1 – Virtual Memory

List of Slides

1  Virtual Memory
4  Memory Heirarchy
5  Why Virtual Memory?
6  Virtual Memory Solutions
7  Virtual Memory Issues
8  Locality of Reference
9  Virtual Memory with Segmentation Only
10 Virtual Paged Memory Architecture
11 The Algorithm for Virtual Paged Memory
12 Demand Page Replacement
13 Page Fault Rate and Locality
14 Cost Measures For Paging
15  A Paging Example
16  The Example Continued
17  Locality in Paged Memory
18  FIFO Replacement
19  FIFO Replacement with 3 Pages
20  FIFO Replacement with 4 Pages
21  Belady’s (aka FIFO) Anomaly
22  The Optimal (OPT) Replacement Algorithm
23  Stack Page Replacement Algorithms
24  Least Recently Used (LRU)
25  Clock Replacement Algorithms
26  Simple Clock Replacement
27  Simple Clock (CLOCK)
28  Gold’s Clock Algorithm
29  Gold’s Clock Algorithm
30  Working Set (WS)
31  Working Set Characteristics
32  Working Set Management
33  Lifetime Curves for memory access
35  Page Fault Frequency (PFF)
36  Example IBM MVS
37  IBM MVS Address Space
39  IBM MVS Address Translation
40  Example Architecture Intel Pentium
41  Intel Pentium Memory Protection
42  Protection — Stack Copying
43  Call Gate Access
44  Paging on Pentium
45  Segmentation On Pentium
46  Paged Segmentation on Pentium
47  Another Paged Segmentation Scheme
2 – Memory Heirarchy

Registers

Cache

Main Memory

Magnetic Disk (Cache)

Remote Access | Optical Juke Box | Backups

Figure 1: Heirarchical Memory
3 – Why Virtual Memory?

Users frequently have a speed vs. cost decision. Users want:

1. To extend fast but expensive memory with slower but cheaper memory.
2. To have fast access.
Virtual memory solutions employ:

1. Segmentation
2. Paging
5 – Virtual Memory Issues

Deitel [1, 5] gives the following issues:

1. Fetch Strategy — Retrieval from secondary storage
   (a) Demand — on an as needed basis
   (b) Anticipatory — aka *Prefetching* or *Prepaging*

2. Placement Strategies — Where does the data go?

3. Replacement Strategies — Which page or segment to load and which one gets swapped out?

We seek to avoid overly frequent replacement (called *thrashing*).
6 – Locality of Reference

Deitel [1] describes *locality of reference* by observing that *processes tend to reference storage in nonuniform, highly localized patterns*. Locality can be

1. Temporal — Looping, subroutines, stacks, counter variables.

2. Spatial — Array traversals, sequential code, data encapsulation

Things that hurt locality:

1. Frequently taken conditional branches

2. Linked data structures

3. Random Access patterns in storage
7 – Virtual Memory with Segmentation Only

Segmentation provides protection and relocation so:

1. The operating system can swap out segments not in use.

2. Data objects can be assigned their own data segments they do not fit in an existing data segment.

3. Sharing can be facilitated using the protection mechanism.

4. No internal fragmentation.

5. Main memory has external fragmentation.

Pure segmented solutions are not currently in fashion.

These techniques combine well with paging (as shown later).
Paging is frequently used to increase the logical memory space.

*Page frames* (or frames for short) are the unit of placement/replacement.

A page which has been updated since it was loaded is *dirty*, otherwise it is *clean*.

A page frame may be *locked* in memory (not replaceable). (e.g. O/S Kernel)

Some architectural challenges include:

1. Protection kept for each page.
2. Dirty/clean/locked status kept for each page.
3. Instructions may span pages.
4. An instruction’s operands may span pages.
5. Iterative instructions (e.g. Intel 80x86) may have data spanning many pages.
9 – The Algorithm for Virtual Paged Memory

For each reference do the following:

1. If the page is resident, use it.

2. If there is an available page, allocate it and load the required nonresident page.

3. If there are no available frames then:
   
   (a) Select a page to be removed (the \textit{victim}).
   
   (b) If the victim is dirty write it to secondary storage.

   (c) Load the nonresident page into the victim’s frame.

Stallings [6] uses the term page fault to mean replacement operations. Many others (including your instructor) use page fault to refer to loading any pages (not just replacement).
10 – Demand Page Replacement

Let $M_t$ be the set of resident pages at time $t$. For a program that runs $T$ steps the memory state is:

$$M_0, M_1, M_2, \ldots, M_T$$

Memory is usually empty when starting a process, so: $M_0 = \emptyset$.

Assume that real memory has $m$ page frames, so $|M_t| \leq m$. Let $X_t$ be the set of newly loaded pages and $Y_t$ be the set of newly replaced pages at time $t$. Let $y \in M_t$ be some page in $M_t$ (if nonempty). Each reference updates memory:

$$M_t = M_{t-1} \cup X_t - Y_t$$

$$= \begin{cases} 
M_{t-1} & \text{if } r_t \in M_{t-1} \\
M_{t-1} + r_t & \text{if } r_t \notin M_{t-1} \land |M_{t-1}| < m \\
M_{t-1} + r_t - y & \text{if } r_t \notin M_{t-1} \land |M_{t-1}|r = m
\end{cases}$$

Prefetching has had limited success in practice.
Maekawa et al. use the following notation [4]. The *reference string*, denoted $\omega$, is the sequence of page frames referenced by a process. $\omega$ is indexed by time:

$$\omega = r_1, r_2, r_3, r_{|\omega|}$$  \hspace{1cm} (1)
12 – Cost Measures For Paging

For each level of heirarchy, a page is either:

1. is resident with probability $1 - p$ and can be referenced with cost 1

2. is not resident with probability $p$ and must be loaded with a cost $F$

The **Effective access time** (EAT) for a process is the average time to access a single memory location:

$$EAT = 1 + pF \quad (2)$$

The **Duty Factor** of a process measures the efficiency of the processor use by the process given its page fault pattern.

$$DF = \frac{\mid \omega \mid}{\mid \omega \mid(1 + pF)} = \frac{1}{1 + pF} = \frac{1}{EAT} \quad (3)$$
Consider the following example [4]:

```plaintext
STEPSIZE = 1;
for (i = 1; i <= n; i = i + STEPSIZE) {
    A[i] = B[i] + C[i];
}
```

With the pseudo assembly code:

```
4000 MOV STEPSIZE, 1   # STEPSIZE = 1
4001 MOV R1, STEPSIZE  # i = 1
4002 MOV R2, n          # R2 = n
4003 CMP R1, R2        # test i > n
4004 BGT 4009          # exit loop
4005 MOV R3, B(R1)     # R3 = B[i]
4006 ADD R3, C(R1)     # R3 = B[i] + C[i]
4007 ADD R1, STEPSIZE  # Increment R1
4008 JMP 4002          # Back to test
4009 ...              # After the loop
```
14 – The Example Continued

<table>
<thead>
<tr>
<th>Storage Address</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>6000 – 6FFF</td>
<td>Storage for A</td>
</tr>
<tr>
<td>7000 – 7FFF</td>
<td>Storage for B</td>
</tr>
<tr>
<td>8000 – 8FFF</td>
<td>Storage for C</td>
</tr>
<tr>
<td>9000</td>
<td>Storage for n</td>
</tr>
<tr>
<td>9001</td>
<td>Storage for STEPSIZE</td>
</tr>
</tbody>
</table>

Table 1: Reference Locations

In this example:

\[ \omega = 494944(47484649444)^n \]  

(4)
15 – Locality in Paged Memory

If we have at least 5 frames of 64KB then we know that the program does 5 page loads and no swaps. otherwise how many page faults are there?

Figure 2: Page Trace of Example for $n = 100$
Suppose that we try FIFO replacement (the oldest page gets replaced). Consider the reference string [1]:

\[ \omega = 1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5 \]  \hspace{1cm} (5)

Derive the number of replacements done when:

- 3 frames are used
- 4 frames are used
### Table 2: Solution of FIFO example for 3 Pages

<table>
<thead>
<tr>
<th>Time</th>
<th>0</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>
<th>11</th>
</tr>
</thead>
<tbody>
<tr>
<td>(r_i)</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>1</td>
<td>2</td>
<td>5</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
</tr>
<tr>
<td>Page 1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td>Page 2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>Page 3</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>Fault?</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>
<tr>
<td>Faults</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>6</td>
<td>6</td>
<td>8</td>
<td>9</td>
<td>9</td>
</tr>
</tbody>
</table>
### 18 – FIFO Replacement with 4 Pages

<table>
<thead>
<tr>
<th>Time</th>
<th>0</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>
<th>11</th>
</tr>
</thead>
<tbody>
<tr>
<td>(r_i)</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>1</td>
<td>2</td>
<td>5</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
</tr>
<tr>
<td>Page 1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>Page 2</td>
<td></td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>5</td>
</tr>
<tr>
<td>Page 3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>Page 4</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>
<tr>
<td>Fault?</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>
<tr>
<td>Faults</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
<td>10</td>
</tr>
</tbody>
</table>

**Table 3: Solution of FIFO example for 3 Pages**
19 – Belady’s (aka FIFO) Anamoly

Recall that when we used a FIFO replacement scheme with 3 frames we got 9 faults and for 4 frames we got 10 faults.

Usually one would expect fewer faults for a larger number of frames.

That did not occur, what happened? Belady discovered this (counter intuitive) problem.
The optimal solution (OPT) is to select the victim such that it is the page that will be referenced the longest time into the future.

<table>
<thead>
<tr>
<th>Time</th>
<th>0</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>
<th>11</th>
</tr>
</thead>
<tbody>
<tr>
<td>( r_i )</td>
<td>2</td>
<td>3</td>
<td>2</td>
<td>1</td>
<td>5</td>
<td>2</td>
<td>4</td>
<td>5</td>
<td>3</td>
<td>2</td>
<td>5</td>
<td>2</td>
</tr>
<tr>
<td>Page 1</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>Page 2</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>Page 3</td>
<td>1</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Fault?</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>
<tr>
<td>Faults</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>4</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>6</td>
<td>6</td>
<td>6</td>
</tr>
</tbody>
</table>

Table 4: An OPT example using 3 Pages [6]

Note at time \( t = 9 \) Page 1 could have been the victim too.
21 – Stack Page Replacement Algorithms

Let \( M(m, \omega) \) represent the set of pages in real memory after processing reference string \( \omega \) in \( m \) frames.

The inclusion property is satisfied if for a given page replacement algorithm:
\[
M(\omega, m) \subseteq M(\omega, m + 1).
\]

*Stack replacement algorithms* [4] satisfy the inclusion property.

e.g. FIFO is NOT a stack page replacement algorithm (inclusion is NOT satisfied), while OPT is.
LRU is a popular stack page replacement strategy. Pages are ordered by time of last access, with the page used furthest into the past being removed.

<table>
<thead>
<tr>
<th>Time</th>
<th>0</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>
<th>11</th>
</tr>
</thead>
<tbody>
<tr>
<td>$r_i$</td>
<td>2</td>
<td>3</td>
<td>2</td>
<td>1</td>
<td>5</td>
<td>2</td>
<td>4</td>
<td>5</td>
<td>3</td>
<td>2</td>
<td>5</td>
<td>2</td>
</tr>
<tr>
<td>Page 1</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>Page 2</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td></td>
</tr>
<tr>
<td>Page 3</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Fault?</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>
<tr>
<td>Faults</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>4</td>
<td>5</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>7</td>
<td></td>
</tr>
</tbody>
</table>

Table 5: An LRU example using 3 Pages [6]
Clock replacement algorithms are stack algorithms. The simplest style marks pages with \( use = 1 \) when referenced and \( use = 0 \) upon a circular scan.

Figure 3: Simple Clock Before Replacement [6]
24 – Simple Clock Replacement

The first page encountered with \( use = 0 \) is selected. The final position of the previous scan is used to start the next scan.

Figure 4: Simple Clock After Replacement [6]
25 – Simple Clock (CLOCK)

CLOCK is a popular stack page replacement strategy, pages in use are underlined.

<table>
<thead>
<tr>
<th>Time</th>
<th>0</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>
<th>11</th>
</tr>
</thead>
<tbody>
<tr>
<td>( r_i )</td>
<td>2</td>
<td>3</td>
<td>2</td>
<td>1</td>
<td>5</td>
<td>2</td>
<td>4</td>
<td>5</td>
<td>3</td>
<td>2</td>
<td>5</td>
<td>2</td>
</tr>
<tr>
<td>Page 1</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>Page 2</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>Page 3</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>5</td>
<td>5</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Hand</td>
<td>2</td>
<td>3</td>
<td>3</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Fault?</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>
<tr>
<td>Faults</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>6</td>
<td>7</td>
<td>7</td>
<td>8</td>
<td>8</td>
</tr>
</tbody>
</table>

Table 6: A CLOCK example using 3 Pages [6]
Gold’s algorithm, also called Not Used Recently (NUR) prefers clean pages as victims.

This algorithm marks pages using a modification bit, with $m = 1$ if updated, otherwise $m = 0$. Use bits indicate read accesses and are denoted $u = 1$ for recent read otherwise $u = 0$.

The steps are:

1. Scan for a page with $u = 0, m = 0$, If one is found stop, otherwise after all pages have been scanned continue.

2. Scan for a page with $u = 0, m = 1$, setting $u = 0$ as the scanning. If one is found stop, otherwise continue.

3. Repeat step 1, pages with $u = 1$ before now have $u = 0$ so step 1 or step 2 will be satisfied.
Figure 5: Gold’s Clock Replacement Algorithm [6]
Working set algorithms are stack algorithms using a parameter $\Delta$ [4, 6, 1]. All pages within the last $\Delta$ references remain resident. The number of pages allocated varies over time (upper bound is $\Delta$).

![Diagram of Working Set Abstraction](image)

The pages referenced by the process during this time interval constitute the process's working set $W(t, w)$.

Figure 6: Working Set Abstraction [1], $w = \Delta$
29 – Working Set Characteristics

Working sets can be characterized as being stable most of the time, and growing larger as a program makes transitions between locals of reference. Choosing \( \Delta \) or \( (w) \) is hard.

![Graph showing the number of primary storage pages allocated to a process over time with working set characteristics labeled.]

Figure 7: Working Set Size \( (w) \) over Time [1]
30 – Working Set Management

Since each process has a dynamically sized working set, and the window, $\Delta$, is too loose an upper bound on working set size, the OS may have to select a process to deactivate.

Maekawa et al. suggest [4] selection criteria:

1. The lowest priority process
2. The process with the largest fault rate
3. The process with the largest working set
4. The process with the smallest working set
5. The most recently activated process
6. The largest remaining quantum
31 – Lifetime Curves for memory access

Similar to inter I/O event times, there are inter page fault times. The inter page fault time is called the lifetime of the page. Lifetime curves are plotted as a function of number of pages of memory allocated ($m$) and typically have a “knee” (performance region) where $L(m)/m$ is maximal (a good value of $m$).
Figure 8: Lifetime $L(m)$ as a function of $m$ [4]
Page fault frequency measures the time since the last page and tracks whether each page allocated to a process (via a use bit) has been accessed since the last page fault.

This information is used, processes with high fault rates are allocated more pages, while jobs with lower fault rates release pages.
IBM’s MVS uses fixed sized partitions called *segments* which are either 64KB or 1MB. Pages are 2 or 4 KB. Sizes depend on the architecture.

![Diagram of MVS Memory Management Internals]

**Figure 9: MVS Memory Management Internals [6]**
IBM’s MVS has a variety of flavors which have developed over time, and have had evolving memory management models.
Figure 10: MVS Memory Layout [6]
IBM’s MVS supports relocation via an address translation scheme.

Figure 11: MVS Address Translation [6]
36 – Example Architecture Intel Pentium

Figure 12: Architectural support for O/S’s [3]
Figure 13: Memory Protection on Intel Pentium [2]
38 – Protection — Stack Copying

The Intel Pentium(R) copies user application space into kernel space. Calls to routines of different privilege pass through call gates. Data gets copied on the stack.

![Stack Copying Diagram](image)

**Figure 14: Stack Copying on Pentium [3]**
39 – Call Gate Access

Call gate access imposes strict control on access across privilege levels.

Figure 15: Call Gate Access [3]
Paging on Pentium allows user selected mixed page sizes, with 4 MB page frames used for kernel access (reducing tlb misses), and pages on the order of 4 KB for applications programs. Unix implementations frequently apply protection at the page level without segmentation.

Figure 16: Page Table Structure [3]
41 – Segmentation On Pentium

The Intel 80x86 architectures are segmented. Operating systems can use this

Figure 17: Segmentation on Intel Pentium [3]
42 – Paged Segmentation on Pentium

Segmentation can be used to create a virtual address as input into a paged addressing scheme. Segmentation provides fine grained protection. Paging provides virtual memory.

Figure 18: Paged Segmentation on Intel Pentium [3]
43 – Another Paged Segmentation Scheme

One simple approach (assuming you have large segments) is to make a page table for each segment (the Intel Pentium(R) supports this).

Figure 19: Assigning a Page Table to each Segment [3]
References


