Processes (Cont.) & Threads

Computer Systems Programming, Spring 2023

Instructor: Travis McGaha

TAs:

Kevin Bernat	Jialin Cai
Mati Davis	Donglun He
Chandravaran Kunjeti	Heyi Liu
Shufan Liu	Eddy Yang

Logistics

- HW1 (FileReaders) Due Thursday 2/9 @ 11:59 pm
  - Released, autograder **coming out later TODAY**
  - You should have everything you need to complete the assignment
  - Recitation last week gave helpful practice with writing POSIX code

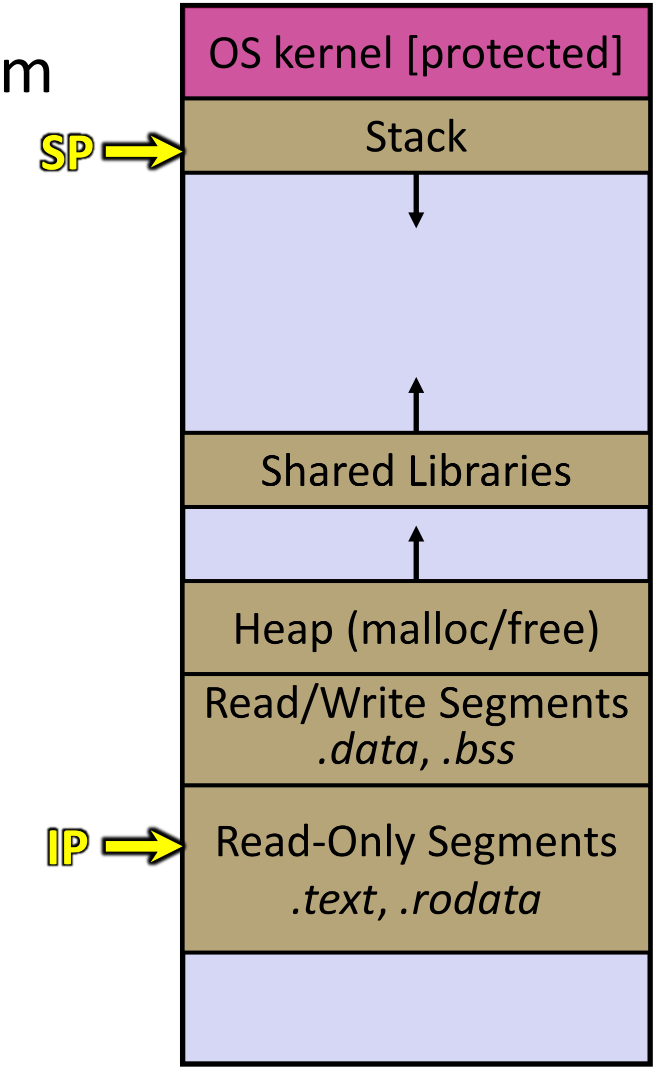
Lecture Outline

- Review of Processes
- Interleaving & Scheduling
- `wait` & `sleep`
- Threads
- `pthreads`

Definition: Process

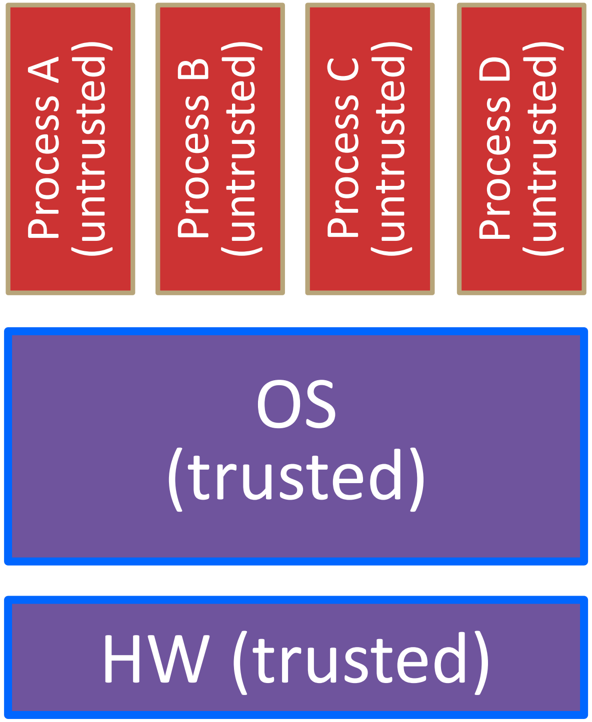
- Definition: An instance of a program that is being executed
  (or is ready for execution)

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



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

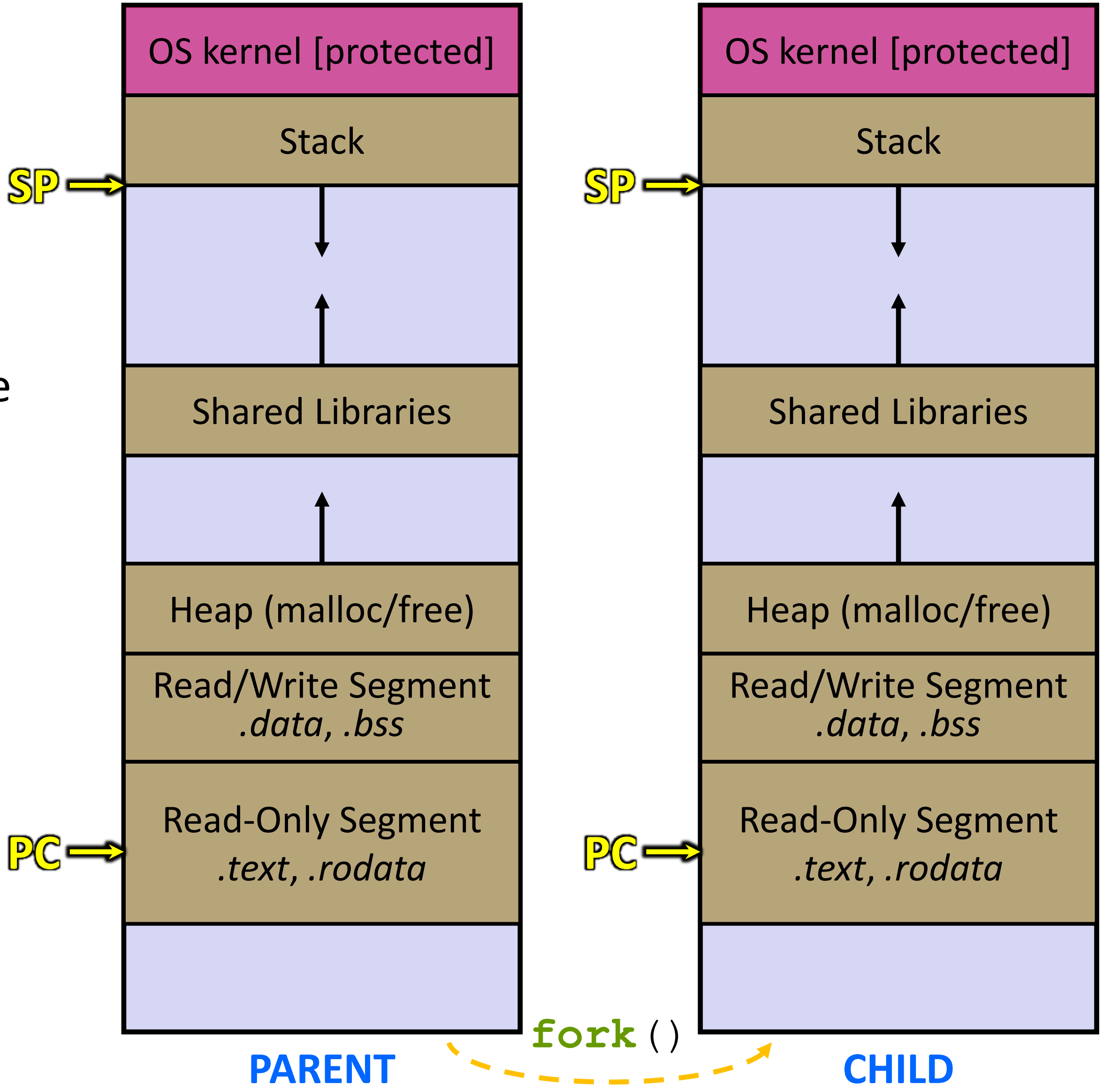


Creating New Processes

- `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

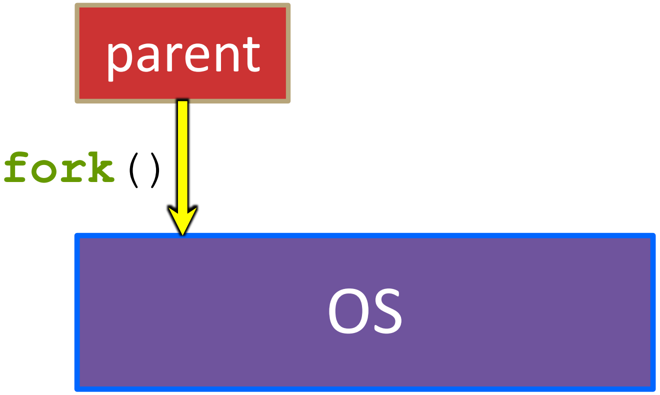
`fork()` and Address Spaces

- 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.



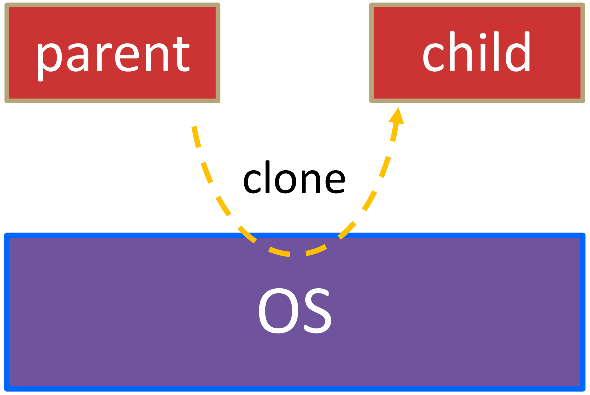
`fork()`

- `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



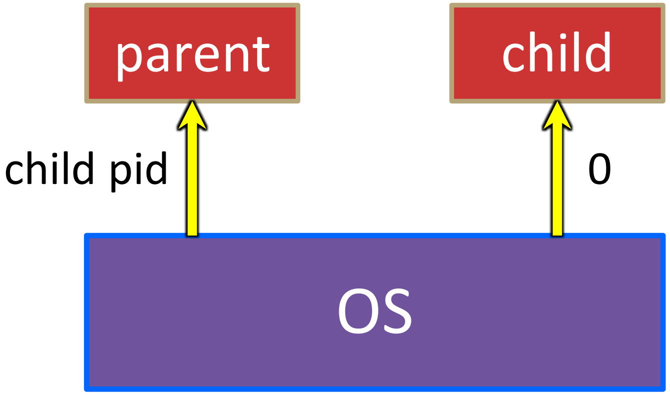
`fork()`

- `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



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.

"simple" fork() example

```
fork();
cout << "Hello!\n";
exit(EXIT_SUCCESS);
```

Prints "Hello!\n" twice, once from each process

"simple" fork() example

```
int x = 3;
fork();
x++;
cout << x << endl;
exit(EXIT_SUCCESS);
```

Prints "4\n" twice, once from each process.
Each process has separate memory, and thus
their own independent copy of x

Process States

From a programmer’s perspective, we can think of a process as being in one of three states

- Running
  - Process is either executing, or waiting to be executed and will eventually be *scheduled* (i.e., chosen to execute) by the kernel
- Stopped
  - Process execution is *suspended* and will not be scheduled until further notice (next lecture when we study signals)
- Terminated
  - Process is stopped permanently

OS: The Scheduler

- When switching between processes, the OS will 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 other 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.

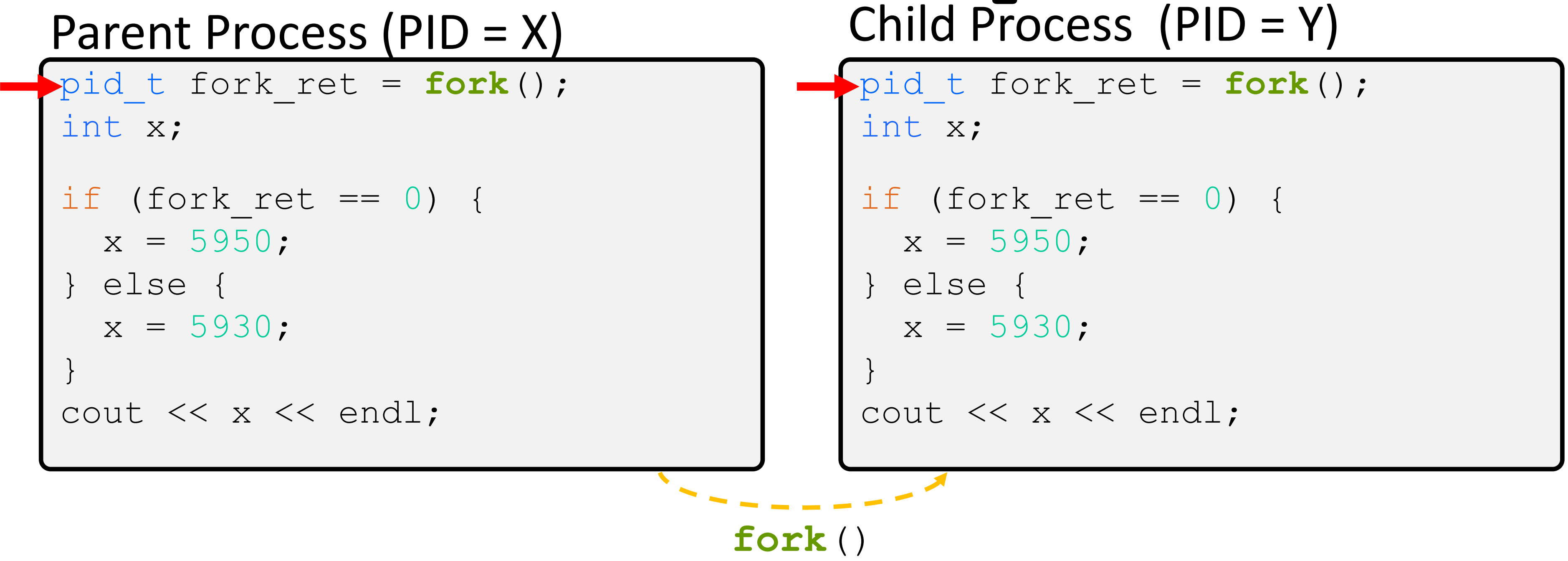
- Algorithms are designed to consider many factors:
  - Fairness: Every program gets to run
  - Liveness: That “something” will eventually happen
  - Throughput: Number of “tasks” completed over an interval of time
  - Wait time: Average time a “task” is “alive” but not running
  - A lot more…

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

Another fork() example

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

if (fork_ret == 0) {
  x = 5950;
} else {
  x = 5930;
}
cout << x << endl;
```

Another `fork()` example



Parent Process (PID = X)

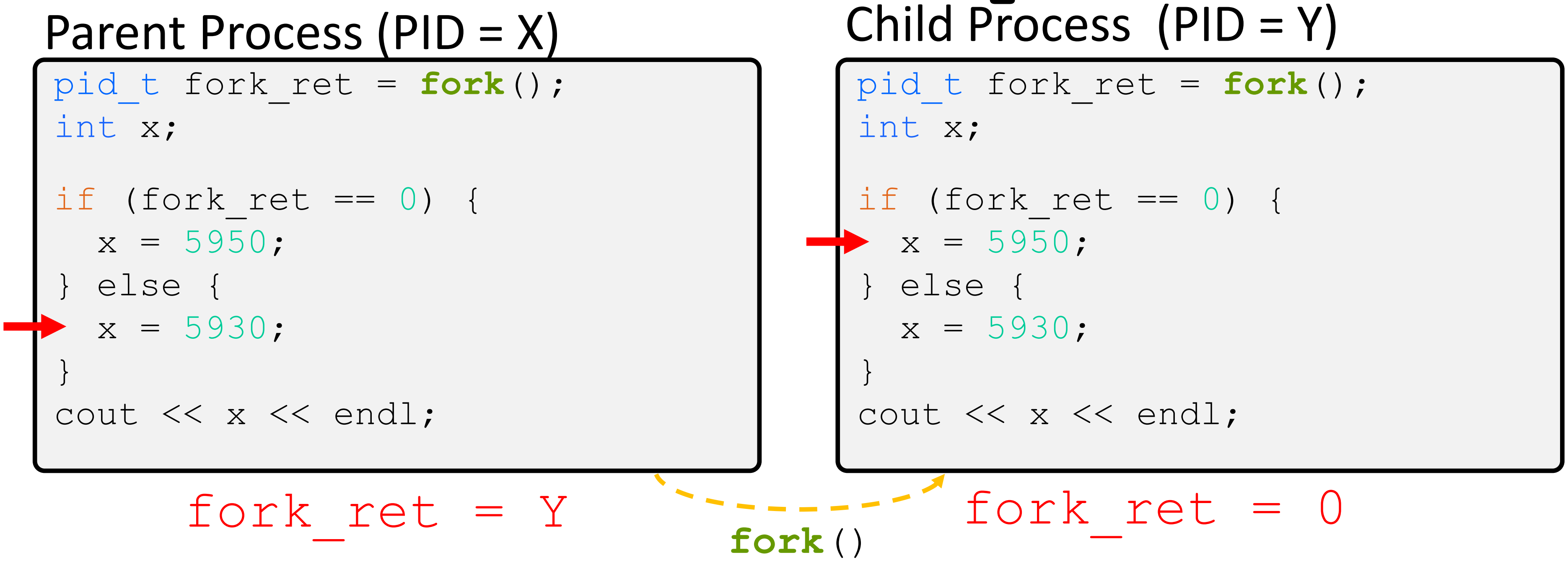
```
pid_t fork_ret = fork();
int x;

if (fork_ret == 0) {
  x = 5950;
} else {
  x = 5930;
}
cout << x << endl;
```

Child Process (PID = Y)

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

if (fork_ret == 0) {
  x = 5950;
} else {
  x = 5930;
}
cout << x << endl;
```

`fork()`

Another `fork()` example



Parent Process (PID = X)

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

if (fork_ret == 0) {
  x = 5950;
} else {
  x = 5930;
}
cout << x << endl;
```

`fork_ret = Y`

Child Process (PID = Y)

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

if (fork_ret == 0) {
  x = 5950;
} else {
  x = 5930;
}
cout << x << endl;
```

`fork_ret = 0`

Always prints "5930"

Always prints "5950"

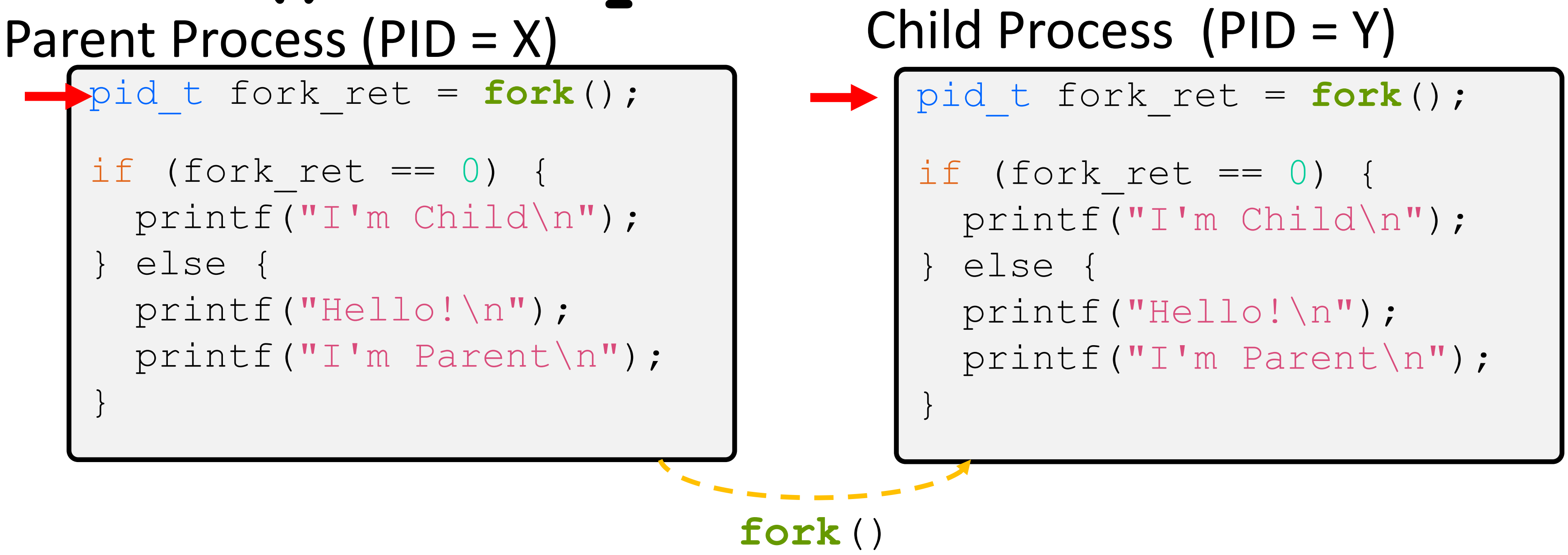
Reminder: Processes have their own address space (and thus, copies of their own variables)

Order is still nondeterministic!!

fork() example

```
pid_t fork_ret = fork();

if (fork_ret == 0) {
  printf("I'm Child\n");
} else {
  printf("Hello!\n");
  printf("I'm Parent\n");
}
```

`fork()` example



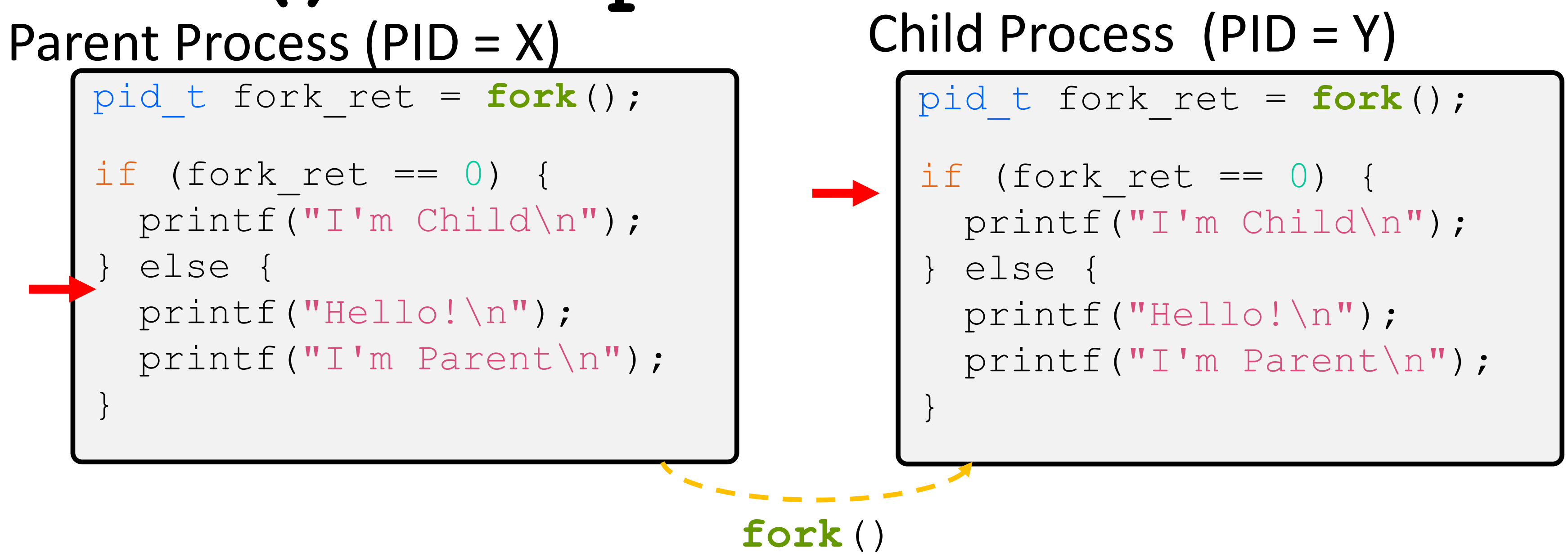
Parent Process (PID = X)

```
pid_t fork_ret = fork();

if (fork_ret == 0) {
  printf("I'm Child\n");
} else {
  printf("Hello!\n");
  printf("I'm Parent\n");
}
```

Child Process (PID = Y)

```
pid_t fork_ret = fork();

if (fork_ret == 0) {
  printf("I'm Child\n");
} else {
  printf("Hello!\n");
  printf("I'm Parent\n");
}
```

`fork()` example



Parent Process (PID = X)

```c
pid_t fork_ret = fork();

if (fork_ret == 0) {
  printf("I'm Child\n");
} else {
  printf("Hello!\n");
  printf("I'm Parent\n");
}
```

Child Process (PID = Y)

```c
pid_t fork_ret = fork();

if (fork_ret == 0) {
  printf("I'm Child\n");
} else {
  printf("Hello!\n");
  printf("I'm Parent\n");
}
```

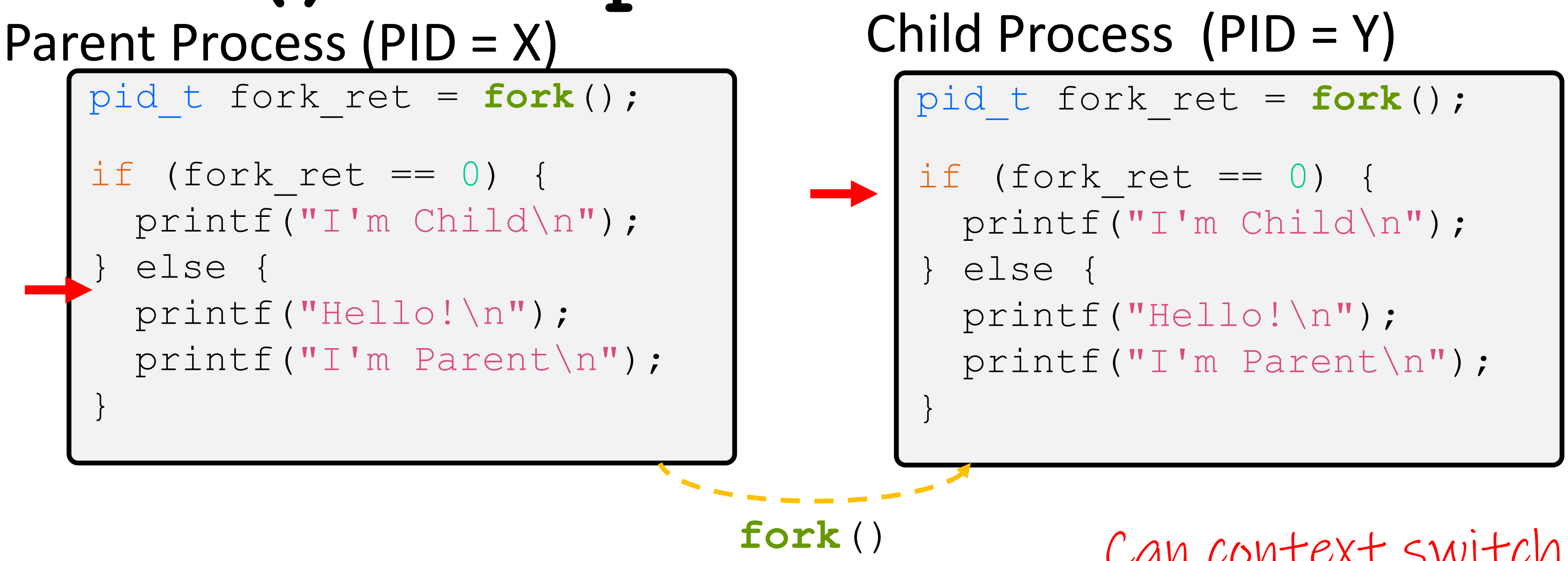
Parent process prints:
"Hello!“
and
"I’m Parent"

Child process prints:
"I’m Child"

What is the ordering of printing?

Non-deterministic

`fork()` example



Parent Process (PID = X)

```
pid_t fork_ret = fork();

if (fork_ret == 0) {
  printf("I'm Child\n");
} else {
  printf("Hello!\n");
  printf("I'm Parent\n");
}
```

Child Process (PID = Y)

```
pid_t fork_ret = fork();

if (fork_ret == 0) {
  printf("I'm Child\n");
} else {
  printf("Hello!\n");
  printf("I'm Parent\n");
}
```

What are the possible ordering of outputs?

1.
"Hello!"
"I'm Parent"
"I'm Child"

2.
"Hello!"
"I'm Child"
"I'm Parent"

3.
"I'm Child"
"Hello!"
"I'm Parent"

Can context switch to child at ANY time

Within a process, must follow sequential logic. (e.g., "Hello" <u>MUST</u> be printed before "I'm parent")

Poll Everywhere

pollev.com/tqm

- Are the following outputs possible?

```
pid_t fork_ret = fork();
if (fork_ret == 0) {
  fork_ret = fork();
  if (fork_ret == 0) {
    cout << "Hi 3!" << endl;
  } else {
    cout << "Hi 2!" << endl;
  }
} else {
  cout << "Hi 1!" << endl;
}
cout << "Bye" << endl;
```

Sequence 1:

```
Hi 1
Bye
Hi 2
Bye
Bye
Hi 3
```

Sequence 2:

```
Hi 3
Hi 1
Hi 2
Bye
Bye
Bye
```

A. No No

B. No Yes

C. Yes No

D. Yes Yes

E. We're lost…

Hint 1: there are three processes

Hint 2: Each prints out twice
"Hi" and "Bye"

Poll Everywhere

pollev.com/tqm

- Are the following outputs possible?

```
pid_t fork_ret = fork();
if (fork_ret == 0) {
  fork_ret = fork();
  if (fork_ret == 0) {
    cout << "Hi 3!" << endl;
  } else {
    cout << "Hi 2!" << endl;
  }
} else {
  cout << "Hi 1!" << endl;
}
cout << "Bye" << endl;
```

Sequence 1:	Sequence 2:
Hi 1	**Hi 3**
Bye	**Hi 1**
Hi 2	**Hi 2**
Bye	**Bye**
Bye	**Bye**
Hi 3	**Bye**

Hint #2 "Hi 3" must be before a "Bye"

A. [✓ No] No

B. [✓ No] Yes

C. Yes No

D. Yes Yes

E. We're lost…

Hint 1: there are three processes

Hint 2: Each prints out twice "Hi" and "Bye"

Hint 3: Events within a single process are "ordered normally"

Poll Everywhere

pollev.com/tqm

- Are the following outputs possible?

```
pid_t fork_ret = fork();
if (fork_ret == 0) {
  fork_ret = fork();
  if (fork_ret == 0) {
    cout << "Hi 3!" << endl;
  } else {
    cout << "Hi 2!" << endl;
  }
} else {
  cout << "Hi 1!" << endl;
}
cout << "Bye" << endl;
```

Sequence 1:	Sequence 2:
Hi 1	**Hi 3**
Bye	**Hi 1**
Hi 2	**Hi 2**
Bye	**Bye**
Bye	**Bye**
Hi 3	**Bye**

OK

Each "hi" comes before a "bye"

Order across processes not guaranteed

	Sequence 1	Sequence 2
A.	No	No
[✓ B.]	No	Yes
C.	Yes	No
D.	Yes	Yes
E.	We're lost…	

Hint 1: there are three processes

Hint 2: Each prints out twice "Hi" and "Bye"

Hint 3: Events within a single process are "ordered normally"

Waiting on a child Process

-
  ```
  pid_t waitpid(pid_t pid, int *wstatus,
                           int options);
  ```
  - Calling process waits for a child process (specified by **`pid`**) to exit
    - Also cleans up the child process
  - Gets the exit status of child process through output parameter **`wstatus`**
  - **`options`** are optional, pass in `0` for default options in most cases
  - Returns process ID of child who was waited for or `-1` on error

-
  ```
  pid_t wait(int *wstatus);
  ```
  - Equivalent of waitpid, but waits for ANY child

`sleep()`

- `unsigned int sleep(unsigned int seconds);`
  - Causes the calling *thread* to sleep until the number of real-time seconds specified elapses
    - (we will get to threads, for now think of it as acting on the calling process)
    - Can return early if it MUST be wakened up
    - Returns the number of seconds left to sleep
      - (0 if slept for specified time)
  - Useful as a brute-force way to "synchronize" things. Similar functions exist in most languages

Demo: `fork_example`

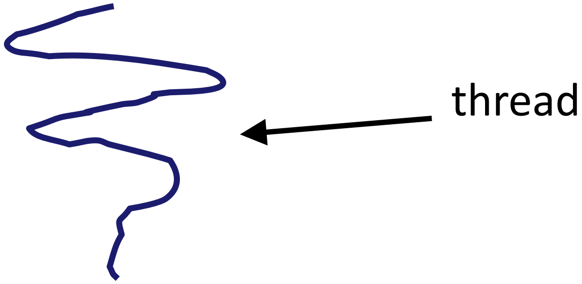
- See `fork_example.cc`
  - Brief code demo to see the various states of a process
    - Running
    - Zombie
    - Terminated
  - Makes use of `sleep()`, `waitpid()` and `exit()`!

Lecture Outline

- Review of Processes
- Interleavings
- Wait & Sleep
- **Threads**
- `pthreads`

Introducing Threads

- Separate the concept of a process from the "*thread of execution*"
  - Threads are contained within a process
  - Usually called a thread, this is a sequential execution stream within a process

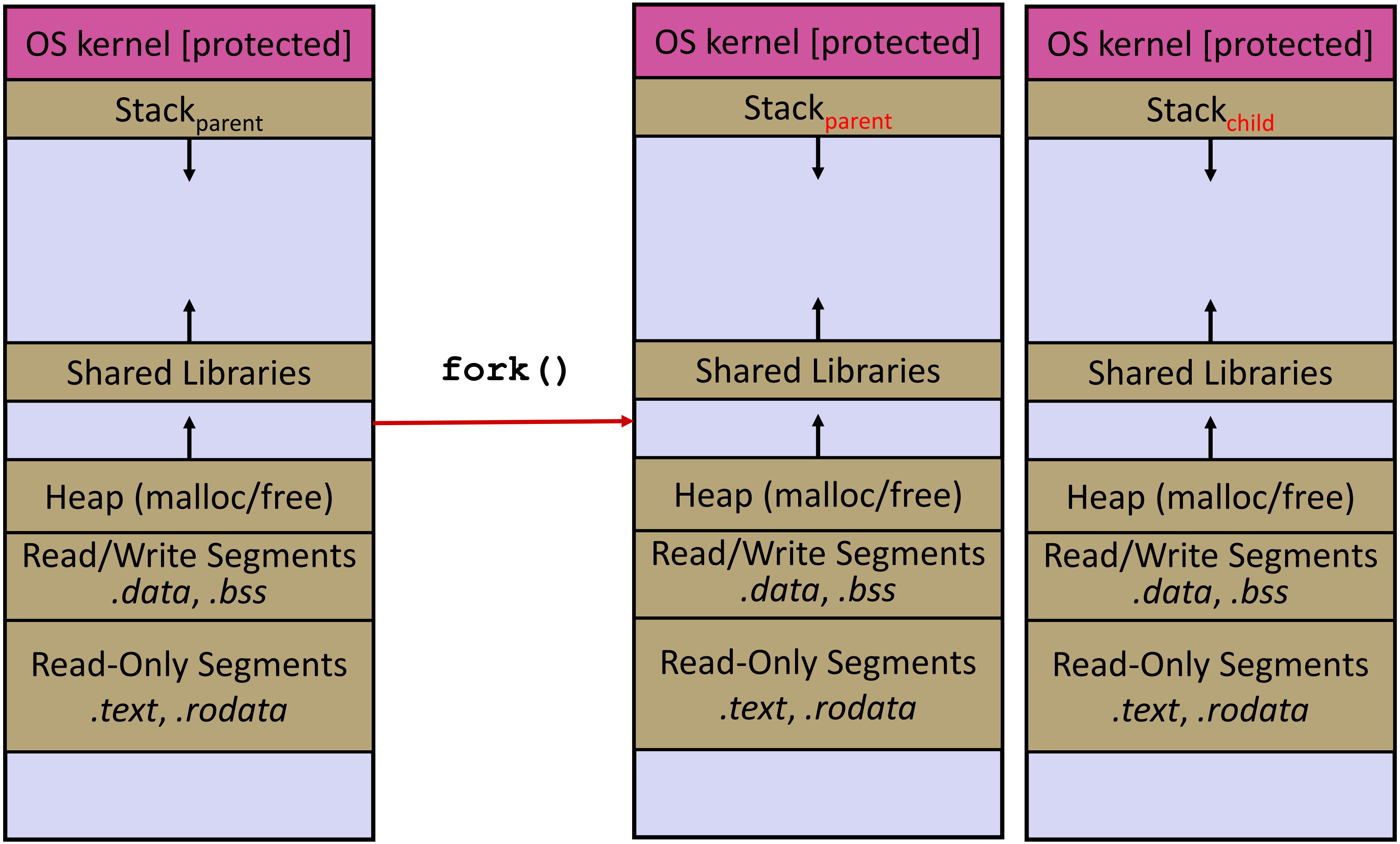


- In most modern OS's:
  - Threads are the *unit of scheduling.*

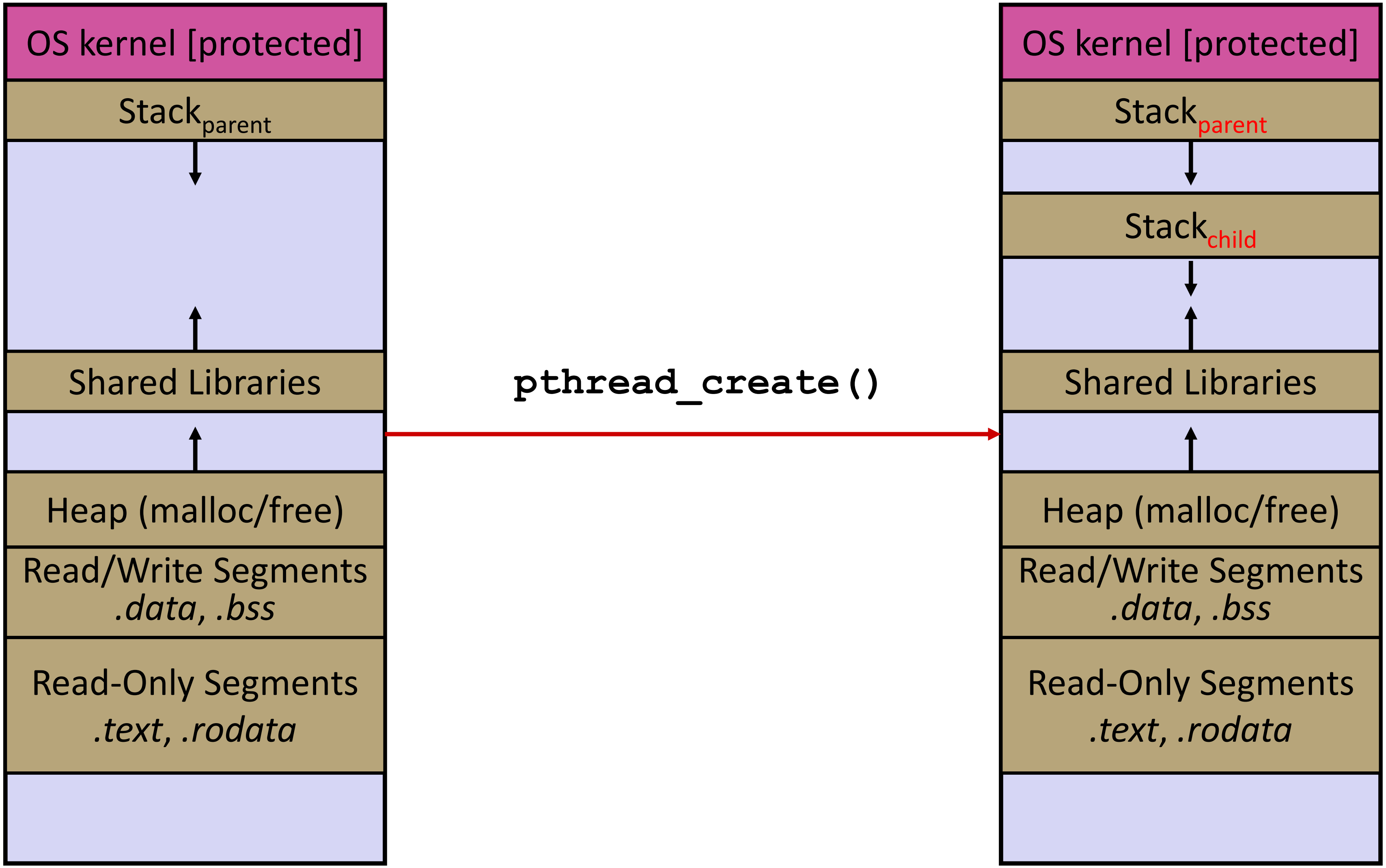
Threads vs. Processes

- In most modern OS's:
  - A Process has a unique: address space, OS resources, & security attributes
  - A Thread has a unique: stack, stack pointer, program counter, & registers
  - Threads are the *unit of scheduling* and processes are their *containers*; every process has at least one thread running in it

Threads vs. Processes



Threads vs. Processes

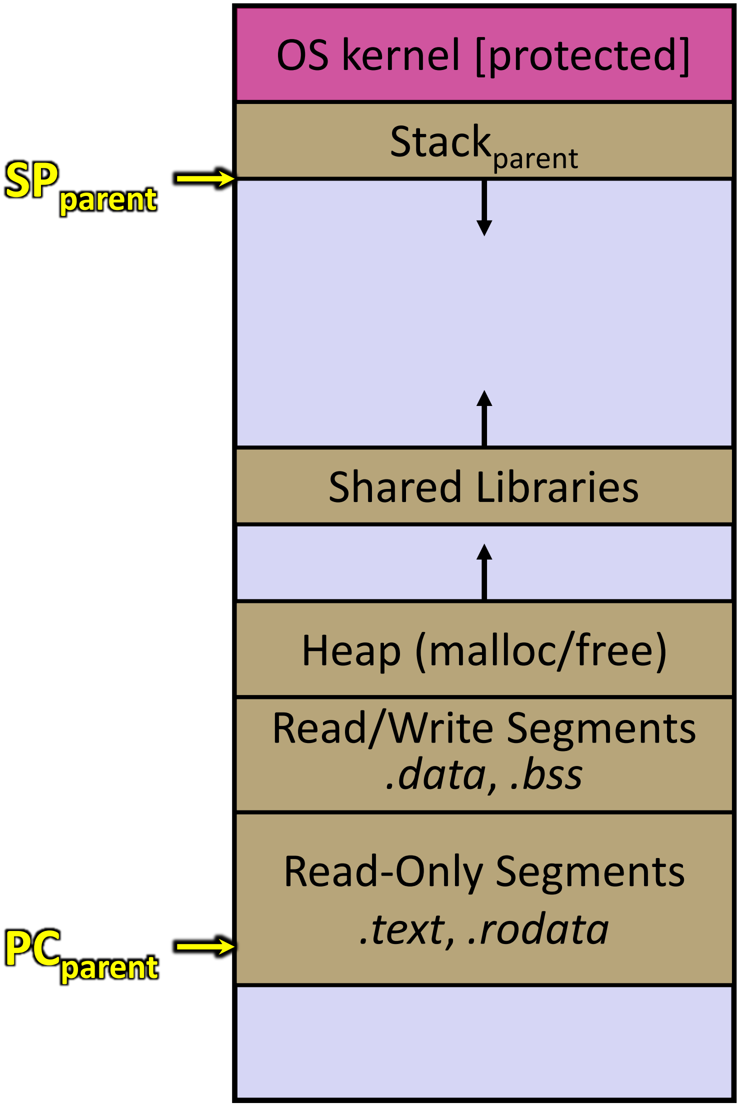


Threads

- Threads are like lightweight processes
  - They execute concurrently like processes
    - Multiple threads can run simultaneously on multiple CPUs/cores
  - Unlike processes, threads cohabitate the same address space
    - Threads within a process see the same heap and globals and can communicate with each other through variables and memory
      - But, they can interfere with each other – need synchronization for shared resources
    - Each thread has its own stack

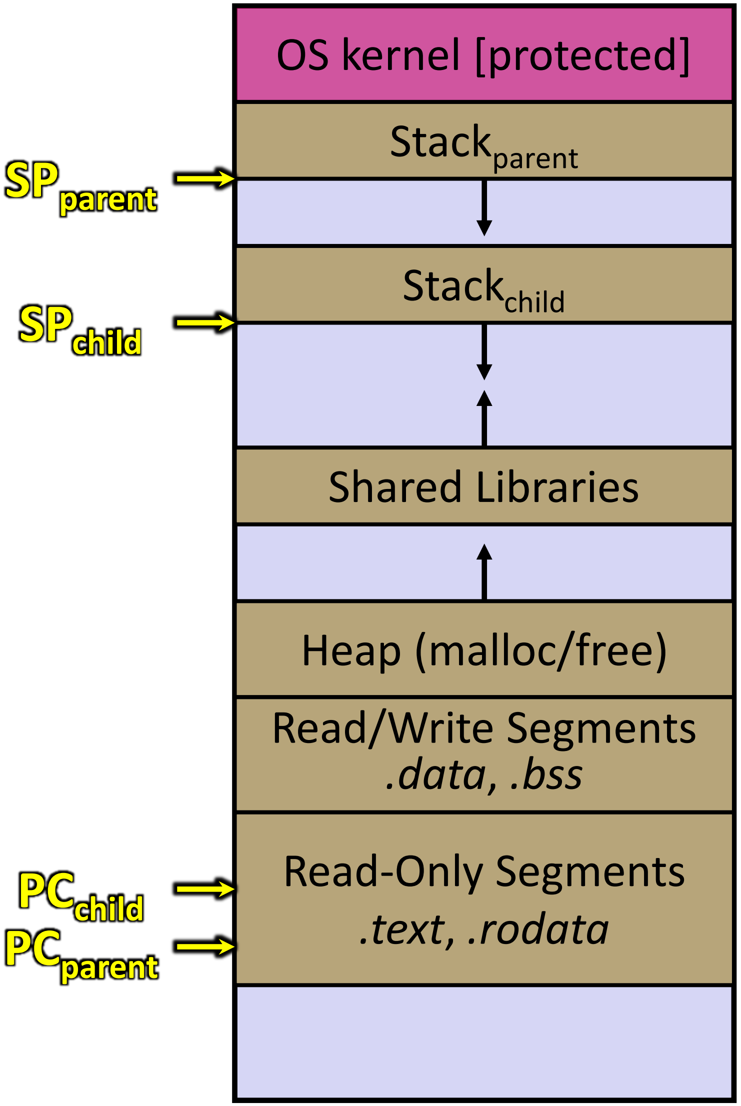
- Analogy: restaurant kitchen
  - Kitchen is process
  - Chefs are threads

Single-Threaded Address Spaces



- Before creating a thread
  - One thread of execution running in the address space
    - One PC, stack, SP
  - That main thread invokes a function to create a new thread
    - Typically `pthread_create()`

Multi-threaded Address Spaces



- After creating a thread
  - *Two* threads of execution running in the address space
    - Original thread (parent) and new thread (child)
    - New stack created for child thread
    - Child thread has its own *values* of the PC and SP
  - Both threads share the other segments (code, heap, globals)
    - They can cooperatively modify shared data

Lecture Outline

- Review of Processes
- OS as a scheduler
- Threads
- `pthreads`

POSIX Threads (pthreads)

- The POSIX APIs for dealing with threads
  - Declared in `pthread.h`
    - Not part of the C/C++ language
  - To enable support for multithreading, must include `-pthread` flag when compiling and linking with `gcc` command
    - `gcc -g -Wall -std=c11 -pthread -o main main.c`
  - Implemented in C
    - Must deal with C programming practices and style

Creating and Terminating Threads

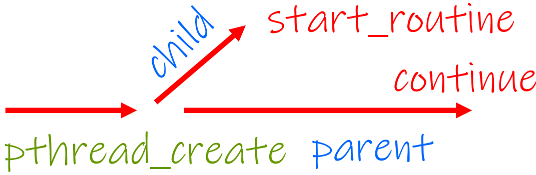
Output parameter.
Gives us a "thread_descriptor"

```
int pthread_create(
            pthread_t* thread,
            const pthread_attr_t* attr,
            void* (*start_routine)(void*),
            void* arg);
```

Function pointer!
Takes & returns void*
to allow "generics" in C

Argument for the thread function

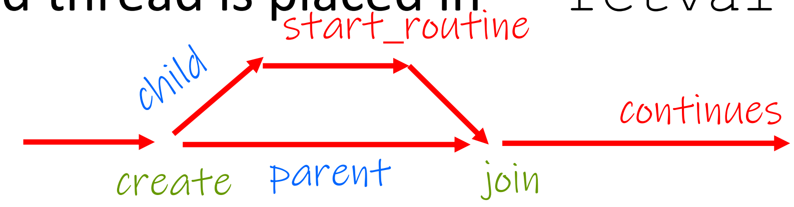
- Creates a new thread into `*thread`, with attributes `*attr` (`NULL` means default attributes)
- Returns **0** on success and an error number on error (can check against error constants)
- The new thread runs **`start_routine`**`(arg)`



What To Do After Forking Threads?

- `int pthread_join(pthread_t thread, void** retval);`
  - Waits for the thread specified by `thread` to terminate
  - The thread equivalent of **`waitpid`**`()`
  - The exit status of the terminated thread is placed in `**retval`

Parent thread waits for child thread to exit, gets the child's return value, and child thread is cleaned up



Thread Example

- See `cthreads.c`
  - How do you properly handle memory management?
    - Who allocates and deallocates memory?
    - How long do you want memory to stick around?