PROJECT 5
Swapping and Paging

You are free to code your solutions using any tools or operating system you choose, but be aware that your project will be graded using OS X 10.9 and/or Ubuntu 12.04. You are responsible for ensuring that your code compiles and runs correctly on these operating systems using standard APIs.

PART ONE
(100 points)

Swapping
The purpose of this project is to simulate a memory swapping system – and illustrate the principle of fragmentation - on a fictitious computer with the following specifications:

1. Your computer has exactly 1040 "bytes" of memory.
2. The operating system kernel – identified by the process name "@" always takes up the first 120 bytes of memory.
3. The computer must support at least 60 processes (including the kernel) that are identified by a one-character name: "a-z", "A-Z", and "1-9". *(Omit the lowercase letter "L" and the uppercase letter "I" from the list of processes, because they look too much alike.)* The processes all share certain characteristics:
    a. We will use a simple approach to segmentation – each process will have only one "segment" of memory assigned to it.
    b. Each process uses between 4 and 160 bytes of memory. The exact amount of memory each process uses will be selected at random, but you will "weight" the random function so that very few processes use large amounts of memory. We will discuss approaches to this in class.
    c. Processes will not all be in memory at the same time. Therefore, the computer must simulate a "backing store" for processes that are or need to be swapped out of memory. When a process is swapped out of memory, there is no minimum or maximum amount of time it has to wait before it comes back in. You can implement the backing store in a simple FCFS manner.
    d. Processes loaded into memory will be presumed to stay in memory and execute for a certain "burst" time and then become "idle", at which time they can – *but are not required to* – be swapped out. The "burst" time will be a random value between 100 and 5000 clock cycles. You may set the "burst" time for each process at your simulator's startup or a new value can be chosen every time a process is swapped back in. This project is about memory swapping, not CPU scheduling, so you need not write a full-blown scheduler to simulate processes executing on the CPU. You only need to keep track of how long the process needs to remain in memory and when it would be eligible to be removed from memory.
4. The simulator will print a "memory map" of all 1040 bytes and what process is using each byte of memory at set intervals (see below).
5. When the program starts, you will prompt the user to choose which memory allocation method is used:
    a. First-fit
    b. Best-fit
    c. Worst-fit
6. If the ratio of free memory fragments to loaded processes exceeds a certain ratio (defined below, and likely to need to be changed) then a memory compaction routine should be run. The simulator should clearly announce out that the limit is exceeded and compaction is necessary. It should also print out the memory map before AND after compaction is performed.

DESIGN CONSTRAINTS
1. You may program this project using C++ or Java. Name your file containing the main method "partone.cpp" or "PartOne.java".
2. You must use a STRUCT (C++) or a CLASS (Java) to represent the data about a process. We will discuss in class what information about each process must be stored, and you will be responsible for this information.
3. You must use reasonable data structures such as queues and/or linked lists to store the various lists of processes and their assignments. You may use the built-in data structures in the Java Collection or standard C/C++ libraries.
4. You will design your program so that it has certain CONSTANT values at the top of the file that contains the main method. If you use C++, we recommend you use #define. If you use Java, they must be public, static, and final. These values must be easily changeable at run time to observe the effects of the simulator. It may be necessary to massage these values to get good performance out of the simulator. These constants are:

```
#define MAX_PROCESSES 60     // This will not ever change
#define PROCESS_COUNT 60     // useful when debugging to limit # of procs
```

```
        #define MIN_BURST 10
        #define MAX_BURST 200
        #define MIN_MEMORY_PER_PROC 10
        #define MAX_MEMORY_PER_PROC 250
        #define MAX_MEMORY 1040
        #define MAX_BLOCK_PROC_RATIO 0.5
        #define ENABLE_COMPACTION 1 // Boolean flag for whether compaction is on/off
        #define PRINT_INTERVAL 500  // # of cpu quanta between memory map printouts
        #define MAX_QUANTA 50000    // # quanta to run before ending simulation.
        #define SLEEP_LENGTH 2500   // Used with the usleep()to slow down sim between
                                    // cycles (makes reading screen in real-time easier!)
```

SAMPLE OUTPUT

```
QUANTA ELAPSED: 5000
MEMORY: 1040b       USED: 764b (73.5%)       FREE: 276b (26.5%)
PROCESSES: 60       LOADED: 43 (71.6%)       UNLOADED: 26 (28.4%)
FREE BLOCKS: 32     LARGEST: 54b   SMALLEST: 2b   BLOCKS/PROCS RATIO: 0.7441
         9        19        29        39        49        59        69        79
----+----|----+----|----+----|----+----|----+----|----+----|----+----|----+----|
@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@
        89        99       109       119       129       139       149       159
----+----|----+----|----+----|----+----|----+----|----+----|----+----|----+----|
@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@ppppppppppppppSSSSSSSSSSSSSSSS     eeeee
       169       179       189       199       209       219       229       239
----+----|----+----|----+----|----+----|----+----|----+----|----+----|----+----|
eeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeee  FFFFFFFFFFFF    ZZZZZZZZZZZZZZZZZZZZZZ
       249       259       269       279       289       299       309       319
----+----|----+----|----+----|----+----|----+----|----+----|----+----|----+----|
ZZZZZ cccccccccccccccc           ppppp      BBBBBBBB     MMMMMMMMuuuuuuuuuuuuuuu
       329       339       349       359       369       379       389       399
----+----|----+----|----+----|----+----|----+----|----+----|----+----|----+----|
uuuuuuuuuuuuuuuuuuuuYYYYYYYYYY  gggggggggggggWWWWWWWWWWWWWWWWWWWWWWqqqqqqqqqqqqq

< MEMORY MAP GOES ON AND ON UNTIL 1040 >
```

HINTS

1. Use the ASCII table and the principle that all letters are really stored as numbers to make printing the process "name" easier.

PART TWO

(100 points)

Instructions

You will implement paging using a similar scheme as before. Recall that paging, unlike swapping, allows the memory of a process to be broken up into separate "frames" of a fixed size.
Your paging simulation will adhere to the following rules

1. **Frames.** Your computer has exactly 280 "frames" of memory. Each "frame" of memory is 4KB in size.
2. **Backing Store.** Your computer has a backing store of exactly 720 "pages" of memory. The backing store MUST be implemented as a vector in memory with actual pages in actual positions and managed on a page-by-page basis. You can't just lump everything not in memory into a queue or list and call that your backing store.
3. **Kernel**. The operating system kernel – identified by the process name "@" always takes up exactly 20 frames of memory.
4. **Processes**. The computer must support at least 22 processes (plus the kernel) that are identified by a one-character name: "A-V". The processes all share certain characteristics:
    a. **Segments.** Each process will be divided into several segments, each of which will be paged. The division of processes into segments is as follows:
        i. **Code Segment**. A code segment will be comprised of 2 frames. All code segments will be identified by the suffix "0", therefore the pages/frames of the code segment for process "A" will be listed as "A0" in the memory map.
        ii. **Stack Segment**. The stack segment will be comprised of 3. All Stack segments will be identified by the suffix "1".
        iii. **Heap Segment**. The stack segment will be comprised of 5. All Stack segments will be identified by the suffix "2".

iv. **Subroutine Segments**. The process will have between 1 and 5 subroutine segments each of which are two frames in size. You may choose how many subroutine segments each process has at random at process creation time. The subroutine segments will be identified by the suffixes "3" to "7".

b. Processes will not all be in memory at the same time and all of the pages of each process may not be in memory at the same time.

c. **Touching Processes**. You will "touch" a process at random every time quanta. When you "touch" a process, you will access its code, stack, heap, and one subroutine segment (at random). If the pages for those segments are not in memory, you must simulate a page fault and load them.

d. **Process Death**. Because a real system has processes being created and destroyed all the time, we need to have a way to terminate and create processes as well. The method for handling death is as follows:

   i. When a process is created, you will pick a "lifetime" expiration time of 20-300 cycles from the current cycle (this range will be set in a constant, see below).

   ii. When its "lifetime" expires, that process ends and is removed from memory (both physical memory and the backing store).

   iii. The page table entry for a non-existent process letter simply reverts to "blanks" and "dashes."

e. **Process Creation**. You will also need to create processes occasionally to replace dead ones. The method for this is simple – if your random "touch" function (part "c" above) selects a process letter that doesn't exist, create a new process as described in part "a" above and proceed normally.

5. **Paging and the Page Table**. You will have to implement paging and a page table. Your page table implementation can be very simple – each process can have a **linked list** that stores the page table entries for the process. The details of the page table are as follows:

   a. The page table size will ALWAYS be 20 pages in size (the maximum number of pages that any process can have!).

   b. Your page table must implement the following:

      i. valid/invalid bit.

      ii. A reference byte for each page (to be used in LRU and Second Chance, see below).

6. The simulator will print a "memory map" of all 280 frames of memory, the page table of each process, and the entire 720 pages on the backing store at set intervals (see below).

7. When the program starts, you will prompt the user to choose which page replacement method is used:

   a. FIFO

   b. LRU

   c. Second Chance

   Note: For the LRU and Second Chance algorithms, use an 8-bit reference bit value. For LRU, all 8 bits refer to access in time periods. The value can propogate (i.e. bit shift) every 10 cycles. For Second Chance, the highest 7 bits refer to access in time periods and the lowest bit is always the second chance bit.

8. You will again include certain CONSTANT values at the top of the main file, as in Part 1. These constants are:

```
#define MAX_PROCESSES 52    // This will not ever change
#define PROCESS_COUNT 23    // useful when debugging to limit # of procs
#define MIN_DEATH_INTERVAL 20
#define MAX_DEATH_INTERVAL 300
#define MAX_FRAMES 280
#define MAX_PAGES 720
#define SHIFT_INTERVAL 10
#define PRINT_INTERVAL 500  // # of cpu quanta between memory map printouts
#define MAX_QUANTA 50000    // # quanta to run before ending simulation.
#define SLEEP_LENGTH 2500   // Used with the usleep()to slow down sim between
                            // cycles (makes reading screen in real-time easier!)
```

SAMPLE OUTPUT

NOTE: THE SAMPLE OUTPUT USES TWO SPACES FOR EACH "FRAME" SO YOU CAN PRINT "A0", FOR EXAMPLE.
NOTE: THE SAMPLE OUTPUT IS 120 COLUMNS WIDE, AND YOURS SHOULD BE TOO.
NOTE: THE KEY FOR THE PAGE TABLES IS AS FOLLOWS: 00 (PAGE) 098 (FRAME) vA8 (valid bit, Hex value of ref byte)

```
QUANTA ELAPSED: 500
FRAMES: 280f         USED: 149f (xx.x%)   FREE: 81f (xx.x%)
SWAP SPACE: 720p     PAGES: 469p (xx.x%)   LOADED: 149p (xx.x%)  FREE: 320p (xx.x%)
PROCESSES: 23      LOADED: 15 (xx.x%)   UNLOADED: 5 (xx.x%)   DEAD: 3 (xx.x%)

PHYSICAL MEMORY (FRAMES)
        04        09        14        19        24        29        34        39        34        39        44        49
--------++--------||--------++--------||--------++--------||--------++--------||--------++--------||--------++--------||
A1D5F7  E3@@Q1Q2@@@@    F2F5R1Y2Y1@@  @@    P1O3O1@@  @@@@T1D3T2D4T5D6@@  @@B2B3C3@@  W1V3@@  @@U1I2I1@@  @@@@@@  D6D0G1
        54        59        64        69        74        79        84        89        94        99       104       109
--------++--------||--------++--------||--------++--------||--------++--------||--------++--------||--------++--------||
< MEMORY MAP GOES ON AND ON UNTIL 280 FRAMES >
       114       119       124       129       134       139       144       149       154       159       164       169
--------++--------||--------++--------||--------++--------||--------++--------||--------++--------||--------++--------||
< MEMORY MAP GOES ON AND ON UNTIL 280 FRAMES >
```

```
PAGE TABLES (NOTE: I have only filled in simulated data for process "A")
    A          B                     D          E          F          G                     I          J          K
00 098 vA8|00 000 ---|-- --- ---|00 000 ---|00 000 ---|00 000 ---|00 000 ---|-- --- ---|00 000 ---|00 000 ---|00 000 ---
01 002 vFF|01 000 ---|-- --- ---|01 000 ---|01 000 ---|01 000 ---|01 000 ---|-- --- ---|00 000 ---|00 000 ---|00 000 ---
02 000 iA3|02 000 ---|-- --- ---|02 000 ---|02 000 ---|02 000 ---|02 000 ---|-- --- ---|00 000 ---|00 000 ---|00 000 ---
03 145 v--|03 000 ---|-- --- ---|03 000 ---|03 000 ---|03 000 ---|03 000 ---|-- --- ---|00 000 ---|00 000 ---|00 000 ---
04 133 i--|04 000 ---|-- --- ---|04 000 ---|04 000 ---|04 000 ---|04 000 ---|-- --- ---|00 000 ---|00 000 ---|00 000 ---
  . . .   |  . . .   |  . . .   |  . . .   |  . . .   |  . . .   |  . . .   |  . . .   |  . . .   |  . . .   |  . . .
19 000 i--|19 000 i--|-- --- ---|19 000 i--|19 000 i--|19 000 i--|19 000 i--|-- --- ---|19 000 i--|19 000 i--|19 000 i--

    L          M          N          O          P          Q                     S          T          U          V
00 098 v--|00 000 ---|00 000 ---|00 000 ---|00 000 ---|00 000 ---|-- --- ---|00 000 ---|00 000 ---|00 000 ---|00 000 ---
  . . .   |  . . .   |  . . .   |  . . .   |  . . .   |  . . .   |  . . .   |  . . .   |  . . .   |  . . .   |  . . .
19 000 i--|19 000 i--|19 000 i--|19 000 i--|19 000 i--|19 000 i--|-- --- ---|19 000 i--|19 000 i--|19 000 i--|19 000 i–

BACKING STORE (PAGES)
        04        09        14        19        24        29        34        39        34        39        44        49
--------++--------||--------++--------||--------++--------||--------++--------||--------++--------||--------++--------||
<LOOKS JUST LIKE THE MEMORY MAP, BUT CONTAINS ALL PAGES)
        54        59        64        69        74        79        84        89        94        99       104       109
--------++--------||--------++--------||--------++--------||--------++--------||--------++--------||--------++--------||
<LOOKS JUST LIKE THE MEMORY MAP, BUT CONTAINS ALL PAGES)
       114       119       124       129       134       139       144       149       154       159       164       169
--------++--------||--------++--------||--------++--------||--------++--------||--------++--------||--------++--------||
< MAP GOES ON AND ON UNTIL 720 FRAMES >
```

Name the file containing your main method for this part of the project **PartTwo.java** or **parttwo.cpp**

PART THREE

(50 points)

Instructions

Answer the following questions in detail and in complete sentences. Be sure to justify your answers with data, when necessary. Store your responses in a file names **partthree.txt**.

1. What are the maximum, minimum, and average amount of external fragmentation over time in Part One for each of the three memory allocation methods (first-fit, best-fit, and worst-fit). Show data to support your decisions.
2. When memory compaction is enabled, how often does compaction occur with an activation threshold of 0.35, 0.5, and 0.75 for each of the three memory allocation methods?
3. Of the three paging algorithms in Part Two – FIFO, LRU, and Second Chance – which one results in the lowest number of page faults in your simulation? What are the average page fault rates for each of the three?
4. If you increase the number of processes to the maximum allowed in Part Two – what do the page fault rates then become?

PROJECT SUBMISSION

1. ALL SOURCE CODE YOU TURN IN MUST CONTAIN THE FOLLOWING AT THE TOP:

```
    // CS3242 Operating Systems
    // Fall 2013
    // Project 5: Swapping and Paging, Part 1
    // John S. Doe and Bob A. Smith
    // Date: 9/23/2013
    // File: partone.cpp
```

2. Zip ALL source code files for your project into a single ZIP file named "DOE_SMITH.ZIP" (where Doe and Smith are the surnames of the two students) and upload that as your submission in Dropbox on D2L by the posted deadline.