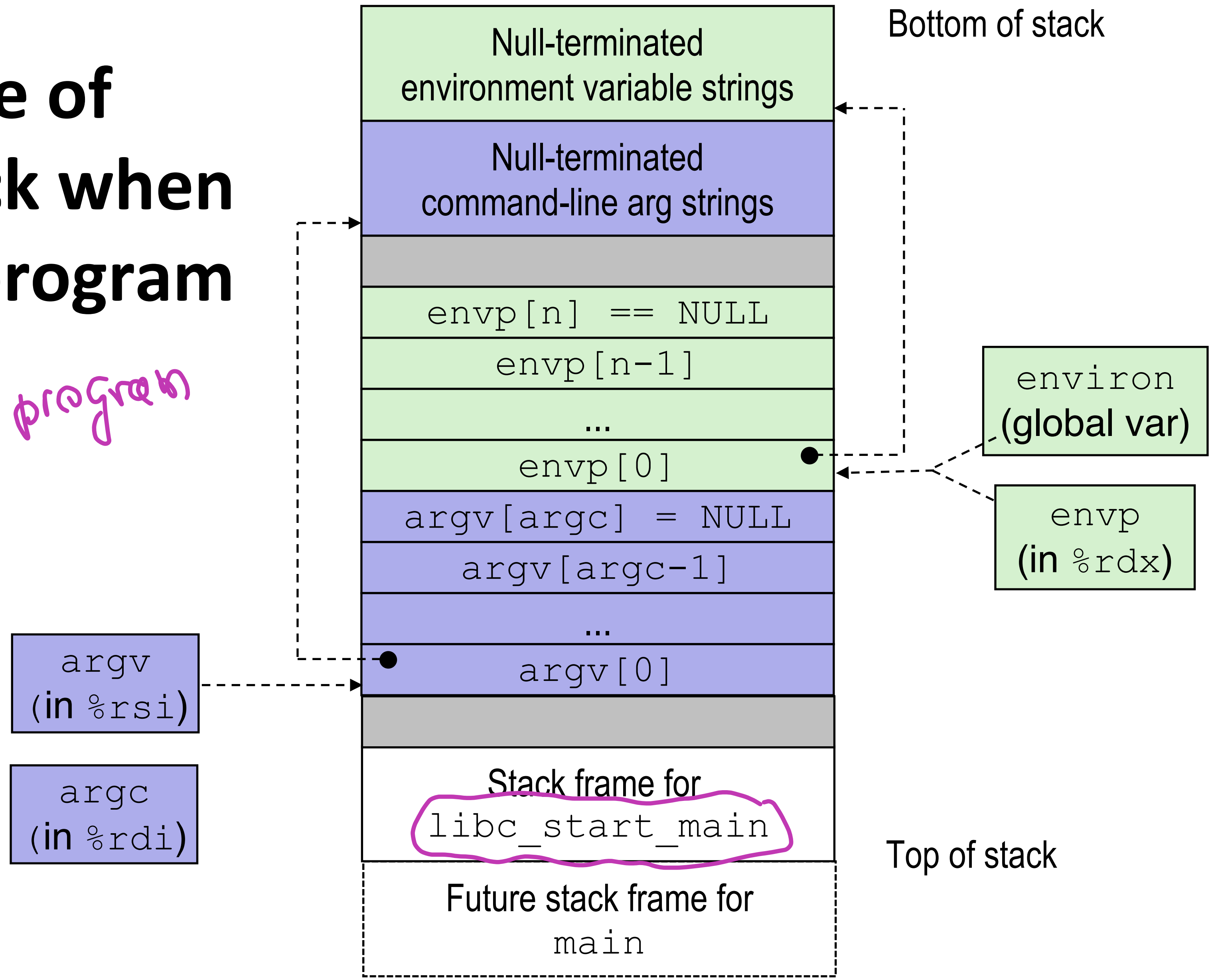
execve → overwrites child

it's present in parent process' data

Structure of the stack when a new program starts

VM ^space of new program

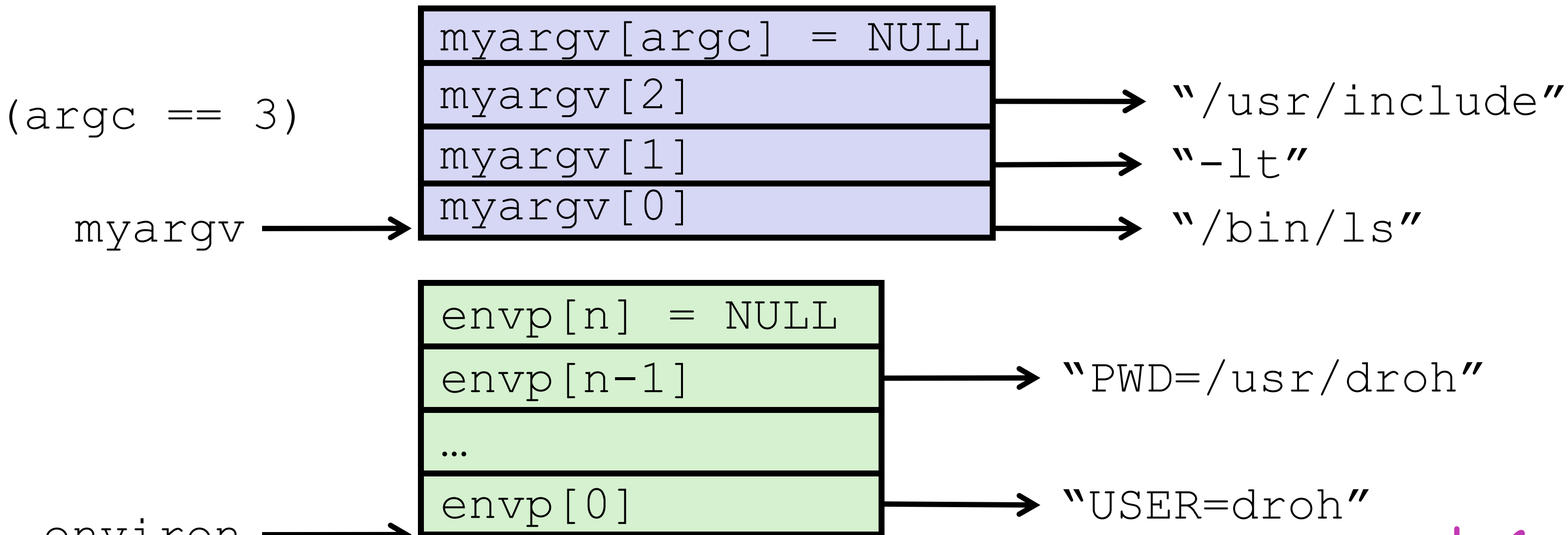
execve will overwrite arguments



execve Example

included as default

- Executes "/bin/ls -lt /usr/include" **in child process using current environment:**



why both → maybe after fork, I would initialize edit before execve

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

Summary

standard control flow → jump, loop, if/else
inside application

- **Exceptions**
  - Events that require nonstandard control flow (system event)
  - Generated externally (interrupts) or internally (traps and faults)
    page

- **Processes**
  - At any given time, system has multiple active processes
  - Only one can execute at a time on a single core, though
  - Each process appears to have total control of processor + private memory space) illusion of each process

Summary (cont.)

- **Spawning processes** duplicate exact same memory space
  - Call `fork`
  - One call, two returns
- **Process completion**
  - Call `exit`
  - One call, no return
- **Reaping and waiting for processes**
  - Call `wait` or `waitpid` Cleanup messes
- **Loading and running programs**
  - Call `execve` (or variant)
  - One call, (normally) no return