# The Heap, Processes

## Computer Systems Programming, Summer 2025

**Instructors:** Joel Ramirez Travis McGaha

**TAs:** Ash Fujiyama Maya Huizar

The Dish, Stanford California

# Administrivia

- First Assignment (HW00 penn-vector)
  - Released already! Should have everything you need after this lecture
  - “Due” Monday (Fast Turnaround) next week 06/02
  - Mostly a C refresher

- Pre semester Survey
  - Anonymous
  - Short!
  - Out Tonight at Midnight, Due Wednesday the 4th

# Administrivia

- Second Assignment (HW01 penn-shredder)
  - Releases after Monday's lecture (should have everything you need by then)
  - Due Friday next week **06/06**
  - Intro to system calls, processes, etc.
  - Short Q&A and demo in lecture on Monday ☺

# Lecture Outline

- **C "Refresher"**
  - **Dynamic Memory vs the Stack**
  - **Structs**
- Processes
  - Overview
  - fork()
  - exec()

# Demo: get_input.c

- Lets code together a small program that:
  - Reads at max 100 characters from stdin (user input)
  - Truncates the input to only the first word
  - Prints that word out
  - Not allowed to use scanf, FILE*, printf, etc

Poll Everywhere

pollev.com/cis4480

- Two things are wrong with this function. What are they?
- How do we fix this function w/o changing the function signature?

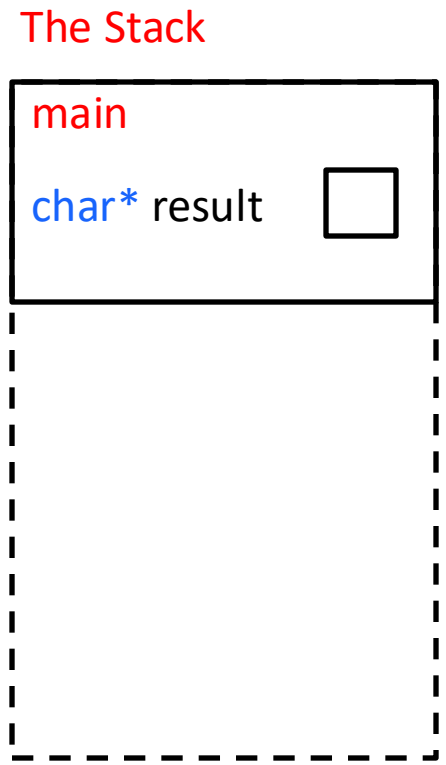
```c
#define MAX_INPUT_SIZE 100

char* read_stdin() {
  char str[MAX_INPUT_SIZE];

  ssize_t res = read(STDIN_FILENO, str, MAX_INPUT_SIZE);

  // error checking
  if (res <= 0) {
    return NULL;
  }

  return str;
}
// assuming this is how the function is called
char* result = read_stdin();
```

Poll Everywhere

pollev.com/cis4480

- Two things are wrong with this function. What are they?
- How do we fix this function w/o changing the function signature?

```c
#define MAX_INPUT_SIZE 100

char* read_stdin() {
  char str[MAX_INPUT_SIZE];

  ssize_t res = read(STDIN_FILENO, str, MAX_INPUT_SIZE);

  // error checking
  if (res <= 0) {
    return NULL;
  }

  return str;
}
// assuming this is how the function is called
char* result = read_stdin();
```

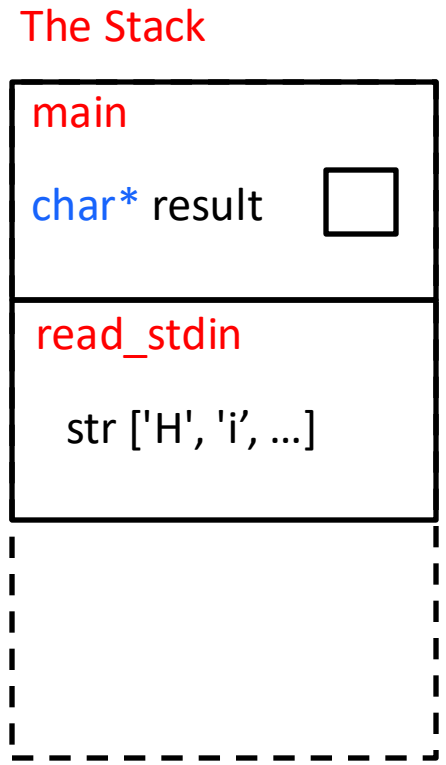


Poll Everywhere

pollev.com/cis4480

- Two things are wrong with this function. What are they?
- How do we fix this function w/o changing the function signature?

```c
#define MAX_INPUT_SIZE 100

char* read_stdin() {
  char str[MAX_INPUT_SIZE];

  ssize_t res = read(STDIN_FILENO, str, MAX_INPUT_SIZE);

  // error checking
  if (res <= 0) {
    return NULL;
  }

  return str;
}
// assuming this is how the function is called
char* result = read_stdin();
```

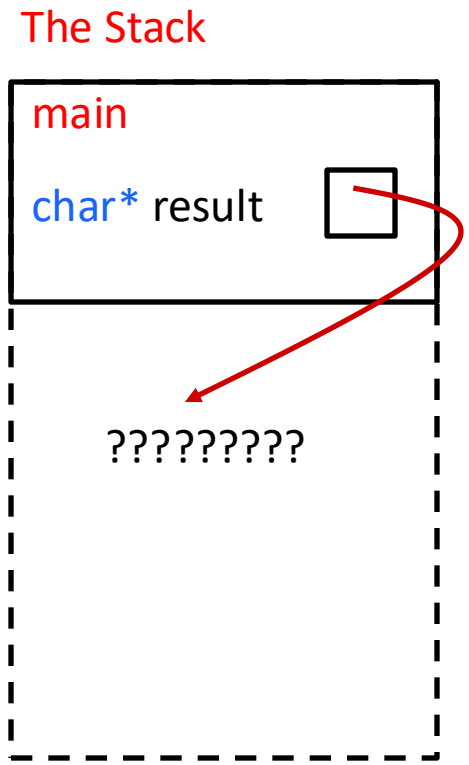


Poll Everywhere

pollev.com/cis4480

- Two things are wrong with this function. What are they?
- How do we fix this function w/o changing the function signature?

```c
#define MAX_INPUT_SIZE 100

char* read_stdin() {
  char str[MAX_INPUT_SIZE];

  ssize_t res = read(STDIN_FILENO, str, MAX_INPUT_SIZE);

  // error checking
  if (res <= 0) {
    return NULL;
  }

  return str;
}
// assuming this is how the function is called
char* result = read_stdin();
```



# Memory Allocation

So far, we have seen two kinds of memory allocation:

```c
int counter = 0;    // global var

int main() {
  counter++;
  printf("count = %d\n",counter);
  return 0;
}
```

- `counter` is statically-allocated
  - Allocated when program is loaded
  - Deallocated when program exits

```c
int foo(int a) {
  int x = a + 1;      // local var
  return x;
}

int main() {
  int y = foo(10);    // local var
  printf("y = %d\n",y);
  return 0;
}
```

- `a`, `x`, `y` are automatically-allocated
  - Allocated when function is called
  - Deallocated when function returns

# Aside: `sizeof`

- **`sizeof`** operator can be applied to a variable or a type and it evaluates to the size of that type in bytes
- Examples:
  - **`sizeof(int)`** – returns the size of an integer
  - **`sizeof(double)`** – returns the size of a double precision number
  - **`struct my_struct s;`**
    - **`sizeof(s)`** – returns the size of the struct s
  - **`my_type *ptr`**
    - **`sizeof (*ptr)`** – returns the size of the type pointed to by ptr

- Very useful for Dynamic Memory

# What is Dynamic Memory Allocation?

- We want Dynamic Memory Allocation
  - Dynamic means “at run-time”
  - The compiler and the programmer don’t have enough information to make a final decision on how much to allocate
  - Your program explicitly requests more memory at run time
  - The language allocates it at runtime, maybe with help of the OS

- Dynamically allocated memory persists until either:
  - A garbage collector collects it (automatic memory management)
  - Your code explicitly deallocates it (manual memory management)

- C requires you to manually manage memory
  - More control, and more headaches

# Heap API

- Dynamic memory is managed in a location in memory called the "Heap"
  - The heap is managed by user-level runetime library (libc)
  - Interface functions found in `<stdlib.h>`
- Most used functions:
  - **`void *malloc(size_t size);`**
    - Allocates memory of specified size
  - **`void free(void *ptr);`**
    - Deallocates memory
- Note: **`void*`** is “generic pointer”. It holds an address, but doesn’t specify what it is pointing at.
- Note 2: **`size_t`** is the integer type of **`sizeof()`**

# malloc()

```
void *malloc(size_t size);
```

**malloc** allocates a block of memory of the requested size

- Returns a pointer to the first byte of that memory
  - And returns NULL if the memory allocation failed!
- You should assume that the memory initially contains garbage
- You'll typically use sizeof to calculate the size you need

```
// allocate a 10-float array
float* arr = malloc(10*sizeof(float));
if (arr == NULL) {
  return errcode;
}
...   // do stuff with arr
```

ALWAYS CHECK FOR NULL

# free()

```
free(pointer);
```

Deallocates the memory pointed-to by the pointer

- Pointer *must* point to the first byte of heap-allocated memory
  - (*i.e.* something previously returned by **malloc**)
- Freed memory becomes eligible for future allocation
- The bits in the pointer are *not changed* by calling free
  - Defensive programming: can set pointer to NULL after freeing it

```
float* arr = malloc(10*sizeof(float));
if (arr == NULL)
  return errcode;
...             // do stuff with arr
free(arr);
arr = NULL;     // Preferred
```

```
free(NULL);
```

p.s. This is a No-Op.

# The Heap

- The Heap is a large pool of available memory to use for Dynamic allocation
- This pool of memory is kept track of with a small data structure indicating which portions have been allocated, and which portions are currently available.
- **`malloc`**:
  - searches for a large enough unused block of memory
  - marks the memory as allocated.
  - Returns a pointer to the beginning of that memory
- **`free`**:
  - Takes in a pointer to a previously allocated address
  - Marks the memory as free to use.

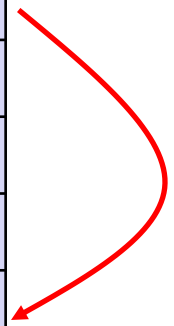
# Dynamic Memory Example

```c
#include <stdlib.h>

int main() {
  char* ptr = malloc(4*sizeof(char));
  if (ptr == NULL)
    return EXIT_FAILURE;
  ...              // do stuff with ptr
  free(ptr);
}
```

| addr | var | value |
|---|---|---|
| 0x2001 | **ptr** | -- |
| ... | **...** | -- |
| 0x4000 | **HEAP START** | USED |
| 0x4001 | | USED |
| 0x4002 | | Free |
| 0x4003 | | Free |
| 0x4004 | | Free |
| 0x4005 | | Free |
| 0x4006 | | Free |
| 0x4007 | | Free |
| 0x4008 | | USED |
| 0x4009 | | USED |

# Dynamic Memory Example

```c
#include <stdlib.h>

int main() {
  char* ptr = malloc(4*sizeof(char));
  if (ptr == NULL)
    return EXIT_FAILURE;
  ...              // do stuff with ptr
  free(ptr);
}
```



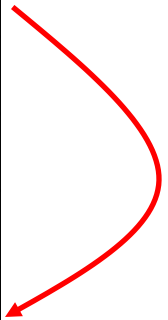
| addr | var | value |
|---|---|---|
| 0x2001 | ptr | -- |
| ... | ... | -- |
| 0x4000 | HEAP START | USED |
| 0x4001 | | USED |
| 0x4002 | h | USED |
| 0x4003 | e | USED |
| 0x4004 | y | USED |
| 0x4005 | \0 | USED |
| 0x4006 | | Free |
| 0x4007 | | Free |
| 0x4008 | | USED |
| 0x4009 | | USED |

# Dynamic Memory Example

```c
#include <stdlib.h>

int main() {
  char* ptr = malloc(4*sizeof(char));
  if (ptr == NULL)
    return EXIT_FAILURE;
  ...              // do stuff with ptr
  free(ptr);
}
```



| addr | var | value |
|---|---|---|
| 0x2001 | ptr | -- |
| ... | ... | -- |
| 0x4000 | HEAP START | USED |
| 0x4001 | | USED |
| 0x4002 | h | Free |
| 0x4003 | e | Free |
| 0x4004 | y | Free |
| 0x4005 | \0 | Free |
| 0x4006 | | Free |
| 0x4007 | | Free |
| 0x4008 | | USED |
| 0x4009 | | USED |



# Partially Fixed read_stdin()

```c
#define MAX_INPUT_SIZE 100

char* read_stdin() {
  char str = (char*) malloc(sizeof(char) * MAX_INPUT_SIZE);
  if (str == NULL) {
    return NULL;
  }

  ssize_t res = read(STDIN_FILENO, str, MAX_INPUT_SIZE);

  // error checking
  if (res <= 0) {
    return NULL;
  }

  return str;
}
```

# Demo (continued): get_input.c

- Lets code together a small program that:
  - Reads at max 100 characters from stdin (user input)
  - Truncates the input to only the first word
  - Prints that word out
  - Not allowed to use scanf, FILE*, printf, etc

- What was the other issue? (other than not using malloc)

# Fully Fixed read_stdin()

```c
#define MAX_INPUT_SIZE 100

char* read_stdin() {
  char str = (char*) malloc(sizeof(char) * MAX_INPUT_SIZE);
  if (str == NULL) {
    return NULL;
  }

  ssize_t res = read(STDIN_FILENO, str, MAX_INPUT_SIZE);

  // error checking
  if (res <= 0) {
    return NULL;
  }
  str[res] = '\0';

  return str;
}
```

The Null-Terminator must be added manually.

Reminder: read is a very *exact function* in that it only writes exactly what there is to read.

Poll Everywhere

pollev.com/cis4480

- Does this function work as intended?

```
typedef struct point_st {
  float x;
  float y;
} Point;

Point make_point() {
  Point p = (Point) {
    .x = 2.0f;
    .y = 1.0f;
  };
  return p;
}
```

Poll Everywhere

pollev.com/cis4480

- Does this function work as intended?

```
typedef struct point_st {
  float x;
  float y;
} Point;

Point* make_point() {
  Point p = (Point) {
    .x = 2.0f;
    .y = 1.0f;
  };
  Point* ptr = &p;
  return res;
}
```

# Dynamic Memory Pitfalls

- Buffer Overflows
  - E.g. ask for 10 bytes, but write 11 bytes
  - Could overwrite information needed to manage the heap
  - Common when forgetting the null-terminator on malloc'd strings
- Not checking for **`NULL`**
  - Malloc returns NULL if out of memory
  - Should check this after every call to malloc
- Giving **`free()`** a pointer to the middle of an allocated region
  - Free won't recognize the block of memory and probably crash
- Giving free() a pointer that has already been freed
  - Will interfere with the management of the heap and likely crash
- **`malloc`** does NOT initialize memory
  - There are other functions like **`calloc`** that will zero out memory

# Memory Leaks

- The most common Memory Pitfall
- What happens if we malloc something, but don't free it?
  - That block of memory cannot be reallocated, even if we don't use it anymore, until it is **`free`**d
  - If this happens enough, we run out of heap space and program may slow down and eventually crash
- Garbage Collection
  - Automatically "frees" anything once the program has lost all references to it
  - Affects performance, but avoid memory leaks
  - Java has this, C doesn't

# Discuss: What is wrong with this code? (Multiple bugs)

```c
int main() {
  char* literal = "Hello!";
  char* duplicate = dup_str(literal);
  char* ptr = duplicate;

  while (*ptr != '\0') {
    printf("%s\n", ptr);
    // printf line is fine
    ptr += 1;
  }

  free(duplicate);
  free(ptr);
  free(literal);
}
```

```c
char* dup_str(char* to_copy) {
  size_t len = strlen(to_copy);
  char* res = malloc(sizeof(char) * len);
  for (size_t i = 0; i < len; i++) {
    res[i] = to_copy[i];
  }
  return res;
}
```

**strlen()**
**returns the number of characters before the null-terminator**

# `static` function variables

- Functions can declare a variable as static

```c
#include <stdio.h>  // for printf
#include <stdlib.h> // for EXIT_SUCCESS

int next_num();

int main(int argc, char** argv) {
  printf("%d\n", next_num()); // prints 1
  printf("%d\n", next_num()); // then 2
  printf("%d\n", next_num()); // then 3
  return EXIT_SUCCESS;
}

int next_num() {
  // marking this variable as static means that
  // the value is preserved between calls to the function
  // this allows the function to "remember" things
  static int counter = 0;
  counter++;
  return counter;
}
```

This is how some functions can "remember" things.

Can be thought of as a global variable that is "private" to a function

# Lecture Outline

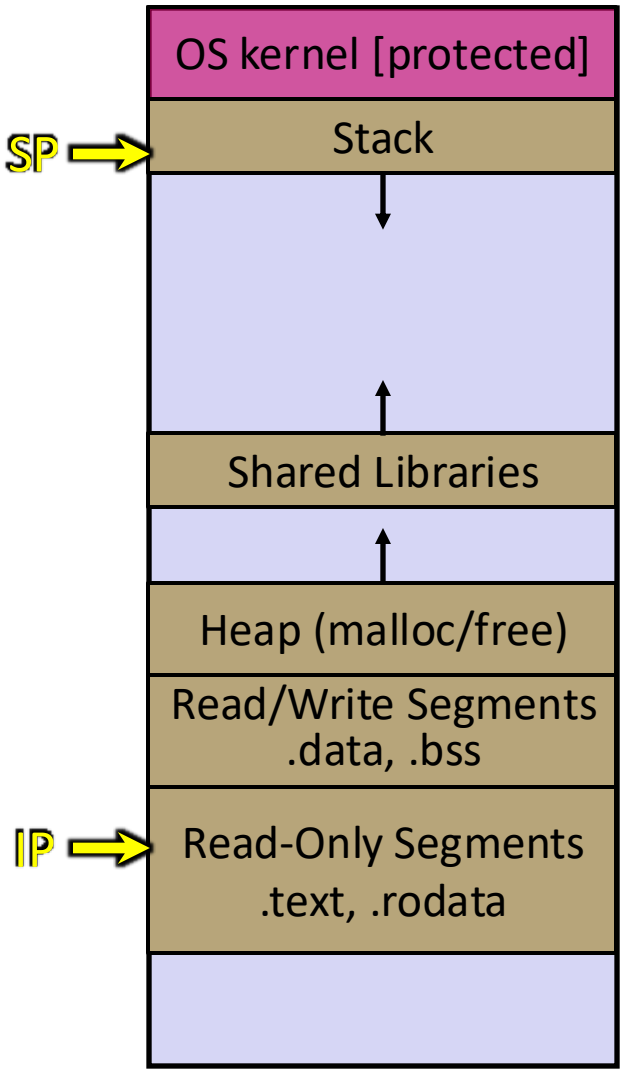
- C “Refresher”
  - Dynamic Memory vs the Stack
  - Structs
- **Processes**
  - **Overview**
  - **fork()**
  - exec()

# Definition: Process

Definition: An instance of a program that is being executed (or is ready for execution)

Consists of:

- Memory (code, heap, stack, etc)
- Registers used to manage execution (stack pointer, program counter, ...)
- Other resources

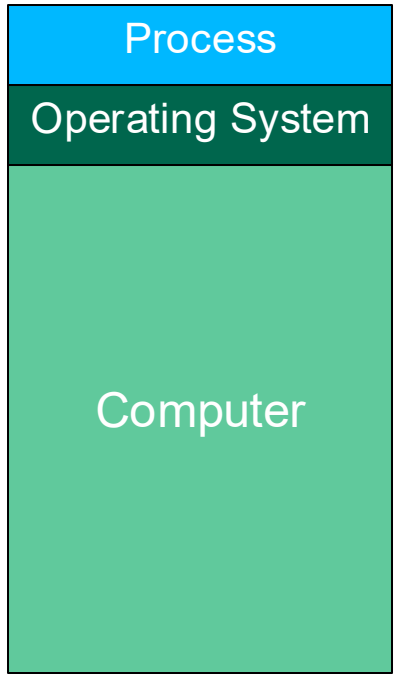



* This isn't quite true
more in a future lecture

# Computers as we know them now

- In CIS 2400, you learned about hardware, transistors, CMOS, gates, etc.
- Once we got to programming, our computer looks something like:

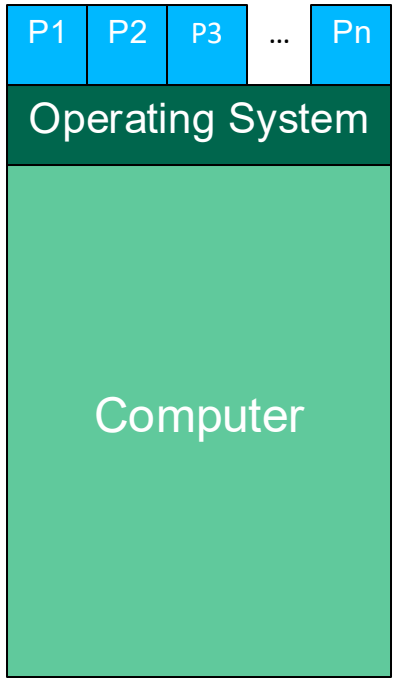
What is missing/wrong with this?




- This model is still useful, and can be used in many settings

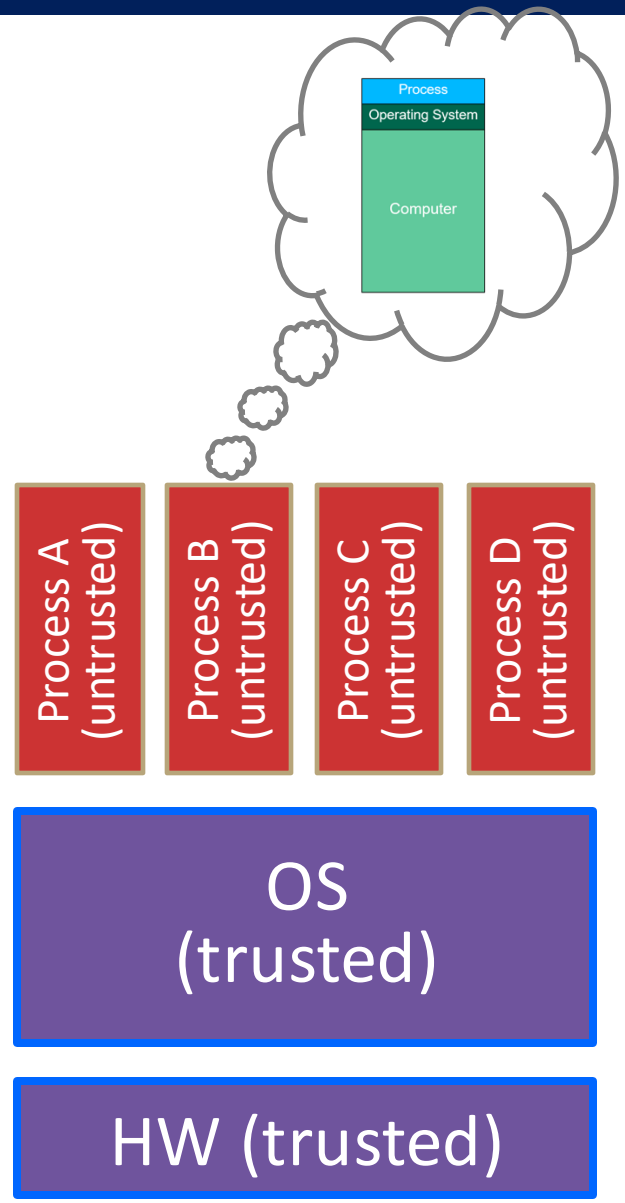
# Multiple Processes

- Computers run multiple processes “at the same time”
- One or more processes for each of the programs on your computer
- Each process has its own…
  - Memory space
  - Registers
  - Resources

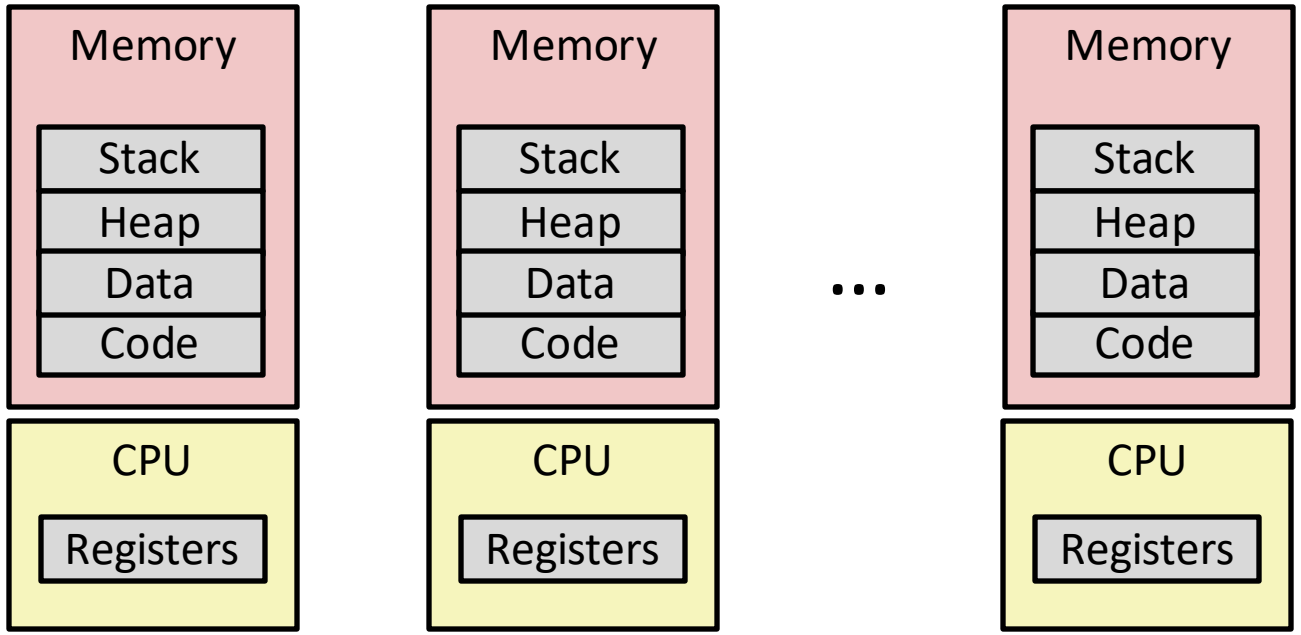


# OS: Protection System

- OS isolates process from each other
  - Each process seems to have exclusive use of memory and the processor.
    - This is an **illusion**
    - More on Memory when we talk about virtual memory later in the course
  - OS permits controlled sharing between processes
    - E.g. through files, the network, etc.
- OS isolates itself from processes
  - Must prevent processes from accessing the hardware directly

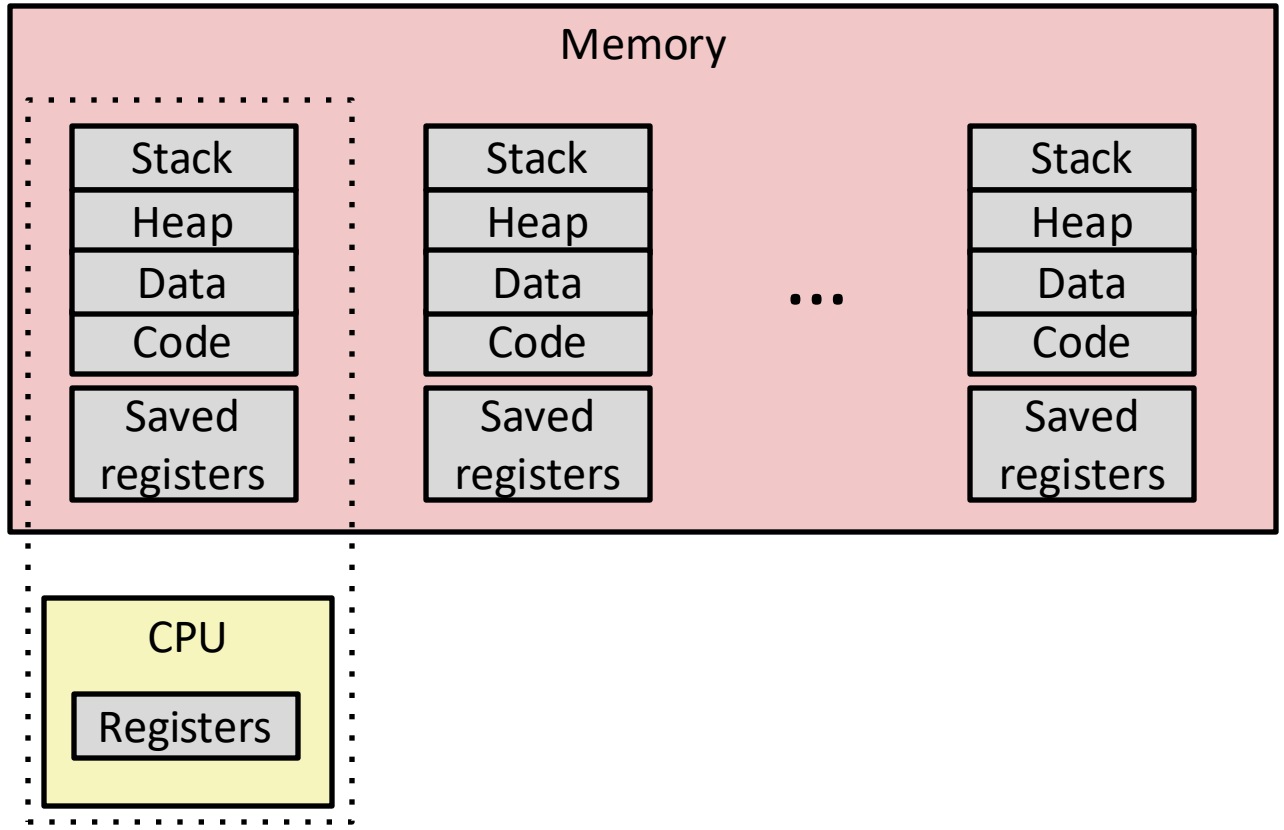


# Multiprocessing: The Illusion



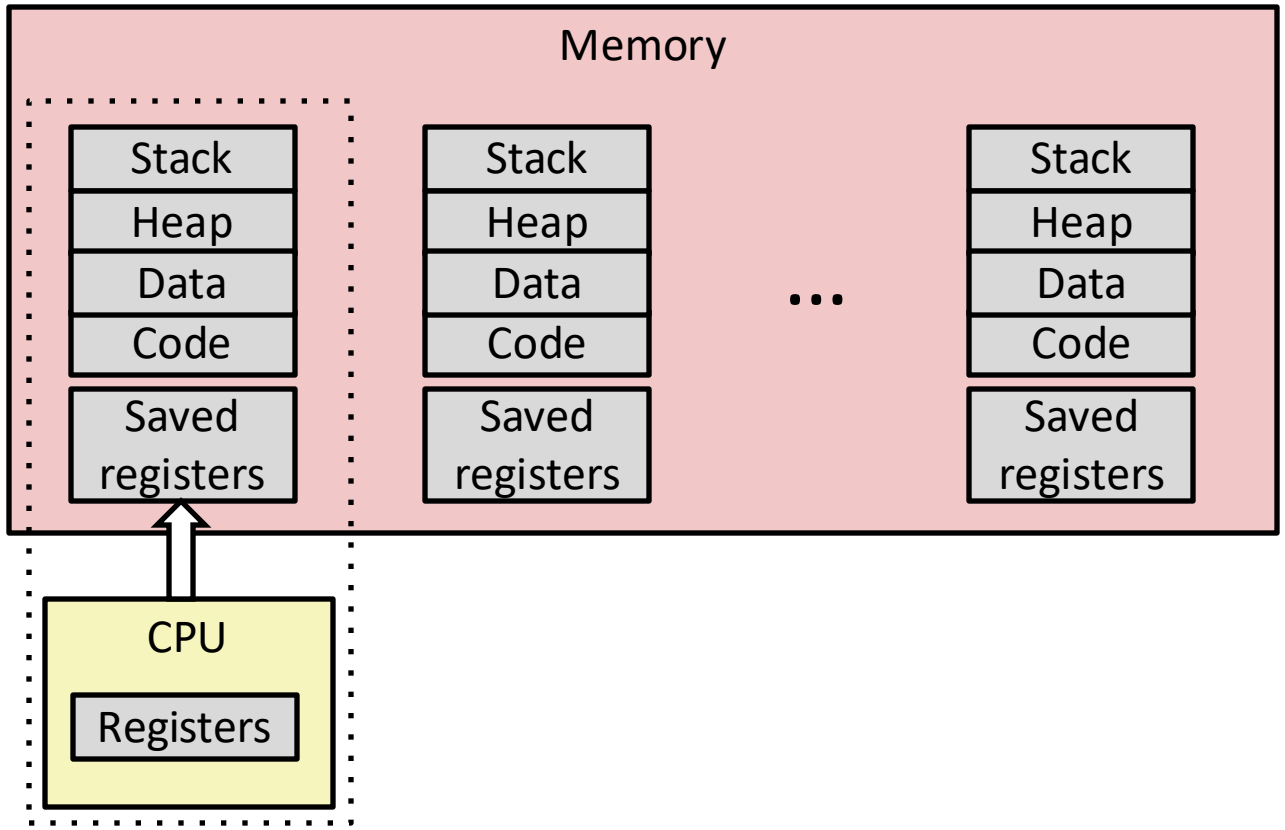

- Computer runs many processes simultaneously
  - Applications for one or more users
    - Web browsers, email clients, editors, …
  - Background tasks
    - Monitoring network & I/O devices

# Multiprocessing: The (Traditional) Reality



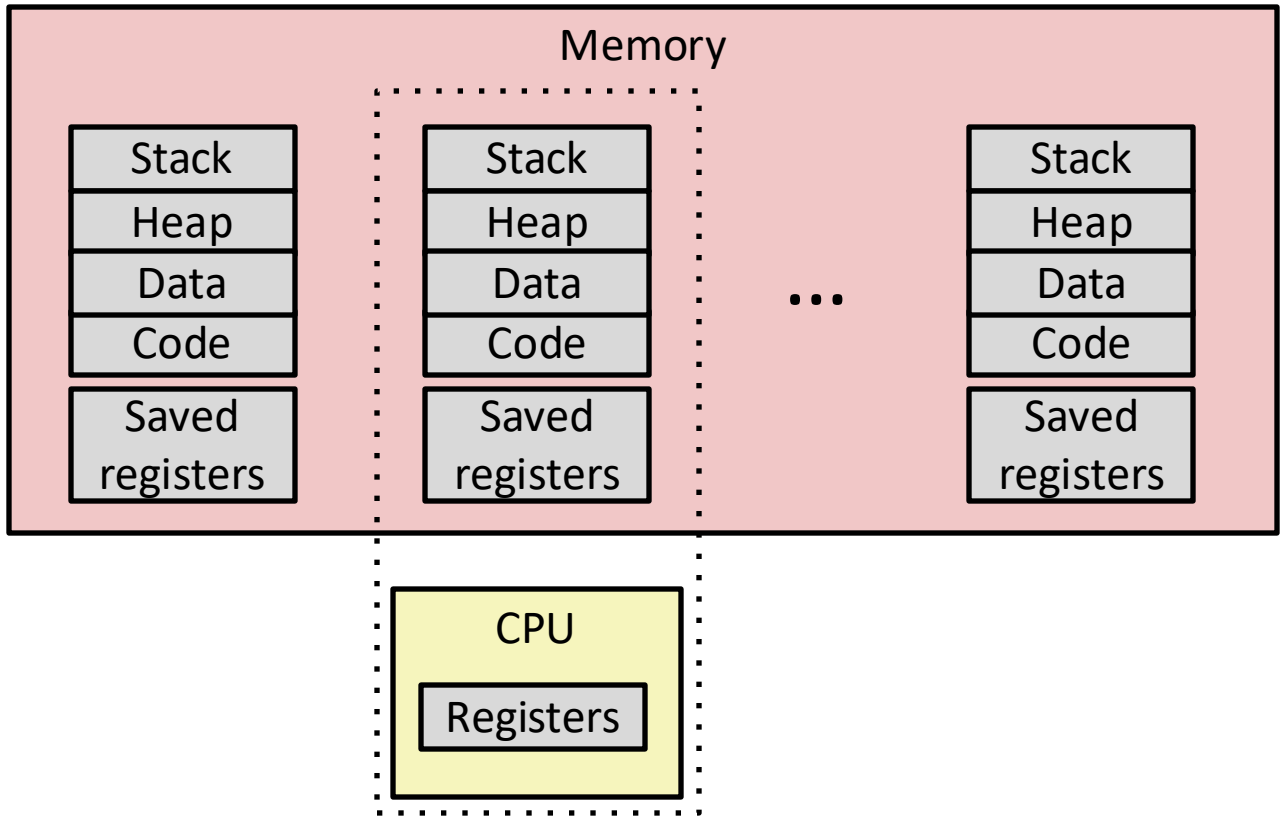

- Single processor executes multiple processes concurrently
  - Process executions interleaved (multitasking)
  - Address spaces managed by virtual memory system (later in course)
  - Register values for nonexecuting processes saved in memory

# Multiprocessing: The (Traditional) Reality



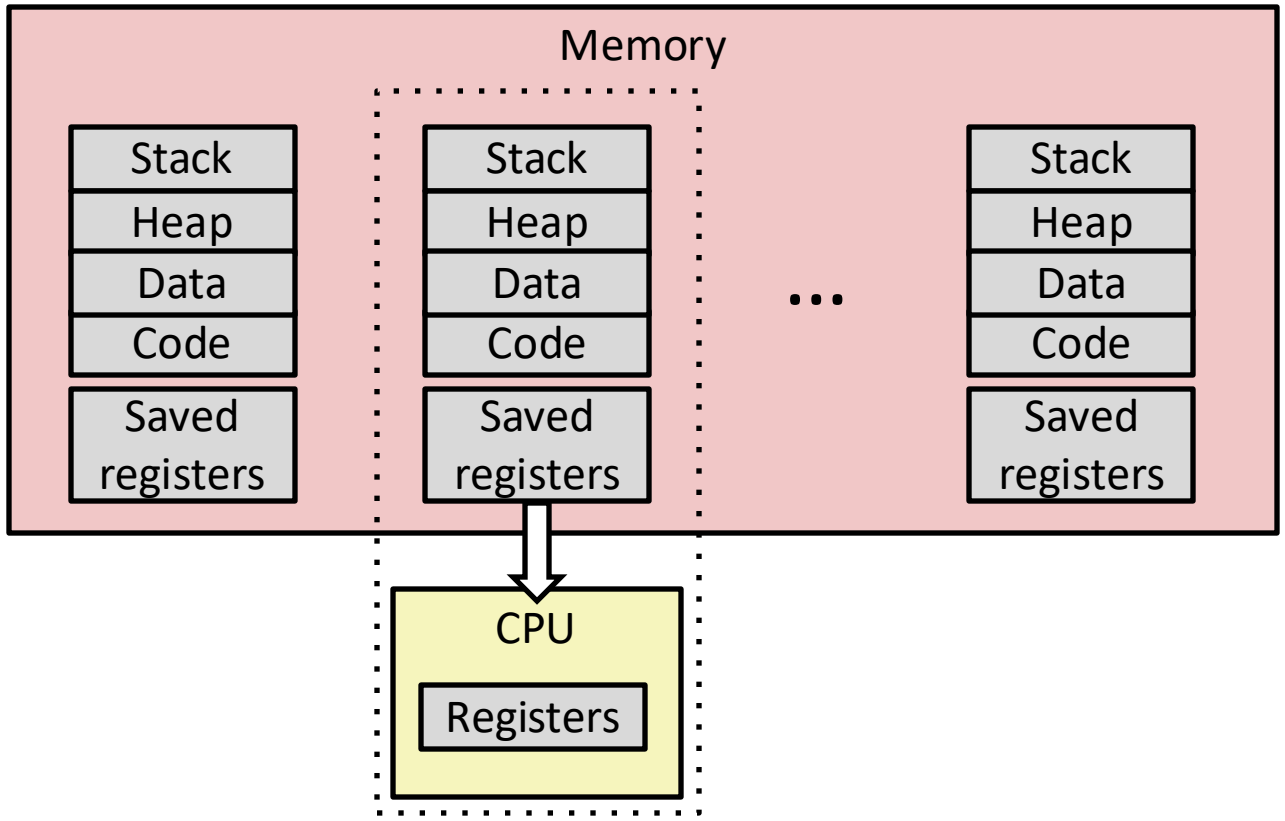

1. Save current registers in memory

# Multiprocessing: The (Traditional) Reality



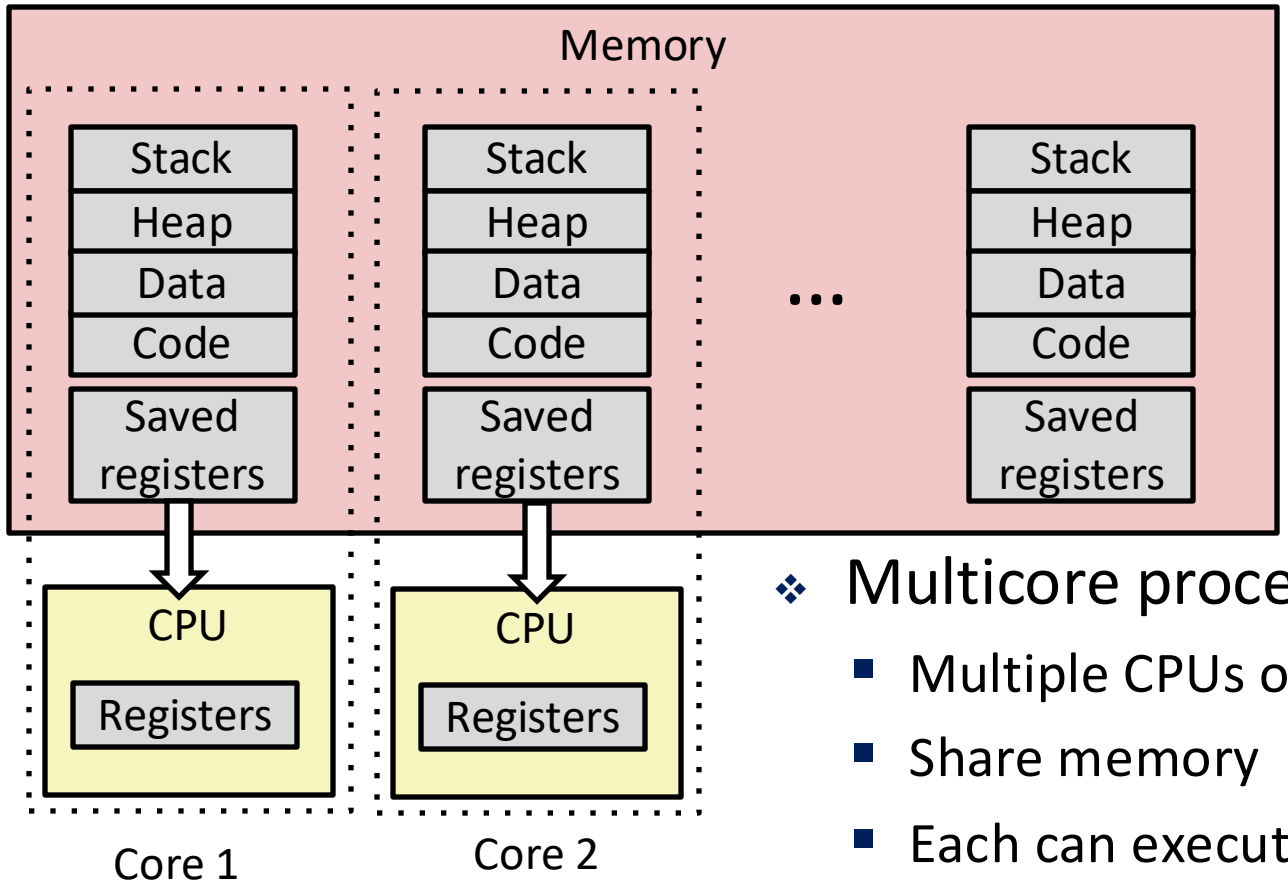

1. Save current registers in memory
2. Schedule next process for execution

# Multiprocessing: The (Traditional) Reality




1. Save current registers in memory
2. Schedule next process for execution
3. Load saved registers and switch address space ***(context switch)***

# Multiprocessing: The (Traditional) Reality




- Multicore processors
  - Multiple CPUs on single chip
  - Share memory
  - Each can execute a separate process
    - Scheduling of processors onto cores done by kernel
  - This is called “Parallelism”

Poll Everywhere

pollev.com/cis4480

- What I just went through was the big picture of processes. Many details left, some will be gone over in future lectures

- Any questions, comments or concerns so far?

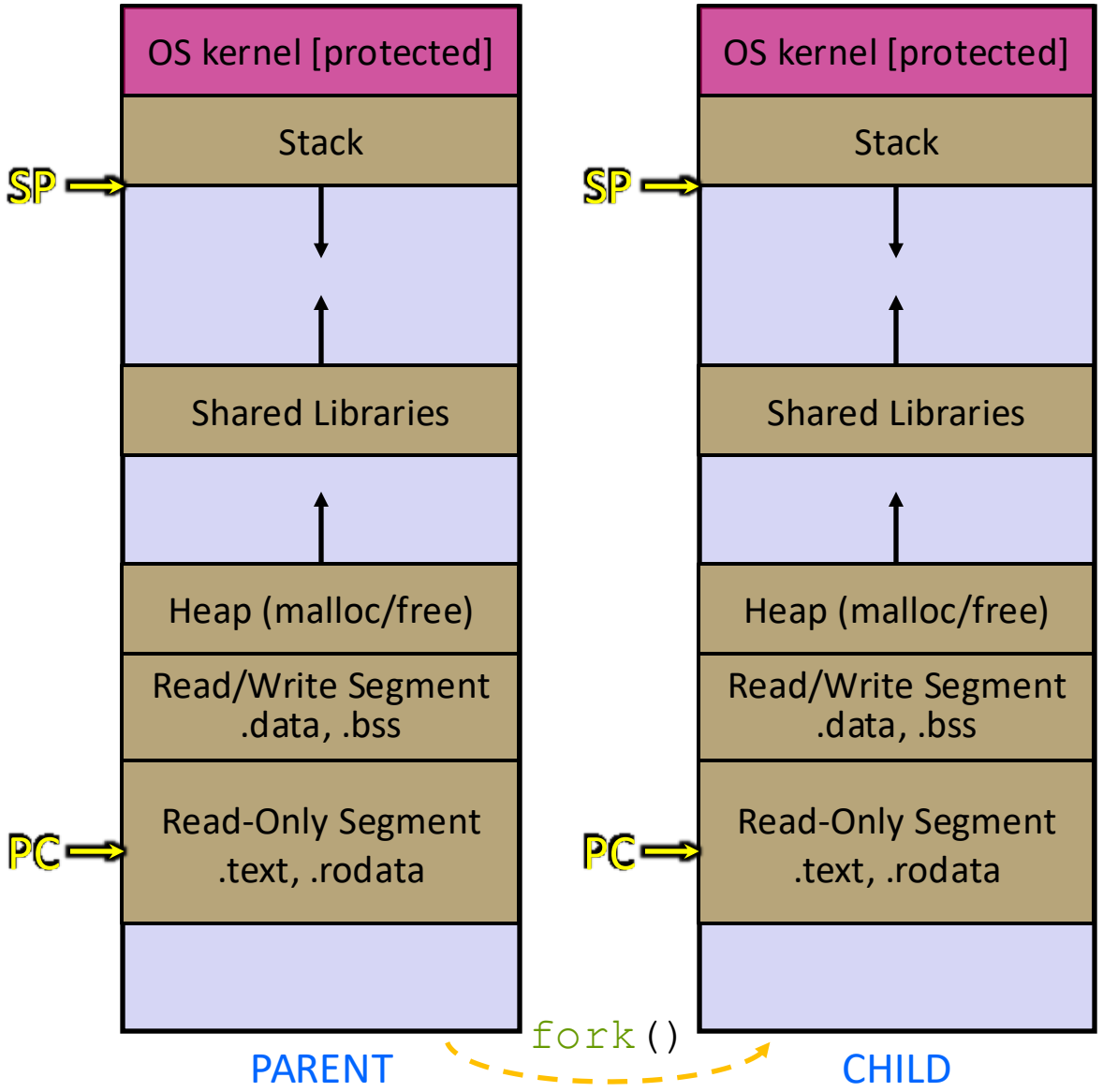
# Creating New Processes

```
pid_t fork();
```

- Creates a new process (the "child") that is an *exact clone** of the current process (the "parent")
  - *almost everything
- The new process has a separate virtual address space from the parent
- Returns a `pid_t` which is an integer type.

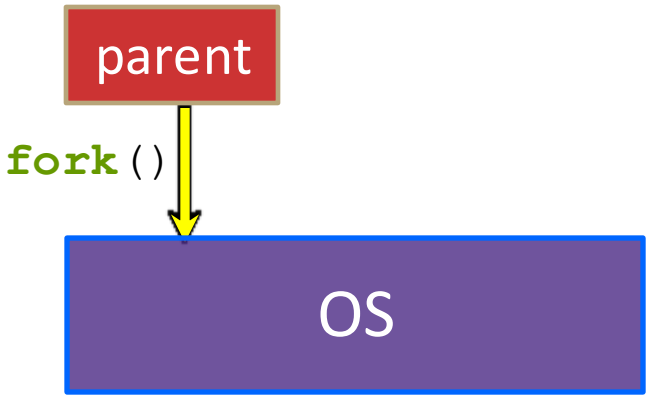
# `fork()` and Address Spaces

- Fork causes the OS to clone the address space
  - The *copies* of the memory segments are (nearly) identical
  - The new process has *copies* of the parent's data, stack-allocated variables, open file descriptors, etc.



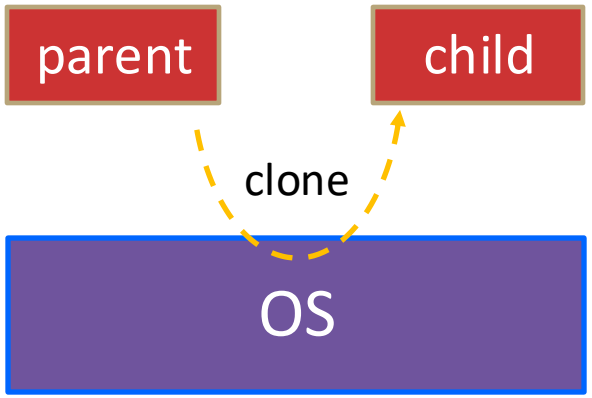
# fork()

- `fork()` has peculiar semantics
  - The parent invokes `fork()`
  - The OS clones the parent
  - *Both* the parent and the child return from fork
    - Parent receives child's pid
    - Child receives a 0



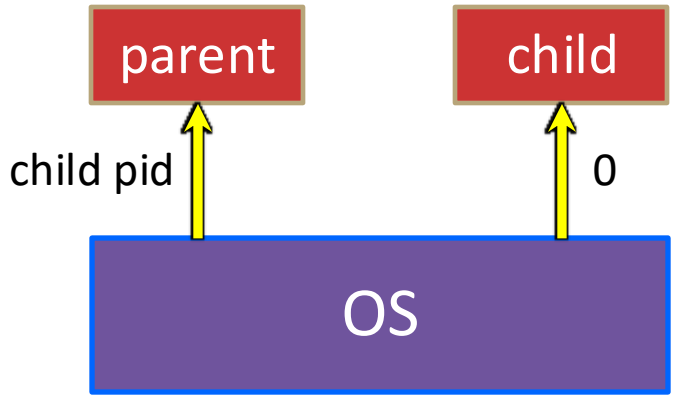
# `fork()`

- `fork()` has peculiar semantics
  - The parent invokes `fork()`
  - The OS clones the parent
  - *Both* the parent and the child return from fork
    - Parent receives child's pid
    - Child receives a 0



# `fork()`

- `fork()` has peculiar semantics
  - The parent invokes `fork()`
  - The OS clones the parent
  - *Both* the parent and the child return from fork
    - Parent receives child's pid
    - Child receives a 0



# "simple" `fork()` example

```
fork();
printf("Hello!\n");
```

- What does this print?

# "simple" `fork()` example

Parent Process (PID = X)

```
fork();
printf("Hello!\n");
```

Child Process (PID = Y)

```
fork();
printf("Hello!\n");
```

- What does this print?

- "Hello!\n" is printed twice

Poll Everywhere

pollev.com/cis4480

```
fork();
fork();
printf("Hello!\n");
```

- What does this print?

Poll Everywhere

pollev.com/cis4480

```c
int x = 3;
fork();
x++;
printf("%d\n", x);
```

- What does this print?

Poll Everywhere

pollev.com/cis4480

```c
pid_t fork_ret = fork();

if (fork_ret == 0) {
  printf("Child\n");
} else {
  printf("Parent\n");
}
```

- What does this print?

# fork() example

Parent Process (PID = X)

```
pid_t fork_ret = fork();

if (fork_ret == 0) {
  printf("Child\n");
} else {
  printf("Parent\n");
}
```

Child Process (PID = Y)

```
pid_t fork_ret = fork();

if (fork_ret == 0) {
  printf("Child\n");
} else {
  printf("Parent\n");
}
```

fork()

# fork() example

Parent Process (PID = X)

```
pid_t fork_ret = fork();

if (fork_ret == 0) {
  printf("Child\n");
} else {
  printf("Parent\n");
}
```

Child Process (PID = Y)

```
pid_t fork_ret = fork();

if (fork_ret == 0) {
  printf("Child\n");
} else {
  printf("Parent\n");
}
```

fork_ret = Y

```
pid_t fork_ret = fork();

if (fork_ret == 0) {
  printf("Child\n");
} else {
  printf("Parent\n");
}
```

Prints "Parent"

fork_ret = 0

```
pid_t fork_ret = fork();

if (fork_ret == 0) {
  printf("Child\n");
} else {
  printf("Parent\n");
}
```

Prints "Child"

Which prints first?

# Process States (incomplete)

**FOR NOW**, we can think of a process as being in one of three states:

*More states in future lectures*

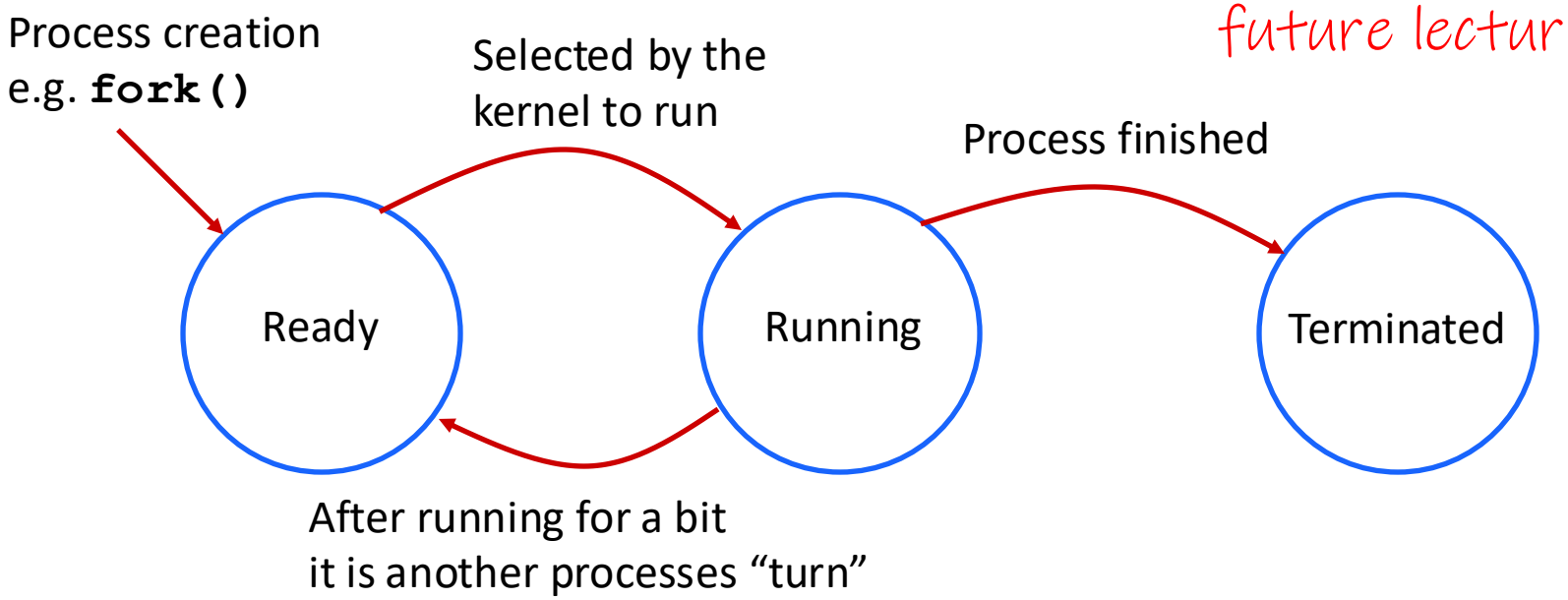
- Running
  - Process is currently executing
- Ready
  - Process is waiting to be executed and will eventually be *scheduled* (i.e., chosen to execute) by the kernel

*Scheduler to be covered in a later lecture*

- Terminated
  - Process is stopped permanently

# Process State Lifetime (incomplete)

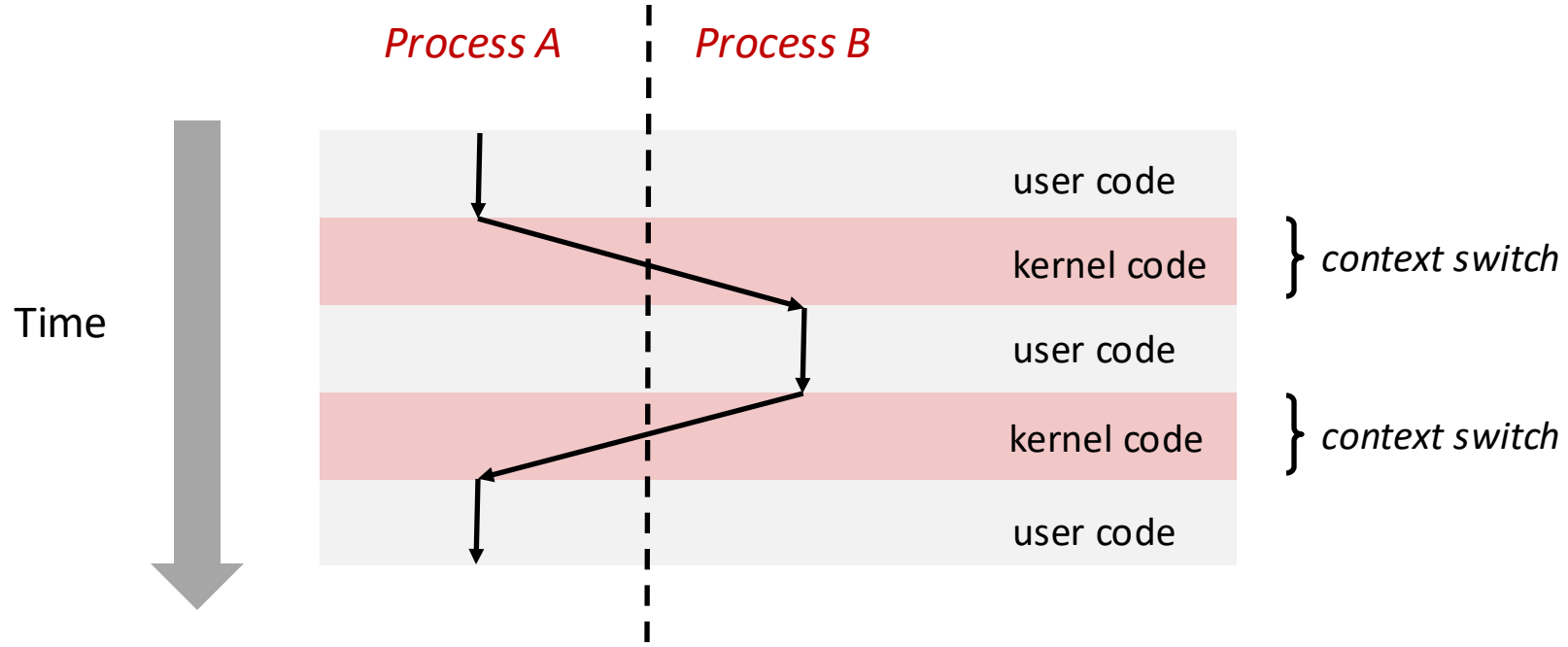
More states in future lectures




Processes can be "interrupted" to stop running. Through something like a hardware timer interrupt

# 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*



# OS: The Scheduler

- When switching between processes, the OS will run some kernel code called the “Scheduler”
- The scheduler runs when a process:
  - starts (“arrives to be scheduled”),
  - Finishes
  - Blocks (e.g., waiting on something, usually some form of I/O)
  - Has run for a certain amount of time

- It is responsible for scheduling processes
  - Choosing which one to run
  - Deciding how long to run it

# Scheduler Considerations

- The scheduler has a scheduling algorithm to decide what runs next.
- Algorithms are designed to consider many factors:
  - Fairness: Every program gets to run
  - Liveness: That "something" will eventually happen
  - Throughput: Number of "tasks" completed over an interval of time
  - Wait time: Average time a "task" is "alive" but not running
  - A lot more...

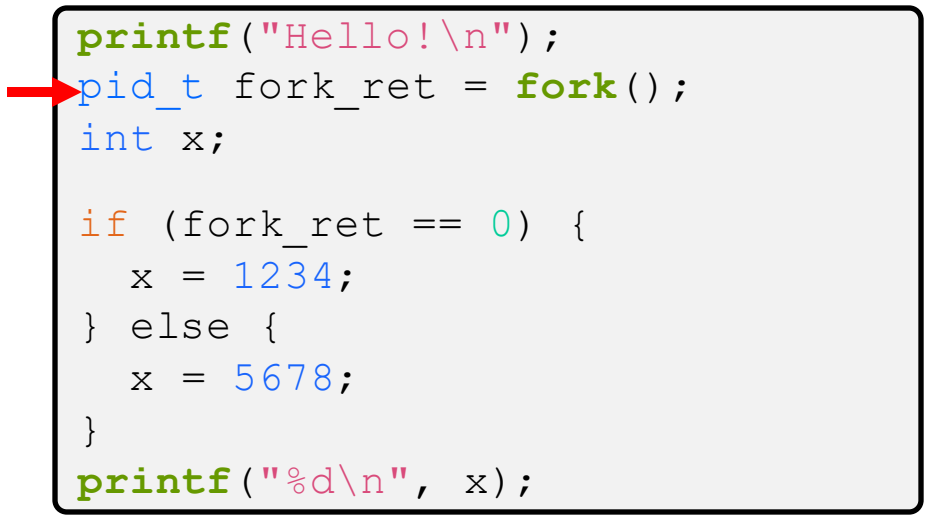
- More on this later. **For now: think of scheduling as non-deterministic**, details handled by the OS.

# fork() example

```
printf("Hello!\n");
pid_t fork_ret = fork();
int x;

if (fork_ret == 0) {
  x = 1234;
} else {
  x = 5678;
}
printf("%d\n", x);
```

Always prints "Hello"

# fork() example



```c
printf("Hello!\n");
pid_t fork_ret = fork();
int x;

if (fork_ret == 0) {
  x = 1234;
} else {
  x = 5678;
}
printf("%d\n", x);
```

Always prints "Hello"

# fork() example

Parent Process (PID = X)

```
printf("Hello!\n");
pid_t fork_ret = fork();
int x;

if (fork_ret == 0) {
  x = 1234;
} else {
  x = 5678;
}
printf("%d\n", x);
```

Child Process (PID = Y)

```
printf("Hello!\n");
pid_t fork_ret = fork();
int x;

if (fork_ret == 0) {
  x = 1234;
} else {
  x = 5678;
}
printf("%d\n", x);
```


Always prints "Hello"

Does NOT print "Hello"

# fork() example

Parent Process (PID = X)

```
printf("Hello!\n");
pid_t fork_ret = fork();
int x;

if (fork_ret == 0) {
  x = 1234;
} else {
  x = 5678;
}
printf("%d\n", x);
```

Child Process (PID = Y)

```
printf("Hello!\n");
pid_t fork_ret = fork();
int x;

if (fork_ret == 0) {
  x = 1234;
} else {
  x = 5678;
}
printf("%d\n", x);
```

fork_ret = Y

fork()

fork_ret = 0

Always prints "Hello"

Always prints "5678"

Always prints "1234"

# Exiting a Process

```
void exit(int status);
```

- Causes the current process to exit normally
- Automatically called by `main()` when main returns
- Exits with a return status (e.g. EXIT_SUCCESS or EXIT_FAILURE)
  - This is the same int returned by `main()`
- The exit status is accessible by the parent process with `wait()` or `waitpid()`.

Poll Everywhere

pollev.com/cis4480

```c
int global_num = 1;

void function() {
  global_num++;
  printf("%d\n", global_num);
}
int main() {
  pid_t id = fork();

  if (id == 0) {
    function();
    id = fork();
    if (id == 0) {
      function();
    }
    return EXIT_SUCCESS;
  }

  global_num += 2;
  printf("%d\n", global_num);
  return EXIT_SUCCESS;
}
```

- How many numbers are printed?
- What number(s) get printed from each process?

Poll Everywhere

pollev.com/cis4480

- How many times is ":)" printed?

```
int main(int argc, char* argv[]) {
  for (int i = 0; i < 4; i++) {
    fork();
  }

  printf(":)\n"); // "\n" is similar to endl
  return EXIT_SUCCESS;
}
```

# Processes & Fork Summary

- Processes are instances of programs that:
  - Each have their own independent ***address space (more on that later!)***
  - Each process is scheduled by the OS
    - Without using some functions we have not talked about (yet), there is no way to guarantee the order processes are executed
  - Processes are created by fork() system call
    - Only difference between processes is their process id and the return value from fork() each process gets

# Lecture Outline

- C “Refresher”
  - Dynamic Memory vs the Stack
  - Structs
- **Processes**
  - Overview
  - fork()
  - **exec()**

# exec*()

- Loads in a new program for execution

- Program Counter (rip in x86, pc in risc-v/arm) , SP, registers, and memory are all reset so that the specified program can run

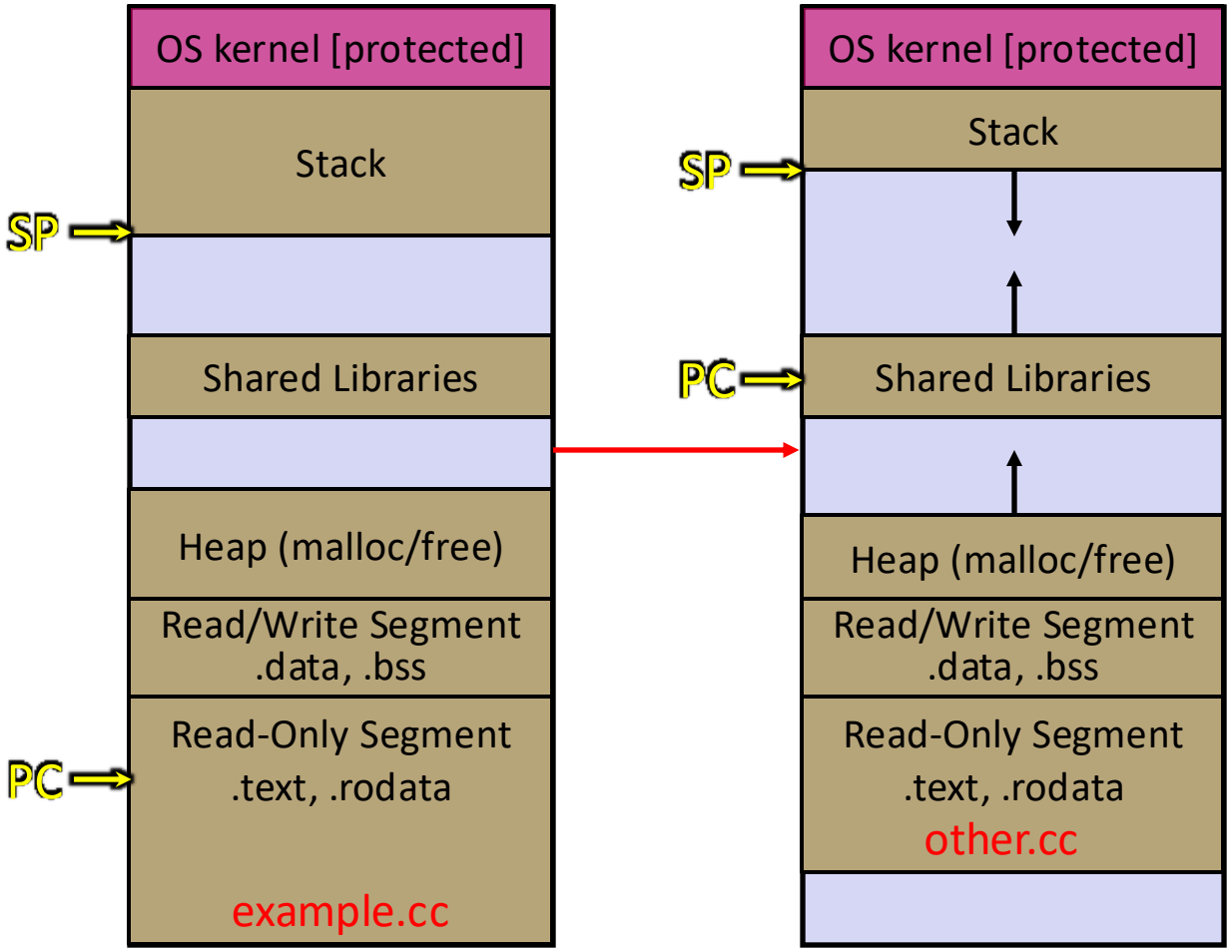
# execve()

```
int execve(const char *file,
           char* const argv[]
           char* const envp[]);
```

- Duplicates the action of the shell (terminal) in terms of finding the command/program to run
- Argv is an array of **`char*`**, the same kind of argv that is passed to `main()` in a C program
  - **`argv[0]`** MUST have the same contents as the file parameter
  - **`argv`** must have NULL as the last entry of the array
- Just pass in an array of `{ NULL };` as envp
- Returns -1 on error. Does NOT return on success

# Exec Visualization

- Exec takes a process and discards or “resets” most of it




NOTE that the following DO change
- The stack
- The heap
- Globals
- Loaded code
- Registers

NOTE that the following do NOT change
- Process ID
- Open files
- The kernel

# Aside: Exiting a Process

```
void exit(int status);
```

- Causes the current process to exit normally
- Automatically called by `main()` when main returns
- Exits with a return status (e.g. EXIT_SUCCESS or EXIT_FAILURE)
  - This is the same int returned by `main()`
- The exit status is accessible by the parent process with `wait()` or `waitpid()`. (more on these functions next lecture)

# Exec Demo

- See `exec_example.c`
  - Brief code demo to see how exec works
  - What happens when we call exec?
  - What happens to allocated memory when we call exec?

Poll Everywhere

pollev.com/cis4480

```c
int main(int argc, char* argv[]) {
  char* envp[] = { NULL };
  // fork a process to exec clang
  pid_t clang_pid = fork();

  if (clang_pid == 0) {
    // we are the child
    char* clang_argv[] = {"/bin/clang", "-o",
              "hello","hello_world.c", NULL};
    execve(clang_argv[0], clang_argv, envp);
    exit(EXIT_FAILURE);
  }

  // fork to run the compiled program
  pid_t hello_pid = fork();
  if (hello_pid == 0) {
    // the process created by fork
    char* hello_argv[] = {"./hello", NULL};
    execve(hello_argv[0], hello_argv, envp);
    exit(EXIT_FAILURE);
  }
  return EXIT_SUCCESS;
}
```

This code is broken. It compiles, but it doesn't do what we want. It is trying to compile some code and then run it.

Why is this broken?

- Clang is a C compiler
- Assume exec'ing the compiler works (hello_world.c compiles correctly)
- Assume I gave the correct args to exec in both cases

broken_autograder.c