Timer.cc

```
int time = 40;
// 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 intertimerupt 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.
//----------------------------------------------------------------------
void
Timer::TimerExpired()
{
   /* Experiment 2 */
   /* Add code below to make the timer periodic. */
   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()
{
  /* Experiment 2 */
  /* Update below code so that it returns a fixed time quantum of 40 time ticks */

    if (randomize)
          return time;
    else
          return TimerTicks;
}
```

Interrupt.cc

```
/* Experiment 2 */
/* Add code to remove pending interrupt from head of pending list. */
void
Interrupt::RemovePendingHead(){
	delete pending->Remove();
}

// String definitions for debugging messages
static char *intLevelNames[] = { "off", "on"};
static char *intTypeNames[] = { "timer", "disk", "console write",
				"console read", "network send", "network recv"};

//----------------------------------------------------------------------
// PendingInterrupt::PendingInterrupt
//	Initialize a hardware device interrupt that is to be scheduled
//	to occur in the near future.
//
//	"func" is the procedure to call when the interrupt occurs
//	"param" is the argument to pass to the procedure
//	"time" is when (in simulated time) the interrupt is to occur
//	"kind" is the hardware device that generated the interrupt
//----------------------------------------------------------------------
PendingInterrupt::PendingInterrupt(VoidFunctionPtr func, _int param, int time,
					IntType kind)
{
   handler = func;
   arg = param;
   when = time;
   type = kind;
}



//----------------------------------------------------------------------
// Interrupt::Interrupt
//	Initialize the simulation of hardware device interrupts.
//
//	Interrupts start disabled, with no interrupts pending, etc.
//----------------------------------------------------------------------
Interrupt::Interrupt()
{
   level = IntOff;
   pending = new List();
   inHandler = FALSE;
   yieldOnReturn = FALSE;
   status = SystemMode;
}
```

```
//----------------------------------------------------------------------
// Interrupt::~Interrupt
//          De-allocate the data structures needed by the interrupt simulation.
//----------------------------------------------------------------------
Interrupt::~Interrupt()
{
    while (!pending->IsEmpty())
            delete pending->Remove();
    delete pending;
}



//----------------------------------------------------------------------
// Interrupt::ChangeLevel
//          Change interrupts to be enabled or disabled, without advancing
//          the simulated time (normally, enabling interrupts advances the time).

//----------------------------------------------------------------------
// Interrupt::ChangeLevel
//          Change interrupts to be enabled or disabled, without advancing
//          the simulated time (normally, enabling interrupts advances the time).
//
//          Used internally.
//
//          "old" -- the old interrupt status
//          "now" -- the new interrupt status
//----------------------------------------------------------------------
void
Interrupt::ChangeLevel(IntStatus old, IntStatus now)
{
    level = now;
    DEBUG('i',"\tinterrupts: %s -> %s\n",intLevelNames[old],intLevelNames[now]);
}
```

```
//----------------------------------------------------------------------
// Interrupt::SetLevel
//      Change interrupts to be enabled or disabled, and if interrupts
//      are being enabled, advance simulated time by calling OneTick().
//
// Returns:
//      The old interrupt status.
// Parameters:
//      "now" -- the new interrupt status
//----------------------------------------------------------------------
IntStatus
Interrupt::SetLevel(IntStatus now)
{
    IntStatus old = level;

    ASSERT((now == IntOff) || (inHandler == FALSE));// interrupt handlers are
                                                    // prohibited from enabling
                                                    // interrupts

    ChangeLevel(old, now);                          // change to new state
    if ((now == IntOn) && (old == IntOff))
        OneTick();                                  // advance simulated time
    return old;
}



//----------------------------------------------------------------------
// Interrupt::Enable
//      Turn interrupts on.  Who cares what they used to be?
//      Used in ThreadRoot, to turn interrupts on when first starting up
//      a thread.
//----------------------------------------------------------------------
void
Interrupt::Enable()
{
    (void) SetLevel(IntOn);
}
```

```
//----------------------------------------------------------------------
// Interrupt::OneTick
//      Advance simulated time and check if there are any pending
//      interrupts to be called.
//
//      Two things can cause OneTick to be called:
//              interrupts are re-enabled
//              a user instruction is executed
//----------------------------------------------------------------------
void
Interrupt::OneTick()
{
   MachineStatus old = status;

// advance simulated time
   if (status == SystemMode) {
      stats->totalTicks += SystemTick;
        stats->systemTicks += SystemTick;
   } else {                                     // USER_PROGRAM
        stats->totalTicks += UserTick;
        stats->userTicks += UserTick;
   }
   DEBUG('i', "\n== Tick %d ==\n", stats->totalTicks);

// check any pending interrupts are now ready to fire
   ChangeLevel(IntOn, IntOff);                  // first, turn off interrupts
                                        // (interrupt handlers run with
                                        // interrupts disabled)
   while (CheckIfDue(FALSE))            // check for pending interrupts
        ;
   ChangeLevel(IntOff, IntOn);                  // re-enable interrupts
   if (yieldOnReturn) {         // if the timer device handler asked
                                        // for a context switch, ok to do it now
        yieldOnReturn = FALSE;
        status = SystemMode;                    // yield is a kernel routine
        currentThread->Yield();
        status = old;
   }
}
```

```
//----------------------------------------------------------------------
// Interrupt::YieldOnReturn
//      Called from within an interrupt handler, to cause a context switch
//      (for example, on a time slice) in the interrupted thread,
//      when the handler returns.
//
//      We can't do the context switch here, because that would switch
//      out the interrupt handler, and we want to switch out the
//      interrupted thread.
//----------------------------------------------------------------------
void
Interrupt::YieldOnReturn()
{
    ASSERT(inHandler == TRUE);
    yieldOnReturn = TRUE;
}



//----------------------------------------------------------------------
// Interrupt::Idle
//      Routine called when there is nothing in the ready queue.
//
//      Since something has to be running in order to put a thread
//      on the ready queue, the only thing to do is to advance
//      simulated time until the next scheduled hardware interrupt.
//
//      If there are no pending interrupts, stop.  There's nothing
//      more for us to do.
//----------------------------------------------------------------------
void
Interrupt::Idle()
{
    DEBUG('i', "Machine idling; checking for interrupts.\n");
    status = IdleMode;
    if (CheckIfDue(TRUE)) {             // check for any pending interrupts
        while (CheckIfDue(FALSE))       // check for any other pending
            ;                           // interrupts
      yieldOnReturn = FALSE;            // since there's nothing in the
                                        // ready queue, the yield is automatic
      status = SystemMode;
        return;                         // return in case there's now
                                        // a runnable thread
    }
    // if there are no pending interrupts, and nothing is on the ready
    // queue, it is time to stop.   If the console or the network is
    // operating, there are *always* pending interrupts, so this code
    // is not reached.  Instead, the halt must be invoked by the user program.
    DEBUG('i', "Machine idle.  No interrupts to do.\n");
    printf("No threads ready or runnable, and no pending interrupts.\n");
    printf("Assuming the program completed.\n");
    Halt();
}
```

```
//----------------------------------------------------------------------
// Interrupt::Halt
//       Shut down Nachos cleanly, printing out performance statistics.
//----------------------------------------------------------------------
void
Interrupt::Halt()
{
   printf("Machine halting!\n\n");
   stats->Print();
   Cleanup();     // Never returns.
}



//----------------------------------------------------------------------
// Interrupt::Schedule
//       Arrange for the CPU to be interrupted when simulated time
//       reaches "now + when".
//
//       Implementation: just put it on a sorted list.
//
//       NOTE: the Nachos kernel should not call this routine directly.
//       Instead, it is only called by the hardware device simulators.
//
//       "handler" is the procedure to call when the interrupt occurs
//       "arg" is the argument to pass to the procedure
//       "fromNow" is how far in the future (in simulated time) the
//                 interrupt is to occur
//       "type" is the hardware device that generated the interrupt
//----------------------------------------------------------------------
void
Interrupt::Schedule(VoidFunctionPtr handler, _int arg, int fromNow, IntType type)
{
   int when = stats->totalTicks + fromNow;
   PendingInterrupt *toOccur = new PendingInterrupt(handler, arg, when, type);

   DEBUG('i', "Scheduling interrupt handler the %s at time = %d\n",
                                         intTypeNames[type], when);
   ASSERT(fromNow > 0);

   pending->SortedInsert(toOccur, when);
}
```

```
//----------------------------------------------------------------------
// Interrupt::CheckIfDue
//	Check if an interrupt is scheduled to occur, and if so, fire it off.
//
// Returns:
//	TRUE, if we fired off any interrupt handlers
// Params:
//	"advanceClock" -- if TRUE, there is nothing in the ready queue,
//		so we should simply advance the clock to when the next
//		pending interrupt would occur (if any).  If the pending
//		interrupt is just the time-slice daemon, however, then
//		we're done!
//----------------------------------------------------------------------
bool
Interrupt::CheckIfDue(bool advanceClock)
{
    MachineStatus old = status;
    int when;
    ASSERT(level == IntOff);		// interrupts need to be disabled,
					// to invoke an interrupt handler
    if (DebugIsEnabled('i'))
	DumpState();
    PendingInterrupt *toOccur =
		(PendingInterrupt *)pending->SortedRemove(&when);

    if (toOccur == NULL)		// no pending interrupts
	return FALSE;

    if (advanceClock && when > stats->totalTicks) {	// advance the clock
	stats->idleTicks += (when - stats->totalTicks);
	stats->totalTicks = when;
    } else if (when > stats->totalTicks) {	// not time yet, put it back
	pending->SortedInsert(toOccur, when);
	return FALSE;
    }
// Check if there is nothing more to do, and if so, quit
    if ((status == IdleMode) && (toOccur->type == TimerInt)
				&& pending->IsEmpty()) {
	 pending->SortedInsert(toOccur, when);
	 return FALSE;
    }
    DEBUG('i', "Invoking interrupt handler for the %s at time %d\n",
			intTypeNames[toOccur->type], toOccur->when);
#ifdef USER_PROGRAM
    if (machine != NULL)
	machine->DelayedLoad(0, 0);
#endif
    inHandler = TRUE;
    status = SystemMode;			// whatever we were doing,
						// we are now going to be
						// running in the kernel
    (*(toOccur->handler))(toOccur->arg);	// call the interrupt handler
    status = old;				// restore the machine status
    inHandler = FALSE;
```

```
    delete toOccur;
    return TRUE;
}



//----------------------------------------------------------------------
// PrintPending
//          Print information about an interrupt that is scheduled to occur.
//          When, where, why, etc.
//----------------------------------------------------------------------
static void
PrintPending(_int arg)
{
    PendingInterrupt *pend = (PendingInterrupt *)arg;

    printf("Interrupt handler %s, scheduled at %d\n",
            intTypeNames[pend->type], pend->when);
}



//----------------------------------------------------------------------
// DumpState
//          Print the complete interrupt state - the status, and all interrupts
//          that are scheduled to occur in the future.
//----------------------------------------------------------------------
void
Interrupt::DumpState()
{
    printf("Time: %d, interrupts %s\n", stats->totalTicks,
                                    intLevelNames[level]);
    printf("Pending interrupts:\n");
    fflush(stdout);
    pending->Mapcar(PrintPending);
    printf("End of pending interrupts\n");
    fflush(stdout);
}
```

System.cc

```
//----------------------------------------------------------------------
// 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)
{
  /* Experiment 2 */
  /*Identify where the timer is initialized in this file. Activate the initialization of the timer by updating appropriate variables.*/

  int argCount;
    char* debugArgs = "";
    bool randomYield = TRUE;

#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
            {
                ASSERT(argc > 1);
                RandomInit(10);             // initialize pseudo-random
                                                    // number generator w/ 10
                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);
            /* Experiment 2*/
            /* Debug message to denote scheduling of timer interrupt*/
            DEBUG('i',"*** Timer interrupt scheduled at %d\n",stats->totalTicks+timer-
>TimeOfNextInterrupt());
    }
```

```
    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
#ifdef CHANGED
    processes=new SList;
    processes->Add(new Process, 0, currentThread);
#endif
    console=new Console(0, 0, ReadConsoleAvail, WriteConsoleDone, 0);
    synchConsole=new SynchConsole;
#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

#ifdef CHANGED
    memoryTable=new MemoryTable[NumPhysPages];
#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;

#ifdef CHANGED
#ifdef USER_PROGRAM
   delete processes;
   delete console;
   delete synchConsole;
#endif
#endif

   delete scheduler;
   delete interrupt;

#ifdef CHANGED
   delete [] memoryTable;
#endif

   Exit(0);
}
```

Thread.cc

```
//----------------------------------------------------------------------
// 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 CHANGED
    join_thereP = new Semaphore("Join?", 0);   // create Finish's sem. to
                                               // see if Join is there
#endif
#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 #%i\n", name, pid);

    ASSERT(this != currentThread);
#ifdef CHANGED
    delete join_thereP;
#endif
    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.
//         "joinP" is 0 if this thread cannot be joined and
//                    1 if this thread will be joined
//----------------------------------------------------------------------
#ifdef CHANGED
void
Thread::Fork(VoidFunctionPtr func, int arg, int joinP)
{
   DEBUG('t', "Forking thread %s #%i with func=0x%x, arg=%d, join=%s\n",
          name, pid, (int) func, arg, (joinP ? "YES" : "NO"));

   will_joinP = joinP;                    // remember if you are joined for
                                           // the finish procedure
   StackAllocate(func, arg);

   IntStatus oldLevel = interrupt->SetLevel(IntOff);
   scheduler->ReadyToRun(this);       // ReadyToRun assumes that interrupts
                                          // are disabled!
   (void) interrupt->SetLevel(oldLevel);
}
```

```
//----------------------------------------------------------------------
// Thread::Join
//        Wait until the specified thread has completed running before
//        the code from the point of the Join call on is run.  Both the
//        Join procedure and the thread will return when, and only when,
//        both have finished.
//----------------------------------------------------------------------
void Thread::Join(Thread *forked) {
   DEBUG('j', "Thread %s #%i is calling join on thread %s #%i\n", name, pid,
           forked->getName(), forked->pid);

   Semaphore *join_sem = new Semaphore("in_join", 0);  // Sem. to see if
                                                        // Finish arrived
   forked->JoinHit(join_sem);         // call in forked to send ^^ sem.
                                      // and tell Finish's sem. that
                                      // you are here
   join_sem->P();                     // wait for Finish
   delete join_sem;
}



//----------------------------------------------------------------------
// Thread::JoinHit
//        Allows the function Join to access the semaphore in the thread's
//        class that is being joined, in order to let it know that Join
//        has been called.  Also it sends it's semaphore so it can know
//        if the joined thread has gotten to it's finish state yet.
//----------------------------------------------------------------------
void Thread::JoinHit(class Semaphore *caller) {
   DEBUG('j', "In JoinHit for Thread %s #%i.\n", getName(), pid);

   join_wait = caller;              // Get Join's sem. for Finish to V()
   join_thereP->V();                // Let Finish know Join is here
}
#endif
```

```
//----------------------------------------------------------------------
// 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.
//----------------------------------------------------------------------

static void
TimerInterruptHandler(int dummy)
{
    if (interrupt->getStatus() != IdleMode)
        interrupt->YieldOnReturn();
}

#ifdef CHANGED
void
Thread::Finish ()
{
    if (will_joinP == 0) {          // this thread will not be joined
        (void) interrupt->SetLevel(IntOff);
        ASSERT(this == currentThread);

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

        threadToBeDestroyed = currentThread;

        /* Experiment 2 */
        /* Add code here to reset the timer interrupt so that the next
           interrupt is triggered after 40 time ticks from now.
      */
        interrupt -> Schedule(TimerInterruptHandler, (_int) this, timer ->TimeOfNextInterrupt(),
TimerInt);
        interrupt -> RemovePendingHead();


        Sleep();                                    // invokes SWITCH
    }
    else {                              // this thread will be joined
      DEBUG('j', "Thread %s #%i is here to revive the thread that "
            "called it\n", getName(), pid);

        join_thereP->P();               // make sure the Join proc has
                                         // been called
        join_wait->V();             // tell that Join proc that you
                                         // are in finish and done
```

```
        (void) interrupt->SetLevel(IntOff);
        ASSERT(this == currentThread);

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

        threadToBeDestroyed = currentThread;

        /* Experiment 2 */
        /* Add code here to reset the timer interrupt so that the next
          interrupt is triggered after 40 time ticks from now.
         */

        interrupt -> ScheduHandler, (_int) this, timer ->TimeOfNextInterrupt(), TimerInt);
        interrupt -> RemovePendingHead();

        Sleep();                                        // invokes SWITCH
    }
    // not reached
}
#endif



//----------------------------------------------------------------------
// 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 #%i\n", getName(), pid);

    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 #%i\n", getName(), pid);

    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
```

Interrupt.h

```
void RemovePendingHead(); //add this line
```