Assignment #3: Mutual Exclusion and Memory Management

Requirements

In this assignment you are to implement two relatively small programs: Dining Philosophers and Simple Memory Management.

Specifications

Part 1.
Implement a solution to the dining philosopher's problem. The pseudo code for this can be found in your textbook. Your program should take two command-line arguments:

1. The number of philosophers (an integer larger than 2)
2. The number of times each philosopher should eat (an integer in the range 1..1000).

Each philosopher should start out thinking and announce when he has begun to eat and when he has begun to think.

Note: a dining philosopher must be implemented either by a thread or a concurrent process, and the mutual exclusion among philosophers may be achieved by using any primitives, such as wait/signal, lock/unlock, *etc*., at your disposal.

Thus, the following might be a transcript:

```
$ ./dine 3 2
Philosopher 1 thinking
Philosopher 1 eating
Philosopher 2 thinking
Philosopher 3 thinking
Philosopher 1 thinking
Philosopher 2 eating
Philosopher 2 thinking
Philosopher 3 eating
Philosopher 3 thinking
Philosopher 1 eating
Philosopher 1 thinking
Philosopher 2 eating
Philosopher 2 thinking
Philosopher 3 eating
Philosopher 3 thinking
```

Part 2.
We discussed four algorithms for placing processes into holes in memory: best fit, worst fit, next fit and first fit. Your job is to write a simulator that will take processes of varying sizes, load them into memory according to each of those rules and swap processes out as needed to create a larger hole.

Let us make the following assumptions:

1. Memory is of size 128MB.
2. The size of each process will be some whole number of megabytes in the range 1..128.
3. An initial list of processes and their sizes is loaded from a file, which is the only command-line argument. These processes should be loaded into a queue of processes waiting to be loaded into memory.
4. Memory is initially empty (we will ignore the space taken by operating system).
5. If a process needs to be loaded but there is no hole large enough to accommodate it, then processes should be swapped out, one at a time, until there is a hole large enough to hold the process that needs to be loaded.
6. If a process needs to be swapped out, then the process that has been "in memory" the longest should be the one to go.
7. When a process has been swapped out, it goes to the end of the queue of processes waiting to be swapped in (see next item).
8. Once a process has been swapped out for a third time, we assume that the process has run to completion and it is not re-queued. Note: not all processes will be swapped out for a third time.

You can use whatever data structure you wish (*e.g*. linked list, bitmap) to simulate memory usage, but please make sure that the comments in the file clearly indicate what data structure you are using and how it is implemented.

The "*process file*" will be in the following format:

Process id (a single letter) <space> process size (an integer)

Here is a sample process file:

A 13
B 99
C 2
D 2
E 44
F 32
G 2
H 9

Your program should do the following:

1. Simulate 4 allocation strategies, namely, first fit, best fit, next fit and worst fit, one by one.
2. Each time a process is loaded into memory, a line should be printed such as:

pid loaded, #processes = 5, #holes = 3, %memusage = 41, cumulative %mem = 40
where %memusage refers to the percent of memory that is currently occupied by processes, and cumulative %mem gives the average of all the %memusages up until and including the current process load.

3. When the queue is empty, a line such as the following should be printed:

Total loads = 33, average #processes = 14.4, average #holes = 6.3, cumulative %mem = 62

This program should be called holes.c, and should take a single command-line argument as indicated above:

$ ./holes testfile1