Administrivia

• Discussion section yesterday
  ♦ Going over project 2

• Discussion section next week
  ♦ Going over project 1 proper implementation
  ♦ If have time, homework 2 and/or homework 3
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.
[lec11] The Common Case

• Situation: 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. 5. MMU combines the PFN and offset into a physical address
  6. MMU then reads from that physical address, returns value to CPU
• Note: This is all done by the hardware
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

We’ll consider each in turn
If the TLB does not have mapping, two possibilities:
1. MMU loads PTE from page table in memory
   » 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

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…
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, mapped files)
  - **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)
Swapping to Disk

- The OS uses a **swap file** or **swap partition** for storing data evicted from physical memory
  - Windows: c:\pagefile.sys
  - Unix traditionally uses a **swap partition**
    » Region of disk just for evicting pages (no file system used)
    » But can also use a file in a file system if desired
  - A separate disk can improve performance
    » Disk I/O for paging does not interfere with disk I/O for files
    » Not as critical today with large physical memories

- Size of swap file/partition determines # of processes
  - Run out of swap → no more processes can be created
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 (1)
  - Techniques: Partitioning, paging, segmentation (1)
  - Page table management, TLBs, VM tricks (2)

- **Policies**
  - Page replacement algorithms (3)
Lecture Overview

- Review paging and page replacement
- Survey page replacement algorithms
- Discuss local vs. global replacement
- Discuss thrashing
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 it is acceptable
  - Processes usually exhibit both kinds of locality during their execution, making paging practical
Demand Paging (OS)

- Recall demand paging from the OS perspective:
  - Pages are evicted to disk when memory is full
  - Pages loaded from disk when referenced again
  - References to evicted pages cause a TLB miss
    - PTE was invalid, causes fault
  - OS allocates a page frame, reads page from disk
  - When I/O completes, the OS fills in PTE, marks it valid, and restarts faulting process
Demand Paging (Process)

- Demand paging is also used when a process first starts up
- When a process is created, it has
  - A brand new page table with all valid bits off
  - No pages in physical memory
- When the process starts executing
  - Instructions fault on code and data pages
  - Faulting stops when all necessary code and data pages are in memory
  - Only code and data needed by a process needs to be loaded
  - This, of course, changes over time…
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
  ♦ Proved by Belady
Belady’s Algorithm

• Belady’s algorithm is known as the optimal page replacement algorithm because it has the lowest fault rate for any page reference stream
  ♦ Idea: Replace the page that will not be used for the longest time in the future
  ♦ Problem: Have to predict the future

• Why is Belady’s useful then? Use it as a yardstick
  ♦ Compare implementations of page replacement algorithms with the optimal to gauge room for improvement
  ♦ If optimal is not much better, then algorithm is pretty good
  ♦ If optimal is much better, then algorithm could use some work
    » Random replacement is often the lower bound
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?
• Why not spatial locality?
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 (Belady’s: future)
  - 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>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**
Approximating LRU

• LRU approximations use the PTE reference bit
  ♦ Keep a counter for each page in OS
  ♦ At regular intervals, for every page do:
    » If ref bit = 0, increment counter
    » If ref bit = 1, zero the counter
    » Zero the reference bit
  ♦ The counter will contain the number of intervals since the last reference to the page
  ♦ The page with the largest counter is the least recently used

• Downside?
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
Clock (cont.)

• If memory is large, “accuracy” of information degrades
  ♦ What does it degrade to?
  ♦ One fix: use two hands (leading erase hand, trailing select hand)

• What happens if all reference bits are 1?

• 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

- 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
x86 Page Table Entry

<table>
<thead>
<tr>
<th>Page frame number</th>
<th>U</th>
<th>P</th>
<th>Cw</th>
<th>G</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>

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

Reserved

12
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
    » if bits are (0,1), record 1st instance
    » Clear ref bit for (1,0) and (1,1), if (0,1) 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
What else can we do to improve miss latency?
Page Fault Handling in demand paging

1. MMU (TLB)
2. Page fault
3. Page replacement
4. Adjust PTE of victim pg
5. Swap in page from swap
6. Resume faulting int
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
Key observation

- Locality in memory references
  - Spatial and temporal

- Want to keep a set of pages in memory that would avoid a lot of page faults
  - “Hot” pages

- Can we formalize it?
Working Set Model

- A working set of a process is used to model the dynamic locality of its memory usage
  - Defined by Peter Denning in 60s

- Definition
  - \( WS(t,w) = \{\text{all the pages that were referenced in the time interval (t, t-w)}\} \)
  - \( t \): time, \( w \): working set window (measured in page refs)

- A page is in the working set (WS) only if it was referenced in the last \( w \) references
**Working Set Size**

Working set size is the number of unique pages in the WS

- The number of pages referenced in the interval \((t, t-w)\)

The working set size changes with program locality

- During periods of poor locality, you reference more pages
- Within that period of time, the working set size is larger

Intuitively, want the working set to be the set of pages a process needs in memory to prevent heavy faulting

- Each process has a parameter \(w\) that determines a working set with few faults
- Denning: Don’t run a process unless working set is in memory

---

![Page reference table](image)

\[
\begin{align*}
\text{WS}(t_1) &= \{1,2,5,6,7\} \\
\text{WS}(t_2) &= \{3,4\}
\end{align*}
\]
Working Sets in the Real World

Working set size

transition, stable
Working Set Problems

- Problems
  - How do we determine $w$?
  - How do we know when the working set changes?
- Too hard to answer
  - So, working set is not used in practice as a page replacement algorithm
- However, it is still used as an abstraction
  - The intuition is still valid
  - When people ask, “How much memory does Firefox need?”, they are in effect asking for the size of Firefox’s working set
The BIG picture

• We’ve talked about single evictions
• Most computers are multiprogrammed
  ♦ Single eviction decision still needed
  ♦ New concern – processes compete for resources
  ♦ How to be “fair enough” and achieve good overall throughput
Possible replacement strategies

- Global replacement:
  - All pages from all processes are lumped into a single replacement pool
  - Most flexibility, least (performance) isolation

- Local replacement
  - Per-process replacement:
    » Each process has a separate pool of pages
  - Per-user replacement:
    » Lump all processes for a given user into a single pool

- In local replacement, must have a mechanism for (slowly) changing the allocations to each pool
Improving CPU utilization in multiprogramming

• In multiprogramming, when OS sees the CPU utilization is low,
  ♦ It thinks most processes are waiting for I/O
  ♦ it needs to increase the degree of multiprogramming (actual behavior of early paging systems)
  ♦ It adds/loads another process to the system
When there are not enough page frames

- Suppose many processes are making frequent references to 50 pages, memory has 49
- Assuming LRU
  - Each time one page is brought in, another page, whose content will soon be referenced, is thrown out
- Btw, what is the optimal strategy here?
  - MRU

- What is the average memory access time?
- The system is spending most of its time paging!

- The progress of programs makes it look like “memory access is as slow as disk”, rather than “disk being as fast as memory”
Thrashing

- Thrashing
  - When most of the time is spent by the OS in paging data back and forth from disk
  - Little time spent doing useful work (making progress)
  - In this situation, the system is overcommitted
Thrashing can lead to vicious cycle

- If a process does not have “enough” pages, the page-fault rate is very high. This leads to:
  - low CPU utilization
  - OS thinks that it needs to increase the degree of multiprogramming (actual behavior of early paging systems)
  - another process added to the system
  - page fault rate goes even higher
Thrashing (Cont.)

![Graph showing the relationship between CPU utilization and degree of multiprogramming, with a curve indicating Thrashing.]
What causes thrashing?

- The system does not know it has taken more work than it can handle

- What do humans do when thrashing?
  - Dropping or degrading a course if taking too many than you can handle 😊
Intuitively, what to do about thrashing?

• If a single process’s locality too large for memory, what can OS do?
  ♦ e.g., pin most data (hotter data) in memory, sacrifice the rest

• If the problem arises from the sum of several processes?
  ♦ Figure out how much memory each process needs – “locality”
  ♦ What can we do?
    » Can limit effects of thrashing using local replacement
    » Or, bring a process’ working set before running it
    » Or, wait till there is enough memory for a process’s need
Summary

• Page replacement algorithms
  ♦ Belady’s – optimal replacement (minimum # of faults)
  ♦ FIFO – replace page loaded furthest in past
  ♦ LRU – replace page referenced furthest in past
    » Approximate using PTE reference bit
  ♦ Clock – replace page that is “old enough”
  ♦ Enhanced Clock – pick clean pages first (for lower miss latency)
  ♦ Working Set – keep the set of pages in memory that has minimal fault rate (the “working set”)

• Multiprogramming
  ♦ Global vs. local replacement
  ♦ Thrashing
Next time...

- Move on to storage systems
- Read Chapters 37, 39
Belady's anomaly states that it is possible to have more page faults when increasing the number of page frames while using FIFO method of frame management. Laszlo Belady demonstrated this in 1970. Previously, it was believed that an increase in the number of page frames would always provide the same number or fewer page faults.
Example

Page Requests
321032432104
Example (Page Faults in Red)

<table>
<thead>
<tr>
<th>Frame</th>
<th>Page Requests – 3 frames</th>
</tr>
</thead>
<tbody>
<tr>
<td>Frame 1</td>
<td>3 2 1 0 3 2 4 3 2 1 0 4</td>
</tr>
<tr>
<td>Frame 2</td>
<td>3 3 3 0 0 0 4 4 4 4 4 4</td>
</tr>
<tr>
<td>Frame 3</td>
<td>2 2 2 3 3 3 3 3 1 1 1 1</td>
</tr>
<tr>
<td></td>
<td>1 1 1 2 2 2 2 2 0 0 0 0</td>
</tr>
</tbody>
</table>
Example (Page Faults in Red)

- Page Requests – 4 frames
- Frame 1
- Frame 2
- Frame 3
- Frame 4
Ideal curve of # of page faults v.s. # of physical pages
FIFO illustrating Belady’s anomaly
Page Fault Frequency (PFF)

- Page Fault Frequency (PFF) is a variable space algorithm that uses a more ad-hoc approach
  - Monitor the fault rate for each process
  - If the fault rate is above a high threshold, give it more memory
    » So that it faults less
    » But not always (FIFO, Belady’s Anomaly)
  - If the fault rate is below a low threshold, take away memory
    » Should fault more
    » But not always

- Hard for PFF to distinguish between changes in locality and changes in size of working set