Formal Design
Operating Systems
Lab 4

March 17, 2024

Submitted by:

Johnathan Howe 100785128

Muhammad Areeb Khan 100821104

Taha Zahid 100822435

Shamail Rahman 100819169

Github Link: johnh-otu/OSLab4: Operating Systems Lab 4 - Dispatcher Simulator

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

First Fit, Best Fit, and Next Fit are common memory allocation algorithms in operating systems. First Fit selects the first available memory block large enough to accommodate a process, but it may lead to fragmentation. Best Fit searches for the smallest available block, aiming to minimize fragmentation, but it has higher computational overhead. Next Fit operates similarly to First Fit but resumes searching from the previous allocation, potentially reducing fragmentation but still facing efficiency issues. These algorithms might not have been chosen due to their respective limitations, such as fragmentation issues or computational overhead.

Opting for the First Available Memory Allocation Algorithm can be justified for its efficiency, low overhead, simplicity, predictability, and fairness. It allocates memory without extensive searching, maintains minimal overhead, and ensures fair allocation among processes. Its simplicity makes it easy to implement and is suitable for systems prioritizing straightforward memory management. Overall, First Available Memory Allocation strikes a balance between efficiency, simplicity, and fairness, making it a great choice.

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

The main structures that we have used for our dispatcher file employ several different mechanisms for process management, signal handling and allocating memory and resources as well. Below is a breakdown of the various mechanisms and their implementations used in our design:

Signal Handling:

- The dispatcher itself is what uses the different signal handlers to respond to SIGINT (interrupt) and SIGSTP (suspend) signals, and this is done via the signal() function. There are also corresponding flags that are set upon receiving the signal, enabling efficient interruption and suspension handling with our design

Process Management:

- With process management, the dispatcher uses the fork() system call in the main process that is provided in the code. It retrieves the main PID using the getpid() function. We also implemented time delays, and simulation ticks which were achieved through the sleep() function within the main loop of the dispatcher.

Logging and Output:

- For logging, the dispatcherPrint() function prints process IDs alongside messages in designated colors, with a separate function, dispatcherPrintTick(), dedicated to tick information. Output is directed to stdout, ensuring immediate flushing with fflush(stdout).

Resource Allocation:

- The provided code delegated the majority of its resource management, such as CPU time and memory, to the operating system. The dispatcher implements resource allocation by setting process priority via the setpriority() function.
- Additionally, the heap.c and queue.c files provide binary heaps and FIFO queues, respectively, which are critical for process management in the dispatch. The binary heap in heap.c allows for efficient process insertion and sorting by arrival timings, which aids in priority depending on scheduling requirements. Meanwhile, queue.c's FIFO queue guarantees that processes are enqueued and dequeued in the correct sequence. Both solutions are thread-safe and use mutex locks to maintain synchronization in concurrent settings. These data structures provide the dispatcher with critical tools for successfully organizing and controlling operations, allowing for efficient prioritizing, scheduling, and resource allocation throughout the system.

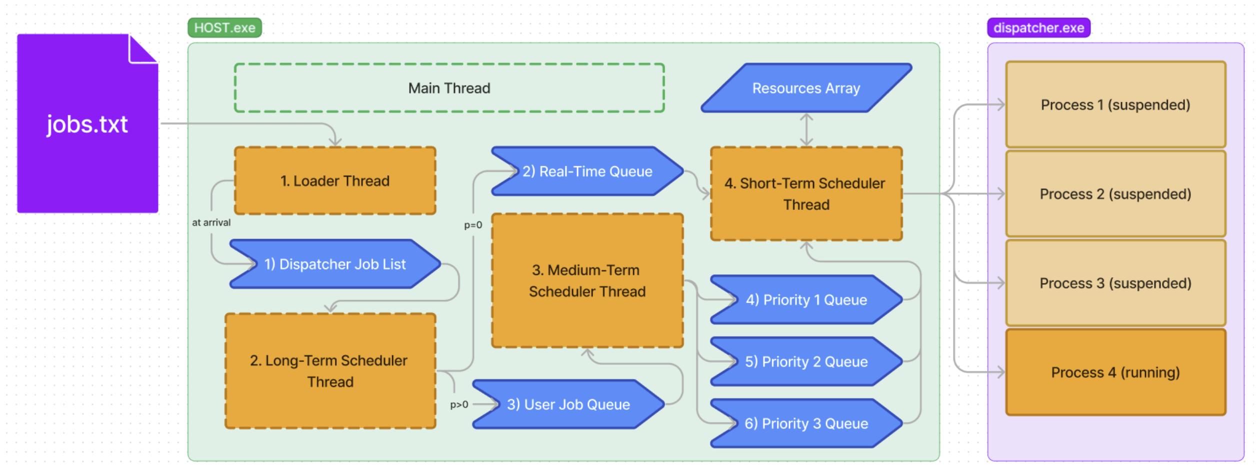
Color Coding:

- Color coding was used for a way to distinguish different log messages from processes. The dispatcher employs this based on the process id % numbers of available colours. In our design we have a colours.h file which defines colour escape sequences using ANSI escape code.

Furthermore, we implemented two different arrays used for memory allocation in our design:

Array for memory, which is used to represent memory allocation. This array keeps track of the memory usage, with each element representing a specific memory unit (bytes, blocks, keys). Additionally, an array of bits is utilized to represent memory availability. Each bit in the array corresponds to a memory unit, with a value of 0 indicating free memory and 1 indicating allocated memory.

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



In our program, we have a jobs.txt file, which is then sent to a **loader module (loader.c)**. This module is responsible for loading processes from the file into the system. The function **'load_queue_from_file(void *args)'** loads processes from a file into a queue data structure and sorts them based on their arrival time using a heap data structure.

The **dispatcher module (dispatcher.c)** manages the execution and scheduling of processes in our system. The main function of our program initialize the dispatcher, set up signal handling, and enter a loop to manage process execution.

Signal handling is used to manage interrupts and suspend processes when received from the system. The function '**sighandler(int sig)**' handles various signals, such as SIGINT and SIGSTP, by setting up different flags. This ultimately allows the dispatcher to effectively respond to appropriate interruptions or suspensions.

Then we have used data structures, which are queues and heaps, for managing processing and system resources:

The Long-Term Scheduler admits processes, manages memory and optimizes resources less frequently than the **Short-Term Scheduler**, which is crucial for system efficiency. When the priority (P) equals 0, processes are directed to the real-time queue; when P is greater than 0, they enter the User Job Queue.

Furthermore, the **Medium-Term Scheduler (MTS**) controls the process count and facilitates swapping between memory and storage, thereby optimizing resources and system performance. Processes from the User Job queue are managed by the MTS, which selects from priorities 1, 2, or 3.

Finally, the Short-Term Scheduler comes into play either through the priority queues or the real-time queue. It rapidly selects processes for CPU execution, ensuring responsiveness.

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 schemes would mostly be used in scheduling algorithms that deal with processes which are easily categorized and distinguishable. Processes are placed at higher or lower priorities depending on how they are evaluated in comparison to each other, in terms of the resources they require. By differentiating between them in this manner, the processor is able to allocate resources such as time and memory more accurately and efficiently between processes, solidifying balance and optimizing performance.

With Multilevel Queue Scheduling, each queue may utilize its own scheduling algorithm. Processes that consume less processor time will usually be placed in higher priority queues, while lower priority queues will not be served until the higher priority queues are complete. As a result, this can easily lead to starvation of the low priority queues.

To counteract this shortcoming, Multilevel Feedback Queue Scheduling allows processes to switch between queues depending on methods that are used to determine how balanced and efficient the processing is. This active way of scheduling can solve the issue of starvation by moving lower priority queues up if they have been waiting too long, or moving higher priority queues down if they are taking too much processor time. Introducing the possibility of multiple priorities and switching between queues could lead to increased complexity overhead, which could be mitigated by optimizing the algorithms or simplifying utilized schemes.

On the other hand, the Round Robin scheduling algorithm provides each process with its own fixed CPU time unit. By enforcing queues through these intervals, the algorithm ensures fairness among processes, but does not reach the same levels of weighted (prioritized) balance and efficiency in resources spent.