Outline

- Principle of locality
- Paging
  - Effect of page size
- Segmentation
- Segmentation and paging
- Role of memory management in OS
  - Fetch, placement and replacement policy
- Page buffering
- Resident set size
- Load Control
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
  - since processes do not fit in memory some part of it is on memory and some part on hard disk.
<table>
<thead>
<tr>
<th><strong>Terminology</strong></th>
</tr>
</thead>
</table>

| **Virtual memory** | 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. |
| **Virtual address** | The address assigned to a location in virtual memory to allow that location to be accessed as though it were part of main memory. |
| **Virtual address space** | The virtual storage assigned to a process. |
| **Address space** | The range of memory addresses available to a process. |
| **Real address** | The address of a storage location in main memory. |
Hardware and Control Structures

- Two characteristics fundamental to memory management [paging/segmentation]:
  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

- Implication: If these two characteristics are satisfied, it is not necessary that all of the pages or segments of a process be in main memory during execution
Execution of a Process (I)

- Operating system brings into main memory a few pieces (page or segment) of the program
- Process issues a \textbf{virtual} address
- It is converted into \textbf{physical} Address
- If the required data is not in memory \textit{[access fault]}:
  - An interrupt is generated when an address is needed that is not in main memory
  - Operating system places the process in a blocking state

- \textbf{Resident set} - portion of process that is in main memory

Continued . . .
To bring the piece of process that contains the logical address into main memory

- operating system issues I/O 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
- When it runs again, required data is in memory

Observation:

✶ Time for this situation (access fault) must be very small
✶ Translation to physical address must be done in hardware
Virtual Memory: Benefits

- **Virtual memory** – program uses virtual memory which can be partially loaded into physical memory

- **Benefits:**
  - Only *part of the program* needs to be in memory for execution
    - more *concurrent* programs
  - Logical address space can therefore be *much larger than physical address space*
    - execute programs larger than RAM size
  - Easy *sharing* of address spaces by several processes
    - Library or a memory segment can be shared
  - Allows for more *efficient process creation*
Example: Simple Paging

Virtual Address (35 bits because of 32G virtual memory)
Number of pages we have?
8M (because we have 23 bits for page#)
Size of page table?
32M

Size of all page tables together: 32M * 256 = 8G
Two-Level Hierarchical Page Table: Example

8 level 2 page tables.

What is the size of Level 1 page table: 8 entries * 4 bytes per entry
Number of entries in the level 2 page table: $2^{20}$
Size of each level 2 page table entry: 4 bytes
Size of 1 level 2 page table: $2^{22}$

What is the size of ALL level 2 page tables? 8 (number of level 2 page tables) * $2^{22} = 2^{25}$

Size of all page tables together: $256 * 2^{25} = 2^{33} = 8G$

Only part of the program needs to be in memory for execution

- more concurrent programs
The key to the success of this approach to memory management is that instruction and data references in a program sequence tend to cluster [recall locality from Chapter 1]

Hence, only a portion of the process need be in memory for efficient execution. This portion is called the working set.

A potentially confusing issue: Working set vs resident set

- **Working set** = set that SHOULD be in memory for efficient execution
- **Resident set** = set that IS in memory.

Hopefully, they are same but need not be.

Moreover, it is possible to predict with good accuracy which instructions/data
Advantages of Partial Loading

- 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
  - it is even possible to use more bits for logical addresses than the bits needed for addressing the physical memory
**Possibility of Thrashing**

To accommodate as many processes as possible, only a few pieces of each process is maintained in main memory

- But main memory may be full: when the OS brings one piece in, it must swap one piece out
- The OS must not swap out a piece of a process just before that piece is needed
- If this scenario occurs too often the result is thrashing:

**Thrashing:** A state in which the processor spends most of its time swapping process pieces in and out of memory rather than executing user instructions

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
Support Needed for Virtual Memory

Two ingredients 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

We will first discuss the hardware aspects and then discuss algorithms used by OS.
Paging
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.
- A bit is needed to indicate whether the page is in main memory or not.

P (Present): the corresponding page is in memory or not.
M (Modify): the contents of the corresponding page have been *altered* since the page was last loaded into memory.
Modify Bit in Page Table

- Modify bit is needed to indicate if the page has been altered since it was last loaded into main memory.

- If no change has been made, the page does not have to be written to the disk when it needs to be swapped out [saving IO time].

Other Control bits

- Depends on OS
- Used if protection is managed at the page level
  - E.g., read/only
  - E.g., kernel only
Paging Address Translation: Hardware Implementation

Figure 8.3 Address Translation in a Paging System
Present Bit

- With each page table entry a Present bit is associated ($P \Rightarrow$ in-memory, $n \Rightarrow$ not-in-memory)
- Initially present bit is set to $n$ on all entries
- Example of a page table snapshot:

<table>
<thead>
<tr>
<th>Frame #</th>
<th>Present bit</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>$n$</td>
</tr>
<tr>
<td></td>
<td>$n$</td>
</tr>
<tr>
<td></td>
<td>$n$</td>
</tr>
<tr>
<td></td>
<td>$n$</td>
</tr>
<tr>
<td></td>
<td>....</td>
</tr>
<tr>
<td></td>
<td>$n$</td>
</tr>
<tr>
<td></td>
<td>$n$</td>
</tr>
</tbody>
</table>

- During address translation, if present bit in page table entry is $n \Rightarrow$ page fault or access fault
The page table itself may be so large that it does not fit in memory (or at least takes up a lot of memory)

- **Solutions:**
  - *Two level scheme*, using page directory to point to page tables (used on Pentium)
  - *Inverted page table*: use hash table to map page numbers to page table entry -- one entry per real frame instead of virtual frame (used on PowerPC and IBM AS/400)
Virtual Memory for Page Table

- The entire page table may take up too much main memory.
- Page tables are also stored in virtual memory.
- When a process is running, part of its page table is in main memory.
Page Table Size: Example

- Size of virtual memory $= 2^{32}$ Byte $= 4$-Gbyte (byte level addressing)
- Size of each page: $2^{12}$ Byte $= 4$-Kbyte
- Number of pages: $2^{20}$
- Size of page table entry: 4 byte
- Size of page table: $2^{20} \times 4$ byte $= 4$-Mbyte ($2^{22}$)!!!
- Number of pages for page table: $2^{10}$ pages!
Two-Level Hierarchical Page Table

- 10 bits
- $2^{10}$ page table entries = $2^{12}$ = 4 Kbyte
- $2^{10}$ pages

Figure 8.4 A Two-Level Hierarchical Page Table
Address Translation

Virtual Address

10 bits 10 bits 12 bits

Root page table ptr

Program

Paging Mechanism

Root page table (contains 1024 PTEs)

4-kbyte page table (contains 1024 PTEs)

Main Memory

Frame # Offset

4-Kbyte \((2^{12})\) Pages

Figure 8.5 Address Translation in a Two-Level Paging System
Virtual memory = 32G ($2^{35}$)
Page size = 4K
Physical memory = 8G
Number of processes = 256
Number of entries in level 1 page table: 8 entries

Number of pages we have?
What is the size of Level 1 page table?
Size of level 2 page table?
Size of all page tables together?
Two-Level Hierarchical Page Table: Example

8 level 2 page tables.

What is the size of Level 1 page table: 8 entries * 4 bytes per entry

Size of level 2 page table = ??

Number of entries in the level 2 page table: $2^{20}$
Size of each level 2 page table entry: 4 bytes
Size of 1 level 2 page table: $2^{22}$

What is the size of ALL level 2 page tables? 8 (number of level 2 page tables) * $2^{22} = 2^{25}$

Size of ALL page tables of ALL processes: $256 * 2^{25} = 2^{33} = 8G$
Example: Simple Paging

Virtual Address (35 bits because of 32G virtual memory)
Number of pages we have?
8M (because we have 23 bits for page#)
Size of page table?
32M
Size of all page tables together:
32M * 256 = 8G
Physical Address (33 bits)
P (1bit)
M (1bit)
Frame#: 21 bits
Need a min of 23 bits for page table entry
4bytes gives me 32 bits, I have 9 for other control bits

The page table itself may be so large!
Page tables are subject to paging!
Page tables are also stored in virtual memory
Inverted Page Table

- 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
Sample Inverted Page Table

<table>
<thead>
<tr>
<th>Process 5</th>
<th>Page 8</th>
<th>More data</th>
</tr>
</thead>
<tbody>
<tr>
<td>Process 9</td>
<td>Page 10</td>
<td>More data</td>
</tr>
<tr>
<td>Process 5</td>
<td>Page 20</td>
<td>...</td>
</tr>
</tbody>
</table>

What is in Frame 0

What is in Frame 1
Implementing the previous idea more efficiently
- Page number
- Process identifier
- Control bits
- Chain pointer

Each entry in the page table includes:

- **Page number**
- **Process identifier**
  - the process that owns this page
- **Control bits**
  - includes flags and protection and locking information
- **Chain pointer**
  - the index value of the next entry in the chain
Inverted Page Table Structure

Figure 8.6 Inverted Page Table Structure
Inverted Page Table: Example

Virtual memory = 32G ($2^{35}$)
Page size = 4K
Physical memory = 8G
Number of processes = 256

Number of pages we have?
What is the size of page table?
Size of all page tables together?
Inverted Page Table: Example

Size of the inverted page table?

Number of entries in the inverted page table: \(2^M (2^{21})\)

What is the size of each inverted page table entry:

Page #: 23 bits (4 bytes being generous)
Process ID: 32 bits (4 bytes, just made up because reasonable)
Pointers: …
Rough estimate: 16 bytes

Size of the inverted page table: \(2^{21} \times 16 = 2^{25} = 32 \text{M}\)
What happens if there is no free frame?

- Page replacement – find some page in memory, but not really in use, swap it out
  - Algorithm? Which page should be remove?
  - performance – want an algorithm which will result in minimum number of page faults

- With page replacement, same page may be brought into memory several times

- Prevent over-allocation of memory by modifying page-fault service routine to include page replacement
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

Which page currently in memory is to be replaced?

Most policies predict the future behavior on the basis of past behavior
Basic Algorithms

Algorithms used for the selection of a page to replace:

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

• Evaluate algorithm by running it on a particular string of memory references (reference string) and computing the number of page faults on that string

Suppose the page address stream formed by executing a program is

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

This means that the first page referenced is 2,
– the second page referenced is 3, etc.

3 frames
LRU Example (I)

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

LRU:

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

Figure 8.15  Behavior of Four Page Replacement Algorithms

LRU Example (II)

7 0 1 2 0 3 0 4 2 3 0 3 2 1 2 0 1 7 0 1
LRU Example

<table>
<thead>
<tr>
<th>reference string</th>
</tr>
</thead>
<tbody>
<tr>
<td>7 0 1 2 0 3 0 4 2 3 0 3 2 1 2 0 1 7 0 1</td>
</tr>
<tr>
<td>7 7 7 2 2 4 4 4 0 1 1 1 0 1 0 0</td>
</tr>
<tr>
<td>0 0 0 0 0 3 3 3 3 0 0 0</td>
</tr>
<tr>
<td>1 1 1 1 1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>page frames</th>
</tr>
</thead>
<tbody>
<tr>
<td>7 0 3 2</td>
</tr>
<tr>
<td>3 2 2 2</td>
</tr>
<tr>
<td>2 2 2 7</td>
</tr>
</tbody>
</table>

9 page faults
LRU Example

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

8 page faults
Implementation of LRU

- Each page could be tagged (in the page table entry) with the time at each memory reference.
- The LRU page is the one with the smallest time value (needs to be searched at each page fault)
- This would require expensive hardware and a great deal of overhead.
- Consequently very few computer systems provide sufficient hardware support for true LRU replacement policy
- Other algorithms are used instead
First-in-First-out (FIFO)

- Treats page frames allocated to a process as a **circular buffer**
- Pages are removed in round-robin style
  - simplest replacement policy to implement
- Page that has been in memory the longest is replaced
- These pages may be needed again very soon
FIFO Example (I)

Figure 8.15  Behavior of Four Page Replacement Algorithms

Page address stream:

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

FIFO:

2  3  3  3  3  2  4  4  4  4  2

F = page fault occurring after the frame allocation is initially filled.
FIFO Example (II)

- 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)

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

9 page faults

- 4 frames

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

10 page faults

- Belady’s Anomaly: more frames ⇒ more page faults
FIFO Example (III)
### FIFO Example

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

#### Page Frames
Optimal Policy

- Selects the page for which the time to the next reference is the **longest** (need perfect knowledge of future events)

- not a real algorithm, but an ideal standard

- Impossible to have perfect knowledge of future events
Optimal Policy

How many page faults?

The optimal policy produces three page faults after the frame allocation has been filled.
Clock Policy

- Each frame has a *use bit*
- When a page is first loaded in memory or referenced, the use bit is set to 1
- When the page is referenced, the use bit is set to 1
- When it is time to replace a page, the OS scans the set flipping all 1’s to 0
- When it is time to replace a page, the first frame encountered with the use bit set to 0 is replaced.
- Any frame with a use bit of 1 is **passed over** by the algorithm
- During the search for replacement, each use bit set to 1 is changed to 0

An attempt to provide cheap LRU
Clock Policy Example

Figure 8.15  Behavior of Four Page Replacement Algorithms
Some Important Steps in Clock Policy

<table>
<thead>
<tr>
<th></th>
<th>2*</th>
<th>2</th>
<th>2</th>
<th>2</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>3*</td>
<td>→3*</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>1*</td>
<td>1*</td>
<td>→1*</td>
<td>1</td>
<td>1</td>
<td></td>
</tr>
</tbody>
</table>
### Combined Examples

<table>
<thead>
<tr>
<th>Page address stream</th>
<th>2</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>5</th>
<th>2</th>
<th>4</th>
<th>5</th>
<th>3</th>
<th>2</th>
<th>5</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td>OPT</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>LRU</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>FIFO</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>CLOCK</td>
<td>2*</td>
<td>3*</td>
<td>2*</td>
<td>5*</td>
<td>5*</td>
<td>5*</td>
<td>3*</td>
<td>3*</td>
<td>3*</td>
<td>3*</td>
<td>3*</td>
<td>3*</td>
</tr>
</tbody>
</table>

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

**Figure 8.15** Behavior of Four Page Replacement Algorithms
Translation Lookaside Buffer (TLB)
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, a special high-speed cache called a translation lookaside buffer (TLB) is set up for page table entries.

- Contains page table entries that have been most recently used.
Translation Lookaside Buffer

- Given a virtual address, processor examines the TLB
- If page table entry is present (TLB hit), the frame number is retrieved and the real address is formed
- If page table entry is not found in the TLB (TLB miss), the page number is used to index the process page table
Translation Lookaside Buffer

- First checks if page is already in main memory
  - If not in main memory a **page fault** is issued
- The TLB is updated to include the new page entry
Use of a TLB

Figure 8.7 Use of a Translation Lookaside Buffer
Figure 8.8  Operation of Paging and Translation Lookaside Buffer (TLB)
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.
Figure 8.9  Direct versus Associative Lookup for Page Table Entries
Figure 8.10  Translation Lookaside Buffer and Cache Operation
Outline

- Principle of locality
- Paging
  - Effect of page size
- Segmentation
- Segmentation and paging
- Role of memory management in OS
  - Fetch, placement and replacement policy
- Page buffering
- Resident set size
- Load Control
Page Size

- The smaller the page size, the lesser the amount of **internal fragmentation**
- The smaller the page size, the more pages required per process
  - more pages per process means **larger page tables**
  - Larger page tables means large portion of page tables in virtual memory
  - Secondary memory is designed to efficiently transfer large blocks of data so a large page size is better

**PAGE SIZE** An important hardware design decision is the size of page to be used.
Page Size

- Small page size, large number of pages will be found in main memory.

- As time goes on during execution, the pages in memory will all contain portions of the process near recent references. Page faults low.

- Increased page size causes pages to contain locations further from any recent reference. Page faults rise.
Contemporary programming techniques (OO & multi-threading) used in large programs tend to decrease the locality of references within a process.

Main memory is getting larger and address space used by applications is also growing.

Most obvious on personal computers where applications are becoming increasingly complex.
Segmentation
Segmentation

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

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

- 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
Segment Tables

- Corresponding segment in main memory
  - 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
Figure 8.12  Address Translation in a Segmentation System
In each segment table entry we have both the starting address and length of the segment. The segment can thus dynamically grow or shrink as needed. Address validity can easily be checked with the length field.

But variable length segments introduce external fragmentation and are more difficult to swap in and out...

It is natural to provide protection and sharing at the segment level since segments are visible to the programmer (pages are not).

Useful protection bits in segment table entry:
- read-only/read-write bit
- supervisor/user bit
Segmentation and Paging
Combined Paging and Segmentation

- Paging is transparent to the programmer
- Segmentation is visible to the programmer
- Each segment is broken into fixed-size pages

(c) Combined segmentation and paging
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
Figure 8.13 Address Translation in a Segmentation/Paging System
Role of memory management in Operating System
Operating System: Role in Memory Management

- Memory management software depends on whether the hardware supports paging, segmentation, or both.
- Pure segmentation systems are rare. Segments are usually paged -- memory management issues are then those of paging.
- We shall thus concentrate on issues associated with paging.
- To achieve good performance we need a low page fault rate.
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
Prepaging

Different Memory Management Policies

- Fetch policy
- Placement policy
- Replacement policy
### 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></td>
<td>Replacement Scope</td>
</tr>
<tr>
<td></td>
<td>Global</td>
</tr>
<tr>
<td></td>
<td>Local</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Placement Policy</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Replacement Policy</strong></td>
</tr>
<tr>
<td>Basic Algorithms</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>Optimal</td>
</tr>
<tr>
<td>Least recently used (LRU)</td>
</tr>
<tr>
<td>First-in-first-out (FIFO)</td>
</tr>
<tr>
<td>Clock</td>
</tr>
<tr>
<td>Page Buffering</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Cleaning Policy</th>
</tr>
</thead>
<tbody>
<tr>
<td>Demand</td>
</tr>
<tr>
<td>Precleaning</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Load Control</th>
</tr>
</thead>
<tbody>
<tr>
<td>Degree of multiprogramming</td>
</tr>
</tbody>
</table>
Fetch Policy

- Determines when a page should be brought into memory

Two main types:

- **Demand Paging**
  - paging only brings pages into main memory when a reference is made to a location on the page
  - Many page faults when process first started

- **Prepaging**
  - brings in more pages than needed
  - More efficient to bring in pages that reside contiguously on the disk
Implementing Virtual Memory

- Virtual memory can be implemented via:
  - **Demand paging**
    - Bring pages into memory when they are used, i.e. allocate memory for pages when they are used
  - **Demand segmentation**
    - Bring segments into memory when they are used, i.e. allocate memory for segments when they are used.
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 (page is not in used portion of address space) ⇒ abort
  – not-in-memory ⇒ bring to memory

• Pager never brings a page into memory unless page will be needed
Placement Policy

- Determines where in real memory a process piece is to reside
- Important design issue in a segmentation system (best-fit, first-fit, etc.)
- Paging or combined paging with segmentation placing is irrelevant (transparent) because hardware performs functions with equal efficiency
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