Virtual Memory Design Issues:
Paging and Caching

Jaswinder Pal Singh and a Fabulous Course Staff
Computer Science Department
Princeton University

(http://www.cs.princeton.edu/courses/cos318/)

Virtual Memory: Paging and Caching

- Need mechanisms for paging between memory and disk
- Need algorithms for managing physical memory as a cache

Today’s Topics
- Paging mechanism
- Page replacement algorithms
- When the cache doesn’t work

Virtual Memory Paging

- Simple world
  - Load entire process into memory. Run it. Exit.
- Problems
  - Slow (especially with big processes)
  - Wasteful of space (doesn’t use all of its memory all the time)
- Solution
  - Demand paging: only bring in pages actually used
  - Paging: goal is only keep frequently used pages in memory
- Mechanism
  - Virtual memory maps some to physical pages, some to disk
VM Paging Steps

- Memory reference (may cause a TLB miss)
- TLB entry invalid triggers a page fault and VM handler takes over
- Move page from disk to memory
- Update TLB entry w/ pp#, valid bit
- Restart the instruction
- Memory reference again

Virtual Memory Issues

- What to page in?
  - Just the faulting page or more?
  - Want to know the future...
- What to replace?
  - Cache (main memory) too small. Which page to replace?
  - Want to know the future...

How Does Page Fault Work?

- User program should not be aware of the page fault
- Fault may have happened in the middle of the instruction!
- Can we skip the faulting instruction?
- Is a faulting instruction always restartable?

What to Page In?

- Page in the faulting page
  - Simplest, but each "page in" has substantial overhead
- Page in more pages each time (prefetch)
  - May reduce page faults if the additional pages are used
  - Waste space and time if they are not used
  - Real systems do some kind of prefetching
- Applications control what to page in
  - Some systems support for user-controlled prefetching
  - But, applications do not always know
VM Page Replacement

- Things are not always available when you want them
  - It is possible that no unused page frame is available
  - VM needs to do page replacement

- On a page fault
  - If there is an unused frame, get it
  - If no unused page frame available,
    - Choose a used page frame
    - If it has been modified, write it to disk*
    - Invalidate its current PTE and TLB entry
  - Load the new page from disk
  - Update the faulting PTE and remove its TLB entry
  - Restart the faulting instruction

* If page to be replaced is shared, find all page table entries that refer to it

Backs Store

- Swap space
  - When process is created, allocate swap space for it on disk
  - Need to load or copy executables to swap space
  - Need to consider swap space growth

- Can you use the executable file as swap space?

Bookkeeping Bits Used by VM Methods

- Has page been modified?
  - “Dirty” or “Modified” bit set by hardware on store instruction
  - In both TLB and page table entry

- Has page been recently used?
  - “Referenced” bit set by hardware in PTE on every TLB miss
  - Can be cleared every now and then, e.g. on timer interrupt

Cache replacement policy

- On a cache miss, how do we choose which entry to replace?
  - Assuming the new entry is more likely to be used in the near future
  - In direct mapped caches, not an issue

- Policy goal: reduce cache misses
  - Improve expected case performance
  - Also: reduce likelihood of very poor performance
Which “Used” Page Frame To Replace?

- Random
- Optimal or MIN algorithm
- NRU (Not Recently Used)
- FIFO (First-In-First-Out)
- FIFO with second chance
- Clock (with second chance)
- Not Recently Used
- LRU (Least Recently Used)
- NFU (Not Frequently Used)
- Aging (approximate LRU)
- Working Set
- WSClock

Optimal or MIN

- Algorithm:
  - Replace the page that won’t be used for the longest time (Know all references in the future)
- Example
  - Reference string: 1 2 3 4 1 2 5 1 2 3 4 5
  - 4 page frames
  - 6 faults
- Pros
  - Optimal solution and can be used as an off-line analysis method
- Cons
  - No on-line implementation

Revisit TLB and Page Table

- Important bits for paging
  - Reference: Set when referencing a location in the page (really on TLB miss; can clear every so often, e.g. on clock interrupt)
  - Modify: Set when writing to a location in the page

Not Recently Used (NRU)

- Algorithm
  - Randomly pick a page from one of the following sets (in this order)
    - Not referenced and not modified
    - Not referenced and modified
    - Referenced and not modified
    - Referenced and modified
  - Clear reference bits
- Example
  - Reference string: 1 2 3 4 1 2 5 1 2 3 4 5
  - 8 page faults
- Pros
  - Implementable
- Cons
  - Require scanning through reference bits and modified bits
First-In-First-Out (FIFO)

- Algorithm
  - Throw out the oldest page
- Example
  - 4 page frames
  - Reference string: 1234 1 2 512345
- Pros
  - Low-overhead implementation
- Cons
  - May replace the heavily used pages (time a page first came in to memory may not be that indicative of its usage)
  - Worst case is program striding through data larger than memory

More Frames → Fewer Page Faults?

- Consider the following with 4 page frames
  - Algorithm: FIFO replacement
  - Reference string: 1234 1 2 512345
  - 10 page faults

- Same string with 3 page frames
  - Algorithm: FIFO replacement
  - Reference string: 1234125 1 2345
  - 9 page faults!

- This is so called “Belady’s anomaly” (Belady, Nelson, Shedler 1969)

FIFO with 2nd Chance

- Address the problem with FIFO
  - Check the reference-bit of the oldest page
  - If it is 0, then replace it (write back if dirty, don’t if clean)
  - If it is 1, clear the reference bit, put the page to the end of the list, updating its “load time” to the current time, and continue searching
  - Looking for an old page not referenced in current clock interval
  - If don’t find one (all pages referenced in current interval) come back to first-checked page again (its R bit is now 0). Degenerates to pure FIFO.
- Example
  - 4 page frames
  - Reference string: 1234 1 2 512345
  - 8 page faults
- Pros
  - Simple to implement
- Cons
  - The worst case may take a long time

Clock

- FIFO Clock algorithm
  - Arrange physical pages in circle
  - Clock hand points to the oldest page
  - On a page fault, follow the hand to inspect pages
- Clock with Second Chance
  - If the reference bit is 1, set it to 0 and advance the hand
  - If the reference bit is 0, use it for replacement
- Compare with FIFO w/2nd chance
  - What’s the difference?
- What if memory is very large
  - Take a long time to go around?
N\textsuperscript{th} chance: Not Recently Used

- Instead of one referenced bit per page, keep an integer
  - notInUseSince: number of sweeps since last use

- Periodically sweep through all page frames

```java
if (page is used) {
    notInUseSince = 0;
} else if (notInUseSince < N) {
    notInUseSince++;
} else {
    replace page;
}
```

Implementation note

- Clock and N\textsuperscript{th} Chance can run synchronously
  - In page fault handler, run algorithm to find next page to evict
  - Might require writing changes back to disk first

- Or asynchronously
  - A thread maintains a pool of recently unused, clean pages
  - Find recently unused dirty pages, write mods back to disk
  - Find recently unused clean pages, mark invalid and move to pool
  - On page fault, check if requested page is in pool
  - If not, evict that page

Least Recently Used

- Algorithm
  - Replace page that hasn’t been used for the longest time
    - Order the pages by time of reference
    - Needs a timestamp for every referenced page

- Example
  - 4 page frames
  - Reference string: 1 2 3 4 1 2 1 1 5
  - 8 page faults

- Pros
  - Good to approximate MIN

- Cons
  - Difficult to implement

Approximation of LRU

- Use CPU ticks
  - For each memory reference, store the ticks in its PTE
  - Find the page with minimal ticks value to replace

- Use a smaller counter

<table>
<thead>
<tr>
<th>Least Recently Used</th>
<th>N categories</th>
</tr>
</thead>
<tbody>
<tr>
<td>LRU</td>
<td></td>
</tr>
<tr>
<td>Pages in order of last reference</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Crude LRU</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pages referenced since the last page fault</td>
</tr>
<tr>
<td>Pages not referenced since the last page fault</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>8-bit count</th>
<th>256 categories</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>2563255</td>
</tr>
</tbody>
</table>
Not Frequently Used (NFU)

- Software counter associated with every page
- Algorithm
  - At every clock interrupt, scan all pages, and for each page add the R bit value to its counter
  - At page fault, pick the page with the smallest counter to replace
- Problem
  - Never forgets anything: pages used a lot in the past will have higher counter values than pages used recently

Not Frequently Used (NFU) with Aging

- Algorithm
  - At every clock interrupt, shift (right) reference bits into counters
  - At page fault, pick the page with the smallest counter to replace
- Old example
  - 4 page frames
  - Reference string: \[1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5\]
- Main difference between NFU and LRU?
  - NFU has a short history (counter length)
  - NFU cannot distinguish reference times within a clock interval
- How many bits are enough?
  - In practice 8 bits are quite good (8*20ms is a lot of history)

Program Behavior (Denning 1968)

- 80/20 rule
  - > 80% memory references are within <20% of memory space
  - > 80% memory references are made by < 20% of code
- Spatial locality
  - Neighbors are likely to be accessed
- Temporal locality
  - The same page is likely to be accessed again in the near future

Working Set

- Main idea (Denning 1968, 1970)
  - Define a working set as the set of pages in the most recent K page references
  - Keep the working set in memory will reduce page faults significantly
- Approximate working set
  - The set of pages of a process used in the last T seconds
- An algorithm
  - On a page fault, scan through all pages of the process
  - If the reference bit is 1, record the current time as “time of last use” for the page
  - If the reference bit is 0, check the “time of last use,”
    - If the page has not been used within T, replace the page
    - Otherwise, go to the next
  - If all pages used within T, pick the oldest page that has R=0.
    Else if no R=0 pages then pick at random.
**WSClock**

- Follow the clock hand
- If the reference bit is 1
  - Set reference bit to 0
  - Set the current time for the page
  - Advance the clock hand
- If the reference bit is 0, check “time of last use”
  - If the page has been used within δ, go to the next
  - If the page has not been used within δ and modify bit is 1
    - Schedule the page for page out and go to the next
  - If the page has not been used within δ and modify bit is 0
    - Replace this page

**Replacement Algorithms**

- The algorithms
  - Random
  - Optimal or MIN algorithm
  - NRU (Not Recently Used)
  - FIFO (First-In-First-Out)
  - FIFO with second chance
  - Clock (with second chance)
- Not Recently Used
- LRU (Least Recently Used)
- NFU (Not Frequently Used)
- Aging (approximate LRU)
- Working Set
- WSClock

**Which are your top two?**

- Not Recently Used
- LRU (Least Recently Used)
- NFU (Not Frequently Used)
- Aging (approximate LRU)
- Working Set
- WSClock

**Thrashing**

- Thrashing
  - Paging in and out all the time, I/O devices fully utilized
  - Processes block, waiting for pages to be fetched from disk
- Reasons
  - Process requires more physical memory than it has
  - Process does not reuse memory well
  - Process reuses memory, but what it needs does not fit
  - Too many processes, even though they individually fit
- Solution: working set
  - Pages referenced (by a process, or by all) in last T seconds
  - Really, the pages that need to cached to get good hit rate

**Making the Best of a Bad Situation**

- Single process thrashing?
  - If process does not fit or does not reuse memory, OS can do nothing except contain damage.
- System thrashing?
  - If thrashing because of the sum of several processes, adapt:
    - Figure out how much memory each process needs
    - Change scheduling priorities to run processes in groups whose memory needs can be satisfied (shedding load)
    - If new processes try to start, can refuse (admission control)
Fitting Working Set in Memory

- Maintain two groups of processes
  - Active: working set loaded
  - Inactive: working set intentionally not loaded
- Two schedulers
  - A short-term scheduler schedules active processes
  - A long-term scheduler decides which are active and which inactive, such that (combined) active working sets fit in memory
- A key design point
  - How to decide which processes should be inactive
  - Typical method is to use a threshold on waiting time

Working Set: Global vs. Local Page Allocation

- The simplest is global allocation only
  - Pros: Pool sizes are adaptable
  - Cons: Too adaptable, little isolation (example?)
- A balanced allocation strategy
  - Each process has its own pool of pages
  - Paging allocates from its own pool and replaces from its own working set
  - Use a “slow” mechanism to change the allocations to each pool while providing isolation
- Design questions:
  - What is “slow?”
  - How big is each pool?
  - When to migrate?

What about Using Memory for I/O?

- Explicit read/write system calls
  - Data copied to user process using system call
  - Application operates on data
  - Data copied back to kernel using system call
- Memory-mapped files
  - Open file as a memory segment
  - Program uses load/store instructions on segment memory, implicitly operating on the file
  - Page fault if portion of file is not yet in memory
  - Kernel brings missing blocks into memory, restarts process

Advantages to memory-mapped Files

- Programming simplicity
- Efficient for large files
  - Operate directly on file, instead of copy in/copy out
- Zero-copy I/O
  - Data brought from disk directly into page frame. No copies in kernel
- Pipelining
  - Process can start working before all the pages are populated
- Inter-process communication
  - Shared memory segment vs. temporary file
Memory-mapped Files and Demand-Paged VM

- Can go further in unifying memory management across file buffer and process memory
- Every process segment is backed by a file on disk
  - Code segment -> code portion of executable
  - Data, heap, stack segments -> temp files
  - Shared libraries -> code file and temp data file
  - Memory-mapped file segments -> memory-mapped files
- When process ends, delete temp files

Address Space in Unix

- Stack
- Data
  - Un-initialized: BSS (Block Started by Symbol)
  - Initialized
  - brk(addr) to grow or shrink
- Text: read-only
- Mapped files
  - Map a file in memory
  - mmap(addr, len, prot, flags, fd, offset)
  - unmap(addr, len)

Virtual Memory in BSD4

- Physical memory partition
  - Core map (pinned): everything about page frames
  - Kernel (pinned): the rest of the kernel memory
  - Frames: for user processes
- Page replacement
  - Run page daemon until there are enough free pages
  - Early BSD used the basic Clock (FIFO with 2nd chance)
  - Later BSD used Two-handed Clock algorithm
    - Second hand runs ahead, writing dirty pages back so there are enough clean pages
    - Swapper runs if page daemon can’t get enough free pages
      - Looks for processes idling for 20 seconds or more
      - Check when a process should be swapped in

Virtual Memory in Linux

- Linux address space for 32-bit machines
  - 3GB user space, 1GB kernel (invisible at user level)
- Backing store
  - Text segment uses executable binary file as backing storage
  - Other segments get backing storage on demand
- Copy-on-write for forking processes
- Multi-level paging
  - Directory, middle (nil for Pentium), page, offset
  - Kernel is pinned
- Replacement
  - Keep certain number of pages free
  - Clock algorithm on paging cache and file buffer cache
  - Clock algorithm on unused shared pages
  - Modified Clock on memory of user processes
Address Space in Windows 2K/XP

- Win2k user address space
  - Upper 2GB for kernel (shared)
  - Lower 2GB – 256MB are for user code and data (Advanced server uses 3GB instead)
  - The 256MB contains system data (counters and stats) for user to read
  - 64KB guard at both ends

- Virtual pages
  - Page size
    - 4KB for x86
    - 8 or 16KB for IA64
  - States
    - Free: not in use and cause a fault
    - Committed: mapped and in use
    - Reserved: not mapped but allocated

Backing Store in Windows 2K/XP

- Backing store allocation
  - Win2k delays backing store page assignments until paging out
  - There are up to 16 paging files, each with initial and max sizes

- Memory mapped files
  - Delayed write back
  - Multiple processes can share mapped files w/ different accesses
  - Implement copy-on-write

Paging in Windows 2K/XP

- Each process has a working set with
  - Min size with initial value of 20-50 pages
  - Max size with initial value of 45-345 pages

- On a page fault
  - If working set < min, add a page to the working set
  - If working set > max, replace a page from the working set

- If a process has a lot of paging activities, increase its max

- Working set manager maintains a large number of free pages
  - In the order of process size and idle time
  - Otherwise, page out the pages with highest “non-reference” counters in a working set for uniprocessors
  - Page out the oldest pages in a working set for multiprocessors

Summary

- VM paging
  - Page fault handler
  - What to page in
  - What to page out

- LRU is good but difficult to implement

- Clock (FIFO with 2nd hand) is considered a good practical solution

- Working set concept is important