Analysis:

We are asked to build a custom Memory Manager which does the house keeping of allocation and de allocation of memory on an initial chunk of 100 MB data. Also, given are the following:

- 50 randomly generated processes with memory requirement between (10 KB, 2 MB)
- Number of cycles per each process in between (200 and 2500 cycles).
- A new process arrives every 50 cycles.

Initial questions:

- Do we need to perform compaction for Dynamic partitioning? (For given simulation, it is not required as each process has a maximum memory request of 2 MB and we are initially requesting 100 MB. 50 * 2 MB is at most 100 MB).
- If we are to compact our dynamic partitions, how often should it be done?
    - After each process is freed?
    - After each new process is allocated?
    - At periodic random intervals (as done in commercial operating systems?)
    - If requested memory is not available?
- How should holes be allocated?
    - First – Fit?
    - Best – Fit?
    - Or Other Memory allocation strategies?
- What if multiple processes request memory at same time? (Need for synchronization mechanisms?).

Implementation of Memory Manager

To implement our Memory Manager, we need to set up our data structures first.

1) **Process:**

Implemented in "**Process.h"** and "**Process.cpp"** files. A process has following fields:

i) **ID**: Each process has an own unique ID.
ii) **Memory Size**: Each process has its own memory requirement.
iii) **Cycles**: Number of cycles a process will be in the system.
iv) **Data**: The actual data allocated to each process.
v) **Current Cycles:** Indicates number of cycles the process has been in the system.

The member functions in the **Process** data structure include the getters and setters of each of the private variables.

2) **Hole:**

Implemented in "**Hole.h"**. A Hole data structure has the following fields:

i) **Starting Address:** Denotes the start of the Hole.
ii) **Ending Address:** Denotes the end boundary of the Hole.
iii) **Size:** Indicates the size of the hole.

3) Memory Manager:

- **i) Data:** stores the initial memory request of 100 MB.
- **ii) Current Address:** The current pointer useful in our allocation algorithms.
- **iii) Initial Address:** The initial address of data requested. (100 MB)
- **iv) Holes:** A dynamic vector array which stores the list of holes produced during our allocation.
- **v) Fixed Partitions:** A vector array storing the starting addresses of fixed partitions (0MB, 5 MB, 10 MB, …..) which is useful in static partitioning.

The Memory Manager implemented in **MemoryManager.cpp** file performs the crux of our logic. Let's look at some of the functions defined in the above file.

Constructor: -

It performs the following functions.

- Requests 100 MB data.
- Stores the initial address and sets the current pointer to that address.
- Populates the fixed partition array in case of static partitioning (if Flag is true)

Dynamic Allocation and Free: -

Performs the memory allocation similar to malloc() and de-allocation similar to free() but in user space. The algorithm is as follows:

- Initially, the total number of holes are 0.
- Search for any hole in our data structure, If any hole is available which can accommodate a process of given size (***FIRST – FIT***), allocate process to the hole and return the pointer.
- If the process is assigned to the hole, the size and addresses of the hole are updated.
- Else, we go the current pointer and check if we reached the end of 100 MB data.
- If we are well within the boundaries, then return the current pointer to the requesting process.
- Increment the current pointer with the size requested.

While freeing the leaving process, simply creates a new hole which we can allocate to other processes.

Compaction in Dynamic partitioning: -

There are several options on deciding when to compact (to counter External Fragmentation). We decided to compact when our initial chunk of 100 MB is at least $3/4^{th}$ full. The algorithm for compaction is simple which scans through all the holes in our system and checks for neighboring holes. If any neighboring hole is present, we coalesce.

Static allocation and Free: -

The static partitioning algorithm is similar to that of dynamic partitioning except for the fact that, we are now working with fixed partitioning of size 5 MB. The algorithm is as follows:

- Scan the holes if any process can fit in it. If not, assign the current pointer memory to the requesting process.
- This leaves a hole of size (5 MB – process request). Now we can move the current pointer by 5 MB.
- If we cannot fit the process either in the hole or at the current pointer, we return **NULL** indicating memory cannot be allocated.

When freeing we check if we can coalesce with neighboring holes. Three cases arise in such situation:

Case 1: (A Hole followed by Process, followed by Hole)

We combine 3 of them into single hole and remove the other holes.

Case 2: (A Process followed by Hole)

We coalesce into single hole starting at process memory address and ending at Hole's ending address.

Case 3: (A Hole followed by the Process)

We simply update the size and ending address of the hole.

If either of the 3 cases are not satisfied, then the leaving process simply leaves a hole of its size.

SIMULATOR DESIGN

Implemented in **Main.cpp** file. Our simulator creates random processes within specified ranges. We then, time our simulation process during the call of dynamic partitioning, static partitioning and system allocation methods.

The memoryManager() function inside the Main.cpp forms the core of our simulator which does the following:

i) If flag =1, the function times the system malloc () and free.
ii) If flag = 2, the function times the dynamic partitioning.
iii) If flag = 3, the function times the static partitioning.

We loop till we have no active processes or waiting processes or till our process array is empty:

- We request memory for process which arrives every 50 cycles.
- If memory cannot be allocated, we add it to waiting process array.
- Else, we assign the data to the process.
- Now, the process is active and runs to its completion.
- We then, increment the number of cycles.

- It is to be noted that, we check if we can allocate memory to any of our waiting processes every cycle. This functionality is accomplished by **allocateMemoryForWaitingProcesses().**
- Also, we update the number of cycles for all active processes.
- We now, need to free the memory for the processes which has completed the required number of cycles.

We perform this simulation algorithm for 50 iterations and take average of the execution times to get more accurate execution times.

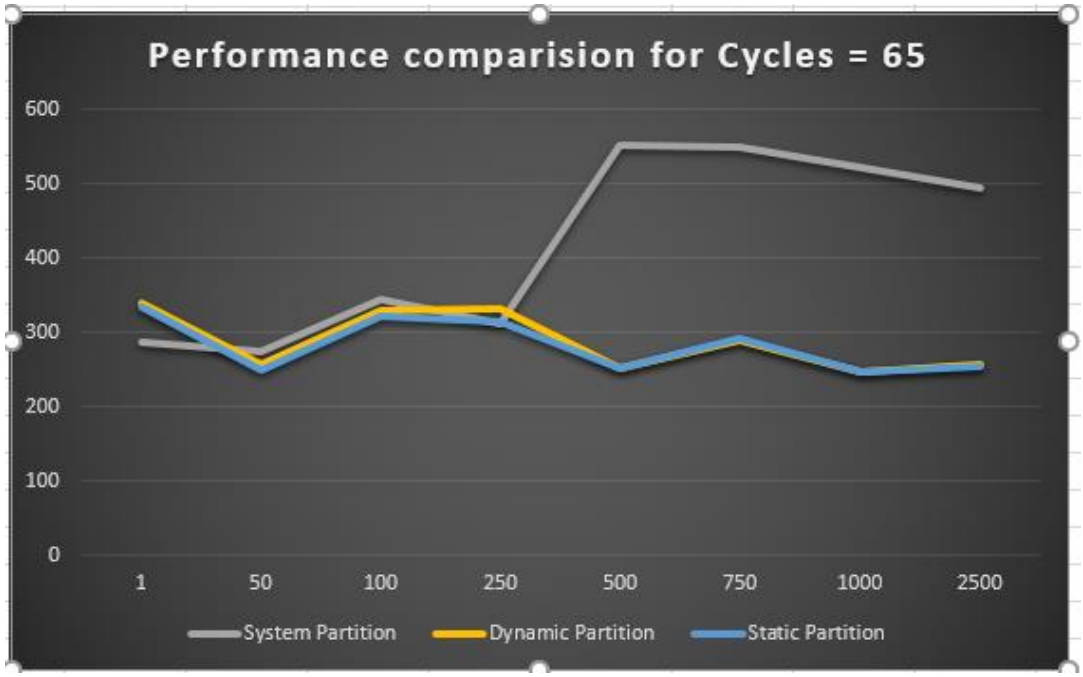
Experiments and Results

a) **Comparison with fixed Cycles for each process**: -

X – axis (Memory size of each process in KB)

Y- axis (Time taken in micro seconds)

Number of cycles for each process is fixed to be 65, each simulation is done 50 times and average is taken. The number of processes is assumed to be 50.



When the memory size is less than 100 KB, we see that system malloc() is performing better than their counter parts. However, once the memory size increases above 100 KB, system malloc() is worse compared to dynamic_alloc and static_alloc.

Initially, we see that dynamic allocation is slower than static allocation. This is because holes are not created in dynamic partitioning until processes are freed. (Remember, we are allocating process to

large enough hole first) On the contrary, static partitioning starts creating a hole right after allocating a single process. However, as the size increases, holes in static partitioning start becoming smaller and smaller. Hence, we see that dynamic partitioning soon catches up with fixed partitioning.

b) Comparison with fixed Memory Size: -

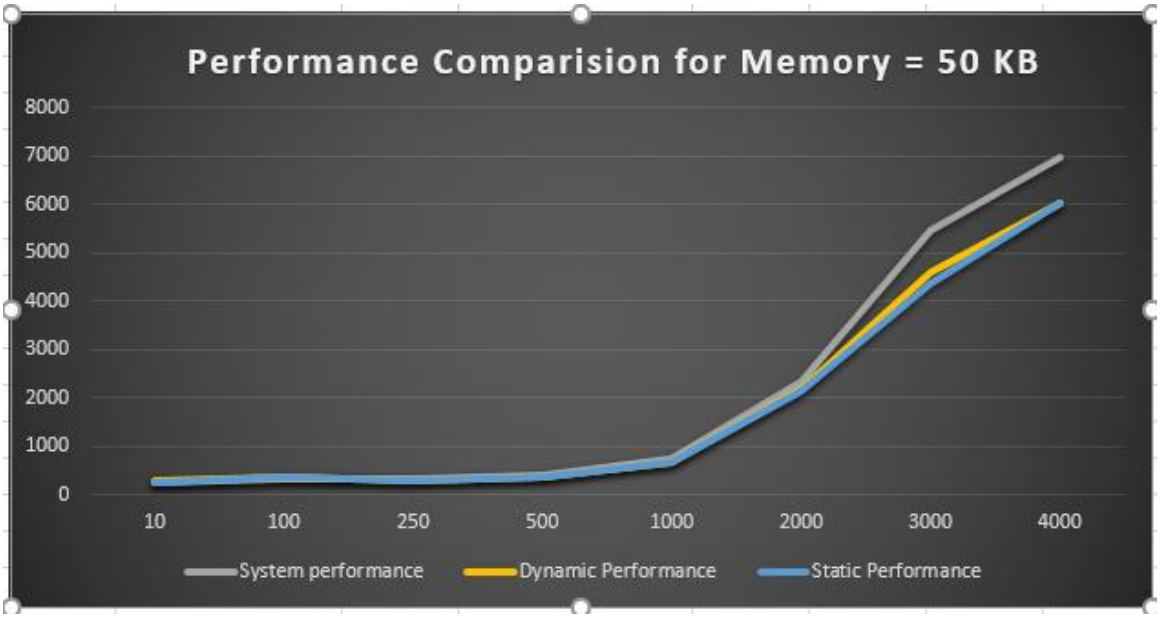
X – axis (Number of Cycles for each process)

Y – axis (Time taken in Micro Seconds).

Number of Processes = 50

Number of Iterations = 50

Memory requirement of each process = 50 KB;



We see that initially, system malloc() is better than it's counter parts. This is because the simulation assumes each process to be arriving at 50 Cycle intervals. So, we are waiting for 50 cycles for new process to arrive in all the three memory management regimes. Also, if number of cycles is below a certain threshold, we are forming more holes. (Remember that both static and dynamic searches for large enough hole first). This causes system malloc() to perform better. However, as we increase number of cycles we can see a clear change in the performance.

C) **Performance comparison with fixed cycles and memory sizes:**

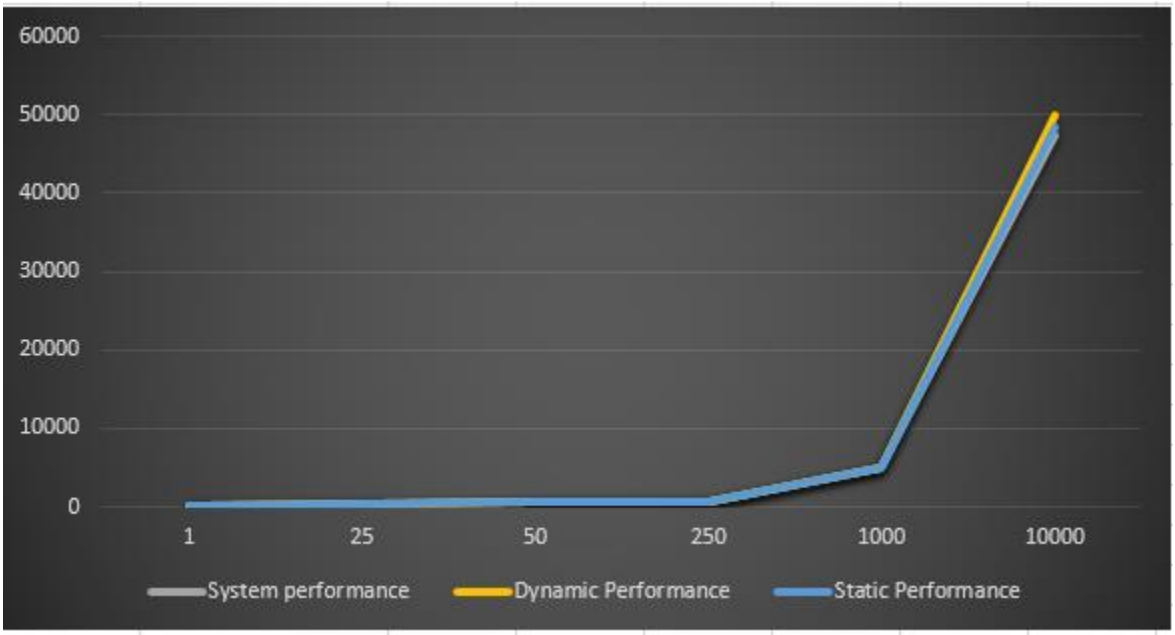
X – axis (Number of Processes simulated)

Y – axis (Time taken in Micro Seconds).

Number of cycles = 2650

Memory size for each process = 50 KB.

Number of Iterations = 50



In the above experiment, we do not see as much difference in timings as we did in the above two experiments. However, it is still significant in the order of ***Nano seconds***. This can be attributed to more number of processes resulting in more number of holes which in turn need more time to coalesce. We see that static partitioning is out performing its counter parts as it doesn't have to coalesce across partitions.

Random Tests

Performance comparison when all processes arrive at once:

In this case, we are potentially running into a process synchronization problem. We need to synchronize our malloc() and free() methods to avoid system inconsistency. For this reason, we implemented mutex locks in the **MemoryManager.cpp** file. The results for this case (with memory 1000 KB and 650 Cycles) is satisfactory as we achieved significant improvement in my_malloc() performance.

LIMITATIONS:

#1: We cannot accommodate any process whose size is more than 100 MB. This is because we are initially requesting only 100 MB chunks of data.

#2: For static partitioning, we cannot fit any process whose size is greater than 5 MB. As for memory is partitioned into fixed chunks of 5 MB each.

#3: If memory sizes, number of cycles and processes are very small, we can experience error prone timing results as we are dealing with Nano – second timers.

ASSUMPTIONS:

#1: We assume that processes are not interrupted during simulations. (In reality, traps set by OS and interrupt requests from other processes cause the active process to go into waiting state).

#2: The system has enough resources to allocate 100 MB in each iteration of our simulation (which eventually will be freed of course!).

#3: The system malloc produces 100 MB contiguous memory in logical address space. If this is not the case, then our entire simulation would fall apart.

Results:

Here are a few sample results based on the requirements specified in the project document.

INITIAL SET OF PROCESSES:

PROCESS: 1 MEMORY: 504KB CYCLES:1155

PROCESS: 2 MEMORY: 1MB CYCLES:1680

PROCESS: 3 MEMORY: 1MB CYCLES:802

PROCESS: 4 MEMORY: 1MB CYCLES:2178

PROCESS: 5 MEMORY: 623KB CYCLES:464

PROCESS: 6 MEMORY: 1013KB CYCLES:1230

PROCESS: 7 MEMORY: 1MB CYCLES:1692

PROCESS: 8 MEMORY: 1MB CYCLES:1571

PROCESS: 9 MEMORY: 1MB CYCLES:1042

PROCESS: 10 MEMORY: 334KB CYCLES:882

PROCESS: 11 MEMORY: 1MB CYCLES:740

PROCESS: 12 MEMORY: 450KB CYCLES:1679

PROCESS: 13 MEMORY: 554KB CYCLES:1361

PROCESS: 14 MEMORY: 534KB CYCLES:1866

PROCESS: 15 MEMORY: 1MB CYCLES:1731

PROCESS: 16 MEMORY: 867KB CYCLES:790

PROCESS: 17 MEMORY: 100KB CYCLES:2238

PROCESS: 18 MEMORY: 1MB CYCLES:287

PROCESS: 19 MEMORY: 1MB CYCLES:2361

PROCESS: 20 MEMORY: 1MB CYCLES:767

PROCESS: 21 MEMORY: 1MB CYCLES:1162

PROCESS: 22 MEMORY: 1MB CYCLES:1984

PROCESS: 23 MEMORY: 533KB CYCLES:1579

PROCESS: 24 MEMORY: 783KB CYCLES:228

PROCESS: 25 MEMORY: 1MB CYCLES:1032

PROCESS: 26 MEMORY: 1MB CYCLES:1497

PROCESS: 27 MEMORY: 892KB CYCLES:1327

PROCESS: 28 MEMORY: 1MB CYCLES:634

PROCESS: 29 MEMORY: 302KB CYCLES:2366

PROCESS: 30 MEMORY: 1MB CYCLES:698

PROCESS: 31 MEMORY: 1MB CYCLES:235

PROCESS: 32 MEMORY: 1MB CYCLES:552

PROCESS: 33 MEMORY: 1MB CYCLES:2128

PROCESS: 34 MEMORY: 801KB CYCLES:1382

PROCESS: 35 MEMORY: 96KB CYCLES:1474

PROCESS: 36 MEMORY: 1MB CYCLES:881

PROCESS: 37 MEMORY: 1MB CYCLES:2316

PROCESS: 38 MEMORY: 1MB CYCLES:824

PROCESS: 39 MEMORY: 1MB CYCLES:2289

PROCESS: 40 MEMORY: 444KB CYCLES:2384

PROCESS: 41 MEMORY: 1MB CYCLES:2098

PROCESS: 42 MEMORY: 761KB CYCLES:961

PROCESS: 43 MEMORY: 1MB CYCLES:2391

PROCESS: 44 MEMORY: 764KB CYCLES:1447

PROCESS: 45 MEMORY: 287KB CYCLES:1731

PROCESS: 46 MEMORY: 1MB CYCLES:2423

PROCESS: 47 MEMORY: 1MB CYCLES:1802

PROCESS: 48 MEMORY: 310KB CYCLES:307

PROCESS: 49 MEMORY: 171KB CYCLES:1448

PROCESS: 50 MEMORY: 176KB CYCLES:1986

COLLECTING STATISTICS:

.....

SIMULATION REPORT

/***/

NUMBER OF SIMULATIONS PERFORMED #50

TOTAL NUMBER OF PROCESSES PER SIMULATION: 50

AVERAGE MEMORY SIZE PER SIMULATION: 50.5876 MB.

AVERAGE NUMBER OF CYCLES REQUIRED PER SIMULATION: 2469 CYCLES.

AVERAGE SYSTEM PARTITION PERFORMANCE: 2215 MICRO SEC

AVERAGE DYNAMIC PARTITION PERFORMANCE: 1789 MICRO SEC

AVERAGE STATIC PARTITION PERFORMANCE: 1663 MICRO SEC

/***/