Swapping: Mechanisms and Policies

CMPU 334 – Operating Systems
Jason Waterman
Beyond Physical Memory: Mechanisms

- How can we maintain the illusion of a large virtual address space?
  - Virtual address space can be much larger than physical memory
  - Support multiprogramming

- Add a new level to our memory hierarchy
  - A place for the OS to stash away portions of the address space that currently aren’t in great demand
  - In modern systems, this role is usually served by a hard disk drive
Swap Space

- Reserve some space on the disk for moving pages back and forth
- OS reads and writes the swap space in **page-sized units**

![Diagram of Physical Memory and Swap Space]

Physical Memory

- PFN 0
- PFN 1
- PFN 2
- PFN 3

Swap Space

- Block 0
- Block 1
- Block 2
- Block 3
- Block 4
- Block 5
- Block 6
- Block 7

Proc 0
[VPN 0]

Proc 1
[VPN 2]

Proc 1
[VPN 3]

Proc 2
[VPN 0]
Present Bit

• Need some machinery higher up in the system in order to support swapping pages to and from the disk

• Add a **present bit** to the PTE
  • When the hardware looks in the PTE, it may find that the page is not present in physical memory

• Accessing a page that is not in physical memory is known as a **Page Fault**

<table>
<thead>
<tr>
<th>Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>page is present in physical memory</td>
</tr>
<tr>
<td>0</td>
<td>The page is not in memory but rather on disk.</td>
</tr>
</tbody>
</table>
The Page Fault

• A page fault occurs by accessing a page that is **not in physical memory**
  • If a page is not present and has been swapped disk, the OS needs to swap the page into memory in order to service the page fault

• How does the OS know where to look for the page on disk?
  • Store the disk address in the PTE

• While OS is fetching the page, the process is blocked (allowing another process to run)

• After the OS fetches the page from disk, update the page table to mark page as present
  • Update the PTE with thePFN of the page now in memory
  • Optionally update the TLB

• Restart the instruction
What If Memory Is Full?

• The OS needs to make room for the new page

• The process of picking one or more pages to swap out is known as page-replacement policy

• Getting this policy right is important to overall performance
  • Hard drives are 10,000 to 100,000 times slower than memory!
Page Fault Control Flow Summary

• After a TLB miss
  • Check present and valid bits of the PTE
    • If valid bit is 0, seg fault
    • If present bit is 1, grab the PFN from the PTE, place in TLB and restart the instruction
    • If present bit is 0, page must be swapped in from disk (page fault)

• Servicing a Page Fault
  • Find an available physical frame for the soon-to-be-swapped-in page
  • If there is not a physical frame available (physical memory is full) need to wait for page replacement algorithm to run and kick some pages out
  • OS issues I/O request to read in page from swap space (disk)
  • When I/O completes, update the page table and restart the instruction
When Replacements Really Occur

• Our current model of page replacement:
  • OS waits until memory is entirely full, and only then replaces a page to make room for some other page
  • This is a little bit unrealistic, and there are many reasons for the OS to keep a small portion of memory free more proactively

• When to start and stop evicting pages?
  • High Watermarks (HW) and Low Watermarks (LW)
  • Fewer than LW available pages, a background process (Page Daemon) starts evicting pages
  • Page Daemon stops when HW number of free pages are available
Swapping Summary

• Swapping is the **mechanism** that allows us to support accessing more memory than is physically present in the system

• Needs extra complexity in the page table
  • Present bit (is page in memory or not)
  • Location of page on disk if not in memory

• The page fault handler runs when a page is not present in memory
  • Transfers the page from disk to physical memory
  • May have to replace some pages in memory to make room for the swapped in page

• The **page replacement policy** can have a great impact on program performance
Beyond Physical Memory: Policies

• **Memory pressure** forces the OS to start **paging out** pages to make room for actively-used pages

• Deciding which page to evict is encapsulated within the replacement policy of the OS
Cache Management

• Goal in picking a replacement policy is to minimize the number of cache misses
• The number of cache hits and misses let us calculate the *average memory access time (AMAT)*
  • Cost of accessing memory ~ 100 ns
  • Cost of accessing disk ~ 10 ms (100,000 times slower)

\[
AMAT = (P_{Hit} \times T_M) + (P_{Miss} \times T_D)
\]

<table>
<thead>
<tr>
<th>Argument</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>(T_M)</td>
<td>The cost of accessing memory</td>
</tr>
<tr>
<td>(T_D)</td>
<td>The cost of accessing disk</td>
</tr>
<tr>
<td>(P_{Hit})</td>
<td>The probability of finding the data item in the cache (a hit)</td>
</tr>
<tr>
<td>(P_{Miss})</td>
<td>The probability of not finding the data in the cache (a miss)</td>
</tr>
</tbody>
</table>
The Optimal Replacement Policy

• Leads to the fewest number of misses overall
  • Replace the page that will be accessed furthest in the future
  • Results in the fewest-possible cache misses

• Not achievable in the real world (we can’t predict the future)
  • However, useful as a comparison point
  • Let's us know how close we are to optimal
Tracing the Optimal Policy

- Can hold three pages in memory
- Memory starts out empty
  - Cold (compulsory) misses

<table>
<thead>
<tr>
<th>Access</th>
<th>Hit/Miss?</th>
<th>Evict</th>
<th>Resulting Cache State</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Miss</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>Miss</td>
<td>0,1</td>
<td>0,1</td>
</tr>
<tr>
<td>2</td>
<td>Miss</td>
<td>0,1,2</td>
<td>0,1,2</td>
</tr>
<tr>
<td>0</td>
<td>Hit</td>
<td>0,1,2</td>
<td>0,1,2</td>
</tr>
<tr>
<td>1</td>
<td>Hit</td>
<td>0,1,2</td>
<td>0,1,2</td>
</tr>
<tr>
<td>3</td>
<td>Miss</td>
<td>2</td>
<td>0,1,3</td>
</tr>
<tr>
<td>0</td>
<td>Hit</td>
<td>0,1,3</td>
<td>0,1,3</td>
</tr>
<tr>
<td>3</td>
<td>Hit</td>
<td>0,1,3</td>
<td>0,1,3</td>
</tr>
<tr>
<td>1</td>
<td>Hit</td>
<td>0,1,3</td>
<td>0,1,3</td>
</tr>
<tr>
<td>2</td>
<td>Miss</td>
<td>3</td>
<td>0,1,2</td>
</tr>
<tr>
<td>1</td>
<td>Hit</td>
<td>0,1,2</td>
<td>0,1,2</td>
</tr>
</tbody>
</table>

Hit rate is \[
\frac{\text{Hits}}{\text{Hits} + \text{Misses}} = 54.6\%\]
A Simple Policy: FIFO

- Pages were placed in a queue when they enter the system.
- When a replacement occurs, the page on the tail of the queue (the “First-in” pages) is evicted.
  - It is simple to implement, but it can’t determine the importance of pages.

<table>
<thead>
<tr>
<th>Access</th>
<th>Hit/Miss?</th>
<th>Evict</th>
<th>Resulting Cache State</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Miss</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>Miss</td>
<td>0,1</td>
<td>0,1</td>
</tr>
<tr>
<td>2</td>
<td>Miss</td>
<td>0,1,2</td>
<td>0,1,2</td>
</tr>
<tr>
<td>0</td>
<td>Hit</td>
<td>0,1,2</td>
<td>0,1,2</td>
</tr>
<tr>
<td>1</td>
<td>Hit</td>
<td>0,1,2</td>
<td>0,1,2</td>
</tr>
<tr>
<td>3</td>
<td>Miss</td>
<td>0</td>
<td>1,2,3</td>
</tr>
<tr>
<td>0</td>
<td>Miss</td>
<td>1</td>
<td>2,3,0</td>
</tr>
<tr>
<td>3</td>
<td>Hit</td>
<td></td>
<td>2,3,0</td>
</tr>
<tr>
<td>1</td>
<td>Miss</td>
<td>2</td>
<td>3,0,1</td>
</tr>
<tr>
<td>2</td>
<td>Miss</td>
<td>3</td>
<td>0,1,2</td>
</tr>
<tr>
<td>1</td>
<td>Hit</td>
<td></td>
<td>0,1,2</td>
</tr>
</tbody>
</table>

Hit rate is \( \frac{\text{Hits}}{\text{Hits} + \text{Misses}} = 36.4\% \)

Even though page 0 had been accessed a number of times, FIFO still kicks it out.
Another Simple Policy: Random

- Picks a random page to replace under memory pressure
  - It doesn’t really try to be too intelligent in picking which blocks to evict
  - Random does depends entirely upon how lucky it is in its choice

<table>
<thead>
<tr>
<th>Access</th>
<th>Hit/Miss?</th>
<th>Evict</th>
<th>Resulting Cache State</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Miss</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>Miss</td>
<td>0,1</td>
<td>0,1</td>
</tr>
<tr>
<td>2</td>
<td>Miss</td>
<td>0,1,2</td>
<td>0,1,2</td>
</tr>
<tr>
<td>0</td>
<td>Hit</td>
<td>0,1,2</td>
<td>0,1,2</td>
</tr>
<tr>
<td>1</td>
<td>Hit</td>
<td>0,1,2</td>
<td>0,1,2</td>
</tr>
<tr>
<td>3</td>
<td>Miss</td>
<td>0</td>
<td>1,2,3</td>
</tr>
<tr>
<td>0</td>
<td>Miss</td>
<td>1</td>
<td>2,3,0</td>
</tr>
<tr>
<td>3</td>
<td>Hit</td>
<td>2,3,0</td>
<td>2,3,0</td>
</tr>
<tr>
<td>1</td>
<td>Miss</td>
<td>3</td>
<td>2,0,1</td>
</tr>
<tr>
<td>2</td>
<td>Hit</td>
<td>2,0,1</td>
<td>2,0,1</td>
</tr>
<tr>
<td>1</td>
<td>Hit</td>
<td>2,0,1</td>
<td>2,0,1</td>
</tr>
</tbody>
</table>
Random Performance

- Sometimes, random is as good as optimal, achieving 6 hits on the example trace
Using History

• Using the past to predict the future
  • Two types of historical information

<table>
<thead>
<tr>
<th>Historical Information</th>
<th>Meaning</th>
<th>Algorithms</th>
</tr>
</thead>
<tbody>
<tr>
<td>recency</td>
<td>The more recently a page has been accessed, the more likely it will be accessed again</td>
<td>LRU</td>
</tr>
<tr>
<td>frequency</td>
<td>If a page has been accessed many times, it should not be replaced as it clearly has some value</td>
<td>LFU</td>
</tr>
</tbody>
</table>
LRU Example

- Replaces the least-recently-used page

<table>
<thead>
<tr>
<th>Access</th>
<th>Hit/Miss?</th>
<th>Evict</th>
<th>Resulting Cache State</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Miss</td>
<td></td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>Miss</td>
<td>0,1</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>Miss</td>
<td>0,1,2</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>Hit</td>
<td>1,2,0</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>Hit</td>
<td>2,0,1</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>Miss</td>
<td>2</td>
<td>0,1,3</td>
</tr>
<tr>
<td>0</td>
<td>Hit</td>
<td>1,3,0</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>Hit</td>
<td>1,0,3</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>Hit</td>
<td>0,3,1</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>Miss</td>
<td>0</td>
<td>3,1,2</td>
</tr>
<tr>
<td>1</td>
<td>Hit</td>
<td>3,2,1</td>
<td></td>
</tr>
</tbody>
</table>
Workload Example: The No-Locality Workload

- Each reference is to a random page within the set of accessed pages
  - Workload accesses 100 unique pages over time
  - Choosing the next page to refer to at random

![Graph showing hit rate vs. cache size for different cache replacement policies.](image)

When the cache is large enough to fit the entire workload, it also doesn’t matter which policy you use.
Workload Example: The 80-20 Workload

- Exhibits locality: 80% of the references are made to 20% of the pages.
- The remaining 20% of the references are made to the remaining 80% of the pages.

![The 80-20 Workload Diagram]

**LRU is more likely to hold onto the hot pages.**
Workload Example: The Looping Sequential

- Refer to 50 pages in sequence
  - Starting at 0, then 1, ... up to page 49, and then we Loop, repeating those accesses, for total of 10,000 accesses to 50 unique pages
- **Worst case** scenario for LRU and FIFO

```
<table>
<thead>
<tr>
<th>Cache Size (Blocks)</th>
<th>OPT</th>
<th>LRU</th>
<th>FIFO</th>
<th>RAND</th>
</tr>
</thead>
<tbody>
<tr>
<td>20</td>
<td>100%</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>40</td>
<td>80%</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>60</td>
<td>60%</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>80</td>
<td>40%</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>100</td>
<td>20%</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

The Looping-Sequential Workload
Implementing Historical Algorithms

• To keep track of which pages have been least-and-recently used, the system has to do some accounting work on every memory reference
  • Need support from hardware
  • Can be expensive

• Can we approximate an LRU policy with less overhead?
Approximating LRU

- Requires some hardware support, in the form of a **use bit**
  - Whenever a page is referenced, the use bit is set by hardware to 1
  - Hardware *never* clears the bit, though; that is the responsibility of the OS

- **Clock Algorithm**
  - All pages of the system are arranged in a circular list
  - A clock hand points to a particular page
  - When a page fault occurs, the page the hand is pointing to is inspected
    - The action taken depends on the use bit
    - The algorithm continues until it finds a use bit that is set to 0

<table>
<thead>
<tr>
<th>Use bit</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Evict the page</td>
</tr>
<tr>
<td>1</td>
<td>Clear use bit and advance hand</td>
</tr>
</tbody>
</table>
Workload with Clock Algorithm

- Clock algorithm doesn’t do as well as LRU
- But does better than approaches that don’t consider history at all
Considering Dirty Pages

• The hardware includes a modified bit (a.k.a dirty bit) for each page
  • This bit is set at any time a page is written
  • If a page has been modified, it must be written back to disk when evicted
  • If a page has not been modified, the eviction is free

• Page replacement algorithm might prefer to evict clean pages over dirty ones
  • E.g., clock algorithm could scan for pages that are both unused and clean to evict first, before evicting unused pages that are dirty
Page Selection Policy

• The OS must decide when to bring a page into memory
• For most pages, the OS uses *demand paging*
  • Brings the page into memory when it is accessed
• Other policy is *prefetching*
  • Guess what page is about to be used
  • Should only be done where there is a good chance of success
Policy for writing pages to disk

- Clustering
  - Collect several pending writes together in memory and write them to disk in one write
  - A single large write is more efficient than many small ones
Thrashing

- When memory is oversubscribed and the memory demands of the set of running processes exceeds the available physical memory

- **Admission control**
  - Decide not to run a set of processes
  - Kill off some processes
  - Reduce the set of processes working sets to fit in memory