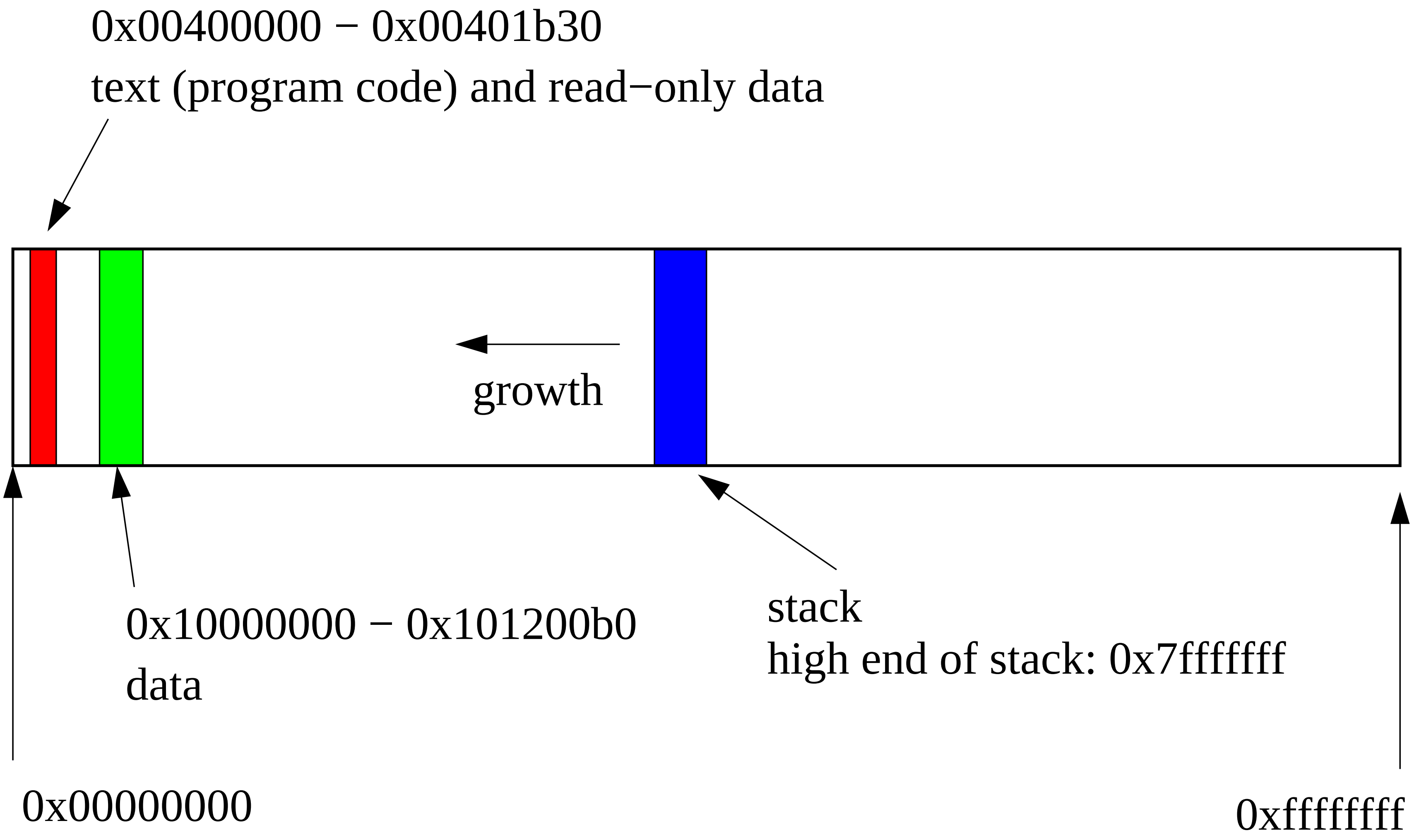
Virtual and Physical Addresses

- Physical addresses are provided directly by the machine.
  - one physical address space per machine
  - the size of a physical address determines the maximum amount of addressable physical memory
- Virtual addresses (or logical addresses) are addresses provided by the OS to processes.
  - one virtual address space *per process*
- Programs use virtual addresses. As a program runs, the hardware (with help from the operating system) converts each virtual address to a physical address.
- the conversion of a virtual address to a physical address is called *address translation*

On the MIPS, virtual addresses and physical addresses are 32 bits long. This limits the size of virtual and physical address spaces.

What is in a Virtual Address Space?

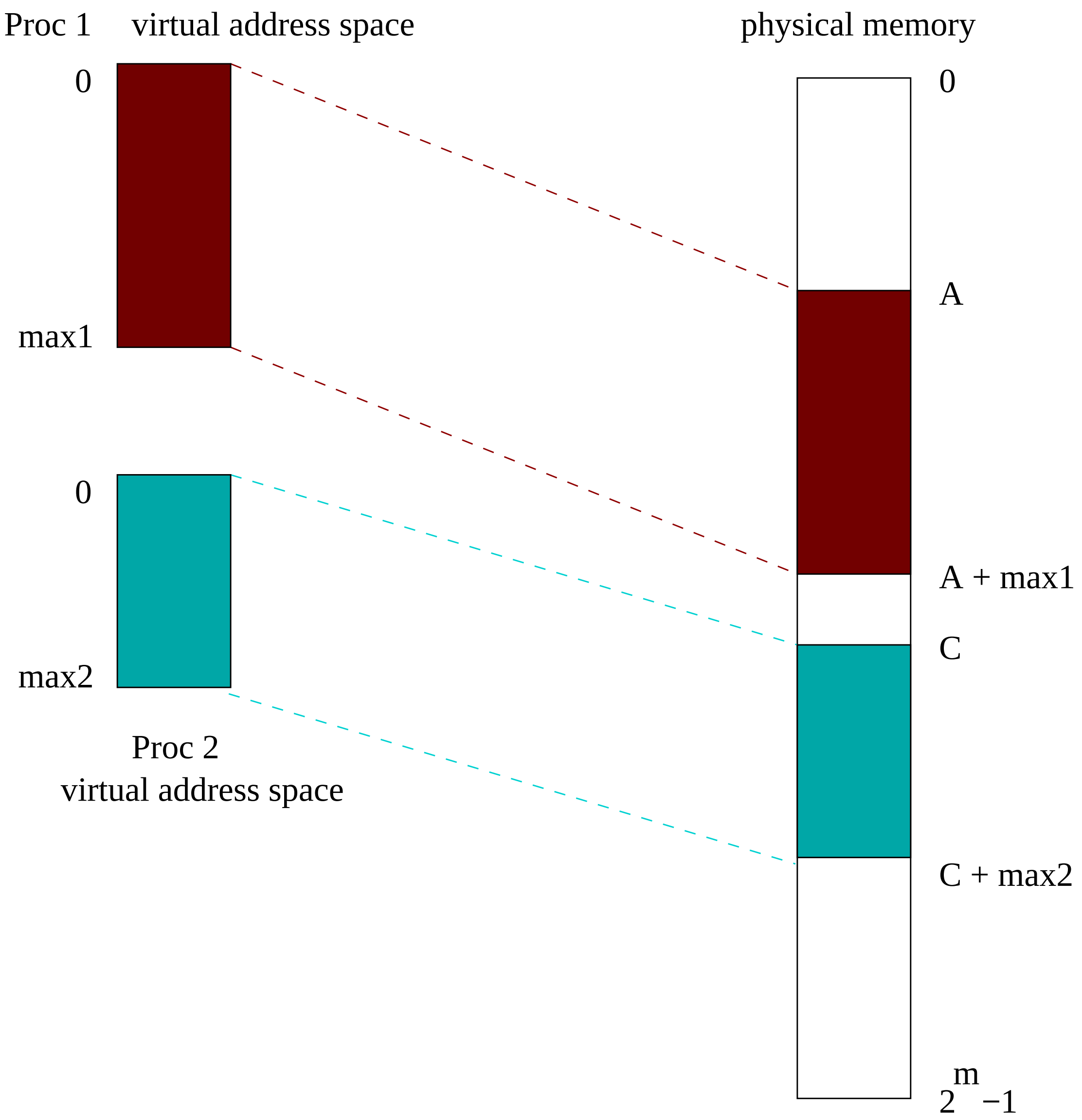


This diagram illustrates the layout of the virtual address space for the OS/161 test application `testbin/sort`

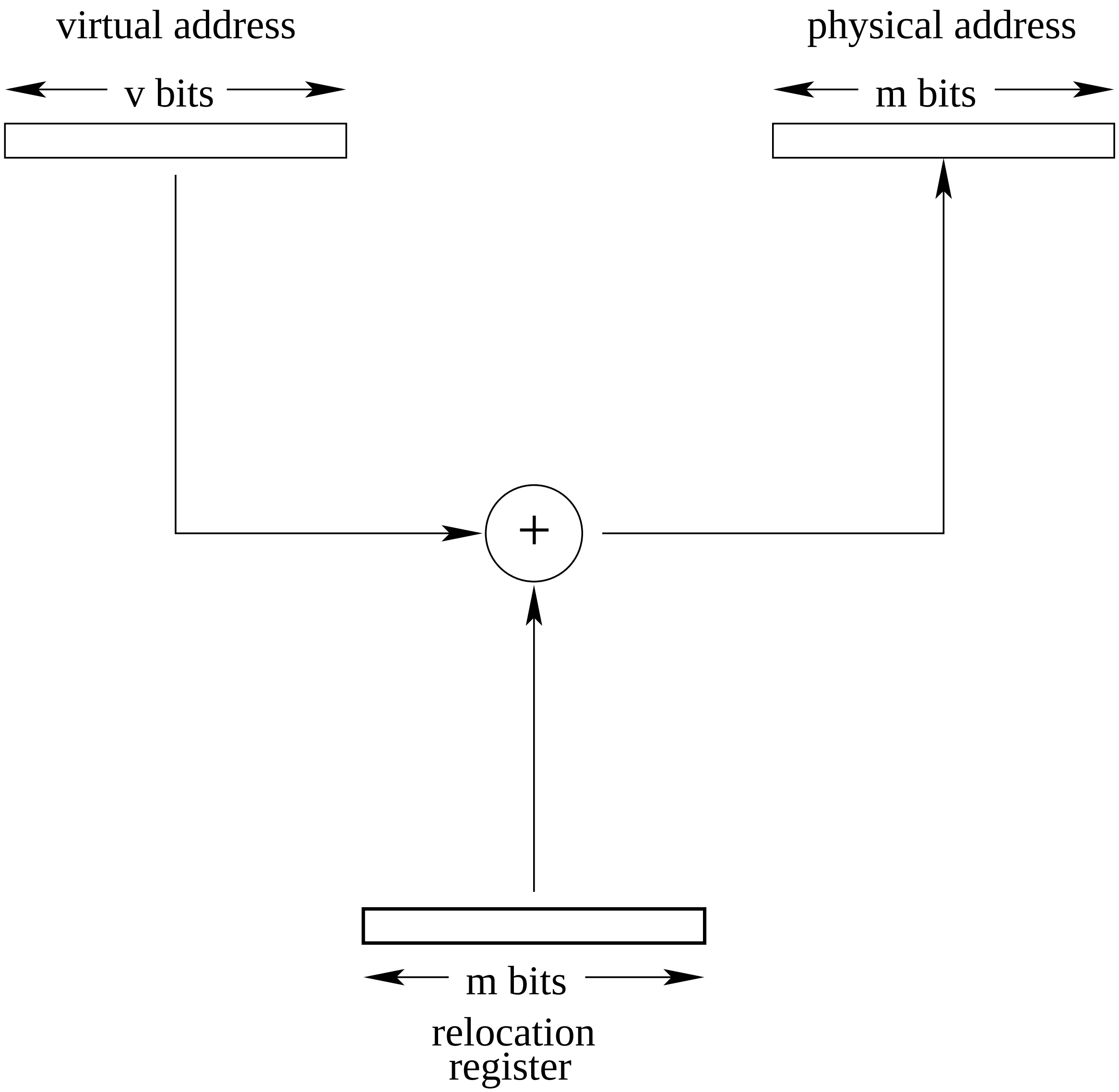
Simple Address Translation: Dynamic Relocation

- hardware provides a *memory management unit* which includes a *relocation register*
- at run-time, the contents of the relocation register are added to each virtual address to determine the corresponding physical address
- the OS maintains a separate relocation register value for each process, and ensures that relocation register is reset on each context switch
- Properties
  - OS must allocate/deallocate variable-sized chunks of physical memory
  - potential for *external fragmentation* of physical memory: wasted, unallocated space
  - each virtual address space corresponds to a contiguous range of physical addresses

Dynamic Relocation: Address Space Diagram



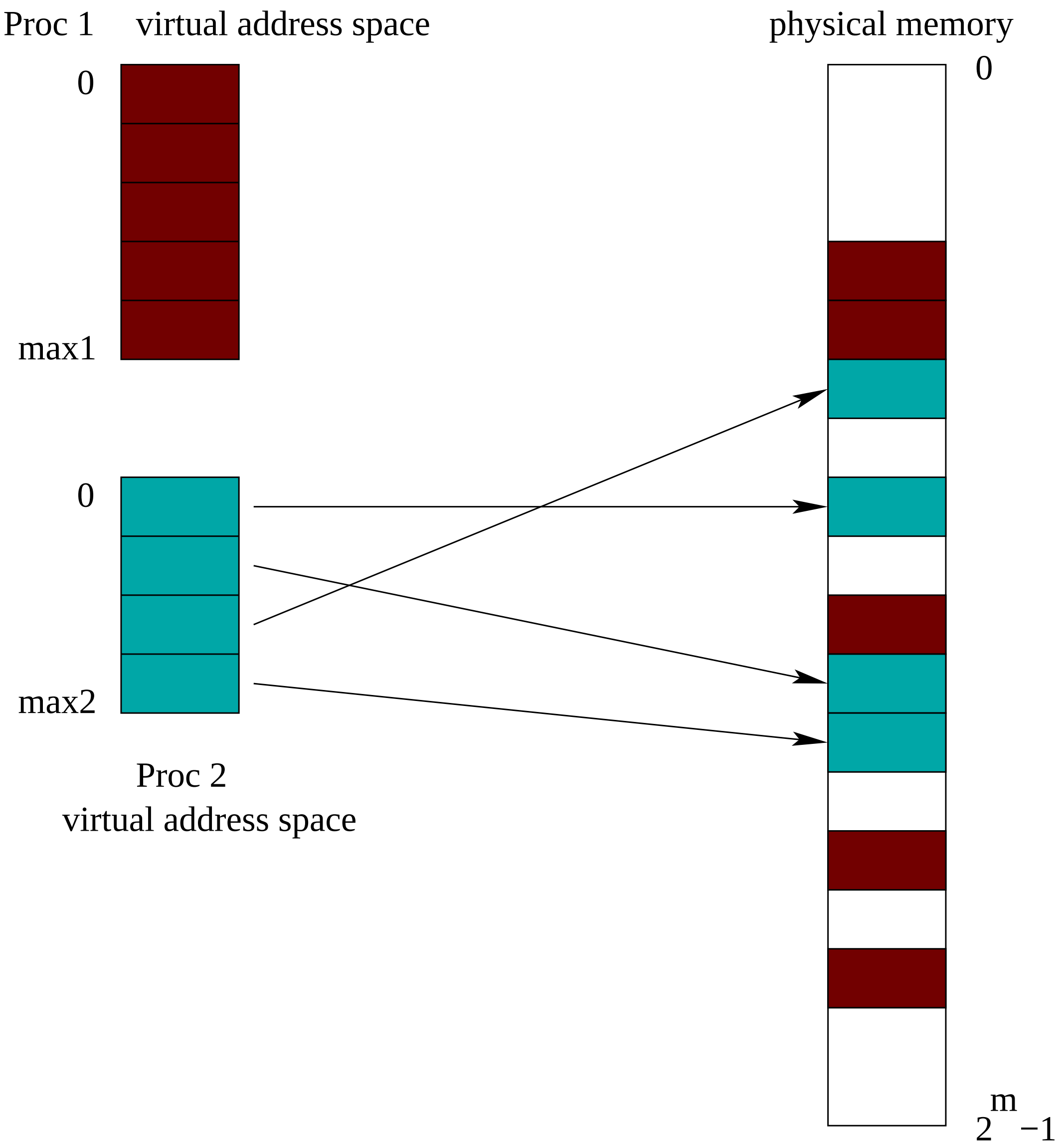
Dynamic Relocation Mechanism



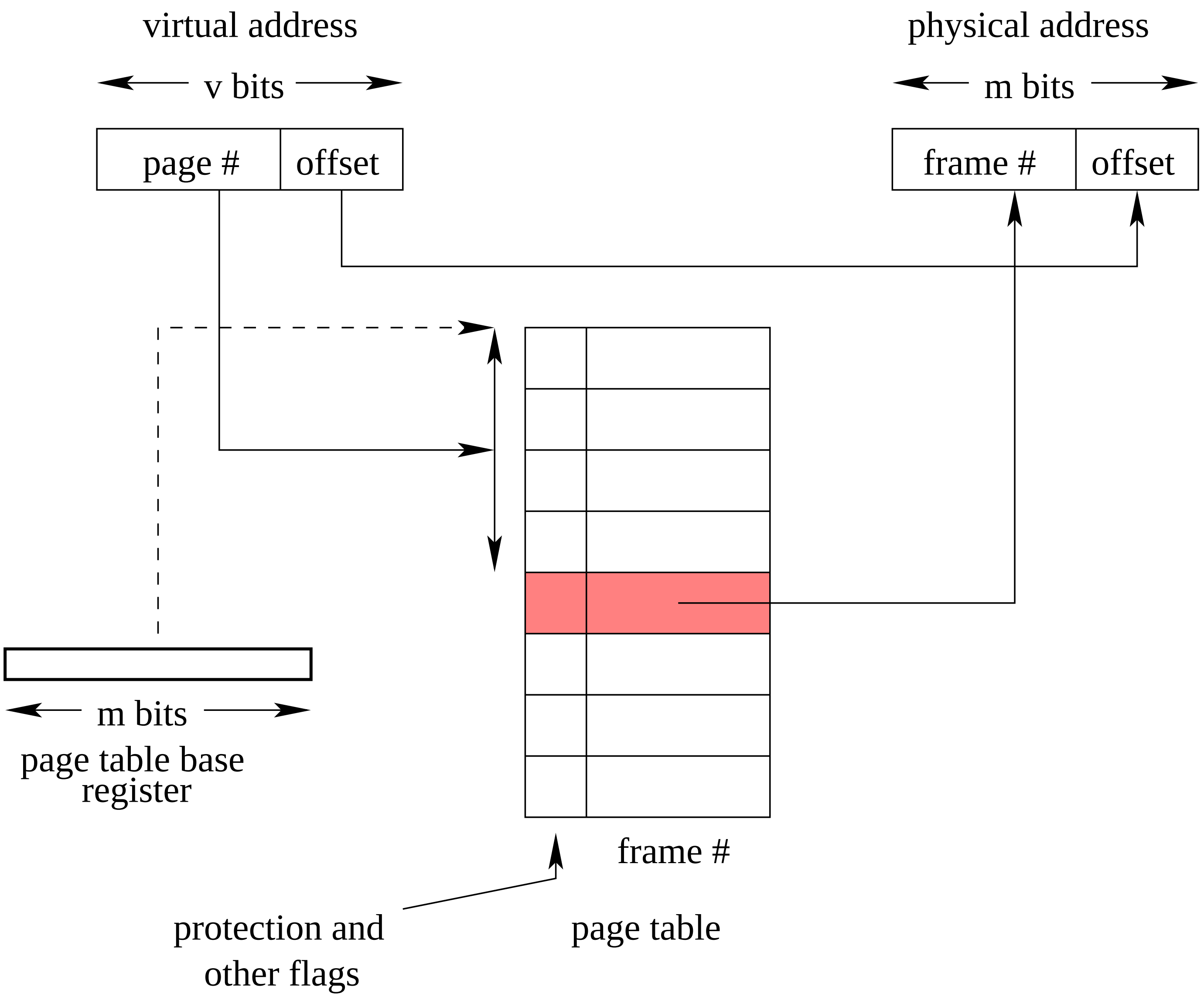
Address Translation: Paging

- 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
  - potential for *internal fragmentation* of physical memory: wasted, allocated space
  - virtual address space need not be physically contiguous in physical space after translation.

Address Space Diagram for Paging



Paging Mechanism



Memory Protection

- during address translation, the MMU checks to ensure that the process uses only *valid* virtual addresses
  - typically, each PTE contains a *valid bit* which indicates whether that PTE contains a valid page mapping
  - the MMU may also check that the virtual page number does not index a PTE beyond the end of the page table
- the MMU may also enforce other protection rules
  - typically, a *read-only* bit each PTE may be set to specify that the corresponding page may not be modified by the process
- if a process attempts to violated these protection rules, the MMU raises an exception, which is handled by the kernel

The kernel controls which pages are valid and which are protected by setting the the contents of PTEs and/or MMU registers.

Roles of the Operating System and the MMU (Summary)

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

Remaining Issues

translation speed: Address translation happens very frequently. (How frequently?) It must be fast.

sparseness: Many programs will only need a small part of the available space for their code and data.

the kernel: Each process has a virtual address space in which to run. What about the kernel? In which address space does it run?

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.
- If address translation can be accomplished using a TLB entry, access to the page table is avoided.
- 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.
- 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).
- If the MMU cannot distinguish TLB entries from different address spaces, then the kernel must clear or invalidate the TLB. (Why?)

The MIPS R3000 TLB

- The MIPS has a software-controlled TLB than can hold 64 entries.
- Each TLB entry includes a virtual page number, a physical frame number, an address space identifier (not used by OS/161), and several flags (valid, read-only)
- OS/161 provides low-level functions for managing the TLB:

  TLB_Write: modify a specified TLB entry

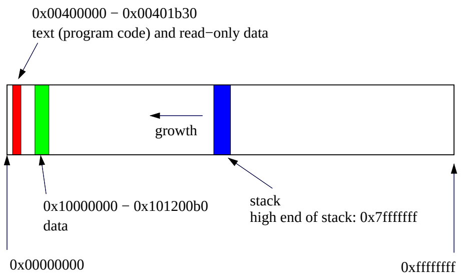
  TLB_Random: modify a random TLB entry

  TLB_Read: read a specified TLB entry

  TLB_Probe: look for a page number in the TLB
- If the MMU cannot translate a virtual address using the TLB it raises an exception, which must be handled by OS/161

See `kern/arch/mips/include/tlb.h`

Handling Sparse Address Spaces: Sparse Page Tables



- Consider the page table for `testbin/sort`, assuming a 4 Kbyte page size:
  - need 2^{19} page table entries (PTEs) to cover the bottom half of the virtual address space.
  - the text segment occupies 2 pages, the data segment occupies 288 pages, and OS/161 sets the initial stack size to 12 pages
- The kernel will mark a PTE as invalid if its page is not mapped.
- In the page table for `testbin/sort`, only 302 of 2^{19} PTEs will be valid.

An attempt by a process to access an invalid page causes the MMU to generate an exception (known as a *page fault*) which must be handled by the operating system.

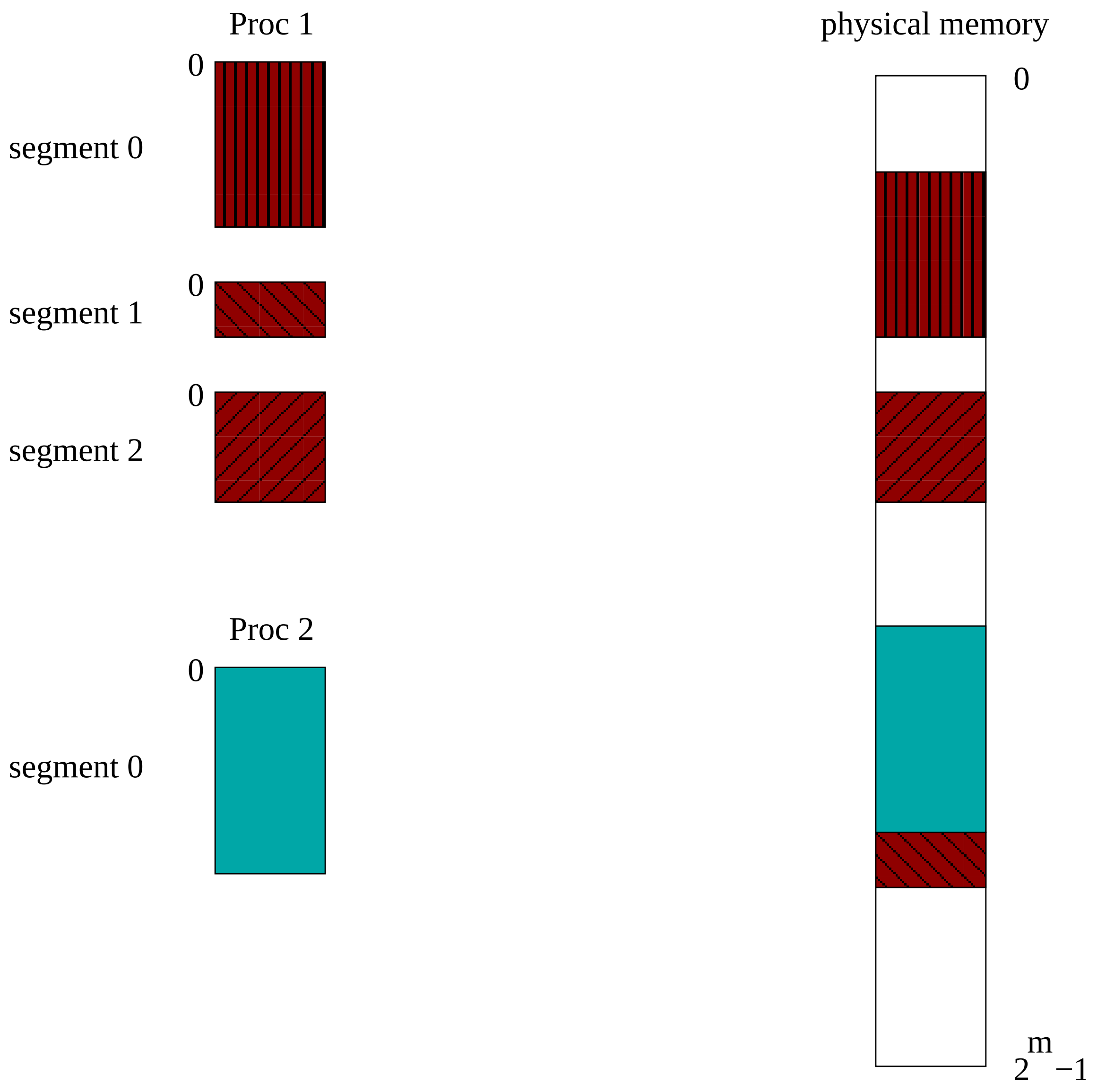
Segmentation

- Often, programs (like `sort`) need several virtual address segments, e.g, for code, data, and stack.
- One way to support this is to turn *segments* into first-class citizens, understood by the application and directly supported by the OS and the MMU.
- Instead of providing a single virtual address space to each process, the OS provides multiple virtual segments. Each segment is like a separate virtual address space, with addresses that start at zero.
- With segmentation, a process virtual address can be thought of as having two parts:

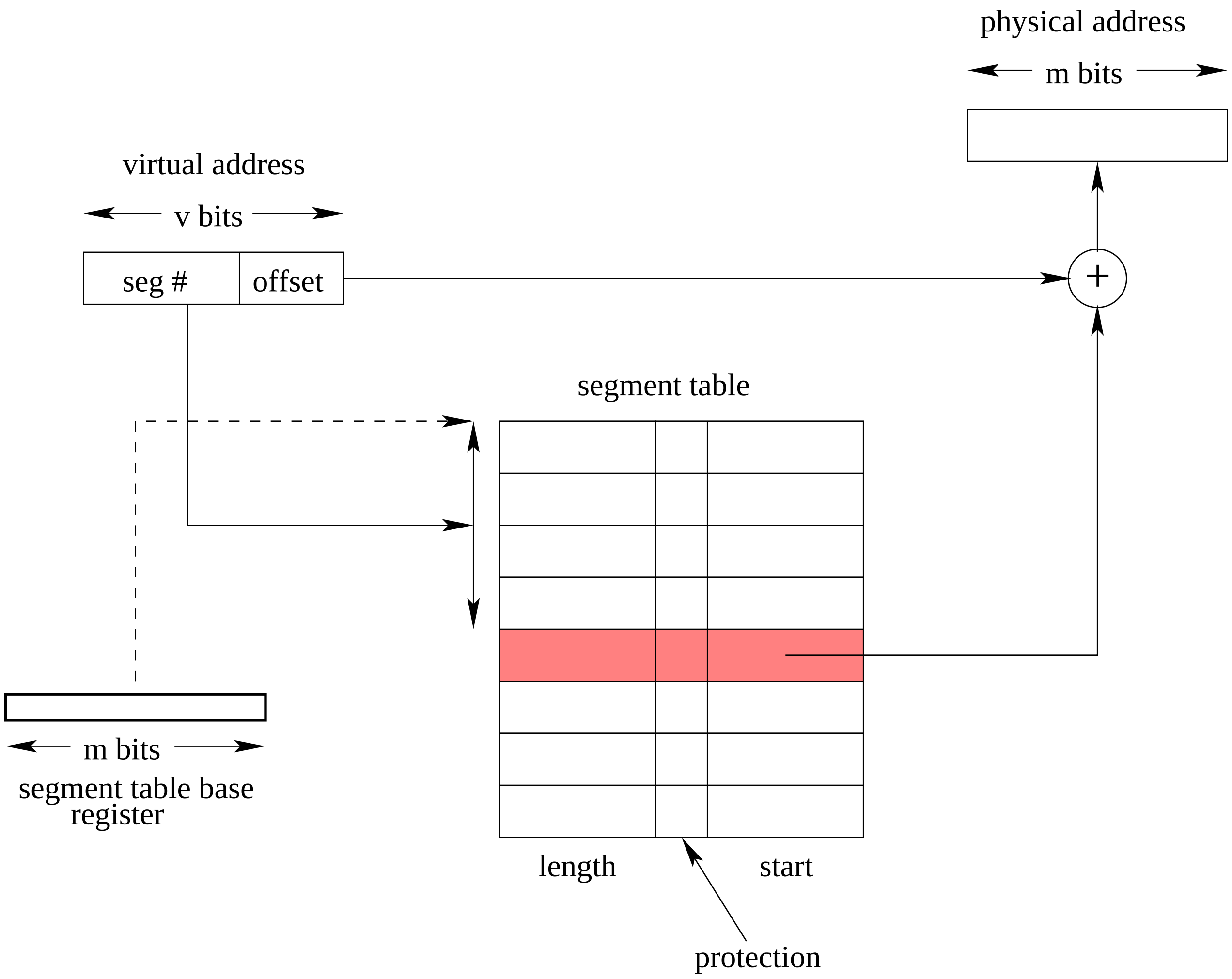
  (segment ID, address within segment)

- Each segment:
  - can grow (or shrink) independently of the other segments, up to some maximum size
  - has its own attributes, e.g, read-only protection

Segmented Address Space Diagram

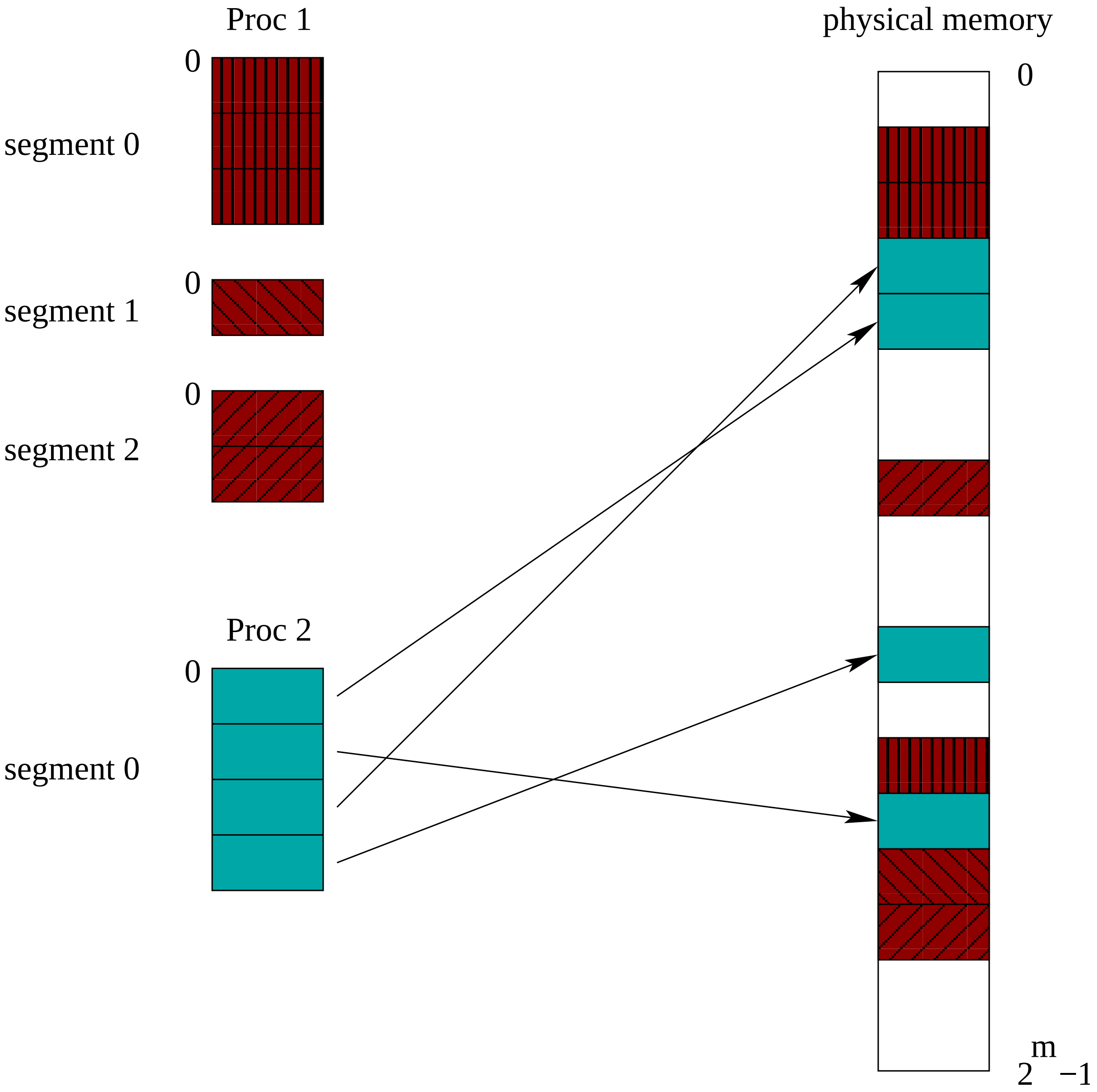


Mechanism for Translating Segmented Addresses

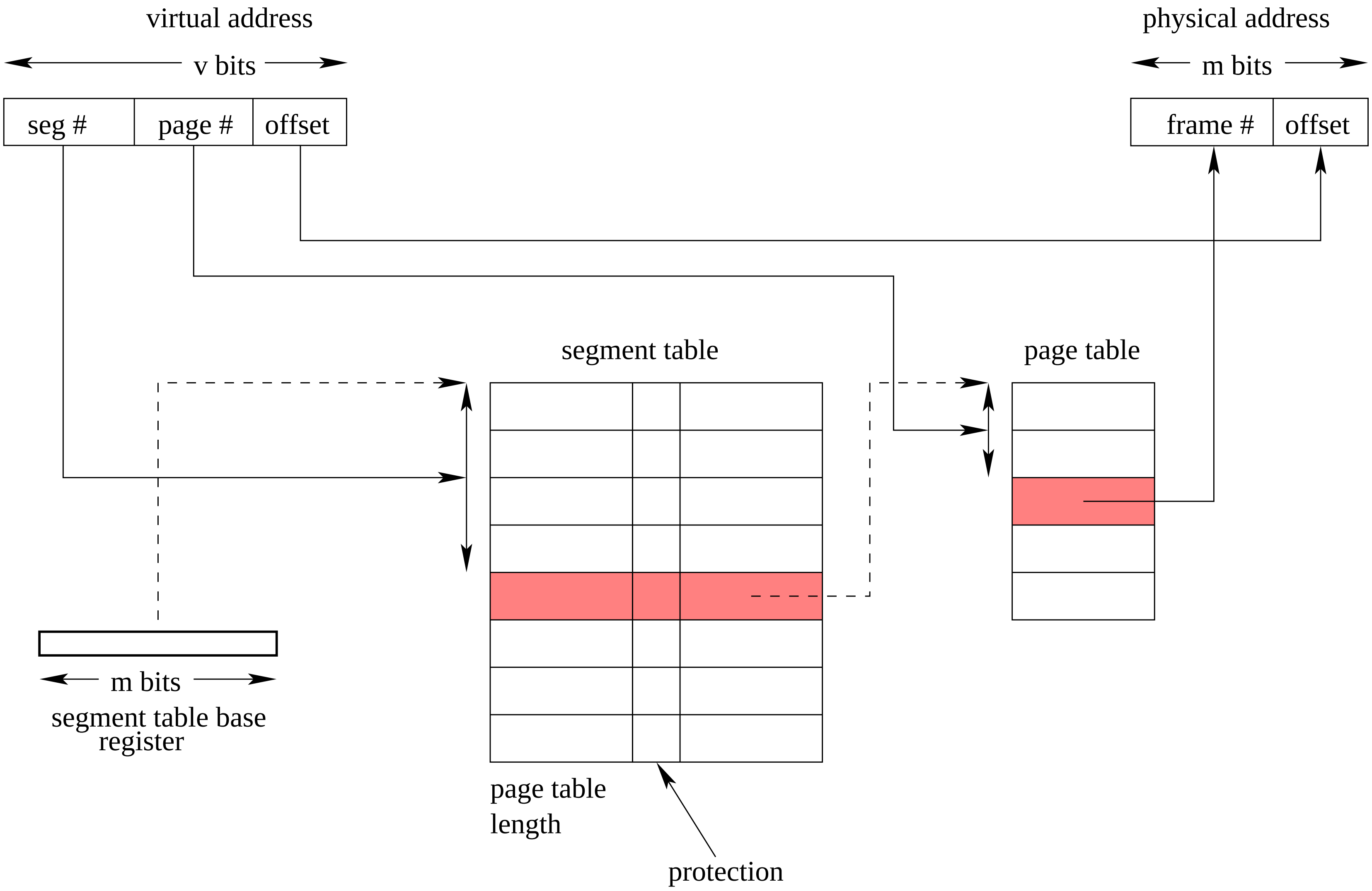


This translation mechanism requires physically contiguous allocation of segments.

Combining Segmentation and Paging



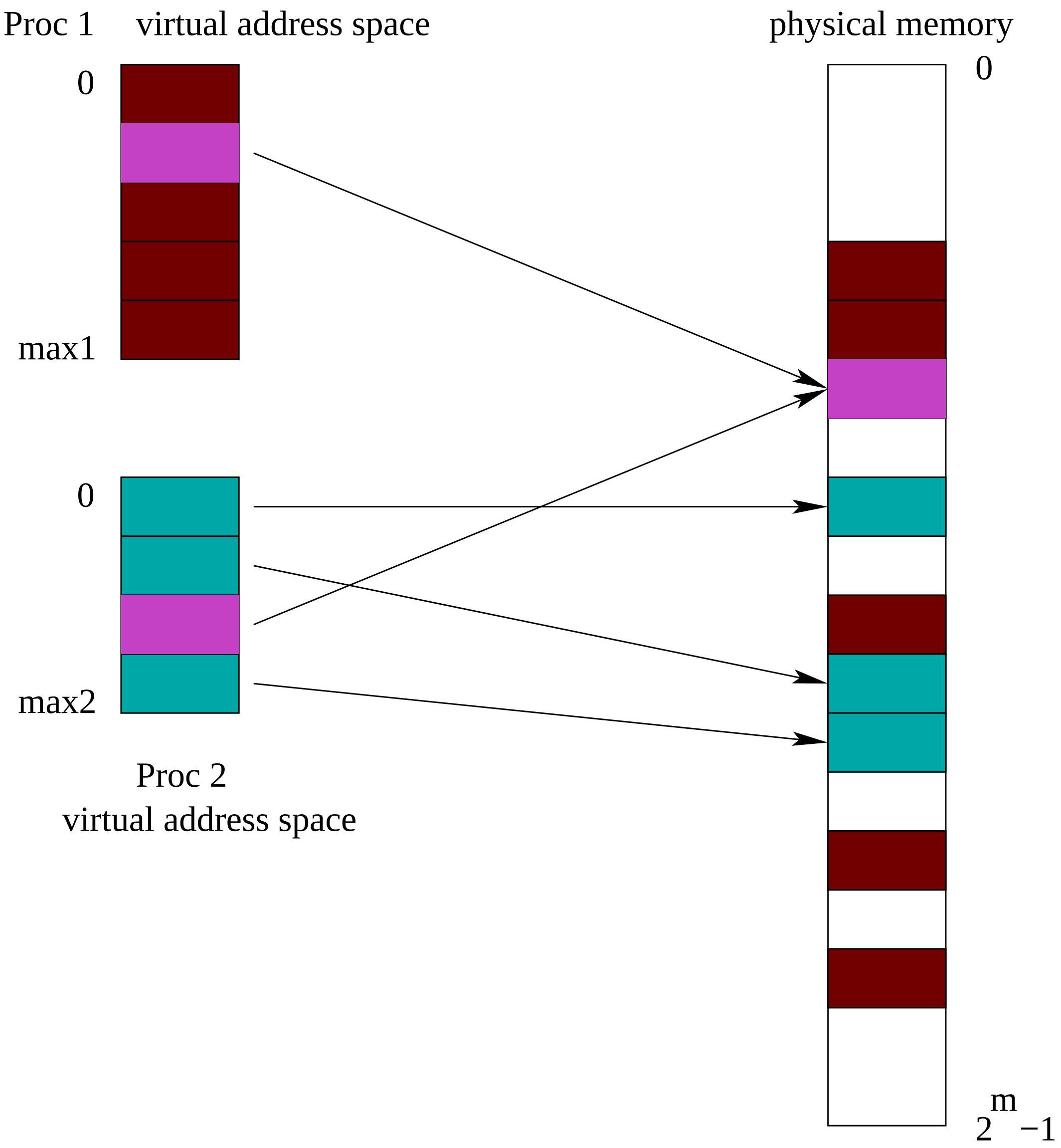
Combining Segmentation and Paging: Translation Mechanism



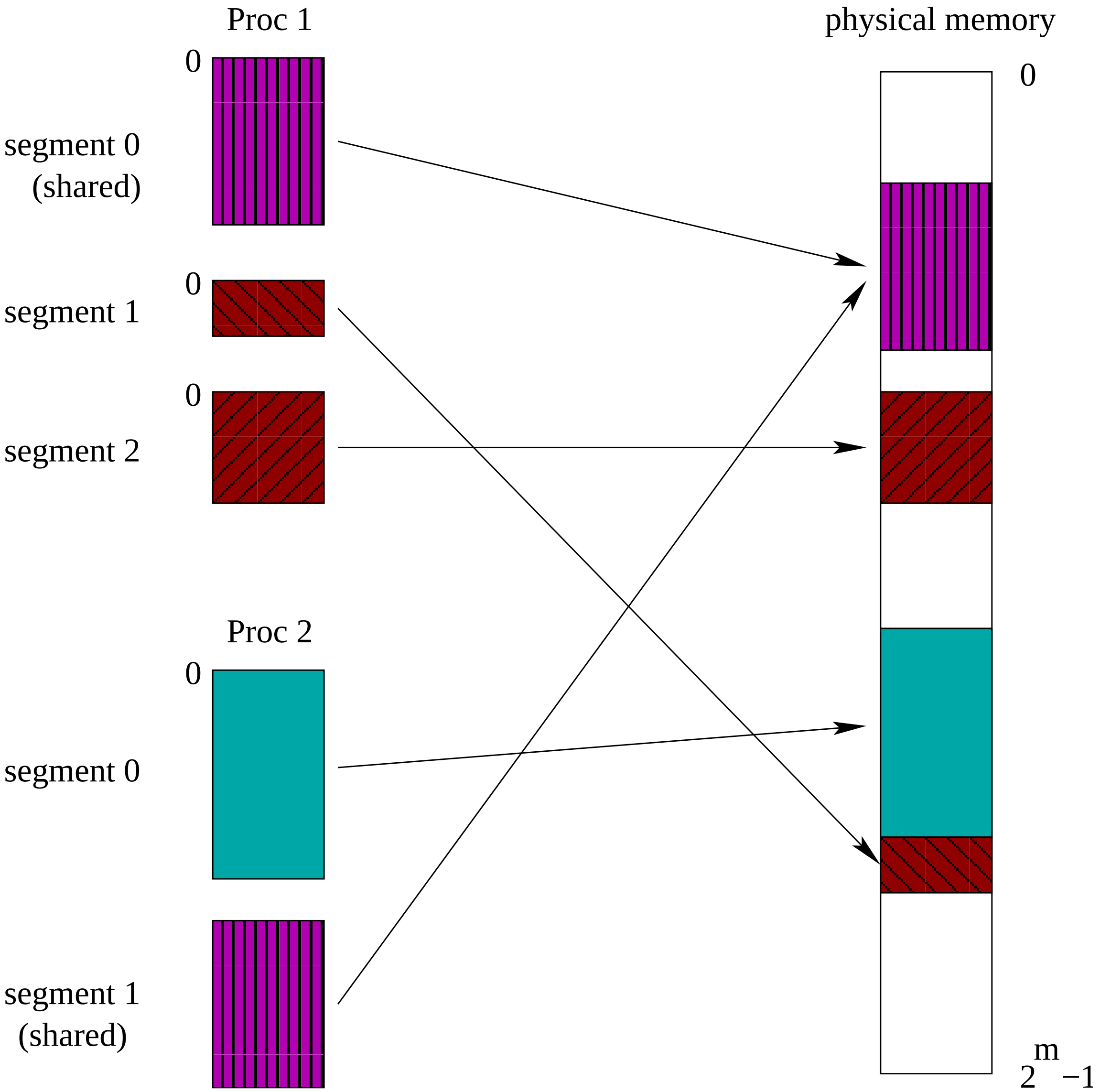
Shared Virtual Memory

- 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



OS/161 Address Spaces: dumbvm

- OS/161 starts with a very simple virtual memory implementation
- virtual address spaces are described by `addrspace` objects, which record the mappings from virtual to physical addresses

```
struct addrspace {
#if OPT_DUMBVM
  vaddr_t as_vbase1; /* base virtual address of code segment */
  paddr_t as_pbase1; /* base physical address of code segment */
  size_t as_npages1; /* size (in pages) of code segment */
  vaddr_t as_vbase2; /* base virtual address of data segment */
  paddr_t as_pbase2; /* base physical address of data segment */
  size_t as_npages2; /* size (in pages) of data segment */
  paddr_t as_stackpbase; /* base physical address of stack */
#else
  /* Put stuff here for your VM system */
#endif
};
```

This amounts to a slightly generalized version of simple dynamic relocation, with three bases rather than one.

Address Translation Under `dumbvm`

- the MIPS MMU tries to translate each virtual address using the entries in the TLB
- If there is no valid entry for the page the MMU is trying to translate, the MMU generates a page fault (called an *address exception*)
- The `vm_fault` function (see `kern/arch/mips/mips/dumbvm.c`) handles this exception for the OS/161 kernel. It uses information from the current process' `addrspace` to construct and load a TLB entry for the page.
- On return from exception, the MIPS retries the instruction that caused the page fault. This time, it may succeed.

`vm_fault` is not very sophisticated. If the TLB fills up, OS/161 will crash!

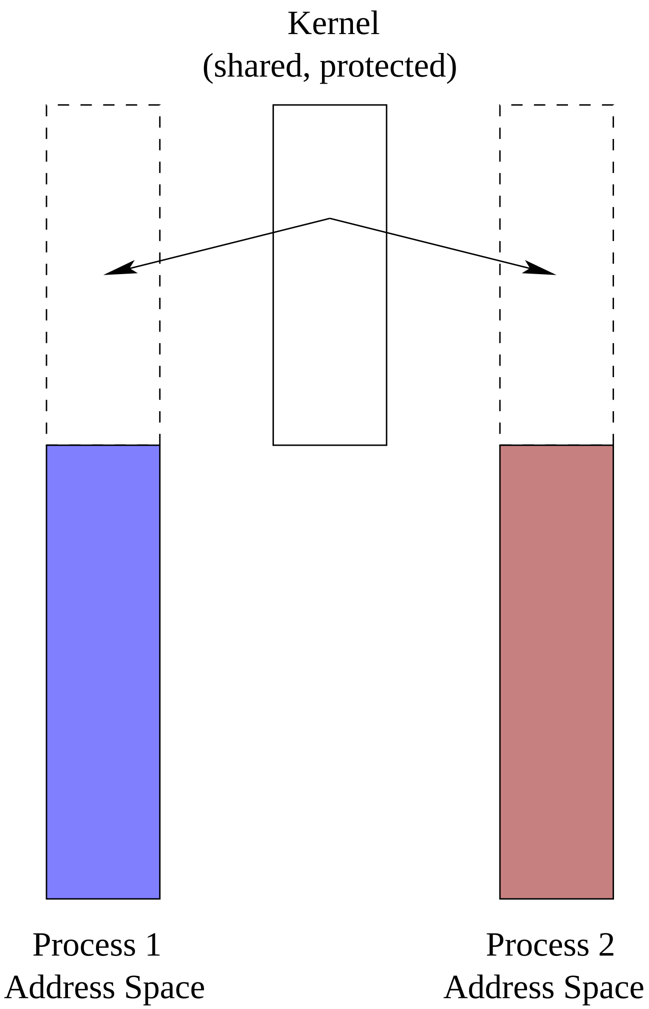
An Address Space for the Kernel

- Each process has its own address space. What about the kernel?
- two possibilities

  Kernel in physical space: disable address translation in privileged system execution mode, enable it in unprivileged mode

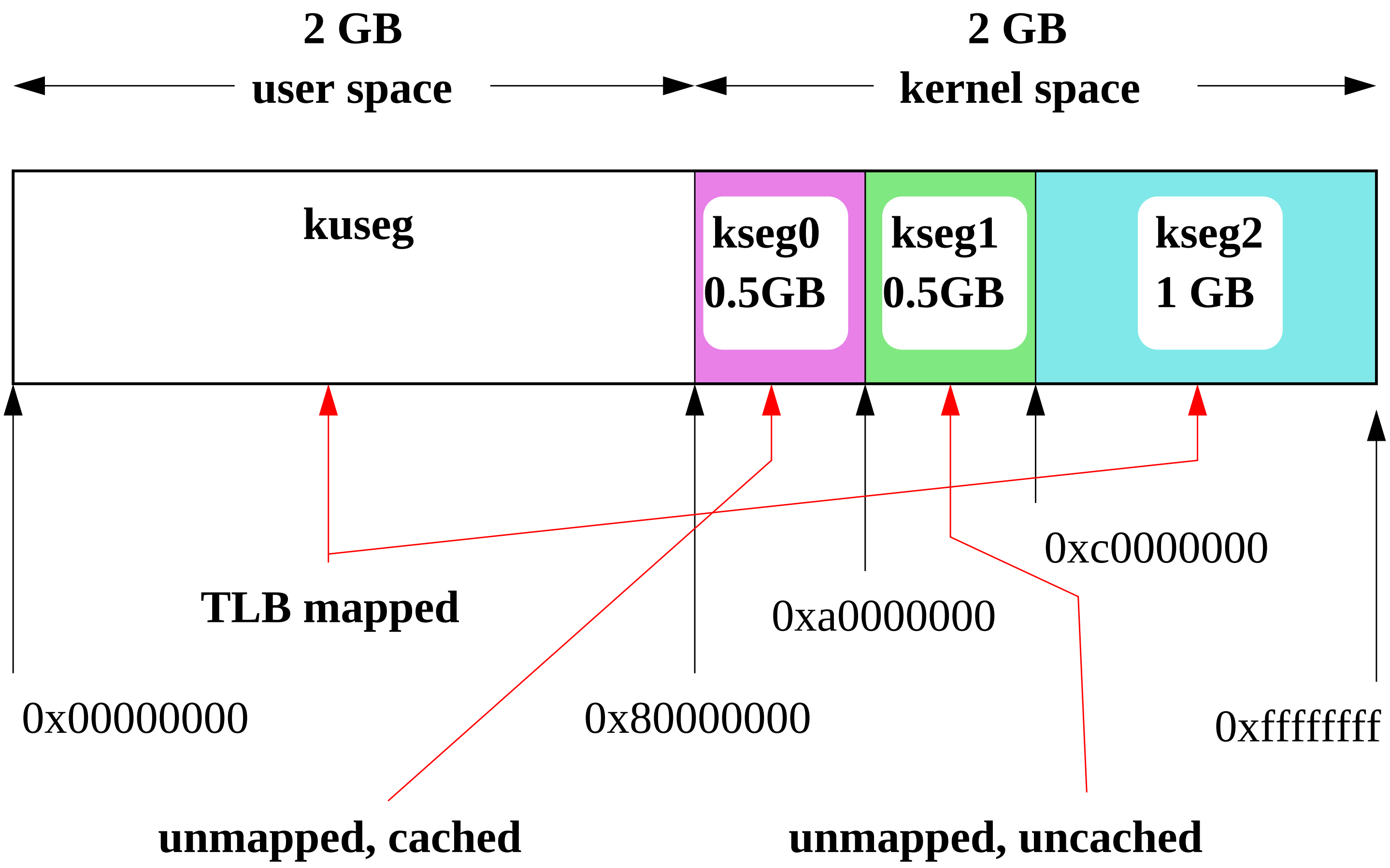
  Kernel in separate virtual address space: need a way to change address translation (e.g., switch page tables) when moving between privileged and unprivileged code
- OS/161, Linux, and other operating systems use a third approach: the kernel is mapped into a portion of the virtual address space of *every process*
- memory protection mechanism is used to isolate the kernel from applications
- one advantage of this approach: application virtual addresses (e.g., system call parameters) are easy for the kernel to use

The Kernel in Process' Address Spaces



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

Address Translation on the MIPS R3000



In OS/161, user programs live in kuseg, kernel code and data structures live in kseg0, devices are accessed through kseg1, and kseg2 is not used.

Loading a Program into an Address Space

- When the kernel creates a process to run a particular program, it must create an address space for the process, and load the program's code and data into that address space
- A program's code and data is described in an *executable file*, which is created when the program is compiled and linked
- OS/161 (and other operating systems) expect executable files to be in ELF (**E**xecutable and **L**inking **F**ormat) format
- the OS/161 `execv` system call, which re-initializes the address space of a process

```
#include <unistd.h>
int
execv(const char *program, char **args)
```

- The `program` parameter of the `execv` system call should be the name of the ELF executable file for the program that is to be loaded into the address space.

ELF Files

- ELF files contain address space segment descriptions, which are useful to the kernel when it is loading a new address space
- the ELF file identifies the (virtual) address of the program's first instruction
- the ELF file also contains lots of other information (e.g., section descriptors, symbol tables) that is useful to compilers, linkers, debuggers, loaders and other tools used to build programs

Address Space Segments in ELF Files

- Each ELF segment describes a contiguous region of the virtual address space.
- For each segment, the ELF file includes a segment *image* and a header, which describes:
  - the virtual address of the start of the segment
  - the length of the segment in the virtual address space
  - the location of the start of the image in the ELF file
  - the length of the image in the ELF file
- the image is an exact copy of the binary data that should be loaded into the specified portion of the virtual address space
- the image may be smaller than the address space segment, in which case the rest of the address space segment is expected to be zero-filled

To initialize an address space, the kernel copies images from the ELF file to the specifed portions of the virtual address space

ELF Files and OS/161

- OS/161's `dumbvm` implementation assumes that an ELF file contains two segments:
  - a *text segment*, containing the program code and any read-only data
  - a *data segment*, containing any other global program data
- the ELF file does not describe the stack (why not?)
- `dumbvm` creates a *stack segment* for each process. It is 12 pages long, ending at virtual address `0x7fffffff`

Look at `kern/userprog/loadelf.c` to see how OS/161 loads segments from ELF files

ELF Sections and Segments

- In the ELF file, a program's code and data are grouped together into *sections*, based on their properties. Some sections:

  .text: program code

  .rodata: read-only global data

  .data: initialized global data

  .bss: uninitialized global data (Block Started by Symbol)

  .sbss: small uninitialized global data

- not all of these sections are present in every ELF file
- normally
  - the `.text` and `.rodata` sections together form the text segment
  - the `.data`, `.bss` and `.sbss` sections together form the data segement
- space for *local* program variables is allocated on the stack when the program runs

The `segments.c` Example Program (1 of 2)

```
#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;
};
```

The `segments.c` Example Program (2 of 2)

```
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 */
}
```

ELF Sections for the Example Program

```
Section Headers:
  [Nr] Name      Type          Addr     Off    Size   ES Flg
  [ 0]           NULL          00000000 000000 000000 00
  [ 1] .reginfo  MIPS_REGINFO  00400094 000094 000018 18   A
  [ 2] .text     PROGBITS      004000b0 0000b0 000200 00  AX
  [ 3] .rodata   PROGBITS      004002b0 0002b0 000020 00   A
  [ 4] .data     PROGBITS      10000000 001000 000010 00  WA
  [ 5] .sbss     NOBITS        10000010 001010 000014 00 WAp
  [ 6] .bss      NOBITS        10000030 00101c 004000 00  WA
  [ 7] .comment  PROGBITS      00000000 00101c 000036 00
...
Flags: W (write), A (alloc), X (execute), p (processor specific)

## Size = number of bytes (e.g., .text is 0x200 = 512 bytes
## Off  = offset into the ELF file
## Addr = virtual address
```

The `cs350-readelf` program can be used to inspect OS/161 MIPS ELF files: `cs350-readelf -a segments`

ELF Segments for the Example Program

```
Program Headers:
Type     Offset   VirtAddr   PhysAddr   FileSiz MemSiz  Flg Align
REGINFO 0x000094 0x00400094 0x00400094 0x00018 0x00018 R   0x4
LOAD    0x000000 0x00400000 0x00400000 0x002d0 0x002d0 R E 0x1000
LOAD    0x001000 0x10000000 0x10000000 0x00010 0x04030 RW  0x1000
```

- segment info, like section info, can be inspected using the `cs350-readelf` program
- the REGINFO section is not used
- the first LOAD segment includes the .text and .rodata sections
- the second LOAD segment includes .data, .sbss, and .bss

Contents of the Example Program’s `.text` Section

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

## Decoding 3c1c1001 to determine instruction
## 0x3c1c1001 = binary 111100000111000001000000000001
## 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
```

The `cs350-objdump` program can be used to inspect OS/161 MIPS ELF file section contents: `cs350-objdump -s segments`

Contents of the Example Program’s `.rodata` Section

```
Contents of section .rodata:
 4002b0 48656c6c 6f20576f 726c640a 00000000  Hello World.....
 4002c0 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,
##  then the 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.
```

The `.rodata` section contains the “Hello World” string literal and the constant integer variable `z`.

Contents of the Example Program’s `.data` Section

```
Contents of section .data:
 10000000 deadbeef 004002b0 00000000 00000000  .....@..........

...
## Size = 0x10 bytes = 16 bytes
## int x = deadbeef (4 bytes)
## char const *str = "Hello World\n"; (4 bytes)
## value stored in str = 0x004002b0.
## NOTE: this is the address of the start
## of the string literal in the .rodata section
```

The `.data` section contains the initialized global variables `str` and `x`.

Contents of the Example Program’s `.bss` and `.sbss` Sections

```
...
10000010 A __bss_start
10000010 A _edata
10000010 A _fbss
10000010 S y3   ## S indicates sbss section
10000014 S y2
10000018 S y1
1000001c S errno
10000020 S __argv
...
10000030 B array  ## B indicates bss section
10004030 A _end
```

The `y1`, `y2`, and `y3` variables are in the `.sbss` section. The `array` variable is in the `.bss` section. There are no values for these variables in the ELF file, as they are uninitialized. The `cs350-nm` program can be used to inspect symbols defined in ELF files: `cs350-nm -b segments`

System Call Interface for Virtual Memory Management

- 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)`

Exploiting Secondary Storage

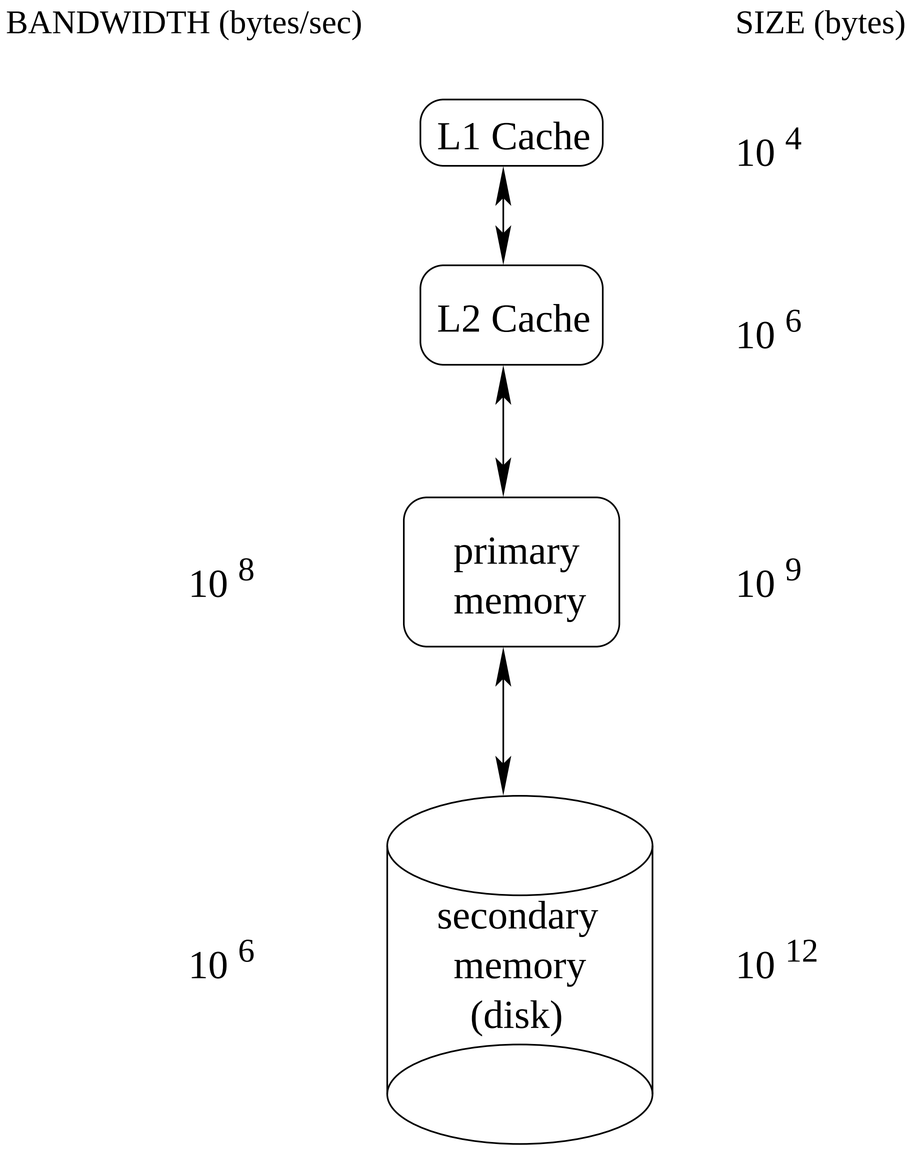
Goals:

- Allow virtual address spaces that are larger than the physical address space.
- Allow greater multiprogramming levels by using less of the available (primary) memory for each process.

Method:

- Allow pages (or segments) from the virtual address space to be stored in secondary memory, as well as primary memory.
- Move pages (or segments) between secondary and primary memory so that they are in primary memory when they are needed.

The Memory Hierarchy



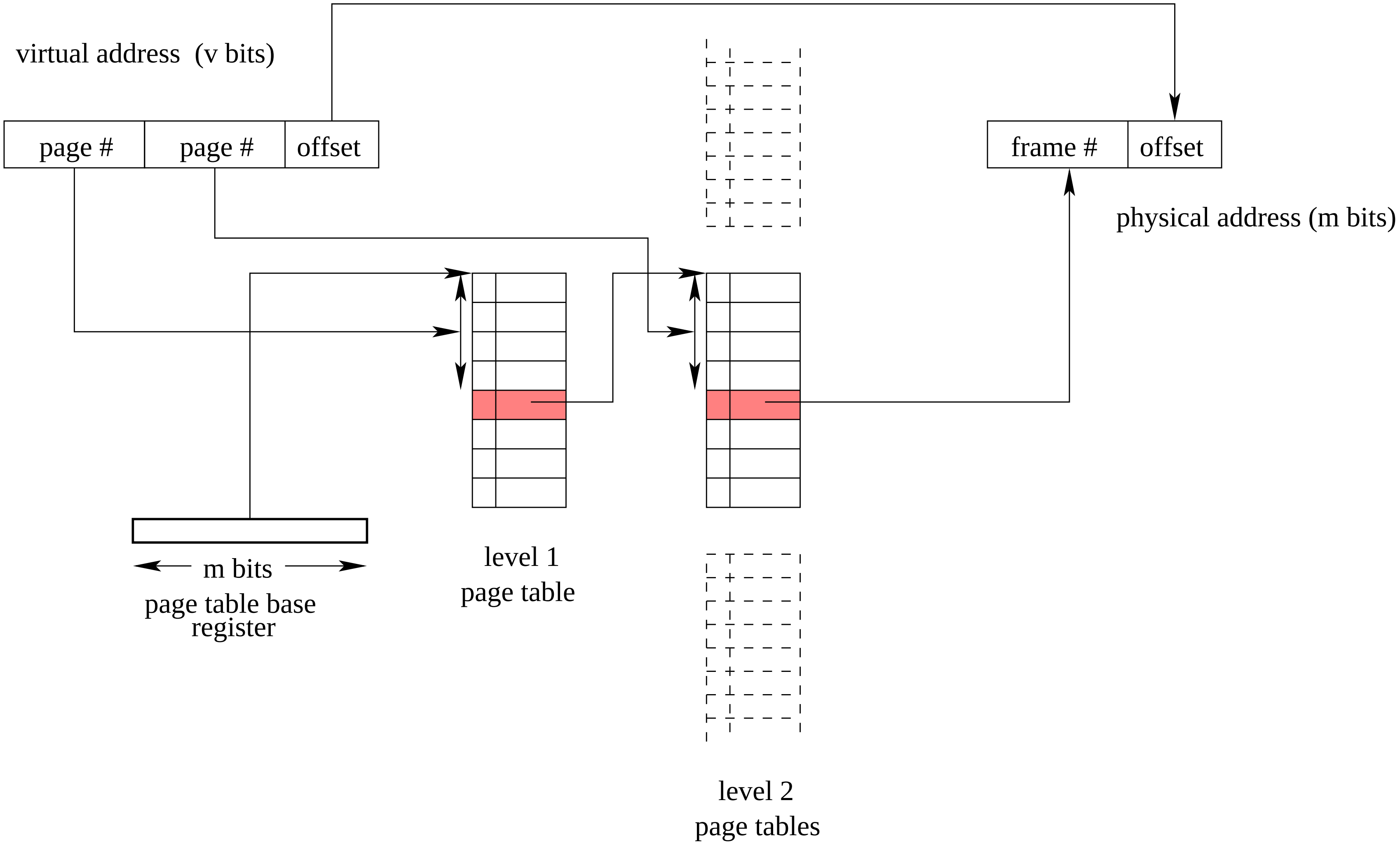
Large Virtual Address Spaces

- Virtual memory allows for very large virtual address spaces, and very large virtual address spaces require large page tables.
- example: 2^{48} byte virtual address space, 8Kbyte (2^{13} byte) pages, 4 byte page table entries means

$$\frac{2^{48}}{2^{13}}2^2 = 2^{37} \text{ bytes per page table}$$

- page tables for large address spaces may be very large, and
  - they must be in memory, and
  - they must be physically contiguous
- some solutions:
  - multi-level page tables - page the page tables
  - inverted page tables

Two-Level Paging



Inverted Page Tables

- A normal page table maps virtual pages to physical frames. An inverted page table maps physical frames to virtual pages.
- Other key differences between normal and inverted page tables:
  - there is only one inverted page table, not one table per process
  - entries in an inverted page table must include a process identifier
- An inverted page table only specifies the location of virtual pages that are located in memory. Some other mechanism (e.g., regular page tables) must be used to locate pages that are not in memory.

Paging Policies

When to Page?:

Demand paging brings pages into memory when they are used. Alternatively, the OS can attempt to guess which pages will be used, and *prefetch* them.

What to Replace?:

Unless there are unused frames, one page must be replaced for each page that is loaded into memory. A *replacement policy* specifies how to determine which page to replace.

Similar issues arise if (pure) segmentation is used, only the unit of data transfer is segments rather than pages. Since segments may vary in size, segmentation also requires a *placement policy*, which specifies where, in memory, a newly-fetched segment should be placed.

Global vs. Local Page Replacement

- When the system's page reference string is generated by more than one process, should the replacement policy take this into account?

  Global Policy: A global policy is applied to all in-memory pages, regardless of the process to which each one "belongs". A page requested by process X may replace a page that belongs another process, Y.

  Local Policy: Under a local policy, the available frames are allocated to processes according to some memory allocation policy. A replacement policy is then applied separately to each process's allocated space. A page requested by process X replaces another page that "belongs" to process X.

Paging Mechanism

- A *valid* bit (V) in each page table entry is used to track which pages are in (primary) memory, and which are not.

  $V = 1$: valid entry which can be used for translation

  $V = 0$: invalid entry. If the MMU encounters an invalid page table entry, it raises a *page fault* exception.

- To handle a page fault exception, the operating system must:
  - Determine which page table entry caused the exception. (In SYS/161, and in real MIPS processors, MMU puts the offending virtual address into a register on the CP0 co-processor (register 8/c0_vaddr/BadVaddr). The kernel can read that register.
  - Ensure that that page is brought into memory.

  On return from the exception handler, the instruction that resulted in the page fault will be retried.

- If (pure) segmentation is being used, there will a valid bit in each segment table entry to indicate whether the segment is in memory.

A Simple Replacement Policy: FIFO

- the FIFO policy: replace the page that has been in memory the longest
- a three-frame example:

Num	1	2	3	4	5	6	7	8	9	10	11	12
Refs	a	b	c	d	a	b	e	a	b	c	d	e
Frame 1	a	a	a	d	d	d	e	e	e	e	e	e
Frame 2		b	b	b	a	a	a	a	a	c	c	c
Frame 3			c	c	c	b	b	b	b	b	d	d
Fault ?	x	x	x	x	x	x	x			x	x	

Optimal Page Replacement

- There is an optimal page replacement policy for demand paging.
- The OPT policy: replace the page that will not be referenced for the longest time.

Num	1	2	3	4	5	6	7	8	9	10	11	12
Refs	a	b	c	d	a	b	e	a	b	c	d	e
Frame 1	a	a	a	a	a	a	a	a	a	c	c	c
Frame 2		b	b	b	b	b	b	b	b	b	d	d
Frame 3			c	d	d	d	e	e	e	e	e	e
Fault ?	x	x	x	x			x			x	x	

- OPT requires knowledge of the future.

Other Replacement Policies

- FIFO is simple, but it does not consider:

  Frequency of Use: how often a page has been used?

  Recency of Use: when was a page last used?

  Cleanliness: has the page been changed while it is in memory?

- The *principle of locality* suggests that usage ought to be considered in a replacement decision.
- Cleanliness may be worth considering for performance reasons.

Locality

- Locality is a property of the page reference string. In other words, it is a property of programs themselves.
- *Temporal locality* says that pages that have been used recently are likely to be used again.
- *Spatial locality* says that pages “close” to those that have been used are likely to be used next.

In practice, page reference strings exhibit strong locality. Why?

Frequency-based Page Replacement

- Another approach to page replacement is to count references to pages. The counts can form the basis of a page replacement decision.
- Example: LFU (Least Frequently Used)
  Replace the page with the smallest reference count.
- Any frequency-based policy requires a reference counting mechanism, e.g., MMU increments a counter each time an in-memory page is referenced.
- Pure frequency-based policies have several potential drawbacks:
  - Old references are never forgotten. This can be addressed by periodically reducing the reference count of every in-memory page.
  - Freshly loaded pages have small reference counts and are likely victims - ignores temporal locality.

Least Recently Used (LRU) Page Replacement

- LRU is based on the principle of temporal locality: replace the page that has not been used for the longest time
- To implement LRU, it is necessary to track each page's recency of use. For example: maintain a list of in-memory pages, and move a page to the front of the list when it is used.
- Although LRU and variants have many applications, LRU is often considered to be impractical for use as a replacement policy in virtual memory systems. Why?

Least Recently Used: LRU

- the same three-frame example:

Num	1	2	3	4	5	6	7	8	9	10	11	12
Refs	a	b	c	d	a	b	e	a	b	c	d	e
Frame 1	a	a	a	d	d	d	e	e	e	c	c	c
Frame 2		b	b	b	a	a	a	a	a	a	d	d
Frame 3			c	c	c	b	b	b	b	b	b	e
Fault ?	x	x	x	x	x	x	x			x	x	x

The “Use” Bit

- A *use bit* (or *reference bit*) is a bit found in each PTE entry that:
  - is set by the MMU each time the page is used, i.e., each time the MMU translates a virtual address on that page
  - can be read and modified by the operating system
  - operating system copies use information into page table
- The use bit provides a small amount of efficiently-maintainable usage information that can be exploited by a page replacement algorithm.

Entries in the MIPS TLB do not include a use bit.

What if the MMU Does Not Provide a “Use” Bit?

- the kernel can emulate the “use” bit, at the cost of extra exceptions
  1. When a page is loaded into memory, mark it as *invalid* (even though it as been loaded) and set its simulated “use” bit to false.
  2. If a program attempts to access the page, an exception will occur.
  3. In its exception handler, the OS sets the page’s simulated “use” bit to “true” and marks the page *valid* so that further accesses do not cause exceptions.
- This technique requires that the OS maintain extra bits of information for each page:
  1. the simulated “use” bit
  2. an “in memory” bit to indicate whether the page is in memory

The Clock Replacement Algorithm

- The clock algorithm (also known as “second chance”) is one of the simplest algorithms that exploits the use bit.
- Clock is identical to FIFO, except that a page is “skipped” if its use bit is set.
- The clock algorithm can be visualized as a victim pointer that cycles through the page frames. The pointer moves whenever a replacement is necessary:

```
while use bit of victim is set
   clear use bit of victim
   victim = (victim + 1) % num_frames
choose victim for replacement
victim = (victim + 1) % num_frames
```

Page Cleanliness: the “Modified” Bit

- A page is *modified* (sometimes called dirty) if it has been changed since it was loaded into memory.
- A modified page is more costly to replace than a clean page. (Why?)
- The MMU identifies modified pages by setting a *modified bit* in the PTE when the contents of the page change.
- Operating system clears the modified bit when it cleans the page
- The modified bit potentially has two roles:
  - Indicates which pages need to be cleaned.
  - Can be used to influence the replacement policy.

MIPS TLB entries do not include a modified bit.

What if the MMU Does Not Provide a “Modified” Bit?

- Can emulate it in similar fashion to the “use” bit
  1. When a page is loaded into memory, mark it as *read-only* (even if it is actually writeable) and set its simulated “modified” bit to false.
  2. If a program attempts to modify the page, a protection exception will occur.
  3. In its exception handler, the OS sets the page’s simulated “modified” bit to “true” and marks the page as writeable, if it is supposed to be a writeable page.
- This technique requires that the OS maintain two extra bits of information for each page:
  1. the simulated “modified” bit
  2. an “writeable” bit to indicate whether the page is supposed to be writeable

Enhanced Second Chance Replacement Algorithm

- Classify pages according to their use and modified bits:

  (0,0): not recently used, clean.

  (0,1): not recently used, modified.

  (1,0): recently used, clean

  (1,1): recently used, modified

- Algorithm:
  1. Sweep once looking for (0,0) page. Don't clear use bits while looking.
  2. If none found, look for (0,0) or (0,1) page, this time clearing "use" bits while looking.

Page Cleaning

- A modified page must be cleaned before it can be replaced, otherwise changes on that page will be lost.
- *Cleaning* a page means copying the page to secondary storage.
- Cleaning is distinct from replacement.
- Page cleaning may be *synchronous* or *asynchronous*:

  synchronous cleaning: happens at the time the page is replaced, during page fault handling. Page is first cleaned by copying it to secondary storage. Then a new page is brought in to replace it.

  asynchronous cleaning: happens before a page is replaced, so that page fault handling can be faster.

  – asynchronous cleaning may be implemented by dedicated OS *page cleaning threads* that sweep through the in-memory pages cleaning modified pages that they encounter.

Belady’s Anomaly

- FIFO replacement, 4 frames

Num	1	2	3	4	5	6	7	8	9	10	11	12
Refs	a	b	c	d	a	b	e	a	b	c	d	e
Frame 1	a	a	a	a	a	a	e	e	e	e	d	d
Frame 2		b	b	b	b	b	b	a	a	a	a	e
Frame 3			c	c	c	c	c	c	b	b	b	b
Frame 4				d	d	d	d	d	d	c	c	c
Fault?	x	x	x	x			x	x	x	x	x	x

- FIFO example on Slide 51 with same reference string had 3 frames and only 9 faults.

More memory does not necessarily mean fewer page faults.

Stack Policies

- Let $B(m,t)$ represent the set of pages in a memory of size m at time t under some given replacement policy, for some given reference string.
- A replacement policy is called a *stack policy* if, for all reference strings, all m and all t:

$$B(m,t) \subseteq B(m+1,t)$$

- If a replacement algorithm imposes a total order, independent of memory size, on the pages and it replaces the largest (or smallest) page according to that order, then it satisfies the definition of a stack policy.
- Examples: LRU is a stack algorithm. FIFO and CLOCK are not stack algorithms. (Why?)

Stack algorithms do not suffer from Belady's anomaly.

Prefetching

- Prefetching means moving virtual pages into memory before they are needed, i.e., before a page fault results.
- The goal of prefetching is *latency hiding*: do the work of bringing a page into memory in advance, not while a process is waiting.
- To prefetch, the operating system must guess which pages will be needed.
- Hazards of prefetching:
  - guessing wrong means the work that was done to prefetch the page was wasted
  - guessing wrong means that some other potentially useful page has been replaced by a page that is not used
- most common form of prefetching is simple sequential prefetching: if a process uses page x, prefetch page $x + 1$.
- sequential prefetching exploits spatial locality of reference

Page Size

- the virtual memory page size must be understood by both the kernel and the MMU
- some MMUs have support a configurable page size
- advantages of larger pages
  - smaller page tables
  - larger *TLB footprint*
  - more efficient I/O
- disadvantages of larger pages
  - greater internal fragmentation
  - increased chance of paging in unnecessary data

OS/161 on the MIPS uses a 4KB virtual memory page size.

How Much Physical Memory Does a Process Need?

- Principle of locality suggests that some portions of the process's virtual address space are more likely to be referenced than others.
- A refinement of this principle is the *working set model* of process reference behaviour.
- According to the working set model, at any given time some portion of a program's address space will be heavily used and the remainder will not be. The heavily used portion of the address space is called the *working set* of the process.
- The working set of a process may change over time.
- The *resident set* of a process is the set of pages that are located in memory.

According to the working set model, if a process's resident set includes its working set, it will rarely page fault.

Resident Set Sizes (Example)

```
PID   VSZ  RSS COMMAND
805 13940 5956 /usr/bin/gnome-session
831  2620  848 /usr/bin/ssh-agent
834  7936 5832 /usr/lib/gconf2/gconfd-2 11
838  6964 2292 gnome-smproxy
840 14720 5008 gnome-settings-daemon
848  8412 3888 sawfish
851 34980 7544 nautilus
853 19804 14208 gnome-panel
857  9656 2672 gpilotd
867  4608 1252 gnome-name-service
```

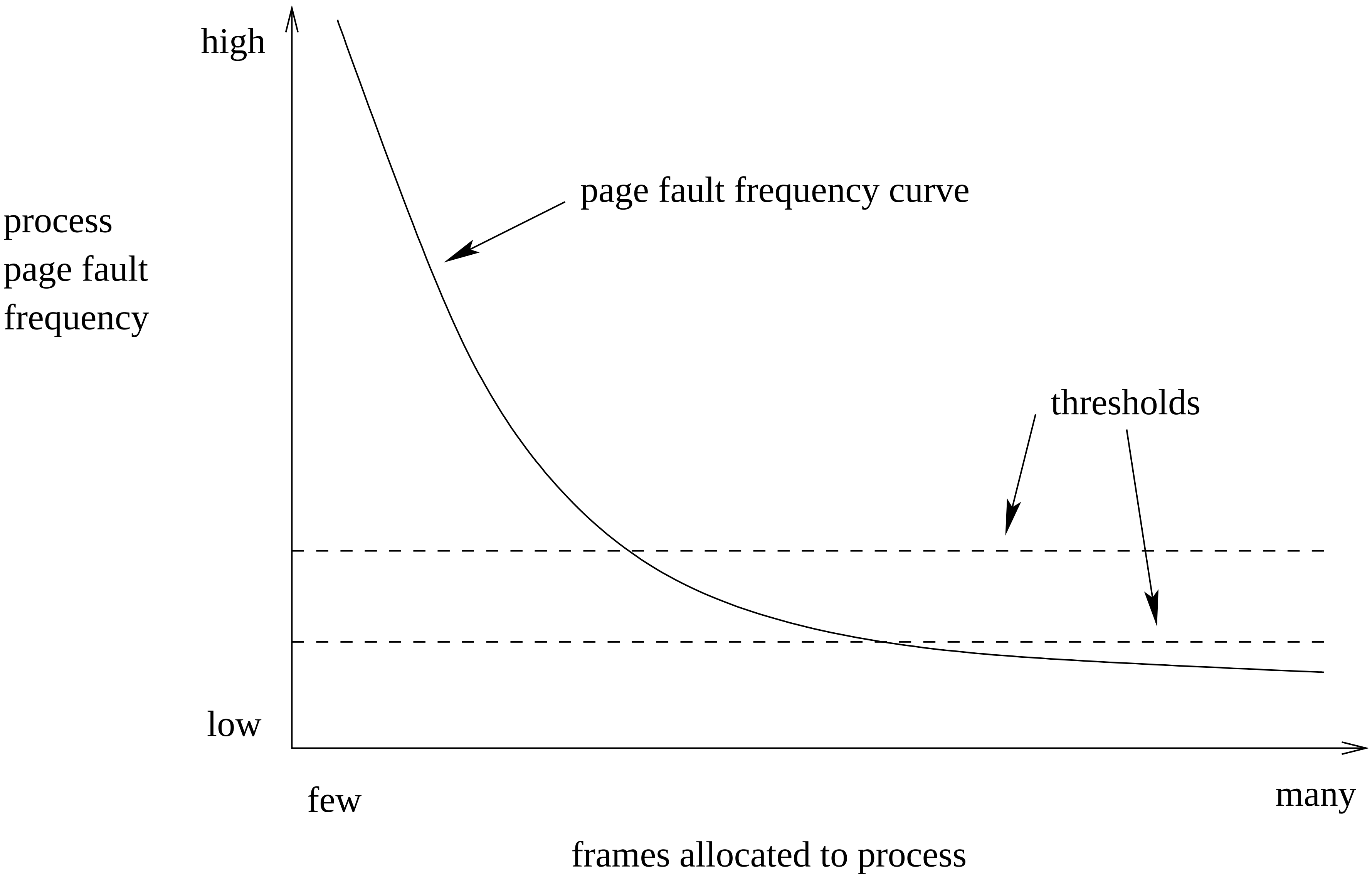
Refining the Working Set Model

- Define $WS(t, \Delta)$ to be the set of pages referenced by a given process during the time interval $(t - \Delta, t)$. $WS(t, \Delta)$ is the working set of the process at time t.
- Define $|WS(t, \Delta)|$ to be the size of $WS(t, \Delta)$, i.e., the number of *distinct* pages referenced by the process.
- If the operating system could track $WS(t, \Delta)$, it could:
  - use $|WS(t, \Delta)|$ to determine the number of frames to allocate to the process under a local page replacement policy
  - use $WS(t, \Delta)$ directly to implement a working-set based page replacement policy: any page that is no longer in the working set is a candidate for replacement

Page Fault Frequency

- A more direct way to allocate memory to processes is to measure their *page fault frequencies* - the number of page faults they generate per unit time.
- If a process's page fault frequency is too high, it needs more memory. If it is low, it may be able to surrender memory.
- The working set model suggests that a page fault frequency plot should have a sharp "knee".

A Page Fault Frequency Plot



Thrashing and Load Control

- What is a good multiprogramming level?
  - If too low: resources are idle
  - If too high: too few resources per process
- A system that is spending too much time paging is said to be *thrashing*. Thrashing occurs when there are too many processes competing for the available memory.
- Thrashing can be cured by load shedding, e.g.,
  - Killing processes (not nice)
  - Suspending and *swapping out* processes (nicer)

Swapping Out Processes

- Swapping a process out means removing all of its pages from memory, or marking them so that they will be removed by the normal page replacement process. Suspending a process ensures that it is not runnable while it is swapped out.
- Which process(es) to suspend?
  - low priority processes
  - blocked processes
  - large processes (lots of space freed) or small processes (easier to reload)
- There must also be a policy for making suspended processes ready when system load has decreased.