Memory Management

Arnav Gupta

March 26, 2024

Contents

1 Memory Management Requirements

Frame: a fixed-length block of main memory

Page: a fixed-length block of data that resides in secondary memory, may be temporarily copied into a frame of main memory

Segment: a variable-length block of data that resides in secondary memory, may be temporarily copied into an available region of main memory (segmentation) or the segment may be divided into pages which can individually be copied into main memory (combined segmentation and paging)

Memory management is intended to satisfy relocation, protection, sharing, logical organization, and physical organization.

1.1 Relocation

For multiprogramming systems, main memory is shared among processes that must be swapped in and out of main memory to maximize processor utilization. During swapping, the process may be required to relocate to a different area of memory.

Since a program's location is not known ahead of time, there are some concerns related to addressing:

- OS needs to know location of process control info, execution stack, and entry point to the program for this process
- processor must deal with memory references within the program such as branch instructions and data reference instructions

1.2 Protection

Each process should be protected against interference from other processes: no referencing memory locations in other processes, for instructions or data, without permission.

All memory references generated by a process must be checked at runtime to ensure they only refer to the memory space allocated to that process.

Memory protection must be satisfied by the processor rather than OS since it is only possible to assess the permissibility of a memory reference at the time of execution of the instruction making the reference.

1.3 Sharing

Protection must still allow processes cooperating on some task to share access to the same data structure.

The memory management system must allow controlled access to shared areas of memory without compromising essential protection.

1.4 Logical Organization

Most programs are organized into modules (some modifiable, some unmodifiable), and so if the OS and computer hardware can effectively deal with programs and data in the form of modules, then some advantages can be realized:

1. Modules can be written and compiled independently, with all references from one module to another resolved by the system at runtime.
2. With some additional overhead, different degrees of protection (read only, execute only, etc) can be given to different modules.
3. Possible to introduce mechanisms by which modules can be shared among processes, and so the advantage of providing sharing on a module level is that this corresponds to the user's way of viewing the problem, hence easy for the user to specify the sharing desired.

Mainly satisfied by segmentation.

1.5 Physical Organization

Moving information between two levels of memory should be a system responsibility because:

1. The main memory available for a program and its data may be insufficient, so the programmer will need to engage in overlaying, where the program and data are organized such that various modules can be assigned the same region of memory with a main program switching modules in and out as needed, though this wastes programmer time.
2. For multiprogramming, the programmer does not know at the time of coding how much space wll be available or where that space will be.

2 Memory Partitioning

Memory management in modern multiprogramming systems involves virtual memory, which is based on the use of segmentation and paging.

Memory management schemes assume the OS occupies some fixed portion of main memory and the remaining memory can be used by multiple processes.

2.1 Fixed Partitioning

Scheme is to partition available memory into regions with fixed boundaries.

Main advantage is they are simple to implement with little OS overhead.

Disadvantages:

- number of partitions specified at system generation time limits the number of active processes in the system
- since partition sizes are preset at system generation time, small jobs will not use partition space efficiently, which is in most cases inefficient (unless storage requirement of all jobs is known beforehand)

2.1.1 Partition Sizes

This could be done with equal-size partitions, but this has the following difficulties:

- a program may be too big to fit into a partition, which would require the programmer to use overlays so only a portion of the program needs to be in main memory at any time
- main memory utilization is inefficient as a program of any size will occupy an entire partition, leading to **internal fragmentation**

These problems can be lessened by unequal-size partitions, so that larger programs can be accommodated without overlays and smaller programs can be accommodated with less internal fragmentation.

2.1.2 Placement Algorithm

With equal-size partitions, a process can be loaded into any arbitrary available partition. If none exists, choosing which to swap out is a scheduling decision.

With unequal-size partitions, there are 2 possible strategies:

- assign each process to the smallest partition within which it will fit, so a scheduling queue is needed to hold swapped out processes destined for that partition
  - advantage: processes are always assigned to minimize wasted memory within a partition (less internal fragmentation)

    - disadvantage: larger partitions could always remain unused

- employ a single queue for all processes, and when loading a process into main memory, the smallest available partition that can hold the process is selected, or if all partitions are ocurpied swapping occurs based on the smallest partition that will hold the process, priority, or blocked proceses

2.2 Dynamic Partitioning

Partitions are of variable length and number, so when a process is brought into main memory it is allocated exactly as much memory as it requires.

This starts well but leads to memory becoming fragmented and memory utilization declining, with the holes between processes causing **external fragmentation**.

To overcome external fragmentation, use **compaction**: where the OS shifts processes so they are contiguous and all free memory is together in one block. However, this is time consuming, wasting processor time, and requires dynamic relocation capability.

Disadvantages are complexity to maintain and overhead of compaction.

2.2.1 Placement Algorithm

To avoid compaction, the OS designer must use an efficient algorithm to assign processes to memory.

Possible algorithms that choose among free blocks of main memory of appropriate size are:

- **best-fit**: chooses the block closes in size to the request
    - usually the worst performer as it leaves the smallest fragment behind
    - result is that main memory is littered with blocks too small to satisfy memory allocation requests so more compaction
- **first-fit**: scans memory from the beginning and chooses the first available block large enough
    - simplest and often best and fastest

    - may litter the front end with small free partitions that must be searched over on each first-fit pass

- **next-fit**: scans memory from the location of the last placement and chooses the next available block large enough
    - slightly worse than first-fit, as it more frequently leads to an allocation from a free block at the end of memory, so the largest block is quickly broken into small fragments leading to more compaction

2.2.2 Replacement Algorithm

When all processes in main memory are blocked, the OS swaps one process out for a process in a Ready-Suspend state.

2.3 Buddy System

Memory blocks are available as size 2^K works for $L \leq K \leq U$ where 2^L is the smallest size block allocated and 2^U is the largest size block allocated (size of the entire memory available).

Process works as follows:

1. Begin with the entire space available for allocation, treated as a single block of size 2^U.
2. If a request of size s such that $2^{U-1} < s \leq 2^U$ is made, the entire block is allocated.
    (a) Otherwise, the block is split into 2 equal buddies of size 2^{U-1}, with this process repeating until the smallest block $\geq s$ is generated and allocated to the request.

At any time, the buddy system maintains a list of holes of size 2^i. Whenever a pair of buddies on the i list is unallocated, they are removed from that list and coalesced into a single block on the $(i+1)$ list.

The following algorithm is used to find a hole of size 2^i:

```
void get_hole(int i)
{
    if (i == (U + 1)) <failure>;
    if (<i_list empty>) {
        get_hole(i + 1);
        <split hole into buddies>;
        <put buddies on i_list>;
    }
    <take first hole on i_list>;
}
```

In a binary tree representation of the buddy allocation system, the leaf nodes represent the current partitioning of the memory, and if two buddies are leaf nodes, at least one must be allocated, otherwise they are coalesced into a larger block.

Works as a reasonable compromise to overcome the disadvantages of fixed and dynamic partitioning (especially for parallel systems), but not as good as paging and segmentation.

2.4 Relocation

For some fixed partitioning, 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, so 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.

For equal-size partitioning, unequal-size partitioning (with a single queue), and dynamic partitioning, a process may be assigned to a different partition after being swapped out. Further, processes are shifted during compaction.

Locations referenced by a process are not fixed, so distinctions are made:

- **logical address**: a reference to a memory location independent of the current assignment of data to memory, translation must be made to a physical address before a memory access occurs
- **relative address**: address is expressed as a location relative to some known point, usually a value in a processor register
  - typically relative to the origin of the program
  - hardware mechanism needed to translate relative addresses to physical main memory addresses at the time of execution
- **physical address**: actual location in main memory

For relative addresses:

1. When a process is assigned to the Running state, a special processor register (base) is loaded with the starting address in main memory of the program.
    (a) A bounds register also indicates the end of the program.
2. When relative addresses are encountered, they go through 2 steps of manipulation by the processor:
    (a) the value in the base register is added to produce an absolute address
    (b) the resulting address is compared to the bounds register, and if it is within bounds the instruction proceeds, else an interrupt is generated to the OS

3 Paging

Pages: chunks of a process

Frames: available chunks of memory.

At any given point in time, some frames in memory are in use and some are free, with a list of free frames being maintained by the OS.

If there is enough contiguous unused space frames to hold a process, they are used. Otherwise, the OS maintains a **page table** for each process that shows the frame location for each page of the process.

The OS also maintains a free-frame list of all the frames in main memory that are currently unoccupied and available for pages.

Within the program, logical addresses consist of a page number and an offset within the page, and this translation is done by the hardware.

Relative and logical addresses are the same when the page size is a power of 2. Further this makes dynamic address translation at runtime easier for the hardware, as only the following steps are needed for address translation:

1. extract the page number from the logical address (leftmost n bits)
2. use the page number as an index into the process page table to find the frame number k

3. the starting physical address of the frame is $k \times 2^m$ and the physical address of the referenced byte is that number plus the offset (which is just appending that number to the offset)

4 Segmentation

A user program can be divided into variable-length segments (with a max length), and similar to paging logical addresses consist of a segment number and an offset.

Similar to dynamic partitioning, but with segmentation a program may occupy more than one partition and these partitions do not need to be contiguous.

Can still cause external fragmentation, though less than dynamic partitioning since a program can be broken into pieces.

Unlike paging, segmentation is visible to the programer, so modular programs can be broken down into multiple segments (within the size limitation).

No simple relation between logical and physical addresses (since unequal size segments). Each segment table entry must have the starting address of the segment in main memory and the length of the segment.

Procedure is similar to paging, except requires checking against size of segment and requires addition rather than appending numbers.