SE 350 Operating Systems

Lecture 2: OS Concepts

Prof. Seyed Majid Zahedi

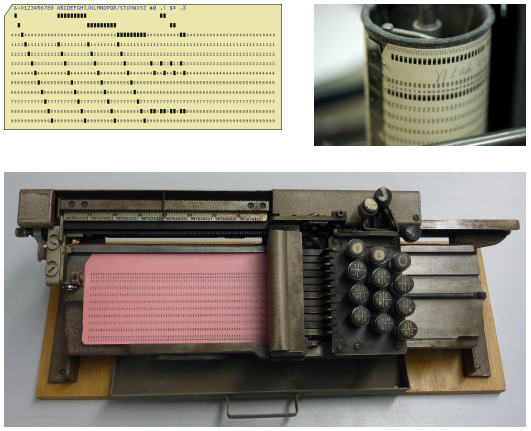
https://ece.uwaterloo.ca/~smzahedi

Outline

- Brief history of OSes
- Four fundamental OS concepts
  - Thread
  - Address space
  - Process
  - Dual-mode operation/protection

Serial Processing

- Machines did not have operating systems
- Run from console with display lights, toggle switches, input device, and printer
- Machine is used by a single user (users had to reserve time to use machines)
- Running programs had long lead time (users had to load compiler and source program, save compiled program, and then load and link it)
- Debugging programs was extremely hard



wikimedia.org

columbia.edu

Evolution of OSes

- Simple batch OS
  - Jobs with same requirement and grouped into batches
  - Special program, called monitor, monitors and manages each program
  - Erroneous or misbehaving jobs could corrupt entire system
  - Automatic job sequencing improves throughput, but I/O is still slow
- Multiprogramming batch OS
  - When running job requires I/O, OS switches to another job
  - While this maximizes CPU utilization, response time could still suffer
- Time-sharing OS
  - Multiple users simultaneously access system through terminals
  - Processor's time is shared among multiple users
  - Primary focus is to minimize response time

Very Brief History of OS

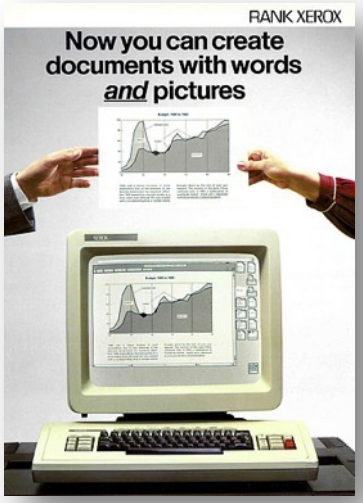
- Several distinct phases:
  - Hardware expensive, humans cheap
    - Eniac, … Multics

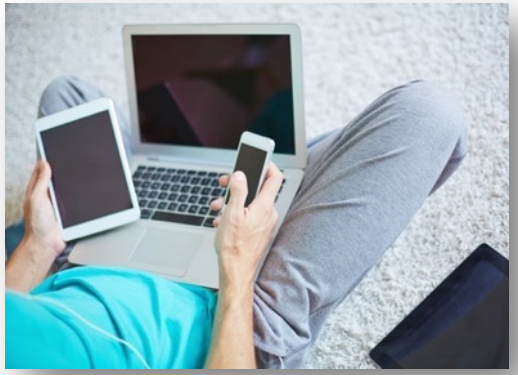
"I think there is a world market for maybe five computers." – *Thomas Watson, chairman of IBM, 1943*

Thomas Watson was often called "the worlds greatest salesman" by the time of his death in 1956

Very Brief History of OS (cont.)

- Several distinct phases:
  - Hardware expensive, humans cheap
    - Eniac, … Multics
  - Hardware cheaper, humans expensive
    - PCs, workstations, rise of GUIs
  - Hardware very cheap, humans very expensive
    - Ubiquitous devices, widespread networking



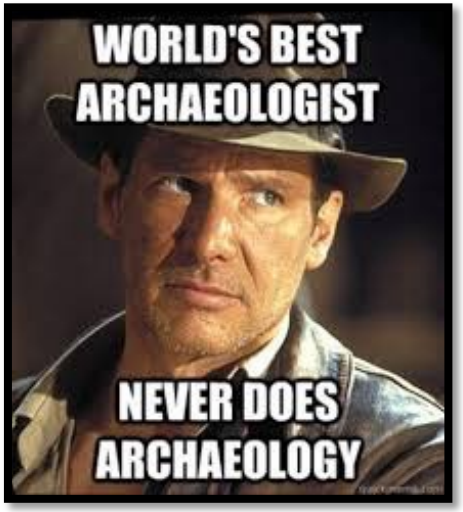


Very Brief History of OS (cont.)

- Several distinct phases:
  - Hardware expensive, humans cheap
    - Eniac, … Multics
  - Hardware cheaper, humans expensive
    - PCs, workstations, rise of GUIs
  - Hardware very cheap, humans very expensive
    - Ubiquitous devices, widespread networking
- Rapid change in hardware leads to changing OS
  - Batch ⇒ multiprogramming ⇒ timesharing ⇒ GUI ⇒ ubiquitous devices
  - Gradual migration of features into smaller machines
- Today
  - Small OS: 100K lines / Large: 20M lines (10M browser!)
  - 100-1000 people-years

OS Archaeology

- Due to high cost of building OS from scratch, most modern OS's have long lineage
- Multics ⇒ AT&T Unix ⇒ BSD Unix ⇒ Ultrix, SunOS, NetBSD,…
- Mach (micro-kernel) + BSD ⇒ NextStep ⇒ XNU ⇒ Apple OS X, iPhone iOS



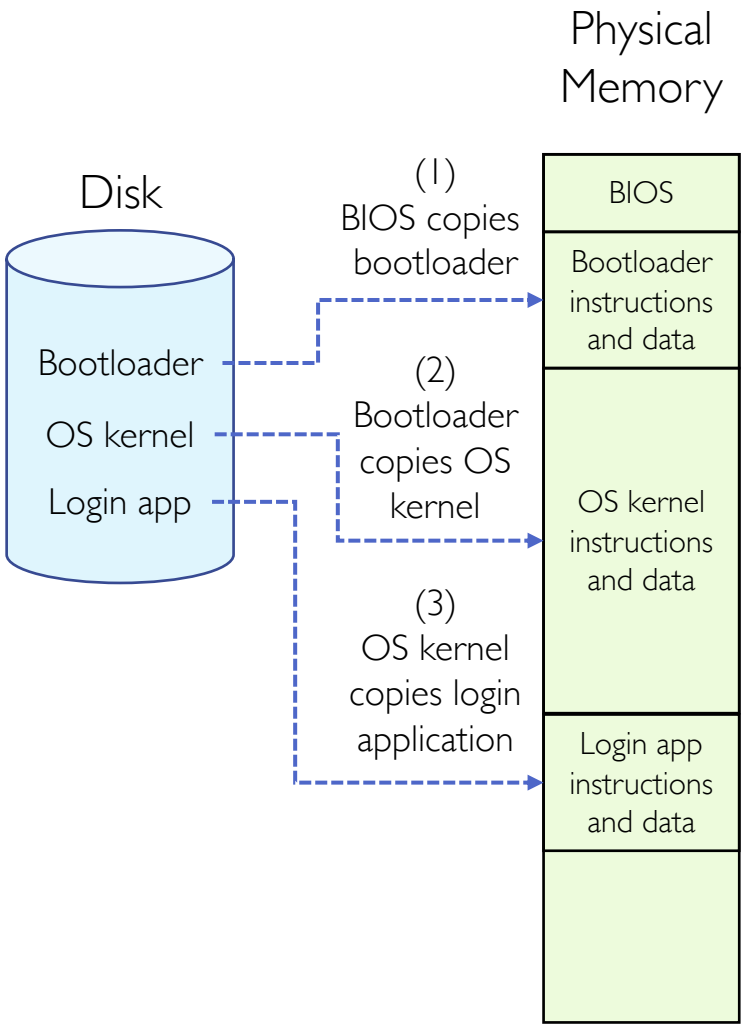
anthropology4u.medium.com

- MINIX ⇒ Linux ⇒ Android, Chrome OS, RedHat, Ubuntu, Fedora, Debian, Suse,…
- CP/M ⇒ QDOS ⇒ MS-DOS ⇒ Windows 3.1 ⇒ NT ⇒ 95 ⇒ 98 ⇒ 2000 ⇒ XP ⇒ Vista ⇒ 7 ⇒ 8 ⇒ 10 ⇒ …

Today: Four Fundamental OS Concepts

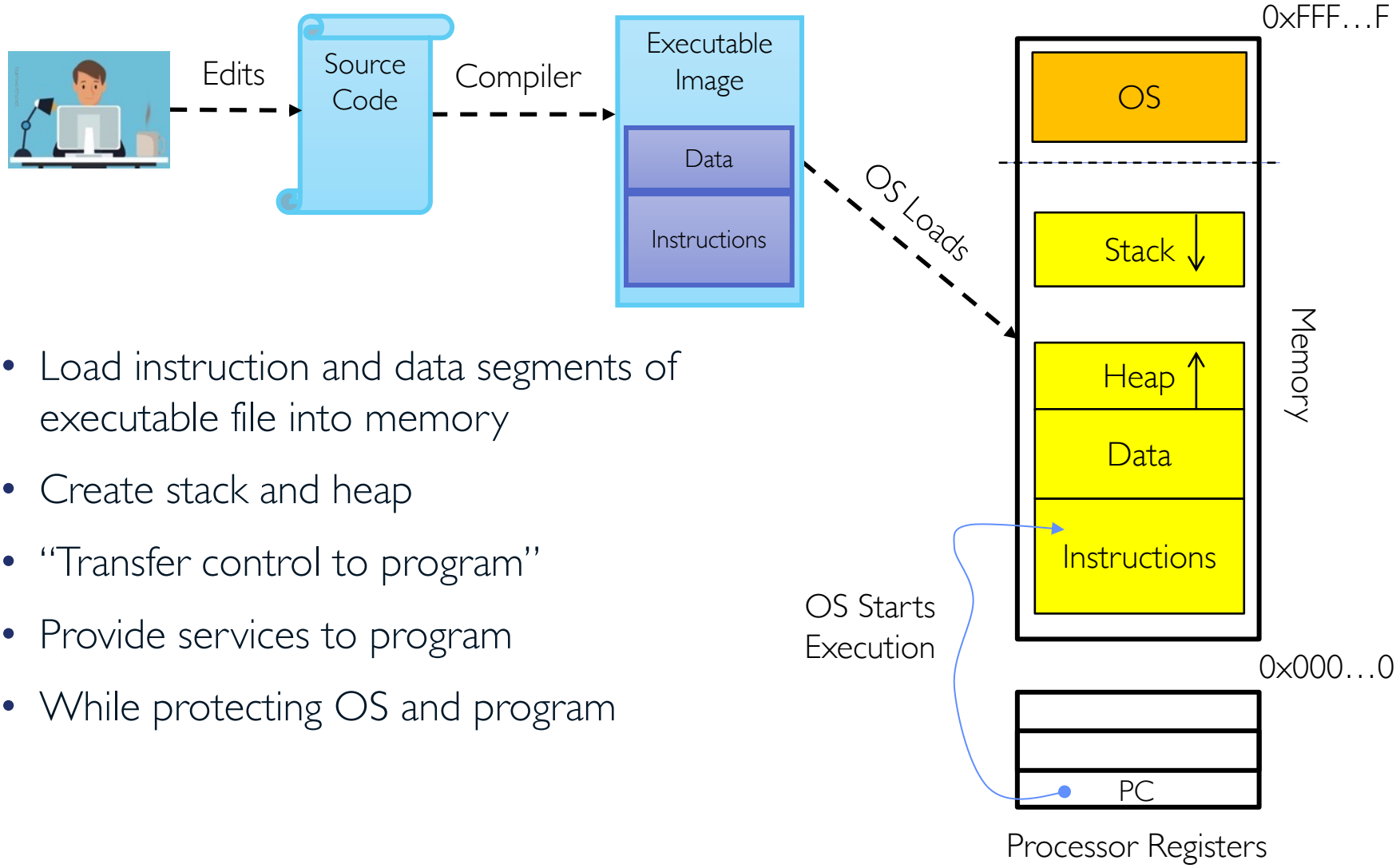
- Thread
  - Single unique execution context which fully describes program state
  - Program counter, registers, execution flags, stack
- Address space (with translation)
  - Address space which is distinct from machine's physical memory addresses
- Process
  - Instance of executing program consisting of address space and 1+ threads
- Dual-mode operation/protection
  - Only "system" can access certain resources
  - OS and hardware are protected from user programs
  - User programs are isolated from one another by controlling translation from program virtual addresses to machine physical addresses

Booting OS



- In most x86 systems, BIOS is stored on Boot ROM
  - Expensive and writing to it is slow
- Why not storing kernel on Boot ROM?
  - Hard to update (OS updates are frequent)
- Why does BIOS load bootloader not OS?
  - Might have multiple OSes installed
  - BIOS needs to read raw bytes from disk, whereas bootloader needs to know how to read from filesystem

OS Bottom Line: Run Programs

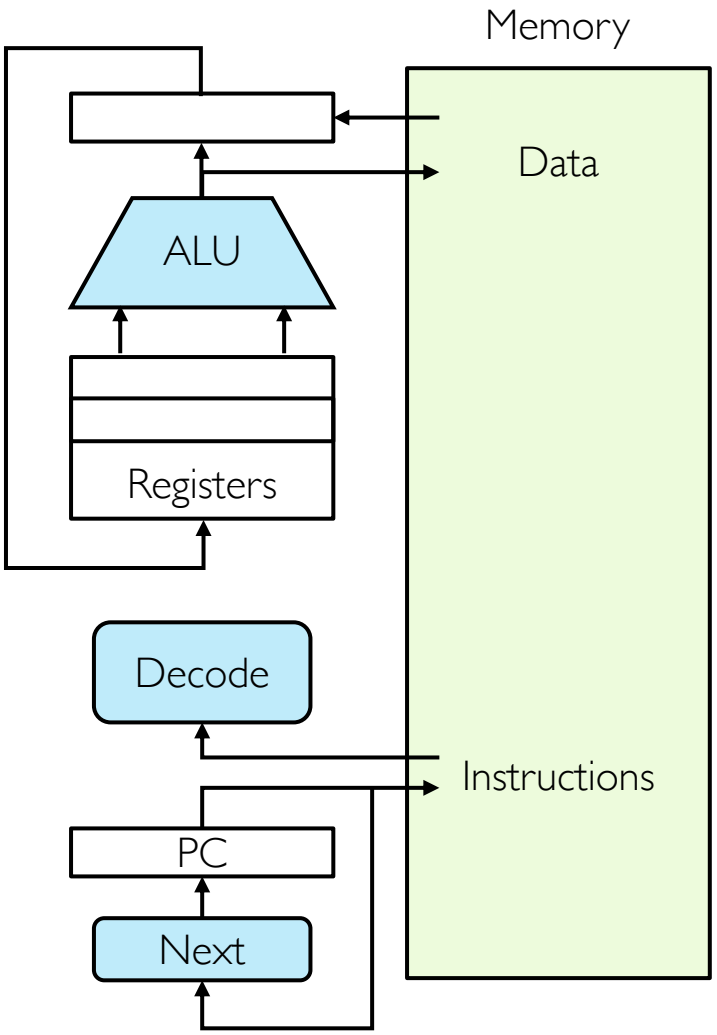


- Load instruction and data segments of executable file into memory
- Create stack and heap
- "Transfer control to program"
- Provide services to program
- While protecting OS and program

Instruction Cycle: Fetch, Decode, Execute

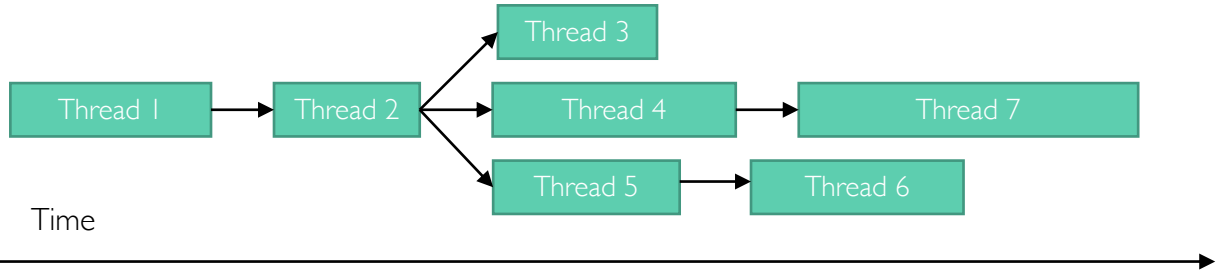
- Execution sequence
  - Fetch instruction at PC
  - Decode
  - Execute (possibly using registers)
  - Write results to registers/memory
  - PC ← *Next*(PC)
  - Repeat

Next instruction or jump to new address …



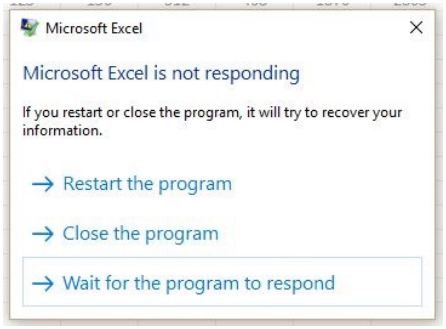
Thread (1st OS Concept)

- Thread is short for thread of execution
- Thread of execution is sequence of executable commands that can run on CPU
- Threads have some state and store some local variables
  - Execution state (ready, running, waiting, …)
  - Saved context when not running
  - Execution stack
  - Local variables
  - …
- Multithreaded programs use more than one thread (some of the time)
  - Program begins with single initial thread (where the **`main`** method is)
  - Threads can be created and destroyed within programs dynamically



Example: UI Thread

- One common way of dividing up program into threads is to separate user interface from other time-consuming actions
- If user interface and upload method share the same thread, then once file upload has started, user will not be able to use UI anymore
  - Not even to click the button that cancels the upload!



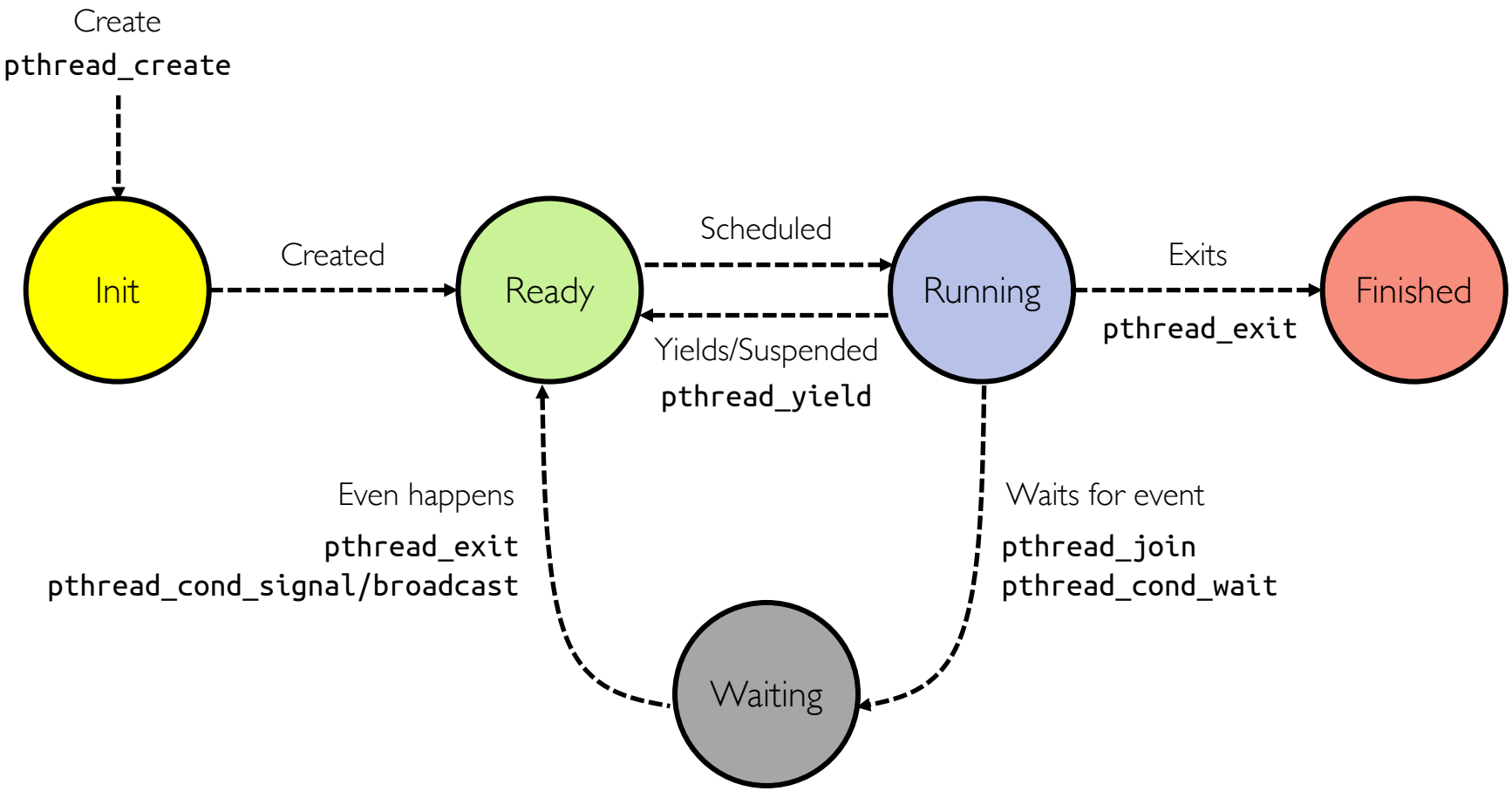
- UI thread can spawn new thread to handle the upload when user clicks "upload"
- UI thread remain responsive as it is not waiting for the upload method to complete

The POSIX Thread

- `pthread` refers to POSIX standard that defines thread behavior in UNIX

- `pthread_create`
  - Creates new thread to run a function
- `pthread_exit`
  - Quit thread and clean up, wake up joiner if any
  - To allow other threads to continue execution, the main thread should terminate by calling pthread_exit() rather than exit(3)
- `pthread_join`
  - In parent, wait for children to exit, then return
- `pthread_yield`
  - Relinquish CPU voluntarily
- …

Thread Lifecycle



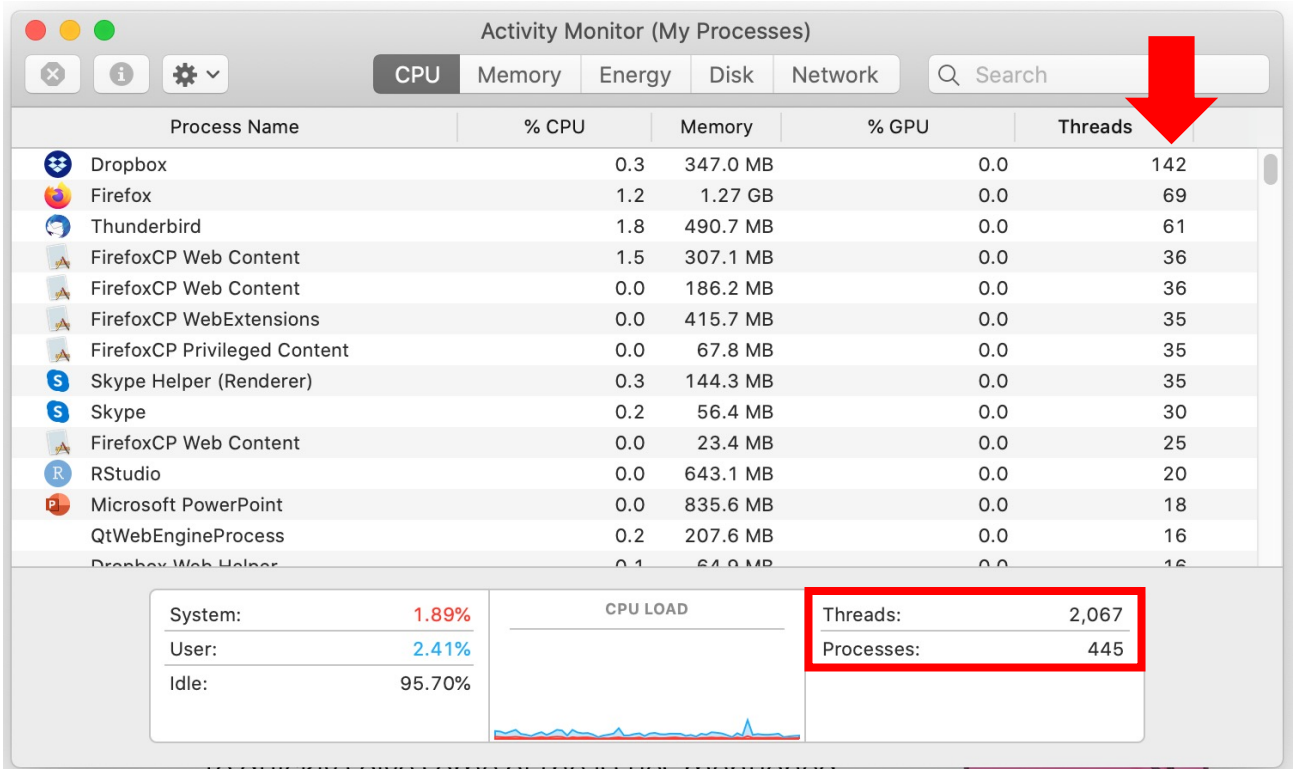
A process can go directly from ready or waiting to finished (example: main thread calls exit)

Thread Control Block (TCB)

- Data structure in OS containing information needed to manage a thread
  - Thread unique identifier (tid)
  - Stack pointer (points to thread's stack in the process)
  - Program counter (points to the current program instruction of the thread)
  - State of the thread (e.g., running, ready, waiting, etc.)
  - Thread's register values
  - Pointer to process control block (PCB) of the process that the thread lives on (more on this soon)

Some Numbers

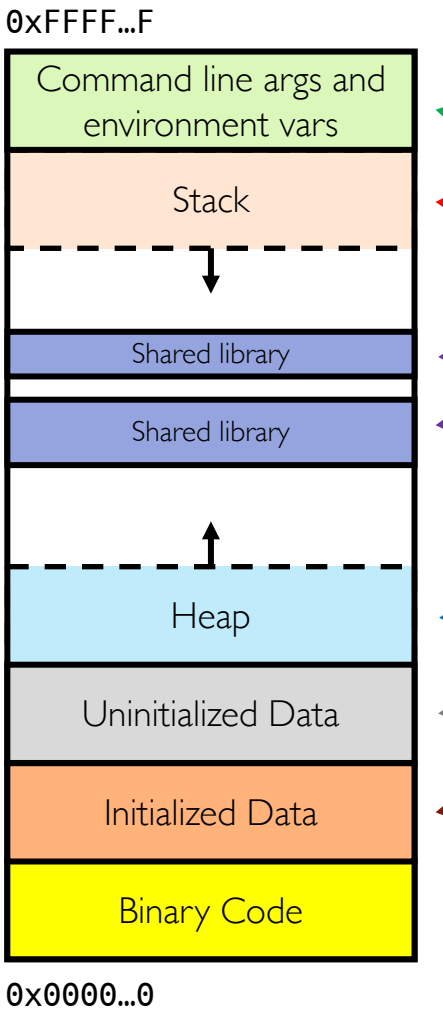
- Many process are multi-threaded, so thread context switches may be either within-process or across-processes



Address Space (2nd OS Concept)

- Address space: set of accessible addresses and their state
- Physical memory: data storage medium
- Physical addresses: addresses available on physical memory
  - For 4GB of memory: 2^{32}B ~ 4 billion addresses
- Virtual addresses: addresses generated by program
  - For 64-bit processor: 2^{64} > 18 quintillion (10^{18}) addresses

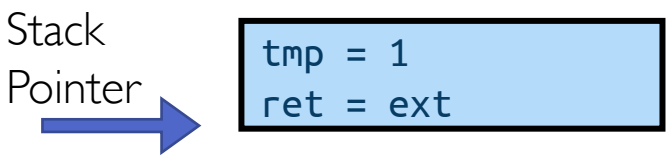
Virtual Address Space Layout of C Programs




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

int x;
int y = 15;

int main(int argc, char *argv[]) {

    int *values;
    int I;

    values = (int *)malloc(sizeof(int)*5);

    for (i = 0; i < 5; i++)
        values[i] = i;

    return 0;
}
```

Stack Example

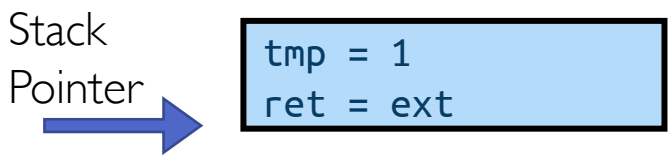
```
A0:  A(int tmp) {
A1:       if (tmp<2)
A2:            B();
A3:       printf(tmp);
A4:  }
B0:  B() {
B1:       C();
B2:  }
C0:  C() {
C1:       A(2);
C2:  }
     A(1);
ext:
```



- Stack holds temporary results
- Permits recursive execution
- Crucial to modern languages

Stack Example

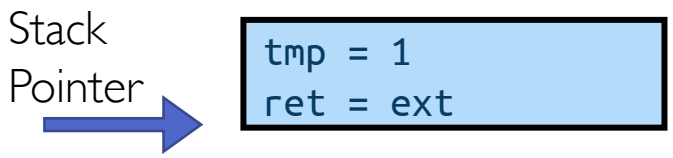
```
A0:  A(int tmp) {
A1:       if (tmp<2)
A2:            B();
A3:       printf(tmp);
A4:  }
B0:  B() {
B1:       C();
B2:  }
C0:  C() {
C1:       A(2);
C2:  }
     A(1);
ext:
```



- Stack holds temporary results
- Permits recursive execution
- Crucial to modern languages

Stack Example

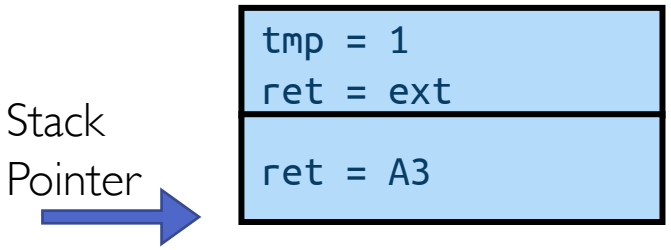
```
A0:  A(int tmp) {
A1:       if (tmp<2)
A2:            B();
A3:       printf(tmp);
A4:  }
B0:  B() {
B1:       C();
B2:  }
C0:  C() {
C1:       A(2);
C2:  }
     A(1);
ext:
```



- Stack holds temporary results
- Permits recursive execution
- Crucial to modern languages

Stack Example

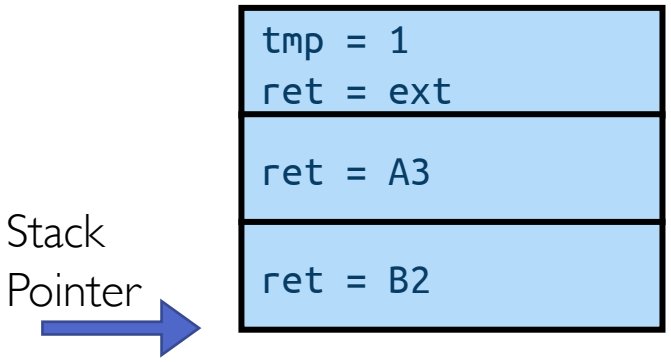
```
A0:  A(int tmp) {
A1:       if (tmp<2)
A2:            B();
A3:       printf(tmp);
A4:  }
B0:  B() {
B1:       C();
B2:  }
C0:  C() {
C1:       A(2);
C2:  }
     A(1);
ext:
```



- Stack holds temporary results
- Permits recursive execution
- Crucial to modern languages

Stack Example

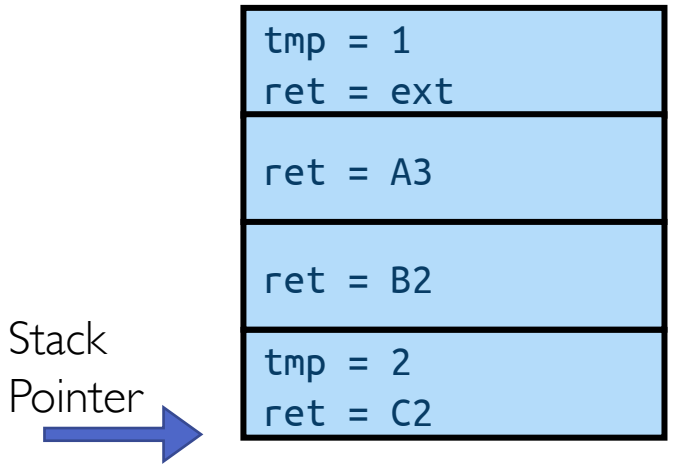
```
A0:  A(int tmp) {
A1:       if (tmp<2)
A2:            B();
A3:       printf(tmp);
A4:  }
B0:  B() {
B1:       C();
B2:  }
C0:  C() {
C1:       A(2);
C2:  }
     A(1);
ext:
```



- Stack holds temporary results
- Permits recursive execution
- Crucial to modern languages

Stack Example

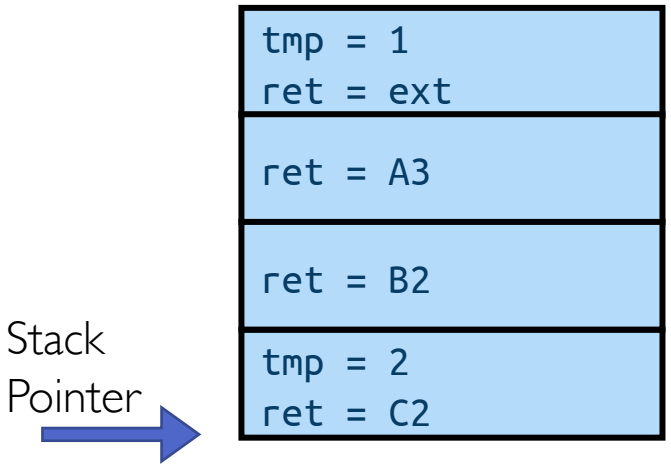
```
A0:  A(int tmp) {
A1:       if (tmp<2)
A2:            B();
A3:       printf(tmp);
A4:  }
B0:  B() {
B1:       C();
B2:  }
C0:  C() {
C1:       A(2);
C2:  }
     A(1);
ext:
```



- Stack holds temporary results
- Permits recursive execution
- Crucial to modern languages

Stack Example

```
A0:  A(int tmp) {
A1:       if (tmp<2)
A2:            B();
A3:       printf(tmp);
A4:  }
B0:  B() {
B1:       C();
B2:  }
C0:  C() {
C1:       A(2);
C2:  }
     A(1);
ext:
```

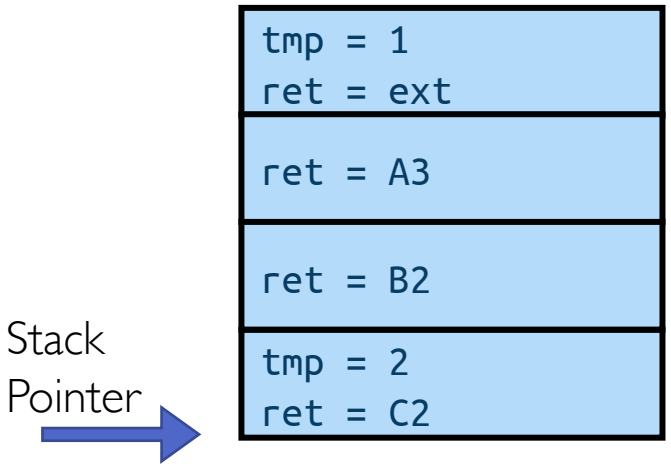



```
> 2
```

- Stack holds temporary results
- Permits recursive execution
- Crucial to modern languages

Stack Example

```
A0:  A(int tmp) {
A1:       if (tmp<2)
A2:            B();
A3:       printf(tmp);
A4:  }
B0:  B() {
B1:       C();
B2:  }
C0:  C() {
C1:       A(2);
C2:  }
     A(1);
ext:
```

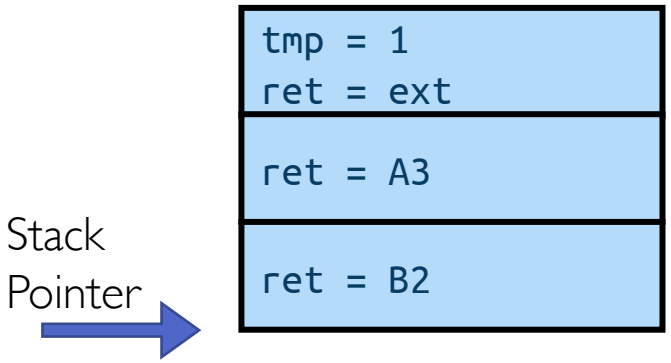



```
> 2
```

- Stack holds temporary results
- Permits recursive execution
- Crucial to modern languages

Stack Example

```
A0:  A(int tmp) {
A1:       if (tmp<2)
A2:            B();
A3:       printf(tmp);
A4:  }
B0:  B() {
B1:       C();
B2:  }
C0:  C() {
C1:       A(2);
C2:  }
     A(1);
ext:
```

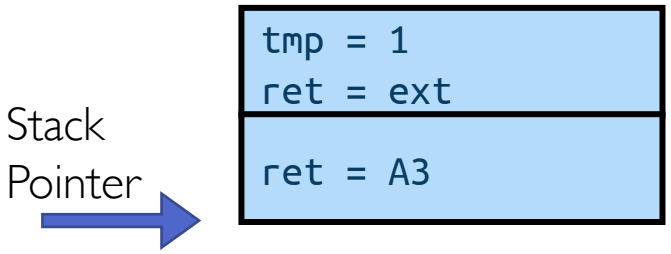



```
> 2
```

- Stack holds temporary results
- Permits recursive execution
- Crucial to modern languages

Stack Example

```
A0:  A(int tmp) {
A1:       if (tmp<2)
A2:            B();
A3:       printf(tmp);
A4:  }
B0:  B() {
B1:       C();
B2:  }
C0:  C() {
C1:       A(2);
C2:  }
     A(1);
ext:
```

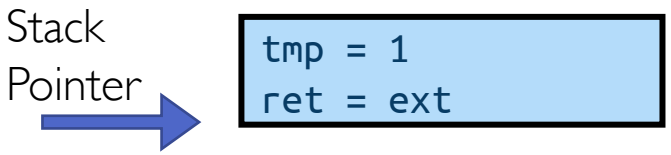



```
> 2
```

- Stack holds temporary results
- Permits recursive execution
- Crucial to modern languages

Stack Example

```
A0:  A(int tmp) {
A1:       if (tmp<2)
A2:            B();
A3:       printf(tmp);
A4:  }
B0:  B() {
B1:       C();
B2:  }
C0:  C() {
C1:       A(2);
C2:  }
     A(1);
ext:
```

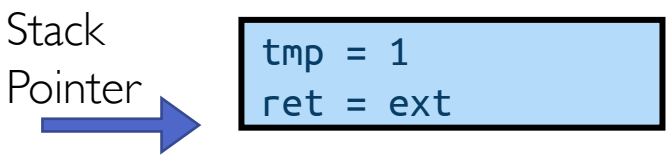



```
> 21
```

- Stack holds temporary results
- Permits recursive execution
- Crucial to modern languages

Stack Example

```
A0:  A(int tmp) {
A1:       if (tmp<2)
A2:            B();
A3:       printf(tmp);
A4:  }
B0:  B() {
B1:       C();
B2:  }
C0:  C() {
C1:       A(2);
C2:  }
     A(1);
ext:
```




```
> 21
```

- Stack holds temporary results
- Permits recursive execution
- Crucial to modern languages

Stack Example

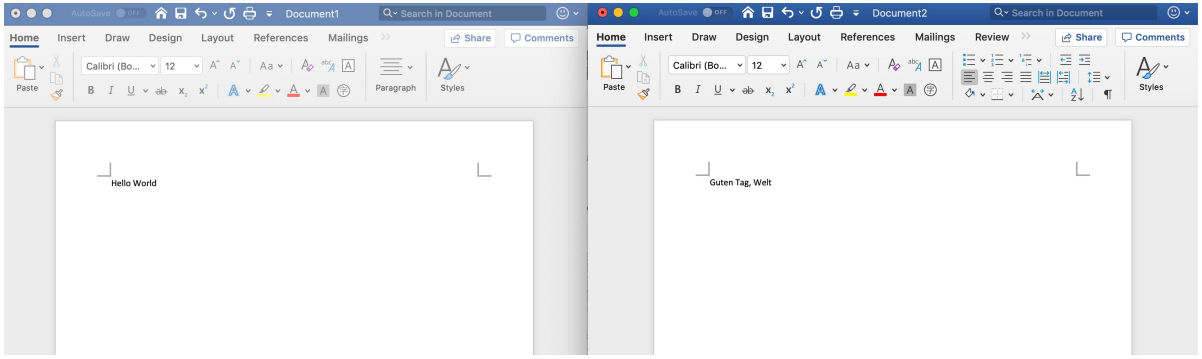
```
A0:  A(int tmp) {
A1:       if (tmp<2)
A2:            B();
A3:       printf(tmp);
A4:  }
B0:  B() {
B1:       C();
B2:  }
C0:  C() {
C1:       A(2);
C2:  }
     A(1);
ext:
```

```
> 21
```

- Stack holds temporary results
- Permits recursive execution
- Crucial to modern languages

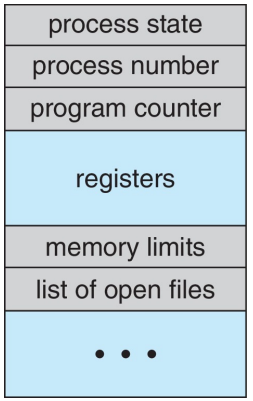
Process (3rd OS Concept)

- A process is a program in execution
- Two instances of same program running equals two processes
  - You may have two windows open for Microsoft Word, and even though they are the same program, they are separate processes
  - Similarly, two users who both use Firefox at the same time on a terminal server are interacting with two different processes



Process Control Block (PCB)

- Is a data structure for managing processes
- Is created and updated by OS for each running process
  - Is kept up to date constantly as process executes
- Is held in memory and maintained in some container (e.g., list) by kernel
- Contains everything OS needs to know about the process
  - Unique process identifier (PID), state, priority
  - Program counter (PC)
  - Register data
  - Memory pointers
  - I/O status information,
  - Accounting information

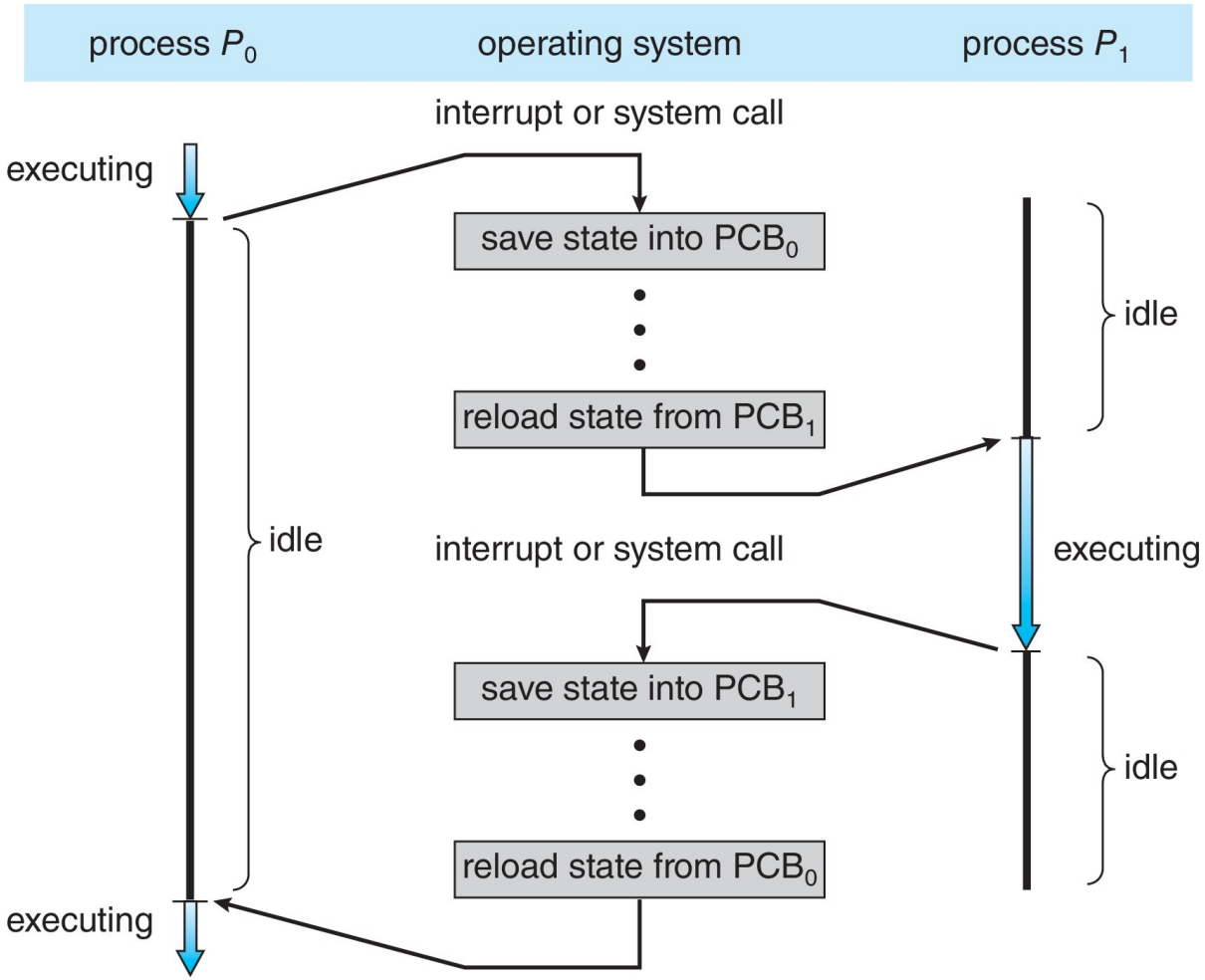


- PC and register data do not need to be updated when program is running
  - They are needed when a system call (trap) or process switch occurs

PCB During Process Life Cycle

- Upon creation, OS creates new PCB for the process
- OS initializes data in new PCB
  - Set variables to their initial values
  - Set initial program state
  - Set instruction pointer to first instruction in main
- OS then adds PCB to the list of PCBs
- After process is terminated and cleaned up, OS removes the PCB from its list of active processes
  - OS might collect some data before removing PCB (e.g., summary of accounting information)

Context Switch: CPU Switch Between Two Processes



Process Creation

- System boot up
  - E.g., login process in Linux
  - Embedded systems often create all processes they will ever run at bootup
- User request
  - E.g., double clicking on icons

- One process spawns another
  - E.g., clicking on a link in an email makes email process start a web browser
  - E.g., entering a command, like **`ls`** or **`top`**, makes shell process start a new process
  - Programs may beak their work up into different logical parts
    - To promote parallelism or fault tolerance
  - Processes, unlike most plants and animals, reproduces asexually
  - Spawning process is the parent and the one spawned is the child
    - Each process has one parent and zero or more children
    - Each process and all its descendants form process group

`fork()`: Spawning New Process in Unix

- `fork()` creates new process as copy of itself with new PID
- Both parent and child continue after `fork()`
- Call to `fork` can return a value
  - Positive value means this is the parent
    - Value is PID of the child
  - Zero value means this is the child
  - Negative value means the `fork` failed
    - Error! Must be handled somehow
    - Running in original process

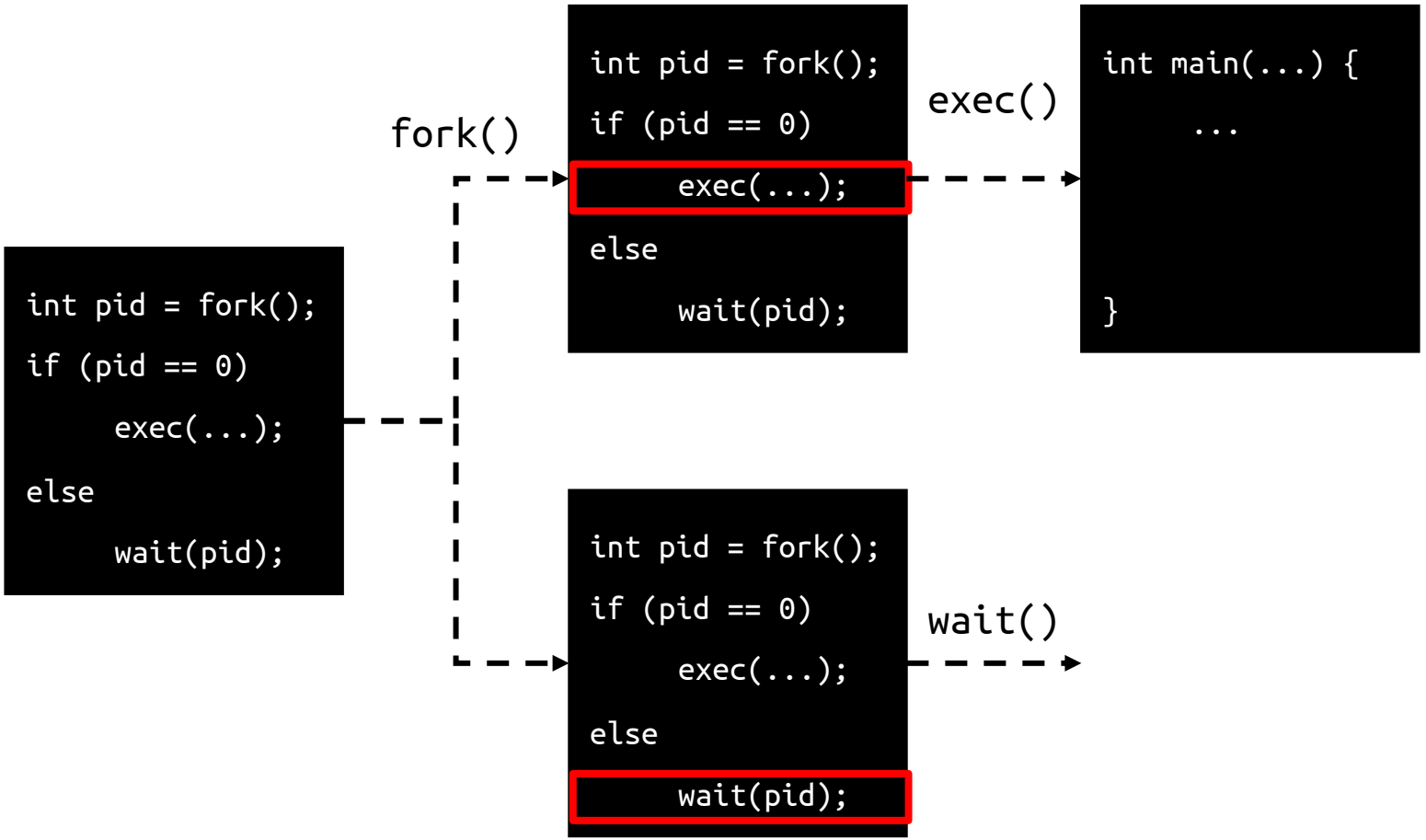
www.fortressofsolitude.co.za

- All state of original process duplicated in both parent and child
  - Memory, file descriptors, etc.

UNIX Process Management

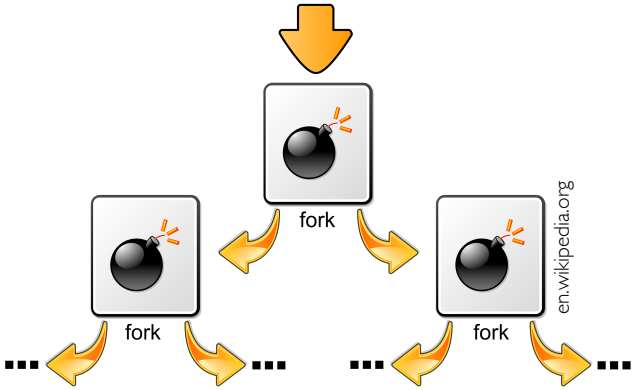
- `fork()`
  - Syscall to create copy of current process and start it
- `exec()`
  - Syscall to change program being run by current process
- `wait()`
  - Syscall to wait for process to finish
- `signal()`
  - Syscall to send notification to another process (e.g., `SIGKILL, SIGINT`)

fork() Example



Aside: Fork Bomb

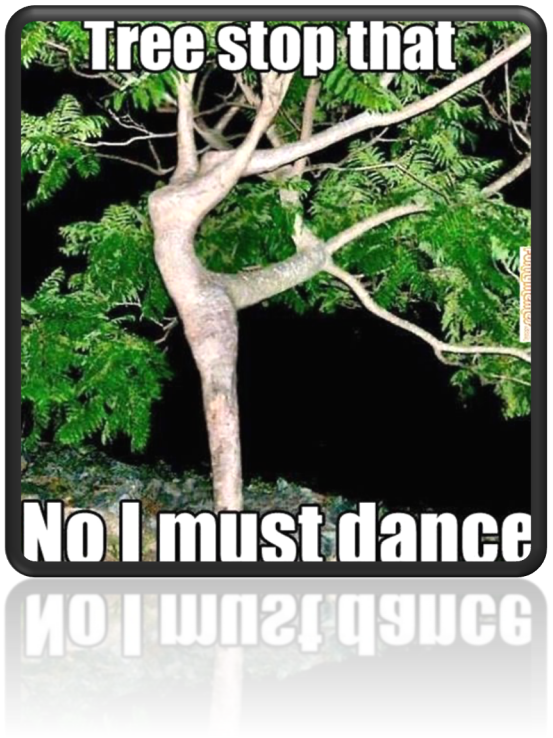
- The idea is to call `fork` repeatedly
- Keep doing this until the system crashes (or no work can get done)



- Exponential growth (2^n) processes after n calls
- OS can defend against this
  - Limit total number of processes per user
  - Limit rate of process spawning
- Note: do not attempt this on University computers

Process Family Tree in Unix

- First process created is called **`init`**
  - Assigned PID of 1
  - Grandparent of all processes
  - Like **`object`** class in Java which is superclass of all classes



- **`init`** is replaced by **`systemd`** in some newer distributions

- Parent of **`init`** is **`swapper`** (or **`sched`**)
  - Part of kernel and responsible for paging (will come back to this later in the course)

- If parent dies before its child, the child becomes orphan
  - Automatically adopted by **`init`** process

Example: pstree Output in FreeBSD

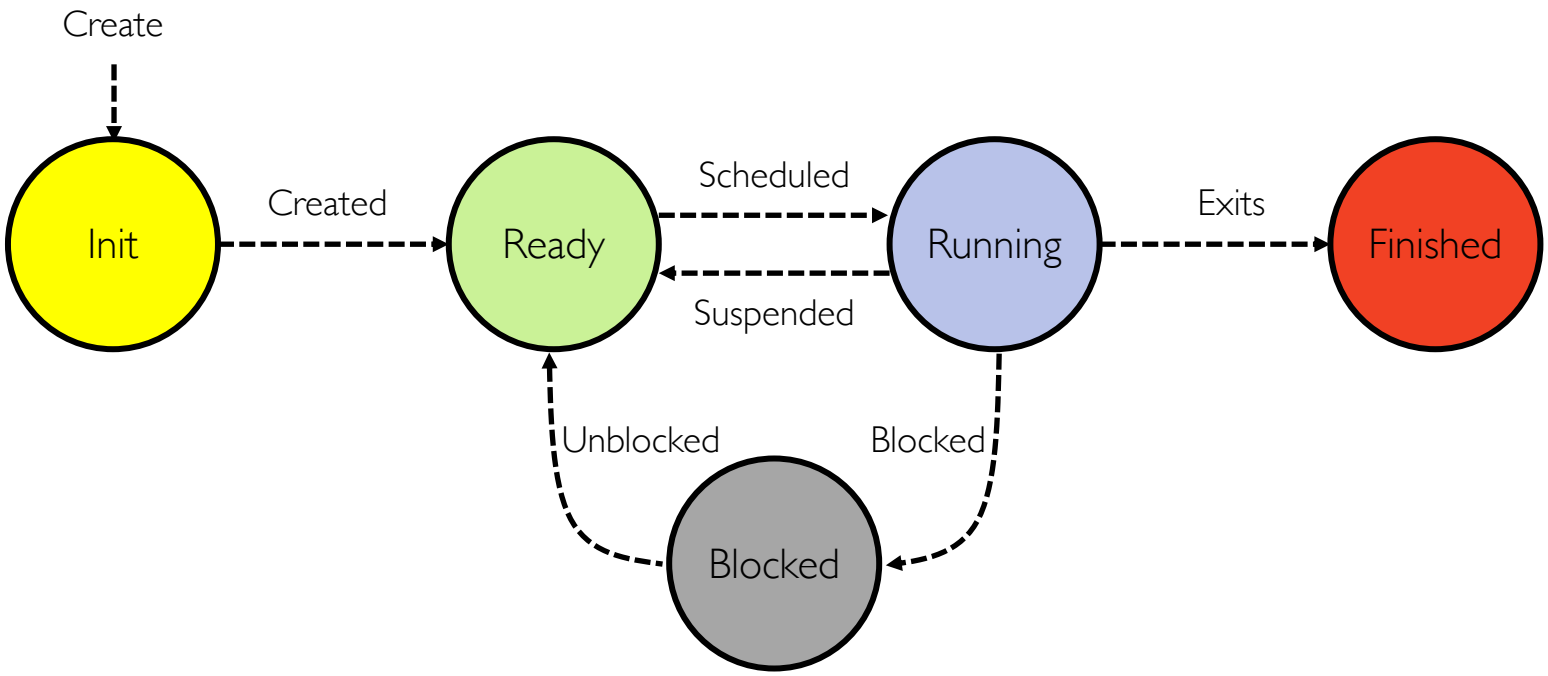
```
-+= 00000 root [swapper]
 ¦-+= 00001 root /sbin/init --
 ¦ ¦--= 00196 root /sbin/devd
 ¦ ¦--= 00230 root /usr/sbin/syslogd -s
 ¦ ¦--= 00344 root sendmail: accepting connections (sendmail)
 ¦ ¦--= 00348 smmsp sendmail: Queue runner@00:30:00 for /var/spool/clientmqueue
 ¦ ¦--= 00354 root /usr/sbin/cron -s
 ¦ ¦--= 00439 _dhcp dhclient: ed0 (dhclient)
 ¦ ¦-+= 00391 root login [pam] (login)
 ¦ ¦ \-+= 00400 root -csh (csh)
 ¦ ¦   ¦-+= 00701 root /usr/local/bin/pstree
 ¦ ¦   ¦ \-+- 00703 root sh -c ps -axwwo user,pid,ppid,pgid,command
 ¦ ¦   ¦   \--- 00704 root ps -axwwo user,pid,ppid,pgid,command
 ¦ ¦   \--- 00702 root less
 ¦ ¦--- 00406 root dhclient: ed0 [priv] (dhclient)
 ¦ ¦--= 00392 root /usr/libexec/getty Pc ttyv1
 ¦ ¦--= 00393 root /usr/libexec/getty Pc ttyv2
 ¦ ¦--= 00394 root /usr/libexec/getty Pc ttyv3
 ¦ ¦--= 00395 root /usr/libexec/getty Pc ttyv4
 ¦ ¦--= 00396 root /usr/libexec/getty Pc ttyv5
 ¦ ¦--= 00397 root /usr/libexec/getty Pc ttyv6
 ¦ ¦--= 00398 root /usr/libexec/getty Pc ttyv7
 ¦ ¦-+- 00387 root sh /etc/rc autoboot
 ¦ ¦ \-+- 00390 root sh /etc/rc autoboot
:
```

en.wikipedia.org

Process Destruction

- Normal exit (voluntary)
  - E.g., when compilation is finished, compiler terminates normally
  - E.g., when you are done editing your document, you click on close button
- Error exit (voluntary)
  - E.g., computer exits with error if you ask it to compile a non-existent file
  - E.g., process required access to temporary directory, but it didn't have permission
- Fatal error (involuntary)
  - E.g., division by zero or segmentation fault
  - OS detects these errors and send it to the program
    - Processes may tell OS that they wish to handle some of these errors by themselves
    - If process can handle the error, it continues
    - Unhandled errors result in involuntary terminations
- Killed by another process (involuntary)
  - Typically, users may only kill processes they have created
    - Exception: system administrator.

Process Lifecycle (5 States)

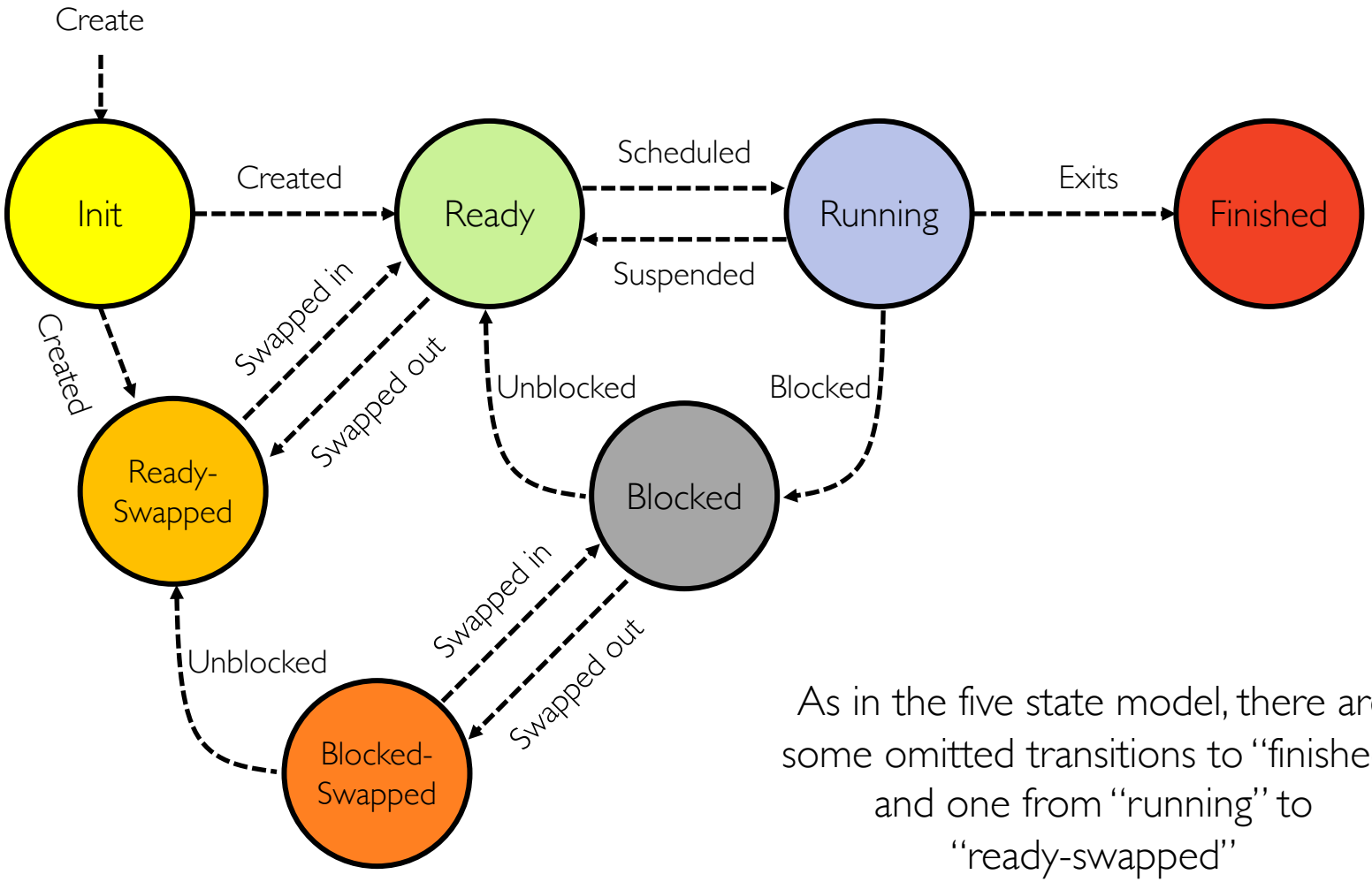


A process can go directly from ready or waiting to finished (example: process is killed!)

Processes and Limited Memory

- Users often want more processes running than fit in memory
- Swapping: when demands for memory exceed available memory, parts of processes will be moved to disk storage to make room
  - This is extremely expensive
- We need to know if a particular process is in memory or on disk
- Is adding a new state (e.g., swapped) be enough?
  - Ideally, we will only swap a process to disk if it is blocked
  - But what if there are no blocked process?
  - Or what if the event a swapped process waited for took place?

Process Lifecycle (7 States)



As in the five state model, there are some omitted transitions to "finished" and one from "running" to "ready-swapped"

Inter-process Communication (IPC)

- Shared memory
  - Normally, each region of memory is associated with one process (its owner)
  - Processes can designate memory as shared
  - OS is involved in setting up (and cleaning up) shared memory regions
- Shared file
  - Processes could read/write to/from files in agreed upon locations
  - OS is still involved in file creation and manipulation
- Message passing
  - Sender gives a message to OS and asks for it to be delivered to recipient
  - OS is obviously involved

Message Passing

- Direct communication
  - Processes must name each other explicitly
  - `send (P, message)` – send a message to process P
  - `receive(Q, message)` – receive a message from process Q

- Indirect communication
  - Messages are directed and received from mailboxes (also called ports)
  - `send(M, message)` – send a message to mailbox M
  - `receive(M, message)` – receive a message from mailbox M

Synchronization

- Message passing may be either blocking or non-blocking
- Blocking is considered synchronous
  - Sender is blocked until the message is received
  - Receiver is blocked until a message is available
- Non-blocking is considered asynchronous
  - Sender sends message and continues
  - Receiver receives a valid message, or Null message

Signals: Limited Form of Direct IPC

- Standardized messages sent to processes to trigger specific behavior
- They don't really contain a message
- The fact that signals contain no message is a limitation that means signals cannot be used for every single IPC scenario

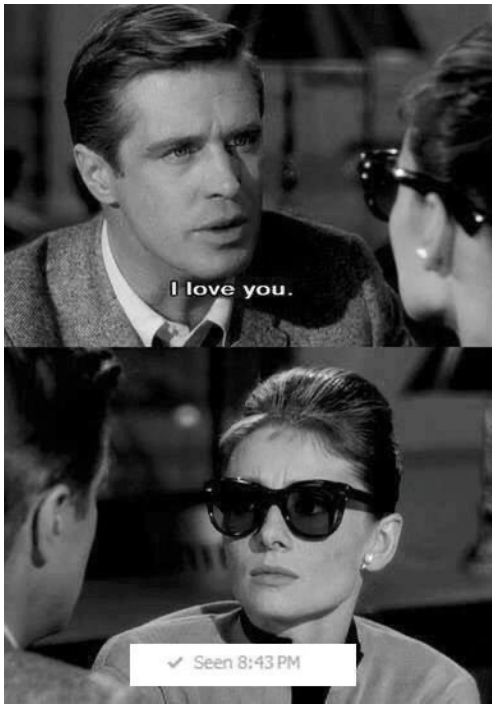
imgflip.com

Signals

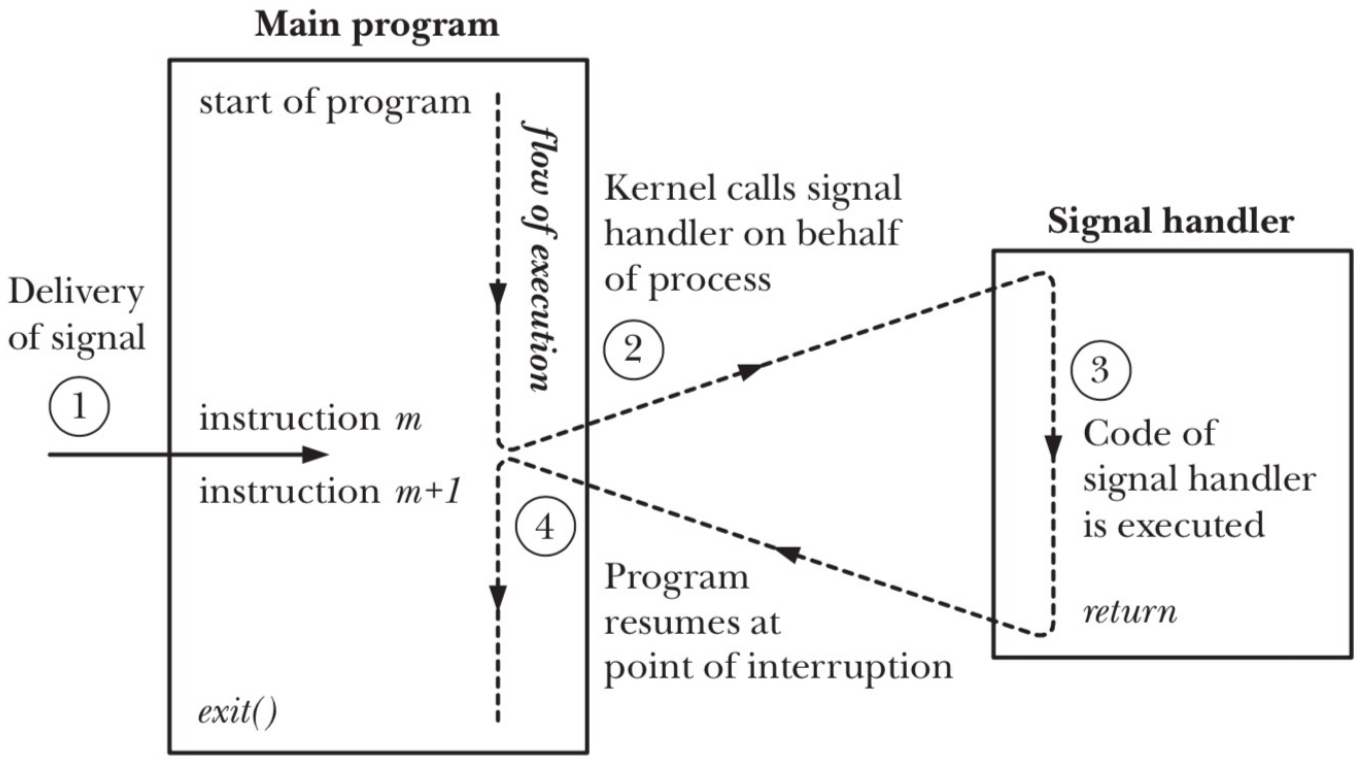
- UNIX systems use signals to indicate events
  - E.g., the Ctrl-C on the console
- It is synchronous if the signal is sent as a result of program execution
  - E.g., dividing by zero or segmentation fault
- It is asynchronous if it comes from outside the process
  - E.g., user pressing Ctrl-C or one process or thread sending a signal to another
- By default, OS handles signals sent to processes with the default handler
- Processes could inform OS they are prepared to handle signal themselves
  - E.g., doing some cleanup when Ctrl-C is received instead of just dying

Signals (cont.)

- Signals can be sent using command line
  - E.g., `kill -9 24601`
  - -9 parameter sends signal 9 (`SIGKILL`)
  - -0 is called the null signal
  - It does not actually send any signal
  - Can be used to check if the recipient process exists
- Process can block signals
  - Exceptions are `SIGKILL` and `SIGSTOP`
  - OS doesn't deliver signal to recipient
- Once signal is delivered, recipient can
  - Ignore it
  - Run the default action
  - Run a signal handler



Signal Handler



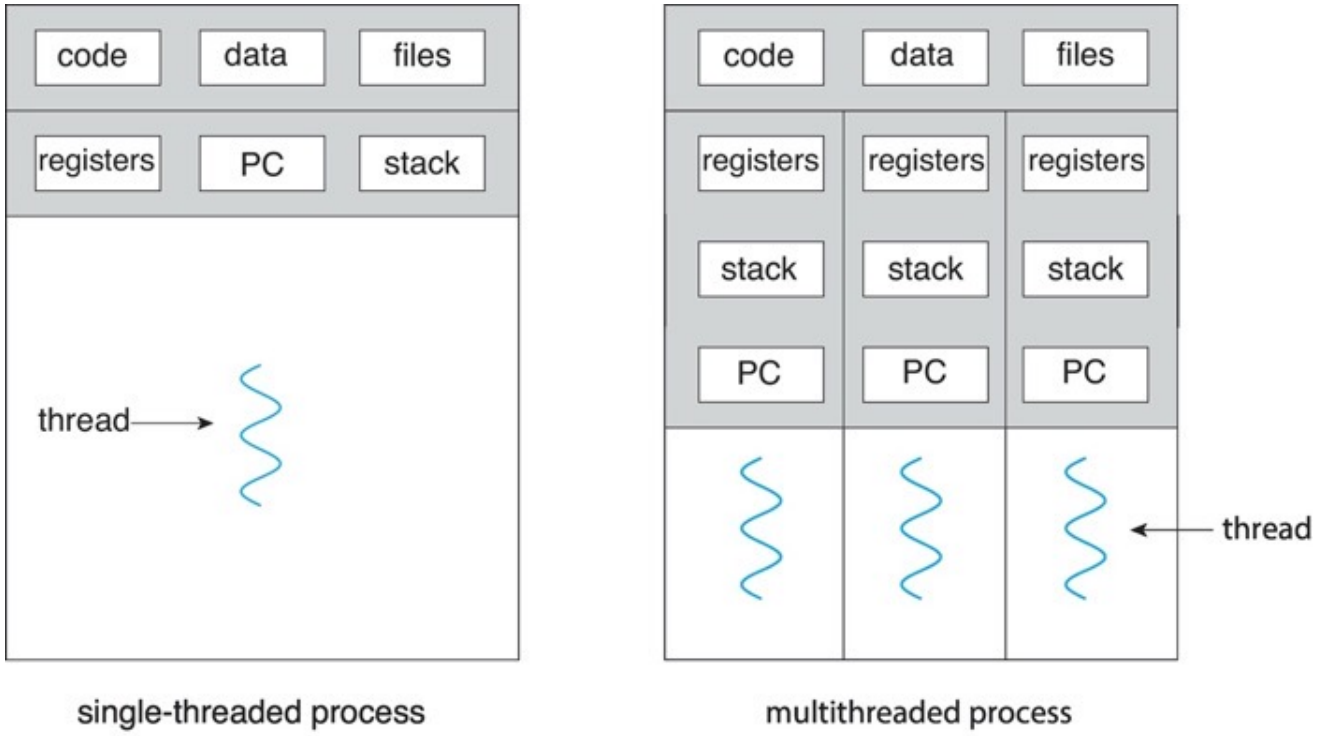
Signal Handler Example

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

volatile int quit = 0;

void handle_it (int signal_num) {
     quit = 1;
}

int main(int argc, char** argv) {
     signal(SIGINT, handle_it);
     while(quite == 0){};

     printf("Time to die.\n");

     return 0;
}
```

Signal Handler: Discussion

- Content of signal handler is restricted
- Because handler deals with interrupts and runs between two instructions, it is important to make sure that signal handler doesn't mess anything up
- If signal handler runs in the middle of `malloc` and signal handler itself calls `malloc` it could put memory management in invalid state
- Signal handler can only use functions that are reentrant
  - There are tables of *safe* functions to be invoked from within a signal handler.

Processes and Threads



single-threaded process

multithreaded process

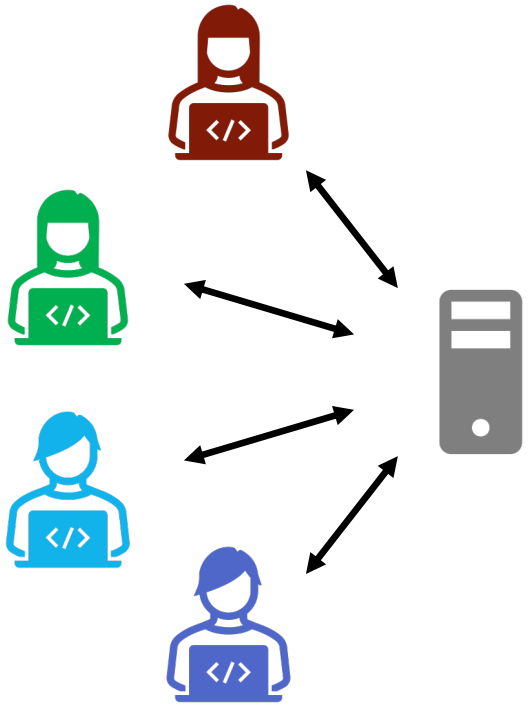
Multithreaded Processes

- Threads encapsulate concurrency and are active components
- Address spaces encapsulate protection and are passive part
  - Keeps buggy program from trashing system
- Why have multiple threads per address space?
  - Processes are expensive to start, switch between, and communicate between

Example: Web Server

- Server must handle many requests
- First option is multiprogramming

```
serverLoop() {
  con = AcceptCon();
  fork(ServiceWebpage, conn);
}
```



- What are some disadvantages of this technique?
  - Expensive to start new process
  - Heavyweight context switch overhead

Example: Web Server (cont.)

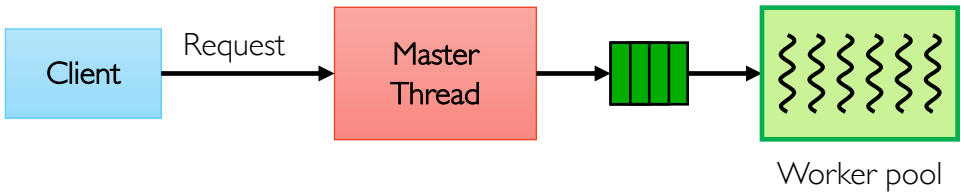
- Second option is multithreading

```
serverLoop() {
    con = AcceptCon();
    thread_create(ServiceWebpage, conn);
}
```

- Looks almost the same, but has many advantages
  - Can share file caches kept in memory
  - Threads are cheaper to create than processes (lower per-request overhead)
- What about denial-of-service (DoS) attacks?

Example: Web Server (cont.)

- Problem with previous version: unbounded number of threads
  - When web-site becomes too popular, throughput sinks
- Solution: allocate bounded "pool" of threads, representing max level of parallelism




```
master() {
  allocThreads(worker, maxLevel);
  while(TRUE) {
    con = acceptCon();
    enqueue(queue, con);
    broadcastToWorkers();
  }
}
```

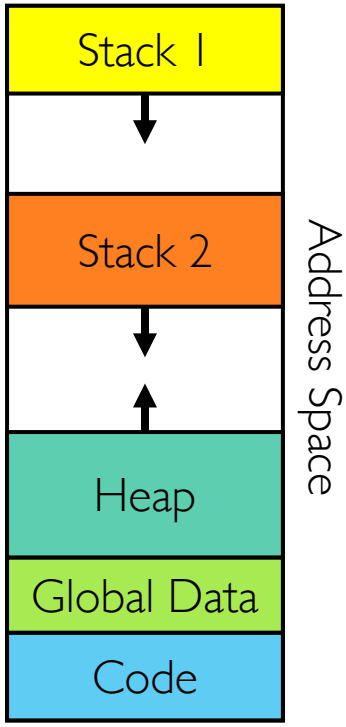
```
worker(queue) {
  while(TRUE) {
    con = dequeue(queue);
    if (con == null)
      waitForSignal();
    else
      serviceWebPage(con);
  }
}
```

Multiple Processes vs. Single Process With Multiple Threads

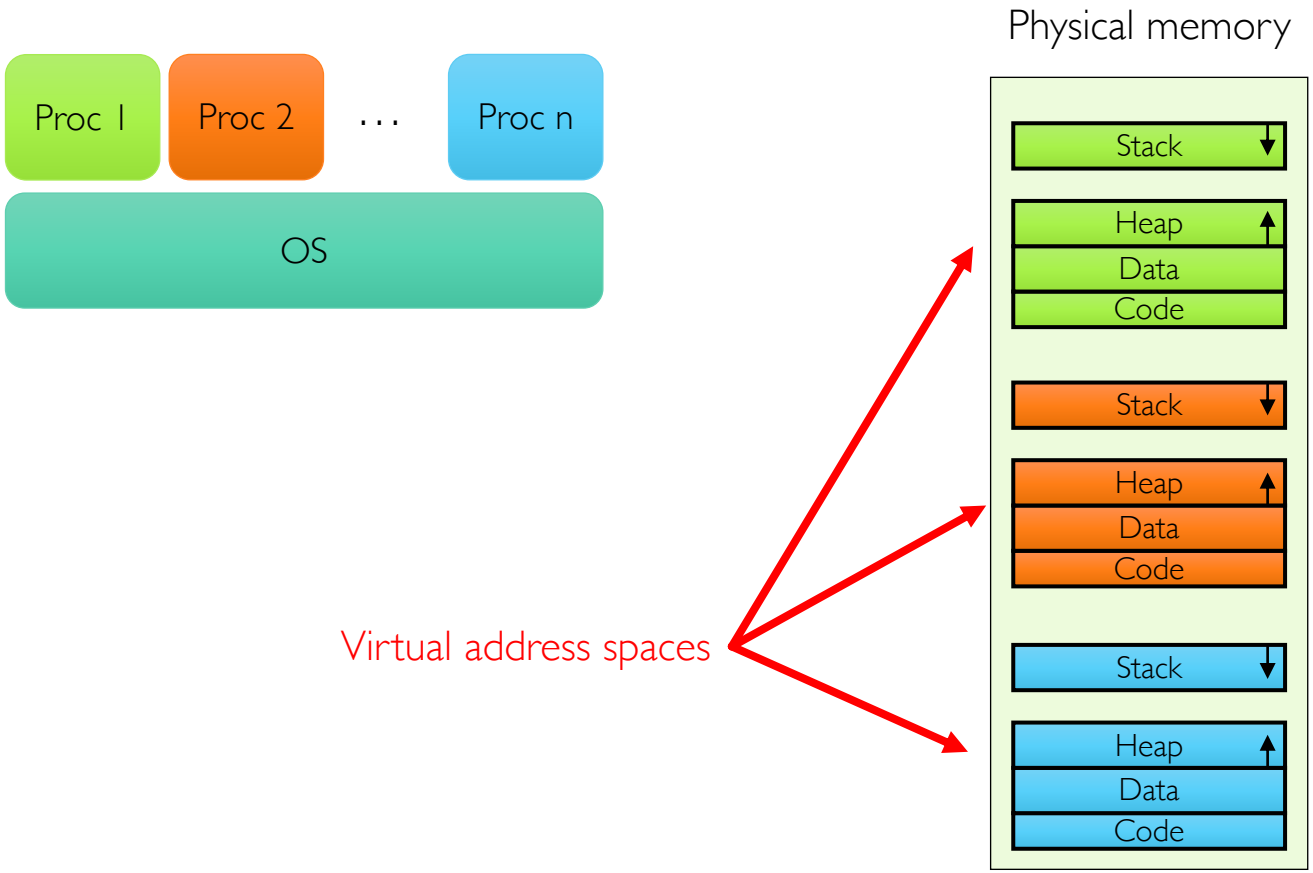
- Fundamental tradeoff between protection and efficiency
- Communication harder between processes
  - This is basically IPC
  - It necessarily involves OS
- Communication easier within a process
  - All threads of process share state and resources of process
  - If one thread opens a file, other threads in the process can access it
  - It does not involve OS

Memory Footprint of Multiple Threads

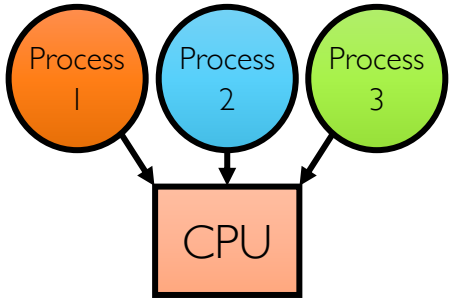
- How do we position stacks relative to each other?
- What maximum size should we choose for stacks?
  - 8KB for kernel-level stacks in Linux on x86
  - Less need for tight space constraint for user-level stacks
- What happens if threads violate this?
  - "… program termination and/or corrupted data"
- How might you catch violations?
  - Place guard values at top and bottom of each stack
  - Check values on every context switch

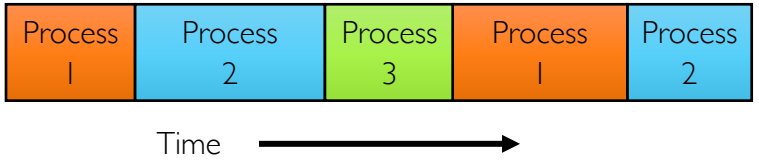


Multiprogramming: Running Multiple Processes



Time Sharing: Multiprogramming on Single CPU





- Illusion: infinite number of processors
  - Each thread runs on dedicated virtual processor
- Reality: few processors, multiple threads running at variable speed
- How can we give illusion of infinite number of processors?
  - Multiplex in time!
- How do we switch from one process to next?
  - Save PC, SP, and registers in current PCB
  - Load PC, SP, and registers from new PCB
- What triggers switch?
  - Timer, voluntary yield, I/O interrupts, …

How Do We Multiplex Processes?

- Scheduling: OS decides which process uses CPU time
  - Only one process is "running" on each CPU at any time
  - Scheduler could give more time to *important* processes

- Protection: OS divides non-CPU resources among processes
  - E.g., give each process their own address space
  - E.g., multiplex I/O through system calls

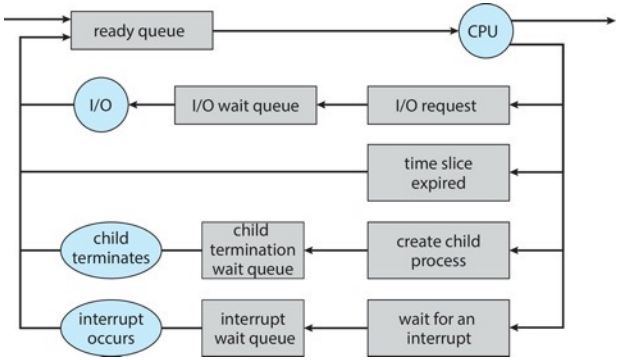
Scheduling

- Kernel scheduler decides which processes/threads receive CPU
- There are variety of scheduling policies for …
  - Fairness or
  - Realtime guarantees or
  - Latency optimization or …
- Kernel scheduler maintains data structure containing PCBs


```
if (readyProcesses(PCBs)) {
      nextPCB = selectProcess(PCBs);
      run(nextPCB);
} else {
      run_idle_process();
}
```

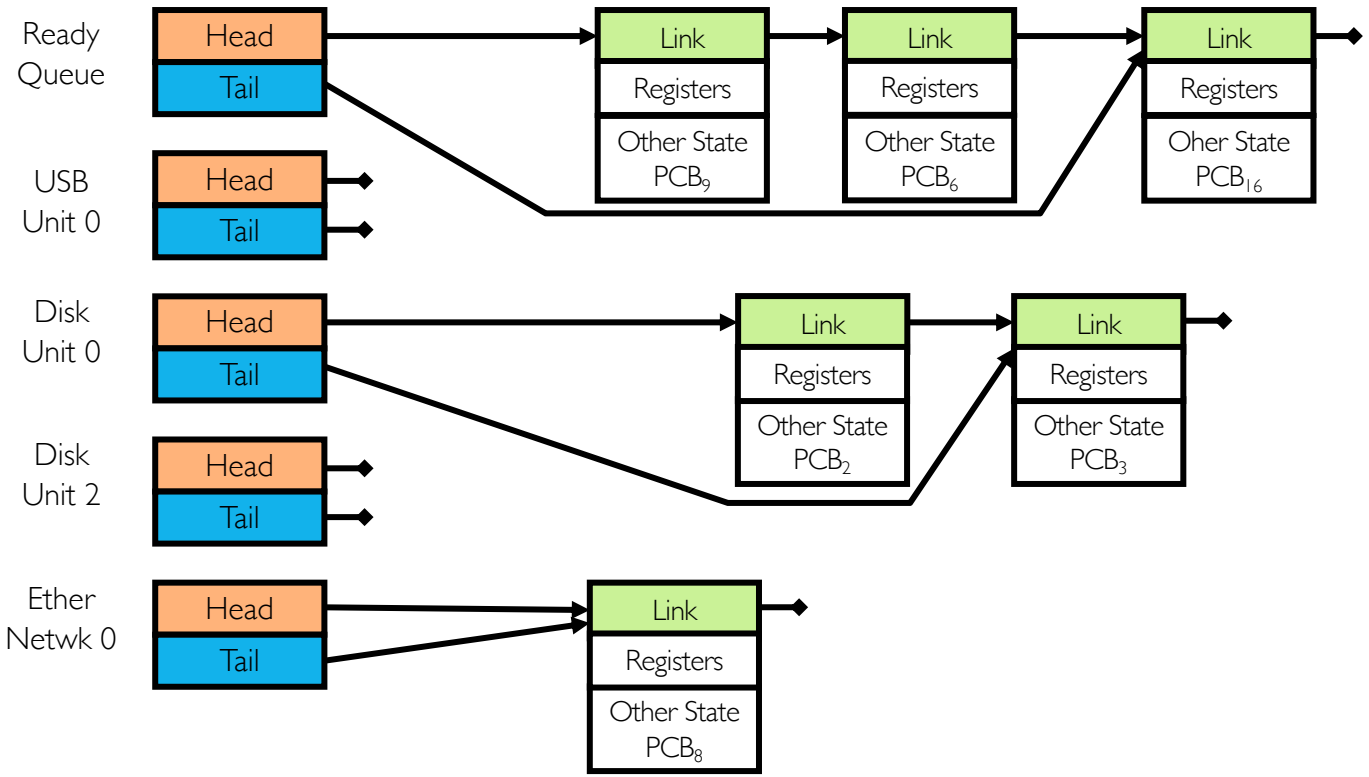
Ready Queue

- PCBs move from queue to queue as they change state
  - Decisions about which order to remove from queues are scheduling decisions
  - Many algorithms possible (more on this in a few weeks)



Ready Queue And I/O Device Queues

- Process not running ⇒ PCB is in some scheduler queue
  - Separate queue for each device/signal/condition
  - Each queue can have different scheduler policy



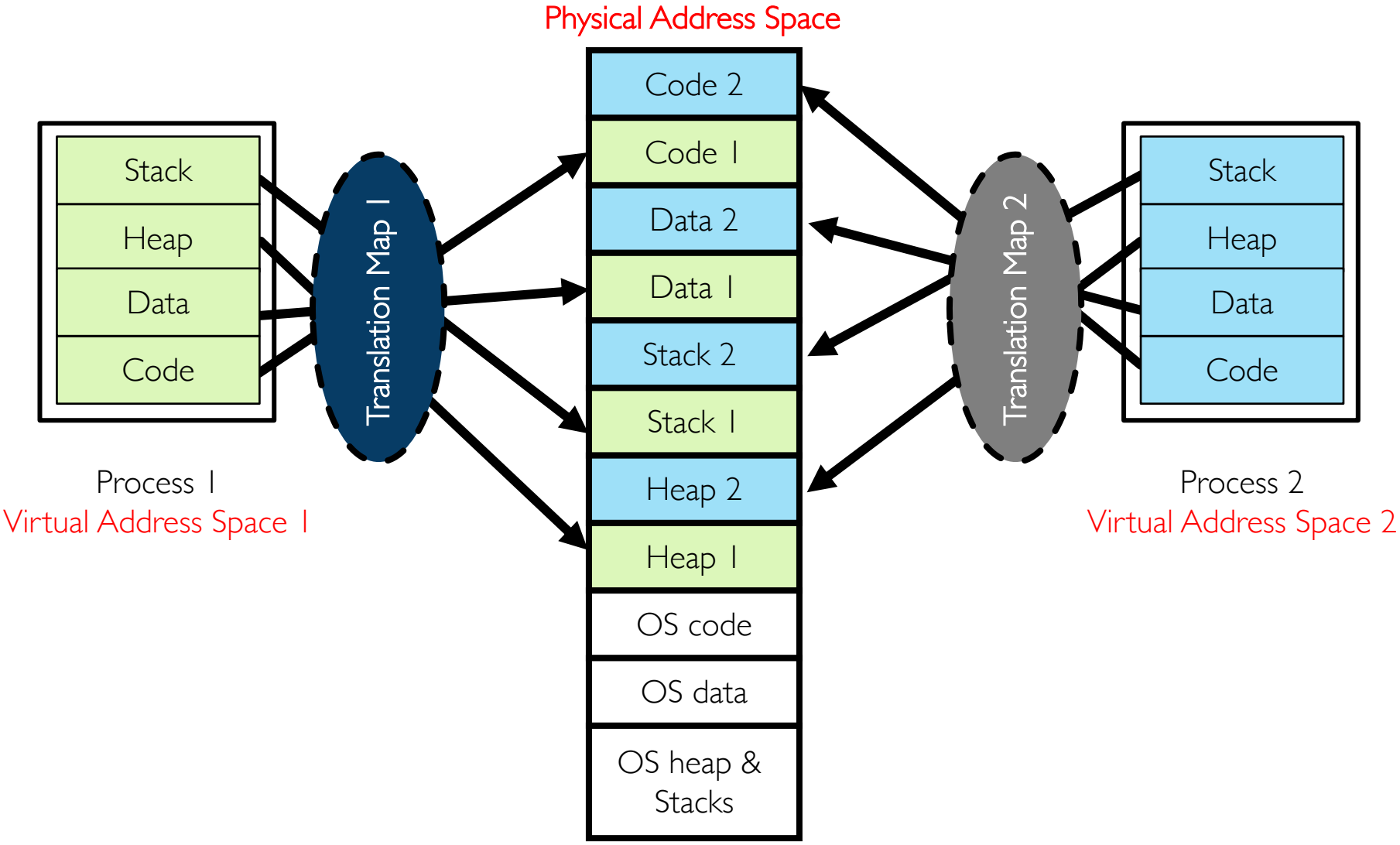
Protection

- OS must protect itself from user programs
  - Reliability: prevent OS from crashing
  - Security: limit scope of what processes can do
  - Privacy: limit data each process can access
  - Fairness: enforce appropriate share of HW
- It must protect user programs from one another
- Main method is to limit translation from virtual to physical address space

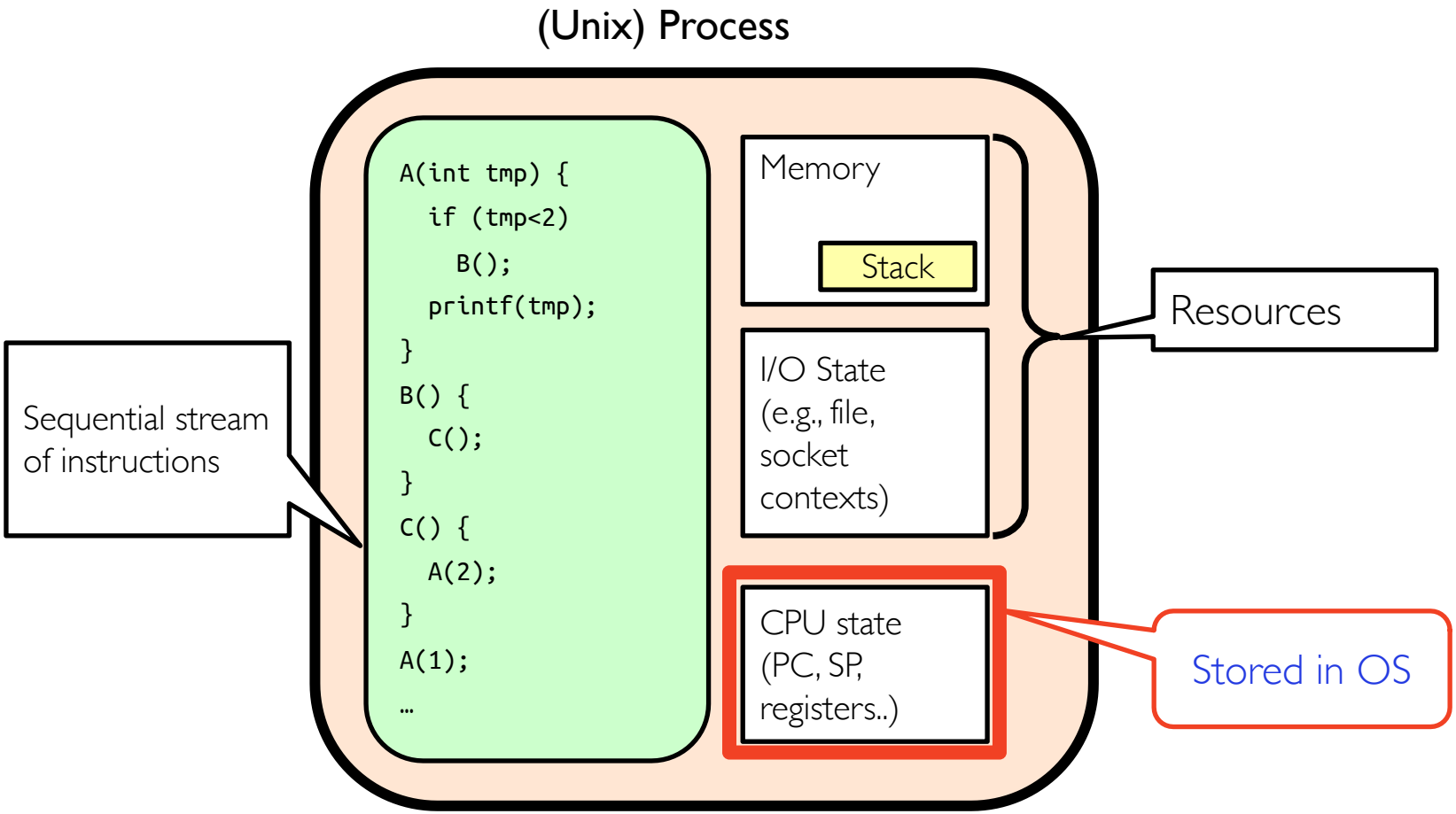
How to Protect Processes from One Another?

- Protection of memory
  - Every process does not have access to all memory
- Protection of I/O devices
  - Every process does not have access to every device
- Protection of access to processor
  - Preemptive switching from process to process
  - Use of timer
  - Must not be possible to disable timer from user code

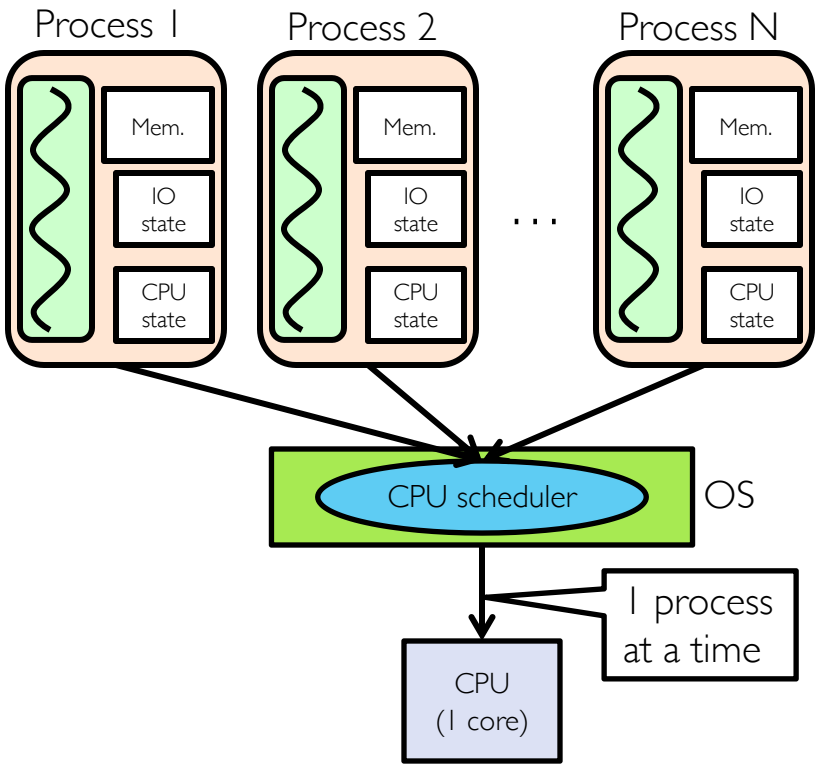
Address Translation Maps: Illusion of Separate Address Space



Putting it Together: Process

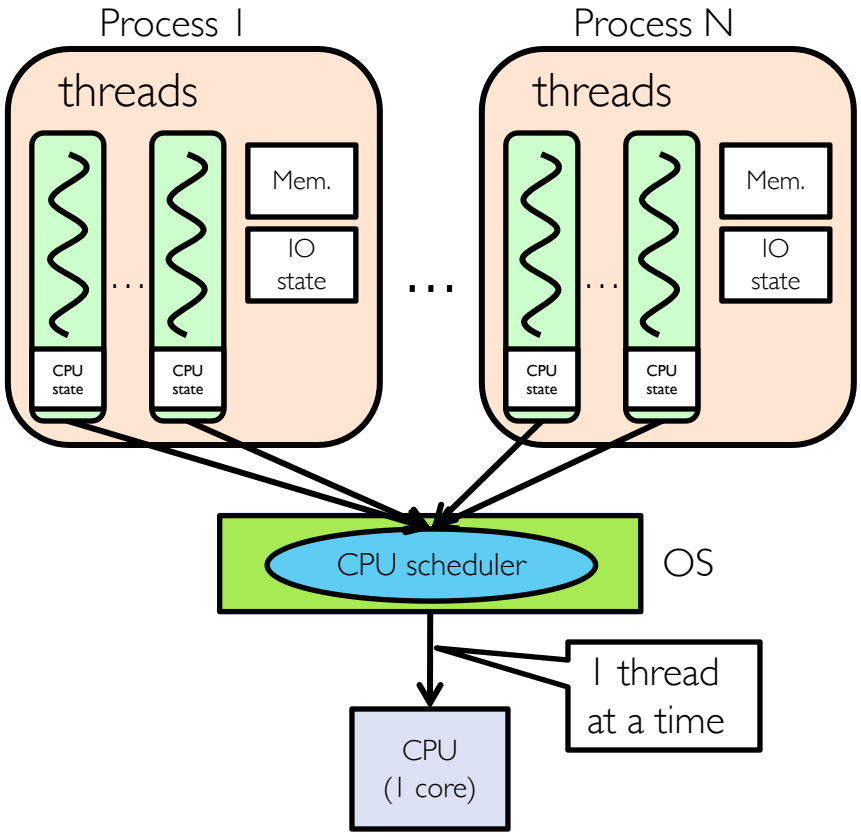


Putting it Together: Processes



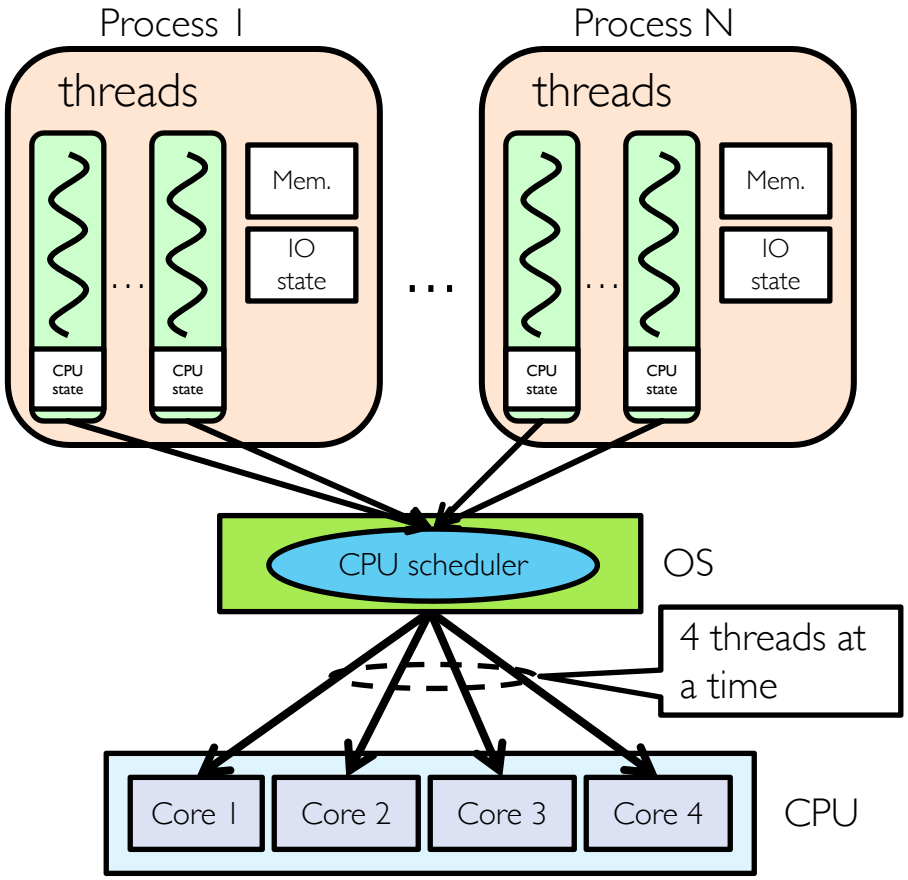
- Switch overhead: high
  - CPU state: low
  - Memory/IO state: high
- Process creation: high
- Protection
  - CPU: yes
  - Memory/IO: yes
- Sharing overhead: high (involves at least one context switch)

Putting it Together: Threads



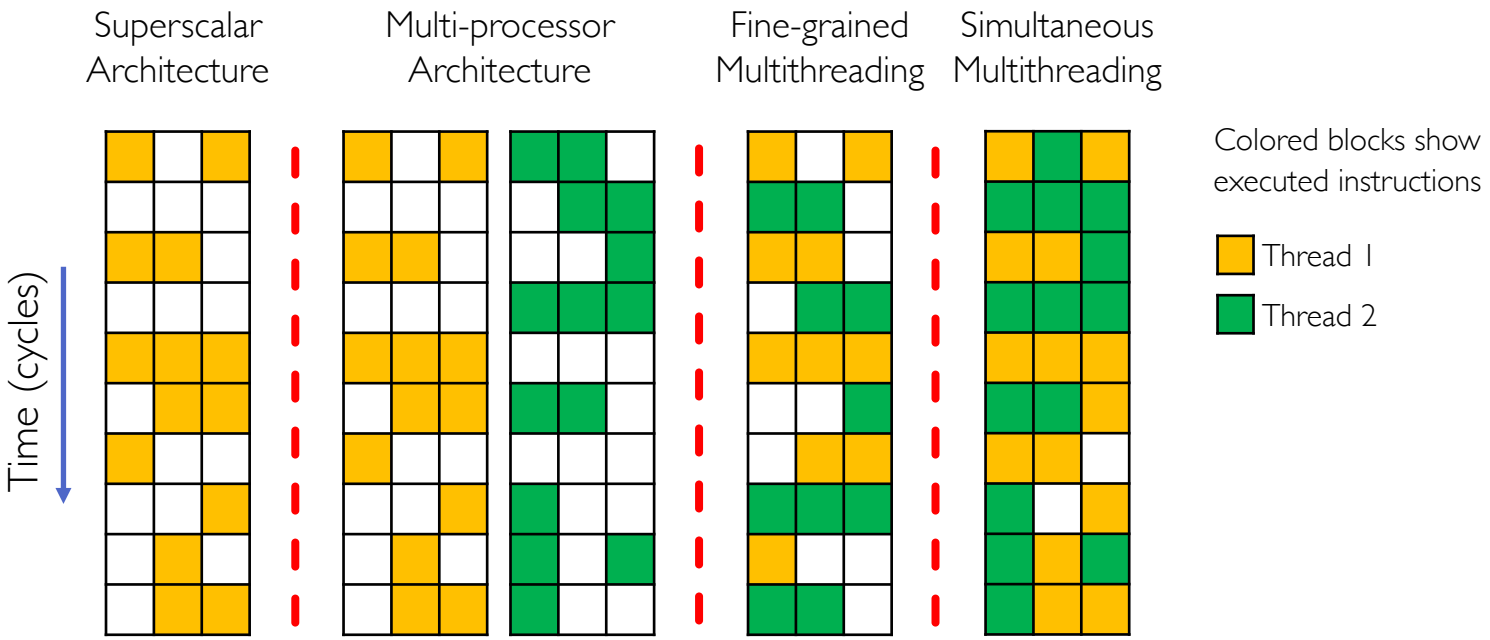
- Switch overhead: medium
  - CPU state: low
- Thread creation: medium
- Protection
  - CPU: yes
  - Memory/IO: no
- Sharing overhead: low(ish) (thread switch overhead low)

Putting it Together: Multi-cores



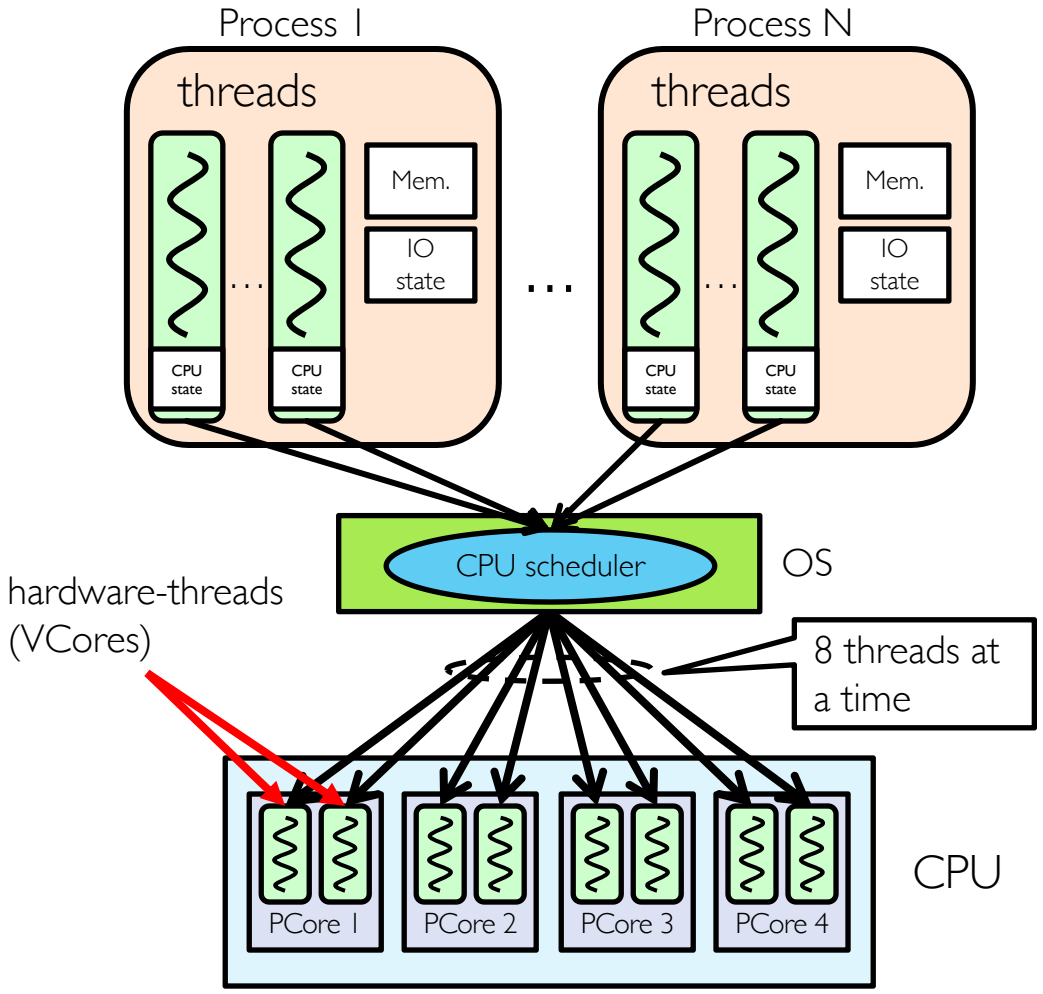
- Switch overhead: low (only CPU state)
- Thread creation: low
- Protection
  - CPU: yes
  - Memory/IO: no
- Sharing overhead: low (thread switch overhead low, may not need to switch at all!)

Hyperthreading



- Superscalar processors can execute multiple instructions that are independent
- Multiprocessors can execute multiple independent threads
- Fine-grained multithreading executes two independent threads by switches between them
- Hyperthreading duplicates register state to make second (hardware) "thread" (virtual core)
  - From OS's point of view, virtual cores are separate CPUs
  - OS can schedule as many threads at a time as there are virtual cores (but, sub-linear speedup!)
  - See: http://www.cs.washington.edu/research/smt/index.html

Putting it Together: Hyperthreading



- Switch overhead between hardware-threads: very-low (done in hardware)
- Contention for ALUs/FPUs may hurt performance

Dual-mode Operation (4th OS Concept)

- Hardware provides at least two modes
  - Kernel mode (or "supervisor" or "protected")
  - User mode, which is how normal programs are executed

- How can hardware support dual-mode operation?
  - Single bit of state (user/system mode bit)
  - Certain operations/actions only permitted in system/kernel mode
    - In user mode they fail or trap
  - User to kernel transition sets system mode AND saves user PC
    - OS code carefully puts aside user state then performs necessary actions
  - Kernel to user transition clears system mode AND restores user PC
    - E.g., `rfi`: return-from-interrupt

Three Types of Mode Transfer

- System call: request for kernel services
  - E.g., **`open`**, **`close`**, **`read`**, **`write`**, **`lseek`**
  - Usually implemented by calling *trap* or *syscall* instruction
    - Special instruction is not strictly required; on some systems, processes trigger system calls by executing some instruction with specific invalid opcode

- Processor exception: internal, synchronous, hardware event
  - E.g., divide by zero, illegal instruction, segmentation fault, page fault
  - Caused by software behavior

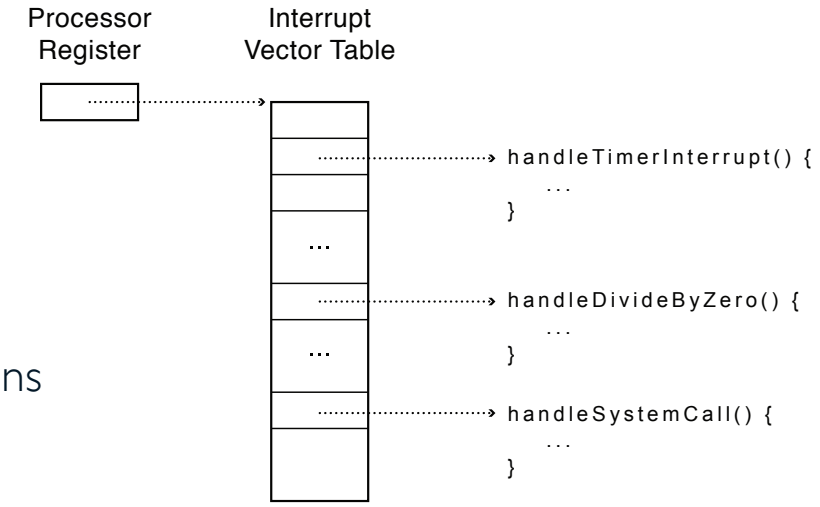
- Interrupt: external asynchronous event
  - E.g., timer, disk ready, network
  - Interrupts can be disabled, exceptions and traps cannot!

Requirements for Safe Mode Transfer

- Limited entry into kernel
  - HW must ensure entry point into kernel is one set up by kernel
  - User programs cannot be allowed to jump to arbitrary locations in kernel
- Atomic changes to processor state
  - In user mode, PC and SP point to memory locations in user process
  - In kernel mode, PC and SP point to memory locations in kernel
  - Mode, PC, SP, and memory protection should all change atomically
- Transparent, restartable execution
  - User-level process could get interrupted between any two instructions
  - OS must restore state of user process exactly as it was before interrupt

Interrupt Vector Table

- Table set up by OS pointing to code to run on system calls, processor exceptions, and interrupts



- On x86, vector numbers 0-31 are for different types of processor exceptions (e.g., divide-by-zero)
- Vector numbers 32-255 are for different types of interrupts (e.g., timer)
- Vector number 64 is for system call handler

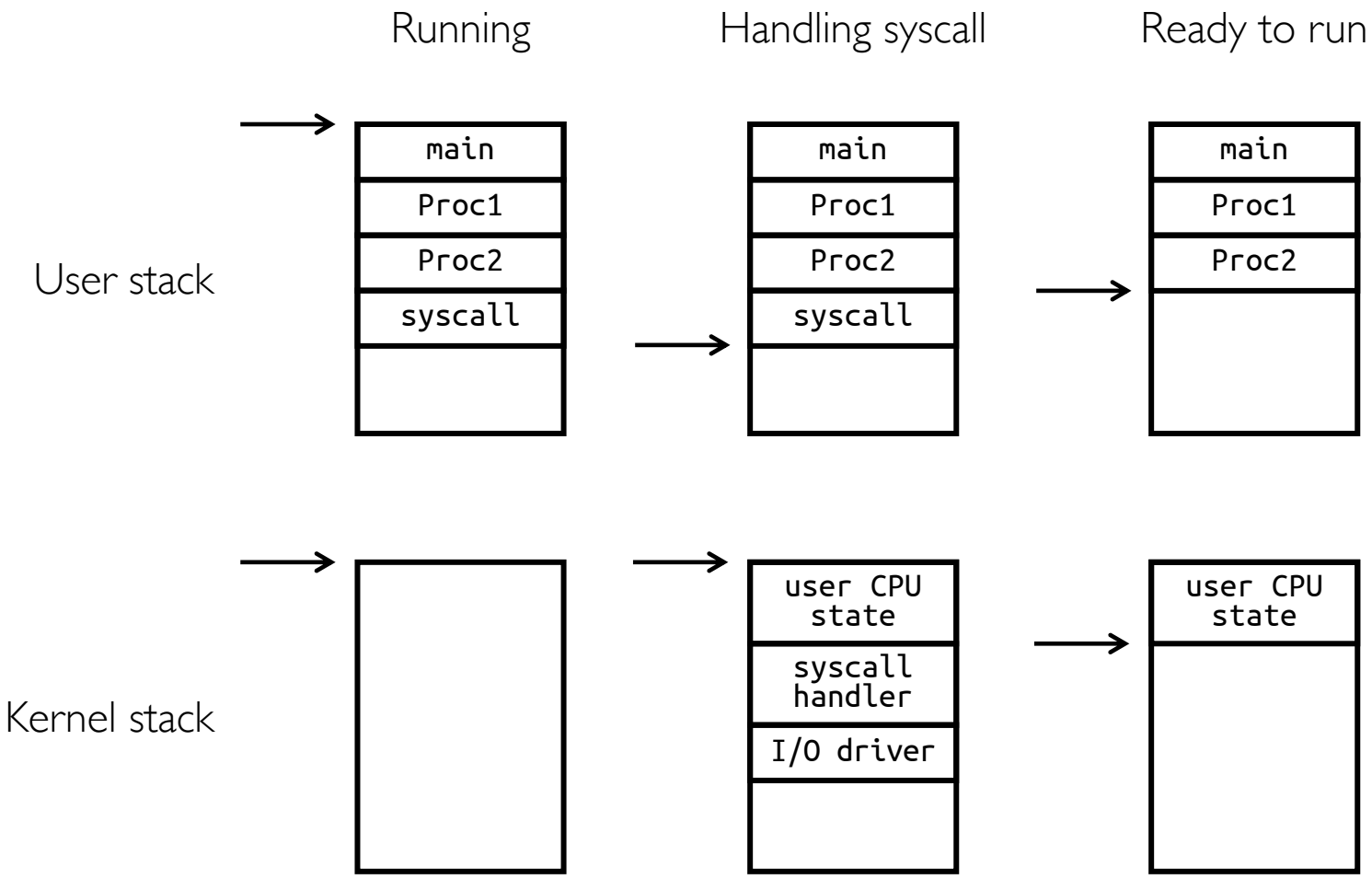
Interrupt Stack

- User process state should be saved

- OS should not save anything on user stack (why?)
  - Reliability: what if user program's SP is not valid?
  - Security: what if other threads in process change kernel's return address?

- Most OSes go one step further and allocate separate kernel interrupt stack (also called kernel stack) for each user-level thread
  - PCB could store pointer to kernel stack

Two-stack Model Example



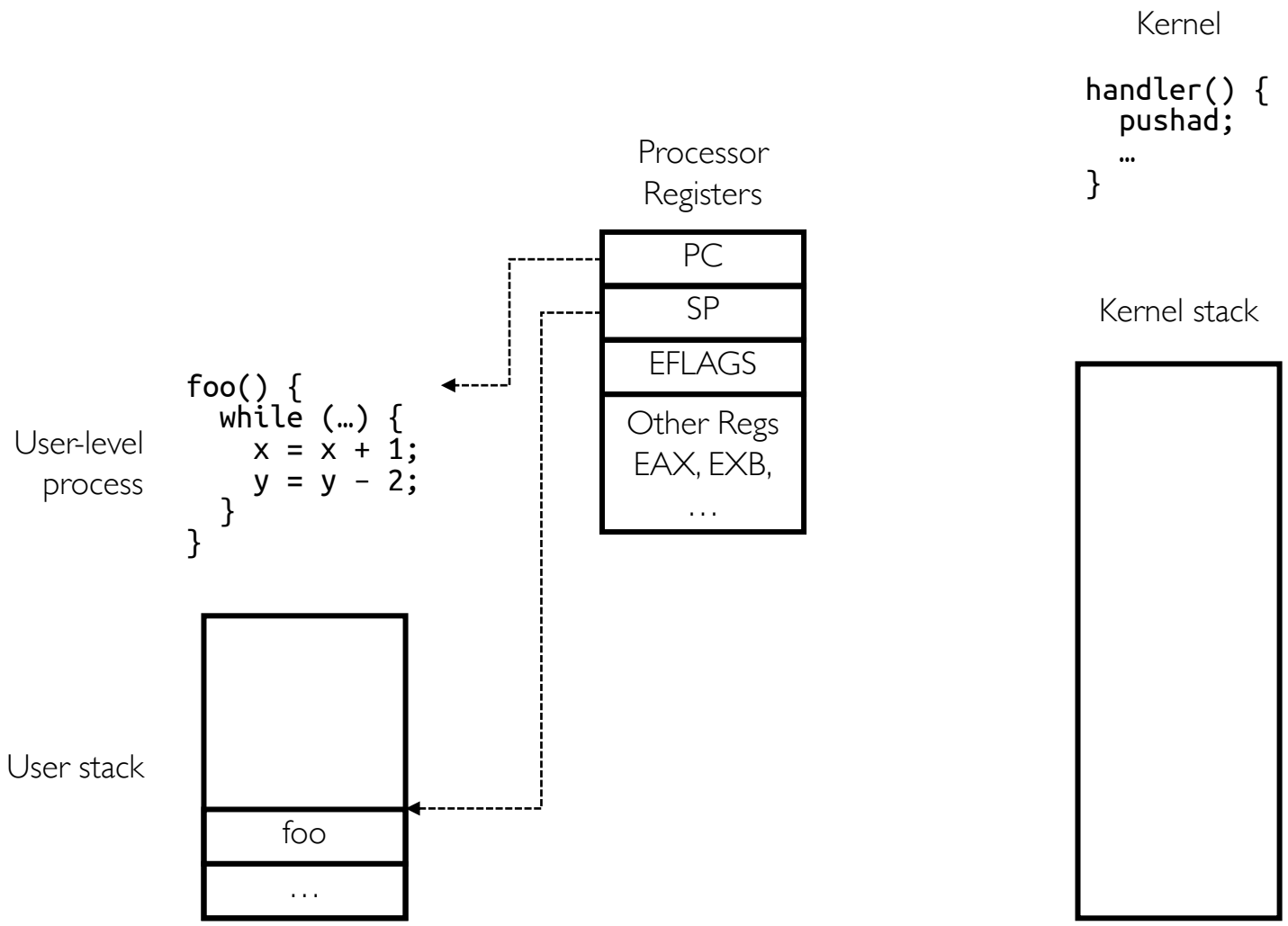
Interrupt Masking

- Interrupt handler runs with interrupts *disabled*
- This simplifies interrupt handling
- Interrupts are re-enabled when interrupt completes
- Interrupts are deferred (masked) not ignored
- HW buffers new interrupts until interrupts are re-enabled
- If interrupt are disabled for long time, some interrupts may be lost
- On x86, **`cli`** disables interrupts and **`sti`** enables interrupts
  - Only applies to current CPU (on a multicore)
  - User programs cannot use these instructions (why?)

Mode Transfer Steps in x86

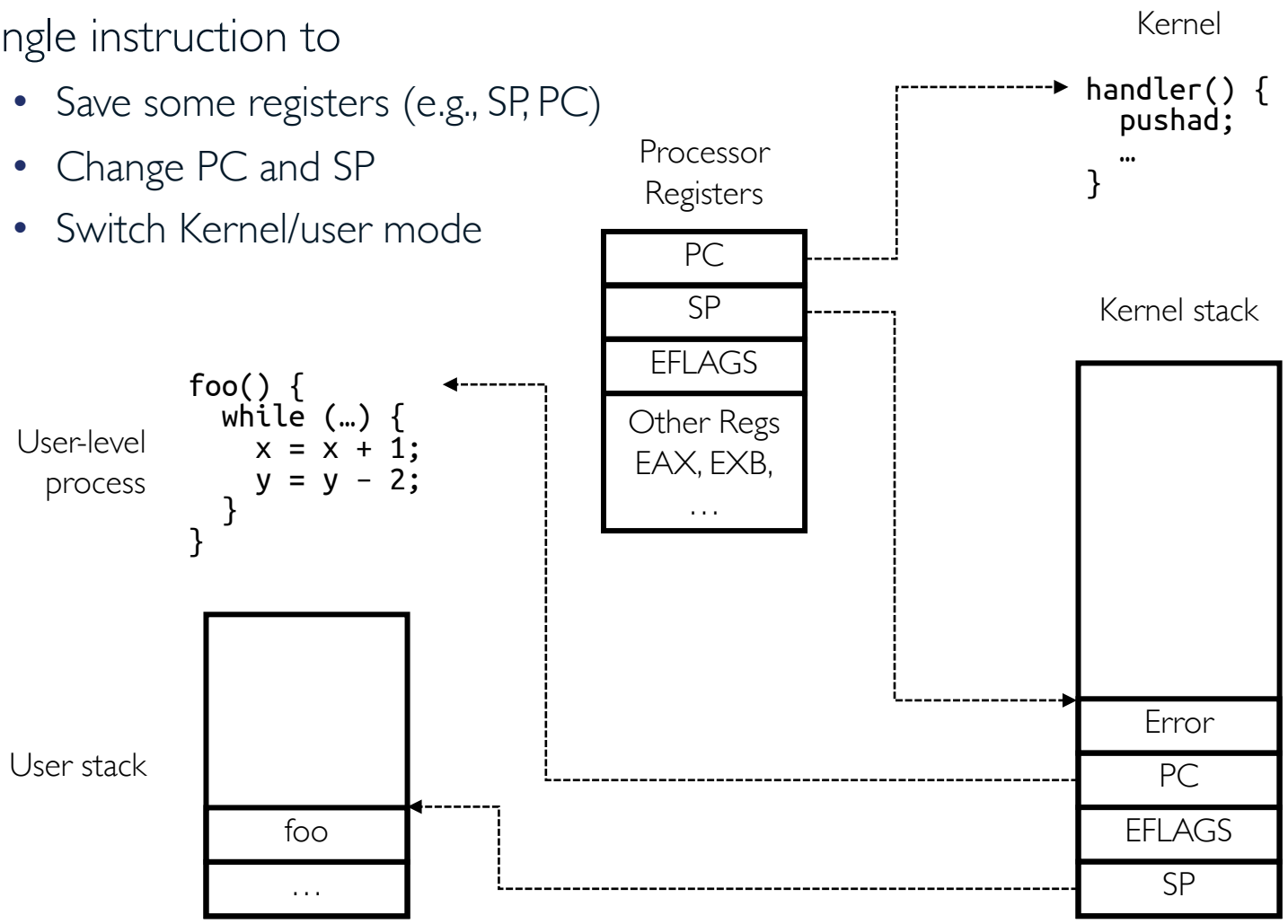
- Mask interrupts
- Save PS, SP, and execution flags in temporary HW registers
- Switch onto kernel interrupt stack (specified in special HW register)
- Push the three key values onto interrupt stack
- Optionally save an error code
- Invoke interrupt handler

Example: x86 Mode Transfer



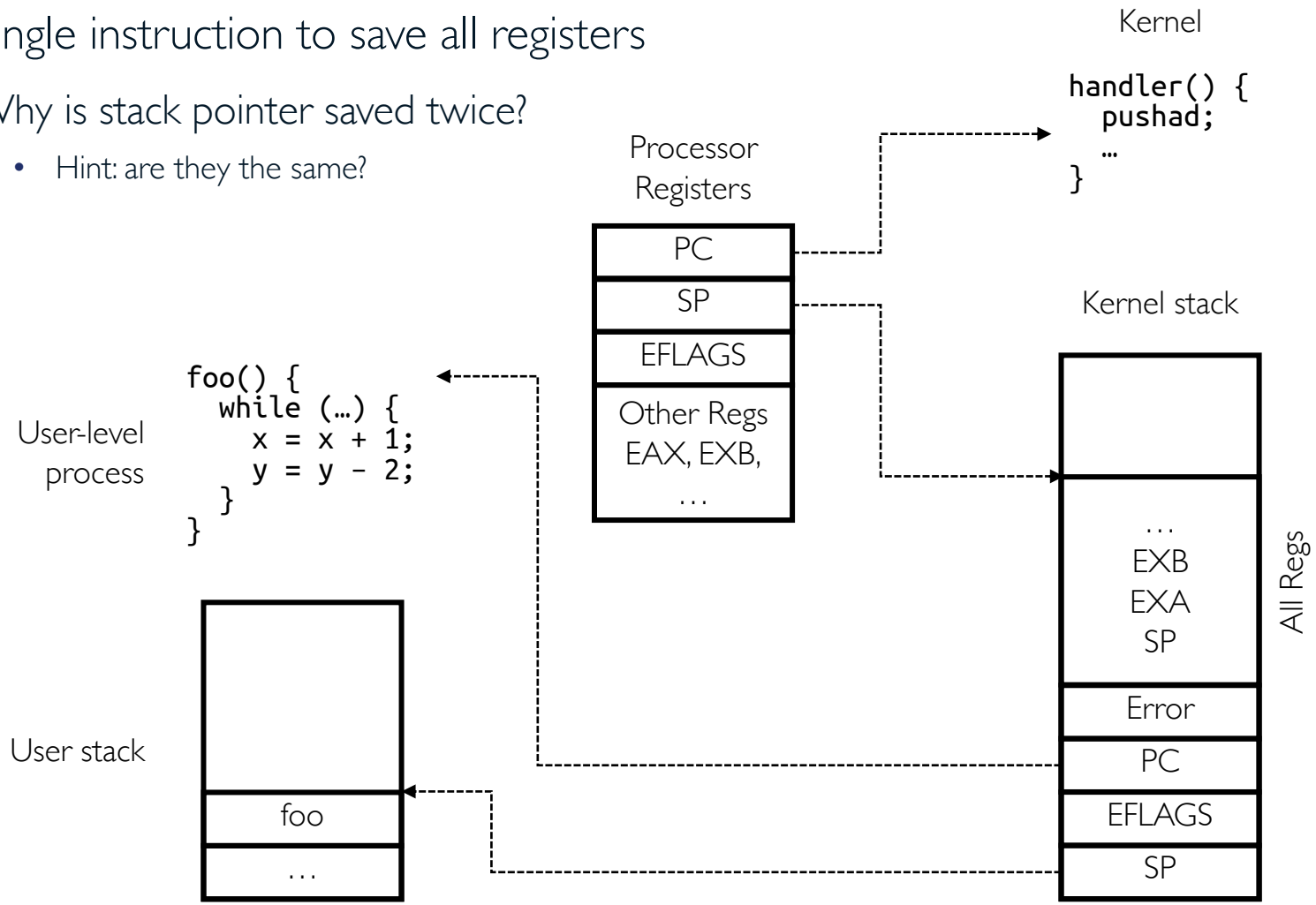
Example: x86 Mode Transfer (cont.)

- Single instruction to
  - Save some registers (e.g., SP, PC)
  - Change PC and SP
  - Switch Kernel/user mode



Example: x86 Mode Transfer (cont.)

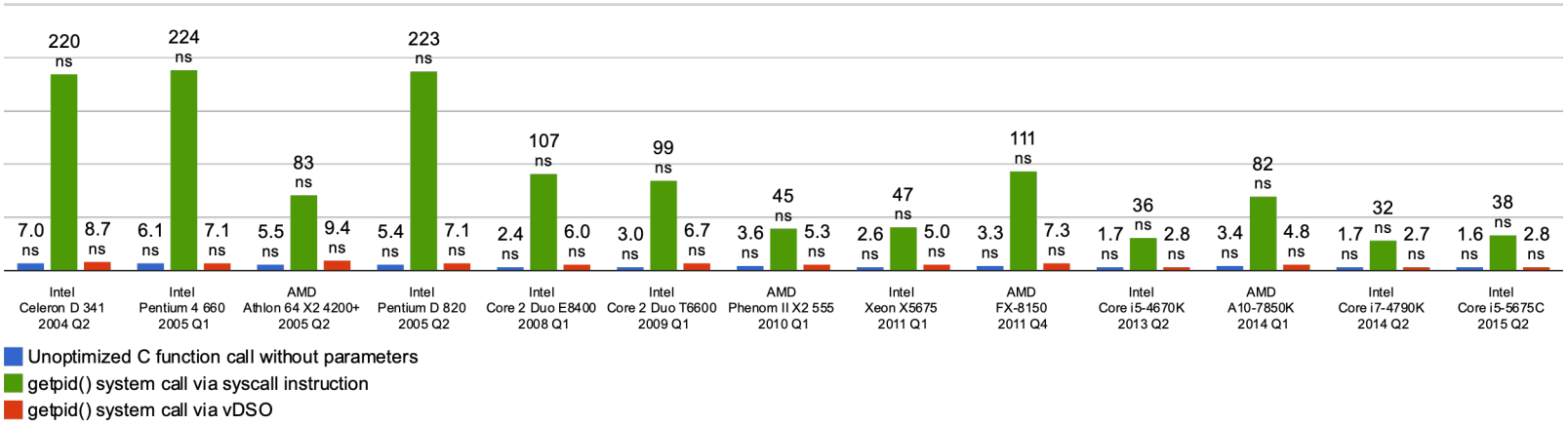
- Single instruction to save all registers
- Why is stack pointer saved twice?
  - Hint: are they the same?



Example: System Call Handler

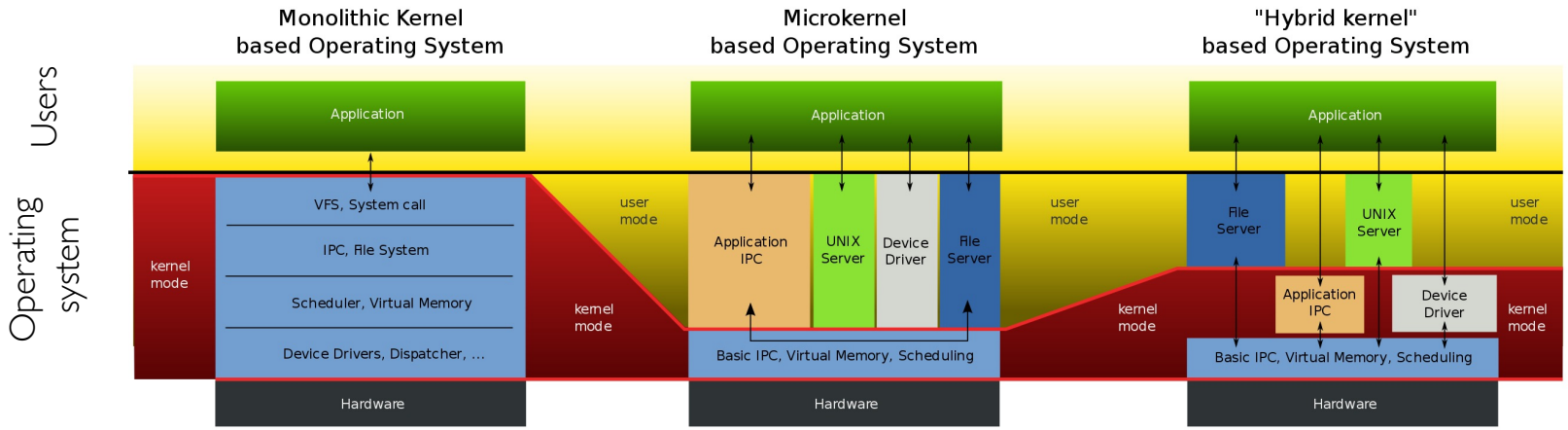
- Vector through well-defined system call entry points!
  - Table mapping system call number to handler
- Locate arguments
  - In registers or on user (!) stack
- Copy arguments (copy before check)
  - From user memory into kernel memory
  - Protect kernel from malicious code evading checks
- Validate arguments
  - Protect kernel from errors in user code
- Copy results back
  - Into user memory

Basic Cost of System Calls



- Min syscall has ~ 25x cost of function call
- Linux vDSO (virtual dynamic shared object) runs some system calls in user space
  - E.g., gettimeofday or getpid

Aside: Monolithic vs Microkernel OS



en.wikipedia.org

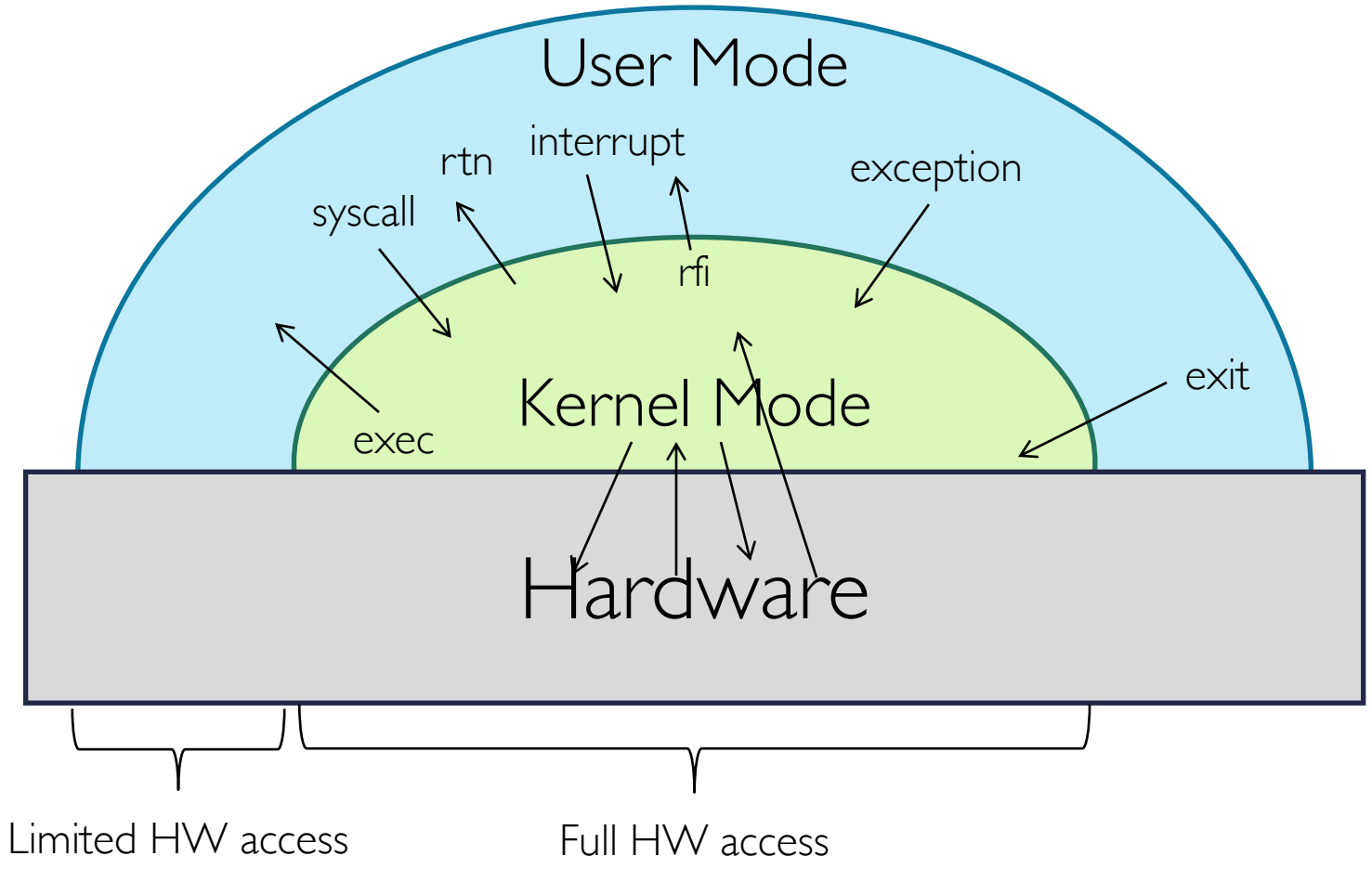
Aside: Influence of Microkernels

- Microkernels provide better modularity, security, and fault tolerance, but they introduce higher communication overhead
  - Too many context switches
- Many OSes provide some services externally, like microkernels
  - OS X and Linux: windowing (graphics and UI)
- Some currently monolithic OSes started as microkernels
  - Windows family originally had microkernel design
  - OS X is hybrid of Mach microkernel and FreeBSD monolithic kernel

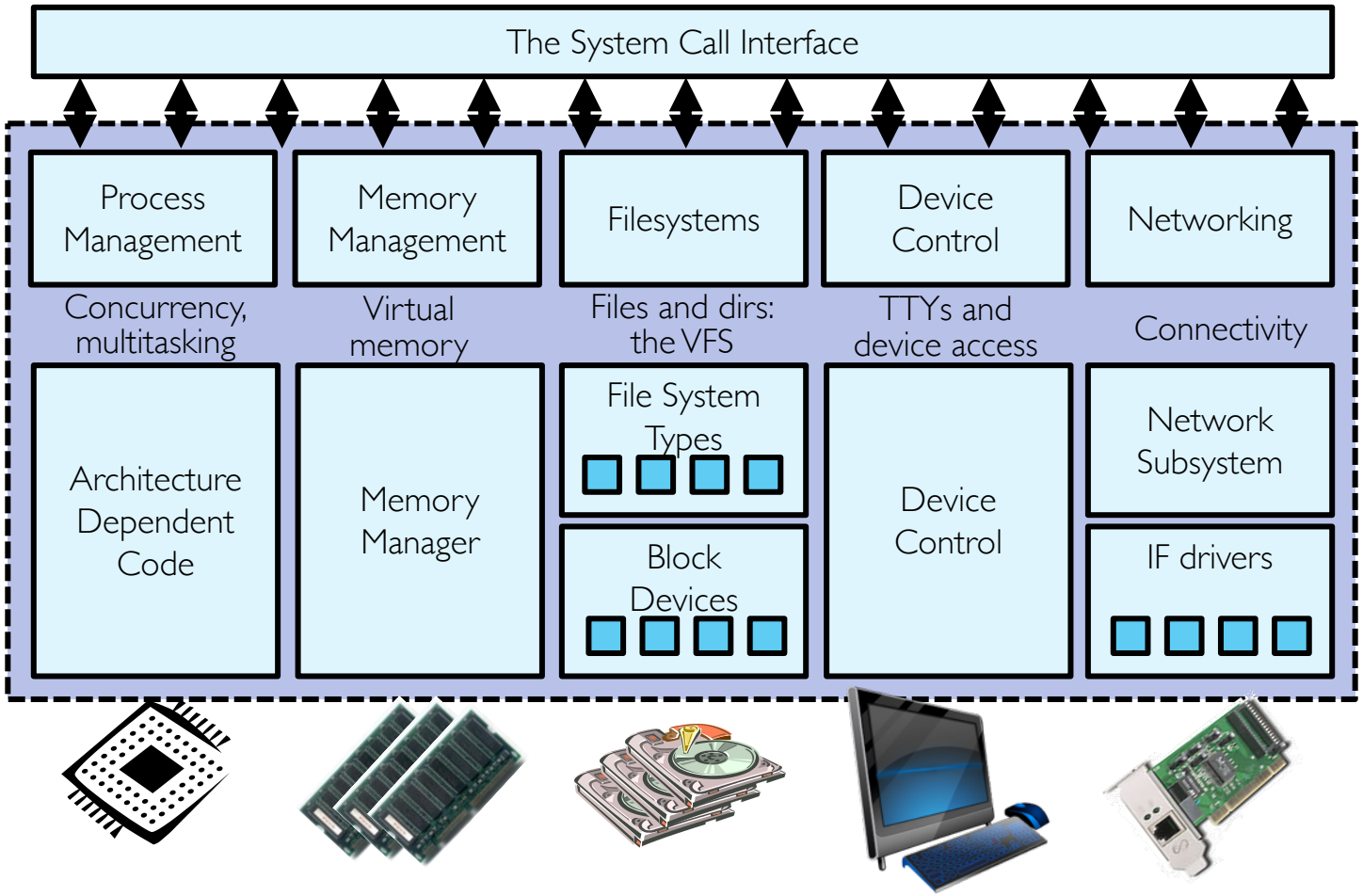
Kernel to User Mode Switch Examples

- New process/new thread start
  - Jump to first instruction in program/thread
- Return from interrupt, exception, system call
  - Resume suspended execution
- Process/thread context switch
  - Resume some other process
- User-level *upcall* (UNIX *signal*)
  - Asynchronous notification to user program
    - Preemptive user-level threads
    - Asynchronous I/O notification
    - Interprocess communication
    - User-level excepting handling
    - User-level resource allocation

Example: User/Kernel Mode Transfers



System Call Interface: Access Point to Hardware Resources

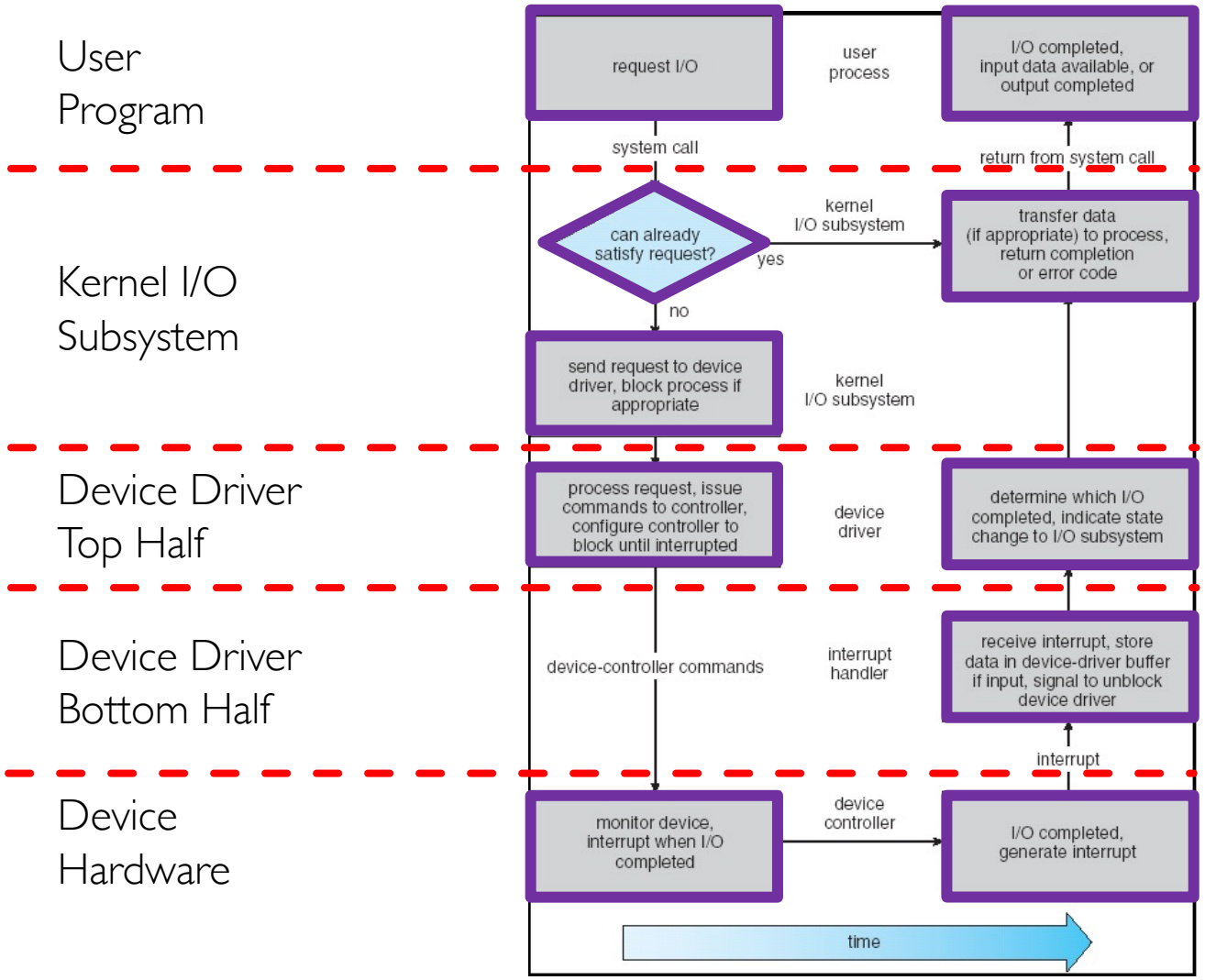


Device Drivers

- Device-specific code in kernel that interacts directly with device hardware
  - Supports standard, internal interface
  - Same kernel I/O system can interact easily with different device drivers
  - Special device-specific configuration supported with `ioctl()` syscall

- Device drivers are typically divided into two pieces
  - Top half: accessed in call path from system calls
    - implements a set of standard, cross-device calls like `open()`, `close()`, `read()`, `write()`, `ioctl()`, etc.
    - This is kernel's interface to device driver
    - Top half will start I/O to device, may put thread to sleep until finished
  - Bottom half: run as interrupt routine
    - Gets input or transfers next block of output
    - May wake sleeping threads if I/O now complete

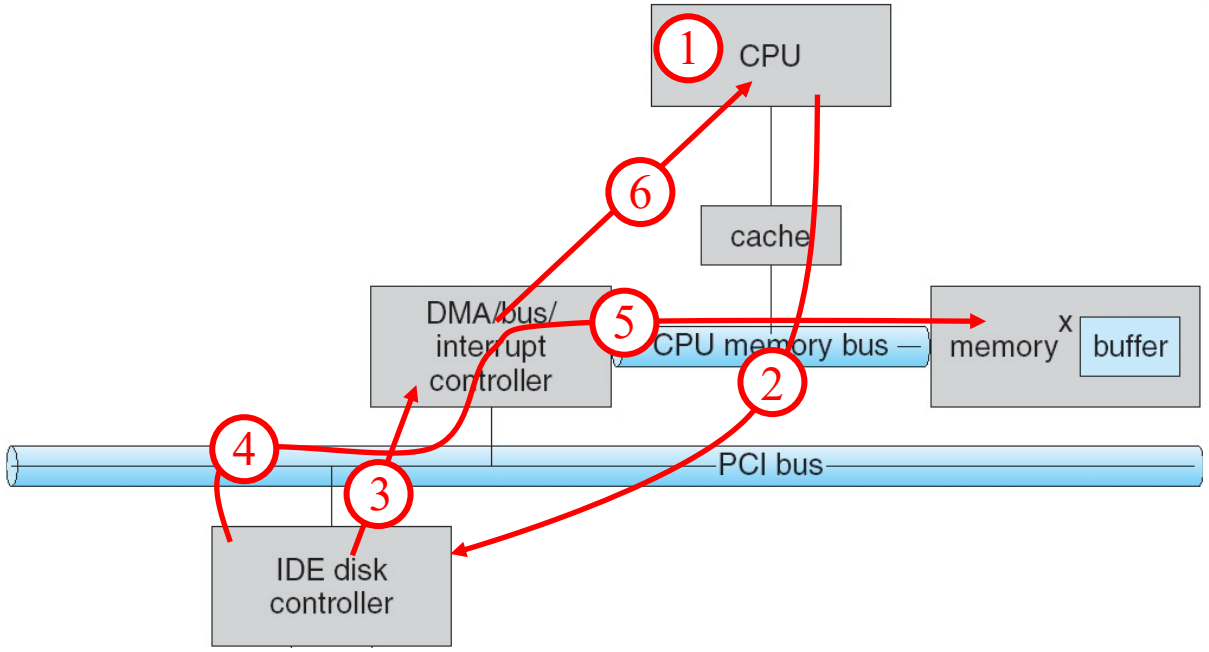
Life Cycle of an I/O Request



I/O Data Transfer

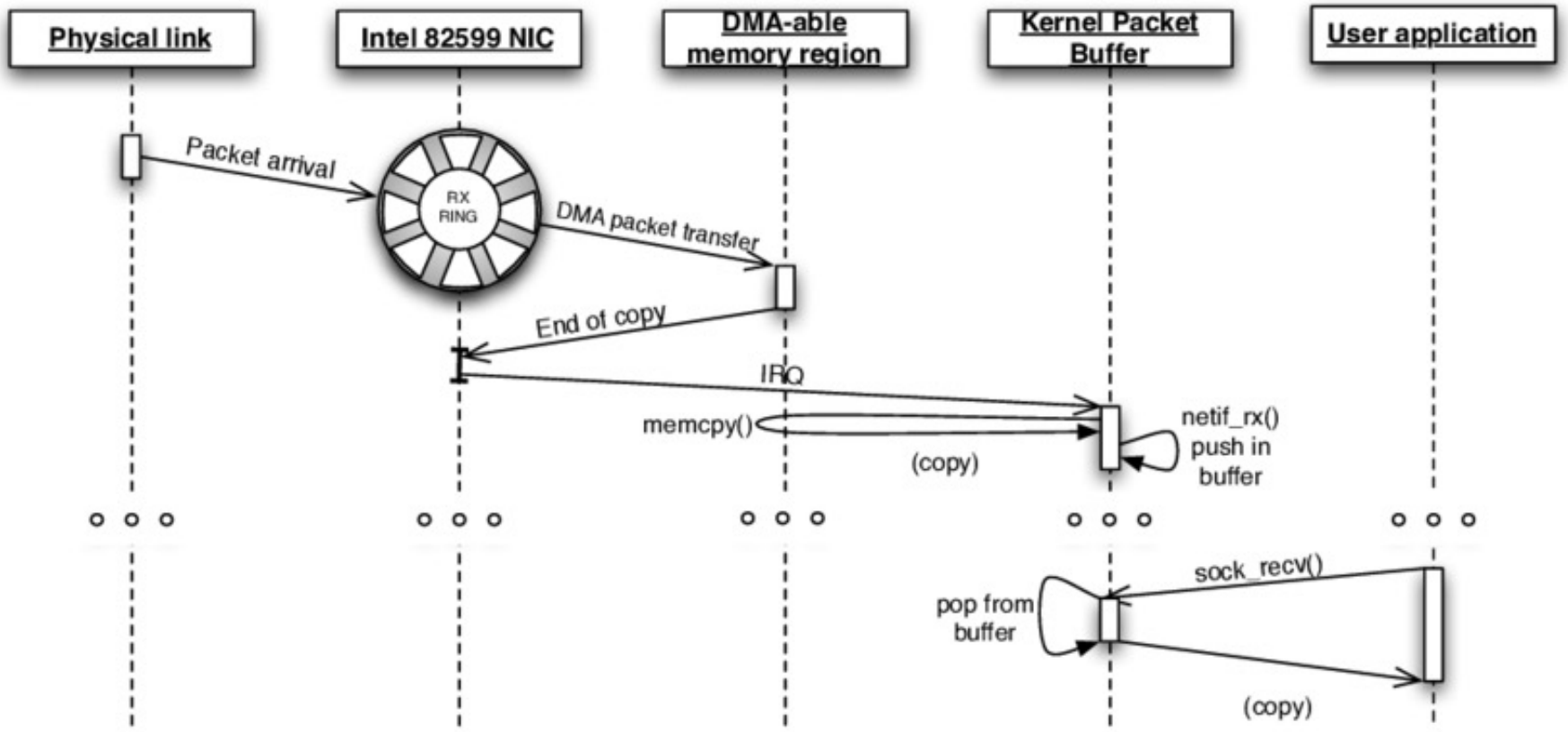
- Programmed I/O
  - Each byte transferred via processor in/out or load/store
  - + Simple hardware, easy to program
  - − Consumes processor cycles proportional to data size
- Direct memory access (DMA)
  - Give controller access to memory bus
  - Ask it to transfer data blocks to/from memory directly

DMA Transfer



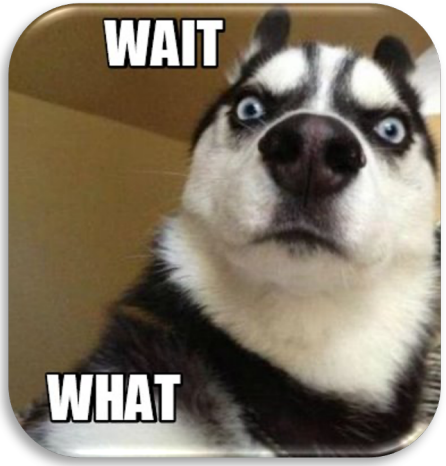
1. Device driver is told to transfer disk data to buffer at address x
2. Device driver tells disk controller to transfer C bytes from disk to buffer at address x
3. Disk controller initiates DMA transfer
4. Disk controller send each byte to DMA controller
5. DMA controller transfers bytes to buffer x, increasing address and decreasing C
6. When C = 0, DMA interrupts CPU to signal transfer completion

DMA Example: Network Stack in Linux Kernels before 2.6



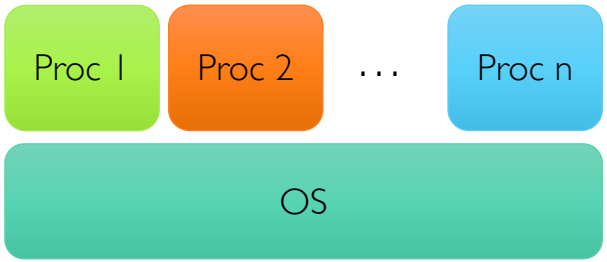
García-Dorado, José Luis, et al. "High-performance network traffic processing systems using commodity hardware." Data traffic monitoring and analysis. Springer, Berlin, Heidelberg, 2013. 3-27.

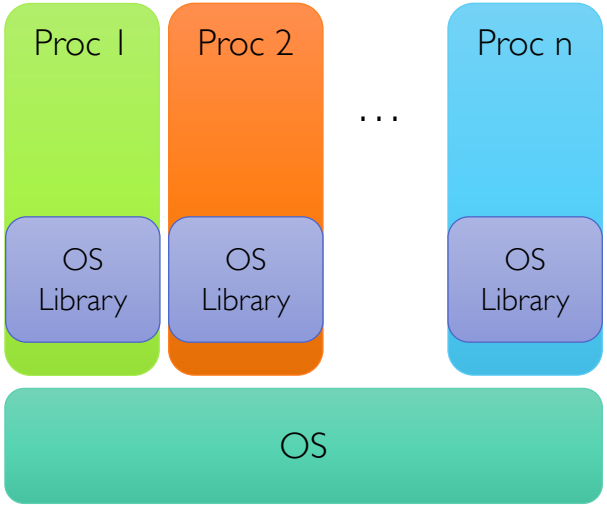
How Does Kernel Provide Services?

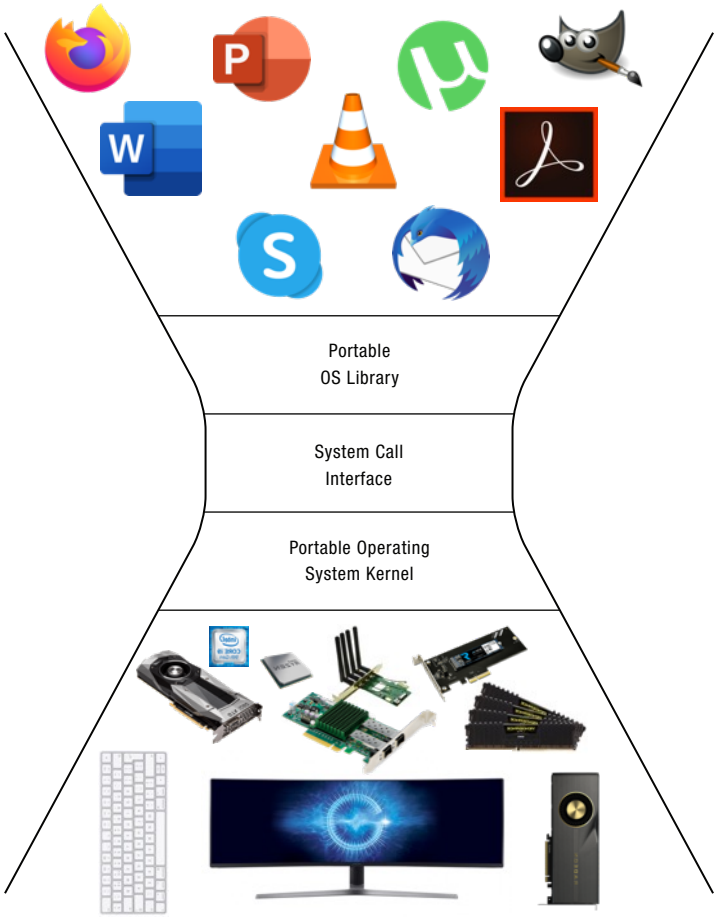


- You said that applications request services from OS via syscall, but …
  - I've been writing all sorts of applications, and I never ever saw a "syscall" !!!
- That's right!
- It was buried in the programming language runtime library (e.g., libc.a)
  - … Layering

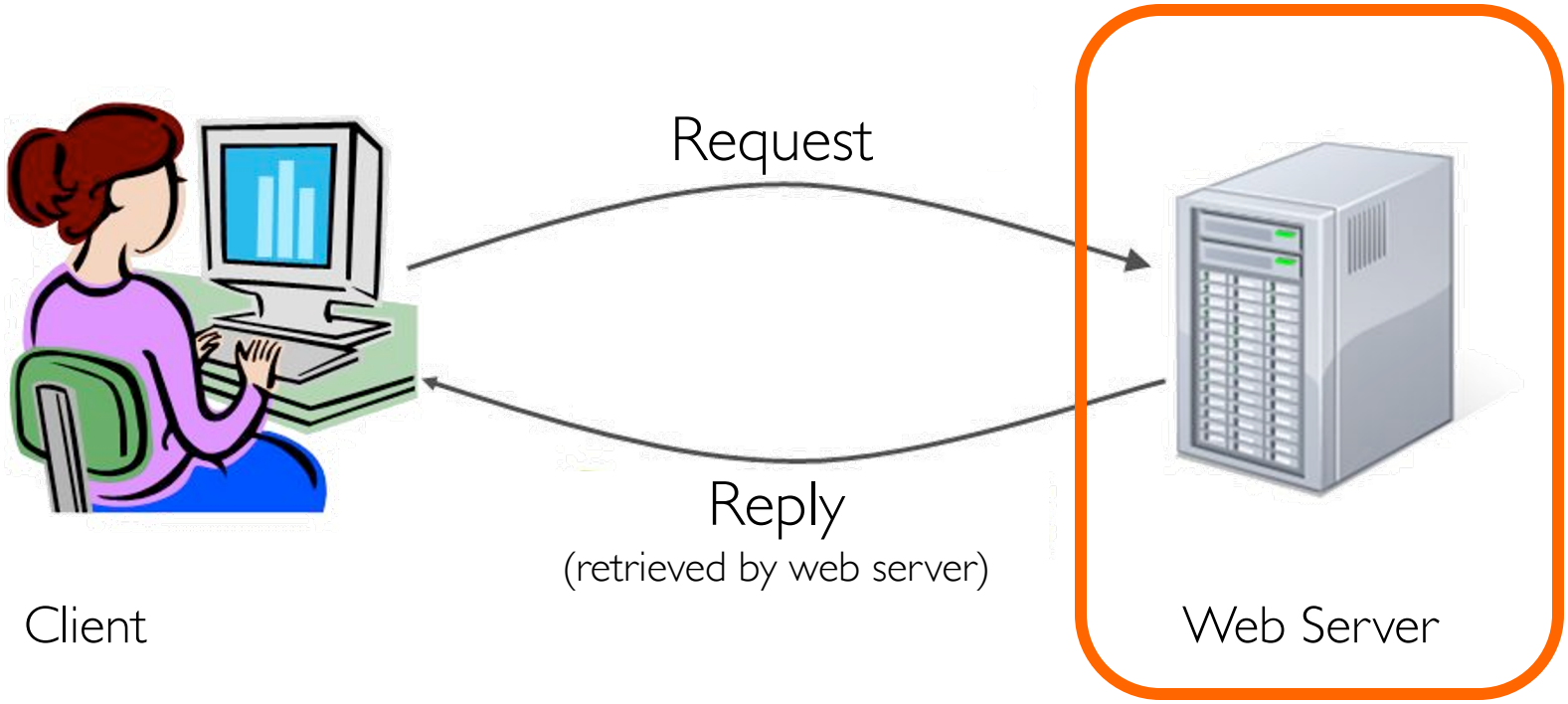
OS Run-time Library



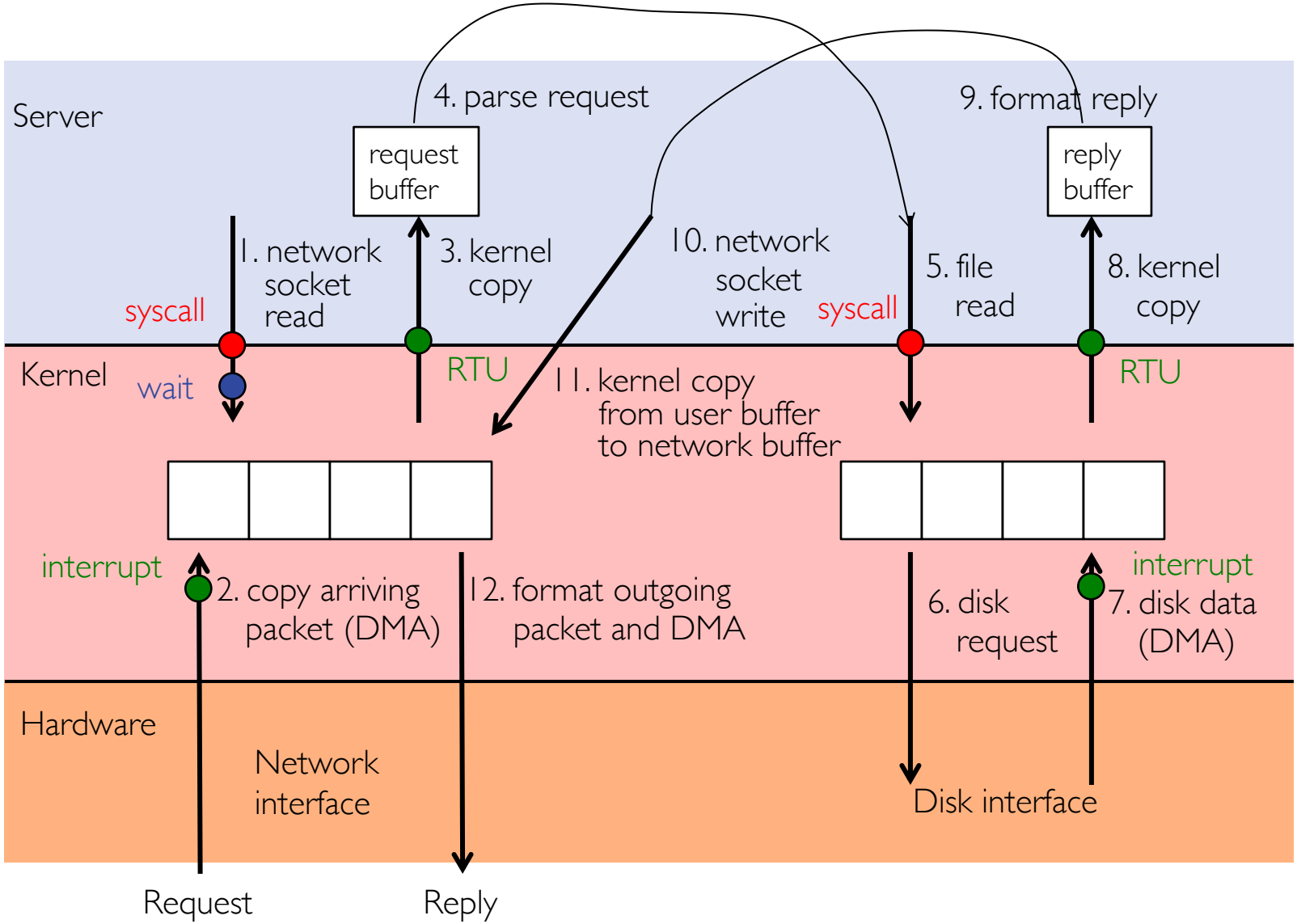




Putting it Together: Web Server



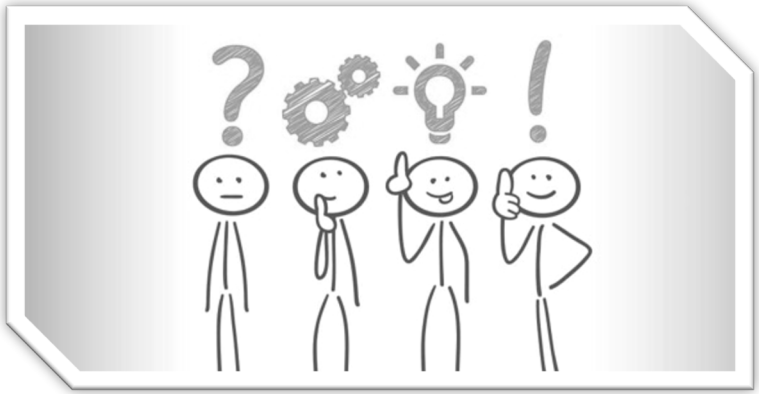
Putting it Together: Web Server (cont.)



Summery

- Thread
  - Single unique execution context which fully describes program state
  - Program counter, registers, execution flags, stack
- Address space (with translation)
  - Address space which is distinct from machine's physical memory addresses
- Process
  - Instance of executing program consisting of address space and 1+ threads
- Dual-mode operation/protection
  - Only "system" can access certain resources
  - OS and hardware are protected from user programs
  - User programs are isolated from one another by controlling translation from program virtual addresses to machine physical addresses

Questions?



Acknowledgment

- Slides by courtesy of Anderson, Culler, Stoica, Silberschatz, Joseph, Zarnett, and Canny