Contents

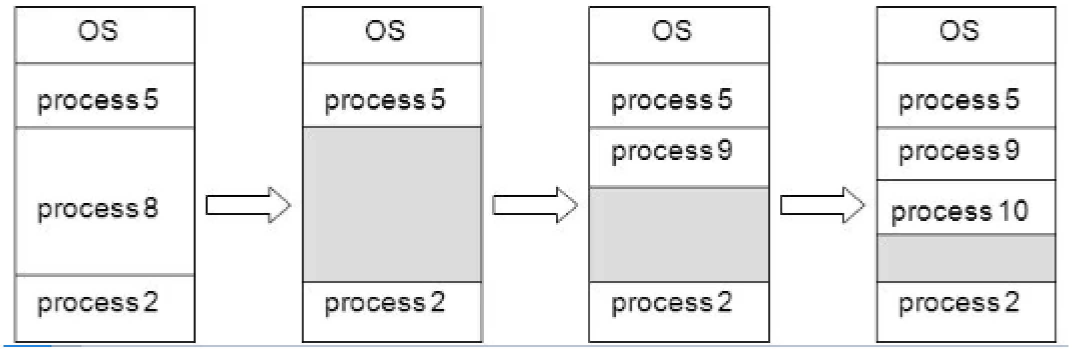
Contiguous Memory Allocation

Contiguous Memory Allocation is an early method of memory management whereby, as new processes are chosen from the READY queue, the OS has to choose a partition size and physical location for this partition. All pages for a process are allocated together in one chunk. The process is allocated memory from a hole large enough to accommodate it. The OS maintains information about the allocated partitions and the free partitions (hole).

The resident operating system is usually held in low memory with the interrupt vector.

The user processes are then held in high memory

Each process contained in single contiguous section of memory



When a process exits, **merging** can be done if there is an existing hole above and/or below it.

Swapping as a Memory Management Technique

Swapping is a useful technique that enables a computer to execute programs and manipulate data files larger than main memory (RAM).

The OS copies as much data as possible into main memory, and leaves the rest on the disk: A process can be swapped temporarily out of memory to a backing store, and then brought back into memory for continued execution

When the operating system needs data from the disk, it exchanges a portion of data (called a ***page*** or ***segment***) in main memory with a portion of data on the disk. This is also known as paging.

Paging is more flexible as only pages of a process are moved. **Swapping** is less flexible as it moves entire process back and forth between main memory and back store. Compared to **paging swapping** allows less processes to reside in main memory.

Performance is usually affected by swapping processes but it helps in running multiple and big processes in parallel. **Swapping** is also known as a technique for **memory compaction**.

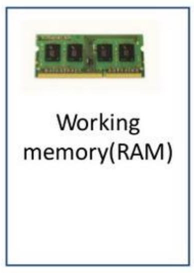


Figure 2 - Powered OFF

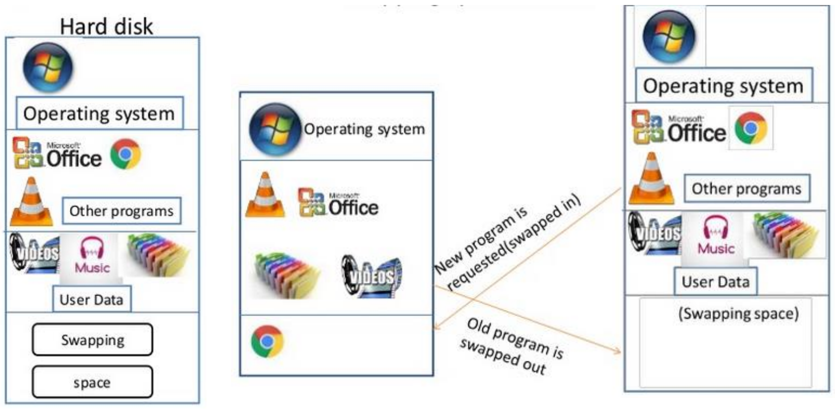


Figure 1 - Swapping in/out of RAM

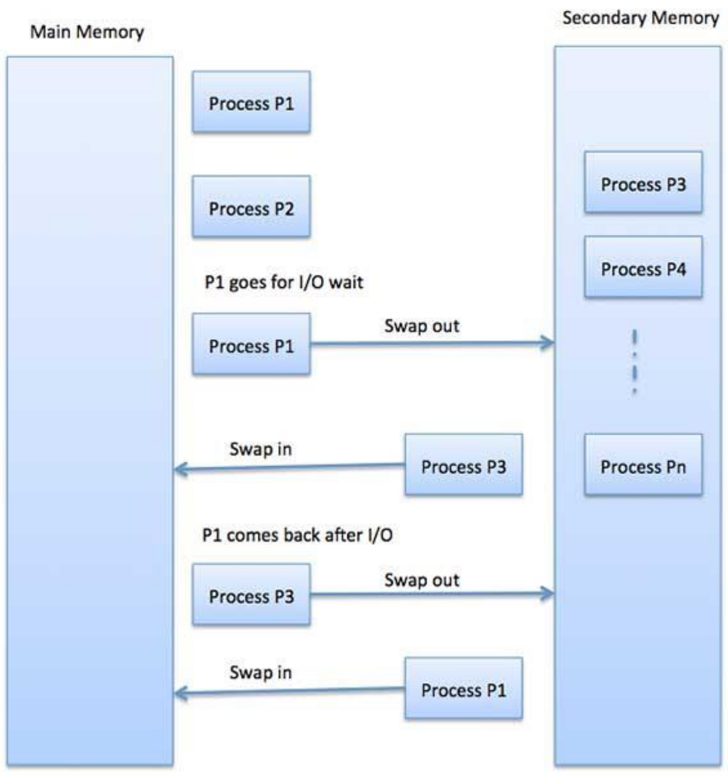


Figure 3 - Process Swapping

The total time taken by swapping process includes the time it takes to move the entire process to a secondary disk and then to copy the process back to memory, as well as the time the process takes to regain main memory.

Roll out, roll in Swapping Technique

This is a swapping variant used for priority-based scheduling algorithms; lower-priority process is swapped out so higher-priority process can be loaded and executed

Swapping on Mobile Systems

Swapping is not typically supported on mobile systems:

They are Flash memory based with a small amount of space

The throughput is low between flash memory and CPU on mobile platforms. Instead they use other methods to free memory if it's low.

iOS asks apps to voluntarily relinquish allocated memory. Here, read-only data is thrown out and reloaded from flash if needed. If there's a failure to free the memory, this can result in termination

Android terminates apps if there is a low amount of free memory, but first writes the application state to flash for fast restart

Both mobile OSs support another technique called ***paging***

Paging

Paging is a memory management technique in which process address space is broken into blocks of the same size called **pages** (size is power of 2, between 512 (2^9) bytes and 8192 (2^{13}) bytes). The size of the process is measured in the number of pages. The physical address space of a process to be non-contiguous.

For each process, all the machine code is in one lump, all the data is in one lump etc.

The concept of **pages** attempts to break the lumps into small physical memory units. The memory can be physically non-contiguous so it can be allocated from wherever available.

A logical address in a paged memory management system begins as a single integer value relative to the starting point of the program.

A logical address is often written as <page, offset>, such as <2, 518> and each logical address of a program is identical to its relative address. The page number is the number of times the page size divides the address and the remainder is the offset.

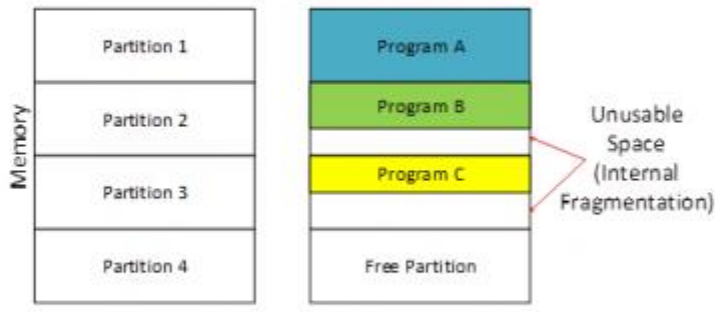
> i.e. a logical address of 2566, with a page size of 1024, corresponds to the 518th byte of page 2 of the process.

Both internal fragmentation and external fragmentation are phenomena where memory is wasted

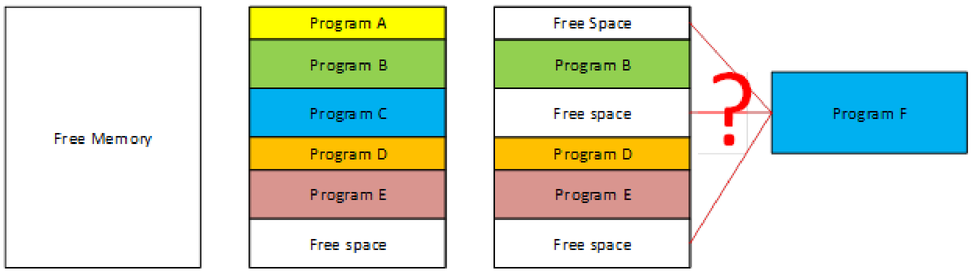
Fragmentation

Fragmentation describes when we have a lot of useless little holes. The system may get to a point when the available memory is enough to start a process, but it is in the form of holes too small to load the process.

i. Internal Fragmentation occurs when fixed size memory allocation technique is used. Memory allocation is based on **fixed-size partitions** where after a small size application is assigned to a slot, the remaining free space of that slot is wasted.



ii. External fragmentation occurs when a **dynamic memory allocation** technique is used where after loading and unloading of several slots here and there, the free space is being distributed rather than being contiguous.



For example: Consider the figure above where memory allocation is done dynamically.

In dynamic memory allocation, the allocator allocates only the exact needed size for that program.

First memory is completely free.

Then the Programs A, B, C, D and E of different sizes are loaded one after the other and they are placed in memory contiguously in that order.

Then later, Program A and Program C closes and they are unloaded from memory.

Now there are three free space areas in the memory, but they are not adjacent.

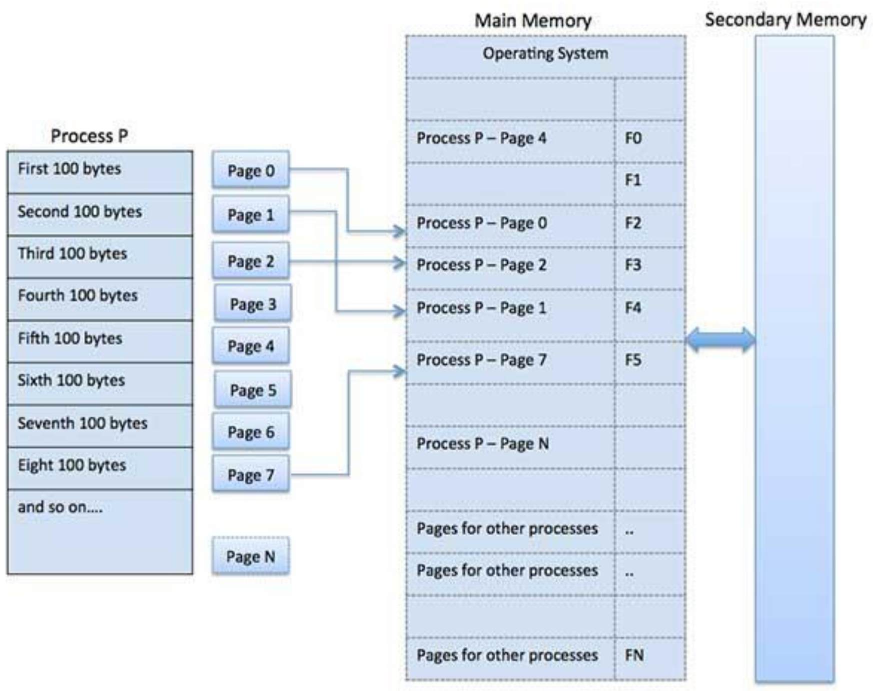
Now a large program called Program F is going to be loaded but neither of the free space block is not enough for Program F.

The addition of all the free spaces is definitely enough for Program F, but due to the lack of adjacency that space is unusable for Program F.

When an allocated partition is occupied by a program that is lesser than the partition, remaining space goes wasted causing **internal fragmentation**. When enough adjacent space cannot be found after loading and unloading of programs, due to the fact that free space is distributed here and there, this causes external fragmentation. Fragmentation can occur in any memory device such as RAM, Hard disk and Flash drives.]

In paging:

- external fragmentation is avoided. All frames (physical memory) can be used by the processes)
- there is possible internal fragmentation
- The physical memory used by a process is no longer contiguous
- The logical memory of a process is still contiguous
- The logical and physical addresses are separated so the process does not see the translation or the difference to having physical memory



The **MMU** maps virtual addresses to physical addresses at runtime - Implementation must be done in hardware for efficiency

To produce the physical address, the page in the Page Map Table (PMT) needs to be looked up to find the frame number in which it is stored.

Then multiply the frame number by the frame size and add the offset to get the physical address

Demand Paging

This is an extension which takes advantage of the fact that not all parts of a program actually have to be in memory at the same time.

At any given point in time the CPU is accessing one page of a process and at that point it doesn't really matter whether the other pages of that process are in memory or not. Pages using demand paging, are only brought into memory on demand

E.g. when a page is referenced, memory is checked to see if it's there to complete the access and if it's not, then the page is swapped – meaning it's brought in from secondary memory into an available frame. Access is finally completed.

Demand paging removed the restriction of there needing to be an upper limit on the process size.

A computer can address more memory than the amount physically installed on the system. This extra memory is actually called virtual memory and it is a section of a hard drive that's set up to emulate the computer's RAM.

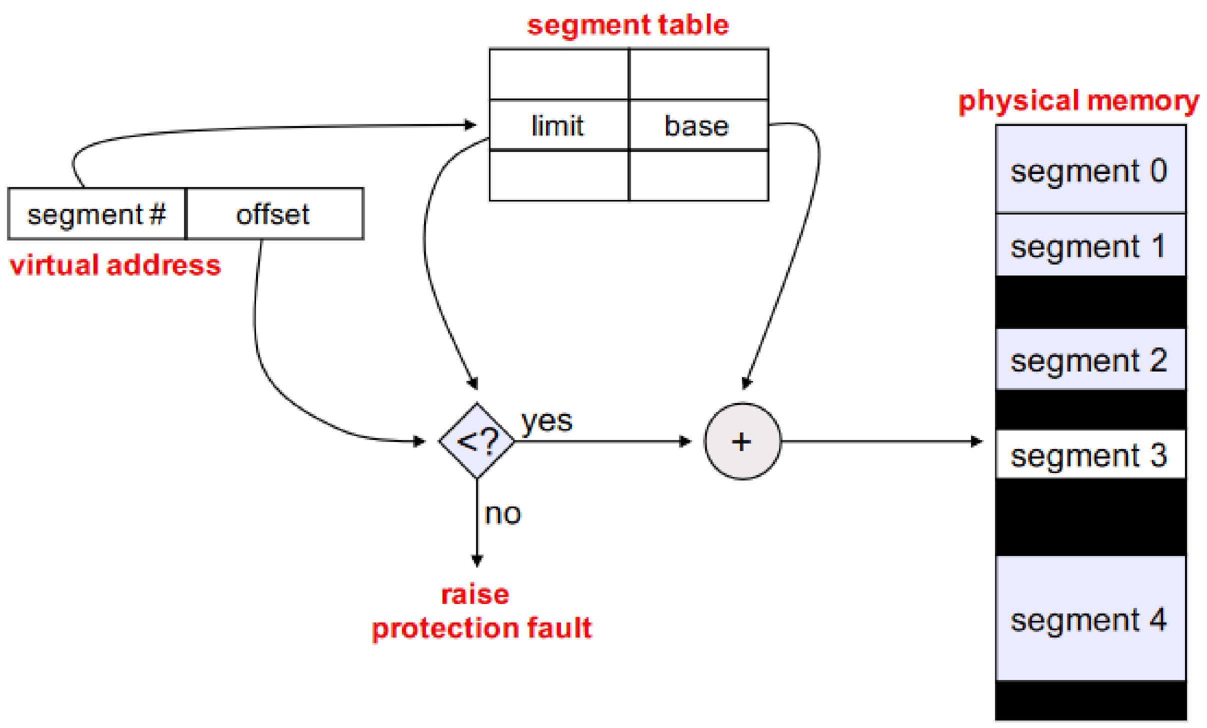
Paging technique, in particular demand paging, plays an important role in implementing ***virtual memory***.

The principle operation of memory management is to bring processes into main memory for execution by the processor. In almost all modern multiprogramming systems this involves a sophisticated scheme known as ***virtual memory***

Virtual memory is in turn based on the use of one or both basic techniques: segmentation and paging.

Segmentation Memory Management Technique

A user program can be subdivided using segmentation, in which the program and its associated data are divided into a number of ***segments***. It is not required that all segments of a program be of the same length, although there is a maximum segment length.



As with paging, a logical address using segmentation consists of two parts, in this case a segment number and an offset.

When a process gets loaded into main memory, its different segments can be located anywhere. Because of the unequal-size segments, segmentation is similar to dynamic partitioning. In the absence of an overlay scheme or the use of virtual memory, it would be required that all of a program's segments be loaded into memory for execution.

variable-sized partition = 1 segment/process

segmentation = many segments/process

i.e. Logical address = segment number + offset

The hardware must map a segment/offset into one-dimensional address.

The difference compared to dynamic partitioning is that with segmentation a program may occupy more than one partition, and these partitions need not be contiguous.

Segmentation eliminates internal fragmentation because each segment is fully packed with instructions/data

Like dynamic partitioning, it suffers from external fragmentation – but this will be less because of the nature of the process being broken up into a number of smaller pieces.

Whereas paging is invisible to the programmer, segmentation is usually visible and is provided as a convenience for organising programs and data (e.g. data in one segment, code in another segment – the programmer/compiler may further break down these segments into multiple segments). The main inconvenience of this service is that the programmer must be aware of the max. segment size limitation

Another issue with these unequal size segments is that there is no simple relationship between logical addresses and physical addresses. Equivalent to paging, a simple segmentation scheme would make use of a segment table for each process and list of free blocks in main memory.

The OS maintains this segment table for each process. Each entry contains:

- the starting physical addresses of that segment.
- the length of that segment (for protection by ensuring invalid addresses are not used)

Simple segmentation and paging comparison

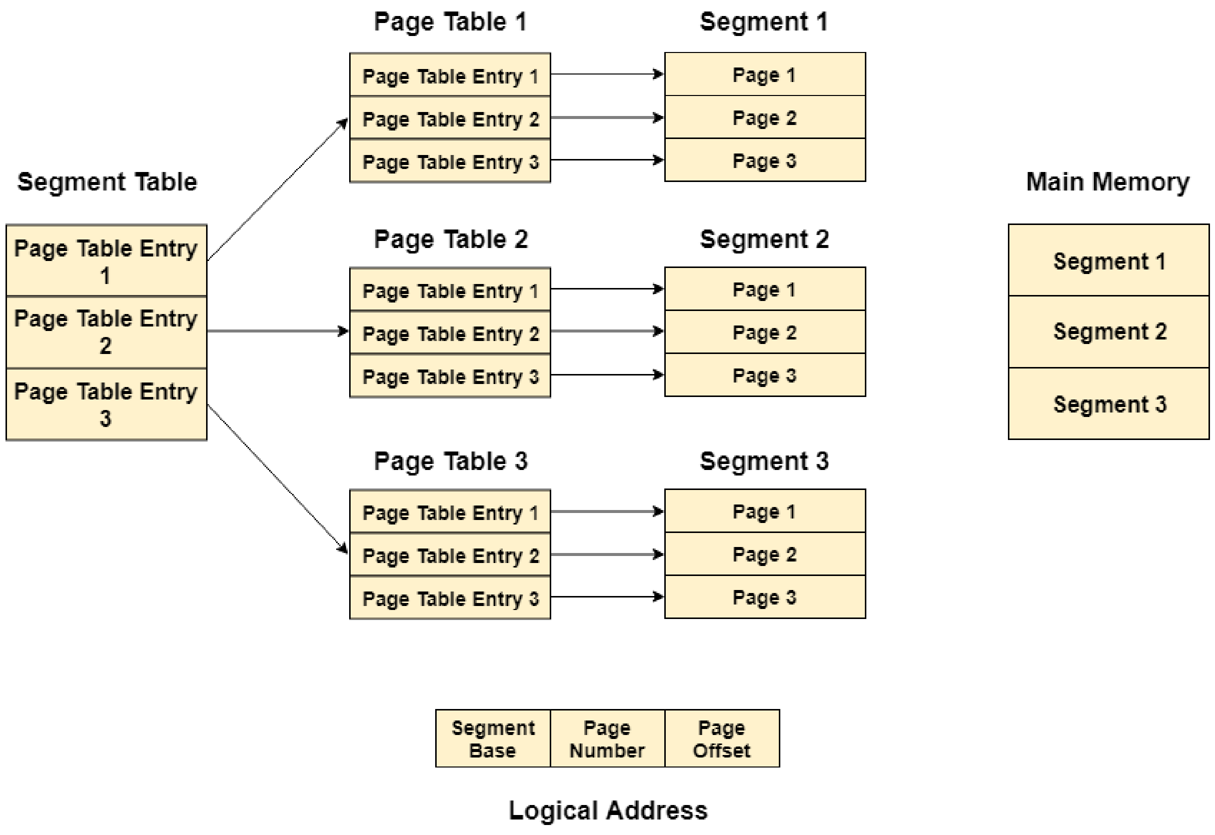
- Segmentation requires more complicated hardware for address translation
- Segmentation suffers from external fragmentation
- Paging only yield a small internal fragmentation
- Segmentation is visible to the programmer whereas paging is transparent
- Segmentation can be viewed as commodity offered to the programmer to organize logically a program into segments and using different kinds of protection (e.g. execute-only for code but read-write for data)
    - For this we need to use protection bits in segment table entries

Segmented Paging

Segmentation is rarely used alone as a memory management technique. The MULTICS** system solved problems of external fragmentation and lengthy search times by paging the segments, called ***Segmented Paging***

In Segmented Paging, the main memory is divided into variable size segments which are further divided into fixed size pages.

1. Pages are smaller than segments.
2. Each Segment has a page table which means every program has multiple page tables.
3. The logical address is represented as Segment Number (base address), Page number and page offset.



Advantages of Segmented Paging

1. It reduces memory usage.
2. Page table size is limited by the segment size.
3. Segment table has only one entry corresponding to one actual segment.
4. External Fragmentation is not there.
5. It simplifies memory allocation.

Disadvantages of Segmented Paging

1. Internal Fragmentation will be there.
2. The complexity level will be much higher as compare to paging.
3. Page Tables need to be contiguously stored in the memory.

Solution differs from pure segmentation in that the segment-table entry contains not the base address of the segment, but rather the base address of a page table for this segment.

***Multics (Multiplexed Information and Computing Service) is an influential early time-sharing operating system, based around the concept of a single-level memory. Virtually all modern operating systems were heavily influenced by Multics (often through Unix)*

Partitions

In most schemes for memory management we can assume that part of the memory holds the operating system, the remainder holds the user program.

One of the first mechanisms used to protect the operating system, and to protect processes from each other was **partitions**.

Partitions can be equal or unequal sizes.

The simplest scheme for managing this available memory is to partition it into regions with fixed boundaries and the partitions are of equal size, called ***fixed partitioning***.

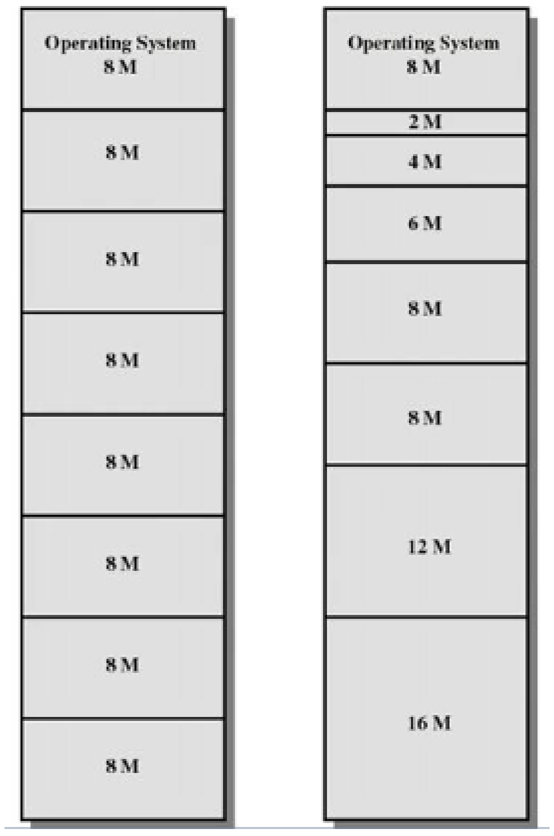


Figure 4 - Fixed Partition of a 64MB RAM v's Dynamic Partitioning

- ➔ Any process whose size is <= partition size can be loaded into any available partition.
- ➔ If all partitions are full and no process is in the READY or RUNNING state, the OS can swap a process out of any of the partitions and load in another process (function is to keep the processor busy at all times!)

Problems with Partitions

1) Memory utilisation is extremely inefficient – whatever size the program is, say only 2MB, it will occupy an 8MB partition when it's swapped in
2) The second major problem is ***fragmentation*** of physical memory as the number of small holes builds up.

The cause of both problems is the fact that the logical memory map is **contiguous:**

Recall, for each process, all the machine code is in one lump, all the data is in one lump etc. The process sees a single region of logical memory, and this is mapped to a single region of physical memory.

If we could make these lumps *appear* to be contiguous to the process (i.e. contiguous logical memory), but actually break the lumps into small physical memory units, then we could fill the unused physical holes with these small units.

Dynamic partitioning

To overcome some of the difficulties with fixed partitioning, an approach known as ***dynamic partitioning*** was developed (but today, this has been replaced by more sophisticated techniques)

The partitions are of variable length and number.

Each process is allocated exactly as much memory as it requires, so the partition size is the same as the process size. Eventually, again, holes are formed in main memory. This is called ***external fragmentation*** (the holes are external to the partitions)

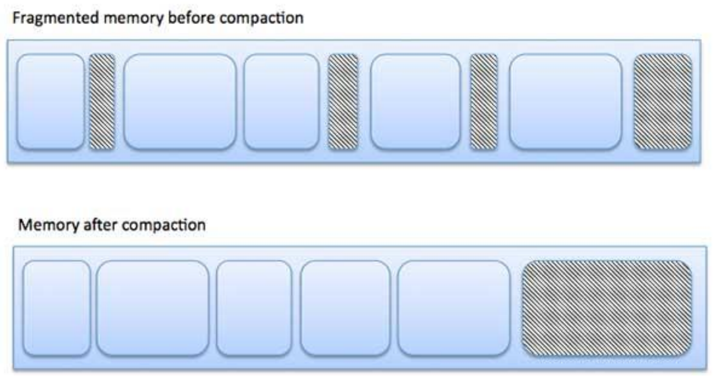
Some form of compaction must be used to shift processes so they are contiguous and all free memory is in one block

Compaction Techniques

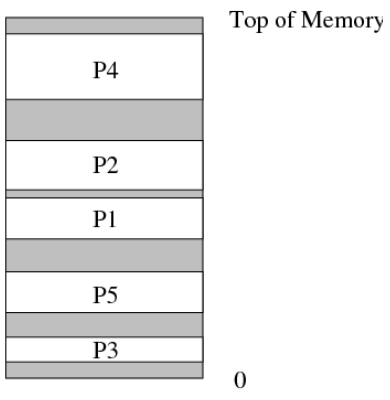
Fragmentation can cause waste of memory and a compaction technique can be used to create more free memory out of fragmented memory.

External fragmentation can be reduced by a technique called ***compaction*** or shuffle memory contents to place all free memory together in one large block. To make compaction feasible, relocation should be dynamic.

The internal fragmentation can be reduced by effectively assigning the smallest partition but large enough for the process.



To the operating system, the available **physical** memory looks like a number of **holes in partitions** (blocks of available memory):



The problems with Compaction are that it is time consuming and a waste of CPU time.

Note that compaction needs dynamic relocation capability in which a program can be moved from one region in memory to another without any issues.

The OS designer must choose the best suited algorithm to assign processes to memory. When it's time to load/swap a process into memory, and if there's more than one free block of memory of sufficient size, then the OS must decide which free block to allocate.

Placement Algorithm

Used to decide which free block to allocate to a process

Goal: to reduce usage of compaction (because it is time consuming).

Three placement algorithms that satisfy a request of size ***n*** from a list of free holes that might be considered are:

1) Best-fit: chooses the block that is closest in size to the request
2) First-fit: begins to scan memory from the beginning and choses the 1st available block that is large enough
3) Next-fit: begins to scan memory from the location of the last placement and choose the next available block that is large enough

First-fit and best-fit better than worst-fit in terms of speed and storage utilisation

Problems with Partitions (continued)

One problem with partitions is how much memory to allocate initially?

i. If too little, the process may run out of memory when data and stack collide.
ii. If too much, then memory is unused and wasted. It is usually impossible to change the base/limit registers once set, as there are other processes are usually immediately above/below.

Uses of partitions

Partitions were used mainly on batch systems: programs were waiting to be started (not in memory as yet), several processes ready to run, one running process, and zero or more blocked processes.

Because partitions sizes are pre-set at system generation time, small jobs will not utilise partition space efficiently. It's reasonable to use where the main storage requirement of all jobs is known beforehand i.e. in batch processing, but in general, other than that it's an inefficient technique.

Addressing

When the fixed partition scheme, we can expect that a process will always be assigned to the same partition – whichever partition is selected when a new process is loaded will always be used to swap that process back into memory after it has been swapped out. With this, a simple relocating loader can be used so that when the process is first loaded, all relative memory references in the code are replaced by absolute main memory addresses, determined by the base address of the loaded process.

In the case of equal-size partitions and in the case of a single process queue for unequal size partitions, a process may occupy different partitions during the course of its execution. When a process image is first created, it is loaded into some partition in main memory.

Later the process may be swapped and when it is subsequently swapped back in, it may be assigned to a different partition when the last time. The same is true from ***dynamic partitioning***.

Furthermore, when ***compaction*** is used, processes are shifted while they are in main memory. This the locations (of instructions and date) referenced by a process are not fixed, but will change each time a process is swapped or shifted.

Because of swapping and compaction, a process may occupy different main memory locations during its lifetime. Hence physical memory references by a process cannot be fixed. This problem is solved by distinguishing between several types of addresses:

- A physical address (absolute address) is a physical location in main memory
- A logical address is a reference to a memory location independent of the current assignment of program/data to memory

  Compilers produce code in which all memory references are logical addresses
- A relative address is an example of logical address in which the address is expressed as a location relative to some known point in the program (e.g.: the beginning)
- Physical addresses are calculated "on the fly" as the instructions are executed

Address Translation

Relative address is the most frequent type of logical address used in program modules (i.e.: executable files). Such modules are loaded in main memory using dynamic run-time loading.

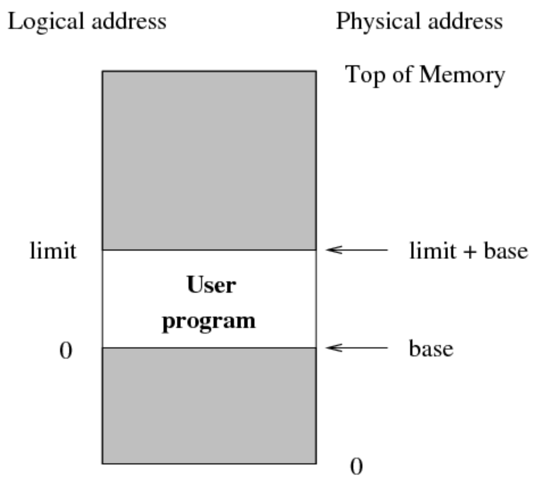
Typically, all of the memory references in the loaded process are relative to the origin of the program and the translation from relative to physical address must be done by hardware mechanism at the time of execution of the instruction that contains the reference.

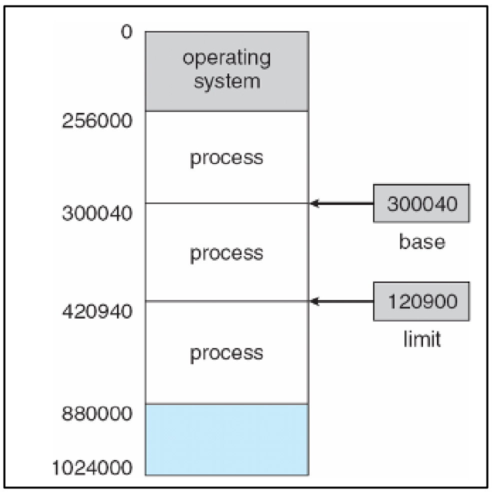
Base Register and Limit Register

A **pair of** special hardware processor registers are added to the memory address decoder: the **base** and **limit** registers, and these **define the logical address space.**

1. The ***base register*** is loaded with the starting address in main memory of the program.
2. The ***bounds*** or ***limit register*** indicates the end location of the program.

These registers must be set when the program is loaded into memory or when the process image is swapped in.





When a process reads from or writes to address `X', the memory decoder adds on the value of the base register, so the actual operation become a read or write to address `base + X'.

The CPU must check every memory access generated in user mode to be sure that it's within the base and limit for that user

If the input address is lower than `0' or higher than `limit', the memory hardware considers this an error, and informs the OS of a memory access error (usually via an ***interrupt***). Thus, processes can only access memory within these limits.

This addition of hardware allows multiple processes to be loaded into memory and to run without interference from each other.

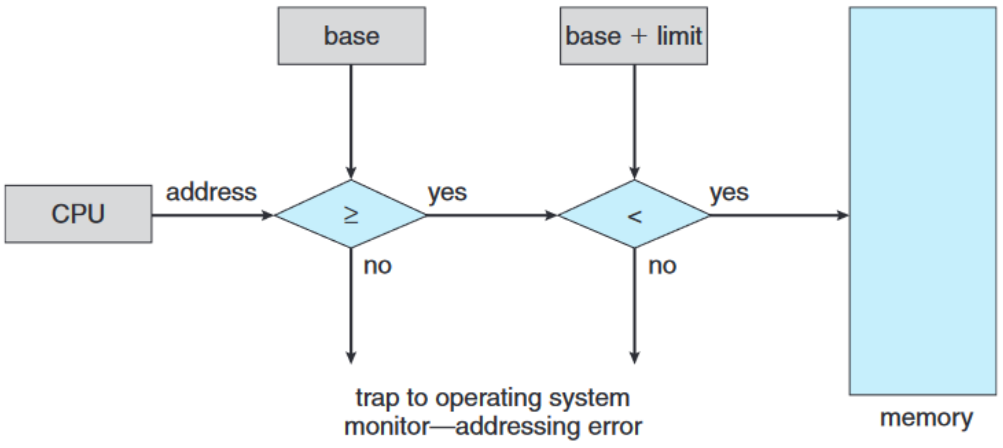


Figure 5 - Hardware Address Protection

Figure 6 - Hardware Support for Relocation below shows the way in which this address translation is typically accomplished.

When a process is assigned to the RUNNING state, when the program is loaded into memory/swapped in, the base register is loaded with the starting address in main memory of the program and the limit register loaded with the ending location of the program.

During the course of execution of the process, relative addresses are encountered. These include the contents of the instruction register, instruction addresses that occur in branch and call instructions and data addresses that occur in load and store instructions.

Each such relative address goes through two steps of manipulation by the processor:

1) The value in the base register is added to the relative address to produce an absolute address
2) The resulting address is compared to the value in the limit register. If the limit is within the bounds, the instruction execution may proceed. Alternatively, an interrupt is generated to the OS, with must respond to the error in some way

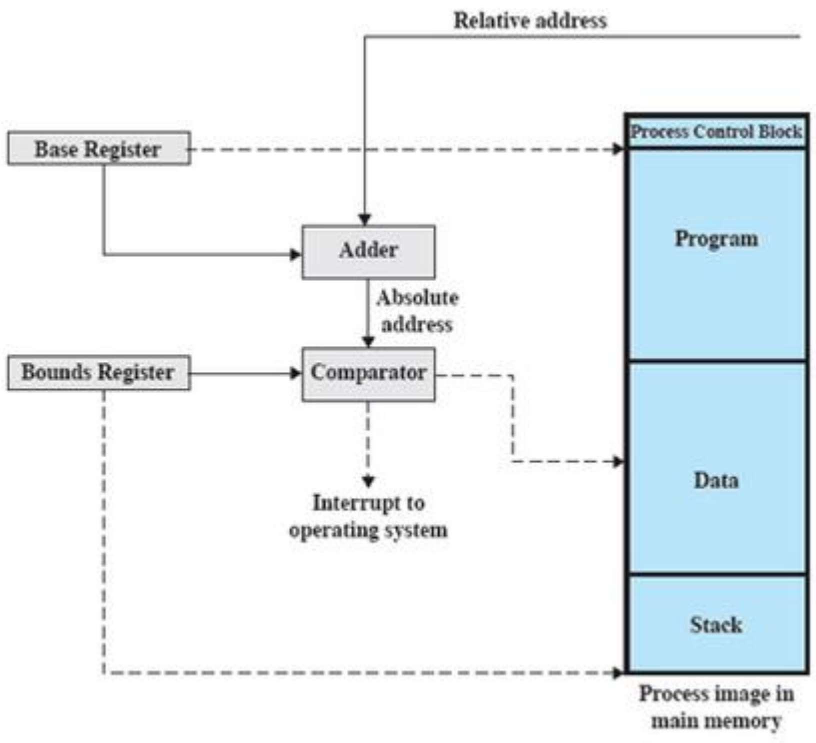


Figure 6 - Hardware Support for Relocation

Figure 6 - Hardware Support for Relocation above allows programs to be swapped in and out of memory during the execution

It also provides a measure of protection: each process images is isolated by the contents of the base and limit registers and unwanted access by other processes.

When the OS performs a context switch, it must remember to switch the *memory maps* of two processes.

This is done by changing the values of the two registers.

Each process has its own individual pair of base/limit registers, and the OS chooses these pairs so that process' memory maps never overlap, and each process has enough memory.