Faculty of Engineering and Applied Science
SOFE 3950U Operating Systems

Lab 4

Umar Qureshi - 100591742

Hamza Farhat - 100657374

Fadi Salback - 100649987

Abdul Zainul-Abedin - 100617275

Date: March 27th, 2020

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

First Fit:

Allocates the first free partition which can hold the process, it finishes after finding a free partition where it can fit.

Advantage: Fastest algorithm because it's search is very minimal.

Disadvantage: The unused memory left after allocation is wasted if they are too small, so requests for larger memory requirements cannot be fulfilled.

Best Fit:

Allocates the smallest free partition which meets the requirement of the requesting process. This algorithm searches the whole list of free partitions first and then considers the smallest hole that is acceptable. It then tries to find a hole which is close to the actual process size needed.

Advantage: Memory utilization is much better than first fit because it searches the smallest free partition available first.

Disadvantage: It is slower and may even tend to fill up memory with tiny useless holes.

Worst Fit

Opposite of Best fit. It locates the largest available free portion so that the portion left will be big enough to be useful.

Advantage: Reduces the rate of production of small gaps.

Disadvantage: If a process requiring larger memory comes along after then it cannot be used as the largest hole is already split and occupied.

The design choice for this lab is to use the First fit algorithm. First fit algorithm is fast, and unlike others allocates memory simply. Although the unused memory may be wasted, we felt it was a better approach than the other algorithms, as they are complex in nature and hard to implement, while First-fit is easy to use, understand, and implement.

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

The data structure used for the dispatch list is a queue. This queue was implemented using the queue.h library. This allowed for the pointers, to allocate memory, check for arrival, as well as the priority of processes. The queue was developed as a basic Linked List. With each node containing some data (a Process in this case) as was a reference (pointer) to the next node. If a next node does not exist, we used a NULL to signify the end of the linked list.

For dispatching, we created an array of queues. This array contained 4 queues (priority, and queues 1 through 3). This made the moving of processes quite easy as the index of the array would just need to be increased. The memory was implemented using a 1024 size int array, where 0 represented the memory location being empty, and 1 represented the memory location being full.

c. 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 the program is a queue data structure. This allows the four level priority dispatcher to maintain the real time and user priority queues. The major functions that are key for the structure implementation are runQueueOne(), checkArrival(), readDispatchList(), and runPriority().

runPriority(): This function is for the Priority Queue. If there are any processes in this queue it will be compiled until completed. It begins by popping the process off queue 0, finds free memory, forks then the child runs the process. It loops until all processes in the queue are completed.

runQueueOne(): This function is for the User Job Queue (queue 1-3). It runs processes for one second in the highest level queue with items in it. After a process is executed for a second and not completed, it is pushed down to the next queue. If it is in queue 3 it gets enqueued to itself.

Once a process is run for a second and not completed, a SIGSTP gets sent to it. The function checks each time if its a process has already started, or if it is a new process. If it sees it's a process that has already been started it sends a SIGCONT to it and it runs for a second. If it is a new process it forks and starts it.

Once a process is completed a SIGTSTP is sent to it to end it.

checkArrival(): Moves items out of temp queue into the correct queue when the arrival time is reached

readDispatchList(): Reads the text file with the processes and data and stores the values in newProc struct.

keepRunning(): This boolean function passes in an argument from queueLevel to return a true or false value in order to check if the queues have items.

d. 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.

Multilevel dispatching prioritizes processes based on their need for processors into queues to execute in order. They are dispatched by a first come first serve schedule. It gives priority to short jobs. This is useful as there is a limited amount of shared memory between processes.

In a real Operating system it is different. It is a bit more complex as there are many other situations handled by the dispatcher. In our program, the implementation would use dynamic partitioning as it increases memory space allocation, decreases fragmentation issues, and would also use chunking, helping in handling small memory allocations.

One shortcoming of limited memory is that this program cannot run a process that requires memory that isn't available in the system as there is only 1024MB of user-level memory. The large process will be deleted from the stack due to its size. As well, if there are other processes running, and there isn't enough memory available, the process will have to wait until other processes free up the memory.