A_8

Assignment VIII

Connor Taffe. T no. 3742

April 14th, 2015

1 Details of Lab 4-5

Here follows an account of laboratory 4, section 5. This section follows the original lab specification to detail a report, the "Report of Lab 4-5" is found in section 2.

Q. 1 Here, I tested my object oriented monitor implementation with the three configurations listed in the original lab.

Q. 1.1 Here follows a test using Configuration 1 with the following qunatities:

- buffer size: 2
- number of producers: 4
- number of messages per producer: 3
- number of consumers: 2

First, I compiled a new `nachos` binary with the above configuration as follows:

```
$ make
...
ln -sf arch/unknown-i386-linux/bin/nachos nachos
```

Then, I ran nachos with the quantities above set appropriately as `./nachos`, which printed the following outputs:

```
$ ./nachos
No threads ready or runnable, and no pending interrupts.
Assuming the program completed.
Machine halting!

Ticks: total 1070, idle 0, system 1070, 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...
$ ls tmp_*
tmp_0  tmp_1
$ cat tmp_0
producer id --> 0; Message number --> 0;
producer id --> 1; Message number --> 0;
producer id --> 2; Message number --> 0;
producer id --> 3; Message number --> 0;
producer id --> 3; Message number --> 1;
producer id --> 1; Message number --> 2;
$ cat tmp_1
producer id --> 0; Message number --> 1;
producer id --> 3; Message number --> 2;
producer id --> 1; Message number --> 1;
producer id --> 0; Message number --> 2;
producer id --> 2; Message number --> 1;
producer id --> 2; Message number --> 2;
```

Following the first run, I ran nachos with a random seed of 99 by using the command `./nachos -rs 99`. The output is recorded in the following:

```
$ ./nachos -rs 99
No threads ready or runnable, and no pending interrupts.
Assuming the program completed.
Machine halting!

Ticks: total 1162, idle 2, system 1160, 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...
$ ls tmp_*
tmp_0  tmp_1
$ cat tmp_0
producer id --> 0; Message number --> 0;
producer id --> 1; Message number --> 0;
producer id --> 2; Message number --> 0;
producer id --> 0; Message number --> 1;
producer id --> 0; Message number --> 2;
producer id --> 3; Message number --> 2;
$ cat tmp_1
producer id --> 2; Message number --> 1;
producer id --> 2; Message number --> 2;
producer id --> 1; Message number --> 1;
producer id --> 3; Message number --> 0;
```

```
producer id --> 1; Message number --> 2;
producer id --> 3; Message number --> 1;
```

As you can see from the outputs recorded above, this first configuration's test is successful.

Q. 1.2 Here follows a test using Configuration 2 with the following qunatities:

- buffer size: 5
- number of producers: 3
- number of messages per producer: 4
- number of consumers: 3

First, I compiled a new `nachos` binary with the above configuration as follows:

```
$ make
...
ln -sf arch/unknown-i386-linux/bin/nachos nachos
```

Then, I ran nachos with the quantities above set appropriately as `./nachos`, which printed the following outputs:

```
$ ./nachos
No threads ready or runnable, and no pending interrupts.
Assuming the program completed.
Machine halting!

Ticks: total 1090, idle 0, system 1090, 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...
$ ls tmp_*
tmp_0  tmp_1  tmp_2
$ cat tmp_0
producer id --> 0; Message number --> 0;
producer id --> 1; Message number --> 0;
producer id --> 2; Message number --> 0;
producer id --> 0; Message number --> 1;
producer id --> 2; Message number --> 2;
producer id --> 0; Message number --> 3;
$ cat tmp_1
producer id --> 1; Message number --> 1;
producer id --> 2; Message number --> 3;
```

```
producer id --> 1; Message number --> 2;
$ cat tmp_2
producer id --> 2; Message number --> 1;
producer id --> 0; Message number --> 2;
producer id --> 1; Message number --> 3;
```

Following the first run, I ran nachos with a random seed of 99 by using the command `./nachos -rs 99`. The output is recorded in the following:

```
$ ./nachos -rs 99
No threads ready or runnable, and no pending interrupts.
Assuming the program completed.
Machine halting!

Ticks: total 1195, idle 5, system 1190, 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...
$ ls tmp_*
tmp_0  tmp_1  tmp_2
$ cat tmp_0
producer id --> 0; Message number --> 0;
producer id --> 1; Message number --> 0;
producer id --> 1; Message number --> 1;
producer id --> 0; Message number --> 2;
$ cat tmp_1
producer id --> 2; Message number --> 0;
producer id --> 2; Message number --> 1;
producer id --> 0; Message number --> 1;
producer id --> 1; Message number --> 2;
producer id --> 0; Message number --> 3;
$ cat tmp_2
producer id --> 2; Message number --> 2;
producer id --> 2; Message number --> 3;
producer id --> 1; Message number --> 3;
```

As you can see from the outputs recorded above, this second configuration's test is successful.

Q. 1.3 Here follows a test using Configuration 3 with the following qunatities:

- buffer size: 5
- number of producers: 3

- number of messages per producer: 4
- number of consumers: 3

First, I compiled a new `nachos` binary with the above configuration as follows:

```
$ make
...
ln -sf arch/unknown-i386-linux/bin/nachos nachos
```

Then, I ran nachos with the quantities above set appropriately as `./nachos`, which printed the following outputs:

```
$ ./nachos
No threads ready or runnable, and no pending interrupts.
Assuming the program completed.
Machine halting!

Ticks: total 7730, idle 0, system 7730, 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...
$ ls tmp_*
tmp_0  tmp_1  tmp_2  tmp_3
$ cat tmp_0
producer id --> 0; Message number --> 0;
producer id --> 1; Message number --> 0;
producer id --> 2; Message number --> 0;
producer id --> 3; Message number --> 0;
producer id --> 4; Message number --> 0;
producer id --> 4; Message number --> 1;
producer id --> 2; Message number --> 1;
producer id --> 3; Message number --> 3;
producer id --> 2; Message number --> 2;
producer id --> 0; Message number --> 1;
producer id --> 1; Message number --> 3;
producer id --> 0; Message number --> 2;
producer id --> 2; Message number --> 4;
producer id --> 0; Message number --> 5;
producer id --> 3; Message number --> 4;
producer id --> 2; Message number --> 5;
producer id --> 4; Message number --> 3;
producer id --> 0; Message number --> 6;
producer id --> 0; Message number --> 7;
producer id --> 4; Message number --> 4;
```

```
producer id --> 0; Message number --> 11;
producer id --> 4; Message number --> 5;
producer id --> 1; Message number --> 13;
producer id --> 4; Message number --> 10;
producer id --> 2; Message number --> 11;
producer id --> 4; Message number --> 11;
producer id --> 3; Message number --> 8;
producer id --> 2; Message number --> 12;
producer id --> 0; Message number --> 17;
producer id --> 4; Message number --> 12;
producer id --> 4; Message number --> 13;
producer id --> 4; Message number --> 18;
producer id --> 3; Message number --> 13;
producer id --> 3; Message number --> 16;
producer id --> 3; Message number --> 19;
producer id --> 2; Message number --> 17;
$ cat tmp_1
producer id --> 3; Message number --> 1;
producer id --> 4; Message number --> 2;
producer id --> 3; Message number --> 2;
producer id --> 1; Message number --> 1;
producer id --> 2; Message number --> 3;
producer id --> 1; Message number --> 2;
producer id --> 0; Message number --> 9;
producer id --> 1; Message number --> 6;
producer id --> 1; Message number --> 10;
producer id --> 4; Message number --> 8;
producer id --> 1; Message number --> 16;
producer id --> 2; Message number --> 9;
producer id --> 4; Message number --> 9;
producer id --> 1; Message number --> 17;
producer id --> 4; Message number --> 14;
producer id --> 2; Message number --> 18;
$ cat tmp_2
producer id --> 0; Message number --> 4;
producer id --> 0; Message number --> 8;
producer id --> 3; Message number --> 5;
producer id --> 1; Message number --> 7;
producer id --> 4; Message number --> 7;
producer id --> 3; Message number --> 6;
producer id --> 0; Message number --> 12;
producer id --> 1; Message number --> 8;
producer id --> 1; Message number --> 9;
producer id --> 0; Message number --> 13;
producer id --> 1; Message number --> 14;
producer id --> 2; Message number --> 8;
producer id --> 0; Message number --> 14;
producer id --> 1; Message number --> 15;
```

```
producer id --> 2; Message number --> 13;
producer id --> 4; Message number --> 15;
producer id --> 3; Message number --> 9;
producer id --> 2; Message number --> 14;
producer id --> 0; Message number --> 18;
producer id --> 4; Message number --> 16;
producer id --> 4; Message number --> 17;
producer id --> 0; Message number --> 19;
producer id --> 3; Message number --> 10;
producer id --> 3; Message number --> 11;
producer id --> 3; Message number --> 14;
producer id --> 3; Message number --> 17;
producer id --> 2; Message number --> 15;
producer id --> 2; Message number --> 19;
$ cat tmp_3
producer id --> 0; Message number --> 3;
producer id --> 0; Message number --> 10;
producer id --> 1; Message number --> 4;
producer id --> 4; Message number --> 6;
producer id --> 1; Message number --> 5;
producer id --> 2; Message number --> 6;
producer id --> 1; Message number --> 11;
producer id --> 3; Message number --> 7;
producer id --> 2; Message number --> 7;
producer id --> 1; Message number --> 12;
producer id --> 0; Message number --> 15;
producer id --> 1; Message number --> 18;
producer id --> 2; Message number --> 10;
producer id --> 0; Message number --> 16;
producer id --> 1; Message number --> 19;
producer id --> 4; Message number --> 19;
producer id --> 3; Message number --> 12;
producer id --> 3; Message number --> 15;
producer id --> 3; Message number --> 18;
producer id --> 2; Message number --> 16;
```

Following the first run, I ran nachos with a random seed of 99 by using the command `./nachos -rs 99`. The output is recorded in the following:

```
$ ./nachos -rs 99
No threads ready or runnable, and no pending interrupts.
Assuming the program completed.
Machine halting!

Ticks: total 8783, idle 113, system 8670, 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...
$ ls tmp_*
tmp_0  tmp_1  tmp_2  tmp_3
$ cat tmp_0
producer id --> 0; Message number --> 0;
producer id --> 1; Message number --> 0;
producer id --> 2; Message number --> 0;
producer id --> 3; Message number --> 0;
producer id --> 4; Message number --> 0;
producer id --> 0; Message number --> 1;
producer id --> 0; Message number --> 2;
producer id --> 3; Message number --> 3;
producer id --> 4; Message number --> 2;
producer id --> 3; Message number --> 4;
producer id --> 2; Message number --> 2;
producer id --> 1; Message number --> 1;
producer id --> 3; Message number --> 5;
producer id --> 0; Message number --> 3;
producer id --> 1; Message number --> 3;
producer id --> 4; Message number --> 9;
producer id --> 4; Message number --> 10;
producer id --> 1; Message number --> 7;
producer id --> 1; Message number --> 9;
producer id --> 0; Message number --> 9;
producer id --> 2; Message number --> 6;
producer id --> 1; Message number --> 12;
producer id --> 1; Message number --> 14;
producer id --> 0; Message number --> 11;
producer id --> 3; Message number --> 15;
producer id --> 2; Message number --> 15;
producer id --> 2; Message number --> 16;
producer id --> 2; Message number --> 17;
producer id --> 2; Message number --> 18;
producer id --> 0; Message number --> 15;
producer id --> 0; Message number --> 19;
$ cat tmp_1
producer id --> 3; Message number --> 1;
producer id --> 3; Message number --> 2;
producer id --> 3; Message number --> 6;
producer id --> 2; Message number --> 3;
producer id --> 4; Message number --> 4;
producer id --> 1; Message number --> 2;
producer id --> 4; Message number --> 5;
producer id --> 0; Message number --> 4;
producer id --> 4; Message number --> 6;
```

```
producer id --> 0; Message number --> 5;
producer id --> 3; Message number --> 7;
producer id --> 1; Message number --> 4;
producer id --> 0; Message number --> 6;
producer id --> 1; Message number --> 5;
producer id --> 0; Message number --> 7;
producer id --> 4; Message number --> 14;
producer id --> 3; Message number --> 12;
producer id --> 1; Message number --> 16;
producer id --> 2; Message number --> 10;
producer id --> 1; Message number --> 17;
producer id --> 2; Message number --> 11;
producer id --> 2; Message number --> 14;
producer id --> 0; Message number --> 12;
producer id --> 2; Message number --> 19;
producer id --> 0; Message number --> 13;
producer id --> 3; Message number --> 18;
producer id --> 3; Message number --> 19;
producer id --> 0; Message number --> 17;
$ cat tmp_2
producer id --> 2; Message number --> 1;
producer id --> 4; Message number --> 3;
producer id --> 4; Message number --> 7;
producer id --> 3; Message number --> 10;
producer id --> 1; Message number --> 6;
producer id --> 2; Message number --> 5;
producer id --> 4; Message number --> 13;
producer id --> 1; Message number --> 10;
producer id --> 1; Message number --> 13;
producer id --> 2; Message number --> 8;
producer id --> 1; Message number --> 15;
producer id --> 4; Message number --> 15;
producer id --> 4; Message number --> 16;
producer id --> 2; Message number --> 9;
producer id --> 1; Message number --> 18;
producer id --> 4; Message number --> 17;
producer id --> 2; Message number --> 12;
producer id --> 2; Message number --> 13;
producer id --> 4; Message number --> 18;
producer id --> 3; Message number --> 16;
producer id --> 0; Message number --> 14;
producer id --> 0; Message number --> 18;
$ cat tmp_3
producer id --> 4; Message number --> 1;
producer id --> 4; Message number --> 8;
producer id --> 2; Message number --> 4;
producer id --> 4; Message number --> 11;
producer id --> 3; Message number --> 8;
```

```
producer id --> 3; Message number --> 9;
producer id --> 4; Message number --> 12;
producer id --> 1; Message number --> 8;
producer id --> 0; Message number --> 8;
producer id --> 1; Message number --> 11;
producer id --> 3; Message number --> 11;
producer id --> 2; Message number --> 7;
producer id --> 0; Message number --> 10;
producer id --> 3; Message number --> 13;
producer id --> 3; Message number --> 14;
producer id --> 1; Message number --> 19;
producer id --> 4; Message number --> 19;
producer id --> 3; Message number --> 17;
producer id --> 0; Message number --> 16;
```

As you can see from the outputs recorded above, this third configuration's test is successful.

In summary, I have shown that all three configurations yield satisfactory results which show that the synchronization in my program is correct.

2 Report of Lab 4-5

Here follows the report for laboratory 4, section 5.

Q. 2 Here I describe my analysis and design of an object oriented monitor which makes use of Hoare style condition variable and semaphore-based monitors as described in our assigned text "Operating Systems Concepts" in section 5.8, as well as the original Hoare paper "Monitors: An Operating System Structuring Concept". I took information from both these sources while implementing this program.

The idea of the monitor superclass is to allow the intheritance of it to a subclass which makes use of a monitor, thusly centralizing the code base and simplifying the creation of said subclasses. For subclasses, the monitor provides two funtions, **Entry** and **Exit**. Entry attempts to acquire a mutex (a semaphore) with a P operation. Complementarily, Exit releases the mutx, but *only* if there are no processes in next. Processes in next have already secured exclusion and suspended themselves with a signal, they must be released on exit rather than releasing the mutex. The number of processes P'd in next is recorded in next_count.

The monitor also provides it's mutex, next semaphore, and next_count number as public members so conditions can use them to wait and signal (as described in the logic of Exit). In particular, the Hoare-style condition variable that was implemented provides a **Wait** and **Signal** procedure as described in Hoare's 1973 paper. The Wait procedure causes a procedure inside the monitor which "causes the calling program to be delayed" or blocks the calling function. The complementary function Signal wakes up only one blocked process, or, if none are waiting, it does nothing. The signalled program must return control to the waiting program immediately, so that a third program cannot seize control of the asset (this is where the Exit logic comes from).

The Ring buffer is implemented using these ideas. It has two condition variables for the two conditions it deals with: notfull and notempty. Since the ring buffer has two operations, Put and Get, it has two complementary conditions. When Put is called it may be full, and cannot accept a put, so it waits on notfull to allow any waiting Get processes to work on the buffer. After completing its operations, Put will signal notempty to tell any waiting get processes that there are more messages to get, which allows get processes to acquire the monitor before any other processes would be able to. Note that Put and Get only call their respective waits if there are no empty/full slots to produce/consume.

This logic works successfully and provides an efficient mechanism for producer/consumer processes, or coroutines.

Q. 3 Here follows the implementation of the monitor.

The following is the code for the provided monitor header:

```
Monitor::Monitor(char *debugName)
    : name(debugName),
    mutex(new Semaphore(name, 1)),
    next(new Semaphore(name, 0)),
    next_count(0) {}

Monitor::~Monitor() {
    delete mutex;
    delete next;
}

// Enter and Exit taken from Hoare's
// "MONITORS: AN OPERATING SYSTEM STRUCTURING CONCEPT" paper.

void Monitor::Entry() {
    mutex->P();
}

void Monitor::Exit() {
    if (next_count > 0) {
        next->V();
    } else {
        mutex->V();
    }
}
```

My use of initializer lists was a stylistic choice. Enter and Exit are as described above and the constructor and destructor are fairly simplistic.

Here is the implementation of the Condition_H class:

```
// condition variables in Hoare's style
Condition_H::Condition_H(char *debugName, Monitor *m)
    : name(debugName),
```

```
    sem(new Semaphore(name, 0)), // uses the same name as cond var
    count(0),
    mon(m) {}

Condition_H::~Condition_H()
{
    delete sem;
}

void Condition_H::Wait()
{
    IntStatus oldLevel = interrupt->SetLevel(IntOff);

    count++;
    if (mon->next_count > 0) {
        mon->next->V();
    } else {
        mon->mutex->V();
    }
    sem->P();
    count--;

    (void) interrupt->SetLevel(oldLevel);
}


void Condition_H::Signal()
{
    IntStatus oldLevel = interrupt->SetLevel(IntOff);

    if (count > 0)  {
        mon->next_count++;
        sem->V();

        mon->next->P();
        mon->next_count--;
    }

    (void) interrupt->SetLevel(oldLevel);
}
```

Again, the initializer lists are my stylistic preference. The Wait and Signal functions are as they were with the calls to passed pointers replaced to calls to the object's monitor pointer.

Here is the code for the Ring Buffer:

```
Ring::Ring(char *debugName, int sz)
    : Monitor(debugName), size(sz), in(0), out(0), buffer(new slot[size]),
```

```
    current(0),
    notfull(new Condition_H("notfull", (Monitor *) this)),
    notempty(new Condition_H("notempty", (Monitor *) this)) {

    if (size < 1) {
        fprintf(stderr, "Error: Ring: size %d too small\n", sz);
        exit(1);
    }
}

Ring::~Ring()
{
    // Some compilers and books tell you to write this as:
    //     delete [size] stack;
    // but apparently G++ doesn't like that.

    delete [] buffer;
    delete notfull;
    delete notempty;
}

void Ring::Put(slot *message) {
    Entry();
    if (Full()) {
        notfull->Wait();
    }
    buffer[in].thread_id = message->thread_id;
    buffer[in].value = message->value;
    in = (in + 1) % size;
    current++; // one more in ring.
    notempty->Signal();
    Exit();
}

void Ring::Get(slot *message) {
    Entry();
    if (Empty()) {
        notempty->Wait();
    }
    message->thread_id = buffer[out].thread_id;
    message->value = buffer[out].value;
    out = (out + 1) % size;
    current--; // one less in ring.
    notfull->Signal();
    Exit();
}

int Ring::Empty() {
```

```
    return current == 0;
}

int Ring::Full() {
    return current == size;
}
```

I implemented the Empty and Full calls and used them in Get and Put as described above. The constructor and destructor were updated and an initializer lists was used in the constructor. The Put call as it was is surrounded by calls to the two condition variables as appropriate and calls to Entry and Exit for its monitor superclass.

Other modifications include removal of the global mutex and next semaphores that were replaced with the monitor.

Q. 4 Here follows testing data with three separate configurations.

Q. 4.1 Here follows a test using Configuration 1 with the following qunatities:

- buffer size: 2
- number of producers: 4
- number of messages per producer: 3
- number of consumers: 2

First, I compiled a new `nachos` binary with the above configuration as follows:

```
$ make
...
ln -sf arch/unknown-i386-linux/bin/nachos nachos
```

Then, I ran nachos with a random seed of 26 by using the command `./nachos -rs 26`. The output is recorded in the following:

```
$ ./nachos -rs 26
No threads ready or runnable, and no pending interrupts.
Assuming the program completed.
Machine halting!

Ticks: total 1325, idle 165, system 1160, 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...
$ ls tmp_*
```

```
tmp_0  tmp_1
$ cat tmp_0
producer id --> 0; Message number --> 0;
producer id --> 1; Message number --> 0;
producer id --> 3; Message number --> 0;
producer id --> 0; Message number --> 1;
producer id --> 3; Message number --> 2;
$ cat tmp_1
producer id --> 1; Message number --> 1;
producer id --> 2; Message number --> 0;
producer id --> 3; Message number --> 1;
producer id --> 2; Message number --> 1;
producer id --> 0; Message number --> 2;
producer id --> 1; Message number --> 2;
producer id --> 2; Message number --> 2;
```

As you can see from the outputs recorded above, this first configuration's test is successful.

Q. 4.2 Here follows a test using Configuration 2 with the following qunatities:

- buffer size: 5
- number of producers: 3
- number of messages per producer: 4
- number of consumers: 3

First, I compiled a new `nachos` binary with the above configuration as follows:

```
$ make
...
ln -sf arch/unknown-i386-linux/bin/nachos nachos
```

Then, I ran nachos with a random seed of 26 by using the command `./nachos -rs 26`. The output is recorded in the following:

```
$ ./nachos -rs 26
No threads ready or runnable, and no pending interrupts.
Assuming the program completed.
Machine halting!

Ticks: total 1325, idle 125, system 1200, 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...
$ ls tmp_*
tmp_0  tmp_1  tmp_2
$ cat tmp_0
producer id --> 0; Message number --> 0;
producer id --> 1; Message number --> 0;
producer id --> 0; Message number --> 1;
producer id --> 2; Message number --> 1;
producer id --> 2; Message number --> 2;
producer id --> 0; Message number --> 3;
$ cat tmp_1
producer id --> 1; Message number --> 1;
producer id --> 1; Message number --> 2;
producer id --> 2; Message number --> 3;
$ cat tmp_2
producer id --> 2; Message number --> 0;
producer id --> 1; Message number --> 3;
producer id --> 0; Message number --> 2;
```

As you can see from the outputs recorded above, this first configuration's test is successful.

In summary, you can see that these tests show my code is functioning properly.