

# TCSS 422: OPERATING SYSTEMS



## Smaller Page Tables, Beyond Physical Memory

**Wes J. Lloyd**  
School of Engineering and Technology  
University of Washington - Tacoma

March 13, 2019      TCSS422: Operating Systems [Winter 2019]  
School of Engineering and Technology, University of Washington - Tacoma

## OBJECTIVES

- **Wed 3/13:**            Prof. Mohamed Ali- UWT CSS Grad Program
- **Assignment 3**
- **Paging Practice Questions**
- **Practice Final Exam**
  
- **Memory Virtualization**
- **Chapter 20 - Smaller Page Tables**
- **Chapter 21/22 - Beyond Physical Memory**

March 13, 2019	TCSS422: Operating Systems [Winter 2019] School of Engineering and Technology, University of Washington - Tacoma	L16.2
----------------	---	-------

## FEEDBACK FROM 3/11

- **Are page sizes dynamic (proportionate to total available memory) or fixed?**
- Page sizes are generally fixed to 4KB (4096 bytes)
- Page sizes are “hard wired” into the Memory Management Unit (MMU) of the CPU
- Recently x86\_64 CPUs now support HUGE PAGES
- Check “cat /proc/cpuinfo” for support:
  - 2MB pages: PSE flag
  - 1GB pages: PDPE1GB flag
- See:
  - <https://wiki.debian.org/Hugepages>

March 13, 2019

TCSS422: Operating Systems [Winter 2019]  
School of Engineering and Technology, University of Washington - Tacoma

L16.3

## FEEDBACK - 2

- **What is a Page Directory Entry? (PDE)**
- Page directory entries are in the page directory of a 2-level page table scheme
- Each PDE points to a Page Table
- **What is a Page Table Entry? (PTE)**
- Page table entries include the Physical Frame Number (PFN) bits and status bits to support Virtual to Physical address translation of virtual memory addresses

March 13, 2019

TCSS422: Operating Systems [Winter 2019]  
School of Engineering and Technology, University of Washington - Tacoma

L16.4

## PAGING PRACTICE - ANSWERS

### 1-level page table:

- 1.  $2^{12}$  pages, which is 4,096 pages
- 2. 12 bits for the VPN
- 3. 12 offset bits
- 4. Offset bits are repurposed as status bits
- 5. 3 bytes for each Page Table Entry
- 6.  $2^{12} \times 3$ , which is  $4,096 \times 3 = 12\text{KB} = 12,288$  bytes

### 2-level page table:

- 7. 6 Page Directory Index bits
- 8. 6 Page Table Index bits
- 9.  $2^6 \times 3$  (PD) +  $2^6 \times 3$  (PT) =  $192 + 192 = 384$  bytes
- 10.  $2^6 \times 4\text{KB} = 64 \times 4\text{KB} = 256\text{KB}$

March 13, 2019

TCSS422: Operating Systems [Winter 2019]  
School of Engineering and Technology, University of Washington - Tacoma

L16.5

## PAGING PRACTICE – ANSWERS - 2

- 11. 384 bytes (2-level page table) / 16 MB (computer RAM)  
 $16\text{MB} = 16\text{MB} * 1024\text{KB} * 1024\text{bytes}$   
 $384 / 16,777,216$  bytes  
 $384 / 16,000,000$  bytes (rounded, less precise)  
.0022 % - 2-level pg tbl as % of computer memory  
16MB can map 43,690 hello world programs
- 12. 12 KB / 16 MB  
 $12\text{KB} / 16,384\text{KB}$   
.0732 % - ratio of memory use 2-level to 1 level
- 13.  $384 / 12\text{KB}$  bytes  
 $384 / 12 * 1024$  bytes  
 $384 / 12,288$   
3.125% memory used  
Savings = 100% - 3.125% = 96.875% !!

March 13, 2019

TCSS422: Operating Systems [Winter 2019]  
School of Engineering and Technology, University of Washington - Tacoma

L16.6

**CHAPTER 20:  
PAGING:  
SMALLER TABLES**



March 13, 2019 TCSS422: Operating Systems [Winter 2019]  
School of Engineering and Technology, University of Washington - Tacoma L16.7

**OBJECTIVES**

- Chapter 20
  - Smaller tables
  - Hybrid tables
  - Multi-level page tables

March 13, 2019 TCSS422: Operating Systems [Winter 2019]  
School of Engineering and Technology, University of Washington - Tacoma L16.8

## MULTI-LEVEL PAGE TABLES - 2

■ Add level of indirection, the “page directory”

**Linear Page Table**

PBTR 201

valid	prot	PFN
1	rx	12
1	rx	13
0	-	-
1	rw	100
0	-	-
0	-	-
0	-	-
0	-	-
0	-	-
1	rw	86
1	rw	15

PFN201  
PFN202  
PFN203

**Multi-level Page Table**

PBTR 200

valid	PFN
1	201
0	-
0	-
1	203

The Page Directory

[Page 1 of PT:Not Allocated]

valid	prot	PFN
1	rx	12
1	rx	13
0	-	-
1	rw	100
0	-	-
0	-	-
0	-	-
0	-	-
1	rw	86
1	rw	15

PFN201  
PFN204

**Linear (Left) And Multi-Level (Right) Page Tables**

March 13, 2019
TCCS422: Operating Systems [Winter 2019]  
School of Engineering and Technology, University of Washington - Tacoma
L16.9

## MULTI-LEVEL PAGE TABLES - 2

■ Add level of indirection, the “page directory”

**Linear Page Table**

PBTR 201

valid	prot	PFN
0	-	-
0	-	-
1	rw	86
1	rw	15

PFN203

**Multi-level Page Table**

PBTR 200

valid	PFN
0	-
0	-
1	203

The Page Directory

valid	prot	PFN
0	-	-
0	-	-
1	rw	86
1	rw	15

PFN204

**Two level page table:  
 $2^{20}$  pages addressed with  
 two level-indexing  
 (page directory index, page table index)**

**Linear (Left) And Multi-Level (Right) Page Tables**

March 13, 2019
TCCS422: Operating Systems [Winter 2019]  
School of Engineering and Technology, University of Washington - Tacoma
L16.10

## MULTI-LEVEL PAGE TABLES - 3

- **Advantages**
  - Only allocates page table space in proportion to the address space actually used
  - Can easily grab next free page to expand page table
  
- **Disadvantages**
  - Multi-level page tables are an example of a time-space tradeoff
  - Sacrifice address translation time (now 2-level) for space
  - Complexity: multi-level schemes are more complex

March 13, 2019	TCSS422: Operating Systems [Winter 2019] School of Engineering and Technology, University of Washington - Tacoma	L16.11
----------------	---	--------

## EXAMPLE

- 16KB address space, 64byte pages
- How large would a one-level page table need to be?
- $2^{14} \text{ (address space)} / 2^6 \text{ (page size)} = 2^8 = 256 \text{ (pages)}$

0000 0000	code
0000 0001	code
...	(free)
	(free)
	heap
	heap
	(free)
	(free)
	stack
1111 1111	stack

Flag	Detail
Address space	16 KB
Page size	64 byte
Virtual address	14 bit
VPN	8 bit
Offset	6 bit
Page table entry	$2^8(256)$

**A 16-KB Address Space With 64-byte Pages**

13	12	11	10	9	8	7	6	5	4	3	2	1	0
←----->								←----->					
								Offset					

March 13, 2019	TCSS422: Operating Systems [Winter 2019] School of Engineering and Technology, University of Washington - Tacoma	L16.12
----------------	---	--------

## EXAMPLE - 2

- 256 total page table entries (64 bytes each)
- 1,024 bytes page table size, stored using 64-byte pages  
=  $(1024/64) = 16$  page directory entries (PDEs)
- Each page directory entry (PDE) can hold 16 page table entries (PTEs) e.g. *lookups*
- 16 page directory entries (PDE) x 16 page table entries (PTE)  
= 256 total PTEs
- **Key idea: the page table is stored using pages too!**

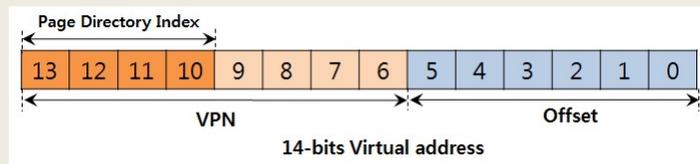
March 13, 2019

TCCS422: Operating Systems [Winter 2019]  
School of Engineering and Technology, University of Washington - Tacoma

L16.13

## PAGE DIRECTORY INDEX

- Now, let's split the page table into two:
  - 8 bit VPN to map 256 pages
  - 4 bits for page directory index (PDI – 1<sup>st</sup> level page table)
  - 6 bits offset into 64-byte page



March 13, 2019

TCCS422: Operating Systems [Winter 2019]  
School of Engineering and Technology, University of Washington - Tacoma

L16.14

## PAGE TABLE INDEX

- 4 bits page directory index (PDI - 1<sup>st</sup> level)
- 4 bits page table index (PTI - 2<sup>nd</sup> level)

14-bits Virtual address

- To dereference one 64-byte memory page,
  - We need one page directory entry (PDE)
  - One page table Index (PTI) - can address 16 pages

March 13, 2019	TCCS422: Operating Systems [Winter 2019] School of Engineering and Technology, University of Washington - Tacoma	L16.15
----------------	---	--------

## EXAMPLE - 3

- For this example, how much space is required to store as a single-level page table with any number of PTEs?
  - 16KB address space, 64 byte pages
  - 256 page frames, 4 byte page size
  - 1,024 bytes required (*single level*)
- How much space is required for a two-level page table with only 4 page table entries (PTEs) ?
  - Page directory = 16 entries x 4 bytes (1 x 64 byte page)
  - Page table = 4/16 entries x 4 bytes (1 x 64 byte page)
  - 128 bytes required (2 x 64 byte pages)
    - Savings = using just 12.5% the space !!!

March 13, 2019	TCCS422: Operating Systems [Winter 2019] School of Engineering and Technology, University of Washington - Tacoma	L16.16
----------------	---	--------

## 32-BIT EXAMPLE

- Consider: 32-bit address space, 4KB pages,  $2^{20}$  pages
- Only 4 mapped pages
  
- Single level: 4 MB (we've done this before)
  
- Two level: (old VPN was 20 bits, split in half)
- Page directory =  $2^{10}$  entries x 4 bytes = 1 x 4 KB page
- Page table = 4 entries x 4 bytes (mapped to 1 4KB page)
- 8KB (8,192 bytes) required
- Savings = using just .78 % the space !!!
  
- 100 sparse processes now require < 1MB for page tables

March 13, 2019	TCCS422: Operating Systems [Winter 2019] School of Engineering and Technology, University of Washington - Tacoma	L16.17
----------------	---	--------

## MORE THAN TWO LEVELS

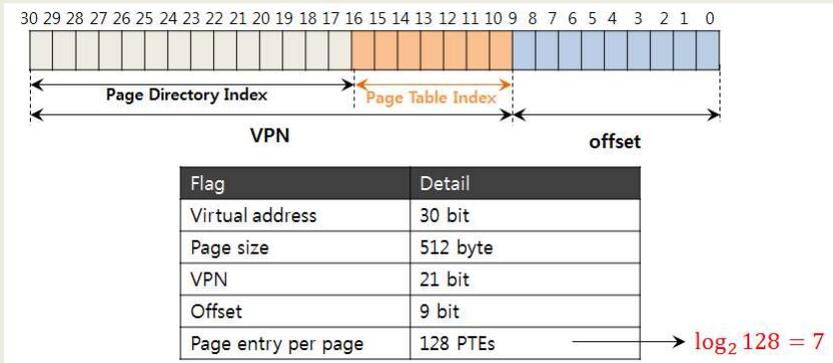
- Consider: page size is  $2^9 = 512$  bytes
- Page size 512 bytes / Page entry size 4 bytes
- VPN is 21 bits

Flag	Detail
Virtual address	30 bit
Page size	512 byte
VPN	21 bit
Offset	9 bit

March 13, 2019	TCCS422: Operating Systems [Winter 2019] School of Engineering and Technology, University of Washington - Tacoma	L16.18
----------------	---	--------

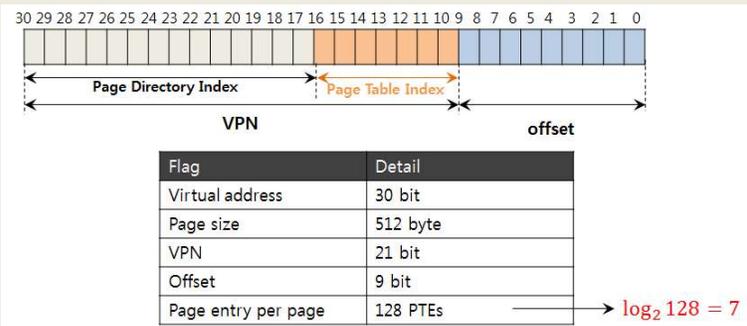
## MORE THAN TWO LEVELS - 2

- Page table entries per page =  $512 / 4 = 128$
- 7 bytes - for page table index (PTI)



## MORE THAN TWO LEVELS - 3

- To map 1 GB address space ( $2^{30}=1\text{GB RAM}$ , 512-byte pages)
- $2^{14} = 16,384$  page directory entries (PDEs) are required
- When using  $2^7$  (128 entry) page tables...
- Page size = 512 bytes / 4 bytes per addr



## MORE THAN TWO LEVELS - 3

- To map 1 GB address space ( $2^{30}=1\text{GB}$  RAM, 512-byte pages)
- $2^{14} = 16,384$  page directory entries (PDEs) are required
- When using  $2^7$  (128 entry) page tables...
- Page size = 512 bytes / 4 bytes per addr

**Can't Store Page Directory with 16K pages, using 512 bytes pages.  
 Pages only dereference 128 addresses  
 (512 bytes / 32 bytes)**

Virtual address	30 bit	
Page size	512 byte	
VPN	21 bit	
Offset	9 bit	
Page entry per page	128 PTEs	→ $\log_2 128 = 7$

March 13, 2019
TCCS422: Operating Systems [Winter 2019]  
 School of Engineering and Technology, University of Washington - Tacoma
L16.21

## MORE THAN TWO LEVELS - 3

- To map 1 GB address space ( $2^{30}=1\text{GB}$  RAM, 512-byte pages)
- $2^{14} = 16,384$  page directory entries (PDEs) are required
- When using  $2^7$  (128 entry) page tables...
- Page size = 512 bytes / 4 bytes per addr

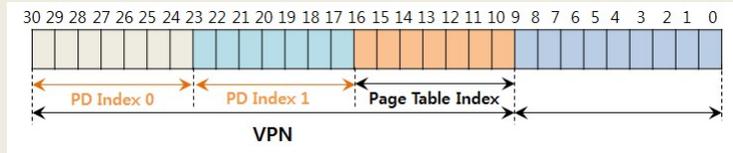
**Need three level page table:  
 Page directory 0 (PD Index 0)  
 Page directory 1 (PD Index 1)  
 Page Table Index**

Virtual address	30 bit	
Page size	512 byte	
VPN	21 bit	
Offset	9 bit	
Page entry per page	128 PTEs	→ $\log_2 128 = 7$

March 13, 2019
TCCS422: Operating Systems [Winter 2019]  
 School of Engineering and Technology, University of Washington - Tacoma
L16.22

# MORE THAN TWO LEVELS - 4

- We can now address 1GB with “fine grained” 512 byte pages
- Using multiple levels of indirection



- Consider the implications for address translation!
- How much space is required for a virtual address space with 4 entries on a 512-byte page? (let’s say 4 32-bit integers)
- PD0 1 page, PD1 1 page, PT 1 page = 1,536 bytes
- Memory Usage=  $1,536 (3-level) / 8,388,608 (1-level) = .0183\% !!!$

March 13, 2019	TCCS422: Operating Systems [Winter 2019] School of Engineering and Technology, University of Washington - Tacoma	L16.23
----------------	---	--------

# ADDRESS TRANSLATION CODE

```

// 5-level Linux page table address lookup
//
// Inputs:
// mm_struct - process's memory map struct
// vpage - virtual page address

// Define page struct pointers
pgd_t *pgd;
p4d_t *p4d;
pud_t *pud;
pmd_t *pmd;
pte_t *pte;
struct page *page;
```

March 13, 2019	TCCS422: Operating Systems [Winter 2019] School of Engineering and Technology, University of Washington - Tacoma	L16.24
----------------	---	--------

## ADDRESS TRANSLATION - 2

```

pgd = pgd_offset(mm, vpage);
if (pgd_none(*pgd) || pgd_bad(*pgd))
    return 0;
p4d = p4d_offset(pgd, vpage);
if (p4d_none(*p4d) || p4d_bad(*p4d))
    return 0;
pud = pud_offset(p4d, vpage);
if (pud_none(*pud) || pud_bad(*pud))
    return 0;
pmd = pmd_offset(pud, vpage);
if (pmd_none(*pmd) || pmd_bad(*pmd))
    return 0;
if (!(pte = pte_offset_map(pmd, vpage)))
    return 0;
if (!(page = pte_page(*pte)))
    return 0;
physical_page_addr = page_to_phys(page);
pte_unmap(pte);
return physical_page_addr; // param to send back
    
```

### **pgd\_offset():**

Takes a vpage address and the mm\_struct for the process, returns the PGD entry that covers the requested address...

### **p4d/pud/pmd\_offset():**

Takes a vpage address and the pgd/p4d/pud entry and returns the relevant p4d/pud/pmd.

### **pte\_unmap()**

release temporary kernel mapping for the page table entry

March 13, 2019

TCSS422: Operating Systems [Winter 2019]  
 School of Engineering and Technology, University of Washington - Tacoma

L16.25

## INVERTED PAGE TABLES



- Keep a single page table for each physical page of memory
- Consider 4GB physical memory
- Using 4KB pages, page table requires 4MB to map all of RAM
- Page table stores
  - Which process uses each page
  - Which process virtual page (from process virtual address space) maps to the physical page
- All processes share the same page table for memory mapping, kernel must isolate all use of the shared structure
- Finding process memory pages requires search of  $2^{20}$  pages
- Hash table: can index memory and speed lookups

March 13, 2019

TCSS422: Operating Systems [Winter 2019]  
 School of Engineering and Technology, University of Washington - Tacoma

L16.26

## MULTI-LEVEL PAGE TABLE EXAMPLE

- Consider a 16 MB computer which indexes memory using 4KB pages
- **(#1)** For a single level page table, how many pages are required to index memory?
- **(#2)** How many bits are required for the VPN?
- **(#3)** Assuming each page table entry (PTE) can index any byte on a 4KB page, how many offset bits are required?
- **(#4)** Assuming there are 8 status bits, how many bytes are required for each page table entry?

March 13, 2019

TCCS422: Operating Systems [Winter 2019]  
School of Engineering and Technology, University of Washington - Tacoma

L16.27

## MULTI LEVEL PAGE TABLE EXAMPLE - 2

- **(#5)** How many bytes (or KB) are required for a single level page table?
- Let's assume a simple HelloWorld.c program.
- HelloWorld.c requires virtual address translation for 4 pages:
  - 1 - code page
  - 1 - heap page
  - 1 - stack page
  - 1 - data segment page
- **(#6)** Assuming a two-level page table scheme, how many bits are required for the Page Directory Index (PDI)?
- **(#7)** How many bits are required for the Page Table Index (PTI)?

March 13, 2019

TCCS422: Operating Systems [Winter 2019]  
School of Engineering and Technology, University of Washington - Tacoma

L16.28

## MULTI LEVEL PAGE TABLE EXAMPLE - 3

- Assume each page directory entry (PDE) and page table entry (PTE) requires 4 bytes:
  - 6 bits for the Page Directory Index (PDI)
  - 6 bits for the Page Table Index (PTI)
  - 12 offset bits
  - 8 status bits
- (#8) How much **total** memory is required to index the HelloWorld.c program using a two-level page table when we only need to translate 4 total pages?
- **HINT:** we need to allocate one Page Directory and one Page Table...
- **HINT:** how many entries are in the PD and PT

March 13, 2019

TCSS422: Operating Systems [Winter 2019]  
School of Engineering and Technology, University of Washington - Tacoma

L16.29

## MULTI LEVEL PAGE TABLE EXAMPLE - 4

- (#9) Using a single page directory entry (PDE) pointing to a single page table (PT), if all of the slots of the page table (PT) are in use, what is the total amount of memory a two-level page table scheme can address?
- (#10) And finally, for this example, as a percentage (%), how much memory does the 2-level page table scheme consume compared to the 1-level scheme?
- **HINT:** two-level memory use / one-level memory use

March 13, 2019

TCSS422: Operating Systems [Winter 2019]  
School of Engineering and Technology, University of Washington - Tacoma

L16.30

## ANSWERS

- #1 - 4096 pages
- #2 - 12 bits
- #3 - 12 bits
- #4 - 4 bytes
- #5 -  $4096 \times 4 = 16,384$  bytes (16KB)
- #6 - 6 bits
- #7 - 6 bits
- #8 - 256 bytes for Page Directory (PD) (64 entries x 4 bytes)  
256 bytes for Page Table (PT) **TOTAL = 512 bytes**
- #9 - 64 entries, where each entry maps a 4,096 byte page  
With 12 offset bits, can address 262,144 bytes (256 KB)
- #10-  $512/16384 = .03125 \rightarrow 3.125\%$

March 13, 2019

TCSS422: Operating Systems [Winter 2019]  
School of Engineering and Technology, University of Washington - Tacoma

L16.31

## CHAPTER 21/22: BEYOND PHYSICAL MEMORY



March 13, 2019

TCSS422: Operating Systems [Winter 2019]  
School of Engineering and Technology, University of Washington - Tacoma

L16.32

## MEMORY HIERARCHY

- Disks (HDD, SSD) provide another level of storage in the memory hierarchy

Registers  
Cache  
Main Memory  
Mass Storage( hard disk, tape, etc...)  
Memory Hierarchy in modern system

March 13, 2019	TCSS422: Operating Systems [Winter 2019] School of Engineering and Technology, University of Washington - Tacoma	L16.33
----------------	---	--------

## MOTIVATION FOR EXPANDING THE ADDRESS SPACE

- Can provide illusion of an address space larger than physical RAM
- For a single process
  - Convenience
  - Ease of use
- For multiple processes
  - Large virtual memory space for many concurrent processes

March 13, 2019	TCSS422: Operating Systems [Winter 2019] School of Engineering and Technology, University of Washington - Tacoma	L16.34
----------------	---	--------

## LATENCY TIMES

- Design considerations
  - SSDs 4x the time of DRAM
  - HDDs 80x the time of DRAM

Action	Latency (ns)	( $\mu$ s)	
L1 cache reference	0.5ns		
L2 cache reference	7 ns		14x L1 cache
Mutex lock/unlock	25 ns		
Main memory reference	100 ns		20x L2 cache, 200x L1
Read 4K randomly from SSD*	150,000 ns	150 $\mu$ s	~1GB/sec SSD
Read 1 MB sequentially from memory	250,000 ns	250 $\mu$ s	
Read 1 MB sequentially from SSD*	1,000,000 ns	1,000 $\mu$ s	1 ms ~1GB/sec SSD, 4X memory
Read 1 MB sequentially from disk	20,000,000 ns	20,000 $\mu$ s	20 ms 80x memory, 20X SSD

- Latency numbers every programmer should know
- From: <https://gist.github.com/jboner/2841832#file-latency-txt>

March 13, 2019	TCSS422: Operating Systems [Winter 2019] School of Engineering and Technology, University of Washington - Tacoma	L16.35
----------------	---	--------

## SWAP SPACE

- Disk space for storing memory pages
- “Swap” them in and out of memory to disk as needed

	PFN 0	PFN 1	PFN 2	PFN 3
<b>Physical Memory</b>	Proc 0 [VPN 0]	Proc 1 [VPN 2]	Proc 1 [VPN 3]	Proc 2 [VPN 0]

	Block 0	Block 1	Block 2	Block 3	Block 4	Block 5	Block 6	Block 7
<b>Swap Space</b>	Proc 0 [VPN 1]	Proc 0 [VPN 2]	[Free]	Proc 1 [VPN 0]	Proc 1 [VPN 1]	Proc 3 [VPN 0]	Proc 2 [VPN 1]	Proc 3 [VPN 1]

**Physical Memory and Swap Space**

March 13, 2019	TCSS422: Operating Systems [Winter 2019] School of Engineering and Technology, University of Washington - Tacoma	L16.36
----------------	---	--------

## PAGE LOCATION

- Page table pages are:
  - Stored in memory
  - Swapped to disk
- Present bit
  - In the page table entry (PTE) indicates if page is present
- Page fault
  - Memory page is accessed, but has been swapped to disk

March 13, 2019

TCCS422: Operating Systems [Winter 2019]  
School of Engineering and Technology, University of Washington - Tacoma

L16.37

## PAGE FAULT

- OS steps in to handle the page fault
- Loading page from disk requires a free memory page
- Page-Fault Algorithm

```
1:     PFN = FindFreePhysicalPage ()
2:     if (PFN == -1)                // no free page found
3:         PFN = EvictPage ()        // run replacement algorithm
4:     DiskRead (PTE.DiskAddr, pfn)  // sleep (waiting for I/O)
5:     PTE.present = True           // set PTE bit to present
6:     PTE.PFN = PFN                // reference new loaded page
7:     RetryInstruction ()          // retry instruction
```

March 13, 2019

TCCS422: Operating Systems [Winter 2019]  
School of Engineering and Technology, University of Washington - Tacoma

L16.38

## PAGE REPLACEMENTS

- Page daemon
  - Background threads which monitors swapped pages
- Low watermark (LW)
  - Threshold for when to swap pages to disk
  - Daemon checks: free pages < LW
  - Begin swapping to disk until reaching the highwater mark
- High watermark (HW)
  - Target threshold of free memory pages
  - Daemon free until: free pages >= HW

March 13, 2019

TCSS422: Operating Systems [Winter 2019]  
School of Engineering and Technology, University of Washington - Tacoma

L16.39

## REPLACEMENT POLICIES



March 13, 2019

TCSS422: Operating Systems [Winter 2019]  
School of Engineering and Technology, University of Washington - Tacoma

L16.4  
0

## CACHE MANAGEMENT

- Replacement policies apply to “any” cache
- Goal is to minimize the number of misses
- Average memory access time can be estimated:

$$AMAT = (P_{Hit} * T_M) + (P_{Miss} * T_D)$$

Argument	Meaning
$T_M$	The cost of accessing memory (time)
$T_D$	The cost of accessing disk (time)
$P_{Hit}$	The probability of finding the data item in the cache(a hit)
$P_{Miss}$	The probability of not finding the data in the cache(a miss)

- Consider  $T_M = 100 \text{ ns}$ ,  $T_D = 10\text{ms}$
- Consider  $P_{hit} = .9$  (90%),  $P_{miss} = .1$
- Consider  $P_{hit} = .999$  (99.9%),  $P_{miss} = .001$

March 13, 2019	TCSS422: Operating Systems [Winter 2019] School of Engineering and Technology, University of Washington - Tacoma	L16.41
----------------	---	--------

## OPTIMAL REPLACEMENT POLICY

- What if:
  - We could predict the future (... with a magical oracle)
  - All future page accesses are known
  - Always replace the page in the cache used farthest in the future
- Used for a comparison
- Provides a “best case” replacement policy
- Consider a 3-element empty cache with the following page accesses:

0 1 2 0 1 3 0 3 1 2 1

What is the hit/miss ratio?

6 hits

March 13, 2019	TCSS422: Operating Systems [Winter 2019] School of Engineering and Technology, University of Washington - Tacoma	L16.42
----------------	---	--------

## FIFO REPLACEMENT

- Queue based
- Always replace the oldest element at the back of cache
- Simple to implement
- Doesn't consider importance... just arrival ordering
- Consider a 3-element empty cache with the following page accesses:

0 1 2 0 1 3 0 3 1 2 1

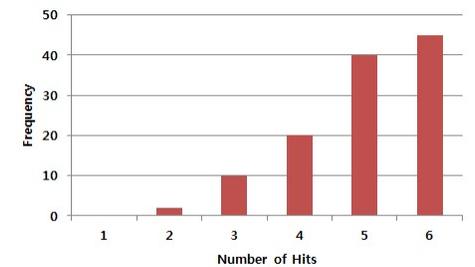
- What is the hit/miss ratio? 4 hits
- How is FIFO different than LRU? LRU incorporates history

March 13, 2019	TCSS422: Operating Systems [Winter 2019] School of Engineering and Technology, University of Washington - Tacoma	L16.43
----------------	---	--------

## RANDOM REPLACEMENT

- Pick a page at random to replace
- Simple and fast implementation
- Performance depends on luck of random choices

0 1 2 0 1 3 0 3 1 2 1



Number of Hits	Frequency
1	0
2	~2
3	10
4	20
5	40
6	45

March 13, 2019	TCSS422: Operating Systems [Winter 2019] School of Engineering and Technology, University of Washington - Tacoma	L16.44
----------------	---	--------

## HISTORY-BASED POLICIES

- LRU: Least recently used
- Always replace page with oldest access time (front)
- Always move end of cache when element is read again
- Considers temporal locality (*when pg was last accessed*)

0 1 2 0 1 3 0 3 1 2 1

What is the hit/miss ratio?

6 hits

- LFU: Least frequently used
- Always replace page with fewest accesses (front)
- Consider frequency of page accesses

0 1 2 0 1 3 0 3 1 2 1

Hit/miss ratio is=

6 hits

March 13, 2019	TCCS422: Operating Systems [Winter 2019] School of Engineering and Technology, University of Washington - Tacoma	L16.45
----------------	---	--------

## WORKLOAD EXAMPLES: NO-LOCALITY

- No-Locality (Random Access) Workload
  - Perform 10,000 random page accesses
  - Across set of 100 memory pages

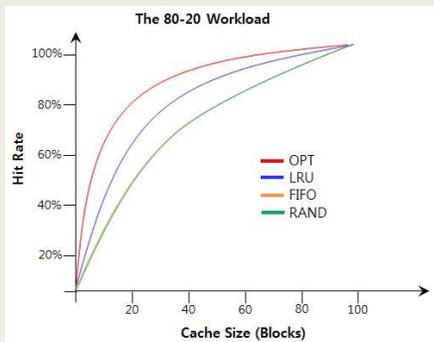
When the cache is large enough to fit the entire workload, it doesn't matter which policy you use.

March 13, 2019	TCCS422: Operating Systems [Winter 2019] School of Engineering and Technology, University of Washington - Tacoma	L16.46
----------------	---	--------

## WORKLOAD EXAMPLES: 80/20

### 80/20 Workload

- Perform 10,000 page accesses, against set of 100 pages
- 80% of accesses are to 20% of pages (hot pages)
- 20% of accesses are to 80% of pages (cold pages)



LRU is more likely to hold onto hot pages  
(recalls history)

March 13, 2019

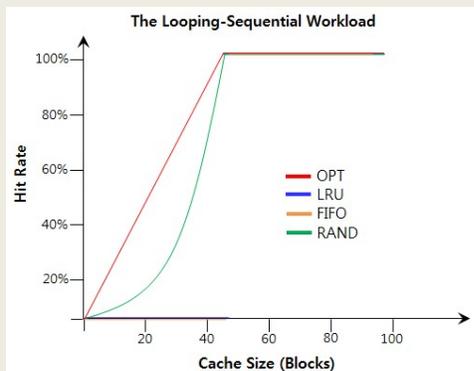
TCSS422: Operating Systems [Winter 2019]  
School of Engineering and Technology, University of Washington - Tacoma

L16.47

## WORKLOAD EXAMPLES: SEQUENTIAL

### Looping sequential workload

- Refer to 50 pages in sequence: 0, 1, ..., 49
- Repeat loop



Random performs better than FIFO and LRU for cache sizes < 50

Algorithms should provide "scan resistance"

March 13, 2019

TCSS422: Operating Systems [Winter 2019]  
School of Engineering and Technology, University of Washington - Tacoma

L16.48

## IMPLEMENTING LRU

- Implementing last recently used (LRU) requires tracking access time for all system memory pages
- Times can be tracked with a list
- For cache eviction, we must scan an entire list
- Consider: 4GB memory system ( $2^{32}$ ), with 4KB pages ( $2^{12}$ )
  
- This requires  $2^{20}$  comparisons !!!
  
- Simplification is needed
  - Consider how to approximate the oldest page access

March 13, 2019

TCSS422: Operating Systems [Winter 2019]  
School of Engineering and Technology, University of Washington - Tacoma

L16.49

## IMPLEMENTING LRU - 2

- Harness the Page Table Entry (PTE) Use Bit
- HW sets to 1 when page is used
- OS sets to 0
  
- Clock algorithm (*approximate LRU*)
  - Refer to pages in a circular list
  - Clock hand points to current page
  - Loops around
    - IF USE\_BIT=1 set to USE\_BIT = 0
    - IF USE\_BIT=0 replace page



March 13, 2019

TCSS422: Operating Systems [Winter 2019]  
School of Engineering and Technology, University of Washington - Tacoma

L16.50

## CLOCK ALGORITHM

- Not as efficient as LRU, but better than other replacement algorithms that do not consider history

Cache Size (Blocks)	OPT	LRU	Clock	FIFO	RAND
10	85%	65%	55%	45%	35%
20	95%	80%	70%	60%	50%
40	100%	95%	85%	75%	65%
60	100%	100%	95%	85%	75%
80	100%	100%	100%	95%	85%
100	100%	100%	100%	100%	100%

March 13, 2019	TCSS422: Operating Systems [Winter 2019] School of Engineering and Technology, University of Washington - Tacoma	L16.51
----------------	---	--------

## CLOCK ALGORITHM - 2

- Consider dirty pages in cache
  - If DIRTY (modified) bit is FALSE
    - No cost to evict page from cache
  - If DIRTY (modified) bit is TRUE
    - Cache eviction requires updating memory
    - Contents have changed
- Clock algorithm should favor no cost eviction

March 13, 2019	TCSS422: Operating Systems [Winter 2019] School of Engineering and Technology, University of Washington - Tacoma	L16.52
----------------	---	--------

## WHEN TO LOAD PAGES

- On demand → demand paging
- Prefetching
  - Preload pages based on anticipated demand
  - Prediction based on locality
  - Access page P, suggest page P+1 may be used
- What other techniques might help anticipate required memory pages?
  - Prediction models, historical analysis
  - In general: accuracy vs. effort tradeoff
  - High analysis techniques struggle to respond in real time

March 13, 2019

TCSS422: Operating Systems [Winter 2019]  
School of Engineering and Technology, University of Washington - Tacoma

L16.53

## OTHER SWAPPING POLICIES

- Page swaps / writes
  - Group/cluster pages together
  - Collect pending writes, perform as batch
  - Grouping disk writes helps amortize latency costs
- Thrashing
  - Occurs when system runs many memory intensive processes and is low in memory
  - Everything is constantly swapped to-and-from disk

March 13, 2019

TCSS422: Operating Systems [Winter 2019]  
School of Engineering and Technology, University of Washington - Tacoma

L16.54

## OTHER SWAPPING POLICIES - 2

- Working sets
  - Groups of related processes
  - When thrashing: prevent one or more working set(s) from running
  - Temporarily reduces memory burden
  - Allows some processes to run, reduces thrashing

March 13, 2019

TCSS422: Operating Systems [Winter 2019]  
School of Engineering and Technology, University of Washington - Tacoma

L16.55

## QUESTIONS

