Chapter 9: Virtual Memory
Logistics

- Hwk #4 due today
- Hwk #5 posted today
- Midterms back now!
- PA #2 posted last week: theme = **deadlock**

- Midterm stats next page
Stats

- Average: 70.84
- Std. Deviation: 14.03
  - So, most grades lie between ~56 and ~85

Histogram:
## Per problem stats

<table>
<thead>
<tr>
<th>Problem Stat</th>
<th>1.1</th>
<th>1.2</th>
<th>2.1</th>
<th>2.2</th>
<th>3.1</th>
<th>4.1</th>
<th>4.2</th>
<th>4.3</th>
</tr>
</thead>
<tbody>
<tr>
<td>MEAN</td>
<td>9.282051282</td>
<td>9</td>
<td>11.25641026</td>
<td>4.025641026</td>
<td>8.435897436</td>
<td>3.692307692</td>
<td>19.41025641</td>
<td>5.743589744</td>
</tr>
<tr>
<td>STD. DEV</td>
<td>1.999325122</td>
<td>4.518965881</td>
<td>3.522244714</td>
<td>3.956920516</td>
<td>2.573088169</td>
<td>1.07981105</td>
<td>4.649379974</td>
<td>2.592681489</td>
</tr>
<tr>
<td>MAX</td>
<td>10</td>
<td>15</td>
<td>15</td>
<td>10</td>
<td>10</td>
<td>4</td>
<td>24</td>
<td>12</td>
</tr>
</tbody>
</table>

### Problem areas:

- **1.2**: Parts of processes/threads; shared vs private
- **2.2**: Caches
- **4.3**: Late arrival
Virtual memory

Last time we concerned ourselves with:

- Managing main memory
- Exploiting the benefits of abstracting logical/virtual addresses from physical addresses.

Virtual memory is a topic that takes this further

- Virtual addressing to hide where data lives – main memory, disk, wherever
- Virtual memory allows programs to be bigger than physical memory
- With a VM system, we can hide all kinds of data bearing or data producing entities behind what appears to be plain old memory addressing.
Virtual memory – separation of user logical memory from physical memory.

- Only part of the program needs to be in memory for execution
- Logical address space can therefore be much larger than physical address space
- Allows address spaces to be shared by several processes
- Allows for more efficient process creation

Virtual memory can be implemented via:

- Demand paging
- Demand segmentation
Virtual Memory That is Larger Than Physical Memory

This is the new part beyond last time. Pages may reside on disk in addition to physical memory.
Virtual-address Space

We still program with this model of process organization.

The paging and VM layer support this abstraction.
VM maintains the ability to share pages as we introduced with paging.
Demand Paging

- Bring a page into memory only when it is needed
  - Less I/O needed
  - Less memory needed
  - Faster response
  - More users

- Page is needed $\Rightarrow$ reference to it
  - invalid reference $\Rightarrow$ abort
  - not-in-memory $\Rightarrow$ bring to memory

- Lazy swapper – never swaps a page into memory unless page will be needed
  - Swapper that deals with pages is a pager
Swappers

- Lazy swapper only brings pages that are needed from disk.
  - Page not referenced => page isn’t swapped from disk to memory.

- The swapper discussed in chapter 8 is not lazy – entire processes are swapped in and out of memory, regardless of whether this includes unreferenced pages or not.

- The term *pager* is simply introduced to distinguish page-level swappers from whole process swappers.
Transfer of a Paged Memory to Contiguous Disk Space

Swap space organized as a set of pages.
Page swapping

- So how do we support this transparently?

- What property of memory pages could be useful in determining if a page in memory is legal to use?
Valid-Invalid Bit

- With each page table entry a valid–invalid bit is associated (\(v \Rightarrow \text{in-memory}, \ i \Rightarrow \text{not-in-memory}\))
- Initially valid–invalid bit is set to \(i\) on all entries
- Example of a page table snapshot:

  ![Diagram of a page table with valid-invalid bits]

- During address translation, if valid–invalid bit in page table entry is \(i \Rightarrow \text{page fault}\)
Page Table When Some Pages Are Not in Main Memory
Page Fault

If there is a reference to a page, first reference to that page will trap to operating system:

**page fault**

1. Operating system looks at another table to decide:
   - Invalid reference $\Rightarrow$ abort
   - Just not in memory
2. Get empty frame
3. Swap page into frame
4. Reset tables
5. Set validation bit $= v$
6. Restart the instruction that caused the page fault

*Key point:* valid/invalid bit is used to allow hardware to trigger the trap.

*The interpretation of the trap is left up to the OS.*
Steps in Handling a Page Fault

1. Trap
2. Page is on backing store
3. Operating system
4. Bring in missing page
5. Reset page table
6. Restart instruction
Cost of demand paging

- Clearly this is not going to be totally transparent.
  - Disk is *slow*.
    - Although we may get lucky if the disk has a fast cache.

- When we looked at pages, we saw that the access time could vary depending on if the page table entry of an access was in the TLB versus main memory.
  - Two memory access costs versus one

- We can quantify the performance impact of demand paging in a similar way.
Performance of Demand Paging

- Page Fault Rate $0 \leq p \leq 1.0$
  - if $p = 0$ no page faults
  - if $p = 1$, every reference is a fault

- Effective Access Time (EAT)

  $$EAT = (1 - p) \times \text{memory access} + p \times \text{page fault overhead} + \text{swap page out} + \text{swap page in} + \text{restart overhead}$$

You don’t want this program...
Demand Paging Example

- Memory access time = 200 nanoseconds
- Average page-fault service time = 8 milliseconds
- EAT = \((1 - p) \times 200 + p \times 8\) milliseconds
  = \((1 - p) \times 200 + p \times 8,000,000\)
  = 200 + p \times 7,999,800

- If one access out of 1,000 causes a page fault, then
  EAT = 8.2 microseconds.
  This is a slowdown by a factor of 40!!
EAT and demand paging

- Takeaway message
  - *Keep demand paging rare*

- This isn’t so hard most of the time.
  - Most of our processes fit in memory
  - As a result, most of the time we swap very rarely and don’t notice the overhead often
Suffering from demand paging

- Quite likely we have experienced this and may not have known what the underlying cause was.
  - Running a program that consumes most of your memory, and suddenly the entire system grinds to an unusable speed.
  - Resuming a program that was in the background for a while, and experiencing unbearable speed problems while it comes back to life.
  - Running too many programs at once, all of them run slowly.

- Each of these are cases where the VM system is doing its job
  - **Pro**: Programs executing fine (if slowly) even though consuming more than physical memory can support. Abstraction works!
  - **Con**: Disk speed is orders of magnitude slower than physical RAM. Usability of interactive apps is gone.
Process Creation

- Virtual memory allows other benefits during process creation:
  - Copy-on-Write
  - Memory-Mapped Files (later)

In fact, these have very interesting applications far beyond simple process creation.
Copy-on-Write

- Copy-on-Write (COW) allows both parent and child processes to initially share the same pages in memory.

- If either process modifies a shared page, only then is the page copied.

- COW allows more efficient process creation as only modified pages are copied.

- Free pages are allocated from a pool of zeroed-out pages.
Before Process 1 Modifies Page C
After Process 1 Modifies Page C
Out of frames?

- What happens when we run out of free frames in memory?
  - Remember:
    - page = virtual chunk of address space
    - frame = physical region of address space.

- Page replacement algorithms come into play here
  - Finding pages that are not in use
    - Swap them to disk, reuse their frame for page that is about to be used.
  - Algorithm for page replacement is important. We want to avoid swapping out pages actively in use.
Page Replacement

- Prevent over-allocation of memory by modifying page-fault service routine to include page replacement

- Use **modify (dirty) bit** to reduce overhead of page transfers – only modified pages are written to disk
  - In other words, if a page is going to be replaced and was never written to while in physical memory, we may be able to overwrite it without swapping to disk if the original version is still out in swap space.

- Page replacement completes separation between logical memory and physical memory – large virtual memory can be provided on a smaller physical memory
Need For Page Replacement

Where does B go in physical memory?
Basic Page Replacement

1. Find the location of the desired page on disk

2. Find a free frame:
   - If there is a free frame, use it
   - If there is no free frame, use a page replacement algorithm to select a *victim* frame

3. Bring the desired page into the (newly) free frame; update the page and frame tables

4. Restart the process
Page Replacement

- Change valid-invalid bit to invalid (frame 0).
- Reset page table for new page (frame f).
- Swap out victim page.
- Swap desired page in.
Page Replacement Algorithms

- Want lowest page-fault rate

- Evaluate algorithm by running it on a particular string of memory references (reference string) and computing the number of page faults on that string

- In all our examples, the reference string is

  1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5
Adding RAM corresponds to adding frames.

Ever wonder why adding RAM to a machine can make it appear to go faster? This graph is why!
First-In-First-Out (FIFO) Algorithm

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

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

- 9 page faults

- 4 frames

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

- 10 page faults

- Belady’s Anomaly: more frames ⇒ more page faults
FIFO Page Replacement

reference string

7 0 1 2 0 3 0 4 2 3 0 3 2 1 2 0 1 7 0 1

page frames

7 7 7 2
0 0 0
1 1

2 2 4 4 4 0
3 3 3 2 2 2
1 1
3 2

0 0
1 1
3 2
2 2

7 7 7
1 0 0
2 2 1
FIFO Illustrating Belady’s Anomaly

![Graph illustrating Belady’s Anomaly]

- Number of page faults vs. number of frames.
- The graph shows a scenario where Belady’s Anomaly occurs.
Optimal page replacement

- FIFO wasn’t ideal. What is?

- To start, we can perform a though experiment.
  - What would be the optimal page replacement algorithm.

- Like SJF for scheduling, let’s design an algorithm that can see into the future.
  - Obviously this won’t be implementable.
  - **BUT**, it will give us a metric to evaluate real algorithms by.
Optimal Algorithm

- Replace page that will not be used for longest period of time
- 4 frames example
  
  1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5

```
  1   4
  2
  3
  4   5
```

- How do you know this?
- Used for measuring how well your algorithm performs

6 page faults
Optimal Page Replacement

reference string

7 0 1 2 0 3 0 4 2 3 0 3 2 1 2 0 1 7 0 1

page frames

7 7 7 2 2 2 2 2 2 7
0 0 0 0 4 0 0 0 0
1 1 1 3 3 3 1 1
Realistic algorithms

- Can’t look into the future. 😞

- But we can use the past to predict what we expect to happen based on what we already saw.

- Look for page that hasn’t been used for the longest period of time.
  - Likely it’s still not going to be used in the near future.

- We can do this.
## 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>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>5</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>5</td>
<td>5</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>4</td>
<td>4</td>
<td>3</td>
<td>3</td>
<td></td>
</tr>
</tbody>
</table>

- **Counter implementation**
  - Every page entry has a counter; every time page is referenced through this entry, copy the clock into the counter
  - When a page needs to be changed, look at the counters to determine which are to change
# LRU Page Replacement

**reference string**

| 7 | 0 | 1 | 2 | 0 | 3 | 0 | 4 | 2 | 3 | 0 | 3 | 2 | 1 | 2 | 0 | 1 | 7 | 0 | 1 |
| 7 | 7 | 7 | 2 | 2 | 4 | 4 | 4 | 0 | 1 | 1 | 1 |

**page frames**

<table>
<thead>
<tr>
<th>0</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>3</th>
<th>3</th>
<th>3</th>
<th>3</th>
<th>0</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>3</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>7</td>
<td></td>
</tr>
</tbody>
</table>
LRU Algorithm (Cont.)

- Stack implementation – keep a stack of page numbers in a double link form:
  - Page referenced:
    - move it to the top
    - requires 6 pointers to be changed
  - No search for replacement
Use Of A Stack to Record The Most Recent Page References

reference string

4 7 0 7 1 0 1 2 1 2 7 1 2

stack before
a

stack after
b

2
1
0
4

7
2
1
0
4

a

b
We have a problem.
- Both techniques would require clock fields or stack to be updated on every reference.
- That would be expensive.

Likely not to have hardware assistance for real LRU.

So, we try to approximate it with what we DO have hardware assistance for.

- “Reference bits” – hardware sets this bit to 1 when a page is referenced.
- Set to zero when page is brought in.
LRU Approximation Algorithms

- **Reference bit**
  - With each page associate a bit, initially = 0
  - When page is referenced bit set to 1
  - Replace the one which is 0 (if one exists)
    - We do not know the order, however

- **Second chance**
  - Need reference bit
  - Clock replacement
  - If page to be replaced (in clock order) has reference bit = 1 then:
    - set reference bit 0
    - leave page in memory
    - replace next page (in clock order), subject to same rules
Second-Chance (clock) Page-Replacement Algorithm

```
reference bits
0
0
1
1
0
... 
1
1

next victim

pages

(circular queue of pages)
(a)

reference bits
0
0
0
0
0
...
1
1
1

pages

circular queue of pages
(b)```