Centralized Synchronization

Monitors
Protected Objects
Monitors

- Centralize all operations on a shared data structure in one place, the monitor.
- Formulate all operations as procedures or functions.
- Prohibit access to data structure, other than by the monitor procedures and functions.
- Assure mutual exclusion of all monitor procedures and functions.
- (Modula-1, Mesa — Dijkstra, Hoare)
Monitors

monitor buffer;
    export append, take;

var (* declare protected vars *)

procedure append (I : integer);
    ...
procedure take (var I : integer);
    ...

begin
    (* initialization)
end;

How to implement condition synchronization?
Monitors with Condition Synchronization

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

Monitors with Condition Synchronization

```pascal
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; ...
```

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

Both signalling and waiting processes are active in the monitor!
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 re-gain access to the monitor.
Monitors in Modula-1

- procedure wait (s, r):
  delays the caller until condition variable s is true (r is the rank (or ‘priority’) of the caller)
- procedure 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)
- function awaited (s) return integer:
  check for waiting processes on s
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.
Monitors in POSIX

- Mutex + condition variables = monitor

```c
typedef ... pthread_mutex_t;
typedef ... pthread_mutex_attr_t;
typedef ... pthread_cond_t;
typedef ... pthread_cond_attr_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);
```
Monitors in POSIX

typedef…pthread_mutex_attr_t;
typedef…pthread_cond_attr_t;

• Attributes include:
  – semantics for trying to lock a mutex which is already locked by the same thread
  – sharing of mutexes and condition variables between processes
  – priority ceiling
  – clock used for timeouts
Monitors in POSIX: operations

```c
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); // unblock at least one
int pthread_cond_broadcast(
    pthread_cond_t *cond); // unblock all threads
```
Monitors in C#

```csharp
using System;
using System.Threading;
static long data_to_protect = 0;
static void Reader() {
    try {
        Monitor.Enter(data_to_protect);
        Monitor.Wait(data_to_protect);
        ... read out protected data
    }
    finally {
        Monitor.Exit(data_to_protect);
    }
}

static void Writer() {
    try {
        Monitor.Enter(data_to_protect);
        ... write protected data
        Monitor.Pulse(data_to_protect);
    }
    finally {
        Monitor.Exit(data_to_protect);
    }
}
```
Object-Orientation and Synchronization

• Since mutual exclusion and condition synchronization schemes must consider all involved methods and guards, new behaviour cannot be added without re-evaluating the class!

• Re-use through inheritance does not translate to synchronized classes (e.g. monitors) and thus need to be considered carefully.

• Parent class might need to be adapted in order to suit the global synchronization scheme.

• Methods to design and analyse expandable synchronized systems are complex and not offered in any concurrent programming language. Alternatively, inheritance can be banned for synchronized objects (e.g. Ada).

Monitors in Sequential Languages

Monitors in POSIX, Visual C++, C#, Visual Basic & Java

- All provide lower-level primitives for the construction of monitors
- All rely on conventions rather than compiler checks
- Visual C++, C# & Visual Basic offer data-encapsulation and connection to the monitor
- Java offers data-encapsulation (yet not with respect to a monitor)
- POSIX (being a collection of library calls) does not provide any data-encapsulation by itself
- Extreme care must be taken when employing object-oriented programming and synchronization (including monitors)
Nested Monitor Calls

• Assuming a thread in a monitor is calling an operation in another monitor and is suspended at a conditional variable there:
  – called monitor is aware of the suspension and allows other threads to enter
  – 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 monitor calls e.g. Modula-1
  – Specify release of monitor lock for remote calls e.g. Ada
Criticism of Monitors

- Mutual exclusion is solved elegantly and safely
- Conditional synchronization is on the level of semaphores
  - all criticism about semaphores applies to monitors
- Mixture of low-level and high-level synchronization constructs
Synchronization by Protected Objects

Combine the *encapsulation* feature of monitors with the *coordinated entries* of conditional critical regions

- All controlled data and operations are *encapsulated*
- Operations are *mutually exclusive* (with exceptions for read-only operations)
- *Guards* (predicates) are syntactically attached to entries
- No protected data is accessible (other than by the defined operations)
- *Fairness* inside operations is guaranteed by queuing (according to priorities)
- *Fairness* across all operations is guaranteed by the "internal progress first" rule
- Re-blocking provided by *re-queuing* to entries (no internal condition variables)
Protected Objects: Simultaneous Read Access

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

```ada
protected type Shared_Data (Initial : Data_Item) is
  function Read return Data_Item;
  procedure Write (New_Value : 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 private data (enforced by compiler)
- protected functions allow simultaneous access (but mutual exclusive with other operations). … there is no defined priority between functions and other protected operations in Ada
Protected Objects: Condition Synchronization

Condition synchronization realized by *protected procedures* combined with *barriers* guarded by boolean predicates (called *entries* in Ada):

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 : Data Item);

**private**

- First : Index := Index’First; Last : Index := Index’Last; Num : Count := 0;
- Buffer : Buffer T;

**end** Bounded_Buffer;
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 : Data_Item) when Num < Buffer_Size is
begin
    Last := Last + 1;
    Buffer (Last) := Item; Num := Num + 1;
end Put;
end Bounded_Buffer;
Protected Objects: Condition Synchronization

```
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 : Data_Item) when Num < Buffer_Size is
  begin
    Last := Last + 1;
    Buffer (Last) := Item; Num := Num + 1;
  end Put;
end Bounded_Buffer;
```
Centralized Synchronization

Protected Objects: Selective Synchronization

Buffer : Bounded_Buffer;

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

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

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

select
    Buffer.Get (Some_Data);
then abort
    -- meanwhile try something else
end select;
Protected Objects: Barrier Evaluation

Barrier in protected objects need to be evaluated only on two occasions:

- on *creating a protected object*, all barriers are evaluated according to the initial values of the internal, protected data
- on *leaving a protected procedure or entry*, all potentially altered barriers are re-evaluated

Barriers are not evaluated while *inside* a protected object or on *leaving a protected function*
Protected Objects: Entry Queues

The `count` attribute indicates number of tasks waiting on a queue:

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

protected body Block_Five is
    entry Proceed
        when Proceed'count > 5
        or Release is
        begin
            Release := Proceed'count > 0;
        end Proceed;
end Block_Five;
```
Centralized Synchronization

Protected Objects: Entry Queues

protected type Broadcast is
    entry Receive (M: out Message);
    procedure Send (M: 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: Message) is
    begin
        New_Message := M;
        Arrived := Receive'count > 0;
    end Send;
end Broadcast;
Protected Objects: Entry Families

Additional, essential primitives for concurrent control flows:

- 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.
Protected Objects: Entry Families

package Modes is

  type Mode_T is
    (Takeoff, Ascent, Cruising,
     Descent, Landing);

  protected Mode_Gate is
    procedure Set_Mode (Mode: 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: 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;
Protected Objects: Requeuing and Private Queues

• 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: external tasks are only considered for queuing on barriers once no internally requeued task can be progressed any further!
• Private entries: Protected entries which are not accessible from outside the protected object, but can be employed as destinations for `requeue` operations.
Protected Objects: Requeuing

How to moderate the flow of incoming calls to a busy server farm?

type Urgency is
  (urgent, not_so_urgent);

type Server_Farm is
  (primary, secondary);

protected Pre_Filter is
  entry Reception (U : Urgency);

private
  entry Server (Server_Farm)
    (U : Urgency);
end Pre_Filter;
Protected Objects: Requeuing

protected body Pre_Filter is
  entry Reception (U : Urgency)
    when Server (primary)’count = 0 or else Server (secondary)’count = 0 is
    begin
      if U = urgent and then Server (primary)’count = 0 then
        requeue Server (primary);
      else
        requeue Server (secondary);
      end if;
    end Reception;
  entry Server (for S in Server_Farm) (U : Urgency) when True is
    begin null; -- a real server might actually handle the request
    end Server;
end Pre_Filter;
Restrictions on Protected Objects

All code inside a protected procedure, function or entry must be non-blocking operations. Thus the following operations are prohibited:

- entry calls
- delay statements
- task creations or activations
- select statements
- accept statements
- … as well as calls to sub-programs which contain any of the above

The requeue facility allows for a potentially blocking operation, and releases the current lock!