Synchronization#2: Monitors and Protected Objects

©2007 Dept Computer Science, Australian National University

Monitors

(Modula-1, Mesa — Dijkstra, Hoare)

Basic idea:

- Collect all operations and data-structures shared in critical regions in one place, the monitor.
- Formulate all operations as procedures or functions.
- Prohibit access to data-structures, other than by the monitor-procedures and functions.
- Assure mutual exclusion of all monitor-procedures and functions.

©2007 Dept Computer Science, Australian National University

Monitors

```
monitor buffer;
  export append, take;
  var (* declare protected vars *)
  procedure append (I : integer);
    …
  procedure take (var I : integer);
    …
begin
  (* initialisation *)
end;
```

- How to realize conditional synchronization?

©2007 Dept Computer Science, Australian National University

Monitors with condition synchronization

(Hoare)

Hoare-monitors:

- Condition variables are implemented by semaphores (Wait and Signal).
- Queues for tasks suspended on condition variables are realized.
- A suspended task releases its lock on the monitor, enabling another task to enter.
  - More efficient evaluation of the guards: the task leaving the monitor can evaluate all guards and the right tasks can be activated.
  - Blocked tasks may be ordered and livelocks prevented.

©2007 Dept Computer Science, Australian National University

Monitors with condition synchronization

```
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;  …
```

©2007 Dept Computer Science, Australian National University

Monitors with condition synchronization

```
…
  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;
```

The signalling and the waiting process are both active in the monitor!

©2007 Dept Computer Science, Australian National University

Monitors with condition synchronization

- Suggestions to overcome the multiple-tasks-in-monitor-problem:
  - 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.
  - Hoare, Modula-1, POSIX: a signal operation which unblocks another process has the side-effect of **blocking the current process**; this process will only execute again once the monitor is unlocked again.
  - A signal operation which unblocks a process does not block the caller, but the unblocked process must **gain access to the monitor again**.

©2007 Dept Computer Science, Australian National University

Monitors in Modula-1

- `wait (s, r):`
  - delays the caller until condition variable s is true (r is the rank (or ‘priority’) of the caller).
- `send (s):`
  - If a process is waiting for the condition variable s, then the process at the top of the queue of the highest filled rank is activated (and the caller suspended).
- `awaited (s):`
  - check for waiting processes on s.

©2007 Dept Computer Science, Australian National University

Monitors in Modula-1

```
INTERFACE MODULE resource_control;
  DEFINE allocate, deallocate;
  VAR busy : BOOLEAN; free : SIGNAL;
  PROCEDURE allocate;
  BEGIN
    IF busy THEN WAIT (free) END;
    busy := TRUE;
  END;
  PROCEDURE deallocate;
  BEGIN
    busy := FALSE;
    SEND (free); -- or: IF AWAITED (free) THEN SEND (free);
  END;
BEGIN
  busy := false;
END.
```

©2007 Dept Computer Science, Australian National University

Monitors in 'C' / POSIX (types and creation)

Synchronization between POSIX-threads:

```
typedef … pthread_mutex_t;
typedef … pthread_mutexattr_t;
typedef … pthread_cond_t;
typedef … pthread_condattr_t;
int pthread_mutex_init      ( pthread_mutex_t     *mutex,
                              const pthread_mutexattr_t *attr);
int pthread_mutex_destroy   ( pthread_mutex_t     *mutex);
int pthread_cond_init       ( pthread_cond_t      *cond,
                              const pthread_condattr_t  *attr);
int pthread_cond_destroy    ( pthread_cond_t      *cond);
…
```

©2007 Dept Computer Science, Australian National University

Monitors in ‘C’ / POSIX (types and creation)

Synchronization between POSIX-threads:

typedef … pthread_mutex_t;
typedef … **pthread_mutexattr_t;**
typedef … pthread_cond_t;
typedef … **pthread_condattr_t;**
int pthread_mutex_init

int pthread_mutex_destroy
int pthread_cond_init

int pthread_cond_destroy
…

Attributes include:

- semantics for trying to lock a mutex which is locked already by the same thread
- sharing of mutexes and condition variables between processes
- priority ceiling
- clock used for timeouts

©2007 Dept Computer Science, Australian National University

Monitors in ‘C’ / POSIX (types and creation)

Synchronization between POSIX-threads:

typedef … pthread_mutex_t;
typedef … pthread_mutexattr_t;
typedef … pthread_cond_t;
typedef … pthread_condattr_t;
int pthread_mutex_init (

int pthread_mutex_destroy (← Undefined, if locked
int pthread_cond_init (

int pthread_cond_destroy (← Undefined, if threads waiting
…

©2007 Dept Computer Science, Australian National University

Monitors in 'C' / POSIX (operators)

```
int pthread_mutex_lock       ( pthread_mutex_t          *mutex);
int pthread_mutex_trylock    ( pthread_mutex_t          *mutex);
int pthread_mutex_timedlock  ( pthread_mutex_t          *mutex,
                               const struct timespec    *abstime);
int pthread_mutex_unlock     ( pthread_mutex_t          *mutex);
int pthread_cond_wait        ( pthread_cond_t           *cond,
                               pthread_mutex_t          *mutex);
int pthread_cond_timedwait   ( pthread_cond_t           *cond,
                               pthread_mutex_t          *mutex,
                               const struct timespec    *abstime);
int pthread_cond_signal      ( pthread_cond_t           *cond);
int pthread_cond_broadcast   ( pthread_cond_t           *cond);
```

©2007 Dept Computer Science, Australian National University

Monitors in 'C' / POSIX (operators)

```
int pthread_mutex_lock       (
int pthread_mutex_trylock    (
int pthread_mutex_timedlock  (

int pthread_mutex_unlock     (
int pthread_cond_wait        (

int pthread_cond_timedwait   (


int pthread_cond_signal      (  <- Unblock at least one thread
int pthread_cond_broadcast   (  <- Unblock all thread
```

©2007 Dept Computer Science, Australian National University

Monitors in 'C' / POSIX (operators)

int pthread_mutex_lock (
int pthread_mutex_trylock (
int pthread_mutex_timedlock (

int pthread_mutex_unlock (
int pthread_cond_wait (

int pthread_cond_timedwait (

int pthread_cond_signal (
int pthread_cond_broadcast (

Undefined if called out of order!

©2007 Dept Computer Science, Australian National University

Monitors in 'C' / POSIX (operators)

int pthread_mutex_lock (
int pthread_mutex_trylock (
int pthread_mutex_timedlock (

int pthread_mutex_unlock (
int pthread_cond_wait (

int pthread_cond_timedwait (

int pthread_cond_signal (
int pthread_cond_broadcast (

Can be called any time, anywhere
(multiple lock reaction can be specified)

©2007 Dept Computer Science, Australian National University

Monitors in 'C' / POSIX (example, definition)

```
#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;
```

©2007 Dept Computer Science, Australian National University

Monitors in 'C' / POSIX (example, operations)

```
int take (int *item, buffer *B) {
  PTHREAD_MUTEX_LOCK (&B->mutex);
  while (B->count == 0) {
    PTHREAD_COND_WAIT (
      &B->buffer_not_empty,
      &B->mutex);
  }
  PTHREAD_MUTEX_UNLOCK (&B->mutex);
  PTHREAD_COND_SIGNAL (
    &B->buffer_not_full);
  return 0;
}
```

```
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);
  }
  PTHREAD_MUTEX_UNLOCK (&B->mutex);
  PTHREAD_COND_SIGNAL (
    &B->buffer_not_empty);
  return 0;
}
```

©2007 Dept Computer Science, Australian National University

Monitors in 'C' / POSIX (example, operations)

```
int take (int *item, buffer *B) {
  PTHREAD_MUTEX_LOCK (&B->mutex);
  while (B->count == 0) {
    PTHREAD_COND_WAIT (
      &B->buffer_not_empty,
      &B->mutex);
  }
  PTHREAD_MUTEX_UNLOCK (&B->mutex);
  PTHREAD_COND_SIGNAL (
    buffer_not_full);
```

correct?

```
int append (int item, buffer *B) {
  PTHREAD_MUTEX_LOCK (&B
  while (B->count == BUFF_SIZE) {
    PTHREAD_COND_WAIT (
      &B->buffer_not_full,
      &B->mutex);
  }
  PTHREAD_MUTEX_UNLOCK (&B->mutex);
  PTHREAD_COND_SIGNAL (
    &B->buffer_not_empty);
  return 0;
}
```

©2007 Dept Computer Science, Australian National University

Monitors in Java

Java provides two mechanisms to construct monitors:

- Synchronized methods and code blocks
  - all methods and code blocks which are using the synchronized tag are mutually exclusive with respect to the addressed class.
- Notification methods: `wait`, `notify`, and `notifyAll`
  - can be used only in synchronized regions and are waking any or all threads, which are waiting in the same synchronized object.

©2007 Dept Computer Science, Australian National University

Synchronized methods and code blocks

- In order to implement a monitor *all* methods in an object need to be synchronized.
  - any other standard method can break the monitor and enter at any time.
- Methods outside the monitor-object can synchronize at this object.
  - it is impossible to analyse a monitor locally, since lock accesses can exist all over the system.
- Static data is shared between all objects of a class.
  - access to static data need to be synchronized with *all* objects of a class.
  - Either in static synchronized blocks: synchronized (this.getClass()) {…} or in static methods: public synchronized static *<method>* {…}

©2007 Dept Computer Science, Australian National University

Notification methods: wait, notify, and notifyAll

- wait suspends the thread and releases the local lock only
  - nested wait-calls will keep all enclosing locks.
- notify and notifyAll do not release the lock.
  - methods, which are activated via notification need to wait for lock-access.
- Java does *not* require any specific release order (like a queue) for wait-suspended threads
  - livelocks are *not* prevented at this level (in opposition to RT-Java).
- There are no explicit conditional variables.
  - notified threads need to wait for the lock to be released **and** to re-evaluate its entry condition

©2007 Dept Computer Science, Australian National University

Monitors in Java (multiple-readers-one-writer-example)

Each of the **readers** uses these monitor.calls:

```
startRead ();
  // read the shared data only
stopRead ();
```

Each of the **writers** uses these monitor.calls:

```
startWrite ();
  // manipulate the shared data
stopWrite ();
```

- Construct a monitor, which allows multiple readers **or** one writer at a time inside the critical regions

©2007 Dept Computer Science, Australian National University

Monitors in Java: wait-notifyAll method

```
public class ReadersWriters
{
  private int     readers        = 0;
  private int     waitingWriters = 0;
  private boolean writing        = false;

…
```

©2007 Dept Computer Science, Australian National University

Monitors in Java: wait-notifyAll method

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

©2007 Dept Computer Science, Australian National University

Monitors in Java: wait-notifyAll method

```
… public synchronized void StartRead () throws InterruptedException
  {
    while (writing || waitingWriters > 0)
    {
      wait();
    }
    readers++;
  }
  public synchronized void StopRead()
  {
    readers--;
    if (readers == 0) notifyAll();
  }
}
```

- Whenever a synchronized region is left:
  - **All** threads are notified
  - **All** threads are re-evaluating their guards

©2007 Dept Computer Science, Australian National University

Standard Monitor Solution in Java

- Declare the monitored data-structures private to the monitor object (non-static).
- Introduce a class `ConditionVariable`:

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

- Introduce synchronization-scopes in monitor-methods:
  - synchronize on the *adequate conditional variables* ***first*** and
  - synchronize on the *monitor-object* ***second.***
- ***make sure that all*** methods in the monitor are implementing the correct synchronizations.
- make sure that ***no other method*** in the whole system is synchronizing on this monitor-object.

©2007 Dept Computer Science, Australian National University

Monitors in Java: use of external conditional variables

```
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 ();
…
```

©2007 Dept Computer Science, Australian National University

Monitors in Java: use of external conditional variables

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

©2007 Dept Computer Science, Australian National University

Monitors in Java: use of external conditional variables

```
… 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;
          }
  } } } } …
```

©2007 Dept Computer Science, Australian National University

Monitors in Java: use of external conditional variables

```
… 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 ();
  } } …
```

©2007 Dept Computer Science, Australian National University

Monitors in Java: use of external conditional variables

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

©2007 Dept Computer Science, Australian National University

Object-orientation and synchronization

- Since mutual exclusion, notification, and condition synchronization schemes need to be designed and analysed considering the implementation of all involved methods and guards:
  - new methods cannot be added without re-evaluating the whole class!
- In opposition to the general re-usage idea of object-oriented programming, the re-usage of synchronized classes (e.g. monitors) need to be considered carefully.
  - The parent class might need to be adapted in order to suit the global synchronization scheme.
  - Inheritance anomaly (Matsuoka & Yonezawa '93)
- Methods to design and analyse expandible synchronized systems exist, but are fairly complex and are not provided in any current object-oriented language.

©2007 Dept Computer Science, Australian National University

Monitors in POSIX & Real-time Java

- Flexible and universal, but relies on conventions rather than compilers
- POSIX offers conditional variables
- Real-time Java is more supportive than POSIX in terms of data-encapsulation
- Extreme care must be taken when employing object-oriented programming and monitors

©2007 Dept Computer Science, Australian National University

Nested monitor calls

- Assuming a thread in a monitor is calling an operation in another monitor and is suspended at a conditional variable there:
  - the called monitor is aware of the suspension and allows other threads to enter.
  - the calling monitor is possibly *not aware* of the suspension and **keeps its lock!**
  - the unjustified locked calling monitor reduces the system performance and leads to potential deadlocks.
- Suggestions to solve this situation:
  - Maintain the lock anyway: e.g. POSIX, Java
  - Prohibit nested procedure calls: e.g. Modula-1
  - Provide constructs which specify the release of a monitor lock for remote calls, e.g. Ada95

©2007 Dept Computer Science, Australian National University

Criticism of monitors

- Mutual exclusion is solved elegantly and safely
- Conditional synchronization is on the level of semaphores still
  - all criticism on semaphores apply
- Mixture of low-level and high-level synchronization constructs

©2007 Dept Computer Science, Australian National University

Synchronization by protected objects

- Combine
  - the **encapsulation** feature of monitors
  - the **coordinated entries** of conditional critical regions
- Protected objects
  - *all* controlled data and operations are encapsulated
  - *all* operations are mutual exclusive
  - entry guards are *attached* to operations
  - the protected interface allows for operations on data
  - no protected data is accessible (other than by defined operations)
  - tasks are queued (according to their priorities)

©2007 Dept Computer Science, Australian National University

ProtObj: simultaneous read-access

- Some read-only operations *do not need to be mutual exclusive*:

```
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 *functions* can have 'in' parameters only and are not allowed to alter the private data (enforced by the compiler).
  - protected functions allow **simultaneous access** (but mutual exclusive with other operations).
- There is no defined priority between functions and other protected operations in Ada95.

©2007 Dept Computer Science, Australian National University

ProtObj: Barriers

- *Condition synchronization* is realized in the form of protected procedures combined with boolean conditional variables (**barriers**) ⇒ **entries** in Ada95:

```
Buffer_Size : constant Integer := 10;
type     Index     is mod Buffer_Size;
subtype Count     is Natural range 0 .. Buffer_Size;
type     Buffer_T 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;
   Buffer : Buffer_T;
end Bounded_Buffer;
```

©2007 Dept Computer Science, Australian National University

ProtObj: Barriers

```
protected body Bounded_Buffer is
 entry Get(Item : out Data_Item) when Num > 0 is
   begin
     Item  := Buffer (First);
     First := First + 1;
     Num   := Num - 1;
   end Get;
 entry Put(Item : in Data_Item) when Num < Buffer_Size is
   begin
     Last           := Last + 1;
     Buffer (Last) := Item;
     Num            := Num + 1;
   end Put;
end Bounded_Buffer;
```

©2007 Dept Computer Science, Australian National University

ProtObj: Protected Entries

- *Protected entries* are used like task entries:

```
Buffer : Bounded_Buffer;
```

```
select
  Buffer.Put (Some_Data);
or
  delay 10.0;
    -- do something after 10 s.
end select;
```

```
select
  delay 10.0;
then abort
  Buffer.Put (Some_Data);
    -- try to enter for 10 s.
end select;
```

```
select
  Buffer.Get (Some_Data);
else
  -- do something else
end select;
```

```
select
  Buffer.Get (Some_Data);
then abort
  -- meanwhile try something else
end select;
```

©2007 Dept Computer Science, Australian National University

ProtObj: barrier evaluation

- Barrier evaluations and task activations:
  - on ***calling a protected entry***, the associated barrier is evaluated (only those parts of the barrier which might have changed since the last evaluation).
  - on ***leaving a protected procedure or entry***, related barriers with tasks queued are evaluated (only those parts of the barriers which might have been altered by this procedure / entry or which might have changed since the last evaluation).
- Barriers are not evaluated *while inside* a protected object or *on leaving a protected function*.

©2007 Dept Computer Science, Australian National University

ProtObj: Operations on entry queues

- The count attribute indicates the number of tasks waiting at a specific queue:

```
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
    Release := Proceed'count > 0;
  end Proceed;
end Blocker;
```

©2007 Dept Computer Science, Australian National University

ProtObj: Operations on entry queues

- The count attribute indicates the number of tasks waiting at a specific queue:

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

```
protected body Broadcast is
  entry Receive (M: out Message)
    when Arrived is
  begin
    M := New_Message
    Arrived := Receive'count > 0;
  end Proceed;
  procedure Send (M: in  Message) is
  begin
    New_Message := M;
    Arrived := Receive'count > 0;
  end Send;
end Broadcast;
```

©2007 Dept Computer Science, Australian National University

ProtObj: entry families, requeue & private entries

- ***Entry families***:
  - a protected entry declaration can contain a discrete subtype selector, which can be evaluated by the barrier (other parameters cannot be evaluated by barriers) and implements an array of protected entries.
- ***Requeue facility***:
  - protected operations can use 'requeue' to redirect tasks to other internal, external, or private entries. The current protected operation is finished and the lock on the object is released.
  - '***Internal progress first'-rule***: internally requeued tasks are placed at the **head** of the waiting queue!
- ***Private entries***:
  - protected entries which are not accessible from outside the protected object, but can be employed as destinations for requeue operations.

©2007 Dept Computer Science, Australian National University

ProtObj: entry families

```
package Modes is
  type Mode_T is
    (Takeoff, Ascent, Cruising,
     Descent, Landing);
  protected Mode_Gate is
    procedure Set_Mode
            (Mode: in Mode_T);
    entry Wait_For_Mode
            (Mode_T);
  private
    Current_Mode : Mode_Type
                := Takeoff;
  end Mode_Gate;
end Modes;
```

```
package body Modes is
  protected body Mode_Gate is
    procedure Set_Mode
          (Mode: in Mode_T) is
      begin
        Current_Mode := Mode;
      end Set_Mode;
    entry Wait_For_Mode
      (for Mode in Mode_T)
      when Current_Mode = Mode is
      begin null;
      end Wait_For_Mode;
  end Mode_Gate;
end Modes;
```

©2007 Dept Computer Science, Australian National University

ProtObj: requeue & private entries

- How to implement a queue, at which every task can be released only once per triggering event?
  - e.g. by employing two entries:

```
package Single_Release is
   entry     Wait;
   procedure Trigger;
private
   Front_Door,
   Main_Door  : Boolean := False;
   entry Queue;
end Single_Release;
```

©2007 Dept Computer Science, Australian National University

ProtObj: requeue & private entries

```
package body Single_Release is
   entry Wait
      when Front_Door is
      begin
         if Wait'Count = 0 then
            Front_Door := False;
            Main_Door  := True;
         end if;
         requeue Queue;
      end Wait;

   entry Queue
      when Main_Door is
      begin
         if Queue'count = 0 then
            Main_Door := False;
         end if;;
      end Queue;
   procedure Trigger is
      begin
         Front_Door := True;
      end Trigger;
end Single_Release;
```

opening the main door before requeuing?

©2007 Dept Computer Science, Australian National University

ProtObj: restrictions

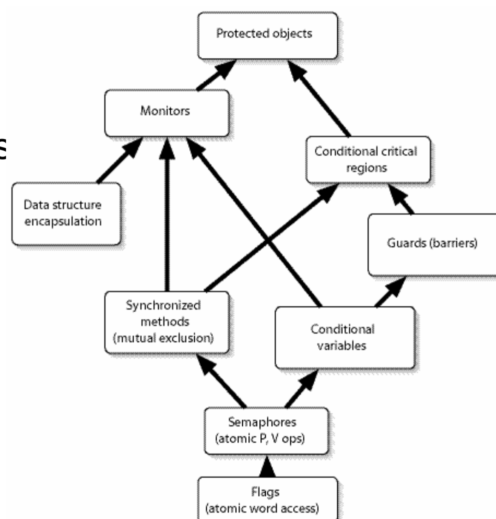
- Code inside a protected procedure, function or entry is bound to non-blocking operations (which would keep the whole protected object locked).
- Thus the following operations are prohibited:
  - entry call statements
  - delay statements
  - task creations or activations
  - calls to sub-programs which contains a potentially blocking operation
  - select statements
  - accept statements
- The requeue facility allows for a potentially blocking operation, but releases the current lock!

©2007 Dept Computer Science, Australian National University

Shared memory synchronization: Summary

Criteria:

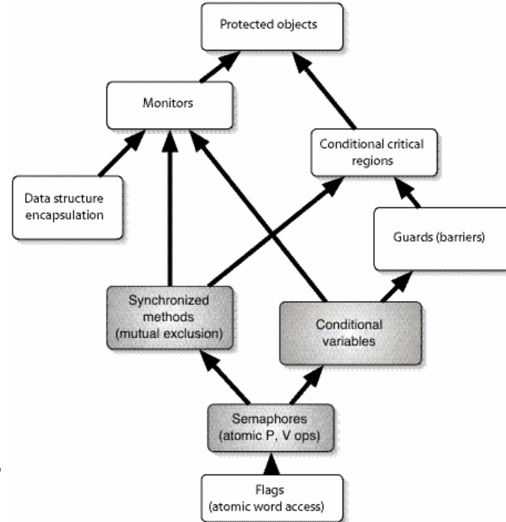
- level of abstraction
- centralized vs. distributed concepts
- support for consistency and correctness validations
- error sensitivity
- predictability
- efficiency



©2007 Dept Computer Science, Australian National University

Shared memory synchronization: ***POSIX***

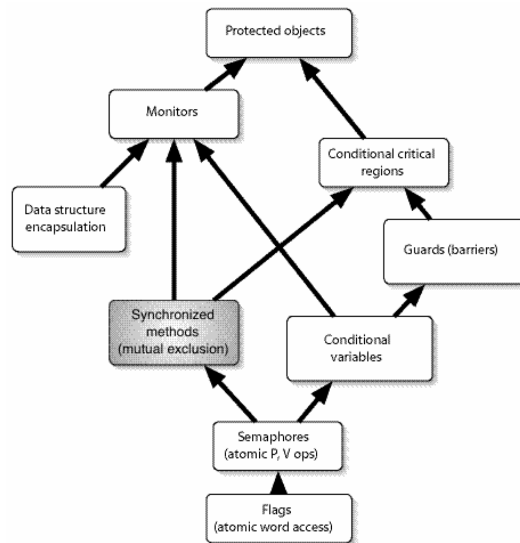
- all low level constructs available
- no connection with the actual data-structures
- error-prone
- non-determinism introduced by 'release some' semantics of conditional variables (cond_signal).



©2007 Dept Computer Science, Australian National University

Shared memory synchronization: ***Java***

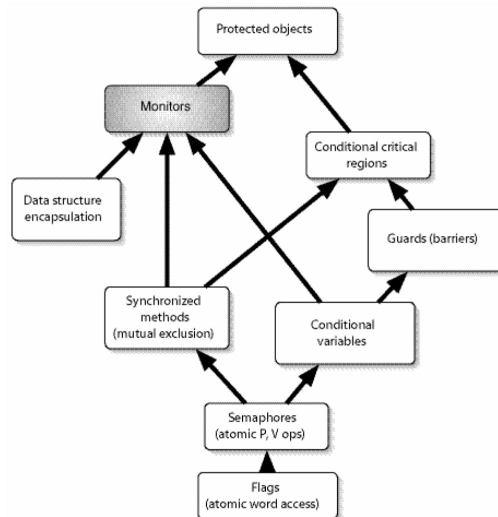
- mutual exclusion (synchronized methods) as the only support.
- general notification feature (no conditional variables)
- non-restricted object oriented extension introduces hard to predict timing behaviours.



©2007 Dept Computer Science, Australian National University

Shared memory synchronization: ***Modula-1, CHILL***

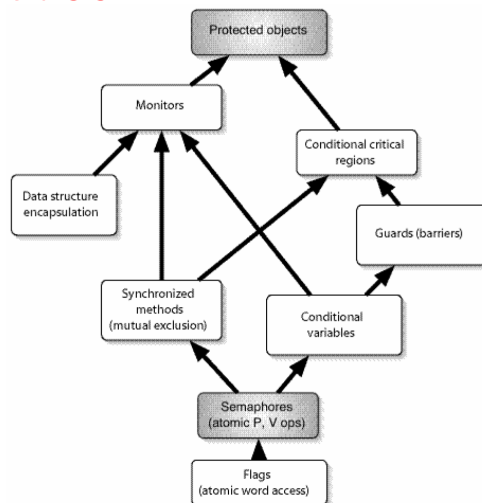
- full monitor implementation (Dijkstra-Hoare monitor concept).
  - no more, no less
  - all features of and criticism about monitors apply.



©2007 Dept Computer Science, Australian National University

Shared memory synchronization: ***Ada95***

- complete synchronization support
- low-level semaphores for very special cases.
- predictable timing (⇒scheduler).
  - most memory oriented synchronization conditions are realized by the compiler or the run-time environment directly rather then the programmer.
- (Ada95 is currently without any mainstream competitor in this field)



©2007 Dept Computer Science, Australian National University