Oct 3 18:08 1993 copyright.h Page 1

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

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

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

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

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

Oct 3 18:08 1993 list.h Page 1

```
// 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
```



```
// 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.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 InitialPC       s0
#define InitialArg      s1
#define WhenDonePC      s2
#define StartupPC       s3

```



```
#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.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
};

// 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;
#ifdef CHANGED
    /* Changed by rjc - 1/9/96 */
    List *queue;                        // threads waiting for the signal
#endif // CHANGED
};
#endif // SYNCH_H
```

```
// 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.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.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
```

Nov 19 02:18 1993 thread.h Page 3

```
                                        // 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
```

Sep 13 17:02 1994 utility.h Page 1

```
// 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)();
```

Sep 13 17:02 1994 utility.h Page 2

```
// 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
```

Oct 3 18:08 1993 list.cc Page 1

```
// 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.
//
// 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 "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;
}

```

Sep 13 16:54 1994 main.cc Page 1

```
// 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"
```

Sep 13 16:54 1994 main.cc Page 2

```


// 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
    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);
}
```

```
// 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).
//
// 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 "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() {}
bool Lock::isHeldByCurrentThread() {}

#ifdef CHANGED
```

```

// The following changes were made by:
//     Russell J Clark
//     January 1996
//     Implementation of condition variables

//----------------------------------------------------------------------
// Condition::Condition
//            Allocate a list.
//            "debugName" is an arbitrary name, useful for debugging.
//----------------------------------------------------------------------
Condition::Condition(char* debugName)
{
    name = debugName;
    queue = new List;   // a queue of waiting threads
}

//----------------------------------------------------------------------
// Condition::~Condition
//             De-allocate the list.
//----------------------------------------------------------------------
Condition::~Condition()
{
    delete queue;              // only the queue uses dynamic memory
}

//----------------------------------------------------------------------
// Condition::Wait
//             release the lock, relinquish the CPU until signaled,
//             then re-acquire the lock
//----------------------------------------------------------------------
void Condition::Wait(Lock* conditionLock)
{

    IntStatus oldLevel = interrupt->SetLevel(IntOff);   // disable interrupts

    ASSERT(conditionLock->isHeldByCurrentThread());  // make sure we have lock

    queue->Append((void *)currentThread);    // Get on the wait list

    conditionLock->Release();                // Release the lock

    currentThread->Sleep();                  // Go to sleep, waiting

    conditionLock->Acquire();              // We are back alive so
                                           // get the lock before returning.

    (void) interrupt->SetLevel(oldLevel);       // re-enable interrupts

}

//----------------------------------------------------------------------
// Condition::Signal
//            wake up a thread, if there are any waiting on
```



```
//            the condition
//----------------------------------------------------------------------
void Condition::Signal(Lock* conditionLock)
{

    Thread *thread;
    IntStatus oldLevel = interrupt->SetLevel(IntOff);   // disable interrupts

    ASSERT(conditionLock->isHeldByCurrentThread());  // Make sure we have lock

    thread = (Thread *)queue->Remove();         // Remove a thread.

    if (thread != NULL)                         // If there is a thread then,
        scheduler->ReadyToRun(thread);          // make it ready to run.

    (void) interrupt->SetLevel(oldLevel);       // re-enable interrupts
}


//----------------------------------------------------------------------
// Condition::Broadcast
//            wake up all threads waiting on the condition
//----------------------------------------------------------------------
void Condition::Broadcast(Lock* conditionLock)
{
    Thread *thread;
    IntStatus oldLevel = interrupt->SetLevel(IntOff);   // disable interrupts

    ASSERT(conditionLock->isHeldByCurrentThread());  // Make sure we have lock

    // Make ALL waiting threads ready to run. Go through the list and
    // pull off each thread.
    while ((thread = (Thread *)queue->Remove()) != NULL)
        scheduler->ReadyToRun(thread);   // make it ready to run.

    (void) interrupt->SetLevel(oldLevel);       // re-enable interrupts
}



#else   // CHANGED

Condition::Condition(char* debugName) { }
Condition::~Condition() { }
void Condition::Wait(Lock* conditionLock) { ASSERT(FALSE); }
void Condition::Signal(Lock* conditionLock) { }
void Condition::Broadcast(Lock* conditionLock) { }

#endif // CHANGED
```

```
// 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();
}
```

```
// 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);
}
```

```
// 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.
//
// 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 "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(*stack == 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
```

```
// 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"

//----------------------------------------------------------------------
// 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
SimpleThread0(int which)
{
    int num;

    for (num = 0; num < 5; num++) {
        printf("*** thread %d looped %d times\n", which, num);
        currentThread->Yield();
    }
}
void
SimpleThread1(int which)
{
    int num;

    for (num = 0; num < 5; num++) {
        printf("*** thread %d looped %d times\n", which, num);
        currentThread->Yield();
    }
}
void
SimpleThread2(int which)
{
    int num;

    for (num = 0; num < 5; num++) {
        printf("*** thread %d looped %d times\n", which, num);
        currentThread->Yield();
    }
}
//----------------------------------------------------------------------
```



```
// ThreadTest
//      Set up a ping-pong between two threads, by forking a thread
//      to call SimpleThread, and then calling SimpleThread ourselves.
//----------------------------------------------------------------------

void
ThreadTest()
{
    DEBUG('t', "Entering SimpleTest");

    Thread *t1 = new Thread("forked thread 1");
    Thread *t2 = new Thread("forked thread 2");

    // t2->Fork(SimpleThread2, 2);
    t1->Fork(SimpleThread1, 1);
    SimpleThread0(0);
}
```

```
// 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 "/usr/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);
    }
}
```