ECS 60 **Programming Assignment #4, 50 points** **Spring 2016**

Due Wednesday, May 18th , 11:59pm.

Files NOT to handin to cs60 p4 are: MemRunner.cpp, MemRunner.h, CPUTimer.h, mynew.cpp, mynew.h

Minimum files to handin to cs60 p4 are: authors.csv, Makefile, MemMan.h, and MemMan.cpp.

You are to write an efficient program that will simulate a memory manager. I have provided the driver program MemRunner.cpp, and the necessary class stubs in MemMan.h, and MemMan.cpp. You may add any classes and/or code you wish to MemMan.cpp, and MemMan.h. You may also use, and alter any Weiss files. MemCreate.out was compiled from MemCreate.cpp, and creates files used for testing. You should study MemCreate.cpp to better understand the probabilities and ranges of the operations requested. All files mentioned, as well as my executable, MemRunner.out, can be found in ~ssdavis/60/p4

Further specifications:

1. Dynamic Memory Allocation.
   - 1.1. You may not use malloc(), free(), maxRAM, or currentRAM anywhere in your source code, including any Weiss files. No entity larger the 80 bytes may be created without using new. If you use any outside files, other than template files, you must add #include "mynew.h" at the top of the files.
   - 1.2. The underlying code of the overloaded new provided by mynew.cpp relies on the fact that the real memory manager in the CSIF 64-bit machines stores that actual size allocated (plus 1) in the eight bytes before the address returned. Using memsize.out, I found that requesting 1 to 24 bytes costs 33 bytes, requesting 25 to 40 bytes costs 49 bytes, and requesting 41 to 56 bytes costs 65 bytes. Thus, the manager works in increments of 16 bytes, and charges 8 bytes for its overhead no matter what size is requested. It is these costs that are summed to determine your RealRAM figure.
2. MemRunner.out takes two parameters. The first is the name of the Mem file to be tested. The second is a single character that indicates whether the program should print out debugging information. If the character is '0', then it will not print info, else it will. The actual testing will use '0' as the second parameter.
3. Operation Files
   - 3.1. These are created by MemCreate.out. The names reflect the number of processes, number of operations, and the seed used to initialize the random number generator.
   - 3.2. The first line contains the Ideal RAM, which is the maximum RAM ever allocated at one moment during the operations specified in the file.
   - 3.3. Each of the rest of the lines have the format: Command <char> Process <int> {BlockNum <int> {Size <int>}}
     - 3.3.1. Commands: 'N' = alloc, 'A' = access, 'D' = dealloc, 'E' = endProc, 'F' = fault.
4. MemMan Operations
   - 4.1. alloc() takes a process number, and size to allocate. It returns the start address of memory block allocated. Occurs 17% of the time, and ranges from 4 - 262,144 bytes with a hyperbolic distribution leaning towards the smaller values.
     - 4.1.1. MemCheck checks to make sure the memory allocated is currently unused, and will print an error it is used.
   - 4.2. access() takes a process number, and address to access. The address need not be the start address of a block.
     - 4.2.1. If the process owns the address then return true. 70% of all operations.
     - 4.2.2. The 'F', fault, commands cause MemRunner to attempt to access an address that is not owned by the process. 1% of all operations. Your access() function must detect this "seg fault", end the offending process by deallocating all of its memory, and returning false.
     - 4.2.3. MemCheck checks to make sure that the returned values are correct, and will print an error when they are not correct.
   - 4.3. dealloc() takes a process number, and the start address of the memory block to deallocate. Occurs 10% of the time. The requests are always valid, and are not checked.
   - 4.4. endProc() takes a process number, and should free all memory assigned to that process. Occurs 2% of the time. These are not checked directly, since there is no detectable error possible until the re-run by checkOperations().
5. Measurements
   - 5.1. CPU time starts just before constructing your MemMan object in main(), and ends after your class deals with all of the operations. Thus, your destructors will not be called during CPU time.
     - 5.1.1. You may not have any global variables since they would be constructed before CPU time starts.
   - 5.2. "Your RAM" is the largest (start address + size) you ever return from alloc().
   - 5.3. "Real RAM" is the maximum size of real dynamic RAM to which your program ever grew.
6. Grading

6.1. The program will be tested using three 100 process, 400000 operation files. The measurements will be the total of the three runs.
6.2. Programs must be compiled without any optimization options. You may not use any precompiled code, including the STL and assembly. Other than Weiss files, you must have written all of the code you submit.
6.3. If there are ANY error messages, then the program will receive zero.
6.4. CPU Time score = min(23, 20 * Sean's CPU / Your CPU)
6.5. Your RAM score = min(17, 15 * Sean's RAM / Your RAM)
6.6. Real RAM score = min(17, 15 * Sean's Real RAM / Your Real RAM)

7. Makefile. The Makefile provided contains the minimum needed to create MemRunner.out. Note that it expects MemCheck.o to already exist.
   7.1. To ensure that you handin all of the files necessary for your program, you should create a temp directory, and copy only *.cpp, *.h, MemCheck.o, and Makefile into it. Then try to make the program. Many students have forgotten to handin dsexceptions.h.
8. Suggestions.
   8.1. Start early, and get something working without errors.
   8.2. Don't fuss about speed or size until you have something working. Too many students fail to have a correct program by the deadline because they spend too much time tweaking early on.
      8.2.1. You will learn a lot just getting it running. Then you can use this knowledge to improve your code. I changed ADTs in 15 minutes at one point because the basics could be reused.
   8.3. Create MemBlock class(es) that is/are stored in your ADTs.
   8.4. To debug, add if-statements that are only true during the state in which you are interested. I left such a line in the loop in main(). I created a couple of functions in MemCheck that are useful for debugging : printOwner(address, endAddress) and printCurrentAllocations(proc). I've left commented out calls to these in strategic locations in MemMan.cpp.
   8.5. Leave lots of time for testing and debugging. Test with all of the available files.

```
[ssdavis@lect1 p4]$ MemCreate.out
Processes: 5
Operations: 25
Seed: 4
[ssdavis@lect1 p4]$ cat Mem-5-25-4.txt
50 5 25
N 3 0 6
N 4 0 9
A 3 0
D 3 0
A 4 0
A 4 0
A 4 0
A 4 0
N 4 1 4
N 4 2 4
N 0 0 7
F 4
A 0 0
N 1 0 13
N 3 1 5
A 1 0
E 1
A 0 0
E 0
N 3 2 20
N 4 0 21
N 1 0 4
D 3 1
A 3 2
A 1 0
[ssdavis@lect1 p4]$
[ssdavis@lect1 p4]$ MemRunner.out Mem-5-25-4.txt 1
Opnum: 0 alloc: proc: 3 address: 0 size: 6
Opnum: 1 alloc: proc: 4 address: 6 size: 9
Opnum: 2 access: proc: 3 address: 1
Opnum: 3 daAlloc: proc: 3 startAddress: 0
Opnum: 4 access: proc: 4 address: 13
Opnum: 5 access: proc: 4 address: 6
Opnum: 6 access: proc: 4 address: 13
Opnum: 7 access: proc: 4 address: 11
Opnum: 8 alloc: proc: 4 address: 15 size: 4
Opnum: 9 alloc: proc: 4 address: 0 size: 4
Opnum: 10 alloc: proc: 0 address: 19 size: 7
Opnum: 11 access: proc: 4 address: 21
Opnum: 11 endProc: proc: 4
Opnum: 12 access: proc: 0 address: 20
Opnum: 13 alloc: proc: 1 address: 26 size: 13
Opnum: 14 alloc: proc: 3 address: 0 size: 5
Opnum: 15 access: proc: 1 address: 33
Opnum: 16 endProc: proc: 1
Opnum: 17 access: proc: 0 address: 21
Opnum: 18 endProc: proc: 0
Opnum: 19 alloc: proc: 3 address: 5 size: 20
Opnum: 20 alloc: proc: 4 address: 25 size: 21
Opnum: 21 alloc: proc: 1 address: 46 size: 4
Opnum: 22 daAlloc: proc: 3 startAddress: 0
Opnum: 23 access: proc: 3 address: 12
Opnum: 24 access: proc: 1 address: 48
CPU Time: 0.001431 Ideal RAM: 50 Your RAM: 50 Real RAM: 688300
 [ssdavis@lect1 p4]$


[ssdavis@lect1 p4]$ MemRunner.out Mem-100-400000-1.txt 0
CPU Time: 0.791837 Ideal RAM: 407454 Your RAM: 1251836 Real RAM: 702358
[ssdavis@lect1 p4]$
```

```
int main(int argc, char* argv[])
{
  int RAM, procs, ops;
  char print = argv[2][0];
  Operation *operations = readFile(argv[1], &RAM, &procs, &ops);
  MemCheck memCheck(RAM, procs, ops);
  CPUTimer ct;
  initializeNew();
  ct.reset();
  MemMan *memMan = new MemMan(RAM, procs, ops, memCheck);

  for(int opNum = 0; opNum < ops; opNum++)
  {
   // if (opNum == 266086)  // example of debugging code.
     // cout << "Help\n";  // uncomment and place breakpoint here for debugging.

    switch(operations[opNum].op)
    {
      case 'F' :
        if(! fault(operations[opNum], memMan, opNum, memCheck, print))
          opNum = ops;  // error so leave
        break;
      case 'E' :
        if(! endProc(operations[opNum], memMan, opNum, memCheck, print))
          opNum = ops; // error so leave
        break;
      case 'N' : alloc(operations[opNum], memMan, opNum, memCheck, print); break;
      case 'D' : deAlloc(operations[opNum], memMan, opNum, memCheck, print); break;
      case 'A' :
        if(! access(operations[opNum], memMan, opNum, memCheck, print))
          opNum = ops; // error so leave
        break;
    } // switch
  } // while more in file

  double CPUTime = ct.cur_CPUTime();
  int maxRAM2 = maxRAM;
  int yourRAM = memCheck.checkOperations(operations);
  cout << "CPU Time: " << CPUTime << " Ideal RAM: " << RAM
    << " Your RAM: " << yourRAM << " Real RAM: " << maxRAM2 << endl;
  delete memMan;  // useful for determining final size of ADTs.
  return 0;
} // main()
[ssdavis@lect1 p4]$
```