Source Code Listing of NACHOS-3.4

Dr. Peiyi Tang
Department of Computer Science
University of Arkansas at Little Rock
Little Rock, AR 72204
U.S.A.

Copyright Statement

```
/*
Copyright (c) 1992-1993 The Regents of the University of California.
All rights reserved.

Permission to use, copy, modify, and distribute this software and its
documentation for any purpose, without fee, and without written agreement is
hereby granted, provided that the above copyright notice and the following
two paragraphs appear in all copies of this software.

IN NO EVENT SHALL THE UNIVERSITY OF CALIFORNIA BE LIABLE TO ANY PARTY FOR
DIRECT, INDIRECT, SPECIAL, INCIDENTAL, OR CONSEQUENTIAL DAMAGES ARISING OUT
OF THE USE OF THIS SOFTWARE AND ITS DOCUMENTATION, EVEN IF THE UNIVERSITY OF
CALIFORNIA HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

THE UNIVERSITY OF CALIFORNIA SPECIFICALLY DISCLAIMS ANY WARRANTIES,
INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY
AND FITNESS FOR A PARTICULAR PURPOSE.  THE SOFTWARE PROVIDED HEREUNDER IS
ON AN "AS IS" BASIS, AND THE UNIVERSITY OF CALIFORNIA HAS NO OBLIGATION TO
PROVIDE MAINTENANCE, SUPPORT, UPDATES, ENHANCEMENTS, OR MODIFICATIONS.
*/

#ifdef MAIN     /* include the copyright message in every executable */
static char *copyright = "Copyright (c) 1992-1993 The Regents of the
University of California.  All rights reserved.";
#endif // MAIN
```

Contents

1 Directory ../threads/ **1**

1.1 copyright.h . . . 1
1.2 list.h . . . 2
1.3 list.cc . . . 3
1.4 main.cc . . . 7
1.5 scheduler.h . . . 10
1.6 scheduler.cc . . . 11
1.7 switch.h . . . 13
1.8 switch.s . . . 17
1.9 synch.h . . . 24
1.10 synch.cc . . . 27
1.11 synchlist.h . . . 31
1.12 synchlist.cc . . . 32
1.13 system.h . . . 34
1.14 system.cc . . . 35
1.15 thread.h . . . 38
1.16 thread.cc . . . 41
1.17 threadtest.cc . . . 47
1.18 utility.h . . . 47
1.19 utility.cc . . . 49

2 Directory ../filesys/ **53**

2.1 directory.h . . . 53
2.2 directory.cc . . . 55
2.3 filehdr.h . . . 58
2.4 filehdr.cc . . . 59
2.5 filesys.h . . . 62
2.6 filesys.cc . . . 64
2.7 fstest.cc . . . 70
2.8 openfile.h . . . 73
2.9 openfile.cc . . . 75
2.10 synchdisk.h . . . 78
2.11 synchdisk.h . . . 79

3 Directory ../machine/ **81**

3.1 console.h . . . 81
3.2 console.cc . . . 83
3.3 disk.h . . . 85
3.4 disk.cc . . . 87
3.5 interrupt.h . . . 92
3.6 interrupt.cc . . . 94
3.7 machine.h . . . 101

3.8 machine.cc . . . 104
3.9 mipssim.h . . . 108
3.10 mipssim.cc . . . 112
3.11 network.h . . . 124
3.12 network.cc . . . 127
3.13 stats.h . . . 129
3.14 stats.cc . . . 131
3.15 sysdep.h . . . 131
3.16 sysdep.cc . . . 133
3.17 timer.h . . . 142
3.18 timer.cc . . . 143
3.19 translate.h . . . 144
3.20 translate.cc . . . 145

4 Directory ../userprog/ 151
4.1 addrspace.h . . . 151
4.2 addrspace.cc . . . 152
4.3 bitmap.h . . . 155
4.4 bitmap.cc . . . 156
4.5 exception.cc . . . 159
4.6 progtest.cc . . . 160
4.7 syscall.h . . . 162

5 Directory ../bin/ 165
5.1 coff.h . . . 165
5.2 coff2flat.c . . . 166
5.3 coff2noff.c . . . 169
5.4 d.c . . . 173
5.5 disasm.c . . . 177
5.6 encode.h . . . 180
5.7 execute.c . . . 182
5.8 instr.h . . . 192
5.9 int.h . . . 192
5.10 main.c . . . 193
5.11 noff.h . . . 196
5.12 opstrings.c . . . 196
5.13 out.c . . . 199
5.14 system.c . . . 203

6 Directory ../test/ 207
6.1 halt-a.s . . . 207
6.2 halt.c . . . 208
6.3 matmult.c . . . 208
6.4 shell.c . . . 209
6.5 sort.c . . . 209
6.6 start.s . . . 210

7 Directory ../network/ 215
7.1 nettest.cc . . . 215
7.2 post.cc . . . 216
7.3 post.h . . . 222

Preface

This document provides the source code listing of NACHOS Operating System 3.4 written by Professor Thomas Anderson at University of California at Berkeley (now University of Washington).

The NACHOS-3.4 is a small but complete operating system designed as a teaching tool for operating system courses in computer science. It includes about 9500 lines of C++ code, about one third of which are comments.

The source code is downloaded from `http://www.cs.washington.edu/homes/tom/nachos/`
as file `nachos-3.4.tar.Z`

I use `Nachos`-3.4 for the laboratory work and assignments in the Operating Systems course (CPSC 3380) at University of Arkansas at Little Rock starting from 2003. We compile this book for the students of the class so that they can read the entire source code in one place.

Chapter 1

Directory ../threads/

Contents

1.1 copyright.h . . . 1
1.2 list.h . . . 2
1.3 list.cc . . . 3
1.4 main.cc . . . 7
1.5 scheduler.h . . . 10
1.6 scheduler.cc . . . 11
1.7 switch.h . . . 13
1.8 switch.s . . . 17
1.9 synch.h . . . 24
1.10 synch.cc . . . 27
1.11 synchlist.h . . . 31
1.12 synchlist.cc . . . 32
1.13 system.h . . . 34
1.14 system.cc . . . 35
1.15 thread.h . . . 38
1.16 thread.cc . . . 41
1.17 threadtest.cc . . . 47
1.18 utility.h . . . 47
1.19 utility.cc . . . 49

This chapter lists all the source codes found in directory `../threads/`. They are:

```
copyright.h     scheduler.cc    synch.cc        system.cc       threadtest.cc
list.cc         scheduler.h     synch.h         system.h        utility.cc
list.h          switch.h        synchlist.cc    thread.cc       utility.h
main.cc         switch.s        synchlist.h     thread.h
```

1.1 copyright.h

```
 1 /*
 2 Copyright (c) 1992-1993 The Regents of the University of California.
 3 All rights reserved.
 4
 5 Permission to use, copy, modify, and distribute this software and its
 6 documentation for any purpose, without fee, and without written agreement is
 7 hereby granted, provided that the above copyright notice and the following
```

```
 8 two paragraphs appear in all copies of this software.
 9
10 IN NO EVENT SHALL THE UNIVERSITY OF CALIFORNIA BE LIABLE TO ANY PARTY FOR
11 DIRECT, INDIRECT, SPECIAL, INCIDENTAL, OR CONSEQUENTIAL DAMAGES ARISING OUT
12 OF THE USE OF THIS SOFTWARE AND ITS DOCUMENTATION, EVEN IF THE UNIVERSITY OF
13 CALIFORNIA HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.
14
15 THE UNIVERSITY OF CALIFORNIA SPECIFICALLY DISCLAIMS ANY WARRANTIES,
16 INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY
17 AND FITNESS FOR A PARTICULAR PURPOSE.  THE SOFTWARE PROVIDED HEREUNDER IS
18 ON AN "AS IS" BASIS, AND THE UNIVERSITY OF CALIFORNIA HAS NO OBLIGATION TO
19 PROVIDE MAINTENANCE, SUPPORT, UPDATES, ENHANCEMENTS, OR MODIFICATIONS.
20 */
21
22 #ifdef MAIN     /* include the copyright message in every executable */
23 static char *copyright = "Copyright (c) 1992-1993 The Regents of
   the University of California.  All rights reserved.";
24 #endif // MAIN
```

1.2 list.h

```
 1 // list.h
 2 //      Data structures to manage LISP-like lists.
 3 //
 4 //      As in LISP, a list can contain any type of data structure
 5 //      as an item on the list: thread control blocks,
 6 //      pending interrupts, etc.  That is why each item is a "void *",
 7 //      or in other words, a "pointers to anything".
 8 //
 9 // Copyright (c) 1992-1993 The Regents of the University of California.
10 // All rights reserved.  See copyright.h for copyright notice and limitation
11 // of liability and disclaimer of warranty provisions.
12
13 #ifndef LIST_H
14 #define LIST_H
15
16 #include "copyright.h"
17 #include "utility.h"
18
19 // The following class defines a "list element" -- which is
20 // used to keep track of one item on a list.  It is equivalent to a
21 // LISP cell, with a "car" ("next") pointing to the next element on the list,
22 // and a "cdr" ("item") pointing to the item on the list.
23 //
24 // Internal data structures kept public so that List operations can
25 // access them directly.
26
27 class ListElement {
28   public:
29     ListElement(void *itemPtr, int sortKey);   // initialize a list element
30
31     ListElement *next;         // next element on list,
32                                // NULL if this is the last
33     int key;                   // priority, for a sorted list
34     void *item;                // pointer to item on the list
35 };
36
```

```
37 // The following class defines a "list" -- a singly linked list of
38 // list elements, each of which points to a single item on the list.
39 //
40 // By using the "Sorted" functions, the list can be kept in sorted
41 // in increasing order by "key" in ListElement.
42
43 class List {
44   public:
45     List();                     // initialize the list
46     ~List();                    // de-allocate the list
47
48     void Prepend(void *item);   // Put item at the beginning of the list
49     void Append(void *item);    // Put item at the end of the list
50     void *Remove();             // Take item off the front of the list
51
52     void Mapcar(VoidFunctionPtr func);  // Apply "func" to every element
53                                         // on the list
54     bool IsEmpty();             // is the list empty?
55
56
57     // Routines to put/get items on/off list in order (sorted by key)
58     void SortedInsert(void *item, int sortKey); // Put item into list
59     void *SortedRemove(int *keyPtr);            // Remove first item from list
60
61   private:
62     ListElement *first;         // Head of the list, NULL if list is empty
63     ListElement *last;          // Last element of list
64 };
65
66 #endif // LIST_H
```

1.3 list.cc

```
 1 // list.cc
 2 //
 3 //      Routines to manage a singly-linked list of "things".
 4 //
 5 //      A "ListElement" is allocated for each item to be put on the
 6 //      list; it is de-allocated when the item is removed. This means
 7 //      we don't need to keep a "next" pointer in every object we
 8 //      want to put on a list.
 9 //
10 //      NOTE: Mutual exclusion must be provided by the caller.
11 //      If you want a synchronized list, you must use the routines
12 //      in synchlist.cc.
13 //
14 // Copyright (c) 1992-1993 The Regents of the University of California.
15 // All rights reserved.  See copyright.h for copyright notice and limitation
16 // of liability and disclaimer of warranty provisions.
17
18 #include "copyright.h"
19 #include "list.h"
20
21 //----------------------------------------------------------------------
22 // ListElement::ListElement
23 //      Initialize a list element, so it can be added somewhere on a list.
24 //
```

```
25 //      "itemPtr" is the item to be put on the list.  It can be a pointer
26 //              to anything.
27 //      "sortKey" is the priority of the item, if any.
28 //----------------------------------------------------------------------
29
30 ListElement::ListElement(void *itemPtr, int sortKey)
31 {
32      item = itemPtr;
33      key = sortKey;
34      next = NULL;       // assume we'll put it at the end of the list
35 }
36
37 //----------------------------------------------------------------------
38 // List::List
39 //      Initialize a list, empty to start with.
40 //      Elements can now be added to the list.
41 //----------------------------------------------------------------------
42
43 List::List()
44 {
45     first = last = NULL;
46 }
47
48 //----------------------------------------------------------------------
49 // List::~List
50 //      Prepare a list for deallocation.  If the list still contains any
51 //      ListElements, de-allocate them.  However, note that we do *not*
52 //      de-allocate the "items" on the list -- this module allocates
53 //      and de-allocates the ListElements to keep track of each item,
54 //      but a given item may be on multiple lists, so we can't
55 //      de-allocate them here.
56 //----------------------------------------------------------------------
57
58 List::~List()
59 {
60     while (Remove() != NULL)
61         ;        // delete all the list elements
62 }
63
64 //----------------------------------------------------------------------
65 // List::Append
66 //      Append an "item" to the end of the list.
67 //
68 //      Allocate a ListElement to keep track of the item.
69 //      If the list is empty, then this will be the only element.
70 //      Otherwise, put it at the end.
71 //
72 //      "item" is the thing to put on the list, it can be a pointer to
73 //              anything.
74 //----------------------------------------------------------------------
75
76 void
77 List::Append(void *item)
78 {
79     ListElement *element = new ListElement(item, 0);
80
81     if (IsEmpty()) {            // list is empty
82         first = element;
```

```
 83         last = element;
 84     } else {                    // else put it after last
 85         last->next = element;
 86         last = element;
 87     }
 88 }
 89
 90 //----------------------------------------------------------------------
 91 // List::Prepend
 92 //      Put an "item" on the front of the list.
 93 //
 94 //      Allocate a ListElement to keep track of the item.
 95 //      If the list is empty, then this will be the only element.
 96 //      Otherwise, put it at the beginning.
 97 //
 98 //      "item" is the thing to put on the list, it can be a pointer to
 99 //              anything.
100 //----------------------------------------------------------------------
101
102 void
103 List::Prepend(void *item)
104 {
105     ListElement *element = new ListElement(item, 0);
106
107     if (IsEmpty()) {            // list is empty
108         first = element;
109         last = element;
110     } else {                    // else put it before first
111         element->next = first;
112         first = element;
113     }
114 }
115
116 //----------------------------------------------------------------------
117 // List::Remove
118 //      Remove the first "item" from the front of the list.
119 //
120 // Returns:
121 //      Pointer to removed item, NULL if nothing on the list.
122 //----------------------------------------------------------------------
123
124 void *
125 List::Remove()
126 {
127     return SortedRemove(NULL);  // Same as SortedRemove, but ignore the key
128 }
129
130 //----------------------------------------------------------------------
131 // List::Mapcar
132 //      Apply a function to each item on the list, by walking through
133 //       the list, one element at a time.
134 //
135 //      Unlike LISP, this mapcar does not return anything!
136 //
137 //      "func" is the procedure to apply to each element of the list.
138 //----------------------------------------------------------------------
139
140 void
```

```
141 List::Mapcar(VoidFunctionPtr func)
142 {
143     for (ListElement *ptr = first; ptr != NULL; ptr = ptr->next) {
144        DEBUG('l', "In mapcar, about to invoke %x(%x)\n", func, ptr->item);
145        (*func)((_int)ptr->item);
146     }
147 }
148
149 //----------------------------------------------------------------------
150 // List::IsEmpty
151 //      Returns TRUE if the list is empty (has no items).
152 //----------------------------------------------------------------------
153
154 bool
155 List::IsEmpty()
156 {
157     if (first == NULL)
158         return TRUE;
159     else
160         return FALSE;
161 }
162
163 //----------------------------------------------------------------------
164 // List::SortedInsert
165 //      Insert an "item" into a list, so that the list elements are
166 //      sorted in increasing order by "sortKey".
167 //
168 //      Allocate a ListElement to keep track of the item.
169 //      If the list is empty, then this will be the only element.
170 //      Otherwise, walk through the list, one element at a time,
171 //      to find where the new item should be placed.
172 //
173 //      "item" is the thing to put on the list, it can be a pointer to
174 //              anything.
175 //      "sortKey" is the priority of the item.
176 //----------------------------------------------------------------------
177
178 void
179 List::SortedInsert(void *item, int sortKey)
180 {
181     ListElement *element = new ListElement(item, sortKey);
182     ListElement *ptr;           // keep track
183
184     if (IsEmpty()) {    // if list is empty, put
185         first = element;
186         last = element;
187     } else if (sortKey < first->key) {
188                 // item goes on front of list
189         element->next = first;
190         first = element;
191     } else {            // look for first elt in list bigger than item
192         for (ptr = first; ptr->next != NULL; ptr = ptr->next) {
193             if (sortKey < ptr->next->key) {
194                 element->next = ptr->next;
195                 ptr->next = element;
196                 return;
197             }
198         }
```

```
199          last->next = element;           // item goes at end of list
200         last = element;
201     }
202 }
203
204 //----------------------------------------------------------------------
205 // List::SortedRemove
206 //      Remove the first "item" from the front of a sorted list.
207 //
208 // Returns:
209 //      Pointer to removed item, NULL if nothing on the list.
210 //      Sets *keyPtr to the priority value of the removed item
211 //      (this is needed by interrupt.cc, for instance).
212 //
213 //      "keyPtr" is a pointer to the location in which to store the
214 //              priority of the removed item.
215 //----------------------------------------------------------------------
216
217 void *
218 List::SortedRemove(int *keyPtr)
219 {
220     ListElement *element = first;
221     void *thing;
222
223     if (IsEmpty())
224         return NULL;
225
226     thing = first->item;
227     if (first == last) {        // list had one item, now has none
228         first = NULL;
229         last = NULL;
230     } else {
231         first = element->next;
232     }
233     if (keyPtr != NULL)
234         *keyPtr = element->key;
235     delete element;
236     return thing;
237 }
238
```

1.4 main.cc

```
 1 // main.cc
 2 //      Bootstrap code to initialize the operating system kernel.
 3 //
 4 //      Allows direct calls into internal operating system functions,
 5 //      to simplify debugging and testing.  In practice, the
 6 //      bootstrap code would just initialize data structures,
 7 //      and start a user program to print the login prompt.
 8 //
 9 //      Most of this file is not needed until later assignments.
10 //
11 // Usage: nachos -d <debugflags> -rs <random seed #>
12 //              -s -x <nachos file> -c <consoleIn> <consoleOut>
13 //              -f -cp <unix file> <nachos file>
14 //              -p <nachos file> -r <nachos file> -l -D -t
```

```
15 //              -n <network reliability> -e <network orderability>
16 //              -m <machine id>
17 //              -o <other machine id>
18 //              -z
19 //
20 //    -d causes certain debugging messages to be printed (cf. utility.h)
21 //    -rs causes Yield to occur at random (but repeatable) spots
22 //    -z prints the copyright message
23 //
24 //  USER_PROGRAM
25 //    -s causes user programs to be executed in single-step mode
26 //    -x runs a user program
27 //    -c tests the console
28 //
29 //  FILESYS
30 //    -f causes the physical disk to be formatted
31 //    -cp copies a file from UNIX to Nachos
32 //    -p prints a Nachos file to stdout
33 //    -r removes a Nachos file from the file system
34 //    -l lists the contents of the Nachos directory
35 //    -D prints the contents of the entire file system
36 //    -t tests the performance of the Nachos file system
37 //
38 //  NETWORK
39 //    -n sets the network reliability
40 //    -e sets the network orderability
41 //    -m sets this machine's host id (needed for the network)
42 //    -o runs a simple test of the Nachos network software
43 //
44 //  NOTE -- flags are ignored until the relevant assignment.
45 //  Some of the flags are interpreted here; some in system.cc.
46 //
47 // Copyright (c) 1992-1993 The Regents of the University of California.
48 // All rights reserved.  See copyright.h for copyright notice and limitation
49 // of liability and disclaimer of warranty provisions.
50
51 #define MAIN
52 #include "copyright.h"
53 #undef MAIN
54
55 #include "utility.h"
56 #include "system.h"
57
58
59 // External functions used by this file
60
61 extern void ThreadTest(void), Copy(char *unixFile, char *nachosFile);
62 extern void Print(char *file), PerformanceTest(void);
63 extern void StartProcess(char *file), ConsoleTest(char *in, char *out);
64 extern void MailTest(int networkID);
65 extern void SynchTest(void);
66
67 //----------------------------------------------------------------------
68 // main
69 //      Bootstrap the operating system kernel.
70 //
71 //      Check command line arguments
72 //      Initialize data structures
```

```
73 //      (optionally) Call test procedure
74 //
75 //      "argc" is the number of command line arguments (including the name
76 //              of the command) -- ex: "nachos -d +" -> argc = 3
77 //      "argv" is an array of strings, one for each command line argument
78 //              ex: "nachos -d +" -> argv = {"nachos", "-d", "+"}
79 //----------------------------------------------------------------------
80
81 int
82 main(int argc, char **argv)
83 {
84     int argCount;                       // the number of arguments
85                                         // for a particular command
86
87     DEBUG('t', "Entering main");
88     (void) Initialize(argc, argv);
89
90 #ifdef THREADS
91     ThreadTest();
92 #if 0
93     SynchTest();
94 #endif
95 #endif
96
97     for (argc--, argv++; argc > 0; argc -= argCount, argv += argCount) {
98         argCount = 1;
99         if (!strcmp(*argv, "-z"))               // print copyright
100            printf (copyright);
101 #ifdef USER_PROGRAM
102        if (!strcmp(*argv, "-x")) {             // run a user program
103            ASSERT(argc > 1);
104            StartProcess(*(argv + 1));
105            argCount = 2;
106        } else if (!strcmp(*argv, "-c")) {      // test the console
107            if (argc == 1)
108                ConsoleTest(NULL, NULL);
109            else {
110                ASSERT(argc > 2);
111                ConsoleTest(*(argv + 1), *(argv + 2));
112                argCount = 3;
113            }
114            interrupt->Halt();          // once we start the console, then
115                                        // Nachos will loop forever waiting
116                                        // for console input
117        }
118 #endif // USER_PROGRAM
119 #ifdef FILESYS
120        if (!strcmp(*argv, "-cp")) {            // copy from UNIX to Nachos
121            ASSERT(argc > 2);
122            Copy(*(argv + 1), *(argv + 2));
123            argCount = 3;
124        } else if (!strcmp(*argv, "-p")) {      // print a Nachos file
125            ASSERT(argc > 1);
126            Print(*(argv + 1));
127            argCount = 2;
128        } else if (!strcmp(*argv, "-r")) {      // remove Nachos file
129            ASSERT(argc > 1);
130            fileSystem->Remove(*(argv + 1));
```

```
131             argCount = 2;
132         } else if (!strcmp(*argv, "-l")) {      // list Nachos directory
133             fileSystem->List();
134         } else if (!strcmp(*argv, "-D")) {      // print entire filesystem
135             fileSystem->Print();
136         } else if (!strcmp(*argv, "-t")) {      // performance test
137             PerformanceTest();
138         }
139 #endif // FILESYS
140 #ifdef NETWORK
141         if (!strcmp(*argv, "-o")) {
142             ASSERT(argc > 1);
143             Delay(2);                           // delay for 2 seconds
144                                                 // to give the user time to
145                                                 // start up another nachos
146             MailTest(atoi(*(argv + 1)));
147             argCount = 2;
148         }
149 #endif // NETWORK
150     }
151
152     currentThread->Finish();    // NOTE: if the procedure "main"
153                                 // returns, then the program "nachos"
154                                 // will exit (as any other normal program
155                                 // would).  But there may be other
156                                 // threads on the ready list.  We switch
157                                 // to those threads by saying that the
158                                 // "main" thread is finished, preventing
159                                 // it from returning.
160     return(0);                  // Not reached...
161 }
```

1.5 scheduler.h

```
 1 // scheduler.h
 2 //      Data structures for the thread dispatcher and scheduler.
 3 //      Primarily, the list of threads that are ready to run.
 4 //
 5 // Copyright (c) 1992-1993 The Regents of the University of California.
 6 // All rights reserved.  See copyright.h for copyright notice and limitation
 7 // of liability and disclaimer of warranty provisions.
 8
 9 #ifndef SCHEDULER_H
10 #define SCHEDULER_H
11
12 #include "copyright.h"
13 #include "list.h"
14 #include "thread.h"
15
16 // The following class defines the scheduler/dispatcher abstraction --
17 // the data structures and operations needed to keep track of which
18 // thread is running, and which threads are ready but not running.
19
20 class Scheduler {
21   public:
22     Scheduler();                        // Initialize list of ready threads
23     ~Scheduler();                       // De-allocate ready list
```

```
24
25     void ReadyToRun(Thread* thread);    // Thread can be dispatched.
26     Thread* FindNextToRun();            // Dequeue first thread on the ready
27                                         // list, if any, and return thread.
28     void Run(Thread* nextThread);       // Cause nextThread to start running
29     void Print();                       // Print contents of ready list
30
31   private:
32     List *readyList;            // queue of threads that are ready to run,
33                                 // but not running
34 };
35
36 #endif // SCHEDULER_H
```

1.6 scheduler.cc

```
 1 // scheduler.cc
 2 //      Routines to choose the next thread to run, and to dispatch to
 3 //      that thread.
 4 //
 5 //      These routines assume that interrupts are already disabled.
 6 //      If interrupts are disabled, we can assume mutual exclusion
 7 //      (since we are on a uniprocessor).
 8 //
 9 //      NOTE: We can't use Locks to provide mutual exclusion here, since
10 //      if we needed to wait for a lock, and the lock was busy, we would
11 //      end up calling FindNextToRun(), and that would put us in an
12 //      infinite loop.
13 //
14 //      Very simple implementation -- no priorities, straight FIFO.
15 //      Might need to be improved in later assignments.
16 //
17 // Copyright (c) 1992-1993 The Regents of the University of California.
18 // All rights reserved.  See copyright.h for copyright notice and limitation
19 // of liability and disclaimer of warranty provisions.
20
21 #include "copyright.h"
22 #include "scheduler.h"
23 #include "system.h"
24
25 //----------------------------------------------------------------------
26 // Scheduler::Scheduler
27 //      Initialize the list of ready but not running threads to empty.
28 //----------------------------------------------------------------------
29
30 Scheduler::Scheduler()
31 {
32     readyList = new List;
33 }
34
35 //----------------------------------------------------------------------
36 // Scheduler::~Scheduler
37 //      De-allocate the list of ready threads.
38 //----------------------------------------------------------------------
39
40 Scheduler::~Scheduler()
41 {
```

```
42     delete readyList;
43 }
44
45 //----------------------------------------------------------------------
46 // Scheduler::ReadyToRun
47 //      Mark a thread as ready, but not running.
48 //      Put it on the ready list, for later scheduling onto the CPU.
49 //
50 //      "thread" is the thread to be put on the ready list.
51 //----------------------------------------------------------------------
52
53 void
54 Scheduler::ReadyToRun (Thread *thread)
55 {
56     DEBUG('t', "Putting thread %s on ready list.\n", thread->getName());
57
58     thread->setStatus(READY);
59     readyList->Append((void *)thread);
60 }
61
62 //----------------------------------------------------------------------
63 // Scheduler::FindNextToRun
64 //      Return the next thread to be scheduled onto the CPU.
65 //      If there are no ready threads, return NULL.
66 // Side effect:
67 //       Thread is removed from the ready list.
68 //----------------------------------------------------------------------
69
70 Thread *
71 Scheduler::FindNextToRun ()
72 {
73     return (Thread *)readyList->Remove();
74 }
75
76 //----------------------------------------------------------------------
77 // Scheduler::Run
78 //      Dispatch the CPU to nextThread.  Save the state of the old thread,
79 //      and load the state of the new thread, by calling the machine
80 //      dependent context switch routine, SWITCH.
81 //
82 //      Note: we assume the state of the previously running thread has
83 //      already been changed from running to blocked or ready (depending).
84 // Side effect:
85 //      The global variable currentThread becomes nextThread.
86 //
87 //      "nextThread" is the thread to be put into the CPU.
88 //----------------------------------------------------------------------
89
90 void
91 Scheduler::Run (Thread *nextThread)
92 {
93     Thread *oldThread = currentThread;
94
95 #ifdef USER_PROGRAM                     // ignore until running user programs
96     if (currentThread->space != NULL) { // if this thread is a user program,
97         currentThread->SaveUserState(); // save the user's CPU registers
98         currentThread->space->SaveState();
99     }
```

```
100 #endif
101
102     oldThread->CheckOverflow();             // check if the old thread
103                                             // had an undetected stack overflow
104
105     currentThread = nextThread;             // switch to the next thread
106     currentThread->setStatus(RUNNING);      // nextThread is now running
107
108     DEBUG('t', "Switching from thread \"%s\" to thread \"%s\"\n",
109           oldThread->getName(), nextThread->getName());
110
111     // This is a machine-dependent assembly language routine defined
112     // in switch.s.  You may have to think
113     // a bit to figure out what happens after this, both from the point
114     // of view of the thread and from the perspective of the "outside world".
115
116     SWITCH(oldThread, nextThread);
117
118     DEBUG('t', "Now in thread \"%s\"\n", currentThread->getName());
119
120     // If the old thread gave up the processor because it was finishing,
121     // we need to delete its carcass.  Note we cannot delete the thread
122     // before now (for example, in Thread::Finish()), because up to this
123     // point, we were still running on the old thread's stack!
124     if (threadToBeDestroyed != NULL) {
125         delete threadToBeDestroyed;
126         threadToBeDestroyed = NULL;
127     }
128
129 #ifdef USER_PROGRAM
130     if (currentThread->space != NULL) {         // if there is an address space
131         currentThread->RestoreUserState();     // to restore, do it.
132         currentThread->space->RestoreState();
133     }
134 #endif
135 }
136
137 //----------------------------------------------------------------------
138 // Scheduler::Print
139 //      Print the scheduler state -- in other words, the contents of
140 //      the ready list.  For debugging.
141 //----------------------------------------------------------------------
142 void
143 Scheduler::Print()
144 {
145     printf("Ready list contents:\n");
146     readyList->Mapcar((VoidFunctionPtr) ThreadPrint);
147 }
```

1.7 switch.h

```
 1 /* switch.h
 2  *      Definitions needed for implementing context switching.
 3  *
 4  *      Context switching is inherently machine dependent, since
 5  *      the registers to be saved, how to set up an initial
 6  *      call frame, etc, are all specific to a processor architecture.
```

```
 7  *
 8  *      This file currently supports the DEC MIPS, SUN SPARC, HP PA-RISC,
 9  *  Intel 386 and DEC ALPHA architectures.
10  */
11
12 /*
13  Copyright (c) 1992-1993 The Regents of the University of California.
14  All rights reserved.  See copyright.h for copyright notice and limitation
15  of liability and disclaimer of warranty provisions.
16  */
17
18 #ifndef SWITCH_H
19 #define SWITCH_H
20
21 #include "copyright.h"
22
23 #ifdef HOST_MIPS
24
25 /* Registers that must be saved during a context switch.
26  * These are the offsets from the beginning of the Thread object,
27  * in bytes, used in switch.s
28  */
29 #define SP 0
30 #define S0 4
31 #define S1 8
32 #define S2 12
33 #define S3 16
34 #define S4 20
35 #define S5 24
36 #define S6 28
37 #define S7 32
38 #define FP 36
39 #define PC 40
40
41 /* To fork a thread, we set up its saved register state, so that
42  * when we switch to the thread, it will start running in ThreadRoot.
43  *
44  * The following are the initial registers we need to set up to
45  * pass values into ThreadRoot (for instance, containing the procedure
46  * for the thread to run).  The first set is the registers as used
47  * by ThreadRoot; the second set is the locations for these initial
48  * values in the Thread object -- used in Thread::AllocateStack().
49  */
50
51 #define InitialPC       s0
52 #define InitialArg      s1
53 #define WhenDonePC      s2
54 #define StartupPC       s3
55
56 #define PCState         (PC/4-1)
57 #define FPState         (FP/4-1)
58 #define InitialPCState  (S0/4-1)
59 #define InitialArgState (S1/4-1)
60 #define WhenDonePCState (S2/4-1)
61 #define StartupPCState  (S3/4-1)
62
63 #endif  // HOST_MIPS
64
```

```
 65 #ifdef HOST_SPARC
 66
 67 /* Registers that must be saved during a context switch.  See comment above. */
 68 #define I0 4
 69 #define I1 8
 70 #define I2 12
 71 #define I3 16
 72 #define I4 20
 73 #define I5 24
 74 #define I6 28
 75 #define I7 32
 76
 77 /* Aliases used for clearing code.  */
 78 #define FP I6
 79 #define PC I7
 80
 81 /* Registers for ThreadRoot.  See comment above. */
 82 #define InitialPC       %o0
 83 #define InitialArg      %o1
 84 #define WhenDonePC      %o2
 85 #define StartupPC       %o3
 86
 87 #define PCState         (PC/4-1)
 88 #define InitialPCState  (I0/4-1)
 89 #define InitialArgState (I1/4-1)
 90 #define WhenDonePCState (I2/4-1)
 91 #define StartupPCState  (I3/4-1)
 92 #endif  // HOST_SPARC
 93
 94 #ifdef HOST_SNAKE
 95
 96 /* Registers that must be saved during a context switch.  See comment above. */
 97 #define   SP   0
 98 #define   S0   4
 99 #define   S1   8
100 #define   S2   12
101 #define   S3   16
102 #define   S4   20
103 #define   S5   24
104 #define   S6   28
105 #define   S7   32
106 #define   S8   36
107 #define   S9   40
108 #define   S10  44
109 #define   S11  48
110 #define   S12  52
111 #define   S13  56
112 #define   S14  60
113 #define   S15  64
114 #define   PC   68
115
116 /* Registers for ThreadRoot.  See comment above. */
117 #define InitialPC       %r3             /* S0 */
118 #define InitialArg      %r4
119 #define WhenDonePC      %r5
120 #define StartupPC       %r6
121
122 #define PCState         (PC/4-1)
```

```
123 #define InitialPCState  (S0/4-1)
124 #define InitialArgState (S1/4-1)
125 #define WhenDonePCState (S2/4-1)
126 #define StartupPCState  (S3/4-1)
127 #endif  // HOST_SNAKE
128
129 #ifdef HOST_i386
130
131 /* the offsets of the registers from the beginning of the thread object */
132 #define _ESP     0
133 #define _EAX     4
134 #define _EBX     8
135 #define _ECX     12
136 #define _EDX     16
137 #define _EBP     20
138 #define _ESI     24
139 #define _EDI     28
140 #define _PC      32
141
142 /* These definitions are used in Thread::AllocateStack(). */
143 #define PCState         (_PC/4-1)
144 #define FPState         (_EBP/4-1)
145 #define InitialPCState  (_ESI/4-1)
146 #define InitialArgState (_EDX/4-1)
147 #define WhenDonePCState (_EDI/4-1)
148 #define StartupPCState  (_ECX/4-1)
149
150 #define InitialPC       %esi
151 #define InitialArg      %edx
152 #define WhenDonePC      %edi
153 #define StartupPC       %ecx
154 #endif // HOST_i386
155
156 // Roberto Rossi (roberto@csr.unibo.it) - 1994
157 #ifdef HOST_ALPHA
158
159 #include <regdef.h>
160
161 /* Registers that must be saved during a context switch.
162  * These are the offsets from the beginning of the Thread object,
163  * in bytes, used in switch.s
164 */
165 #define SP  0
166 #define S0  8
167 #define S1 16
168 #define S2 24
169 #define S3 32
170 #define S4 40
171 #define S5 48
172 #define FP 56
173 #define GP 64
174 #define PC 72
175
176 /* To fork a thread, we set up its saved register state, so that
177  * when we switch to the thread, it will start running in ThreadRoot.
178  *
179  * The following are the initial registers we need to set up to
180  * pass values into ThreadRoot (for instance, containing the procedure
```

```
181  * for the thread to run).  The first set is the registers as used
182  * by ThreadRoot; the second set is the locations for these initial
183  * values in the Thread object -- used in Thread::AllocateStack().
184  */
185 #define InitialPC       s0
186 #define InitialArg      s1
187 #define WhenDonePC      s2
188 #define StartupPC       s3
189
190 #define PCState         (PC/8-1)
191 #define FPState         (FP/8-1)
192 #define InitialPCState  (S0/8-1)
193 #define InitialArgState (S1/8-1)
194 #define WhenDonePCState (S2/8-1)
195 #define StartupPCState  (S3/8-1)
196 #endif // HOST_ALPHA
197
198 #endif // SWITCH_H
```

1.8 switch.s

```
  1 /* switch.s
  2  *      Machine dependent context switch routines.  DO NOT MODIFY THESE!
  3  *
  4  *      Context switching is inherently machine dependent, since
  5  *      the registers to be saved, how to set up an initial
  6  *      call frame, etc, are all specific to a processor architecture.
  7  *
  8  *      This file currently supports the following architectures:
  9  *          DEC MIPS
 10  *          SUN SPARC
 11  *          HP PA-RISC
 12  *          Intel 386
 13  *          DEC ALPHA
 14  *
 15  * We define two routines for each architecture:
 16  *
 17  * ThreadRoot(InitialPC, InitialArg, WhenDonePC, StartupPC)
 18  *      InitialPC  - The program counter of the procedure to run
 19  *                   in this thread.
 20  *      InitialArg - The single argument to the thread.
 21  *      WhenDonePC - The routine to call when the thread returns.
 22  *      StartupPC  - Routine to call when the thread is started.
 23  *
 24  *      ThreadRoot is called from the SWITCH() routine to start
 25  *      a thread for the first time.
 26  *
 27  * SWITCH(oldThread, newThread)
 28  *      oldThread  - The current thread that was running, where the
 29  *              CPU register state is to be saved.
 30  *      newThread  - The new thread to be run, where the CPU register
 31  *              state is to be loaded from.
 32  */
 33
 34 /*
 35  Copyright (c) 1992-1993 The Regents of the University of California.
 36  All rights reserved.  See copyright.h for copyright notice and limitation
```

```
37  of liability and disclaimer of warranty provisions.
38  */
39
40 #if defined(HOST_i386) && defined(HOST_LINUX) && defined(HOST_ELF)
41 #define _ThreadRoot ThreadRoot
42 #define _SWITCH SWITCH
43 #endif
44
45 #include "copyright.h"
46 #include "switch.h"
47
48 #ifdef HOST_MIPS
49
50 /* Symbolic register names */
51 #define z       $0      /* zero register */
52 #define a0      $4      /* argument registers */
53 #define a1      $5
54 #define s0      $16     /* callee saved */
55 #define s1      $17
56 #define s2      $18
57 #define s3      $19
58 #define s4      $20
59 #define s5      $21
60 #define s6      $22
61 #define s7      $23
62 #define sp      $29     /* stack pointer */
63 #define fp      $30     /* frame pointer */
64 #define ra      $31     /* return address */
65
66         .text
67         .align  2
68
69         .globl ThreadRoot
70         .ent    ThreadRoot,0
71 ThreadRoot:
72         or      fp,z,z          # Clearing the frame pointer here
73                                 # makes gdb backtraces of thread stacks
74                                 # end here (I hope!)
75
76         jal     StartupPC       # call startup procedure
77         move    a0, InitialArg
78         jal     InitialPC       # call main procedure
79         jal     WhenDonePC      # when we are done, call clean up procedure
80
81         # NEVER REACHED
82         .end ThreadRoot
83
84         # a0 -- pointer to old Thread
85         # a1 -- pointer to new Thread
86         .globl SWITCH
87         .ent    SWITCH,0
88 SWITCH:
89         sw      sp, SP(a0)              # save new stack pointer
90         sw      s0, S0(a0)              # save all the callee-save registers
91         sw      s1, S1(a0)
92         sw      s2, S2(a0)
93         sw      s3, S3(a0)
94         sw      s4, S4(a0)
```

```
 95         sw      s5, S5(a0)
 96         sw      s6, S6(a0)
 97         sw      s7, S7(a0)
 98         sw      fp, FP(a0)              # save frame pointer
 99         sw      ra, PC(a0)              # save return address
100
101         lw      sp, SP(a1)              # load the new stack pointer
102         lw      s0, S0(a1)              # load the callee-save registers
103         lw      s1, S1(a1)
104         lw      s2, S2(a1)
105         lw      s3, S3(a1)
106         lw      s4, S4(a1)
107         lw      s5, S5(a1)
108         lw      s6, S6(a1)
109         lw      s7, S7(a1)
110         lw      fp, FP(a1)
111         lw      ra, PC(a1)              # load the return address
112
113         j       ra
114         .end SWITCH
115 #endif HOST_MIPS
116
117 #ifdef HOST_SPARC
118
119 /* NOTE!  These files appear not to exist on Solaris --
120  *  you need to find where (the SPARC-specific) MINFRAME, ST_FLUSH_WINDOWS, ...
121  *  are defined.  (I don't have a Solaris machine, so I have no way to tell.)
122  */
123 #include <sun4/trap.h>
124 #include <sun4/asm_linkage.h>
125 .seg    "text"
126
127 /* SPECIAL to the SPARC:
128  *      The first two instruction of ThreadRoot are skipped because
129  *      the address of ThreadRoot is made the return address of SWITCH()
130  *      by the routine Thread::StackAllocate.  SWITCH() jumps here on the
131  *      "ret" instruction which is really at "jmp %o7+8".  The 8 skips the
132  *      two nops at the beginning of the routine.
133  */
134
135 .globl  _ThreadRoot
136 _ThreadRoot:
137         nop  ; nop          /* These 2 nops are skipped because we are called
138                              * with a jmp+8 instruction. */
139         clr     %fp         /* Clearing the frame pointer makes gdb backtraces
140                              * of thread stacks end here. */
141                             /* Currently the arguments are in out registers we
142                              * save them into local registers so they won't be
143                              * trashed during the calls we make. */
144         mov     InitialPC, %l0
145         mov     InitialArg, %l1
146         mov     WhenDonePC, %l2
147                             /* Execute the code:
148                             *    call StartupPC();
149                             *    call InitialPC(InitialArg);
150                             *    call WhenDonePC();
151                             */
152         call    StartupPC,0
```

```
153         nop
154         call    %l0, 1
155         mov     %l1, %o0   /* Using delay slot to setup argument to InitialPC */
156         call    %l2, 0
157         nop
158                             /* WhenDonePC call should never return.  If it does
159                              * we execute a trap into the debugger.  */
160         ta      ST_BREAKPOINT
161
162
163 .globl  _SWITCH
164 _SWITCH:
165         save    %sp, -SA(MINFRAME), %sp
166         st      %fp, [%i0]
167         st      %i0, [%i0+I0]
168         st      %i1, [%i0+I1]
169         st      %i2, [%i0+I2]
170         st      %i3, [%i0+I3]
171         st      %i4, [%i0+I4]
172         st      %i5, [%i0+I5]
173         st      %i7, [%i0+I7]
174         ta      ST_FLUSH_WINDOWS
175         nop
176         mov     %i1, %l0
177         ld      [%l0+I0], %i0
178         ld      [%l0+I1], %i1
179         ld      [%l0+I2], %i2
180         ld      [%l0+I3], %i3
181         ld      [%l0+I4], %i4
182         ld      [%l0+I5], %i5
183         ld      [%l0+I7], %i7
184         ld      [%l0], %i6
185         ret
186         restore
187
188 #endif HOST_SPARC
189
190 #ifdef HOST_SNAKE
191
192     ;rp = r2,   sp = r30
193     ;arg0 = r26,  arg1 = r25,  arg2 = r24,  arg3 = r23
194
195         .SPACE  $TEXT$
196         .SUBSPA $CODE$
197 ThreadRoot
198         .PROC
199          .CALLINFO CALLER,FRAME=0
200         .ENTRY
201
202         .CALL
203         ble  0(%r6)             ;call StartupPC
204         or   %r31, 0, %rp       ;put return address in proper register
205         or   %r4, 0, %arg0      ;load InitialArg
206         .CALL   ;in=26
207         ble  0(%r3)             ;call InitialPC
208         or   %r31, 0, %rp       ;put return address in proper register
209         .CALL
210         ble  0(%r5)             ;call WhenDonePC
```

```
211         .EXIT
212         or   %r31, 0, %rp       ;shouldn't really matter - doesn't return
213
214         .PROCEND
215
216
217 SWITCH
218         .PROC
219         .CALLINFO CALLER,FRAME=0
220         .ENTRY
221
222     ; save process state of oldThread
223         stw  %sp, SP(%arg0)     ;save stack pointer
224         stw  %r3, S0(%arg0)     ;save callee-save registers
225         stw  %r4, S1(%arg0)
226         stw  %r5, S2(%arg0)
227         stw  %r6, S3(%arg0)
228         stw  %r7, S4(%arg0)
229         stw  %r8, S5(%arg0)
230         stw  %r9, S6(%arg0)
231         stw  %r10, S7(%arg0)
232         stw  %r11, S8(%arg0)
233         stw  %r12, S9(%arg0)
234         stw  %r13, S10(%arg0)
235         stw  %r14, S11(%arg0)
236         stw  %r15, S12(%arg0)
237         stw  %r16, S13(%arg0)
238         stw  %r17, S14(%arg0)
239         stw  %r18, S15(%arg0)
240         stw  %rp, PC(%arg0)     ;save program counter
241
242     ; restore process state of nextThread
243         ldw  SP(%arg1), %sp     ;restore stack pointer
244         ldw  S0(%arg1), %r3     ;restore callee-save registers
245         ldw  S1(%arg1), %r4
246         ldw  S2(%arg1), %r5
247         ldw  S3(%arg1), %r6
248         ldw  S4(%arg1), %r7
249         ldw  S5(%arg1), %r8
250         ldw  S6(%arg1), %r9
251         ldw  S7(%arg1), %r10
252         ldw  S8(%arg1), %r11
253         ldw  S9(%arg1), %r12
254         ldw  S10(%arg1), %r13
255         ldw  S11(%arg1), %r14
256         ldw  S12(%arg1), %r15
257         ldw  S13(%arg1), %r16
258         ldw  S14(%arg1), %r17
259         ldw  PC(%arg1), %rp     ;save program counter
260         bv   0(%rp)
261         .EXIT
262         ldw  S15(%arg1), %r18
263
264         .PROCEND
265
266         .EXPORT SWITCH,ENTRY,PRIV_LEV=3,RTNVAL=GR
267         .EXPORT ThreadRoot,ENTRY,PRIV_LEV=3,RTNVAL=GR
268
```

```
269 #endif
270
271 #ifdef HOST_i386
272
273         .text
274         .align  2
275
276         .globl  _ThreadRoot
277
278 /* void ThreadRoot( void )
279 **
280 ** expects the following registers to be initialized:
281 **      eax     points to startup function (interrupt enable)
282 **      edx     contains inital argument to thread function
283 **      esi     points to thread function
284 **      edi     point to Thread::Finish()
285 */
286 _ThreadRoot:
287         pushl   %ebp
288         movl    %esp,%ebp
289         pushl   InitialArg
290         call    StartupPC
291         call    InitialPC
292         call    WhenDonePC
293
294         # NOT REACHED
295         movl    %ebp,%esp
296         popl    %ebp
297         ret
298
299
300
301 /* void SWITCH( thread *t1, thread *t2 )
302 **
303 ** on entry, stack looks like this:
304 **      8(esp)  ->              thread *t2
305 **      4(esp)  ->              thread *t1
306 **       (esp)  ->              return address
307 **
308 ** we push the current eax on the stack so that we can use it as
309 ** a pointer to t1, this decrements esp by 4, so when we use it
310 ** to reference stuff on the stack, we add 4 to the offset.
311 */
312         .comm   _eax_save,4
313
314         .globl  _SWITCH
315 _SWITCH:
316         movl    %eax,_eax_save          # save the value of eax
317         movl    4(%esp),%eax            # move pointer to t1 into eax
318         movl    %ebx,_EBX(%eax)         # save registers
319         movl    %ecx,_ECX(%eax)
320         movl    %edx,_EDX(%eax)
321         movl    %esi,_ESI(%eax)
322         movl    %edi,_EDI(%eax)
323         movl    %ebp,_EBP(%eax)
324         movl    %esp,_ESP(%eax)         # save stack pointer
325         movl    _eax_save,%ebx          # get the saved value of eax
326         movl    %ebx,_EAX(%eax)         # store it
```

```
327         movl    0(%esp),%ebx            # get return address from stack into ebx
328         movl    %ebx,_PC(%eax)          # save it into the pc storage
329
330         movl    8(%esp),%eax            # move pointer to t2 into eax
331
332         movl    _EAX(%eax),%ebx         # get new value for eax into ebx
333         movl    %ebx,_eax_save          # save it
334         movl    _EBX(%eax),%ebx         # retore old registers
335         movl    _ECX(%eax),%ecx
336         movl    _EDX(%eax),%edx
337         movl    _ESI(%eax),%esi
338         movl    _EDI(%eax),%edi
339         movl    _EBP(%eax),%ebp
340         movl    _ESP(%eax),%esp         # restore stack pointer
341         movl    _PC(%eax),%eax          # restore return address into eax
342         movl    %eax,4(%esp)            # copy over the ret address on the stack
343         movl    _eax_save,%eax
344
345         ret
346
347 #endif // HOST_i386
348
349 // Roberto Rossi (roberto@csr.unibo.it) - 1994
350 #ifdef HOST_ALPHA
351
352         .set noreorder
353         .set volatile
354         .set noat
355
356         .text
357         .align 3
358
359         .globl ThreadRoot
360         .ent ThreadRoot
361 ThreadRoot:
362         ldgp gp,0(pv)
363         .frame sp,0,ra,0
364         .prologue 1
365
366         mov StartupPC,pv
367         jsr ra,(pv),0           # call startup procedure
368         ldgp gp,0(ra)
369         mov InitialArg,a0
370         mov InitialPC,pv
371         jsr ra,(pv),0           # call main procedure
372         ldgp gp,0(ra)
373         mov WhenDonePC,pv
374         jsr ra,(pv),0           # when we are done, call the cleanup procedure
375         ldgp gp,0(ra)
376
377         # NEVER REACHED
378         ret zero,(ra),1
379         .end ThreadRoot
380
381         # a0 -- pointer to old thread
382         # a1 -- pointer to new thread
383         .globl SWITCH
384         .ent SWITCH
```

```
385 SWITCH:
386         ldgp gp,0(pv)
387         .frame sp,0,ra,0
388         .prologue 1
389
390         stq sp,SP(a0)           # save new stack pointer
391         stq s0,S0(a0)           # save all the callee-save registers
392         stq s1,S1(a0)
393         stq s2,S2(a0)
394         stq s3,S3(a0)
395         stq s4,S4(a0)
396         stq s5,S5(a0)
397          stq fp,FP(a0)           # save frame pointer
398         stq gp,GP(a0)           # save global pointer
399         stq ra,PC(a0)           # save return address
400
401         ldq sp,SP(a1)           # load the new stack pointer
402         ldq s0,S0(a1)           # load the callee-save registers
403         ldq s1,S1(a1)
404         ldq s2,S2(a1)
405         ldq s3,S3(a1)
406         ldq s4,S4(a1)
407         ldq s5,S5(a1)
408         ldq fp,FP(a1)           # load frame pointer
409         ldq gp,GP(a1)           # load global pointer
410         ldq ra,PC(a1)           # load return address
411
412         jmp (ra)                        # execute context-switch
413         ldgp gp,0(ra)
414         ret zero,(ra),1
415         .end SWITCH
416
417 #endif // HOST_ALPHA
```

1.9 synch.h

```
 1 // synch.h
 2 //      Data structures for synchronizing threads.
 3 //
 4 //      Three kinds of synchronization are defined here: semaphores,
 5 //      locks, and condition variables.  The implementation for
 6 //      semaphores is given; for the latter two, only the procedure
 7 //      interface is given -- they are to be implemented as part of
 8 //      the first assignment.
 9 //
10 //      Note that all the synchronization objects take a "name" as
11 //      part of the initialization.  This is solely for debugging purposes.
12 //
13 // Copyright (c) 1992-1993 The Regents of the University of California.
14 // All rights reserved.  See copyright.h for copyright notice and limitation
15 // synch.h -- synchronization primitives.
16
17 #ifndef SYNCH_H
18 #define SYNCH_H
19
20 #include "copyright.h"
21 #include "thread.h"
```

```
22 #include "list.h"
23
24
25 // The following class defines a "semaphore" whose value is a non-negative
26 // integer.  The semaphore has only two operations P() and V():
27 //
28 //      P() -- waits until value > 0, then decrement
29 //
30 //      V() -- increment, waking up a thread waiting in P() if necessary
31 //
32 // Note that the interface does *not* allow a thread to read the value of
33 // the semaphore directly -- even if you did read the value, the
34 // only thing you would know is what the value used to be.  You don't
35 // know what the value is now, because by the time you get the value
36 // into a register, a context switch might have occurred,
37 // and some other thread might have called P or V, so the true value might
38 // now be different.
39
40 class Semaphore {
41   public:
42     Semaphore(char* debugName, int initialValue);       // set initial value
43     ~Semaphore();                                       // de-allocate semaphore
44     char* getName() { return name;}                     // debugging assist
45
46     void P();    // these are the only operations on a semaphore
47     void V();    // they are both *atomic*
48
49   private:
50     char* name;        // useful for debugging
51     int value;         // semaphore value, always >= 0
52     List *queue;       // threads waiting in P() for the value to be > 0
53 };
54
55 // The following class defines a "lock".  A lock can be BUSY or FREE.
56 // There are only two operations allowed on a lock:
57 //
58 //      Acquire -- wait until the lock is FREE, then set it to BUSY
59 //
60 //      Release -- set lock to be FREE, waking up a thread waiting
61 //              in Acquire if necessary
62 //
63 // In addition, by convention, only the thread that acquired the lock
64 // may release it.  As with semaphores, you can't read the lock value
65 // (because the value might change immediately after you read it).
66
67 class Lock {
68   public:
69     Lock(char* debugName);              // initialize lock to be FREE
70     ~Lock();                            // deallocate lock
71     char* getName() { return name; }    // debugging assist
72
73     void Acquire(); // these are the only operations on a lock
74     void Release(); // they are both *atomic*
75
76     bool isHeldByCurrentThread();       // true if the current thread
77                                         // holds this lock.  Useful for
78                                         // checking in Release, and in
79                                         // Condition variable ops below.
```

```
 80
 81   private:
 82     char* name;                         // for debugging
 83     Thread *owner;                      // remember who acquired the lock
 84     Semaphore *lock;                    // use semaphore for the actual lock
 85 };
 86
 87 // The following class defines a "condition variable".  A condition
 88 // variable does not have a value, but threads may be queued, waiting
 89 // on the variable.  These are only operations on a condition variable:
 90 //
 91 //      Wait() -- release the lock, relinquish the CPU until signaled,
 92 //              then re-acquire the lock
 93 //
 94 //      Signal() -- wake up a thread, if there are any waiting on
 95 //              the condition
 96 //
 97 //      Broadcast() -- wake up all threads waiting on the condition
 98 //
 99 // All operations on a condition variable must be made while
100 // the current thread has acquired a lock.  Indeed, all accesses
101 // to a given condition variable must be protected by the same lock.
102 // In other words, mutual exclusion must be enforced among threads calling
103 // the condition variable operations.
104 //
105 // In Nachos, condition variables are assumed to obey *Mesa*-style
106 // semantics.  When a Signal or Broadcast wakes up another thread,
107 // it simply puts the thread on the ready list, and it is the responsibility
108 // of the woken thread to re-acquire the lock (this re-acquire is
109 // taken care of within Wait()).  By contrast, some define condition
110 // variables according to *Hoare*-style semantics -- where the signalling
111 // thread gives up control over the lock and the CPU to the woken thread,
112 // which runs immediately and gives back control over the lock to the
113 // signaller when the woken thread leaves the critical section.
114 //
115 // The consequence of using Mesa-style semantics is that some other thread
116 // can acquire the lock, and change data structures, before the woken
117 // thread gets a chance to run.
118
119 class Condition {
120   public:
121     Condition(char* debugName);         // initialize condition to
122                                         // "no one waiting"
123     ~Condition();                       // deallocate the condition
124     char* getName() { return (name); }
125
126     void Wait(Lock *conditionLock);     // these are the 3 operations on
127                                         // condition variables; releasing the
128                                         // lock and going to sleep are
129                                         // *atomic* in Wait()
130     void Signal(Lock *conditionLock);   // conditionLock must be held by
131     void Broadcast(Lock *conditionLock);// the currentThread for all of
132                                         // these operations
133
134   private:
135     char* name;
136     List* queue;  // threads waiting on the condition
137     Lock* lock;   // debugging aid:  used to check correctness of
```

```
138                     // arguments to Wait, Signal and Broacast
139 };
140 #endif // SYNCH_H
```

1.10 synch.cc

```
  1 // synch.cc
  2 //      Routines for synchronizing threads.  Three kinds of
  3 //      synchronization routines are defined here: semaphores, locks
  4 //      and condition variables (the implementation of the last two
  5 //      are left to the reader).
  6 //
  7 // Any implementation of a synchronization routine needs some
  8 // primitive atomic operation.  We assume Nachos is running on
  9 // a uniprocessor, and thus atomicity can be provided by
 10 // turning off interrupts.  While interrupts are disabled, no
 11 // context switch can occur, and thus the current thread is guaranteed
 12 // to hold the CPU throughout, until interrupts are reenabled.
 13 //
 14 // Because some of these routines might be called with interrupts
 15 // already disabled (Semaphore::V for one), instead of turning
 16 // on interrupts at the end of the atomic operation, we always simply
 17 // re-set the interrupt state back to its original value (whether
 18 // that be disabled or enabled).
 19 //
 20 // Copyright (c) 1992-1993 The Regents of the University of California.
 21 // All rights reserved.  See copyright.h for copyright notice and limitation
 22 // of liability and disclaimer of warranty provisions.
 23
 24 #include "copyright.h"
 25 #include "synch.h"
 26 #include "system.h"
 27
 28 //----------------------------------------------------------------------
 29 // Semaphore::Semaphore
 30 //      Initialize a semaphore, so that it can be used for synchronization.
 31 //
 32 //      "debugName" is an arbitrary name, useful for debugging.
 33 //      "initialValue" is the initial value of the semaphore.
 34 //----------------------------------------------------------------------
 35
 36 Semaphore::Semaphore(char* debugName, int initialValue)
 37 {
 38     name = debugName;
 39     value = initialValue;
 40     queue = new List;
 41 }
 42
 43 //----------------------------------------------------------------------
 44 // Semaphore::~Semaphore
 45 //      De-allocate semaphore, when no longer needed.  Assume no one
 46 //      is still waiting on the semaphore!
 47 //----------------------------------------------------------------------
 48
 49 Semaphore::~Semaphore()
 50 {
 51     delete queue;
```

```
52 }
53
54 //----------------------------------------------------------------------
55 // Semaphore::P
56 //      Wait until semaphore value > 0, then decrement.  Checking the
57 //      value and decrementing must be done atomically, so we
58 //      need to disable interrupts before checking the value.
59 //
60 //      Note that Thread::Sleep assumes that interrupts are disabled
61 //      when it is called.
62 //----------------------------------------------------------------------
63
64 void
65 Semaphore::P()
66 {
67     IntStatus oldLevel = interrupt->SetLevel(IntOff);    // disable interrupts
68
69     while (value == 0) {                        // semaphore not available
70         queue->Append((void *)currentThread);   // so go to sleep
71         currentThread->Sleep();
72     }
73     value--;                                    // semaphore available,
74                                                 // consume its value
75
76     (void) interrupt->SetLevel(oldLevel);       // re-enable interrupts
77 }
78
79 //----------------------------------------------------------------------
80 // Semaphore::V
81 //      Increment semaphore value, waking up a waiter if necessary.
82 //      As with P(), this operation must be atomic, so we need to disable
83 //      interrupts.  Scheduler::ReadyToRun() assumes that threads
84 //      are disabled when it is called.
85 //----------------------------------------------------------------------
86
87 void
88 Semaphore::V()
89 {
90     Thread *thread;
91     IntStatus oldLevel = interrupt->SetLevel(IntOff);
92
93     thread = (Thread *)queue->Remove();
94     if (thread != NULL)    // make thread ready, consuming the V immediately
95         scheduler->ReadyToRun(thread);
96     value++;
97     (void) interrupt->SetLevel(oldLevel);
98 }
99
100
101 //----------------------------------------------------------------------
102 // Lock::Lock
103 //      Initialize a lock, so that it can be used for synchronization.
104 //
105 //      "debugName" is an arbitrary name, useful for debugging.
106 //----------------------------------------------------------------------
107
108
109 Lock::Lock(char* debugName)
```

```
110 {
111     name = debugName;
112     owner = NULL;
113     lock = new Semaphore(name,1);
114 }
115
116
117 //----------------------------------------------------------------------
118 // Lock::~Lock
119 //      De-allocate lock, when no longer needed.  As with semaphore,
120 //      assume no one is still waiting on the lock.
121 //----------------------------------------------------------------------
122 Lock::~Lock()
123 {
124     delete lock;
125 }
126
127 //----------------------------------------------------------------------
128 // Lock::Acquire
129 //      Use a binary semaphore to implement the lock.  Record which
130 //      thread acquired the lock in order to assure that only the
131 //      same thread releases it.
132 //----------------------------------------------------------------------
133 void Lock::Acquire()
134 {
135     IntStatus oldLevel = interrupt->SetLevel(IntOff);  // disable interrupts
136
137     lock->P();                              // procure the semaphore
138     owner = currentThread;                  // record the new owner of the lock
139     (void) interrupt->SetLevel(oldLevel); // re-enable interrupts
140 }
141
142 //----------------------------------------------------------------------
143 // Lock::Release
144 //      Set the lock to be free (i.e. vanquish the semaphore).  Check
145 //      that the currentThread is allowed to release this lock.
146 //----------------------------------------------------------------------
147 void Lock::Release()
148 {
149     IntStatus oldLevel = interrupt->SetLevel(IntOff);  // disable interrupts
150
151     // Ensure: a) lock is BUSY  b) this thread is the same one that acquired it.
152     ASSERT(currentThread == owner);
153     owner = NULL;                           // clear the owner
154     lock->V();                              // vanquish the semaphore
155     (void) interrupt->SetLevel(oldLevel);
156 }
157
158
159 //----------------------------------------------------------------------
160 // Lock::isHeldByCurrentThread
161 //----------------------------------------------------------------------
162 bool Lock::isHeldByCurrentThread()
163 {
164     bool result;
165     IntStatus oldLevel = interrupt->SetLevel(IntOff);
166
167     result = currentThread == owner;
```

```
168     (void) interrupt->SetLevel(oldLevel);
169     return(result);
170 }
171
172 //----------------------------------------------------------------------
173 // Condition::Condition
174 //      Initialize a condition variable, so that it can be used for
175 //      synchronization.
176 //
177 //      "debugName" is an arbitrary name, useful for debugging.
178 //----------------------------------------------------------------------
179 Condition::Condition(char* debugName)
180 {
181     name = debugName;
182     queue = new List;
183     lock = NULL;
184 }
185
186 //----------------------------------------------------------------------
187 // Condition::~Condition
188 //      De-allocate a condition variable, when no longer needed.  As
189 //      with semaphore, assume no one is still waiting on the condition.
190 //----------------------------------------------------------------------
191
192 Condition::~Condition()
193 {
194     delete queue;
195 }
196
197 //----------------------------------------------------------------------
198 // Condition::Wait
199 //
200 //      Release the lock, relinquish the CPU until signaled, then
201 //      re-acquire the lock.
202 //
203 //      Pre-conditions:  currentThread is holding the lock; threads in
204 //      the queue are waiting on the same lock.
205 //----------------------------------------------------------------------
206 void Condition::Wait(Lock* conditionLock)
207 {
208     IntStatus oldLevel = interrupt->SetLevel(IntOff);
209
210     ASSERT(conditionLock->isHeldByCurrentThread());  // check pre-condition
211     if(queue->IsEmpty()) {
212         lock = conditionLock;  // helps to enforce pre-condition
213     }
214     ASSERT(lock == conditionLock); // another pre-condition
215     queue->Append(currentThread);  // add this thread to the waiting list
216     conditionLock->Release();      // release the lock
217     currentThread->Sleep();        // goto sleep
218     conditionLock->Acquire();      // awaken: re-acquire the lock
219     (void) interrupt->SetLevel(oldLevel);
220 }
221
222 //----------------------------------------------------------------------
223 // Condition::Signal
224 //      Wake up a thread, if there are any waiting on the condition.
225 //
```

```
226 //      Pre-conditions:  currentThread is holding the lock; threads in
227 //      the queue are waiting on the same lock.
228 //----------------------------------------------------------------------
229 void Condition::Signal(Lock* conditionLock)
230 {
231     Thread *nextThread;
232     IntStatus oldLevel = interrupt->SetLevel(IntOff);
233
234     ASSERT(conditionLock->isHeldByCurrentThread());
235     if(!queue->IsEmpty()) {
236         ASSERT(lock == conditionLock);
237         nextThread = (Thread *)queue->Remove();
238         scheduler->ReadyToRun(nextThread);      // wake up the thread
239     }
240     (void) interrupt->SetLevel(oldLevel);
241 }
242
243 //----------------------------------------------------------------------
244 // Condition::Broadcast
245 //      Wake up all threads waiting on the condition.
246 //
247 //      Pre-conditions:  currentThread is holding the lock; threads in
248 //      the queue are waiting on the same lock.
249 //----------------------------------------------------------------------
250 void Condition::Broadcast(Lock* conditionLock)
251 {
252     Thread *nextThread;
253     IntStatus oldLevel = interrupt->SetLevel(IntOff);
254
255     ASSERT(conditionLock->isHeldByCurrentThread());
256     if(!queue->IsEmpty()) {
257         ASSERT(lock == conditionLock);
258         while(nextThread = (Thread *)queue->Remove()) {
259             scheduler->ReadyToRun(nextThread);  // wake up the thread
260         }
261     }
262     (void) interrupt->SetLevel(oldLevel);
263 }
```

1.11 synchlist.h

```
 1 // synchlist.h
 2 //      Data structures for synchronized access to a list.
 3 //
 4 //      Implemented by surrounding the List abstraction
 5 //      with synchronization routines.
 6 //
 7 // Copyright (c) 1992-1993 The Regents of the University of California.
 8 // All rights reserved.  See copyright.h for copyright notice and limitation
 9 // of liability and disclaimer of warranty provisions.
10
11 #ifndef SYNCHLIST_H
12 #define SYNCHLIST_H
13
14 #include "copyright.h"
15 #include "list.h"
16 #include "synch.h"
```

```
17
18 // The following class defines a "synchronized list" -- a list for which:
19 // these constraints hold:
20 //      1. Threads trying to remove an item from a list will
21 //      wait until the list has an element on it.
22 //      2. One thread at a time can access list data structures
23
24 class SynchList {
25   public:
26     SynchList();               // initialize a synchronized list
27     ~SynchList();              // de-allocate a synchronized list
28
29     void Append(void *item);   // append item to the end of the list,
30                                // and wake up any thread waiting in remove
31     void *Remove();            // remove the first item from the front of
32                                // the list, waiting if the list is empty
33                                // apply function to every item in the list
34     void Mapcar(VoidFunctionPtr func);
35
36   private:
37     List *list;                // the unsynchronized list
38     Lock *lock;                // enforce mutual exclusive access to the list
39     Condition *listEmpty;      // wait in Remove if the list is empty
40 };
41
42 #endif // SYNCHLIST_H
```

1.12 synchlist.cc

```
 1 // synchlist.cc
 2 //      Routines for synchronized access to a list.
 3 //
 4 //      Implemented by surrounding the List abstraction
 5 //      with synchronization routines.
 6 //
 7 //      Implemented in "monitor"-style -- surround each procedure with a
 8 //      lock acquire and release pair, using condition signal and wait for
 9 //      synchronization.
10 //
11 // Copyright (c) 1992-1993 The Regents of the University of California.
12 // All rights reserved.  See copyright.h for copyright notice and limitation
13 // of liability and disclaimer of warranty provisions.
14
15 #include "copyright.h"
16 #include "synchlist.h"
17
18 //----------------------------------------------------------------------
19 // SynchList::SynchList
20 //      Allocate and initialize the data structures needed for a
21 //      synchronized list, empty to start with.
22 //      Elements can now be added to the list.
23 //----------------------------------------------------------------------
24
25 SynchList::SynchList()
26 {
27     list = new List();
28     lock = new Lock("list lock");
```

```
29     listEmpty = new Condition("list empty cond");
30 }
31
32 //----------------------------------------------------------------------
33 // SynchList::~SynchList
34 //      De-allocate the data structures created for synchronizing a list.
35 //----------------------------------------------------------------------
36
37 SynchList::~SynchList()
38 {
39     delete list;
40     delete lock;
41     delete listEmpty;
42 }
43
44 //----------------------------------------------------------------------
45 // SynchList::Append
46 //      Append an "item" to the end of the list.  Wake up anyone
47 //      waiting for an element to be appended.
48 //
49 //      "item" is the thing to put on the list, it can be a pointer to
50 //              anything.
51 //----------------------------------------------------------------------
52
53 void
54 SynchList::Append(void *item)
55 {
56     lock->Acquire();            // enforce mutual exclusive access to the list
57     list->Append(item);
58     listEmpty->Signal(lock);    // wake up a waiter, if any
59     lock->Release();
60 }
61
62 //----------------------------------------------------------------------
63 // SynchList::Remove
64 //      Remove an "item" from the beginning of the list.  Wait if
65 //      the list is empty.
66 // Returns:
67 //       The removed item.
68 //----------------------------------------------------------------------
69
70 void *
71 SynchList::Remove()
72 {
73     void *item;
74
75     lock->Acquire();                    // enforce mutual exclusion
76     while (list->IsEmpty())
77         listEmpty->Wait(lock);          // wait until list isn't empty
78     item = list->Remove();
79     ASSERT(item != NULL);
80     lock->Release();
81     return item;
82 }
83
84 //----------------------------------------------------------------------
85 // SynchList::Mapcar
86 //      Apply function to every item on the list.  Obey mutual exclusion
```

```
87 //      constraints.
88 //
89 //      "func" is the procedure to be applied.
90 //----------------------------------------------------------------------
91
92 void
93 SynchList::Mapcar(VoidFunctionPtr func)
94 {
95     lock->Acquire();
96     list->Mapcar(func);
97     lock->Release();
98 }
```

1.13 system.h

```
 1 // system.h
 2 //      All global variables used in Nachos are defined here.
 3 //
 4 // Copyright (c) 1992-1993 The Regents of the University of California.
 5 // All rights reserved.  See copyright.h for copyright notice and limitation
 6 // of liability and disclaimer of warranty provisions.
 7
 8 #ifndef SYSTEM_H
 9 #define SYSTEM_H
10
11 #include "copyright.h"
12 #include "utility.h"
13 #include "thread.h"
14 #include "scheduler.h"
15 #include "interrupt.h"
16 #include "stats.h"
17 #include "timer.h"
18
19 // Initialization and cleanup routines
20 extern void Initialize(int argc, char **argv);  // Initialization,
21                                                 // called before anything else
22 extern void Cleanup();                          // Cleanup, called when
23                                                 // Nachos is done.
24
25 extern Thread *currentThread;                   // the thread holding the CPU
26 extern Thread *threadToBeDestroyed;             // the thread that just finished
27 extern Scheduler *scheduler;                    // the ready list
28 extern Interrupt *interrupt;                    // interrupt status
29 extern Statistics *stats;                       // performance metrics
30 extern Timer *timer;                            // the hardware alarm clock
31
32 #ifdef USER_PROGRAM
33 #include "machine.h"
34 extern Machine* machine;        // user program memory and registers
35 #endif
36
37 #ifdef FILESYS_NEEDED           // FILESYS or FILESYS_STUB
38 #include "filesys.h"
39 extern FileSystem  *fileSystem;
40 #endif
41
42 #ifdef FILESYS
```

```
43 #include "synchdisk.h"
44 extern SynchDisk   *synchDisk;
45 #endif
46
47 #ifdef NETWORK
48 #include "post.h"
49 extern PostOffice* postOffice;
50 #endif
51
52 #endif // SYSTEM_H
```

1.14 system.cc

```
 1 // system.cc
 2 //      Nachos initialization and cleanup routines.
 3 //
 4 // Copyright (c) 1992-1993 The Regents of the University of California.
 5 // All rights reserved.  See copyright.h for copyright notice and limitation
 6 // of liability and disclaimer of warranty provisions.
 7
 8 #include "copyright.h"
 9 #include "system.h"
10
11 // This defines *all* of the global data structures used by Nachos.
12 // These are all initialized and de-allocated by this file.
13
14 Thread *currentThread;                  // the thread we are running now
15 Thread *threadToBeDestroyed;            // the thread that just finished
16 Scheduler *scheduler;                   // the ready list
17 Interrupt *interrupt;                   // interrupt status
18 Statistics *stats;                      // performance metrics
19 Timer *timer;                           // the hardware timer device,
20                                         // for invoking context switches
21
22 #ifdef FILESYS_NEEDED
23 FileSystem  *fileSystem;
24 #endif
25
26 #ifdef FILESYS
27 SynchDisk   *synchDisk;
28 #endif
29
30 #ifdef USER_PROGRAM     // requires either FILESYS or FILESYS_STUB
31 Machine *machine;       // user program memory and registers
32 #endif
33
34 #ifdef NETWORK
35 PostOffice *postOffice;
36 #endif
37
38
39 // External definition, to allow us to take a pointer to this function
40 extern void Cleanup();
41
42
43 //----------------------------------------------------------------------
44 // TimerInterruptHandler
```

```
45 //      Interrupt handler for the timer device.  The timer device is
46 //      set up to interrupt the CPU periodically (once every TimerTicks).
47 //      This routine is called each time there is a timer interrupt,
48 //      with interrupts disabled.
49 //
50 //      Note that instead of calling Yield() directly (which would
51 //      suspend the interrupt handler, not the interrupted thread
52 //      which is what we wanted to context switch), we set a flag
53 //      so that once the interrupt handler is done, it will appear as
54 //      if the interrupted thread called Yield at the point it is
55 //      was interrupted.
56 //
57 //      "dummy" is because every interrupt handler takes one argument,
58 //              whether it needs it or not.
59 //----------------------------------------------------------------------
60 static void
61 TimerInterruptHandler(_int dummy)
62 {
63     if (interrupt->getStatus() != IdleMode)
64         interrupt->YieldOnReturn();
65 }
66
67 //----------------------------------------------------------------------
68 // Initialize
69 //      Initialize Nachos global data structures.  Interpret command
70 //      line arguments in order to determine flags for the initialization.
71 //
72 //      "argc" is the number of command line arguments (including the name
73 //              of the command) -- ex: "nachos -d +" -> argc = 3
74 //      "argv" is an array of strings, one for each command line argument
75 //              ex: "nachos -d +" -> argv = {"nachos", "-d", "+"}
76 //----------------------------------------------------------------------
77 void
78 Initialize(int argc, char **argv)
79 {
80     int argCount;
81     char* debugArgs = "";
82     bool randomYield = FALSE;
83
84 #ifdef USER_PROGRAM
85     bool debugUserProg = FALSE; // single step user program
86 #endif
87 #ifdef FILESYS_NEEDED
88     bool format = FALSE;        // format disk
89 #endif
90 #ifdef NETWORK
91     double rely = 1;            // network reliability
92     double order = 1;           // network orderability
93     int netname = 0;            // UNIX socket name
94 #endif
95
96     for (argc--, argv++; argc > 0; argc -= argCount, argv += argCount) {
97         argCount = 1;
98         if (!strcmp(*argv, "-d")) {
99             if (argc == 1)
100                debugArgs = "+";        // turn on all debug flags
101            else {
102                debugArgs = *(argv + 1);
```

```
103                 argCount = 2;
104             }
105         } else if (!strcmp(*argv, "-rs")) {
106             ASSERT(argc > 1);
107             RandomInit(atoi(*(argv + 1)));      // initialize pseudo-random
108                                                 // number generator
109             randomYield = TRUE;
110             argCount = 2;
111         }
112 #ifdef USER_PROGRAM
113         if (!strcmp(*argv, "-s"))
114             debugUserProg = TRUE;
115 #endif
116 #ifdef FILESYS_NEEDED
117         if (!strcmp(*argv, "-f"))
118             format = TRUE;
119 #endif
120 #ifdef NETWORK
121         if (!strcmp(*argv, "-n")) {
122             ASSERT(argc > 1);
123             rely = atof(*(argv + 1));
124             argCount = 2;
125         } else if (!strcmp(*argv, "-e")) {
126             ASSERT(argc > 1);
127             order = atof(*(argv + 1));
128             argCount = 2;
129         } else if (!strcmp(*argv, "-m")) {
130             ASSERT(argc > 1);
131             netname = atoi(*(argv + 1));
132             argCount = 2;
133          }
134 #endif
135     }
136
137     DebugInit(debugArgs);                       // initialize DEBUG messages
138     stats = new Statistics();                   // collect statistics
139     interrupt = new Interrupt;                  // start up interrupt handling
140     scheduler = new Scheduler();                // initialize the ready queue
141     // if (randomYield)                         // start the timer (if needed)
142         timer = new Timer(TimerInterruptHandler, 0, randomYield);
143
144     threadToBeDestroyed = NULL;
145
146     // We didn't explicitly allocate the current thread we are running in.
147     // But if it ever tries to give up the CPU, we better have a Thread
148     // object to save its state.
149     currentThread = new Thread("main");
150     currentThread->setStatus(RUNNING);
151
152     interrupt->Enable();
153     CallOnUserAbort(Cleanup);                   // if user hits ctl-C
154
155 #ifdef USER_PROGRAM
156     machine = new Machine(debugUserProg);       // this must come first
157 #endif
158
159 #ifdef FILESYS
160     synchDisk = new SynchDisk("DISK");
```

```
161 #endif
162
163 #ifdef FILESYS_NEEDED
164     fileSystem = new FileSystem(format);
165 #endif
166
167 #ifdef NETWORK
168     postOffice = new PostOffice(netname, rely, order, 10);
169 #endif
170 }
171
172 //----------------------------------------------------------------------
173 // Cleanup
174 //      Nachos is halting.  De-allocate global data structures.
175 //----------------------------------------------------------------------
176 void
177 Cleanup()
178 {
179     printf("\nCleaning up...\n");
180 #ifdef NETWORK
181     delete postOffice;
182 #endif
183
184 #ifdef USER_PROGRAM
185     delete machine;
186 #endif
187
188 #ifdef FILESYS_NEEDED
189     delete fileSystem;
190 #endif
191
192 #ifdef FILESYS
193     delete synchDisk;
194 #endif
195
196     delete timer;
197     delete scheduler;
198     delete interrupt;
199
200     Exit(0);
201 }
202
```

1.15 thread.h

```
 1 // thread.h
 2 //      Data structures for managing threads.  A thread represents
 3 //      sequential execution of code within a program.
 4 //      So the state of a thread includes the program counter,
 5 //      the processor registers, and the execution stack.
 6 //
 7 //      Note that because we allocate a fixed size stack for each
 8 //      thread, it is possible to overflow the stack -- for instance,
 9 //      by recursing to too deep a level.  The most common reason
10 //      for this occuring is allocating large data structures
11 //      on the stack.  For instance, this will cause problems:
12 //
```

```
13 //              void foo() { int buf[1000]; ...}
14 //
15 //      Instead, you should allocate all data structures dynamically:
16 //
17 //              void foo() { int *buf = new int[1000]; ...}
18 //
19 //
20 //      Bad things happen if you overflow the stack, and in the worst
21 //      case, the problem may not be caught explicitly.  Instead,
22 //      the only symptom may be bizarre segmentation faults.  (Of course,
23 //      other problems can cause seg faults, so that isn't a sure sign
24 //      that your thread stacks are too small.)
25 //
26 //      One thing to try if you find yourself with seg faults is to
27 //      increase the size of thread stack -- ThreadStackSize.
28 //
29 //      In this interface, forking a thread takes two steps.
30 //      We must first allocate a data structure for it: "t = new Thread".
31 //      Only then can we do the fork: "t->fork(f, arg)".
32 //
33 // Copyright (c) 1992-1993 The Regents of the University of California.
34 // All rights reserved.  See copyright.h for copyright notice and limitation
35 // of liability and disclaimer of warranty provisions.
36
37 #ifndef THREAD_H
38 #define THREAD_H
39
40 #include "copyright.h"
41 #include "utility.h"
42
43 #ifdef USER_PROGRAM
44 #include "machine.h"
45 #include "addrspace.h"
46 #endif
47
48
49 // CPU register state to be saved on context switch.
50 // The SPARC and MIPS only need 10 registers, but the Snake needs 18.
51 // For simplicity, this is just the max over all architectures.
52 #define MachineStateSize 18
53
54
55 // Size of the thread's private execution stack.
56 // WATCH OUT IF THIS ISN'T BIG ENOUGH!!!!!
57 #define StackSize       (sizeof(_int) * 1024)   // in words
58
59
60 // Thread state
61 enum ThreadStatus { JUST_CREATED, RUNNING, READY, BLOCKED };
62
63 // external function, dummy routine whose sole job is to call Thread::Print
64 extern void ThreadPrint(_int arg);
65
66 // The following class defines a "thread control block" -- which
67 // represents a single thread of execution.
68 //
69 //  Every thread has:
70 //     an execution stack for activation records ("stackTop" and "stack")
```

```
71 //     space to save CPU registers while not running ("machineState")
72 //     a "status" (running/ready/blocked)
73 //
74 //  Some threads also belong to a user address space; threads
75 //  that only run in the kernel have a NULL address space.
76
77 class Thread {
78   private:
79     // NOTE: DO NOT CHANGE the order of these first two members.
80     // THEY MUST be in this position for SWITCH to work.
81     int* stackTop;                       // the current stack pointer
82     _int machineState[MachineStateSize];  // all registers except for stackTop
83
84   public:
85     Thread(char* debugName);            // initialize a Thread
86     ~Thread();                          // deallocate a Thread
87                                         // NOTE -- thread being deleted
88                                         // must not be running when delete
89                                         // is called
90
91     // basic thread operations
92
93     void Fork(VoidFunctionPtr func, _int arg);  // Make thread run (*func)(arg)
94     void Yield();                               // Relinquish the CPU if any
95                                                 // other thread is runnable
96     void Sleep();                               // Put the thread to sleep and
97                                                 // relinquish the processor
98     void Finish();                              // The thread is done executing
99
100    void CheckOverflow();                       // Check if thread has
101                                                // overflowed its stack
102    void setStatus(ThreadStatus st) { status = st; }
103    char* getName() { return (name); }
104    void Print() { printf("%s, ", name); }
105
106  private:
107    // some of the private data for this class is listed above
108
109    int* stack;                         // Bottom of the stack
110                                        // NULL if this is the main thread
111                                        // (If NULL, don't deallocate stack)
112    ThreadStatus status;                // ready, running or blocked
113    char* name;
114
115    void StackAllocate(VoidFunctionPtr func, _int arg);
116                                        // Allocate a stack for thread.
117                                        // Used internally by Fork()
118
119 #ifdef USER_PROGRAM
120 // A thread running a user program actually has *two* sets of CPU registers --
121 // one for its state while executing user code, one for its state
122 // while executing kernel code.
123
124    int userRegisters[NumTotalRegs];    // user-level CPU register state
125
126  public:
127    void SaveUserState();               // save user-level register state
128    void RestoreUserState();            // restore user-level register state
```

```
129
130     AddrSpace *space;                   // User code this thread is running.
131 #endif
132 };
133
134 // Magical machine-dependent routines, defined in switch.s
135
136 extern "C" {
137 // First frame on thread execution stack;
138 //      enable interrupts
139 //      call "func"
140 //      (when func returns, if ever) call ThreadFinish()
141 void ThreadRoot();
142
143 // Stop running oldThread and start running newThread
144 void SWITCH(Thread *oldThread, Thread *newThread);
145 }
146
147 #endif // THREAD_H
```

1.16 thread.cc

```
  1 // thread.cc
  2 //      Routines to manage threads.  There are four main operations:
  3 //
  4 //      Fork -- create a thread to run a procedure concurrently
  5 //              with the caller (this is done in two steps -- first
  6 //              allocate the Thread object, then call Fork on it)
  7 //      Finish -- called when the forked procedure finishes, to clean up
  8 //      Yield -- relinquish control over the CPU to another ready thread
  9 //      Sleep -- relinquish control over the CPU, but thread is now blocked.
 10 //              In other words, it will not run again, until explicitly
 11 //              put back on the ready queue.
 12 //
 13 // Copyright (c) 1992-1993 The Regents of the University of California.
 14 // All rights reserved.  See copyright.h for copyright notice and limitation
 15 // of liability and disclaimer of warranty provisions.
 16
 17 #include "copyright.h"
 18 #include "thread.h"
 19 #include "switch.h"
 20 #include "synch.h"
 21 #include "system.h"
 22
 23 #define STACK_FENCEPOST 0xdeadbeef      // this is put at the top of the
 24                                         // execution stack, for detecting
 25                                         // stack overflows
 26
 27 //----------------------------------------------------------------------
 28 // Thread::Thread
 29 //      Initialize a thread control block, so that we can then call
 30 //      Thread::Fork.
 31 //
 32 //      "threadName" is an arbitrary string, useful for debugging.
 33 //----------------------------------------------------------------------
 34
 35 Thread::Thread(char* threadName)
```

```
36 {
37     name = threadName;
38     stackTop = NULL;
39     stack = NULL;
40     status = JUST_CREATED;
41 #ifdef USER_PROGRAM
42     space = NULL;
43 #endif
44 }
45
46 //----------------------------------------------------------------------
47 // Thread::~Thread
48 //      De-allocate a thread.
49 //
50 //      NOTE: the current thread *cannot* delete itself directly,
51 //      since it is still running on the stack that we need to delete.
52 //
53 //      NOTE: if this is the main thread, we can't delete the stack
54 //      because we didn't allocate it -- we got it automatically
55 //      as part of starting up Nachos.
56 //----------------------------------------------------------------------
57
58 Thread::~Thread()
59 {
60     DEBUG('t', "Deleting thread \"%s\"\n", name);
61
62     ASSERT(this != currentThread);
63     if (stack != NULL)
64                 DeallocBoundedArray((char *) stack, StackSize * sizeof(_int));
65 }
66
67 //----------------------------------------------------------------------
68 // Thread::Fork
69 //      Invoke (*func)(arg), allowing caller and callee to execute
70 //      concurrently.
71 //
72 //      NOTE: although our definition allows only a single integer argument
73 //      to be passed to the procedure, it is possible to pass multiple
74 //      arguments by making them fields of a structure, and passing a pointer
75 //      to the structure as "arg".
76 //
77 //      Implemented as the following steps:
78 //              1. Allocate a stack
79 //              2. Initialize the stack so that a call to SWITCH will
80 //              cause it to run the procedure
81 //              3. Put the thread on the ready queue
82 //
83 //      "func" is the procedure to run concurrently.
84 //      "arg" is a single argument to be passed to the procedure.
85 //----------------------------------------------------------------------
86
87 void
88 Thread::Fork(VoidFunctionPtr func, _int arg)
89 {
90 #ifdef HOST_ALPHA
91     DEBUG('t', "Forking thread \"%s\" with func = 0x%lx, arg = %ld\n",
92           name, (long) func, arg);
93 #else
```

```
94     DEBUG('t', "Forking thread \"%s\" with func = 0x%x, arg = %d\n",
95           name, (int) func, arg);
96 #endif
97
98     StackAllocate(func, arg);
99
100    IntStatus oldLevel = interrupt->SetLevel(IntOff);
101    scheduler->ReadyToRun(this);        // ReadyToRun assumes that interrupts
102                                        // are disabled!
103    (void) interrupt->SetLevel(oldLevel);
104 }
105
106 //----------------------------------------------------------------------
107 // Thread::CheckOverflow
108 //      Check a thread's stack to see if it has overrun the space
109 //      that has been allocated for it.  If we had a smarter compiler,
110 //      we wouldn't need to worry about this, but we don't.
111 //
112 //      NOTE: Nachos will not catch all stack overflow conditions.
113 //      In other words, your program may still crash because of an overflow.
114 //
115 //      If you get bizarre results (such as seg faults where there is no code)
116 //      then you *may* need to increase the stack size.  You can avoid stack
117 //      overflows by not putting large data structures on the stack.
118 //      Don't do this: void foo() { int bigArray[10000]; ... }
119 //----------------------------------------------------------------------
120
121 void
122 Thread::CheckOverflow()
123 {
124     if (stack != NULL)
125 #ifdef HOST_SNAKE                       // Stacks grow upward on the Snakes
126         ASSERT((unsigned int)stack[StackSize - 1] == STACK_FENCEPOST);
127 #else
128         ASSERT((unsigned int)*stack == STACK_FENCEPOST);
129 #endif
130 }
131
132 //----------------------------------------------------------------------
133 // Thread::Finish
134 //      Called by ThreadRoot when a thread is done executing the
135 //      forked procedure.
136 //
137 //      NOTE: we don't immediately de-allocate the thread data structure
138 //      or the execution stack, because we're still running in the thread
139 //      and we're still on the stack!  Instead, we set "threadToBeDestroyed",
140 //      so that Scheduler::Run() will call the destructor, once we're
141 //      running in the context of a different thread.
142 //
143 //      NOTE: we disable interrupts, so that we don't get a time slice
144 //      between setting threadToBeDestroyed, and going to sleep.
145 //----------------------------------------------------------------------
146
147 //
148 void
149 Thread::Finish ()
150 {
151     (void) interrupt->SetLevel(IntOff);
```

```
152     ASSERT(this == currentThread);
153
154     DEBUG('t', "Finishing thread \"%s\"\n", getName());
155
156     threadToBeDestroyed = currentThread;
157     Sleep();                                    // invokes SWITCH
158     // not reached
159 }
160
161 //----------------------------------------------------------------------
162 // Thread::Yield
163 //      Relinquish the CPU if any other thread is ready to run.
164 //      If so, put the thread on the end of the ready list, so that
165 //      it will eventually be re-scheduled.
166 //
167 //      NOTE: returns immediately if no other thread on the ready queue.
168 //      Otherwise returns when the thread eventually works its way
169 //      to the front of the ready list and gets re-scheduled.
170 //
171 //      NOTE: we disable interrupts, so that looking at the thread
172 //      on the front of the ready list, and switching to it, can be done
173 //      atomically.  On return, we re-set the interrupt level to its
174 //      original state, in case we are called with interrupts disabled.
175 //
176 //      Similar to Thread::Sleep(), but a little different.
177 //----------------------------------------------------------------------
178
179 void
180 Thread::Yield ()
181 {
182     Thread *nextThread;
183     IntStatus oldLevel = interrupt->SetLevel(IntOff);
184
185     ASSERT(this == currentThread);
186
187     DEBUG('t', "Yielding thread \"%s\"\n", getName());
188
189     nextThread = scheduler->FindNextToRun();
190     if (nextThread != NULL) {
191         scheduler->ReadyToRun(this);
192         scheduler->Run(nextThread);
193     }
194     (void) interrupt->SetLevel(oldLevel);
195 }
196
197 //----------------------------------------------------------------------
198 // Thread::Sleep
199 //       Relinquish the CPU, because the current thread is blocked
200 //      waiting on a synchronization variable (Semaphore, Lock, or Condition).
201 //      Eventually, some thread will wake this thread up, and put it
202 //      back on the ready queue, so that it can be re-scheduled.
203 //
204 //      NOTE: if there are no threads on the ready queue, that means
205 //      we have no thread to run.  "Interrupt::Idle" is called
206 //      to signify that we should idle the CPU until the next I/O interrupt
207 //      occurs (the only thing that could cause a thread to become
208 //      ready to run).
209 //
```

```
210 //      NOTE: we assume interrupts are already disabled, because it
211 //      is called from the synchronization routines which must
212 //      disable interrupts for atomicity.   We need interrupts off
213 //      so that there can't be a time slice between pulling the first thread
214 //      off the ready list, and switching to it.
215 //----------------------------------------------------------------------
216 void
217 Thread::Sleep ()
218 {
219     Thread *nextThread;
220
221     ASSERT(this == currentThread);
222     ASSERT(interrupt->getLevel() == IntOff);
223
224     DEBUG('t', "Sleeping thread \"%s\"\n", getName());
225
226     status = BLOCKED;
227     while ((nextThread = scheduler->FindNextToRun()) == NULL)
228         interrupt->Idle();      // no one to run, wait for an interrupt
229
230     scheduler->Run(nextThread); // returns when we've been signalled
231 }
232
233 //----------------------------------------------------------------------
234 // ThreadFinish, InterruptEnable, ThreadPrint
235 //      Dummy functions because C++ does not allow a pointer to a member
236 //      function.  So in order to do this, we create a dummy C function
237 //      (which we can pass a pointer to), that then simply calls the
238 //      member function.
239 //----------------------------------------------------------------------
240
241 static void ThreadFinish()    { currentThread->Finish(); }
242 static void InterruptEnable() { interrupt->Enable(); }
243 void ThreadPrint(_int arg){ Thread *t = (Thread *)arg; t->Print(); }
244
245 //----------------------------------------------------------------------
246 // Thread::StackAllocate
247 //      Allocate and initialize an execution stack.  The stack is
248 //      initialized with an initial stack frame for ThreadRoot, which:
249 //              enables interrupts
250 //              calls (*func)(arg)
251 //              calls Thread::Finish
252 //
253 //      "func" is the procedure to be forked
254 //      "arg" is the parameter to be passed to the procedure
255 //----------------------------------------------------------------------
256
257 void
258 Thread::StackAllocate (VoidFunctionPtr func, _int arg)
259 {
260     stack = (int *) AllocBoundedArray(StackSize * sizeof(_int));
261
262 #ifdef HOST_SNAKE
263     // HP stack works from low addresses to high addresses
264     stackTop = stack + 16;      // HP requires 64-byte frame marker
265     stack[StackSize - 1] = STACK_FENCEPOST;
266 #else
267     // i386 & MIPS & SPARC & ALPHA stack works from high addresses to low addresses
```

```
268 #ifdef HOST_SPARC
269     // SPARC stack must contains at least 1 activation record to start with.
270     stackTop = stack + StackSize - 96;
271 #else  // HOST_MIPS  || HOST_i386 || HOST_ALPHA
272     stackTop = stack + StackSize - 4;   // -4 to be on the safe side!
273 #ifdef HOST_i386
274     // the 80386 passes the return address on the stack.  In order for
275     // SWITCH() to go to ThreadRoot when we switch to this thread, the
276     // return addres used in SWITCH() must be the starting address of
277     // ThreadRoot.
278     *(--stackTop) = (int)ThreadRoot;
279 #endif
280 #endif  // HOST_SPARC
281     *stack = STACK_FENCEPOST;
282 #endif  // HOST_SNAKE
283
284     machineState[PCState] = (_int) ThreadRoot;
285     machineState[StartupPCState] = (_int) InterruptEnable;
286     machineState[InitialPCState] = (_int) func;
287     machineState[InitialArgState] = arg;
288     machineState[WhenDonePCState] = (_int) ThreadFinish;
289 }
290
291 #ifdef USER_PROGRAM
292 #include "machine.h"
293
294 //----------------------------------------------------------------------
295 // Thread::SaveUserState
296 //      Save the CPU state of a user program on a context switch.
297 //
298 //      Note that a user program thread has *two* sets of CPU registers --
299 //      one for its state while executing user code, one for its state
300 //      while executing kernel code.  This routine saves the former.
301 //----------------------------------------------------------------------
302
303 void
304 Thread::SaveUserState()
305 {
306     for (int i = 0; i < NumTotalRegs; i++)
307         userRegisters[i] = machine->ReadRegister(i);
308 }
309
310 //----------------------------------------------------------------------
311 // Thread::RestoreUserState
312 //      Restore the CPU state of a user program on a context switch.
313 //
314 //      Note that a user program thread has *two* sets of CPU registers --
315 //      one for its state while executing user code, one for its state
316 //      while executing kernel code.  This routine restores the former.
317 //----------------------------------------------------------------------
318
319 void
320 Thread::RestoreUserState()
321 {
322     for (int i = 0; i < NumTotalRegs; i++)
323         machine->WriteRegister(i, userRegisters[i]);
324 }
325 #endif
```

1.17 threadtest.cc

```
 1 // threadtest.cc
 2 //      Simple test case for the threads assignment.
 3 //
 4 //      Create two threads, and have them context switch
 5 //      back and forth between themselves by calling Thread::Yield,
 6 //      to illustratethe inner workings of the thread system.
 7 //
 8 // Copyright (c) 1992-1993 The Regents of the University of California.
 9 // All rights reserved.  See copyright.h for copyright notice and limitation
10 // of liability and disclaimer of warranty provisions.
11
12 #include "copyright.h"
13 #include "system.h"
14
15 //----------------------------------------------------------------------
16 // SimpleThread
17 //      Loop 5 times, yielding the CPU to another ready thread
18 //      each iteration.
19 //
20 //      "which" is simply a number identifying the thread, for debugging
21 //      purposes.
22 //----------------------------------------------------------------------
23
24 void
25 SimpleThread(_int which)
26 {
27     int num;
28
29     for (num = 0; num < 5; num++) {
30         printf("*** thread %d looped %d times\n", (int) which, num);
31         currentThread->Yield();
32     }
33 }
34
35 //----------------------------------------------------------------------
36 // ThreadTest
37 //      Set up a ping-pong between two threads, by forking a thread
38 //      to call SimpleThread, and then calling SimpleThread ourselves.
39 //----------------------------------------------------------------------
40
41 void
42 ThreadTest()
43 {
44     DEBUG('t', "Entering SimpleTest");
45
46     Thread *t = new Thread("forked thread");
47
48     t->Fork(SimpleThread, 1);
49     SimpleThread(0);
50 }
51
```

1.18 utility.h

```
 1 // utility.h
```

```
 2 //      Miscellaneous useful definitions, including debugging routines.
 3 //
 4 //      The debugging routines allow the user to turn on selected
 5 //      debugging messages, controllable from the command line arguments
 6 //      passed to Nachos (-d).  You are encouraged to add your own
 7 //      debugging flags.  The pre-defined debugging flags are:
 8 //
 9 //      '+' -- turn on all debug messages
10 //      't' -- thread system
11 //      's' -- semaphores, locks, and conditions
12 //      'i' -- interrupt emulation
13 //      'm' -- machine emulation (USER_PROGRAM)
14 //      'd' -- disk emulation (FILESYS)
15 //      'f' -- file system (FILESYS)
16 //      'a' -- address spaces (USER_PROGRAM)
17 //      'n' -- network emulation (NETWORK)
18 //
19 // Copyright (c) 1992-1993 The Regents of the University of California.
20 // All rights reserved.  See copyright.h for copyright notice and limitation
21 // of liability and disclaimer of warranty provisions.
22
23 #ifndef UTILITY_H
24 #define UTILITY_H
25
26 #include "copyright.h"
27
28 #ifdef HOST_ALPHA               // Needed because of gcc uses 64 bit pointers and
29 #define _int long               // 32 bit integers on the DEC ALPHA architecture.
30 #else
31 #define _int int
32 #endif
33
34 // Miscellaneous useful routines
35
36 #include <bool.h>
37                                                 // Boolean values.
38                                                 // This is the same definition
39                                                 // as in the g++ library.
40 /*
41 #ifdef FALSE
42 #undef FALSE
43 #endif
44 #ifdef TRUE
45 #undef TRUE
46 #endif
47
48 #define FALSE 0
49 #define TRUE  1
50
51 #define bool int                // Needed to avoid problems if the bool type
52                                 // is already defined and when boolean values
53                                 // are assigned to integer variables.
54 */
55
56 #define min(a,b)  (((a) < (b)) ? (a) : (b))
57 #define max(a,b)  (((a) > (b)) ? (a) : (b))
58
59 // Divide and either round up or down
```

```
60 #define divRoundDown(n,s)  ((n) / (s))
61 #define divRoundUp(n,s)    (((n) / (s)) + ((((n) % (s)) > 0) ? 1 : 0))
62
63 // This declares the type "VoidFunctionPtr" to be a "pointer to a
64 // function taking an integer argument and returning nothing".  With
65 // such a function pointer (say it is "func"), we can call it like this:
66 //
67 //       (*func) (17);
68 //
69 // This is used by Thread::Fork and for interrupt handlers, as well
70 // as a couple of other places.
71
72 typedef void (*VoidFunctionPtr)(_int arg);
73 typedef void (*VoidNoArgFunctionPtr)();
74
75
76 // Include interface that isolates us from the host machine system library.
77 // Requires definition of bool, and VoidFunctionPtr
78 #include "sysdep.h"
79
80 // Interface to debugging routines.
81
82 extern void DebugInit(char* flags);     // enable printing debug messages
83
84 extern bool DebugIsEnabled(char flag);  // Is this debug flag enabled?
85
86 extern void DEBUG (char flag, char* format, ...);       // Print debug message
87                                                         // if flag is enabled
88
89 //----------------------------------------------------------------------
90 // ASSERT
91 //      If condition is false,  print a message and dump core.
92 //      Useful for documenting assumptions in the code.
93 //
94 //      NOTE: needs to be a #define, to be able to print the location
95 //      where the error occurred.
96 //----------------------------------------------------------------------
97 #define ASSERT(condition)                                                   \
98     if (!(condition)) {                                                     \
99         fprintf(stderr, "Assertion failed: line %d, file \"%s\"\n",         \
100                 __LINE__, __FILE__);                                       \
101         fflush(stderr);                                                    \
102         Abort();                                                           \
103     }
104
105
106 #endif UTILITY_H
```

1.19 utility.cc

```
1 // utility.cc
2 //      Debugging routines.  Allows users to control whether to
3 //      print DEBUG statements, based on a command line argument.
4 //
5 // Copyright (c) 1992-1993 The Regents of the University of California.
6 // All rights reserved.  See copyright.h for copyright notice and limitation
7 // of liability and disclaimer of warranty provisions.
```

```
 8
 9 #include "copyright.h"
10 #include "utility.h"
11
12 // this seems to be dependent on how the compiler is configured.
13 // if you have problems with va_start, try both of these alternatives
14 #include <stdarg.h>
15
16 static char *enableFlags = NULL; // controls which DEBUG messages are printed
17
18 //----------------------------------------------------------------------
19 // DebugInit
20 //      Initialize so that only DEBUG messages with a flag in flagList
21 //      will be printed.
22 //
23 //      If the flag is "+", we enable all DEBUG messages.
24 //
25 //      "flagList" is a string of characters for whose DEBUG messages are
26 //              to be enabled.
27 //----------------------------------------------------------------------
28
29 void
30 DebugInit(char *flagList)
31 {
32     enableFlags = flagList;
33 }
34
35 //----------------------------------------------------------------------
36 // DebugIsEnabled
37 //      Return TRUE if DEBUG messages with "flag" are to be printed.
38 //----------------------------------------------------------------------
39
40 bool
41 DebugIsEnabled(char flag)
42 {
43     if (enableFlags != NULL)
44        return (bool)((strchr(enableFlags, flag) != 0)
45                || (strchr(enableFlags, '+') != 0));
46     else
47       return FALSE;
48 }
49
50 //----------------------------------------------------------------------
51 // DEBUG
52 //      Print a debug message, if flag is enabled.  Like printf,
53 //      only with an extra argument on the front.
54 //----------------------------------------------------------------------
55
56 void
57 DEBUG(char flag, char *format, ...)
58 {
59     if (DebugIsEnabled(flag)) {
60         va_list ap;
61         // You will get an unused variable message here -- ignore it.
62         va_start(ap, format);
63         vfprintf(stdout, format, ap);
64         va_end(ap);
65         fflush(stdout);
```

```
66     }
67 }
```

Chapter 2

Directory ../filesys/

Contents

2.1 directory.h . . . 53
2.2 directory.cc . . . 55
2.3 filehdr.h . . . 58
2.4 filehdr.cc . . . 59
2.5 filesys.h . . . 62
2.6 filesys.cc . . . 64
2.7 fstest.cc . . . 70
2.8 openfile.h . . . 73
2.9 openfile.cc . . . 75
2.10 synchdisk.h . . . 78
2.11 synchdisk.h . . . 79

This chapter lists all the source codes found in directory `../filesys/`. They are:

```
directory.cc   filehdr.h      fstest.cc      synchdisk.cc
directory.h    filesys.cc     openfile.cc    synchdisk.h
filehdr.cc     filesys.h      openfile.h
```

2.1 directory.h

```
 1 // directory.h
 2 //      Data structures to manage a UNIX-like directory of file names.
 3 //
 4 //      A directory is a table of pairs: <file name, sector #>,
 5 //      giving the name of each file in the directory, and
 6 //      where to find its file header (the data structure describing
 7 //      where to find the file's data blocks) on disk.
 8 //
 9 //      We assume mutual exclusion is provided by the caller.
10 //
11 // Copyright (c) 1992-1993 The Regents of the University of California.
12 // All rights reserved.  See copyright.h for copyright notice and limitation
13 // of liability and disclaimer of warranty provisions.
14
15 #include "copyright.h"
16
17 #ifndef DIRECTORY_H
```

```
18 #define DIRECTORY_H
19
20 #include "openfile.h"
21
22 #define FileNameMaxLen          9       // for simplicity, we assume
23                                         // file names are <= 9 characters long
24
25 // The following class defines a "directory entry", representing a file
26 // in the directory.  Each entry gives the name of the file, and where
27 // the file's header is to be found on disk.
28 //
29 // Internal data structures kept public so that Directory operations can
30 // access them directly.
31
32 class DirectoryEntry {
33   public:
34     bool inUse;                         // Is this directory entry in use?
35     int sector;                         // Location on disk to find the
36                                         //   FileHeader for this file
37     char name[FileNameMaxLen + 1];      // Text name for file, with +1 for
38                                         // the trailing '\0'
39 };
40
41 // The following class defines a UNIX-like "directory".  Each entry in
42 // the directory describes a file, and where to find it on disk.
43 //
44 // The directory data structure can be stored in memory, or on disk.
45 // When it is on disk, it is stored as a regular Nachos file.
46 //
47 // The constructor initializes a directory structure in memory; the
48 // FetchFrom/WriteBack operations shuffle the directory information
49 // from/to disk.
50
51 class Directory {
52   public:
53     Directory(int size);                // Initialize an empty directory
54                                         // with space for "size" files
55     ~Directory();                       // De-allocate the directory
56
57     void FetchFrom(OpenFile *file);     // Init directory contents from disk
58     void WriteBack(OpenFile *file);     // Write modifications to
59                                         // directory contents back to disk
60
61     int Find(char *name);               // Find the sector number of the
62                                         // FileHeader for file: "name"
63
64     bool Add(char *name, int newSector);  // Add a file name into the directory
65
66     bool Remove(char *name);            // Remove a file from the directory
67
68     void List();                        // Print the names of all the files
69                                         //  in the directory
70     void Print();                       // Verbose print of the contents
71                                         //  of the directory -- all the file
72                                         //  names and their contents.
73
74   private:
75     int tableSize;                      // Number of directory entries
```

```
76     DirectoryEntry *table;              // Table of pairs:
77                                         // <file name, file header location>
78
79     int FindIndex(char *name);          // Find the index into the directory
80                                         //  table corresponding to "name"
81 };
82
83 #endif // DIRECTORY_H
```

2.2 directory.cc

```
 1 // directory.cc
 2 //      Routines to manage a directory of file names.
 3 //
 4 //      The directory is a table of fixed length entries; each
 5 //      entry represents a single file, and contains the file name,
 6 //      and the location of the file header on disk.  The fixed size
 7 //      of each directory entry means that we have the restriction
 8 //      of a fixed maximum size for file names.
 9 //
10 //      The constructor initializes an empty directory of a certain size;
11 //      we use ReadFrom/WriteBack to fetch the contents of the directory
12 //      from disk, and to write back any modifications back to disk.
13 //
14 //      Also, this implementation has the restriction that the size
15 //      of the directory cannot expand.  In other words, once all the
16 //      entries in the directory are used, no more files can be created.
17 //      Fixing this is one of the parts to the assignment.
18 //
19 // Copyright (c) 1992-1993 The Regents of the University of California.
20 // All rights reserved.  See copyright.h for copyright notice and limitation
21 // of liability and disclaimer of warranty provisions.
22
23 #include "copyright.h"
24 #include "utility.h"
25 #include "filehdr.h"
26 #include "directory.h"
27
28 //----------------------------------------------------------------------
29 // Directory::Directory
30 //      Initialize a directory; initially, the directory is completely
31 //      empty.  If the disk is being formatted, an empty directory
32 //      is all we need, but otherwise, we need to call FetchFrom in order
33 //      to initialize it from disk.
34 //
35 //      "size" is the number of entries in the directory
36 //----------------------------------------------------------------------
37
38 Directory::Directory(int size)
39 {
40     table = new DirectoryEntry[size];
41     tableSize = size;
42     for (int i = 0; i < tableSize; i++)
43         table[i].inUse = FALSE;
44 }
45
46 //----------------------------------------------------------------------
```

```
47 // Directory::~Directory
48 //      De-allocate directory data structure.
49 //----------------------------------------------------------------------
50
51 Directory::~Directory()
52 {
53     delete [] table;
54 }
55
56 //----------------------------------------------------------------------
57 // Directory::FetchFrom
58 //      Read the contents of the directory from disk.
59 //
60 //      "file" -- file containing the directory contents
61 //----------------------------------------------------------------------
62
63 void
64 Directory::FetchFrom(OpenFile *file)
65 {
66     (void) file->ReadAt((char *)table, tableSize * sizeof(DirectoryEntry), 0);
67 }
68
69 //----------------------------------------------------------------------
70 // Directory::WriteBack
71 //      Write any modifications to the directory back to disk
72 //
73 //      "file" -- file to contain the new directory contents
74 //----------------------------------------------------------------------
75
76 void
77 Directory::WriteBack(OpenFile *file)
78 {
79     (void) file->WriteAt((char *)table, tableSize * sizeof(DirectoryEntry), 0);
80 }
81
82 //----------------------------------------------------------------------
83 // Directory::FindIndex
84 //      Look up file name in directory, and return its location in the table of
85 //      directory entries.  Return -1 if the name isn't in the directory.
86 //
87 //      "name" -- the file name to look up
88 //----------------------------------------------------------------------
89
90 int
91 Directory::FindIndex(char *name)
92 {
93     for (int i = 0; i < tableSize; i++)
94         if (table[i].inUse && !strncmp(table[i].name, name, FileNameMaxLen))
95             return i;
96     return -1;          // name not in directory
97 }
98
99 //----------------------------------------------------------------------
100 // Directory::Find
101 //      Look up file name in directory, and return the disk sector number
102 //      where the file's header is stored. Return -1 if the name isn't
103 //      in the directory.
104 //
```

```
105 //      "name" -- the file name to look up
106 //----------------------------------------------------------------------
107
108 int
109 Directory::Find(char *name)
110 {
111     int i = FindIndex(name);
112
113     if (i != -1)
114         return table[i].sector;
115     return -1;
116 }
117
118 //----------------------------------------------------------------------
119 // Directory::Add
120 //      Add a file into the directory.  Return TRUE if successful;
121 //      return FALSE if the file name is already in the directory, or if
122 //      the directory is completely full, and has no more space for
123 //      additional file names.
124 //
125 //      "name" -- the name of the file being added
126 //      "newSector" -- the disk sector containing the added file's header
127 //----------------------------------------------------------------------
128
129 bool
130 Directory::Add(char *name, int newSector)
131 {
132     if (FindIndex(name) != -1)
133          return FALSE;
134
135     for (int i = 0; i < tableSize; i++)
136         if (!table[i].inUse) {
137             table[i].inUse = TRUE;
138             strncpy(table[i].name, name, FileNameMaxLen);
139             table[i].sector = newSector;
140         return TRUE;
141         }
142     return FALSE;       // no space.  Fix when we have extensible files.
143 }
144
145 //----------------------------------------------------------------------
146 // Directory::Remove
147 //      Remove a file name from the directory.  Return TRUE if successful;
148 //      return FALSE if the file isn't in the directory.
149 //
150 //      "name" -- the file name to be removed
151 //----------------------------------------------------------------------
152
153 bool
154 Directory::Remove(char *name)
155 {
156     int i = FindIndex(name);
157
158     if (i == -1)
159         return FALSE;           // name not in directory
160     table[i].inUse = FALSE;
161     return TRUE;
162 }
```

```
163
164 //----------------------------------------------------------------------
165 // Directory::List
166 //      List all the file names in the directory.
167 //----------------------------------------------------------------------
168
169 void
170 Directory::List()
171 {
172    for (int i = 0; i < tableSize; i++)
173         if (table[i].inUse)
174             printf("%s\n", table[i].name);
175 }
176
177 //----------------------------------------------------------------------
178 // Directory::Print
179 //      List all the file names in the directory, their FileHeader locations,
180 //      and the contents of each file.  For debugging.
181 //----------------------------------------------------------------------
182
183 void
184 Directory::Print()
185 {
186     FileHeader *hdr = new FileHeader;
187
188     printf("Directory contents:\n");
189     for (int i = 0; i < tableSize; i++)
190         if (table[i].inUse) {
191             printf("Name: %s, Sector: %d\n", table[i].name, table[i].sector);
192             hdr->FetchFrom(table[i].sector);
193             hdr->Print();
194         }
195     printf("\n");
196     delete hdr;
197 }
```

2.3 filehdr.h

```
 1 // filehdr.h
 2 //      Data structures for managing a disk file header.
 3 //
 4 //      A file header describes where on disk to find the data in a file,
 5 //      along with other information about the file (for instance, its
 6 //      length, owner, etc.)
 7 //
 8 // Copyright (c) 1992-1993 The Regents of the University of California.
 9 // All rights reserved.  See copyright.h for copyright notice and limitation
10 // of liability and disclaimer of warranty provisions.
11
12 #include "copyright.h"
13
14 #ifndef FILEHDR_H
15 #define FILEHDR_H
16
17 #include "disk.h"
18 #include "bitmap.h"
19
```

```
20 #define NumDirect       ((SectorSize - 2 * sizeof(int)) / sizeof(int))
21 #define MaxFileSize     (NumDirect * SectorSize)
22
23 // The following class defines the Nachos "file header" (in UNIX terms,
24 // the "i-node"), describing where on disk to find all of the data in the file.
25 // The file header is organized as a simple table of pointers to
26 // data blocks.
27 //
28 // The file header data structure can be stored in memory or on disk.
29 // When it is on disk, it is stored in a single sector -- this means
30 // that we assume the size of this data structure to be the same
31 // as one disk sector.  Without indirect addressing, this
32 // limits the maximum file length to just under 4K bytes.
33 //
34 // There is no constructor; rather the file header can be initialized
35 // by allocating blocks for the file (if it is a new file), or by
36 // reading it from disk.
37
38 class FileHeader {
39   public:
40     bool Allocate(BitMap *bitMap, int fileSize);// Initialize a file header,
41                                                 //  including allocating space
42                                                 //  on disk for the file data
43     void Deallocate(BitMap *bitMap);            // De-allocate this file's
44                                                 //  data blocks
45
46     void FetchFrom(int sectorNumber);   // Initialize file header from disk
47     void WriteBack(int sectorNumber);   // Write modifications to file header
48                                         //  back to disk
49
50     int ByteToSector(int offset);       // Convert a byte offset into the file
51                                         // to the disk sector containing
52                                         // the byte
53
54     int FileLength();                   // Return the length of the file
55                                         // in bytes
56
57     void Print();                       // Print the contents of the file.
58
59   private:
60     int numBytes;                       // Number of bytes in the file
61     int numSectors;                     // Number of data sectors in the file
62     int dataSectors[NumDirect];         // Disk sector numbers for each data
63                                         // block in the file
64 };
65
66 #endif // FILEHDR_H
```

2.4 filehdr.cc

```
1 // filehdr.cc
2 //      Routines for managing the disk file header (in UNIX, this
3 //      would be called the i-node).
4 //
5 //      The file header is used to locate where on disk the
6 //      file's data is stored.  We implement this as a fixed size
7 //      table of pointers -- each entry in the table points to the
```

```
 8 //      disk sector containing that portion of the file data
 9 //      (in other words, there are no indirect or doubly indirect
10 //      blocks). The table size is chosen so that the file header
11 //      will be just big enough to fit in one disk sector,
12 //
13 //      Unlike in a real system, we do not keep track of file permissions,
14 //      ownership, last modification date, etc., in the file header.
15 //
16 //      A file header can be initialized in two ways:
17 //         for a new file, by modifying the in-memory data structure
18 //           to point to the newly allocated data blocks
19 //         for a file already on disk, by reading the file header from disk
20 //
21 // Copyright (c) 1992-1993 The Regents of the University of California.
22 // All rights reserved.  See copyright.h for copyright notice and limitation
23 // of liability and disclaimer of warranty provisions.
24
25 #include "copyright.h"
26
27 #include "system.h"
28 #include "filehdr.h"
29
30 //----------------------------------------------------------------------
31 // FileHeader::Allocate
32 //      Initialize a fresh file header for a newly created file.
33 //      Allocate data blocks for the file out of the map of free disk blocks.
34 //      Return FALSE if there are not enough free blocks to accomodate
35 //      the new file.
36 //
37 //      "freeMap" is the bit map of free disk sectors
38 //      "fileSize" is the bit map of free disk sectors
39 //----------------------------------------------------------------------
40
41 bool
42 FileHeader::Allocate(BitMap *freeMap, int fileSize)
43 {
44     numBytes = fileSize;
45     numSectors  = divRoundUp(fileSize, SectorSize);
46     if (freeMap->NumClear() < numSectors)
47         return FALSE;           // not enough space
48
49     for (int i = 0; i < numSectors; i++)
50         dataSectors[i] = freeMap->Find();
51     return TRUE;
52 }
53
54 //----------------------------------------------------------------------
55 // FileHeader::Deallocate
56 //      De-allocate all the space allocated for data blocks for this file.
57 //
58 //      "freeMap" is the bit map of free disk sectors
59 //----------------------------------------------------------------------
60
61 void
62 FileHeader::Deallocate(BitMap *freeMap)
63 {
64     for (int i = 0; i < numSectors; i++) {
65         ASSERT(freeMap->Test((int) dataSectors[i]));  // ought to be marked!
```

```
66          freeMap->Clear((int) dataSectors[i]);
67      }
68  }
69
70  //----------------------------------------------------------------------
71  // FileHeader::FetchFrom
72  //      Fetch contents of file header from disk.
73  //
74  //      "sector" is the disk sector containing the file header
75  //----------------------------------------------------------------------
76
77  void
78  FileHeader::FetchFrom(int sector)
79  {
80      synchDisk->ReadSector(sector, (char *)this);
81  }
82
83  //----------------------------------------------------------------------
84  // FileHeader::WriteBack
85  //      Write the modified contents of the file header back to disk.
86  //
87  //      "sector" is the disk sector to contain the file header
88  //----------------------------------------------------------------------
89
90  void
91  FileHeader::WriteBack(int sector)
92  {
93      synchDisk->WriteSector(sector, (char *)this);
94  }
95
96  //----------------------------------------------------------------------
97  // FileHeader::ByteToSector
98  //      Return which disk sector is storing a particular byte within the file.
99  //      This is essentially a translation from a virtual address (the
100 //      offset in the file) to a physical address (the sector where the
101 //      data at the offset is stored).
102 //
103 //      "offset" is the location within the file of the byte in question
104 //----------------------------------------------------------------------
105
106 int
107 FileHeader::ByteToSector(int offset)
108 {
109     return(dataSectors[offset / SectorSize]);
110 }
111
112 //----------------------------------------------------------------------
113 // FileHeader::FileLength
114 //      Return the number of bytes in the file.
115 //----------------------------------------------------------------------
116
117 int
118 FileHeader::FileLength()
119 {
120     return numBytes;
121 }
122
123 //----------------------------------------------------------------------
```

```
124 // FileHeader::Print
125 //      Print the contents of the file header, and the contents of all
126 //      the data blocks pointed to by the file header.
127 //----------------------------------------------------------------------
128
129 void
130 FileHeader::Print()
131 {
132     int i, j, k;
133     char *data = new char[SectorSize];
134
135     printf("FileHeader contents.  File size: %d.  File blocks:\n", numBytes);
136     for (i = 0; i < numSectors; i++)
137         printf("%d ", dataSectors[i]);
138     printf("\nFile contents:\n");
139     for (i = k = 0; i < numSectors; i++) {
140         synchDisk->ReadSector(dataSectors[i], data);
141         for (j = 0; (j < SectorSize) && (k < numBytes); j++, k++) {
142             if ('\040' <= data[j] && data[j] <= '\176')   // isprint(data[j])
143                 printf("%c", data[j]);
144             else
145                 printf("\\%x", (unsigned char)data[j]);
146         }
147         printf("\n");
148     }
149     delete [] data;
150 }
```

2.5 filesys.h

```
 1 // filesys.h
 2 //      Data structures to represent the Nachos file system.
 3 //
 4 //      A file system is a set of files stored on disk, organized
 5 //      into directories.  Operations on the file system have to
 6 //      do with "naming" -- creating, opening, and deleting files,
 7 //      given a textual file name.  Operations on an individual
 8 //      "open" file (read, write, close) are to be found in the OpenFile
 9 //      class (openfile.h).
10 //
11 //      We define two separate implementations of the file system.
12 //      The "STUB" version just re-defines the Nachos file system
13 //      operations as operations on the native UNIX file system on the machine
14 //      running the Nachos simulation.  This is provided in case the
15 //      multiprogramming and virtual memory assignments (which make use
16 //      of the file system) are done before the file system assignment.
17 //
18 //      The other version is a "real" file system, built on top of
19 //      a disk simulator.  The disk is simulated using the native UNIX
20 //      file system (in a file named "DISK").
21 //
22 //      In the "real" implementation, there are two key data structures used
23 //      in the file system.  There is a single "root" directory, listing
24 //      all of the files in the file system; unlike UNIX, the baseline
25 //      system does not provide a hierarchical directory structure.
26 //      In addition, there is a bitmap for allocating
27 //      disk sectors.  Both the root directory and the bitmap are themselves
```

```
28 //      stored as files in the Nachos file system -- this causes an interesting
29 //      bootstrap problem when the simulated disk is initialized.
30 //
31 // Copyright (c) 1992-1993 The Regents of the University of California.
32 // All rights reserved.  See copyright.h for copyright notice and limitation
33 // of liability and disclaimer of warranty provisions.
34
35 #ifndef FS_H
36 #define FS_H
37
38 #include "copyright.h"
39 #include "openfile.h"
40
41 #ifdef FILESYS_STUB             // Temporarily implement file system calls as
42                                 // calls to UNIX, until the real file system
43                                 // implementation is available
44 class FileSystem {
45   public:
46     FileSystem(bool format) {}
47
48     bool Create(char *name, int initialSize) {
49         int fileDescriptor = OpenForWrite(name);
50
51         if (fileDescriptor == -1) return FALSE;
52         Close(fileDescriptor);
53         return TRUE;
54         }
55
56     OpenFile* Open(char *name) {
57           int fileDescriptor = OpenForReadWrite(name, FALSE);
58
59           if (fileDescriptor == -1) return NULL;
60           return new OpenFile(fileDescriptor);
61       }
62
63     bool Remove(char *name) { return (bool)(Unlink(name) == 0); }
64
65 };
66
67 #else // FILESYS
68 class FileSystem {
69   public:
70     FileSystem(bool format);            // Initialize the file system.
71                                         // Must be called *after* "synchDisk"
72                                         // has been initialized.
73                                         // If "format", there is nothing on
74                                         // the disk, so initialize the directory
75                                         // and the bitmap of free blocks.
76
77     bool Create(char *name, int initialSize);
78                                         // Create a file (UNIX creat)
79
80     OpenFile* Open(char *name);         // Open a file (UNIX open)
81
82     bool Remove(char *name);            // Delete a file (UNIX unlink)
83
84     void List();                        // List all the files in the file system
85
```

```
86     void Print();                       // List all the files and their contents
87
88   private:
89    OpenFile* freeMapFile;               // Bit map of free disk blocks,
90                                         // represented as a file
91    OpenFile* directoryFile;             // "Root" directory -- list of
92                                         // file names, represented as a file
93 };
94
95 #endif // FILESYS
96
97 #endif // FS_H
```

2.6 filesys.cc

```
 1 // filesys.cc
 2 //      Routines to manage the overall operation of the file system.
 3 //      Implements routines to map from textual file names to files.
 4 //
 5 //      Each file in the file system has:
 6 //         A file header, stored in a sector on disk
 7 //              (the size of the file header data structure is arranged
 8 //              to be precisely the size of 1 disk sector)
 9 //         A number of data blocks
10 //         An entry in the file system directory
11 //
12 //      The file system consists of several data structures:
13 //         A bitmap of free disk sectors (cf. bitmap.h)
14 //         A directory of file names and file headers
15 //
16 //      Both the bitmap and the directory are represented as normal
17 //      files.  Their file headers are located in specific sectors
18 //      (sector 0 and sector 1), so that the file system can find them
19 //      on bootup.
20 //
21 //      The file system assumes that the bitmap and directory files are
22 //      kept "open" continuously while Nachos is running.
23 //
24 //      For those operations (such as Create, Remove) that modify the
25 //      directory and/or bitmap, if the operation succeeds, the changes
26 //      are written immediately back to disk (the two files are kept
27 //      open during all this time).  If the operation fails, and we have
28 //      modified part of the directory and/or bitmap, we simply discard
29 //      the changed version, without writing it back to disk.
30 //
31 //      Our implementation at this point has the following restrictions:
32 //
33 //         there is no synchronization for concurrent accesses
34 //         files have a fixed size, set when the file is created
35 //         files cannot be bigger than about 3KB in size
36 //         there is no hierarchical directory structure, and only a limited
37 //           number of files can be added to the system
38 //         there is no attempt to make the system robust to failures
39 //          (if Nachos exits in the middle of an operation that modifies
40 //          the file system, it may corrupt the disk)
41 //
42 // Copyright (c) 1992-1993 The Regents of the University of California.
```

```
43 // All rights reserved.  See copyright.h for copyright notice and limitation
44 // of liability and disclaimer of warranty provisions.
45
46 #include "copyright.h"
47
48 #include "disk.h"
49 #include "bitmap.h"
50 #include "directory.h"
51 #include "filehdr.h"
52 #include "filesys.h"
53
54 // Sectors containing the file headers for the bitmap of free sectors,
55 // and the directory of files.  These file headers are placed in well-known
56 // sectors, so that they can be located on boot-up.
57 #define FreeMapSector           0
58 #define DirectorySector         1
59
60 // Initial file sizes for the bitmap and directory; until the file system
61 // supports extensible files, the directory size sets the maximum number
62 // of files that can be loaded onto the disk.
63 #define FreeMapFileSize         (NumSectors / BitsInByte)
64 #define NumDirEntries           10
65 #define DirectoryFileSize       (sizeof(DirectoryEntry) * NumDirEntries)
66
67 //----------------------------------------------------------------------
68 // FileSystem::FileSystem
69 //      Initialize the file system.  If format = TRUE, the disk has
70 //      nothing on it, and we need to initialize the disk to contain
71 //      an empty directory, and a bitmap of free sectors (with almost but
72 //      not all of the sectors marked as free).
73 //
74 //      If format = FALSE, we just have to open the files
75 //      representing the bitmap and the directory.
76 //
77 //      "format" -- should we initialize the disk?
78 //----------------------------------------------------------------------
79
80 FileSystem::FileSystem(bool format)
81 {
82     DEBUG('f', "Initializing the file system.\n");
83     if (format) {
84         BitMap *freeMap = new BitMap(NumSectors);
85         Directory *directory = new Directory(NumDirEntries);
86         FileHeader *mapHdr = new FileHeader;
87         FileHeader *dirHdr = new FileHeader;
88
89         DEBUG('f', "Formatting the file system.\n");
90
91     // First, allocate space for FileHeaders for the directory and bitmap
92     // (make sure no one else grabs these!)
93         freeMap->Mark(FreeMapSector);
94         freeMap->Mark(DirectorySector);
95
96     // Second, allocate space for the data blocks containing the contents
97     // of the directory and bitmap files.  There better be enough space!
98
99         ASSERT(mapHdr->Allocate(freeMap, FreeMapFileSize));
100        ASSERT(dirHdr->Allocate(freeMap, DirectoryFileSize));
```

```
101
102     // Flush the bitmap and directory FileHeaders back to disk
103     // We need to do this before we can "Open" the file, since open
104     // reads the file header off of disk (and currently the disk has garbage
105     // on it!).
106
107         DEBUG('f', "Writing headers back to disk.\n");
108         mapHdr->WriteBack(FreeMapSector);
109         dirHdr->WriteBack(DirectorySector);
110
111     // OK to open the bitmap and directory files now
112     // The file system operations assume these two files are left open
113     // while Nachos is running.
114
115         freeMapFile = new OpenFile(FreeMapSector);
116         directoryFile = new OpenFile(DirectorySector);
117
118     // Once we have the files "open", we can write the initial version
119     // of each file back to disk.  The directory at this point is completely
120     // empty; but the bitmap has been changed to reflect the fact that
121     // sectors on the disk have been allocated for the file headers and
122     // to hold the file data for the directory and bitmap.
123
124         DEBUG('f', "Writing bitmap and directory back to disk.\n");
125         freeMap->WriteBack(freeMapFile);         // flush changes to disk
126         directory->WriteBack(directoryFile);
127
128         if (DebugIsEnabled('f')) {
129             freeMap->Print();
130             directory->Print();
131
132         delete freeMap;
133          delete directory;
134         delete mapHdr;
135         delete dirHdr;
136         }
137     } else {
138     // if we are not formatting the disk, just open the files representing
139     // the bitmap and directory; these are left open while Nachos is running
140         freeMapFile = new OpenFile(FreeMapSector);
141         directoryFile = new OpenFile(DirectorySector);
142     }
143 }
144
145 //----------------------------------------------------------------------
146 // FileSystem::Create
147 //      Create a file in the Nachos file system (similar to UNIX create).
148 //      Since we can't increase the size of files dynamically, we have
149 //      to give Create the initial size of the file.
150 //
151 //      The steps to create a file are:
152 //        Make sure the file doesn't already exist
153 //        Allocate a sector for the file header
154 //        Allocate space on disk for the data blocks for the file
155 //        Add the name to the directory
156 //        Store the new file header on disk
157 //        Flush the changes to the bitmap and the directory back to disk
158 //
```

```
159 //      Return TRUE if everything goes ok, otherwise, return FALSE.
160 //
161 //      Create fails if:
162 //              file is already in directory
163 //              no free space for file header
164 //              no free entry for file in directory
165 //              no free space for data blocks for the file
166 //
167 //      Note that this implementation assumes there is no concurrent access
168 //      to the file system!
169 //
170 //      "name" -- name of file to be created
171 //      "initialSize" -- size of file to be created
172 //----------------------------------------------------------------------
173
174 bool
175 FileSystem::Create(char *name, int initialSize)
176 {
177     Directory *directory;
178     BitMap *freeMap;
179     FileHeader *hdr;
180     int sector;
181     bool success;
182
183     DEBUG('f', "Creating file %s, size %d\n", name, initialSize);
184
185     directory = new Directory(NumDirEntries);
186     directory->FetchFrom(directoryFile);
187
188     if (directory->Find(name) != -1)
189       success = FALSE;                  // file is already in directory
190     else {
191         freeMap = new BitMap(NumSectors);
192         freeMap->FetchFrom(freeMapFile);
193         sector = freeMap->Find();       // find a sector to hold the file header
194         if (sector == -1)
195             success = FALSE;            // no free block for file header
196         else if (!directory->Add(name, sector))
197             success = FALSE;    // no space in directory
198         else {
199              hdr = new FileHeader;
200             if (!hdr->Allocate(freeMap, initialSize))
201                 success = FALSE;        // no space on disk for data
202             else {
203                 success = TRUE;
204                 // everthing worked, flush all changes back to disk
205                 hdr->WriteBack(sector);
206                 directory->WriteBack(directoryFile);
207                 freeMap->WriteBack(freeMapFile);
208             }
209             delete hdr;
210         }
211         delete freeMap;
212     }
213     delete directory;
214     return success;
215 }
216
```

```
217 //----------------------------------------------------------------------
218 // FileSystem::Open
219 //      Open a file for reading and writing.
220 //      To open a file:
221 //        Find the location of the file's header, using the directory
222 //        Bring the header into memory
223 //
224 //      "name" -- the text name of the file to be opened
225 //----------------------------------------------------------------------
226
227 OpenFile *
228 FileSystem::Open(char *name)
229 {
230     Directory *directory = new Directory(NumDirEntries);
231     OpenFile *openFile = NULL;
232     int sector;
233
234     DEBUG('f', "Opening file %s\n", name);
235     directory->FetchFrom(directoryFile);
236     sector = directory->Find(name);
237     if (sector >= 0)
238         openFile = new OpenFile(sector);        // name was found in directory
239     delete directory;
240     return openFile;                            // return NULL if not found
241 }
242
243 //----------------------------------------------------------------------
244 // FileSystem::Remove
245 //      Delete a file from the file system.  This requires:
246 //          Remove it from the directory
247 //          Delete the space for its header
248 //          Delete the space for its data blocks
249 //          Write changes to directory, bitmap back to disk
250 //
251 //      Return TRUE if the file was deleted, FALSE if the file wasn't
252 //      in the file system.
253 //
254 //      "name" -- the text name of the file to be removed
255 //----------------------------------------------------------------------
256
257 bool
258 FileSystem::Remove(char *name)
259 {
260     Directory *directory;
261     BitMap *freeMap;
262     FileHeader *fileHdr;
263     int sector;
264
265     directory = new Directory(NumDirEntries);
266     directory->FetchFrom(directoryFile);
267     sector = directory->Find(name);
268     if (sector == -1) {
269        delete directory;
270        return FALSE;                    // file not found
271     }
272     fileHdr = new FileHeader;
273     fileHdr->FetchFrom(sector);
274
```

```
275     freeMap = new BitMap(NumSectors);
276     freeMap->FetchFrom(freeMapFile);
277
278     fileHdr->Deallocate(freeMap);               // remove data blocks
279     freeMap->Clear(sector);                     // remove header block
280     directory->Remove(name);
281
282     freeMap->WriteBack(freeMapFile);            // flush to disk
283     directory->WriteBack(directoryFile);        // flush to disk
284     delete fileHdr;
285     delete directory;
286     delete freeMap;
287     return TRUE;
288 }
289
290 //----------------------------------------------------------------------
291 // FileSystem::List
292 //      List all the files in the file system directory.
293 //----------------------------------------------------------------------
294
295 void
296 FileSystem::List()
297 {
298     Directory *directory = new Directory(NumDirEntries);
299
300     directory->FetchFrom(directoryFile);
301     directory->List();
302     delete directory;
303 }
304
305 //----------------------------------------------------------------------
306 // FileSystem::Print
307 //      Print everything about the file system:
308 //        the contents of the bitmap
309 //        the contents of the directory
310 //        for each file in the directory,
311 //            the contents of the file header
312 //            the data in the file
313 //----------------------------------------------------------------------
314
315 void
316 FileSystem::Print()
317 {
318     FileHeader *bitHdr = new FileHeader;
319     FileHeader *dirHdr = new FileHeader;
320     BitMap *freeMap = new BitMap(NumSectors);
321     Directory *directory = new Directory(NumDirEntries);
322
323     printf("Bit map file header:\n");
324     bitHdr->FetchFrom(FreeMapSector);
325     bitHdr->Print();
326
327     printf("Directory file header:\n");
328     dirHdr->FetchFrom(DirectorySector);
329     dirHdr->Print();
330
331     freeMap->FetchFrom(freeMapFile);
332     freeMap->Print();
```

```
333
334     directory->FetchFrom(directoryFile);
335     directory->Print();
336
337     delete bitHdr;
338     delete dirHdr;
339     delete freeMap;
340     delete directory;
341 }
```

2.7 fstest.cc

```
 1 // fstest.cc
 2 //      Simple test routines for the file system.
 3 //
 4 //      We implement:
 5 //         Copy -- copy a file from UNIX to Nachos
 6 //         Print -- cat the contents of a Nachos file
 7 //         Perftest -- a stress test for the Nachos file system
 8 //              read and write a really large file in tiny chunks
 9 //              (won't work on baseline system!)
10 //
11 // Copyright (c) 1992-1993 The Regents of the University of California.
12 // All rights reserved.  See copyright.h for copyright notice and limitation
13 // of liability and disclaimer of warranty provisions.
14
15 #include "copyright.h"
16
17 #include "utility.h"
18 #include "filesys.h"
19 #include "system.h"
20 #include "thread.h"
21 #include "disk.h"
22 #include "stats.h"
23
24 #define TransferSize    10      // make it small, just to be difficult
25
26 //----------------------------------------------------------------------
27 // Copy
28 //      Copy the contents of the UNIX file "from" to the Nachos file "to"
29 //----------------------------------------------------------------------
30
31 void
32 Copy(char *from, char *to)
33 {
34     FILE *fp;
35     OpenFile* openFile;
36     int amountRead, fileLength;
37     char *buffer;
38
39 // Open UNIX file
40     if ((fp = fopen(from, "r")) == NULL) {
41         printf("Copy: couldn't open input file %s\n", from);
42         return;
43     }
44
45 // Figure out length of UNIX file
```

```
46     fseek(fp, 0, 2);
47     fileLength = ftell(fp);
48     fseek(fp, 0, 0);
49
50 // Create a Nachos file of the same length
51     DEBUG('f', "Copying file %s, size %d, to file %s\n", from, fileLength, to);
52     if (!fileSystem->Create(to, fileLength)) {   // Create Nachos file
53         printf("Copy: couldn't create output file %s\n", to);
54         fclose(fp);
55         return;
56     }
57
58     openFile = fileSystem->Open(to);
59     ASSERT(openFile != NULL);
60
61 // Copy the data in TransferSize chunks
62     buffer = new char[TransferSize];
63     while ((amountRead = fread(buffer, sizeof(char), TransferSize, fp)) > 0)
64         openFile->Write(buffer, amountRead);
65     delete [] buffer;
66
67 // Close the UNIX and the Nachos files
68     delete openFile;
69     fclose(fp);
70 }
71
72 //----------------------------------------------------------------------
73 // Print
74 //      Print the contents of the Nachos file "name".
75 //----------------------------------------------------------------------
76
77 void
78 Print(char *name)
79 {
80     OpenFile *openFile;
81     int i, amountRead;
82     char *buffer;
83
84     if ((openFile = fileSystem->Open(name)) == NULL) {
85         printf("Print: unable to open file %s\n", name);
86         return;
87     }
88
89     buffer = new char[TransferSize];
90     while ((amountRead = openFile->Read(buffer, TransferSize)) > 0)
91         for (i = 0; i < amountRead; i++)
92             printf("%c", buffer[i]);
93     delete [] buffer;
94
95     delete openFile;            // close the Nachos file
96     return;
97 }
98
99 //----------------------------------------------------------------------
100 // PerformanceTest
101 //      Stress the Nachos file system by creating a large file, writing
102 //      it out a bit at a time, reading it back a bit at a time, and then
103 //      deleting the file.
```

```
104 //
105 //      Implemented as three separate routines:
106 //        FileWrite -- write the file
107 //        FileRead -- read the file
108 //        PerformanceTest -- overall control, and print out performance #'s
109 //----------------------------------------------------------------------
110
111 #define FileName        "TestFile"
112 #define Contents        "1234567890"
113 #define ContentSize (int)strlen(Contents)
114 #define FileSize        ((int)(ContentSize * 5000))
115
116 static void
117 FileWrite()
118 {
119     OpenFile *openFile;
120     int i, numBytes;
121
122     printf("Sequential write of %d byte file, in %d byte chunks\n",
123         FileSize, ContentSize);
124     if (!fileSystem->Create(FileName, 0)) {
125       printf("Perf test: can't create %s\n", FileName);
126       return;
127     }
128     openFile = fileSystem->Open(FileName);
129     if (openFile == NULL) {
130         printf("Perf test: unable to open %s\n", FileName);
131         return;
132     }
133     for (i = 0; i < FileSize; i += ContentSize) {
134         numBytes = openFile->Write(Contents, ContentSize);
135         if (numBytes < 10) {
136             printf("Perf test: unable to write %s\n", FileName);
137             delete openFile;
138             return;
139         }
140     }
141     delete openFile;    // close file
142 }
143
144 static void
145 FileRead()
146 {
147     OpenFile *openFile;
148     char *buffer = new char[ContentSize];
149     int i, numBytes;
150
151     printf("Sequential read of %d byte file, in %d byte chunks\n",
152         FileSize, ContentSize);
153
154     if ((openFile = fileSystem->Open(FileName)) == NULL) {
155         printf("Perf test: unable to open file %s\n", FileName);
156         delete [] buffer;
157         return;
158     }
159     for (i = 0; i < FileSize; i += ContentSize) {
160         numBytes = openFile->Read(buffer, ContentSize);
161         if ((numBytes < 10) || strncmp(buffer, Contents, ContentSize)) {
```

```
162             printf("Perf test: unable to read %s\n", FileName);
163             delete openFile;
164             delete [] buffer;
165             return;
166         }
167     }
168     delete [] buffer;
169     delete openFile;    // close file
170 }
171
172 void
173 PerformanceTest()
174 {
175     printf("Starting file system performance test:\n");
176     stats->Print();
177     FileWrite();
178     FileRead();
179     if (!fileSystem->Remove(FileName)) {
180       printf("Perf test: unable to remove %s\n", FileName);
181       return;
182     }
183     stats->Print();
184 }
185
```

2.8 openfile.h

```
  1 // openfile.h
  2 //      Data structures for opening, closing, reading and writing to
  3 //      individual files.  The operations supported are similar to
  4 //      the UNIX ones -- type 'man open' to the UNIX prompt.
  5 //
  6 //      There are two implementations.  One is a "STUB" that directly
  7 //      turns the file operations into the underlying UNIX operations.
  8 //      (cf. comment in filesys.h).
  9 //
 10 //      The other is the "real" implementation, that turns these
 11 //      operations into read and write disk sector requests.
 12 //      In this baseline implementation of the file system, we don't
 13 //      worry about concurrent accesses to the file system
 14 //      by different threads -- this is part of the assignment.
 15 //
 16 // Copyright (c) 1992-1993 The Regents of the University of California.
 17 // All rights reserved.  See copyright.h for copyright notice and limitation
 18 // of liability and disclaimer of warranty provisions.
 19
 20 #ifndef OPENFILE_H
 21 #define OPENFILE_H
 22
 23 #include "copyright.h"
 24 #include "utility.h"
 25
 26 #ifdef FILESYS_STUB                     // Temporarily implement calls to
 27                                         // Nachos file system as calls to UNIX!
 28                                         // See definitions listed under #else
 29 class OpenFile {
 30   public:
```

```
31     OpenFile(int f) { file = f; currentOffset = 0; }    // open the file
32     ~OpenFile() { Close(file); }                        // close the file
33
34     int ReadAt(char *into, int numBytes, int position) {
35                 Lseek(file, position, 0);
36                 return ReadPartial(file, into, numBytes);
37                 }
38     int WriteAt(char *from, int numBytes, int position) {
39                 Lseek(file, position, 0);
40                 WriteFile(file, from, numBytes);
41                 return numBytes;
42                 }
43     int Read(char *into, int numBytes) {
44                 int numRead = ReadAt(into, numBytes, currentOffset);
45                 currentOffset += numRead;
46                 return numRead;
47                 }
48     int Write(char *from, int numBytes) {
49                 int numWritten = WriteAt(from, numBytes, currentOffset);
50                 currentOffset += numWritten;
51                 return numWritten;
52                 }
53
54     int Length() { Lseek(file, 0, 2); return Tell(file); }
55
56   private:
57     int file;
58     int currentOffset;
59 };
60
61 #else // FILESYS
62 class FileHeader;
63
64 class OpenFile {
65   public:
66     OpenFile(int sector);               // Open a file whose header is located
67                                          // at "sector" on the disk
68     ~OpenFile();                        // Close the file
69
70     void Seek(int position);            // Set the position from which to
71                                         // start reading/writing -- UNIX lseek
72
73     int Read(char *into, int numBytes); // Read/write bytes from the file,
74                                         // starting at the implicit position.
75                                         // Return the # actually read/written,
76                                         // and increment position in file.
77     int Write(char *from, int numBytes);
78
79     int ReadAt(char *into, int numBytes, int position);
80                                         // Read/write bytes from the file,
81                                         // bypassing the implicit position.
82     int WriteAt(char *from, int numBytes, int position);
83
84     int Length();                       // Return the number of bytes in the
85                                         // file (this interface is simpler
86                                         // than the UNIX idiom -- lseek to
87                                         // end of file, tell, lseek back
88
```

```
89   private:
90     FileHeader *hdr;                    // Header for this file
91     int seekPosition;                   // Current position within the file
92 };
93
94 #endif // FILESYS
95
96 #endif // OPENFILE_H
```

2.9 openfile.cc

```
 1 // openfile.cc
 2 //      Routines to manage an open Nachos file.  As in UNIX, a
 3 //      file must be open before we can read or write to it.
 4 //      Once we're all done, we can close it (in Nachos, by deleting
 5 //      the OpenFile data structure).
 6 //
 7 //      Also as in UNIX, for convenience, we keep the file header in
 8 //      memory while the file is open.
 9 //
10 // Copyright (c) 1992-1993 The Regents of the University of California.
11 // All rights reserved.  See copyright.h for copyright notice and limitation
12 // of liability and disclaimer of warranty provisions.
13
14 #include "copyright.h"
15 #include "filehdr.h"
16 #include "openfile.h"
17 #include "system.h"
18
19 //----------------------------------------------------------------------
20 // OpenFile::OpenFile
21 //      Open a Nachos file for reading and writing.  Bring the file header
22 //      into memory while the file is open.
23 //
24 //      "sector" -- the location on disk of the file header for this file
25 //----------------------------------------------------------------------
26
27 OpenFile::OpenFile(int sector)
28 {
29     hdr = new FileHeader;
30     hdr->FetchFrom(sector);
31     seekPosition = 0;
32 }
33
34 //----------------------------------------------------------------------
35 // OpenFile::~OpenFile
36 //      Close a Nachos file, de-allocating any in-memory data structures.
37 //----------------------------------------------------------------------
38
39 OpenFile::~OpenFile()
40 {
41     delete hdr;
42 }
43
44 //----------------------------------------------------------------------
45 // OpenFile::Seek
46 //      Change the current location within the open file -- the point at
```

```
47 //      which the next Read or Write will start from.
48 //
49 //      "position" -- the location within the file for the next Read/Write
50 //----------------------------------------------------------------------
51
52 void
53 OpenFile::Seek(int position)
54 {
55     seekPosition = position;
56 }
57
58 //----------------------------------------------------------------------
59 // OpenFile::Read/Write
60 //      Read/write a portion of a file, starting from seekPosition.
61 //      Return the number of bytes actually written or read, and as a
62 //      side effect, increment the current position within the file.
63 //
64 //      Implemented using the more primitive ReadAt/WriteAt.
65 //
66 //      "into" -- the buffer to contain the data to be read from disk
67 //       "from" -- the buffer containing the data to be written to disk
68 //      "numBytes" -- the number of bytes to transfer
69 //----------------------------------------------------------------------
70
71 int
72 OpenFile::Read(char *into, int numBytes)
73 {
74    int result = ReadAt(into, numBytes, seekPosition);
75    seekPosition += result;
76    return result;
77 }
78
79 int
80 OpenFile::Write(char *into, int numBytes)
81 {
82    int result = WriteAt(into, numBytes, seekPosition);
83    seekPosition += result;
84    return result;
85 }
86
87 //----------------------------------------------------------------------
88 // OpenFile::ReadAt/WriteAt
89 //      Read/write a portion of a file, starting at "position".
90 //      Return the number of bytes actually written or read, but has
91 //      no side effects (except that Write modifies the file, of course).
92 //
93 //      There is no guarantee the request starts or ends on an even disk sector
94 //      boundary; however the disk only knows how to read/write a whole disk
95 //      sector at a time.  Thus:
96 //
97 //      For ReadAt:
98 //         We read in all of the full or partial sectors that are part of the
99 //         request, but we only copy the part we are interested in.
100 //      For WriteAt:
101 //         We must first read in any sectors that will be partially written,
102 //         so that we don't overwrite the unmodified portion.  We then copy
103 //         in the data that will be modified, and write back all the full
104 //         or partial sectors that are part of the request.
```

```
105 //
106 //      "into" -- the buffer to contain the data to be read from disk
107 //      "from" -- the buffer containing the data to be written to disk
108 //      "numBytes" -- the number of bytes to transfer
109 //      "position" -- the offset within the file of the first byte to be
110 //                      read/written
111 //----------------------------------------------------------------------
112
113 int
114 OpenFile::ReadAt(char *into, int numBytes, int position)
115 {
116     int fileLength = hdr->FileLength();
117     int i, firstSector, lastSector, numSectors;
118     char *buf;
119
120     if ((numBytes <= 0) || (position >= fileLength))
121         return 0;                               // check request
122     if ((position + numBytes) > fileLength)
123         numBytes = fileLength - position;
124     DEBUG('f', "Reading %d bytes at %d, from file of length %d.\n",
125                         numBytes, position, fileLength);
126
127     firstSector = divRoundDown(position, SectorSize);
128     lastSector = divRoundDown(position + numBytes - 1, SectorSize);
129     numSectors = 1 + lastSector - firstSector;
130
131     // read in all the full and partial sectors that we need
132     buf = new char[numSectors * SectorSize];
133     for (i = firstSector; i <= lastSector; i++)
134         synchDisk->ReadSector(hdr->ByteToSector(i * SectorSize),
135                                       &buf[(i - firstSector) * SectorSize]);
136
137     // copy the part we want
138     bcopy(&buf[position - (firstSector * SectorSize)], into, numBytes);
139     delete [] buf;
140     return numBytes;
141 }
142
143 int
144 OpenFile::WriteAt(char *from, int numBytes, int position)
145 {
146     int fileLength = hdr->FileLength();
147     int i, firstSector, lastSector, numSectors;
148     bool firstAligned, lastAligned;
149     char *buf;
150
151     if ((numBytes <= 0) || (position >= fileLength))
152         return 0;                               // check request
153     if ((position + numBytes) > fileLength)
154         numBytes = fileLength - position;
155     DEBUG('f', "Writing %d bytes at %d, from file of length %d.\n",
156                         numBytes, position, fileLength);
157
158     firstSector = divRoundDown(position, SectorSize);
159     lastSector = divRoundDown(position + numBytes - 1, SectorSize);
160     numSectors = 1 + lastSector - firstSector;
161
162     buf = new char[numSectors * SectorSize];
```

```
163
164     firstAligned = (bool)(position == (firstSector * SectorSize));
165     lastAligned = (bool)((position + numBytes) == ((lastSector + 1) * SectorSize));
166
167 // read in first and last sector, if they are to be partially modified
168     if (!firstAligned)
169         ReadAt(buf, SectorSize, firstSector * SectorSize);
170     if (!lastAligned && ((firstSector != lastSector) || firstAligned))
171         ReadAt(&buf[(lastSector - firstSector) * SectorSize],
172                                 SectorSize, lastSector * SectorSize);
173
174 // copy in the bytes we want to change
175     bcopy(from, &buf[position - (firstSector * SectorSize)], numBytes);
176
177 // write modified sectors back
178     for (i = firstSector; i <= lastSector; i++)
179         synchDisk->WriteSector(hdr->ByteToSector(i * SectorSize),
180                                         &buf[(i - firstSector) * SectorSize]);
181     delete [] buf;
182     return numBytes;
183 }
184
185 //----------------------------------------------------------------------
186 // OpenFile::Length
187 //      Return the number of bytes in the file.
188 //----------------------------------------------------------------------
189
190 int
191 OpenFile::Length()
192 {
193     return hdr->FileLength();
194 }
```

2.10 synchdisk.h

```
 1 // synchdisk.h
 2 //      Data structures to export a synchronous interface to the raw
 3 //      disk device.
 4 //
 5 // Copyright (c) 1992-1993 The Regents of the University of California.
 6 // All rights reserved.  See copyright.h for copyright notice and limitation
 7 // of liability and disclaimer of warranty provisions.
 8
 9 #include "copyright.h"
10
11 #ifndef SYNCHDISK_H
12 #define SYNCHDISK_H
13
14 #include "disk.h"
15 #include "synch.h"
16
17 // The following class defines a "synchronous" disk abstraction.
18 // As with other I/O devices, the raw physical disk is an asynchronous device --
19 // requests to read or write portions of the disk return immediately,
20 // and an interrupt occurs later to signal that the operation completed.
21 // (Also, the physical characteristics of the disk device assume that
22 // only one operation can be requested at a time).
```

```
23 //
24 // This class provides the abstraction that for any individual thread
25 // making a request, it waits around until the operation finishes before
26 // returning.
27 class SynchDisk {
28   public:
29     SynchDisk(char* name);              // Initialize a synchronous disk,
30                                         // by initializing the raw Disk.
31     ~SynchDisk();                       // De-allocate the synch disk data
32
33     void ReadSector(int sectorNumber, char* data);
34                                         // Read/write a disk sector, returning
35                                         // only once the data is actually read
36                                         // or written.  These call
37                                         // Disk::ReadRequest/WriteRequest and
38                                         // then wait until the request is done.
39     void WriteSector(int sectorNumber, char* data);
40
41     void RequestDone();                 // Called by the disk device interrupt
42                                         // handler, to signal that the
43                                         // current disk operation is complete.
44
45   private:
46     Disk *disk;                         // Raw disk device
47     Semaphore *semaphore;               // To synchronize requesting thread
48                                         // with the interrupt handler
49     Lock *lock;                         // Only one read/write request
50                                         // can be sent to the disk at a time
51 };
52
53 #endif // SYNCHDISK_H
```

2.11 synchdisk.h

```
 1 // synchdisk.h
 2 //      Data structures to export a synchronous interface to the raw
 3 //      disk device.
 4 //
 5 // Copyright (c) 1992-1993 The Regents of the University of California.
 6 // All rights reserved.  See copyright.h for copyright notice and limitation
 7 // of liability and disclaimer of warranty provisions.
 8
 9 #include "copyright.h"
10
11 #ifndef SYNCHDISK_H
12 #define SYNCHDISK_H
13
14 #include "disk.h"
15 #include "synch.h"
16
17 // The following class defines a "synchronous" disk abstraction.
18 // As with other I/O devices, the raw physical disk is an asynchronous device --
19 // requests to read or write portions of the disk return immediately,
20 // and an interrupt occurs later to signal that the operation completed.
21 // (Also, the physical characteristics of the disk device assume that
22 // only one operation can be requested at a time).
23 //
```

```
24 // This class provides the abstraction that for any individual thread
25 // making a request, it waits around until the operation finishes before
26 // returning.
27 class SynchDisk {
28   public:
29     SynchDisk(char* name);              // Initialize a synchronous disk,
30                                         // by initializing the raw Disk.
31     ~SynchDisk();                       // De-allocate the synch disk data
32
33     void ReadSector(int sectorNumber, char* data);
34                                         // Read/write a disk sector, returning
35                                         // only once the data is actually read
36                                         // or written.  These call
37                                         // Disk::ReadRequest/WriteRequest and
38                                         // then wait until the request is done.
39     void WriteSector(int sectorNumber, char* data);
40
41     void RequestDone();                 // Called by the disk device interrupt
42                                         // handler, to signal that the
43                                         // current disk operation is complete.
44
45   private:
46     Disk *disk;                         // Raw disk device
47     Semaphore *semaphore;               // To synchronize requesting thread
48                                         // with the interrupt handler
49     Lock *lock;                         // Only one read/write request
50                                         // can be sent to the disk at a time
51 };
52
53 #endif // SYNCHDISK_H
```

Chapter 3

Directory ../machine/

Contents

3.1 console.h . . . 81
3.2 console.cc . . . 83
3.3 disk.h . . . 85
3.4 disk.cc . . . 87
3.5 interrupt.h . . . 92
3.6 interrupt.cc . . . 94
3.7 machine.h . . . 101
3.8 machine.cc . . . 104
3.9 mipssim.h . . . 108
3.10 mipssim.cc . . . 112
3.11 network.h . . . 124
3.12 network.cc . . . 127
3.13 stats.h . . . 129
3.14 stats.cc . . . 131
3.15 sysdep.h . . . 131
3.16 sysdep.cc . . . 133
3.17 timer.h . . . 142
3.18 timer.cc . . . 143
3.19 translate.h . . . 144
3.20 translate.cc . . . 145

This chapter lists all the source codes found in directory `../machine/`. They are:

```
console.cc     interrupt.h    network.cc     sysdep.h
console.h      machine.cc     network.h      timer.cc
disk.cc        machine.h      stats.cc       timer.h
disk.h         mipssim.cc     stats.h        translate.cc
interrupt.cc   mipssim.h      sysdep.cc      translate.h
```

3.1 console.h

```
1 // console.h
2 //      Data structures to simulate the behavior of a terminal
3 //      I/O device.  A terminal has two parts -- a keyboard input,
4 //      and a display output, each of which produces/accepts
5 //      characters sequentially.
```

```
 6 //
 7 //      The console hardware device is asynchronous.  When a character is
 8 //      written to the device, the routine returns immediately, and an
 9 //      interrupt handler is called later when the I/O completes.
10 //      For reads, an interrupt handler is called when a character arrives.
11 //
12 //      The user of the device can specify the routines to be called when
13 //      the read/write interrupts occur.  There is a separate interrupt
14 //      for read and write, and the device is "duplex" -- a character
15 //      can be outgoing and incoming at the same time.
16 //
17 //  DO NOT CHANGE -- part of the machine emulation
18 //
19 // Copyright (c) 1992-1993 The Regents of the University of California.
20 // All rights reserved.  See copyright.h for copyright notice and limitation
21 // of liability and disclaimer of warranty provisions.
22
23 #ifndef CONSOLE_H
24 #define CONSOLE_H
25
26 #include "copyright.h"
27 #include "utility.h"
28
29 // The following class defines a hardware console device.
30 // Input and output to the device is simulated by reading
31 // and writing to UNIX files ("readFile" and "writeFile").
32 //
33 // Since the device is asynchronous, the interrupt handler "readAvail"
34 // is called when a character has arrived, ready to be read in.
35 // The interrupt handler "writeDone" is called when an output character
36 // has been "put", so that the next character can be written.
37
38 class Console {
39   public:
40     Console(char *readFile, char *writeFile, VoidFunctionPtr readAvail,
41         VoidFunctionPtr writeDone, _int callArg);
42                                 // initialize the hardware console device
43     ~Console();                 // clean up console emulation
44
45 // external interface -- Nachos kernel code can call these
46     void PutChar(char ch);      // Write "ch" to the console display,
47                                 // and return immediately.  "writeHandler"
48                                 // is called when the I/O completes.
49
50     char GetChar();             // Poll the console input.  If a char is
51                                 // available, return it.  Otherwise, return EOF.
52                                 // "readHandler" is called whenever there is
53                                 // a char to be gotten
54
55 // internal emulation routines -- DO NOT call these.
56     void WriteDone();           // internal routines to signal I/O completion
57     void CheckCharAvail();
58
59   private:
60     int readFileNo;                     // UNIX file emulating the keyboard
61     int writeFileNo;                    // UNIX file emulating the display
62     VoidFunctionPtr writeHandler;       // Interrupt handler to call when
63                                         // the PutChar I/O completes
```

```
64     VoidFunctionPtr readHandler;        // Interrupt handler to call when
65                                         // a character arrives from the keyboard
66     _int handlerArg;                    // argument to be passed to the
67                                          // interrupt handlers
68     bool putBusy;                       // Is a PutChar operation in progress?
69                                         // If so, you can't do another one!
70     char incoming;                      // Contains the character to be read,
71                                         // if there is one available.
72                                         // Otherwise contains EOF.
73 };
74
75 #endif // CONSOLE_H
```

3.2 console.cc

```
 1 // console.cc
 2 //      Routines to simulate a serial port to a console device.
 3 //      A console has input (a keyboard) and output (a display).
 4 //      These are each simulated by operations on UNIX files.
 5 //      The simulated device is asynchronous,
 6 //      so we have to invoke the interrupt handler (after a simulated
 7 //      delay), to signal that a byte has arrived and/or that a written
 8 //      byte has departed.
 9 //
10 //  DO NOT CHANGE -- part of the machine emulation
11 //
12 // Copyright (c) 1992-1993 The Regents of the University of California.
13 // All rights reserved.  See copyright.h for copyright notice and limitation
14 // of liability and disclaimer of warranty provisions.
15
16 #include "copyright.h"
17 #include "console.h"
18 #include "system.h"
19
20 // Dummy functions because C++ is weird about pointers to member functions
21 static void ConsoleReadPoll(_int c)
22 { Console *console = (Console *)c; console->CheckCharAvail(); }
23 static void ConsoleWriteDone(_int c)
24 { Console *console = (Console *)c; console->WriteDone(); }
25
26 //----------------------------------------------------------------------
27 // Console::Console
28 //      Initialize the simulation of a hardware console device.
29 //
30 //      "readFile" -- UNIX file simulating the keyboard (NULL -> use stdin)
31 //      "writeFile" -- UNIX file simulating the display (NULL -> use stdout)
32 //      "readAvail" is the interrupt handler called when a character arrives
33 //              from the keyboard
34 //      "writeDone" is the interrupt handler called when a character has
35 //              been output, so that it is ok to request the next char be
36 //              output
37 //----------------------------------------------------------------------
38
39 Console::Console(char *readFile, char *writeFile, VoidFunctionPtr readAvail,
40                 VoidFunctionPtr writeDone, _int callArg)
41 {
42     if (readFile == NULL)
```

```
43          readFileNo = 0;                                 // keyboard = stdin
44      else
45          readFileNo = OpenForReadWrite(readFile, TRUE);  // should be read-only
46      if (writeFile == NULL)
47          writeFileNo = 1;                                // display = stdout
48      else
49          writeFileNo = OpenForWrite(writeFile);
50
51      // set up the stuff to emulate asynchronous interrupts
52      writeHandler = writeDone;
53      readHandler = readAvail;
54      handlerArg = callArg;
55      putBusy = FALSE;
56      incoming = EOF;
57
58      // start polling for incoming packets
59      interrupt->Schedule(ConsoleReadPoll, (_int)this, ConsoleTime, ConsoleReadInt);
60  }
61
62  //----------------------------------------------------------------------
63  // Console::~Console
64  //      Clean up console emulation
65  //----------------------------------------------------------------------
66
67  Console::~Console()
68  {
69      if (readFileNo != 0)
70          Close(readFileNo);
71      if (writeFileNo != 1)
72          Close(writeFileNo);
73  }
74
75  //----------------------------------------------------------------------
76  // Console::CheckCharAvail()
77  //      Periodically called to check if a character is available for
78  //      input from the simulated keyboard (eg, has it been typed?).
79  //
80  //      Only read it in if there is buffer space for it (if the previous
81  //      character has been grabbed out of the buffer by the Nachos kernel).
82  //      Invoke the "read" interrupt handler, once the character has been
83  //      put into the buffer.
84  //----------------------------------------------------------------------
85
86  void
87  Console::CheckCharAvail()
88  {
89      char c;
90
91      // schedule the next time to poll for a packet
92      interrupt->Schedule(ConsoleReadPoll, (_int)this, ConsoleTime,
93                          ConsoleReadInt);
94
95      // do nothing if character is already buffered, or none to be read
96      if ((incoming != EOF) || !PollFile(readFileNo))
97          return;
98
99      // otherwise, read character and tell user about it
100     Read(readFileNo, &c, sizeof(char));
```

```
101     incoming = c ;
102     stats->numConsoleCharsRead++;
103     (*readHandler)(handlerArg);
104 }
105
106 //----------------------------------------------------------------------
107 // Console::WriteDone()
108 //      Internal routine called when it is time to invoke the interrupt
109 //      handler to tell the Nachos kernel that the output character has
110 //      completed.
111 //----------------------------------------------------------------------
112
113 void
114 Console::WriteDone()
115 {
116     putBusy = FALSE;
117     stats->numConsoleCharsWritten++;
118     (*writeHandler)(handlerArg);
119 }
120
121 //----------------------------------------------------------------------
122 // Console::GetChar()
123 //      Read a character from the input buffer, if there is any there.
124 //      Either return the character, or EOF if none buffered.
125 //----------------------------------------------------------------------
126
127 char
128 Console::GetChar()
129 {
130    char ch = incoming;
131
132    incoming = EOF;
133    return ch;
134 }
135
136 //----------------------------------------------------------------------
137 // Console::PutChar()
138 //      Write a character to the simulated display, schedule an interrupt
139 //      to occur in the future, and return.
140 //----------------------------------------------------------------------
141
142 void
143 Console::PutChar(char ch)
144 {
145     ASSERT(putBusy == FALSE);
146     WriteFile(writeFileNo, &ch, sizeof(char));
147     putBusy = TRUE;
148     interrupt->Schedule(ConsoleWriteDone, (_int)this, ConsoleTime,
149                                         ConsoleWriteInt);
150 }
```

3.3 disk.h

```
1 // disk.h
2 //      Data structures to emulate a physical disk.  A physical disk
3 //      can accept (one at a time) requests to read/write a disk sector;
4 //      when the request is satisfied, the CPU gets an interrupt, and
```

```
//      the next request can be sent to the disk.
//
//      Disk contents are preserved across machine crashes, but if
//      a file system operation (eg, create a file) is in progress when the
//      system shuts down, the file system may be corrupted.
//
//  DO NOT CHANGE -- part of the machine emulation
//
// Copyright (c) 1992-1993 The Regents of the University of California.
// All rights reserved.  See copyright.h for copyright notice and limitation
// of liability and disclaimer of warranty provisions.

#ifndef DISK_H
#define DISK_H

#include "copyright.h"
#include "utility.h"

// The following class defines a physical disk I/O device.  The disk
// has a single surface, split up into "tracks", and each track split
// up into "sectors" (the same number of sectors on each track, and each
// sector has the same number of bytes of storage).
//
// Addressing is by sector number -- each sector on the disk is given
// a unique number: track * SectorsPerTrack + offset within a track.
//
// As with other I/O devices, the raw physical disk is an asynchronous device --
// requests to read or write portions of the disk return immediately,
// and an interrupt is invoked later to signal that the operation completed.
//
// The physical disk is in fact simulated via operations on a UNIX file.
//
// To make life a little more realistic, the simulated time for
// each operation reflects a "track buffer" -- RAM to store the contents
// of the current track as the disk head passes by.  The idea is that the
// disk always transfers to the track buffer, in case that data is requested
// later on.  This has the benefit of eliminating the need for
// "skip-sector" scheduling -- a read request which comes in shortly after
// the head has passed the beginning of the sector can be satisfied more
// quickly, because its contents are in the track buffer.  Most
// disks these days now come with a track buffer.
//
// The track buffer simulation can be disabled by compiling with -DNOTRACKBUF

#define SectorSize              128     // number of bytes per disk sector
#define SectorsPerTrack         32      // number of sectors per disk track
#define NumTracks               32      // number of tracks per disk
#define NumSectors              (SectorsPerTrack * NumTracks)
                                        // total # of sectors per disk

class Disk {
  public:
    Disk(char* name, VoidFunctionPtr callWhenDone, _int callArg);
                                        // Create a simulated disk.
                                        // Invoke (*callWhenDone)(callArg)
                                        // every time a request completes.
    ~Disk();                            // Deallocate the disk.

```

```
63     void ReadRequest(int sectorNumber, char* data);
64                                       // Read/write an single disk sector.
65                                       // These routines send a request to
66                                       // the disk and return immediately.
67                                        // Only one request allowed at a time!
68     void WriteRequest(int sectorNumber, char* data);
69
70     void HandleInterrupt();            // Interrupt handler, invoked when
71                                       // disk request finishes.
72
73     int ComputeLatency(int newSector, bool writing);
74                                       // Return how long a request to
75                                       // newSector will take:
76                                       // (seek + rotational delay + transfer)
77
78   private:
79     int fileno;                       // UNIX file number for simulated disk
80     VoidFunctionPtr handler;          // Interrupt handler, to be invoked
81                                       // when any disk request finishes
82     _int handlerArg;                  // Argument to interrupt handler
83     bool active;                      // Is a disk operation in progress?
84     int lastSector;                   // The previous disk request
85     int bufferInit;                   // When the track buffer started
86                                       // being loaded
87
88     int TimeToSeek(int newSector, int *rotate); // time to get to the new track
89     int ModuloDiff(int to, int from);        // # sectors between to and from
90     void UpdateLast(int newSector);
91 };
92
93 #endif // DISK_H
```

3.4 disk.cc

```
 1 // disk.cc
 2 //      Routines to simulate a physical disk device; reading and writing
 3 //      to the disk is simulated as reading and writing to a UNIX file.
 4 //      See disk.h for details about the behavior of disks (and
 5 //      therefore about the behavior of this simulation).
 6 //
 7 //      Disk operations are asynchronous, so we have to invoke an interrupt
 8 //      handler when the simulated operation completes.
 9 //
10 //  DO NOT CHANGE -- part of the machine emulation
11 //
12 // Copyright (c) 1992-1993 The Regents of the University of California.
13 // All rights reserved.  See copyright.h for copyright notice and limitation
14 // of liability and disclaimer of warranty provisions.
15
16 #include "copyright.h"
17 #include "disk.h"
18 #include "system.h"
19
20 // We put this at the front of the UNIX file representing the
21 // disk, to make it less likely we will accidentally treat a useful file
22 // as a disk (which would probably trash the file's contents).
23 #define MagicNumber     0x456789ab
```

```
24 #define MagicSize       sizeof(int)
25
26 #define DiskSize        (MagicSize + (NumSectors * SectorSize))
27
28 // dummy procedure because we can't take a pointer of a member function
29 static void DiskDone(_int arg) { ((Disk *)arg)->HandleInterrupt(); }
30
31 //----------------------------------------------------------------------
32 // Disk::Disk()
33 //      Initialize a simulated disk.  Open the UNIX file (creating it
34 //      if it doesn't exist), and check the magic number to make sure it's
35 //      ok to treat it as Nachos disk storage.
36 //
37 //      "name" -- text name of the file simulating the Nachos disk
38 //      "callWhenDone" -- interrupt handler to be called when disk read/write
39 //         request completes
40 //      "callArg" -- argument to pass the interrupt handler
41 //----------------------------------------------------------------------
42
43 Disk::Disk(char* name, VoidFunctionPtr callWhenDone, _int callArg)
44 {
45     int magicNum;
46     int tmp = 0;
47
48     DEBUG('d', "Initializing the disk, 0x%x 0x%x\n", callWhenDone, callArg);
49     handler = callWhenDone;
50     handlerArg = callArg;
51     lastSector = 0;
52     bufferInit = 0;
53
54     fileno = OpenForReadWrite(name, FALSE);
55     if (fileno >= 0) {                  // file exists, check magic number
56         Read(fileno, (char *) &magicNum, MagicSize);
57         ASSERT(magicNum == MagicNumber);
58     } else {                            // file doesn't exist, create it
59         fileno = OpenForWrite(name);
60         magicNum = MagicNumber;
61         WriteFile(fileno, (char *) &magicNum, MagicSize); // write magic number
62
63         // need to write at end of file, so that reads will not return EOF
64         Lseek(fileno, DiskSize - sizeof(int), 0);
65         WriteFile(fileno, (char *)&tmp, sizeof(int));
66     }
67     active = FALSE;
68 }
69
70 //----------------------------------------------------------------------
71 // Disk::~Disk()
72 //      Clean up disk simulation, by closing the UNIX file representing the
73 //      disk.
74 //----------------------------------------------------------------------
75
76 Disk::~Disk()
77 {
78     Close(fileno);
79 }
80
81 //----------------------------------------------------------------------
```

```
82 // Disk::PrintSector()
83 //      Dump the data in a disk read/write request, for debugging.
84 //----------------------------------------------------------------------
85
86 static void
87 PrintSector (bool writing, int sector, char *data)
88 {
89     int *p = (int *) data;
90
91     if (writing)
92         printf("Writing sector: %d\n", sector);
93     else
94         printf("Reading sector: %d\n", sector);
95     for (unsigned int i = 0; i < (SectorSize/sizeof(int)); i++)
96         printf("%x ", p[i]);
97     printf("\n");
98 }
99
100 //----------------------------------------------------------------------
101 // Disk::ReadRequest/WriteRequest
102 //      Simulate a request to read/write a single disk sector
103 //         Do the read/write immediately to the UNIX file
104 //         Set up an interrupt handler to be called later,
105 //            that will notify the caller when the simulator says
106 //            the operation has completed.
107 //
108 //      Note that a disk only allows an entire sector to be read/written,
109 //      not part of a sector.
110 //
111 //      "sectorNumber" -- the disk sector to read/write
112 //      "data" -- the bytes to be written, the buffer to hold the incoming bytes
113 //----------------------------------------------------------------------
114
115 void
116 Disk::ReadRequest(int sectorNumber, char* data)
117 {
118     int ticks = ComputeLatency(sectorNumber, FALSE);
119
120     ASSERT(!active);                            // only one request at a time
121     ASSERT((sectorNumber >= 0) && (sectorNumber < NumSectors));
122
123     DEBUG('d', "Reading from sector %d\n", sectorNumber);
124     Lseek(fileno, SectorSize * sectorNumber + MagicSize, 0);
125     Read(fileno, data, SectorSize);
126     if (DebugIsEnabled('d'))
127         PrintSector(FALSE, sectorNumber, data);
128
129     active = TRUE;
130     UpdateLast(sectorNumber);
131     stats->numDiskReads++;
132     interrupt->Schedule(DiskDone, (_int) this, ticks, DiskInt);
133 }
134
135 void
136 Disk::WriteRequest(int sectorNumber, char* data)
137 {
138     int ticks = ComputeLatency(sectorNumber, TRUE);
139
```

```
140     ASSERT(!active);
141     ASSERT((sectorNumber >= 0) && (sectorNumber < NumSectors));
142
143     DEBUG('d', "Writing to sector %d\n", sectorNumber);
144     Lseek(fileno, SectorSize * sectorNumber + MagicSize, 0);
145     WriteFile(fileno, data, SectorSize);
146     if (DebugIsEnabled('d'))
147         PrintSector(TRUE, sectorNumber, data);
148
149     active = TRUE;
150     UpdateLast(sectorNumber);
151     stats->numDiskWrites++;
152     interrupt->Schedule(DiskDone, (_int) this, ticks, DiskInt);
153 }
154
155 //----------------------------------------------------------------------
156 // Disk::HandleInterrupt()
157 //      Called when it is time to invoke the disk interrupt handler,
158 //      to tell the Nachos kernel that the disk request is done.
159 //----------------------------------------------------------------------
160
161 void
162 Disk::HandleInterrupt ()
163 {
164     active = FALSE;
165     (*handler)(handlerArg);
166 }
167
168 //----------------------------------------------------------------------
169 // Disk::TimeToSeek()
170 //      Returns how long it will take to position the disk head over the correct
171 //      track on the disk.  Since when we finish seeking, we are likely
172 //      to be in the middle of a sector that is rotating past the head,
173 //      we also return how long until the head is at the next sector boundary.
174 //
175 //      Disk seeks at one track per SeekTime ticks (cf. stats.h)
176 //      and rotates at one sector per RotationTime ticks
177 //----------------------------------------------------------------------
178
179 int
180 Disk::TimeToSeek(int newSector, int *rotation)
181 {
182     int newTrack = newSector / SectorsPerTrack;
183     int oldTrack = lastSector / SectorsPerTrack;
184     int seek = abs(newTrack - oldTrack) * SeekTime;
185                                 // how long will seek take?
186     int over = (stats->totalTicks + seek) % RotationTime;
187                                 // will we be in the middle of a sector when
188                                 // we finish the seek?
189
190     *rotation = 0;
191     if (over > 0)               // if so, need to round up to next full sector
192         *rotation = RotationTime - over;
193     return seek;
194 }
195
196 //----------------------------------------------------------------------
197 // Disk::ModuloDiff()
```

```
198 //      Return number of sectors of rotational delay between target sector
199 //       "to" and current sector position "from"
200 //----------------------------------------------------------------------
201
202 int
203 Disk::ModuloDiff(int to, int from)
204 {
205     int toOffset = to % SectorsPerTrack;
206     int fromOffset = from % SectorsPerTrack;
207
208     return ((toOffset - fromOffset) + SectorsPerTrack) % SectorsPerTrack;
209 }
210
211 //----------------------------------------------------------------------
212 // Disk::ComputeLatency()
213 //      Return how long will it take to read/write a disk sector, from
214 //      the current position of the disk head.
215 //
216 //      Latency = seek time + rotational latency + transfer time
217 //      Disk seeks at one track per SeekTime ticks (cf. stats.h)
218 //      and rotates at one sector per RotationTime ticks
219 //
220 //      To find the rotational latency, we first must figure out where the
221 //      disk head will be after the seek (if any).  We then figure out
222 //      how long it will take to rotate completely past newSector after
223 //      that point.
224 //
225 //      The disk also has a "track buffer"; the disk continuously reads
226 //      the contents of the current disk track into the buffer.  This allows
227 //      read requests to the current track to be satisfied more quickly.
228 //      The contents of the track buffer are discarded after every seek to
229 //      a new track.
230 //----------------------------------------------------------------------
231
232 int
233 Disk::ComputeLatency(int newSector, bool writing)
234 {
235     int rotation;
236     int seek = TimeToSeek(newSector, &rotation);
237     int timeAfter = stats->totalTicks + seek + rotation;
238
239 #ifndef NOTRACKBUF      // turn this on if you don't want the track buffer stuff
240     // check if track buffer applies
241     if ((writing == FALSE) && (seek == 0)
242                 && (((timeAfter - bufferInit) / RotationTime)
243                         > ModuloDiff(newSector, bufferInit / RotationTime))) {
244         DEBUG('d', "Request latency = %d\n", RotationTime);
245         return RotationTime; // time to transfer sector from the track buffer
246     }
247 #endif
248
249     rotation += ModuloDiff(newSector, timeAfter / RotationTime) * RotationTime;
250
251     DEBUG('d', "Request latency = %d\n", seek + rotation + RotationTime);
252     return(seek + rotation + RotationTime);
253 }
254
255 //----------------------------------------------------------------------
```

```
256 // Disk::UpdateLast
257 //      Keep track of the most recently requested sector.  So we can know
258 //      what is in the track buffer.
259 //----------------------------------------------------------------------
260
261 void
262 Disk::UpdateLast(int newSector)
263 {
264     int rotate;
265     int seek = TimeToSeek(newSector, &rotate);
266
267     if (seek != 0)
268         bufferInit = stats->totalTicks + seek + rotate;
269     lastSector = newSector;
270     DEBUG('d', "Updating last sector = %d, %d\n", lastSector, bufferInit);
271 }
```

3.5 interrupt.h

```
 1 // interrupt.h
 2 //      Data structures to emulate low-level interrupt hardware.
 3 //
 4 //      The hardware provides a routine (SetLevel) to enable or disable
 5 //      interrupts.
 6 //
 7 //      In order to emulate the hardware, we need to keep track of all
 8 //      interrupts the hardware devices would cause, and when they
 9 //      are supposed to occur.
10 //
11 //      This module also keeps track of simulated time.  Time advances
12 //      only when the following occur:
13 //              interrupts are re-enabled
14 //              a user instruction is executed
15 //              there is nothing in the ready queue
16 //
17 //      As a result, unlike real hardware, interrupts (and thus time-slice
18 //      context switches) cannot occur anywhere in the code where interrupts
19 //      are enabled, but rather only at those places in the code where
20 //      simulated time advances (so that it becomes time to invoke an
21 //      interrupt in the hardware simulation).
22 //
23 //      NOTE: this means that incorrectly synchronized code may work
24 //      fine on this hardware simulation (even with randomized time slices),
25 //      but it wouldn't work on real hardware.  (Just because we can't
26 //      always detect when your program would fail in real life, does not
27 //      mean it's ok to write incorrectly synchronized code!)
28 //
29 //  DO NOT CHANGE -- part of the machine emulation
30 //
31 // Copyright (c) 1992-1993 The Regents of the University of California.
32 // All rights reserved.  See copyright.h for copyright notice and limitation
33 // of liability and disclaimer of warranty provisions.
34
35 #ifndef INTERRUPT_H
36 #define INTERRUPT_H
37
38 #include "copyright.h"
```

```
39 #include "list.h"
40
41 // Interrupts can be disabled (IntOff) or enabled (IntOn)
42 enum IntStatus { IntOff, IntOn };
43
44 // Nachos can be running kernel code (SystemMode), user code (UserMode),
45 // or there can be no runnable thread, because the ready list
46 // is empty (IdleMode).
47 enum MachineStatus {IdleMode, SystemMode, UserMode};
48
49 // IntType records which hardware device generated an interrupt.
50 // In Nachos, we support a hardware timer device, a disk, a console
51 // display and keyboard, and a network.
52 enum IntType { TimerInt, DiskInt, ConsoleWriteInt, ConsoleReadInt,
53                                NetworkSendInt, NetworkRecvInt};
54
55 // The following class defines an interrupt that is scheduled
56 // to occur in the future.  The internal data structures are
57 // left public to make it simpler to manipulate.
58
59 class PendingInterrupt {
60   public:
61     PendingInterrupt(VoidFunctionPtr func, _int param, int time, IntType kind);
62                                // initialize an interrupt that will
63                                // occur in the future
64
65     VoidFunctionPtr handler;    // The function (in the hardware device
66                                // emulator) to call when the interrupt occurs
67     _int arg;           // The argument to the function.
68     int when;                  // When the interrupt is supposed to fire
69     IntType type;              // for debugging
70 };
71
72 // The following class defines the data structures for the simulation
73 // of hardware interrupts.  We record whether interrupts are enabled
74 // or disabled, and any hardware interrupts that are scheduled to occur
75 // in the future.
76
77 class Interrupt {
78   public:
79     Interrupt();                       // initialize the interrupt simulation
80     ~Interrupt();                      // de-allocate data structures
81
82     IntStatus SetLevel(IntStatus level);// Disable or enable interrupts
83                                        // and return previous setting.
84
85     void Enable();                     // Enable interrupts.
86     IntStatus getLevel() {return level;}// Return whether interrupts
87                                        // are enabled or disabled
88
89     void Idle();                       // The ready queue is empty, roll
90                                        // simulated time forward until the
91                                        // next interrupt
92
93     void Halt();                       // quit and print out stats
94
95     void YieldOnReturn();              // cause a context switch on return
96                                        // from an interrupt handler
```

```
 97
 98     MachineStatus getStatus() { return status; } // idle, kernel, user
 99     void setStatus(MachineStatus st) { status = st; }
100
101     void DumpState();                   // Print interrupt state
102
103
104     // NOTE: the following are internal to the hardware simulation code.
105     // DO NOT call these directly.  I should make them "private",
106     // but they need to be public since they are called by the
107     // hardware device simulators.
108
109     void Schedule(VoidFunctionPtr handler,// Schedule an interrupt to occur
110         _int arg, int when, IntType type);// at time ``when''.  This is called
111                                         // by the hardware device simulators.
112
113     void OneTick();                     // Advance simulated time
114
115   private:
116     IntStatus level;            // are interrupts enabled or disabled?
117     List *pending;              // the list of interrupts scheduled
118                                 // to occur in the future
119     bool inHandler;             // TRUE if we are running an interrupt handler
120     bool yieldOnReturn;         // TRUE if we are to context switch
121                                 // on return from the interrupt handler
122     MachineStatus status;       // idle, kernel mode, user mode
123
124     // these functions are internal to the interrupt simulation code
125
126     bool CheckIfDue(bool advanceClock); // Check if an interrupt is supposed
127                                         // to occur now
128
129     void ChangeLevel(IntStatus old,     // SetLevel, without advancing the
130         IntStatus now);                 // simulated time
131 };
132
133 #endif // INTERRRUPT_H
```

3.6 interrupt.cc

```
 1 // interrupt.cc
 2 //      Routines to simulate hardware interrupts.
 3 //
 4 //      The hardware provides a routine (SetLevel) to enable or disable
 5 //      interrupts.
 6 //
 7 //      In order to emulate the hardware, we need to keep track of all
 8 //      interrupts the hardware devices would cause, and when they
 9 //      are supposed to occur.
10 //
11 //      This module also keeps track of simulated time.  Time advances
12 //      only when the following occur:
13 //              interrupts are re-enabled
14 //              a user instruction is executed
15 //              there is nothing in the ready queue
16 //
17 //  DO NOT CHANGE -- part of the machine emulation
```

```
18 //
19 // Copyright (c) 1992-1993 The Regents of the University of California.
20 // All rights reserved.  See copyright.h for copyright notice and limitation
21 // of liability and disclaimer of warranty provisions.
22
23 #include "copyright.h"
24 #include "interrupt.h"
25 #include "system.h"
26
27 // String definitions for debugging messages
28
29 static char *intLevelNames[] = { "off", "on"};
30 static char *intTypeNames[] = { "timer", "disk", "console write",
31                         "console read", "network send", "network recv"};
32
33 //----------------------------------------------------------------------
34 // PendingInterrupt::PendingInterrupt
35 //      Initialize a hardware device interrupt that is to be scheduled
36 //      to occur in the near future.
37 //
38 //      "func" is the procedure to call when the interrupt occurs
39 //      "param" is the argument to pass to the procedure
40 //      "time" is when (in simulated time) the interrupt is to occur
41 //      "kind" is the hardware device that generated the interrupt
42 //----------------------------------------------------------------------
43
44 PendingInterrupt::PendingInterrupt(VoidFunctionPtr func, _int param, int time,
45                                IntType kind)
46 {
47     handler = func;
48     arg = param;
49     when = time;
50     type = kind;
51 }
52
53 //----------------------------------------------------------------------
54 // Interrupt::Interrupt
55 //      Initialize the simulation of hardware device interrupts.
56 //
57 //      Interrupts start disabled, with no interrupts pending, etc.
58 //----------------------------------------------------------------------
59
60 Interrupt::Interrupt()
61 {
62     level = IntOff;
63     pending = new List();
64     inHandler = FALSE;
65     yieldOnReturn = FALSE;
66     status = SystemMode;
67 }
68
69 //----------------------------------------------------------------------
70 // Interrupt::~Interrupt
71 //      De-allocate the data structures needed by the interrupt simulation.
72 //----------------------------------------------------------------------
73
74 Interrupt::~Interrupt()
75 {
```

```
76     while (!pending->IsEmpty())
77         delete pending->Remove();
78     delete pending;
79 }
80
81 //----------------------------------------------------------------------
82 // Interrupt::ChangeLevel
83 //      Change interrupts to be enabled or disabled, without advancing
84 //      the simulated time (normally, enabling interrupts advances the time).
85
86 //----------------------------------------------------------------------
87 // Interrupt::ChangeLevel
88 //      Change interrupts to be enabled or disabled, without advancing
89 //      the simulated time (normally, enabling interrupts advances the time).
90 //
91 //      Used internally.
92 //
93 //      "old" -- the old interrupt status
94 //      "now" -- the new interrupt status
95 //----------------------------------------------------------------------
96 void
97 Interrupt::ChangeLevel(IntStatus old, IntStatus now)
98 {
99     level = now;
100    DEBUG('i',"\tinterrupts: %s -> %s\n",intLevelNames[old],intLevelNames[now]);
101 }
102
103 //----------------------------------------------------------------------
104 // Interrupt::SetLevel
105 //      Change interrupts to be enabled or disabled, and if interrupts
106 //      are being enabled, advance simulated time by calling OneTick().
107 //
108 // Returns:
109 //      The old interrupt status.
110 // Parameters:
111 //      "now" -- the new interrupt status
112 //----------------------------------------------------------------------
113
114 IntStatus
115 Interrupt::SetLevel(IntStatus now)
116 {
117     IntStatus old = level;
118
119     ASSERT((now == IntOff) || (inHandler == FALSE));// interrupt handlers are
120                                                     // prohibited from enabling
121                                                     // interrupts
122
123     ChangeLevel(old, now);                          // change to new state
124     if ((now == IntOn) && (old == IntOff))
125         OneTick();                                  // advance simulated time
126     return old;
127 }
128
129 //----------------------------------------------------------------------
130 // Interrupt::Enable
131 //      Turn interrupts on.  Who cares what they used to be?
132 //      Used in ThreadRoot, to turn interrupts on when first starting up
133 //       a thread.
```

```
134 //----------------------------------------------------------------------
135 void
136 Interrupt::Enable()
137 {
138     (void) SetLevel(IntOn);
139 }
140
141 //----------------------------------------------------------------------
142 // Interrupt::OneTick
143 //      Advance simulated time and check if there are any pending
144 //      interrupts to be called.
145 //
146 //      Two things can cause OneTick to be called:
147 //              interrupts are re-enabled
148 //              a user instruction is executed
149 //----------------------------------------------------------------------
150 void
151 Interrupt::OneTick()
152 {
153     MachineStatus old = status;
154
155 // advance simulated time
156     if (status == SystemMode) {
157         stats->totalTicks += SystemTick;
158         stats->systemTicks += SystemTick;
159     } else {                                    // USER_PROGRAM
160         stats->totalTicks += UserTick;
161         stats->userTicks += UserTick;
162     }
163     DEBUG('i', "\n== Tick %d ==\n", stats->totalTicks);
164
165 // check any pending interrupts are now ready to fire
166     ChangeLevel(IntOn, IntOff);         // first, turn off interrupts
167                                         // (interrupt handlers run with
168                                         // interrupts disabled)
169     while (CheckIfDue(FALSE))           // check for pending interrupts
170         ;
171     ChangeLevel(IntOff, IntOn);         // re-enable interrupts
172     if (yieldOnReturn) {                // if the timer device handler asked
173                                         // for a context switch, ok to do it now
174         yieldOnReturn = FALSE;
175         status = SystemMode;            // yield is a kernel routine
176         currentThread->Yield();
177         status = old;
178     }
179 }
180
181 //----------------------------------------------------------------------
182 // Interrupt::YieldOnReturn
183 //      Called from within an interrupt handler, to cause a context switch
184 //      (for example, on a time slice) in the interrupted thread,
185 //      when the handler returns.
186 //
187 //      We can't do the context switch here, because that would switch
188 //      out the interrupt handler, and we want to switch out the
189 //      interrupted thread.
190 //----------------------------------------------------------------------
191
```

```
192 void
193 Interrupt::YieldOnReturn()
194 {
195     ASSERT(inHandler == TRUE);
196     yieldOnReturn = TRUE;
197 }
198
199 //----------------------------------------------------------------------
200 // Interrupt::Idle
201 //      Routine called when there is nothing in the ready queue.
202 //
203 //      Since something has to be running in order to put a thread
204 //      on the ready queue, the only thing to do is to advance
205 //      simulated time until the next scheduled hardware interrupt.
206 //
207 //      If there are no pending interrupts, stop.  There's nothing
208 //      more for us to do.
209 //----------------------------------------------------------------------
210 void
211 Interrupt::Idle()
212 {
213     DEBUG('i', "Machine idling; checking for interrupts.\n");
214     status = IdleMode;
215     if (CheckIfDue(TRUE)) {             // check for any pending interrupts
216         while (CheckIfDue(FALSE))       // check for any other pending
217             ;                           // interrupts
218         yieldOnReturn = FALSE;          // since there's nothing in the
219                                         // ready queue, the yield is automatic
220         status = SystemMode;
221         return;                         // return in case there's now
222                                         // a runnable thread
223     }
224
225     // if there are no pending interrupts, and nothing is on the ready
226     // queue, it is time to stop.   If the console or the network is
227     // operating, there are *always* pending interrupts, so this code
228     // is not reached.  Instead, the halt must be invoked by the user program.
229
230     DEBUG('i', "Machine idle.  No interrupts to do.\n");
231     printf("No threads ready or runnable, and no pending interrupts.\n");
232     printf("Assuming the program completed.\n");
233     Halt();
234 }
235
236 //----------------------------------------------------------------------
237 // Interrupt::Halt
238 //      Shut down Nachos cleanly, printing out performance statistics.
239 //----------------------------------------------------------------------
240 void
241 Interrupt::Halt()
242 {
243     printf("Machine halting!\n\n");
244     stats->Print();
245     Cleanup();      // Never returns.
246 }
247
248 //----------------------------------------------------------------------
249 // Interrupt::Schedule
```

```
250 //      Arrange for the CPU to be interrupted when simulated time
251 //      reaches "now + when".
252 //
253 //      Implementation: just put it on a sorted list.
254 //
255 //      NOTE: the Nachos kernel should not call this routine directly.
256 //      Instead, it is only called by the hardware device simulators.
257 //
258 //      "handler" is the procedure to call when the interrupt occurs
259 //      "arg" is the argument to pass to the procedure
260 //      "fromNow" is how far in the future (in simulated time) the
261 //               interrupt is to occur
262 //      "type" is the hardware device that generated the interrupt
263 //----------------------------------------------------------------------
264 void
265 Interrupt::Schedule(VoidFunctionPtr handler, _int arg, int fromNow, IntType type)
266 {
267     int when = stats->totalTicks + fromNow;
268     PendingInterrupt *toOccur = new PendingInterrupt(handler, arg, when, type);
269
270     DEBUG('i', "Scheduling interrupt handler the %s at time = %d\n",
271                                         intTypeNames[type], when);
272     ASSERT(fromNow > 0);
273
274     pending->SortedInsert(toOccur, when);
275 }
276
277 //----------------------------------------------------------------------
278 // Interrupt::CheckIfDue
279 //      Check if an interrupt is scheduled to occur, and if so, fire it off.
280 //
281 // Returns:
282 //      TRUE, if we fired off any interrupt handlers
283 // Params:
284 //      "advanceClock" -- if TRUE, there is nothing in the ready queue,
285 //              so we should simply advance the clock to when the next
286 //              pending interrupt would occur (if any).  If the pending
287 //              interrupt is just the time-slice daemon, however, then
288 //              we're done!
289 //----------------------------------------------------------------------
290 bool
291 Interrupt::CheckIfDue(bool advanceClock)
292 {
293     MachineStatus old = status;
294     int when;
295
296     ASSERT(level == IntOff);            // interrupts need to be disabled,
297                                         // to invoke an interrupt handler
298     if (DebugIsEnabled('i'))
299         DumpState();
300     PendingInterrupt *toOccur =
301                 (PendingInterrupt *)pending->SortedRemove(&when);
302
303     if (toOccur == NULL)                // no pending interrupts
304         return FALSE;
305
306     if (advanceClock && when > stats->totalTicks) {     // advance the clock
307         stats->idleTicks += (when - stats->totalTicks);
```

```
308         stats->totalTicks = when;
309     } else if (when > stats->totalTicks) {      // not time yet, put it back
310         pending->SortedInsert(toOccur, when);
311         return FALSE;
312     }
313
314 // Check if there is nothing more to do, and if so, quit
315     if ((status == IdleMode) && (toOccur->type == TimerInt)
316                                 && pending->IsEmpty()) {
317          pending->SortedInsert(toOccur, when);
318          return FALSE;
319     }
320
321     DEBUG('i', "Invoking interrupt handler for the %s at time %d\n",
322                           intTypeNames[toOccur->type], toOccur->when);
323 #ifdef USER_PROGRAM
324     if (machine != NULL)
325         machine->DelayedLoad(0, 0);
326 #endif
327     inHandler = TRUE;
328     status = SystemMode;                        // whatever we were doing,
329                                                 // we are now going to be
330                                                 // running in the kernel
331     (*(toOccur->handler))(toOccur->arg); // call the interrupt handler
332     status = old;                               // restore the machine status
333     inHandler = FALSE;
334     delete toOccur;
335     return TRUE;
336 }
337
338 //----------------------------------------------------------------------
339 // PrintPending
340 //      Print information about an interrupt that is scheduled to occur.
341 //      When, where, why, etc.
342 //----------------------------------------------------------------------
343
344 static void
345 PrintPending(_int arg)
346 {
347     PendingInterrupt *pend = (PendingInterrupt *)arg;
348
349     printf("Interrupt handler %s, scheduled at %d\n",
350         intTypeNames[pend->type], pend->when);
351 }
352
353 //----------------------------------------------------------------------
354 // DumpState
355 //      Print the complete interrupt state - the status, and all interrupts
356 //      that are scheduled to occur in the future.
357 //----------------------------------------------------------------------
358
359 void
360 Interrupt::DumpState()
361 {
362     printf("Time: %d, interrupts %s\n", stats->totalTicks,
363                                         intLevelNames[level]);
364     printf("Pending interrupts:\n");
365     fflush(stdout);
```

```
366     pending->Mapcar(PrintPending);
367     printf("End of pending interrupts\n");
368     fflush(stdout);
369 }
```

3.7 machine.h

```
  1 // machine.h
  2 //      Data structures for simulating the execution of user programs
  3 //      running on top of Nachos.
  4 //
  5 //      User programs are loaded into "mainMemory"; to Nachos,
  6 //      this looks just like an array of bytes.  Of course, the Nachos
  7 //      kernel is in memory too -- but as in most machines these days,
  8 //      the kernel is loaded into a separate memory region from user
  9 //      programs, and accesses to kernel memory are not translated or paged.
 10 //
 11 //      In Nachos, user programs are executed one instruction at a time,
 12 //      by the simulator.  Each memory reference is translated, checked
 13 //      for errors, etc.
 14 //
 15 //  DO NOT CHANGE -- part of the machine emulation
 16 //
 17 // Copyright (c) 1992-1993 The Regents of the University of California.
 18 // All rights reserved.  See copyright.h for copyright notice and limitation
 19 // of liability and disclaimer of warranty provisions.
 20
 21 #ifndef MACHINE_H
 22 #define MACHINE_H
 23
 24 #include "copyright.h"
 25 #include "utility.h"
 26 #include "translate.h"
 27 #include "disk.h"
 28
 29 // Definitions related to the size, and format of user memory
 30
 31 #define PageSize        SectorSize      // set the page size equal to
 32                                         // the disk sector size, for
 33                                         // simplicity
 34
 35 #define NumPhysPages    32
 36 #define MemorySize      (NumPhysPages * PageSize)
 37 #define TLBSize         4               // if there is a TLB, make it small
 38
 39 enum ExceptionType { NoException,           // Everything ok!
 40                      SyscallException,      // A program executed a system call.
 41                      PageFaultException,    // No valid translation found
 42                      ReadOnlyException,     // Write attempted to page marked
 43                                             // "read-only"
 44                      BusErrorException,     // Translation resulted in an
 45                                             // invalid physical address
 46                      AddressErrorException, // Unaligned reference or one that
 47                                             // was beyond the end of the
 48                                             // address space
 49                      OverflowException,     // Integer overflow in add or sub.
 50                      IllegalInstrException, // Unimplemented or reserved instr.
```

```
51
52                      NumExceptionTypes
53 };
54
55 // User program CPU state.  The full set of MIPS registers, plus a few
56 // more because we need to be able to start/stop a user program between
57 // any two instructions (thus we need to keep track of things like load
58 // delay slots, etc.)
59
60 #define StackReg        29      // User's stack pointer
61 #define RetAddrReg      31      // Holds return address for procedure calls
62 #define NumGPRegs       32      // 32 general purpose registers on MIPS
63 #define HiReg           32      // Double register to hold multiply result
64 #define LoReg           33
65 #define PCReg           34      // Current program counter
66 #define NextPCReg       35      // Next program counter (for branch delay)
67 #define PrevPCReg        36      // Previous program counter (for debugging)
68 #define LoadReg         37      // The register target of a delayed load.
69 #define LoadValueReg    38      // The value to be loaded by a delayed load.
70 #define BadVAddrReg     39      // The failing virtual address on an exception
71
72 #define NumTotalRegs    40
73
74 // The following class defines an instruction, represented in both
75 //      undecoded binary form
76 //      decoded to identify
77 //          operation to do
78 //          registers to act on
79 //          any immediate operand value
80
81 class Instruction {
82   public:
83     void Decode();      // decode the binary representation of the instruction
84
85     unsigned int value; // binary representation of the instruction
86
87     char opCode;     // Type of instruction.  This is NOT the same as the
88                      // opcode field from the instruction: see defs in mips.h
89     char rs, rt, rd; // Three registers from instruction.
90     int extra;       // Immediate or target or shamt field or offset.
91                      // Immediates are sign-extended.
92 };
93
94 // The following class defines the simulated host workstation hardware, as
95 // seen by user programs -- the CPU registers, main memory, etc.
96 // User programs shouldn't be able to tell that they are running on our
97 // simulator or on the real hardware, except
98 //      we don't support floating point instructions
99 //      the system call interface to Nachos is not the same as UNIX
100 //       (10 system calls in Nachos vs. 200 in UNIX!)
101 // If we were to implement more of the UNIX system calls, we ought to be
102 // able to run Nachos on top of Nachos!
103 //
104 // The procedures in this class are defined in machine.cc, mipssim.cc, and
105 // translate.cc.
106
107 class Machine {
108   public:
```

```
109     Machine(bool debug);        // Initialize the simulation of the hardware
110                                 // for running user programs
111     ~Machine();                 // De-allocate the data structures
112
113 // Routines callable by the Nachos kernel
114     void Run();                 // Run a user program
115
116     int ReadRegister(int num);  // read the contents of a CPU register
117
118     void WriteRegister(int num, int value);
119                                 // store a value into a CPU register
120
121
122 // Routines internal to the machine simulation -- DO NOT call these
123
124     void OneInstruction(Instruction *instr);
125                                 // Run one instruction of a user program.
126     void DelayedLoad(int nextReg, int nextVal);
127                                 // Do a pending delayed load (modifying a reg)
128
129     bool ReadMem(int addr, int size, int* value);
130     bool WriteMem(int addr, int size, int value);
131                                 // Read or write 1, 2, or 4 bytes of virtual
132                                 // memory (at addr).  Return FALSE if a
133                                  // correct translation couldn't be found.
134
135     ExceptionType Translate(int virtAddr, int* physAddr, int size,bool writing);
136                                 // Translate an address, and check for
137                                 // alignment.  Set the use and dirty bits in
138                                 // the translation entry appropriately,
139                                 // and return an exception code if the
140                                 // translation couldn't be completed.
141
142     void RaiseException(ExceptionType which, int badVAddr);
143                                 // Trap to the Nachos kernel, because of a
144                                 // system call or other exception.
145
146     void Debugger();            // invoke the user program debugger
147     void DumpState();           // print the user CPU and memory state
148
149
150 // Data structures -- all of these are accessible to Nachos kernel code.
151 // "public" for convenience.
152 //
153 // Note that *all* communication between the user program and the kernel
154 // are in terms of these data structures.
155
156     char *mainMemory;           // physical memory to store user program,
157                                 // code and data, while executing
158     int registers[NumTotalRegs]; // CPU registers, for executing user programs
159
160
161 // NOTE: the hardware translation of virtual addresses in the user program
162 // to physical addresses (relative to the beginning of "mainMemory")
163 // can be controlled by one of:
164 //      a traditional linear page table
165 //      a software-loaded translation lookaside buffer (tlb) -- a cache of
166 //        mappings of virtual page #'s to physical page #'s
```

```
167 //
168 // If "tlb" is NULL, the linear page table is used
169 // If "tlb" is non-NULL, the Nachos kernel is responsible for managing
170 //      the contents of the TLB.  But the kernel can use any data structure
171 //      it wants (eg, segmented paging) for handling TLB cache misses.
172 //
173 // For simplicity, both the page table pointer and the TLB pointer are
174 // public.  However, while there can be multiple page tables (one per address
175 // space, stored in memory), there is only one TLB (implemented in hardware).
176 // Thus the TLB pointer should be considered as *read-only*, although
177 // the contents of the TLB are free to be modified by the kernel software.
178
179     TranslationEntry *tlb;              // this pointer should be considered
180                                         // "read-only" to Nachos kernel code
181
182     TranslationEntry *pageTable;
183     unsigned int pageTableSize;
184
185   private:
186     bool singleStep;            // drop back into the debugger after each
187                                 // simulated instruction
188     int runUntilTime;           // drop back into the debugger when simulated
189                                 // time reaches this value
190 };
191
192 extern void ExceptionHandler(ExceptionType which);
193                                 // Entry point into Nachos for handling
194                                 // user system calls and exceptions
195                                 // Defined in exception.cc
196
197
198 // Routines for converting Words and Short Words to and from the
199 // simulated machine's format of little endian.  If the host machine
200 // is little endian (DEC and Intel), these end up being NOPs.
201 //
202 // What is stored in each format:
203 //      host byte ordering:
204 //         kernel data structures
205 //         user registers
206 //      simulated machine byte ordering:
207 //         contents of main memory
208
209 unsigned int WordToHost(unsigned int word);
210 unsigned short ShortToHost(unsigned short shortword);
211 unsigned int WordToMachine(unsigned int word);
212 unsigned short ShortToMachine(unsigned short shortword);
213
214 #endif // MACHINE_H
```

3.8 machine.cc

```
  1 // machine.cc
  2 //      Routines for simulating the execution of user programs.
  3 //
  4 //  DO NOT CHANGE -- part of the machine emulation
  5 //
  6 // Copyright (c) 1992-1993 The Regents of the University of California.
```

```
 7 // All rights reserved.  See copyright.h for copyright notice and limitation
 8 // of liability and disclaimer of warranty provisions.
 9
10 #include "copyright.h"
11 #include "machine.h"
12 #include "system.h"
13
14 // Textual names of the exceptions that can be generated by user program
15 // execution, for debugging.
16 static char* exceptionNames[] = { "no exception", "syscall",
17                                   "page fault/no TLB entry", "page read only",
18                                   "bus error", "address error", "overflow",
19                                   "illegal instruction" };
20
21 //----------------------------------------------------------------------
22 // CheckEndian
23 //      Check to be sure that the host really uses the format it says it
24 //      does, for storing the bytes of an integer.  Stop on error.
25 //----------------------------------------------------------------------
26
27 static
28 void CheckEndian()
29 {
30     union checkit {
31         char charword[4];
32         unsigned int intword;
33     } check;
34
35     check.charword[0] = 1;
36     check.charword[1] = 2;
37     check.charword[2] = 3;
38     check.charword[3] = 4;
39
40 #ifdef HOST_IS_BIG_ENDIAN
41     ASSERT (check.intword == 0x01020304);
42 #else
43     ASSERT (check.intword == 0x04030201);
44 #endif
45 }
46
47 //----------------------------------------------------------------------
48 // Machine::Machine
49 //      Initialize the simulation of user program execution.
50 //
51 //      "debug" -- if TRUE, drop into the debugger after each user instruction
52 //              is executed.
53 //----------------------------------------------------------------------
54
55 Machine::Machine(bool debug)
56 {
57     int i;
58
59     for (i = 0; i < NumTotalRegs; i++)
60         registers[i] = 0;
61     mainMemory = new char[MemorySize];
62     for (i = 0; i < MemorySize; i++)
63         mainMemory[i] = 0;
64 #ifdef USE_TLB
```

```
65      tlb = new TranslationEntry[TLBSize];
66      for (i = 0; i < TLBSize; i++)
67           tlb[i].valid = FALSE;
68      pageTable = NULL;
69  #else   // use linear page table
70      tlb = NULL;
71      pageTable = NULL;
72  #endif
73
74      singleStep = debug;
75      CheckEndian();
76  }
77
78  //----------------------------------------------------------------------
79  // Machine::~Machine
80  //      De-allocate the data structures used to simulate user program execution.
81  //----------------------------------------------------------------------
82
83  Machine::~Machine()
84  {
85      delete [] mainMemory;
86      if (tlb != NULL)
87          delete [] tlb;
88  }
89
90  //----------------------------------------------------------------------
91  // Machine::RaiseException
92  //      Transfer control to the Nachos kernel from user mode, because
93  //      the user program either invoked a system call, or some exception
94  //      occured (such as the address translation failed).
95  //
96  //      "which" -- the cause of the kernel trap
97  //      "badVaddr" -- the virtual address causing the trap, if appropriate
98  //----------------------------------------------------------------------
99
100 void
101 Machine::RaiseException(ExceptionType which, int badVAddr)
102 {
103     DEBUG('m', "Exception: %s\n", exceptionNames[which]);
104
105 //  ASSERT(interrupt->getStatus() == UserMode);
106     registers[BadVAddrReg] = badVAddr;
107     DelayedLoad(0, 0);                  // finish anything in progress
108     interrupt->setStatus(SystemMode);
109     ExceptionHandler(which);            // interrupts are enabled at this point
110     interrupt->setStatus(UserMode);
111 }
112
113 //----------------------------------------------------------------------
114 // Machine::Debugger
115 //      Primitive debugger for user programs.  Note that we can't use
116 //      gdb to debug user programs, since gdb doesn't run on top of Nachos.
117 //      It could, but you'd have to implement *a lot* more system calls
118 //      to get it to work!
119 //
120 //      So just allow single-stepping, and printing the contents of memory.
121 //----------------------------------------------------------------------
122
```

```
123 void Machine::Debugger()
124 {
125     char *buf = new char[80];
126     int num;
127
128     interrupt->DumpState();
129     DumpState();
130     printf("%d> ", stats->totalTicks);
131     fflush(stdout);
132     fgets(buf, 80, stdin);
133     if (sscanf(buf, "%d", &num) == 1)
134         runUntilTime = num;
135     else {
136         runUntilTime = 0;
137         switch (*buf) {
138           case '\n':
139             break;
140
141           case 'c':
142             singleStep = FALSE;
143             break;
144
145           case '?':
146             printf("Machine commands:\n");
147             printf("    <return>  execute one instruction\n");
148             printf("    <number>  run until the given timer tick\n");
149             printf("    c         run until completion\n");
150             printf("    ?         print help message\n");
151             break;
152         }
153     }
154     delete [] buf;
155 }
156
157 //----------------------------------------------------------------------
158 // Machine::DumpState
159 //      Print the user program's CPU state.  We might print the contents
160 //      of memory, but that seemed like overkill.
161 //----------------------------------------------------------------------
162
163 void
164 Machine::DumpState()
165 {
166     int i;
167
168     printf("Machine registers:\n");
169     for (i = 0; i < NumGPRegs; i++)
170         switch (i) {
171           case StackReg:
172             printf("\tSP(%d):\t0x%x%s", i, registers[i],
173                    ((i % 4) == 3) ? "\n" : "");
174             break;
175
176           case RetAddrReg:
177             printf("\tRA(%d):\t0x%x%s", i, registers[i],
178                    ((i % 4) == 3) ? "\n" : "");
179             break;
180
```

```
181           default:
182             printf("\t%d:\t0x%x%s", i, registers[i],
183                    ((i % 4) == 3) ? "\n" : "");
184             break;
185         }
186
187     printf("\tHi:\t0x%x", registers[HiReg]);
188     printf("\tLo:\t0x%x\n", registers[LoReg]);
189     printf("\tPC:\t0x%x", registers[PCReg]);
190     printf("\tNextPC:\t0x%x", registers[NextPCReg]);
191     printf("\tPrevPC:\t0x%x\n", registers[PrevPCReg]);
192     printf("\tLoad:\t0x%x", registers[LoadReg]);
193     printf("\tLoadV:\t0x%x\n", registers[LoadValueReg]);
194     printf("\n");
195 }
196
197 //----------------------------------------------------------------------
198 // Machine::ReadRegister/WriteRegister
199 //       Fetch or write the contents of a user program register.
200 //----------------------------------------------------------------------
201
202 int Machine::ReadRegister(int num)
203     {
204         ASSERT((num >= 0) && (num < NumTotalRegs));
205         return registers[num];
206     }
207
208 void Machine::WriteRegister(int num, int value)
209     {
210         ASSERT((num >= 0) && (num < NumTotalRegs));
211         // DEBUG('m', "WriteRegister %d, value %d\n", num, value);
212         registers[num] = value;
213     }
214
```

3.9 mipssim.h

```
 1 // mipssim.h
 2 //      Internal data structures for simulating the MIPS instruction set.
 3 //
 4 //  DO NOT CHANGE -- part of the machine emulation
 5 //
 6 // Copyright (c) 1992-1993 The Regents of the University of California.
 7 // All rights reserved.  See copyright.h for copyright notice and limitation
 8 // of liability and disclaimer of warranty provisions.
 9
10 #ifndef MIPSSIM_H
11 #define MIPSSIM_H
12
13 #include "copyright.h"
14
15 /*
16  * OpCode values.  The names are straight from the MIPS
17  * manual except for the following special ones:
18  *
19  * OP_UNIMP -           means that this instruction is legal, but hasn't
20  *                      been implemented in the simulator yet.
```

```
21  * OP_RES -             means that this is a reserved opcode (it isn't
22  *                      supported by the architecture).
23  */
24
25 #define OP_ADD          1
26 #define OP_ADDI         2
27 #define OP_ADDIU        3
28 #define OP_ADDU         4
29 #define OP_AND          5
30 #define OP_ANDI         6
31 #define OP_BEQ          7
32 #define OP_BGEZ         8
33 #define OP_BGEZAL       9
34 #define OP_BGTZ         10
35 #define OP_BLEZ         11
36 #define OP_BLTZ         12
37 #define OP_BLTZAL       13
38 #define OP_BNE          14
39
40 #define OP_DIV          16
41 #define OP_DIVU         17
42 #define OP_J            18
43 #define OP_JAL          19
44 #define OP_JALR         20
45 #define OP_JR           21
46 #define OP_LB           22
47 #define OP_LBU          23
48 #define OP_LH           24
49 #define OP_LHU          25
50 #define OP_LUI          26
51 #define OP_LW           27
52 #define OP_LWL          28
53 #define OP_LWR          29
54
55 #define OP_MFHI         31
56 #define OP_MFLO         32
57
58 #define OP_MTHI         34
59 #define OP_MTLO         35
60 #define OP_MULT         36
61 #define OP_MULTU        37
62 #define OP_NOR          38
63 #define OP_OR           39
64 #define OP_ORI          40
65 #define OP_RFE          41
66 #define OP_SB           42
67 #define OP_SH           43
68 #define OP_SLL          44
69 #define OP_SLLV         45
70 #define OP_SLT          46
71 #define OP_SLTI         47
72 #define OP_SLTIU        48
73 #define OP_SLTU         49
74 #define OP_SRA          50
75 #define OP_SRAV         51
76 #define OP_SRL          52
77 #define OP_SRLV         53
78 #define OP_SUB          54
```

```
 79 #define OP_SUBU         55
 80 #define OP_SW           56
 81 #define OP_SWL          57
 82 #define OP_SWR          58
 83 #define OP_XOR          59
 84 #define OP_XORI         60
 85 #define OP_SYSCALL      61
 86 #define OP_UNIMP        62
 87 #define OP_RES          63
 88 #define MaxOpcode       63
 89
 90 /*
 91  * Miscellaneous definitions:
 92  */
 93
 94 #define IndexToAddr(x) ((x) << 2)
 95
 96 #define SIGN_BIT        0x80000000
 97 #define R31             31
 98
 99 /*
100  * The table below is used to translate bits 31:26 of the instruction
101  * into a value suitable for the "opCode" field of a MemWord structure,
102  * or into a special value for further decoding.
103  */
104
105 #define SPECIAL 100
106 #define BCOND   101
107
108 #define IFMT 1
109 #define JFMT 2
110 #define RFMT 3
111
112 struct OpInfo {
113     int opCode;         /* Translated op code. */
114     int format;         /* Format type (IFMT or JFMT or RFMT) */
115 };
116
117 static OpInfo opTable[] = {
118     {SPECIAL, RFMT}, {BCOND, IFMT}, {OP_J, JFMT}, {OP_JAL, JFMT},
119     {OP_BEQ, IFMT}, {OP_BNE, IFMT}, {OP_BLEZ, IFMT}, {OP_BGTZ, IFMT},
120     {OP_ADDI, IFMT}, {OP_ADDIU, IFMT}, {OP_SLTI, IFMT}, {OP_SLTIU, IFMT},
121     {OP_ANDI, IFMT}, {OP_ORI, IFMT}, {OP_XORI, IFMT}, {OP_LUI, IFMT},
122     {OP_UNIMP, IFMT}, {OP_UNIMP, IFMT}, {OP_UNIMP, IFMT}, {OP_UNIMP, IFMT},
123     {OP_RES, IFMT}, {OP_RES, IFMT}, {OP_RES, IFMT}, {OP_RES, IFMT},
124     {OP_RES, IFMT}, {OP_RES, IFMT}, {OP_RES, IFMT}, {OP_RES, IFMT},
125     {OP_RES, IFMT}, {OP_RES, IFMT}, {OP_RES, IFMT}, {OP_RES, IFMT},
126     {OP_LB, IFMT}, {OP_LH, IFMT}, {OP_LWL, IFMT}, {OP_LW, IFMT},
127     {OP_LBU, IFMT}, {OP_LHU, IFMT}, {OP_LWR, IFMT}, {OP_RES, IFMT},
128     {OP_SB, IFMT}, {OP_SH, IFMT}, {OP_SWL, IFMT}, {OP_SW, IFMT},
129     {OP_RES, IFMT}, {OP_RES, IFMT}, {OP_SWR, IFMT}, {OP_RES, IFMT},
130     {OP_UNIMP, IFMT}, {OP_UNIMP, IFMT}, {OP_UNIMP, IFMT}, {OP_UNIMP, IFMT},
131     {OP_RES, IFMT}, {OP_RES, IFMT}, {OP_RES, IFMT}, {OP_RES, IFMT},
132     {OP_UNIMP, IFMT}, {OP_UNIMP, IFMT}, {OP_UNIMP, IFMT}, {OP_UNIMP, IFMT},
133     {OP_RES, IFMT}, {OP_RES, IFMT}, {OP_RES, IFMT}, {OP_RES, IFMT}
134 };
135
136 /*
```

```
137  * The table below is used to convert the "funct" field of SPECIAL
138  * instructions into the "opCode" field of a MemWord.
139  */
140
141 static int specialTable[] = {
142     OP_SLL, OP_RES, OP_SRL, OP_SRA, OP_SLLV, OP_RES, OP_SRLV, OP_SRAV,
143     OP_JR, OP_JALR, OP_RES, OP_RES, OP_SYSCALL, OP_UNIMP, OP_RES, OP_RES,
144     OP_MFHI, OP_MTHI, OP_MFLO, OP_MTLO, OP_RES, OP_RES, OP_RES, OP_RES,
145     OP_MULT, OP_MULTU, OP_DIV, OP_DIVU, OP_RES, OP_RES, OP_RES, OP_RES,
146     OP_ADD, OP_ADDU, OP_SUB, OP_SUBU, OP_AND, OP_OR, OP_XOR, OP_NOR,
147     OP_RES, OP_RES, OP_SLT, OP_SLTU, OP_RES, OP_RES, OP_RES, OP_RES,
148     OP_RES, OP_RES, OP_RES, OP_RES, OP_RES, OP_RES, OP_RES, OP_RES,
149     OP_RES, OP_RES, OP_RES, OP_RES, OP_RES, OP_RES, OP_RES, OP_RES
150 };
151
152
153 // Stuff to help print out each instruction, for debugging
154
155 enum RegType { NONE, RS, RT, RD, EXTRA };
156
157 struct OpString {
158     char *string;       // Printed version of instruction
159     RegType args[3];
160 };
161
162 static struct OpString opStrings[] = {
163         {"Shouldn't happen", {NONE, NONE, NONE}},
164         {"ADD r%d,r%d,r%d", {RD, RS, RT}},
165         {"ADDI r%d,r%d,%d", {RT, RS, EXTRA}},
166         {"ADDIU r%d,r%d,%d", {RT, RS, EXTRA}},
167         {"ADDU r%d,r%d,r%d", {RD, RS, RT}},
168         {"AND r%d,r%d,r%d", {RD, RS, RT}},
169         {"ANDI r%d,r%d,%d", {RT, RS, EXTRA}},
170         {"BEQ r%d,r%d,%d", {RS, RT, EXTRA}},
171         {"BGEZ r%d,%d", {RS, EXTRA, NONE}},
172         {"BGEZAL r%d,%d", {RS, EXTRA, NONE}},
173         {"BGTZ r%d,%d", {RS, EXTRA, NONE}},
174         {"BLEZ r%d,%d", {RS, EXTRA, NONE}},
175         {"BLTZ r%d,%d", {RS, EXTRA, NONE}},
176         {"BLTZAL r%d,%d", {RS, EXTRA, NONE}},
177         {"BNE r%d,r%d,%d", {RS, RT, EXTRA}},
178         {"Shouldn't happen", {NONE, NONE, NONE}},
179         {"DIV r%d,r%d", {RS, RT, NONE}},
180         {"DIVU r%d,r%d", {RS, RT, NONE}},
181         {"J %d", {EXTRA, NONE, NONE}},
182         {"JAL %d", {EXTRA, NONE, NONE}},
183         {"JALR r%d,r%d", {RD, RS, NONE}},
184         {"JR r%d,r%d", {RD, RS, NONE}},
185         {"LB r%d,%d(r%d)", {RT, EXTRA, RS}},
186         {"LBU r%d,%d(r%d)", {RT, EXTRA, RS}},
187         {"LH r%d,%d(r%d)", {RT, EXTRA, RS}},
188         {"LHU r%d,%d(r%d)", {RT, EXTRA, RS}},
189         {"LUI r%d,%d", {RT, EXTRA, NONE}},
190         {"LW r%d,%d(r%d)", {RT, EXTRA, RS}},
191         {"LWL r%d,%d(r%d)", {RT, EXTRA, RS}},
192         {"LWR r%d,%d(r%d)", {RT, EXTRA, RS}},
193         {"Shouldn't happen", {NONE, NONE, NONE}},
194         {"MFHI r%d", {RD, NONE, NONE}},
```

```
195         {"MFLO r%d", {RD, NONE, NONE}},
196         {"Shouldn't happen", {NONE, NONE, NONE}},
197         {"MTHI r%d", {RS, NONE, NONE}},
198         {"MTLO r%d", {RS, NONE, NONE}},
199          {"MULT r%d,r%d", {RS, RT, NONE}},
200         {"MULTU r%d,r%d", {RS, RT, NONE}},
201         {"NOR r%d,r%d,r%d", {RD, RS, RT}},
202         {"OR r%d,r%d,r%d", {RD, RS, RT}},
203         {"ORI r%d,r%d,%d", {RT, RS, EXTRA}},
204         {"RFE", {NONE, NONE, NONE}},
205         {"SB r%d,%d(r%d)", {RT, EXTRA, RS}},
206         {"SH r%d,%d(r%d)", {RT, EXTRA, RS}},
207         {"SLL r%d,r%d,%d", {RD, RT, EXTRA}},
208         {"SLLV r%d,r%d,r%d", {RD, RT, RS}},
209         {"SLT r%d,r%d,r%d", {RD, RS, RT}},
210         {"SLTI r%d,r%d,%d", {RT, RS, EXTRA}},
211         {"SLTIU r%d,r%d,%d", {RT, RS, EXTRA}},
212         {"SLTU r%d,r%d,r%d", {RD, RS, RT}},
213         {"SRA r%d,r%d,%d", {RD, RT, EXTRA}},
214         {"SRAV r%d,r%d,r%d", {RD, RT, RS}},
215         {"SRL r%d,r%d,%d", {RD, RT, EXTRA}},
216         {"SRLV r%d,r%d,r%d", {RD, RT, RS}},
217         {"SUB r%d,r%d,r%d", {RD, RS, RT}},
218         {"SUBU r%d,r%d,r%d", {RD, RS, RT}},
219         {"SW r%d,%d(r%d)", {RT, EXTRA, RS}},
220         {"SWL r%d,%d(r%d)", {RT, EXTRA, RS}},
221         {"SWR r%d,%d(r%d)", {RT, EXTRA, RS}},
222         {"XOR r%d,r%d,r%d", {RD, RS, RT}},
223         {"XORI r%d,r%d,%d", {RT, RS, EXTRA}},
224         {"SYSCALL", {NONE, NONE, NONE}},
225         {"Unimplemented", {NONE, NONE, NONE}},
226         {"Reserved", {NONE, NONE, NONE}}
227       };
228
229 #endif // MIPSSIM_H
```

3.10 mipssim.cc

```
 1 // mipssim.cc -- simulate a MIPS R2/3000 processor
 2 //
 3 //   This code has been adapted from Ousterhout's MIPSSIM package.
 4 //   Byte ordering is little-endian, so we can be compatible with
 5 //   DEC RISC systems.
 6 //
 7 //   DO NOT CHANGE -- part of the machine emulation
 8 //
 9 // Copyright (c) 1992-1993 The Regents of the University of California.
10 // All rights reserved.  See copyright.h for copyright notice and limitation
11 // of liability and disclaimer of warranty provisions.
12
13 #include "copyright.h"
14
15 #include "machine.h"
16 #include "mipssim.h"
17 #include "system.h"
18
19 static void Mult(int a, int b, bool signedArith, int* hiPtr, int* loPtr);
```

```
20
21 //----------------------------------------------------------------------
22 // Machine::Run
23 //      Simulate the execution of a user-level program on Nachos.
24 //      Called by the kernel when the program starts up; never returns.
25 //
26 //      This routine is re-entrant, in that it can be called multiple
27 //      times concurrently -- one for each thread executing user code.
28 //----------------------------------------------------------------------
29
30 void
31 Machine::Run()
32 {
33     Instruction *instr = new Instruction;  // storage for decoded instruction
34
35     if(DebugIsEnabled('m'))
36         printf("Starting thread \"%s\" at time %d\n",
37                currentThread->getName(), stats->totalTicks);
38     interrupt->setStatus(UserMode);
39     for (;;) {
40         OneInstruction(instr);
41         interrupt->OneTick();
42         if (singleStep && (runUntilTime <= stats->totalTicks))
43           Debugger();
44     }
45 }
46
47
48 //----------------------------------------------------------------------
49 // TypeToReg
50 //      Retrieve the register # referred to in an instruction.
51 //----------------------------------------------------------------------
52
53 static int
54 TypeToReg(RegType reg, Instruction *instr)
55 {
56     switch (reg) {
57       case RS:
58         return instr->rs;
59       case RT:
60         return instr->rt;
61       case RD:
62         return instr->rd;
63       case EXTRA:
64         return instr->extra;
65       default:
66         return -1;
67     }
68 }
69
70 //----------------------------------------------------------------------
71 // Machine::OneInstruction
72 //      Execute one instruction from a user-level program
73 //
74 //      If there is any kind of exception or interrupt, we invoke the
75 //      exception handler, and when it returns, we return to Run(), which
76 //      will re-invoke us in a loop.  This allows us to
77 //      re-start the instruction execution from the beginning, in
```

```
78 //      case any of our state has changed.  On a syscall,
79 //      the OS software must increment the PC so execution begins
80 //      at the instruction immediately after the syscall.
81 //
82 //      This routine is re-entrant, in that it can be called multiple
83 //      times concurrently -- one for each thread executing user code.
84 //      We get re-entrancy by never caching any data -- we always re-start the
85 //      simulation from scratch each time we are called (or after trapping
86 //      back to the Nachos kernel on an exception or interrupt), and we always
87 //      store all data back to the machine registers and memory before
88 //      leaving.  This allows the Nachos kernel to control our behavior
89 //      by controlling the contents of memory, the translation table,
90 //      and the register set.
91 //----------------------------------------------------------------------
92
93 void
94 Machine::OneInstruction(Instruction *instr)
95 {
96     int raw;
97     int nextLoadReg = 0;
98     int nextLoadValue = 0;      // record delayed load operation, to apply
99                                 // in the future
100
101     // Fetch instruction
102     if (!machine->ReadMem(registers[PCReg], 4, &raw))
103         return;                 // exception occurred
104     instr->value = raw;
105     instr->Decode();
106
107     if (DebugIsEnabled('m')) {
108        struct OpString *str = &opStrings[instr->opCode];
109
110        ASSERT(instr->opCode <= MaxOpcode);
111        printf("At PC = 0x%x: ", registers[PCReg]);
112        printf(str->string, TypeToReg(str->args[0], instr),
113                 TypeToReg(str->args[1], instr), TypeToReg(str->args[2], instr));
114        printf("\n");
115        }
116
117     // Compute next pc, but don't install in case there's an error or branch.
118     int pcAfter = registers[NextPCReg] + 4;
119     int sum, diff, tmp, value;
120     unsigned int rs, rt, imm;
121
122     // Execute the instruction (cf. Kane's book)
123     switch (instr->opCode) {
124
125       case OP_ADD:
126         sum = registers[instr->rs] + registers[instr->rt];
127         if (!((registers[instr->rs] ^ registers[instr->rt]) & SIGN_BIT) &&
128             ((registers[instr->rs] ^ sum) & SIGN_BIT)) {
129             RaiseException(OverflowException, 0);
130             return;
131         }
132         registers[instr->rd] = sum;
133          break;
134
135       case OP_ADDI:
```

```
136         sum = registers[instr->rs] + instr->extra;
137         if (!((registers[instr->rs] ^ instr->extra) & SIGN_BIT) &&
138             ((instr->extra ^ sum) & SIGN_BIT)) {
139             RaiseException(OverflowException, 0);
140             return;
141         }
142         registers[instr->rt] = sum;
143         break;
144
145       case OP_ADDIU:
146         registers[instr->rt] = registers[instr->rs] + instr->extra;
147         break;
148
149       case OP_ADDU:
150         registers[instr->rd] = registers[instr->rs] + registers[instr->rt];
151         break;
152
153       case OP_AND:
154         registers[instr->rd] = registers[instr->rs] & registers[instr->rt];
155         break;
156
157       case OP_ANDI:
158         registers[instr->rt] = registers[instr->rs] & (instr->extra & 0xffff);
159         break;
160
161       case OP_BEQ:
162         if (registers[instr->rs] == registers[instr->rt])
163             pcAfter = registers[NextPCReg] + IndexToAddr(instr->extra);
164         break;
165
166       case OP_BGEZAL:
167         registers[R31] = registers[NextPCReg] + 4;
168       case OP_BGEZ:
169         if (!(registers[instr->rs] & SIGN_BIT))
170             pcAfter = registers[NextPCReg] + IndexToAddr(instr->extra);
171         break;
172
173       case OP_BGTZ:
174         if (registers[instr->rs] > 0)
175             pcAfter = registers[NextPCReg] + IndexToAddr(instr->extra);
176         break;
177
178       case OP_BLEZ:
179         if (registers[instr->rs] <= 0)
180             pcAfter = registers[NextPCReg] + IndexToAddr(instr->extra);
181         break;
182
183       case OP_BLTZAL:
184         registers[R31] = registers[NextPCReg] + 4;
185       case OP_BLTZ:
186         if (registers[instr->rs] & SIGN_BIT)
187             pcAfter = registers[NextPCReg] + IndexToAddr(instr->extra);
188         break;
189
190       case OP_BNE:
191         if (registers[instr->rs] != registers[instr->rt])
192             pcAfter = registers[NextPCReg] + IndexToAddr(instr->extra);
193         break;
```

```
194
195       case OP_DIV:
196         if (registers[instr->rt] == 0) {
197             registers[LoReg] = 0;
198             registers[HiReg] = 0;
199          } else {
200             registers[LoReg] =  registers[instr->rs] / registers[instr->rt];
201             registers[HiReg] = registers[instr->rs] % registers[instr->rt];
202         }
203         break;
204
205       case OP_DIVU:
206           rs = (unsigned int) registers[instr->rs];
207           rt = (unsigned int) registers[instr->rt];
208           if (rt == 0) {
209               registers[LoReg] = 0;
210               registers[HiReg] = 0;
211           } else {
212               tmp = rs / rt;
213               registers[LoReg] = (int) tmp;
214               tmp = rs % rt;
215               registers[HiReg] = (int) tmp;
216           }
217           break;
218
219       case OP_JAL:
220         registers[R31] = registers[NextPCReg] + 4;
221       case OP_J:
222         pcAfter = (pcAfter & 0xf0000000) | IndexToAddr(instr->extra);
223         break;
224
225       case OP_JALR:
226         registers[instr->rd] = registers[NextPCReg] + 4;
227       case OP_JR:
228         pcAfter = registers[instr->rs];
229         break;
230
231       case OP_LB:
232       case OP_LBU:
233         tmp = registers[instr->rs] + instr->extra;
234         if (!machine->ReadMem(tmp, 1, &value))
235             return;
236
237         if ((value & 0x80) && (instr->opCode == OP_LB))
238             value |= 0xffffff00;
239         else
240             value &= 0xff;
241         nextLoadReg = instr->rt;
242         nextLoadValue = value;
243         break;
244
245       case OP_LH:
246       case OP_LHU:
247         tmp = registers[instr->rs] + instr->extra;
248         if (tmp & 0x1) {
249             RaiseException(AddressErrorException, tmp);
250             return;
251         }
```

```
252         if (!machine->ReadMem(tmp, 2, &value))
253             return;
254
255         if ((value & 0x8000) && (instr->opCode == OP_LH))
256             value |= 0xffff0000;
257         else
258             value &= 0xffff;
259         nextLoadReg = instr->rt;
260         nextLoadValue = value;
261         break;
262
263       case OP_LUI:
264         DEBUG('m', "Executing: LUI r%d,%d\n", instr->rt, instr->extra);
265          registers[instr->rt] = instr->extra << 16;
266         break;
267
268       case OP_LW:
269         tmp = registers[instr->rs] + instr->extra;
270         if (tmp & 0x3) {
271             RaiseException(AddressErrorException, tmp);
272             return;
273         }
274         if (!machine->ReadMem(tmp, 4, &value))
275             return;
276         nextLoadReg = instr->rt;
277         nextLoadValue = value;
278         break;
279
280       case OP_LWL:
281         tmp = registers[instr->rs] + instr->extra;
282
283         // ReadMem assumes all 4 byte requests are aligned on an even
284         // word boundary.  Also, the little endian/big endian swap code would
285         // fail (I think) if the other cases are ever exercised.
286         ASSERT((tmp & 0x3) == 0);
287
288         if (!machine->ReadMem(tmp, 4, &value))
289             return;
290         if (registers[LoadReg] == instr->rt)
291             nextLoadValue = registers[LoadValueReg];
292         else
293             nextLoadValue = registers[instr->rt];
294         switch (tmp & 0x3) {
295           case 0:
296             nextLoadValue = value;
297             break;
298           case 1:
299             nextLoadValue = (nextLoadValue & 0xff) | (value << 8);
300             break;
301           case 2:
302             nextLoadValue = (nextLoadValue & 0xffff) | (value << 16);
303             break;
304           case 3:
305             nextLoadValue = (nextLoadValue & 0xffffff) | (value << 24);
306             break;
307         }
308         nextLoadReg = instr->rt;
309         break;
```

```
310
311       case OP_LWR:
312         tmp = registers[instr->rs] + instr->extra;
313
314         // ReadMem assumes all 4 byte requests are aligned on an even
315         // word boundary.  Also, the little endian/big endian swap code would
316         // fail (I think) if the other cases are ever exercised.
317         ASSERT((tmp & 0x3) == 0);
318
319         if (!machine->ReadMem(tmp, 4, &value))
320             return;
321         if (registers[LoadReg] == instr->rt)
322             nextLoadValue = registers[LoadValueReg];
323         else
324             nextLoadValue = registers[instr->rt];
325         switch (tmp & 0x3) {
326           case 0:
327             nextLoadValue = (nextLoadValue & 0xffffff00) |
328                 ((value >> 24) & 0xff);
329             break;
330           case 1:
331              nextLoadValue = (nextLoadValue & 0xffff0000) |
332                 ((value >> 16) & 0xffff);
333             break;
334           case 2:
335             nextLoadValue = (nextLoadValue & 0xff000000)
336                 | ((value >> 8) & 0xffffff);
337             break;
338           case 3:
339             nextLoadValue = value;
340             break;
341         }
342         nextLoadReg = instr->rt;
343         break;
344
345       case OP_MFHI:
346         registers[instr->rd] = registers[HiReg];
347         break;
348
349       case OP_MFLO:
350         registers[instr->rd] = registers[LoReg];
351         break;
352
353       case OP_MTHI:
354         registers[HiReg] = registers[instr->rs];
355         break;
356
357       case OP_MTLO:
358         registers[LoReg] = registers[instr->rs];
359         break;
360
361       case OP_MULT:
362         Mult(registers[instr->rs], registers[instr->rt], TRUE,
363              &registers[HiReg], &registers[LoReg]);
364         break;
365
366       case OP_MULTU:
367         Mult(registers[instr->rs], registers[instr->rt], FALSE,
```

```
368             &registers[HiReg], &registers[LoReg]);
369         break;
370
371       case OP_NOR:
372         registers[instr->rd] = ~(registers[instr->rs] | registers[instr->rt]);
373         break;
374
375       case OP_OR:
376         registers[instr->rd] = registers[instr->rs] | registers[instr->rs];
377         break;
378
379       case OP_ORI:
380         registers[instr->rt] = registers[instr->rs] | (instr->extra & 0xffff);
381         break;
382
383       case OP_SB:
384         if (!machine->WriteMem((unsigned)
385                 (registers[instr->rs] + instr->extra), 1, registers[instr->rt]))
386             return;
387         break;
388
389       case OP_SH:
390         if (!machine->WriteMem((unsigned)
391                 (registers[instr->rs] + instr->extra), 2, registers[instr->rt]))
392             return;
393         break;
394
395       case OP_SLL:
396         registers[instr->rd] = registers[instr->rt] << instr->extra;
397          break;
398
399       case OP_SLLV:
400         registers[instr->rd] = registers[instr->rt] <<
401             (registers[instr->rs] & 0x1f);
402         break;
403
404       case OP_SLT:
405         if (registers[instr->rs] < registers[instr->rt])
406             registers[instr->rd] = 1;
407         else
408             registers[instr->rd] = 0;
409         break;
410
411       case OP_SLTI:
412         if (registers[instr->rs] < instr->extra)
413             registers[instr->rt] = 1;
414         else
415             registers[instr->rt] = 0;
416         break;
417
418       case OP_SLTIU:
419         rs = registers[instr->rs];
420         imm = instr->extra;
421         if (rs < imm)
422             registers[instr->rt] = 1;
423         else
424             registers[instr->rt] = 0;
425         break;
```

```
426
427       case OP_SLTU:
428         rs = registers[instr->rs];
429         rt = registers[instr->rt];
430         if (rs < rt)
431             registers[instr->rd] = 1;
432         else
433             registers[instr->rd] = 0;
434         break;
435
436       case OP_SRA:
437         registers[instr->rd] = registers[instr->rt] >> instr->extra;
438         break;
439
440       case OP_SRAV:
441         registers[instr->rd] = registers[instr->rt] >>
442             (registers[instr->rs] & 0x1f);
443         break;
444
445       case OP_SRL:
446         tmp = registers[instr->rt];
447         tmp >>= instr->extra;
448         registers[instr->rd] = tmp;
449         break;
450
451       case OP_SRLV:
452         tmp = registers[instr->rt];
453         tmp >>= (registers[instr->rs] & 0x1f);
454         registers[instr->rd] = tmp;
455         break;
456
457       case OP_SUB:
458         diff = registers[instr->rs] - registers[instr->rt];
459         if (((registers[instr->rs] ^ registers[instr->rt]) & SIGN_BIT) &&
460             ((registers[instr->rs] ^ diff) & SIGN_BIT)) {
461             RaiseException(OverflowException, 0);
462             return;
463          }
464         registers[instr->rd] = diff;
465         break;
466
467       case OP_SUBU:
468         registers[instr->rd] = registers[instr->rs] - registers[instr->rt];
469         break;
470
471       case OP_SW:
472         if (!machine->WriteMem((unsigned)
473                 (registers[instr->rs] + instr->extra), 4, registers[instr->rt]))
474             return;
475         break;
476
477       case OP_SWL:
478         tmp = registers[instr->rs] + instr->extra;
479
480         // The little endian/big endian swap code would
481         // fail (I think) if the other cases are ever exercised.
482         ASSERT((tmp & 0x3) == 0);
483
```

```
484         if (!machine->ReadMem((tmp & ~0x3), 4, &value))
485             return;
486         switch (tmp & 0x3) {
487           case 0:
488             value = registers[instr->rt];
489             break;
490           case 1:
491             value = (value & 0xff000000) | ((registers[instr->rt] >> 8) &
492                                             0xffffff);
493             break;
494           case 2:
495             value = (value & 0xffff0000) | ((registers[instr->rt] >> 16) &
496                                             0xffff);
497             break;
498           case 3:
499             value = (value & 0xffffff00) | ((registers[instr->rt] >> 24) &
500                                             0xff);
501             break;
502         }
503         if (!machine->WriteMem((tmp & ~0x3), 4, value))
504             return;
505         break;
506 
507       case OP_SWR:
508         tmp = registers[instr->rs] + instr->extra;
509 
510         // The little endian/big endian swap code would
511         // fail (I think) if the other cases are ever exercised.
512         ASSERT((tmp & 0x3) == 0);
513 
514         if (!machine->ReadMem((tmp & ~0x3), 4, &value))
515             return;
516         switch (tmp & 0x3) {
517           case 0:
518             value = (value & 0xffffff) | (registers[instr->rt] << 24);
519             break;
520           case 1:
521             value = (value & 0xffff) | (registers[instr->rt] << 16);
522             break;
523           case 2:
524             value = (value & 0xff) | (registers[instr->rt] << 8);
525             break;
526           case 3:
527             value = registers[instr->rt];
528             break;
529          }
530         if (!machine->WriteMem((tmp & ~0x3), 4, value))
531             return;
532         break;
533 
534       case OP_SYSCALL:
535         RaiseException(SyscallException, 0);
536         return;
537 
538       case OP_XOR:
539         registers[instr->rd] = registers[instr->rs] ^ registers[instr->rt];
540         break;
541 
```

```
542       case OP_XORI:
543         registers[instr->rt] = registers[instr->rs] ^ (instr->extra & 0xffff);
544         break;
545
546       case OP_RES:
547       case OP_UNIMP:
548         RaiseException(IllegalInstrException, 0);
549         return;
550
551       default:
552         ASSERT(FALSE);
553     }
554
555     // Now we have successfully executed the instruction.
556
557     // Do any delayed load operation
558     DelayedLoad(nextLoadReg, nextLoadValue);
559
560     // Advance program counters.
561     registers[PrevPCReg] = registers[PCReg];    // for debugging, in case we
562                                                 // are jumping into lala-land
563     registers[PCReg] = registers[NextPCReg];
564     registers[NextPCReg] = pcAfter;
565 }
566
567 //----------------------------------------------------------------------
568 // Machine::DelayedLoad
569 //      Simulate effects of a delayed load.
570 //
571 //      NOTE -- RaiseException/CheckInterrupts must also call DelayedLoad,
572 //      since any delayed load must get applied before we trap to the kernel.
573 //----------------------------------------------------------------------
574
575 void
576 Machine::DelayedLoad(int nextReg, int nextValue)
577 {
578     registers[registers[LoadReg]] = registers[LoadValueReg];
579     registers[LoadReg] = nextReg;
580     registers[LoadValueReg] = nextValue;
581     registers[0] = 0;   // and always make sure R0 stays zero.
582 }
583
584 //----------------------------------------------------------------------
585 // Instruction::Decode
586 //      Decode a MIPS instruction
587 //----------------------------------------------------------------------
588
589 void
590 Instruction::Decode()
591 {
592     OpInfo *opPtr;
593
594     rs = (value >> 21) & 0x1f;
595     rt = (value >> 16) & 0x1f;
596     rd = (value >> 11) & 0x1f;
597     opPtr = &opTable[(value >> 26) & 0x3f];
598     opCode = opPtr->opCode;
599     if (opPtr->format == IFMT) {
```

```
600         extra = value & 0xffff;
601         if (extra & 0x8000) {
602            extra |= 0xffff0000;
603         }
604     } else if (opPtr->format == RFMT) {
605         extra = (value >> 6) & 0x1f;
606     } else {
607         extra = value & 0x3ffffff;
608     }
609     if (opCode == SPECIAL) {
610         opCode = specialTable[value & 0x3f];
611     } else if (opCode == BCOND) {
612         int i = value & 0x1f0000;
613 
614         if (i == 0) {
615             opCode = OP_BLTZ;
616         } else if (i == 0x10000) {
617             opCode = OP_BGEZ;
618         } else if (i == 0x100000) {
619             opCode = OP_BLTZAL;
620         } else if (i == 0x110000) {
621             opCode = OP_BGEZAL;
622         } else {
623             opCode = OP_UNIMP;
624         }
625     }
626 }
627 
628 //----------------------------------------------------------------------
629 // Mult
630 //      Simulate R2000 multiplication.
631 //      The words at *hiPtr and *loPtr are overwritten with the
632 //      double-length result of the multiplication.
633 //----------------------------------------------------------------------
634 
635 static void
636 Mult(int a, int b, bool signedArith, int* hiPtr, int* loPtr)
637 {
638     if ((a == 0) || (b == 0)) {
639         *hiPtr = *loPtr = 0;
640         return;
641     }
642 
643     // Compute the sign of the result, then make everything positive
644     // so unsigned computation can be done in the main loop.
645     bool negative = FALSE;
646     if (signedArith) {
647         if (a < 0) {
648             negative = (bool)!negative;
649             a = -a;
650         }
651         if (b < 0) {
652             negative = (bool)!negative;
653             b = -b;
654         }
655     }
656 
657     // Compute the result in unsigned arithmetic (check a's bits one at
```

```
658     // a time, and add in a shifted value of b).
659     unsigned int bLo = b;
660     unsigned int bHi = 0;
661     unsigned int lo = 0;
662     unsigned int hi = 0;
663     for (int i = 0; i < 32; i++) {
664         if (a & 1) {
665             lo += bLo;
666             if (lo < bLo)  // Carry out of the low bits?
667                 hi += 1;
668             hi += bHi;
669             if ((a & 0xfffffffe) == 0)
670                 break;
671         }
672         bHi <<= 1;
673         if (bLo & 0x80000000)
674             bHi |= 1;
675 
676         bLo <<= 1;
677         a >>= 1;
678     }
679 
680     // If the result is supposed to be negative, compute the two's
681     // complement of the double-word result.
682     if (negative) {
683         hi = ~hi;
684         lo = ~lo;
685         lo++;
686         if (lo == 0)
687             hi++;
688     }
689 
690     *hiPtr = (int) hi;
691     *loPtr = (int) lo;
692 }
```

3.11 network.h

```
 1 // network.h
 2 //      Data structures to emulate a physical network connection.
 3 //      The network provides the abstraction of unordered, unreliable,
 4 //      fixed-size packet delivery to other machines on the network.
 5 //
 6 //      You may note that the interface to the network is similar to
 7 //      the console device -- both are full duplex channels.
 8 //
 9 //  DO NOT CHANGE -- part of the machine emulation
10 //
11 // Copyright (c) 1992-1993 The Regents of the University of California.
12 // All rights reserved.  See copyright.h for copyright notice and limitation
13 // of liability and disclaimer of warranty provisions.
14 //
15 // Modifications:
16 //
17 //   Date: July, 1995
18 //   Author: K. Salem
19 //   Description:  added packet delays so that packet delivery is
```

```
20 //                 not guaranteed to be ordered
21 //
22
23 #ifndef NETWORK_H
24 #define NETWORK_H
25
26 #include "copyright.h"
27 #include "utility.h"
28
29 // Network address -- uniquely identifies a machine.  This machine's ID
30 //  is given on the command line.
31 typedef int NetworkAddress;
32
33 // The following class defines the network packet header.
34 // The packet header is prepended to the data payload by the Network driver,
35 // before the packet is sent over the wire.  The format on the wire is:
36 //      packet header (PacketHeader)
37 //      data (containing MailHeader from the PostOffice!)
38
39 class PacketHeader {
40   public:
41     NetworkAddress to;          // Destination machine ID
42     NetworkAddress from;        // source machine ID
43     unsigned length;            // bytes of packet data, excluding the
44                                 // packet header (but including the
45                                 // MailHeader prepended by the post office)
46 };
47
48 #define MaxWireSize     64      // largest packet that can go out on the wire
49 #define MaxPacketSize   (MaxWireSize - sizeof(struct PacketHeader))
50                                 // data "payload" of the largest packet
51
52
53 // The following class defines a physical network device.  The network
54 // is capable of delivering fixed sized packets
55 // to other machines connected to the network.
56 // Packet delivery is neither ordered nor reliable.
57 //
58 // The "reliability" of the network can be specified to the constructor.
59 // This number, between 0 and 1, is the chance that the network will not
60 // lose a packet.
61 //
62 // The "orderability" of the network can also be specified to the constructor.
63 // This number, between 0 and 1, is the chance that a packet will not be
64 // delayed in the network, *given that it is not lost*.  A delayed
65 // packet will be delivered eventually.  However, other packets that
66 // are sent to the same destination after the delayed packet may arrive
67 // at the destination before the delayed packet.
68 //
69 // The orderability parameter is used only for packets that are not lost.
70 // So, if reliability is set to 0.9 and orderability is set to 0.9, then
71 // there is an 81% chance that any packet will be sent without loss and
72 // without delay.  There is a 10% chance it will be lost, and a 9%
73 // chance that it will be delayed.
74 //
75 // Note that you can change the seed for the random number
76 // generator, by changing the arguments to RandomInit() in Initialize().
77 // The random number generator is used to choose which packets to drop
```

```
78 // or delay.
79
80 class Network {
81   public:
82     Network(NetworkAddress addr, double reliability, double orderability,
83           VoidFunctionPtr readAvail, VoidFunctionPtr writeDone, _int callArg);
84                                 // Allocate and initialize network driver
85     ~Network();                 // De-allocate the network driver data
86
87     void Send(PacketHeader hdr, char* data);
88                                 // Send the packet data to a remote machine,
89                                 // specified by "hdr".  Returns after a
90                                 // successful send.
91                                 // "writeHandler" is invoked once the next
92                                 // packet can be sent.  Note that writeHandler
93                                 // is called whether or not the packet is
94                                 // dropped, and note that the "from" field of
95                                 // the PacketHeader is filled in automatically
96                                 // by Send().
97
98     PacketHeader Receive(char* data);
99                                 // Poll the network for incoming messages.
100                                // If there is a packet waiting, copy the
101                                // packet into "data" and return the header.
102                                // If no packet is waiting, return a header
103                                // with length 0.
104
105    void SendDone();            // Interrupt handler, called when message is
106                                // sent
107    void CheckPktAvail();       // Check if there is an incoming packet
108
109  private:
110    NetworkAddress ident;       // This machine's network address
111    double chanceToWork;        // Likelihood packet will not be dropped
112    double chanceToNotDelay;       // Likelihood packet will not be delayed
113    int sock;                   // UNIX socket number for incoming packets
114    char sockName[32];          // File name corresponding to UNIX socket
115    VoidFunctionPtr writeHandler; // Interrupt handler, signalling next packet
116                                //      can be sent.
117    VoidFunctionPtr readHandler;  // Interrupt handler, signalling packet has
118                                //      arrived.
119    _int handlerArg;            // Argument to be passed to interrupt handler
120                                //   (pointer to post office)
121    bool sendBusy;              // Packet is being sent.
122    bool packetAvail;           // Packet has arrived, can be pulled off of
123                                //   network
124    PacketHeader inHdr;         // Information about arrived packet
125    char inbox[MaxPacketSize];  // Data for arrived packet
126    char delayBuf[MaxWireSize];  // Place to save a delayed packet
127    char delayToName[32];       // Place to send delayed packet, eventually
128    bool delayBufFull;          // Is delayBuf in use?
129 };
130
131 #endif // NETWORK_H
```

3.12 network.cc

```
 1 // network.cc
 2 //      Routines to simulate a network interface, using UNIX sockets
 3 //      to deliver packets between multiple invocations of nachos.
 4 //
 5 //  DO NOT CHANGE -- part of the machine emulation
 6 //
 7 // Copyright (c) 1992-1993 The Regents of the University of California.
 8 // All rights reserved.  See copyright.h for copyright notice and limitation
 9 // of liability and disclaimer of warranty provisions.
10 // Modifications:
11 //
12 //   Date: July, 1995
13 //   Author: K. Salem
14 //   Description:  added packet delays so that packet delivery is
15 //                  not guaranteed to be ordered
16 //
17
18 #include "copyright.h"
19 #include "system.h"
20
21 // Dummy functions because C++ can't call member functions indirectly
22 static void NetworkReadPoll(_int arg)
23 { Network *net = (Network *)arg; net->CheckPktAvail(); }
24 static void NetworkSendDone(_int arg)
25 { Network *net = (Network *)arg; net->SendDone(); }
26
27 // Initialize the network emulation
28 //   addr is used to generate the socket name
29 //   reliability says whether we drop packets to emulate unreliable links
30 //   readAvail, writeDone, callArg -- analogous to console
31 Network::Network(NetworkAddress addr, double reliability, double orderability,
32         VoidFunctionPtr readAvail, VoidFunctionPtr writeDone, _int callArg)
33 {
34     ident = addr;
35     if (reliability < 0) chanceToWork = 0;
36     else if (reliability > 1) chanceToWork = 1;
37     else chanceToWork = reliability;
38
39     if (orderability < 0) chanceToNotDelay = 0;
40     else if (orderability > 1) chanceToNotDelay = 1;
41     else chanceToNotDelay = orderability;
42
43     // set up the stuff to emulate asynchronous interrupts
44     writeHandler = writeDone;
45     readHandler = readAvail;
46     handlerArg = callArg;
47     sendBusy = FALSE;
48     inHdr.length = 0;
49     delayBufFull = FALSE;
50
51     sock = OpenSocket();
52     sprintf(sockName, "SOCKET_%d", (int)addr);
53     AssignNameToSocket(sockName, sock);          // Bind socket to a filename
54                                                  // in the current directory.
55
56     // start polling for incoming packets
```

```
57     interrupt->Schedule(NetworkReadPoll, (_int)this, NetworkTime, NetworkRecvInt);
58 }
59
60 Network::~Network()
61 {
62     CloseSocket(sock);
63     DeAssignNameToSocket(sockName);
64 }
65
66 // if a packet is already buffered, we simply delay reading
67 // the incoming packet.  In real life, the incoming
68 // packet might be dropped if we can't read it in time.
69 void
70 Network::CheckPktAvail()
71 {
72     // schedule the next time to poll for a packet
73     interrupt->Schedule(NetworkReadPoll, (_int)this, NetworkTime, NetworkRecvInt);
74
75     if (inHdr.length != 0)      // do nothing if packet is already buffered
76         return;
77     if (!PollSocket(sock))      // do nothing if no packet to be read
78         return;
79
80     // otherwise, read packet in
81     char *buffer = new char[MaxWireSize];
82     ReadFromSocket(sock, buffer, MaxWireSize);
83
84     // divide packet into header and data
85     inHdr = *(PacketHeader *)buffer;
86     ASSERT((inHdr.to == ident) && (inHdr.length <= MaxPacketSize));
87     bcopy(buffer + sizeof(PacketHeader), inbox, inHdr.length);
88     delete []buffer ;
89
90     DEBUG('n', "Network received packet from %d, length %d...\n",
91                                         (int) inHdr.from, inHdr.length);
92     stats->numPacketsRecvd++;
93
94     // tell post office that the packet has arrived
95     (*readHandler)(handlerArg);
96 }
97
98 // notify user that another packet can be sent
99 void
100 Network::SendDone()
101 {
102     sendBusy = FALSE;
103     stats->numPacketsSent++;
104     (*writeHandler)(handlerArg);
105 }
106
107 // send a packet by concatenating hdr and data, and schedule
108 // an interrupt to tell the user when the next packet can be sent
109 //
110 // Note we always pad out a packet to MaxWireSize before putting it into
111 // the socket, because it's simpler at the receive end.
112 void
113 Network::Send(PacketHeader hdr, char* data)
114 {
```

```
115     char toName[32];
116
117     ASSERT((sendBusy == FALSE) && (hdr.length > 0)
118                 && (hdr.length <= MaxPacketSize) && (hdr.from == ident));
119     DEBUG('n', "Sending to addr %d, %d bytes... ", hdr.to, hdr.length);
120
121     interrupt->Schedule(NetworkSendDone, (_int)this, NetworkTime, NetworkSendInt);
122
123     if (Random() % 100 >= chanceToWork * 100) { // emulate a lost packet
124         DEBUG('n', "oops, lost it!\n");
125         return;
126     }
127     if (Random() % 100 >= chanceToNotDelay * 100) { // emulate delay
128       // to delay a packet, we simply save it in a buffer
129       // it remains there until another packet is delayed, at which
130       //  point we send it out
131       if (delayBufFull == TRUE) {
132         SendToSocket(sock, delayBuf, MaxWireSize, delayToName);
133       }
134       sprintf(delayToName, "SOCKET_%d", (int)hdr.to);
135       *(PacketHeader *)delayBuf = hdr;
136       bcopy(data, delayBuf + sizeof(PacketHeader), hdr.length);
137       delayBufFull = TRUE;
138       return;
139     }
140
141     // packet is neither lost nor delayed - send it now
142
143     sprintf(toName, "SOCKET_%d", (int)hdr.to);
144     // concatenate hdr and data into a single buffer, and send it out
145     char *buffer = new char[MaxWireSize];
146     *(PacketHeader *)buffer = hdr;
147     bcopy(data, buffer + sizeof(PacketHeader), hdr.length);
148     SendToSocket(sock, buffer, MaxWireSize, toName);
149     delete []buffer;
150 }
151
152 // read a packet, if one is buffered
153 PacketHeader
154 Network::Receive(char* data)
155 {
156     PacketHeader hdr = inHdr;
157
158     inHdr.length = 0;
159     if (hdr.length != 0)
160         bcopy(inbox, data, hdr.length);
161     return hdr;
162 }
```

3.13 stats.h

```
1 // stats.h
2 //      Data structures for gathering statistics about Nachos performance.
3 //
4 // DO NOT CHANGE -- these stats are maintained by the machine emulation
5 //
6 //
```

```
 7 // Copyright (c) 1992-1993 The Regents of the University of California.
 8 // All rights reserved.  See copyright.h for copyright notice and limitation
 9 // of liability and disclaimer of warranty provisions.
10
11 #ifndef STATS_H
12 #define STATS_H
13
14 #include "copyright.h"
15
16 // The following class defines the statistics that are to be kept
17 // about Nachos behavior -- how much time (ticks) elapsed, how
18 // many user instructions executed, etc.
19 //
20 // The fields in this class are public to make it easier to update.
21
22 class Statistics {
23   public:
24     int totalTicks;             // Total time running Nachos
25     int idleTicks;              // Time spent idle (no threads to run)
26     int systemTicks;            // Time spent executing system code
27     int userTicks;              // Time spent executing user code
28                                 // (this is also equal to # of
29                                 // user instructions executed)
30
31     int numDiskReads;           // number of disk read requests
32     int numDiskWrites;          // number of disk write requests
33     int numConsoleCharsRead;    // number of characters read from the keyboard
34     int numConsoleCharsWritten; // number of characters written to the display
35     int numPageFaults;          // number of virtual memory page faults
36     int numPacketsSent;         // number of packets sent over the network
37     int numPacketsRecvd;        // number of packets received over the network
38
39     Statistics();               // initialize everything to zero
40
41     void Print();               // print collected statistics
42 };
43
44 // Constants used to reflect the relative time an operation would
45 // take in a real system.  A "tick" is a just a unit of time -- if you
46 // like, a microsecond.
47 //
48 // Since Nachos kernel code is directly executed, and the time spent
49 // in the kernel measured by the number of calls to enable interrupts,
50 // these time constants are none too exact.
51
52 #define UserTick        1       // advance for each user-level instruction
53 #define SystemTick      10      // advance each time interrupts are enabled
54 #define RotationTime    500     // time disk takes to rotate one sector
55 #define SeekTime        500     // time disk takes to seek past one track
56 #define ConsoleTime     100     // time to read or write one character
57 #define NetworkTime     100     // time to send or receive one packet
58 #define TimerTicks      100     // (average) time between timer interrupts
59
60 #endif // STATS_H
```

3.14 stats.cc

```
 1 // stats.h
 2 //      Routines for managing statistics about Nachos performance.
 3 //
 4 // DO NOT CHANGE -- these stats are maintained by the machine emulation.
 5 //
 6 // Copyright (c) 1992-1993 The Regents of the University of California.
 7 // All rights reserved.  See copyright.h for copyright notice and limitation
 8 // of liability and disclaimer of warranty provisions.
 9
10 #include "copyright.h"
11 #include "utility.h"
12 #include "stats.h"
13
14 //----------------------------------------------------------------------
15 // Statistics::Statistics
16 //      Initialize performance metrics to zero, at system startup.
17 //----------------------------------------------------------------------
18
19 Statistics::Statistics()
20 {
21     totalTicks = idleTicks = systemTicks = userTicks = 0;
22     numDiskReads = numDiskWrites = 0;
23     numConsoleCharsRead = numConsoleCharsWritten = 0;
24     numPageFaults = numPacketsSent = numPacketsRecvd = 0;
25 }
26
27 //----------------------------------------------------------------------
28 // Statistics::Print
29 //      Print performance metrics, when we've finished everything
30 //      at system shutdown.
31 //----------------------------------------------------------------------
32
33 void
34 Statistics::Print()
35 {
36     printf("Ticks: total %d, idle %d, system %d, user %d\n", totalTicks,
37         idleTicks, systemTicks, userTicks);
38     printf("Disk I/O: reads %d, writes %d\n", numDiskReads, numDiskWrites);
39     printf("Console I/O: reads %d, writes %d\n", numConsoleCharsRead,
40         numConsoleCharsWritten);
41     printf("Paging: faults %d\n", numPageFaults);
42     printf("Network I/O: packets received %d, sent %d\n", numPacketsRecvd,
43         numPacketsSent);
44 }
```

3.15 sysdep.h

```
 1 // sysdep.h
 2 //      System-dependent interface.  Nachos uses the routines defined
 3 //      here, rather than directly calling the UNIX library functions, to
 4 //      simplify porting between versions of UNIX, and even to
 5 //      other systems, such as MSDOS and the Macintosh.
 6 //
 7 // Copyright (c) 1992-1993 The Regents of the University of California.
 8 // All rights reserved.  See copyright.h for copyright notice and limitation
```

```
 9 // of liability and disclaimer of warranty provisions.
10
11 #ifndef SYSDEP_H
12 #define SYSDEP_H
13
14 #include "copyright.h"
15
16 // Check file to see if there are any characters to be read.
17 // If no characters in the file, return without waiting.
18 extern bool PollFile(int fd);
19
20 // File operations: open/read/write/lseek/close, and check for error
21 // For simulating the disk and the console devices.
22 extern int OpenForWrite(char *name);
23 extern int OpenForReadWrite(char *name, bool crashOnError);
24 extern void Read(int fd, char *buffer, int nBytes);
25 extern int ReadPartial(int fd, char *buffer, int nBytes);
26 extern void WriteFile(int fd, char *buffer, int nBytes);
27 extern void Lseek(int fd, int offset, int whence);
28 extern int Tell(int fd);
29 extern void Close(int fd);
30 //extern bool Unlink(char *name);
31 extern int Unlink(char *name);
32
33 // Interprocess communication operations, for simulating the network
34 extern int OpenSocket();
35 extern void CloseSocket(int sockID);
36 extern void AssignNameToSocket(char *socketName, int sockID);
37 extern void DeAssignNameToSocket(char *socketName);
38 extern bool PollSocket(int sockID);
39 extern void ReadFromSocket(int sockID, char *buffer, int packetSize);
40 extern void SendToSocket(int sockID, char *buffer, int packetSize,char *toName);
41
42 // Process control: abort, exit, and sleep
43 extern void Abort();
44 extern void Exit(int exitCode);
45 extern void Delay(int seconds);
46
47 // Initialize system so that cleanUp routine is called when user hits ctl-C
48 extern void CallOnUserAbort(VoidNoArgFunctionPtr cleanUp);
49
50 // Initialize the pseudo random number generator
51 extern void RandomInit(unsigned seed);
52 extern int Random();
53
54 // Allocate, de-allocate an array, such that de-referencing
55 // just beyond either end of the array will cause an error
56 extern char *AllocBoundedArray(int size);
57 extern void DeallocBoundedArray(char *p, int size);
58
59 // Other C library routines that are used by Nachos.
60 // These are assumed to be portable, so we don't include a wrapper.
61 extern "C" {
62 int atoi(const char *str);
63 double atof(const char *str);
64 int abs(int i);
65
66 #include <stdio.h>              // for printf, fprintf
```

```
67 #include <string.h>              // for DEBUG, etc.
68 }
69
70 #endif // SYSDEP_H
```

3.16 sysdep.cc

```
 1 // sysdep.cc
 2 //      Implementation of system-dependent interface.  Nachos uses the
 3 //      routines defined here, rather than directly calling the UNIX library,
 4 //      to simplify porting between versions of UNIX, and even to
 5 //      other systems, such as MSDOS.
 6 //
 7 //      On UNIX, almost all of these routines are simple wrappers
 8 //      for the underlying UNIX system calls.
 9 //
10 //      NOTE: all of these routines refer to operations on the underlying
11 //      host machine (e.g., the DECstation, SPARC, etc.), supporting the
12 //      Nachos simulation code.  Nachos implements similar operations,
13 //      (such as opening a file), but those are implemented in terms
14 //      of hardware devices, which are simulated by calls to the underlying
15 //      routines in the host workstation OS.
16 //
17 //      This file includes lots of calls to C routines.  C++ requires
18 //      us to wrap all C definitions with a "extern "C" block".
19 //      This prevents the internal forms of the names from being
20 //      changed by the C++ compiler.
21 //
22 // Copyright (c) 1992-1993 The Regents of the University of California.
23 // All rights reserved.  See copyright.h for copyright notice and limitation
24 // of liability and disclaimer of warranty provisions.
25
26 #include "copyright.h"
27
28 extern "C" {
29 #include <stdio.h>
30 #include <string.h>
31 #include <signal.h>
32 #include <sys/types.h>
33 #include <sys/time.h>
34 #include <sys/socket.h>
35 #include <sys/file.h>
36 #include <sys/un.h>
37 #include <sys/mman.h>
38 #include <sys/errno.h>
39 #ifdef HOST_i386
40 #include <sys/time.h>
41 #endif
42 #ifdef HOST_SPARC
43 #include <sys/time.h>
44 #endif
45 #ifdef HOST_ALPHA
46 #include <sys/time.h>
47 #endif
48
49 // UNIX routines called by procedures in this file
50
```

```
51 #ifdef HOST_SNAKE
52 // int creat(char *name, unsigned short mode);
53 // int open(const char *name, int flags, ...);
54 #else
55 #ifndef HOST_ALPHA
56 #ifndef HOST_LINUX
57 int creat(const char *name, unsigned short mode);
58 int open(const char *name, int flags, ...);
59 #endif
60 #endif
61 // void signal(int sig, VoidFunctionPtr func); -- this may work now!
62 #if defined(HOST_i386) || defined(HOST_ALPHA)
63 int select(int nfds, fd_set *readfds, fd_set *writefds, fd_set *exceptfds,
64              struct timeval *timeout);
65 #else
66 int select(int numBits, void *readFds, void *writeFds, void *exceptFds,
67          struct timeval *timeout);
68 #endif
69 #endif
70
71 int unlink(char *name);
72 int read(int filedes, char *buf, int numBytes);
73 int write(int filedes, char *buf, int numBytes);
74 int lseek(int filedes, int offset, int whence);
75 int tell(int filedes);
76 int close(int filedes);
77 int unlink(char *name);
78
79 // definition varies slightly from platform to platform, so don't
80 // define unless gcc complains
81 //extern int recvfrom(int s, void *buf, int len, int flags, void *from, int *fromlen);
82 //extern int sendto(int s, void *msg, int len, int flags, void *to, int tolen);
83
84
85 void srand(unsigned seed);
86 int rand(void);
87 unsigned sleep(unsigned);
88 void abort();
89 void exit();
90 int getpagesize();
91
92 #ifndef HOST_ALPHA
93 #ifndef HOST_LINUX
94 int mprotect(char *addr, int len, int prot);
95
96 int socket(int, int, int);
97 int bind (int, const void*, int);
98 int recvfrom (int, void*, int, int, void*, int *);
99 int sendto (int, const void*, int, int, void*, int);
100 #endif
101 #endif
102 }
103
104 #include "interrupt.h"
105 #include "system.h"
106
107 //----------------------------------------------------------------------
108 // PollFile
```

```
109 //      Check open file or open socket to see if there are any
110 //      characters that can be read immediately.  If so, read them
111 //      in, and return TRUE.
112 //
113 //      In the network case, if there are no threads for us to run,
114 //      and no characters to be read,
115 //      we need to give the other side a chance to get our host's CPU
116 //      (otherwise, we'll go really slowly, since UNIX time-slices
117 //      infrequently, and this would be like busy-waiting).  So we
118 //      delay for a short fixed time, before allowing ourselves to be
119 //      re-scheduled (sort of like a Yield, but cast in terms of UNIX).
120 //
121 //      "fd" -- the file descriptor of the file to be polled
122 //----------------------------------------------------------------------
123
124 bool
125 PollFile(int fd)
126 {
127     int rfd = (1 << fd), wfd = 0, xfd = 0, retVal;
128     struct timeval pollTime;
129
130 // decide how long to wait if there are no characters on the file
131     pollTime.tv_sec = 0;
132     if (interrupt->getStatus() == IdleMode)
133         pollTime.tv_usec = 20000;               // delay to let other nachos run
134     else
135         pollTime.tv_usec = 0;                   // no delay
136
137 // poll file or socket
138 #if defined(HOST_i386) || defined(HOST_ALPHA)
139     retVal = select(32, (fd_set*)&rfd, (fd_set*)&wfd, (fd_set*)&xfd, &pollTime);
140 #else
141     retVal = select(32, &rfd, &wfd, &xfd, &pollTime);
142 #endif
143
144     ASSERT((retVal == 0) || (retVal == 1));
145     if (retVal == 0)
146         return FALSE;                           // no char waiting to be read
147     return TRUE;
148 }
149
150 //----------------------------------------------------------------------
151 // OpenForWrite
152 //      Open a file for writing.  Create it if it doesn't exist; truncate it
153 //      if it does already exist.  Return the file descriptor.
154 //
155 //      "name" -- file name
156 //----------------------------------------------------------------------
157
158 int
159 OpenForWrite(char *name)
160 {
161     int fd = open(name, O_RDWR|O_CREAT|O_TRUNC, 0666);
162
163     ASSERT(fd >= 0);
164     return fd;
165 }
166
```

```
167 //----------------------------------------------------------------------
168 // OpenForReadWrite
169 //      Open a file for reading or writing.
170 //      Return the file descriptor, or error if it doesn't exist.
171 //
172 //      "name" -- file name
173 //----------------------------------------------------------------------
174
175 int
176 OpenForReadWrite(char *name, bool crashOnError)
177 {
178     int fd = open(name, O_RDWR, 0);
179
180     ASSERT(!crashOnError || fd >= 0);
181     return fd;
182 }
183
184 //----------------------------------------------------------------------
185 // Read
186 //      Read characters from an open file.  Abort if read fails.
187 //----------------------------------------------------------------------
188
189 void
190 Read(int fd, char *buffer, int nBytes)
191 {
192     int retVal = read(fd, buffer, nBytes);
193     ASSERT(retVal == nBytes);
194 }
195
196 //----------------------------------------------------------------------
197 // ReadPartial
198 //      Read characters from an open file, returning as many as are
199 //       available.
200 //----------------------------------------------------------------------
201
202 int
203 ReadPartial(int fd, char *buffer, int nBytes)
204 {
205     return read(fd, buffer, nBytes);
206 }
207
208
209 //----------------------------------------------------------------------
210 // WriteFile
211 //      Write characters to an open file.  Abort if write fails.
212 //----------------------------------------------------------------------
213
214 void
215 WriteFile(int fd, char *buffer, int nBytes)
216 {
217     int retVal = write(fd, buffer, nBytes);
218     ASSERT(retVal == nBytes);
219 }
220
221 //----------------------------------------------------------------------
222 // Lseek
223 //      Change the location within an open file.  Abort on error.
224 //----------------------------------------------------------------------
```

```
225
226 void
227 Lseek(int fd, int offset, int whence)
228 {
229     int retVal = lseek(fd, offset, whence);
230     ASSERT(retVal >= 0);
231 }
232
233 //----------------------------------------------------------------------
234 // Tell
235 //      Report the current location within an open file.
236 //----------------------------------------------------------------------
237
238 int
239 Tell(int fd)
240 {
241 #ifdef HOST_i386
242     return lseek(fd,0,SEEK_CUR); // 386BSD doesn't have the tell() system call
243 #else
244     return tell(fd);
245 #endif
246 }
247
248
249 //----------------------------------------------------------------------
250 // Close
251 //      Close a file.  Abort on error.
252 //----------------------------------------------------------------------
253
254 void
255 Close(int fd)
256 {
257     int retVal = close(fd);
258     ASSERT(retVal >= 0);
259 }
260
261 //----------------------------------------------------------------------
262 // Unlink
263 //      Delete a file.
264 //----------------------------------------------------------------------
265
266 // bool
267 int
268 Unlink(char *name)
269 {
270     return (bool)unlink(name);
271 }
272
273 //----------------------------------------------------------------------
274 // OpenSocket
275 //      Open an interprocess communication (IPC) connection.  For now,
276 //      just open a datagram port where other Nachos (simulating
277 //      workstations on a network) can send messages to this Nachos.
278 //----------------------------------------------------------------------
279
280 int
281 OpenSocket()
282 {
```

```
283     int sockID;
284
285     sockID = socket(AF_UNIX, SOCK_DGRAM, 0);
286     ASSERT(sockID >= 0);
287
288     return sockID;
289 }
290
291 //----------------------------------------------------------------------
292 // CloseSocket
293 //      Close the IPC connection.
294 //----------------------------------------------------------------------
295
296 void
297 CloseSocket(int sockID)
298 {
299     (void) close(sockID);
300 }
301
302 //----------------------------------------------------------------------
303 // InitSocketName
304 //      Initialize a UNIX socket address -- magical!
305 //----------------------------------------------------------------------
306
307 static void
308 InitSocketName(struct sockaddr_un *uname, char *name)
309 {
310     uname->sun_family = AF_UNIX;
311     strcpy(uname->sun_path, name);
312 }
313
314 //----------------------------------------------------------------------
315 // AssignNameToSocket
316 //      Give a UNIX file name to the IPC port, so other instances of Nachos
317 //      can locate the port.
318 //----------------------------------------------------------------------
319
320 void
321 AssignNameToSocket(char *socketName, int sockID)
322 {
323     struct sockaddr_un uName;
324     int retVal;
325
326     (void) unlink(socketName);    // in case it's still around from last time
327
328     InitSocketName(&uName, socketName);
329     retVal = bind(sockID, (struct sockaddr *) &uName, sizeof(uName));
330     ASSERT(retVal >= 0);
331     DEBUG('n', "Created socket %s\n", socketName);
332 }
333
334 //----------------------------------------------------------------------
335 // DeAssignNameToSocket
336 //      Delete the UNIX file name we assigned to our IPC port, on cleanup.
337 //----------------------------------------------------------------------
338 void
339 DeAssignNameToSocket(char *socketName)
340 {
```

```
341     (void) unlink(socketName);
342 }
343
344 //----------------------------------------------------------------------
345 // PollSocket
346 //      Return TRUE if there are any messages waiting to arrive on the
347 //      IPC port.
348 //----------------------------------------------------------------------
349 bool
350 PollSocket(int sockID)
351 {
352     return PollFile(sockID);    // on UNIX, socket ID's are just file ID's
353 }
354
355 //----------------------------------------------------------------------
356 // ReadFromSocket
357 //      Read a fixed size packet off the IPC port.  Abort on error.
358 //----------------------------------------------------------------------
359 void
360 ReadFromSocket(int sockID, char *buffer, int packetSize)
361 {
362     int retVal;
363     extern int errno;
364     struct sockaddr_un uName;
365     int size = sizeof(uName);
366
367     retVal = recvfrom(sockID, buffer, packetSize, 0,
368                                     (struct sockaddr *) &uName, &size);
369
370     if (retVal != packetSize) {
371         perror("in recvfrom");
372 #ifdef HOST_ALPHA
373         printf("called: %lx, got back %d, %d\n", (long) buffer, retVal, errno);
374 #else
375         printf("called: %x, got back %d, %d\n", (int) buffer, retVal, errno);
376 #endif
377     }
378     ASSERT(retVal == packetSize);
379 }
380
381 //----------------------------------------------------------------------
382 // SendToSocket
383 //      Transmit a fixed size packet to another Nachos' IPC port.
384 //
385 //----------------------------------------------------------------------
386 void
387 SendToSocket(int sockID, char *buffer, int packetSize, char *toName)
388 {
389   extern int errno;
390     struct sockaddr_un uName;
391     int retVal;
392
393     InitSocketName(&uName, toName);
394
395   /*
396    * Modified by Marcello Lioy: March 4 1996
397    *
398    * This now loops until the packet sends successfully, or fails for some
```

```
399    * other reason than a full socket.
400    */
401   while(1)
402     {
403 #if defined(HOST_LINUX) || defined(HOST_ALPHA)
404       retVal = sendto(sockID, buffer, packetSize, 0,
405                       (struct sockaddr *) &uName, sizeof(uName));
406 #else
407       retVal = sendto(sockID, buffer, packetSize, 0,
408                       (char *)&uName, sizeof(uName));
409 #endif /* HOST_LINUX */
410       if( !(retVal < 0) )
411         break;
412       else if( retVal < 0 && errno != ENOBUFS )
413         {
414           perror("socket write failed:");
415           ASSERT(0);
416         }
417       sleep(1);         // this gives the receiver a chance to read
418     }
419
420     return;
421 }
422
423
424 //----------------------------------------------------------------------
425 // CallOnUserAbort
426 //      Arrange that "func" will be called when the user aborts (e.g., by
427 //      hitting ctl-C.
428 //----------------------------------------------------------------------
429
430 void
431 CallOnUserAbort(VoidNoArgFunctionPtr func)
432 {
433 #ifdef HOST_ALPHA
434     (void)signal(SIGINT, (void (*)(int)) func);
435 #else
436     (void)signal(SIGINT, (VoidFunctionPtr) func);
437 #endif
438 }
439
440 //----------------------------------------------------------------------
441 // Sleep
442 //      Put the UNIX process running Nachos to sleep for x seconds,
443 //      to give the user time to start up another invocation of Nachos
444 //      in a different UNIX shell.
445 //----------------------------------------------------------------------
446
447 void
448 Delay(int seconds)
449 {
450     (void) sleep((unsigned) seconds);
451 }
452
453 //----------------------------------------------------------------------
454 // Abort
455 //      Quit and drop core.
456 //----------------------------------------------------------------------
```

```
457
458 void
459 Abort()
460 {
461     abort();
462 }
463
464 //----------------------------------------------------------------------
465 // Exit
466 //      Quit without dropping core.
467 //----------------------------------------------------------------------
468
469 void
470 Exit(int exitCode)
471 {
472     exit(exitCode);
473 }
474
475 //----------------------------------------------------------------------
476 // RandomInit
477 //      Initialize the pseudo-random number generator.  We use the
478 //      now obsolete "srand" and "rand" because they are more portable!
479 //----------------------------------------------------------------------
480
481 void
482 RandomInit(unsigned seed)
483 {
484     srand(seed);
485 }
486
487 //----------------------------------------------------------------------
488 // Random
489 //      Return a pseudo-random number.
490 //----------------------------------------------------------------------
491
492 int
493 Random()
494 {
495     return rand();
496 }
497
498 //----------------------------------------------------------------------
499 // AllocBoundedArray
500 //      Return an array, with the two pages just before
501 //      and after the array unmapped, to catch illegal references off
502 //      the end of the array.  Particularly useful for catching overflow
503 //      beyond fixed-size thread execution stacks.
504 //
505 //      Note: Just return the useful part!
506 //
507 //      "size" -- amount of useful space needed (in bytes)
508 //----------------------------------------------------------------------
509
510 char *
511 AllocBoundedArray(int size)
512 {
513     int pgSize = getpagesize();
514     char *ptr = new char[pgSize * 2 + size];
```

```
515
516     mprotect(ptr, pgSize, 0);
517     mprotect(ptr + pgSize + size, pgSize, 0);
518     return ptr + pgSize;
519 }
520
521 //----------------------------------------------------------------------
522 // DeallocBoundedArray
523 //      Deallocate an array of integers, unprotecting its two boundary pages.
524 //
525 //      "ptr" -- the array to be deallocated
526 //      "size" -- amount of useful space in the array (in bytes)
527 //----------------------------------------------------------------------
528
529 void
530 DeallocBoundedArray(char *ptr, int size)
531 {
532     int pgSize = getpagesize();
533
534     mprotect(ptr - pgSize, pgSize, PROT_READ | PROT_WRITE | PROT_EXEC);
535     mprotect(ptr + size, pgSize, PROT_READ | PROT_WRITE | PROT_EXEC);
536     delete [] (ptr - pgSize);
537 }
```

3.17 timer.h

```
 1 // timer.h
 2 //      Data structures to emulate a hardware timer.
 3 //
 4 //      A hardware timer generates a CPU interrupt every X milliseconds.
 5 //      This means it can be used for implementing time-slicing, or for
 6 //      having a thread go to sleep for a specific period of time.
 7 //
 8 //      We emulate a hardware timer by scheduling an interrupt to occur
 9 //      every time stats->totalTicks has increased by TimerTicks.
10 //
11 //      In order to introduce some randomness into time-slicing, if "doRandom"
12 //      is set, then the interrupt comes after a random number of ticks.
13 //
14 //  DO NOT CHANGE -- part of the machine emulation
15 //
16 // Copyright (c) 1992-1993 The Regents of the University of California.
17 // All rights reserved.  See copyright.h for copyright notice and limitation
18 // of liability and disclaimer of warranty provisions.
19
20 #ifndef TIMER_H
21 #define TIMER_H
22
23 #include "copyright.h"
24 #include "utility.h"
25
26 // The following class defines a hardware timer.
27 class Timer {
28   public:
29     Timer(VoidFunctionPtr timerHandler, _int callArg, bool doRandom);
30                                 // Initialize the timer, to call the interrupt
31                                 // handler "timerHandler" every time slice.
```

```
32     ~Timer() {}
33
34 // Internal routines to the timer emulation -- DO NOT call these
35
36     void TimerExpired();        // called internally when the hardware
37                                 // timer generates an interrupt
38
39     int TimeOfNextInterrupt();  // figure out when the timer will generate
40                                 // its next interrupt
41
42   private:
43     bool randomize;             // set if we need to use a random timeout delay
44     VoidFunctionPtr handler;    // timer interrupt handler
45     _int arg;                   // argument to pass to interrupt handler
46
47 };
48
49 #endif // TIMER_H
```

3.18 timer.cc

```
 1 // timer.cc
 2 //      Routines to emulate a hardware timer device.
 3 //
 4 //      A hardware timer generates a CPU interrupt every X milliseconds.
 5 //      This means it can be used for implementing time-slicing.
 6 //
 7 //      We emulate a hardware timer by scheduling an interrupt to occur
 8 //      every time stats->totalTicks has increased by TimerTicks.
 9 //
10 //      In order to introduce some randomness into time-slicing, if "doRandom"
11 //      is set, then the interrupt is comes after a random number of ticks.
12 //
13 //      Remember -- nothing in here is part of Nachos.  It is just
14 //      an emulation for the hardware that Nachos is running on top of.
15 //
16 //  DO NOT CHANGE -- part of the machine emulation
17 //
18 // Copyright (c) 1992-1993 The Regents of the University of California.
19 // All rights reserved.  See copyright.h for copyright notice and limitation
20 // of liability and disclaimer of warranty provisions.
21
22 #include "copyright.h"
23 #include "timer.h"
24 #include "system.h"
25
26 // dummy function because C++ does not allow pointers to member functions
27 static void TimerHandler(_int arg)
28 { Timer *p = (Timer *)arg; p->TimerExpired(); }
29
30 //----------------------------------------------------------------------
31 // Timer::Timer
32 //      Initialize a hardware timer device.  Save the place to call
33 //      on each interrupt, and then arrange for the timer to start
34 //      generating interrupts.
35 //
36 //      "timerHandler" is the interrupt handler for the timer device.
```

```
37 //              It is called with interrupts disabled every time the
38 //              the timer expires.
39 //      "callArg" is the parameter to be passed to the interrupt handler.
40 //      "doRandom" -- if true, arrange for the interrupts to occur
41 //              at random, instead of fixed, intervals.
42 //----------------------------------------------------------------------
43
44 Timer::Timer(VoidFunctionPtr timerHandler, _int callArg, bool doRandom)
45 {
46     randomize = doRandom;
47     handler = timerHandler;
48     arg = callArg;
49
50     // schedule the first interrupt from the timer device
51     interrupt->Schedule(TimerHandler, (_int) this, TimeOfNextInterrupt(),
52                 TimerInt);
53 }
54
55 //----------------------------------------------------------------------
56 // Timer::TimerExpired
57 //      Routine to simulate the interrupt generated by the hardware
58 //      timer device.  Schedule the next interrupt, and invoke the
59 //      interrupt handler.
60 //----------------------------------------------------------------------
61 void
62 Timer::TimerExpired()
63 {
64     // schedule the next timer device interrupt
65     interrupt->Schedule(TimerHandler, (_int) this, TimeOfNextInterrupt(),
66                 TimerInt);
67
68     // invoke the Nachos interrupt handler for this device
69     (*handler)(arg);
70 }
71
72 //----------------------------------------------------------------------
73 // Timer::TimeOfNextInterrupt
74 //      Return when the hardware timer device will next cause an interrupt.
75 //      If randomize is turned on, make it a (pseudo-)random delay.
76 //----------------------------------------------------------------------
77
78 int
79 Timer::TimeOfNextInterrupt()
80 {
81     if (randomize)
82         return 1 + (Random() % (TimerTicks * 2));
83     else
84         return TimerTicks;
85 }
```

3.19 translate.h

```
1 // translate.h
2 //      Data structures for managing the translation from
3 //      virtual page # -> physical page #, used for managing
4 //      physical memory on behalf of user programs.
5 //
```

```
 6 //      The data structures in this file are "dual-use" - they
 7 //      serve both as a page table entry, and as an entry in
 8 //      a software-managed translation lookaside buffer (TLB).
 9 //      Either way, each entry is of the form:
10 //      <virtual page #, physical page #>.
11 //
12 // DO NOT CHANGE -- part of the machine emulation
13 //
14 // Copyright (c) 1992-1993 The Regents of the University of California.
15 // All rights reserved.  See copyright.h for copyright notice and limitation
16 // of liability and disclaimer of warranty provisions.
17
18 #ifndef TLB_H
19 #define TLB_H
20
21 #include "copyright.h"
22 #include "utility.h"
23
24 // The following class defines an entry in a translation table -- either
25 // in a page table or a TLB.  Each entry defines a mapping from one
26 // virtual page to one physical page.
27 // In addition, there are some extra bits for access control (valid and
28 // read-only) and some bits for usage information (use and dirty).
29
30 class TranslationEntry {
31   public:
32     int virtualPage;    // The page number in virtual memory.
33     int physicalPage;   // The page number in real memory (relative to the
34                         //  start of "mainMemory"
35     bool valid;         // If this bit is set, the translation is ignored.
36                         // (In other words, the entry hasn't been initialized.)
37     bool readOnly;      // If this bit is set, the user program is not allowed
38                         // to modify the contents of the page.
39     bool use;           // This bit is set by the hardware every time the
40                         // page is referenced or modified.
41     bool dirty;         // This bit is set by the hardware every time the
42                         // page is modified.
43 };
44
45 #endif
```

3.20 translate.cc

```
 1 // translate.cc
 2 //      Routines to translate virtual addresses to physical addresses.
 3 //      Software sets up a table of legal translations.  We look up
 4 //      in the table on every memory reference to find the true physical
 5 //      memory location.
 6 //
 7 // Two types of translation are supported here.
 8 //
 9 //      Linear page table -- the virtual page # is used as an index
10 //      into the table, to find the physical page #.
11 //
12 //      Translation lookaside buffer -- associative lookup in the table
13 //      to find an entry with the same virtual page #.  If found,
14 //      this entry is used for the translation.
```

```
15 //      If not, it traps to software with an exception.
16 //
17 //      In practice, the TLB is much smaller than the amount of physical
18 //      memory (16 entries is common on a machine that has 1000's of
19 //      pages).  Thus, there must also be a backup translation scheme
20 //      (such as page tables), but the hardware doesn't need to know
21 //      anything at all about that.
22 //
23 //      Note that the contents of the TLB are specific to an address space.
24 //      If the address space changes, so does the contents of the TLB!
25 //
26 // DO NOT CHANGE -- part of the machine emulation
27 //
28 // Copyright (c) 1992-1993 The Regents of the University of California.
29 // All rights reserved.  See copyright.h for copyright notice and limitation
30 // of liability and disclaimer of warranty provisions.
31
32 #include "copyright.h"
33 #include "machine.h"
34 #include "addrspace.h"
35 #include "system.h"
36
37 // Routines for converting Words and Short Words to and from the
38 // simulated machine's format of little endian.  These end up
39 // being NOPs when the host machine is also little endian (DEC and Intel).
40
41 unsigned int
42 WordToHost(unsigned int word) {
43 #ifdef HOST_IS_BIG_ENDIAN
44          register unsigned long result;
45          result = (word >> 24) & 0x000000ff;
46          result |= (word >> 8) & 0x0000ff00;
47          result |= (word << 8) & 0x00ff0000;
48          result |= (word << 24) & 0xff000000;
49          return result;
50 #else
51          return word;
52 #endif /* HOST_IS_BIG_ENDIAN */
53 }
54
55 unsigned short
56 ShortToHost(unsigned short shortword) {
57 #ifdef HOST_IS_BIG_ENDIAN
58          register unsigned short result;
59          result = (shortword << 8) & 0xff00;
60          result |= (shortword >> 8) & 0x00ff;
61          return result;
62 #else
63          return shortword;
64 #endif /* HOST_IS_BIG_ENDIAN */
65 }
66
67 unsigned int
68 WordToMachine(unsigned int word) { return WordToHost(word); }
69
70 unsigned short
71 ShortToMachine(unsigned short shortword) { return ShortToHost(shortword); }
72
```

```
 73
 74 //----------------------------------------------------------------------
 75 // Machine::ReadMem
 76 //      Read "size" (1, 2, or 4) bytes of virtual memory at "addr" into
 77 //      the location pointed to by "value".
 78 //
 79 //      Returns FALSE if the translation step from virtual to physical memory
 80 //      failed.
 81 //
 82 //      "addr" -- the virtual address to read from
 83 //      "size" -- the number of bytes to read (1, 2, or 4)
 84 //      "value" -- the place to write the result
 85 //----------------------------------------------------------------------
 86
 87 bool
 88 Machine::ReadMem(int addr, int size, int *value)
 89 {
 90     int data;
 91     ExceptionType exception;
 92     int physicalAddress;
 93
 94     DEBUG('a', "Reading VA 0x%x, size %d\n", addr, size);
 95
 96     exception = Translate(addr, &physicalAddress, size, FALSE);
 97     if (exception != NoException) {
 98         machine->RaiseException(exception, addr);
 99         return FALSE;
100     }
101     switch (size) {
102       case 1:
103         data = machine->mainMemory[physicalAddress];
104         *value = data;
105         break;
106
107       case 2:
108         data = *(unsigned short *) &machine->mainMemory[physicalAddress];
109         *value = ShortToHost(data);
110         break;
111
112       case 4:
113         data = *(unsigned int *) &machine->mainMemory[physicalAddress];
114         *value = WordToHost(data);
115         break;
116
117       default: ASSERT(FALSE);
118     }
119
120     DEBUG('a', "\tvalue read = %8.8x\n", *value);
121     return (TRUE);
122 }
123
124 //----------------------------------------------------------------------
125 // Machine::WriteMem
126 //      Write "size" (1, 2, or 4) bytes of the contents of "value" into
127 //      virtual memory at location "addr".
128 //
129 //      Returns FALSE if the translation step from virtual to physical memory
130 //      failed.
```

```
131 //
132 //      "addr" -- the virtual address to write to
133 //       "size" -- the number of bytes to be written (1, 2, or 4)
134 //      "value" -- the data to be written
135 //----------------------------------------------------------------------
136
137 bool
138 Machine::WriteMem(int addr, int size, int value)
139 {
140     ExceptionType exception;
141     int physicalAddress;
142
143     DEBUG('a', "Writing VA 0x%x, size %d, value 0x%x\n", addr, size, value);
144
145     exception = Translate(addr, &physicalAddress, size, TRUE);
146     if (exception != NoException) {
147         machine->RaiseException(exception, addr);
148         return FALSE;
149     }
150     switch (size) {
151       case 1:
152         machine->mainMemory[physicalAddress] = (unsigned char) (value & 0xff);
153         break;
154
155       case 2:
156         *(unsigned short *) &machine->mainMemory[physicalAddress]
157                 = ShortToMachine((unsigned short) (value & 0xffff));
158         break;
159
160       case 4:
161         *(unsigned int *) &machine->mainMemory[physicalAddress]
162                 = WordToMachine((unsigned int) value);
163         break;
164
165       default: ASSERT(FALSE);
166     }
167
168     return TRUE;
169 }
170
171 //----------------------------------------------------------------------
172 // Machine::Translate
173 //      Translate a virtual address into a physical address, using
174 //      either a page table or a TLB.  Check for alignment and all sorts
175 //      of other errors, and if everything is ok, set the use/dirty bits in
176 //      the translation table entry, and store the translated physical
177 //      address in "physAddr".  If there was an error, returns the type
178 //      of the exception.
179 //
180 //      "virtAddr" -- the virtual address to translate
181 //      "physAddr" -- the place to store the physical address
182 //      "size" -- the amount of memory being read or written
183 //      "writing" -- if TRUE, check the "read-only" bit in the TLB
184 //----------------------------------------------------------------------
185
186 ExceptionType
187 Machine::Translate(int virtAddr, int* physAddr, int size, bool writing)
188 {
```

```
189     int i;
190     unsigned int vpn, offset;
191     TranslationEntry *entry;
192     unsigned int pageFrame;
193
194     DEBUG('a', "\tTranslate 0x%x, %s: ", virtAddr, writing ? "write" : "read");
195
196 // check for alignment errors
197     if (((size == 4) && (virtAddr & 0x3)) || ((size == 2) && (virtAddr & 0x1))){
198         DEBUG('a', "alignment problem at %d, size %d!\n", virtAddr, size);
199          return AddressErrorException;
200     }
201
202     // we must have either a TLB or a page table, but not both!
203     ASSERT(tlb == NULL || pageTable == NULL);
204     ASSERT(tlb != NULL || pageTable != NULL);
205
206 // calculate the virtual page number, and offset within the page,
207 // from the virtual address
208     vpn = (unsigned) virtAddr / PageSize;
209     offset = (unsigned) virtAddr % PageSize;
210
211     if (tlb == NULL) {          // => page table => vpn is index into table
212         if (vpn >= pageTableSize) {
213             DEBUG('a', "virtual page # %d too large for page table size %d!\n",
214                         virtAddr, pageTableSize);
215             return AddressErrorException;
216         } else if (!pageTable[vpn].valid) {
217             DEBUG('a', "virtual page # %d too large for page table size %d!\n",
218                         virtAddr, pageTableSize);
219             return PageFaultException;
220         }
221         entry = &pageTable[vpn];
222     } else {
223         for (entry = NULL, i = 0; i < TLBSize; i++)
224             if (tlb[i].valid && ((unsigned int)tlb[i].virtualPage == vpn)) {
225                 entry = &tlb[i];                        // FOUND!
226                 break;
227             }
228         if (entry == NULL) {                            // not found
229             DEBUG('a', "*** no valid TLB entry found for this virtual page!\n");
230             return PageFaultException;          // really, this is a TLB fault,
231                                                 // the page may be in memory,
232                                                 // but not in the TLB
233         }
234     }
235
236     if (entry->readOnly && writing) {   // trying to write to a read-only page
237         DEBUG('a', "%d mapped read-only at %d in TLB!\n", virtAddr, i);
238         return ReadOnlyException;
239     }
240     pageFrame = entry->physicalPage;
241
242     // if the pageFrame is too big, there is something really wrong!
243     // An invalid translation was loaded into the page table or TLB.
244     if (pageFrame >= NumPhysPages) {
245         DEBUG('a', "*** frame %d > %d!\n", pageFrame, NumPhysPages);
246         return BusErrorException;
```

```
247     }
248     entry->use = TRUE;          // set the use, dirty bits
249     if (writing)
250         entry->dirty = TRUE;
251     *physAddr = pageFrame * PageSize + offset;
252     ASSERT((*physAddr >= 0) && ((*physAddr + size) <= MemorySize));
253     DEBUG('a', "phys addr = 0x%x\n", *physAddr);
254     return NoException;
255 }
```

Chapter 4

Directory ../userprog/

Contents

4.1 addrspace.h 151
4.2 addrspace.cc 152
4.3 bitmap.h 155
4.4 bitmap.cc 156
4.5 exception.cc 159
4.6 progtest.cc 160
4.7 syscall.h 162

This chapter lists all the source codes found in directory `../userprog/`. They are:

```
addrspace.cc   bitmap.cc      exception.cc   syscall.h
addrspace.h    bitmap.h       progtest.cc
```

4.1 addrspace.h

```
 1 // addrspace.h
 2 //      Data structures to keep track of executing user programs
 3 //      (address spaces).
 4 //
 5 //      For now, we don't keep any information about address spaces.
 6 //      The user level CPU state is saved and restored in the thread
 7 //      executing the user program (see thread.h).
 8 //
 9 // Copyright (c) 1992-1993 The Regents of the University of California.
10 // All rights reserved.  See copyright.h for copyright notice and limitation
11 // of liability and disclaimer of warranty provisions.
12
13 #ifndef ADDRSPACE_H
14 #define ADDRSPACE_H
15
16 #include "copyright.h"
17 #include "filesys.h"
18
19 #define UserStackSize           1024    // increase this as necessary!
20
21 class AddrSpace {
22   public:
23     AddrSpace(OpenFile *executable);    // Create an address space,
```

```
24                                      // initializing it with the program
25                                      // stored in the file "executable"
26     ~AddrSpace();                    // De-allocate an address space
27
28     void InitRegisters();            // Initialize user-level CPU registers,
29                                      // before jumping to user code
30
31     void SaveState();                // Save/restore address space-specific
32     void RestoreState();             // info on a context switch
33
34   private:
35     TranslationEntry *pageTable;     // Assume linear page table translation
36                                      // for now!
37     unsigned int numPages;           // Number of pages in the virtual
38                                      // address space
39 };
40
41 #endif // ADDRSPACE_H
```

4.2 addrspace.cc

```
 1 // addrspace.cc
 2 //      Routines to manage address spaces (executing user programs).
 3 //
 4 //      In order to run a user program, you must:
 5 //
 6 //      1. link with the -N -T 0 option
 7 //      2. run coff2noff to convert the object file to Nachos format
 8 //              (Nachos object code format is essentially just a simpler
 9 //              version of the UNIX executable object code format)
10 //      3. load the NOFF file into the Nachos file system
11 //              (if you haven't implemented the file system yet, you
12 //              don't need to do this last step)
13 //
14 // Copyright (c) 1992-1993 The Regents of the University of California.
15 // All rights reserved.  See copyright.h for copyright notice and limitation
16 // of liability and disclaimer of warranty provisions.
17
18 #include "copyright.h"
19 #include "system.h"
20 #include "addrspace.h"
21 #include "noff.h"
22
23 //----------------------------------------------------------------------
24 // SwapHeader
25 //      Do little endian to big endian conversion on the bytes in the
26 //      object file header, in case the file was generated on a little
27 //      endian machine, and we're now running on a big endian machine.
28 //----------------------------------------------------------------------
29
30 static void
31 SwapHeader (NoffHeader *noffH)
32 {
33         noffH->noffMagic = WordToHost(noffH->noffMagic);
34         noffH->code.size = WordToHost(noffH->code.size);
35         noffH->code.virtualAddr = WordToHost(noffH->code.virtualAddr);
36         noffH->code.inFileAddr = WordToHost(noffH->code.inFileAddr);
```

```
37         noffH->initData.size = WordToHost(noffH->initData.size);
38         noffH->initData.virtualAddr = WordToHost(noffH->initData.virtualAddr);
39         noffH->initData.inFileAddr = WordToHost(noffH->initData.inFileAddr);
40         noffH->uninitData.size = WordToHost(noffH->uninitData.size);
41         noffH->uninitData.virtualAddr = WordToHost(noffH->uninitData.virtualAddr);
42         noffH->uninitData.inFileAddr = WordToHost(noffH->uninitData.inFileAddr);
43 }
44
45 //----------------------------------------------------------------------
46 // AddrSpace::AddrSpace
47 //      Create an address space to run a user program.
48 //      Load the program from a file "executable", and set everything
49 //      up so that we can start executing user instructions.
50 //
51 //      Assumes that the object code file is in NOFF format.
52 //
53 //      First, set up the translation from program memory to physical
54 //      memory.  For now, this is really simple (1:1), since we are
55 //      only uniprogramming, and we have a single unsegmented page table
56 //
57 //      "executable" is the file containing the object code to load into memory
58 //----------------------------------------------------------------------
59
60 AddrSpace::AddrSpace(OpenFile *executable)
61 {
62     NoffHeader noffH;
63     unsigned int i, size;
64
65     executable->ReadAt((char *)&noffH, sizeof(noffH), 0);
66     if ((noffH.noffMagic != NOFFMAGIC) &&
67                 (WordToHost(noffH.noffMagic) == NOFFMAGIC))
68         SwapHeader(&noffH);
69     ASSERT(noffH.noffMagic == NOFFMAGIC);
70
71 // how big is address space?
72     size = noffH.code.size + noffH.initData.size + noffH.uninitData.size
73                         + UserStackSize;        // we need to increase the size
74                                                 // to leave room for the stack
75     numPages = divRoundUp(size, PageSize);
76     size = numPages * PageSize;
77
78     ASSERT(numPages <= NumPhysPages);           // check we're not trying
79                                                 // to run anything too big --
80                                                 // at least until we have
81                                                 // virtual memory
82
83     DEBUG('a', "Initializing address space, num pages %d, size %d\n",
84                                         numPages, size);
85 // first, set up the translation
86     pageTable = new TranslationEntry[numPages];
87     for (i = 0; i < numPages; i++) {
88         pageTable[i].virtualPage = i;   // for now, virtual page # = phys page #
89         pageTable[i].physicalPage = i;
90         pageTable[i].valid = TRUE;
91         pageTable[i].use = FALSE;
92         pageTable[i].dirty = FALSE;
93         pageTable[i].readOnly = FALSE;  // if the code segment was entirely on
94                                         // a separate page, we could set its
```

```
95                                          // pages to be read-only
96     }
97
98 // zero out the entire address space, to zero the unitialized data segment
99 // and the stack segment
100    bzero(machine->mainMemory, size);
101
102 // then, copy in the code and data segments into memory
103    if (noffH.code.size > 0) {
104        DEBUG('a', "Initializing code segment, at 0x%x, size %d\n",
105                        noffH.code.virtualAddr, noffH.code.size);
106        executable->ReadAt(&(machine->mainMemory[noffH.code.virtualAddr]),
107                        noffH.code.size, noffH.code.inFileAddr);
108    }
109    if (noffH.initData.size > 0) {
110        DEBUG('a', "Initializing data segment, at 0x%x, size %d\n",
111                        noffH.initData.virtualAddr, noffH.initData.size);
112        executable->ReadAt(&(machine->mainMemory[noffH.initData.virtualAddr]),
113                        noffH.initData.size, noffH.initData.inFileAddr);
114    }
115
116 }
117
118 //----------------------------------------------------------------------
119 // AddrSpace::~AddrSpace
120 //      Dealloate an address space.  Nothing for now!
121 //----------------------------------------------------------------------
122
123 AddrSpace::~AddrSpace()
124 {
125    delete [] pageTable;
126 }
127
128 //----------------------------------------------------------------------
129 // AddrSpace::InitRegisters
130 //      Set the initial values for the user-level register set.
131 //
132 //      We write these directly into the "machine" registers, so
133 //       that we can immediately jump to user code.  Note that these
134 //      will be saved/restored into the currentThread->userRegisters
135 //      when this thread is context switched out.
136 //----------------------------------------------------------------------
137
138 void
139 AddrSpace::InitRegisters()
140 {
141    int i;
142
143    for (i = 0; i < NumTotalRegs; i++)
144        machine->WriteRegister(i, 0);
145
146    // Initial program counter -- must be location of "Start"
147    machine->WriteRegister(PCReg, 0);
148
149    // Need to also tell MIPS where next instruction is, because
150    // of branch delay possibility
151    machine->WriteRegister(NextPCReg, 4);
152
```

```
153    // Set the stack register to the end of the address space, where we
154    // allocated the stack; but subtract off a bit, to make sure we don't
155    // accidentally reference off the end!
156     machine->WriteRegister(StackReg, numPages * PageSize - 16);
157     DEBUG('a', "Initializing stack register to %d\n", numPages * PageSize - 16);
158 }
159
160 //----------------------------------------------------------------------
161 // AddrSpace::SaveState
162 //      On a context switch, save any machine state, specific
163 //      to this address space, that needs saving.
164 //
165 //      For now, nothing!
166 //----------------------------------------------------------------------
167
168 void AddrSpace::SaveState()
169 {}
170
171 //----------------------------------------------------------------------
172 // AddrSpace::RestoreState
173 //      On a context switch, restore the machine state so that
174 //      this address space can run.
175 //
176 //      For now, tell the machine where to find the page table.
177 //----------------------------------------------------------------------
178
179 void AddrSpace::RestoreState()
180 {
181     machine->pageTable = pageTable;
182     machine->pageTableSize = numPages;
183 }
```

4.3 bitmap.h

```
 1 // bitmap.h
 2 //      Data structures defining a bitmap -- an array of bits each of which
 3 //      can be either on or off.
 4 //
 5 //      Represented as an array of unsigned integers, on which we do
 6 //      modulo arithmetic to find the bit we are interested in.
 7 //
 8 //      The bitmap can be parameterized with with the number of bits being
 9 //      managed.
10 //
11 // Copyright (c) 1992-1993 The Regents of the University of California.
12 // All rights reserved.  See copyright.h for copyright notice and limitation
13 // of liability and disclaimer of warranty provisions.
14
15 #ifndef BITMAP_H
16 #define BITMAP_H
17
18 #include "copyright.h"
19 #include "utility.h"
20 #include "openfile.h"
21
22 // Definitions helpful for representing a bitmap as an array of integers
23 #define BitsInByte      8
```

```
24 #define BitsInWord      32
25
26 // The following class defines a "bitmap" -- an array of bits,
27 // each of which can be independently set, cleared, and tested.
28 //
29 // Most useful for managing the allocation of the elements of an array --
30 // for instance, disk sectors, or main memory pages.
31 // Each bit represents whether the corresponding sector or page is
32 // in use or free.
33
34 class BitMap {
35   public:
36     BitMap(int nitems);         // Initialize a bitmap, with "nitems" bits
37                                 // initially, all bits are cleared.
38     ~BitMap();                  // De-allocate bitmap
39
40     void Mark(int which);       // Set the "nth" bit
41     void Clear(int which);      // Clear the "nth" bit
42     bool Test(int which);       // Is the "nth" bit set?
43     int Find();                 // Return the # of a clear bit, and as a side
44                                 // effect, set the bit.
45                                 // If no bits are clear, return -1.
46     int NumClear();             // Return the number of clear bits
47
48     void Print();               // Print contents of bitmap
49
50     // These aren't needed until FILESYS, when we will need to read and
51     // write the bitmap to a file
52     void FetchFrom(OpenFile *file);     // fetch contents from disk
53     void WriteBack(OpenFile *file);     // write contents to disk
54
55   private:
56     int numBits;                        // number of bits in the bitmap
57     int numWords;                       // number of words of bitmap storage
58                                         // (rounded up if numBits is not a
59                                         //  multiple of the number of bits in
60                                         //  a word)
61     unsigned int *map;                  // bit storage
62 };
63
64 #endif // BITMAP_H
```

4.4 bitmap.cc

```
 1 // bitmap.c
 2 //      Routines to manage a bitmap -- an array of bits each of which
 3 //      can be either on or off.  Represented as an array of integers.
 4 //
 5 // Copyright (c) 1992-1993 The Regents of the University of California.
 6 // All rights reserved.  See copyright.h for copyright notice and limitation
 7 // of liability and disclaimer of warranty provisions.
 8
 9 #include "copyright.h"
10 #include "bitmap.h"
11
12 //----------------------------------------------------------------------
13 // BitMap::BitMap
```

```
14 //      Initialize a bitmap with "nitems" bits, so that every bit is clear.
15 //      it can be added somewhere on a list.
16 //
17 //      "nitems" is the number of bits in the bitmap.
18 //----------------------------------------------------------------------
19
20 BitMap::BitMap(int nitems)
21 {
22     numBits = nitems;
23     numWords = divRoundUp(numBits, BitsInWord);
24     map = new unsigned int[numWords];
25     for (int i = 0; i < numBits; i++)
26         Clear(i);
27 }
28
29 //----------------------------------------------------------------------
30 // BitMap::~BitMap
31 //      De-allocate a bitmap.
32 //----------------------------------------------------------------------
33
34 BitMap::~BitMap()
35 {
36     delete map;
37 }
38
39 //----------------------------------------------------------------------
40 // BitMap::Set
41 //      Set the "nth" bit in a bitmap.
42 //
43 //      "which" is the number of the bit to be set.
44 //----------------------------------------------------------------------
45
46 void
47 BitMap::Mark(int which)
48 {
49     ASSERT(which >= 0 && which < numBits);
50     map[which / BitsInWord] |= 1 << (which % BitsInWord);
51 }
52
53 //----------------------------------------------------------------------
54 // BitMap::Clear
55 //      Clear the "nth" bit in a bitmap.
56 //
57 //      "which" is the number of the bit to be cleared.
58 //----------------------------------------------------------------------
59
60 void
61 BitMap::Clear(int which)
62 {
63     ASSERT(which >= 0 && which < numBits);
64     map[which / BitsInWord] &= ~(1 << (which % BitsInWord));
65 }
66
67 //----------------------------------------------------------------------
68 // BitMap::Test
69 //      Return TRUE if the "nth" bit is set.
70 //
71 //      "which" is the number of the bit to be tested.
```

```
 72 //----------------------------------------------------------------------
 73
 74 bool
 75 BitMap::Test(int which)
 76 {
 77     ASSERT(which >= 0 && which < numBits);
 78
 79     if (map[which / BitsInWord] & (1 << (which % BitsInWord)))
 80         return TRUE;
 81     else
 82         return FALSE;
 83 }
 84
 85 //----------------------------------------------------------------------
 86 // BitMap::Find
 87 //      Return the number of the first bit which is clear.
 88 //      As a side effect, set the bit (mark it as in use).
 89 //      (In other words, find and allocate a bit.)
 90 //
 91 //      If no bits are clear, return -1.
 92 //----------------------------------------------------------------------
 93
 94 int
 95 BitMap::Find()
 96 {
 97     for (int i = 0; i < numBits; i++)
 98         if (!Test(i)) {
 99             Mark(i);
100             return i;
101         }
102     return -1;
103 }
104
105 //----------------------------------------------------------------------
106 // BitMap::NumClear
107 //      Return the number of clear bits in the bitmap.
108 //      (In other words, how many bits are unallocated?)
109 //----------------------------------------------------------------------
110
111 int
112 BitMap::NumClear()
113 {
114     int count = 0;
115
116     for (int i = 0; i < numBits; i++)
117         if (!Test(i)) count++;
118     return count;
119 }
120
121 //----------------------------------------------------------------------
122 // BitMap::Print
123 //      Print the contents of the bitmap, for debugging.
124 //
125 //      Could be done in a number of ways, but we just print the #'s of
126 //      all the bits that are set in the bitmap.
127 //----------------------------------------------------------------------
128
129 void
```

```
130 BitMap::Print()
131 {
132     printf("Bitmap set:\n");
133     for (int i = 0; i < numBits; i++)
134         if (Test(i))
135             printf("%d, ", i);
136     printf("\n");
137 }
138
139 // These aren't needed until the FILESYS assignment
140
141 //----------------------------------------------------------------------
142 // BitMap::FetchFromFile
143 //      Initialize the contents of a bitmap from a Nachos file.
144 //
145 //      "file" is the place to read the bitmap from
146 //----------------------------------------------------------------------
147
148 void
149 BitMap::FetchFrom(OpenFile *file)
150 {
151     file->ReadAt((char *)map, numWords * sizeof(unsigned), 0);
152 }
153
154 //----------------------------------------------------------------------
155 // BitMap::WriteBack
156 //      Store the contents of a bitmap to a Nachos file.
157 //
158 //      "file" is the place to write the bitmap to
159 //----------------------------------------------------------------------
160
161 void
162 BitMap::WriteBack(OpenFile *file)
163 {
164    file->WriteAt((char *)map, numWords * sizeof(unsigned), 0);
165 }
```

4.5 exception.cc

```
 1 // exception.cc
 2 //      Entry point into the Nachos kernel from user programs.
 3 //      There are two kinds of things that can cause control to
 4 //      transfer back to here from user code:
 5 //
 6 //      syscall -- The user code explicitly requests to call a procedure
 7 //      in the Nachos kernel.  Right now, the only function we support is
 8 //      "Halt".
 9 //
10 //      exceptions -- The user code does something that the CPU can't handle.
11 //      For instance, accessing memory that doesn't exist, arithmetic errors,
12 //      etc.
13 //
14 //      Interrupts (which can also cause control to transfer from user
15 //      code into the Nachos kernel) are handled elsewhere.
16 //
17 // For now, this only handles the Halt() system call.
18 // Everything else core dumps.
```

```
19 //
20 // Copyright (c) 1992-1993 The Regents of the University of California.
21 // All rights reserved.  See copyright.h for copyright notice and limitation
22 // of liability and disclaimer of warranty provisions.
23
24 #include "copyright.h"
25 #include "system.h"
26 #include "syscall.h"
27
28 //----------------------------------------------------------------------
29 // ExceptionHandler
30 //      Entry point into the Nachos kernel.  Called when a user program
31 //      is executing, and either does a syscall, or generates an addressing
32 //      or arithmetic exception.
33 //
34 //      For system calls, the following is the calling convention:
35 //
36 //      system call code -- r2
37 //              arg1 -- r4
38 //              arg2 -- r5
39 //              arg3 -- r6
40 //              arg4 -- r7
41 //
42 //      The result of the system call, if any, must be put back into r2.
43 //
44 // And don't forget to increment the pc before returning. (Or else you'll
45 // loop making the same system call forever!
46 //
47 //      "which" is the kind of exception.  The list of possible exceptions
48 //      are in machine.h.
49 //----------------------------------------------------------------------
50
51 void
52 ExceptionHandler(ExceptionType which)
53 {
54     int type = machine->ReadRegister(2);
55
56     if ((which == SyscallException) && (type == SC_Halt)) {
57         DEBUG('a', "Shutdown, initiated by user program.\n");
58         interrupt->Halt();
59     } else {
60         printf("Unexpected user mode exception %d %d\n", which, type);
61         ASSERT(FALSE);
62     }
63 }
```

4.6 progtest.cc

```
 1 // progtest.cc
 2 //      Test routines for demonstrating that Nachos can load
 3 //      a user program and execute it.
 4 //
 5 //      Also, routines for testing the Console hardware device.
 6 //
 7 // Copyright (c) 1992-1993 The Regents of the University of California.
 8 // All rights reserved.  See copyright.h for copyright notice and limitation
 9 // of liability and disclaimer of warranty provisions.
```

```
10
11 #include "copyright.h"
12 #include "system.h"
13 #include "console.h"
14 #include "addrspace.h"
15 #include "synch.h"
16
17 //----------------------------------------------------------------------
18 // StartProcess
19 //      Run a user program.  Open the executable, load it into
20 //      memory, and jump to it.
21 //----------------------------------------------------------------------
22
23 void
24 StartProcess(char *filename)
25 {
26     OpenFile *executable = fileSystem->Open(filename);
27     AddrSpace *space;
28
29     if (executable == NULL) {
30         printf("Unable to open file %s\n", filename);
31         return;
32     }
33     space = new AddrSpace(executable);
34     currentThread->space = space;
35
36     delete executable;                  // close file
37
38     space->InitRegisters();             // set the initial register values
39     space->RestoreState();              // load page table register
40
41     machine->Run();                     // jump to the user progam
42     ASSERT(FALSE);                      // machine->Run never returns;
43                                         // the address space exits
44                                         // by doing the syscall "exit"
45 }
46
47 // Data structures needed for the console test.  Threads making
48 // I/O requests wait on a Semaphore to delay until the I/O completes.
49
50 static Console *console;
51 static Semaphore *readAvail;
52 static Semaphore *writeDone;
53
54 //----------------------------------------------------------------------
55 // ConsoleInterruptHandlers
56 //      Wake up the thread that requested the I/O.
57 //----------------------------------------------------------------------
58
59 static void ReadAvail(_int arg) { readAvail->V(); }
60 static void WriteDone(_int arg) { writeDone->V(); }
61
62 //----------------------------------------------------------------------
63 // ConsoleTest
64 //      Test the console by echoing characters typed at the input onto
65 //      the output.  Stop when the user types a 'q'.
66 //----------------------------------------------------------------------
67
```

```
68 void
69 ConsoleTest (char *in, char *out)
70 {
71     char ch;
72
73     console = new Console(in, out, ReadAvail, WriteDone, 0);
74     readAvail = new Semaphore("read avail", 0);
75     writeDone = new Semaphore("write done", 0);
76
77     for (;;) {
78         readAvail->P();         // wait for character to arrive
79         ch = console->GetChar();
80         console->PutChar(ch);   // echo it!
81         writeDone->P() ;        // wait for write to finish
82         if (ch == 'q') return;  // if q, quit
83     }
84 }
```

4.7 syscall.h

```
 1 /* syscalls.h
 2  *      Nachos system call interface.  These are Nachos kernel operations
 3  *      that can be invoked from user programs, by trapping to the kernel
 4  *      via the "syscall" instruction.
 5  *
 6  *      This file is included by user programs and by the Nachos kernel.
 7  *
 8  * Copyright (c) 1992-1993 The Regents of the University of California.
 9  * All rights reserved.  See copyright.h for copyright notice and limitation
10  * of liability and disclaimer of warranty provisions.
11  */
12
13 #ifndef SYSCALLS_H
14 #define SYSCALLS_H
15
16 #include "copyright.h"
17
18 /* system call codes -- used by the stubs to tell the kernel which system call
19  * is being asked for
20  */
21 #define SC_Halt         0
22 #define SC_Exit         1
23 #define SC_Exec         2
24 #define SC_Join         3
25 #define SC_Create       4
26 #define SC_Open         5
27 #define SC_Read         6
28 #define SC_Write        7
29 #define SC_Close        8
30 #define SC_Fork         9
31 #define SC_Yield        10
32
33 #ifndef IN_ASM
34
35 /* The system call interface.  These are the operations the Nachos
36  * kernel needs to support, to be able to run user programs.
37  *
```

```
38  * Each of these is invoked by a user program by simply calling the
39  * procedure; an assembly language stub stuffs the system call code
40  * into a register, and traps to the kernel.  The kernel procedures
41  * are then invoked in the Nachos kernel, after appropriate error checking,
42  * from the system call entry point in exception.cc.
43  */
44
45 /* Stop Nachos, and print out performance stats */
46 void Halt();
47
48
49 /* Address space control operations: Exit, Exec, and Join */
50
51 /* This user program is done (status = 0 means exited normally). */
52 void Exit(int status);
53
54 /* A unique identifier for an executing user program (address space) */
55 typedef int SpaceId;
56
57 /* Run the executable, stored in the Nachos file "name", and return the
58  * address space identifier
59  */
60 SpaceId Exec(char *name);
61
62 /* Only return once the the user program "id" has finished.
63  * Return the exit status.
64  */
65 int Join(SpaceId id);
66
67
68 /* File system operations: Create, Open, Read, Write, Close
69  * These functions are patterned after UNIX -- files represent
70  * both files *and* hardware I/O devices.
71  *
72  * If this assignment is done before doing the file system assignment,
73  * note that the Nachos file system has a stub implementation, which
74  * will work for the purposes of testing out these routines.
75  */
76
77 /* A unique identifier for an open Nachos file. */
78 typedef int OpenFileId;
79
80 /* when an address space starts up, it has two open files, representing
81  * keyboard input and display output (in UNIX terms, stdin and stdout).
82  * Read and Write can be used directly on these, without first opening
83  * the console device.
84  */
85
86 #define ConsoleInput    0
87 #define ConsoleOutput   1
88
89 /* Create a Nachos file, with "name" */
90 void Create(char *name);
91
92 /* Open the Nachos file "name", and return an "OpenFileId" that can
93  * be used to read and write to the file.
94  */
95 OpenFileId Open(char *name);
```

```
 96
 97 /* Write "size" bytes from "buffer" to the open file. */
 98 void Write(char *buffer, int size, OpenFileId id);
 99
100 /* Read "size" bytes from the open file into "buffer".
101  * Return the number of bytes actually read -- if the open file isn't
102  * long enough, or if it is an I/O device, and there aren't enough
103  * characters to read, return whatever is available (for I/O devices,
104  * you should always wait until you can return at least one character).
105  */
106 int Read(char *buffer, int size, OpenFileId id);
107
108 /* Close the file, we're done reading and writing to it. */
109 void Close(OpenFileId id);
110
111
112
113 /* User-level thread operations: Fork and Yield.  To allow multiple
114  * threads to run within a user program.
115  */
116
117 /* Fork a thread to run a procedure ("func") in the *same* address space
118  * as the current thread.
119  */
120 void Fork(void (*func)());
121
122 /* Yield the CPU to another runnable thread, whether in this address space
123  * or not.
124  */
125 void Yield();
126
127 #endif /* IN_ASM */
128
129 #endif /* SYSCALL_H */
```

Chapter 5

Directory ../bin/

Contents

5.1 coff.h . . . 165
5.2 coff2flat.c . . . 166
5.3 coff2noff.c . . . 169
5.4 d.c . . . 173
5.5 disasm.c . . . 177
5.6 encode.h . . . 180
5.7 execute.c . . . 182
5.8 instr.h . . . 192
5.9 int.h . . . 192
5.10 main.c . . . 193
5.11 noff.h . . . 196
5.12 opstrings.c . . . 196
5.13 out.c . . . 199
5.14 system.c . . . 203

This chapter lists all the source codes found in directory `../bin/`. They are:

```
coff.h        disasm.c      int.h         out.c
coff2flat.c   encode.h      main.c        system.c
coff2noff.c   execute.c     noff.h
d.c           instr.h       opstrings.c
```

5.1 coff.h

```
 1 /* coff.h
 2  *   Data structures that describe the MIPS COFF format.
 3  */
 4
 5 #ifdef HOST_ALPHA              /* Needed because of gcc uses 64 bit long  */
 6 #define _long int              /* integers on the DEC ALPHA architecture. */
 7 #else
 8 #define _long long
 9 #endif
10
11 struct filehdr {
12         unsigned short  f_magic;        /* magic number */
13         unsigned short  f_nscns;        /* number of sections */
```

```
14         _long           f_timdat;       /* time & date stamp */
15         _long           f_symptr;       /* file pointer to symbolic header */
16         _long           f_nsyms;        /* sizeof(symbolic hdr) */
17         unsigned short  f_opthdr;       /* sizeof(optional hdr) */
18         unsigned short  f_flags;        /* flags */
19       };
20
21 #define  MIPSELMAGIC    0x0162
22
23 #define OMAGIC  0407
24 #define SOMAGIC 0x0701
25
26 typedef struct aouthdr {
27         short   magic;          /* see above                            */
28         short   vstamp;         /* version stamp                        */
29         _long    tsize;         /* text size in bytes, padded to DW bdry*/
30         _long    dsize;         /* initialized data "  "                */
31         _long    bsize;         /* uninitialized data "   "             */
32         _long    entry;         /* entry pt.                            */
33         _long    text_start;    /* base of text used for this file      */
34         _long    data_start;    /* base of data used for this file      */
35         _long    bss_start;     /* base of bss used for this file       */
36         _long    gprmask;       /* general purpose register mask        */
37         _long    cprmask[4];    /* co-processor register masks          */
38         _long    gp_value;      /* the gp value used for this object    */
39       } AOUTHDR;
40 #define AOUTHSZ sizeof(AOUTHDR)
41
42
43 struct scnhdr {
44         char            s_name[8];      /* section name */
45         _long           s_paddr;        /* physical address, aliased s_nlib */
46         _long           s_vaddr;        /* virtual address */
47         _long           s_size;         /* section size */
48         _long           s_scnptr;       /* file ptr to raw data for section */
49         _long           s_relptr;       /* file ptr to relocation */
50         _long           s_lnnoptr;      /* file ptr to gp histogram */
51         unsigned short  s_nreloc;       /* number of relocation entries */
52         unsigned short  s_nlnno;        /* number of gp histogram entries */
53         _long           s_flags;        /* flags */
54       };
55
```

5.2 coff2flat.c

```
 1 /*
 2  Copyright (c) 1992 The Regents of the University of California.
 3  All rights reserved.  See copyright.h for copyright notice and limitation
 4  of liability and disclaimer of warranty provisions.
 5  */
 6
 7 /* This program reads in a COFF format file, and outputs a flat file --
 8  * the flat file can then be copied directly to virtual memory and executed.
 9  * In other words, the various pieces of the object code are loaded at
10  * the appropriate offset in the flat file.
11  *
12  * Assumes coff file compiled with -N -T 0 to make sure it's not shared text.
```

```
 */

#define MAIN
#include "copyright.h"
#undef MAIN
/*
#include <filehdr.h>
#include <aouthdr.h>
#include <scnhdr.h>
#include <reloc.h>
#include <syms.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <limits.h>
#include <stdio.h>
*/
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <limits.h>
#include <stdio.h>

#include "coff.h"
#include "noff.h"

/* NOTE -- once you have implemented large files, it's ok to make this bigger! */
#define StackSize               1024      /* in bytes */
#define ReadStruct(f,s)         Read(f,(char *)&s,sizeof(s))

extern char *malloc();

unsigned int
WordToHost(unsigned int word) {
#ifdef HOST_IS_BIG_ENDIAN
         register unsigned long result;
         result = (word >> 24) & 0x000000ff;
         result |= (word >> 8) & 0x0000ff00;
         result |= (word << 8) & 0x00ff0000;
         result |= (word << 24) & 0xff000000;
         return result;
#else
         return word;
#endif /* HOST_IS_BIG_ENDIAN */
}

unsigned short
ShortToHost(unsigned short shortword) {
#if HOST_IS_BIG_ENDIAN
         register unsigned short result;
         result = (shortword << 8) & 0xff00;
         result |= (shortword >> 8) & 0x00ff;
         return result;
#else
          return shortword;
#endif /* HOST_IS_BIG_ENDIAN */
}

```

```
 71
 72 /* read and check for error */
 73 void Read(int fd, char *buf, int nBytes)
 74 {
 75     if (read(fd, buf, nBytes) != nBytes) {
 76         fprintf(stderr, "File is too short\n");
 77         exit(1);
 78     }
 79 }
 80
 81 /* write and check for error */
 82 void Write(int fd, char *buf, int nBytes)
 83 {
 84     if (write(fd, buf, nBytes) != nBytes) {
 85         fprintf(stderr, "Unable to write file\n");
 86         exit(1);
 87     }
 88 }
 89
 90 /* do the real work */
 91 main (int argc, char **argv)
 92 {
 93     int fdIn, fdOut, numsections, i, top, tmp;
 94     struct filehdr fileh;
 95     struct aouthdr systemh;
 96     struct scnhdr *sections;
 97     char *buffer;
 98
 99     if (argc < 2) {
100         fprintf(stderr, "Usage: %s <coffFileName> <flatFileName>\n", argv[0]);
101         exit(1);
102     }
103
104 /* open the COFF file (input) */
105     fdIn = open(argv[1], O_RDONLY, 0);
106     if (fdIn == -1) {
107         perror(argv[1]);
108         exit(1);
109     }
110
111 /* open the NOFF file (output) */
112     fdOut = open(argv[2], O_WRONLY|O_CREAT|O_TRUNC , 0666);
113     if (fdIn == -1) {
114         perror(argv[2]);
115         exit(1);
116     }
117
118 /* Read in the file header and check the magic number. */
119     ReadStruct(fdIn,fileh);
120     fileh.f_magic = ShortToHost(fileh.f_magic);
121     fileh.f_nscns = ShortToHost(fileh.f_nscns);
122     if (fileh.f_magic != MIPSELMAGIC) {
123         fprintf(stderr, "File is not a MIPSEL COFF file\n");
124         exit(1);
125     }
126
127 /* Read in the system header and check the magic number */
128     ReadStruct(fdIn,systemh);
```

```
129     systemh.magic = ShortToHost(systemh.magic);
130     if (systemh.magic != OMAGIC) {
131         fprintf(stderr, "File is not a OMAGIC file\n");
132         exit(1);
133     }
134
135 /* Read in the section headers. */
136     numsections = fileh.f_nscns;
137     sections = (struct scnhdr *)malloc(fileh.f_nscns * sizeof(struct scnhdr));
138     Read(fdIn, (char *) sections, fileh.f_nscns * sizeof(struct scnhdr));
139
140  /* Copy the segments in */
141     printf("Loading %d sections:\n", fileh.f_nscns);
142     for (top = 0, i = 0; i < fileh.f_nscns; i++) {
143         printf("\t\"%s\", filepos 0x%x, mempos 0x%x, size 0x%x\n",
144               sections[i].s_name, sections[i].s_scnptr,
145               sections[i].s_paddr, sections[i].s_size);
146         if ((sections[i].s_paddr + sections[i].s_size) > top)
147             top = sections[i].s_paddr + sections[i].s_size;
148         if (strcmp(sections[i].s_name, ".bss") && /* no need to copy if .bss */
149                 strcmp(sections[i].s_name, ".sbss")) {
150             lseek(fdIn, sections[i].s_scnptr, 0);
151             buffer = malloc(sections[i].s_size);
152             Read(fdIn, buffer, sections[i].s_size);
153             Write(fdOut, buffer, sections[i].s_size);
154             free(buffer);
155         }
156     }
157 /* put a blank word at the end, so we know where the end is! */
158     printf("Adding stack of size: %d\n", StackSize);
159     lseek(fdOut, top + StackSize - 4, 0);
160     tmp = 0;
161     Write(fdOut, (char *)&tmp, 4);
162
163     close(fdIn);
164     close(fdOut);
165 }
```

5.3 coff2noff.c

```
 1 /* coff2noff.c
 2  *
 3  * This program reads in a COFF format file, and outputs a NOFF format file.
 4  * The NOFF format is essentially just a simpler version of the COFF file,
 5  * recording where each segment is in the NOFF file, and where it is to
 6  * go in the virtual address space.
 7  *
 8  * Assumes coff file is linked with either
 9  *      gld with -N -Ttext 0
10  *      ld with  -N -T 0
11  * to make sure the object file has no shared text.
12  *
13  * Also assumes that the COFF file has at most 3 segments:
14  *      .text   -- read-only executable instructions
15  *      .data   -- initialized data
16  *      .bss/.sbss -- uninitialized data (should be zero'd on program startup)
17  *
```

```
18  * Copyright (c) 1992-1993 The Regents of the University of California.
19  * All rights reserved.  See copyright.h for copyright notice and limitation
20  * of liability and disclaimer of warranty provisions.
21  */
22
23 #define MAIN
24 #include "copyright.h"
25 #undef MAIN
26
27 #include <sys/types.h>
28 #include <sys/stat.h>
29 #include <fcntl.h>
30 #include <limits.h>
31 #include <stdio.h>
32
33 #include "coff.h"
34 #include "noff.h"
35
36 /* Routines for converting words and short words to and from the
37  * simulated machine's format of little endian.  These end up
38  * being NOPs when the host machine is little endian.
39  */
40
41 unsigned int
42 WordToHost(unsigned int word) {
43 #ifdef HOST_IS_BIG_ENDIAN
44          register unsigned long result;
45          result = (word >> 24) & 0x000000ff;
46          result |= (word >> 8) & 0x0000ff00;
47          result |= (word << 8) & 0x00ff0000;
48          result |= (word << 24) & 0xff000000;
49          return result;
50 #else
51          return word;
52 #endif /* HOST_IS_BIG_ENDIAN */
53 }
54
55 unsigned short
56 ShortToHost(unsigned short shortword) {
57 #if HOST_IS_BIG_ENDIAN
58          register unsigned short result;
59          result = (shortword << 8) & 0xff00;
60          result |= (shortword >> 8) & 0x00ff;
61          return result;
62 #else
63          return shortword;
64 #endif /* HOST_IS_BIG_ENDIAN */
65 }
66
67 #define ReadStruct(f,s)  Read(f,(char *)&s,sizeof(s))
68
69 extern char *malloc();
70 char *noffFileName = NULL;
71
72 /* read and check for error */
73 void Read(int fd, char *buf, int nBytes)
74 {
75     if (read(fd, buf, nBytes) != nBytes) {
```

```
 76         fprintf(stderr, "File is too short\n");
 77         unlink(noffFileName);
 78         exit(1);
 79     }
 80 }
 81
 82 /* write and check for error */
 83 void Write(int fd, char *buf, int nBytes)
 84 {
 85     if (write(fd, buf, nBytes) != nBytes) {
 86         fprintf(stderr, "Unable to write file\n");
 87         unlink(noffFileName);
 88         exit(1);
 89     }
 90 }
 91
 92 main (int argc, char **argv)
 93 {
 94     int fdIn, fdOut, numsections, i, inNoffFile;
 95     struct filehdr fileh;
 96     struct aouthdr systemh;
 97     struct scnhdr *sections;
 98     char *buffer;
 99     NoffHeader noffH;
100
101     if (argc < 2) {
102         fprintf(stderr, "Usage: %s <coffFileName> <noffFileName>\n", argv[0]);
103         exit(1);
104     }
105
106 /* open the COFF file (input) */
107     fdIn = open(argv[1], O_RDONLY, 0);
108     if (fdIn == -1) {
109         perror(argv[1]);
110         exit(1);
111     }
112
113 /* open the NOFF file (output) */
114     fdOut = open(argv[2], O_WRONLY|O_CREAT|O_TRUNC , 0666);
115     if (fdIn == -1) {
116         perror(argv[2]);
117         exit(1);
118     }
119     noffFileName = argv[2];
120
121 /* Read in the file header and check the magic number. */
122     ReadStruct(fdIn,fileh);
123     fileh.f_magic = ShortToHost(fileh.f_magic);
124     fileh.f_nscns = ShortToHost(fileh.f_nscns);
125     if (fileh.f_magic != MIPSELMAGIC) {
126         fprintf(stderr, "File is not a MIPSEL COFF file\n");
127         unlink(noffFileName);
128         exit(1);
129     }
130
131 /* Read in the system header and check the magic number */
132     ReadStruct(fdIn,systemh);
133     systemh.magic = ShortToHost(systemh.magic);
```

```
134     if (systemh.magic != OMAGIC) {
135         fprintf(stderr, "File is not a OMAGIC file\n");
136         unlink(noffFileName);
137         exit(1);
138     }
139
140 /* Read in the section headers. */
141     numsections = fileh.f_nscns;
142     printf("numsections %d \n",numsections);
143     sections = (struct scnhdr *)malloc(numsections * sizeof(struct scnhdr));
144     Read(fdIn, (char *) sections, numsections * sizeof(struct scnhdr));
145
146    for (i = 0; i < numsections; i++) {
147      sections[i].s_paddr =  WordToHost(sections[i].s_paddr);
148      sections[i].s_size = WordToHost(sections[i].s_size);
149      sections[i].s_scnptr = WordToHost(sections[i].s_scnptr);
150    }
151
152  /* initialize the NOFF header, in case not all the segments are defined
153   * in the COFF file
154   */
155     noffH.noffMagic = NOFFMAGIC;
156     noffH.code.size = 0;
157     noffH.initData.size = 0;
158     noffH.uninitData.size = 0;
159
160  /* Copy the segments in */
161     inNoffFile = sizeof(NoffHeader);
162     lseek(fdOut, inNoffFile, 0);
163     printf("Loading %d sections:\n", numsections);
164     for (i = 0; i < numsections; i++) {
165         printf("\t\"%s\", filepos 0x%x, mempos 0x%x, size 0x%x\n",
166               sections[i].s_name, sections[i].s_scnptr,
167               sections[i].s_paddr, sections[i].s_size);
168         if (sections[i].s_size == 0) {
169                 /* do nothing! */
170         } else if (!strcmp(sections[i].s_name, ".text")) {
171             noffH.code.virtualAddr = sections[i].s_paddr;
172             noffH.code.inFileAddr = inNoffFile;
173             noffH.code.size = sections[i].s_size;
174             lseek(fdIn, sections[i].s_scnptr, 0);
175             buffer = malloc(sections[i].s_size);
176             Read(fdIn, buffer, sections[i].s_size);
177             Write(fdOut, buffer, sections[i].s_size);
178             free(buffer);
179             inNoffFile += sections[i].s_size;
180         } else if (!strcmp(sections[i].s_name, ".data")
181                         || !strcmp(sections[i].s_name, ".rdata")) {
182             /* need to check if we have both .data and .rdata
183              *  -- make sure one or the other is empty! */
184             if (noffH.initData.size != 0) {
185                 fprintf(stderr, "Can't handle both data and rdata\n");
186                 unlink(noffFileName);
187                 exit(1);
188             }
189             noffH.initData.virtualAddr = sections[i].s_paddr;
190             noffH.initData.inFileAddr = inNoffFile;
191             noffH.initData.size = sections[i].s_size;
```

```
192             lseek(fdIn, sections[i].s_scnptr, 0);
193             buffer = malloc(sections[i].s_size);
194             Read(fdIn, buffer, sections[i].s_size);
195             Write(fdOut, buffer, sections[i].s_size);
196             free(buffer);
197             inNoffFile += sections[i].s_size;
198         } else if (!strcmp(sections[i].s_name, ".bss") ||
199                         !strcmp(sections[i].s_name, ".sbss")) {
200             /* need to check if we have both .bss and .sbss -- make sure they
201              * are contiguous
202              */
203             if (noffH.uninitData.size != 0) {
204                 if (sections[i].s_paddr == (noffH.uninitData.virtualAddr +
205                                             noffH.uninitData.size)) {
206                     fprintf(stderr, "Can't handle both bss and sbss\n");
207                     unlink(noffFileName);
208                     exit(1);
209                 }
210                 noffH.uninitData.size += sections[i].s_size;
211             } else {
212                 noffH.uninitData.virtualAddr = sections[i].s_paddr;
213                 noffH.uninitData.size = sections[i].s_size;
214             }
215             /* we don't need to copy the uninitialized data! */
216         } else {
217             fprintf(stderr, "Unknown segment type: %s\n", sections[i].s_name);
218             unlink(noffFileName);
219             exit(1);
220         }
221     }
222     lseek(fdOut, 0, 0);
223     Write(fdOut, (char *)&noffH, sizeof(NoffHeader));
224     close(fdIn);
225     close(fdOut);
226     exit(0);
227 }
```

5.4 d.c

```
 1 /*
 2  Copyright (c) 1992-1993 The Regents of the University of California.
 3  All rights reserved.  See copyright.h for copyright notice and limitation
 4  of liability and disclaimer of warranty provisions.
 5  */
 6
 7 #include "copyright.h"
 8 #include "instr.h"
 9 #include "encode.h"
10
11 #ifndef NULL
12 #define NULL    0
13 #endif
14
15 int sptr;
16 int longdis = 1;
17
18 extern char *normalops[], *specialops[];
```

```
19
20
21 char *regstrings[] =
22 {
23 "0", "r1", "r2", "r3", "r4", "r5", "r6", "r7", "r8", "r9",
24 "r10", "r11", "r12", "r13", "r14", "r15", "r16", "r17", "r18", "r19",
25 "r20", "r21", "r22", "r23", "r24", "r25", "r26", "r27", "gp", "sp",
26 "r30", "r31"
27 };
28
29 #define R(i)    regstrings[i]
30
31
32 dump_ascii(instruction, pc)
33 int instruction, pc;
34 {
35         int addr;
36         char *s;
37         int opcode;
38
39         if  ( longdis )  printf("%08x: %08x  ", pc, instruction);
40         printf("\t");
41         opcode = (unsigned) instruction >> 26;
42         if ( instruction == I_NOP) {
43           printf("nop");
44         }
45         else if  ( opcode == I_SPECIAL )
46         {
47                 opcode = instruction & 0x3f;
48                 printf("%s\t", specialops[opcode]);
49
50                 switch( opcode )
51                 {
52
53                         /* rd,rt,shamt */
54                         case I_SLL:
55                         case I_SRL:
56                         case I_SRA:
57                                 printf("%s,%s,0x%x",
58                                         R(rd(instruction)),
59                                         R(rt(instruction)),
60                                         shamt(instruction));
61                                 break;
62
63                         /* rd,rt,rs */
64                         case I_SLLV:
65                         case I_SRLV:
66                         case I_SRAV:
67                                  printf("%s,%s,%s",
68                                         R(rd(instruction)),
69                                         R(rt(instruction)),
70                                         R(rs(instruction)));
71                                 break;
72
73                         /* rs */
74                         case I_JR:
75                         case I_JALR:
76                         case I_MFLO:
```

```
77                          case I_MTLO:
78                                  printf("%s", R(rs(instruction)));
79                                  break;
80
81                          case I_SYSCALL:
82                          case I_BREAK:
83                                  break;
84
85                          /* rd */
86                          case I_MFHI:
87                          case I_MTHI:
88                                  printf("%s", R(rd(instruction)));
89                                  break;
90
91                          /* rs,rt */
92                          case I_MULT:
93                          case I_MULTU:
94                          case I_DIV:
95                          case I_DIVU:
96                                  printf("%s,%s",
97                                          R(rs(instruction)),
98                                          R(rt(instruction)));
99                                  break;
100
101                         /* rd,rs,rt */
102                         case I_ADD:
103                         case I_ADDU:
104                         case I_SUB:
105                         case I_SUBU:
106                         case I_AND:
107                         case I_OR:
108                         case I_XOR:
109                         case I_NOR:
110                         case I_SLT:
111                         case I_SLTU:
112                                 printf("%s,%s,%s",
113                                         R(rd(instruction)),
114                                         R(rs(instruction)),
115                                         R(rt(instruction)));
116                                 break;
117
118                 }
119         }
120         else if  ( opcode == I_BCOND )
121         {
122                 switch ( rt(instruction) )      /* this field encodes the op */
123                 {
124                     case I_BLTZ:
125                             printf("bltz");
126                             break;
127                     case I_BGEZ:
128                             printf("bgez");
129                             break;
130                     case I_BLTZAL:
131                             printf("bltzal");
132                             break;
133                      case I_BGEZAL:
134                             printf("bgezal");
```

```
135                         break;
136                 default :
137                         printf("BCOND");
138         }
139         printf("\t%s,%08x",
140                R(rs(instruction)),
141                off16(instruction)+pc+4);
142 }
143 else
144 {
145         printf("%s\t", normalops[opcode]);
146
147         switch ( opcode )
148         {
149                 /* 26-bit_target */
150                 case I_J:
151                 case I_JAL:
152                         printf("%08x",
153                                top4(pc)|off26(instruction));
154                         break;
155
156                 /* rs,rt,16-bit_offset */
157                 case I_BEQ:
158                 case I_BNE:
159                         printf("%s,%s,%08x",
160                                R(rt(instruction)),
161                                R(rs(instruction)),
162                                off16(instruction)+pc+4);
163                         break;
164
165                 /* rt,rs,immediate */
166                 case I_ADDI:
167                 case I_ADDIU:
168                 case I_SLTI:
169                 case I_SLTIU:
170                 case I_ANDI:
171                 case I_ORI:
172                 case I_XORI:
173                         printf("%s,%s,0x%x",
174                                 R(rt(instruction)),
175                                 R(rs(instruction)),
176                                 immed(instruction));
177                         break;
178
179                  /* rt, immed */
180                 case I_LUI:
181                         printf("%s,0x%x",
182                                 R(rt(instruction)),
183                                 immed(instruction));
184                         break;
185
186                 /* coprocessor garbage */
187                 case I_COP0:
188                 case I_COP1:
189                 case I_COP2:
190                 case I_COP3:
191                         break;
192
```

```
193                         /* rt,offset(rs) */
194                         case I_LB:
195                         case I_LH:
196                         case I_LWL:
197                         case I_LW:
198                         case I_LBU:
199                          case I_LHU:
200                         case I_LWR:
201                         case I_SB:
202                         case I_SH:
203                         case I_SWL:
204                         case I_SW:
205                         case I_SWR:
206                         case I_LWC0:
207                         case I_LWC1:
208                         case I_LWC2:
209                         case I_LWC3 :
210                         case I_SWC0:
211                         case I_SWC1:
212                         case I_SWC2:
213                         case I_SWC3:
214                                 printf("%s,0x%x(%s)",
215                                         R(rt(instruction)),
216                                         immed(instruction),
217                                         R(rs(instruction)));
218                                 break;
219                 }
220         }
221 }
```

5.5 disasm.c

```
 1 /*
 2  Copyright (c) 1992-1993 The Regents of the University of California.
 3  All rights reserved.  See copyright.h for copyright notice and limitation
 4  of liability and disclaimer of warranty provisions.
 5  */
 6
 7 #include "copyright.h"
 8
 9 /* MIPS instruction disassembler */
10
11 #include <stdio.h>
12 #include <filehdr.h>
13 #include <scnhdr.h>
14 #include <syms.h>
15 #include <ldfcn.h>
16 #include "int.h"
17
18 static FILE *fp;
19 static LDFILE *ldptr;
20 static SCNHDR texthead, rdatahead, datahead, sdatahead, sbsshead, bsshead;
21
22 static char filename[1000] = "a.out";   /* default a.out file */
23 static char self[256];                  /* name of invoking program */
24
25 char mem[MEMSIZE];              /* main memory. use malloc later */
```

```
26 int TRACE, Traptrace, Regtrace;
27 int NROWS=64, ASSOC=1, LINESIZE=4, RAND=0, LRD=0;
28 int pc;
29
30 extern char *strcpy();
31
32 main(argc, argv)
33 int argc;
34 char *argv[];
35 {
36         register char *s;
37         char *fakeargv[3];
38
39         strcpy(self, argv[0]);
40         while  ( argc > 1  &&  argv[1][0] == '-' )
41         {
42                 --argc; ++argv;
43                 for  ( s=argv[0]+1; *s != '\0'; ++s )
44                         switch  ( *s )
45                         {
46                         }
47         }
48
49         if (argc >= 2)
50                 strcpy(filename, argv[1]);
51         fp = fopen(filename, "r");
52         if (fp == NULL)
53         {
54                 fprintf(stderr, "%s: Could not open '%s'\n", self, filename);
55                 exit(0);
56         }
57         fclose(fp);
58         load_program(filename);
59         if  ( argv[1] == NULL )
60         {
61                 fakeargv[1] = "a.out";
62                 fakeargv[2] = NULL;
63                 argv = fakeargv;
64                 ++argc;
65         }
66         disasm(memoffset, argc-1, argv+1); /* where things normally start */
67 }
68
69 #define LOADSECTION(head) load_section(&head);
70
71 load_section(hd)
72 register SCNHDR *hd;
73 {
74   register int pc, i;
75   if  ( hd->s_scnptr != 0 ) {
76     /* printf("loading %s\n", hd->s_name); */
77     pc = hd->s_vaddr;
78     FSEEK(ldptr, hd->s_scnptr, 0);
79     for  ( i=0; i<hd->s_size; ++i ) {
80       if (pc-memoffset >= MEMSIZE)
81         { printf("MEMSIZE too small. Fix and recompile.\n");
82           exit(1); }
83       *(char *) ((mem-memoffset)+pc++) = getc(fp);
```

```
 84     }
 85   }
 86 }
 87
 88 load_program(filename)
 89 char *filename;
 90 {
 91         ldptr = ldopen(filename, NULL);
 92         if  ( ldptr == NULL )
 93         {
 94                 fprintf(stderr, "%s: Load read error on %s\n", self, filename);
 95                 exit(0);
 96         }
 97         if  ( TYPE(ldptr) != 0x162 )
 98         {
 99                 fprintf(stderr,
100                         "big-endian object file (little-endian interp)\n");
101                 exit(0);
102         }
103
104         if  ( ldnshread(ldptr, ".text", &texthead) != 1 )
105                 printf("text section header missing\n");
106         else
107                 LOADSECTION(texthead)
108
109         if  ( ldnshread(ldptr, ".rdata", &rdatahead) != 1 )
110                 printf("rdata section header missing\n");
111         else
112                 LOADSECTION(rdatahead)
113
114         if  ( ldnshread(ldptr, ".data", &datahead) != 1 )
115                 printf("data section header missing\n");
116         else
117                 LOADSECTION(datahead)
118
119         if  ( ldnshread(ldptr, ".sdata", &sdatahead) != 1 )
120                 printf("sdata section header missing\n");
121         else
122                 LOADSECTION(sdatahead)
123
124         if  ( ldnshread(ldptr, ".sbss", &sbsshead) != 1 )
125                 printf("sbss section header missing\n");
126         else
127                 LOADSECTION(sbsshead)
128
129         if  ( ldnshread(ldptr, ".bss", &bsshead) != 1 )
130                 printf("bss section header missing\n");
131         else
132                 LOADSECTION(bsshead)
133
134
135         /* BSS is already zeroed (statically-allocated mem) */
136         /* this version ignores relocation info */
137 }
138
139
140 int *m_alloc(n)
141 int n;
```

```
142 {
143         extern char *malloc();
144 
145         return (int *) (int) malloc((unsigned) n);
146 }
147 
148 disasm(startpc, argc, argv)
149 int startpc, argc;
150 char *argv[];
151 {
152         int i;
153 
154         pc = memoffset;
155         for  ( i=0; i<texthead.s_size; i += 4 )
156         {
157                 dis1(pc);
158                 pc = pc + 4;
159         }
160 }
161 
162 dis1(xpc)
163 int xpc;
164 {
165         register int instr;
166 
167         instr = fetch(pc);
168         dump_ascii(instr, pc);
169         printf("\n");
170 }
171 
172 
```

5.6 encode.h

```
 1 /*
 2  Copyright (c) 1992-1993 The Regents of the University of California.
 3  All rights reserved.  See copyright.h for copyright notice and limitation
 4  of liability and disclaimer of warranty provisions.
 5  */
 6 
 7 #include "copyright.h"
 8 
 9 
10 /* normal opcodes */
11 
12 #define I_SPECIAL       000
13 #define I_BCOND         001
14 #define I_J             002
15 #define I_JAL           003
16 #define I_BEQ           004
17 #define I_BNE           005
18 #define I_BLEZ          006
19 #define I_BGTZ          007
20 #define I_ADDI          010
21 #define I_ADDIU         011
22 #define I_SLTI          012
23 #define I_SLTIU         013
```

```
24 #define I_ANDI          014
25 #define I_ORI           015
26 #define I_XORI          016
27 #define I_LUI           017
28 #define I_COP0          020
29 #define I_COP1          021
30 #define I_COP2          022
31 #define I_COP3          023
32
33 #define I_LB            040
34 #define I_LH            041
35 #define I_LWL           042
36 #define I_LW            043
37 #define I_LBU           044
38 #define I_LHU           045
39 #define I_LWR           046
40
41 #define I_SB            050
42 #define I_SH            051
43 #define I_SWL           052
44 #define I_SW            053
45
46 #define I_SWR           056
47
48 #define I_LWC0          060
49 #define I_LWC1          061
50 #define I_LWC2          062
51 #define I_LWC3          063
52
53 #define I_SWC0          070
54 #define I_SWC1          071
55 #define I_SWC2          072
56 #define I_SWC3          073
57
58 /* special opcodes */
59
60 #define I_SLL           000
61
62 #define I_SRL           002
63 #define I_SRA           003
64 #define I_SLLV          004
65
66 #define I_SRLV          006
67 #define I_SRAV           007
68 #define I_JR            010
69 #define I_JALR          011
70
71 #define I_SYSCALL       014
72 #define I_BREAK         015
73
74 #define I_MFHI          020
75 #define I_MTHI          021
76 #define I_MFLO          022
77 #define I_MTLO          023
78
79 #define I_MULT          030
80 #define I_MULTU         031
81 #define I_DIV           032
```

```
 82 #define I_DIVU          033
 83
 84 #define I_ADD           040
 85 #define I_ADDU          041
 86 #define I_SUB           042
 87 #define I_SUBU          043
 88 #define I_AND           044
 89 #define I_OR            045
 90 #define I_XOR           046
 91 #define I_NOR           047
 92
 93 #define I_SLT           052
 94 #define I_SLTU          053
 95
 96 /* bcond opcodes */
 97
 98
 99 #define I_BLTZ          000
100 #define I_BGEZ          001
101
102 #define I_BLTZAL        020
103 #define I_BGEZAL        021
104
105 /* whole instructions */
106
107 #define I_NOP           000
```

5.7 execute.c

```
  1 /*
  2  Copyright (c) 1992-1993 The Regents of the University of California.
  3  All rights reserved.  See copyright.h for copyright notice and limitation
  4  of liability and disclaimer of warranty provisions.
  5  */
  6
  7 #include "copyright.h"
  8
  9 #include <stdio.h>
 10 #include "instr.h"
 11 #include "encode.h"
 12 #include "int.h"
 13
 14 #define FAST    0
 15 #define true    1
 16 #define false   0
 17
 18
 19 extern char mem[];
 20 extern int TRACE, Regtrace;
 21
 22 /* Machine registers */
 23 int Reg[32];                    /* GPR's */
 24 int HI, LO;                     /* mul/div machine registers */
 25
 26 /* statistics gathering places */
 27 int numjmpls;
 28 int arch1cycles;
```

```
29
30 /* Condition-code calculations */
31 #define  b31(z)         (((z) >>31 )&0x1)       /* extract bit 31 */
32
33 /* code looks funny but is fast thanx to MIPS! */
34 #define cc_add(rr, op1, op2)    \
35         N = (rr < 0);   \
36         Z = (rr == 0);  \
37         C = ((unsigned) rr < (unsigned) op2);   \
38         V = ((op1^op2) >= 0  &&  (op1^rr) < 0);
39
40 #define cc_sub(rr, op1, op2)    \
41         N = (rr < 0);   \
42         Z = (rr == 0);  \
43         V = b31((op1 & ~op2 & ~rr) | (~op1 & op2 & rr));        \
44         C = ((unsigned) op1 < (unsigned) op2);
45
46         /* C = b31((~op1 & op2) | (rr & (~op1 | op2))); /* */
47
48 #define cc_logic(rr)    \
49         N = (rr < 0);   \
50         Z = (rr == 0);  \
51         V = 0;  \
52         C = 0;
53
54 #define cc_mulscc(rr, op1, op2) \
55         N = (rr < 0);   \
56         Z = (rr == 0);  \
57         V = b31((op1 & op2 & ~rr) | (~op1 & ~op2 & rr));        \
58         C = b31((op1 & op2) | (~rr & (op1 | op2)));
59
60
61 runprogram(startpc, argc, argv)
62 int startpc, argc;
63 char *argv[];
64 {
65     int aci, ai, j;
66     register int instr, pc, xpc, npc;
67     register int i;              /* temporary for local stuff */
68     register int icount;
69     extern char *strcpy();
70
71     icount = 0;
72     pc = startpc; npc = pc + 4;
73     i = MEMSIZE - 1024 + memoffset;     /* Initial SP value */
74     Reg[29] = i;                        /* Initialize SP  */
75     /* setup argc and argv stuff (icky!) */
76     store(i, argc);
77     aci = i + 4;
78     ai = aci + 32;
79     for  ( j=0; j<argc; ++j )
80     {
81         strcpy((mem-memoffset)+ai, argv[j]);
82         store(aci, ai);
83         aci += 4;
84         ai += strlen(argv[j]) + 1;
85     }
86
```

```
 87
 88     for  ( ; ; )
 89     {
 90         ++icount;
 91         xpc = pc; pc = npc; npc = pc + 4;
 92         instr = ifetch(xpc);
 93         Reg[0] = 0;     /* Force r0 = 0 */
 94
 95         if  ( instr != 0 )      /* eliminate no-ops */
 96         {
 97         switch ( (instr>>26) & 0x0000003f)
 98         {
 99                 case I_SPECIAL:
100                 {
101                     switch ( instr & 0x0000003f )
102                     {
103
104                         case I_SLL:
105                             Reg[rd(instr)] = Reg[rt(instr)] << shamt(instr);
106                             break;
107                         case I_SRL:
108                             Reg[rd(instr)] =
109                                 (unsigned) Reg[rt(instr)] >> shamt(instr);
110                             break;
111                         case I_SRA:
112                             Reg[rd(instr)] = Reg[rt(instr)] >> shamt(instr);
113                             break;
114                         case I_SLLV:
115                             Reg[rd(instr)] = Reg[rt(instr)] << Reg[rs(instr)];
116                             break;
117                         case I_SRLV:
118                             Reg[rd(instr)] =
119                                 (unsigned) Reg[rt(instr)] >> Reg[rs(instr)];
120                             break;
121                         case I_SRAV:
122                             Reg[rd(instr)] = Reg[rt(instr)] >> Reg[rs(instr)];
123                             break;
124                         case I_JR:
125                             npc = Reg[rs(instr)];
126                             break;
127                         case I_JALR:
128                             npc = Reg[rs(instr)];
129                             Reg[rd(instr)] = xpc + 8;
130                             break;
131
132                         case I_SYSCALL: system_trap(); break;
133                          case I_BREAK: system_break(); break;
134
135                         case I_MFHI: Reg[rd(instr)] = HI; break;
136                         case I_MTHI: HI = Reg[rs(instr)]; break;
137                         case I_MFLO: Reg[rd(instr)] = LO; break;
138                         case I_MTLO: LO = Reg[rs(instr)]; break;
139
140                         case I_MULT:
141                             {
142                                 int t1, t2;
143                                 int t1l, t1h, t2l, t2h;
144                                 int neg;
```

```
145
146                             t1 = Reg[rs(instr)];
147                             t2 = Reg[rt(instr)];
148                             neg = 0;
149                             if ( t1 < 0 ) { t1 = -t1 ; neg = !neg; }
150                             if ( t2 < 0 ) { t2 = -t2 ; neg = !neg; }
151                             LO = t1 * t2;
152                             t1l = t1 & 0xffff;
153                             t1h = (t1 >> 16) & 0xffff;
154                             t2l = t2 & 0xffff;
155                             t2h = (t2 >> 16) & 0xffff;
156                             HI = t1h*t2h+((t1h*t2l)>>16)+((t2h*t1l)>>16);
157                             if ( neg )
158                             {
159                                     LO = ~LO; HI = ~HI; LO = LO + 1;
160                                     if  ( LO == 0 )  HI = HI + 1;
161                             }
162                         }
163                         break;
164                     case I_MULTU:
165                         {
166                             int t1, t2;
167                             int t1l, t1h, t2l, t2h;
168
169                             t1 = Reg[rs(instr)];
170                             t2 = Reg[rt(instr)];
171                             t1l = t1 & 0xffff;
172                             t1h = (t1 >> 16) & 0xffff;
173                             t2l = t2 & 0xffff;
174                             t2h = (t2 >> 16) & 0xffff;
175                             LO = t1*t2;
176                             HI = t1h*t2h+((t1h*t2l)>>16)+((t2h*t1l)>>16);
177                         }break;
178                     case I_DIV:
179                         LO = Reg[rs(instr)] / Reg[rt(instr)];
180                         HI = Reg[rs(instr)] % Reg[rt(instr)];
181                         break;
182                     case I_DIVU:
183                         LO =
184                         (unsigned)Reg[rs(instr)] / (unsigned)Reg[rt(instr)];
185                         HI =
186                         (unsigned)Reg[rs(instr)] % (unsigned)Reg[rt(instr)];
187                         break;
188
189                     case I_ADD:
190                     case I_ADDU:
191                         Reg[rd(instr)] = Reg[rs(instr)] + Reg[rt(instr)];
192                         break;
193                     case I_SUB:
194                     case I_SUBU:
195                         Reg[rd(instr)] = Reg[rs(instr)] - Reg[rt(instr)];
196                         break;
197                     case I_AND:
198                         Reg[rd(instr)] = Reg[rs(instr)] & Reg[rt(instr)];
199                          break;
200                     case I_OR:
201                         Reg[rd(instr)] = Reg[rs(instr)] | Reg[rt(instr)];
202                         break;
```

```
203                         case I_XOR:
204                             Reg[rd(instr)] = Reg[rs(instr)] ^ Reg[rt(instr)];
205                             break;
206                         case I_NOR:
207                             Reg[rd(instr)] = ~(Reg[rs(instr)] | Reg[rt(instr)]);
208                             break;
209 
210                         case I_SLT:
211                             Reg[rd(instr)] = (Reg[rs(instr)] < Reg[rt(instr)]);
212                             break;
213                         case I_SLTU:
214                             Reg[rd(instr)] =
215                                 ((unsigned) Reg[rs(instr)]
216                                         < (unsigned) Reg[rt(instr)]);
217                             break;
218                         default: u(); break;
219                     }
220                 } break;
221 
222                 case I_BCOND:
223                 {
224                     switch ( rt(instr) )        /* this field encodes the op */
225                     {
226                         case I_BLTZ:
227                                 if ( Reg[rs(instr)] < 0 )
228                                         npc = xpc + 4 + (immed(instr)<<2);
229                                 break;
230                         case I_BGEZ:
231                                 if ( Reg[rs(instr)] >= 0 )
232                                         npc = xpc + 4 + (immed(instr)<<2);
233                                 break;
234 
235                         case I_BLTZAL:
236                                 Reg[31] = xpc + 8;
237                                 if ( Reg[rs(instr)] < 0 )
238                                         npc = xpc + 4 + (immed(instr)<<2);
239                                 break;
240                         case I_BGEZAL:
241                                 Reg[31] = xpc + 8;
242                                 if ( Reg[rs(instr)] >= 0 )
243                                         npc = xpc + 4 + (immed(instr)<<2);
244                                 break;
245                         default: u(); break;
246                     }
247 
248                 } break;
249 
250                 case I_J:
251                         npc = (xpc & 0xf0000000) | ((instr & 0x03ffffff) << 2);
252                         break;
253                 case I_JAL:
254                         Reg[31] = xpc + 8;
255                         npc = (xpc & 0xf0000000) | ((instr & 0x03ffffff) << 2);
256                         break;
257                 case I_BEQ:
258                         if  ( Reg[rs(instr)] == Reg[rt(instr)] )
259                                 npc = xpc + 4 + (immed(instr) << 2);
260                         break;
```

```
261                 case I_BNE:
262                         if  ( Reg[rs(instr)] != Reg[rt(instr)] )
263                                 npc = xpc + 4 + (immed(instr) << 2);
264                         break;
265                  case I_BLEZ:
266                         if  ( Reg[rs(instr)] <= 0 )
267                                 npc = xpc + 4 + (immed(instr) << 2);
268                         break;
269                 case I_BGTZ:
270                         if  ( Reg[rs(instr)] > 0 )
271                                 npc = xpc + 4 + (immed(instr) << 2);
272                         break;
273                 case I_ADDI:
274                         Reg[rt(instr)] = Reg[rs(instr)] + immed(instr);
275                         break;
276                 case I_ADDIU:
277                         Reg[rt(instr)] = Reg[rs(instr)] + immed(instr);
278                         break;
279                 case I_SLTI:
280                         Reg[rt(instr)] = (Reg[rs(instr)] < immed(instr));
281                         break;
282                 case I_SLTIU:
283                         Reg[rt(instr)] =
284                           ((unsigned) Reg[rs(instr)] < (unsigned) immed(instr));
285                         break;
286                 case I_ANDI:
287                         Reg[rt(instr)] = Reg[rs(instr)] & immed(instr);
288                         break;
289                 case I_ORI:
290                         Reg[rt(instr)] = Reg[rs(instr)] | immed(instr);
291                         break;
292                 case I_XORI:
293                         Reg[rt(instr)] = Reg[rs(instr)] ^ immed(instr);
294                         break;
295                 case I_LUI:
296                         Reg[rt(instr)] = instr << 16;
297                         break;
298
299                 case I_LB:
300                         Reg[rt(instr)] = cfetch(Reg[rs(instr)] + immed(instr));
301                         break;
302                 case I_LH:
303                         Reg[rt(instr)] = sfetch(Reg[rs(instr)] + immed(instr));
304                         break;
305                 case I_LWL:
306                     i = Reg[rs(instr)] + immed(instr);
307                     Reg[rt(instr)] &= (-1 >> 8*((-i) & 0x03));
308                     Reg[rt(instr)] |= ((fetch(i & 0xfffffffc)) << 8*(i & 0x03));
309                     break;
310                 case I_LW:
311                         Reg[rt(instr)] = fetch(Reg[rs(instr)] + immed(instr));
312                         break;
313                 case I_LBU:
314                         Reg[rt(instr)] = ucfetch(Reg[rs(instr)] + immed(instr));
315                         break;
316                 case I_LHU:
317                         Reg[rt(instr)] = usfetch(Reg[rs(instr)] + immed(instr));
318                         break;
```

```
319                 case I_LWR:
320                     i = Reg[rs(instr)] + immed(instr);
321                     Reg[rt(instr)] &= (-1 << 8*(i & 0x03));
322                     if  ( (i & 0x03)== 0 )
323                         Reg[rt(instr)] = 0;
324                     Reg[rt(instr)] |=
325                         ((fetch(i & 0xfffffffc)) >> 8*((-i) & 0x03));
326                     break;
327 
328                 case I_SB:
329                         cstore(Reg[rs(instr)] + immed(instr), Reg[rt(instr)]);
330                         break;
331                  case I_SH:
332                         sstore(Reg[rs(instr)] + immed(instr), Reg[rt(instr)]);
333                         break;
334                 case I_SWL:
335                         fprintf(stderr, "sorry, no SWL yet.\n");
336                         u();
337                         break;
338                 case I_SW:
339                         store(Reg[rs(instr)] + immed(instr), Reg[rt(instr)]);
340                         break;
341 
342                 case I_SWR:
343                         fprintf(stderr, "sorry, no SWR yet.\n");
344                         u();
345                         break;
346 
347                 case I_LWC0: case I_LWC1:
348                 case I_LWC2: case I_LWC3:
349                 case I_SWC0: case I_SWC1:
350                 case I_SWC2: case I_SWC3:
351                 case I_COP0: case I_COP1:
352                 case I_COP2: case I_COP3:
353                         fprintf(stderr, "Sorry, no coprocessors.\n");
354                         exit(2);
355                         break;
356 
357                 default: u(); break;
358         }
359         }
360 
361 #ifdef DEBUG
362 /*
363 printf(" %d(%x) = %d(%x) op  %d(%x)\n", Reg[rd], Reg[rd], op1, op1, op2, op2);
364 /* */
365 #endif
366 #if !FAST
367         if  ( TRACE )
368         {
369             dump_ascii(instr, xpc); printf("\n"); /* */
370             if  ( Regtrace )  dump_reg();
371         }
372 #endif
373     }
374 }
375 
376 
```

```
377
378 u()                            /* unimplemented */
379 {
380         printf("Unimplemented Instruction\n"); exit(2);
381 }
382
383 ny()
384 {
385         printf("This opcode not implemeted yet.\n"); exit(2);
386 }
387
388
389 /* debug aids */
390
391 RS(i)
392 int i;
393 {
394         return rs(i);
395 }
396
397 RT(i)
398 int i;
399 {
400         return rt(i);
401 }
402
403 RD(i)
404 int i;
405 {
406         return rd(i);
407 }
408
409 IM(i)
410 int i;
411 {
412         return immed(i);
413 }
414
415
416
417 dump_reg()
418 {
419         int j;
420
421         printf(" 0:"); for  ( j=0; j<8; ++j ) printf(" %08x", Reg[j]);
422                 printf("\n");
423         printf(" 8:"); for  ( ; j<16; ++j ) printf(" %08x", Reg[j]);
424                 printf("\n");
425         printf("16:"); for  ( ; j<24; ++j ) printf(" %08x", Reg[j]);
426                 printf("\n");
427         printf("24:"); for  ( ; j<32; ++j ) printf(" %08x", Reg[j]);
428                 printf("\n");
429
430 }
431
432
433
434 /*
```

```
435         0 -> 0
436         1 -> 1
437         2 -> 1
438         3 -> 2
439         4 -> 2
440         5 -> 2
441         6 -> 2
442         7 -> 3
443         8 -> 3
444         9 -> 3  ...
445         Treats all ints as unsigned numbers.
446 */
447 ilog2(i)
448 int i;
449 {
450         int j, l;
451
452         if  ( i == 0 )  return 0;
453         j = 0;
454         l = 1;
455         if  ( (j=(i&0xffff0000)) != 0 ) { i = j; l += 16; }
456         if  ( (j=(i&0xff00ff00)) != 0 ) { i = j; l += 8; }
457         if  ( (j=(i&0xf0f0f0f0)) != 0 ) { i = j; l += 4; }
458         if  ( (j=(i&0xcccccccc)) != 0 ) { i = j; l += 2; }
459         if  ( (j=(i&0xaaaaaaaa)) != 0 ) { i = j; l += 1; }
460         return l;
461 }
462
463
464
465 #define NH     32
466 #define NNN     33
467
468 static int hists[NH][NNN];
469 int hoflo[NH], htotal[NH];
470
471 void henters(n, hist)
472 int n, hist;
473 {
474         if  ( 0 <= n  &&  n < NNN ) ++hists[hist][n]; else ++hoflo[hist];
475         ++htotal[hist];
476 }
477
478 hprint()
479 {
480         int h, i;
481         double I;
482
483         for  ( h=0; h<=NH; ++h ) if  ( htotal[h] > 0 )
484         {
485                 printf("\nhisto %d:\n", h);
486                 I = 0.0;
487                 for  ( i=0; i<NNN; ++i )
488                 {
489                         I += hists[h][i];
490                         printf("%d\t%d\t%5.2f%%\t%5.2f%%\n",
491                                 i, hists[h][i],
492                                 (double) 100*hists[h][i] / htotal[h],
```

```
493                                 (double) 100*I/htotal[h]);
494                 }
495                 printf("oflo %d:\t%d/%d\t%5.2f%%\n",
496                         h, hoflo[h], htotal[h],
497                         (double) 100*hoflo[h] / htotal[h]);
498         }
499 }
500
501 int numadds=1, numsubs=1, numsuccesses, numcarries;
502 int addtable[33][33];
503 int subtable[33][33];
504
505 char fmt[] = "%6d";
506 char fmt2[] = "------";
507
508 patable(tab)
509 int tab[33][33];
510 {
511         int i, j;
512
513         printf("  |");
514         for  ( j=0; j<33; ++j )
515                 printf(fmt, j);
516         putchar('\n');
517         printf("  |");
518         for  ( j=0; j<33; ++j )
519                 printf(fmt2);
520         putchar('\n');
521         for  ( i=0; i<33; ++i )
522         {
523                 printf("%2d|", i);
524                 for  ( j=0; j<33; ++j )
525                         printf(fmt, tab[i][j]);
526                 putchar('\n');
527         }
528 }
529
530
531
532
533 printstatistics()
534 {
535         /*
536         printhist();
537         /*
538         printf("numjmpls = %d / %d = %5.2f%%\n",
539                 numjmpls, arch1cycles, 100.0*numjmpls/arch1cycles);
540         printf("numadds = %d, numsubs = %d, numcycles = %d, frac = %5.2f%%\n",
541                 numadds, numsubs,
542                 arch1cycles, (double) 100 * (numadds+numsubs) / arch1cycles);
543         printf("numsuccesses = %d (%5.2f%%) numcarries = %d\n",
544                 numsuccesses, 100.0*numsuccesses/(numadds+numsubs), numcarries);
545
546         /*
547         hprint();
548         printf("\nADD table:\n");patable(addtable);
549         printf("\nSUB table:\n");patable(subtable);
550         */
```

```
551 }
552
553
554
555 #define NNNN    (64)
556
557 static int hist[NNNN];
558
559 henter(n)
560 int n;
561 {
562         if  ( 0 <= n  &&  n < NNNN )
563                 ++hist[n];
564 }
565
566 printhist()
567 {
568         int i;
569
570         for  ( i=0; i<NNNN; ++i )
571                 printf("%d %d\n", i, hist[i]);
572 }
573
574
```

5.8 instr.h

```
 1 /*
 2  Copyright (c) 1992-1993 The Regents of the University of California.
 3  All rights reserved.  See copyright.h for copyright notice and limitation
 4  of liability and disclaimer of warranty provisions.
 5  */
 6
 7 #include "copyright.h"
 8
 9 /* Instruction formats */
10
11 #define rd(i)           (((i) >> 11) & 0x1f)
12 #define rt(i)           (((i) >> 16) & 0x1f)
13 #define rs(i)           (((i) >> 21) & 0x1f)
14 #define shamt(i)        (((i) >> 6) & 0x1f)
15 #define immed(i)        (((i) & 0x8000) ? (i)|(-0x8000) : (i)&0x7fff)
16
17 #define off26(i)        (((i)&((1<<26)-1))<<2)
18 #define top4(i)         (((i)&(~((1<<28)-1))))
19 #define off16(i)        (immed(i)<<2)
20
21 #define extend(i, hibitmask)    (((i)&(hibitmask)) ? ((i)|(-(hibitmask))) : (i))
```

5.9 int.h

```
 1
 2 /*
 3  Copyright (c) 1992-1993 The Regents of the University of California.
 4  All rights reserved.  See copyright.h for copyright notice and limitation
 5  of liability and disclaimer of warranty provisions.
 6  */
```

```
 7
 8 #include "copyright.h"
 9
10
11 #define MEMSIZE (1<<24)
12 #define memoffset 0x10000000
13
14 /* centralized memory-access primitives */
15 #define amark(x)        x
16 #define imark(x)        x
17
18 #define ifetch(addr)    (*(int *)(int) (&(mem-memoffset)[imark(addr)]))
19 #define fetch(addr)     (*(int *)(int) (&(mem-memoffset)[amark(addr)]))
20 #define sfetch(addr)    (*(short *)(int) (&(mem-memoffset)[amark(addr)]))
21 #define usfetch(addr)   (*(unsigned short *)(int)(&(mem-memoffset)[amark(addr)]))
22 #define cfetch(addr)    (*(char *)(int) (&(mem-memoffset)[amark(addr)]))
23 #define ucfetch(addr)   (*(unsigned char *)(int)(&(mem-memoffset)[amark(addr)]))
24
25 #define store(addr, i)  \
26         ((*(int *)(int) (&(mem-memoffset)[amark(addr)]) = (i)))
27 #define sstore(addr, i) \
28         ((*(short *)(int) (&(mem-memoffset)[amark(addr)]) = (i)))
29 #define cstore(addr, i) \
30         (((mem-memoffset)[amark(addr)] = (i)))
31
```

5.10 main.c

```
 1 /*
 2  Copyright (c) 1992-1993 The Regents of the University of California.
 3  All rights reserved.  See copyright.h for copyright notice and limitation
 4  of liability and disclaimer of warranty provisions.
 5  */
 6
 7 #include "copyright.h"
 8
 9 /* MIPS instruction interpreter */
10
11 #include <stdio.h>
12 #include <filehdr.h>
13 #include <scnhdr.h>
14 #include <syms.h>
15 #include <ldfcn.h>
16 #include "int.h"
17
18 static FILE *fp;
19 static LDFILE *ldptr;
20 static SCNHDR texthead, rdatahead, datahead, sdatahead, sbsshead, bsshead;
21
22 static char filename[1000] = "a.out";   /* default a.out file */
23 static char self[256];                  /* name of invoking program */
24
25 char mem[MEMSIZE];              /* main memory. use malloc later */
26 int TRACE, Traptrace, Regtrace;
27 int NROWS=64, ASSOC=1, LINESIZE=4, RAND=0, LRD=0;
28
29 extern char *strcpy();
```

```
30
31 main(argc, argv)
32 int argc;
33 char *argv[];
34 {
35         register char *s;
36         char *fakeargv[3];
37
38         strcpy(self, argv[0]);
39         while  ( argc > 1  &&  argv[1][0] == '-' )
40         {
41                 --argc; ++argv;
42                 for  ( s=argv[0]+1; *s != '\0'; ++s )
43                         switch  ( *s )
44                         {
45                                 case 't':  TRACE = 1; break;
46                                 case 'T':  Traptrace = 1; break;
47                                 case 'r':  Regtrace = 1; break;
48                                 case 'm':
49                                         NROWS = atoi(*++argv);
50                                         ASSOC = atoi(*++argv);
51                                         LINESIZE = atoi(*++argv);
52                                         RAND = ((*++argv)[0] == 'r');
53                                         LRD = ((*argv)[0] == 'l')
54                                            && ((*argv)[1] == 'r')
55                                            && ((*argv)[2] == 'd');
56                                         argc -= 4;
57                                         break;
58                         }
59         }
60
61         if (argc >= 2)
62                 strcpy(filename, argv[1]);
63         fp = fopen(filename, "r");
64         if (fp == NULL)
65         {
66                 fprintf(stderr, "%s: Could not open '%s'\n", self, filename);
67                  exit(0);
68         }
69         fclose(fp);
70         load_program(filename);
71         if  ( argv[1] == NULL )
72         {
73                 fakeargv[1] = "a.out";
74                 fakeargv[2] = NULL;
75                 argv = fakeargv;
76                 ++argc;
77         }
78         runprogram(memoffset, argc-1, argv+1); /* where things normally start */
79 }
80
81 char *string(s)
82 char *s;
83 {
84         char *p;
85         extern char *malloc();
86
87         p = malloc((unsigned) strlen(s)+1);
```

```
 88         strcpy(p, s);
 89         return p;
 90 }
 91
 92 load_program(filename)
 93 char *filename;
 94 {
 95         register int pc, i, j, strindex, stl;
 96         char str[1111];
 97         int rc1, rc2;
 98
 99         ldptr = ldopen(filename, NULL);
100         if  ( ldptr == NULL )
101         {
102                 fprintf(stderr, "%s: Load read error on %s\n", self, filename);
103                 exit(0);
104         }
105         if  ( TYPE(ldptr) != 0x162 )
106         {
107                 fprintf(stderr,
108                         "big-endian object file (little-endian interp)\n");
109                 exit(0);
110         }
111
112 #define LOADSECTION(head)       \
113         if  ( head.s_scnptr != 0 ) \
114         { \
115                 /* printf("loading %s\n", head.s_name); /* */ \
116                 pc = head.s_vaddr; \
117                 FSEEK(ldptr, head.s_scnptr, 0); \
118                 for  ( i=0; i<head.s_size; ++i ) \
119                         *(char *) ((mem-memoffset)+pc++) = getc(fp); \
120                 if (pc-memoffset >= MEMSIZE) \
121                         { printf("MEMSIZE too small. Fix and recompile.\n"); \
122                         exit(1); } \
123         }
124
125         if  ( ldnshread(ldptr, ".text", &texthead) != 1 )
126                 printf("text section header missing\n");
127         else
128                 LOADSECTION(texthead)
129
130         if  ( ldnshread(ldptr, ".rdata", &rdatahead) != 1 )
131                 printf("rdata section header missing\n");
132         else
133                  LOADSECTION(rdatahead)
134
135         if  ( ldnshread(ldptr, ".data", &datahead) != 1 )
136                 printf("data section header missing\n");
137         else
138                 LOADSECTION(datahead)
139
140         if  ( ldnshread(ldptr, ".sdata", &sdatahead) != 1 )
141                 printf("sdata section header missing\n");
142         else
143                 LOADSECTION(sdatahead)
144
145         if  ( ldnshread(ldptr, ".sbss", &sbsshead) != 1 )
```

```
146                 printf("sbss section header missing\n");
147         else
148                 LOADSECTION(sbsshead)
149
150         if  ( ldnshread(ldptr, ".bss", &bsshead) != 1 )
151                 printf("bss section header missing\n");
152         else
153                 LOADSECTION(bsshead)
154
155         /* BSS is already zeroed (statically-allocated mem) */
156         /* this version ignores relocation info */
157 }
158
159
160 int *m_alloc(n)
161 int n;
162 {
163         extern char *malloc();
164
165         return (int *) (int) malloc((unsigned) n);
166 }
167
```

5.11 noff.h

```
 1 /* noff.h
 2  *     Data structures defining the Nachos Object Code Format
 3  *
 4  *     Basically, we only know about three types of segments:
 5  *      code (read-only), initialized data, and unitialized data
 6  */
 7
 8 #define NOFFMAGIC       0xbadfad        /* magic number denoting Nachos
 9                                          * object code file
10                                          */
11
12 typedef struct segment {
13   int virtualAddr;              /* location of segment in virt addr space */
14   int inFileAddr;               /* location of segment in this file */
15   int size;                     /* size of segment */
16 } Segment;
17
18 typedef struct noffHeader {
19    int noffMagic;               /* should be NOFFMAGIC */
20    Segment code;                /* executable code segment */
21    Segment initData;            /* initialized data segment */
22    Segment uninitData;          /* uninitialized data segment --
23                                  * should be zero'ed before use
24                                  */
25 } NoffHeader;
```

5.12 opstrings.c

```
1 /*
2  Copyright (c) 1992-1993 The Regents of the University of California.
3  All rights reserved.  See copyright.h for copyright notice and limitation
4  of liability and disclaimer of warranty provisions.
```

```
 */

#include "copyright.h"

char *normalops[] = {
        "special",
        "bcond",
        "j",
        "jal",
        "beq",
        "bne",
        "blez",
        "bgtz",
        "addi",
        "addiu",
        "slti",
        "sltiu",
        "andi",
        "ori",
        "xori",
        "lui",
        "cop0",
        "cop1",
        "cop2",
        "cop3",
        "024",
        "025",
        "026",
        "027",
        "030",
        "031",
        "032",
        "033",
        "034",
        "035",
        "036",
        "037",
        "lb",
        "lh",
        "lwl",
        "lw",
        "lbu",
        "lhu",
        "lwr",
        "047",
        "sb",
        "sh",
        "swl",
        "sw",
        "054",
        "055",
        "swr",
        "057",
        "lwc0",
        "lwc1",
        "lwc2",
        "lwc3",
        "064",
```

```
 63         "065",
 64         "066",
 65         "067",
 66         "swc0",
 67          "swc1",
 68         "swc2",
 69         "swc3",
 70         "074",
 71         "075",
 72         "076",
 73         "077"
 74 };
 75
 76 char *specialops[] = {
 77         "sll",
 78         "001",
 79         "srl",
 80         "sra",
 81         "sllv",
 82         "005",
 83         "srlv",
 84         "srav",
 85         "jr",
 86         "jalr",
 87         "012",
 88         "013",
 89         "syscall",
 90         "break",
 91         "016",
 92         "017",
 93         "mfhi",
 94         "mthi",
 95         "mflo",
 96         "mtlo",
 97         "024",
 98         "025",
 99         "026",
100         "027",
101         "mult",
102         "multu",
103         "div",
104         "divu",
105         "034",
106         "035",
107         "036",
108         "037",
109         "add",
110         "addu",
111         "sub",
112         "subu",
113         "and",
114         "or",
115         "xor",
116         "nor",
117         "050",
118         "051",
119         "slt",
120         "sltu",
```

```
121         "054",
122         "055",
123         "056",
124         "057",
125         "060",
126         "061",
127         "062",
128         "063",
129         "064",
130         "065",
131         "066",
132         "067",
133          "070",
134         "071",
135         "072",
136         "073",
137         "074",
138         "075",
139         "076",
140         "077",
141 };
142
```

5.13 out.c

```
  1 /*
  2  Copyright (c) 1992-1993 The Regents of the University of California.
  3  All rights reserved.  See copyright.h for copyright notice and limitation
  4  of liability and disclaimer of warranty provisions.
  5  */
  6
  7 #define MAIN
  8 #include "copyright.h"
  9 #undef MAIN
 10
 11 /*
 12  * OUT.C
 13  * Looking at a.out formats.
 14  *
 15  * First task:
 16  * Look at mips COFF stuff:
 17  * Print out the contents of a file and do the following:
 18  *    For data, print the value and give relocation information
 19  *    For code, disassemble and give relocation information
 20  */
 21
 22 #include <filehdr.h>
 23 #include <aouthdr.h>
 24 #include <scnhdr.h>
 25 #include <reloc.h>
 26 #include <syms.h>
 27 #include <stdio.h>
 28
 29 #define read_struct(f,s) (fread(&s,sizeof(s),1,f)==1)
 30
 31 #define MAXRELOCS 1000
 32
```

```
33
34 #define MAXDATA 10000
35
36 struct data {
37   long data[MAXDATA];
38   struct reloc reloc[MAXRELOCS];
39   int length;
40   int relocs;
41 };
42
43 #define MAXSCNS 10
44 #define MAXSYMS 300
45 #define MAXSSPACE 20000
46
47 struct filehdr filehdr;
48 struct aouthdr aouthdr;
49 struct scnhdr scnhdr[MAXSCNS];
50 struct data section[MAXSCNS];
51 HDRR symhdr;
52 EXTR symbols[MAXSYMS];
53 char sspace[20000];
54
55 char *symbol_type[] = {
56   "Nil", "Global", "Static", "Param", "Local", "Label", "Proc", "Block",
57   "End", "Member", "Type", "File", "Register", "Forward", "StaticProc",
58   "Constant" };
59
60 char *storage_class[] = {
61   "Nil", "Text", "Data", "Bss", "Register", "Abs", "Undefined", "CdbLocal",
62   "Bits", "CdbSystem", "RegImage", "Info", "UserStruct", "SData", "SBss",
63   "RData", "Var", "Common", "SCommon", "VarRegister", "Variant", "SUndefined",
64   "Init" };
65
66 main(argc,argv)
67 int argc;
68 char *argv[];
69 {
70   char *filename = "a.out";
71   FILE *f;
72   int i;
73   long l;
74 /*   EXTR filesym; */
75   char buf[100];
76
77   if (argc == 2) filename = argv[1];
78   if ((f = fopen(filename,"r")) == NULL) {
79     printf("out: could not open %s\n",filename);
80     perror("out");
81     exit(1);
82   }
83   if (!read_struct(f,filehdr) ||
84       !read_struct(f,aouthdr) ||
85       filehdr.f_magic != MIPSELMAGIC) {
86     printf("out: %s is not a MIPS Little-Endian COFF object file\n",filename);
87     exit(1);
88   }
89   if (filehdr.f_nscns > MAXSCNS) {
90     printf("out: Too many COFF sections.\n");
```

```
 91     exit(1);
 92   }
 93   for (i=0; i < filehdr.f_nscns; ++i) {
 94     read_struct(f,scnhdr[i]);
 95     if (scnhdr[i].s_size > MAXDATA*sizeof(long) &&
 96         scnhdr[i].s_scnptr != 0 ||
 97         scnhdr[i].s_nreloc > MAXRELOCS) {
 98       printf("section %s is too big.\n",scnhdr[i].s_name);
 99       exit(1);
100     }
101   }
102   for (i=0; i < filehdr.f_nscns; ++i) {
103     if (scnhdr[i].s_scnptr != 0) {
104       section[i].length = scnhdr[i].s_size/4;
105       fseek(f,scnhdr[i].s_scnptr,0);
106       fread(section[i].data,sizeof(long),section[i].length,f);
107       section[i].relocs = scnhdr[i].s_nreloc;
108       fseek(f,scnhdr[i].s_relptr,0);
109       fread(section[i].reloc,sizeof(struct reloc),section[i].relocs,f);
110     } else {
111       section[i].length = 0;
112     }
113   }
114   fseek(f,filehdr.f_symptr,0);
115   read_struct(f,symhdr);
116   if (symhdr.iextMax > MAXSYMS) {
117     printf("too many symbols to store.\n");
118   }
119   fseek(f,symhdr.cbExtOffset,0);
120   for (i=0; i < MAXSYMS && i<symhdr.iextMax; ++i) {
121     read_struct(f,symbols[i]);
122   }
123   if (symhdr.issExtMax > MAXSSPACE) {
124     printf("too large a string space.\n");
125     exit(1);
126   }
127   fseek(f,symhdr.cbSsExtOffset,0);
128   fread(sspace,1,symhdr.issExtMax,f);
129
130   for (i=0; i<filehdr.f_nscns; ++i) {
131     print_section(i);
132   }
133
134   printf("External Symbols:\nValue\t Type\t\tStorage Class\tName\n");
135   for (i=0; i < MAXSYMS && i < symhdr.iextMax; ++i) {
136     SYMR *sym = &symbols[i].asym;
137     if (sym->sc == scUndefined) myprintf("\t ");
138     else myprintf("%08x ",sym->value);
139     myprintf("%s",symbol_type[sym->st]);
140     mytab(25);
141     myprintf("%s",storage_class[sym->sc]);
142     mytab(41);
143     myprintf("%s\n",&sspace[sym->iss]);
144   }
145   return 0;
146 }
147
148 static column = 1;
```

```
149 static FILE *outfile = stdout;
150
151 #include <varargs.h>
152 /*VARARGS0*/
153 myprintf(va_alist)
154 va_dcl
155 {
156   va_list ap;
157   char *form;
158   char buf[100];
159
160   va_start(ap);
161   form = va_arg(ap,char *);
162   vsprintf(buf,form,ap);
163   va_end(ap);
164
165   fputs(buf,outfile);
166
167   for (form = buf; *form != '\0'; ++form) {
168     if (*form == '\n') column = 1;
169     else if (*form == '\t') column = ((column + 7)&~7)+1;
170     else column += 1;
171   }
172 }
173
174 mytab(n)
175 int n;
176 {
177   while (column < n) {
178     fputc(' ',outfile);
179     ++column;
180   }
181   return column == n;
182 }
183
184 mysetfile(f)
185 FILE *f;
186 {
187   outfile = f;
188 }
189
190 #define printf myprintf
191 #include "d.c"
192
193 print_section(i)
194 int i;
195 {
196   int j,k;
197   int is_text;
198   long pc;
199   long word;
200   char *s;
201
202   printf("Section: %s\t%d/%d\n",scnhdr[i].s_name,
203          scnhdr[i].s_size,section[i].relocs);
204   is_text =  (strncmp(scnhdr[i].s_name,".text",5) == 0);
205
206   for (j=0; j < section[i].length; ++j) {
```

```
207     pc = scnhdr[i].s_vaddr+j*4;
208     word = section[i].data[j];
209     if (is_text) {
210       dump_ascii(word,pc);
211     } else {
212       printf("%08x: %08x  ", pc,word);
213       s = (char *)&word;
214       for (k=0;k<4;++k) {
215         if (s[k] >= ' ' && s[k] < 127) printf("%c",s[k]);
216         else printf(".");
217       }
218       printf("\t%d",word);
219     }
220     print_reloc(pc,i,j);
221   }
222 }
223
224 char *section_name[] = {
225   "(null)", ".text", ".rdata", ".data", ".sdata", ".sbss", ".bss",
226   ".init", ".lit8", ".lit4"
227 };
228
229 char *reloc_type[] = {
230   "abs", "16", "32", "26", "hi16", "lo16", "gpdata", "gplit"
231 };
232
233 print_reloc(vaddr,i,j)
234 int i,j;
235 {
236   int k;
237   struct reloc *rp;
238   for (k=0; k < section[i].relocs; ++k) {
239     rp = &section[i].reloc[k];
240     if (vaddr == rp->r_vaddr) {
241       mytab(57);
242       if (rp->r_extern) {
243         if (rp->r_symndx >= MAXSYMS) {
244           printf("sym $%d",rp->r_symndx);
245         } else {
246           printf("\"%s\"",&sspace[symbols[rp->r_symndx].asym.iss]);
247         }
248       } else {
249         printf("%s",section_name[rp->r_symndx]);
250       }
251       printf(" %s",reloc_type[rp->r_type]);
252       break;
253     }
254   }
255   printf("\n");
256 }
```

5.14 system.c

```
1 /*
2  Copyright (c) 1992-1993 The Regents of the University of California.
3  All rights reserved.  See copyright.h for copyright notice and limitation
4  of liability and disclaimer of warranty provisions.
```

```
 5  */
 6
 7 #include "copyright.h"
 8 #include <stdio.h>
 9 #include <syscall.h>
10 #include "int.h"
11
12 extern int Reg[];
13 extern char mem[];
14 extern int Traptrace;
15
16 char *u_to_int_addr();
17
18 /* handle system calls */
19 system_break()
20 {
21         if  ( Traptrace )
22                 printf("**breakpoint ");
23         system_trap();
24 }
25
26 system_trap()
27 {
28         int o0, o1, o2;         /* user out register values */
29         int syscallno;
30         extern long lseek();
31
32         if  ( Traptrace )
33         {
34                 printf("**System call %d\n", Reg[2]);
35                 dump_reg();
36         }
37
38 /*      if (Reg[1] == 0)
39 /*      {                       /* SYS_indir */
40 /*              syscallno = Reg[8];     /* out reg 0 */
41 /*              o0 = Reg[9];
42 /*              o1 = Reg[10];
43 /*              o2 = Reg[11];
44 /*      }
45 /*      else    /* */
46         {
47                 syscallno = Reg[2];
48                 o0 = Reg[4];
49                 o1 = Reg[5];
50                 o2 = Reg[6];
51         }
52
53         switch (syscallno)
54         {
55                 case SYS_exit:  /*1*/
56                         printstatistics();
57                         fflush(stdout);
58                         exit(0);
59                         break;
60                 case SYS_read:  /*3*/
61                         Reg[1] =
62                                 read(u_to_int_fd(o0), u_to_int_addr(o1), o2);
```

```
63                      break;
64              case SYS_write: /*4*/
65                      Reg[1] =
66                              write(u_to_int_fd(o0), u_to_int_addr(o1), o2);
67                       break;
68
69              case SYS_open:  /*5*/
70                      Reg[1] = open(u_to_int_addr(o0), o1, o2); /* */
71                      break;
72
73              case SYS_close: /*6*/
74                      Reg[1] = 0;     /* hack */
75                      break;
76
77              case 17:        /* 17 */
78                      /* old sbreak. where did it go? */
79                      Reg[1] = ((o0 / 8192) + 1) * 8192;
80                      break;
81
82              case SYS_lseek: /*19*/
83                      Reg[1] = (int) lseek(u_to_int_fd(o0), (long) o1, o2);
84                      break;
85
86              case SYS_ioctl:/* 54 */
87                      {       /* copied from sas -- I don't understand yet. */
88                              /* see dave weaver */
89 #define IOCPARM_MASK    0x7f    /* parameters must be < 128 bytes */
90                              int size = (o1 >> 16) & IOCPARM_MASK;
91                              char ioctl_group = (o1 >> 8) & 0x00ff;
92                              if ((ioctl_group == 't') && (size == 8))
93                              {
94                                      size = 6;
95                                      o1 = (o1 & ~((IOCPARM_MASK << 16)))
96                                         | (size << 16);
97                              }
98                      }
99                      Reg[1] = ioctl(u_to_int_fd(o0),o1,u_to_int_addr(o2));
100                     Reg[1] = 0;     /* hack */
101                     break;
102
103             case SYS_fstat:         /* 62 */
104                     Reg[1] = fstat(o1, o2);
105                     break;
106
107             case SYS_getpagesize:   /* 64 */
108                     Reg[1] = getpagesize();
109                     break;
110
111             default:
112                     printf("Unknown System call %d\n", syscallno);
113                     if  ( ! Traptrace )
114                             dump_reg();
115                     exit(2);
116                     break;
117         }
118         if  ( Traptrace )
119         {
120                 printf("**Afterwards:\n");
```

```
121                 dump_reg();
122         }
123 }
124
125 char *u_to_int_addr(ptr)
126 int ptr;
127 {       /* convert a user pointer to the real address   */
128         /* used in the interpreter                      */
129
130          return ((char *) ((int) mem - memoffset  + ptr));
131 }
132
133 u_to_int_fd(fd)
134 {
135         if (fd > 2)
136         {
137                 /*
138                 printf("No general file descriptors yet\n");
139                 exit(2);
140                 */
141         }
142         return (fd);            /* assume we can handle it for now */
143 }
```

Chapter 6

Directory ../test/

Contents

6.1 halt-a.s 207
6.2 halt.c 208
6.3 matmult.c 208
6.4 shell.c 209
6.5 sort.c 209
6.6 start.s 210

This chapter lists all the source codes found in directory `../test/`. They are:

```
halt-a.s    matmult.c   sort.c
halt.c      shell.c     start.s
```

6.1 halt-a.s

```
 1          .file   1 "halt.c"
 2
 3  # GNU C 2.6.3 [AL 1.1, MM 40] DECstation running ultrix compiled by GNU C
 4
 5  # Cc1 defaults:
 6
 7  # Cc1 arguments (-G value = 8, Cpu = 3000, ISA = 1):
 8  # -quiet -dumpbase -o
 9
10 gcc2_compiled.:
11 __gnu_compiled_c:
12         .text
13         .align  2
14         .globl  main
15         .ent    main
16 main:
17         .frame  $fp,24,$31              # vars= 0, regs= 2/0, args= 16, extra= 0
18         .mask   0xc0000000,-4
19         .fmask  0x00000000,0
20         subu    $sp,$sp,24
21         sw      $31,20($sp)
22         sw      $fp,16($sp)
23         move    $fp,$sp
24         jal     __main
```

```
25          jal     Halt
26  $L1:
27          move    $sp,$fp                 # sp not trusted here
28          lw      $31,20($sp)
29          lw      $fp,16($sp)
30          addu    $sp,$sp,24
31          j       $31
32          .end    main
```

6.2 halt.c

```
 1  /* halt.c
 2   *      Simple program to test whether running a user program works.
 3   *
 4   *      Just do a "syscall" that shuts down the OS.
 5   *
 6   *      NOTE: for some reason, user programs with global data structures
 7   *      sometimes haven't worked in the Nachos environment.  So be careful
 8   *      out there!  One option is to allocate data structures as
 9   *      automatics within a procedure, but if you do this, you have to
10   *      be careful to allocate a big enough stack to hold the automatics!
11   */
12
13  #include "syscall.h"
14
15  int
16  main()
17  {
18      Halt();
19      /* not reached */
20  }
```

6.3 matmult.c

```
 1  /* matmult.c
 2   *    Test program to do matrix multiplication on large arrays.
 3   *
 4   *    Intended to stress virtual memory system.
 5   *
 6   *    Ideally, we could read the matrices off of the file system,
 7   *      and store the result back to the file system!
 8   */
 9
10  #include "syscall.h"
11
12  #define Dim     20      /* sum total of the arrays doesn't fit in
13                           * physical memory
14                           */
15
16  int A[Dim][Dim];
17  int B[Dim][Dim];
18  int C[Dim][Dim];
19
20  int
21  main()
22  {
23      int i, j, k;
```

```
24
25      for (i = 0; i < Dim; i++)           /* first initialize the matrices */
26          for (j = 0; j < Dim; j++) {
27               A[i][j] = i;
28               B[i][j] = j;
29               C[i][j] = 0;
30          }
31
32      for (i = 0; i < Dim; i++)           /* then multiply them together */
33          for (j = 0; j < Dim; j++)
34              for (k = 0; k < Dim; k++)
35                   C[i][j] += A[i][k] * B[k][j];
36
37      Exit(C[Dim-1][Dim-1]);              /* and then we're done */
38  }
```

6.4 shell.c

```
 1  #include "syscall.h"
 2
 3  int
 4  main()
 5  {
 6      SpaceId newProc;
 7      OpenFileId input = ConsoleInput;
 8      OpenFileId output = ConsoleOutput;
 9      char prompt[2], ch, buffer[60];
10      int i;
11
12      prompt[0] = '-';
13      prompt[1] = '-';
14
15      while( 1 )
16      {
17          Write(prompt, 2, output);
18
19          i = 0;
20
21          do {
22
23              Read(&buffer[i], 1, input);
24
25          } while( buffer[i++] != '\n' );
26
27          buffer[--i] = '\0';
28
29          if( i > 0 ) {
30                  newProc = Exec(buffer);
31                  Join(newProc);
32          }
33      }
34  }
35
```

6.5 sort.c

```
1  /* sort.c
```

```
 2  *    Test program to sort a large number of integers.
 3  *
 4  *    Intention is to stress virtual memory system.
 5  *
 6  *    Ideally, we could read the unsorted array off of the file system,
 7  *      and store the result back to the file system!
 8  */
 9
10 #include "syscall.h"
11
12 /* size of physical memory; with code, we'll run out of space!*/
13 #define ARRAYSIZE 1024
14
15 int A[ARRAYSIZE];
16
17 int
18 main()
19 {
20     int i, j, tmp;
21
22     /* first initialize the array, in reverse sorted order */
23     for (i = 0; i < ARRAYSIZE; i++)
24         A[i] = ARRAYSIZE - i - 1;
25
26     /* then sort! */
27     for (i = 0; i < (ARRAYSIZE - 1); i++)
28         for (j = 0; j < ((ARRAYSIZE - 1) - i); j++)
29            if (A[j] > A[j + 1]) {       /* out of order -> need to swap ! */
30               tmp = A[j];
31               A[j] = A[j + 1];
32               A[j + 1] = tmp;
33            }
34     Exit(A[0]);         /* and then we're done -- should be 0! */
35 }
```

6.6 start.s

```
 1 /* Start.s
 2  *      Assembly language assist for user programs running on top of Nachos.
 3  *
 4  *      Since we don't want to pull in the entire C library, we define
 5  *      what we need for a user program here, namely Start and the system
 6  *      calls.
 7  */
 8
 9 #define IN_ASM
10 #include "syscall.h"
11
12         .text
13         .align  2
14
15 /* -------------------------------------------------------------
16  * __start
17  *      Initialize running a C program, by calling "main".
18  *
19  *      NOTE: This has to be first, so that it gets loaded at location 0.
20  *      The Nachos kernel always starts a program by jumping to location 0.
```

```
 * ------------------------------------------------------------
 */

        .globl __start
        .ent    __start
__start:
        jal     main
        move    $4,$0
        jal     Exit     /* if we return from main, exit(0) */
        .end __start

/* ------------------------------------------------------------
 * System call stubs:
 *      Assembly language assist to make system calls to the Nachos kernel.
 *      There is one stub per system call, that places the code for the
 *      system call into register r2, and leaves the arguments to the
 *      system call alone (in other words, arg1 is in r4, arg2 is
 *      in r5, arg3 is in r6, arg4 is in r7)
 *
 *      The return value is in r2. This follows the standard C calling
 *      convention on the MIPS.
 * ------------------------------------------------------------
 */

        .globl Halt
        .ent    Halt
Halt:
        addiu $2,$0,SC_Halt
        syscall
        j       $31
        .end Halt

        .globl Exit
        .ent    Exit
Exit:
        addiu $2,$0,SC_Exit
        syscall
        j       $31
        .end Exit

        .globl Exec
        .ent    Exec
Exec:
        addiu $2,$0,SC_Exec
        syscall
        j       $31
         .end Exec

        .globl Join
        .ent    Join
Join:
        addiu $2,$0,SC_Join
        syscall
        j       $31
        .end Join

        .globl Create
        .ent    Create
```

```
 79 Create:
 80         addiu $2,$0,SC_Create
 81         syscall
 82         j       $31
 83         .end Create
 84
 85         .globl Open
 86         .ent    Open
 87 Open:
 88         addiu $2,$0,SC_Open
 89         syscall
 90         j       $31
 91         .end Open
 92
 93         .globl Read
 94         .ent    Read
 95 Read:
 96         addiu $2,$0,SC_Read
 97         syscall
 98         j       $31
 99         .end Read
100
101         .globl Write
102         .ent    Write
103 Write:
104         addiu $2,$0,SC_Write
105         syscall
106         j       $31
107         .end Write
108
109         .globl Close
110         .ent    Close
111 Close:
112         addiu $2,$0,SC_Close
113         syscall
114         j       $31
115         .end Close
116
117         .globl Fork
118         .ent    Fork
119 Fork:
120         addiu $2,$0,SC_Fork
121         syscall
122         j       $31
123         .end Fork
124
125         .globl Yield
126         .ent    Yield
127 Yield:
128         addiu $2,$0,SC_Yield
129         syscall
130         j       $31
131         .end Yield
132
133 /* dummy function to keep gcc happy */
134         .globl  __main
135         .ent    __main
136 __main:
```

```
137         j       $31
138         .end    __main
139
```

Chapter 7

Directory ../network/

Contents

7.1 nettest.cc 215
7.2 post.cc 216
7.3 post.h 222

This chapter lists all the source codes found in directory `../network/`. They are:

```
nettest.cc   post.cc      post.h
```

7.1 nettest.cc

```
  1 // nettest.cc
  2 //      Test out message delivery between two "Nachos" machines,
  3 //      using the Post Office to coordinate delivery.
  4 //
  5 //      One caveats:
  6 //        1. Two copies of Nachos must be running, with machine ID's 0 and 1:
  7 //               ./nachos -m 0 -o 1 &
  8 //               ./nachos -m 1 -o 0 &
  9 //
 10 // Copyright (c) 1992-1993 The Regents of the University of California.
 11 // All rights reserved.  See copyright.h for copyright notice and limitation
 12 // of liability and disclaimer of warranty provisions.
 13
 14 #include "copyright.h"
 15
 16 #include "system.h"
 17 #include "network.h"
 18 #include "post.h"
 19 #include "interrupt.h"
 20
 21 // Test out message delivery, by doing the following:
 22 //      1. send a message to the machine with ID "farAddr", at mail box #0
 23 //      2. wait for the other machine's message to arrive (in our mailbox #0)
 24 //      3. send an acknowledgment for the other machine's message
 25 //      4. wait for an acknowledgement from the other machine to our
 26 //          original message
 27
 28 void
 29 MailTest(int farAddr)
```

```
30 {
31     PacketHeader outPktHdr, inPktHdr;
32     MailHeader outMailHdr, inMailHdr;
33     char *data = "Hello there!";
34     char *ack = "Got it!";
35     char buffer[MaxMailSize];
36
37     // construct packet, mail header for original message
38     // To: destination machine, mailbox 0
39     // From: our machine, reply to: mailbox 1
40     outPktHdr.to = farAddr;
41     outMailHdr.to = 0;
42     outMailHdr.from = 1;
43     outMailHdr.length = strlen(data) + 1;
44
45     // Send the first message
46     postOffice->Send(outPktHdr, outMailHdr, data);
47
48     // Wait for the first message from the other machine
49     postOffice->Receive(0, &inPktHdr, &inMailHdr, buffer);
50     printf("Got \"%s\" from %d, box %d\n",buffer,inPktHdr.from,inMailHdr.from);
51     fflush(stdout);
52
53     // Send acknowledgement to the other machine (using "reply to" mailbox
54     // in the message that just arrived
55     outPktHdr.to = inPktHdr.from;
56     outMailHdr.to = inMailHdr.from;
57     outMailHdr.length = strlen(ack) + 1;
58     postOffice->Send(outPktHdr, outMailHdr, ack);
59
60     // Wait for the ack from the other machine to the first message we sent.
61     postOffice->Receive(1, &inPktHdr, &inMailHdr, buffer);
62     printf("Got \"%s\" from %d, box %d\n",buffer,inPktHdr.from,inMailHdr.from);
63     fflush(stdout);
64
65     // Then we're done!
66     interrupt->Halt();
67 }
```

7.2 post.cc

```
 1 // post.cc
 2 //      Routines to deliver incoming network messages to the correct
 3 //      "address" -- a mailbox, or a holding area for incoming messages.
 4 //      This module operates just like the US postal service (in other
 5 //      words, it works, but it's slow, and you can't really be sure if
 6 //      your mail really got through!).
 7 //
 8 //      Note that once we prepend the MailHdr to the outgoing message data,
 9 //      the combination (MailHdr plus data) looks like "data" to the Network
10 //      device.
11 //
12 //      The implementation synchronizes incoming messages with threads
13 //      waiting for those messages.
14 //
15 // Copyright (c) 1992-1993 The Regents of the University of California.
16 // All rights reserved.  See copyright.h for copyright notice and limitation
```

```
17 // of liability and disclaimer of warranty provisions.
18
19 #include "copyright.h"
20 #include "post.h"
21
22 //----------------------------------------------------------------------
23 // Mail::Mail
24 //      Initialize a single mail message, by concatenating the headers to
25 //      the data.
26 //
27 //      "pktH" -- source, destination machine ID's
28 //      "mailH" -- source, destination mailbox ID's
29 //      "data" -- payload data
30 //----------------------------------------------------------------------
31
32 Mail::Mail(PacketHeader pktH, MailHeader mailH, char *msgData)
33 {
34     ASSERT(mailH.length <= MaxMailSize);
35
36     pktHdr = pktH;
37     mailHdr = mailH;
38     bcopy(msgData, data, mailHdr.length);
39 }
40
41 //----------------------------------------------------------------------
42 // MailBox::MailBox
43 //      Initialize a single mail box within the post office, so that it
44 //      can receive incoming messages.
45 //
46 //      Just initialize a list of messages, representing the mailbox.
47 //----------------------------------------------------------------------
48
49
50 MailBox::MailBox()
51 {
52     messages = new SynchList();
53 }
54
55 //----------------------------------------------------------------------
56 // MailBox::~MailBox
57 //      De-allocate a single mail box within the post office.
58 //
59 //      Just delete the mailbox, and throw away all the queued messages
60 //      in the mailbox.
61 //----------------------------------------------------------------------
62
63 MailBox::~MailBox()
64 {
65     delete messages;
66 }
67
68 //----------------------------------------------------------------------
69 // PrintHeader
70 //      Print the message header -- the destination machine ID and mailbox
71 //      #, source machine ID and mailbox #, and message length.
72 //
73 //      "pktHdr" -- source, destination machine ID's
74 //      "mailHdr" -- source, destination mailbox ID's
```

```
75 //----------------------------------------------------------------------
76
77 static void
78 PrintHeader(PacketHeader pktHdr, MailHeader mailHdr)
79 {
80     printf("From (%d, %d) to (%d, %d) bytes %d\n",
81             pktHdr.from, mailHdr.from, pktHdr.to, mailHdr.to, mailHdr.length);
82 }
83
84 //----------------------------------------------------------------------
85 // MailBox::Put
86 //      Add a message to the mailbox.  If anyone is waiting for message
87 //      arrival, wake them up!
88 //
89 //      We need to reconstruct the Mail message (by concatenating the headers
90 //      to the data), to simplify queueing the message on the SynchList.
91 //
92 //      "pktHdr" -- source, destination machine ID's
93 //      "mailHdr" -- source, destination mailbox ID's
94 //      "data" -- payload message data
95 //----------------------------------------------------------------------
96
97 void
98 MailBox::Put(PacketHeader pktHdr, MailHeader mailHdr, char *data)
99 {
100     Mail *mail = new Mail(pktHdr, mailHdr, data);
101
102     messages->Append((void *)mail);     // put on the end of the list of
103                                         // arrived messages, and wake up
104                                         // any waiters
105 }
106
107 //----------------------------------------------------------------------
108 // MailBox::Get
109 //      Get a message from a mailbox, parsing it into the packet header,
110 //      mailbox header, and data.
111 //
112 //      The calling thread waits if there are no messages in the mailbox.
113 //
114 //      "pktHdr" -- address to put: source, destination machine ID's
115 //      "mailHdr" -- address to put: source, destination mailbox ID's
116 //      "data" -- address to put: payload message data
117 //----------------------------------------------------------------------
118
119 void
120 MailBox::Get(PacketHeader *pktHdr, MailHeader *mailHdr, char *data)
121 {
122     DEBUG('n', "Waiting for mail in mailbox\n");
123     Mail *mail = (Mail *) messages->Remove();   // remove message from list;
124                                                 // will wait if list is empty
125
126     *pktHdr = mail->pktHdr;
127     *mailHdr = mail->mailHdr;
128     if (DebugIsEnabled('n')) {
129         printf("Got mail from mailbox: ");
130         PrintHeader(*pktHdr, *mailHdr);
131     }
132     bcopy(mail->data, data, mail->mailHdr.length);
```

```
133                                         // copy the message data into
134                                         // the caller's buffer
135     delete mail;                        // we've copied out the stuff we
136                                         // need, we can now discard the message
137 }
138
139 //----------------------------------------------------------------------
140 // PostalHelper, ReadAvail, WriteDone
141 //      Dummy functions because C++ can't indirectly invoke member functions
142 //      The first is forked as part of the "postal worker thread; the
143 //      later two are called by the network interrupt handler.
144 //
145 //      "arg" -- pointer to the Post Office managing the Network
146 //----------------------------------------------------------------------
147
148 static void PostalHelper(_int arg)
149 { PostOffice* po = (PostOffice *) arg; po->PostalDelivery(); }
150 static void ReadAvail(_int arg)
151 { PostOffice* po = (PostOffice *) arg; po->IncomingPacket(); }
152 static void WriteDone(_int arg)
153 { PostOffice* po = (PostOffice *) arg; po->PacketSent(); }
154
155 //----------------------------------------------------------------------
156 // PostOffice::PostOffice
157 //      Initialize a post office as a collection of mailboxes.
158 //      Also initialize the network device, to allow post offices
159 //      on different machines to deliver messages to one another.
160 //
161 //      We use a separate thread "the postal worker" to wait for messages
162 //      to arrive, and deliver them to the correct mailbox.  Note that
163 //      delivering messages to the mailboxes can't be done directly
164 //      by the interrupt handlers, because it requires a Lock.
165 //
166 //      "addr" is this machine's network ID
167 //      "reliability" is the probability that a network packet will
168 //        be delivered (e.g., reliability = 1 means the network never
169 //        drops any packets; reliability = 0 means the network never
170 //        delivers any packets)
171 //      "orderability" is the probability that a network packet that
172 //        is delivered is delivered without delay (e.g., orderability = 1
173 //        means that delivered packets are never delayed)
174 //      "nBoxes" is the number of mail boxes in this Post Office
175 //----------------------------------------------------------------------
176
177 PostOffice::PostOffice(NetworkAddress addr, double reliability,
178                        double orderability, int nBoxes)
179 {
180 // First, initialize the synchronization with the interrupt handlers
181     messageAvailable = new Semaphore("message available", 0);
182     messageSent = new Semaphore("message sent", 0);
183     sendLock = new Lock("message send lock");
184
185 // Second, initialize the mailboxes
186     netAddr = addr;
187     numBoxes = nBoxes;
188     boxes = new MailBox[nBoxes];
189
190 // Third, initialize the network; tell it which interrupt handlers to call
```

```
191     network = new Network(addr, reliability, orderability,
192                           ReadAvail, WriteDone, (_int) this);
193
194
195 // Finally, create a thread whose sole job is to wait for incoming messages,
196 //   and put them in the right mailbox.
197     Thread *t = new Thread("postal worker");
198
199     t->Fork(PostalHelper, (_int) this);
200 }
201
202 //----------------------------------------------------------------------
203 // PostOffice::~PostOffice
204 //      De-allocate the post office data structures.
205 //----------------------------------------------------------------------
206
207 PostOffice::~PostOffice()
208 {
209     delete network;
210     delete [] boxes;
211     delete messageAvailable;
212     delete messageSent;
213     delete sendLock;
214 }
215
216 //----------------------------------------------------------------------
217 // PostOffice::PostalDelivery
218 //      Wait for incoming messages, and put them in the right mailbox.
219 //
220 //      Incoming messages have had the PacketHeader stripped off,
221 //      but the MailHeader is still tacked on the front of the data.
222 //----------------------------------------------------------------------
223
224 void
225 PostOffice::PostalDelivery()
226 {
227     PacketHeader pktHdr;
228     MailHeader mailHdr;
229     char *buffer = new char[MaxPacketSize];
230
231     for (;;) {
232         // first, wait for a message
233         messageAvailable->P();
234         pktHdr = network->Receive(buffer);
235
236         mailHdr = *(MailHeader *)buffer;
237         if (DebugIsEnabled('n')) {
238             printf("Putting mail into mailbox: ");
239             PrintHeader(pktHdr, mailHdr);
240         }
241
242         // check that arriving message is legal!
243         ASSERT(0 <= mailHdr.to && mailHdr.to < numBoxes);
244         ASSERT(mailHdr.length <= MaxMailSize);
245
246         // put into mailbox
247         boxes[mailHdr.to].Put(pktHdr, mailHdr, buffer + sizeof(MailHeader));
248     }
```

```
249 }
250
251 //----------------------------------------------------------------------
252 // PostOffice::Send
253 //      Concatenate the MailHeader to the front of the data, and pass
254 //      the result to the Network for delivery to the destination machine.
255 //
256 //      Note that the MailHeader + data looks just like normal payload
257 //      data to the Network.
258 //
259 //      "pktHdr" -- source, destination machine ID's
260 //      "mailHdr" -- source, destination mailbox ID's
261 //      "data" -- payload message data
262 //----------------------------------------------------------------------
263
264 void
265 PostOffice::Send(PacketHeader pktHdr, MailHeader mailHdr, char* data)
266 {
267     char* buffer = new char[MaxPacketSize];     // space to hold concatenated
268                                                 // mailHdr + data
269
270     if (DebugIsEnabled('n')) {
271         printf("Post send: ");
272         PrintHeader(pktHdr, mailHdr);
273     }
274     ASSERT(mailHdr.length <= MaxMailSize);
275     ASSERT(0 <= mailHdr.to && mailHdr.to < numBoxes);
276
277     // fill in pktHdr, for the Network layer
278     pktHdr.from = netAddr;
279     pktHdr.length = mailHdr.length + sizeof(MailHeader);
280
281     // concatenate MailHeader and data
282 #ifdef HOST_ALPHA
283     bcopy((const char *)&mailHdr, buffer, sizeof(MailHeader));
284 #else
285     bcopy(&mailHdr, buffer, sizeof(MailHeader));
286 #endif
287     bcopy(data, buffer + sizeof(MailHeader), mailHdr.length);
288
289     sendLock->Acquire();                // only one message can be sent
290                                         // to the network at any one time
291     network->Send(pktHdr, buffer);
292     messageSent->P();                   // wait for interrupt to tell us
293                                         // ok to send the next message
294     sendLock->Release();
295
296     delete [] buffer;                   // we've sent the message, so
297                                         // we can delete our buffer
298 }
299
300 //----------------------------------------------------------------------
301 // PostOffice::Send
302 //      Retrieve a message from a specific box if one is available,
303 //      otherwise wait for a message to arrive in the box.
304 //
305 //      Note that the MailHeader + data looks just like normal payload
306 //      data to the Network.
```

```
307 //
308 //
309 //      "box" -- mailbox ID in which to look for message
310 //      "pktHdr" -- address to put: source, destination machine ID's
311 //      "mailHdr" -- address to put: source, destination mailbox ID's
312 //      "data" -- address to put: payload message data
313 //----------------------------------------------------------------------
314
315 void
316 PostOffice::Receive(int box, PacketHeader *pktHdr,
317                                 MailHeader *mailHdr, char* data)
318 {
319     ASSERT((box >= 0) && (box < numBoxes));
320
321     boxes[box].Get(pktHdr, mailHdr, data);
322     ASSERT(mailHdr->length <= MaxMailSize);
323 }
324
325 //----------------------------------------------------------------------
326 // PostOffice::IncomingPacket
327 //      Interrupt handler, called when a packet arrives from the network.
328 //
329 //      Signal the PostalDelivery routine that it is time to get to work!
330 //----------------------------------------------------------------------
331
332 void
333 PostOffice::IncomingPacket()
334 {
335     messageAvailable->V();
336 }
337
338 //----------------------------------------------------------------------
339 // PostOffice::PacketSent
340 //      Interrupt handler, called when the next packet can be put onto the
341 //      network.
342 //
343 //      The name of this routine is a misnomer; if "reliability < 1",
344 //      the packet could have been dropped by the network, so it won't get
345 //      through.
346 //----------------------------------------------------------------------
347
348 void
349 PostOffice::PacketSent()
350 {
351     messageSent->V();
352 }
353
```

7.3 post.h

```
1 // post.h
2 //      Data structures for providing the abstraction of unreliable,
3 //      unordered, fixed-size message delivery to mailboxes on other
4 //      (directly connected) machines.  Messages can be dropped or delayed by
5 //      the network, but they are never corrupted.
6 //
7 //      The US Post Office delivers mail to the addressed mailbox.
```

```
 8 //      By analogy, our post office delivers packets to a specific buffer
 9 //      (MailBox), based on the mailbox number stored in the packet header.
10 //      Mail waits in the box until a thread asks for it; if the mailbox
11 //      is empty, threads can wait for mail to arrive in it.
12 //
13 //      Thus, the service our post office provides is to de-multiplex
14 //      incoming packets, delivering them to the appropriate thread.
15 //
16 //      With each message, you get a return address, which consists of a "from
17 //      address", which is the id of the machine that sent the message, and
18 //      a "from box", which is the number of a mailbox on the sending machine
19 //      to which you can send an acknowledgement, if your protocol requires
20 //      this.
21 //
22 // Copyright (c) 1992-1993 The Regents of the University of California.
23 // All rights reserved.  See copyright.h for copyright notice and limitation
24 // of liability and disclaimer of warranty provisions.
25
26 #include "copyright.h"
27
28 #ifndef POST_H
29 #define POST_H
30
31 #include "network.h"
32 #include "synchlist.h"
33
34 // Mailbox address -- uniquely identifies a mailbox on a given machine.
35 // A mailbox is just a place for temporary storage for messages.
36 typedef int MailBoxAddress;
37
38 // The following class defines part of the message header.
39 // This is prepended to the message by the PostOffice, before the message
40 // is sent to the Network.
41
42 class MailHeader {
43   public:
44     MailBoxAddress to;          // Destination mail box
45     MailBoxAddress from;        // Mail box to reply to
46     unsigned length;            // Bytes of message data (excluding the
47                                 // mail header)
48 };
49
50 // Maximum "payload" -- real data -- that can included in a single message
51 // Excluding the MailHeader and the PacketHeader
52
53 #define MaxMailSize     (MaxPacketSize - sizeof(MailHeader))
54
55
56 // The following class defines the format of an incoming/outgoing
57 // "Mail" message.  The message format is layered:
58 //      network header (PacketHeader)
59 //      post office header (MailHeader)
60 //      data
61
62 class Mail {
63   public:
64      Mail(PacketHeader pktH, MailHeader mailH, char *msgData);
65                                 // Initialize a mail message by
```

```
66                                // concatenating the headers to the data
67
68      PacketHeader pktHdr;      // Header appended by Network
69      MailHeader mailHdr;       // Header appended by PostOffice
70      char data[MaxMailSize];   // Payload -- message data
71 };
72
73 // The following class defines a single mailbox, or temporary storage
74 // for messages.   Incoming messages are put by the PostOffice into the
75 // appropriate mailbox, and these messages can then be retrieved by
76 // threads on this machine.
77
78 class MailBox {
79   public:
80     MailBox();                   // Allocate and initialize mail box
81     ~MailBox();                  // De-allocate mail box
82
83     void Put(PacketHeader pktHdr, MailHeader mailHdr, char *data);
84                                  // Atomically put a message into the mailbox
85     void Get(PacketHeader *pktHdr, MailHeader *mailHdr, char *data);
86                                  // Atomically get a message out of the
87                                  // mailbox (and wait if there is no message
88                                  // to get!)
89   private:
90     SynchList *messages;         // A mailbox is just a list of arrived messages
91 };
92
93 // The following class defines a "Post Office", or a collection of
94 // mailboxes.  The Post Office is a synchronization object that provides
95 // two main operations: Send -- send a message to a mailbox on a remote
96 // machine, and Receive -- wait until a message is in the mailbox,
97 // then remove and return it.
98 //
99 // Incoming messages are put by the PostOffice into the
100 // appropriate mailbox, waking up any threads waiting on Receive.
101
102 class PostOffice {
103   public:
104     PostOffice(NetworkAddress addr, double reliability,
105                double orderability, int nBoxes);
106                                  // Allocate and initialize Post Office
107                                  //   "reliability" is how many packets
108                                  //   get dropped by the underlying network
109     ~PostOffice();               // De-allocate Post Office data
110
111     void Send(PacketHeader pktHdr, MailHeader mailHdr, char *data);
112                                  // Send a message to a mailbox on a remote
113                                  // machine.  The fromBox in the MailHeader is
114                                  // the return box for ack's.
115
116     void Receive(int box, PacketHeader *pktHdr,
117                  MailHeader *mailHdr, char *data);
118                                  // Retrieve a message from "box".  Wait if
119                                  // there is no message in the box.
120
121     void PostalDelivery();       // Wait for incoming messages,
122                                  // and then put them in the correct mailbox
123
```

```
124     void PacketSent();          // Interrupt handler, called when outgoing
125                                 // packet has been put on network; next
126                                 // packet can now be sent
127     void IncomingPacket();      // Interrupt handler, called when incoming
128                                 // packet has arrived and can be pulled
129                                 // off of network (i.e., time to call
130                                 // PostalDelivery)
131
132   private:
133     Network *network;            // Physical network connection
134     NetworkAddress netAddr;     // Network address of this machine
135     MailBox *boxes;             // Table of mail boxes to hold incoming mail
136     int numBoxes;               // Number of mail boxes
137     Semaphore *messageAvailable;// V'ed when message has arrived from network
138     Semaphore *messageSent;     // V'ed when next message can be sent to network
139     Lock *sendLock;             // Only one outgoing message at a time
140 };
141
142 #endif
```