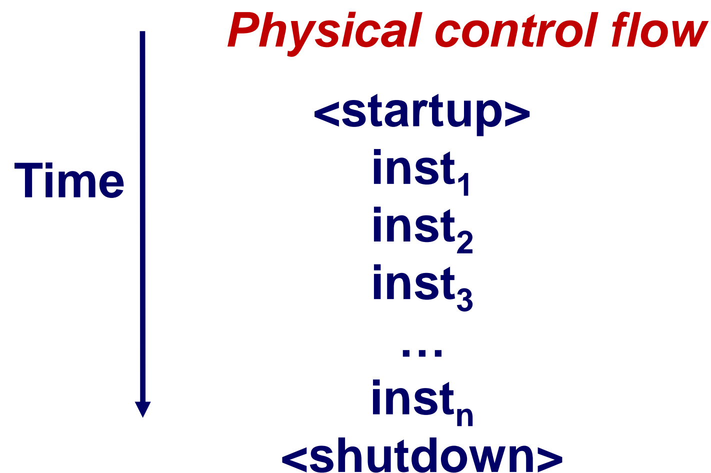
Exceptional Flow Control
Part I

Control Flow

Computers 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 system's physical control flow (or flow of control).



Altering the Control Flow

Up to Now: two mechanisms for changing control flow:

- Jumps/branches
- Call and return using the stack
- Both react to changes in internal program state.

Insufficient for a useful system

- Need CPU to react to changes in system state as well!
  - 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

Mechanisms exist at all levels of a computer system

- Change in control flow in response to a system event (i.e., change in system state)

Low level Mechanisms

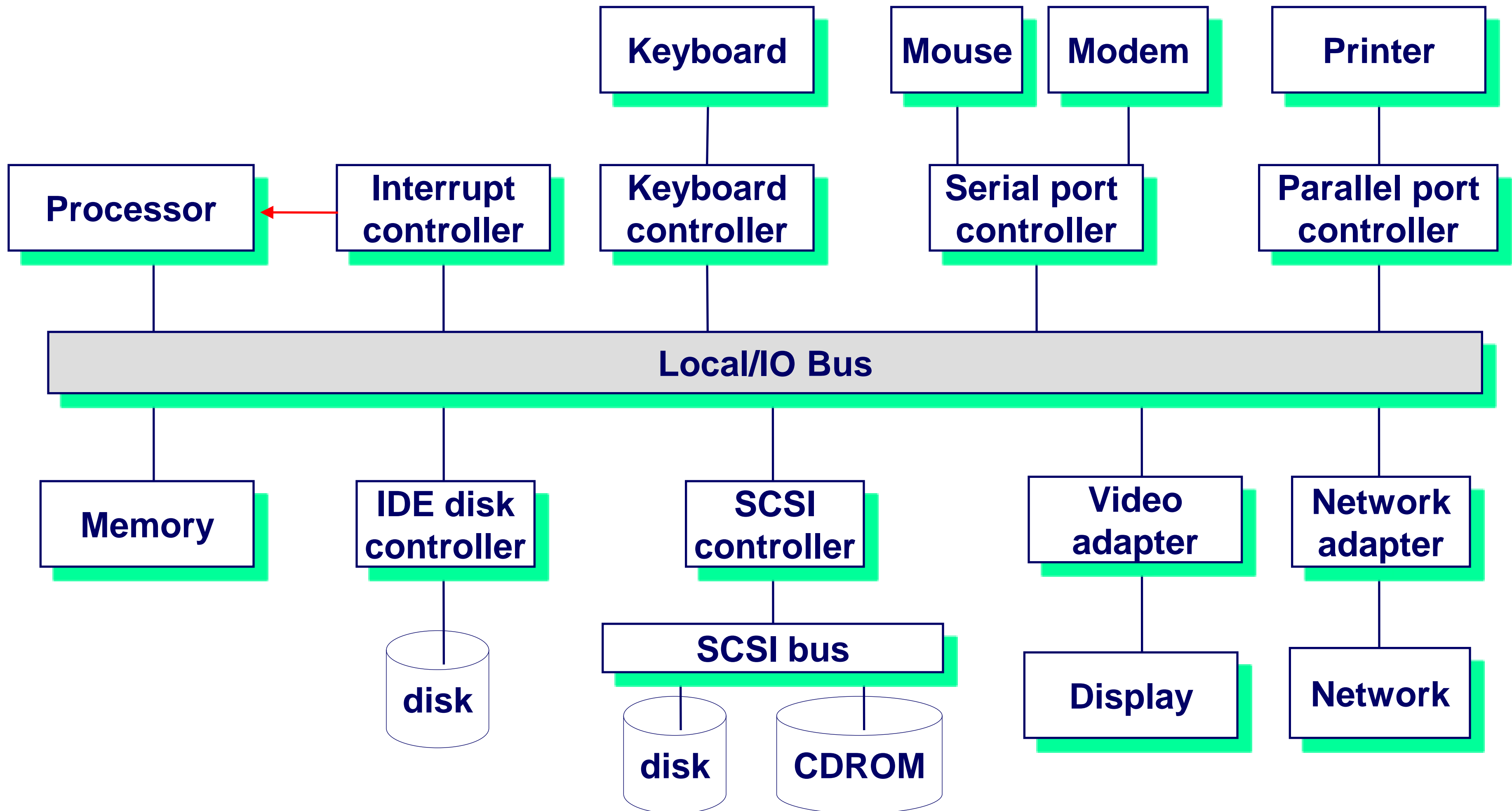
1. Exceptions and interrupts
   - Change in control flow in response to a system event (i.e., change in system state)
   - Implemented with a combination of hardware and OS software

Higher Level Mechanisms

2. Process context switch
   - Implemented via hardware timer and OS software
3. Signals
   - Implemented via OS
4. Nonlocal jumps (setjmp/longjmp)
   - Implemented via C language runtime library

Exceptions and interrupts

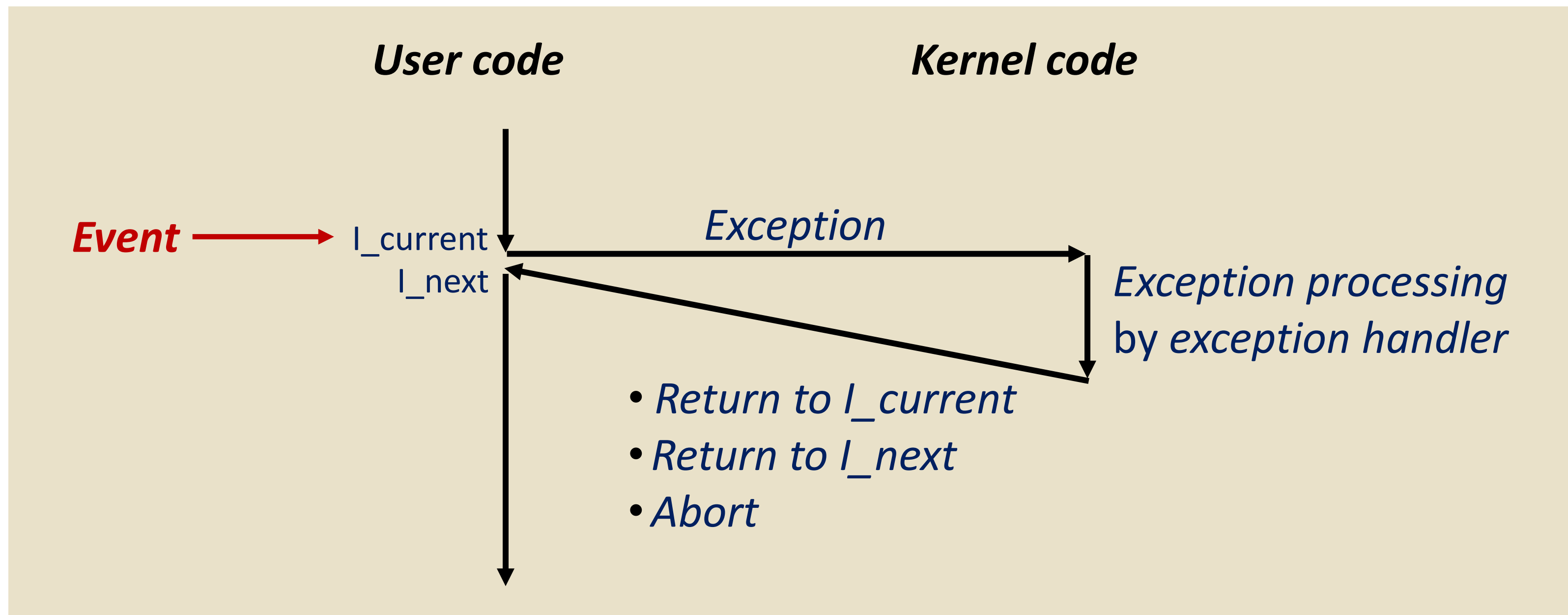
System context for exceptions



Exceptions and interrupts

Transfer of control to the OS in response to some event (i.e., change in processor state)

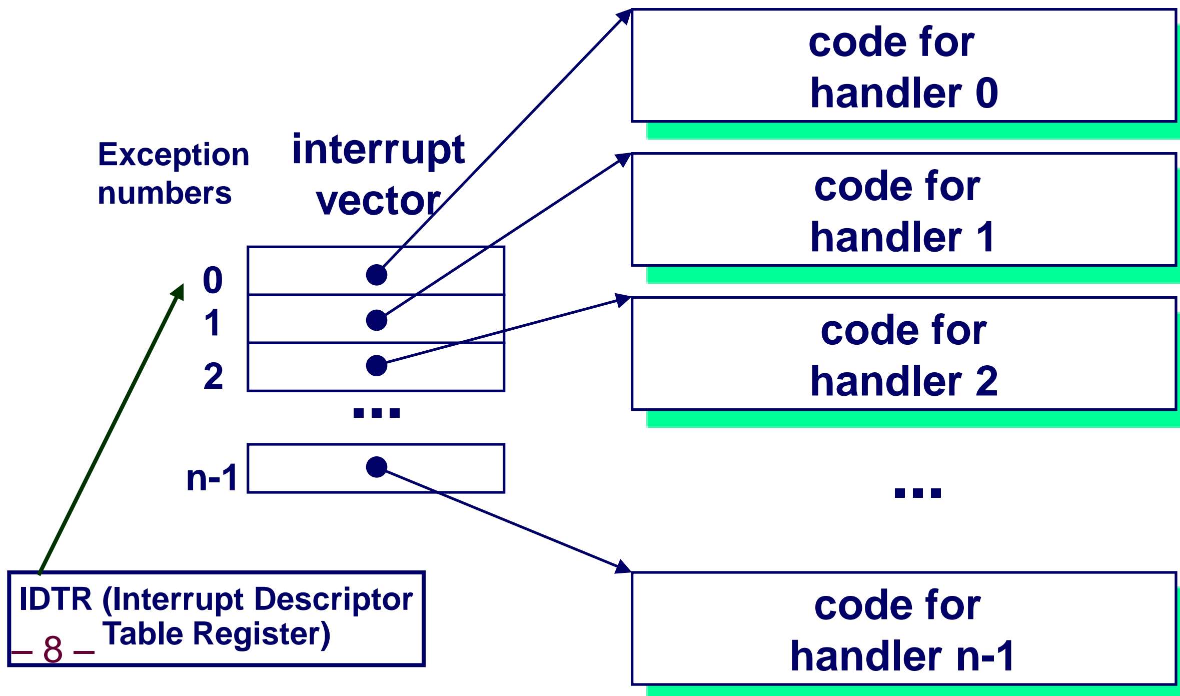
- Require mode change from user to kernel/supervisor
- Example events: Divide by 0, arithmetic overflow, page fault, I/O request completes, typing Ctrl-C



Interrupt Vectors

Many types of interrupts and exceptions

- Each type of event has a unique exception number k
- Index into jump table (a.k.a., interrupt vector table)
- Jump table entry k points to a function (exception handler).
- 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
- Causes a handler to run
- Handler returns to "next" instruction when finished

Examples:

- Timer interrupt
  - Every few ms, an external timer triggers an interrupt
  - Used by the kernel to take back control from user programs
- I/O interrupts
  - hitting Ctrl-c at the keyboard
  - arrival of a packet from a network
  - arrival of a data sector from a disk

Synchronous Exceptions

Caused by events that occur as a result of executing an instruction:

- **Traps**
  - Intentional
  - Examples: system calls, breakpoint traps, special instructions
  - Returns control to "next" instruction
- **Faults**
  - Unintentional but possibly recoverable
  - Examples: page faults (recoverable), protection faults (unrecoverable).
  - Either re-executes faulting ("current") instruction or aborts.
- **Aborts**
  - unintentional and unrecoverable
  - Examples: parity error, machine check.
  - Aborts current program

Examples of x86-64 Exceptions

Exception Number	*Description*	*Exception Class*
0	**Divide by zero**	**Fault**
13	**General protection fault**	**Fault**
14	**Page fault**	**Fault**
18	**Machine check**	**Abort**
128	**System call**	**Trap**
32-255	**OS-defined exceptions**	**Interrupt or trap**

System Call

Synchronous exception calls into OS

- Implemented via `int 0x80` or `syscall` instruction
- Each x86-64 system call has a unique ID number that is passed in `%rax`

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 trap example

Opening a File

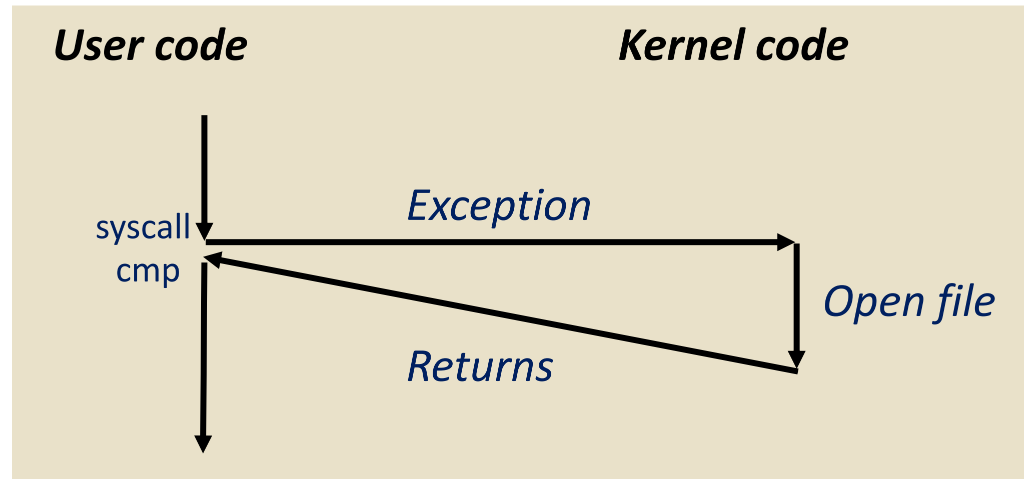
- User calls: `open(filename, options)`
- Calls `__open` function, which invokes system call instruction `syscall`
  - Function `__open` executes system call instruction

```
00000000000e5d70 <__open>:
...
e5d79:   b8 02 00 00 00       mov  $0x2,%eax   # open is syscall #2
e5d7e:   0f 05                syscall          # Return value in %rax
e5d80:   48 3d 01 f0 ff ff    cmp  $0xfffffffffffff001,%rax
...
e5dfa:   c3                   retq
```

  - `%rax` contains syscall number (arguments in `%rdi`, `%rsi`, `%rdx`, `%r10`, `%r8`, `%r9`)
  - Negative value is an error corresponding to negative `errno`
- OS must find or create file, get it ready for reading or writing
- Returns integer file descriptor

System Call Example: Opening File

```
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
```



Fault Example #1

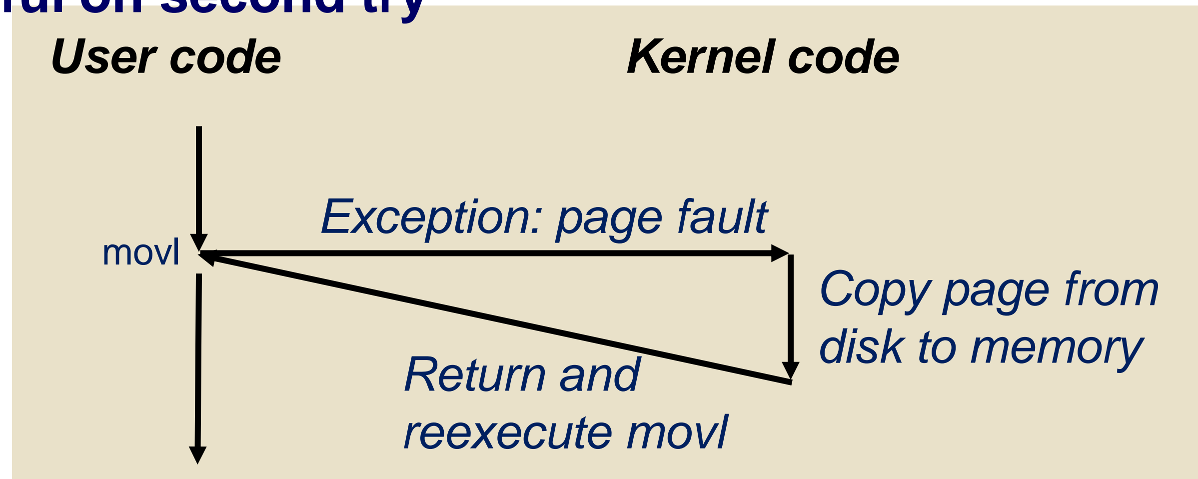
```
int a[1000];
main ()
{
    a[500] = 13;
}
```

Memory Reference

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

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

- OS page handler must load page into physical memory
- Returns to faulting instruction
- Successful on second try



Fault Example #2

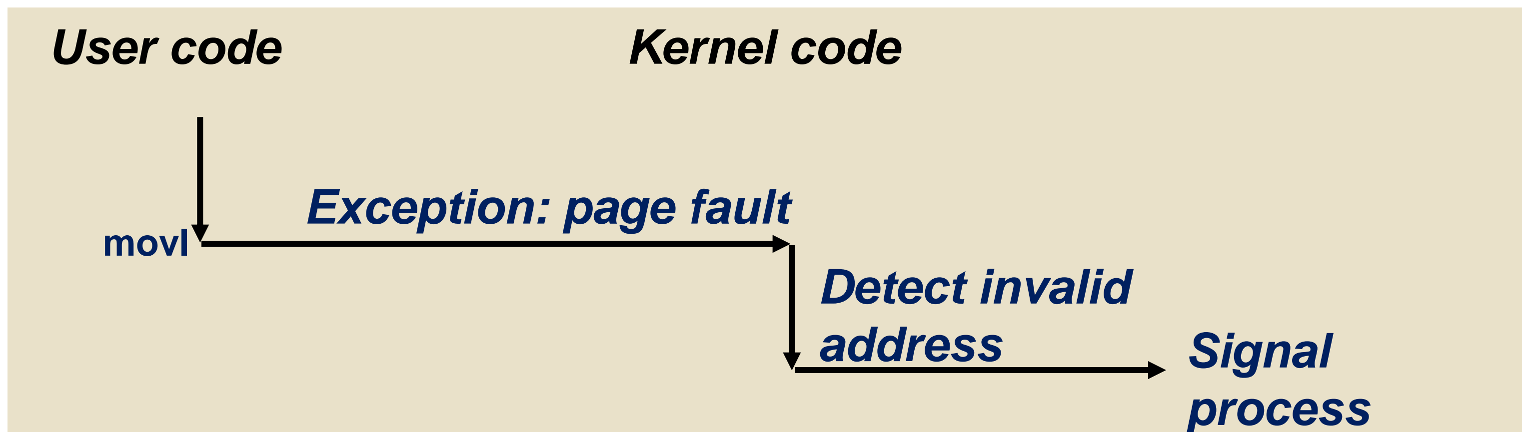
```
int a[1000];
main(){
    a[5000] = 13;
}
```

Memory Reference

- User writes to memory location
- Address is not valid

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

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



Exceptional Control Flow

Mechanisms exist at all levels of a computer system

- Change in control flow in response to a system event (i.e., change in system state)

Low level Mechanisms

1. Exceptions and interrupts
   - Change in control flow in response to a system event (i.e., change in system state)
   - Implemented with a combination of hardware and OS software

Higher Level Mechanisms

2. Process context switch
   - Implemented via hardware timer and OS software
3. Signals
   - Implemented via OS
4. Nonlocal jumps (setjmp/longjmp)
   - Implemented via C language runtime library

Processes

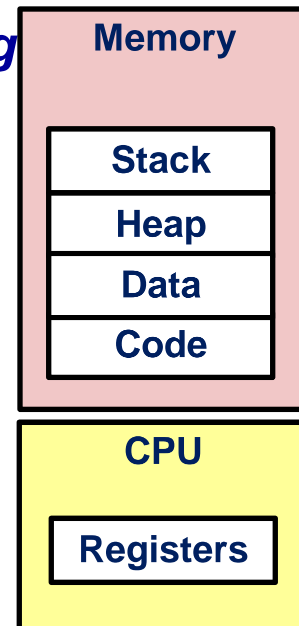
Exceptions and Processes

Exceptions instrumental for process management

A process is an instance of a running program.

Process provides each program with two key abstractions:

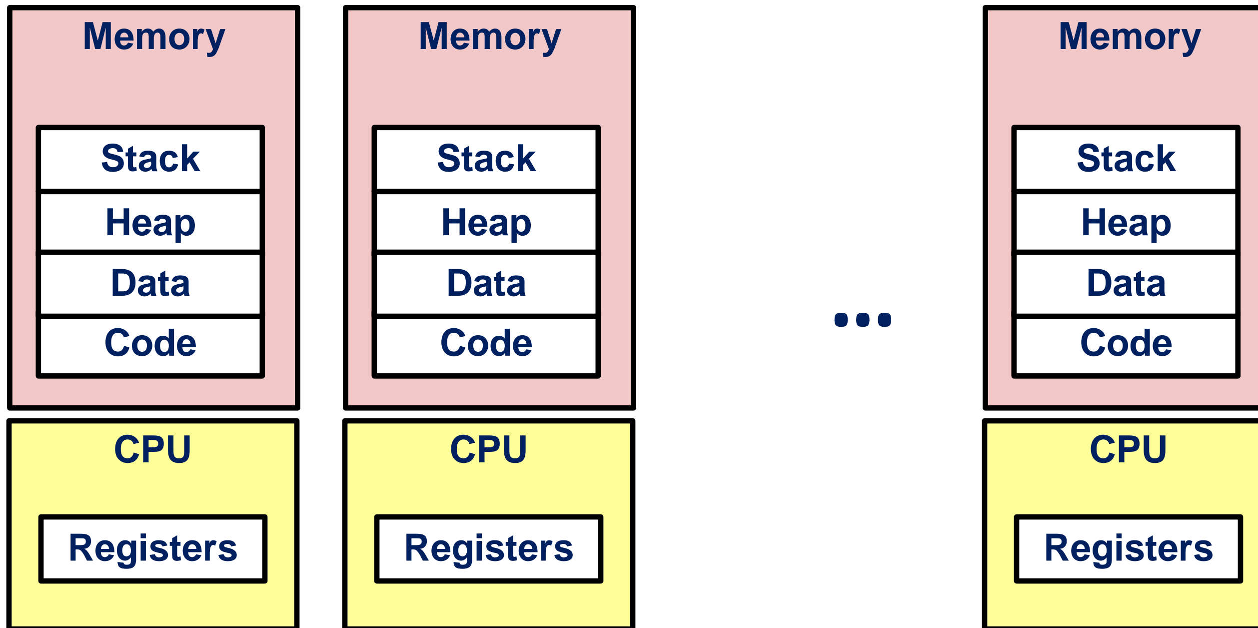
- **Logical control flow**
  - **Each program seems to have exclusive use of the CPU**
  - **Provided by kernel mechanism called *context switching***
- **Private address space**
  - **Each program seems to have exclusive use of main memory.**
  - **Provided by kernel mechanism called *virtual memory***



How are these Illusions maintained?

- **Process executions interleaved (multitasking)**
- **Address spaces managed by virtual memory system**

Multiprocessing: The Illusion



CPU runs many processes

- Applications and background tasks (browsers, email, network services)

Processes continually switch

- When process needs I/O resource or timer event occurs
- CPU runs one process at a time, but it appears to user(s) as if all processes executing simultaneously

Multiprocessing

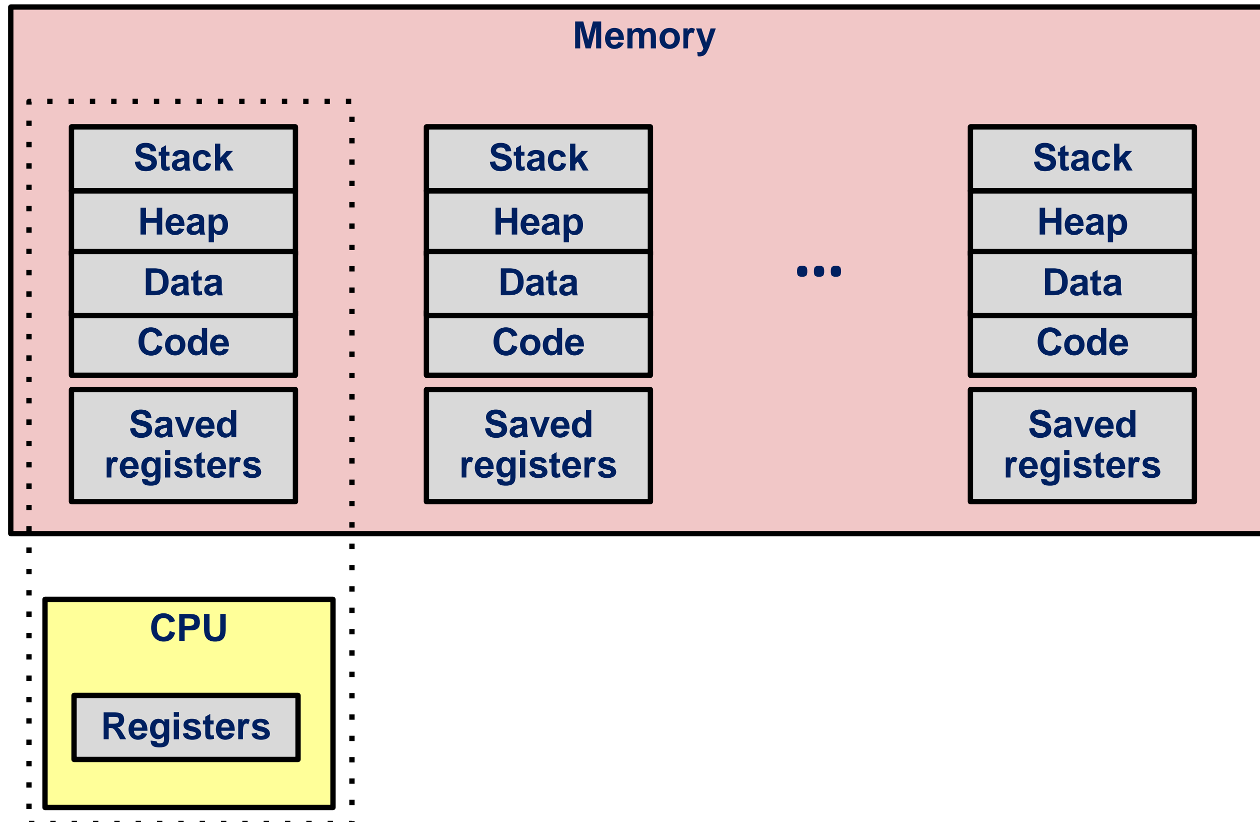
```
Tasks: 254 total,   1 running, 253 sleeping,   0 stopped,   0 zombie
%Cpu(s):  1.7 us,  1.6 sy,  0.0 ni, 96.4 id,  0.3 wa,  0.0 hi,  0.0 si,  0.0 st
KiB Mem:  32890072 total, 32204380 used,   685692 free,   782968 buffers
KiB Swap: 33459196 total,    23372 used, 33435824 free. 14354472 cached Mem

  PID USER      PR  NI    VIRT    RES    SHR S  %CPU %MEM     TIME+ COMMAND
 2375 wuchang   20   0 3175820 914904 864160 S   5.0  2.8   1601:26 VirtualBox
 8994 wuchang   20   0 1425804 126280  56668 S   4.6  0.4   0:07.72 chrome
 9035 wuchang   20   0  449308  64872  38552 S   3.0  0.2   0:02.72 chrome
25310 root      20   0  320724 119724  38060 S   2.3  0.4 127:30.36 Xorg
 9121 wuchang   20   0  903836 183412  26108 S   1.7  0.6   0:08.28 chrome
25783 wuchang   20   0  653192  31364  14136 S   1.0  0.1   6:23.03 gnome-term+
```

Running program “top”

- System has 254 processes
- Identified by Process ID (PID)

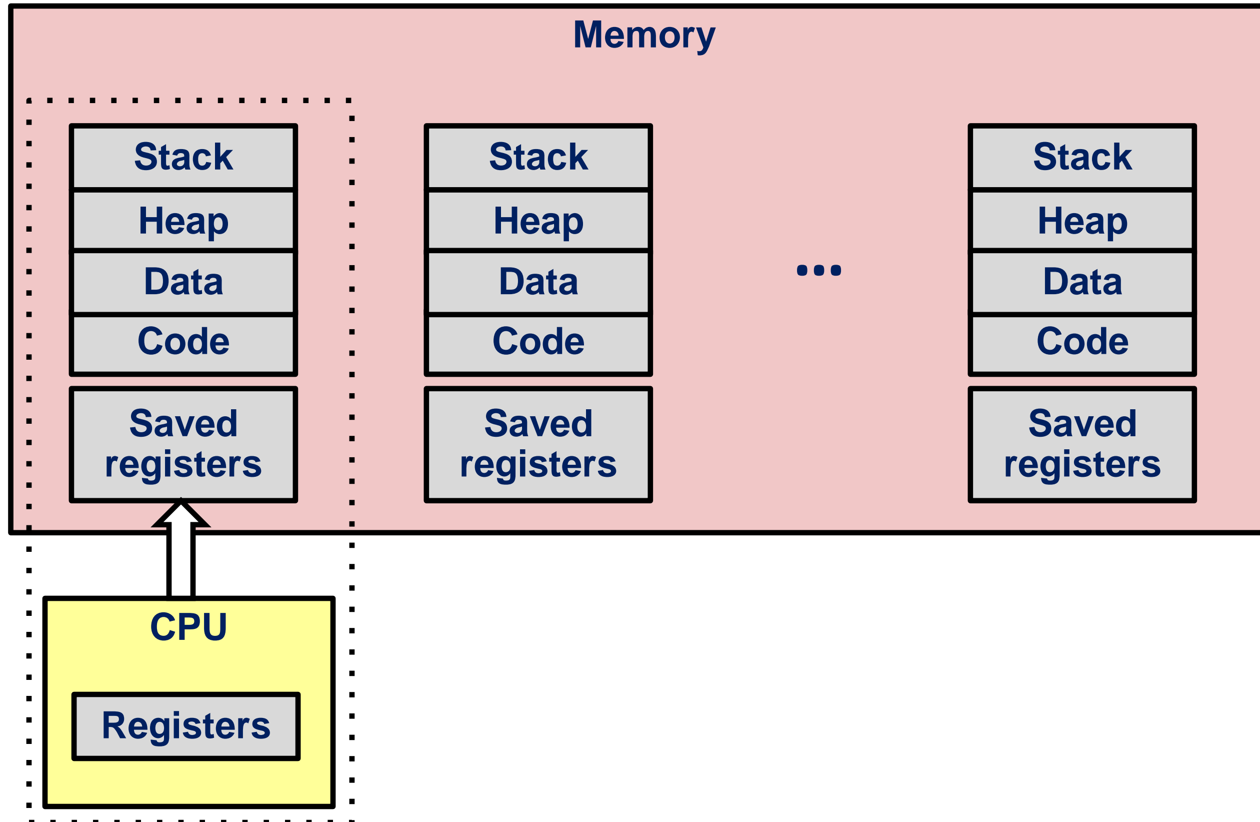
Multiprocessing: The Reality



Single processor executes multiple processes concurrently

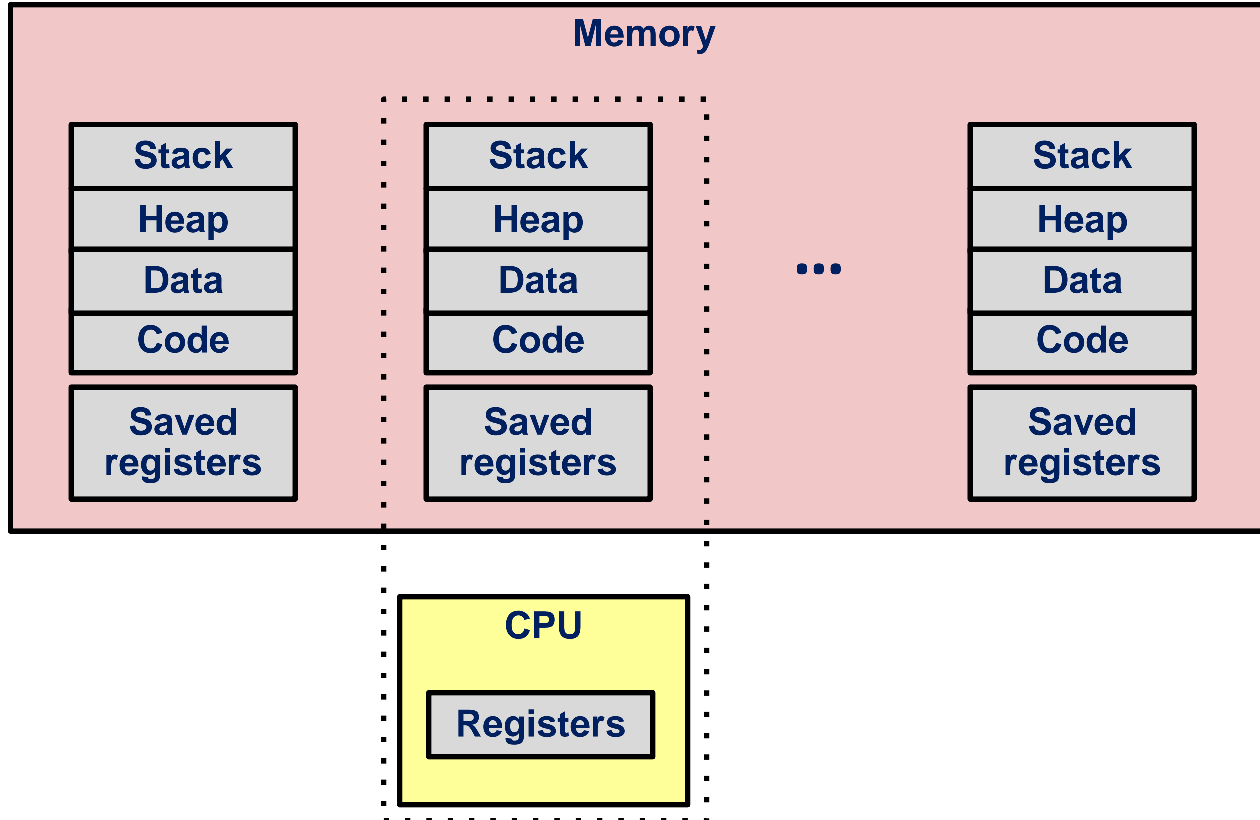
- Process executions interleaved (multitasking)
- Address spaces managed by virtual memory system (later)
- Register values for nonexecuting processes saved in memory (usually)

Multiprocessing: The Reality



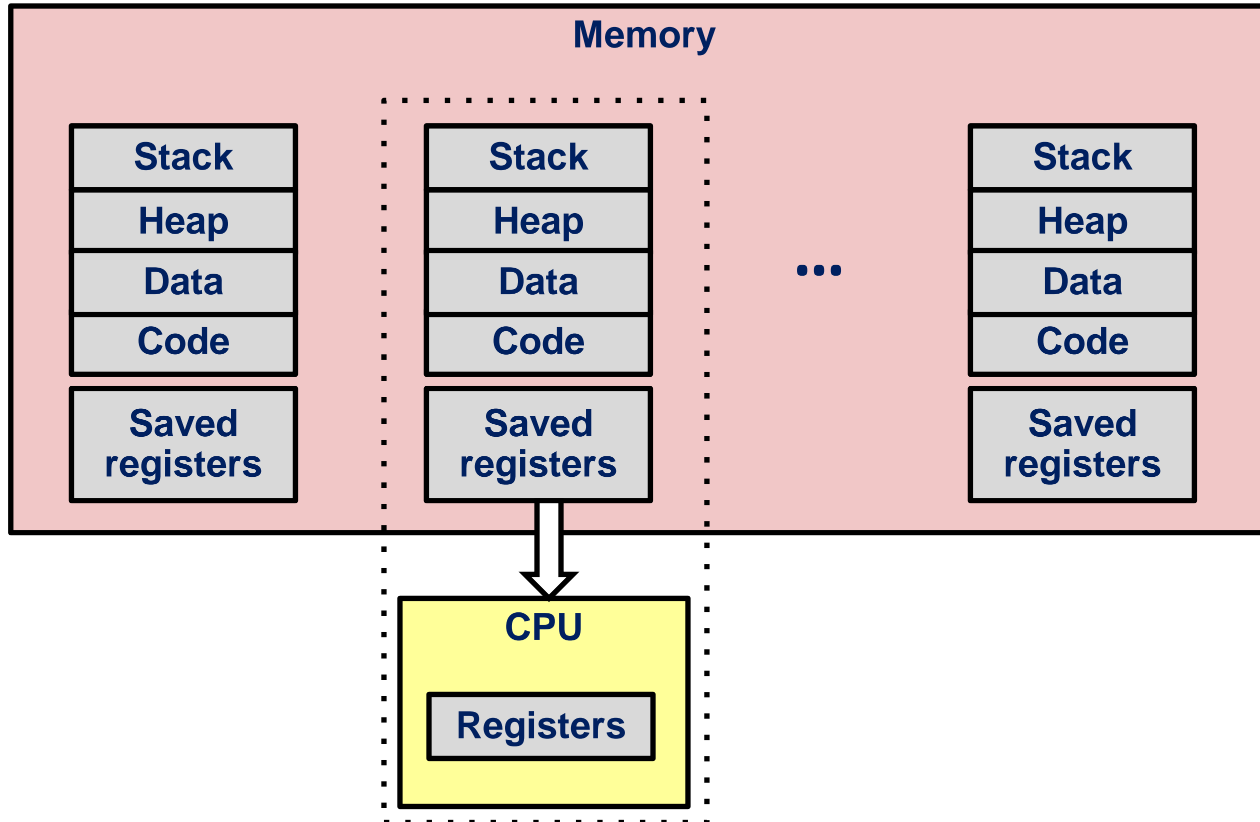
Save current registers in memory

Multiprocessing: The Reality



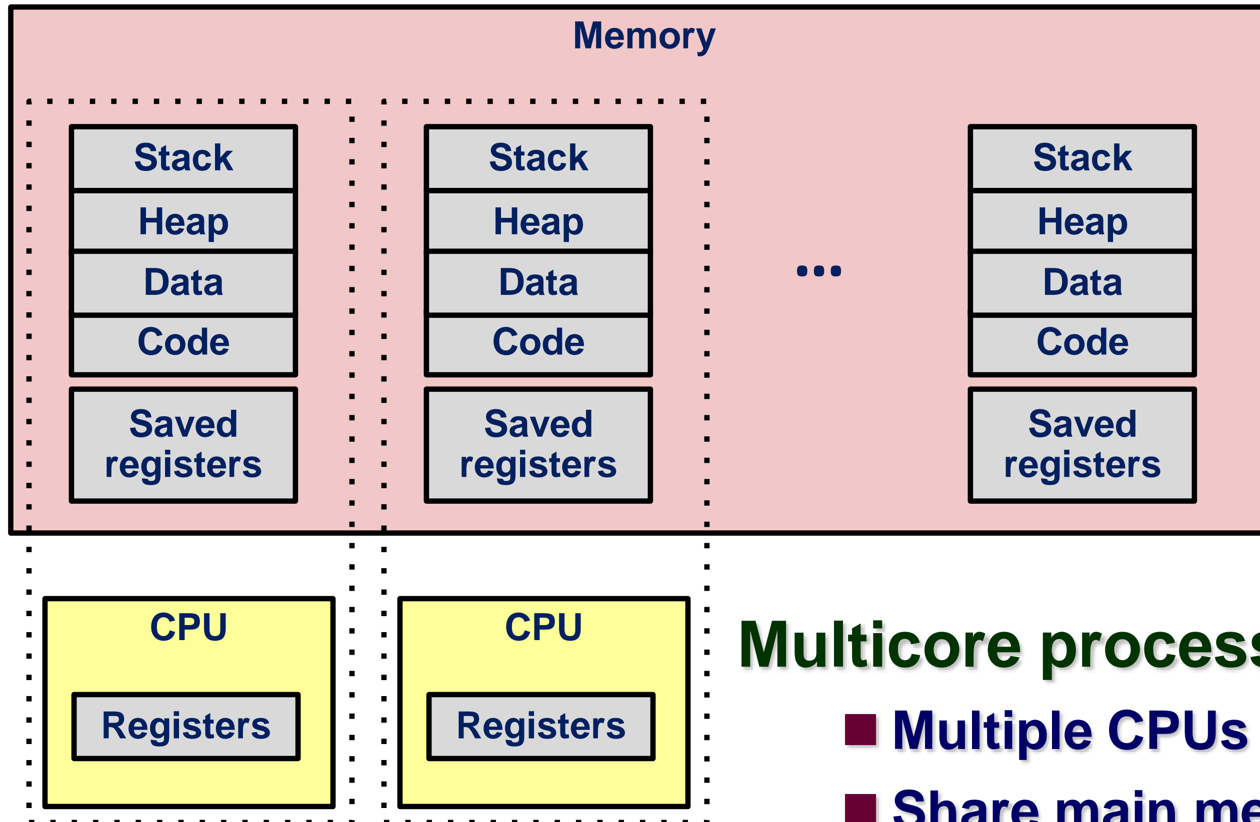
Schedule next process for execution

Multiprocessing: The Reality



Load saved registers and switch address space (context switch)

Multiprocessing: The Reality



Multicore processors

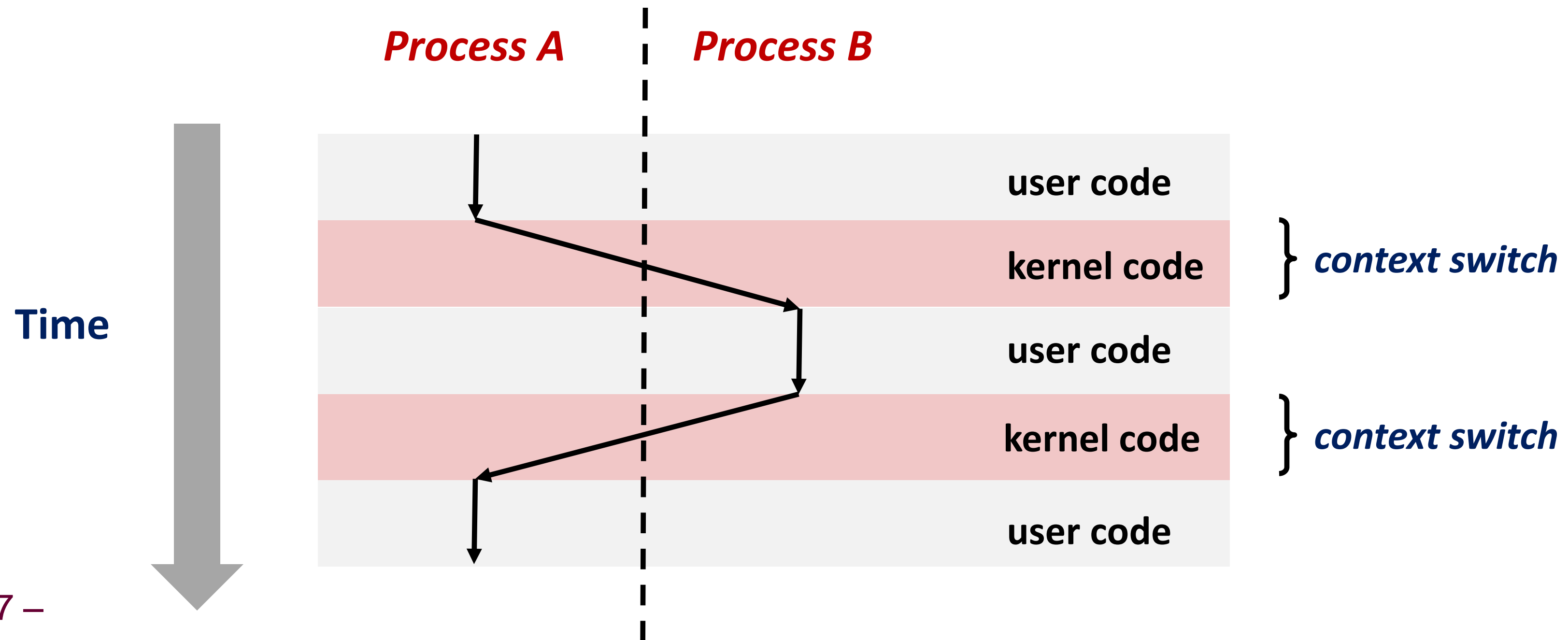
- Multiple CPUs on single chip
- Share main memory (and some of the caches)
- Each executes a separate process
- Scheduling of processors onto cores done by kernel

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 control in C

Basic Functions

- `fork()` spawns new process
  - Called once, returns twice
- `exit()` terminates own process
  - Called once, never returns
  - Puts it into “zombie” status
- `wait()` and `waitpid()` wait for and reap terminated children
- `execl()` and `execve()` run a new program in an existing process
  - Called once, (normally) never returns

Terminating Processes

Process terminates for one of three reasons:

- Receiving a signal whose default action is to terminate
- 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 child's process ID (PID) to parent process
- Returns 0 to the child process,
- Child is *almost* identical to parent

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

Fork

Call once, return twice

- Distinguish by return value of `fork`

Concurrent execution

- Can't predict execution order of parent and child

Duplicate, separate address spaces

- `x` has a value of `1` when `fork` returns in both parent and child
  - Subsequent changes to `x` are independent
- Child gets identical copies of the parent's open file descriptors
  - `stdout` same in both parent and child
- Only differences
  - Return value from fork different
  - Child has a different PID than parent

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

What about errors?

System Call Error Handling

On error, Unix system-level functions typically return -1 and set global variable errno to indicate cause.

- Return status should be checked after every system-level function

Example:

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

Error-handling wrappers

Can be simplified using wrappers:

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

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

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

```
void unix_error(char *msg) /* Unix-style error */
{
    fprintf(stderr, "%s: %s\n", msg, strerror(errno));
    exit(0);
}
```

Fork Example #1

```
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("PID %d with x = %d\n", getpid(), x);
}
…
Parent has x = 0
PID 23223 with x = 0
Child has x = 2
PID 23224 with x = 2
```

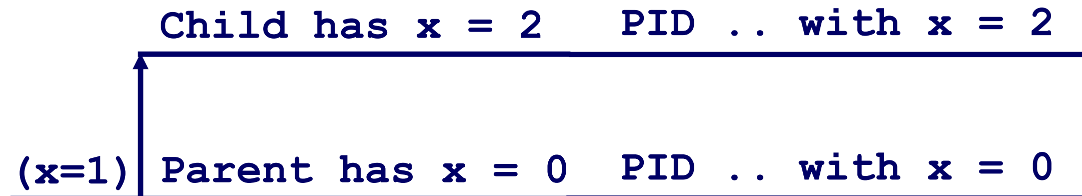
```
#include <sys/types.h>
#include <unistd.h>

pid_t getpid(void);  /* Get process ID */
pid_t getppid(void); /* Get parent process ID */
```

Fork Example #1

Graph visualization

- Time on x-axis
- Vertices are `fork` calls
- Child spawned on y-axis

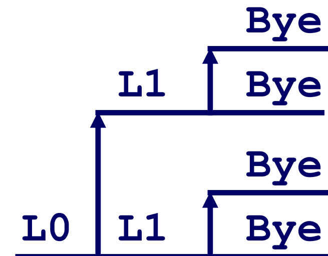



```
void fork1() {
    int x = 1;
    pid_t pid = fork();
    if (pid == 0) {
        printf("Child has x = %d\n", ++x);
    } else {
        printf("Parent has x = %d\n", --x);
    }
    printf("PID %d with x = %d\n", getpid(), x);
}
```

Fork Example #2

Both parent and child continue forking

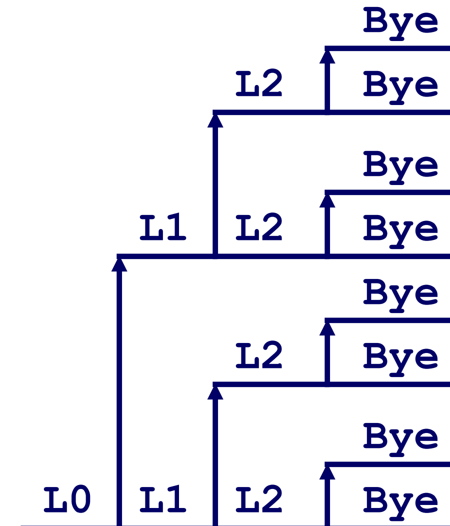
```
void fork2()
{
    printf("L0\n");
    fork();
    printf("L1\n");
    fork();
    printf("Bye\n");
}
```



Fork Example #3

Both parent and child continue forking

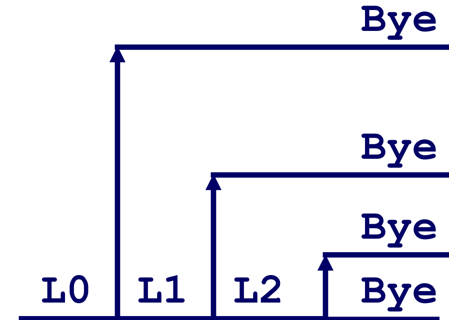
```
void fork3()
{
    printf("L0\n");
    fork();
    printf("L1\n");
    fork();
    printf("L2\n");
    fork();
    printf("Bye\n");
}
```



Fork Example #4

Nested `fork` in parent

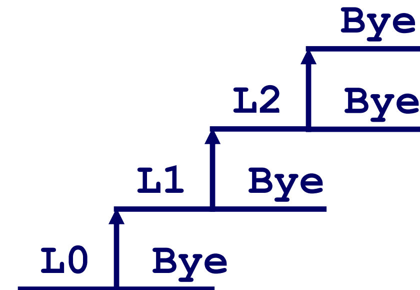
```
void fork4()
{
    printf("L0\n");
    if (fork() != 0) {
        printf("L1\n");
        if (fork() != 0) {
            printf("L2\n");
            fork();
        }
    }
    printf("Bye\n");
}
```



Fork Example #5

Nested `fork` in child

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



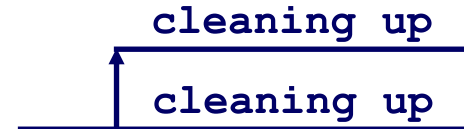
Fork Example #6

`atexit()`

- Registers a function to be executed upon exit

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

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



Practice problem 8.2

Consider the following program

```
int main()
{
    int x = 1;
    if (fork() == 0)
       printf("printf1: x=%d\n",++x);
    printf("printf2: x=%d\n",--x);
    exit(0);
}
```

- **What is the output of the child process?**

  printf1: x=2

  printf2: x=1

- **What is the output of the parent process?**

  printf2: x=0

Practice problem 8.11

Consider the following program

```
int main()
{
    int i;
    for (i = 0; i < 2; i++)
        fork();
    printf("hello!\n");
    exit(0);
}
```

- **How many "hello" output lines does this program print?**

4

Practice problem 8.12

Consider the following program

```
void doit() {
    fork();
    fork();
    printf("hello\n");
    return;
}

int main()
{
    doit();
    printf("hello\n");
    exit(0);
}
```

- **How many "hello" output lines does this program print?**

8

Reaping Child Processes

When a child process terminates, it stays around and consumes system resources until reaped by parent

- Must keep its exit status to deliver to parent
- Called a "zombie"
  - Living corpse, half alive and half dead

Parent must "reap" terminated child

- Performed by parent on child via `wait` or `waitpid`
- Parent is given exit status information
- Kernel then deletes zombie child process

What if Parent Doesn't Reap?

- Child zombie stays around
- If parent terminates without reaping a child, then child will be reaped by `init` process

Zombie Example

```
void fork7() {
    if (fork() == 0) {
        /* Child */
        printf("Terminating Child, PID = %d\n", getpid());
        exit(0);
    } else {
        printf("Running Parent, PID = %d\n", getpid());
        while (1)
            ; /* Infinite loop */
    }
}
```

```
linux> ./forks 7 &
[1] 6639
Running Parent, PID = 6639
Terminating Child, PID = 6640
linux> ps
  PID TTY          TIME CMD
 6585 ttyp9    00:00:00 tcsh
 6639 ttyp9    00:00:03 forks
 6640 ttyp9    00:00:00 forks <defunct>
 6641 ttyp9    00:00:00 ps
linux> kill 6639
[1]    Terminated
linux> ps
  PID TTY          TIME CMD
 6585 ttyp9    00:00:00 tcsh
 6642 ttyp9    00:00:00 ps
```

`ps` shows zombie 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);
    }
}
```

```
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
```

`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 set to a status indicating why the child process terminated
- Checked using macros defined in wait.h

  `WIFEXITED, WEXITSTATUS, WIFSIGNALED, WTERMSIG,`

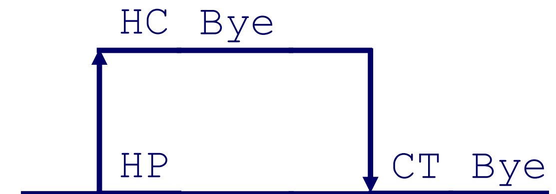
  `WIFSTOPPED, WSTOPSIG, WIFCONTINUED`

  - See textbook for details

wait: Synchronizing with children

```
void fork9() {
    int child_status;

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



Wait Example

Arbitrary order when multiple children

- WIFEXITED macro to see if child exited normally
- WEXITSTATUS macro 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);
    }
}
```

```
Child 3565 terminated with exit status 103
Child 3564 terminated with exit status 102
Child 3563 terminated with exit status 101
Child 3562 terminated with exit status 100
Child 3566 terminated with exit status 104
```

waitpid

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

- Suspends process until specific child terminates

```
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);
    }
}
```

`wait/waitpid` Examples

Using `wait` (`fork10`)

```
Child 3565 terminated with exit status 103
Child 3564 terminated with exit status 102
Child 3563 terminated with exit status 101
Child 3562 terminated with exit status 100
Child 3566 terminated with exit status 104
```

Using `waitpid` (`fork11`)

```
Child 3572 terminated with exit status 104
Child 3571 terminated with exit status 103
Child 3570 terminated with exit status 102
Child 3569 terminated with exit status 101
Child 3568 terminated with exit status 100
```

Practice problem 8.3

Consider the following program

```
int main()
{
    if (fork() == 0)
       printf("a");
    else {
       printf("b");
       waitpid(-1,NULL,0);
    }
    printf("c");
    exit(0);
}
```

- **List all possible output sequences of this program.**

  abcc, bacc, acbc
  Can not have bcac!

Practice problem 8.4

Consider the following program

```
int main()
{
    int status;
    pid_t pid;
    printf("Hello\n");
    pid = fork();
    printf("%d\n", !pid);
    if (pid != 0) {
        if (waitpid(-1,&status,0) > 0) {
            if (WIFEXITED(status) != 0)
                printf("%d\n", WEXITSTATUS(status));
    }
    printf("Bye\n");
    exit(2);
}
```

- **How many output lines does this program generate?**

6

- **What is one possible ordering of these output lines?**

Hello => 0 => 1 => Bye => 2 => Bye

Suspending processes

Two methods

`sleep()`

- Suspends a running process for a specified period of time

```
#include <unistd.h>
unsigned int sleep(unsigned int secs);
```

- Returns 0 if the requested amount of time has elapsed
- Returns the number of seconds still left to sleep otherwise
  - Process can prematurely wakeup if interrupted by a signal

`pause()`

- Suspends a running process until a signal is received

```
#include <unistd.h>
int pause(void);
```

Practice problem 8.5

Write a wrapper function for `sleep()` called `snooze()` that behaves exactly as `sleep()` but prints out a message describing how long the process actually slept

```
unsigned int snooze(unsigned int secs) {
    unsigned int rc = sleep(secs);
    printf("Slept for %u of %u secs.\n", secs-rc, secs);
    return rc;
}
```

Running new programs

`fork` creates an identical copy of a process

- How can one run a new program not a duplicate of one?

`execve`: Loading and Running Programs

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

Loads and runs in current process

- Executable file `filename`
- … with argument list `argv`
- … and environment variable list `envp`
  - "name=value" strings (e.g. `SHELL=/bin/zsh`)

Overwrites code, data, and stack

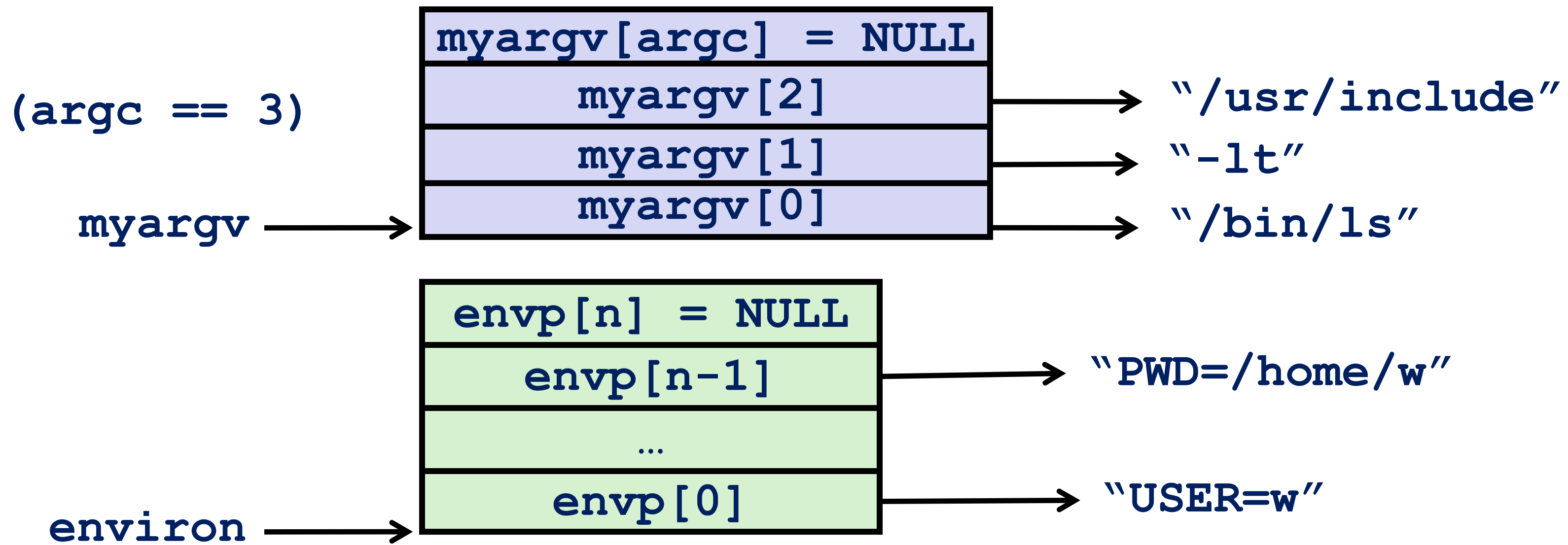
- Retains only PID, open files, and signal context

Called once and never returns

- … unless there is an error

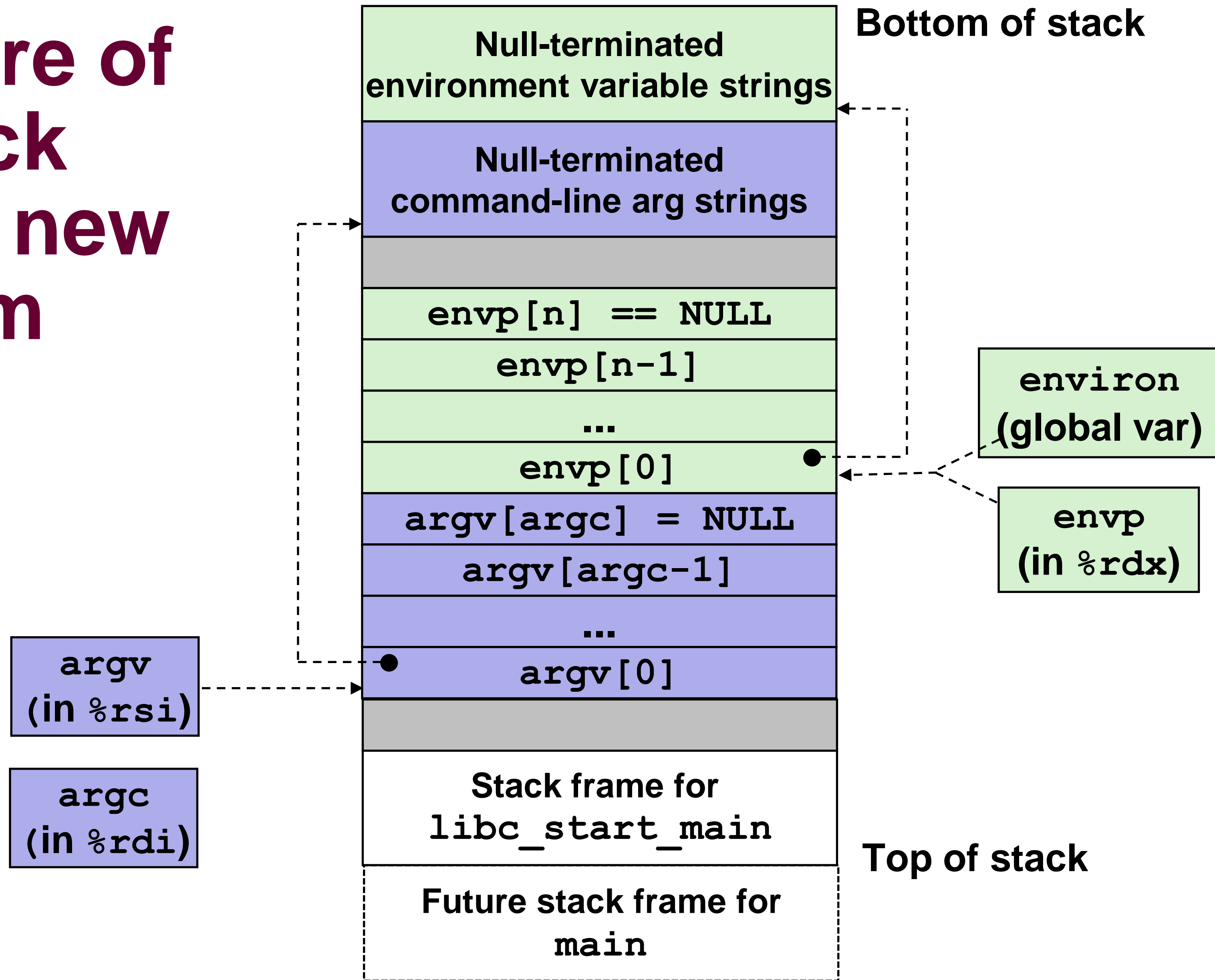
execve Example

- Executes "/bin/ls -lt /usr/include" 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);
    }
}
```

Structure of the stack when a new program starts



Practice problem 8.6

Write a program called `myecho` that prints its command line arguments and environment variables

```
int main()
{
    int i;
    printf("Command line arguments:\n");
    for (i=0; argv[i] != NULL; i++)
        printf("    argv[%2d]: %s\n", i, argv[i]);
    printf("\n");
    printf("Environment variables:\n");
    for (i=0; envp[i] != NULL; i++)
        printf("    envp[%2d]: %s\n", i, envp[i]);
    exit(0);
}
```

Summarizing

Exceptions

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

Processes

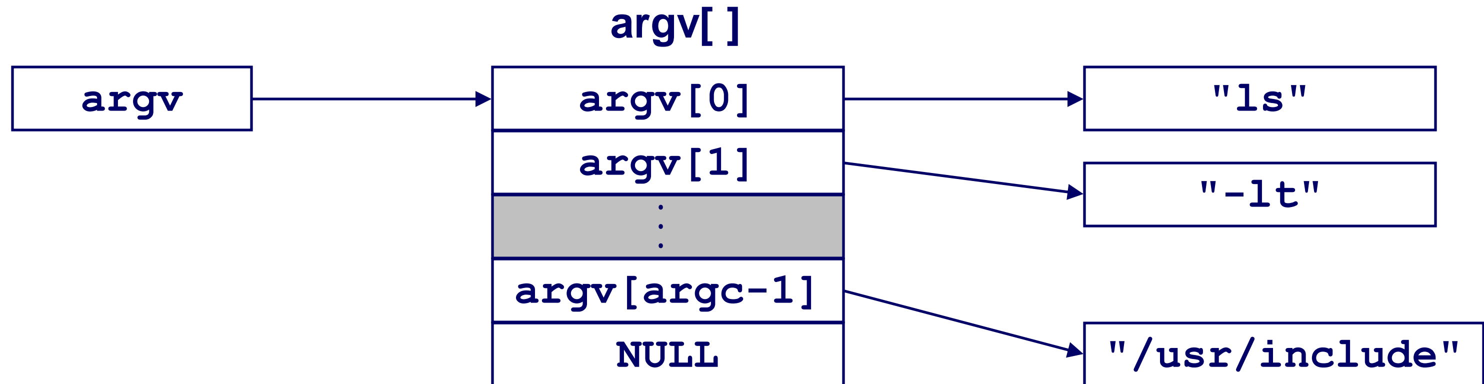
- At any given time, system has multiple active processes
- Each process appears to have total control of processor + private memory space
- Only one can execute at a time, though

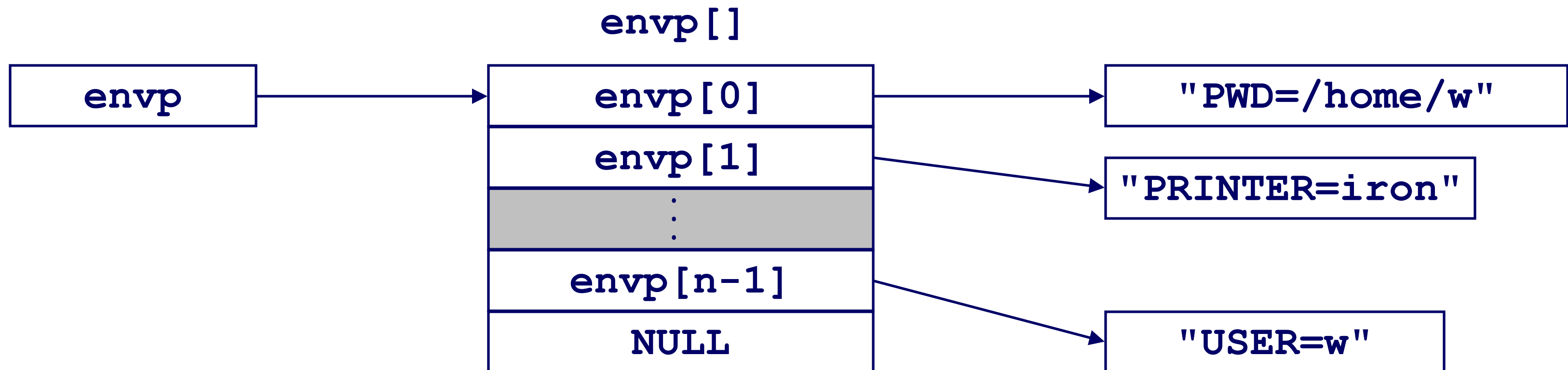
Process control

- Spawning (`fork`), terminating (`exit`), and reaping (`wait`) processes
- Executing programs (`exec`)

Extra slides

argv and envp





Environment variables

Strings that are specified as name-value pairs in the form `NAME=VALUE`

- Type `` `printenv' `` to see settings in the shell
- Some environment variables
  - `PATH` : Path for finding commands
  - `LD_LIBRARY_PATH` : Path for finding dynamic libraries
  - `USER` : Name of user
  - `SHELL` : Current shell
  - `HOSTNAME` : Name of machine
  - `HOME` : Path to user's home directory
  - `PWD` : Current directory

Environment variables

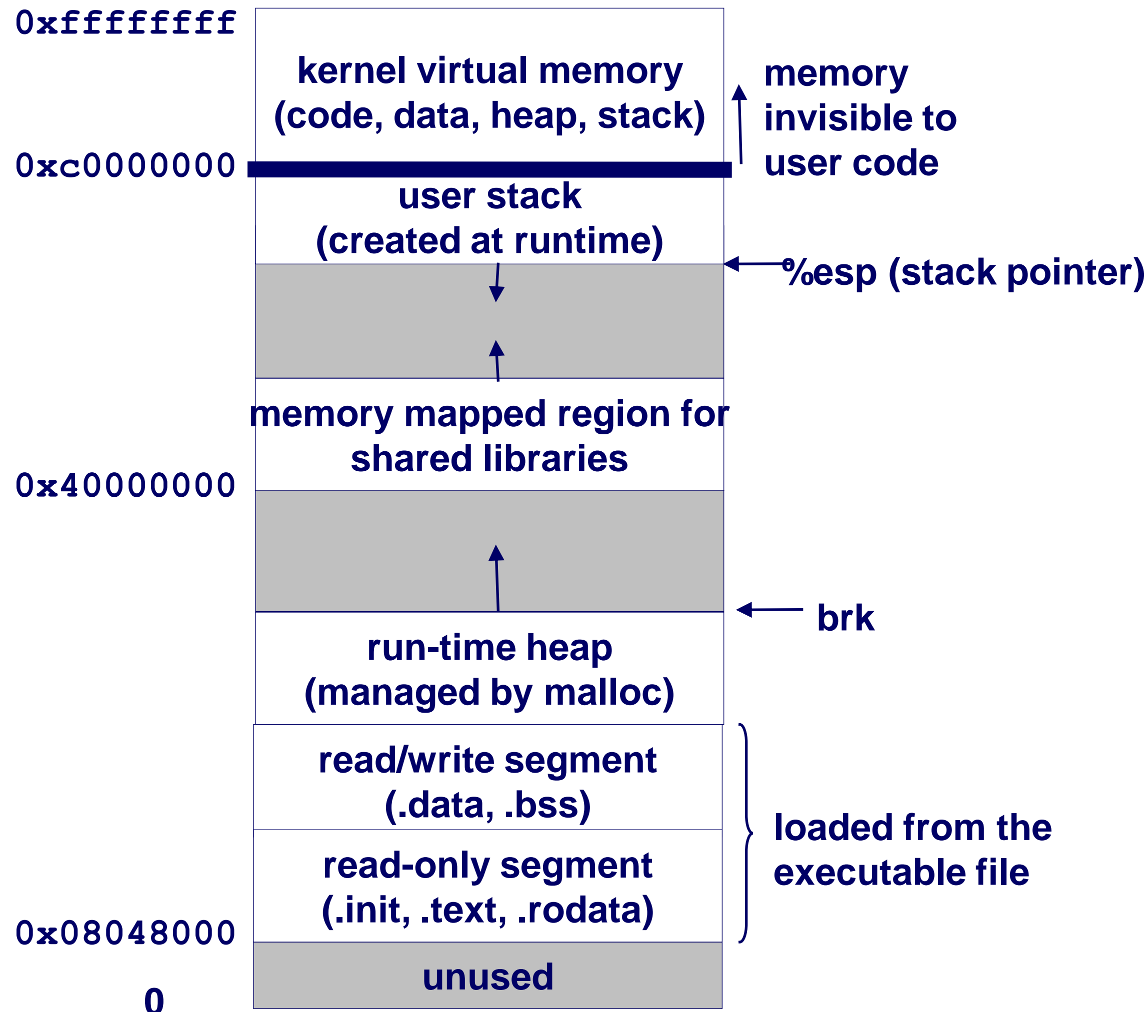
Setting up the environment within C

```
#include <stdlib.h>
char *getenv(const char *name);
int setenv(const char *name, const char
  *newvalue, int overwrite);
void unsetenv(const char *name);
```

- `getenv`: Given a name, it returns a pointer to a string containing its value or NULL if not set in environment
- `setenv`: Sets an environment variable pointed to by name to a value pointed to by newvalue. Replaces the old value if it exists, if overwrite field is non-zero
- `unsetenv`: Deletes an environment variable and its setting

Private Address Spaces

Each process has its own private address space.



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 is *suspended* and will not be scheduled until further notice
- Terminated
  - Process is stopped permanently