In this lecture, we covered some of the (unwanted) behavior that synchronization constructs can exhibit; namely, race conditions and deadlock.

1 Race conditions

Consider the following setup. There is a group of graduate students (GS) and a group of undergraduates (UG). There is a room in which a study will be conducted. One GS and one UG go into the room at a time; the GS observes the UG performing a task for some period of time; the UG leaves; the GS leaves; finally, the GS writes a report about what he or she observed.

This setup has some similarities with multiple threads running in an operating system:

- if only a GS shows up to the room, he or she must wait until a UG arrives (and vice-versa)
- the room is an exclusive resource, as only two people can use it at a time
- the behavior of the two groups must be synchronized

The room represents the monitor synchronization primitive.

The following picture illustrates the setup.
We can implement this system in pseudocode:

```plaintext
int g, u; // count of GSs and UGs
    // at least one must be 0
bool g_in_rm, u_in_rm;

grad:
    if (u > 0)
        signal C_u;
    else {
        g++;
        wait C_g;
        g--;
    }

if (g_in_rm)
    wait C_rm;
    g_in_rm = T;

wait U_done;
<perform task>
signal U_done;

g_in_rm = F;
signal C_rm;
<write report>
return;

ugrad:
    if (g > 0)
        signal C_g;
    else {
        u++;
        wait C_u;
        u--;
    }

if (u_in_rm)
    wait C_rm;
    u_in_rm = T;

return;
```

In fact, a race condition exists in this example, as shown by the following sequence of events:

- G1 and U1 are already in the room; G2 and U2 are waiting to enter
- U1 leaves, signals C_{rm}
- G2 enters, sets g_{in_rm} to True
- G1 leaves, sets g_{in_rm} to False
- U2 enters

After this sequence, the room has one GS and one UG, but the flags indicate that it only has a UG. This race condition has a simple fix: make two conditions C_{g_{rm}} and C_{u_{rm}} to control entrance to the room, one for graduate students and one for undergraduates.
Another type of synchronization primitive is the **rendezvous**; it is used by the Ada programming language, and illustrated in the following picture.

When the first process calls `r.meet()`, it waits until the second does also, and then they exchange values. A rendezvous could be implemented as follows:

```c
condition C;
int val1, val2;
bool second;

meet(x):
    1   if (not(second)) {
    2       second = T;
    3       val1 = x;
    4       wait C;
    5       return val2;
    }
    else {
    6       val2 = x;
    7       second = F;
    8       signal C;
    9       return val1;
    }
```

A race condition could also occur with this primitive. Assume that A and B are the first two threads to run (in that order), and that we have two additional threads called C and D. If C and D are both waiting when B finishes line 9, and they both run before A is allowed to execute line 5, then A will return the `val2` set by D rather than the one set by B.

This issue can be fixed with a counting semaphore, used to ensure that only two threads are running at any given time.
2 Deadlock

Deadlock occurs when two (or more) processes are waiting for resources, and neither can proceed until the other releases its resources. Here are the two most basic threads with which this could happen (m1 and m2 are mutexes).

```
1 ------ lock(m1); ------ lock(m2); ------
2 ------ lock(m2); ------ lock(m1); ------
3 ------ unlock(m1,m2); ------ unlock(m2,m1); ------
```

If B’s first line runs between A’s first and second lines, both processes will wait forever at their second line. Using the numbered regions in the code above, the following diagram illustrates the possible transitions. As you can see, there is no way out of the (2,2) state.

There are four conditions that are necessary for deadlock to occur:

1. **mutual exclusion** (there are a limited number of resources, and they can’t be shared)
2. **hold and wait** (threads don’t release their resources when they have to wait for more)
3. **no preemption** (a thread can’t be forced to release its resources)
4. a **circular chain** in the graph of resource dependencies exists at some time during execution
A classic example of deadlock in an operating system is illustrated by the following pseudocode.

```
getbuffer:  spinlock(ℓ)  syscall:  ::
<remove>
spinunlock(ℓ)

interrupt_handler:  getbuffer()
```

Here, deadlock would occur if an interrupt occurred in the `<remove>` section of `getbuffer`.

Another less obvious example could occur when a system is low on memory. In this case, what should happen is that some dirty pages from memory be written off somewhere else so they can be freed. However, it could be possible that the very act of writing the dirty pages requires more memory than is available, causing the system to freeze up.

Two common ways of preventing deadlock are to (a) design your threads so that they will not be competing for the same resources, or (b) impose an ordering on resources, and require that threads only lock resources in increasing order, and only unlock them in decreasing order.