Virtual Memory

- Overview / Motivation
- Locality of Reference
- Demand Paging
- Policies
  - Placement
  - Replacement
  - Allocation

Demand Paging

- "Lazy Swapper": only swap in pages that are needed.
- Whenever CPU tries to access a page that is not swapped in, a page fault occurs.
Virtual Memory

Mechanics of a Memory Reference

1. Reference
2. Page table
3. Complete reference
4. Access memory
5. Some frame
6. Physical memory

Mechanics of a Page Fault

1. Reference
2. Exception!
3. Page is on backing store
4. Load page
5. Update page table
6. Frame used
7. Physical memory
8. Restart instruction
Locality of Reference

- Page faults are expensive!
- **Thrashing**: Process spends most of the time paging in and out instead of executing code.
- Most programs display a pattern of behavior called the **principle of locality of reference**.

**Locality of Reference**: A program that references a location \( n \) at some point in time is likely to reference the same location \( n \) and locations in the immediate vicinity of \( n \) in the near future.

Memory Access Trace: Example
Performance of Demand Paging

**Operations during Page Fault:**

1. Service page fault
2. Swap in page
3. Restart process

**Effective Memory Access Time** $ema$:

$$ema = (1-p) \times ma + p \times \text{“page fault time”}$$

where

- $p$ = probability of a page fault
- $ma$ = memory access time

Interlude: Architectural Considerations

- Must be able to **restart** any instruction after a page fault, e.g.,

  $$ADD A,B \ TO \ C$$

- What about operations that **modify several locations** in memory?
  - e.g. block copy operations?

- What about operations with **side effects**?
  - e.g. PDP-11, 80x86 auto-decrement, auto-increment operations?
  - Add mechanism for OS to “undo” instructions.
OS Policies for Virtual Memory

- **Fetch Policy**
  - How/when to get pages into physical memory.
  - demand paging vs. pre-paging.
- **Placement Policy**
  - Where in physical memory to put pages.
  - Only relevant in NUMA machines.
- **Replacement Policy**
  - Physical memory is full. Which frame to page out?
- **Resident Set Management Policy**
  - How many frames to allocate to process?
  - Replace someone else’s frame?
- **Cleaning Policy**
  - When to write a modified page to disk.
- **Load Control**

Configuring the Windows Memory Manager

- Registry Values that Affect the Memory Manager:
  
  ClearPageFileAtShutdown
  DisablePagingExecutive
  IoPageLockLimit
  LargePageMinimum
  LargeSystemCache
  NonPagedPoolQuota
  NonPagedPoolSize
  PagedPoolQuota
  PagedPoolSize
  SystemPages
Page Replacement

- Virtual memory allows higher degrees of multiprogramming by over-allocating memory.

Mechanics of Page Replacement

Invoked whenever no free frame can be found.

Problem: Need two page transfers
Solution: Dirty bit.
Page Replacement Algorithms

- Objective: Minimize page fault rate.
- Why bother?

- Example

```java
for(int i=0; i<10; i++) {
    a = x * a;
}
```

- Evaluation: Sequence of memory references: reference string.

---

FIFO Page Replacement

1. Select victim
2. Swap out victim page
3. Invalidate entry for victim page
4. Swap in new page
5. Update entry for new page
6. Enter frame in FIFO queue
FIFO Page Replacement (cont.)

- Example:

<table>
<thead>
<tr>
<th>time</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
</tr>
</thead>
<tbody>
<tr>
<td>ref.</td>
<td>c</td>
<td>a</td>
<td>d</td>
<td>b</td>
<td>e</td>
<td>b</td>
<td>a</td>
<td>b</td>
<td>c</td>
<td>d</td>
</tr>
<tr>
<td>frames</td>
<td>a</td>
<td>a</td>
<td>a</td>
<td>a</td>
<td>a</td>
<td>e</td>
<td>e</td>
<td>e</td>
<td>e</td>
<td>d</td>
</tr>
<tr>
<td></td>
<td>b</td>
<td>b</td>
<td>b</td>
<td>b</td>
<td>b</td>
<td>a</td>
<td>a</td>
<td>a</td>
<td>a</td>
<td>a</td>
</tr>
<tr>
<td></td>
<td>c</td>
<td>c</td>
<td>c</td>
<td>c</td>
<td>c</td>
<td>c</td>
<td>c</td>
<td>b</td>
<td>b</td>
<td>b</td>
</tr>
<tr>
<td></td>
<td>d</td>
<td>d</td>
<td>d</td>
<td>d</td>
<td>d</td>
<td>d</td>
<td>d</td>
<td>c</td>
<td>c</td>
<td>c</td>
</tr>
</tbody>
</table>

- Advantage: simplicity
- Disadvantage: Assumes that pages residing the longest in memory are the least likely to be referenced in the future (does not exploit principle of locality).

Optimal Replacement Algorithm

Algorithm with provably lowest page fault rate of all algorithms:

Replace that page which will not be used for the longest period of time (in the future).

<table>
<thead>
<tr>
<th>time</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
</tr>
</thead>
<tbody>
<tr>
<td>ref.</td>
<td>c</td>
<td>a</td>
<td>d</td>
<td>b</td>
<td>e</td>
<td>b</td>
<td>a</td>
<td>b</td>
<td>c</td>
<td>d</td>
</tr>
<tr>
<td>frames</td>
<td>a</td>
<td>a</td>
<td>a</td>
<td>a</td>
<td>a</td>
<td>a</td>
<td>a</td>
<td>a</td>
<td>a</td>
<td>d</td>
</tr>
<tr>
<td></td>
<td>b</td>
<td>b</td>
<td>b</td>
<td>b</td>
<td>b</td>
<td>b</td>
<td>b</td>
<td>b</td>
<td>b</td>
<td>b</td>
</tr>
<tr>
<td></td>
<td>c</td>
<td>c</td>
<td>c</td>
<td>c</td>
<td>c</td>
<td>c</td>
<td>c</td>
<td>c</td>
<td>c</td>
<td>c</td>
</tr>
<tr>
<td></td>
<td>d</td>
<td>d</td>
<td>d</td>
<td>d</td>
<td>d</td>
<td>e</td>
<td>e</td>
<td>e</td>
<td>e</td>
<td>e</td>
</tr>
<tr>
<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>
Approximation to Optimal: LRU

Least Recently Used: replace the page that has not been accessed for longest period of time (in the past).

<table>
<thead>
<tr>
<th>time</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
</tr>
</thead>
<tbody>
<tr>
<td>ref. string</td>
<td>c</td>
<td>a</td>
<td>d</td>
<td>b</td>
<td>e</td>
<td>b</td>
<td>a</td>
<td>b</td>
<td>c</td>
<td>d</td>
</tr>
<tr>
<td>frames</td>
<td>a</td>
<td>a</td>
<td>a</td>
<td>a</td>
<td>a</td>
<td>a</td>
<td>a</td>
<td>a</td>
<td>a</td>
<td>b</td>
</tr>
<tr>
<td></td>
<td>b</td>
<td>b</td>
<td>b</td>
<td>b</td>
<td>b</td>
<td>b</td>
<td>b</td>
<td>b</td>
<td>b</td>
<td>b</td>
</tr>
<tr>
<td></td>
<td>c</td>
<td>c</td>
<td>c</td>
<td>c</td>
<td>c</td>
<td>e</td>
<td>e</td>
<td>e</td>
<td>e</td>
<td>d</td>
</tr>
<tr>
<td></td>
<td>d</td>
<td>d</td>
<td>d</td>
<td>d</td>
<td>d</td>
<td>d</td>
<td>d</td>
<td>d</td>
<td>c</td>
<td>c</td>
</tr>
</tbody>
</table>

LRU: Implementation

Problem: We need to keep chronological history of page references; need to be reordered upon each reference.

- **Stack:**

  stack

  ![stack diagram]

- **Capacitors:** Associate a capacitor with each memory frame. Capacitor is charged with every reference to the frame. The subsequent exponential decay of the charge can be directly converted into a time interval.

- **Aging registers:** Associate aging register of n bits \( R_{n-1}, \ldots, R_0 \) with each frame in memory. Set \( R_{n-1} \) to 1 for each reference. Periodically shift registers to the right.
Approximation to LRU: Clock Algorithm

Associate a *use_bit* with every frame in memory.
- Upon each reference, set *use_bit* to 1.
- Keep a pointer to first “victim candidate” page.
- To select victim: If current frame’s *use_bit* is 0, select frame and increment pointer. Otherwise delete *use_bit* and increment pointer.

<table>
<thead>
<tr>
<th>time</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
</tr>
</thead>
<tbody>
<tr>
<td>reference string</td>
<td>c</td>
<td>a</td>
<td>d</td>
<td>b</td>
<td>e</td>
<td>b</td>
<td>a</td>
<td>b</td>
<td>c</td>
<td>d</td>
</tr>
<tr>
<td>frames</td>
<td>a/1</td>
<td>a/1</td>
<td>a/1</td>
<td>a/1</td>
<td>a/1</td>
<td>e/1</td>
<td>e/1</td>
<td>e/1</td>
<td>e/1</td>
<td>d/1</td>
</tr>
<tr>
<td></td>
<td>b/1</td>
<td>b/1</td>
<td>b/1</td>
<td>b/0</td>
<td>b/1</td>
<td>b/0</td>
<td>b/1</td>
<td>b/1</td>
<td>b/0</td>
<td></td>
</tr>
<tr>
<td></td>
<td>c/1</td>
<td>c/1</td>
<td>c/1</td>
<td>c/0</td>
<td>c/0</td>
<td>a/1</td>
<td>a/1</td>
<td>a/1</td>
<td>a/0</td>
<td></td>
</tr>
<tr>
<td></td>
<td>d/1</td>
<td>d/1</td>
<td>d/1</td>
<td>d/0</td>
<td>d/0</td>
<td>d/0</td>
<td>d/0</td>
<td>c/1</td>
<td>c/0</td>
<td></td>
</tr>
</tbody>
</table>

Improvement on Clock Algorithm (Second Chance Algorithm)

- Consider read/write activity of page: *dirty_bit* (or *modify_bit*).
- Algorithm same as clock algorithm, except that we scan for frame with both *use_bit* and *dirty_bit* equal to 0.
- Each time the pointer advances, the *use_bit* and *dirty_bit* are updated as follows:

<table>
<thead>
<tr>
<th>ud</th>
<th>ud</th>
<th>ud</th>
<th>ud</th>
</tr>
</thead>
<tbody>
<tr>
<td>before</td>
<td>11</td>
<td>10</td>
<td>01</td>
</tr>
<tr>
<td>after</td>
<td>01</td>
<td>00</td>
<td>00* (select)</td>
</tr>
</tbody>
</table>

- Called *Second Chance* because a frame that has been written to is not removed until two full scans of the list later.
- Note: Other authors (e.g., Stallings) describe a slightly different algorithm!
Improved Clock (cont)

- Example:

<table>
<thead>
<tr>
<th>time</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
</tr>
</thead>
<tbody>
<tr>
<td>reference string</td>
<td>c</td>
<td>a°</td>
<td>d</td>
<td>b°</td>
<td>e</td>
<td>b</td>
<td>a°</td>
<td>b</td>
<td>c</td>
<td>d</td>
</tr>
<tr>
<td>frames</td>
<td>a/10</td>
<td>a/10</td>
<td>a/11</td>
<td>a/11</td>
<td>a/00°</td>
<td>a/00°</td>
<td>a/11</td>
<td>a/11</td>
<td>a/11</td>
<td>a/11</td>
</tr>
<tr>
<td></td>
<td>b/10</td>
<td>b/10</td>
<td>b/10</td>
<td>b/10</td>
<td>b/00°</td>
<td>b/00°</td>
<td>b/10°</td>
<td>b/10°</td>
<td>b/10°</td>
<td>b/10°</td>
</tr>
<tr>
<td></td>
<td>c/10</td>
<td>c/10</td>
<td>c/10</td>
<td>c/10</td>
<td>e/10</td>
<td>e/10</td>
<td>e/10</td>
<td>e/10</td>
<td>e/10</td>
<td>e/10</td>
</tr>
<tr>
<td></td>
<td>d/10</td>
<td>d/10</td>
<td>d/10</td>
<td>d/10</td>
<td>d/00</td>
<td>d/00</td>
<td>d/00</td>
<td>d/00</td>
<td>d/00</td>
<td>c/10</td>
</tr>
</tbody>
</table>

The Macintosh VM Scheme (see Stallings)

- Uses use_bit and modify_bit.

- **Step 1:** Scan the frame buffer. Select first frame with use_bit and modify_bit cleared.

- **Step 2:** If Step 1 fails, scan frame buffer for frame with use_bit cleared and modify_bit set. During scan, clear use_bit on each bypassed frame.

- Now all use_bit’s are cleared. Repeat Step 1 and, if necessary, Step 2.
The Macintosh Scheme (cont)

- Example:

<table>
<thead>
<tr>
<th>time</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
</tr>
</thead>
<tbody>
<tr>
<td>frames</td>
<td>a/10</td>
<td>a/10</td>
<td>a/11</td>
<td>a/11</td>
<td>a/01</td>
<td>a/01</td>
<td>a/11</td>
<td>a/11</td>
<td>a/11</td>
<td>!</td>
</tr>
<tr>
<td></td>
<td>b/10</td>
<td>b/10</td>
<td>b/10</td>
<td>b/11</td>
<td>b/01</td>
<td>b/11</td>
<td>b/11</td>
<td>b/11</td>
<td>b/11</td>
<td>!</td>
</tr>
<tr>
<td></td>
<td>c/10</td>
<td>c/10</td>
<td>c/10</td>
<td>c/10</td>
<td>c/10</td>
<td>c/10</td>
<td>e/10</td>
<td>e/10</td>
<td>e/10</td>
<td>!</td>
</tr>
<tr>
<td></td>
<td>d/10</td>
<td>d/10</td>
<td>d/10</td>
<td>d/00</td>
<td>d/00</td>
<td>d/00</td>
<td>d/00</td>
<td>d/00</td>
<td>d/10</td>
<td>!</td>
</tr>
<tr>
<td>reference string</td>
<td>c</td>
<td>a^w</td>
<td>d</td>
<td>b^w</td>
<td>e</td>
<td>b</td>
<td>a^w</td>
<td>b</td>
<td>c</td>
<td>d</td>
</tr>
</tbody>
</table>

Resident Set Management

- Local vs. Global replacement policy:
  - Local: The page to be replaced is selected from the resident set of pages of the faulting process.
  - Global: The page to be replaced may belong to any of the processes in memory.
- Each program requires a certain minimum set of pages to be resident in memory to run efficiently.
- The size of this set changes dynamically as a program executes.
- This leads to algorithms that attempt to maintain an optimal resident set for each active program. (Page replacement with variable number of frames.)
The Working Set Model

**Working Set** $W(t, \Delta)$: Set of pages referenced by process during time interval $(t-\Delta, t)$.

\[ \|W(t, 1)\| = 1 \quad 1 \leq \|W(t, \Delta)\| \leq \min(\Delta, N) \]

The storage management strategy follows two rules:

**Rule 1:** At each reference, the current working set is determined and only those pages belonging to the working set are retained in memory.

**Rule 2:** A program may run only if its entire current working set is in memory.

**Underlying Assumption:** Size of working set remains constant over small time intervals.

Working Set Model (cont.)

- Example: ($\Delta = 4$)

<table>
<thead>
<tr>
<th>time</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
</tr>
</thead>
<tbody>
<tr>
<td>reference string</td>
<td>e</td>
<td>d</td>
<td>a</td>
<td>c</td>
<td>c</td>
<td>d</td>
<td>b</td>
<td>c</td>
<td>e</td>
<td>e</td>
</tr>
<tr>
<td>working set</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Problems:
- Difficulty in keeping track of working set.
- Estimation of appropriate window size $\Delta$. 
**Improve Paging Performance: Page Buffering**

- Victim frames are not overwritten directly, but are removed from page table of process, and put into:
  - free frame list (clean frames)
  - modified frame list (modified frames)
- Victims are picked from the free frame list in FIFO order.
- If referenced page is in free or modified list, simply reclaim it.
- Periodically (or when running out of free frames) write modified frame list to disk.

**Page Buffering and Page Stealer**

- Kernel process (e.g., pageout in Solaris) swaps out memory frames that are no longer part of a working set of a process, using reference bits.
- Periodically increments age field in valid pages.
- Page stealer wakes up when available free memory is below low-water mark. Swaps out frames until available free memory exceeds high-water mark.
- Page stealer collects frames to swap and swaps them out in a single run. Until then, frames still available for reference.
fork() System Call in Paging Systems

- **Naive**: `fork()` makes a physical copy of parent address space. However, `fork()` mostly followed by an `exec()` call, which overwrites the address space.

- **Lazy Copy**: Use `copy_on_write` bit:
  - During `fork()` system call, only page table is copied. All `copy_on_write` bits of pages are set. If either process writes to the page, incurs protection fault, and, in handling the fault, kernel makes a new copy of the page for the faulting process.

- In practice, this is a bit trickier...

---

Implementation of Demand Paging in UNIX SVR4

<table>
<thead>
<tr>
<th>frame address</th>
<th>age</th>
<th>cp/wrt</th>
<th>mod</th>
<th>ref</th>
<th>val</th>
<th>prot</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>page table entry</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>swap dev</th>
<th>block num</th>
<th>type (swap, file, fill 0, demand fill)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>disk block descriptor</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>page state</th>
<th>ref count</th>
<th>logical device</th>
<th>block number</th>
<th>pfdata pointer</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>frame table entry</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Virtual Memory
Linux Frame Table

- Every page is represented by:

```c
struct page {
    ulong flags;       // dirty, locked, etc.
    atomic_t count;   // reference counter
    struct list_head list;
    struct AS *mapping; // address space associated with page
    ulong index;
    struct list_head lru;
    (pte)
    (private)
    void * virtual;    // virtual address (could be null)
    /* ... etc. */
}
```

Demand Paging on Less-Sophisticated Hardware

- Demand paging most efficient if hardware sets the reference and dirty bits and causes a protection fault when a process writes a page whose copy_on_write bit is set.
- Can duplicate valid bit by a software-valid bit and have the kernel turn off the valid bit. The other bits can then be simulated in software.

Example: Reference Bit:
- If process references a page, it incurs a page fault because valid bit is off. Page fault handler then checks software-valid bit.
- If set, kernel knows that page is really valid and can set software-reference bit.

<table>
<thead>
<tr>
<th>Hardware Valid</th>
<th>Software Valid</th>
<th>Software Reference</th>
<th>Hardware Valid</th>
<th>Software Valid</th>
<th>Software Reference</th>
</tr>
</thead>
<tbody>
<tr>
<td>Off</td>
<td>On</td>
<td>Off</td>
<td>On</td>
<td>On</td>
<td>On</td>
</tr>
</tbody>
</table>

before referencing page after referencing page
Exercise: Implement Copy-on-Write

Given:
- valid bit
- read-only bit

Implement:
- copy-on-write bit