Chapter 9: Virtual-Memory Management

Objectives

- To describe the benefits of a virtual memory system
- To explain the concepts of demand paging, page-replacement algorithms, and allocation of page frame
- To discuss the principle of the working-set model

9.1 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`

```
memory map
(page table)
physical memory
```
9.2 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 ⇒ reference to it
  - invalid reference ⇒ abort
  - not-in-memory ⇒ bring to memory
- **Lazy swapper** – never swaps a page into memory unless that page will be needed
  - A *swapper* manipulates entire processes
  - A *pager* deals with pages of a process

Lazy swapper – never swaps a page into memory unless that page will be needed.
Valid-Invalid Bit

- With each page table entry a valid–invalid bit is associated (v ⇒ in-memory, i ⇒ not-in-memory)
- Initially valid–invalid bit is set to i 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>v</td>
<td>v</td>
</tr>
<tr>
<td>v</td>
<td>v</td>
</tr>
<tr>
<td>v</td>
<td>v</td>
</tr>
<tr>
<td>i</td>
<td>i</td>
</tr>
<tr>
<td>....</td>
<td></td>
</tr>
<tr>
<td>i</td>
<td>i</td>
</tr>
</tbody>
</table>

During address translation, if valid–invalid bit in page table entry is i ⇒ page fault

Page Table When Some Pages Are Not in Main Memory

During address translation, if valid–invalid bit in page table entry is i ⇒ page fault

Page Fault

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

Steps in Handling a Page Fault:
1. Operating system looks at an internal table
2. If invalid reference ⇒ abort; If just not in memory ⇒ continue
3. Get an empty frame
4. Read the desired page into frame (swap in)
5. Reset the internal table (set validation bit = v ) and page table.
6. Restart the instruction that caused the page fault

Steps in Handling a Page Fault

- Trap
- Load M
- Restart instruction
- Page is on backing store
- Page table
- Free frame
- Physical memory
- Invalid page table
- Restarting
**Page Fault**

- **Pure demand paging**: never bring a page into memory until it is required.

- Multiple page faults per instruction possible?
  - Fortunately, programs tend to have locality of reference.

- A page fault in a three-address instruction (like \( z = x + y \)) will require fetching the instruction again, decoding it again, fetching the two operands again, and then adding again.

**Page Fault Problem**

- Restart instruction difficulty
  - block move of 256 bytes (p. 365)

- Solutions
  1. Attempts to access both block ends first
  2. Uses temporary registers to hold values of overwritten locations. If page fault, then all the old values are written back into memory before the trap occurs. This restores memory into its state before the 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)**
  
  \[
  EAT = (1 - p) \times memory\ access + p \times (\text{page fault overhead} + \text{swap page out} + \text{swap page in} + \text{restart overhead})
  \]

**Demand Paging Example**

- Memory access time = 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!!

  If we want performance degradation less than 10%,
  
  \[
  200 + p \times 7,999,800 < 220 \Rightarrow p < 0.0000025 \text{ (1 out of 399,990)}
  \]

  To implement demand paging
  
  - Need frame-allocation and page-replacement algorithm
**Demand Paging**

- Handling and overall use of swap space
  - Disk I/O to swap space is generally faster than that to the file system
  - Better paging throughput can be gained
    - by copying an entire file image into the swap space at process startup and then perform demand paging from the swap space.
    - Or to demand pages from the file system initially but to write the pages to swap space as they are replaced.

- Some systems limit the amount of swap space used through demand paging of binary files. These frames can simply be overwritten.

**9.3 Copy-on-Write**

- During process creation (check p. 113), **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 (all 0's)

**9.4 What happens if there is no free frame?**

- If a process of ten pages actually uses only 5 pages, then demand paging saves the I/O to load the 5 pages never used. The degree of multi-programming could be increased twice. We may over-allocating memory.

- Buffers for I/O also consume a lot of memory.

- **Page replacement** – find some page in memory, but not really in use, swap it out
  - want an page replacement algorithm which will result in minimum number of page faults

- Same page may be brought into memory several times
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

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
   - Write the victim frame to the disk; change the page and frame tables
3. Bring the desired page into the (newly) free frame; update the page and frame tables
4. Restart the process

Need For Page Replacement

Section 9.6 consider reducing the level of multiprogramming
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

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)

  \[
  \begin{array}{ccc}
  7 & 0 & 1 \\
  0 & 0 & 2 \\
  1 & 1 & 1 \\
  0 & 0 & 0 \\
  3 & 2 & 2 \\
  3 & 0 & 3 \\
  2 & 2 & 0 \\
  4 & 1 & 7 \\
  0 & 3 & 1 \\
  1 & 2 & 0 \\
  2 & 1 & 2 \\
  3 & 0 & 1 \\
  \end{array}
  \]

- 4 frames

  \[
  \begin{array}{ccc}
  7 & 0 & 1 \\
  0 & 0 & 2 \\
  1 & 1 & 1 \\
  0 & 0 & 0 \\
  3 & 2 & 2 \\
  3 & 0 & 3 \\
  2 & 2 & 0 \\
  4 & 1 & 7 \\
  0 & 3 & 1 \\
  1 & 2 & 0 \\
  2 & 1 & 2 \\
  3 & 0 & 1 \\
  \end{array}
  \]

  Exercise: Try the 4 frame case

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
  - 6 page faults

Exercise: Try the 3 frame case

Optimal Page Replacement

9.4.4 Least Recently Used (LRU) Algorithm
- Reference string: 1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5

Exercise: Try the 3 frame case

Counter implementation
- Add to CPU a logical clock (counter); every page entry has a time-of-use field. Every time page is referenced through this entry, copy the clock into the time-of-use field.
- When a page needs to be replaced, look at the time-of-use fields to determine which are to be replaced
**LRU Page Replacement**

Reference string:

```
7 7 7 2 2 4 4 4 0 1 1 1 1
0 0 0 0 0 3 3 3 3 0 0 0 0
1 1 3 2 2 2 2 2 2 2 2 7
```

Page frames:

- Stack implementation (to record the most recent page references) - keep a stack of page numbers in a double link form:
  - Page referenced:
    - move it to the top
    - Use doubly-linked list: requires 6 pointers to be changed
  - 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**

- Stack implementation (to record the most recent page references) - keep a stack of page numbers in a double link form:
  - Page referenced:
    - move it to the top
  - Use doubly-linked list: requires 6 pointers to be changed
  - No search for replacement
### Enhanced Second-Chance Algorithm

- Considering the reference bit and the modification bit, we have the following four classes:
  - (0,0): neither recently used nor modified
  - (0,1): not recently used but modified
  - (1,0): recently used but clean
  - (1,1): recently used and modified

### 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

### Page-Buffering Algorithms

- Keep a pool of free frames.
- When page faults, the desired page is read into a free frame from the pool before the victim is written to disk.
  - Allow the process to restart as soon as possible

### 9.38 Counting Algorithms

#### 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

### Page-Buffering Algorithms

- Keep a pool of free frames.
- When page faults, the desired page is read into a free frame from the pool before the victim is written to disk.
  - Allow the process to restart as soon as possible

### 9.40 Allocation of Frames

#### 9.5 Allocation of Frames

- The simple case: the single user system
  - Demand paging: Besides frames for the operating system, put all other frames in the free-frame list.

- Variations on the simple strategy
  - Require the OS allocate all its buffers and table space from the free-frame list. When this space is not in use by the OS, it can be used to support user paging.
  - Keep three free frames reserved on the free-frame list at all times.

#### 9.5.1 Allocation of Frames

- We cannot allocate more than the total available frames.
  - Each process needs *minimum* number of pages
    - Example: machine with all memory access: at least two memory accesses per instruction. If indirect addressing, then paging requires at least 3 frames per process
  - Example: IBM 370 – 6 pages to handle MVC (multiple move) instruction:
    - instruction is 6 bytes, might span 2 pages
    - *source block* might straddle 2 pages
    - *destination block* might straddle 2 pages
  - Worst case: multiple level indirection (must have a limit)
  - Two major allocation schemes
    - equal allocation
    - priority allocation
**Equal 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.

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

**Example**

\[
\begin{align*}
  m &= 62 \\
  s_1 &= 10 \\
  s_2 &= 127 \\
  a_1 &= \frac{10}{137} \times 62 \approx 4 \\
  a_2 &= \frac{127}{137} \times 62 \approx 57
\end{align*}
\]

**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
  - Allows a high-priority process to increase its frame allocation at the expense of a low-priority process
  - Low-priority process cannot control its own page-fault rate

- **Local replacement** – each process selects from only its own set of allocated frames

- Global replacement generally results in greater system throughput

**9.6 Thrashing** (痛擊)

- 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. It spends more time in paging than executing.
**Demand Paging and Thrashing**

- Why does demand paging work? **Answer: Locality model**
  - Process migrates from one locality to another
  - Localities may overlap
  - A process page faults when it changes locality
  - If we allocate fewer frames than size of the current locality, the process will thrash

- For a system, why does thrashing occur?
  
  \[ \text{sum of size of locality} > \text{total memory size} \]

- We can limit the effect of thrashing by using a **local replacement algorithm**

- If processes are thrashing, the average service time for a page fault will increase because of the longer queue for the paging device

**Working-Set Model**

- \( \Delta \equiv \text{working-set window} \equiv \text{a fixed number of page references} \)
  - Example: 10,000 instruction

**Page Reference Table**

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

- \( \text{WS}(t_1) = \{1, 2, 5, 6, 7\} \)
- \( \text{WS}(t_2) = \{3, 4\} \)
WSS \(_i\) (working set size 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 \(\Delta = \infty\) ⇒ will encompass entire program

\(D = \sum WSS_i\) = total demand frames of all processes

if \(D > m\) (total number of available frames) ⇒ Thrashing

Policy: if \(D > m\), then suspend one of the processes

---

**Page-Fault Frequency Scheme**

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

---

**Working Sets and Page Fault Rates**

- A peak in the page fault rate occurs when we begin demand-paging a new locality.
9.7 Memory-Mapped Files (skip)

- 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 Shared Memory in Windows

- Each I/O controller includes registers to hold commands and the data being transferred.
  - Special instructions allow data transfers between these registers and system memory.
- Memory-mapped I/O provides more convenient access to I/O devices.
  - Ranges of memory addresses are mapped to the device registers.
  - Appropriate for devices that have fast responses, like video controllers and screens.
  - Also convenient for serial and parallel ports to connect modems and printers to a computer.
    - By setting or clearing a bit in the device control register.
    - Polling (programmed I/O) or interrupt driven.

Memory-Mapped I/O
9.8 Allocating Kernel Memory

- Treated differently from user mode memory
  1. Kernel requests memory for structures of varying sizes
     - Some of them are less than a page in size
  2. Some kernel memory needs to be contiguous.
     - Ex: Hardware devices interact directly with physical memory

- Often allocated from a free-memory pool different from the list used to satisfy ordinary user-mode processes

---

Buddy System

- Allocates memory from fixed-size segment consisting of physically-contiguous pages
- Memory allocated using **power-of-2 allocator**
  - Satisfies requests in units sized as power of 2
  - Request rounded up to next highest power of 2
  - When smaller allocation needed than current available, current chunk split into two buddies of next-lower power of 2
    - Continue until appropriate sized chunk available

- Advantage: adjacent buddies can be combined quickly to form larger segment using **coalescing**
- Drawback: very likely to cause fragmentation within allocated segments

---

Buddy System Allocator

- **physically contiguous pages**
  - 256 KB
  - 128 KB
  - 64 KB
  - 32 KB

---

Alternate strategy: **Slab Allocation**

- **Slab** is one or more physically contiguous pages
- **Cache** consists of one or more slabs
- Single cache for each unique kernel data structure
  - Each cache filled with **objects** – instantiations of the data structure
- When cache created, filled with objects marked as **free**
- When structures stored, objects marked as **used**
- If slab is full of used objects, next object allocated from empty slab
  - If no empty slabs, new slab allocated
- Benefits include **no fragmentation, fast memory request satisfaction**
Prepaging
To reduce the large number of page faults that occurs at process startup
In a system with working-set model, we keep with each process a list of pages in its working set. Before suspending a process, its working set is saved.
Prepage all or some of the pages a process will need before they are referenced
But if prepped pages are unused, I/O and memory was wasted
Assume $s$ pages are prepaged and $\alpha$ of the pages is used
Is cost of the $s \cdot \alpha$ saved pages faults greater or less than the cost of prepaging $s \cdot (1 - \alpha)$ unnecessary pages?
$\alpha$ near zero $\Rightarrow$ prepaging loses
$\alpha$ near one $\Rightarrow$ prepaging wins

Page size selection must take the following into consideration:
- Size of the page table
  - A large page size is preferred
- Internal fragmentation
  - Need a small page size
- Time to read or write a page
  - Need a larger page size
- Locality
  - Smaller page size to match program locality more accurately
- Number of page faults
  - Need a larger page size to reduce number of page faults

TLB: expensive and power hungry
- TLB Reach - The amount of memory accessible from TLB
  - TLB Reach = (TLB Size) X (Page Size)
- Ideally, the working set of each process is stored in TLB
  - Otherwise there is a high degree of page faults
- Increase the Page Size to increase TLB reach
  - 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
  - Recent trends is to move toward software-managed TLB

An external page table must be kept for demand paging
Other Issues – Program Structure

- Program structure
  - ```
    int[128,128] data;
    for (j = 0; j < 128; j++)
      for (i = 0; i < 128; i++)
        data[i,j] = 0;
  ```
  - Each row is stored in one page
  - Program 1
    - ```
      for (j = 0; j < 128; j++)
        for (i = 0; i < 128; i++)
          data[i,j] = 0;
    ```
    - 128 x 128 = 16,384 page faults
  - Program 2
    - ```
      for (i = 0; i < 128; i++)
        for (j = 0; j < 128; j++)
          data[i,j] = 0;
    ```
    - 128 page faults

- Stack has good locality, hash table has bad locality

- In addition to locality, other factors:
  - search speed, total number of memory references, total number of pages touched

- Compiler and loader
  - Code pages are always read-only
  - Loader can avoid placing routines across page boundaries
  - Routines that call each other can be packed into one page

- Language
  - The use of pointers in C and C++ tend to randomize access to memory, thereby potentially diminishing a process’s locality
  - OO programs tend to have a poor locality of reference

Other Issues – I/O interlock (skip)

- 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

- Solutions
  - Never execute I/O to user memory. Use system memory instead. Extra copying between user memory and system memory.
  - Allow pages to be locked into memory with a lock bit.
  - Lock-bit can be used in preventing replacement of a newly brought-in page until it can be used at least once. Useful for low-priority process.

Reason Why Frames Used For I/O Must Be In Memory