1. **Describe and discuss what memory allocation algorithms you could have used and justify your final design choice**

Some memory allocation algorithms we could have used include first fit, best fit, worst fit, and next fit.

There are 2 main ways that memory can be partitioned for a user and that is through fixed, multiprogramming or with variable, multiprogramming. In a fixed partition, the number of available partitions are fixed and the processes are of different sizes with each partition handling a single process. The first of these is first fit which allocates processes in the first free partition that can contain it. The best fit allocated a process in the smallest possible partition available. Oppositely, the worst fit searches for the largest possible memory partition for the process. Finally, the next fit is very similar to the first fit but it does not begin from the first partition. This means that if a partition is taken up by a process, it will begin the search for the next available partition from the memory address that directly follows the last taken partition. Next fit memory allocation was the algorithm used in our HOSTD dispatcher. Since first fit and next fit are so similar, the decision of which algorithm to use was between these two. If the amount of jobs was significantly smaller than the required amount HOSTD needs to handle, we might have chosen first fit, but because the required load of HOSTD was up to 1000, we opted for next fit.

2. **Describe and Discuss the structures used by the dispatcher for queueing, dispatching, and allocating memory and other resources.**

As mentioned above, the memory allocation algorithm used was next fit and the structures used for queuing, and dispatching were linked list. Since C does not directly support linked lists, a node structure was created which recursively creates another node within it which acts as a pointer to the next element in the “list”. By abusing this structure and storing processes, and available resources as their own structure, our dispatcher is able to enqueue and dequeue in the same way a linked list queue data structure would be able to.

An important tool used to correctly interpret a job from the dispatch list was a tokenizer which was used to separate integers based on comma and space separations in the buffer. By doing this, the process structure could be used to simulate the job requirements within the dispatcher before performing the task which then allowed the system to allocate resources and memory effectively as needed based on logical comparisons.

3. **Describe and justify the overall structure of your program, describing various modules and major functions**

There are 3 major source files included in our provided files: hostd.c, utility.c, and queue.c ; each of which contained their own header file. In queue.c the functions enqueue and dequeue are defined to either go to the end of the recursive struct and add in another process and to remove the first process in a chain respectively. In queue.h, all the structures used throughout the project are created. A resources struct carries all available resources. A node_pointer struct points at the next process and a process struct carries process

information. Utility.c contains all of the major functions used by hostd throughout its execution. These include functions such as init, free, clear, and allocate memory as well as free, clear, and allocate resources. It also contains the function to load the dispatch list into a queue. Finally, Hostd.c is the most complex file in the program. This is where programs are run and queued to be run. Hostd.c determines where each process goes with a series of logical checks that are outlined in the comments in the provided source code. Upon making these logical checks, it queues the process in the corresponding priority queue and then checks to see if there are any real time processes that need to be run. If there are, it runs them to completion and if there aren't then it runs the task for a second, decrements the process_timer, increments a global timer and the job priority (up to 3), and dequeues/enqueues it into the new corresponding priority queue. Hostd will also only allow a job to be shifted into a queue if there are resources available. If resources are unavailable, it will run the next available job to make resources available.

4. **Discuss why such a multilevel dispatching scheme would be used, comparing it with schemes used by "real" operating systems. Outline shortcomings in such a scheme, suggesting possible improvements. Include the memory and resource allocation schemes in your discussions.**

A dispatcher is very essential in an operating system because it provides a way to monitor the status of a job and save the jobs state allowing for the pausing and resuming the process as needed. A scheduler assigns jobs to queues which are then executed by the dispatcher. In this sense, the dispatcher is responsible for making sure that jobs execute within their priority level and ensures that things like starvation do not occur while the system is running. It also ensures that there are no deadlocks, and aims to resolve deadlocks through avoidance or mediation. Without the dispatcher, jobs would run as soon as they arrive and if multiple jobs arrive at the same time then the system is at risk of failure. A dispatcher solves this issue by setting up a procedure for handling jobs that arrive, as they arrive in a manner that allows the operating system to operate in the most efficient manner we know how to achieve.

Of course, a dispatcher is not perfect. In a perfect system, there would need to be infinite resources or, more accurately, resources would need to be able to be shared between multiple jobs. A dispatcher decides the state of job as "ready" or "paused" or "running" or "waiting", etc. It would be ideal if there was some way to share resources between multiple ready jobs as that would allow jobs to run concurrently and without deadlock. Realistically, in an ideal system a dispatcher is not needed and the fact that a dispatcher is in place means that the system will never be perfect; the dispatcher will never be perfect.

One of the biggest issues a dispatcher runs into is that although it can mitigate the effects of deadlock by avoiding the deadlock, it can never completely reduce the time lost. This means that as long as a dispatcher as we know them now is in place, there will always be an overhead. The overhead without a dispatcher is much higher but the fact remains that if multiple jobs arrive at the same time, the second job has to wait for the next time slice before it can execute. On a microprocessor level, this can be visualized as a NOP cycle within a pipeline structure.

Although dispatchers cannot be brought to perfection, the hostd dispatcher designed for this lab could be better improved by changing the memory allocation/partitioning method used. In this dispatcher, we used next fit to allocate a process to a partitioned space but a better algorithm would have been best fit. This would ensure that as many resources were available at any given time as possible so multiple jobs could run in the same time slice. If this dispatcher had to handle a higher volume of jobs, this would greatly improve the efficiency of the system. Unfortunately, since this dispatcher only has to handle a thousand jobs at maximum capacity, the efficiency increase provided by a best fit algorithm is too negligible for the time it would take to implement.