Australian National University

S02

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

C.A.R. Hoare (1974) *Monitors: an operating system structuring concept*, Communications of the ACM 17 (10)

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; …
```

```
…
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

```
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

```
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#

```
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).

S. Matsuoka and A. Yonezawa (1993). Analysis of inheritance anomaly in object-oriented concurrent programming languages.

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