

# Unit 3

## Shared-memory Communication

### 3.2 CONDITIONAL CRITICAL REGIONS

#### 3.1 Semaphores

#### 3.2 CCR

##### 3.2.1 Introduction

##### 3.2.2 Critical Region

##### 3.2.3 Conditional Critical Region

##### 3.2.4 Producer-Consumer problem

##### 3.2.5 Readers-Writers problem

##### 3.2.6 The dining philosophers problem

#### 3.3 Monitors

### 3.1 Semaphores

### 3.2 CCR

#### 3.2.1 Introduction

#### 3.2.2 Critical Region

#### 3.2.3 Condition Critical Region

#### 3.2.4 Producer-Consumer problem

#### 3.2.5 Readers-Writers problem

#### 3.2.6 The dining philosophers problem

### 3.3 Monitors

## 3.2.1 Introduction

- Semaphores:
  - are low level abstraction tools for mutual exclusion and synchronization.
  - their syntax is the same when they are used for both kinds of interaction among processes.
  - It is easy to forget one wait or signal operation.
- In 1972, Hoare and Brinch Hansen proposed and made popular the notion of Conditional Critical Region (CCR).
- CCRs tell the compiler where the mutual exclusion (CR) and condition synchronization (CCR) statements are, and it deals with the deployment of wait and signal operations.

### 3.1 Semaphores

### 3.2 CCR

#### 3.2.1 Introduction

#### 3.2.2 Critical Region

#### 3.2.3 Condition Critical Region

#### 3.2.4 Producer-Consumer problem

#### 3.2.5 Readers-Writers problem

#### 3.2.6 The dining philosophers problem

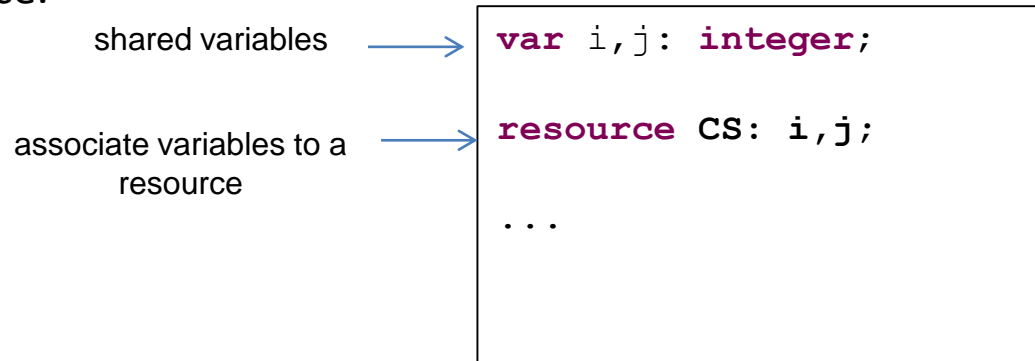
### 3.3 Monitors

## 3.2.2 Critical Region

- A critical section is a piece of code that should be executed under mutual exclusion. It depends on the programmer that this happens.
- A **critical region (CR)** is a piece of code that is executed under mutual exclusion. The programmer does not need to take care of protocols nor correct use of wait/signal calls.
- CR and CCRs can be seen as precursor of monitors, and they are available just in a few programming languages (e.g. Ada 9X).
- CR and CCR are not available in Pascal-FC nor Java. We will learn its use using Pascal-FC syntax and reserved word *resource*, but it must be read just as pseudo-code.

## 3.2.2 Critical Region

- Shared variables must be declared as usual, and then associate them to a resource.



- Since variables are attached to a resource, they can only be accessed by explicitly using the keyword *region*, and the name of the resource which holds the variable.

```
region CS do  
begin  
  i := i+1;  
  ...  
end;
```

## 3.2.2 Critical Region

- All **regions with the same resource** name will be executed under mutual exclusion.
- If a variable associated to a shared resource is **accessed directly** from code, the compiler flags an error.
- One variable can only be **associated to 1 resource**.
- **Nested** CRs may lead to a deadlock situation:

```
process p1:  
...  
region A do  
    region B do  
        S;
```

```
process p2:  
...  
region B do  
    region A do  
        S;
```

### 3.1 Semaphores

### 3.2 CCR

#### 3.2.1 Introduction

#### 3.2.2 Critical Region

#### 3.2.3 Conditional Critical Region

#### 3.2.4 Producer-Consumer problem

#### 3.2.5 Readers-Writers problem

#### 3.2.6 The dining philosophers problem

### 3.3 Monitors

## 3.2.3 Conditional Critical Region

- So CRs provide mutual exclusion by making the compiler be responsible of the use of semaphores.
- A **Conditional Critical Region (CCR)** is an extension of CR which lets express a condition to gain access to the resource: condition synchronization.
- Resources are declared the same way than CRs.
- Access to resources now includes a condition:

```
var i,j: integer;  
  
resource CCS i,j;  
...  
region CCS when <condition> do  
begin  
...  
end;
```

## 3.2.3 Conditional Critical Region

- Thus, a process trying to enter a CCR runs as follows:
  1. A process remains blocked in the **main queue** until it gains mutual exclusion access to the region.
  2. Once access is obtained,
    1. if *condition* is true, statements in the CCR are executed.
    2. else, the process releases the mutual exclusion access and is blocked in the **events queue**.
  3. Once the execution of statements finishes:
    1. Processes in the events queue are allowed to test the condition again. If the condition is met, the first process is unblocked.
    2. Else, the first process in the main queue is unblocked.
- This means that a process which has gained mutual exclusion access once is not required to gain it again. That is, it has a higher priority than new incoming processes.

## 3.2.3 Conditional Critical Region

- In order to give a full example, let's see how to implement semaphores using CCR:
  - One shared variable to be used as counter of permits
  - Wait can only be executed under a given condition if we do not allow negative semaphores.
  - Signals do not need condition

```
program semaphore;  
var  
  s: integer;  
  resource sem : s;  
  
procedure wait;  
begin  
  {if s=0, process is blocked in the events queue}  
  region sem when s>0 do  
    s:= s-1;  
  end;
```

```
procedure signal;  
begin  
  region sem do  
    s:= s+1;  
  end;
```

### 3.1 Semaphores

### 3.2 CCR

#### 3.2.1 Introduction

#### 3.2.2 Critical Region

#### 3.2.3 Conditional Critical Region

#### 3.2.4 Producer-Consumer problem

#### 3.2.5 Readers-Writers problem

#### 3.2.6 The dining philosophers problem

### 3.3 Monitors

## 3.2.4 Producer-Consumer problem

- Again, we instantiate the problem with a finite buffer, and several producers and consumers.
- Critical sections are those in which the following variables are accessed:  
*insertIndex, removeIndex, numItems*
- We define 2 condition synchronizations between processes:
  - Producers cannot insert items if it is full.
  - Consumers cannot remove items if it is empty.

## 3.2.4 Producer-Consumer problem

```
program producerConsumerCCR;
const SIZE=8;
var
  numItems, insertIndex,
      removeIndex, i: integer;
data: array[1..SIZE] of integer;
resource buffer: insertIndex,
               removeIndex, numItems;

process type tProducer;
var
  item: integer;

begin
  repeat
    item := random(200);
    region buffer when numItems < SIZE do
      begin
        data[insertIndex]:=item;
        insertIndex:=insertIndex MOD SIZE + 1;
        numItems:=numItems+1;
      end;
    forever
  end;
```

```
process type tConsumer;
begin
  var
    item: integer;
  repeat
    region buffer when numItems>0 do
      begin
        item := data[removeIndex];
        removeIndex := removeIndex MOD SIZE+ 1;
        numItems:=numItems-1;
      end;
    forever
  end;

var
  prod:array [1..5] of tProducer;
  cons:array [1..3] of tConsumer;
begin
  numItems:=0;
  insertIndex:=0;
  removeIndex:=0;
  cobegin
    for i:=1 to 5 do
      prod[i];
    for i:=1 to 3 do
      cons[i];
    coend;
end.
```

### 3.1 Semaphores

### 3.2 CCR

#### 3.2.1 Introduction

#### 3.2.2 Critical Region

#### 3.2.3 Conditional Critical Region

#### 3.2.4 Producer-Consumer problem

#### 3.2.5 Readers-Writers problem

#### 3.2.6 The dining philosophers problem

### 3.3 Monitors

## 4.2.5 Readers/Writers problem

- In this problem, several readers and writers access to the same resource (e.g. database).
- Several readers can read at the same time if no writers are writing.
- If one writer is writing, no reader nor extra writer can access.
- The solution is slightly different depending on which kind of process has priority when both are waiting to access:
  - Readers first
  - Writers first
- In this example, we decide to give priority to writers:
  - if a writer wants to access, it waits until current readers stop reading. Then no reader may enter until the waiting writer finishes its action.
  - We need 3 variables:
    - numReaders: number of processes of type Reader reading
    - numWwaiting: num of processes of type Writer waiting to access
    - isWriting: true when a writer is accessing it

## 4.2.5 Readers/Writers problem

```
program readersWriters;
{PRIORITARY WRITERS}
var
  numReading, numWwaiting: integer;
  isWriting: boolean;
  resource data: numReading, numWwaiting,
                 isWriting;

process type tReader;
begin
  (*readers wait if a writer is
  writing or waiting*)
  region data when not isWriting and
    numWwaiting=0 do
    numReading:=numReading+1;
    writeln('Reading...');
  region data do
    numReading:=numReading-1;
end;
```

Thanks to the unqueue FIFO policy of the events queue, starvation of readers is not possible (it is using semaphores).

```
process type tWriter;
begin
  {announce a new writer is waiting}
  region data do
    numWwaiting:=numWwaiting+1;
  {wait if a writer or reader is in}
  region data when numReading=0 and
    not isWriting
  begin
    isWriting:=true;
    numWwaiting := numWwaiting-1;
  end;
  writeln('Writing...');
  region data do isWriting:=false;
end;

var
  read:array[1...5] of tReader;
  wri:array[1...3] of tWriter;
  i:integer;
begin
  numReading:=0;
  isWriting:=false;
cobegin
  for i:=1 to 5 do
    read[i];
  for i:=1 to 3 do
    wri[i];
  coend
end.
```

### 3.1 Semaphores

### 3.2 CCR

#### 3.2.1 Introduction

#### 3.2.2 Critical Region

#### 3.2.3 Conditional Critical Region

#### 3.2.4 Producer-Consumer problem

#### 3.2.5 Readers-Writers problem

#### 3.2.6 The dining philosophers problem

### 3.3 Monitors

## 3.2.6 The dining philosophers problem

- Critical sections: those in which chopsticks are taken.
- Cond. Synch: philosopher  $i$  cannot eat if he cannot take chopstick  $i$  and  $i+1$

```
program diningPhilosophers;
const N=5;
var
{true when they are available}
chopstick:array[1..N] of boolean;
resource chopsCCR:chopstick;

process type tPhilosopher(id:integer);
begin
repeat
  sleep(random(2)); {THINK and SIT}
  region chopsCCR when chopstick[id] and
    chopstick[(id+1) MOD N] do
    begin
      chopstick[id]:=false;
      chopstick[(id+1)MOD N]:=false;
    end;
  sleep(random(2)); {EAT}
  region chopsCCR do
    begin
      chopstick[id]:=true;
      chopstick[(id+1)MOD N]:=true;
    end;
  forever
end;
```

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

Using semaphores (solution 1) deadlock Could happen.  
Using CCRs this is not possible, since a philosopher only enters in the CCR when both chopsticks are available.

# Conclusions

- CCRs let us distinguish between critical sections and condition synchronization.
- They may be difficult to implement (two process queues)
- Deadlock risk when embedding CCRS
- They still keep one problem of semaphores: their use is widespread along the code: difficult to maintain.



# Potential Mid-term Exam Questions

1. Which processes queue has higher priority in CCR?
2. What correctness problem do CCRs solve in the Dining Philosophers problem which needed a special solution when using semaphores?
3. What is the difference between CR and CCR?

# The Elevator problem

- An elevator has a capacity of 4 people.
- Several people call it from different levels.
- A person leaves the elevator when it reaches its level.
  
- A constant is the number of levels
- Several processes of type Person. Each process has a variable which indicates the level it wants to access to and the level from which it is calling.
- One process is the elevator:
  - with boolean variable: up. (false means down).
  - an integer with current level (starts in 0).
  - Num of people inside.
  
- What are the CS?
- What are the cond. synch.?
- Pseudo-code the solution: available as Task 2 (see Unit Zero).

# Keywords phonetics

- region /'ri:dʒən/ 
- resource /rɪ'zɔ:s/ 
- priority /praɪ'ɒrɪtɪ/ 