Last Time: Address Translation

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

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

- Each process has illusion of 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
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
A Virtual Page Can Be

**Mapped**
- to a physical frame

**Not Mapped → 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
  → Segmentation Fault
Supporting Virtual Memory

Modify Page Tables with a *present* bit

- Page in memory $\rightarrow$ *present* = 1
- Page not in memory $\rightarrow$ PT lookup triggers **page fault**

<table>
<thead>
<tr>
<th>Page Address</th>
<th><em>Present</em></th>
</tr>
</thead>
<tbody>
<tr>
<td>32</td>
<td>P=1</td>
</tr>
<tr>
<td>4183</td>
<td>P=0</td>
</tr>
<tr>
<td>177</td>
<td>P=1</td>
</tr>
<tr>
<td>5721</td>
<td>P=0</td>
</tr>
</tbody>
</table>

Page Table

Disk

Mem
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 C.O.W. (copy on write) 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 another process
• When disk request completes, update PTE
  • frame number, present bit, RWX bits
• Put current process in ready queue
When a page needs to be swapped out...

- Find all page table entries that refer to old page
  - Frame might be shared
  - Maintain a **Core Map** (frames \( \rightarrow \) 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. If page is present, goto 10
5. Convert virtual address to disk block number
6. Allocate frame
   • evict one if needed
7. Initiate disk block read into frame
8. Disk interrupt when DMA complete
9. Mark page present
10. Update TLB
11. Resume process at faulting instruction
12. Execute instruction

Software handling page fault between arrows
Demand Paging, x86 style

1. TLB miss
2. Page table walk
3. If page is present/valid, update TLB and re-execute instruction
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 DMA complete
9. Mark page present
10. Resume process at faulting instruction
11. TLB miss
12. Page table walk to fetch translation
13. Execute instruction
   Software handling page fault between arrows
Updated Context Switch

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

**Process Creation**
- Create & initialize page table & PCB

**Process Execution**
- Reset MMU (PTBR) for new process
- Context switch: flush TLB
  - unless TLB is tagged with PIDs
- Handle page faults

**Process Termination**
- Release pages

**Page Daemon**
- Clean dirty pages proactively
• Virtual Memory
• Caching
What are some examples of caching?

- TLBs
- virtual memory
- hardware caches
- internet naming
- web content
- incremental compilation
- just in time translation
- file systems
- branch prediction
Every layer is a cache for the layer below it.

<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>
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.
Caching

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

• Adding a layer of indirection (page mapping) 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 sometimes effectively smaller
  - unpredictable performance
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: whom 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 time
  - 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>Reference</th>
<th>Frames</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td>1</td>
<td>1</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>5</td>
<td>4</td>
</tr>
<tr>
<td>1</td>
<td>5</td>
</tr>
<tr>
<td>2</td>
<td>5</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>5</td>
<td>3</td>
</tr>
</tbody>
</table>

.contents of frames after reference

- page fault (miss)
- hit
- marks arrival time

9 page faults
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>reference</th>
<th>frames</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>1</td>
<td>4</td>
</tr>
<tr>
<td>2</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
</tr>
</tbody>
</table>

 contents of frames after 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

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

<p>| | | | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

page fault
hit
marks most recent use

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
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