Concurrent and Distributed Systems

Talk Three

Synchronizing Processes

Maxim Mozgovoy

The Problem of Coordination

Different processes have to be synchronized in order to do something useful (in the same way as the builders coordinate their actions when they build a house).

Most concurrent programs do some synchronization. Without synchronization, a program may suffer from race condition.

Race condition is a situation when the output of the system depends on uncontrollable timing of events (i.e., luck).

One of the most basic coordinating/synchronizing tools in Java is semaphore (invented by E. Dijkstra in 1965).

The Semaphores

Semaphore is an integer-valued object

with the atomic methods acquire() and release():

```
int value = …; // initial value set by the user

acquire()
{
    while(value < 1)
        do_nothing();
    value--;
}

release()
{
    value++;
}
```

How To Do Nothing

How can do_nothing() be implemented?

Version 1: Run an empty loop (spinlock semaphore):

```
while (value < 1)
    ;
```

Version 2: Ask the system scheduler to suspend the process.
Suspended processes do not consume CPU resources.

Java semaphores use Version 2.
Another common function that suspends a process is sleep().

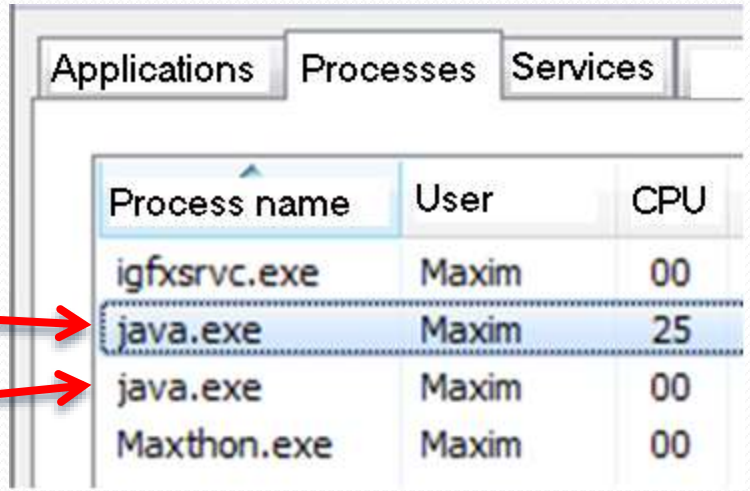
Sleep vs. Loop

```
class Sleep {

static public
void main(String args[])
{
   try {
      for(;;)
         Thread.sleep(100);
   }
   catch(Exception e) {}
}
}
```

```
class Loop {

static public
void main(String args[])
{
   for(;;)
      ;
}
}
```

java Loop

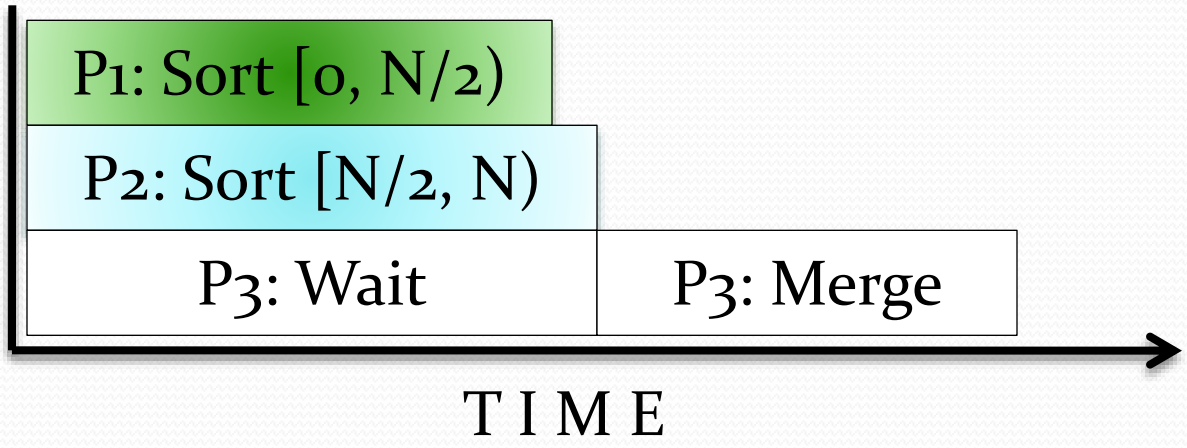
java Sleep



Coordination in Array Sorting

Example: Semaphore-supported coordination can be used in parallel array sorting according to the following idea:

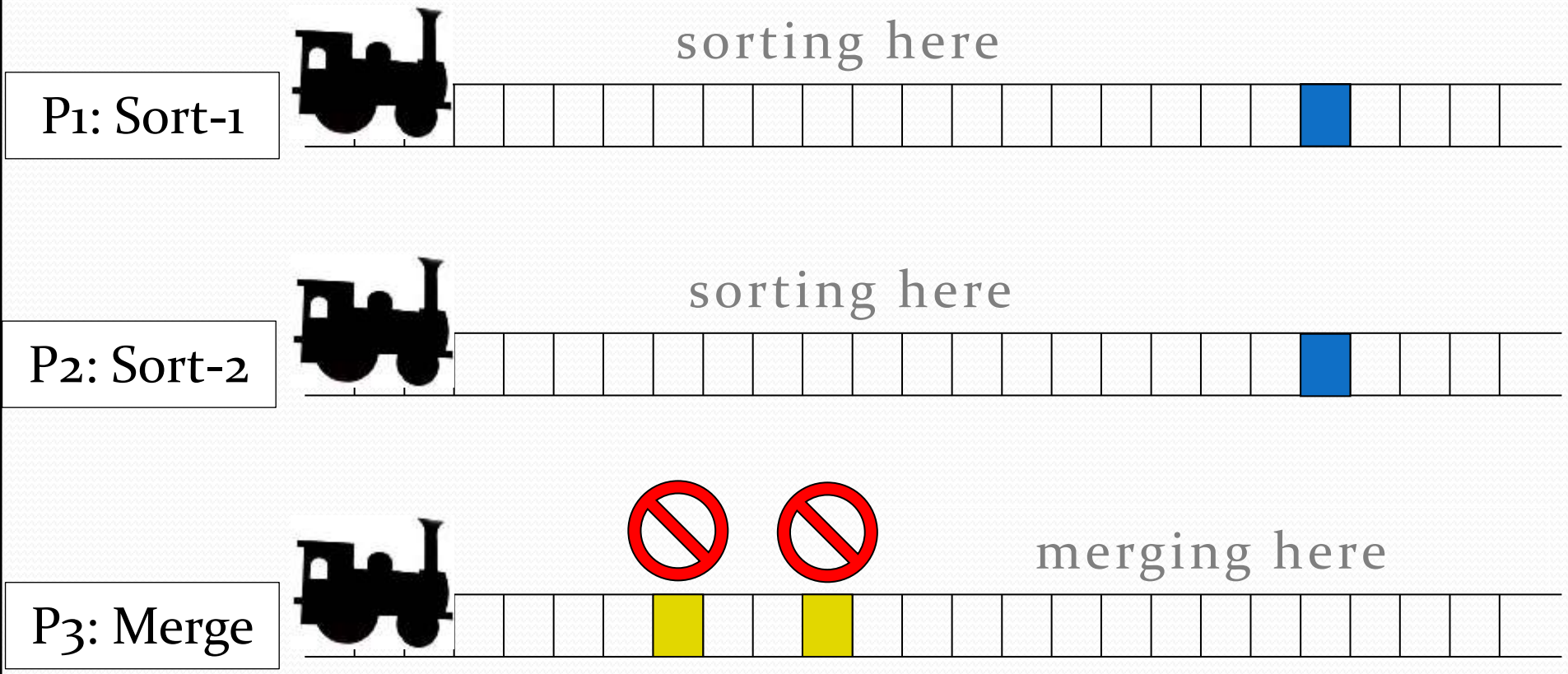
- Process 1: sort first half of the array.
- Process 2: sort second half of the array.
- Process 3: wait until both halves are sorted.
  Then merge them into the resulting array.



Parallel Array Sorting (Pseudocode)

```
int A[N];
Semaphore S;
S.value = 0;    // any call to acquire()
                // will suspend the process
// process P1
sort A[0...N/2)
release(S);     // S.value = S.value + 1

// process P2
sort A[N/2...N)
release(S);     // S.value = S.value + 1

// process P3
acquire(S);     // waits for release() call
acquire(S);     // waits for release() call
merge A[0...N/2) with A[N/2...N)
```

Parallel Array Sorting (Demo)



Parallel Array Sorting in Java

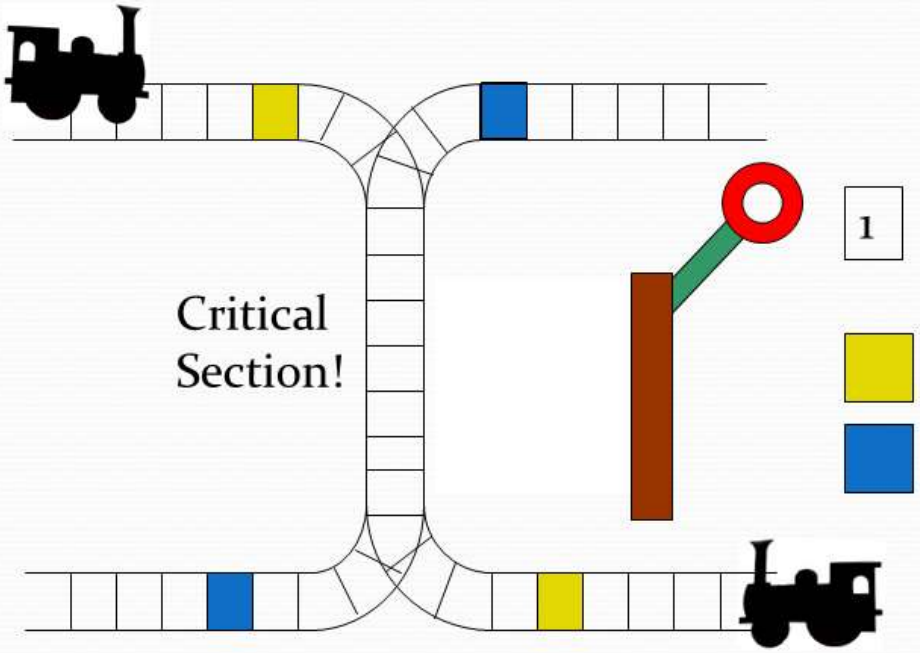
```
// some useful fragments:
// semaphore package:
import java.util.concurrent.Semaphore;

Semaphore S = new Semaphore(0); // make a semaphore
                                // set its value

// acquire: (can raise an exception, must catch it)
try { S.acquire(); } catch(Exception e){}

// release:
S.release();
```

See the complete listing in ex03_sort2Proc.java

(Alternative to semaphore in this case: **CountDownLatch** class)

Critical Sections

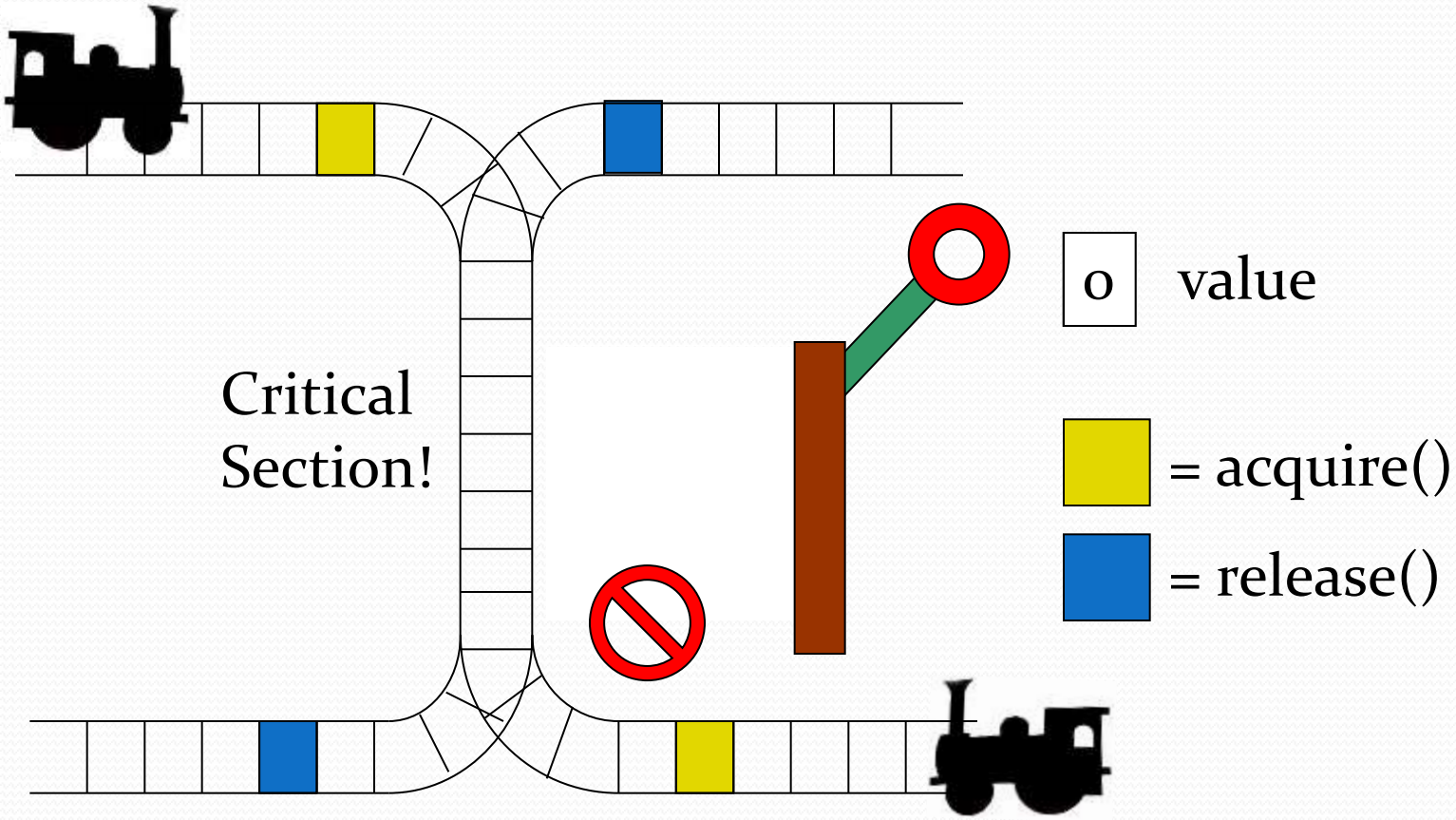
Critical section: a code fragment that should not be executed simultaneously by two or more processes.



To organize a critical section, you need a binary semaphore (with value = 0, 1).

(Every time the section allows at most one process to work).

Critical Sections (Demo)



Critical Sections: Money Example

A husband and a wife have one shared bank account. Each spouse wants to put there 500 000 money units.

Money added unit by unit (one unit at a time).

Task: We need to model this situation.
Let's do it as follows:

- Husband and wife are two processes.
- The code in main() runs husband and wife,
  waits until they finish their work, and prints results.

Money Example: Attempt 1

```
int Account = 0;                          // global variable
Semaphore S = new Semaphore(0);           // S: to wait
                                          // before print
class Spouse implements Runnable {
    private int Sum;
    public Spouse(int sum) { Sum = sum; }

    public void run() {
        for (int i = 0; i < Sum; i++)
            Account++;

        S.release();
    }
}
```

Money Example: Attempt 1 (Cont.)

```
static public void main(String args[]) {
    Spouse husband = new Spouse(500000);
    Spouse wife = new Spouse(500000);

    new Thread(husband).start();
    new Thread(wife).start();

    // wait until both are finished
    try {
        S.acquire();
        S.acquire();
    }
    catch(Exception e){}

    System.out.println(Account);
  }
}
```

Printout:
536031, 627985, 520398, …

(**Expected: 1000000**)

Why Does It Fail?

Example scenario:

```
// Account = 5;
```

```
Account++;
```

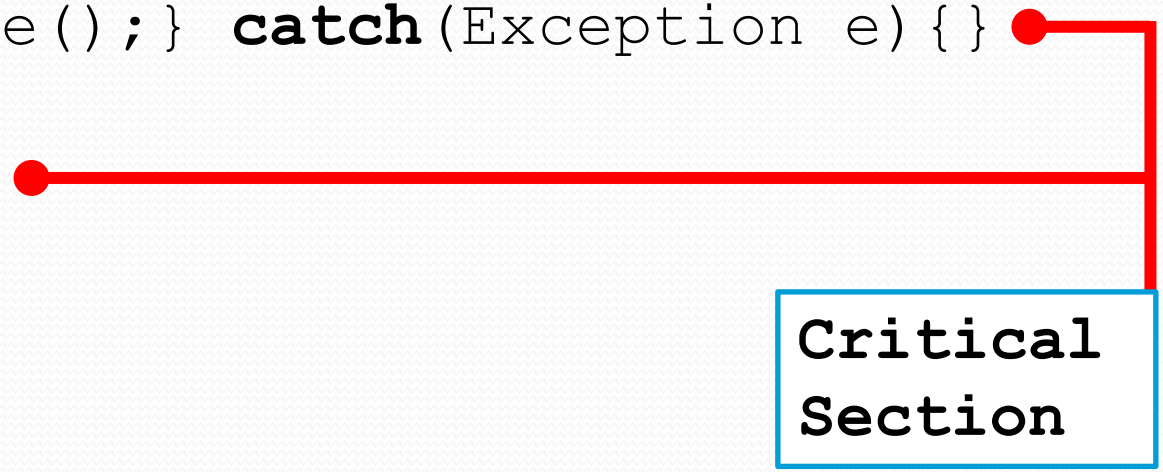
```
Husband:   read register X, Account   // 5
Wife:      read register X, Account   // 5
Husband:   increase X                 // 6
Wife:      increase X                 // 6
Husband:   write Account, register X  // 6
Wife:      write Account, register X  // 6
```

So, husband adds one unit, wife adds one unit (2 units in total), but the account is increased by one (5 → 6)!

The processes should not access the account at the same time!

Money Example: Using The Critical Section

```
Semaphore M = new Semaphore(1);

class Spouse implements Runnable {
  public void run() {
    for (int i = 0; i < Sum; i++)
    {
      try {M.acquire();} catch(Exception e){}
      Account++;
      M.release();
    }

    S.release();
  }
}
```



(See ex03_money.java). Prints: 1000000, 1000000…

Mutex: a Binary Semaphore

Mutex (**Mut**ual **Ex**clusion) is a version of a binary semaphore for critical sections. In Java, any object can serve as a mutex:

```
// Binary Semaphore
Semaphore M =
     new Semaphore(1);

void f()
{
     M.acquire();

     … // critical sec

     M.release();
}
```

```
// Mutex
Object M =
     new Object();

void f()
{
     synchronized(M)
     {
          … // critical sec
     }
}
```

Monitors (1)

Another useful synchronization idiom is monitor.

A monitor is an object, having mutually exclusive methods.

You can simulate a monitor in Java with a mutex object:

```
class Monitor {
    private Object M = new Object();

    public void f() { synchronized(M) { … } }
    public void g() { synchronized(M) { … } }
    public void h() { synchronized(M) { … } }
}
```

Monitors (2)

Java also has a special syntax for monitors:
just declare class methods as **synchronized**,
and they will be mutually exclusive
("**this**" object will be implicitly used as a mutex)

```
class Monitor {
    synchronized public void f() { … }
    synchronized public void g() { … }
    synchronized public void h() { … }
}
```

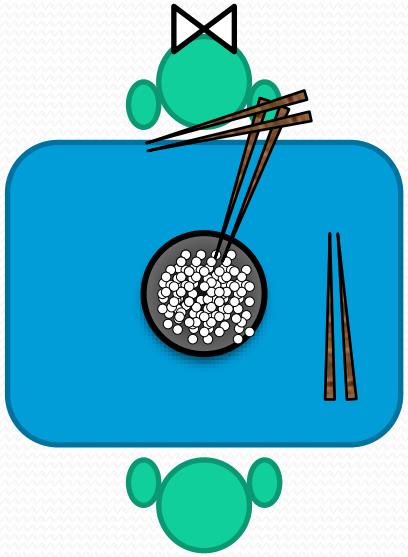
Troubles with Synchronization

Synchronization must be done with care.

The most common pitfalls are deadlock and livelock.

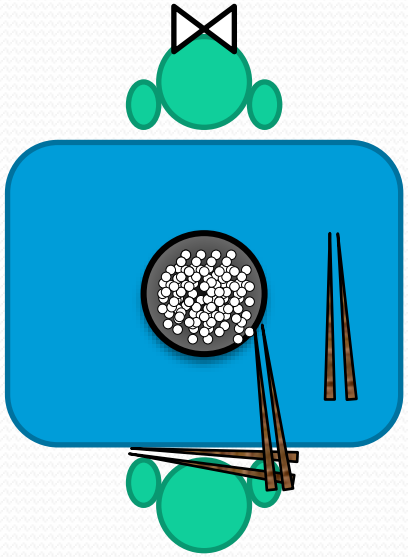
Deadlock (1)

(Almost) a real-life example: eating rice together!



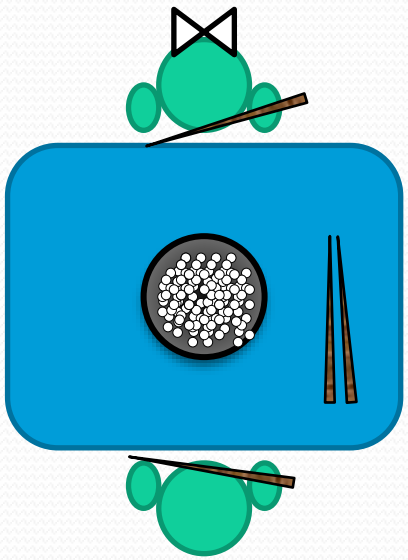
Deadlock (1)

(Almost) a real-life example: eating rice together!



Deadlock (1)

(Almost) a real-life example: eating rice together!



Waits for the chopstick

Waits for the chopstick

Deadlock (2)

A deadlock occurs when two processes (eaters) want to allocate the same resource (chopsticks), but block each other.

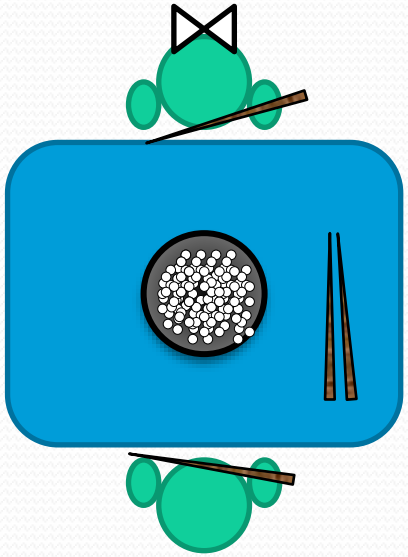
A deadlock can be often prevented with an improved resource allocation scheme. E.g., let's assign each stick an ID (1 and 2). Then each eater should first try to get stick 1, and only then stick 2. This will guarantee that nobody will try to get stick 2 if stick 1 is already taken.

There are situations when preventing deadlocks is not possible (e.g., DBMS with many users who can lock tables).

In these cases, we can detect the deadlock and shutdown one of the competing processes.

Livelock

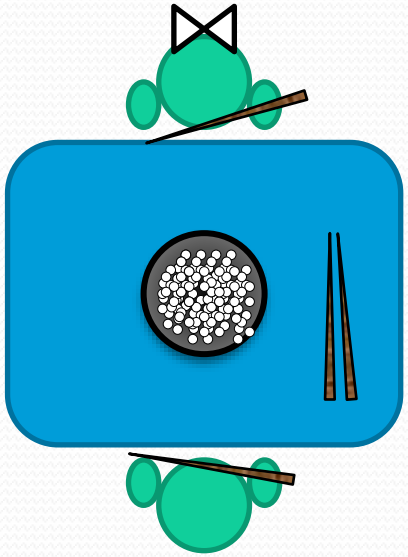
Livelocks are less common:



Oops!

Livelock

Livelocks are less common:



Oops!

Here each process tries to "help" others by freeing its resources, but as a result nobody does anything useful.

A simple solution: randomize the strategy. Each process can randomly decide whether to keep or to release its chopstick.

Conclusions

- Individual processes of a concurrent program should work together in a coordinated way (like house builders).
- The simplest coordination tools are semaphore and mutex. You can use them to make some processes wait while other processes work.
- A typical use case is a critical section problem: to make some part of the program executable by only one process at time.
- Normally, "sleeping" processes do not consume resources – it is operating system's job to ensure that.
- Improper coordination may lead to deadlocks and livelocks.