Memory

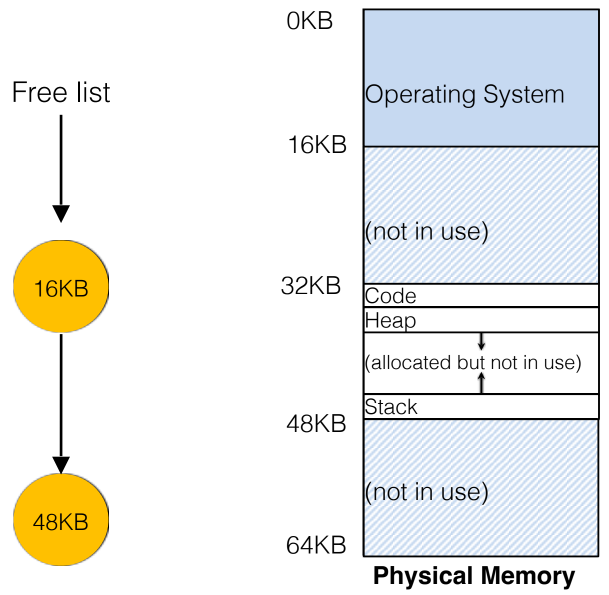
- 14 - Memory API
- 17 - Free Space Management
- 13 - Address Spaces
- 15 - Address Translation
- 16 - Segmentation
- 18 - Paging
- 19 - Translation Lookaside Buffers
- 20 - Advanced Paging
- 21 - Swapping
- 22 - Swapping Policy

Issues (still Base and Bounds)

- OS intervenes 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

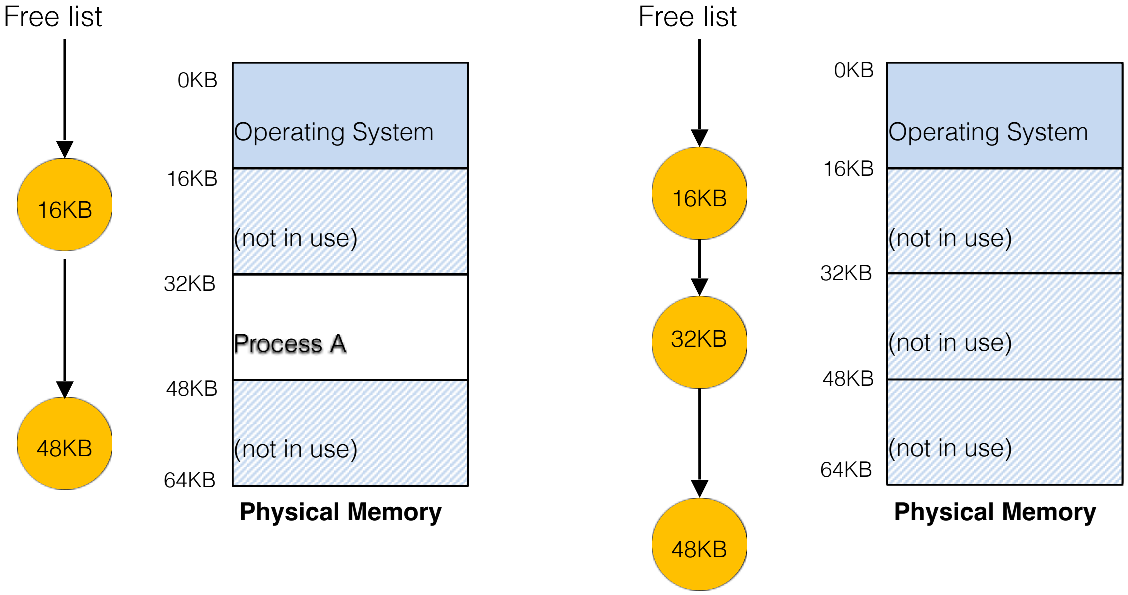
OS Issues: When a Process Starts Running

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



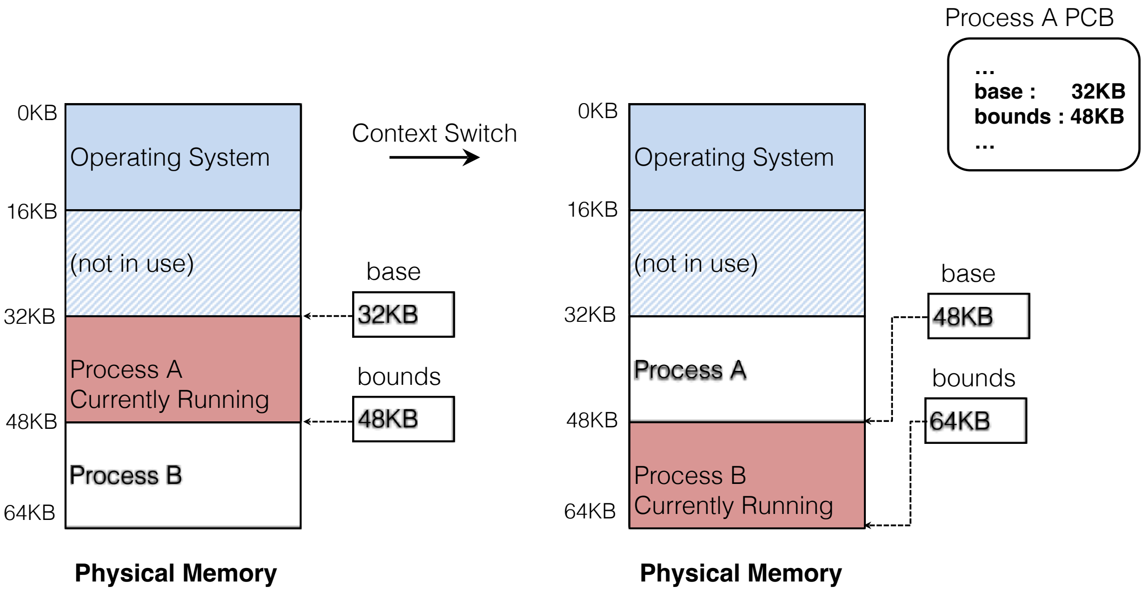
OS Issues: Process Termination

- OS must put memory back on the free list.



OS Issues: Context Switches

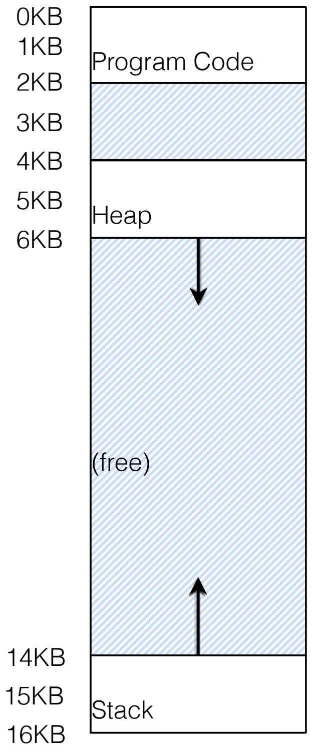
- OS must save and restore base-and-bounds pair.
  - In process structure or process control block(PCB)



Process A PCB

...
base : 32KB
bounds : 48KB
...

Not Always Efficient



- Need big chunk of “free” space
  - physically consecutive memory

- Cannot run when address space **does not fit** into physical memory

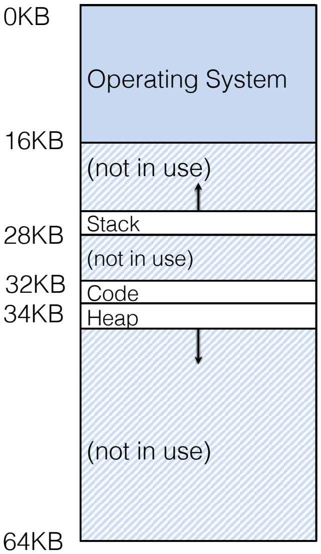
Memory

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

Segmentation

- Segment is a *contiguous* portion of the address space:
  - Several types: code, stack, heap, …
- Segments can be placed anywhere in physical memory.
  - Slightly modified base and bounds per segment

Segment	**Base**	**Size**
Code	32K	2K
Heap	34K	2K
Stack	28K	2K



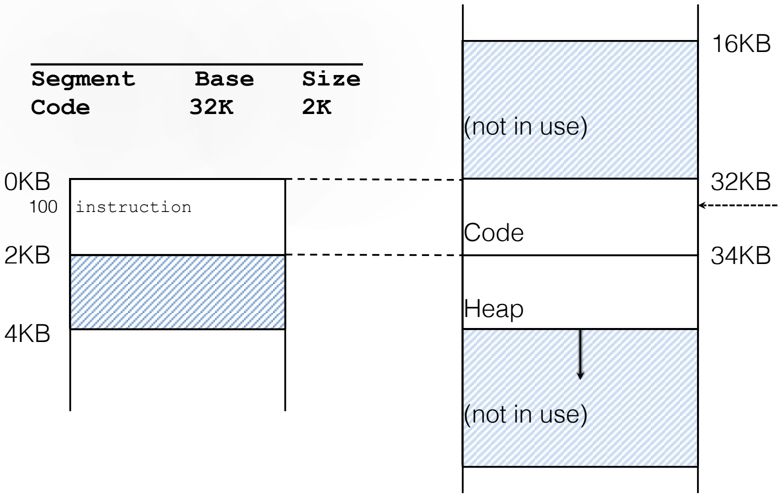
Physical Memory

Address Translation for Segments

physical address = ***offset*** + *base*

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

Segment	Base	Size
Code	32K	2K



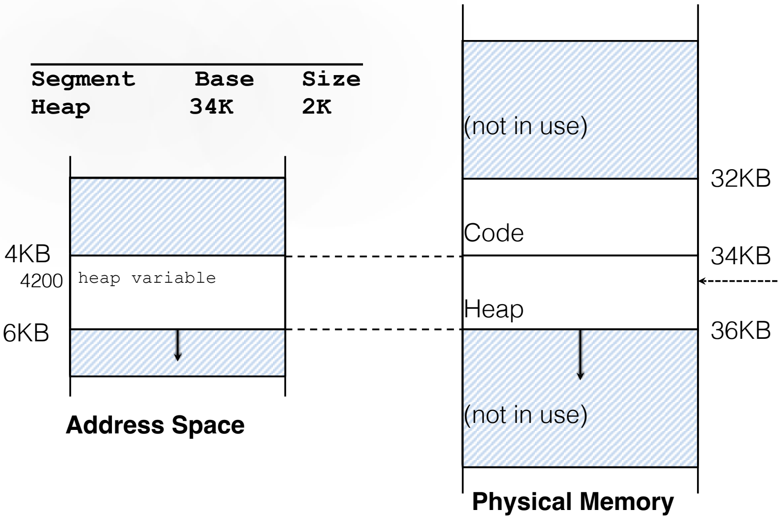
100 + 32*K* or 32868
physical address

Address Translation for Segments

Offset + *base*, not virtual address + base

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

Segment	Base	Size
Heap	34K	2K

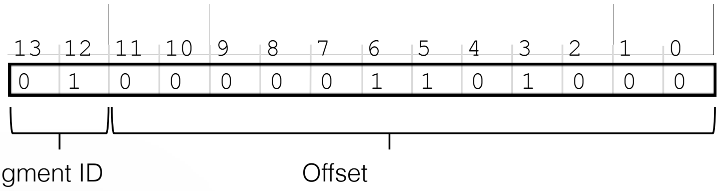


104 + 34K = 3490
physical address

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)



13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	0	0	0	1	1	0	1	0	0	0

Segment	**bits**
Code	**00**
Heap	**01**
Stack	**10**
-	**11**

Segment Descriptors

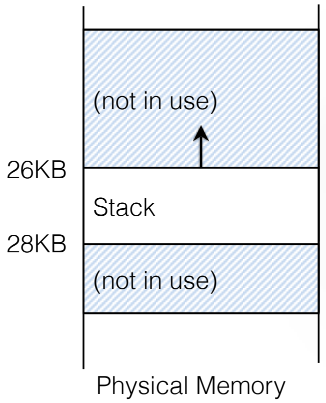
- Bits
  - `SEG_MASK = 0x3000(11000000000000)`
  - `SEG_SHIFT = 12`
  - `OFFSET_MASK = 0xFFF (00111111111111)`

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

// now get offset
Offset = VirtualAddress & OFFSET_MASK
if (Offset >= Bounds[Segment])
     RaiseException(PROTECTION_FAULT)
else
     PhysAddr = Base[Segment] + Offset
     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

address = base + offset - sizeof(stack)

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

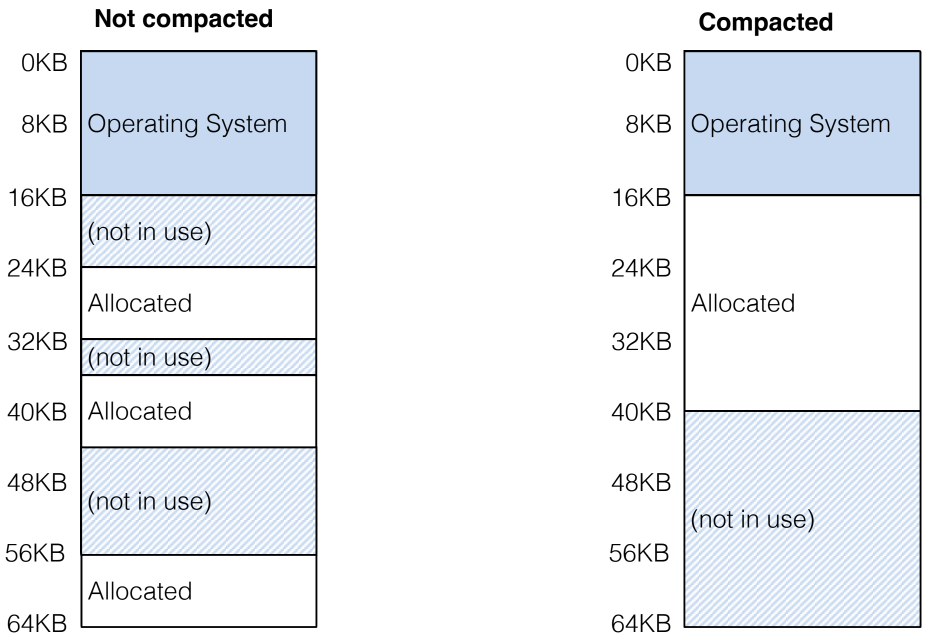
Fine-Grained and Coarse-Grained

- 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** satisfy the **20KB request**.
- Compaction: consolidating existing segments in physical memory.
  - Compaction is **costly**.
    - **Stop** running process.
    - **Copy** data to somewhere.
    - **Change** segment register value.

Memory Compaction



Not compacted

0KB
8KB Operating System
16KB
(not in use)
24KB
Allocated
32KB
(not in use)
40KB Allocated
48KB
(not in use)
56KB
Allocated
64KB

Compacted

0KB
8KB Operating System
16KB
24KB
Allocated
32KB
40KB
48KB
(not in use)
56KB
64KB

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

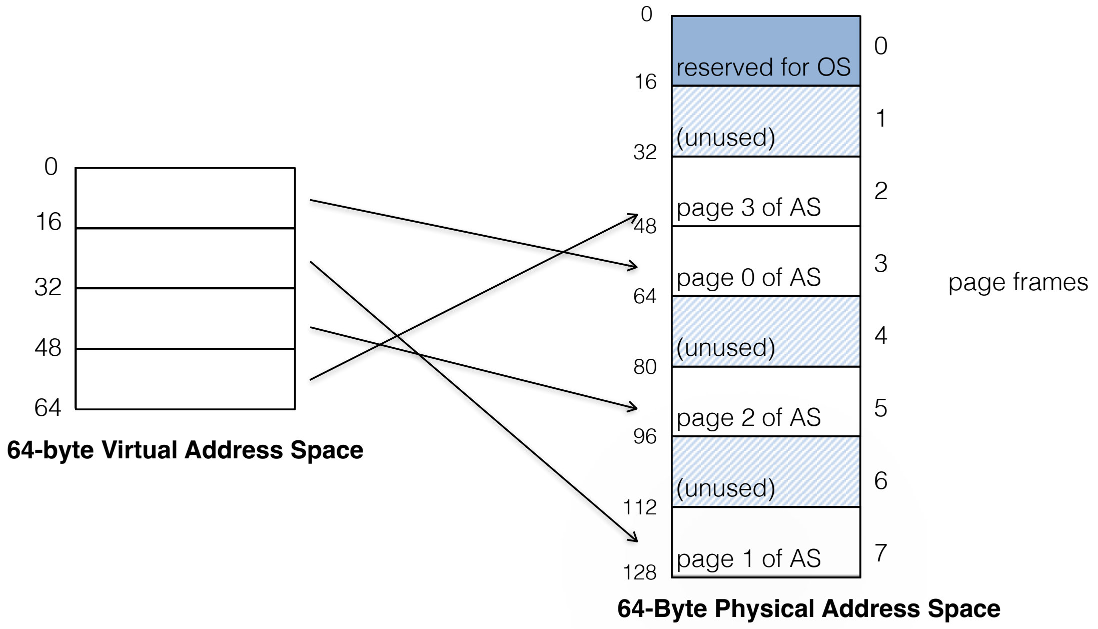
- 14 - Memory API
- 13 - Address Spaces
- 15 - Address Translation
- 16 - Segmentation
- 17 - Free Space Management
- 18 - Paging
- 19 - Translation Lookaside Buffers
- 20 - Advanced Paging
- 21 - Swapping
- 22 - Swapping Policy

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*
- Page table per process
  - translates 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…

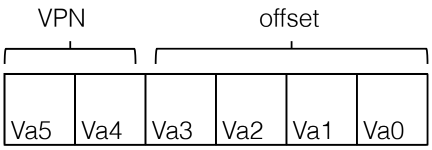
Paging Example

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



Address Translation

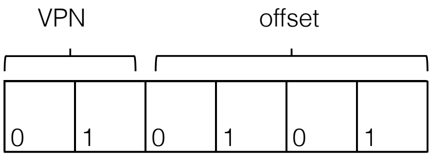
- Two components in the virtual address
  - VPN: virtual page number
  - Offset: offset within the page



VPN offset

Va5 Va4 Va3 Va2 Va1 Va0

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

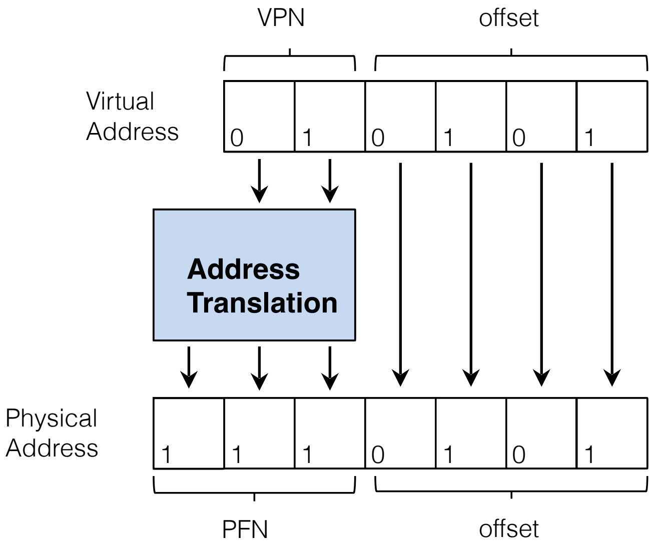


VPN offset

0 1 0 1 0 1

Example: Address Translation

- The virtual address 21 in 64-byte address space



VPN offset

Virtual Address

0 1 0 1 0 1

Address Translation

Physical Address

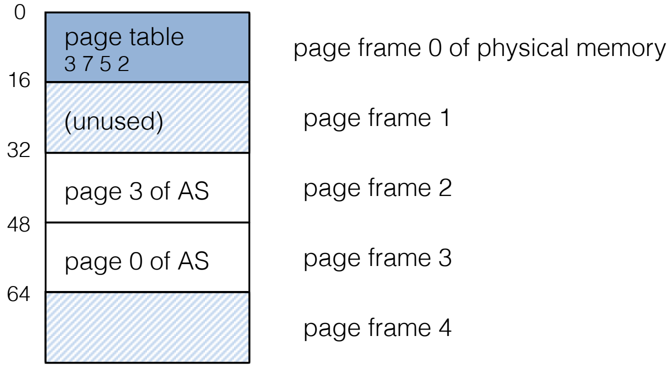
1 1 1 0 1 0 1

PFN offset

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…



0

page table
3 7 5 2

page frame 0 of physical memory

16

(unused)

page frame 1

32

page 3 of AS

page frame 2

48

page 0 of AS

page frame 3

64

page frame 4

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

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
PFN																							G	PAT	D	A	PCD	PWT	U/S	R/W	P

- 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

```
// Extract the VPN from the virtual address
VPN = (VirtualAddress & VPN_MASK) >> SHIFT

// Form the address of the page-table entry (PTE)
PTEAddr = PTBR + (VPN * sizeof(PTE))

// Fetch the PTE
PTE = AccessMemory(PTEAddr)

// Check if process can access the page
if (PTE.Valid == False)
        RaiseException(SEGMENTATION_FAULT)
else if (CanAccess(PTE.ProtectBits) == False)
        RaiseException(PROTECTION_FAULT)
else
        // Access is OK: form physical address and fetch
        offset = VirtualAddress & OFFSET_MASK
        PhysAddr = (PTE.PFN << PFN_SHIFT) | offset
        Register = AccessMemory(PhysAddr)
```

A Memory Trace

- Example: A Simple Memory Access

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

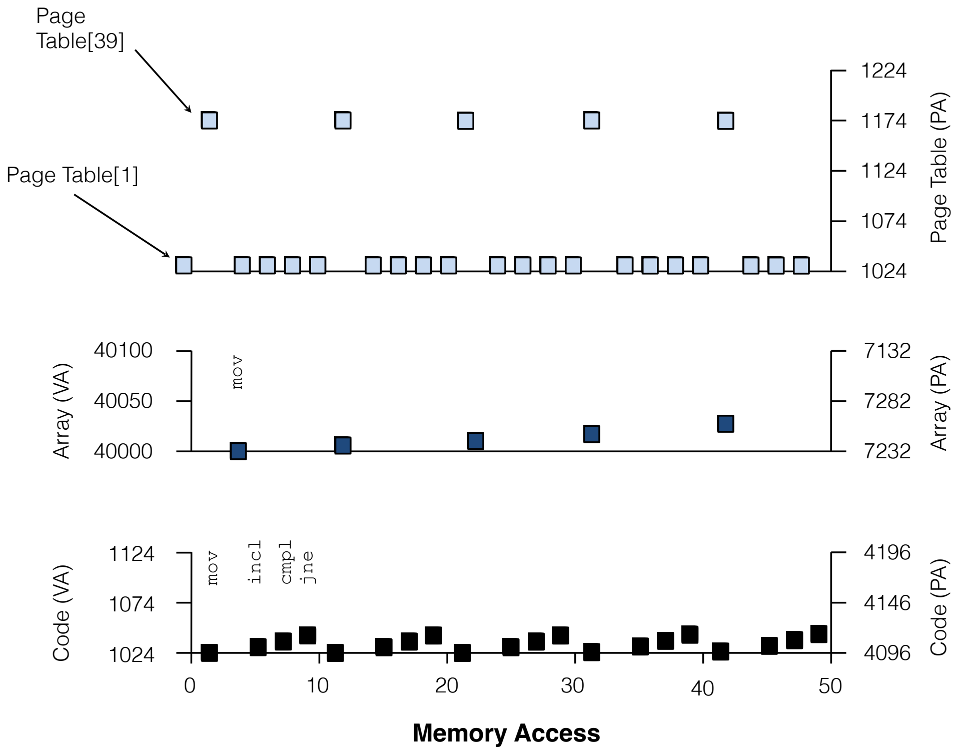
- Compile and execute

```
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

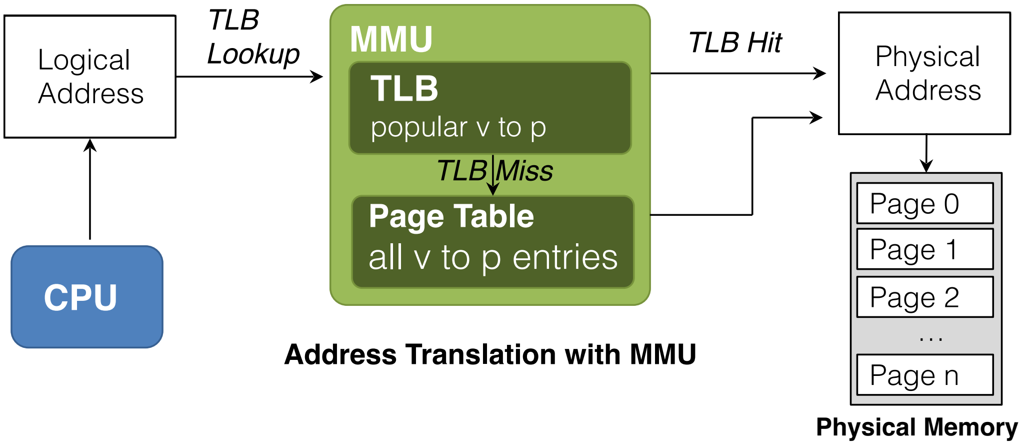


Virtual Memory

- 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

TLB Basic Algorithms

- 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

```
VPN = (VirtualAddress & VPN_MASK ) >> SHIFT
(Success , TlbEntry) = TLB_Lookup(VPN)
   if (Success == True){ // TLB Hit
       if (CanAccess(TlbEntry.ProtectBit) == True ){
           offset = VirtualAddress & OFFSET_MASK
           PhysAddr = (TlbEntry.PFN << SHIFT) | Offset
           AccessMemory( PhysAddr )
       } else RaiseException(PROTECTION_ERROR)
```

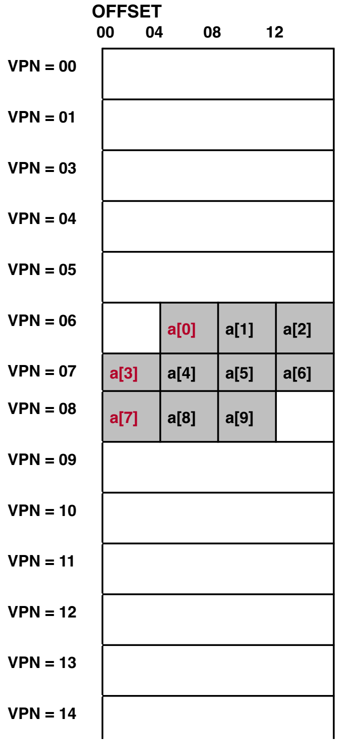
TLB Basic Algorithms (Cont.)

- (11-12 lines) The hardware accesses the page table to find the translation.
- (16 lines) updates the TLB with the translation.

```
11:     } else { //TLB Miss
12:         PTEAddr = PTBR + (VPN * sizeof(PTE))
13:         PTE = AccessMemory(PTEAddr)
14:         (…)
15:     } else {
16:         TLB_Insert( VPN , PTE.PFN , PTE.ProtectBits)
17:         RetryInstruction()
18:     }
19:}
```

Example: Accessing An Array

- How a TLB can improve its performance.



OFFSET

	00	04	08	12
VPN = 00				
VPN = 01				
VPN = 03				
VPN = 04				
VPN = 05				
VPN = 06		a[0]	a[1]	a[2]
VPN = 07	a[3]	a[4]	a[5]	a[6]
VPN = 08	a[7]	a[8]	a[9]	
VPN = 09				
VPN = 10				
VPN = 11				
VPN = 12				
VPN = 13				
VPN = 14				

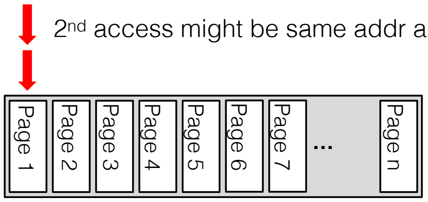
The TLB improves performance due to spatial locality

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

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

Locality

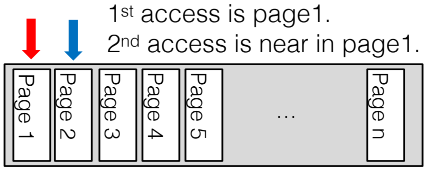
- Temporal Locality
  - An instruction or data item that has been recently accessed will likely be re-accessed soon in the future.



2nd access might be same addr as first

Virtual Memory

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



1st access is page1.
2nd access is near in page1.

Virtual Memory