**Implementing Optimal, Paging, Best Fit, and Worst Fit Memory Allocation Policies in an Emulated OS Environment**

**Jase Bell and Evan Sheehan**

**Lab Report #3**

**COMP 3500**

**Professor Saad Biaz**

**Auburn University**

**1 Abstract**

In an operating system that supports multiprogramming, processes are continually in need of being allocated memory. For multiprogramming to create an efficient and responsive environment, the long-term scheduler needs to make smart decisions with regards to how and where processes should be allocating memory. This report surveys four memory allocation policies applied to a simulated program that mimics the relationship between processes and the long-term scheduler in a true operating system. The four memory allocation policies implemented into this system are Optimal Memory Allocation Policy (OMAP), Paging, Best-Fit, and Worst-Fit. Metrics were gathered for each algorithm and compared. The metrics that we collected to compare these algorithms were Average Turnaround Time, Average Response Time, Average Waiting Time, CPU Utilization, Throughput, and Average Time in Job Queue. The best policy was Optimal Memory Allocation Policy (OMAP). The only downside is that it is purely theoretical and cannot be implemented and utilized in real world scenarios. Best-Fit, Worst-Fit, and Paging at 256 bytes all did very well and can actually be implemented in real world scenarios. We must take into account that this lab is a simulation and not representative of how these algorithms would typically perform. These policies all performed better than normal due to not having accurate representations of the costs of memory compaction and reallocation. We concluded that further testing would be needed that took this into account.

**2 Introduction**

For the purpose of this lab, the majority of the work was done for us. The only areas that we were responsible for were the Dispatcher, Bookkeeping, LongtermScheduler, and a function we created ourselves called getStartAddress. getStartAddress is where we implemented the four memory allocation policies that the lab is focused on. The reason for not having us handle sections like the I/O operations and the Short Term Scheduler was to have us focus on understanding the memory allocation policies that the long term scheduler might employ and how they could be successfully and efficiently implemented. Bookkeeping was merely for us to be able to generate and record the data on the six metrics we used. This data allowed us to analyze and compare the advantages and disadvantages of each memory allocation algorithm we implemented. The Dispatcher forced us to be responsible for reallocating and reorganizing memory of processes that entered the exit queue. The long-term scheduler was where we had to calculate our Waiting Time in Job Queue metric and check whether or not memory allocation requests from processes can be granted. Overall, this lab required in-depth knowledge and comprehension of all four memory allocation algorithms, the metrics gathered for each, as well as how the processes are allocated and deallocated memory during their lifetimes. Every bit of this knowledge was crucial for success. Below we have organized our report into a body, results, and conclusion section. In the body section, we will detail the meaning of each metric, function, and algorithm that we utilized while implementing our solutions. This includes all the scheduler information and process control block information that was helpful in executing the logic required for each algorithm. In the results section, we comprehensively analyze the algorithms and with respect to each individual metric that we computed and compare them against each other. Our conclusion then gives an overall ranking of each memory allocation policy by taking into account our data we collected and known knowledge about how each policy.

**3 Body**

**3.1 Processesgenerator.o**

Processesgenerator is a C object file provided by the instructor. The data of processesgenerator is obfuscated to prevent the addition of unneeded complexity. The function of this object file is to simulate the generation of processes to be handled by the ProcessesManagement2.c file.

**3.2 Common2.h**

Common.h is a C header file which holds the declarations of functions and variables used in processes management. The relevant contents of this file include the structure for a ProcessControlBlock object, but also include instructor-implemented functions such as EnqueueProcess and DequeueProcess.

**3.3 ProcessesManagement2.c**

The process management file is the meat of the CPU scheduling implementation. This section gives a brief summary of each of the methods used in ProcessesManagement2.c and section 3.4 linearly explores how these functions utilize one another to emulate the function of a CPU scheduling policy.

**3.4 Program Overview**

**3.4.1 Main**

The main function is the function first run when ProcessesManagement2.c is called. This function takes two arguments: int argc and char \*\*argv. These parameters are used to check for command line arguments by running the built-in C function ‘Initialization’ and passing the previous parameters. If the ‘Initialization’ function finds that there are parameters, which are necessary in order to select which scheduler to run, the main function continues and runs the ‘ManageProcesses’ function.

**3.4.2 ManageProcesses**

The ‘ManageProcesses’ function is called by the main function at the beginning of the ProcessesManagement2.c execution cycle. The function returns void and takes void as a parameter, that is to say it does not take any parameters. ‘ManageProcesses’ first calls the ‘ManagementInitialization’ method once and then runs the ‘IO’, ‘CPUScheduler’, and ‘Dispatcher’ functions in a constant loop.

**3.4.3 IO**

The ‘IO’ function is a function ran within ‘ManageProcesses’ that is meant to emulate an I/O request event. This function takes no parameters and returns void. The IO function dequeues processes from the running queue, sets their state to waiting, simulates an IO burst, then waits to be put back onto the ready queue. The second part of this function scans the waiting queue for processes that have completed their IO burst. Processes that are waiting and have completed their IO burst are put back onto the ready queue.

**3.4.4 CPUScheduler**

‘CPUScheduler’ is a function that selects which CPU scheduling policy to run and puts the process returned by that policy on the running queue. ‘CPUScheduler’ is executed by ‘ManageProcesses.’ It takes a policy number as a parameter and returns void. The policy number is an int that tells the CPU scheduler which CPU scheduling policy needs to be run. A switch data structure is used to select which policy to run, and that policy is executed. The returned result, which is a ‘ProcessControlBlock’ object pointer, is then put in the running queue, and the state of the process control block is set to running.

**3.4.6 Dispatcher**

‘Dispatcher’ is a function that returns void and takes no parameters. Dispatcher is the workhorse of PocessesManagement2.c and handles the processes that are put on the running queue. ‘Dispatcher’ is called in the ‘ManageProcesses’ function after ‘CPUScheduler’ and dequeues the first process from the running queue. The dispatcher then checks to see if this process needs more work done by comparing the processes’ ‘TimeInCpu’ to its ‘TotalJobDuration.’ If the process doesn’t need more work, it is put in the exit queue and the necessary metrics are updated.

If more work is needed and the process is being put on the CPU for the first time, the necessary metrics are updated. If more work is needed, and round robin is being run, the ‘CPUBurstTime’ is set to ‘Quantum.’ If more work is needed, and the remaining CPU burst time is less than the total, the total CPU burst time is updated to the remaining. After the conditional checks, the process is put on the CPU using the ‘OnCPU’ function. The process is then placed on the running queue and ‘Dispatcher’ finishes executing.

**3.4.7 NewJobIn**

‘NewJobIn’ is a function which returns void and takes in a ‘ProcessControlBlock’ object as a parameter. This function is run whenever a job is added to the Job Queue. NewJobIn is mainly concerned with allocating memory for the recently-admitted process and putting that process on the job queue. It also calls the DisplayQueue function located in common.h and executes the LongtermScheduler function.

**3.4.8 BookKeeping**

‘BookKeeping’ is a function that is called automatically when 250 processes arrive on the job queue. This function computes and prints the final metrics gathered during runtime. This includes calculations for average turnaround time (TAT), average response time (RT), CPU busy time/CPU utilization (CBT), throughput (THGT), average waiting time (WT), and average waiting time in job queue (AWTJQ). Because bookkeeping is the final process to run, it exits the program once the calculations are completed and printed.

**3.4.9 LongtermScheduler**

‘LongTermScheduler’ is a function that returns void and takes void as a parameter.  It is called at the end of the ‘NewJobIn’ function. In accordance with its name, this function acts as a long term scheduler for the ProcessesManagement2.c file. ‘LongtermScheduler’ dequeues the first process from the job queue, updates the necessary metrics such as ‘TimeInJobQueue’ and‘JobStartTime’, and places the process onto the ready queue, setting its state to ready. This process repeats until the program ends or the job queue empties.

**3.4.10 ManagementInitialization**

‘ManagementInitialization’ is the first function run by ‘ManageProcesses.’ It returns a flag, always true in this case, and takes no parameters. This function simply runs through the ‘NumberofJobs’ and ‘SumMetrics’ and sets each value to zero based on the constant assigned to ‘MAXMEXTRICS.’

**3.5 Process Control Block**

A process is a program that is active and making use of the CPU and memory. Each Process contains a Process Control Block, or PCB. The Process Control Block keeps track of important information for the process. This is important because this information can be reloaded, and the process resumed after a break that stops the process such as I/O operations or CPU preemptions. After this information is reloaded the process can begin running successfully as before. We utilize a simulated Process Control Block in the common2.h program file for this lab.

**3.6.1 What is new? Memory attributes for PCB**

There are three attributed from the Process Control Block (PCB) that we did not utilize in lab 1 that are relevant to lab 3. These are TopOfMemory, MemoryAllocated, and MemoryRequested. TopOfMemory is the address of the top of the allocated memory block for each process. MemoryAllocated is the amount of allocated memory for each process given in bytes. This is a variable that we are responsible for keeping track of during program execution. MemoryRequested is the amount of requested memory for each process given in bytes. This value is generated automatically by ProcessesGenerator.o and is a constant for each process admitted into the system.

**3.7 Metrics Overview**

During our program we collect data to compute our metric results. These metrics are Average Response Time (ART), Average Turnaround Time (ATAT), Average Waiting Time (AWT), Computer Busy Time (CBT), Throughput (TGHT), and most importantly Average Waiting Time in Job Queue (AWTJQ). We define Average Response Time as the average time the process was first placed on the cpu and entered the running state minus the JobArrivalTime for each process. We define Turnaround Time as the average time the process completed and was placed in the exit queue and done state minus the JobArrivalTime for each process. Average Waiting Time is defined as the sum of the total time spent in the ready state by each process divided by the total number of processes. Computer Busy Time is a percentage defined as the total time spent on the cpu by all the processes up until the current time divided by the system time. Throughput is a measure of productivity is defined as the number of jobs completed divided by the system time. Average Time in Job Queue is defined as the average time the process enters the ready queue minus the JobArrivalTime for each process. All of these metrics are constantly changing. Our results are the current values for these metrics at the point when 250 processes have arrived.

**3.8 Memory Management Concepts**

Memory management involves focusing on two core principles. These principles are maximizing the number of processes in the memory and minimizing the access time to memory. The reason that these two principles are important is that they help us to satisfy what the programmer or user wants. The user wants a flat, fast, inexpensive, and infinite memory space that starts at address 0. The user is motivated by convenience and simplicity. The goal of the Operating System is to create the illusion that the memory space is infinite by deallocating and reallocating memory to processes as needed. The Operating System partners with a piece of hardware called the Memory Management Unit (MMU) to manage the physical space. The flat, infinite space that the user sees is known as the logical space. The mapping of processes logical address to its allocated physical address is done using dynamic translation and page tables.

**3.8.1 Process Placement in Physical Space**

The methodology of how processes are placed in the physical space is determined by the memory allocation policy used. The memory allocation policies that we used in this lab and will detail below are Optimal Memory Allocation Policy (OMAP), Paging, Best-Fit, and Worst-Fit. These policies can be subdivided by whether or not they are contiguous or noncontiguous. Contiguous policies require that a process be allocated to one single chunk of memory while non-contiguous policies allow for processes to be allocated to multiple chunks of memory scattered throughout the physical space. In either case, these chunks of available memory are known as holes. The Operating System must keep track of which sections of the physical space are allocated to processes and also the holes of available memory.

**3.8.2 Fragmentation**

Fragmentation is when holes become too small to be utilized by incoming processes arriving in the job queue. Fragmentation can be subdivided into two categories. These categories are known as Internal Fragmentation and External Fragmentation. External Fragmentation occurs when the total memory space exists to satisfy a process’s memory allocation request, but the memory is not contiguous. This always happens when variable partitioning is used. Internal Fragmentation is also where the where the available memory can satisfy the request, but the memory not being utilized is internal to partitions of the physical space usually called frames. This type of fragmentation always occurs when using fixed partitioning of the physical space.

**3.9 Algorithms**

In this lab we are simulating four memory allocation policies. Each policy or algorithm has its own pros and cons. The memory allocation policies Optimal, Paging, Best-fit, and Worst-fit are described in detail below.

**3.9.1 Optimal (OMAP)**

The Optimal Memory Allocation Policy (OMAP) is the absolute best memory allocation policy in terms of Average Waiting Time in Job Queue (AWTJQ). This should be expected from its name. This algorithm is very simple. AvailableMemory is a variable that is constantly updated in this simulation. If a process requests to be allocated a certain amount of bytes and that number is less than the AvailableMemory then the request is granted and the memory allocated to the process. This algorithm is purely theoretical and does not have to take into account practicalities of managing memory holes, memory fragmentation, or techniques to alleviate these issues such as memory compaction. Such an algorithm has no practical purpose since it cannot be implemented in real world applications. The reason for this is that there is no way to know the value of AvailableMemory automatically without memory management.

**3.9.2 Paging**

Paging is a noncontiguous memory allocation policy. This means that the logical address space of a process can be divided. In this case we divide the logical address space into pieces called pages. The physical memory is divided in much the same way with blocks of memory the same size as pages called frames. Processes request to be allocated a certain amount of bytes of memory. This request is granted if there are enough available frames totaling equal to or more than the number of bytes requested. Paging does not care if these frames are scattered around the physical memory since it is noncontiguous. Memory Management is needed for this memory allocation policy to keep track of all available frames. Normally, this is implemented using page tables. However, for the purpose of this lab we have simplified this process. As long as there are frames available in physical memory then we can assign any page to any frame. For this reason, our implementation of paging detailed below will not completely correlate to how paging is implemented in real world scenarios.

There are several benefits to the paging memory allocation policy. The first is that this policy allows for increased flexibility to place processes. This is because the memory does not have to be contiguous and only relies on their being enough available frames for the process. Another benefit is that external fragmentation is eliminated. With all of the physical memory being divided into frames of equal size, there is no chance of their being enough frames for a process and no hole big enough to place the process. Also, internal fragmentation is limited. The size of a process is of course not always equal to a multiple of the page size. This means that the last page allocated to a process will not be filled and will have internal fragmentation. However, it is limited to only the last page.

There are also drawbacks to the paging memory allocation policy. The first drawback is that each data access requires two memory accesses. The reason for this is that you must first access the page table and then access the data using the physical address obtained from the page table. The second drawback is the space overhead that is required for maintaining the page table. However, in our simulation we do not encounter either of these drawbacks since we are not implementing page tables.

In this lab we will explore the paging algorithm with varying page sizes. The two page sizes that are utilized are 256 bytes and 8192 bytes. There are times when larger and smaller page sizes are useful. This will be covered in more detail in the implementation and results sections.

**3.9.3 Best-Fit**

The Best-Fit memory allocation policy is a contiguous memory allocation policy. This means that the physical memory is not divided into sections like in paging. Since there are no partitions of the physical memory, a process can only be a contiguous space of memory. This means that a process will only be allocated its memory request if there is a hole as large or larger than the size of the memory request measured in bytes. The Best-Fit algorithm works by searching through the entire linked list of holes and returning the smallest hole that is large enough to accommodate the process. This memory allocation policy will always produce the smallest leftover hole.

There are benefits and drawbacks to the Best-Fit memory allocation policy. The benefit is that this policy is better than the other contiguous policy worst-fit in terms of both speed and storage utilization. The drawback of the Best-Fit memory allocation policy is that it creates many small holes and suffers heavily from external fragmentation. While compaction will resolve this, the overhead cost of compacting a bunch of extremely small pieces of memory is oftentimes not worth it.

**3.9.4 Worst-Fit**

The Worst-Fit memory allocation policy is also a contiguous memory allocation policy. Again this means that a process will only be allocated its memory request if there is a hole as large or larger than the size of the memory request measured in bytes. The Worst-Fit algorithm works by searching through the entire linked list of holes and returning the largest hole that is large enough to accommodate the process. This memory allocation policy will always produce the largest leftover hole.

There are benefits and drawbacks to the Worst-Fit memory allocation policy. One benefit is that this policy leaves the most amount of large memory chucks possible. This helps to minimize instances of processes having to wait before being allocated memory. Memory compaction is also less costly here since the memory blocks being compacted are usually worth the overhead cost. The drawback of the Worst-Fit memory allocation policy is that it is slower than best-fit and does not utilize storage as effectively.

**3.10 Algorithm Implementations**

The purpose of this lab was to be able to see the benefits and drawbacks of each memory allocation policy in regards to each other. The following section will detail our strategy for implementing each of these algorithms. This is a simulated environment and some things were abstracted by the instructor to simplify the complexity of the implementation process.

**3.10.1 Optimal (OMAP)**

The Optimal Memory Allocation Policy (OMAP) was the easiest policy to implement as it is purely theoretical and requires no memory management. The instructor provided us with a value called AvailableMemory that kept track of how much memory was available in the physical space at all times. Normally you would have to use some sort of memory management to figure this out but Optimal relies on knowing this value without any leg work. With this being known, Optimal simply allocates memory to any process provided that the AvailableMemory is equal to or higher than the requested number of bytes. All we have to do is decrement AvailableMemory by the MemoryRequested value of each process admitted into the system. We also increment AvailableMemory by the MemoryRequested value of each process once each process exits the system and has its memory deallocated.

**3.10.2 Paging**

For this lab, the implementation for paging was simplified as opposed to a real-world implementation of paging. First, we needed to run paging using two different page size: 256 bytes, and 8,192 bytes. These values are defined as a variable named pageSize which is initialized in the Global data section of ProcessesManagement2. To change the page size, the user must go into the program and hard-code the value. While this is inconvenient from the user end, it was easy for us to implement. The implementation for the paging policy is written under the getStartAddress method.

If the paging memory policy is selected, we first calculate the number of frames AvailableMemory is split up into. This is checked and calculated in the ManageProcesses method because we only want to run this calculation once. Next, we grab the process we are working with and divide it into pages. We do this by dividing the processes’ MemoryRequested by the pageSize variable; the number of frames is calculated the same way. Our paging implementation is simplified as such: the processes that are segmented never need to be rebuilt in our simulation. This means that we can place our pages anywhere in the physical memory without needing to keep track of where we placed these pieces. Because we are not keeping track of which page corresponds to which frame, we have no use for a page table. This also means that this implementation of paging looks remarkably similar to our implementation of optimal, the only difference that we work with memory in the context of page size.

**3.10.3 Best-Fit**

Best fit is one of the more complex algorithms to implement. As with the previous policies, best fit is implemented in the getStartAddress function. With a best fit implementation, the goal is to search all free memory holes in the physical memory and find the smallest hole that is large enough to accommodate the process.

Keeping up with free memory holes is a complex process. We used the instructor’s provided methods and data structures to help manage a queue of holes that are currently being used and holes that are currently free. These methods are EnqueueMemoryHole, DequeueMemoryHole, and initializeMemoryHoles, the latter being our adaptation of the instructor’s main method from the provided class. Each time best fit runs, it will dequeue a free memory hole from the free hole memory queue. Best fit will set this memory hole as the current best, initially. Next, best fit will loop through every free memory hole in the free memory hole queue. If, for each free memory hole, the selected hole is smaller than the current best hole, while also being large enough to accommodate the memory requested by the process, the current best hole gets sent back to the free hole queue and the selected hole is set as the new current best. When best fit finishes looping through the free hole queue, the process is allocated to the current best hole. This hole is then sent to the parking queue, and is later restored whenever the process occupying that space is sent to the exit queue.

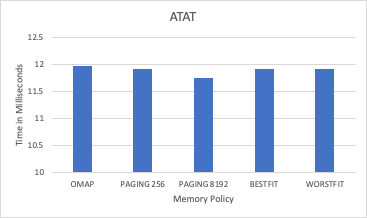
**3.10.4 Worst-Fit**

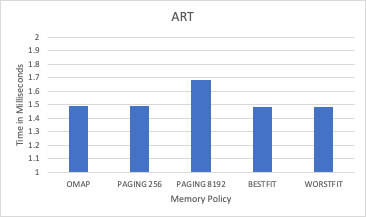
The worst fit algorithm, along with our implementation, is almost identical to best fit. Once the framework for best fit is complete, we can and do utilize the same structures. The difference here is that worst fit looks for the largest hole contained in the free hole queue that is large enough to accommodate the process.

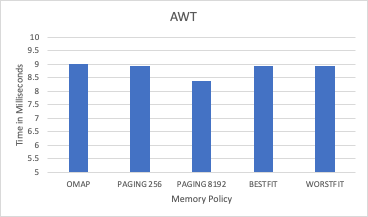
**4 Results**

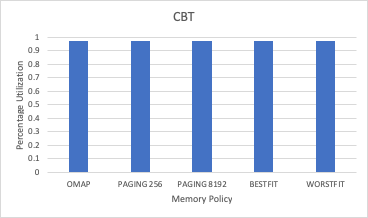
This next section describes the results we encountered after implementing each memory allocation policy and collecting metrics, post-runtime. With the presentation of our actual results, we discuss and compare the expected results. Note that each of the graphed metrics below are combined averages of every CPU scheduling policy implemented in lab 1: First Come First Serve, Shortest Remaining Time First, and Round Robin at various Quantums ranging from 10ms to 500ms.

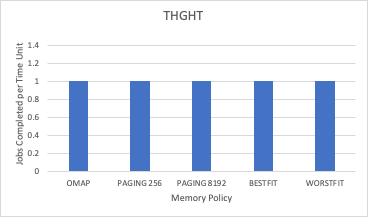
**4.1 ATAT, ART, AWT, CBT, and THGT**

****

****

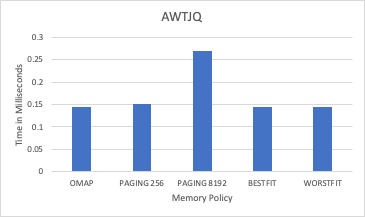
****

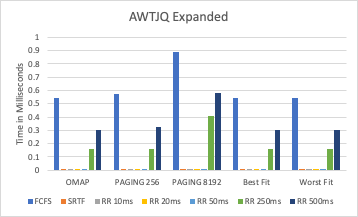
****

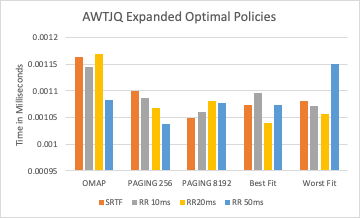
****

The data above for ATAT, AWT, and ART is to be expected. These metrics are mainly affected by the CPU scheduler and the CPU scheduling algorithm utilized by the Operating System. We confirmed this to be true in lab 1. Here we see that despite different Memory Allocation Policies being used, we do not see any significant change in these metrics. The long-term scheduler and the memory allocation policy used only affect how the processes are allocated memory and admitted from the job queue to the ready queue. This has nothing to do with how these processes are allocated the CPU and moved between the ready, running, and exit queues.

**4.2 AWTJQ**

****

****

****

As you can see from our data Optimal Memory Allocation Policy (OMAP) was slightly better in terms of Average Waiting Time in Job Queue (AWTJQ). This is due to the fact that this is the theoretical limit to optimizing this metric. With no memory management required, this algorithm cannot get any better. Following very closely behind was Best-Fit and then Worst-Fit. Both algorithms are contiguous algorithms and do a very good job. The cons of using these algorithms is that normally you have to worry about the cost of memory compaction and sub-optimal resource utilization. However, in our simulated environment the ‘cost’ of memory compaction is simply adding the memory back to the linked list of available memory. This of course does not reflect the true cost or time that the system would have to give to make this memory available for reallocation. For this reason, these two algorithms are showing up as better than normal and are providing almost optimal AWTJQ. Following slightly behind these two policies is Paging with a page size of 256 bytes. The metrics of paging will vary heavily depending on the page size. An optimal page size would be one that balanced maximizing memory utilization and allowing for low AWTJQ. This can be done by selecting a page size large enough that the operating system does not have to keep track of and reallocate large amounts of pages for each process. The page size also needs to be small enough that it limits internal fragmentation and utilizes memory effectively. We only tested two values for this metric, but it seems like 256 bytes is a very effective page size for AWTJQ. I believe that a slightly worse memory utilization was the reason that paging at 256 bytes had higher AWTJQ than Best-Fit and Worst-Fit. Paging with page size of 8192 bytes had terrible values for AWTJQ. This was likely due to the fact that for the processes in this simulation the page size was much too large which caused low memory utilization.

**5 Conclusion**

It is apparent from our data and knowledge of the algorithms that the optimal memory allocation scheduling policy is of course the policy literally named Optimal Memory Allocation Policy (OMAP). This algorithm provided the best results or close to the best results for all six metrics that we collected. It also provided the best results for the metric that matters the most for memory allocation which is Average Time in Job Queue. This algorithm obviously has lots of advantages. It is efficient, fast, and absolutely perfect in terms of memory utilization. The only downside is that it is purely theoretical and cannot be implemented and utilized in real world scenarios. The reason for this is that there is no way to know the value of AvailableMemory without implementing costly memory management techniques. Best-Fit, Worst-Fit, and Paging at 256 bytes all did very well and can actually be implemented in real world scenarios. However, I will note that this lab is a simulation and not representative of how these algorithms would typically perform. Best-Fit and Worst-Fit did not have accurate costs for memory compaction or memory reallocation which led to better than expected results. Paging at 256 bytes was the same way and did not have accurate costs for memory reallocation. Further testing with programs that could simulate this better or testing on actual real-world scenarios would give better insight into the performance of these memory allocation policies. Varying scenarios would also give us more insight into the pros and cons of applying and implementing these policies.