TCSS 422: OPERATING SYSTEMS

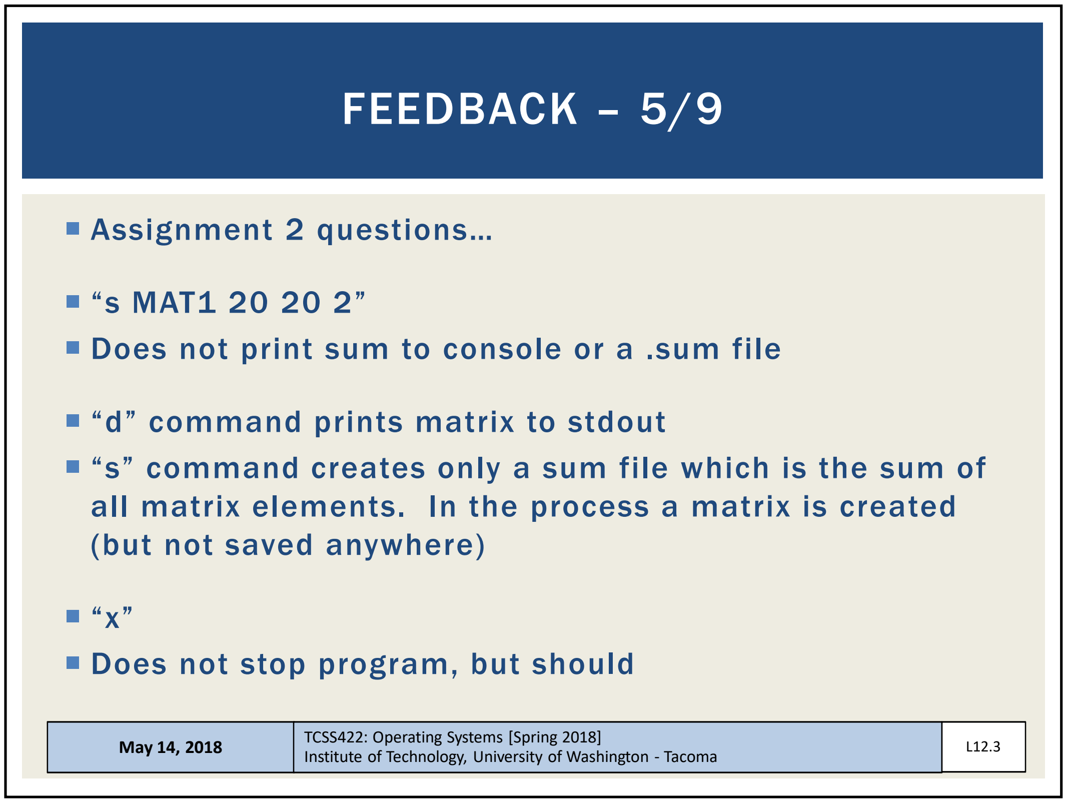
Three Easy Pieces:
Free Space Management,
Introduction to Paging

Wes J. Lloyd
Institute of Technology
University of Washington - Tacoma

OBJECTIVES

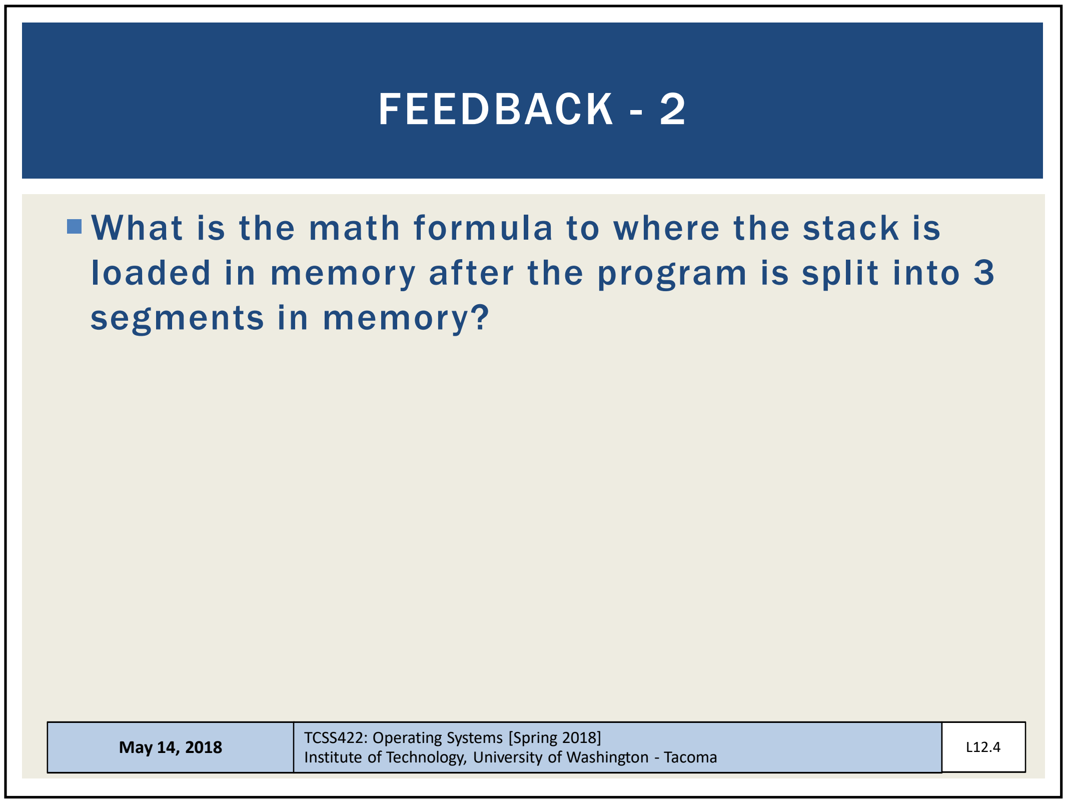
- Assignment 2 – Matrix Task Processor
- Assignment 3 – Posted Tuesday…
- Active reading Quiz #4– Chapter 19
- "Group" Quiz #5 – Wednesday in class

- <u>Memory Virtualization</u>
- Free Space Management – Ch. 17
- Introduction to Paging – Ch. 18
- Translation Lookaside Buffer – Ch. 19



FEEDBACK – 5/9

- Assignment 2 questions…
- "s MAT1 20 20 2"
- Does not print sum to console or a .sum file
- "d" command prints matrix to stdout
- "s" command creates only a sum file which is the sum of all matrix elements. In the process a matrix is created (but not saved anywhere)
- "x"
- Does not stop program, but should



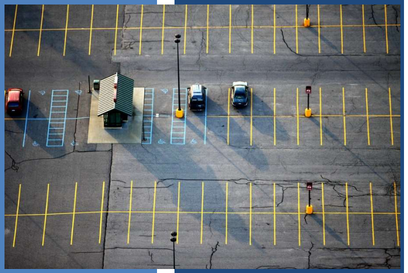
FEEDBACK - 2

- What is the math formula to where the stack is loaded in memory after the program is split into 3 segments in memory?

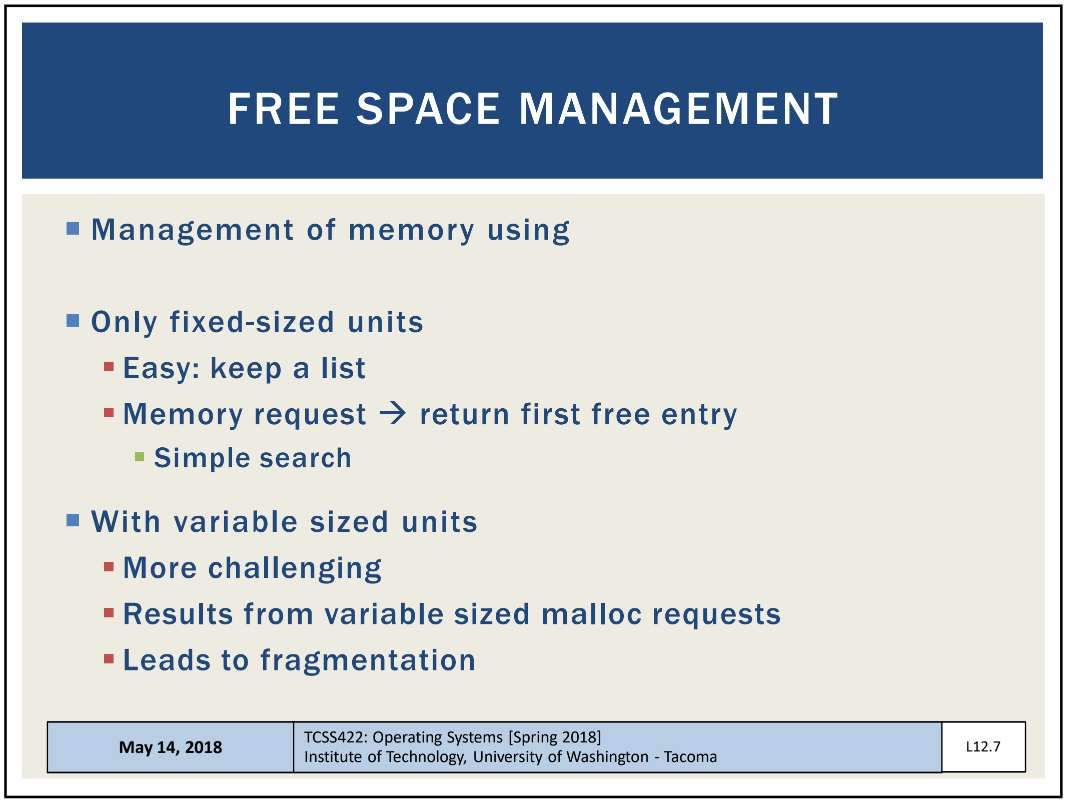
FEEDBACK - 3

- Segment registers – first two bits identify segment type
- Stack bits are "10"
- Consider virtual address: 4200
- VIRTUAL ADDRESS = 1000001101000 (on stack)
- SEG_MASK=0x3000 (10000000000000) **LOGICAL AND THE MASK**
- ***<u>ZEROES OUT everything but the segment bits to learn the segment</u>***
- SEG_SHIFT = 10 → *stack* (mask gives us segment code)
- OFFSET_MASK=0xFFF (00111111111111) ** LOGICAL AND THE MASK **
- ***<u>ZEROES OUT segment bits to reveal the offset address</u>***
- OFFSET = 000001101000 = 104 (isolates segment offset)
- OFFSET < BOUNDS : 104 < 2048

Offset address is the same in virtual & physical memory

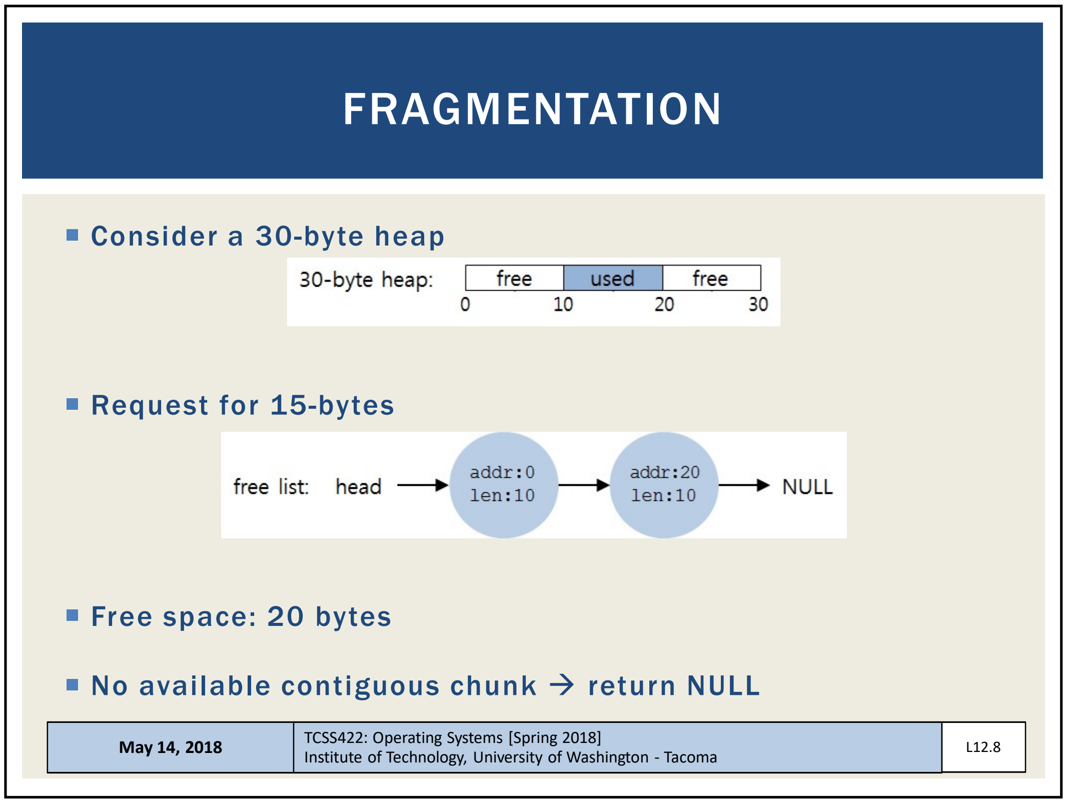


CHAPTER 17: FREE SPACE MANAGEMENT



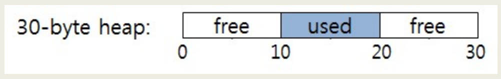
FREE SPACE MANAGEMENT

- Management of memory using
- Only fixed-sized units
  - Easy: keep a list
  - Memory request → return first free entry
    - Simple search
- With variable sized units
  - More challenging
  - Results from variable sized malloc requests
  - Leads to fragmentation

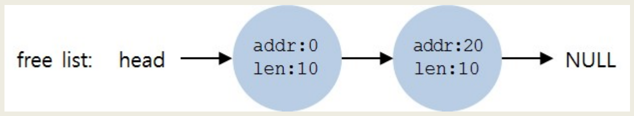


FRAGMENTATION

- Consider a 30-byte heap



- Request for 15-bytes



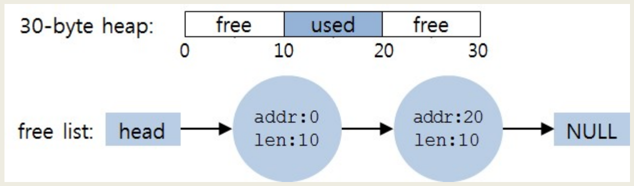
- Free space: 20 bytes
- No available contiguous chunk → return NULL

FRAGMENTATION - 2

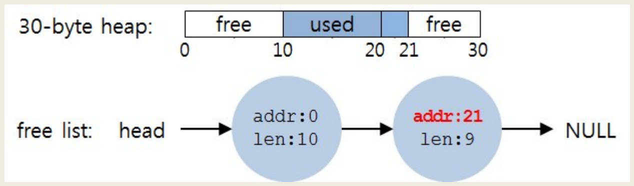
- **External:** *OS can compact*
  - Example: Client asks for 100 bytes: malloc(100)
  - OS: No 100 byte contiguous chunk is available: returns NULL
  - Memory is externally fragmented - - Compaction can fix!
- **Internal:** *lost space – OS can't compact*
  - OS returns memory units that are too large
  - Example: Client asks for 100 bytes: malloc(100)
  - OS: Returns 125 byte chunk
  - Fragmentation is *in* the allocated chunk
  - Memory is lost, and unaccounted for – can't compact

ALLOCATION STRATEGY: SPLITTING

- Request for 1 byte of memory: malloc(1)

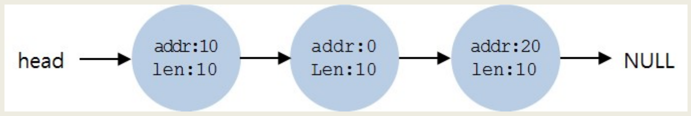


- OS locates a free chunk to satisfy request
- Splits chunk into two, returns first chunk

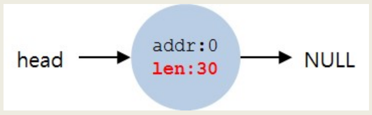


ALLOCATION STRATEGY: COALESCING

- Consider 30-byte heap
- Free() frees all 10 bytes segments *(list of 3-free 10-byte chunks)*



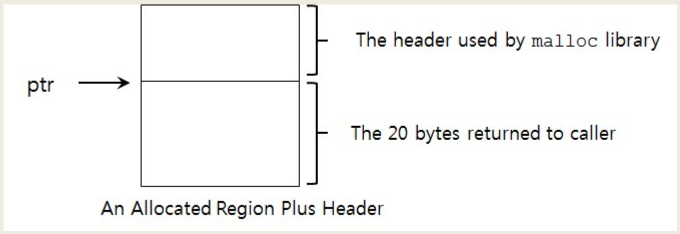
- Request arrives: malloc(30)
- ***SPLIT DOES NOT WORK*** - no contiguous 30-byte chunk exists!
- Coalescing regroups chunks into contiguous chunk



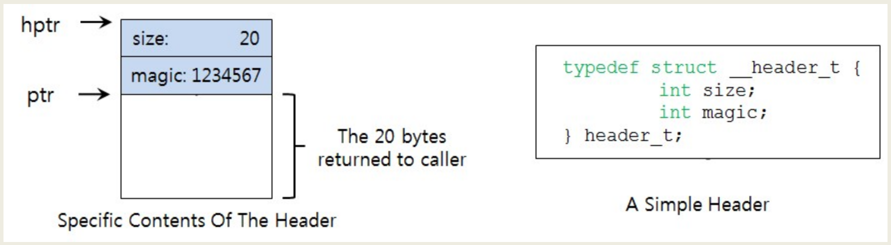
- Allocation can now proceed
- Coalescing is defragmentation of the free space list

MEMORY HEADERS

- free(void *ptr): Does not require a size parameter
- *How does the OS know how much memory to free?*
- Header block
  - Small descriptive block of memory at start of chunk


ptr
The header used by malloc library
The 20 bytes returned to caller
An Allocated Region Plus Header

MEMORY HEADERS - 2


hptr
ptr
size: 20
magic: 1234567
The 20 bytes returned to caller
Specific Contents Of The Header
A Simple Header

```
typedef struct __header_t {
        int size;
        int magic;
} header_t;
```

- Contains size
- Pointers: for faster memory access
- Magic number: integrity checking

MEMORY HEADERS - 3

- Size of memory chunk is:
- Header size + user malloc size
- N bytes + sizeof(header)

- Easy to determine address of header

```
void free(void *ptr) {
        header_t *hptr = (void *)ptr - sizeof(header_t);
}
```

THE FREE LIST

- Simple free list struct

```
typedef struct __node_t {
        int size;
        struct __node_t *next;
} nodet_t;
```

- Use mmap to create free list
- 4kb heap, 4 byte header, one contiguous free chunk

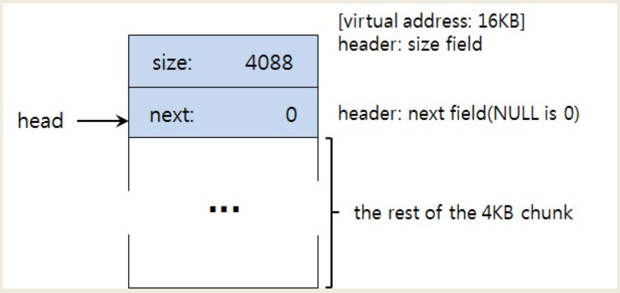
```
// mmap() returns a pointer to a chunk of free space
node_t *head = mmap(NULL, 4096, PROT_READ|PROT_WRITE,
                            MAP_ANON|MAP_PRIVATE, -1, 0);
head->size = 4096 - sizeof(node_t);
head->next = NULL;
```

FREE LIST - 2

- Create and initialize free-list "heap"

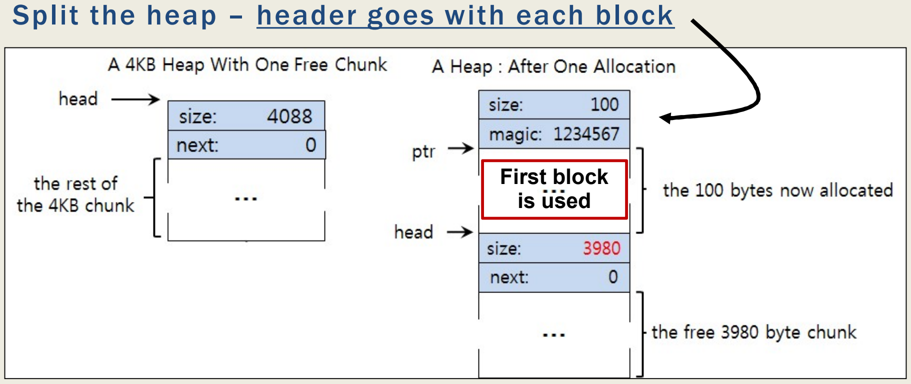
```
// mmap() returns a pointer to a chunk of free space
node_t *head = mmap(NULL, 4096, PROT_READ|PROT_WRITE,
                            MAP_ANON|MAP_PRIVATE, -1, 0);
head->size = 4096 - sizeof(node_t);
head->next = NULL;
```

- Heap layout:



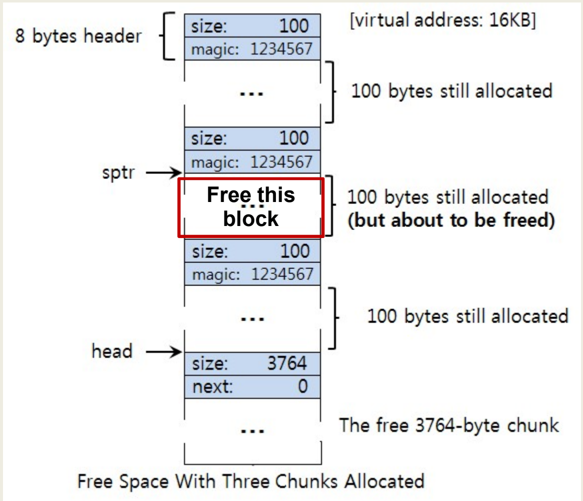
FREE LIST: MALLOC() CALL

- Consider a request for a 100 bytes: malloc(100)
- Header block requires 8 bytes
  - 4 bytes for size, 4 bytes for magic number
- Split the heap – <u>header goes with each block</u>



FREE LIST: FREE() CALL

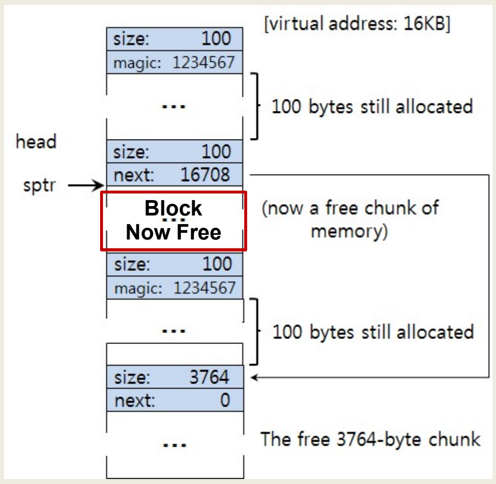
- Addresses of chunks
- Start=16384
  + 108 (end of 1st chunk)
  + 108 (end of 2nd chunk)
  + 108 (end of 3rd chunk)
  = 16708



Free Space With Three Chunks Allocated

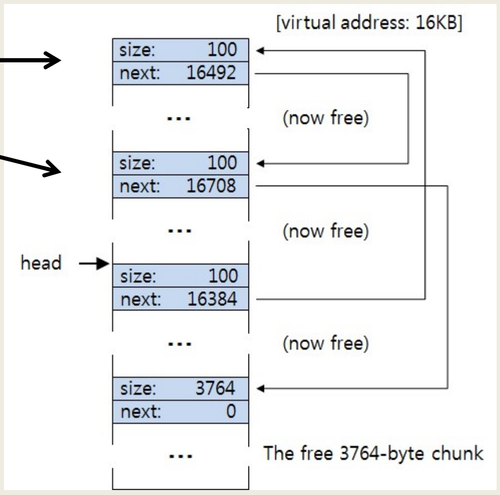
FREE LIST: FREE() CHUNK #2

- Free(sptr)
- Our 3 chunks start at 16 KB (@ 16,384 bytes)
- Free chunk #2 - sptr
- Sptr = 16500
  - addr – sizeof(node_t)
- Actual start of chunk #2
  - 16492



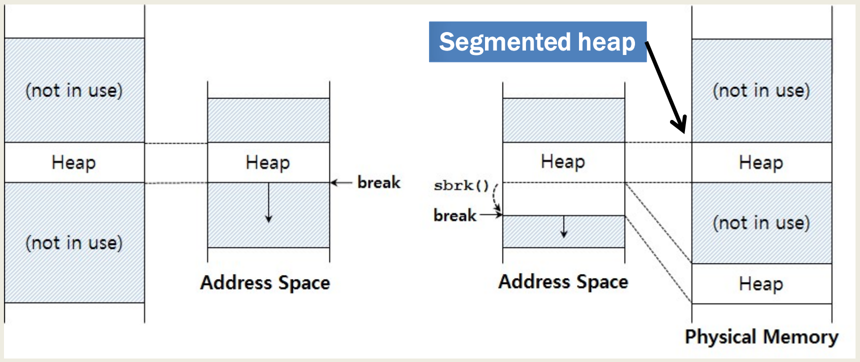
FREE LIST- FREE ALL CHUNKS

- Now free remaining chunks:
- Free(16392)
- Free(16608)
- Walk back 8 bytes for actual start of chunk
- External fragmentation
- Free chunk pointers out of order
- Coalescing of next pointers is needed



GROWING THE HEAP

- Start with small sized heap
- Request more memory when full
- sbrk(), brk()



MEMORY ALLOCATION STRATEGIES

- Best fit
  - Traverse free list
  - Identify all candidate free chunks
  - Note which is smallest (has best fit)
  - When splitting, "leftover" pieces are small (and potentially less useful -- fragmented)

- Worst fit
  - Traverse free list
  - Identify largest free chunk
  - Split largest free chunk, leaving a <u>still large free chunk</u>

EXAMPLES

- Allocation request for 15 bytes

$$\begin{tikzcd} \text{head} \arrow[r] & 10 \arrow[r] & 30 \arrow[r] & 20 \arrow[r] & \text{NULL} \end{tikzcd}$$

- Result of Best Fit

$$\begin{tikzcd} \text{head} \arrow[r] & 10 \arrow[r] & 30 \arrow[r] & 5 \arrow[r] & \text{NULL} \end{tikzcd}$$

- Result of Worst Fit

$$\begin{tikzcd} \text{head} \arrow[r] & 10 \arrow[r] & 15 \arrow[r] & 20 \arrow[r] & \text{NULL} \end{tikzcd}$$

MEMORY ALLOCATION STRATEGIES - 2

- First fit
  - Start search at beginning of free list
  - Find first chunk large enough for request
  - Split chunk, returning a "fit" chunk, saving the remainder
  - Avoids full free list traversal of best and worst fit
- Next fit
  - Similar to first fit, but start search at last search location
  - Maintain a pointer that "cycles" through the list
  - Helps balance chunk distribution vs. first fit
  - Find first chunk, that is large enough for the request, and split
  - Avoids full free list traversal

SEGREGATED LISTS

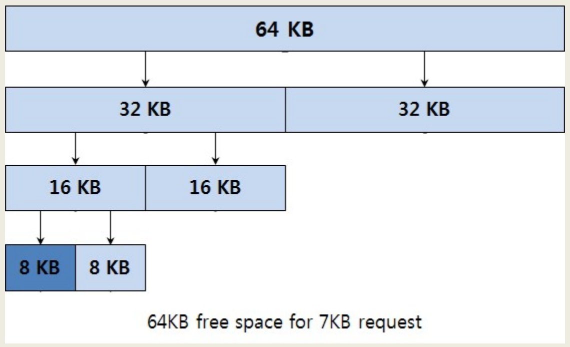
- For popular sized requests
  e.g. for kernel objects such as locks, inodes, etc.
- Manage as segregated free lists
- Provide object caches: stores pre-initialized objects

- How much memory should be dedicated for specialized requests (object caches)?

- If a given cache is low in memory, can request "*slabs*" of memory from the general allocator for caches.
- General allocator will reclaim slabs when not used

BUDDY ALLOCATION

- Binary buddy allocation
  - Divides free space by two to find a block that is big enough to accommodate the request; the next split is too small…
- Consider a 7KB request

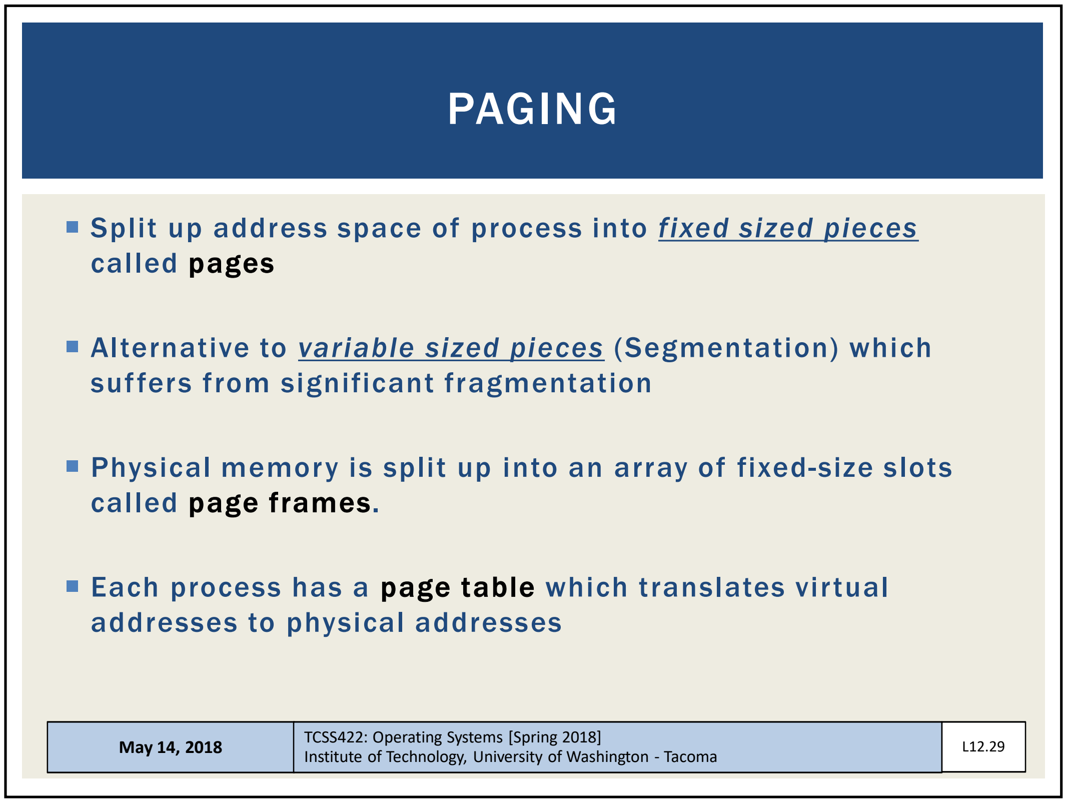


64KB free space for 7KB request

BUDDY ALLOCATION - 2

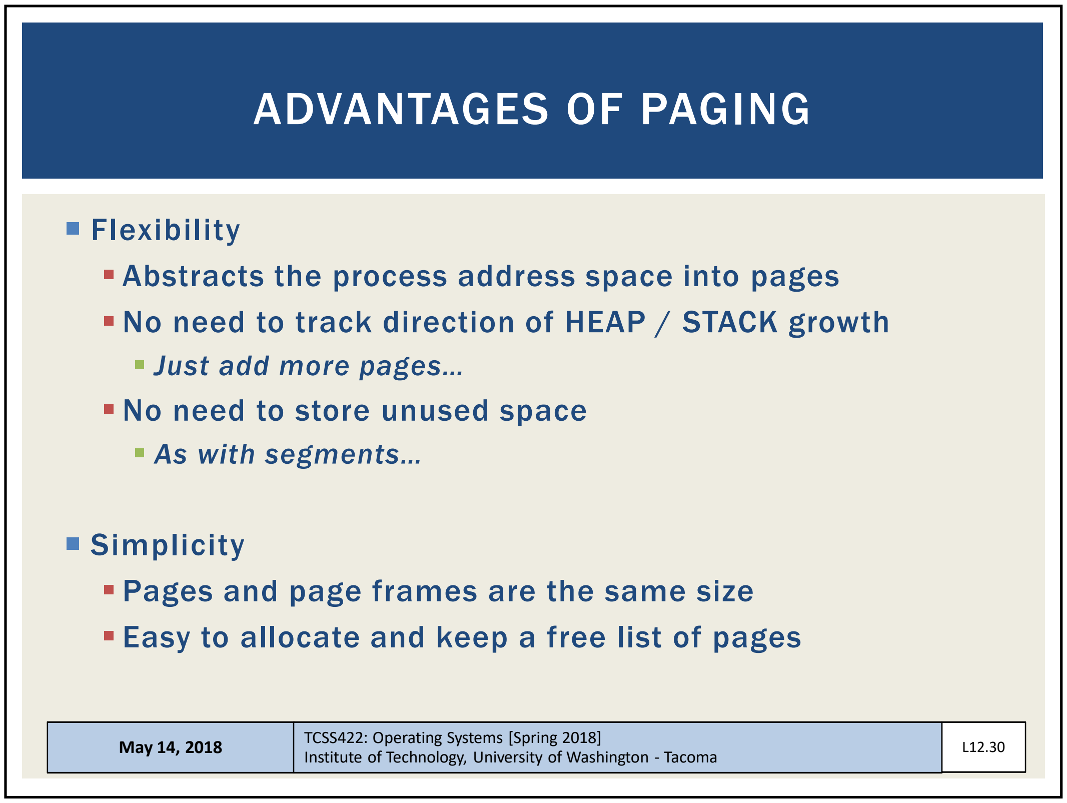
- Buddy allocation: suffers from internal fragmentation
- Allocated fragments, typically too large
- Coalescing is simple
  - Two adjacent blocks are promoted up

CHAPTER 18: INTRODUCTION TO PAGING



PAGING

- Split up address space of process into *fixed sized pieces* called **pages**
- Alternative to *variable sized pieces* (Segmentation) which suffers from significant fragmentation
- Physical memory is split up into an array of fixed-size slots called **page frames**.
- Each process has a **page table** which translates virtual addresses to physical addresses



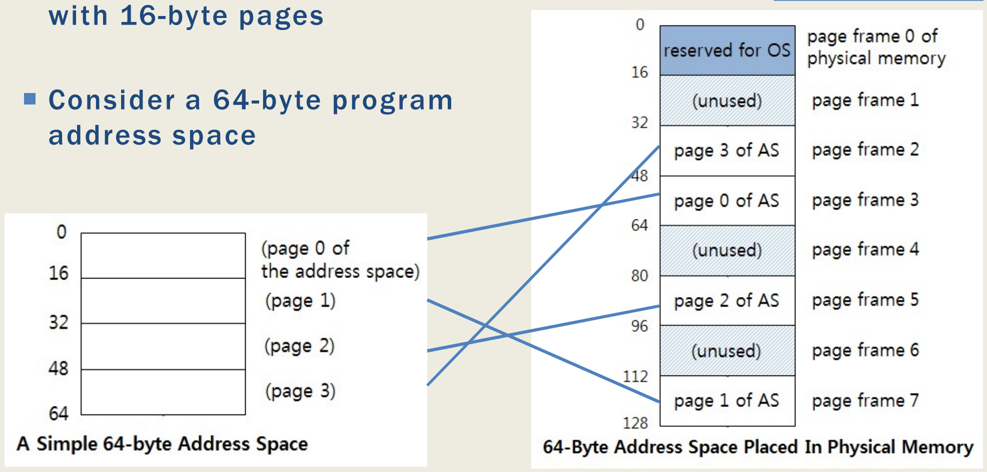
ADVANTAGES OF PAGING

- Flexibility
  - Abstracts the process address space into pages
  - No need to track direction of HEAP / STACK growth
    - *Just add more pages…*
  - No need to store unused space
    - *As with segments…*
- Simplicity
  - Pages and page frames are the same size
  - Easy to allocate and keep a free list of pages

PAGING: EXAMPLE

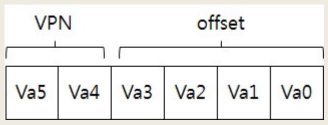
Page Table:
VP0 → PF3
VP1 → PF7
VP2 → PF5
VP3 → PF2

- Consider a 128 byte address space with 16-byte pages
- Consider a 64-byte program address space

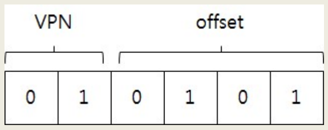


PAGING: ADDRESS TRANSLATION

- PAGE: Has two address components
  - VPN: Virtual Page Number
  - Offset: Offset within a Page



- Example:
  Page Size: 16-bytes, Address Space: 64-bytes



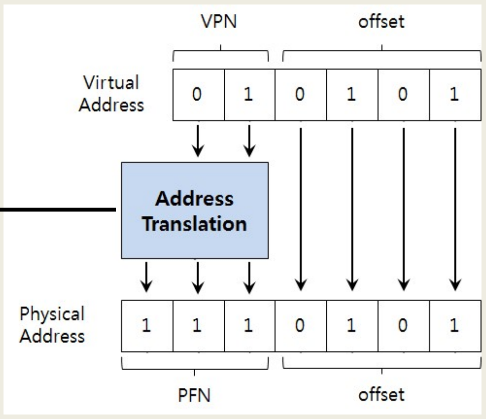
Here there are just four pages…

EXAMPLE: PAGING ADDRESS TRANSLATION

- Consider a 64-byte program address space (4 pages)
- Stored in 128-byte physical memory (8 frames)

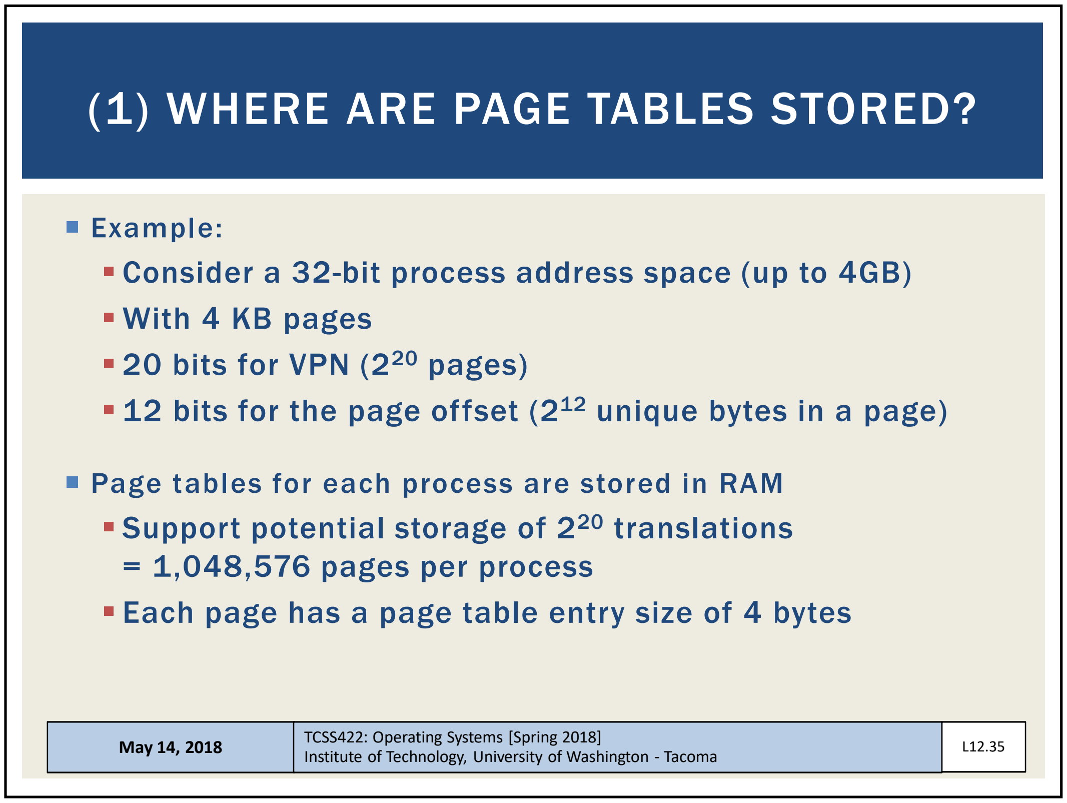
- Offset is preserved
- VPN is looked up

Page Table:
VP0 → PF3
VP1 → PF7
VP2 → PF5
VP3 → PF2



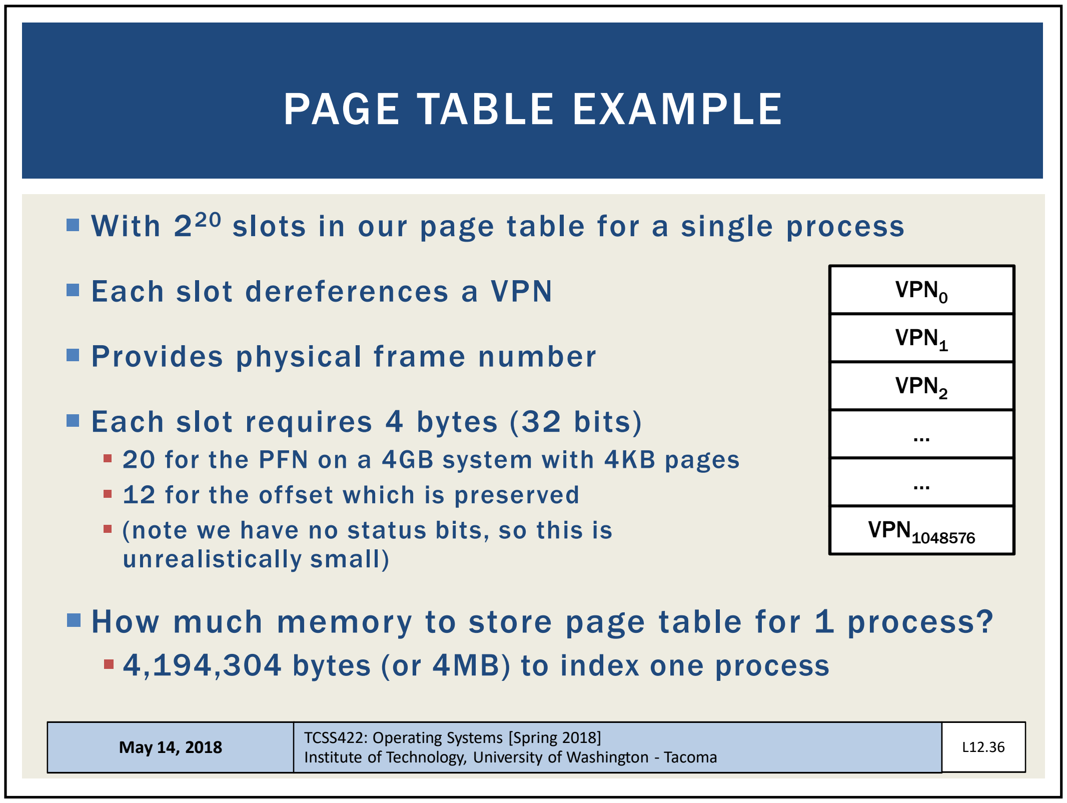
PAGING DESIGN QUESTIONS

- (1) Where are page tables stored?
- (2) What are the typical contents of the page table?
- (3) How big are page tables?
- (4) Does paging make the system too slow?



(1) WHERE ARE PAGE TABLES STORED?

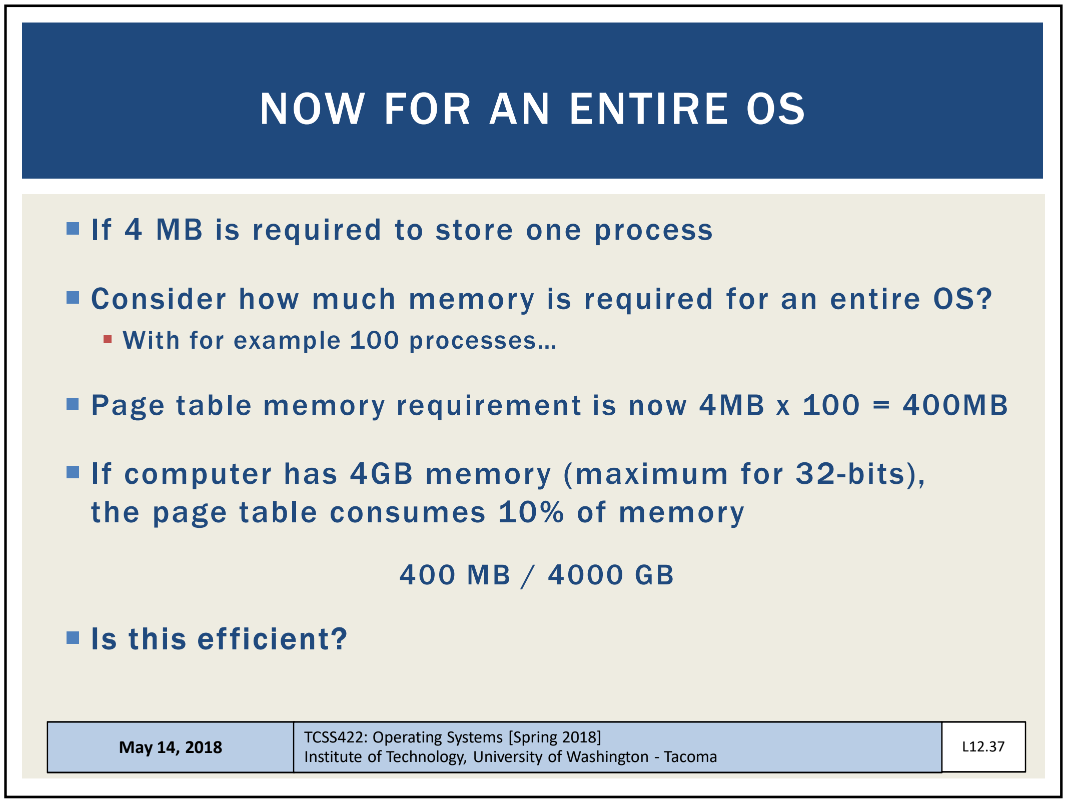
- Example:
  - Consider a 32-bit process address space (up to 4GB)
  - With 4 KB pages
  - 20 bits for VPN (2^{20} pages)
  - 12 bits for the page offset (2^{12} unique bytes in a page)
- Page tables for each process are stored in RAM
  - Support potential storage of 2^{20} translations = 1,048,576 pages per process
  - Each page has a page table entry size of 4 bytes



PAGE TABLE EXAMPLE

- With 2^{20} slots in our page table for a single process
- Each slot dereferences a VPN
- Provides physical frame number
- Each slot requires 4 bytes (32 bits)
  - 20 for the PFN on a 4GB system with 4KB pages
  - 12 for the offset which is preserved
  - (note we have no status bits, so this is unrealistically small)
- How much memory to store page table for 1 process?
  - 4,194,304 bytes (or 4MB) to index one process

VPN_0
VPN_1
VPN_2
…
…
$VPN_{1048576}$



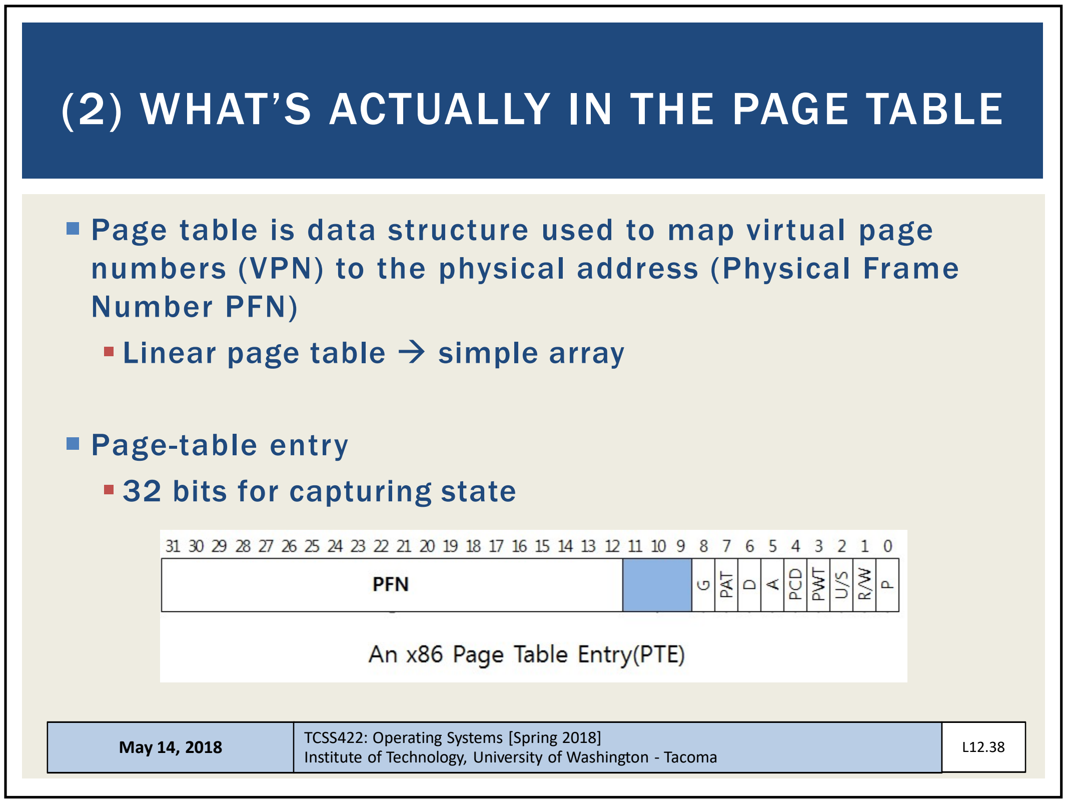
NOW FOR AN ENTIRE OS

- If 4 MB is required to store one process
- Consider how much memory is required for an entire OS?
  - With for example 100 processes…
- Page table memory requirement is now 4MB x 100 = 400MB
- If computer has 4GB memory (maximum for 32-bits), the page table consumes 10% of memory

400 MB / 4000 GB

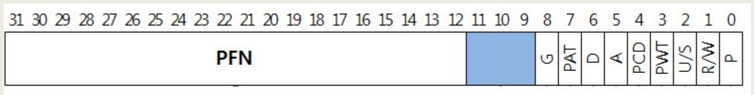
- Is this efficient?

May 14, 2018	TCSS422: Operating Systems [Spring 2018] Institute of Technology, University of Washington - Tacoma	L12.37



(2) WHAT'S ACTUALLY IN THE PAGE TABLE

- Page table is data structure used to map virtual page numbers (VPN) to the physical address (Physical Frame Number PFN)
  - Linear page table → simple array
- Page-table entry
  - 32 bits for capturing state

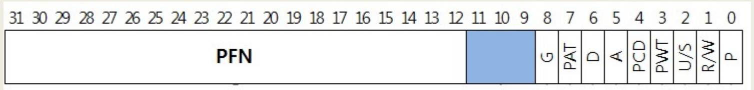


An x86 Page Table Entry(PTE)

May 14, 2018	TCSS422: Operating Systems [Spring 2018] Institute of Technology, University of Washington - Tacoma	L12.38

PAGE TABLE ENTRY

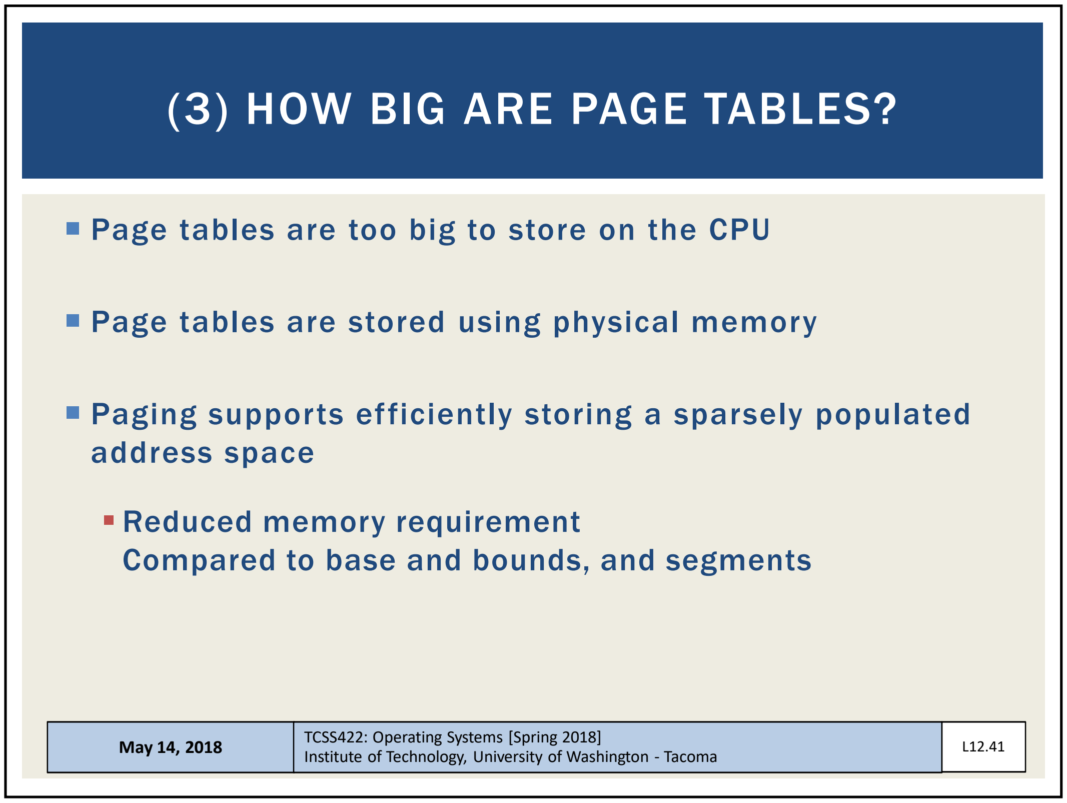
- P: present
- R/W: read/write bit
- U/S: supervisor
- A: accessed bit
- D: dirty bit
- PFN: the page frame number



An x86 Page Table Entry(PTE)

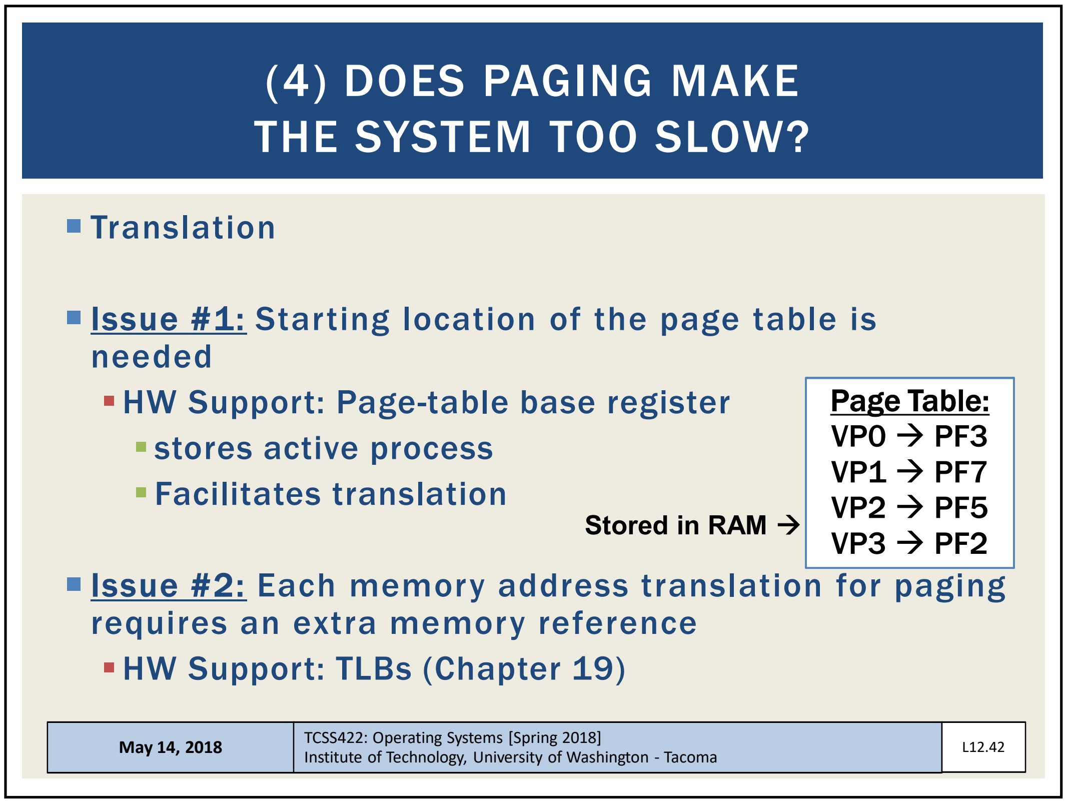
PAGE TABLE ENTRY - 2

- Common flags:
- **Valid Bit:** Indicating whether the particular translation is valid.
- **Protection Bit:** Indicating whether the page could be read from, written to, or executed from
- **Present Bit:** Indicating whether this page is in physical memory or on disk(swapped out)
- **Dirty Bit:** Indicating whether the page has been modified since it was brought into memory
- **Reference Bit(Accessed Bit):** Indicating that a page has been accessed



(3) HOW BIG ARE PAGE TABLES?

- Page tables are too big to store on the CPU
- Page tables are stored using physical memory
- Paging supports efficiently storing a sparsely populated address space
  - Reduced memory requirement
    Compared to base and bounds, and segments



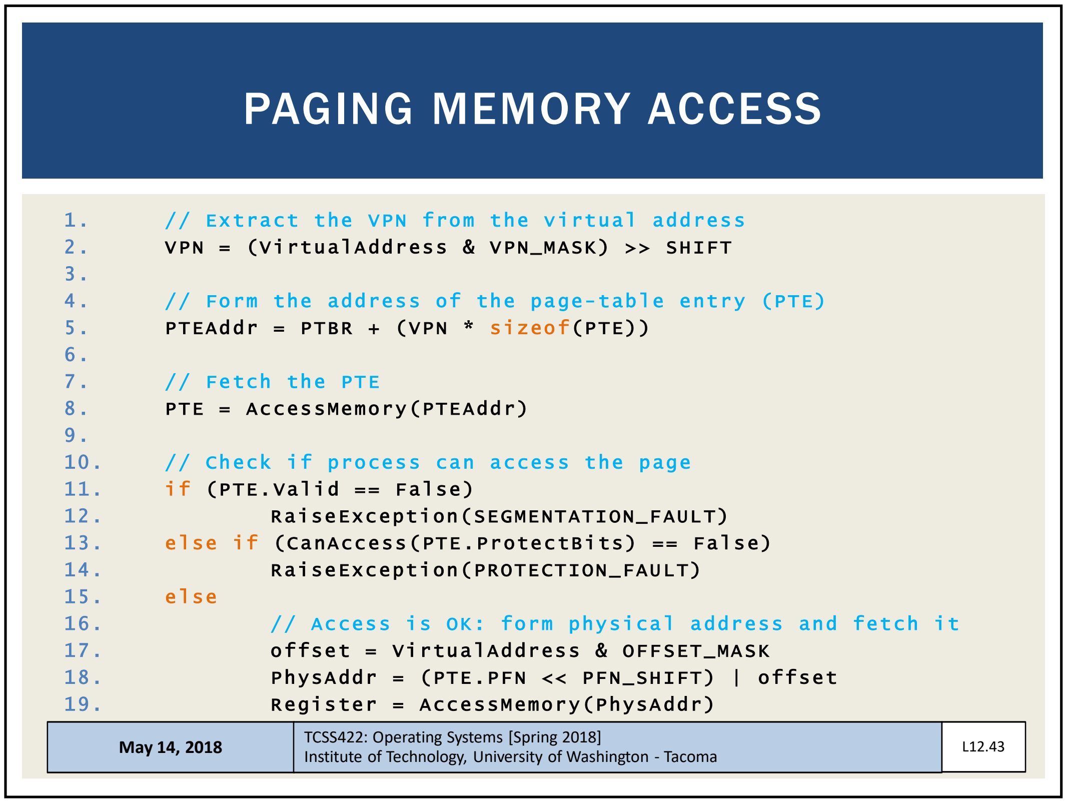
(4) DOES PAGING MAKE THE SYSTEM TOO SLOW?

- Translation
- __Issue #1:__ Starting location of the page table is needed
  - HW Support: Page-table base register
    - stores active process
    - Facilitates translation

Stored in RAM →

__Page Table:__
VP0 → PF3
VP1 → PF7
VP2 → PF5
VP3 → PF2

- __Issue #2:__ Each memory address translation for paging requires an extra memory reference
  - HW Support: TLBs (Chapter 19)



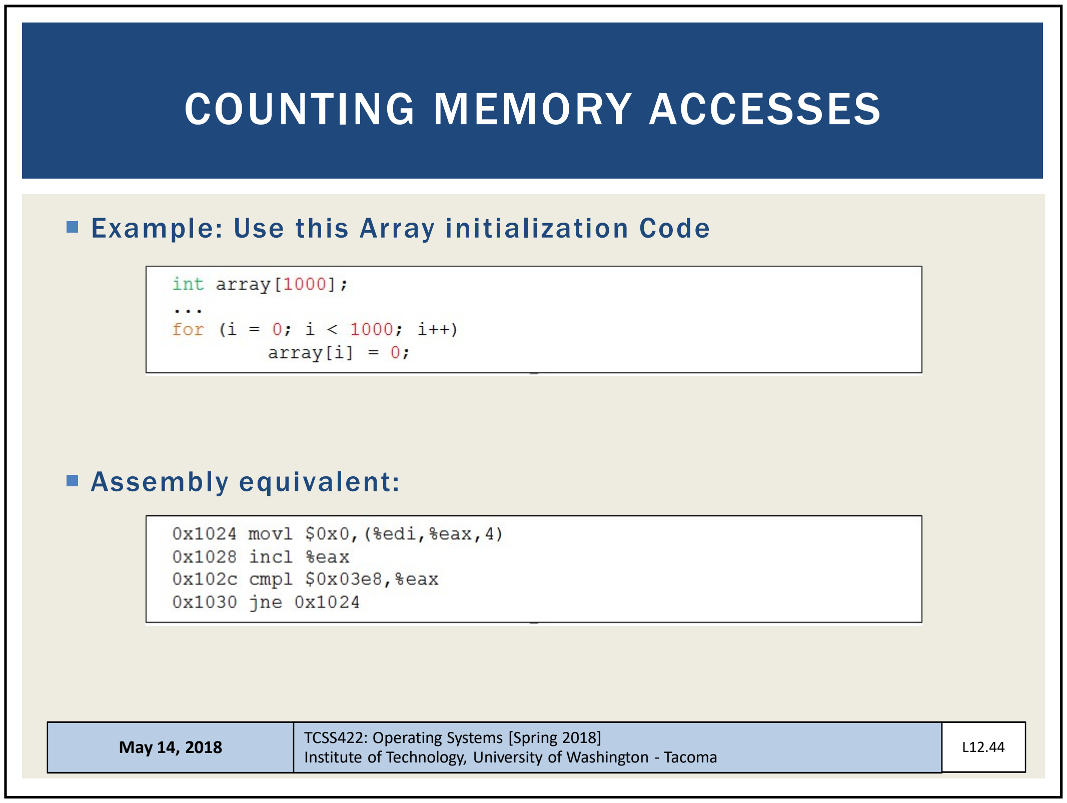
PAGING MEMORY ACCESS

```
// 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 it
        offset = VirtualAddress & OFFSET_MASK
        PhysAddr = (PTE.PFN << PFN_SHIFT) | offset
        Register = AccessMemory(PhysAddr)
```



COUNTING MEMORY ACCESSES

- Example: Use this Array initialization Code

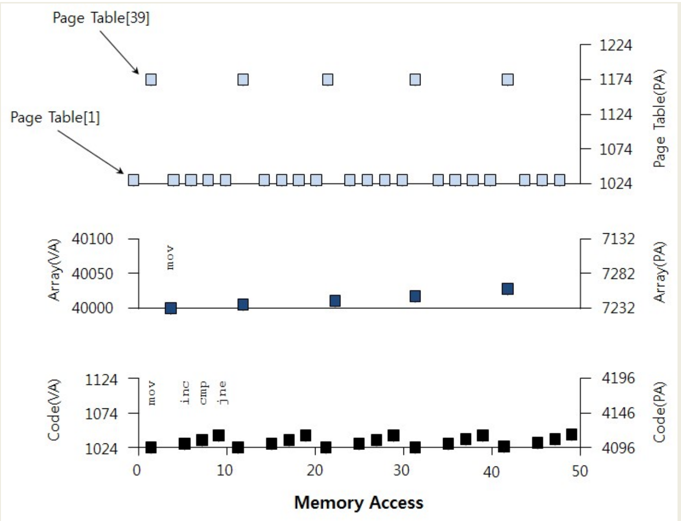
```
int array[1000];
...
for (i = 0; i < 1000; i++)
        array[i] = 0;
```

- Assembly equivalent:

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

VISUALIZING MEMORY ACCESSES: FOR THE FIRST 5 LOOP ITERATIONS

- Locations:
  - Page table
  - Array
  - Code
- 50 accesses for 5 loop iterations



PAGING SYSTEM EXAMPLE

- Consider a 4GB Computer:
- With a 4096-byte page size (4KB)
- How many pages would fit in physical memory?

- Now consider a page table:
- For the page table entry, how many bits are required for the VPN?
- If we assume the use of 4-byte (32 bit) page table entries, how many bits are available for status bits?
- How much space does this page table require? Page Table Entries x Number of pages
- How many page tables (for user processes) would fill the entire 4GB of memory?

CHAPTER 19: TRANSLATION LOOKASIDE BUFFER (TLB)

OBJECTIVES

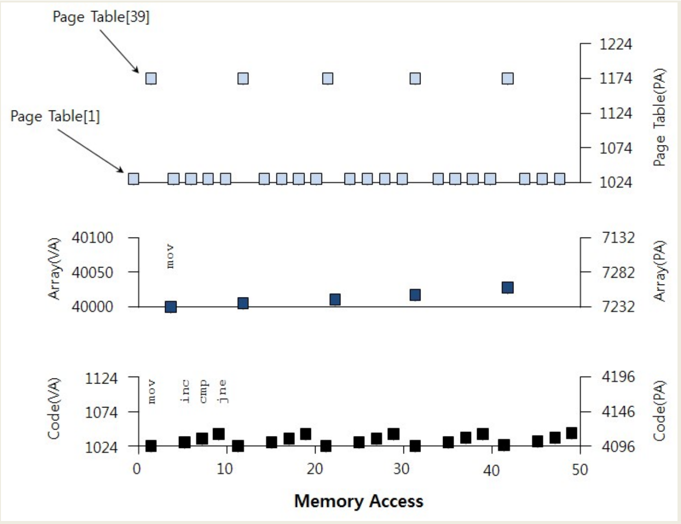
- Chapter 19
  - TLB Algorithm
  - TLB Tradeoffs
  - TLB Context Switch

TRANSLATION LOOKASIDE BUFFER

- Legacy name…
- Better name, “Address Translation Cache”
- TLB is an on CPU cache of address translations
  - virtual → physical memory

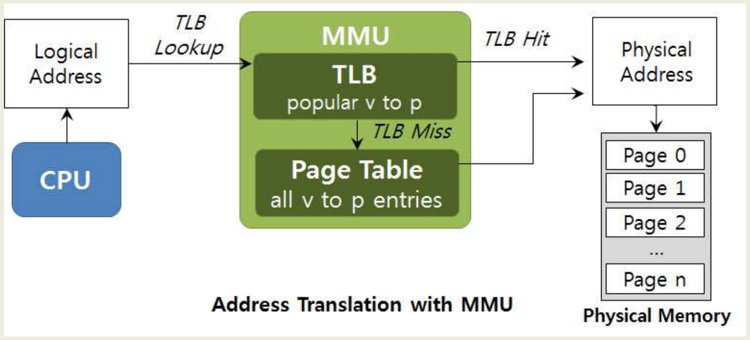
TRANSLATION LOOKASIDE BUFFER - 2

- Goal:
  Reduce access to the page tables
- Example:
  50 RAM accesses for first 5 for-loop iterations
- Move lookups from RAM to TLB by caching page table entries



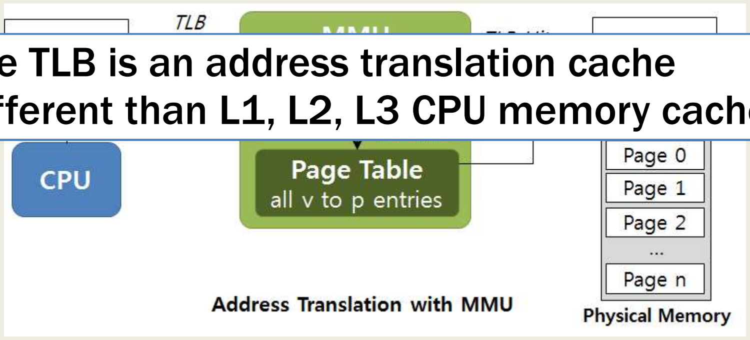
TRANSLATION LOOKASIDE BUFFER (TLB)

- Part of the CPU's Memory Management Unit (MMU)
- Address translation cache



TRANSLATION LOOKASIDE BUFFER (TLB)

- Part of the CPU's Memory Management Unit (MMU)
- Address translation cache



The TLB is an address translation cache
Different than L1, L2, L3 CPU memory caches

TLB BASIC ALGORITHM

- For: array based page table
- Hardware managed TLB

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

Generate the physical address to access memory

TLB BASIC ALGORITHM - 2

```
    else{ //TLB Miss
        PTEAddr = PTBR + (VPN * sizeof(PTE))
        PTE = AccessMemory(PTEAddr)
        (…)  // Check for, and raise exceptions…

        TLB_Insert( VPN , PTE.PFN , PTE.ProtectBits)
        RetryInstruction()
    }
}
```

Retry the instruction... (requery the TLB)

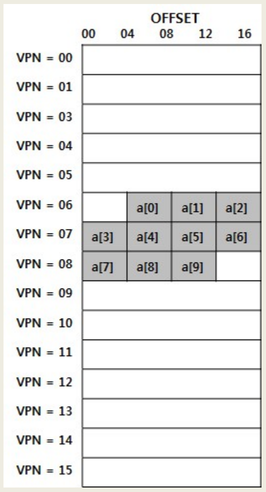
TLB – ADDRESS TRANSLATION CACHE

- Key detail:
- For a TLB miss, we first access the page table in RAM to populate the TLB… we then **requery** the TLB
- **All address translations go through the TLB**

TLB EXAMPLE

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

- Example:
- Program address space: 256-byte
  - Addressable using 8 total bits (2^8)
  - 4 bits for the VPN (16 total pages)
- Page size: 16 bytes
  - Offset is addressable using 4-bits
- Store an array: of (10) 4-byte integers

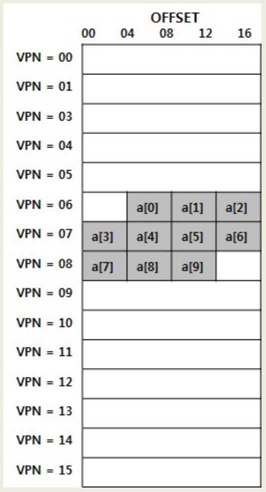


	OFFSET				
	00	04	08	12	16
VPN = 00					
VPN = 01					
VPN = 03					
VPN = 04					
VPN = 05					
VPN = 06		a[0]	a[1]	a[2]	
VPN = 07	a[3]	a[4]	a[5]	a[6]	
VPN = 08	a[7]	a[8]	a[9]		
VPN = 09					
VPN = 10					
VPN = 11					
VPN = 12					
VPN = 13					
VPN = 14					
VPN = 15					

TLB EXAMPLE - 2

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

- Consider the code above:
- Initially the TLB does not know where a[] is
- Consider the accesses:
- a[0], a[1], a[2], a[3], a[4], a[5], a[6], a[7], a[8], a[9]
- How many pages are accessed?
- What happens when accessing a page not in the TLB?

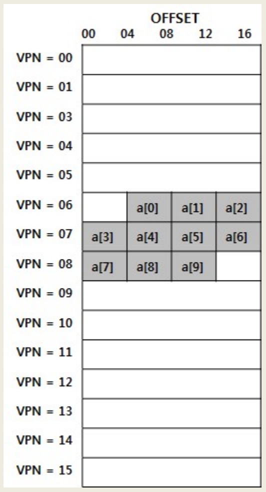

OFFSET 00 04 08 12 16

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				
VPN = 15				

TLB EXAMPLE - 3

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

- For the accesses: a[0], a[1], a[2], a[3], a[4],
- a[5], a[6], a[7], a[8], a[9]
- How many are hits?
- How many are misses?
- What is the hit rate? (%)
  - 70% (3 misses one for each VP, 7 hits)

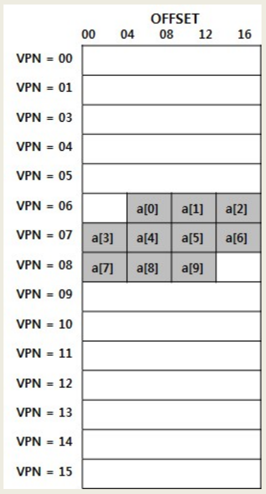

OFFSET 00 04 08 12 16

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				
VPN = 15				

TLB EXAMPLE - 4

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

- What factors affect the hit/miss rate?
  - Page size
  - Data locality
  - Temporal locality



QUESTIONS