Creating the Process Abstraction

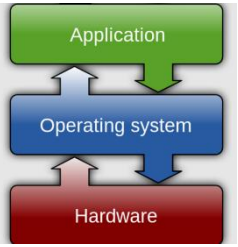
Lecture 6

Hartmut Kaiser

https://teaching.hkaiser.org/spring2026/csc4103/

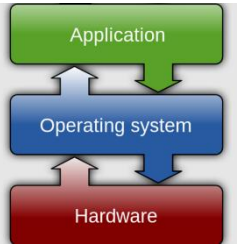
Recall: Process

- Definition: execution environment with restricted rights
  - One or more threads executing in a single address space
  - Owns file descriptors, network connections
- Instance of a running program
  - When you run an executable, it runs in its own process
  - Application: one or more processes working together
- Protected from each other; OS protected from them
- In modern OSes, anything that runs outside of the kernel runs in a process



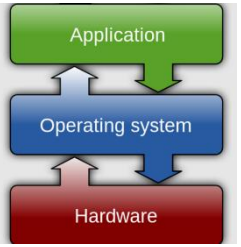
Today: How Does the OS Support the Process Abstraction?

- How does the kernel build the abstractions we have studied?
  - Dual-Mode Operation and Address Spaces?
  - Threads?
  - File I/O?
- What role does hardware play in serving syscalls/interrupts/traps?
- How is the kernel structured?
- And, along the way, getting you ready to tackle Project 1 …



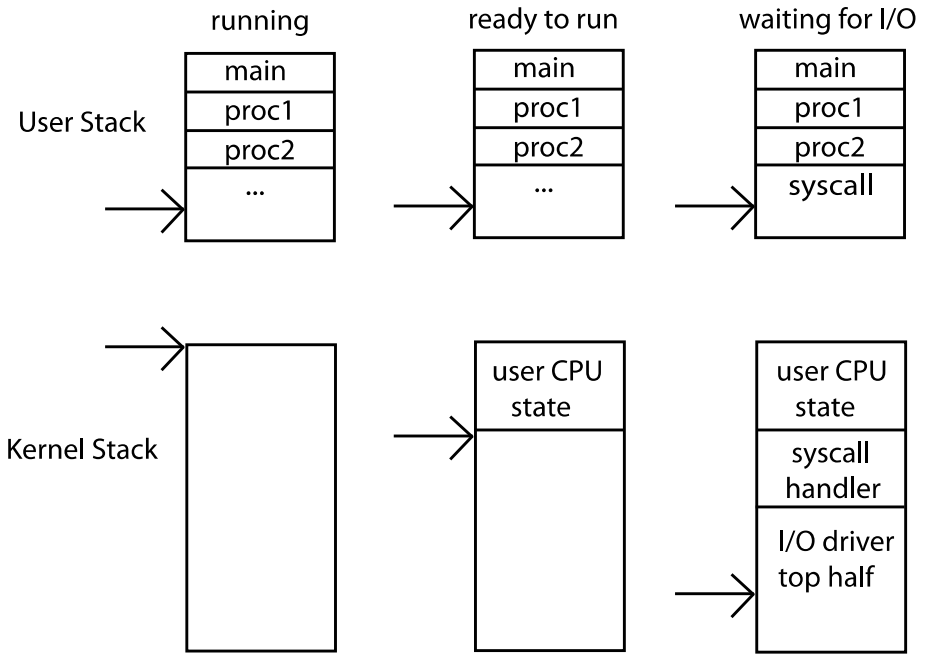
Today: How Does the OS Support the Process Abstraction?

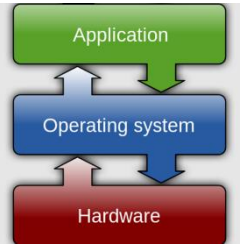
- Support for threads and kernel structure
- Memory layout
- Support for process operations
- Support for I/O
- Influence of IPC/RPC on kernel structure



Recall: Kernel Stacks

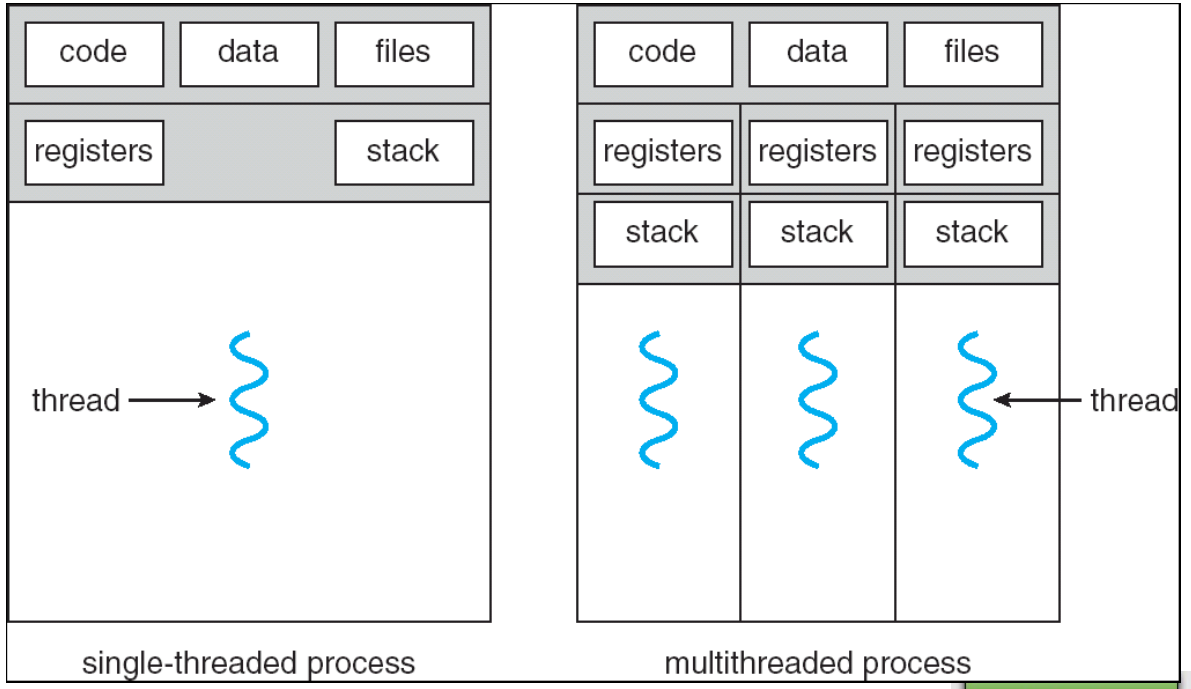
- Interrupt handlers want a stack
- System call handlers want a stack
- Can't just use the user stack [why?]
- One Solution: two-stack model
  - Each thread has user stack and a kernel stack
  - Kernel stack stores user's registers during an exception
  - Kernel stack used to execute exception handler in the kernel

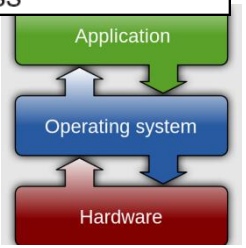




Recall: Single and Multithreaded Processes

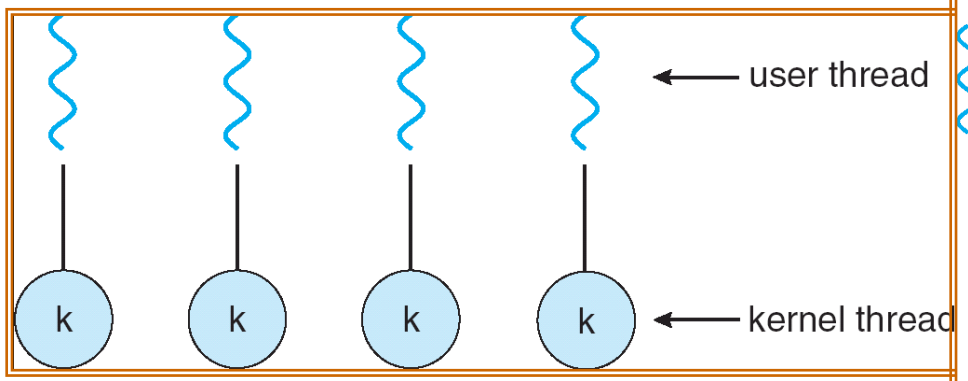
- Threads encapsulate concurrency
  - "Active" component
- Address space encapsulate protection:
  - "Passive" component
  - Keeps bugs from crashing the entire system
- Why have multiple threads per address space?



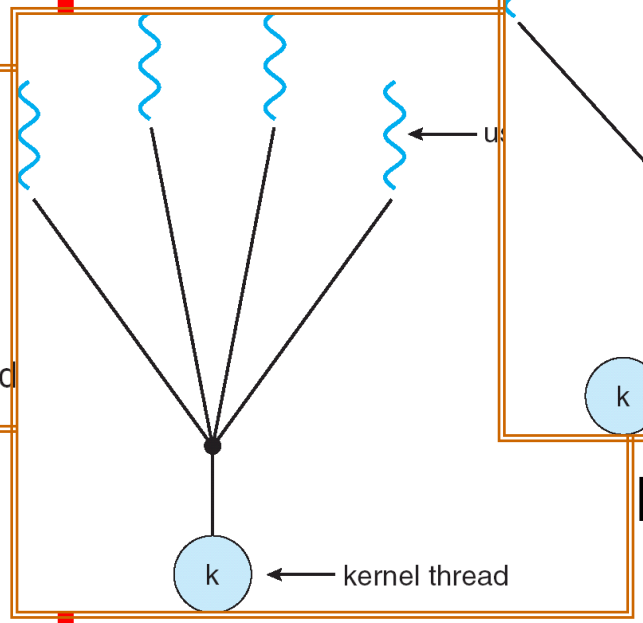


User/Kernel Threading Models

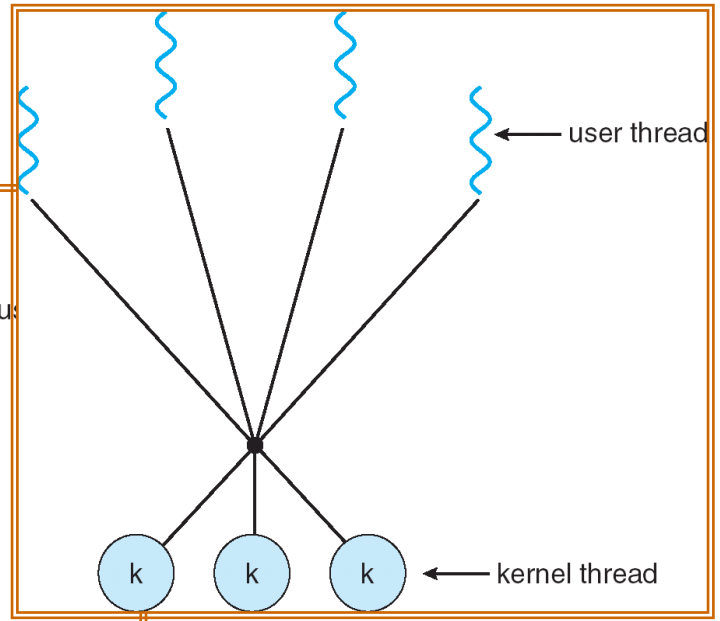
Almost all current implementations



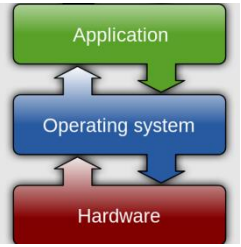
Simple One-to-One Threading Model



Many-to-One

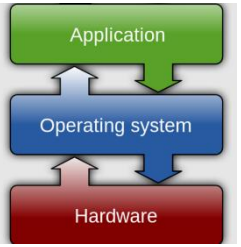


Many-to-Many

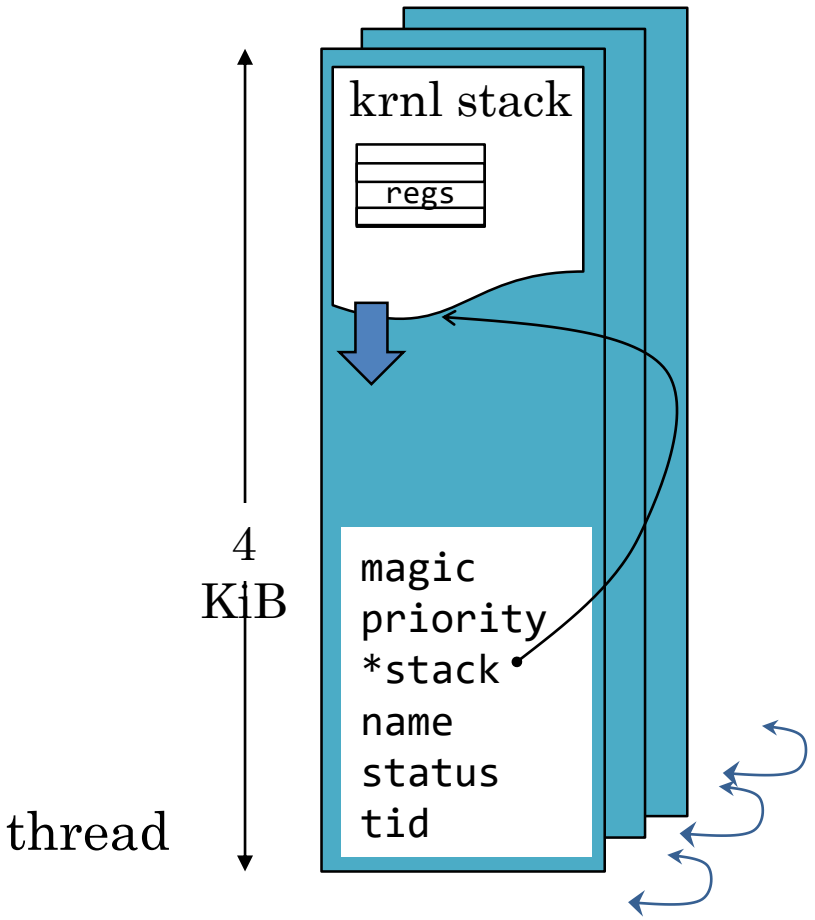


Thread State in the Kernel

- For every thread in a process, the kernel maintains:
  - The thread's "thread control block" (TCB)
  - A kernel stack used for syscalls/interrupts/traps
- Additionally, some threads just do work in the kernel
  - Still has TCB
  - Still has kernel stack
  - But not part of any process, and never executes in user mode

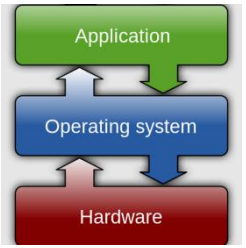


PintOS Thread



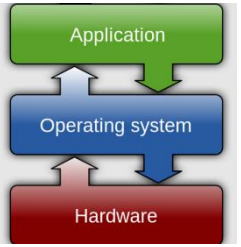
- Single page (4 KiB)
  - Stack growing from the top (high addresses)
  - `struct thread` at the bottom (low addresses)
- **`struct thread`** defines the TCB structure in PintOS
- `thread_current()` retrieves pointer to current thread's TCB

PintOS: thread.c, thread.h

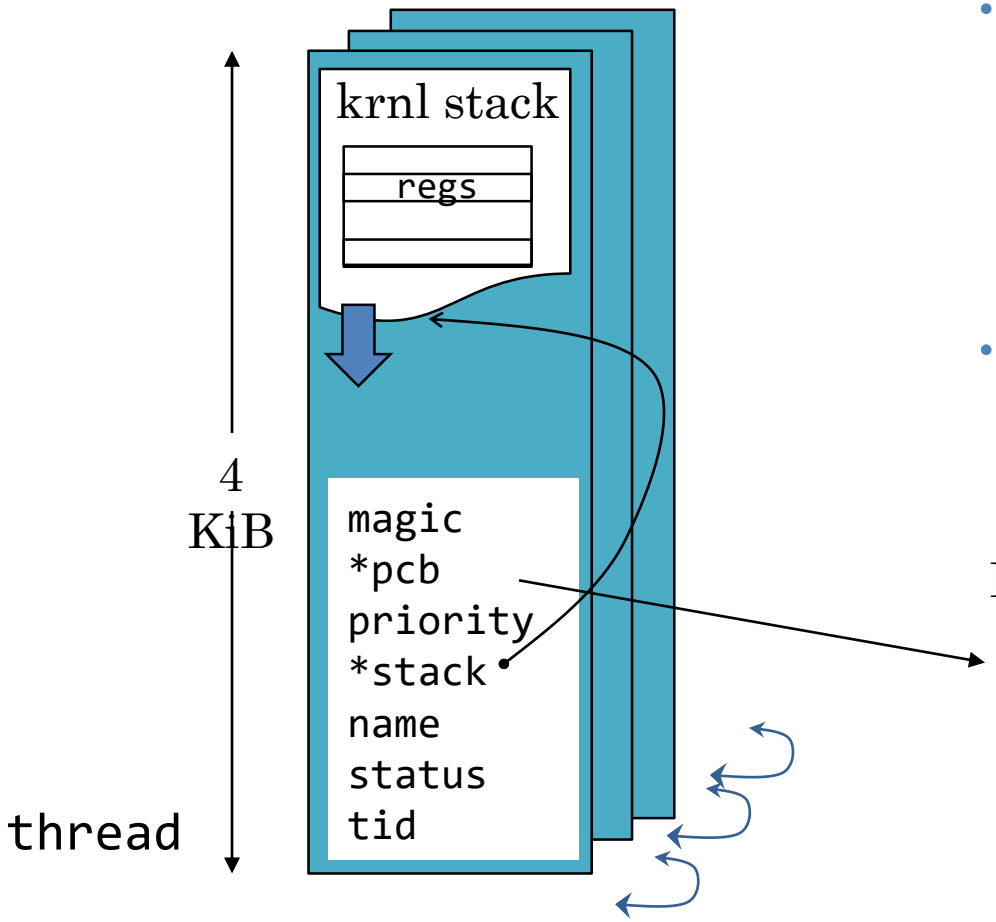


In PintOS, Processes are Single-Threaded (for now)

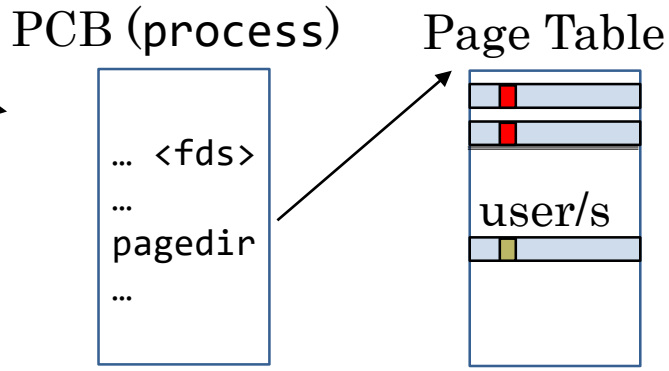
- Processes can contain exactly one thread, for simplicity
- Approach used by older systems
- Project 2 adds thread support



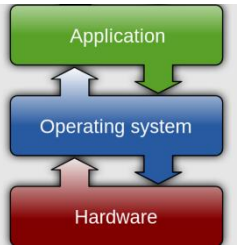
PintOS Thread



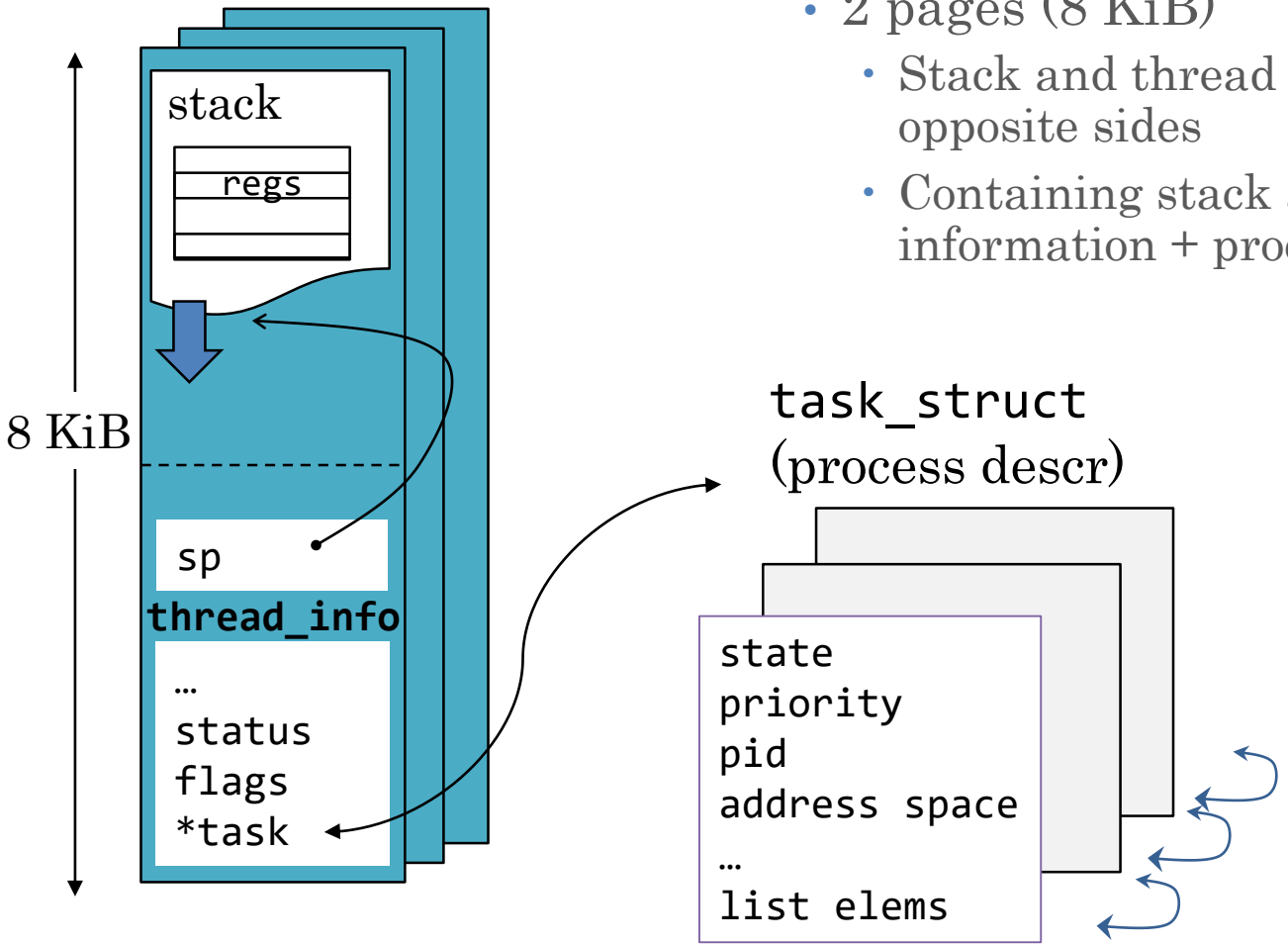
- Single page (4 KiB)
  - Stack growing from the top (high addresses)
  - `struct thread` at the bottom (low addresses)
- **`struct thread`** defines the TCB structure *and should refer to PCB structure* in PintOS



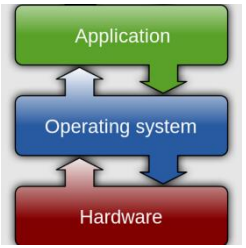
PintOS: thread.c



Linux "Task"

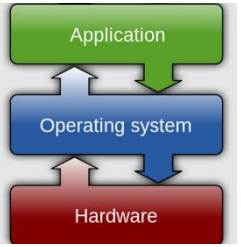


- 2 pages (8 KiB)
  - Stack and thread information on opposite sides
  - Containing stack and thread information + process descriptor



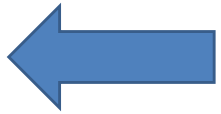
Multithreaded Processes

- Traditional implementation strategy:
  - One PCB (`struct process`) per process
  - Each PCB stores pointer to main thread's TCB
  - Each TCB stores pointer to PCB
- Linux's strategy:
  - One `task_struct` per thread
- Threads belonging to the same process happen to share some resources
  - Like address space, file descriptor table, etc.

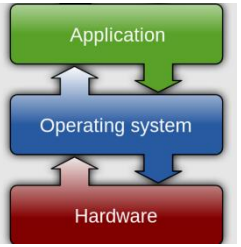


Process Creation (Projects 0 and 1)

- Allocate and initialize Process object
- Allocate and initialize kernel thread mini-stack and associated Thread object
- Allocate and initialize page table for process
  - Referenced by process object
- Load code and static data into user pages
- Build initial User Stack
  - Initial register contents, argv, …
- Schedule (post) process/thread for execution
- …
- Eventually switch to user mode (switching to user stack and registers) …

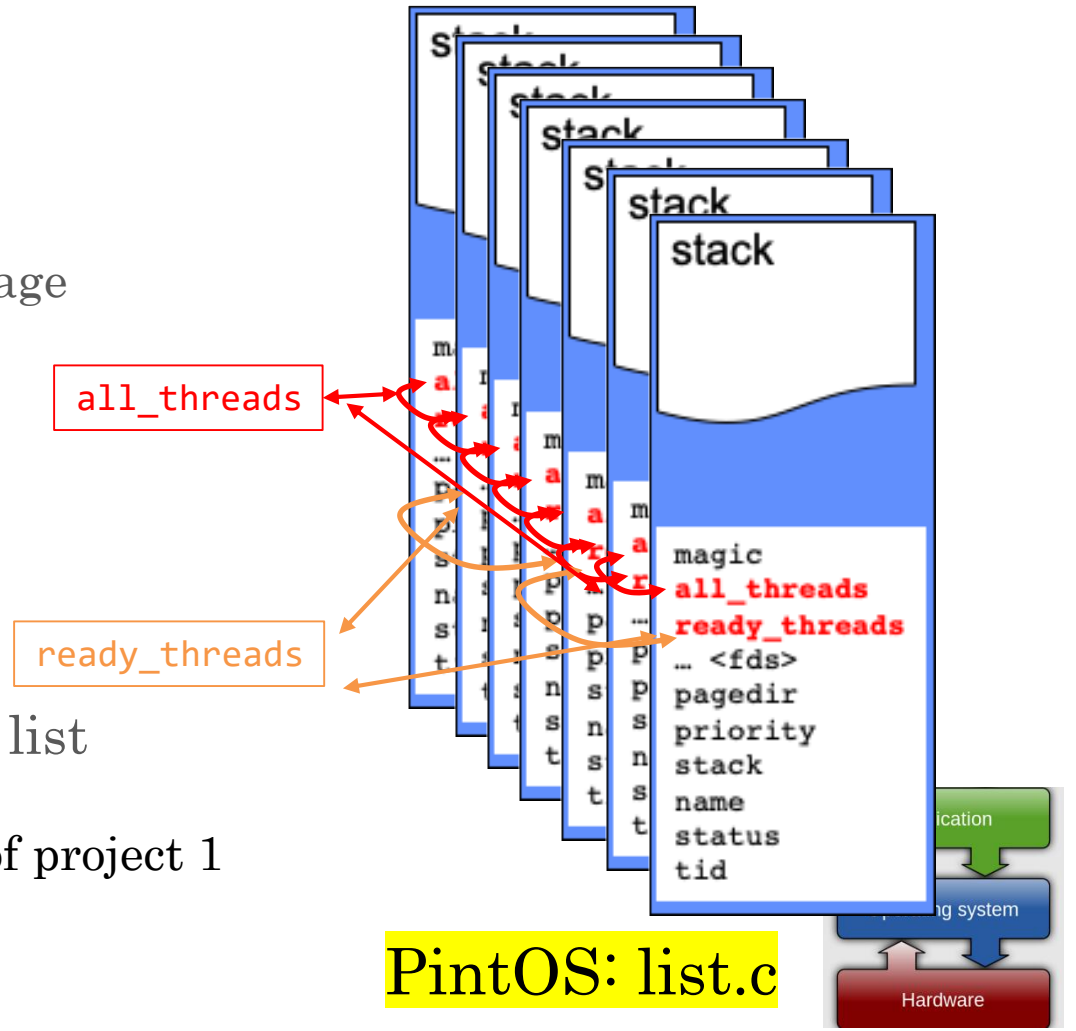


Part of project 1



Aside: Polymorphic Linked Lists in C

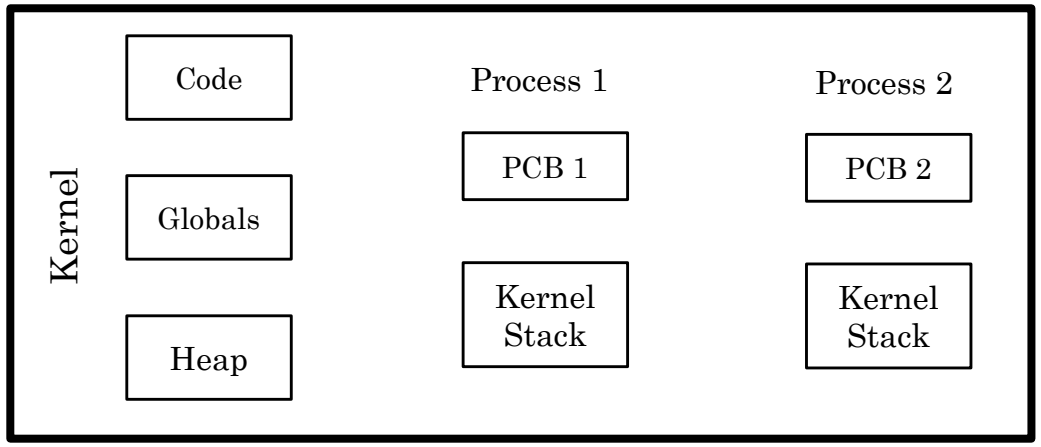
- Many places in the kernel need to maintain a "list of X"
  - This is tricky in C, which has no polymorphism
  - Essentially adding an interface to a package
- In Linux and PintOS this is done by embedding a `list_elem` in the struct
  - Macros allow shift of view between object and list
  - You saw this in Assignment 1
- Needed for building and maintaining a list of child processes

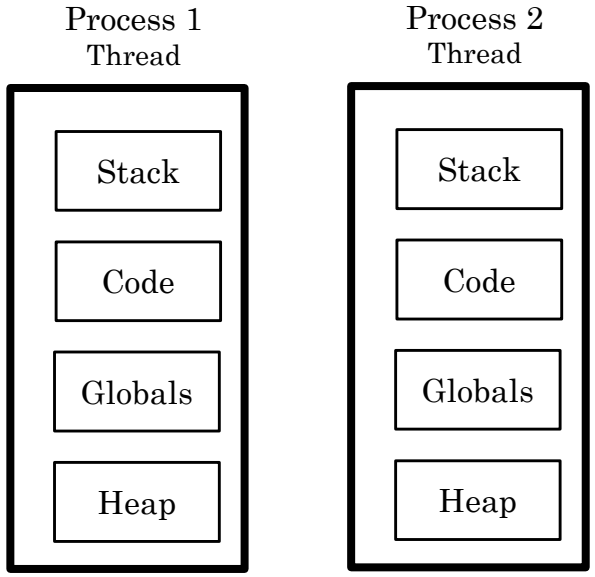


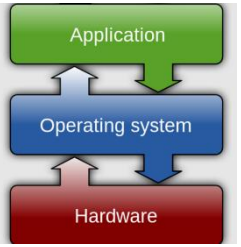
Part of project 1

PintOS: list.c

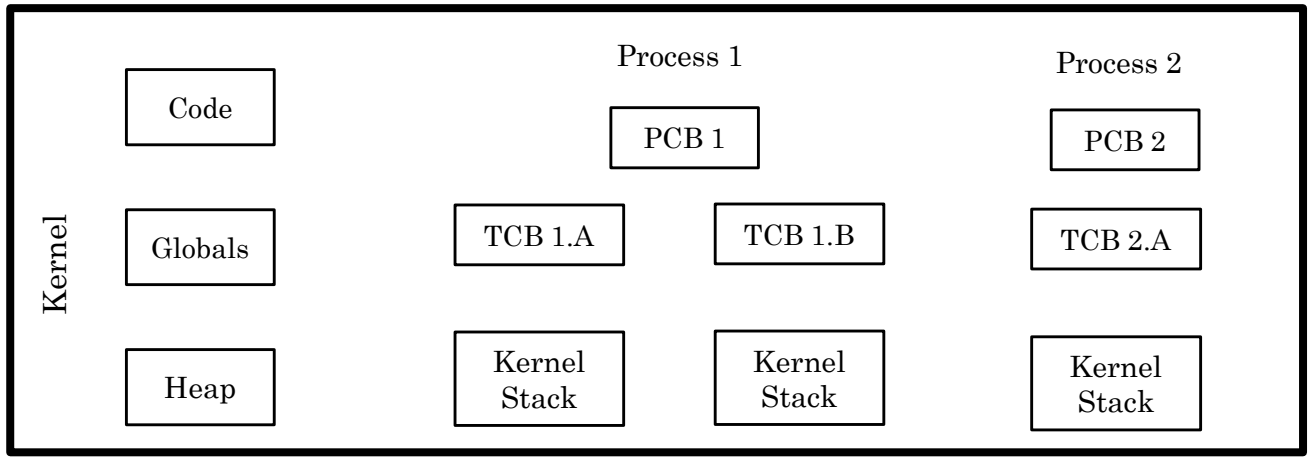
Kernel Structure So Far (1/3)

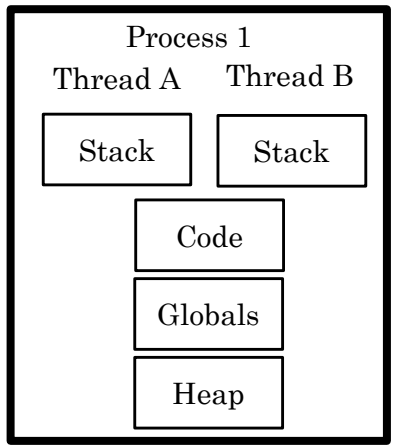


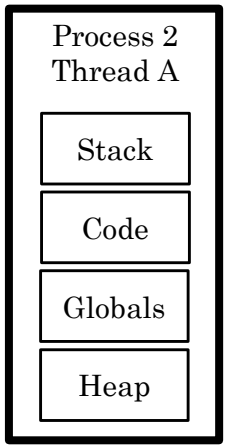


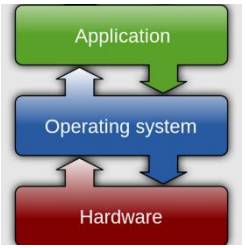


Kernel Structure So Far (2/3)

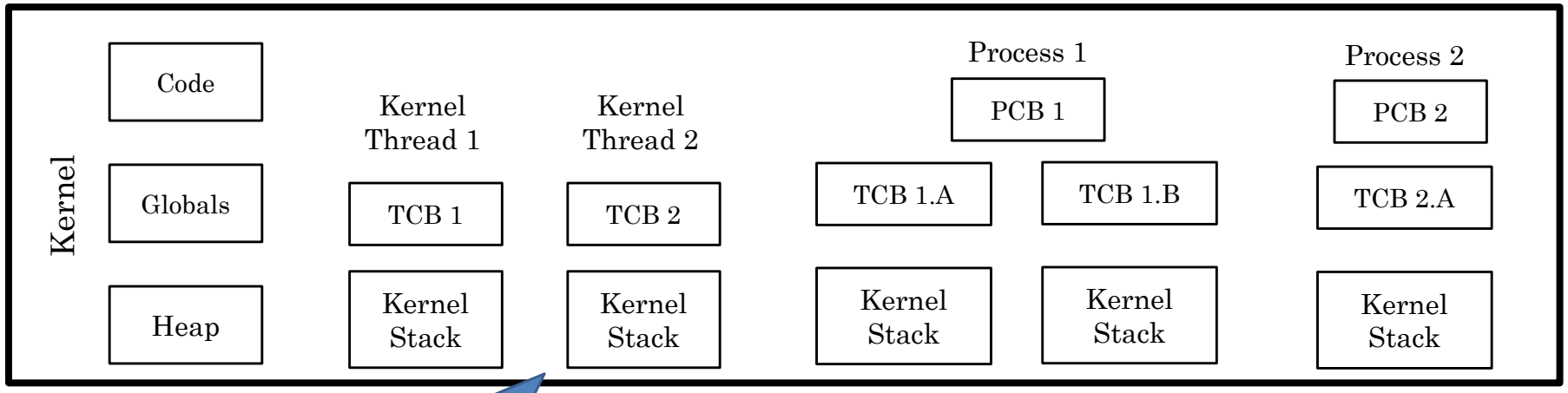




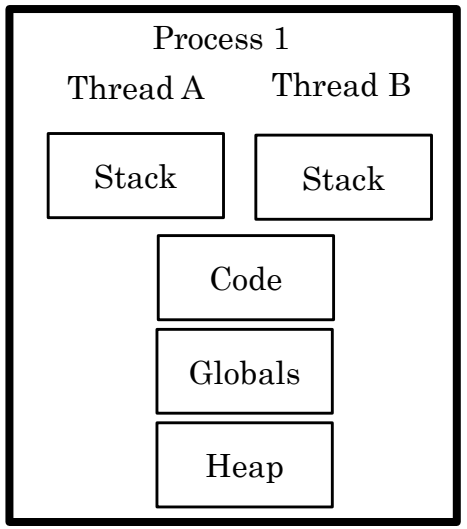


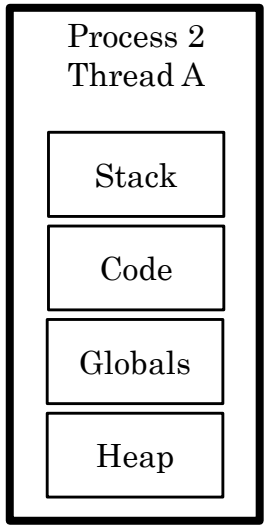


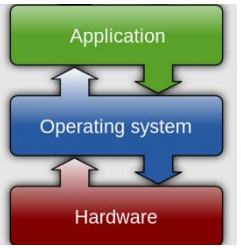
Kernel Structure So Far (3/3)



These threads:
- Are used internally by the kernel
- Don't correspond to any particular user thread or process

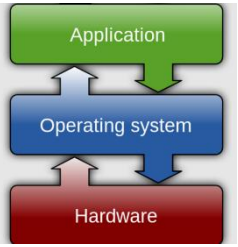






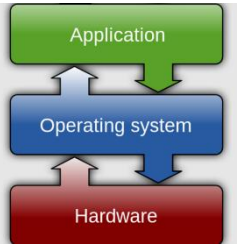
Today: How Does the OS Support the Process Abstraction?

- Support for threads and kernel structure
- Memory layout
- Support for process operations
- Support for I/O
- Influence of IPC/RPC on kernel structure



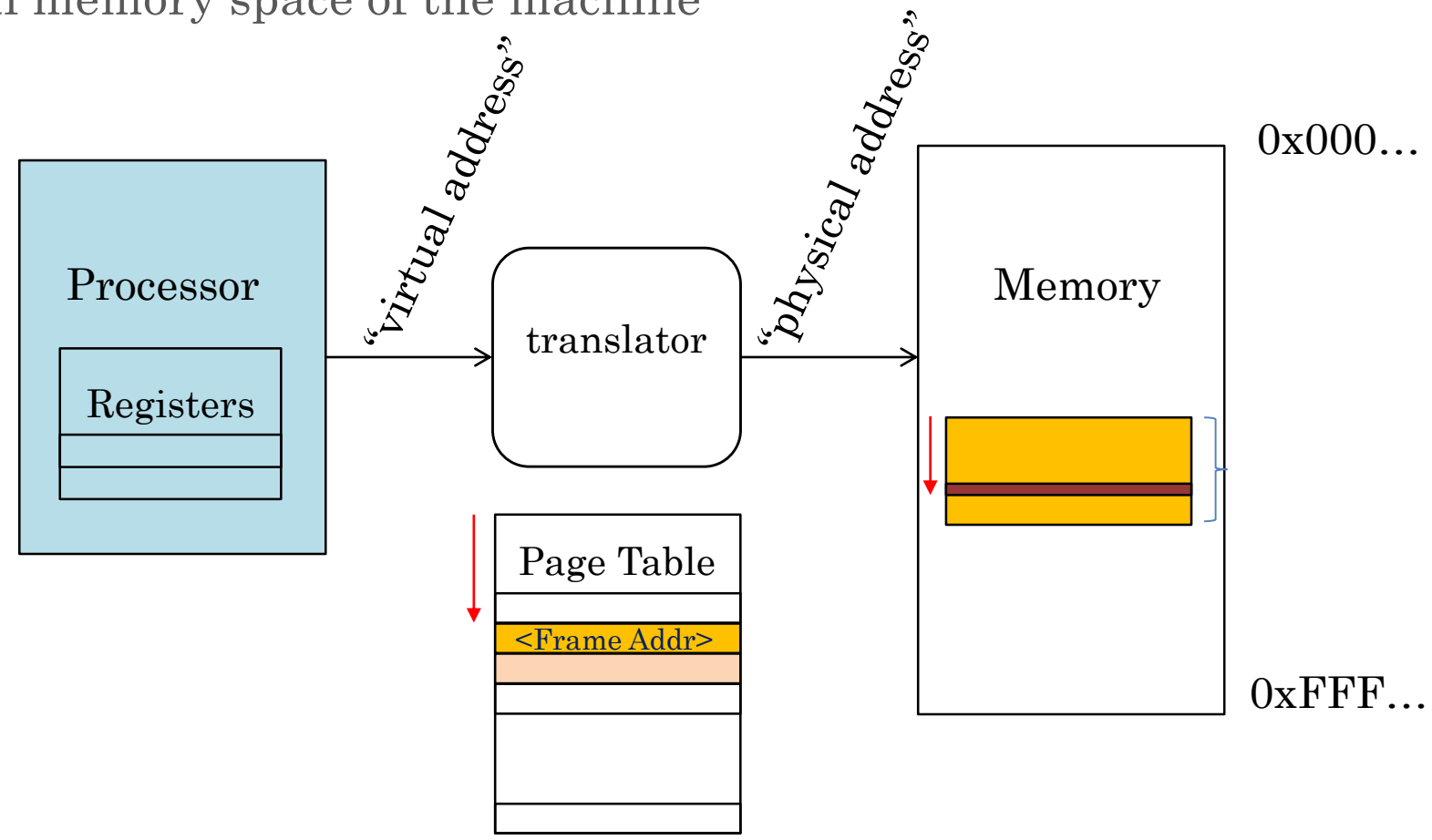
Recall: Process Control Block (PCB)

- Kernel representation of each process
  - Status (running, ready, blocked)
  - Pointer to thread control block (TCB) of main thread
    - Register state (if not running)
  - Process ID
  - Execution time
  - Address space — How is this represented?
  - List of open file descriptions
  - List of pointers to child process PCBs
  - Pointer to parent process PCB
  - Exit code
  - Semaphore to synchronize with parent on wait
  - Etc.



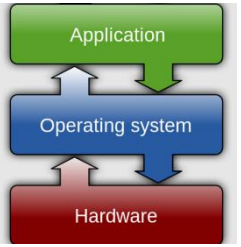
Recall: Address Space

- Program operates in an address space that is distinct from the physical memory space of the machine



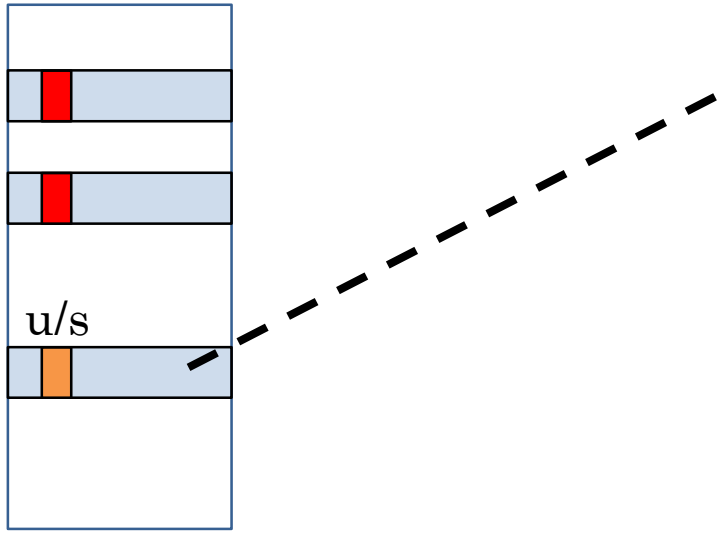
Understanding "Address Space"

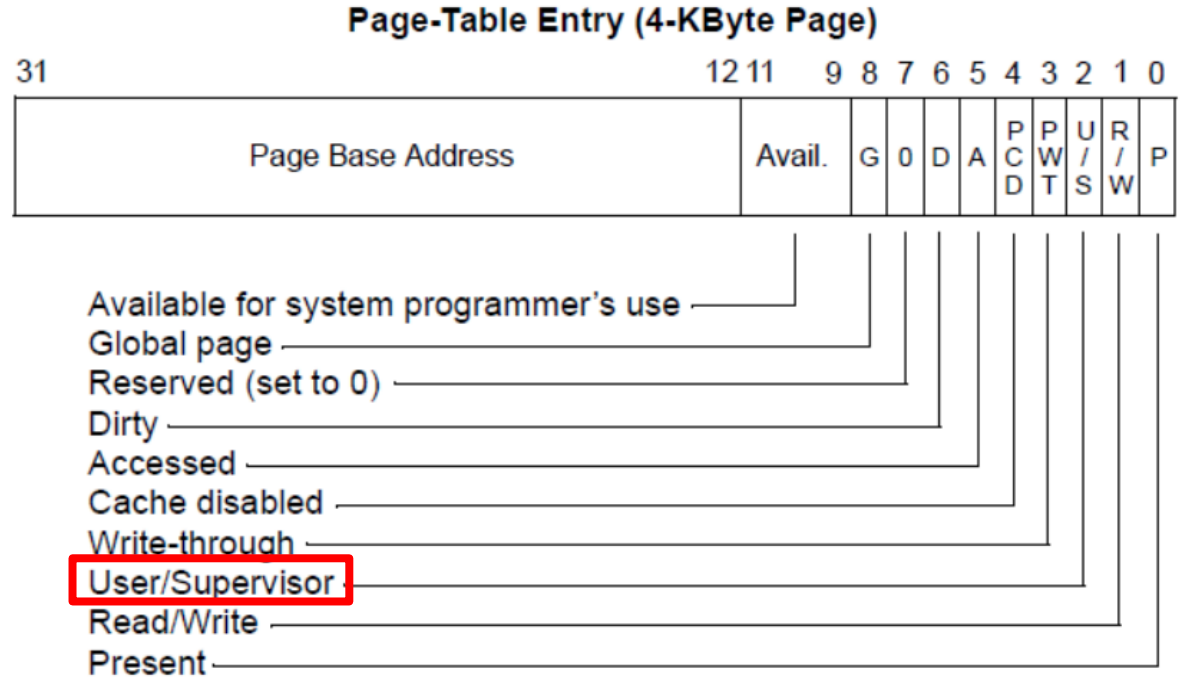
- Page table is the primary mechanism
- Privilege Level determines which regions can be accessed
  - Which entries can be used
- System (PL=0) can access all, User (PL=3) only part
- Each process has its own address space
- The "System" part of all of them is the same
- All system threads share the same system address space and same memory



Aside: x86 (32-bit) Page Table Entry

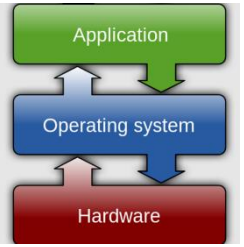
Page Table



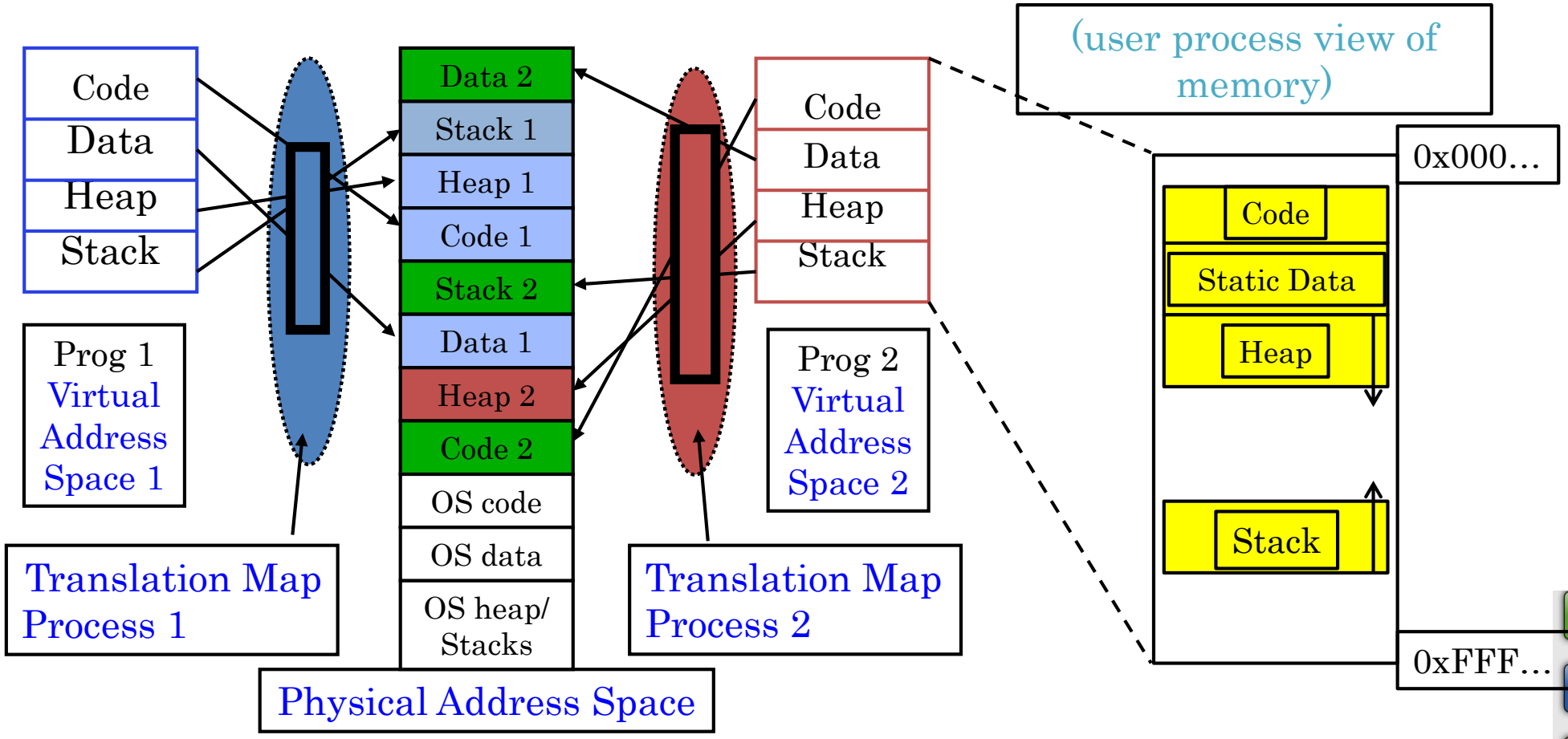


- Controls many aspects of access
- Later – discuss page table organization
  - For 32 (64?) bit VAS, how large? vs size of memory?
  - Used sparsely

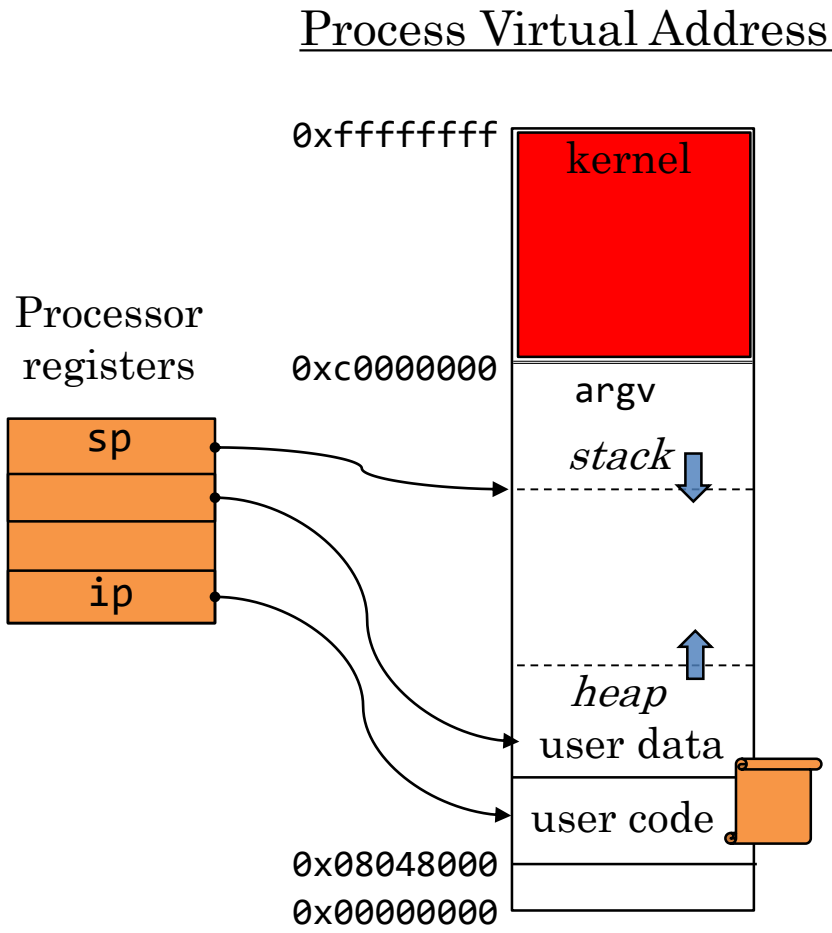
Pintos: page_dir.c

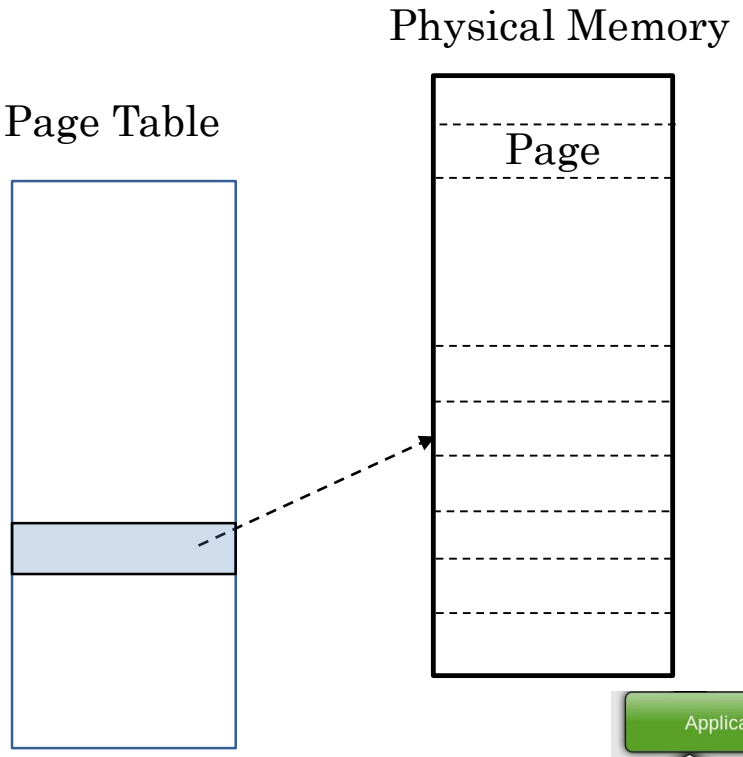


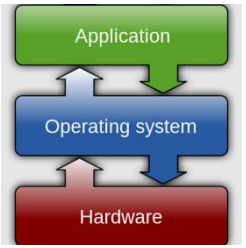
Page Table Mapping (Rough Idea)



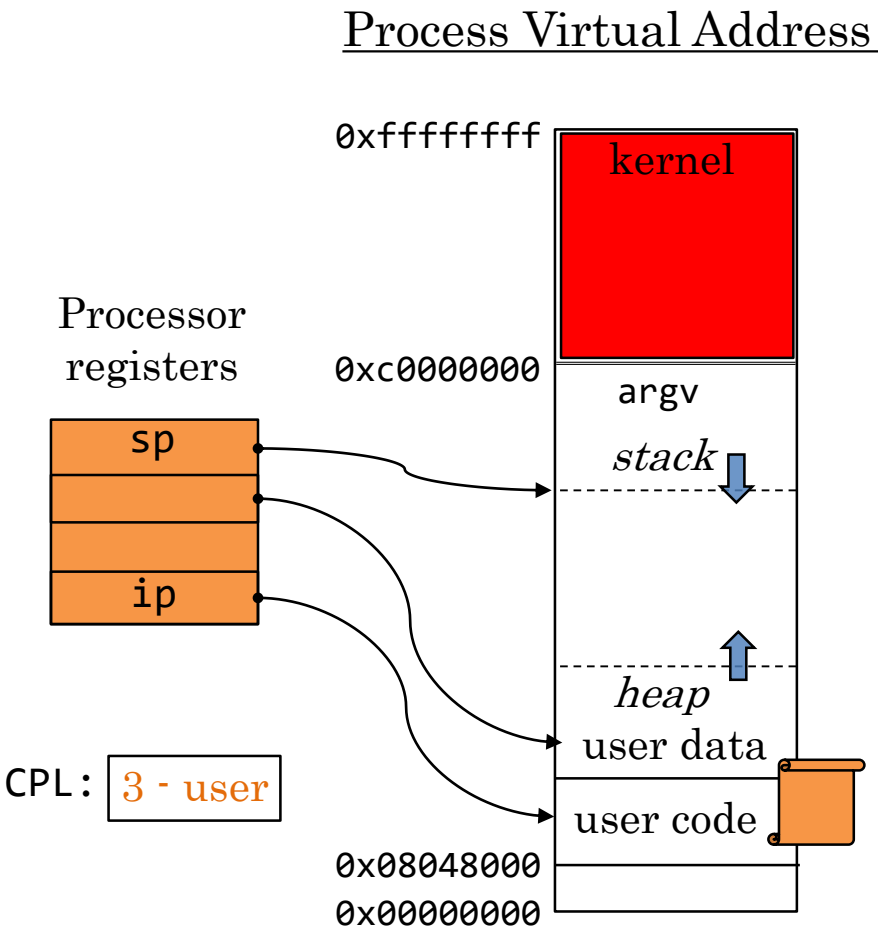
User Process View of Memory

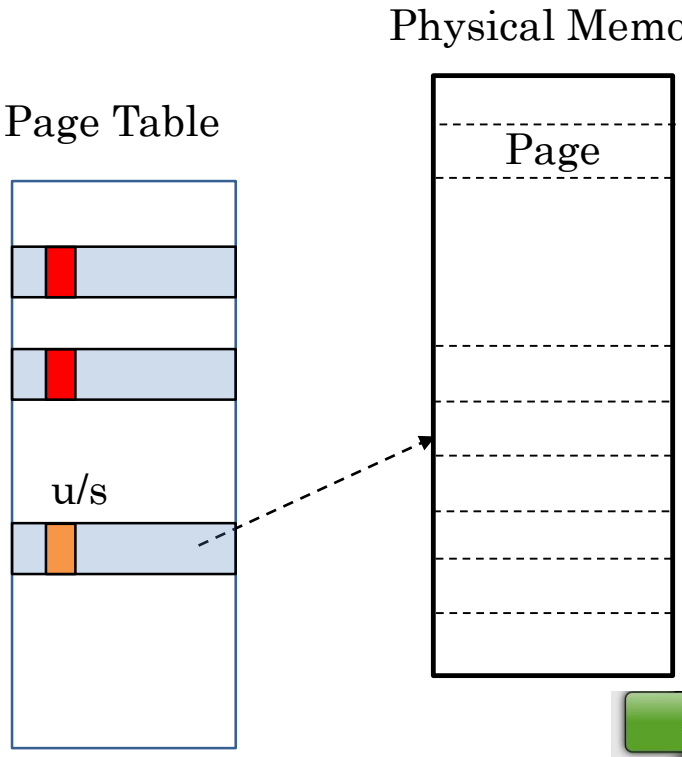


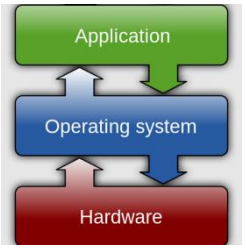




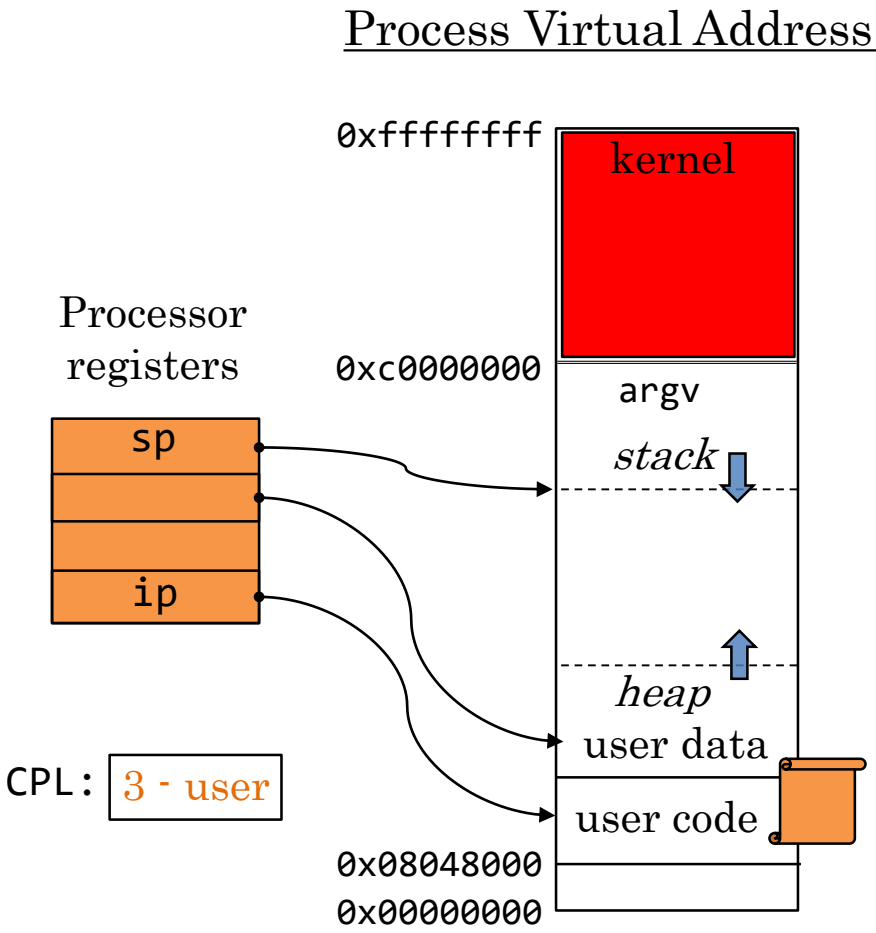
Processor Mode (Privilege Level)

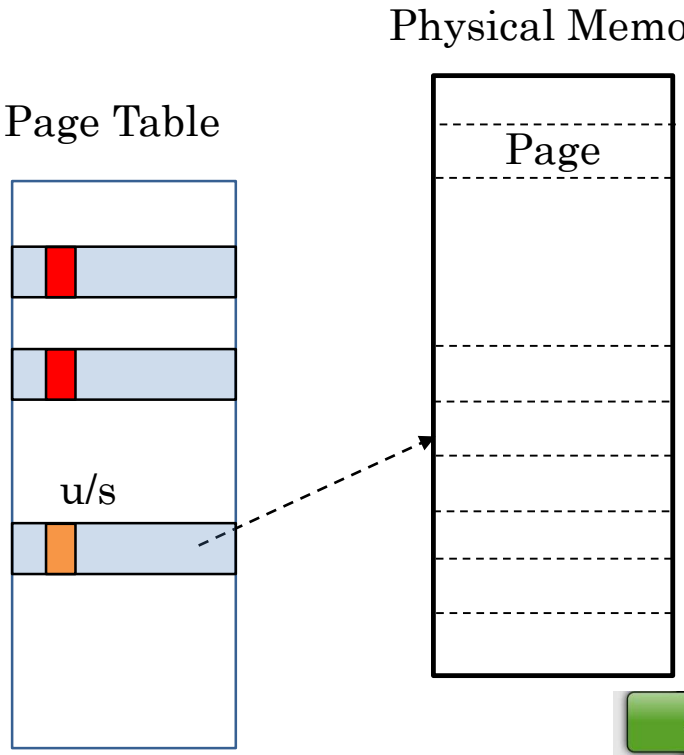


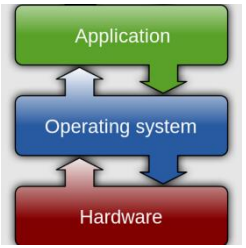




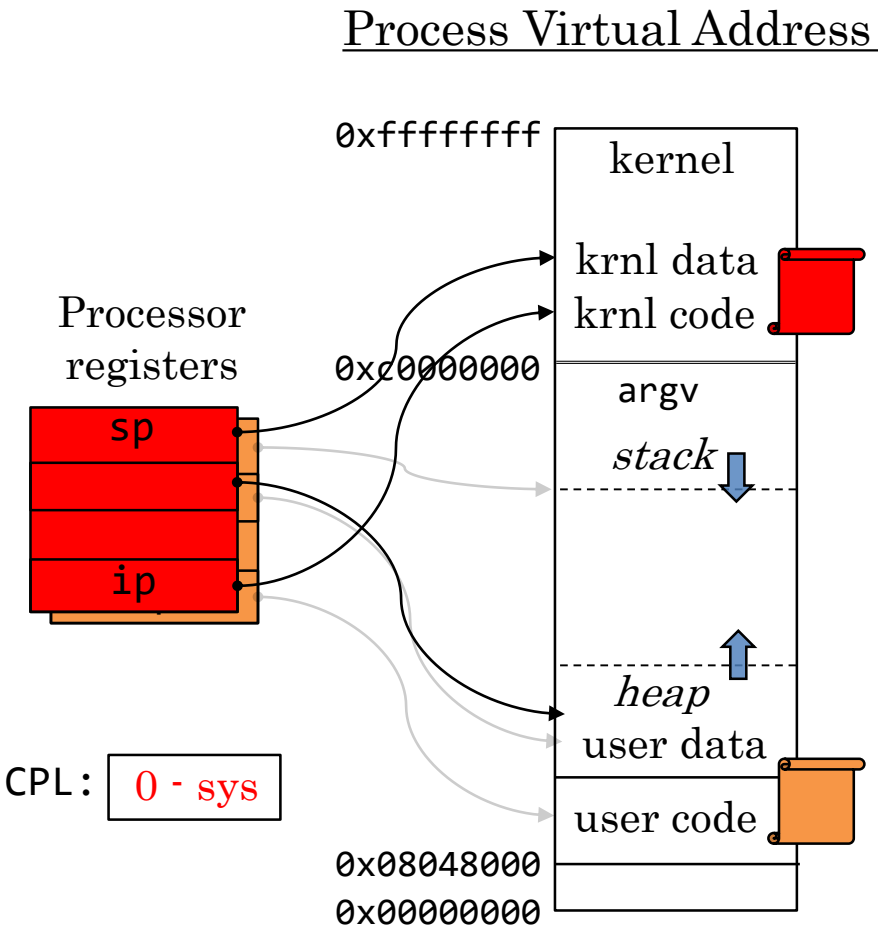
User → Kernel

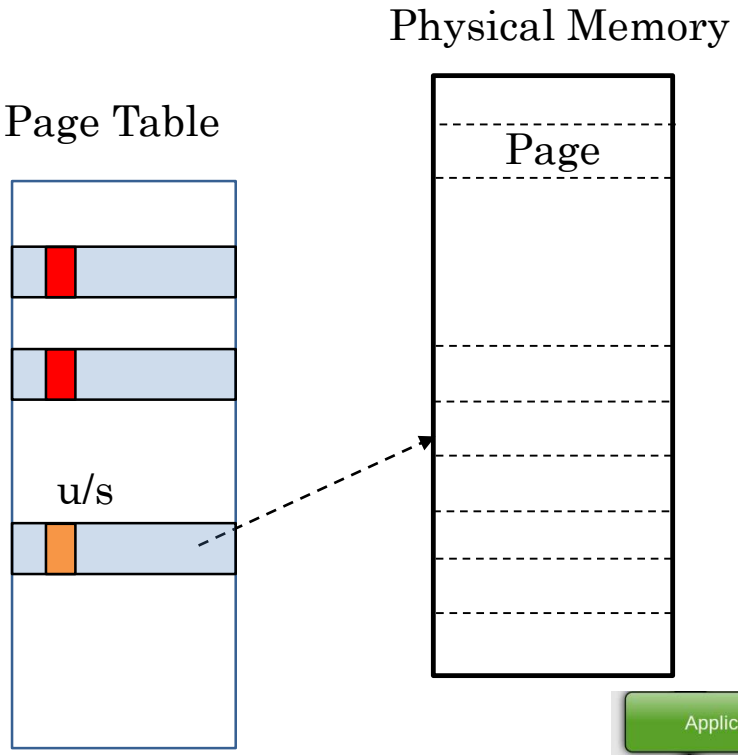

Process Virtual Address Space
0xffffffff
kernel
Processor registers
0xc0000000
argv
sp
stack
ip
heap
user data
CPL: 3 - user
user code
0x08048000
0x00000000

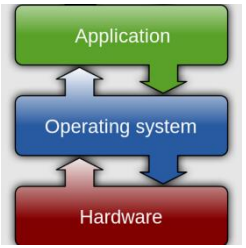

Physical Memory
Page Table
Page
u/s


Application
Operating system
Hardware

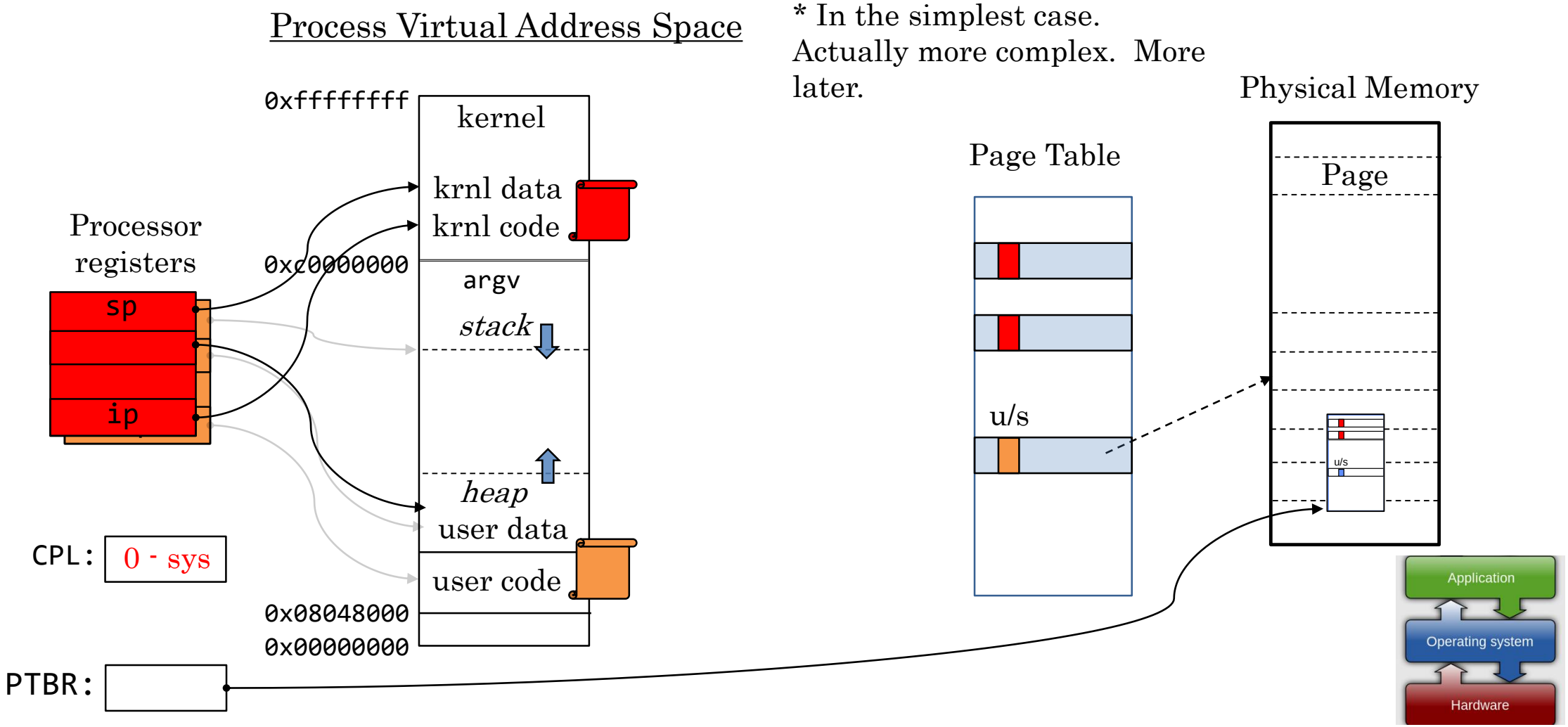
User → Kernel


Process Virtual Address Space
0xffffffff
kernel
krnl data
krnl code
0xc0000000
argv
stack
heap
user data
user code
0x08048000
0x00000000
Processor registers
sp
ip
CPL:
0 - sys


Physical Memory
Page Table
Page
u/s

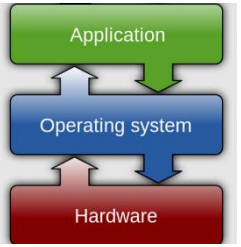

Application
Operating system
Hardware

Page Table Resides in Memory*

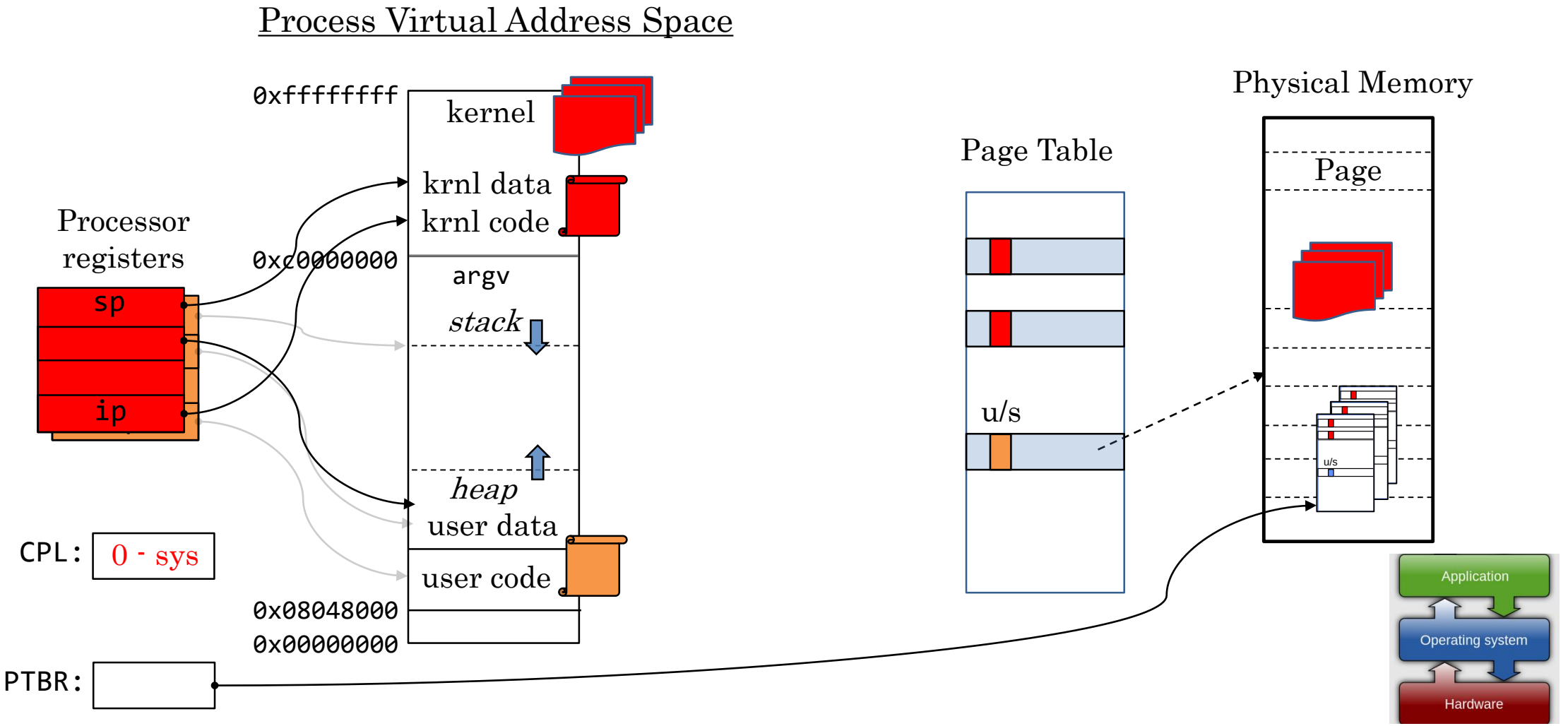


Kernel Portion of Address Space

- Kernel memory is mapped into address space of every process
  - Just only accessible in kernel mode
- Contains the kernel code
  - Loaded when the machine booted
- Explicitly mapped to physical memory
  - OS creates the page table
- Used to contain all kernel data structures
  - Lists of processes/threads
  - Page tables
  - Open file descriptions, sockets, ttys, …
- Kernel stack for each thread

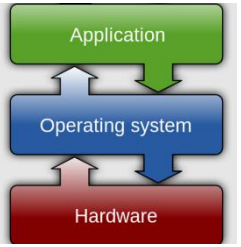


1 Kernel Code, Many Kernel Stacks

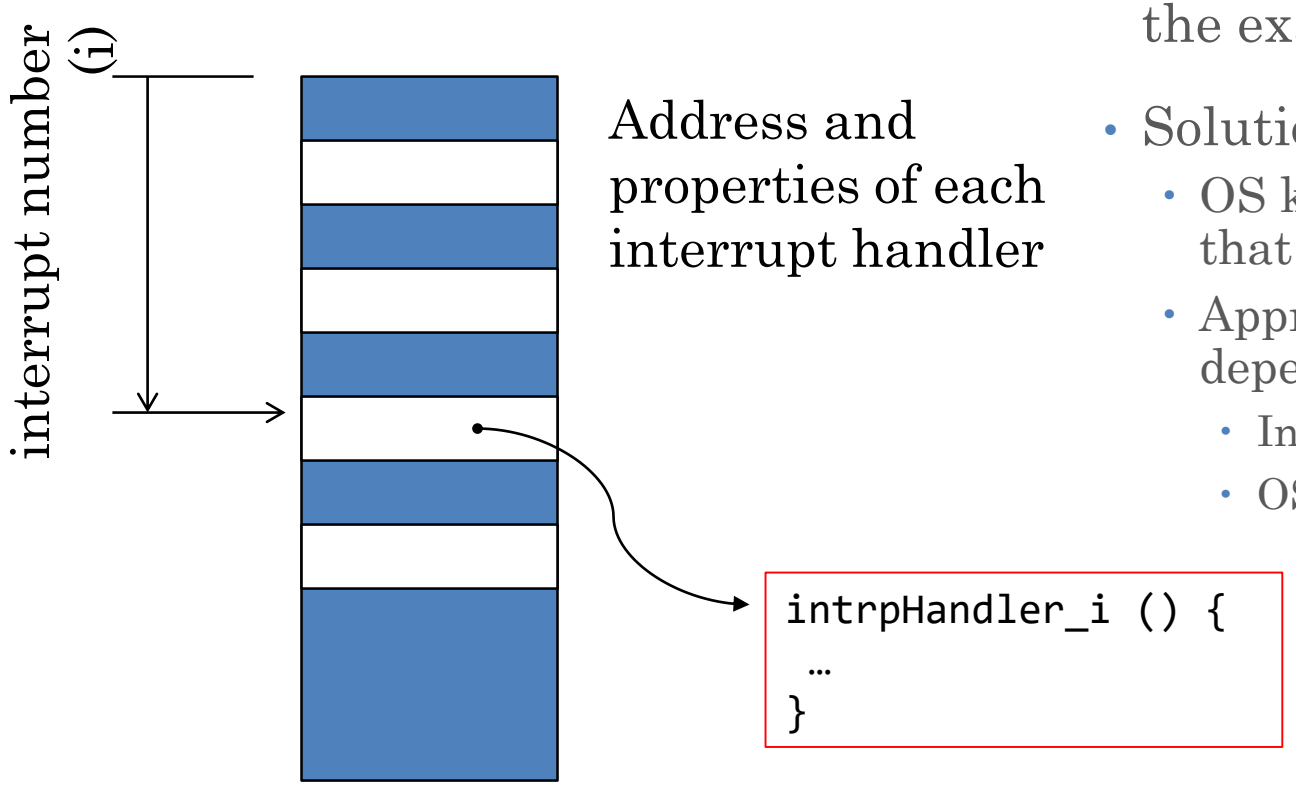


How to Get to the Correct Kernel Stack?

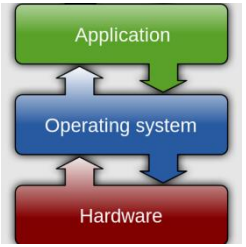
- The hardware helps us out!



Recall: Where do User → Kernel Mode Transfers Go?



- Cannot let user programs specify the exact address!
- Solution: Interrupt Vector
  - OS kernel specifies a set of functions that are entry points to kernel mode
  - Appropriate function is chosen depending on the type of transition
    - Interrupt Number (i)
    - OS may do additional dispatch



Hardware Support for Switching Stacks

- Syscall/Intr (U → K)
  - PL 3 → 0;
  - TSS ← EFLAGS, CS:EIP;
  - SS:ESP ← k-thread stack (TSS PL 0);
  - push (old) SS:ESP onto (new) k-stack
  - push (old) EFLAGS, CS:EIP, <err>
  - CS:EIP ← <k target handler>
- Then
  - Handler then saves other regs, etc.
  - Does all its work, possibly choosing other threads, changing PTBR (CR3)
  - Kernel thread has set up user GPRs
- iret (K → U)
  - PL 0 → 3;
  - Eflags, CS:EIP ← popped off k-stack
  - SS:ESP ← user thread stack (TSS PL 3);

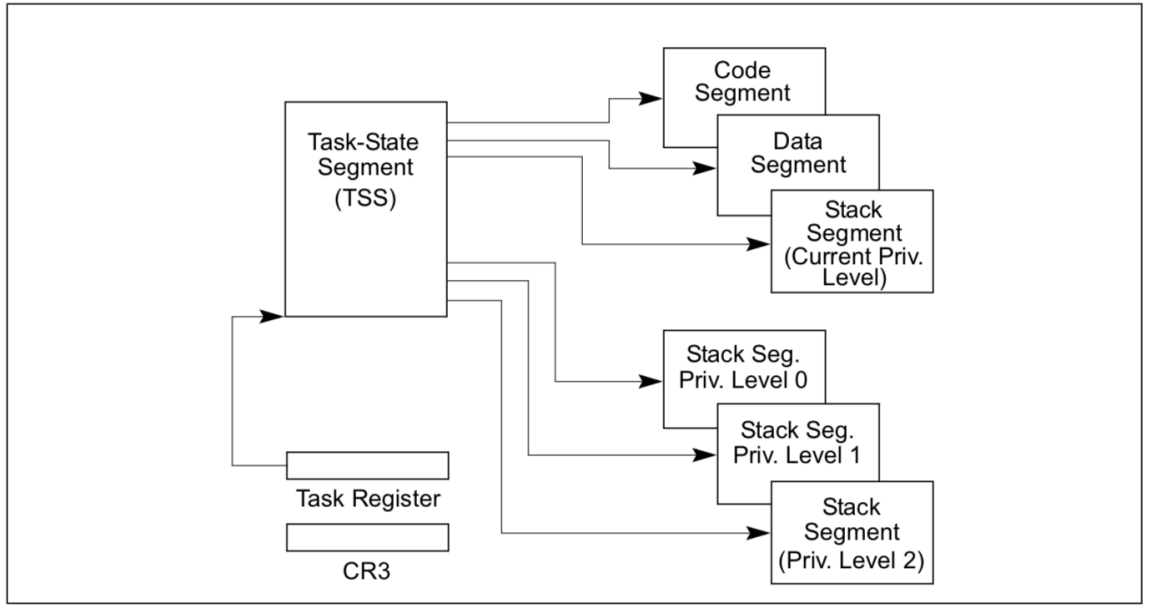
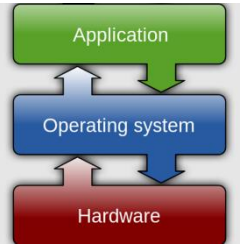


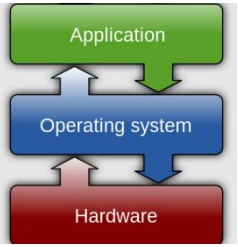
Figure 7-1. Structure of a Task

PintOS: tss.c, intr-stubs.S



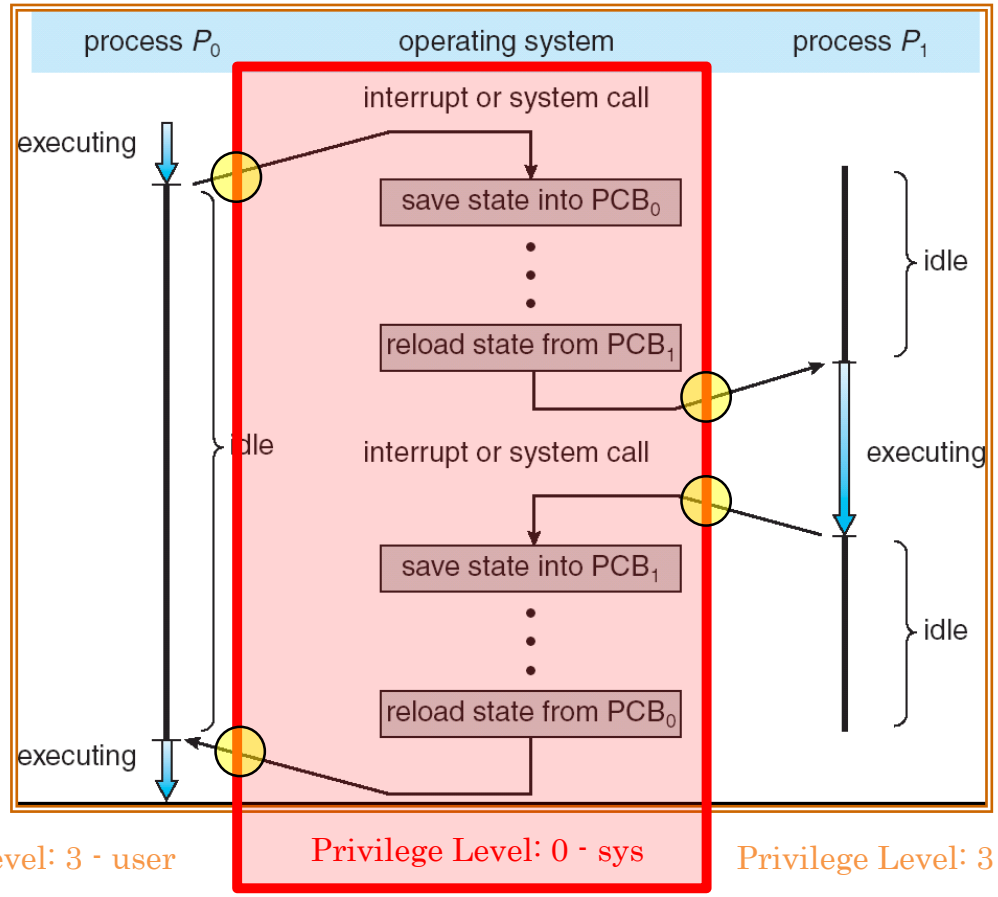
Recall: The Process

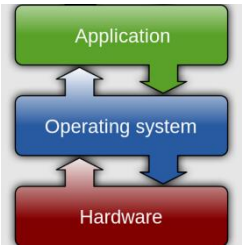
- Definition: execution environment with restricted rights
  - Address Space with One or More Threads
    - Page table per process!
  - Owns memory (mapped pages)
  - Owns file descriptors, file system context, …
  - Encapsulates one or more threads sharing process resources
- Application program executes as a process
  - Complex applications can fork/exec child processes [later]
- Why processes?
  - Protected from each other. OS Protected from them.
  - Execute concurrently [trade-offs with threads? later]
  - Basic unit OS deals with



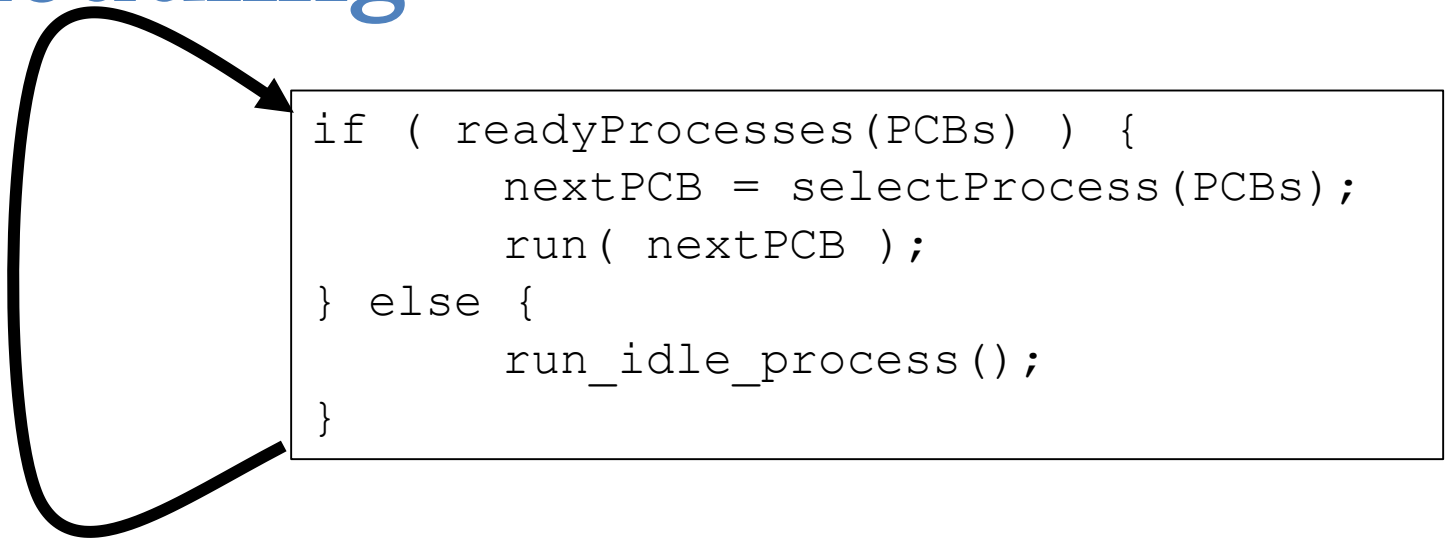
Context Switch

- Diagram assumes single-threaded processes
- For multi-threaded process, substitute process → thread and PCB → TCB



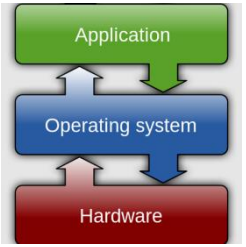


Recall: Scheduling



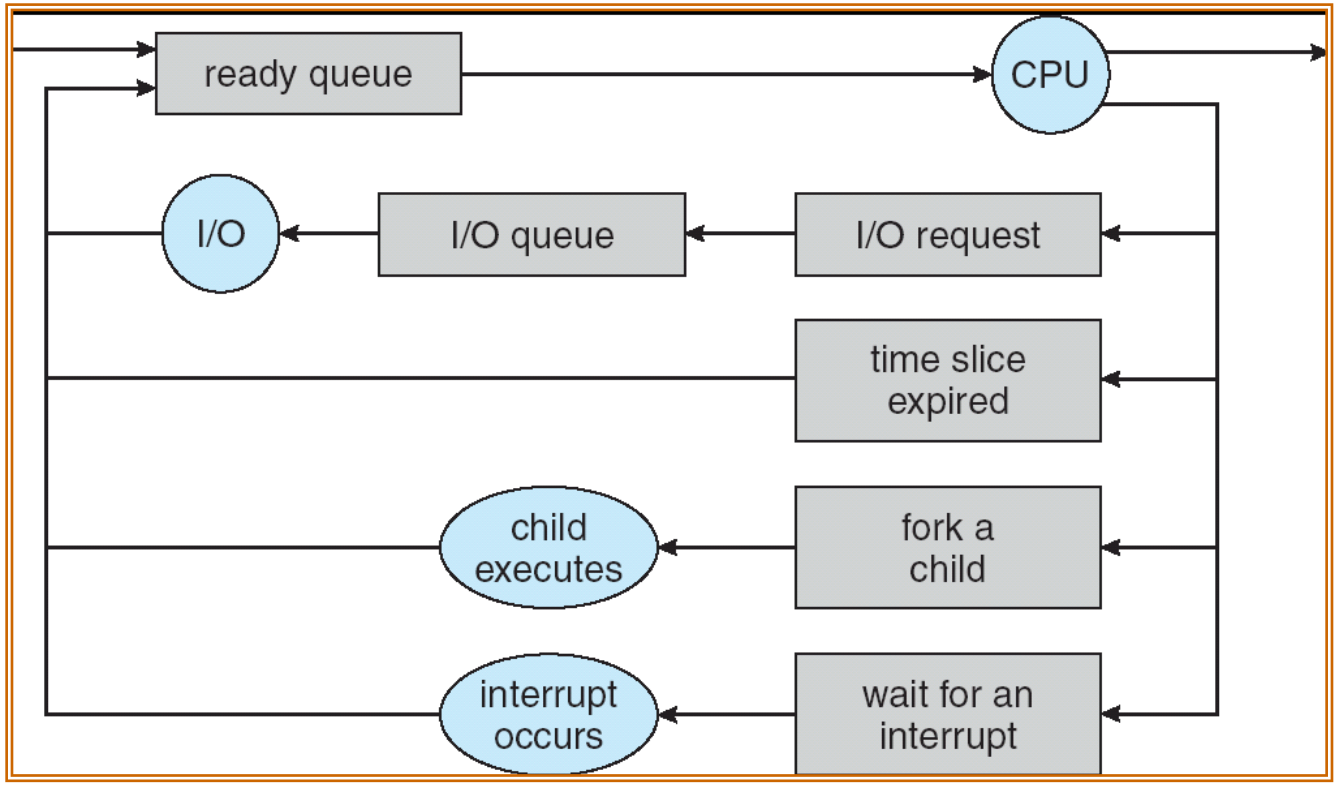
```
if ( readyProcesses(PCBs) ) {
        nextPCB = selectProcess(PCBs);
        run( nextPCB );
} else {
        run_idle_process();
}
```

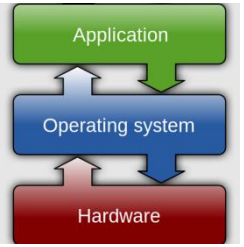
- Scheduling: Mechanism for deciding which processes/threads receive the CPU
- Lots of different scheduling policies provide …
  - Fairness or
  - Realtime guarantees or
  - Latency optimization or …



Scheduling: All About Queues

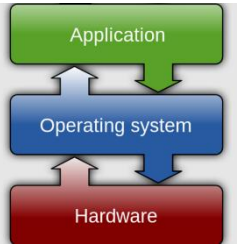
- TCBs move from queue to queue
- Scheduling: which order to remove from queue of "ready" threads





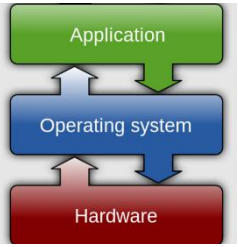
Announcements

- Assignment 1 due tonight
  - Please finish submitting in time
- Project 1 was posted, due March 24 (design document due March 10)
  - Walkthrough for project 1 will be March 17
  - Don't postpone work for this
- Assignment 2 will be available later this week
- Mardi-Gras break: March 3
- Midterm review: March 10, midterm exam: March 12



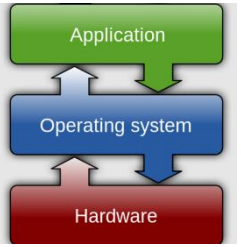
Today: How Does the OS Support the Process Abstraction?

- Support for threads and kernel structure
- Memory layout
- Support for process operations
- Support for I/O
- Influence of IPC/RPC on kernel structure

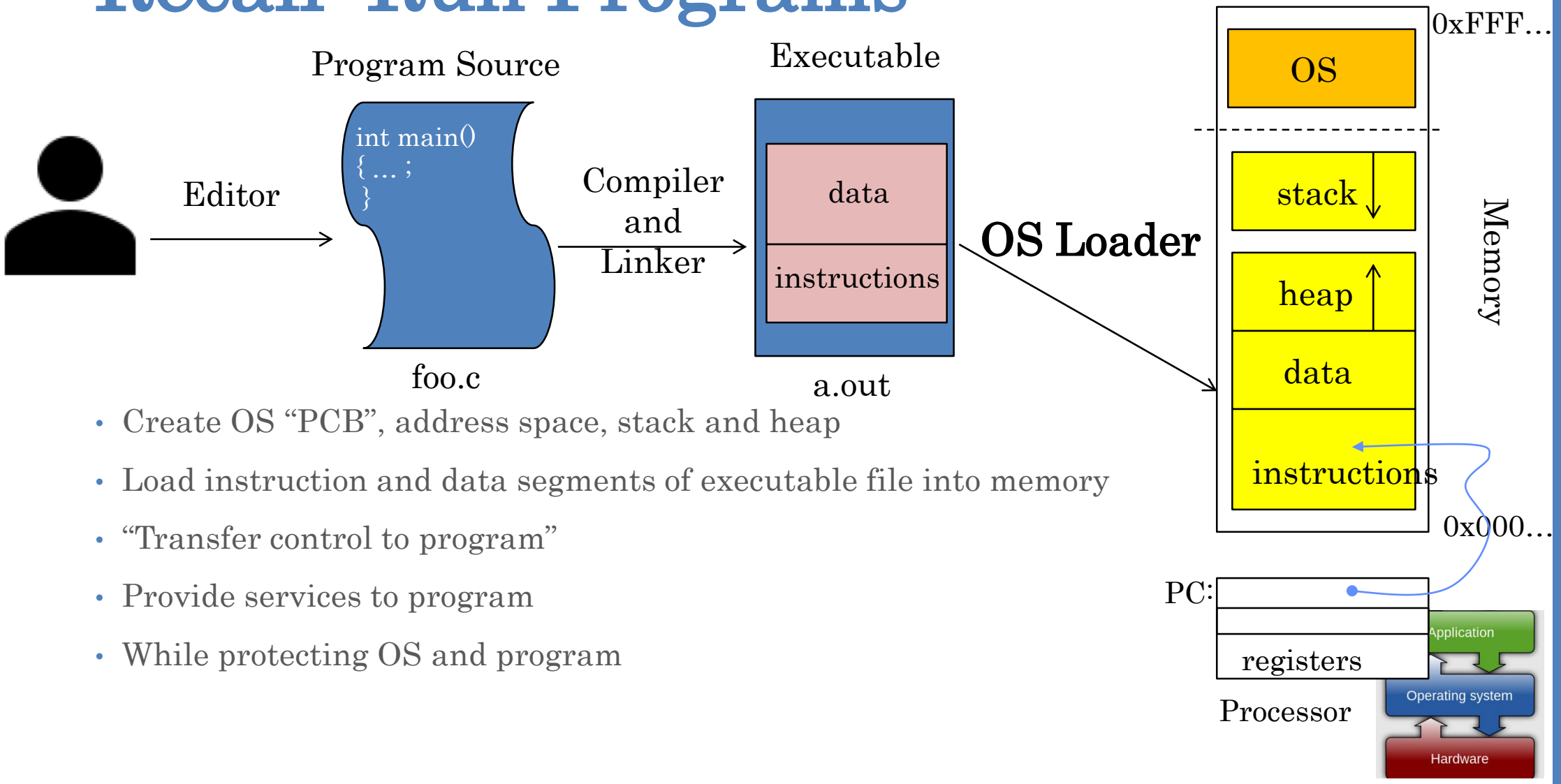


Operations on Process State

- Process related
  - `fork()/exec()` (PintOS: `process_create()`)
  - `wait()`
- File-system related
  - `open()/close()`

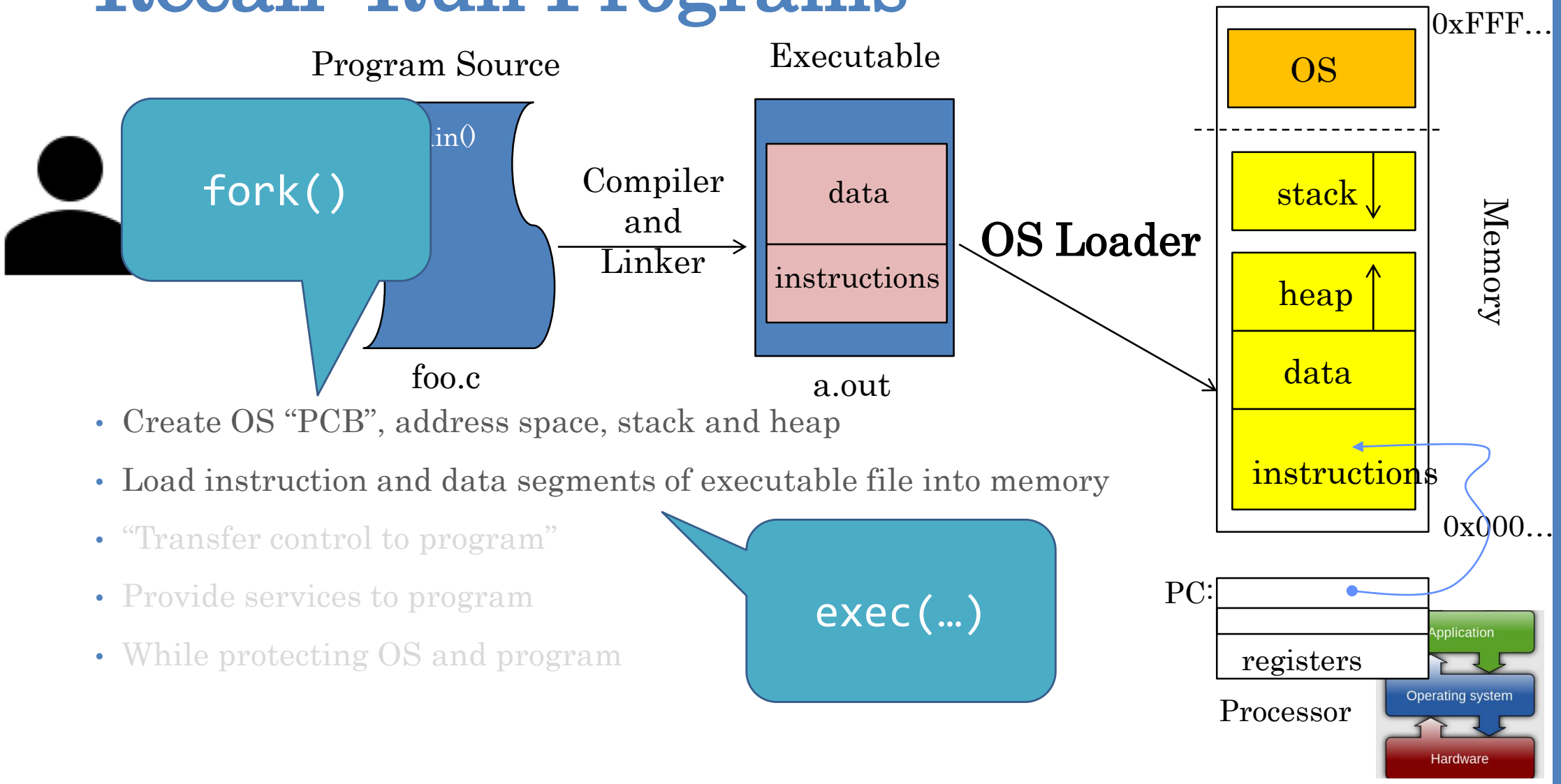


Recall: Run Programs



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

Recall: Run Programs

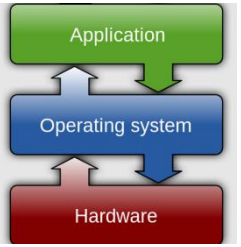


- Create OS "PCB", address space, stack and heap
- Load instruction and data segments of executable file into memory
- "Transfer control to program"
- Provide services to program
- While protecting OS and program

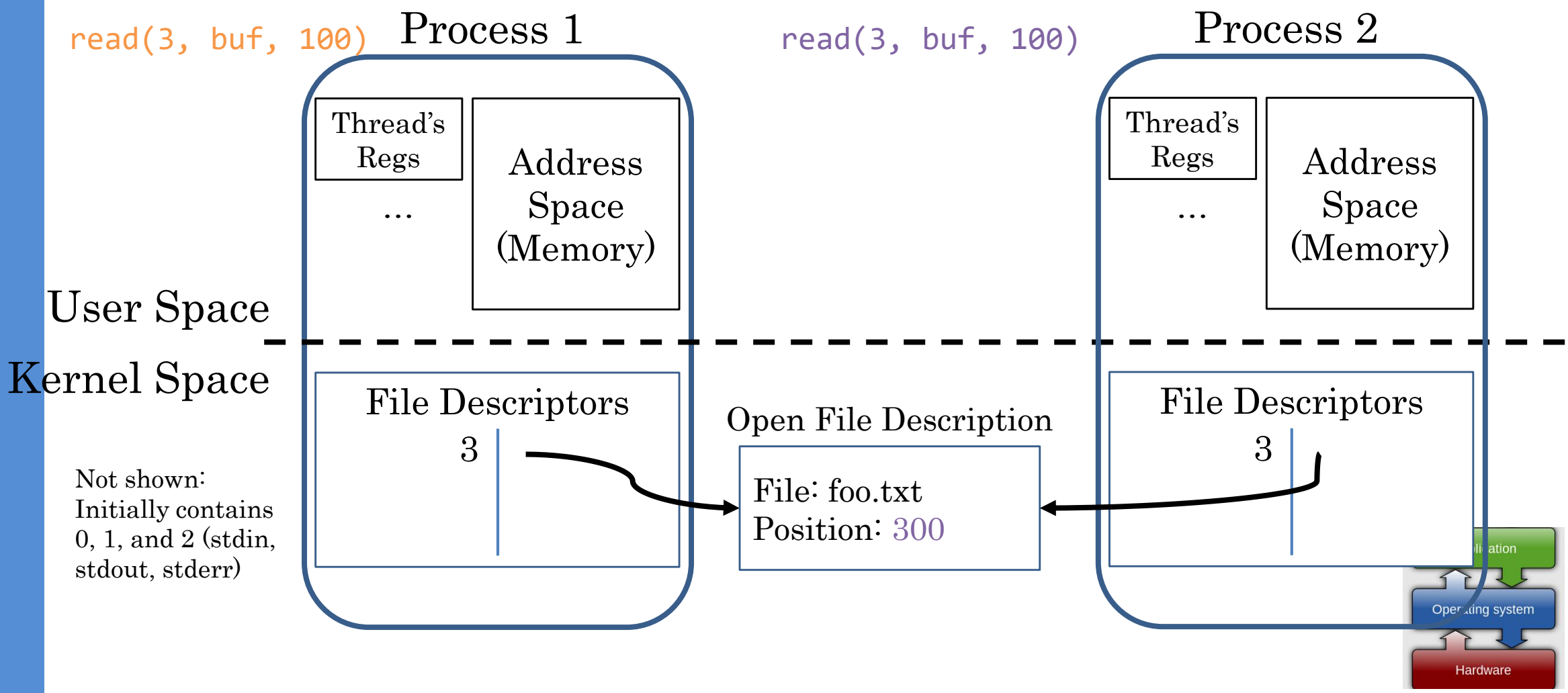
How to fork() efficiently?

- Alias the pages
  - Same physical address!
  - If we stopped here, the data would be shared (not what we want)
- Mark PTEs read-only
  - If a process tries to write → trap to the OS
- On first write to a page after `fork()`, kernel copies the page, marks PTEs as writeable
- Illusion of separate memory, but really aliased until first write

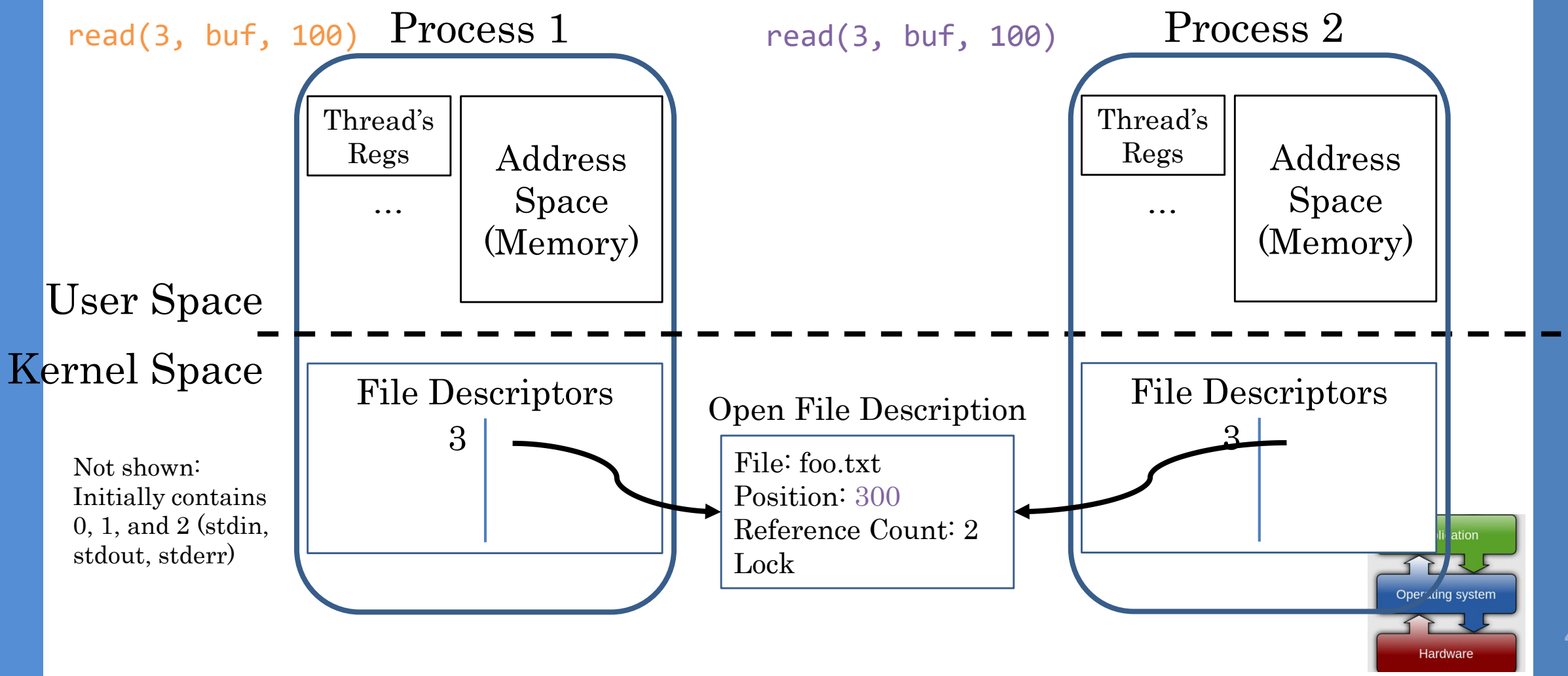
PintOS doesn't support `fork()`, just `process_create()`



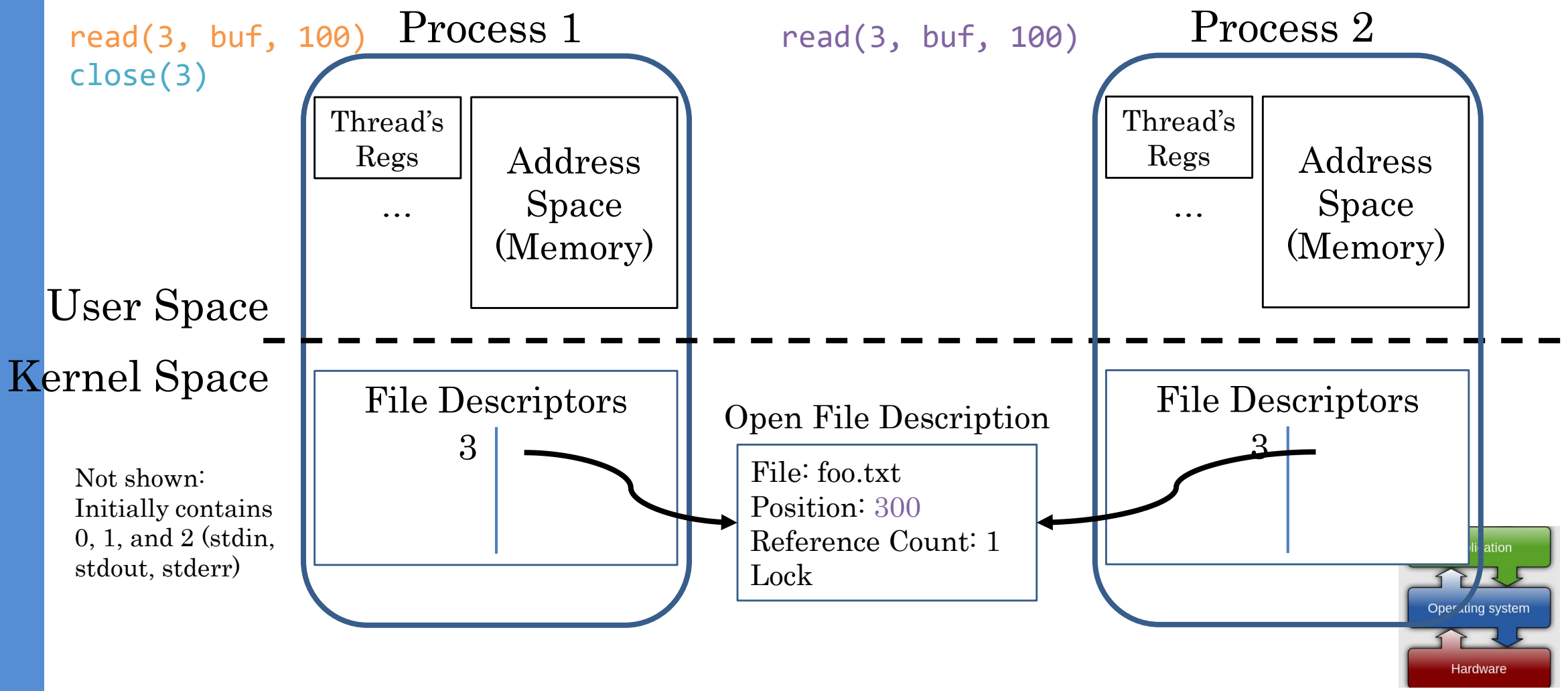
Recall: Open File Description is Aliased



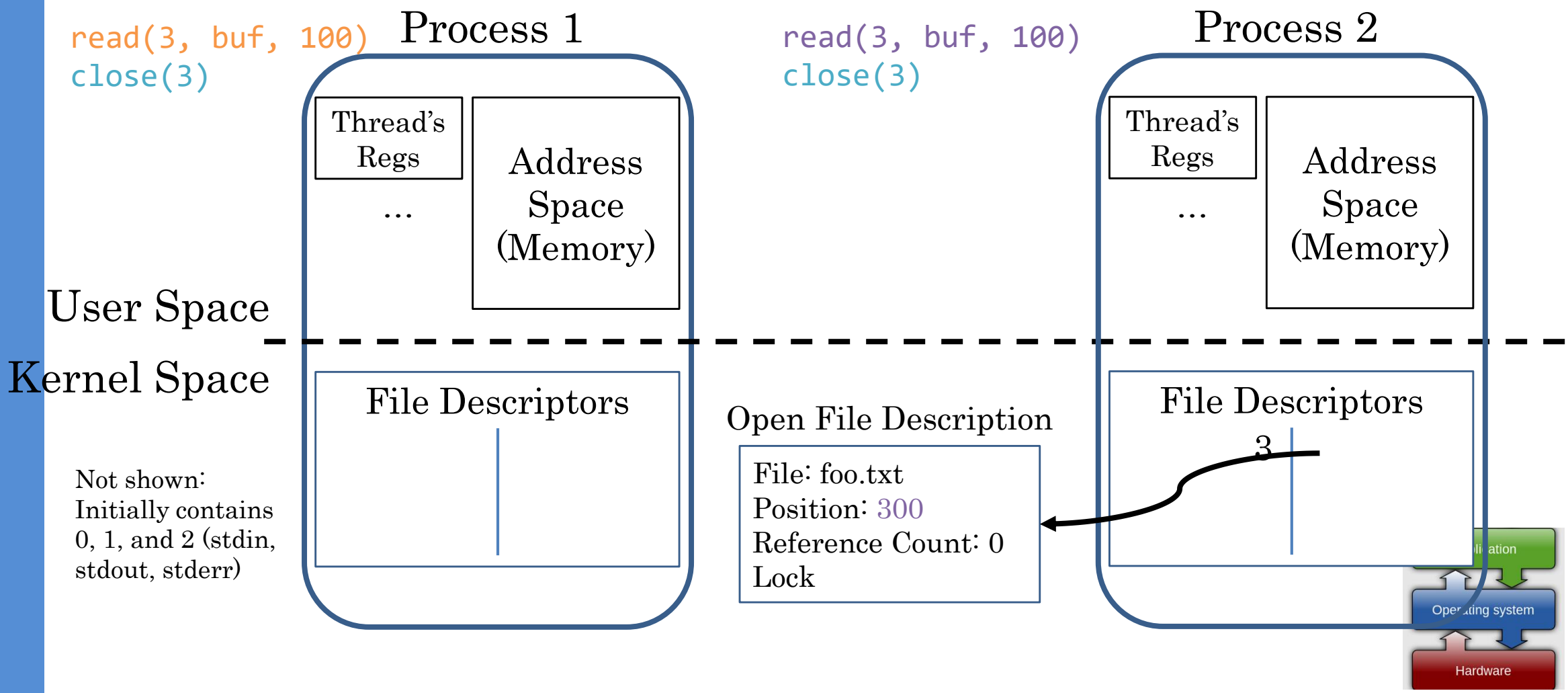
Solution: Reference Counting



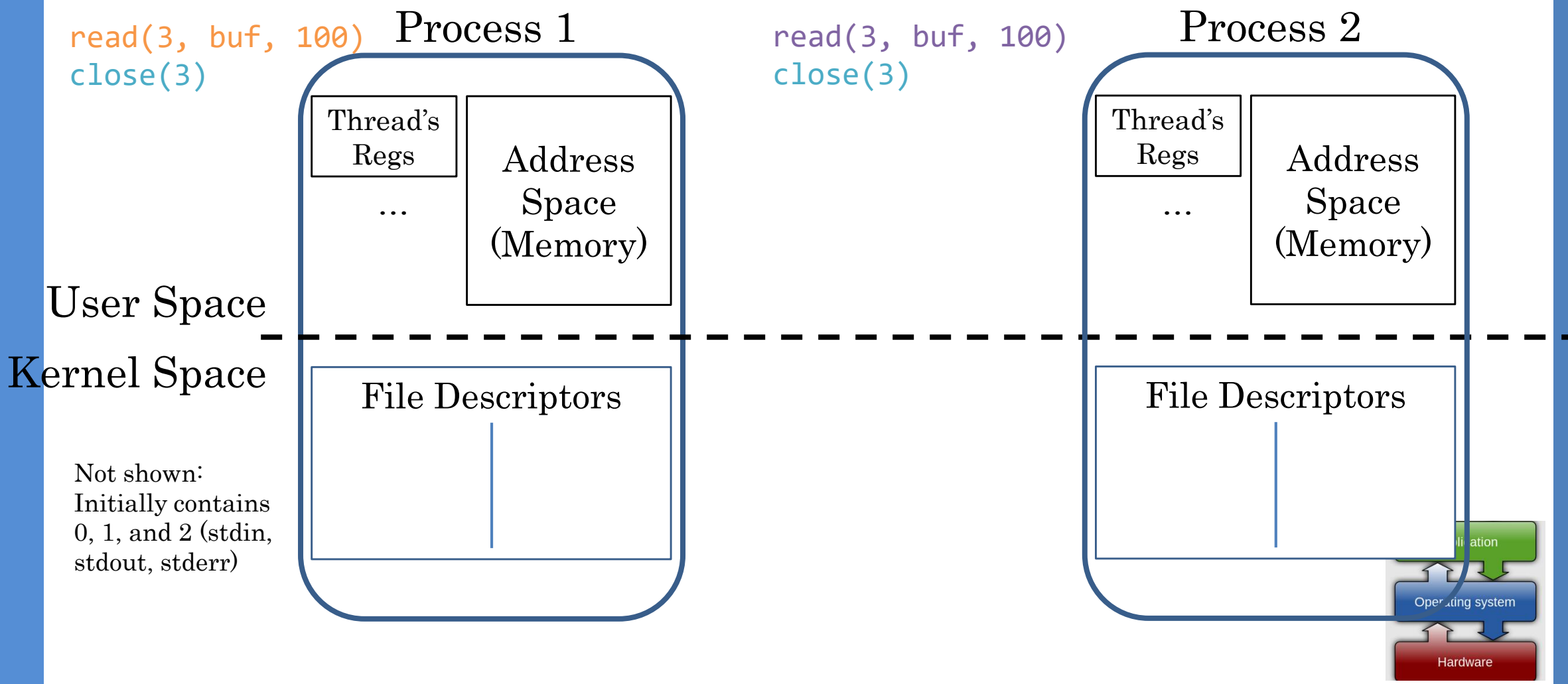
Solution: Reference Counting



Solution: Reference Counting

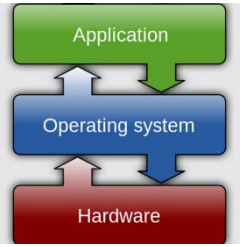


Solution: Reference Counting



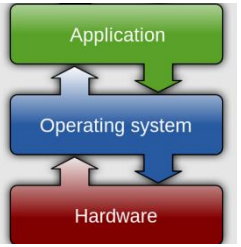
What about wait()?

- The parent process needs to get the exit code
- The following events may happen in any order (or concurrently)
  - Parent process calls `wait()` or `exit()`
  - Child process calls `exit()`
- Where should the child put its exit code?
  - Needs to work even if the parent has exited
- Where should the parent search for the exit code?
  - Needs to work even if the child has exited already

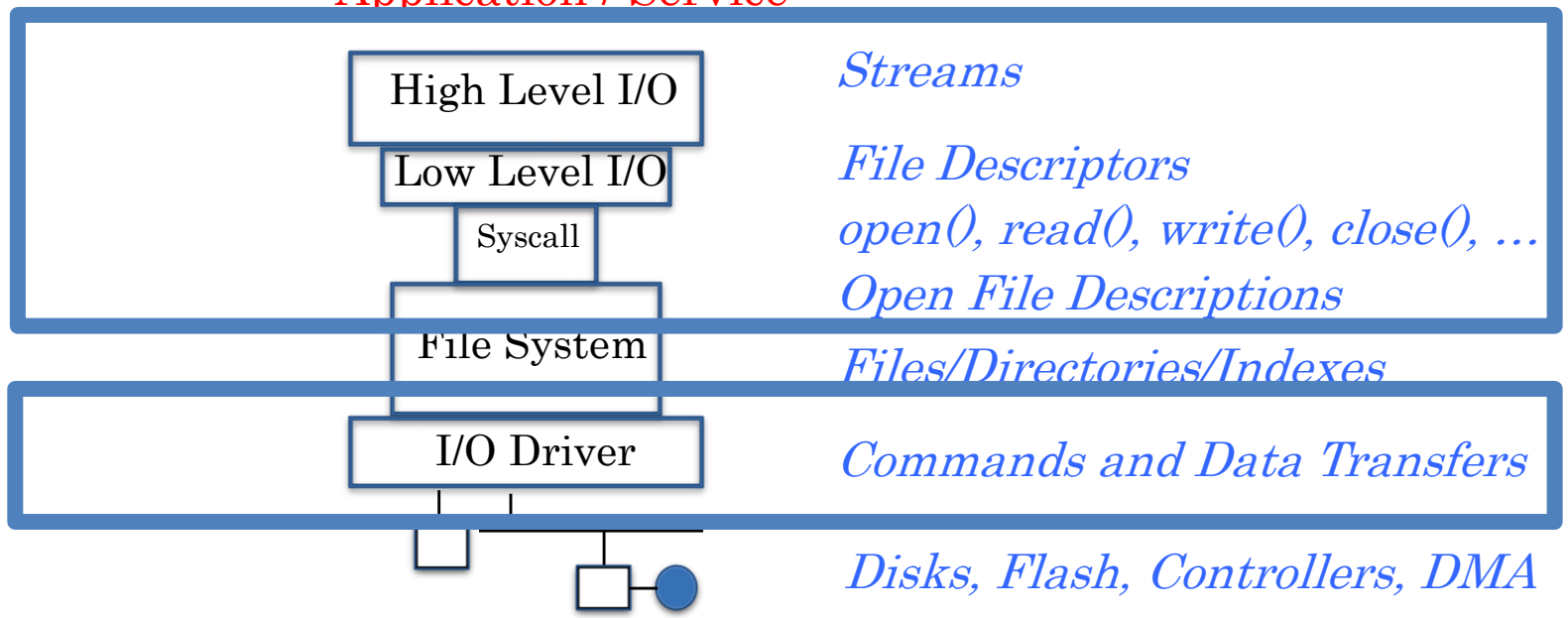


Today: How Does the OS Support the Process Abstraction?

- Support for threads and kernel structure
- Memory layout
- Support for process operations
- Support for I/O
- Influence of IPC/RPC on kernel structure

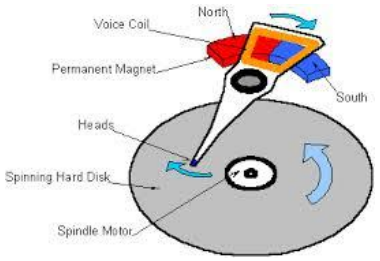


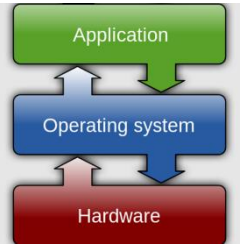
Recall: I/O and Storage Layers



What we've covered so far…

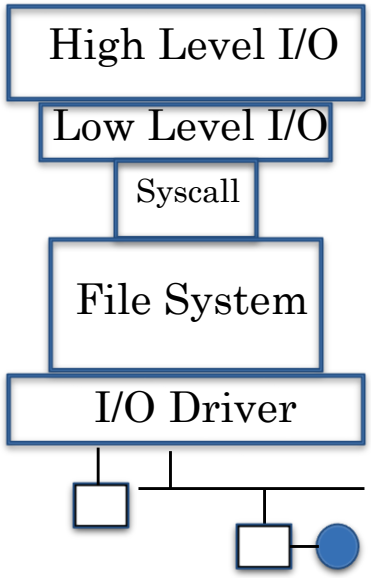
What we'll peek at today





Layers…

Application / Service



User App

```
length = read(input_fd, buffer, BUFFER_SIZE);
```

User library

```
ssize_t read(int, void *, size_t){
   marshal args into registers
   issue syscall
   register result of syscall to rtn value
};
```

Kernel

Exception U→K, interrupt processing

```
void syscall_handler (struct intr_frame *f) {
   unmarshall call#, args from regs
   dispatch : handlers[call#](args)
   marshal results from syscall ret
}
```

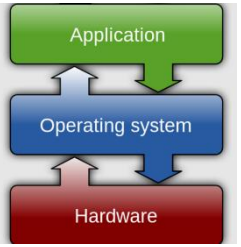
Device Driver

```
ssize_t vfs_read(struct file *file, char __user
*buf, size_t count, loff_t *pos)
{
     User Process/File System relationship
     call device driver to do the work
}
```

Low-Level Driver

- Associated with particular hardware device
- Registers / Unregisters itself with the kernel
- Handler functions for each of the file operations

```
struct file_operations {
    struct module *owner;
    loff_t (*llseek) (struct file *, loff_t, int);
    ssize_t (*read) (struct file *, char __user *, size_t, loff_t *);
    ssize_t (*write) (struct file *, const char __user *, size_t, loff_t *);
    ssize_t (*aio_read) (struct kiocb *, const struct iovec *, unsigned long, loff_t);
    ssize_t (*aio_write) (struct kiocb *, const struct iovec *, unsigned long, loff_t);
    int (*readdir) (struct file *, void *, filldir_t);
    unsigned int (*poll) (struct file *, struct poll_table_struct *);
    int (*ioctl) (struct inode *, struct file *, unsigned int, unsigned long);
    int (*mmap) (struct file *, struct vm_area_struct *);
    int (*open) (struct inode *, struct file *);
    int (*flush) (struct file *, fl_owner_t id);
    int (*release) (struct inode *, struct file *);
    int (*fsync) (struct file *, struct dentry *, int datasync);
    int (*fasync) (int, struct file *, int);
    int (*flock) (struct file *, int, struct file_lock *);
    [...]
};
```

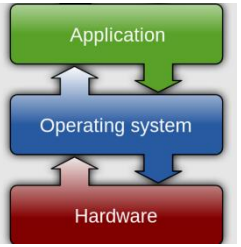


File System: From Syscall to Driver

```
ssize_t vfs_read(struct file *file, char __user *buf, size_t count, loff_t *pos)
{
  ssize_t ret;
  if (!(file->f_mode & FMODE_READ)) return -EBADF;
  if (!file->f_op || (!file->f_op->read && !file->f_op->aio_read))
    return -EINVAL;
  if (unlikely(!access_ok(VERIFY_WRITE, buf, count))) return -EFAULT;
  ret = rw_verify_area(READ, file, pos, count);
  if (ret >= 0) {
    count = ret;
    if (file->f_op->read)
      ret = file->f_op->read(file, buf, count, pos);
    else
      ret = do_sync_read(file, buf, count, pos);
    if (ret > 0) {
      fsnotify_access(file->f_path.dentry);
      add_rchar(current, ret);
    }
    inc_syscr(current);
  }
  return ret;
}
```

- Read up to `count` bytes from `file` starting from `pos` into `buf`.
- Return error or number of bytes read.

Linux: `fs/read_write.c`

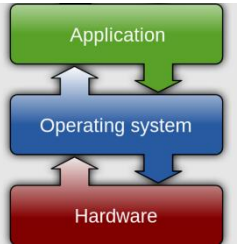


File System: From Syscall to Driver

```
ssize_t vfs_read(struct file *file, char __user *buf, size_t count, loff_t *pos)
{
  ssize_t ret;
  if (!(file->f_mode & FMODE_READ)) return -EBADF;
  if (!file->f_op || (!file->f_op->read && !file->f_op->aio_read))
    return -EINVAL;
  if (unlikely(!access_ok(VERIFY_WRITE, buf, count))) return -EFAULT;
  ret = rw_verify_area(READ, file, pos, count);
  if (ret >= 0) {
    count = ret;
    if (file->f_op->read)
      ret = file->f_op->read(file, buf, count, pos);
    else
      ret = do_sync_read(file, buf, count, pos);
    if (ret > 0) {
      fsnotify_access(file->f_path.dentry);
      add_rchar(current, ret);
    }
    inc_syscr(current);
  }
  return ret;
}
```

Make sure we are allowed to read this file

Linux: `fs/read_write.c`

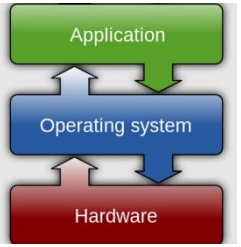


File System: From Syscall to Driver

```
ssize_t vfs_read(struct file *file, char __user *buf, size_t count, loff_t *pos)
{
  ssize_t ret;
  if (!(file->f_mode & FMODE_READ)) return -EBADF;
  if (!file->f_op || (!file->f_op->read && !file->f_op->aio_read))
    return -EINVAL;
  if (unlikely(!access_ok(VERIFY_WRITE, buf, count))) return -EFAULT;
  ret = rw_verify_area(READ, file, pos, count);
  if (ret >= 0) {
    count = ret;
    if (file->f_op->read)
      ret = file->f_op->read(file, buf, count, pos);
    else
      ret = do_sync_read(file, buf, count, pos);
    if (ret > 0) {
      fsnotify_access(file->f_path.dentry);
      add_rchar(current, ret);
    }
    inc_syscr(current);
  }
  return ret;
}
```

Check if file has read methods

Linux: `fs/read_write.c`

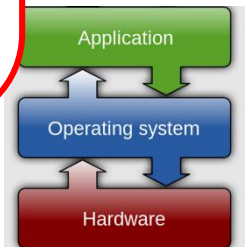


File System: From Syscall to Driver

```
ssize_t vfs_read(struct file *file, char __user *buf, size_t count, loff_t *pos)
{
  ssize_t ret;
  if (!(file->f_mode & FMODE_READ)) return -EBADF;
  if (!file->f_op || (!file->f_op->read && !file->f_op->aio_read))
    return -EINVAL;
  if (unlikely(!access_ok(VERIFY_WRITE, buf, count))) return -EFAULT;
  ret = rw_verify_area(READ, file, pos, count);
  if (ret >= 0) {
    count = ret;
    if (file->f_op->read)
      ret = file->f_op->read(file, buf, count, pos);
    else
      ret = do_sync_read(file, buf, count, pos);
    if (ret > 0) {
      fsnotify_access(file->f_path.dentry);
      add_rchar(current, ret);
    }
    inc_syscr(current);
  }
  return ret;
}
```

- Check whether we can write to `buf` (e.g., `buf` is in the user space range)
- `unlikely()`: hint to branch prediction this condition is unlikely

Linux: `fs/read_write.c`

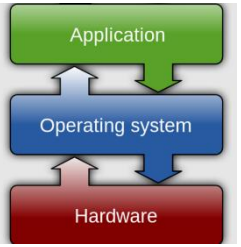


File System: From Syscall to Driver

```
ssize_t vfs_read(struct file *file, char __user *buf, size_t count, loff_t *pos)
{
  ssize_t ret;
  if (!(file->f_mode & FMODE_READ)) return -EBADF;
  if (!file->f_op || (!file->f_op->read && !file->f_op->aio_read))
    return -EINVAL;
  if (unlikely(!access_ok(VERIFY_WRITE, buf, count))) return -EFAULT;
  ret = rw_verify_area(READ, file, pos, count);
  if (ret >= 0) {
    count = ret;
    if (file->f_op->read)
      ret = file->f_op->read(file, buf, count, pos);
    else
      ret = do_sync_read(file, buf, count, pos);
    if (ret > 0) {
      fsnotify_access(file->f_path.dentry);
      add_rchar(current, ret);
    }
    inc_syscr(current);
  }
  return ret;
}
```

Check whether we read from a valid range in the file.

Linux: `fs/read_write.c`

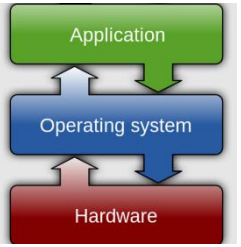


File System: From Syscall to Driver

```
ssize_t vfs_read(struct file *file, char __user *buf, size_t count, loff_t *pos)
{
  ssize_t ret;
  if (!(file->f_mode & FMODE_READ)) return -EBADF;
  if (!file->f_op || (!file->f_op->read && !file->f_op->aio_read))
    return -EINVAL;
  if (unlikely(!access_ok(VERIFY_WRITE, buf, count))) return -EFAULT;
  ret = rw_verify_area(READ, file, pos, count);
  if (ret >= 0) {
    count = ret;
    if (file->f_op->read)
      ret = file->f_op->read(file, buf, count, pos);
    else
      ret = do_sync_read(file, buf, count, pos);
    if (ret > 0) {
      fsnotify_access(file->f_path.dentry);
      add_rchar(current, ret);
    }
    inc_syscr(current);
  }
  return ret;
}
```

If driver provide a read function (`f_op->read`) use it; otherwise use `do_sync_read()`

Linux: `fs/read_write.c`

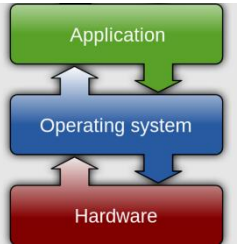


File System: From Syscall to Driver

```
ssize_t vfs_read(struct file *file, char __user *buf, size_t count, loff_t *pos)
{
  ssize_t ret;
  if (!(file->f_mode & FMODE_READ)) return -EBADF;
  if (!file->f_op || (!file->f_op->read && !file->f_op->aio_read))
    return -EINVAL;
  if (unlikely(!access_ok(VERIFY_WRITE, buf, count))) return -EFAULT;
  ret = rw_verify_area(READ, file, pos, count);
  if (ret >= 0) {
    count = ret;
    if (file->f_op->read)
      ret = file->f_op->read(file,
    else
      ret = do_sync_read(file, buf, count, pos);
    if (ret > 0) {
      fsnotify_access(file->f_path.dentry);
      add_rchar(current, ret);
    }
    inc_syscr(current);
  }
  return ret;
}
```

Notify the parent of this file that the file was read (see http://www.fieldses.org/~bfields/kernel/vfs.txt)

Linux: `fs/read_write.c`

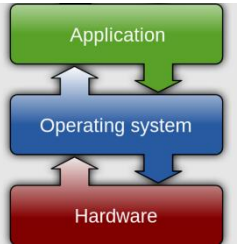


File System: From Syscall to Driver

```
ssize_t vfs_read(struct file *file, char __user *buf, size_t count, loff_t *pos)
{
  ssize_t ret;
  if (!(file->f_mode & FMODE_READ)) return -EBADF;
  if (!file->f_op || (!file->f_op->read && !file->f_op->aio_read))
    return -EINVAL;
  if (unlikely(!access_ok(VERIFY_WRITE, buf, count))) return -EFAULT;
  ret = rw_verify_area(READ, file, pos, count);
  if (ret >= 0) {
    count = ret;
    if (file->f_op->read)
      ret = file->f_op->read(file, buf, count, pos);
    else
      ret = do_sync_read(file, buf, count, pos);
    if (ret > 0) {
      fsnotify_access(file->f_path.dentry);
      add_rchar(current, ret);
    }
    inc_syscr(current);
  }
  return ret;
}
```

Update the number of bytes read by "current" task (for scheduling purposes)

Linux: `fs/read_write.c`

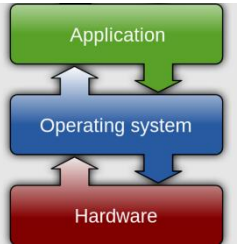


File System: From Syscall to Driver

```
ssize_t vfs_read(struct file *file, char __user *buf, size_t count, loff_t *pos)
{
  ssize_t ret;
  if (!(file->f_mode & FMODE_READ)) return -EBADF;
  if (!file->f_op || (!file->f_op->read && !file->f_op->aio_read))
    return -EINVAL;
  if (unlikely(!access_ok(VERIFY_WRITE, buf, count))) return -EFAULT;
  ret = rw_verify_area(READ, file, pos, count);
  if (ret >= 0) {
    count = ret;
    if (file->f_op->read)
      ret = file->f_op->read(file, buf, count, pos);
    else
      ret = do_sync_read(file, buf, count, pos);
    if (ret > 0) {
      fsnotify_access(file->f_path.dentry);
      add_rchar(current, ret);
    }
    inc_syscr(current);
  }
  return ret;
}
```

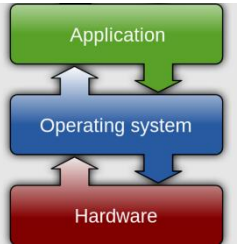
Update the number of read syscalls by "current" task (for scheduling purposes)

Linux: `fs/read_write.c`



Device Drivers

- Device-specific code in the kernel that interacts directly with the device hardware
  - Supports a standard, internal interface
  - Same kernel I/O system can interact easily with different device drivers
  - Special device-specific configuration supported with the `ioctl()` system call
- Device Drivers 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(), strategy()
    - This is the kernel's interface to the 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



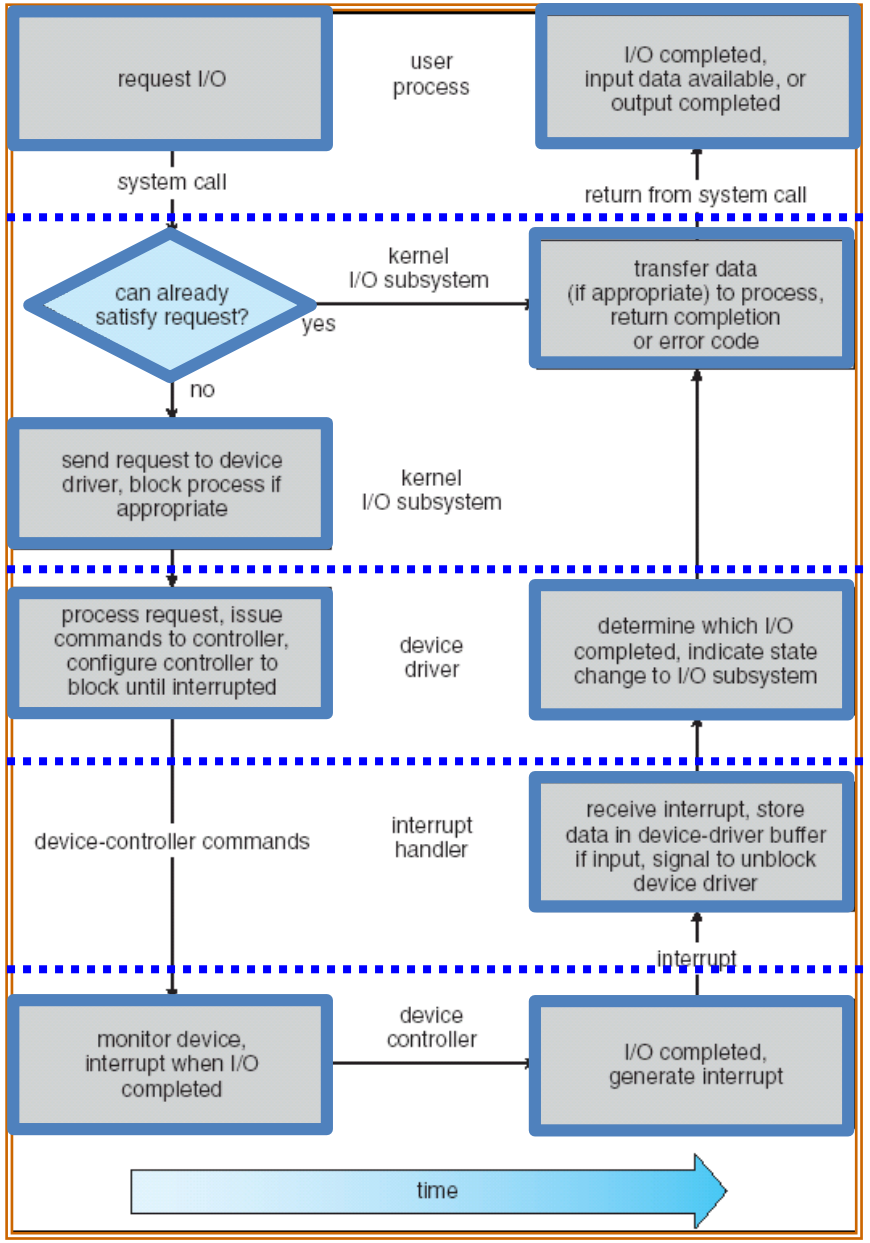
User Program

Kernel I/O Subsystem

Device Driver Top Half

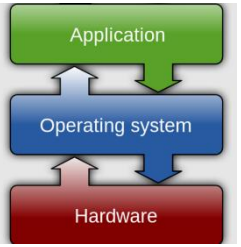
Device Driver Bottom Half

Device Hardware

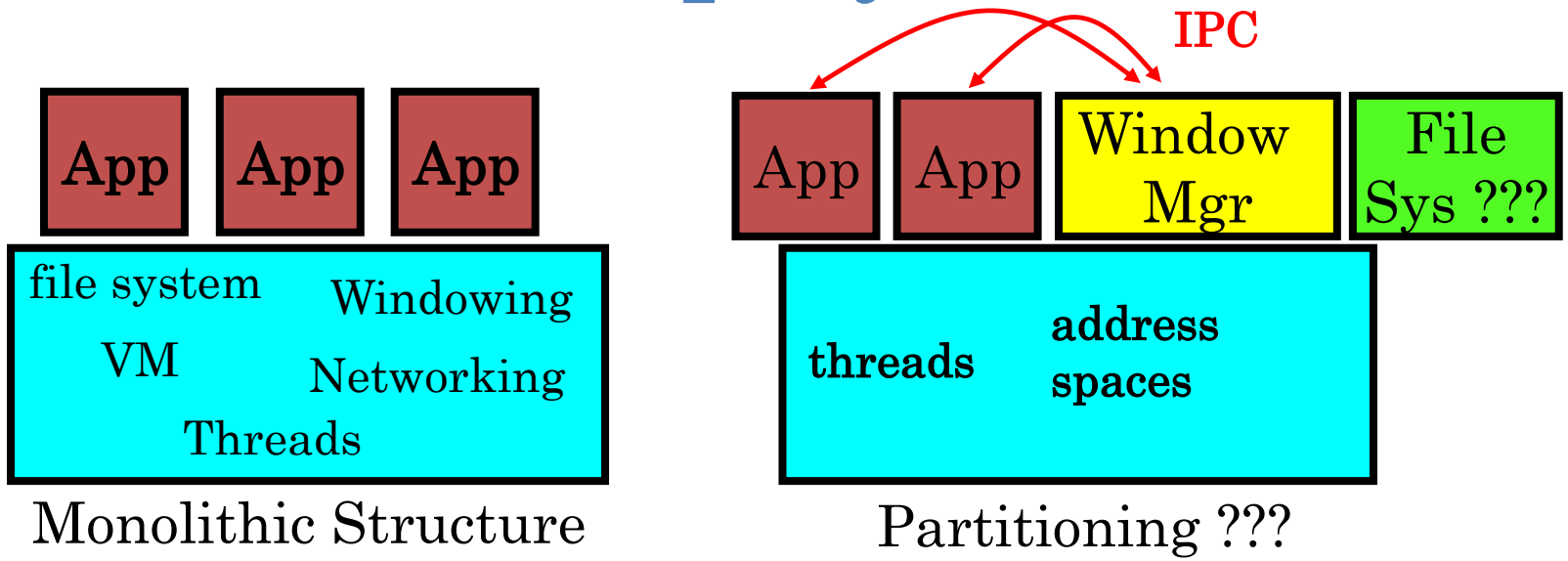


Today: How Does the OS Support the Process Abstraction?

- Support for threads and kernel structure
- Memory layout
- Support for process operations
- Support for I/O
- Influence of IPC/RPC on kernel structure

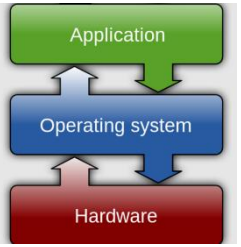


Recall: IPC to Simplify OS



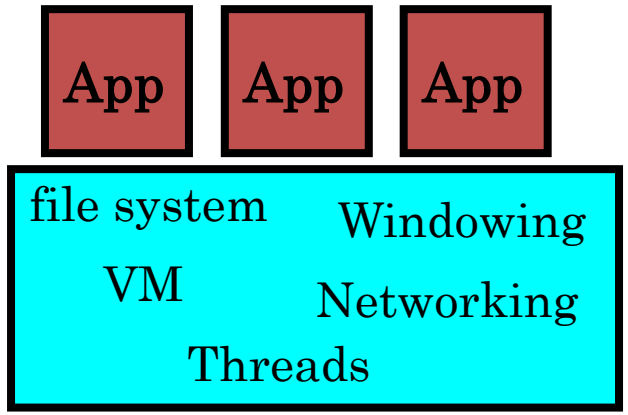
What if the file system is not local to the machine, but on the network?

- Is there a general mechanism for providing services to other processes?
  - Do the protocols we run on top of IPC generalize as well?

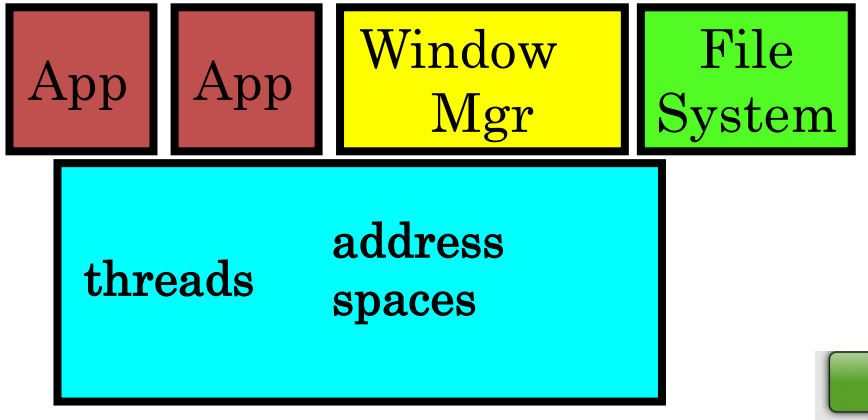


Microkernels

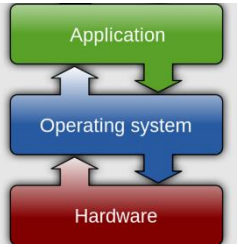
- Split OS into separate processes
  - Example: File System, Network Driver are processes outside of the kernel
- Pass messages among these components (e.g., via RPC) instead of system calls



Monolithic Structure

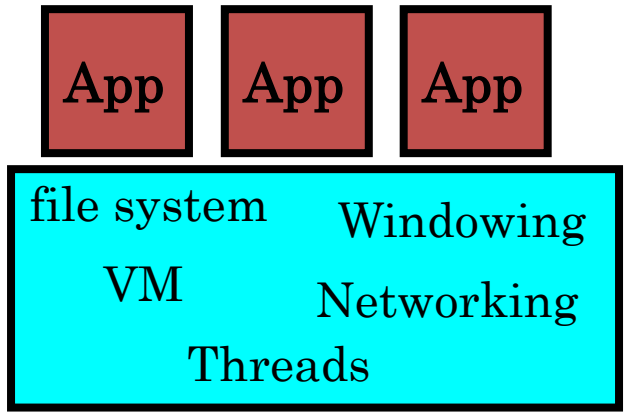


Microkernel Structure

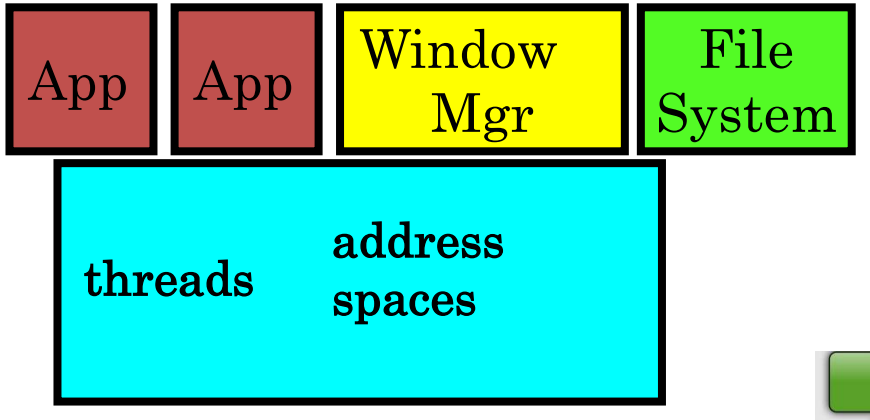


Microkernels

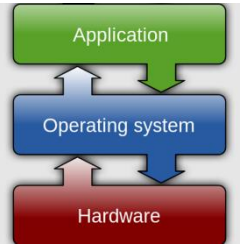
- Microkernel itself provides only essential services
  - Communication
  - Address space management
  - Thread scheduling
  - Almost-direct access to hardware devices (for driver processes)



Monolithic Structure



Microkernel Structure



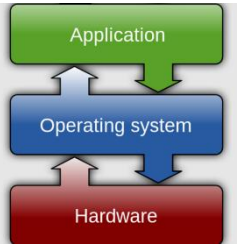
Why Microkernels?

Pros

- Failure Isolation
- Easier to update/replace parts
- Easier to distribute – build one OS that encompasses multiple machines

Cons

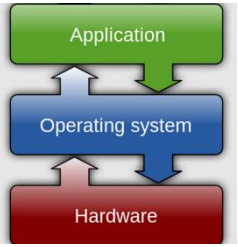
- More communication overhead and context switching
- Harder to implement?



Flashback: What is an OS?

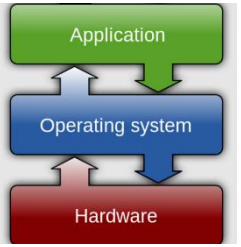
- Always:
  - Memory Management
  - I/O Management
  - CPU Scheduling
  - Communications
  - Multitasking/multiprogramming
- Maybe:
  - File System?
  - Multimedia Support?
  - User Interface?
  - Web Browser?

Not provided in a strict microkernel



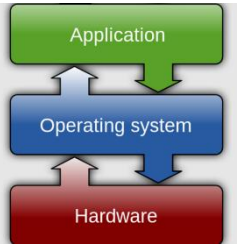
Influence of Microkernels

- Many operating systems provide some services externally, similar to a microkernel
  - OS X and Linux: Windowing (graphics and UI)
- Some currently monolithic OSes started as microkernels
  - Windows family originally had microkernel design
  - OS X: Hybrid of Mach microkernel and FreeBSD monolithic kernel



Conclusion

- We studied the structure of the kernel
  - Kernel thread backing every user thread
- We saw how the kernel organizes a process' memory
  - Kernel memory mapped into each process' virtual address space
- We saw how the kernel supports operations on processes
  - `fork`, `wait`, `exec`, `open` file descriptions…
- We saw how the kernel handles I/O
  - Device drivers
- We saw how IPC influences the structure of the kernel
  - Service provide by other processes



LSU CENTER FOR COMPUTATION & TECHNOLOGY

STE||AR GROUP