```
// list.cc
//
//     Routines to manage a singly-linked list of "things".
//
// A "ListElement" is allocated for each item to be put on the
//list; it is de-allocated when the item is removed. This means
//      we don't need to keep a "next" pointer in every object we
//      want to put on a list.
//     NOTE: Mutual exclusion must be provided by the caller.
//  If you want a synchronized list, you must use the routines
//in synchlist.cc.
#include "copyright.h"
#include "list.h"
//-------------------------------------------------------------------
// ListElement::ListElement
// Initialize a list element, so it can be added somewhere on a list.
//
//"itemPtr" is the item to be put on the list.  It can be a pointer
//to anything.
//"sortKey" is the priority of the item, if any.
//-------------------------------------------------------------------
ListElement::ListElement(void *itemPtr, int sortKey)
{
     item = itemPtr;
     key = sortKey;
     next = NULL;// assume we'll put it at the end of the list
}
//-------------------------------------------------------------------
// List::List
//Initialize a list, empty to start with.
//Elements can now be added to the list.
//-------------------------------------------------------------------
List::List()
{
    first = last = NULL;
}
//-------------------------------------------------------------------
// List::~List
//Prepare a list for deallocation.  If the list still contains any
//ListElements, de-allocate them.  However, note that we do *not*
//de-allocate the "items" on the list -- this module allocates
//and de-allocates the ListElements to keep track of each item,
//but a given item may be on multiple lists, so we can't
//de-allocate them here.
//-------------------------------------------------------------------
List::~List()
{
    while (Remove() != NULL)
; // delete all the list elements
}
//-------------------------------------------------------------------
// List::Append
//      Append an "item" to the end of the list.
//Allocate a ListElement to keep track of the item.
//      If the list is empty, then this will be the only element.
//Otherwise, put it at the end.
//"item" is the thing to put on the list, it can be a pointer to
anything.
//-------------------------------------------------------------------
void
List::Append(void *item)
{
    ListElement *element = new ListElement(item, 0);
    if (IsEmpty()) {// list is empty
first = element;
last = element;
    } else {// else put it after last
last->next = element;
last = element;
    }
}
//-------------------------------------------------------------------
// List::Prepend
//      Put an "item" on the front of the list.
//Allocate a ListElement to keep track of the item.
//      If the list is empty, then this will be the only element.
//Otherwise, put it at the beginning.
//"item" is the thing to put on the list, it can be a pointer to
anything.
//-------------------------------------------------------------------
void
List::Prepend(void *item)
{
    ListElement *element = new ListElement(item, 0);
    if (IsEmpty()) {// list is empty
first = element;
last = element;
    } else {// else put it before first
element->next = first;
first = element;
    }
}
//-------------------------------------------------------------------
// List::Remove
//      Remove the first "item" from the front of the list.
// Returns:
//Pointer to removed item, NULL if nothing on the list.
//-------------------------------------------------------------------
void *
List::Remove()
{
    return SortedRemove(NULL);  // Same as SortedRemove, but ignore
the key
}
//-------------------------------------------------------------------
// List::Mapcar
//Apply a function to each item on the list, by walking through
//the list, one element at a time.
//Unlike LISP, this mapcar does not return anything!
//"func" is the procedure to apply to each element of the list.
//-------------------------------------------------------------------
void
```

```
List::Mapcar(VoidFunctionPtr func)
{
    for (ListElement *ptr = first; ptr != NULL; ptr = ptr->next) {
       DEBUG('l', "In mapcar, about to invoke %x(%x)\n", func, ptr-
>item);
       (*func)((int)ptr->item);
    }
}
//----------------------------------------------------------------------
// List::IsEmpty
//      Returns TRUE if the list is empty (has no items).
//----------------------------------------------------------------------
bool
List::IsEmpty()
{
    if (first == NULL)
        return TRUE;
    else
return FALSE;
}
//----------------------------------------------------------------------
// List::SortedInsert
//      Insert an "item" into a list, so that the list elements are
//sorted in increasing order by "sortKey".
//Allocate a ListElement to keep track of the item.
//      If the list is empty, then this will be the only element.
//Otherwise, walk through the list, one element at a time,
//to find where the new item should be placed.
//"item" is the thing to put on the list, it can be a pointer to
//anything.
//"sortKey" is the priority of the item.
//----------------------------------------------------------------------
void
List::SortedInsert(void *item, int sortKey)
{
    ListElement *element = new ListElement(item, sortKey);
    ListElement *ptr;// keep track
    if (IsEmpty()) {// if list is empty, put
        first = element;
        last = element;
    } else if (sortKey < first->key) {
// item goes on front of list
element->next = first;
first = element;
    } else {// look for first elt in list bigger than item
        for (ptr = first; ptr->next != NULL; ptr = ptr->next) {
            if (sortKey < ptr->next->key) {
element->next = ptr->next;
        ptr->next = element;
return;
    }
}
last->next = element;// item goes at end of list
last = element;
    }
}
```

```
//----------------------------------------------------------------------
// List::SortedRemove
//      Remove the first "item" from the front of a sorted list.
// Returns:
//Pointer to removed item, NULL if nothing on the list.
//Sets *keyPtr to the priority value of the removed item
//(this is needed by interrupt.cc, for instance).
//"keyPtr" is a pointer to the location in which to store the
//priority of the removed item.
//----------------------------------------------------------------------
void *
List::SortedRemove(int *keyPtr)
{
    ListElement *element = first;
    void *thing;
    if (IsEmpty())
return NULL;
    thing = first->item;
    if (first == last) {// list had one item, now has none
        first = NULL;
last = NULL;
    } else {
        first = element->next;
    }
    if (keyPtr != NULL)
        *keyPtr = element->key;
    delete element;
    return thing;
}
```

```
// list.h
//Data structures to manage LISP-like lists.
//
//      As in LISP, a list can contain any type of data structure
//as an item on the list: thread control blocks,
//pending interrupts, etc.  That is why each item is a "void *",
//or in other words, a "pointers to anything".
//
// 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 LIST_H
#define LIST_H
#include "copyright.h"
#include "utility.h"
// The following class defines a "list element" -- which is
// used to keep track of one item on a list.  It is equivalent to a
// LISP cell, with a "car" ("next") pointing to the next element on the 
list,
// and a "cdr" ("item") pointing to the item on the list.
//
// Internal data structures kept public so that List operations can
// access them directly.
class ListElement {
   public:
     ListElement(void *itemPtr, int sortKey);// initialize a list 
element
     ListElement *next;// next element on list, 
// NULL if this is the last
     int key;    // priority, for a sorted list
     void *item;     // pointer to item on the list
};
// The following class defines a "list" -- a singly linked list of
// list elements, each of which points to a single item on the list.
//
// By using the "Sorted" functions, the list can be kept in sorted
// in increasing order by "key" in ListElement.
class List {
  public:
    List();// initialize the list
    ~List();// de-allocate the list
    void Prepend(void *item); // Put item at the beginning of the list
    void Append(void *item); // Put item at the end of the list
    void *Remove();  // Take item off the front of the list
    void Mapcar(VoidFunctionPtr func);// Apply "func" to every element 
// on the list
    bool IsEmpty();// is the list empty? 

    // Routines to put/get items on/off list in order (sorted by key)
    void SortedInsert(void *item, int sortKey);// Put item into list
    void *SortedRemove(int *keyPtr);   // Remove first item from list
  private:
    ListElement *first;  // Head of the list, NULL if list is empty
    ListElement *last;// Last element of list
};
#endif // LIST_H
```

```
// main.cc
//Bootstrap code to initialize the operating system kernel.
//
//Allows direct calls into internal operating system functions,
//to simplify debugging and testing.  In practice, the
//bootstrap code would just initialize data structures,
//and start a user program to print the login prompt.
//
// Most of this file is not needed until later assignments.
//
// Usage: nachos -d <debugflags> -rs <random seed #>
//-s -x <nachos file> -c <consoleIn> <consoleOut>
//-f -cp <unix file> <nachos file>
//-p <nachos file> -r <nachos file> -l -D -t
//              -n <network reliability> -m <machine id>
//              -o <other machine id>
//              -z
//
//    -d causes certain debugging messages to be printed (cf.
utility.h)
//    -rs causes Yield to occur at random (but repeatable) spots
//    -z prints the copyright message
//
//  USER_PROGRAM
//    -s causes user programs to be executed in single-step mode
//    -x runs a user program
//    -c tests the console
//
//  FILESYS
//    -f causes the physical disk to be formatted
//    -cp copies a file from UNIX to Nachos
//    -p prints a Nachos file to stdout
//    -r removes a Nachos file from the file system
//    -l lists the contents of the Nachos directory
//    -D prints the contents of the entire file system
//    -t tests the performance of the Nachos file system
//
//  NETWORK
//    -n sets the network reliability
//    -m sets this machine's host id (needed for the network)
//    -o runs a simple test of the Nachos network software
//
//  NOTE -- flags are ignored until the relevant assignment.
//  Some of the flags are interpreted here; some in system.cc.
//
// 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.
#define MAIN
#include "copyright.h"
#undef MAIN
#include "utility.h"
#include "system.h"
#ifdef THREADS
extern int testnum;
#endif
// External functions used by this file
extern void ThreadTest(void), Copy(char *unixFile, char *nachosFile);
extern void Print(char *file), PerformanceTest(void);
extern void StartProcess(char *file), ConsoleTest(char *in, char
*out);
extern void MailTest(int networkID);
//-------------------------------------------------------------------
// main
// Bootstrap the operating system kernel.
//
//Check command line arguments
//Initialize data structures
//(optionally) Call test procedure
//
//"argc" is the number of command line arguments (including the name
//of the command) -- ex: "nachos -d +" -> argc = 3
//"argv" is an array of strings, one for each command line argument
//ex: "nachos -d +" -> argv = {"nachos", "-d", "+"}
//-------------------------------------------------------------------
int
main(int argc, char **argv)
{
    int argCount;// the number of arguments
// for a particular command
    DEBUG('t', "Entering main");
    (void) Initialize(argc, argv);

#ifdef THREADS
    for (argc--, argv++; argc > 0; argc -= argCount, argv += argCount)
{
      argCount = 1;
      switch (argv[0][1]) {
      case 'q':
        testnum = atoi(argv[1]);
        argCount++;
        break;
      default:
        testnum = 1;
        break;
      }
    }
    ThreadTest();
#endif
    for (argc--, argv++; argc > 0; argc -= argCount, argv += argCount)
{
argCount = 1;
        if (!strcmp(*argv, "-z"))               // print copyright
            printf (copyright);
#ifdef USER_PROGRAM
        if (!strcmp(*argv, "-x")) {        // run a user program
    ASSERT(argc > 1);
            StartProcess(*(argv + 1));
            argCount = 2;
        } else if (!strcmp(*argv, "-c")) {      // test the console
    if (argc == 1)
```

```
        ConsoleTest(NULL, NULL);
    else {
ASSERT(argc > 2);
        ConsoleTest(*(argv + 1), *(argv + 2));
        argCount = 3;
    }
    interrupt->Halt();// once we start the console, then
// Nachos will loop forever waiting
// for console input
}
#endif // USER_PROGRAM
#ifdef FILESYS
if (!strcmp(*argv, "-cp")) { // copy from UNIX to Nachos
    ASSERT(argc > 2);
    Copy(*(argv + 1), *(argv + 2));
    argCount = 3;
} else if (!strcmp(*argv, "-p")) {// print a Nachos file
    ASSERT(argc > 1);
    Print(*(argv + 1));
    argCount = 2;
} else if (!strcmp(*argv, "-r")) {// remove Nachos file
    ASSERT(argc > 1);
    fileSystem->Remove(*(argv + 1));
    argCount = 2;
} else if (!strcmp(*argv, "-l")) {// list Nachos directory
            fileSystem->List();
} else if (!strcmp(*argv, "-D")) {// print entire filesystem
            fileSystem->Print();
} else if (!strcmp(*argv, "-t")) {// performance test
            PerformanceTest();
}
#endif // FILESYS
#ifdef NETWORK
        if (!strcmp(*argv, "-o")) {
    ASSERT(argc > 1);
            Delay(2); // delay for 2 seconds
// to give the user time to
// start up another nachos
            MailTest(atoi(*(argv + 1)));
            argCount = 2;
        }
#endif // NETWORK
    }
    currentThread->Finish();// NOTE: if the procedure "main"
// returns, then the program "nachos"
// will exit (as any other normal program
// would).  But there may be other
// threads on the ready list.  We switch
// to those threads by saying that the
// "main" thread is finished, preventing
// it from returning.
    return(0);// Not reached...
}
```

```
// scheduler.cc
//Routines to choose the next thread to run, and to dispatch to
//that thread.
//
// These routines assume that interrupts are already disabled.
//If interrupts are disabled, we can assume mutual exclusion
//(since we are on a uniprocessor).
//
// NOTE: We can't use Locks to provide mutual exclusion here, since
// if we needed to wait for a lock, and the lock was busy, we would
//end up calling FindNextToRun(), and that would put us in an
//infinite loop.
//
// Very simple implementation -- no priorities, straight FIFO.
//Might need to be improved in later assignments.
//
// 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.
#include "copyright.h"
#include "scheduler.h"
#include "system.h"
//-------------------------------------------------------------------
// Scheduler::Scheduler
// Initialize the list of ready but not running threads to empty.
//-------------------------------------------------------------------
Scheduler::Scheduler()
{
    readyList = new List;
}
//-------------------------------------------------------------------
// Scheduler::~Scheduler
// De-allocate the list of ready threads.
//-------------------------------------------------------------------
Scheduler::~Scheduler()
{
    delete readyList;
}
//-------------------------------------------------------------------
// Scheduler::ReadyToRun
// Mark a thread as ready, but not running.
//Put it on the ready list, for later scheduling onto the CPU.
//
//"thread" is the thread to be put on the ready list.
//-------------------------------------------------------------------
void
Scheduler::ReadyToRun (Thread *thread)
{
    DEBUG('t', "Putting thread %s on ready list.\n", thread-
>getName());
    thread->setStatus(READY);
    readyList->Append((void *)thread);
}
//-------------------------------------------------------------------
// Scheduler::FindNextToRun
// Return the next thread to be scheduled onto the CPU.
//If there are no ready threads, return NULL.
// Side effect:
//Thread is removed from the ready list.
//-------------------------------------------------------------------
Thread *
Scheduler::FindNextToRun ()
{
    return (Thread *)readyList->Remove();
}
//-------------------------------------------------------------------
// Scheduler::Run
// Dispatch the CPU to nextThread.  Save the state of the old thread,
//and load the state of the new thread, by calling the machine
//dependent context switch routine, SWITCH.
//
//      Note: we assume the state of the previously running thread has
//already been changed from running to blocked or ready (depending).
// Side effect:
//The global variable currentThread becomes nextThread.
//
//"nextThread" is the thread to be put into the CPU.
//-------------------------------------------------------------------
void
Scheduler::Run (Thread *nextThread)
{
    Thread *oldThread = currentThread;

#ifdef USER_PROGRAM// ignore until running user programs
    if (currentThread->space != NULL) {// if this thread is a user
program,
        currentThread->SaveUserState(); // save the user's CPU
registers
currentThread->space->SaveState();
    }
#endif

    oldThread->CheckOverflow();    // check if the old thread
    // had an undetected stack overflow
    currentThread = nextThread;    // switch to the next thread
    currentThread->setStatus(RUNNING);      // nextThread is now
running

    DEBUG('t', "Switching from thread \"%s\" to thread \"%s\"\n",
  oldThread->getName(), nextThread->getName());

    // This is a machine-dependent assembly language routine defined
    // in switch.s.  You may have to think
    // a bit to figure out what happens after this, both from the
point
    // of view of the thread and from the perspective of the "outside
world".
    SWITCH(oldThread, nextThread);

    DEBUG('t', "Now in thread \"%s\"\n", currentThread->getName());
    // If the old thread gave up the processor because it was
finishing,
    // we need to delete its carcass.  Note we cannot delete the thread
    // before now (for example, in Thread::Finish()), because up to
this
    // point, we were still running on the old thread's stack!
    if (threadToBeDestroyed != NULL) {
        delete threadToBeDestroyed;
threadToBeDestroyed = NULL;
    }

#ifdef USER_PROGRAM
    if (currentThread->space != NULL) {// if there is an address space
        currentThread->RestoreUserState();     // to restore, do it.
currentThread->space->RestoreState();
    }
#endif
}
//----------------------------------------------------------------------
// Scheduler::Print
// Print the scheduler state -- in other words, the contents of
//the ready list.  For debugging.
//----------------------------------------------------------------------
void
Scheduler::Print()
{
    printf("Ready list contents:\n");
    readyList->Mapcar((VoidFunctionPtr) ThreadPrint);
}
```

```
// scheduler.h
//Data structures for the thread dispatcher and scheduler.
//Primarily, the list of threads that are ready to run.
//
// 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 SCHEDULER_H
#define SCHEDULER_H
#include "copyright.h"
#include "list.h"
#include "thread.h"
// The following class defines the scheduler/dispatcher abstraction --
// the data structures and operations needed to keep track of which
// thread is running, and which threads are ready but not running.
class Scheduler {
  public:
    Scheduler();// Initialize list of ready threads
    ~Scheduler();// De-allocate ready list
    void ReadyToRun(Thread* thread);// Thread can be dispatched.
    Thread* FindNextToRun();// Dequeue first thread on the ready
// list, if any, and return thread.
    void Run(Thread* nextThread);// Cause nextThread to start running
    void Print();// Print contents of ready list

  private:
    List *readyList;  // queue of threads that are ready to run,
// but not running
};
#endif // SCHEDULER_H
```

```
/* switch.s
 *   Machine dependent context switch routines.  DO NOT MODIFY THESE!
 *
 *Context switching is inherently machine dependent, since
 *the registers to be saved, how to set up an initial
 *call frame, etc, are all specific to a processor architecture.
 *
 * This file currently supports the following architectures:
 *    DEC MIPS
 *    SUN SPARC
 *    HP PA-RISC
 *    Intel 386
 *
 * We define two routines for each architecture:
 *
 * ThreadRoot(InitialPC, InitialArg, WhenDonePC, StartupPC)
 *InitialPC  - The program counter of the procedure to run
 *in this thread.
 *      InitialArg - The single argument to the thread.
 *WhenDonePC - The routine to call when the thread returns.
 *StartupPC  - Routine to call when the thread is started.
 *
 *ThreadRoot is called from the SWITCH() routine to start
 *a thread for the first time.
 *
 * SWITCH(oldThread, newThread)
 * oldThread  - The current thread that was running, where the
 *CPU register state is to be saved.
 * newThread  - The new thread to be run, where the CPU register
 *state is to be loaded from.
 */
#include "copyright.h"
#include "switch.h"
#ifdef HOST_MIPS
/* Symbolic register names */
#define z       $0      /* zero register */
#define a0      $4      /* argument registers */
#define a1      $5
#define s0      $16     /* callee saved */
#define s1      $17
#define s2      $18
#define s3      $19
#define s4      $20
#define s5      $21
#define s6      $22
#define s7      $23
#define sp      $29     /* stack pointer */
#define fp      $30     /* frame pointer */
#define ra      $31     /* return address */
        .text
        .align  2
.globl ThreadRoot
.entThreadRoot,0
ThreadRoot:
orfp,z,z# Clearing the frame pointer here
# makes gdb backtraces of thread stacks
# end here (I hope!)
jalStartupPC# call startup procedure
movea0, InitialArg
jalInitialPC# call main procedure
jal WhenDonePC# when were done, call clean up procedure
# NEVER REACHED
.end ThreadRoot
# a0 -- pointer to old Thread
# a1 -- pointer to new Thread
.globl SWITCH
.entSWITCH,0
SWITCH:
swsp, SP(a0)# save new stack pointer
sws0, S0(a0)# save all the callee-save registers
sws1, S1(a0)
sws2, S2(a0)
sws3, S3(a0)
sws4, S4(a0)
sws5, S5(a0)
sws6, S6(a0)
sws7, S7(a0)
swfp, FP(a0)# save frame pointer
swra, PC(a0)# save return address
lwsp, SP(a1)# load the new stack pointer
lws0, S0(a1)# load the callee-save registers
lws1, S1(a1)
lws2, S2(a1)
lws3, S3(a1)
lws4, S4(a1)
lws5, S5(a1)
lws6, S6(a1)
lws7, S7(a1)
lwfp, FP(a1)
lwra, PC(a1)# load the return address
jra
.end SWITCH
#endif // HOST_MIPS
#ifdef HOST_SPARC
/* NOTE!   deleted ..................
 * These files appear not to exist on Solaris --
 *  you need to find where (the SPARC-specific) MINFRAME,
ST_FLUSH_WINDOWS, ...
 *  are defined.  (I don't have a Solaris machine, so I have no way to
tell.)
 */
/* From sys/trap.h and sys/asm_linkage.h */
#endif // HOST_SPARC
#ifdef HOST_SNAKE
/* NOTE!   deleted ..................
 * These files appear not to exist on Solaris --
 *  you need to find where (the SPARC-specific) MINFRAME,
ST_FLUSH_WINDOWS, ...
 *  are defined.  (I don't have a Solaris machine, so I have no way to
tell.)
 */
/* From sys/trap.h and sys/asm_linkage.h */
```

```
#endif
#ifdef HOST_i386
        .text
        .align  2
        .globl  ThreadRoot
/* void ThreadRoot( void )
**
** expects the following registers to be initialized:
**      eax     points to startup function (interrupt enable)
**      edx     contains inital argument to thread function
**      esi     points to thread function
**      edi     point to Thread::Finish()
*/
ThreadRoot:
        pushl   %ebp
        movl    %esp,%ebp
        pushl   InitialArg
        call    *StartupPC
        call    *InitialPC
        call    *WhenDonePC
        // NOT REACHED
        movl    %ebp,%esp
        popl    %ebp
        ret
/* void SWITCH( thread *t1, thread *t2 )
**
** on entry, stack looks like this:
**      8(esp)  ->              thread *t2
**      4(esp)  ->              thread *t1
**       (esp)  ->              return address
**
** we push the current eax on the stack so that we can use it as
** a pointer to t1, this decrements esp by 4, so when we use it
** to reference stuff on the stack, we add 4 to the offset.
*/
        .comm   _eax_save,4
        .globl  SWITCH
SWITCH:
        movl    %eax,_eax_save          # save the value of eax
        movl    4(%esp),%eax            # move pointer to t1 into eax
        movl    %ebx,_EBX(%eax)         # save registers
        movl    %ecx,_ECX(%eax)
        movl    %edx,_EDX(%eax)
        movl    %esi,_ESI(%eax)
        movl    %edi,_EDI(%eax)
        movl    %ebp,_EBP(%eax)
        movl    %esp,_ESP(%eax)         # save stack pointer
        movl    _eax_save,%ebx          # get the saved value of eax
        movl    %ebx,_EAX(%eax)         # store it
        movl    0(%esp),%ebx            # get return address from stack
into ebx
        movl    %ebx,_PC(%eax)          # save it into the pc storage
        movl    8(%esp),%eax            # move pointer to t2 into eax
        movl    _EAX(%eax),%ebx         # get new value for eax into
ebx
        movl    %ebx,_eax_save          # save it
        movl    _EBX(%eax),%ebx         # retore old registers
        movl    _ECX(%eax),%ecx
        movl    _EDX(%eax),%edx
        movl    _ESI(%eax),%esi
        movl    _EDI(%eax),%edi
        movl    _EBP(%eax),%ebp
        movl    _ESP(%eax),%esp         # restore stack pointer
        movl    _PC(%eax),%eax          # restore return address into
eax
        movl    %eax,4(%esp)            # copy over the ret address on
the stack
        movl    _eax_save,%eax
        ret
#endif
```

```
/* switch.h
 *Definitions needed for implementing context switching.
 *
 *Context switching is inherently machine dependent, since
 *the registers to be saved, how to set up an initial
 *call frame, etc, are all specific to a processor architecture.
 *
 * This file currently supports the DEC MIPS and SUN SPARC
architectures.
 */
/*
 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 SWITCH_H
#define SWITCH_H
#include "copyright.h"
#ifdef HOST_MIPS
/* Registers that must be saved during a context switch.
 * These are the offsets from the beginning of the Thread object,
 * in bytes, used in switch.s
 */
#define SP 0
#define S0 4
#define S1 8
#define S2 12
#define S3 16
#define S4 20
#define S5 24
#define S6 28
#define S7 32
#define FP 36
#define PC 40
/* To fork a thread, we set up its saved register state, so that
 * when we switch to the thread, it will start running in ThreadRoot.
 *
 * The following are the initial registers we need to set up to
 * pass values into ThreadRoot (for instance, containing the procedure
 * for the thread to run).  The first set is the registers as used
 * by ThreadRoot; the second set is the locations for these initial
 * values in the Thread object -- used in Thread::AllocateStack().
 */
#define InitialPCs0
#define InitialArgs1
#define WhenDonePCs2
#define StartupPCs3
#define PCState(PC/4-1)
#define FPState(FP/4-1)
#define InitialPCState(S0/4-1)
#define InitialArgState(S1/4-1)
#define WhenDonePCState(S2/4-1)
#define StartupPCState(S3/4-1)
#endif // HOST_MIPS
#ifdef HOST_SPARC
/* Registers that must be saved during a context switch.  See comment
above. */
#define I0 4
#define I1 8
#define I2 12
#define I3 16
#define I4 20
#define I5 24
#define I6 28
#define I7 32
/* Aliases used for clearing code.  */
#define FP I6
#define PC I7
/* Registers for ThreadRoot.  See comment above. */
#define InitialPC       %o0
#define InitialArg      %o1
#define WhenDonePC      %o2
#define StartupPC       %o3
#define PCState         (PC/4-1)
#define InitialPCState  (I0/4-1)
#define InitialArgState (I1/4-1)
#define WhenDonePCState (I2/4-1)
#define StartupPCState  (I3/4-1)
#endif // HOST_SPARC
#ifdef HOST_SNAKE
/* Registers that must be saved during a context switch.  See comment
above. */
#define   SP   0
#define   S0   4
#define   S1   8
#define   S2   12
#define   S3   16
#define   S4   20
#define   S5   24
#define   S6   28
#define   S7   32
#define   S8   36
#define   S9   40
#define   S10  44
#define   S11  48
#define   S12  52
#define   S13  56
#define   S14  60
#define   S15  64
#define   PC   68
/* Registers for ThreadRoot.  See comment above. */
#define InitialPC       %r3/* S0 */
#define InitialArg      %r4
#define WhenDonePC      %r5
#define StartupPC       %r6
#define PCState         (PC/4-1)
#define InitialPCState  (S0/4-1)
#define InitialArgState (S1/4-1)
#define WhenDonePCState (S2/4-1)
#define StartupPCState  (S3/4-1)
#endif // HOST_SNAKE
```

```
#ifdef HOST_i386
/* the offsets of the registers from the beginning of the thread object
*/
#define _ESP     0
#define _EAX     4
#define _EBX     8
#define _ECX     12
#define _EDX     16
#define _EBP     20
#define _ESI     24
#define _EDI     28
#define _PC      32
/* These definitions are used in Thread::AllocateStack(). */
#define PCState         (_PC/4-1)
#define FPState         (_EBP/4-1)
#define InitialPCState  (_ESI/4-1)
#define InitialArgState (_EDX/4-1)
#define WhenDonePCState (_EDI/4-1)
#define StartupPCState  (_ECX/4-1)
#define InitialPC       %esi
#define InitialArg      %edx
#define WhenDonePC      %edi
#define StartupPC       %ecx
#endif
#endif // SWITCH_H
```

```
// synch.cc
//Routines for synchronizing threads.  Three kinds of
//synchronization routines are defined here: semaphores, locks
//   and condition variables (the implementation of the last two
//are left to the reader).
// Any implementation of a synchronization routine needs some
// primitive atomic operation.  We assume Nachos is running on
// a uniprocessor, and thus atomicity can be provided by
// turning off interrupts.  While interrupts are disabled, no
// context switch can occur, and thus the current thread is guaranteed
// to hold the CPU throughout, until interrupts are reenabled.
// Because some of these routines might be called with interrupts
// already disabled (Semaphore::V for one), instead of turning
// on interrupts at the end of the atomic operation, we always simply
// re-set the interrupt state back to its original value (whether
// that be disabled or enabled).
#include "copyright.h"
#include "synch.h"
#include "system.h"
//------------------------------------------------------------------
// Semaphore::Semaphore
// Initialize a semaphore, so that it can be used for synchronization.
//
//"debugName" is an arbitrary name, useful for debugging.
//"initialValue" is the initial value of the semaphore.
//------------------------------------------------------------------
Semaphore::Semaphore(char* debugName, int initialValue)
{
    name = debugName;
    value = initialValue;
    queue = new List;
}
//------------------------------------------------------------------
// Semaphore::Semaphore
// De-allocate semaphore, when no longer needed.  Assume no one
//is still waiting on the semaphore!
//------------------------------------------------------------------
Semaphore::~Semaphore()
{
    delete queue;
}
//------------------------------------------------------------------
// Semaphore::P
// Wait until semaphore value > 0, then decrement.  Checking the
//value and decrementing must be done atomically, so we
//need to disable interrupts before checking the value.
//Note that Thread::Sleep assumes that interrupts are disabled
//when it is called.
//------------------------------------------------------------------
void
Semaphore::P()
{
    IntStatus oldLevel = interrupt->SetLevel(IntOff);// disable
interrupts

    while (value == 0) { // semaphore not available
queue->Append((void *)currentThread);// so go to sleep
currentThread->Sleep();
    }
    value--; // semaphore available,
// consume its value

    (void) interrupt->SetLevel(oldLevel);// re-enable interrupts
}
//------------------------------------------------------------------
// Semaphore::V
// Increment semaphore value, waking up a waiter if necessary.
//As with P(), this operation must be atomic, so we need to disable
//interrupts.  Scheduler::ReadyToRun() assumes that threads
//are disabled when it is called.
//------------------------------------------------------------------
void
Semaphore::V()
{
    Thread *thread;
    IntStatus oldLevel = interrupt->SetLevel(IntOff);
    thread = (Thread *)queue->Remove();
    if (thread != NULL)   // make thread ready, consuming the V
immediately
scheduler->ReadyToRun(thread);
    value++;
    (void) interrupt->SetLevel(oldLevel);
}
// Dummy functions -- so we can compile our later assignments
// Note -- without a correct implementation of Condition::Wait(),
// the test case in the network assignment won't work!
Lock::Lock(char* debugName) {}
Lock::~Lock() {}
void Lock::Acquire() {}
void Lock::Release() {}
Condition::Condition(char* debugName) { }
Condition::~Condition() { }
void Condition::Wait(Lock* conditionLock) { ASSERT(FALSE); }
void Condition::Signal(Lock* conditionLock) { }
void Condition::Broadcast(Lock* conditionLock) { }
```

```
// synch.h
//Data structures for synchronizing threads.
//
//Three kinds of synchronization are defined here: semaphores,
//locks, and condition variables.  The implementation for
//semaphores is given; for the latter two, only the procedure
//interface is given -- they are to be implemented as part of
//the first assignment.
//
//Note that all the synchronization objects take a "name" as
//part of the initialization.  This is solely for debugging purposes.
//
// Copyright (c) 1992-1993 The Regents of the University of California.
// All rights reserved.  See copyright.h for copyright notice and
limitation
// synch.h -- synchronization primitives.
#ifndef SYNCH_H
#define SYNCH_H
#include "copyright.h"
#include "thread.h"
#include "list.h"
// The following class defines a "semaphore" whose value is a non-
negative
// integer.  The semaphore has only two operations P() and V():
//
//P() -- waits until value > 0, then decrement
//
//V() -- increment, waking up a thread waiting in P() if necessary
//
// Note that the interface does *not* allow a thread to read the value
of
// the semaphore directly -- even if you did read the value, the
// only thing you would know is what the value used to be.  You don't
// know what the value is now, because by the time you get the value
// into a register, a context switch might have occurred,
// and some other thread might have called P or V, so the true value
might
// now be different.
class Semaphore {
  public:
    Semaphore(char* debugName, int initialValue);// set initial value
    ~Semaphore();   // de-allocate semaphore
    char* getName() { return name;}// debugging assist

    void P(); // these are the only operations on a semaphore
    void V(); // they are both *atomic*

  private:
    char* name;        // useful for debugging
    int value;         // semaphore value, always >= 0
    List *queue;       // threads waiting in P() for the value to be >
0
};
// The following class defines a "lock".  A lock can be BUSY or FREE.
// There are only two operations allowed on a lock:
//
//Acquire -- wait until the lock is FREE, then set it to BUSY
//
//Release -- set lock to be FREE, waking up a thread waiting
//in Acquire if necessary
//
// In addition, by convention, only the thread that acquired the lock
// may release it.  As with semaphores, you can't read the lock value
// (because the value might change immediately after you read it).
class Lock {
  public:
    Lock(char* debugName);  // initialize lock to be FREE
    ~Lock();// deallocate lock
    char* getName() { return name; }// debugging assist
    void Acquire(); // these are the only operations on a lock
    void Release(); // they are both *atomic*
    bool isHeldByCurrentThread();// true if the current thread
// holds this lock.  Useful for
// checking in Release, and in
// Condition variable ops below.
  private:
    char* name;// for debugging
    // plus some other stuff you'll need to define
};
// The following class defines a "condition variable".  A condition
// variable does not have a value, but threads may be queued, waiting
// on the variable.  These are only operations on a condition
variable:
//
//Wait() -- release the lock, relinquish the CPU until signaled,
//then re-acquire the lock
//
//Signal() -- wake up a thread, if there are any waiting on
//the condition
//
//Broadcast() -- wake up all threads waiting on the condition
//
// All operations on a condition variable must be made while
// the current thread has acquired a lock.  Indeed, all accesses
// to a given condition variable must be protected by the same lock.
// In other words, mutual exclusion must be enforced among threads
calling
// the condition variable operations.
//
// In Nachos, condition variables are assumed to obey *Mesa*-style
// semantics.  When a Signal or Broadcast wakes up another thread,
// it simply puts the thread on the ready list, and it is the
responsibility
// of the woken thread to re-acquire the lock (this re-acquire is
// taken care of within Wait()).  By contrast, some define condition
// variables according to *Hoare*-style semantics -- where the
signalling
// thread gives up control over the lock and the CPU to the woken
thread,
// which runs immediately and gives back control over the lock to the
// signaller when the woken thread leaves the critical section.
//
```

```
// The consequence of using Mesa-style semantics is that some other
thread
// can acquire the lock, and change data structures, before the woken
// thread gets a chance to run.
class Condition {
  public:
    Condition(char* debugName);// initialize condition to
// "no one waiting"
    ~Condition();// deallocate the condition
    char* getName() { return (name); }

    void Wait(Lock *conditionLock); // these are the 3 operations on
// condition variables; releasing the
// lock and going to sleep are
// *atomic* in Wait()
    void Signal(Lock *conditionLock);   // conditionLock must be held
by
    void Broadcast(Lock *conditionLock);// the currentThread for all of
// these operations
  private:
    char* name;
    // plus some other stuff you'll need to define
};
#endif // SYNCH_H
```

```
// synchlist.cc
//Routines for synchronized access to a list.
//
//Implemented by surrounding the List abstraction
//with synchronization routines.
//
// Implemented in "monitor"-style -- surround each procedure with a
// lock acquire and release pair, using condition signal and wait for
// synchronization.
//
// 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.
#include "copyright.h"
#include "synchlist.h"
//----------------------------------------------------------------------
// SynchList::SynchList
//Allocate and initialize the data structures needed for a
//synchronized list, empty to start with.
//Elements can now be added to the list.
//----------------------------------------------------------------------
SynchList::SynchList()
{
    list = new List();
    lock = new Lock("list lock");
    listEmpty = new Condition("list empty cond");
}
//----------------------------------------------------------------------
// SynchList::~SynchList
//De-allocate the data structures created for synchronizing a list.
//----------------------------------------------------------------------
SynchList::~SynchList()
{
    delete list;
    delete lock;
    delete listEmpty;
}
//----------------------------------------------------------------------
// SynchList::Append
//      Append an "item" to the end of the list.  Wake up anyone
//waiting for an element to be appended.
//
//"item" is the thing to put on the list, it can be a pointer to
//anything.
//----------------------------------------------------------------------
void
SynchList::Append(void *item)
{
    lock->Acquire();// enforce mutual exclusive access to the list
    list->Append(item);
    listEmpty->Signal(lock);// wake up a waiter, if any
    lock->Release();
}
//----------------------------------------------------------------------
// SynchList::Remove
//      Remove an "item" from the beginning of the list.  Wait if
//the list is empty.
// Returns:
//The removed item.
//----------------------------------------------------------------------
void *
SynchList::Remove()
{
    void *item;
    lock->Acquire();// enforce mutual exclusion
    while (list->IsEmpty())
listEmpty->Wait(lock);// wait until list isn't empty
    item = list->Remove();
    ASSERT(item != NULL);
    lock->Release();
    return item;
}
//----------------------------------------------------------------------
// SynchList::Mapcar
//      Apply function to every item on the list.  Obey mutual
exclusion
//constraints.
//
//"func" is the procedure to be applied.
//----------------------------------------------------------------------
void
SynchList::Mapcar(VoidFunctionPtr func)
{
    lock->Acquire();
    list->Mapcar(func);
    lock->Release();
}
```

```
// synchlist.h
//Data structures for synchronized access to a list.
//
//Implemented by surrounding the List abstraction
//with synchronization routines.
//
// 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 SYNCHLIST_H
#define SYNCHLIST_H
#include "copyright.h"
#include "list.h"
#include "synch.h"
// The following class defines a "synchronized list" -- a list for 
which:
// these constraints hold:
//1. Threads trying to remove an item from a list will
//wait until the list has an element on it.
//2. One thread at a time can access list data structures
class SynchList {
  public:
    SynchList();// initialize a synchronized list
    ~SynchList();// de-allocate a synchronized list
    void Append(void *item);// append item to the end of the list,
// and wake up any thread waiting in remove
    void *Remove();// remove the first item from the front of
// the list, waiting if the list is empty
// apply function to every item in the list
    void Mapcar(VoidFunctionPtr func);
  private:
    List *list;// the unsynchronized list
    Lock *lock;// enforce mutual exclusive access to the list
    Condition *listEmpty;// wait in Remove if the list is empty
};
#endif // SYNCHLIST_H
```

```
// system.cc
//Nachos initialization and cleanup routines.
//
// 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.
#include "copyright.h"
#include "system.h"
// This defines *all* of the global data structures used by Nachos.
// These are all initialized and de-allocated by this file.
Thread *currentThread;// the thread we are running now
Thread *threadToBeDestroyed;  // the thread that just finished
Scheduler *scheduler;// the ready list
Interrupt *interrupt;// interrupt status
Statistics *stats;// performance metrics
Timer *timer;// the hardware timer device,
// for invoking context switches
#ifdef FILESYS_NEEDED
FileSystem  *fileSystem;
#endif
#ifdef FILESYS
SynchDisk   *synchDisk;
#endif
#ifdef USER_PROGRAM// requires either FILESYS or FILESYS_STUB
Machine *machine;// user program memory and registers
#endif
#ifdef NETWORK
PostOffice *postOffice;
#endif
// External definition, to allow us to take a pointer to this function
extern void Cleanup();
//----------------------------------------------------------------------
// TimerInterruptHandler
// Interrupt handler for the timer device.  The timer device is
//set up to interrupt the CPU periodically (once every TimerTicks).
//This routine is called each time there is a timer interrupt,
//with interrupts disabled.
//
//Note that instead of calling Yield() directly (which would
//suspend the interrupt handler, not the interrupted thread
//which is what we wanted to context switch), we set a flag
//so that once the interrupt handler is done, it will appear as
//if the interrupted thread called Yield at the point it is
//was interrupted.
//
//"dummy" is because every interrupt handler takes one argument,
//whether it needs it or not.
//----------------------------------------------------------------------
static void
TimerInterruptHandler(int dummy)
{
    if (interrupt->getStatus() != IdleMode)
interrupt->YieldOnReturn();
}
//----------------------------------------------------------------------
// Initialize
// Initialize Nachos global data structures.  Interpret command
//line arguments in order to determine flags for the initialization.
//
//"argc" is the number of command line arguments (including the name
//of the command) -- ex: "nachos -d +" -> argc = 3
//"argv" is an array of strings, one for each command line argument
//ex: "nachos -d +" -> argv = {"nachos", "-d", "+"}
//----------------------------------------------------------------------
void
Initialize(int argc, char **argv)
{
    int argCount;
    char* debugArgs = "";
    bool randomYield = FALSE;
#ifdef USER_PROGRAM
    bool debugUserProg = FALSE;// single step user program
#endif
#ifdef FILESYS_NEEDED
    bool format = FALSE;// format disk
#endif
#ifdef NETWORK
    double rely = 1;// network reliability
    int netname = 0;// UNIX socket name
#endif

    for (argc--, argv++; argc > 0; argc -= argCount, argv += argCount)
{
argCount = 1;
if (!strcmp(*argv, "-d")) {
    if (argc == 1)
debugArgs = "+";// turn on all debug flags
    else {
    debugArgs = *(argv + 1);
    argCount = 2;
    }
} else if (!strcmp(*argv, "-rs")) {
    ASSERT(argc > 1);
    RandomInit(atoi(*(argv + 1)));// initialize pseudo-random
// number generator
    randomYield = TRUE;
    argCount = 2;
}
#ifdef USER_PROGRAM
if (!strcmp(*argv, "-s"))
    debugUserProg = TRUE;
#endif
#ifdef FILESYS_NEEDED
if (!strcmp(*argv, "-f"))
    format = TRUE;
#endif
#ifdef NETWORK
if (!strcmp(*argv, "-l")) {
    ASSERT(argc > 1);
    rely = atof(*(argv + 1));
    argCount = 2;
} else if (!strcmp(*argv, "-m")) {
    ASSERT(argc > 1);
    netname = atoi(*(argv + 1));
    argCount = 2;
}
#endif
    }
    DebugInit(debugArgs);// initialize DEBUG messages
    stats = new Statistics();// collect statistics
    interrupt = new Interrupt;// start up interrupt handling
    scheduler = new Scheduler();// initialize the ready queue
    if (randomYield)// start the timer (if needed)
timer = new Timer(TimerInterruptHandler, 0, randomYield);
    threadToBeDestroyed = NULL;
    // We didn't explicitly allocate the current thread we are running
in.
    // But if it ever tries to give up the CPU, we better have a Thread
    // object to save its state.
    currentThread = new Thread("main");
    currentThread->setStatus(RUNNING);
    interrupt->Enable();
    CallOnUserAbort(Cleanup);// if user hits ctl-C

#ifdef USER_PROGRAM
    machine = new Machine(debugUserProg);// this must come first
#endif
#ifdef FILESYS
    synchDisk = new SynchDisk("DISK");
#endif
#ifdef FILESYS_NEEDED
    fileSystem = new FileSystem(format);
#endif
#ifdef NETWORK
    postOffice = new PostOffice(netname, rely, 10);
#endif
}
//----------------------------------------------------------------------
// Cleanup
// Nachos is halting.  De-allocate global data structures.
//----------------------------------------------------------------------
void
Cleanup()
{
    printf("\nCleaning up...\n");
#ifdef NETWORK
    delete postOffice;
#endif

#ifdef USER_PROGRAM
    delete machine;
#endif
#ifdef FILESYS_NEEDED
    delete fileSystem;
#endif
#ifdef FILESYS
    delete synchDisk;
#endif

    delete timer;
    delete scheduler;
    delete interrupt;

    Exit(0);
}
```

```
// system.h
//All global variables used in Nachos are defined here.
//
// 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 SYSTEM_H
#define SYSTEM_H
#include "copyright.h"
#include "utility.h"
#include "thread.h"
#include "scheduler.h"
#include "interrupt.h"
#include "stats.h"
#include "timer.h"
// Initialization and cleanup routines
extern void Initialize(int argc, char **argv); // Initialization,
// called before anything else
extern void Cleanup();// Cleanup, called when
// Nachos is done.
extern Thread *currentThread;// the thread holding the CPU
extern Thread *threadToBeDestroyed;  // the thread that just finished
extern Scheduler *scheduler;// the ready list
extern Interrupt *interrupt;// interrupt status
extern Statistics *stats;// performance metrics
extern Timer *timer;// the hardware alarm clock
#ifdef USER_PROGRAM
#include "machine.h"
extern Machine* machine;// user program memory and registers
#endif
#ifdef FILESYS_NEEDED // FILESYS or FILESYS_STUB
#include "filesys.h"
extern FileSystem  *fileSystem;
#endif
#ifdef FILESYS
#include "synchdisk.h"
extern SynchDisk   *synchDisk;
#endif
#ifdef NETWORK
#include "post.h"
extern PostOffice* postOffice;
#endif
#endif // SYSTEM_H
```

```
// thread.cc
//Routines to manage threads.  There are four main operations:
//Fork -- create a thread to run a procedure concurrently
//with the caller (this is done in two steps -- first
//allocate the Thread object, then call Fork on it)
//Finish -- called when the forked procedure finishes, to clean up
//Yield -- relinquish control over the CPU to another ready thread
//Sleep -- relinquish control over the CPU, but thread is now blocked.
//In other words, it will not run again, until explicitly
//put back on the ready queue.
#include "copyright.h"
#include "thread.h"
#include "switch.h"
#include "synch.h"
#include "system.h"
#define STACK_FENCEPOST 0xdeadbeef// this is put at the top of the
// execution stack, for detecting  stack overflows
//-------------------------------------------------------------------
// Thread::Thread
// Initialize a thread control block, so that we can then call
//Thread::Fork.
//"threadName" is an arbitrary string, useful for debugging.
//-------------------------------------------------------------------
Thread::Thread(char* threadName)
{
    name = threadName;
    stackTop = NULL;
    stack = NULL;
    status = JUST_CREATED;
#ifdef USER_PROGRAM
    space = NULL;
#endif
}
//-------------------------------------------------------------------
// Thread::~Thread
// De-allocate a thread.
// NOTE: the current thread *cannot* delete itself directly,
//since it is still running on the stack that we need to delete.
//      NOTE: if this is the main thread, we can't delete the stack
//      because we didn't allocate it -- we got it automatically
//      as part of starting up Nachos.
//-------------------------------------------------------------------
Thread::~Thread()
{
    DEBUG('t', "Deleting thread \"%s\"\n", name);
    ASSERT(this != currentThread);
    if (stack != NULL)
DeallocBoundedArray((char *) stack, StackSize * sizeof(int));
}
//-------------------------------------------------------------------
// Thread::Fork
// Invoke (*func)(arg), allowing caller and callee to execute
concurrently.
//NOTE: although our definition allows only a single integer argument
//to be passed to the procedure, it is possible to pass multiple
//arguments by making them fields of a structure, and passing a pointer
//to the structure as "arg".
//
// Implemented as the following steps:
//1. Allocate a stack
//2. Initialize the stack so that a call to SWITCH will
//cause it to run the procedure
//3. Put the thread on the ready queue
//
//"func" is the procedure to run concurrently.
//"arg" is a single argument to be passed to the procedure.
//-------------------------------------------------------------------
void
Thread::Fork(VoidFunctionPtr func, int arg)
{
    DEBUG('t', "Forking thread \"%s\" with func = 0x%x, arg = %d\n",
  name, (int) func, arg);

    StackAllocate(func, arg);
    IntStatus oldLevel = interrupt->SetLevel(IntOff);
    scheduler->ReadyToRun(this);// ReadyToRun assumes that interrupts
// are disabled!
    (void) interrupt->SetLevel(oldLevel);
}
//-------------------------------------------------------------------
// Thread::CheckOverflow
// Check a thread's stack to see if it has overrun the space
//that has been allocated for it.  If we had a smarter compiler,
//we wouldn't need to worry about this, but we don't.
//
// NOTE: Nachos will not catch all stack overflow conditions.
//In other words, your program may still crash because of an overflow.
//
// If you get bizarre results (such as seg faults where there is no
code)
// then you *may* need to increase the stack size.  You can avoid
stack
// overflows by not putting large data structures on the stack.
// Don't do this: void foo() { int bigArray[10000]; ... }
//-------------------------------------------------------------------
void
Thread::CheckOverflow()
{
    if (stack != NULL)
#ifdef HOST_SNAKE// Stacks grow upward on the Snakes
ASSERT(stack[StackSize - 1] == STACK_FENCEPOST);
#else
ASSERT((int) *stack == (int) STACK_FENCEPOST);
#endif
}
//-------------------------------------------------------------------
// Thread::Finish
// Called by ThreadRoot when a thread is done executing the
//forked procedure.
//
// NOTE: we don't immediately de-allocate the thread data structure
//or the execution stack, because we're still running in the thread
```

```
//and we're still on the stack!  Instead, we set "threadToBeDestroyed",
//so that Scheduler::Run() will call the destructor, once we're
//running in the context of a different thread.
//
// NOTE: we disable interrupts, so that we don't get a time slice
//between setting threadToBeDestroyed, and going to sleep.
//----------------------------------------------------------------------
void
Thread::Finish ()
{
    (void) interrupt->SetLevel(IntOff);
    ASSERT(this == currentThread);

    DEBUG('t', "Finishing thread \"%s\"\n", getName());

    threadToBeDestroyed = currentThread;
    Sleep();// invokes SWITCH
    // not reached
}
//----------------------------------------------------------------------
// Thread::Yield
// Relinquish the CPU if any other thread is ready to run.
//If so, put the thread on the end of the ready list, so that
//it will eventually be re-scheduled.
//
//NOTE: returns immediately if no other thread on the ready queue.
//Otherwise returns when the thread eventually works its way
//to the front of the ready list and gets re-scheduled.
//
//NOTE: we disable interrupts, so that looking at the thread
//on the front of the ready list, and switching to it, can be done
//atomically.  On return, we re-set the interrupt level to its
//original state, in case we are called with interrupts disabled.
//
// Similar to Thread::Sleep(), but a little different.
//----------------------------------------------------------------------
void
Thread::Yield ()
{
    Thread *nextThread;
    IntStatus oldLevel = interrupt->SetLevel(IntOff);

    ASSERT(this == currentThread);

    DEBUG('t', "Yielding thread \"%s\"\n", getName());

    nextThread = scheduler->FindNextToRun();
    if (nextThread != NULL) {
scheduler->ReadyToRun(this);
scheduler->Run(nextThread);
    }
    (void) interrupt->SetLevel(oldLevel);
}
//----------------------------------------------------------------------
// Thread::Sleep
// Relinquish the CPU, because the current thread is blocked
//waiting on a synchronization variable (Semaphore, Lock, or
Condition).
//Eventually, some thread will wake this thread up, and put it
//back on the ready queue, so that it can be re-scheduled.
//
//NOTE: if there are no threads on the ready queue, that means
//we have no thread to run.  "Interrupt::Idle" is called
//to signify that we should idle the CPU until the next I/O interrupt
//occurs (the only thing that could cause a thread to become
//ready to run).
//
//NOTE: we assume interrupts are already disabled, because it
//is called from the synchronization routines which must
//disable interrupts for atomicity.   We need interrupts off
//so that there can't be a time slice between pulling the first thread
//off the ready list, and switching to it.
//----------------------------------------------------------------------
void
Thread::Sleep ()
{
    Thread *nextThread;

    ASSERT(this == currentThread);
    ASSERT(interrupt->getLevel() == IntOff);

    DEBUG('t', "Sleeping thread \"%s\"\n", getName());
    status = BLOCKED;
    while ((nextThread = scheduler->FindNextToRun()) == NULL)
interrupt->Idle();// no one to run, wait for an interrupt

    scheduler->Run(nextThread); // returns when we've been signalled
}
//----------------------------------------------------------------------
// ThreadFinish, InterruptEnable, ThreadPrint
//Dummy functions because C++ does not allow a pointer to a member
//function.  So in order to do this, we create a dummy C function
//(which we can pass a pointer to), that then simply calls the
//member function.
//----------------------------------------------------------------------
static void ThreadFinish()    { currentThread->Finish(); }
static void InterruptEnable() { interrupt->Enable(); }
void ThreadPrint(int arg){ Thread *t = (Thread *)arg; t->Print(); }
//----------------------------------------------------------------------
// Thread::StackAllocate
//Allocate and initialize an execution stack.  The stack is
//initialized with an initial stack frame for ThreadRoot, which:
//enables interrupts
//calls (*func)(arg)
//calls Thread::Finish
//
//"func" is the procedure to be forked
//"arg" is the parameter to be passed to the procedure
//----------------------------------------------------------------------
void
Thread::StackAllocate (VoidFunctionPtr func, int arg)
{
    stack = (int *) AllocBoundedArray(StackSize * sizeof(int));
#ifdef HOST_SNAKE
    // HP stack works from low addresses to high addresses
    stackTop = stack + 16;// HP requires 64-byte frame marker
    stack[StackSize - 1] = STACK_FENCEPOST;
#else
    // i386 & MIPS & SPARC stack works from high addresses to low
addresses
#ifdef HOST_SPARC
    // SPARC stack must contains at least 1 activation record to start
with.
    stackTop = stack + StackSize - 96;
#else  // HOST_MIPS  || HOST_i386
    stackTop = stack + StackSize - 4;// -4 to be on the safe side!
#ifdef HOST_i386
    // the 80386 passes the return address on the stack.  In order for
    // SWITCH() to go to ThreadRoot when we switch to this thread, the
    // return addres used in SWITCH() must be the starting address of
    // ThreadRoot.
    *(--stackTop) = (int)ThreadRoot;
#endif
#endif  // HOST_SPARC
   *stack = STACK_FENCEPOST;
#endif  // HOST_SNAKE

    machineState[PCState] = (int) ThreadRoot;
    machineState[StartupPCState] = (int) InterruptEnable;
    machineState[InitialPCState] = (int) func;
    machineState[InitialArgState] = arg;
    machineState[WhenDonePCState] = (int) ThreadFinish;
}
#ifdef USER_PROGRAM
#include "machine.h"
//----------------------------------------------------------------------
// Thread::SaveUserState
//Save the CPU state of a user program on a context switch.
//
//Note that a user program thread has *two* sets of CPU registers --
//one for its state while executing user code, one for its state
//while executing kernel code.  This routine saves the former.
//----------------------------------------------------------------------
void
Thread::SaveUserState()
{
    for (int i = 0; i < NumTotalRegs; i++)
userRegisters[i] = machine->ReadRegister(i);
}
//----------------------------------------------------------------------
// Thread::RestoreUserState
//Restore the CPU state of a user program on a context switch.
//
//Note that a user program thread has *two* sets of CPU registers --
//one for its state while executing user code, one for its state
//while executing kernel code.  This routine restores the former.
//----------------------------------------------------------------------
void
Thread::RestoreUserState()
{
    for (int i = 0; i < NumTotalRegs; i++)
machine->WriteRegister(i, userRegisters[i]);
}
#endif
```

```
// thread.h
//Data structures for managing threads.  A thread represents
//sequential execution of code within a program.
//So the state of a thread includes the program counter,
//the processor registers, and the execution stack.
//
// Note that because we allocate a fixed size stack for each
//thread, it is possible to overflow the stack -- for instance,
//by recursing to too deep a level.  The most common reason
//for this occuring is allocating large data structures
//on the stack.  For instance, this will cause problems:
//
//void foo() { int buf[1000]; ...}
//
//Instead, you should allocate all data structures dynamically:
//
//void foo() { int *buf = new int[1000]; ...}
//
//
// Bad things happen if you overflow the stack, and in the worst
//case, the problem may not be caught explicitly.  Instead,
//the only symptom may be bizarre segmentation faults.  (Of course,
//other problems can cause seg faults, so that isn't a sure sign
//that your thread stacks are too small.)
//
//One thing to try if you find yourself with seg faults is to
//increase the size of thread stack -- ThreadStackSize.
//
//  In this interface, forking a thread takes two steps.
//We must first allocate a data structure for it: "t = new Thread".
//Only then can we do the fork: "t->fork(f, arg)".
//
// 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 THREAD_H
#define THREAD_H
#include "copyright.h"
#include "utility.h"
#ifdef USER_PROGRAM
#include "machine.h"
#include "addrspace.h"
#endif
// CPU register state to be saved on context switch.
// The SPARC and MIPS only need 10 registers, but the Snake needs 18.
// For simplicity, this is just the max over all architectures.
#define MachineStateSize 18
// Size of the thread's private execution stack.
// WATCH OUT IF THIS ISN'T BIG ENOUGH!!!!!
#define StackSize(4 * 1024)// in words
// Thread state
enum ThreadStatus { JUST_CREATED, RUNNING, READY, BLOCKED };
// external function, dummy routine whose sole job is to call
Thread::Print
extern void ThreadPrint(int arg);
// The following class defines a "thread control block" -- which
// represents a single thread of execution.
//
//  Every thread has:
//     an execution stack for activation records ("stackTop" and
"stack")
//     space to save CPU registers while not running ("machineState")
//     a "status" (running/ready/blocked)
//
//  Some threads also belong to a user address space; threads
//  that only run in the kernel have a NULL address space.
class Thread {
  private:
    // NOTE: DO NOT CHANGE the order of these first two members.
    // THEY MUST be in this position for SWITCH to work.
    int* stackTop; // the current stack pointer
    int machineState[MachineStateSize];  // all registers except for
stackTop
  public:
    Thread(char* debugName);// initialize a Thread
    ~Thread(); // deallocate a Thread
// NOTE -- thread being deleted
// must not be running when delete
// is called
    // basic thread operations
    void Fork(VoidFunctionPtr func, int arg); // Make thread run
(*func)(arg)
    void Yield();  // Relinquish the CPU if any
// other thread is runnable
    void Sleep();  // Put the thread to sleep and
// relinquish the processor
    void Finish();  // The thread is done executing

    void CheckOverflow();   // Check if thread has
// overflowed its stack
    void setStatus(ThreadStatus st) { status = st; }
    char* getName() { return (name); }
    void Print() { printf("%s, ", name); }
  private:
    // some of the private data for this class is listed above

    int* stack;  // Bottom of the stack
// NULL if this is the main thread
// (If NULL, don't deallocate stack)
    ThreadStatus status;// ready, running or blocked
    char* name;
    void StackAllocate(VoidFunctionPtr func, int arg);
    // Allocate a stack for thread.
// Used internally by Fork()
#ifdef USER_PROGRAM
// A thread running a user program actually has *two* sets of CPU
registers --
// one for its state while executing user code, one for its state
// while executing kernel code.
    int userRegisters[NumTotalRegs];// user-level CPU register state
  public:
    void SaveUserState();// save user-level register state
    void RestoreUserState();// restore user-level register state
    AddrSpace *space;// User code this thread is running.
#endif
};
// Magical machine-dependent routines, defined in switch.s
extern "C" {
// First frame on thread execution stack;
//   enable interrupts
//call "func"
//(when func returns, if ever) call ThreadFinish()
void ThreadRoot();
// Stop running oldThread and start running newThread
void SWITCH(Thread *oldThread, Thread *newThread);
}
#endif // THREAD_H
```

```
// threadtest.cc
//Simple test case for the threads assignment.
//
//Create two threads, and have them context switch
//back and forth between themselves by calling Thread::Yield,
//to illustratethe inner workings of the thread system.
//
// 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.
#include "copyright.h"
#include "system.h"
// testnum is set in main.cc
int testnum = 1;
//-------------------------------------------------------------------
// SimpleThread
// Loop 5 times, yielding the CPU to another ready thread
//each iteration.
//
//"which" is simply a number identifying the thread, for debugging
//purposes.
//-------------------------------------------------------------------
void
SimpleThread(int which)
{
    int num;

    for (num = 0; num < 5; num++) {
printf("*** thread %d looped %d times\n", which, num);
        currentThread->Yield();
    }
}
//-------------------------------------------------------------------
// ThreadTest1
// Set up a ping-pong between two threads, by forking a thread
//to call SimpleThread, and then calling SimpleThread ourselves.
//-------------------------------------------------------------------
void
ThreadTest1()
{
    DEBUG('t', "Entering ThreadTest1");
    Thread *t = new Thread("forked thread");
    t->Fork(SimpleThread, 1);
    SimpleThread(0);
}
//-------------------------------------------------------------------
// ThreadTest
// Invoke a test routine.
//-------------------------------------------------------------------
void
ThreadTest()
{
    switch (testnum) {
    case 1:
ThreadTest1();
break;
    default:
printf("No test specified.\n");
break;
    }
}
```

```
// utility.cc
//Debugging routines.  Allows users to control whether to
//print DEBUG statements, based on a command line argument.
//
// 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.
#include "copyright.h"
#include "utility.h"
// this seems to be dependent on how the compiler is configured.
// if you have problems with va_start, try both of these alternatives
#ifdef HOST_SNAKE
#include <stdarg.h>
#else
#ifdef HOST_SPARC
#include <stdarg.h>
#else
#include "stdarg.h"
#endif
#endif
static char *enableFlags = NULL; // controls which DEBUG messages are
printed
//----------------------------------------------------------------------
// DebugInit
//      Initialize so that only DEBUG messages with a flag in flagList
//will be printed.
//
//If the flag is "+", we enable all DEBUG messages.
//
// "flagList" is a string of characters for whose DEBUG messages are
//to be enabled.
//----------------------------------------------------------------------
void
DebugInit(char *flagList)
{
    enableFlags = flagList;
}
//----------------------------------------------------------------------
// DebugIsEnabled
//      Return TRUE if DEBUG messages with "flag" are to be printed.
//----------------------------------------------------------------------
bool
DebugIsEnabled(char flag)
{
    if (enableFlags != NULL)
       return (strchr(enableFlags, flag) != 0)
|| (strchr(enableFlags, '+') != 0);
    else
      return FALSE;
}
//----------------------------------------------------------------------
// DEBUG
//      Print a debug message, if flag is enabled.  Like printf,
//only with an extra argument on the front.
//----------------------------------------------------------------------
void
DEBUG(char flag, char *format, ...)
{
    if (DebugIsEnabled(flag)) {
va_list ap;
// You will get an unused variable message here -- ignore it.
va_start(ap, format);
vfprintf(stdout, format, ap);
va_end(ap);
fflush(stdout);
    }
}
```

```
// utility.h
//Miscellaneous useful definitions, including debugging routines.
//
//The debugging routines allow the user to turn on selected
//debugging messages, controllable from the command line arguments
//passed to Nachos (-d).  You are encouraged to add your own
//debugging flags.  The pre-defined debugging flags are:
//
//'+' -- turn on all debug messages
//   't' -- thread system
//   's' -- semaphores, locks, and conditions
//   'i' -- interrupt emulation
//   'm' -- machine emulation (USER_PROGRAM)
//   'd' -- disk emulation (FILESYS)
//   'f' -- file system (FILESYS)
//   'a' -- address spaces (USER_PROGRAM)
//   'n' -- network emulation (NETWORK)
//
// 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 UTILITY_H
#define UTILITY_H
#include "copyright.h"
// Miscellaneous useful routines
#include "bool.h"
 // Boolean values.
// This is the same definition
// as in the g++ library.
#define min(a,b)  (((a) < (b)) ? (a) : (b))
#define max(a,b)  (((a) > (b)) ? (a) : (b))
// Divide and either round up or down
#define divRoundDown(n,s)  ((n) / (s))
#define divRoundUp(n,s)    (((n) / (s)) + ((((n) % (s)) > 0) ? 1 : 0))
// This declares the type "VoidFunctionPtr" to be a "pointer to a
// function taking an integer argument and returning nothing".  With
// such a function pointer (say it is "func"), we can call it like
this:
//
//(*func) (17);
//
// This is used by Thread::Fork and for interrupt handlers, as well
// as a couple of other places.
typedef void (*VoidFunctionPtr)(int arg);
typedef void (*VoidNoArgFunctionPtr)();
// Include interface that isolates us from the host machine system
library.
// Requires definition of bool, and VoidFunctionPtr
#include "sysdep.h"
// Interface to debugging routines.
extern void DebugInit(char* flags);// enable printing debug messages
extern bool DebugIsEnabled(char flag); // Is this debug flag enabled?
extern void DEBUG (char flag, char* format, ...);  // Print debug
message
// if flag is enabled
//-------------------------------------------------------------------
// ASSERT
//      If condition is false,  print a message and dump core.
//Useful for documenting assumptions in the code.
//
//NOTE: needs to be a #define, to be able to print the location
//where the error occurred.
//-------------------------------------------------------------------
#define ASSERT(condition)
\
    if (!(condition)) {
\
        fprintf(stderr, "Assertion failed: line %d, file \"%s\"\n",
\
                __LINE__, __FILE__);
\
fflush(stderr);      \
        Abort();
\
    }
#endif // UTILITY_H
```

```
// directory.cc
//Routines to manage a directory of file names.
//
//The directory is a table of fixed length entries; each
//entry represents a single file, and contains the file name,
//and the location of the file header on disk.  The fixed size
//of each directory entry means that we have the restriction
//of a fixed maximum size for file names.
//
//The constructor initializes an empty directory of a certain size;
//we use ReadFrom/WriteBack to fetch the contents of the directory
//from disk, and to write back any modifications back to disk.
//
//Also, this implementation has the restriction that the size
//of the directory cannot expand.  In other words, once all the
//entries in the directory are used, no more files can be created.
//Fixing this is one of the parts to the assignment.
//
// 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.
#include "copyright.h"
#include "utility.h"
#include "filehdr.h"
#include "directory.h"
//------------------------------------------------------------------
// Directory::Directory
// Initialize a directory; initially, the directory is completely
//empty.  If the disk is being formatted, an empty directory
//is all we need, but otherwise, we need to call FetchFrom in order
//to initialize it from disk.
//
//"size" is the number of entries in the directory
//------------------------------------------------------------------
Directory::Directory(int size)
{
    table = new DirectoryEntry[size];
    tableSize = size;
    for (int i = 0; i < tableSize; i++)
table[i].inUse = FALSE;
}
//------------------------------------------------------------------
// Directory::~Directory
// De-allocate directory data structure.
//------------------------------------------------------------------
Directory::~Directory()
{
    delete [] table;
}
//------------------------------------------------------------------
// Directory::FetchFrom
// Read the contents of the directory from disk.
//
//"file" -- file containing the directory contents
//------------------------------------------------------------------
void
Directory::FetchFrom(OpenFile *file)
{
    (void) file->ReadAt((char *)table, tableSize *
sizeof(DirectoryEntry), 0);
}
//------------------------------------------------------------------
// Directory::WriteBack
// Write any modifications to the directory back to disk
//
//"file" -- file to contain the new directory contents
//------------------------------------------------------------------
void
Directory::WriteBack(OpenFile *file)
{
    (void) file->WriteAt((char *)table, tableSize *
sizeof(DirectoryEntry), 0);
}
//------------------------------------------------------------------
// Directory::FindIndex
// Look up file name in directory, and return its location in the
table of
//directory entries.  Return -1 if the name isn't in the directory.
//
//"name" -- the file name to look up
//------------------------------------------------------------------
int
Directory::FindIndex(char *name)
{
    for (int i = 0; i < tableSize; i++)
        if (table[i].inUse && !strncmp(table[i].name, name,
FileNameMaxLen))
    return i;
    return -1;// name not in directory
}
//------------------------------------------------------------------
// Directory::Find
// Look up file name in directory, and return the disk sector number
//where the file's header is stored. Return -1 if the name isn't
//in the directory.
//
//"name" -- the file name to look up
//------------------------------------------------------------------
int
Directory::Find(char *name)
{
    int i = FindIndex(name);
    if (i != -1)
return table[i].sector;
    return -1;
}
//------------------------------------------------------------------
// Directory::Add
// Add a file into the directory.  Return TRUE if successful;
//return FALSE if the file name is already in the directory, or if
//the directory is completely full, and has no more space for
```

```
//additional file names.
//
//"name" -- the name of the file being added
//"newSector" -- the disk sector containing the added file's header
//----------------------------------------------------------------------
bool
Directory::Add(char *name, int newSector)
{
    if (FindIndex(name) != -1)
return FALSE;
    for (int i = 0; i < tableSize; i++)
        if (!table[i].inUse) {
            table[i].inUse = TRUE;
            strncpy(table[i].name, name, FileNameMaxLen);
            table[i].sector = newSector;
        return TRUE;
}
    return FALSE;// no space.  Fix when we have extensible files.
}
//----------------------------------------------------------------------
// Directory::Remove
// Remove a file name from the directory.  Return TRUE if successful;
//return FALSE if the file isn't in the directory.
//
//"name" -- the file name to be removed
//----------------------------------------------------------------------
bool
Directory::Remove(char *name)
{
    int i = FindIndex(name);
    if (i == -1)
return FALSE; // name not in directory
    table[i].inUse = FALSE;
    return TRUE;
}
//----------------------------------------------------------------------
// Directory::List
// List all the file names in the directory.
//----------------------------------------------------------------------
void
Directory::List()
{
   for (int i = 0; i < tableSize; i++)
if (table[i].inUse)
    printf("%s\n", table[i].name);
}
//----------------------------------------------------------------------
// Directory::Print
// List all the file names in the directory, their FileHeader
locations,
//and the contents of each file.  For debugging.
//----------------------------------------------------------------------
void
Directory::Print()
{
    FileHeader *hdr = new FileHeader;
```

```
    printf("Directory contents:\n");
    for (int i = 0; i < tableSize; i++)
if (table[i].inUse) {
    printf("Name: %s, Sector: %d\n", table[i].name, table[i].sector);
    hdr->FetchFrom(table[i].sector);
    hdr->Print();
}
    printf("\n");
    delete hdr;
}
```

```
// directory.h
//Data structures to manage a UNIX-like directory of file names.
//
//      A directory is a table of pairs: <file name, sector #>,
//giving the name of each file in the directory, and
//where to find its file header (the data structure describing
//where to find the file's data blocks) on disk.
//
//      We assume mutual exclusion is provided by the caller.
//
// 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.
#include "copyright.h"
#ifndef DIRECTORY_H
#define DIRECTORY_H
#include "openfile.h"
#define FileNameMaxLen 9// for simplicity, we assume
// file names are <= 9 characters long
// The following class defines a "directory entry", representing a file
// in the directory.  Each entry gives the name of the file, and where
// the file's header is to be found on disk.
//
// Internal data structures kept public so that Directory operations
can
// access them directly.
class DirectoryEntry {
  public:
    bool inUse;// Is this directory entry in use?
    int sector;// Location on disk to find the
//   FileHeader for this file
    char name[FileNameMaxLen + 1];// Text name for file, with +1 for
// the trailing '\0'
};
// The following class defines a UNIX-like "directory".  Each entry in
// the directory describes a file, and where to find it on disk.
//
// The directory data structure can be stored in memory, or on disk.
// When it is on disk, it is stored as a regular Nachos file.
//
// The constructor initializes a directory structure in memory; the
// FetchFrom/WriteBack operations shuffle the directory information
// from/to disk.
class Directory {
  public:
    Directory(int size); // Initialize an empty directory
// with space for "size" files
    ~Directory();// De-allocate the directory
    void FetchFrom(OpenFile *file);  // Init directory contents from
disk
    void WriteBack(OpenFile *file);// Write modifications to
// directory contents back to disk
    int Find(char *name);// Find the sector number of the
// FileHeader for file: "name"
    bool Add(char *name, int newSector);  // Add a file name into the
directory
    bool Remove(char *name);// Remove a file from the directory
    void List();// Print the names of all the files
//  in the directory
    void Print();// Verbose print of the contents
//  of the directory -- all the file
//  names and their contents.
  private:
    int tableSize;// Number of directory entries
    DirectoryEntry *table;// Table of pairs:
// <file name, file header location>
    int FindIndex(char *name);// Find the index into the directory
//  table corresponding to "name"
};
#endif // DIRECTORY_H
```

```
// filehdr.cc
//Routines for managing the disk file header (in UNIX, this
//would be called the i-node).
//
//The file header is used to locate where on disk the
//file's data is stored.  We implement this as a fixed size
//table of pointers -- each entry in the table points to the
//disk sector containing that portion of the file data
//(in other words, there are no indirect or doubly indirect
//blocks). The table size is chosen so that the file header
//will be just big enough to fit in one disk sector,
//
//      Unlike in a real system, we do not keep track of file
permissions,
//ownership, last modification date, etc., in the file header.
//
//A file header can be initialized in two ways:
//   for a new file, by modifying the in-memory data structure
//     to point to the newly allocated data blocks
//   for a file already on disk, by reading the file header from disk
//
// 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.
#include "copyright.h"
#include "system.h"
#include "filehdr.h"
//----------------------------------------------------------------------
// FileHeader::Allocate
// Initialize a fresh file header for a newly created file.
//Allocate data blocks for the file out of the map of free disk blocks.
//Return FALSE if there are not enough free blocks to accomodate
//the new file.
//
//"freeMap" is the bit map of free disk sectors
//"fileSize" is the bit map of free disk sectors
//----------------------------------------------------------------------
bool
FileHeader::Allocate(BitMap *freeMap, int fileSize)
{
    numBytes = fileSize;
    numSectors  = divRoundUp(fileSize, SectorSize);
    if (freeMap->NumClear() < numSectors)
return FALSE;// not enough space
    for (int i = 0; i < numSectors; i++)
dataSectors[i] = freeMap->Find();
    return TRUE;
}
//----------------------------------------------------------------------
// FileHeader::Deallocate
// De-allocate all the space allocated for data blocks for this file.
//
//"freeMap" is the bit map of free disk sectors
//----------------------------------------------------------------------
void
FileHeader::Deallocate(BitMap *freeMap)
{
    for (int i = 0; i < numSectors; i++) {
ASSERT(freeMap->Test((int) dataSectors[i]));  // ought to be marked!
freeMap->Clear((int) dataSectors[i]);
    }
}
//----------------------------------------------------------------------
// FileHeader::FetchFrom
// Fetch contents of file header from disk.
//
//"sector" is the disk sector containing the file header
//----------------------------------------------------------------------
void
FileHeader::FetchFrom(int sector)
{
    synchDisk->ReadSector(sector, (char *)this);
}
//----------------------------------------------------------------------
// FileHeader::WriteBack
// Write the modified contents of the file header back to disk.
//
//"sector" is the disk sector to contain the file header
//----------------------------------------------------------------------
void
FileHeader::WriteBack(int sector)
{
    synchDisk->WriteSector(sector, (char *)this);
}
//----------------------------------------------------------------------
// FileHeader::ByteToSector
// Return which disk sector is storing a particular byte within the
file.
//      This is essentially a translation from a virtual address (the
//offset in the file) to a physical address (the sector where the
//data at the offset is stored).
//
//"offset" is the location within the file of the byte in question
//----------------------------------------------------------------------
int
FileHeader::ByteToSector(int offset)
{
    return(dataSectors[offset / SectorSize]);
}
//----------------------------------------------------------------------
// FileHeader::FileLength
// Return the number of bytes in the file.
//----------------------------------------------------------------------
int
FileHeader::FileLength()
{
    return numBytes;
}
//----------------------------------------------------------------------
// FileHeader::Print
// Print the contents of the file header, and the contents of all
```

```
//the data blocks pointed to by the file header.
//----------------------------------------------------------------------
void
FileHeader::Print()
{
    int i, j, k;
    char *data = new char[SectorSize];
    printf("FileHeader contents.  File size: %d.  File blocks:\n", 
numBytes);
    for (i = 0; i < numSectors; i++)
printf("%d ", dataSectors[i]);
    printf("\nFile contents:\n");
    for (i = k = 0; i < numSectors; i++) {
synchDisk->ReadSector(dataSectors[i], data);
        for (j = 0; (j < SectorSize) && (k < numBytes); j++, k++) {
    if ('\040' <= data[j] && data[j] <= '\176')   // isprint(data[j])
printf("%c", data[j]);
            else
printf("\\%x", (unsigned char)data[j]);
}
        printf("\n");
    }
    delete [] data;
}
```

```
// filehdr.h
//Data structures for managing a disk file header.
//
//A file header describes where on disk to find the data in a file,
//along with other information about the file (for instance, its
//length, owner, etc.)
//
// 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.
#include "copyright.h"
#ifndef FILEHDR_H
#define FILEHDR_H
#include "disk.h"
#include "bitmap.h"
#define NumDirect ((SectorSize - 2 * sizeof(int)) / sizeof(int))
#define MaxFileSize (NumDirect * SectorSize)
// The following class defines the Nachos "file header" (in UNIX terms,
// the "i-node"), describing where on disk to find all of the data in
the file.
// The file header is organized as a simple table of pointers to
// data blocks.
//
// The file header data structure can be stored in memory or on disk.
// When it is on disk, it is stored in a single sector -- this means
// that we assume the size of this data structure to be the same
// as one disk sector.  Without indirect addressing, this
// limits the maximum file length to just under 4K bytes.
//
// There is no constructor; rather the file header can be initialized
// by allocating blocks for the file (if it is a new file), or by
// reading it from disk.
class FileHeader {
  public:
    bool Allocate(BitMap *bitMap, int fileSize);// Initialize a file
header,
//  including allocating space
//  on disk for the file data
    void Deallocate(BitMap *bitMap);  // De-allocate this file's
//  data blocks
    void FetchFrom(int sectorNumber); // Initialize file header from
disk
    void WriteBack(int sectorNumber); // Write modifications to file
header
//  back to disk
    int ByteToSector(int offset);// Convert a byte offset into the file
// to the disk sector containing
// the byte
    int FileLength();// Return the length of the file
// in bytes
    void Print();// Print the contents of the file.
  private:
    int numBytes;// Number of bytes in the file
    int numSectors;// Number of data sectors in the file
    int dataSectors[NumDirect];// Disk sector numbers for each data
// block in the file
};
#endif // FILEHDR_H
```

```
// filesys.cc
//Routines to manage the overall operation of the file system.
//Implements routines to map from textual file names to files.
//
//Each file in the file system has:
//   A file header, stored in a sector on disk
//(the size of the file header data structure is arranged
//to be precisely the size of 1 disk sector)
//   A number of data blocks
//   An entry in the file system directory
//
// The file system consists of several data structures:
//   A bitmap of free disk sectors (cf. bitmap.h)
//   A directory of file names and file headers
//
//      Both the bitmap and the directory are represented as normal
//files.  Their file headers are located in specific sectors
//(sector 0 and sector 1), so that the file system can find them
//on bootup.
//
//The file system assumes that the bitmap and directory files are
//kept "open" continuously while Nachos is running.
//
//For those operations (such as Create, Remove) that modify the
//directory and/or bitmap, if the operation succeeds, the changes
//are written immediately back to disk (the two files are kept
//open during all this time).  If the operation fails, and we have
//modified part of the directory and/or bitmap, we simply discard
//the changed version, without writing it back to disk.
//
// Our implementation at this point has the following restrictions:
//
//   there is no synchronization for concurrent accesses
//   files have a fixed size, set when the file is created
//   files cannot be bigger than about 3KB in size
//   there is no hierarchical directory structure, and only a limited
//     number of files can be added to the system
//   there is no attempt to make the system robust to failures
//    (if Nachos exits in the middle of an operation that modifies
//    the file system, it may corrupt the disk)
//
// 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.
#include "copyright.h"
#include "disk.h"
#include "bitmap.h"
#include "directory.h"
#include "filehdr.h"
#include "filesys.h"
// Sectors containing the file headers for the bitmap of free sectors,
// and the directory of files.  These file headers are placed in well-
known
// sectors, so that they can be located on boot-up.
#define FreeMapSector 0
#define DirectorySector 1
// Initial file sizes for the bitmap and directory; until the file
system
// supports extensible files, the directory size sets the maximum
number
// of files that can be loaded onto the disk.
#define FreeMapFileSize (NumSectors / BitsInByte)
#define NumDirEntries 10
#define DirectoryFileSize (sizeof(DirectoryEntry) * NumDirEntries)
//----------------------------------------------------------------------
// FileSystem::FileSystem
// Initialize the file system.  If format = TRUE, the disk has
//nothing on it, and we need to initialize the disk to contain
//an empty directory, and a bitmap of free sectors (with almost but
//not all of the sectors marked as free).
//
//If format = FALSE, we just have to open the files
//representing the bitmap and the directory.
//
//"format" -- should we initialize the disk?
//----------------------------------------------------------------------
FileSystem::FileSystem(bool format)
{
    DEBUG('f', "Initializing the file system.\n");
    if (format) {
        BitMap *freeMap = new BitMap(NumSectors);
        Directory *directory = new Directory(NumDirEntries);
FileHeader *mapHdr = new FileHeader;
FileHeader *dirHdr = new FileHeader;
        DEBUG('f', "Formatting the file system.\n");
    // First, allocate space for FileHeaders for the directory and
bitmap
    // (make sure no one else grabs these!)
freeMap->Mark(FreeMapSector);
freeMap->Mark(DirectorySector);
    // Second, allocate space for the data blocks containing the
contents
    // of the directory and bitmap files.  There better be enough
space!
ASSERT(mapHdr->Allocate(freeMap, FreeMapFileSize));
ASSERT(dirHdr->Allocate(freeMap, DirectoryFileSize));
    // Flush the bitmap and directory FileHeaders back to disk
    // We need to do this before we can "Open" the file, since open
    // reads the file header off of disk (and currently the disk has
garbage
    // on it!).
        DEBUG('f', "Writing headers back to disk.\n");
mapHdr->WriteBack(FreeMapSector);
dirHdr->WriteBack(DirectorySector);
    // OK to open the bitmap and directory files now
    // The file system operations assume these two files are left open
    // while Nachos is running.
        freeMapFile = new OpenFile(FreeMapSector);
        directoryFile = new OpenFile(DirectorySector);

    // Once we have the files "open", we can write the initial version
    // of each file back to disk.  The directory at this point is
completely
    // empty; but the bitmap has been changed to reflect the fact that
    // sectors on the disk have been allocated for the file headers and
    // to hold the file data for the directory and bitmap.
        DEBUG('f', "Writing bitmap and directory back to disk.\n");
freeMap->WriteBack(freeMapFile); // flush changes to disk
directory->WriteBack(directoryFile);
if (DebugIsEnabled('f')) {
    freeMap->Print();
    directory->Print();
        delete freeMap;
delete directory;
delete mapHdr;
delete dirHdr;
}
    } else {
    // if we are not formatting the disk, just open the files
representing
    // the bitmap and directory; these are left open while Nachos is
running
        freeMapFile = new OpenFile(FreeMapSector);
        directoryFile = new OpenFile(DirectorySector);
    }
}
//-------------------------------------------------------------------
// FileSystem::Create
// Create a file in the Nachos file system (similar to UNIX create).
//Since we can't increase the size of files dynamically, we have
//to give Create the initial size of the file.
//
//The steps to create a file are:
//  Make sure the file doesn't already exist
//        Allocate a sector for the file header
//   Allocate space on disk for the data blocks for the file
//  Add the name to the directory
//  Store the new file header on disk
//  Flush the changes to the bitmap and the directory back to disk
//
//Return TRUE if everything goes ok, otherwise, return FALSE.
//
// Create fails if:
//   file is already in directory
// no free space for file header
// no free entry for file in directory
// no free space for data blocks for the file
//
// Note that this implementation assumes there is no concurrent access
//to the file system!
//
//"name" -- name of file to be created
//"initialSize" -- size of file to be created
//-------------------------------------------------------------------
bool
FileSystem::Create(char *name, int initialSize)
{
    Directory *directory;
    BitMap *freeMap;
    FileHeader *hdr;
    int sector;
    bool success;
    DEBUG('f', "Creating file %s, size %d\n", name, initialSize);
    directory = new Directory(NumDirEntries);
    directory->FetchFrom(directoryFile);
    if (directory->Find(name) != -1)
      success = FALSE;// file is already in directory
    else {
        freeMap = new BitMap(NumSectors);
        freeMap->FetchFrom(freeMapFile);
        sector = freeMap->Find();// find a sector to hold the file
header
    if (sector == -1)
            success = FALSE;// no free block for file header
        else if (!directory->Add(name, sector))
            success = FALSE;// no space in directory
else {
        hdr = new FileHeader;
    if (!hdr->Allocate(freeMap, initialSize))
            success = FALSE;// no space on disk for data
    else {
    success = TRUE;
// everthing worked, flush all changes back to disk
        hdr->WriteBack(sector);
        directory->WriteBack(directoryFile);
        freeMap->WriteBack(freeMapFile);
    }
            delete hdr;
}
        delete freeMap;
    }
    delete directory;
    return success;
}
//-------------------------------------------------------------------
// FileSystem::Open
// Open a file for reading and writing.
//To open a file:
//  Find the location of the file's header, using the directory
//  Bring the header into memory
//
//"name" -- the text name of the file to be opened
//-------------------------------------------------------------------
OpenFile *
FileSystem::Open(char *name)
{
    Directory *directory = new Directory(NumDirEntries);
    OpenFile *openFile = NULL;
    int sector;
    DEBUG('f', "Opening file %s\n", name);
    directory->FetchFrom(directoryFile);
    sector = directory->Find(name);
    if (sector >= 0)
```

```
openFile = new OpenFile(sector);// name was found in directory
    delete directory;
    return openFile;// return NULL if not found
}
//----------------------------------------------------------------------
// FileSystem::Remove
// Delete a file from the file system.  This requires:
//    Remove it from the directory
//    Delete the space for its header
//    Delete the space for its data blocks
//    Write changes to directory, bitmap back to disk
//
//Return TRUE if the file was deleted, FALSE if the file wasn't
//in the file system.
//
//"name" -- the text name of the file to be removed
//----------------------------------------------------------------------
bool
FileSystem::Remove(char *name)
{
    Directory *directory;
    BitMap *freeMap;
    FileHeader *fileHdr;
    int sector;

    directory = new Directory(NumDirEntries);
    directory->FetchFrom(directoryFile);
    sector = directory->Find(name);
    if (sector == -1) {
       delete directory;
       return FALSE; // file not found
    }
    fileHdr = new FileHeader;
    fileHdr->FetchFrom(sector);
    freeMap = new BitMap(NumSectors);
    freeMap->FetchFrom(freeMapFile);
    fileHdr->Deallocate(freeMap);  // remove data blocks
    freeMap->Clear(sector);// remove header block
    directory->Remove(name);
    freeMap->WriteBack(freeMapFile);// flush to disk
    directory->WriteBack(directoryFile);        // flush to disk
    delete fileHdr;
    delete directory;
    delete freeMap;
    return TRUE;
}
//----------------------------------------------------------------------
// FileSystem::List
// List all the files in the file system directory.
//----------------------------------------------------------------------
void
FileSystem::List()
{
    Directory *directory = new Directory(NumDirEntries);
    directory->FetchFrom(directoryFile);
    directory->List();
    delete directory;
}
//----------------------------------------------------------------------
// FileSystem::Print
// Print everything about the file system:
//  the contents of the bitmap
//  the contents of the directory
//  for each file in the directory,
//      the contents of the file header
//      the data in the file
//----------------------------------------------------------------------
void
FileSystem::Print()
{
    FileHeader *bitHdr = new FileHeader;
    FileHeader *dirHdr = new FileHeader;
    BitMap *freeMap = new BitMap(NumSectors);
    Directory *directory = new Directory(NumDirEntries);
    printf("Bit map file header:\n");
    bitHdr->FetchFrom(FreeMapSector);
    bitHdr->Print();
    printf("Directory file header:\n");
    dirHdr->FetchFrom(DirectorySector);
    dirHdr->Print();
    freeMap->FetchFrom(freeMapFile);
    freeMap->Print();
    directory->FetchFrom(directoryFile);
    directory->Print();
    delete bitHdr;
    delete dirHdr;
    delete freeMap;
    delete directory;
}
```

```
// filesys.h
//Data structures to represent the Nachos file system.
//
//A file system is a set of files stored on disk, organized
//into directories.  Operations on the file system have to
//do with "naming" -- creating, opening, and deleting files,
//given a textual file name.  Operations on an individual
//"open" file (read, write, close) are to be found in the OpenFile
//class (openfile.h).
//
//We define two separate implementations of the file system.
//The "STUB" version just re-defines the Nachos file system
//operations as operations on the native UNIX file system on the
machine
//running the Nachos simulation.  This is provided in case the
//multiprogramming and virtual memory assignments (which make use
//of the file system) are done before the file system assignment.
//
//The other version is a "real" file system, built on top of
//a disk simulator.  The disk is simulated using the native UNIX
//file system (in a file named "DISK").
//
//In the "real" implementation, there are two key data structures used
//in the file system.  There is a single "root" directory, listing
//all of the files in the file system; unlike UNIX, the baseline
//system does not provide a hierarchical directory structure.
//In addition, there is a bitmap for allocating
//disk sectors.  Both the root directory and the bitmap are themselves
//stored as files in the Nachos file system -- this causes an
interesting
//bootstrap problem when the simulated disk is initialized.
//
// 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 FS_H
#define FS_H
#include "copyright.h"
#include "openfile.h"
#ifdef FILESYS_STUB // Temporarily implement file system calls as
// calls to UNIX, until the real file system
// implementation is available
class FileSystem {
  public:
    FileSystem(bool format) {}
    bool Create(char *name, int initialSize) {
int fileDescriptor = OpenForWrite(name);
if (fileDescriptor == -1) return FALSE;
Close(fileDescriptor);
return TRUE;
}
    OpenFile* Open(char *name) {
  int fileDescriptor = OpenForReadWrite(name, FALSE);
  if (fileDescriptor == -1) return NULL;
  return new OpenFile(fileDescriptor);
      }
    bool Remove(char *name) { return Unlink(name) == 0; }
};
#else // FILESYS
class FileSystem {
  public:
    FileSystem(bool format);// Initialize the file system.
// Must be called *after* "synchDisk"
// has been initialized.
    // If "format", there is nothing on
// the disk, so initialize the directory
    // and the bitmap of free blocks.
    bool Create(char *name, int initialSize);
// Create a file (UNIX creat)
    OpenFile* Open(char *name); // Open a file (UNIX open)
    bool Remove(char *name);  // Delete a file (UNIX unlink)
    void List();// List all the files in the file system
    void Print();// List all the files and their contents
  private:
   OpenFile* freeMapFile;// Bit map of free disk blocks,
// represented as a file
   OpenFile* directoryFile;// "Root" directory -- list of
// file names, represented as a file
};
#endif // FILESYS
#endif // FS_H
```

```
// fstest.cc
//Simple test routines for the file system.
//
//We implement:
//   Copy -- copy a file from UNIX to Nachos
//   Print -- cat the contents of a Nachos file
//   Perftest -- a stress test for the Nachos file system
//read and write a really large file in tiny chunks
//(won't work on baseline system!)
//
// 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.
#include "copyright.h"
#include "utility.h"
#include "filesys.h"
#include "system.h"
#include "thread.h"
#include "disk.h"
#include "stats.h"
#define TransferSize 10 // make it small, just to be difficult
//-------------------------------------------------------------------
// Copy
// Copy the contents of the UNIX file "from" to the Nachos file "to"
//-------------------------------------------------------------------
void
Copy(char *from, char *to)
{
    FILE *fp;
    OpenFile* openFile;
    int amountRead, fileLength;
    char *buffer;
// Open UNIX file
    if ((fp = fopen(from, "r")) == NULL) {
printf("Copy: couldn't open input file %s\n", from);
return;
    }
// Figure out length of UNIX file
    fseek(fp, 0, 2);
    fileLength = ftell(fp);
    fseek(fp, 0, 0);
// Create a Nachos file of the same length
    DEBUG('f', "Copying file %s, size %d, to file %s\n", from,
fileLength, to);
    if (!fileSystem->Create(to, fileLength)) { // Create Nachos file
printf("Copy: couldn't create output file %s\n", to);
fclose(fp);
return;
    }

    openFile = fileSystem->Open(to);
    ASSERT(openFile != NULL);

// Copy the data in TransferSize chunks
    buffer = new char[TransferSize];
    while ((amountRead = fread(buffer, sizeof(char), TransferSize,
fp)) > 0)
openFile->Write(buffer, amountRead);
    delete [] buffer;
// Close the UNIX and the Nachos files
    delete openFile;
    fclose(fp);
}
//-------------------------------------------------------------------
// Print
// Print the contents of the Nachos file "name".
//-------------------------------------------------------------------
void
Print(char *name)
{
    OpenFile *openFile;
    int i, amountRead;
    char *buffer;
    if ((openFile = fileSystem->Open(name)) == NULL) {
printf("Print: unable to open file %s\n", name);
return;
    }

    buffer = new char[TransferSize];
    while ((amountRead = openFile->Read(buffer, TransferSize)) > 0)
for (i = 0; i < amountRead; i++)
    printf("%c", buffer[i]);
    delete [] buffer;
    delete openFile;// close the Nachos file
    return;
}
//-------------------------------------------------------------------
// PerformanceTest
// Stress the Nachos file system by creating a large file, writing
//it out a bit at a time, reading it back a bit at a time, and then
//deleting the file.
//
//Implemented as three separate routines:
//  FileWrite -- write the file
//  FileRead -- read the file
//  PerformanceTest -- overall control, and print out performance #'s
//-------------------------------------------------------------------
#define FileName "TestFile"
#define Contents "1234567890"
#define ContentSize strlen(Contents)
#define FileSize ((int)(ContentSize * 5000))
static void
FileWrite()
{
    OpenFile *openFile;
    int i, numBytes;
    printf("Sequential write of %d byte file, in %d byte chunks\n",
FileSize, ContentSize);
    if (!fileSystem->Create(FileName, 0)) {
      printf("Perf test: can't create %s\n", FileName);
      return;
    }
    openFile = fileSystem->Open(FileName);
    if (openFile == NULL) {
printf("Perf test: unable to open %s\n", FileName);
return;
    }
    for (i = 0; i < FileSize; i += ContentSize) {
        numBytes = openFile->Write(Contents, ContentSize);
if (numBytes < 10) {
    printf("Perf test: unable to write %s\n", FileName);
    delete openFile;
    return;
}
    }
    delete openFile;// close file
}
static void
FileRead()
{
    OpenFile *openFile;
    char *buffer = new char[ContentSize];
    int i, numBytes;
    printf("Sequential read of %d byte file, in %d byte chunks\n",
FileSize, ContentSize);
    if ((openFile = fileSystem->Open(FileName)) == NULL) {
printf("Perf test: unable to open file %s\n", FileName);
delete [] buffer;
return;
    }
    for (i = 0; i < FileSize; i += ContentSize) {
        numBytes = openFile->Read(buffer, ContentSize);
if ((numBytes < 10) || strncmp(buffer, Contents, ContentSize)) {
    printf("Perf test: unable to read %s\n", FileName);
    delete openFile;
    delete [] buffer;
    return;
}
    }
    delete [] buffer;
    delete openFile;// close file
}
void
PerformanceTest()
{
    printf("Starting file system performance test:\n");
    stats->Print();
    FileWrite();
    FileRead();
    if (!fileSystem->Remove(FileName)) {
      printf("Perf test: unable to remove %s\n", FileName);
      return;
    }
    stats->Print();
}
```

```
// openfile.cc
//Routines to manage an open Nachos file.  As in UNIX, a
//file must be open before we can read or write to it.
//Once we're all done, we can close it (in Nachos, by deleting
//the OpenFile data structure).
//
//Also as in UNIX, for convenience, we keep the file header in
//memory while the file is open.
//
// 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.
#include "copyright.h"
#include "filehdr.h"
#include "openfile.h"
#include "system.h"
#ifdef HOST_SPARC
#include <strings.h>
#endif
//----------------------------------------------------------------------
// OpenFile::OpenFile
// Open a Nachos file for reading and writing.  Bring the file header
//into memory while the file is open.
//
//"sector" -- the location on disk of the file header for this file
//----------------------------------------------------------------------
OpenFile::OpenFile(int sector)
{
    hdr = new FileHeader;
    hdr->FetchFrom(sector);
    seekPosition = 0;
}
//----------------------------------------------------------------------
// OpenFile::~OpenFile
// Close a Nachos file, de-allocating any in-memory data structures.
//----------------------------------------------------------------------
OpenFile::~OpenFile()
{
    delete hdr;
}
//----------------------------------------------------------------------
// OpenFile::Seek
// Change the current location within the open file -- the point at
//which the next Read or Write will start from.
//
//"position" -- the location within the file for the next Read/Write
//----------------------------------------------------------------------
void
OpenFile::Seek(int position)
{
    seekPosition = position;
}
//----------------------------------------------------------------------
// OpenFile::Read/Write
// Read/write a portion of a file, starting from seekPosition.
//Return the number of bytes actually written or read, and as a
//side effect, increment the current position within the file.
//
//Implemented using the more primitive ReadAt/WriteAt.
//
//"into" -- the buffer to contain the data to be read from disk
//"from" -- the buffer containing the data to be written to disk
//"numBytes" -- the number of bytes to transfer
//----------------------------------------------------------------------
int
OpenFile::Read(char *into, int numBytes)
{
   int result = ReadAt(into, numBytes, seekPosition);
   seekPosition += result;
   return result;
}
int
OpenFile::Write(char *into, int numBytes)
{
   int result = WriteAt(into, numBytes, seekPosition);
   seekPosition += result;
   return result;
}
//----------------------------------------------------------------------
// OpenFile::ReadAt/WriteAt
// Read/write a portion of a file, starting at "position".
//Return the number of bytes actually written or read, but has
//no side effects (except that Write modifies the file, of course).
//
//There is no guarantee the request starts or ends on an even disk
sector
//boundary; however the disk only knows how to read/write a whole disk
//sector at a time.  Thus:
//
//For ReadAt:
//   We read in all of the full or partial sectors that are part of
the
//   request, but we only copy the part we are interested in.
//For WriteAt:
//   We must first read in any sectors that will be partially written,
//   so that we don't overwrite the unmodified portion.  We then copy
//   in the data that will be modified, and write back all the full
//   or partial sectors that are part of the request.
//
//"into" -- the buffer to contain the data to be read from disk
//"from" -- the buffer containing the data to be written to disk
//"numBytes" -- the number of bytes to transfer
//"position" -- the offset within the file of the first byte to be
//read/written
//----------------------------------------------------------------------
int
OpenFile::ReadAt(char *into, int numBytes, int position)
{
    int fileLength = hdr->FileLength();
    int i, firstSector, lastSector, numSectors;
    char *buf;
    if ((numBytes <= 0) || (position >= fileLength))
    return 0; // check request
    if ((position + numBytes) > fileLength)
numBytes = fileLength - position;
    DEBUG('f', "Reading %d bytes at %d, from file of length %d.\n",
numBytes, position, fileLength);
    firstSector = divRoundDown(position, SectorSize);
    lastSector = divRoundDown(position + numBytes - 1, SectorSize);
    numSectors = 1 + lastSector - firstSector;
    // read in all the full and partial sectors that we need
    buf = new char[numSectors * SectorSize];
    for (i = firstSector; i <= lastSector; i++)
        synchDisk->ReadSector(hdr->ByteToSector(i * SectorSize),
&buf[(i - firstSector) * SectorSize]);
    // copy the part we want
    bcopy(&buf[position - (firstSector * SectorSize)], into, numBytes);
    delete [] buf;
    return numBytes;
}
int
OpenFile::WriteAt(char *from, int numBytes, int position)
{
    int fileLength = hdr->FileLength();
    int i, firstSector, lastSector, numSectors;
    bool firstAligned, lastAligned;
    char *buf;
    if ((numBytes <= 0) || (position >= fileLength))
return 0;// check request
    if ((position + numBytes) > fileLength)
numBytes = fileLength - position;
    DEBUG('f', "Writing %d bytes at %d, from file of length %d.\n",
numBytes, position, fileLength);
    firstSector = divRoundDown(position, SectorSize);
    lastSector = divRoundDown(position + numBytes - 1, SectorSize);
    numSectors = 1 + lastSector - firstSector;
    buf = new char[numSectors * SectorSize];
    firstAligned = (position == (firstSector * SectorSize));
    lastAligned = ((position + numBytes) == ((lastSector + 1) *
SectorSize));
// read in first and last sector, if they are to be partially modified
    if (!firstAligned)
        ReadAt(buf, SectorSize, firstSector * SectorSize);
    if (!lastAligned && ((firstSector != lastSector) || firstAligned))
        ReadAt(&buf[(lastSector - firstSector) * SectorSize],
SectorSize, lastSector * SectorSize);
// copy in the bytes we want to change
    bcopy(from, &buf[position - (firstSector * SectorSize)], numBytes);
// write modified sectors back
    for (i = firstSector; i <= lastSector; i++)
        synchDisk->WriteSector(hdr->ByteToSector(i * SectorSize),
&buf[(i - firstSector) * SectorSize]);
    delete [] buf;
    return numBytes;
}
//------------------------------------------------------------------
// OpenFile::Length
// Return the number of bytes in the file.
//------------------------------------------------------------------
int
OpenFile::Length()
{
    return hdr->FileLength();
}
```

```
// openfile.h
//Data structures for opening, closing, reading and writing to
//individual files.  The operations supported are similar to
//the UNIX ones -- type 'man open' to the UNIX prompt.
//
//There are two implementations.  One is a "STUB" that directly
//turns the file operations into the underlying UNIX operations.
//(cf. comment in filesys.h).
//
//The other is the "real" implementation, that turns these
//operations into read and write disk sector requests.
//In this baseline implementation of the file system, we don't
//worry about concurrent accesses to the file system
//by different threads -- this is part of the assignment.
//
// 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 OPENFILE_H
#define OPENFILE_H
#include "copyright.h"
#include "utility.h"
#ifdef FILESYS_STUB// Temporarily implement calls to
// Nachos file system as calls to UNIX!
// See definitions listed under #else
class OpenFile {
  public:
    OpenFile(int f) { file = f; currentOffset = 0; }// open the file
    ~OpenFile() { Close(file); }// close the file
    int ReadAt(char *into, int numBytes, int position) {
    Lseek(file, position, 0);
return ReadPartial(file, into, numBytes);
}
    int WriteAt(char *from, int numBytes, int position) {
    Lseek(file, position, 0);
WriteFile(file, from, numBytes);
return numBytes;
}
    int Read(char *into, int numBytes) {
int numRead = ReadAt(into, numBytes, currentOffset);
currentOffset += numRead;
return numRead;
    }
    int Write(char *from, int numBytes) {
int numWritten = WriteAt(from, numBytes, currentOffset);
currentOffset += numWritten;
return numWritten;
}
    int Length() { Lseek(file, 0, 2); return Tell(file); }

  private:
    int file;
    int currentOffset;
};
#else // FILESYS
class FileHeader;
class OpenFile {
  public:
    OpenFile(int sector);// Open a file whose header is located
// at "sector" on the disk
    ~OpenFile();// Close the file
    void Seek(int position); // Set the position from which to
// start reading/writing -- UNIX lseek
    int Read(char *into, int numBytes); // Read/write bytes from the
file,
// starting at the implicit position.
// Return the # actually read/written,
// and increment position in file.
    int Write(char *from, int numBytes);
    int ReadAt(char *into, int numBytes, int position);
    // Read/write bytes from the file,
// bypassing the implicit position.
    int WriteAt(char *from, int numBytes, int position);
    int Length(); // Return the number of bytes in the
// file (this interface is simpler
// than the UNIX idiom -- lseek to
// end of file, tell, lseek back

  private:
    FileHeader *hdr;// Header for this file
    int seekPosition;// Current position within the file
};
#endif // FILESYS
#endif // OPENFILE_H
```

```
// synchdisk.cc
//Routines to synchronously access the disk.  The physical disk
//is an asynchronous device (disk requests return immediately, and
//an interrupt happens later on).  This is a layer on top of
//the disk providing a synchronous interface (requests wait until
//the request completes).
//
//Use a semaphore to synchronize the interrupt handlers with the
//pending requests.  And, because the physical disk can only
//handle one operation at a time, use a lock to enforce mutual
//exclusion.
#include "copyright.h"
#include "synchdisk.h"
//----------------------------------------------------------------------
// DiskRequestDone
// Disk interrupt handler.  Need this to be a C routine, because
//C++ can't handle pointers to member functions.
//----------------------------------------------------------------------
static void
DiskRequestDone (int arg)
{
    SynchDisk* disk = (SynchDisk *)arg;
    disk->RequestDone();
}
//----------------------------------------------------------------------
// SynchDisk::SynchDisk
// Initialize the synchronous interface to the physical disk, in turn
//initializing the physical disk.
//
//"name" -- UNIX file name to be used as storage for the disk data
//   (usually, "DISK")
//----------------------------------------------------------------------
SynchDisk::SynchDisk(char* name)
{
    semaphore = new Semaphore("synch disk", 0);
    lock = new Lock("synch disk lock");
    disk = new Disk(name, DiskRequestDone, (int) this);
}
//----------------------------------------------------------------------
// SynchDisk::~SynchDisk
// De-allocate data structures needed for the synchronous disk
//abstraction.
//----------------------------------------------------------------------
SynchDisk::~SynchDisk()
{
    delete disk;
    delete lock;
    delete semaphore;
}
//----------------------------------------------------------------------
// SynchDisk::ReadSector
// Read the contents of a disk sector into a buffer.  Return only
//after the data has been read.
//
//"sectorNumber" -- the disk sector to read
//"data" -- the buffer to hold the contents of the disk sector
//----------------------------------------------------------------------
void
SynchDisk::ReadSector(int sectorNumber, char* data)
{
    lock->Acquire();// only one disk I/O at a time
    disk->ReadRequest(sectorNumber, data);
    semaphore->P();// wait for interrupt
    lock->Release();
}
//----------------------------------------------------------------------
// SynchDisk::WriteSector
// Write the contents of a buffer into a disk sector.  Return only
//after the data has been written.
//
//"sectorNumber" -- the disk sector to be written
//"data" -- the new contents of the disk sector
//----------------------------------------------------------------------
void
SynchDisk::WriteSector(int sectorNumber, char* data)
{
    lock->Acquire();// only one disk I/O at a time
    disk->WriteRequest(sectorNumber, data);
    semaphore->P();// wait for interrupt
    lock->Release();
}
//----------------------------------------------------------------------
// SynchDisk::RequestDone
// Disk interrupt handler.  Wake up any thread waiting for the disk
//request to finish.
//----------------------------------------------------------------------
void
SynchDisk::RequestDone()
{
    semaphore->V();
}
```

```
// synchdisk.h
// Data structures to export a synchronous interface to the raw
//disk device.
//
// 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.
#include "copyright.h"
#ifndef SYNCHDISK_H
#define SYNCHDISK_H
#include "disk.h"
#include "synch.h"
// The following class defines a "synchronous" disk abstraction.
// 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 occurs later to signal that the operation
completed.
// (Also, the physical characteristics of the disk device assume that
// only one operation can be requested at a time).
//
// This class provides the abstraction that for any individual thread
// making a request, it waits around until the operation finishes
before
// returning.
class SynchDisk {
  public:
    SynchDisk(char* name);    // Initialize a synchronous disk,
// by initializing the raw Disk.
    ~SynchDisk();// De-allocate the synch disk data

    void ReadSector(int sectorNumber, char* data);
    // Read/write a disk sector, returning
    // only once the data is actually read
// or written.  These call
    // Disk::ReadRequest/WriteRequest and
// then wait until the request is done.
    void WriteSector(int sectorNumber, char* data);

    void RequestDone();// Called by the disk device interrupt
// handler, to signal that the
// current disk operation is complete.
  private:
    Disk *disk;  // Raw disk device
    Semaphore *semaphore; // To synchronize requesting thread
// with the interrupt handler
    Lock *lock;  // Only one read/write request
// can be sent to the disk at a time
};
#endif // SYNCHDISK_H
```

```
// addrspace.cc
//Routines to manage address spaces (executing user programs).
//
//In order to run a user program, you must:
//
//1. link with the -N -T 0 option
//2. run coff2noff to convert the object file to Nachos format
//(Nachos object code format is essentially just a simpler
//version of the UNIX executable object code format)
//3. load the NOFF file into the Nachos file system
//(if you haven't implemented the file system yet, you
//don't need to do this last step)
//
// 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.
#include "copyright.h"
#include "system.h"
#include "addrspace.h"
#include "noff.h"
#ifdef HOST_SPARC
#include <strings.h>
#endif
//-------------------------------------------------------------------
// SwapHeader
// Do little endian to big endian conversion on the bytes in the
//object file header, in case the file was generated on a little
//endian machine, and we're now running on a big endian machine.
//-------------------------------------------------------------------
static void
SwapHeader (NoffHeader *noffH)
{
noffH->noffMagic = WordToHost(noffH->noffMagic);
noffH->code.size = WordToHost(noffH->code.size);
noffH->code.virtualAddr = WordToHost(noffH->code.virtualAddr);
noffH->code.inFileAddr = WordToHost(noffH->code.inFileAddr);
noffH->initData.size = WordToHost(noffH->initData.size);
noffH->initData.virtualAddr = WordToHost(noffH->initData.virtualAddr);
noffH->initData.inFileAddr = WordToHost(noffH->initData.inFileAddr);
noffH->uninitData.size = WordToHost(noffH->uninitData.size);
noffH->uninitData.virtualAddr = WordToHost(noffH-
>uninitData.virtualAddr);
noffH->uninitData.inFileAddr = WordToHost(noffH-
>uninitData.inFileAddr);
}
//-------------------------------------------------------------------
// AddrSpace::AddrSpace
// Create an address space to run a user program.
//Load the program from a file "executable", and set everything
//up so that we can start executing user instructions.
//
//Assumes that the object code file is in NOFF format.
//
//First, set up the translation from program memory to physical
//memory.  For now, this is really simple (1:1), since we are
//only uniprogramming, and we have a single unsegmented page table
//
//"executable" is the file containing the object code to load into
memory
//-------------------------------------------------------------------
AddrSpace::AddrSpace(OpenFile *executable)
{
    NoffHeader noffH;
    unsigned int i, size;
    executable->ReadAt((char *)&noffH, sizeof(noffH), 0);
    if ((noffH.noffMagic != NOFFMAGIC) &&
(WordToHost(noffH.noffMagic) == NOFFMAGIC))
    SwapHeader(&noffH);
    ASSERT(noffH.noffMagic == NOFFMAGIC);
// how big is address space?
    size = noffH.code.size + noffH.initData.size +
noffH.uninitData.size
+ UserStackSize;// we need to increase the size
// to leave room for the stack
    numPages = divRoundUp(size, PageSize);
    size = numPages * PageSize;
    ASSERT(numPages <= NumPhysPages);// check we're not trying
// to run anything too big --
// at least until we have
// virtual memory
    DEBUG('a', "Initializing address space, num pages %d, size %d\n",
numPages, size);
// first, set up the translation
    pageTable = new TranslationEntry[numPages];
    for (i = 0; i < numPages; i++) {
pageTable[i].virtualPage = i;// for now, virtual page # = phys page #
pageTable[i].physicalPage = i;
pageTable[i].valid = TRUE;
pageTable[i].use = FALSE;
pageTable[i].dirty = FALSE;
pageTable[i].readOnly = FALSE;  // if the code segment was entirely on
// a separate page, we could set its
// pages to be read-only
    }

// zero out the entire address space, to zero the unitialized data
segment
// and the stack segment
    bzero(machine->mainMemory, size);
// then, copy in the code and data segments into memory
    if (noffH.code.size > 0) {
        DEBUG('a', "Initializing code segment, at 0x%x, size %d\n",
noffH.code.virtualAddr, noffH.code.size);
        executable->ReadAt(&(machine-
>mainMemory[noffH.code.virtualAddr]),
noffH.code.size, noffH.code.inFileAddr);
    }
    if (noffH.initData.size > 0) {
        DEBUG('a', "Initializing data segment, at 0x%x, size %d\n",
noffH.initData.virtualAddr, noffH.initData.size);
        executable->ReadAt(&(machine-
>mainMemory[noffH.initData.virtualAddr]),
noffH.initData.size, noffH.initData.inFileAddr);
    }
}
//-----------------------------------------------------------------
// AddrSpace::~AddrSpace
// Dealloate an address space.  Nothing for now!
//-----------------------------------------------------------------
AddrSpace::~AddrSpace()
{
   delete pageTable;
}
//-----------------------------------------------------------------
// AddrSpace::InitRegisters
// Set the initial values for the user-level register set.
//
// We write these directly into the "machine" registers, so
//that we can immediately jump to user code.  Note that these
//will be saved/restored into the currentThread->userRegisters
//when this thread is context switched out.
//-----------------------------------------------------------------
void
AddrSpace::InitRegisters()
{
    int i;
    for (i = 0; i < NumTotalRegs; i++)
machine->WriteRegister(i, 0);
    // Initial program counter -- must be location of "Start"
    machine->WriteRegister(PCReg, 0);
    // Need to also tell MIPS where next instruction is, because
    // of branch delay possibility
    machine->WriteRegister(NextPCReg, 4);
   // Set the stack register to the end of the address space, where we
   // allocated the stack; but subtract off a bit, to make sure we
don't
   // accidentally reference off the end!
    machine->WriteRegister(StackReg, numPages * PageSize - 16);
    DEBUG('a', "Initializing stack register to %d\n", numPages *
PageSize - 16);
}
//-----------------------------------------------------------------
// AddrSpace::SaveState
// On a context switch, save any machine state, specific
//to this address space, that needs saving.
//
//For now, nothing!
//-----------------------------------------------------------------
void AddrSpace::SaveState()
{}
//-----------------------------------------------------------------
// AddrSpace::RestoreState
// On a context switch, restore the machine state so that
//this address space can run.
//
//      For now, tell the machine where to find the page table.
//-----------------------------------------------------------------
void AddrSpace::RestoreState()
{
    machine->pageTable = pageTable;
    machine->pageTableSize = numPages;
}
```

```
// addrspace.h
//Data structures to keep track of executing user programs
//(address spaces).
//
//For now, we don't keep any information about address spaces.
//The user level CPU state is saved and restored in the thread
//executing the user program (see thread.h).
//
// 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 ADDRSPACE_H
#define ADDRSPACE_H
#include "copyright.h"
#include "filesys.h"
#define UserStackSize1024 // increase this as necessary!
class AddrSpace {
  public:
    AddrSpace(OpenFile *executable);// Create an address space,
// initializing it with the program
// stored in the file "executable"
    ~AddrSpace();// De-allocate an address space
    void InitRegisters();// Initialize user-level CPU registers,
// before jumping to user code
    void SaveState();// Save/restore address space-specific
    void RestoreState();// info on a context switch
  private:
    TranslationEntry *pageTable;// Assume linear page table translation
// for now!
    unsigned int numPages;// Number of pages in the virtual
// address space
};
#endif // ADDRSPACE_H
```

```
// bitmap.c
//Routines to manage a bitmap -- an array of bits each of which
//can be either on or off.  Represented as an array of integers.
//
// 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.
#include "copyright.h"
#include "bitmap.h"
//----------------------------------------------------------------------
// BitMap::BitMap
// Initialize a bitmap with "nitems" bits, so that every bit is clear.
//it can be added somewhere on a list.
//
//"nitems" is the number of bits in the bitmap.
//----------------------------------------------------------------------
BitMap::BitMap(int nitems)
{
    numBits = nitems;
    numWords = divRoundUp(numBits, BitsInWord);
    map = new unsigned int[numWords];
    for (int i = 0; i < numBits; i++)
        Clear(i);
}
//----------------------------------------------------------------------
// BitMap::~BitMap
// De-allocate a bitmap.
//----------------------------------------------------------------------
BitMap::~BitMap()
{
    delete map;
}
//----------------------------------------------------------------------
// BitMap::Set
// Set the "nth" bit in a bitmap.
//
//"which" is the number of the bit to be set.
//----------------------------------------------------------------------
void
BitMap::Mark(int which)
{
    ASSERT(which >= 0 && which < numBits);
    map[which / BitsInWord] |= 1 << (which % BitsInWord);
}

//----------------------------------------------------------------------
// BitMap::Clear
// Clear the "nth" bit in a bitmap.
//
//"which" is the number of the bit to be cleared.
//----------------------------------------------------------------------
void
BitMap::Clear(int which)
{
    ASSERT(which >= 0 && which < numBits);
    map[which / BitsInWord] &= ~(1 << (which % BitsInWord));
}
//----------------------------------------------------------------------
// BitMap::Test
// Return TRUE if the "nth" bit is set.
//
//"which" is the number of the bit to be tested.
//----------------------------------------------------------------------
bool
BitMap::Test(int which)
{
    ASSERT(which >= 0 && which < numBits);

    if (map[which / BitsInWord] & (1 << (which % BitsInWord)))
return TRUE;
    else
return FALSE;
}
//----------------------------------------------------------------------
// BitMap::Find
// Return the number of the first bit which is clear.
//As a side effect, set the bit (mark it as in use).
//(In other words, find and allocate a bit.)
//
//If no bits are clear, return -1.
//----------------------------------------------------------------------
int
BitMap::Find()
{
    for (int i = 0; i < numBits; i++)
if (!Test(i)) {
    Mark(i);
    return i;
}
    return -1;
}
//----------------------------------------------------------------------
// BitMap::NumClear
// Return the number of clear bits in the bitmap.
//(In other words, how many bits are unallocated?)
//----------------------------------------------------------------------
int
BitMap::NumClear()
{
    int count = 0;
    for (int i = 0; i < numBits; i++)
if (!Test(i)) count++;
    return count;
}
//----------------------------------------------------------------------
// BitMap::Print
// Print the contents of the bitmap, for debugging.
//
//Could be done in a number of ways, but we just print the #'s of
//all the bits that are set in the bitmap.
//----------------------------------------------------------------------
```

```
void
BitMap::Print()
{
    printf("Bitmap set:\n");
    for (int i = 0; i < numBits; i++)
if (Test(i))
    printf("%d, ", i);
    printf("\n");
}
// These aren't needed until the FILESYS assignment
//-------------------------------------------------------------------
// BitMap::FetchFromFile
// Initialize the contents of a bitmap from a Nachos file.
//
//"file" is the place to read the bitmap from
//-------------------------------------------------------------------
void
BitMap::FetchFrom(OpenFile *file)
{
    file->ReadAt((char *)map, numWords * sizeof(unsigned), 0);
}
//-------------------------------------------------------------------
// BitMap::WriteBack
// Store the contents of a bitmap to a Nachos file.
//
//"file" is the place to write the bitmap to
//-------------------------------------------------------------------
void
BitMap::WriteBack(OpenFile *file)
{
   file->WriteAt((char *)map, numWords * sizeof(unsigned), 0);
}
```

```
// bitmap.h
//Data structures defining a bitmap -- an array of bits each of which
//can be either on or off.
//
//Represented as an array of unsigned integers, on which we do
//modulo arithmetic to find the bit we are interested in.
//
//The bitmap can be parameterized with with the number of bits being
//managed.
//
// 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 BITMAP_H
#define BITMAP_H
#include "copyright.h"
#include "utility.h"
#include "openfile.h"
// Definitions helpful for representing a bitmap as an array of
integers
#define BitsInByte 8
#define BitsInWord 32
// The following class defines a "bitmap" -- an array of bits,
// each of which can be independently set, cleared, and tested.
//
// Most useful for managing the allocation of the elements of an array
--
// for instance, disk sectors, or main memory pages.
// Each bit represents whether the corresponding sector or page is
// in use or free.
class BitMap {
  public:
    BitMap(int nitems);// Initialize a bitmap, with "nitems" bits
// initially, all bits are cleared.
    ~BitMap();// De-allocate bitmap

    void Mark(int which);   // Set the "nth" bit
    void Clear(int which);  // Clear the "nth" bit
    bool Test(int which);   // Is the "nth" bit set?
    int Find();            // Return the # of a clear bit, and as a
side
// effect, set the bit.
// If no bits are clear, return -1.
    int NumClear();// Return the number of clear bits
    void Print();// Print contents of bitmap

    // These aren't needed until FILESYS, when we will need to read and
    // write the bitmap to a file
    void FetchFrom(OpenFile *file); // fetch contents from disk
    void WriteBack(OpenFile *file); // write contents to disk
  private:
    int numBits;// number of bits in the bitmap
    int numWords;// number of words of bitmap storage
// (rounded up if numBits is not a
//  multiple of the number of bits in
//  a word)
    unsigned int *map;// bit storage
};
#endif // BITMAP_H
```

```
// exception.cc
//Entry point into the Nachos kernel from user programs.
//There are two kinds of things that can cause control to
//transfer back to here from user code:
//
//syscall -- The user code explicitly requests to call a procedure
//in the Nachos kernel.  Right now, the only function we support is
//"Halt".
//
//exceptions -- The user code does something that the CPU can't handle.
//For instance, accessing memory that doesn't exist, arithmetic errors,
//etc.
//
//Interrupts (which can also cause control to transfer from user
//code into the Nachos kernel) are handled elsewhere.
//
// For now, this only handles the Halt() system call.
// Everything else core dumps.
//
// 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.
#include "copyright.h"
#include "system.h"
#include "syscall.h"
//----------------------------------------------------------------------
// ExceptionHandler
// Entry point into the Nachos kernel.  Called when a user program
//is executing, and either does a syscall, or generates an addressing
//or arithmetic exception.
//
// For system calls, the following is the calling convention:
//
// system call code -- r2
//arg1 -- r4
//arg2 -- r5
//arg3 -- r6
//arg4 -- r7
//
//The result of the system call, if any, must be put back into r2.
//
// And don't forget to increment the pc before returning. (Or else
you'll
// loop making the same system call forever!
//
//"which" is the kind of exception.  The list of possible exceptions
//are in machine.h.
//----------------------------------------------------------------------
void
ExceptionHandler(ExceptionType which)
{
    int type = machine->ReadRegister(2);
    if ((which == SyscallException) && (type == SC_Halt)) {
DEBUG('a', "Shutdown, initiated by user program.\n");
   interrupt->Halt();
    } else {
printf("Unexpected user mode exception %d %d\n", which, type);
ASSERT(FALSE);
    }
}
```

```
// progtest.cc
//Test routines for demonstrating that Nachos can load
//a user program and execute it.
//
//Also, routines for testing the Console hardware device.
//
// 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.
#include "copyright.h"
#include "system.h"
#include "console.h"
#include "addrspace.h"
#include "synch.h"
//----------------------------------------------------------------------
// StartProcess
// Run a user program.  Open the executable, load it into
//memory, and jump to it.
//----------------------------------------------------------------------
void
StartProcess(char *filename)
{
    OpenFile *executable = fileSystem->Open(filename);
    AddrSpace *space;
    if (executable == NULL) {
printf("Unable to open file %s\n", filename);
return;
    }
    space = new AddrSpace(executable);
    currentThread->space = space;
    delete executable;// close file
    space->InitRegisters();// set the initial register values
    space->RestoreState();// load page table register
    machine->Run();// jump to the user progam
    ASSERT(FALSE);// machine->Run never returns;
// the address space exits
// by doing the syscall "exit"
}
// Data structures needed for the console test.  Threads making
// I/O requests wait on a Semaphore to delay until the I/O completes.
static Console *console;
static Semaphore *readAvail;
static Semaphore *writeDone;
//----------------------------------------------------------------------
// ConsoleInterruptHandlers
// Wake up the thread that requested the I/O.
//----------------------------------------------------------------------
static void ReadAvail(int arg) { readAvail->V(); }
static void WriteDone(int arg) { writeDone->V(); }
//----------------------------------------------------------------------
// ConsoleTest
// Test the console by echoing characters typed at the input onto
//the output.  Stop when the user types a 'q'.
//----------------------------------------------------------------------
void
ConsoleTest (char *in, char *out)
{
    char ch;
    console = new Console(in, out, ReadAvail, WriteDone, 0);
    readAvail = new Semaphore("read avail", 0);
    writeDone = new Semaphore("write done", 0);

    for (;;) {
readAvail->P();// wait for character to arrive
ch = console->GetChar();
console->PutChar(ch);// echo it!
writeDone->P() ;        // wait for write to finish
if (ch == 'q') return;  // if q, quit
    }
}
```

```
/* syscalls.h
 * Nachos system call interface.  These are Nachos kernel operations
 * that can be invoked from user programs, by trapping to the kernel
 *via the "syscall" instruction.
 *
 *This file is included by user programs and by the Nachos kernel.
#ifndef SYSCALLS_H
#define SYSCALLS_H
/* system call codes -- used by the stubs to tell the kernel which
system call
 * is being asked for
 */
#define SC_Halt0
#define SC_Exit1
#define SC_Exec2
#define SC_Join3
#define SC_Create4
#define SC_Open5
#define SC_Read6
#define SC_Write7
#define SC_Close8
#define SC_Fork9
#define SC_Yield10
#ifndef IN_ASM
/* The system call interface.  These are the operations the Nachos
 * kernel needs to support, to be able to run user programs.
 *
 * Each of these is invoked by a user program by simply calling the
 * procedure; an assembly language stub stuffs the system call code
 * into a register, and traps to the kernel.  The kernel procedures
 * are then invoked in the Nachos kernel, after appropriate error
checking,
 * from the system call entry point in exception.cc.
 */
/* Stop Nachos, and print out performance stats */
void Halt();
/* Address space control operations: Exit, Exec, and Join */
/* This user program is done (status = 0 means exited normally). */
void Exit(int status);
/* A unique identifier for an executing user program (address space) */
typedef int SpaceId;
/* Run the executable, stored in the Nachos file "name", and return the
 * address space identifier */
SpaceId Exec(char *name);
/* Only return once the the user program "id" has finished.
 * Return the exit status. */
int Join(SpaceId id);
/* File system operations: Create, Open, Read, Write, Close
 * These functions are patterned after UNIX -- files represent
 * both files *and* hardware I/O devices.
 * If this assignment is done before doing the file system assignment,
 * note that the Nachos file system has a stub implementation, which
 * will work for the purposes of testing out these routines. */
/* A unique identifier for an open Nachos file. */
typedef int OpenFileId;
/* when an address space starts up, it has two open files, representing
 * keyboard input and display output (in UNIX terms, stdin and
stdout).
 * Read and Write can be used directly on these, without first opening
 * the console device. */
#define ConsoleInput0
#define ConsoleOutput1
/* Create a Nachos file, with "name" */
void Create(char *name);
/* Open the Nachos file "name", and return an "OpenFileId" that can
 * be used to read and write to the file. */
OpenFileId Open(char *name);
/* Write "size" bytes from "buffer" to the open file. */
void Write(char *buffer, int size, OpenFileId id);
/* Read "size" bytes from the open file into "buffer".
 * Return the number of bytes actually read -- if the open file isn't
 * long enough, or if it is an I/O device, and there aren't enough
 * characters to read, return whatever is available (for I/O devices,
 * you should always wait until you can return at least one
character). */
int Read(char *buffer, int size, OpenFileId id);
/* Close the file, we're done reading and writing to it. */
void Close(OpenFileId id);
/* User-level thread operations: Fork and Yield.  To allow multiple
 * threads to run within a user program.  */
/* Fork a thread to run a procedure ("func") in the *same* address
space
 * as the current thread. */
void Fork(void (*func)());
/* Yield the CPU to another runnable thread, whether in this address
space
 * or not.  */
void Yield();
#endif /* IN_ASM */
#endif /* SYSCALL_H */
```

```
// console.cc
//Routines to simulate a serial port to a console device.
//A console has input (a keyboard) and output (a display).
//These are each simulated by operations on UNIX files.
//The simulated device is asynchronous,
//so we have to invoke the interrupt handler (after a simulated
//delay), to signal that a byte has arrived and/or that a written
//byte has departed.
//
//  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.
#include "copyright.h"
#include "console.h"
#include "system.h"
// Dummy functions because C++ is weird about pointers to member
functions
static void ConsoleReadPoll(int c)
{ Console *console = (Console *)c; console->CheckCharAvail(); }
static void ConsoleWriteDone(int c)
{ Console *console = (Console *)c; console->WriteDone(); }
//-----------------------------------------------------------------
// Console::Console
// Initialize the simulation of a hardware console device.
//
//"readFile" -- UNIX file simulating the keyboard (NULL -> use stdin)
//"writeFile" -- UNIX file simulating the display (NULL -> use stdout)
// "readAvail" is the interrupt handler called when a character arrives
//from the keyboard
// "writeDone" is the interrupt handler called when a character has
//been output, so that it is ok to request the next char be
//output
//-----------------------------------------------------------------
Console::Console(char *readFile, char *writeFile, VoidFunctionPtr
readAvail,
VoidFunctionPtr writeDone, int callArg)
{
    if (readFile == NULL)
readFileNo = 0;// keyboard = stdin
    else
    readFileNo = OpenForReadWrite(readFile, TRUE);// should be read-
only
    if (writeFile == NULL)
writeFileNo = 1;// display = stdout
    else
    writeFileNo = OpenForWrite(writeFile);
    // set up the stuff to emulate asynchronous interrupts
    writeHandler = writeDone;
    readHandler = readAvail;
    handlerArg = callArg;
    putBusy = FALSE;
    incoming = EOF;
    // start polling for incoming packets
    interrupt->Schedule(ConsoleReadPoll, (int)this, ConsoleTime,
ConsoleReadInt);
}
//-----------------------------------------------------------------
// Console::~Console
// Clean up console emulation
//-----------------------------------------------------------------
Console::~Console()
{
    if (readFileNo != 0)
Close(readFileNo);
    if (writeFileNo != 1)
Close(writeFileNo);
}
//-----------------------------------------------------------------
// Console::CheckCharAvail()
// Periodically called to check if a character is available for
//input from the simulated keyboard (eg, has it been typed?).
//
//Only read it in if there is buffer space for it (if the previous
//character has been grabbed out of the buffer by the Nachos kernel).
//Invoke the "read" interrupt handler, once the character has been
//put into the buffer.
//-----------------------------------------------------------------
void
Console::CheckCharAvail()
{
    char c;
    // schedule the next time to poll for a packet
    interrupt->Schedule(ConsoleReadPoll, (int)this, ConsoleTime,
ConsoleReadInt);
    // do nothing if character is already buffered, or none to be read
    if ((incoming != EOF) || !PollFile(readFileNo))
return;
    // otherwise, read character and tell user about it
    Read(readFileNo, &c, sizeof(char));
    incoming = c ;
    stats->numConsoleCharsRead++;
    (*readHandler)(handlerArg);
}
//-----------------------------------------------------------------
// Console::WriteDone()
// Internal routine called when it is time to invoke the interrupt
//handler to tell the Nachos kernel that the output character has
//completed.
//-----------------------------------------------------------------
void
Console::WriteDone()
{
    putBusy = FALSE;
    stats->numConsoleCharsWritten++;
    (*writeHandler)(handlerArg);
}
//-----------------------------------------------------------------
// Console::GetChar()
// Read a character from the input buffer, if there is any there.
```

```
//Either return the character, or EOF if none buffered.
//--------------------------------------------------------------------
char
Console::GetChar()
{
   char ch = incoming;
   incoming = EOF;
   return ch;
}
//--------------------------------------------------------------------
// Console::PutChar()
// Write a character to the simulated display, schedule an interrupt
//to occur in the future, and return.
//--------------------------------------------------------------------
void
Console::PutChar(char ch)
{
    ASSERT(putBusy == FALSE);
    WriteFile(writeFileNo, &ch, sizeof(char));
    putBusy = TRUE;
    interrupt->Schedule(ConsoleWriteDone, (int)this, ConsoleTime,
ConsoleWriteInt);
}
```

```
// console.h
//Data structures to simulate the behavior of a terminal
//I/O device.  A terminal has two parts -- a keyboard input,
//and a display output, each of which produces/accepts
//characters sequentially.
//
//The console hardware device is asynchronous.  When a character is
//written to the device, the routine returns immediately, and an
//interrupt handler is called later when the I/O completes.
//For reads, an interrupt handler is called when a character arrives.
//
//The user of the device can specify the routines to be called when
//the read/write interrupts occur.  There is a separate interrupt
//for read and write, and the device is "duplex" -- a character
//can be outgoing and incoming at the same time.
//
//  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 CONSOLE_H
#define CONSOLE_H
#include "copyright.h"
#include "utility.h"
// The following class defines a hardware console device.
// Input and output to the device is simulated by reading
// and writing to UNIX files ("readFile" and "writeFile").
//
// Since the device is asynchronous, the interrupt handler "readAvail"
// is called when a character has arrived, ready to be read in.
// The interrupt handler "writeDone" is called when an output character
// has been "put", so that the next character can be written.
class Console {
  public:
    Console(char *readFile, char *writeFile, VoidFunctionPtr readAvail,
VoidFunctionPtr writeDone, int callArg);
// initialize the hardware console device
    ~Console();// clean up console emulation
// external interface -- Nachos kernel code can call these
    void PutChar(char ch);// Write "ch" to the console display,
// and return immediately.  "writeHandler"
// is called when the I/O completes.
    char GetChar();   // Poll the console input.  If a char is
// available, return it.  Otherwise, return EOF.
    // "readHandler" is called whenever there is
// a char to be gotten
// internal emulation routines -- DO NOT call these.
    void WriteDone(); // internal routines to signal I/O completion
    void CheckCharAvail();
  private:
    int readFileNo;// UNIX file emulating the keyboard
    int writeFileNo;// UNIX file emulating the display
    VoidFunctionPtr writeHandler; // Interrupt handler to call when
// the PutChar I/O completes
    VoidFunctionPtr readHandler; // Interrupt handler to call when
// a character arrives from the keyboard
    int handlerArg;// argument to be passed to the
// interrupt handlers
    bool putBusy;    // Is a PutChar operation in progress?
// If so, you can't do another one!
    char incoming;    // Contains the character to be read,
// if there is one available.
// Otherwise contains EOF.
};
#endif // CONSOLE_H
```

```
// disk.cc
//Routines to simulate a physical disk device; reading and writing
//to the disk is simulated as reading and writing to a UNIX file.
//See disk.h for details about the behavior of disks (and
//therefore about the behavior of this simulation).
//
//Disk operations are asynchronous, so we have to invoke an interrupt
//handler when the simulated operation completes.
//
//  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.
#include "copyright.h"
#include "disk.h"
#include "system.h"
// We put this at the front of the UNIX file representing the
// disk, to make it less likely we will accidentally treat a useful
file
// as a disk (which would probably trash the file's contents).
#define MagicNumber 0x456789ab
#define MagicSize sizeof(int)
#define DiskSize (MagicSize + (NumSectors * SectorSize))
// dummy procedure because we can't take a pointer of a member function
static void DiskDone(int arg) { ((Disk *)arg)->HandleInterrupt(); }
//-------------------------------------------------------------------
// Disk::Disk()
// Initialize a simulated disk.  Open the UNIX file (creating it
//if it doesn't exist), and check the magic number to make sure it's
// ok to treat it as Nachos disk storage.
//
//"name" -- text name of the file simulating the Nachos disk
//"callWhenDone" -- interrupt handler to be called when disk read/write
//   request completes
//"callArg" -- argument to pass the interrupt handler
//-------------------------------------------------------------------
Disk::Disk(char* name, VoidFunctionPtr callWhenDone, int callArg)
{
    int magicNum;
    int tmp = 0;
    DEBUG('d', "Initializing the disk, 0x%x 0x%x\n", callWhenDone,
callArg);
    handler = callWhenDone;
    handlerArg = callArg;
    lastSector = 0;
    bufferInit = 0;

    fileno = OpenForReadWrite(name, FALSE);
    if (fileno >= 0) { // file exists, check magic number
Read(fileno, (char *) &magicNum, MagicSize);
ASSERT(magicNum == MagicNumber);
    } else {// file doesn't exist, create it
        fileno = OpenForWrite(name);
magicNum = MagicNumber;
WriteFile(fileno, (char *) &magicNum, MagicSize); // write magic
number
// need to write at end of file, so that reads will not return EOF
        Lseek(fileno, DiskSize - sizeof(int), 0);
WriteFile(fileno, (char *)&tmp, sizeof(int));
    }
    active = FALSE;
}
//-------------------------------------------------------------------
// Disk::~Disk()
// Clean up disk simulation, by closing the UNIX file representing the
//disk.
//-------------------------------------------------------------------
Disk::~Disk()
{
    Close(fileno);
}
//-------------------------------------------------------------------
// Disk::PrintSector()
// Dump the data in a disk read/write request, for debugging.
//-------------------------------------------------------------------
static void
PrintSector (bool writing, int sector, char *data)
{
    int *p = (int *) data;
    if (writing)
        printf("Writing sector: %d\n", sector);
    else
        printf("Reading sector: %d\n", sector);
    for (unsigned int i = 0; i < (SectorSize/sizeof(int)); i++)
printf("%x ", p[i]);
    printf("\n");
}
//-------------------------------------------------------------------
// Disk::ReadRequest/WriteRequest
// Simulate a request to read/write a single disk sector
//   Do the read/write immediately to the UNIX file
//   Set up an interrupt handler to be called later,
//      that will notify the caller when the simulator says
//      the operation has completed.
//
//Note that a disk only allows an entire sector to be read/written,
//not part of a sector.
//
//"sectorNumber" -- the disk sector to read/write
//"data" -- the bytes to be written, the buffer to hold the incoming
bytes
//-------------------------------------------------------------------
void
Disk::ReadRequest(int sectorNumber, char* data)
{
    int ticks = ComputeLatency(sectorNumber, FALSE);
    ASSERT(!active);// only one request at a time
    ASSERT((sectorNumber >= 0) && (sectorNumber < NumSectors));

    DEBUG('d', "Reading from sector %d\n", sectorNumber);
    Lseek(fileno, SectorSize * sectorNumber + MagicSize, 0);
    Read(fileno, data, SectorSize);
    if (DebugIsEnabled('d'))
PrintSector(FALSE, sectorNumber, data);

    active = TRUE;
    UpdateLast(sectorNumber);
    stats->numDiskReads++;
    interrupt->Schedule(DiskDone, (int) this, ticks, DiskInt);
}
void
Disk::WriteRequest(int sectorNumber, char* data)
{
    int ticks = ComputeLatency(sectorNumber, TRUE);
    ASSERT(!active);
    ASSERT((sectorNumber >= 0) && (sectorNumber < NumSectors));

    DEBUG('d', "Writing to sector %d\n", sectorNumber);
    Lseek(fileno, SectorSize * sectorNumber + MagicSize, 0);
    WriteFile(fileno, data, SectorSize);
    if (DebugIsEnabled('d'))
PrintSector(TRUE, sectorNumber, data);

    active = TRUE;
    UpdateLast(sectorNumber);
    stats->numDiskWrites++;
    interrupt->Schedule(DiskDone, (int) this, ticks, DiskInt);
}
//----------------------------------------------------------------------
// Disk::HandleInterrupt()
// Called when it is time to invoke the disk interrupt handler,
//to tell the Nachos kernel that the disk request is done.
//----------------------------------------------------------------------
void
Disk::HandleInterrupt ()
{
    active = FALSE;
    (*handler)(handlerArg);
}
//----------------------------------------------------------------------
// Disk::TimeToSeek()
//Returns how long it will take to position the disk head over the
correct
//track on the disk.  Since when we finish seeking, we are likely
//to be in the middle of a sector that is rotating past the head,
//we also return how long until the head is at the next sector
boundary.
//
//   Disk seeks at one track per SeekTime ticks (cf. stats.h)
//   and rotates at one sector per RotationTime ticks
//----------------------------------------------------------------------
int
Disk::TimeToSeek(int newSector, int *rotation)
{
    int newTrack = newSector / SectorsPerTrack;
    int oldTrack = lastSector / SectorsPerTrack;
    int seek = abs(newTrack - oldTrack) * SeekTime;
// how long will seek take?
    int over = (stats->totalTicks + seek) % RotationTime;
// will we be in the middle of a sector when
// we finish the seek?
    *rotation = 0;
    if (over > 0) // if so, need to round up to next full sector
   *rotation = RotationTime - over;
    return seek;
}
//----------------------------------------------------------------------
// Disk::ModuloDiff()
// Return number of sectors of rotational delay between target sector
//"to" and current sector position "from"
//----------------------------------------------------------------------
int
Disk::ModuloDiff(int to, int from)
{
    int toOffset = to % SectorsPerTrack;
    int fromOffset = from % SectorsPerTrack;
    return ((toOffset - fromOffset) + SectorsPerTrack) %
SectorsPerTrack;
}
//----------------------------------------------------------------------
// Disk::ComputeLatency()
// Return how long will it take to read/write a disk sector, from
//the current position of the disk head.
//
//   Latency = seek time + rotational latency + transfer time
//   Disk seeks at one track per SeekTime ticks (cf. stats.h)
//   and rotates at one sector per RotationTime ticks
//
//   To find the rotational latency, we first must figure out where
the
//   disk head will be after the seek (if any).  We then figure out
//   how long it will take to rotate completely past newSector after
//that point.
//
//   The disk also has a "track buffer"; the disk continuously reads
//   the contents of the current disk track into the buffer.  This
allows
//   read requests to the current track to be satisfied more quickly.
//   The contents of the track buffer are discarded after every seek
to
//   a new track.
//----------------------------------------------------------------------
int
Disk::ComputeLatency(int newSector, bool writing)
{
    int rotation;
    int seek = TimeToSeek(newSector, &rotation);
    int timeAfter = stats->totalTicks + seek + rotation;
#ifndef NOTRACKBUF// turn this on if you don't want the track buffer
stuff
    // check if track buffer applies
    if ((writing == FALSE) && (seek == 0)
&& (((timeAfter - bufferInit) / RotationTime)
     > ModuloDiff(newSector, bufferInit / RotationTime))) {
        DEBUG('d', "Request latency = %d\n", RotationTime);
return RotationTime; // time to transfer sector from the track buffer
    }
#endif
    rotation += ModuloDiff(newSector, timeAfter / RotationTime) *
RotationTime;
    DEBUG('d', "Request latency = %d\n", seek + rotation +
RotationTime);
    return(seek + rotation + RotationTime);
}
//----------------------------------------------------------------------
// Disk::UpdateLast
//   Keep track of the most recently requested sector.  So we can know
//what is in the track buffer.
//----------------------------------------------------------------------
void
Disk::UpdateLast(int newSector)
{
    int rotate;
    int seek = TimeToSeek(newSector, &rotate);

    if (seek != 0)
bufferInit = stats->totalTicks + seek + rotate;
    lastSector = newSector;
    DEBUG('d', "Updating last sector = %d, %d\n", lastSector,
bufferInit);
}
```

```
// disk.h
//Data structures to emulate a physical disk.  A physical disk
//can accept (one at a time) requests to read/write a disk sector;
//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
#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.

    void ReadRequest(int sectorNumber, char* data);
    // Read/write an single disk sector.
// These routines send a request to
    // the disk and return immediately.
    // Only one request allowed at a time!
    void WriteRequest(int sectorNumber, char* data);
    void HandleInterrupt();// Interrupt handler, invoked when
// disk request finishes.
    int ComputeLatency(int newSector, bool writing);
    // Return how long a request to
// newSector will take:
// (seek + rotational delay + transfer)
  private:
    int fileno;// UNIX file number for simulated disk
    VoidFunctionPtr handler;// Interrupt handler, to be invoked
// when any disk request finishes
    int handlerArg;// Argument to interrupt handler
    bool active;     // Is a disk operation in progress?
    int lastSector;// The previous disk request
    int bufferInit;// When the track buffer started
// being loaded
    int TimeToSeek(int newSector, int *rotate); // time to get to the
new track
    int ModuloDiff(int to, int from);        // # sectors between to
and from
    void UpdateLast(int newSector);
};
#endif // DISK_H
```

```
// machine.cc
//Routines for simulating the execution of user programs.
//
//  DO NOT CHANGE -- part of the machine emulation
#include "copyright.h"
#include "machine.h"
#include "system.h"
// Textual names of the exceptions that can be generated by user
program
// execution, for debugging.
static char* exceptionNames[] = { "no exception", "syscall",
"page fault/no TLB entry", "page read only",
"bus error", "address error", "overflow",
"illegal instruction" };
//-----------------------------------------------------------------
// CheckEndian
// Check to be sure that the host really uses the format it says it
//does, for storing the bytes of an integer.  Stop on error.
//-----------------------------------------------------------------
static
void CheckEndian()
{
    union checkit {
        char charword[4];
        unsigned int intword;
    } check;
    check.charword[0] = 1;
    check.charword[1] = 2;
    check.charword[2] = 3;
    check.charword[3] = 4;
#ifdef HOST_IS_BIG_ENDIAN
    ASSERT (check.intword == 0x01020304);
#else
    ASSERT (check.intword == 0x04030201);
#endif
}
//-----------------------------------------------------------------
// Machine::Machine
// Initialize the simulation of user program execution.
//
//"debug" -- if TRUE, drop into the debugger after each user
instruction
//is executed.
//-----------------------------------------------------------------
Machine::Machine(bool debug)
{
    int i;
    for (i = 0; i < NumTotalRegs; i++)
        registers[i] = 0;
    mainMemory = new char[MemorySize];
    for (i = 0; i < MemorySize; i++)
      mainMemory[i] = 0;
#ifdef USE_TLB
    tlb = new TranslationEntry[TLBSize];
    for (i = 0; i < TLBSize; i++)
tlb[i].valid = FALSE;
    pageTable = NULL;
#else// use linear page table
    tlb = NULL;
    pageTable = NULL;
#endif
    singleStep = debug;
    CheckEndian();
}
//-----------------------------------------------------------------
// Machine::~Machine
// De-allocate the data structures used to simulate user program
execution.
//-----------------------------------------------------------------
Machine::~Machine()
{
    delete [] mainMemory;
    if (tlb != NULL)
        delete [] tlb;
}
//-----------------------------------------------------------------
// Machine::RaiseException
// Transfer control to the Nachos kernel from user mode, because
//the user program either invoked a system call, or some exception
//occured (such as the address translation failed).
//
//"which" -- the cause of the kernel trap
//"badVaddr" -- the virtual address causing the trap, if appropriate
//-----------------------------------------------------------------
void
Machine::RaiseException(ExceptionType which, int badVAddr)
{
    DEBUG('m', "Exception: %s\n", exceptionNames[which]);

//  ASSERT(interrupt->getStatus() == UserMode);
    registers[BadVAddrReg] = badVAddr;
    DelayedLoad(0, 0);// finish anything in progress
    interrupt->setStatus(SystemMode);
    ExceptionHandler(which);// interrupts are enabled at this point
    interrupt->setStatus(UserMode);
}
//-----------------------------------------------------------------
// Machine::Debugger
// Primitive debugger for user programs.  Note that we can't use
//gdb to debug user programs, since gdb doesn't run on top of Nachos.
//It could, but you'd have to implement *a lot* more system calls
//to get it to work!
//
//So just allow single-stepping, and printing the contents of memory.
//-----------------------------------------------------------------
void Machine::Debugger()
{
    char *buf = new char[80];
    int num;
    interrupt->DumpState();
    DumpState();
    printf("%d> ", stats->totalTicks);
    fflush(stdout);
    fgets(buf, 80, stdin);
    if (sscanf(buf, "%d", &num) == 1)
runUntilTime = num;
    else {
runUntilTime = 0;
switch (*buf) {
  case '\n':
    break;

  case 'c':
    singleStep = FALSE;
    break;

  case '?':
    printf("Machine commands:\n");
    printf("    <return>  execute one instruction\n");
    printf("    <number>  run until the given timer tick\n");
    printf("    c         run until completion\n");
    printf("    ?         print help message\n");
    break;
}
    }
    delete [] buf;
}

//----------------------------------------------------------------------
// Machine::DumpState
// Print the user program's CPU state.  We might print the contents
//of memory, but that seemed like overkill.
//----------------------------------------------------------------------
void
Machine::DumpState()
{
    int i;

    printf("Machine registers:\n");
    for (i = 0; i < NumGPRegs; i++)
switch (i) {
  case StackReg:
    printf("\tSP(%d):\t0x%x%s", i, registers[i],
   ((i % 4) == 3) ? "\n" : "");
    break;

  case RetAddrReg:
    printf("\tRA(%d):\t0x%x%s", i, registers[i],
   ((i % 4) == 3) ? "\n" : "");
    break;

  default:
    printf("\t%d:\t0x%x%s", i, registers[i],
   ((i % 4) == 3) ? "\n" : "");
    break;
}

    printf("\tHi:\t0x%x", registers[HiReg]);
    printf("\tLo:\t0x%x\n", registers[LoReg]);
    printf("\tPC:\t0x%x", registers[PCReg]);
    printf("\tNextPC:\t0x%x", registers[NextPCReg]);
    printf("\tPrevPC:\t0x%x\n", registers[PrevPCReg]);
    printf("\tLoad:\t0x%x", registers[LoadReg]);
    printf("\tLoadV:\t0x%x\n", registers[LoadValueReg]);
    printf("\n");
}
//----------------------------------------------------------------------
// Machine::ReadRegister/WriteRegister
//   Fetch or write the contents of a user program register.
//----------------------------------------------------------------------
int Machine::ReadRegister(int num)
    {
ASSERT((num >= 0) && (num < NumTotalRegs));
return registers[num];
    }
void Machine::WriteRegister(int num, int value)
    {
ASSERT((num >= 0) && (num < NumTotalRegs));
// DEBUG('m', "WriteRegister %d, value %d\n", num, value);
registers[num] = value;
    }
```

```
// machine.h
//Data structures for simulating the execution of user programs
//running on top of Nachos.
//User programs are loaded into "mainMemory"; to Nachos,
//this looks just like an array of bytes.  Of course, the Nachos
//kernel is in memory too -- but as in most machines these days,
//the kernel is loaded into a separate memory region from user
//programs, and accesses to kernel memory are not translated or paged.
//In Nachos, user programs are executed one instruction at a time,
//by the simulator.  Each memory reference is translated, checked
//for errors, etc.
#ifndef MACHINE_H
#define MACHINE_H
#include "copyright.h"
#include "utility.h"
#include "translate.h"
#include "disk.h"
// Definitions related to the size, and format of user memory
#define PageSize SectorSize // set the page size equal to
// the disk sector size, for simplicity
#define NumPhysPages    32
#define MemorySize (NumPhysPages * PageSize)
#define TLBSize4// if there is a TLB, make it small
enum ExceptionType { NoException,           // Everything ok!
     SyscallException,      // A program executed a system call.
     PageFaultException,    // No valid translation found
     ReadOnlyException,     // Write attempted to page marked  "read-
only"
     BusErrorException,     // Translation resulted in an
// invalid physical address
     AddressErrorException, // Unaligned reference or one that
// was beyond the end of the address space
     OverflowException,     // Integer overflow in add or sub.
     IllegalInstrException, // Unimplemented or reserved instr.
     NumExceptionTypes
};
// User program CPU state.  The full set of MIPS registers, plus a few
// more because we need to be able to start/stop a user program between
// any two instructions (thus we need to keep track of things like load
// delay slots, etc.)
#define StackReg29// User's stack pointer
#define RetAddrReg31// Holds return address for procedure calls
#define NumGPRegs32// 32 general purpose registers on MIPS
#define HiReg32// Double register to hold multiply result
#define LoReg33
#define PCReg34// Current program counter
#define NextPCReg35// Next program counter (for branch delay)
#define PrevPCReg36// Previous program counter (for debugging)
#define LoadReg37// The register target of a delayed load.
#define LoadValueReg 38// The value to be loaded by a delayed load.
#define BadVAddrReg39// The failing virtual address on an exception
#define NumTotalRegs 40
// The following class defines an instruction, represented in both
// undecoded binary form decoded to identify  operation to do
//registers to act on  any immediate operand value
class Instruction {
  public:
    void Decode();// decode the binary representation of the
instruction
    unsigned int value; // binary representation of the instruction
    char opCode;     // Type of instruction.  This is NOT the same as
the
          // opcode field from the instruction: see defs in mips.h
    char rs, rt, rd; // Three registers from instruction.
    int extra;       // Immediate or target or shamt field or offset.
                     // Immediates are sign-extended.
};
// The following class defines the simulated host workstation
hardware, as
// seen by user programs -- the CPU registers, main memory, etc.
// User programs shouldn't be able to tell that they are running on
our
// simulator or on the real hardware, except
//we don't support floating point instructions
//the system call interface to Nachos is not the same as UNIX
//  (10 system calls in Nachos vs. 200 in UNIX!)
// If we were to implement more of the UNIX system calls, we ought to
be
// able to run Nachos on top of Nachos!
// The procedures in this class are defined in machine.cc, mipssim.cc,
and
// translate.cc.
class Machine {
  public:
    Machine(bool debug);// Initialize the simulation of the hardware
// for running user programs
    ~Machine();// De-allocate the data structures
// Routines callable by the Nachos kernel
    void Run(); // Run a user program
    int ReadRegister(int num);// read the contents of a CPU register
    void WriteRegister(int num, int value);
// store a value into a CPU register
// Routines internal to the machine simulation -- DO NOT call these
    void OneInstruction(Instruction *instr);
    // Run one instruction of a user program.
    void DelayedLoad(int nextReg, int nextVal);
// Do a pending delayed load (modifying a reg)

    bool ReadMem(int addr, int size, int* value);
    bool WriteMem(int addr, int size, int value);
    // Read or write 1, 2, or 4 bytes of virtual
// memory (at addr).  Return FALSE if a
// correct translation couldn't be found.

    ExceptionType Translate(int virtAddr, int* physAddr, int size,bool
writing);
    // Translate an address, and check for
// alignment.  Set the use and dirty bits in
// the translation entry appropriately,
    // and return an exception code if the
// translation couldn't be completed.
    void RaiseException(ExceptionType which, int badVAddr);
// Trap to the Nachos kernel, because of a
// system call or other exception.
    void Debugger();// invoke the user program debugger
    void DumpState();// print the user CPU and memory state
// Data structures -- all of these are accessible to Nachos kernel
code.
// "public" for convenience.
// Note that *all* communication between the user program and the
kernel
// are in terms of these data structures.
    char *mainMemory;// physical memory to store user program,
// code and data, while executing
    int registers[NumTotalRegs]; // CPU registers, for executing user
programs
// NOTE: the hardware translation of virtual addresses in the user
program
// to physical addresses (relative to the beginning of "mainMemory")
// can be controlled by one of:
//a traditional linear page table
//  a software-loaded translation lookaside buffer (tlb) -- a cache of
//  mappings of virtual page #'s to physical page #'s
// If "tlb" is NULL, the linear page table is used
// If "tlb" is non-NULL, the Nachos kernel is responsible for managing
//the contents of the TLB.  But the kernel can use any data structure
//it wants (eg, segmented paging) for handling TLB cache misses.
// For simplicity, both the page table pointer and the TLB pointer are
// public.  However, while there can be multiple page tables (one per
address
// space, stored in memory), there is only one TLB (implemented in
hardware).
// Thus the TLB pointer should be considered as *read-only*, although
// the contents of the TLB are free to be modified by the kernel
software.
    TranslationEntry *tlb;// this pointer should be considered
// "read-only" to Nachos kernel code
    TranslationEntry *pageTable;
    unsigned int pageTableSize;
  private:
    bool singleStep;// drop back into the debugger after each
// simulated instruction
    int runUntilTime;// drop back into the debugger when simulated
// time reaches this value
};
extern void ExceptionHandler(ExceptionType which);
// Entry point into Nachos for handling
// user system calls and exceptions
// Defined in exception.cc
// Routines for converting Words and Short Words to and from the
// simulated machine's format of little endian.  If the host machine
// is little endian (DEC and Intel), these end up being NOPs.
// What is stored in each format:
//host byte ordering:
//   kernel data structures
//   user registers
//simulated machine byte ordering:
//   contents of main memory
unsigned int WordToHost(unsigned int word);
unsigned short ShortToHost(unsigned short shortword);
unsigned int WordToMachine(unsigned int word);
unsigned short ShortToMachine(unsigned short shortword);
#endif // MACHINE_H
```

```
// timer.cc
//Routines to emulate a hardware timer device.
//
//      A hardware timer generates a CPU interrupt every X
milliseconds.
//      This means it can be used for implementing time-slicing.
//
//      We emulate a hardware timer by scheduling an interrupt to occur
//      every time stats->totalTicks has increased by TimerTicks.
//
//      In order to introduce some randomness into time-slicing, if
"doRandom"
//      is set, then the interrupt is comes after a random number of
ticks.
//
//Remember -- nothing in here is part of Nachos.  It is just
//an emulation for the hardware that Nachos is running on top of.
//
//  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.
#include "copyright.h"
#include "timer.h"
#include "system.h"
// dummy function because C++ does not allow pointers to member
functions
static void TimerHandler(int arg)
{ Timer *p = (Timer *)arg; p->TimerExpired(); }
//----------------------------------------------------------------------
// Timer::Timer
//      Initialize a hardware timer device.  Save the place to call
//on each interrupt, and then arrange for the timer to start
//generating interrupts.
//
//      "timerHandler" is the interrupt handler for the timer device.
//It is called with interrupts disabled every time the
//the timer expires.
//      "callArg" is the parameter to be passed to the interrupt
handler.
//      "doRandom" -- if true, arrange for the interrupts to occur
//at random, instead of fixed, intervals.
//----------------------------------------------------------------------
Timer::Timer(VoidFunctionPtr timerHandler, int callArg, bool doRandom)
{
    randomize = doRandom;
    handler = timerHandler;
    arg = callArg;
    // schedule the first interrupt from the timer device
    interrupt->Schedule(TimerHandler, (int) this,
TimeOfNextInterrupt(),
TimerInt);
}
//----------------------------------------------------------------------
// Timer::TimerExpired
//      Routine to simulate the interrupt generated by the hardware
//timer device.  Schedule the next interrupt, and invoke the
//interrupt handler.
//----------------------------------------------------------------------
void
Timer::TimerExpired()
{
    // schedule the next timer device interrupt
    interrupt->Schedule(TimerHandler, (int) this,
TimeOfNextInterrupt(),
TimerInt);
    // invoke the Nachos interrupt handler for this device
    (*handler)(arg);
}
//----------------------------------------------------------------------
// Timer::TimeOfNextInterrupt
//      Return when the hardware timer device will next cause an
interrupt.
//If randomize is turned on, make it a (pseudo-)random delay.
//----------------------------------------------------------------------
int
Timer::TimeOfNextInterrupt()
{
    if (randomize)
return 1 + (Random() % (TimerTicks * 2));
    else
return TimerTicks;
}
```

```
// timer.h
//Data structures to emulate a hardware timer.
//
//A hardware timer generates a CPU interrupt every X milliseconds.
//This means it can be used for implementing time-slicing, or for
//having a thread go to sleep for a specific period of time.
//
//We emulate a hardware timer by scheduling an interrupt to occur
//every time stats->totalTicks has increased by TimerTicks.
//
//In order to introduce some randomness into time-slicing, if
"doRandom"
//is set, then the interrupt comes after a random number of ticks.
//
//  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 TIMER_H
#define TIMER_H
#include "copyright.h"
#include "utility.h"
// The following class defines a hardware timer.
class Timer {
  public:
    Timer(VoidFunctionPtr timerHandler, int callArg, bool doRandom);
// Initialize the timer, to call the interrupt
// handler "timerHandler" every time slice.
    ~Timer() {}
// Internal routines to the timer emulation -- DO NOT call these
    void TimerExpired();// called internally when the hardware
// timer generates an interrupt
    int TimeOfNextInterrupt();  // figure out when the timer will
generate
// its next interrupt
  private:
    bool randomize;// set if we need to use a random timeout delay
    VoidFunctionPtr handler;// timer interrupt handler
    int arg;// argument to pass to interrupt handler
};
#endif // TIMER_H
```

```
// translate.cc
//Routines to translate virtual addresses to physical addresses.
//Software sets up a table of legal translations.  We look up
//in the table on every memory reference to find the true physical
//memory location.
//
// Two types of translation are supported here.
//
//Linear page table -- the virtual page # is used as an index
//into the table, to find the physical page #.
//
//Translation lookaside buffer -- associative lookup in the table
//to find an entry with the same virtual page #.  If found,
//this entry is used for the translation.
//If not, it traps to software with an exception.
//
//In practice, the TLB is much smaller than the amount of physical
//memory (16 entries is common on a machine that has 1000's of
//pages).  Thus, there must also be a backup translation scheme
//(such as page tables), but the hardware doesn't need to know
//anything at all about that.
//
//Note that the contents of the TLB are specific to an address space.
//If the address space changes, so does the contents of the TLB!
//
// 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.
#include "copyright.h"
#include "machine.h"
#include "addrspace.h"
#include "system.h"
// Routines for converting Words and Short Words to and from the
// simulated machine's format of little endian.  These end up
// being NOPs when the host machine is also little endian (DEC and
Intel).
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) {
#ifdef 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 */
}
unsigned int
WordToMachine(unsigned int word) { return WordToHost(word); }
unsigned short
ShortToMachine(unsigned short shortword) { return
ShortToHost(shortword); }
//----------------------------------------------------------------------
// Machine::ReadMem
//      Read "size" (1, 2, or 4) bytes of virtual memory at "addr"
into
//the location pointed to by "value".
//
//   Returns FALSE if the translation step from virtual to physical
memory
//   failed.
//
//"addr" -- the virtual address to read from
//"size" -- the number of bytes to read (1, 2, or 4)
//"value" -- the place to write the result
//----------------------------------------------------------------------
bool
Machine::ReadMem(int addr, int size, int *value)
{
    int data;
    ExceptionType exception;
    int physicalAddress;

    DEBUG('a', "Reading VA 0x%x, size %d\n", addr, size);

    exception = Translate(addr, &physicalAddress, size, FALSE);
    if (exception != NoException) {
machine->RaiseException(exception, addr);
return FALSE;
    }
    switch (size) {
      case 1:
data = machine->mainMemory[physicalAddress];
*value = data;
break;
      case 2:
data = *(unsigned short *) &machine->mainMemory[physicalAddress];
*value = ShortToHost(data);
break;
      case 4:
data = *(unsigned int *) &machine->mainMemory[physicalAddress];
*value = WordToHost(data);
break;
      default: ASSERT(FALSE);
    }
    DEBUG('a', "\tvalue read = %8.8x\n", *value);
    return (TRUE);
}
//----------------------------------------------------------------------
// Machine::WriteMem
//      Write "size" (1, 2, or 4) bytes of the contents of "value" into
//virtual memory at location "addr".
//
//   Returns FALSE if the translation step from virtual to physical
memory
//   failed.
//
//"addr" -- the virtual address to write to
//"size" -- the number of bytes to be written (1, 2, or 4)
//"value" -- the data to be written
//----------------------------------------------------------------------
bool
Machine::WriteMem(int addr, int size, int value)
{
    ExceptionType exception;
    int physicalAddress;

    DEBUG('a', "Writing VA 0x%x, size %d, value 0x%x\n", addr, size,
value);
    exception = Translate(addr, &physicalAddress, size, TRUE);
    if (exception != NoException) {
machine->RaiseException(exception, addr);
return FALSE;
    }
    switch (size) {
      case 1:
machine->mainMemory[physicalAddress] = (unsigned char) (value & 0xff);
break;
      case 2:
*(unsigned short *) &machine->mainMemory[physicalAddress]
= ShortToMachine((unsigned short) (value & 0xffff));
break;

      case 4:
*(unsigned int *) &machine->mainMemory[physicalAddress]
= WordToMachine((unsigned int) value);
break;
      default: ASSERT(FALSE);
    }

    return TRUE;
}
//----------------------------------------------------------------------
// Machine::Translate
// Translate a virtual address into a physical address, using
//either a page table or a TLB.  Check for alignment and all sorts
//of other errors, and if everything is ok, set the use/dirty bits in
//the translation table entry, and store the translated physical
//address in "physAddr".  If there was an error, returns the type
//of the exception.
//
//"virtAddr" -- the virtual address to translate
//"physAddr" -- the place to store the physical address
//"size" -- the amount of memory being read or written
// "writing" -- if TRUE, check the "read-only" bit in the TLB
//----------------------------------------------------------------------
ExceptionType
Machine::Translate(int virtAddr, int* physAddr, int size, bool
writing)
{
    int i;
    unsigned int vpn, offset;
    TranslationEntry *entry;
    unsigned int pageFrame;
    DEBUG('a', "\tTranslate 0x%x, %s: ", virtAddr, writing ? "write" :
"read");
// check for alignment errors
    if (((size == 4) && (virtAddr & 0x3)) || ((size == 2) && (virtAddr
& 0x1))){
DEBUG('a', "alignment problem at %d, size %d!\n", virtAddr, size);
return AddressErrorException;
    }

    // we must have either a TLB or a page table, but not both!
    ASSERT(tlb == NULL || pageTable == NULL);
    ASSERT(tlb != NULL || pageTable != NULL);
// calculate the virtual page number, and offset within the page,
// from the virtual address
    vpn = (unsigned) virtAddr / PageSize;
    offset = (unsigned) virtAddr % PageSize;

    if (tlb == NULL) {// => page table => vpn is index into table
if (vpn >= pageTableSize) {
    DEBUG('a', "virtual page # %d too large for page table size
%d!\n",
virtAddr, pageTableSize);
    return AddressErrorException;
} else if (!pageTable[vpn].valid) {
    DEBUG('a', "virtual page # %d too large for page table size
%d!\n",
virtAddr, pageTableSize);
    return PageFaultException;
}
entry = &pageTable[vpn];
    } else {
        for (entry = NULL, i = 0; i < TLBSize; i++)
        if (tlb[i].valid && (tlb[i].virtualPage == vpn)) {
entry = &tlb[i];// FOUND!
break;
    }
if (entry == NULL) {// not found
        DEBUG('a', "*** no valid TLB entry found for this virtual
page!\n");
        return PageFaultException;// really, this is a TLB fault,
// the page may be in memory,
// but not in the TLB
}
```

```
    }
    if (entry->readOnly && writing) {// trying to write to a read-only
page
DEBUG('a', "%d mapped read-only at %d in TLB!\n", virtAddr, i);
return ReadOnlyException;
    }
    pageFrame = entry->physicalPage;
    // if the pageFrame is too big, there is something really wrong!
    // An invalid translation was loaded into the page table or TLB.
    if (pageFrame >= NumPhysPages) {
DEBUG('a', "*** frame %d > %d!\n", pageFrame, NumPhysPages);
return BusErrorException;
    }
    entry->use = TRUE;// set the use, dirty bits
    if (writing)
entry->dirty = TRUE;
    *physAddr = pageFrame * PageSize + offset;
    ASSERT((*physAddr >= 0) && ((*physAddr + size) <= MemorySize));
    DEBUG('a', "phys addr = 0x%x\n", *physAddr);
    return NoException;
}
```

```
// translate.h
//Data structures for managing the translation from
//virtual page # -> physical page #, used for managing
//physical memory on behalf of user programs.
//
//The data structures in this file are "dual-use" - they
//serve both as a page table entry, and as an entry in
//a software-managed translation lookaside buffer (TLB).
//Either way, each entry is of the form:
//<virtual page #, physical page #>.
//
// 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 TLB_H
#define TLB_H
#include "copyright.h"
#include "utility.h"
// The following class defines an entry in a translation table --
either
// in a page table or a TLB.  Each entry defines a mapping from one
// virtual page to one physical page.
// In addition, there are some extra bits for access control (valid and
// read-only) and some bits for usage information (use and dirty).
class TranslationEntry {
  public:
    int virtualPage;  // The page number in virtual memory.
    int physicalPage;  // The page number in real memory (relative to
the
//  start of "mainMemory"
    bool valid;         // If this bit is set, the translation is
ignored.
// (In other words, the entry hasn't been initialized.)
    bool readOnly;// If this bit is set, the user program is not
allowed
// to modify the contents of the page.
    bool use;           // This bit is set by the hardware every time
the
// page is referenced or modified.
    bool dirty;         // This bit is set by the hardware every time
the
// page is modified.
};
#endif
```

```
#include "syscall.h"
int
main()
{
    SpaceId newProc;
    OpenFileId input = ConsoleInput;
    OpenFileId output = ConsoleOutput;
    char prompt[2], ch, buffer[60];
    int i;
    prompt[0] = '-';
    prompt[1] = '-';
    while( 1 )
    {
Write(prompt, 2, output);
i = 0;
do {
    Read(&buffer[i], 1, input);
} while( buffer[i++] != '\n' );
buffer[--i] = '\0';
if( i > 0 ) {
newProc = Exec(buffer);
Join(newProc);
}
    }
}
```

```
/* Start.s
 *Assembly language assist for user programs running on top of Nachos.
 *
 *Since we don't want to pull in the entire C library, we define
 *what we need for a user program here, namely Start and the system
 *calls.
 */
#define IN_ASM
#include "syscall.h"
        .text
        .align  2
/* -------------------------------------------------------------
 * __start
 *Initialize running a C program, by calling "main".
 *
 * NOTE: This has to be first, so that it gets loaded at location 0.
 *The Nachos kernel always starts a program by jumping to location 0.
 * -------------------------------------------------------------
 */
.globl __start
.ent__start
__start:
jalmain
move$4,$0
jalExit /* 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
.entHalt
Halt:
addiu $2,$0,SC_Halt
syscall
j$31
.end Halt
.globl Exit
.entExit
Exit:
addiu $2,$0,SC_Exit
syscall
j$31
.end Exit
.globl Exec
.entExec
Exec:
addiu $2,$0,SC_Exec
syscall
j$31
.end Exec
.globl Join
.entJoin
Join:
addiu $2,$0,SC_Join
syscall
j$31
.end Join
.globl Create
.entCreate
Create:
addiu $2,$0,SC_Create
syscall
j$31
.end Create
.globl Open
.entOpen
Open:
addiu $2,$0,SC_Open
syscall
j$31
.end Open
.globl Read
.entRead
Read:
addiu $2,$0,SC_Read
syscall
j$31
.end Read
.globl Write
.entWrite
Write:
addiu $2,$0,SC_Write
syscall
j$31
.end Write
.globl Close
.entClose
Close:
addiu $2,$0,SC_Close
syscall
j$31
.end Close
.globl Fork
.entFork
Fork:
addiu $2,$0,SC_Fork
syscall
j$31
.end Fork
.globl Yield
.entYield
Yield:
addiu $2,$0,SC_Yield
```

```
syscall
j$31
.end Yield
/* dummy function to keep gcc happy */
        .globl  __main
        .ent    __main
__main:
        j       $31
        .end    __main
```