

Unit 3

Shared-memory Communication

3.1 SEMAPHORES

3.1 Semaphores

3.1.1 Introduction

3.1.2 Mutual Exclusion

3.1.3 Condition Synchronization

3.1.4 Producer-Consumer Problem

3.1.5 The Dining Philosophers

3.2 CCR

3.3 Monitors

3.1 Semaphores

3.1.1 Introduction

3.1.2 Mutual Exclusion

3.1.3 Condition Synchronization

3.1.4 Producer-Consumer Problem

3.1.5 The Dining Philosophers

3.2 CCR

3.3 Monitors

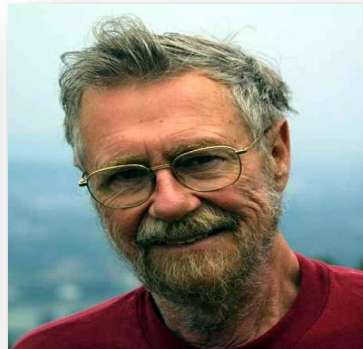
3 Shared-memory Communication

- In Unit 2, we learned techniques based on busy-wait which allowed the programmer to synchronize processes so that:
 - Processes do not execute some action if a condition is not met
 - Processes access shared resources under mutual exclusion
- We learned that passive-wait (blocked process) is much more efficient than using busy-wait and protocols.
- In multiprogramming, processes share memory. This fact has been used to develop tools which make synchronization easier and less error-prone.
- The most important and most frequently found in concurrent languages, are:
 - Semaphores
 - Conditional Critical Regions
 - Monitors

ordered increasingly by level of abstraction.

3.1.1 Introduction to semaphores

- Dijkstra (1968) creates the first synchronization primitive tool using passive-waiting: semaphores.
- A **semaphore** is a low level tool which helps the programmer achieve conditional synchronization among processes and access to critical regions under mutual exclusion.



Dijkstra
(1930-2002)

3.1.1 Introduction to semaphores

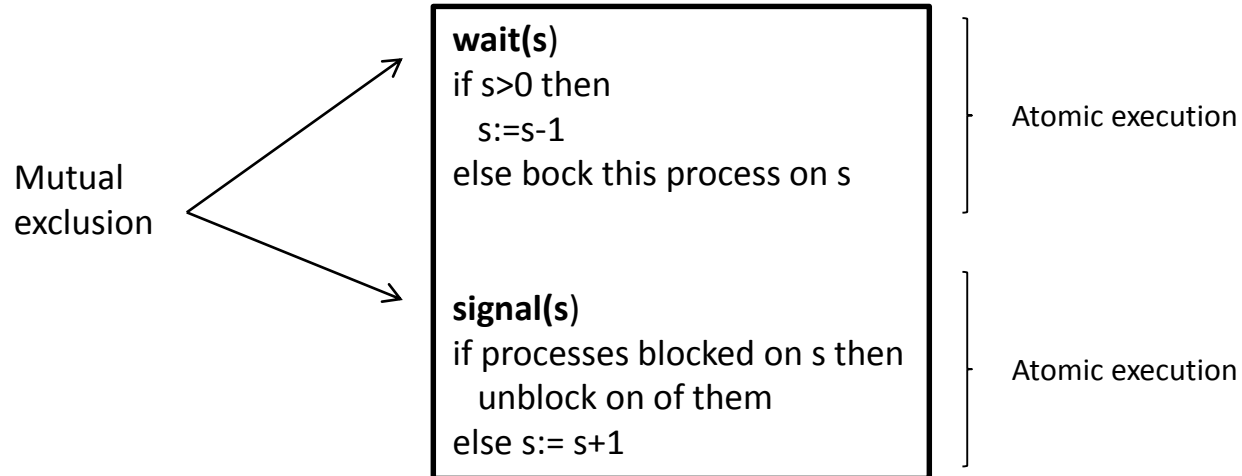
- A semaphore may be implemented as a structured data type or as an object, depending on the programming language.
- In order to **implement a semaphore from scratch**, we need:
 - A private counter of *permits*.
 - A private set of *blocked processes*
 - *Public methods* which change the counter and, according to the new value, may make changes in the state of processes (block / unblock).
- **Permissible values** for the counter of permits depends of the **type of semaphore**:
 - General or counting semaphore: non-negative integer value.
 - Binary semaphore: 0 or 1. Only 1 bit of storage is required!
- In practice, we refer to the current value of the counter of permits by saying “the value of the semaphore”. So, given semaphore s , “ $s > 0$ ” stands for “the counter of permits in s is greater than 0”.

3.1.1 Introduction to semaphores

- **Permissible operations.** There are only 2 operations which processes can carry on semaphores: *wait(s)* and *signal(s)*.
- **wait(s):** decreases *s* or blocks the process.
 - If $s > 0$, $s := s - 1$
 - If $s = 0$, block the process and queue it in blocked processes set.
- **signal(s):** increases *s* or unblocks a process.
 - If there are blocked processes in *s*, unblock one process.
 - else, $s := s + 1$
- A call to any of these operations involves testing the value of *s* or the set of *blocked processes* and, when appropriate, perform a modification. Semaphores grant that all this is executed as **an indivisible action**. Internally, **this is not implemented with busy-wait** but using OS functions to block and grant mutual exclusion of both operations..

3.1.1 Introduction to semaphores

- Having 2 operations called on a semaphore, op1 and op2, both are to be executed but we cannot predict the order. We just know that they will be run under mutual exclusion.



- In the signal operation, if there are more than 1 blocked process, the way to **select the process to be unblocked** depends on the implementation of the semaphore: FIFO , priorities, random (Pascal-FC),...

3.1.1 Introduction to semaphores

- If **signal** is called on a **binary semaphore** which already has a **value of 1**, in most programming languages this is tackled by becoming this call a no-operation; that is, it has no effect.
- **Caution**, the name of operation *wait* maybe a bit misleading. A process calling *wait* only does wait (blocks) if $s=0$. That is, if it cannot take any permit.
- In Java:
 - keywords *wait* and *signal*, without parameters, are reserved for processes signaling. They can be regarded as **semaphore with 0 permits** by default in any execution. We will work with them in lab assignments.
 - Object *java.util.concurrent.Semaphore* provides the corresponding methods *acquire* and *release*.

3.1.1 Introduction to semaphores

- Pascal-FC provides non-negative general semaphores. So if you want a binary semaphore, it is your responsibility to correctly deploy wait and signal operations.
- Examples for **declaring** variables, an **array** and a **record** using type **semaphore**.

```
var
  s1,s2: semaphore;
  semArray: array [1..5] of semaphore;
  semRec: record
    i: integer;
    s: semaphore;
  end;
```

- **Restrictions** in declaring semaphores:
 - they may be declared
 - in the **global declaration space right before the main** block and pass them by reference .
 - In the **global declaration space of the program**.
 - In order to pass them by reference, do not forget to use the **var** modifier.

3.1.1 Introduction to semaphores

- Why do you *think* semaphores need to be passed as reference?
- Procedure **initial**(*s, value*) initiates semaphore *s* to the given value.
- Semaphores must be **initiated in the main block** and out of the **cobegin-coend** keywords .
- Pascal provides the 2 semaphore operations, implemented as procedures:
 - **wait**(*s*) and **signal**(*s*),
 - Where *s* is of type **semaphore**.
- Procedure **write** can use a semaphore as argument to output its value on screen.
- **Java** provides class `java.util.concurrent.Semaphore`.
Semaphore mutex=**new** Semaphore(1); *// binary semaphore*

3.1 Semaphores

3.1.1 Introduction

3.1.2 Mutual Exclusion

3.1.3 Condition Synchronization

3.1.4 Producer-Consumer Problem

3.1.5 The Dining Philosophers

3.2 CCR

3.3 Monitors

3.1.2 Mutual Exclusion

- By using semaphores, we can **forget about using protocols** to access and exit critical sections.
- Given a critical section CS, we achieve mutual exclusion access by:
 - using a binary semaphore s
 - calling *wait(s)* right before entering CS
 - calling *signal(s)* right after exiting CS
- If $s=1$, CS is free.
- If $s=0$, a process is inside CS.
- If we have **several critical sections**, we need the same binary semaphore for each subset of critical sections which need mutual exclusion among them.

3.1.2 Mutual Exclusion (Pascal-FC)

```
program mutexSem;
var
  cont : integer;

process type MyProcess(var sem: semaphore; id:integer);
begin
  repeat
    wait(sem);

    (* CS *)
    cont := cont +1;
    writeln('[',id,'] ',cont);

    signal(sem);
    (* non-CS*)

  forever
end;

var
  p1,p2:MyProcess;
  mutex:semaphore;

begin
  initial(mutex,1);
  cont := 0;
  cobegin
    p1(mutex,1);
    p2(mutex,2);
  coend;
end.
```

What do you think the output is?

What may happen if we move the write command out of CS?

3.1.2 Mutual Exclusion (Java)

```
class Counter {  
  
    int cont = 0;  
    Semaphore mutex = new Semaphore(1);  
  
    void increment(int id) {  
        try {  
            mutex.acquire();  
        } catch (InterruptedException e) {  
            e.printStackTrace();  
        }  
        cont++;  
        System.out.println "[" + id + " " + cont);  
        mutex.release();  
    }  
}  
  
class counterThread extends Thread {  
    Counter c;  
    int ID;  
  
    counterThread(Counter counter, int id) {  
        c = counter;  
        ID = id;  
    }  
  
    public void run() {  
        for (int i = 0; i < 100; i++) {  
            c.increment(ID);  
        }  
    }  
}
```

```
public class mutexSem {  
    public static void main(String args[]) {  
        Counter counter = new Counter();  
        for (int i = 0; i < 5; i++)  
            (new counterThread(counter, i)).start();  
    }  
}
```

*Do you think the same reference to mutex
is shared among processes?*

3.1.2 Mutual Exclusion

- **General semaphores** can be used for sharing a given number of available instances of a **resource**; or to provide **multiple exclusion** (more than 1 process in CS).
- E.g.: number of tickets for a concert; number of chairs in a room;...
- Thus:
 - Binary semaphore: mutual exclusion
e.g. pascal-FC `initial(mutex,1);`
 - General semaphore:
 - Share or allocate resources.
e.g. pascal-FC `initial(tickets,5);`
 - Multiple exclusion: $N > 1$ processes in CS
e.g. pascal-FC `initial(multiplex,3);`

3.1.2 Mutual Exclusion

- When a general semaphore is used to **share resources** (e.g. chair in waiting room), once one instance of a resource is returned the process must call *signal(s)*.
- If a general semaphore is used to **allocate resources** which are never returned (e.g. books for sale), then *signal(s)* is never called. If there are more processes than the initial value of the semaphore, some processes will never get the resource and fall in deadlock.

3.1 Semaphores

3.1.1 Introduction

3.1.2 Mutual Exclusion

3.1.3 Condition Synchronization

3.1.4 Producer-Consumer Problem

3.1.5 The Dining Philosophers

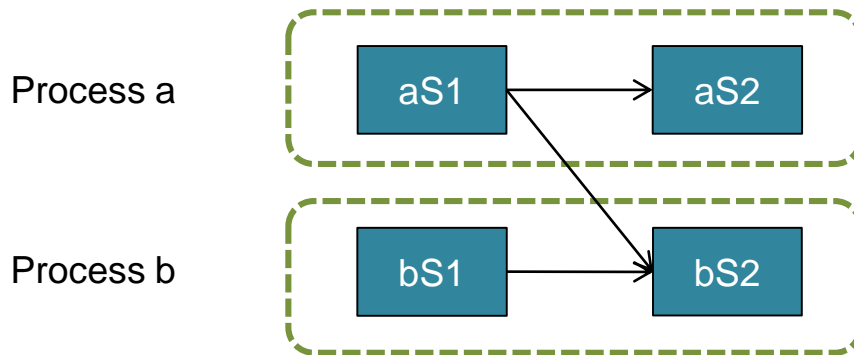
3.2 CCR

3.3 Monitors

3.1.3 Condition Synchronization

- Remember from Unit 2 that a condition synchronization happens when one process halts waiting for a condition to be met thanks to the action of another process.

A possible solution for this precedence diagram is the use of busy-wait:



```
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;
end;

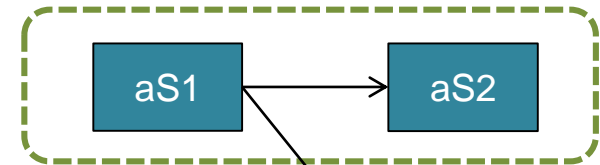
begin
    continue := false;
    cobegin
        ProcessA;
        ProcessB;
    coend
end.
```

3.1.3 Condition Synchronization

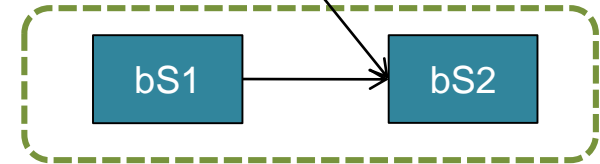
Solution using a binary semaphore in Pascal-FC

```
program synchronization;  
  
var aSldone: semaphore;  
  
process ProcessA;  
begin  
    write('aS1 ');  
    signal(aSldone);  
    write('aS2 ');  
end;  
  
process ProcessB;  
begin  
    write('bS1 ');  
    wait(aSldone);  
    write('bS2 ');  
end;  
  
begin  
    initial(aSldone, 0);  
cobegin  
    ProcessA;  
    ProcessB;  
coend  
end.
```

Process a



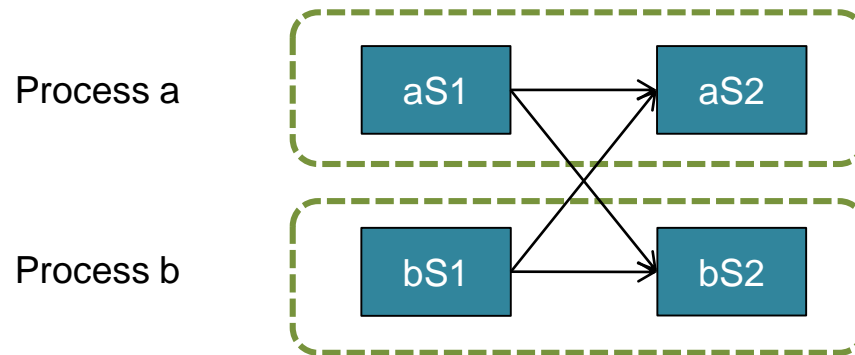
Process b



Try to code this solution in Java at home!

3.1.3 Condition Synchronization

- A **rendezvous** is a meeting between 2 processes: both wait for each other



aS2 and bS2 will not be written until aS1 and bS1 are.

```
program rendezvous;

var aS1done: semaphore;
var bS1done: semaphore;

process ProcessA;
begin
    write('aS1 ');
    signal(aS1done);
    wait(bS1done);
    write('aS2 ');
end;
```

```
process ProcessB;
begin
    write('bS1 ');
    signal(bS1done);
    wait(aS1done);
    write('bS2 ');
end;

begin
    initial(aS1done, 0);
    initial(bS1done, 0);
cobegin
    ProcessA;
    ProcessB;
coend
end.
```

Semaphores

To achieve a rendezvous, each process must *signal* its semaphore (announce itself) and then *wait* on the semaphore of the other process.

Java code this solution at home!

3.1.3 Condition Synchronization

- A **barrier**
 - is a type of condition synchronization in which processes are blocked in the same point until all of them reach that point.
 - Is a generalization of rendezvous, which only works for 2 processes.
- Pascal-FC does not have a primitive type for barriers.
- A common way to implement a barrier using semaphores is using the **turnstile solution**: once all processes are blocked in the same statement, pairs of wait-signal are executed to let all processes go through a binary semaphore.
- E.g.: 5 processes write 'A'. When all of them finish, then they write B.

Draw the precedence diagram

3.1.3 Condition Synchronization (Pascal-FC)

```
program turnstileBarrier;

const NPR=5;
var
  mutex : semaphore;
  Sbarrier: semaphore;
  pCounter: integer;

process type writer;
begin
  write('A');
  wait(mutex);  (*mutex to increment counter*)
  pCounter := pCounter + 1;
  signal (mutex);

  if pCounter = NPR then signal(Sbarrier);
  (*turnstile: once a process is unblocked
  fromt wait, the rest will be unblocked
  one after the other*)
  wait(Sbarrier);
  signal(Sbarrier);

  write('B');
end;
```

```
var
  i:integer;
  writers:array [1..5] of writer;

begin

  pCounter := 0;
  initial(Sbarrier,0);
  initial(mutex,1);
  cobegin
    for i:=1 to 5 do
      writers[i];
    coend
end.
```

3.1.3 Condition Synchronization (Java)

```
import java.util.concurrent.CyclicBarrier ;

class Writer extends Thread{
    CyclicBarrier barrier;

    Writer(CyclicBarrier b){
        barrier=b;
    }
    public void run(){
        System.out.println("A");
        try {
            barrier.await();
        } catch (Exception e) {
            e.printStackTrace();
        }
        System.out.println("B");
    }
}

public class Barrier {
    static final int NPR=5;

    public static void main(String args[]){

        CyclicBarrier cb=new CyclicBarrier(NPR);

        for(int i=0;i<NPR;i++) (new Writer(cb)).start();
    }
}
```

*The concurrent package of Java
Provides an object CyclicBarrier
which abstracts us from the barrier
implementation*

3.1.3 Condition Synchronization (Java)

- A **cyclic barrier** allows to re-use the barrier again after the waiting threads are released.
- For example, it can be used in a loop to repeat the AAAAABBBBBB printing 2 times.

```
public void run() {  
    for (int loop = 0; loop < 2; loop++) {  
        System.out.println("A");  
        try {  
            barrier.await();  
            System.out.println("B");  
            barrier.await();  
        } catch (Exception e) {  
            e.printStackTrace();  
        }  
    }  
}
```

This code gives a new precedence restraint: the second set of 'A' cannot be printed until the previous set of 'B' are written. Otherwise, we would only get the first iteration right.

3.1.3 Condition Synchronization (Pascal-FC)

- Implementing a cyclic barrier in Pascal-FC is trickier because we need to build the barrier using semaphores.
- We need:
 - 2 semaphore-barriers
 - 2 turnstile protocols
 - Decrease the counter before going through the second turnstile protocol
 - Block the second semaphore-barrier before unblocking the first.
 - Block the first semaphore-barrier before unblocking the second.

3.1.3 Condition Synchronization (Pascal-FC)

```
program cyclicBarrier;
const NPR=5; ITERATIONS=3;
var
  mutex, Sbarrier1, Sbarrier2 : semaphore;
  pCounter: integer;

process type rewriter;
  var it:integer;
  begin
    for it:=1 to ITERATIONS do
      begin
        write('A');
        wait(mutex);  (*mutex to increment counter*)
        pCounter := pCounter + 1;
        if pCounter = NPR then
          begin
            wait(Sbarrier2); {block Sbarrier2}
            signal(Sbarrier1); {unblock Sbarrier1}
          end;
        signal (mutex);

        wait(Sbarrier1); (*1st turnstile*)
        signal(Sbarrier1);

        write('B');
        wait(mutex);  (*mutex to decrement counter*)
        pCounter := pCounter - 1;
        if pCounter = 0 then
          begin
            wait(Sbarrier1); {block Sbarrier1}
            signal(Sbarrier2); {unblock Sbarrier2}
          end;
        signal(mutex);

        wait(Sbarrier2); (*2nd turnstile*)
        signal(Sbarrier2);

      end;
    end;
end;
```

```
var
  i:integer;
  rewriters:array [1..5] of rewriter;
begin

  pCounter := 0;
  initial(Sbarrier1,0);
  initial(Sbarrier2,1);
  initial(mutex,1);
  cobegin
    for i:=1 to 5 do
      rewriters[i];
    coend
end.
```

Why do you think mutex is used for both the increment and decrement?

Why do you think each mutex controls the access to six lines of code?

3.1.3 Condition Synchronization

- Thus, semaphores can be used to provide:
 - Mutual exclusion in access to shared resources
 - Synchronize processes
- There exist several classic Concurrent Programming problems which need to be solved using both mutual exclusion and synchronization.
- We will solve 2 problems:
 - Producer-Consumer
 - The dining philosophers

3.1 Semaphores

3.1.1 Introduction

3.1.2 Mutual Exclusion

3.1.3 Condition Synchronization

3.1.4 Producer-Consumer Problem

3.1.5 The Dining Philosophers

3.2 CCR

3.3 Monitors

3.1.4 Producer-Consumer problem

- This problem can be set with different levels of difficulty. We will solve an advanced version of the problem, with 4 producers, 3 consumers and a buffer of Size 8.
- Producers insert data in a circular buffer.
 - Only 1 producer can insert an item in a given slot.
 - A position cannot be written if it is being read.
 - If the buffer is full, no item can be inserted.
- Consumers remove data from the circular buffer.
 - Remove in a FIFO manner
 - 2 consumers cannot remove an item from the same slot
 - A slot cannot be consumed if it is being written.
 - A consumer cannot remove data from an empty buffer.

*Identify which restrictions need condition synchronization, and which need mutual exclusion.
Do we need more than 1 semaphore for mutual exclusion?*

3.1.4 Producer-Consumer problem

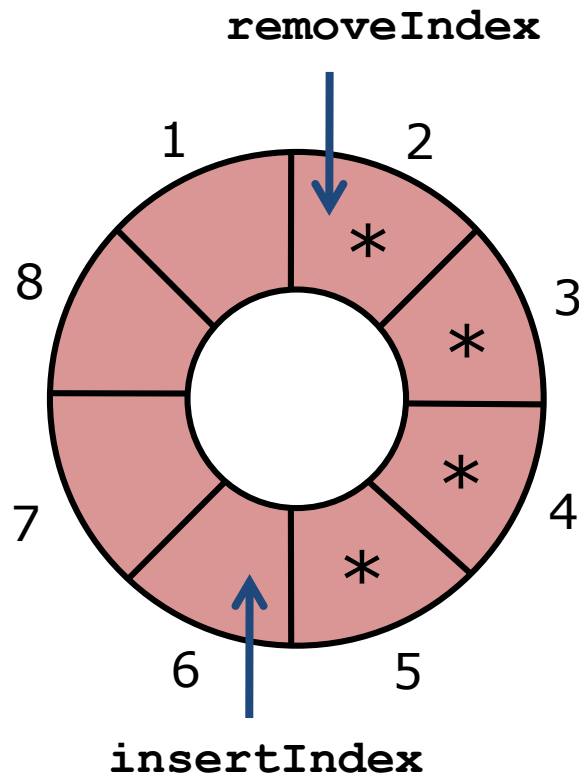
Necessary data:

removeIndex \rightarrow integer

insertIndex \rightarrow integer

buffer \rightarrow record.

slots \rightarrow array of integer variables



Necessary semaphores:

Mutual exclusion \rightarrow binary

Counter of empty slots \rightarrow general

Counter of available items \rightarrow general

*Why do we use semaphores to count,
Instead of integer variables?*

3.1.4 Producer-Consumer problem

```

program producerConsumer;

const SIZE=8;
{new buffer type}
type tBuffer = record
    data: array [1..SIZE] of integer;
    insertIndex, removeIndex: integer;
    Sitems, Sempty, Smutex: semaphore;
end;

procedure init(var buffer:tBuffer);
begin
    buffer.insertIndex := 1;
    buffer.removeIndex := 1;
    initial(buffer.Sitems,0);
    initial(buffer.Sempty,SIZE);
    initial(buffer.Smutex,1);
end;

```

What operations on buffer are protected by mutual exclusion?

```

procedure insert(item:integer; var buffer:tBuffer);
begin
    {block if buffer has no empty slots}
    wait(buffer.Sempty);
    wait(buffer.Smutex);
    buffer.data[buffer.insertIndex]:= item;
    writeln('-->',item);
    buffer.insertIndex:=buffer.insertIndex MOD SIZE + 1;
    signal(buffer.Smutex);
    {increase the counter of items}
    signal(buffer.Sitems);
end;

procedure remove(var item:integer; var
buffer:tBuffer);
begin
    {block if buffer has 0 items}
    wait(buffer.Sitems);
    wait(buffer.Smutex);
    item := buffer.data[buffer.removeIndex];
    writeln('<--',item);
    {our array starts by index 1}
    buffer.removeIndex:=buffer.removeIndex MOD SIZE +1;
    signal(buffer.Smutex);
    {increase the counter of empty slots}
    signal(buffer.Sempty);
end;

```

3.1.4 Producer-Consumer problem

```
process type tProducer
    (var buffer:tBuffer);

var
    item: integer;
begin
    repeat
        item := random(200);
        insert(item,buffer);
    forever
end;

process type tConsumer
    (var buffer:tBuffer);

var
    item: integer;
begin
    repeat
        remove(item,buffer);
        writeln(item);
    forever
end;
```

```
var
    buffer:tBuffer;
    i:integer;
    prod:array [1..5] of tProducer;
    cons:array [1..3] of tConsumer;
begin
    init(buffer);
    cobegin
        for i:=1 to 5 do
            prod[i](buffer);

            for i:=1 to 3 do
                cons[i](buffer);
            coend;
        end.
```

3.1.4 Producer-Consumer problem

- A similar solution can be implemented in Java using class Semaphore, and creating class Buffer instead of a record.
- In order to gain mutual exclusion, Java provides the keyword *synchronized*.
- Methods in the same class with **synchronized** modifier are executed under mutual exclusion. That is, only 1 thread can exist at a time in all synchronized methods.
- The Producer-Consumer problem is now solved using this keyword.
- In order to block and resume a thread, Java provides methods **wait()**, **notify()**, **notifyAll()**. Do not confuse them with semaphores procedures. They are signals, which will be taught in more detail in Lab Sessions.
- We cannot know which thread is going to be resumed by *notify()*.

3.1.4 Producer-Consumer problem

```

class Buffer {
    static final int SIZE = 8;
    int insertIndex, removeIndex, empty, items;
    int[] data;

    Buffer() {
        data = new int[SIZE];
        insertIndex = 0;
        removeIndex = 0;
        empty = SIZE;
        items = 0;
    }

    synchronized void insert(int item)
        throws InterruptedException {

        // block if buffer has no empty slots
        while (empty == 0) {
            wait();
        }
        data[insertIndex] = item;
        //arrays start by index 0
        insertIndex = (1+insertIndex)%SIZE;

        items++; //increase counter of items
        notifyAll();
    }

```

```

    synchronized int remove()
        throws InterruptedException {

        // block if buffer has 0 items
        while (items == 0) {
            wait();
        }
        int x = data[removeIndex];
        removeIndex = (1+removeIndex)%SIZE;

        empty++; //increase counter of empty slots

        notifyAll();
        return x;
    }
}

```

Why do we call notifyAll instead of notify?

Why do we enclose wait in a while loop instead of an if condition?

3.1.4 Producer-Consumer problem

```
class Producer extends Thread {
    Buffer buffer;

    Producer(Buffer b) {
        buffer = b;
    }

    public void run() {
        java.util.Random r = new java.util.Random();
        while (true) {
            try {
                buffer.insert(r.nextInt(200));
            } catch (InterruptedException e) {
                e.printStackTrace(); } } }

    }

class Consumer extends Thread {
    Buffer buffer;
    Consumer(Buffer b) {
        buffer = b;
    }

    public void run() {
        while (true) {
            try {
                int x = buffer.remove();
                System.out.println(x);
            } catch (InterruptedException e) {
                e.printStackTrace(); } } }

    }
```

```
public class ProducerConsumer {

    public static void main(String args[])
    {

        Buffer buffer = new Buffer();
        for (int i = 0; i < 4; i++)
            (new Producer(buffer)).start();
        for (int i = 0; i < 3; i++)
            (new Consumer(buffer)).start();
    }

}
```

3.1 Semaphores

3.1.1 Introduction

3.1.2 Mutual Exclusion

3.1.3 Condition Synchronization

3.1.4 Producer-Consumer Problem

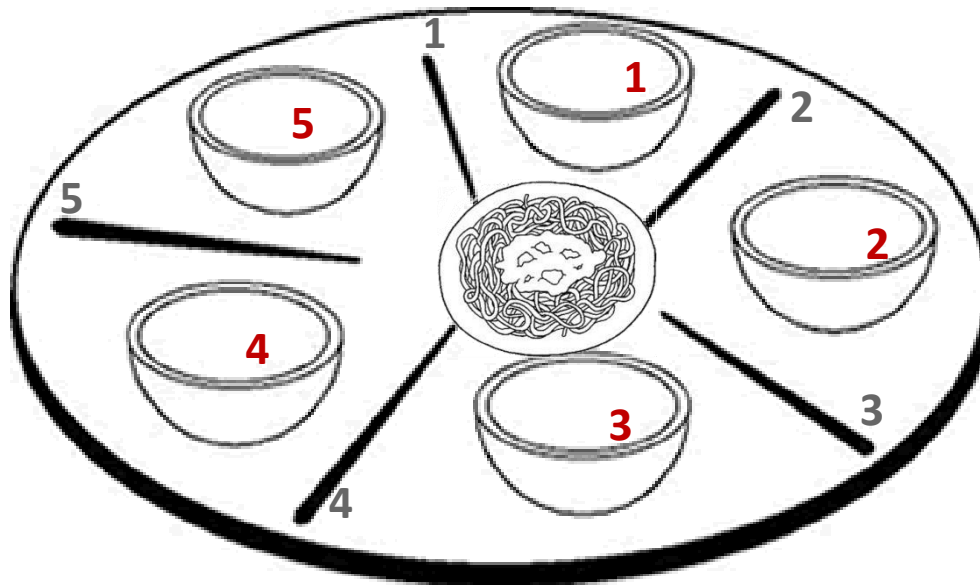
3.1.5 The Dining Philosophers

3.2 CCR

3.3 Monitors

3.1.5 The dining philosophers

- Five philosophers are engaged in only two activities: thinking and eating.
- Meals are taken at a table set with five plates and five chopsticks. In the center of the table is a bowl of spaghetti that is endlessly replenished.
- When a philosopher is hungry, he sits down and **uses the chopsticks on his left and right sides**. Thus, two philosophers sitting together cannot eat at the same time.



Chopsticks are the shared resources

- **Philosopher i takes chopstick i and $i+1$**

3.1.5 The dining philosophers

- Being each philosopher a process, they will do the following actions:

```
process type philosopher(id:: integer);  
begin  
  repeat  
    think;  
    sit;  
    take left and right chopsticks  
    eat;  
    release chopsticks  
  forever  
end;
```

- How can we code the *think*, *sit* and *eat* actions?
- **The core problem is how to synchronize them when taking the chopsticks**

3.1.5 The dining philosophers

- The given solution should meet the following criteria:
 1. One chopstick is hold by just 1 philosopher: mutual exclusion
 2. One philosopher can eat only if he holds the left and right chopsticks: condition synchronization.
 3. Free from deadlock and livelock
 4. Free from starvation
 5. *If possible, be efficient: more than 1 philosopher should be able to eat at the same time.*
- We will implement 2 solutions:
 1. Mutex semaphores for chopsticks (produces deadlock)
 2. Allow only 4 philosopher to be sitting (it is not efficient)

3.1.5 The dining philosophers – Sol.1

```

program diningPhilosophersSol1;
{Sol1: mutex chopsticks}
const N=5; {5 philosophers}
var chopstick: array[1..N] of semaphore;

process type tPhilosopher(id: integer);
begin
  repeat
    sleep(random(2)); {THINK and SIT}
    wait(chopstick[id]);
    wait(chopstick[(id MOD N)+1]);
    writeln('[', id, '] eating;');
    sleep(random(2)); {EAT}
    signal(chopstick[id]);
    signal(chopstick[(id MOD N)+1]);
  forever;
end;

var
  phils : array[1..N] of tPhilosopher;
  i: integer;
begin
  for i:=1 to N do
    initial(chopstick[i], 1);
  cobegin
    for i:=1 to N do
      phils[i](i);
    coend;
end.

```

*If the interleaving of instructions
Is such that all philosophers take their
left chopstick, they will wait forever
eating for their right chopstick: deadlock*

*Sol.2: allow only N-1 philosophers to sit down at
the same time.*

*What kind of semaphore do we use to implement
this idea: only N-1 chairs available.*

3.1.5 The dining philosophers – Sol.2

PASCAL-FC

```
program diningPhilosophersSol2;
{Sol2: mutex chopsticks, and N-1 chairs}
const N=5; {5 philosophers}
var
  chopstick: array[1..N] of semaphore;
  chairs: semaphore;

process type tPhilosopher(id: integer);
begin
  repeat
    sleep(random(2)); {THINK}
    wait(chairs); {SIT}
    wait(chopstick[id]);
    wait(chopstick[(id MOD N)+1]);
    writeln('[', id, '] eating;');
    sleep(random(2)); {EAT}
    signal(chopstick[id]);
    signal(chopstick[(id MOD N)+1]);
    signal(chairs);
  forever;
end;
Var

  phils : array[1..N] of tPhilosopher;
  i: integer;
begin
  for i:=1 to N do
    initial(chopstick[i], 1);
  initial(chairs, N-1);
  cobegin
    for i:=1 to N do
      phils[i](i);
    coend;
end.
```

This solution is correct.

At any moment, at least 1 philosopher will be able to eat.

But having just 1 out of N philosophers eating is not efficient.

There exist more efficient solutions, which can be presented as Task 1:

Sol3:

- *Odd philosophers first wait on left chopstick.*
- *Even philosophers first wait on right chopstick.*

Sol4:

-The last philosopher first waits on a different chopstick than the others.

3.1.5 The dining philosophers

- Next, Solution 2 is implemented in Java.
- Chopsticks are classes which provide methods *take()* and *release()*.
- Actions on chairs are performed through the methods provided in class Chairs.
- Philosophers are threads.
- Mutual exclusion is necessary in actions on chopsticks and chairs. This can be achieved by using object Semaphore, or with the *synchronized* modifier. We will use the latter.

3.1.5 The dining philosophers Sol.2

```
class Chopstick{
    boolean free=true;

    synchronized public void take() {
        while(!free) {
            try{
                wait();
            } catch (Exception e) {
                e.printStackTrace();
            }
            free=false;
        }

        synchronized public void release() {
            free=true;
            notifyAll();
            //all blocked phil. will try to
            //take it again
        }
    }
}
```

```
class Chairs{

    int max, busy;

    Chairs(int m) {
        max=m;
        busy=0;
    }

    synchronized public void sit() {
        while(busy==max) {
            try{wait();
            } catch (Exception e) {
                e.printStackTrace();
            }
            busy++;
        }

        synchronized public void standUp() {
            busy--;
            notifyAll();
        }
    }
}
```

3.1.5 The dining philosophers Sol.2

```
class Philosopher extends Thread{
    int id;
    Chopstick rightCh, leftCh;
    Chairs chairs;

    public Philosopher(int ID, Chairs c,
        Chopstick r, Chopstick l){
        id=ID;
        chairs=c;
        rightCh=r;
        leftCh=l;
    }

    public void run(){
        java.util.Random r=new java.util.Random();
        while(true){
            try { //THINK
                Thread.sleep(r.nextInt(2000));
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
            chairs.sit(); //SIT
            rightCh.take();
            leftCh.take();
            System.out.println("[ "+id+" ] eating."); //EAT
            rightCh.release();
            leftCh.release();
            chairs.standUp();
        }
    }
}
```

```
public class DiningPhilosophersSol2{
    public static final int N=5;

    public static void main(String args[]){
        Chairs chairs=new Chairs(N-1);
        Chopstick chopstick[]=new Chopstick[N];

        for(int i=0;i<N;i++){
            chopstick[i]=new Chopstick();
        }

        for(int i=0;i<N;i++){
            new Philosopher(i, chairs, chopstick[i],
                chopstick[(i+1)%N]).start();
        }
    }
}
```

Conclusions

You may have already reached the following conclusion:

- Semaphores have advantages:
 - Easy and efficient solution for mutual exclusion and synchronization
 - Their primitives are available in most of concurrent programming languages
- But they have several disadvantages:
 - Their level of abstraction is low
 - Error-prone: their use depends on the programmer (lost signal...)
 - Readability and maintenance of the code is not straightforward (they are deployed along the code without any structure).
 - At first glance, we cannot tell which semaphores are used for mutual exclusion or which for synchronization.
- These disadvantages (specially, the last one) are alleviated by using CCR and Monitors. But they are not available in all languages!

Potential Mid-term Exam Questions

- 1. What is a cyclic barrier? How can we use it in Pascal-FC and Java?
- 2. If we were to implement a semaphore, what variables and procedures do we need to code?
- 3. What is the most general purpose of binary semaphores? And general?

Potential Mid-term Exam Questions

- 4. What effect do *wait*, *notify* and *notifyAll* signals have on threads in Java?
- 5. Read again Solution 1 of the Dining Philosophers problem. Write a possible interleaving which leads to a deadlock situation

Keywords phonetics

- semaphore /'seməfɔːr / 
- procedure /prə'siːdʒər/ 
- rendezvous /'rɒndeɪvuː/ 
- cyclic /'saɪklɪk/ 
- barrier /'bæriər/ 
- philosopher /fɪ'lɒsəfər/ 
- signal /'sɪgnl/ 
- binary /'baɪnəri/ 
- multiple /'mʌltɪpl/ 