TP n°4: Management memory

1. Objectives

This practical work has the following objectives:

- Understand how the main memory allocation algorithms work,
- Familiarize yourself with the programming environment for memory allocation algorithms,
- Know the performance of memory allocation algorithms.

2. Management memory

Memory is an important resource that needs to be managed carefully. Even though the amount of memory in a computer has increased significantly, the size of programs is also increasing.

2.1 Memory and the memory manager

Memory is presented as a large array of words (bytes), each with its own address.

The CPU extracts instructions from memory according to the value of an instruction counter.

Memory manager system: part of the OS that manages the storage hierarchy.

- Tracks which parts of memory are used and which are not.
- Allocate/release memory space to processes.
- Control swapping between main memory and disk.

2.1.1 Logical and physical addresses

- Logical or virtual address: Address generated by the CPU. These are the addresses allocated after a program has been compiled and the process is seen to be alone.
- Physical address: Address seen by the memory unit.
- MMU (Memory Management Unit): Hardware device that converts virtual to physical addresses.

The user program never sees physical addresses; it processes logical addresses.

2.1.2 Basic types of management

Memory management systems can be divided into two groups:

- Those that move a process between disk and memory during execution.
- Those that do not (simpler).

2.2 Multiprogramming

Most modern OS allow multiple processes to run at the same time. When one process is stuck waiting for I/O, another can use the CPU.

2.2.1 Multi-programming with fixed partitions

The simplest way of multi-programming is to divide the memory into n partitions of fixed size where each partition can contain exactly 1 process.

SE maintains a table showing which parts of memory are available and which are occupied. Hole sets a block of available memory. So, since the partitions are fixed, any unused space is lost.

This partitioning mode is managed using two strategies (as shown in Figure 1):

- Fixed partition with different queues
- Fixed partition with a single different queue.

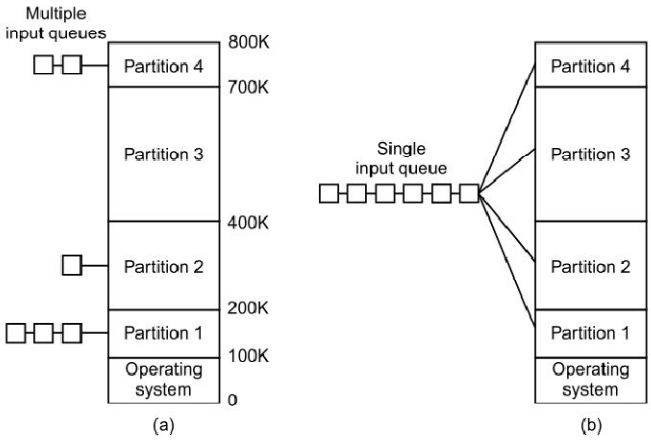

Figure 1: Multi-programming strategies with fixed partitions

Drawback of different queues: the queue for a large partition is empty, while the queue for a small partition is full. This can be solved with a single queue. However, the strategy with a single queue may lead to blocking if there is a large leading process followed by small processes which may be contained in available partitions (blocking queue problem).

2.2.2 Swapping

With time-sharing systems, sometimes the main memory is insufficient to keep all the current processes active: the additional processes have to be kept on a disk. In this case, a process is temporarily transferred from main memory to an auxiliary store and then returned to memory to continue execution. Auxiliary memory (backing store) is generally a fast disk large enough to store copies of all the memory images of all the users. Most swapping time consists of transfer time. The total transfer time is directly proportional to the amount of memory transferred.

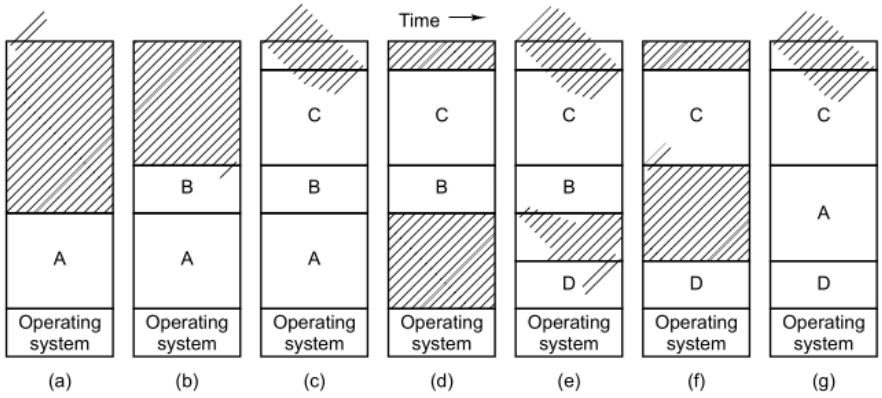

Figure 2: Swapping process

2.2.3 Multi-programming with variable partitions (dynamic partitioning)

The main difference compared with fixed partitions is that their number, location and size vary dynamically as processes come and go. In this memory allocation strategy, memory release and allocation are more complicated.

2.3 Memory allocation for a process

If the processes are created with a fixed size, the OS allocates exactly the memory required.

If the data segments of the processes have to expand, for example through dynamic allocation, a problem arises every time a program tries to expand:

- If a hole is adjacent to the process, that hole can be allocated to it.
- If the process is adjacent to another process, the growing process should be moved to a hole big enough for it.
- If the process cannot expand in memory and the swapping area is full, the process will have to wait or be killed.

It is therefore a good idea to anticipate that most processes will grow as they run, in which case it is a good idea to allocate a little extra memory each time a process is loaded or moved.

3. Memory allocation algorithms

Several memory allocation algorithms are possible, the most common being:

- **First-fit**: Allows the first hole to be large enough. The hole is then divided into two parts, one for the process and the other for unused memory, unless the process and the hole are the same size.
- **Next-fit**: Variation of First-fit. Starts looking for free space in the list from where it left off the previous time.
- **Best-fit**: Allocates the smallest hole big enough. Rather than breaking a hole, it tries to find a turn that matches the requested size. It is necessary to go through the whole list unless the list is sorted by size.
- **Worst-fit**: Allocates the largest hole (it scans the entire list).

4. Performance criteria of memory allocation algorithms

The performance of a memory allocation algorithm can be measured according to criteria such as:

- *External fragmentation*: there is enough total memory space to satisfy a request, but it is not contiguous.
- *Internal fragmentation*: the memory allocated may be slightly larger than the memory required. This difference is internal to a partition but is not used.

Exercises

Exercise n°1

1) In this exercise, you are asked to write a program in C that implements the concept of memory management using the "first fit" algorithm and calculates internal fragmentation.

   First-fit is executed using the following algorithm:

```
1. Begin
2. Declare the array size p[10] and b[10] where:
   - p: is an array containing the size of each process,
   - b: is an array containing the size of each process,
3. Read the number of processes;
4. Read the number of blocks;
5. Read the size of processes;
6. Calculate internal fragmentation;
7. Display the result with allocations;
8. End
```

2) Perform a trial run on the following example to check your program:

INPUT

- no of process: 3
- no of blocks: 3
- Size of each process:
  - Process 0: 100
  - Process 1: 150
  - Process 2: 200
- block sizes:
  - Block 0: 300
  - Block 1: 350
  - Block 2: 200

Exercise n°2

1) In this exercise, you are asked to write a program in C that implements the concept of memory management using the "Best fit" algorithm and calculates internal fragmentation.
2) Perform a trial run on the following example to check your program:

INPUT

- no of process: 3
- no of blocks: 3
- Size of each process:
  - Process 0: 100
  - Process 1: 150
  - Process 2: 200
- block sizes:
  - Block 0: 300
  - Block 1: 350
  - Block 2: 200

Exercise n°3

1) In this exercise, you are asked to write a program in C that implements the concept of memory management using the "Worst fit" algorithm and calculates internal fragmentation.
2) Perform a trial run on the following example to check your program:

INPUT

- no of process: 3
- no of blocks: 3
- Size of each process:
  - Process 0: 100
  - Process 1: 150
  - Process 2: 200
- block sizes:
  - Block 0: 300
  - Block 1: 350
  - Block 2: 200

Exercise n° 4

1) You are requested to implement paging concept for memory management in which you calculate the physical address from the logical address. Proceed as follows:

```
1. Start the program;
2. Enter the logical memory address;
3. Enter the page table which has offset and page frame;
4. The corresponding physical address can be calculated by:
```

$$@Physical = frame\ page\ no. * Size_{page} + offset$$

```
5. Print the physical address for the corresponding logical address;
6. End
```

2) Perform a trial run on the following example to check your program:

INPUT:

- Enter the memory size: 1000
- Enter the page size :100
- Enter no. of pages required for a process: 6
- Enter pagetable for a process: 1, 4, 2, 7, 3, 0
- Enter logical address.