```
1   /*     Authors: Austin Blythe, Matthew Francis
2    *       Acct#: cs441102
3    *       Class: CSC 441, Dr. Stader
4    *  Start Date: 20 Feb 2013
5    * Finish Date: 20 Mar 2013
6    * Description: This program builds on OS_Part 2, this iteration deals with memory
     management,
7    *                   implementing page tables for users and having memory split into
     frames.
8    */
9
10  #include <stdlib.h>
11  #include <stdio.h>
12
13  // Global Constants
14     const int OPCODE_LEN = 4;     // Opcode field length
15     const int MODE_LEN = 1;       // Mode field length
16     const int REG_LEN = 3;        // Register field length
17     const int ADDRE_LEN = 8;      // Address field length
18     const int MAX_BITS = 16;      // Size of a single memory location in bits
19     const int TICKS_PER_USER = 4; // # of Ticks allowed per cycle for user
20     const int MEMORY_LENGTH = 4;  // Size of the pages and frame
21
22     #define MAX_MEMORY 256        // Size of memory array
23     #define MAX_REGISTER 4        // Number of registers in the machine
24     #define DISK_SIZE 256         // Size of the disk
25     #define true 1                   // Setting keyword true to 1
26     #define false 0               // Setting keyword false to 0
27     #define numberOfUsers 3          // Number of users in the system
28     #define numberOfProcesses 10  // Number of simultaneous processes (possible) in the
       system
29
30  // Global Varibles
31     typedef int bool;             // Create bool type as C does not have one
32
33     unsigned short CC;            // Condition code
34     unsigned short PC;            // Program Counter
35     unsigned short IR;            // Instruction Register
36
37     unsigned short opcode;        // Opcode field
38     unsigned short mode;          // Mode field
39     unsigned short reg;           // Register field
40     signed short address;         // Address field
41
42
43
44     // Internal
45     bool haltFlag;                // Controls system halt
46     bool diskLocked;              // Disk lock
47     int programClock;             // Internal programClock
48     bool stopOS;                  // Changes to true if stp is issues
49     int currentTick;              // Current tick that the user is on (resets to 0 when
       limit reached)
```

```
50      int frameTick;                  // Count instructions within a frame
51      int pageTick;                   // Current page to reference for memory location
52      int currentUser;                // Current user in the RR
53      int validCommand;
54
55      // Structure format creation for users and O/S
56      struct user{
57         int memoryLocation;          // Starting location for that users program
58         bool hasProcess;             // If true, user has a process in the queue / loaded
           into memory
59         int progLength;              // Number of instructions in the user's program
60         int pageTable[MAX_MEMORY / 4];  // Page table for the user owned process(s),
           allows for largest possible load.
61         int currentPage;
62      };
63      struct user userArray[numberOfUsers];  // Array for creation of users + 1 to allow
        extra spot
64
65      struct processBlock {
66         int pid;                     // Stores who owns the process
67         unsigned short pCounter;     // Stores what memory location the program is
           currently at
68         bool isRunning;              // Used to determine locked/queue status
69         bool isComplete;             // Used for cleanUp method to shift queue
70         unsigned short lAddress;
71      };
72
73      struct semaphore {
74         int count;                   // Used for determining where in the queue a new
           process is placed
75      } semaphore;
76
77      struct processBlock processArray[numberOfProcesses]; // Process queue, only one can
        be processes at once currently f
78
79      unsigned short disk[DISK_SIZE];          // 1D-array of short int
80      unsigned short mainMemory[MAX_MEMORY];  // Main memory
81      int memoryFrames[MAX_MEMORY / 4];       // Frame number to memory location mapping
82      int usedFrames[MAX_MEMORY / 4];         // Frame usage bit vector
83      signed short Registers[MAX_REGISTER];  // Registers array (0 is Accumulator)
84
85
86      char userIn[5] = {0};                  // Array for user input
87      char* controlCommand = userIn;         // Pointer to userIn array
88
89    // Method Declarations
90      int main(void);
91
92      // Operating System
93      void initializeOS();
94      void loader();
95      void scheduler();
96      void dispatcher();
```

```
97      void userInterface();
98      void interpreter();
99      bool userHasProcess();                  // Checks if the user has a process in the queue
100     void cleanUp();                         // Shifts the process queue as necessary
101
102     // UI Commands
103     void run();
104     void dmp();
105     void stp();
106     void dumpPageTable();
107
108     // Memory
109     void mmu(int, int);
110
111     // Interpreter
112     void Fetch();
113     void Decode();
114     void Execute();
115
116  // Functions
117     unsigned short convertNumber(char*);
118     void printBin(unsigned short);
119     void printHex(unsigned short);
120     void changeCondition(int);
121
122  // Instruction Set
123     void load();
124     void store();
125     void add();
126     void sub();
127     void adr();
128     void sur();
129     void and();
130     void or();
131     void not();
132     void jmp();
133     void jeq();
134     void jgt();
135     void jlt();
136     void compare();
137     void clear();
138     void halt();
139
140  // User-defined header files:
141  #include "instructions.h"  // Needs to be below variable declarations
142
143  // ******************** MAIN ********************
144
145     int main (void) {
146
147        // OS Initialization
148        initializeOS();
149
```

```
150         // Round robin scheduler (user1, user2, o/s)
151         scheduler();
152 
153         return 0;
154      }
155 
156 
157   // ******************* OPERATING SYSTEM ********************
158 
159      void initializeOS(){
160      // Initialize Values
161         programClock = 0;
162         CC = 0;
163         PC = 0;
164         IR = 0;
165         haltFlag = false;
166         diskLocked = false;
167         stopOS = false;
168         currentUser = 1;  // Starts with user 1
169         currentTick = 0;  // User 1 starts with 0 ticks on their cycle
170         frameTick = 0;
171         pageTick = 0;
172         validCommand = false;
173 
174      // Zero out the mainMemory and map frame locations
175      // Initialize the usedFrames array so they are all available
176 
177         int p = 0;
178         for(p; p < MAX_MEMORY; p++){
179            mainMemory[p] = 0x0000;
180            if (p % MEMORY_LENGTH == 0) {
181               memoryFrames[p / MEMORY_LENGTH] = mainMemory[p];
182               usedFrames[p] = 4;
183            }
184         }
185 
186         int i, j;
187      // Initialize page table
188         for (j = 1; j < numberOfUsers; j++) {
189            for(i = 0; i < MAX_MEMORY / 4; i++) {
190               userArray[j].pageTable[i] = -1;
191            }
192         }
193 
194      // Zero out the disk
195         p = 0;
196         for(p; p < DISK_SIZE; p++){
197            disk[p] = 0x0000;
198         }
199 
200 
201 
202      // User programs on the disk
```

```
203            // User 1 data set
204         disk[0]   = 0x080A; // Location 000 // Load Immediate R0 #10
205         disk[1]   = 0x1006; // Location 001 // Store R0 6
206         disk[2]   = 0x0905; // Location 002 // Load Immediate R1 #5
207         disk[3]   = 0x4100; // Location 003 // AddR R1
208         disk[4]   = 0x1007; // Location 004 // Store R0 7
209         disk[5]   = 0xF000; // Location 005 // Halt
210
211            // User 2 data set
212         disk[100] = 0x0819; // Location 100 // LOAD I R0 #25
213         disk[101] = 0x1006; // Location 101 // STO R0 6
214         disk[102] = 0x0905; // Location 102 // LOD I R1 #5
215         disk[103] = 0x5100; // Location 103 // SUR R1
216         disk[104] = 0x1007; // Location 104 // STO R0 7
217         disk[105] = 0xF000; // Location 105 // HALT
218
219
220      // Create user(s)
221            // OS
222         userArray[0].memoryLocation = 0;
223
224            // User1
225         userArray[1].memoryLocation = 0;
226         userArray[1].progLength = 6;
227         userArray[1].hasProcess = false;
228
229            // User2
230         userArray[2].memoryLocation = 100;
231         userArray[2].progLength = 6;
232         userArray[2].hasProcess = false;
233
234         semaphore.count = 0;
235      }
236
237      void loader() {
238         int p, i;
239         int currentFrame;
240         int currentPage = 0;
241         int memoryLoc;
242
243      // Place user program pages into main memory frames
244         for (p = userArray[currentUser].memoryLocation; p < (userArray[currentUser].
         progLength + userArray[currentUser].memoryLocation);) {
245            currentFrame = rand() % 64;
246            memoryLoc = currentFrame * 4;
247            if (currentPage == 0){
248               processArray[semaphore.count].pCounter = memoryLoc;
249            }
250            if (usedFrames[currentFrame] == 0) {
251               for (i = 0; i < 4; i++, p++) {
252                  mainMemory[memoryLoc + i] = disk[p];
253               }
254               mmu(currentPage, currentFrame);
```

```
255                currentPage++;
256                usedFrames[currentFrame] = currentUser;
257             }
258          }
259       }
260
261       void scheduler() {
262          while (stopOS == false) {
263             currentTick = 0;
264
265             dispatcher();
266
267             if (currentUser == numberOfUsers-1) {
268                currentUser = 0;
269             }
270             else {
271                currentUser++;
272             }
273          }
274       }
275
276       // Directs users/OS based on command entered and semaphore status
277       void dispatcher() {
278          validCommand = false;
279          if (processArray[0].pid == currentUser && currentUser > 0) {
280             PC = processArray[0].pCounter;
281             interpreter();
282          }
283
284          if(currentTick < TICKS_PER_USER) {
285             while (!validCommand) {
286                userInterface();
287                if(controlCommand[0] == 'r' && controlCommand[1] == 'u' && controlCommand[2]
                    == 'n'){
288                   run();
289                }
290
291                else if(controlCommand[0] == 'd' && controlCommand[1] == 'm' &&
                   controlCommand[2] == 'p'){
292                   if( currentUser == 0) {
293                      dmp();
294                      validCommand = true;
295                   }
296                   else {
297                      printf("You are not authorized to issue that command.\n");
298                   }
299                   programClock++;
300                   currentTick++;
301                }
302
303                else if(controlCommand[0] == 'n' && controlCommand[1] == 'o' &&
                   controlCommand[2] == 'p'){
304                   printf("\tNo operation performed.\n");
```

```
305                validCommand = true;
306                programClock++;
307                currentTick++;
308             }
309
310             else if(controlCommand[0] == 's' && controlCommand[1] == 't' &&
                controlCommand[2] == 'p'){
311                stp();
312                programClock++;
313                currentTick++;
314             }
315
316             else {
317                printf("\tInvalid command entered\n");
318             }
319          }
320       }
321    }
322
323    // Interactive command-line user interface
324    void userInterface() {
325       if (currentUser == 0) printf("\n\tO/S");
326       else printf("\n\tUser %i", currentUser);
327       printf("\nPlease enter a command: ");
328
329       fgets(controlCommand, 5, stdin);
330       printf("\n");
331    }
332
333    // responsible for the machine languare interpretation and execution
334    void interpreter() {
335       if (currentUser > 0) printf("\nUser %d Running...\n", currentUser);
336       processArray[0].isRunning = true;
337       while(haltFlag == false && currentTick < TICKS_PER_USER) {
338          Fetch();
339          Decode();
340          mmu(-1, -1);
341          Execute();
342          if (frameTick % 4 == 0) {
343             pageTick++;
344             processArray[0].pCounter = userArray[currentUser].pageTable[pageTick] * 4;
345             PC = processArray[0].pCounter;
346          }
347       }
348
349       if (haltFlag == true) {
350          processArray[0].isComplete = true;
351          dumpPageTable();
352          mmu(-2,-2);
353       }
354       else {
355          processArray[0].pCounter = PC;
356       }
```

```
357         cleanUp();
358         haltFlag = false; // reset halt flag for subsequent program runs
359      }
360
361      // Clean up
362      void cleanUp() {
363         int i;
364         if (processArray[0].isComplete == true) {
365            processArray[0].isRunning = false;
366            semaphore.count--;
367            userArray[currentUser].hasProcess = false;
368            pageTick = 0;
369            frameTick = 0;
370
371            for (i = 0; i < numberOfProcesses - 1; i++) {
372               processArray[i].pid = processArray[i+1].pid;
373               processArray[i].pCounter = processArray[i+1].pCounter;
374               processArray[i].isComplete = processArray[i+1].isComplete;
375               processArray[i].isRunning = processArray[i+1].isRunning;
376            }
377
378            processArray[numberOfProcesses - 1].pid = 0;
379            processArray[numberOfProcesses - 1].pCounter = 0;
380            processArray[numberOfProcesses - 1].isComplete = false;
381            processArray[numberOfProcesses - 1].isRunning = false;
382         }
383      }
384
385   // ******************** UI ****************
386
387      void run(){
388         if(currentUser > 0 && userArray[currentUser].hasProcess == false) {
389            loader();
390            processArray[semaphore.count].pid = currentUser;
391            semaphore.count++;
392            userArray[currentUser].hasProcess = true;
393            programClock++;
394            currentTick++;
395            if (processArray[0].pid == currentUser) {
396               PC = processArray[0].pCounter;
397               processArray[0].isRunning = true;
398               interpreter();
399            }
400            else {
401               printf("A processes is already running. Your process has been added to the
                  queue.\n");
402            }
403            validCommand = true;
404         }
405         else if (currentUser > 0 && userArray[currentUser].hasProcess == true) {
406            printf("Your process is already queued. Please wait.\n");
407         }
408         else {
```

```
409            printf("You are not authorized to issue that command.\n");
410         }
411      }
412
413      // This will create a dump of the data in the program
414      void dmp() {
415         programClock++;
416         currentTick++;
417
418         char reg_names [4] = {'A', '1', '2', '3'};
419         int i = 0;
420
421         printf("REGISTERS\n--------------------------------\n");
422         while(i < MAX_REGISTER) {
423            printf("%1c    ", reg_names[i]);
424            printHex(Registers[i]);
425            printBin(Registers[i]);
426            printf("\n");
427            ++i;
428         }
429
430         printf("PC   ");
431         printHex(PC);
432         printBin(PC);
433         printf("\n");
434
435         printf("CC   ");
436         printHex(CC);
437         printBin(CC);
438         printf("\n");
439
440         printf("IR   ");
441         printHex(IR);
442         printBin(IR);
443         printf("\n");
444
445         printf("programClock: %d\n", programClock);
446
447         printf("\nMEMORY\n--------------------------------\n");
448
449         for (i = 0; i < MAX_MEMORY; i++) {
450            if(mainMemory[i] != 0x0000){
451               printf("%-3d  ", i);
452               printHex(mainMemory[i]);
453               printBin(mainMemory[i]);
454               printf("\n");
455            }
456         }
457
458         printf("\nDISK\n--------------------------------\n");
459
460         for (i = 0; i < DISK_SIZE; i++) {
461            if(disk[i] != 0x0000){
```

```
462                printf("%-3d  ", i);
463                printHex(disk[i]);
464                printBin(disk[i]);
465                printf("\n");
466             }
467          }
468
469          printf("\nPROCESS QUEUE\n--------------------------------\n");
470
471          if (processArray[0].pid > 0) {
472             printf("Process Owner\tMem Location\t Is Running\t\n");
473             for (i = 0; i < numberOfProcesses; i++) {
474                if (processArray[i].pid != 0) {
475                   printf("\t%d\t\t%d\t\t", processArray[i].pid, processArray[i].pCounter);
476                   if (processArray[i].isRunning == true) {
477                      printf("*\n");
478                   }
479                   else {
480                      printf("-\n");
481                   }
482                }
483             }
484          }
485          else {
486             printf("Process queue empty\n");
487          }
488
489          printf("\n--------------------------------\nDUMP
             COMPLETE\n--------------------------------\n");
490       }
491
492       // Called from halt instruction
493       void dumpPageTable(){
494          printf("\nUser %d Page Table\n", currentUser);
495          int h,k;
496          printf("Page \t| \tFrame\n");
497          for (h = 0; h < (MAX_MEMORY / 4); h++){
498             if(userArray[currentUser].pageTable[h] > -1){
499                printf("%d \t| \t%d\n",h,userArray[currentUser].pageTable[h]);
500                printf("\t\t\tFrame %d Contents\n", userArray[currentUser].pageTable[h]);
501                for(k = 0; k < 4; k++){
502                   printf("\t\t\t\t%d: ",userArray[currentUser].pageTable[h]*4 + k);
503                   printHex(mainMemory[userArray[currentUser].pageTable[h]*4 + k]);
504                   printBin(mainMemory[userArray[currentUser].pageTable[h]*4 + k]);
505                   printf("\n");
506                }
507                usedFrames[h] = 0;
508             }
509          }
510       }
511
512       void stp(){
513
```

```
514         if(currentUser == 0){
515            stopOS = true;
516            dmp();
517            printf("\n\tMachine halted.\n\n");
518         }
519         else {
520            printf("You are not authorized to issue that command.\n");
521         }
522         validCommand = true;
523      }
524
525
526   // ******************* MEMORY *******************
527
528      void mmu(int page, int frame) {
529         int pageNum;
530         int offset;
531         if (page == -1 && frame == -1 && opcode == 1) {
532            pageNum = address & 252;
533            pageNum = pageNum >> 2;
534            offset = address & 3;
535            address = (userArray[currentUser].pageTable[pageNum] * 4) + offset;
536         }
537         else if (page > -1 && frame > -1) {
538            userArray[currentUser].pageTable[page] = frame;
539         }
540
541         int i;
542         if(page == -2 && frame == -2){
543            // Below is code to clean up users page table, zero's out their table (as they
               should only have one processes in queue)
544            for(i = 0; i < MAX_MEMORY/4; i++){
545               userArray[currentUser].pageTable[i] = -1;
546            }
547         }
548      }
549
550
551   // ******************* INTERPRETER *******************
552
553      // Fetches next instruction from mainMemory, then increments PC
554      void Fetch() {
555         IR = mainMemory[PC];
556         PC++;
557      }
558
559      // Decode instructions into four fields: opcode, mode, register, address
560      void Decode() {
561
562         char temp[16];
563         char* tempPointer = temp;
564
565         unsigned int i = 1<<(sizeof(IR) * 8-1);
```

```
566
567         int count = 0;
568         int k = 0;
569
570         while(i > 0){
571            if(IR & i)
572               temp[k] = '1';
573            else
574               temp[k] = '0';
575            i >>= 1;
576
577            ++k;
578
579            if(count == 3){
580               opcode = convertNumber(tempPointer);
581               k = 0;
582            }
583            else if(count == 4){
584               if(temp[k-1] == '0')
585                  mode = 0;
586               else
587                  mode = 1;
588               k = 0;
589            }
590            else if(count == 7){
591               temp[k] = 0;
592               reg = convertNumber(tempPointer);
593               k = 0;
594            }
595            else if(count == 15){
596               address = (short)convertNumber(tempPointer);
597               k = 0;
598            }
599
600            ++count;
601         }
602      }
603
604      // Based on opcode, execute the instruction
605      void Execute() {
606         switch (opcode) {
607            case 0:     load(mainMemory, Registers);
608               break;
609            case 1:     store(mainMemory, Registers);
610               break;
611            case 2:     add(mainMemory, Registers);
612               break;
613            case 3:     sub(mainMemory, Registers);
614               break;
615            case 4:     adr(mainMemory, Registers);
616               break;
617            case 5:     sur(mainMemory, Registers);
618               break;
```

```
619             case 6:     and(mainMemory, Registers);
620                break;
621             case 7:     or(mainMemory, Registers);
622                break;
623             case 8:     not(mainMemory, Registers);
624                break;
625             case 9:     jmp(mainMemory);
626                break;
627             case 10:    jeq(mainMemory);
628                break;
629             case 11:    jgt(mainMemory);
630                break;
631             case 12:    jlt(mainMemory);
632                break;
633             case 13:    compare(mainMemory, Registers);
634                break;
635             case 14:    clear(Registers);
636                break;
637             case 15:    halt();
638                break;
639             default:
640                break;
641          }
642          programClock++;
643          currentTick++;
644          frameTick++;
645       }
646
647
648    // ******************** FUNCTIONS ********************
649
650       // Converts the string into an unsigned short
651       unsigned short convertNumber(char* num){
652          return (unsigned short)strtoul(num, NULL, 2);
653       }
654
655       // Prints the passed integer in binary format
656       void printBin(unsigned short a) {
657
658          unsigned int i;
659          i = 1<<(sizeof(a) * 8-1);
660          int k = 0;
661
662          while(i > 0) {
663             if(a & i)
664                printf("1");
665             else
666                printf("0");
667             i >>= 1;
668             ++k;
669             if(k == 4){
670                printf(" ");
671                k = 0;
```

```
672            }
673         }
674      }
675
676      // Prints the passed integer in hex format
677      void printHex(unsigned short a) {
678         printf("x%04X    ", a);
679      }
680
681      // Sets condition code of register to positive, zero, or negative
682      void changeCondition(int regValue) {
683         if (Registers[regValue] > 0) CC = 1;
684         else if (Registers[regValue] == 0) CC = 2;
685         else if (Registers[regValue] < 0) CC = 4;
686         else {}
687      }
688
```