CS162 - Operating Systems and Systems Programming

Address Translation

David E. Culler

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

Lecture #14

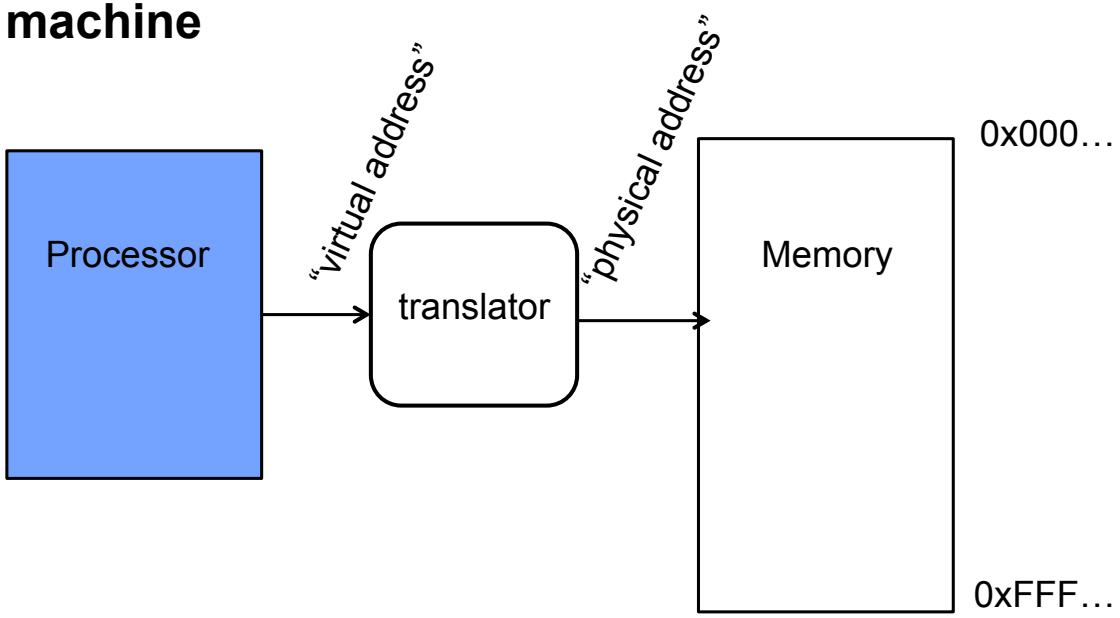
Oct 1, 2014

Reading: A&D 2.7, 8.1-2
HW 3 deferred
Proj 1 CP #2 10/2

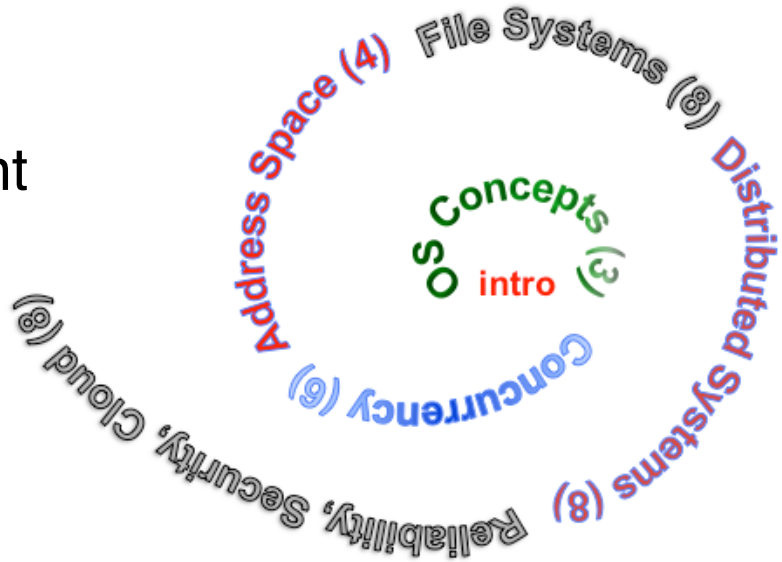
way back when …

Key OS Concept: Address Space

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



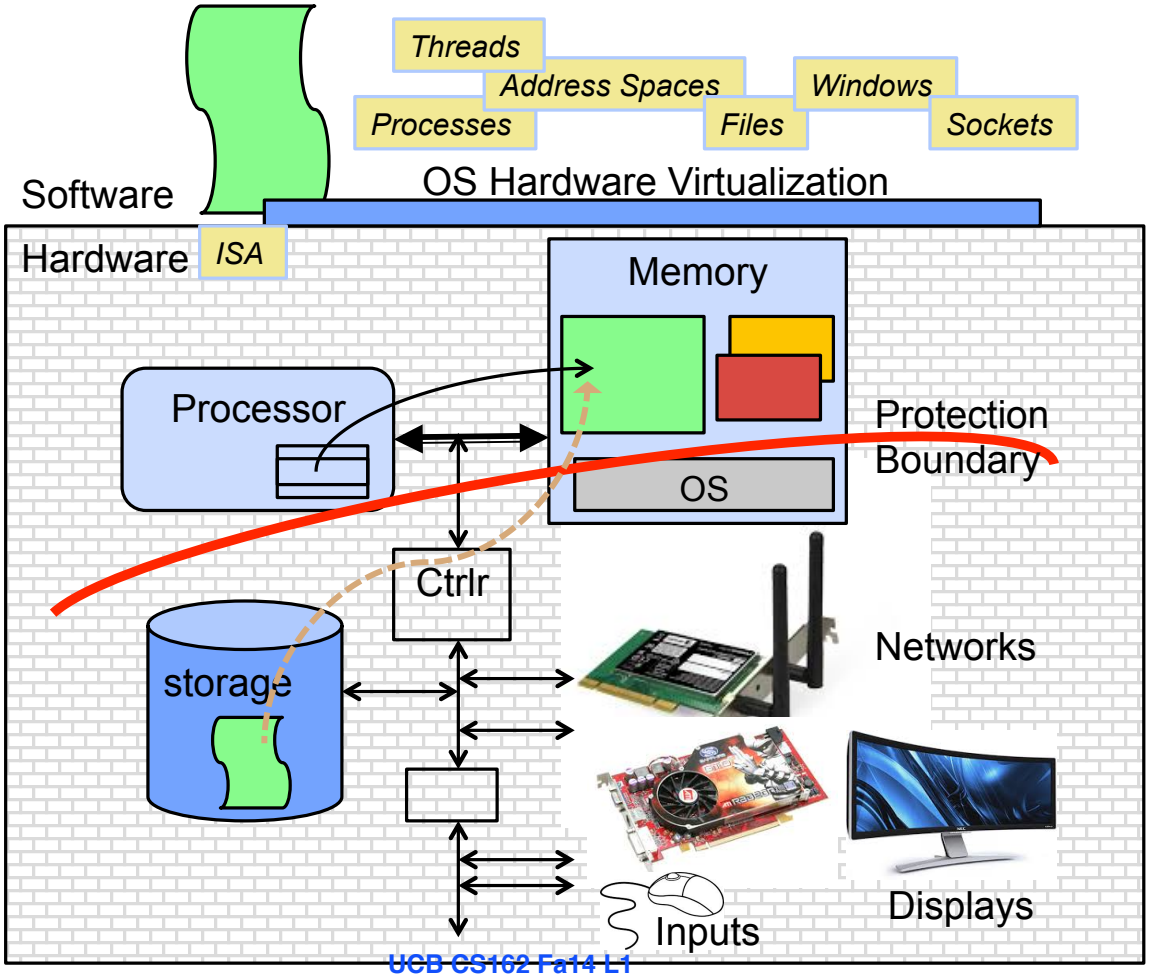
Objective



- Dive deeper into the concepts and mechanisms of 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
- Today: Linking, Segmentation, Paged Virtual Address

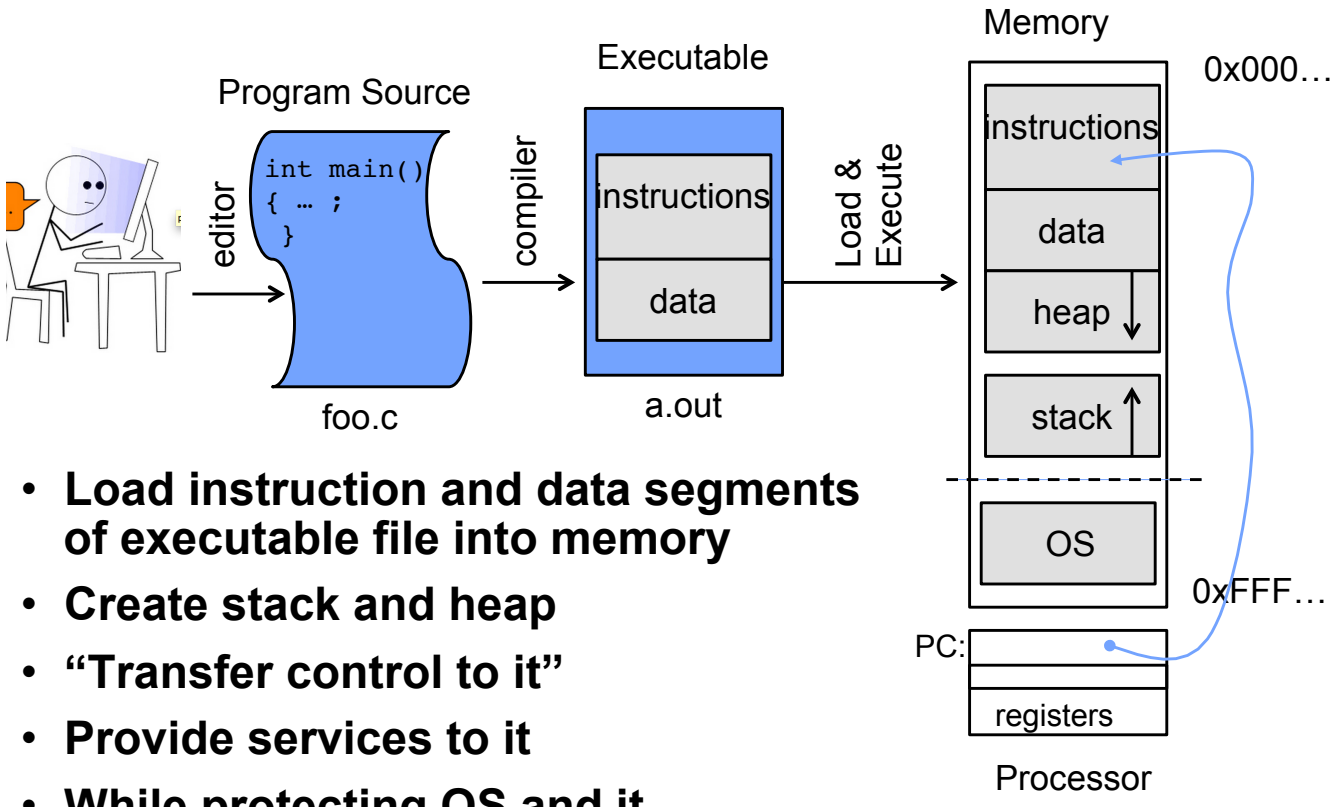
Recall: address translation is key to protection

OS Basics: Loading



Recall: Loading and Managing Memory

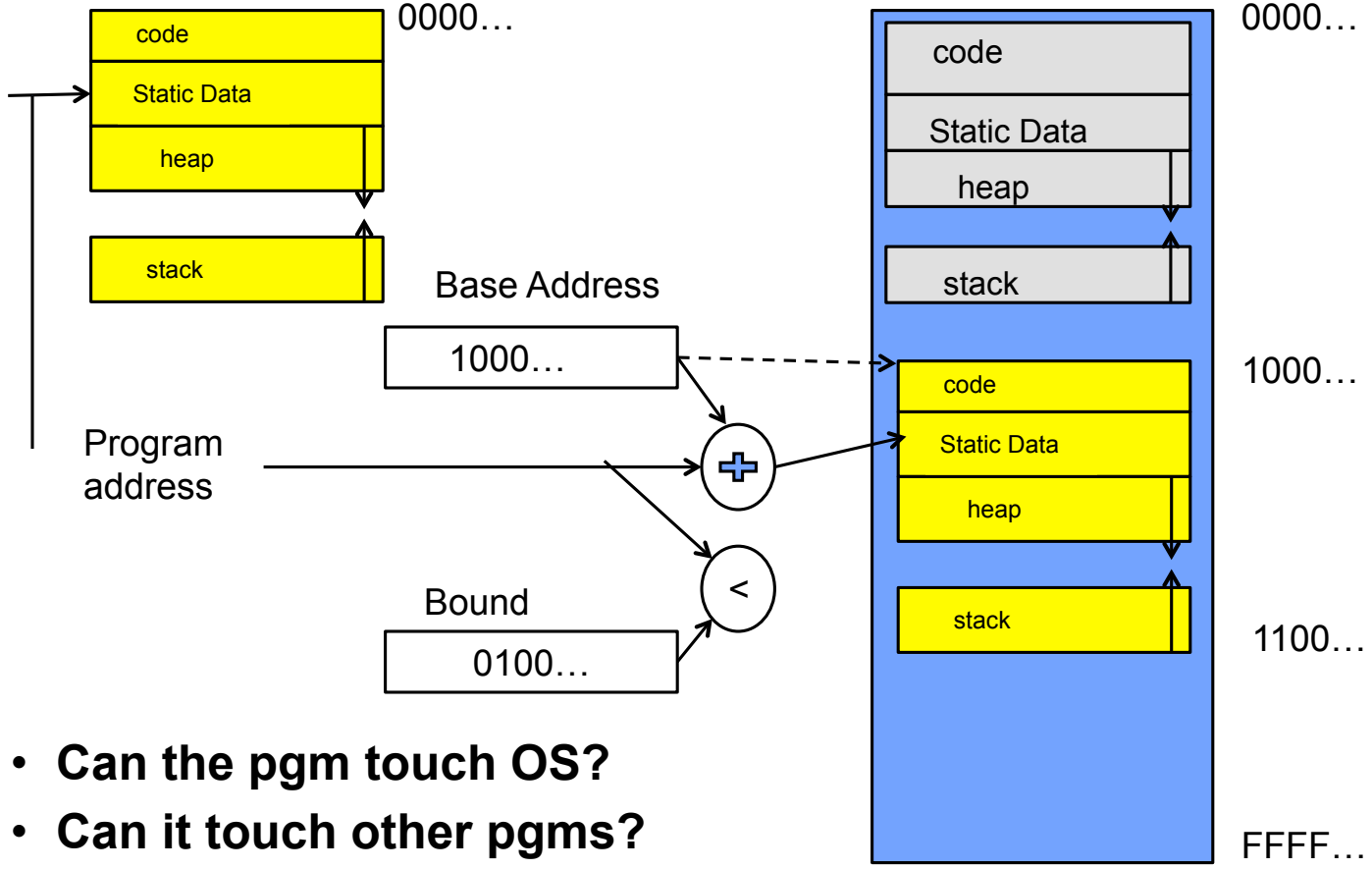
OS Bottom Line: Run Programs



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

Recall Further: L2

A simple address translation: B&B

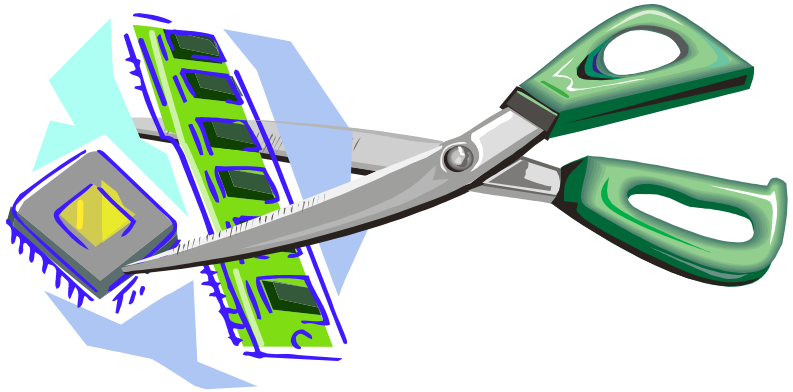


- **Can the pgm touch OS?**
- **Can it touch other pgms?**

Question

- What is the cost of a process fork?
- Depending on what?

Virtualizing Resources



- Physical Reality: Processes/Threads share the same hardware
  - Need to multiplex CPU (Scheduling, Concurrency, Synchronization)
  - Need to multiplex use of Memory (Today)

- Why worry about memory multiplexing?
  - The complete working state of a process and/or kernel is defined by its data in memory (and registers)
  - Consequently, cannot just let different processes use the same memory
  - Generally don't want different processes access to each other's memory (protection)

Important Aspects of Memory Multiplexing

- Controlled overlap:
  - Processes should not collide in physical memory
  - Conversely, would like the ability to share memory when desired (for communication)
- 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
- Translation:
  - Ability to translate accesses from one address space (virtual) to a different one (physical)
  - When translation exists, process uses virtual addresses, physical memory uses physical addresses

Diving down to the instruction level

Binding of Instructions & Data to Memory

Process view of memory

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

Physi

```
0x0300  0C          20
   …                …
0x0900  8C2000C0
0x0904  0C000280
0x0908  2021FFFF
0x090C  14200242
  …
0x0A00
```

Assume 4byte words
0x300 = 4 * 0x0C0
0x0C0 = 0000 1100 0000
0x300 = 0011 0000 0000

Binding of Instructions & Data to Memory

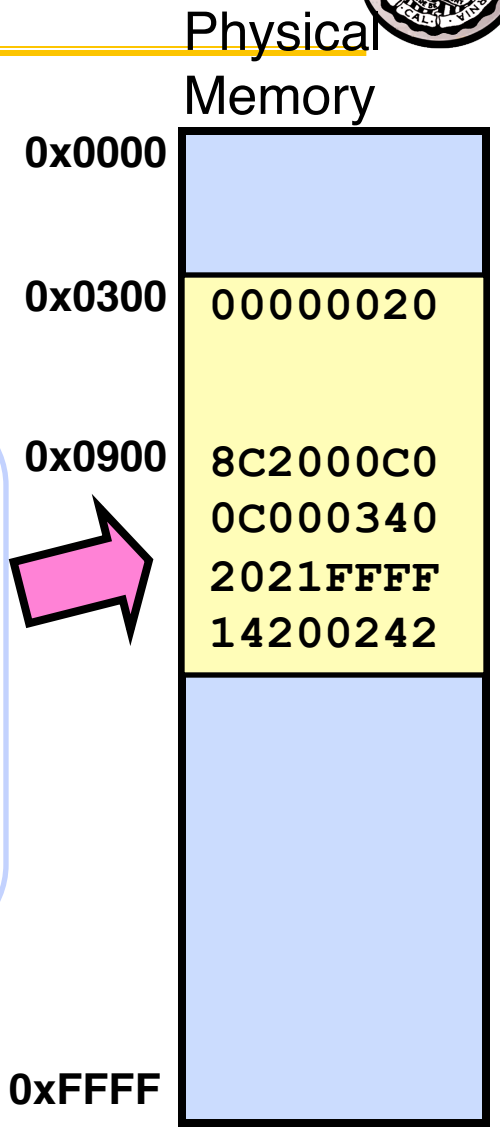
Process view of memory

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

Physical addresses

```
0x0300  00000020
   …       …
0x0900  8C2000C0
0x0904  0C000280
0x0908  2021FFFF
0x090C  14200242
  …
0x0A00
```

Physical Memory



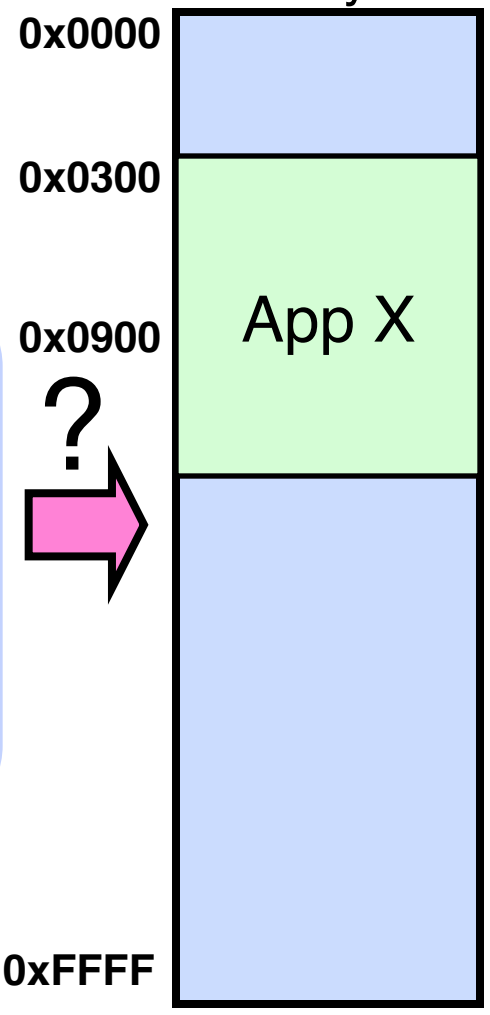
Binding of Instructions and Data to Memory

Process view of memory

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

Physical addresses

```
0x300    00000020
  …        …
0x900    8C2000C0
0x904    0C000280
0x908    2021FFFF
0x90C    14200242
 …
0x0A00
```



Need address translation!

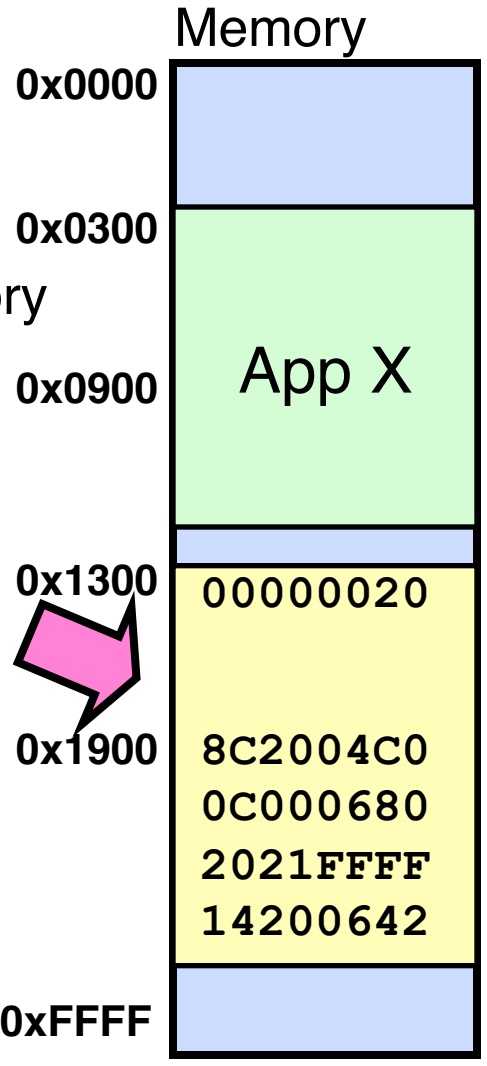
Binding of Instructions and Data to Memory

Process view of memory

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

Processor view of memory

```
0x1300  00000020
   …       …
0x1900  8C2004C0
0x1904  0C000680
0x1908  2021FFFF
0x190C  14200642
  …
0x1A00
```



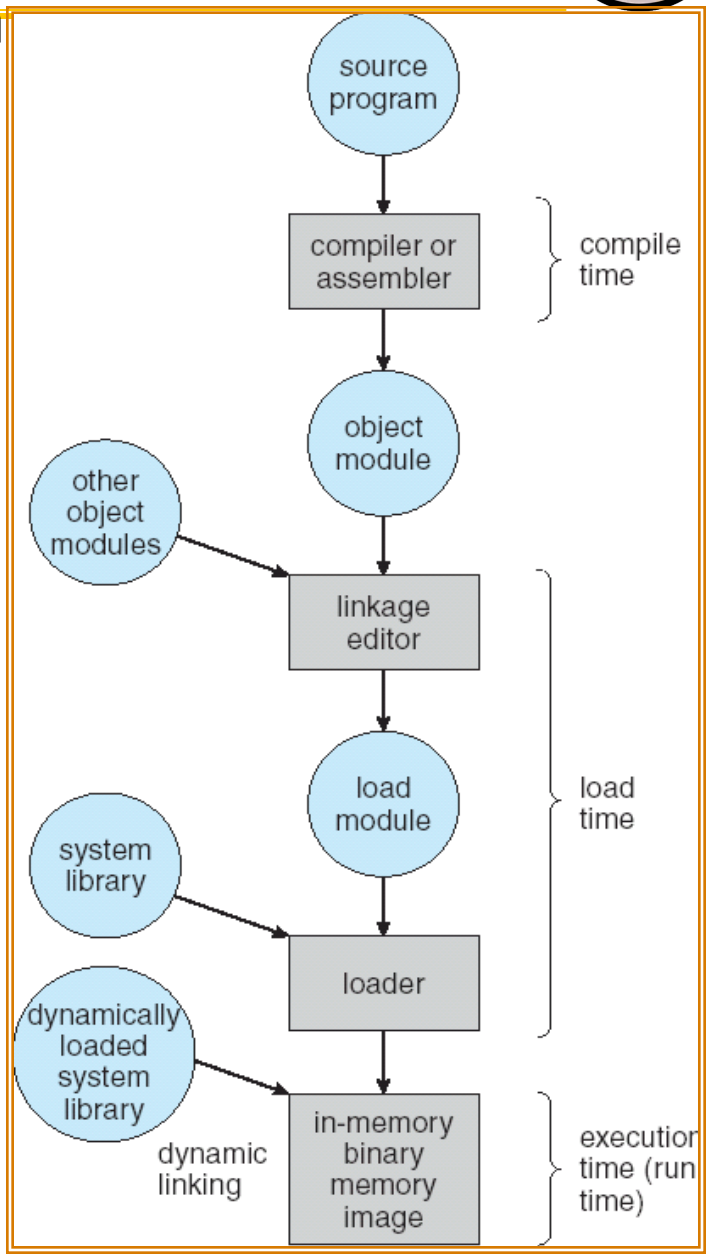
- One of many possible translations!
- Where does translation take place?
  - Compile time, Link/Load time, or Execution time?

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

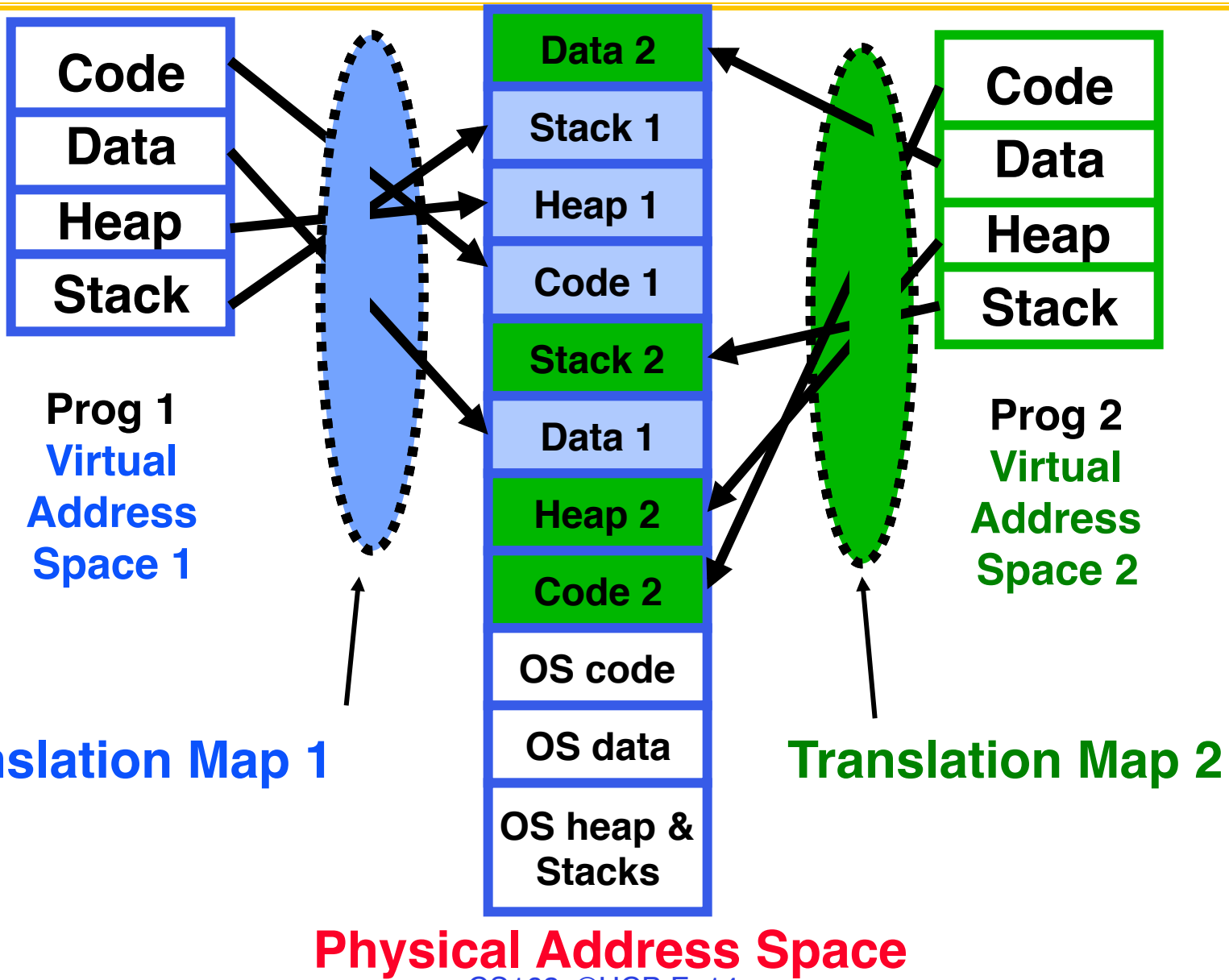
- Dynamic Libraries
  - Linking postponed until execution
  - Small piece of code, *stub*, used to locate appropriate memory-resident library routine
  - Stub replaces itself with the address of the routine, and executes routine



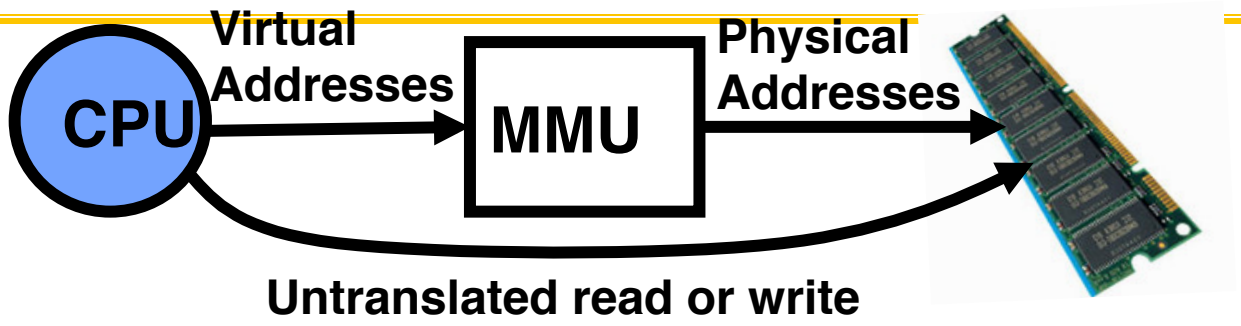
Exercise curiousity

- HW1
- vagrant
- gcc –g –o io.o io.c
- objdump –x io.o
- objdump –d io.o
- gcc –g –o shell shell.o parse.o io.o
- gdb shell
  - disa freadln
  - disa fgets
  - b freadln
  - info registers
  - info scope
  - bt
  - frame, info frame

Example of General Address Translation



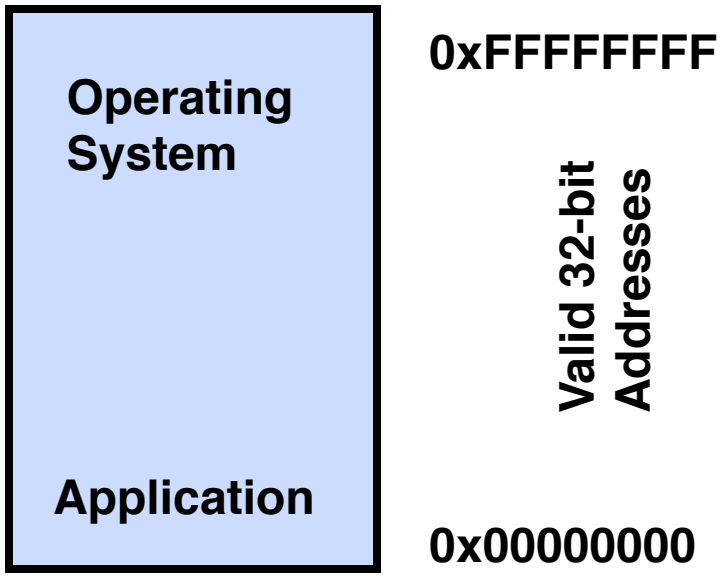
Two Views of Memory



- 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 from memory (physical memory)
  - Translation box (MMU) converts between the two views
- Translation essential to implementing 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

Uniprogramming

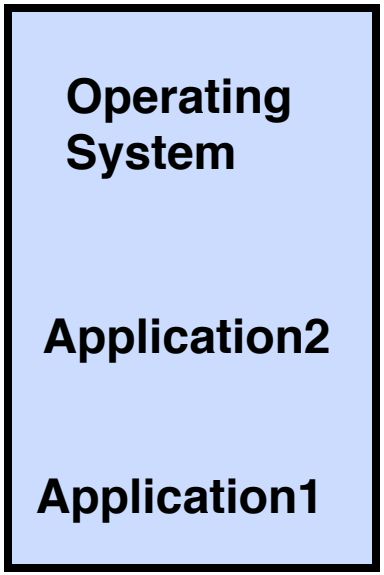
- Uniprogramming (no Translation or Protection)
  - Application always runs at same place in physical memory since only one application at a time
  - Application can access any physical address



  - Application given illusion of dedicated machine by giving it reality of a dedicated machine

Multiprogramming (primitive stage)

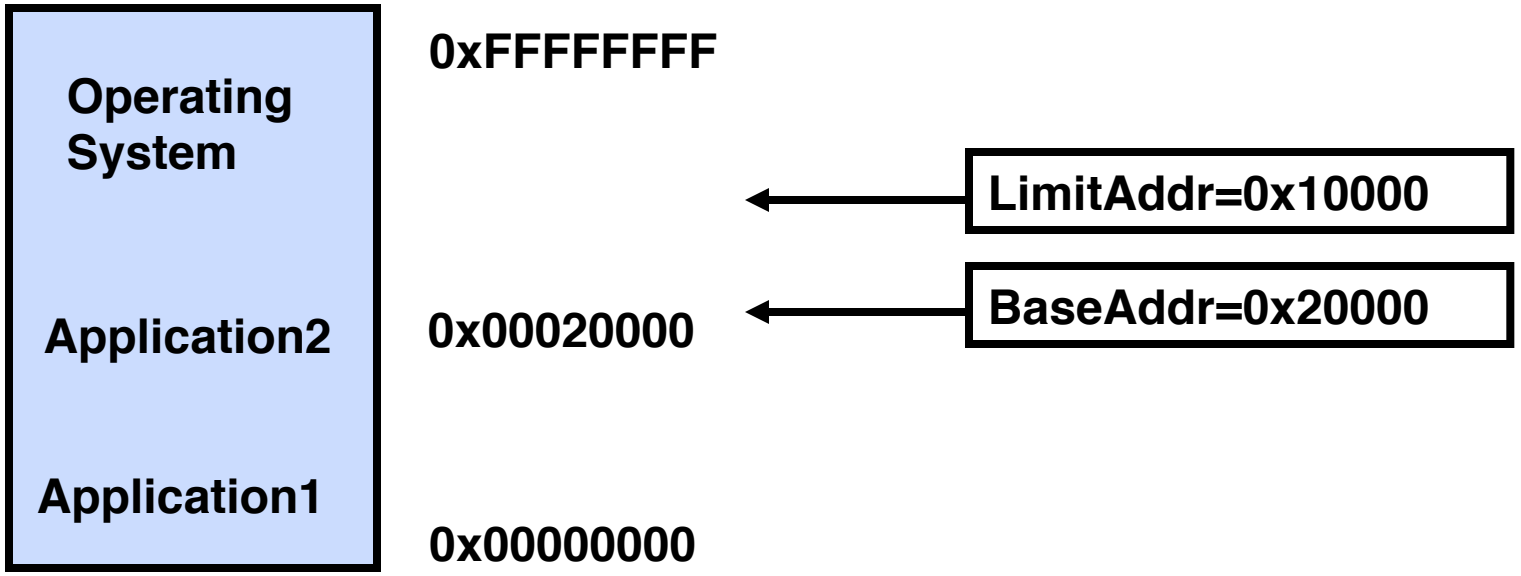
- Multiprogramming without Translation or Protection
  - Must somehow prevent address overlap between threads



  - 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 (relocation)
    - » Common in early days (… till Windows 3.x, 95?)
- With this solution, no protection: bugs in any program can cause other programs to crash or even the OS

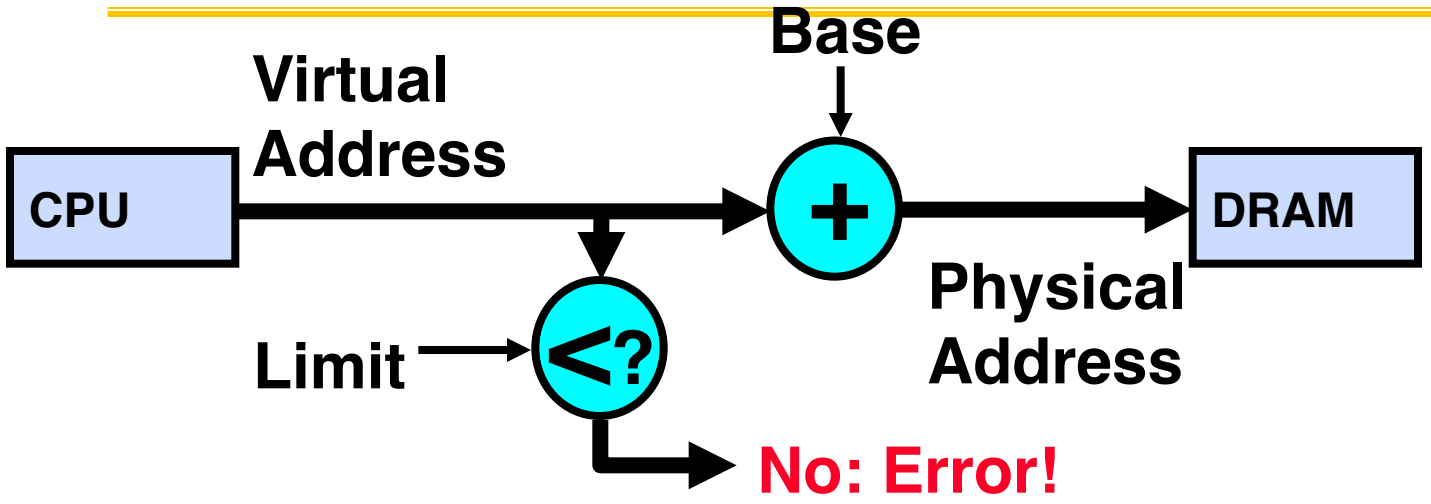
Multiprogramming (Version with Protection)

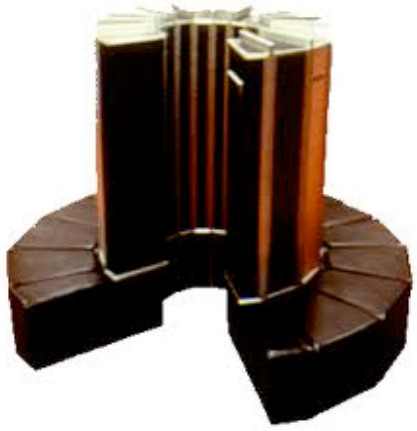
- Can we protect programs from each other without translation?



  - 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 PCB (Process Control Block)
    - » User not allowed to change base/limit registers

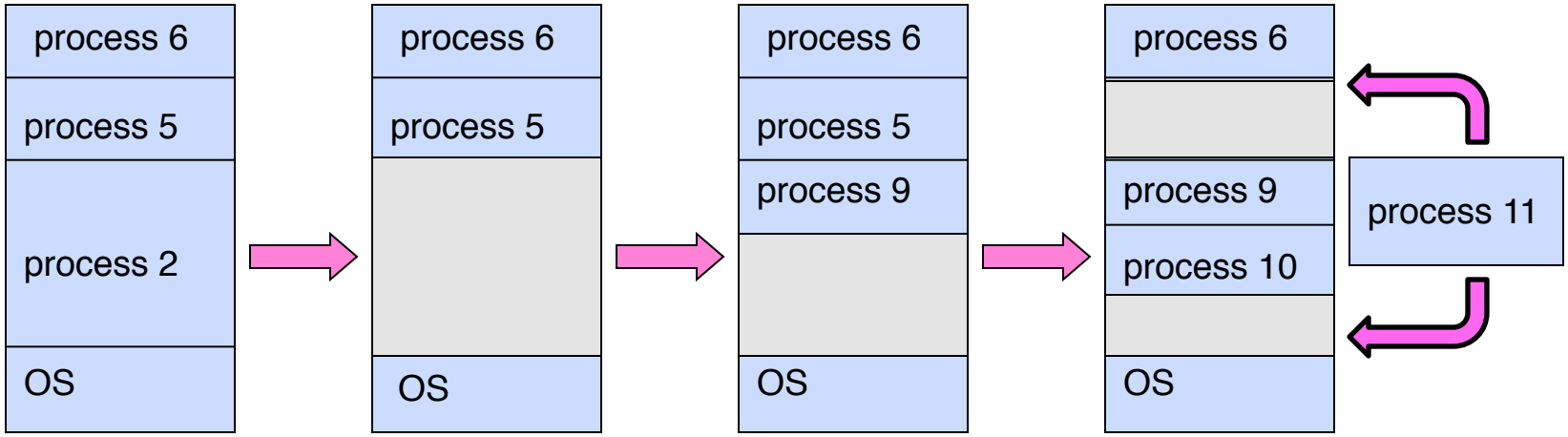
Simple Base and Bounds (CRAY-1)





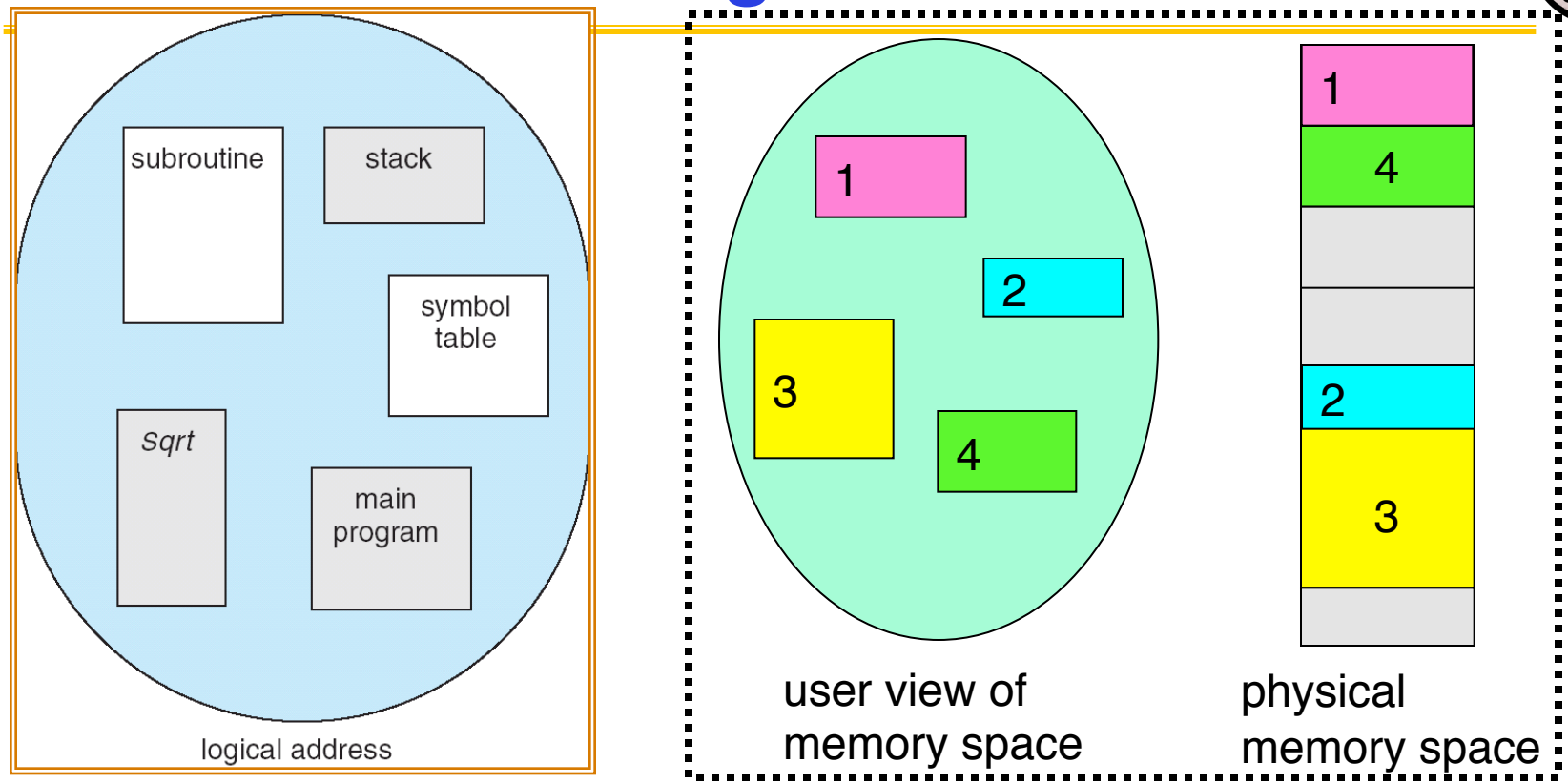
- Could use base/limit for dynamic address translation – translation happens at execution:
  - Alter address of every load/store by adding "base"
  - Generate error if address bigger than limit
- This gives program the illusion that it is running on its own dedicated machine, with memory starting at 0
  - Program gets continuous region of memory
  - Addresses within program do not have to be relocated when program placed in different region of DRAM

Issues with Simple B&B Method



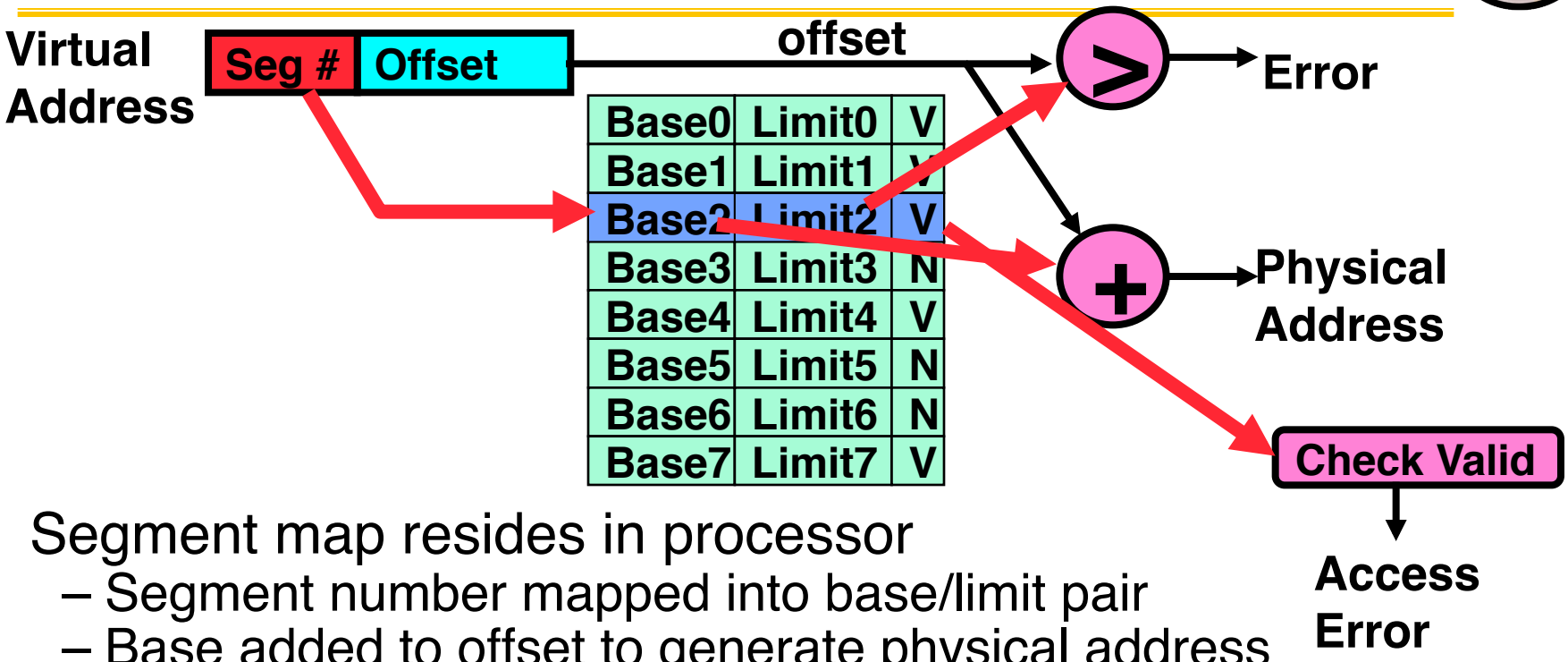
- Fragmentation problem
  - Not every process is the same size
  - Over time, memory space becomes fragmented
- Hard to do inter-process sharing
  - Want to share code segments when possible
  - Want to share memory between processes
  - Helped by providing multiple segments per process

More Flexible Segmentation



- Logical View: multiple separate segments
  - Typical: Code, Data, Stack
  - Others: memory sharing, etc
- Each segment is given region of contiguous memory
  - Has a base and limit
  - Can reside anywhere in physical memory

Implementation of Multi-Segment Model

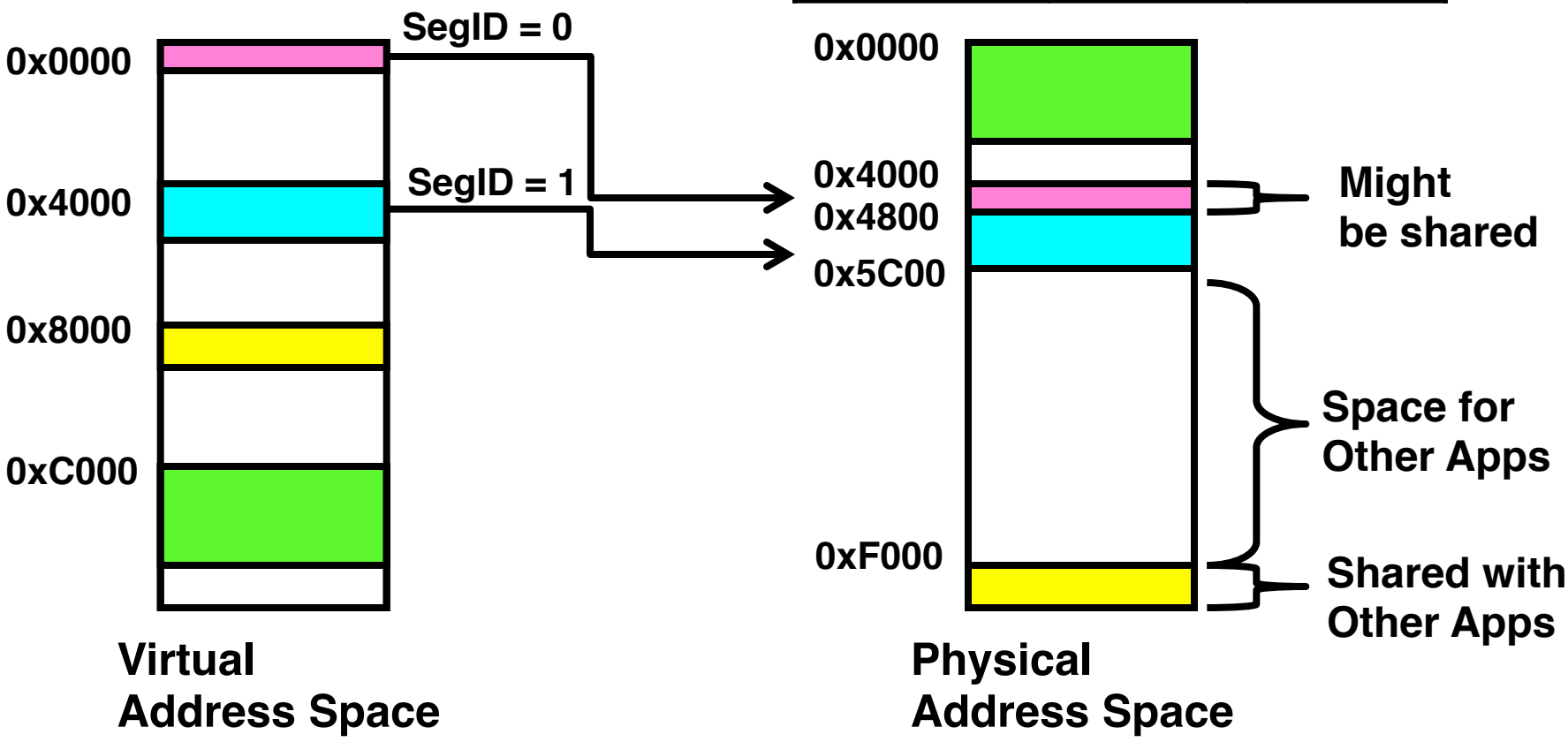


- Segment map resides in processor
  - Segment number mapped into base/limit pair
  - Base added to offset to generate physical address
  - Error check catches offset out of range
- As many chunks of physical memory as entries
  - Segment addressed by portion of virtual address
  - However, could be 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

Example: Four Segments (16 bit addresses)

Virtual Address Format

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

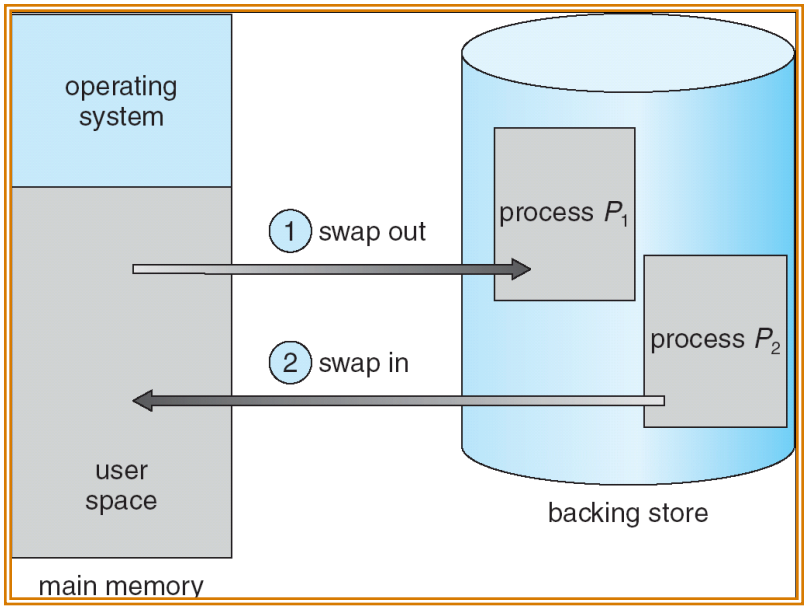


Administrative Break

How do we run more programs than fit in memory ?

Schematic View of "Swapping"

- Q: What if not all processes fit in memory?
- A: Swapping: Extreme form of Context Switch
  - In order to make room for next process, some or all of the previous process is moved to disk
  - This greatly increases the cost of context-switching



- 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

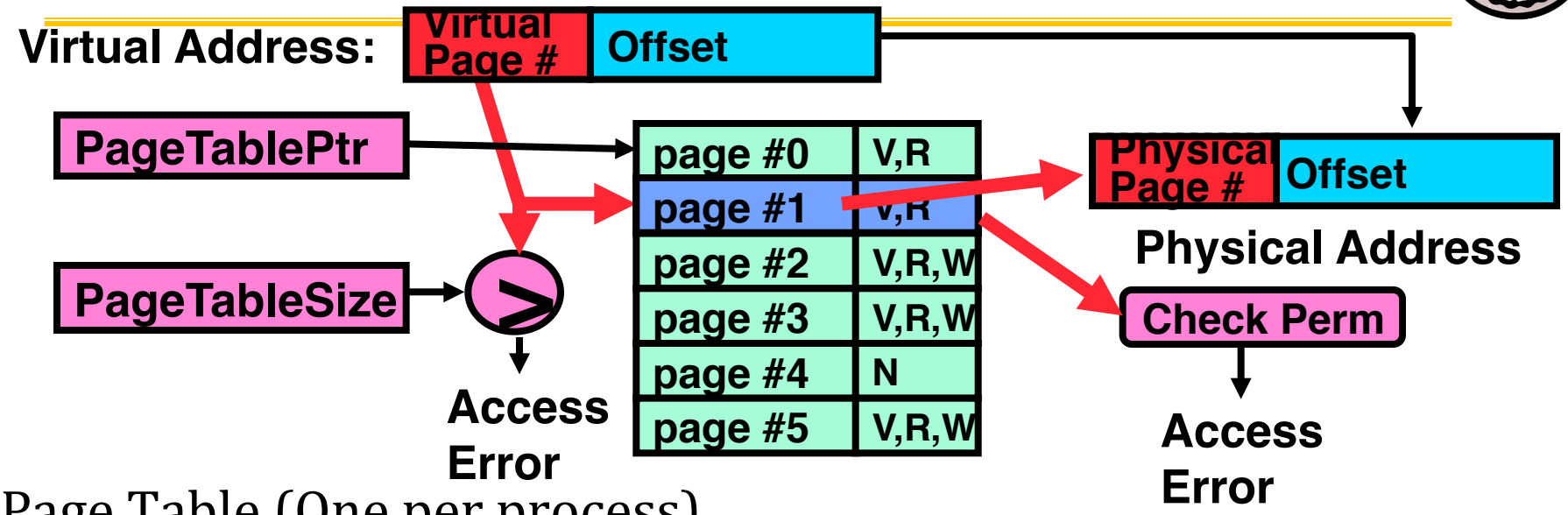
- Must fit variable-sized chunks into physical memory
- May move processes multiple times to fit everything
- Limited options for swapping to disk
- Fragmentation: wasted space
  - External: free gaps between allocated chunks
  - Internal: don't need all memory within allocated chunks

Paging: Physical Memory in Fixed Size Chunks

- Solution to fragmentation from segments?
  - Allocate physical memory in fixed size chunks ("pages")
  - Every chunk of physical memory is equivalent
    - » Can use simple vector of bits to handle allocation:
      0011000111000 1101 … 110010
    - » Each bit represents page of physical memory
      1⇒allocated, 0⇒free

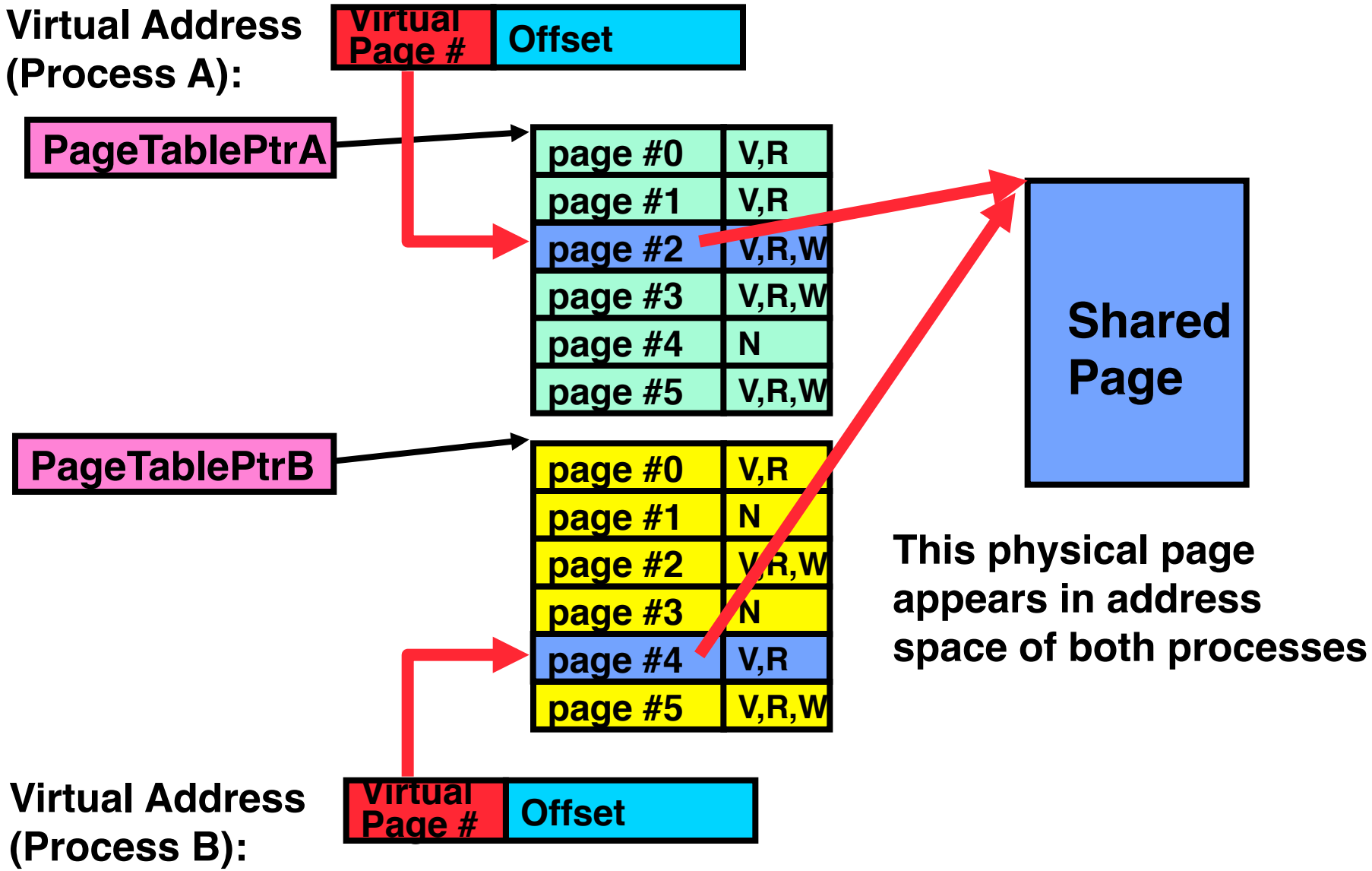
- Should pages be as big as our previous segments?
  - No: Can lead to lots of internal fragmentation
    - » Typically have small pages (1K-16K)
  - Consequently: need multiple pages/segment

How to Implement Paging?

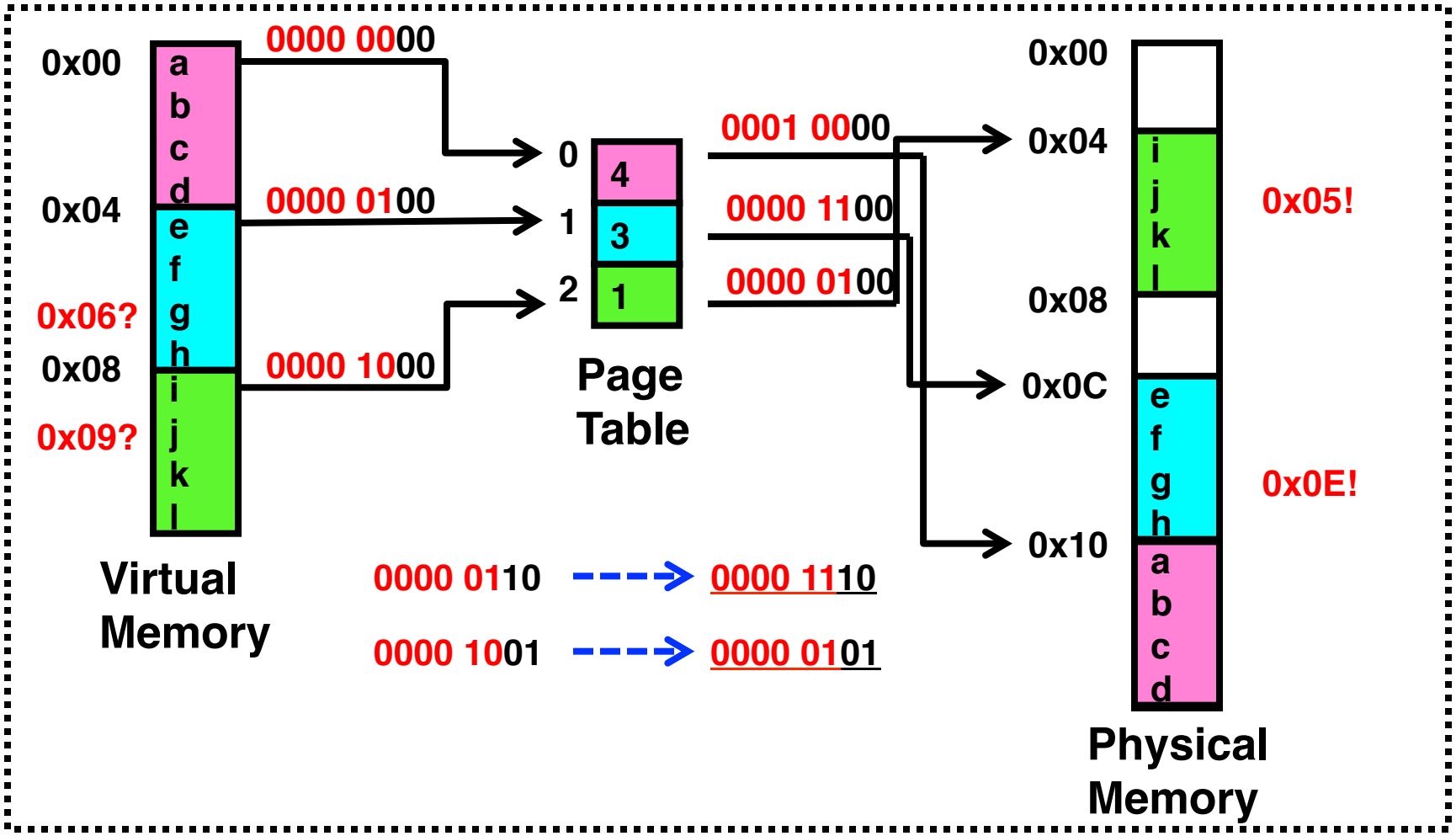


- Page Table (One per process)
  - Resides in physical memory
  - Contains physical page and permission for each virtual page
    - » Permissions include: Valid bits, Read, Write, etc
- Virtual address mapping
  - Offset from Virtual address copied to Physical Address
    - » Example: 10 bit offset ⇒ 1024-byte pages
  - Virtual page # is all remaining bits
    - » Example for 32-bits: 32-10 = 22 bits, i.e. 4 million entries
    - » Physical page # copied from table into physical address
  - Check Page Table bounds and permissions

What about Sharing?



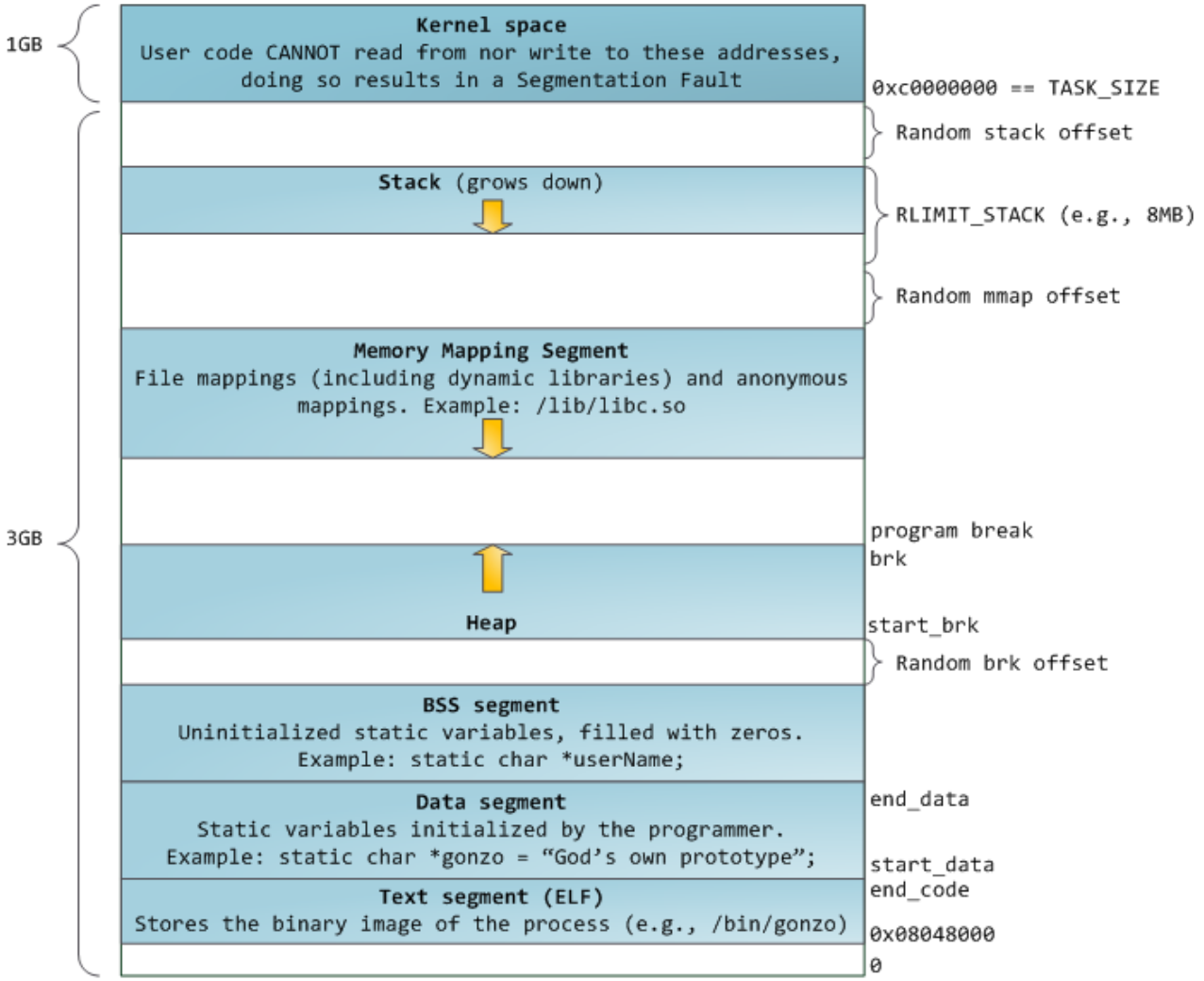
Simple Page Table Example



Page Table Discussion

- What needs to be switched on a context switch?
  - Page table pointer and limit
- Analysis
  - Pros
    - » Simple memory allocation
    - » Easy to Share
  - Con: What if address space is sparse?
    - » E.g. on UNIX, code starts at 0, stack starts at $(2^{31}-1)$.
    - » With 1K pages, need 2 million page table entries!
  - Con: What if table really big?
    - » Not all pages used all the time ⇒ would be nice to have working set of page table in memory
- How about combining paging and segmentation?

E.g., Linux 32-bit



http://static.duartes.org/img/blogPosts/linuxFlexibleAddressSpaceLayout.png

Summary

- Memory is a resource that must be multiplexed
  - 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
- Page Tables
  - Memory divided into fixed-sized chunks of memory
  - Offset of virtual address same as physical address