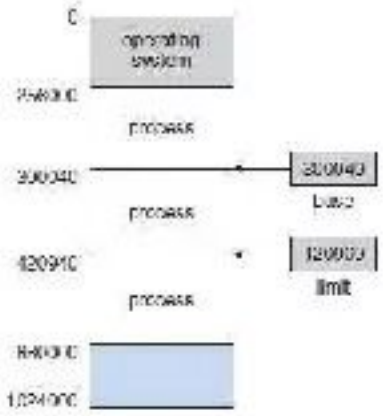
BASIC HARDWARE

- Program must be

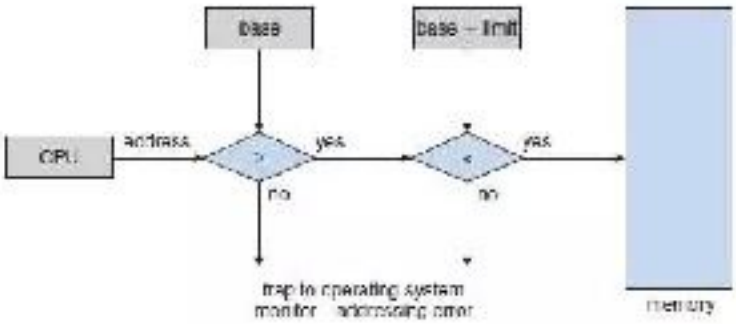
→ brought (from disk) into memory and

→ placed within a process for it to be run.

- Main-memory and registers are only storage CPU can access directly.
- Register access in one CPU clock.
- Main-memory can take many cycles.
- Cache sits between main-memory and CPU registers.
- Protection of memory required to ensure correct operation.
- A pair of base- and limit-registers define the logical (virtual) address space.



A base and a limit-register define a logical-address space



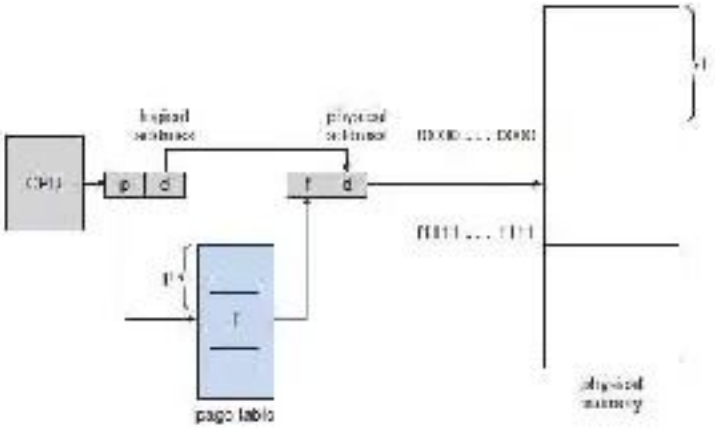
Hardware address protection with base and limit-registers

Paging

• Paging is a memory-management scheme.
• This permits the physical-address space of a process to be non-contiguous.
• This also solves the considerable problem of fitting memory-chunks of varying sizes onto the
backing-store.
• Traditionally: Support for paging has been handled by hardware.
Recent designs: The hardware & OS are closely integrated.

Basic Method of Paging

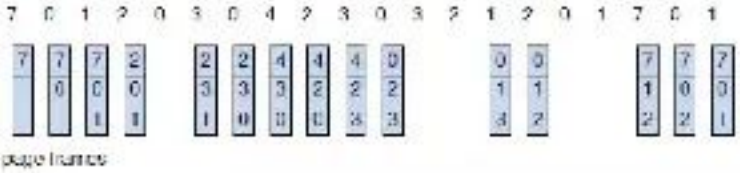
• Physical-memory is broken into fixed-sized blocks called **frames**(Figure 5.7).
Logical-memory is broken into same-sized blocks called **pages.**
• When a process is to be executed, its pages are loaded into any available memory-frames from the
backing-store.
• The backing-store is divided into fixed-sized blocks that are of the same size as the memory-frames.



Paging hardware

• The page-table contains the base-address of each page in physical-memory.
• Address generated by CPU is divided into 2 parts (Figure 5.8):
1) **Page-number**(p) is used as an index to the page-table and
2) **Offset**(d) is combined with the base-address to define the physical-address.
This physical-address is sent to the memory-unit.

• Example: Consider the following references string with frames initially empty.



FIFO page-replacement algorithm

- The first three references(7, 0, 1) cause page-faults and are brought into these empty
frames.
- The next reference(2) replaces page 7, because page 7 was brought in first.
- Since 0 is the next reference and 0 is already in memory, we have no fault for this reference.
- The first reference to 3 results in replacement of page 0, since it is now first in line.
- This process continues till the end of string.
- There are fifteen faults altogether.

• Advantage:
1. Easy to understand & program.

• Disadvantages:
1. Performance is not always good (Figure 5.26).
2. A bad replacement choice increases the page-fault rate (Belady's anomaly).

• For some algorithms, the page-fault rate may increase as the number of allocated frames increases.
This is known as **Belady's anomaly**.

• Example: Consider the following reference string:
1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5
For this example, the number of faults for four frames (ten) is *greater* than the number of faults
for three frames (nine)!

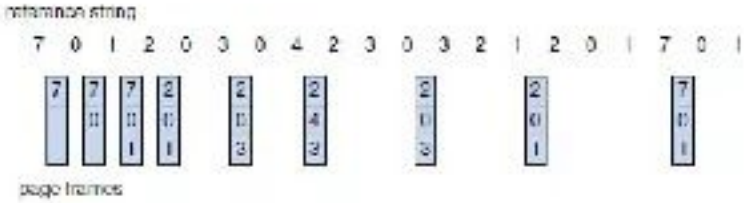
Page-fault curve for FIFO replacement on a reference string

Optimal Page Replacement

• Working principle: Replace the page that will not be used for the longest period of time (Figure 5.27).

• This is used mainly to solve the problem of Belady's Anamoly.

• This has the lowest page-fault rate of all algorithms.
• Consider the following reference string:



Optimal page-replacement algorithm

- The first three references cause faults that fill the three empty frames.
- The reference to page 2 replaces page 7, because page 7 will not be used until reference 18.
- The page 0 will be used at 5, and page 1 at 14.
- With only nine page-faults, optimal replacement is much better than a FIFO algorithm, which
results in fifteen faults.

• Advantage:
1. Guarantees the lowest possible page-fault rate for a fixed number of frames.

• Disadvantage:
1. Difficult to implement, because it requires future knowledge of the reference string.

LRU Page Replacement

• The key difference between FIFO and OPT:
→ FIFO uses the time when a page was brought into memory.
→ OPT uses the time when a page is to be used.
• Working principle: Replace the page that has not been used for the longest period of time.
• Each page is associated with the time of that page's last use (Figure 5.28).
• Example: Consider the following reference string:

LRU page-replacement algorithm

- The first five faults are the same as those for optimal replacement.
- When the reference to page 4 occurs, LRU sees that of the three frames, page 2 was used
least recently. Thus, the LRU replaces page 2.
- The LRU algorithm produces twelve faults.

• Two methods of implementing LRU:

1. Counters

- Each page-table entry is associated with a **time-of-use** field.
- A **counter(or logical clock)** is added to the CPU.
- The clock is incremented for every memory-reference.
- Whenever a reference to a page is made, the contents of the clock register are copied to the time-of-use field in the page-table entry for that page.
- We replace the page with the smallest time value.

2. Stack

- Keep a stack of page-numbers (Figure 5.29).
- Whenever a page is referenced, the page is removed from the stack and put on the top.
- The most recently used page is always at the top of the stack.

The least recently used page is always at the bottom.

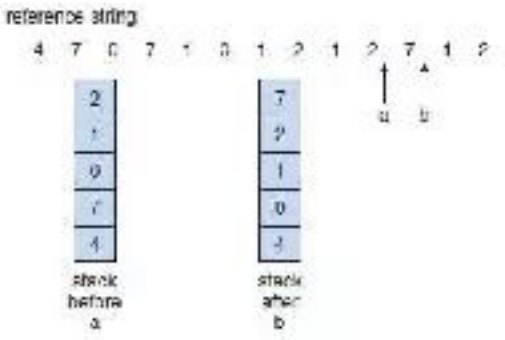
- Stack is best implement by a doubly linked-list.
- Advantage:

1. Does not suffer from Belady's anomaly.

- Disadvantage:

1. Few computer systems provide sufficient h/w support for true LRU page replacement.

- Both LRU & OPT are called stack algorithms.



Use of a stack to record the most recent page references

LRU-Approximation Page Replacement

- Some systems provide a **reference bit** for each page.
- Initially, all bits are cleared(to 0) by the OS.
- As a user-process executes, the bit associated with each page referenced is set (to 1) by the hardware.
- By examining the reference bits, we can determine

→ which pages have been used and

→ which have not been used.

- This information is the basis for many page-replacement algorithms that approximate LRU replacement.

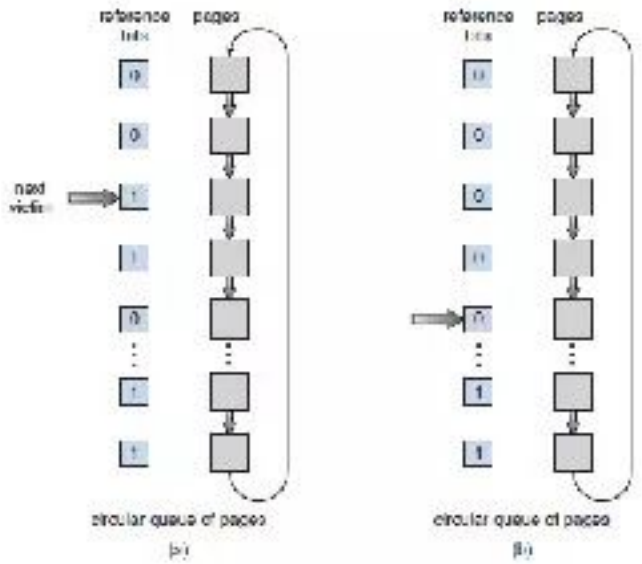
Additional-Reference-Bits Algorithm

- We can gain additional ***ordering information*** by recording the reference bits at regular intervals.
- A 8-bit byte is used for each page in a table in memory.

• At regular intervals, a timer-interrupt transfers control to the OS.
• The OS shifts the reference bit for each page into the high-order bit of its 8-bit byte.
• These 8-bit shift registers contain the history of page use, for the last eight time periods.
• Examples:
00000000 - This page has not been used in the last 8 time units (800 ms).
11111111 - Page has been used every time unit in the past 8 time units.
11000100 has been used more recently than 01110111.
• The page with the lowest number is the LRU page, and it can be replaced.
• If numbers are equal, FCFS is used

Second-Chance Algorithm

• The number of bits of history included in the shift register can be varied to make the updating as fast
as possible.
• In the extreme case, the number can be reduced to zero, leaving only the reference bit itself. This
algorithm is called the **second-chance algorithm.**
• Basic algorithm is a FIFO replacement algorithm.
• Procedure:
- When a page has been selected, we inspect its reference bit.
- If reference bit=0, we proceed to replace this page.
- If reference bit=1, we give the page a second chance & move on to select next FIFO page.
- When a page gets a second chance, its reference bit is cleared, and its arrival time is reset.



Second-chance (clock) page-replacement algorithm

• A circular queue can be used to implement the second-chance algorithm (Figure 5.30).
- *A* pointer (that is, a hand on the clock) indicates which page is *to* be replaced next.
- When a frame is needed, the pointer advances until it finds a page with a 0 reference bit.
- As it advances, it clears the reference bits.

☂Once a victim page is found, the page is replaced, and the new page is inserted in the circular
queue in that position.

Enhanced Second-Chance Algorithm

• We can enhance the second-chance algorithm by considering
1) Reference bit and 2) modify-bit.
• We have following 4 possible classes:
1. **(0, 0)** neither recently used nor modified -best page to replace.
2. **(0, 1)** not recently used hut modified-not quite as good, because the page will need to be
written out before replacement.
3. **(1, 0)** recently used but clean-probably will be used again soon.
4. **(1, 1)** recently used and modified -probably will be used again soon, and the page will be
need to be written out to disk before it can be replaced.
• Each page is in one of these four classes.
• When page replacement is called for, we examine the class to which that page belongs.
• We replace the first page encountered in the lowest nonempty class.

Counting-Based Page Replacement

1. LFU page-replacement algorithm

☂Working principle: The page with the smallest count will be replaced.
☂The reason for this selection is that an actively used page should have a large reference
count.
☂Problem:
When a page is used heavily during initial phase of a process but then is never used again.
Since it was used heavily, it has a large count and remains in memory even though it is no
longer needed.
Solution:
Shift the counts right by 1 bit at regular intervals, forming an exponentially decaying
average usage count.

2. MFU (Most Frequently Used) page-replacement algorithm

☂Working principle: The page with the smallest count was probably just brought in and has
yet to be used.

File Concepts

• A ***file*** is a named collection of related info. on secondary-storage.
• Commonly, file represents
→ program and
→ data.
• Data in file may be
→ numeric
→ alphabetic or
→ binary.
• Four types of file:

1. **Text file**: sequence of characters organized into lines.
2. **Source file**: sequence of subroutines & functions.
3. **Object file**: sequence of bytes organized into blocks.
4. **Executable file**: series of code sections.

File Attributes

1. Name

• The only information kept in human-readable form.

2. Identifier

• It is a unique number which identifies the file within file-system.
• It is in non-human-readable form.

3. Type

• It is used to identify different types of files.

4. Location

• It is a pointer to
→ device and
→ location of file.

5. Size

• Current-size of file in terms of bytes, words, or blocks.
• It also includes maximum allowed size.

6. Protection

• Access-control info. determines who can do
→ reading
→ writing and
→ executing.

7. Time, date, & user identification

• These info. can be kept for
→ creation
→ last modification and
→ last use.
• These data can be useful for
→ protection
→ security and
→ usage monitoring.
• Information about files are kept in the ***directory-structure***, which is maintained on the disk.

File Operations

1. Creating a file

☂Two steps are:
i) Find the space in the file-system for the file.
ii) An entry for the new file is made in the directory.

2. Writing a file

☂Make a system-call specifying both
→ file-name and
→ info. to be written to the file.
☂The system searches the directory to find the file's location. (The system keeps a *writepointer(*
wp) to the location in the file where the next write is to take place).
☂The write-pointer must be updated whenever a write-operation occurs.

3. Reading a file

- Make a system-call specifying both
→ file-name and
→ location of the next block of the file in the memory.
- The system searches the directory to find the file's location. (The system keeps a *readpointer(rp)* to the location in the file where the next read is to take place).
- The read-pointer must be updated whenever a read-operation occurs.
- Same pointer (rp & wp) is used for both read- & write-operations. This results in
→ saving space and
→ reducing system-complexity.

4. Repositioning within a file

- Two steps are:

i) Search the directory for the appropriate entry.
ii) Set the current-file-position to a given value.

- This file-operation is also known as ***file seek***.

5. Deleting a file

- Two steps are:

i) Search the directory for the named-file.
ii) Release all file-space and erase the directory-entry.

6. Truncating a file

- The contents of a file are erased but its attributes remain unchanged.
- Only file-length attribute is set to zero.

(Most of the above file-operations involve searching the directory for the entry associated with the file.
To avoid this constant searching, many systems require that an 'open' system-call be used before that
file is first used).

• The OS keeps a small table which contains info. about all open files (called **open-file table**).
• If a file-operation is requested, then
→ file is specified via an index into open-file table
→ so no searching is required.
• If the file is no longer actively used, then
→ process closes the file and
→ OS removes its entry in the open-file table.
• Two levels of internal tables:

1. Per-process Table

- Tracks all files that a process had opened.
- Includes access-rights to
→ file and
→ accounting info.
- Each entry in the table in turn points to a system-wide table

2. System-wide Table

- Contains process-independent info. such as
→ file-location on the disk
→ file-size and
→ access-dates.

Access Methods

Sequential Access

• This is based on a tape model of a file.
• This works both on
→ sequential-access devices and
→ random-access devices.
• Info. in the file is *processed in order* (Figure 6.2).

For ex: editors and compilers

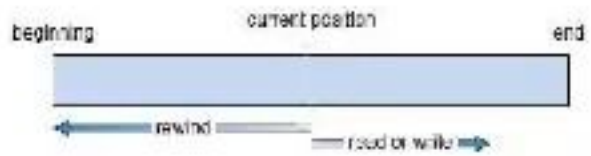
• Reading and writing are the 2 main operations on the file.
• File-operations:

1. read next

This is used to
→ read the next portion of the file and
→ advance a file-pointer, which tracks the I/O location.

2. write next

This is used to
→ append to the end of the file and
→ advance to the new end of file.



Sequential-access file

Direct Access (Relative Access)

• This is based on a disk model of a file (since disks allow random access to any file-block).
• A file is made up of fixed length *logical records*.
• Programs can read and write records rapidly in no particular order.
• Disadvantages:

1. Useful for immediate access to large amounts of info.
2. Databases are often of this type.

• File-operations include a relative block-number as parameter.
• The **relative block-number** is an index relative to the beginning of the file.
• File-operations (Figure 6.3):

1. read n
2. write n

where n is the block-number

• Use of relative block-numbers:
→ allows OS to decide where the file should be placed and
→ helps to prevent user from accessing portions of file-system that may not be part of his file.

sequential access	implementation for direct access
reset	cp = 0;
read_next	read cp; cp = cp + 1;
write_next	write cp; cp = cp + 1;

Simulation of sequential access on a direct-access file

Other Access Methods

• These methods generally involve constructing a **file-index**.

• The index contains pointers to the various blocks (like an index in the back of a book).

• To find a record in the file(Figure 6.4):

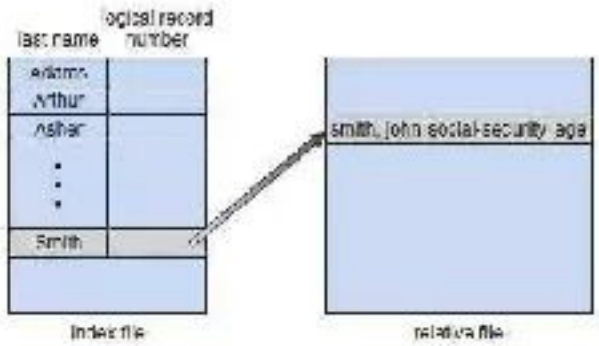
1. First, search the index and
2. Then, use the pointer to

→ access the file directly and

→ find the desired record.

• Problem: With large files, the index-file itself may become too large to be kept in memory.

Solution: Create an index for the index-file. (The primary index-file may contain pointers to secondary
index-files, which would point to the actual data-items).



Example of index and relative files

Disk Structure

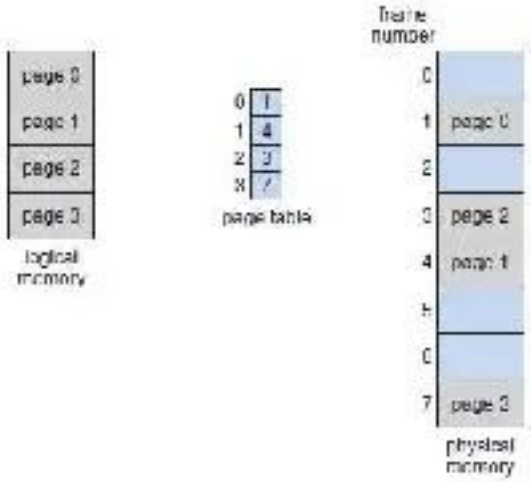
Modern magnetic disk drives are addressed as large one-dimensional arrays of **logical blocks**, where the logical block is the smallest unit of transfer. The size of a logical block is usually 512 bytes, although some disks can be **low-level formatted** to have a different logical block size, such as 1,024 bytes. This option is described in Section 10.5.1. The one-dimensional array of logical blocks is mapped onto the sectors of the disk sequentially. Sector 0 is the first sector of the first track on the outermost cylinder. The mapping proceeds in order through that track, then through the rest of the tracks in that cylinder, and then through the rest of the cylinders from outermost to innermost.

By using this mapping, we can—at least in theory—convert a logical block number into an old-style disk address that consists of a cylinder number, a track number within that cylinder, and a sector number within that track. In practice,

it is difficult to perform this translation, for two reasons. First, most disks have some defective sectors, but the mapping hides this by substituting spare sectors from elsewhere on the disk. Second, the number of sectors per track is not a constant on some drives.

Let's look more closely at the second reason. On media that use **constant linear velocity (CLV)**, the density of bits per track is uniform. The farther a track is from the center of the disk, the greater its length, so the more sectors it can hold. As we move from outer zones to inner zones, the number of sectors per track decreases. Tracks in the outermost zone typically hold 40 percent more sectors than do tracks in the innermost zone. The drive increases its rotation speed as the head moves from the outer to the inner tracks to keep the same rate of data moving under the head. This method is used in CD-ROM and DVD-ROM drives. Alternatively, the disk rotation speed can stay constant; in this case, the density of bits decreases from inner tracks to outer tracks to keep the data rate constant. This method is used in hard disks and is known as **constant angular velocity (CAV)**.

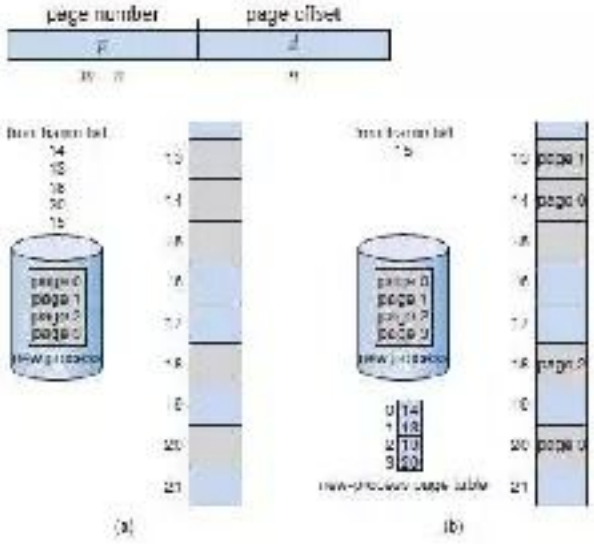
The number of sectors per track has been increasing as disk technology improves, and the outer zone of a disk usually has several hundred sectors per track. Similarly, the number of cylinders per disk has been increasing; large disks have tens of thousands of cylinders.



Paging model of logical and physical-memory

- The page-size (like the frame size) is defined by the hardware (Figure 5.9).
- If the size of the logical-address space is 2m, and a page-size is 2n addressing-units (bytes or words)

then the high-order *m-n* bits of a logical-address designate the page-number, and the *n* low-order bits
designate the page-offset.



Free frames (a) before allocation and (b) after allocation

Structure (Implementing) of the Page Table

1. Hierarchical Paging
2. Hashed Page-tables
3. Inverted Page-tables

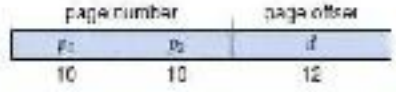
Hierarchical Paging

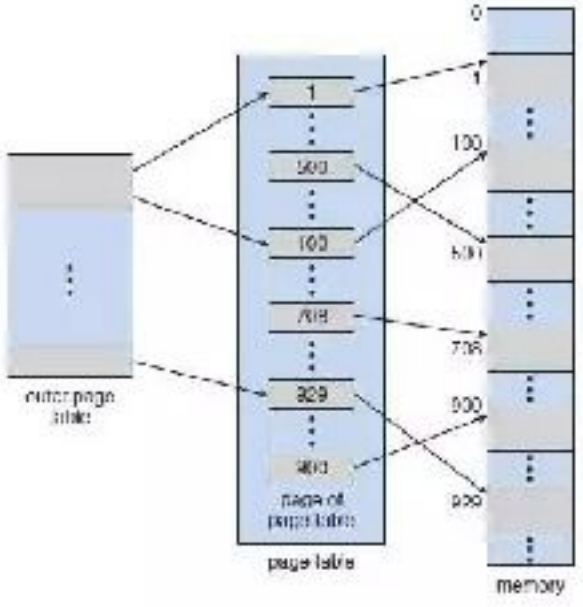
• Problem: Most computers support a large logical-address space (232 to 264). In these systems, the
page-table itself becomes excessively large.
Solution: Divide the page-table into smaller pieces.

Two Level Paging Algorithm

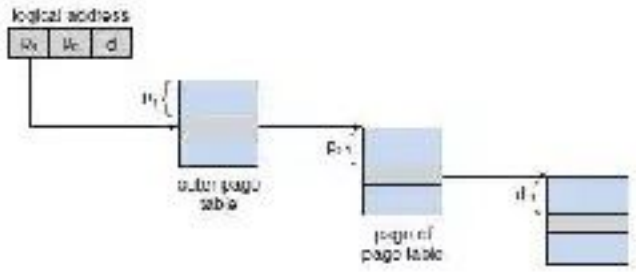
• The page-table itself is also paged (Figure 5.13).
• This is also known as a forward-mapped page-table because address translation works from the
outer page-table inwards.
• For example (Figure 5.14):

- Consider the system with a 32-bit logical-address space and a page-size of 4 KB.
- A logical-address is divided into
  → 20-bit page-number and
  → 12-bit page-offset.
- Since the page-table is paged, the page-number is further divided into
  → 10-bit page-number and
  → 10-bit page-offset.
- Thus, a logical-address is as follows:
-





A two-level page-table scheme



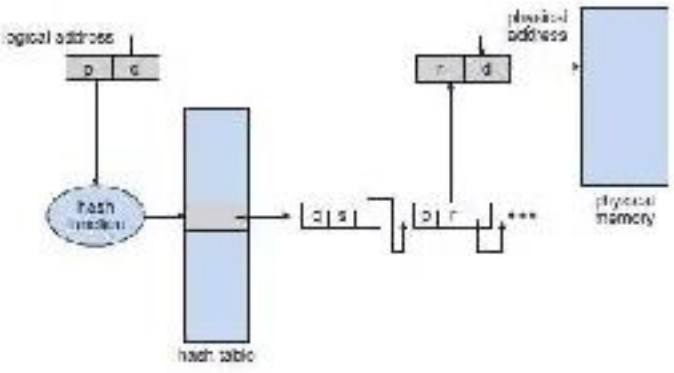
Address translation for a two-level 32-bit paging architecture

Hashed Page Tables

• This approach is used for handling address spaces larger than 32 bits.
• The hash-value is the virtual page-number.
• Each entry in the hash-table contains a linked-list of elements that hash to the same location (to
handle collisions).
• Each element consists of 3 fields:
1. Virtual page-number
2. Value of the mapped page-frame and
3. Pointer to the next element in the linked-list.
• The algorithm works as follows (Figure 5.15):
1. The virtual page-number is hashed into the hash-table.
2. The virtual page-number is compared with the first element in the linked-list.
3. If there is a match, the corresponding page-frame (field 2) is used to form the desired
physical-address.
4. If there is no match, subsequent entries in the linked-list are searched for a matching virtual
page-number.

Clustered Page Tables

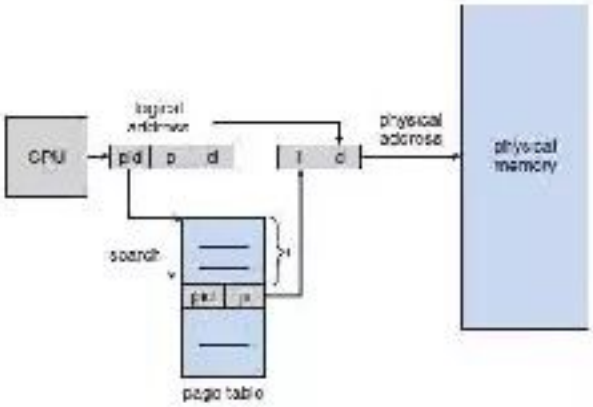
• These are similar to hashed page-tables except that each entry in the hash-table refers to several
pages rather than a single page.
• Advantages:
1. Favorable for 64-bit address spaces.
2. Useful for address spaces, where memory-references are noncontiguous and scattered
throughout the address space.



Hashed page-table

Inverted Page Tables

• Has one entry for each real page of memory.
• Each entry consists of
→ virtual-address of the page stored in that real memory-location and
→ information about the process that owns the page.



Inverted page-table

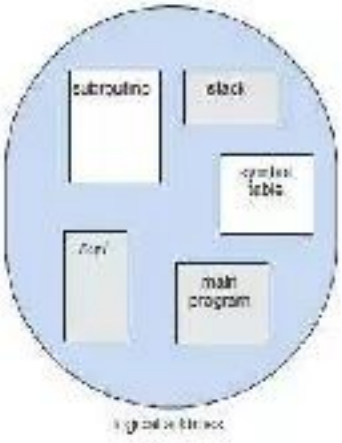
• Each virtual-address consists of a triplet (Figure 5.16):
<process-id, page-number, offset>.
• Each inverted page-table entry is a pair <process-id, page-number>
• The algorithm works as follows:
1. When a memory-reference occurs, part of the virtual-address, consisting of <process-id,
page-number>, is presented to the memory subsystem.
2. The inverted page-table is then searched for a match.
3. If a match is found, at entry i-then the physical-address <i, offset> is generated.
4. If no match is found, then an illegal address access has been attempted.
• Advantage:
1. Decreases memory needed to store each page-table
• Disadvantages:
1. Increases amount of time needed to search table when a page reference occurs.
2. Difficulty implementing shared-memory.

Segmentation

Basic Method of Segmentation

• This is a memory-management scheme that supports user-view of memory(Figure 5.17).
• A logical-address space is a collection of segments.
• Each segment has a name and a length.
• The addresses specify both
→ segment-name and
→ offset within the segment.

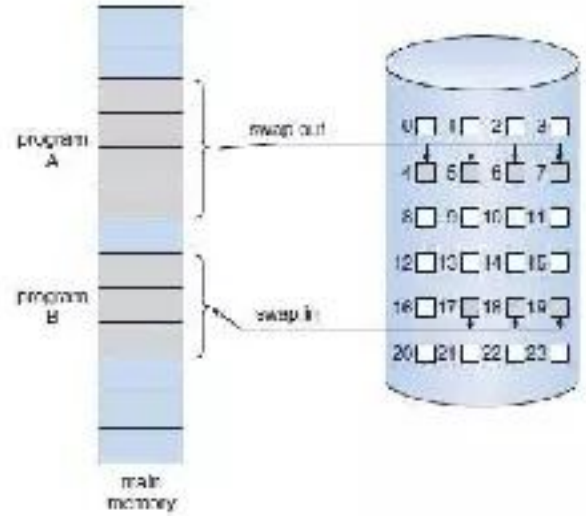
• Normally, the user-program is compiled, and the compiler automatically constructs segments
reflecting the input program.
For ex:
→ The code → Global variables
→ The heap, from which memory is allocated → The stacks used by each thread
→ The standard C library



Programmer's view of a program

Demand Paging

• A demand-paging system is similar to a paging-system with swapping (Figure 5.20).
• Processes reside in secondary-memory (usually a disk).
• When we want to execute a process, we swap it into memory.
• Instead of swapping in a whole process, **lazy swapper** brings only those necessary pages into
memory.



Transfer of a paged memory to contiguous disk space

Basic Concepts of Demand Paging

• Instead of swapping in a whole process, the pager brings only those necessary pages into memory.

• Advantages:

1. Avoids reading into memory-pages that will not be used,
2. Decreases the swap-time and
3. Decreases the amount of physical-memory needed.

• The *valid-invalid bit scheme* can be used to distinguish between

→ pages that are in memory and

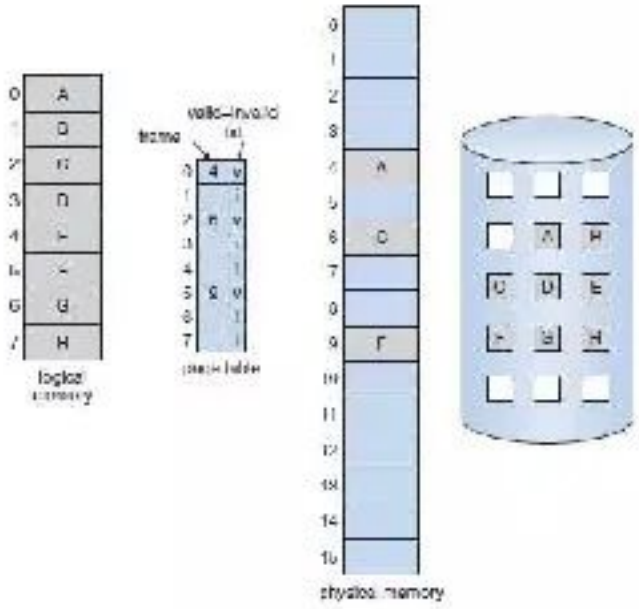
→ pages that are on the disk.

• If the bit is set to **valid**, the associated page is both legal and in memory.

• If the bit is set to **invalid**, the page either

→ is not valid (i.e. not in the logical-address space of the process) or

→ is valid but is currently on the disk



Page-table when some pages are not in main-memory

Pure demand paging: Never bring pages into memory until required.

• Some programs may access several new pages of memory with each instruction, causing multiple
page-faults and poor performance.

• Programs tend to have locality of reference, so this results in reasonable performance from demand
paging.

• Hardware support:

1. Page-table

• Mark an entry invalid through a valid-invalid bit.

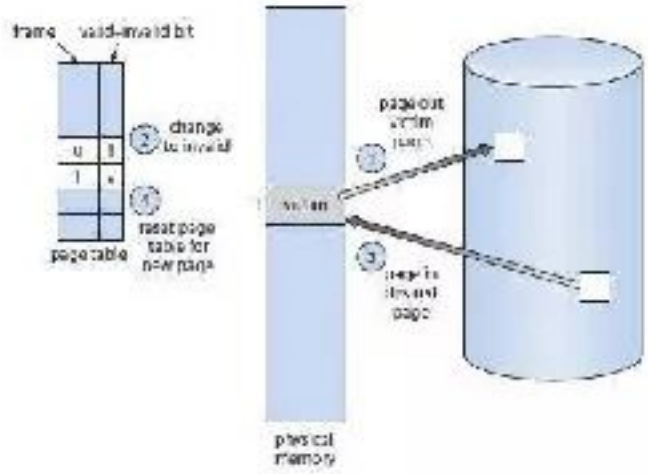
2. Secondary memory

• It holds pages that are not present in main-memory.

• It is usually a high-speed disk.
• It is known as the **swap device** (and the section of disk used for this purpose is known as
swap space).

Page Replacement
1. FIFO page replacement
2. Optimal page replacement
3. LRU page replacement (Least Recently Used)
4. LFU page replacement (Least Frequently Used)

Basic Page Replacement
• Basic page replacement approach:
→ If no frame is free, we find one that is not currently being used and free it (Figure 5.24).
• Page replacement takes the following steps:
1. Find the location of the desired page on the disk.
2. Find a free frame:
- If there is a free frame, use it.
- If there is no free frame, use a page-replacement algorithm to select a *victim-frame*.
- Write the victim-frame to the disk; change the page and frame-tables accordingly.
3. Read the desired page into the newly freed frame; change the page and frame-tables.
4. Restart the user-process.



Page replacement

• Problem: If no frames are free, 2 page transfers (1 out & 1 in) are required. This situation
→ doubles the page-fault service-time and
→ increases the EAT accordingly.
Solution: Use a *modify-bit (or dirty bit).*
• Each page or frame has a modify-bit associated with the hardware.
• The **modify-bit** for a page is set by the hardware whenever any word is written into the page
(indicating that the page has been modified).
• Working:
1. When we select a page for replacement, we examine it's modify-bit.
2. *If the modify-bit =1,* the page has been modified. So, we must write the page to the disk.
3. *If the modify-bit=0,* the page has *not* been modified. So, we need not write the page to the
disk, it is already there.
• Advantage:
1. Can reduce the time required to service a page-fault.
• We must solve 2 major problems to implement demand paging:
1. Develop a ***Frame-allocation algorithm:***
- If we have multiple processes in memory, we must decide how many frames to allocate to
each process.
2. Develop a ***Page-replacement algorithm:***
- We must select the frames that are to be replaced.

FIFO Page Replacement
• Each page is associated with the time when that page was brought into memory.
• When a page must be replaced, the oldest page is chosen.
• We use a **FIFO queue** to hold all pages in memory (Figure 5.25).
When a page must be replaced, we replace the page at the *head* of the queue
When a page is brought into memory, we insert it at the *tail* of the queue.