

Memory 1: Address Translation

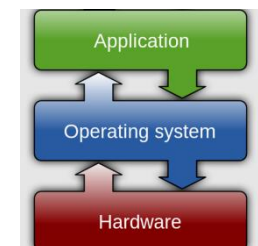
Lecture 13

Hartmut Kaiser

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

Goals for Today

- Finish up discussion of highly concurrent systems
- Start exploring OS memory management (if time permits)

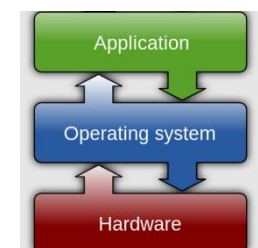


System Performance and Highly Concurrent Systems

Part 2

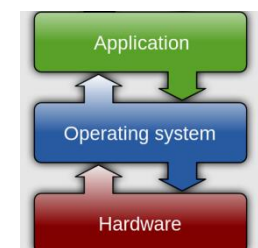
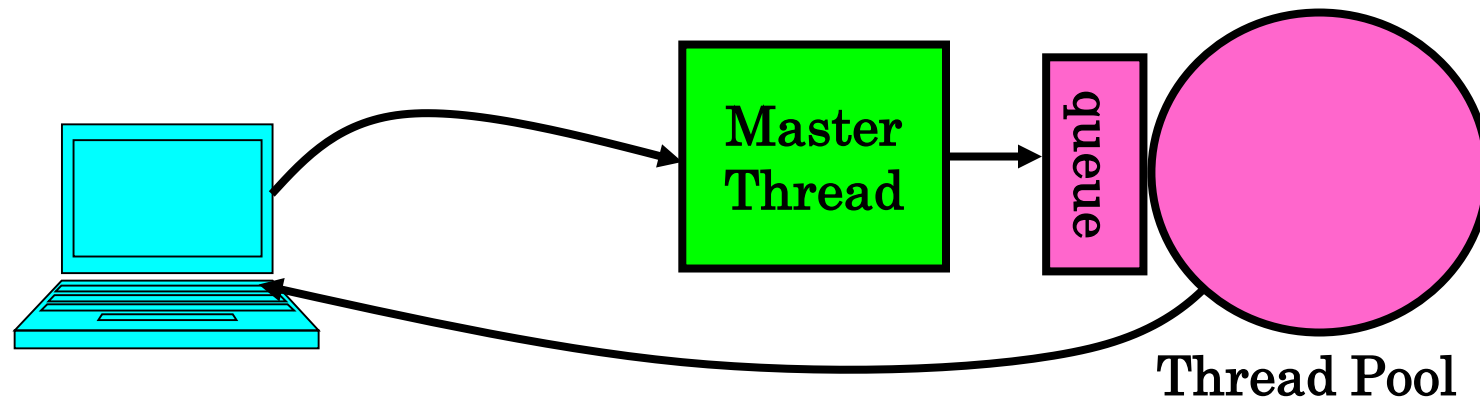
Concurrent, Well-Conditioned Systems

- Thread Pools
- User-Mode Threads
- Event-Driven Execution



Thread Pools

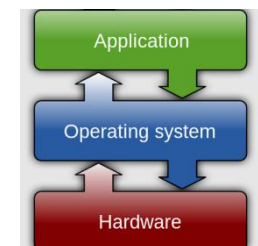
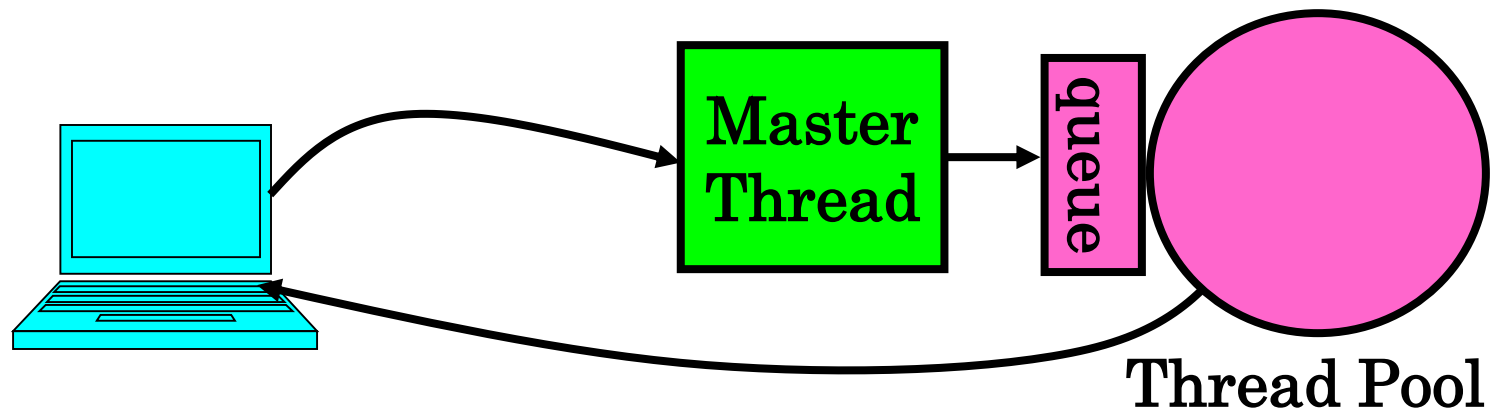
- Key idea: limit the number of threads
 - Before throughput starts to degrade
- Instead, allocate a bounded “pool” of worker threads, representing the maximum level of multiprogramming



Thread Pools

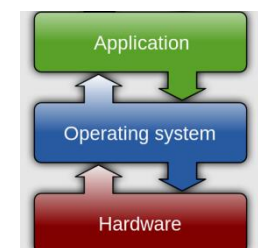
```
master() {  
  allocThreads(worker, queue);  
  while(TRUE) {  
    con = AcceptCon();  
    Enqueue(queue, con);  
  }  
}
```

```
worker(queue) {  
  while(TRUE) {  
    // Blocks if empty  
    con = Dequeue(queue);  
    ServiceWebPage(con);  
  }  
}
```



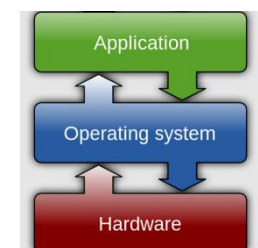
Highly Concurrent Well-Conditioned Systems

- Thread pools work well, but they somewhat limit concurrency
- Is there a good alternative?



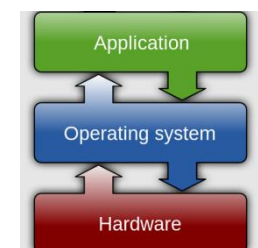
We've Looked At: Kernel-Supported Threads

- Threads run and block (e.g., on I/O) independently
 - One process may have multiple threads waiting on different things
 - Two mode switches for every context switch (expensive)
 - Create threads with syscalls
-
- Alternative: multiplex several streams of execution (at user level) on top of a single OS thread
 - E.g., Java, Go, ... (and many many user-level threads libraries before it)



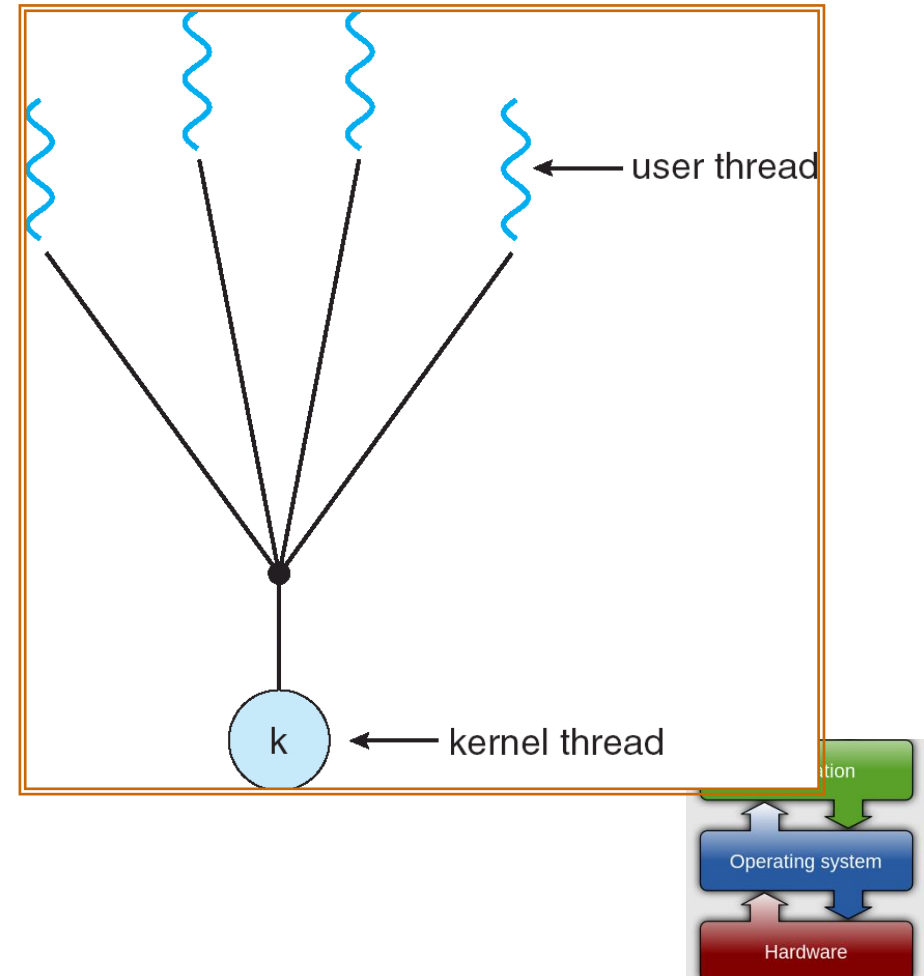
Concurrent, Well-Conditioned Systems

- Thread Pools
- User-Mode Threads
- Event-Driven Execution



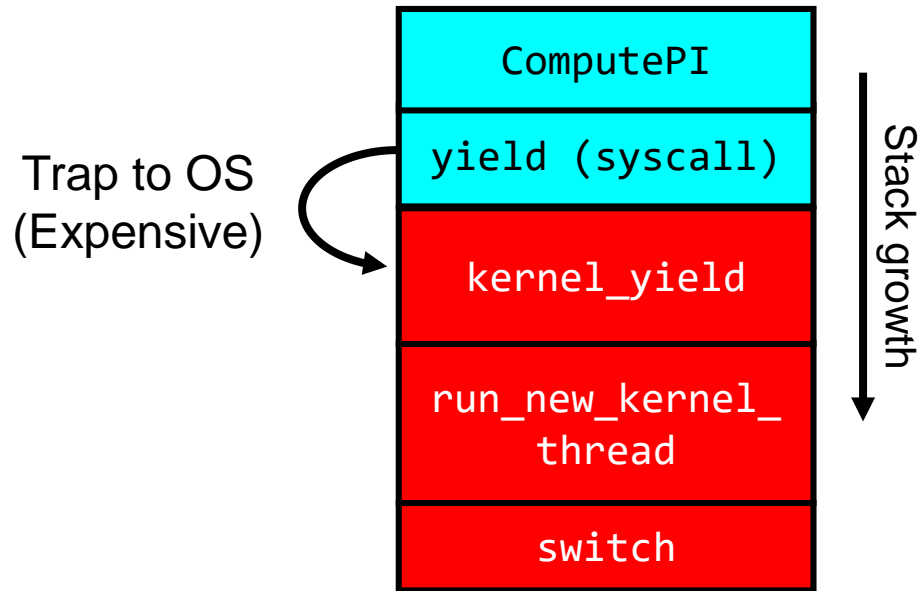
User-Mode Threads

- User program contains its own scheduler
- Several user threads per kernel thread
- User threads may be scheduled non-preemptively
 - Only switch on yield
- Context switches cheaper
 - Copy registers and jump (switch in userspace)

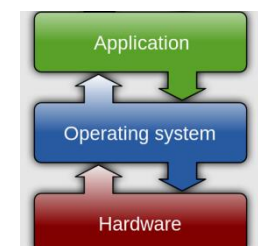
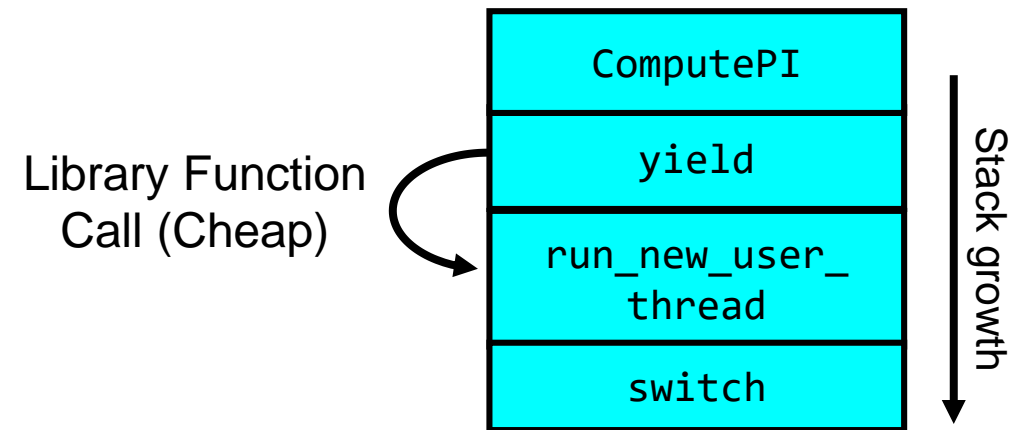


Thread Yield

Kernel-Supported Threads

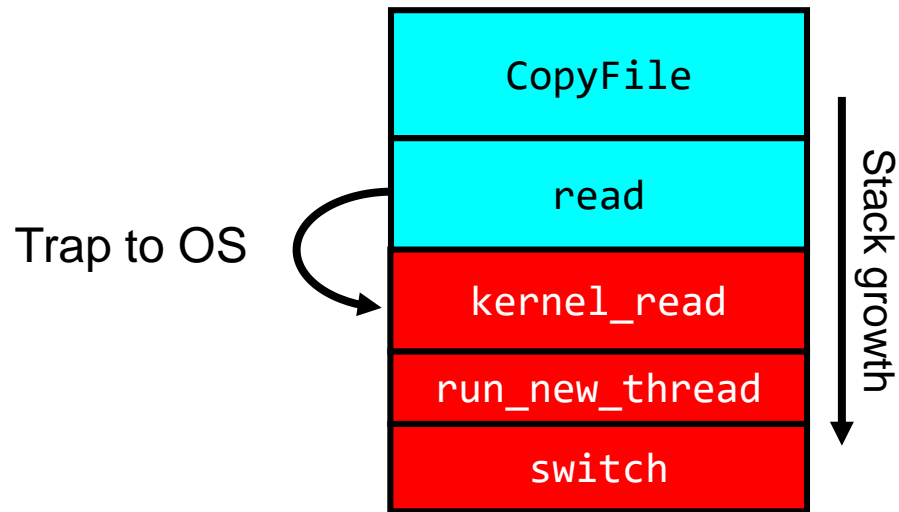


User-Mode Threads

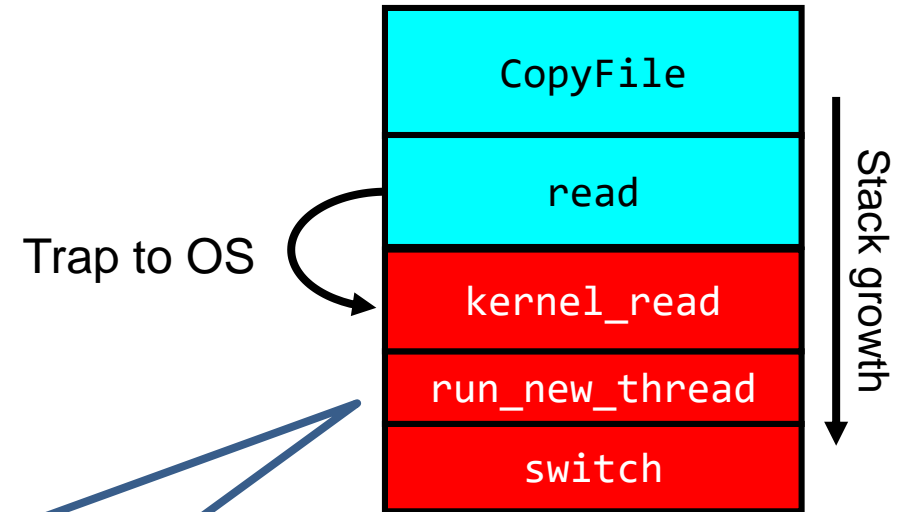


Thread I/O

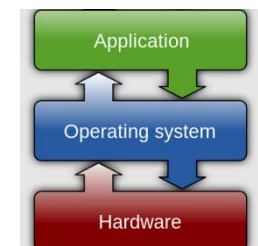
Kernel-Supported Threads



User-Mode Threads

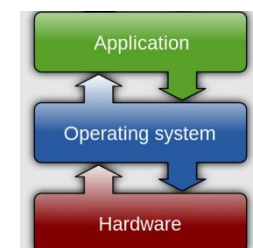


- Selects a new *kernel thread* to run
- Bypassing user-level scheduler



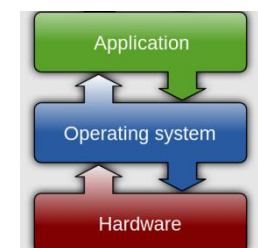
User-Mode Threads: Problems

- One user-level thread blocks on I/O: they all do
 - Kernel cannot adjust scheduling among threads it doesn't know about
- Multiple Cores?
- Can't completely avoid blocking (syscalls, page fault)
- One Solution: Scheduler Activations
 - Have kernel inform user-level scheduler when a thread blocks
 - Evolving the contract between OS and application
- Alternative Solution: Language Support?
 - Make the scheduler aware of the blocking operation



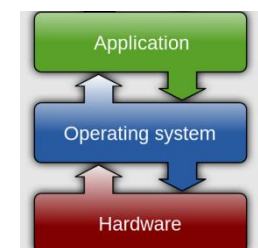
Go Goroutines

- Goroutines are lightweight, user-level threads
 - Scheduling not preemptive (relies on goroutines to yield)
 - Yield statements inserted by compiler
- Advantages relative to regular threads (e.g., pthreads)
 - More lightweight
 - Faster context-switch time
- Disadvantages
 - Less sophisticated scheduling at the user-level
 - OS is not aware of user-level threads



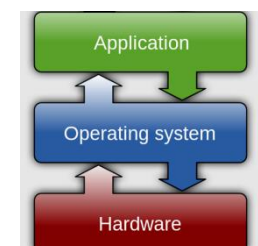
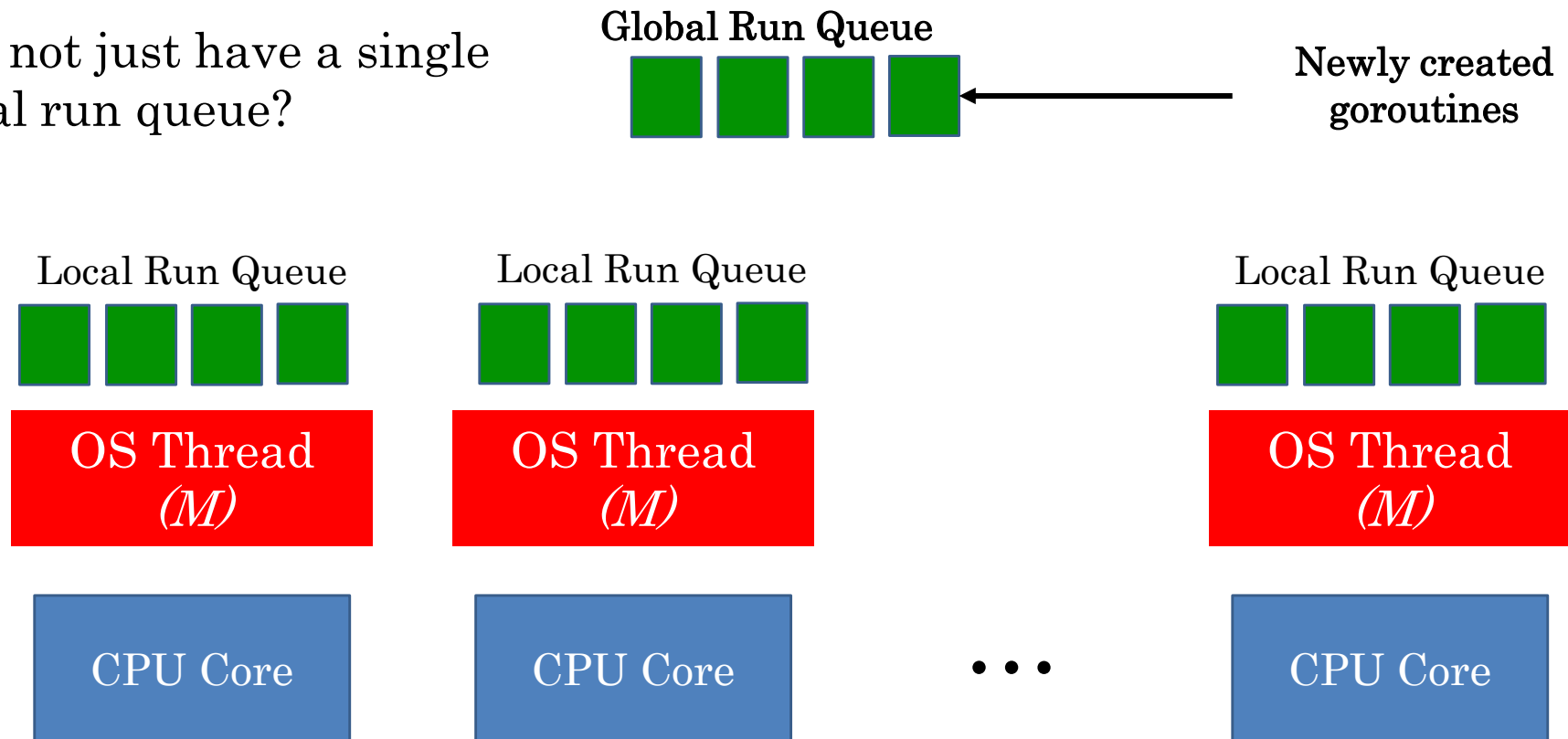
Go User-Level Scheduler

- Why this approach?
- 1 OS (kernel-supported) thread per CPU core: allows Go program to achieve parallelism not just concurrency
 - Fewer OS threads? Not utilizing all CPUs
 - More OS threads? No additional benefit
 - We'll see one exception to this involving syscalls
- Keep goroutine on same OS thread: affinity, nice for caching and performance



Go User-Level Thread Scheduler

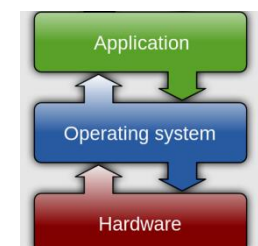
- Why not just have a single global run queue?



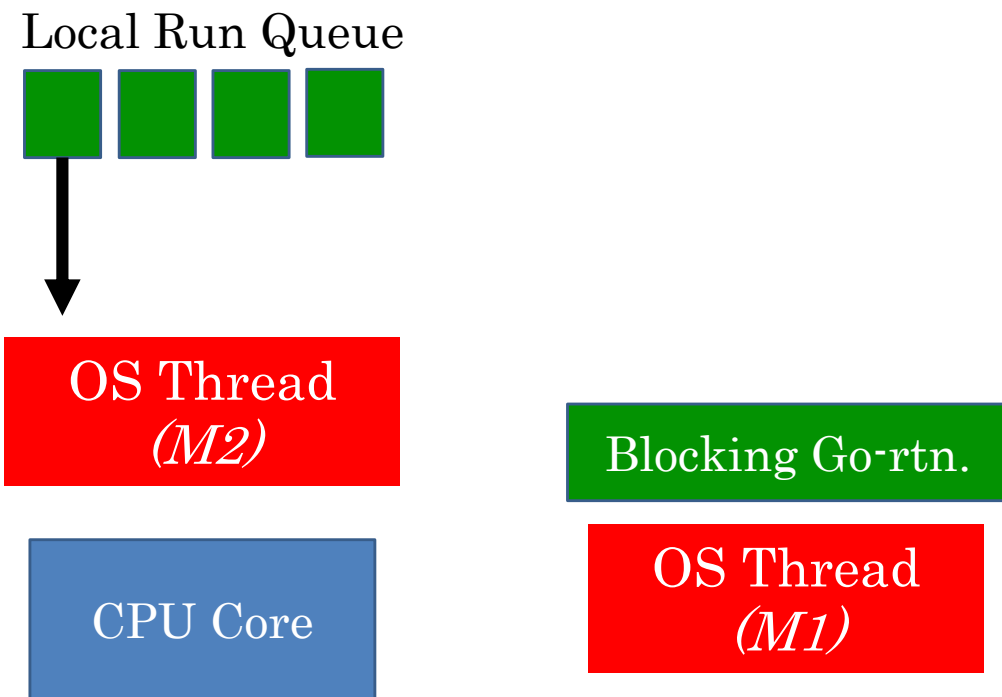
Dealing with Blocking Syscalls



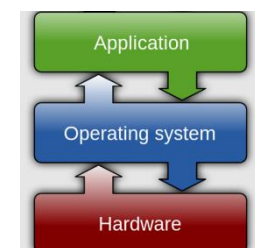
- What if a goroutine wants to make a blocking syscall?
 - Example: File I/O



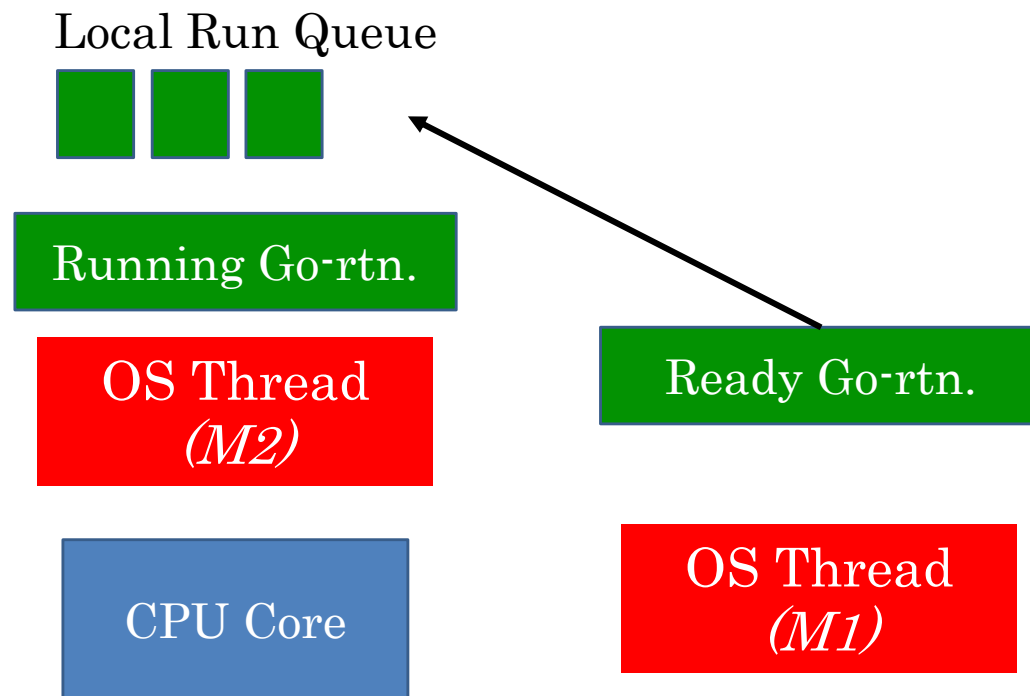
Dealing with Blocking Syscalls



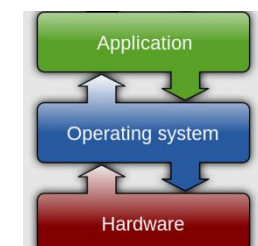
- What if a goroutine wants to make a blocking syscall?
 - Example: File I/O
- While syscall is blocking, allocate new OS thread (M2)
 - M1 is blocked by kernel, M2 lets us continue using CPU



Dealing with Blocking Syscalls

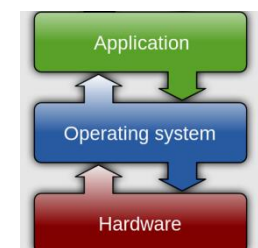


- Syscall completes: Put invoking goroutine back on queue
- Keep M1 around in a spare pool
- Swap it with M2 upon next syscall, no need to pay thread creation cost



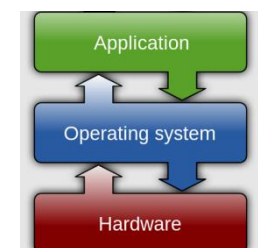
Concurrent, Well-Conditioned Systems

- Thread Pools
- User-Mode Threads
- Event-Driven Execution



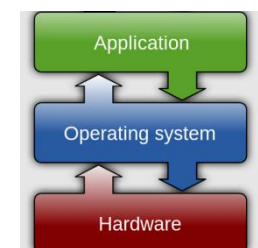
Event-Driven Execution

- Allows a system to handle MTAO with a single thread
 - Very lightweight
- Key idea: juggle different tasks within a single thread
 - All tasks' CPU bursts execute within a single thread
 - I/O bursts for each task happen in the background without a backing thread



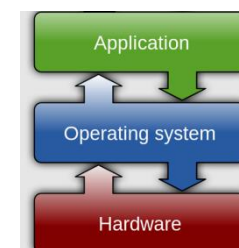
Event-Driven Server Concept

```
while (true) {  
    int task_id = <wait for a task to become ready>  
    <look up state for task_id>  
    <execute next CPU burst for this task>  
    if (task is done) {  
        <forget state for task_id>  
        continue;  
    }  
    <issue task's next (I/O) operation>  
    <update state for task_id>  
}
```



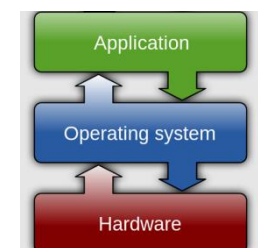
How to “Issue Task’s Next I/O Operation”?

- So far, we’ve seen read and write, which block the calling thread
- We can put file descriptors into non-blocking mode
 - read: Just return whatever data is available
 - write: Just write whatever the kernel can buffer in its memory for now
 - So read/write calls may not read or write anything
- How to wait for the next task to become ready to perform its I/O?



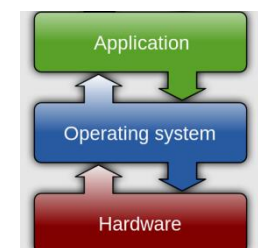
How to “Wait for Task to Become Ready”?

- POSIX provides a way to wait for one of several files to have data available
 - `select/poll` system calls
 - Provide a list of file descriptors
 - Blocks until at least one has “ready” data, then returns which ones do
 - Mixes well with non-blocking I/O, especially sockets



Alternative Asynchronous I/O APIs

- Unfortunately, non-blocking mode and `select/poll` don't work well with regular files
- Instead: there's the asynchronous I/O interface
 - `io_submit` issues a disk I/O
 - `io_getevents` syscall reaps completion of disk I/Os issued with `io_submit`
- Newer, better APIs still emerging (e.g., `io_uring`)

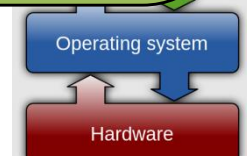


Event-Driven Server Concept

```
while (true) {  
    int task_id = <wait for a task to become ready>  
    <look up state for task_id>  
    <execute next CPU burst for this task>  
    if (task is done) {  
        <forget state for task_id>  
        continue;  
    }  
    <issue task's next (I/O) operation>  
    <update state for task_id>  
}
```

This looks kind of like the OS thread scheduler...

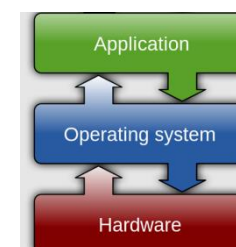
But it runs in the user program!



User-Mode Scheduler Based on Event Loop

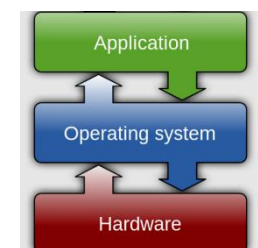
- User-mode scheduler can be an event-loop
- User threads use I/O library that issues async I/O operations
- Now user-mode scheduler can properly suspend the thread...

- But only works for I/O operations for which the kernel supports an asynchronous interface



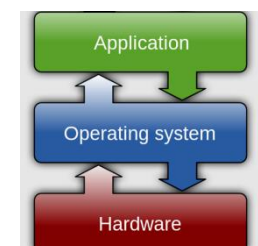
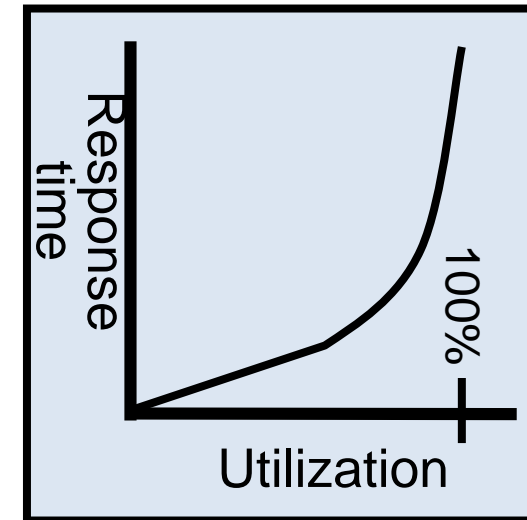
User-Mode Scheduling vs. Event Loops

- In user-mode scheduling:
 - You're still maintaining a separate stack for each thread
 - Must save PC, stack, registers when switching
 - Even if you use async I/O operations to properly suspend the user thread
- In pure event-driven scheduling:
 - All events execute in the same stack
 - All state to resume each task (e.g., which stage we're at) must be stored explicitly



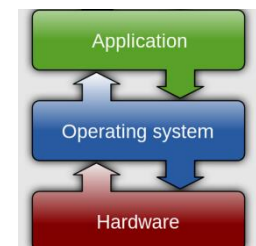
Final Word on Scheduling

- When do the details of the scheduling policy and fairness really matter?
- When should you simply buy a faster computer?
 - One approach: Buy it when it will pay for itself in improved response time
 - Perhaps you're paying for worse response time in reduced productivity, customer angst, etc...
 - Might think that you should buy a faster X when X is utilized 100%, but usually, response time goes to infinity as utilization \Rightarrow 100%
- An interesting implication of this curve:
 - Most scheduling algorithms work fine in the “linear” portion of the load curve, fail otherwise
 - Argues for buying a faster X when hit “knee” of curve



Announcements

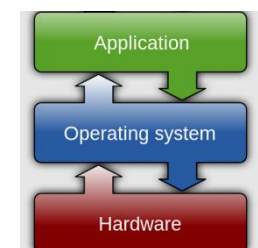
- Assignment 3 due April 27
 - No deadline extensions will be possible



Address Translation

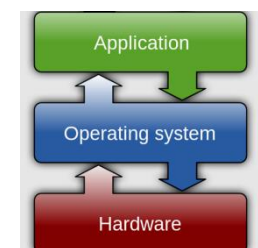
Next Objective

- Dive deeper into the concepts and mechanisms of memory sharing and address translation
- Enabler of many key aspects of operating systems
 - Protection
 - Multi-programming
 - Isolation
 - Memory resource management
 - I/O efficiency
 - Sharing
 - Inter-process communication
 - Debugging
 - Demand paging



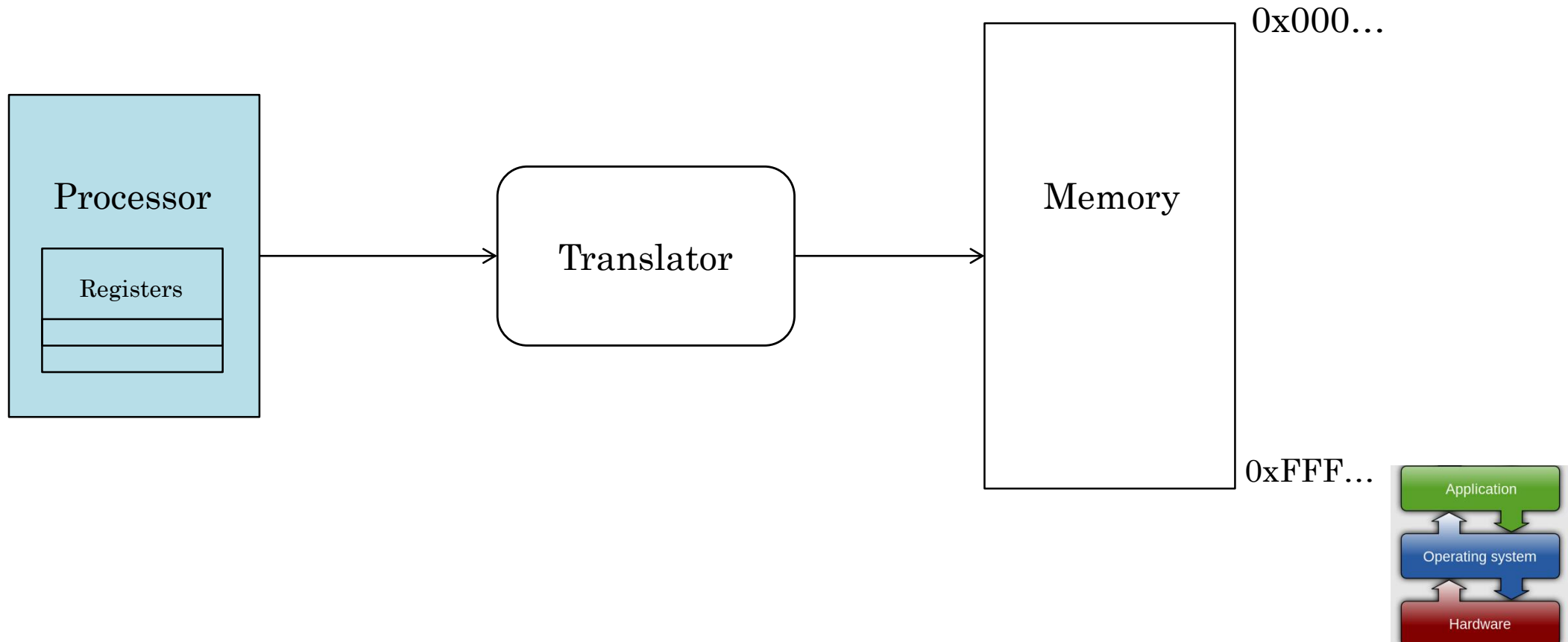
Recall: Four Fundamental OS Concepts

- **Thread**: Execution Context
 - Program Counter, Registers, Execution Flags, Stack
- **Address Space** (with Translation)
 - Program's view of memory is distinct from physical machine
- **Process**: Instance of a Running Program
 - Address space + one or more threads + ...
- **Dual-Mode Operation and Protection**
 - Only the “system” can access certain resources
 - Combined with translation, isolates programs from each other



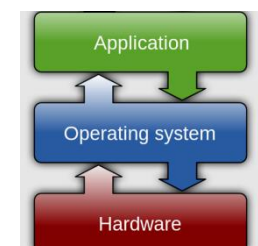
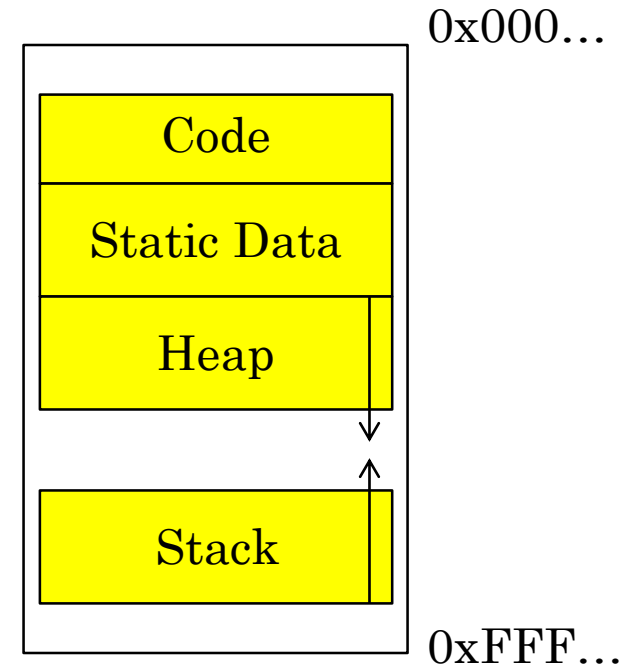
Key OS Concept: Address Space

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

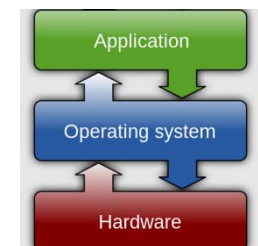
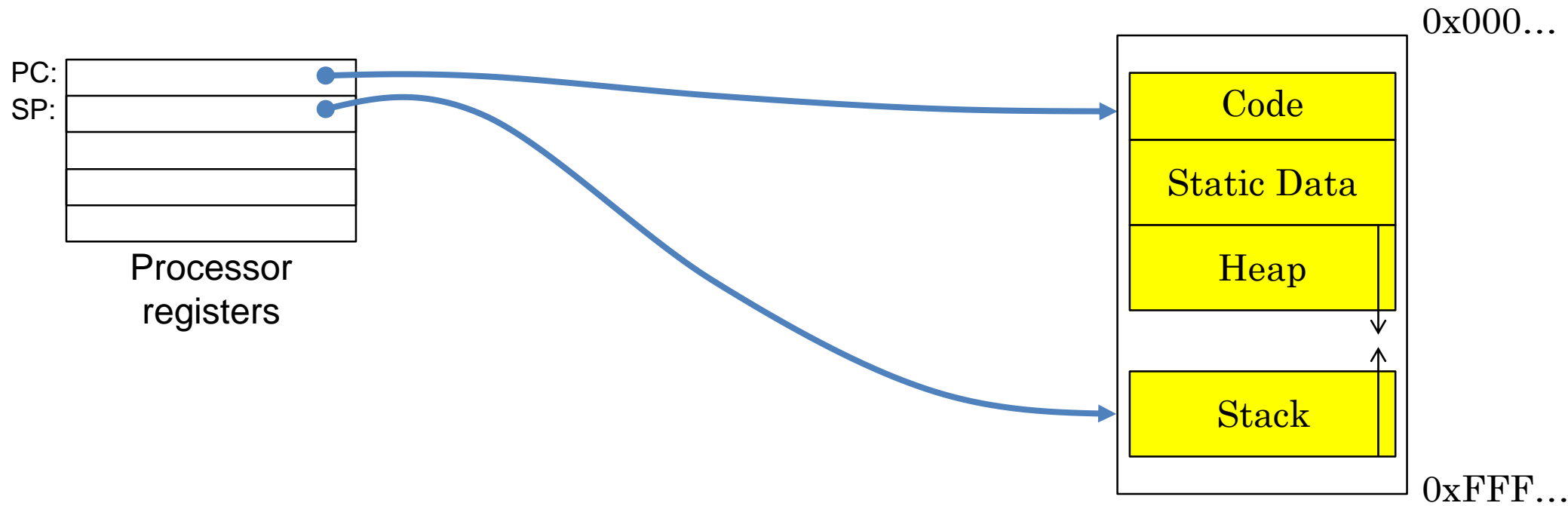


Recall: Address Space

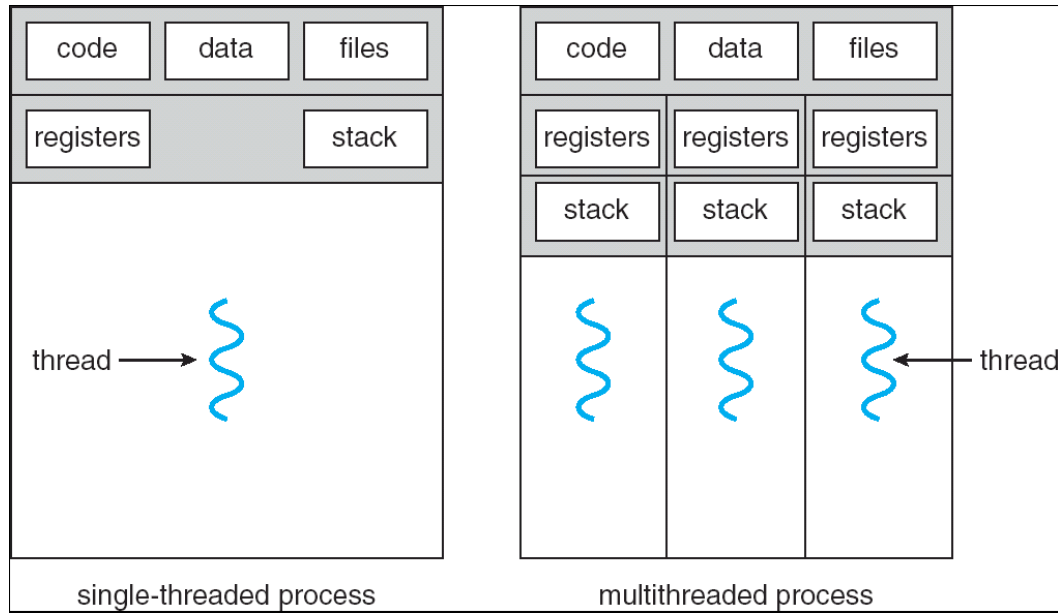
- Definition: Set of accessible addresses and the state associated with them
 - $2^{32} = \sim 4$ billion on a 32-bit machine
- What happens when you read or write to an address?
 - Perhaps acts like regular memory
 - Perhaps causes I/O operation
 - (Memory-mapped I/O)
 - Causes program to abort (segfault)?
 - Communicate with another program
 - ...



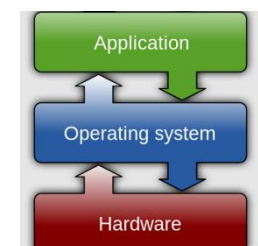
Recall: Typical Address Space Structure



Recall: Single and Multithreaded Processes

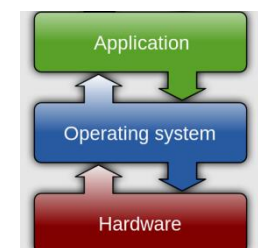


- Threads encapsulate concurrency
 - “Active” component
- Address space encapsulate protection:
 - “Passive” component
 - Keeps bugs from crashing the entire system
- Why have multiple threads per address space?



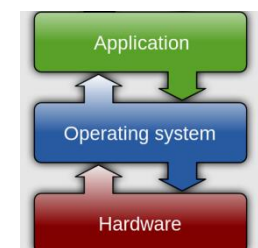
Important Aspects of Memory Multiplexing

- Protection
 - Prevent access to private memory of other process or kernel
- Translation
 - Gives uniform view of memory to programs
 - Allows for efficient “tricks”
 - E.g., in implementation of `fork()`
- Controlled Overlap
 - Read-only data, execute-only shared libraries
 - Inter-process communication



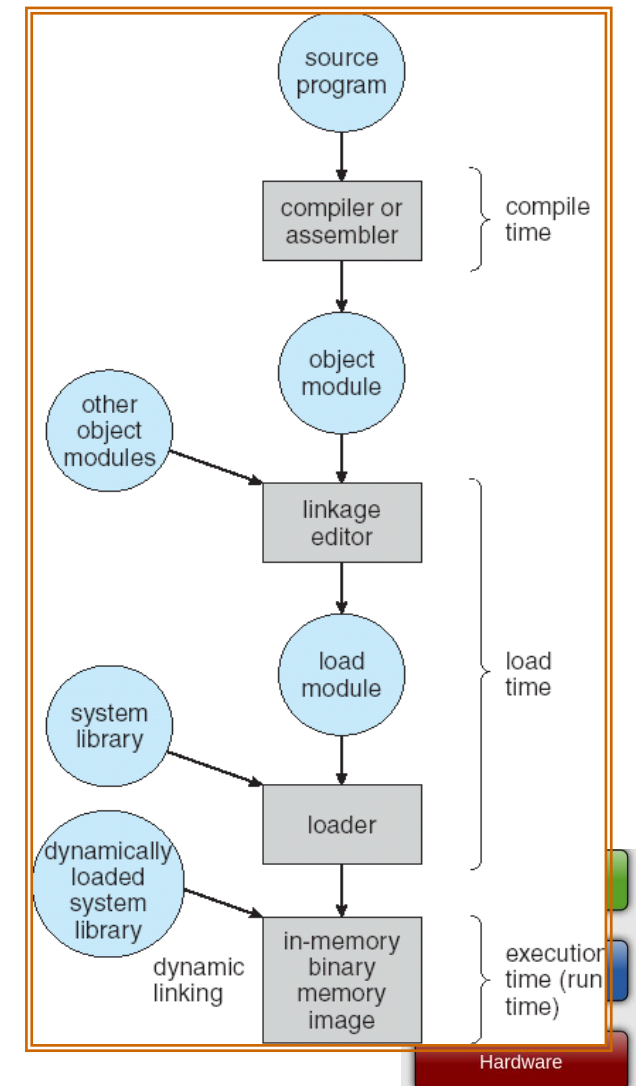
Alternative View: Interposing on Process Behavior

- OS interposes on process' I/O operations
 - How? All I/O happens via syscalls.
- OS interposes on process' CPU usage
 - How? Interrupt lets OS preempt current thread
- Question: How can the OS interpose on process' memory accesses?
 - Too slow for the OS to interpose every memory access
 - Translation: hardware support to accelerate the common case
 - Page fault: uncommon cases trap to the OS to handle



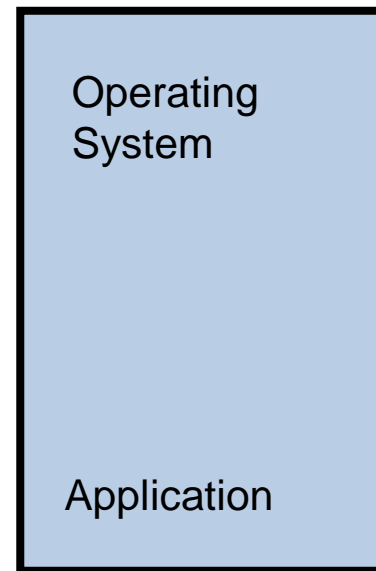
From Program to Process

- Preparation of a program for execution involves components at:
 - Compile time (i.e., “gcc”)
 - Link/Load time (UNIX “ld” does link)
 - Execution time (e.g., dynamic libs)
- Addresses can be bound to final values anywhere in this path
 - Depends on hardware support
 - Also depends on operating system
- Dynamic Libraries
 - Linking postponed until execution
 - Small piece of code (i.e. the stub), locates appropriate memory-resident library routine
 - Stub replaces itself with the address of the routine, and executes routine



Uniprogramming: One Process at a Time

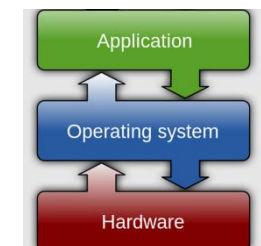
- No Translation or Protection
 - Application always runs at same place in physical memory since only one application at a time
 - Application can access any physical address
 - Application given illusion of dedicated machine by giving it reality of a dedicated machine



0xFFFFFFFF

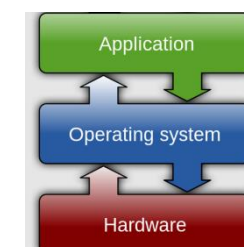
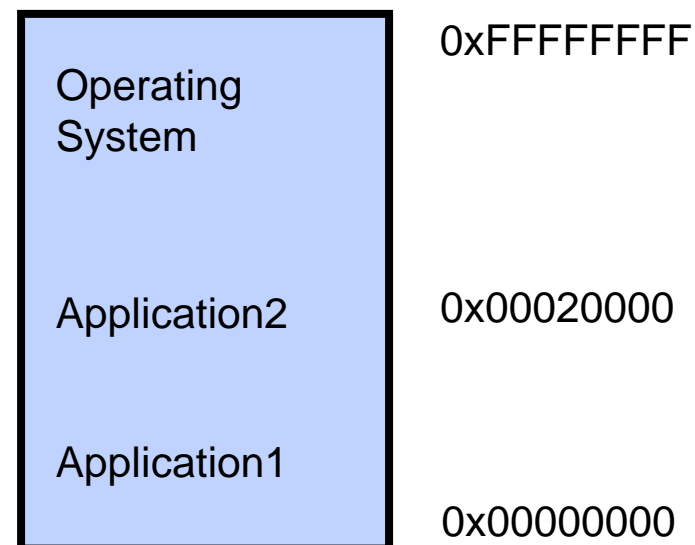
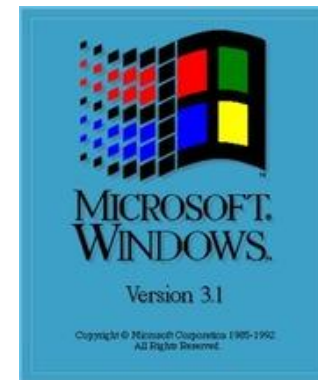
Valid 32-bit
Addresses

0x00000000



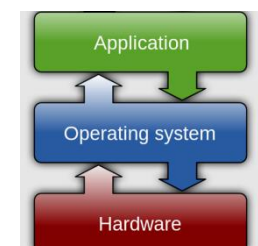
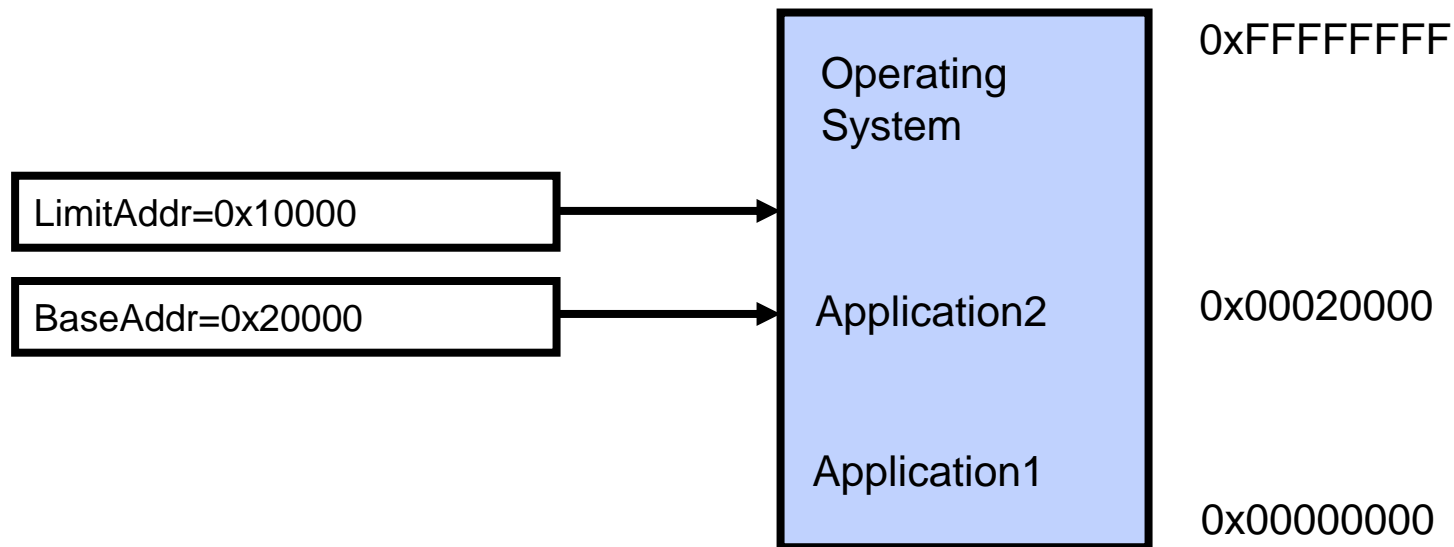
Primitive Multiprogramming

- Multiprogramming without Translation or Protection
- Use Loader/Linker: Adjust addresses while program loaded into memory (loads, stores, jumps)
 - Everything adjusted to memory location where OS put program
 - Translation done by a linker-loader (relocation)
- No protection!



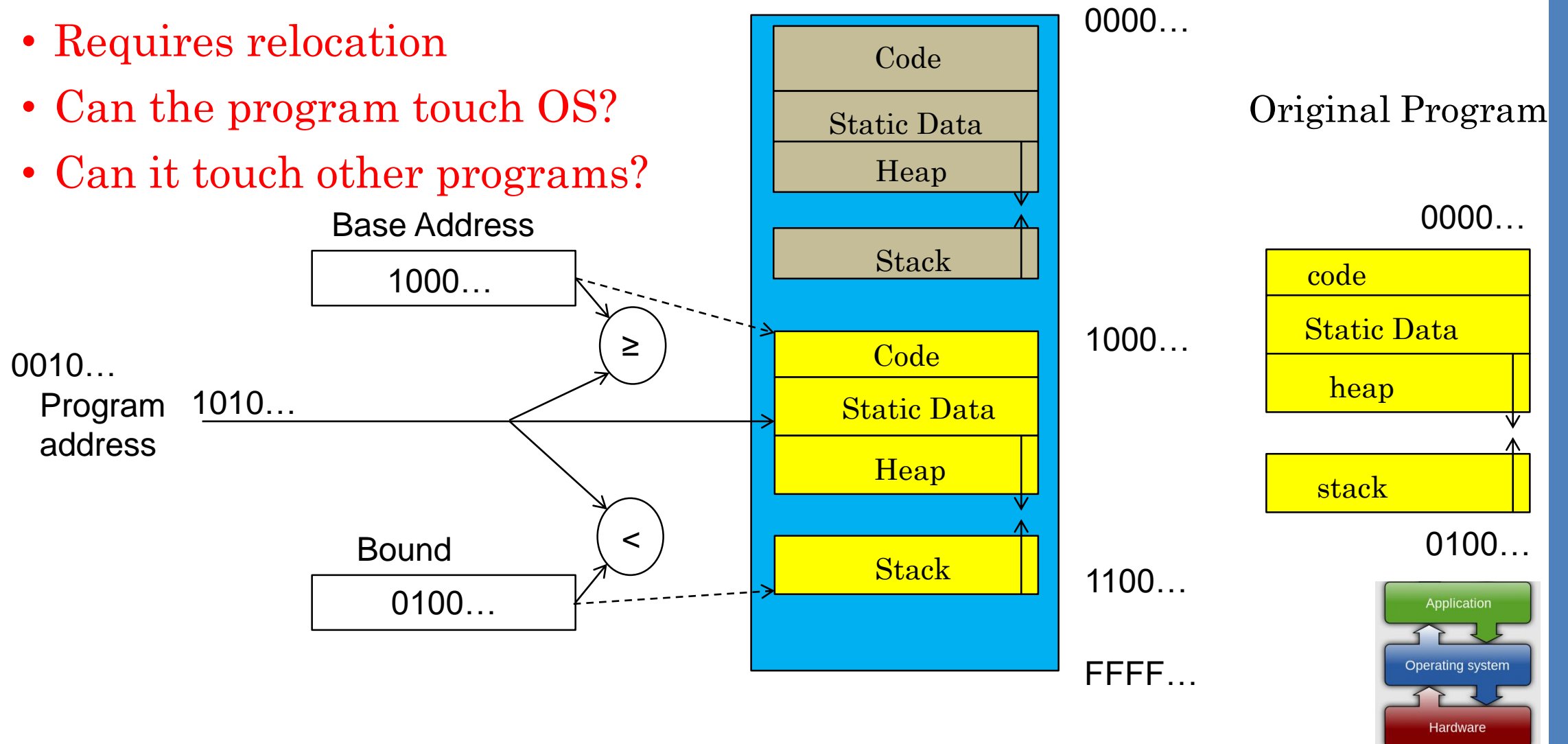
Multiprogramming with Protection

- Can we protect programs from each other without translation?
 - Yes: Base and Bound!
 - Used by, e.g., Cray-1 supercomputer

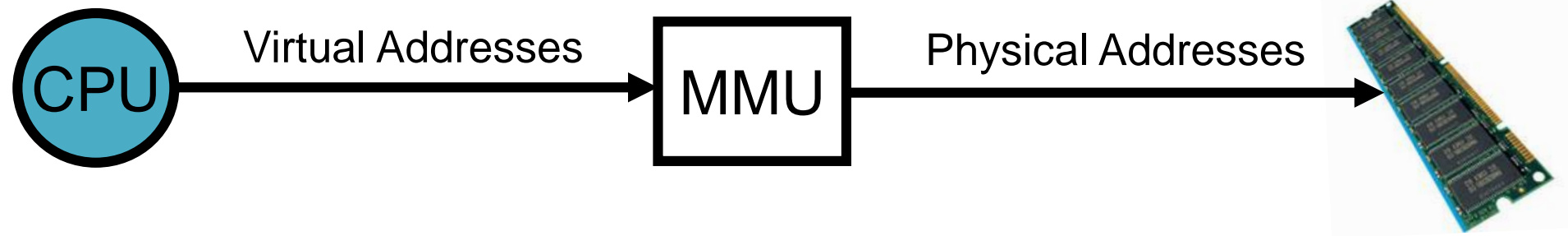


Recall: Base and Bound (no Translation)

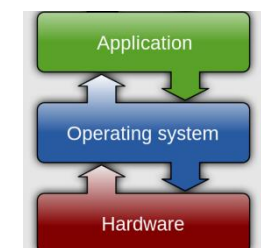
- Requires relocation
- Can the program touch OS?
- Can it touch other programs?



General Translation

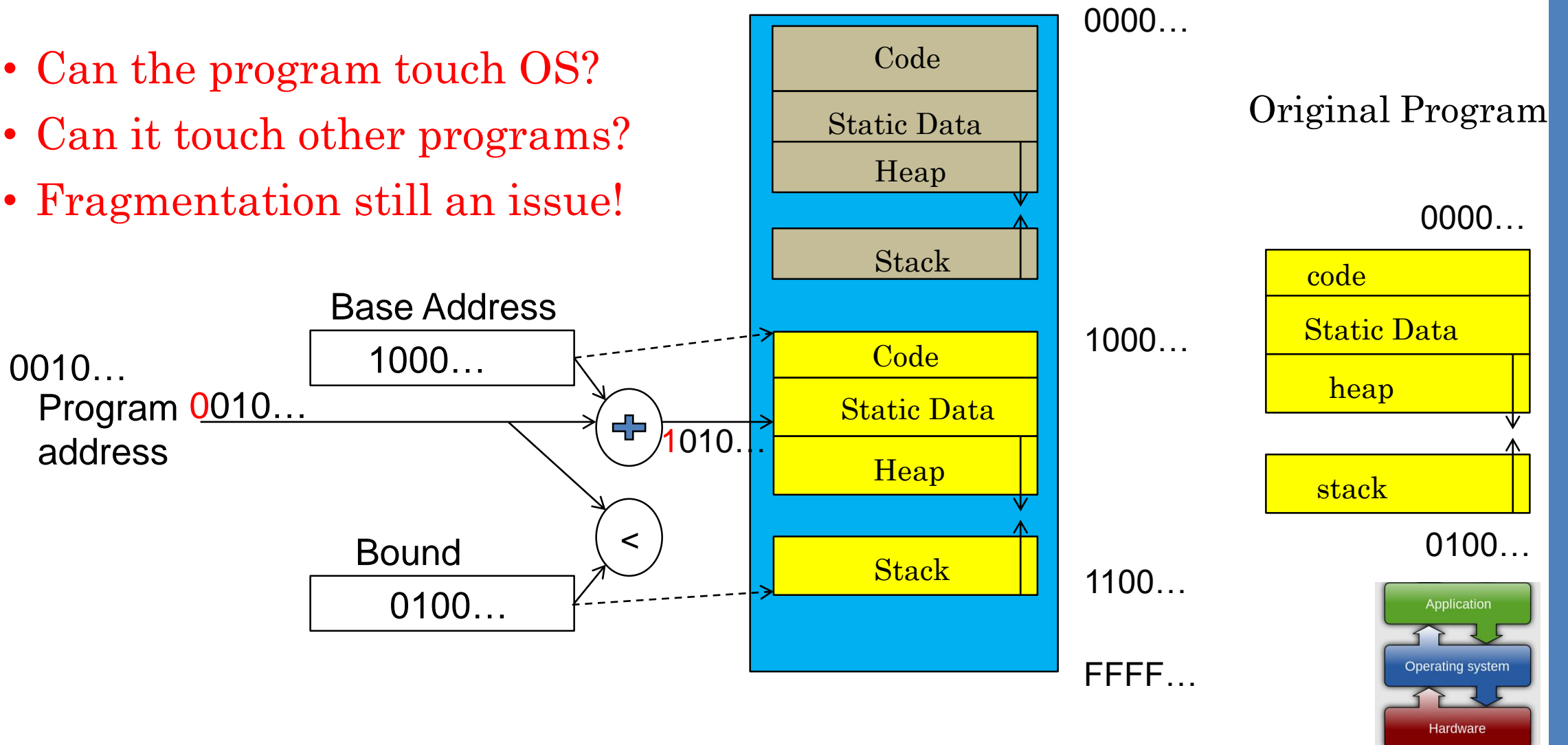


- Two views of memory:
 - View from the CPU (what program sees, virtual memory)
 - View from memory (physical memory)
 - Hardware translator (Memory Management Unit or MMU) converts between the two views
- With translation, every program can be linked/loaded into same region of user address space

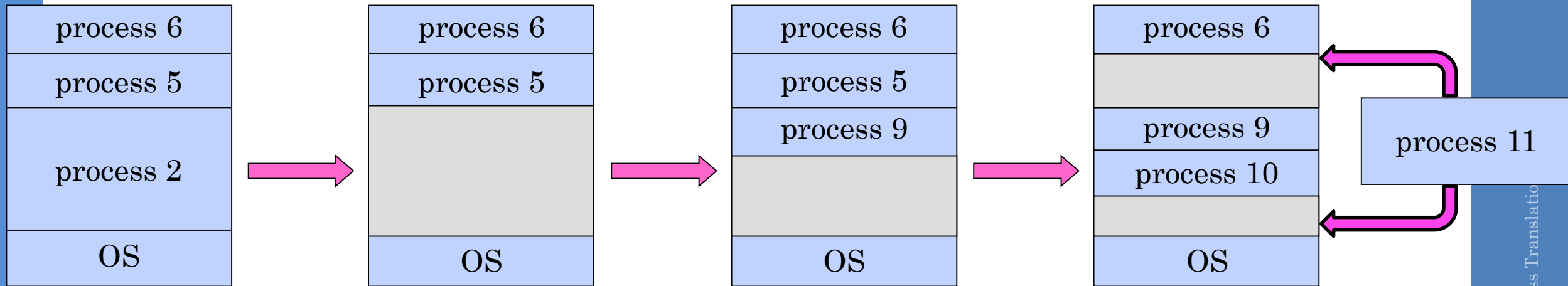


Recall: Base and Bound (with Translation)

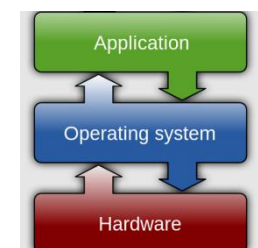
- Can the program touch OS?
- Can it touch other programs?
- Fragmentation still an issue!



Issues with Simple Base and Bound

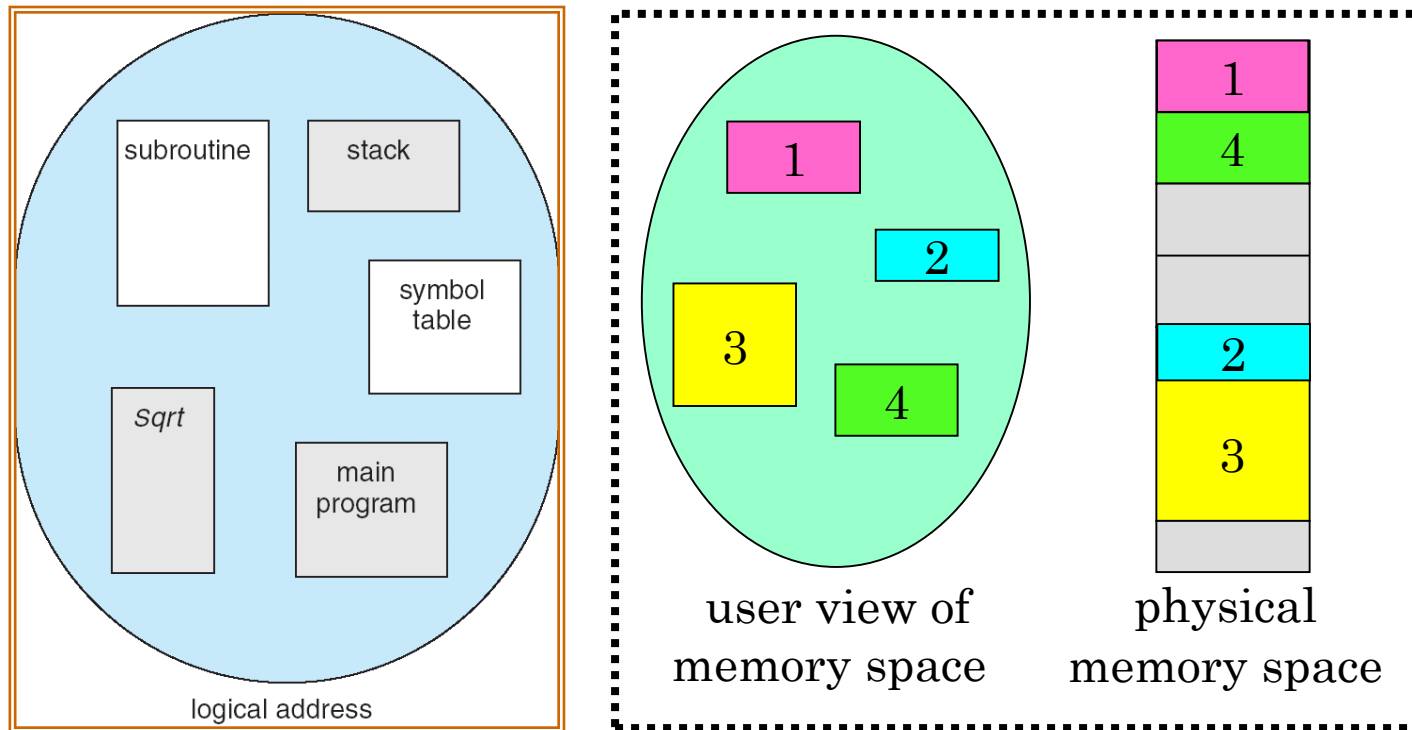


- Fragmentation problem over time
- No support for sparse address space
- Hard to do interprocess sharing
 - E.g., to share code

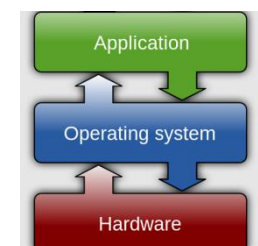


Segmentation

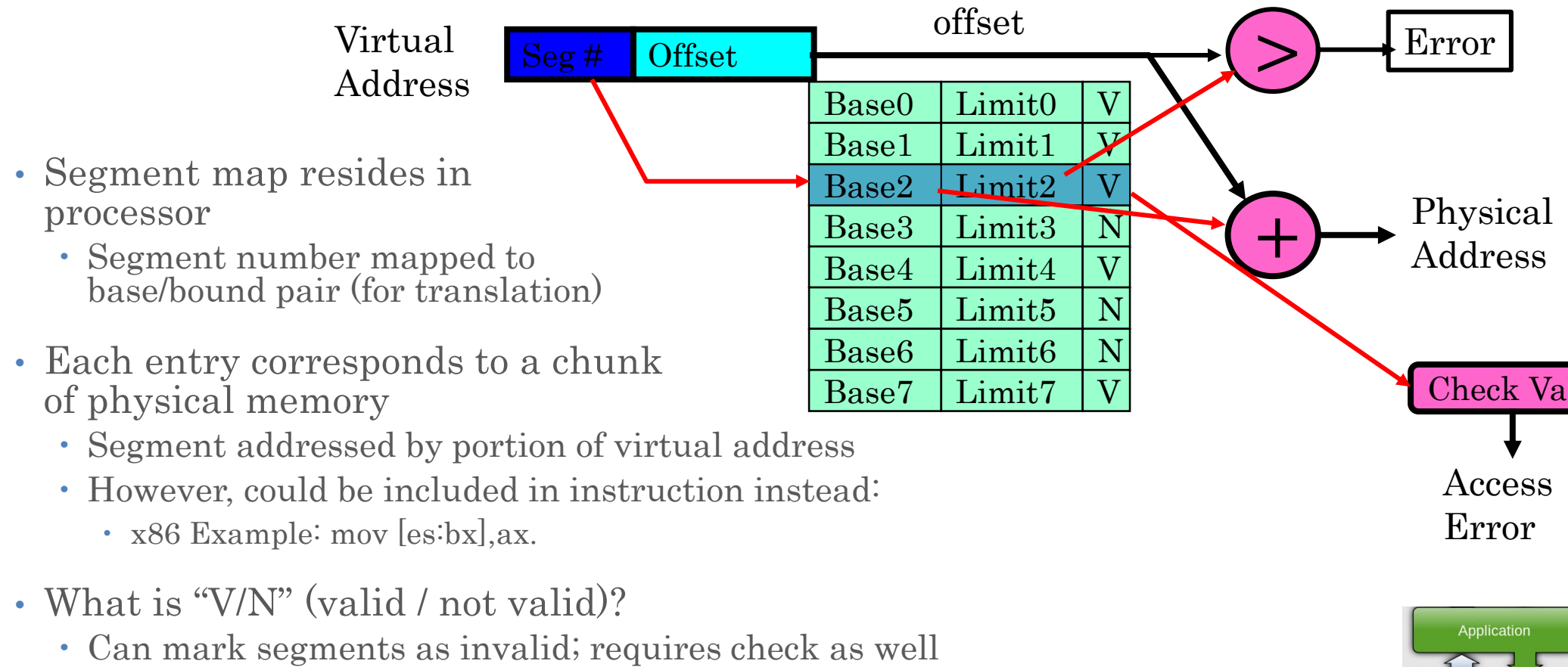
Segmentation



- Program's view of memory: multiple separate segments
- Each segment is given a region of contiguous memory
 - Has a base and limit
- Memory address consists of segment ID and offset

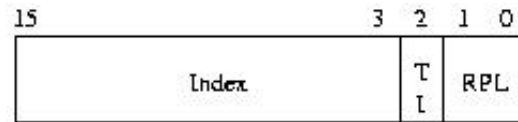


Hardware Support for Segmentation



- Segment map resides in processor
 - Segment number mapped to base/bound pair (for translation)
- Each entry corresponds to a chunk of physical memory
 - Segment addressed by portion of virtual address
 - However, could be included in instruction instead:
 - x86 Example: `mov [es:bx],ax.`
- What is “V/N” (valid / not valid)?
 - Can mark segments as invalid; requires check as well

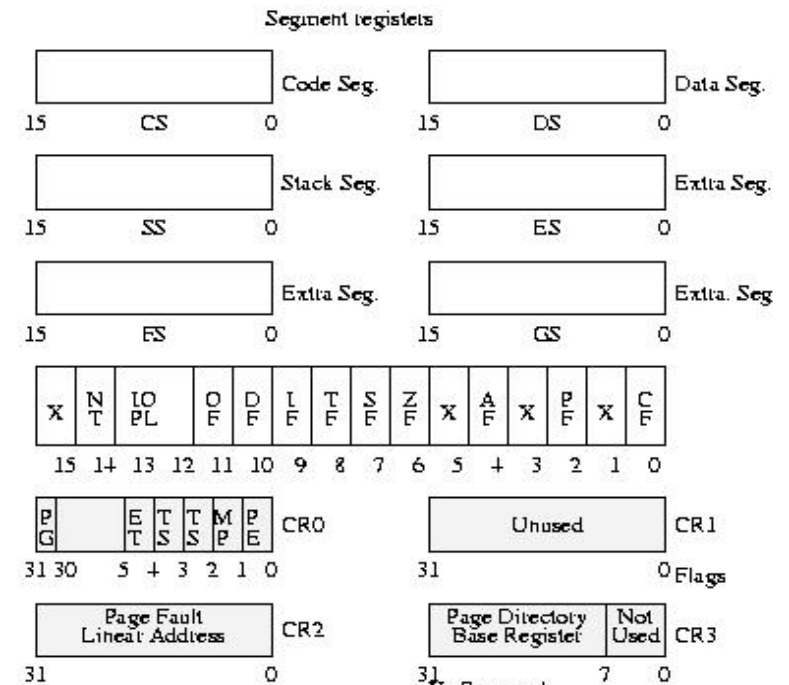
Intel x86 Special Registers



RPL = Requestor Privilege Level
 TI = Table Indicator
 (0 = GDT, 1 = LDT)
 Index = Index into table

Protected Mode segment selector

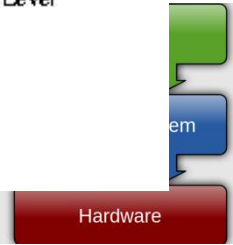
80386 Special Registers



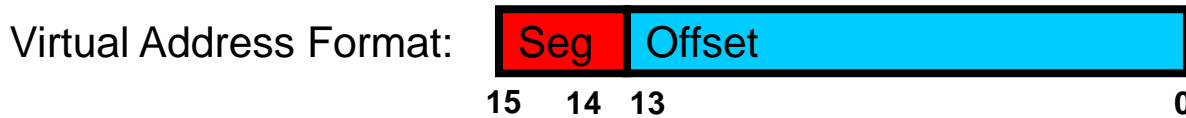
PG=Paging Enable
 ET=Emulation Type
 TS=Task Switched
 EM=Emulate Coprocessor
 MP=Math coprocessor present
 PE=Protected Mode enable

X=Reserved
 NT=Nested Task
 IOPL=IO Privilege Level
 OF=Overflow Flag
 DF=Direction Flag
 IF=Interrupt Flag
 TF=Trap Flag
 SF=Sign Flag
 ZF=Zero Flag
 AF=Auxiliary Flag
 PF=Parity Flag
 CF=Carry Flag

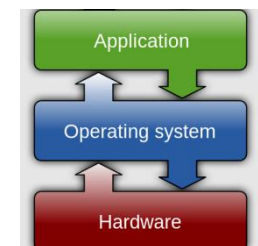
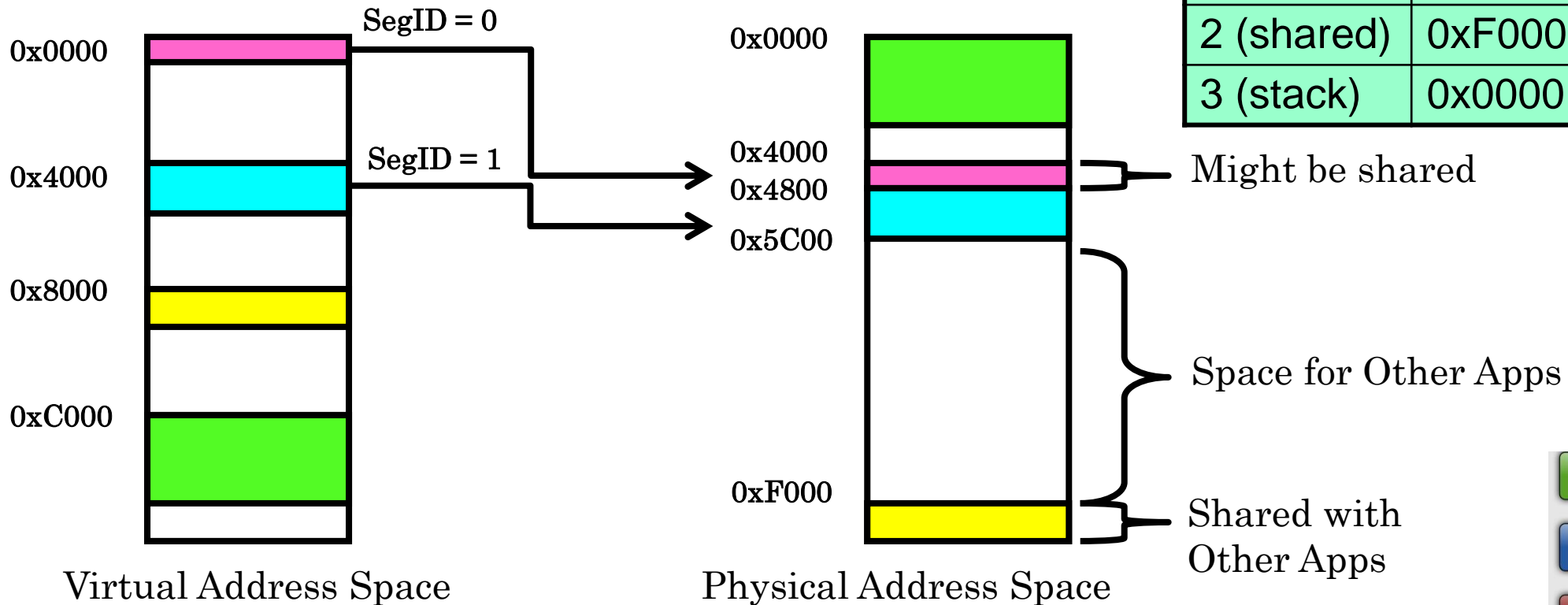
- Segmentation can't be just "turned off"
- What if we just want to use paging?
 - Set base and bound to all of memory, in all segments



Example: Four Segments

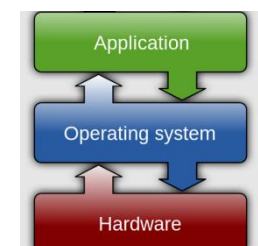


Seg ID #	Base	Limit
0 (code)	0x4000	0x0800
1 (data)	0x4800	0x1400
2 (shared)	0xF000	0x1000
3 (stack)	0x0000	0x3000

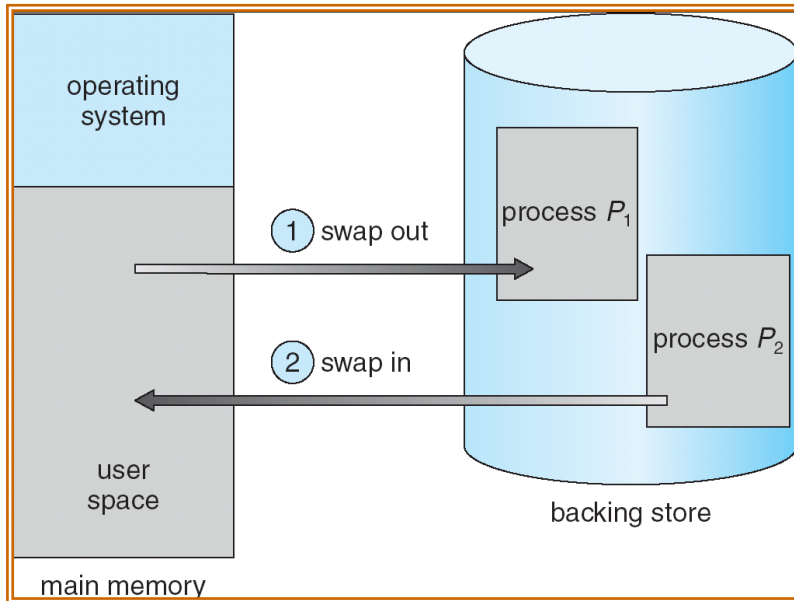


Observations about Segmentation

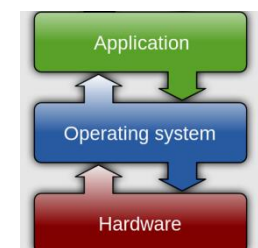
- Translation on every instruction fetch, load or store
- Virtual address space has holes
 - Segmentation efficient for sparse address spaces
- When it is OK to address outside valid range?
 - This is how the stack (and heap?) is allowed to grow
 - For instance, stack takes fault, system automatically increases size of stack
- Need protection mode in segment table
 - For example, code segment would be read-only
 - Data and stack would be read-write (stores allowed)
- What must be saved/restored on context switch?
 - Segment table stored in CPU, not in memory (small)
 - Might store all of processes memory onto disk when switched (called “swapping”)



What if not all segments fit in memory?



- Extreme form of Context Switch: Swapping
 - In order to make room for next process, some or all of the previous process is moved to disk
 - Likely need to send out complete segments
 - This greatly increases the cost of context-switching
- What might be a desirable alternative?
 - Some way to keep only active portions of a process in memory at any one time
 - Need finer granularity control over physical memory



Problems with Segmentation

- Must fit variable-sized chunks into physical memory
- May move processes multiple times to fit everything
- Limited options for swapping to disk
- Fragmentation: wasted space
 - External: free gaps between allocated chunks
 - Internal: don't need all memory within allocated chunks

