```
/*     Authors: Austin Blythe, Matthew Francis
 *       Acct#: cs441104
 *       Class: CSC 441, Dr. Stader
 *  Start Date: 20 Mar 2013
 * Finish Date: 8 Apr 2013
 * Description: This program builds on OS_Part 3, implementing queues
 */

#include <stdlib.h>
#include <stdio.h>

// Global Constants
   const int OPCODE_LEN = 4;     // Opcode field length
   const int MODE_LEN = 1;       // Mode field length
   const int REG_LEN = 3;        // Register field length
   const int ADDRE_LEN = 8;      // Address field length
   const int MAX_BITS = 16;      // Size of a single memory location in bits
   const int TICKS_PER_USER = 4; // # of Ticks allowed per cycle for user

#define MEMORY_LENGTH 4
#define MAX_MEMORY 256        // Size of memory array
#define MAX_REGISTER 4        // Number of registers in the machine
#define DISK_SIZE 256         // Size of the disk
#define true 1                // Setting keyword true to 1
#define false 0               // Setting keyword false to 0
#define numberOfUsers 3       // Number of users in the system
#define numberOfProcesses 3   // Number of simultaneous processes (possible) in the system

// Global Varibles
   typedef int bool;             // Create bool type as C does not have one

   unsigned short CC;            // Condition code
   unsigned short PC;            // Program Counter
   unsigned short IR;            // Instruction Register

   unsigned short opcode;        // Opcode field
   unsigned short mode;          // Mode field
   unsigned short reg;           // Register field
   signed short address;         // Address field



// Internal
   bool haltFlag;                // Controls system halt
   int programClock;             // Internal programClock
   bool stopOS;                  // Changes to true if stp is issues
   int currentTick;              // Current tick that the user is on (resets to 0 when limit
   reached)
   int currentUser;              // Current user in the RR
   int currentPriority;          // Which process array is being used
   int currentPosition;          // Position in the queueArray
```

```
// Structure format creation for users and O/S
   struct user
   {
      int memoryLocation;        // Starting location for that users program
      bool hasProcess;           // If true, user has a process in the queue / loaded into memory
      int progLength;            // Number of instructions in the user's program
      int pageTable[MAX_MEMORY / MEMORY_LENGTH];  // Page table for the user owned process(s),
      allows for largest possible load.
      int myPriority;
   };

   struct user userArray[numberOfUsers];  // Array for creation of users + 1 to allow extra spot

   struct processBlock
   {
      int pid;                   // Stores who owns the process
      int processType;           // Determines if it is a run(1), dmp(2), or stp(3) command
      unsigned short pCounter;   // Stores what memory location the program is currently at
      bool isRunning;            // Used to determine locked/queue status
      bool isComplete;           // Used for cleanUp method to shift queue
      int executionTime;
      unsigned short processCC;
      signed short processRegisters[MAX_REGISTER];
      int pageTick;
      int frameTick;
   };

   struct processBlock processArray[numberOfProcesses]; // Process queue, only one can be
   processes at once currently f

   struct processBlock queueArray[2][numberOfProcesses];
   int queueOnePosition;
   int queueOneShadowPosition;
   struct processBlock queueShadow[2][numberOfProcesses];
   int queueTwoPosition;
   int queueTwoShadowPosition;

   unsigned short disk[DISK_SIZE];          // 1D-array of short int
   unsigned short mainMemory[MAX_MEMORY];  // Main memory
   int usedFrames[MAX_MEMORY / MEMORY_LENGTH];         // Frame usage bit vector
   signed short Registers[MAX_REGISTER];  // Registers array (0 is Accumulator)


   char userIn[5] = {0};                  // Array for user input
   char *controlCommand = userIn;         // Pointer to userIn array
   int commandCode;                       // Codes for UI commands
// run (1), dmp(2), stp(3)

// Method Declarations
   int main(void);

// Operating System
   void initializeOS();
```

```c
   void loader();
   void scheduler();
   void dispatcher();
   void userInterface();
   void interpreter();
   void cleanUp();                       // Shifts the process queue as necessary

// UI Commands
   void run();
   void dmp();
   void stp();

// Memory
   void mmu(int, int);

// Interpreter
   void Fetch();
   void Decode();
   void Execute();

// Functions
   void dumpPageTable();
   unsigned short convertNumber(char *);
   void printBin(unsigned short);
   void printHex(unsigned short);
   void changeCondition(int);
   bool isValidCommand();
   void placeInQueue(); // Pass the process block and the queue number
   void placeInShadow(); // Place current process in shadow queue
   bool queueHasProcess();
   void dumpQueues();
   void promoteProcesses();
   void updateQueuePositions();

// Instruction Set
   void load();
   void store();
   void add();
   void sub();
   void adr();
   void sur();
   void and ();
   void or ();
   void not();
   void jmp();
   void jeq();
   void jgt();
   void jlt();
   void compare();
   void clear();
   void halt();

// User-defined header files:
```

```
#include "instructions.h"  // Needs to be below variable declarations

// ******************** MAIN ********************

   int main (void)
   {

    // OS Initialization
      initializeOS();

    // User Interface loop
      while (stopOS == false)
      {
         userInterface();
      }

      printf("\n\n---------------------------------\n\tMACHINE
      HALTED\n---------------------------------\n\n");

      return 0;
   }


// ******************** OPERATING SYSTEM ********************

   void initializeOS()
   {
    // Initialize Values
      programClock = 0;
      CC = 0;
      PC = 0;
      IR = 0;
      haltFlag = false;
      stopOS = false;
      currentUser = 1;  // Starts with user 1
      currentTick = 0;  // User 1 starts with 0 ticks on their cycle
      commandCode = 0;
      queueOnePosition = 0;
      queueTwoPosition = 0;

    // Zero out the mainMemory and map frame locations
    // Initialize the usedFrames array so they are all available

      int p = 0;
      for (p; p < MAX_MEMORY; p++)
      {
         mainMemory[p] = 00;
         if (p % MEMORY_LENGTH == 0)
         {
            usedFrames[p / MEMORY_LENGTH] = 0;
         }
      }
      int i, j;
    // Initialize page table
      for (j = 1; j < numberOfUsers; j++)
      {
         for (i = 0; i < MAX_MEMORY / MEMORY_LENGTH; i++)
         {
            userArray[j].pageTable[i] = -1;
         }
      }

    // Zero out the disk
      p = 0;
      for (p; p < DISK_SIZE; p++)
      {
         disk[p] = 0x0000;
      }

    // Set pid for processes to -1
      p = 0;
      for (p; p < numberOfProcesses; p++)
      {
         queueArray[0][p].pid = -1;
         queueArray[1][p].pid = -1;
         queueShadow[0][p].pid = -1;
         queueShadow[1][p].pid = -1;
      }


    // User programs on the disk
    // User 1 data set
      disk[0]   = 0x080A; // Location 000 // Load Immediate R0 #10
      disk[1]   = 0x1006; // Location 001 // Store R0 6
      disk[2]   = 0x0905; // Location 002 // Load Immediate R1 #5
      disk[3]   = 0x4100; // Location 003 // AddR R1
      disk[4]   = 0x1007; // Location 004 // Store R0 7
      disk[5]   = 0xF000; // Location 005 // Halt

    // User 2 data set
      disk[100] = 0x0819; // Location 100 // LOAD I R0 #25
      disk[101] = 0x1006; // Location 101 // STO R0 6
      disk[102] = 0x0905; // Location 102 // LOD I R1 #5
      disk[103] = 0x5100; // Location 103 // SUR R1
      disk[104] = 0x1007; // Location 104 // STO R0 7
      disk[105] = 0xF000; // Location 105 // HALT


    // Create user(s)
    // OS
      userArray[0].memoryLocation = 0;
      userArray[0].myPriority = 0;
      userArray[0].hasProcess = false;

    // User1
```

```
      userArray[1].memoryLocation = 0;
      userArray[1].progLength = 6;
      userArray[1].hasProcess = false;
      userArray[1].myPriority = 1;

    // User2
      userArray[2].memoryLocation = 100;
      userArray[2].progLength = 6;
      userArray[2].hasProcess = false;
      userArray[2].myPriority = 1;

   }



// Interactive command-line user interface
   void userInterface()
   {
      bool processRequests = false;

      printf("\n\nINPUT REQUESTS\n--------------\n");

    // Get a request from each user and OS
      while (processRequests == false)
      {
         if (currentUser == 0)
         {
            printf("\nO/S:");
            processRequests = true;
         }
         else printf("\nUser %i:", currentUser);

         if (userArray[currentUser].hasProcess == false)
         {

            printf("\n\tPlease enter a command: ");
            fgets(controlCommand, 5, stdin);

            if (isValidCommand())
            {
               placeInQueue(); // place run, dmp, and stp requests in the proper queue

               if (currentUser == numberOfUsers - 1) currentUser = 0;
               else currentUser++;
            }
            else
            {
               printf("\n\tInvalid command entered\n");
               processRequests = false;
            }
         }
         else
         {
            printf("\n\tYou already have a process queued\n");
            if (currentUser == numberOfUsers - 1) currentUser = 0;
            else currentUser++;
         }
      }

      scheduler();   // Process a single request based on priority

      if (queueArray[currentPriority][0].isRunning == true)
         cleanUp(2);
      else
         cleanUp(1);
   }

// Process a request based on priority
   void scheduler()
   {
      printf("\n\nPROCESS REQUEST\n---------------\n");
      currentTick = 0;
      if (queueHasProcess(0))
      {
         currentPriority = 0;
      }
      else if (queueHasProcess(1))
      {
         currentPriority = 1;
      }
      else
         printf("\n\tBoth queues are empty; no operations performed\n");

      dispatcher();
   }

// Directs users/OS based on command entered
   void dispatcher()
   {
      switch (queueArray[currentPriority][0].processType)
      {
         case 1: run();
            break;
         case 2: dmp();
            break;
         case 3: stp();
            break;
      }
   }


// responsible for the machine languare interpretation and execution
   void interpreter()
   {
      if (queueArray[currentPriority][0].pid > 0) printf("\nUser %d Running...\n", queueArray[
      currentPriority][0].pid);
      while (haltFlag == false && currentTick < TICKS_PER_USER)
      {
         Fetch();
         Decode();
         mmu(-1, -1);
         Execute();
         if (queueArray[currentPriority][0].frameTick % MEMORY_LENGTH == 0)
         {
            queueArray[currentPriority][0].pageTick++;
            queueArray[currentPriority][0].pCounter = userArray[queueArray[currentPriority][0].
            pid].pageTable[queueArray[currentPriority][0].pageTick] * MEMORY_LENGTH;
            PC = queueArray[currentPriority][0].pCounter;
         }
      }

      if (haltFlag == true)
      {
         queueArray[currentPriority][0].isComplete = true;
         dumpPageTable();
         mmu(-2, -2);
         userArray[queueArray[currentPriority][0].pid].hasProcess = false;
         queueArray[currentPriority][0].isRunning = false;
      }
      else
      {
         queueArray[currentPriority][0].pCounter = PC;
         queueArray[currentPriority][0].processCC = CC;
         CC = 0x0000;
         int i = 0;
         for (i; i < MAX_REGISTER; i++)
            queueArray[currentPriority][0].processRegisters[i] = Registers[i];
         Registers[i] = 0x0000;
      }
      haltFlag = false; // reset halt flag for subsequent program runs
   }

   void loader()
   {
      int p, i;
      int currentFrame;
      int currentPage = 0;
      int memoryLoc;

    // Place user program pages into main memory frames
      for (p = userArray[queueArray[currentPriority][0].pid].memoryLocation;
            p < (userArray[queueArray[currentPriority][0].pid].progLength +
                 userArray[queueArray[currentPriority][0].pid].memoryLocation););
      {
         currentFrame = rand() % 64;
         memoryLoc = currentFrame * MEMORY_LENGTH;
         if (currentPage == 0)
         {
            queueArray[currentPriority][0].pCounter = memoryLoc;

         }
         if (usedFrames[currentFrame] == 0)
         {
            for (i = 0; i < MEMORY_LENGTH; i++, p++)
            {
               mainMemory[memoryLoc + i] = disk[p];
            }
            usedFrames[currentFrame] = 1;
            mmu(currentPage, currentFrame);
            currentPage++;
         }
      }

   }

// Clean up
   void cleanUp(int type)
   {
      int i = 0;
      int p = 0;


      if (type == 1) // all finished processes (dmp, stp, and second part of user programs)
      {
         userArray[queueArray[currentPriority][0].pid].hasProcess = false;

         if (currentPriority == 0) queueOnePosition--;
         if (currentPriority == 1) queueTwoPosition--;
         printf("\nUser %d process removed from queue %d\n", queueArray[currentPriority][0].pid,
          currentPriority + 1);
      }
      else // unfinished requests, move to shadow queue
      {
         placeInShadow();
      }

      for (i = 0; i < numberOfProcesses - 1; i++)
      {
         queueArray[currentPriority][i].pid = queueArray[currentPriority][i + 1].pid;
         queueArray[currentPriority][i].pCounter = queueArray[currentPriority][i + 1].pCounter;
         queueArray[currentPriority][i].isRunning = queueArray[currentPriority][i + 1].isRunning;
         queueArray[currentPriority][i].isComplete = queueArray[currentPriority][i + 1].
         isComplete;
         queueArray[currentPriority][i].processType = queueArray[currentPriority][i + 1].
         processType;
         queueArray[currentPriority][i].pageTick = queueArray[currentPriority][i + 1].pageTick;
         queueArray[currentPriority][i].frameTick = queueArray[currentPriority][i + 1].frameTick;
         queueArray[currentPriority][i].executionTime = queueArray[currentPriority][i + 1].
         executionTime;
         queueArray[currentPriority][i].processCC = queueArray[currentPriority][i + 1].processCC;
         p = 0;
         for (p; p < MAX_REGISTER; p++)
            queueArray[currentPriority][i].processRegisters[p] = queueArray[currentPriority][i +
             1].processRegisters[p];
      }

      queueArray[currentPriority][numberOfProcesses - 1].pid = -1;

      updateQueuePositions();

      dumpQueues();

      promoteProcesses();

      printf("\nCopying shadow queues\n");

      if (queueArray[0][0].pid == -1)
      {
         for (i = 0; i < numberOfProcesses - 1; i++)
         {
            queueArray[0][i].pid = queueShadow[0][i].pid;
            queueArray[0][i].pCounter = queueShadow[0][i].pCounter;
            queueArray[0][i].isComplete = queueShadow[0][i].isComplete;
            queueArray[0][i].isRunning = queueShadow[0][i].isRunning;
            queueArray[0][i].processType = queueShadow[0][i].processType;
            queueArray[0][i].pageTick = queueShadow[0][i].pageTick;
            queueArray[0][i].frameTick = queueShadow[0][i].frameTick;
            queueArray[0][i].executionTime = queueShadow[0][i].executionTime;
            queueArray[0][i].processCC = queueShadow[0][i].processCC;
            p = 0;
            for (p; p < MAX_REGISTER; p++)
               queueArray[0][i].processRegisters[p] = queueShadow[0][i].processRegisters[p];
            queueShadow[0][i].pid = -1;
         }
      }
      if (queueArray[1][0].pid == -1)
      {
         for (i = 0; i < numberOfProcesses - 1; i++)
         {
            queueArray[1][i].pid = queueShadow[1][i].pid;
            queueArray[1][i].pCounter = queueShadow[1][i].pCounter;
            queueArray[1][i].isComplete = queueShadow[1][i].isComplete;
            queueArray[1][i].isRunning = queueShadow[1][i].isRunning;
            queueArray[1][i].processType = queueShadow[1][i].processType;
            queueArray[1][i].pageTick = queueShadow[1][i].pageTick;
            queueArray[1][i].frameTick = queueShadow[1][i].frameTick;
            queueArray[1][i].executionTime = queueShadow[1][i].executionTime;
            queueArray[1][i].processCC = queueShadow[1][i].processCC;
            p = 0;
            for (p; p < MAX_REGISTER; p++)
               queueArray[1][i].processRegisters[p] = queueShadow[1][i].processRegisters[p];
            queueShadow[1][i].pid = -1;
         }
      }
      updateQueuePositions();

      dumpQueues();
   }

// ******************** UI ****************

   void run()
   {
    // Continue running a process that already started running
      if (queueArray[currentPriority][0].isRunning == true)
      {
         PC = queueArray[currentPriority][0].pCounter;
         CC = queueArray[currentPriority][0].processCC;
         int i = 0;
         for (i; i < MAX_REGISTER; i++)
            Registers[i] = queueArray[currentPriority][0].processRegisters[i];
         interpreter();
      }
      else
      {
        // First run
         loader();
         programClock++;
         currentTick++;
         PC = queueArray[currentPriority][0].pCounter;
         queueArray[currentPriority][0].isRunning = true;
         interpreter();
      }
   }

// This will create a dump of the data in the program
   void dmp()
   {

      programClock++;
      currentTick++;

      char reg_names [4] = {'A', '1', '2', '3'};
      int i = 0;

      printf("\nClock: %d\n", programClock);

      printf("\nREGISTERS\n----------------------------------\n");
      while (i < MAX_REGISTER)
      {
         printf("%1c\t", reg_names[i]);
         printHex(Registers[i]);
        //printBin(Registers[i]);
         printf("\n");
         ++i;
      }
```

```c
   printf("PC\t");
   printHex(PC);
 //printBin(PC);
   printf("\n");

   printf("CC\t");
   printHex(CC);
 //printBin(CC);
   printf("\n");

   printf("IR\t");
   printHex(IR);
 //printBin(IR);
   printf("\n");

   printf("\nMEMORY\n---------------------------------\n");

   for (i = 0; i < MAX_MEMORY; i++)
   {
      if (mainMemory[i] != 00)
      {
         printf("%-3d\t", i);
         printHex(mainMemory[i]);
         //printBin(mainMemory[i]);
         if (usedFrames[i / MEMORY_LENGTH] == 1)
            printf("LOCKED    USER X\n");
         else if (usedFrames[i / MEMORY_LENGTH] == 0)
            printf("UNLOCKED\n");
      }
   }

   printf("\nDISK\n---------------------------------\n");

   for (i = 0; i < DISK_SIZE; i++)
   {
      if (disk[i] != 0x0000)
      {
         printf("%-3d\t", i);
         printHex(disk[i]);
         //printBin(disk[i]);
         printf("\n");
      }
   }

   printf("\nSCHEDULING QUEUES\n---------------------------------\n");
   dumpQueues();

   printf("\n---------------------------------\n\tDUMP
   COMPLETE\n---------------------------------\n");

   queueArray[currentPriority][0].isComplete = true;
}
```

```
   void stp()
   {
      programClock++;
      stopOS = true;
      dmp();
   }


// ******************** MEMORY ********************

   void mmu(int page, int frame)
   {
      int pageNum;
      int offset;
      if (page == -1 && frame == -1 && opcode == 1)
      {
         pageNum = address & 252;
         pageNum = pageNum >> 2;
         offset = address & 3;
         address = (userArray[queueArray[currentPriority][0].pid].pageTable[pageNum] *
         MEMORY_LENGTH) + offset;
      }
      else if (page > -1 && frame > -1)
      {
         userArray[queueArray[currentPriority][0].pid].pageTable[page] = frame;
      }

      int i, j;
      if (page == -2 && frame == -2)
      {
        // Below is code to clean up users page table, zero's out their table (as they should
        only have one processes in queue)
         for (i = 0; i < MAX_MEMORY / MEMORY_LENGTH; i++)
         {
            for (j = 0; j < MAX_MEMORY / MEMORY_LENGTH; j++)
            {
               if (userArray[queueArray[currentPriority][0].pid].pageTable[i] == j)
               {
                  userArray[queueArray[currentPriority][0].pid].pageTable[i] = -1;
                  usedFrames[j] = 0;
               }
            }
         }
      }
   }


// ******************** INTERPRETER ********************

// Fetches next instruction from mainMemory, then increments PC
   void Fetch()
   {
      IR = mainMemory[PC];
      PC++;
   }

// Decode instructions into four fields: opcode, mode, register, address
   void Decode()
   {
      char temp[16];
      char *tempPointer = temp;

      unsigned int i = 1 << (sizeof(IR) * 8 - 1);

      int count = 0;
      int k = 0;

      while (i > 0)
      {
         if (IR & i)
            temp[k] = '1';
         else
            temp[k] = '0';
         i >>= 1;

         ++k;

         if (count == 3)
         {
            opcode = convertNumber(tempPointer);
            k = 0;
         }
         else if (count == 4)
         {
            if (temp[k - 1] == '0')
               mode = 0;
            else
               mode = 1;
            k = 0;
         }
         else if (count == 7)
         {
            temp[k] = 0;
            reg = convertNumber(tempPointer);
            k = 0;
         }
         else if (count == 15)
         {
            address = (short)convertNumber(tempPointer);
            k = 0;
         }

         ++count;
      }
   }
```

```
// Based on opcode, execute the instruction
   void Execute()
   {
      switch (opcode)
      {
         case 0:  load(mainMemory, Registers);
            break;
         case 1:  store(mainMemory, Registers);
            break;
         case 2:  add(mainMemory, Registers);
            break;
         case 3:  sub(mainMemory, Registers);
            break;
         case 4:  adr(mainMemory, Registers);
            break;
         case 5:  sur(mainMemory, Registers);
            break;
         case 6:  and (mainMemory, Registers);
            break;
         case 7:  or (mainMemory, Registers);
            break;
         case 8:  not(mainMemory, Registers);
            break;
         case 9:  jmp(mainMemory);
            break;
         case 10: jeq(mainMemory);
            break;
         case 11: jgt(mainMemory);
            break;
         case 12: jlt(mainMemory);
            break;
         case 13: compare(mainMemory, Registers);
            break;
         case 14: clear(Registers);
            break;
         case 15: halt();
            break;
      }
      programClock++;
      currentTick++;
      queueArray[currentPriority][0].frameTick++;
      queueArray[currentPriority][0].executionTime++;
   }


// ******************** FUNCTIONS ********************

// Called from halt instruction
   void dumpPageTable()
   {
      printf("\nUser %d Page Table\n", queueArray[currentPriority][0].pid);
      int h, k;
      printf("Page \t| \tFrame\n");
      for (h = 0; h < (MAX_MEMORY / MEMORY_LENGTH); h++)
      {
         if (userArray[queueArray[currentPriority][0].pid].pageTable[h] > -1)
         {
            printf("%d \t| \t%d\n", h, userArray[queueArray[currentPriority][0].pid].pageTable[h
            ]);
            printf("\n");
            for (k = 0; k < MEMORY_LENGTH; k++)
            {
               printf("\t%d:\t", userArray[queueArray[currentPriority][0].pid].pageTable[h]*
               MEMORY_LENGTH + k);
               printHex(mainMemory[userArray[queueArray[currentPriority][0].pid].pageTable[h]*
               MEMORY_LENGTH + k]);

                //printBin(mainMemory[userArray[queueArray[currentPriority][0].pid].pageTable[h]*
                MEMORY_LENGTH + k]);
               printf("\n");
            }
            printf("\n");
         }
      }
   }

// Converts the string into an unsigned short
   unsigned short convertNumber(char *num)
   {
      return (unsigned short)strtoul(num, NULL, 2);
   }

// Prints the passed integer in binary format
   void printBin(unsigned short a)
   {

      unsigned int i;
      i = 1 << (sizeof(a) * 8 - 1);
      int k = 0;

      while (i > 0)
      {
         if (a & i)
            printf("1");
         else
            printf("0");
         i >>= 1;
         ++k;
         if (k == 4)
         {
            printf(" ");
            k = 0;
         }
      }
   }
```

```c
// Prints the passed integer in hex format
   void printHex(unsigned short a)
   {
      printf("x%04X    ", a);
   }

// Sets condition code of register to positive, zero, or negative
   void changeCondition(int regValue)
   {
      if (Registers[regValue] > 0) CC = 1;
      else if (Registers[regValue] == 0) CC = 2;
      else if (Registers[regValue] < 0) CC = 4;
      else {}
   }

   bool isValidCommand ()
   {
      if (controlCommand[0] == 'r' && controlCommand[1] == 'u' && controlCommand[2] == 'n' &&
      currentUser > 0)
      {
         commandCode = 1;
         return true;
      }
      else if (controlCommand[0] == 'd' && controlCommand[1] == 'm' && controlCommand[2] == 'p'
      && currentUser == 0)
      {
         commandCode = 2;
         return true;
      }
      else if (controlCommand[0] == 's' && controlCommand[1] == 't' && controlCommand[2] == 'p'
      && currentUser == 0)
      {
         commandCode = 3;
         return true;
      }
      else if (controlCommand[0] == 'n' && controlCommand[1] == 'o' && controlCommand[2] == 'p')
      {
         commandCode = 4;
         return true;
      }
      else
         return false;
   }

   void placeInQueue()
   {
      int currentPosition; // Local variable for current position in the queue's
      if (commandCode == 2 || commandCode == 3)
      {
         currentPosition = queueOnePosition;
         queueOnePosition++;
         currentPriority = 0;
```

```
            printf("\n\tPriority %d request added\n", currentPriority + 1);
        }
        else if (commandCode == 1)
        {
            currentPosition = queueTwoPosition;
            queueTwoPosition++;
            currentPriority = 1;
            printf("\n\tPriority %d request added\n", currentPriority + 1);
        }
        else
        {
            printf("\n\tNo operation performed\n");
            return;
        }

        queueArray[currentPriority][currentPosition].pid = currentUser;
        queueArray[currentPriority][currentPosition].processType = commandCode;
        queueArray[currentPriority][currentPosition].pageTick = 0;
        queueArray[currentPriority][currentPosition].frameTick = 0;
        queueArray[currentPriority][currentPosition].isRunning = false;
        queueArray[currentPriority][currentPosition].executionTime = 0;

        userArray[currentUser].hasProcess = true;
    }

    void placeInShadow()
    {
        int currentPosition;
        if (currentPriority == 0)
        {
            currentPosition = queueOneShadowPosition;
            queueOneShadowPosition++;
        }
        if (currentPriority == 1)
        {
            currentPosition = queueTwoShadowPosition;
            queueTwoShadowPosition++;
        }

        queueShadow[currentPriority][currentPosition].pid = queueArray[currentPriority][0].pid;
        queueShadow[currentPriority][currentPosition].pCounter = queueArray[currentPriority][0].
        pCounter;
        queueShadow[currentPriority][currentPosition].isComplete = queueArray[currentPriority][0].
        isComplete;
        queueShadow[currentPriority][currentPosition].isRunning = queueArray[currentPriority][0].
        isRunning;
        queueShadow[currentPriority][currentPosition].processType = queueArray[currentPriority][0
        ].processType;
        queueShadow[currentPriority][currentPosition].pageTick = queueArray[currentPriority][0].
        pageTick;
        queueShadow[currentPriority][currentPosition].frameTick = queueArray[currentPriority][0].
        frameTick;
        queueShadow[currentPriority][currentPosition].executionTime = queueArray[currentPriority][
        0].executionTime;
        queueShadow[currentPriority][currentPosition].processCC = queueArray[currentPriority][0].
        processCC;
        int p = 0;
        for (p; p < MAX_REGISTER; p++)
           queueShadow[currentPriority][currentPosition].processRegisters[p] = queueArray[
           currentPriority][0].processRegisters[p];

        printf("\nUser %d process moved to shadow queue\n", queueArray[currentPriority][0].pid);
     }

     bool queueHasProcess(int priority)
     {
        if (queueArray[priority][0].pid > -1)
           return true;
        else
           return false;
     }

     void dumpQueues()
     {
        int i = 0;
        printf("\tPrimary\t\tShadow\n");
        printf("One:\t");
        for (i = 0; i < numberOfProcesses; ++i)
        {
           if (queueArray[0][i].pid > -1)
              printf("%d   ", queueArray[0][i].pid);
           else
              printf("    ");
        }
        printf("\t");
        for (i = 0; i < numberOfProcesses; ++i)
        {
           if (queueShadow[0][i].pid > -1)
              printf("%d   ", queueShadow[0][i].pid);
           else
              printf("    ");
        }
        printf("\n");
        printf("Two:\t");
        for (i = 0; i < numberOfProcesses; ++i)
        {
           if (queueArray[1][i].pid > -1)
              printf("%d   ", queueArray[1][i].pid);
           else
              printf("    ");
        }
        printf("\t");
        for (i = 0; i < numberOfProcesses; ++i)
        {
           if (queueShadow[1][i].pid > -1)
              printf("%d   ", queueShadow[1][i].pid);
           else
              printf("    ");
        }
        printf("\n");
     }

     void promoteProcesses()
     {
        int i = 0;
        int p = 0;

        for (i; i < numberOfProcesses; i++)
        {
           if (queueArray[1][i].executionTime == 0 && queueArray[1][i].pid > -1)
           {
              printf("\nUser %d process elevated to priority 1\n", queueArray[1][i].pid);
              queueArray[0][queueOnePosition].pid = queueArray[1][i].pid;
              queueArray[0][queueOnePosition].pCounter = queueArray[1][i].pCounter;
              queueArray[0][queueOnePosition].isComplete = queueArray[1][i].isComplete;
              queueArray[0][queueOnePosition].isRunning = queueArray[1][i].isRunning;
              queueArray[0][queueOnePosition].processType = queueArray[1][i].processType;
              queueArray[0][queueOnePosition].pageTick = queueArray[1][i].pageTick;
              queueArray[0][queueOnePosition].frameTick = queueArray[1][i].frameTick;
              queueArray[0][queueOnePosition].executionTime = queueArray[1][i].executionTime;
              queueArray[0][queueOnePosition].processCC = queueArray[1][i].processCC;
              for (p; p < MAX_REGISTER; p++)
                 queueArray[0][queueOnePosition].processRegisters[p] = queueArray[1][i].
                 processRegisters[p];
              queueOnePosition++;
              queueArray[1][i].pid = -1;
              dumpQueues();
           }
        }
     }

     void updateQueuePositions()
     {
        int p;
        queueOnePosition = 0;
        queueTwoPosition = 0;
        queueOneShadowPosition = 0;
        queueTwoShadowPosition = 0;

        for (p = 0; p < numberOfProcesses; p++)
        {
           if (queueArray[0][p].pid > -1) queueOnePosition = p + 1;
           if (queueArray[1][p].pid > -1) queueTwoPosition = p + 1;
           if (queueShadow[0][p].pid > -1) queueOneShadowPosition = p + 1;
           if (queueShadow[1][p].pid > -1) queueTwoShadowPosition = p + 1;
        }
     }
```