COMP3520 Operating System Internals

James Peter Cooper-Stanbury `312154402` james@cooperstanbury.com

Memory Allocation Algorithms

First Fit

The first fit memory algorithm attempts to place memory in the first available location that is large enough. The selected space is then split into 2 peices, one for the memory, and then one for the unused memory so it can be used for another job. Once the system has been used for a while, the start of the list may become very fragmented, meaning larger search times because the search must pass over every small block for every search performed.

Best Fit

The best fit algorithm searches the list and finds the smallest (but large enough) available free block. This means you should be using the most optimal memory space, however it is much slower than first fit because the entire list must be searched every time to ensure the most optimal block is found. In this method, there may be many extra blocks, too small for any process to run in which means less total memory can be used at any one time.

Worst Fit

Worst fit is the opposite of best fit, still scanning the entire list each time, worst fit attempts to find the largest free memory space. this way when the block gets split, the remaining memory has a higher chance of being usable.

Next Fit

Next fit is very similar to first fit, however with next fit, a pointer to the last used memory location is stored, meaning the start of the list does not need to be scanned everytime. This reduces the likelyhood of the start of the list becoming fragmented and can improve search times.

Buddy System

With buddy systems, a tree structure is used, rather than a list structure. While this greatly improves search times, fragmentation can increase, since typically buddy systems split on powers of 2. This means each block that gets used has a much higher chance of fragmenting. Allocating memory may also require multiple splits first.

My Choice - First Fit

For no particular reason, i chose to implement first fit algorithm. The benefit here is it is easy to

For no particular reason, I chose to implement first fit algorithm. The benefit here is it is easy to debug, and we arent having a speed race. It also manages to use memory quite efficiently due to the splitting and joining nature, allowing for a fairly smooth run.

Unfortunatly i couldnt get my first fit implementation working well enough, so i have chosen not to include it in my submission.

Dispatcher structures

First Come First Served (FCFS)

First come first serve does what it says on the tin, the first process to come along is the first to get run and also the first to finish executing. The program will block until the current process on the FCFS queue has terminated. This is good for high priority processes that need to be run in real time.

Feedback

Feedback dispatching runs all the currently active processes for one clock cycle each. Theoretically if 3 processes come in at the same time and require the same cpu resources, they should all start and finish within 3 clock cycles of each other. This is good for lower priority processes that you just want to finish, but not nessecarrily with any sort of priority.

Round Robin

Round robin is simply a multi-tiered feedback queue. Once a process has been run for one cycle, its priority gets lowered and it gets put on the next queue. This means if 3 processes are running at the lowest priority (call this 3) and a new process comes in with a higher priority (say, 2) the new process will run for a few cycles before joining the other processes at a lower priority.

Program Structure

My program is structured fairly linearly, to avoid any confusion it has been moderately commented.

I created an extra struct `queue` for the purpose of storing the pointers to the first and last element of each of the queues i use. A simple method `createQueue()` allocates the appropriate memore and returns the pointer to the queue object in memory. For the round robin queue structure, i created an array of queues `QueuePtr rr[NUM_RR]` where `NUM_RR` is the number of levels in the round robin system. Theoretically a user could choose to implement a 100 level round robin system. For what application, who knows.

The next section is reading in from the file specified in the first command line argument. The file is read line by line and a new pcb object is created. all the initial variables for each pcb are set, and the pcb added to the dispatch queue. At this point i print a quick status line with the arrrival time, priority and remianing cpu time of the pcb.

Here is where the fun begins. with a series of logic in a largish while loop, the behaviours as defined by the Dispatcher structures is executed. Pointers are handed around to different functions like business cards at a health conference. Such functions like `startPcB(x)` start the actual process for the specified pcb.

In this main logic, some neat helper functions are used, like `highest_priority_process(x)` which returns the index of the highest priority feedback queue with a waiting process, and `process_in_queues(x)` which takes as input the array of round robin lists and returns `1` or `0` based on if there anre any waiting processes in these queues.

Other useful functions like `deqPcb(q)` and `enqPcb(q, x)` queue and dequeue processes on to the queue specified.

The layout of these helper functions is sensical, with Memory functions at the bottom, queue functions above that, pcb functions above that, and priority helper functions above that, right below the main.

A header file `main.h` is provided with the function prototypes and structs defined.