C (cont.) & Intro to Processes

Computer Operating Systems, Spring 2024

Instructor: Travis McGaha

Head TAs: Nate Hoaglund & Seungmin Han

TAs:

Adam Gorka	Haoyun Qin	Kyrie Dowling	Ryoma Harris
Andy Jiang	Jeff Yang	Oliver Hendrych	Shyam Mehta
Charis Gao	Jerry Wang	Maxi Liu	Tom Holland
Daniel Da	Jinghao Zhang	Rohan Verma	Tina Kokoshvili
Emily Shen	Julius Snipes	Ryan Boyle	Zhiyan Lu

Poll Everywhere

pollev.com/tqm

- How are you?

Administrivia

- Project 0 penn-parser:
  - out later tonight
  - "due" Tuesday Jan 30
  - Actual due date: submit with penn-shredder, but you need to finish it before penn-shredder will work anyways.
  - Your first C programming assignment

- Pre-semester survey:
  - out at 7pm
  - "due" wed Jan 31
  - Just a short survey

Lecture Outline

- **C Refresher**
  - **C Strings**
  - Dynamic memory (malloc & realloc)
  - Structs
- Processes
  - Overview
  - fork()
  - exec()

Strings without Objects

- Strings are central to C, very important for I/O
- In C, we don't have Objects but we need strings
- If a string is just a sequence of characters, we can have use array of characters as a string

- Example:

```
char str_arr[] = "Hello World!";
char *str_ptr = "Hello World!";
```

Null Termination

DO NOT FORGET THIS. THIS IS THE CAUSE OF MANY BUGS

- Arrays don't have a length, but we **mark the end of a string with the null terminator character.**
  - The null terminator has value `0x00` or `'\0'`
  - Well formed strings ***MUST*** be null terminated
- Example: `char str[] = "Hello";`
  - Takes up 6 characters, 5 for "Hello" and 1 for the null terminator

address	0x2000	0x2001	0x2002	0x2003	0x2004	0x2005
value	'H'	'e'	'l'	'l'	'o'	'\0'

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/tqm

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

```
#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;
}
```

Poll Everywhere

pollev.com/tqm

- There are two things wrong with this function
- What are they? How do we fix this function w/o changing the function sig?

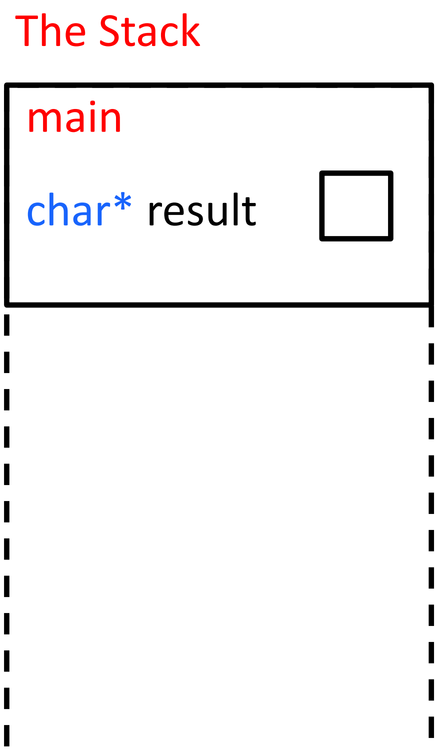
```
#define MAX_INPUT_SIZE 100

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/tqm

- There are two things wrong with this function
- What are they? How do we fix this function w/o changing the function sig?




```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;
}
```

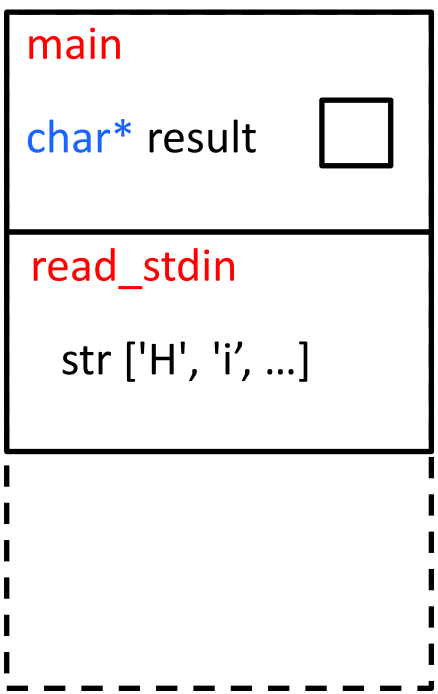
```c
// assuming this is how the function is called
char* result = read_stdin();
```

Poll Everywhere

pollev.com/tqm

- There are two things wrong with this function
- What are they? How do we fix this function w/o changing the function sig?

The Stack



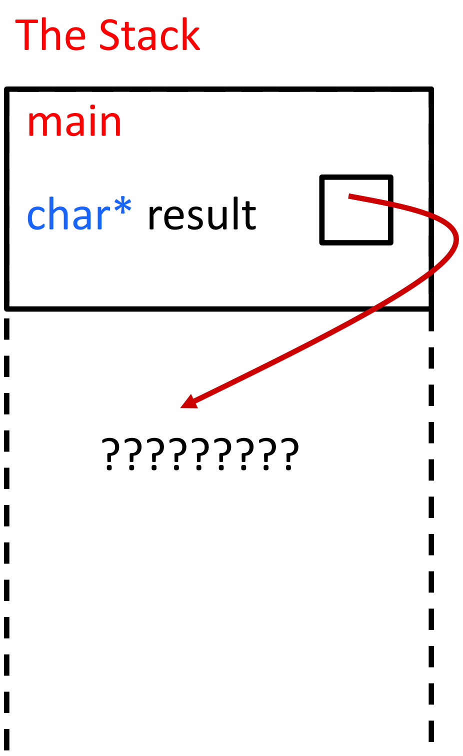

```c
#define MAX_INPUT_SIZE 100

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;
}
```

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

Poll Everywhere

pollev.com/tqm

- There are two things wrong with this function
- What are they? How do we fix this function w/o changing the function sig?




```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;
}
```

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

Lecture Outline

- **C Refresher**
  - C Strings
  - **Dynamic memory (malloc & realloc)**
  - Structs
- Processes
  - Overview
  - fork()
  - exec()

Memory Allocation

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

```
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

```
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()

- Usage: `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
  - `free(NULL);` does nothing.
  - 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;     // OPTIONAL
```

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		
0x4003		
0x4004		
0x4005		
0x4006		
0x4007		
0x4008		USED
0x4009		USED

Dynamic Memory Example

```
#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	0x4002
...	...	--
0x4000	HEAP START	USED
0x4001		USED
0x4002		**USED**
0x4003		**USED**
0x4004		**USED**
0x4005		**USED**
0x4006		
0x4007		
0x4008		USED
0x4009		USED

Dynamic Memory Example

```
#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	0x4002
...	...	--
0x4000	HEAP START	USED
0x4001		USED
0x4002		
0x4003		
0x4004		
0x4005		
0x4006		
0x4007		
0x4008		USED
0x4009		USED

Fixed read_stdin()

```
#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)

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

`static` function variables

- Functions can declare a variable as static

```
#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 (like one in proj0) can "remember" things.

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

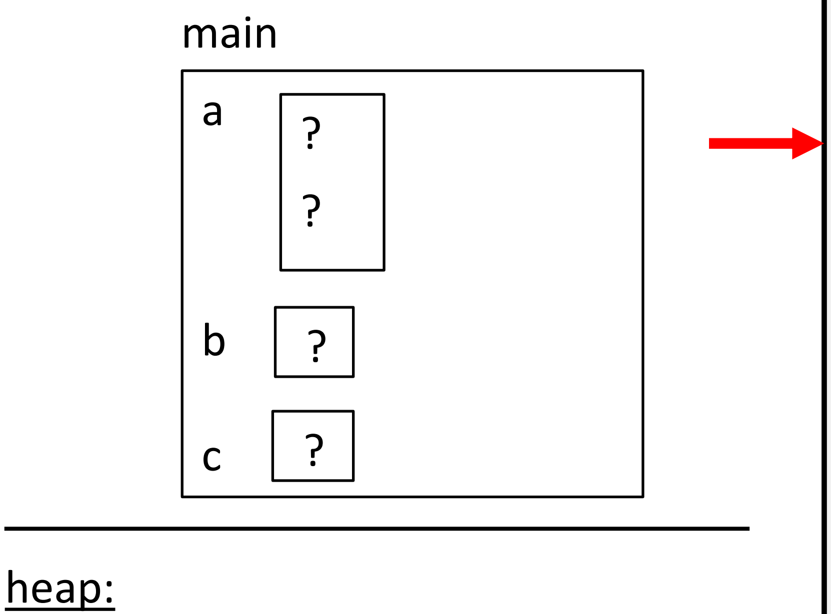
Poll Everywhere

pollev.com/tqm

- Which line below is first to (most likely) cause a crash?
  - Yes, there are a lot of bugs, but not all cause a crash ☺
  - See if you can find all the bugs!

```
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char** argv) {
  int a[2];
  int* b = malloc(2*sizeof(int));
  int* c;

1 a[2] = 5;
2 b[0] += 2;
3 c = b+3;
4 free(&(a[0]));
5 free(b);
6 free(b);
7 b[0] = 5;

  return 0;
}
```

Memory Corruption - What Happens?

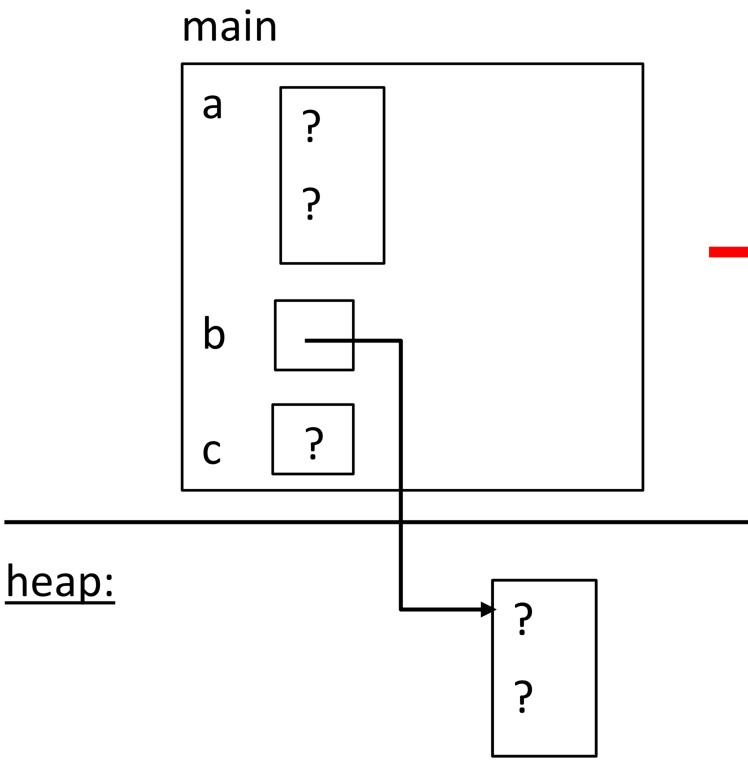



```c
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char** argv) {
  int a[2];
  int* b = malloc(2*sizeof(int));
  int* c;

  a[2] = 5;    // assigns past the end of an array
  b[0] += 2;   // assumes malloc zeros out memory
  c = b+3;     // Ok, but if we use c, problem
  free(&(a[0]));  // free something not malloc'ed
  free(b);
  free(b);     // double-free the same block
  b[0] = 5;    // use a freed (dangling) pointer

  // any many more!
  return 0;
}
```

Note: Arrow points to *next* instruction.

memcorrupt.c

Memory Corruption - What Happens?

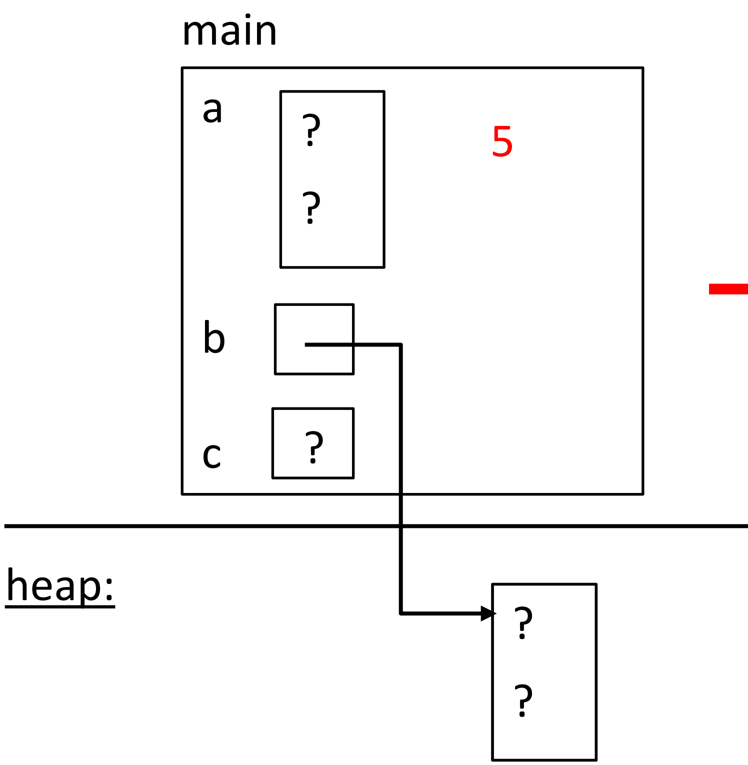



```c
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char** argv) {
  int a[2];
  int* b = malloc(2*sizeof(int));
  int* c;

  a[2] = 5;   // assigns past the end of an array
  b[0] += 2;  // assumes malloc zeros out memory
  c = b+3;    // Ok, but if we use c, problem
  free(&(a[0]));  // free something not malloc'ed
  free(b);
  free(b);    // double-free the same block
  b[0] = 5;   // use a freed (dangling) pointer

  // any many more!
  return 0;
}
```

Note: Arrow points to *next* instruction.

memcorrupt.c

Memory Corruption - What Happens?

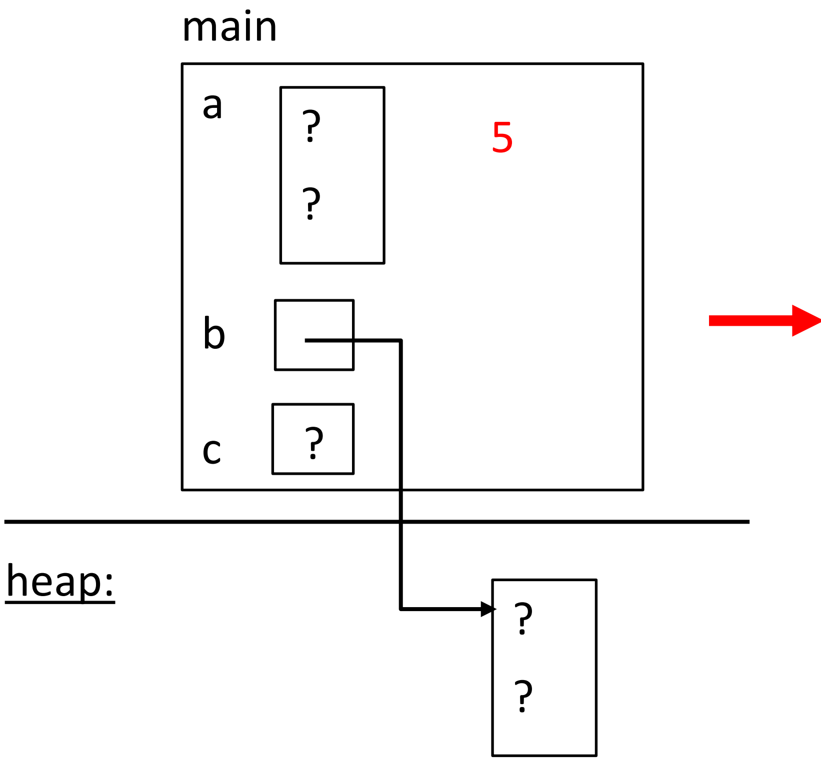



```c
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char** argv) {
  int a[2];
  int* b = malloc(2*sizeof(int));
  int* c;

  a[2] = 5;    // assigns past the end of an array
  b[0] += 2;   // assumes malloc zeros out memory
  c = b+3;     // Ok, but if we use c, problem
  free(&(a[0]));   // free something not malloc'ed
  free(b);
  free(b);     // double-free the same block
  b[0] = 5;    // use a freed (dangling) pointer

  // any many more!
  return 0;
}
```

Note: Arrow points to *next* instruction.

memcorrupt.c

Memory Corruption - What Happens?

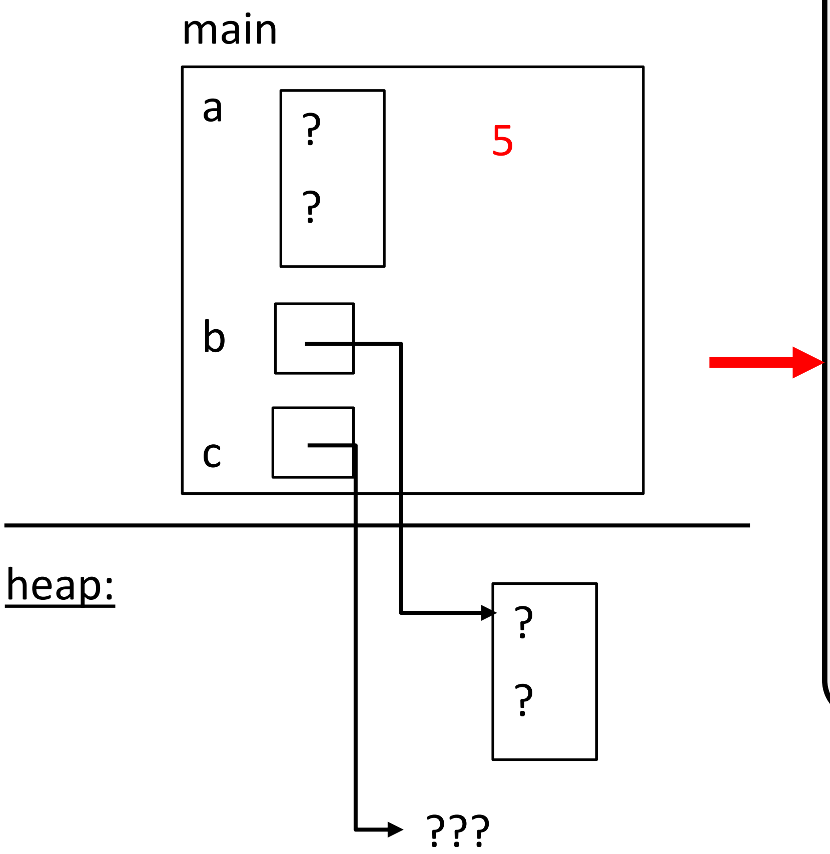



```c
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char** argv) {
  int a[2];
  int* b = malloc(2*sizeof(int));
  int* c;

  a[2] = 5;    // assigns past the end of an array
  b[0] += 2;   // assumes malloc zeros out memory
  c = b+3;     // Ok, but if we use c, problem
  free(&(a[0]));  // free something not malloc'ed
  free(b);
  free(b);     // double-free the same block
  b[0] = 5;    // use a freed (dangling) pointer

  // any many more!
  return 0;
}
```

Note: Arrow points to *next* instruction.

memcorrupt.c

Memory Corruption - What Happens?

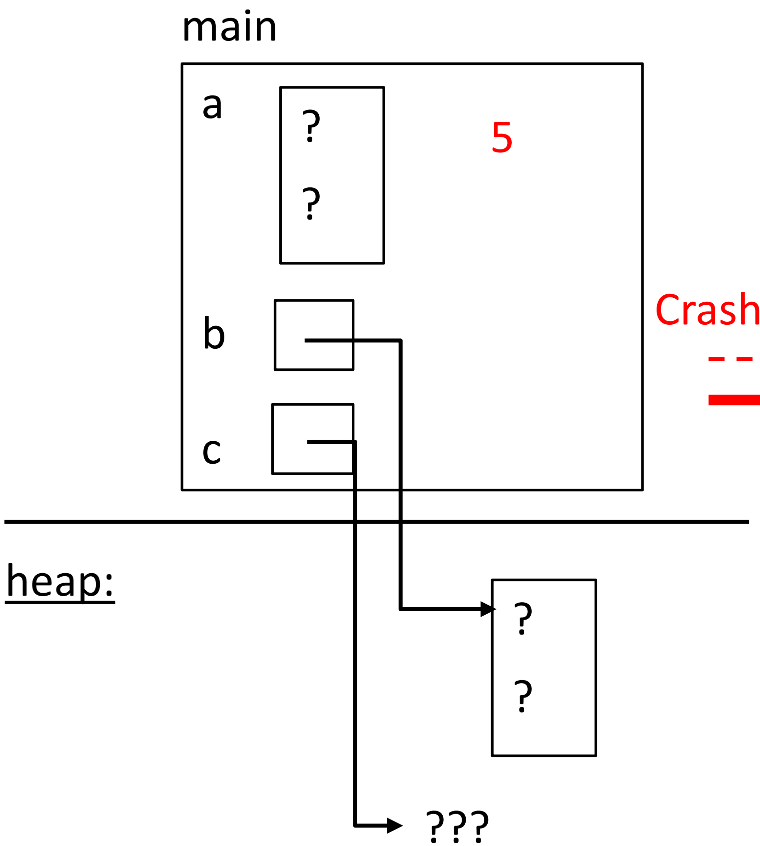



```
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char** argv) {
  int a[2];
  int* b = malloc(2*sizeof(int));
  int* c;

  a[2] = 5;    // assigns past the end of an array
  b[0] += 2;   // assumes malloc zeros out memory
  c = b+3;     // Ok, but if we use c, problem
  free(&(a[0]));   // free something not malloc'ed
  free(b);
  free(b);     // double-free the same block
  b[0] = 5;    // use a freed (dangling) pointer

  // any many more!
  return 0;
}
```

Note: Arrow points to *next* instruction.

memcorrupt.c

Memory Corruption - What Happens?



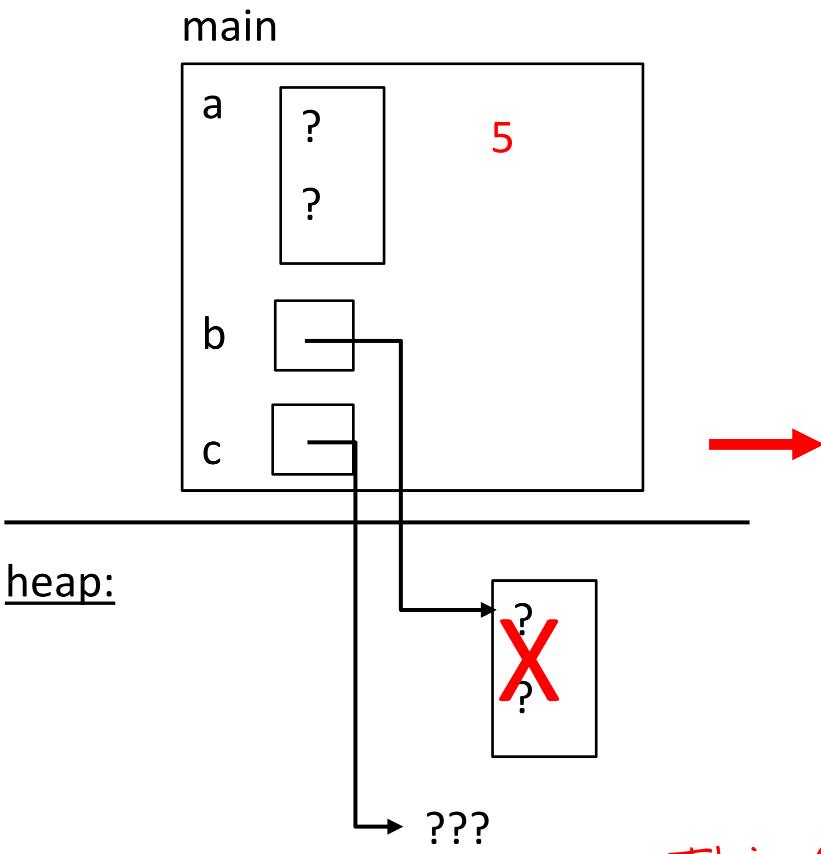
Crash!

```c
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char** argv) {
  int a[2];
  int* b = malloc(2*sizeof(int));
  int* c;

  a[2] = 5;   // assigns past the end of an array
  b[0] += 2;  // assumes malloc zeros out memory
  c = b+3;    // Ok, but if we use c, problem
  free(&(a[0]));  // free something not malloc'ed
  free(b);
  free(b);    // double-free the same block
  b[0] = 5;   // use a freed (dangling) pointer

  // any many more!
  return 0;
}
```

<u>Note</u>: Arrow points to *next* instruction.

memcorrupt.c

Memory Corruption - What Happens?



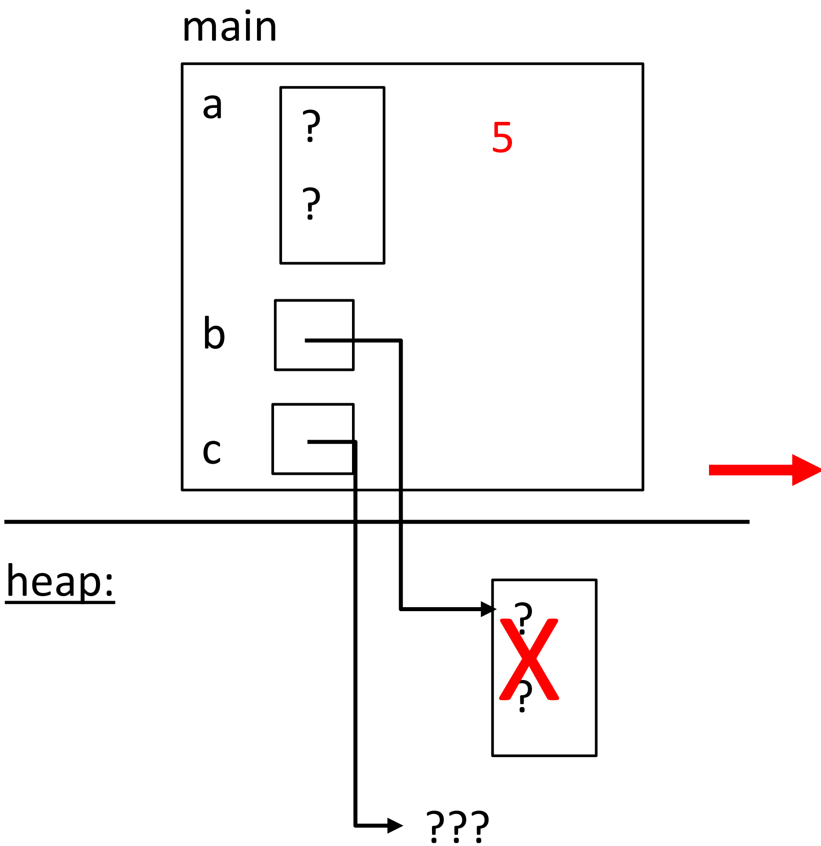

```c
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char** argv) {
  int a[2];
  int* b = malloc(2*sizeof(int));
  int* c;

  a[2] = 5;    // assigns past the end of an array
  b[0] += 2;   // assumes malloc zeros out memory
  c = b+3;     // Ok, but if we use c, problem
  free(&(a[0]));  // free something not malloc'ed
  free(b);
  free(b);     // double-free the same block
  b[0] = 5;    // use a freed (dangling) pointer

  // any many more!
  return 0;
}
```

Note: Arrow points to *next* instruction.

This "double free" would also cause the program to crash

memcorrupt.c

Memory Corruption - What Happens?

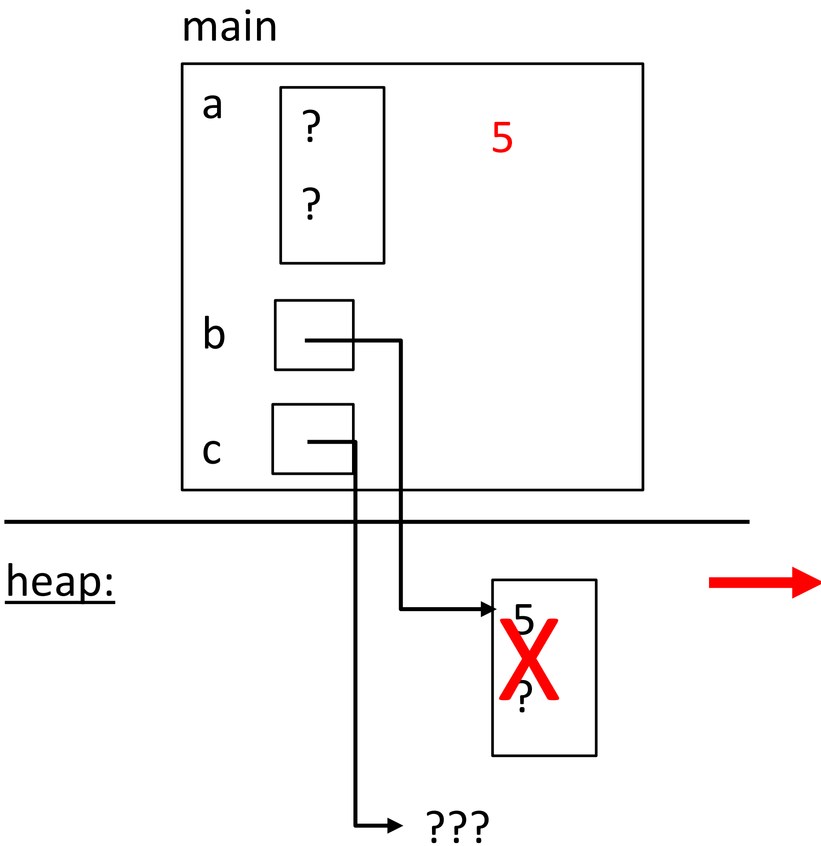



```c
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char** argv) {
  int a[2];
  int* b = malloc(2*sizeof(int));
  int* c;

  a[2] = 5;    // assigns past the end of an array
  b[0] += 2;   // assumes malloc zeros out memory
  c = b+3;     // Ok, but if we use c, problem
  free(&(a[0]));   // free something not malloc'ed
  free(b);
  free(b);     // double-free the same block
  b[0] = 5;    // use a freed (dangling) pointer

  // any many more!
  return 0;
}
```

Note: Arrow points to *next* instruction.

memcorrupt.c

Memory Corruption - What Happens?




```c
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char** argv) {
  int a[2];
  int* b = malloc(2*sizeof(int));
  int* c;

  a[2] = 5;    // assigns past the end of an array
  b[0] += 2;   // assumes malloc zeros out memory
  c = b+3;     // Ok, but if we use c, problem
  free(&(a[0]));   // free something not malloc'ed
  free(b);
  free(b);     // double-free the same block
  b[0] = 5;    // use a freed (dangling) pointer

  // any many more!
  return 0;
}
```

Note: Arrow points to *next* instruction.

memcorrupt.c

Lecture Outline

- **C Refresher**
  - C Strings
  - Dynamic memory (malloc & **realloc**)
  - **Structs**
- Processes
  - Overview
  - fork()
  - exec()

Structured Data

- A `struct` is a C datatype that contains a set of fields
  - Similar to a Java class, but with no methods or constructors
  - Useful for defining new structured types of data
  - Acts similarly to primitive variables
- Generic declaration:

```
// declaring the struct type
struct point {
  float x;
  float y;
};

// declaring a variable
struct point pt;
```

```
// declaring the struct type
typedef struct point_st {
  float x;
  float y;
} point;

// declaring a variable
point pt;
```

Structured Data Initialization

- A `struct` is a C datatype that contains a set of fields
  - Acts similarly to primitive variables
- Generic declaration:

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

point pt;
point origin = {0.0f, 0.0f};
point other = (point) {
  .x = 3.14f,
  .y = 3.800f,
};

pt = origin; // pt now contains 0.0f, 0.0f
```

Default values are still garbage! (arrow to `point pt;`)

<- Initializer List

<- with designators

^ same as `pt.x = origin.x;`
`pt.y = origin.y;`

Accessing struct Fields

- Use " . " to refer to a field in a struct
- Use "->" to refer to a field from a struct pointer
  - Dereferences pointer first, then accesses field

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

int main(int argc, char** argv) {
  Point p1 = {0.0, 0.0};
  Point* p1_ptr = &p1;

  p1.x = 1.0;
  p1_ptr->y = 2.0;  // equivalent to (*p1_ptr).y = 2.0;
  return 0;
}
```

Output parameters (again)

- One way to handle multiple return values through output parameters
  - This function generates an array of `int` and returns the length (or -1 on error)

```
ssize_t gen_arr(int** output_arr);
```

`ssize_t` is just a signed integer type to represent a size

Signed **SIZE** **T**ype

Poll Everywhere

pollev.com/tqm

- How do you think we call this function?
  - It generates an array of `int` and returns the length (or -1 on error)

```
ssize_t gen_arr(int** output_arr);
```

Poll Everywhere

pollev.com/tqm

- How do you think we call this function?
  - It generates an array of `int` and returns the length (or -1 on error)

```
int* arr;
ssize_t res = gen_arr(&arr);
if (res < 0) {
  // handle error
}
```

Structs vs output parameters

- The parameter `output_arr` is entirely for output, it messes with our common understanding of a parameters as input

```
ssize_t gen_arr(int** output_arr);
```

- An alternative way this function could be written is with a struct that contains both values:

- Which do you think is more readable?

```
typedef struct int_arr_st {
  int* eles
  size_t length;
} int_arr;

int_arr gen_arr();
```

Another example

- Another common example are functions that produce something but can error.
- Consider this function that produces some struct (lets call it `struct addrinfo`) but can error.

```
bool addr_info(struct addrinfo* output);
```

```
typedef struct optional_addrinfo_st {
  bool has_value;
  struct addr_info value;
} optional_addrinfo;

optional_addrinfo gen_arr();
```

- The first is more common in C and the C stdlib, but you can do either in functions you write

Demo: implementing a simple int vector

- Demo: `vec_int.c` inside of 01-code.zip
  - Starting from `blank_vec_int.c`
  - Explaining design
  - How do we implement **`vec_push()`**?
  - Why do we need to pass in a `vec_int*` instead of just `vec_int`?

`realloc()`

- ```
  void *realloc(void* ptr, size_t size);
  ```
  `realloc` is used to "re-allocate" a block of memory to be the requested size
  - This means previous values in **`ptr`** will be in the reallocated memory

- Returns a pointer to the first byte of that memory
  - And returns `NULL` if the memory allocation failed!

- `realloc(NULL, size)` is equal to `malloc(size)`

- See `vec_int.c` for an example of how realloc is useful
  ```

Lecture Outline

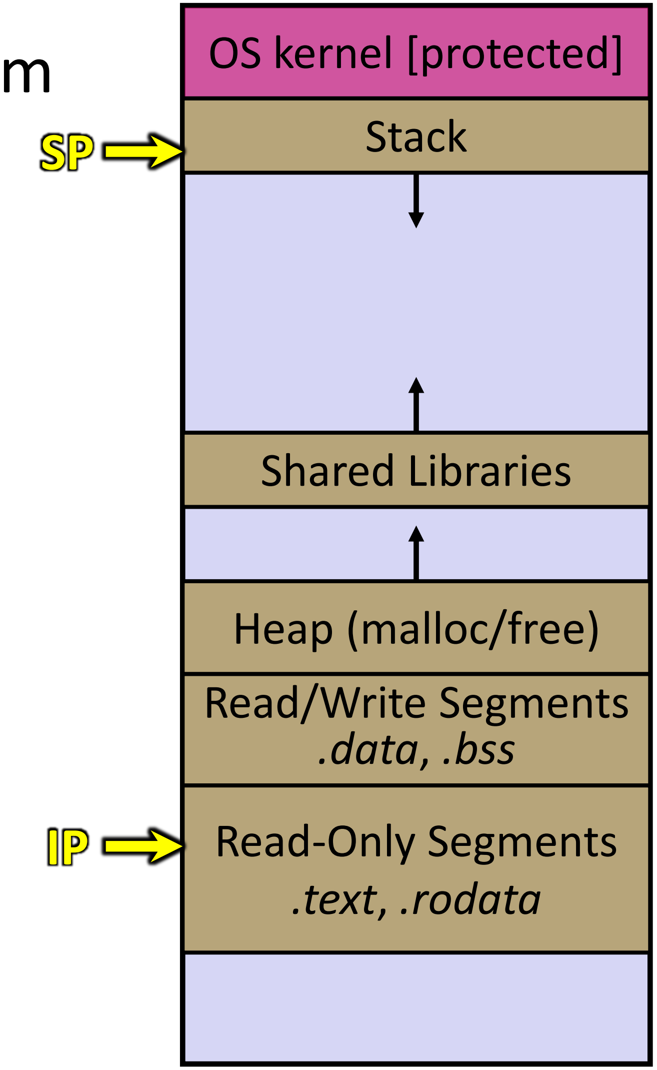
- C Refresher
  - C Strings
  - Dynamic memory (malloc & realloc)
  - Structs
- **Processes**
  - **Overview**
  - fork()
  - exec()

Definition: Process

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

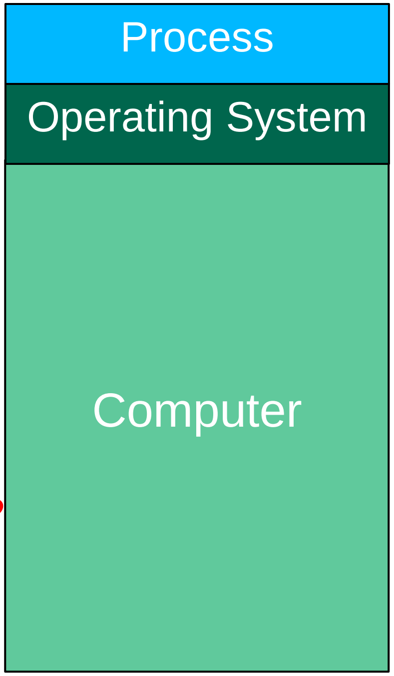
- Consists of:
  - Memory (code, heap, stack, etc)
  - Registers used to manage execution (stack pointer, program counter, ...)
  - Other resources

* 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:

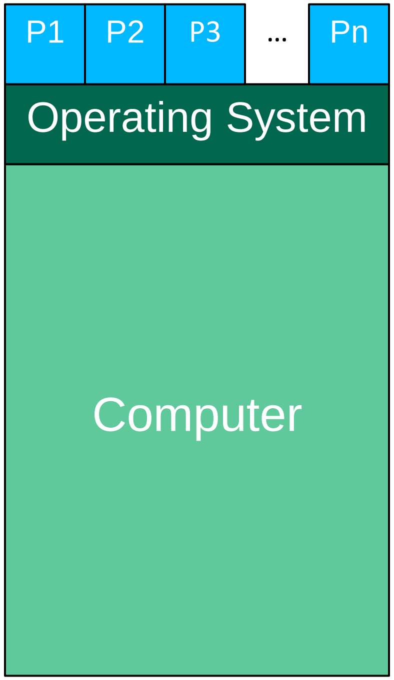


What is missing/wrong with this?

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

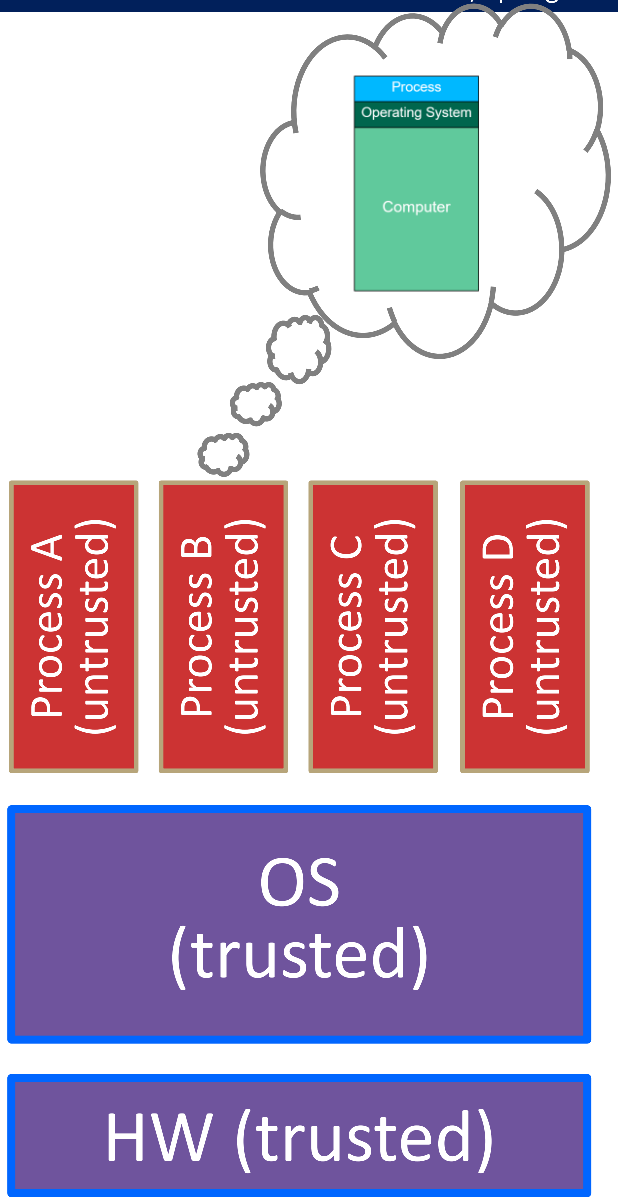
Multiple Processes

- 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

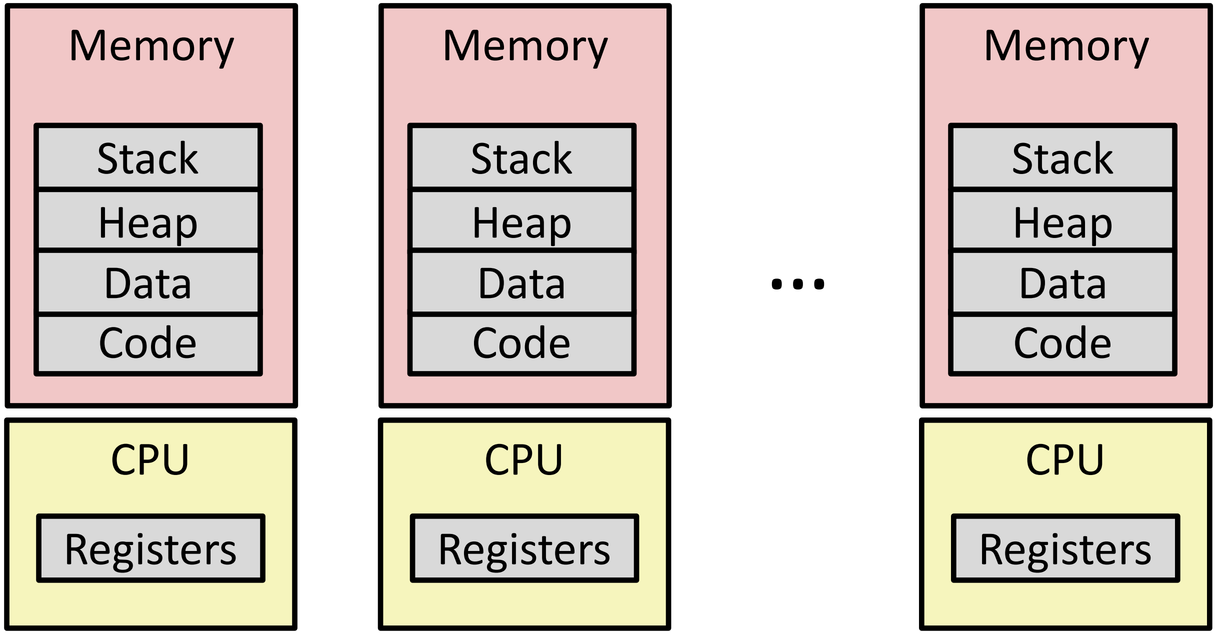


OS: Protection System

- 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

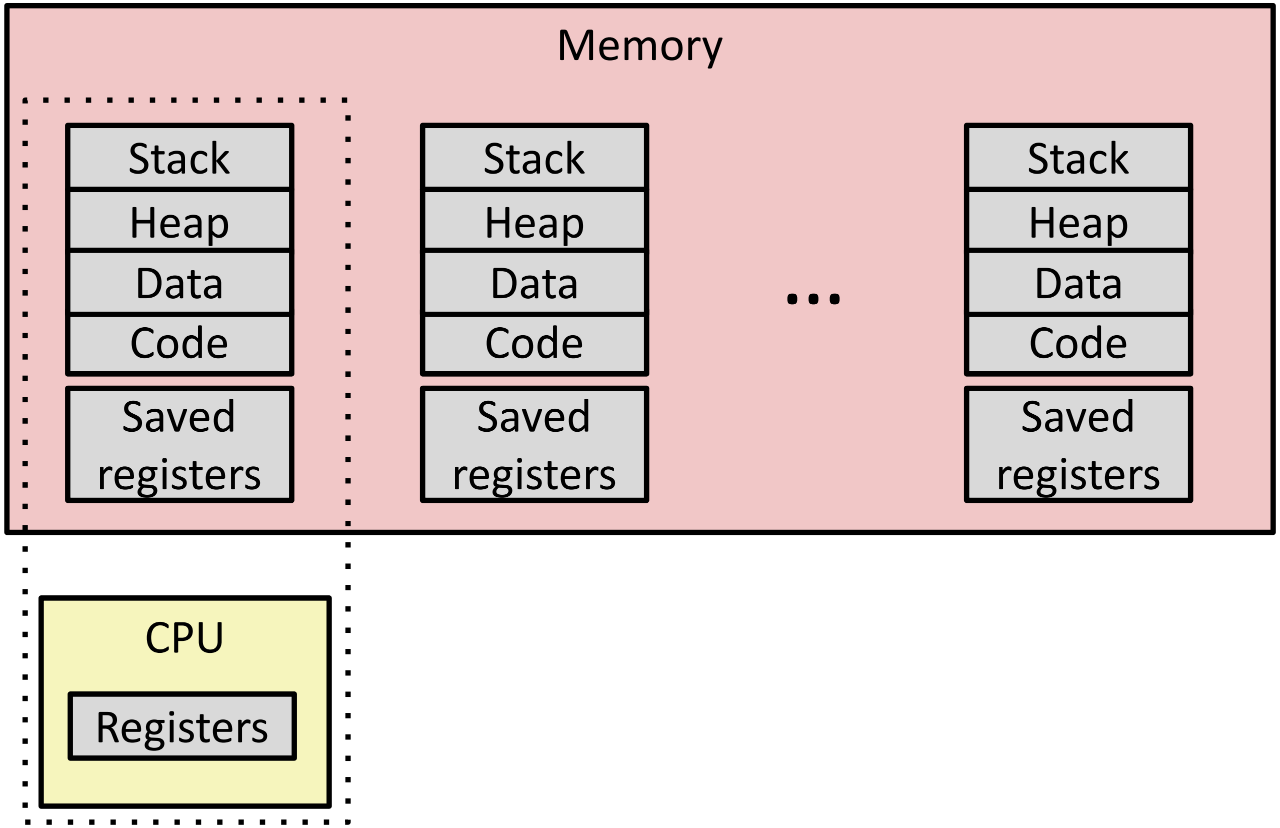


Multiprocessing: The Illusion



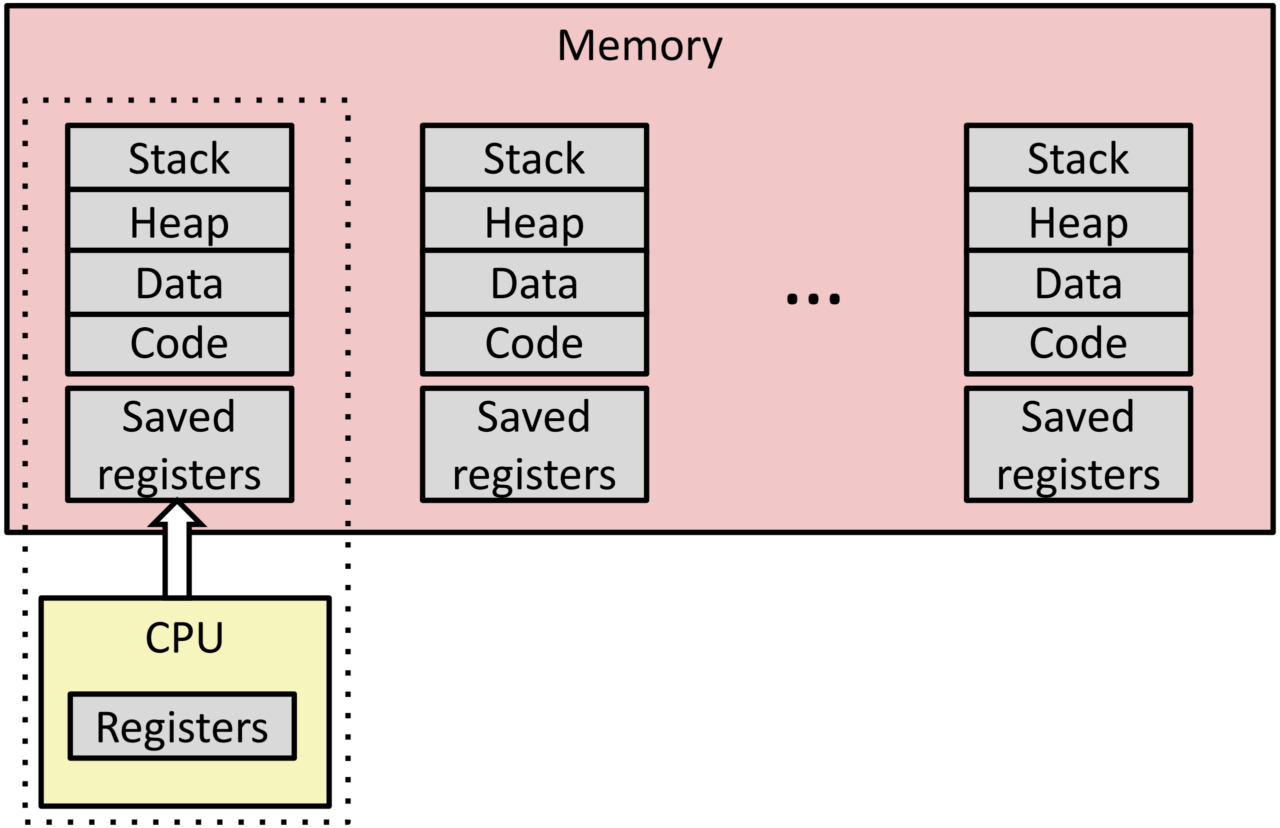
- Computer runs many processes simultaneously
  - Applications for one or more users
    - Web browsers, email clients, editors, …
  - Background tasks
    - Monitoring network & I/O devices

Multiprocessing: The (Traditional) Reality



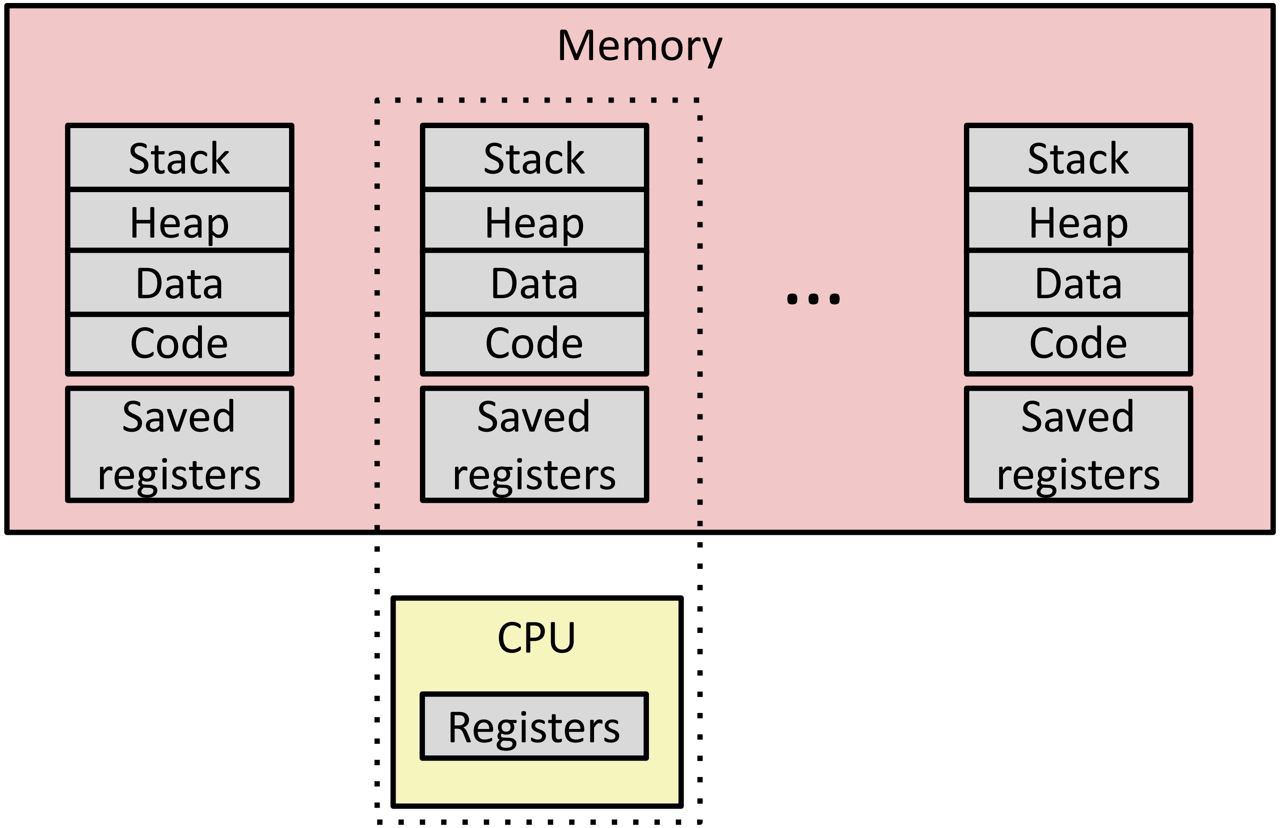
- Single processor executes multiple processes concurrently
  - Process executions interleaved (multitasking)
  - Address spaces managed by virtual memory system (later in course)
  - Register values for nonexecuting processes saved in memory

Multiprocessing: The (Traditional) Reality



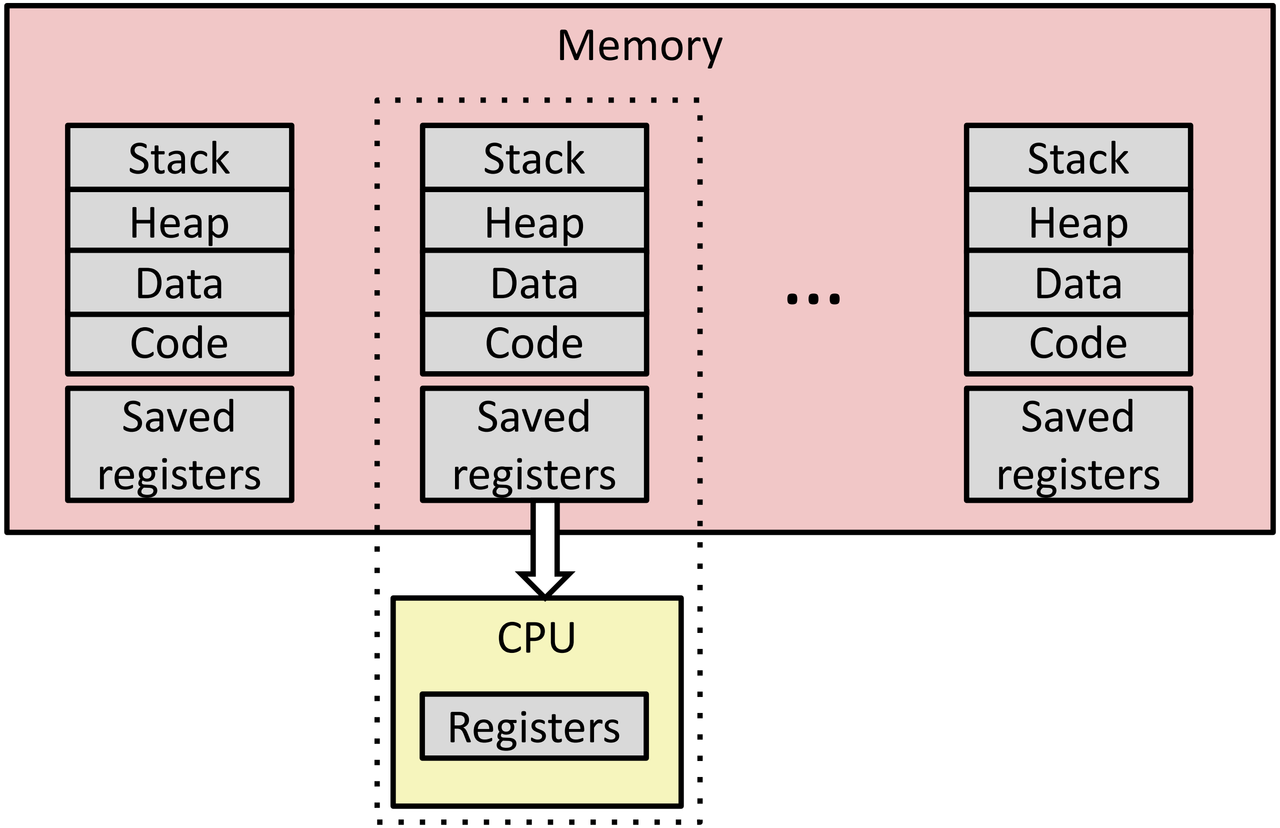
1. Save current registers in memory

Multiprocessing: The (Traditional) Reality



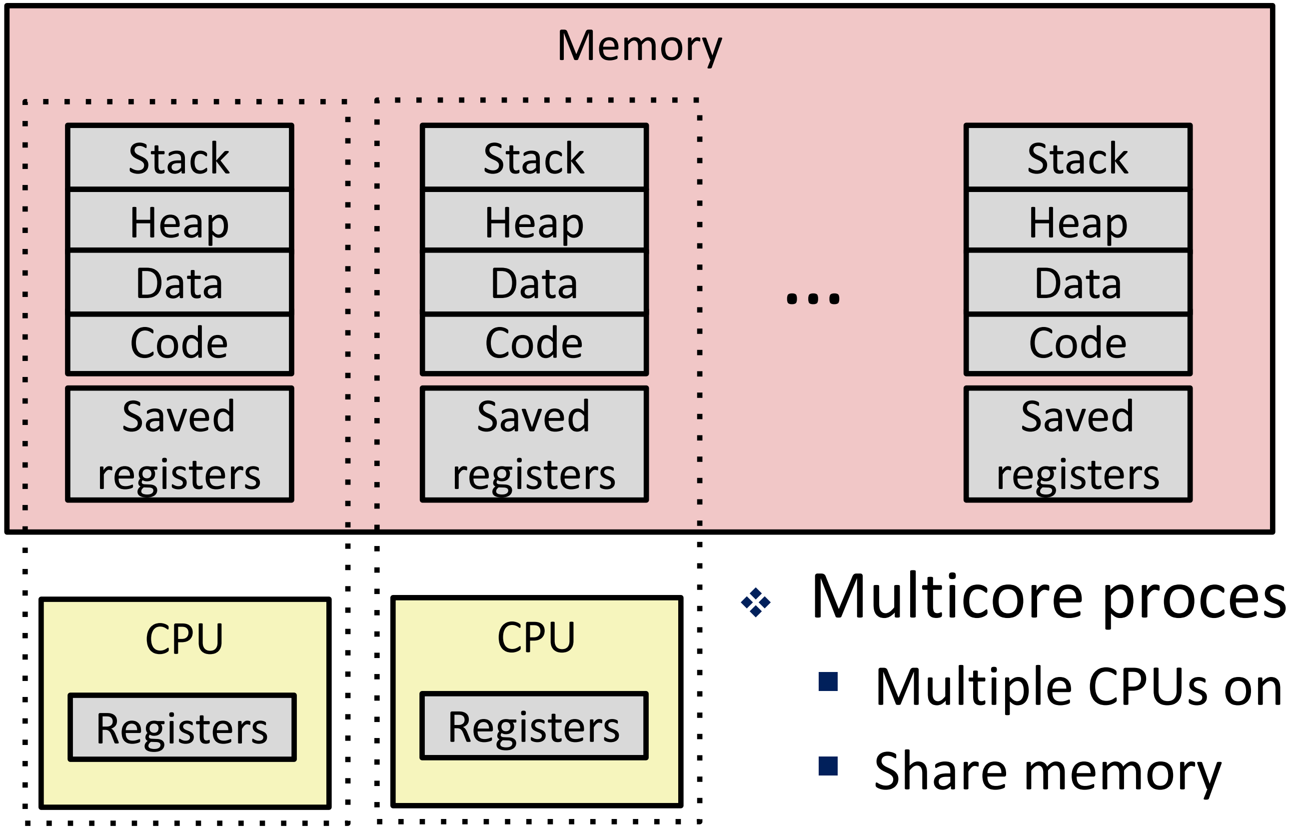
1. Save current registers in memory
2. Schedule next process for execution

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 (Modern) 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/tqm

- 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?

Process States (incomplete)

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

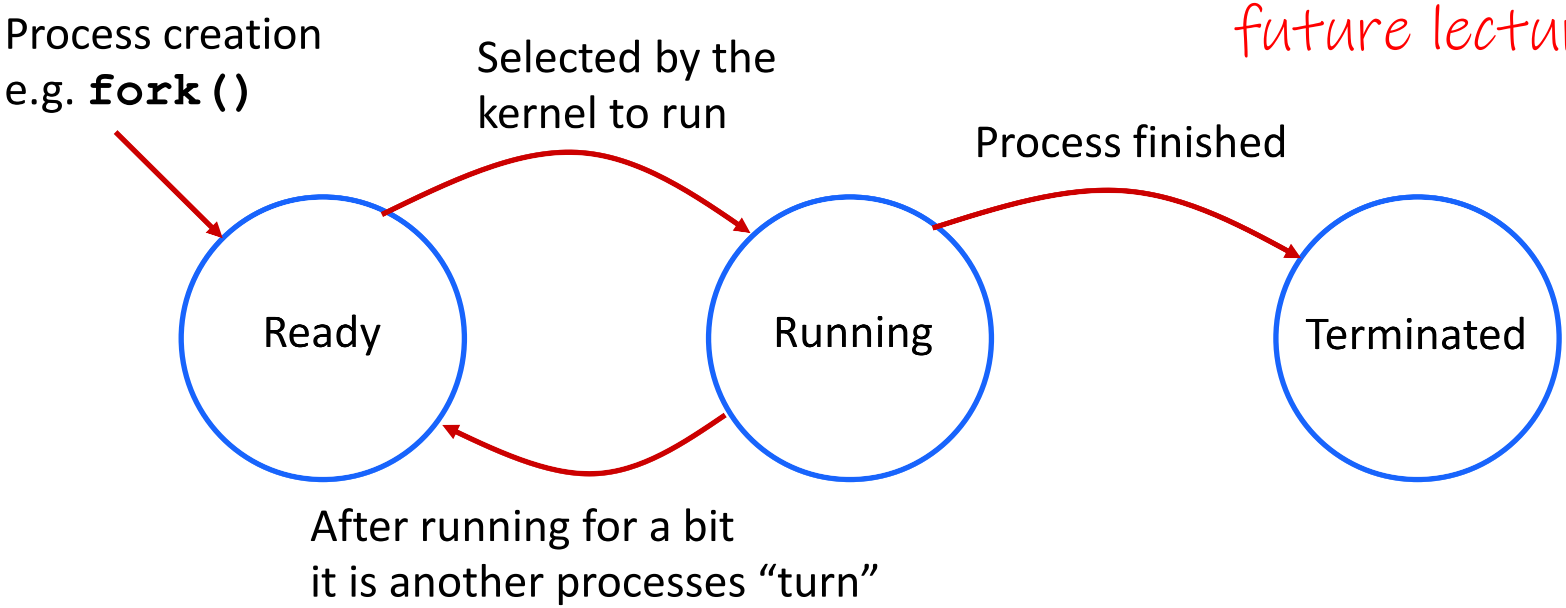
More states in future lectures

- 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
- Terminated
  - Process is stopped permanently

Scheduler to be covered in a later lecture

Process State Lifetime (incomplete)

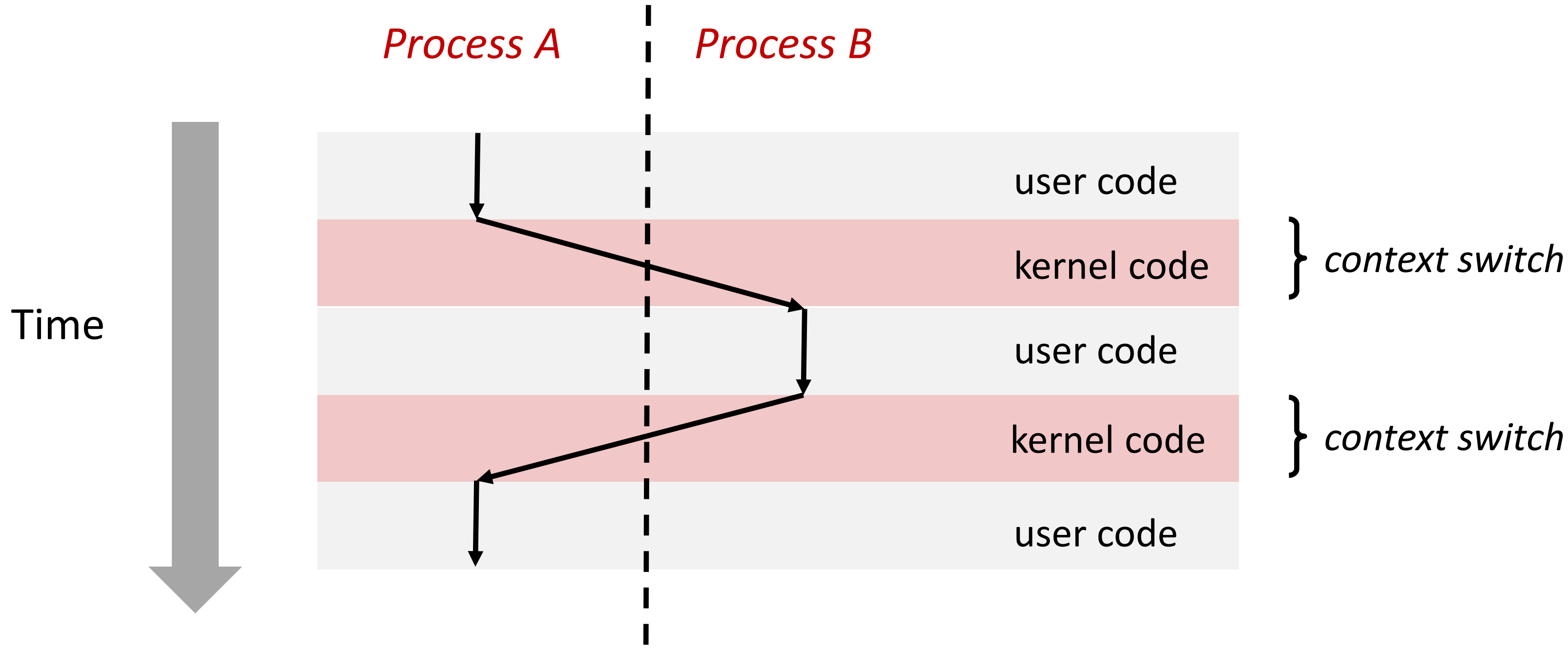
More states in future lectures



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.

Lecture Outline

- C Refresher
  - C Strings
  - Dynamic memory (malloc & realloc)
  - Structs
- **Processes**
  - Overview
  - **fork()**
  - exec()

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

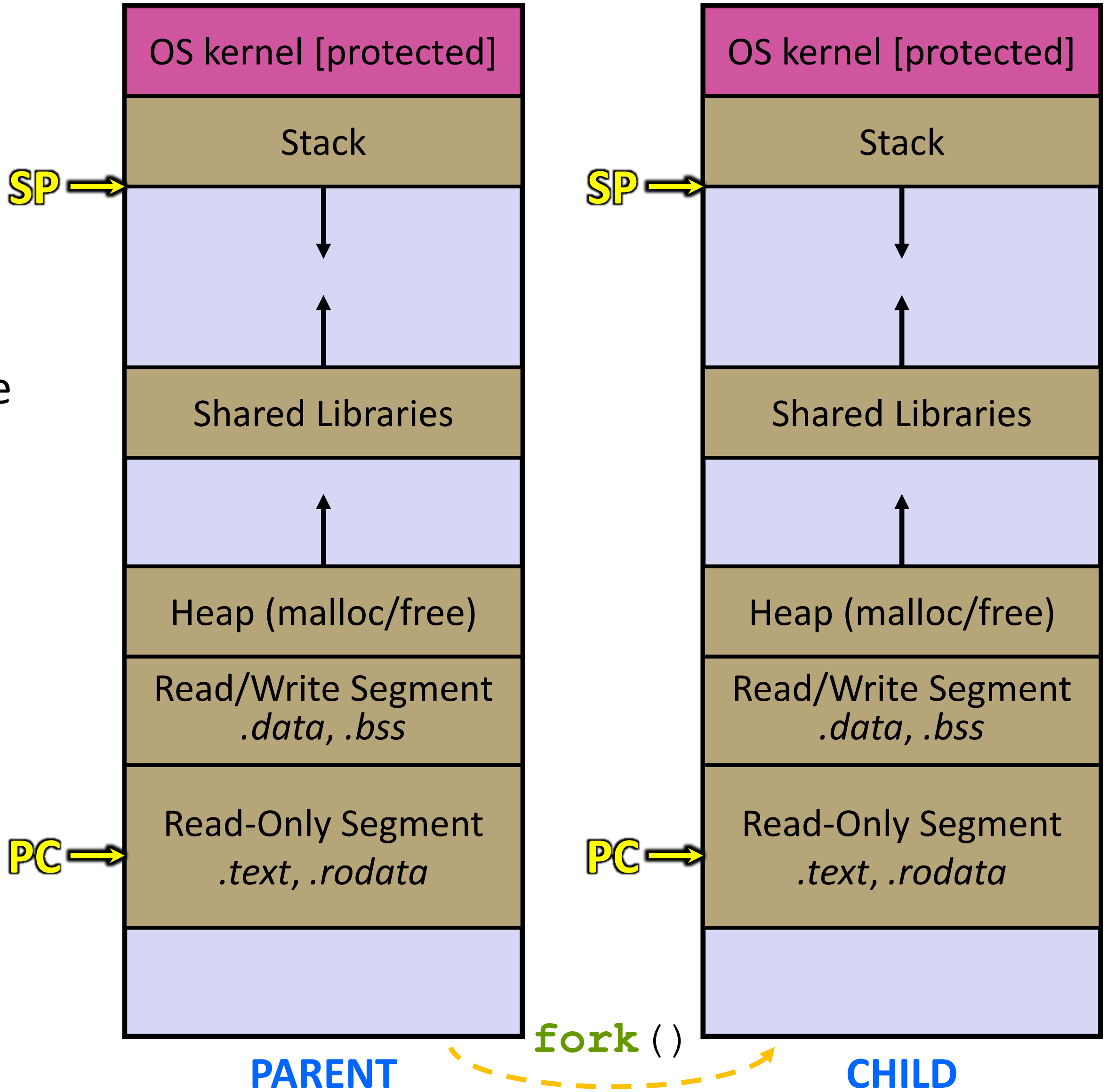
- `exit` is called once but never returns.

Creating 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.

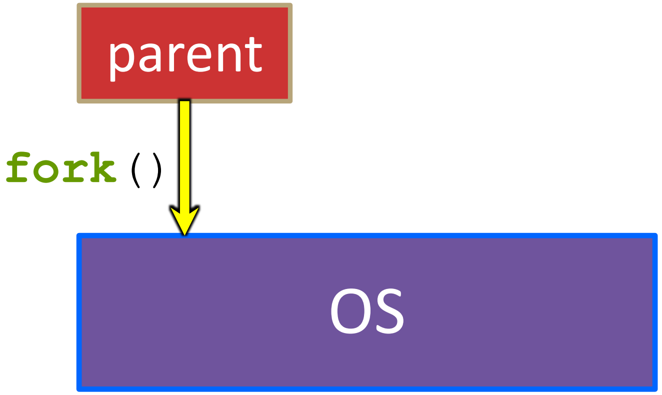
`fork()` and Address Spaces

- 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.



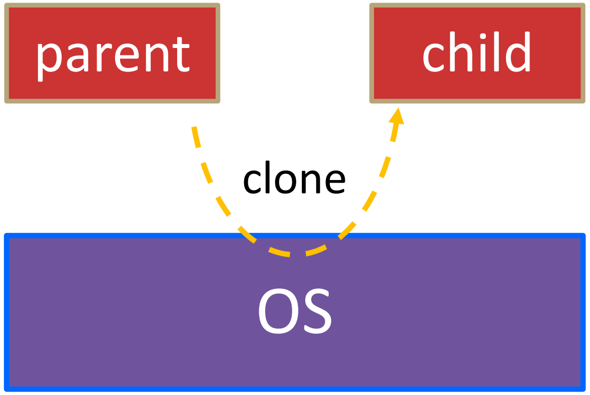
`fork()`

- `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



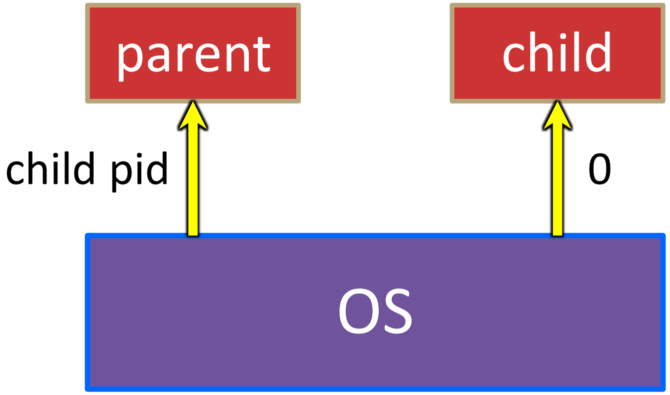
`fork()`

- `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

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

- What does this print?

fork() example

```
pid_t fork_ret = fork();

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

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?

Non-deterministic

Another fork() example

```
pid_t fork_ret = fork();
int x;

if (fork_ret == 0) {
  x = 3800;
} else {
  x = 2400;
}
printf("%d\n", x);
```

Another fork() example

Parent Process (PID = X)

```c
pid_t fork_ret = fork();
int x;

if (fork_ret == 0) {
  x = 3800;
} else {
  x = 2400;
}
printf("%d\n", x);
```

Child Process (PID = Y)

```c
pid_t fork_ret = fork();
int x;

if (fork_ret == 0) {
  x = 3800;
} else {
  x = 2400;
}
printf("%d\n", x);
```

fork()

Another fork() example

Parent Process (PID = X)

```
pid_t fork_ret = fork();
int x;

if (fork_ret == 0) {
  x = 3800;
} else {
  x = 2400;
}
printf("%d\n", x);
```

Child Process (PID = Y)

```
pid_t fork_ret = fork();
int x;

if (fork_ret == 0) {
  x = 3800;
} else {
  x = 2400;
}
printf("%d\n", x);
```

fork_ret = Y

fork()

fork_ret = 0

Always prints "2400"

Always prints "3800"

Reminder: Processes have their own address space (and thus, copies of their own variables)

Order is still nondeterministic!!

Lecture Outline

- C Refresher
  - C Strings
  - Dynamic memory (malloc & realloc)
  - Structs
- **Processes**
  - Overview
  - fork()
  - **exec()**

exec*()

- Loads in a new program for execution

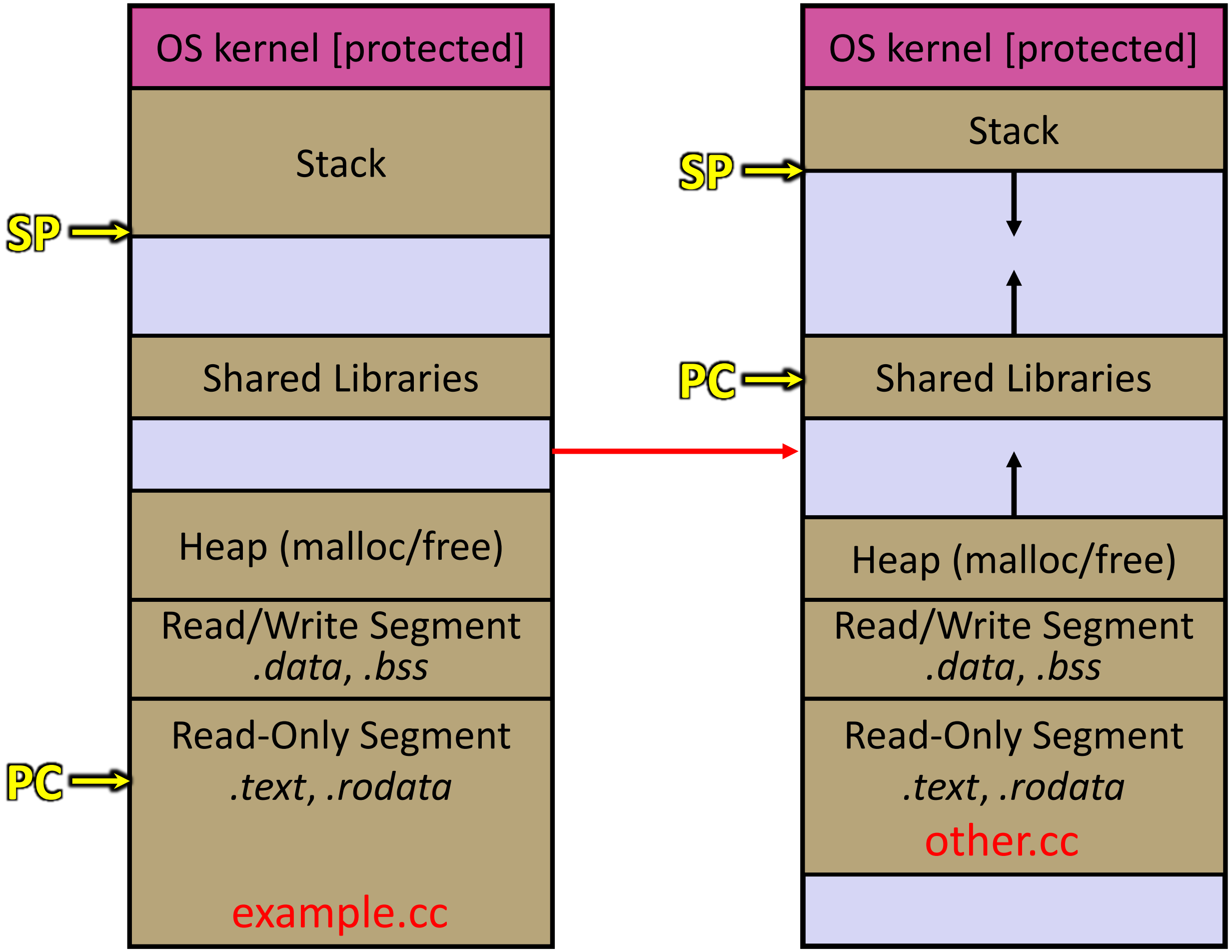
- PC, SP, registers, and memory are all reset so that the specified program can run

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/tqm

```
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;
}
```

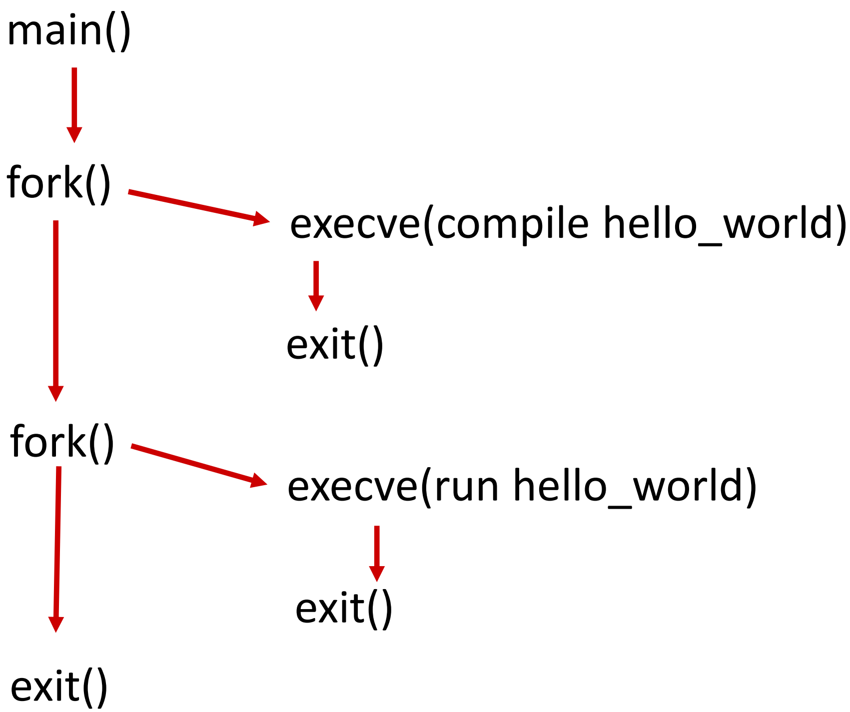
broken_autograder.c

This code is broken. It compiles, but it doesn't do what we want. Why?

- Clang is a C compiler
- Assume it compiles
- Assume I gave the correct args to exec

Poll Everywhere

pollev.com/tqm



This code is broken. It compiles, but it doesn’t do what we want. Why?

- Clang is a C compiler
- Assume it compiles
- Assume I gave the correct args to exec