COMP3520 Operating Systems Internals Assignment 3: Stage 2 – Simple Memory Management

General Instructions

This is Stage 2 of Assignment 3. It is about multi-level queue dispatcher under the memory constraints and consists of two compulsory tasks:

1. **Revise Assignment 3 Stage 1 program** to add a simple memory management scheme for a uniprocessor system (**70%**); and
2. Answer the discussion document questions (**30%**).

This assignment is an individual assignment. Whilst you are permitted to discuss this assignment with other students, the work that you submit must be your own work. You should not incorporate the work of other students (past or present) into your source code or discussion document.

Your work for Stage 2 will be marked (30% of the total marks for Assignment 3). You must first complete tasks for Stage 1 before tackling problems for Stage 2. You will obtain a **Failing** mark if you only submit your work for Stage 2 without submitting your work for Stage 1.

You are required to submit your **source code**, **job dispatch list files**, **makefile**, and **discussion document** together in a **zip/tar file** to the submission inbox in the COMP3520 Canvas website.

Simple Memory Management

In Stage 2 of Assignment 3, you will add a simple **First Fit memory allocation** scheme to the multi-level queue dispatcher you have done in Stage 1 of the assignment. To manage memory allocation, you need a structure of **memory allocation block**, or `mab` in a **doubly linked list**:

```
struct mab {
       int offset;   //starting location of the block
       int size;     //size of the block
       int allocated;//whether allocated, or free
       struct mab * next;
       struct mab * prev;
};
typedef struct mab Mab;
typedef Mab * MabPtr;
```

Allocating memory is a process of finding a block of 'free' memory big enough to hold the requested memory block. The free block is then split (if necessary) with the right size block marked as 'allocated' and the remaining block (if any) marked as 'free'.

Freeing a memory block is done by changing the flag on the block to 'free'. After this the blocks on both sides of the newly 'freed' block are examined. If either (or both of them are 'free') the 'free' blocks must be merged together to form just one 'free' block.

The `mab` blocks are connected using a doubly linked list to make it easier to check for adjacent unallocated blocks that need to be merged when freeing up a memory block.

The following set of subroutines you may need to manipulate the data structure (`mab` and doubly linked list) for the simple memory management:

```
MabPtr memMerge(MabPtr m);            // merge two memory blocks
MabPtr memSplit(MabPtr m, int size); // split a memory block

MabPtr memChk(MabPtr m, int size);    // check if memory available
MabPtr memAlloc(MabPtr m, int size); // allocate memory block
MabPtr memFree(MabPtr m);             // free memory block
```

Several of these subroutines have been implemented for you in files “mab.c” and “mab.h”. However, you need to implement your own `memChk()`, `memAlloc()` and `memFree()` and add them to “mab.c” for First Fit dynamic memory allocation scheme.

Job Scheduling under Memory Constraints

In Stage 2 of Assignment 3, you will have a list of randomly generated jobs that you need to schedule for execution depending on the job’s time of arrival, its allowed execution time, priority and memory requirement (in MB). There are two types of jobs, i.e., real-time jobs, and normal jobs. Real-time jobs are given a priority value 0 and normal jobs are initially given a priority value 1. The larger the priority value, the lower is the priority. The job dispatch list format is given as follows:

<arrival time>, <cputime>, <priority>, <mem requirement>

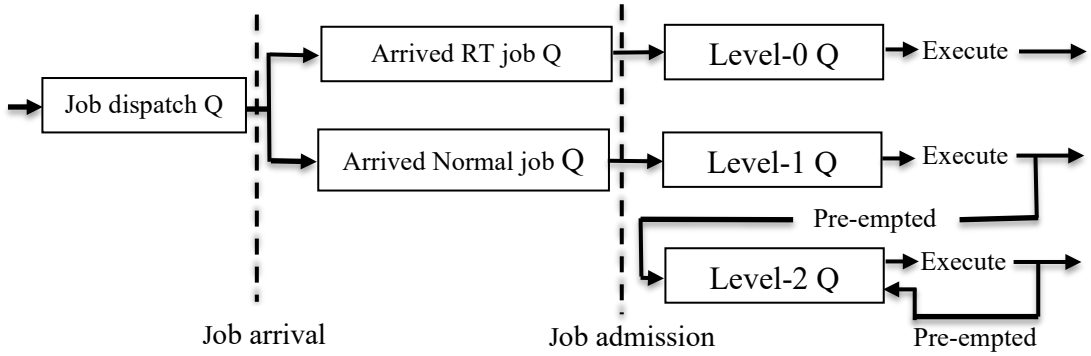
0, 3, 1, 128

2, 6, 0, 180

4, 4, 1, 65

6, 5, 1, 350

8, 2, 0, 284



As depicted in the above figure, for this stage of the assignment you need to implement **ONE** Job dispatch queue, **TWO** Arrived job queues **and** a **THREE LEVEL** ready

queue to manage and dispatch jobs under the constraints of **a "simulated" memory of size 1GB**.

1. At the beginning all jobs in the job dispatch list file are loaded to the Job dispatch queue.
2. Wen jobs have "arrived", they will be dequeued from Job dispatch queue and enqueued to the Arrived job queues, with real-time jobs being loaded to the Arrived RT job queue and normal jobs loaded to the Arrived Normal job queue.
3. When memory can be allocated, "arrived" jobs will be moved from the Arrived job queues to the three-level ready queue and ready to be scheduled for execution. Real-time jobs are moved to level-0 queue at the top and normal jobs are moved to level-1 queue in the middle of this three-level ready queue.
4. Real-time jobs in Level-0 queue are dispatched in a FCFS manner. When a real-time job is dispatched, it will run to the completion without interruption.
5. Normal jobs in Level-1 queue are also dispatched in a FCFS manner. However, a normal job in Level-1 queue is only given a time-quantum for execution when being dispatched. It will run without interruption within the given time-quantum. If it cannot complete in the time-quantum, the job will be pre-empted and moved to the end (or tail) of Level-2 queue, and its priority value changed to 2 (i.e., its priority is reduced).
6. Jobs in Level-2 queue can be scheduled to run only when both Level-0 and Level-1 queues are empty. When a new job arrives (which will be placed in either Level-0, or Level-1 queue), the currently running Level-2 job will be pre-empted immediately. Instead of moving it back to the end (or tail) of Level-2 queue, it will be placed in the front of Level-2 queue. Therefore, jobs in Level-2 queue are scheduled in a strictly FCFS manner, though the running Level-2 job may be pre-empted by higher priority jobs.
7. The dispatcher will immediately schedule another job for execution when the currently running process is terminated (completed) or suspended (pre-empted), and it always selects a job with the highest priority for execution.
8. When a job is terminated, the memory block allocated to it is freed. The system will consider admission of arrived jobs in the arrived job queues immediately due to the change of memory condition (i.e., more free memory space now in the system). If memory can be allocated, the arrived jobs will be moved to the multi-level ready queue immediately and ready to be scheduled for execution.

In the above procedure, the process scheduling scheme is the same as that in Stage 1 of the assignment for Multi-level Queue Dispatcher except the admission of arrived jobs to the three-level ready queue for execution under the memory constraints, and management of memory immediately after job termination (i.e., item 8 in the above procedure).

For the admission of arrived jobs to the multi-level ready queue for execution:

1. Both Arrived job queues are FCFS queues.
2. The system always first considers real-time jobs in the Arrived RT job queue for admission if memory can be allocated.
3. Only when Arrived RT job queue is empty, jobs in the Arrived Normal job queue are considered for admission.
4. Again, the First Fit allocation scheme is used for dynamic memory management and the "simulated" memory is of size 1GB.

Requirements

Source Code

1. Your program must ask the user to enter an integer value for time-quantum for Level-1 queue.
2. You must keep the logical structure of the program for Multi-level Queue Dispatcher in Stage 1 of Assignment 3 intact, i.e., you can only make necessary changes to those programs.
3. Your program must be able to correctly calculate and print out on the screen the average turnaround time and average waiting time.
4. Your program must be implemented in the C programming language. C++ features are not permitted except those that are part of an official C standard.
5. Your source code needs to be properly commented and appropriately structured to allow another programmer who has a working knowledge of C to understand and easily maintain your code.
6. You need to include a *makefile* that allows for the compilation of your source code using the *make* command on the School of Computer Science servers. It is crucial that you ensure that your source code compiles correctly on the School of Computer Science servers. If your source code cannot be compiled on the School servers, you will receive a **failing** mark for it.

Discussion Document

1. Write a pseudo code for your Multi-Level Queue Dispatcher with First Fit dynamic memory allocation. Your pseudo code must have a similar style to that for Round Robin Dispatcher given in Assignment 2 description.
2. Give **2, or 3** job dispatch list files, each list containing **at least 6** jobs, you designed for testing and debugging various aspects of your source code to make sure your code is bug free and can produce correct results. You must explain **how** these job dispatch lists are designed for **what** purposes.