CMSC 321: Operating Systems

Lecture 13

Virtual Memory
Think Of Nachos...

- Exec-ing a new process:
  - copy the executable into RAM
  - executable: split into 128B pages
  - RAM: "divided" into frames (128B * # pages)

  - need: page tables, frame table (bitmap)

- What happens if not enough free frames?
Paging/Segmentation

- Memory references translated from logical to physical @ runtime
  - MMU: Memory Management Unit
  - process may occupy different locations in RAM over its lifetime

- Process is broken into segments/pages

- What do these buy?
## Paging/Segmentation

- All pages/segments don’t need to be in RAM

- Sufficient if RAM contains next inst & data  
  - i.e., execution may proceed

- **Resident set**: portion of a process in RAM  
  - if all memory references are in resident set, OK
Before VM, programmer was responsible for dealing with situation where executable was larger than RAM by specifying overlays – parts of process image that could be loaded dynamically as required.

Virtual Memory

- Separation of logical from physical memory
  - process may not be contiguous in RAM
  - process may not be entirely in RAM

- Relieves constraints of (relatively) small RAM

- A process may be larger than all of RAM!
Virtual vs. Physical Memory
Page Faults

Page Fault: When a memory reference is not in the resident set

1. OS blocks the process, issues I/O request
2. another process selected to run
3. once I/O is finished, interrupt
4. original process moves to Ready state
### Implications of VM

- **more processes in RAM @ a time**
  - higher CPU utilization and throughput

- **not constrained by amount of physical RAM**
  - a process can be larger than all of RAM

- **less I/O needed to swap/load a process**
  - only need to load a portion

- **must avoid thrashing:**
  - i.e., spend more time swapping than executing
<table>
<thead>
<tr>
<th>Facilitating VM</th>
</tr>
</thead>
</table>

- **hardware**: support for paging/segmentation  
  - previous lectures  
  - page size, page tables, TLB, MMU, etc.

- **software**:  
  - for moving pages between RAM & disk

- **principle of locality**: program/data refs tend to cluster  
  - Temporal locality  
  - Spatial locality
### VM Software

- **Goal:** minimize page fault rate
  - want low probability of referencing a missing page
  - no single best policy

- **Overhead:**
  - deciding which pages to replace
  - actual I/O of replacement
Software Policy Issues

- **Fetch**: when should pages be brought into RAM?
- **Replacement**: which page should be replaced?
- **Resident Set Management**:
  - how many page frames per process?
  - local or global page replacement?
- **Cleaning**: when to write modified pages to disk?
- **Load Control**: how many processes in RAM?
## 1) Fetch Policy

- When should pages be brought into RAM?

- Two alternatives:
  - **demand paging**:
    - page into RAM only when a reference is made
  - **pre-paging**:
    - bring several pages in from disk at once
    - goal is to load in pages likely to be referenced
    - principle of locality: near on disk
Demand Paging

- When process starts, flurry of page faults
- As more pages are brought in, POL says page faults drop

- How can we implement paging?
  - valid/invalid bit with each page in page table
    - valid: address is legal and page in RAM
    - invalid: address not legal or page not in RAM
  - access to invalid page causes page fault
Valid/Invalid Bits in Page Table
### What Happens on Page Fault?

- Trap to OS
- Save user registers and process state
- Determine that interrupt was page fault
- Check that reference was legal; determine location on disk
- Issue a read from disk to a free frame
- While waiting, allocate CPU to another process
- Interrupt from disk (I/O completed)
- Save registers and process state of other process
- Determine that interrupt was from disk
- Correct page table to show page is now in RAM
- Wait for CPU to be allocated to this process again
- Restore user registers, process state, pg table; then resume interrupted instruction
Demand Paging

Because state is saved, process can resume exactly as it was.
2) Replacement Policy

- Which page should be replaced on page fault?

- Page *least likely* to be referenced soon

- Often high correlation in past behavior and future
  - based on principle of locality
  - try to predict future based on past

- One restriction: *frame locking*
  - certain frames are *locked* and cannot be swapped
  - e.g., parts of kernel, I/O buffers (temporary)
  - use locking bit in page table
Basic Replacement Approach

- Find location of desired page on disk

- Find a free frame in memory
  - if one free, use it
    - perhaps maintain a list of free frames
  - if none free:
    - select a “victim” and write to disk if necessary
    - change page and frame tables

- Read desired page into free frame

- Restart the faulting process
Page Replacement

1. Swap out victim page
2. Change to invalid
3. SSAP desired page in
4. Reset page table for new page
Page Replacement Algorithms

- How do we select a victim to evict?

- Want a “good” algorithm
- Why? Disk I/O is expensive

- Basic algorithms:
  - Optimal (an ideal, for comparison only)
  - FIFO
  - LRU (typically approximated)
  - Clock (AKA Second Chance)
Optimal Algorithm

- An ideal: impossible to implement
  - OS must know all future events
  - used as a standard for judging others

- Replace page whose next time to reference is longest

- Consider 3-page resident set
  - Page addr stream: 2 3 2 1 5 2 4 5 3 2 5 2
  - How many page faults?
## Optimal Algorithm

- **An ideal:** impossible to implement
  - OS must know *all* future events
  - used as a standard for judging others

- **Replace page whose next time to reference is longest**

- **Consider 3-page resident set**
  - Page addr stream: 2 3 2 1 5 2 4 5 3 2 5 2
  - 3 page faults
## Least Recently Used (LRU)

- Replace page that has not been referenced for the longest time
- Performs almost as well as optimal
- Difficult to implement: overhead
  - tag each page on every reference?
  - instead, try to approximate LRU (later)
- For previous example, # page faults?
### Least Recently Used (LRU)

- Replace page that has not been referenced for the longest time

- Performs almost as well as optimal

- Difficult to implement: overhead
  - tag each page on every reference?
  - instead, try to approximate LRU (later)

- For previous example, 4 page faults
FIFO

• Consider page frames as a circular buffer removed round robin style

• Easiest to implement
• Motivation: page brought in first may have fallen out of use (often wrong!)

• For previous example, # page faults?
<table>
<thead>
<tr>
<th>FIFO</th>
</tr>
</thead>
<tbody>
<tr>
<td>• Consider page frames as a circular buffer removed round robin style</td>
</tr>
<tr>
<td>• Easiest to implement</td>
</tr>
<tr>
<td>• Motivation: page brought in first may have fallen out of use (often wrong!)</td>
</tr>
<tr>
<td>• For previous example, 6 page faults</td>
</tr>
</tbody>
</table>
Clock Policy

- Approximates LRU

- For each frame, have \texttt{use} bit
  - when frame into RAM, \texttt{use} set to 1
  - when referenced, \texttt{use} set to 1

- Set of frames considered as a circular buffer
  - when page replaced, ptr moves to next in buffer

- Algorithm:
  - start @ ptr, scan until find \texttt{use} == 0
  - each \texttt{use} that is 1, set to 0

- Similar to FIFO, except pages get a 2nd chance
- For previous example, \# page faults?
Clock Policy

- Approximates LRU

- For each frame, have *use* bit
  - when frame into RAM, *use* set to 1
  - when referenced, *use* set to 1

- Set of frames considered as a circular buffer
  - when page replaced, ptr moves to next in buffer

- Algorithm:
  - start @ ptr, scan until find *use* == 0
  - each *use* that is 1, set to 0

- Similar to FIFO, except pages get a 2nd chance
- For previous example, 5 page faults
Improved Clock Policy

- For each frame, have two bits: **use** and **modify**

- Now four categories:
  - $u = 0$, $m = 0$: not recently used, not modified
  - ...
  - $u = 1$, $m = 1$: recently used, modified

- Algorithm:
  - scan buffer to find ($u = 0$, $m = 0$)
    - no changes to **use** bit
  - if fail, scan to find ($u = 0$, $m = 1$)
    - set $u$ to 0 on each bypassed
  - if fail, repeat step 1, and if necessary step 2

- Unchanged pages preferred for replacement
Page Buffering

- Modified pages: cost of evicting is greater

- Solution:
# Page Buffering

- **Modified pages:** cost of evicting is greater

- **Solution:** *free page list & modified page list*
  - try to keep some small # frames free
  - replacement algorithm is simple FIFO
  - when unmodified pg evicted, frame is added to tail of FPL
  - when modified pg to be replaced, added to tail of MPL

- **Note:** pages are not physically moved in RAM
  - use pointer manipulations

- **Page to be replaced remains in RAM**
  - FPL & MPL act as cache
  - modified pages written in clusters to save on I/O!
3) Resident Set Management

- Two main issues:
  - Allocation: How many frames to each process?
  - Replacement Scope: global or local?

- Effect of fewer pages per process?
  - more processes in RAM
  - higher page fault rate

- Beyond certain size, more pages won’t reduce page fault rate (POL)
## Allocation of Frames

- **Fixed:**
  - fixed # of frames to each process
  - can be different # per process
  - decided @ load time
    - if page fault, one of the allocated pages *must* be replaced

- **Variable:**
  - # of frames can vary over process lifetime
  - ideally, suffering process receives more
    (process below threshold can lose frames w/o affecting)
  - software overhead -- OS must assess active processes
Frame Replacement Scope

- Local replacement:
  - victim frame can only come from process that generated page fault

- Global replacement:
  - victim frame can come from any unlocked frame

- Note: global is easier to implement
Combining Allocation & Scope

- Fixed allocation / global replacement:
  - not possible

- Fixed allocation / local replacement:
  - processes in RAM have fixed # of frames
  - on fault, which frame from resident set goes?
  - if allocation too small --> high pg fault rate
  - if too large --> too few processes in RAM, considerable time swapping
Combining Allocation & Scope

- Variable allocation / global replacement:
  - easiest to implement
  - maintain set of free frames
  - when no free frames, choose from all unlocked
    - use page replacement algorithm from before; or
    - use page buffering

- Variable allocation / local replacement:
  - on load, allocate a certain # of frames
    - pre-paging or demand paging
  - on fault, select from among resident set
  - periodically evaluate allocation to the process
Variable Allocation / Local Replacement

- Decision to increase/decrease is deliberate
- Complex: assessing likely future demands

- Key elements of variable/local:
  - criteria to determine allocation
  - timing of change

- Use working set: $W(t, \Delta)$
  - at time $t$, the set of pages referenced in last $\Delta$
  - non-decreasing function of $\Delta$
    - larger $\Delta$ $\rightarrow$ larger working set
### Working Set Example

- **Sequence of page references:**
  
  24 15 18 23 24 17 18 24 18 17 17 15 24 17 24 18

<table>
<thead>
<tr>
<th>Sequence of Page References</th>
<th>Window Size, $\delta$</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>2</td>
</tr>
<tr>
<td>24</td>
<td>24</td>
</tr>
<tr>
<td>15</td>
<td>24 15</td>
</tr>
<tr>
<td>18</td>
<td>15 18</td>
</tr>
<tr>
<td>23</td>
<td>18 23</td>
</tr>
<tr>
<td>24</td>
<td>24 24</td>
</tr>
<tr>
<td>17</td>
<td>24 17</td>
</tr>
<tr>
<td>18</td>
<td>17 15</td>
</tr>
<tr>
<td>24</td>
<td>18 24</td>
</tr>
<tr>
<td>10</td>
<td>*</td>
</tr>
<tr>
<td>17</td>
<td>18 17</td>
</tr>
<tr>
<td>18</td>
<td>17 15</td>
</tr>
<tr>
<td>17</td>
<td>24 17</td>
</tr>
<tr>
<td>18</td>
<td>17 15</td>
</tr>
<tr>
<td>18</td>
<td>24 17</td>
</tr>
</tbody>
</table>
Working Set Strategy

- Use working set to manage resident set size:
  - monitor $W(t, \Delta)$ of each process
  - periodically remove from resident set any pages not in $W(t, \Delta)$
  - a process may execute only if $W(t, \Delta)$ in RAM

- Note: working set size transitions from stable to transient states across time
Working Set Strategy Problems

- Past does not necessarily predict future
  - size and membership of $W(t, \Delta)$ changes

- True measurement of $W(t, \Delta)$ impractical
  - requires many timestamps, queue of pages, etc.

- Optimal $\Delta$ is unknown, and would vary

- Instead, approximate:
  - consider page fault rate of a process
  - if below threshold, system benefits by reducing resident set size
  - if above, system benefits by increasing resident set size
4) Cleaning Policy

- When should pages be written to disk?

  - **Demand cleaning**: write out when evicted
  - **Pre-cleaning**: write out pages before frames are needed
    - permits writing in batches

- Careful of extremes:
  - prc cleaning:
    - demand cleaning:
### 4) Cleaning Policy

- **When should pages be written to disk?**

  - **Demand cleaning**: write out when evicted
  - **Pre-cleaning**: write out pages before frames are needed
    - permits writing in batches

- **Careful of extremes:**
  - **prc cleaning**: may write out pages that are modified again before being replaced
  - **demand cleaning**: pg faulting process must wait for two page transfers

- **Use page buffering**: modified, unmodified page lists
5) Load Control

- i.e., level of multiprogramming
- if too few processes -->
- if too many processes -->
5) Load Control

- i.e., level of multiprogramming
- if too few processes --> lots of swapping
  - e.g., if all processes in memory are blocked
- if too many processes --> lots of page faults
  - thrashing -- must reduce level of multiprogramming
Reducing Level of Multiprogramming

Which process should be suspended?

- lowest priority
- faulting process
  - blocks process about to be blocked anyway
- last process activated
  - least likely to have its working set resident
- one with smallest resident set
  - least effort to reload
- largest process
  - gives most free frames
- largest remaining execution window
  - approximates Shortest Processing Time First (SPTF)