CSE 120
Principles of Operating Systems

Fall 2021

Lecture 13: Page Replacement, Storage Devices
Yiying Zhang
Announcements

• Project 2 due this Friday, no more extension!

• Project 3 announced
  ♦ You should start as soon as you can if you want to work on it
  ♦ You can change team for PR3
  ♦ We’ll take the better weighted grade of the two grading options if you choose to do PR3
  ♦ Fair warning that PR3 will be time consuming!

• Last two problems cut in HW3
We started this topic with the high-level problem of translating virtual addresses into physical addresses.

We’ve covered all of the pieces:
- Virtual and physical addresses
- Virtual pages and physical page frames
- Multi-level page tables and page table entries (PTEs)
- TLBs
- Demand paging

Now let’s put it together, bottom to top.
The Common Case

- The compiler compiles source code into binaries (containing memory instructions)
- OS loads the executable (a.out) into memory and starts its execution

- Process is executing on the CPU, and it issues a read to an address
  - What kind of address is it, virtual or physical?
- The read goes to the TLB in the MMU
  1. TLB does a lookup using the page number of the address
  2. Common case is that the page number matches, returning the physical page frame and protection bits for this address
  3. TLB validates that the protection bits allows reads (in this example)
  4. MMU combines the PFN and offset into a physical address
  5. MMU then reads from that physical address, returns value to CPU

- Note: The above execution is all done by the hardware
TLB Misses

- At this point, two other things can happen
  1. TLB does not have this virtual address
  2. Mapping in TLB, but memory access violates protection bits or the invalid bit is set
- We’ll consider each in turn
Reloading the TLB

If the TLB does not have mapping, two possibilities:

1. MMU loads PTE from page table in memory (a page table walk)
   » Hardware managed TLB, OS not involved in this step
2. Trap to the OS
   » Software managed TLB, OS intervenes at this point

A machine will only support one method or the other (all modern computers have hardware-managed TLB)

When TLB has PTE, it restarts translation

Common case is that the PTE refers to a valid page in memory
» Hardware just reads PTE from the page table and loads it into TLB

Uncommon case is that TLB faults again on PTE because of PTE protection/valid bits (e.g., page is invalid (not in memory))
» Becomes a page fault...
[lec12] Page Faults

- PTE can indicate the type of a page fault
  - Read/write/execute – operation not permitted on page
  - Invalid – page not in physical memory
- TLB traps to the OS (software takes over)
  - R/W/E – OS usually will send fault back up to user process, or use for other purposes (e.g., copy on write)
  - Invalid
    - Page not in physical memory because this is the first access
      - OS allocates physical frame and sets up the PTE (and flush TLB)
    - Page not in physical memory because it has been swapped out
      - Finds an empty frame in physical memory (if none, need to swap out something first), reads the page from disk, sets up the PTE to point to the new physical frame (and flush TLB)
Who calls `malloc`?
What happens at `malloc` time?

What is `brk`?
Who calls `brk`?
What happens at `brk` time?

When is physical memory allocated?
[lec12] malloc and brk / mmap

Application

Allocator (libc)
1. malloc()
   free()
   realloc()
   calloc()

Virtual Memory

Heap

Mappings

Physical Memory

Process Address Space

MMU

look up

2. brk()

3. mmap()
   munmap()

4. page fault
Memory Management

The real final lecture on memory management:

• Goals of memory management
  ♦ To provide a convenient abstraction for programming
  ♦ To allocate scarce memory resources among competing processes to maximize performance with minimal overhead

• Mechanisms
  ♦ Physical and virtual addressing
  ♦ Techniques: Partitioning, paging, segmentation
  ♦ Page table management, TLBs
  ♦ Memory allocation

• Policies
  ♦ Page replacement algorithms
Locality

• All paging schemes depend on locality
  ♦ Processes reference pages in localized patterns

• Temporal locality
  ♦ Locations referenced recently likely to be referenced again

• Spatial locality
  ♦ Locations near recently referenced locations are likely to be referenced soon

• Although the cost of paging is high, if it is infrequent enough that it is acceptable
  ♦ Processes usually exhibit both kinds of locality during their execution, making paging practical
The BIG picture: Running at Memory Capacity

- Expect to run with all phy. pages in use
- Every demand paging request (e.g., swap-in, new phys page allocation) requires an eviction
- Goal of page replacement
  - Maximize hit rate → kick out the page that’s least useful
- Challenge: how do we determine utility?
  - Kick out pages that aren’t likely to be used again

- Page replacement is a difficult policy problem
Performance metric for page replacement policies

• Give a sequence of memory accesses, minimize the # of page faults
  ♦ Similar to cache miss rate
  ♦ What about hit latency and miss latency?

• The best page to evict is the one never touched again
  ♦ Will never fault on it

• Never is a long time, so picking the page closest to “never” is the next best thing
  ♦ Evicting the page that won’t be used for the longest period of time minimizes the number of page faults
What makes finding the least useful page hard?

- Don’t know future!

- Past behavior is a good indication of future behavior! (e.g. LRU)
  » temporal locality → kick out pages that have not been used recently

- Perfect (past) reference stream hard to get
  ♦ Every memory access would need bookkeeping
  ♦ Is this feasible (in software? In hardware?)

- Minimize overhead
  ♦ If no memory pressure, ideally no bookkeeping
  ♦ In other words, make the common case fast (page hit)

⇒ Get imperfect information, while guaranteeing foreground perf
  ♦ What is minimum hardware support that need to added?
What can we do without extra hardware support?
First-In-First-Out (FIFO)

- **Algorithm**
  - Maintain a list of pages in order in which they were paged in
  - On replacement, evict the one brought in longest time ago

- **Why might this be good?**
  - Maybe the one brought in the longest ago is not being used
  - Low-overhead implementation

- **Cons**
  - No frequency/no recency → may replace the heavily used pages

- **FIFO suffers from “Belady’s Anomaly”**
  - The fault rate might actually **increase** when the algorithm is given more memory (**very bad**), see backup slides for an example
Predicting future based on past

• “Principle of locality”
  ♦ Recency:
    » Page recently used are likely to be used again in the near future
  ♦ Frequency:
    » Pages frequently used (recently) are likely to be used frequently again in the near future

• Is this temporal or spatial locality?

• The Working Set of a process: the set of memory that is referenced in the current time window. WSS (working set size): size of a working set. (more in backup slides)
  ♦ Goal: want to fit working sets of processes in main memory
Least Recently Used (LRU)

• LRU uses reference information to make a more informed replacement decision
  ♦ Idea: We can’t predict the future, but we can make a guess based upon past experience
  ♦ On replacement, evict the page that has not been used for the longest time in the past
  ♦ When does LRU do well? When does LRU do poorly?

• Implementation
  ♦ To be perfect, need to time stamp every reference (or maintain a stack) – much too costly
  ♦ So we need to approximate it
Exploiting locality needs some hardware support

• Reference bit
  • A hardware bit that is set whenever the page is referenced (read or written)

• Why not in software?
### x86 Page Table Entry

<table>
<thead>
<tr>
<th>Page frame number</th>
<th>U</th>
<th>P</th>
<th>Cw</th>
<th>Gl</th>
<th>L</th>
<th>D</th>
<th>A</th>
<th>Cd</th>
<th>Wt</th>
<th>O</th>
<th>W</th>
<th>V</th>
</tr>
</thead>
<tbody>
<tr>
<td>31</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- **Reserved**: 12 bits
- **Valid (present)**
- **Read/write**
- **Owner (user/kernel)**
- **Write-through**
- **Cache disabled**
- **Accessed (referenced)**
- **Dirty**
- **PDE maps 4MB**
- **Global**
LRU Clock  
(Not Recently Used)

- Clock algorithm – Used by Unix
- Idea: Replace page that is “old enough”
- Arrange all of physical page frames in a big circle (clock)
- A clock hand is used to select a good LRU candidate
  - Sweep through the pages in circular order like a clock
  - If the ref bit is off, it hasn’t been used recently
    » Pick it for page replacement (victim page)
    » What is the minimum “age” if ref bit is off?
  - If the ref bit is on, turn it off and go to next page. (why turn off?)
- Low overhead when plenty of memory
What happens if all reference bits are 1?

If memory is large, “accuracy” of information degrades
  ♦ What does it degrade to?

What does it suggest if observing clock hand is sweeping very fast?

What does it suggest if clock hand is sweeping very slow?
We’ve focused on miss rate. What about miss latency?

• Key observation: it is cheaper to pick a “clean” page over a “dirty” page
  ♦ Clean page does not need to be swapped to disk (after it has been previously swapped out)

• Challenge:
  ♦ How to get this info?
Refinement by adding extra hardware support

• Reference bit
  ♦ A hardware bit that is set whenever the page is referenced (read or written)

• Modified bit (dirty bit)
  ♦ A hardware bit that is set whenever the page is written into
**[lec11] x86 Page Table Entry**

<table>
<thead>
<tr>
<th>Page frame number</th>
<th>U</th>
<th>P</th>
<th>Cw</th>
<th>Gl</th>
<th>L</th>
<th>D</th>
<th>A</th>
<th>Cd</th>
<th>Wt</th>
<th>O</th>
<th>W</th>
<th>V</th>
</tr>
</thead>
<tbody>
<tr>
<td>31</td>
<td></td>
<td></td>
<td>12</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- **Valid (present)**
- **Read/write**
- **Owner (user/kernel)**
- **Write-through**
- **Cache disabled**
- **Accessed (referenced)**
- **Dirty**
- **PDE maps 4MB**
- **Global**

Reserved
Enhanced Clock

- Same as the basic Clock, except that it considers both (reference bit, modified bit)
  - (0,0): neither recently used nor modified (good)
  - (0,1): not recently used but dirty (not as good)
  - (1,0): recently used but clean (not good)
  - (1,1): recently used and dirty (bad)
- On page fault, follow hand to inspect pages:
  - Round 1:
    - If bits are (0,0), take it and stops
    - If bits are (0,1), record 1st instance
    - Clear ref bit for (1,0) and (1,1), if (0,1)/(0,0) not found yet
  - At end of round 1, if (0,1) was found, take it
  - If round 1 does not succeed, try 1 more round
Enhanced Clock

- **Pros**
  - Avoid write back

- **Cons**
  - More complicated, worse case scans multiple rounds
Summary

• Page replacement algorithms
  ♦ Optimal – replace page referenced furthest in the future
  ♦ FIFO – replace page loaded furthest in past
  ♦ LRU – replace page referenced furthest in past
  ♦ Clock – replace page that is “old enough”
  ♦ Enhanced Clock – pick clean pages first (for lower miss latency)

• We are finally done with Memory Management!
Next time...

- Chapters 39, 40, 41
What else can we do to improve miss latency?
Page out on critical path?

- If no free page in physical memory, swap in has to wait till a current page in physical memory is swapped out
  - Page fault handling time = proc. overhead + 2 * I/Os

- There is a chance of swapped out page being referenced soon
Page buffering techniques

OS maintains a pool of free pages

♦ When a page fault occurs, victim page chosen as before
♦ But desired page swapped into a free page (a slot in the free page pool) right away before victim page paged out
♦ OS swaps out dirty victim pages in the background, off the page fault critical path (to make more room in the free page pool)
Page buffering techniques

• Maintaining a list of free physical pages enables another important optimization

• Recall that the page replacement algorithm is a rough approximation of LRU
  ♦ Can certainly make mistakes
  ♦ LRU does not necessarily work well for all program behaviors

• Idea: If a page is on the free list, and it is accessed by a process before being reallocated, rescue it from the free list and give it back to the process
  ♦ Recovers from poor choices made by replacement algorithm