Chapter 9: Virtual Memory
Chapter 9: Virtual Memory

- Background
- Demand Paging
- Page Replacement
- Thrashing
Objectives

- To describe the benefits of a virtual memory system

- To explain the concepts of demand paging, page-replacement algorithms, and allocation of page frames

- To discuss the thrashing problem.
Virtual Memory

- Virtual memory is a technique that allows the execution of processes that are not completely in memory.
  - One major advantage of this scheme is that programs can be larger than physical memory.
  - Further, virtual memory abstracts main memory into an extremely large, uniform array of storage, separating logical memory as viewed by the user from physical memory.
Virtual memory involves the 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

![Diagram showing virtual memory pages mapping to physical memory]

- Page 0
- Page 1
- Page 2
- Virtual memory
- Page ν
- Memory map
- Physical memory
Virtual-address Space

- Stack
- Heap
- Data
- Code
Shared Library Using Virtual Memory

Diagram showing the structure of memory with stack, shared library, heap, data, and code layers. The shared library is connected to the heap and data layers through dashed lines, indicating virtual memory sharing.
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

- **Lazy swapper** – never swaps a page into memory unless page will be needed
  - Swapper that deals with pages is a **pager**
Transfer of a Paged Memory to Contiguous Disk Space
Valid-Invalid Bit

- With each page table entry a valid–invalid bit is associated ($v \Rightarrow$ in-memory, $i \Rightarrow$ not-in-memory)
- Initially valid–invalid bit is set to $i$ on all entries
- Example of a page table snapshot:

```
Frame # | valid-invalid bit
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>$v$</td>
</tr>
<tr>
<td></td>
<td>$v$</td>
</tr>
<tr>
<td></td>
<td>$v$</td>
</tr>
<tr>
<td></td>
<td>$v$</td>
</tr>
<tr>
<td></td>
<td>$v$</td>
</tr>
<tr>
<td></td>
<td>$i$</td>
</tr>
<tr>
<td>....</td>
<td></td>
</tr>
<tr>
<td></td>
<td>$i$</td>
</tr>
<tr>
<td></td>
<td>$i$</td>
</tr>
</tbody>
</table>
```

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

Logical memory:

- A
- B
- C
- D
- E
- F
- G
- H

Page table:

- Frame 0: Valid (v)
- Frame 1: Invalid (i)
- Frame 2: Valid (v)
- Frame 3: Invalid (i)
- Frame 4: Valid (v)
- Frame 5: Invalid (i)
- Frame 6: Valid (v)
- Frame 7: Invalid (i)

Physical memory:

- Frame 0
- Frame 1
- Frame 2
- Frame 3
- Frame 4
- Frame 5
- Frame 6
- Frame 7

- Frame 8
- Frame 9
- Frame 10
- Frame 11
- Frame 12
- Frame 13
- Frame 14
- Frame 15

Legend:

- A
- B
- C
- D
- E
- F
- G
- H

Valid-invalid bit:

- Frame 0: Valid (v)
- Frame 1: Invalid (i)
- Frame 2: Valid (v)
- Frame 3: Invalid (i)
- Frame 4: Valid (v)
- Frame 5: Invalid (i)
- Frame 6: Valid (v)
- Frame 7: Invalid (i)
Page Fault

- If there is a reference to a page, first reference to that page will trap to operating system:

  page fault

1. Operating system looks at another table to decide:
   - Invalid reference ⇒ abort
   - Just not in memory
2. Get empty frame
3. Swap page into frame
4. Reset tables
5. Set validation bit = v
6. Restart the instruction that caused the page fault
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

Load M

Operating system

Physical memory

Free frame

Page table

Silberschatz, Galvin and Gagne ©2009
Virtual to Physical Address Translation (Mapping) Algorithm In The Paging Scheme

- The machine uses TLB (translation look-aside buffer) in the cache and PT (page tables) in the main memory.

- Given the page size (= frame size), the virtual addresses generated by the CPU which consists of: Page #, offset (p,d) and the access type(AT): Read-only(R), Read-Write(RW), or Execute-only (E), use the following algorithm:
IF \( p \geq PTLR \) THEN trap ("Invalid page number")

// * page# >= # of pages for this process *//

IF \( p \) in TLB THEN

IF NOT AT in protection THEN trap ("memory-protection violation")

ELSE

"Cache Hit"

Physical add. = frame# * page(frame) size + d

ENDIF

ELSE

ELSEIF in PT presence/absence bit (valid/invalid bit) = present (valid) THEN

// * this page is currently loaded in a frame of the main memory *//

IF NOT AT in protection THEN trap ("memory-protection violation")

ELSE

"Cache Miss"

Physical add. = frame# * page(frame) size + d

ENDIF

ELSE

// * this page is not in the main memory, it's in the disk. must be swapped in according to the paging replacement algorithm *//

Trap ("page fault") ENDIF
Performance of Demand Paging

- Demand paging can significantly affect the performance of a computer system. Let's compute the effective access time for a demand-paged memory.

  - For most computer systems, the memory-access time (ma) ranges from 10 to 200 nanoseconds.
  - As long as we have no page faults, the effective access time is equal to the memory access time.
  - If, however a page fault occurs, we must first read the relevant page from disk and then access the desired word.

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

- Memory access time = 200 nanoseconds

- Average page-fault service time = 8 milliseconds

- \[ EAT = (1 - p) \times 200 + p \times 8 \text{ milliseconds} \]
  \[ = (1 - p) \times 200 + p \times 8,000,000 \]
  \[ = 200 + p \times 7,999,800 \]

- If one access out of 1,000 causes a page fault, then
  \[ EAT = 8.2 \text{ microseconds.} \]
  This is a slowdown by a factor of 40!!
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
Page Replacement

- If we increase the degree of multiprogramming we are **over-allocating** memory.
- Prevent over-allocation of memory by modifying page-fault service routine to include page replacement
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.

- But if we use demand paging we must solve Two major problems:
  - Develop Frame-allocation algorithms, if we have multiple processes in memory, we must decide how many frames to allocate to each process.
  - Develop Page-replacement algorithms, When page replacement is required, we must select the frames that are to be replaced.
Need For Page Replacement
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. Bring the desired page into the (newly) free frame; update the page and frame tables

4. Restart the process
Page Replacement
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
FIFO Page Replacement

- The simplest page-replacement algorithm is a first-in, first-out (FIFO) algorithm.

- A FIFO replacement algorithm associates with each page the time when that page was brought into memory (FIFO queue).

- When a page must be replaced, the oldest page is chosen.

- The FIFO page-replacement algorithm is easy to understand and program. However, its performance is not always good.
FIFO Page Replacement

there are 15 faults altogether
## 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)

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

- 9 page faults

## 4 frames

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

- 10 page faults

- Belady’s Anomaly: more frames ⇒ more page faults
FIFO Illustrating Belady’s Anomaly

The graph illustrates the number of page faults as a function of the number of frames. It shows a decrease in page faults as the number of frames increases, highlighting the phenomenon of Belady’s Anomaly.
An optimal page-replacement algorithm has the lowest page-fault rate of all algorithms (called OPT or MIN). It is simply this:

- Replace the 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

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

6 page faults

How do you know this?

Used for measuring how well your algorithm performs
Optimal Page Replacement

reference string

<p>| | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>7</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>0</td>
<td>3</td>
<td>0</td>
<td>4</td>
</tr>
</tbody>
</table>

<p>| | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>7</td>
<td>0</td>
</tr>
</tbody>
</table>

page frames

<p>| | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>7</td>
<td>7</td>
<td>7</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

<p>| | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></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>
</tr>
</tbody>
</table>

<p>| | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<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>
</tr>
</tbody>
</table>

9 page faults
Least Recently Used (LRU) Algorithm

- replace the page that has not been used for the longest period of time
- Reference string: 1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5

```
   1  1  1  1  5  
   2  2  2  2  2  
   3  5  5  4  4  
   4  4  3  3  3  
```

- Counter implementation
  - Every page entry has a time-of-use field; every time page is referenced, copy the CPU clock/counter into the time-of-use field
  - When a page needs to be replaced, look at the time-of-use field values to determine which page should be replaced
LRU Page Replacement

reference string

12 page faults
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

reference string

4  7  0  7  1  0  1  2  1  2  7  1  2

stack before a

2
1
0
7
4

stack after b

7
2
1
0
4

a  b
Counting Algorithms

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

- **The least frequently used (LFU) Algorithm**: replaces page with smallest count

- **The most frequently used (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

- How do we allocate the fixed amount of free memory among the various processes?
- Each process needs *minimum* number of pages
- Example: IBM 370 – 6 pages to handle Storage location to storage location 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

- The easiest way to split \( m \) frames among \( n \) processes is to give everyone an equal share, \( m/n \) frames. This scheme is called Equal allocation.
- For example, if there are 100 frames and 5 processes, give each process 20 frames.
Proportional allocation

we allocate available memory to each process according to its size.

\[ s_i = \text{size of process } p_i \]
\[ S = \sum s_i \]
\[ m = \text{total number of frames} \]
\[ a_i = \text{allocation for } p_i = \frac{s_i}{S} \times m \]

\[ m = 64 \]
\[ s_1 = 10 \]
\[ s_2 = 127 \]
\[ a_1 = \frac{10}{137} \times 64 \approx 5 \]
\[ a_2 = \frac{127}{137} \times 64 \approx 59 \]
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

With a local replacement strategy, the number of frames allocated to a process does not change.

With global replacement, a process may happen to select only frames allocated to other processes, thus increasing the number of frames allocated to it (assuming that other processes do not choose its frames for replacement).

Global replacement generally results in greater system throughput and is therefore the more common method.
Thrashing

- If a process does not have "enough" frames, the page-fault rate is very high.
  - If the process does not have the number of frames it needs to support pages in active use, it will quickly page-fault.
  - At this point, it must replace some page.
  - However, since all its pages are in active use, it must replace a page that will be needed again right away.
  - Consequently, it quickly faults again, and again, and again, replacing pages that it must bring back in immediately.

- **Thrashing** \( \equiv \) a process is busy swapping pages in and out, it is spending more time paging than executing.
Cause of thrashing

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 (Cont.)

![Graph showing the relationship between CPU utilization and degree of multiprogramming, indicating thrashing at a certain point.]
Demand Paging and Thrashing

- Why does demand paging work?
  Locality model
  - A process is composed of several different localities. Process migrates from one locality to another.
  - A locality is a set of pages that are actively used together.
  - Localities may overlap

- Why does thrashing occur?
  \( \Sigma \text{ size of locality} > \text{total memory size} \)
Thrashing

- Working-Set Model.
- Page-Fault Frequency Scheme.
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

Page reference table

\[ \ldots 2 \ 6 \ 1 \ 5 \ 7 \ 7 \ 7 \ 7 \ 5 \ 1 \ 6 \ 2 \ 3 \ 4 \ 1 \ 2 \ 3 \ 4 \ 4 \ 3 \ 4 \ 4 \ 4 \ 1 \ 3 \ 2 \ 3 \ 4 \ 4 \ 3 \ 4 \ 4 \ 4 \ 4 \ 4 \ 4 \ldots \]

\[ \Delta \]

\[ t_1 \]

\[ WS(t_1) = \{1, 2, 5, 6, 7\} \]

\[ \Delta \]

\[ t_2 \]

\[ WS(t_2) = \{3, 4\} \]
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
Conclusion

- Virtual memory is commonly implemented by demand paging.

- Demand paging is used to reduce the number of frames allocated to a process.

- We need both page-replacement and frame-allocation algorithms.
End of Chapter 9