

Unit 2

Busy Wait Synchronization

- 2.1 Introduction
- 2.2 Condition Synchronization
- 2.3 Mutual Exclusion
 - 2.3.1 First Attempt
 - 2.3.2 Second Attempt
 - 2.3.3 Third Attempt
 - 2.3.4 Fourth Attempt
 - 2.3.5 Dekker's Algorithm
 - 2.3.6 Several Critical Sections
- 2.4 Busy wait VS. passive wait
- 2.5 Conclusions

2.1 Introduction

2.2 Condition Synchronization

2.3 Mutual Exclusion

2.3.1 First Attempt

2.3.2 Second Attempt

2.3.3 Third Attempt

2.3.4 Fourth Attempt

2.3.5 Dekker's Algorithm

2.3.6 Several Critical Sections

2.4 Busy wait VS. passive wait

2.5 Conclusions

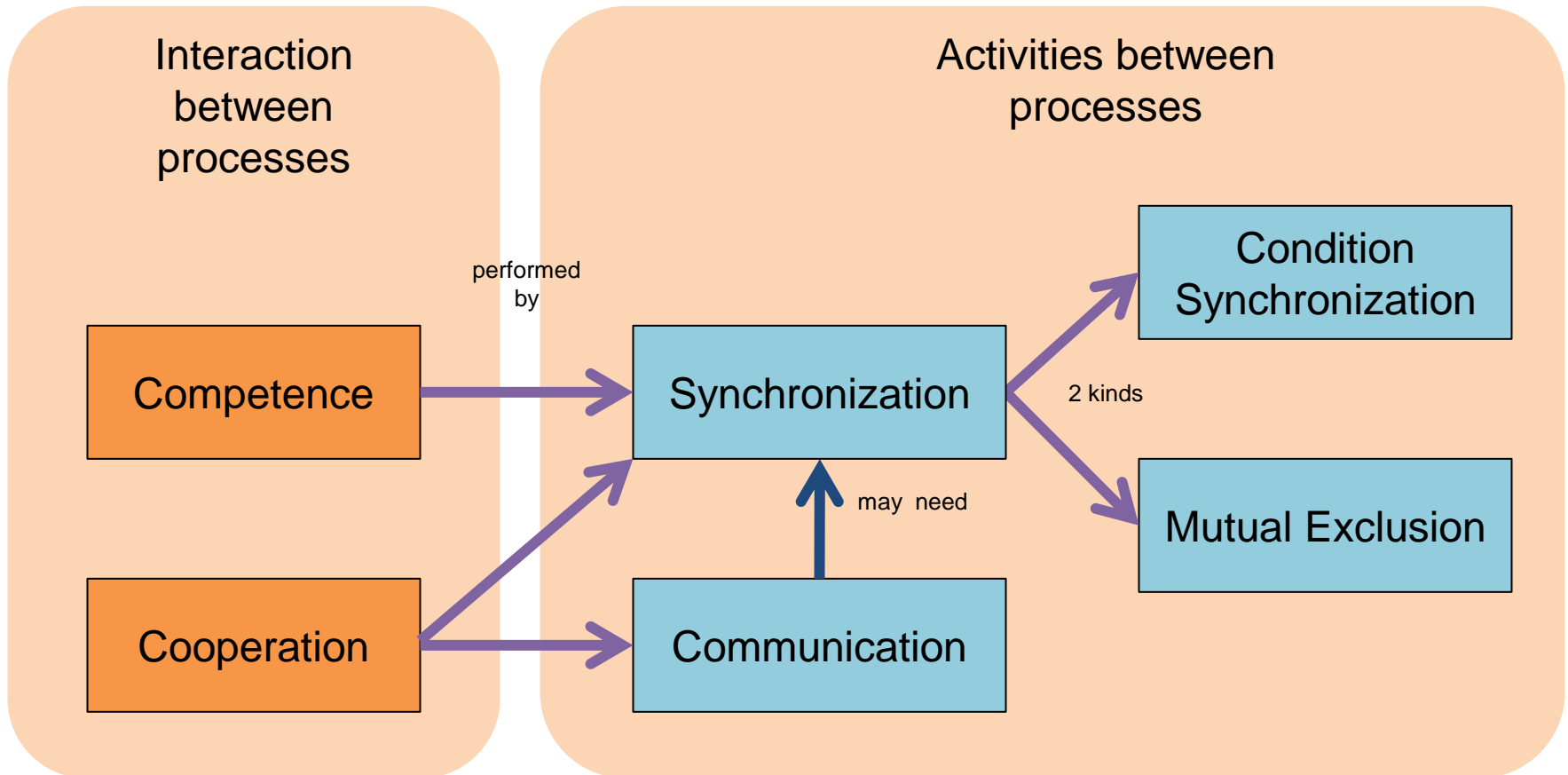
2.1 Introduction

- Uniprocessors and multiprocessor/multicore use shared memory:
 - The compiler and OS avoid the use of same memory addresses
 - But, shared variables can be written and read by several processes.
 - What happens if 2 processes read and/or write the same variable 'at the same time'?
- By the abstraction of concurrent programming, we must think that any interleaving is possible for all non-atomic statements.
- Java and Pascal-FC provide atomic read/write operations in primitive variables (except double and long).

2.1 Introduction

- If 2 processes read a variable at the same time...
→ both process read same value
- If 2 processes write at the same time (any after the other):
→ the last written value remains, but we cannot predict it.
- If one process reads and the other writes at the same time (any after the other):
→ the read value could be the former or the latter, but we cannot predict it.

2.1 Introduction



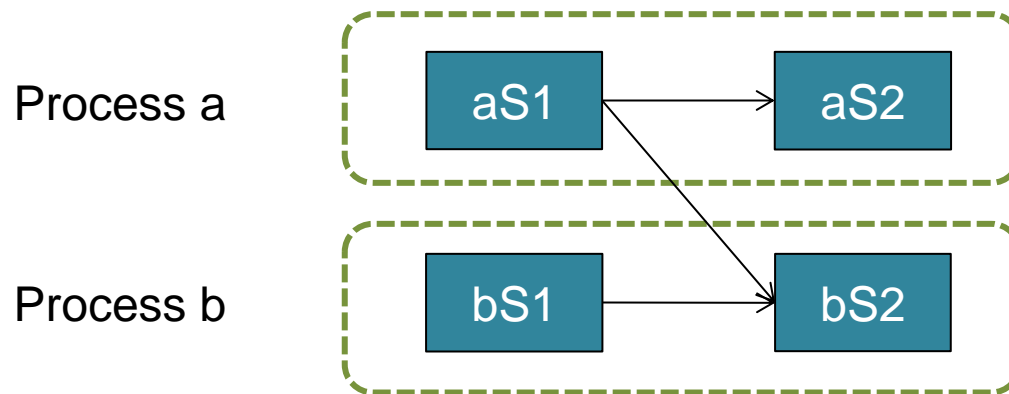
2.1 Introduction

- Shared-memory **communication**:
 - Shared variables let processes share information by reading and writing on them
- Shared-memory **synchronization**:
 - **Condition Synchronization**: depending on the value of one or more variables
 - **Mutual Exclusion**: use of structures or algorithms which provide exclusive access to critical sections.

- 2.1 Introduction**
- 2.2 Condition Synchronization**
- 2.3 Mutual Exclusion**
 - 2.3.1 First Attempt
 - 2.3.2 Second Attempt
 - 2.3.3 Third Attempt
 - 2.3.4 Fourth Attempt
 - 2.3.5 Dekker's Algorithm
 - 2.3.6 Several Critical Sections
- 2.4 Busy wait VS. passive wait**
- 2.5 Conclusions**

2.2 Condition Synchronization

- One process halts waiting for a condition to be met thanks to the action of another process.
- Let us consider the following precedence diagram of the statements of 2 processes:



2.2 Condition Synchronization

- Assume that the last instruction in each block of statements is to print the id of such block (*write* from Pascal/FC and *System.out.print* in Java provide atomic access to screen).

Some possible outputs:

aS1 aS2 bS1 bS2

aS1 bS1 aS2 bS2

bS1 aS1 aS2 bS2

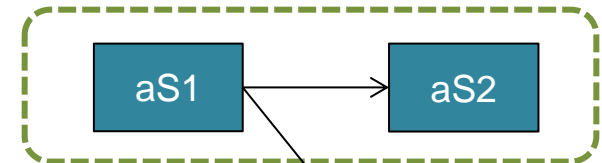
bS1 aS1 aS2 bS2

...think more...

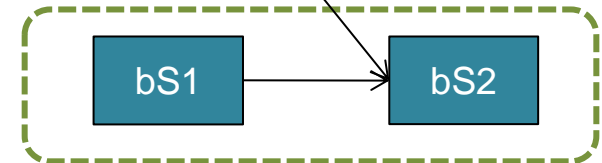
An impossible output:

~~bS1 bS2 aS1 aS2~~

Process a



Process b

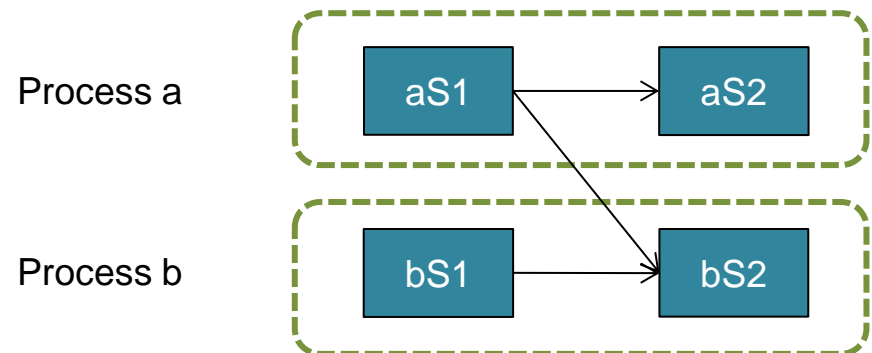


2.2 Condition Synchronization

Pascal-FC code

```
program conditSynch;  
  
var  
  continue: boolean;  
  
process ProcessA;  
begin  
  write('aS1 ');  
  continue := true;  
  write('aS2 ');  
end;  
  
process ProcessB;  
begin  
  write('bS1 ');  
  while not continue do;  
    write('bS2 ');  
  end;  
  
begin  
  continue := false;  
cobegin  
  ProcessA;  
  ProcessB;  
coend  
end.
```

- Shared variable: *continue*
- The work done by *Process b* waiting for the condition to be set, is called **busy wait**. That is, waste processor time doing *nothing* until the other process sets the condition.



2.2 Condition Synchronization

JAVA code

```
class ThreadA extends Thread{

    SharedObject s;

    public ThreadA(SharedObject shared){
        s=shared;
    }

    public void run(){
        System.out.println("aS1");
        s.setGo();
        System.out.println("aS2");
    }
}
```

```
class ThreadB extends Thread{

    SharedObject s;

    public ThreadB(SharedObject shared){
        s=shared;
    }

    public void run(){
        System.out.println("bS1");
        while(!s.getGo());
        System.out.println("bS2");
    }
}
```

```
class SharedObject{

    boolean go=false;

    public boolean getGo(){
        return go;
    }

    public void setGo(){
        go=true;
    }
}
```

Why isn't it necessary to add the volatile modifier to boolean go?

```
public class CondSynch {

    public static void main(String args[]){
        SharedObject s=new SharedObject();
        (new ThreadA(s)).start();
        (new ThreadB(s)).start();
    }
}
```

2.2 Condition Synchronization

- Interleaving of instructions to get the output:

aS1 bS1 aS2 bS2

	Process A	Process B	continue
1	write('aS1 ');		false
2		write('bS1 ');	false
3	continue := true;		true
4	write('aS2 ');		true
5		while not continue	true
6		write('bS2 ');	true

2.2 Condition Synchronization

- Interleaving of instructions to get the output:

<code>bS1 aS1 aS2 bS2</code>

	Process A	Process B	continue
1		<code>write('bS1 ');</code>	false
2		<code>while not continue</code>	false
3		<code>while not continue</code>	false
4		<code>while not continue</code>	false
5	<code>write('aS1 ');</code>		false
6	<code>continue := true;</code>		true
7	<code>write('aS2 ');</code>		true
8		<code>while not continue</code>	true
9		<code>write('bS2 ');</code>	true

2.2 Condition Synchronization

- Interleaving of instructions to get the output: *complete the table*

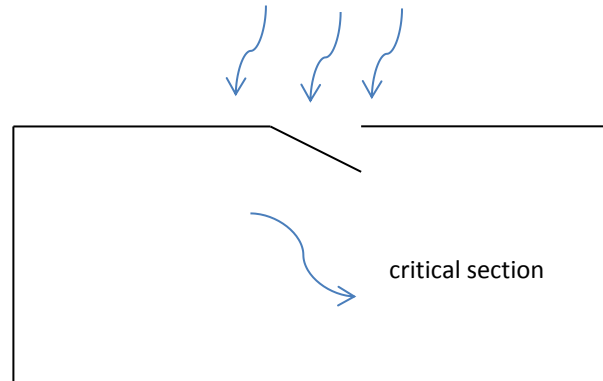
aS1	aS2	bS1	bS2
-----	-----	-----	-----

	Process A	Process B	continue
1	...		false
2	...		true
3	...		true
4		...	true
5		...	true
6		...	true

- 2.1 Introduction**
- 2.2 Condition Synchronization**
- 2.3 Mutual Exclusion**
 - 2.3.1 First Attempt
 - 2.3.2 Second Attempt
 - 2.3.3 Third Attempt
 - 2.3.4 Fourth Attempt
 - 2.3.5 Dekker's Algorithm
 - 2.3.6 Several Critical Sections
- 2.4 Busy wait VS. passive wait
- 2.5 Conclusions

2.3 Mutual Exclusion

- When two or more processes need to access a shared variable, object or set of statements which require exclusive access, **processes need to be synchronized** in order to guarantee that only 1 of them gains access (enters the critical section).



Ben-Ari et al. Chapter 3. 2006.

- The **synchronization by mutual exclusion** is the execution of a set of instructions before entering the critical section (**pre-protocol**) and another set of instructions immediately after leaving the critical section (**post-protocol**).
- The **pre-protocol** guarantees mutual exclusion in the access.
- The **post-protocol** communicates to the other processes it is not waiting to enter in the critical section anymore.

2.3 Mutual Exclusion

- A **correct solution** of mutual exclusion fulfills:
 - Mutual exclusion is granted.
 - Avoids livelock and starvation of processes trying to enter the critical section.
- And, it would be good that:
 - No variables used in the critical and non-critical sections are used in the protocols. That is, variables used in protocols are created for their **exclusive use** in protocols.
 - Pre and post-protocols should use **little** memory and CPU clock-time.

2.3 Mutual Exclusion

- Solutions using protocols assume that the only atomic instructions available are read and write on primitive variables.
- In 1965, Dijkstra published a solution for mutual exclusion in the case of 2 processes. In order to explain it, he first presents **4** wrong approaches or **attempts** in which the most common errors of concurrent programming appear.
- The correct solution is based on a mathematician called Dekker, so Dijkstra called it **Dekker's algorithm**.
- Dijkstra improved Dekker's Algorithm for $n > 1$ processes: *Dijkstra's algorithm*.
- The Eisenberg-Mcguire's algorithm is an optimization of Dijkstra's algorithm.

- 2.1 Introduction**
- 2.2 Condition Synchronization**
- 2.3 Mutual Exclusion**
 - 2.3.1 First Attempt**
 - 2.3.2 Second Attempt
 - 2.3.3 Third Attempt
 - 2.3.4 Fourth Attempt
 - 2.3.5 Dekker's Algorithm
 - 2.3.6 Several Critical Sections
- 2.4 Busy wait VS. passive wait
- 2.5 Conclusions

2.3 Mutual Exclusion

1st attempt (Pascal-FC)

- Based on busy-wait: processes share 1 variable to tell which process may enter in CS

```
program FirstAttempt;
var turn:Integer;

process P1;
begin
    repeat
        while turn <> 1 do;
            writeln('P1 is in CS');
            writeln('P1 is leaving CS');
            turn:=2;
            writeln('P1 is in non-CS');
        forever
    end;

process P2;
begin
    repeat
        while turn <> 2 do;
            writeln('P2 is in CS');
            writeln('P2 is leaving CS');
            turn:=1;
            writeln('P2 is in non-CS');
        forever
    end;
```

```
begin
    turn:=1;
cobegin
    P1;
    P2;
coend
end.
```

Identify the pre-protocol, the Critical Section and the post-protocol in both processes

2.3 Mutual Exclusion

1st attempt

- Commonly, this approach would work but...
- The **1st attempt is not correct because:**
 - it is not free of **starvation** if one process fails.

	P1	P2	turn
1	while turn <> 1 do ;		1
2	writeln('P1 is in CS');		1
3		while turn <> 2 do ;	1
4	<i>process 1 crashes!</i>		1
		<i>Remains forever in busy-wait</i>	1

- **Alternation is mandatory:** access to CS is granted in turns, so if a process is very slow (long non-CS) the other cannot enter the CS until the other changes the value of *turn*.

2.3 Mutual Exclusion

1st attempt (Java)

```
class CS1 {
    int turn = 1;

    public void enterCS(int ID) {
        // pre-protocol
        while (turn != ID);
        // CS
        System.out.println("P" + ID + " is in CS");
        System.out.println("P" + ID + " is leaving CS");
        //non-CS
        // post-protocol
        turn = ID-1;

        System.out.println("P" + ID + " is in non-CS");
    }
}
```

```
class MyThread extends Thread{
    CS1 sharedCS;
    int ID;

    MyThread(CS1 cs, int id){
        sharedCS=cs;
        ID=id;
    }

    public void run(){
        while(true){
            sharedCS.enterCS(ID);
        }
    }
}
```

```
public class FirstAttempt {

    public static void main(String args[]){
        CS1 criticalSection=new CS1();

        (new MyThread(criticalSection, 0)).start();
        (new MyThread(criticalSection, 1)).start();
    }
}
```

- 2.1 Introduction**
- 2.2 Condition Synchronization**
- 2.3 Mutual Exclusion**
 - 2.3.1 First Attempt**
 - 2.3.2 Second Attempt**
 - 2.3.3 Third Attempt
 - 2.3.4 Fourth Attempt
 - 2.3.5 Dekker's Algorithm
 - 2.3.6 Several Critical Sections
- 2.4 Busy wait VS. passive wait
- 2.5 Conclusions

2.3 Mutual Exclusion

2nd attempt (Pasca-FC)

- In order to solve the problem of having just one variable which leads to mandatory alternation, each process uses a flag to indicate it is entering in CS, but it will only do that if the other flag is not set.

```
program SecondAttempt;
var flag1, flag2: Boolean;

process P1;
begin
    repeat
        while flag2 do;
            flag1:=true;
            writeln('P1 is in CS');
            writeln('P1 is leaving CS');
            flag1:=false;
            writeln('P1 is in non-CS');
        forever
    end;

process P2;
begin
    repeat
        while flag1 do;
            flag2:=true;
            writeln('P2 is in CS');
            writeln('P2 is leaving CS');
            flag2:=false;
            writeln('P2 is in non-CS');
        forever
    end;
```

But **the 2nd attempt is not correct because**
- **Mutual Exclusion is not guaranteed.**

P2 is in CS

P1 is in CS

-Moreover, **starvation may occur** if one process repeats its loop and sets its flag before the other process leaves its busy-wait.

*Think what interleaving of statements
leads to common access to CS*

2.3 Mutual Exclusion

2nd attempt

Interleaving which leads to common access to CS:

	P1	P2	flag1	flag2
1	<code>while flag2 do</code>		<code>false</code>	<code>false</code>
2		<code>while flag1 do</code>	<code>false</code>	<code>false</code>
3	<code>flag1:=true</code>		<code>true</code>	<code>false</code>
4		<code>flag2:=true;</code>	<code>true</code>	<code>true</code>
5		<code>writeln('P2 is in CS');</code>	<code>true</code>	<code>true</code>
6	<code>writeln('P1 is in CS');</code>		<code>true</code>	<code>true</code>

2.3 Mutual Exclusion

2nd attempt

Starvation does not happen on the fail of one process in its non-CS because its flag would remain false.

	P1	P2	flag1	flag2
1	... flag1:= false ;		false	false
2	<i>P1 halts forever!</i>		false	false
3		while flag1 do	false	false
4		flag2:= true ;	false	true
		writeln('P2 is in CS');	false	true
		writeln('P2 is leaving CS');	false	true
		flag2:= false ;	false	false
		writeln('P2 is in non-CS');	false	false
		while flag1 do	false	false
		flag2:= true ;	false	true
		writeln('P2 is in CS');...	false	true
		<i>Keeps entering in CS forever</i>		

2.3 Mutual Exclusion

2nd attempt (Java)

```
class CS2 {  
  
    boolean[] flag={false,false};  
  
    public void enterCS(int ID) {  
        // pre-protocol  
        while (flag[1-ID]);  
        flag[ID]=true;  
        // CS  
        System.out.println("P" + ID + " is in CS");  
        System.out.println("P" + ID + " is leaving CS");  
        // non-CS  
        // post-protocol  
        flag[ID]=false;  
  
        System.out.println("P" + ID + " is in non-CS");  
    }  
}
```

```
class MyThread2 extends Thread{  
    CS2 sharedCS;  
    int ID;  
  
    MyThread2(CS2 cs, int id){  
        sharedCS=cs;  
        ID=id;  
    }  
  
    public void run(){  
        while(true){  
            sharedCS.enterCS(ID);  
        }  
    }  
}
```

```
public class SecondAttempt{  
  
    public static void main(String args[]){  
        CS2 criticalSection=new CS2();  
  
        (new MyThread2(criticalSection, 0)).start();  
        (new MyThread2(criticalSection, 1)).start();  
  
    }  
}
```

- 2.1 Introduction**
- 2.2 Condition Synchronization**
- 2.3 Mutual Exclusion**
 - 2.3.1 First Attempt**
 - 2.3.2 Second Attempt**
 - 2.3.3 Third Attempt**
 - 2.3.4 Fourth Attempt
 - 2.3.5 Dekker's Algorithm
 - 2.3.6 Several Critical Sections
- 2.4 Busy wait VS. passive wait
- 2.5 Conclusions

2.3 Mutual Exclusion

3rd attempt

- 1st attempt → because of the use of 1 variable, it forces alternation, and one process starves when the other halts.
- 2nd attempt → mutual exclusion is not granted because one process may check the status of the other process before it is updated.
- So now, we still use **2 variables** but they do not indicate the status of being inside the CS or not, but they will to enter **before trying to enter**.

2.3 Mutual Exclusion

3rd attempt (Pascal-FC)

```
program ThirdAttempt;

var wantsCS1,wantsCS2:Boolean;

process P1;
begin
  repeat
    wantsCS1:=true;
    while wantsCS2 do;
      writeln('P1 is in CS');
      writeln('P1 is leaving CS');
      wantsCS1:=false;
      writeln('P1 is in non-CS');
    forever
  end;

process P2;
begin
  repeat
    wantsCS2:=true;
    while wantsCS1 do;
      writeln('P2 is in CS');
      writeln('P2 is leaving CS');
      wantsCS2:=false;
      writeln('P2 is in non-CS');
    forever
  end;
```

```
begin
  wantsCS1:=false;
  wantsCS2:=false;
  cobegin
    P1;
    P2;
  coend
end.
```

I do not enter in CS if the other is willing to enter

*Besides global variables, Pascal-FC allows sharing information by passing **variables per reference** (next slide)*

2.3 Mutual Exclusion

3rd attempt (Pascal-FC)

- Same solution but passing per reference a record which holds the 2 variables

```
program ThirdAttemptPerRef;

type willsRecord = record
    wantsCS1,wantsCS2: boolean;
end;

process P1(var w: willsRecord);
begin
    repeat
        w.wantsCS1:=true;
        while w.wantsCS2 do;
            writeln('P1 is in CS');
            writeln('P1 is leaving CS');
            w.wantsCS1:=false;
            writeln('P1 is in non-CS');
        forever
    end;

process P2(var w: willsRecord);
begin
    repeat
        w.wantsCS2:=true;
        while w.wantsCS1 do;
            writeln('P2 is in CS');
            writeln('P2 is leaving CS');
            w.wantsCS2:=false;
            writeln('P2 is in non-CS');
        forever
    end;
```

```
var wills:willsRecord;
begin
    wills.wantsCS1:=false;
    wills.wantsCS2:=false;
    cobegin
        P1(wills);
        P2(wills);
    coend
end.
```

The 3rd attempt may fall in a **livelock**.
Can you guess the interleaving which leads to that situation?

2.3 Mutual Exclusion

3rd attempt

	P1	P2	wantsCS1	wantsCS2
1	w.wantsCS1:= true ;		true	false
2		w.wantsCS2:= true ;	true	true
3		while w.wantsCS1 do ;	true	true
4	while w.wantsCS2 do ;		true	true
	<i>livelock</i>		true	true

2.3 Mutual Exclusion

3rd attempt (Java)

```
class CS3 {  
  
    boolean[] wantsCS={false,false};  
  
    public void enterCS(int ID) {  
        // pre-protocol  
        wantsCS[ID]=true;  
        while (wantsCS[1-ID]);  
        // CS  
        System.out.println("P" + ID + " is in CS");  
        System.out.println("P" + ID + " is leaving CS");  
        // non-CS  
        // post-protocol  
        wantsCS[ID]=false;  
  
        System.out.println("P" + ID + " is in non-CS");  
    }  
}
```

- 2.1 Introduction**
- 2.2 Condition Synchronization**
- 2.3 Mutual Exclusion**
 - 2.3.1 First Attempt**
 - 2.3.2 Second Attempt**
 - 2.3.3 Third Attempt**
 - 2.3.4 Fourth Attempt**
 - 2.3.5 Dekker's Algorithm
 - 2.3.6 Several Critical Sections
- 2.4 Busy wait VS. passive wait
- 2.5 Conclusions

2.3 Mutual Exclusion

4th attempt

- 1st attempt → because of the use of 1 variable, it forces alternation, and one process starves when the other halts.
- 2nd attempt → mutual exclusion is not granted because one process may check the status of the other process before it is updated.
- 3rd attempt → if both processes want to enter (**contention for access**), none of them renounces its will. So both will wait forever.
- The fourth attempt resolves contentions by making a process renounce, during a short period of time, its will to enter if the other process wants to enter.

2.3 Mutual Exclusion 4th attempt (Pascal-FC)

```
program FourthAttempt;

type willsRecord = record
    wantsCS1,wantsCS2: boolean;
end;

process P1(var w: willsRecord);
begin
    repeat
        w.wantsCS1:=true;
        while w.wantsCS2 do
            begin
                w.wantsCS1:=false;
                (*do anything, e.g. sleep*)
                w.wantsCS1:=true;
            end;
        writeln('P1 is in CS');
        writeln('P1 is leaving CS');
        w.wantsCS1:=false;
        writeln('P1 is in non-CS');
    forever
end;

process P2(var w: willsRecord);
begin
    repeat
        w.wantsCS2:=true;
        while w.wantsCS1 do
            begin
                w.wantsCS2:=false;
                (*do anything, e.g. sleep*)
                w.wantsCS2:=true;
            end;
        writeln('P2 is in CS');
        writeln('P2 is leaving CS');
        w.wantsCS2:=false;
        writeln('P2 is in non-CS');
    forever
end;
```

```
var wills:willsRecord;
begin
    wills.wantsCS1:=false;
    wills.wantsCS2:=false;
    cobegin
        P1(wills);
        P2(wills);
    coend
end.
```

Both processes may give way to each other during a long period. Livelock and starvation will not last forever, because a process will eventually gain access to CS.

This solution is **correct** but **lacks efficiency**.

2.3 Mutual Exclusion

4th attempt (Java)

```
class CS4 {  
  
    boolean[] wantsCS={false,false};  
  
    public void enterCS(int ID) {  
        // pre-protocol  
        wantsCS[ID]=true;  
        while (wantsCS[1-ID]){  
            wantsCS[ID]=false;  
            try {  
                Thread.sleep(100);  
            } catch (InterruptedException e) {  
                e.printStackTrace();  
            }  
            wantsCS[ID]=true;  
        }  
        // CS  
        System.out.println("P" + ID + " is in CS");  
        System.out.println("P" + ID + " is leaving CS");  
        // non-CS  
        // post-protocol  
        wantsCS[ID]=false;  
  
        System.out.println("P" + ID + " is in non-CS");  
    }  
}
```

- 2.1 Introduction**
- 2.2 Condition Synchronization**
- 2.3 Mutual Exclusion**
 - 2.3.1 First Attempt**
 - 2.3.2 Second Attempt**
 - 2.3.3 Third Attempt**
 - 2.3.4 Fourth Attempt**
 - 2.3.5 Dekker's Algorithm**
 - 2.3.6 Several Critical Sections
- 2.4 Busy wait VS. passive wait
- 2.5 Conclusions

2.3 Mutual Exclusion

Dekker's Algorithm

- Dekker decided to join the first, third and fourth attempts:
 - each process has a *flag* to announce its will to enter in CS.
 - when there is a contention, a common variable *turn* decides which must give way to the other.
- So Dekker's solution uses 3 shared variables: two boolean flags (one per process) and an integer for *turn*.
- It fulfills all requirements to be a correct solution using protocols:
 - Mutual exclusion is assured
 - Livelock does not happen
 - Starvation does not happen
- And it is efficient! (except for the use of busy-wait)

2.3 Mutual Exclusion

Dekker's Algorithm (Pascal-FC)

```
program Dekker;

type willsRecord = record
    wantsCS1,wantsCS2: boolean;
    turn:integer;
end;

process P1(var w: willsRecord);
begin
    repeat
        w.wantsCS1:=true;
        while w.wantsCS2 do
            if w.turn = 2 then
                begin
                    w.wantsCS1:=false;
                    while w.turn = 2 do;
                        w.wantsCS1:=true;
                    end;
                end;
            writeln('P1 is in CS');
            writeln('P1 is leaving CS');
            w.turn:=2;
            w.wantsCS1:=false;
            writeln('P1 is in non-CS');
        forever
    end;
```

```
process P2(var w: willsRecord);
begin
    repeat
        w.wantsCS2:=true;
        while w.wantsCS1 do
            if w.turn = 1 then
                begin
                    w.wantsCS2:=false;
                    while w.turn = 1 do;
                        w.wantsCS2:=true;
                    end;
                end;
            writeln('P2 is in CS');
            writeln('P2 is leaving CS');
            w.turn:=1;
            w.wantsCS2:=false;
            writeln('P2 is in non-CS');
        forever
    end;

var wills:willsRecord;
begin
    wills.wantsCS1:=false;
    wills.wantsCS2:=false;
    wills.turn:=1;
    cobegin
        P1(wills);
        P2(wills);
    coend
end.
```

2.3 Mutual Exclusion

Dekker's Algorithm (Pascal-FC)

Rewrite Dekker's algorithm using global variables instead of a record passed by reference.

2.3 Mutual Exclusion

Dekker's Algorithm (Java)

```
class CSdekker {  
  
    boolean[] wantsCS={false,false};  
    volatile int turn=0;  
  
    public void enterCS(int ID) {  
        // pre-protocol  
        wantsCS[ID]=true;  
        //if the other process does not  
        //want to enter, I enter even  
        //if it is not my turn  
        while (wantsCS[1-ID]) {  
            if(turn==1-ID) {  
                wantsCS[ID]=false;  
                while(turn==1-ID);  
                wantsCS[ID]=true;  
            }  
        }  
        // CS  
        System.out.println("P" + ID + " is in CS");  
        System.out.println("P" + ID + " is leaving CS");  
        // non-CS  
        // post-protocol  
        turn=1-ID;  
        wantsCS[ID]=false;  
  
        System.out.println("P" + ID + " is in non-CS");  
    }  
}
```

Do you think the *volatile* modifier
Is necessary in variable *turn*?

2.3 Mutual Exclusion

- Other algorithms :
 - Peterson's (1981) developed an easier pre-protocol to grant exclusive access to CS.
 - Dijkstra's (1965) is an extension of Dekker's algorithm for n processes.
 - Eisenber-McGuire's (1972) improved the efficiency of Dijkstra's algorithm.
 - Lamport's algorithm, also known as the Bakery algorithm (1974), was developed for n processes running in distributed systems, where there is only read-access for shared variables which belong to other processes
- Hardware solutions: there exist processors which provide special atomic instructions, which grant mutual exclusion avoiding the use of protocols.

Increment & Decrement instructions (also Fetch-and-Add)

```
INC(int x) { int v = x; x = x + 1; return v }
```

Machine instruction from processor IA32

- You can choose any of these algorithms or hardware solution as Task 1 (see proposed tasks in Unit Zero). Use *[Palma et al. Chapter 3.]* as reference.

- 2.1 Introduction**
- 2.2 Condition Synchronization**
- 2.3 Mutual Exclusion**
 - 2.3.1 First Attempt**
 - 2.3.2 Second Attempt**
 - 2.3.3 Third Attempt**
 - 2.3.4 Fourth Attempt**
 - 2.3.5 Dekker's Algorithm**
 - 2.3.6 Several Critical Sections**
- 2.4 Busy wait VS. passive wait
- 2.5 Conclusions

2.3 Mutual Exclusion

Several CSs

- There exist two kinds of atomic execution:
 - **Fine-grained**
 - Provided by the programming language to the developer.
 - They are compiled to atomic machine instructions executed by the processor.
 - **Coarse-grained**
 - Set of instructions executed without interleaving of other processes.
 - There exist programming (protocols, semaphores,...) and hardware solutions which provide tools to make a set of sentences be executed in an atomic manner.
- Given these definition, we can say that the **Critical Section** instructions together are a **coarse-grained instruction** because two processes cannot interleave critical section statements (they can mix one CS statements with non-CS from other process).

2.3 Mutual Exclusion

Several CSs

- The work done inside a critical section usually performs changes in shared variables. If this change is done only in one piece of code, then the program only has 1 CS.
- But the same **shared variable** may be accessed/changed **in several situations** inside the same program:
 - Each piece of code which makes access counts as 1 CS.
 - **Flags and variables used in protocols are not replicated, they are used in access to all CSs.**
- E.g.: increments and decrements of the same variable

2.3 Mutual Exclusion Several CSs

- Variable x is used in 2 critical sections. Protocols may be any which is correct, e.g. Dekker's algorithm.

```
program incdec;

process type inc(var x:integer);
begin
    (*preprotocol(x)*)
    x:=x+1;
    (*postprotocol(x)*)
end;

process type dec(var x:integer);
begin
    (*preprotocol(x)*)
    x:=x-1;
    (*postprotocol(x)*)
end;
```

```
var
    x:integer;
    pInc:inc; pDec:dec;
begin
    x:=0;
    cobegin
        pInc(x);
        pDec(x);
    coend;
    writeln(x)
end.
```

- 2.1 Introduction**
- 2.2 Condition Synchronization**
- 2.3 Mutual Exclusion**
 - 2.3.1 First Attempt**
 - 2.3.2 Second Attempt**
 - 2.3.3 Third Attempt**
 - 2.3.4 Fourth Attempt**
 - 2.3.5 Dekker's Algorithm**
 - 2.3.6 Several Critical Sections**
- 2.4 Busy wait VS. passive wait**
- 2.5 Conclusions**

2.4 Busy wait VS. passive wait

- As we saw, busy-wait is the execution of instructions used to make the process wait for a condition to be fulfilled in order to go on doing actual progress in the program.

```
while not continue do;
```

- So the process uses processor time by interleaving statements which “do nothing”.
- Thus, busy-wait is known to be a very inefficient way to make a process wait.
- **Problems of busy-wait:**
 - Processes doing busy-wait are wasting processor time which could be used by other processes willing to do useful work.
 - A processor working consumes energy and generates heat.
- It is necessary to find another approach to make processes wait

2.4 Busy wait VS. passive wait

- A processes which uses **passive-wait** (also called **blocked-wait**) enters in state *blocked* (or similar). In that state, the process does not execute any instruction.
- A process exits the *blocked* state due to an action of another process (probably change of one condition variable). (see Unit 1, states diagrams).
- Clearly, passive-wait is more efficient than busy-wait.
- **Busy-wait**, as a means to achieve synchronization, is **recommended only** when the programming environment does not provide passive-wait tools or primitives.

2.4 Busy wait VS. passive wait

- Primitive calls, methods or objects provided by some languages allow a **greater abstraction for synchronization** than using busy-wait-based protocols.
- Among these programming tools, some of them may still make our code confusing and error-prone.
- Not all synchronization tools are available in all programming languages.
- **Synchronization tools** are available for two models of communication:
 - Shared-memory
 - Semaphores
 - Critical Regions
 - Conditional Critical Regions (CCR)
 - Monitors
 - Message Passing
 - (A)synchronous message passing
 - Remote invocation

- 2.1 Introduction**
- 2.2 Condition Synchronization**
- 2.3 Mutual Exclusion**
 - 2.3.1 First Attempt**
 - 2.3.2 Second Attempt**
 - 2.3.3 Third Attempt**
 - 2.3.4 Fourth Attempt**
 - 2.3.5 Dekker's Algorithm**
 - 2.3.6 Several Critical Sections**
- 2.4 Busy wait VS. passive wait**
- 2.5 Conclusions**

2.5 Conclusions

- Processes may need to be synchronized in order to:
 - start/end an action (Condition Synchronization)
 - access a shared resource (mutual exclusion)
- In any programming environment, **mutual exclusion** access to a critical section can be achieved using **protocols** which make use of shared variables and busy-wait.
- In order to **say these protocols are correct**, they must:
 - Grant mutual exclusion
 - Avoid livelock
 - Avoid starvation
 - (avoid deadlock, but none of the solutions introduced fall in deadlock)

2.5 Conclusions

- **Passive-wait** primitives are available in **some concurrent programming languages** to achieve a more efficient execution, and to make our code easier to read and less error-prone.
- **You are lucky**, we will study these tools in the following units!

Potential Midterm Exam Questions

1. What interactions between processes need synchronization?
2. What do we mean when we say that a process running a busy-wait is *doing nothing*?
3. In program *ConditSynch* (section 2.2), what interleaving of instructions leads to output “aS1 as2 bS1 bS2”?

	Process A	Process B	continue
1			
2			
3			
4			
5			
6			

Potential Midterm Exam Questions

4. What are the requirements for a correct solution of mutual exclusion synchronization?

5. What problems can you find in this attempt of achieving mutual exclusion?

```
program FirstAttempt;
var turn: Integer;







process P1;
begin
    repeat
        while turn <> 1 do;
            writeln('P1 is in CS');
            writeln('P1 is leaving CS');
            turn:=2;
            writeln('P1 is in non-CS');
        forever
    end;

process P2;
begin
    repeat
        while turn <> 2 do;
            writeln('P2 is in CS');
            writeln('P2 is leaving CS');
            turn:=1;
            writeln('P2 is in non-CS');
        forever
    end;
```

Potential Midterm Exam Questions

6. From the point of view of atomic execution, what can we say about critical sections?
7. What is the upper bound of critical sections in a program?

Keywords phonetics

- attempt /ə'tempt/ 
- renounce /rɪ'naʊns/ 
- mutual /'mju:tʃuəl/ 
- exclusion /ɪk'sklu:ʒən/ 
- coarse /kɔ:s/ 
- critical /'krɪtɪkəl/ 
- section /'sekʃən/ 
- contention /kən'tenʃən/ 