CS 320: Lab 0
Memory Management

This activity is to be completed in lieu of a class meeting on 10/3. While it is not graded, it will form the foundation for an upcoming assignment. Think of it as a head start. Try to have this completed by next Tuesday. If you have questions, come see me!

1. The attached code provides a very rough model of a Memory Management Unit (MMU). Study the attached code to understand how it functions. Observe that:

    - It has a structure in place to simulate physical memory.
    - The program is using 'unsigned int' instead of int. Does it matter?
    - Memory is addressed (indexed) using a hexadecimal value (e.g. 0x3e). This is for notational purposes only. Recall that hex is just a shorthand notation for an integer – which, from the machine's perspective is just a binary number.
    - Each location in memory is allocated a string (which is an arbitrary size) --- this is fake for the purposes of the simulation. In reality, each address would hold a byte of data.

    Note: If you want, you should feel free to adjust the formatting of the memoryDump function. It worked well (enough) for my screen, if it doesn't work for you, modify it.

    How much physical memory (in bytes) does the simulation represent?

2. Assume that each process is allocated a contiguous block of 256 bytes. For the sake of our simulation, assume that each new process would be comprised of Code: between 15-60 bytes; Heap: between 10-90 bytes and Stack 50-100 bytes.

    How many processes can fit into our simulated memory using this scheme?

    - Add some code that allocates memory consistent with the parameters above for as many processes as you can fit into memory. For the "owner" you should identify a process id and a segment id. For example, part of your memory may look like:

```
0000:PID1:code  0001:PID1:code  0002:PID1:code  0003:PID1:code  0004:PID1:code  0005:PID1:code  0006:PID1:code  0007:PID1:code
0008:PID1:code  0009:PID1:code  000a:PID1:code  000b:PID1:code  000c:PID1:code  000d:PID1:code  000e:PID1:code  000f:PID1:code
0010:PID1:code  0011:PID1:code  0012:PID1:code  0013:PID1:code  0014:PID1:code  0015:PID1:code  0016:PID1:code  0017:PID1:code
0018:PID1:code  0019:PID1:code  001a:PID1:code  001b:PID1:code  001c:PID1:code  001d:PID1:code  001e:PID1:code  001f:PID1:code
0020:PID1:code  0021:PID1:code  0022:PID1:heap  0023:PID1:heap  0024:PID1:heap  0025:PID1:heap  0026:PID1:heap  0027:PID1:heap
0028:PID1:heap  0029:PID1:heap  002a:PID1:heap  002b:PID1:heap  002c:PID1:heap  002d:PID1:heap  002e:PID1:heap  002f:PID1:heap
0030:PID1:heap  0031:PID1:heap  0032:PID1:heap  0033:PID1:heap  0034:PID1:heap  0035:PID1:heap  0036:PID1:heap  0037:PID1:heap
0038:PID1:heap  0039:PID1:heap  003a:Free       003b:Free       003c:Free       003d:Free       003e:Free       003f:Free
0040:Free       0041:Free       0042:Free       0043:Free       0044:Free       0045:Free       0046:Free       0047:Free
0048:Free       0049:Free       004a:Free       004b:Free       004c:Free       004d:Free       004e:Free       004f:Free
0050:Free       0051:Free       0052:Free       0053:Free       0054:Free       0055:Free       0056:Free       0057:Free
0058:Free       0059:Free       005a:Free       005b:Free       005c:Free       005d:Free       005e:Free       005f:Free
0060:Free       0061:Free       0062:Free       0063:Free       0064:Free       0065:Free       0066:Free       0067:Free
0068:Free       0069:Free       006a:Free       006b:Free       006c:Free       006d:Free       006e:Free       006f:Free
0070:Free       0071:Free       0072:Free       0073:Free       0074:Free       0075:Free       0076:Free       0077:Free
0078:Free       0079:Free       007a:Free       007b:Free       007c:Free       007d:Free       007e:Free       007f:Free
0080:Free       0081:Free       0082:Free       0083:Free       0084:Free       0085:Free       0086:Free       0087:Free
0088:Free       0089:Free       008a:Free       008b:Free       008c:Free       008d:Free       008e:Free       008f:Free
0090:Free       0091:Free       0092:Free       0093:Free       0094:Free       0095:Free       0096:Free       0097:Free
0098:Free       0099:Free       009a:Free       009b:Free       009c:Free       009d:Free       009e:Free       009f:Free
00a0:Free       00a1:Free       00a2:Free       00a3:Free       00a4:Free       00a5:Free       00a6:Free       00a7:Free
00a8:Free       00a9:Free       00aa:Free       00ab:Free       00ac:Free       00ad:Free       00ae:Free       00af:Free
00b0:Free       00b1:Free       00b2:Free       00b3:Free       00b4:Free       00b5:Free       00b6:Free       00b7:PID1:stack
00b8:PID1:stack 00b9:PID1:stack 00ba:PID1:stack 00bb:PID1:stack 00bc:PID1:stack 00bd:PID1:stack 00be:PID1:stack 00bf:PID1:stack
00c0:PID1:stack 00c1:PID1:stack 00c2:PID1:stack 00c3:PID1:stack 00c4:PID1:stack 00c5:PID1:stack 00c6:PID1:stack 00c7:PID1:stack
00c8:PID1:stack 00c9:PID1:stack 00ca:PID1:stack 00cb:PID1:stack 00cc:PID1:stack 00cd:PID1:stack 00ce:PID1:stack 00cf:PID1:stack
00d0:PID1:stack 00d1:PID1:stack 00d2:PID1:stack 00d3:PID1:stack 00d4:PID1:stack 00d5:PID1:stack 00d6:PID1:stack 00d7:PID1:stack
00d8:PID1:stack 00d9:PID1:stack 00da:PID1:stack 00db:PID1:stack 00dc:PID1:stack 00dd:PID1:stack 00de:PID1:stack 00df:PID1:stack
00e0:PID1:stack 00e1:PID1:stack 00e2:PID1:stack 00e3:PID1:stack 00e4:PID1:stack 00e5:PID1:stack 00e6:PID1:stack 00e7:PID1:stack
00e8:PID1:stack 00e9:PID1:stack 00ea:PID1:stack 00eb:PID1:stack 00ec:PID1:stack 00ed:PID1:stack 00ee:PID1:stack 00ef:PID1:stack
00f0:PID1:stack 00f1:PID1:stack 00f2:PID1:stack 00f3:PID1:stack 00f4:PID1:stack 00f5:PID1:stack 00f6:PID1:stack 00f7:PID1:stack
00f8:PID1:stack 00f9:PID1:stack 00fa:PID1:stack 00fb:PID1:stack 00fc:PID1:stack 00fd:PID1:stack 00fe:PID1:stack 00ff:PID1:stack
```

- Clearly, this strategy will result in ***internal fragmentation***. Add some additional code to determine how much (Express this as a percentage of the total memory).

3. Clear the memory and let's try an alternate strategy: Segmentation. Instead of allocating a memory in a contiguous block, we can try to satisfy the processes' memory needs as individual segments. You should sense that you are effectively trading off internal fragmentation for more processes in memory.
   - See how many more processes you can fit into memory using the parameters above.
   - You should run your code a couple of times to get an average (Since the allocations are no longer a uniform size, your results may vary).
4. If step 3 seemed like a bit of a farce, you have good intuition. Our goal isn't to see how many processes we can shove into memory and then stop. Moreover, once a process gets into memory, it doesn't stay there forever. Notice that there is a "freeBlock" command. We can use this to simulate what happens when a block of memory is returned to the OS. That complicates things, because now we have to keep track of free memory… but that is a job for another day.