```
/*     Authors: Austin Blythe, Matthew Francis
 *       Acct#: cs441104
 *       Class: CSC 441, Dr. Stader
 *  Start Date: 8 Apr 2013
 * Finish Date: 24 Apr 2013
 * Description: This program adds a more in depth I/O system, using an input file
 *              a FAT (File Allocation Table) is created.
 */

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

// Global Constants
   #define MEM_SIZE 256           // Size of memory array
   #define DISK_SIZE 256          // Size of the disk array
   #define PAGE_SIZE 4            // page/frame length
   #define NUM_REGISTERS 4        // Number of registers in the machine
   #define NUM_USERS 3            // Number of users in the system
   #define NUM_PROCESSES 10       // Number of simultaneous processes (possible) in the system
   #define true 1                 // Setting keyword true to 1
   #define false 0                // Setting keyword false to 0

   const int TICKS_PER_USER = 4; // # of Ticks allowed per cycle for user

   // 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
   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
   typedef struct
   {
      int memoryLocation;                    // Starting location for that users program
      int progLength;                        // Number of instructions in the user's program
      int pageTable[MEM_SIZE / PAGE_SIZE];   // Page table for the user owned process(s),
      allows for largest possible load.
   } user;
   user *userArray[NUM_USERS];  // Array for creation of users + 1 to allow extra spot

   typedef struct
   {
      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[NUM_REGISTERS];
      int pageTick;
      int frameTick;
   } processBlock;

   processBlock *queueArray[2][NUM_PROCESSES];
   int queueOneRear;
   int queueTwoRear;

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

   int diskPrint[DISK_SIZE];     // locations on disk to print in dmp
   int memoryPrint[MEM_SIZE];    // locations in memory to print in dmp

   char uiCommand[100];
   //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), nop(4)

// Method Declarations
   int main();

   void initializeOS();
   void userInterface();

   void scheduler();
   void dispatcher();

   void run();
   void dmp();
   void stp();
   void nop();

   void loader();
   void interpreter();
   void cleanUp();

   void mmu(int, int);
   void Fetch();
   void Decode();
   void Execute();

   void dumpPageTable();
   unsigned short convertNumber(char *);
   void printBin(unsigned short);
   void printHex(unsigned short);
   void changeCondition(int);
   bool isValidCommand();
   void placeInQueue();
   bool queueHasProcess();
   void dumpQueues();
   void promoteProcess();
   void demoteProcess();
   void removeProcess();
   void rotateProcess();

   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 file:
#include "instructions.h" // Instruction methods

// ******************** MAIN ********************
   int main()
   {
    // OS Initialization
      initializeOS();

      printf("To run a program, type 'run programName'\n");
      printf("Valid program names: 'user1' or 'user2'\n");

    // User Interface loop
      while (true)
      {
         userInterface();
      }
      return 0;
   }

// ******************** OPERATING SYSTEM ********************
   void initializeOS()
   {
    // Initialize Values
      programClock = 0;
      CC = 0;
      PC = 0;
      IR = 0;
      haltFlag = false;
      currentUser = 1;  // Starts with user 1
      currentTick = 0;  // User 1 starts with 0 ticks on their cycle
      commandCode = 0;
      queueOneRear = 0;
      queueTwoRear = 0;

    // Operating System
      userArray[0] = malloc(sizeof(user));

    // User1
      userArray[1] = malloc(sizeof(user));
      userArray[1]->memoryLocation = 0;
      userArray[1]->progLength = 6;

    // User2
      userArray[2] = malloc(sizeof(user));
      userArray[2]->memoryLocation = 100;
      userArray[2]->progLength = 6;

    // Zero out the mainMemory and map frame locations
    // Initialize the usedFrames array so they are all available
      int i = 0;
      int j = 0;
      for (i; i < MEM_SIZE; i++)
      {
         mainMemory[i] = 0;
         memoryPrint[i] = 0;
         if (i % PAGE_SIZE == 0)
         {
            usedFrames[i / PAGE_SIZE] = 0;
         }
      }

    // Initialize page table
      for (j = 1; j < NUM_USERS; j++)
      {
         for (i = 0; i < MEM_SIZE / PAGE_SIZE; i++)
         {
            userArray[j]->pageTable[i] = -1;
         }
      }

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

    // Set pid for processes to -1
      for (i = 0; i < NUM_PROCESSES; i++)
      {
         queueArray[0][i] = NULL;
         queueArray[1][i] = NULL;
      }

      FILE *fp = fopen("userPrograms.txt", "r");

      char filename[10];
      int loc, length;

      printf("\n\tCreating FAT\n");

      fscanf(fp, "%s", filename);
      fscanf(fp, "%d", &loc);
      fscanf(fp, "%d", &length);

   //    printf("%s\n%d\n%d\n", filename, loc, length);

      for (i = loc; i < loc + length; i++)
      {
         fscanf(fp, "%hX", &disk[i]);
      //    printf("%04X\n", disk[i]);
      }

      fscanf(fp, "%s", filename);
      fscanf(fp, "%d", &loc);
      fscanf(fp, "%d", &length);

   //    printf("%s\n%d\n%d\n", filename, loc, length);

      for (i = loc; i < loc + length; i++)
      {
         fscanf(fp, "%hX", &disk[i]);
      //    printf("%04X\n", disk[i]);
      }
   }

   // Interactive command-line user interface
   void userInterface()
   {
      printf("\n\n");
    // User prompt
```

```
   if (currentUser == 0) printf("Operating System: ");
   else printf("User %i: ", currentUser);

   char filename[10];

 // Get command
   scanf("%s", uiCommand);
   if ( strcmp(&uiCommand, "run") == 0 )
      scanf("%s", filename);

   printf("\n");

   if (isValidCommand())
   {
      switch (commandCode)
      {
         case 1: placeInQueue();
            break;
         case 2: dmp();
            break;
         case 3: placeInQueue();
            break;
         case 4: nop();
            break;
      }

     // Schedule next execution request
      scheduler();

     // Routine cleanup after every execution; queue manipulation, process removal
      if (queueArray[currentPriority][0] != NULL)
         cleanUp();

      dumpQueues();

     // Cycle to the next user
      if (currentUser == NUM_USERS - 1) currentUser = 0;
      else currentUser++;
   }
   else printf("Invalid command entered\n");
}

// Process a request based on priority
void scheduler()
{
   currentTick = 0;

   if (queueHasProcess(0))
   {
      currentPriority = 0;
      dispatcher();
   }
   else if (queueHasProcess(1))
   {
      currentPriority = 1;
      dispatcher();
   }
   else
      printf("Both queues are empty; no operations performed\n");
}

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

// Run the current user program request
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 < NUM_REGISTERS; i++)
         Registers[i] = queueArray[currentPriority][0]->processRegisters[i];
      programClock++;
      interpreter();
   }
   else
   {
     // First run
      loader();
      programClock++;
      currentTick++;
      PC = queueArray[currentPriority][0]->pCounter;
      queueArray[currentPriority][0]->isRunning = true;
      interpreter();
   }
}

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 * PAGE_SIZE;
      if (currentPage == 0)
      {
         queueArray[currentPriority][0]->pCounter = memoryLoc;

      }
      if (usedFrames[currentFrame] == 0)
      {
         for (i = 0; i < PAGE_SIZE; i++, p++)
         {
            mainMemory[memoryLoc + i] = disk[p];
            memoryPrint[memoryLoc + i] = queueArray[currentPriority][0]->pid;
            diskPrint[p] = 1;
         }
         usedFrames[currentFrame] = 1;
         mmu(currentPage, currentFrame);
         currentPage++;
      }
   }

}

// responsible for the machine language interpretation and execution
void interpreter()
{
   printf("\n");
   if (queueArray[currentPriority][0]->pid > 0) printf("User %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 % PAGE_SIZE == 0)
      {
         queueArray[currentPriority][0]->pageTick++;
         queueArray[currentPriority][0]->pCounter = userArray[queueArray[currentPriority][0
         ]->pid]->pageTable[queueArray[currentPriority][0]->pageTick] * PAGE_SIZE;
         PC = queueArray[currentPriority][0]->pCounter;
      }
   }

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

// This will create a dump of the data in the program
void dmp()
{
   printf("---------------------------------\n\tDUMP
   START\n---------------------------------\n\n");
   programClock++;
   currentTick++;

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

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

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

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

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

   printf("IR\t");
   printHex(IR);

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

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

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

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

   dumpQueues();

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

void stp()
{
   programClock++;
   queueArray[currentPriority][0]->isComplete = true;
   cleanUp();
   dmp();
   printf("\n\n---------------------------------\n\tMACHINE
   HALTED\n---------------------------------\n");
   exit(0);
}

void nop()
{
   programClock++;
   printf("No request added\n");
}

// Clean up
void cleanUp()
{
   printf("\n");
   int i = 0;
```

```
      if (queueArray[currentPriority][0]->isComplete == true)
         removeProcess();   // remove completed processes and shift queue
      else
      {
         if (currentPriority == 0) demoteProcess();
         else rotateProcess();
      }

      promoteProcess();  // check for and promote priority 2 process
   }

// ******************** 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] *
         PAGE_SIZE) + 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 < MEM_SIZE / PAGE_SIZE; i++)
         {
            for (j = 0; j < MEM_SIZE / PAGE_SIZE; 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;
   }
```

```c
   }

   // 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;
      }
      printf("\n");
      programClock++;
      currentTick++;
      queueArray[currentPriority][0]->frameTick++;
      queueArray[currentPriority][0]->executionTime++;
   }

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

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

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

```c
}

// 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;
}

bool isValidCommand()
{
   if (strcmp(&uiCommand, "run") == 0 && currentUser > 0)
   {
      commandCode = 1;
      return true;
   }
   else if (strcmp(&uiCommand, "dmp") == 0 && currentUser == 0)
   {
      commandCode = 2;
      return true;
   }
   else if (strcmp(&uiCommand, "stp") == 0 && currentUser == 0)
   {
      commandCode = 3;
      return true;
   }
   else if (strcmp(&uiCommand, "nop") == 0)
   {
      commandCode = 4;
      return true;
   }
   else
      return false;
}

void placeInQueue()
{
   int currentPosition; // Local variable for current position in the queue's
   if (commandCode == 3)
   {
      currentPosition = queueOneRear;
      queueOneRear++;
      currentPriority = 0;
      printf("Priority %d request added\n", currentPriority + 1);
   }
   else if (commandCode == 1)
   {
      currentPosition = queueTwoRear;
      queueTwoRear++;
      currentPriority = 1;
      printf("Priority %d request added\n", currentPriority + 1);
   }
   queueArray[currentPriority][currentPosition] = malloc(sizeof(processBlock));

   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;
}

bool queueHasProcess(int priority)
{
   if (queueArray[priority][0] != NULL)
      return true;
   else
      return false;
}

void dumpQueues()
{
   int i;
   printf("\nPriority Queues\n---------------\nOne:   ");
   for (i = 0; i < NUM_PROCESSES; ++i)
   {
      if (queueArray[0][i] != NULL)
         printf("%d   ", queueArray[0][i]->pid);
      else
         printf("");
   }

   printf("\nTwo:   ");
   for (i = 0; i < NUM_PROCESSES; ++i)
   {
      if (queueArray[1][i] != NULL)
         printf("%d   ", i /*queueArray[1][i]->pid*/);
      else
         printf("");
   }
   printf("\n");
}

void promoteProcess()
{
   int i;

   if (queueArray[1][0] != NULL && queueArray[1][0]->executionTime == 0)
   {
      printf("User %d process promoted to priority 1\n", queueArray[1][0]->pid);

      queueArray[0][queueOneRear] = queueArray[1][0];
      queueOneRear++;
```

```
         for (i = 0; i < queueTwoRear; i++)
            queueArray[1][i] = queueArray[1][i + 1];

         queueTwoRear--;
      }
   }

   void demoteProcess()
   {
      int i;

      if (queueArray[0][0] != NULL && queueArray[0][0]->executionTime > 0)
      {
         printf("User %d process demoted to priority 2\n", queueArray[0][0]->pid);

         queueArray[1][queueTwoRear] = queueArray[0][0];
         queueTwoRear++;

         for (i = 0; i < queueOneRear; i++)
            queueArray[0][i] = queueArray[0][i + 1];

         queueOneRear--;
      }
   }

   void removeProcess()
   {
      printf("User %d process removed from queue %d\n", queueArray[currentPriority][0]->pid,
      currentPriority + 1);

      int i;
      int rear;

      if (currentPriority == 0) rear = queueOneRear;
      if (currentPriority == 1) rear = queueTwoRear;

      for (i = 0; i < rear; i++)
         queueArray[currentPriority][i] = queueArray[currentPriority][i + 1];

      if (currentPriority == 0) queueOneRear--;
      if (currentPriority == 1) queueTwoRear--;
   }

   void rotateProcess()
   {
      printf("User %d process moved to rear of queue %d\n", queueArray[currentPriority][0]->pid,
       currentPriority + 1);

      int i;
      int rear;

      if (currentPriority == 0) rear = queueOneRear;
      if (currentPriority == 1) rear = queueTwoRear;
```

```
        queueArray[currentPriority][rear] = queueArray[currentPriority][0];

        for (i = 0; i < rear + 1; i++)
            queueArray[currentPriority][i] = queueArray[currentPriority][i + 1];
    }
```