Operating Systems 412

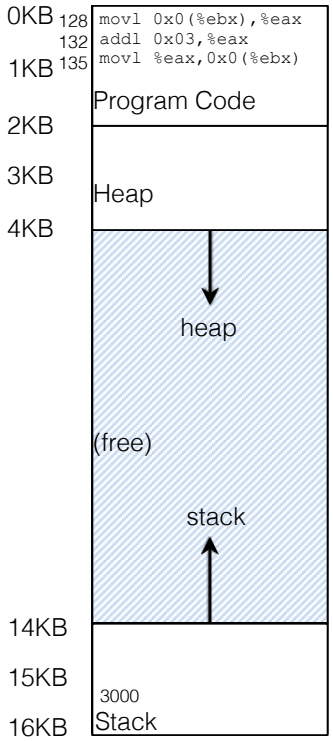
Pete Keleher

Memory

Memory

- 13 - Address Spaces
- 15 - Address Translation
- 16 - Segmentation
- 17 - Free Space Management
- 18 - Paging
- 19 - Translation Lookaside Buffers
- 20 - Advanced Paging
- 21 - Swapping
- 22 - Swapping Policy

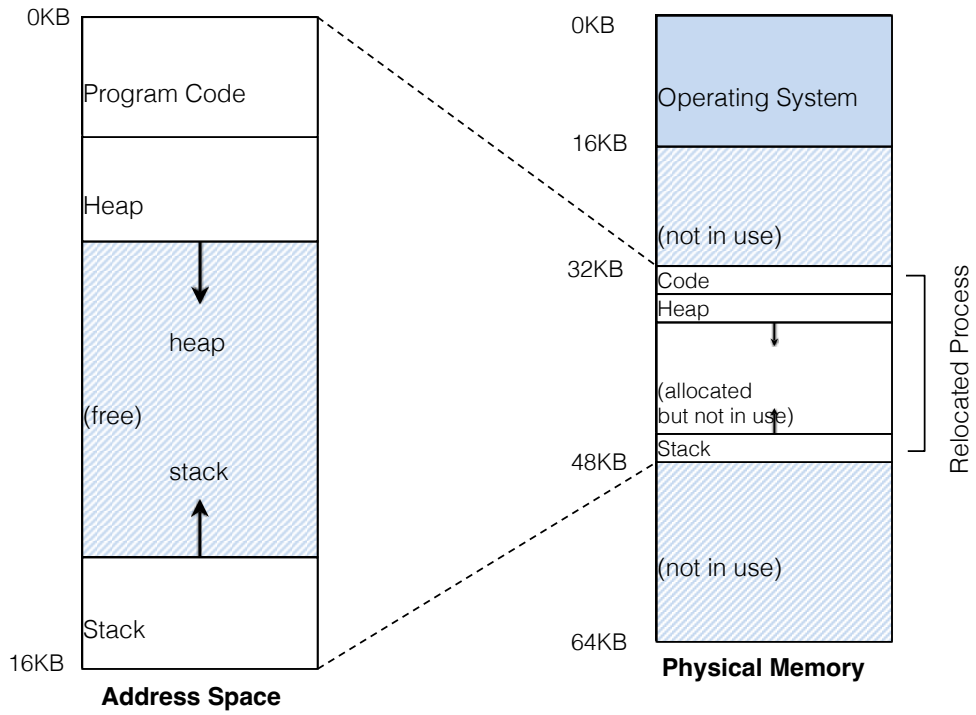
Example: Address Translation



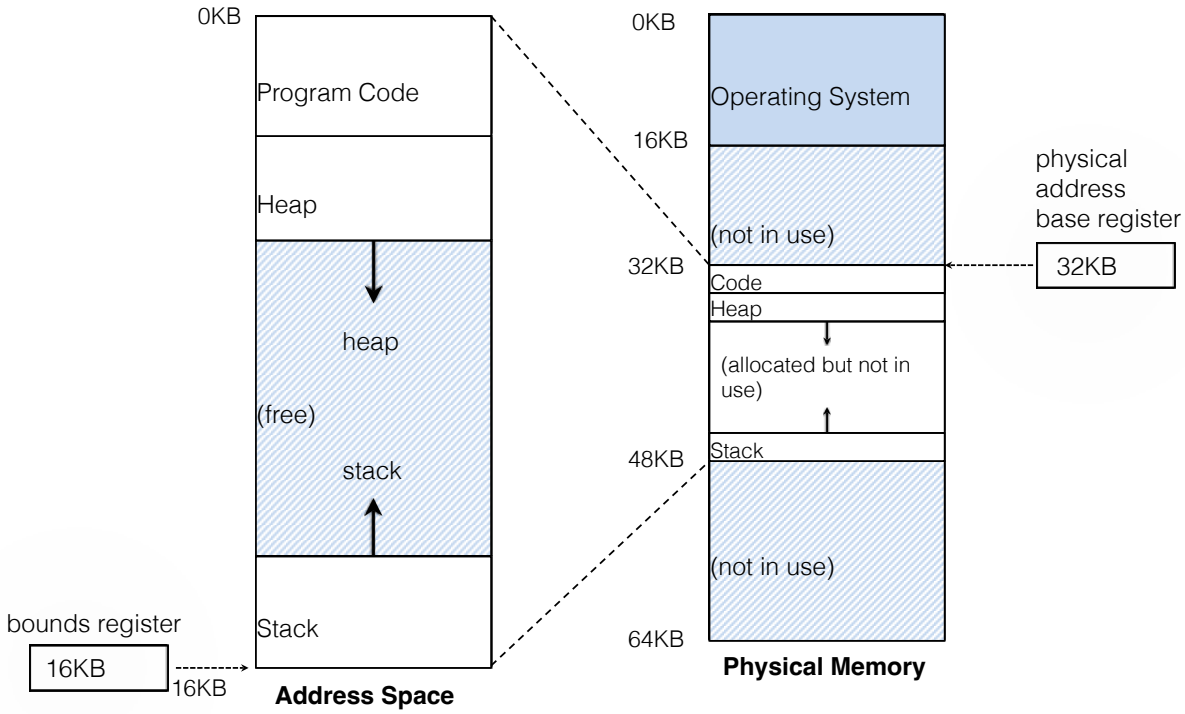
- Fetch instruction at address 128
- Execute instruction (load from address 15KB)
- Fetch instruction at address 132
- Execute instruction (no memory reference)
- Fetch the instruction at address 135
- Execute instruction (store to address 15 KB)

But not all programs can be at location 0

A Single Relocated Process



One Approach *base and bounds*



Dynamic(Hardware base) Relocation

- OS decides where in physical memory a process is loaded.
  - Set the **base** register:
    physical address = virtual address + base
  - Virtual addresses must **not be greater than bound** or **negative:**
    0 <= virtual address < bound

Address Translation *base and bounds*

```
128 : movl 0x0(%ebx), %eax
```

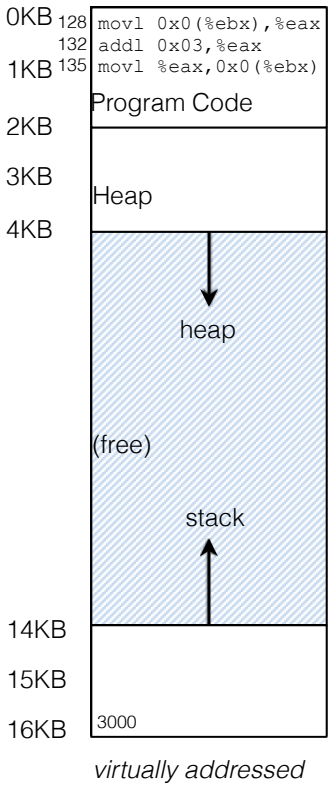
- **Fetch** instruction at address 128

$$32896 = 128 + 32KB(base)$$

- **Execute** this instruction
  - Load from address 15KB

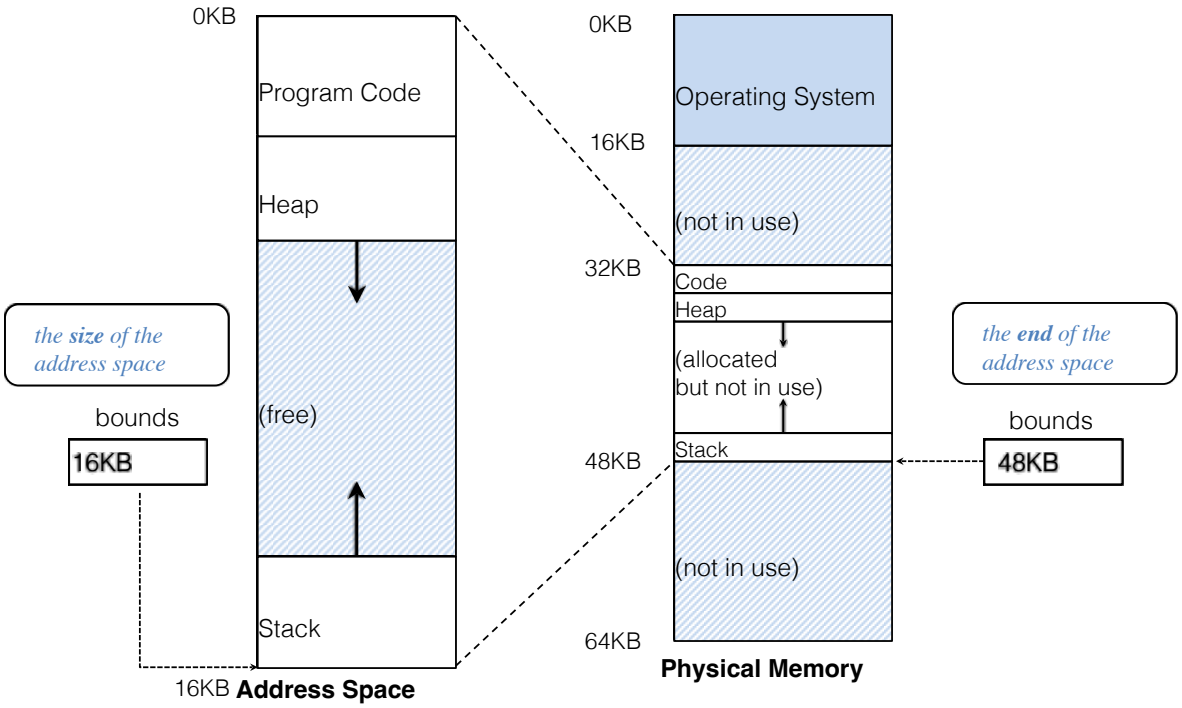
$$47KB = 15KB + 32KB(base)$$

physical address = offset + base



virtually addressed

Two ways to Use the Bounds Register



Hardware Requirements *for base and bounds*

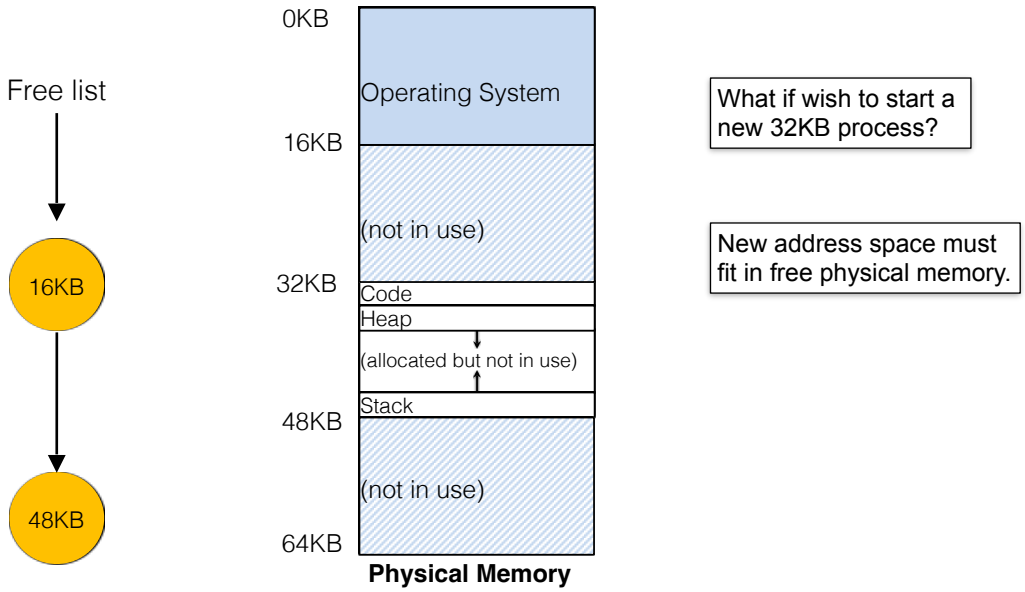
- Privileged mode
  - user processes should not execute privileged operations
- Base and bounds registers
    - per CPU / core
- Translate virtual addresses, and check bounds
  - instructions
  - data
- Privileged instructions to update base/bound registers
- Ability to raise exceptions
  - out-of-bound accesses

OS Requirements *for base and bounds*

- OS must intervene at three critical junctures:
  - When a process starts running:
    - find space for address space in physical memory
  - When a process is terminated:
    - reclaims the memory for use
  - When context switch occurs:
    - Save and store the base-and-bounds pair

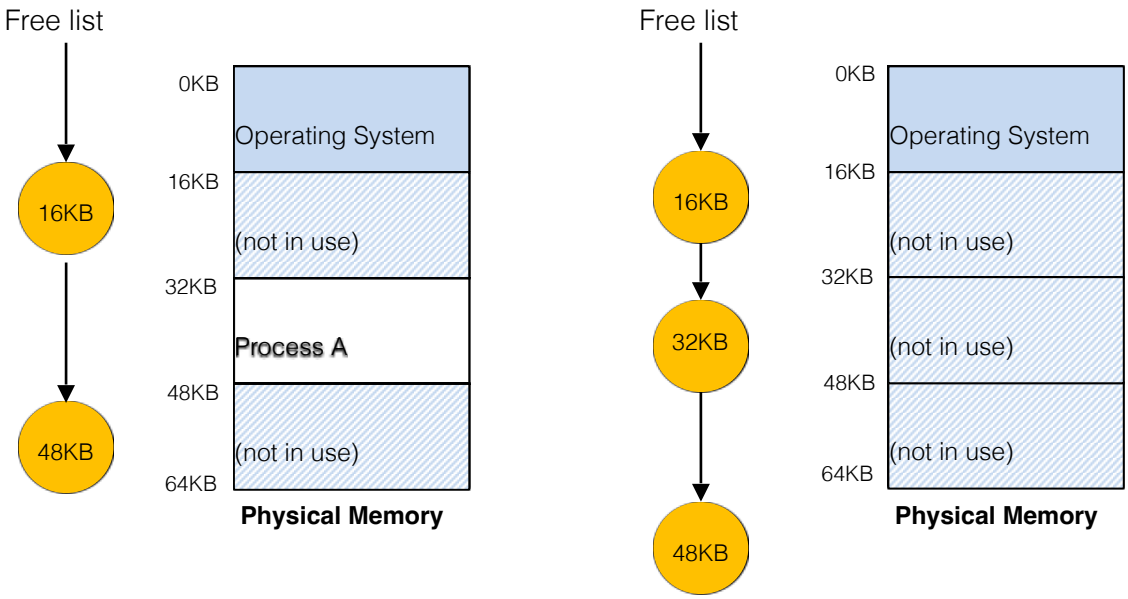
OS Issues: *single chunk of memory*

- The OS must find a room for a new address space.
  - free list : a list of unused ranges of physical memory



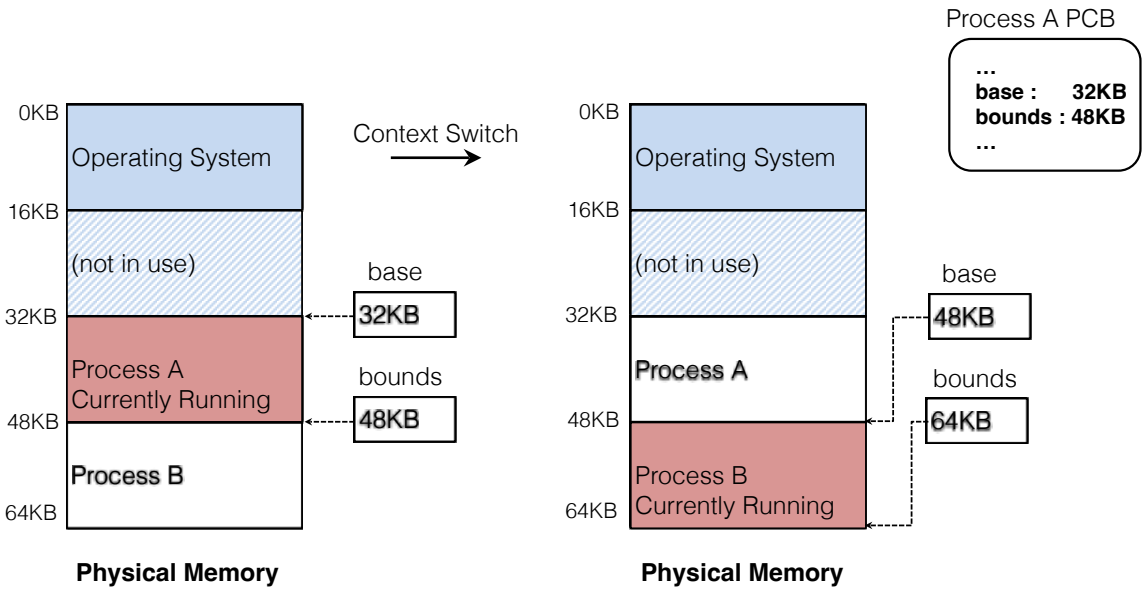
OS Issues: Process Termination

- OS must put memory back on the free list.



OS Issues: Context Switches

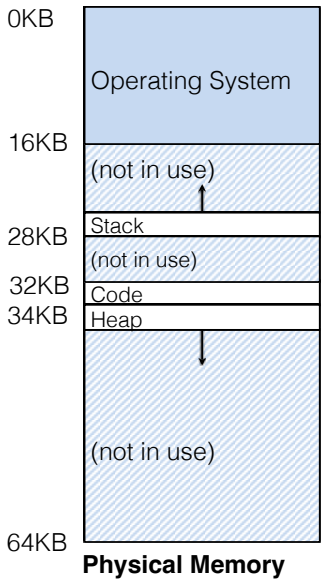
- OS must save and restore base-and-bounds pair.
  - In process structure or process control block (PCB)



Segmentation *another approach*

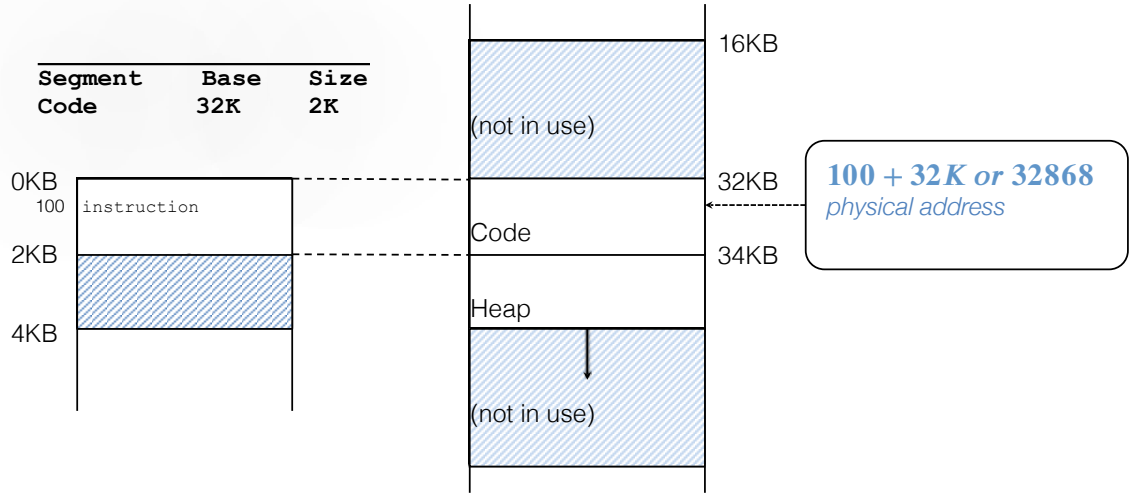
- Segment is a *contiguous* portion of the address space:
  - code, stack, heap, …
- Can be placed anywhere in *contiguous* physical memory.
  - Basically base and bounds *per segment*

Segment	Base	Bound
Code	32K	2K
Heap	34K	2K
Stack	28K	2K



Address Translation for Segments

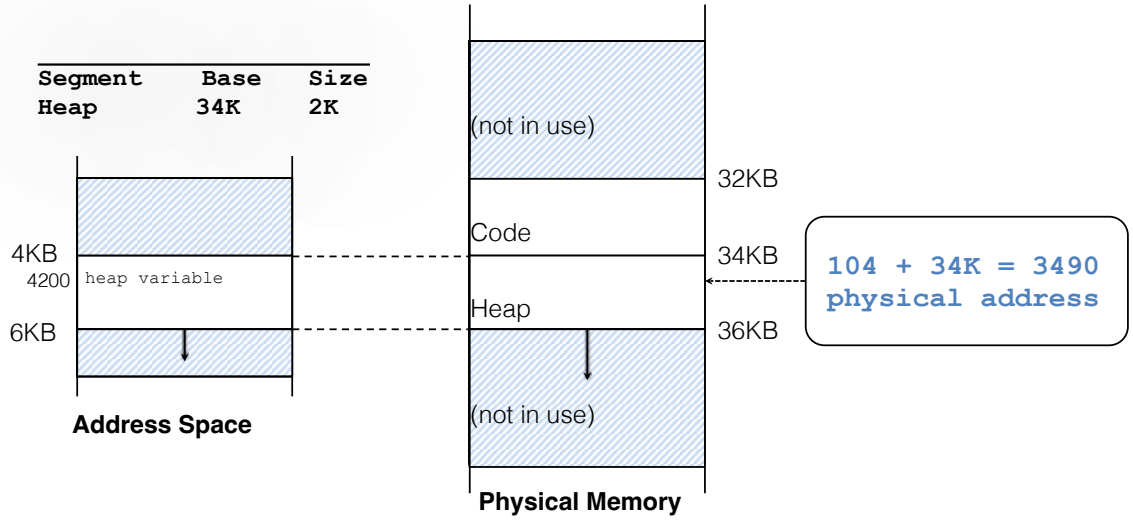
- The `offset` of virtual address 100 is: `100`
  - The code segment **starts at virtual address 0** in address space.



physical address = offset + base

Address Translation for Segments

- The `offset` of virtual address 4200 is `104`.
  - The heap segment **starts at virtual address 4096** in address space.

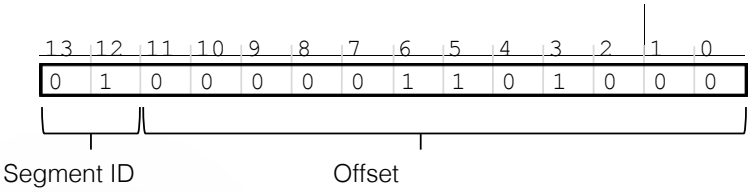


physical address = offset + base

Segment Descriptors

- Explicit approach
  - Chop up the address space into segments based on the **top few bits** of virtual address.

- Example: virtual address 4200 (01000001101000)



Segment	bits
Code	00
Heap	01
Stack	10
-	11

Segment Descriptors

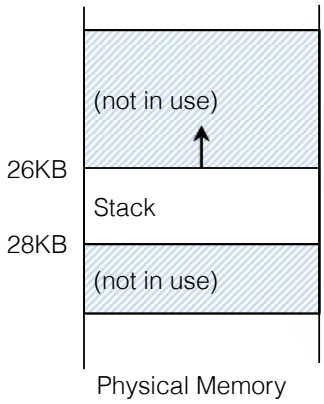
- Bits
  - `SEG_MASK = 0x3000(11000000000000)`
  - `SEG_SHIFT = 12`
  - `OFFSET_MASK = 0xFFF (00111111111111)`

```
1   // get top 2 bits of 14-bit VA
2   Segment = (VirtualAddress & SEG_MASK) >> SEG_SHIFT

3   // now get offset
4   Offset = VirtualAddress & OFFSET_MASK
5   if (Offset >= Bounds[Segment])
6        RaiseException(PROTECTION_FAULT)
7   else
8        PhysAddr = Base[Segment] + Offset
9        Register = AccessMemory(PhysAddr)
```

Referring to Stack Segment

- Stack grows *backward*.
- *Extra hardware support* needed.
  - The hardware checks which way the segment grows.
  - 1: positive direction, 0: negative direction



Segment Register(with Negative-Growth Support)

Segment	Base	Size	Grows Positive?
Code	32K	2K	1
Heap	34K	2K	1
Stack	28K	2K	0

physical address = base + offset - sizeof(stack)

two's complement arithmetic

Support for Sharing

- Segments can be shared between address spaces
  - Code sharing still used
- Need hardware support in form of *protection* bits.
  - Bits indicate read, write and execute permissions.

Segment Register Values(with Protection)

Segment	Base	Size	Grows Positive?	Protection
Code	32K	2K	1	Read-Execute
Heap	34K	2K	1	Read-Write
Stack	28K	2K	0	Read-Write

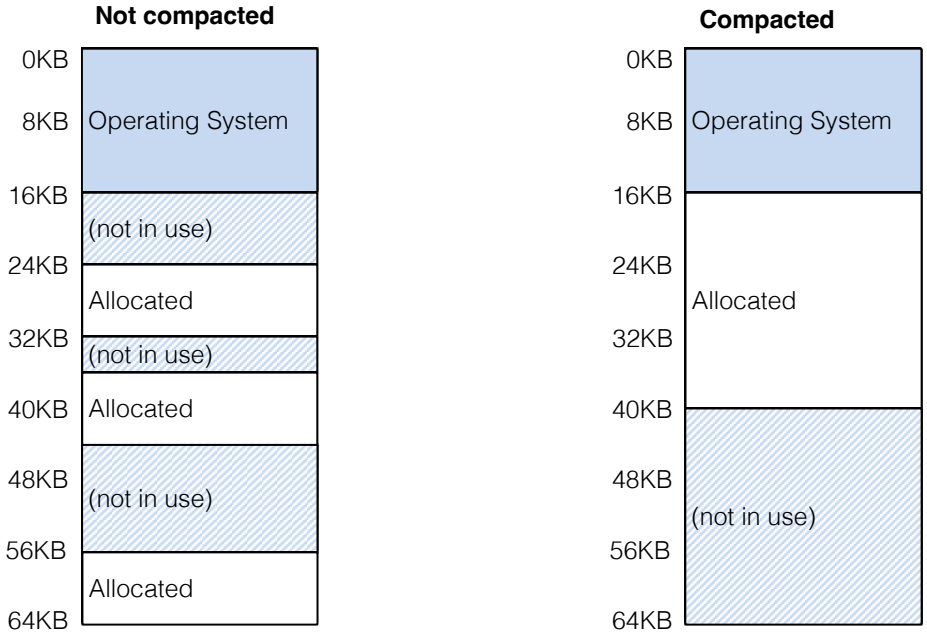
Fine-Grained and Coarse-Grained

- Coarse-Grained is small number of segments
  - e.g., code, heap, stack.
- Fine-Grained segmentation allows more flexibility
  - Hardware-supported segment tables

OS support: Fragmentation

- External Fragmentation:
  - Distinct runs of free space in physical memory
  - Might be **24KB free**, but **not in one contiguous** segment.
  - The OS **cannot** immediately satisfy the **20KB request**.
- Compaction: consolidating segments in physical memory.
  - Compaction is **costly**.
    - **Stop** running process.
    - **Copy** data to somewhere.
    - **Change** segment register value.

Memory Compaction



GeekOS

- segmented memory addresses
  - 16-bit “segment selector”, 32-bit offset
  - segment selector has:
    - 1 bit: GDT or LDT
    - 13 bits: index into GDT or LDT
    - 2 bits: protection level of segment
  - segment descriptor (from table) has:
    - linear *base* physical address of segment: 32 bits
    - limit (size) of segment: 20 bits
    - descriptor privilege level (dpl): 2 bits
    - type of segment (data, code, system, tss, gate): 4 bits
    - present (in-memory): 1 bit
    - etc.

GeekOS

- GDT
  - entries point to kernel segments, optionally user segments
  - entry 0 (null selector) is not used to access memory
  - `gdtr` register points to the GDT

- LDT similar, but
  - points to segments of a single process
  - entry 0 can be used
  - any number of LDTs can be in memory
  - `ldtr` register points (via GDT) to currently used LDT

Virtual Memory

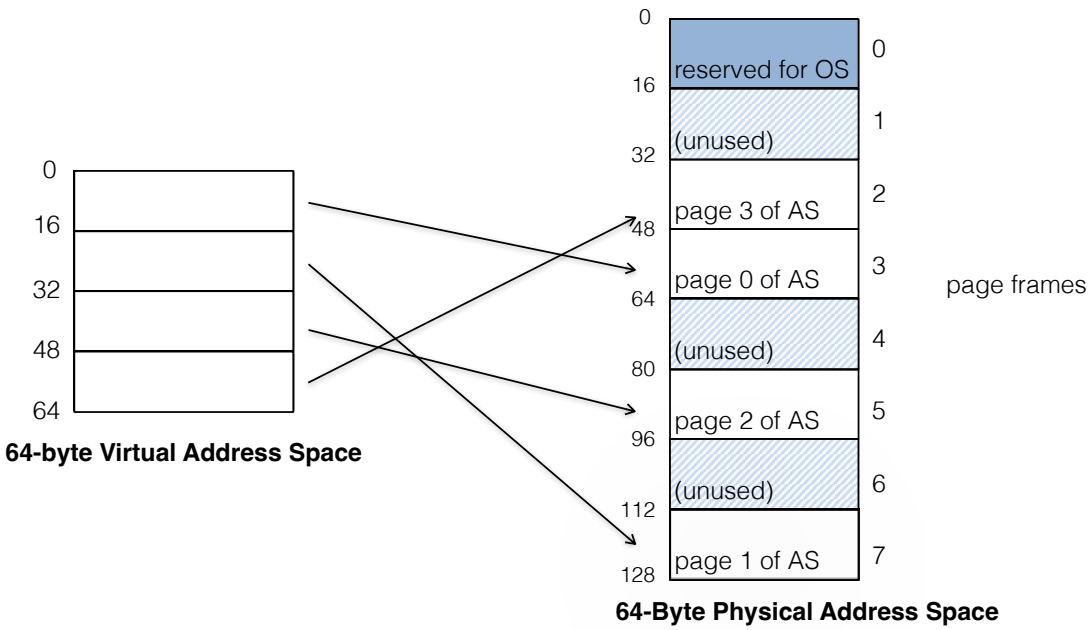
- 14 - Memory API
- 13 - Address Spaces
- 15 - Address Translation
- 16 - Segmentation
- 17 - Free Space Management
- 18 - Paging
- 19 - Translation Lookaside Buffers
- 20 - Advanced Paging
- 21 - Swapping
- 22 - Swapping Policy

Paging

- Paging splits address space into fixed-size *pages*.
  - vs segmentation: variable size of logical segments
- Physical memory holding a page is the *page frame*
- Per-process page tables
  - translate virtual address to physical address.
- Flexibility:
  - No assumptions on how heap and stack grow or are used
- Simplicity: ease of free-space management
  - All pages and page frames are the same size
  - Free lists are easy…

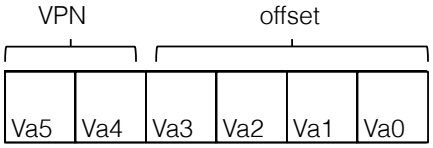
Paging Example

- 128-byte physical memory with eight 16-byte page frames
- 64-byte address space with 16-byte pages

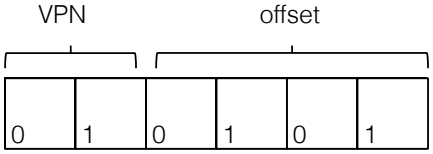


Address Translation

- Two components in the virtual address
  - VPN: virtual page number
  - Offset: offset within the page

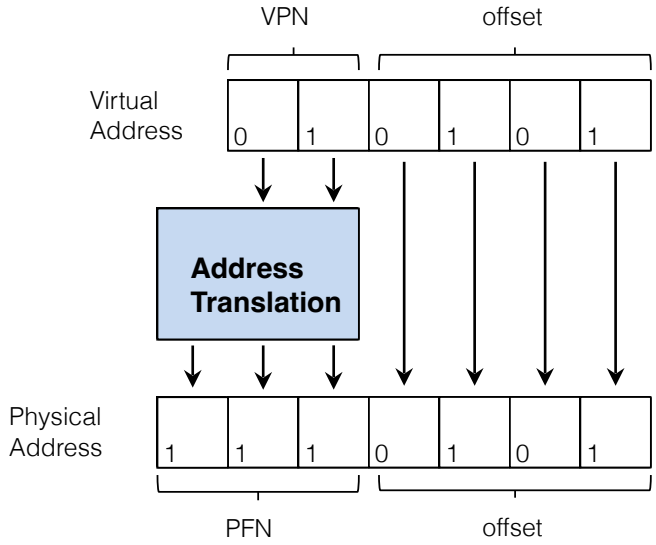


- Example: virtual address 21 in 64-byte address space



Example: Address Translation

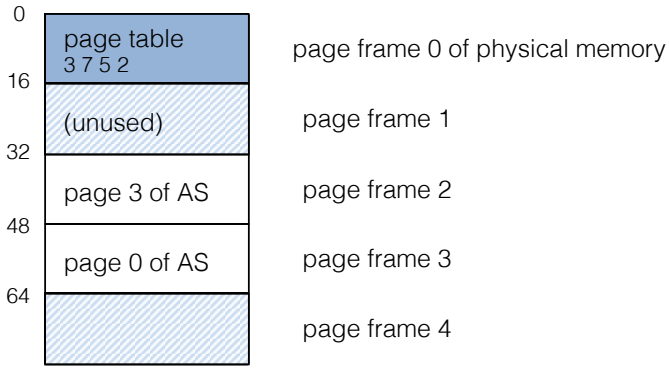
- The virtual address 21 in 64-byte address space



Where Are Page Tables Stored?

- Page tables can be large…
  - 32-bit address space with 4-KB pages, 20 bits for VPN
    - assume entry is 4 bytes:
    - page table size is $2^{20} * 4 = 2^{22} =$ 4MB of space

- Page tables for each process are stored in memory…



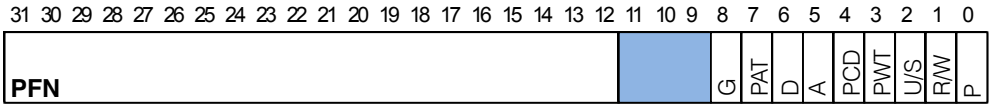
What Is In The Page Table?

- A page table is just a **data structure** that is used to map the virtual address to physical address.
  - Simplest form: a linear page table, an array

- The OS/hardware accesses a page-table entry by indexing into the array by virtual page-number

- Common bits:
  - Valid Bit: whether the particular translation is valid.
  - Protection Bit: read, write, execute
  - Present Bit: in physical memory or swapped out
  - Dirty Bit: page modified since it brought into memory
  - Reference Bit (Accessed Bit): page has been accessed

Example: x86 Page Table Entry



- P: present
- R/W: read/write bit
- U/S: supervisor
- A: accessed bit
- D: dirty bit
- PFN: the page frame number
- Others: mostly caching directives

Paging: Too Slow

- To find a location of the desired PTE, the starting location of the page table is needed.

- For every memory reference, paging requires the OS to perform one extra memory reference.

Accessing Memory With Paging

```
1       // Extract the VPN from the virtual address
2       VPN = (VirtualAddress & VPN_MASK) >> SHIFT
3
4       // Form the address of the page-table entry (PTE)
5       PTEAddr = PTBR + (VPN * sizeof(PTE))
6
7       // Fetch the PTE
8       PTE = AccessMemory(PTEAddr)
9
10      // Check if process can access the page
11      if (PTE.Valid == False)
12              RaiseException(SEGMENTATION_FAULT)
13      else if (CanAccess(PTE.ProtectBits) == False)
14              RaiseException(PROTECTION_FAULT)
15      else
16              // Access is OK: form physical address and fetch
17              offset = VirtualAddress & OFFSET_MASK
18              PhysAddr = (PTE.PFN << PFN_SHIFT) | offset
19              Register = AccessMemory(PhysAddr)
```


A Memory Trace

- Example: A Simple Memory Access

```
int array[1000];
...
for (i = 0; i < 1000; i++)
        array[i] = 0;
```

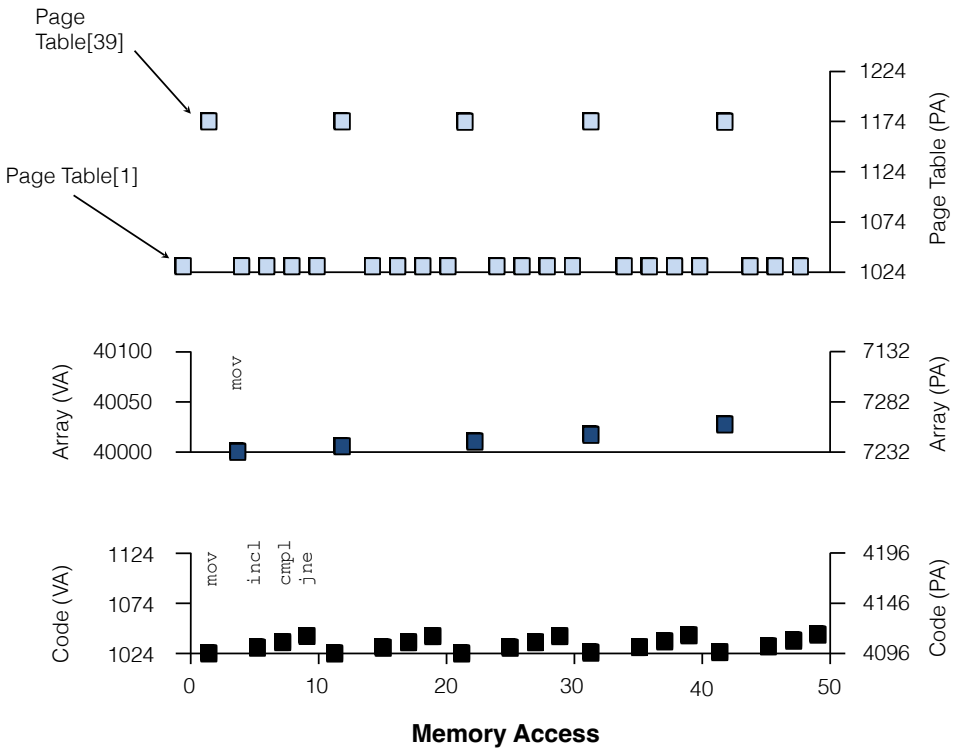
- Compile and execute

```
prompt> gcc -o array array.c -Wall -o
prompt>./array
```

- Resulting Assembly code

```
0x1024 movl $0x0,(%edi,%eax,4)
0x1028 incl %eax
0x102c cmpl $0x03e8,%eax
0x1030 jne 0x1024
```

A Virtual(And Physical) Memory Trace

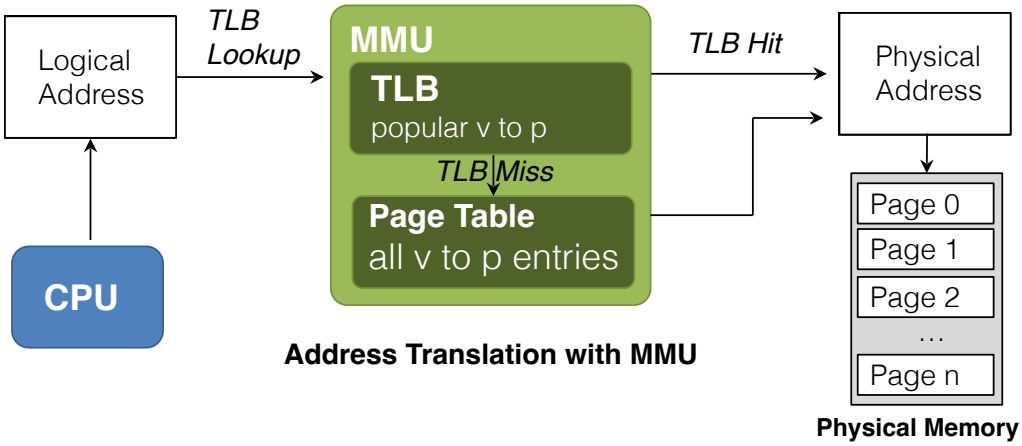


Virtual Memory

- 13 - Address Spaces
- 14 - Memory API
- 15 - Address Translation
- 16 - Segmentation
- 17 - Free Space Management
- 18 - Paging
- 19 - Translation Lookaside Buffers
- 20 - Advanced Paging
- 21 - Swapping
- 22 - Swapping Policy

TLB

- Part of the chip's memory-management unit (MMU).
- A hardware cache of **popular** virtual-to-physical address translation.



Address Translation with MMU

Basic TLB Algorithm

- extract the virtual page number (VPN).
- check for hit in the the TLB
- extract page frame number from relevant TLB entry, form desired physical address, and access memory

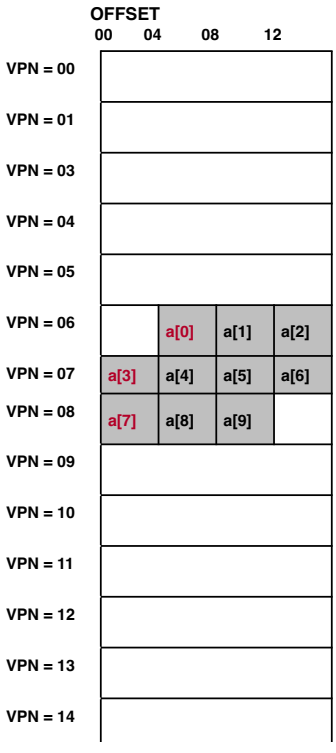
Basic TLB Algorithm

```
1  VPN = (VirtualAddress & VPN_MASK) >> SHIFT
2  (Success, TlbEntry) = TLB_Lookup(VPN)
3  if (Success == True)    // TLB Hit
4      if (CanAccess(TlbEntry.ProtectBits) == True)
5          Offset   = VirtualAddress & OFFSET_MASK
6          PhysAddr = (TlbEntry.PFN << SHIFT) | Offset
7          Register = AccessMemory(PhysAddr)
8      else
9          RaiseException(PROTECTION_FAULT)
10 else                    // TLB Miss
11     PTEAddr = PTBR + (VPN * sizeof(PTE))
12     PTE = AccessMemory(PTEAddr)
13     if (PTE.Valid == False)
14         RaiseException(SEGMENTATION_FAULT)
15     else if (CanAccess(PTE.ProtectBits) == False)
16         RaiseException(PROTECTION_FAULT)
17     else
18         TLB_Insert(VPN, PTE.PFN, PTE.ProtectBits)
19         RetryInstruction()
```

Figure 19.1: **TLB Control Flow Algorithm**

Example: Accessing An Array

- How a TLB can improve its performance.



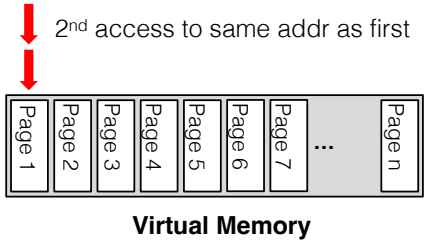
The TLB improves performance due to spatial locality

```
0:      int sum = 0 ;
1:      for( i=0; i<10; i++){
2:              sum+=a[i];
3:      }
```

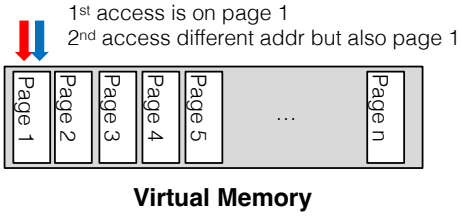
3 TLB misses and 7 hits.
Thus TLB hit rate is 70%.

Locality

- Temporal Locality
  - An instruction or data item that has been recently accessed will likely be re-accessed soon in the future.



- Spatial Locality
  - If a program accesses memory at address `x`, it will likely soon access memory near `x`.



Virtual Memory

- 13 - Address Spaces
- 14 - Memory API
- 15 - Address Translation
- 16 - Segmentation
- 17 - Free Space Management
- 18 - Paging
- 19 - Translation Lookaside Buffers
- 20 - Advanced Paging
- 21 - Swapping
- 22 - Swapping Policy

Who Handles The TLB Miss?

- Hardware handles the TLB miss entirely on CISC processors.
  - The hardware know where the page tables are located
  - … "walks" the page table, finding the correct entry and extracting the desired translation, update and retry instruction.
  - this is a hardware-managed TLB.

- RISC processors often manage TLBs in software.
  - On a TLB miss, the hardware raises an exception
    - Trap handler is code within the OS that is written with the express purpose of handling TLB misses.

TLB Control Flow algorithm (OS Handled)

- The hardware would do the following:

```
1:        VPN = (VirtualAddress & VPN_MASK) >> SHIFT
2:        (Success, TlbEntry) = TLB_Lookup(VPN)
3:        if (Success == True) // TLB Hit
4:                if (CanAccess(TlbEntry.ProtectBits) == True)
5:                        Offset = VirtualAddress & OFFSET_MASK
6:                        PhysAddr = (TlbEntry.PFN << SHIFT) | Offset
7:                        Register = AccessMemory(PhysAddr)
8:                else
9:                        RaiseException(PROTECTION_FAULT)
10:       else // TLB Miss
11:                RaiseException(TLB_MISS)
```

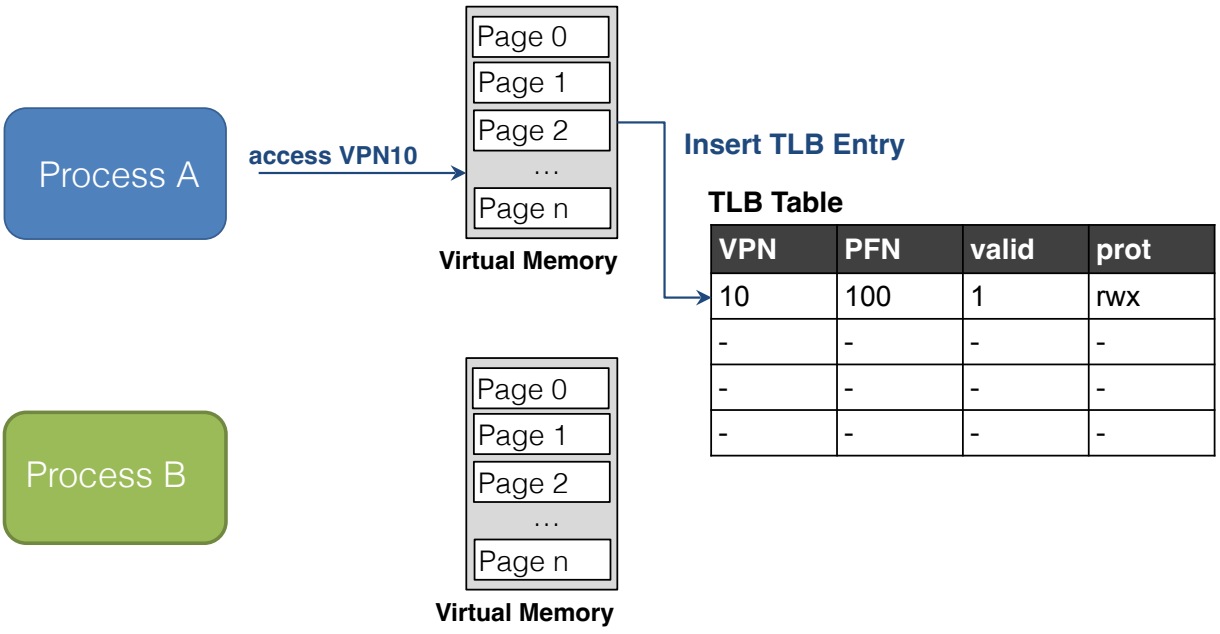
- But might be slow, why not just use the hardware approach?

TLB entry

- TLB entries are often *fully associative* (any entry for any mapping)
  - A typical TLB might have 32, 64, or 128 entries.
  - Hardware searches the TLB in parallel to find the translation.
  - other bits: valid, protection, address-space identifier, dirty bit

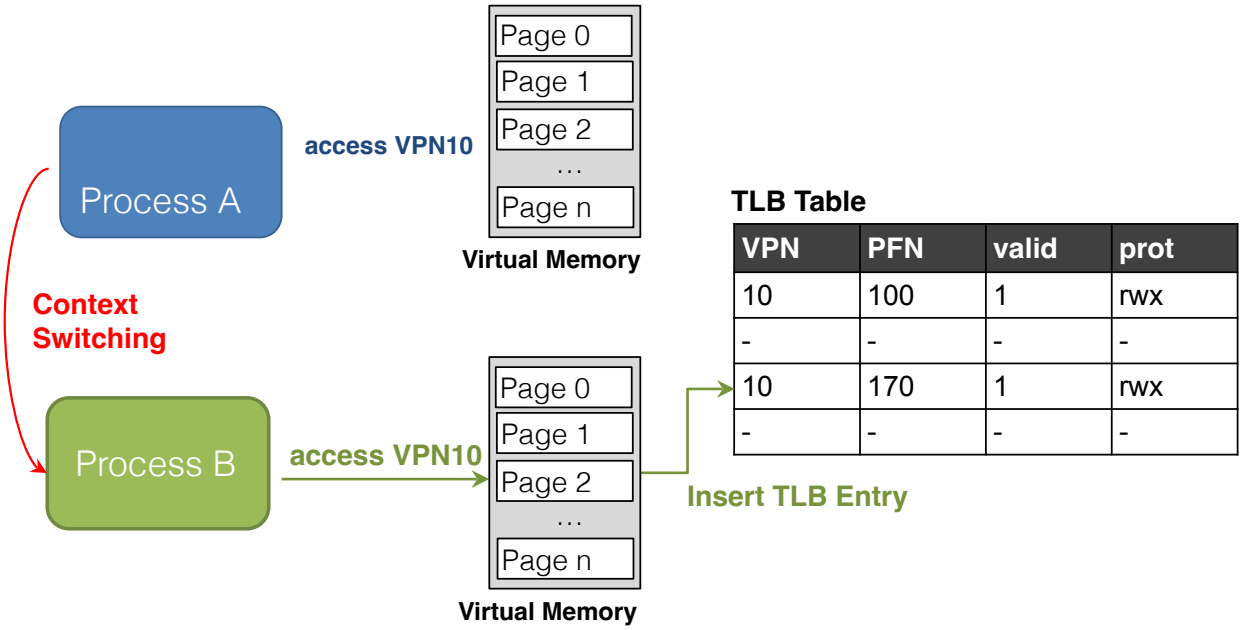
Typical TLB entry

TLB Issue: Context Switching



VPN	PFN	valid	prot
10	100	1	rwx
-	-	-	-
-	-	-	-
-	-	-	-

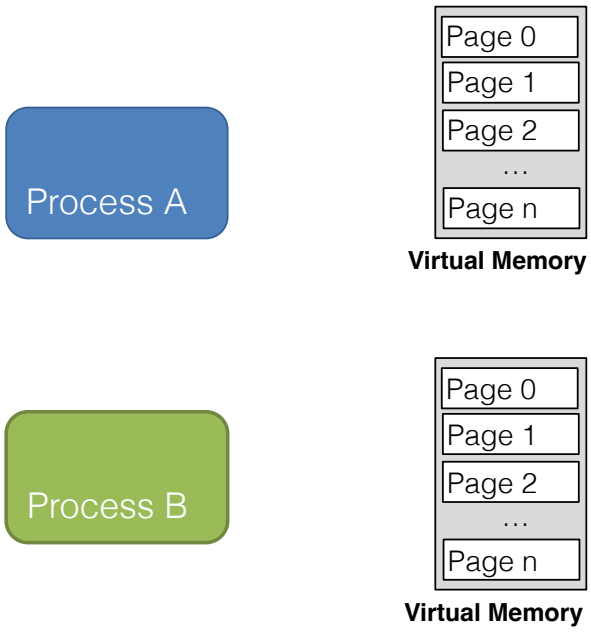
TLB Issue: Context Switching



TLB Table

VPN	PFN	valid	prot
10	100	1	rwx
-	-	-	-
10	170	1	rwx
-	-	-	-

TLB Issue: Context Switching



TLB Table

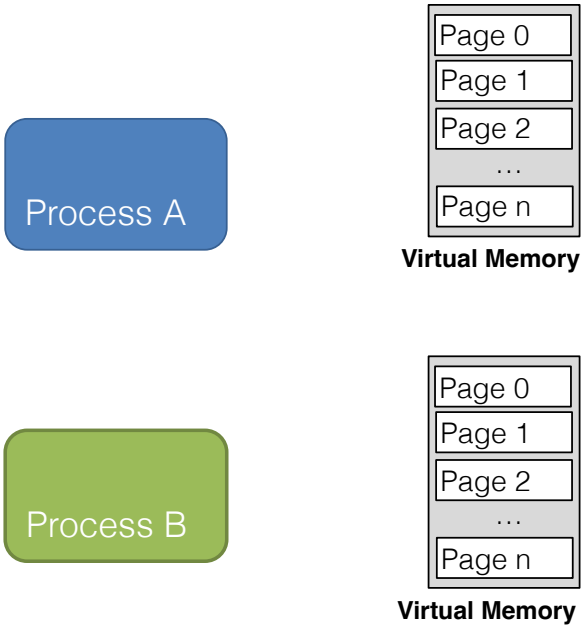
VPN	PFN	valid	prot
10	100	1	rwx
-	-	-	-
10	170	1	rwx
-	-	-	-

Can't Distinguish which entry is meant for which process

Could just flush the TLB on every context switch…

Disambiguating Address Spaces

- Provide an address space identifier(ASID) field in the TLB.



TLB Table

VPN	PFN	valid	prot	ASID
10	100	1	rwx	1
-	-	-	-	-
10	170	1	rwx	2
-	-	-	-	-

Another Case

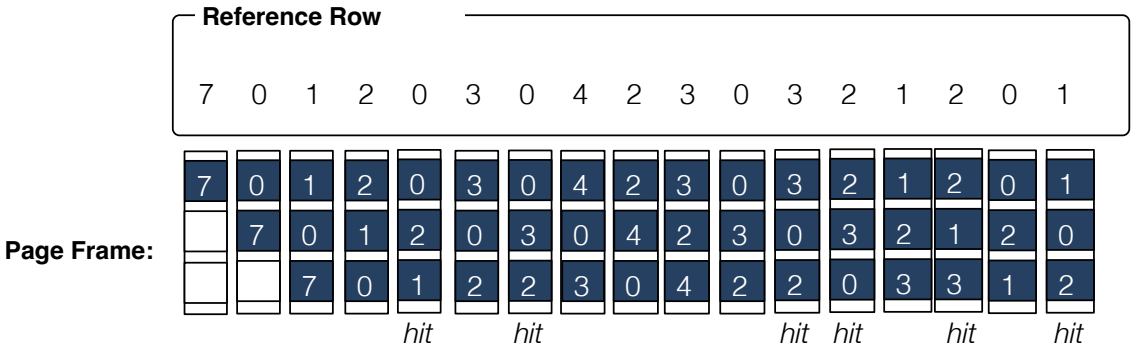
- Two processes share a page.
  - Process 1 is sharing physical page 101 with Process2.
  - P1 maps this page into the 10th page of its address space.
  - P2 maps this page to the 50th page of its address space.

VPN	PFN	valid	prot	ASID
10	101	1	rwx	1
-	-	-	-	-
50	101	1	rwx	2
-	-	-	-	-

Sharing of pages is useful as it reduces the number of physical pages in use.

TLB Replacement Policy

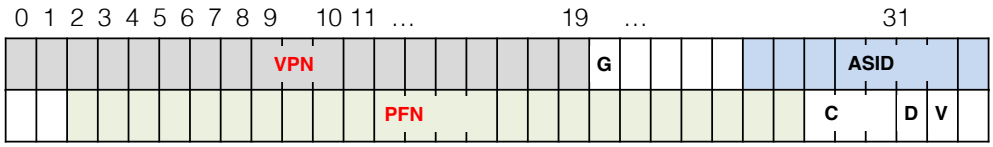
- LRU (Least Recently Used)
  - Evict an entry that has not recently been used.
  - Take advantage of ***locality*** in the memory-reference stream.



- 6 hits, 11 misses

A Real TLB Entry

64-bit MIPS R4000 TLB entry



Flag	Content
19-bit VPN	The rest reserved for the kernel.
24-bit PFN	Systems can support with up to 64GB of main memory(pages).
Global bit(G)	Used for pages that are globally-shared among processes.
ASID	OS can use to distinguish between address spaces.
Coherence bit(C)	determine how a page is cached by the hardware.
Dirty bit(D)	marking when the page has been written.
Valid bit(V)	tells the hardware if there is a valid translation present in the entry.

Virtual Memory

- 13 - Address Spaces
- 14 - Memory API
- 15 - Address Translation
- 16 - Segmentation
- 17 - Free Space Management
- 18 - Paging
- 19 - Translation Lookaside Buffers
- 20 - Advanced Paging
- 21 - Swapping
- 22 - Swapping Policy