Protection: Address Spaces

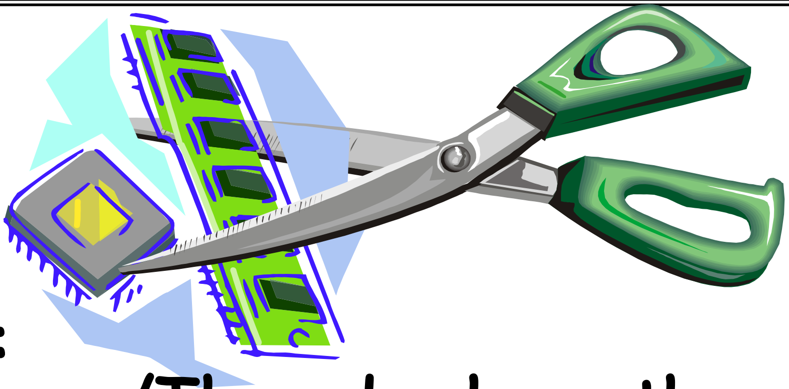
http://inst.eecs.berkeley.edu/~cs162

Goals for Today

- Kernel vs User Mode
- What is an Address Space?
- How is it Implemented?
- Discussion of Dual-Mode operation
- Comparison among options

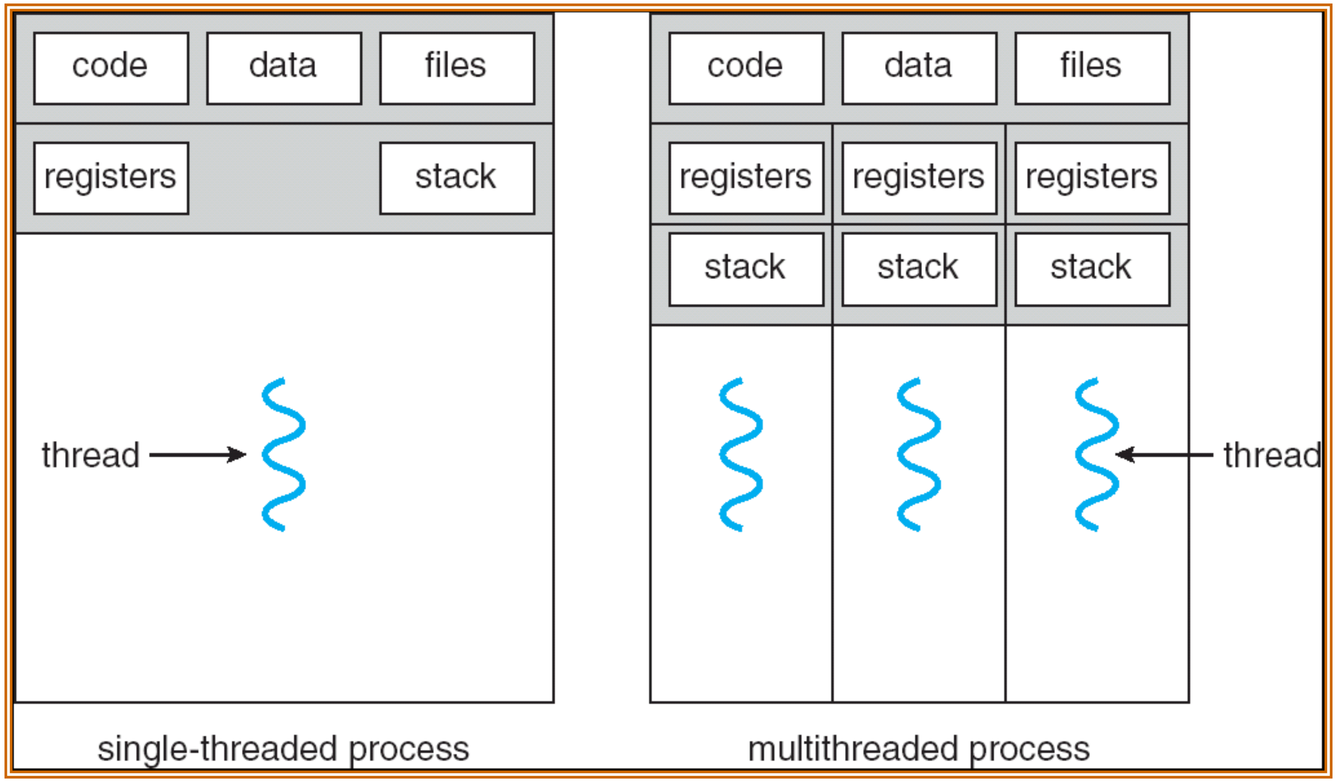
Note: Some slides and/or pictures in the following are adapted from slides ©2005 Silberschatz, Galvin, and Gagne

Virtualizing Resources



- Physical Reality:
  Different Processes/Threads share the same hardware
  - Need to multiplex CPU (Just finished: scheduling)
  - Need to multiplex use of Memory (Today)
  - Need to multiplex disk and devices (later in term)
- Why worry about memory sharing?
  - The complete working state of a process and/or kernel is defined by its data in memory (and registers)
  - Consequently, cannot just let different threads of control use the same memory
    - » Physics: two different pieces of data cannot occupy the same locations in memory
  - Probably don't want different threads to even have access to each other's memory (protection)

Recall: Single and Multithreaded Processes



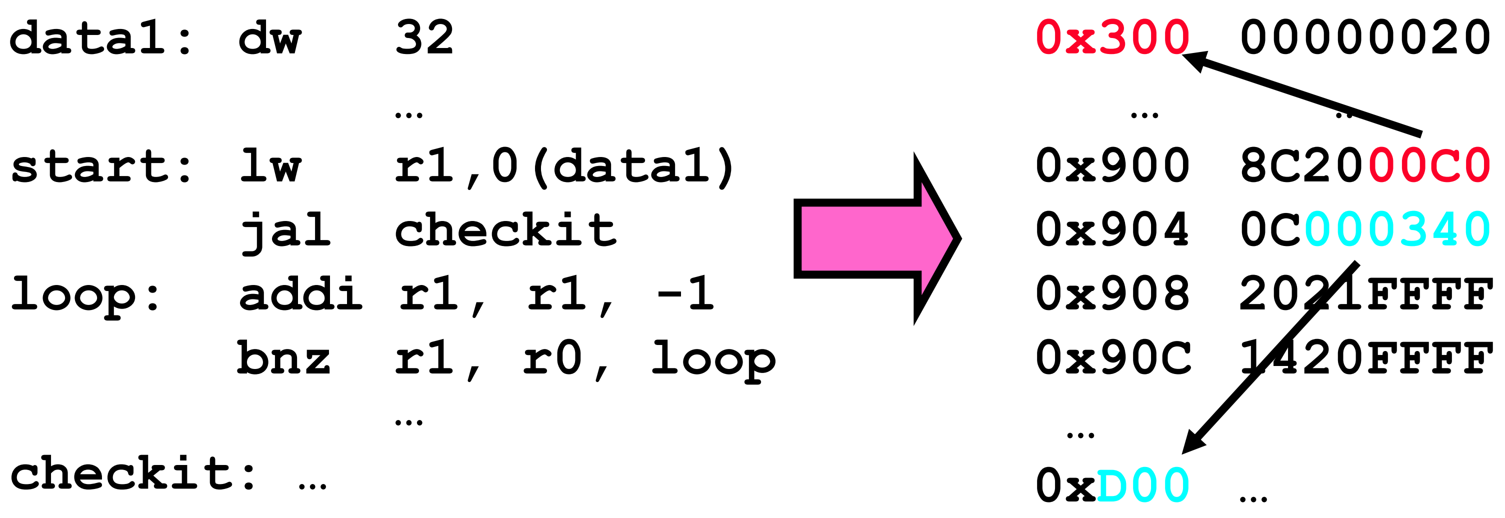
- Threads encapsulate concurrency
  - "Active" component of a process
- Address spaces encapsulate protection
  - Keeps buggy program from trashing the system
  - "Passive" component of a process

Important Aspects of Memory Multiplexing

- Controlled overlap:
  - Separate state of threads should not collide in physical memory. Obviously, unexpected overlap causes chaos!
  - Conversely, would like the ability to overlap when desired (for communication)
- Translation:
  - Ability to translate accesses from one address space (virtual) to a different one (physical)
  - When translation exists, processor uses virtual addresses, physical memory uses physical addresses
  - Side effects:
    - » Can be used to avoid overlap
    - » Can be used to give uniform view of memory to programs
- Protection:
  - Prevent access to private memory of other processes
    - » Different pages of memory can be given special behavior (Read Only, Invisible to user programs, etc).
    - » Kernel data protected from User programs
    - » Programs protected from themselves

Binding of Instructions and Data to Memory

- Binding of instructions and data to addresses:
  - Choose addresses for instructions and data from the standpoint of the processor



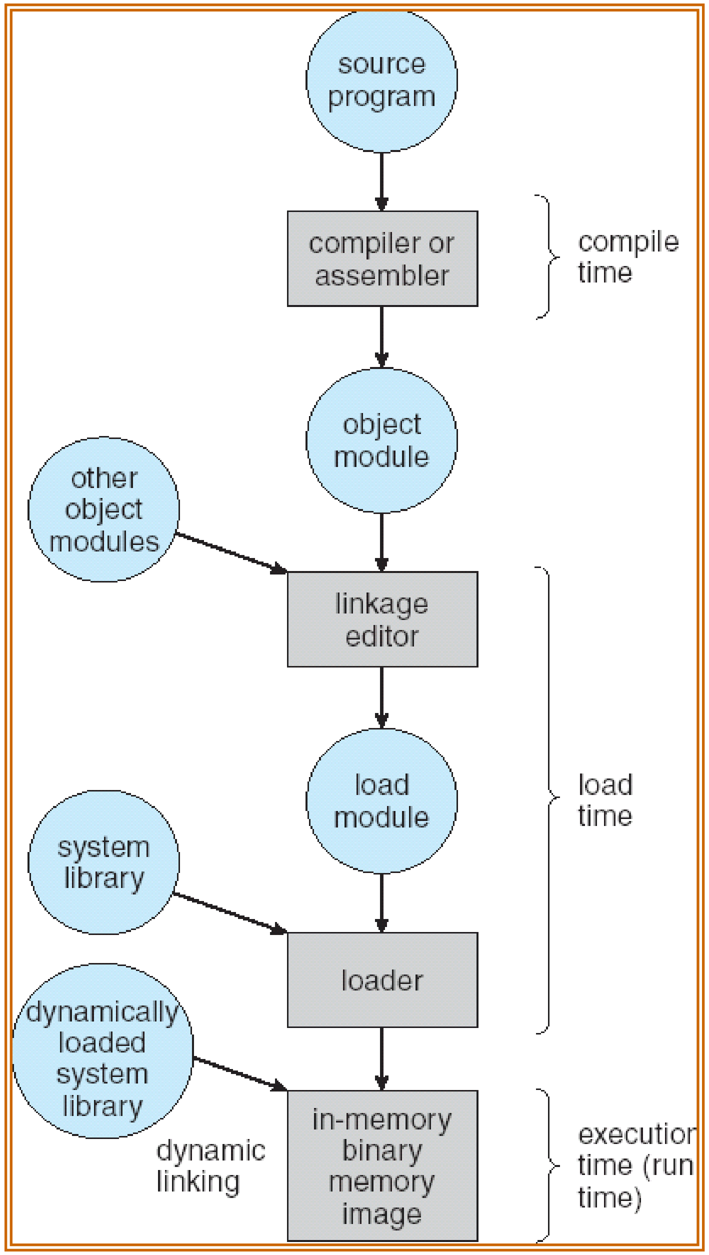
```
data1:  dw    32
              …
start:  lw    r1,0(data1)
        jal   checkit
loop:   addi  r1, r1, -1
        bnz   r1, r0, loop
              …
checkit: …
```

```
0x300  00000020
   …
0x900  8C2000C0
0x904  0C000340
0x908  2021FFFF
0x90C  1420FFFF
 …
0xD00  …
```

  - Could we place `data1`, `start`, and/or `checkit` at different addresses?
    - » Yes
    - » When? Compile time/Load time/Execution time
  - Related: which physical memory locations hold particular instructions or data?

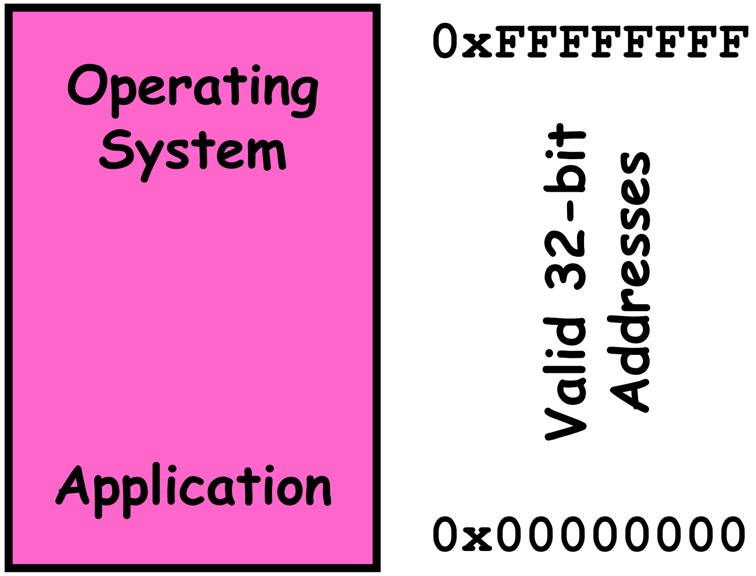
Multi-step Processing of a Program for Execution

- 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, *stub*, used to locate the appropriate memory-resident library routine
  - Stub replaces itself with the address of the routine, and executes routine



Recall: Uniprogramming

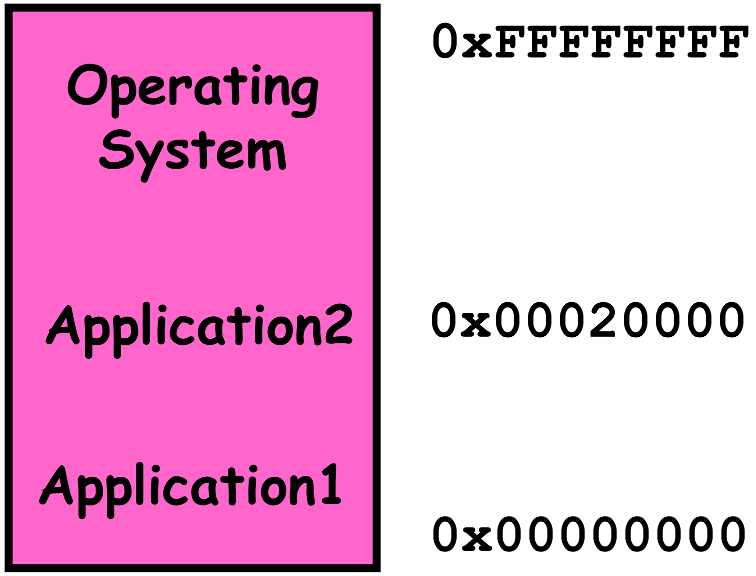
- Uniprogramming (no Translation or Protection)
  - Application always runs at same place in physical memory since only one application at a time
  - Application can access any physical address



  - Application given illusion of dedicated machine by giving it reality of a dedicated machine
- Of course, this doesn't help us with multithreading

Multiprogramming (First Version)

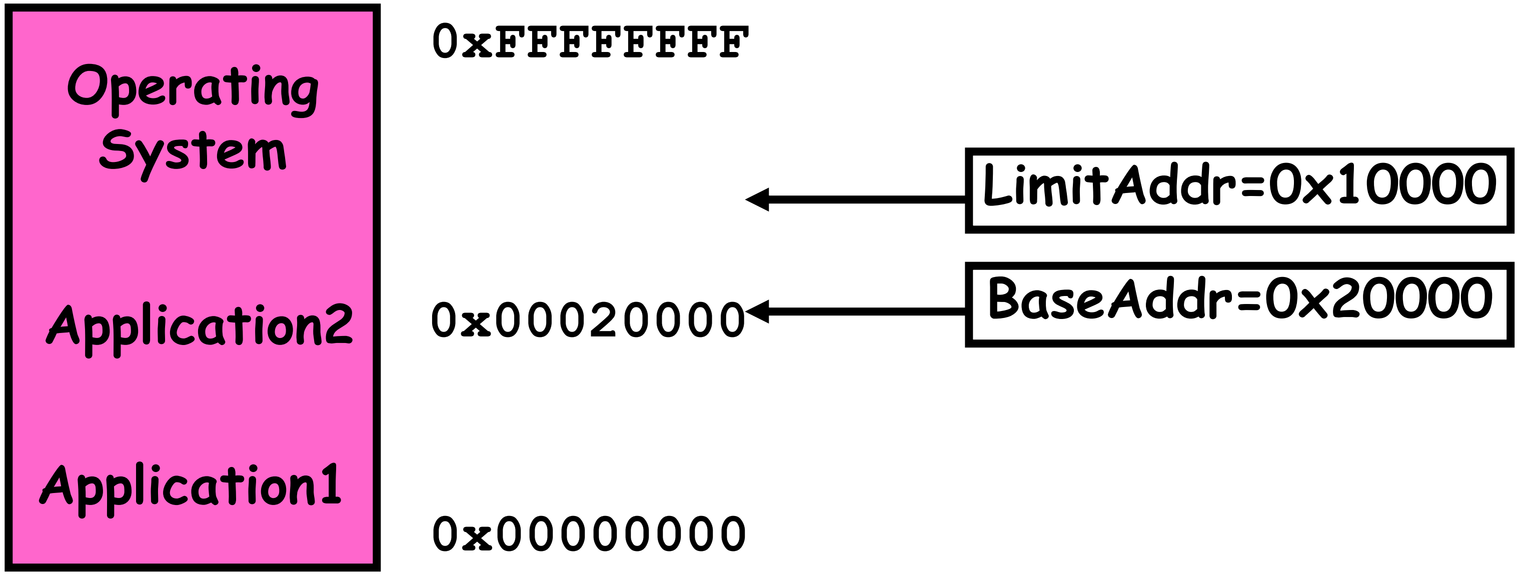
- Multiprogramming without Translation or Protection
  - Must somehow prevent address overlap between threads



  - Trick: Use Loader/Linker: Adjust addresses while program loaded into memory (loads, stores, jumps)
    - » Everything adjusted to memory location of program
    - » Translation done by a linker-loader
    - » Was pretty common in early days
- With this solution, no protection: bugs in any program can cause other programs to crash or even the OS

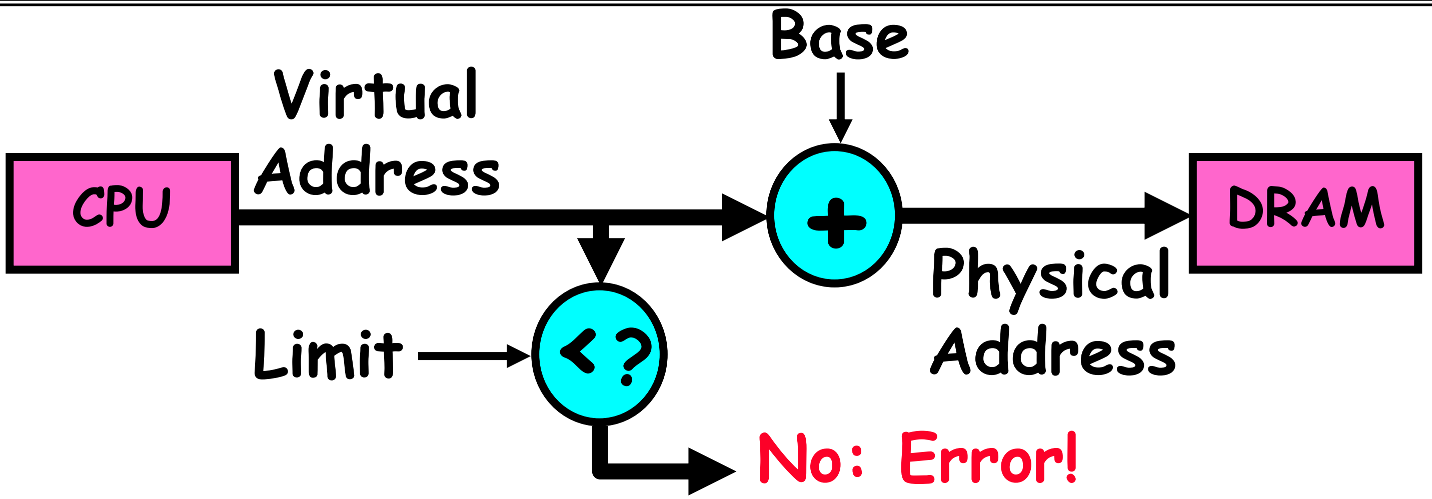
Multiprogramming (Version with Protection)

- Can we protect programs from each other without translation?



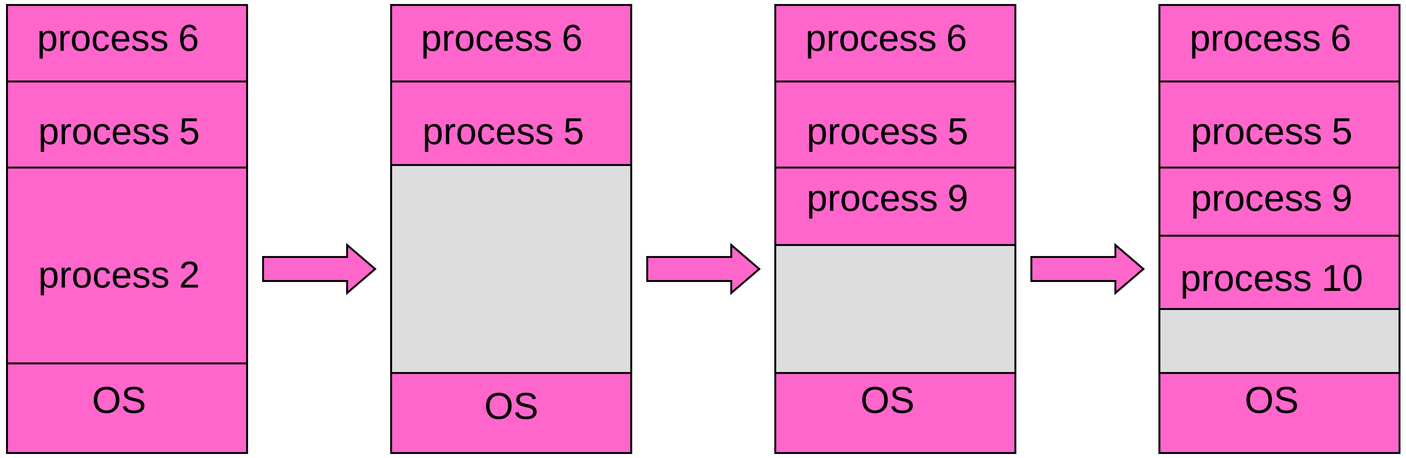
  - Yes: use two special registers *BaseAddr* and *LimitAddr* to prevent user from straying outside designated area
    - » If user tries to access an illegal address, cause an error
  - During switch, kernel loads new base/limit from TCB
    - » User not allowed to change base/limit registers

Segmentation with Base and Limit registers



- Could use base/limit for dynamic address translation (often called "segmentation"):
  - Alter address of every load/store by adding "base"
  - User allowed to read/write within segment
    - » Accesses are relative to segment so don't have to be relocated when program moved to different segment
  - User may have multiple segments available (e.g x86)
    - » Loads and stores include segment ID in opcode:
      x86 Example: `mov [es:bx],ax.`
    - » Operating system moves around segment base pointers as necessary

Issues with simple segmentation method

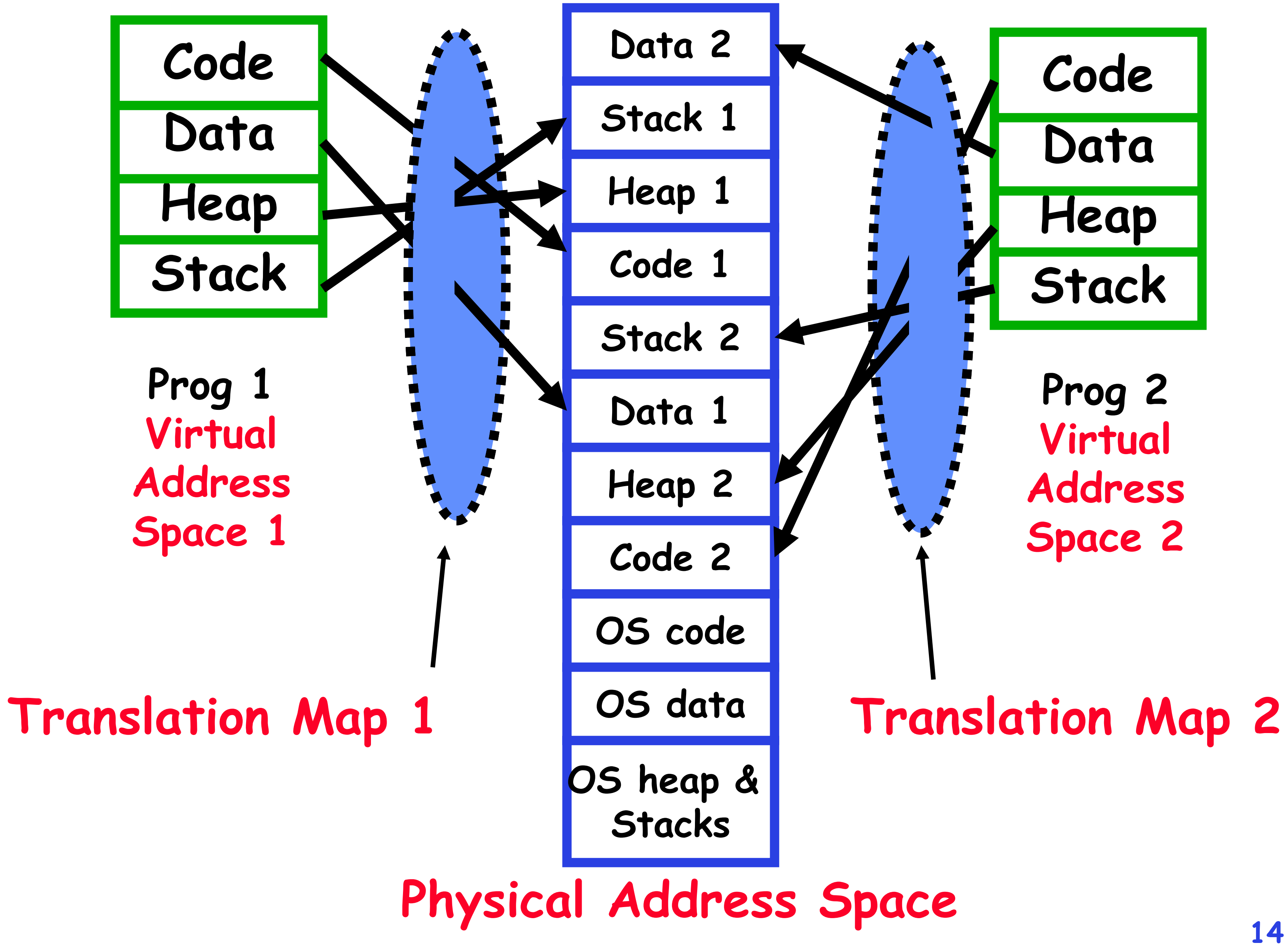


- Fragmentation problem
  - Not every process is the same size
  - Over time, memory space becomes fragmented
- Hard to do inter-process sharing
  - Want to share code segments when possible
  - Want to share memory between processes
  - Helped by by providing multiple segments per process
- Need enough physical memory for every process

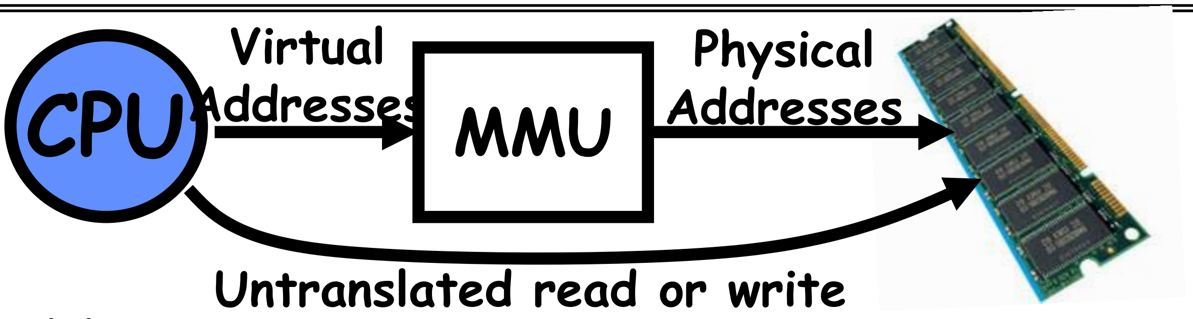
Multiprogramming (Translation and Protection version 2)

- Problem: Run multiple applications in such a way that they are protected from one another
- Goals:
  - Isolate processes and kernel from one another
  - Allow flexible translation that:
    - » Doesn't lead to fragmentation
    - » Allows easy sharing between processes
    - » Allows only part of process to be resident in physical memory
- (Some of the required) Hardware Mechanisms:
  - General Address Translation
    - » Flexible: Can fit physical chunks of memory into arbitrary places in users address space
    - » Not limited to small number of segments
    - » Think of this as providing a large number (thousands) of fixed-sized segments (called "pages")
  - Dual Mode Operation
    - » Protection base involving kernel/user distinction

Example of General Address Translation

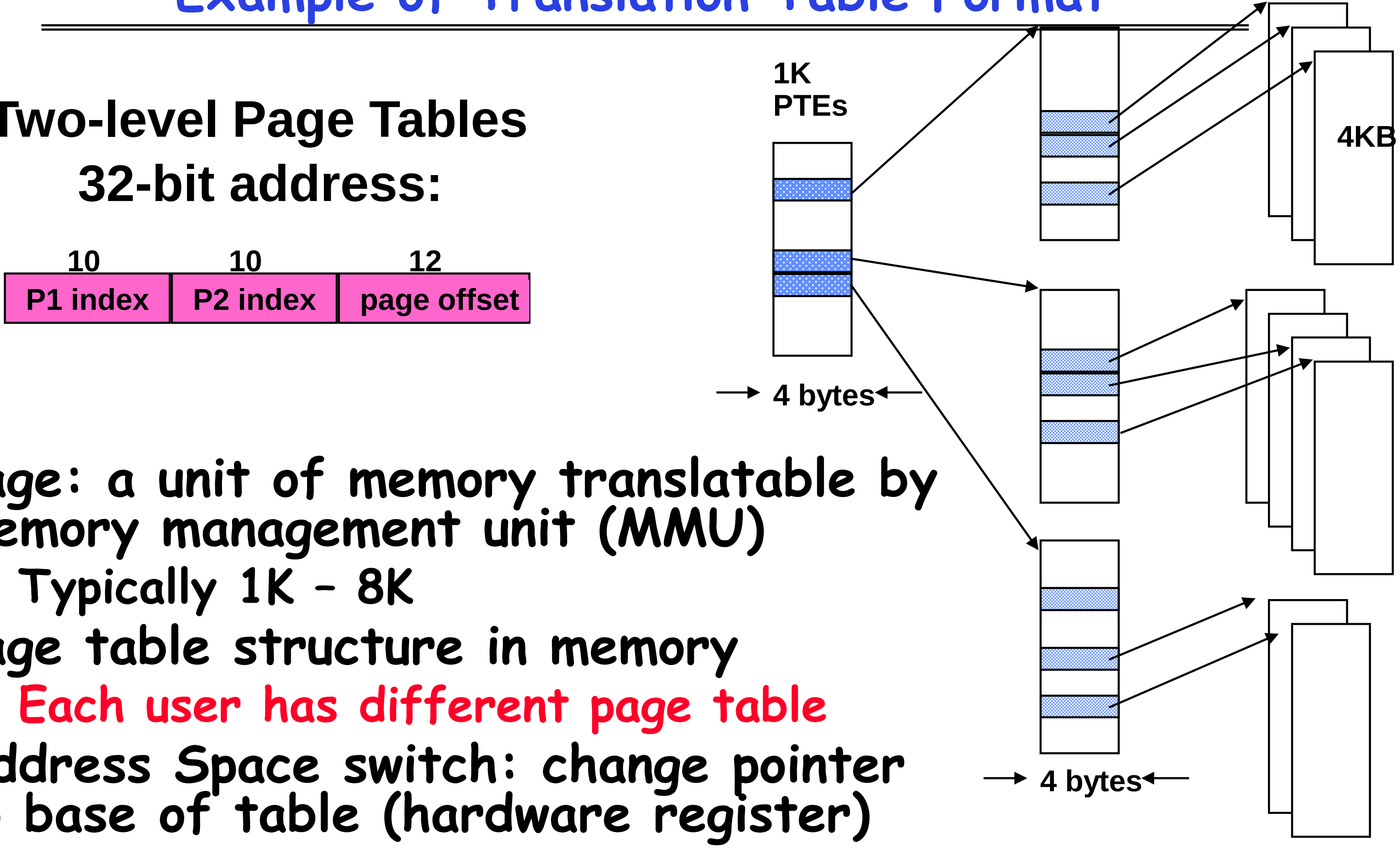


Two Views of Memory



- Recall: Address Space:
  - All the addresses and state a process can touch
  - Each process and kernel has different address space
- Consequently: two views of memory:
  - View from the CPU (what program sees, virtual memory)
  - View fom memory (physical memory)
  - Translation box converts between the two views
- Translation helps to implement protection
  - If task A cannot even gain access to task B's data, no way for A to adversely affect B
- With translation, every program can be linked/loaded into same region of user address space
  - Overlap avoided through translation, not relocation

Example of Translation Table Format



- Page: a unit of memory translatable by memory management unit (MMU)
  - Typically 1K – 8K
- Page table structure in memory
  - Each user has different page table
- Address Space switch: change pointer to base of table (hardware register)
  - Hardware traverses page table (for many architectures)
  - MIPS uses software to traverse table

Dual-Mode Operation

- Can Application Modify its own translation tables?
  - If it could, could get access to all of physical memory
  - Has to be restricted somehow
- To Assist with Protection, Hardware provides at least two modes (Dual-Mode Operation):
  - "Kernel" mode (or "supervisor" or "protected")
  - "User" mode (Normal program mode)
  - Mode set with bits in special control register only accessible in kernel-mode
- Intel processor actually has four "rings" of protection:
  - PL (Priviledge Level) from 0 – 3
    - » PL0 has full access, PL3 has least
  - Privilege Level set in code segment descriptor (CS)
  - Mirrored "IOPL" bits in condition register gives permission to programs to use the I/O instructions
  - Typical OS kernels on Intel processors only use PL0 ("user") and PL3 ("kernel")

For Protection, Lock User-Programs in Asylum

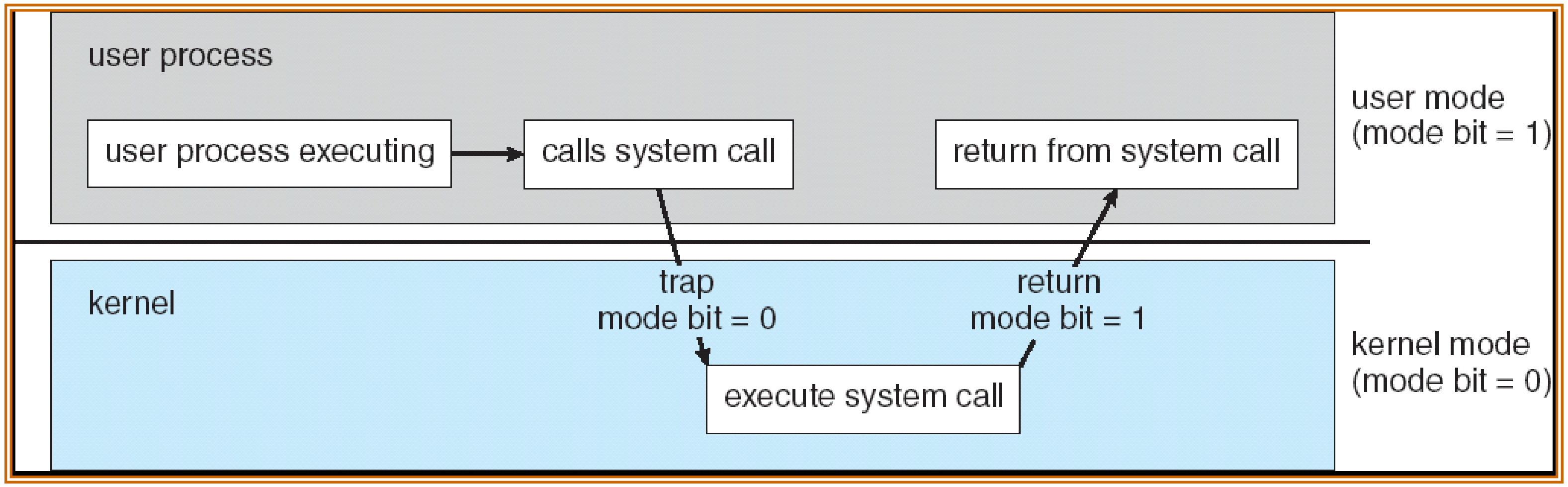
- Idea: Lock user programs in padded cell with no exit or sharp objects
  - Cannot change mode to kernel mode
  - User cannot modify page table mapping
  - Limited access to memory: cannot adversely effect other processes
    - » Side-effect: Limited access to memory-mapped I/O operations (I/O that occurs by reading/writing memory locations)
  - Limited access to interrupt controller
  - What else needs to be protected?
- A couple of issues
  - How to share CPU between kernel and user programs?
    - » Kinda like both the inmates and the warden in asylum are the same person. How do you manage this???
  - How do programs interact?
  - How does one switch between kernel and user modes?
    - » OS → user (kernel → user mode): *getting into cell*
    - » User→ OS (user → kernel mode): *getting out of cell*

How to get from Kernel→User

- What does the kernel do to create a new user process?
  - Allocate and initialize address-space control block
  - Read program off disk and store in memory
  - Allocate and initialize translation table
    - » Point at code in memory so program can execute
    - » Possibly point at statically initialized data
  - Run Program:
    - » Set machine registers
    - » Set hardware pointer to translation table
    - » Set processor status word for user mode
    - » Jump to start of program
- How does kernel switch between processes?
  - Same saving/restoring of registers as before
  - Save/restore PSL (hardware pointer to translation table)

User→Kernel (System Call)

- Can't let inmate (user) get out of padded cell on own
  - Would defeat purpose of protection!
  - So, how does the user program get back into kernel?



- System call: Voluntary procedure call into kernel
  - Hardware for controlled User→Kernel transition
  - Can any kernel routine be called?
    - » No! Only specific ones.
  - System call ID encoded into system call instruction
    - » Index forces well-defined interface with kernel

System Call Continued

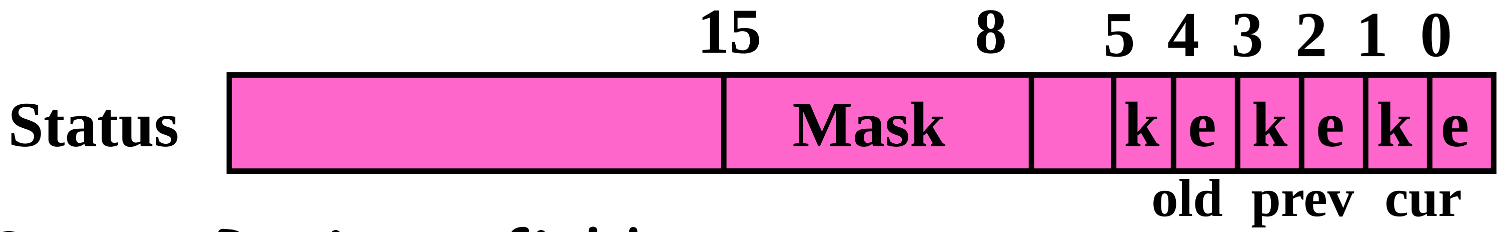
- What are some system calls?
  - I/O: open, close, read, write, lseek
  - Files: delete, mkdir, rmdir, truncate, chown, chgrp, ..
  - Process: fork, exit, wait (like join)
  - Network: socket create, set options
- Are system calls constant across operating systems?
  - Not entirely, but there are lots of commonalities
  - Also some standardization attempts (POSIX)
- What happens at beginning of system call?
    - » On entry to kernel, sets system to kernel mode
    - » Handler address fetched from table/Handler started
- System Call argument passing:
  - In registers (not very much can be passed)
  - Write into user memory, kernel copies into kernel mem
    - » User addresses must be translated!w
    - » Kernel has different view of memory than user
  - Every Argument must be explicitly checked!

User→Kernel (Exceptions: Traps and Interrupts)

- A system call instruction causes a synchronous exception (or "trap")
  - In fact, often called a software "trap" instruction
- Other sources of *Synchronous Exceptions:*
  - Divide by zero, Illegal instruction, Bus error (bad address, e.g. unaligned access)
  - Segmentation Fault (address out of range)
  - Page Fault (for illusion of infinite-sized memory)
- Interrupts are *Asynchronous Exceptions*
  - Examples: timer, disk ready, network, etc….
  - Interrupts can be disabled, traps cannot!
- On system call, exception, or interrupt:
  - Hardware enters kernel mode with interrupts disabled
  - Saves PC, then jumps to appropriate handler in kernel
  - For some processors (x86), processor also saves registers, changes stack, etc.
- Actual handler typically saves registers, other CPU state, and switches to kernel stack

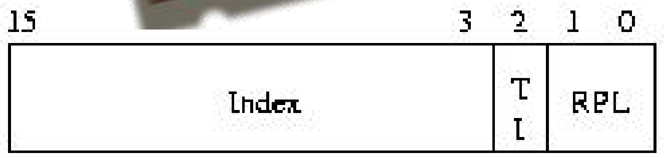
Additions to MIPS ISA to support Exceptions?

- Exception state is kept in “Coprocessor 0”
  - Use mfc0 read contents of these registers:
    - » BadVAddr (register 8): contains memory address at which memory reference error occurred
    - » Status (register 12): interrupt mask and enable bits
    - » Cause (register 13): the cause of the exception
    - » EPC (register 14): address of the affected instruction



- Status Register fields:
  - Mask: Interrupt enable
    - » 1 bit for each of 5 hardware and 3 software interrupts
  - k = kernel/user: 0⇒kernel mode
  - e = interrupt enable: 0⇒interrupts disabled
  - Exception⇒6 LSB shifted left 2 bits, setting 2 LSB to 0:
    - » run in kernel mode with interrupts disabled

Intel x86 Special Registers

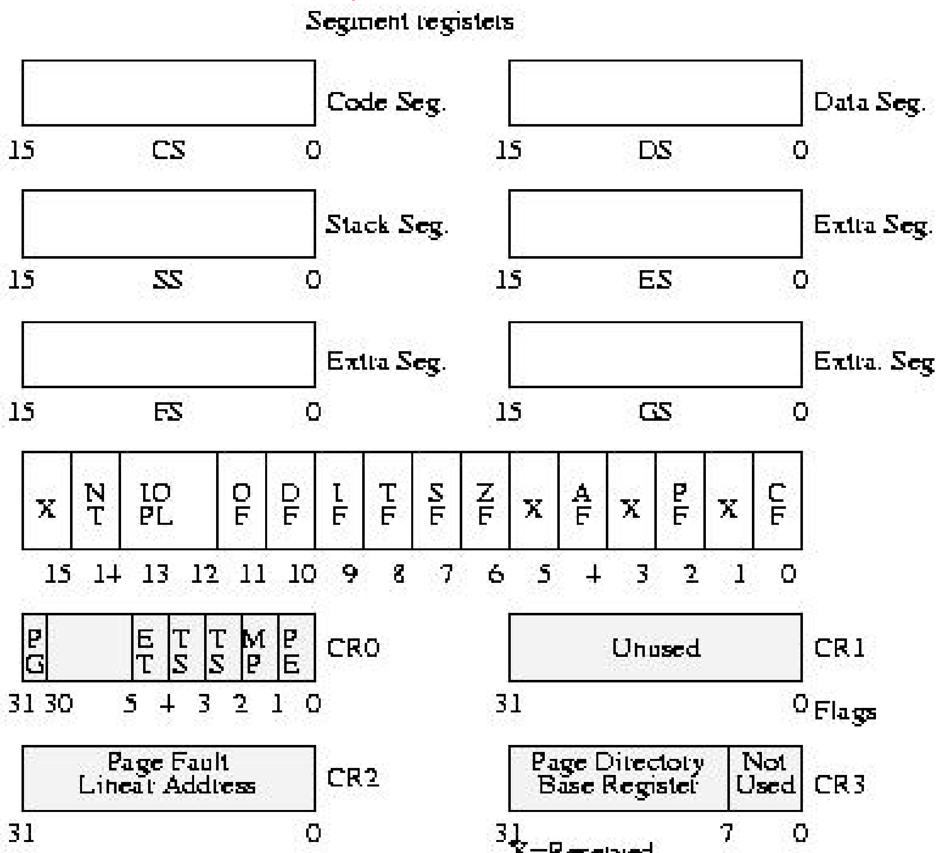


RPL = Requestor Privilege Level
TI = Table Indicator
(0 = GDT, 1 = LDT)
Index = Index into table

Protected Mode segment selector

Typical Segment Register Current Priority is RPL Of Code Segment (CS)

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=I/O 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

Communication

- Now that we have isolated processes, how can they communicate?
  - Shared memory: common mapping to physical page
    - » As long as place objects in shared memory address range, threads from each process can communicate
    - » Note that processes A and B can talk to shared memory through different addresses
    - » In some sense, this violates the whole notion of protection that we have been developing
  - If address spaces don't share memory, all inter-address space communication must go through kernel (via system calls)
    - » Byte stream producer/consumer (put/get): Example, communicate through pipes connecting `stdin/stdout`
    - » Message passing (send/receive): Will explain later how you can use this to build remote procedure call (RPC) abstraction so that you can have one program make procedure calls to another
    - » File System (read/write): File system is shared state!

Closing thought: Protection without Hardware

- Does protection require hardware support for translation and dual-mode behavior?
  - No: Normally use hardware, but anything you can do in hardware can also do in software (possibly expensive)
- Protection via Strong Typing
  - Restrict programming language so that you can't express program that would trash another program
  - Loader needs to make sure that program produced by valid compiler or all bets are off
  - Example languages: LISP, Ada, Modula-3 and Java
- Protection via software fault isolation:
  - Language independent approach: have compiler generate object code that provably can't step out of bounds
    - » Compiler puts in checks for every "dangerous" operation (loads, stores, etc). Again, need special loader.
    - » Alternative, compiler generates "proof" that code cannot do certain things (Proof Carrying Code)
  - Or: use virtual machine to guarantee safe behavior (loads and stores recompiled on fly to check bounds)

Summary

- Memory is a resource that must be shared
  - Controlled Overlap: only shared when appropriate
  - Translation: Change Virtual Addresses into Physical Addresses
  - Protection: Prevent unauthorized Sharing of resources
- Simple Protection through Segmentation
  - Base+limit registers restrict memory accessible to user
  - Can be used to translate as well
- Full translation of addresses through Memory Management Unit (MMU)
  - Every Access translated through page table
  - Changing of page tables only available to user

Summary (2/2)

- Dual-Mode
  - Kernel/User distinction: User restricted
  - User$\rightarrow$Kernel: System calls, Traps, or Interrupts
  - Inter-process communication: shared memory, or through kernel (system calls)

- Exceptions
  - Synchronous Exceptions: Traps (including system calls)
  - Asynchronous Exceptions: Interrupts