WELCOME
TO
15-213
14-513
18-213
18-613
15-513

Exceptional Control Flow: Exceptions and Processes

18-213/18-613: Introduction to Computer Systems
18th Lecture, July 12th, 2022

Today

Exceptional Control Flow	CSAPP 8
Exceptions	CSAPP 8.1
Processes	CSAPP 8.2
Process Control	CSAPP 8.3-8.4

Control Flow

- **Processors do only one thing:**
  - From startup to shutdown, each CPU core simply reads and executes (interprets) a sequence of instructions, one at a time *
  - This sequence is the CPU's *control flow* (or *flow of control*)

Physical control flow

<startup>
inst$_1$
inst$_2$
inst$_3$
…
inst$_n$
<shutdown>

* Externally, from an architectural viewpoint (internally, the CPU may use parallel out-of-order execution)

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

- **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

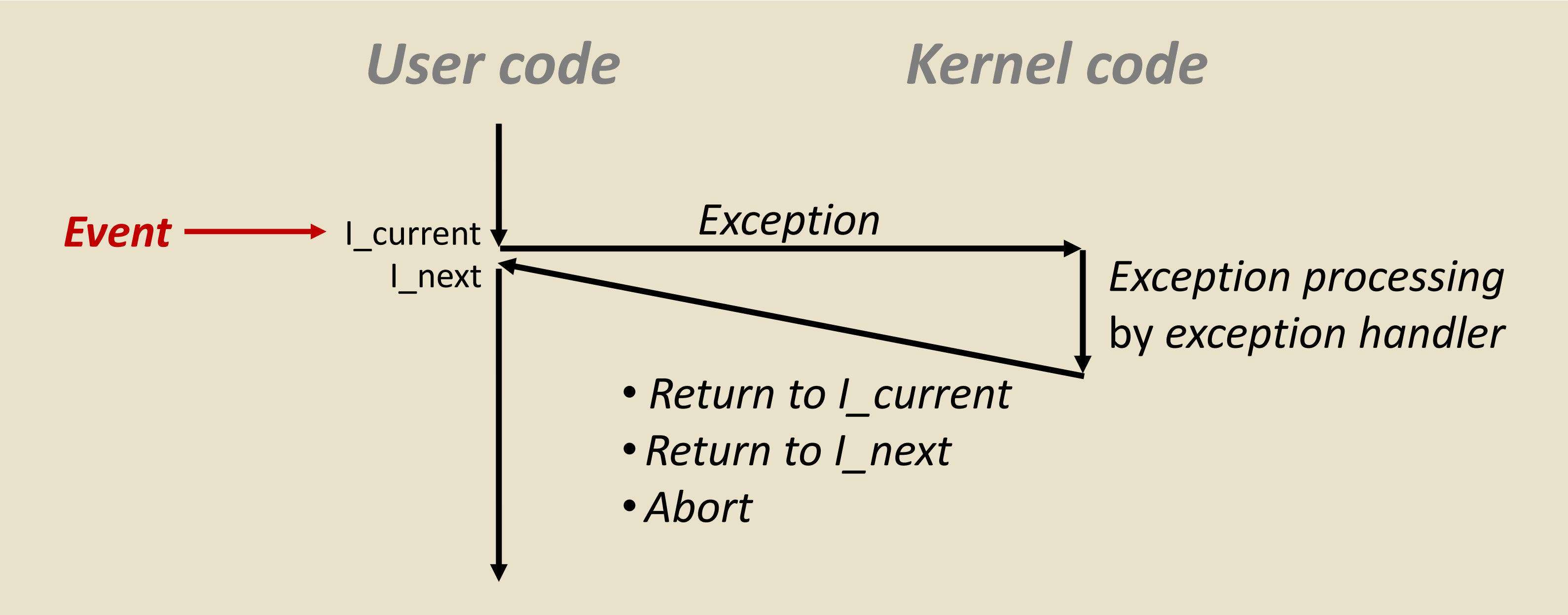
- **Higher level mechanisms**
  - 2. **Process context switch**
    - Implemented by OS software and hardware timer
  - 3. **Signals**
    - Implemented by OS software
  - 4. **Nonlocal jumps**: `setjmp()` and `longjmp()`
    - Implemented by C runtime library

Today

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

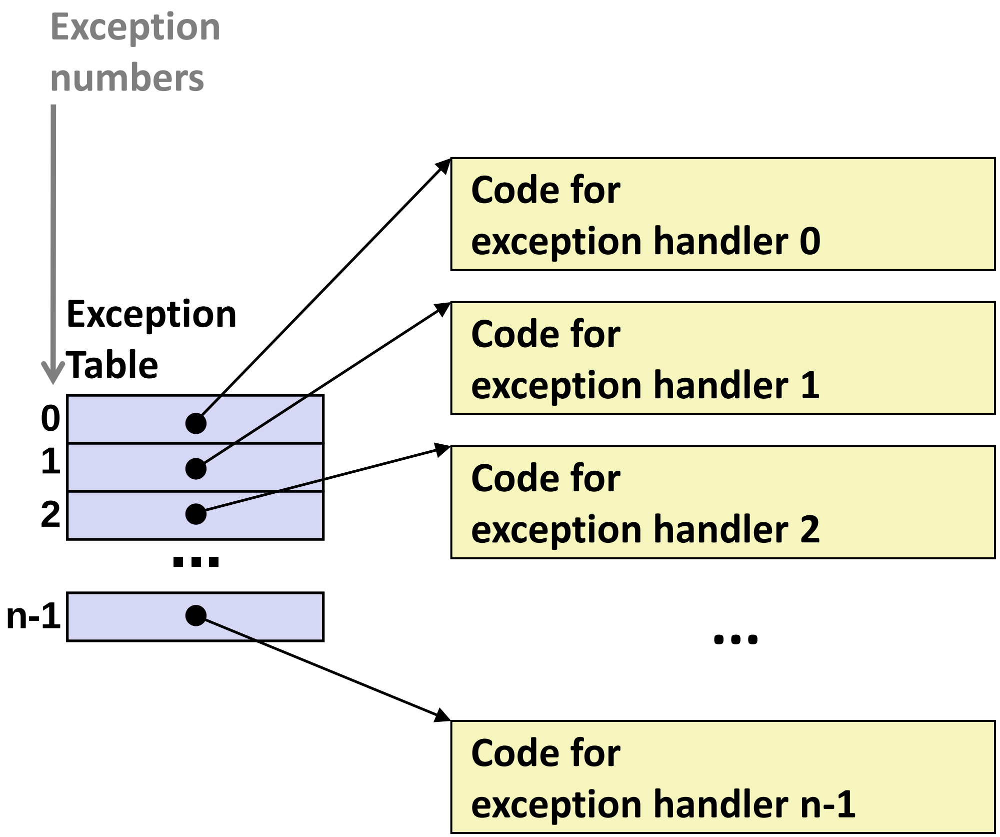
Exceptions

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



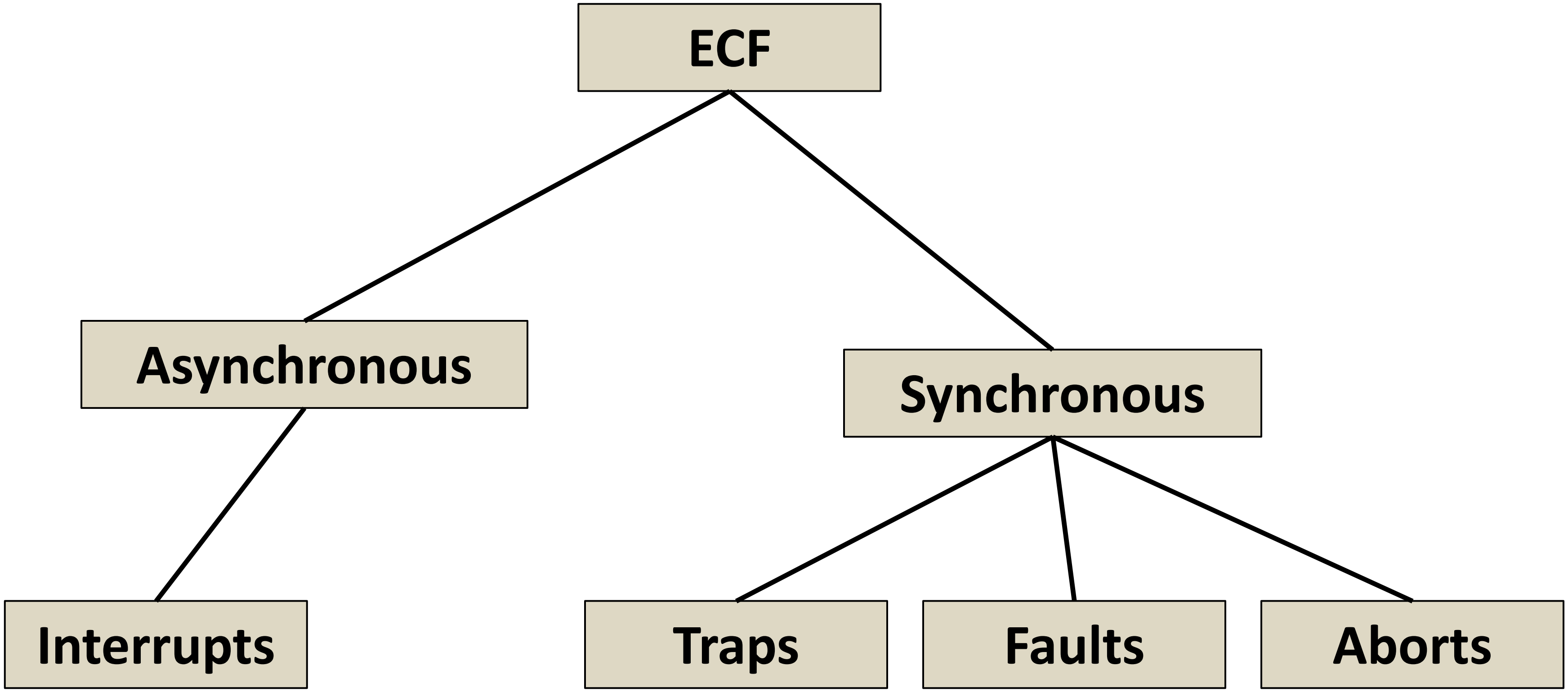
User code
Kernel code
Event
I_current
I_next
Exception
Exception processing by exception handler
• Return to I_current
• Return to I_next
• Abort

Exception Tables


Exception
numbers
Exception
Table
0
1
2
...
n-1
Code for
exception handler 0
Code for
exception handler 1
Code for
exception handler 2
...
Code for
exception handler n-1

- 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

(partial) Taxonomy


ECF
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

Synchronous Exceptions

- **Caused by events that occur as a result of executing an instruction:**
  - ***Traps***
    - Intentional, set program up to "trip the trap" and do something
    - Examples: ***system calls***, gdb breakpoints
    - Returns control to "next" instruction
  - ***Faults***
    - Unintentional but possibly recoverable
    - Examples: page faults (recoverable), protection faults (unrecoverable), floating point exceptions
    - Either re-executes faulting ("current") instruction or aborts
  - ***Aborts***
    - Unintentional and unrecoverable
    - Examples: illegal instruction, parity error, machine check
    - Aborts current program

System Calls

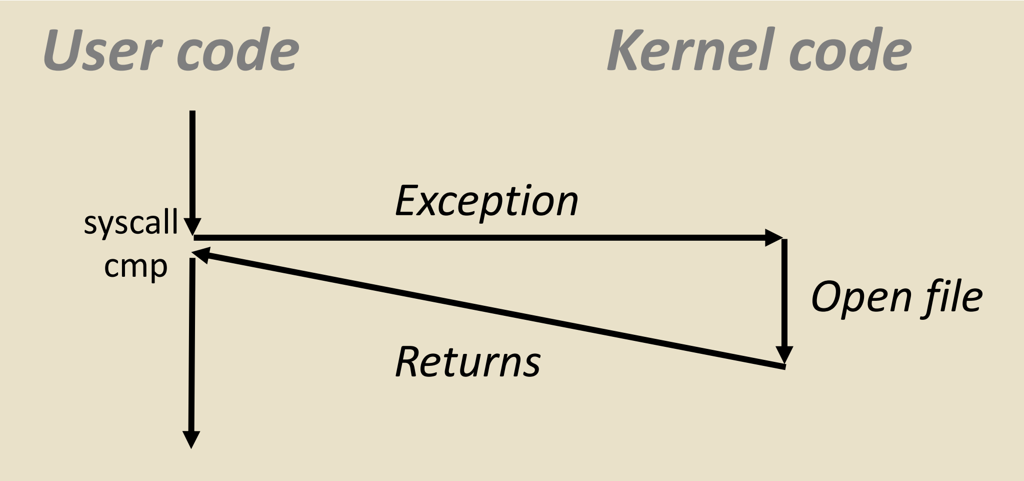
- **Each x86-64 system call has a unique ID number**
- **Examples:**

Number	*Name*	*Description*
0	`read`	Read file
1	`write`	Write file
2	`open`	Open file
3	`close`	Close file
4	`stat`	Get info about file
57	`fork`	Create process
59	`execve`	Execute a program
60	`_exit`	Terminate process
62	`kill`	Send signal to process

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
```


User code
Kernel code
Exception
syscall
cmp
Open file
Returns

- `%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`

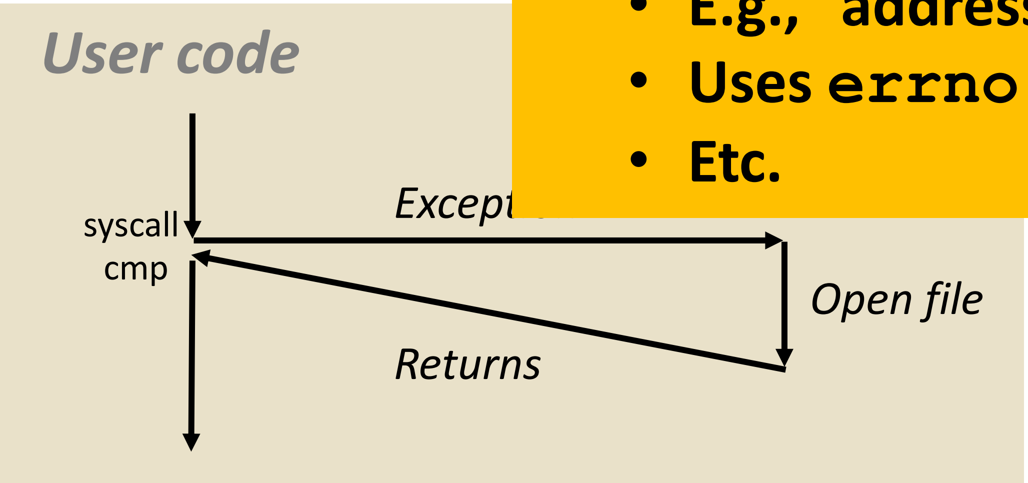
System Call I

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

```
00000000000e5d70 <__op
...
e5d79:   b8 02 00 00 00
e5d7e:   0f 05              sysca
e5d80:   48 3d 01 f0 ff ff   c
...
e5dfa:   c3                 retq
```

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.**



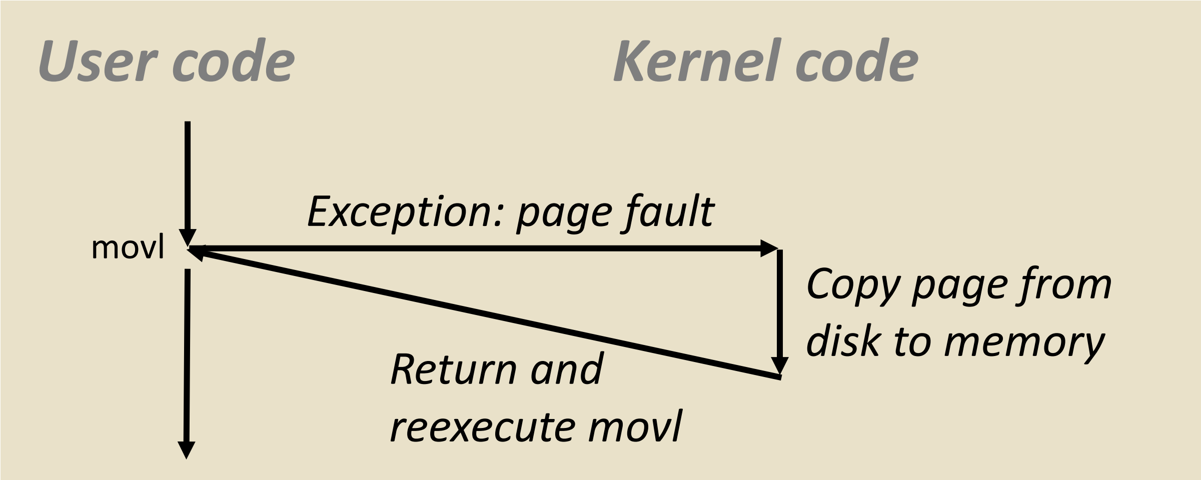
- Return value in `%rax`
- Negative value is an error corresponding to negative `errno`

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
```



User code

Kernel code

Exception: page fault

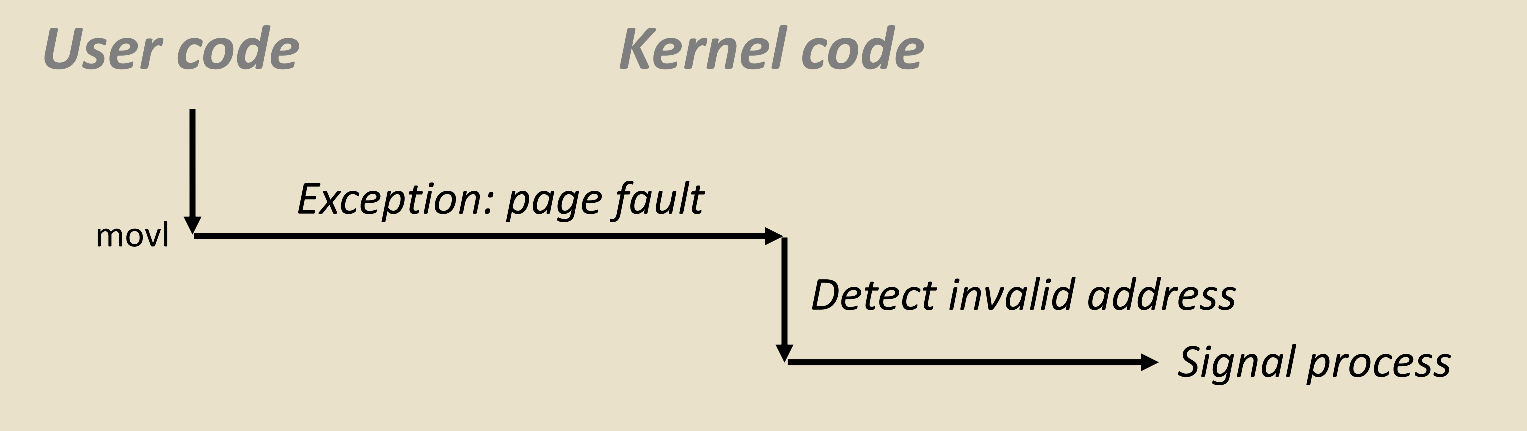
Copy page from disk to memory

Return and reexecute movl

Fault Example: Invalid Memory Reference

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

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



User code

Kernel code

movl

Exception: page fault

Detect invalid address

Signal process

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

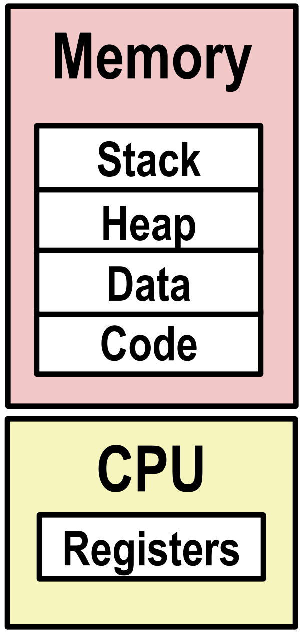
Today

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

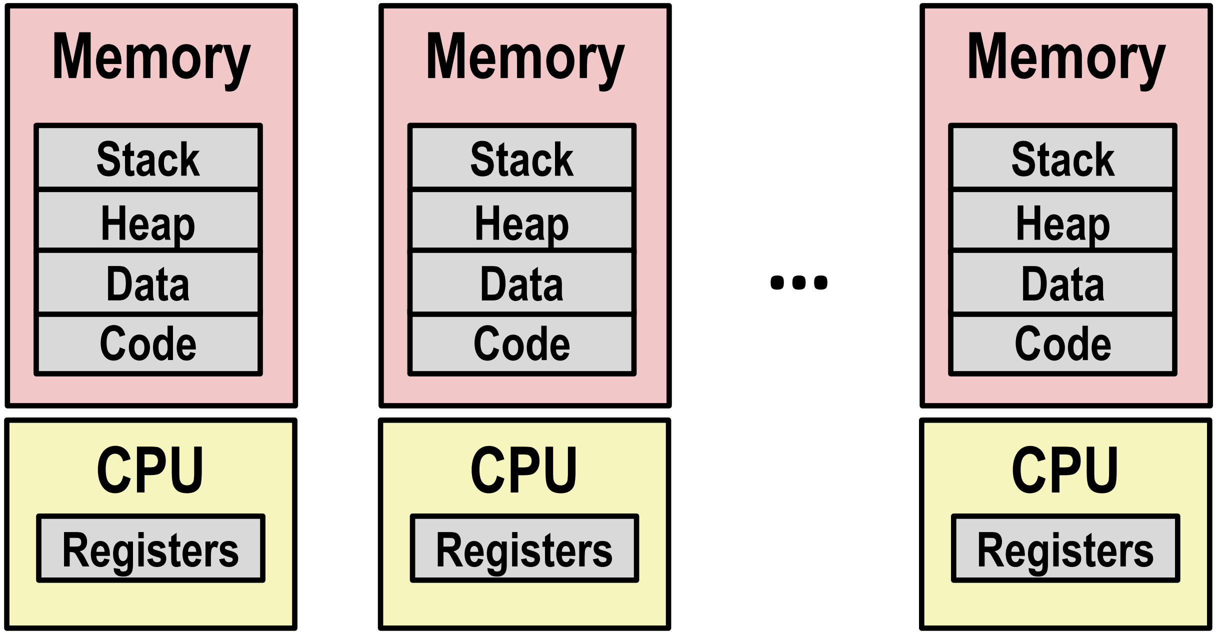
Processes

- **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 kernel mechanism called *virtual memory*

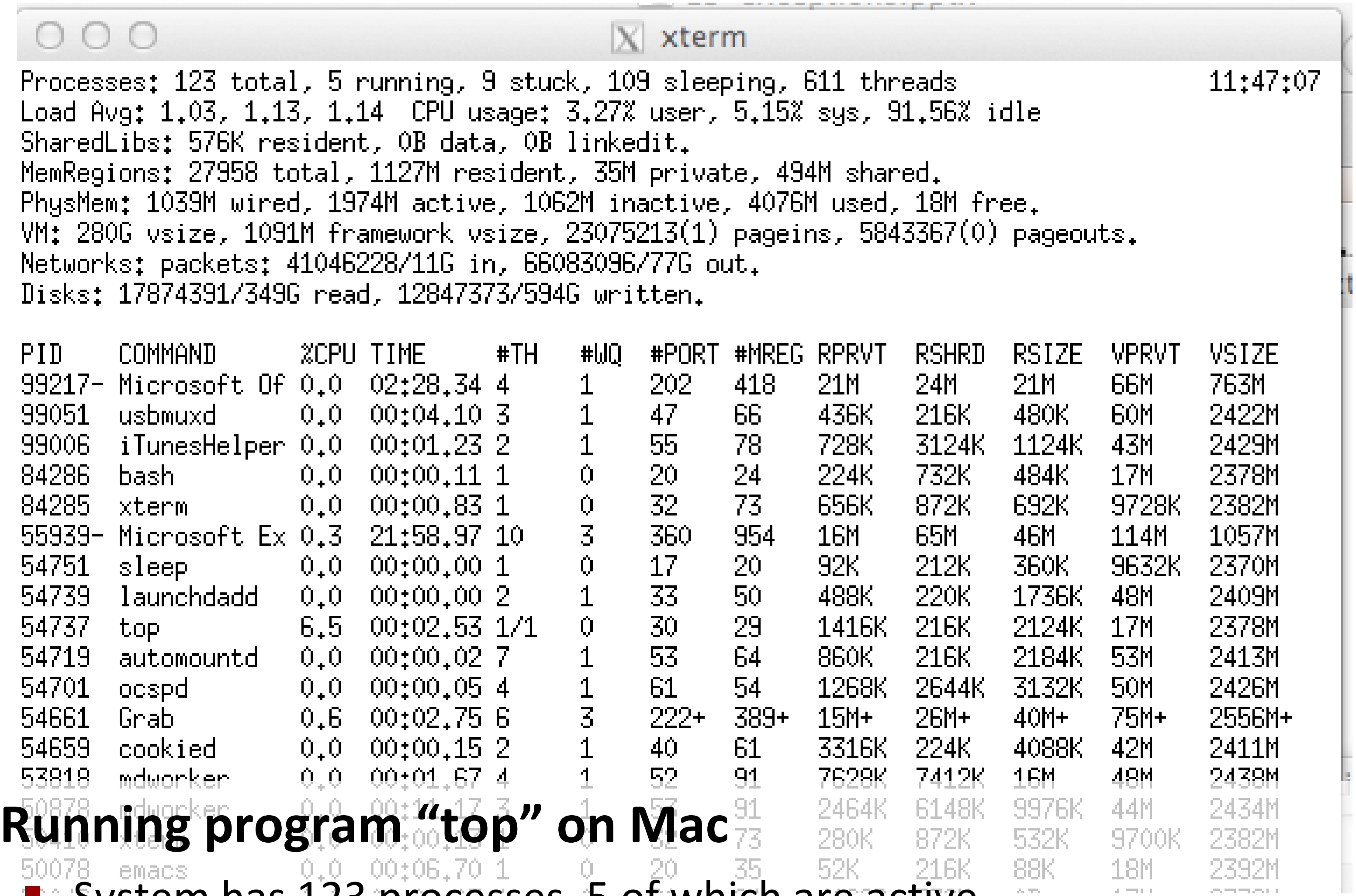


Multiprocessing: The Illusion


Memory
Stack
Heap
Data
Code
CPU
Registers
Memory
Stack
Heap
Data
Code
CPU
Registers
…
Memory
Stack
Heap
Data
Code
CPU
Registers

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

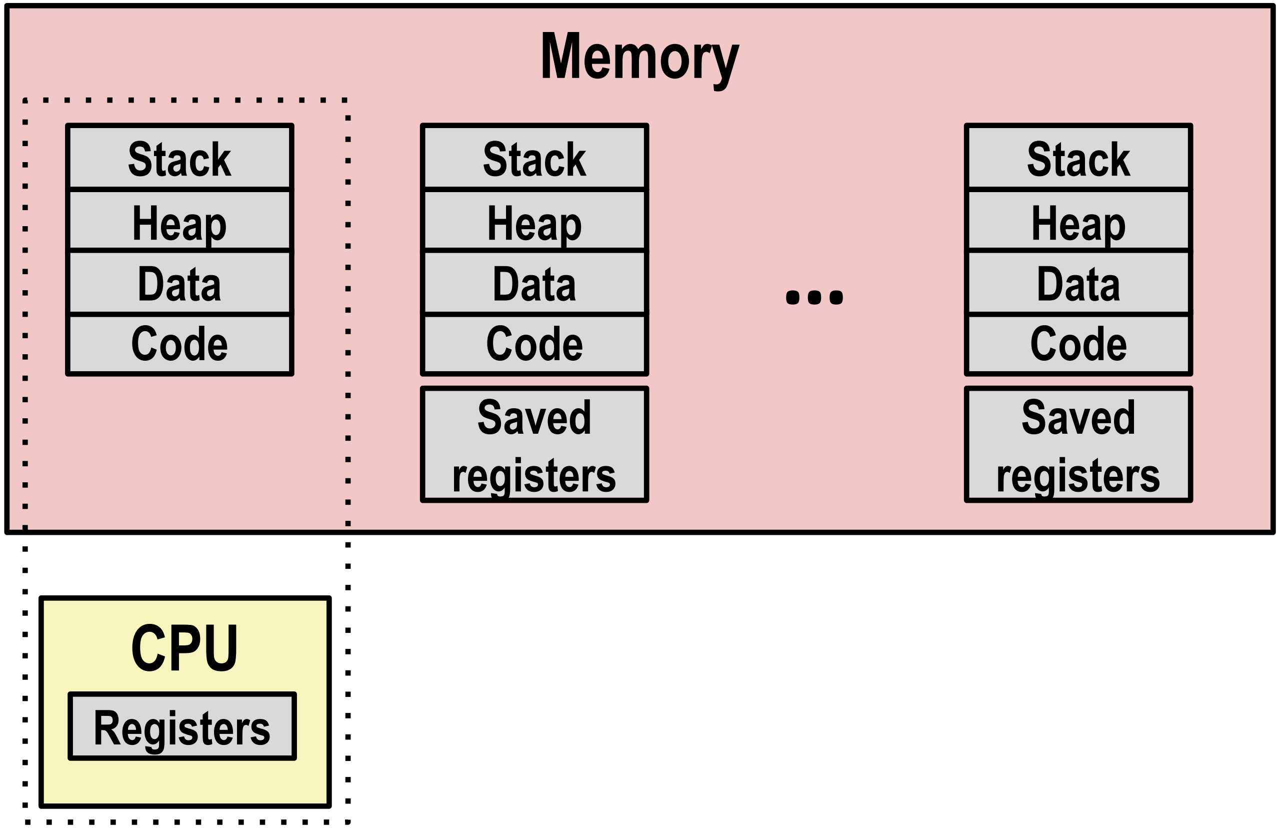
Multiprocessing Example



```
Processes: 123 total, 5 running, 9 stuck, 109 sleeping, 611 threads          11:47:07
Load Avg: 1.03, 1.13, 1.14  CPU usage: 3.27% user, 5.15% sys, 91.56% idle
SharedLibs: 576K resident, 0B data, 0B linkedit.
MemRegions: 27958 total, 1127M resident, 35M private, 494M shared.
PhysMem: 1039M wired, 1974M active, 1062M inactive, 4076M used, 18M free.
VM: 280G vsize, 1091M framework vsize, 23075213(1) pageins, 5843367(0) pageouts.
Networks: packets: 41046228/11G in, 66083096/77G out.
Disks: 17874391/349G read, 12847373/594G written.

PID    COMMAND      %CPU TIME     #TH   #WQ  #PORT #MREG RPRVT  RSHRD  RSIZE  VPRVT  VSIZE
99217- Microsoft Of 0.0  02:28.34 4     1    202   418   21M    24M    21M    66M    763M
99051  usbmuxd      0.0  00:04.10 3     1    47    66    436K   216K   480K   60M    2422M
99006  iTunesHelper 0.0  00:01.23 2     1    55    78    728K   3124K  1124K  43M    2429M
84286  bash         0.0  00:00.11 1     0    20    24    224K   732K   484K   17M    2378M
84285  xterm        0.0  00:00.83 1     0    32    73    656K   872K   692K   9728K  2382M
55939- Microsoft Ex 0.3  21:58.97 10    3    360   954   16M    65M    46M    114M   1057M
54751  sleep        0.0  00:00.00 1     0    17    20    92K    212K   360K   9632K  2370M
54739  launchdadd   0.0  00:00.00 2     1    33    50    488K   220K   1736K  48M    2409M
54737  top          6.5  00:02.53 1/1   0    30    29    1416K  216K   2124K  17M    2378M
54719  automountd   0.0  00:00.02 7     1    53    64    860K   216K   2184K  53M    2413M
54701  ocspd        0.0  00:00.05 4     1    61    54    1268K  2644K  3132K  50M    2426M
54661  Grab         0.6  00:02.75 6     3    222+  389+  15M+   26M+   40M+   75M+   2556M+
54659  cookied      0.0  00:00.15 2     1    40    61    3316K  224K   4088K  42M    2411M
53818  mdworker     0.0  00:01.67 4     1    52    91    7628K  7412K  16M    48M    2438M
50878  mdworker     0.0  00:14.17 3     1    57    91    2464K  6148K  9976K  44M    2434M
50410  xterm        0.0  [illegible] 1  0    32    73    280K   872K   532K   9700K  2382M
50078  emacs        0.0  00:06.70 1     0    20    35    52K    216K   88K    18M    2392M
```

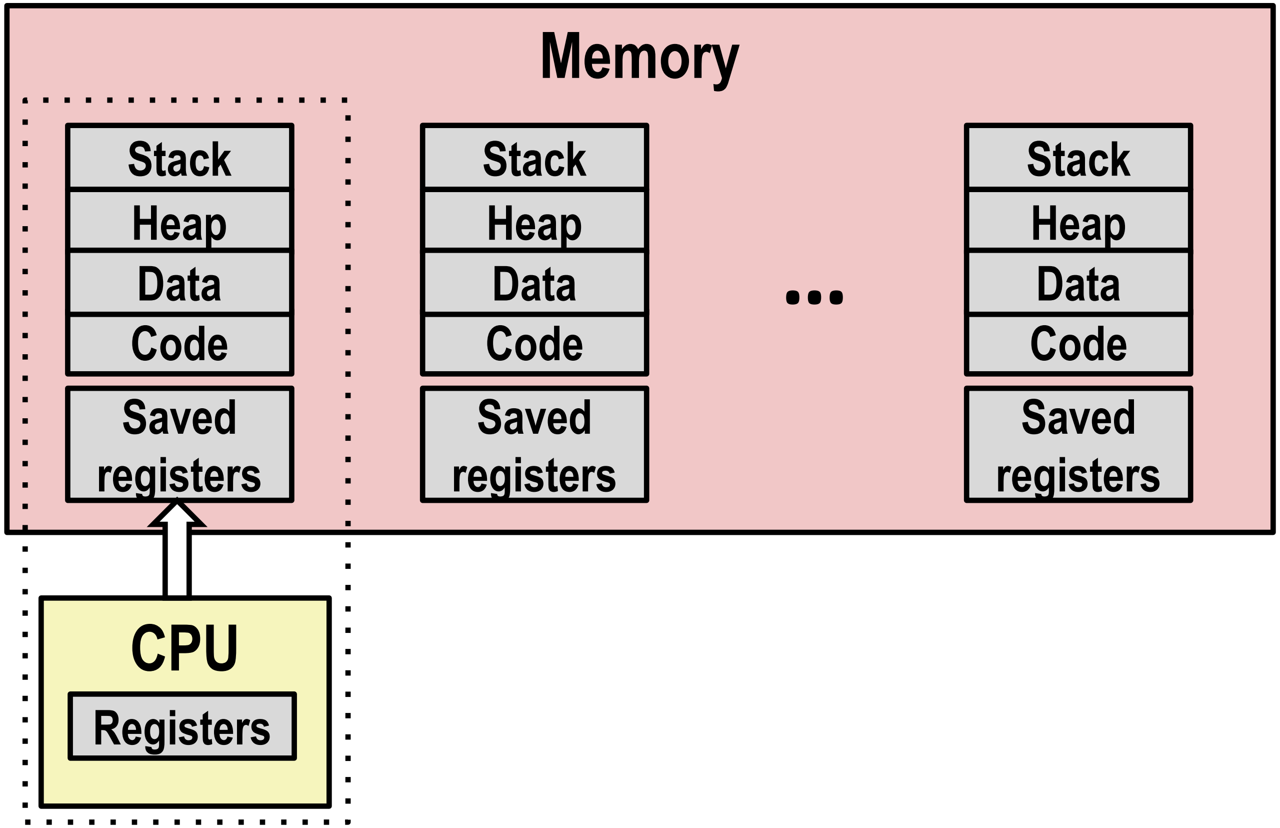
- **Running program “top” on Mac**
  - System has 123 processes, 5 of which are active
  - Identified by Process ID (PID)

Multiprocessing: The (Traditional) Reality


Memory
Stack
Heap
Data
Code
Stack
Heap
Data
Code
Saved registers
…
Stack
Heap
Data
Code
Saved registers
CPU
Registers

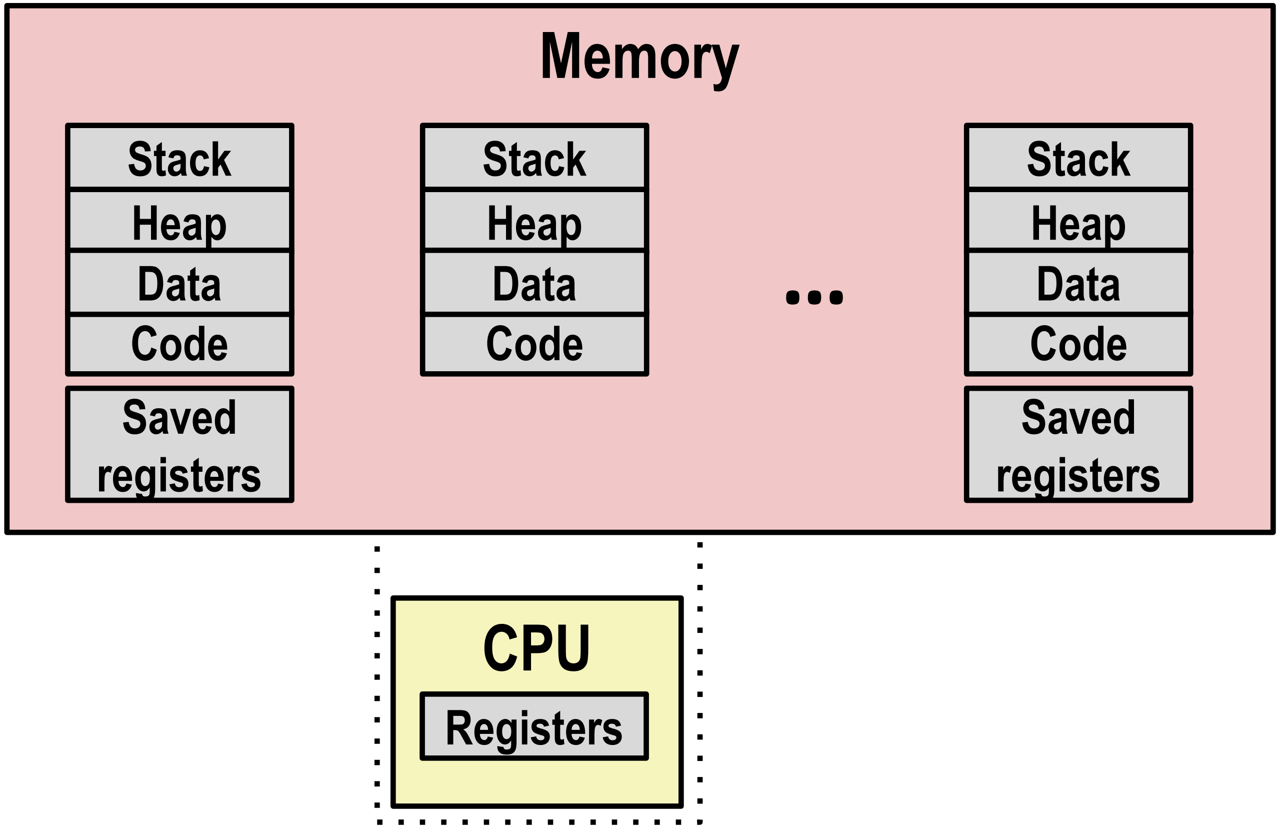
- **Single processor executes multiple processes concurrently**
  - Process executions interleaved (multitasking)
  - Address spaces managed by virtual memory system (like last week)
  - Register values for nonexecuting processes saved in memory

Multiprocessing: The (Traditional) Reality


Memory
Stack
Heap
Data
Code
Saved registers
Stack
Heap
Data
Code
Saved registers
…
Stack
Heap
Data
Code
Saved registers
CPU
Registers

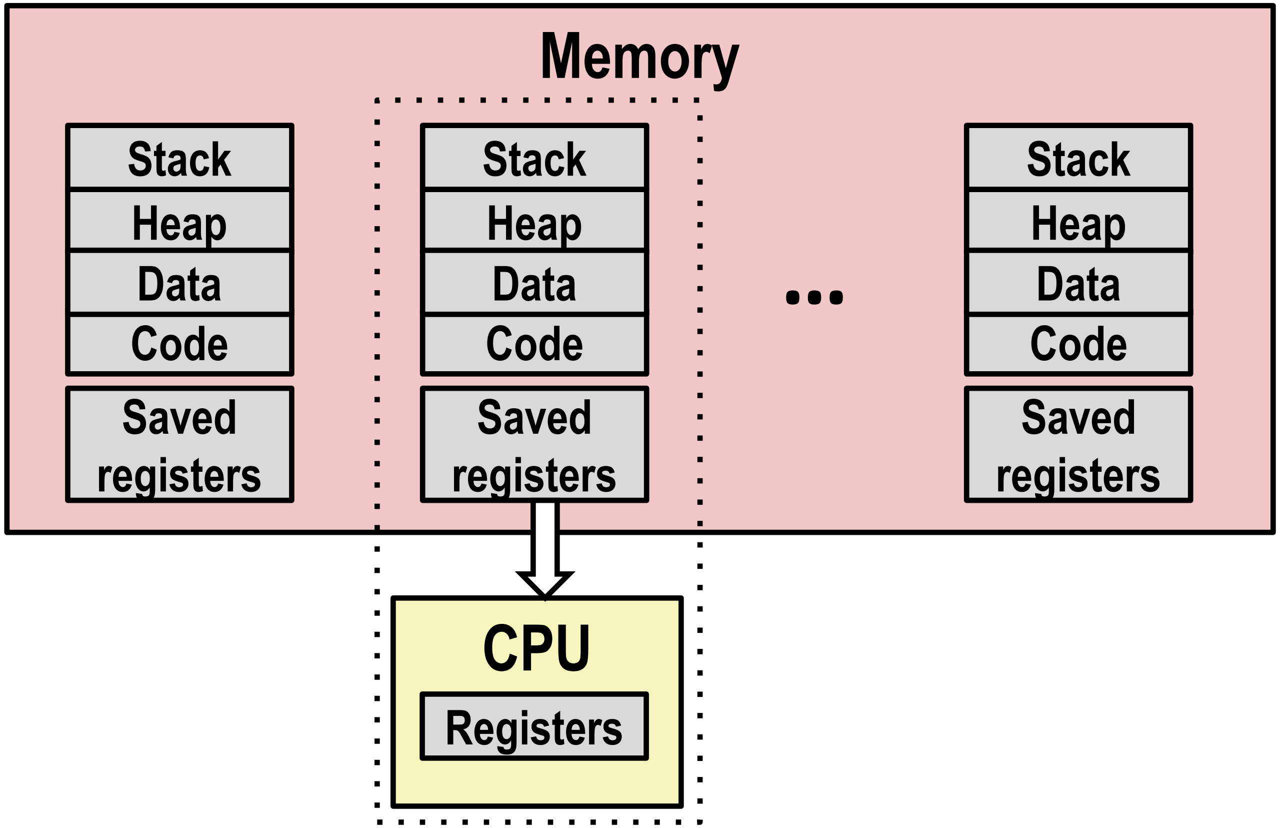
- Save current registers in memory

Multiprocessing: The (Traditional) Reality


Memory
Stack
Heap
Data
Code
Saved registers
Stack
Heap
Data
Code
…
Stack
Heap
Data
Code
Saved registers
CPU
Registers

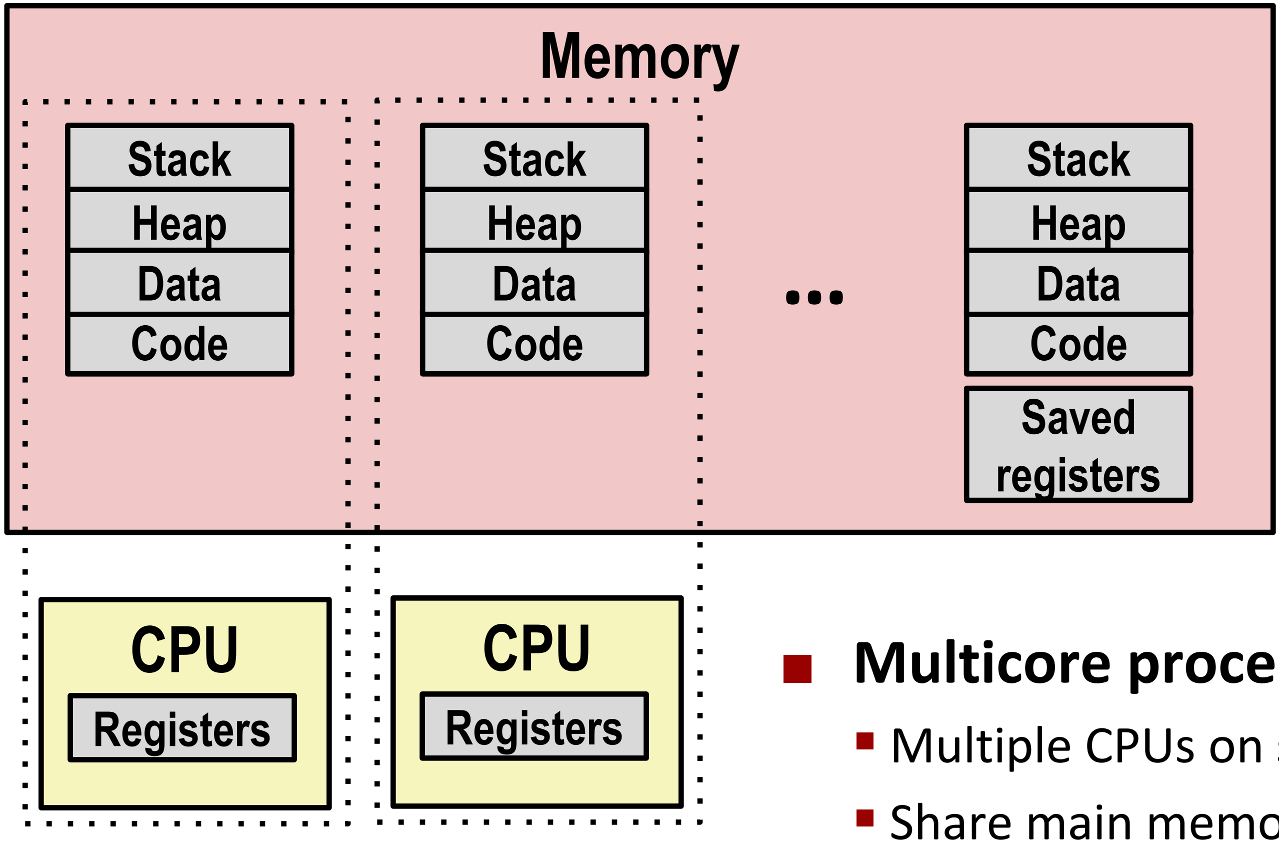
- Schedule next process for execution

Multiprocessing: The (Traditional) Reality


Memory
Stack
Heap
Data
Code
Saved registers
Stack
Heap
Data
Code
Saved registers
…
Stack
Heap
Data
Code
Saved registers
CPU
Registers

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

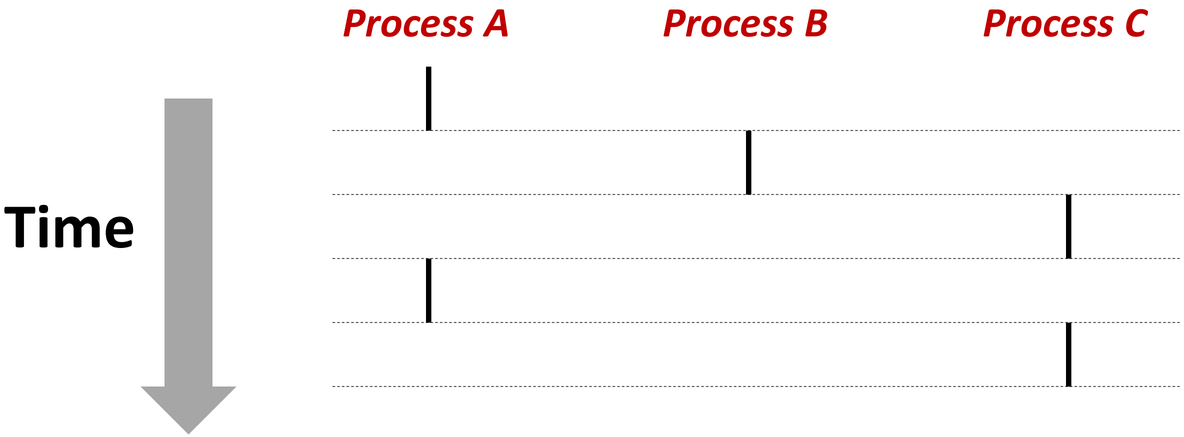
Multiprocessing: The (Modern) Reality


Memory
Stack
Heap
Data
Code
Stack
Heap
Data
Code
…
Stack
Heap
Data
Code
Saved registers
CPU
Registers
CPU
Registers

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

Concurrent Processes

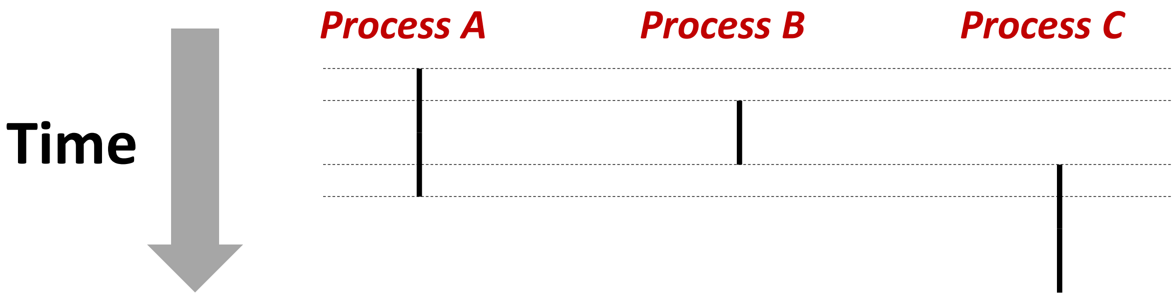
- **Each process is a logical control flow.**
- **Two processes *run concurrently* (*are concurrent)* if their flows overlap in time**
- **Otherwise, they are *sequential***
- **Examples (running on single core):**
  - Concurrent: A & B, A & C
  - Sequential: B & C



User View of Concurrent Processes

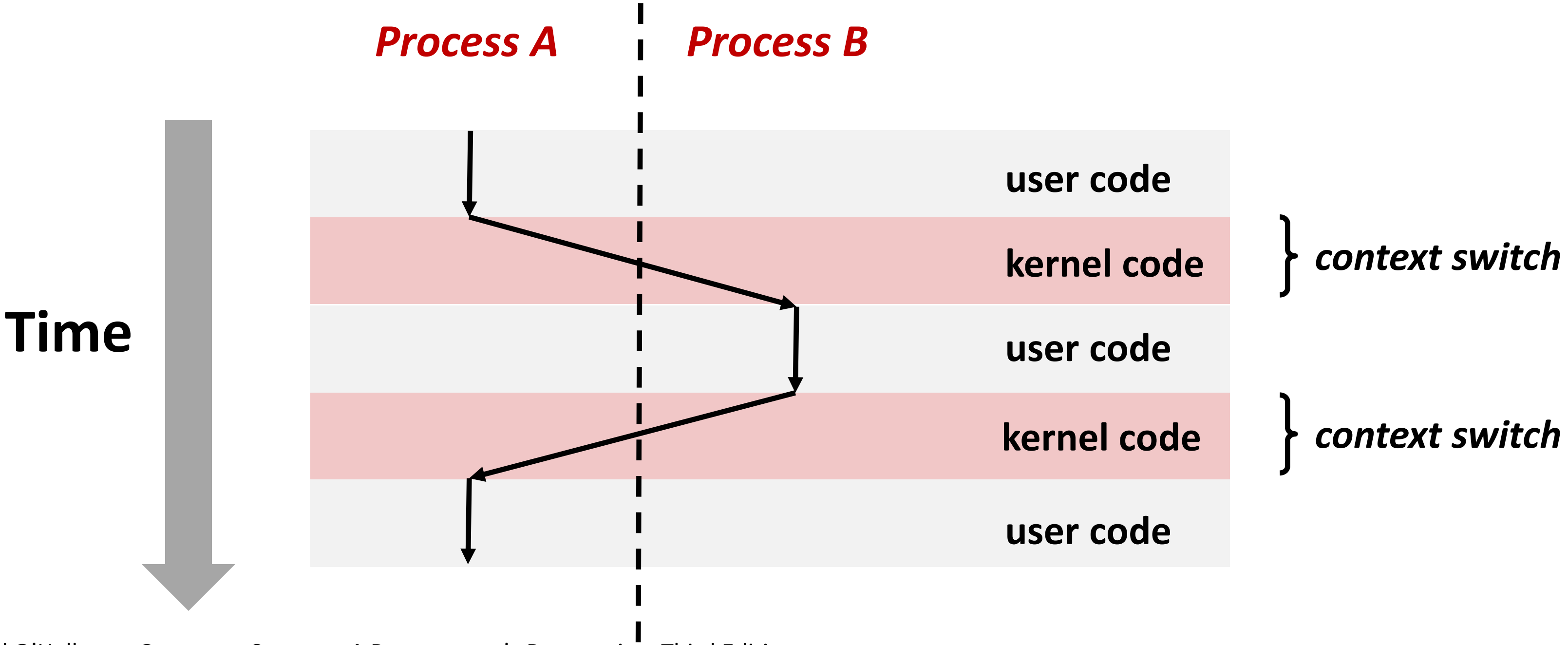
- Control flows for concurrent processes are physically disjoint in time

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


Process A
Process B
Process C
Time

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***


Process A
Process B
Time
user code
kernel code
user code
kernel code
user code
context switch
context switch

Today

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

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");
```

Note: csapp.c exits with 0.

- But, must think about application. Not alway appropriate to exit when something goes wrong.

Error-handling Wrappers

- We simplify the code we present to you even further by using Stevens[1]-style error-handling wrappers:

```
pid_t Fork(void)
{
    pid_t pid;

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

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

- NOT what you generally want to do in a real application

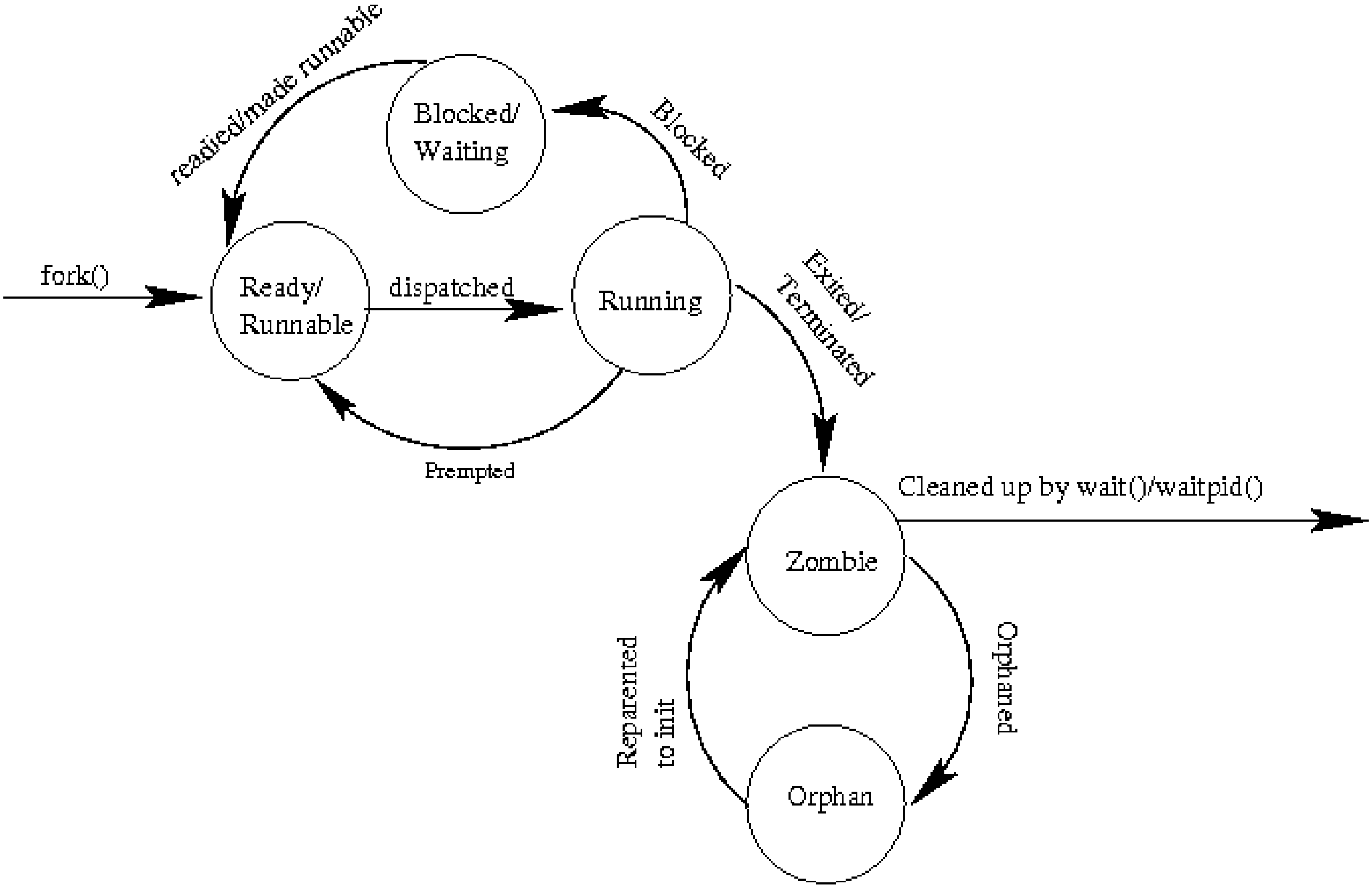
[1] e.g., in “UNIX Network Programming: The sockets networking API“ W. Richard Stevens

Obtaining Process IDs

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

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

Process Lifecycle


readied/made runnable
Blocked/
Waiting
Blocked
fork()
Ready/
Runnable
dispatched
Running
Exited/
Terminated
Prempted
Cleaned up by wait()/waitpid()
Zombie
Reparented
to init
Orphaned
Orphan

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:**
  - 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

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

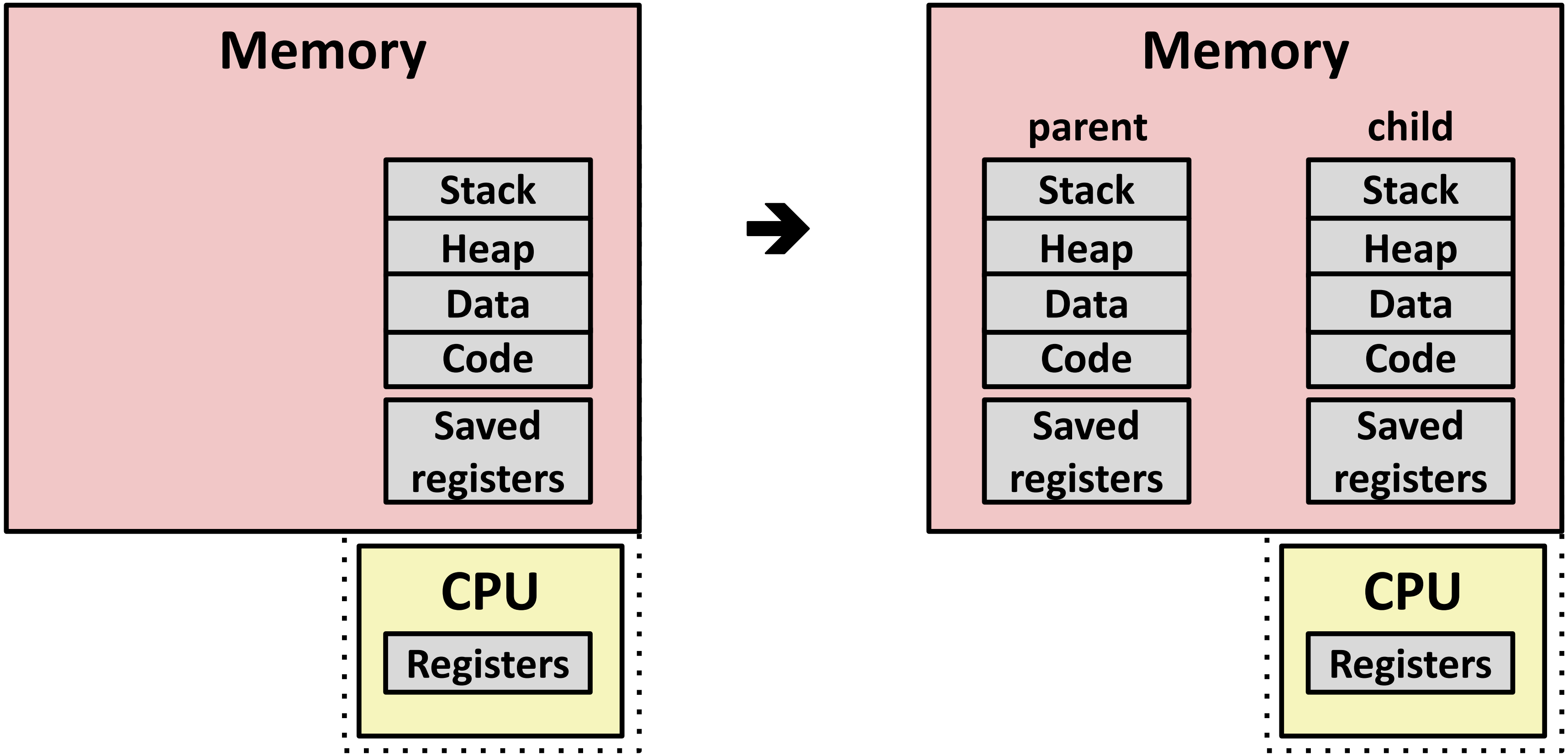
Creating Processes

- ***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***

Conceptual View of `fork`


Memory
Stack
Heap
Data
Code
Saved registers
CPU
Registers
➔
Memory
parent
child
Stack
Heap
Data
Code
Saved registers
Stack
Heap
Data
Code
Saved registers
CPU
Registers

- **Make complete copy of execution state**
  - Designate one as parent and one as child
  - Resume execution of parent or child

The `fork` Function Revisited

- **VM and memory mapping explain how `fork` provides private address space for each process.**
- **To create virtual address for 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.**

`fork` Example

```c
int main(int argc, char** argv)
{
    pid_t pid;
    int x = 1;

    pid = Fork();
    if (pid == 0) {   /* Child */
        printf("child : x=%d\n", ++x);
        return 0;
    }

    /* Parent */
    printf("parent: x=%d\n", --x);
    return 0;
}
```

fork.c

- Call once, return twice
- Concurrent execution
  - Can't predict execution order of parent and child

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

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

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

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

`fork` Example

```
int main(int argc, char** argv)
{
    pid_t pid;
    int x = 1;

    pid = Fork();
    if (pid == 0) {  /* Child */
        printf("child : x=%d\n", ++x);
        return 0;
    }

    /* Parent */
    printf("parent: x=%d\n", --x);
    return 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
- Shared open files
    - `stdout` is the same in both parent and child

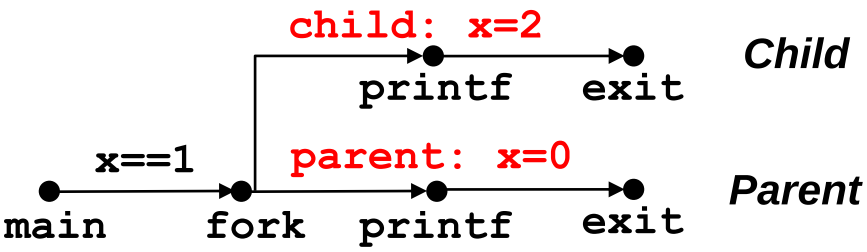
Modeling `fork` with Process Graphs

- **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

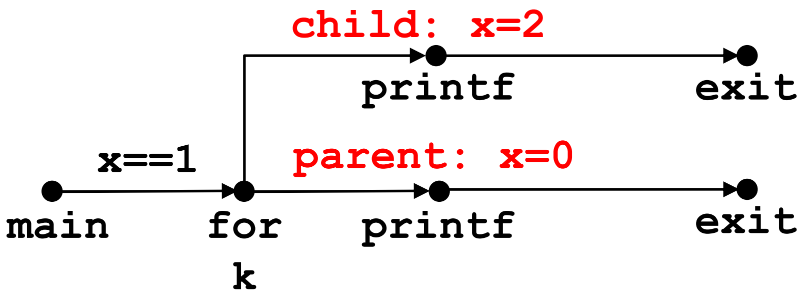
```
int main(int argc, char** argv)
{
    pid_t pid;
    int x = 1;

    pid = Fork();
    if (pid == 0) {  /* Child */
        printf("child : x=%d\n", ++x);
        return 0;
    }

    /* Parent */
    printf("parent: x=%d\n", --x);
    return 0;
}
```

fork.c

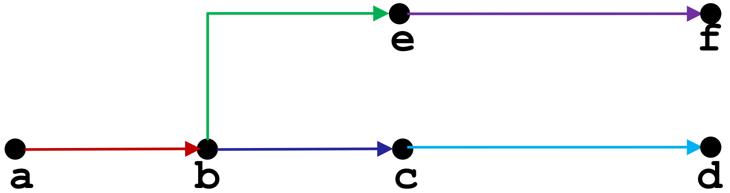

child: x=2
Child
printf
exit
x==1
parent: x=0
Parent
main
fork
printf
exit

Interpreting Process Graphs

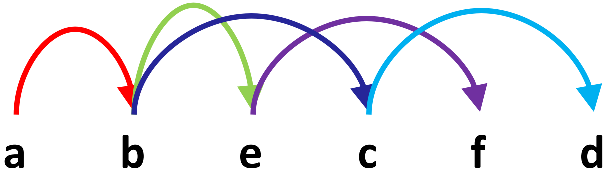
- Original graph:


child: x=2
printf
exit
x==1
parent: x=0
main
for
k
printf
exit

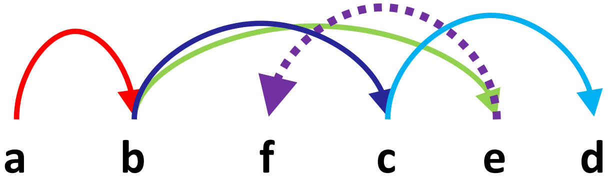
- Relabled graph:



Feasible total ordering:



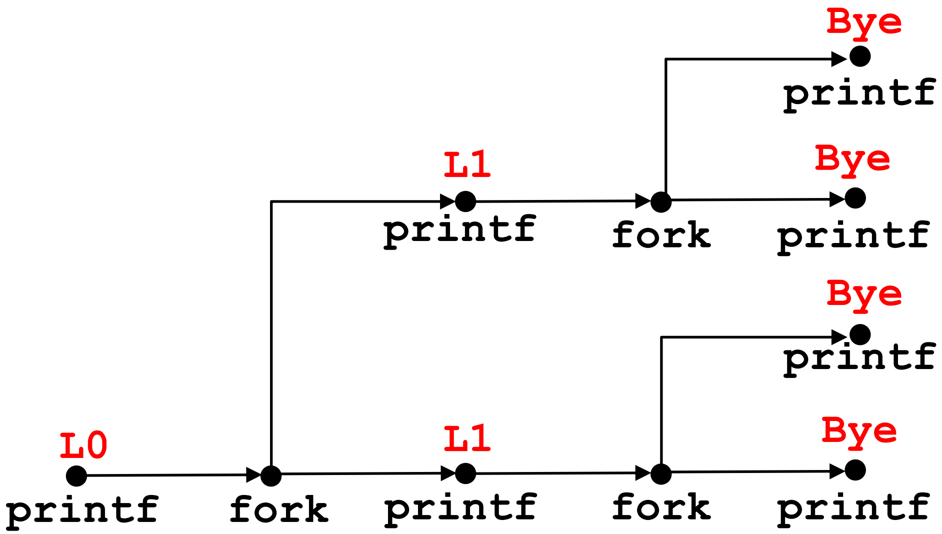
Feasible or Infeasible?



Infeasible: not a topological sort

`fork` Example: Two consecutive `fork`s

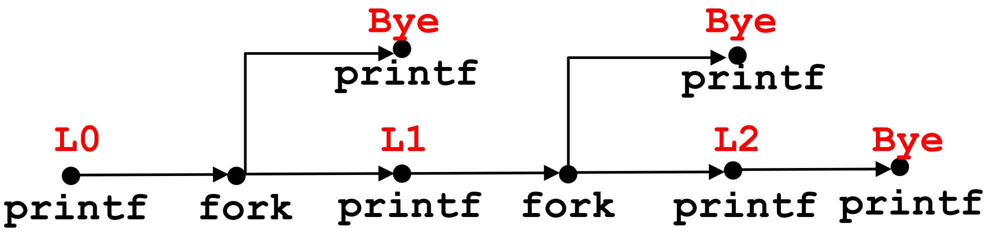
```
void fork2()
{
    printf("L0\n");
    fork();
    printf("L1\n");
    fork();
    printf("Bye\n");
}                        forks.c
```


Bye
printf
L1
Bye
printf
fork
printf
Bye
printf
L0
L1
Bye
printf
fork
printf
fork
printf

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

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

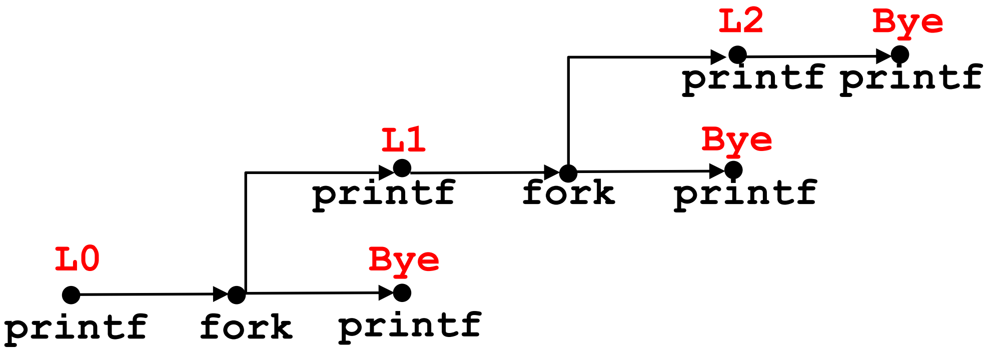
```
void fork4()
{
    printf("L0\n");
    if (fork() != 0) {
        printf("L1\n");
        if (fork() != 0) {
            printf("L2\n");
        }
    }
    printf("Bye\n");
}                                 forks.c
```


Bye
printf
Bye
printf
L0
L1
L2
Bye
printf
fork
printf
fork
printf
printf

Feasible or Infeasible?	Feasible or Infeasible?
L0	L0
Bye	L1
L1	Bye
Bye	Bye
Bye	L2
L2	Bye
Infeasible	Feasible

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

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



L2
Bye
printf
printf
L1
Bye
printf
fork
printf
L0
Bye
printf
fork
printf

Feasible or Infeasible?	Feasible or Infeasible?
L0	L0
Bye	Bye
L1	L1
Bye	L2
Bye	Bye
L2	Bye
Infeasible	Feasible

Quiz Time!

Day 17 – ECF / Exceptions and Processes

Reaping Child Processes

- **Idea**
  - When process terminates, it still consumes system resources
    - Examples: Exit status, various OS tables
  - Called a “zombie”
    - Living corpse, half alive and half dead
- **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 should be reaped by **`init`** process (pid == 1)
    - Unless ppid == 1! Then need to reboot…
  - So, only need explicit reaping in long-running processes
    - e.g., shells and servers

Zombie Example

```c
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

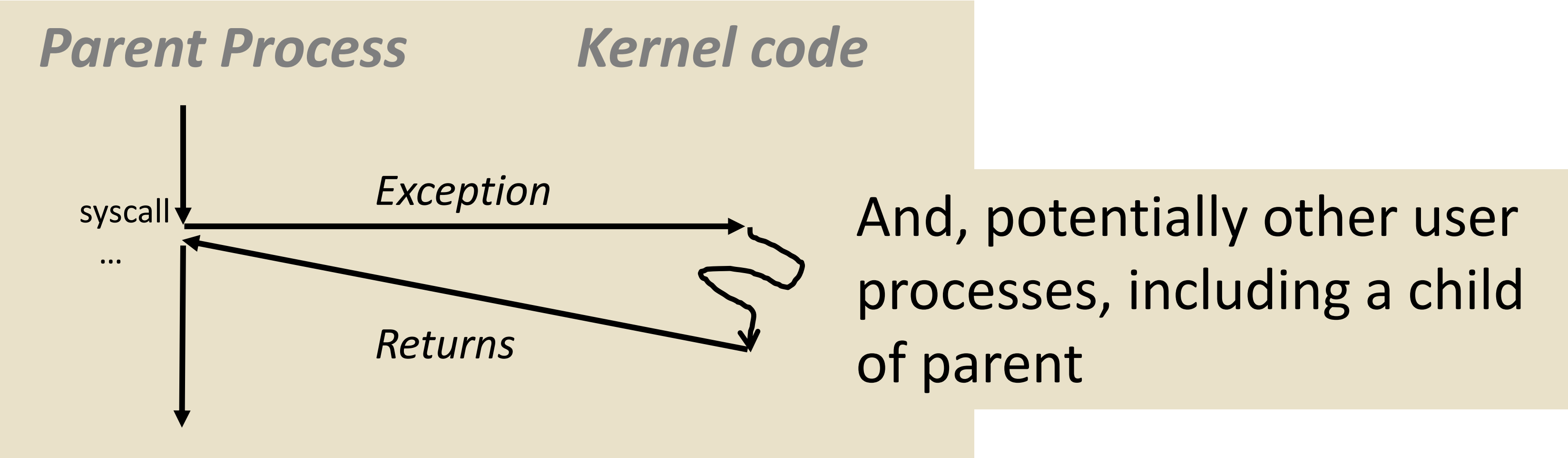
```
void fork8()
{
    if (fork() == 0) {
        /* Child */
        printf("Running Child, PID = %d\n",
               getpid());
        while (1)
            ; /* Infinite loop */
    } else {
        printf("Terminating Parent, PID = %d\n",
               getpid());
        exit(0);
    }
}
```

```
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

`wait`: Synchronizing with Children

- **Parent reaps a child by calling the `wait` function**
- **`int wait(int *child_status)`**
  - Suspends current process until one of its children terminates
  - Implemented as syscall


Parent Process
Kernel code
syscall
…
Exception
Returns

And, potentially other user processes, including a child of parent

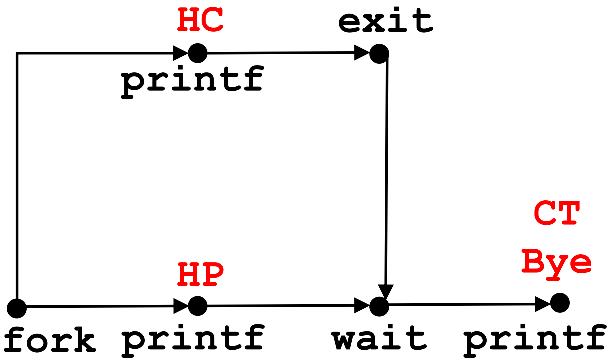
`wait`: Synchronizing with Children

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

- **`int wait(int *child_status)`**
  - 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

`wait`: Synchronizing with Children

```
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
```


HC
exit
printf
CT
Bye
HP
fork
printf
wait
printf

Feasible output(s):

HC	HP
HP	HC
CT	CT
Bye	Bye

Infeasible output:

HP
CT
Bye
HC

Another `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, 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

`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 (see textbook)

```
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);
    }
}
```

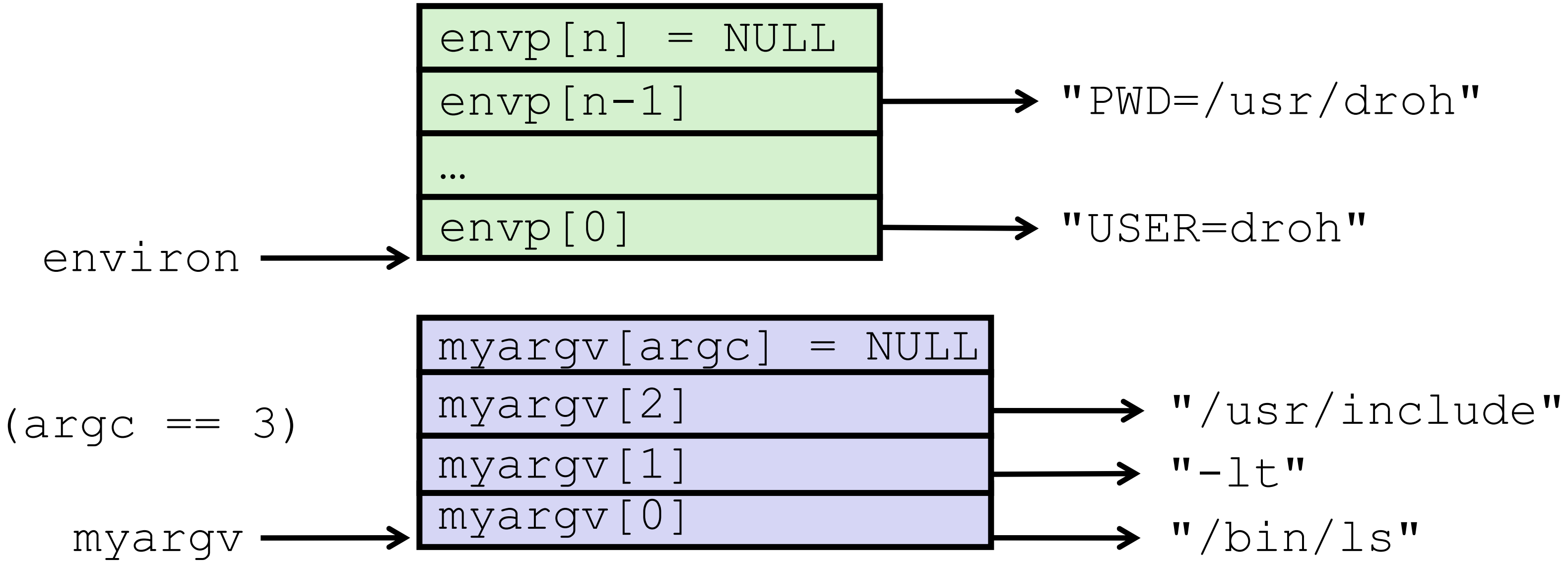
forks.c

`execve`: Loading and Running Programs

- **`int execve(char *filename, char *argv[], char *envp[])`**
- **Loads and runs in the current process:**
  - Executable file **`filename`**
    - Can be object file or script file beginning with `#!interpreter` (e.g., `#!/bin/bash`)
  - …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**
  - Retains PID, open files and signal context
- **Called once and never returns**
  - …except if there is an error

`execve` Example

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

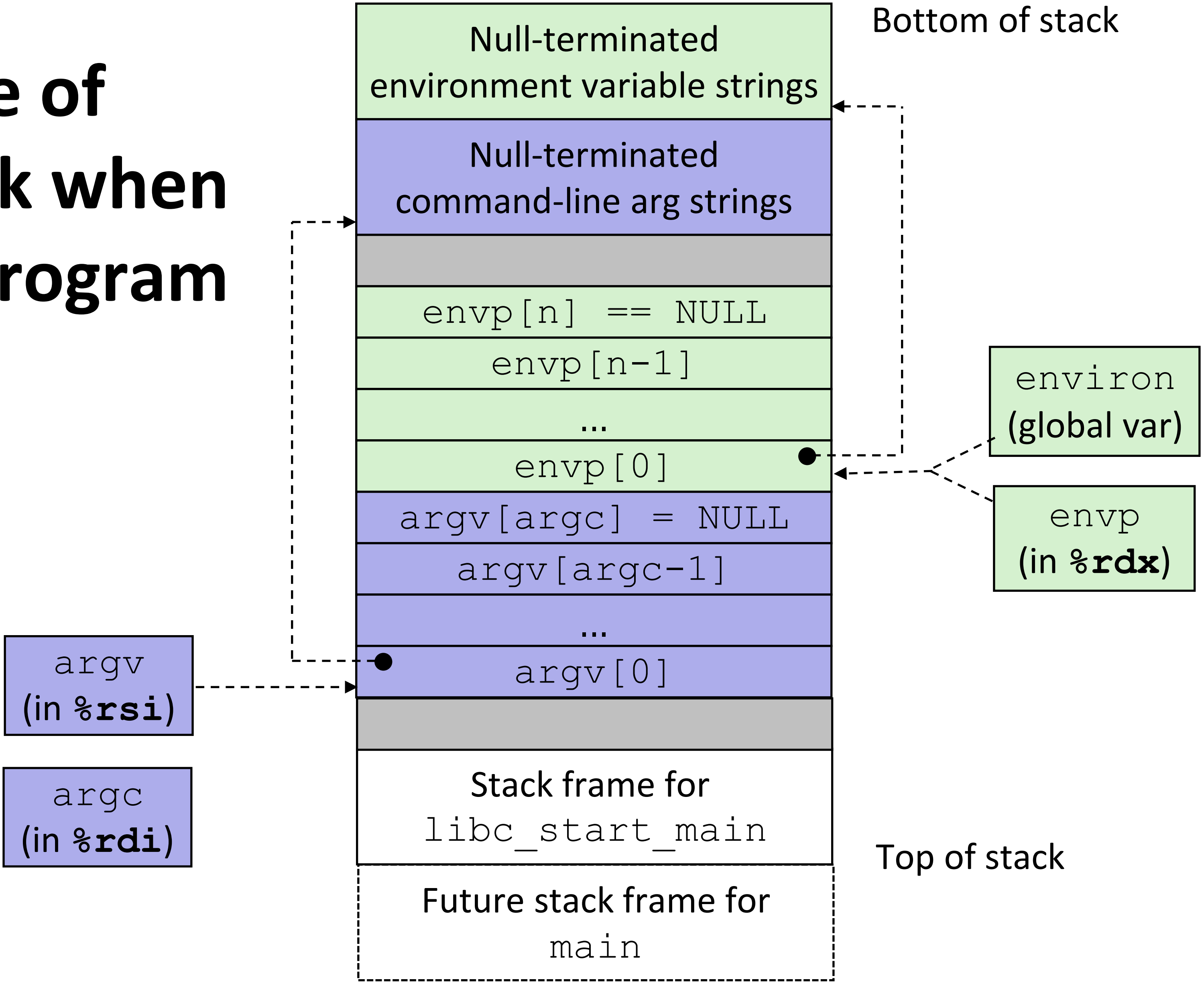


envp[n] = NULL
envp[n-1] → "PWD=/usr/droh"
…
envp[0] → "USER=droh"
environ → envp[0]

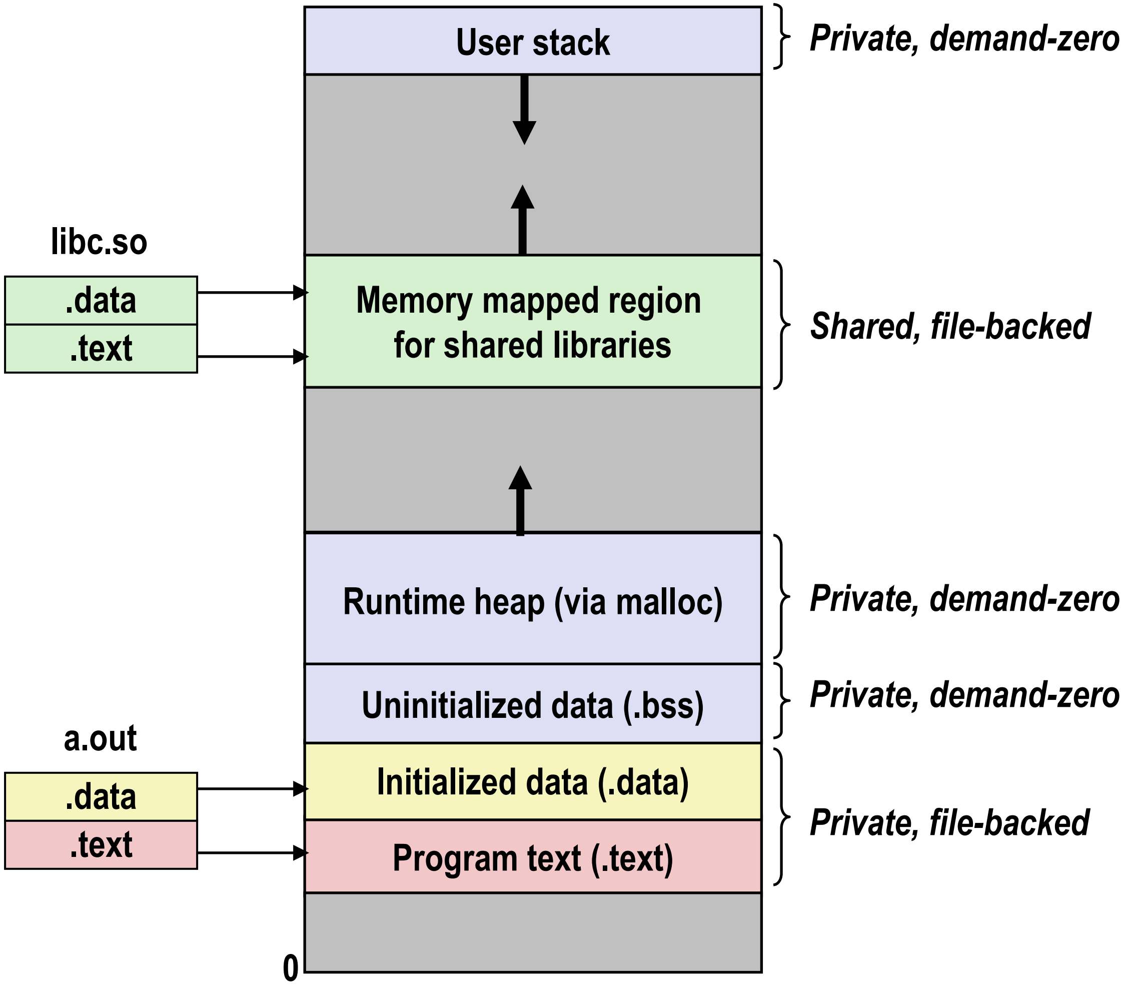
myargv[argc] = NULL
myargv[2] → "/usr/include"
myargv[1] → "-lt"
myargv[0] → "/bin/ls"
(argc == 3)
myargv → myargv[0]

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

Structure of the stack when a new program starts


Bottom of stack
Null-terminated environment variable strings
Null-terminated command-line arg strings
envp[n] == NULL
envp[n-1]
...
envp[0]
argv[argc] = NULL
argv[argc-1]
...
argv[0]
Stack frame for libc_start_main
Future stack frame for main
Top of stack
environ (global var)
envp (in %rdx)
argv (in %rsi)
argc (in %rdi)

The `execve` Function Revisited


User stack
Private, demand-zero
libc.so
.data
.text
Memory mapped region for shared libraries
Shared, file-backed
Runtime heap (via malloc)
Private, demand-zero
Uninitialized data (.bss)
Private, demand-zero
a.out
.data
.text
Initialized data (.data)
Program text (.text)
Private, file-backed
0

- **To load and run a new program `a.out` in the current process using `execve`:**
- **Free `vm_area_struct`'s and page tables for old areas**
- **Create `vm_area_struct`'s and page tables for new areas**
  - Programs and initialized data backed by object files.
  - `.bss` and stack backed by anonymous files.
- **Set PC to entry point in `.text`**
  - Linux will fault in code and data pages as needed.

Summary

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

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

Summary (cont.)

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

Making `fork` More Nondeterministic

- Problem
  - Linux scheduler does not create much run-to-run variance
  - Hides potential race conditions in nondeterministic programs
    - E.g., does `fork` return to child first, or to parent?
- Solution
  - Create custom version of library routine that inserts random delays along different branches
    - E.g., for parent and child in `fork`
  - Use runtime interpositioning to have program use special version of library code

Variable delay `fork`

```
/* fork wrapper function */
pid_t fork(void) {
    initialize();
    int parent_delay = choose_delay();
    int child_delay = choose_delay();
    pid_t parent_pid = getpid();
    pid_t child_pid_or_zero = real_fork();
    if (child_pid_or_zero > 0) {
        /* Parent */
        if (verbose) {
            printf(
"Fork.  Child pid=%d, delay = %dms.  Parent pid=%d, delay = %dms\n",
                   child_pid_or_zero, child_delay,
                   parent_pid, parent_delay);
            fflush(stdout);
        }
        ms_sleep(parent_delay);
    } else {
        /* Child */
        ms_sleep(child_delay);
    }
    return child_pid_or_zero;
}
```

myfork.c