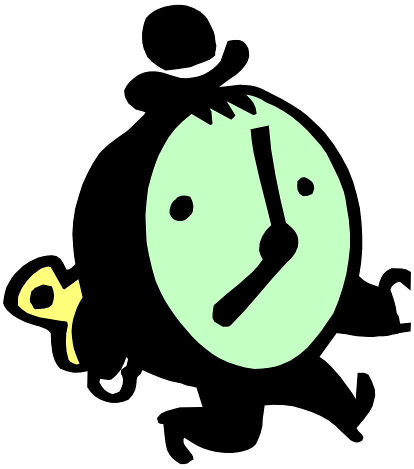
Real-Time Systems Programming

Summer-Semester 2002
Lecture 20
21 June 2002



Synchronization and Communication Part 2

Overview

1) Language support for synchronization
2) Communication and synchronisation based on message passing

Where are we?

1) ***Language support for synchronization***
   - ***Ada 95: protected objects***
   - ***Java: synchronized methods***
2) Communication and synchronisation based on message passing

Ada: Protected Objects

- A ***protected object***:
  - Encapsulates data items and allows access to them only via protected actions — ***protected subprograms*** or ***protected entries***
  - The language guarantees that the data will only be updated under ***mutual exclusion***, and that all data read will be internally consistent
  - May be declared as a type or as a single instance

Syntax

```
protected type Name (Discriminant) is
  function Fname(Params)
           return Type_Name;
  procedure Pname(Params);
  entry E1_Name(Params);

private
  entry E2_Name(Params);
  O_Name : T_Name;
end Name;
```

Only subprograms and entries

Only subprograms, entries and object declarations

No type declarations

Protected Types and Mutual Exclusion

```
protected type Shared_Data(Initial : Data_Item) is
  function Read return Data_Item;
  procedure Write (New_Value : in Data_Item);

private
  The_Data : Data_Item := Initial;
end Shared_Data_Item;
protected body Shared_Data_Item is
  function Read return Data_Item is
  begin
    return The_Data;
  end Read;

  procedure Write (New_Value : in Data_Item) is
  begin
    The_Data := New_Value;
  end Write;
end Shared_Data_Item;
```

Protected Procedures and Functions

- A protected procedure provides ***mutually exclusive read/write access*** to the data encapsulated
- Concurrent calls to **`Write`** will be executed one at a time
- Protected functions provide ***concurrent read only*** access to the encapsulated data
- Concurrent calls to **`Read`** may be executed simultaneously
- Procedure and function calls are mutually exclusive
- The core language does not define which calls take ***priority***

Protected Entries and Synchronisation

- A protected entry is similar to a protected procedure
  - Calls are executed in mutual exclusion
  - Have read/write access to the data
- A protected entry can be guarded by a boolean expression (called a ***barrier***)
- If this barrier evaluates to false when the entry call is made:
  - Calling task is suspended and remains suspended while
    - barrier evaluates to false, ***or***
    - other tasks currently active inside the protected unit
- Hence protected entry calls can be used to implement ***condition synchronisation***

Ada Bounded Buffer I

```
-- a bounded buffer
Buffer_Size : constant Integer :=10;
type Index is mod Buffer_Size;
subtype Count is Natural range 0 .. Buffer_Size;
type Buffer is array (Index) of Data_Item;

protected type Bounded_Buffer is
  entry Get (Item : out Data_Item);
  entry Put (Item : in Data_Item);

private
  First : Index := Index'First;
  Last : Index := Index'Last;
  Num : Count := 0;
  Buf : Buffer;
end Bounded_Buffer;
```

Ada Bounded Buffer II

```
protected body Bounded_Buffer is

  entry Get (Item : out Data_Item)
        when Num /= 0 is
  begin
    Item := Buf(First);
    First := First + 1;
    Num := Num - 1;
  end Get;

  entry Put (Item : in Data_Item)
        when Num /= Buffer_Size is
  begin
    Last := Last + 1;
    Buf(Last) := Item
    Num := Num + 1;
  end Put;
end Bounded_Buffer;

My_Buffer : Bounded_Buffer;
```

The Readers and Writers Problem

- ***Example:*** a file which needs mutual exclusion
  - between writers and reader
  - not between multiple readers
- Protected objects can implement the readers/writers algorithm if
  - Read operation is encoded as a function ***and***
  - Write encoded as a procedure
- *However:*
  - ***Cannot easily control the order of access***
  - Cannot prefer writes over reads
  - If the read or write operations are ***potentially blocking***, then they cannot be made from within a protected object
- Must implement ***access control protocol*** for the read and write operations (rather than encapsulate them)

Readers/Writers I

```
with Data_Items; use Data_Items;
package Readers_Writers is
  -- for some type Item
  procedure Read (I : out Item);
  procedure Write (I : Item);
end Readers_Writers;
package body Readers_Writers is

  procedure Read_File(I : out Item) is separate;
  procedure Write_File(I : Item) is separate;

  protected Control is
    entry Start_Read;
    procedure Stop_Read;
    entry Request_Write;
    entry Start_Write;
    procedure Stop_Write;

  private
    Readers : Natural := 0; -- no. of current readers
    Writers : Boolean := False; -- Writers present
  end Control;
```

Readers/Writers II

```
procedure Read (I : out Item) is
begin
  Control.Start_Read;
    Read_File(I);
  Control.Stop_Read;
end Read;

procedure Write (I : Item) is
begin
  Control.Request_Write; -- indicate writer present
  Control.Start_Write;
    Write_File(I);
  Control.Stop_Write;
end Write;
```

Readers/Writers III

```
  protected body Control is
     entry Start_Read
         when not Writers and Request_Write'Count = 0 is
     begin Readers := Readers + 1; end Start_Read;

     procedure Stop_Read is
     begin Readers := Readers - 1; end Stop_Read;

     entry Request_Write
         when not Writers is
     begin Writers := True; end Request_Write;

     entry Start_Write
         when Readers = 0 is
     begin null; end Start_Write;

     procedure Stop_Write is
     begin
       Writers := False;
     end Stop_Write;
  end Control;
end Readers_Writers;
```

Ada: Protected Objects

- A ***protected object***:
  - Encapsulates data items and allows access to them only via protected actions — ***protected subprograms*** or ***protected entries***
  - The language guarantees that the data will only be updated under ***mutual exclusion***, and that all data read will be internally consistent
  - May be declared as a type or as a single instance

- Ada's protected objects are similar to objects – however, they do not support inheritance
- Java provides a mechanism by which monitors can be implemented in the context of classes and objects

Java: Synchronized Methods

- Java provides a mechanism by which monitors can be implemented in the context of classes and objects
- There is a ***lock*** associated with each object which cannot be accessed directly by the application but is affected by
  - the method modifier **`synchronized`**
  - block synchronization

- When a method is labeled **`synchronized`**:
- Access to the method can only proceed once the lock associated with the object has been obtained
- Hence synchronized methods have ***mutually exclusive access*** to the data encapsulated by the object, if that data is only accessed by other synchronized methods
- ***However:***
  - Non-synchronized methods do not require the lock and, therefore, can be called at any time

Example of Synchronized Methods

```
class SharedInteger
{
  private int theData;

  public SharedInteger(int initialValue)
  { theData = initialValue; };

  public synchronized int read()
  { return theData; };

  public synchronized void write(int newValue)
  { theData = newValue; };

  public synchronized void incrementBy(int by)
  { theData = theData + by; };
}

SharedInteger myData = new SharedInteger(42);
```

Block Synchronization

- The **`synchronized`** keyword takes as a parameter an object whose lock it needs to obtain before it can continue
- Hence synchronized methods are effectively implementable as:

```
public int read()
{
  synchronized(this) {
    return theData;
  }
}
```

Warning

- Monitor-like mechanisms:
  - Encapsulate synchronization constraints associated with an object into a single place in the program
  - Can understand the synchronization associated with a particular object by just looking at the object itself
- However, this can be undermined with synchronized block:
  - Other objects can name an object in a synchronized statement
  - No ***composability***

- However with careful use, this facility augments the basic model and allows more expressive synchronization constraints to be programmed

Static Data

- Static data are shared between ***all*** objects created from the class
- To obtain mutually exclusive access to this data:
  - ***All*** objects must be locked
- In Java, classes themselves are also objects and therefore there is a lock associated with the class
- This lock may be accessed by
  - labeling a static method with the **`synchronized`** modifier, ***or***
  - by identifying the class's object (the **`Object`** class associated with the object) in a synchronized block statement
- *However:* this class-wide lock is not obtained when synchronizing on the object

Static Data Example

```
class StaticSharedVariable
{
  private static int shared;
  ...

  public synchronized int Read()
  {
    synchronized(this.getClass())
    {
      return shared;
    };
  }

  public static void Write(int I)
  {
    synchronized(this.getClass())
    {
      shared = I;
    };
  };
}
```

Could have used:

```
public static synchronized void Write(int I)
```

Waiting and Notifying

- To obtain conditional synchronization requires the methods provided in the predefined object class

```
public void wait();
            // throws IllegalMonitorStateException
public void notify();
            // throws IllegalMonitorStateException
public void notifyAll();
            // throws IllegalMonitorStateException
```

- These methods should be used only from within methods which ***hold the object lock***
- If called without the lock, the exception **`IllegalMonitorStateException`** is thrown

- The `wait` method always blocks the calling thread and releases the lock associated with the object
- The `notify` method wakes up one waiting thread (Java does not define which one; RT Java does)
  - Does not release the lock
  - The woken thread must still wait until it can obtain the lock before it can continue
- The `notifyAll` method wakes up *all* waiting thread
  - All awoken threads must contend for lock when it becomes free

Java Bounded Buffer I

```
public class BoundedBuffer {
  private int buffer[];
  private int first;
  private int last;
  private int numberInBuffer = 0;
  private int size;

  public BoundedBuffer(int length) {
    size = length;
    buffer = new int[size];
    last = 0;
    first = 0;
  };
```

Java Bounded Buffer II

```java
public synchronized void put(int item)
      throws InterruptedException
{
  if (numberInBuffer == size) {
    wait();
  };
  last = (last + 1) % size ;
  numberInBuffer++;
  buffer[last] = item;
  notify();
};
```

```java
public synchronized int get()
      throws InterruptedException
{
    if (numberInBuffer == 0) {
      wait();
    };
    first = (first + 1) % size ;
    numberInBuffer--;
    notify();
    return buffer[first];
};
}
```

Mutually exclusive waiting

- There are no explicit condition variables
- If more than one condition exists and they are not mutually exclusive:
  - Awoken thread should evaluate the condition on which it is waiting

The Readers-Writers Problem revisited

- Standard solution in monitors is to have two condition variables: **OkToRead** and **OkToWrite**
- This cannot be directly expressed using a single class

```
public class ReadersWriters   // first solution
{

  private int readers = 0;
  private int waitingWriters = 0;
  private boolean writing = false;
```

Readers-Writers Problem II

```
  public synchronized void StartWrite()
           throws InterruptedException
  {
    while(readers > 0 || writing)
    {
      waitingWriters++;
      wait();
      waitingWriters--;
    }
    writing = true;
  }


public synchronized void StopWrite()
  {
    writing = false;
    notifyAll();
  }
```

Loop to re-test the condition

Wakeup everyone

Readers-Writers Problem III

```
  public synchronized void StartRead()
         throws InterruptedException
  {
    while(writing || waitingWriters > 0) wait();
    readers++;
  }

  public synchronized void StopRead()
  {
    readers--;
    if(readers == 0) notifyAll();
  }
}
```

Arguably, this is inefficient as all threads are woken

Implementing Condition Variables

- ***Approach:*** use another class and block synchronization
  1) Get lock on ***condition variable*** on which you might want to sleep or notify
  2) Then get ***monitor lock***

```
public class ConditionVariable {
  public boolean wantToSleep = false;
}

public class ReadersWriters    // Alternative solution
{
  private int readers = 0;
  private int waitingReaders = 0;
  private int waitingWriters = 0;
  private boolean writing = false;

  ConditionVariable OkToRead = new ConditionVariable();
  ConditionVariable OkToWrite = new ConditionVariable();
```

Implementing Condition Variables II

```
public void StartWrite() throws InterruptedException
{
  synchronized(OkToWrite) // get condition variable lock
  {
    synchronized(this)    // get monitor lock
    {
      if(writing | readers > 0) {
        waitingWriters++;
        OkToWrite.wantToSleep = true;
      } else  {
        writing = true;
        OkToWrite.wantToSleep = false;
      }
    } //give up monitor lock
    if(OkToWrite.wantToSleep) OkToWrite.wait();
  }
}
```

Note order of synchronized statements

Implementing Condition Variables III

```
public void StopWrite()
{
  synchronized(OkToRead)
  {
    synchronized(OkToWrite)
    {
      synchronized(this)
      {
        if(waitingWriters > 0) {
          waitingWriters--;
          OkToWrite.notify();     // wakeup one writer
        } else {
          writing = false;
          OkToRead.notifyAll();   // wakeup all readers
          readers = waitingReaders;
          waitingReaders = 0;
        }
      }
    }
  }
}
```

Important for all methods to use the same order – otherwise deadlock will occur

Implementing Condition Variables IV

```
public void StartRead()
       throws InterruptedException
{
  synchronized(OkToRead) {
    synchronized(this)
    {
      if(writing | waitingWriters > 0) {
        waitingReaders++;
        OkToRead.wantToSleep = true;
      } else {
        readers++;
        OkToRead.wantToSleep = false;
      }
    }
    if(OkToRead.wantToSleep) OkToRead.wait();
  }
}
```

Implementing Condition Variables V

```
  public void StopRead()
  {
    synchronized(OkToWrite)
    {
      synchronized(this)
      {
        readers--;
        if(readers == 0 & waitingWriters > 0)
        {
          waitingWriters--;
          OkToWrite.notify();
        }
      }
    }
  }
}
```

Where are we?

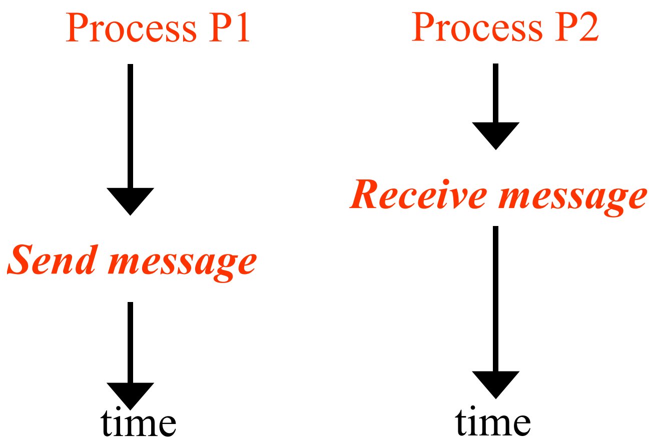
1) Language support for synchronization

2) ***Communication and synchronisation based on message passing***

   - ***Synchronisation models***
   - ***Process naming***
   - ***Message structures***

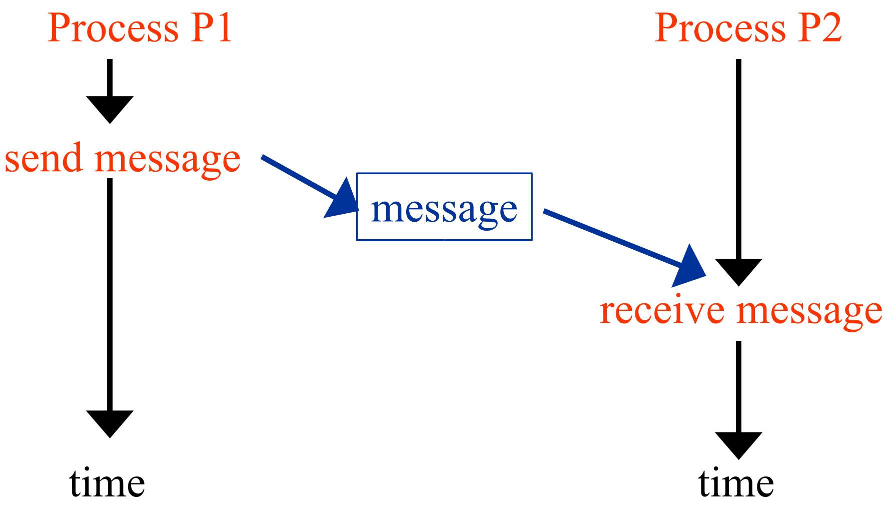
Message-Based Comm. and Synchronisation

- Use of a single construct for both ***synchronisation*** and ***communication***
- Three issues:
  - the model of ***synchronisation***
  - the method of ***process naming***
  - the ***message structure***



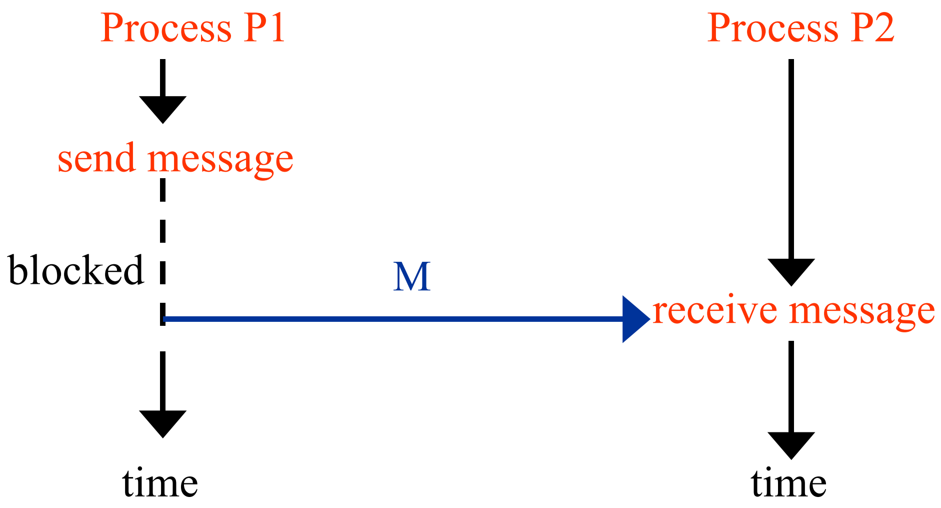
Process Synchronisation

- Variations in the process synchronisation model arise from the semantics of the ***send*** operation
- ***Asynchronous*** (or no-wait) (e.g. ***POSIX***)
  - Requires buffer space. What happens if the buffer is full?



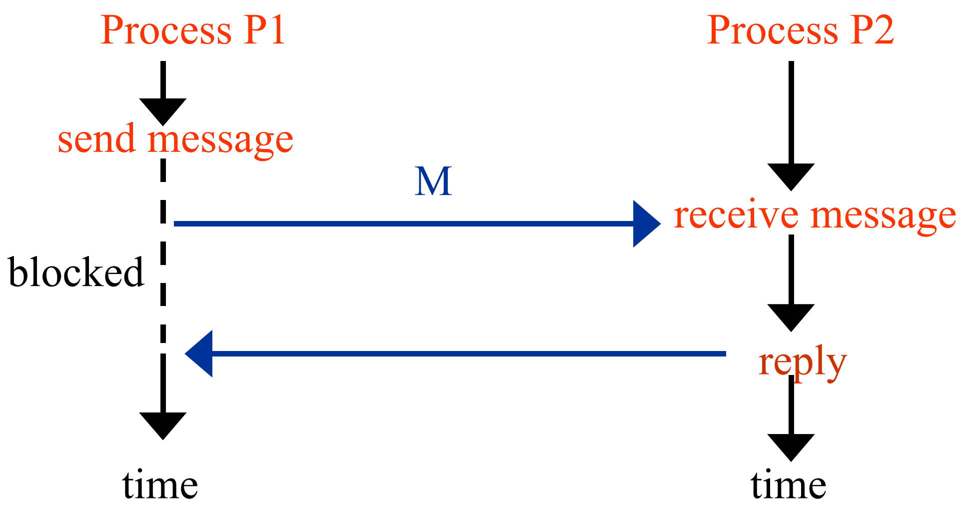
Process Synchronisation

- ***Synchronous*** (e.g. CSP, occam2)
  - No buffer space required
  - Known as a ***rendezvous***



Process Synchronisation

- ***Remote invocation*** (e.g. Ada)
  - Known as an ***extended rendezvous***



Asynchronous and Synchronous Sends

- Asynchronous communication can implement synchronous communication:

P1	P2
asyn_send (M)	wait (M)
wait (ack)	asyn_send (ack)

- Two synchronous communications can be used to construct a remote invocation:

P1	P2
syn_send (message)	wait (message)
wait (reply)	...
	construct reply
...	syn_send (reply)

Disadvantages of Asynchronous Send

- Potentially ***infinite buffers*** are needed to store unread messages
- Most sends are programmed to expect an acknowledgement
- More communications are needed with the asynchronous model, hence programs are ***more complex***
- It is more difficult to prove the ***correctness*** of the complete system
- Where asynchronous communication is desired with synchronised message passing then ***buffer processes*** can be constructed – but this can be costly

Process Naming

- The issues:
  - direction versus indirection
  - symmetry
- ***Direct naming:*** the sender explicitly names the receiver
  - ***Structure***: send <message> to <process-name>
  - ***Advantage***: Simplicity

Process Naming

- ***Indirect naming:*** the sender names an intermediate entity (e.g. a ***channel***, ***mailbox***, ***link*** or ***pipe***)
  - ***Structure***: send <message> to <mailbox>
  - Message passing can still be synchronous
  - ***Advantage***: Aids the decomposition of the software
  - A mailbox can be seen as an interface between program parts
- Possible ***structures*** of the intermediary:
  - Many-to-one
  - Many-to-manyh
  - One-to-one
  - One-to-many

Process Naming

- A naming scheme is ***symmetric*** if both sender and receiver name each other (directly or indirectly)

  send <message> to <process-name>
  wait <message> from <process-name>

  send <message> to <mailbox>
  wait <message> from <mailbox>

- It is ***asymmetric*** if the receiver names no specific source but accepts messages from any process (or mailbox)

  wait <message>

  - Fits the ***client-server*** paradigm

Message Structure

- Ideally, a language allows ***any data object*** of any defined type (predefined or user) to be transmitted in a message
  - This is not trivial:
    - Data may be represented differently at sender and receiver
    - How to deal with pointers?
  - Early languages restricted the data types to be transmitted (e.g., ***occam-1***)
- Need to convert to a standard format for transmission across a network in a heterogeneous environment
- OS allow only ***arrays of bytes*** to be sent

The occam2 Model

- Occam2 supports ***indirect symmetric synchronous message passing***
- Occam2 processes are not named – therefore use ***named channels***
- Each channel restricted to single writers and single readers

```
ch ! X   -- Write value of expression X
         -- onto channel ch

ch ? Y   -- Read from channel ch
         -- into variable y

         -- This can be viewed as
         -- distributed assignment Y := X
```

The Ada Model

- Ada supports ***direct asymmetric remote invocation***
- Based on a ***client/server*** model of interaction
- The server declares a set of services that it is prepared to offer other tasks (its clients)
- It does this by declaring one or more ***public entries*** in its task specification
- Each entry identifies the ***name*** of the service, the ***parameters*** that are required with the request, and the ***results*** that will be returned
- This has many similarities with a procedure call
- *Will not discuss this further in class – is covered in additional lecture slides appended here*

- ***Ada*** has remote invocation with direct asymmetric naming
- Communication in Ada requires one task to define an entry and then, within its body, accept any incoming call. A ***rendezvous*** occurs when one task calls an entry in another
- ***Selective waiting*** allows a process to wait for more than one message to arrive.
- Ada's select statement has two extra facilities: an ***else*** part and a ***terminate*** alternative

Summary

- The semantics of message-based communication are defined by three issues:
  - the model of ***synchronisation***
  - the method of ***process naming***
  - the ***message structure***
- Variations in the process synchronisation model arise from the ***semantics of the send operation.***
  - ***asynchronous***, ***synchronous*** or ***remote invocation***
  - Remote invocation can be made to appear syntactically similar to a procedure call
- Process naming involves two distinct issues
  - ***direct*** or ***indirect***
  - ***symmetry***

Summary

- ***POSIX mutexes*** and ***condition variables*** give monitors with a procedural interface
- ***Ada's protected objects*** give structured mutual exclusion and high-level synchronization via barriers
- ***Java's synchronized methods*** provide monitors within an object-oriented framework

Problem Set 10 – Due 3 July 2002

1) Implement a binary semaphore using the counting semaphore of ***POSIX*** and illustrate its operation. **(2 pts)**
2) Using binary semaphores from Problem 1, implement a counting semaphore and illustrate its operation. **(2 pts)**
3) Show how the reader/writers problem can be implemented in ***Java*** where priority is given to readers and where writers are guaranteed to be serviced in FIFO order. **(2 pts)**
4) Modify the robot you built last week such that for each of the exceptional events identified (a crossed line is thicker than 10cm, *or* an obstacle is encountered, *or* a button is pressed, *or* the time-out occurs), it performs a particular driving maneuver specific to that event (e.g., turn left 90 degrees if a button is pressed). Each driving maneuver constitutes a critical region that must not be interrupted by another driving maneuver. Implement this using semaphores. **(3 pts)**

Announcement

Excursion to Philips Medical Systems GmbH (Hamburg)

July 11, 2002, departing CAU at 11:00

If interested, please contact rvh@informatik.uni-kiel.de

The bus has 9 spaces – *first come, first serve*

For current information, visit http://www.informatik.uni-kiel.de/inf/von-Hanxleden/Exkursionen/ss02-philips.html

Appendix: The Ada Model

- Ada supports ***direct asymmetric remote invocation***
- Based on a ***client/server*** model of interaction
- The server declares a set of services that it is prepared to offer other tasks (its clients)
- It does this by declaring one or more ***public entries*** in its task specification
- Each entry identifies the ***name*** of the service, the ***parameters*** that are required with the request, and the ***results*** that will be returned
- This has many similarities with a procedure call
- *Will not discuss this further in class – is covered in additional lecture slides appended here*

Entries

- ***Format:***

```
entry_declaration ::=
  entry defining_identifier[(discrete_subtype_definition)]
    parameter_profile;
```

- ***Examples:***

```
entry Syn;
entry Send(V : Value_Type);
entry Get(V : out Value_Type);
entry Update(V : in out Value_Type);
entry Mixed(A : Integer; B : out Float);
entry Family(Boolean)(V : Value_Type);
```

- Note that data can be transferred in either direction – hence this is usually not referred to as „message passing“, but instead as ***extended rendezvous***

Example

```
task type Telephone_Operator is
  entry Directory_Enquiry(
    Person : in Name;
    Addr : Address;
    Num : out Number);
  -- other services possible
end Telephone_Operator;

An_Op : Telephone_Operator;

-- client task executes
An_Op.Directory_Enquiry ("Stuart_Jones",
                         "11 Main, Street, York"
                         Stuarts_Number);
```

Accept Statement

```
accept_statement ::=
  accept entry_direct_name[(entry_index)]
    parameter_profile [do
      handled_sequence_of_statements
  end [entry_identifier]];
```

```
accept Family(True)(V : Value_Type) do
  -- sequence of statements
exception
  -- handlers
end Family;
```

Server Task Example

```
task body Telephone_Operator is
begin
  ...
  loop

    --prepare to accept next call
    accept Directory_Enquiry (...) do
      -- look up telephone number

    exception
      when Illegal_Number =>
        -- propagate error to client

    end Directory_Enquiry;
    -- undertake housekeeping
  end loop;
  ...
end Telephone_Operator;
```

Client Task Example

```
task type Subscriber;
task body Subscriber is
begin
  ...
  loop
    ...
    An_Op.Directory_Enquiry(...);
    ...
  end loop;
  ...
end Subscriber;
```

Protocol

```
T.E(A,B)
```

```
task T is ...

accept E(X : int; Y: out int) do
   -- use X
   -- undertake computation
   -- produce Y
   -- complete computation
end E;
```


- Both tasks must be prepared to enter into the communication
- If one is ready and the other is not, then the ready one ***waits*** for the other
- Once both are ready, the client's parameters are passed to the server
- The server then executes the code inside the accept statement
- At the end of the accept, the results are returned to the client
- Both tasks are then free to continue independently

Bus Driver Example

```
task type Bus_Driver (Num : Natural) is
  entry Get_Ticket (R: in Request, M: in Money;
                    G : out Ticket) ;
  -- money given with request, no change given!
end Bus_Driver;


task body Bus_Driver is
begin
  loop
    accept Get_Ticket (R: Request,
                       M: Money; G : out Ticket) do
      -- take money
      G := Next_Ticket(R);
    end Get_Ticket;
  end loop;
end Bus_Driver;
```

```
type Bus_T (N : Natural) is
  record
    ....
    Driver : Bus_Driver(N);
  end record;

Number31 : Bus_T(31);
Number60 : Bus_T(60);
Number70 : Bus_T(70);
```

Shop Keeper Example

```
task Shopkeeper is
  entry Serve(X : Request; A: out Goods);
  entry Get_Money(M : Money; Change : out Money);
end Shopkeeper;

task body Shopkeeper is
begin
  loop
    accept Serve(X : Request; A: out Goods) do
      A := Get_Goods;
    end Serve;

    accept Get_Money(M : Money; Change : out Money) do
      -- take money return change
    end Get_Money;
  end loop;
end Shopkeeper;
```

What is wrong with this algorithm?

Customer

```
task Customer;
task body Customer is
begin
  -- go to shop
  Shopkeeper.Serve(Weekly_Shoping, Trolley);
  -- leave shop in a hurry!
end Customer;
```

Rider

```
task type Rider;
task body Rider is
begin
  ...
  -- go to bus stop and wait for bus
  while Bus /= Number31 loop
    -- moan about bus service
  end loop;

  Bus.Bus_Driver.Get_Ticket(Heslington, Fiftyp, Ticket);
  -- get in line
  -- board bus, notice three more number 31 buses
  ...
end Rider;
```

Other Facilities

- `'Count` gives number of tasks queued on an entry
- ***Entry families*** allow the programmer to declare, in effect, a single dimension ***array of entries***
- ***Nested accept statements*** allow more than two tasks to communicate and synchronise
- A task executing inside an accept statement can also execute an entry call
- ***Exceptions*** not handled in a rendezvous are propagated to ***both*** the caller and the called tasks
- An accept statement can have exception handlers

Families

```
task Multiplexer is
  entry Channel(1..3)(X : Data);
end Multiplexer;

task body Multiplexer is
begin
  loop
    for I in 1..3 loop
      accept Channel(I)(X : Data) do
      -- consume input data on channel I
      end Channel;
    end loop;
  end loop;
end Multiplexer;
```

A family declaration

Grocery Store

```
type Counter is (Meat, Cheese, Wine);
task Edeka_Server is
  entry Serve(Counter)(Request: . . .);
end Edeka_Server;

task body Edeka_Server is
begin
  loop
    accept Serve(Meat)(. . .) do . . . end Serve;
    accept Serve(Cheese)(. . .) do . . . end Serve;
    accept Serve(Wine)(. . .) do . . . end Serve;
  end loop
end Edeka_Server;
```

- What happens if all queues are full?
- What happens if the Meat queue is empty?

Nested Accepts

```
task body Controller is
begin
  loop
    accept Doio (I : out Integer) do
      accept Start;
      accept Completed (K : Integer) do
        I := K;
      end Completed;
    end Doio;
  end loop;
end Controller;
```

Shopkeeper Example

```
task Shopkeeper is
  entry Serve_Groceries(. . .);
  entry Serve_Tobacco( . . .);
  entry Serve_Alcohol(. . .);
end Shopkeeper;

task body Shopkeeper is
begin
  . . .
  accept Serve_Groceries (. . .) do
    -- no change for a €10 note
    accept Serve_ Alcohol(. . .) do
      -- serve another Customer,
      -- get more change
    end Serve_ Alcohol
  end Serve_Groceries
  . . .
end Shopkeeper;
```

Can not have

```
accept Serve_Groceries (. . .) do
  accept Serve_Groceries(. . .) do
    . . .
  end Serve_Groceries
end Serve_Groceries
```

Entry Call within Accept Statement

```
task Car_Spares_Server is
   entry Serve_Car_Part(Number: Part_ID; . . .);
end Car_Spares_Server ;

task body Car_Spares_Server is
begin
  . . .
  accept Serve_Car_Part(Number: Part_ID; . . .) do
    -- part not is stock
    Dealer.Phone_Order(. . .);
  end Serve_Car_Part;
  . . .
end Car_Spares_Server;
```

Exceptions

```
accept Get(R : out Rec; Valid_Read : out Boolean) do
  loop
    begin
      Put("VALUE OF I?"); Get(R.I);
      Put("VALUE OF F?"); Get(R.F);
      Put("VALUE OF S?"); Get(R.S);
      Valid_Read := True;
      return;
    exception
      when Ada.Text_IO.Data_Error =>
        Put("INVALID INPUT: START AGAIN");
    end;
  end loop;
exception
  when Ada.Text_IO.Mode_Error =>
    Valid_Read := False;
end Get;
```

return from accept

exception raised

could be handled here

or here

If not handled anywhere exception raised in calling task and the 'accept' task

Private Entries

- ***Public entries*** are visible to all tasks which have visibility to the owning task's declaration
- ***Private entries*** are only visible to the owning task
  - if the task has several tasks declared internally; these tasks have access to the private entry
  - if the entry is to be used internally by the task for requeuing purposes
  - if the entry is an interrupt entry, and the programmer does not wish any software task to call this entry

Private Entries II

```
task type Telephone_Operator is
  entry Report_Fault(N : Number);

private
  entry Allocate_Repair_Worker(N : out Number);
end Telephone_Operator;

task body Telephone_Operator is
  Failed : Number;
  task type Repair_Worker;
  Work_Force:array (1.. Num_Workers) of Repair_Worker;

  task body Repair_Worker is
    Job : Number:
  begin
    ...
    Telephone_Operator.Allocate_Repair_Worker(Job);
    ...
  end Repair_Worker;
```

private entry

internal task

Private Entries III

```
begin
  loop
    accept Report_Fault(N : Number) do
      Failed := N;
    end Report_Fault;

    -- log faulty line
    if New_Fault(Failed) then -- new fault
      accept Allocate_Repair_Worker(N : out Number) do
        N := Failed;
      end Allocate_Repair_Worker;
    end if;
  end loop;
end Telephone_Operator;
```

Selective Accept

The selective accept allows the server to:

- wait for ***several rendezvous*** at any one time
- ***time-out*** if no rendezvous is forthcoming within a specified time
- ***withdraw*** its offer to communicate if no rendezvous is available immediately
- ***terminate*** if no clients can possibly call its entries

The select statement comes in four forms:

```
Select_statement ::=
            selective_accept |
            conditional_entry_call |
            timed_entry_call |
            asynchronous_select
```


- So far, the receiver of a message must wait until the specified process, or mailbox, delivers the communication
- A receiver process may actually wish to wait for ***any one*** of a number of processes to call it
- Server processes receive request messages from a number of clients; the order in which the clients call being ***unknown*** to the servers
- To facilitate this common program structure, receiver processes are allowed to wait selectively for a number of possible messages
- Based on Dijkstra's guarded commands (1975)

Syntax Definition

```
Selective_accept ::=
  select
    [guard]
     selective_accept_alternative
{ or
    [guard]
    selective_accept_alternative
[ else
    sequence_of_statements ]
  end select;

guard ::= when <condition> =>
```

```
selective_accept_alternative ::=
  accept_alternative |
  delay_alternative  |
  terminate_alternative

accept_alternative ::=
   accept_statement [ sequence_of_statements ]

delay_alternative ::=
   delay_statement [ sequence_of_statements ]

terminate_alternative ::=
   terminate;
```

Overview Example

```
task Server is
  entry S1(...);
  entry S2(...);
end Server;

task body Server is
  ...
begin
  loop
    select
      accept S1(...) do
        -- code for this service
      end S1;
    or
      accept S2(...) do
        -- code for this service
      end S2;
    end select;
  end loop;
end Server;
```

Simple select with two possible actions

Example

```
task type Telephone_Operator is
  entry Directory_Enquiry (P : Name; A : Address;
                           N : out Number);
  entry Directory_Enquiry (P : Name; PC : Postal_Code;
                           N : out Number);
  entry Report_Fault(N : Number);

private
  entry Allocate_Repair_Worker (N : out Number);
end Telephone_Operator;

task body Telephone_Operator is
  Failed : Number;
  task type Repair_Worker;
  Work_Force : array(1.. Num_Workers) of
               Repair_Worker;

  task body Repair_Worker is separate;
```

Example II

```
begin
  loop
    select
      accept Directory_Enquiry( ... ; A: Address...) do
        -- look up number based on address
      end Directory_Enquiry;
    or
      accept Directory_Enquiry( ... ;
                                   PC: Postal_Code...) do
        -- look up number based on ZIP
      end Directory_Enquiry;
    or
```

Example III

```
    or
      accept Report_Fault(N : Number) do
        ...
      end Report_Fault;

      if New_Fault(Failed) then
        accept Allocate_Repair_Worker (N : out
               Number) do
          N := Failed;
        end Allocate_Repair_Worker;
        -- update record of failed unallocated numbers
      end if;
    end select;
  end loop;
end Telephone_Operator;
```

Note

- If no rendezvous are available, the select statement ***waits*** for one to become available
- If one is available, it is chosen immediately
- If more than one is available, the one chosen is implementation dependent (RT Annex allows order to be defined)
- More than one task can be queued on the same entry; default queuing policy is ***FIFO*** (RT Annex allows priority order to be defined)

Edeka revisited

```
type Counter is (Meat, Cheese, Wine);
task Edeka_Server is
  entry Serve(Counter)(Request: . . .);
end Edeka_Server;

task body Edeka_Server is
begin
  loop
    select
      accept Serve(Meat)(. . .) do . . . end Serve;
    or
      accept Serve(Cheese)(. . .) do . . . end Serve;
    or
      accept Serve(Wine)(. . .) do . . . end Serve;
    end select
  end loop
end Edeka_Server;
```

- What happens if all queues are full?
- What happens if the Meat queue is empty?

Guarded Alternatives

- Each select accept alternative can have an associated ***guard***
  - The guard is a boolean expression which is evaluated when the select statement is executed
  - If the guard evaluates to true, the alternative is eligible for selection
  - If it is false, the alternative is not eligible for selection during this execution of the select statement (even if client tasks are waiting on the associated entry)

Example of Guard

```
task body Telephone_Operator is
begin
  ...
  select
    accept Directory_Enquiry (...) do ... end;
  or
    accept Directory_Enquiry (...) do ... end;
  or
    when Workers_Available =>
      accept Report_Fault (...) do ... end;
  end select;
end Telephone_Operator;
```

Delay Alternative

- The delay alternative of the select statement allows the server to ***time-out*** if an entry call is not received within a certain period
- The timeout is expressed using a delay statement, and therefore ***can be relative or absolute***
- If the relative time is negative, or the absolute time has passed, the delay alternative becomes equivalent to the ***else*** alternative
- More than one delay is allowed

Example: Periodic Execution

- Consider a task which reads a sensors every 10 seconds, however, it may be required to change its periods during certain modes of operation

```
task Sensor_Monitor is
  entry New_Period(P : Duration);
end Sensor_Monitor;
```

Periodic Execution II

```
task body Sensor_Monitor is
  Current_Period : Duration := 10.0;
  Next_Cycle : Time := Clock + Current_Period;
begin
  loop
    -- read sensor value etc.
    select
      accept New_Period(P : Duration) do
        Current_Period := P;
      end New_Period;
      Next_Cycle := Clock + Current_Period;
    or
      delay until Next_Cycle;
      Next_Cycle := Next_Cycle + Current_Period;
    end select;
  end loop;
end Sensor_Monitor;
```

Delay Alternative: Error Detection

```
task body Watchdog is
  Client_Failed : Boolean := False;
begin
  loop
    select
      accept All_Is_Well;
    or
      delay 10.0;
      -- signal alarm
      Client_Failed := True;
    end select;

    exit when Client_Failed;
  end loop;
end Watchdog;
```

The Else Part

- The else alternative is executed when no other alternative is ***immediately*** executable

```
task body Sensor_Monitor is
  Current_Period : Duration := 10.0;
  Next_Cycle : Time := Clock + Current_Period;
begin
  loop
    -- read sensor value etc.
    select
      accept New_Period(P : Duration) do
        Current_Period := P;
      end New_Period;
    else -- cannot be guarded
      null;
    end select;

    Next_Cycle := Clock + Current_Period;
    delay until Next_Cycle;
  end loop;
end Sensor_Monitor;
```

else part

The Delay and the Else Part

- Cannot mix else part and delay in the same select statement.
- The following are equivalent

```
select
  accept A;
or
  accept B;
else
  C;
end select;
```

```
select
  accept A;
or
  accept B;
or
  delay 0.0;
  C;
end select;
```

More on Delay

What is the difference?

```
select
  accept A;
or
  delay 10.0;
end select;
```

```
select
  accept A;
else
  delay 10.0;
end select;
```

```
select
  accept A;
or
  delay 5.0;
  delay 5.0;
end select;
```

The Terminate Alternative

- In general a server task ***only*** needs to exist when there are clients to serve
- The very nature of the client server model is that the server does ***not*** know the identity of its clients
- The ***terminate alternative*** in the select statement allows a server to indicate its willingness to terminate if there are no clients that could possibly request its service
- The server terminates when a master of the server is completed and all its dependants are either already terminated or are blocked at a select with an open terminate alternative

Program Error

- If all the accept alternatives have guards then there is the possibility in certain circumstances that ***all the guards will be closed***
- If the select statement does not contain an else clause then it becomes impossible for the statement to be executed
- The exception **`Program_Error`** is raised at the point of the select statement if no alternatives are open

The Selective Accept : Summary

- A selective accept must contain ***at least one accept alternative*** (each possibly guarded)
- A selective accept may contain one and only one of the following :
  - a ***terminate alternative*** (possibly guarded), or
  - one or more ***delay alternatives*** (each possibly guarded), or
  - an ***else*** part

The Selective Accept : Summary II

- A select alternative is ***open*** if it does not contain a guard or if the boolean condition associated with the guard evaluates to true; otherwise the alternative is ***closed***
- On execution, all of the following are evaluated:
  - all guards
  - open delay expressions
  - open entry family expressions
- A choice is made from the open alternatives

Non-determinism and Selective Waiting

- Concurrent languages make few assumptions about the ***execution order*** of processes
- A scheduler is assumed to schedule processes ***non-deterministically***
- Consider a process P that will execute a selective wait construct upon which processes S and T could call

Possible Execution Orders

1) ***P runs first***; it is blocked on the select. ***S (or T) then runs*** and rendezvous with P
2) ***S (or T) runs***, blocks on the call to P; ***P runs*** and executes the select; a rendezvous takes place with S (or T)
3) ***S (or T) runs first*** and blocks on the call to P; ***T (or S) now runs*** and is also blocked on P. Finally P runs and executes the select on which T and S are waiting

Comparison

- The three possible interleavings lead to P having none, one or two calls outstanding on the selective wait
- If P, S and T can execute in any order then, in latter case, P should be able to choose to rendezvous with S or T — ***it will not affect the programs correctness***

Non-determinism and Selective Waiting

- A similar argument applies to any queue that a synchronisation primitive defines
- Non-deterministic scheduling implies all queues should release processes in a non-deterministic order
- Semaphore queues are often defined in this way; entry queues and monitor queues are specified to be FIFO
- The rationale here is that FIFO queues prohibit starvation but if the scheduler is non-deterministic then starvation can occur anyway!

Timed Entry Calls

- A ***timed entry call*** issues an entry call which is cancelled if the call is not accepted within the specified period (relative or absolute)
- ***Note:*** only one delay alternative and one entry call can be specified

Timed Entry Calls II

```
task body Subscriber is
  Stuarts_Number : Number;
begin
  loop
    ...
    select
      An_Op.Directory_Enquiry("Stuart Jones",
         "10 Main Street, York", Stuarts_Number);
      -- log the cost of a directory enquiry call
    or
      delay 10.0;
      -- phone up Stuart's parents and ask them;
      -- log the cost of a long distance call
    end select;
    ...
   end loop;
end Subscriber;
```

Timed Entry Calls III

```
task body Telephone_Operator is
  ...
begin
  loop
    -- prepare to accept next request
    select
      accept Directory_Enquiry(Person : Name;
         Addr   : Address; Num  : out Number) do
         delay 3600.0; -- take a lunch break
      end Directory_Enquiry; or
      ...
    end select;
    ...
   end loop;
end Telephone_Operator;
```

Time-out is on the start of the rendezvous not the finish

Conditional Entry Call

- The conditional entry call allows the client to ***withdraw*** the offer to communicate if the server task is not prepared to accept the call immediately
- It has the same meaning as a timed entry call where the expiry time is immediate

```
select
  Security_Op.Turn_Lights_On;
else
  null; -- assume they are on already
end select;
```

Conditional Entry Call II

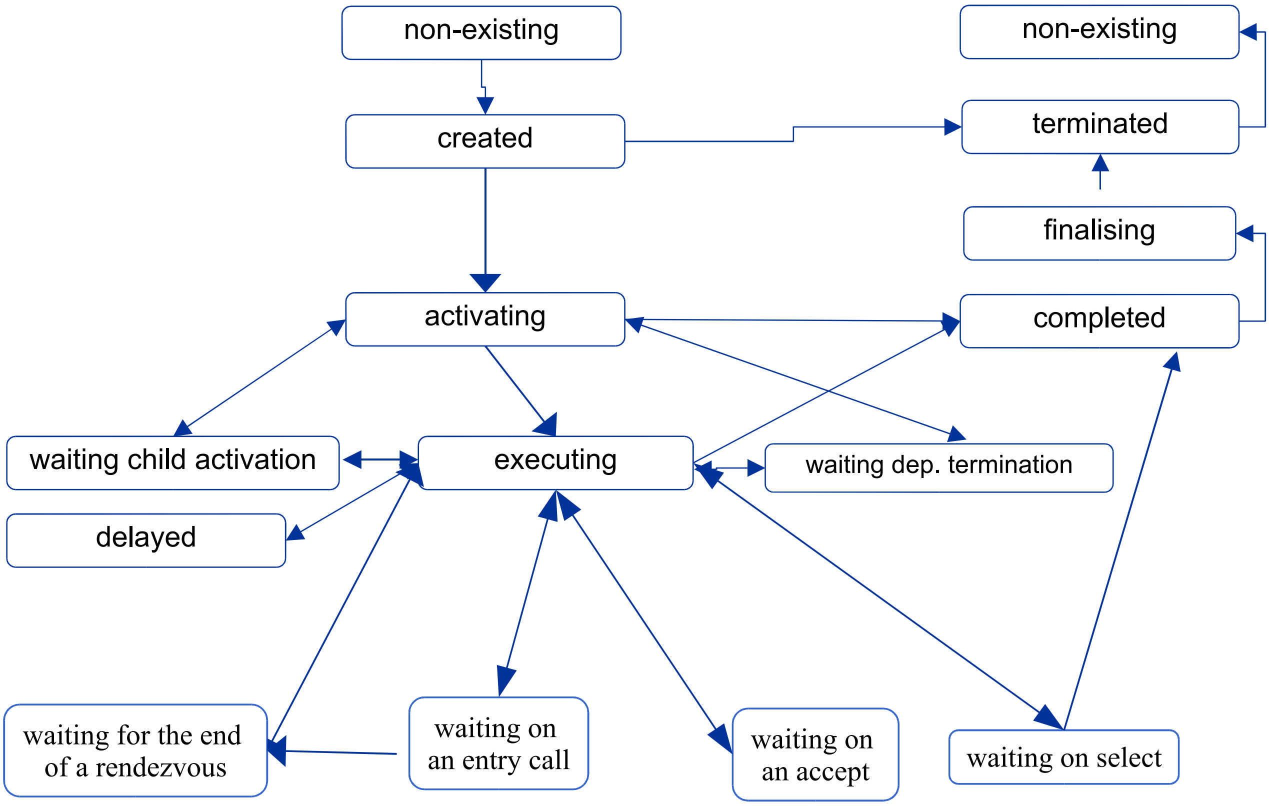
- A conditional entry call should only be used when the task can genuinely do other productive work, if the call is not accepted
- Care should be taken not to program polling, or busy-wait, solutions unless they are explicitly required
- Note, the conditional entry call uses an **else,** the timed entry call an **or**

Conditional Entry Call III

- They cannot be mixed, nor can two entry call statements be included
- A client task can not therefore wait for more than one entry call to be serviced

- The asynchronous select statement allows some of these restrictions to be overcome

Task States



non-existing
created
activating
waiting child activation
delayed
executing
waiting dep. termination
non-existing
terminated
finalising
completed
waiting for the end of a rendezvous
waiting on an entry call
waiting on an accept
waiting on select

Summary Ada Message Passing

- ***Ada*** has remote invocation with direct asymmetric naming
- Communication in Ada requires one task to define an entry and then, within its body, accept any incoming call. A ***rendezvous*** occurs when one task calls an entry in another
- ***Selective waiting*** allows a process to wait for more than one message to arrive.
- Ada's select statement has two extra facilities: an ***else*** part and a ***terminate*** alternative