1 Readers-Writers again

- readerCount tells us, how many readers are in the database.
- dbWriting tells, whether we have 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 (readerCount != 0 || dbWriting == true) {
        try {
            wait();
        catch (InterruptedException e) { }
    }
    dbWriting = true;
}

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

2 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 usually works as well, but it may be less efficient.

3 Granularity of Synchronization

- Often we have a choice how *detailed* we synchronize.

P(S);	P(S);
critical	critical
V(S);	non–critical
non–critical	critical
P(S);	V(S);
critical	
V(S);	

- or

```
while(...) {              P(S);
    P(S);                 while(...) {
    critical                critical
    V(S);                   non–critical
    non–critical          }
}                         V(S);
```

Q: what are the advantages/disadvantages?

4 A Design Choice

- We say that the one with smaller critical sections synchronizes with *finer* granularity.
- Finer granularity allows to switch more often between threads.
- The coarser solution spends less time on synchronization.
- The coarser solution might deadlock, when the left one does not.
- Often we have the design choice between
  - Allowing more things to happen concurrently.
  - Spending less time on synchronization.

5 Example: Array, synchronized as a whole

```
class synchArray {
	privat int[ ] a;
	synchArray(int i) {
		a = new int[i];
	}

	synchronized void set(int[ ] b) {
		System.arrayCopy(a, 0, b, 0, a.length);
	}

	synchronized int[ ] get() {
		int[ ] b = new int[a.length];
		System.arrayCopy(b, 0, a, 0, a.length);
		return b;
	}
}
```

6 Array, synchronized as a whole (2)

```
class synchArray {
    privat int[ ] a;
    privat Database db;
    synchArray(int i) {
        a = new int[i];
    }

    void set(int[ ] b) {
        db.startWrite();
        System.arrayCopy(a, 0, b, 0, a.length);
        db.endWrite();
    }
```

```
    int[ ] get() {
        db.startRead();
        int[ ] b = new int[a.length];
        System.arrayCopy(b, 0, a, 0, a.length);
        db.endRead();
        return b;
    }
}
```

- The second solution does a finer synchronization.
- There is more synchronization work to do.
  - We call more synchronized methods.
  - Calling a synchronized method is expensive.
- More can go on concurrently.
  - On a multiprocessor system, array-readers may work simultaneously.

7 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 fine-grained synchronization easier.

8 Block synchronization (2)

- As opposed to monitors, we do not have condition variables in Java.
- wait and notify work on object not on variables.
- So we can use an Object as a condition variable:

```
Object o = new Object();
...
synchronized (o) {
    o.wait();
}
...
synchronized (o) {
    o.notify();
}
```

- Often we can use Objects that we have anyway.

9 Other things to be aware of

- Each object has a lock (a door with a key).
- We say a thread *owns* the lock if it has the key.
- A "door" is always associated with an object.
- A "key" can be either available or owned by a thread.
- A thread that owns the lock (has the key) 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.

10 Stopping Threads

- When we write code for a thread we extend `Thread` and override its run method.
- We want to write a PrinterThread, that removes integers from a bounded buffer and prints them.
- We want to be able to stop that thread.
- We introduce a variable `stop`.
- When we want to stop the thread we set a variable stop to true.
- The thread looks into this variable and stops if it is true.

```
class PrinterThread extends Thread {
        BoundedBuffer b;
        boolean stop;

        public PrinterThread(BoundedBuffer b) {
                this.b = b;
                this.stop = false;
        }
        public void run() {
                System.out.println("start printer");
                while (!stop) {
                        int i = b.remove();
                        if (!stop)
                                System.out.println(i);
                }
                System.out.println("end printer");
        }
}
```

11 The Main Thread

- In the main thread we create a bounded buffer and a printer thread.
- We start the printer thread with t.start().
- Then we stop it with t.stop = true.

```
public static void main(String[ ] args)
        throws IOException, InterruptedException {
    System.out.println("start main");
    BoundedBuffer b = new BoundedBuffer();
    PrinterThread t = new PrinterThread(b);
    t.start();
    b.add(3);
    System.in.read();
    t.stop = true;
    System.in.read();
    System.out.println("end main");
}
```

12 Second Attempt

- The thread will not stop.
- It is waiting in the wait of the remove().

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

```
synchronized int remove() throws InterruptedException {
    while (count == 0) {
            wait();
    }
    count = count - 1;
    int o = buffer[out];
    out = (out + 1) % BUFFER_SIZE;
    notify();
    return o;
}
```

Now we have to check for the expression in the PrinterThread.

```
public void run() {
    System.out.println("start printer");
    while (!stop) {
        int i = 0;
        try {
            i = b.remove();
        } catch (InterruptedException e) { }
        if (!stop)
            System.out.println(i);
    }
    System.out.println("end printer");
}
```

And we have to call t.interrupt in the MainThread.

```
class MainThread {
    public static void main(String[ ] args)
            throws IOException, InterruptedException {
        System.out.println("start main");
        BoundedBuffer b = new BoundedBuffer();
        PrinterThread t = new PrinterThread(b);
        t.start();
        b.add(3);
        System.in.read();
        t.stop = true;
        t.interrupt();
        System.in.read();
        System.out.println("end main");
    }
```

13 Deprecated Java Features

- stop() to stop a Java thread immediately.
- suspend() and resume(), to stop a Java thread temporarily and restart it later.

```
public static void main(String[ ] args) throws IOException {
    System.out.println("start main");
    BoundedBuffer b = new BoundedBuffer();
    PrinterThread t = new PrinterThread(b);
    t.start();
    b.add(3);
    System.in.read();
    t.stop();
    System.in.read();
    System.out.println("end main");
}
```

DO NOT USE THESE FEATURES!!

Q: Why not?

14 Summary Concurrency

Two ways to implement concurrency

- Processes
    - Do not share code and other resources.
    - Operating systems.
    - Distribution.
- Threads
    - Share code and other resources.
    - Single Application, that needs concurrency (Browser)
    - Server.

15 Problems

- First Problem: Race Condition.
  - Shared variables, that are accessed concurrently.
- Critical sections and mutual exclusion.
  - Critical section are parts of the code, that need to run exclusively in time.
  - These are typically sections, where we address shared variables.
- Second problem: Deadlocks, Starvation.
  - Threads have to wait for each other.
  - If two or more threads are waiting for each other we have a deadlock.
  - If we do not have a deadlock, but one thread does not get to run anymore, we call it starvation.
- The goal is to write concurrent programs, that avoid race conditions, deadlocks and starvation.

16 Example Problems

- bounded buffer with producer and consumers
- readers and writers
- dining philosophers

17 Solution: Semaphore

- Binary semaphores for critical sections

```
P(S)
        critical section
V(S)
```

- Counting semaphores: We have n resources, so we allow up to n threads to be in certain code sections.

18 Solution: Java-Synchronization/Monitors

- We have synchronized methods
- No two threads can execute synchronized methods of one object at the same time.
- We can call a method wait, if we have to wait for a certain condition to come true.
- We can call notify and notifyAll, to tell other threads that a condition has come true/might have come true.