Virtual Memory & Caching

(Chapter 12-17)

CS 4410
Operating Systems
Last Time: Address Translation

- Paged Translation
- Efficient Address Translation
  - Multi-Level Page Tables
  - Inverted Page Tables
  - TLBs

This time: **Virtual Memory & Caching**
• Virtual Memory
• Caching
What is Virtual Memory?

- Each process has an illusion of a large address space
- \(2^x\) bytes for \(x\)-bit addressing
- However, physical memory is usually much smaller
- How do we give this illusion to multiple processes?
- Virtual Memory: some addresses reside in disk

![Diagram of Virtual Memory](image-url)
Process executes from disk!

RAM is really just another layer of cache
Swapping vs. Paging

**Swapping**
- Loads entire process in memory
- “Swap in” (from disk) or “Swap out” (to disk) a process
- Slow (for large processes)
- Wasteful (might not require everything)
- Does not support sharing of code segments
- Virtual memory limited by size of physical memory

**Paging**
- Runs all processes concurrently
- A few pages from each process live in memory
- Finer granularity, higher performance
- Large virtual mem supported by small physical mem
- Certain pages (read-only ones, for example) can be shared among processes
(the contents of) **A Virtual Page Can Be**

**Mapped**
- to a physical frame

**Not Mapped** \(\rightarrow\) **Page Fault**
- in a physical frame, but not currently mapped
- or still in the original program file
- or zero-filled (heap/BSS, stack)
- or on backing store ("paged or swapped out")
- or illegal: not part of a segment
  \(\rightarrow\) Segmentation Fault
Supporting Virtual Memory

Modify Page Tables with a valid bit (= “present bit”)

- Page in memory $\rightarrow$ valid $= 1$
- Page not in memory $\rightarrow$ PT lookup triggers page fault
Handling a Page Fault

Identify page and reason (r/w/x)

- access inconsistent w/ segment access rights → terminate process
- access a page that is kept on disk:
  → does frame with the code/data already exist? No? Allocate a frame & bring page in (next slide)
- access of zero-initialized data (BSS) or stack
  - Allocate a frame, fill page with zero bytes
- access of COW page
  - Allocate a frame and copy
When a page needs to be brought in...

- Find a free frame
  - evict one if there are no free frames
- Issue disk request to fetch data for page
- Block current process
- Context switch to new process
- When disk completes, update PTE
  - frame number, valid bit, RWX bits
- Put current process in ready queue
When a page is swapped out...

- Find all page table entries that refer to old page
  - Frame might be shared
  - Core Map (frames → pages)
- Set each page table entry to invalid
- Remove any TLB entries
  - “TLB Shootdown”
- Write changes on page back to disk, if needed
  - Dirty/Modified bit in PTE indicates need
  - Text segments are (still) on program image on disk
Demand Paging, MIPS style

1. TLB miss
2. Trap to kernel
3. Page table walk
4. Find page is invalid
5. Convert virtual address to disk block number
6. Allocate frame
   • evict if needed
7. Initiate disk block read into frame
8. Disk interrupt when DMA complete
9. Mark page valid
10. Update TLB
11. Resume process at faulting instruction
12. Execute instruction

Software handling page fault between arrows
1. TLB miss
2. Page table walk
3. Page fault (find page is invalid)
4. Trap to kernel
5. Convert virtual address to disk block number
6. Allocate frame
   • evict if needed
7. Initiate disk block read into frame
8. Disk interrupt when
9. DMA complete
10. Mark page valid
11. Resume process at faulting instruction
12. TLB miss
13. Page table walk to fetch translation
14. Execute instruction
Updated Context Switch

• Save current process’ registers in PCB
• Also Page Table Base Register (PTBR)
• **Flush TLB** *(unless TLB is tagged)*
• Restore registers and PTBR of next process to run
• “Return from Interrupt”
OS Support for Paging

**Process Creation**
- Allocate frames, create & initialize page table & PCB

**Process Execution**
- Reset MMU (PTBR) for new process
- Context switch: flush TLB (or TLB has pids)
- Handle page faults

**Process Termination**
- Release pages
• Virtual Memory
• Caching
What are some examples of caching?

- TLBs
- hardware caches
- internet naming
- web content
- incremental compilation
- just in time translation
- virtual memory
- file systems
- branch prediction
Memory Hierarchy

<table>
<thead>
<tr>
<th>Cache</th>
<th>Hit Cost</th>
<th>Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>1st level cache / 1st level TLB</td>
<td>1 ns</td>
<td>64 KB</td>
</tr>
<tr>
<td>2nd level cache / 2nd level TLB</td>
<td>4 ns</td>
<td>256 KB</td>
</tr>
<tr>
<td>3rd level cache</td>
<td>12 ns</td>
<td>2 MB</td>
</tr>
<tr>
<td>Memory (DRAM)</td>
<td>100 ns</td>
<td>10 GB</td>
</tr>
<tr>
<td>Data center memory (DRAM)</td>
<td>100 µs</td>
<td>100 TB</td>
</tr>
<tr>
<td>Local non-volatile memory</td>
<td>100 µs</td>
<td>100 GB</td>
</tr>
<tr>
<td>Local disk</td>
<td>10 ms</td>
<td>1 TB</td>
</tr>
<tr>
<td>Data center disk</td>
<td>10 ms</td>
<td>100 PB</td>
</tr>
<tr>
<td>Remote data center disk</td>
<td>200 ms</td>
<td>1 XB</td>
</tr>
</tbody>
</table>

Every layer is a cache for the layer below it.
Working Set

1. Collection of a process’ most recently used pages
   (The Working Set Model for Program Behavior, Denning,’68)

2. Pages referenced by process in last $\Delta$ time-units
Thrashing

Excessive rate of paging
Cache lines evicted before they can be reused

Causes:
- Too many processes in the system
- Cache not big enough to fit working set
- Bad luck (conflicts)
- Bad eviction policies (later)

Prevention:
- Restructure code to reduce working set
- Increase cache size
- Improve caching policies
“Thrash” dates from the 1960’s, when disk drives were as large as washing machines. If a program’s working set did not fit in memory, the system would need to shuffle memory pages back and forth to disk. This burst of activity would violently shake the disk drive.

The first hard disk drive—the IBM Model 350 Disk File (came w/IBM 305 RAMAC, 1956).

Total storage = 5 million characters (just under 5 MB).

Caching

• *Assignment*: where do you put the data?
• *Replacement*: who do you kick out?
Address Translation Problem

- Adding a layer of indirection disrupts the spatial locality of caching
- CPU cache is usually physically indexed
- Adjacent pages may end up sharing the same CPU cache lines

➤ **BIG PROBLEM:**

cache effectively smaller
Solution: Cache Coloring (Page Coloring)

1. Color frames according to cache configuration.

2. Spread each process’ pages across as many colors as possible.
Cache Coloring Example
Caching

• Assignment: where do you put the data?
• Replacement: who do you kick out?

What do you do when memory is full?
Page Replacement Algorithms

- **Random**: Pick any page to eject at random
  - Used mainly for comparison
- **FIFO**: The page brought in earliest is evicted
  - Ignores usage
- **OPT**: Belady’s algorithm
  - Select page not used for longest time
- **LRU**: Evict page that hasn’t been used for the longest
  - Assumes past is a good predictor of the future
- **MRU**: Evict the most recently used page
- **LFU**: Evict least frequently used page
- And many approximation algorithms
First-In-First-Out (FIFO) Algorithm

- **Reference string:** 1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5
- **3 frames** (3 pages in memory at a time per process):

<table>
<thead>
<tr>
<th>frames</th>
<th>reference</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>4</td>
<td>3</td>
</tr>
<tr>
<td>4</td>
<td>3</td>
</tr>
</tbody>
</table>

← contents of frames at time of reference

- 9 page faults

- page fault
- hit
- 4 marks arrival time
First-In-First-Out (FIFO) Algorithm

- **Reference string:** 1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5
- **4 frames** (4 pages in memory at a time per process):

<table>
<thead>
<tr>
<th>frames</th>
<th>reference</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>4</td>
<td>3</td>
</tr>
<tr>
<td>4</td>
<td>3</td>
</tr>
<tr>
<td>4</td>
<td>3</td>
</tr>
<tr>
<td>4</td>
<td>3</td>
</tr>
<tr>
<td>4</td>
<td>3</td>
</tr>
<tr>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>2</td>
</tr>
</tbody>
</table>

  ➔ contents of frames at time of reference

  page fault
  hit
  4 marks arrival time

  10 page faults

  more frames ➔ more page faults?

  Belady’s Anomaly
Optimal Algorithm (OPT)

- Replace frame that will not be used for the longest
- 4 frames example

6 page faults

Question: How do we tell the future?
Answer: We can’t

OPT used as upper-bound in measuring how well your algorithm performs
OPT Approximation

In real life, we do not have access to the future page request stream of a program

→ Need to make a guess at which pages will not be used for the longest time
Least Recently Used (LRU) Algorithm

Reference string: 1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5

8 page faults
Implementing LRU

- On reference: Timestamp each page
- On eviction: Scan for oldest page

Problems:
- Large page lists
- Timestamps are costly

Solution: approximate LRU
- Note: LRU is already an approximation
- Exploit use (REF) bit in PTE
Not Recently Used

- Periodically (say, each clock tick), clear all *use* (aka REF) bits in PTEs
  - Ideally done in hardware
- When evicting a frame, scan for a frame that hasn’t recently been referenced
  - *use* bit is clear in PTE
  - may require a scan of all frames, so keep track of last evicted frame
- If no such frame exists, select any
Working Set Algorithm (WS)

• Maintain for each frame the approximate time the frame was last used
• At each clock tick
  • Update this time to the current time for all frames that were referenced since the last clock tick
    – i.e., the ones with use (REF) bits set
  • Clear all use bits
• Put all frames that have not been used for some time $\Delta$ (working set parameter) on the free list
• When a frame is needed, use free list
  • If empty, pick any frame

*Note: requires scan of all frames at each clock tick*
Using virtual time or real time?

• Often, it is argued it would be more fair to use “virtual time” (the time that a process has the CPU) instead of “real time” in the WS algorithm and variants or processes that do not use the CPU much would be more likely to be paged out

• However, maybe processes that do not use the CPU also tend to use fewer pages per time unit

• And waiting processes do not age at all in virtual time, and do not use any pages
Clock Algorithm

- To allocate a frame, inspect the use bit in the PTE at clock hand and advance clock hand.
- Used? Clear use bit and repeat.
Two-Handed Clock

One-handed clock: What if memory is very large?

Use two hands! (at fixed angle)

Leading hand clears *use* bit
- slowly clears history
- finds victim candidates

Trailing hand evicts frames with *use* bit set to 0

Big angle? Small angle?

---

Evicts 1st use=0 frame it finds

Blue 1’s were referenced after use bit was cleared by green hand
WSCLOCK

• Merge WS and CLOCK algorithms
• Maintain $timestamp$ for each frame
• When allocating a frame:
  • Inspect $use$ bit of frame under hand
  • If set
    – Clear the $use$ bit
    – Update the timestamp
    – Continue with next frame
  • If clear but now – $timestamp$ < $\Delta$
    – Continue with next frame (do not update timestamp)
• Otherwise evict frame

Note: can go into an infinite loop...
Stack Algorithms

• Let $M(m, r)$ be the set the of virtual pages in physical memory given that there are $m$ frames at reference string $r$

• A page replacement algorithm is called a “stack algorithm” if for all #frames $m$ and all reference strings $r$:

  $M(m, r)$ is a subset of $M(m + 1, r)$

i.e., a stack algorithm does not suffer from Belady’s anomaly:

  more frames $\Rightarrow$ not more misses
FIFO algorithm, m = 3 vs m = 4

Stack algorithm “subset property” violated
Theorem: LRU and MRU are stack algorithms

- By definition:
  - For LRU: $M(m + 1, r)$ contains the $m$ most frequently used frames, so $M(m, r)$ is a subset of $M(m + 1, r)$
  - Similar for MRU
  - Similar also for LFU (Least Frequently Used)
Theorem: OPT is a stack algorithm

• Proof non-trivial. See paper that introduced the concept of stack algorithms:
Local versus Global Replacement

• So far we have tacitly assumed that all frames are shared by all processes
  • This is called “global replacement”
• But is it fair?
  • Badly behaved processes can ruin the experience of processes with good locality
• Local replacement: divided the frames up evenly between the processes
  • Can lead to under-utilization