Chapter 9: Virtual Memory
**Background**

- **Virtual memory** – separation of user logical memory from physical memory.
  - Only part of the program needs to be in memory for execution.
  - Logical address space can therefore be much larger than physical address space.
  - Allows address spaces to be shared by several processes.
  - Allows for more efficient process creation.

- Virtual memory can be implemented via:
  - Demand paging
  - Demand segmentation
Virtual Memory That is Larger Than Physical Memory

- Page 0
- Page 1
- Page 2

Virtual memory

⇒

Memory map

Physical memory
Virtual-address Space
Shared Library Using Virtual Memory

The diagram illustrates the concept of shared libraries using virtual memory. It shows the layout of memory regions for two processes, with shared pages in the middle connecting the two.

- **Stack**: Located at the top of the memory for each process.
- **Shared Library**: Located below the stack in each process.
- **Heap**: Located below the shared library in each process.
- **Data**: Located below the heap in each process.
- **Code**: Located at the bottom of the memory in each process.

The dashed lines indicate the shared pages that are accessible to both processes.
**Demand Paging**

- Bring a page into memory only when it is needed
  - Less I/O needed
  - Less memory needed
  - Faster response
  - More users

- Page is needed $\Rightarrow$ reference to it
  - invalid reference $\Rightarrow$ abort
  - not-in-memory $\Rightarrow$ bring to memory
Transfer of a Paged Memory to Contiguous Disk Space

The diagram illustrates the process of transferring a paged memory to contiguous disk space. The memory is divided into pages, and the diagram shows the pages being swapped out from the main memory to disk space and then swapped back in. The disk space is organized in a cylinder, and the pages are moved from the memory to disk and vice versa as needed.
Valid-Invalid Bit

- With each page table entry a valid–invalid bit is associated (1 ⇒ in-memory, 0 ⇒ not-in-memory)
- Initially valid–invalid but is set to 0 on all entries
- Example of a page table snapshot:

<table>
<thead>
<tr>
<th>Frame #</th>
<th>valid-invalid bit</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>...</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

- During address translation, if valid–invalid bit in page table entry is 0 ⇒ page fault
Page Table When Some Pages Are Not in Main Memory

![Page Table Diagram]

The diagram illustrates a page table when some pages are not in main memory. Each page in the logical memory is represented by a box labeled with a letter (A to H). The page table indicates which pages are valid (V) or invalid (I) in main memory. The physical memory is represented by a series of boxes, some of which are filled to indicate which pages are present. The valid-invalid bit indicates the status of each page in the main memory.
Page Fault

- If there is ever a reference to a page, first reference will trap to OS ⇒ page fault
- OS looks at another table to decide:
  - Invalid reference ⇒ abort.
  - Just not in memory.
- Get empty frame.
- Swap page into frame.
- Reset tables, validation bit = 1.
- Restart instruction: Least Recently Used
  - block move
  - auto increment/decrement location
Steps in Handling a Page Fault

1. reference
2. trap
3. page is on backing store
4. bring in missing page
5. reset page table
6. restart instruction
What happens if there is no free frame?

- Page replacement – find some page in memory, but not really in use, swap it out
  - algorithm
  - performance – want an algorithm which will result in minimum number of page faults
- Same page may be brought into memory several times
Performance of Demand Paging

- Page Fault Rate $0 \leq p \leq 1.0$
  - if $p = 0$ no page faults
  - if $p = 1$, every reference is a fault

- Effective Access Time (EAT)
  
  $\text{EAT} = (1 - p) \times \text{memory access} + p \times (\text{page fault overhead}) + \text{swap page out} + \text{swap page in} + \text{restart overhead}$
Demand Paging Example

- Memory access time = 1 microsecond

- 50% of the time the page that is being replaced has been modified and therefore needs to be swapped out

- Swap Page Time = 10 msec = 10,000 msec

  \[ EAT = (1 - p) \times 1 + p \times 15000 \]

  \[ 1 + 15000P \] (in msec)
Process Creation

- Virtual memory allows other benefits during process creation:
  - Copy-on-Write
  - Memory-Mapped Files (later)
Copy-on-Write (COW) allows both parent and child processes to initially share the same pages in memory. If either process modifies a shared page, only then is the page copied. COW allows more efficient process creation as only modified pages are copied. Free pages are allocated from a pool of zeroed-out pages.
Page Replacement

- Prevent over-allocation of memory by modifying page-fault service routine to include page replacement

- Use modify (dirty) bit to reduce overhead of page transfers – only modified pages are written to disk

- Page replacement completes separation between logical memory and physical memory – large virtual memory can be provided on a smaller physical memory
Need For Page Replacement

- Logical memory for user 1:
  - Frame 0: H
  - Frame 1: load M
  - Frame 2: J
  - Frame 3: M

- Logical memory for user 2:
  - Frame 0: A
  - Frame 1: B
  - Frame 2: D
  - Frame 3: E

- Page table for user 1:
  - Frame 3: valid
  - Frame 4: invalid

- Page table for user 2:
  - Frame 6: valid
  - Frame 7: invalid

- Physical memory:
  - Frame 0: monitor
  - Frame 1: D
  - Frame 2: H
  - Frame 3: load M
  - Frame 4: J
  - Frame 5: A
  - Frame 6: E
  - Frame 7: M

- Valid-invalid bit:
  - Frame 3: valid (v)
  - Frame 4: invalid (i)
Basic Page Replacement

1. Find the location of the desired page on disk

2. Find a free frame:
   - If there is a free frame, use it
   - If there is no free frame, use a page replacement algorithm to select a victim frame

3. Read the desired page into the (newly) free frame. Update the page and frame tables.

4. Restart the process
Page Replacement

1. Swap out victim page
2. Change to invalid
3. Swap desired page in
4. Reset page table for new page
Page Replacement Algorithms

- Want lowest page-fault rate
- Evaluate algorithm by running it on a particular string of memory references (reference string) and computing the number of page faults on that string
- In all our examples, the reference string is
  1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5
Graph of Page Faults Versus The Number of Frames

![Graph of Page Faults](image_url)
First-In-First-Out (FIFO) Algorithm

- Reference string: 1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5
- 3 frames (3 pages can be in memory at a time per process)

```
1  1  4  5
2  2  1  3  9 page faults
```

- 4 frames

```
1  1
2  2  1  3
3  3  2  4
```

```
1  1  5  4
2  2  1  5  10 page faults
3  3  2
4  4  3
```

- FIFO Replacement – Belady’s Anomaly
  - more frames $\Rightarrow$ more page faults
**FIFO Page Replacement**

<table>
<thead>
<tr>
<th>reference string</th>
<th>page frames</th>
</tr>
</thead>
<tbody>
<tr>
<td>7 0 1 2 0 3 0 4 2 3 0 3 2 1 2 0 1 7 0 1</td>
<td>7 7 7 0 0 1 1</td>
</tr>
<tr>
<td>7 7 7 2 2 3 3 3 2 2 2 1 1 1 0 0 3 3</td>
<td>7 7 7 0 0 1 1</td>
</tr>
<tr>
<td>0 0 0 3 3 2 2 2 1 2 2 3 2 2 1</td>
<td>1 0 0 3 2 2 1</td>
</tr>
</tbody>
</table>
FIFO Illustrating Belady’s Anomaly
Optimal Algorithm

- Replace page that will not be used for longest period of time
- 4 frames example

1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5

![Diagram showing page faults](image)

6 page faults

- How do you know this?
- Used for measuring how well your algorithm performs
## Optimal Page Replacement

### 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>2</th>
<th>2</th>
<th>2</th>
<th>2</th>
<th>2</th>
<th>2</th>
<th>2</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>4</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>
Least Recently Used (LRU) Algorithm

- Reference string: 1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- 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 determine which are to change
LRU Page Replacement

<table>
<thead>
<tr>
<th>Reference String</th>
<th>Page Frames</th>
</tr>
</thead>
<tbody>
<tr>
<td>7 0 1 2 0 3 0 4 2 3 0 3 2 1 2 0 1 7 0 1</td>
<td>7 7 7 2 2 4 4 4 0 1 1 1 3 0 0 0 0 3 3 3 2 2 2 2 2 2 7</td>
</tr>
</tbody>
</table>

...
LRU Algorithm (Cont.)

- 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
  - No search for replacement
Use Of A Stack to Record The Most Recent Page References

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

<table>
<thead>
<tr>
<th>Stack Before</th>
</tr>
</thead>
<tbody>
<tr>
<td>2 1 0 7 4</td>
</tr>
<tr>
<td>a</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Stack After</th>
</tr>
</thead>
<tbody>
<tr>
<td>7 2 1 0 4</td>
</tr>
<tr>
<td>b</td>
</tr>
</tbody>
</table>
LRU Approximation Algorithms

- Reference bit
  - With each page associate a bit, initially = 0
  - When page is referenced bit set to 1
  - Replace the one which is 0 (if one exists). We do not know the order, however.

- Second chance
  - Need reference bit
  - Clock replacement
  - If page to be replaced (in clock order) has reference bit = 1 then:
    - set reference bit 0
    - leave page in memory
    - replace next page (in clock order), subject to same rules
Second-Chance (clock) Page-Replacement Algorithm

- Reference bits
- Pages

(a) Circular queue of pages

(b) Circular queue of pages

Next victim
Counting Algorithms

- Keep a counter of the number of references that have been made to each page

- **LFU Algorithm**: replaces page with smallest count

- **MFU Algorithm**: based on the argument that the page with the smallest count was probably just brought in and has yet to be used
Allocation of Frames

- Each process needs *minimum* number of pages
- Example: IBM 370 – 6 pages to handle SS MOVE instruction:
  - instruction is 6 bytes, might span 2 pages
  - 2 pages to handle *from*
  - 2 pages to handle *to*
- Two major allocation schemes
  - fixed allocation
  - priority allocation
Fixed Allocation

- Equal allocation – For example, if there are 100 frames and 5 processes, give each process 20 frames.

- Proportional allocation – Allocate according to the size of process
  - $s_i = \text{size of process } p_i$
  - $S = \sum s_i$
  - $m = \text{total number of frames}$
  - $a_i = \text{allocation for } p_i = \frac{s_i}{S} \times m$

  $m = 64$
  $s_i = 10$
  $s_2 = 127$
  $a_1 = \frac{10}{137} \times 64 \approx 5$
  $a_2 = \frac{127}{137} \times 64 \approx 59$
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
  - 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
- **Local replacement** – each process selects from only its own set of allocated frames
Thrashing

- If a process does not have “enough” pages, the page-fault rate is very high. This leads to:
  - low CPU utilization
  - operating system thinks that it needs to increase the degree of multiprogramming
  - another process added to the system

- **Thrashing** ≡ a process is busy swapping pages in and out
Thrashing (Cont.)

![Graph showing CPU utilization vs degree of multiprogramming with a threshold for thrashing.]

- [Operating System Concepts](#) 9.41
- Silberschatz, Galvin and Gagne ©2005
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?
  $\sum$ size of locality > total memory size
Locality In A Memory-Reference Pattern
Working-Set Model

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

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

- $D = \sum WSS_i \equiv$ total demand frames

  - if $D > m \Rightarrow$ Thrashing

  - Policy if $D > m$, then suspend one of the processes
### Working-set model

<table>
<thead>
<tr>
<th>Page Reference Table</th>
</tr>
</thead>
<tbody>
<tr>
<td>. . . 2 6 1 5 7 7 7 7 5 1 6 2 3 4 1 2 3 4 4 3 4 3 4 4 4 1 3 2 3 4 4 4 3 4 4 4 . . .</td>
</tr>
</tbody>
</table>

- **\( \Delta \)**
- **\( t_1 \)**
- **\( t_2 \)**

\[
\text{WS}(t_1) = \{1, 2, 5, 6, 7\} \quad \text{WS}(t_2) = \{3, 4\}
\]
Keeping Track of the Working Set

- Approximate with interval timer + a reference bit
- Example: $\Delta = 10,000$
  - Timer interrupts after every 5000 time units
  - Keep in memory 2 bits for each page
  - Whenever a timer interrupts copy and sets the values of all reference bits to 0
  - If one of the bits in memory = 1 $\Rightarrow$ page in working set
- Why is this not completely accurate?
- Improvement = 10 bits and interrupt every 1000 time units
Page-Fault Frequency Scheme

- Establish “acceptable” page-fault rate
  - If actual rate too low, process loses frame
  - If actual rate too high, process gains frame

![Graph showing the relationship between page-fault rate and number of frames. The graph includes two bounds: an upper bound and a lower bound. Points on the graph indicate that increasing the number of frames decreases the page-fault rate, and vice versa.](image)
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.

- 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 file access by treating file I/O through memory rather than `read()` `write()` system calls.

- Also allows several processes to map the same file allowing the pages in memory to be shared.
Memory Mapped Files
import java.io.*;
import java.nio.*;
import java.nio.channels.*;
public class MemoryMapReadOnly
{
    // Assume the page size is 4 KB
    public static final int PAGE_SIZE = 4096;
    public static void main(String args[]) throws IOException {
        RandomAccessFile inFile = new RandomAccessFile(args[0],"r");
        FileChannel in = inFile.getChannel();
        MappedByteBuffer mappedBuffer =
            in.map(FileChannel.MapMode.READ_ONLY, 0, in.size());
        long numPages = in.size() / (long)PAGE_SIZE;
        if (in.size() % PAGE_SIZE > 0)
            ++numPages;
Memory-Mapped Files in Java (cont)

// we will "touch" the first byte of every page
int position = 0;
for (long i = 0; i < numPages; i++) {
    byte item = mappedBuffer.get(position);
    position += PAGE_SIZE;
}
in.close();
inFile.close();

■ The API for the map() method is as follows:
map(mode, position, size)
Other Issues -- 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 $\alpha$ of the pages is used
    - Is cost of $s \cdot \alpha$ save pages faults > or < than the cost of prepaging $s \cdot (1 - \alpha)$ unnecessary pages?
    - $\alpha$ near zero $\Rightarrow$ prepaging loses
Other Issues – Page Size

- Page size selection must take into consideration:
  - fragmentation
  - table size
  - I/O overhead
  - locality
Other Issues – TLB Reach

- TLB Reach - The amount of memory accessible from the TLB
- TLB Reach = (TLB Size) × (Page Size)
- Ideally, the working set of each process is stored in the TLB. Otherwise there is a high degree of page faults.
- Increase the Page Size. This may lead to an increase in fragmentation as not all applications require a large page size
- Provide Multiple Page Sizes. This allows applications that require larger page sizes the opportunity to use them without an increase in fragmentation.
Program structure

- \texttt{Int[128,128] data;}
- Each row is stored in one page
- Program 1

\begin{verbatim}
for (j = 0; j < 128; j++)
    for (i = 0; i < 128; i++)
        data[i,j] = 0;
\end{verbatim}

128 x 128 = 16,384 page faults

- Program 2

\begin{verbatim}
for (i = 0; i < 128; i++)
    for (j = 0; j < 128; j++)
        data[i,j] = 0;
\end{verbatim}

128 page faults
Other Issues – I/O interlock

- **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.
Reason Why Frames Used For I/O Must Be In Memory
Operating System Examples

- Windows XP
- Solaris
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
Solaris

- Maintains a list of free pages to assign faulting processes
- *Lotsfree* – threshold parameter (amount of free memory) to begin paging
- *Desfree* – threshold parameter to increasing paging
- *Minfree* – threshold parameter to being swapping
- Paging is performed by *pageout* process
- Pageout scans pages using modified clock algorithm
- *Scanrate* is the rate at which pages are scanned. This ranges from *slowscan* to *fastscan*
- Pageout is called more frequently depending upon the amount of free memory available
Solaris 2 Page Scanner

- 8192 fastscan
- 100 slowscan

scan rate vs. amount of free memory:

- minfree
- desfree
- lotsfree
End of Chapter 9