1 Problems with Semaphores

- Errors in concurrent programs are difficult to find (especially if they relate to concurrency issues).
  - We need to consider interleaving of different threads.
  - The control-flow is non-deterministic.
  - We don't know exactly, what happens.
  - Errors are not reproducable.
- Incorrect use of semaphores results in difficult errors.
- One misbehaved thread is enough to destroy everything.

  Q: Which of the above examples destroys mutual exclusion?

  Q: Which of the examples may lead to a deadlock?

- We accidentally exchange mutex.P() and mutex.V().

```
mutex.V();
        criticalSection
mutex.P();
```

- We accidentally use one incorrect:

```
mutex.P();
        criticalSection
mutex.P();
```

- or

```
mutex.V();
        criticalSection
mutex.V();
```

- We accidentally forget one:

```
mutex.P();
        criticalSection
```

- or

```
        criticalSection
mutex.V();
```

2 Monitors

- A *monitor* presents a set of programmer-defined operations, that are provided with mutal exclusion.
- In Java pseudo-code:

```
monitor name {
        // variable declarations
        public entry void m1() { ... }
        public entry void m2() { ... }
}
```

- In the monitor we can access only monitor variables, parameters to the call, and local variables.
- All procedures defined in the monitor are accessed exclusively.
- Only one thread can be active in the monitor.
- If m1() is called and then m2() is called, before m1 returns, m2 will block.

3 Conditions

- We have special variables of type `condition`.
- If we have a defintion

  `condition x,y;`

  then we have two operations on them available:
- `x.wait()`. This *suspends* the current thread, until another thread invokes
- `x.signal()`. This operation *resumes* exactly one thread.
- If we suspend a thread it will be blocked until it is resumed.

4 signal and wait

- Suppose P calls x.signal and Q is waiting for x.
- Now both threads could do something in the monitor, but of course we cannot allow both to continue!
- We have two possiblities:
  - Signal-and-Wait: P has to wait.
  - Signal-and-Continue: Q has to wait.
- Both methods have advantages.
- P is already executing, so Signal-and-Continue seems more reasonable.
- However, by the time Q is resumed, the logical condition may no longer hold.

5 Philosophers again

- We distinguish three states for a philosopher, THINKING, HUNGRY, and EATING.
- We store the states in an array **int**[ ] state = **new** **int**[5];
- We set state[i]=EATING only if philosopher i is HUNGRY and both neigbors are not EATING.
- For that we declare condition[ ] self = **new** condition[5] .
- A philosopher i, who wants to eat (in state HUNGRY) waits on self[i].
- The routine testCondition(i) checks whether the condition is fulfilled. If that is the case it sets the state to EATING and calls signal.

```
monitor diningPhilosophers {
    int[ ] state = new int[5];
    static final int THINKING = 0;
    static final int HUNGRY = 1;
    static final int EATING = 2;
    condition[ ] self = new condition[5];

    public diningPhilosophers {
        for (int i = 0; i < 5; i ++)
            state[i] = THINKING;
    }

    public entry void pickUp(int i) {
        state[i] = HUNGRY;
        testCondition(i);
        if (state[i] != EATING)
            self[i].wait;
    }
```

```
        public entry void putDown(int i) {
                state[i] = THINKING;
                testCondition((i + 4) % 5);
                testCondition((i + 4) % 5);
        }

        private testCondition(int i) {
                if (state[(i+4) %5 ] != EATING
                        && state[i] = HUNGRY
                        && state[(i+1) %5 ] != EATING) {
                        state[i] = EATING;
                        self[i].signal;
                }
        }
}
```

6 Is it working?

A philosopher now does

```
dp.pickUp(i);
eat();
dp.putDown(i);
```

If this is the only way dp.pickUp(i), eat(), and dp.putDown(i) are called.

- No two neighbors can eat simultaneously.
- No deadlock can occur.
- However, a philosopher can starve.

Q: How can we assure ourselves of this?

7 Solution

Mutual exclusion:

- Before we set state[i] = EATING we check, that the two neighbors are not eating.
- No one could modify the states of the neighbors in between because we do it in one monitor routine.

Deadlock:

- A lock is never kept in a loop.
- If a thread T is waiting long for a thread T', then T is a HUNGRY philosopher and T' is EATING.
- Then T' does not wait.

Starving:

- If the philosophers 1 and 3 eat a lot, philosopher 2 might stay hungry and starve.

8 Java Synchronization

The original bounded buffer solution had two problems:

- The race condition on the variable `count`
- If one of the threads had to wait, they used busy-waiting in a loop.

We want to find a Java solution, that resolves both problems.

Java synchronisation works almost like a monitor.

9 Locks

- A *lock* can be imagined as a door with a key.
- *acquiring the lock* means getting the key.
- *releasing the lock* means putting the key back.
- a lock is *available* if the key is there for taking.
- The door is always locked, so only the key-owner can pass it.
- Each object in Java has such a lock associated with it.

10 synchronized

Java has a keyword **synchronized**, which is used for synchronization purposes.

- Each object in Java has a lock associated with it.
- In a normal method call the lock is ignored.
- If a method is declared **synchronized**, calling the method requires acquiring that lock.
- There is an *entry set* for the lock of an object.
- It represents the threads waiting for the lock to become available.
- If the lock is available at a call, the method acquires the lock and continues.
- On return it releases the lock again.
- If the lock is not available at a call, the thread is put in the entry set.
- If a lock is released and the entry set is non-empty, one of the threads in the entry set acquires the lock and continues.

11 Bounded Buffers again

We use the lock of the bounded buffer object.

```
synchronized void add(Object o) {
    while (count == BUFFER_SIZE)
        yield();
    count = count + 1;
    buffer[in] = o;
    in = (in + 1) % BUFFER_SIZE;
}
synchronized Object remove(Object o) {
    while (count == 0)
        yield();
    count = count − 1;
    Object o = buffer[out];
    out = (out + 1) % BUFFER_SIZE;
    return o;
}
```

This solves the race condition on the variable count. Q: What would happen?

12 wait and notify

Java provides two method-calls, wait() and notify(), that are very similar to wait() and signal() in the monitor.

- Every object also has a *wait set*.
- If a thread calls wait(),
  - It releases the lock for the object.
  - The state of the thread is Blocked.
  - The thread is placed in the wait set for the object.
- If a thread calls notify()
  - An arbitrary thread T from the wait set is moved to its entry set.
  - The state of T becomes Runnable

13 Solution

```
synchronized void add(Object o) {
    while (count == BUFFER_SIZE) {
        try {
            wait();
        catch (InterruptedException e) { }
    }
    count = count + 1;
    buffer[in] = o;
    in = (in + 1) % BUFFER_SIZE;
    notify();
}
```

```
synchronized Object remove() {
    while (count == 0) {
        try {
            wait();
        catch (InterruptedException e) { }
    }
    count = count – 1;
    Object o = buffer[out];
    out = (out + 1) % BUFFER_SIZE;
    notify();
    return o;
}
```

For now we don't care about the InterruptedException.

14 An example run

- We assume, that the buffer is full, and the lock available.
- The producer calls `add()`.
  - The lock is available and it enters the method.
  - It sees, that the buffer is full and calls `wait`.
  - This releases the lock, makes the producer blocked and puts the producer in the wait set.
- The consumer ultimately calls `remove()`.
  - The lock is available and it enters the method.
  - It removes an item from the buffer.
  - it calls `notify()`.
  - This moves the producer from the wait set to the entry set and makes it runnable.
  - The consumer exits `remove()` and releases the lock.

- When the producer runs again it tries to acquire the lock.
  - If it succeeds, it continues from the wait() call.
  - It checks the condition of the while-loop, succeeds, and adds an item to the buffer.
  - notify() is ignored, since there is nothing in the wait set.

15 Readers-Writers again

- readerCount tells us, how many readers are in the database.
- dbWriting tells, whether we have a writers in the database

```
private int readerCount;
private boolean dbWriting;

public Database() {
        readerCount = 0;
        dbWriting = false;
}
```

```
public synchronized void startRead() {
        while (dbWriting == true) {
                try {
                        wait();
                catch (InterruptedException e) { }
        }
        readerCount ++;
}

public synchronized void endRead() {
        readerCount −−;
        notifyAll();
}
```

```
public synchronized void startWrite() {
        while (dbReading == true || dbWriting == true) {
                try {
                        wait();
                catch (InterruptedException e) { }
        }
        dbWriting = true;
}

public synchronized void endWrite() {
        dbWriting = false;
        notifyAll();
}
```

16 notifyAll

- Before notify() took a thread from the wait set to the entry set.
- Now notifyAll() takes all the threads from the wait set to the entry set.

Why do we need that?

- Before we had always exactly one thread in the wait set of an object.
- Here we may have many threads in one wait set.
- Also we have different conditions to wait for.

Q: What would happen if we replace one of them by notify

If notify works then notifyAll works as well, but it may be less efficient.

17 Block synchronization

- Java not only allows to synchronize complete methods, but also blocks.
- The following is equivalent to declaring someMethod synchronized.

```
public void someMethod() {
    synchronized(this) {
        // body
    }
}
```

- **synchronized**(o) { ... } tries to acquire the lock of o. After acquiring it, it executes the block and releases the lock.
- Inside such a block we may use o.wait() and o.notify().
- This allows us to make synchronization more fine-grained.
  - We have less blocking.
  - More work synchronizing.
  - We often have to find this trade-off.

18 Other things to be aware of

- A thread that owns the lock for an object can enter other synchronized methods or blocks for that object.
- A thread can nest synchronized method invocations for different objects. So it can own locks for more than one object.
- If a method is not declared synchronized it can be called, even if another thread is executing a synchronized method.
- If the wait set is empty, a call to `notify()` has no effect.