Memory Management
Memory Management

• Memory is an important resource that needs to be managed by the OS
• Memory manager is the component of the OS responsible for managing memory
• Memory management systems fall into two classes

1. Memory managers that move processes back and forth between memory and disk during execution

2. Memory managers that require that all processes are in memory at all times

We will study both techniques. The former requires swapping and paging techniques. The latter has only been used in simple operating systems.
Monoprogramming without Swapping or Paging

- One process in memory at all times
- Process has entire memory available to it
- Only one process at a time can be running
- Not practical for multiprogramming
- Used on older OS such as MS-DOS

Memory Layout for Monoprogramming Systems
Multiprogramming

- Multiprogramming is required to support multiple users (processes)

- Used in just about all modern OS

- Requires multiple processes to be resident in memory at the same time

- Increases processor utilization
  - Especially for I/O bound processes

- Able to analyze and predict system performance

- Able to model the relationship between the quantity of memory and CPU utilization
Modeling Multiprogramming

CPU Utilization = 1 - \( p^n \)

- **Goal:** Maximize CPU utilization
- Assume a process spends a fraction \( p \) of its time in an I/O wait state
- With \( n \) processes, the probability that all \( n \) processes are waiting for I/O is \( p^n \)
- As the following graph shows, the smaller the I/O wait percentage, the larger number of processes that can be handled to maximize CPU utilization
Multiprogramming with Fixed Partitions

- **Desire**: Have more then one program in memory at one time

- **Solution**: Divide memory up into partitions
  - Partitions may be equally or variably sized

- Create an input queue to place processes in the smallest partition that is large enough to hold the process

- **Problems**:
  - Some memory partitions may have processes waiting while other partitions are unused
  - The space in the partition, which is larger then the process memory requirement, is not used - thus it is wasted

- May be used for batch systems, where memory requirements can be modeled

- Not good for interactive systems that often have dynamic memory requirements
Multiprogramming with Fixed Partitions

Partition 4
A → B → C

Partition 3
D → E

Partition 2
F

Partition 1
Operating System

Multiple Queues

One Queue
A → B → C → D → E → F

Operating System
Partition 4
Partition 3
Partition 2
Partition 1
 Operating System
Relocation and Protection

• Multiprogramming introduces two problems that must be solved - relocation and protection

• **Relocation**: When programs are compiled and linked, pointers reference memory locations within the process
  – These pointers must be treated as offsets within the process
  – Allows the program to be loaded into any memory partition and run correctly

• **Solution**: As program is being loaded into memory, add base address to all pointers
  – Does not allow the program to be relocated to another memory partition
  – Must be careful not to fixup addresses that point to locations outside of the processes
    • Interrupt vectors, system resources, ...
Relocation and Protection

- Relocation does not solve the protection problem
- Must prevent programs from referencing addresses outside of its partition
  - Recall one process should not be able to negatively impact another process
- Thus using absolute memory addresses are not safe and are often not used
- **Solution:** Use Base and Limit registers
  - Let compiler and linker generate absolute inter-process addresses
  - Base register: Load with the base address of the memory partition
  - Add value (or offset) in base register to every pointer in the program - do it at runtime.
  - Limit register: Load with size of partition. Used to enforce protection
Relocation and Protection

- Using base and limit registers has several advantages
- Requires hardware support for dynamic address calculation and protection enforcement
- Base and Limit register mechanism allows programs to be relocated
  - Just update the base and limit register values
- Relocation of programs is often necessary
  - Interactive programs require additional resources
  - Memory becomes fragmented and needs to be reorganized
    - Java Garbage Collection
Memory Manager Issues

• Often the number of processes that an operating system needs to manage have memory requirements that exceed the amount of available memory

• Use disk space as an extended or virtual memory

• Move process images to and from disk into main memory

• Disk access is much slower then memory access

• Need efficient memory management algorithms to efficiently manage all system processes

• Common techniques:
  – Swapping
  – Paging
Swapping

• When the number of processes has memory requirements that exceed the amount of main memory, some processes need to be kept on disk

• Moving processes from main memory to disk and back again is called **swapping**

• Often swapping systems use variable-sized partitions

• Using variable sized partitions reduces the wasted memory associated with “fitting” processes into fixed sized partitions that must be large enough to hold the process memory image

• With variable partitions the size and number of memory partitions dynamically varies

• Goal: Improve memory utilization
Variable Partitions

A

B

C

D
Variable Partitions

• Allocate “just large enough”

• Good idea to make the variable partitions a little larger then needed for “growing” memory requirements

• Memory can be compacted to consolidate holes
  – Move memory partitions down as far as possible
  – Compaction is inefficient because of excessive CPU requirements to reorganize memory partitions

• If memory requirements grow beyond a processes partition size, move the partition to a new partition
  – Requires relocation
Keeping Track of Partitions

- The operating system must keep track of allocated and free areas of memory

- Bit Maps

- Linked Lists

- Buddy System
Variable Partitions with Bit Maps

- Memory is divided into allocation units
  - Few words to several K per allocation unit

- Each allocation unit is tracked by a bit map
  - 1 indicates allocation unit is in use
  - 0 indicates allocation unit is free

- Size of allocation unit is inversely proportional to the size of the bit map

- Searching a bit map for free allocation units is slow
  - Linear search of the bitmap array

- Heavy fragmentation over time

- Must balance allocation unit/bitmap size
Variable Partitions with Bit Maps

Bit Map

A

B

C

Memory

Brian Mitchell (bmitchel@mcs.drexel.edu) - Operating Systems
Memory Management with Linked Lists

- Maintain a linked list of allocated and free memory segments
  - Each node in the linked list represents a processes memory image or a hole between two processes

- Advantages of maintaining the list by sorted addresses:
  - Easy to consolidate adjacent “small” holes into a single large hole

- Linked lists often take less storage to manage then bit maps
  - Maintain a record per process rather then a bit per allocation unit
Memory Management with Linked Lists

<table>
<thead>
<tr>
<th>Process or Hole (P/H)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Starting Segment Number</td>
</tr>
<tr>
<td>Segment Length</td>
</tr>
<tr>
<td>Pointer to Next Segment</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Process or Hole (P/H)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Starting Segment Number</td>
</tr>
<tr>
<td>Segment Length</td>
</tr>
<tr>
<td>Pointer to Next Segment</td>
</tr>
</tbody>
</table>
Memory Management with Linked Lists

A

B

C

P 0 4

H 4 2

P 6 6

H 12 1

P 13 3 X
Allocating Memory using Linked Lists

- Algorithms:
  - **First Fit**: Scan linked list, use first block large enough to hold the process
    - Break the hole into two pieces
  - **Next Fit**: Same as first fit, but do not start the search from the beginning of the list
    - Maintain the location of the last allocation
    - Wrap around when the last record in the list is reached
    - Spreads out memory allocation
  - **Best Fit**: Search the entire linked list and choose the smallest hole that is large enough to hold the process memory image
  - **Worst Fit**: Search the entire linked list and choose the largest hole to split
    - Produces large holes
  - Simulation has shown that first fit is the most efficient
Memory Management with Linked Lists

- **Optimization:** Maintain separate linked lists for the allocated and free memory blocks

- Prevents OS from having to search entries in the linked list that are allocated when trying to find a free block
Fragmentation

- **Internal Fragmentation:** When the allocated partition is larger than the needed memory requirements
  - Allocation with fixed partitions

- **External Fragmentation:** When there are holes between the allocated partitions but no wasted space within the partitions
  - Bit Maps
  - Linked Lists
  - Also known as checkerboarding

- Excessive fragmentation reduces memory utilization
Swapping

- In swapping systems the disk is used to back main memory when there is not enough memory to hold all of the running processes.
- Disk space is allocated to each process so that there is a place on disk to hold the process when it is removed from memory and put onto disk.
- Swapping systems require that the entire memory image is in main memory in order to run the process.
- Often a subset of a processes entire memory image is required to run a process – The working set.
- It is desirable to keep only the a processes working set in memory during execution – **Solution**: Virtual Memory and Paging.
Overlays

• In the early days when programs were too large to fit into their partitions, programmers created overlays
  – Overlays were physical divisions of the program that were established by the programmer
  – When the program started running overlay 0 would be loaded into memory
  – As the program progressed, the program dynamically loaded and unloaded its overlays

• Using overlays is complex, error prone and yields hard to debug programs

• **Desire:** Have the OS, not the programmer, manage the process of splitting up the program and dynamically loading its overlays
  – This technique is known as virtual memory
Virtual Memory

• Virtual memory
  – The size of the program data and stack may exceed the available amount of physical memory
  – Keep the parts of the program in use in main memory and the remaining parts on disk
  – Virtual memory manager dynamically moves process pieces between memory and disk

• Virtual memory systems must be efficient to avoid excessive disk to memory and memory to disk activities
Virtual Memory Architecture

Virtual Address Space

Virtual Memory Manager

Physical Memory

Disk Swap Area
Virtual Memory

• With virtual memory systems, programs run with a large virtual memory address space
  – 4 GB in Windows NT

• The virtual memory manager abstracts the location of the physical address from the running program

• A process in a virtual memory system generates and uses virtual addresses in a virtual address space
  – Typically the translation of a virtual address to a physical address is handled by the MMU hardware
Pages

- The virtual address space is divided into equally sized units called **pages**
  - Pages are typically 1 to 8 K (2-4K is common)
- The physical memory is also divided into page frames
- The page frames in physical memory (and on disk) are the same size as pages in the virtual address space
- Virtual memory system maps pages from the virtual address space into page frames in physical memory or on disk
- These systems allow the process working set to be in memory by only keeping the active page frames resident in physical memory
Page Tables

- Page tables are used to map virtual to physical addresses
Virtual Memory Mapping Example

MOV AX, 8194

- 8194 is in virtual page 2 which handles virtual addresses between 8 and 12 K
- Virtual page two is mapped to physical page 6
- Physical page 6 corresponds to physical addresses in the range 24-28K
- 8194 is two bytes into virtual page 2
- They physical address is also offset 2 bytes into the physical page
- The physical address is \((24576+2) = 24578\)
- Thus MOV AX, 8194 is in reality MOV AX, 24578
Page Tables

- Resident and managed by the MMU
- Loaded during process switches by the scheduler
- Each page must have a bit associated with it that indicates if the virtual page is mapped
- If a process accesses a page that is not mapped then a page fault trap is generated
- The operating system handles the page fault trap by finding an available physical memory page
  - A page must be free or it is evicted to disk via the swapper
- The desired page is then loaded into the physical page
- The page table entry is mapped to the newly loaded page
Virtual Memory - How it works!

- Virtual memory systems work by partitioning the virtual address into a number of fields
  - Index into the page table
  - Page offset

![Diagram of Virtual Memory Addressing]

**Virtual Table Index**: 0010 0000000000100

**Offset**

<table>
<thead>
<tr>
<th>0</th>
<th>110</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>001</td>
</tr>
<tr>
<td>2</td>
<td>110</td>
</tr>
</tbody>
</table>

**Physical Address**: 24580: 110 0000000000100
Page Table Issues

• Depending on the page size, the page table size can become very large
  – Ex: 32 bit virtual address space
  – With a 4K page, the 32 bit address space has 1 million pages
  – Each process needs its own page table with 1 million entries
  – It takes time to load an entire page table

• The mapping from virtual to physical address must be fast
  – Address translation must take place on every memory reference

• Can keep page in memory or special registers

• Issue: It is impractical to manage these large page tables as one entity
Multilevel Page Tables

• Want to avoid having huge page tables in memory
• Solution: Use a multilevel page table
  – Virtual address is divided into multiple page table components and an offset into the physical address page

• Only need to keep active page tables in memory
  – The rest can be kept out on disk

• Desirable because most processes only use a fraction of their total virtual address space
Multilevel Page Tables

PT1  PT2  Offset

to pages
Multilevel Page Tables

- MMU uses the top level page table and the first part of the virtual address to locate the second level page table.

- MMU uses the next level page table and the second part of the virtual address to locate the page frame.

- Can be done to any level of indirection:
  - Two level page table
  - Three level page table
  - Four level page tables
Other Information Kept in the Page Table

- The following fields are often found in a page table
  - **Page frame number**: The actual physical memory frame
  - **Present/Absent bit**: Used to control page faults. The OS manages the page faults by using its swapping subsystem
  - **Protection bit**: Describes the protection of the page frame (read/write, read-only)
  - **Modified bit**: Set when the page is written to. Helpful in optimizing the swapping algorithm
  - **Referenced bit**: Set when the page is referenced. Helpful in optimizing the swapping algorithm
  - **Caching enabled/disabled**: Enables or disables caching of the page.
Example Paging Architecture: SPARC

- The SPARC microprocessor is a high-performance RISC chip
- 32 bit addressing
- The SPARC uses a three-level paging architecture
- The SPARC also uses a context table
  - Context table used to improve performance
  - Avoid loading page tables during context switches
- Context table has 4096 entries
- Page tables accessed by dividing the 32 bit address
  - High 8 bits - first page table
  - Second 6 bits - second page table
  - Next 6 bits - third page table
  - Remaining 12 bits - 4096 memory page
The SPARC Architecture

SPARC 32-bit Address

Context Table

First Level

Second Level

Third Level

4K Page

256 Entries

64 Entries

64 Entries

Address Offset

8 bits 6 bits 6 bits 12 bits
Associative Memory

- If page tables are kept in memory then each address requires many memory accesses
  - Access memory for each address translation
    - Based on page table levels
  - Access memory for actual memory access
- **Observation:** Most programs make a large number of references to a small number of pages
  - Principle of locality
- **Solution:** Use hardware to map physical to virtual addresses without going through the page table
- Device known as **associated memory** or a **translation lookaside buffer** (TLB)
Associative Memory

- TLB typically has between 8 and 32 entries
- Used an address translation cache
- Typical caching issues apply
  - Hit/Miss issues, What to evict?

<table>
<thead>
<tr>
<th>Valid</th>
<th>Virtual Page</th>
<th>Modified</th>
<th>Protection</th>
<th>Page Frame</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>140</td>
<td>1</td>
<td>RW</td>
<td>31</td>
</tr>
<tr>
<td>1</td>
<td>20</td>
<td>0</td>
<td>R X</td>
<td>38</td>
</tr>
<tr>
<td>1</td>
<td>130</td>
<td>1</td>
<td>RW</td>
<td>29</td>
</tr>
<tr>
<td>1</td>
<td>129</td>
<td>1</td>
<td>RW</td>
<td>62</td>
</tr>
</tbody>
</table>
Using Associative Memory

• During each memory reference
• Virtual page number is searched for in the TLB
• Hit:
  – Does access violate protection bits?
    • Fault
  – Does access modify the memory page?
    • Change modified bit
  – Return the physical page frame
• Miss
  – Address translation performed via the standard page table
  – Evicts a member from TLB (if full) and puts the new page in the TLB
  – Evicted pages update page tables modified bits
  – TLB modified flag set by page table value
Page Replacement Algorithms

- When a page fault occurs the OS has to choose a page to remove from memory
- Space used for new page that caused the original fault
- If evicted page has been modified it must be written to the disk swap area
- If evicted page has not been modified it is evicted
- Choices
  - Pick page at random
    - Not good performance if evicted page is being heavily used
  - Use an algorithm to choose a lightly used page
    - Principle of locality
    - Use heuristics to choose page to evict
Optimal Algorithm

- Best algorithm, although impossible to implement
- Optimal Algorithm
  - When a page fault occurs look at all of the pages in memory
  - With each page in memory calculate the number of instructions that will be executed before the page is next referenced
  - Choose the page with the highest value
- The optimal algorithm is unrealizable because it is impossible to determine the number of instructions that will be executed in the future before the page will be referenced
  - Depends on the scheduling algorithm
  - Even if possible, this algorithm would be slow because calculating this measurement is expensive
Not-Recently-Used Replacement Algorithm

• Have operating system collect statistics about pages in memory

• Use the statistics to choose a page to evict

• **Heuristic:** Pages recently used are more likely to be used again soon

• Use 2 status bits with each page
  – $R$ bit set whenever the page is referenced
  – $M$ bit set whenever the page is modified

• Initialize bits to zero (0), when set to one (1) it remains 1 until the OS resets the bit
Not-Recently-Used Page Replacement Algorithm

• Each page in memory is classified according to the referenced and modified bits
  – **Class 0**: not referenced, not modified
  – **Class 1**: not referenced, modified
  – **Class 2**: referenced, not modified
  – **Class 3**: referenced, modified

• NRU Algorithm
  – *Divide pages into classes*
  – *Evict a page at random from the lowest number non-empty class*

• The OS periodically resets the *R* bit
  – Used to track page activity

• NRU is easy to understand and relatively fast

• Not optimal solution
First-In, First-Out (FIFO) Page Replacement Algorithm

- Simple algorithm based on a FIFO queue

- Algorithm
  - As each page is loaded into main memory, insert it on the end of the queue
  - Remove pages from the front of the queue
  - Items on queue the longest are removed

- **Problem**: Page eviction mechanism not based on usage. Oldest page might be heavily used

- FIFO algorithm often not used, however, it can be used as the basis of a better algorithm - The Second Chance Page Replacement Algorithm
Second Chance Page Replacement Algorithm

- Based on FIFO algorithm, but it utilizes the referenced (R) bit
- Algorithm
  - As each page is loaded into main memory, insert it on the end of the queue
  - If the page at the front of the queue has R=1, then clear the R bit and place the page on the end of the queue
  - Remove the first item found at the front of the queue with R=0
- The second chance algorithm gives pages that have been recently referenced a second chance by requeueing them
- If all pages have been referenced then the original page with the R bit cleared will reappear at the front of the queue
The Clock Page Replacement Algorithm

- The clock algorithm is an extension of the second chance algorithm
- Second chance algorithm is slow because pages are being moved within a list
- Clock algorithm arranges pages in a circular list and maintain a pointer to the oldest element in the list

Algorithm

\[ R = 0; \text{ evict page} \]
\[ R = 1; \text{ clear R and advance hand} \]
Least Recently Used (LRU) Algorithm

• LRU is a good approximation to the optimal algorithm
• Pages heavily used in the last set of instructions will probably be used in the next set of instructions
• LRU is a very expensive algorithm
• Algorithm
  – *Maintain a list of pages in memory*
  – *Most recently used page at the front of the list*
  – *Least recently used page at the rear of the list*
  – *List must be updated during every page reference*
• Other techniques:
  – Use counter to track LRU
LRU Algorithm with Matrices

- If there are N pages in memory, create a NxN matrix
- Algorithm
  - If page k is referenced:
  - Set all bits in row k to 1
  - Set all bits in column k to 0
  - Row with the lowest value is LRU

Pages Referenced: 0123

<table>
<thead>
<tr>
<th></th>
<th>0123</th>
<th></th>
<th>0123</th>
<th></th>
<th>0123</th>
<th></th>
<th>0123</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0111</td>
<td>0</td>
<td>0011</td>
<td>0</td>
<td>0001</td>
<td>0</td>
<td>0000</td>
</tr>
<tr>
<td>1</td>
<td>0000</td>
<td>1</td>
<td>1011</td>
<td>1</td>
<td>1001</td>
<td>1</td>
<td>1000</td>
</tr>
<tr>
<td>2</td>
<td>0000</td>
<td>2</td>
<td>0000</td>
<td>2</td>
<td>1101</td>
<td>2</td>
<td>1100</td>
</tr>
<tr>
<td>3</td>
<td>0000</td>
<td>3</td>
<td>0000</td>
<td>3</td>
<td>0000</td>
<td>3</td>
<td>1110</td>
</tr>
</tbody>
</table>
Not Frequently Used (NFU) Algorithm

• Simulation of LRU
• Requires a software counter associated with each page
  – Initially zero
• Algorithm
  – During each clock interrupt, add the value of the R bit to the counter
  – Clear the R bit
  – Page with lowest counter is least frequently used
• While good, the NFU algorithm may keep pages around for a long time that have high counts even though they have not been used recently
  – Page used a lot a long time ago
• A small modification can be made to make NFU simulate LRU quite well
Aging Algorithm

- Aging algorithm based on NFU to simulate LRU
- Algorithm
  - During each clock tick
    * Shift counter one bit to the right
    * Add R bit to leftmost bit in the counter
  - Remove a page with the lowest counter value

<table>
<thead>
<tr>
<th>R BITS FOR PAGES 0–5</th>
<th>101011</th>
<th>110010</th>
<th>110101</th>
</tr>
</thead>
<tbody>
<tr>
<td>Page 0</td>
<td>10000000</td>
<td>11000000</td>
<td>11100000</td>
</tr>
<tr>
<td>Page 1</td>
<td>00000000</td>
<td>10000000</td>
<td>11000000</td>
</tr>
<tr>
<td>Page 2</td>
<td>10000000</td>
<td>01000000</td>
<td>00100000</td>
</tr>
<tr>
<td>Page 3</td>
<td>00000000</td>
<td>00000000</td>
<td>10000000</td>
</tr>
<tr>
<td>Page 4</td>
<td>10000000</td>
<td>11000000</td>
<td>01100000</td>
</tr>
<tr>
<td>Page 5</td>
<td>10000000</td>
<td>01000000</td>
<td>10100000</td>
</tr>
</tbody>
</table>
Modeling Paging Algorithms

- Intuitively, the more pages in memory the less page faults
- Belady’s anomaly is a counter example

Pages Referenced: 012301401234

Youngest 012301444233
          01230111422
Oldest   0123000144
PPPPPPPP PP

3 Pages
9 Page Faults

Youngest 012333401234
          01222340123
          0111234012
Oldest   000123401
PPPP PPPPPP

4 Pages
10 Page Faults
Modeling Paging Algorithms

• Observation: Every process generates a sequence of memory references as it runs

• Reference string: Sequence of pages that a process references

• Categorizing a paging system
  – The reference string of an executing process
  – The page replacement algorithm
  – The number of page frame available in memory

• Enables a paging system to be thought of as a virtual machine
  – Enables paging algorithms to be simulated
Modeling Paging Algorithms: Example

LRU Algorithm Analysis

Reference String: 021354637473355311172341
021354637473355311172341
02135463747733533317234
0213546334477755531723
0213546666644477753172
0211555556664445517
0221111111166664455
00222222222226666
00000000000000000

Page Fault: PPPPPPPP P P P P PP
Distance String: iiiiiii4i423151261147465

\[ i = \infty \]
Distance String

- Distance string depends on the reference string and paging algorithm
- Able to perform statistical analysis on distance string to predict performance
- Predicting Page Faults
  - C vector; \( C[i] \) is the number of times \( i \) occurs in the distance string
  - F vector:
    \[
    F[i] = \sum_{k=m+1}^{n} C_k + C_\infty
    \]
- F vector can be used to predict page faults
- The value of \( F[i] \) is the number of page faults if the number of pages in memory is \( i \)
Predicting Page Faults: Example

<table>
<thead>
<tr>
<th>Reference String:</th>
<th>021354637473355311172341</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>021354637473355311172341</td>
</tr>
<tr>
<td></td>
<td>02135463747733533317234</td>
</tr>
<tr>
<td></td>
<td>0213546334477755531723</td>
</tr>
<tr>
<td></td>
<td>0213546666644477753172</td>
</tr>
<tr>
<td></td>
<td>02115555566644445517</td>
</tr>
<tr>
<td></td>
<td>02211111111166664455</td>
</tr>
<tr>
<td></td>
<td>00222222222226666</td>
</tr>
<tr>
<td></td>
<td>00000000000000000</td>
</tr>
</tbody>
</table>

| Distance String:  | ii ii ii ii ii ii ii ii i4 i4 23 15 12 61 14 7 4 6 5  |

\[
i = \infty\]

<p>| | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>C1</td>
<td>=</td>
<td>4</td>
<td>F1</td>
<td>=</td>
<td>20</td>
</tr>
<tr>
<td>C2</td>
<td>=</td>
<td>2</td>
<td>F2</td>
<td>=</td>
<td>18</td>
</tr>
<tr>
<td>C3</td>
<td>=</td>
<td>1</td>
<td>F3</td>
<td>=</td>
<td>17</td>
</tr>
<tr>
<td>C4</td>
<td>=</td>
<td>4</td>
<td>F4</td>
<td>=</td>
<td>13</td>
</tr>
<tr>
<td>C5</td>
<td>=</td>
<td>2</td>
<td>F5</td>
<td>=</td>
<td>11</td>
</tr>
<tr>
<td>C6</td>
<td>=</td>
<td>2</td>
<td>F6</td>
<td>=</td>
<td>9</td>
</tr>
<tr>
<td>C7</td>
<td>=</td>
<td>1</td>
<td>F7</td>
<td>=</td>
<td>8</td>
</tr>
<tr>
<td>C\infty</td>
<td>=</td>
<td>8</td>
<td>F8</td>
<td>=</td>
<td>8</td>
</tr>
</tbody>
</table>

With 4 pages there will be 13 page faults
Design Issues for Paging Systems

• In addition to executing the memory management algorithms, the OS must choose policies to make the memory manager work well

• **Demand Paging**: Pages are loaded into memory as page faults occur
  – Caused many page faults during process startup

• Most processes exhibit the **locality of reference** property
  – During any phase of execution a process references a relatively small percentage of its pages

• The set of pages currently being used by a process is called the processes **working set**
Working Set

• If the entire working set of a process is in memory no page faults will occur
• If a small percentage of a processes working set is in memory, many page faults will occur
• **Goal:** Have operating system keep track of a processes working set
• Make sure that the processes working set is in memory before a processes is scheduled to run
• **Prepaging:** The process of preloading a processes working set
• By carefully managing the processes working set, the performance of the memory manager will be greatly improved
• Working set can be modeled using the aging algorithm
Global versus Local Page Replacement

• When evicting a page, should the OS choose between all available pages, or a subset of main memory used for the current process

• Global
  – Give the process more options
  – Apply replacement algorithm to all pages
  – Some processes may dominate memory

• Local
  – Prevents a process from dominating main memory
  – Page replacement algorithm applies to pages owned by the current process
  – May lead to page faults with available memory

• OS must adopt a policy to ensure fairness
Page Size

• Selecting the page size is another tradeoff that the OS must manage

• Common page sizes range from 512 bytes to 8K

• The larger the page, the more internal fragmentation

• The smaller the page, the more the OS must manage
Memory Manager
Implementation Issues

- In addition to the memory management algorithm, and the implementation policies (page size, local vs. global allocation) there are a number of implementation issues that the memory manager must address

1. Instruction Backup
2. Locking Memory Pages
3. Shared Pages
4. Backing Store
5. Paging Daemons
Instruction Backup

• During a page fault, the instruction causing the fault is stopped during execution and a trap to the OS occurs

• After the OS handles the page fault the original instruction must be reexecuted

• This could be difficult
  – Especially in CISC architectures where instructions can consist of many words

• Solutions:
  – Hardware register that contains the PC before each instruction
  – Processor dumps state information on the stack to enable the instruction restart

MOVE AX,6(BX)

<table>
<thead>
<tr>
<th>1000</th>
<th>1002</th>
<th>1004</th>
</tr>
</thead>
<tbody>
<tr>
<td>MOVE</td>
<td>AX</td>
<td>BX+6</td>
</tr>
</tbody>
</table>

Restart Point

FAULT
Locking Pages In Memory

- Memory managers must have the ability to lock pages

- Useful for I/O bound programs where an I/O device was given a page to write data

- If the page is removed, the I/O device must either be given a new page or the device will write to another processes page

- May also use separate kernel and user buffers
  - All device I/O goes to kernel buffers and transferred to user buffers later

  - May cause a page fault during the transfer
Shared Pages

• It is often desirable for the memory manager to share pages

• Map the common physical page into the virtual address space of two or more processes

• Increases efficiency by preventing the same page from being in memory more than one time
  – Code pages
  – OS interface pages
  – Enables memory consistency
  – Solves protection-related issues

• Must use a reference count mechanism to prevent shared pages from being swapped out frequently
Backimg Store

• In virtual memory systems we used disk-based storage to “back up” main memory

• How do we create and manage the disk-based swap area
  – Preallocate a swap area for each process in advance
    • Keep pointer to swap area in the process table
    • Must manage growing processes
  – Preallocate nothing
    • Allocate pages on disk as needed
    • Must keep a disk address for each page table entry (not one per process)

• What algorithm do we use to manage the swap space
  – Variable/fixed partitions
Paging Daemons

- Memory managers work efficiently (and quickly) when there are few page faults
- **Goal**: Leverage idle processor time to evict probable old pages
- Enables plenty of free pages when processes are running (reduces page faults)
- The **paging daemon** is a process that is assigned a very low priority
  - Sleeps most of the time
  - When awakened the paging daemon inspects all memory pages
  - If the number of free pages is less then a system defined threshold, the paging daemon locates pages to evict
  - The eviction continues until the number of free page exceeds the system defined threshold
Segmentation

- Most processors support a flat memory model
- Virtual memory is one dimensional
  - One virtual address space per process
- Segmentation is when the hardware supports many completely independent address spaces
  - Each address space is called a segment
- Segments grow and shrink independent of each other
- Elegant solution for dynamic systems that have growing and shrinking memory requirements
Comparing Linear and Segmented Memory

<table>
<thead>
<tr>
<th>ISSUE</th>
<th>Linear</th>
<th>Segmented</th>
</tr>
</thead>
<tbody>
<tr>
<td>Programmer aware of memory organization?</td>
<td>NO</td>
<td>YES SEG:OFFSET</td>
</tr>
<tr>
<td>Number of linear address spaces</td>
<td>1</td>
<td>MANY</td>
</tr>
<tr>
<td>Can virtual address space exceed physical memory?</td>
<td>YES</td>
<td>YES</td>
</tr>
<tr>
<td>Can processes and data be distinguished separately?</td>
<td>NO</td>
<td>YES</td>
</tr>
<tr>
<td>Can dynamically growing data structures be easily accommodated?</td>
<td>NO</td>
<td>YES</td>
</tr>
<tr>
<td>Ease of sharing code between processes</td>
<td>Not as Easy</td>
<td>EASY</td>
</tr>
<tr>
<td>Purpose of technique</td>
<td>Large linear address space without a large memory</td>
<td>Easy sharing. Allow many logical address spaces</td>
</tr>
</tbody>
</table>
Example: Intel x86 Processors

- Intel processors were originally designed with support for segmented addresses only.
- Programmer managed addresses with segment:offset convention.
- Programmer must be cognizant if the address was in the same or different segment.
- Allowed 32 bit addressing in a 16 bit architecture.
- With 386 and beyond, most 32 bit operating systems only supported a single linear address space.
- Intel processors include a technique called **thunking** to manage translation between linear and segmented addresses.
  - Mainly used for backward compatibility.
  - Technique assumes a 4K page.