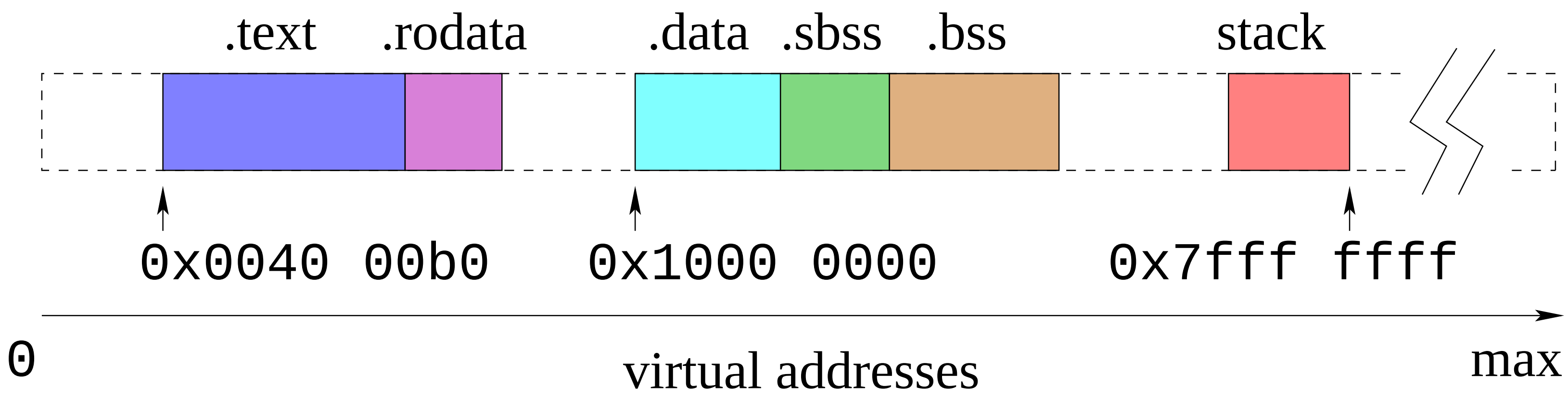
Virtual and Physical Addresses

- Physical addresses are provided directly by the machine.
  - one physical address space per machine
  - addresses typically range from some minumum (sometimes 0) to some maximum, though some portions of this range are usually used by the OS and/or devices, and are not available for user processes
- Virtual addresses (or logical addresses) are addresses provided by the OS to processes.
  - one virtual address space per process
  - addresses typically start at zero, but not necessarily
  - space may consist of several *segments*
- Address translation (or address binding) means mapping virtual addresses to physical addresses.

Address Space Layout



- Size of each section except stack is specified in ELF file
- Code (i.e., text), read-only data and initialized data segments are initialized from the ELF file. Remaining sections are initially zero-filled.
- Sections have their own specified alignment and segments are page aligned.
- 3 segments = (.text + .rodata), (.data + .sbss + .bss), (stack)
- Note: not all programs contain this many segments and sections.

C Code for Sections and Segments Example

```
#include <unistd.h>

#define N   (200)

int x = 0xdeadbeef;
int y1;
int y2;
int y3;
int array[4096];
char const *str = "Hello World\n";
const int z = 0xabcddcba;

struct example {
  int ypos;
  int xpos;
};
```

C Code for Sections and Segments Example (cont'd)

```
int
main()
{
  int count = 0;
  const int value = 1;
  y1 = N;
  y2 = 2;
  count = x + y1;
  y2 = z + y2 + value;

  reboot(RB_POWEROFF);
  return 0; /* avoid compiler warnings */
}
```

cs350-readelf Output: Sections and Segments

```
% cs350-readelf -a segments > readelf.out
% cat readelf.out

Section Headers:
[Nr] Name        Type         Addr     Off    Size   ES Flg  ... Al
[ 0]             NULL         00000000 000000 000000 00      ...  0
[ 1] .reginfo    MIPS_REGINFO 00400094 000094 000018 18   A  ...  4
[ 2] .text       PROGBITS     004000b0 0000b0 000250 00  AX  ... 16
[ 3] .rodata     PROGBITS     00400300 000300 000020 00   A  ... 16
[ 4] .data       PROGBITS     10000000 001000 000010 00  WA  ... 16
[ 5] .sbss       NOBITS       10000010 001010 000014 00 WAp  ...  4
[ 6] .bss        NOBITS       10000030 00101c 004000 00  WA  ... 16
[ 7] .comment    PROGBITS     00000000 00101c 000036 00      ...  1
...
Key to Flags:
  W (write), A (alloc), X (execute), M (merge), S (strings)
  I (info), L (link order), G (group), x (unknown)
  O (extra OS processing required) o (OS specific), p (processor s

## Size = number of bytes (e.g., .text is 0x250 = 592 bytes
## Off  = offset into the ELF file
## Addr = virtual address
```

cs350-readelf Output: Sections and Segments

```
Program Headers:
Type     Offset   VirtAddr   PhysAddr   FileSiz MemSiz  Flg Align
REGINFO  0x000094 0x00400094 0x00400094 0x00018 0x00018 R   0x4
LOAD     0x000000 0x00400000 0x00400000 0x00320 0x00320 R E 0x1000
LOAD     0x001000 0x10000000 0x10000000 0x00010 0x04030 RW  0x1000

 Section to Segment mapping:
  Segment Sections...
   00     .reginfo
   01     .reginfo .text .rodata
   02     .data .sbss .bss

## .reginfo = register info (not used)
## .text    = code/machine instructions
## .rodata  = read-only data (e.g., string literals, consts)
## .data    = data (i.e., global variables)
## .bss     = Block Started by Symbol
##            has a symbol but no value (i.e, uninitialized data)
## .sbss    = ?small bss?
```

cs350-objdump -s output: Sections and Segments

```
% cs350-objdump -s segments > objdump.out
% cat objdump.out
...

Contents of section .text:
 4000b0 3c1c1001 279c8000 3c08ffff 3508fff8  <...'...<...5...
...

## Decoding 3c1c1001 to determine instruction
## 0x3c1c1001 = binary 111100000111000000010000000000001
## 0011 1100 0001 1100 0001 0000 0000 0001
## instr  | rs    | rt    |   immediate
## 6 bits | 5 bits| 5 bits|   16 bits
## 001111 | 00000 | 11100 |  0001 0000 0000 0001
## LUI    | 0     | reg 28|   0x1001
## LUI    | unused| reg 28|   0x1001
## Load unsigned immediate into rt (register target)
## lui gp, 0x1001
```

cs350-objdump -s output: Sections and Segments

```
Contents of section .rodata:
 400300 48656c6c 6f20576f 726c640a 00000000  Hello World.....
 400310 abcddcba 00000000 00000000 00000000  ................
...
## 0x48 = ’H’ 0x65 = ’e’ 0x0a = ’\n’ 0x00 = ’\0’
## Align next int to 4 byte boundary
## const int z = 0xabcddcba
## If compiler doesn’t prevent z from being written/hardware could
## Size = 0x20 = 32 bytes "Hello World\n\0" = 13 + 3 padding = 16
##               +  const int z = 4 = 20
## Then align to the next 16 byte boundry at 32 bytes.
```

cs350-objdump -s output: Sections and Segments

```
Contents of section .data:
 10000000 deadbeef 00400300 00000000 00000000  .....@..........
...
## Size = 0x10 bytes = 16 bytes
## int x = deadbeef (4 bytes)
## char const *str = "Hello World\n"; (4 bytes)
## value stored in str = 0x00400300.
## NOTE: this is the address of the start
## of the .rodata section (i.e., address of ’H’).
```

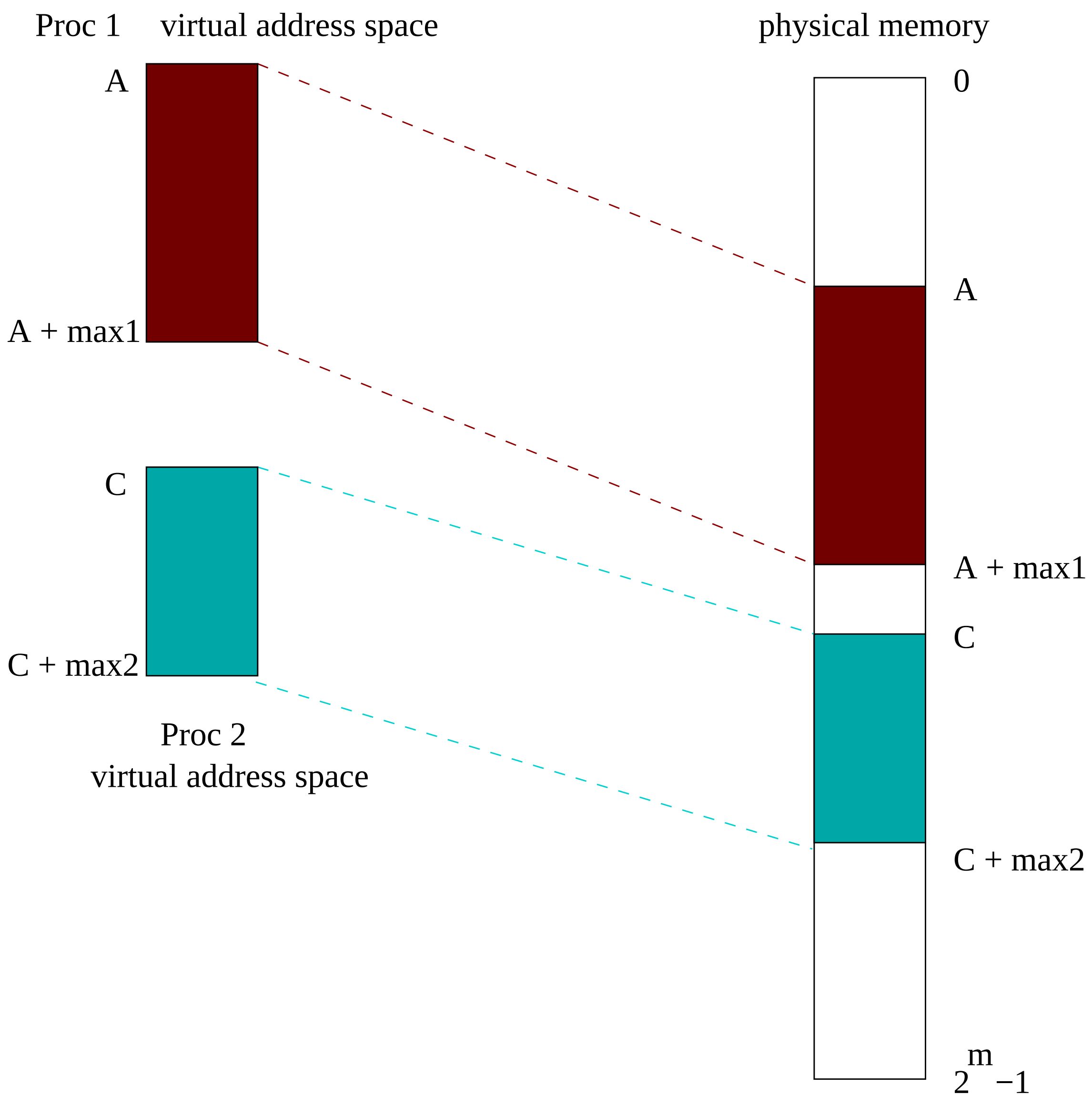
cs350-objdump -s output: Sections and Segments

```
% cs350-nm -n segments > nm.out
% cat nm.out
...
10000010 A __bss_start
10000010 A _edata
10000010 A _fbss
10000010 S y3
10000014 S y2
10000018 S y1
1000001c S errno
10000020 S __argv
...
## .bss  Size = 0x4000 = 16384 = 4096 * sizeof(int)
10000030 B array
10004030 A _end
```

Example 1: A Simple Address Translation Mechanism

- OS divides physical memory into partitions of different sizes.
- Each partition is made available by the OS as a possible virtual address space for processes.
- Properties:
  - virtual addresses are identical to physical addresses
  - address binding is performed by compiler, linker, or loader, not the OS
  - changing partitions means changing the virtual addresses in the application program
    - by recompiling
    - or by *relocating* if the compiler produces relocatable output
  - degree of multiprogramming is limited by the number of partitions
  - size of programs is limited by the size of the partitions

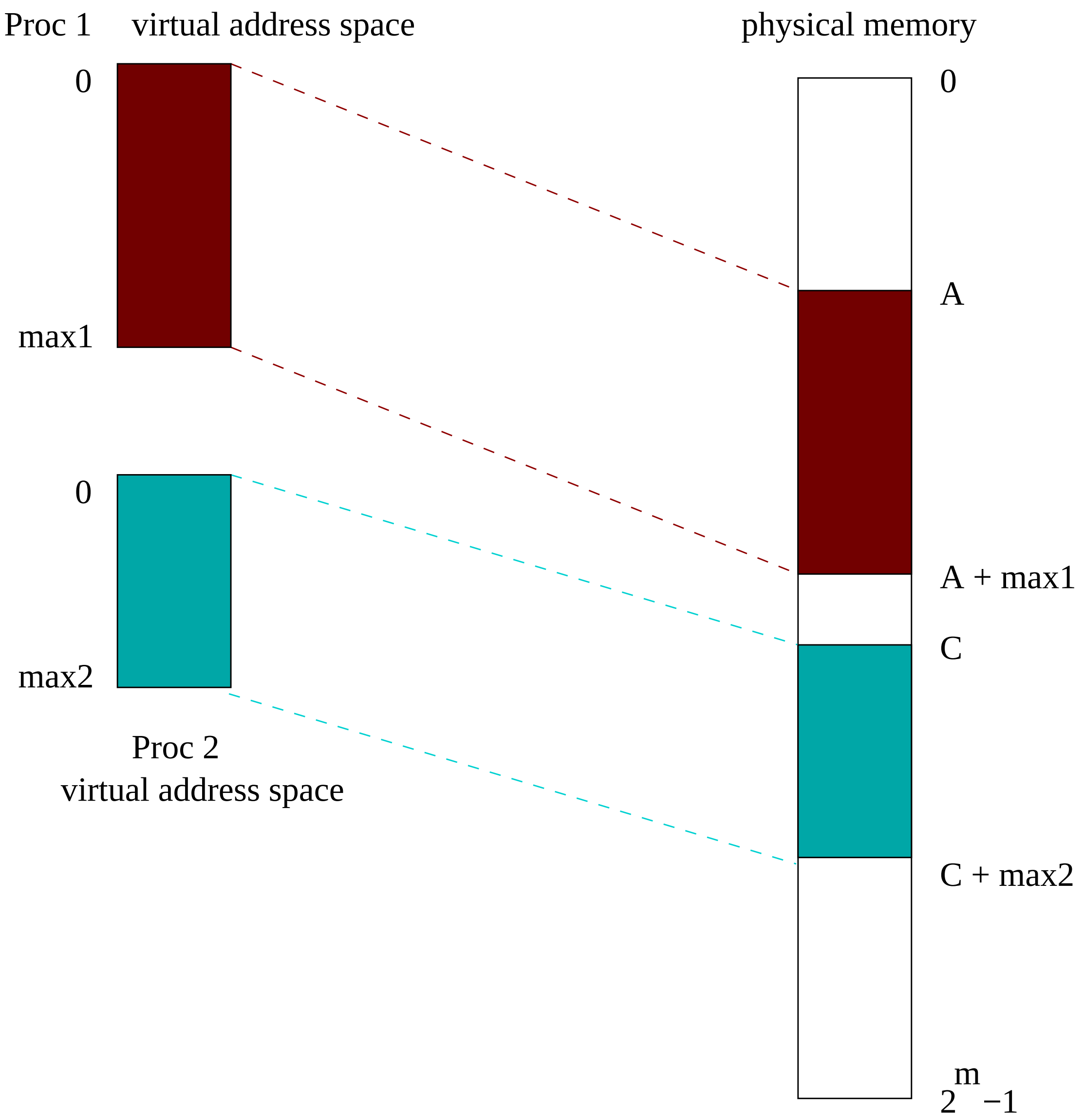
Example 1: Address Space Diagram



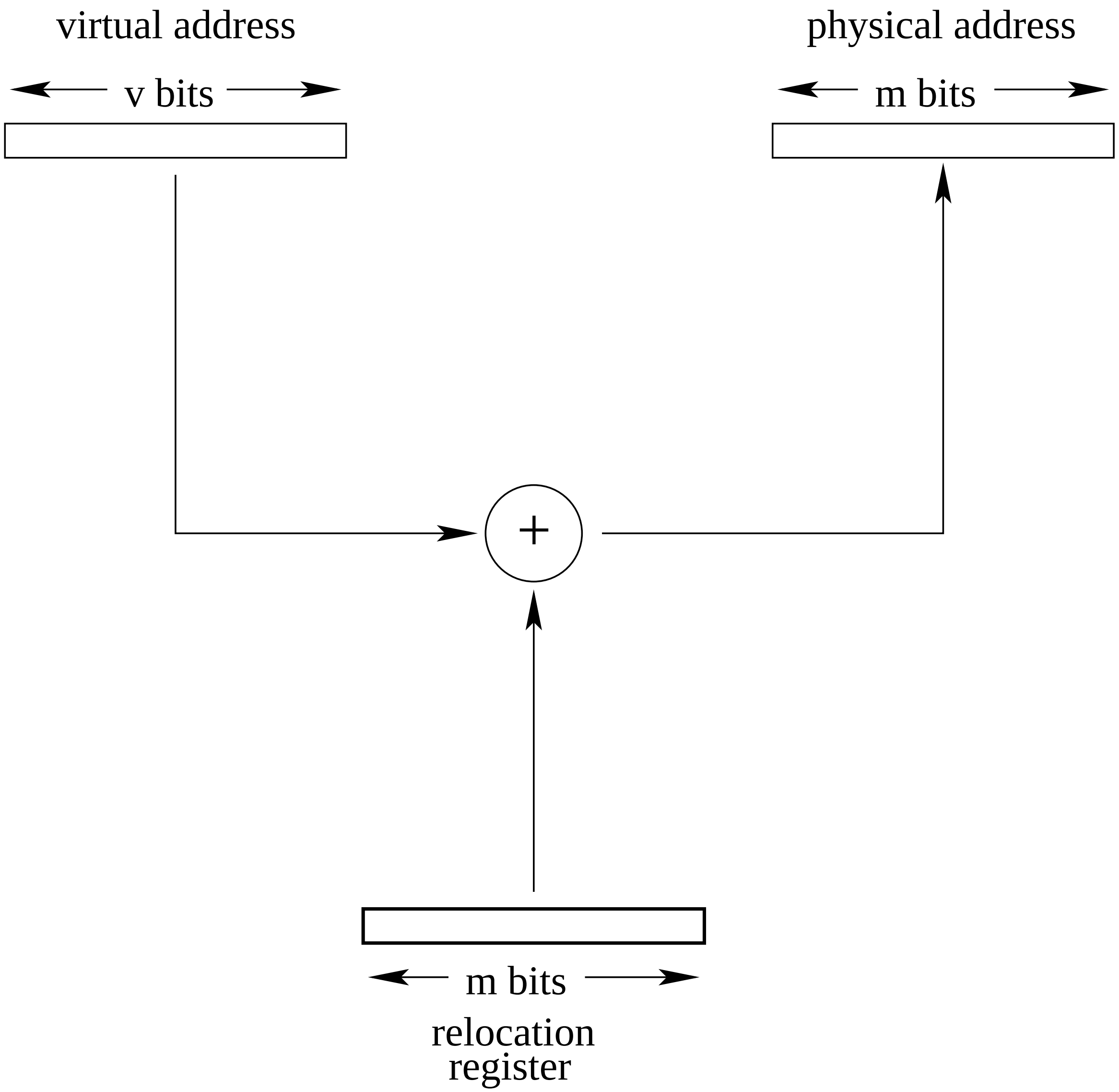
Example 2: Dynamic Relocation

- hardware provides a *memory management unit* which includes a *relocation register*
- *dynamic binding:* at run-time, the contents of the relocation register are added to each virtual address to determine the corresponding physical address
- OS maintains a separate relocation register value for each process, and ensures that relocation register is reset on each context switch
- Properties
  - all programs can have address spaces that start with address 0
  - OS can relocate a process without changing the process's program
  - OS can allocate physical memory dynamically (physical partitions can change over time), again without changing user programs
  - each virtual address space still corresponds to a contiguous range of physical addresses

Example 2: Address Space Diagram



Example 2: Relocation Mechanism



Address Spaces

- OS/161 starts with a dumb/simple address space (`addrspace`)
- `addrspace` maintains the mappings from virtual to physical addresses

```
struct addrspace {
#if OPT_DUMBVM
  vaddr_t as_vbase1;
  paddr_t as_pbase1;
  size_t as_npages1;
  vaddr_t as_vbase2;
  paddr_t as_pbase2;
  size_t as_npages2;
  paddr_t as_stackpbase;
#else
  /* Put stuff here for your VM system */
#endif
};
```

Address Spaces

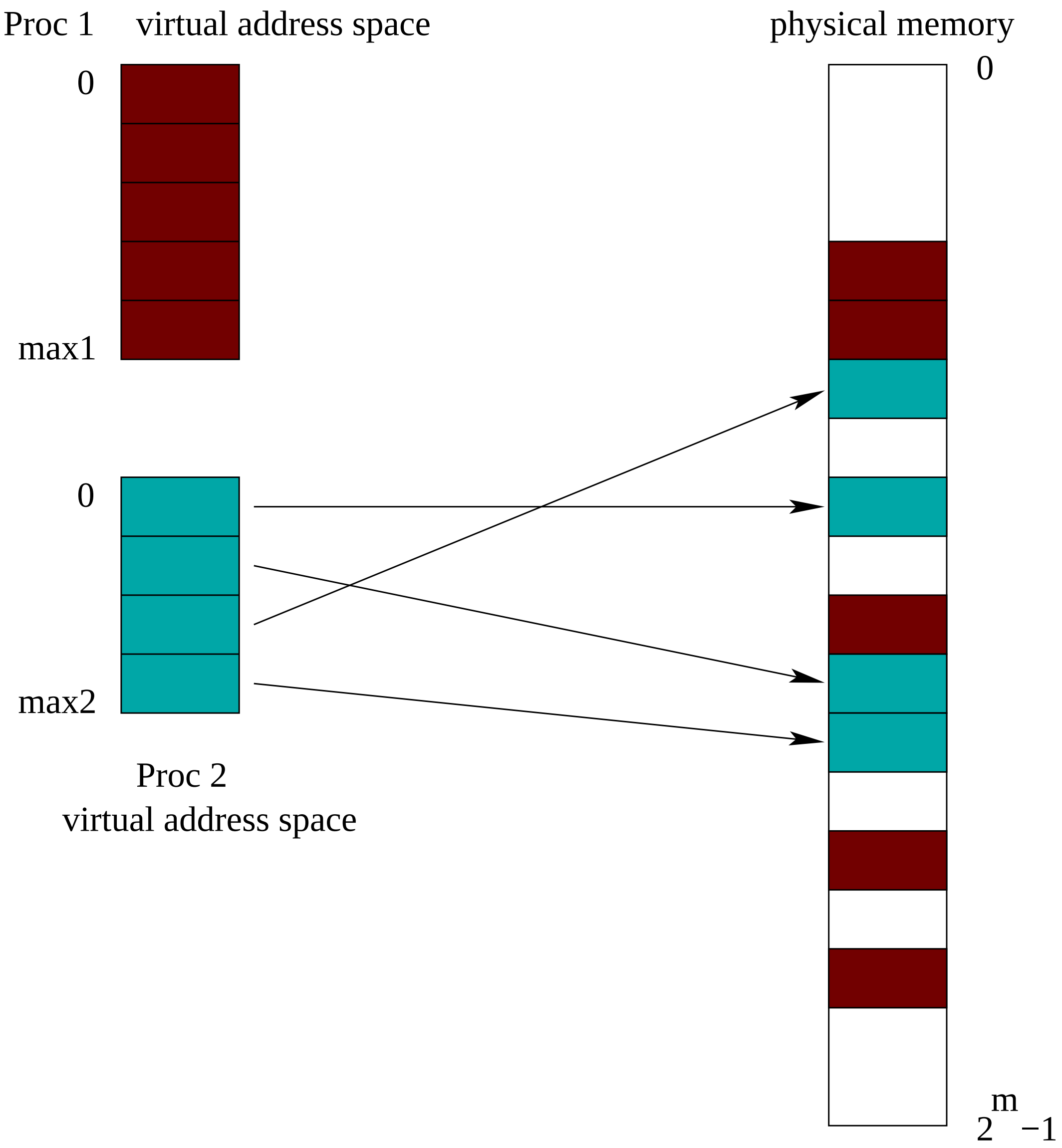
- Think of an address space as ALL of the virtual addresses that can be legally accessed by a thread.
  - .text, .rodata, .data, .sbss, .bss, and stack (if all sections are present).
- kern/arch/mips/mips/dumbvm.c contains functions for creating and managing address spaces
  - `as_create`, `as_destroy`, `as_copy`, `vm_fault`
- kern/lib/copyinout.c contains functions for copying data between kernel and user address spaces
  - `copyin`, `copyout`, `copyinstr`, `copyoutstr`

Why do we need `copyin`, `copyout`, etc.?

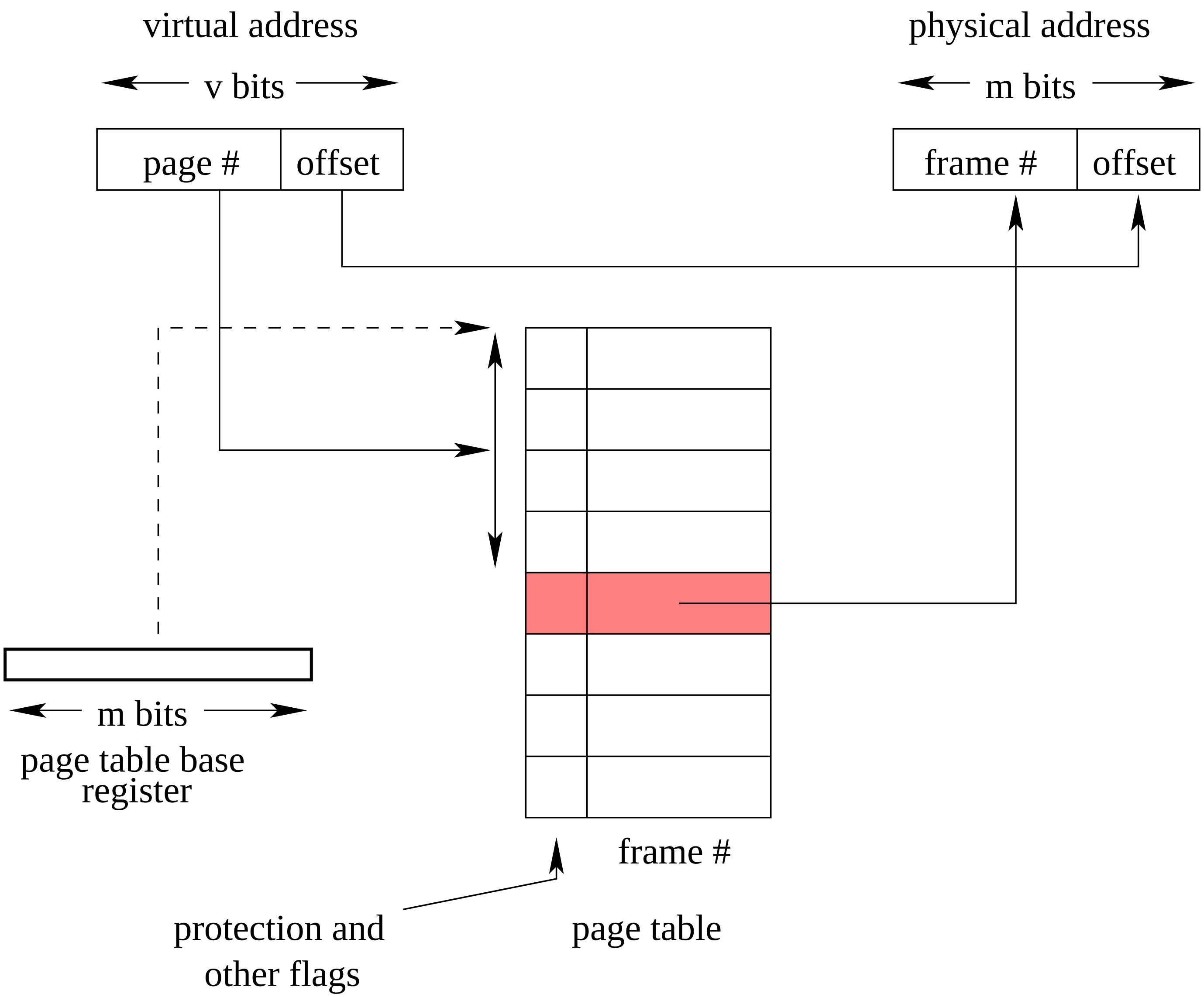
Example 3: Paging

- Each virtual address space is divided into fixed-size chunks called *pages*
- The physical address space is divided into *frames*. Frame size matches page size.
- OS maintains a *page table* for each process. Page table specifies the frame in which each of the process's pages is located.
- At run time, MMU translates virtual addresses to physical using the page table of the running process.
- Properties
  - simple physical memory management
  - virtual address space need not be physically contiguous in physical space after translation.

Example 3: Address Space Diagram



Example 3: Page Table Mechanism



Summary of Binding and Memory Management Properties

address binding time:

- compile time: relocating program requires recompilation
- load time: compiler produces relocatable code
- dynamic (run time): hardware MMU performs translation

physical memory allocation:

- fixed or dynamic partitions
- fixed size partitions (frames) or variable size partitions

physical contiguity:

- virtual space is contiguous or uncontiguous in physical space

Physical Memory Allocation

fixed allocation size:

- space tracking and placement are simple
- *internal* fragmentation

variable allocation size:

- space tracking and placement more complex
  - placement heuristics: first fit, best fit, worst fit
- *external* fragmentation

Memory Protection

- ensure that each process accesses only the physical memory that its virtual address space is bound to.
  - threat: virtual address is too large
  - solution: MMU *limit register* checks each virtual address
    - for simple dynamic relocation, limit register contains the maximum virtual address of the running process
    - for paging, limit register contains the maximum page number of the running process
  - MMU generates exception if the limit is exceeded
- restrict the use of some portions of an address space
  - example: read-only memory
  - approach (paging):
    - include read-only flag in each page table entry
    - MMU raises exception on attempt to write to a read-only page

Roles of the Operating System and the MMU (Summary)

- operating system:
  - save/restore MMU state on context switches
  - handle exceptions raised by the MMU
  - manage and allocate physical memory
- MMU (hardware):
  - translate virtual addresses to physical addresses
  - check for protection violations
  - raise exceptions when necessary

Speed of Address Translation

- Execution of each machine instruction may involve one, two or more memory operations
  - one to fetch instruction
  - one or more for instruction operands
- Address translation through a page table adds one extra memory operation (for page table entry lookup) for each memory operation performed during instruction execution
  - Simple address translation through a page table can cut instruction execution rate in half.
  - More complex translation schemes (e.g., multi-level paging) are even more expensive.
- Solution: include a Translation Lookaside Buffer (TLB) in the MMU
  - TLB is a fast, fully associative address translation cache
  - TLB hit avoids page table lookup

TLB

- Each entry in the TLB contains a (page number,frame number) pair, plus copies of some or all of the page's protection bits, use bit, and dirty bit.
- If address translation can be accomplished using a TLB entry, access to the page table is avoided.
- TLB lookup is much faster than a memory access. TLB is an associative memory - page numbers of all entries are checked simultaneously for a match. However, the TLB is typically small (10^2 to 10^3 entries).
- Otherwise, translate through the page table, and add the resulting translation to the TLB, replacing an existing entry if necessary. In a *hardware controlled* TLB, this is done by the MMU. In a *software controlled* TLB, it is done by the kernel.
- On a context switch, the kernel must clear or invalidate the TLB. (Why?)

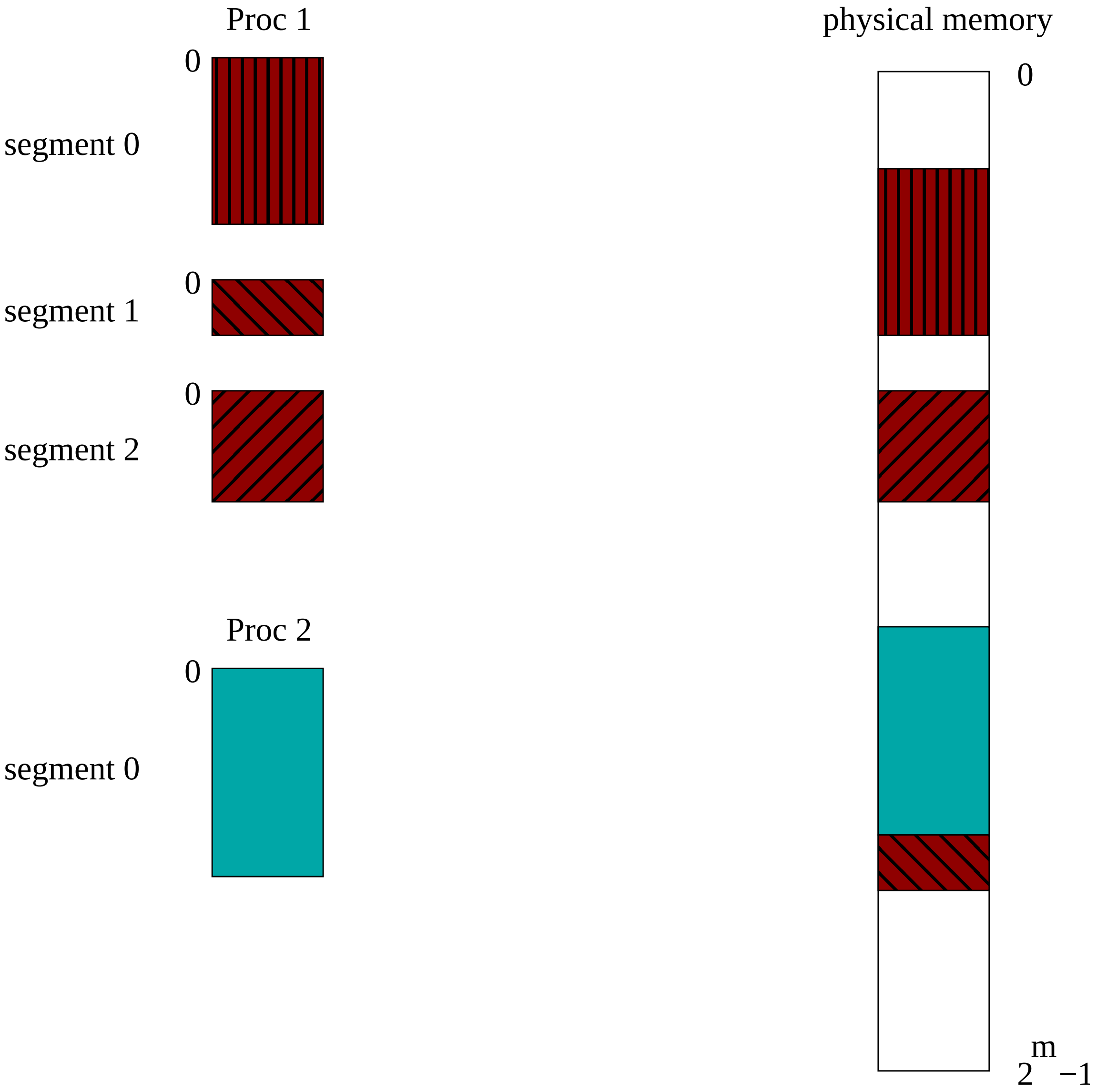
Segmentation

- An OS that supports segmentation (e.g., Multics, OS/2) can provide more than one address space to each process.
- The individual address spaces are called *segments*.
- A logical address consists of two parts:

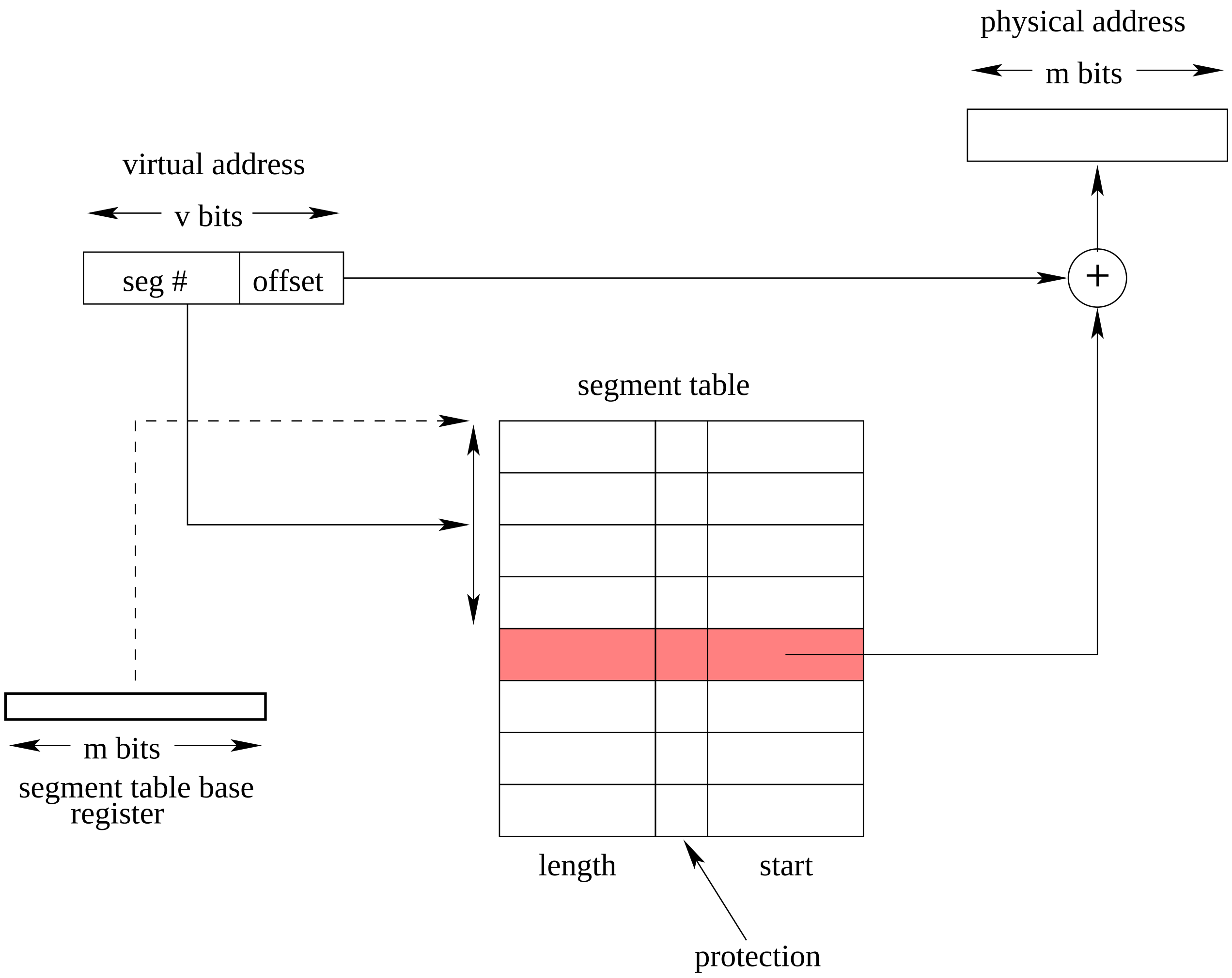
  (segment ID, address within segment)

- Each segment:
  - can grow or shrink independently of the other segments
  - has its own memory protection attributes
- For example, process could use separate segments for code, data, and stack.

Segmented Address Space Diagram

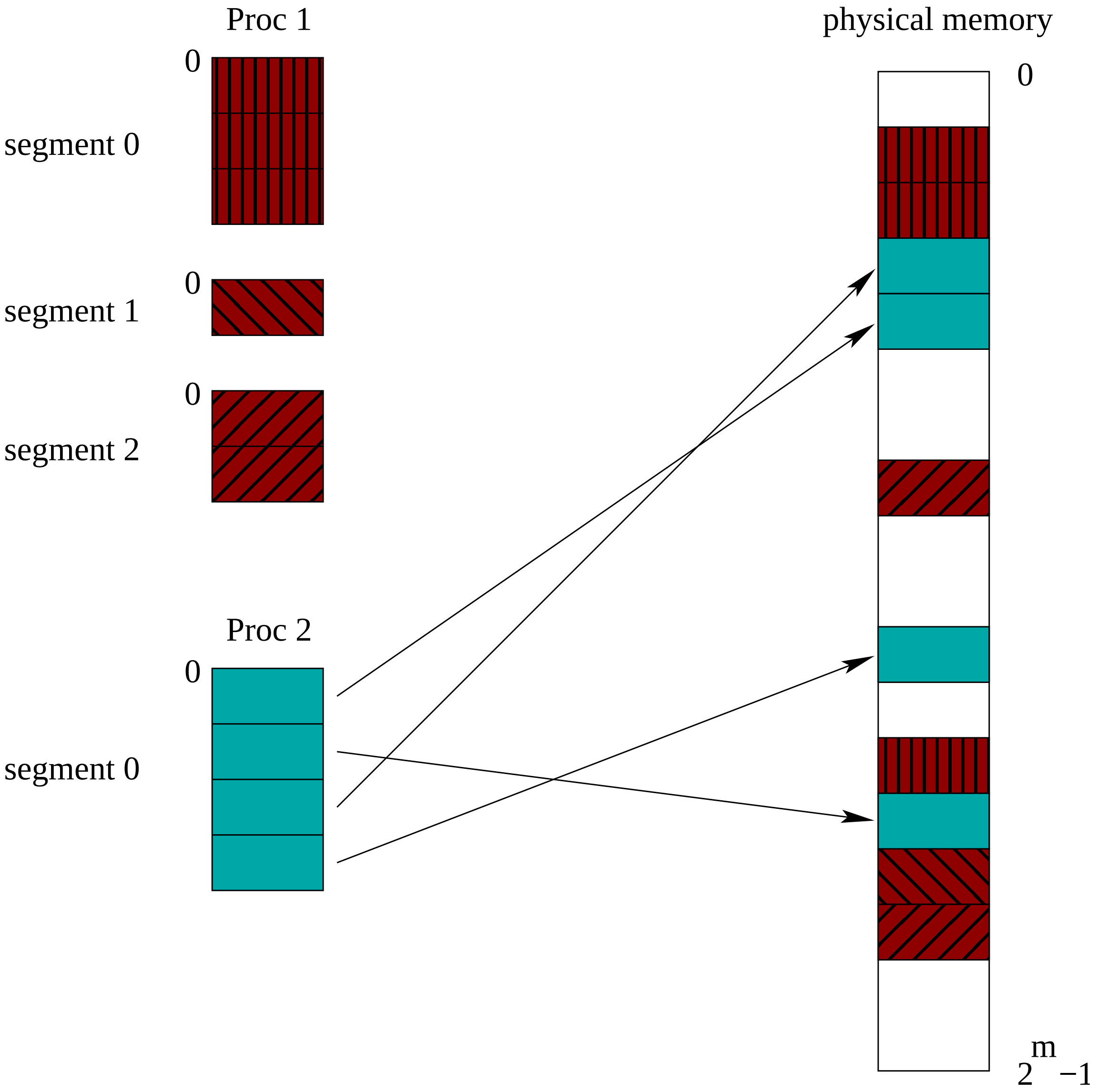


Mechanism for Translating Segmented Addresses

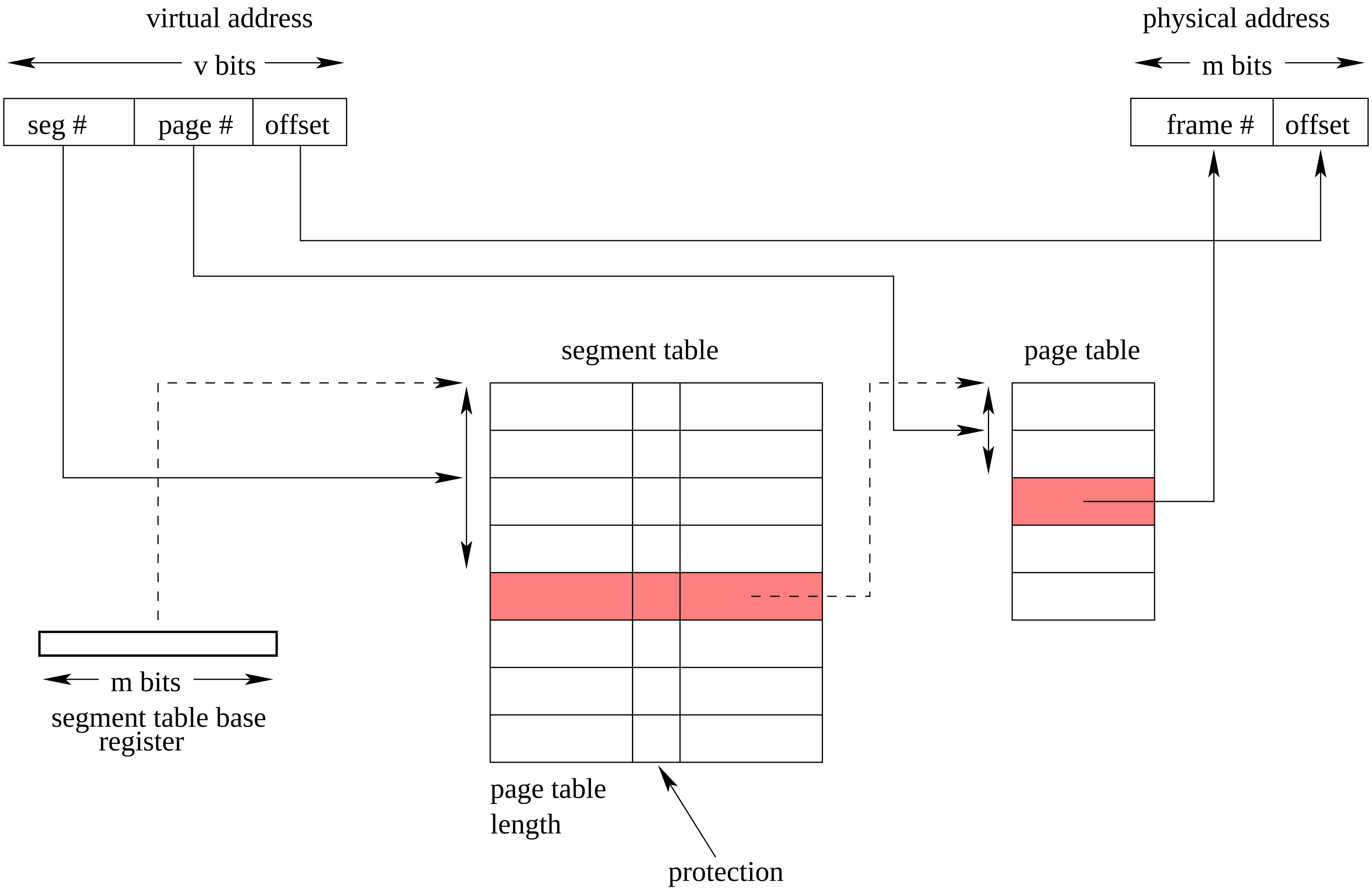


This translation mechanism requires physically contiguous allocation of segments.

Combining Segmentation and Paging

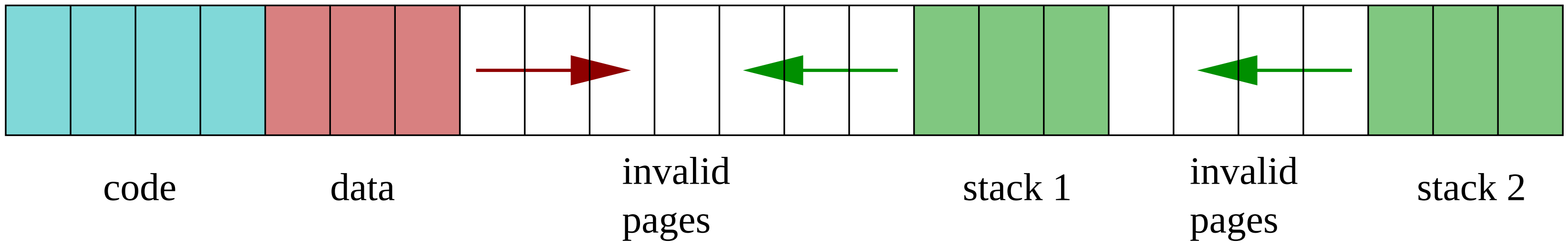


Combining Segmentation and Paging: Translation Mechanism



Simulating Segmentation with Paging

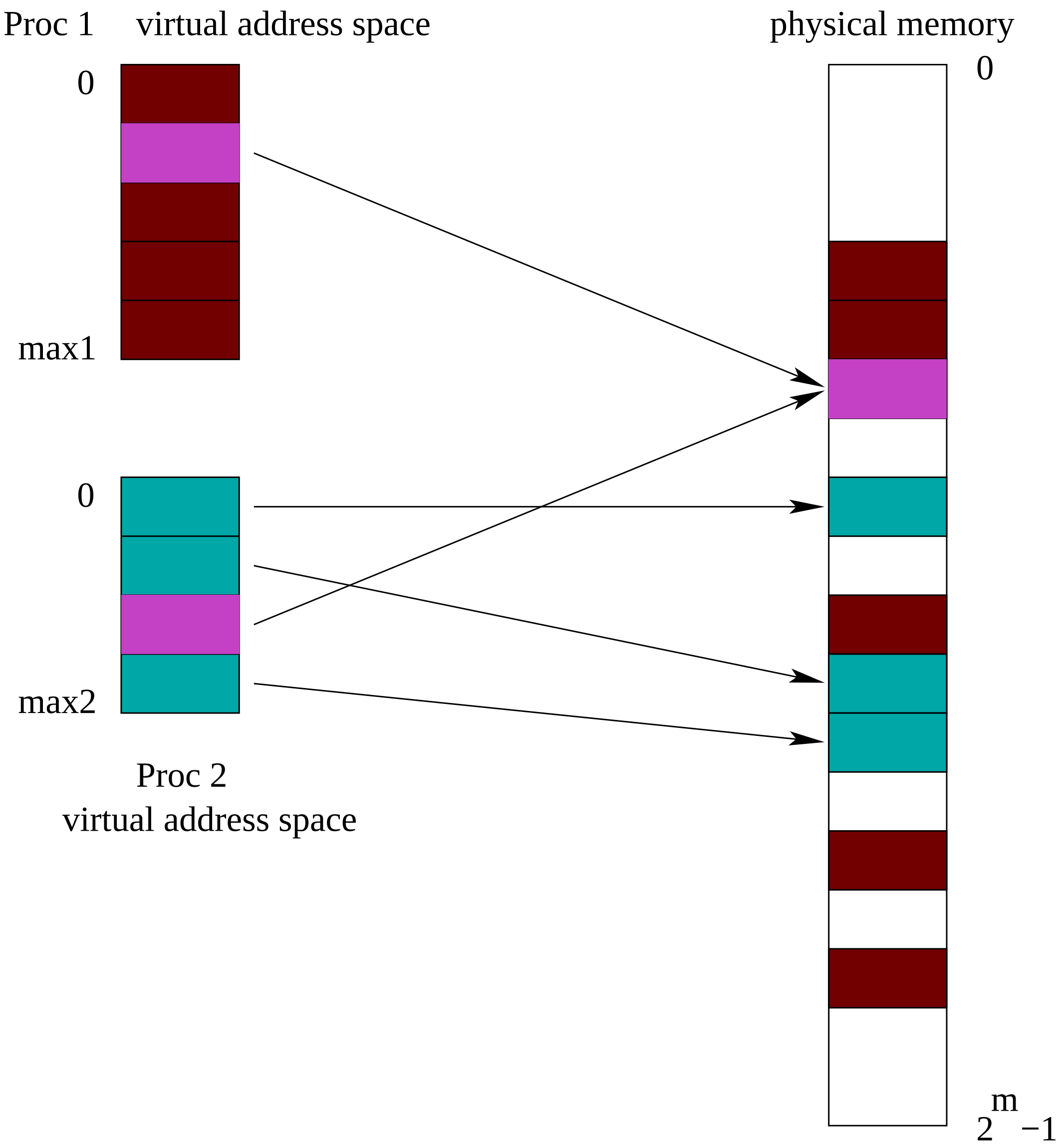
virtual address space



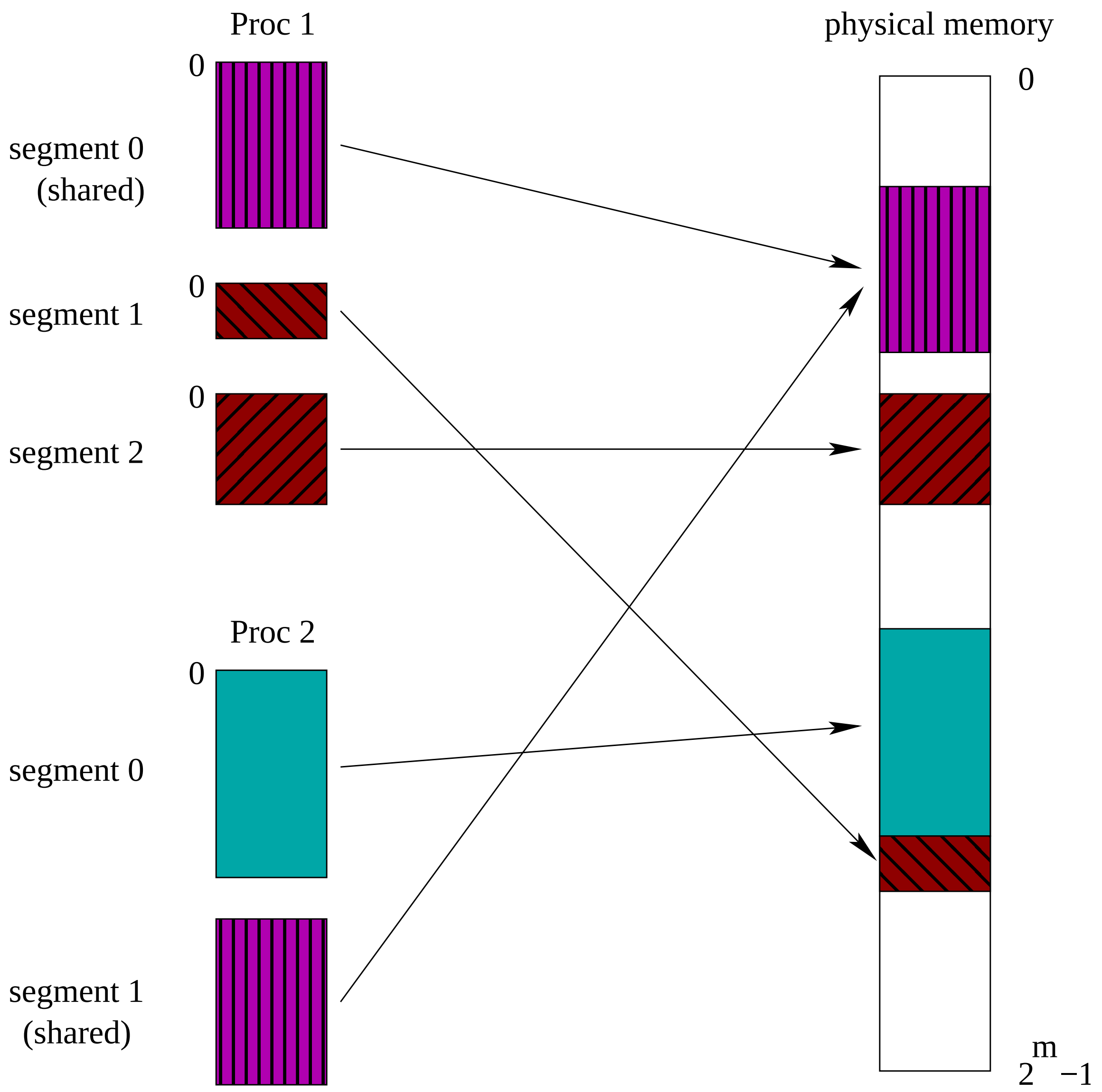
Shared Virtual Memory

- virtual memory sharing allows parts of two or more address spaces to overlap
- shared virtual memory is:
  - a way to use physical memory more efficiently, e.g., one copy of a program can be shared by several processes
  - a mechanism for interprocess communication
- sharing is accomplished by mapping virtual addresses from several processes to the same physical address
- unit of sharing can be a page or a segment

Shared Pages Diagram



Shared Segments Diagram



An Address Space for the Kernel

Option 1: Kernel in physical space

- mechanism: disable MMU in system mode, enable it in user mode
- accessing process address spaces: OS must interpret process page tables
- OS must be entirely memory resident

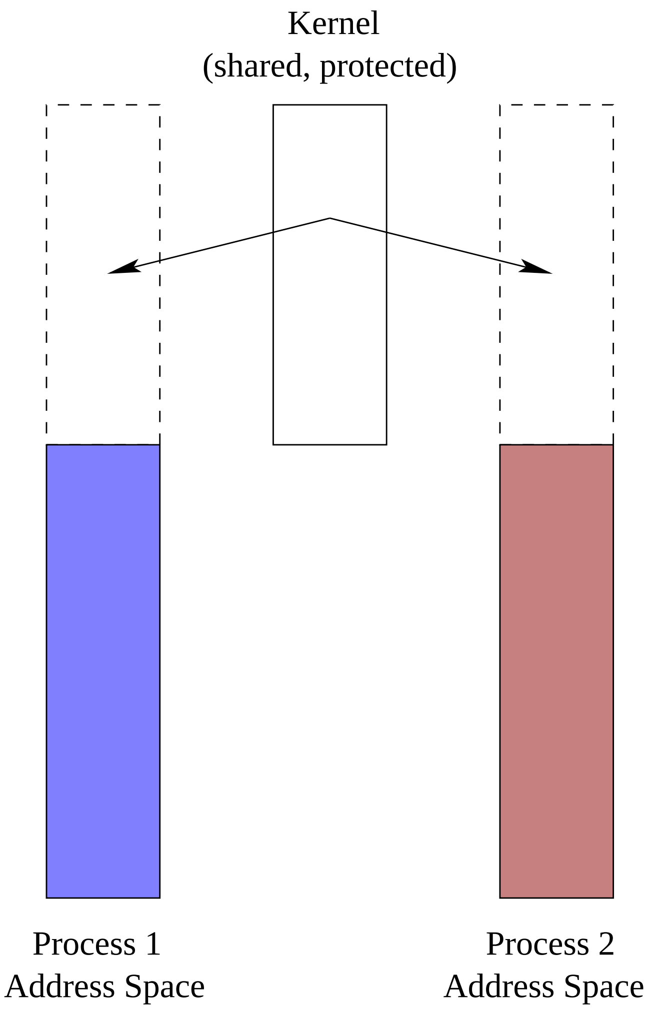
Option 2: Kernel in separate logical address space

- mechanism: MMU has separate state for user and system modes
- accessing process address spaces: difficult
- portions of the OS may be non-resident

Option 3: Kernel shares logical space with each process

- memory protection mechanism is used to isolate the OS
- accessing process address space: easy (process and kernel share the same address space)
- portions of the OS may be non-resident

The Kernel in Process' Address Spaces



Attempts to access kernel code/data in user mode result in memory protection exceptions, not invalid address exceptions.

Memory Management Interface

- much memory allocation is implicit, e.g.:
  - allocation for address space of new process
  - implicit stack growth on overflow
- OS may support explicit requests to grow/shrink address space, e.g., Unix `brk` system call.
- shared virtual memory (simplified Solaris example):

Create: `shmid = shmget(key,size)`

Attach: `vaddr = shmat(shmid, vaddr)`

Detach: `shmdt(vaddr)`

Delete: `shmctl(shmid,IPC_RMID)`