



CSE 451: Operating Systems, 26au

L6: Process Syscalls, Threads

Instructor: Naomi Alterman

TAs:

Jonathan Trinh (Head)

Mahir Emran

Ted Guan

Jai Rungrachakanont

Soham Raut

Sagar Saby

Andrew Wood

Adapted from slides by Tom Anderson

Administrivia

- The old lab swapperroo

- Lab 1 due

- Lab 2 out

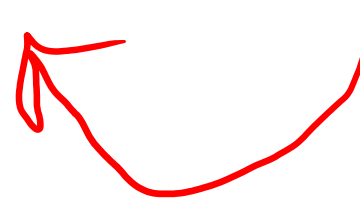
- Part 1 design doc due this Friday 10/16 **6 PM**

- Part 1 code due next Friday, 10/23 at midnight 🍬 👻

- Part 2 comes after all that

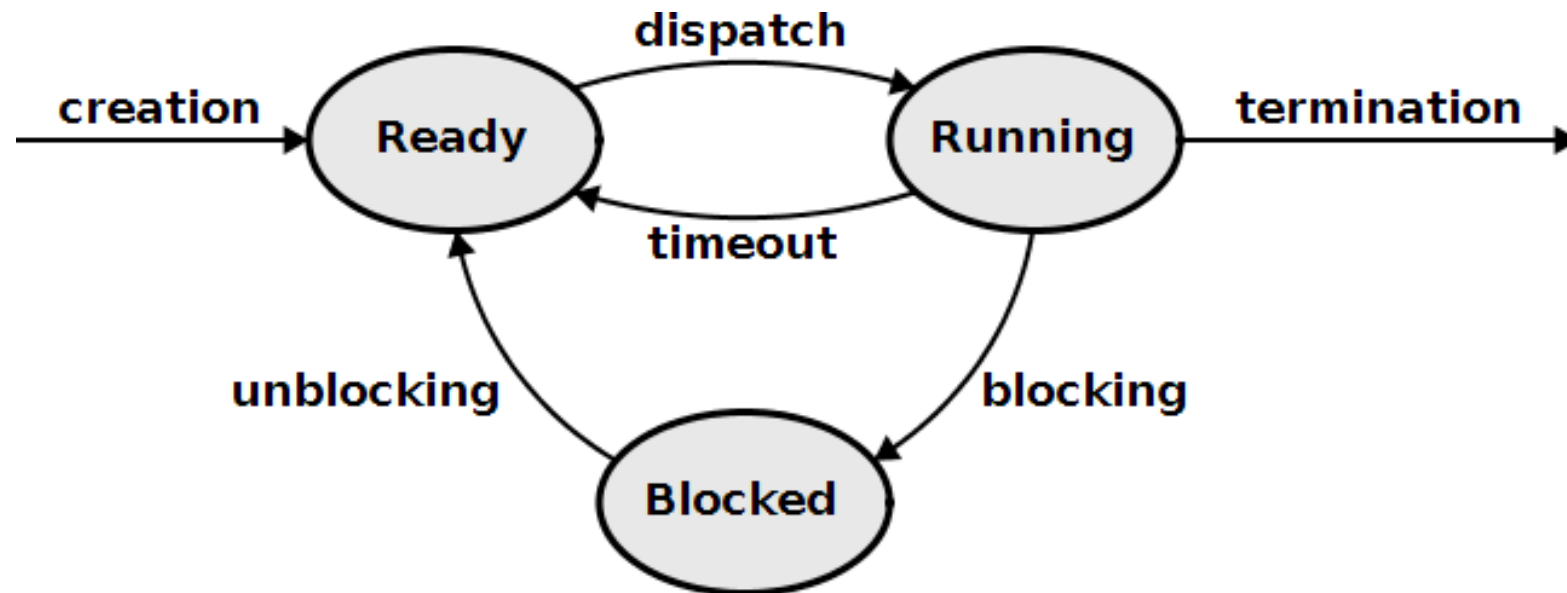
Naomi:

fix website!



Processes

- Memory arranged into an address space (stacks, a heap, program code, the classics)
- Threads (paths of execution through the program code)
- Private metadata structures in the kernel to keep track of all that stuff



But where do processes *come from?*

Formats and bootstrapping

Shells



User applications whose purpose is to start other programs

- ✳ Often in a way that is scriptable, with its own language
- 🔊 And, just as often, in a way that is graphical

```
$ grep "spaghetti" recipes.txt # run grep, output to stdout
$ grep "spaghetti" recipes.txt > search_results # run grep, output to search_results
$ grep "spaghetti" recipes.txt > search_results & # same, but do it in the background
$ grep "spaghetti" recipes.txt | wc -l # run grep and wc, stdout of grep is
# stdin of wc
```

What do shells need from the OS?

```
# run grep, output to stdout  
$ grep "spaghetti" recipes.txt
```

```
# run grep, output to search_results  
$ grep "spaghetti" recipes.txt > search_results
```

```
# same, but do it in the background  
$ grep "spaghetti" recipes.txt > search_results &
```

```
# run grep and wc, stdout of grep is stdin of wc  
$ grep "spaghetti" recipes.txt | wc -l
```

UNIX Process API

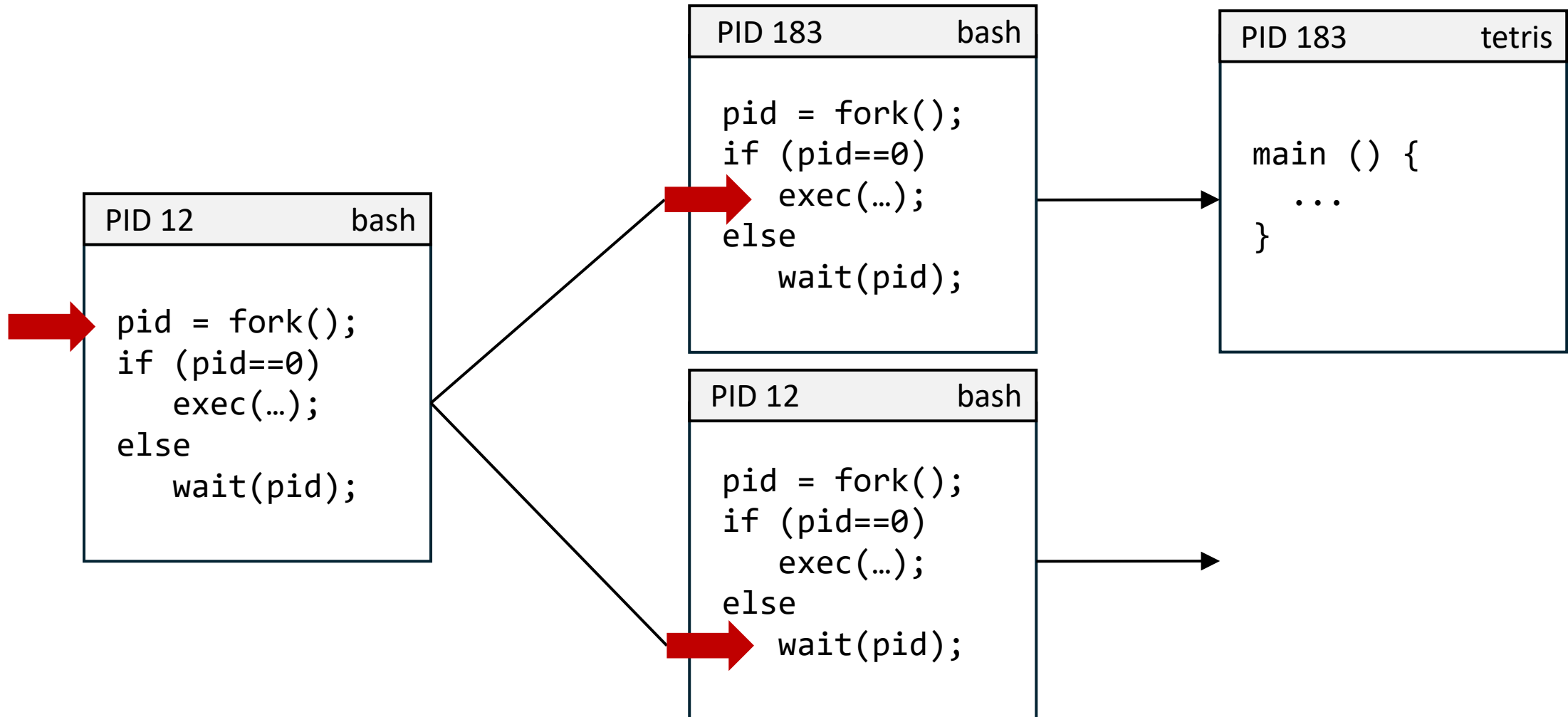
- **fork()**
create a copy of the current process
- **exec(file, arg1, arg2, ...)**
wipe current process's memory and replace it with a new program
(with specified arguments to the new `main()`)
- **wait(pid)**
block until process `pid` finishes
- **exit(status)**
close the current process, returning `status` to parent

(Aside: Windows CreateProcess API)

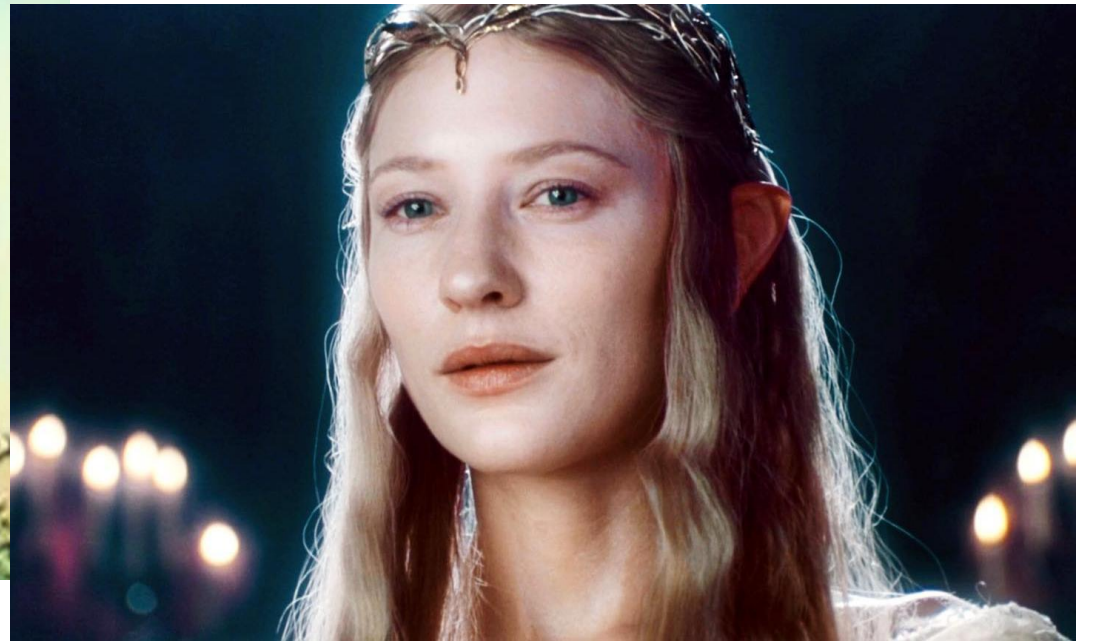
↓ type hints

```
BOOL CreateProcessA(  
    LPCSTR          lpApplicationName,    // [in, optional]  
    LPSTR           lpCommandLine,       // [in, out, optional]  
    LPSECURITY_ATTRIBUTES lpProcessAttributes, // [in, optional]  
    LPSECURITY_ATTRIBUTES lpThreadAttributes, // [in, optional]  
    BOOL            bInheritHandles,      // [in]  
    DWORD           dwCreationFlags,      // [in]  
    LPVOID          lpEnvironment,       // [in, optional]  
    LPCSTR          lpCurrentDirectory,   // [in, optional]  
    LPSTARTUPINFOA  lpStartupInfo,       // [in]  
    LPPROCESS_INFORMATION lpProcessInformation // [out]  
);
```

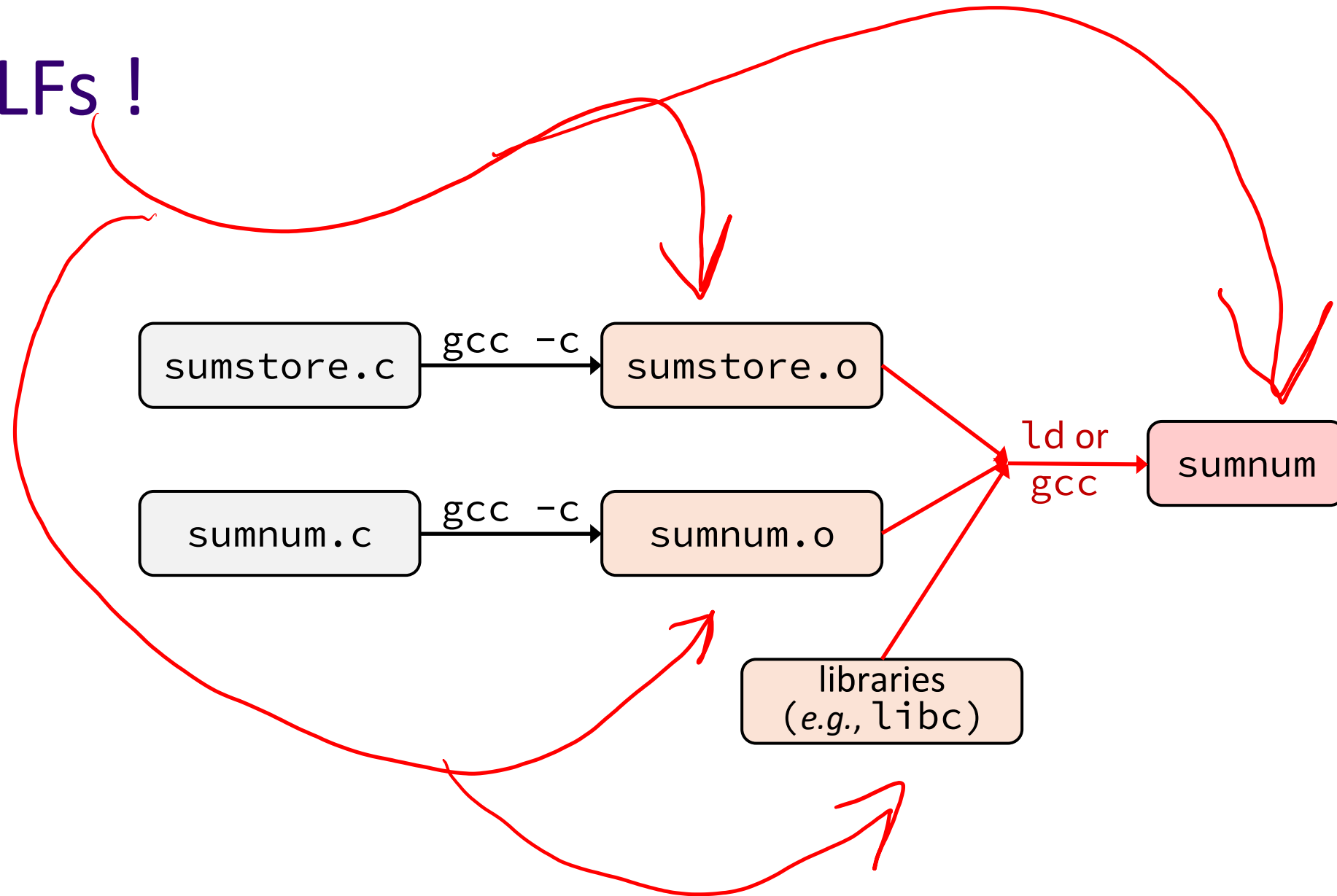
UNIX Process Management



ELFs ?



ELFs !



ELFs .

- **Executable and Linking Format**
 - File format for efficiently storing the memory layout of a computer program (or part of a computer program) on disk
 - Consists of arrays of:
 - **memory segments** (actual data: machine code, static variables)
 - **program headers** (where that data should go in virtual memory)
 - **section headers** (metadata about that data's purpose)
 - Three primary uses:
 - **Executables** – Programs! Apps!
 - **Relocatables** – Fragments of programs, statically linked libraries
 - **Dynamic Libraries** – Libraries linked by the OS at program load
- See also: Windows PE (**P**ortable **E**xecutable)

Starting a new process, xk style

- `fork()`:
 - Create and initialize the PCB
 - Assign each process a unique ID (process ID)
 - Create and initialize a new address space
- `exec("elf file", args...)`:
 - Load the program from its ELF into the address space
 - Copy command line arguments into the address space
 - Set up the start of the stack by hand
 - Set up a trap frame that will lead the CPU into `main()`
 - Inform the scheduler that the new process is ready to run



Questions

- In what situations could fork() fail?
 - :(){ :|:& };:  fork bomb

- In what situations could exec() fail?

- Can UNIX wait() ever return immediately? Why?

Fork() {
 Fork() |
 fork() }
 }
fork

UNIX fork performance

- Make copy of the entire address space? Really?
 - What if the process is pretty large?
- Instead: virtual copy of address space
 - Make a copy of the page table
 - Both parent and child point to the same physical pages
 - Mark both parent and child pages as “read-only”
- What if parent or child wants to modify memory (eg, stack)?
 - Hardware will page fault (write to read-only page)
 - Kernel can **copy-on-write** – make a copy of each page that’s modified
 - Resume execution with page table entry as “read-write”

Riddle: What does this code print?

```
int child_pid = fork();

if (child_pid == 0) {
    // I'm the child process
    printf("I am process #%d\n", getpid());
} else {
    // I'm the parent process
    printf("I am parent of #%d\n", child_pid);
}

return 0;
```



Inter-Process Communication

What do shells need from the OS?

```
# run grep, output to stdout  
$ grep "spaghetti" recipes.txt
```

```
# run grep, output to search_results  
$ grep "spaghetti" recipes.txt > search_results
```

```
# same, but do it in the background  
$ grep "spaghetti" recipes.txt > search_results &
```

```
# run grep and wc, stdout of grep is stdin of wc  
$ grep "spaghetti" recipes.txt | wc -l
```

Base case: create a process with arguments

```
$ grep "spaghetti" recipes.txt
```

```
void cmd (char *prog, char **args) {  
    int child_pid = fork();  
    if (child_pid == 0) {  
        → exec(prog, args);  
    } else {  
        wait(child_pid);  
    }  
}
```

Redirect output to a file !



```
$ grep "spaghetti" recipes.txt > search_results
```

```
void cmdWithOutputRedirection (char *prog, char **args, char *out_filename) {  
    → int child_pid = fork();  
        if (child_pid == 0) {  
            → int fd = open(out_filename);  
            → dup2(fd, stdout);  
                exec(prog, args);  
        } else {  
            wait(child_pid);  
        }  
    }  
}
```

Pipes

- **Idea:**
 - We *could* do IPC by having a parent thread read from one child process FD and write to another
 - But this wastes time and memory with copies!
 - Instead, OS provides a shared buffer called a “**pipe**”
- **OS pipe() syscall**
 - Creates a ring buffer that can be accessed by a process and its children
 - Returns **two FDs**:
 - One for writing data into the buffer
 - One for reading data from the buffer

Piping data

```
$ grep "spaghetti" recipes.txt | wc -l
```

```
void cmdsWithPipe (char *prog1, char **args1, char *prog2, char **args2) {  
    int childPids[2], pipeFds[2];  
    pipe(&pipeFds);  
    if ((child[0] = fork()) == 0) {  
        dup2(fd[1], stdout);  
        exec(prog1, args1);  
    } else if (child2 = fork()) == 0) {  
        dup2(fd[0], stdin);  
        exec(prog2, args2);  
    } else {  
        wait(child1); wait(child2);  
    }  
}
```

Handwritten annotations:

- grep* (under `prog1`)
- wc* (under `prog2`)
- "write" fd* (pointing to `fd[1]`)
- "read" fd* (pointing to `fd[0]`)

Class concluded here on 10/12
and will resume on 10/14



Threads

Concurrency: Multiprocessing

- We've been discussing a form of **concurrency** called **multiprocessing**
 - Our OS gets multiple things done concurrently by giving each task its own virtual address space and periodically switching the context with a timer interrupt
- What if we don't need all that memory isolation?
- What if we need to **not** have memory isolation?
 - Electron apps (eg, vscode) work this way: one thread runs a javascript interpreter, the other thread runs native C code to do I/O

Concurrency: Threads

- Allow multiple tasks to share the same address space
 - Each task is called a **thread**
 - Each thread has **its own stack**
 - All threads in a process **share** the **heap, globals and code segment**
 - Kernel threads run in kernel space, but otherwise the same principle
 - Thread **scheduling is handled by the OS kernel**, just like processes

Déjà vu?

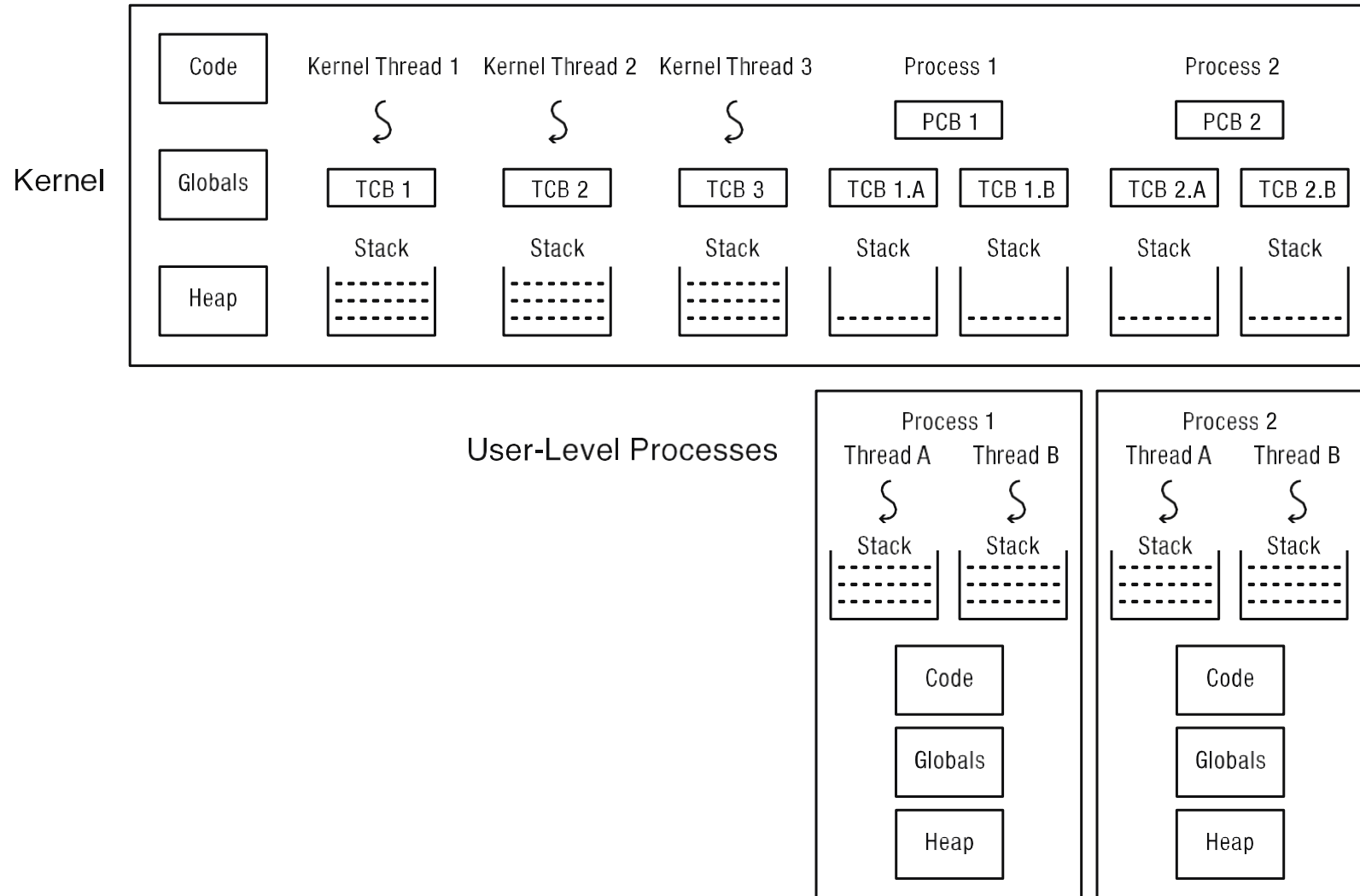
Didn't we learn all about concurrency in CSE 332 and CSE 333?



Déjà vu?

- Didn't we learn all about concurrency in CSE 332/333?
 - More practice
 - Realistic examples, especially in the project
 - Design patterns and pitfalls
 - Methodology for writing correct concurrent code
 - Implementation
 - How do threads work at the machine level?
 - CPU scheduling
 - If multiple threads to run, which do we do first?

Threading model



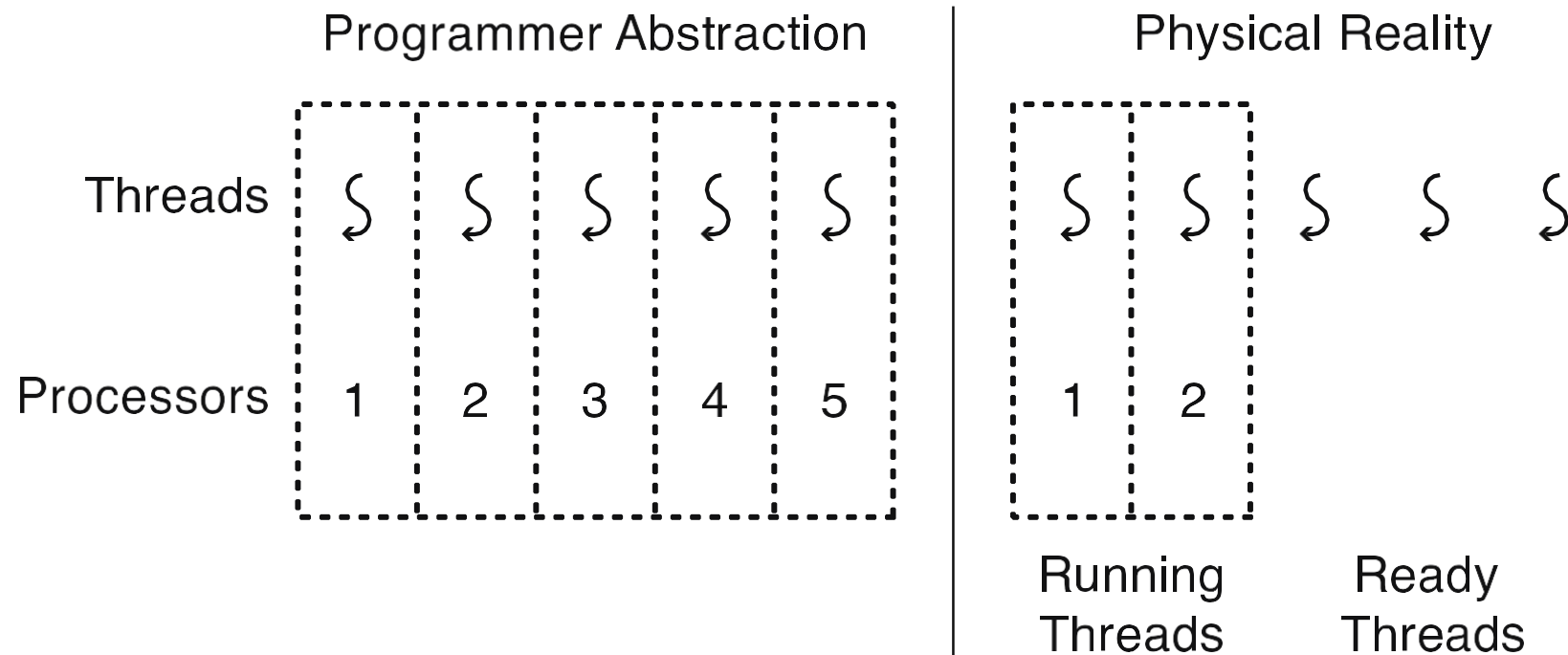


OS as referee

- What if a thread puts too many procedures on its stack?
 - What happens in Java?
 - What happens in the Linux kernel?
 - What happens in xk?
 - What *should* happen?

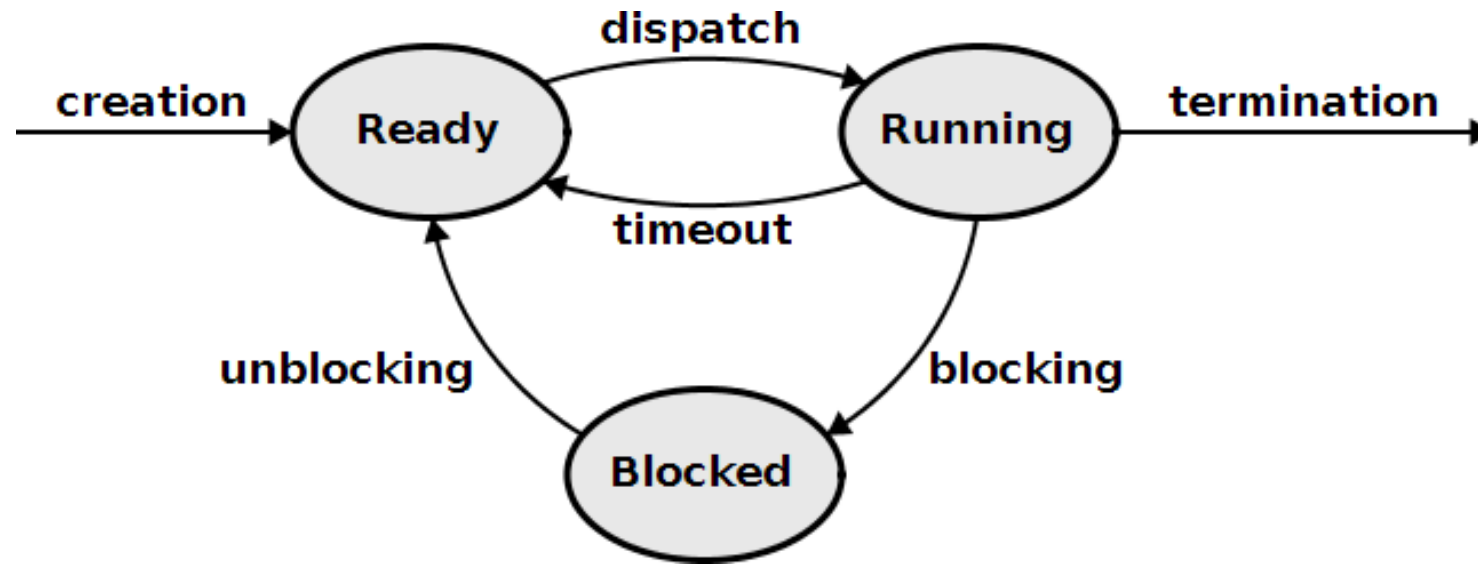
🎩 OS as illusionist, but only to an extent...

- Infinite number of processors
- Threads execute with variable speed
- Programs must be designed to work with any schedule



~~Process~~

Thread Lifetime



Thread Syscalls

- **thread_create(thread, func, args)**
 - Create a new thread to run func(args)
- **thread_yield()**
 - Relinquish CPU time voluntarily
- **thread_join(thread)**
 - In parent, wait for child thread to exit, then return
- **thread_exit()**
 - Quit thread and clean up, wake up joiner if any

Implementing threads

- **thread_create(func, args)**
 - Allocate thread control block (**TCB**)
 - Allocate user-space stack
 - Build stack frame for base of user stack (stub)
 - Put func, args on stack
 - Put thread on ready list
 - Will run sometime later (maybe right away? maybe not!??)
- **stub(func, args):**
 - Call (*func)(args)
 - If func returns, call thread_exit()

Planting a thread

- Setting up a new thread by hand is a little subtle
- We don't technically start at in the thread's function; we start with an OS-installed wrapper
 - The init process does this too! See “preface” in `user/lib/preface.S`
- We enter the stub with **swtch.S**, passing in a hand-constructed trap frame that “resumes” into the new thread

Planting a thread

```
int main () {  
    // ...  
    thread_create(&fun, args);  
    // ...  
}
```

1 →

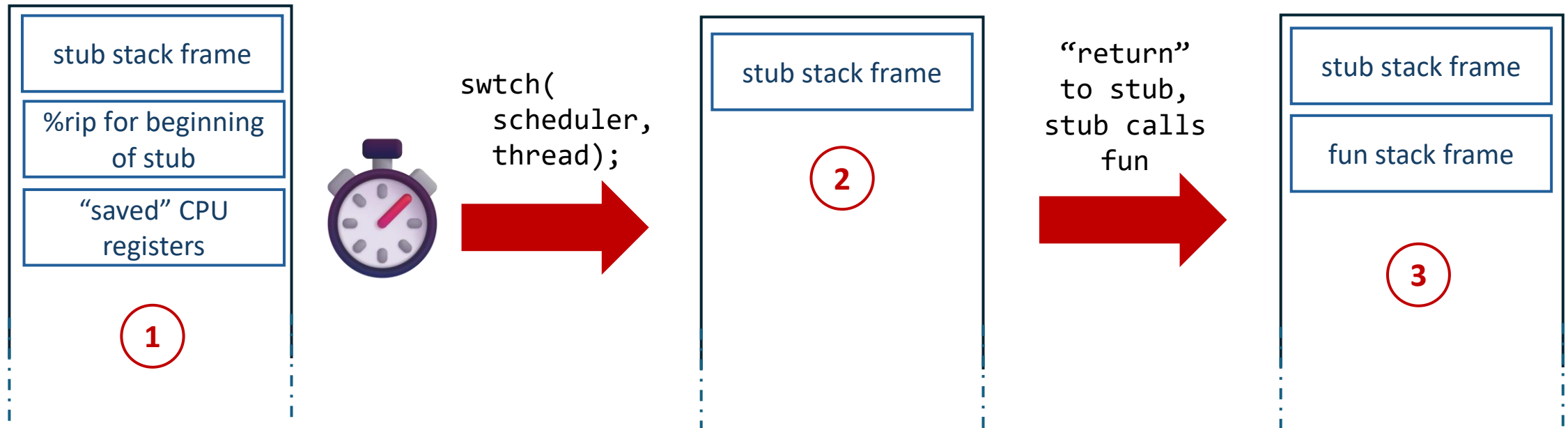
stub:

3 → `call fun`
`call thread_exit`

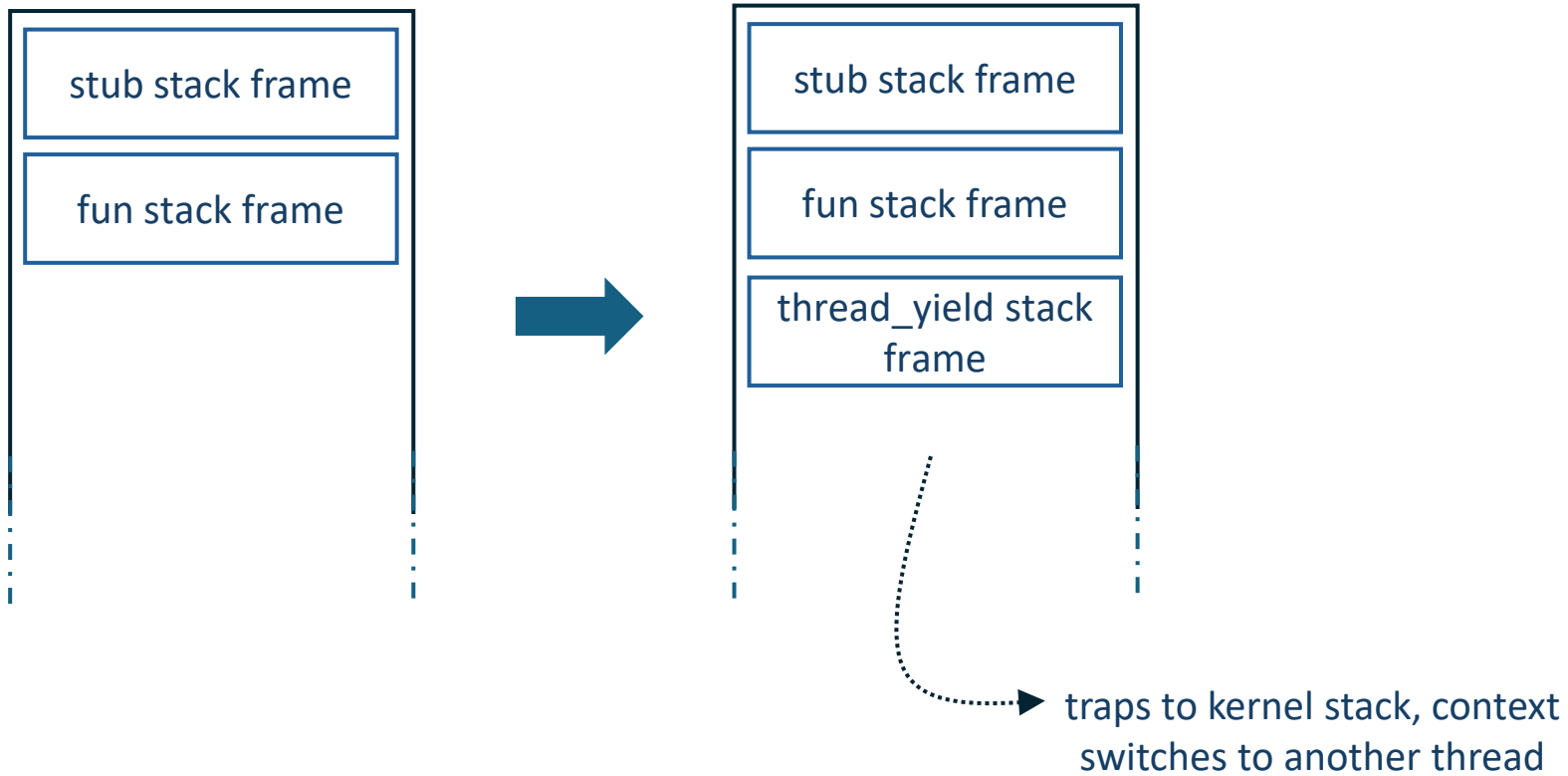
swtch:

```
# ..save scheduler regs..  
mov %rsp, (%rdi)  
mov %rsi, %rsp  
# ..pop stub regs..
```

2 →



Yielding a thread



Two Threads Call Yield

Thread 1's instructions

"return" from thread_switch
into stub
call go
call thread_yield
choose another thread
call thread_switch
save thread 1 state to TCB
load thread 2 state

return from thread_switch
return from thread_yield
call thread_yield
choose another thread
call thread_switch

Thread 2's instructions

"return" from thread_switch
into stub
call go
call thread_yield
choose another thread
call thread_switch
save thread 2 state to TCB
load thread 1 state

Processor's instructions

"return" from thread_switch
into stub
call go
call thread_yield
choose another thread
call thread_switch
save thread 1 state to TCB
load thread 2 state
"return" from thread_switch
into stub
call go
call thread_yield
choose another thread
call thread_switch
save thread 2 state to TCB
load thread 1 state
return from thread_switch
return from thread_yield
call thread_yield
choose another thread
call thread_switch

Synchronization

Haha uh oh

- **NB:** concurrency is hard
 - Humans are not very good at keeping track of multiple things happening simultaneously
 - *Any* concurrency (processes, threads) means multiple syscalls can be running at once
 - Multiple syscalls running at once means we need to protect shared kernel data from getting corrupted

Programmer vs. Processor View

Programmer's View

.
.
.
x = x + 1;
y = y + x;
z = x + 5y;
.
.
.

Possible Execution #1

.
.
.
x = x + 1;
y = y + x;
z = x + 5y;
.
.
.

Possible Execution #2

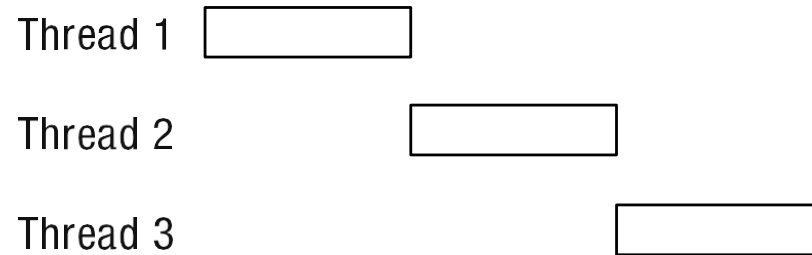
.
.
.
x = x + 1;
.....
Thread is suspended.
Other thread(s) run.
Thread is resumed.
.....
y = y + x;
z = x + 5y;

Possible Execution #3

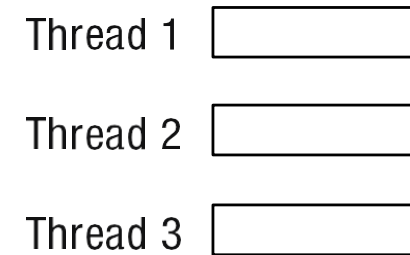
.
.
.
x = x + 1;
y = y + x;
.....
Thread is suspended.
Other thread(s) run.
Thread is resumed.
.....
z = x + 5y;

Possible Executions

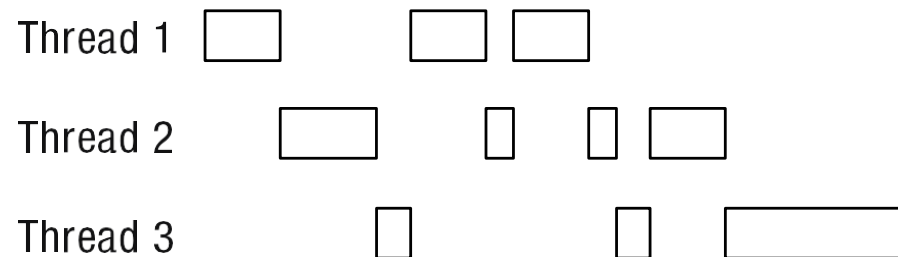
One Execution



Another Execution



Another Execution



Example: threadHello

```
#define NTHREADS 10
thread_t threads[NTHREADS];
Void main() {
    for (i = 0; i < NTHREADS; i++) {
        thread_create(&threads[i], &go, i);
    }
    for (i = 0; i < NTHREADS; i++) {
        exitValue = thread_join(threads[i]);
        printf("Thread %d returned with %ld\n", i, exitValue);
    }
    printf("Main thread done.\n");
}
void go (int n) {
    printf("Hello from thread %d\n", n);
    thread_exit(100 + n);
}
```

```
bash-3.2$ ./threadHello
Hello from thread 0
Hello from thread 1
Thread 0 returned 100
Hello from thread 3
Hello from thread 4
Thread 1 returned 101
Hello from thread 5
Hello from thread 2
Hello from thread 6
Hello from thread 8
Hello from thread 7
Hello from thread 9
Thread 2 returned 102
Thread 3 returned 103
Thread 4 returned 104
Thread 5 returned 105
Thread 6 returned 106
Thread 7 returned 107
Thread 8 returned 108
Thread 9 returned 109
Main thread done.
```