Chapter 8
Virtual Memory

Seventh Edition
William Stallings
You’re gonna need a bigger boat.

— Steven Spielberg, 

*JAWS*, 1975
Two characteristics fundamental to memory management:

1) all memory references are logical addresses that are dynamically translated into physical addresses at run time.

2) a process may be broken up into a number of pieces that don’t need to be contiguously located in main memory during execution.

If these two characteristics are present, it is not necessary that all of the pages or segments of a process be in main memory during execution.
## Terminology

<table>
<thead>
<tr>
<th>Term</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>Virtual memory</td>
<td>A storage allocation scheme in which secondary memory can be addressed as though it were part of main memory. The addresses a program may use to reference memory are distinguished from the addresses the memory system uses to identify physical storage sites, and program-generated addresses are translated automatically to the corresponding machine addresses. The size of virtual storage is limited by the addressing scheme of the computer system and by the amount of secondary memory available and not by the actual number of main storage locations.</td>
</tr>
<tr>
<td>Virtual address</td>
<td>The address assigned to a location in virtual memory to allow that location to be accessed as though it were part of main memory.</td>
</tr>
<tr>
<td>Virtual address space</td>
<td>The virtual storage assigned to a process.</td>
</tr>
<tr>
<td>Address space</td>
<td>The range of memory addresses available to a process.</td>
</tr>
<tr>
<td>Real address</td>
<td>The address of a storage location in main memory.</td>
</tr>
</tbody>
</table>
Execution of a Process

- Operating system brings into main memory a few pieces of the program
- Resident set - portion of process that is in main memory
- An interrupt is generated when an address is needed that is not in main memory
- Operating system places the process in a blocking state

Continued . . .
Execution of a Process

- Piece of process that contains the logical address is brought into main memory
- Operating system issues a disk I/O Read request
- Another process is dispatched to run while the disk I/O takes place
- An interrupt is issued when disk I/O is complete, which causes the operating system to place the affected process in the Ready state
Implications

- More processes may be maintained in main memory
  - only load in some of the pieces of each process
  - with so many processes in main memory, it is very likely a process will be in the Ready state at any particular time

- A process may be larger than all of main memory
Real and Virtual Memory

Real memory

- main memory, the actual RAM

Virtual memory

- memory on disk
- allows for effective multiprogramming and relieves the user of tight constraints of main memory
**Table 8.2**

**Characteristics of Paging and Segmentation**

<table>
<thead>
<tr>
<th>Simple Paging</th>
<th>Virtual Memory Paging</th>
<th>Simple Segmentation</th>
<th>Virtual Memory Segmentation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Main memory partitioned into small fixed-size chunks called frames</td>
<td>Main memory partitioned into small fixed-size chunks called frames</td>
<td>Main memory not partitioned</td>
<td>Main memory not partitioned</td>
</tr>
<tr>
<td>Program broken into pages by the compiler or memory management system</td>
<td>Program broken into pages by the compiler or memory management system</td>
<td>Program segments specified by the programmer to the compiler (i.e., the decision is made by the programmer)</td>
<td>Program segments specified by the programmer to the compiler (i.e., the decision is made by the programmer)</td>
</tr>
<tr>
<td>Internal fragmentation within frames</td>
<td>Internal fragmentation within frames</td>
<td>No internal fragmentation</td>
<td>No internal fragmentation</td>
</tr>
<tr>
<td>No external fragmentation</td>
<td>No external fragmentation</td>
<td>External fragmentation</td>
<td>External fragmentation</td>
</tr>
<tr>
<td>Operating system must maintain a page table for each process showing which frame each page occupies</td>
<td>Operating system must maintain a page table for each process showing which frame each page occupies</td>
<td>Operating system must maintain a segment table for each process showing the load address and length of each segment</td>
<td>Operating system must maintain a segment table for each process showing the load address and length of each segment</td>
</tr>
<tr>
<td>Operating system must maintain a free frame list</td>
<td>Operating system must maintain a free frame list</td>
<td>Operating system must maintain a list of free holes in main memory</td>
<td>Operating system must maintain a list of free holes in main memory</td>
</tr>
<tr>
<td>Processor uses page number, offset to calculate absolute address</td>
<td>Processor uses page number, offset to calculate absolute address</td>
<td>Processor uses segment number, offset to calculate absolute address</td>
<td>Processor uses segment number, offset to calculate absolute address</td>
</tr>
<tr>
<td>All the pages of a process must be in main memory for process to run, unless overlays are used</td>
<td>Not all pages of a process need be in main memory frames for the process to run. Pages may be read in as needed</td>
<td>All the segments of a process must be in main memory for process to run, unless overlays are used</td>
<td>Not all segments of a process need be in main memory for the process to run. Segments may be read in as needed</td>
</tr>
<tr>
<td>Reading a page into main memory may require writing a page out to disk</td>
<td></td>
<td>Reading a segment into main memory may require writing one or more segments out to disk</td>
<td></td>
</tr>
</tbody>
</table>
To avoid this, the operating system tries to guess, based on recent history, which pieces are least likely to be used in the near future.
Principle of Locality

- Program and data references within a process tend to cluster.
- Only a few pieces of a process will be needed over a short period of time.
- Therefore it is possible to make intelligent guesses about which pieces will be needed in the future.
- Avoids thrashing.
Paging Behavior

- During the lifetime of the process, references are confined to a subset of pages
For virtual memory to be practical and effective:

- hardware must support paging and segmentation
- operating system must include software for managing the movement of pages and/or segments between secondary memory and main memory
The term *virtual memory* is usually associated with systems that employ paging.

Use of paging to achieve virtual memory was first reported for the Atlas computer.

Each process has its own page table:
- Each page table entry contains the frame number of the corresponding page in main memory.
### Memory Management Formats

#### (a) Paging only

<table>
<thead>
<tr>
<th>Virtual Address</th>
<th>Page Table Entry</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Page Number</td>
</tr>
</tbody>
</table>

#### (b) Segmentation only

<table>
<thead>
<tr>
<th>Virtual Address</th>
<th>Segment Table Entry</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Segment Number</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Segment Table Entry</th>
<th>Page Table Entry</th>
</tr>
</thead>
<tbody>
<tr>
<td>Control Bits</td>
<td>Length</td>
</tr>
</tbody>
</table>

#### (c) Combined segmentation and paging
Address Translation

Figure 8.3  Address Translation in a Paging System
Two-Level Hierarchical Page Table

Figure 8.4 A Two-Level Hierarchical Page Table
Address Translation

Figure 8.5 Address Translation in a Two-Level Paging System
Page number portion of a virtual address is mapped into a hash value
- hash value points to inverted page table

Fixed proportion of real memory is required for the tables regardless of the number of processes or virtual pages supported
- Structure is called *inverted* because it indexes page table entries by frame number rather than by virtual page number
Inverted Page Table

Figure 8.6 Inverted Page Table Structure
Inverted Page Table

Each entry in the page table includes:

- **Page number**
  - the process that owns this page

- **Process identifier**
  - includes flags and protection and locking information

- **Control bits**
  - the index value of the next entry in the chain

- **Chain pointer**
Translation Lookaside Buffer (TLB)

- Each virtual memory reference can cause two physical memory accesses:
  - one to fetch the page table entry
  - one to fetch the data

- To overcome the effect of doubling the memory access time, most virtual memory schemes make use of a special high-speed cache called a translation lookaside buffer
Use of a TLB

Figure 8.7 Use of a Translation Lookaside Buffer
TLB Operation

Figure 8.8  Operation of Paging and Translation Lookaside Buffer (TLB) [FURH87]
The TLB only contains some of the page table entries so we cannot simply index into the TLB based on page number.

Each TLB entry must include the page number as well as the complete page table entry.

The processor is equipped with hardware that allows it to interrogate simultaneously a number of TLB entries to determine if there is a match on page number.
Direct Versus Associative Lookup

(a) Direct mapping

(b) Associative mapping

Figure 8.9  Direct Versus Associative Lookup for Page Table Entries
TLB and Cache Operation

Figure 8.10 Translation Lookaside Buffer and Cache Operation
Page Size

- The smaller the page size, the lesser the amount of internal fragmentation
  - however, more pages are required per process
  - more pages per process means larger page tables
  - for large programs in a heavily multiprogrammed environment some portion of the page tables of active processes must be in virtual memory instead of main memory
  - the physical characteristics of most secondary-memory devices favor a larger page size for more efficient block transfer of data
Paging Behavior of a Program

(a) Page Size

(b) Number of Page Frames Allocated

$P = \text{size of entire process}$

$W = \text{working set size}$

$N = \text{total number of pages in process}$

Figure 8.11 Typical Paging Behavior of a Program
### Example: Page Sizes

<table>
<thead>
<tr>
<th>Computer</th>
<th>Page Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>Atlas</td>
<td>512 48-bit words</td>
</tr>
<tr>
<td>Honeywell-Multics</td>
<td>1024 36-bit words</td>
</tr>
<tr>
<td>IBM 370/XA and 370/ESA</td>
<td>4 Kbytes</td>
</tr>
<tr>
<td>VAX family</td>
<td>512 bytes</td>
</tr>
<tr>
<td>IBM AS/400</td>
<td>512 bytes</td>
</tr>
<tr>
<td>DEC Alpha</td>
<td>8 Kbytes</td>
</tr>
<tr>
<td>MIPS</td>
<td>4 Kbytes to 16 Mbytes</td>
</tr>
<tr>
<td>UltraSPARC</td>
<td>8 Kbytes to 4 Mbytes</td>
</tr>
<tr>
<td>Pentium</td>
<td>4 Kbytes or 4 Mbytes</td>
</tr>
<tr>
<td>IBM POWER</td>
<td>4 Kbytes</td>
</tr>
<tr>
<td>Itanium</td>
<td>4 Kbytes to 256 Mbytes</td>
</tr>
</tbody>
</table>
The design issue of page size is related to the size of physical main memory and program size. Main memory is getting larger and address space used by applications is also growing. Contemporary programming techniques used in large programs tend to decrease the locality of references within a process. Most obvious on personal computers where applications are becoming increasingly complex.
Segmentation

Advantages:

- simplifies handling of growing data structures
- allows programs to be altered and recompiled independently
- lends itself to sharing data among processes
- lends itself to protection

Segmentation allows the programmer to view memory as consisting of multiple address spaces or segments
Segment Organization

- Each segment table entry contains the starting address of the corresponding segment in main memory and the length of the segment.
- A bit is needed to determine if the segment is already in main memory.
- Another bit is needed to determine if the segment has been modified since it was loaded in main memory.
Address Translation

Figure 8.12 Address Translation in a Segmentation System
In a combined paging/segmentation system a user’s address space is broken up into a number of segments. Each segment is broken up into a number of fixed-sized pages which are equal in length to a main memory frame.

Segmentation is visible to the programmer.

Paging is transparent to the programmer.
Address Translation

Figure 8.13 Address Translation in a Segmentation/Paging System
### Combined Segmentation and Paging

**Virtual Address**

<table>
<thead>
<tr>
<th>Segment Number</th>
<th>Page Number</th>
<th>Offset</th>
</tr>
</thead>
</table>

**Segment Table Entry**

<table>
<thead>
<tr>
<th>Control Bits</th>
<th>Length</th>
<th>Segment Base</th>
</tr>
</thead>
</table>

**Page Table Entry**

| P | M | Other Control Bits | Frame Number |

(c) Combined segmentation and paging

P = present bit  
M = Modified bit
Protection and Sharing

- Segmentation lends itself to the implementation of protection and sharing policies.
- Each entry has a base address and length so inadvertent memory access can be controlled.
- Sharing can be achieved by segments referencing multiple processes.
Protection Relationships

Figure 8.14 Protection Relationships Between Segments
The design of the memory management portion of an operating system depends on three fundamental areas of choice:

- whether or not to use virtual memory techniques
- the use of paging or segmentation or both
- the algorithms employed for various aspects of memory management
Policies for Virtual Memory

- Key issue: Performance
  - minimize page faults

<table>
<thead>
<tr>
<th>Fetch Policy</th>
<th>Resident Set Management</th>
</tr>
</thead>
<tbody>
<tr>
<td>Demand paging</td>
<td>Resident set size</td>
</tr>
<tr>
<td>Prepaging</td>
<td>Fixed</td>
</tr>
<tr>
<td></td>
<td>Variable</td>
</tr>
<tr>
<td>Placement Policy</td>
<td>Replacement Scope</td>
</tr>
<tr>
<td>Replacement Policy</td>
<td>Global</td>
</tr>
<tr>
<td>Basic Algorithms</td>
<td>Local</td>
</tr>
<tr>
<td>Optimal</td>
<td></td>
</tr>
<tr>
<td>Least recently used (LRU)</td>
<td></td>
</tr>
<tr>
<td>First-in-first-out (FIFO)</td>
<td></td>
</tr>
<tr>
<td>Clock</td>
<td>Cleaning Policy</td>
</tr>
<tr>
<td>Page Buffering</td>
<td>Demand</td>
</tr>
<tr>
<td></td>
<td>Precleaning</td>
</tr>
<tr>
<td>Load Control</td>
<td>Degree of multiprogramming</td>
</tr>
</tbody>
</table>
Fetch Policy

Determines when a page should be brought into memory

Two main types:

- Demand Paging
- Prepaging
Demand Paging

- **Demand Paging**
  - only brings pages into main memory when a reference is made to a location on the page
  - many page faults when process is first started
  - principle of locality suggests that as more and more pages are brought in, most future references will be to pages that have recently been brought in, and page faults should drop to a very low level
Prepaging

- Prepaging
  - pages other than the one demanded by a page fault are brought in
  - exploits the characteristics of most secondary memory devices
  - if pages of a process are stored contiguously in secondary memory it is more efficient to bring in a number of pages at one time
  - ineffective if extra pages are not referenced
  - should not be confused with “swapping”
Placement Policy

- Determines where in real memory a process piece is to reside
- Important design issue in a segmentation system
- Paging or combined paging with segmentation placing is irrelevant because hardware performs functions with equal efficiency
- For NUMA systems an automatic placement strategy is desirable
Replacement Policy

- Deals with the selection of a page in main memory to be replaced when a new page must be brought in.
  - Objective is that the page that is removed be the page least likely to be referenced in the near future.
- The more elaborate the replacement policy the greater the hardware and software overhead to implement it.
When a frame is locked the page currently stored in that frame may not be replaced

- kernel of the OS as well as key control structures are held in locked frames
- I/O buffers and time-critical areas may be locked into main memory frames
- locking is achieved by associating a lock bit with each frame
### Basic Algorithms

Algorithms used for the selection of a page to replace:

- Optimal
- Least recently used (LRU)
- First-in-first-out (FIFO)
- Clock
Optimal Policy

- Selects the page for which the time to the next reference is the longest
- Produces three page faults after the frame allocation has been filled
Least Recently Used (LRU)

- Replaces the page that has not been referenced for the longest time
- By the principle of locality, this should be the page least likely to be referenced in the near future
- Difficult to implement
  - one approach is to tag each page with the time of last reference
    - this requires a great deal of overhead
LRU Example

Figure 8.15  Behavior of Four Page Replacement Algorithms
First-in-First-out (FIFO)

- Treats page frames allocated to a process as a circular buffer
- Pages are removed in round-robin style
  - simple replacement policy to implement
- Page that has been in memory the longest is replaced
FIFO Example

Page address stream: 2 3 2 1 5 2 4 5 3 2 5 2

FIFO:

F = page fault occurring after the frame allocation is initially filled

Figure 8.15 Behavior of Four Page Replacement Algorithms
Clock Policy

- Requires the association of an additional bit with each frame
  - referred to as the use bit

- When a page is first loaded in memory or referenced, the use bit is set to 1

- The set of frames is considered to be a circular buffer

- Any frame with a use bit of 1 is passed over by the algorithm

- Page frames visualized as laid out in a circle
Clock Policy Example

**Figure 8.15** Behavior of Four Page Replacement Algorithms
Clock Policy

Figure 8.16  Example of Clock Policy Operation
Comparison of Algorithms

![Graph showing comparison of different page replacement algorithms](Figure 8.17 Comparison of Fixed-Allocation, Local Page Replacement Algorithms)
Figure 8.18  The Clock Page-Replacement Algorithm [GOLD89]
Combined Examples

Page address stream

<table>
<thead>
<tr>
<th>OPT</th>
<th>LRU</th>
<th>FIFO</th>
<th>CLOCK</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2*</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td>2*</td>
<td>3*</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>5</td>
<td>2*</td>
</tr>
<tr>
<td>1</td>
<td>5</td>
<td>5</td>
<td>3</td>
</tr>
<tr>
<td>5</td>
<td>5</td>
<td>2</td>
<td>3*</td>
</tr>
<tr>
<td>2</td>
<td>3*</td>
<td>3</td>
<td>3*</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>3</td>
<td>3*</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>3</td>
<td>3*</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>3</td>
<td>3*</td>
</tr>
<tr>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5</td>
</tr>
</tbody>
</table>

F = page fault occurring after the frame allocation is initially filled

Figure 8.15  Behavior of Four Page Replacement Algorithms
Page Buffering

- Improves paging performance and allows the use of a simpler page replacement policy

  A replaced page is not lost, but rather assigned to one of two lists:

  - Free page list
  - Modified page list

  - Free page list: list of page frames available for reading in pages
  - Modified page list: pages are written out in clusters
Replacement Policy and Cache Size

- With large caches, replacement of pages can have a performance impact.
  - if the page frame selected for replacement is in the cache, that cache block is lost as well as the page that it holds.
  - in systems using page buffering, cache performance can be improved with a policy for page placement in the page buffer.
- most operating systems place pages by selecting an arbitrary page frame from the page buffer.
Resident Set Management

- The OS must decide how many pages to bring into main memory
  - the smaller the amount of memory allocated to each process, the more processes can reside in memory
  - small number of pages loaded increases page faults
  - beyond a certain size, further allocations of pages will not affect the page fault rate
Resident Set Size

**Fixed-allocation**
- Gives a process a fixed number of frames in main memory within which to execute
- When a page fault occurs, one of the pages of that process must be replaced

**Variable-allocation**
- Allows the number of page frames allocated to a process to be varied over the lifetime of the process
The scope of a replacement strategy can be categorized as **global** or **local**
- both types are activated by a page fault when there are no free page frames

- **Local**
  - chooses only among the resident pages of the process that generated the page fault

- **Global**
  - considers all unlocked pages in main memory
## Resident Set Management Summary

<table>
<thead>
<tr>
<th></th>
<th>Local Replacement</th>
<th>Global Replacement</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Fixed Allocation</strong></td>
<td>• Number of frames allocated to a process is fixed.</td>
<td>• Not possible.</td>
</tr>
<tr>
<td></td>
<td>• Page to be replaced is chosen from among the frames allocated to that process.</td>
<td></td>
</tr>
<tr>
<td><strong>Variable Allocation</strong></td>
<td>• The number of frames allocated to a process may be changed from time to time to maintain the working set of the process.</td>
<td>• Page to be replaced is chosen from all available frames in main memory; this causes the size of the resident set of processes to vary.</td>
</tr>
<tr>
<td></td>
<td>• Page to be replaced is chosen from among the frames allocated to that process.</td>
<td></td>
</tr>
</tbody>
</table>
Fixed Allocation, Local Scope

- Necessary to decide ahead of time the amount of allocation to give a process
- If allocation is too small, there will be a high page fault rate

If allocation is too large, there will be too few programs in main memory
- increased processor idle time
- increased time spent in swapping
Variable Allocation
Global Scope

- Easiest to implement
  - adopted in a number of operating systems
- OS maintains a list of free frames
- Free frame is added to resident set of process when a page fault occurs
- If no frames are available the OS must choose a page currently in memory
- One way to counter potential problems is to use page buffering
When a new process is loaded into main memory, allocate to it a certain number of page frames as its resident set.

When a page fault occurs, select the page to replace from among the resident set of the process that suffers the fault.

Reevaluate the allocation provided to the process and increase or decrease it to improve overall performance.
Decision to increase or decrease a resident set size is based on the assessment of the likely future demands of active processes.

Key elements:
- criteria used to determine resident set size
- the timing of changes
<table>
<thead>
<tr>
<th>Sequence of Page References</th>
<th>Window Size, $\Delta$</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>2</td>
</tr>
<tr>
<td>24</td>
<td>24</td>
</tr>
<tr>
<td>15</td>
<td>24 15</td>
</tr>
<tr>
<td>18</td>
<td>24 15 18</td>
</tr>
<tr>
<td>23</td>
<td>24 15 18</td>
</tr>
<tr>
<td>24</td>
<td>23 24</td>
</tr>
<tr>
<td>17</td>
<td>23 24</td>
</tr>
<tr>
<td>18</td>
<td>18 24</td>
</tr>
<tr>
<td>24</td>
<td>18 24</td>
</tr>
<tr>
<td>18</td>
<td>18 24</td>
</tr>
<tr>
<td>17</td>
<td>18 24</td>
</tr>
<tr>
<td>17</td>
<td>18 17</td>
</tr>
<tr>
<td>17</td>
<td>17 18</td>
</tr>
<tr>
<td>15</td>
<td>17 15</td>
</tr>
<tr>
<td>24</td>
<td>17 15</td>
</tr>
<tr>
<td>17</td>
<td>17 15</td>
</tr>
<tr>
<td>24</td>
<td>17 15 24</td>
</tr>
<tr>
<td>18</td>
<td>24 17 18</td>
</tr>
</tbody>
</table>

**Figure 8.19**

**Working Set of Process as Defined by Window Size**
Page Fault Frequency (PFF)

- Requires a use bit to be associated with each page in memory
- Bit is set to 1 when that page is accessed
- When a page fault occurs, the OS notes the virtual time since the last page fault for that process
- Does not perform well during the transient periods when there is a shift to a new locality
Evaluates the working set of a process at sampling instances based on elapsed virtual time

Driven by three parameters:

- the minimum duration of the sampling interval
- the maximum duration of the sampling interval
- the number of page faults that are allowed to occur between sampling instances
Cleaning Policy

- Concerned with determining when a modified page should be written out to secondary memory

Demand Cleaning

- A page is written out to secondary memory only when it has been selected for replacement

Precleaning

- Allows the writing of pages in batches
Load Control

- Determines the number of processes that will be resident in main memory
  - multiprogramming level

- Critical in effective memory management

- Too few processes, many occasions when all processes will be blocked and much time will be spent in swapping

- Too many processes will lead to thrashing
Multiprogramming

Figure 8.21 Multiprogramming Effects
If the degree of multiprogramming is to be reduced, one or more of the currently resident processes must be swapped out.

Six possibilities exist:

- Lowest-priority process
- Faulting process
- Last process activated
- Process with the smallest resident set
- Largest process
- Process with the largest remaining execution window
Unix

- Intended to be machine independent so its memory management schemes will vary
  - early Unix: variable partitioning with no virtual memory scheme
  - current implementations of UNIX and Solaris make use of

SVR4 and Solaris use two separate schemes:

- paging system
- kernel memory allocator
Paging System and Kernel Memory Allocator

Paging system
- provides a virtual memory capability that allocates page frames in main memory to processes
- allocates page frames to disk block buffers

Kernel Memory Allocator
- allocates memory for the kernel
UNIX SVR4
Memory Management Formats

Figure 8.22  UNIX SVR4 Memory Management Formats
Table 8.6

UNIX SVR4
Memory
Management
Parameters
(page 1 of 2)

<table>
<thead>
<tr>
<th>Page Table Entry</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Page frame number</strong></td>
</tr>
<tr>
<td><strong>Age</strong></td>
</tr>
<tr>
<td><strong>Copy on write</strong></td>
</tr>
<tr>
<td><strong>Modify</strong></td>
</tr>
<tr>
<td><strong>Reference</strong></td>
</tr>
<tr>
<td><strong>Valid</strong></td>
</tr>
<tr>
<td><strong>Protect</strong></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Disk Block Descriptor</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Swap device number</strong></td>
</tr>
<tr>
<td><strong>Device block number</strong></td>
</tr>
<tr>
<td><strong>Type of storage</strong></td>
</tr>
</tbody>
</table>
Table 8.6

UNIX SVR4 Memory Management Parameters (page 2 of 2)

<table>
<thead>
<tr>
<th>Page Frame Data Table Entry</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Page state</strong></td>
</tr>
<tr>
<td>Indicates whether this frame is available or has an associated page. In the latter case, the status of the page is specified: on swap device, in executable file, or DMA in progress.</td>
</tr>
<tr>
<td><strong>Reference count</strong></td>
</tr>
<tr>
<td>Number of processes that reference the page.</td>
</tr>
<tr>
<td><strong>Logical device</strong></td>
</tr>
<tr>
<td>Logical device that contains a copy of the page.</td>
</tr>
<tr>
<td><strong>Block number</strong></td>
</tr>
<tr>
<td>Block location of the page copy on the logical device.</td>
</tr>
<tr>
<td><strong>Pfdata pointer</strong></td>
</tr>
<tr>
<td>Pointer to other pfdata table entries on a list of free pages and on a hash queue of pages.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Swap-Use Table Entry</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Reference count</strong></td>
</tr>
<tr>
<td>Number of page table entries that point to a page on the swap device.</td>
</tr>
<tr>
<td><strong>Page/storage unit number</strong></td>
</tr>
<tr>
<td>Page identifier on storage unit.</td>
</tr>
</tbody>
</table>
Page Replacement

- The page frame data table is used for page replacement
- Pointers are used to create lists within the table
  - all available frames are linked together in a list of free frames available for bringing in pages
  - when the number of available frames drops below a certain threshold, the kernel will steal a number of frames to compensate
"Two Handed" Clock Page Replacement

Figure 8.23 Two-Handed Clock Page-Replacement Algorithm
The kernel generates and destroys small tables and buffers frequently during the course of execution, each of which requires dynamic memory allocation.

Most of these blocks are significantly smaller than typical pages (therefore paging would be inefficient)

Allocations and free operations must be made as fast as possible
Lazy Buddy

- Technique adopted for SVR4
- UNIX often exhibits steady-state behavior in kernel memory demand
  - i.e. the amount of demand for blocks of a particular size varies slowly in time
- Defers coalescing until it seems likely that it is needed, and then coalesces as many blocks as possible
Lazy Buddy System Algorithm

![Algorithm Description]

Figure 8.24 Lazy Buddy System Algorithm
Linux Memory Management

- Shares many characteristics with Unix
- Is quite complex

Two main aspects
- process virtual memory
- kernel memory allocation
Linux Virtual Memory

Three level page table structure:

- Page directory:
  - Process has a single page directory
  - Each entry points to one page of the page middle directory
  - Must be in main memory for an active process

- Page middle directory:
  - May span multiple pages
  - Each entry points to one page in the page table

- Page table:
  - May also span multiple pages
  - Each entry refers to one virtual page of the process
Address Translation

Figure 8.25  Address Translation in Linux Virtual Memory Scheme
Linux Page Replacement

- Based on the clock algorithm
- The use bit is replaced with an 8-bit age variable
  - incremented each time the page is accessed
- Periodically decrements the age bits
  - a page with an age of 0 is an “old” page that has not been referenced is some time and is the best candidate for replacement
- A form of least frequently used policy
Kernel Memory Allocation

- Kernel memory capability manages physical main memory page frames
- primary function is to allocate and deallocate frames for particular uses

Possible owners of a frame include:

- user-space processes
- dynamically allocated kernel data
- static kernel code
- page cache

- A buddy algorithm is used so that memory for the kernel can be allocated and deallocated in units of one or more pages
- Page allocator alone would be inefficient because the kernel requires small short-term memory chunks in odd sizes
- Slab allocation
  - used by Linux to accommodate small chunks
Windows Memory Management

- Virtual memory manager controls how memory is allocated and how paging is performed
- Designed to operate over a variety of platforms
- Uses page sizes ranging from 4 Kbytes to 64 Kbytes
On 32 bit platforms each user process sees a separate 32 bit address space allowing 4 Gbytes of virtual memory per process
  - by default half is reserved for the OS
- Large memory intensive applications run more effectively using 64-bit Windows
- Most modern PCs use the AMD64 processor architecture which is capable of running as either a 32-bit or 64-bit system
32-Bit Windows Address Space

Figure 8.26 Windows Default 32-bit Virtual Address Space
On creation, a process can make use of the entire user space of almost 2 Gbytes

This space is divided into fixed-size pages managed in contiguous regions allocated on 64 Kbyte boundaries

Regions may be in one of three states:

- available
- reserved
- committed
Resident Set Management System

- Windows uses variable allocation, local scope
- When activated, a process is assigned a data structure to manage its working set
- Working sets of active processes are adjusted depending on the availability of main memory
Desirable to:
- maintain as many processes in main memory as possible
- free programmers from size restrictions in program development

With virtual memory:
- all address references are logical references that are translated at run time to real addresses
- a process can be broken up into pieces
- two approaches are paging and segmentation
- management scheme requires both hardware and software support