

Beyond Physical Memory

CS 537: Introduction to Operating Systems

Louis Oliphant

University of Wisconsin - Madison

Fall 2024

Administrivia

- Project 3 out – Due Oct 8th @ 11:59pm
- Midterm 1 Scheduled:
 - Regular Time: Oct 15th, 5:45-7:15, Van Vleck B102, Ingraham B10
 - Unable to attend? Fill out [this form](#)
 - Alternate Time: Oct 16th, 5:45-7:15, Psych Bldg 121
 - McBurney Time: Oct 15th, 5:45-8:45pm, CS 1257
 - Bring **#2 Pencil** and **UW Student ID**

Review: Smaller Tables

There are multiple ways to shrink the size of a page table:

- Increase the size of a page (and decrease the number of pages)
- Use a combination of segmentation and paging, keeping a separate page table for each segment and each table can grow independently (using base and bounds to represent location of table and # pages of table)
- Use a multi-level page table where the VPN can be divided into the page directory index and the page table index. First index into the page directory to find the proper page of the page table then index into that page to find the PTE

Review: Multi-Level Page Table

Page Size: 32 bytes (5 bits)

Virtual Address Space: 1024 pages (10 bits)

Physical Memory: 128 pages (7 bits)

The system uses a multi-level page table where the upper five bits of the VA is the index into the page directory; the PDE, if valid, points to a page of the page table. Each page table page holds 32 page-table entries (PTEs). Each PTE, if valid, holds the desired translation (PFN) of the virtual page.

The format of a PTE is thus:

```
VALID | PFN6 ... PFN0
```

and is thus 8 bits or 1 byte.

The format of a PDE is essentially identical:

```
VALID | PT6 ... PT0
```

You are given the PDBR and a dump of the pages of memory.

You must convert a virtual address to a physical address.

Virtual Address 0x4920: Translates To What Physical Address? What value would be fetched from that address?

Review: Multi-Level Page Table (cont.)

PDBR: 13 (decimal) [This means the page directory is held in this page]

[illegible]

• • •

Review: Multi-Level Page Table Solution

Virtual Address: 0x4920 (binary: 0100 1001 0010 0000)

PD index: 10010 PT index: 01001 offset: 00000

Look up at PDBR (page 13) + index of 18 on page = 0x97 (binary: 1001 0111)

top-bit: valid, remainder: 0010111 (decimal 23)

Look up at page 23 + index of 9 on page = 0x82 (PTE, 1000 0010)

top-bit: valid, remainder is PFN: page 2

Physical address: PFN combined with offset = 0000 0100 0000 (0x040)

Value fetched: 0x15

Quiz 8: Multi-Level Page Tables

<http://tinyurl.com/cs537-fa24-q8>



Agenda

Memory Virtualization

- How to support virtual memory larger than physical memory?
- What are the **Mechanisms** and **Policies** for this?

Motivation

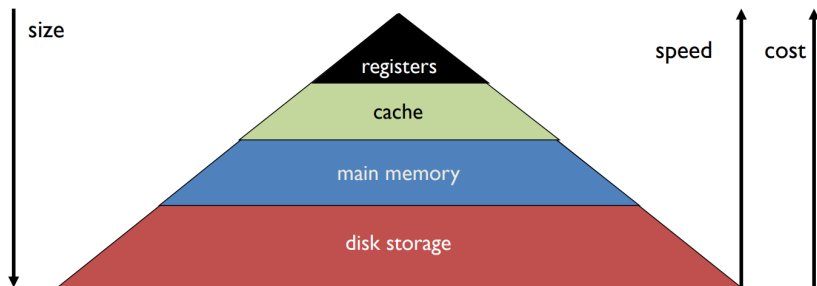
- OS Goal: Support processes when not enough physical memory
 - Single process with very large address space
- User code should be independent of amount of physical memory
 - Correctness, if not performance
- Virtual Memory: OS provides illusion of more physical memory

Why This Works

- Leverage **locality of references** within a process:
 - **Spatial** – reference memory address near previously referenced addresses
 - **Temporal** – reference memory address that have referenced recently in the past
 - Process spend majority of time in small portion of code
 - Estimate: 90% of time in 10% of code
- Implication:
 - Process only uses small amount of address space **at any moment**
 - Only small amount of address space must be resident in physical memory

Memory Hierarchy

Leverage **memory hierarchy** of machine architecture
Each layer acts as “backing store” for layer above



Swapping Intuition

Idea: OS keeps unreferenced pages on disk

- Slower, cheaper “backing store” than memory

Process can run when not all pages are loaded into memory

OS and hardware cooperate to make large disk seem like memory

- Same behavior as if all of address space in memory

Requirements:

- OS must have **mechanism** to identify location of each page in address space – either in memory or on disk
- OS must have **policy** to determine which pages live in memory and which on disk

Virtual Address Space Mechanisms

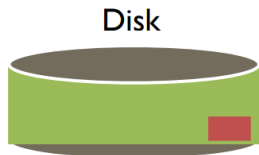
Each page in Virtual Address space maps to one of three locations:

- Physical Memory: Small, fast, expensive
- Disk (backing store): Large, slow, cheap
- Nothing (error): Free

Extend page table entries with an extra bit: **present**

- permissions (r/w), valid, **present**
- page in memory: present bit set in PTE
- page on disk: present bit cleared
 - PTE points to block on disk
 - Causes trap into OS when page is referenced
 - **Trap: page fault**

Example Page Table (with Present Bit)



Phys Memory



PFN	valid	prot	present
10	1	r-x	1
-	0	-	-
23	1	rw-	0
-	0	-	-
-	0	-	-
-	0	-	-
-	0	-	-
-	0	-	-
-	0	-	-
-	0	-	-
28	1	rw-	0
4	1	rw-	1

What if access vpn 0xb?

Virtual Memory Mechanisms

First, hardware checks TLB for virtual address

- If TLB hit, address translation done; page in physical memory

Else

- Hardware or OS checks page table in memory
- If PTE designates present, page in physical memory

Else

- Trap into OS:
 - OS selects victim page in memory to replace
 - Writes victim page out to disk if modified (dirty bit is set)
 - OS reads referenced page from disk into memory
 - Page Table is updated, present bit is set
 - Process continues execution

Swapping Policies

Goal: minimize number of page faults - Page faults require milliseconds to handle (reading from disk) - The cost of disk access is so high, even a tiny page fault rate will dominate memory access time

Optimal Policy: Replace the page that will be accessed *furthest in the future* (requires knowing the future!).

Assume a program accesses the following stream of virtual pages: 0, 1, 2, 0, 1, 3, 0, 3, 1, 2, 1 And Physical memory can hold 3 pages – Trace the optimal policy

Optimal Policy Trace

Access	Hit/Miss?	Evict	Resulting Cache State
0	Miss		0
1	Miss		0, 1
2	Miss		0, 1, 2
0	Hit		0, 1, 2
1	Hit		0, 1, 2
3	Miss	2	0, 1, 3
0	Hit		0, 1, 3
3	Hit		0, 1, 3
1	Hit		0, 1, 3
2	Miss	3	0, 1, 2
1	Hit		0, 1, 2

Page Accesses:

0, 1, 2, 0, 1, 3,
0, 3, 1, 2, 1

Hit Rate: $6 / 11$
 $= 54.5\%$

Figure 22.1: Tracing The Optimal Policy

Simple Policy: FIFO

Access	Hit/Miss?	Evict	Resulting Cache State
0	Miss		First-in→ 0
1	Miss		First-in→ 0, 1
2	Miss		First-in→ 0, 1, 2
0	Hit		First-in→ 0, 1, 2
1	Hit		First-in→ 0, 1, 2
3	Miss	0	First-in→ 1, 2, 3
0	Miss	1	First-in→ 2, 3, 0
3	Hit		First-in→ 2, 3, 0
1	Miss	2	First-in→ 3, 0, 1
2	Miss	3	First-in→ 0, 1, 2
1	Hit		First-in→ 0, 1, 2

Page Accesses:

0, 1, 2, 0, 1, 3,
0, 3, 1, 2, 1

Hit Rate: $4 / 11$
 $= 36.4\%$

Figure 22.2: Tracing The FIFO Policy

Simple Policy: Random

Access	Hit/Miss?	Evict	Resulting Cache State
0	Miss		0
1	Miss		0, 1
2	Miss		0, 1, 2
0	Hit		0, 1, 2
1	Hit		0, 1, 2
3	Miss	0	1, 2, 3
0	Miss	1	2, 3, 0
3	Hit		2, 3, 0
1	Miss	3	2, 0, 1
2	Hit		2, 0, 1
1	Hit		2, 0, 1

Page Accesses:

0, 1, 2, 0, 1, 3,
0, 3, 1, 2, 1

Hit Rate: $5 / 11$
 $= 45.5\%$

How random
does depends on
the luck of the
draw

Figure 22.3: Tracing The Random Policy

Policy : LRU (use past to predict future)

Access	Hit/Miss?	Evict	Resulting Cache State
0	Miss		LRU→ 0
1	Miss		LRU→ 0, 1
2	Miss		LRU→ 0, 1, 2
0	Hit		LRU→ 1, 2, 0
1	Hit		LRU→ 2, 0, 1
3	Miss	2	LRU→ 0, 1, 3
0	Hit		LRU→ 1, 3, 0
3	Hit		LRU→ 1, 0, 3
1	Hit		LRU→ 0, 3, 1
2	Miss	0	LRU→ 3, 1, 2
1	Hit		LRU→ 3, 2, 1

Page Accesses:

0, 1, 2, 0, 1, 3,
0, 3, 1, 2, 1

Hit Rate: $6 / 11$
 $= 54.5\%$

Figure 22.5: Tracing The LRU Policy

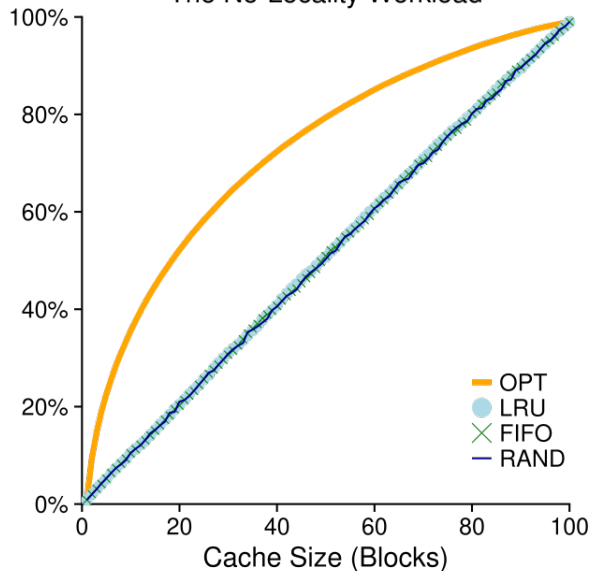
More Realistic Workloads

The example trace was for one specific workload. What about more realistic workloads:

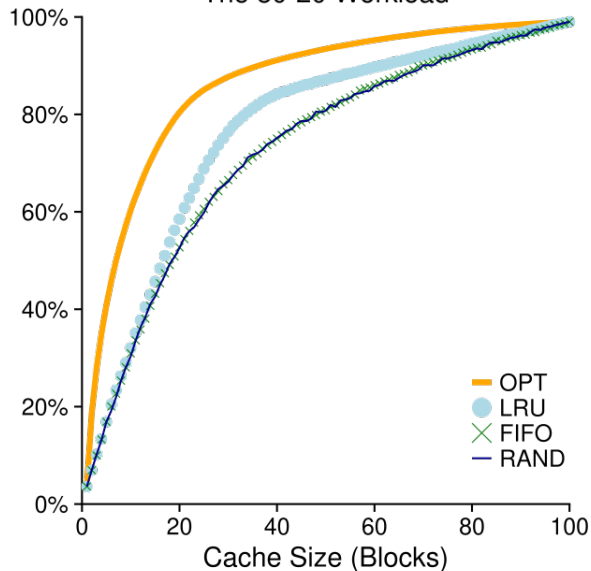
Imagine a virtual address space with 100 pages

- Randomly choose a page 10,000 times (**No-Locality**)
- 80% of accesses are to 20% of pages (hot pages), rest of accesses are to remaining pages (cold pages) (**80-20**)
- Refer to 50 pages in sequence then loop for 10,000 accesses (**looping**)

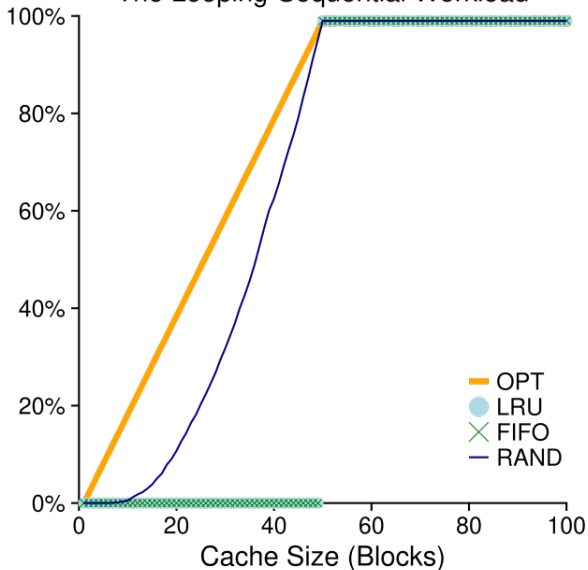
The No-Locality Workload



The 80-20 Workload



The Looping-Sequential Workload



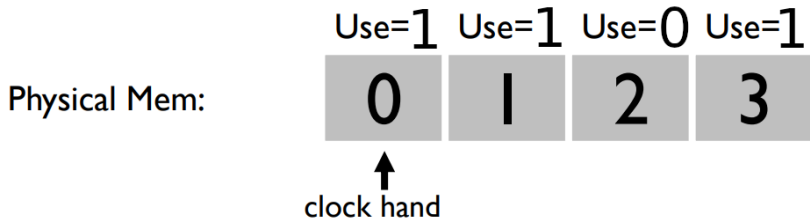
Approximating LRU

LRU is costly to run, requires every memory access to mark time page was accessed then on replacement, scan all pages to find the LRU page.

Clock algorithm: approximates LRU:

- Add **use bit** to PTE, whenever page is referenced, bit set to 1
- Imagine all the pages of the system arranged in a circular list
- A clock hand points to some particular page, P
- When replacement needs to happen, OS checks use bit of page P
 - if 1, (not good candidate) set use bit to 0 and advance P, keep looking
 - if 0, (good candidate) replace this page

Clock Algorithm Example



Clock Extensions

- Replace multiple pages at once
 - intuition: expensive to run replacement algorithm and write a single page to disk
 - find multiple victims each time
- Use dirty bit to give preference to dirty pages
 - intuition: more expensive to replace dirty pages
 - dirty pages must be written to disk, clean pages do not
 - replace pages that have use bit AND dirty bit cleared

Summary: Virtual Memory

- Abstraction: Virtual Address Space with code, heap, stack
- Address Translation
 - Segmentation with base/bounds
 - Paging with page tables
 - Paging Challenges:
 - extra memory references: avoid with TLB
 - page table size: avoid with multi-level paging, hybrid approach
- Large Address Spaces: Swapping mechanisms and policies (LRU, Clock)