

Unit 3

Shared-memory Communication

3.3 Monitors

3.1 Semaphores

3.2 CCR

3.3 Monitors

3.3.1 Introduction

3.3.2 Monitors

-Mutual Exclusion

-Condition Synchronization

3.3.3 Producer-Consumer problem

3.3.4 Proposed problems

3.3.5 Monitors in Java

3.1 Semaphores

3.2 CCR

3.3 Monitors

3.3.1 Introduction

3.3.2 Monitors

- Mutual Exclusion

- Condition Synchronization

3.3.3 Producer-Consumer problem

3.3.4 Proposed problems

3.3.5 Monitors in Java

3.3.1 Introduction

- We learned that semaphores are concurrent programming solutions which had several problems. Can you remember a few of them?
- CCR solved some of these problems:
- But still provided a resulting code with sparse use of global variables aimed to control the concurrent execution.
- Solution: Monitors

3.3.1 Introduction

- Monitors were proposed by C.A.R. Hoare in 1974.
- A **monitor** is an encapsulation of resources, and operations which can be applied on them.
- Variables declared in a monitor can be accessed **only** from a procedure **exported** from the same monitor.
- All procedures are executed under **mutual exclusion**; the programmer does not need to explicitly manage access to critical regions anymore.
- Processes are queued inside a **Condition queue**, when a condition is not met. It will wait until another process makes the necessary change.

3.3.1 Introduction

- Consequently, a **monitor** solves all problems mentioned when using semaphores and CCR:
 - **Modular**: the code written to control concurrent processes is inside a unique structure.
 - Local variables: variables used are local in the monitor. **The programmer will not find global variables** along the code.
 - The programmer cannot access, by error, shared variables.
 - **Meaningful**: the code inside the monitor helps the programmer to know what is the aim of the monitor (mutual exclusion or synchronization).
- **Active process**: a process which calls a procedure inside the monitor. We say the process is *inside* the monitor.
- If the monitor creates processes inside it, these are called **passive processes**.

3.1 Semaphores

3.2 CCR

3.3 Monitors

3.3.1 Introduction

3.3.2 Monitors

-Mutual Exclusion

-Condition Synchronization

3.3.3 Producer-Consumer problem

3.3.4 Proposed problems

3.3.5 Monitors in Java

3.3.2 Monitors

- Monitors are available as primitive in Pascal-FC, but not in Java. The generic structure in Pascal-FC is:

```
monitor name;  
  export exported_procedures;  
  var local_variables;  
  
  procedure P1 (parameters);  
    var local_variables;  
    begin  
      {some code}  
    end;  
  ...  
  procedure Pn (parameters);  
    var local_variables;  
    begin  
      {some code}  
    end;  
begin  
  {initiation code}  
end;
```

- A set of local variables called **permanent** variables. They indicate the state of the resource represented by the monitor.
- An **initiation code**. This code is executed once, when the monitor is created. It is used to initiate the permanent variables.
- One or more procedures which manage the value of the permanent variables.
- A list of **exported procedures** which can be accessed from outside the monitor by the programmer. The *export* keyword must be the first to appear in the monitor code.
- Procedures which are not exported are inner procedures.

3.3.2 Monitors

- When an **active process needs to use a shared resource** which is represented/controlled by a monitor, the process **must call an exported procedure** from the monitor.
- This call is performed in Pascal-FC as follows:
monitorName.procedure(arguments)
- That is, the name of the monitor followed by the name of the exported procedure (and arguments, if required).
- Let us see how the monitor provides:
 - mutual exclusion access to procedures inside the monitor, and
 - processes synchronization

3.3.2 Monitors – mutual exclusion

- A monitor uses a processes queue called **monitor queue**.
- The monitor queue is managed as follows:
 - When an active process is inside the monitor, and **another active process tries to enter** the monitor through another (or the same) exported procedure than the former, the latter is blocked in the monitor queue (which has a FIFO behaviour).
 - When an **active process finishes** the execution of an exported procedure, it leaves the monitor. Then:
 - If the monitor queue is empty, the first active process which tries to gain access will enter.
 - Else, the process in the head of the monitor queue is released and gains access to the monitor.

3.3.2 Monitors – mutual exclusion

- Write a Pascal-FC monitor which lets us increment the value of a variable and printing its value. The access must be performed under mutual exclusion.

```
program incrementing;

monitor monit;
export inc, value;
var i:integer;

procedure inc;
begin
  i:=i+1;
end;

procedure value;
begin
  writeln('---->',i)
end;

begin
  i:=0;
end;
```

```
process type P;
begin
  repeat
    monit.inc;
    monit.value;
  forever
end;

var
  p1,p2,p3:P;
begin
  cobegin
    p1;
    p2;
    p3;
  coend
end.
```

3.3.2 Monitors – condition synch.

- The previous example allows us to increment and print the value of a variable under mutual exclusion. But the increment&printing operations are not synchronized, so the output may be a little 'dumb'.
- We need to learn how to synchronize processes under a given condition inside monitors.
- This is performed by using **condition variables**: variables inside a monitor which represent FIFO queues.
- An active process is **blocked in a condition variable** when it cannot continue its execution (e.g.: producer blocked because buffer is full). It will resume its execution when the situation changes (e.g.: the consumer takes an item from the buffer).

3.3.2 Monitors – condition synch.

- Declaration in Pascal-FC of 2 condition variables and an array of condition variables:

```
var  
cond_A, cond_B: condition;  
conditions: array [1..5] of condition;
```

- **Disambiguation** of term “condition”:
 1. The situation which makes a process block until it is changed by another process.
 2. A type of variable which represents a FIFO queue. If there are several kinds of processes, then each kind of process is queue in a different condition variable. For example, in the Producer/Consumer problem, producers are blocked in condition variable A, and consumers in condition variable B.

3.3.2 Monitors – condition synch.

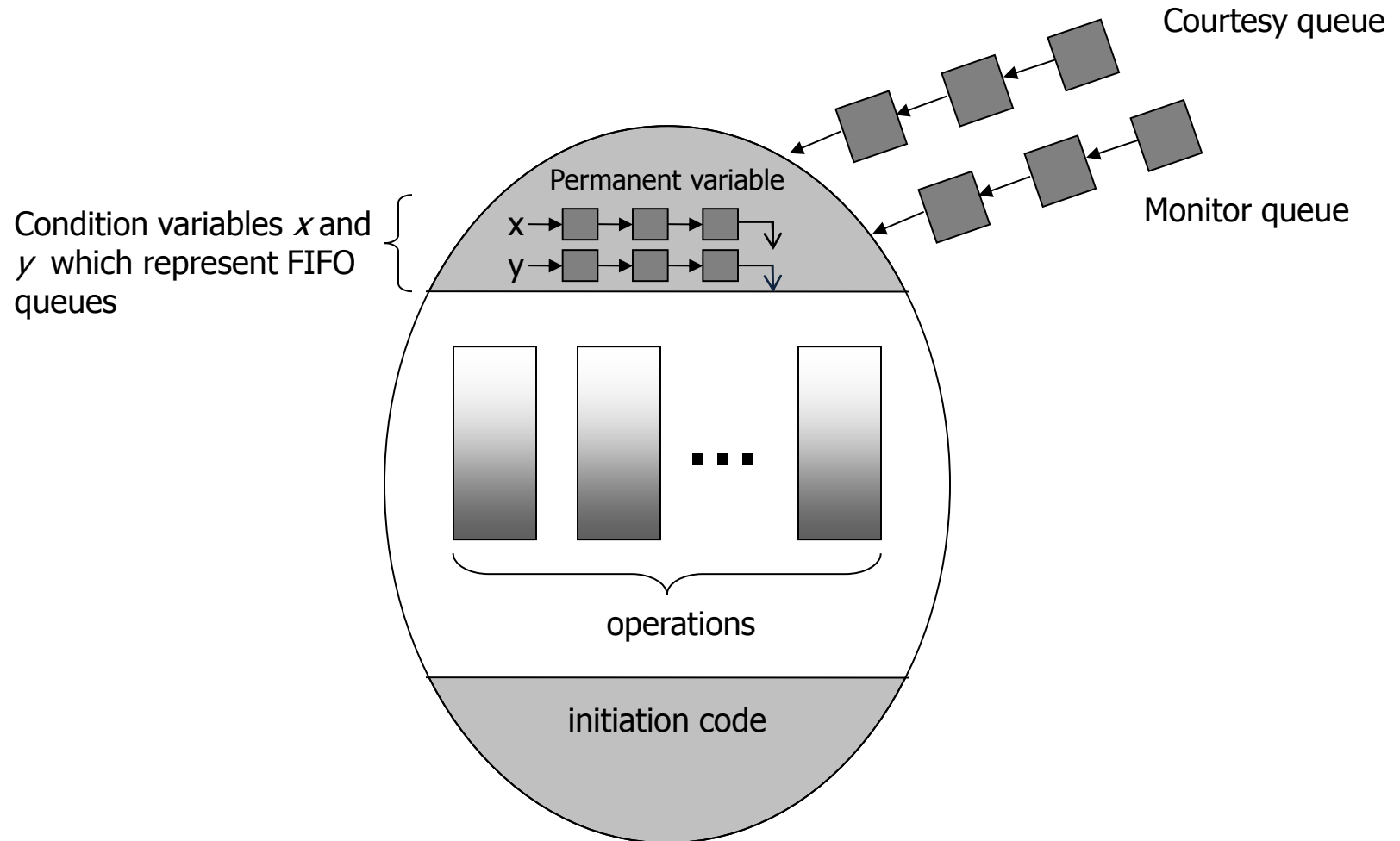
- Pascal-FC provides three necessary operations on a condition variable C.
 - **empty**(C): it returns a boolean value. True if there are not active processes blocked in C, False otherwise.
 - **delay**(C): the active process which executes this operation **releases the mutual exclusion** hold on the monitor, **and is queued** in condition C. This is different from *wait(semaphore)* because the active process is always blocked in the condition, while semaphores only block processes when their value is 0.
 - **resume**(C): the active **process blocked in the head** of the queue in C is **set ready** for execution. If the queue is empty, this operation is null (does nothing); this is different from *signal(semaphore)* because signal always increases the value of the semaphore so it is never a null operation.

3.3.2 Monitors – condition synch.

- When an active process P1 executes *resume(C)* and then P2 is set ready for execution, which process goes on running: P1 or P2? (both together cannot because this violates the mutual exclusion guaranteed by the monitor).
- The possible solutions are known as **semantic of the resume operation**.
 - **Resume and Continue (RC)**: P2 is inserted back in the monitor queue. The situation which blocked it needs to be reevaluated in a while loop because once it re-enters the monitor, the situation might be true again. Java wait-notify signals use the semantic RC.
 - **Resume and Exit (RE)**: P1 returns from the monitor and P2 resumes its execution inside the monitor. Thus, **P2** does not need to reevaluate the situation. Concurrent Pascal uses RE.
 - **Resume-and-Wait (RW)**: P1 returns from the monitor and is inserted again in the monitor queue. P2 resumes its execution. Modula-2, Concurrent Euclid.
 - **Resume-and-Urgent-Wait (RUW)**: the same than RW, but now P1 is inserted in the **courtesy queue**, which has higher priority than the monitor queue when trying to gain access to the monitor. This is the semantic of resume in Pascal-FC.

3.3.2 Monitors – condition synch.

- Thus, the final structure of a monitor (with RUW semantic) is this:



3.1 Semaphores

3.2 CCR

3.3 Monitors

3.3.1 Introduction

3.3.2 Monitors

-Mutual Exclusion

-Condition Synchronization

3.3.3 Producer-Consumer problem

3.3.4 Proposed problems

3.3.5 Monitors in Java

3.3.3 Producer-Consumer problem

Let us solve the problem with limited buffer and several producers and consumers

- What resource is to be represented by the monitor?
- What are the permanent variables?
- What are the active processes?
- Do we need condition variables to queue active processes?
- What operations should the monitor export?

3.3.3 Producer-Consumer problem

```
program ProducerConsumer;  
  
monitor buffer;  
{list of operations to export}  
export insert, remove;  
{permanent variables}  
const SIZE=10;  
var  
    numItems, insertIndex, removeIndex:integer;  
    consumersC, producersC: condition;  
    data: array[1..SIZE] of integer;  
  
    {operations}  
    procedure insert(item:integer);  
    begin  
        if numItems=SIZE then delay(producersC);  
        data[insertIndex]:=item;  
        writeln('--->',item);  
        insertIndex:= (insertIndex MOD SIZE) + 1;  
        numItems:=numItems+1;  
        resume(consumersC);  
    end;
```

```
    procedure remove(var item:integer);  
        {the item is passed by reference}  
    begin  
        if numItems=0 then delay(consumersC);  
        item:=data[removeIndex];  
        writeln('<---',item);  
        removeIndex:=(removeIndex MOD SIZE)+1;  
        numItems:=numItems-1;  
        resume(producersC);  
    end;  
  
begin {initiation code}  
    insertIndex:=1;  
    removeIndex:=1;  
    numItems:=0;  
end;
```

3.3.3 Producer-Consumer problem

Now write the code necessary to run the solution with 2 consumers and 2 producers

```
process type tProducer;
var
  item: integer;
begin
  repeat
    item := random(200);
    buffer.insert(item);
  forever
end;

process type tConsumer;
var
  item: integer;
begin
  repeat
    buffer.extract(item);
  forever
end;
```

```
var
  {*the buffer monitor is a global variable
  we do not need to declare it*}
  consumer1, consumer2:tProducer;
  producer1,producer2: tConsumer;
begin
  cobegin
    consumer1;
    consumer2;
    producer1;
    producer2;
  coend
end.
```

3.1 Semaphores

3.2 CCR

3.3 Monitors

3.3.1 Introduction

3.3.2 Monitors

-Mutual Exclusion

-Condition Synchronization

3.3.3 Producer-Consumer problem

3.3.4 Proposed problems

3.3.5 Monitors in Java

3.3.4 Proposed problems

- Using Pascal-FC:
 - Implement a binary semaphore using monitors.
 - Solve the Elevator problem using monitors

Always think:

- What resource is to be represented by the monitor?
- What are the permanent variables?
- What are the active processes?
- Do we need condition variables to queue active processes?
- What operations should the monitor export?

3.1 Semaphores

3.2 CCR

3.3 Monitors

3.3.1 Introduction

3.3.2 Monitors

-Mutual Exclusion

-Condition Synchronization

3.3.3 Producer-Consumer problem

3.3.4 Proposed problems

3.3.5 Monitors in Java

3.3.5 Monitors in Java

- Java language does not provide monitors as built-in objects.
- Think of a class with a set of **private variables**, and **several public synchronized methods** which operate on that variable:
 - that is a monitor which guarantees mutual exclusion operations on such variables.
 - Why?
- But, what about condition synchronization?
 - You can use signals *wait* and *notify* to block processes but...
 - Can you see a problem here?

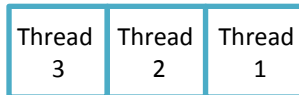
3.3.5 Monitors in Java

- By default, signals block and resume threads on the same queue.
- If we want to imitate the behaviour of **condition variables**, we can call these **signals on different Object** objects.

Thread 1 Thread 2 Thread 3



```
synchronized public void queueMe() throws Exception{  
    wait();  
}
```



Thread 1 Thread 2 Thread 3



```
public void queueMe(Object ob) throws Exception{  
    //imagine each thread passes a different Object instance  
    synchronized(ob){ob.wait();}  
}
```



- Since we are using more than 1 condition variable, the synchronization must be done per blocks instead of the whole method. And then each block is synchronized on the Object on which a wait or signal may be called.

3.3.5 Monitors in Java

- Now we can build a monitor for the buffer of the Producer-Consumer problem.

```
class BufferMonitor {
    private final int SIZE=10;
    private Object consumersC = new Object();
    private Object producersC = new Object();
    private int insertIndex=0,removeIndex=0,numItems=0;
    private int[] data=new int[SIZE];

    public void insert(int item){
        synchronized(producersC){
            //Note:we need to reevaluate the condition
            //because JAVA uses Resume and Continue semantic
            while(numItems==SIZE)
                try {producersC.wait();}
                catch (InterruptedException e) {
                    e.printStackTrace();
                }
            data[insertIndex]=item;
            System.out.println("--->" + item);
            insertIndex= (insertIndex +1) % SIZE;
            numItems++;
        }

        synchronized(consumersC){
            consumersC.notify();
        }
    }
}
```

```
    public int remove(){
        int item;
        synchronized(consumersC){
            while(numItems==0){
                try {
                    consumersC.wait();
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
            }
            item=data[removeIndex];
            System.out.println("<---" + item);
            removeIndex=(removeIndex+1)%SIZE;
            numItems--;
        }

        synchronized(producersC){
            producersC.notify();
        }
        return item;
    }
}
```

3.3.5 Monitors in Java

```
class Producer extends Thread {
    BufferMonitor buffer;

    Producer(BufferMonitor b) {
        buffer = b;
    }

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

class Consumer extends Thread {
    BufferMonitor buffer;

    Consumer(BufferMonitor b) {
        buffer = b;
    }

    public void run() {
        while (true) {
            int x = buffer.remove();
        }
    }
}
```

```
public class ProducerConsumer{

    public static void main(String args[]){

        BufferMonitor buffer = new BufferMonitor();

        //SHARED INSTANCE OF THE BUFFERMONITOR!!
        for (int i = 0; i < 2; i++)
            (new Producer(buffer)).start();
        for (int i = 0; i < 2; i++)
            (new Consumer(buffer)).start();




    }

}
```

Potential Mid-term Exam Questions

1. What is the structure of a monitor which provides condition synchronization?
2. What can you say about a Java class with a private variable and a set of public and synchronized methods?
3. What do we mean by 'semantic' of the resume operation on blocked processes?
4. How many queues do we have in a monitor?

Keywords phonetics

- active /'æktɪv/ 
- disambiguation /ˌdɪsæmbɪɡjʊ'eɪʃən/ 
- resume /rɪ'zju:m/ 
- courtesy /'kɜ:təsi/ 