Some page replacement algorithms:

- Belady
- Not-recently-used (NRU)
- First-in-first-out (FIFO)
- Second chance
- Least-recently-used (LRU)
- Least-frequently-used
- Most-frequently-used

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?
- On average performs very well (close to Belady)
- But ....
Implementing Exact LRU

- Option 1:
  - Time stamp every reference
  - Evict page with oldest time stamp
  - Problems:
    - Need to make PTE large enough to hold meaningful time stamp (may double size of page tables, TLBs)
    - Need to examine every page on eviction to find one with oldest time stamp
- Option 2:
  - Keep pages in a stack. On reference, move the page to the top of the stack. On eviction, replace page at bottom.
  - Problems:
    - Need costly software operation to manipulate stack on EVERY memory reference!

Counting-based Replacement

- Count number of uses of a page
- Least-Frequently-Used (LFU)
  - Replace the page used least often
  - Pages that are heavily used at one time tend to stick around even when not needed anymore
  - Newly allocated pages haven’t had a chance to be used much
- Most-Frequently-Used (MFU)
  - Favours new pages
  - Neither is common, both are poor approximations of OPT

What are possible replacement algorithms?

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Ease of implementation</th>
<th>Performance</th>
</tr>
</thead>
<tbody>
<tr>
<td>Not-recently-used (NRU)</td>
<td>+</td>
<td>~</td>
</tr>
<tr>
<td>First-in-first-out (FIFO)</td>
<td>+</td>
<td>~</td>
</tr>
<tr>
<td>Least-recently-used (LRU)</td>
<td>-</td>
<td>++</td>
</tr>
<tr>
<td>Least-frequently-used</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>Most-frequently-used</td>
<td>-</td>
<td></td>
</tr>
</tbody>
</table>

What are possible replacement algorithms?

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Ease of implementation</th>
<th>Performance</th>
</tr>
</thead>
<tbody>
<tr>
<td>Not-recently-used (NRU)</td>
<td>+</td>
<td>~</td>
</tr>
<tr>
<td>First-in-first-out (FIFO)</td>
<td>+</td>
<td>~</td>
</tr>
<tr>
<td>Least-recently-used (LRU)</td>
<td>-</td>
<td>++</td>
</tr>
<tr>
<td>Least-frequently-used</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>Most-frequently-used</td>
<td>-</td>
<td></td>
</tr>
</tbody>
</table>
What are possible replacement algorithms?

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Ease of Implementation</th>
<th>Performance</th>
</tr>
</thead>
<tbody>
<tr>
<td>Not-recently-used (NRU)</td>
<td>+</td>
<td>~</td>
</tr>
<tr>
<td>First-in-first-out (FIFO)</td>
<td>+</td>
<td>~</td>
</tr>
<tr>
<td>Least-recently-used (LRU)</td>
<td>-</td>
<td>++</td>
</tr>
<tr>
<td>Least-frequently-used</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Most-frequently-used</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Second chance</td>
<td>+</td>
<td>+</td>
</tr>
</tbody>
</table>

On A4 you are going to implement two page replacement algorithms for OS161!

Fixed vs. Variable Space

- In a multiprogramming system, we need a way to allocate memory to competing processes
- Problem: How to determine how much memory to give to each process?
  - Fixed space algorithms
    - Each process is given a limit of pages it can use
    - When it reaches the limit, it replaces from its own pages
    - **Local replacement**
      - Some processes may do well while others suffer
  - Variable space algorithms
    - Process' set of pages grows and shrinks dynamically
    - **Global replacement** - one process can ruin it for the rest
    - **Local replacement** - replacement and set size are separate

Working Set Model

- How do you decide how large the fixed or variable space for a process should be?
- Depends on access pattern …

<table>
<thead>
<tr>
<th>Process 1</th>
<th>Process 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>6 1 5 2 1 6 2 7 5 1 7</td>
<td>4 4 3 4 1 3 4 4 4</td>
</tr>
<tr>
<td>5 pages: {1,2,5,6,7}</td>
<td>2 or 3 pages: {3,4} or {1,3,4}</td>
</tr>
</tbody>
</table>
## 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 60's
- **Definition**
  - \( WS(t, \Delta) = \{\text{pages } P \text{ such that } P \text{ was referenced in the time interval } (t, t-\Delta)\} \)
  - \( t = \text{time, } \Delta = \text{working set window (measured in page refs)} \)
- A page is in the working set \( WS \) only if it was referenced in the last \( \Delta \) references

\[
\begin{array}{cccccccccccc}
... & 2 & 6 & 1 & 5 & 2 & 1 & 6 & 7 & 5 & 1 & 6 \\
& \Delta & \uparrow & & & & & & & \Delta & \uparrow & \text{WS(t1) = \{1,2,5,6,7\}} & \text{WS(t2) = \{3,4\}} & t_1 & t_2
\end{array}
\]

## Working Set Size

- The working set size is the number of pages in the working set
  - The number of pages referenced in the interval \( (t, t-\Delta) \)
  - 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 \( \Delta \) that determines a working set with few faults
    - Denning: Don't run a process unless working set is in memory

## Working Set Problems

- **Problems**
  - How do we determine \( \Delta \)?
  - 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

- **So then how do we decide how much memory space to allocate for a process?**
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 to use PFF to distinguish between changes in locality and changes in size of working set.

Thrashing

- When more time is spent by the OS in paging data back and forth from disk than executing user programs.
- No time spent doing useful work (making progress).
- In this situation, the system is overcommitted.
  - No idea which pages should be in memory to reduce faults.
  - Could just be that there isn’t enough physical memory for all of the processes in the system.
  - Ex: Running Windows with 4 MB of memory…

Windows XP Paging Policy

- Local page replacement.
  - Per-process FIFO.
  - Processes start with a default of 50 pages.
- XP monitors page fault rate and adjusts working-set size accordingly.
  - On page fault, cluster of pages around the missing page are brought into memory.
Linux Paging

- Global replacement, like most Unix
- Modified second-chance clock algorithm
  - Pages *age* with each pass of the clock hand
  - Pages that are not used for a long time will eventually have a value of zero
- Continually under development…

How much space does a page table take up?

- Need one PTE per page
- 32 bit virtual address space w/ 4K pages
  - $2^{20}$ PTEs
- 4 bytes/PTE = 4MB/page table
- 25 processes = 100MB just for page tables!
  - And modern processors have 64-bit address spaces -> 16 petabytes for page table!

**Solutions**
- Hashed page tables
- Hierarchical (multi-level) page tables

Managing Page Tables

- How can we reduce space overhead?
  - Observation: Only need to map the portion of the address space actually being used (tiny fraction of entire addr space)
- How do we only map what is being used?
  - Can dynamically extend page table…
  - Does not work if addr space is sparse (internal fragmentation)
- Use another level of indirection: two-level page tables (or multi-level page tables)

Motivation: two-level page tables

How does address translation work now?
Two-Level Page Tables
Virtual addresses (VAs) have three parts:
- Master page number, secondary page number, and offset
- Master page table maps VAs to secondary page table
- Secondary page table maps page number to physical frame
- Offset selects address within physical frame

2-Level Paging Example
- 32-bit virtual address space
  - 4K pages, 4 bytes/PTE
  - How many bits in offset?
    - 4K = 12 bits, leaves 20 bits
  - Want master/secondary page tables in 1 page each:
    - 4K/4 bytes = 1K entries.
      - How many bits to address 1K entries?
        - 10 bits
      - secondary = 10 bits
      - offset = 12 bits
      - master = 32 – 10 – 12 = 10 bits
  - This is why 4K is common page size!

Pentium Address Translation

New topic:
- File systems!
What do file systems do?

- They provide a nice abstraction of storage:

**Reality**
- Hard Disks
  - Read block
  - Write block

**Abstraction**
- ugr
  - demsz
  - schneider
  - peterson
  - grants
  - papers

File Systems

- Implement an abstraction (files) for secondary storage
- Organize files logically (directories)
- Permit sharing of data between processes, people, and machines
- Protect data from unwanted access (security)

File Concept

- A file is named collection of data with some attributes
  - Name
  - Owner
  - Location
  - Size
  - Protection
  - Creation time
  - Time of last access

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</td>
<td>read to run machine-language programs</td>
</tr>
<tr>
<td>Obj</td>
<td>obj, o</td>
<td>compiled, machine language, not known</td>
</tr>
<tr>
<td>Source Code</td>
<td>c, c++, java, perl, awk</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>literal data, documents</td>
</tr>
<tr>
<td>Word Processor</td>
<td>txt, tex, rt, doc</td>
<td>various word-processor formats</td>
</tr>
<tr>
<td>Library</td>
<td>lib, a, so, dll</td>
<td>libraries of routines for programming</td>
</tr>
</tbody>
</table>

- A file's type can be encoded in its name or contents
  - Windows encodes type in name
    - .exe, .doc, .jpg, etc.
  - Unix encodes type in contents (sometimes)
    - Magic numbers, initial characters (e.g., #! for shell scripts)
Conceptual File Operation

- Create
- Write
- Read
- Repositioning within file
- Delete
- Truncating a file
- Open
- Close

Unix (C library)

- creat(name)
- write(fd, buf, len)
- read(fd, buf, len)
- lseek(fd, pos)
- unlink(name)
- truncate(fd, length)
- open(name, mode)
- close(fd)

Where in the file do write and read operations operate?

File Access Methods

- General-purpose file systems support simple methods
  - Sequential access – read bytes one at a time, in order
    - read next
    - write next
  - Direct access – random access given block/byte number
    - read n (byte at offset n)
    - write n
- What does Unix use? [both]

Unix (C library)

- read(fd, buf, len)
- write(fd, buf, len)
- lseek(fd, pos)

File Access Methods

- Database systems support more sophisticated methods
  - Record access
  - Indexed access

- Modern OS file systems support only simple methods (direct access, sequential access)
  - Why?

Get record where name equals "David"

<table>
<thead>
<tr>
<th>Index</th>
<th>John</th>
<th>75837</th>
<th>A</th>
</tr>
</thead>
<tbody>
<tr>
<td>Index</td>
<td>David</td>
<td>63548</td>
<td>C+</td>
</tr>
<tr>
<td>Index</td>
<td>Alice</td>
<td>92746</td>
<td>A</td>
</tr>
</tbody>
</table>

Conceptual File Operation

- Open
- Close

Why do we need open and close operations?

Unix (C library)

- creat(name)
- write(fd, buf, len)
- read(fd, buf, len)
- seek(fd, pos)
- unlink(name)
- truncate(fd, length)
- close(fd)
Handling operations on files

- Involves searching the directory for the entry associated with the named file
  - when the file is first used actively, store its attribute info in a system-wide open-file table; the index into this table is used on subsequent operations ⇒ no searching

Unix example (open, read, write are syscalls):

```c
main() {
    char onebyte;
    int fd = open("sample.txt", "r");
    read(fd, &onebyte, 1);
    write(STDOUT, &onebyte, 1);
    close(fd);
}
```

Shared open files

- there are actually 2 levels of internal tables
  - a per-process table of all files that each process has open (this holds the current file position for the process)
  - each entry in the per-process table points to an entry in the system-wide open-file table (for process independent info)

Directories

- Directories serve multiple purposes
  - For users, they provide a structured way to organize files
  - For the file system, they provide a convenient naming interface that allows the implementation to separate logical file organization from physical file placement on the disk
  - Also store information about files (owner, permission, etc.)

Directories

- Most file systems support multi-level directories
  - Naming hierarchies (/, /usr, /usr/local, …)
What is a directory at the OS level?

- A directory is a list of entries – names and associated metadata
  - Metadata is not the data itself, but information that describes properties of the data (size, protection, location, etc.)
- List is usually unordered (effectively random)
  - Entries usually sorted by program that reads directory
- Directories typically stored in files
  - Only need to manage one kind of secondary storage unit

Operations on Directories

- Search
  - find a particular file within directory
- Create file
  - add a new entry to the directory
- Delete file
  - remove an entry from the directory
- List directory
  - Return file names and requested attributes of entries
- Update directory
  - Record a change to some file’s attributes

Path Name Translation

- Let’s say you want to open “/one/two/three”
- What does the file system do?
  - Open directory “/” (the root, well known, can always find)
  - Search for the entry “one”, get location of “one” (in directory entry)
  - Open directory “one”; search for “two”, get location of “two”
  - Open directory “two”, search for “three”, get location of “three”
  - Open file “three”

- Systems spend a lot of time walking directory paths
  - This is why open is separate from read/write
- OS will cache prefix lookups for performance
  - /a/b, /a/bb, /a/bbb, etc., all share “/a” prefix

Why do we need open and close operations?

Possible Directory Implementations

- single-level, two-level, tree-structured
- acyclic-graph directories: allows for shared directories
  - the same file or subdirectory may be in 2 different directories

Tree-structured:

Acyclic graph:
**File Links**

- Sharing can be implemented by creating a new directory entry called a *link*: a pointer to another file or subdirectory
  - Symbolic, or soft, link
  - Hard links

**Symbolic vs. Hard Links**

- Symbolic, or soft, link:
  - Directory entry contains "true" path to the file
- Hard links:
  - Second directory entry identical to the first

**Symbolic vs. Hard Links**

- In the case of a hard link: what happens if either Alice or Bob remove *lpage.c*?
  - It will still be accessible for the other person
  - The OS maintains a count of how many hard links there are
  - File gets only deleted if all users have removed it

- In the case of a soft link: what happens if Bob removes *lpage.c*?
  - It will still be accessible to Alice
  - The OS will only remove the "link" entry in Bob's directory
  - What if Alice removes *lpage.c*?
    - The file will be removed
    - Bob can still see link entry in his directory, but cannot access the file