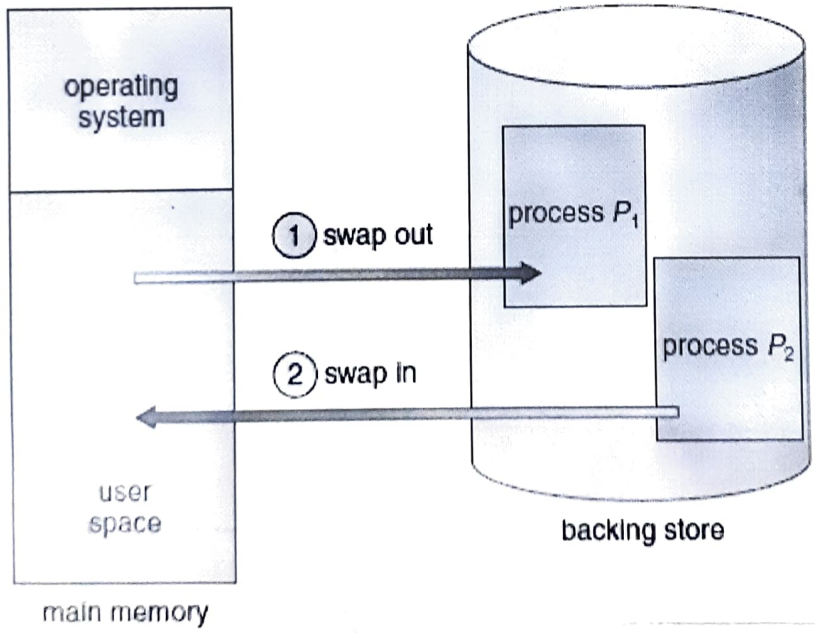
Swapping

A process can be swapped temporarily out of memory to a backing store and then brought back into memory for continued execution.

Swapping makes it possible for the total physical address space of all processes to exceed the real physical memory of the system, thus increasing the degree of multiprogramming.

Standard swapping involves moving processes between main memory and a backing store. The backing store is commonly a fast disk.



The system maintains a ready queue consisting of all processes whose memory images are on the backing store or in memory and are ready to run.

The dispatcher checks to see whether the next process in the queue is in memory. If it is not, free, and if there is no free memory region, the dispatcher swaps out a process currently in memory and swaps in the desired process.

It then reloads registers and transfers control to the selected process.

The context-switch time in such a swapping system is fairly high.
Example:
User process size = 100MB
Backing Store Transfer rate = 50MB/sec
Transfer time = $\frac{100\,MB}{50\,MB/sec}$ = 2 sec

Considering to and fro = 4 sec.
(majority of swap time is transfer time)

Process waiting for an I/O operation cannot be swapped. Or execute I/O operations only into OS buffers. Transfers between OS buffers and process memory then occur only when the process is swapped in. This double-buffering itself adds overhead. We need to copy the data again, from kernel memory to user memory, before the user process can access it.

To sum up,

- If there is a Pending I/O, we cannot swap as I/O would occur with wrong process
- Transfer I/O to kernel space & then to I/O, adds overhead.

Usually modern operating systems swap only when free memory is extremely low.

Contiguous Memory Allocation

The main memory must accommodate both the OS and the various user processes. Hence the main memory needs to be allocated the most efficient way.

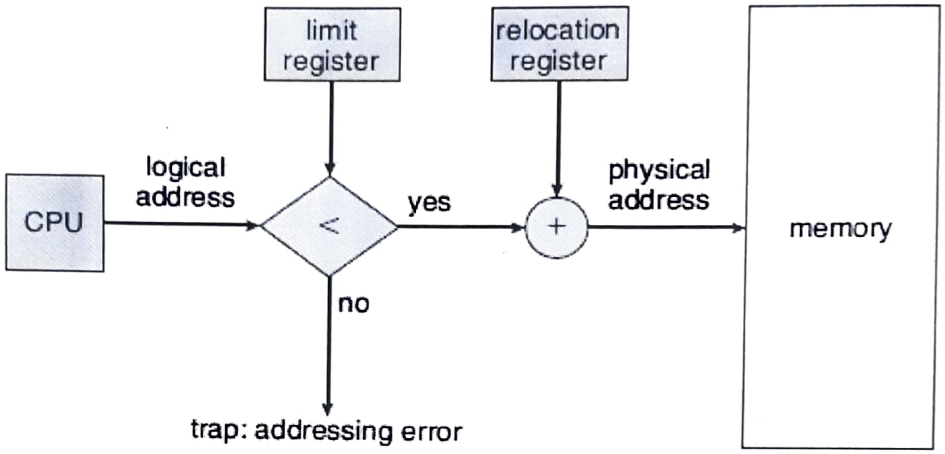
The memory is usually divided into two partitions

- One for resident OS
- One for the user processes.

OS can be placed either in low memory or high memory. As interrupt vector is often in low memory, programmers usually place the OS in low memory as well.

In contiguous memory allocation, each process is contained in a single section of memory that is contiguous to the section containing the next process.

Memory protection



When CPU scheduler selects a process for execution, the dispatcher loads the relocation and limit registers with the correct values as a part of the context switch. Because every address generated by a CPU is checked against these registers, we can protect both the OS and other users programs and data from being modified by this running process.

The relocation-register scheme provides an effective way to allow the OS size to change dynamically. Eg: a driver code might not always be required in memory. A transient OS code comes and goes as needed.

Using a transient code changes the size of OS during program execution.

prakash hegade

Memory Allocation

Fixed-Sized Partitions

- divide memory into several fixed-sized partitions
- Each partition may contain exactly one process
- The degree of multiprogramming is bound by the number of partitions
- The partition is used when free and returned back after use

Variable Partition

- OS keeps a table indicating occupied and free memory
- Initially its one large block of available memory called a hole. Eventually, there will be holes of various sizes.

Dynamic Storage-Allocation Problem

- How to satisfy a request of size n, from a list of free holes.

Most Commonly used Strategies:

→ first-fit
→ best-fit
→ worst-fit

to select a free hole from the set of available holes.

First Fit

- Allocate the first hole that is big enough
- Search can start at beginning or where previous first-fit ended.

Best Fit

- Allocate smallest hole that is big enough.
- Search entire list, unless list is ordered by size
- Produces smallest leftover hole

Worst Fit

- Allocate the largest hole
- Search entire list, unless sorted by size
- Produces Largest left over hole, which may be more useful than smallest leftover hole from best fit.

Simulations show that

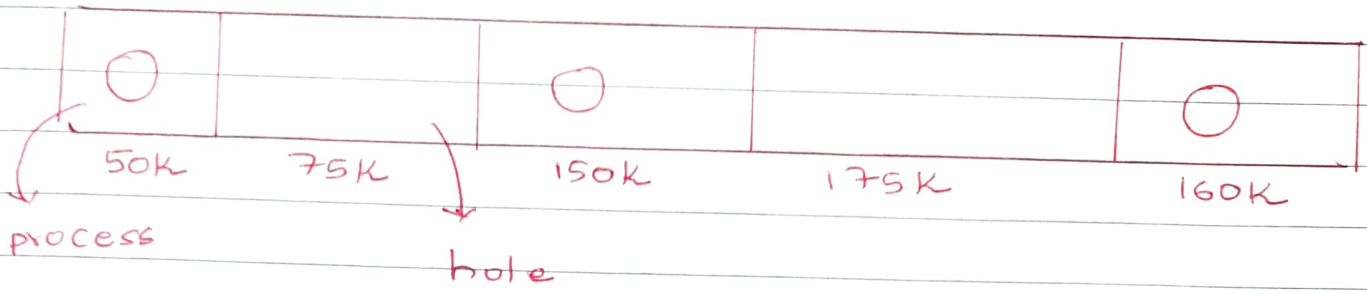
- both first fit and best fit are better that worst fit in terms of decreasing time and storage utilization

- first fit is generally faster

- In terms of storage utilization, neither first fit nor best fit is better than the other.

Problem

Consider the memory below:



Consider the next requirement for 25K. What happens in each case?

First fit: Gets space in second partition which is free. Allocates 25K & remaining 50K is hole

Best fit: After scanning for all free, 75K is better one. 50K is produced as hole

Worst fit: Allocates from 175K. A hole of 150K is created.

Fragmentation

Both first and best fit strategies suffer from external fragmentation.

↓

When storage is fragmented into large number of small holes, there might be enough memory for a new request but is not contiguous.

Depending on the total amount of memory storage and the average process size, external fragmentation may be a minor or a major problem.

Statistical analysis of first fit, per say, reveal that, even with some optimization, given N allocated blocks, another 0.5N blocks will be lost to fragmentation. That is, one-third of memory may be unusable.

↳ 50-percent rule

Consider, hole = 18,464 bytes & request is 18,462 bytes.
The overhead of keeping track of this hole will be larger than available memory.
The general approach to avoid this is to break the memory into fixed sized blocks & allocate in units based on block size.
The unused memory internal to partition is called internal fragmentation.

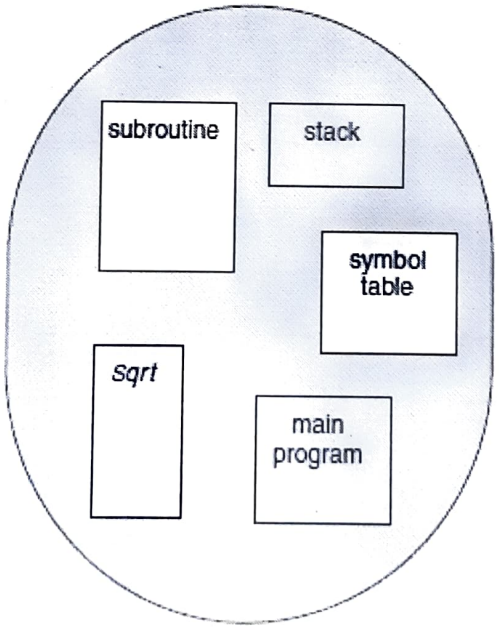
Solutions to External fragmentation:

- Compaction: shuffle memory so as to place all free memory together in one large block. Compaction is not always possible. It is possible only when relocation is dynamic and is done at execution time.

- Permit logical address space of the process to be non-contagious.
  - ↳ Segmentation
  - ↳ Paging

Segmentation

Programmers prefer to view memory as a collection of variable-sized segments, with no necessary ordering among the segments.



Segmentation is a memory management scheme that supports programmer view of memory.

A logical address space is collection of segments.

Each segment will have a name and a length. The addresses specify both the segment name and the offset within the segment. The programmer hence specifies each address by two quantities:

- a segment name
- an offset

Segment names are usually numbers. Thus, a logical address consists of two tuples:

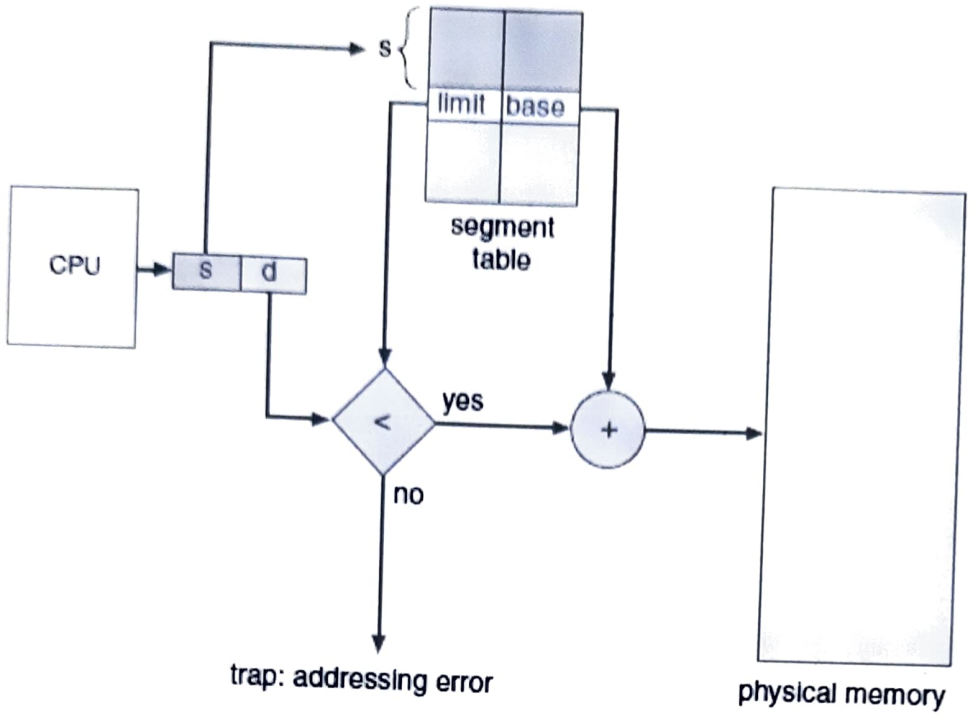
<segment-number, offset>

Usually, a compiler automatically constructs segments for (C-compiler)

- code
- global variables
- heap
- stacks used by each thread
- standard C library.

Libraries that are linked in during compile time might be assigned separate segments. The loader would take all these segments & assign them segment number.

We now need to map two-dimensional user-defined addresses into one-dimensional physical addresses.



Segmentation

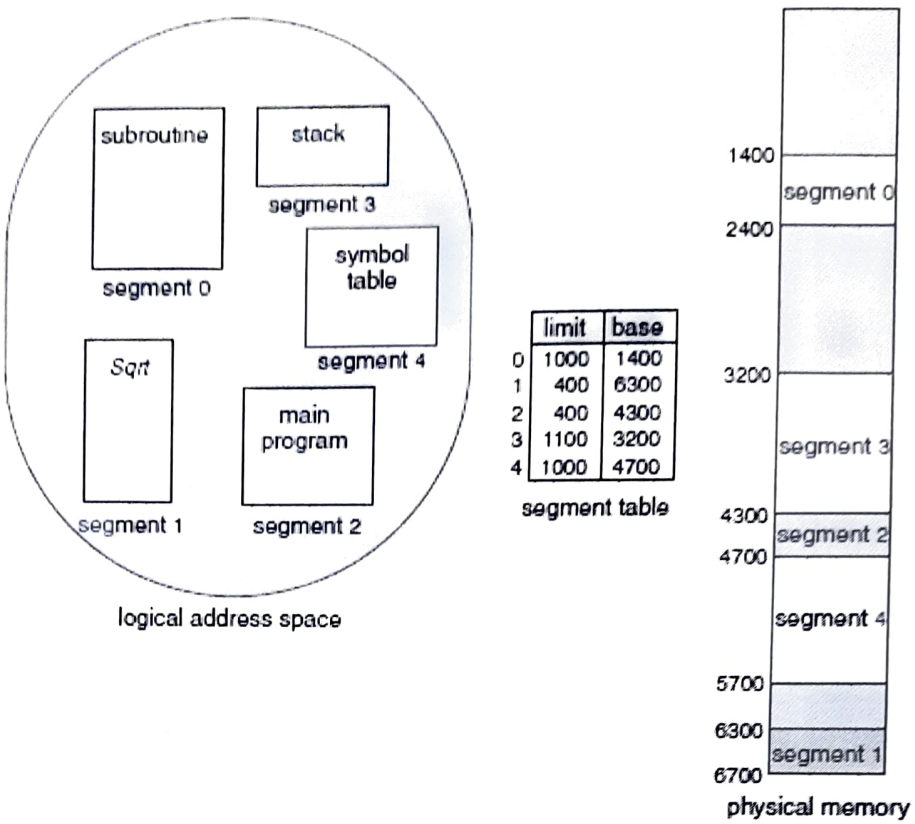
Hardware

Each entry in Segment table has Segment base and Segment limit.
The Segment base contains the starting physical address where the Segment resides in memory and the Segment limit specifies the length of the Segment.

A logical address consists of 2 parts
- Segment s
- offset d

Segment number is used as index for Segment table. When an offset is legal, it is added to the Segment base to produce address in physical memory of the desired byte.

Example:



- Segment 2 is 400 bytes long and begins at location 4300.
  A reference to byte 53 of Segment 2 is mapped onto location 4300 + 53 = 4353

- A reference to Segment 3, byte 852, is mapped to 3200 + 852 = 4052

- A reference to byte 1222 of Segment 0 would result in a trap to the OS as this Segment is only 1,000 bytes long.