threads / scheduler.cc ✓
threads / scheduler.h ✓
threads / thread.cc
threads / thread.h
machine / interrupt.cc
machine / interrupt.h
machine / timer.cc ✓
machine / timer.h ✓

(Exp1) threadtest.cc	(Exp2) threadtest.cc

Difference in void SimpleThread:
- Exp1 calls currentThread->Yield() [Thread::Yield] after printf
- Exp2 just printf upon thread completion.

↑ context switch between the 2 threads for each iteration

void ThreadTest() is the same.

Scheduler

Scheduler()	Initialize list of ready threads
~Scheduler()	De-allocate ready list
ReadyToRun()	Thread can be dispatched → mark as READY
FindNextToRun()	Dequeue (Remove) first thread on ready list & return thread. → if no ready threads, return NULL
Run()	Cause NextThread to start running → if thread is user program, save user CPU registers.
Print()	Print contents of ready list → debugging

Timer

Timer()	Initialize timer to call interrupt handler every time slice → has variable "timerHandler" — the interrupt handler for timer device; called with interrupts disabled every time timer expires.
TimeExpired()	called when hardware timer generates interrupt
TimeOfNextInterrupt()	figure out when the timer will generate next interrupt → In Exp 2, it returns 40.
TimerReset()	make a new Timer

↳ interrupt->Schedule(TimerHandler, (int) this, TimeOfNextInterrupt(), TimerInt);

Thread

Thread()	Constructor that sets thread status to JUST_CREATED; Initialize thread control block, calls Fork()
~Thread()	De-allocates thread → current thread cannot delete itself directly as still running on stack to be deleted
Fork()	Allocate stack → initialize stack, call SWITCH to run procedure → put thread on ready queue
Yield()	
Sleep()	Relinquish CPU as current thread is blocked → thread can wake this thread up & put back on ready queue; interrupts disabled (for correctness)
Finish()	Called by ThreadRoot when a thread is done executing forked procedure → interrupts disable so that no time slice

Difference: Yield does not block calling thread

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

```
//----------------------------------------------------------------------
// 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
```

Interrupt

Interrupt()	initialize interrupt simulation
~Interrupt()	de-allocate data structure
Remove()	remove pending timer interrupt from pending list
SetLevel()	Disable or enable interrupts & return previous settings. → interrupt handlers not allowed to enable interrupts
GetLevel()	return whether interrupts are enabled or disabled
Idle()	ready queue empty, roll simulated time forward until next interrupt → using checkIfDue() → If no pending interrupts, stop.
Halt()	quit and print stats
YieldOnReturn()	context switch on return from interrupt handler → context switch on interrupted thread, not kernel → or else switch out the interrupt handler instead of interrupt thread
Schedule()	schedule interrupt at time 'when' → 'now + when'
OneTick()	Advance simulated time → caused by: enabling interrupts OR user instruction executed
CheckIfDue()	Check if interrupt suppose to happen now
ChangeLevel()	Set level but without advancing the simulated time → normally, enabling interrupts advances time.

Schedule() → just put into sorted list

If a thread is not completed at tick 50, interrupt schedule = 80 (no change)

at tick 80 : interrupt scheduled at 80
A thread completed at tick 50
tick 50 : reset schedule → erase current interrupt schedule at time 50 → set to 90
⇒ 40 + 40 = 80
⇒ 50 + 40 = 90

→ reset timer interrupt so that next interrupt is triggered at now + 40.

```
interrupt -> Remove();
timer -> TimerReset();
Sleep();
```

Thread()

↓

Fork()

↓

SimpleThread()