```c
/*     Authors: Austin Blythe, Matthew Francis
 *       Acct#: cs441102
 *       Class: CSC 441, Dr. Stader
 *  Start Date: 20 Feb 2013
 * Finish Date: 20 Mar 2013
 * Description: This program builds on OS_Part 2, this iteration deals with memory management,
 *                  implementing page tables for users and having memory split into frames.
 */

#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
   const int MEMORY_LENGTH = 4;  // Size of the pages and frame

   #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 10  // 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
   bool diskLocked;              // Disk lock
   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 frameTick;                // Count instructions within a frame
   int pageTick;                 // Current page to reference for memory location
   int currentUser;              // Current user in the RR
   int validCommand;

   // 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 / 4];  // Page table for the user owned process(s), allows for largest possible load.
      int currentPage;
   };
   struct user userArray[numberOfUsers];  // Array for creation of users + 1 to allow extra spot

   struct processBlock {
      int pid;                   // Stores who owns the process
      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
      unsigned short lAddress;
   };

   struct semaphore {
      int count;                 // Used for determining where in the queue a new process is placed
   } semaphore;

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

   unsigned short disk[DISK_SIZE];          // 1D-array of short int
   unsigned short mainMemory[MAX_MEMORY];  // Main memory
   int memoryFrames[MAX_MEMORY / 4];        // Frame number to memory location mapping
   int usedFrames[MAX_MEMORY / 4];          // 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

// Method Declarations
   int main(void);

   // Operating System
   void initializeOS();
   void loader();
   void scheduler();
   void dispatcher();
   void userInterface();
   void interpreter();
   bool userHasProcess();                // Checks if the user has a process in the queue
   void cleanUp();                       // Shifts the process queue as necessary

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

   // Memory
   void mmu(int, int);

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

// Functions
   unsigned short convertNumber(char*);
   void printBin(unsigned short);
   void printHex(unsigned short);
   void changeCondition(int);

// 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();

```

```c
      // Round robin scheduler (user1, user2, o/s)
      scheduler();

      return 0;
   }


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

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

   // 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] = 0x0000;
         if (p % MEMORY_LENGTH == 0) {
            memoryFrames[p / MEMORY_LENGTH] = mainMemory[p];
            usedFrames[p] = 4;
         }
      }

      int i, j;
   // Initialize page table
      for (j = 1; j < numberOfUsers; j++) {
         for(i = 0; i < MAX_MEMORY / 4; i++) {
            userArray[j].pageTable[i] = -1;
         }
      }

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



   // 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;

      // User1
   userArray[1].memoryLocation = 0;
   userArray[1].progLength = 6;
   userArray[1].hasProcess = false;

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

   semaphore.count = 0;
}

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

// Place user program pages into main memory frames
   for (p = userArray[currentUser].memoryLocation; p < (userArray[currentUser].progLength + userArray[currentUser].memoryLocation);) {
      currentFrame = rand() % 64;
      memoryLoc = currentFrame * 4;
      if (currentPage == 0){
         processArray[semaphore.count].pCounter = memoryLoc;
      }
      if (usedFrames[currentFrame] == 0) {
         for (i = 0; i < 4; i++, p++) {
            mainMemory[memoryLoc + i] = disk[p];
         }
         mmu(currentPage, currentFrame);
```

```c
         currentPage++;
         usedFrames[currentFrame] = currentUser;
      }
   }
}

void scheduler() {
   while (stopOS == false) {
      currentTick = 0;

      dispatcher();

      if (currentUser == numberOfUsers-1) {
         currentUser = 0;
      }
      else {
         currentUser++;
      }
   }
}

// Directs users/OS based on command entered and semaphore status
void dispatcher() {
   validCommand = false;
   if (processArray[0].pid == currentUser && currentUser > 0) {
      PC = processArray[0].pCounter;
      interpreter();
   }

   if(currentTick < TICKS_PER_USER) {
      while (!validCommand) {
         userInterface();
         if(controlCommand[0] == 'r' && controlCommand[1] == 'u' && controlCommand[2] == 'n'){
            run();
         }

         else if(controlCommand[0] == 'd' && controlCommand[1] == 'm' && controlCommand[2] == 'p'){
            if( currentUser == 0) {
               dmp();
               validCommand = true;
            }
            else {
               printf("You are not authorized to issue that command.\n");
            }
            programClock++;
            currentTick++;
         }

         else if(controlCommand[0] == 'n' && controlCommand[1] == 'o' && controlCommand[2] == 'p'){
            printf("\tNo operation performed.\n");
            validCommand = true;
            programClock++;
            currentTick++;
         }

         else if(controlCommand[0] == 's' && controlCommand[1] == 't' && controlCommand[2] == 'p'){
            stp();
            programClock++;
            currentTick++;
         }

         else {
            printf("\tInvalid command entered\n");
         }
      }
   }
}

// Interactive command-line user interface
void userInterface() {
   if (currentUser == 0) printf("\n\tO/S");
   else printf("\n\tUser %i", currentUser);
   printf("\nPlease enter a command: ");

   fgets(controlCommand, 5, stdin);
   printf("\n");
}

// responsible for the machine languare interpretation and execution
void interpreter() {
   if (currentUser > 0) printf("\nUser %d Running...\n", currentUser);
   processArray[0].isRunning = true;
   while(haltFlag == false && currentTick < TICKS_PER_USER) {
      Fetch();
      Decode();
      mmu(-1, -1);
      Execute();
      if (frameTick % 4 == 0) {
         pageTick++;
         processArray[0].pCounter = userArray[currentUser].pageTable[pageTick] * 4;
         PC = processArray[0].pCounter;
      }
   }

   if (haltFlag == true) {
      processArray[0].isComplete = true;
      dumpPageTable();
      mmu(-2,-2);
   }
   else {
      processArray[0].pCounter = PC;
   }
```

```c
      cleanUp();
      haltFlag = false; // reset halt flag for subsequent program runs
   }

   // Clean up
   void cleanUp() {
      int i;
      if (processArray[0].isComplete == true) {
         processArray[0].isRunning = false;
         semaphore.count--;
         userArray[currentUser].hasProcess = false;
         pageTick = 0;
         frameTick = 0;

         for (i = 0; i < numberOfProcesses - 1; i++) {
            processArray[i].pid = processArray[i+1].pid;
            processArray[i].pCounter = processArray[i+1].pCounter;
            processArray[i].isComplete = processArray[i+1].isComplete;
            processArray[i].isRunning = processArray[i+1].isRunning;
         }

         processArray[numberOfProcesses - 1].pid = 0;
         processArray[numberOfProcesses - 1].pCounter = 0;
         processArray[numberOfProcesses - 1].isComplete = false;
         processArray[numberOfProcesses - 1].isRunning = false;
      }
   }

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

   void run(){
      if(currentUser > 0 && userArray[currentUser].hasProcess == false) {
         loader();
         processArray[semaphore.count].pid = currentUser;
         semaphore.count++;
         userArray[currentUser].hasProcess = true;
         programClock++;
         currentTick++;
         if (processArray[0].pid == currentUser) {
            PC = processArray[0].pCounter;
            processArray[0].isRunning = true;
            interpreter();
         }
         else {
            printf("A processes is already running. Your process has been added to the queue.\n");
         }
         validCommand = true;
      }
      else if (currentUser > 0 && userArray[currentUser].hasProcess == true) {
         printf("Your process is already queued. Please wait.\n");
      }
      else {
```

```
      printf("You are not authorized to issue that command.\n");
   }
}

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

   printf("PC   ");
   printHex(PC);
   printBin(PC);
   printf("\n");

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

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

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

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

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

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

   for (i = 0; i < DISK_SIZE; i++) {
      if(disk[i] != 0x0000){
```

```c
         printf("%-3d  ", i);
         printHex(disk[i]);
         printBin(disk[i]);
         printf("\n");
      }
   }

   printf("\nPROCESS QUEUE\n---------------------------------\n");

   if (processArray[0].pid > 0) {
      printf("Process Owner\tMem Location\t Is Running\t\n");
      for (i = 0; i < numberOfProcesses; i++) {
         if (processArray[i].pid != 0) {
            printf("\t%d\t\t%d\t\t", processArray[i].pid, processArray[i].pCounter);
            if (processArray[i].isRunning == true) {
               printf("*\n");
            }
            else {
               printf("-\n");
            }
         }
      }
   }
   else {
      printf("Process queue empty\n");
   }

   printf("\n---------------------------------\nDUMP COMPLETE\n---------------------------------\n");
}

// Called from halt instruction
void dumpPageTable(){
   printf("\nUser %d Page Table\n", currentUser);
   int h,k;
   printf("Page \t| \tFrame\n");
   for (h = 0; h < (MAX_MEMORY / 4); h++){
      if(userArray[currentUser].pageTable[h] > -1){
         printf("%d \t| \t%d\n",h,userArray[currentUser].pageTable[h]);
         printf("\t\t\tFrame %d Contents\n", userArray[currentUser].pageTable[h]);
         for(k = 0; k < 4; k++){
            printf("\t\t\t\t%d: ",userArray[currentUser].pageTable[h]*4 + k);
            printHex(mainMemory[userArray[currentUser].pageTable[h]*4 + k]);
            printBin(mainMemory[userArray[currentUser].pageTable[h]*4 + k]);
            printf("\n");
         }
         usedFrames[h] = 0;
      }
   }
}

void stp(){

      if(currentUser == 0){
         stopOS = true;
         dmp();
         printf("\n\tMachine halted.\n\n");
      }
      else {
         printf("You are not authorized to issue that command.\n");
      }
      validCommand = true;
   }


// ******************** 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[currentUser].pageTable[pageNum] * 4) + offset;
      }
      else if (page > -1 && frame > -1) {
         userArray[currentUser].pageTable[page] = frame;
      }

      int i;
      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/4; i++){
            userArray[currentUser].pageTable[i] = -1;
         }
      }
   }


// ******************** 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;
         default:
            break;
      }
      programClock++;
      currentTick++;
      frameTick++;
   }


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

   // 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 {}
}

```