Virtual Memory
Questions from last time

• Page number: bits needed to identify a frame
• Page offset: bits needed to address a byte in a page/frame. 4K page needs 12 bits.
• How are page tables started? Allocation by OS
• Why multi-level page tables? Very Large contiguous page table causes problems
• Where is virtual memory? Its virtual. Main memory-Secondary memory interface.
• Paging algorithms seem to assume both temporal and spatial locality... how true is this?
• Optimal Page Replacement: Why can’t future page numbers be predicted?
  – Bringing in a page of information, we are already exploiting spatial locality.
  – LRU assumes some temporal locality.
• Page replacement policies vs cache replacement policies which we did not discuss
• Can more than one page loaded into memory when a process starts? prefetching
• Why are disk addresses of non-resident pages not stored in the page table? Generally contains only information used on page hits.
• Is a page replaced when the process is finished with it?
• Dirty bit: indicates that a page has been modified, hence is no longer an exact copy of the page on the disk.
• Why context switch when a page fault occurs? Allocate CPU to a different while disk is accessed.
• Why LRU and OPT not affected by Belady’s anomaly? Stack type. When frames are less they are a subset of pages when frame are more.
First-In-First-Out (FIFO) Algorithm

- Reference string: \(7,0,1,2,0,3,0,4,2,3,0,3,2,1,2,0,1,7,0,1\)
- 3 frames (3 pages can be in memory at a time per process)
- 15 page faults (out of 20 accesses)
- Sometimes a page is needed soon after replacement: \(7,0,1,2,0,3 (0\text{ out}),0, \ldots\)
“Optimal” Algorithm

- Replace page that will not be used for longest period of time
  
  4th access: replace 7 because we will not use if got the longest time...
  
  9 page replacements is optimal for the example

- But how do we know this?
  
  Can’t read the future in reality.

- Used for *measuring* how well an algorithm performs
Least Recently Used (LRU) Algorithm

- Use past knowledge rather than future
- Replace page that has not been used in the most amount of time (4th access – page 7 is least recently used ...)
- Associate time of last use with each page

reference string

| 7 | 0 | 1 | 2 | 0 | 3 | 0 | 4 | 2 | 3 | 0 | 3 | 2 | 1 | 2 | 0 | 1 | 7 | 0 | 1 |

page frames

<table>
<thead>
<tr>
<th>7</th>
<th>7</th>
<th>7</th>
<th>2</th>
<th>2</th>
<th>4</th>
<th>4</th>
<th>4</th>
<th>0</th>
<th>1</th>
<th>1</th>
<th>1</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td></td>
</tr>
</tbody>
</table>

- 12 faults – better than FIFO (15) but worse than OPT (9)
- Generally good algorithm and frequently used
- But how to implement it by tracking the page usage?
Least Recently Used (LRU) Algorithm

- LRU page number is marked (*).
- Unmarked if that page is accessed.
LRU Algorithm: Implementations

Possible implementations

• Counter implementation
  – Every page entry has a counter; every time page is referenced through this entry, copy the clock into the counter
  – When a page needs to be changed, look at the counters to find smallest value
    • Search through table needed

• Stack implementation
  – Keep a stack of page numbers in a double link form:
  – Page referenced:
    • move it to the top
    • requires 6 pointers to be changed
  – Each update expensive
  – No search for replacement needed (bottom is least recently used)

LRU and OPT are cases of stack algorithms that don’t have Belady’s Anomaly
Use Of A Stack to Record Most Recent Page References

reference string

4 7 0 7 1 0 1 2 1 2 7 1 2

Most recently used ->

2
1
0
7
4

stack before
a

7
2
1
0
4

stack after
b

Least recently used ->

This shows tracking stack, not actual frames.

Too slow if done in software
Use Of A Stack to Record Most Recent Page References

<table>
<thead>
<tr>
<th>Most recently used</th>
<th>Least recently used</th>
</tr>
</thead>
<tbody>
<tr>
<td>4 7 0 7 1 0 1 2 1 2 7 1 2</td>
<td></td>
</tr>
<tr>
<td>4 7 0 7 1 0 1 2 1 2 7 1 2</td>
<td></td>
</tr>
<tr>
<td>4 7 0 7 1 0 1 2 1 2 7 1 2</td>
<td></td>
</tr>
<tr>
<td>4 7 1 0 7 1 0 1 2 1 2 7 1 2</td>
<td></td>
</tr>
<tr>
<td>4 7 1 0 7 1 0 1 2 1 2 7 1 2</td>
<td></td>
</tr>
</tbody>
</table>

Detailed version of previous slide. This shows tracking stack, not actual frames.
Use Of A Stack to Record Most Recent Page References

reference string

| 7 | 0 | 1 | 2 | 0 | 3 | 0 | 4 | 2 | 3 | 0 | 3 | 2 | 1 | 2 | 0 | 1 | 7 | 0 | 1 |
| 7 | 7 | 7 | 2 | 2 | 4 | 4 | 4 | 0 | 1 | 1 | 1 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 0 | 0 | 0 | 0 | 3 | 3 | 3 | 3 | 3 | 3 | 3 | 3 | 3 | 3 | 3 | 3 | 3 | 3 |
| 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |

page frames

Earlier problem (upper) revisited.
This shows tracking stack, not actual frames.
LRU Approximation Algorithms

• LRU needs special hardware and still slow

• **Reference bit**
  – With each page associate a bit, initially = 0
  – When the page is referenced, bit set to 1
  – Replace any page with reference bit = 0 (if one exists)
    • 0 implies not used since initialization
    • We do not know the order, however.
      – pick a non-dirty page first
    – Periodically clear the reference bit.

• Advanced schemes using more bits: preserve more information about the order
LRU approximation

Ref bit: 1 indicates used, Shift register records history

Ex: 3-period history

<table>
<thead>
<tr>
<th>Ref Bit</th>
<th>Shift Register</th>
<th>Shift Register after OS timer interrupt</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0000 0000</td>
<td>1000 0000</td>
</tr>
<tr>
<td>1</td>
<td>1001 0001</td>
<td>1100 1000</td>
</tr>
<tr>
<td>0</td>
<td>0110 0011</td>
<td>0011 0001</td>
</tr>
</tbody>
</table>

- Interpret 8-bit bytes as **unsigned integers**
- Page with the *lowest* number is the LRU page: replace.

Examples:
- 00000000: Not used in last 8 periods
- 01100101: Used 4 times in the last 8 periods
- 11000100 used more recently than 01110111
Second-chance (clock) algorithm

- **Second-chance algorithm (“clock algo”)**
  - Round robin selection of victim page, recently used page gets second chance.
  - **Clock** replacement (using circular queue): hand as a pointer
  - Page referenced: reference bit = 1
  - Page replacement: Consider next page
    - Reference bit = 0 -> replace it
    - reference bit = 1 then: **give it another chance**
      - set reference bit 0, leave page in memory
      - consider next page, subject to same rules
Second-Chance (clock) Page-Replacement Algorithm

- **Clock replacement:** hand as a pointer
- **Consider next page**
  - Reference bit $= 0$ -> replace it
  - Reference bit $= 1$ then:
    - set reference bit 0,
      leave page in memory
    - consider next page, subject to same rules

(Fig a) change to 0
(Fig b) replace page
Enhanced Second-Chance Algorithm

- Improve algorithm by using reference bit and modify bit (if available) in concert clean page: better replacement candidate
- Take ordered pair (reference, modify)
  1. (0, 0) neither recently used not modified – best page to replace
  2. (0, 1) not recently used but modified – not quite as good, must write out before replacement
  3. (1, 0) recently used but clean – probably will be used again soon
  4. (1, 1) recently used and modified – probably will be used again soon and need to write out before replacement
- When page replacement called for, use the clock scheme but use the four classes replace page in lowest non-empty class
  - Might need to search circular queue several times
Counting Algorithms

• Keep a counter of the number of references that have been made to each page
  – Not common

• **Least Frequently Used (LFU) Algorithm:** replaces page with smallest count

• **Most Frequently Used (MFU) Algorithm:** based on the argument that the page with the smallest count was probably just brought in and has yet to be used
Clever Techniques for enhancing Perf

• Keep a buffer (pool) of free frames, always
  – Then frame available when needed, not found at fault time
  – Read page into free frame and select victim to evict and add to free pool
    – When convenient, evict victim
• Keep list of modified pages
  – When backing store is otherwise idle, write pages there and set to non-dirty (being proactive!)
• Keep free frame previous contents intact and note what is in them
  – If referenced again before reused, no need to load contents again from disk
  – Generally useful to reduce penalty if wrong victim frame selected
Buffering and applications

• Some applications (like databases) often understand their memory/disk usage better than the OS
  – Provide their own buffering schemes
  – If both the OS and the application were to buffer
    • Twice the I/O is being utilized for a given I/O
  – OS may provide “raw access” disk to special programs without file system services.
Allocation of Frames
How to allocate frames to processes?

- Each process needs *minimum* number of frames
  Depending on specific needs of the process
- *Maximum* of course is total frames in the system

- Two major allocation schemes
  - fixed allocation
  - priority allocation

- Many variations
Fixed Allocation

- **Equal allocation** – For example, if there are 100 frames (after allocating frames for the OS) and 5 processes, give each process 20 frames
  - Keep some as free frame buffer pool

- **Proportional allocation** – Allocate according to the size of process (need based)
  - Dynamic as degree of multiprogramming, process sizes change

\[
s_i = \text{size of process } p_i \\
S = \sum s_i \\
m = \text{total number of frames} \\
a_i = \text{allocation for } p_i = \frac{s_i}{S} \times m
\]

Example:
Processes P1, P2

\[
m = 64 \\
s_1 = 10 \\
s_2 = 127 \\
\frac{10}{137} \times 62 \approx 4 \\
\frac{127}{137} \times 62 \approx 57
\]
Priority Allocation

• Use a proportional allocation scheme using priorities rather than size

• If process $P_i$ generates a page fault,
  – select for replacement one of its frames or
  – select for replacement a frame from a process with lower priority number
Global vs. Local Allocation

• **Global replacement** – process selects a replacement frame from the set of all frames; one process can take a frame from another
  – But then process execution time can vary greatly
  – But greater throughput, so more common

• **Local replacement** – each process selects from only its own set of allocated frames
  – More consistent per-process performance
  – But possibly underutilized memory
Problem: Thrashing

• If a process does not have “enough” pages, the page-fault rate is very high
  – Page fault to get page
  – Replace existing frame
  – But quickly need replaced frame back
  – This leads to:
    • Low CPU utilization, leading to
    • Operating system thinking that it needs to increase the degree of multiprogramming leading to
    • Another process added to the system

• **Thrashing** ⇔ a process is busy swapping pages in and out
Thrashing (Cont.)

[Graph showing CPU utilization and degree of multiprogramming]

- X-axis: Degree of multiprogramming
- Y-axis: CPU utilization

 thrashing
Demand Paging and Thrashing

- Why does demand paging work? **Locality model**
  - Process migrates from one locality to another
  - Localities may overlap

- Why does thrashing occur?

  size of locality > total memory size allocated

  - Limit effects by using local or priority page replacement
Locality In A Memory-Reference Pattern
Working-Set Model

- $\Delta \equiv$ **working-set window** $\equiv$ a fixed number of page references
  
  Example: 10,000 instructions

```
\Delta = 10 \text{ page references}
```

- \( WSS_i \) (working set of Process \( P_i \)) =
  
  total number of pages referenced in the most recent $\Delta$ (varies in time)
  
  - if $\Delta$ too small, working set will not encompass entire locality
  - if $\Delta$ too large, ws will encompass several localities
  - if $\Delta = \infty \Rightarrow$ ws will encompass entire program
  - ws is an approximation of locality

- \( D = \Sigma WSS_i \equiv \text{total demand for frames} \) for all processes
  
  - if $D > m \Rightarrow$ **Thrashing**
  
  - **Policy** if $D > m$, then suspend or swap out one of the processes
Page-Fault Frequency Approach

• More direct approach than WSS
• Establish “acceptable” page-fault frequency (PFF) rate and use local replacement policy
  – If actual rate too low, process loses frame
  – If actual rate too high, process gains frame
Working Sets and Page Fault Rates

- Direct relationship between working set of a process and its page-fault rate
- Working set changes over time
- Peaks and valleys over time

Peaks occur at locality changes: 3 working sets
Memory-Mapped Files

- Memory-mapped file I/O allows file I/O to be treated as routine memory access by mapping a disk block to a page in memory
- File is then in memory instead of disk
- A file is initially read using demand paging
  - A page-sized portion of the file is read from the file system into a physical page
  - Subsequent reads/writes to/from the file are treated as ordinary memory accesses
- Simplifies and speeds file access by driving file I/O through memory rather than read() and write() system calls
- Also allows several processes to map the same file allowing the pages in memory to be shared
- But when does written data make it to disk?
  - Periodically and/or at file close() time
  - For example, when the pager scans for dirty pages
Memory Mapped Files

Disk File uses 6 blocks
Page tables used for mapping
Allocating Kernel Memory

• Treated differently from user memory
• Often allocated from a free-memory pool
  – Kernel requests memory for structures of varying sizes
    • Process descriptors, semaphores, file objects etc.
    • Often much smaller than page size
  – Some kernel memory needs to be contiguous
    • I.e. for device I/O
  – approaches (skipped)
Other Considerations -- Prepaging

• Prepaging
  – To reduce the large number of page faults that occurs at process startup
  – Prepage all or some of the pages a process will need, before they are referenced
  – But if prepaged pages are unused, I/O and memory was wasted
  – Assume $s$ pages are prepaged and fraction $\alpha$ of the pages is used
    • Is cost of $s \cdot \alpha$ saved pages faults > or < than the cost of prepaging $s \cdot (1-\alpha)$ unnecessary pages?
    • $\alpha$ near zero $\Rightarrow$ greater prepaging loses
Other Issues – Page Size

• Sometimes OS designers have a choice
  – Especially if running on custom-built CPU

• Page size selection must take into consideration:
  – Fragmentation
  – Page table size
  – I/O overhead
  – Number of page faults
  – Locality
  – TLB size and effectiveness

• Always power of 2, usually in the range $2^{12}$ (4,096 bytes) to $2^{22}$ (4,194,304 bytes)

• On average, growing over time
Page size issues – TLB Reach

- TLB Reach - The amount of memory accessible from the TLB
- TLB Reach = (TLB Size) X (Page Size)
- Ideally, the working set of each process is stored in the TLB
  - Otherwise there is a high degree of page faults
Other Issues – Program Structure

- Program structure
  - int[128,128] data;  \( i \): row, \( j \): column
  - Each row is stored in one page
  - Program 1
    \[
    \text{for } (j = 0; j < 128; j++), \ dup{for} (i = 0; i < 128; i++) \quad \text{data}[i,j] = 0;
    \]
    \[
    128 \times 128 = 16,384 \text{ page faults}
    \]
  - Program 2  inner loop = 1 row = 1 page
    \[
    \text{for } (i = 0; i < 128; i++), \ dup{for} (j = 0; j < 128; j++) \quad \text{data}[i,j] = 0;
    \]
    \[
    128 \text{ page faults}
    \]
• **I/O Interlock** – Pages must sometimes be locked into memory

• Consider I/O - Pages that are used for copying a file from a device must be locked from being selected for eviction by a page replacement algorithm

• **Pinning** of pages to lock into memory
Windows

- Uses demand paging with clustering. Clustering brings in pages surrounding the faulting page.
- Processes are assigned working set minimum and working set maximum.
- Working set minimum is the minimum number of pages the process is guaranteed to have in memory.
- A process may be assigned as many pages up to its working set maximum.
- When the amount of free memory in the system falls below a threshold, automatic working set trimming is performed to restore the amount of free memory.
- Working set trimming removes pages from processes that have pages in excess of their working set minimum.
CS370 Operating Systems

Colorado State University
Yashwant K Malaiya
Spring 2018  Lecture T

File-system Interface

Slides based on
• Text by Silberschatz, Galvin, Gagne
• Various sources
Chapter 11: File-System Interface

- File Concept
- Access Methods
- Disk and Directory Structure
- File-System Mounting
- File Sharing
- Protection
"MS. GRIMMETT, I SORT OF LIKED THE OLD FILING SYSTEM...IN THE FILE CABINETS."
Outline

• File Concept, types
• Attributes, Access Methods, operations, Protection
• Directory Structure, namespace, File-System Mounting, File Sharing
• Next in File System Implementation
  – **Storage abstraction**: File system metadata (size, freelists), File metadata (attributes, disk block maps), datablocks
  – **Allocation of blocks to files**: contiguous, sequential, linked list allocation, indexed
  – **In memory info**: Mount table, directory structure cache, open file table, buffers
  – **Unix**: inodes numbers for directories and files
# File types

<table>
<thead>
<tr>
<th>file type</th>
<th>usual extension</th>
<th>function</th>
</tr>
</thead>
<tbody>
<tr>
<td>executable</td>
<td>exe, com, bin, or none</td>
<td>ready-to-run machine-language program</td>
</tr>
<tr>
<td>object</td>
<td>obj, o</td>
<td>compiled, machine language, not linked</td>
</tr>
<tr>
<td>source code</td>
<td>c, cc, java, pas, asm, a</td>
<td>source code in various languages</td>
</tr>
<tr>
<td>batch</td>
<td>bat, sh</td>
<td>commands to the command interpreter</td>
</tr>
<tr>
<td>text</td>
<td>txt, doc</td>
<td>textual data, documents</td>
</tr>
<tr>
<td>word processor</td>
<td>wp, tex, rtf, doc</td>
<td>various word-processor formats</td>
</tr>
<tr>
<td>library</td>
<td>lib, a, so, dll</td>
<td>libraries of routines for programmers</td>
</tr>
<tr>
<td>print or view</td>
<td>ps, pdf, jpg</td>
<td>ASCII or binary file in a format for printing or viewing</td>
</tr>
<tr>
<td>archive</td>
<td>arc, zip, tar</td>
<td>related files grouped into one file, sometimes compressed, for archiving or storage</td>
</tr>
<tr>
<td>multimedia</td>
<td>mpeg, mov, rm, mp3, avi</td>
<td>binary file containing audio or A/V information</td>
</tr>
</tbody>
</table>
File Attributes

- **Name** – only information kept in human-readable form
- **Identifier** – unique tag (number) identifies file within file system
- **Type** – needed for systems that support different types
- **Location** – pointer to file location on device
- **Size** – current file size
- **Protection** – controls who can do reading, writing, executing
- **Time, date, and user identification** – data for protection, security, and usage monitoring
- Information about files are kept in the directory structure, which is maintained on the disk
- Many variations, including extended file attributes such as file checksum
Disk Structure

- Disk can be subdivided into partitions
- Disks or partitions can be RAID protected against failure
- Partition can be formatted with a file system
- Entity containing file system known as a volume
- Each volume containing file system also tracks that file system’s info in device directory or volume table of contents
- As well as general-purpose file systems there are many special-purpose file systems, frequently all within the same operating system or computer
A collection of nodes containing information about all files

Both the directory structure and the files reside on disk
Operations Performed on Directory

- Traverse the file system
- List a directory
- Search for a file
- Create/Delete/Rename a file
Directory Organization

The directory is organized logically to obtain

- **Efficiency** – locating a file quickly
- **Naming** – convenient to users
  - Two users can have same name for different files
  - The same file can have several different names
- **Grouping** – logical grouping of files by properties, (e.g., all Java programs, all games, ...)