A_5

the Fifth Assignment

Connor Taffe

February 19, 2015

The following is my report for Assignment 5 and the enumerated tasks outlined in it.

$\mathbf{T}_1$

First, I wrote a report on the lab tasks in §3 of Lab 2, as follows:

$\mathbf{T}_{2.3}$

1. First, I moved to the `../threads` directory and cleaned the `../threads/arch/` subdirectory by typing `make clean`:

```
$ cd nachos-3.4/code/threads/
$ ls
arch         main.cc         switch.h        synchlist.h   threadtest.cc
bool.h       Makefile        switch-linux.s  synchtest.cc  utility.cc
copyright.h  Makefile.local  switch.s        system.cc     utility.h
dump         nachos          synch.cc        system.h
list.cc      scheduler.cc    synch.h         thread.cc
list.h       scheduler.h     synchlist.cc    thread.h
$ make clean
rm -f ‘find arch/unknown-i386-linux -type f -print | egrep -v ’(CVS|cvsignore)’‘
rm -f nachos coff2noff coff2flat
rm -f *.noff *.flat
```

2. I then moved to the `../lab2` directory and copied the empty `arch/` directory recursively and `Makefile` and `Makefile.local` from the `../threads` direcotry:

```
$ pwd
/home/cptaffe/nachos-3.4/code/threads
$ cd ../lab2/
```

```
$ ls
$ cp -r ../threads/arch/ .
$ cp ../threads/Makefile .
$ cp ../threads/Makefile.local .
$ ls
arch  Makefile  Makefile.local
```

3. I then copied the needed scheduler files (`scheduler.cc` and `scheduler.h`) from `../threads/` to `lab2` as follows:

```
$ ls
arch  Makefile  Makefile.local
$ cp ../threads/scheduler.* .
$ ls
arch  Makefile  Makefile.local  scheduler.cc  scheduler.h
```

4. I then modified `Makefile.local`'s `INCPATH` variable to be the following:

```
INCPATH += -I- -I../lab2 -I../threads -I../machine
```

   This allows for the compilation of threads with a modified `scheduler.cc` and `scheduler.h` by forcing the compiler to look for header files using the `-I` options following first, so it finds the modified header.

```
$ ls
arch  Makefile  Makefile.local  scheduler.cc  scheduler.h
$ emacs -nw Makefile.local
(screen overwritten by emacs)
$ ls
arch  Makefile  Makefile.local  Makefile.local~  scheduler.cc  scheduler.h
$ diff Makefile.local Makefile.local~
28c28
< INCPATH += -I- -I../lab2 -I../threads -I../machine
---
> INCPATH += -I../threads -I../machine
```

5. I then compiled NachOS from the `../lab2` directory.

```
$ make
...
g++ arch/unknown-i386-linux/objects/main.o ...
ln -sf arch/unknown-i386-linux/bin/nachos nachos
```

6. I then tested that the current directory was used to compile nachos by updating last modified file dates, the mechanism by which `make` uses to determine if a target needs to be remade.

```
$  touch ../threads/scheduler.h
$ make
make: ‘arch/unknown-i386-linux/bin/nachos’ is up to date.
```

This means that our `nachos` build does not depend on that header, this is good.

```
$ touch scheduler.h
$ make
...
g++ arch/unknown-i386-linux/objects/main.o ...
ln -sf arch/unknown-i386-linux/bin/nachos nachos
```

Here, `nachos` did recompile, meaning that our version of this header is the one our `nachos` build depends upon.

$\mathbf{T}_2$

Then, I complete the following tasks and write a report on them for §2.

1. I then created a new directory called `ass5` in my `code/` directory to build an experimental new `nachos`.

```
$ cd ..
$ mkdir ass5
$ pwd
/home/cptaffe/nachos-3.4/code
$ ls
ass3  bin      lab5     Makefile.common  network  userprog
ass4  filesys  lab7-8   Makefile.dep     test     vm
ass5  lab2     machine  monitor          thread
```

2. I then copied `threadtest.cc` from `../threads/` to `../ass5/` and changed the two functions `SimpleThread()` and `ThreadTest()`.

```
$ cp threads/threadtest.cc ass5/
$ cd ass5/
$ ls
threadtest.cc
$ emacs -nw threadtest.cc
(emacs writes to screen)
$ ls
threadtest.cc  threadtest.cc~
$ diff threadtest.cc threadtest.cc~
27,29c27,32
```

```
<   printf("Thread %d before Yield() \n", which);
<   currentThread->Yield();
<   printf("Thread %d after Yield() \n", which);
---
>     int num;
>
>     for (num = 0; num < 5; num++) {
>       printf("*** thread %d looped %d times\n", (int) which, num);
>         currentThread->Yield();
>     }
41,43c44
<   DEBUG('t', "Entering SimpleTest");
<
<   Thread *t;
---
>     DEBUG('t', "Entering SimpleTest");
45,49c46,49
<   for (int i=0; i < 3; i++) {
<     t = new Thread("forked thread");
<     t->Fork(SimpleThread, i);
<   }
<   printf("Main Thread forked 3 threads. \n");
---
>     Thread *t = new Thread("forked thread");
>
>     t->Fork(SimpleThread, 1);
>     SimpleThread(0);
50a51
>
```

3. I then compiled a new `nachos` with this `threadtest.cc`.

```
$ cp ../lab2/Makefile* .
$ ls
Makefile  Makefile.local  Makefile.local~  threadtest.cc  threadtest.cc~
$ cp -r ../lab2/arch/ .
$ ls
arch  Makefile  Makefile.local  Makefile.local~  threadtest.cc  threadtest.cc~
$ make clean
rm -f `find arch/unknown-i386-linux -type f -print | egrep -v '(CVS|cvsignore)'`
rm -f nachos coff2noff coff2flat
rm -f *.noff *.flat
$ make
...
g++ arch/unknown-i386-linux/objects/main.o ...
ln -sf arch/unknown-i386-linux/bin/nachos nachos
```

4. I then ran the new `nachos` and reported the output as follows:

```
$ ./nachos
Main Thread forked 3 threads.
Thread 0 before Yield()
Thread 1 before Yield()
Thread 2 before Yield()
Thread 0 after Yield()
Thread 1 after Yield()
Thread 2 after Yield()
No threads ready or runnable, and no pending interrupts.
Assuming the program completed.
Machine halting!

Ticks: total 100, idle 0, system 100, user 0
Disk I/O: reads 0, writes 0
Console I/O: reads 0, writes 0
Paging: faults 0
Network I/O: packets received 0, sent 0

Cleaning up...
```

$\mathbf{T}_3$

1. The contents of the ready queue when each of the message is as follows:

```
$ emacs -nw
(emacs takes over screen)
M-x gdb
Run gdb (like this): gdb nachos
... (gdb startup messages)
(gdb) break 88
Breakpoint 1 at 0x8048b5e: file ../threads/main.cc, line 88.
(gdb) r
Starting program: /home/cptaffe/nachos-3.4/code/ass5/nachos

Breakpoint 1, main (argc=1, argv=0xbfffbfc4) at ../threads/main.cc:88
(gdb) next
(gdb) step
ThreadTest () at threadtest.cc:41
(gdb) list
...
(gdb) list -
...
27        printf("Thread %d before Yield() \n", which);
```

```
28        currentThread->Yield();
29        printf("Thread %d after Yield() \n", which);
...
(gdb) break 27
Breakpoint 2 at 0x804a962: file threadtest.cc, line 27.
(gdb) break 29
Breakpoint 3 at 0x804a982: file threadtest.cc, line 29.
(gdb) break 49
Breakpoint 4 at 0x804a949: file threadtest.cc, line 49.
```

At this point, I have set breakpoints for main's printf message, and the printf message before and after each forked thread calls `Yeild`.

```
(gdb) continue
Continuing.

Breakpoint 4, ThreadTest () at threadtest.cc:49
(gdb) print *scheduler->readyList
$1 = {first = 0x80551b0, last = 0x80612a0}
(gdb) print *scheduler->readyList->first
$2 = {next = 0x805b228, key = 0, item = 0x804f148}
(gdb) print *((Thread *) scheduler->readyList->first->item)
$3 = {stackTop = 0x8054198, machineState = {0, 0, 134521628, 0, 0, 134523228,
134522172, 134529592, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0}, stack = 0x80501a8,
status = READY, name = 0x804c654 "forked thread"}
(gdb) print *(scheduler->readyList->first->next)
$4 = {next = 0x80612a0, key = 0, item = 0x80551c0}
(gdb) print *((Thread *) scheduler->readyList->first->next->item)
$5 = {stackTop = 0x805a210, machineState = {0, 0, 134521628, 1, 0, 134523228,
134522172, 134529592, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0}, stack = 0x8056220,
status = READY, name = 0x804c654 "forked thread"}
(gdb) print *(scheduler->readyList->first->next->next)
$6 = {next = 0x0, key = 0, item = 0x805b238}
(gdb) print *((Thread *) scheduler->readyList->first->next->next->item)
$7 = {stackTop = 0x8060288, machineState = {0, 0, 134521628, 2, 0, 134523228,
134522172, 134529592, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0}, stack = 0x805c298,
status = READY, name = 0x804c654 "forked thread"}
(gdb) print *currentThread
$33 = {stackTop = 0x0, machineState = {0 <repeats 18 times>}, stack = 0x0,
status = RUNNING, name = 0x804c56a "main"}
```

Here, main has just created three new threads all named "forked thread". As we can see, the ready queue now contains these three threads. All of which are initialized and in a `READY` state. They have yet to be run.

```
(gdb) c
```

```
Continuing.
Main Thread forked 3 threads.

Breakpoint 2, SimpleThread (which=0) at threadtest.cc:27
(gdb) print *scheduler->readyList
$8 = {first = 0x805b228, last = 0x80612a0}
(gdb) print *scheduler->readyList->first
$9 = {next = 0x80612a0, key = 0, item = 0x80551c0}
(gdb) print *scheduler->readyList->first->next
$10 = {next = 0x0, key = 0, item = 0x805b238}
(gdb) print *((Thread *) scheduler->readyList->first->item)
$11 = {stackTop = 0x805a210, machineState = {0, 0, 134521628, 1, 0,
134523228, 134522172, 134529592, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0},
stack = 0x8056220, status = READY, name = 0x804c654 "forked thread"}
(gdb) print *((Thread *) scheduler->readyList->first->next->item)
$12 = {stackTop = 0x8060288, machineState = {0, 0, 134521628, 2, 0,
134523228, 134522172, 134529592, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0},
stack = 0x805c298, status = READY, name = 0x804c654 "forked thread"}
(gdb) print *currentThread
$13 = {stackTop = 0x8054198, machineState = {0, 0, 134521628, 0, 0,
134523228, 134522172, 134529592, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0},
stack = 0x80501a8, status = RUNNING, name = 0x804c654 "forked thread"}
```

Here, we can see that "main" is not preserved as a thread, and that there only two threads in the ready queue, both of which are "forked thread"s. The current thread is also a "forked thread." Thusly, it seems execution is switching between these three "forked thread"s, and since the state of the "main" function's execution has been thrown away, this program will not be able to return to it and exit properly, which leads to our deadlock.

```
(gdb) c
Continuing.
Thread 0 before Yield()

Breakpoint 2, SimpleThread (which=1) at threadtest.cc:27
(gdb) print *scheduler->readyList
$14 = {first = 0x80612a0, last = 0x805b228}
(gdb) print *scheduler->readyList->first
$15 = {next = 0x805b228, key = 0, item = 0x805b238}
(gdb) print *scheduler->readyList->first->next
$16 = {next = 0x0, key = 0, item = 0x804f148}
(gdb) print *((Thread *) scheduler->readyList->first->item)
$17 = {stackTop = 0x8060288, machineState = {0, 0, 134521628, 2, 0,
134523228, 134522172, 134529592, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0},
stack = 0x805c298, status = READY, name = 0x804c654 "forked thread"}
(gdb) print *((Thread *) scheduler->readyList->first->next->item)
```

```
$18 = {stackTop = 0x8054120, machineState = {134541640, 134530644, 6565120,
724249387, 134562124, 134523228, 134522172, 134516763, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0}, stack = 0x80501a8, status = READY,
name = 0x804c654 "forked thread"}
(gdb) print *currentThread
$19 = {stackTop = 0x805a210, machineState = {0, 0, 134521628, 1, 0,
134523228, 134522172, 134529592, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0},
stack = 0x8056220, status = RUNNING, name = 0x804c654 "forked thread"}
```

We can tell from the address of the `Thread` objects that `0x80551c0` is currently running and `0x804f148` has been put in the ready queue, so the threads are indeed switching between themselves.

The next few breakpoints will be a cursory glance at the ready queue, as we have no further need for in depth information about each thread.

```
(gdb) c
Continuing.
Thread 1 before Yield()

Breakpoint 2, SimpleThread (which=2) at threadtest.cc:27
(gdb) print *scheduler->readyList
$20 = {first = 0x805b228, last = 0x80612a0}
(gdb) print *scheduler->readyList->first
$21 = {next = 0x80612a0, key = 0, item = 0x804f148}
(gdb) print *scheduler->readyList->first->next
$22 = {next = 0x0, key = 0, item = 0x80551c0}
(gdb) print currentThread
$23 = (Thread *) 0x805b238

(gdb) c
Continuing.
Thread 2 before Yield()

Breakpoint 3, SimpleThread (which=0) at threadtest.cc:29
(gdb) print *scheduler->readyList
$24 = {first = 0x80612a0, last = 0x805b228}
(gdb) print *scheduler->readyList->first
$25 = {next = 0x805b228, key = 0, item = 0x80551c0}
(gdb) print *scheduler->readyList->first->next
$26 = {next = 0x0, key = 0, item = 0x805b238}
(gdb) print currentThread
$27 = (Thread *) 0x804f148
```

This is right before "Thread 0 after Yield()" is printed, so this thread, `0x804f148`, is Thread 0, and is about to be removed after `SimpleThread`

returns. When it returns there will be nothing left to run and this thread will not be put back in the ready queue.

```
(gdb) c
Continuing.
Thread 0 after Yield()

Breakpoint 3, SimpleThread (which=1) at threadtest.cc:29
(gdb) print *scheduler->readyList
$28 = {first = 0x805b228, last = 0x805b228}
(gdb) print *scheduler->readyList->first
$29 = {next = 0x0, key = 0, item = 0x805b238}
(gdb) print currentThread
$30 = (Thread *) 0x80551c0
```

Here, there are only two threads, the currently scheduled thread, `0x80551c0`, and the thread in the ready queue, `0x805b238`. Thread `0x804f148` finished execution at the last breakpoint as `SimpleThread` returned and the thread logic removed it. At this breakpoint, `0x80551c0` will be removed.

```
(gdb) c
Continuing.
Thread 1 after Yield()

Breakpoint 3, SimpleThread (which=2) at threadtest.cc:29
(gdb) print *scheduler->readyList
$31 = {first = 0x0, last = 0x0}
(gdb) print currentThread
$32 = (Thread *) 0x805b238
```

Here, the last thread, `0x805b238`, is removed.

```
Continuing.
Thread 2 after Yield()
No threads ready or runnable, and no pending interrupts.
Assuming the program completed.
Machine halting!

Ticks: total 100, idle 0, system 100, user 0
Disk I/O: reads 0, writes 0
Console I/O: reads 0, writes 0
Paging: faults 0
Network I/O: packets received 0, sent 0

Cleaning up...

Program exited normally.
```

In summary, the three "forked thread"s are switched about, and then sequentially return and are no longer.

2. No. It is running initially, spawns three threads, then runs `currentThread->Finish()`, which means it basically kills itself instead of yielding so it will never be rescheduled because it has nothing to run. On finishing, the thread scheduler automatically picks another ready thread and runs it.

3. The "main" thread deleted the main thread object after it returned from `ThreadTest` it executed `currentThread->Finish()` which, since there was nothing else to run, destroyed its `Thread` object and it was never rescheduled because `Thread::Finish` sets `currentThread` to be destroyed and then calls `Sleep()`.

4. Following is a tabular display of the requested information.

Context Switch Number	Current Thread	Next Thread	Causal Function
1	M	F_0	`Sleep()`
2	F_0	F_1	`Yield()`
3	F_1	F_2	`Yield()`
4	F_2	F_0	`Yield()`
5	F_0	F_1	`Sleep()`
6	F_1	F_2	`Sleep()`
7	F_2	...	`Sleep()`

The "..." indicates there is not another thread to schedule, which means `Sleep` calls `Idle` which calls `Halt` which prints statistics and calls `Cleanup`, which does the cleanup and ends execution by calling `Exit` which calls `exit` (stdlib function). In essense, "..." represents the program end point.