COMP3520 Assignment 2 Report

Alexander Ling 430391570

Q1. Describe and discuss what memory allocation algorithms you could have used and justify your final design choice.

Best fit

The best fit algorithm searches through all of memory for the 'best' fit according to the memory requirement of the process. The algorithm will search through all free memory blocks until it finds the smallest block of memory that can be allocated to the process. As a result the algorithm must scan through all of the possible memory blocks that can be allocated to the process. An advantage of using this algorithm is that there will be a minimum amount of memory wastage as the smallest block of memory is allocated each time. However, a disadvantage of this algorithm is that it is clearly slower than the other algorithms as all of memory must be searched to find the best fit. Moreover this algorithm has the worst external fragmentation problem out of all of the other algorithms. This is when memory is allocated and freed for small processes, leaving small holes of memory which are difficult to allocate to any other processes because of the small size. A large amount of these holes located in memory would lead to memory wastage. Furthermore, because of the nature of this algorithm, these holes in memory have a higher chance of being smaller than the holes left behind by the first and next fit algorithms.

Worst fit

The worst fit algorithm is similar to the best fit algorithm in that it will search through all of memory, but for the worst fit. In other words it will search for and allocate the largest block of unallocated memory to the process. An advantage of using this algorithm is that it attempts to reduce external fragmentation by allocating the largest memory block each time. As a result the holes of free memory are large and can be used to allocate memory for other incoming processes. However, just like the best fit algorithm, the worst fit algorithm will generally be slower than the next/first fit algorithms as all of memory must be searched for the worst fit.

First fit

The first fit algorithm will simply scan from the beginning of memory and chooses the first block of memory that is large enough to accommodate the process. A benefit of using this algorithm is that it is very quick at selecting the block to be used for allocation, especially when compared to the best fit algorithm. This is because first fit simply allocates the first block it sees that meets the memory requirement of the processes. This is much faster compared to the best fit algorithm which may continue searching for a better sized block of memory and waste time in doing so, especially if there are many holes in memory. On the other hand the first fit algorithm is also disadvantaged when it comes to being memory efficient as it may also suffer from external fragmentation. However it has

been shown that the worst case fragmentation is “much worse for the best fit than the first fit system”. [1] Another small disadvantage of using the first fit algorithm is that it will mainly allocate memory at the beginning of memory and hence clutter most of the allocated memory blocks at the beginning.

Next fit

The next fit algorithm is similar to the first fit algorithm in that it chooses the first block of memory that is large enough. However the next fit algorithm starts scanning the memory from where it stopped searching last time, instead of scanning from the start of memory. The next fit algorithm also benefits from quick memory allocation as it chooses the first memory block that fits, exactly like the first fit algorithm. The next fit algorithm also has the disadvantage of external fragmentation for the same reasons as the first fit algorithm. Moreover when comparing first fit and next fit would depend on the memory already allocated and where the allocated memory is located. For example if next fit started searching from the middle of memory when the second half of memory has mostly been allocated it would be very inefficient. Conversely if the first half of memory had mostly been allocated then the first fit algorithm would also waste a lot of time going through small patches of memory in the first half.

Buddy System

The buddy system is a method of splitting up memory and then allocating it to processes. The buddy algorithm attempts to split memory in halves until there is a block of memory that is suitable to allocate to the process. For example, when starting with 512Kb and a request for 100Kb of memory the buddy algorithm will firstly split the 512Kb block into two 256Kb blocks. When splitting the memory blocks into two smaller blocks, these two memory blocks are ‘buddies’. Now as the 256Kb block can be halved and still have room to allocate the memory for the process we then halve only one of the 256Kb blocks to form two 128Kb blocks. As halving the 128Kb block would result in a block that is too small to allocate for the process the buddy system stops and allocates the 128Kb block to the process. When freeing the memory, the buddy system looks at all of the free memory blocks and if there are two unallocated ‘buddies’ they are merged back together. An advantage in using the buddy system is that there is little external fragmentation as most small blocks which are allocated would can be merged back together once freed. Moreover the buddy system is quick when finding memory blocks to allocate as buddy does not have to search through all of memory for a suitable memory block. However a disadvantage of using the buddy system is that it is complex compared to the other discussed algorithms and may take longer to implement. Furthermore the buddy system can suffer from internal fragmentation which is when more memory than is needed is allocated to a process. For example a process requiring 70Kb of memory would instead be allocated a 128Kb memory block.

I implemented the first fit algorithm as it is the simplest algorithm to implement, resulting in easier debugging and a fast implementation time. Moreover compared to best fit, it has less external fragmentation in the worst case and hence is a better choice. I chose this over next fit as we are using

a linked list for memory. As a result looping through the linked list to find where I last allocated memory would be a hassle and waste more time compared to using first fit.

Q2. Describe and discuss the structures used by the dispatcher for queuing, dispatching and allocating memory and other resources.

Below are the structures which are involved in queuing and dispatching. The job dispatch list and the three types of queues are all implemented using a singly linked list of processes. The processes are enqueued and dequeued in a FIFO manner.

Job Dispatch List

The job dispatcher list holds all of the processes that have arrived at the current point in time and are requesting running time. This is the first queue that all processes must enter. In this queue the processes are split up depending on their priorities and sent to either the user job queue or the real time queue. If the process has a real-time priority it is sent to the real-time queue, otherwise it is sent to the user job queue.

Real-time Queue

The real-time queue holds all of the processes with the highest priority. The processes are required to be run immediately on a first come first serve basis. Moreover as soon as a real-time process arrives other lower priority processes must be pre-empted to give time for the real-time process to run till completion, only then can lower priority processes run. In other words as long as there are processes waiting in the real-time queue no lower priority processes receive running time as all running time is given to the real-time processes. In regards to memory allocation, there is a constant block of memory that is allocated *only* for the real-time processes. Therefore the real-time processes should not face any memory allocation issues. Furthermore real-time processes will not request resources so we do not worry about that.

User Job Queue

Processes sent to this queue are the processes which do not have real-time level priority. However the processes do not start running from this queue, instead, processes are sent to a feedback queue depending on the process priority. Moreover these processes may only be sent to feedback queues when their requested memory and resources are available. If there is either insufficient memory or resources for the process it is not allocated onto a feedback queue.

Feedback Queue

Once a process is on a feedback queue it means that memory and resources have been allocated for it and it is able to run. Unlike the real-time queue there will be more than one feedback queue. However, each feedback queue will have it's own priority and processes are sorted into the respective feedback queues based on the process priority. Clearly the processes on the highest priority feedback will run before the processes on a lower priority feedback queue. The choice of which is process is to be started depends on the contents of the queues. Processes in the highest priority, non-empty

feedback queue will be run first based on their arrival time as in each feedback queue the processes are chosen to run in a first come first serve basis. After one quantum of running time the process is then suspended and put on the feedback queue one priority lower than the queue it started from. Moreover if all except the lowest priority feedback queue are empty then the processes are run in a round-robin type fashion. In other words each process will be picked from the start of the queue, will have one quantum of running time, and then be put on the back of the queue. Once processes are finished running they are terminated and not put back on any of the feedback queues.

The structures responsible for memory and devices are represented using C structs.
The processes in the dispatcher are initialized using Pcb structs and are initialized based on user input. The Pcb structs include all the information of the process, including its pid, memory required, priority, running time required, resources required and more. Memory in the dispatcher is handled as a linked list of different sized memory blocks which are the Mab structs. The Mab structs consist of the offset of the memory block, the size, whether it is allocated or not, and next/previous pointers to other Mab blocks. These pointers are used as we use a doubly linked list of Mab structs to represent memory. Lastly the resources are handled using the 'rsrc' struct.The rsrc struct will contain the available number of each of the resources while the dispatcher is running. The resources include scanners, printers, modems, and cd's. While the dispatcher is running it will simply subtract the number of available resources when allocating and add when freeing resources.

Q3. Describe and justify the overall structure of your program, describing the various modules and major functions (descriptions of the function 'interfaces' are expected).

The overall structure of my program includes the use of three different structures: the 'Pcb' struct which represents the process control block; 'Mab' which represents a memory block, and 'Rsrc' which represents the number of resources. These structs are used by the dispatcher to allocate/free memory, start/suspend/terminate processes and to allocate/free resources. The dispatcher handles the high level control of these structs to correctly run our processes. The 'Pcb', 'Mab', and 'Rsrc' modules are described in further detail below.

Processes and Pcb struct
Each of the Pcb structs is that they also include a 'next' pointer to another Pcb struct. This is used to create the linked list queues mentioned in question two.
In the dispatcher queues the Pcb structs are are stored as linked lists through the use of the 'next' pointer in each Pcb struct. This is done through the enqueue and dequeue functions which are a part of the Pcb interface. These two functions allow for dequeuing and enqueuing from any queue which is useful for moving processes to and from the number of queues in the dispatcher. The use of a queue implemented with a linked list is clearly justified as we are using a 'first in first out' principle in all of the queues. This principle is exactly the way a queue data structure works. Moreover the fact that a linked list is used allows for the ability to dynamically add and remove processes instead of using an array which would not allow this.

Another part of the interface includes the suspend and terminate functions which will a signal to the process to terminate/suspend it. However the most important function in the Pcb interface is the 'startPcb' function. This function uses 'fork' and 'exec' to start a new process or to resume a suspended process by sending it a signal.

Memory and Mab struct

Using these next and previous pointers memory is stored in a doubly linked list.
At the beginning of the dispatcher the memory will consist of two memory blocks: One 64Mb memory block for real-time processes; and one larger 960 Mb memory block for all other processes. When enqueueing processes onto the feedback queues the function 'MemChk' is used to check if there is sufficient unallocated memory that can be used to store the process memory. This function will return the first block of memory that it sees with a sufficient size as we are using a first fit allocation scheme. Afterwards 'MemAlloc' and 'MemSplit' are used in conjunction to split the memory block into a correctly sized memory block (which will have the same size as the process's memory requirement to prevent internal fragmentation) and to change the appropriate variables in the struct such as the size and allocated variables. Moreover, once a process has terminated the memory is freed by changing the 'allocated' variable in the struct. Furthermore once a memory block is freed it is also merged with all adjacent unallocated memory blocks by checking the Mab's in the next and prev pointers. This is done using the 'MemMerge' function.
The use of a doubly linked list is justified as the ability to dynamically add and remove memory blocks behind/after each other is needed. This would not be allowed or would be very difficult to implement using another data structure such as an array where the size is fixed. Moreover using a singly linked list would make it more difficult to insert memory blocks behind each other, hence a doubly linked list is used.

Resources and Rsrc struct

There is only one block of resources in the dispatcher and it simply includes the number of each resource. At the beginning of the dispatcher the main rsrc block is set with a certain number of resources, in my case the dispatcher begins with 2 printers, 1 scanner, 1 modem, and 2 cd's. Whenever a process is about to be enqueued onto a feedback queue the dispatcher will check if there are sufficient resources to run the process using 'rsrcChk'. This is possible as the Pcb struct will include information on how many resources are needed. If there are sufficient resources the dispatcher will use 'rsrcAlloc' to subtract from the number of available resources to show that the resources have been taken up by a process. Once a process has been terminated the resources will be freed by adding the number of taken resources back to the available resources struct. This is to ensure that new processes may check and allocate resources for themselves. The use of this simple structure is justified as there is no need for anything more complex, we are simply checking the number of resources available. All this requires is the ability to change four integers that keep track of available resources, and the 'Rsrc' struct and its interface does exactly that.

Q4. Discuss why such a multi-level 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.

The multi-level dispatching scheme used in this dispatcher is done using the three different feedback queues and the real-time queue. What this means is that processes can be separated and categorized based on the process attributes and put on different queues to determine their chances of running. This kind of structure is used as different processes should have a higher priority to run than others. For example a process that manages keyboard and mouse input should have much higher than a process of less importance such as a text editor. As a result different processes are assigned different priorities and will have more or less running time than the others. This is reflected in the Windows OS where process priorities where "The priority levels range from zero (lowest priority) to 31 (highest priority)"[2]. The use of the multilevel feedback queues allow our dispatcher to run high priority processes and then enqueue them on lower priority queues once they have run for one quantum of time. This is especially useful when trying to avoid the starvation of low priority processes as high priority processes will be put on lower priority queues after a duration of time. The use of different priority queues also allows for memory and resource checking before enqueuing on the real-time or feedback queues.

Another advantage of using a multi-level dispatching scheme is that each queue may use a different scheduling algorithm. This is exceptionally useful as the real-time queue may use a FIFO algorithm while the feedback queues may use a round robin approach. As a result the real-time processes will run for their full duration while processes on the feedback queues will take turns running, which is the desired functionality. Furthermore the ability to move processes up and down priorities is useful as a process may have been waiting in a low priority queue for a long time and may need to receive running time, hence it may be moved to a higher priority queue. Also, through the use of the user job queue the dispatcher is able to check if memory and resources can be allocated to a process before actually running these processes. This avoids any problems involving processes running without actually being allocated sufficient memory or resources.

On the other hand one disadvantage of the HOSTD dispatcher compared to "real" operating systems includes the inability to prioritize threads of a process. Both the Windows and MAC OS may allocate different priorities for the threads of a process while our HOSTD only has a single priority level for each process. The MAC OS allows for "thread priority changes"[3] while in the Windows OS "each thread is assigned a scheduling priority"[2]. However in our HOSTD dispatcher once we assign a priority to a process this essentially alters the priority of every single thread in our process. This is because we move the whole process block to another queue when we are changing it's priority. Therefore our processes may not have threads with different priorities. This is a useful feature to have as not all threads of the process should have the same priority. One strategy utilizing thread prioritization is used by the Windows OS to assign a high priority to "the process's input thread, to

ensure that the application is responsive to the user"[2] and to assign lower priority threads to the process background threads, "to ensure that they can be preempted when necessary"[2]. Clearly this is not possible in our HOSTD dispatcher and would be a great improvement to our dispatcher. Another disadvantage of our HOSTD dispatcher is that it does not allow the priority of real-time processes to be decreased. This may be detrimental as in our dispatcher real-time process will block all other processes from running until it is complete. If the real-time process is exceptionally long then other processes will not get much running time. However, the MAC OS implements this feature and allows threads to be "knocked down to normal thread priority"[3]. This could be yet another improvement for our HOSTD dispatcher.

In regards to memory allocation, my HOSTD dispatcher uses the first fit allocation scheme to allocate process memory onto the heap. The memory blocks are stored in a doubly linked list format and to allocate memory for a process the dispatcher searches from the start of the list and allocates the first block of memory big enough. Moreover the HOSTD dispatcher uses dynamic memory allocation, much like most "real" operating systems. This is so we can allocate more or less memory to processes as needed, which would not be possible with a fixed memory allocation system. The use of this simple memory allocation system does have it's downfalls in that external fragmentation can occur. However the Windows OS and many other OS's also use a paging system and "paged memory pool"[4] to store process memory. These pages are contiguous blocks of memory which have been brought from secondary memory into memory to be used by the processes. Moreover since pages have a fixed size, main memory is usually split into blocks which are the same size as the pages. As a result any risk of external fragmentation is removed as the pages will slot perfectly into the page sized slots in memory.
The most useful part of this paging system is that it allows for the non-contiguous storage of process memory. This is done through the use of a variety of tables stored in main memory which tell the OS the physical addresses of the pages and which process the page is used for. As a result process pages may be split up over memory to prevent storage and fragmentation problems. This type of paging system is clearly not used in our HOSTD dispatcher as we store the memory for each process in one large contiguous block. Implementing a similar system would greatly benefit our dispatcher.

References:

1. Worst case fragmentation of first fit and best fit storage allocation strategies - J.M Robson January 1975
2. https://msdn.microsoft.com/en-us/library/windows/desktop/ms685100(v=vs.85).aspx - Microsoft Windows Dev Center
3. https://developer.apple.com/library/mac/documentation/Darwin/Conceptual/KernelProgramming/scheduler/scheduler.html -Mac Developer Library
4. https://msdn.microsoft.com/en-us/library/windows/desktop/aa965226(v=vs.85).aspx