SOFE 3950: Operating Systems

LAB #4: The HOST Dispatcher - Formal Design

Group: Lab Groups - CRN 74025 5

Members:

Name	Student ID
Jason Stuckless	100248154
Noah Toma	100825559
Okiki Ojo	100790236
Bilal Khalil	100825635

Table of Contents:

Introduction:

In this lab we were tasked with creating a Hypothetical Operating System Testbed (HOST), which is a multiprogramming system working within the limitations of limited resources and featuring a four-level priority process dispatcher. A Testbed is a facility where new technologies, computational tools, and scientific hypotheses can be thoroughly, openly, and reproducibly tested. A Dispatcher is an Operating System component that controls how processes are carried out according to their priority; processes with higher priorities are allocated CPU resources before processes with lower priorities. The short-term scheduler's chosen process receives CPU control from the dispatcher module. This involves context switching, switching to user mode, and in the user program, jumping to the proper location to restart. Since a dispatcher is used during every process switch it should be as fast as possible to minimise dispatch latency, which is the amount of time it takes to stop a process and start another by the dispatcher.

A. Memory Allocation Algorithms:

Operating systems utilise a memory management technique called contiguous memory allocation to assign a block of contiguous memory to a process. When a process is allocated contiguous memory, the memory is divided into segments or partitions of a fixed size. When a process is created, the operating system then allocates one of these segments to it. The two main contiguous memory allocation algorithms to implement contiguous memory management are Single-Partition Allocation (a.k.a. Fixed-Size Partition or Static Partition) and Variable-Partition Allocation.

1. Single-Partition Allocation

The simplest method for allocating memory, which divides it into several fixed-size partitions which contain only one process. This limits the degree of multiprogramming available for processes created by users to the number of partitions available. Usually, the operating system chooses the size of the partition, taking into account several parameters like the estimated size of the processes and the amount of physical memory available. [1]

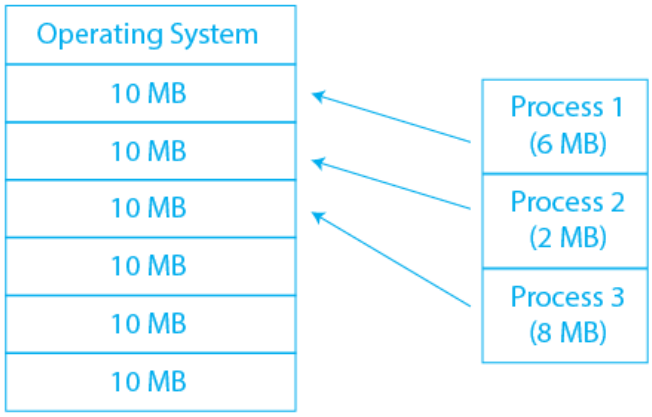

Figure Ai1.[1] Visualization of fixed size memory allocation

The advantages of using this schema are its simplicity, behaviour predictability, ease of management and allocation efficiency. [1] Single-Partition allocation is a straightforward method. This technique is simple to implement and allows the operating system to allocate and manage memory with ease. Since the partition's size is fixed, it is predictable. The operating system can allocate memory appropriately once it has an easy time figuring out how much is available. Because the operating system can keep track of the start and end addresses of each allocated partition, it is simple to maintain. This facilitates freeing memory after a process ends. It is a memory allocation method that is efficient. The operating system can locate a free partition of the right size and assign it to the process very fast. As a result, the overhead of memory allocation is decreased.

The disadvantages of using Single-Partition allocation are limited flexibility, internal fragmentation, and unequal partition sizes. [1] Because each partition has a definite size, it is inflexible. In systems with different memory requirements or processes of different sizes, this could be a drawback. This method cannot support processes whose memory requirements exceed the

size of a partition. When a process uses less memory than allotted to it, internal fragmentation takes place, leaving unused memory in the partition. This unused memory can build up over time and lower the memory management system's overall effectiveness. If the memory size cannot be divided by the partition size, it could result in different partition sizes. Since certain partitions could go unused as a result, this could result in inefficiencies in memory usage.

2. Variable-Partition Allocation

In this schema, a table kept by the OS which shows what parts of memory are free or in use is utilised. Initially, all memory will be one continuous block of free memory. When a process is created it can be given a variable size partition which is sized to its needs. This leads to processes being dispersed throughout the memory, based on what memory location was available at the time of process creation.

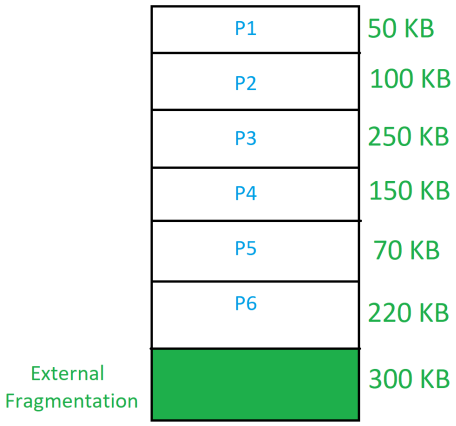

Figure Aii1.[2] Visualization of variable size memory allocation

The advantages of using Variable-Partition allocation are memory utilisation efficiency, flexibility, and internal fragmentation reduction. [1] Because the operating system can assign memory blocks that are precisely the proper size for each operation, it makes memory utilisation more efficient and minimises memory waste. This improves the system's overall efficiency and makes the most of the memory that is available. It is more flexible than Single-Partition allocation since it can handle various process sizes. Because of this, it works well with systems that have different memory needs. Because the OS allots the minimum amount of memory needed for each process, it can lessen internal fragmentation. This can enhance system performance by lowering the amount of memory that is left unused.

The disadvantages of this schema are external fragmentation, increased complexity, and higher overhead. [1] Small spaces between memory blocks cause external fragmentation, which makes memory allocation for larger processes challenging. These gaps may compound over time and lower the memory management system's overall effectiveness. This type of allocation can be more difficult to administer. The start and end addresses of each memory block that are allotted must be tracked by the operating system, which makes it more challenging to release memory when a process ends. This schema necessitates additional overhead due to the operating

system's need to keep track of accessible memory blocks and their status in a list or table, which can be resource intensive and time consuming.

3. Design Choice

The design choice we made is a combination of Single-Partition and Variable-Partition allocation. Since real-time processes are always given a block of 64MB it made logical sense to give real-time processes a static Single-Partition of 64MB within the 1024MB we have available. That 64MB is in the same location at all times and does not change at any point in the running of the HOST program. For user processes however, we chose to go with a Variable-Partition allocation scheme, contained within the remaining 960MB. Processes are allotted memory from this block after they are selected from the queue, before they are executed. An array of size 1024 contains information of which memory is available, so that memory can be allocated dynamically as needed, and to limit the amount of external fragmentation that can occur.

B. Queuing, Dispatching and Memory Allocation Structures:

1. Queuing

The structure used for Queuing is that of a First In First Out queue, which in C we created functions for push, which adds processes to the queue, and pop, which removes processes from the queue. The queue is implemented using linked lists, which are the most effective and straightforward example of a dynamic data structure that is implemented using pointers. [3] Put simply, they work as an array that can expand or contract as needs arise, however, there is no need to define an initial size in a link list, like in an array. Also, items can be added to and removed from any place in the linked list. However, since we're just pushing and popping from the queue, from the tail and front of the linked list, this linked list functionality will not be utilised. One of the disadvantages of using a linked list over an array is that when we want to pop a process from the queue, we cannot just navigate simply to the front of the list, instead we need to iterate over the entire queue until we get to the head. We also used C structs to define the details of each process, such as their arrival time, priority, processing time, memory and so on.

2. Dispatching

For dispatching, we iterate through each queue individually in order of priority, looking for process structs. When a process is located it is popped from the queue in which it was found. A resource struct is then checked to ensure that the resources required by the process are available. If the required resources are available the process is signalled to start and once the process is complete a signal is sent to notify the program that the process has been completed and another process can be selected.

3. Memory Allocation

For memory we start with a contiguous block of 1024MB. 64MB of that block of memory are reserved for real-time processes, which leaves 960MB for user processes found in any of the user queues. A memory array of size 1024 within the resources struct, one index for each MB of memory available, is used to determine what memory is available, the value in the array being a 0 if the memory is available and 1 if the memory is in use. If the process to be allocated memory is a real-time process it is assigned the memory block from index 960 to 1023. If the process to be allocated memory is a user process it is assigned memory from indexes from 0 to 959, depending on what is available at the time.

C. Overall Structure:

1. main.c

This file is the main C file which contains the main function. It contains global variables for queue pointers, the running time, a while loop boolean terminator, and an array of processes for the dispatch list. The queues are initialised and allocated memory. The resources struct is initialised with maximum values. Then it takes the list of processes from the dispatch file, sorts them by arrival time and then adds them to their respective queues. Then a loop checks the queues from highest priority to lowest running a process once one is found and printing the process details, found in the process struct, to the terminal. Lastly the program checks all the queues and if they are empty, terminates the program. If they are not empty, running time is incremented by one quantum and the search for the next process to execute continues.

2. queue.c

This queue code file contains the functions needed to manipulate the linked list queue of nodes containing processes.

i. push function

The push function receives a process and the queue it is assigned to as parameters, and creates a new node in the linked list queue containing the process and sets the pointers for the node and the queue to the appropriate targets. This function is void and as such has no return value.

ii. pop function

The pop function receives a queue as a parameter and returns the process found in the node at the head of the linked list queue. The function iterates through the linked list until it comes to the head of the queue, removes that node from the queue, and alters the pointer for the previous node accordingly. The process found in that node is extracted and returned.

3. queue.h

This queue header file contains a struct for nodes found in the linked list queue, and contains the process that was assigned to the node, and a pointer to the next node in the linked list (NULL if it is the head node).

4. util.c

This file contains one function, arrival_time, which is used to sort the dispatch list array.

5. util.h

This file contains reference to a function in util.c.

6. memory.c

This file contains the functions for allocating and deallocating memory to and from processes.

i. allocate_memory function

Allocates memory to a process based on what type of process it is. If it is a real-time process, it is allocated the 64MB set aside specifically for those processes, and if it is a user process it is allocated memory from a location within the remaining 960MB that is available.

ii. deallocate_memory function

Deallocates memory that was allocated to a process that has completed.

7. memory.h

This file contains constants for memory size and modes.

8. process.h

This file contains a struct for processes, which contains all the information about the process, derived from the dispatch list, such as the arrival time, priority, processing time, memory required, resources required and the location where the process' memory is located.

9. resource.h

This file contains a struct for the resources available, that being, the number of printers, scanners, modems, and CD drives, and the memory available.

D. Discussion:

Multilevel dispatching is used to prioritise certain processes over others based on their importance to the OS and the user, with the most important processes being given the highest priority, in this lab we call them Real-Time processes, while processes with lower priority have to wait for more important processes to complete before execution. In this lab we have two separate sections to the priority levels, with Real-Time processes at level 0, and user processes in a three-level feedback queue. Real-time processes are always given First Come First Served (FCFS) priority, regardless of home many user processes currently exist in the queue, or are currently being processed.

As a shortcoming of this particular scheme, one process is executed at a time while in schemes used by “real” operating systems, multiple processes can be executed at the same time using the functionality of multiprocessor and multi-core systems. [4] In fact, modern processors further include hyperthreading, in which each core acts in such a way to mimic it being two cores, to further extend the processor’s ability to execute multiple processes simultaneously. Though not used in this lab, multilevel dispatching can be done using this functionality to improve performance. Another shortcoming of this approach is that processes in lower queues may never see CPU time, or at least be very delayed in execution, if the higher priority queues never become empty. [5] A possible solution to this issue could be time splitting between higher and lower queues, with higher queues getting a higher share of the total CPU processing time than the lower ones. Though, of course, this comes with its own drawback, that being the increased overhead time involved in constant context switching between processes. Perhaps a better solution would be splitting the processor cores for each queue, again with the higher priority processes getting a higher share. For example, a processor with 8 cores could be split in such a way that the most important processes, level 0, have 4 cores dedicated to them, level 1 has 2 cores, and level 2 and 3 have 1 core each. Though again, there are shortcomings to this idea, as this could result in cores being not used when nothing is located in their matching priority level. To overcome this, when queues are empty their corresponding cores could be assigned to higher priority processes, or lower priority processes if no higher priority processes exist. This does involve further overhead though, in the time it takes to process checking and switching queues.

Another shortcoming of this scheme is that high priority user processes can degrade in priority. [6] While that seems useful in some cases, at other times you can have important user processes that are not real-time processes, but are still very high priority, fall to the same priority as other user processes with little to no importance, depending on the number of real-time processes in the queue. To improve this we could perhaps use a system where user processes of the highest priority are also given their own priority level that does not degrade over time. Or alternatively, user processes of the highest priority could degrade at a slower rate than other processes, requiring more than one quantum to drop a level in the queue. As with most modifications to the system that add complexity, this does lead to more overhead, either by adding another queue level to be maintained or by adding code to specify a user process’ importance.

In the schema used in this lab there is no differentiation between user processes other than the three priorities given, but in Windows, there are four different types of user processes each containing their own inherent priority. These are special system processes, which are user services needed for system management, service processes, which are processes that cooperate with device drivers, network services and others, environment subsystems, which provide the OS

different environments (personalities) that are subsystems to perform tasks shared among all applications, and lastly, user applications, which are executable programs run by the user to make use of the system. [7]

Another drastic difference between the schema used in this lab, and “real” operating systems is the scope of the dispatcher, in terms of number of processes and resources, which is significantly reduced. We only have a few processes, however many are put into the dispatch list, which is drastically different from the number found typically running concurrently in a Windows system, which typically is around 50-60 but can be upwards of 100-200 depending on the user and memory specifications, to say nothing of the total number of processes that get executed daily by a system. [8] Furthermore, we only have 1 GB of memory, while in modern computers 32 GB is becoming standard more and more each day for new computers, though you’re more likely to encounter 16 GB in most modern computers, and even 8 GB in some older ones or lightweight laptops. Even an iPhone 12 Pro has more memory coming in at 6 GB.

Conclusion:

In this lab we have completed code in the C language to design a Hypothetical Operating System Testbed (HOST) to better understand how a multiprogramming system with a multilevel priority process dispatcher, albeit within the confines of a drastically scaled down system with finite resources available. We have implemented the four-level priority dispatcher, resource constraints, memory allocation, and process life-cycle as described in the project document to the best of our ability with no errors, and all the necessary structure and comments required. In this document, we have described all of our requirements and choices made, including the memory allocation algorithm, queuing, dispatching and memory allocation structures, and the overall structure of the program, which describes the modules/functions featured in our code. Furthermore we have discussed the project and how it differs from real world applications, and considered how we might improve upon it. This lab has been successful in teaching the concepts involved and furthered our understanding of process scheduling and resource allocation.

References:

[1] Prepbytes, “Contiguous Memory Allocation in os,” PrepBytes Blog, https://www.prepbytes.com/blog/operating-system/contiguous-memory-allocation-in-os/#:~:text=Contiguous%20memory%20allocation%20is%20a,fixed%2Dsized%20partitions%20or%20segments. (accessed Mar. 9, 2024).

[2] GeeksforGeeks, “Implementation of contiguous memory management techniques,” GeeksforGeeks, https://www.geeksforgeeks.org/implementation-of-contiguous-memory-management-techniques/ (accessed Mar. 9, 2024).

[3] learn-c.org, “Linked lists,” Learn C - Free Interactive C Tutorial, https://www.learn-c.org/en/Linked_lists (accessed Mar. 9, 2024).

[4] J. Ousterhout, “CS 140: Operating Systems (spring 2014),” Threads, Processes, and Dispatching, https://web.stanford.edu/~ouster/cgi-bin/cs140-spring14/lecture.php?topic=thread (accessed Mar. 10, 2024).

[5] GeeksforGeeks, “Multilevel Queue (MLQ) CPU scheduling,” GeeksforGeeks, https://www.geeksforgeeks.org/multilevel-queue-mlq-cpu-scheduling/ (accessed Mar. 10, 2024).

[6] GeeksforGeeks, “Multilevel feedback queue scheduling (MLFQ) CPU scheduling,” GeeksforGeeks, https://www.geeksforgeeks.org/multilevel-feedback-queue-scheduling-mlfq-cpu-scheduling/ (accessed Mar. 10, 2024).

[7] W. Stallings, *Operating Systems: Internals and Design Principles*. Boston: Pearson, 2018.

[8] V. Turiceanu et al., “How many processes should be running on Windows 11?,” Windows Report, https://windowsreport.com/windows-11-how-many-processes-should-be-running/ (accessed Mar. 10, 2024).