Chapter 9: Virtual Memory
Multiprogramming Memory Management

… so far

1. Dynamic Loading

- The main Program gets loaded into memory
- Routines are stored in Relocatable Load format on disk
- As main program (or subsequently loaded routine) executes it calls other routines
  - The calling program checks if called routine is loaded?
    - Reference the called routine
  - Else:
    - Calling program calls Relocatable Linker Loader to load Routine into memory
    - Relocatable Linker Loader updates address space of calling program

2. Dynamic Linking

- A stub is included into the image of each library routine reference
  - Locates appropriate memory-resident library routine(s)
  - Indicates how to load the library routine(s)
3. Swapping

A process is swapped temporarily out of memory to fast and large disk, and then brought back into memory for continued execution.

Round-Robin CPU Scheduling using time slice

4. Paging

5. Segmentation
Let’s Summarize

- **Goal:**
  - Simultaneously keep several processes in memory to facilitate multiprogramming
- **Require an entire process to be in memory to execute**
  - Dynamic Loading and Dynamic Linking are exceptions
    - But require extra work by programmer
- **Program size cannot exceed size of physical memory**
Multiprogramming Memory Management Technique

Virtual Memory

Background

- Allows execution of processes that are not completely in memory
  - Program not constrained by size of physical memory
  - Increased CPU utilization
  - Increased throughput
Virtual memory – separation of user’s view of logical memory from physical memory.

- Virtual memory is typically larger than physical memory
- Process Logical address space can therefore be much larger than physical address space.
- Process is modeled as a sequence of Pages
- At time $t_i$, Page may be in either
  - Physical memory
  - High speed disk
Virtual Memory
Address Space

- Virtual Address Space:
  - Logical view of how process is stored in memory
    - **Contiguous** page frames
      - Fixed starting logical address (0)
    - MMU maps contiguous logical frames to non contiguous physical frames
  - Utilizing the Hole
    - Heap – used for dynamic memory allocation
    - Stack - used for successive function calls
    - Dynamically link libraries during execution
    - Dynamically link shared objects during execution
- **Utilizing the Hole (Sparse address)**
  - Allows files and memory to be shared by two or more processes during execution
    - System libraries
      - Object(s) mapped to process virtual address space
    - Shared memory
      - Inter-process communication
      - Process P creates a set of pages for sharing with other Processes
        - The other processes consider shared pages as part of their virtual address space
Virtual Memory Systems

Demand Paging

- How are executable programs loaded from disk to memory?
  - At init, one or more virtual pages will be in Physical memory or fast high speed disk (secondary memory)
    - The pager (“swapper”) guesses which pages should be swapped in before the process is swapped out
  - Next, Virtual Pages are loaded into Memory as they are needed (Demand Paging)
    - eg., When a page fault occurs
      - Too many page faults may lead to performance bottlenecks
      - Need a technique to decrease swap time
Demand Paging

Basic Concept

- Need to distinguish between pages that are in memory and pages on the disk
- Use Valid-Invalid Bits:
  - Valid →
    - page is legal and in memory
  - invalid →
    - page is not in logical address space of the process or
    - page is valid, but currently on disk
- Access to a page designated as an invalid in the page table:
  - Page fault trap to OS
    - Trap → OS cannot move desired page into memory
Demand Paging
Handling Page Fault

1. Check internal table in PCB if reference is valid
2. If invalid (i), terminate process. If valid but page is not in physical memory (i) → bring page in mem:
   - Seek a free frame
3.  Schedule disk operation to move
   - The page into free frame (pure demand paging)
   - Alt. move page and its neighbors to free frames (locality of reference)
4.  When step 4 is complete, modify page table and internal table
5.  Restart interrupted instruction

[Diagram showing the process steps with labels: 1. load M, 2. trap, 3. page is on backing store, 4. bring in missing page, 5. reset page table, 6. restart instruction]
Demand Paging
Performance

- 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)

  \[
  \text{EAT} = (1 - p) \times \text{memory access} + p \times (\text{page fault overhead} + \text{swap page out} + \text{swap page in} + \text{restart overhead})
  \]
Performance of 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
  \[ EAT = (1 - p) \times 1 + p \left( \frac{15000}{1 + 15000p} \right) \text{ (in msec)} \]
Page Replacement

Basic Method

- The OS needs to allocate a frame to a process but there are no available free frames, what happens?
  - OS makes room in memory by copying a page from its frame to the disk (backing store)
    - Each Page of a frame is associated with a Modify bit (or dirty bit)
      - Bit set by hardware whenever word/byte is written to frame
  1. Select location of desired page on disk....
  2. Find a free frame
    - If one exists, use it
    - Otherwise use a Page Replacement algorithm to select a “victim”
      - Select a page for replacement & Examine its Modify bit
        - If modify bit is set → Page has been modified since it was copied into memory
          a. Write victim Page to disk
          b. Change the victim Page and Frame tables
        - Otherwise, Page has not been modified since it was brought into memory
          a. No need to write the page to disk
  3. Read the desired Page into newly freed frame; change the page and frame table
  4. Restart the user process

- How does OS select a page to copy to disk?
  - Use a page Replacement policy
    - Random
      - Take any page (victim)
    - FIFO
      - Select Page in Memory for the longest
    - LEAST Recently Used
      - Use the page that has not been referenced for the longest period
Page Replacement

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

frame
valid-invalid bit

0 i
f v

page table

physical memory
Demand Paging
Page-Replacement and Frame Allocation

- Page replacement & Frame Allocation compliment Demand Paging
  - Process with 20 pages can be executed with 10 or less frames. How?
    - Use Page Replacement Algorithm
- Assume multiple Processes reside in memory
  - How do you efficiently allocate frames to each process?
    - Use page Allocation Algorithm
Page-Replacement Algorithm

Attributes

- **Goal:**
  - Seek 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

- **Reference String:**
  - Assume memory address sequence with 100 bytes/page:
    - 0100, 0102, 0202, 0103, 0300, 0401, 0106, 0207,
    - 0505, 0106, 0204, 0301, 0404, 0508
  - Reference String:
    - 1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5
Page-Replacement Algorithm

First-In First-out (FIFO)

FIFO
- Record time when each page is brought into memory
- Replace Page with oldest time

Assume reference string:
1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5

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

9 page faults with 3 frames

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

10 page faults with 4 frames

3 frames → 3 pages can be in memory at a time per process
1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5

4 frames → 4 pages can be in memory at a time per process
1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5
FIFO Page-Replacement Algorithm

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

page frames
Optimal Page-Replacement Algorithm
OPT

- Basic idea:
  - Replace the page that **will not** be used for the longest period of time

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

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

  6 page faults
  with
  4 frames

- Challenge:
  - Implementation requires knowledge of the reference string a priori

- Benefit:
  - To compare relative efficiency of other (or new) algorithms
  - Has the lowest page-fault rate
Optimal Page-Replacement Algorithm

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

PAGE FRAMES
Least Recently Used Algorithm

**LRU**

- Basic idea:
  - 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  5
2
3  5  4
4  3
```

- Counter implementation
  - Associate each page entry with a counter
    - Set the counter-timer each time the page is referenced
  - When a page needs to be changed, look at the counters and select one with the longest period
LRU Page Replacement Algorithm

|    | 7 | 0 | 1 | 2 | 0 | 3 | 0 | 4 | 2 | 3 | 0 | 3 | 2 | 1 | 2 | 0 | 1 | 7 | 0 | 1 |
| reference string | 7 | 7 | 7 | 2 | 2 | 4 | 4 | 4 | 0 | 1 | 1 | 1 |
| page frames      | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 3 | 3 | 3 | 0 | 2 | 2 | 2 | 0 |

Operating Systems CS 33211
Allocation of Frames

- How do we allocate fixed amount of free memory amongst the various processes?
  - Suppose there are 100 frames and two processes, how many does each get?
    - Each process needs at least minimum number of pages. Why?
      - Minimum value dictated by computer architecture
- Two major allocation schemes
  - fixed allocation
  - priority allocation
Allocation Algorithms

- Equal allocation Algorithm
  - For example, if there are 100 frames and 5 processes, give each process 20 frames.
  - Do you see a disadvantage with this algorithm?

- Proportional allocation Algorithm
  - Allocate according to the size of process

Let
\[ m = \text{number of available frames} \]
\[ s_i = \text{size of process } P_i \]

Then number of frames \( a_i \) allocated to \( P_i \):

\[ a_i = \left( m \frac{s_i}{\sum s_i} \right) \]
Proportional Allocation Algorithm

Priority

- proportional allocation based on priority
- A high-priority process selects frames from lower-priority processes for replacement
  - 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

Disadvantage:

- Low priority process cannot control its page fault rate. Why?
Trashing

- If a process does not have “enough” frames to support pages, its page-fault rate will be very high. Why?
- 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
  - Performance degradation

Use Local replacement or priority replacement to limit the effect of trashing