Virtual Memory

- Virtual Memory

Reading: Silbershatz, Chapter 9

Virtual Memory, MIPS-Style

Virtual Memory

- Overview / Motivation
- Simple Approach: Overlays
- Locality of Reference
- Demand Paging
- Policies
  - Placement
  - Replacement
  - Allocation
- Case Studies: Unix SystemV

Reading: Silberschatz, Chapter 9
Virtual Memory

- Allow execution of processes that may not be completely in memory.
  - 1990: Run dBaseIV on MS/DOS without expanded memory.
  - 1995: Run X and Netscape on a Sun with 12MB memory.

- Benefits:
  - Program size not constrained by amount of physical memory available.
  - More programs can be run simultaneously
  - Less need for swapping

Virtual Memory at its Simplest: Overlays

- Keep in memory only those instructions and data that are needed at any given time.
- Special relocation and linking needed to construct overlays.
- Don’t need special support from OS.
- Require proper design of overlay structure.
### 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.

![Demand Paging Diagram](image)

### Mechanics of a Page Fault

1. **CPU** requests a page to be loaded into memory.
2. **trap** mechanism is triggered by the page fault.
3. **OS** checks the page table for the requested page.
4. If the page is on the backing store, **load page** into memory.
5. **update page table** and allocate a new frame.
6. **restart instruction** to continue execution.

![Mechanics of a Page Fault Diagram](image)
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
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.

Performance of Demand Paging

- 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
- Operations during Page Fault:
  1. service page fault interrupt
  2. swap in page
  3. restart process
OS Policies for Virtual Memory

- **Fetch Policy**
  - How/when to get pages into physical memory.
  - Demand paging vs. prepaging.
- **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 Win2k 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.

256kB → 256kB → 256kB → 1024kB

- Example:

<table>
<thead>
<tr>
<th>Frame</th>
<th>Page</th>
<th>V</th>
<th>State</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>K</td>
<td>V</td>
<td>Used</td>
</tr>
<tr>
<td>1</td>
<td>L</td>
<td>V</td>
<td>Used</td>
</tr>
<tr>
<td>2</td>
<td>M</td>
<td>V</td>
<td>Used</td>
</tr>
<tr>
<td>3</td>
<td>N</td>
<td>V</td>
<td>Used</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Frame</th>
<th>Page</th>
<th>V</th>
<th>State</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>A</td>
<td>V</td>
<td>Used</td>
</tr>
<tr>
<td>1</td>
<td>B</td>
<td>V</td>
<td>Used</td>
</tr>
<tr>
<td>2</td>
<td>C</td>
<td>V</td>
<td>Used</td>
</tr>
<tr>
<td>3</td>
<td>D</td>
<td>V</td>
<td>Used</td>
</tr>
</tbody>
</table>

- Mechanics of Page Replacement

- Invoked whenever no free frame can be found.

1. Swap out victim page
2. Invalidate entry for victim page
3. Swap in new page
4. Update entry for new page

- 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>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>b</td>
<td>b</td>
<td>c</td>
<td>c</td>
<td>b</td>
<td>b</td>
<td>a</td>
<td>a</td>
</tr>
<tr>
<td></td>
<td>b</td>
<td>b</td>
<td>c</td>
<td>c</td>
<td>c</td>
<td>c</td>
<td>b</td>
<td>b</td>
<td>d</td>
<td>d</td>
</tr>
<tr>
<td></td>
<td>c</td>
<td>d</td>
<td>c</td>
<td>d</td>
<td>d</td>
<td>d</td>
<td>d</td>
<td>e</td>
<td>e</td>
<td>e</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 lowest page fault rate of all algorithms:

  Replace that page which will not be used for the longest period of time.

- 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>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>b</td>
<td>a</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>e</td>
<td>e</td>
<td>e</td>
<td>e</td>
<td>e</td>
<td>c</td>
<td>c</td>
</tr>
<tr>
<td></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>
<td>e</td>
</tr>
</tbody>
</table>
Approximation to Optimal: LRU

- **Least Recently Used**: replace the page that has not been accessed for longest period of time.
- **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>c</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>a</td>
</tr>
<tr>
<td>a</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>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>e</td>
<td>d</td>
</tr>
<tr>
<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>d</td>
<td>d</td>
<td>c</td>
</tr>
</tbody>
</table>

Frames: a, b, c, d

LRU: Implementation

- **Need to keep chronological history of page references; need to be reordered upon each reference.**
- **Stack**:  

<table>
<thead>
<tr>
<th>stack</th>
<th>?</th>
<th>?</th>
<th>c</th>
<th>a</th>
<th>d</th>
<th>b</th>
<th>e</th>
<th>b</th>
<th>a</th>
<th>b</th>
<th>c</th>
<th>d</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>?</td>
<td>?</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>?</td>
</tr>
<tr>
<td></td>
<td>?</td>
<td>?</td>
<td>?</td>
<td>?</td>
<td>c</td>
<td>a</td>
<td>d</td>
<td>d</td>
<td>e</td>
<td>e</td>
<td>a</td>
<td>b</td>
</tr>
<tr>
<td></td>
<td>?</td>
<td>?</td>
<td>?</td>
<td>?</td>
<td>c</td>
<td>a</td>
<td>a</td>
<td>d</td>
<td>d</td>
<td>e</td>
<td>a</td>
<td>a</td>
</tr>
</tbody>
</table>

- **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}, ..., R_{n}) 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>frames</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>c/1</td>
<td>c/1</td>
<td>d/1</td>
</tr>
<tr>
<td></td>
<td>b/1</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>
</tr>
<tr>
<td></td>
<td>c/1</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>
</tr>
<tr>
<td></td>
<td>d/1</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>c/1</td>
<td>c/1</td>
<td>c/0</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>u</th>
<th>d</th>
<th>u</th>
<th>d</th>
</tr>
</thead>
<tbody>
<tr>
<td>before</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>after</td>
<td>0</td>
<td>1</td>
<td>0</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: Stallings describes a slightly different algorithm!
Improved Clock (cont)

- Example:

<table>
<thead>
<tr>
<th>frames</th>
<th>time</th>
</tr>
</thead>
<tbody>
<tr>
<td>a/10</td>
<td>1</td>
</tr>
<tr>
<td>b/10</td>
<td>2</td>
</tr>
<tr>
<td>c/10</td>
<td>3</td>
</tr>
<tr>
<td>d/10</td>
<td>4</td>
</tr>
<tr>
<td>a/11</td>
<td>5</td>
</tr>
<tr>
<td>b/10</td>
<td>6</td>
</tr>
<tr>
<td>c/10</td>
<td>7</td>
</tr>
<tr>
<td>d/10</td>
<td>8</td>
</tr>
<tr>
<td>a/11</td>
<td>9</td>
</tr>
<tr>
<td>c/10</td>
<td>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>frames</th>
<th>c/10</th>
<th>b/10</th>
<th>c/10</th>
<th>b/10</th>
<th>a/11</th>
<th>a/11</th>
<th>a/11</th>
<th>a/01</th>
<th>a/01</th>
<th>a/11</th>
<th>a/11</th>
<th>a/11</th>
</tr>
</thead>
<tbody>
<tr>
<td>time</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
<td>10</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Resident Set Management

- **Local** vs. **Global** replacement policy:
  - The page to be replaced is selected from the resident set of pages of the **faulting** process. (local)
  - 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:
  - At each reference, the current working set is determined and only those pages belonging to the working set are retained in memory.
  - A program may run only if its entire current working set is in memory.

- Underlying Assumption: cardinality 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>e</td>
<td>d</td>
<td>a</td>
<td>c</td>
<td>c</td>
<td>d</td>
<td>b</td>
<td>e</td>
<td>e</td>
<td>c</td>
<td>a</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) that swaps out memory frames that are no longer part of a working set of a process.
- 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.
Implementation of Demand Paging in UNIX SVR4

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

disk block descriptor

<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>
</table>

frame table entry

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>
</tr>
</thead>
<tbody>
<tr>
<td>Off</td>
<td>On</td>
<td>Off</td>
</tr>
<tr>
<td>before referencing page</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Hardware Valid</th>
<th>Software Valid</th>
<th>Software Reference</th>
</tr>
</thead>
<tbody>
<tr>
<td>On</td>
<td>On</td>
<td>On</td>
</tr>
<tr>
<td>after referencing page</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
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.

- **System V**: Use `copy_on_write` bit:
  - During `fork()` system call, all `copy_on_write` bits of pages of process are turned on. 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.

- **BSD**: Offers `vfork()` system call, which does not copy address space. Tricky! (May corrupt process memory.)

---

Virtual Memory

- Virtual Memory

- **Reading**: Silberschatz, Chapter 9

- Virtual Memory, MIPS-Style
Virtual Memory - MIPS Style

- Split virtual address
- Concatenate more-significant bits with Process ASID to form page address.
- Look in the TLB to see if we find translation entry for page.
- If YES, take high-order physical address bits.
  - (Extra bits stored with PFN control access to frame.)
- If NO, system must locate page entry in main-memory-resident page table, load it into TLB, and start again.

Memory Translation -- VAX style
Memory Translation -- MIPS Style

- In principle: Do the same as VAX, but with as little hardware as possible.
- Apart from register with ASID, the MMU is just a TLB.
- The rest is all implemented in software!
- When TLB cannot translate an address, a special exception (TLB refill) is raised.
- Question: This is easy in principle, but tricky to do efficiently.

MIPS TLB Entry Fields

- **VPN**: higher order bits of virtual address
- **ASID**: identifies the address space
- **G**: if set, disables the matching with the ASID
- **PFN**: Physical frame number
- **N**: 0 - cacheable, 1 - noncacheable
- **D**: write-control bit (set to 1 if writeable)
- **V**: valid bit
MIPS Translation Process

- CPU generates a program (virtual) address on a instruction fetch, a load, or a store.
- The 12 low-end bits are separated off.
- TLB matches key:
  - Matching entry is selected, and PFN is glued to low-order bits of the program address.
  - Valid?: The V and D bits are checked. If problem, raise exception, and set BadVAddr register with offending program address.
  - Cached?: IF C bit is set, the CPU looks in the cache for a copy of the physical location’s data. If C bit is cleared, it neither looks in nor refills the cache.

---

TLB Refill Exception

- Figure out if this was a correct translation. If not, trap to handling of address errors.
- If translation correct, construct TLB entry.
- If TLB already full, select an entry to discard.
- Write the new entry into the TLB.