Processes and System Calls

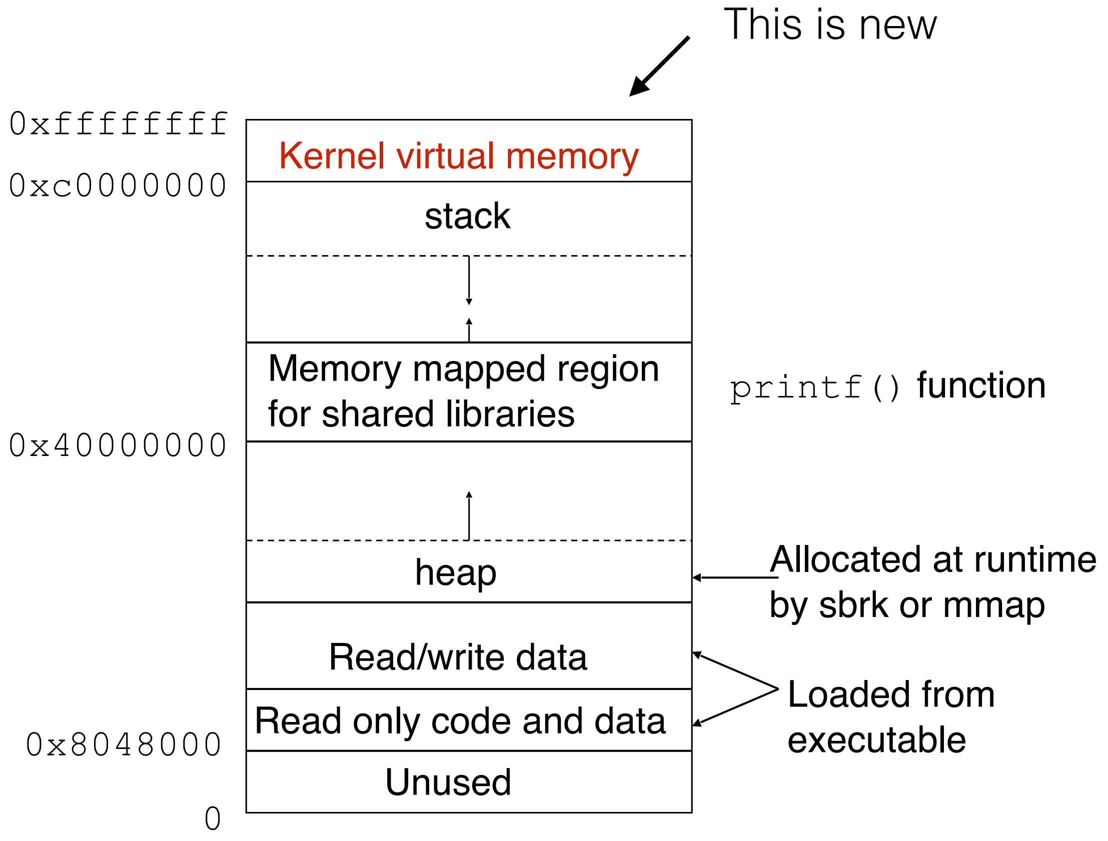
OS view of a process?

- A process is a running program
- What data do we need to keep track of about a process?
- What does a process look like in memory?

Breakout

- Do question 1 of Exercise 6
- Also, talk to your group about what you remember about how a process is laid out in memory.
  - What are the different regions of memory used for? (CSC209 refresher)

Program layout in memory



OS View of a Process

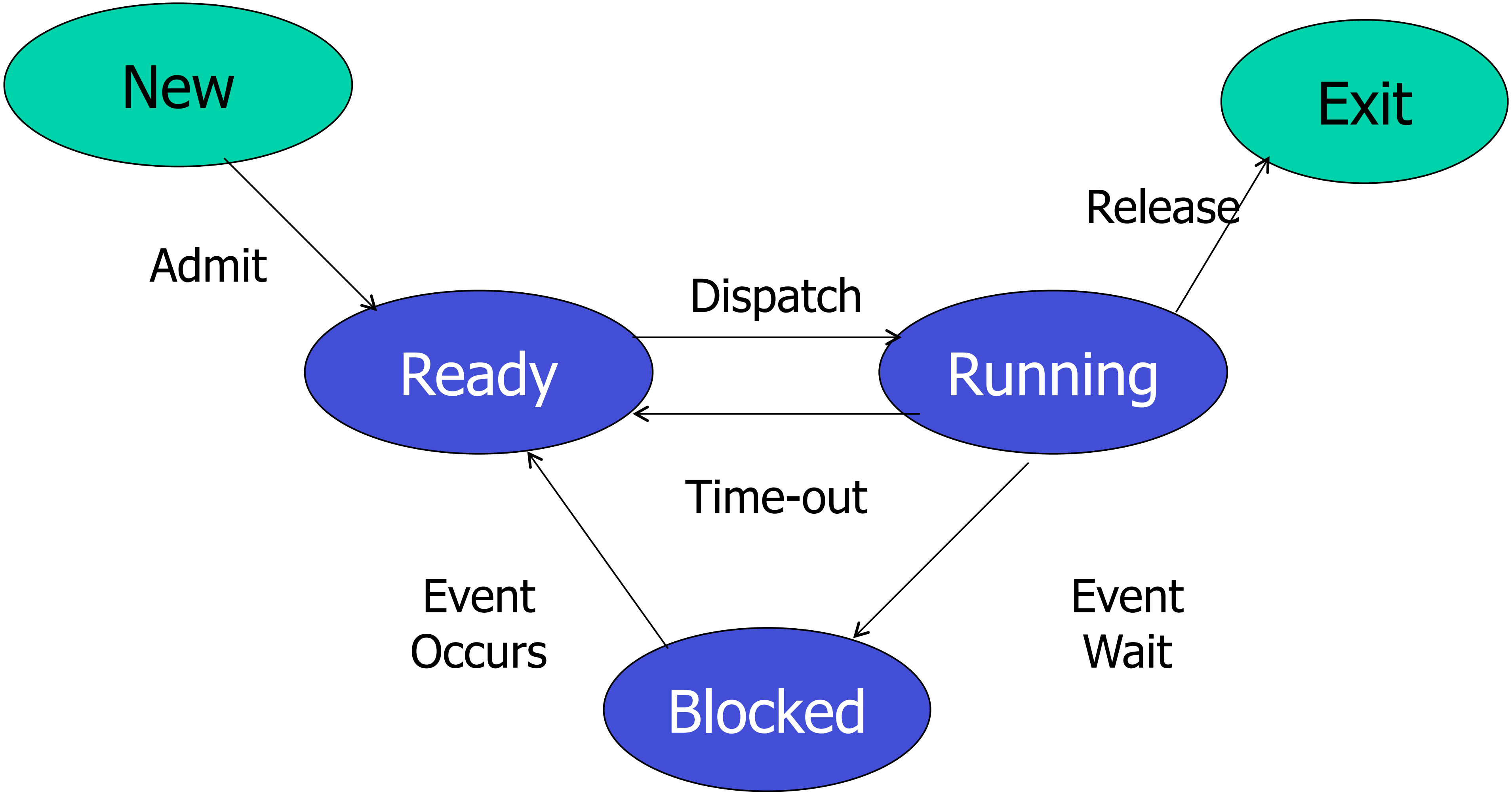
- A process contains all of the state for a program in execution
    - An address space
    - The code + data for the executing program
    - An execution stack encapsulating the state of procedure calls
    - The program counter (PC) indicating the next instruction
    - A set of general-purpose registers with current values
    - A set of operating system resources
        - Open files, network connections, signals, etc.
- A process is named using its process ID (PID)

Linux Process Control Block

- Called task_struct in Linux
  - On teach.cs in /usr/src/linux-headers-4.15.0-118/include/linux/sched.h

```
struct task_struct {
/* these are hardcoded - don't touch */
  volatile long state; /* -1 unrunnable, 0 runnable, >0 stopped */
  long counter; long priority; unsigned long signal;
  unsigned long blocked; /* bitmap of masked signals */
  unsigned long flags; /* per process flags, defined below */
  int errno; long debugreg[8]; /* Hardware debugging registers */
  struct exec_domain *exec_domain;
  /* various fields */
  struct linux_binfmt *binfmt;
  struct task_struct *next_run, *prev_run;
  unsigned long saved_kernel_stack;
  unsigned long kernel_stack_page;
  int exit_code, exit_signal;
  struct files_struct *files;
  struct signal_struct *signal;
  struct sighand_struct*sighand;
…
```

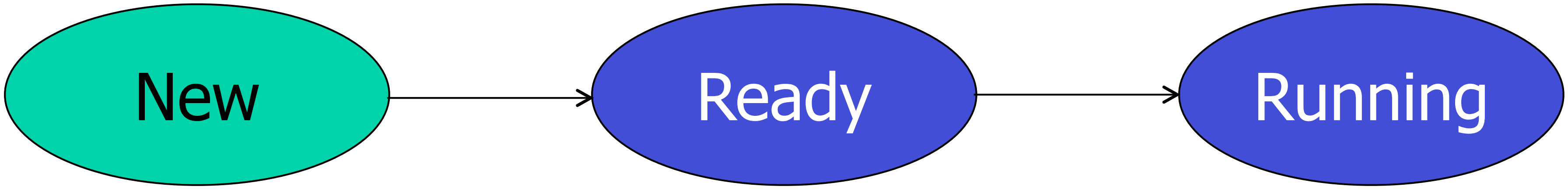
Process Life Cycle



Keeping track of processes

- OS maintains a collection of state queues that represent the state of all processes in the system
- Typically one queue for each state (ready, waiting for event X)
- As a process changes state, its PCB is unlinked from one queue and linked into another

From Program to Process



1. Create new process
   - Create new Process Control Block (PCB) and user address space structure
   - Allocate memory
2. Load executable
   - Initialize start state for process
   - Change state to “ready”
3. Dispatch process
   - Change state to “running”

State Change: Ready to Running

- context switch == switch the CPU to another process by:
    - saving the state of the old process
    - loading the saved state for the new process

- When can this happen?
    1. Process calls `yield`() system call (voluntarily)
    2. Process makes a system call and is blocked
    3. Timer interrupt handler decides to switch processes
        - Why would we ever need this?

Process Creation

- A process is created by another process
- Parent is creator, child is created
- In Linux, the parent is the “PPID” field of “ps –f”
- What creates the first process?
    - init (PID 1)
- In some systems, the parent defines (or donates) resources and privileges for its children
    - Unix: Process User ID is inherited – children of your shell execute with your privileges
- After creating a child, the parent may either wait for it to finish its task or continue in parallel (or both)

Process Creation: Unix

- In Unix, processes are created using fork() system call `int fork()`
- `fork()`
  - Creates and initializes a new PCB
  - Creates a new address space
  - Initializes the address space with a copy of the entire contents of the address space of the parent
  - Initializes the kernel resources to point to the resources used by parent (e.g., open files)
  - Places the PCB on the ready queue
- Fork returns in two processes
  - Returns the child’s PID to the parent, “0” to the child

Fork example

```
int main(int argc, char *argv[]) {

  char *name = argv[0];
  int child_pid = fork();

  if (child_pid == 0) {
    printf(“Child of %s is %d\n”, name, getpid());
    return 3;
  } else {
    printf(“My child is %d\n”, child_pid);
    return 0;
  }
}
```

What does this program print?

Process Creation: Unix (2)

- How do we actually start a new program?

```
int exec(char *prog, char *argv[])
```

- **`exec()`**
  - Stops the current process
  - Loads program “prog” into the process’ address space
  - Initializes hardware context and args for the new program
  - Places the PCB onto the ready queue
  - Note: It **does not** create a new process

- What does it mean for **`exec`** to return?

Process Destruction

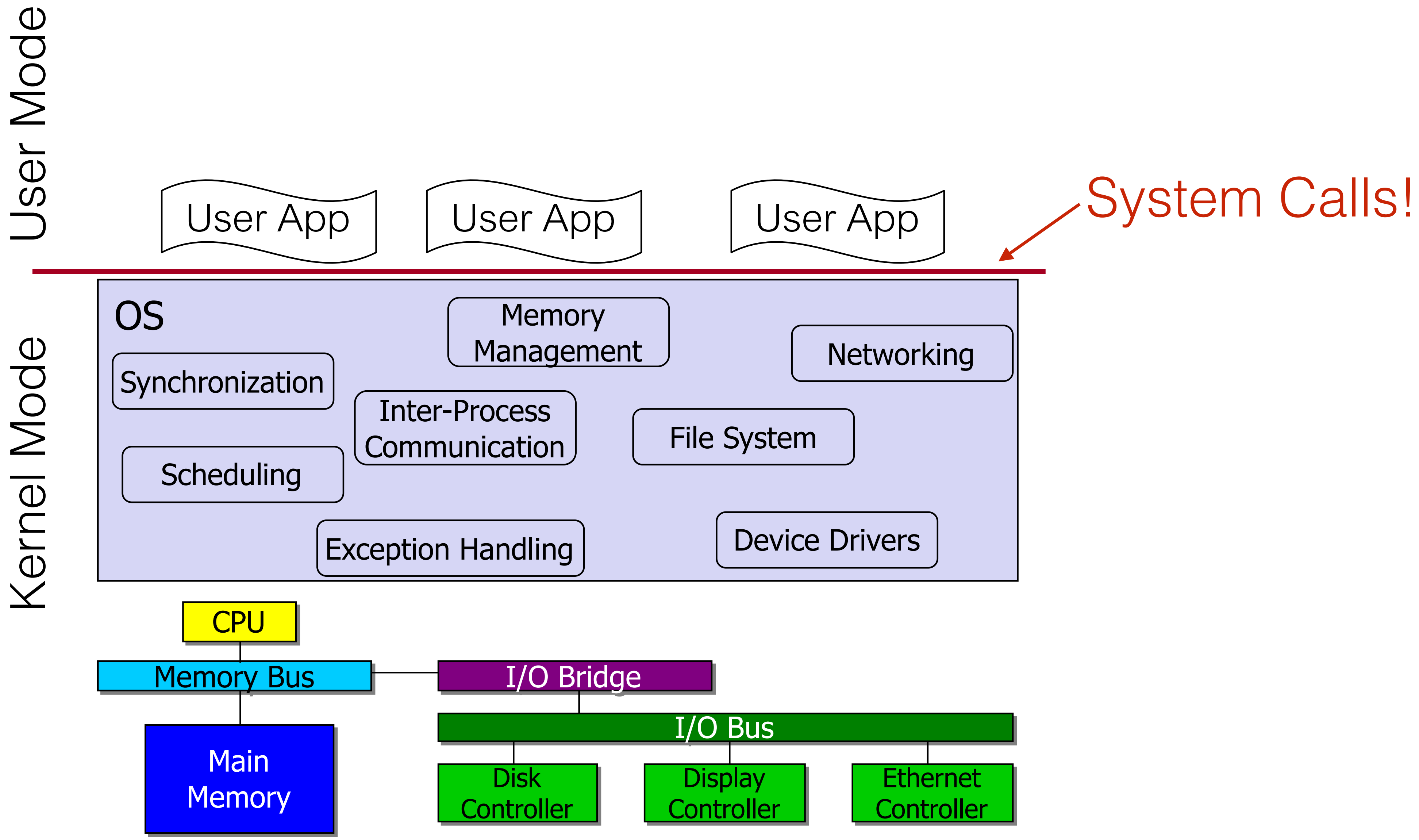
- `exit()`
  - On `exit()`, a process voluntarily releases all resources
  - But… OS can't discard everything immediately
- Why?
  - Must stop running the process to free everything
  - Requires *context switch* to another process
  - Parent may be waiting or asking for the return value
- `exit()` doesn't cause all data to be freed!

Zombies

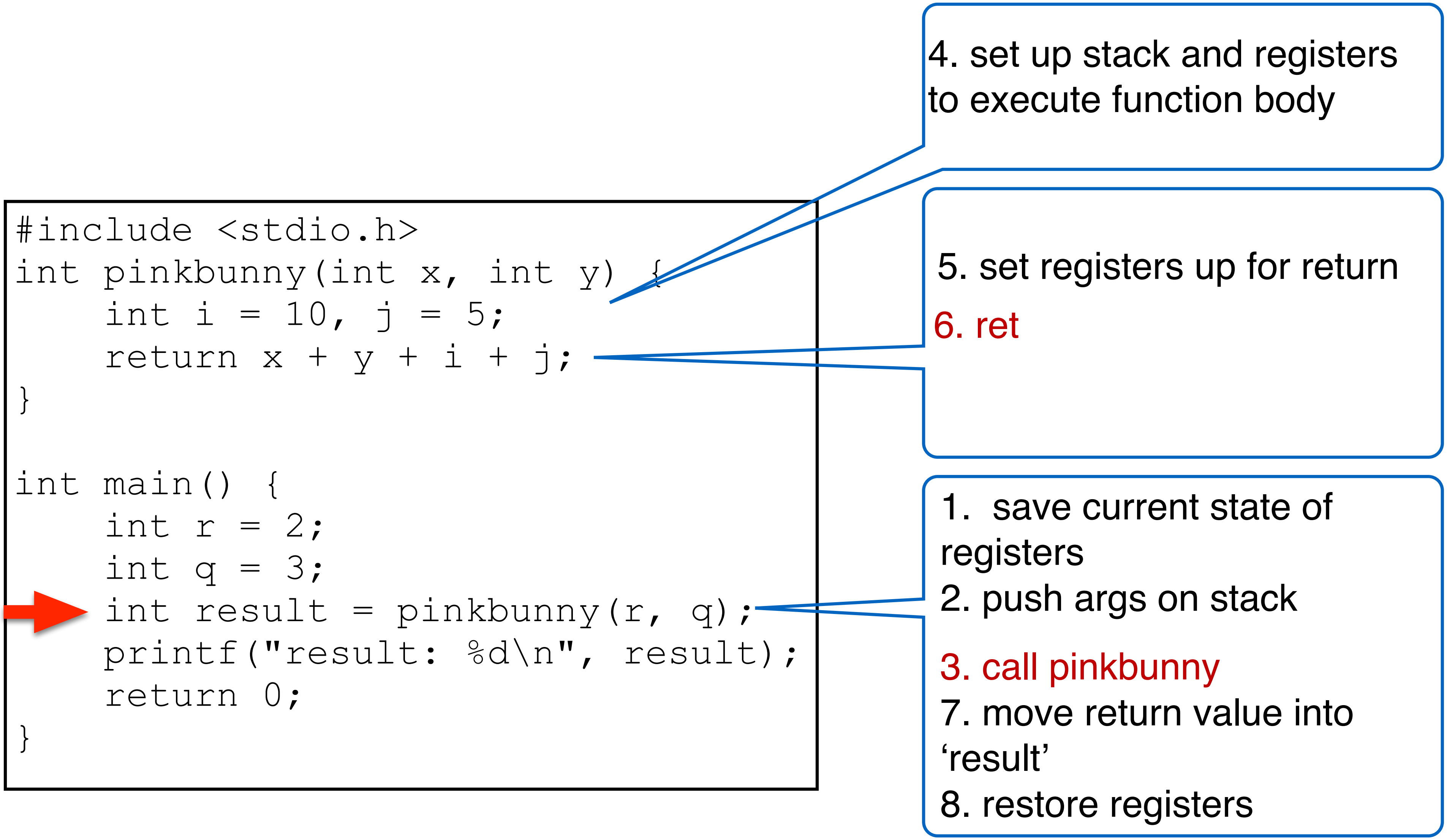
- When a process exits, almost all of its resources are deallocated
  - Address space is freed, files are closed, etc.
- Some OS data structures retain the process's exit state
- The process retains its process PID
- It is a zombie until its parent cleans it up

Source: Plants vs Zombies

Requesting OS Services



But first, what is a function call?



```
#include <stdio.h>
int pinkbunny(int x, int y) {
    int i = 10, j = 5;
    return x + y + i + j;
}

int main() {
    int r = 2;
    int q = 3;
    int result = pinkbunny(r, q);
    printf("result: %d\n", result);
    return 0;
}
```

4. set up stack and registers to execute function body

5. set registers up for return

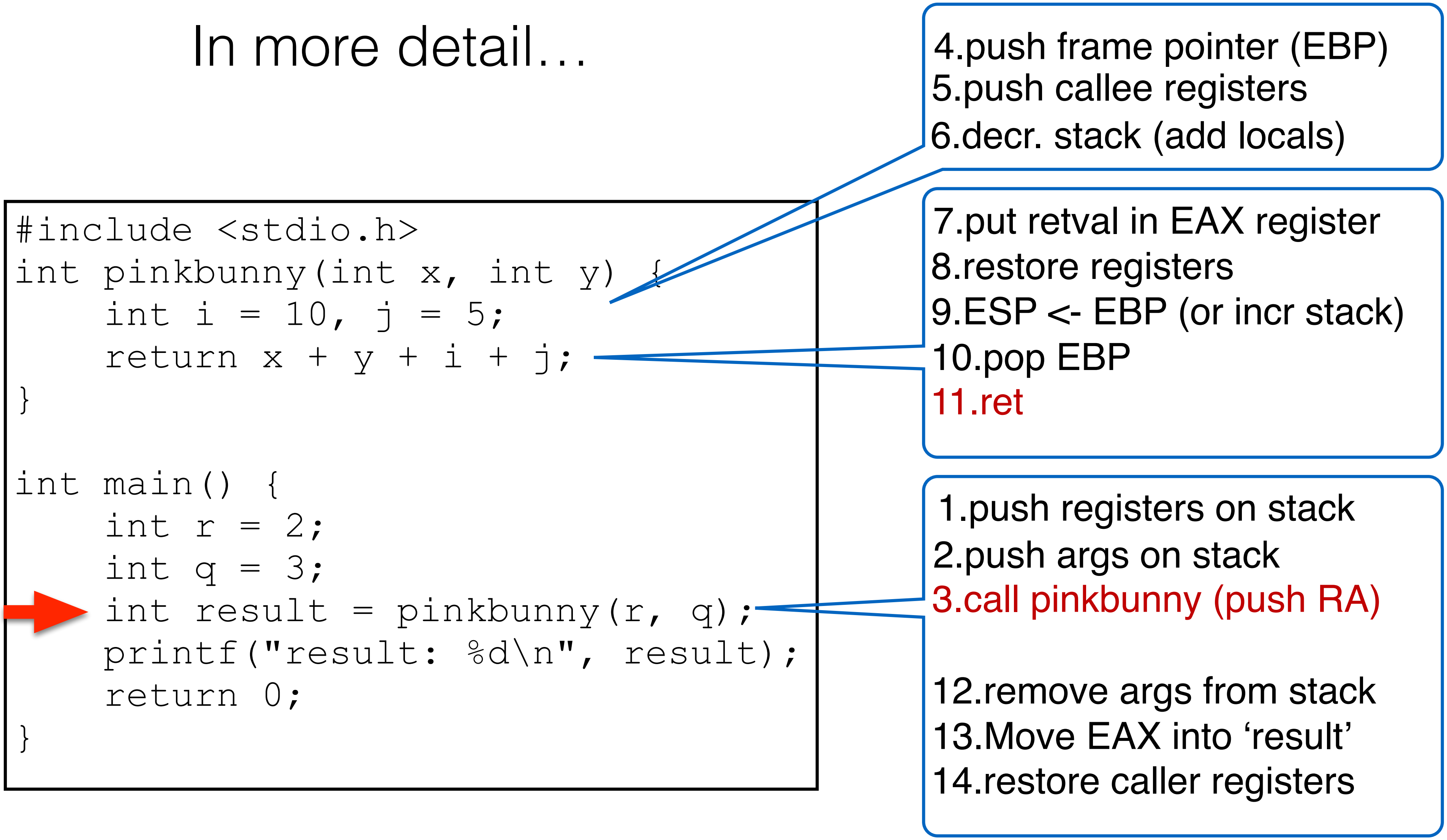
6. ret

1. save current state of registers
2. push args on stack
3. call pinkbunny
7. move return value into ‘result’
8. restore registers

Src: http://www.cs.virginia.edu/~evans/cs216/guides/x86.html

But first, what is a function call?

In more detail…



```
#include <stdio.h>
int pinkbunny(int x, int y) {
    int i = 10, j = 5;
    return x + y + i + j;
}

int main() {
    int r = 2;
    int q = 3;
    int result = pinkbunny(r, q);
    printf("result: %d\n", result);
    return 0;
}
```

4.push frame pointer (EBP)
5.push callee registers
6.decr. stack (add locals)

7.put retval in EAX register
8.restore registers
9.ESP <- EBP (or incr stack)
10.pop EBP
11.ret

1.push registers on stack
2.push args on stack
3.call pinkbunny (push RA)

12.remove args from stack
13.Move EAX into ‘result’
14.restore caller registers

Src: http://www.cs.virginia.edu/~evans/cs216/guides/x86.html

How does a process execute?

CPU reads one instruction at a time:

Fetch

- read instruction into instruction **register**

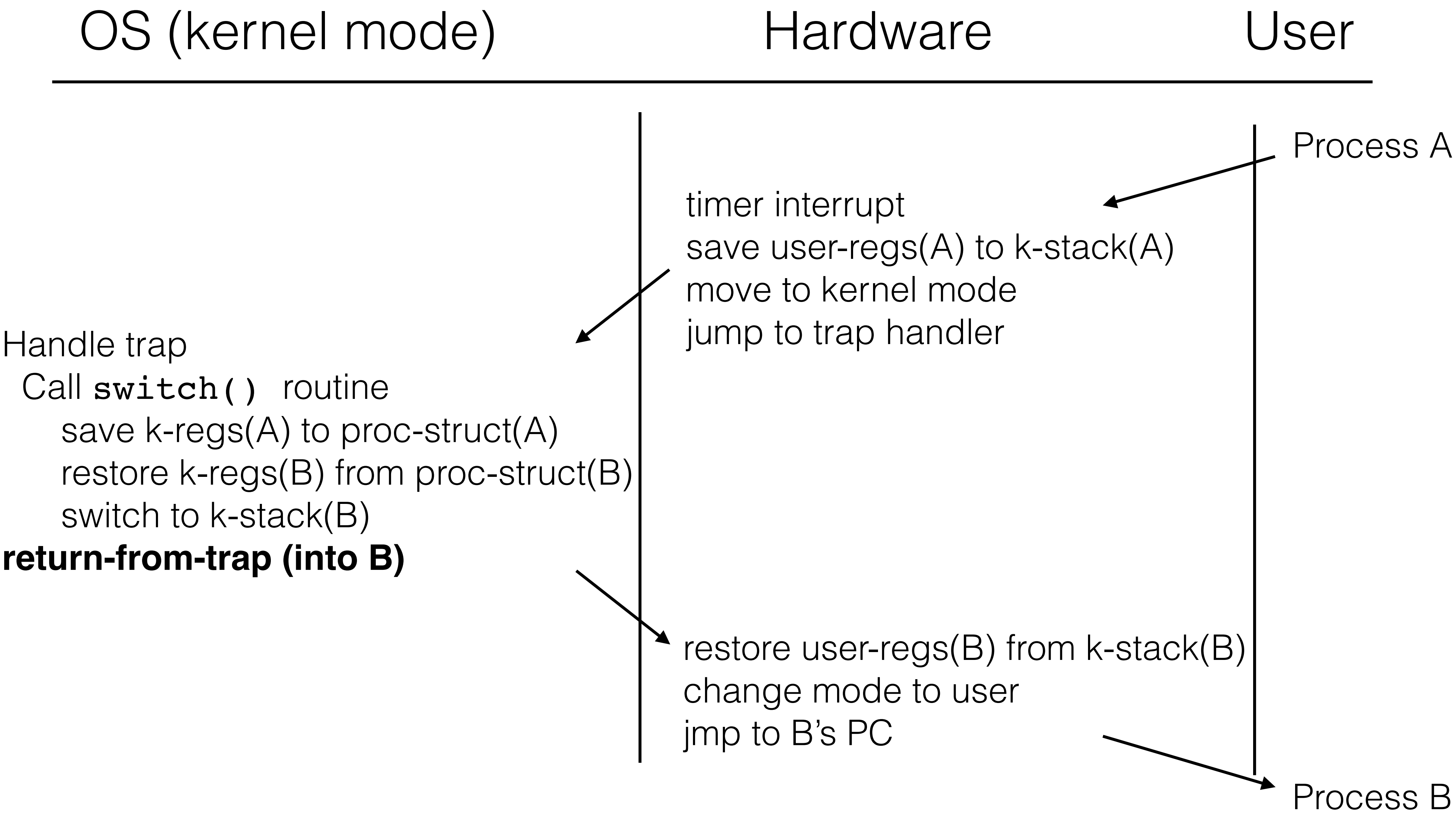
Decode

- read any needed data into **register(s)**

Execute

- pass values from **registers** to ALU (arithmetic logic unit), write result back to **register**
- store result to main memory

Context Switch



OS (kernel mode)

Hardware

User

Process A

timer interrupt
save user-regs(A) to k-stack(A)
move to kernel mode
jump to trap handler

Handle trap
Call **`switch()`** routine
save k-regs(A) to proc-struct(A)
restore k-regs(B) from proc-struct(B)
switch to k-stack(B)
return-from-trap (into B)

restore user-regs(B) from k-stack(B)
change mode to user
jmp to B's PC

Process B

What is a system call?

- A system call is a function call that invokes the operating system”

- Whenever an application wants to use a resource that the OS manages, it asks permission!

- How do you actually invoke the operating system?

- How do we keep applications from just using a resource without asking permission?

Interrupts

- Can be caused by hardware or software
- Interrupts signal CPU that a hardware device has an event that needs attention
    - E.g. Disk I/O completes, etc.
- Interrupts signal errors or requests for OS intervention (a system call)
    - Often called an “exception” or “trap”
- CPU jumps to a pre-defined routine (the interrupt handler)
- An OS is an event-driven programs
    - The OS “responds” to requests

Boundary Crossings

- Getting to kernel mode
  - Explicit system call – request for service by application
  - Hardware interrupt
  - Software trap or exception
    - Hardware has table of “Interrupt service routines”
    - Saves registers before switching to kernel mode

- Kernel to user
  - When the OS is finished its task, get back to application
    - OS sets up registers, MMU, mode for application
    - Jumps to next application instruction

Enforcing Restrictions

- Hardware runs in user mode or system mode
- Some instructions are privileged instructions: they can only run in system mode
- On a “system call interrupt”, the mode bit is switched to allow privileged instructions to occur

Privileged instructions

- Access I/O device
  - Poll for IO, perform DMA, catch hardware interrupt
- Manipulate memory management
  - Set up page tables, load/flush the TLB and CPU caches (we'll see this later), etc.
- Configure various "mode bits"
  - Interrupt priority level, software trap vectors, etc.
- Call halt instruction
  - Put CPU into low-power or idle state until next interrupt
- These are enforced by the CPU hardware itself
  - Reminder: CPU has at least 2 protection levels: Kernel and user mode
  - CPU checks current protection level on each instruction!
  - What happens if user program tries to execute a privileged instruction?

System Call Interface

- User program calls C library function with arguments

- C library function passes arguments to OS
  - Includes a system call identifier!

- Executes special instruction (x86: INT) to trap to system mode
  - Interrupt/trap vector transfers control to a handler routine

- Syscall handler figures out which system call is needed and calls a routine for that operation

- How does this differ from a normal C language function call? Why is it done this way?

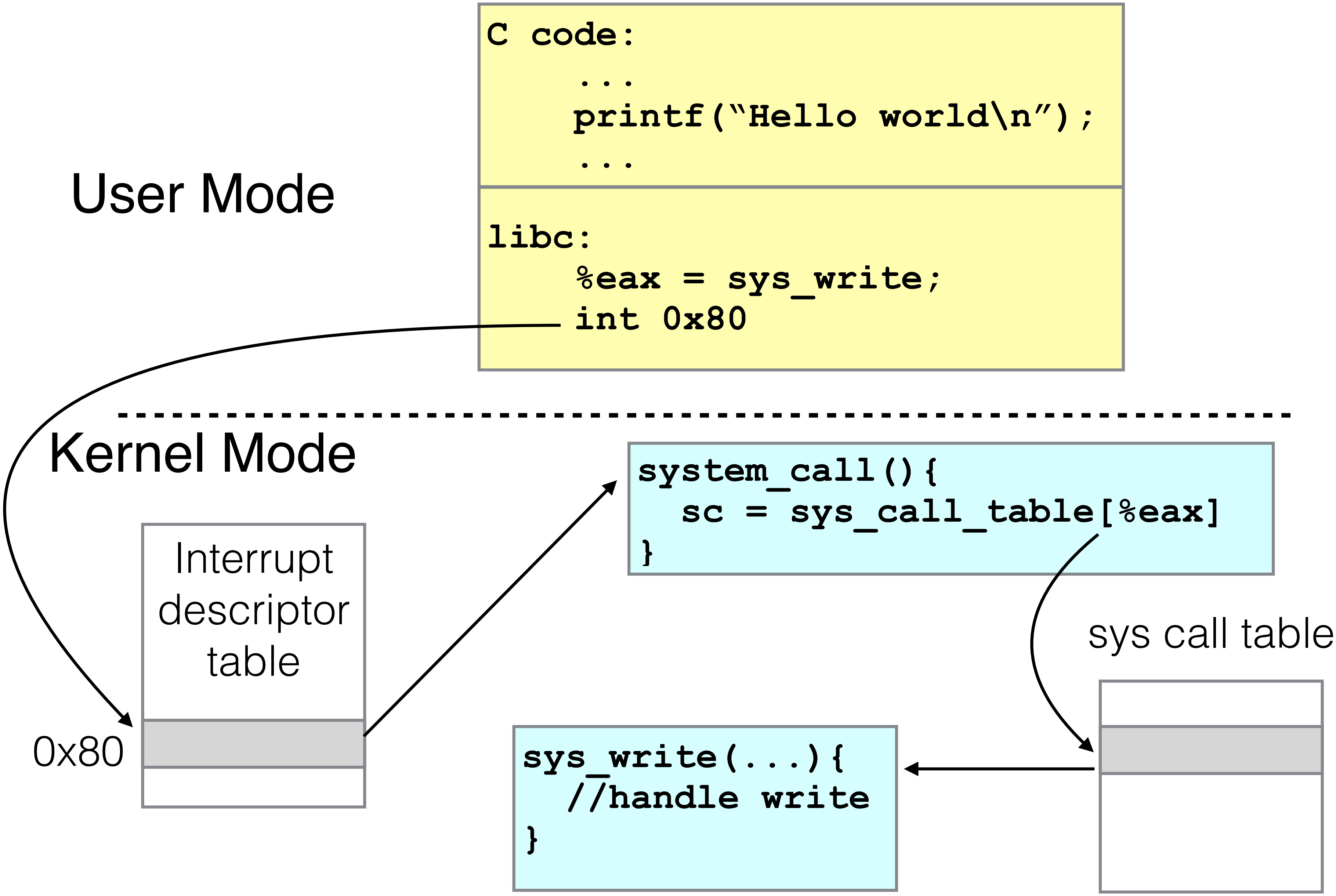
System Call Operation

- Kernel must verify arguments that it is passed
  - Why?

- A fixed number of arguments can be passed in registers
  - Often pass the address of a user buffer containing data (e.g., for write())
  - Kernel must copy data from user space into its own buffers

- Result of system call is returned in register EAX

Example: Linux `write` system call



```
C code:
    ...
    printf(“Hello world\n”);
    ...
```

```
libc:
    %eax = sys_write;
    int 0x80
```

```
system_call(){
  sc = sys_call_table[%eax]
}
```

```
sys_write(...){
  //handle write
}
```

System Calls in Linux

- Can invoke any system call from userspace using the syscall() function

```
syscall(syscall_no, arg1, arg2, arg3, ..)
```

- E.g.,
  - `const char msg[] = "Hello World!";`
  - `syscall(4, STDOUT_FILENO, msg, sizeof(msg)-1);`
  - Equivalent to: `write(STDOUT_FILENO, msg, sizeof(msg)-1);`

- Tracing system calls:
  - Powerful mechanism to trace system call execution for an application
  - Use the `strace` command
  - The `ptrace()` system call is used to implement `strace` (also used by gdb)
  - Library calls can be traced using `ltrace` command

System Call Dispatch

- Why do we need a system call table?
  - How would you get to the right routine?
  - If-then-else for each system call number?
  - Too inefficient!
- A system call is identified by a unique number
  - The system call number is passed into register %eax
  - Offers an index into an array of function pointers: the system call table!
- System call table: `sys_call_table[__NR_syscalls]` (approximately 300)
- See all syscalls in VM: /usr/src/linux-source-2.6.32/arch/x86/kernel/syscall_table_32.S

System call dispatch

1. Kernel assigns each system call type a system call number

2. Kernel initializes system call table, mapping each system call number to a function implementing that system call

3. User process sets up system call number and arguments

4. User process runs `int N` (on Linux, N=0x80)

5. Hardware switches to kernel mode and invokes kernel's interrupt handler for X (interrupt dispatch)

6. Kernel looks up syscall table using system call number

7. Kernel invokes the corresponding function

8. Kernel returns by running iret (interrupt return)

Passing System Call Parameters

- The first parameter is always the syscall number
  - Stored in eax
- Can pass up to 6 parameters:
  - ebx, ecx, edx, esi, edi, ebp
- If more than 6 parameters are needed, package the rest in a struct and pass a pointer to it as the 6th parameter
- Problem: must validate user pointers. Why? How?
- Solution: safe functions to access user pointers: copy_from_user(), copy_to_user(), etc.