TCSS 422: OPERATING SYSTEMS

Paging: Smaller Tables, Beyond Physical Memory

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

OBJECTIVES – 5/26

- Questions from 5/21
- Tutorial 2 (pthreads, locks, conditions) – due Thurs June 4
- Quiz 3 posted – Active Reading Ch. 19 – due Tues June 2
- Assignment 2 (based on Ch. 30) – due …… *Thurs May 28*
- Assignment 3 – on Linux kernel programming – to be offered in "tutorial" format - posting by ~May 28
- Chapter 20: Paging: Smaller Tables
  - Smaller Tables, Hybrid Tables, Multi-level Page Tables
- Chapter 21/22: Beyond Physical Memory
  - Swapping Mechanisms
  - Swapping Policies

MATERIAL / PACE

- Please classify your perspective on material covered in today's class (46 respondents):
- 1-mostly review, 5-equal new/review, 10-mostly new
- **Average – 6.29 (↓ *from 6.53*)**

- Please rate the pace of today's class:
- 1-slow, 5-just right, 10-fast
- **Average – 5.70 (↑ from 5.66)**

FEEDBACK FROM 5/21

- ***From email: On assignment 2, I am confused about examining load with "top -d .1"***

(from A2) On a multicore machine, when monitoring load with "top -d .1", the max percent CPU utilization demonstrates the highest degree of parallelism achieved. On an 8-hyperthead computer, 800% is possible. On a 4-hyperthread computer, 400% is possible.

- ***The top command displays load average. Is that the value we are interested in?***
- We are interested in **process CPU utilization**
- Top shows the %CPU utilization for each Linux process for the last update interval (-d .1 is an update interval of 0.1 secs)
- 100% CPU indicates 1 core is 100% busy for last interval

FEEDBACK - 2

- If the computer has 4 virtual cores, then:
  - 100% CPU = 1 thread at full CPU utilization
  - 200% CPU = 2 thread at full CPU utilization
  - 300% CPU = 3 thread at full CPU utilization
  - 400% CPU = 4 thread at full CPU utilization
- CPU utilization of any process will not exceed 400% on 4-core system at any update interval.
- (See Tutorial #2 for a good example of this.)

- ***But, what is load average?***

FEEDBACK - 3

- ***What is load average?***
- Load average is the average number of blocked processes waiting for the CPU over the last 1, 5, and 15 minutes

- High load average and high CPU utilization should correlate, but they are measuring different things

- When a process (e.g. PC Matrix) occupies all CPU cores, then every other Linux process will block and wait
- Other processes (e.g. Google Chrome, etc.) will struggle to get CPU time

FEEDBACK - 4

- For the get() and put() routines in assignment 2, I noticed that the put() routine in prodcons.h is defined as returning an int
- I am confused what we are intended to return with the put() routine.
- ***Should put() return a status, address, or matrix?***
- For "int put()", the "int" ***could be*** used to return a status code, but feel free to redefine this however you'd like as there is ***no specific requirement***

ASSIGNMENT 2 – EXTRA CREDIT

- Please remember to add comments to pcMatrix.c to indicate if extra credit should be graded for Assignment #2:

- EXTRA CREDIT- COMMENTS ARE REQUIRED:
Comments must be included at the top of the mash.c file to indicate which extra credit features (EC1, EC2, EC3, and EC4) have been implemented to receive credit. If there is no indication that extra credit features are implemented, no extra credit will be awarded.

*Example of **required** comment:*

```
// EXTRA CREDIT FEATURES: EC2, EC3 implemented
```

- Helps graders identify if they should evaluate extra credit
- If comments are missing from ***assignment #1***, please go to assignment #1, click: "**Submission Details**" link on the RIGHT
- Add a comment in the "**Add a comment**" box
- Indicate which extra credit (EC1, EC2, EC3, EC4) needs graded

OBJECTIVES – 5/26

- Questions from 5/21
- Tutorial 2 (pthreads, locks, conditions) – due Thurs June 4
- Quiz 3 posted – Active Reading Ch. 19 – due Tues June 2
- Assignment 2 (based on Ch. 30) – due …… *Thurs May 28*
- Assignment 3 – on Linux kernel programming – to be offered in "tutorial" format - posting by ~May 28
- Chapter 20: Paging: Smaller Tables
  - Smaller Tables, Hybrid Tables, Multi-level Page Tables
- Chapter 21/22: Beyond Physical Memory
  - Swapping Mechanisms
  - Swapping Policies

OBJECTIVES – 5/26

- Questions from 5/21
- Tutorial 2 (pthreads, locks, conditions) – due Thurs June 4
- Quiz 3 posted – Active Reading Ch. 19 – due Tues June 2
- Assignment 2 (based on Ch. 30) – due …… *Thurs May 28*
- Assignment 3 – on Linux kernel programming – to be offered in "tutorial" format - posting by ~May 28
- Chapter 20: Paging: Smaller Tables
  - Smaller Tables, Hybrid Tables, Multi-level Page Tables
- Chapter 21/22: Beyond Physical Memory
  - Swapping Mechanisms
  - Swapping Policies

OBJECTIVES – 5/26

- Questions from 5/21
- Tutorial 2 (pthreads, locks, conditions) – due Thurs June 4
- Quiz 3 posted – Active Reading Ch. 19 – due Tues June 2
- Assignment 2 (based on Ch. 30) – due …… *Thurs May 28*
- Assignment 3 – on Linux kernel programming – to be offered in "tutorial" format - posting by ~May 28
- Chapter 20: Paging: Smaller Tables
  - Smaller Tables, Hybrid Tables, Multi-level Page Tables
- Chapter 21/22: Beyond Physical Memory
  - Swapping Mechanisms
  - Swapping Policies

OBJECTIVES – 5/26

- Questions from 5/21
- Tutorial 2 (pthreads, locks, conditions) – due Thurs June 4
- Quiz 3 posted – Active Reading Ch. 19 – due Tues June 2
- Assignment 2 (based on Ch. 30) – due …… *Thurs May 28*
- Assignment 3 – on Linux kernel programming – to be offered in "tutorial" format - posting by ~May 28
- Chapter 20: Paging: Smaller Tables
  - Smaller Tables, Hybrid Tables, Multi-level Page Tables
- Chapter 21/22: Beyond Physical Memory
  - Swapping Mechanisms
  - Swapping Policies

CHAPTER 20: PAGING: SMALLER TABLES

May 26, 2020 | TCSS422: Operating Systems [Spring 2020] School of Engineering and Technology, University of Washington - Tacoma | L16.13

OBJECTIVES – 5/26

- Questions from 5/21
- Tutorial 2 (pthreads, locks, conditions) – due Thurs June 4
- Quiz 3 posted – Active Reading Ch. 19 – due Tues June 2
- Assignment 2 (based on Ch. 30) – due …… *Thurs May 28*
- Assignment 3 – on Linux kernel programming – to be offered in "tutorial" format - posting by ~May 28
- Chapter 20: Paging: Smaller Tables
  - Smaller Tables, Hybrid Tables, Multi-level Page Tables
- Chapter 21/22: Beyond Physical Memory
  - Swapping Mechanisms
  - Swapping Policies

May 26, 2020 | TCSS422: Operating Systems [Spring 2020] School of Engineering and Technology, University of Washington - Tacoma | L16.14

LINEAR PAGE TABLES

- Consider single-level (array-based) page tables:
  - Each process has its own page table
  - Consider a 32-bit process address space (up to 4GB)
  - Indexed using 4 KB (2^{12}) pages
  - Total # of pages = 2^{32} / 2^{12}
  - Total # of pages = 2^{20}
  - Virtual Address has: 20 bits for VPN
  - Virtual Address has: 12 bits for the page offset
  - ***Each Virtual Address requires 32 bits (4 bytes)***

May 26, 2020 | TCSS422: Operating Systems [Spring 2020] School of Engineering and Technology, University of Washington - Tacoma | L16.15

LINEAR PAGE TABLES - 2

- Page tables stored in RAM
- Support potential storage of 2^{20} translations = maps 1,048,576 pages per process @ 4 bytes/entry
- Page table size 4MB / process

Page table size = $\frac{2^{32}}{2^{12}} * 4Byte = 4MByte$

- Consider OS with 100 processes:
  - Requires ~400 MB of RAM to store page tables for virtual to physical address translation

May 26, 2020 | TCSS422: Operating Systems [Spring 2020] School of Engineering and Technology, University of Washington - Tacoma | L16.16

LINEAR PAGE TABLES - 2

- Page tables stored in RAM
- Suppo
  = m
- Pag

Page tables are TOO BIG and consume TOO MUCH memory.

Need Solutions … !!

- Con
  - Requires ~400 MB of RAM to store page tables for virtual to physical address translation

May 26, 2020 | TCSS422: Operating Systems [Spring 2020] School of Engineering and Technology, University of Washington - Tacoma | L16.17

PAGING: USE LARGER PAGES

- Larger pages = 16KB = 2^{14}
- 32-bit address space: 2^{32}
- 2^{18} = 262,144 pages

$\frac{2^{32}}{2^{14}} * 4 = 1MB$ per page table

- Memory requirement cut to ¼
- However pages are huge
- Internal fragmentation results
- 16KB page(s) allocated for small programs with only a few variables

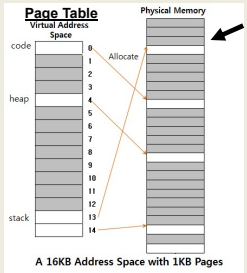
May 26, 2020 | TCSS422: Operating Systems [Spring 2020] School of Engineering and Technology, University of Washington - Tacoma | L16.18

PAGING: USE LARGER PAGES

- Consider a small computer
- 26KB Computer
- Computer has only 26 physical page frames
- 16KB Virtual Address Space for Programs
- System uses 1 KB pages
- Programs consist of: code, heap, stack segments
- Consider a simple program with:
  1 code page, 1 heap page, 2 stack pages

PAGE TABLES: WASTED SPACE

- Process: 16KB Address Space w/ 1KB pages



A 16KB Address Space with 1KB Pages

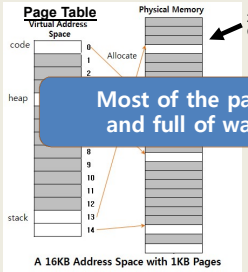
Page table is mostly empty…

PFN	valid	prot	present	dirty
10	1	r-x	1	0
-	0	-	-	-
-	0	-	-	-
-	0	-	-	-
15	1	rw-	1	1
...	...	...	...	...
-	0	-	-	-
3	1	rw-	1	1
23	1	rw-	1	1

A Page Table For 16KB Address Space

PAGE TABLES: WASTED SPACE

- Process: 16KB Address Space w/ 1KB pages



A 16KB Address Space with 1KB Pages

Page table is mostly empty…

Most of the page table is unused and full of wasted space. (73%)

PFN	valid	prot	present	dirty
10	1	r-x	1	0
-	0	-	-	-
-	0	-	-	-
-	0	-	-	-
15	1	rw-	1	1
...	...	...	...	...
-	0	-	-	-
3	1	rw-	1	1
23	1	rw-	1	1

A Page Table For 16KB Address Space

MULTI-LEVEL PAGE TABLES

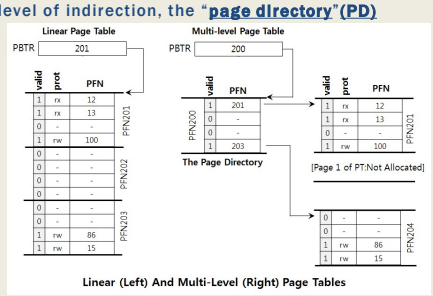
- Consider a single-level page table:
- Virtual address space: 32-bit addressing, 4KB pages
- 2^{20} page table entries
- Even if memory is sparsely populated the *per process* page table requires:

$$\text{Page table size} = \frac{2^{32}}{2^{12}} * 4Byte = 4MByte$$

- Often most of the 4MB *per process* page table is empty
- Page table must be placed in 4MB contiguous block of RAM
- MUST SAVE MEMORY!

MULTI-LEVEL PAGE TABLES - 2

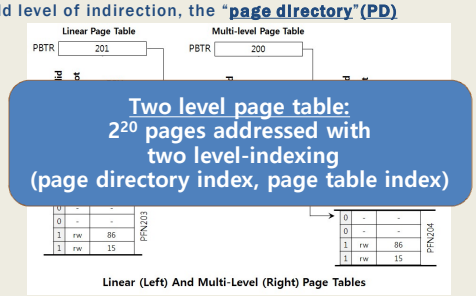
- Add level of indirection, the "**page directory**" **(PD)**



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

MULTI-LEVEL PAGE TABLES - 2

- Add level of indirection, the "**page directory**" **(PD)**



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

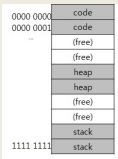
Linear (Left) And Multi-Level (Right) Page Tables

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

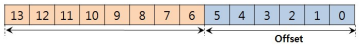
EXAMPLE

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



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

A 16-KB Address Space With 64-byte Pages

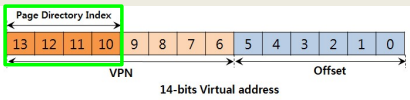


EXAMPLE - 2

- 256 total pages (64 bytes each)
- Assume page table entries are 4 bytes each includes *extra space for status bits*
- Single-level page table: (256 x 4) = 1,024 bytes page table size
- Single-level page table stored using 64-byte pages would span: = (1024/64) = 16 pages
- **Key Idea: the page table is stored using pages too!**

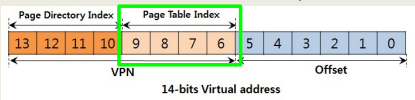
TWO-LEVEL PAGE TABLE: PAGE DIRECTORY INDEX (PDI)

- Now, let's split the page table into two page tables:
  - 1-level page table: 8 bit VPN maps 256 pages
  - **SPLIT VPN in HALF:**
  - PD – 1st level table
  - First half: 4 bits used for the page directory index
  - Indexes the **page directory** (16 ptrs to small page tables)



TWO LEVEL PAGE TABLE: PAGE TABLE INDEX (PTI)

- THE OTHER HALF:
- PT – 2nd level table
- Second half: lower 4 bits used for the page table index
- Indexes a small **page table**
  - Provides lower 4 bits to form 8-bit VPN w/PDI



- Offset bits: 6 bits index any byte on a 64-byte page
- To dereference one 64-byte memory page:
  - We use one page directory entry (PDE)
  - And one page table entry (PTE) – to address a max of 16 pages

EXAMPLE - 3

- Each page directory (PD) holds up to 16 entries (PDEs)
- *These 16 entries can point to a small page table*
- *Or they can be unused*
- **Hello.c**: 1 entry is used, 15 unused
- **Hello.c**: *4 PTEs (stack, code, heap, data segments), 1 PDE*
- **Largest program**: 16 entries (PDEs) point to 16 page tables
- Max memory indexed = 16 x 16 x 64 = 16,384 (16KB)
- 16 page directory entries (PDE) x 16 page table entries (PTE) = 256 maximum PTEs – *fully populated*

EXAMPLE - 4

- **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*)
- **Consider hello.c w/ 4 PTEs: (1 stack, code, data, heap pages)**
- **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 used/12 empty entries x 4 bytes (1 x 64 byte page)
- 128 bytes required (2 x 64 byte pages)
  - Savings = using just 12.5% the space !!!

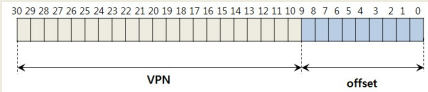
32-BIT EXAMPLE

- Consider: 32-bit address space, 4KB pages, 2^{20} pages
- Only 4 mapped pages **(hello.c)**
- **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 !!!
- How much memory is required to index 100 sparse processes?
  - 100 processes x 8KB now require < 1MB for page tables

TCSS 422 WILL RETURN AT ~2:50PM

MORE THAN TWO LEVELS

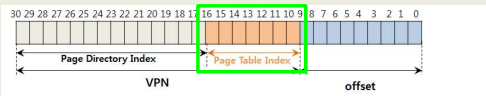
- Consider: Address space of 1 GB
- Page size is 2^9 = 512 bytes
- Page size 512 bytes / Page entry size 4 bytes
- VPN is 21 bits



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

MORE THAN TWO LEVELS - 2

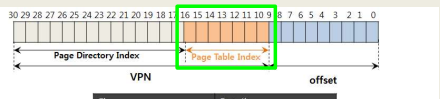
- Page table entries per page = 512 / 4 = 128
- **SPLIT 21 bit VPN:** 7 bits – for page table index (PTI)



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

MORE THAN TWO LEVELS - 3

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



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

MORE THAN TWO LEVELS - 3

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

Can't Store Page Directory with 16K pages, using 512 bytes pages. Pages only fit (dereference) 128 addresses = (512 bytes / 4 bytes)

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

MORE THAN TWO LEVELS - 3

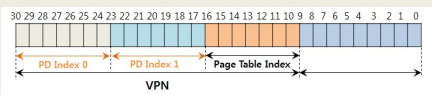
- To map 1 GB address space (2^{30}=1GB RAM, 512-byte pages)
- 2^{14} = 16,384 page directory entries (PDEs) are required
- When using 2^7 (128 entry) page tables…
- Page size = 128 entries x 4 bytes per addr = 512 bytes

Need three level page table:
Page directory 0 (PD Index 0- 7bit)
Page directory 1 (PD Index 1- 7bit)
Small Page Table (Page Table Index- 7bit)

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

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 only 4 entries on a 512-byte page? (e.g. 4 x 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% !!!

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 *pmt;
pte_t *pte;
struct page *page;
```


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

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

(#1) Consider a 16 MB Address Space (2^24) which is indexed using 4KB pages. For a single-level page table, how many pages are required to index memory?

- 2^8 pages
- 2^10 pages
- 2^12 pages
- 2^14 pages
- 2^16 pages

(#2) For this 16 MB Address Space (2^24) indexed using 4KB pages, how many bits are required for the VPN?

- 8 bits
- 16 bits
- 10 bits
- 14 bits
- 12 bits

(#3) Assuming 4 KB pages, how many offset bits are required to index any byte on the page?

- 6 bits
- 10 bits
- 8 bits
- 12 bits
- 14 bits

(#4) Assuming there are 20 status bits, how many bytes are required for each page table entry (PTE)?

- 1 byte
- 2 bytes
- 3 bytes
- 4 bytes
- 0 bytes

(#5) How many kilobytes (KB) are required for a single level page table?

- 32 KB
- 16 KB
- 64 KB
- 8 KB
- 24 KB

(#6) For HelloWorld.c with 4 memory pages: 1 code, stack, heap, data segment, assuming a 2-level page table, how many bits are required for the Page Directory Index (PDI) ?

- 6 bits
- 12 bits
- 10 bits
- 8 bits
- 14 bits

(#7) For HelloWorld.c with 4 memory pages: 1 code, stack, heap, data segment, assuming a 2-level page table, how many bits are required for the Page Table Index (PTI)

14 bits
12 bits
10 bits
8 bits
6 bits

(#8) How much total memory is required to index HelloWorld.c using a two-level page table with just 4 total pages (1 code, stack, heap, data segment page). Hint: need 1 PD and 1 PT

256 bytes
512 bytes
1024 bytes
2048 bytes
4096 bytes

(#9) For a 2-level page table, using a single Page Directory Entry (PDE) pointing to a single Page Table (PT), where all slots of the PT are used, how much memory can be addressed?

16 entries x 4096 bytes = 64 KB
32 entries x 4096 bytes = 128 KB
64 entries x 4096 bytes = 256 KB
256 entries x 4096 bytes = 1024 KB
4096 entries x 4096 bytes = 16384 KB

(#10) For the previous example where one PDE points to a fully used PT, what percentage of memory does the 2-level page table consume vs. a 1-level page table?

256 / 16384
512 / 16384
1024 / 16384
4096 / 16384
100%

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 20 status bits, how many bytes are required for each page table entry?

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 – stack page
  - 1 – heap 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)?

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
  - 20 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

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

ANSWERS

- #1 – 4096 pages
- #2 – 12 bits
- #3 – 12 bits
- #4 – 4 bytes
- #5 – 4096 x 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 → 3.125%

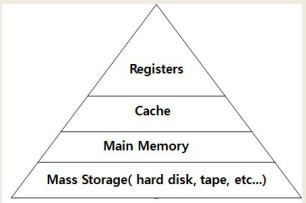
CHAPTER 21/22: BEYOND PHYSICAL MEMORY

OBJECTIVES – 5/26

- **Questions from 5/21**
- **Tutorial 2 (pthreads, locks, conditions) – due Thurs June 4**
- **Quiz 3 posted – Active Reading Ch. 19 – due Tues June 2**
- **Assignment 2 (based on Ch. 30) – due …… *Thurs May 28***
- **Assignment 3 – on Linux kernel programming – to be offered in "tutorial" format - posting by ~May 28**
- **Chapter 20: Paging: Smaller Tables**
  - **Smaller Tables, Hybrid Tables, Multi-level Page Tables**
- **Chapter 21/22: Beyond Physical Memory**
  - **Swapping Mechanisms**
  - **Swapping Policies**

MEMORY HIERARCHY

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



Memory Hierarchy in modern system

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

LATENCY TIMES

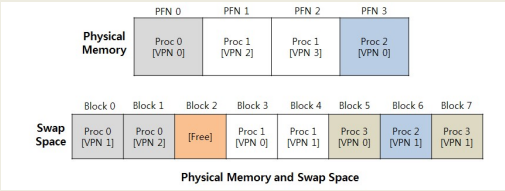
- Design considerations
  - SSDs 4x the time of DRAM
  - HDDs 80x the time of DRAM

Action	Latency (ns)	(µs)	
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 µs	~1GB/sec SSD
Read 1 MB sequentially from memory	250,000 ns	250 µs	
Read 1 MB sequentially from SSD*	1,000,000 ns	1,000 µs	1 ms ~1GB/sec SSD, 4X memory
Read 1 MB sequentially from disk	20,000,000 ns	20,000 µs	20 ms 80x memory, 20X SSD

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

SWAP SPACE

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



Physical Memory and Swap Space

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

PAGE FAULT

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

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


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

REPLACEMENT POLICIES

CACHE MANAGEMENT

- Replacement policies apply to "any" cache
- Goal is to minimize the number of misses
- Average memory access time (AMAT) 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 ns, T_D = 10ms
- Consider P_{hit} = .9 (90%), P_{miss} = .1
- Consider P_{hit} = .999 (99.9%), P_{miss} = .001

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

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:

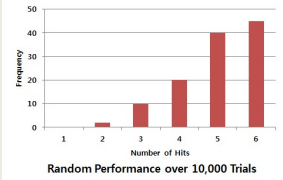
0 1 2 0 1 3 0 3 1 2 1

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

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



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 the fewest # of accesses (front)
- Consider frequency of page accesses

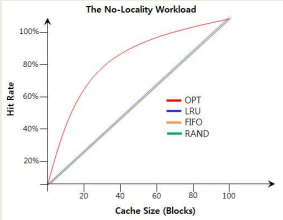
0 1 2 0 1 3 0 3 1 2 1

Hit/miss ratio is=

6 hits

WORKLOAD EXAMPLES: NO-LOCALITY

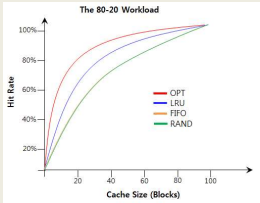
- No-Locality (Random Access) Workload
  - Perform 10,000 random page accesses
  - Across set of 100 memory pages



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

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)

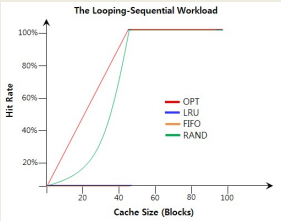


LRU is more likely to hold onto hot pages

(recalls history)

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"

IMPLEMENTING LRU

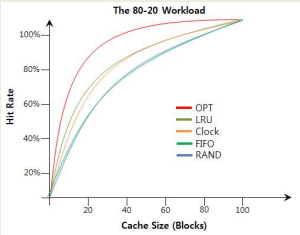
- Implementing last recently used (LRU) requires tracking access time for all system memory pages
- Times can be tracked with a list
- For cache eviction, we must scan an entire list
- Consider: 4GB memory system (2^{32}), with 4KB pages (2^{12})
- This requires 2^{20} comparisons !!!
- Simplification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

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

CLOCK ALGORITHM

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



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

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

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

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

QUESTIONS

WILL RETURN IN A FEW MINUTES