MODULE 3 :MEMORY MANAGEMENT

Main Memory Management Strategies

Every program to be executed has to be executed must be in memory. The instruction must be
fetched from memory before it is executed.

In multi-tasking OS memory management is complex, because as processes are swapped in and out
of the CPU, their code and data must be swapped in and out of memory.

Basic Hardware

Main memory, cache and CPU registers in the processors are the only storage spaces that CPU can
access directly.

The program and data must be bought into the memory from the disk, for the process to run. Each
process has a separate memory space and must access only this range of legal addresses. Protection of
memory is required to ensure correct operation. This prevention is provided by hardware implementation.

Two registers are used - a base register and a limit register. The base register holds the smallest
legal physical memory address; the limit register specifies the size of the range.

For example, The base register holds the smallest legal physical memory address; the limit register
specifies the size of the range. For example, if the base register holds 300040 and limit register is
120900, then the program can legally access all addresses from 300040 through 420940 (inclusive).

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

The base and limit registers can be loaded only by the operating system, which uses a special
privileged instruction. Since privileged instructions can be executed only in kernel mode only the
operating system can load the base and limit registers.

Figure: Hardware address protection with base and limit-registers

Address Binding

User programs typically refer to memory addresses with symbolic names. These symbolic names
must be mapped or bound to physical memory addresses.

Address binding of instructions to memory-addresses can happen at 3 different stages.

1. Compile Time - If it is known at compile time where a program will reside in physical memory, then absolute code can be generated by the compiler, containing actual physical addresses. However, if
the load address changes at some later time, then the program will have to be recompiled.
2. Load Time - If the location at which a program will be loaded is not known at compile time, then
the compiler must generate relocatable code, which references addresses relative to the start of the
program. If that starting address changes, then the program must be reloaded but not recompiled.
3. Execution Time - If a program can be moved around in memory during the course of its execution,
then binding must be delayed until execution time.

Figure: Multistep processing of a user program

Logical Versus Physical Address Space

The address generated by the CPU is a logical address, whereas the memory address where
programs are actually stored is a physical address.

The set of all logical addresses used by a program composes the logical address space, and the set of
all corresponding physical addresses composes the physical address space.

The run time mapping of logical to physical addresses is handled by the memorymanagement unit
(MMU).

One of the simplest is a modification of the base-register scheme.

The base register is termed a relocation register

The value in the relocation-register is added to every address generated by a user-process at the time
it is sent to memory.

The user-program deals with logical-addresses; it never sees the real physicaladdresses.

Figure: Dynamic relocation using a relocation-register

Dynamic Loading

This can be used to obtain better memory-space utilization.

A routine is not loaded until it is called.

This works as follows:

1. Initially, all routines are kept on disk in a relocatable-load format.
2. Firstly, the main-program is loaded into memory and is executed.
3. When a main-program calls the routine, the main-program first checks to see whether the routine has
been loaded.
4. If routine has been not yet loaded, the loader is called to load desired routine into memory.
5. Finally, control is passed to the newly loaded-routine.

Advantages:

1. An unused routine is never loaded.
2. Useful when large amounts of code are needed to handle infrequently occurring cases.
3. Although the total program-size may be large, the portion that is used (and hence loaded) may be
much smaller.
4. Does not require special support from the OS.

Dynamic Linking and Shared Libraries

With static linking library modules get fully included in executable modules, wasting both disk space and main memory usage, because every program that included a certain routine from the library
would have to have their own copy of that routine linked into their executable code.

With dynamic linking, however, only a stub is linked into the executable module, containing references to the actual library module linked in at run time.

The stub is a small piece of code used to locate the appropriate memory-resident library-routine.

This method saves disk space, because the library routines do not need to be fully included in the
executable modules, only the stubs.

An added benefit of dynamically linked libraries (DLLs, also known as shared libraries or shared
objects on UNIX systems) involves easy upgrades and updates.

Shared libraries

A library may be replaced by a new version, and all programs that reference the library will automatically use the new one.

Version info. is included in both program & library so that programs won't accidentally execute incompatible versions.

Figure: Swapping of two processes using a disk as a backing store

Example:

Assume that the user process is 10 MB in size and the backing store is a standard hard disk with a
transfer rate of 40 MB per second.

The actual transfer of the 10-MB process to or from main memory takes 10000 KB/40000 KB per
second = 1/4 second = 250 milliseconds.

Assuming that no head seeks are necessary, and assuming an average latency of 8 milliseconds, the
swap time is 258 milliseconds. Since we must both swap out and swap in, the total swap time is about
516 milliseconds.

Contiguous Memory Allocation

The main memory must accommodate both the operating system and the various user processes.
Therefore we need to allocate the parts of the main memory in the most efficient way possible.

Memory is usually divided into 2 partitions: One for the resident OS. One for the user processes.

Each process is contained in a single contiguous section of memory.

1. Memory Mapping and Protection

Memory-protection means protecting OS from user-process and protecting userprocesses from one
another.

Memory-protection is done using

o Relocation-register: contains the value of the smallest physical-address.

o Limit-register: contains the range of logical-addresses.

Each logical-address must be less than the limit-register.

The MMU maps the logical-address dynamically by adding the value in the relocationregister. This
mapped-address is sent to memory

When the CPU scheduler selects a process for execution, the dispatcher loads the relocation and
limit-registers with the correct values.

Because every address generated by the CPU is checked against these registers, we can protect the
OS from the running-process.

The relocation-register scheme provides an effective way to allow the OS size to change dynamically.

Transient OS code: Code that comes & goes as needed to save memory-space and overhead for
unnecessary swapping.

Figure: Hardware support for relocation and limit-registers

2. Memory Allocation

Two types of memory partitioning are:

1. Fixed-sized partitioning
2. Variable-sized partitioning

1. Fixed-sized Partitioning

The memory is divided into fixed-sized partitions.

Each partition may contain exactly one process.

The degree of multi programming is bound by the number of partitions.

When a partition is free, a process is selected from the input queue and loaded into the free partition.

When the process terminates, the partition becomes available for another process.

2. Variable-sized Partitioning

The OS keeps a table indicating which parts of memory are available and which parts are occupied.

A hole is a block of available memory. Normally, memory contains a set of holes of various sizes.

Initially, all memory is available for user-processes and considered one large hole.

When a process arrives, the process is allocated memory from a large hole.

If we find the hole, we allocate only as much memory as is needed and keep the remaining memory
available to satisfy future requests.

Three strategies used to select a free hole from the set of available holes:

1. First Fit: Allocate the first hole that is big enough. Searching can start either at the beginning of the
set of holes or at the location where the previous first-fit search ended.
2. Best Fit: Allocate the smallest hole that is big enough. We must search the entire list, unless the list
is ordered by size. This strategy produces the smallest leftover hole.
3. Worst Fit: Allocate the largest hole. Again, we must search the entire list, unless it is sorted by size.
This strategy produces the largest leftover hole.

First-fit and best fit are better than worst fit in terms of decreasing time and storage utilization.

3. Fragmentation

Two types of memory fragmentation:

1. Internal fragmentation
2. External fragmentation

1. Internal Fragmentation

The general approach is to break the physical-memory into fixed-sized blocks and allocate memory
in units based on block size.

The allocated-memory to a process may be slightly larger than the requested-memory.

The difference between requested-memory and allocated-memory is called internal fragmentation
i.e. Unused memory that is internal to a partition.

2. External Fragmentation

External fragmentation occurs when there is enough total memory-space to satisfy a request but the
available-spaces are not contiguous. (i.e. storage is fragmented into a large number of small holes).

Both the first-fit and best-fit strategies for memory-allocation suffer from external fragmentation.

Statistical analysis of first-fit reveals that given N allocated blocks, another 0.5 N blocks will be lost
to fragmentation. This property is known as the 50-percent rule.
Two solutions to external fragmentation:
Compaction: The goal is to shuffle the memory-contents to place all free memory together in one
large hole. Compaction is possible only if relocation is dynamic and done at execution-time.Permit the
logical-address space of the processes to be non-contiguous. This allows a process to be allocated
physical-memory wherever such memory is available. Two techniques achieve this solution: 1) Paging
and 2) Segmentation.

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
The basic method for implementing paging involves breaking physical memory into fixed-sized blocks called frames and breaking logical memory into blocks of the same size 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.
The hardware support for paging is illustrated in Figure 1.

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

The page table maps the page number to a frame number, to yield a physical address which also has two parts: The frame number and the offset within that frame.

The number of bits in the frame number determines how many frames the system can address, and the number of bits in the offset determines the size of each frame. The paging model of memory is shown in Figure 2.

Figure 2: Paging model of logical and physical memory.

The page size (like the frame size) is defined by the hardware.

The size of a page is typically a power of 2, varying between 512 bytes and 16 MB per page, depending on the computer architecture.

The selection of a power of 2 as a page size makes the translation of a logical address into a page number and page offset.

If the size of 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.

Thus, the logical address is as follows:

When a process requests memory (e.g. when its code is loaded in from disk), free frames are allocated from a free-frame list, and inserted into that process's page table.

Processes are blocked from accessing anyone else's memory because all of their memory requests are mapped through their page table. There is no way for them to generate an address that maps into any other process's memory space.

The operating system must keep track of each individual process's page table, updating it whenever the process's pages get moved in and out of memory, and applying the correct page table when processing system calls for a particular process. This all increases the overhead involved when swapping processes in and out of the CPU.

Figure: Free frames (a) before allocation and (b) after allocation.

Hardware Support

Translation Look aside Buffer

A special, small, fast lookup hardware cache, called a translation look-aside buffer (TLB).

Each entry in the TLB consists of two parts: a key (or tag) and a value.

When the associative memory is presented with an item, the item is compared with all keys simultaneously. If the item is found, the corresponding value field is returned. The search is fast; the
hardware, however, is expensive. Typically, the number of entries in a TLB is small, often numbering
between 64 and 1,024.

The TLB contains only a few of the page-table entries.

Working:

When a logical-address is generated by the CPU, its page-number is presented to the TLB.

If the page-number is found (TLB hit), its frame-number is immediately available and used to access memory

If page-number is not in TLB (TLB miss), a memory-reference to page table must be made. The
obtained frame-number can be used to access memory (Figure 1)

Figure 1: Paging hardware with TLB

In addition, we add the page-number and frame-number to the TLB, so that they will be found quickly on the next reference.

If the TLB is already full of entries, the OS must select one for replacement.

Percentage of times that a particular page-number is found in the TLB is called hit ratio.

Advantage: Search operation is fast.

Disadvantage: Hardware is expensive.

Some TLBs have wired down entries that can't be removed.

Some TLBs store ASID (address-space identifier) in each entry of the TLB that uniquely identify
each process and provide address space protection for that process.

Protection

Memory-protection is achieved by protection-bits for each frame.

The protection-bits are kept in the page-table.

One protection-bit can define a page to be read-write or read-only.

Every reference to memory goes through the page-table to find the correct framenumber.

Firstly, the physical-address is computed. At the same time, the protection-bit is checked to verify
that no writes are being made to a read-only page.

An attempt to write to a read-only page causes a hardware-trap to the OS (or memory protection
violation).

Valid Invalid Bit

This bit is attached to each entry in the page-table.
Valid bit: “valid” indicates that the associated page is in the process’ logical address space, and is
thus a legal page
Invalid bit: “invalid” indicates that the page is not in the process’ logical address space Illegal addresses are trapped by use of valid-invalid bit. The OS sets this bit for each page to allow or disallow
access to the page.

Figure: Valid (v) or invalid (i) bit in a page-table

Shared Pages
An advantage of paging is the possibility of sharing common code.
Re-entrant code (Pure Code) is non-self-modifying code, it never changes during execution.
Two or more processes can execute the same code at the same time.
Each process has its own copy of registers and data-storage to hold the data for the process's execution.
The data for 2 different processes will be different.
Only one copy of the editor need be kept in physical-memory (Figure 5.12).
Each user's page-table maps onto the same physical copy of the editor, but data pages are mapped
onto different frames.
Disadvantage:
Systems that use inverted page-tables have difficulty implementing shared-memory.

Figure: Sharing of code in a paging environment

Structure of the Page Table
The most common techniques for structuring the page table:
1. Hierarchical Paging
2. Hashed Page-tables
3. Inverted Page-tables
1. 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.
This is also known as a forward-mapped page-table because address translation works from the
outer page-table inwards.

Figure: A two-level page-table scheme

For example:
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:

where p1 is an index into the outer page table, and p2 is the displacement within the page of the
inner page table
The address-translation method for this architecture is shown in below figure. Because address translation works from the outer page table inward, this scheme is also known as a forwardmapped
page table.

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

2. 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:
The virtual page-number is hashed into the hash-table.
The virtual page-number is compared with the first element in the linked-list.
If there is a match, the corresponding page-frame (field 2) is used to form the desired physical-
address.
If there is no match, subsequent entries in the linked-list are searched for a matching virtual page-

number.

Figure: Hashed page-table

3. 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.

Each virtual-address consists of a triplet <process-id, page-number, offset>.

Each inverted page-table entry is a pair <process-id, page-number>

Figure: Inverted page-table

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 1).

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

Figure: Programmer's view of a program

Hardware support for Segmentation

Segment-table maps 2 dimensional user-defined addresses into one-dimensional physical addresses.

In the segment-table, each entry has following 2 fields:

1. Segment-base contains starting physical-address where the segment resides in memory.
2. Segment-limit specifies the length of the segment (Figure 2).

A logical-address consists of 2 parts:

1. Segment-number(s) is used as an index to the segment-table
2. Offset(d) must be between 0 and the segment-limit.

If offset is not between 0 & segment-limit, then we trap to the OS(logical-addressing attempt beyond end of segment).

If offset is legal, then it is added to the segment-base to produce the physical-memory address.

VIRTUAL MEMORYMANAGEMENT

Virtual memory is a technique that allows for the execution of partially loaded process.

Advantages:

A program will not be limited by the amount of physical memory that is available user can able to
write in to large virtual space.

Since each program takes less amount of physical memory, more than one program could be run at
the same time which can increase the throughput and CPU utilization.

Less i/o operation is needed to swap or load user program in to memory. So each user program
could run faster.

Fig: Virtual memory that is larger than physical memory.

Virtual memory is the separation of users logical memory from physical memory. This separation
allows an extremely large virtual memory to be provided when these is less physical memory.

Separating logical memory from physical memory also allows files and memory to be shared by
several different processes through page sharing.

Fig: Virtual memory that is larger than physical memory.

Virtual memory is the separation of users logical memory from physical memory. This separation
allows an extremely large virtual memory to be provided when these is less physical memory.

Separating logical memory from physical memory also allows files and memory to be shared by

several different processes through page sharing.

Fig: Shared Library using Virtual Memory

Virtual memory is implemented using Demand Paging.

Virtual address space: Every process has a virtual address space i.e used as the stack or heap grows
in size.

Fig: Virtual address space

DEMAND PAGING

A demand paging is similar to paging system with swapping when we want to execute a process we
swap the process the in to memory otherwise it will not be loaded in to memory.

A swapper manipulates the entire processes, where as a pager manipulates individual pages of the
process.

Bring a page into memory only when it is needed

Less I/O needed

Less memory needed

Faster response

More users

Page is needed ⇒ reference to it

invalid reference ⇒abort

not-in-memory ⇒ bring to memory

Lazy swapper– never swaps a page into memory unless page will be needed Swapper that deals
with pages is a pag

Fig: Transfer of a paged memory into continuous disk space

Basic concept: Instead of swapping the whole process the pager swaps only the necessary pages in to memory. Thus it avoids reading unused pages and decreases the swap time and amount of
physical memory needed.

The valid-invalid bit scheme can be used to distinguish between the pages that are on the disk and
that are in memory.
With each page table entry a valid–invalid bit is associated (v ⇒ in-memory,
i⇒not-in-memory)
Initially valid–invalid bit is set to ion all entries
Example of a page table snapshot:
During address translation, if valid–invalid bit in page table entry is I ⇒ page fault.
If the bit is valid then the page is both legal and is in memory.
If the bit is invalid then either page is not valid or is valid but is currently on the disk. Markinga page as invalid will have no effect if the processes never access to that page. Suppose if it access the
page which is marked invalid, causes a page fault trap. This may result in failure of OS to bring the
desired page in to memory.

Fig: Page Table when some pages are not in main memory

Page Fault
If a page is needed that was not originally loaded up, then a page fault trap is generated.
Steps in Handling a Page Fault
1. The memory address requested is first checked, to make sure it was a valid memory request.
2. If the reference is to an invalid page, the process is terminated. Otherwise, if the page is not present
in memory, it must be paged in.
3. A free frame is located, possibly from a free-frame list.
4. A disk operation is scheduled to bring in the necessary page from disk.
5. After the page is loaded to memory, the process's page table is updated with the new frame number,
and the invalid bit is changed to indicate that this is now a valid page reference.
6. The instruction that caused the page fault must now be restarted from the beginning.

Fig: steps in handling page fault
Pure Demand Paging: Never bring a page into main memory until it is required.
We can start executing a process without loading any of its pages into main memory.
Page fault occurs for the non memory resident pages.
After the page is brought into memory, process continues to execute.
Again page fault occurs for the next page.

Hardware support: For demand paging the same hardware is required as paging and swapping.
1. Page table:-Has the ability to mark an entry invalid through valid-invalid bit.
2. Secondary memory:-This holds the pages that are not present in main memory.
Performance of Demand Paging: Demand paging can have significant effect on the performance of
the computer system.
 Let P be the probability of the page fault (0<=P<=1)
 Effective access time = (1-P) * ma + P * page fault.
 Where P = page fault and ma = memory access time.
 Effective access time is directly proportional to page fault rate. It is important to keep page fault rate
low in demand paging.
Demand Paging Example
 Memory access time = 200 nanoseconds
 Average page-fault service time = 8milliseconds
 EAT = (1 – p) x 200 + p (8milliseconds) = (1 – p x 200 + p x 8,000,000 = 200 + p x 7,999,800
 If one access out of 1,000 causes a page fault, then EAT = 8.2 microseconds. This is a slowdown by a factor of 40.

COPY-ON-WRITE
 Technique initially allows the parent and the child to share the same pages. These pages are marked
as copy on- write pages i.e., if either process writes to a shared page, a copy of shared page is created.
 Eg:-If a child process try to modify a page containing portions of the stack; the OS recognizes them
as a copy-on-write page and create a copy of this page and maps it on to the address space of the child
process. So the child process will modify its copied page and not the page belonging to parent. The
new pages are obtained from the pool of free pages.
 The previous contents of pages are erased before getting them into main memory. This is called
Zero – on fill demand.
a) Before Process 1 modifies pageC

b)
b) After process 1 modifies page C

PAGEREPLACEMENT

Page replacement policy deals with the solution of pages in memory to be replaced by a new page
that must be brought in. When a user process is executing a page fault occurs.

The hardware traps to the operating system, which checks the internal table to see that this is a page
fault and not an illegal memory access.

The operating system determines where the derived page is residing on the disk, and this finds that
there are no free frames on the list of free frames.

When all the frames are in main memory, it is necessary to bring a new page to satisfy the page fault,
replacement policy is concerned with selecting a page currently in memory to be replaced.

The page i,e to be removed should be the page i,e least likely to be referenced in future.

Fig: Page Replacement

Working of Page Replacement Algorithm

1. Find the location of derived page on the disk.
2. Find a free frame x If there is a free frame, use it. x Otherwise, use a replacement algorithm to select
a victim.

Write the victim page to the disk.

Change the page and frame tables accordingly.

3. Read the desired page into the free frame; change the page and frame tables.
4. Restart the user process.

Victim Page

The page that is supported out of physical memory is called victim page. If no frames are free,
the two page transforms come (out and one in) are read. This will see the effective access time.

Each page or frame may have a dirty (modify) bit associated with the hardware. The modify bit for
a page is set by the hardware whenever any word or byte in the page is written into, indicating that the
page has been modified.

When we select the page for replacement, we check its modify bit. If the bit is set, then the page is
modified since it was read from the disk.

If the bit was not set, the page has not been modified since it was read into memory. Therefore, if
the copy of the page has not been modified we can avoid writing the memory page to the disk, if it is
already there. Sum pages cannot be modified.

Modify bit/ Dirty bit :

Each page/frame has a modify bit associated with it.

If the page is not modified (read-only) then one can discard such page without writing it onto the
disk. Modify bit of such page is set to0.
Modify bit is set to 1, if the page has been modified. Such pages must be written to the disk.
Modify bit is used to reduce overhead of page transfers – only modified pages are written to disk

PAGE REPLACEMENT ALGORITHMS
Want lowest page-fault rate
Evaluate algorithm by running it on a particular string of memory references (reference string) and
computing the number of page faults on that string
In all our examples, the reference string is
1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5
FIFO Algorithm:
This is the simplest page replacement algorithm. A FIFO replacement algorithm associates each
page the time when that page was brought into memory.
When a Page is to be replaced the oldest one is selected.
We replace the queue at the head of the queue. When a page is brought into memory, we insert it at
the tail of the queue.
In the following example, a reference string is given and there are 3 free frames. There are 20 page
requests, which results in 15 page faults
Belady’s Anomaly
For some page replacement algorithm, the page fault may increase as the number of allocated
frames increases. FIFO replacement algorithm may face this problem. more frames ⇒ more page
faults
Example: Consider the following references string with frames initially empty.
The first three references (7,0,1) cases page faults and are brought into the empty frames.
The next references 2 replaces page 7 because the page 7 was brought in first. x Since 0 is the next
references and 0 is already in memory e has no page faults.
The next references 3 results in page 0 being replaced so that the next references to 0 causer page
fault. This will continue till the end of string. There are 15 faults all together.

Optimal Algorithm

Optimal page replacement algorithm is mainly to solve the problem of Belady's Anomaly.
Optimal page replacement algorithm has the lowest page fault rate of all algorithms.
An optimal page replacement algorithm exists and has been called OPT. The working is simple
"Replace the page that will not be used for the longest period of time" Example: consider the following
reference string
The first three references cause faults that fill the three empty frames.
The references to page 2 replaces page 7, because 7 will not be used until reference 18. x
The page 0 will be used at 5 and page 1 at 14.
With only 9 page faults, optimal replacement is much better than a FIFO, which had 15 faults. This
algorithm is difficult t implement because it requires future knowledge of reference strings.
Replace page that will not be used for longest period of time
Least Recently Used (LRU) Algorithm
The LRU (Least Recently Used) algorithm, predicts that the page that has not been used in the
longest time is the one that will not be used again in the near future.
Some view LRU as analogous to OPT, but here we look backwards in time instead of forwards.

The main problem to how to implement LRU is the LRU requires additional h/w assistance.
Two implementation are possible:
1. Counters: In this we associate each page table entry a time -of -use field, and add to the cpu a
logical clock or counter. The clock is incremented for each memory reference. When 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. In this way we have the time of last reference to each page we replace the page with
smallest time value. The time must also be maintained when page tables are changed.
2. Stack: Another approach to implement LRU replacement is to keep a stack of page numbers when a
page is referenced it is removed from the stack and put on to the top of stack. In this way the top of
stack is always the most recently used page and the bottom in least recently used page. Since the entries
are removed from the stack it is best implement by a doubly linked list. With a head and tail pointer.

Note: Neither optimal replacement nor LRU replacement suffers from Belady's Anamoly. These are
called stack algorithms.
LRU-Approximation Page Replacement
Many systems offer some degree of hardware support, enough to approximate LRU.
In particular, many systems provide a reference bit for every entry in a page table, which is set
anytime that page is accessed. Initially all bits are set to zero, and they can also all be cleared at any
time. One bit distinguishes pages that have been accessed since the last clear from those that have not
been accessed.
Additional-Reference-Bits Algorithm
An 8-bit byte (reference bit) is stored for each page in a table in memory.
At regular intervals (say, every 100 milliseconds), a timer interrupt transfers control to the operating
system. The operating system shifts the reference bit for each page into the high-order bit of its 8-bit
byte, shifting the other bits right by 1 bit and discarding the loworder bit.
These 8-bit shift registers contain the history of page use for the last eight time periods.
If the shift register contains 00000000, then the page has not been used for eight time periods.
A page with a history register value of 11000100 has been used more recently than one with a value
of 01110111.
Second- chance (clock) page replacement algorithm
The second chance algorithm is a FIFO replacement algorithm, except the reference bit is used to
give pages a second chance at staying in the page table.
When a page must be replaced, the page table is scanned in a FIFO (circular queue) manner.
If a page is found with its reference bit as '0', then that page is selected as the next victim.
If the reference bitvalueis'1', then the page is given a second chance and its reference bitvalue is
cleared (assigned as'0').
Thus, a page that is given a second chance will not be replaced until all other pages have been
replaced (or given second chances). In addition, if a page is used often, then it sets its reference bit
again.
This algorithm is also known as the clock algorithm.
Enhanced Second-Chance Algorithm
The enhanced second chance algorithm looks at the reference bit and the modify bit (dirty
bit) as an ordered page, and classifies pages into one of four classes:
1. (0, 0) - Neither recently used nor modified.

2. (0, 1) - Not recently used, but modified.
3. (1, 0) - Recently used, but clean.
4. (1, 1) - Recently used and modified.

This algorithm searches the page table in a circular fashion, looking for the first page it can find in
the lowest numbered category. i.e. it first makes a pass looking for a (0, 0), and then if it can't find one,
it makes another pass looking for a(0,1),etc.

The main difference between this algorithm and the previous one is the preference for replacing
clean pages if possible.

Count Based Page Replacement

There is many other algorithms that can be used for page replacement, we can keep a counter of the
number of references that has made to a page.

a) LFU (least frequently used):

This causes the page with the smallest count to be replaced. The reason for this selection is that actively
used page should have a large reference count.

This algorithm suffers from the situation in which a page is used heavily during the initial phase of a
process but never used again. Since it was used heavily, it has a large count and remains in memory
even though it is no longer needed.

b)MFU Algorithm:

based on the argument that the page with the smallest count was probably just brought in and has yet to
be used

ALLOCATION OF FRAMES

The absolute minimum number of frames that a process must be allocated is dependent on system
architecture.

The maximum number is defined by the amount of available physical memory.

Allocation Algorithms

After loading of OS, there are two ways in which the allocation of frames can be done to the processes.

Equal Allocation- If there are m frames available and n processes to share them, each process
gets m / n frames, and the left over’s are kept in a free-frame buffer pool.

Proportional Allocation - Allocate the frames proportionally depending on the size of the
process. If the size of process i is Si, and S is the sum of size of all processes in the system, then the
allocation for process Pi is ai= m * Si/ S. where m is the free frames available in the system.

Consider a system with a 1KB frame size. If a small student process of 10 KB and an interactive
database of 127 KB are the only two processes running in a system with 62 free frames.
with proportional allocation, we would split 62 frames between two processes, as follows
m=62, S = (10+127)=137
Allocation for process 1 = 62 X 10/137 ~ 4 Allocation for process 2 = 62 X
127/137 ~57
Thus allocates 4 frames and 57 frames to student process and database respectively.
Variations on proportional allocation could consider priority of process rather than just their size.

Global versus Local Allocation

Page replacement can occur both at local or global level.
With local replacement, the number of pages allocated to a process is fixed, and page replacement
occurs only amongst the pages allocated to this process.
With global replacement, any page may be a potential victim, whether it currently belongs to the
process seeking a free frame or not.
Local page replacement allows processes to better control their own page fault rates, and leads to
more consistent performance of a given process over different system load levels.
Global page replacement is over all more efficient, and is the more commonly used approach.

Non-Uniform Memory Access (New)

Usually the time required to access all memory in a system is equivalent.
This may not be the case in multiple-processor systems, especially where each CPU is physically
located on a separate circuit board which also holds some portion of the overall system memory.
In such systems, CPU s can access memory that is physically located on the same board much faster
than the memory on the other boards.
The basic solution is akin to processor affinity - At the same time that we try to schedule processes
on the same CPU to minimize cache misses, we also try to allocate memory for those processes on the
same boards, to minimize access times.

THRASHING

If the number of frames allocated to a low-priority process falls below the minimum number
required by the computer architecture then we suspend the process execution.
A process is thrashing if it is spending more time in paging than executing.
If the processes do not have enough number of frames, it will quickly page fault.
During this it must replace some page that is not currently in use. Consequently it quickly faults again
and again.

The process continues to fault, replacing pages for which it then faults and brings back. This high
paging activity is called thrashing. The phenomenon of excessively moving pages back and forth b/w
memory and secondary has been called thrashing.
Cause of Thrashing
Thrashing results in severe performance problem.
The operating system monitors the cpu utilization is low. We increase the degree of multi programming by introducing new process to the system.
A global page replacement algorithm replaces pages with no regards to the process to which they
belong.

The figure shows the thrashing
As the degree of multi programming increases, more slowly until a maximum is reached. If the
degree of multi programming is increased further thrashing sets in and the cpu utilization drops sharply.
At this point, to increases CPU utilization and stop thrashing, we must increase degree
of multiprogramming.
we can limit the effect of thrashing by using a local replacement algorithm. To prevent thrashing,
we must provide a process as many frames as it needs.
Locality of Reference:
As the process executes it moves from locality to locality.
A locality is a set of pages that are actively used.
A program may consist of several different localities, which may overlap.
Locality is caused by loops in code that find to reference arrays and other data structures by indices.
The ordered list of page number accessed by a program is called reference string.
Locality is of two types :
1. spatial locality 2. temporal locality
Working set model
Working set model algorithm uses the current memory requirements to determine the number of
page frames to allocate to the process, an informal definition is “the collection of pages that a process is
working with and which must be resident if the process to avoid thrashing”. The idea is to use the
recent needs of a process to predict its future reader.
The working set is an approximation of programs locality. Ex: given a sequence of memory

reference, if the working set window size to memory references, then working set at time t1 is{1,2,5,6,7} and at t2 is changed to {3,4}

At any given time, all pages referenced by a process in its last 4 seconds of execution are considered
to compromise its working set.

A process will never execute until its working set is resident in main memory. Pages outside the
working set can be discarded at any movement.

Working sets are not enough and we must also introduce balance set. If the sum of the working
sets of all the run able process is greater than the size of memory the refuse some process for a while.

Divide the run able process into two groups, active and inactive. The collection of active set is
called the balance set. When a process is made active its working set is loaded.

Some algorithm must be provided for moving process into and out of the balance set. As a working
set is changed, corresponding change is made to the balance set.

Working set presents thrashing by keeping the degree of multi programming as high as possible.
Thus if optimizes the CPU utilization. The main disadvantage of this is keeping track of the working
set.

Page-Fault Frequency

When page- fault rate is too high, the process needs more frames and when it is too low, the process
may have too many frames.

The upper and lower bounds can be established on the page-fault rate. If the actual page-fault rate
exceeds the upper limit, allocate the process another frame or suspend the process.

If the page-fault rate falls below the lower limit, remove a frame from the process. Thus, we can
directly measure and control the page-fault rate to prevent thrashing.