A_4

the Fourth Assignment

Connor Taffe

February 12, 2015

The following is my report for Assignment 4 and the enumerated tasks outlined in it.

$\mathbf{T}_1$

First, I started emacs and, using the M-x prompt, started gdb to debug nachos binary. Then I added a breakpoint at the function call `Initialize(argc, argv)`.

```
$ emacs -nw
(screen is overwritten by emacs)
M-x gdb
Run gdb (like this): gdb nachos
```

At this point gdb starts and prints out some licensing information and details.

```
(gdb) list
78      //              ex: "nachos -d +" -> argv = {"nachos", "-d", "+"}
79      //---------------------------------------------------------------------\
-
80
81      int
82      main(int argc, char **argv)
83      {
84          int argCount;                       // the number of arguments
85                                              // for a particular command
86
87          DEBUG('t', "Entering main");
(gdb) list
88          (void) Initialize(argc, argv);
89
90      #ifdef THREADS
```

```
91          ThreadTest();
92      #if 0
93          SynchTest();
94      #endif
95      #endif
96
97          for (argc--, argv++; argc > 0; argc -= argCount, argv += argCount) \
{
(gdb) break 88
Breakpoint 1 at 0x8048b5e: file main.cc, line 88.
(gdb)
```

$\mathbf{T}_2$

Then I finished `Initialize(argc, argv)` without stepping into it.

```
(gdb) r
Starting program: /home/cptaffe/nachos-3.4/code/threads/nachos

Breakpoint 1, main (argc=1, argv=0xbfffc1d4) at main.cc:88
(gdb) next
(gdb)
```

The lower panel of emacs shows the following after running the above, it is reproduced in the following.

```
{
int argCount;                       // the number of arguments
// for a particular command

DEBUG('t', "Entering main");
(void) Initialize(argc, argv);

#ifdef THREADS
=>  ThreadTest();
#if 0
```

Which means that I have finished `Initialize(argc, argv)`.

$\mathbf{T}_3$

Then I printed the value of `currentThread` and the binary value of `*currentThread` as follows.

```
(gdb) print currentThread
$1 = (Thread *) 0x804f0e8
```

```
(gdb) print /x *currentThread
$2 = {stackTop = 0x0, machineState = {0x0 <repeats 18 times>}, stack = 0x0,
status = 0x1, name = 0x804c54e}
(gdb) print *currentThread
$3 = {stackTop = 0x0, machineState = {0 <repeats 18 times>}, stack = 0x0,
status = RUNNING, name = 0x804c54e "main"}
(gdb)
```

This means that `currentThread` is an object of type `Thread *` (a Thread pointer) which has an address of `0x804c54e`, and that the address `0x804c54e` is where the following `Thread` structure is stored (found with `*currentThread`):

```
{stackTop = 0x0, machineState = {0x0 <repeats 18 times>}, stack = 0x0,
status = 0x1, name = 0x804c54e}
```

Since we just called `Initialize(argc, argv)`, this structure is not yet storing the state of a thread. We can see that `stackTop` is null, as is `machineState` and `stack`.

The following is the same structure printed without the `/x` option, which shows the name of the constant `RUNNING` stored in status, and the string stored in name, `"main"`.

```
{stackTop = 0x0, machineState = {0 <repeats 18 times>}, stack = 0x0,
status = RUNNING, name = 0x804c54e "main"}
```

This structure represents the current thread of execution, as it has not saved state yet, is `RUNNING`, and is named “main.”

$\mathbf{T}_4$

I then stepped into `ThreadTest`.

```
(gdb) step
ThreadTest () at threadtest.cc:44
(gdb)
```

The lower panel of emacs displays the following source code.

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

void
ThreadTest()
{
=>  DEBUG(’t’, "Entering SimpleTest");

    Thread *t = new Thread("forked thread");

    t->Fork(SimpleThread, 1);
    SimpleThread(0);
```

$\mathbf{T}_5$

I then finished all the statements up to `SimpleThread(0);` and printed the value of `t` and the binary value of `*t`.

```
(gdb) next
(gdb)
(gdb)
(gdb) print t
$4 = (Thread *) 0x804f148
(gdb) print /x *t
$5 = {stackTop = 0x8054198, machineState = {0x0, 0x0, 0x804a31c, 0x1, 0x0,
0x804a8b8, 0x804a53c, 0x804c22c, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0,
0x0, 0x0}, stack = 0x80501a8, status = 0x2, name = 0x804c64b}
(gdb) print *t
$6 = {stackTop = 0x8054198, machineState = {0, 0, 134521628, 1, 0, 134523064,
134522172, 134529580, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0}, stack = 0x80501a8,
status = READY, name = 0x804c64b "forked thread"}
(gdb) print SimpleThread
$7 = {void (int)} 0x804a8b8 <SimpleThread(int)>
(gdb) print *scheduler->readyList
$8 = {first = 0x80551b0, last = 0x80551b0}
(gdb) print *(scheduler->readyList->first)
$9 = {next = 0x0, key = 0, item = 0x804f148}
```

`t` is a new thread named “forked thread.” There now exist two threads, the “main” thread, which is currently executing, and the “forked thread” which is `READY` (ready to be scheduled), but not `RUNNING` (currently executing).

a. Two threads have been created so far. They are the “main” thread and the “forked thread” thread, which are both objects of type `Thread`. “main” was created in the `main` function and represents the currently running thread while “forked thread” was just created in `ThreadTest`.

b. The “main” thread is the currently running thread as it is in the `RUNNING` state and not the `READY` state. It was also created earlier and our execution state has not been saved yet or our context changed.

c. The ready queue, `*scheduler->readyList`, for Nachos contains the following:

   ```
   {first = 0x80551b0, last = 0x80551b0}
   ```

   There is one element in the ready queue because `first` and `last` are the same address, 0x80551b0. The ready queueu contains the “forked thread” thread, which makes sense as “forked thread”’s state is `READY`. The address of `first` and `last`, 0x80551b0, is the address to a stucture, `*(scheduler->readyList->first)`, which contains a member `item` as follows:

```
{next = 0x0, key = 0, item = 0x804f148}
```

`item` is a pointer to address `0x804f148`. This address is the address of "forked thread", the newly created thread stored in variable `t`.

T_6

I then stepped into function `SimpleThread(0)`.

```
(gdb) step
SimpleThread (which=0) at threadtest.cc:29
(gdb)
```

The lower panel in emacs shows the following code snippet.

```
void
SimpleThread(_int which)
{
    int num;

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

T_7

I then finished the `printf()` statement of the first iteration of the loop.

```
(gdb) next
(gdb)
*** thread 0 looped 0 times
(gdb) print which
$12 = 0
(gdb)
```

a. The output of this `printf` statement is "*** thread 0 looped 0 times".

b. `which` has a value of 0.

T_8

I then stepped into `currentThread->Yield();`.

```
(gdb) step
Thread::Yield (this=0x804f0e8) at thread.cc:183
(gdb)
```

The lower panel in emacs contains the following code snippet.

```
void
Thread::Yield ()
{
    Thread *nextThread;
=>  IntStatus oldLevel = interrupt->SetLevel(IntOff);

    ASSERT(this == currentThread);

    DEBUG('t', "Yielding thread \"%s\"\n", getName());
```

T$_9$

Then I finished all the statements up to the `if (nextThread != NULL)` statement.

```
(gdb)next
(gdb)
(gdb)
(gdb)
(gdb) print nextThread
$14 = (Thread *) 0x804f148
(gdb) print *nextThread
$15 = {stackTop = 0x8054100, machineState = {134541640, 134530382, 6565120,
724249387, 134562092, 134523064, 134522172, 134516763, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0}, stack = 0x80501a8, status = READY,
name = 0x804c64b "forked thread"}
(gdb) print *scheduler->readyList
$16 = {first = 0x0, last = 0x0}
```

a. `nextThread` points to the “forked thread” thread because the pointer address, 0x804f148, is the same address from `t` in `ThreadTest` and `*nextThread` yeilds an attribute `name` that is `"forked thread"`.

b. The ready queue is empty, because both the `first` and `last` attributes are 0x0. This is because we are still running the “main” thread, and the `scheduler->FindNextToRun()` function removed “forked thread” from the queue, so both threads are currently not on the queue.

$\mathbf{T}_{10}$

I then finished `scheduler->ReadyToRun(this);`.

```
(gdb) next
(gdb)
(gdb) print *scheduler->readyList
$17 = {first = 0x80551b0, last = 0x80551b0}
(gdb) print *(scheduler->readyList->first)
$18 = {next = 0x0, key = 0, item = 0x804f0e8}
```

a. The ready queue contains one element because the `first` and `last` elements of the `scheduler->readyList` structure are the same pointer value (0x80551b0). That address points to the following structure.

   ```
   {next = 0x0, key = 0, item = 0x804f0e8}
   ```

   The `item` element is an address, the same address that was the value of `currentThread` in `main` after `Initialize(argc, argv)` which makes it the “main” thread.

$\mathbf{T}_{11}$

I then stepped into `scheduler->Run(nextThread);` and finished all the statements up to function call `SWITCH(oldThread, nextThread);`.

```
(gdb) step
Scheduler::Run (this=0x804f0c8, nextThread=0x804f148) at scheduler.cc:93
(gdb) next
(gdb)
(gdb)
(gdb)
(gdb)
(gdb) print oldThread
$20 = (Thread *) 0x804f0e8
(gdb) print *oldThread
$21 = {stackTop = 0xbfffc02c, machineState = {134541544, 134530635, 6565120,
724249387, -1073758120, 3415200, 0, 134516763, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0}, stack = 0x0, status = READY, name = 0x804c54e "main"}
(gdb) print nextThread
$22 = (Thread *) 0x804f148
(gdb) print *nextThread
$23 = {stackTop = 0x8054100, machineState = {134541640, 134530382, 6565120,
724249387, 134562092, 134523064, 134522172, 134516763, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0}, stack = 0x80501a8, status = RUNNING,
name = 0x804c64b "forked thread"}
```

```
(gdb) print currentThread
$24 = (Thread *) 0x804f148
```

After stepping into `scheduler->Run(nextThread);`, the lower panel in emacs displays the following code snippet.

```
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();
```

a. `oldThread` points to the "main" thread, while `newThread` points to the "forked thread" thread.

b. The "main" thread is currently running as we have not yet executed the `SWITCH` function to switch threads. Although `currentThread` was assigned to "forked thread" (`nextThread`) and its status set to `RUNNING`, we have not actually switched threads yet.

T$_{12}$

I then disassembled the current function and found the value of the program counter (i.e. `eip`).

```
(gdb) disas
Dump of assembler code for function Scheduler::Run(Thread*):
0x08048f9e <Scheduler::Run(Thread*)+0>: push   %ebp
0x08048f9f <Scheduler::Run(Thread*)+1>: mov    %esp,%ebp
0x08048fa1 <Scheduler::Run(Thread*)+3>: push   %ebx
0x08048fa2 <Scheduler::Run(Thread*)+4>: sub    $0x24,%esp
0x08048fa5 <Scheduler::Run(Thread*)+7>: mov    0x804e1d4,%eax
0x08048faa <Scheduler::Run(Thread*)+12>:        mov    %eax,-0x8(%ebp)
0x08048fad <Scheduler::Run(Thread*)+15>:        mov    -0x8(%ebp),%eax
0x08048fb0 <Scheduler::Run(Thread*)+18>:        mov    %eax,(%esp)
0x08048fb3 <Scheduler::Run(Thread*)+21>:        call   0x804a34c <Thread::Check\
Overflow()>
0x08048fb8 <Scheduler::Run(Thread*)+26>:        mov    0xc(%ebp),%eax
0x08048fbb <Scheduler::Run(Thread*)+29>:        mov    %eax,0x804e1d4
0x08048fc0 <Scheduler::Run(Thread*)+34>:        mov    0x804e1d4,%eax
0x08048fc5 <Scheduler::Run(Thread*)+39>:        movl   $0x1,0x4(%esp)
0x08048fcd <Scheduler::Run(Thread*)+47>:        mov    %eax,(%esp)
```

```
0x08048fd0 <Scheduler::Run(Thread*)+50>:        call   0x80491de <Thread::setSt\
atus(ThreadStatus)>
0x08048fd5 <Scheduler::Run(Thread*)+55>:        mov    0xc(%ebp),%eax
0x08048fd8 <Scheduler::Run(Thread*)+58>:        mov    %eax,(%esp)
0x08048fdb <Scheduler::Run(Thread*)+61>:        call   0x80491ec <Thread::getNa\
me()>
0x08048fe0 <Scheduler::Run(Thread*)+66>:        mov    %eax,%ebx
0x08048fe2 <Scheduler::Run(Thread*)+68>:        mov    -0x8(%ebp),%eax
0x08048fe5 <Scheduler::Run(Thread*)+71>:        mov    %eax,(%esp)
0x08048fe8 <Scheduler::Run(Thread*)+74>:        call   0x80491ec <Thread::getNa\
me()>
0x08048fed <Scheduler::Run(Thread*)+79>:        mov    %ebx,0xc(%esp)
0x08048ff1 <Scheduler::Run(Thread*)+83>:        mov    %eax,0x8(%esp)
0x08048ff5 <Scheduler::Run(Thread*)+87>:        movl   $0x804c41c,0x4(%esp)
0x08048ffd <Scheduler::Run(Thread*)+95>:        movl   $0x74,(%esp)
0x08049004 <Scheduler::Run(Thread*)+102>:       call   0x804a86a <DEBUG(char, c\
har*, ...)>
0x08049009 <Scheduler::Run(Thread*)+107>:       mov    0xc(%ebp),%eax
0x0804900c <Scheduler::Run(Thread*)+110>:       mov    %eax,0x4(%esp)
0x08049010 <Scheduler::Run(Thread*)+114>:       mov    -0x8(%ebp),%eax
0x08049013 <Scheduler::Run(Thread*)+117>:       mov    %eax,(%esp)
0x08049016 <Scheduler::Run(Thread*)+120>:       call   0x804c23a <SWITCH>
0x0804901b <Scheduler::Run(Thread*)+125>:       mov    0x804e1d4,%eax
0x08049020 <Scheduler::Run(Thread*)+130>:       mov    %eax,(%esp)
0x08049023 <Scheduler::Run(Thread*)+133>:       call   0x80491ec <Thread::getNa\
me()>
0x08049028 <Scheduler::Run(Thread*)+138>:       mov    %eax,0x8(%esp)
0x0804902c <Scheduler::Run(Thread*)+142>:       movl   $0x804c447,0x4(%esp)
0x08049034 <Scheduler::Run(Thread*)+150>:       movl   $0x74,(%esp)
0x0804903b <Scheduler::Run(Thread*)+157>:       call   0x804a86a <DEBUG(char, c\
har*, ...)>
0x08049040 <Scheduler::Run(Thread*)+162>:       mov    0x804e1d8,%eax
0x08049045 <Scheduler::Run(Thread*)+167>:       test   %eax,%eax
0x08049047 <Scheduler::Run(Thread*)+169>:       je     0x8049077 <Scheduler::Ru\
n(Thread*)+217>
0x08049049 <Scheduler::Run(Thread*)+171>:       mov    0x804e1d8,%eax
0x0804904e <Scheduler::Run(Thread*)+176>:       mov    %eax,-0x18(%ebp)
0x08049051 <Scheduler::Run(Thread*)+179>:       cmpl   $0x0,-0x18(%ebp)
0x08049055 <Scheduler::Run(Thread*)+183>:       je     0x804906d <Scheduler::Ru\
n(Thread*)+207>
0x08049057 <Scheduler::Run(Thread*)+185>:       mov    -0x18(%ebp),%eax
0x0804905a <Scheduler::Run(Thread*)+188>:       mov    %eax,(%esp)
0x0804905d <Scheduler::Run(Thread*)+191>:       call   0x804a6bc <Thread::~Thre\
ad()>
0x08049062 <Scheduler::Run(Thread*)+196>:       mov    -0x18(%ebp),%eax
0x08049065 <Scheduler::Run(Thread*)+199>:       mov    %eax,(%esp)
```

```
0x08049068 <Scheduler::Run(Thread*)+202>:       call   0x8048878 <_ZdlPv@plt>
0x0804906d <Scheduler::Run(Thread*)+207>:       movl   $0x0,0x804e1d8
0x08049077 <Scheduler::Run(Thread*)+217>:       add    $0x24,%esp
0x0804907a <Scheduler::Run(Thread*)+220>:       pop    %ebx
0x0804907b <Scheduler::Run(Thread*)+221>:       pop    %ebp
0x0804907c <Scheduler::Run(Thread*)+222>:       ret
End of assembler dump.
(gdb) print $eip
$26 = (void (*)(void)) 0x8049009 <Scheduler::Run(Thread*)+107>
(gdb)
```

$\mathbf{T}_{13}$

I then traced the program in the machine level and finish the instructions up to instruction `0x08049016 <Scheduler::Run(Thread*)+120>:  call 0x804c23a <SWITCH>`

```
(gdb) nexti
(gdb)
(gdb)
(gdb)
(gdb) print $eip
$27 = (void (*)(void)) 0x8049016 <Scheduler::Run(Thread*)+120>
(gdb)
```

$\mathbf{T}_{14}$

I then stepped into assembly function `SWITCH` using stepi and disassembled it.

```
(gdb) stepi
0x0804c23a in SWITCH ()
(gdb) disas
Dump of assembler code for function SWITCH:
0x0804c23a <SWITCH+0>:  mov    %eax,0x804e1f4
0x0804c23f <SWITCH+5>:  mov    0x4(%esp),%eax
0x0804c243 <SWITCH+9>:  mov    %ebx,0x8(%eax)
0x0804c246 <SWITCH+12>: mov    %ecx,0xc(%eax)
0x0804c249 <SWITCH+15>: mov    %edx,0x10(%eax)
0x0804c24c <SWITCH+18>: mov    %esi,0x18(%eax)
0x0804c24f <SWITCH+21>: mov    %edi,0x1c(%eax)
0x0804c252 <SWITCH+24>: mov    %ebp,0x14(%eax)
0x0804c255 <SWITCH+27>: mov    %esp,(%eax)
0x0804c257 <SWITCH+29>: mov    0x804e1f4,%ebx
0x0804c25d <SWITCH+35>: mov    %ebx,0x4(%eax)
0x0804c260 <SWITCH+38>: mov    (%esp),%ebx
```

```
0x0804c263 <SWITCH+41>: mov    %ebx,0x20(%eax)
0x0804c266 <SWITCH+44>: mov    0x8(%esp),%eax
0x0804c26a <SWITCH+48>: mov    0x4(%eax),%ebx
0x0804c26d <SWITCH+51>: mov    %ebx,0x804e1f4
0x0804c273 <SWITCH+57>: mov    0x8(%eax),%ebx
0x0804c276 <SWITCH+60>: mov    0xc(%eax),%ecx
0x0804c279 <SWITCH+63>: mov    0x10(%eax),%edx
0x0804c27c <SWITCH+66>: mov    0x18(%eax),%esi
0x0804c27f <SWITCH+69>: mov    0x1c(%eax),%edi
0x0804c282 <SWITCH+72>: mov    0x14(%eax),%ebp
0x0804c285 <SWITCH+75>: mov    (%eax),%esp
0x0804c287 <SWITCH+77>: mov    0x20(%eax),%eax
0x0804c28a <SWITCH+80>: mov    %eax,(%esp)
0x0804c28d <SWITCH+83>: mov    0x804e1f4,%eax
0x0804c292 <SWITCH+88>: ret
0x0804c293 <SWITCH+89>: nop
... (several nop instructions omitted)
0x0804c29f <SWITCH+101>:        nop
End of assembler dump.
(gdb) x /3w $esp
0xbfffc02c:     0x0804901b      0x0804f0e8      0x0804f148
```

a. The following are three memory addresses pointed to be `%esp`.

```
0xbfffc02c:     0x0804901b      0x0804f0e8      0x0804f148
```

b. The first address, 0x0804901b, is the address of the next instruction in `scheduler->Run(nextThread);` after the `call` for this function. The next address, 0x0804f0e8, is the address of the “main” thread. The next address, 0x0804f148, is the address of the “forked thread” thread.

   These addresses are here becuase two of them, the “main” thread from `oldThread`, and “forked thread” from `nextThread` were passed here as arguments; and the final one is the return address pushed by the `call` instruction.

T$_{15}$

I then finished all the instructions up to `ret` instruction.

```
(gdb) print $eip
$29 = (void (*)(void)) 0x804c23a <SWITCH>
(gdb) nexti
0x0804c23f in SWITCH ()
(gdb)
... (several returns omitted)
0x0804c292 in SWITCH ()
```

$\mathbf{T}_{16}$

I then finished the `ret` instruction with `nexti`.

```
(gdb) nexti
Scheduler::Run (this=0x804f0c8, nextThread=0x804f0e8) at scheduler.cc:118
```

a. I am in `Scheduler::Run`, disassembly is as follows.

```
(gdb) disas
Dump of assembler code for function ThreadRoot:
0x0804c22c <ThreadRoot+0>:      push   %ebp
0x0804c22d <ThreadRoot+1>:      mov    %esp,%ebp
0x0804c22f <ThreadRoot+3>:      push   %edx
0x0804c230 <ThreadRoot+4>:      call   *%ecx
0x0804c232 <ThreadRoot+6>:      call   *%esi
0x0804c234 <ThreadRoot+8>:      call   *%edi
0x0804c236 <ThreadRoot+10>:     mov    %ebp,%esp
0x0804c238 <ThreadRoot+12>:     pop    %ebp
0x0804c239 <ThreadRoot+13>:     ret
End of assembler dump.
(gdb)
```

a. `SWITCH` returns here because it uses the return address as a way to switch into the last state of another thread. After storing registers, it starts off the other thread like nothing happened by returning from the function that the new thread would have called when it was put into a `READY` state from a `RUNNING`.

b. `SWITCH` stores the return value for that the "main" thread and will use it to resume the "main" thread (which is no longer active) when a switch to that thread occurs. `SWITCH`'s job was to switch to "forked thread", so it returned control to it instead of "main" after saving "main"'s state and restoring "forked thread"'s.

c. Although we can be sure that this thread is "forked thread" because it is not "main" and `SWITCH` performed a context switch and returned control to the second parameter, `newThread` or "forked thread"; it is a simple task to check `currentThread`, as it is changed during switches like this.

```
(gdb) print currentThread
$30 = (Thread *) 0x804f148
```

0x804f148 is the address for the "forked thread" thread, so that is the thread that we are running.

$\mathbf{T}_{17}$

I then finished all the instructions up to `call *%esi`.

```
(gdb) print $eip
$3 = (void (*)(void)) 0x804c22f <ThreadRoot+3>
(gdb) nexti
0x0804c230 in ThreadRoot ()
(gdb)
0x0804c232 in ThreadRoot ()
(gdb)
```

$\mathbf{T}_{18}$

I then stepped into the function pointed by `%esi` by using `stepi`.

```
(gdb) stepi
SimpleThread (which=1) at threadtest.cc:25
(gdb)
```

Here the lower pane updates with the following snippet.

```
void
=>mpleThread(_int which)
{
    int num;

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

a. I am now in `SimpleThread`.

b. We are still in the same thread, nothing has changed the thread state, we are just in the thread assembly that sets up the thread, runs a job, and returns the thread. But just to be sure, we could check `currentThread`.

   ```
   (gdb) print currentThread
   $4 = (Thread *) 0x804f148
   ```

   Yes, we are still in the “forked thread” thread.

$\mathbf{T}_{19}$

Then I switched to trace the program at the source level. I finished all statements up to the `printf()` statement and found the value of `which`.

```
(gdb) next
(gdb)
(gdb) print which
$5 = 1
(gdb)
```

The value of `which` is 1.

$\mathbf{T}_{20}$

Then I finished the `printf()` and stepped into `currentThread->Yield();`.

```
(gdb) next
*** thread 1 looped 0 times
(gdb) step
Thread::Yield (this=0x804f148) at thread.cc:183
(gdb)
```

$\mathbf{T}_{21}$

I then finished all the statements up to `SWITCH(oldThread, nextThread);`.

```
(gdb) n
(gdb)
(gdb)
(gdb)
(gdb)
(gdb)
(gdb) s
Scheduler::Run (this=0x804f0c8, nextThread=0x804f0e8) at scheduler.cc:93
(gdb) n
(gdb) n
(gdb) n
(gdb) n
(gdb) n
(gdb) print oldThread
$1 = (Thread *) 0x804f148
(gdb) print nextThread
$2 = (Thread *) 0x804f0e8
(gdb)
```

a. `oldThread` is a pointer to the address 0x804f148, `nextThread` is a pointer to the address 0x804f0e8.

b. `oldThread` is the “forked thread” thread, `nextThread` is the “main” thread.

$\mathbf{T}_{22}$

I then stepped into assembly function `SWITCH` using `stepi` and dissembled it.

```
(gdb) stepi
(gdb) disas
Dump of assembler code for function Scheduler::Run(Thread*):
... (duplicate of dissasembly on page 10-11)
End of assembler dump.
```

$\mathbf{T}_{23}$

I then finished all the instructions up to instruction `ret`.

```
(gdb) nexti
(gdb) print $eip
$3 = (void (*)(void)) 0x8049010 <Scheduler::Run(Thread*)+114>
(gdb) nexti
```

$\mathbf{T}_{24}$

I then finish the `ret` instruction by `nexti`.

a. The current function is `Scheduler::Run`.

b. The current `%eip` value is as follows,

```
(gdb) print $eip
$1 = (void (*)(void)) 0x804901b <Scheduler::Run(Thread*)+125>
```

so the current instruction is the first in the following list, and the next instruction to execute follows it.

```
0x0804901b <Scheduler::Run(Thread*)+125>:       mov    0x804e1d4,%eax
0x08049020 <Scheduler::Run(Thread*)+130>:       mov    %eax,(%esp)
```

c. The current thread is “main” because it has gone through a second context switch with only two possible threads, so “main” is our only option.

```
(gdb) print currentThread
$2 = (Thread *) 0x804f0e8
```

As we expected, 0x804f0e8 coresponds to the “main” thread.

d. The `ret` instruction reads the value at `esp` and changes `eip` to that address (in short). Any value can be placed at `esp`. A `jmp` instruction could have been used to achieve the same effect. In short, it is no hurdle for the CPU to do this, the context of the thread is only conceptual, all the CPU knows is that it is pushing/poping, jumping, etc.