Segmentation

- Memory-management scheme that supports user view of memory
- A program is a collection of segments.
- A segment is a logical unit such as:
  - main program, procedure,
  - function, method,
  - object, local variables, global variables,
  - common block, stack,
  - symbol table, arrays
Logical View of Segmentation

user space

physical memory space
Segmentation Hardware

- Logical address: <segment-number, offset>,</p>
- **Segment table** –
  - **base** – starting physical address
  - **limit** – length of the segment
Virtual Memory

- **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

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

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

page table

valid-invalid bit

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

- Memory access time = 200 nanoseconds

- Average page-fault service time = 8 milliseconds

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

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

- 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.

![Diagram of Copy-on-Write (COW)]
After Process 1 Modifies Page C
What happens if there is no free frame?

- Page replacement
  - find some page in memory, but not really in use, swap it out
  - performance – want an algorithm which will result in minimum number of page fault
- Use **modify (dirty) bit** to reduce overhead of page transfers
  – only modified pages are written to disk
Need For Page Replacement

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

Page table for user 1:
- Page 0: v
- Page 1: v
- Page 2: v
- Page 3: i

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

Page table for user 2:
- Page 0: v
- Page 1: i
- Page 2: v
- Page 3: v

Invalid bit:
- Frame 3: i

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

PC: 1

M

B
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

- Minimize page-fault rate

![Graph showing the number of page faults versus the number of frames. The graph depicts a decreasing trend as the number of frames increases.]
Page Replacement Algorithms

- Minimize page-fault rate
Page Replacement Algorithms

- Minimize page-fault rate
FIFO 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>7</th>
<th>7</th>
<th>2</th>
<th>2</th>
<th>2</th>
<th>4</th>
<th>4</th>
<th>4</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>7</th>
<th>7</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<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>
<td>3</td>
<td>3</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>1</td>
</tr>
</tbody>
</table>
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)

- 4 frames
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
  3 3 2 4
```

- 4 frames

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

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

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

6 page faults
### Optimal 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 2 2 2 7</td>
</tr>
<tr>
<td></td>
<td>0 0 0 0 4 0 0 0 0</td>
</tr>
<tr>
<td></td>
<td>1 1 3 3 3 3 1 1 1</td>
</tr>
</tbody>
</table>
Least Recently Used (LRU) Algorithm

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

- 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

reference string

| 7 | 0 | 1 | 2 | 0 | 3 | 0 | 4 | 2 | 3 | 0 | 3 | 2 | 1 | 2 | 0 | 1 | 7 | 0 | 1 |
| 7 | 7 | 7 | 2 | 2 | 4 | 4 | 4 | 0 | 1 | 1 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 1 | 1 | 3 | 3 | 3 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 7 |

page frames
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

Stack implementation – keep a stack of page numbers in a double link form:

- Page referenced: move to the top
- No search for replacement
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

(a) circular queue of pages

(b) circular queue of pages
See you next time