Beyond Physical Memory

CS 537: Introduction to Operating Systems

Louis Oliphant

University of Wisconsin - Madison

Fall 2023

Administrivia

- Project 3 out – Due Oct 10 @ 11:59pm
  - The GNU C Library Manual has loads of great stuff, including a section on implementing a job control shell
- **Midterm 1**: Oct 12 in-class
  - Bring a **#2 Pencil** and your **UW ID**
  - Review material will be posted over the weekend
- Cancel office hours for Louis Oliphant today

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 table to find the proper page of the page table then index into that page to find the PTE
- Use an inverted page table (only need one to share across all processes)

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]

page   0:0000000000000000000000000000000000000000000000000000000000000000
page   1:7f7f7f7f7f7f7fe57f7f7f7f7f7f7f7f7f7f7f7f7f7fde7f7f7f7f7f7f7fb77f
page   2:151d0d0a111d080905130e070c01091e12081d0b07010406071b0807121c0917
page   3:7f7f7f7f7f7f7f7fed7f7f7f7f7f7f7f8e7f7f7f7f7f7f7f7f7f7f7f7f7f7f7f
page   4:7f7f7f7f7f7f7f7f7f7f7f7f7f877f7f7f7f7f7f7f7f7f7f7f7f7f7fe17f7f7f
page   5:1c010a0f061b03021e00060c1b0a111813190010001a00020d130013030a0116
page   6:0000000000000000000000000000000000000000000000000000000000000000
page   7:0d0104011e0e08040803181c1902121a0c180010170d031e190816051316120d
page   8:7f7f7f7f7f9f7f7f7f7f7f7f7f7f7f7f7f7f7f7f7f7f7f7f7f7f7f7f7f7f7f7f
page   9:0000000000000000000000000000000000000000000000000000000000000000
page  10:0000000000000000000000000000000000000000000000000000000000000000
page  11:0e111413081114091a041e1d1e000c0216121616001a1d13081d101b131e1007
page  12:0d040a0e080a0e1606050e090704191803140d02021e0310151715020b031618
page  13:8384fe9588a57f9bc1cfebccd0e87fa79ef3977ffda3f8d5ecc3a97f7f909981
page  14:07091c0408110e0d0004091a1318041e190d1d0e0a160415051c131a1b141206
page  15:00021b1307090f161c04061e08020f0c100907171d0f05141a1d0f1714001002
page  16:7ffcfa7f7f7f7f7f7f7f7f7f7f7f7f7f7f7f7f7f7f7f7f7f7f7fbd7f7f7f7f
page  17:130a18141d06021b13080903130c0810140e0b1b131716011a0710141e171206
page  18:0614140a1c1411010c080e1c1a01151c10021a0d1e1b191c021809040b12000d
page  19:0000000000000000000000000000000000000000000000000000000000000000
page  20:071500160519121b1e19131a0d0b0f190a100d001404160217000304150f0618
page  21:7f7f7f7f7f7f7f7f7f7f7f7f7f7f7f7f7f7f7f7f7f7f7fac7f7f7f7f7f7f7f7f
page  22:0000000000000000000000000000000000000000000000000000000000000000
page  23:7f7f7fc07f7f7f7f7f827f7f7f7f7f7f7f7f7f7f7f7f7f7f7f7f7fd17f7f7f7f
page  24:0000000000000000000000000000000000000000000000000000000000000000
...
```

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

https://tinyurl.com/cs537-fa23-q8

Agenda

Memory Virtualization

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

Motivation

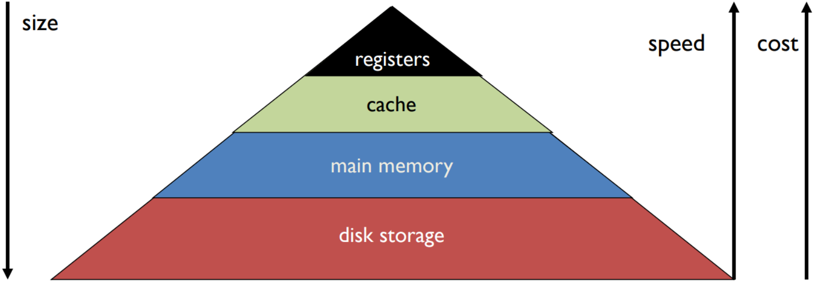
- OS Goal: Support processes when not enough physical memory
  - Single process with very large address space
- User code should be independent of amount of physical memory
  - Correctness, if not performance
- Virtual Memory: OS provides illusion of more physical memory

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)

Disk

Phys Memory

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

Figure 22.1: **Tracing The Optimal Policy**

Page Accesses: 0, 1, 2, 0, 1, 3, 0, 3, 1, 2, 1

Hit Rate: 6 / 11 = 54.5%

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

Figure 22.2: **Tracing The FIFO Policy**

Page Accesses:
0, 1, 2, 0, 1, 3,
0, 3, 1, 2, 1

Hit Rate: 4 / 11
= 36.4%

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

Figure 22.3: **Tracing The Random Policy**

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

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

Figure 22.5: **Tracing The LRU Policy**

Page Accesses:
0, 1, 2, 0, 1, 3, 0, 3, 1, 2, 1

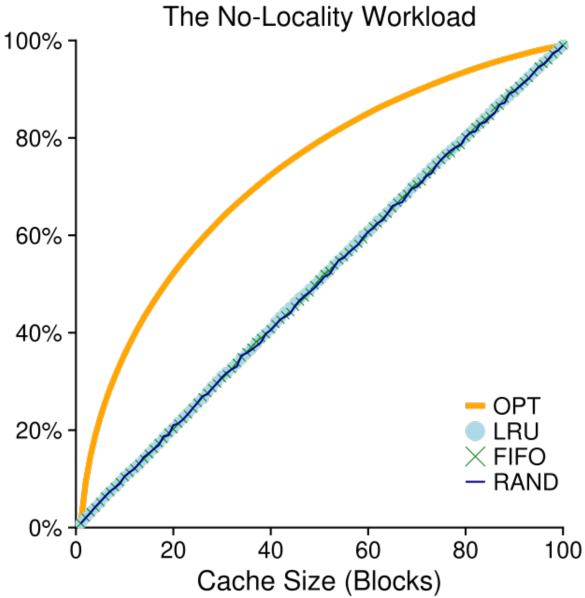
Hit Rate: 6 / 11 = 54.5%

More Realistic Workloads

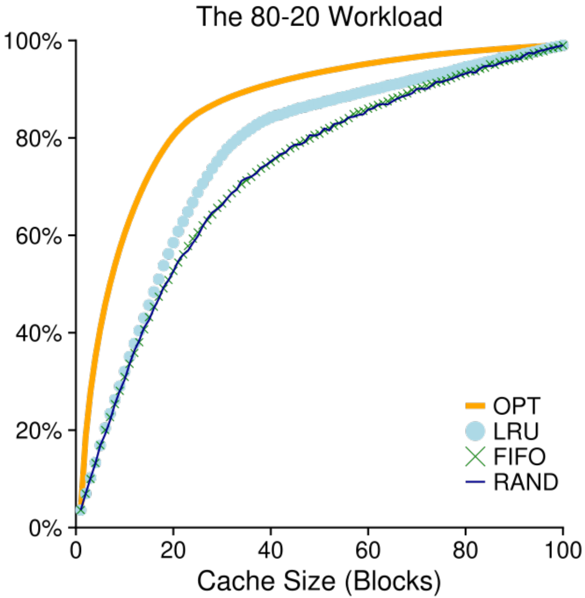
The example trace was for one specific workload. What about more realistic workloads:

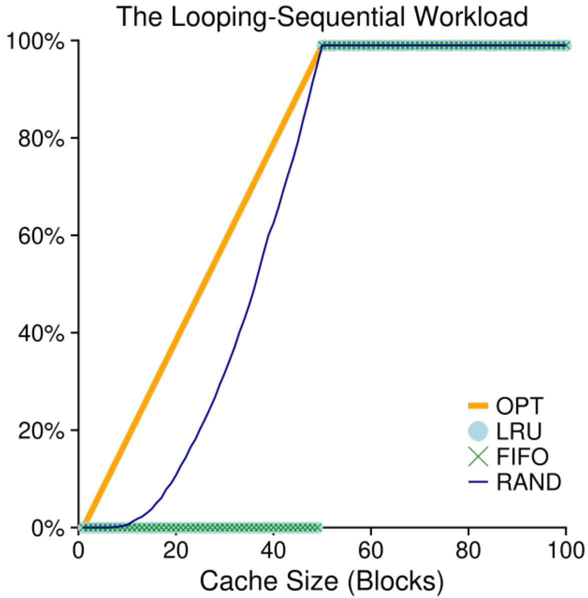
Imagine a virtual address space with 100 pages

- 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
100%
80%
60%
40%
20%
0%
0
20
40
60
80
100
Cache Size (Blocks)
OPT
LRU
FIFO
RAND


The 80-20 Workload
100%
80%
60%
40%
20%
0%
0
20
40
60
80
100
Cache Size (Blocks)
OPT
LRU
FIFO
RAND


The Looping-Sequential Workload
100%
80%
60%
40%
20%
0%
0
20
40
60
80
100
Cache Size (Blocks)
OPT
LRU
FIFO
RAND

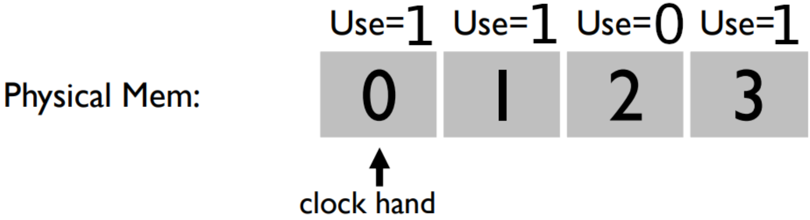
Approximating LRU

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 victimes 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)