CSE451 Paging and More VM
Spring 2001

Gary Kimura
Lecture #16
April 30, 2001

Today

- This week
- Work though a paging example
- Illustrate how the Operating System can actually utilize the hardware to manage Memory
- Tie together Virtual Memory pages, Physical Frames, PTE/TLB, backing store, and some of the software organization needed
- Lead into various paging strategies

Your Job This Week

- Readings in Silberschatz
  - Finish chapter 9 and start chapter 10
- Homework #5
  - Out today Monday April 30, 2001
  - Due next Monday May 7, 2001
  - Silberschatz questions 9.1, 9.5, and 9.20

Some Basic Structure For Our Example

- Physical memory
  - Divided into equal sized (4KB) frames (256 frames per 1MB of memory, so 64MB equals 16,384 frames)
  - MM keeps various lists of frames
- Virtual address space (32 bits translates to 1 million pages)
  - The VA is divided into ranges of valid and invalid sections
  - A processes VA is full of holes
  - MM keeps track of which ranges are valid
  - The granularity is 4KB (so sections are 4KB, 8KB, 12KB, etc, in size)
- Backing store
  - Each allocated section of memory is assigned a range or section in a file on disk
  - When not in physical memory the pages of a section are stored on disk

Allocating Virtual Memory

- Suppose a program calls into the system to allocate memory. What does that system do?
  - It creates a section in the VA for the allocation
    - Wise VA space management is needed to avoid external fragmentation
  - It finds disk space to support the allocation (e.g., a paging file)
    - Easy lookup and fragmentation is an issue
  - It modifies the PTE to still be invalid but puts information it can use to locate the page within the paging file

Page Faults

- Suppose the program now writes into the newly allocated memory. What happens?
  - There is a hardware fault because the PTE for the allocated memory still shows the page as invalid
  - The system uses the information stored in the invalid PTE to locate the page on disk
  - The system finds a free frame (how it does this we will talk about later)
  - The system reads in the page from disk into the new frame
  - When the I/O is done the system will validate the PTE and restart the process

Page Fault Variations

- A “hard page fault” is when the operating system must go out to backing store to get the page into memory
- A “soft page fault” is when the operating system can materialize the page (i.e., correct the fault) without reading from backing store
  - Copy on write pages can be done this way
  - Replacement strategies can use this to track which page are actually being used
  - It is possible to keep track of where free pages came from and so they can be added back to a processes PTE if necessary

Some of the Issues With Paging

- Hard page faults are very expensive and too many of them cause the system to slow to a crawl
- If there were an endless supply of free frames then paging would be really simple only at most one hard fault per page ever
- However when a system runs out of free frames then something in use must be removed from physical
- The goal in picking the pages to replace is to reduce the number of hard faults.
- Likewise schemes such as pre-fetching also can reduce the number of hard faults

All Paging Schemes Depend on Locality

- Processes tend to reference pages in localized patterns. That is over a small window of time a process will typically only access a few of its many valid pages.
- So the goal is to keep those few pages in memory because all of the pages cannot fit, but which pages are they?
- Temporal locality property
  - Recently reference variable are more likely to be referenced again
- Spatial locality property
  - Variables often come in clumps
- Both properties are true for both data and code
- The goal of a paging system is
  - Stay out of the way when there is plenty of available memory
  - Don't bring the system to its knees when there is not

A Side Note for Programmers

- There is somewhat of a distorted cause and affect factor here
- As a programmer knowing how the system pages will dictate how you want to layout our data structures and organize your code
- As a OS developer knowing how programmers layout their data structures and organize their code helps dictate who to page the system

Demand Paging

- *Demand Paging* refers to a technique where a processes pages are loaded from disk into memory as they are referenced. The earlier page fault slides is an example of demand paging.
- Each reference to a page not previously touched causes a page fault.
- The fault occurs because the hardware found a page table entry with its valid bit off.
- As a result of the page fault, the OS allocates a new page frame and reads the faulted page from the disk.
- When the I/O completes, the OS fills in the PTE, sets its valid bit, and restarts the faulting process.

How Much to Page in at a Time

- The good and bad points with demand paging are
  - It doesn't load a page until it is absolutely necessary thus keeping physical memory usage to a minimum
  - Its biggest drawback is that reading only one page at a time is slower than batching them
- Prepaging
  - Is where the system tries to anticipate fault before they happen
  - For example, linear code can be prefetched
  - The downside with prepaging is that it is hard to predict the future
  - Some simple schemes (hints from programmer or program behavior) can work. The Windows NT system tries to keep track of memory access to predict strides, etc, but not always with the greatest success.

Page Replacement

- When there are no available free frames to handle a fault we must find a page to replace.
- How it does this is determined by the *page replacement algorithm*.
- The goal of the replacement algorithm is to reduce the fault rate by selecting the best victim page to remove.

Finding the Best Page

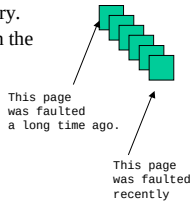
- A good property
  - If you put more memory on the machine, then your page fault rate will go down.
  - Increasing the size of the resource pool helps everyone.
- The best page to toss out is the one you'll never need again
  - That way, no faults.
- Never is a long time, so picking the one closest to "never" is the next best thing.
  - Replacing the page that won't be used for the longest period of time absolutely minimizes the number of page faults.

Three Replacement Algorithms

- There are essentially three replacement algorithms to compare.
- The Optimal algorithm is the theoretical best. It is one that always chooses and replaces the page that will lead to the lowest fault rate.
  - This algorithm is not practical but is used as a yardstick for the other algorithms
- The FIFO algorithm is where the oldest faulted in page is the page to replace
- The LRU algorithm is where the least recently accessed page is the page to replace
- Some system use a combination of FIFO and LRU

FIFO

- FIFO is an obvious algorithm and simple to implement.
- Basic idea, maintain a list or queue of pages in the order in which they were paged into memory.
- On replacement, remove the one brought in the longest time ago.
- Why might it work?
  - Maybe the one brought in the longest ago is one we're not using now.
- Why it might not work?
  - Maybe it's not.
  - We have no real information to tell us if it's being used or not.
- FIFO suffers from "Belady's anomaly"
  - the fault rate might actually increase when the algorithm is given more memory -- a bad property.



An Example of Optimal and FIFO in Action

Reference stream is A B C A B D A D B C

OPTIMAL

A B C A B D A D B C B

5 Faults — toss C — toss A or D

A
B
C
D
A
B
C

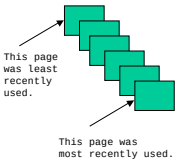
FIFO

A B C A B D A D B C B

7 Faults — toss A — toss ?

Least Recently Used (LRU)

- Basic idea: we can't look into the future, but let's look at past experience to make a good guess.
- LRU: on replacement, remove the page that has not been used for the longest time in the past.
- Implementation: to really implement this, we would need to time stamp every reference, or maintain a stack that's updated on every reference. This would be too costly.
- We can't implement this exactly, but we can try to approximate it.



Our "bet" is that pages which you used recently are ones which you will use again (principle of locality) and, by implication, those that you didn't, you won't.

Approximating LRU

- A poor man's LRU can be implemented using the Reference Bit in the PTE
- Essentially we keep a counter for every page and at regular intervals we do the following work:

```
for every page
  if the ref bit == 0 then increment the
  counter
  if the ref bit == 1 then zero the
  counter
  reset the reference bit
```

- The counter contains the number of intervals since the last reference to the page.
  - The page with the largest counter will be least recently used one.
- If we don't have a reference bit, we can simulate it using the VALID bit and taking a soft fault.

Fixed Space Vs. Variable Space

- In a multiprocessing system there are essentially two ways to divide up the frames.
- Local Replacement: each process "own" a fixes number of frames.
  - Frames for the process comes from this pool.
  - This is very easy to implement
  - But deciding on the right size is hard because processes each have individual needs that can change dynamically as the program runs
- Global Replacement: the number of frames for each process to shrink and grow over time.
  - It is not always easy to predict or know what the number should be
  - But multiprocessing with the system typically have the best performance

Mixing FIFO and LRU

- Some systems use a mixture of FIFO and LRU by mixing local and global replacement
  - Local replacement is LRU
  - Global replacement is FIFO
- Try for the best of both worlds
- Each process has a fixed number of local frames but instead of getting initially swapped against disk storage the pages are swapped from the global frame pool

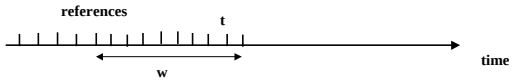
Working Set Model

- Peter Denning defined the working set of a program as a way to model the dynamic *locality* of a program in execution.
- Definition:

  WS(t,w) = {pages i s.t. i was referenced in the interval (t,t-w)}

  t is a time, w is the working set window, a backward looking interval,measured in *references*.
- So, a page is in the WS only if it was referenced in the last w references.

Working Set Size



- The working set size is the *number* of pages in the working set; i.e., the number of pages touched in the interval (t, t-w).
- The working set size changes with program locality.
  - during periods of poor locality, you reference more pages.
  - so, within that period of time, you will have a larger working set size.
- For some parameter w, we could keep the working sets of each process in memory.
- Don't run process unless working set is in memory.

Working Set Continued

- But, we have two problems:
  - **how do we select w?**
  - **how do we know when the working set changes?**
- So, working set is not used in practice.
- VAX/VMS and NT use the phrase "working set" in their documentation. It is NOT the same as the workings sets discussed in OS design and academia

Next Time

- Some important implementation Issues
- How does the free page list get replenished?
- What do we do with dirty pages?
- Have to avoid trashing
- How can caching interact with MM
- What about allocating memory in smaller units than a page
- What parts of the operating system always needs to be resident in physical memory?