**Dispatcher Design Document**

Georgy Zakharov && Thomas Blain

**Memory Allocation Algorithms**

First Fit: Allocate the first free contiguous memory chunk that is of the required length to accommodate the process.

Best Fit: Looks through entire list of free partitions and considers the smallest partition that will accommodate the process, algorithm then tries to find a partition that is closest to actual size needed.

Worst Fit: Allocates largest free memory chunk to ensure that the partition that is left will be big enough to be useful, reverse of best fit.

Buddy System: Memory is partitioned into chunks of integral power of 2 (i.e 2/4/8/16/etc) up to max size. When a memory chunk is requested of size 2k, a 2k + 1 size chunk is split in two (“buddies”) and used to satisfy the process request.

Next Fit: Similar to first fit in that it allocates the first free contiguous memory chunk, however, when called the next time for a following process request it will continue searching from where it left off not from the beginning like first fit.

We used the First Fit memory allocation algorithm because it is simple to implement and performs well compared to the other algorithms, there are fewer searches but larger chances of “gaps” of memory remaining that are too small to be allocated to any process. The faster performance helps with a smaller program like our shell dispatcher to help maintain responsiveness.

**Structures Used by Dispatcher**

A Resources struct is used to keep track of all resources available for allocation to processes that arrive in the queue, it keeps track of number of available printers, scanners, modems, cd drives and memory.

A Proc (process) struct keeps track of the information for each process this includes: arrival time, priority status, CPU time needed, starting index of memory which it was allocated, and requested resources (printers, scanners, modems, cd drives and memory size).

A node struct represents the single-linked list structure used for our queues, it keeps track of the process that it contains and a pointer to the next node object after it.

Lastly, the dispatchlist.txt file provides details about the processes that will need to be created and serviced.

**Overall Structure**

The main program consists of the Utility and Queue headers and their implementations, the Utility functions take care of defining the structs used, allocating and freeing memory as well as loading the given process details from the dispatch file.

The Queue functions take care of the functionality of the linked list structure that is used to represent our priority queues, functions like push and pop to control the data coming in/leaving each queue.

**Why Multilevel Dispatching?**

Multilevel dispatching allows the dispatcher to service real-time requests in a timely manner, once the highest priority processes are finished, the dispatcher moves on to the lower priority processes.

With existing operating systems, no single algorithm is the best and often times OSes uses hybrids/combinations of existing algorithms (such as Round-Robin, FIFO, fixed-priority preemptive scheduling, etc) or by extending them and tailoring to specific needs. Each algorithm has its own pros/cons and depending on the end users’ needs a scheduling algorithm can be tailored to provide optimal performance. The problem with our multilevel dispatching is that some low-priority processes may experience starvation if for example, there will be continuously added real-time processes with sizable processing time requirements. A possible improvement is implementing aging, where processes are promoted back up into higher priority queues incrementally until they are processed, a decision can be made whether or not a process can be promoted to real-time priority or only up to a maximum of level 1 priority.

The memory allocation scheme we are using is similar to the segmented memory management strategy for operating systems. Most current OSes use paged memory management which allows the use of the secondary storage (such as SSD/HDD) as a slower memory tool by swapping “pages” in and out of memory when needed. This is much more complex to implement than our simple allocation mechanism but proves to be more robust in handling situations where we run out of memory but need to use more and have processes that are waiting for I/O or a response from another process.