**Project Report on Memory Management**

**Madhulika Bayyavarapu** **Radhika Jaju**

**INTRODUCTION:**

**Memory Management:**

The memory management function keeps track of the status of each memory location, either allocated or available. Memory manager allocates primary memory to processes, maps process address space to primary memory, minimizes access time using cost-effective memory configuration, also may use static or dynamic techniques. It determines how memory would be allocated among competing processes, deciding which gets memory, when they receive it, and how much they are allowed.

Basic function used to implement the project are:-

* **void \*malloc(size\_t size)**:- malloc() allocates memory block of given size (in bytes) and returns a pointer to the beginning of the block. malloc() doesn’t initialize the allocated memory. If we try to acess the content of memory block then we’ll get garbage values.
* **void free(void \*ptr):-** free() function deallocates the memory that is previously allocated using the malloc() function where the “\*ptr” points to the memory block that was allocated.
* **clock\_gettime():-** The *clock\_gettime()* function gets the current time of the clock specified by *clock\_id*, and puts it into the buffer pointed to by *tp*. The only supported clock ID is CLOCK\_REALTIME.

When memory is allocated it determines which memory locations will be assigned, and keeps track when memory is deallocated and updates the status. Several methods have been devised that increase the effectiveness of memory management. [Virtual memory](https://en.wikipedia.org/wiki/Virtual_memory) systems separate the memory addresses used by a process from actual physical addresses, allowing separation of processes and increasing the size of the [virtual address space](https://en.wikipedia.org/wiki/Virtual_address_space) beyond the available amount of [RAM](https://en.wikipedia.org/wiki/Random-access_memory) using [paging](https://en.wikipedia.org/wiki/Paging) or swapping to [secondary storage](https://en.wikipedia.org/wiki/Secondary_storage). The quality of the virtual memory manager can have an extensive effect on overall system performance.

Memory manager binds instruction and data to memory and brings program into memory and places within a process for it to be run. Address binding of instructions and data to memory addresses can happen at three different stages.

1) **Compile time**: If memory location known a priori, absolute code can be generated.

2) **Load time**: Must generate relocatable code if memory location is not known at compile time.

3) **Execution time**: Binding delayed until run time if the process can be moved during its execution from one memory segment to another.

Contiguous Allocation Main memory usually into two partitions:

1. Single-partition allocation:- Relocation-register scheme is used to protect user processes from each other, and from changing operating-system code and data.
2. Multiple-partition allocation:- Holes, blocks of available memory of various size are scattered throughout memory. When a process arrives, it is allocated memory from a hole large enough to accommodate it. Operating system maintains information about: a) allocated partitions b) free partitions (hole)

Each program must be loaded to main memory before it can execute. Allow for multi-programming where multiple programs have to share memory. There are 3 approaches to structuring and allocating memory:

1. Fixed Partitions

2. Variable Partitions

3. The Buddy System

Dynamic Storage Allocation Mechanisms:

1. **First-fit:** In the first fit approach the first free partition or hole large enough which can accommodate the process is allocated.

**Pro:** It is the fastest algorithm as it searches as little as possible.

**Cons:** If there is any other partition which would fit the request more accurately in the latter parts of memory which aren’t used leads to internal fragmentation.

1. **Best-fit:** Allocate the smallest hole that is big enough; must search entire list, unless ordered by size. Produces the smallest leftover hole.

**Pro:** Memory utilization is much better than First-fit approach as it searches for the smallest free partition large enough for the request first available.

**Cons:** It is slower and may even tend to fill up memory with tiny holes which cant be allocated to any arriving requests due to their small size.

1. **Worst-fit:** Allocate the largest hole; must also search entire list. Produces the largest leftover hole.

**Pro:** It reduces the rate of formation of holes compared to best-fit and the holes resulting due to internal fragmentation can potentially be allocated to an arriving request as they wouldn’t be too small.

**Cons:** If a process requiring larger memory chunk arrives at a latter stage then it can’t be accommodated as the largest hole is already occupied and split.

1. **Next-Fit:** Next fit is a modified version of first fit. It begins as first fit to find a free partition. When called next time it starts searching from where it left off, not from the beginning.

**Pro:-** Memory is used more efficiently and evenly as a free partition is searched from the latest allocated memory address rather than the beginning of the list.

**Cons:**-This method suffers External Fragmentation.

**Algorithm:**

1)      A set of 50 processes with the mentioned memory and cycles requirements are generated by using random function.

2)      Three different cases are given in the project:

**CASE-1:-**In the first case, malloc() and free() predefined library functions are used.  After every 50 cycles a process enters the system and malloc() function will be called each time to dynamically allocate the required memory to the process. After the process completes its execution, free() function is used to deallocate the memory given to that process.

**CASE-2:-**In the second case, initially a predefined memory block of 100MB is created and a new process arrives at every 50 cycles. Then, my\_malloc\_dynamic() function is used to perform the dynamic partitioning of that 100MB allocated memory. Whenever a new process arrives in the system, the my\_malloc\_dynamic(int elem\_length) logic will search for chunk of memory that is most suitable for the process and allocates it to the process. This logic is implemented with best-fit approach. So, every time a process arrives, it is checked from the location where it left off, if there is any memory chunk that is free and can be allotted for the arriving process. Once a process completes its execution, the memory allocated to it is deallocated and made available for the arriving processes using my\_free\_dynamic(void \*p).

**CASE-3:-**In the third case, similar to the first two cases the processes arrive each for every 50 cycles. A pre-defined memory of 100MB is created. The 100MB allocated memory is divided into 20 chunks of 20MB memory as we are implementing static partitioning in this case using my\_malloc\_static(int length). In this case memory is allocated using next fit scheme, i-e whenever a process occurs, it is checked if there is any enough memory to serve the request of the arriving process from the location where it left off, in the 100MB memory chunk. After the completion of execution of every process its memory is deallocated and made available for the other arriving processes using my\_free\_static(int process, int memory) function.

3) The time taken for the implementation i-e for allocating memory, executing the processes and deallocating the memory after completion of the process is calculated for all 50 processes in all the three cases using clock\_gettime() function.

**Screenshots of output:-**

**A screen shot of a computer

Description generated with very high confidence**

Fig 1**:** output displaying the processes created with the memory and cycles details

A picture containing text

Description generated with high confidence

Fig 2: Output of case-1 displaying the time taken to complete the execution.

A screenshot of a computer

Description generated with high confidence

Fig 3: Output of case-2 displaying the time taken to complete the execution.

A screenshot of a computer

Description generated with high confidence

Fig 4: Output of case-3 displaying the time taken to complete the execution.

A screenshot of a computer

Description generated with very high confidence

Fig 5: Time taken for 3 cases for different runs.

A close up of text on a white background

Description generated with very high confidence

Fig 6: Analysis of the time taken for 3 cases for different runs (shown in the fig. 5)

**Analysis:** In most of the cases the time taken to complete the execution is decreasing from case-1 to case2 to case-3. This is because we are allocating, deallocating memory to every process each time it arrives which leads to context switching to a great extent for the case-1. In case-2 we are allocating an entire chunk of memory of 100MB with just one malloc() call and later using my\_malloc\_dynamic() function to allocate memory to implement dynamic partitioning, hence we decreased the context switching time in this case. In case-3 we allocated the memory of 100MB into 20 static partitions each of 5MB and used my\_malloc\_static(int length) to implement allocation of memory to the processes. As we created partitions at the beginning which is fixed time consumed in this case would decrease further.

**Note:** In some cases, the time taken disn’t come out as expected. This could happen because of various reasons like if the processes take maximum memory chunk in one of the cases and lesser memory in other cases, the time taken for the execution ould vary as the process occupying larger memory has to free the memory to accommodate the arriving process else it’d be waiting which increases the time taken for the execution.