1 Synchronisation

- The bounded buffer problem.
- Critical regions, mutual exclusion.
- Two-task solutions
- Semaphores
- Other classical problems
- Monitors
- Java synchronization

We will consider threads only.

- From a synchronization perspective, the difficulty is the same.
- Communication is much easier with shared memory.

2 Bounded Buffers

- We want to implement a buffer of a given size.
- There is a method **void** add(Object o), which adds an Object to the buffer.
- There is a method Object remove(), which takes an Object out of the buffer.

- We assume that two threads run at the same time.
- One thread, the *producer* calls add repeatedly.
- The other thread, the *consumer* calls remove repeatedly.

3 Bounded Buffers (2)

- buffer is an array that keeps the objects.
- in is the place, where to put the next element.
- out is the place, where to get the next element.
- count is the number of elements in the buffer.

```
class BoundedBuffer {
    static final int BUFFER_SIZE = 128;
    int count, in , out;
    Object[ ] buffer;

    public BoundedBuffer() {
        count = 0;
        in = 0;
        out = 0;
        buffer = new Object[BUFFER_SIZE];
    }
```



```
    void add(Object o) {
        while (count == BUFFER_SIZE)
            ;
        count = count + 1;
        buffer[in] = o;
        in = (in + 1)
    }

    Object remove() {
        while (count == 0)
            ;
        count = count − 1;
        Object o = buffer[out];
        out = (out + 1)
    }
```

Q: What is the problem with this solution?

4 The Problem

- The increment and decrement of `count` might happen concurrently.
- If we have bad luck the increment and decrement together make one increment.
- They should cancel.
- This is unacceptable.

This situation is called a *race condition*.

A side-remark:

- `add` waits until the buffer is not full, then adds.
- `remove` waits until the buffer is non-empty, then removes.
- Waiting in a loop like this is called *busy waiting*.

5 Critical Section

- When multiple threads operate on shared data, we have to control access to that data.
- Some parts of the code of the different threads shouldn't be executed concurrently.
- We call these parts *critical sections*.
- The idea is, that we make the execution of critical sections *exclusive in time*.
- Never should two threads be in a critical section at the same time.

Q: What are the critical sections in the bounded buffer example.

6 Requirements

A programming solution two the critical section problem must satisfy the following three requirements:

- *Mutual Exclusion*: If a thread T is in its critical section, no other thread is in its critical section.
- *Progress*: If no thread is executing in its critical section and some threads want to enter their critical sections, then one of them will ultimately be allowed to enter.
- *Bounded Waiting*: There exists a bound on the number of times, that other threads are allowed to enter their critical sections after a thread has made a request to enter a critical section.

7 Simple-minded Solutions

Allow any thread to enter its critical section at any time?

Q: Which requirement would not be met?

Never allow any thread to enter its critical section?

Q: Which requirement would not be met?

Q: Can you think of something which does not meet bounded waiting?

8 Two Threads

We first restrict the problem to two threads.

- The threads are called T_0 and T_1.
- For thread T_i, the other thread is T_{1-i}.
- Before entering a critical section, a thread must call enterCritical(**int** t) with its own number.
- After leaving a critical section, a thread must call leaveCritical(**int** t) with its own number.

```
class MutualExclusion {
        public MutualExclusion() { ... }
        public void enterCritical(int t) { ... }
        public void leaveCritical(int t) { ... }
}
```

The goal is to find an implementation for MutualExclusion that guaranties mutual exclusion, progress, and bounded waiting.


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


9 Assumptions

We assume, that the basic machine instructions load and store are executed atomically.

If a load and a store are executed concurrently, then the load will get the old or the new value, but not a mixture.

Q: Is it also reasonable to assume, that an increment operation incr is executed atomically?


```
class MutualExclusion1 {
        private volatile int turn;
        public MutualExclusion1() {
                turn = 0;
        }
        public void enterCritical(int t) {
                while (turn != t)
                        Thread.yield();
        }
        public void leaveCritical(int t) {
                turn = 1 − t;
        }
}
```

- Thread.yield() is a form of non-busy waiting.
- **volatile** tells the compiler not keep turn in a register.

```
class MutualExclusion2 {
    private volatile boolean[2] flag = new boolean[2];
    private volatile int turn;

    public MutualExclusion2() {
        flag[0] = false;
        flag[1] = false;
    }
    public void enterCritical(int t) {
        int other = 1 – t;
        flag[t] = true;
        while (flag[other] == true)
            Thread.yield();
    }
    public void leaveCritical(int t) {
        flag[t] = false;
    }
}
```



```
class MutualExclusion3 {
    private volatile boolean[2] flag = new boolean[2];

    public MutualExclusion3() {
        flag[0] = false;
        flag[1] = false;
        turn = 0;
    }
    public void enterCritical(int t) {
        int other = 1 – t;
        flag[t] = true;
        turn = other;
        while (flag[other] == true && turn == other)
            Thread.yield();
    }
    public void leaveCritical(int t) {
        flag[t] = false;
    }
}
```


10 Semaphores

- The above solution is not easy to generalize to complex problems.
- To overcome this, we use a synchronisation tool called *semaphore*.
- A semaphore is an integer variable, that is only accessed
  - At initialization
  - Through operation P
  - Through operation V

```
P(S) {
    while (S ≤ 0)
        ;
    S —
}
V(S) {
    S ++;
}
```


11 Semaphore Usage

- Often we use a semaphore only with values 0 and 1.
- We call it a *binary semaphores*.
- We can use a binary semaphore to guard critical sections. (similar to enterCritical and leaveCritical).

```
P(S);
  criticalSection
V(S);
```

- Before entering the critical section S is set to 0.
- Now, no other thread can enter a critical section guarded by S.
- After finishing the critical section, S is set to 1 again.
- Now the next thread can enter its critical section.

12 Using Counting Semaphores

- Semaphores, where we also use higher values are called *counting sempaphores*
- A counting semaphore is useful, if we have a finite number of resources.
- The semaphore is initialized to the number of resources.
- If a thread wishes to use a resource it calls P.
- If it has released the resource again it calls V.
- So S is always the number of available resources.

13 Bounded Buffer with Semaphores

```
class BoundedBuffer {
    static final int BUFFER_SIZE = 128;
    int count, in , out;
    Object[ ] buffer;
    Semaphore mutex, empty, full;

    public BoundedBuffer() {
        count = 0;
        in = 0;
        out = 0;
        buffer = new Object[BUFFER_SIZE];
        mutex = new Semaphore(1);
        empty = new Semaphore(BUFFER_SIZE);
        full = new Semaphore(0);
    }
```



```
    void add(Object o) {
        empty.P();
        mutex.P();

        count = count + 1;
        buffer[in] = o;
        in = (in + 1)

        mutex.V();
        full.V();
    }
```



```
    Object remove() {
        full.P();
        mutex.P();

        count = count − 1;
        Object o = buffer[out];
        out = (out + 1)

        mutex.V();
        empty.V();
    }
}
```

14 Inside a Semaphore

- Modifications to the integer value must be executed indivisibly.
- Moreover, testing S and decrementing it must also be executed without interruption.
- We can make the critical section more explicit.

```
P(S) {
    boolean finished = false;
    while (!finished) {
        if (S > 0) {
            S --;
            finished = true;
        }
    }
}

V(S) {
    S ++;
}
```


15 Implementing a Semaphore

- We could implement semaphores using a two task solutions.
- The main disadvantage of these solutions and the semaphore definition given above is, that they require *busy waiting*
- If one thread has to wait for another thread to leave the critical section, it loops.
- This is a waste of CPU-cycles.
- We call a semaphore with busy waiting a *spinlock*.
  - Spinlocks may be useful in a multiprocessor environment.
  - They don't need a context switch.
  - If the waiting time is short, waiting might be more efficient than a context switch.
- To overcome the need for busy waiting we modify P and S.
- We define a semaphore as an integer value and a list of processes.


```
P(S) {
    value --;
    if (value < 0) {
        add this thread to list for semaphore
        block;
    }
}

V(S) {
    value ++;
    if (value ≤ 0)
        remove a thread T from list for semaphore
        wakeup (T);
}
```

- If a thread calls P, when value ≤ 0, it enters itself into the list and blocks.
- If another thread calls V and value ≤ 0 (threads waiting), it wakes up one of the threads.

16 More Implementation Issues

- The critical aspect is to execute P and V well, that is respecting their critical zones.
- In a one processor environment, we can prohibit interrupts in semaphores.
- In a multiprocessor environment prohibiting interrupts does not work.
- Instructions from different processors can be arbitraryly interleaved.
- If the hardware doesn't have special instructions, we have to implement one of the two-task solutions.

Q: We have again busy waiting. Is this a problem here?

17 Not a Problem Here

- We only wait for another thread to leave its critical section within a semaphore.
- These critical sections are very short.
- Busy waiting occurs rarely, and then for a short time.
- A critical section of an application program might be very long.
- But we do not need to wait for that.

18 Deadlocks

- When two or more threads are waiting for an event that can only be caused by them, they will wait forever.

A() {	B() {
S.P();	Q.P();
Q.P();	S.P();
...	...
Q.V();	S.V();
S.V();	Q.V();
}	}

- Assume A runs, and calls S.P(), returns.
- Then B runs, and calls Q.P(), returns.
- None of the two will come any further.
- Such a situation is called a *deadlock*

19 The Readers-Writers Problem

- A data base is to be shared among several threads.
- Some threads may want to read from the data base.
- Some threads may want to write to the data base.
- We call these threads *readers* and *writers*
- Two readers accesing the data base simultaneously do no harm.
- However, if a writer and some other thread access the data base at the same time, bad things may happen.
- To ensure that this doesn't happen we require that writers have exclusive access to the data base.
- This is called the *readers-writers problem*

20 Variants of the Readers-Writers Problem

- No reader will be kept waiting until a writer has obtained access to the data base.
- If a writer is waiting, no reader gets access, until the writer is finished.
- First come, first serve.

Q: What are the advantages/disadvantages of the variants?

21 Starvation

- We speak of *starvation* when one of the threads is kept in a semaphore forever, although no deadlock exists.
- If we always have writers waiting, readers will starve in the example, where writers have priority.
- This happens, although the semaphore will change from 0 to 1 and back all the time.
- In a deadlock, there is at least one resource, that doesn't get released any more.


```
class database {
    Semaphore mutex;
    Semaphore db;
    int readerCount;

    public database () {
        readerCount = 0;
        mutex = new Semaphore(1);
        db = new Semaphore(1);
    }

    void startRead() {
        mutex.P();
        readerCount ++;
        if (readerCount == 1)
            db.P();
        mutex.V();
    }
```



```
    void endRead() {
        mutex.P();
        readerCount —;
        if (readerCount == 0)
            db.V();
        mutex.V();
    }

    void startWrite() {
        db.P();
    }
    void endWrite() {
        db.V();
    }
}
```


22 The Dining Philosopher Problem

- Five philosophers spend their lives thinking and eating.
- The philosophers share a round table.
- Each philosopher has its own plate, and their is a bowl of rice in the middle.
- There are five single chopsticks, one between each two plates.
- If a philosopher wants to eat, he/she tries to pick the chop sticks to his/her left and right (one after another!).
- After eating he/she puts the chopsticks back again.

```
class philosophers {
    Semaphore[5] sticks;

    startEat(int p) {
        sticks[i].P();
        sticks[(i + 1) % 5].P()
    }

    stopEat(int p) {
        sticks[i].V();
        sticks[(i + 1) % 5].V()
    }
}
```

Q: What is the problem?

23 The Dining Philosopher Problem(2)

If every philosopher picks the left chop stick, we have= a deadlock!

What can we do

- Allow at most 4 philosophers sitting simultaneously at a table.
- Allow a philosopher to pick up the chopsticks only if both are available.
- Use an asymmetric solution (odd philosophers take the left chopstick first, even philosophers the right one).
- Use a single semaphore. Note, that this might be a performance bottleneck.