Ellie Makin (erm67)

Concurrent and Distributed Systems supervision 1

Semaphores, generalised producer-consumer, and priorities

Q0 Semaphores

a. Initialising to 1 is the classic case when there is just one resource to be allocated, such as a lock. An arbitrary n can be used to allocate n resources. Initialising to 0 is slightly different: it can be used so that some threads only continue after another thread signals.

b.

```
// Thread 1:
wait(sem1);
sleep(1);
wait(sem2);
// do something
signal(sem2);
signal(sem1);

// Thread 2:
wait(sem2);
sleep(1);
wait(sem1);
// do something else
signal(sem1);
signal(sem2);
```

c. Suppose you have a producer-consumer situation like this:

```
int nItems = 0;

void producer() {
    while (true) {
        produce_item();
        if (nItems == BUFFER_SIZE)
            wait(sem);
        push_item();
        nItems++;
        if(nItems == 1)
            signal(sem);
    }
}

void consumer() {
    while (true) {
        if (nItems == 0)
```

```
            wait(sem);
        pop_item();
        nItems--;
        if (nItems == BUFFER_SIZE - 1)
            signal(sem);
        consume_item();
    }
}
```

If the producer finds that the buffer is full, and is about to run `wait` before being interrupted by the consumer, then the consumer will `signal` before the producer has started waiting. Then when the producer resumes, it will run `wait`, but will never be signalled, and the consumer will continue to consume items until the buffer is empty, when it will also wait, putting the program into deadlock.

d. If `guard` is removed, it becomes possible for two producers to write to `buffer[in]` before `in` is incremented, with the result that one of the items written is lost.

e. If the expected wait time is low, it may be more efficient for a thread to spin rather than block, e.g. if a critical section is only a few instructions. Generally, it is preferable to spin if the expected wait time is less than the time it would take for the operating system (or userspace threading library) to stop and then restart the thread.

f. If the `test_and_set` function for `wait` was not atomic, at would be possible for two threads to test the semaphore simultaneously, which would allow more than one thread to resume at once. The opposite problem arises with `signal` (if the semaphore is a counting semaphore rather than a binary one), since then one thread may read the semaphore before another has finished writing back an incremented value, resulting in only one increment rather than two.

Q1 Priority and work distribution

a.

```
int buffer[N]; int in = 0, out = 0;
int nConsumers = 10;
spaces = new Semaphore(N);
spaces = new Semaphore(N);
// cycle through an array of semaphores, one for each consumer
items  = new Array(Semaphore(0), nConsumers);
guard  = new Semaphore(1);

int nextConsumer = 0;

void producer() {
    while(true) {
        item = produce();
        wait(spaces);
        wait(guard);
        buffer[in] = item;
        in = (in + 1) % N;
        nextConsumer = (nextConsumer + 1) % nConsumers;
        signal(guard);
        signal(items[nextConsumer]);
```

```
    }
}

void consumer(int id) {
    while(true) {
        wait(items[id]);
        wait(guard);
        item = buffer[out];
        out = (out+1) % N;
        signal(guard);
        signal(spaces);
        consume(item);
    }
}
```

b.

```
int buffer[N]; int in = 0, out = 0;
int nConsumers = 10;
spaces = new Semaphore(N);
// array of semaphores, one for each consumer
items  = new Array(Semaphore(0), nConsumers);
guard  = new Semaphore(1);

// a queue of thread ids for each priority level
queueA = new Queue<int>;
queueB = new Queue<int>;
queueC = new Queue<int>;

void producer() {
    int nextConsumer = 0;
    while(true) {
        item = produce();
        wait(spaces);
        wait(guard);
        buffer[in] = item;
        in = (in + 1) % N;
        signal(guard);
        while (true) {
            if (queueA.isNotEmpty()):
                nextConsumer = queueA.dequeue();
                break;
            elif (queueB.isNotEmpty()):
                nextConsumer = queueB.dequeue();
                break;
            elif (queueC.isNotEmpty()):
                nextConsumer = queueC.dequeue();
                break;
        }
        signal(items[nextConsumer]);
    }
}

void consumer(int id, Queue<int> queue) {
    queue.enqueue(id);
    while(true) {
        wait(items[id]);
```

```
        wait(guard);
        item = buffer[out];
        out = (out+1) % N;
        signal(guard);
        signal(spaces);
        queue.enqueue(id);
        consume(item);
    }
}
```

c.

```
int buffer[N]; int in = 0, out = 0;
int nConsumers = 10;
spaces = new Semaphore(N);
// array of semaphores, one for each consumer
items  = new Array(Semaphore(0), nConsumers);
guard  = new Semaphore(1);
history = new Stack<int>();

void producer() {
    int nextConsumer = 0;
    while(true) {
        item = produce();
        wait(spaces);
        wait(guard);
        buffer[in] = item;
        in = (in + 1) % N;
        signal(guard);
        nextConsumer = -1
        while (nextConsumer == -1)
            nextConsumer = history.pop();
        signal(items[nextConsumer]);
    }
}

void consumer(int id) {
    while(true) {
        wait(items[id]);
        wait(guard);
        item = buffer[out];
        out = (out+1) % N;
        signal(guard);
        signal(spaces);
        consume(item);
    }
}
```

d. Round-robin distribution may improve performance if it is important that each thread gets an even distribution of the workload, since the other techniques may starve the lower priority threads. Prioritised work distribution may improve performance if some threads run on cores with higher throughput. MRU improves performance if the `consume` function involves memory accesses that may be cache-dependent, so accessing the same memory again would be preferable.

Q2 Contention

a.

```
int buffer[N]; int in = 0, out = 0;
spaces = new Semaphore(N);
items  = new Semaphore(0);
guardIn  = new Semaphore(1);
guardOut = new Semaphore(1);

void producer() {
    while(true) {
        item = produce();
        wait(spaces);
        wait(guardIn);
        buffer[in] = item;
        in = (in + 1) % N;
        signal(guardIn);
        signal(items);
    }
}

void consumer(int id) {
    while(true) {
        wait(items);
        wait(guardOut);
        item = buffer[out];
        out = (out+1) % N;
        signal(guardOut);
        signal(spaces);
        consume(item);
    }
}
```

b. We can consider what would happen if there was a context switch at various stages of execution, and see if there are any cases that cause a hazard.

Q3 Priority inversion

a. I don't understand - is `guard` not being used as a mutex already?

b. This problem could be addressed by using the prioritised work distribution described in Q1.

c. This problem could be mitigated by not allowing the low-priority threads to be interrupted.