CS162
Operating Systems and
Systems Programming
Lecture 16

Memory 4: Demand Paging Policies

March 16th, 2021
Profs. Natacha Crooks and Anthony D. Joseph
http://cs162.eecs.Berkeley.edu
Recall 61C: Average Memory Access Time

- Used to compute access time probabilistically:
  \[
  \text{AMAT} = \text{Hit Rate}_L \times \text{Hit Time}_L + \text{Miss Rate}_L \times \text{Miss Time}_L
  \]
  
  \[
  \text{Hit Rate}_L + \text{Miss Rate}_L = 1
  \]
  
  \[
  \text{Hit Time}_L = \text{Time to get value from L1 cache.}
  \]
  
  \[
  \text{Miss Time}_L = \text{Hit Time}_L + \text{Miss Penalty}_L
  \]
  
  \[
  \text{Miss Penalty}_L = \text{AVG Time to get value from lower level (DRAM)}
  \]
  
  So, \[
  \text{AMAT} = \text{Hit Time}_L + \text{Miss Rate}_L \times \text{Miss Penalty}_L
  \]

- What about more levels of hierarchy?
  \[
  \text{AMAT} = \text{Hit Time}_L + \text{Miss Rate}_L \times \text{Miss Penalty}_L
  \]
  
  \[
  \text{Miss Penalty}_L = \text{AVG time to get value from lower level (L2)}
  \]
  
  \[
  = \text{Hit Time}_L + \text{Miss Rate}_L \times \text{Miss Penalty}_L
  \]
  
  \[
  \text{Miss Penalty}_L = \text{Average Time to fetch from below L2 (DRAM)}
  \]
  
  \[
  \text{AMAT} = \text{Hit Time}_L + \]
  
  \[
  \text{Miss Rate}_L \times (\text{Hit Time}_L + \text{Miss Rate}_L \times \text{Miss Penalty}_L)
  \]

- And so on … (can do this recursively for more levels!)
Recall: Caching Applied to Address Translation

- Question is one of page locality: does it exist?
  - Instruction accesses spend a lot of time on the same page (accesses sequential)
  - Stack accesses have definite locality of reference
  - Data accesses have less page locality, but still some...
- Can we have a TLB hierarchy?
  - Sure: multiple levels at different sizes/speeds
Management & Access to the Memory Hierarchy

<table>
<thead>
<tr>
<th>Level</th>
<th>Speed (ns)</th>
<th>Size (bytes)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Processor</td>
<td>0.3</td>
<td>100Bs</td>
</tr>
<tr>
<td>L1 Cache</td>
<td>1</td>
<td>10kBs</td>
</tr>
<tr>
<td>L2 Cache</td>
<td>3</td>
<td>100kB</td>
</tr>
<tr>
<td>L3 Cache (shared)</td>
<td>10-30</td>
<td>MBs</td>
</tr>
<tr>
<td>Main Memory (DRAM)</td>
<td>100</td>
<td>100GBs</td>
</tr>
<tr>
<td>Secondary Storage (SSD)</td>
<td>100,000 (0.1 ms)</td>
<td>1TBs</td>
</tr>
<tr>
<td>Secondary Storage (Disk)</td>
<td>10,000,000 (10 ms)</td>
<td>TBs</td>
</tr>
</tbody>
</table>

Accessed in Hardware

Managed in Hardware

Managed in Software - OS
Recall: Demand Paging Mechanisms

- PTE makes demand paging implementable
  - Valid $\Rightarrow$ Page in memory, PTE points at physical page
  - Not Valid $\Rightarrow$ Page not in memory; use info in PTE to find it on disk when necessary

- Suppose user references page with invalid PTE?
  - Memory Management Unit (MMU) traps to OS
    » Resulting trap is a “Page Fault”
  - What does OS do on a Page Fault?:
    » Choose an old page to replace
    » If old page modified (“D=1”), write contents back to disk
    » Change its PTE and any cached TLB to be invalid
    » Load new page into memory from disk
    » Update page table entry, invalidate TLB for new entry
    » Continue thread from original faulting location
  - TLB for new page will be loaded when thread continued!
  - While pulling pages off disk for one process, OS runs another process from ready queue
    » Suspended process sits on wait queue
Recall: Steps in Handling a Page Fault

1. Trap
2. Reference
3. Page is on backing store
4. Bring in missing page
5. Reset page table
6. Restart instruction
Recall: Working Set Model

- As a program executes it transitions through a sequence of “working sets” consisting of varying sized subsets of the address space.
Demand Paging Cost Model

• Since Demand Paging like caching, can compute average access time! ("Effective Access Time")
  – EAT = Hit Rate x Hit Time + Miss Rate x Miss Time
  – EAT = Hit Time + Miss Rate x Miss Penalty

• Example:
  – Memory access time = 200 nanoseconds
  – Average page-fault service time = 8 milliseconds
  – Suppose p = Probability of miss, 1-p = Probably of hit
  – Then, we can compute EAT as follows:
    \[ \text{EAT} = 200\text{ns} + p \times 8\text{ms} \]
    \[ = 200\text{ns} + p \times 8,000,000\text{ns} \]

• If one access out of 1,000 causes a page fault, then EAT = 8.2 μs:
  – This is a slowdown by a factor of 40!

• What if want slowdown by less than 10%?
  – EAT < 200ns x 1.1 \(\Rightarrow\) p < 2.5 x 10^{-6}
  – This is about 1 page fault in 400,000!
What Factors Lead to Misses in Page Cache?

• Compulsory Misses:
  – Pages that have never been paged into memory before
  – How might we remove these misses?
    » Prefetching: loading them into memory before needed
    » Need to predict future somehow! More later

• Capacity Misses:
  – Not enough memory. Must somehow increase available memory size.
  – Can we do this?
    » One option: Increase amount of DRAM (not quick fix!)
    » Another option: If multiple processes in memory: adjust percentage of memory allocated to each one!

• Conflict Misses:
  – Technically, conflict misses don’t exist in virtual memory, since it is a “fully-associative” cache

• Policy Misses:
  – Caused when pages were in memory, but kicked out prematurely because of the replacement policy
  – How to fix? Better replacement policy
Page Replacement Policies

• Why do we care about Replacement Policy?
  – Replacement is an issue with any cache
  – Particularly important with pages
    » The cost of being wrong is high: must go to disk
    » Must keep important pages in memory, not toss them out

• FIFO (First In, First Out)
  – Throw out oldest page. Be fair – let every page live in memory for same amount of time.
  – Bad – throws out heavily used pages instead of infrequently used

• RANDOM:
  – Pick random page for every replacement
  – Typical solution for TLB’s. Simple hardware
  – Pretty unpredictable – makes it hard to make real-time guarantees

• MIN (Minimum):
  – Replace page that won’t be used for the longest time
  – Great (provably optimal), but can’t really know future…
  – But past is a good predictor of the future …
Replacement Policies (Con’t)

• LRU (Least Recently Used):
  – Replace page that hasn’t been used for the longest time
  – Programs have locality, so if something not used for a while, unlikely to be used in the near future.
  – Seems like LRU should be a good approximation to MIN.

• How to implement LRU? Use a list:
  – On each use, remove page from list and place at head
  – LRU page is at tail

• Problems with this scheme for paging?
  – Need to know immediately when page used so that can change position in list…
  – Many instructions for each hardware access

• In practice, people approximate LRU (more later)
Example: FIFO (strawman)

- Suppose we have 3 page frames, 4 virtual pages, and following reference stream:
  - A B C A B D A D B C B
- Consider FIFO Page replacement:

<table>
<thead>
<tr>
<th>Ref: Page:</th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>A</th>
<th>B</th>
<th>D</th>
<th>A</th>
<th>D</th>
<th>B</th>
<th>C</th>
<th>B</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>A</td>
<td></td>
<td></td>
<td></td>
<td>D</td>
<td></td>
<td></td>
<td></td>
<td>C</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td>B</td>
<td></td>
<td>A</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>C</td>
<td></td>
<td></td>
<td>B</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- FIFO: 7 faults
- When referencing D, replacing A is bad choice, since need A again right away
Example: MIN / LRU

• Suppose we have the same reference stream:
  – A B C A B D A D B C B

• Consider MIN Page replacement:

<table>
<thead>
<tr>
<th>Ref: Page</th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>A</th>
<th>B</th>
<th>D</th>
<th>A</th>
<th>D</th>
<th>B</th>
<th>C</th>
<th>B</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>A</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>C</td>
</tr>
<tr>
<td>2</td>
<td>B</td>
<td></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>C</td>
<td></td>
<td>D</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

• MIN: 5 faults
  – Where will D be brought in? Look for page not referenced farthest in future

• What will LRU do?
  – Same decisions as MIN here, but won't always be true!
Is LRU guaranteed to perform well?

- Consider the following: A B C D A B C D A B C D
- LRU Performs as follows (same as FIFO here):

<table>
<thead>
<tr>
<th>Ref: Page</th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>A</td>
<td>D</td>
<td>C</td>
<td>B</td>
<td>A</td>
<td>D</td>
<td>C</td>
<td>B</td>
<td>A</td>
<td>D</td>
<td>C</td>
<td>B</td>
</tr>
<tr>
<td>2</td>
<td>B</td>
<td>A</td>
<td>D</td>
<td>C</td>
<td>B</td>
<td>A</td>
<td>D</td>
<td>C</td>
<td>B</td>
<td>A</td>
<td>D</td>
<td>C</td>
</tr>
<tr>
<td>3</td>
<td>C</td>
<td>B</td>
<td>A</td>
<td>D</td>
<td>C</td>
<td>B</td>
<td>A</td>
<td>D</td>
<td>C</td>
<td>B</td>
<td>A</td>
<td>D</td>
</tr>
</tbody>
</table>

- Every reference is a page fault!
- Fairly contrived example of working set of N+1 on N frames
When will LRU perform badly?

- Consider the following: A B C D A B C D A B C D
- LRU Performs as follows (same as FIFO here):

<table>
<thead>
<tr>
<th></th>
<th>Ref:</th>
<th>Page:</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>A</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>B</td>
<td></td>
</tr>
<tr>
<td></td>
<td>C</td>
<td></td>
</tr>
<tr>
<td></td>
<td>D</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
<td>A</td>
</tr>
<tr>
<td>2</td>
<td></td>
<td>B</td>
</tr>
<tr>
<td>3</td>
<td></td>
<td>C</td>
</tr>
</tbody>
</table>

- Every reference is a page fault!

- MIN Does much better:

<table>
<thead>
<tr>
<th></th>
<th>Ref:</th>
<th>Page:</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>A</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>B</td>
<td></td>
</tr>
<tr>
<td></td>
<td>C</td>
<td></td>
</tr>
<tr>
<td></td>
<td>D</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
<td>A</td>
</tr>
<tr>
<td>2</td>
<td></td>
<td>B</td>
</tr>
<tr>
<td>3</td>
<td></td>
<td>C</td>
</tr>
</tbody>
</table>

3/16/21
• One desirable property: When you add memory the miss rate drops (stack property)
  – Does this always happen?
  – Seems like it should, right?
• No: Bélády’s anomaly
  – Certain replacement algorithms (FIFO) don’t have this obvious property!

Graph of Page Faults Versus The Number of Frames
Adding Memory Doesn’t Always Help Fault Rate

• Does adding memory reduce number of page faults?
  – Yes for LRU and MIN
  – Not necessarily for FIFO! (Called Bélády’s anomaly)

<table>
<thead>
<tr>
<th>Ref: Page</th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>A</th>
<th>B</th>
<th>E</th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>A</td>
<td></td>
<td>D</td>
<td></td>
<td></td>
<td>E</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>B</td>
<td></td>
<td>A</td>
<td></td>
<td></td>
<td>C</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>C</td>
<td></td>
<td>B</td>
<td></td>
<td></td>
<td>D</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Ref: Page</th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>A</th>
<th>B</th>
<th>E</th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>A</td>
<td></td>
<td>E</td>
<td></td>
<td></td>
<td>D</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>B</td>
<td></td>
<td>A</td>
<td></td>
<td></td>
<td>E</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>C</td>
<td></td>
<td>B</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>D</td>
<td></td>
<td>C</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

• After adding memory:
  – With FIFO, contents can be completely different
  – In contrast, with LRU or MIN, contents of memory with X pages are a subset of contents with X+1 pages
Approximating LRU: Clock Algorithm

Set of all pages in Memory

Single Clock Hand:
Advances only on page fault!
Check for pages not used recently
Mark pages as not used recently

• **Clock Algorithm:** Arrange physical pages in circle with single clock hand
  – Approximate LRU (*approximation to approximation to MIN*)
  – Replace an old page, not the oldest page
• **Details:**
  – Hardware “*use*” bit per physical page (called “*accessed*” in Intel architecture):
    » Hardware sets *use* bit on each reference
    » If *use* bit isn’t set, means not referenced in a long time
    » Some hardware sets *use* bit in the TLB; must be copied back to page TLB entry gets replaced
  – On page fault:
    » Advance clock hand (not real time)
    » Check *use* bit: 1 → used recently; clear and leave alone
      0 → selected candidate for replacement
Clock Algorithm: More details

- Will always find a page or loop forever?
  - Even if all use bits set, will eventually loop all the way around ⇒ FIFO
- What if hand moving slowly?
  - Good sign or bad sign?
    » Not many page faults
    » Or find page quickly
- What if hand is moving quickly?
  - Lots of page faults and/or lots of reference bits set
- One way to view clock algorithm:
  - Crude partitioning of pages into two groups: young and old
  - Why not partition into more than 2 groups?
**N**\(^{th}\) Chance version of Clock Algorithm

- **N**\(^{th}\) chance algorithm: Give page N chances
  - OS keeps counter per page: # sweeps
  - On page fault, OS checks use bit:
    - 1 \(\rightarrow\) clear use and also clear counter (used in last sweep)
    - 0 \(\rightarrow\) increment counter; if count=N, replace page
  - Means that clock hand has to sweep by N times without page being used before page is replaced

- How do we pick N?
  - Why pick large N? Better approximation to LRU
    - If N \(\sim\) 1K, really good approximation
  - Why pick small N? More efficient
    - Otherwise might have to look a long way to find free page

- What about “**modified**” (or “**dirty**”) pages?
  - Takes extra overhead to replace a dirty page, so give dirty pages an extra chance before replacing?
  - Common approach:
    - Clean pages, use N=1
    - Dirty pages, use N=2 (and write back to disk when N=1)
Recall: Meaning of PTE bits

- Which bits of a PTE entry are useful to us for the Clock Algorithm? Remember Intel PTE:

<table>
<thead>
<tr>
<th>PTE:</th>
<th>Page Frame Number (Physical Page Number)</th>
<th>Free (OS)</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
</tr>
</thead>
<tbody>
<tr>
<td>31-12</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- The "Present" bit (called "Valid" elsewhere):
  - P==0: Page is invalid and a reference will cause page fault
  - P==1: Page frame number is valid and MMU is allowed to proceed with translation

- The "Writable" bit (could have opposite sense and be called "Read-only"):
  - W==0: Page is read-only and cannot be written.
  - W==1: Page can be written

- The "Accessed" bit (called "Use" elsewhere):
  - A==0: Page has not been accessed (or used) since last time software set A→0
  - A==1: Page has been accessed (or used) since last time software set A→0

- The "Dirty" bit (called "Modified" elsewhere):
  - D==0: Page has not been modified (written) since PTE was loaded
  - D==1: Page has changed since PTE was loaded
Clock Algorithms Variations

• Do we really need hardware-supported “modified” bit?
  – No. Can emulate it using read-only bit
    » Need software DB of which pages are allowed to be written (needed this anyway)
    » We will tell MMU that pages have more restricted permissions than the actually do to force page faults (and allow us notice when page is written)
  – Algorithm (Clock-Emulated-M):
    » Initially, mark all pages as read-only ($W \rightarrow 0$), even writable data pages.
      Further, clear all software versions of the “modified” bit → 0 (page not dirty)
    » Writes will cause a page fault. Assuming write is allowed, OS sets software “modified” bit → 1, and marks page as writable ($W \rightarrow 1$).
    » Whenever page written back to disk, clear “modified” bit → 0, mark read-only
Clock Algorithms Variations (continued)

• Do we really need a hardware-supported “use” bit?
  – No. Can emulate it similar to above (e.g. for read operation)
    » Kernel keeps a “use” bit and “modified” bit for each page
  – Algorithm (Clock-Emulated-Use-and-M):
    » Mark all pages as invalid, even if in memory.
      Clear emulated “use” bits → 0 and “modified” bits → 0 for all pages (not used, not dirty)
    » Read or write to invalid page traps to OS to tell use page has been used
    » OS sets “use” bit → 1 in software to indicate that page has been “used”.
      Further:
      1) If read, mark page as read-only, W→0 (will catch future writes)
      2) If write (and write allowed), set “modified” bit → 1, mark page as writable (W→1)
    » When clock hand passes, reset emulated “use” bit → 0 and mark page as invalid again
    » Note that “modified” bit left alone until page written back to disk

• Remember, however, clock is just an approximation of LRU!
  – Can we do a better approximation, given that we have to
    take page faults on some reads and writes to collect use information?
  – Need to identify an old page, not oldest page!
  – Answer: second chance list
Second-Chance List Algorithm (VAX/VMS)

- Split memory in two: Active list (RW), SC list (Invalid)
- Access pages in Active list at full speed
- Otherwise, Page Fault
  - Always move overflow page from end of Active list to front of Second-chance list (SC) and mark invalid
  - Desired Page On SC List: move to front of Active list, mark RW
  - Not on SC list: page in to front of Active list, mark RW; page out LRU victim at end of SC list
Second-Chance List Algorithm (continued)

• How many pages for second chance list?
  – If 0 ⇒ FIFO
  – If all ⇒ LRU, but page fault on every page reference

• Pick intermediate value. Result is:
  – Pro: Few disk accesses (page only goes to disk if unused for a long time)
  – Con: Increased overhead trapping to OS (software / hardware tradeoff)

• With page translation, we can adapt to any kind of access the program makes
  – Later, we will show how to use page translation / protection to share memory between threads on widely separated machines

• History: The VAX architecture did not include a “use” bit. Why did that omission happen???
  – Strecker (architect) asked OS people, they said they didn’t need it, so didn’t implement it
  – He later got blamed, but VAX did OK anyway
Free List

- Keep set of free pages ready for use in demand paging
  - Freelist filled in background by Clock algorithm or other technique (“Pageout demon”)
  - Dirty pages start copying back to disk when enter list
- Like VAX second-chance list
  - If page needed before reused, just return to active set
- Advantage: faster for page fault
  - Can always use page (or pages) immediately on fault
Reverse Page Mapping (Sometimes called “Coremap”)  

- When evicting a page frame, how to know which PTEs to invalidate?  
  - Hard in the presence of shared pages (forked processes, shared memory, …)  
- Reverse mapping mechanism must be very fast  
  - Must track down all page tables pointing at given page frame when freeing a page  
  - Must track down all PTEs when seeing if pages “active”  
- Implementation options:  
  - For every page descriptor, keep linked list of page table entries that point to it  
    » Management nightmare – expensive  
  - Linux: Object-based reverse mapping  
    » Link together memory region descriptors instead (much coarser granularity)
Allocation of Page Frames (Memory Pages)

• How do we allocate memory among different processes?
  – Does every process get the same fraction of memory? Different fractions?
  – Should we completely swap some processes out of memory?

• Each process needs *minimum* number of pages
  – Want to make sure that all processes that are loaded into memory can make forward progress
  – Example: IBM 370 – 6 pages to handle SS MOVE instruction:
    » instruction is 6 bytes, might span 2 pages
    » 2 pages to handle from
    » 2 pages to handle to

• Possible Replacement Scopes:
  – *Global replacement* – process selects replacement frame from set of all frames;
    one process can take a frame from another
  – *Local replacement* – each process selects from only its own set of allocated frames
Fixed/Priority Allocation

- **Equal allocation** (Fixed Scheme):
  - Every process gets same amount of memory
  - Example: 100 frames, 5 processes → process gets 20 frames

- **Proportional allocation** (Fixed Scheme)
  - Allocate according to the size of process
  - Computation proceeds as follows:
    \[ s_i = \text{size of process } p_i \text{ and } S = \sum s_i \]
    \[ m = \text{total number of physical frames in the system} \]
    \[ a_i = (\text{allocation for } p_i) = \frac{s_i}{S} \times m \]

- **Priority Allocation**:
  - Proportional scheme using priorities rather than size
    - Same type of computation as previous scheme
  - Possible behavior: If process \( p_i \) generates a page fault, select for replacement a frame from a process with lower priority number

- Perhaps we should use an adaptive scheme instead???
  - What if some application just needs more memory?
Page-Fault Frequency Allocation

• Can we reduce Capacity misses by dynamically changing the number of pages/application?

• Establish “acceptable” page-fault rate
  – If actual rate too low, process loses frame
  – If actual rate too high, process gains frame

• Question: What if we just don’t have enough memory?
Thrashing

- If a process does not have “enough” pages, the page-fault rate is very high. This leads to:
  - low CPU utilization
  - operating system spends most of its time swapping to disk
- **Thrashing** $\equiv$ a process is busy swapping pages in and out with little or no actual progress
- Questions:
  - How do we detect Thrashing?
  - What is best response to Thrashing?
Locality In A Memory-Reference Pattern

- Program Memory Access Patterns have temporal and spatial locality
  - Group of Pages accessed along a given time slice called the “Working Set”
  - Working Set defines minimum number of pages for process to behave well
- Not enough memory for Working Set ⇒ Thrashing
  - Better to swap out process?
Working-Set Model

- Δ ≡ working-set window ≡ fixed number of page references
  - Example: 10,000 instructions
- WS_i (working set of Process P_i) = total set of pages referenced in the most recent Δ (varies in time)
  - if Δ too small will not encompass entire locality
  - if Δ too large will encompass several localities
  - if Δ = ∞ ⇒ will encompass entire program
- D = Σ|WS_i| ≡ total demand frames
- if D > m ⇒ Thrashing
  - Policy: if D > m, then suspend/swap out processes
  - This can improve overall system behavior by a lot!

page reference table

... 2 6 1 5 7 7 7 5 1 6 2 3 4 1 2 3 4 4 3 4 4 4 1 3 2 3 4 4 3 4 4 4 ...

WS(t_1) = {1,2,5,6,7}  WS(t_2) = {3,4}
What about Compulsory Misses?

- Recall that compulsory misses are misses that occur the first time that a page is seen
  - Pages that are touched for the first time
  - Pages that are touched after process is swapped out/swapped back in

- **Clustering:**
  - On a page-fault, bring in multiple pages “around” the faulting page
  - Since efficiency of disk reads increases with sequential reads, makes sense to read several sequential pages

- **Working Set Tracking:**
  - Use algorithm to try to track working set of application
  - When swapping process back in, swap in working set
Linux Memory Details?

- Memory management in Linux considerably more complex than the examples we have been discussing
- Memory Zones: physical memory categories
  - ZONE_DMA: < 16MB memory, DMA-able on ISA bus
  - ZONE_NORMAL: 16MB → 896MB (mapped at 0xC0000000)
  - ZONE_HIGHMEM: Everything else (> 896MB)
- Each zone has 1 freelist, 2 LRU lists (Active/Inactive)
- Many different types of allocation
  - SLAB allocators, per-page allocators, mapped/unmapped
- Many different types of allocated memory:
  - Anonymous memory (not backed by a file, heap/stack)
  - Mapped memory (backed by a file)
- Allocation priorities
  - Is blocking allowed/etc.
Linux Virtual Memory Map (Pre-Meltdown)

32-Bit Virtual Address Space

User Addresses

0x00000000

Kernel Addresses

0xFFFFFFFF

1GB

896MB Physical

0xC0000000

3GB Total

0x00000000

64-Bit Virtual Address Space

User Addresses

0x00000000

Kernel Addresses

0xFFFFFFFF

128TiB

64 TiB Physical

0xFFFF800000000000

“Canonical Hole”

0x0000000000000000

0x00007FFFFFFF

0xFFFF800000000000

0xFFFFFFFF

0x0000000000000000

0xFFFFFFFF

128TiB

896MB

128 TiB

64 TiB

32-Bit Virtual Address Space
Pre-Meltdown Virtual Map (Details)

• Kernel memory not generally visible to user
  – Exception: special VDSO (virtual dynamically linked shared objects) facility that maps kernel code into user space to aid in system calls (and to provide certain actual system calls such as gettimeofday())

• Every physical page described by a “page” structure
  – Collected together in lower physical memory
  – Can be accessed in kernel virtual space
  – Linked together in various “LRU” lists

• For 32-bit virtual memory architectures:
  – When physical memory < 896MB
    » All physical memory mapped at 0xC0000000
  – When physical memory >= 896MB
    » Not all physical memory mapped in kernel space all the time
    » Can be temporarily mapped with addresses > 0xCC000000

• For 64-bit virtual memory architectures:
  – All physical memory mapped above 0xFFFF800000000000
Post Meltdown Memory Map

• Meltdown flaw (2018, Intel x86, IBM Power, ARM)
  – Exploit speculative execution to observe contents of kernel memory

```
1:  // Set up side channel (array flushed from cache)
2:  uchar array[256 * 4096];
3:  flush(array);  // Make sure array out of cache

4:  try {
5:    uchar result = *(uchar *)kernel_address; // Try access!
6:    uchar dummy = array[result * 4096];  // leak info!
7:  } catch(){;}  // Could use signal() and setjmp/longjmp

8:  // scan through 256 array slots to determine which loaded
```

– Some details:
  » Reason we skip 4096 for each value: avoid hardware cache prefetch
  » Note that value detected by fact that one cache line is loaded
  » Catch and ignore page fault: set signal handler for SIGSEGV, can use setjump/longjmp.

• Patch: Need different page tables for user and kernel
  – Without PCID tag in TLB, flush TLB twice on syscall (800% overhead!)
  – Need at least Linux v 4.14 which utilizes PCID tag in new hardware to avoid flushing when change address space

• Fix: better hardware without timing side-channels
  – Will be coming, but still in works
Summary

- **Replacement policies**
  - FIFO: Place pages on queue, replace page at end
  - MIN: Replace page that will be used farthest in future
  - LRU: Replace page used farthest in past
- **Clock Algorithm: Approximation to LRU**
  - Arrange all pages in circular list
  - Sweep through them, marking as not “in use”
  - If page not “in use” for one pass, then can replace
- **Nth-chance clock algorithm: Another approximate LRU**
  - Give pages multiple passes of clock hand before replacing
- **Second-Chance List algorithm: Yet another approximate LRU**
  - Divide pages into two groups, one of which is truly LRU and managed on page faults.
- **Working Set:**
  - Set of pages touched by a process recently
- **Thrashing:** a process is busy swapping pages in and out
  - Process will thrash if working set doesn’t fit in memory
  - Need to swap out a process