Shared Variable-Based Synchronization and Communication

- To understand the requirements for communication and synchronisation based on shared variables

- To briefly review semaphores, monitors and conditional critical regions

- To understand Ada 95 protected objects, POSIX mutexes and Java synchronized methods

Prerequisites

- **Understanding the issues of busy-waiting and semaphores from an Operating System Course.**

- **However:**
  - Course book give full details on busy-waiting, semaphores, conditional critical regions, monitors etc.

Synchronisation and Communication

- The correct behaviour of a concurrent program depends on synchronisation and communication between its processes

- Synchronisation: the satisfaction of constraints on the interleaving of the actions of processes (e.g. an action by one process only occurring after an action by another)

- Communication: the passing of information from one process to another
  - Concepts are linked since communication requires synchronisation, and synchronisation can be considered as contentless communication.
  - Data communication is usually based upon either shared variables or message passing.

Shared Variable Communication

- Examples: busy waiting, semaphores and monitors

- Unrestricted use of shared variables is unreliable and unsafe due to multiple update problems

- Consider two processes updating a shared variable, X, with the assignment: X:= X+1

  - load the value of X into some register

  - increment the value in the register by 1 and

  - store the value in the register back to X

- As the three operations are not indivisible, two processes simultaneously updating the variable could follow an interleaving that would produce an incorrect result

Shared Resource Communication

```
type Coordinates is

  record

    X : Integer;

    Y : Integer;

  end record;

Shared_Cordinate: Coordinates;
```

```
task body Helicopter is

  Next: Coordinates;

begin

  loop

   Compute_New_Cordinates(Next);

    Shared_Cordinates := Next;

  end loop

end;
```

```
task body Police_Car is

begin

  loop

    Plot(Shared_Cordinates);

  end loop;

end;
```

X = 5

Y = 4

5

4

1,1
2,2
3,3
4,4
5,5
6,6
...

Villain's Escape

Route

(seen by helicopter)

Villain Escapes!

1,1
2.2
3,3
4,4
4,5

Police Car's

Pursuit Route

Avoiding Interference

- The parts of a process that access shared variables must be executed indivisibly with respect to each other

- These parts are called critical sections

- The required protection is called mutual exclusion

Mutual Exclusion

- A sequence of statements that must appear to be executed indivisibly is called a critical section

- The synchronisation required to protect a critical section is known as mutual exclusion

- Atomicity is assumed to be present at the memory level. If one process is executing X:= 5, simultaneously with another executing X:= 6, the result will be either 5 or 6 (not some other value)

- If two processes are updating a structured object, this atomicity will only apply at the single word element level

Condition Synchronisation

- Condition synchronisation is needed when a process wishes to perform an operation that can only sensibly, or safely, be performed if another process has itself taken some action or is in some defined state

- E.g. a bounded buffer has 2 condition synchronisation:
  - the producer processes must not attempt to deposit data onto the buffer if the buffer is full
  - the consumer processes cannot be allowed to extract objects from the buffer if the buffer is empty
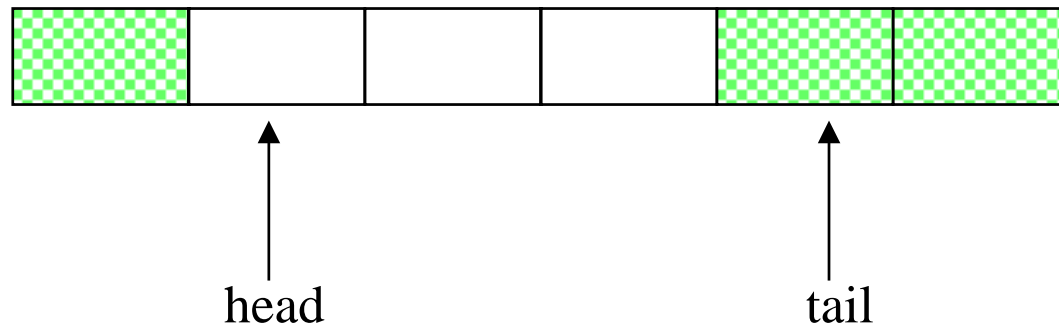
Is mutual exclusion necessary?

head   tail

Busy Waiting

- One way to implement synchronisation is to have processes set and check shared variables that are acting as flags

- This approach works well for condition synchronisation but no simple method for mutual exclusion exists

- Busy wait algorithms are in general inefficient; they involve processes using up processing cycles when they cannot perform useful work

- Even on a multiprocessor system they can give rise to excessive traffic on the memory bus or network (if distributed)

Semaphores

- A semaphore is a non-negative integer variable that apart from initialization can only be acted upon by two procedures P (or WAIT) and V (or SIGNAL)

- WAIT(S)   If the value of S > 0 then decrement its value by one; otherwise delay the process until S > 0 (and then decrement its value).

- SIGNAL(S)   Increment the value of S by one.

- WAIT and SIGNAL are atomic (indivisible). Two processes both executing WAIT operations on the same semaphore cannot interfere with each other and cannot fail during the execution of a semaphore operation

Condition synchronisation

```
var consyn : semaphore (* init 0 *)
```

```
process P1;
   (* waiting process *)
   statement X;
   wait (consyn)
   statement Y;
end P1;
```

```
process P2;
   (* signalling proc *)
   statement A;
   signal (consyn)
   statement B;
end  P2;
```

In what order will the statements execute?
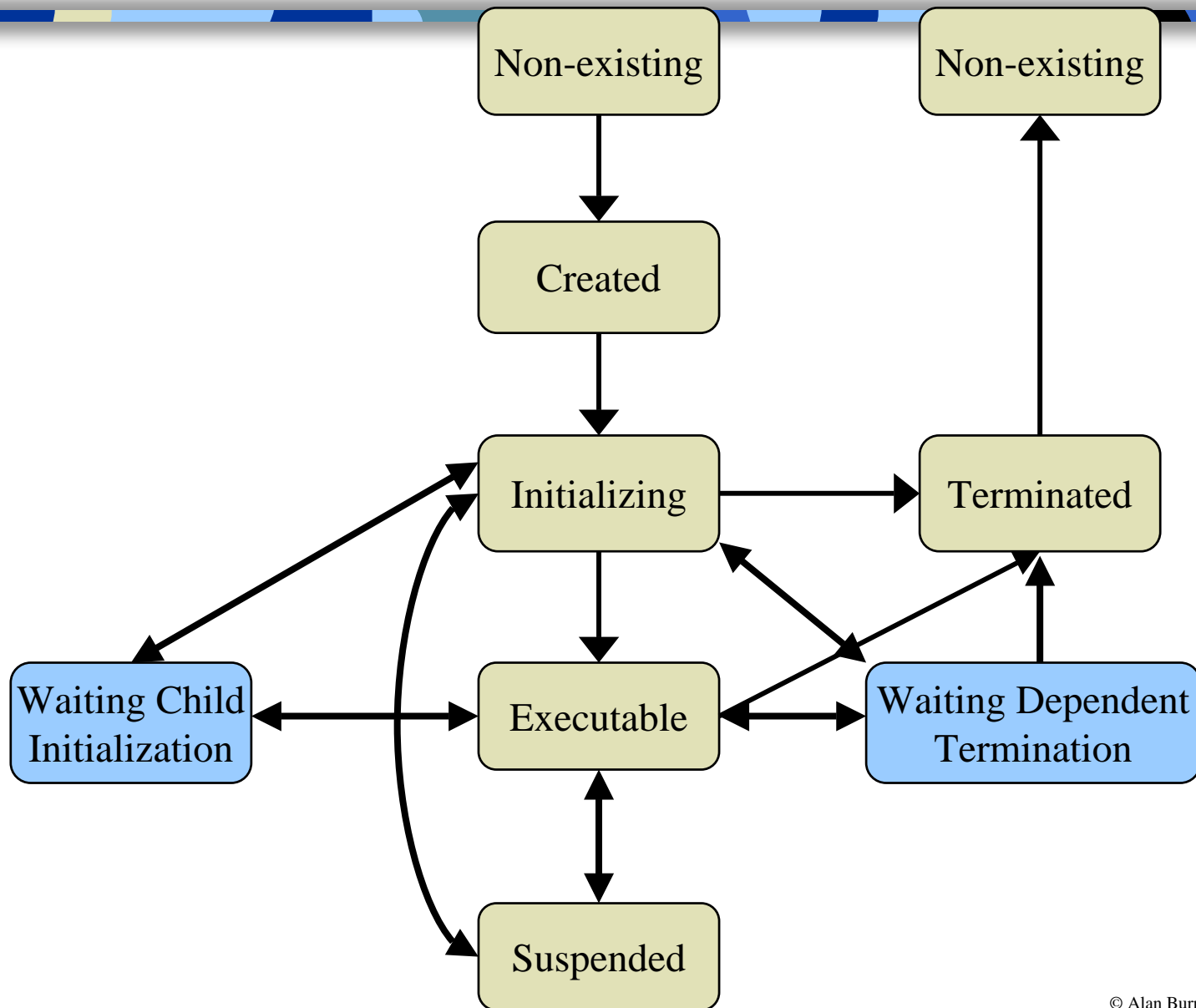
Mutual Exclusion

```
(* mutual exclusion *)
var mutex : semaphore; (* initially 1 *)
```

```
process P1;
   statement X
   wait (mutex);
      statement Y
   signal (mutex);
   statement Z
end P1;
```

```
process P2;
   statement A;
   wait (mutex);
      statement B;
   signal (mutex);
   statement C;
end P2;
```

In what order will the statements execute?

Process States

Non-existing

Non-existing

Created

Initializing

Terminated

Waiting Child
Initialization

Executable

Waiting Dependent
Termination

Suspended

Deadlock

- Two processes are deadlocked if each is holding a resource while waiting for a resource held by the other

```
type Sem is ...;
X : Sem := 1; Y : Sem := 1;
```

```
task A;
task body A is
begin

    ...

    Wait(X);
    Wait(Y);

    ...

end A;
```

```
task B;
task body B is
begin

    ...

    Wait(Y);
    Wait(X);

    ...

end B;
```

Livelock

- Two processes are livelocked if each is executing but neither is able to make progress.

```
type Flag is (Up, Down);
Flag1 : Flag := Up;
```

```
task A;
task body A is
begin
  ...
  while Flag1 = Up loop
    null;
  end loop;
  ...
end A;
```

```
task B;
task body B is
begin
  ...
  while Flag1 = Up loop
    null;
  end loop;
  ...
end A;
```

Binary and quantity semaphores

- A general semaphore is a non-negative integer; its value can rise to any supported positive number

- A binary semaphore only takes the value 0 and 1; the signalling of a semaphore which has the value 1 has no effect - the semaphore retains the value 1

- A general semaphore can be implemented by two binary semaphores and an integer. Try it!

- With a quantity semaphore the amount to be decremented by WAIT (and incremented by SIGNAL) is given as a parameter; e.g. WAIT (S, i)

Example semaphore programs in Ada

```
package Semaphore_Package is
  type Semaphore(Initial : Natural) is limited private;
  procedure Wait (S : Semaphore);
  procedure signal (S : Semaphore);
private
  type Semaphore ...
end Semaphore_Package;
```

- Ada does not directly support semaphores; the wait and signal procedures can, however, be constructed from the Ada synchronisation primitives
- The essence of abstract data types is that they can be used without knowledge of their implementation!

The Bounded Buffer

```ada
package Buffer is
  procedure Append (I : Integer);
  procedure Take (I : out Integer);
end Buffer;

package body Buffer is
  Size : constant Natural := 32;
  type Buffer_Range is mod Size;
  Buf : array (Buffer_Range) of Integer;
  Top, Base : Buffer_Range := 0;
  Mutex : Semaphore(1);
  Item_Available : Semaphore(0);
  Space_Available : Semaphore(Size);
  procedure Append (I : Integer) is separate;
  procedure Take (I : out Integer) is separate;
end Buffer;
```

The Bounded Buffer

```
procedure Append(I : Integer) is
begin
  Wait(Space_Available);
  Wait(Mutex;
    Buf(Top) := I;
    Top := Top+1
  Signal(Mutex;
  Signal(Item_Available);
end Append;
```

```
procedure Take(I : out Integer) is
begin
  Wait(Item_Available);
  Wait(Mutex);
    I := BUF(base);
    Base := Base+1;
  Signal(Mutex);
  Signal(Space_Available);
end Take;
```

Criticisms of semaphores

- Semaphore are an elegant low-level synchronisation primitive, however, their use is error-prone

- If a semaphore is omitted or misplaced, the entire program to collapse. Mutual exclusion may not be assured and deadlock may appear just when the software is dealing with a rare but critical event

- A more structured synchronisation primitive is required

- No high-level concurrent programming language relies entirely on semaphores; they are important historically but are arguably not adequate for the real-time domain

Conditional Critical Regions (CCR)

- A critical region is a section of code that is guaranteed to be executed in mutual exclusion

- Shared variables are grouped together into named regions and are tagged as being resources

- Processes are prohibited from entering a region in which another process is already active

- Condition synchronisation is provided by guards. When a process wishes to enter a critical region it evaluates the guard (under mutual exclusion); if the guard evaluates true it may enter, but if it is false the process is delayed

- As with semaphores, no access order can be assumed

The Bounded Buffer

```
program buffer_eg;
   type buffer_t is record
      slots       : array(1..N) of character;
      size        : integer range 0..N;
      head, tail : integer range 1..N;
   end record;
   buffer : buffer_t;
   resource buf : buffer;

   process producer is separate;
   process consumer is separate;
end.
```

The Bounded Buffer

```
process producer;
   loop
      region buf when buffer.size < N do
         -- place char in buffer etc
      end region
   end loop;
end producer

process consumer;
   loop
      region buf when buffer.size > 0 do
         -- take char from buffer etc
      end region
   end loop;
end consumer
```

Problem

- One problem with CCRs is that processes must re-evaluate their guards every time a CCR naming that resource is left. A suspended process must become executable again in order to test the guard; if it is still false it must return to the suspended state

- A version of CCRs has been implemented in Edison, a language intended for embedded applications, implemented on multiprocessor systems. Each processor only executes a single process so it may continually evaluate its guards if necessary

Monitors

- A problem with CCRs is that they can be dispersed throughout the program

- Monitors provide encapsulation, and efficient condition synchronisation

- The critical regions are written as procedures and are encapsulated together into a single module

- All variables that must be accessed under mutual exclusion are hidden; all procedure calls into the module are guaranteed to be mutually exclusive

- Only the operations are visible outside the monitor

The Bounded Buffer

```
monitor buffer;

   export append, take;

     var (*declare necessary vars*)

     procedure append (I : integer);
        ...
     end;


     procedure take (var I : integer);
        ...
     end;
begin
  (* initialisation *)
 end;
```

How do we get condition synchronisation?

Condition Variables

- Different semantics exist

- In Hoare's monitors: a condition variable is acted upon by two semaphore-like operators WAIT and SIGNAL

- A process issuing a WAIT is blocked (suspended) and placed on a queue associated with the condition variable (cf semaphores: a wait on a condition variable always blocks unlike a wait on a semaphore)

- A blocked process releases its hold on the monitor, allowing another process to enter

- A SIGNAL releases one blocked process. If no process is blocked then the signal has no effect (cf semaphores)

The Bounded Buffer

```
monitor buffer;
  export append, take;

  var BUF : array[ . . . ] of integer;
  top, base : 0..size-1;  NumberInBuffer : integer;
  spaceavailable, itemavailable : condition;

  procedure append (I : integer);
  begin
    if NumberInBuffer = size then
      wait(spaceavailable);
    end if;
    BUF[top] := I;
    NumberInBuffer := NumberInBuffer+1;
    top := (top+1) mod size;
    signal(itemavailable)
  end append;
```

The Bounded Buffer

```
procedure take (var I : integer);
begin
  if NumberInBuffer = 0 then
    wait(itemavailable);
  end if;
  I := BUF[base];
  base := (base+1) mod size;
  NumberInBuffer := NumberInBuffer-1;
  signal(spaceavailable);
end take;

begin (* initialisation *)
  NumberInBuffer := 0;
  top := 0; base := 0
end;
```

- If a process calls take when there is nothing in the buffer then it will become suspended on itemavailable.

- A process appending an item will, however, signal this suspended process when an item does become available.
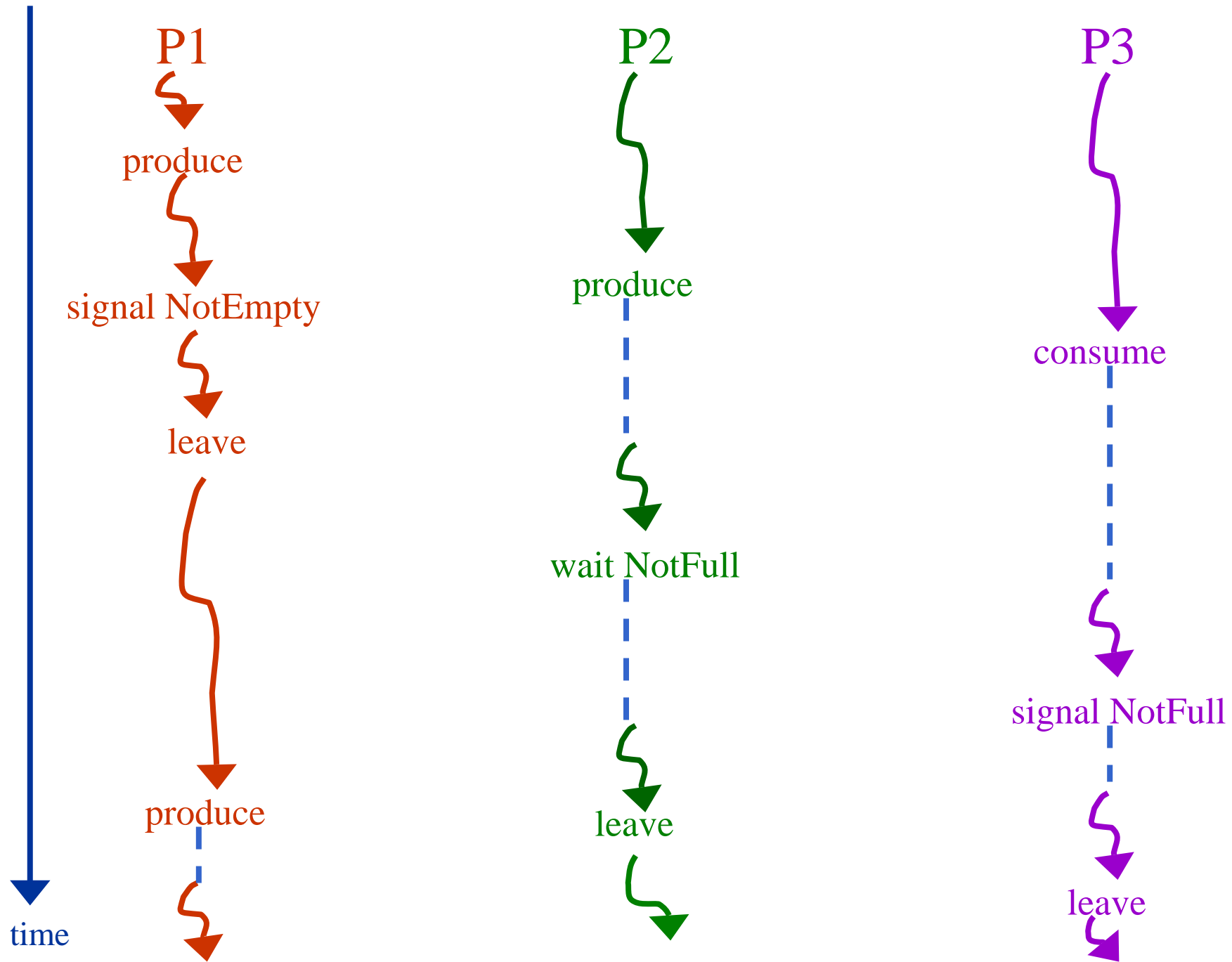
The semantics of SIGNAL

- What happens to the signalling process and the process that is restarted? Both must not be active in the monitor
- There are various semantics for SIGNAL

The Semantics of SIGNAL

- A signal is allowed only as the last action of a process before it leaves the monitor

- A signal operation has the side-effect of executing a return statement, i.e. the process is forced to leave

- A signal operation which unblocks another process has the effect of blocking itself; this process will only execute again when the monitor is free

- A signal operation which unblocks a process does not block the caller. The unblocked process must gain access to the monitor again

P1	P2	P3
produce	produce	consume
signal NotEmpty	wait NotFull	signal NotFull
leave	leave	leave
produce		

time

POSIX Mutexes and Condition Variables

- Provide the equivalent of a monitor for communication and synchronisation between threads

- Mutexes and condition variables have associated attribute objects; we will use default attributes only

- Example attributes:
  - set the semantics for a thread trying to lock a mutex it already has locked
  - allow sharing of mutexes and condition variables between processes
  - set/get priority ceiling
  - set/get the clock used for timeouts

```
typedef ... pthread_mutex_t;
typedef ... pthread_mutexattr_t;
typedef ... pthread_cond_t;
typedef ... pthread_condattr_t;
```

```c
int pthread_mutex_init(pthread_mutex_t *mutex,
                          const pthread_mutexattr_t *attr);
  /* initialises a mutex with certain attributes */

int pthread_mutex_destroy(pthread_mutex_t *mutex);
  /* destroys a mutex */
  /* undefined  behaviour if the mutex is locked  */



int pthread_cond_init(pthread_cond_t *cond,
                          const pthread_condattr_t *attr);
  /* initialises a condition variable with certain attributes */

int pthread_cond_destroy(pthread_cond_t *cond);
  /* destroys a condition variable */
  /* undefined, if threads are waiting on the cond.  variable  */
```

```c
int pthread_mutex_lock(pthread_mutex_t *mutex);
  /* lock the mutex; if locked already suspend calling thread */
  /* the owner of the mutex is the thread which locked it */

int pthread_mutex_trylock(pthread_mutex_t *mutex);
  /* as lock but gives an error if mutex is already locked */

int pthread_mutex_timedlock(pthread_mutex_t *mutex,
                            const struct timespec *abstime);
  /* as lock but gives an error if mutex cannot be obtained */
  /* by the timeout  */


int pthread_mutex_unlock(pthread_mutex_t *mutex);
  /* unlocks the mutex if called by the owning thread */
  /* undefined behaviour if calling thread is not the owner  */
  /* undefined behaviour if the mutex is not locked } */
  /* when successful, a blocked thread is released */
```

```c
int pthread_cond_wait(pthread_cond_t *cond,
                           pthread_mutex_t *mutex);
   /* called by thread which owns a locked mutex */
   /* undefined behaviour if the mutex is not locked */
   /* atomically blocks the caller on the cond variable and */
   /* releases the lock on mutex */
   /* a successful return indicates the mutex has been locked */

int pthread_cond_timedwait(pthread_cond_t *cond,
        pthread_mutex_t *mutex, const struct timespec *abstime);
   /* the same as pthread_cond_wait, except that a error is */
 /* returned if the timeout expires */
```

```
int pthread_cond_signal(pthread_cond_t *cond);
  /* unblocks at least one blocked thread */
  /* no effect if no threads are blocked */


int pthread_cond_broadcast(pthread_cond_t *cond);
  /* unblocks all blocked threads */
  /* no effect if no threads are blocked */


  /*all  unblocked threads automatically contend for */
  /* the associated mutex */
```

All functions return 0 if successful

Bounded Buffer in POSIX

```
#define BUFF_SIZE 10

typedef struct {
  pthread_mutex_t mutex;
  pthread_cond_t buffer_not_full;
  pthread_cond_t buffer_not_empty;
  int count, first, last;
  int buf[BUFF_SIZE];
  } buffer;

int append(int item, buffer *B ) {
  PTHREAD_MUTEX_LOCK(&B->mutex);
  while(B->count == BUFF_SIZE) {
    PTHREAD_COND_WAIT(&B->buffer_not_full, &B->mutex); }
  /* put data in the buffer and update count and last */
  PTHREAD_MUTEX_UNLOCK(&B->mutex);
  PTHREAD_COND_SIGNAL(&B->buffer_not_empty);
  return 0;
}
```

```
int take(int *item, buffer *B ) {
  PTHREAD_MUTEX_LOCK(&B->mutex);
  while(B->count == 0) {
    PTHREAD_COND_WAIT(&B->buffer_not_empty, &B->mutex);
  }
  /* get data from the buffer and update count and first */
  PTHREAD_MUTEX_UNLOCK(&B->mutex);
  PTHREAD_COND_SIGNAL(&B->buffer_not_full);
 return 0;
}

int initialize(buffer *B) {
  /* set the attribute objects and initialize the */
  /* mutexes and condition variable */
}
```

Readers/Writers Problem

Block of Data

How can monitors be used to allow many concurrent readers or a single writer but not both?

reader   reader   writer   writer
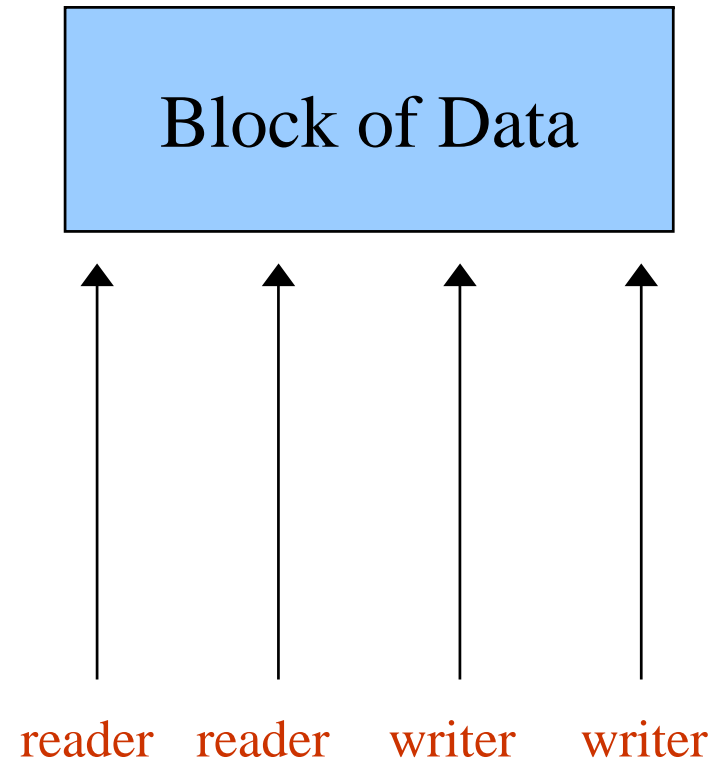
Hint

You will need to have an entry and exit protocol

Reader:

    start_read

    . . .

    stop_read

Writer:

    start_write

    . . .

    stop_write

Nested Monitor Calls

- What should be done if a process having made a nested monitor call is suspended in another monitor?

- The mutual exclusion in the last monitor call will be relinquished by the process, due to the semantics of the wait operation

- However, mutual exclusion will not be relinquished by processes in monitors from which the nested calls have been made; processes that attempt to invoke procedures in these monitors will become blocked

- Maintain the lock: e.g. POSIX

- Prohibit nested procedure calls altogether: e.g. Modula-1

- Provide constructs which specify that certain monitor procedures may release their mutual exclusion lock during remote calls

Criticisms of Monitors

- The monitor gives a structured and elegant solution to mutual exclusion problems such as the bounded buffer

- It does not, however, deal well with condition synchronization — requiring low-level condition variables

- All the criticisms surrounding the use of semaphores apply equally to condition variables

Protected Objects

- Combines the advantages of monitors with the advantages of conditional critical regions

- Data and operations are encapsulated

- Operations have automatic mutual exclusion

- Guards can be placed on operations for condition synchronization

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

- A protected unit 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;
```

The Protected Unit Body

```
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 in that calls are executed in mutual exclusion and 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, the calling task is suspended and remains suspended while the barrier evaluates to false, or there are other tasks currently active inside the protected unit

- Hence protected entry calls can be used to implement condition synchronisation

Condition Synchronisation Example

```ada
-- 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;
```

Bounded Buffer

```
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;
```

barriers

Subprogram Calls, Entry Calls and Barriers

- To call a protected object, simply name the object and the subprogram or entry:

```
My_Buffer.Put(Some_Data);
```

- As with task entry calls, the caller can use the select statement to issue timed or conditional protected entry calls

```
select                              select
  My_Buffer.Put(Some_Data);           My_Buffer.Put(Some_Data);
or                                  else
  delay 10.0;                           -- do something else
  -- do something else            end select;
end select;
```
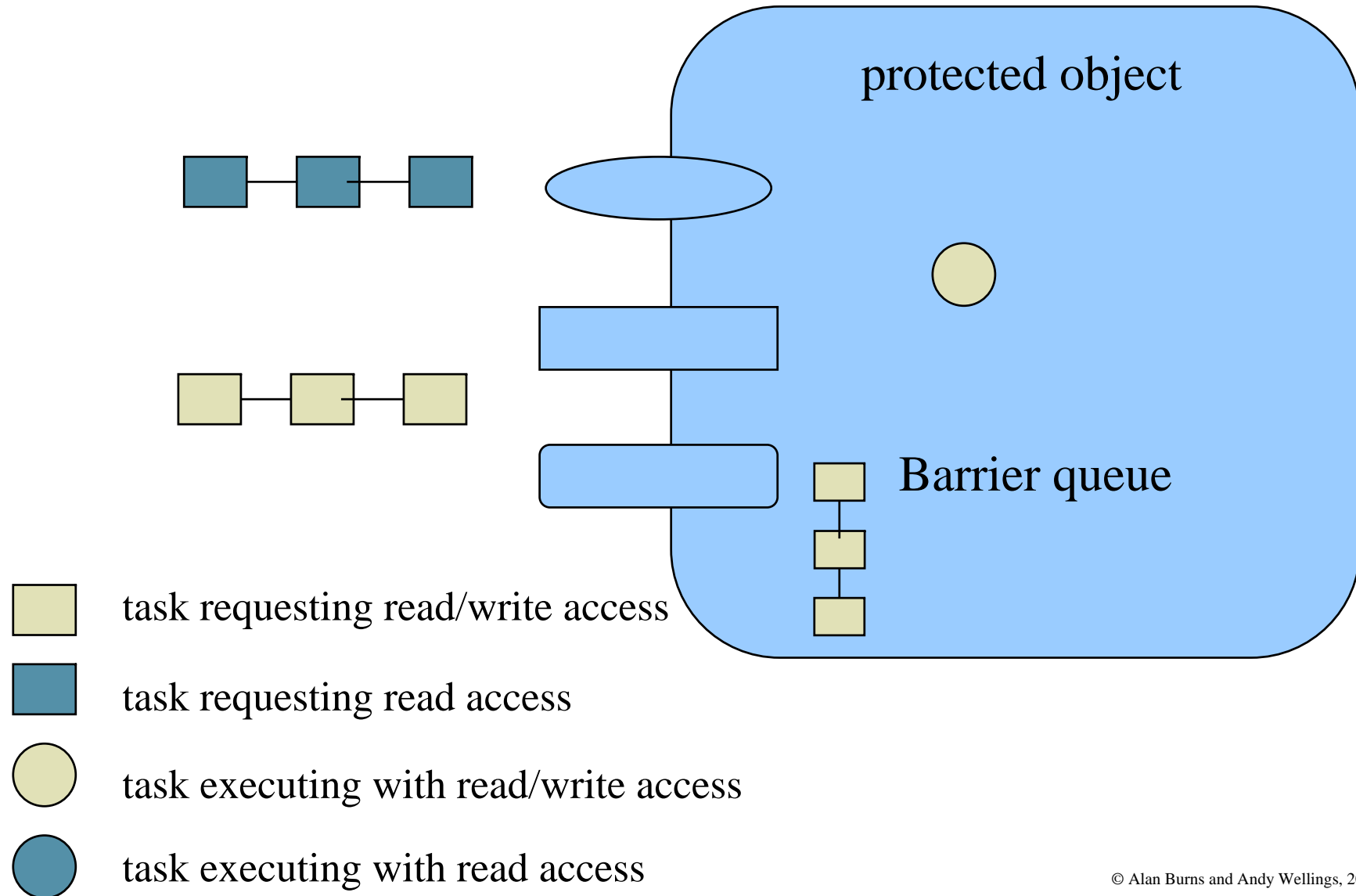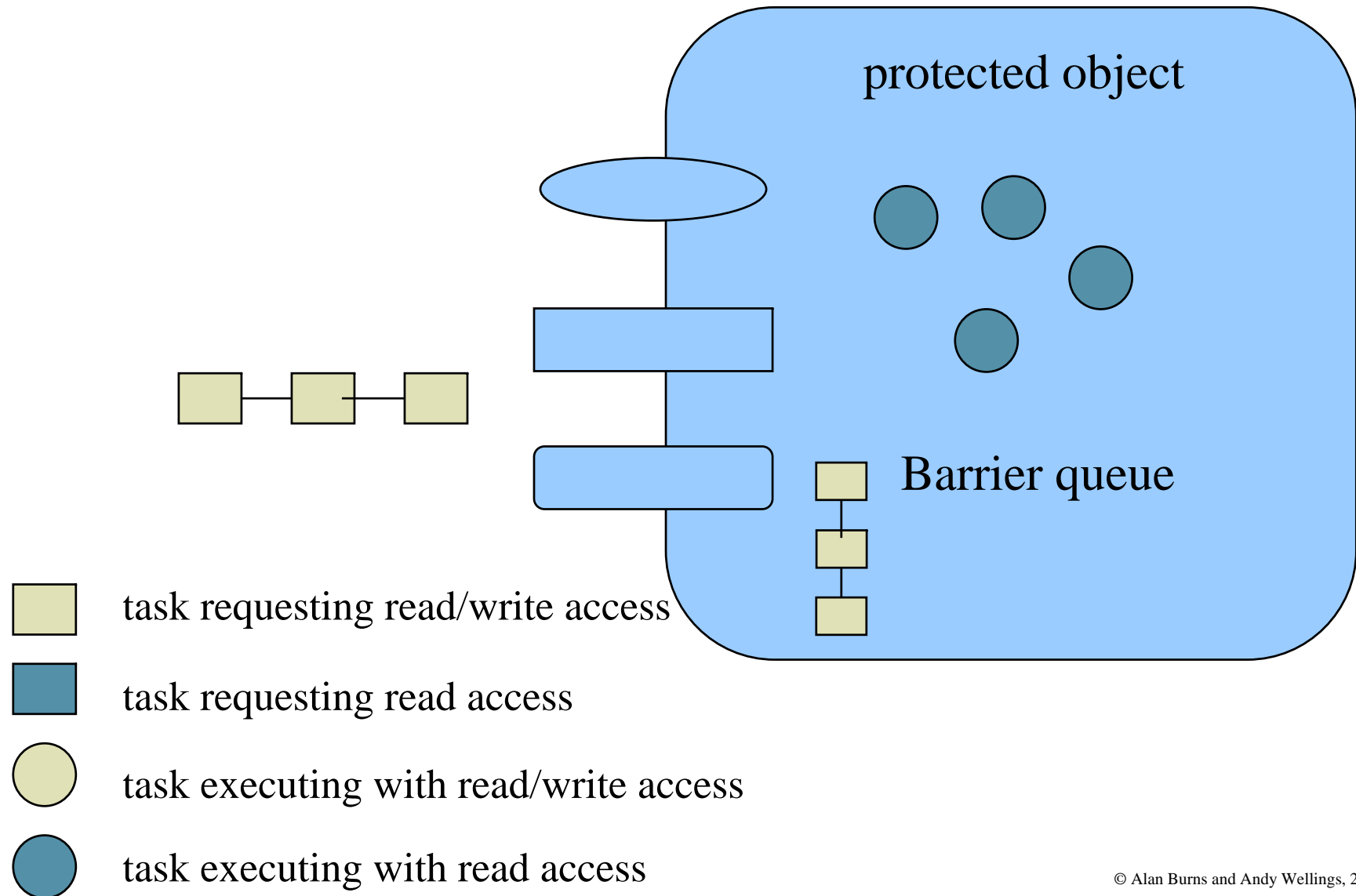
Barrier Evaluation

- At any instance in time, a barrier is either open or closed; it is open if the boolean expression evaluates to true, otherwise it is closed

- Barriers are evaluated when:

  1. a task calls one of its protected entries and the associated barrier references a variable or an attribute which might have changed since the barrier was last evaluated

  2. a task executes and leaves a protected procedure or entry, and there are tasks queued on entries whose barriers reference variables or attributes which might have changed since the barriers were last evaluated

Why are barriers not evaluated after a function call?

Write Access to a Protected Object

protected object

Barrier queue

task requesting read/write access

task requesting read access

task executing with read/write access

task executing with read access

© Alan Burns and Andy Wellings, 2001

Read Access to a Protected Object

protected object

Barrier queue

task requesting read/write access

task requesting read access

task executing with read/write access

task executing with read access

Resource Control Example

```
protected Resource_Control is
    entry Allocate;
    procedure Deallocate;
private
    Free : Boolean := True;
end Resource_Control;
```

Assuming a single resource, what is the body of this protected object?

Answer is in RTSPL book chapter 8

The Count Attribute

- The Count attribute defines the number of tasks queued on an entry

- Its evaluation requires the read/write lock

```
protected Blocker is
  entry Proceed;
private
  Release : Boolean := False;
end Blocker;
```

```
protected body Blocker is
  entry Proceed when
        Proceed'Count = 5 or
        Release is
  begin
    if Proceed'Count = 0 then
      Release := False;
    else
      Release := True;
    end if;
  end Proceed;
end Blocker;
```

Broadcast

```
protected type Broadcast is
  entry Receive(M : out message);
  procedure Send(M : message);
private
  New_Message : Message;
  Message_Arrived : Boolean := False;
end Broadcast;
```

Everyone queued on Receive should receive the message when send is called

Answer is in RTSPL book chapter 8

Semaphores

```ada
package Semaphore_Package is
  type Semaphore(Initial : Natural :=1)
          is limited private;
  procedure Wait (S : in out Semaphore);
  procedure Signal (S : in out Semaphore);
private
  protected type Semaphore(Initial : Natural :=1) is
    entry Wait_Imp;
    procedure Signal_Imp;
  private
    Value : Natural := Initial;
  end Semaphore;
end Semaphore_Package;
```

How would you implement this package?

Answer is in RTSPL
book chapter 8

Private Entries and Entry Families

- As with tasks, protected types can have private entries and entry families

- A protected type's private entries may be used during requeue operations

- A family can be declared by placing a discrete subtype definition in the specification of the entry

- The barrier associated with the entry can use the index of the family (usually to index into an array of booleans)

An Example of an Entry Family

```
type Group is range 1 .. 10;

type Group_Data_Arrived is array(Group) of Boolean;

protected type Group_Controller is

   procedure Send(To_Group : Group; Data : Data_Item);

   entry Receive(Group) (Data : out Data_Item);
private

   Arrived : Group_Data_Arrived := (others => False);

   The_Data : Data_Item;
end Group_Controller;

My_Controller : Group_Controller;
```

Entry Families Continued

```
protected body Group_Controller is
   procedure Send(To_Group : Group; Data : Data_Item) is
   begin
      if Receive(To_Group)'Count > 0 then
         Arrived(To_Group) := True;
         The_Data := Data;
      end if;
   end Send;

   entry Receive(for From in Group) (Data : out Data_Item)
            when Arrived(From) is
   begin
      if Receive(From)'Count = 0 then
         Arrived(From) := False;
      end if;
      Data := The_Data;
   end Receive;
end Group_Controller;
```

can't use this syntax for task entries

last one out closes the door!

Family of entries

Restrictions on Protected Objects

- Code inside a PO should be as short as possible
- ARM disallows potentially blocking operations
  - an entry call statement
  - a delay statement
  - task creation or activation
  - a call to a subprogram which contains a potentially blocking operation
  - a select statement
  - an accept statement
- `Program_Error` is raised if a blocking operation is called
- A call to an external protected procedure/function is not considered potentially blocking

Access Variables

```
protected type Broadcast is
   procedure Send (This_Altitude : Altitude);
   entry Receive (An_Altitude : out Altitude);
private
   Altitude_Arrived : Boolean := False;
   The_Altitude : Altitude;
end Broadcast;

type Prt_Broadcast is access all Broadcast;
```

a pointer to an object on the heap or
a statically aliased object

Broadcast Example

```
procedure Register (G: Ptr_Broadcast; Name : String);

function Find (G: String) return Ptr_Broadcast;
...

task body Barometric_Pressure_Reader is

   My_Group : Ptr_Broadcast := new Broadcast;

   -- My_Group : aliased Broadcast;
begin

   Register(My_Group, "Barometric_Pressure");

   -- Register(My_Group'Access, "Barometric_Pressure");
   ...

   My_Group.Send(Altitude_Reading);

   ...
end Barometric_Pressure_Reader;
```

Broadcast Example II

```ada
task Auto_Pilot;
task body Auto_Pilot is
  Bp_Reader : Ptr_Broadcast;

  Current_Altitude : Altitude;
begin
  Bp_Reader := Find("Barometric_Pressure");
  ...
  select
    Bp_Reader.Receive(Current_Altitude);
  or
    delay 0.1;
  end;
  ...
end Auto_Pilot;
```

Access to Protected Subprograms

- As well as declaring access types for protected types, Ada also allows the programmer to declare an access type to a protected subprogram

```
access_to_subprogram_definition ::=
 access [protected] procedure parameter_profile |
 access [protected] function parameter_and_result_profile
```

An example of this will be given later

Note, there is no access to a protected entry. Why?

Elaboration and Finalisation

- A protected object is elaborated when it comes into scope in the usual way

- Finalisation of a protected object requires that any tasks left on entry queues have the exception `Program_Error` raised. Generally they are two situations where this can happen:

  - a protected object is unchecked deallocated via an access pointer to it
  - a task calls an entry in another task which requeues the first task on a protected object which then goes out of scope

Example of Program_Error

```ada
task Client;
task body Client is
begin
   ...
   Server.Service;
   ...
end Client;
```

```ada
task Server is
   entry Service;
end Server;
```

```
task body Server is

   protected Local is

      entry Queue1;

      entry Queue2;

   end Local;

   protected body Local is separate;

   -- body not important here

begin
   ...
   accept Service do

      requeue Local.Queue1;

   end Service;
   ...
end Server;
```

It is possible for the Server task to terminate with a Client queued on the Local protected

Exceptions and Protected Objects

- `Program_Error` is raised when a protected action issues a potentially blocking operation (if detected)

- Any exception raised during the evaluation of a barrier, results in `Program_Error` being raised in all tasks currently waiting on the entry queues

- Any exception raised and not handled whilst executing a protected subprogram or entry, is propagated to the task that issued the protected call

- A task queued on a protected entry whose protected object is subsequently finalised has `Program_Error` raised

The Readers and Writers Problem

- Consider a file which needs mutual exclusion between writers and reader but not between multiple readers

- Protected objects can implement the readers/writers algorithm if the read operation is encoded as a function and the write as a procedure; however:

  - The programmer cannot easily control the order of access to the protected object; specifically, it is not possible to give preference to write operations over reads

  - If the read or write operations are potentially blocking, then they cannot be made from within a protected object

- To overcome these difficulties the PO must be used to implement an access control protocol for the read and write operations (rather than encapsulate them)

Readers/Writers

```ada
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;
```

Readers/Writers II

```
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 III

```ada
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 IV

```ada
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;
```

requeue allows a more robust solution

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 with the `synchronized` modifier, 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

- Non-synchronized methods do not require the lock and, therefore, can be called at any time

Example of Synchronized Methods

```java
public 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

- Provides a mechanism whereby a block can be labeled as synchronized

- 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;
  }
}
```

- Where **this** is the Java mechanism for obtaining the current object

Warning

- Used in its full generality, the synchronized block can undermine one of the advantages of monitor-like mechanisms, that of encapsulating synchronization constraints associate with an object into a single place in the program

- This is because it is not possible to understand the synchronization associated with a particular object by just looking at the object itself when other objects can name that object in a synchronized statement.

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

Static Data

- Static data is shared between all objects created from the class

- To obtain mutually exclusive access to this data requires all objects to 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 either labeling a static method with the synchronized modifier or by identifying the class's object in a synchronized block statement

- The latter can be obtained from the `Object` class associated with the object

- Note, however, that this class-wide lock is not obtained when synchronizing on the object

Static Data

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

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

  public 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 InterruptedException;
          // also 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 `IllegalMonitor-StateException` is thrown

Waiting and Notifying

- The `wait` method always blocks the calling thread and releases the lock associated with the object
- A `wait` within a nested monitor releases only the inner lock
- The `notify` method wakes up one waiting thread; the one woken is not defined by the Java language
- `Notify` does not release the lock; hence the woken thread must wait until it can obtain the lock before proceeding
- To wake up all waiting threads requires use of the `notifyAll` method
- If no thread is waiting, then `notify` and `notifyAll` have no effect

Thread Interruption

- A waiting thread can also be awoken if it is interrupted by another thread

- In this case the InterruptedException is thrown (see later in the course)

Condition Variables

- There are no explicit condition variables. An awoken thread should usually evaluate the condition on which it is waiting (if more than one exists and they are not mutually exclusive)

```
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
public synchronized void put(int item)
        throws InterruptedException
{

    while (numberInBuffer == size) {
      wait();
    };
    last = (last + 1) % size ; // % is modulus
    numberInBuffer++;
    buffer[last] = item;
    notify();
};

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

Mutually exclusive waiting

Readers-Writers Problem

- 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

```
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

```
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
- Get lock on condition variable on which you might want to sleep or notify, then get monitor lock

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

Readers-Writers Problem: Solution 2

```
public class ReadersWriters
{

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

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

```java
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

```java
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

```java
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();
  }
}
```

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

Summary

- **critical section** — code that must be executed under mutual exclusion

- **producer-consumer system** — two or more processes exchanging data via a finite buffer

- **busy waiting** — a process continually checking a condition to see if it is now able to proceed

- **livelock** — an error condition in which one or more processes are prohibited from progressing whilst using up processing cycles

- **deadlock** — a collection of suspended processes that cannot proceed

- **indefinite postponement** — a process being unable to proceed as resources are not made available

Summary

- **semaphore** — a non-negative integer that can only be acted upon by WAIT and SIGNAL atomic procedures

- Two more structured primitives are: condition critical regions and monitors

- Suspension in a monitor is achieved using condition variable

- 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