A_9

Assignment IX

Connor Taffe. T no. 3742

April 21st, 2015

1 Details of Lab 4-6

Here follows an account of laboratory 4, section 6. This section follows the original lab specification to detail a report, the "Report of Lab 4-6" is found in section 2.

Q. 1 The *Mesa-style* condition variables as first described in "Experience with Processes and Monitors in Mesa" can be described in the same way as the provided Hoare-style condition variable as follows:

- `Wait()` method

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

- `Signal()` method

```
if (count > 0) sem->V();
```

Notice the lack of a `next-count` and `next` variable. This is so because Mesa-style condition variables do not preserve a relationship between signaller and waiter.

Q. 2 After implementing a new monitor, ring, and Mesa-style condition variable, I tested my results as follows:

```
$ make
...
ln -sf arch/unknown-i386-linux/bin/nachos nachos
$ ./nachos
No threads ready or runnable, and no pending interrupts.
Assuming the program completed.
Machine halting!

Ticks: total 1150, idle 0, system 1150, 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 --> 1; Message number --> 1;
producer id --> 2; Message number --> 0;
producer id --> 3; Message number --> 0;
producer id --> 3; Message number --> 2;
$ cat tmp_1
producer id --> 0; Message number --> 1;
producer id --> 2; Message number --> 1;
producer id --> 0; Message number --> 2;
producer id --> 3; Message number --> 1;
producer id --> 1; Message number --> 2;
producer id --> 2; Message number --> 2;
```

As you can see, the messages are ordered numerically by producer, and there are no repeated messages. This is a sign of a working program.

2 Report of Lab 4-6

Q. 1 I will now describe my analysis and design (including the algorithm) of Mesa style conditional variables (i.e. Section 2 of Lab4-6) using semaphores.

My first submission was based on an incorrect assumption. That is that "Experience with Processes and Monitors in Mesa," like the Hoare paper, specified a new monitor implementation from the ground up. Unfortunately, it was meant to be taken in context as an update over the existing Hoare monitor implementation. This aside, the first submission detailed the simpler signal method well. The main idea is to simplify and basically asynchonize the siganlling of waiting processes.

Q. 2 I will now submit the implementation, that is, the relevant source codes of the monitor class, ring class, Condition M class, and other related codes

Here follows the Mesa style condition variable code:

```
class Condition_M {
  public:
    Condition_M(char* debugName, Monitor *m);   // initialize condition to
                                                // "no one waiting"
    ~Condition_M();                             // deallocate the condition
    char* getName() { return (name); }

    void Wait();
    void Signal();

  private:
    char* name;
    // plus some other stuff you'll need to define

    Semaphore *sem; // semaphore for the waiting threads;
    int  count;     // the number of waiting threads;

    Monitor* mon;   // the pointer to the Monitor to which this condition
                    // variable belongs
};

// condition variables in Mesa's style
Condition_M::Condition_M(char *debugName, Monitor *m)
    : name(debugName),
    sem(new Semaphore(name, 0)), // uses the same name as cond var
    count(0),
    mon(m) {}

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

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

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

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


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

    if (count > 0) sem->V();

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

Here follows the monitor class:

```
class Monitor {
public:
    Monitor(char *debugName);   // default contructuctor
    ~Monitor();               // default destructor

    // const function attribute guarantees no side effects.
    char *getName() const { return name; }
private:
    char *name;
public:
    Semaphore* mutex;   // the semaphore for mutual exclusion
    Semaphore* next;    //the semaphore to hold internal threads which are
                        // blocked after waking up some threads in Signal()
                        // of a condition variable

    int next_count;     // the number of threads blocked in Semaphore next
protected:
    void Entry();       // The entry for critical section of monitor functions
    void Exit();        // The exit for critical section of monitor functions
};

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;
}

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

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

Note that the semaphore `next` and integer `next-count` were removed from the class completely.

Here follows the ring class's changed portions:

```
// class of the slot in the ring-buffer
class slot {
    public:
    slot(int id, int number);
    slot() { thread_id = 0; value = 0;};

    int thread_id;
    int value;
 };


class Ring : public Monitor{
  public:
    Ring(char *debugName, int sz);    // Constructor:  initialize variables,
                                      // allocate space.
    ~Ring();         // Destructor:   deallocate space allocated above.


    void Put(slot *message); // Put a message the next empty slot.

    void Get(slot *message); // Get a message from the next  full slot.

    int Full();       // Returns non-0 if the ring is full, 0 otherwise.
    int Empty();      // Returns non-0 if the ring is empty, 0 otherwise.

  private:
    int size;           // The size of the ring buffer.
    int in, out;        // Index of
    slot *buffer;       // A pointer to an array for the ring buffer.
    int current;        // the current number of full slots in the buffer

    Condition_M *notfull;   // condition variable to wait until not full
    Condition_M *notempty;  // condition variable to wait until not empty
};

slot::slot(int id, int number)
{
    thread_id = id;
    value = number;
}

Ring::Ring(char *debugName, int sz)
    : Monitor(debugName), size(sz), in(0), out(0), buffer(new slot[size]),
    current(0),
    notfull(new Condition_M("notfull", (Monitor *) this)),
    notempty(new Condition_M("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();
    while (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();
    while (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;
}
```

Note the only change in this class is the while loop on Waits.

Q. 3 Then I submitted the testing results (i.e. the contents of the tmp x files) for the following configurations using random seed 96.

- Buffer Size: 2
- Number of Producers: 4
- Number of Messages per Producer: 3
- Number of Consumers: 2

The following is the output for configuration one:

```
$ make
...
ln -sf arch/unknown-i386-linux/bin/nachos nachos
$ ./nachos -rs 96
No threads ready or runnable, and no pending interrupts.
Assuming the program completed.
Machine halting!

Ticks: total 1297, idle 37, system 1260, 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 --> 0; Message number --> 2;
producer id --> 2; Message number --> 1;
producer id --> 3; Message number --> 1;
producer id --> 3; Message number --> 2;
$ cat tmp_1
producer id --> 0; Message number --> 1;
producer id --> 2; Message number --> 0;
producer id --> 1; Message number --> 0;
producer id --> 3; Message number --> 0;
producer id --> 1; Message number --> 1;
producer id --> 2; Message number --> 2;
producer id --> 1; Message number --> 2;
```

- Buffer Size: 5
- Number of Producers: 3
- Number of Messages per Producer: 4
- Number of Consumers: 3

The following is the output for configuration two:

```
$ make
...
ln -sf arch/unknown-i386-linux/bin/nachos nachos
$ ./nachos -rs 96
No threads ready or runnable, and no pending interrupts.
Assuming the program completed.
Machine halting!

Ticks: total 1297, idle 57, system 1240, 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 --> 0; Message number --> 1;
producer id --> 1; Message number --> 0;
producer id --> 2; Message number --> 0;
producer id --> 0; Message number --> 2;
producer id --> 0; Message number --> 3;
producer id --> 1; Message number --> 2;
$ cat tmp_1
producer id --> 2; Message number --> 1;
producer id --> 1; Message number --> 3;
$ cat tmp_2
producer id --> 1; Message number --> 1;
producer id --> 2; Message number --> 2;
producer id --> 2; Message number --> 3;
```

The order of the outputs and the number of occurences of a number per producer equaling one shows that this program is operational.