Project 3 - Memory Allocation

In this project, you will extend part C from project 2 to handle memory allocation. You may assume that the make believe operatin system has a 1024 Mbyte Memory available for processes

Memory allocation must be as a **contiguous** block of memory for each process that remains assigned to the process for the lifetime of that process. DO NOT USE FIXED SIZED PARTITIONS, USE VARIABLE SIZED PARTITION BASED ON THE SIZE OF THE REQUESTED MEMORY FROM THE PROCESS. Do you need to do some compaction? What if not eanough memory is available at process creation time? What if the process requests more memory than physically is present?

Enough contiguous spare memory must be left so that the Real Time processes are not blocked from execution - 64 Mbytes for a running Real-Time job leaving 960 Mbytes to be shared amongst "active" User jobs.

The HOST hardware MMU can not support virtual memory so no swapping of memory to disk is possible. Neither is it a paged system.

Within these constraints, any suitable varible partition memory allocation scheme (First Fit, Next Fit, Best Fit, Worst Fit) may be used.

From Part C in the last project, the life-cycle of a process is:

1. The process is submitted to the dispatcher input queues via an initial process list which designates the arrival time, priority, processor time required (in seconds), and memory block size.
2. A process is "ready-to-run" when it has "arrived" and memory is available.
3. Any pending Real-Time jobs are submitted for execution on a First Come First Served basis.
4. If enough memory is available for a lower priority User process, the process is transferred to the appropriate priority queue within the feedback dispatcher unit, and the remaining memory indicator (memory list) updated.
5. When a job is started (fork and exec("process",...)), the dispatcher will display the job parameters (Process ID, priority, processor time remaining (in seconds), memory location and block size) before performing the exec.
6. A Real-Time process is allowed to run until its time has expired when the dispatcher kills it by sending a SIGINT signal to it.
7. A low priority User job is allowed to run for one dispatcher tick (one second) before it is suspended (SIGTSTP) or terminated (SIGINT) if its time has expired. If suspended, its priority level is lowered (if possible) and it is re-queued on the appropriate priority queue. The User job should not be suspended and moved to a lower priority level unless another process is waiting to be (re)started.
8. Provided no higher priority Real Time jobs are pending in the submission queue, the highest priority pending process in the feedback queues is started or restarted (SIGCONT).

9. When a process is terminated, the memory it used is returned to the dispatcher for reallocation to further processes.
10. When there are no more processes in the dispatch list, the input queues and the feedback queues, the dispatcher exits.

The various methods of dynamic segment partitioning are described in the chapter on memory aloccation (and from class).

If you wish to implement (and you do) Best Fit, First Fit, Next Fit, or Worst Fit memory allocation policy, it is probably best to do this by describing the memory as a structure in a linked list:

```
struct mab {
    int offset;
    int size;
    int allocated;
    struct mab * next;
    struct mab * prev;
};

typedef struct mab Mab;
typedef Mab * MabPtr;
```

The following set of prototypes give a guide as to the functionality you will need to provide:

```
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

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

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 structure above has been constructed with a reverse link as well as a forward link. This makes it easier to check for adjacent unallocated blocks that need to be merged when freeing up a memory block. Without the reverse link, you will need to do a separate pass through the memory arena after marking a block as free to merge any adjacent free blocks.

You should go directly to implementing the routines in your Feedback dispatcher (part c from last project, or use my solution). For this you will need to insert a separate queue in between the input queue and the Feedback queues.

Implementing the extra stage, processes are passed from the input queue (Job Dispatch List) to the Feedback pending queue (User Job Queue) when they have "arrived". They are dequeued and enlisted on the appropriate Feedback queue when sufficient memory is available for them in the memory arena - the memory is allocated when they are enqueued and deallocated when they are terminated.

The modified logic (highlighted in red) will be:

```
1. Initialize dispatcher queues (input queue, user job queue, and feedback queues);
2. Fill input queue from dispatch list file;
3. Start dispatcher timer (dispatcher timer = 0);
4. While there's anything in any of the queues or there is a currently running process:
     i. Unload pending processes from the input queue:
        While (head-of-input-queue.arrival-time <= dispatcher timer)
        dequeue process from input queue and enqueue on user job queue;
    ii. Unload pending processes from the user job queue:
        While (head-of-user-job-queue.mbytes can be allocated)
        dequeue process from user job queue, allocate memory to the process and
        enqueue on highest priority feedback queue (assigning it the appropriate
        priority);
   iii. If a process is currently running:
          a. Decrement process remainingcputime;
          b. If times up:
               A. Send SIGINT to the process to terminate it;
               B. Free memory allocated to the process;
               C. Free up process structure memory;
          c. else if other processes are waiting in any of the feedback queues:
               A. Send SIGTSTP to suspend it;
               B. Reduce the priority of the process (if possible) and enqueue it on
                  the appropriate feedback queue
    iv. If no process currently running && feedback queues are not all empty:
          a. Dequeue a process from the highest priority feedback queue that is not
             empty
          b. If already started but suspended, restart it (send SIGCONT to it)
             else start it (fork & exec)
          c. Set it as currently running process;
     v. sleep for one second;
    vi. Increment dispatcher timer;
   vii. Go back to 4.
5. Exit.
```

or something like that!

Try this with the following job list with a quantum of 1 sec and available memory of 1024 Mbyte:

```
0, 1, 1, 192, 0, 0, 0, 0
0, 1, 3, 32, 0, 0, 0, 0
0, 1, 1, 416, 0, 0, 0, 0
0, 1, 3, 32, 0, 0, 0, 0
0, 1, 1, 160, 0, 0, 0, 0
0, 1, 3, 128, 0, 0, 0, 0
7, 1, 2, 108, 0, 0, 0, 0
7, 1, 2, 256, 0, 0, 0, 0
7, 1, 2, 80, 0, 0, 0, 0
```

This interesting mixture should run processes 0-5 in the first 6 seconds and leave the memory allocation as:

offset	**size**	**allocated**
0	192	FALSE
192	32	TRUE
224	416	FALSE
640	32	TRUE
672	160	FALSE
832	128	TRUE
960	64	FALSE

with the Next Fit pointer at offset 960. This will be the case whichever of the four allocation policies you are using. The next three processes should then be loaded into the queue and dequeued, enlisted and then executed in the order: process 6, 7, then 8.

Under First Fit, the 108 Mbyte, 256 Mbyte, and 80 Mbyte processes will be allocated at offset 0, 224, and 108 respectively.

Under Next Fit, they will be allocated at offset 0, 224, and 480 respectively.

Under Best Fit, they will be allocated at offset 672, 224, and 480 respectively.

Under Worst Fit, they will be allocated at offset 224, 332, and 0 respectively.

Your project will be run as follows: (note the options on the command line):

```
./hostd [-mf|-mn|-mb|-mw] "dispatch file"

where
    "dispatch file" is list of process parameters as specified above
    -mx is optional selection of memory allocation algorithm as follows:
        -mf First Fit (default)
        -mn Next Fit
        -mb Best Fit
        -mw Worst Fit
```