



CSE 451: Operating Systems, 26au

L5: Processes

Instructor: Naomi Alterman

TAs:

Jonathan Trinh (Head)

Mahir Emran

Ted Guan

Jai Rungrachakanont

Soham Raut

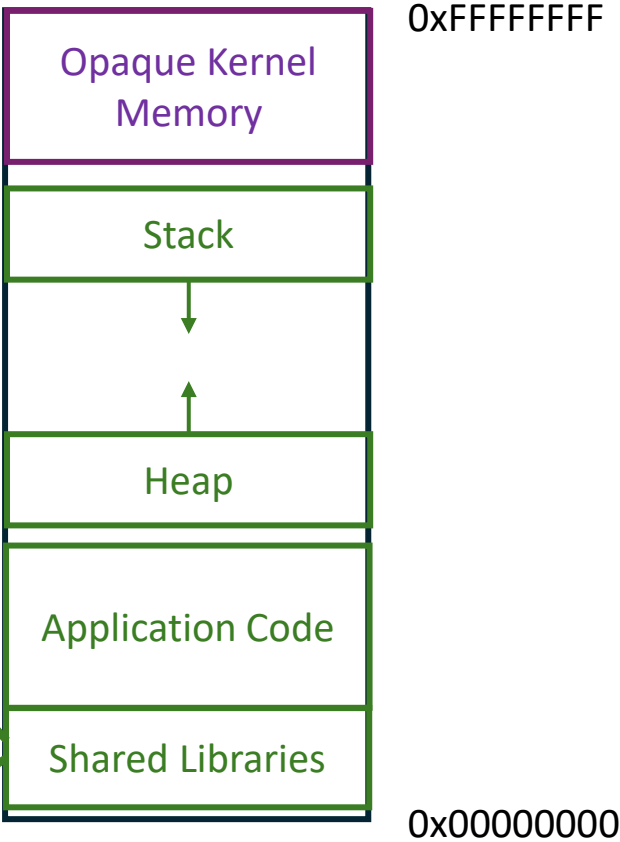
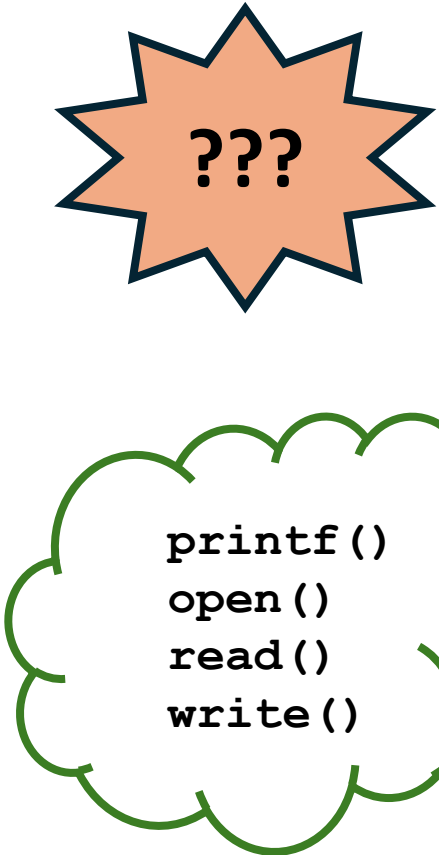
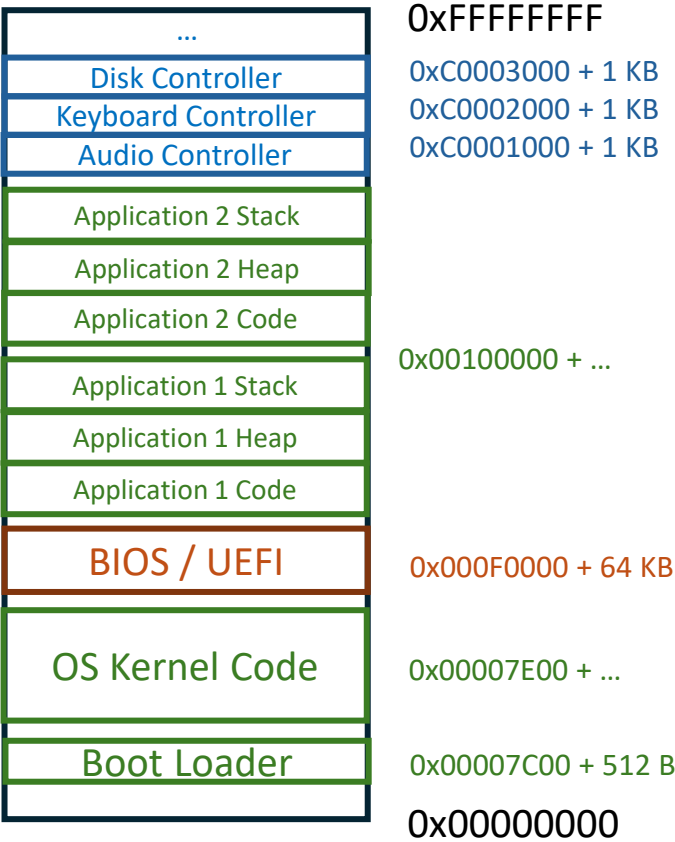
Sagar Saby

Andrew Wood

Adapted from slides by Tom Anderson

Physical Address Space

Process Memory (“user space”)



Review: The Process Abstraction

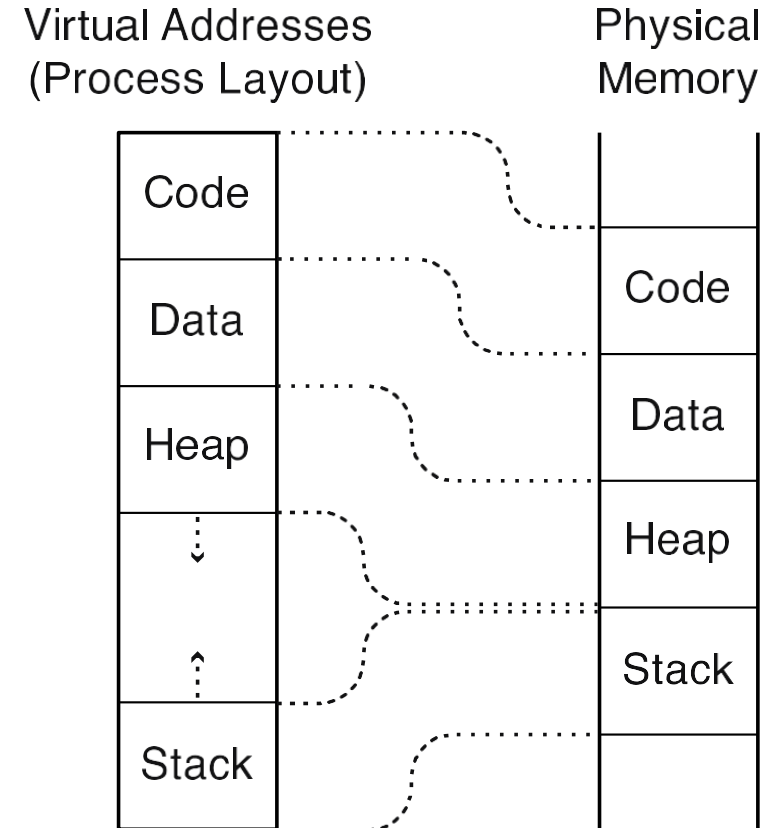
- Process: a *single running instance* of a program on a computer
- Consists of:
 - Memory arranged into an address space (stacks, a heap, program code, the classics)
 - Threads (paths of execution through the program code)
 - Private metadata structures in the kernel to keep track of all that stuff

Virtual Memory

(a very brief introduction)

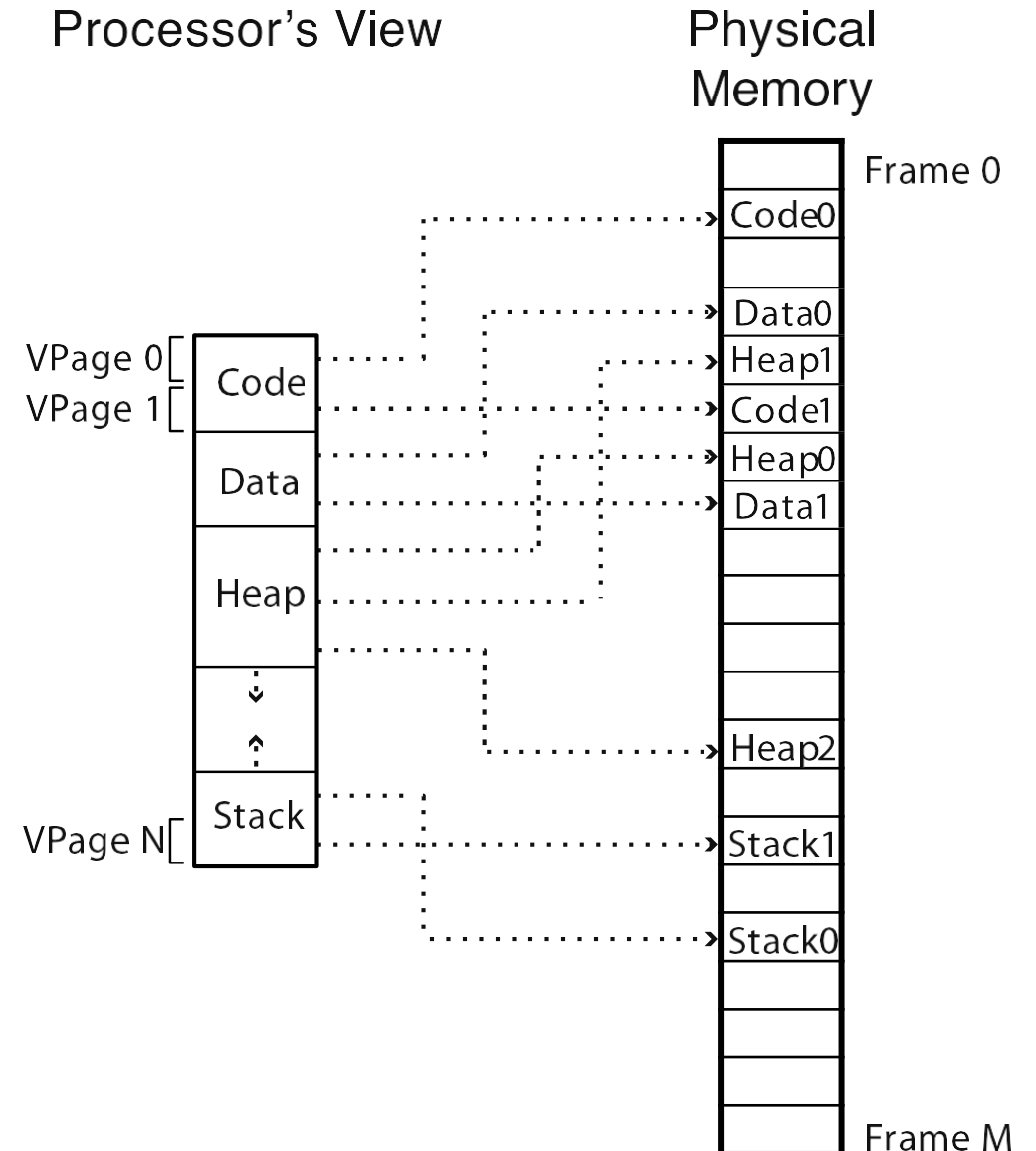
Virtual Addresses

- Translation done in hardware, using a table
 - On every instruction!
- Table set up by OS kernel
 - Eg, a page table points to fixed size page frames
 - Or a multi-level page table to handle very large or sparse virtual address space



Virtual Addresses

- Translation done in hardware, using a table
 - On every instruction!
- Table set up by OS kernel
 - Eg, a page table points to fixed size page frames
 - Or a multi-level page table to handle very large or sparse virtual address space



Benefits of Virtual Memory Translation

- Dynamically sized stack and heap
- Unmapped regions between valid ones to catch bugs
 - What if many threads? Need many stacks with many spaces between them
- Dynamically linked libraries
 - (Shared code regions between processes)
- “Zero copy” shared data between processes
- Shared data memory between chips/servers in the cloud
 - Remote DMA (RDMA)
- Memory mapped files

Experiment to see if our code lives in The Matrix

```
int staticVar = 0;          // a static variable
void main() {
    int localVar = 0; // a procedure local variable

    staticVar += 1;
    localVar += 1;
    sleep(10);           // sleep for x seconds
    printf ("static address: 0x%08x, value: %d\n",
            &staticVar, staticVar);
    printf ("local address: 0x%08x, value: %d\n",
            &localVar,  localVar);
}
```

Virtual Address != Physical Address

- The **same virtual address** in two different processes can refer to **different physical addresses...why?**
- The **same virtual address** in two different processes can refer to the **same physical address...why?**
- **Different virtual addresses** can refer to the **same physical address...why?**

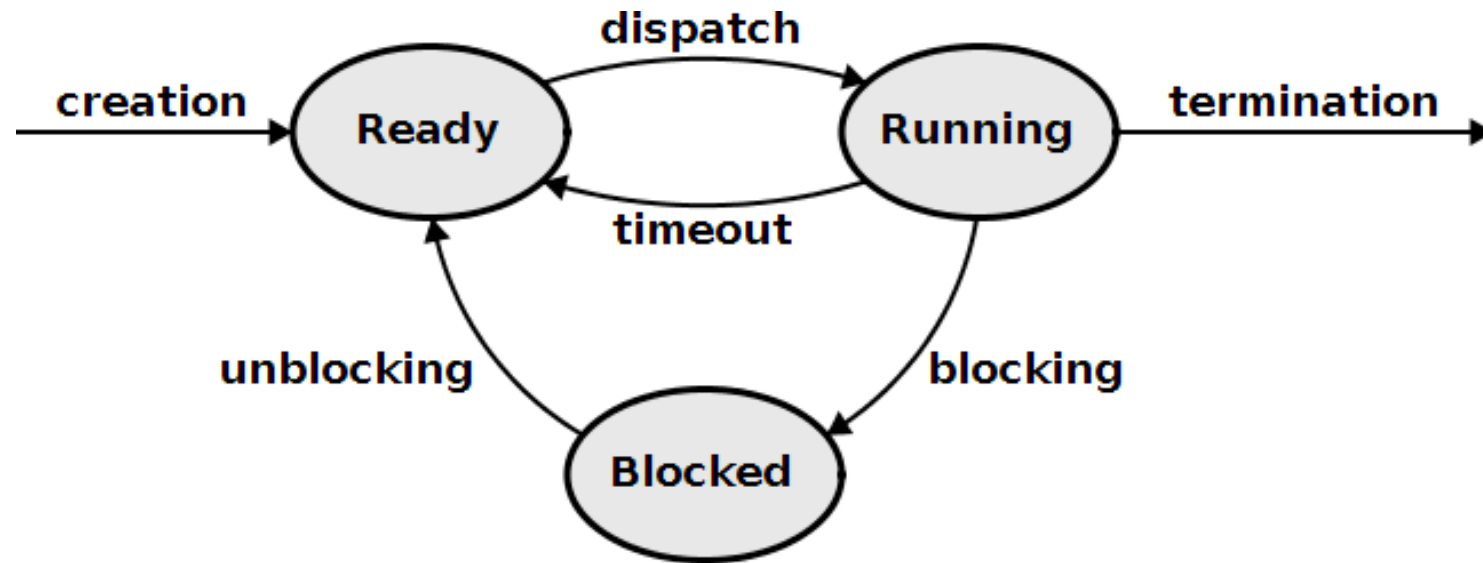
Process Lifetime

Structures and strategies in the kernel

Review: The Process Abstraction

- Process: a *single running instance* of a program on a computer
- Consists of:
 - 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

Process Lifetime



The scheduler

- A part of the kernel that chooses who gets the CPU next
 - Choosing among every thread in every user process, along with kernel threads
- In xk:
 - “Round robin” scheduler (we just loop in some order)
 - The scheduler itself acts sort of like a little tiny process
 - Has its own stack and context
 - Programs block or get kicked off the CPU, execution transfers to scheduler
 - Scheduler chooses a new program to get control
- More on this in a few weeks...

Let's get gritty

- `proc.h` – For each process the kernel tracks in a Process Control Block (“**PCB**”):
 - Physical<->virtual memory mappings (“**vspace**”)
 - A pointer to the kernel stack
 - The state of the process (runnable? sleeping?)
 - The saved CPU state, if the process is asleep (“**context**”)
 - A trap frame, if the process is in a syscall (“**tf**”)
- `proc.c` – The scheduler
- `switch.S` – ✦ switches the active process ✦



Class concluded here on 10/9
and will resume on 10/12

swtch(&university()->classes, p->weekend);



But where do processes *come from?*

Formats and bootstrapping

Shells

A user application that runs other programs

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

```
% grep "To be or not" Shakespeare.txt // run grep, output to stdout
% grep "To be or not" Shakespeare.txt > logfile // run grep, output to logfile
% grep "To be or not" Shakespeare.txt > logfile & // same, but do it in the background
% grep "To be or not" Shakespeare.txt | wc // run grep and wc, output of grep
// goes to input of wc
```

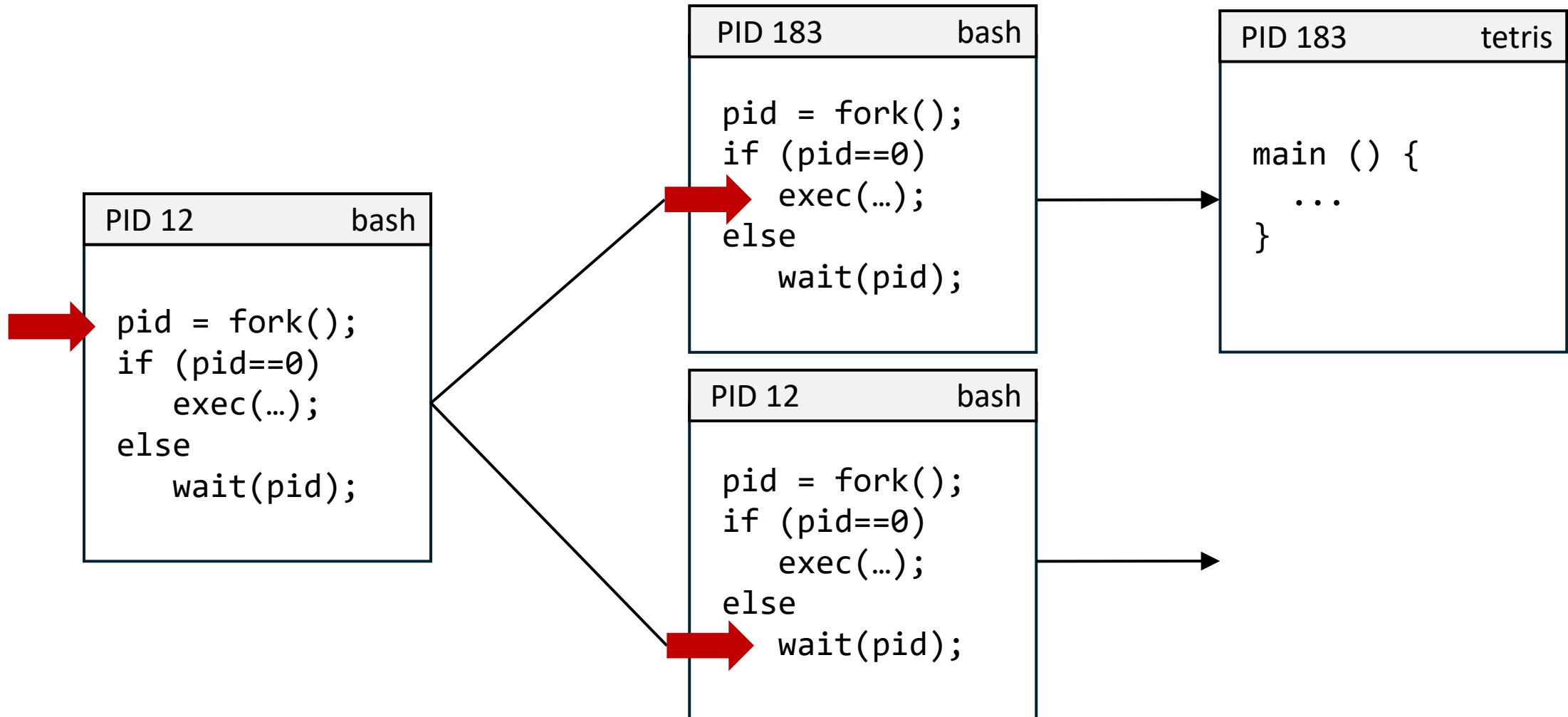
UNIX Process API

- `fork()` – create a copy of the current process, and start it running
 - No arguments!
- `exec()` – replace current process with another ELF
 - Pass the **name of the ELF file** and **string arguments**
- `wait()` – wait for a process to finish
- `exit()` – program is complete (waiter can return)
- `signal()` – send a notification to another process (eg, to kill a runaway process)

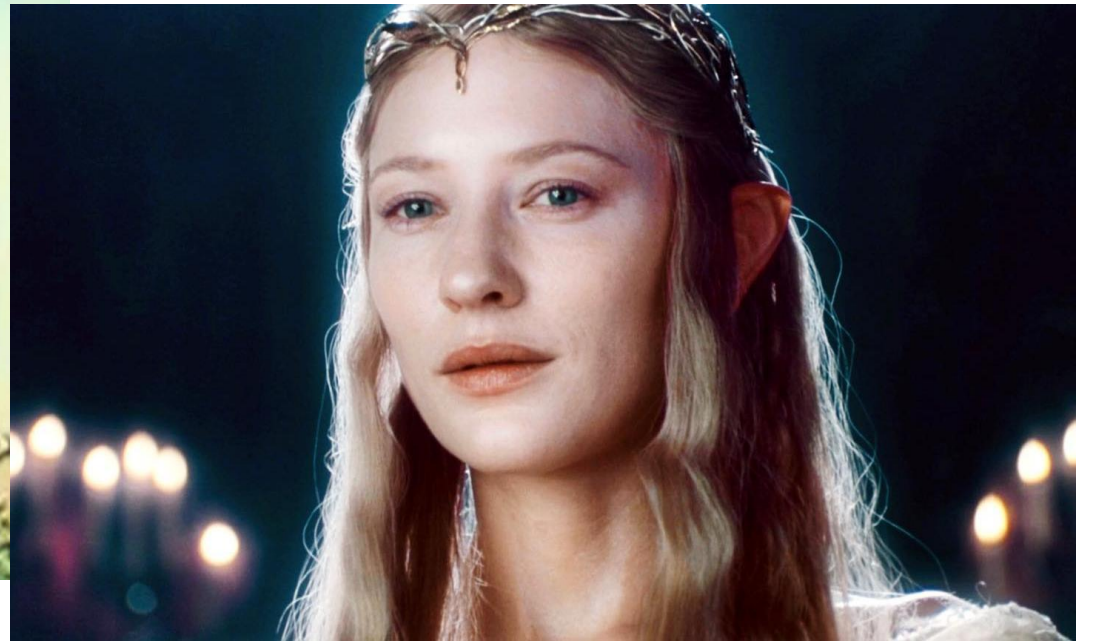
Windows CreateProcess API

```
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]  
);
```

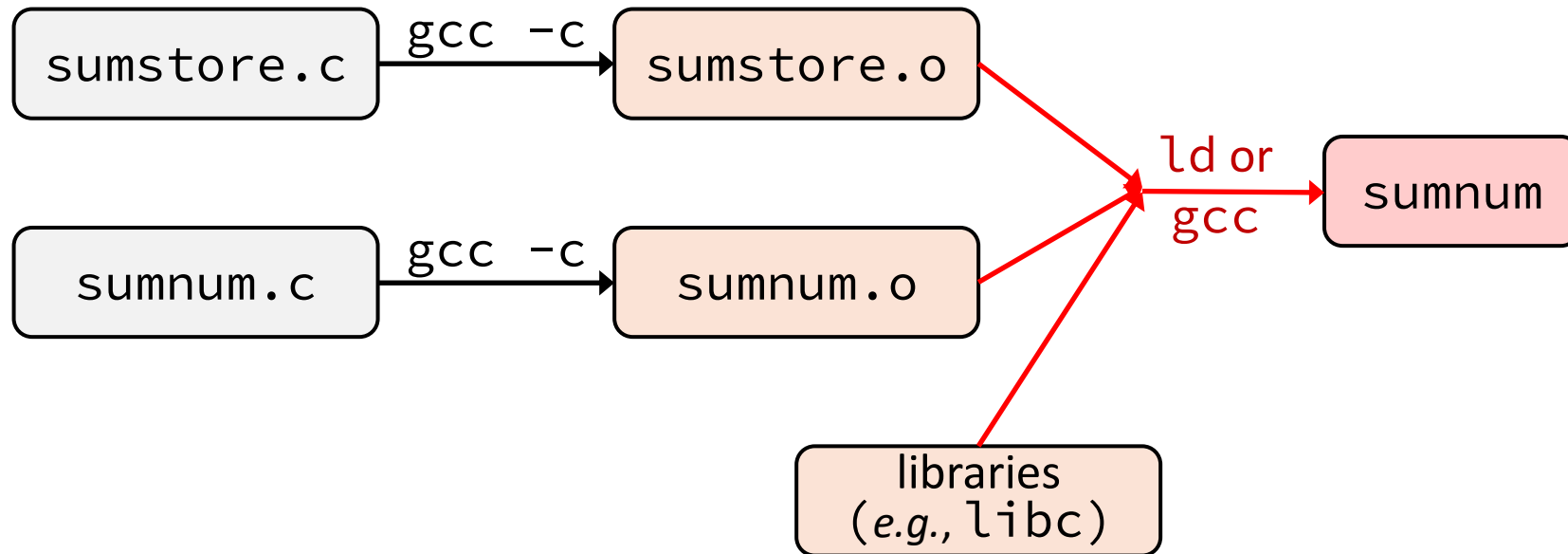
UNIX Process Management



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 `main()`'s stack frame by hand
 - Set up a trap frame to start executing at `main()`
 - ...technically, trap frame will point to "preface" in `user/lib/preface.S`
 - When `main` returns, `start.S` calls `process exit`
 - Inform the scheduler that the new process is ready to run



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());
    return 0;
} else {
    // I'm the parent process
    printf("I am parent of #d\n", child_pid);
    return 0;
}
```



Questions

- Can UNIX `fork()` return an error? Why?
 - `:(){ :|:& };;`
- Can UNIX `exec()` return an error? Why?
- Can UNIX `wait()` ever return immediately? Why?

Is UNIX fork too slow?

- 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”