Memory Management

1 Introduction

The **base register** holds the smallest legal physical memory address; the **limit register** specifies the size of the range. The base and limit registers can be loaded only by the operating system, which uses a special privileged instruction in kernel mode preventing users from updating it.

Address Binding

The processes on the disk that are waiting to be brought into memory for execution form the **input queue**. A compiler will typically **bind** symbolic addresses to relocatable addresses ("14 bytes from the beginning of this module"). The linkage editor or loader will in turn bind the relocatable addresses to absolute addresses. Each binding is a mapping from one address space to another.

- **Compile time**. If you know at compile time where the process will reside in memory, then absolute code can be generated i.e physical address is embedded to the executable of the program during compilation. Loading the executable as a process in memory is very fast. But if the generated address space is preoccupied by other process, then the program crashes and it becomes necessary to recompile the program to change the address space.
- **Load time**. If it is not known at compile time where the process will reside in memory, then the compiler must generate relocatable code. In this case, final binding is delayed until load time. The base address of the process in main memory is added to all logical addresses by the loader to generate absolute address. In this if the base address of the process changes then we need to reload the process again.
- **Execution time**. If the process can be moved during its execution from one memory segment to another, then binding must be delayed until run-time. This is used if process can be moved from one memory to another during execution. Eg- Compaction.

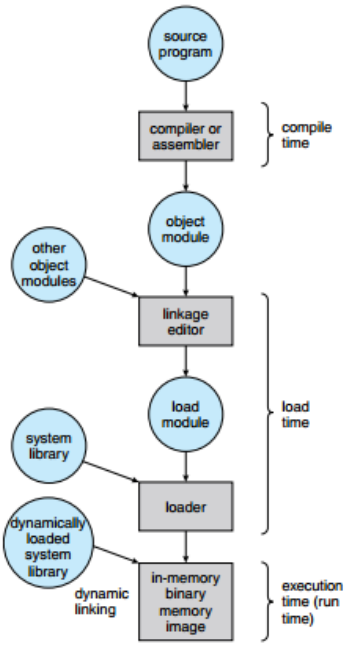
Logical And Physical Address Space

Logical address is an address generated by the CPU, whereas an address seen by the memory unit, the one loaded into the **memory-address register** of the memory—is **physical address**. The set of all logical addresses generated by a program is a **logical address space** and the set of all physical addresses corresponding to these logical addresses is a **physical address space**.

In execution-time address-binding scheme, the run-time mapping from virtual(logical) to physical addresses is done by a hardware device called the **memory-management unit (MMU)**. The compile-time and load-time address-binding methods generate identical logical and physical addresses.

Dynamic Loading

To obtain better memory-space utilization, we use **dynamic loading**, in which a routine is not loaded until it is called. All routines are kept on disk in a relocatable load format. Only after the main program begins execution and calls a routine, do we load the routine into memory. The control is handed to the newly loaded routine. Dynamic loading does not require special support from the operating system.



Dynamic Linking

Requirements of Memory Management System

- **Relocation**. The available memory is shared among many processes. It is not possible to know in advance which other programs will be resident in main memory at the time of execution of a program. We may need to relocate the process to a different area of memory as the memory location of previous execution may be occupied by another process after swapping.
- **Protection**. A process's memory must be protected from the access of another process. There is a trade-off between ease of relocation and protection of memory.
- **Sharing**. The protection mechanism must allow privileged processes to access the same portion of memory.
- **Abstraction**. The logical view of memory for a programmer must be of a continuous memory. One must not have to worry about the where the process is being executed and how it is being swapped between main memory and secondary memory.

2 Swapping

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

The backing store is commonly a **fast disk**. It must be large enough to accommodate copies of all memory images for all users, and it must provide direct access to these memory images. 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. Whenever the CPU scheduler decides to execute a process, it calls the dispatcher which checks if the next process in the queue is in memory. If it is not, 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 **roll out, roll in** swapping scheme is used for priority-based scheduling. If a higher-priority process wants service, lower-priority process is swapped out till the former completes and then the latter can be swapped back in and continued.

We must ensure that a process is **completely idle** before swapping it. It may be waiting for an I/O operation. We can either never swap a process with pending I/O, or execute I/O operations only into OS buffers. Transfers between OS buffers and process memory occur when process is swapped in.

3 Contiguous Memory Allocation

Since **interrupt vector** is often in low memory, the OS is usually placed in low memory. In **contiguous memory allocation**, each process is contained in a single contiguous section of memory.

Memory Mapping and Protection

Memory mapping and protection is provided by relocation register, together with limit register. When the CPU scheduler selects a process for execution, the dispatcher loads the relocation and limit registers with the correct values as part of the context switch.

This scheme provides a way to alter OS's size dynamically. If a device driver (or other OS service) is not commonly used, we do not want to keep the code and data in memory. The OS code that comes and goes when needed is called **transient OS code**.

Memory Allocation

- **MFT**. Multiprogramming with Fixed Number of Tasks. In this, memory is divided into several fixed-size partitions. Each partition can contain at-most one process.
- **MVT**, Multiprogramming with Variable Number of Tasks. In this, OS keeps a table indicating which parts of memory are available. Initially, all memory is available as a single block, called hole. Gradually, memory blocks comprise a set of holes of various sizes scattered through memory. When a process requires memory, it is assigned a block. If it is too large, it is split into two parts. If the new hole is adjacent to other holes, they can be merged.

  Dynamic storage allocation problem is resolved by:

  - **First fit**. Allocate the first hole that is big enough. Searching starts either at beginning of the set of holes or at the location where the previous first-fit search ended (next fit). It allocates memory at the beginning of memory and may lead to internal fragments there.
  - **Best fit**. Allocate the smallest hole that is big enough producing smallest leftover hole. We must search the entire list, unless the list is ordered by size.
  - **Worst fit**. Allocate the largest hole producing the largest leftover hole, which may be more useful than the smaller leftover hole from a best-fit approach.

Fragmentation

- **External Fragmentation** exists when there is enough total memory space to satisfy a request but available spaces are not contiguous. It occurs mostly in first-fit and best-first approach. The **50 percent rule** states that given N allocated blocks, another 0.5 blocks will be lost to fragmentation.

- **Internal Fragmentation** exists when a larger block of memory is allocated to a process than required to avoid the overhead of tracking the hole formed after splitting resulting in unused memory internal to a partition.

Shuffling the memory contents to merge all the free memory into a large block is called **compaction**. The backing store has the same fragmentation problems as with main memory, but access is much slower, so compaction is impossible.

4 Buddy System

Static partition schemes suffer from the limitation of having the fixed number of active processes and the usage of space may also not be optimal.

Buddy allocation system is an algorithm in which a larger memory block is divided into small parts to satisfy the request. This algorithm is used to give **best fit**. The two smaller parts of block are of equal size and called as **buddies**. One of the two buddies will further divide into smaller parts until the request is fulfilled.

Benefit of this technique is that the two buddies can combine to form the block of larger size according to the memory request minimizing external fragmentation. However, internal fragmentation can be large.

5 Paging

Paging is a memory-management scheme that permits the physical address space of a process to be non-contiguous. The physical memory is broken into fixed-sized blocks called **frames** and the logical memory is broken into blocks of same size called **pages**. The backing store is divided into fixed-sized blocks that are of the same size as the memory frames.

Every logical address is split into **page number** and **page offset**. The page number acts as an index into **page table**, which contains the base address of each page in memory.

Paging itself is a form of **dynamic relocation**. Every logical address is bound by the paging hardware to some physical address. Using paging is similar to using a table of base (or relocation) registers, one for each frame of memory.

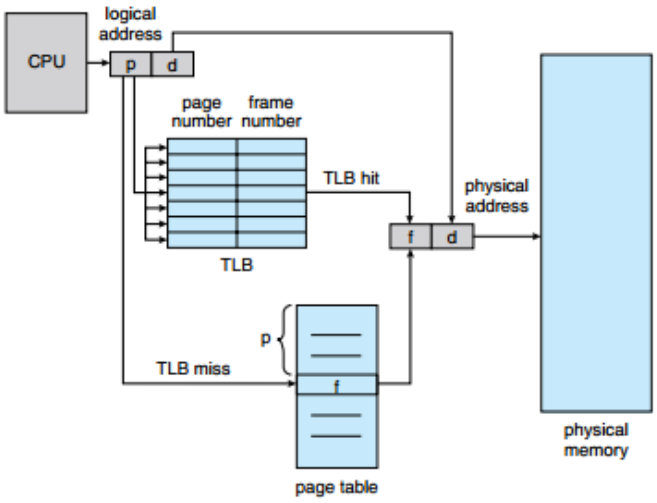
Internal fragmentation averages **one-half page** per process. So, small page sizes could be desirable. However, overhead is involved in each page-table entry which is reduced as the size of the pages increases. Also, disk I/O is more efficient when the amount data being transferred is larger.

Frame table has the information about all the allocation details of physical memory, which is required by the OS to manage physical memory.

Page Table Entries

- **Frame Number** specifies the location of the frame in memory.
- **Valid bit** is set if page is present in memory.
- **Protection bit** specifies the type of protection (read, write, etc).
- **Referenced bit** is set if the page has been referred in the last clock cycle.
- **Dirty bit** is set if the page has been modified and is not saved in disk.
- **Cache bit** is unset if we dont want to save page in cache(if it is accesed only once, etc)

Hardware support



The OS maintains a copy of page table for each process for quick translation, thereby increasing context-switch time. The page table is kept in main memory due to limited register capacity, and a **page-table base register (PTBR)** points to the page table. Changing page tables requires changing only this one register, substantially reducing context-switch time.

To deal with the increase in memory accesses, we use a special, small, fastlookup hardware cache, called a **translation look-aside buffer**. The TLB is associative, high-speed memory. 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. Some TLBs allow certain entries to be **wired down**.

Protection

Shared Pages

If the code is **reentrant code(or pure code)**, common code can be shared among all users. with user only requiring to store its own data page.

6 Structure of Page Table

Hierarchical Paging

Hashed Page Tables

Inverted Page Tables

7 Segmentation

Segmentation is a memory-management scheme that supports user view of memory as a collection of segments. Each segment has a name and a length. The user specifies each address by two quantities:

a **segment name** and an **offset**. (In the paging scheme, user specifies only a single address, which is partitioned by the hardware into a page number and an offset). Each entry in the segment table has a segment base and a segment limit.

There is no internal fragmentation and the segment table consumes less space in comparison to page table. However, as processes are loaded and removed from the memory, the memory can be fragmented into small chunks causing external fragmentation.