Possible Questions

- What does ThreadTest() do?
- What do the 3 parameter in Fork() represent?
- What does Fork() do?
- What does the parameter "which" refer to in simpleThread()?
- Each thread executes the function simpleThread(), true or false?
- What is the purpose of simpleThread()?
- Why are interrupts disabled in Yield()?
- What status does ReadyToRun(thread) give to a thread?
- ReadyToRun(thread) starts the thread, True or False?
- FindNextToRun() returns the thread at the _____ of the ready list.
- What does "finishing thread main #0" mean?
- What does "Sleeping thread main #0" mean?
- What does "Deleting thread main #0" mean?
- What would cause a machine to idle?
- When is the scheduler invoked?
- When a machine is idling, it is doing nothing. True or False?
- What happens when there are no interrupts or threads to run?
- What kind of scheduling policy is used?
- In which cc file is the read queue maintained at?
- Multiple threads can be in the running state. True or False?
- What does it mean when the thread status = blocked?

Lab experiment 1

User Program → threadtest.cc

Trace a run of this Nachos test program:

>> ./nachos -d > output.txt

ThreadTest()

```
void ThreadTest() {
    DEBUG('t', "Entering SimpleTest");
    Thread *t1 = new Thread("child 1");
    t1 -> Fork(SimpleThread, 1, 0);
    Thread *t2 = new Thread("child 2");
    t2 -> Fork(SimpleThread, 2, 0);
    SimpleThread(0);
}
```

// Thread constructor does only minimal initialization

// Turns a thread into one that can be scheduled and executed by the CPU

// After starting two new threads, execute the SimpleThread function

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

```
// 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);
}
```

Fork(function, arg, join)

- function: the procedure to be executed concurrently (E.g. SimpleThread)
- arg: A single parameter which is passed to the procedure (E.g. ID of the thread, t1 ID = 1)
- join: a flag which indicates whether the thread can be joined

- Fork(function, arg, flag)
  - Turns a thread into one that can be executed
  - Calls readyToRun()
- Yield()
  - Finds the next thread to run using findNextToRun()
  - If another thread has been found
    - Call readyToRun() for the old thread
    - Run the new thread using Run()
- Sleep()
  - Set status to BLOCKED
  - Find another thread to run using findNextToRun()
  - If another thread has been found
    - Run the new thread using Run()
- Finish()
  - Called at the end of execution
  - Marks a thread for termination

SimpleThread()

```
void SimpleThread(_int which) {
    int num;
    for (num = 0; num < 3; num++) {
        printf("*** thread %d looped %d times\n", (int) which, num);
        currentThread -> Yield();
    }
}
```

// which equivalent to Fork()'s arg → "which" thread; a number identifying the thread for debugging purposes

// Causes the current thread to give up the CPU; yielding CPU to another ready thread each iteration

→ Each thread is executing this function

→ Purpose of SimpleThread(): prints a status line 3 times

Yield()

```
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);
}
```

Purpose: 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 rescheduled

Note:
- returns immediately if no other thread on the ready queue, otherwise returns when the thread eventually works it way to the front of the ready list and gets re-scheduled.
- 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.

Nachos — Scheduler

ReadyToRun(thread)	FindNextToRun(): Thread	Run(thread)
- Makes a thread ready to run (set state to READY) NOT RUNNING - Adds it to the ready list * Does not start the thread yet!	- returns the thread at the front of the ready list - If no ready threads, return NULL - Side effect: Thread is removed from the ready list.	- Switches from one thread to another: 1. Check whether current thread overflowed its stack 2. Change state of new thread to RUNNING 3. Perform the actual Context Switch (call SWITCH()) 4. Terminate previous thread (if applicable)

```
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.
```

→ Maintains a list of threads that are ready to run: ready list (ready queue)
→ The scheduler is invoked whenever the current thread gives up the CPU (non-preemptive)
→ Simple scheduling policy is used
  - Assume equal priority for all threads
  - Select threads in FCFS fashion
  - Append at the end and remove from the front.

Overall for this experiment:
- Short-term CPU scheduler in NachOS
- non-preemptive
- scheduler.cc implements a FIFO scheduling algorithm for ready list ready queue

Nachos — Thread

→ Thread States:
  - READY: eligible to run
  - RUNNING: only one thread can be in this state
  - BLOCKED: waiting for some external event
  - JUST_CREATED: temporary state used during creation.

Tick	ready list	current thread	printf message	Context switch
10	empty	main		
20	child1	main		
30	child1, child2	main	thread 0 looped 0 times	main -> child1
40	child2,main	child1	thread 1 looped 0 times	child1 -> child2
50	main,child1	child2	thread 2 looped 0 times	child2 -> main
60	child1,child2	main	thread 0 looped 1 times	main -> child1
70	child2,main	child1	thread 1 looped 1 times	child1 -> child2
80	main,child1	child2	thread 2 looped 1 times	child2 -> main
90	child1, child2	main	thread 0 looped 2 times	main -> child1
100	child2,main	child1	thread 1 looped 2 times	child1 -> child2
110	main,child1	child2	thread 2 looped 2 times	child2 -> main
120	child1, child2	main		main -> child1
130	child2	child1		child1 -> child2
140	empty	child2		

→ ready list still empty, still putting child 1 into ready list here.
→ child 1 has been placed on ready list, now putting child 2 on ready list here.

```
== Tick 10 ==
        interrupts: on -> off
Time: 10, interrupts off
Pending interrupts:
End of pending interrupts
        interrupts: off -> on
Entering SimpleTestForking thread child1 #0 with func=0x804a640, arg=1,
join=NO
        interrupts: on -> off
Putting thread child1 #0 on ready list.
        interrupts: off -> on

== Tick 20 ==
        interrupts: on -> off
Time: 20, interrupts off
Pending interrupts:
End of pending interrupts
        interrupts: off -> on
Forking thread child2 #0 with func=0x804a640, arg=2, join=NO
        interrupts: on -> off
Putting thread child2 #0 on ready list.
        interrupts: off -> on

== Tick 30 ==
        interrupts: on -> off
Time: 30, interrupts off
Pending interrupts:
End of pending interrupts
        interrupts: off -> on
*** thread 0 looped 0 times
        interrupts: on -> off
Yielding thread main #0
Putting thread main #0 on ready list.
Switching from thread main #0 to thread child1 #0
        interrupts: off -> on
```

PID

→ Yield() disables interrupts so that context switch can happen automatically

→ ReadyToRun(thread) called here

→ Fork() is called

→ SimpleThread() called

→ Run(thread) is called

```
== Tick 120 ==
        interrupts: on -> off
Time: 120, interrupts off
Pending interrupts:
End of pending interrupts
        interrupts: off -> on
        interrupts: on -> off
Finishing thread main #0
Sleeping thread main #0
Switching from thread main #0 to thread child1 #0
Now in thread child1 #0
Deleting thread main #0
        interrupts: off -> on

== Tick 130 ==
        interrupts: on -> off
Time: 130, interrupts off
Pending interrupts:
End of pending interrupts
        interrupts: off -> on
        interrupts: on -> off
Finishing thread child1 #0
Sleeping thread child1 #0
Switching from thread child1 #0 to thread child2 #0
Now in thread child2 #0
Deleting thread child1 #0
        interrupts: off -> on

== Tick 140 ==
        interrupts: on -> off
Time: 140, interrupts off
Pending interrupts:
End of pending interrupts
        interrupts: off -> on
        interrupts: on -> off
Finishing thread child2 #0
Sleeping thread child2 #0
Machine idling; checking for interrupts.
Time: 140, interrupts off
Pending interrupts:
End of pending interrupts
Machine idle.  No interrupts to do.
No threads ready or runnable, and no pending interrupts.
Assuming the program completed.
```

Need to be OFF; for atomacity; no time slice between pulling the first thread off the ready list, and switching to it.

Called when a thread is done executing forked process

To relinquish the CPU, because current thread is blocked waiting for a synchronization variable (e.g. semaphore); main will NOT be placed back inside ready list

```
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.
```

scheduler → FindNextToRun() == NULL ⇒ interrupt → Idle(); // wait for interrupt

} from Interrupt::Idle() ⇒ After checking for pending interrupts and there are none; Halt() invoked