Virtual Memory

Today

- Virtual memory
- Page replacement algorithms
- Modeling page replacement algorithms
Virtual memory

- Handling processes >> than allocated memory
- Keep in memory only what’s needed
  - Full address space does not need to be resident in memory
    - Leave it on disk
  - OS uses main memory as a cache
- Overlay approach
  - Implemented by user
  - Easy on the OS, hard on the programmer

Overlay for a two-pass assembler:

<table>
<thead>
<tr>
<th>Pass</th>
<th>Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pass 1</td>
<td>70KB</td>
</tr>
<tr>
<td>Pass 2</td>
<td>80KB</td>
</tr>
<tr>
<td>Symbol Table</td>
<td>20KB</td>
</tr>
<tr>
<td>Common Routines</td>
<td>30KB</td>
</tr>
<tr>
<td>Total</td>
<td>200KB</td>
</tr>
</tbody>
</table>

Two overlays: 120 + 130KB
Virtual memory

- Hide the complexity – let the OS do the job
- Virtual address space split into pages
- Physical memory split into page frames
- Page & page frames = size (512B … 64KB)
- Map pages into page frames
  - Doing the translation – OS + MMU
Pages, page frames and tables

A simple example with
- 64KB virtual address space
- 4KB pages
- 32KB physical address space
- 16 pages and 8 page frames

Try to access:

- MOV REG, 0
  Virtual address 0
  Page frame 2
  Physical address 8192

- MOV REG, 8192
  Virtual address 8192
  Page frame 6
  Physical address 24576

- MOV REG, 20500
  Virtual address 20500 (20480 + 20)
  Page frame 3
  Physical address 20+12288
Since virtual memory >> physical memory

- Use a present/absent bit
- MMU checks –
  - If not there, “page fault” to the OS (trap)
  - OS picks a victim (?)
  - … sends victim to disk
  - … brings new one
  - … updates page table

MOVE REG, 32780
Virtual address 32780
Virtual page 8, byte 12 (32768+12)
Page is unmapped – page fault!
Details of the MMU work

- MMU with 16 4KB pages
- Page # (first 4 bits) index into page table
- If not there
  - Page fault
- Else
  - Output register +
  - 12 bit offset →
  - 15 bit physical address
Page table entry

- Looking at the details of a single entry

- Page frame number – the most important field
- Protection – 1 bit for R&W or R or 3 bits for RWX
- Present/absent bit
  - Says whether or not the virtual address is used
- Modified (M): dirty bit
  - Set when a write to the page has occurred
- Referenced (R): Has it being used?
- To ensure we are not reading from cache (D)
  - Key for pages that map onto device registers rather than memory
Page replacement algorithms

- OS uses main memory as (page) cache
  - If only load *when* reference – demand paging

- Page fault – cache miss
  - Need room for new page? Page replacement algorithm
  - What’s your best candidate for removal?
    - The one you will never touch again – duh!

- **What do you do with victim page?**
  - A modified page must first be saved
  - An unmodified one just overwritten
  - Better not to choose an often used page
    - It will probably need to be brought back in soon
How can any of this work?!?!?

- **Locality**
  - Temporal locality – location recently referenced tend to be referenced again soon
  - Spatial locality – locations near recently referenced are more likely to be referenced soon

- **Locality means paging could be infrequent**
  - Once you brought a page in, you’ll use it many times
  - Some issues that may play against you
    - Degree of locality of application
    - Page replacement policy and application reference pattern
    - Amount of physical memory and application footprint
Optimal algorithm (Belady’s algorithm)

- For now, assume a process pages against itself, using a fixed number of page frames
- Best page to replace – the one you’ll never need again
  - Replace page needed at the farthest point in future
  - Optimal but unrealizable
- Estimate by …
  - Logging page use on previous runs of process
  - Although impractical, useful for comparison

<table>
<thead>
<tr>
<th>1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5</th>
</tr>
</thead>
<tbody>
<tr>
<td>Need room for this one</td>
</tr>
<tr>
<td>Your ideal victim!</td>
</tr>
</tbody>
</table>

1, 1, 1, 1, 1, 1, 1, 1
Not recently used (NRU) algorithm

- Each page has *Reference* and *Modified* bits
  - Set when page is referenced, modified
  - R bit set means recently referenced, so you must clear it every now and then

- Pages are classified

<table>
<thead>
<tr>
<th>R</th>
<th>M</th>
<th>Class</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>Not referenced, not modified (0,0 → 0)</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>Not referenced, modified (0,1 → 1)</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>Referenced, but not modified (1,0 → 2)</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>Referenced and modified (1,1 → 3)</td>
</tr>
</tbody>
</table>

- NRU removes page at random
  - from lowest numbered, non-empty class

- Easy to understand, relatively efficient to implement and sort-of OK performance
FIFO algorithm

- Maintain a linked list of all pages – in order of arrival
- Victim is first page of list
  - Maybe the oldest page will not be used again …
- Disadvantage
  - But maybe it will – the fact is, you have no idea!
  - Increasing physical memory *might* increase page faults (Belady’s anomaly)

A, B, C, D, A, B, E, A, B, C, D, E
Second chance algorithm

- Simple modification of FIFO
  - Avoid throwing out a heavily used page – look at the R bit
- Operation of second chance
  - Pages sorted in FIFO order
  - Page list if fault occurs at time 20, A has R bit set (time is loading time)

<table>
<thead>
<tr>
<th>Page</th>
<th>Time</th>
<th>R</th>
</tr>
</thead>
<tbody>
<tr>
<td>H</td>
<td>18</td>
<td>X</td>
</tr>
<tr>
<td>G</td>
<td>15</td>
<td>X</td>
</tr>
<tr>
<td>F</td>
<td>14</td>
<td>X</td>
</tr>
<tr>
<td>E</td>
<td>12</td>
<td>X</td>
</tr>
<tr>
<td>D</td>
<td>8</td>
<td>X</td>
</tr>
<tr>
<td>C</td>
<td>7</td>
<td>X</td>
</tr>
<tr>
<td>B</td>
<td>3</td>
<td>0</td>
</tr>
</tbody>
</table>

Most recently loaded

<table>
<thead>
<tr>
<th>Page</th>
<th>Time</th>
<th>R</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>20</td>
<td>0</td>
</tr>
<tr>
<td>H</td>
<td>18</td>
<td>X</td>
</tr>
<tr>
<td>G</td>
<td>15</td>
<td>X</td>
</tr>
<tr>
<td>F</td>
<td>14</td>
<td>X</td>
</tr>
<tr>
<td>E</td>
<td>12</td>
<td>X</td>
</tr>
<tr>
<td>D</td>
<td>8</td>
<td>X</td>
</tr>
<tr>
<td>C</td>
<td>7</td>
<td>X</td>
</tr>
<tr>
<td>B</td>
<td>3</td>
<td>0</td>
</tr>
</tbody>
</table>

Oldest page
Clock algorithm

- Quit moving pages around – move a pointer?
- Same as Second chance but for implementation
  - When page fault
  - Look at page pointed at by hand
    - If R = 0, evict page
    - If R = 1, clear R & move hand

Evict this one!
Least recently used (LRU) algorithm

- Pages used recently will be used again soon
  - Throw out page unused for longest time
  - Idea: past experience is a decent predictor of future behavior
    - LRU looks at the past, Belady’s wants to look at the future
    - *how is LRU different from FIFO?*

- Must keep a linked list of pages
  - Most recently used at front, least at rear
  - Update this list every memory reference!!
  - Too expensive in memory bandwidth, algorithm execution time, etc

- Alternatively keep counter in page table entry
  - Choose page with lowest value counter
  - Periodically zero the counter
A second HW LRU implementation

- Use a matrix – \( n \) page frames – \( n \times n \) matrix
- Page \( k \) is reference
  - Set all bits of row \( k \) to 1
  - Set all bits of column \( k \) to 0
- Page of lowest row is LRU

0,1,2,3,2,1,0,3,2
Simulating LRU in software

- Not Frequently Used
  - Software counter associated with each page
  - At clock interrupt – add R to counter for each page
  - Problem - it never forgets!

- Better – Aging
  - Push R from the left, drop bit on the right
  - How is this *not* LRU? One bit per tick & a finite number of bits per counter
Most programs show locality of reference
  – Over a short time, just a few common pages

Working set
  – Models the dynamic locality of a process’ memory usage
  – i.e. the set of pages currently needed by a process

Definition
  – $\text{ws}(k, t) = \{\text{pages } p \text{ such that } p \text{ was referenced in the } k \text{ most recent memory references}\}$ ($k$ is WS window size)
  – What bounds $\text{ws}(k, t)$ as you increase $k$?

Clearly $\text{ws}(k_i, t) \leq \text{ws}(k_j, t)$ for $i < j$
Working set

- Demand paging
  - Simplest strategy, load page when needed
- Can you do better knowing a process WS?
  - How could you use this to reduce turnaround time?

Prepaging

- Intuitively, working set must be in memory, otherwise you’ll experience heavy faulting (thrashing)
  - What does it mean ‘how much memory does program x need?” – what is program x average/worst-case working set size?
- Working set sizes changes over time
Load control

- Despite good designs, system may still thrash
  - Sum of working sets > physical memory
- Page Fault Frequency (PFF) indicates that
  - Some processes need more memory
  - but no process needs less
- Way out: Swapping
  - So yes, even with paging you still need swapping
  - Reduce number of processes competing for memory
  - ~ two-level scheduling – careful with which process to swap out (there’s more than just paging to worry about!)

*What would you like of the remaining processes?*
Working set algorithm

- Working set and page replacement
  - Victim – a page *not* in the working set
- At each clock interrupt – scan the page table
  - R = 1? Write Current Virtual Time (CVT) into *Time of Last Use*
  - R = 0? CVT – *Time of Last Use > Threshold* ? out! else see if there’s someone and evict oldest (w/ R=0)
  - If all are in the working set (all R = 1) random

Current virtual time

<table>
<thead>
<tr>
<th>Time of last use</th>
<th>R bit</th>
</tr>
</thead>
<tbody>
<tr>
<td>1213</td>
<td>0</td>
</tr>
<tr>
<td>2014</td>
<td>1</td>
</tr>
<tr>
<td>2020</td>
<td>1</td>
</tr>
<tr>
<td>2032</td>
<td>1</td>
</tr>
<tr>
<td>1620</td>
<td>0</td>
</tr>
</tbody>
</table>

Information about a page
WSClock algorithm

- Problem with WS algorithm – Scans the whole table
- Combine clock & working set
  - If R = 1, same as working set
  - If R = 0, if age > T and page clean, out
  - If dirty, schedule write and check next one
  - If loop around,
    - There’s 1+ write scheduled – you’ll have a clean page soon
    - There’s none, pick any one

\[ R = 0 \& 2204 - 1213 > T \]
Cleaning policy

- To avoid having to write pages out when needed – paging daemon
  - Periodically inspects state of memory
  - Keep enough pages free
  - If we need the page before it’s overwritten – reclaim it!

- Two hands for better performance (BSD)
  - First one clears R, second checks it
  - If hands are kept close, only heavily used pages have a chance
  - If back is just ahead of front hand (359 degrees), original clock
  - Two key parameters, adjusted at runtime
    - Scanrate – rate at which hands move through the list
    - Handspread – gap between them
Design issues – global vs. local policy

- When you need a page frame, pick a victim from
  - Among your own resident pages – Local
  - Among all pages – Global

- Local algorithms
  - Basically every process gets a fixed % of memory

- Global algorithms
  - Dynamically allocate frames among processes
  - Better, especially if working set size changes at runtime
  - How many page frames per process?
    - Start with basic set & react to Page Fault Frequency (PFF)

- Most replacement algorithms can work both ways except for those based on working set
  Why not working set based algorithms?
Next time ...

- You now understand how things work, i.e. the mechanism ...
- We’ll now consider design & implementation issues for paging systems
  - Things you want/need to pay attention for good performance