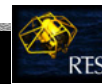


# RES

# 6

## Synchronization

Uwe R. Zimmer – The Australian National University



## References for this chapter

### [Ada95RM] (link to on-line version)

Ada Working Group  
ISO/IEC JTC1/SC 22/WG 9  
*Ada 95 Reference Manual*  
– *Language and Standard Libraries*  
ISO/IEC 8652:1995(E) with COR.1:2000,  
June 2001

### [Burns01]

Alan Burns and Andy Wellings  
*Real-Time Systems and Programming Languages*  
Addison Wesley, third edition, 2001

### [Cohen96]

Norman H. Cohen  
*Ada as a second language*  
McGraw-Hill series in computer science, 2nd  
edition

### [Bollella01]

Greg Bollella, Ben Brosgol, Steve Furr, David  
Hardin, Peter Dibble, James Gosling, Mark  
Turnbull & Rudy Belliard  
*The Real-Time Specification for Java*  
<http://www.rti.org>

all references and links are available on the course page



## Synchronization



## Synchronization methods

### • Shared memory based synchronization

- Semaphores
  - Conditional critical regions
  - Monitors
  - Mutexes & conditional variables
  - Synchronized methods
  - Protected objects
- ☞ 'C', POSIX – Dijkstra
  - ☞ Edison (experimental)
  - ☞ Modula-1, Mesa – Dijkstra, Hoare, ...
  - ☞ POSIX
  - ☞ Real-time Java
  - ☞ Ada95

### • Message based synchronization

- Asynchronous messages
  - Synchronous messages
  - Remote invocation, remote procedure call
  - Synchronization in distributed systems
- ☞ e.g. POSIX, ...
  - ☞ e.g. Ada95, CHILL, Occam2
  - ☞ e.g. Ada95, ...
  - ☞ e.g. CORBA, ...



## Synchronization



## Synchronization in real-time systems

☞ There are many concurrent entities in a real-time systems:

- Interrupt handlers
- Tasks
- Dispatchers
- Timers
- ...

... and ... real-time systems are often complex and is possibly expanded at a later stage ...

Thus *all* data is declared ...

- ☞ ... **either** local (and protected by language-, os-, or hardware-mechanisms)
- ☞ ... **or** it is 'out in the open' and all access need to be synchronized!



## Synchronization in real-time systems

Synchronization: the run-time overhead?

- ☞ Is the potential overhead justified for simple data-structures:

```

int i;
.....
i++; {in one thread} | i=0; {in another thread}
  
```

- Are those operations atomic?
- Do we really need to introduce full featured synchronization methods here?



## Synchronization in real-time systems

```

int i;
.....
i++; {in one thread} | i=0; {in another thread}
  
```

- Depending on the hardware and the compiler, it might be atomic, it might be not:
  - ☞ Handling a 64-bit integer on a 8- or 16-bit controller *will not be atomic* ... but perhaps it is an 8-bit integer.
  - ☞ Any manipulations on the main memory *will not be atomic* ... but perhaps it is a register.
  - ☞ Broken down to a load-operate-store cycle, the operations *will not be atomic* ... but perhaps the processor supplies atomic operations for the actual case.
- ☞ Assuming that all 'perhapses' are applying: how to expand this code?



## Synchronization in real-time systems

```

int i;
.....
i++; {in one thread} | i=0; {in another thread}
  
```

- ☞ Unfortunately: the chances that such programming errors turn out are usually small and some implicit by chance synchronization in the rest of the system might prevent them at all.
  - Many effects stemming from asynchronous memory accesses are interpreted as (hardware) 'glitches', since they are rare and effect usually only some parts of the data.
  - On assembler level: synchronization by employing knowledge about the atomicity of CPU-operations and interrupt structures is nevertheless possible and done frequently.
- In anything higher than assembler level on small, predictable µcontrollers:

☞ Measures for synchronization are required!



## Some synchronization terms:

- **Condition synchronization:**  
synchronize a task with an event given by another task.
- **Critical sections:**  
code fragments which contain access to shared resources and need to be executed without interference with other critical sections, sharing the same resources.
- **Mutual exclusion:**  
protection against asynchronous access to critical sections.
- **Atomic operations:**  
the set of operations, which atomicity is guaranteed by the underlying system (e.g. hardware).
  - ☞ there must be a set of atomic operations to start with!



### Synchronization by flags

Word-access atomicity:

Assuming that any access to a word in the system is an atomic operation:

e.g. assigning two values (not wider than the size of word) to a memory cell simultaneously:

Task 1: `x := 0;`      |      Task 2: `x := 5;`

will result in **either** `x = 0` **or** `x = 5` — and no other value is ever observable.



### Synchronization by flags

Assuming further that there is a shared memory area between two processes:

- A set of processes agree on a (word-size) atomic variable operating as a flag to indicate synchronization conditions.



### Condition synchronization by flags

`var Flag : boolean := false;`

```

process P1;
  statement X;
  repeat until Flag;
  statement Y;
end P1;

```

```

process P2;
  statement A;
  Flag := true;
  statement B;
end P2;

```

Sequence of operations: `[A | X] → [B | Y]`



### Synchronization by flags

Assuming further that there is a shared memory between two processes:

- A set of processes agree on a (word-size) atomic variable operating as a flag to indicate synchronization conditions:

Memory flag method is ok for simple condition synchronization, but ...

- ☞ ... is not sufficient for general mutual exclusion in critical sections!
- ☞ ... busy-waiting is required to poll the synchronization condition!

- ☞ More powerful synchronization operations are required for critical sections



## Synchronization

### Synchronization by semaphores

(Dijkstra 1968)

Assuming further that there is a shared memory between two processes:

- a set of processes agree on a variable  $S$  operating as a flag to indicate synchronization conditions ... and ...
  - an atomic operation  $P$  on  $S$  —  $P$  stands for 'passen' (Dutch for 'pass'):
    - $P$ : `[if  $S > 0$  then  $S := S - 1$ ]` also: 'Wait', 'Suspend\_Until\_True'
  - an atomic operation  $V$  on  $S$  —  $V$  stands for 'vrygeven' (Dutch for 'to release'):
    - $V$ : `[ $S := S + 1$ ]` also: 'Signal', 'Set\_True'
- ☞ the variable  $S$  is then called a **semaphore**.

OS-level:  $P$  is usually also suspending the current task until  $S > 0$ .

CPU-level:  $P$  indicates whether it was successful, but the operation is not blocking.

## Synchronization

### Condition synchronization by semaphores

```
var sync : semaphore := 0;
```

```
process P1;
  statement X;
  wait (sync);
  statement Y;
end P1;
```

```
process P2;
  statement A;
  signal (sync);
  statement B;
end P2;
```

Sequence of operations:  $[A | X] \mapsto [B | Y]$

## Synchronization

### Mutual exclusion by semaphores

```
var mutex : semaphore := 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;
```

Sequence of operations:  $[A | X] \mapsto [B \mapsto Y \text{ xor } Y \mapsto B] \mapsto [C | Z]$

## Synchronization

### Semaphores



Types of semaphores:

- **General semaphores (counting semaphores):** non-negative number; (range limited by the system)  $P$  and  $V$  increment and decrement the semaphore by one.
- **Binary semaphores:** restricted to  $[0, 1]$ ; Multiple  $V$  ( $Signal$ ) calls have the same effect than 1 call.
  - binary semaphores are sufficient to create all other semaphore forms.
  - atomic 'test-and-set' operations at hardware level are usually binary semaphores.
- **Quantity semaphores:** The increment (and decrement) value for the semaphore is specified as a parameter with  $P$  and  $V$ .



### Semaphores in Ada95

```
package Ada.Synchronous_Task_Control is
  type Suspension_Object is limited private;
  procedure Set_True (S : in out Suspension_Object);
  procedure Set_False (S : in out Suspension_Object);
  function Current_State (S : Suspension_Object) return Boolean;
  procedure Suspend_Until_True (S : in out Suspension_Object);
private
  ... -- not specified by the language
end Ada.Synchronous_Task_Control;
```

- only one task can be blocked at `Suspend_Until_True`! ('strict version of a binary semaphore')  
(`Program_Error` will be raised with the second task trying to suspend itself)
- ☞ no queues! ☞ minimal run-time overhead



### Semaphores in Ada95

```
package Ada.Synchronous_Task_Control is
  type Suspension_Object is limited private;
  procedure Set_True (S : in out Suspension_Object);
  procedure Set_False (S : in out Suspension_Object);
  function Current_State (S : Suspension_Object) return Boolean;
  procedure Suspend_Until_True (S : in out Suspension_Object);
private
  ... -- not specified by the language
end Ada.Synchronous_Task_Control;
```

- only one task can be blocked at `Suspend_Until_True`! ('strict version of a binary semaphore')  
(`Program_Error` will be raised with the second task trying to suspend itself)
- ☞ no queues ☞ minimal run-time overhead



### Semaphores in POSIX

```
int sem_init (sem_t *sem_location, int pshared, unsigned int value);
int sem_destroy (sem_t *sem_location);
int sem_wait (sem_t *sem_location);
int sem_trywait (sem_t *sem_location);
int sem_timedwait (sem_t *sem_location, const struct timespec *abstime);
int sem_post (sem_t *sem_location);
int sem_getvalue (sem_t *sem_location, int *value);
```

generate semaphore for usage between processes  
(otherwise for threads of the same process only)



### Semaphores in POSIX

```
int sem_init (sem_t *sem_location, int pshared, unsigned int value);
int sem_destroy (sem_t *sem_location);
int sem_wait (sem_t *sem_location);
int sem_trywait (sem_t *sem_location);
int sem_timedwait (sem_t *sem_location, const struct timespec *abstime);
int sem_post (sem_t *sem_location);
int sem_getvalue (sem_t *sem_location, int *value);
```

delivers the number of waiting processes as a negative integer,  
if there are processes waiting on this semaphore

## Synchronization

### Semaphores in POSIX

```

void allocate (priority_t P)
{
    sem_wait (&mutex);
    if (busy) {
        sem_post (&mutex);
        sem_wait (&cond[P]);
    }
    busy = 1;
    sem_post (&mutex);
}

void deallocate (priority_t P)
{
    sem_wait (&mutex);
    busy = 0;
    sem_getvalue (&cond[high],
                 &waiting);
    if (waiting < 0) {
        sem_post (&cond[high]);
    }
    else {
        sem_getvalue (&cond[low],
                     &waiting);
        if (waiting < 0) {
            sem_post (&cond[low]);
        }
        else {
            sem_post (&mutex);
        }
    }
}

sem_t mutex, cond[2];
typedef enum {low, high} priority_t;
int waiting
int busy
    
```

## Synchronization

### Deadlock by semaphores

```

with Ada.Synchronous_Task_Control; use Ada.Synchronous_Task_Control;
X, Y : Suspension_Object;

task B;
task body B is
begin
    ...
    Suspend_Until_True (Y);
    Suspend_Until_True (X);
    ...
end B;

task A;
task body A is
begin
    ...
    Suspend_Until_True (X);
    Suspend_Until_True (Y);
    ...
end A;
    
```

☞ could raise a `Program_Error` in Ada95.

☞ produces a potential **deadlock** when implemented with general semaphores.

☞ **Deadlocks can be generated by all kinds of synchronization methods.**

## Synchronization

### Criticism of semaphores

- Semaphores are not bound to any resource or method or region
  - ☞ Adding or deleting a single semaphore operation some place might stall the whole system
- Semaphores are scattered all over the code
  - ☞ hard to read, error-prone
- ☞ Semaphores are considered not adequate for the real-time domain.

(all concurrent and real-time languages offer more abstract and safer synchronization methods).

## Synchronization

### Conditional critical regions

Basic idea:

- Critical regions are a set of code sections in different processes, which are guaranteed to be **executed in mutual exclusion**:
  - Shared data structures are grouped in named regions and are tagged as being private resources.
  - Processes are prohibited from entering a critical region, when another process is active in any associated critical region.
- **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 false, the process is suspended / delayed.
- As with semaphores, no access order can be assumed.



## Synchronization

### Conditional critical regions

```

buffer : buffer_t;
resource critical_buffer_region : buffer;

```

```

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

```

```

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

```



## Synchronization

### Criticism of conditional critical regions

- All guards need to be re-evaluated, when any conditional critical region is left:
  - ↳ all involved processes are activated to test their guards
  - ↳ there is no order in the re-evaluation phase ↳ potential livelocks
- As with semaphores the conditional critical regions are scattered all over the code.
  - ↳ on a larger scale: same problems as with semaphores.

The language Edison uses conditional critical regions for synchronization in a multiprocessor environment (each process is associated with exactly one processor).



## Synchronization

### 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.
- Assure mutual exclusion of the monitor-procedures.



## Synchronization

### 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?



## 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.



## 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;
  ...
```



## 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!



## 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**.



## Synchronization

### 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`.



## Synchronization

### 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.
```



## Synchronization

### 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);
...
```



## Synchronization

### 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 (
  const
int pthread_mutex_destroy (
int pthread_cond_init (
  const
int pthread_cond_destroy (
  const
...
```

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
- ... ..



## Synchronization

### 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,
                        pthread_mutexattr_t *attr );
int pthread_mutex_destroy ( pthread_mutex_t *mutex );
int pthread_cond_init ( pthread_cond_t *cond,
                       pthread_condattr_t *attr );
int pthread_cond_destroy ( pthread_cond_t *cond );
...

```

Annotations:

- Red box: Undefined, if locked (points to pthread\_mutex\_init)
- Red box: Undefined, if threads are waiting (points to pthread\_cond\_init)



## Synchronization

### 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);

```



## Synchronization

### 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);

```

Annotations:

- Red box: unblocking 'at least one' thread (points to pthread\_cond\_wait)
- Red box: unblocking all threads (points to pthread\_cond\_broadcast)



## Synchronization

### 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);

```

Annotation:

- Red box: undefined, if called out of order! (points to pthread\_mutex\_unlock, pthread\_cond\_wait, and pthread\_cond\_timedwait)

## Synchronization

## 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);

```

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

## Synchronization

## Monitors in 'C' / POSIX

(example, definitions)

```

#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;

```

## Synchronization

## Monitors in 'C' / POSIX

(example, operations)

```

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;
}

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;
}

```

## Synchronization

## Monitors in Real-time 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.



### Monitors in Real-time Java

Considerations:

#### 1. 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 over the whole class.

Either in static synchronized blocks: `synchronized (this.getClass()) {...}`  
 or in static methods: `public synchronized static <method> {...}`



### Monitors in Real-time Java

Considerations:

#### 2. 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` does not release the lock.
  - ☞ methods, which are activated via notification need to wait for lock-access.
- `wait`-suspended threads are hold in a queue, thus `notify{All}` is waking the threads in order
  - ☞ livelocks are prevented at this level (in opposition to Java).
- There are no explicit conditional variables.
  - ☞ every notified thread needs to wait for the lock to be released **and** to re-evaluate its entry condition

### Monitors in Real-time 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



### Monitors in Real-time Java

(multiple-readers-one-writer-example: `wait-notifyAll` method)

`public class ReadersWriters`

```
{
    private int    readers      = 0;
    private int    waitingWriters = 0;
    private boolean writing     = false;
    ...
}
```



## Monitors in Real-time Java

(multiple-readers-one-writer-example: 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 ();
} ...
```



## Monitors in Real-time Java

(multiple-readers-one-writer-example: 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 thread are notified
  - all threads are re-evaluating their guards



## Monitors in Real-time Java

## Standard monitor solution:

- 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.



## Monitors in Real-time Java

(multiple-readers-one-writer-example: usage 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 ();

    ...
}
```



## Monitors in Real-time Java

```

... 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 ();
    } } ...

```



## Monitors in Real-time 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;
                }
            }
        }
    } } } } ...

```



## Monitors in Real-time 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 ();
    } } ...

```



## Monitors in Real-time Java

```

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

```



### 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.



### 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



### 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, Real-time 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



### 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.



## Synchronization by protected objects

Combine

- the **encapsulation** feature of monitors

with

- the **coordinated entries** of conditional critical regions

to

☞ 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)



## Synchronization by protected objects in Ada95

(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.



## Synchronization by protected objects in Ada95

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;
```



## Synchronization by protected objects in Ada95

(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;
```

## Synchronization

### Synchronization by protected objects in Ada95

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
  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 s.
end select;

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

```

## Synchronization

### Synchronization by protected objects in Ada95

(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*.

## Synchronization

### Synchronization by protected objects in Ada95

(operations on entry queues)

The count attribute indicate 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;

```

## Synchronization

### Synchronization by protected objects in Ada95

(operations on entry queues)

The count attribute indicate 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 Receive;

  procedure Send (M: in Message) is
  begin
    New_Message := M;
    Arrived := Receive'count > 0;
  end Send;
end Broadcast;

```



## Synchronization

### Synchronization by protected objects in Ada95

(entry families, *requeue* & private entries)

Further refinements on task control by:

- **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.

## Synchronization

### Synchronization by protected objects in Ada95

(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;
  
```

## Synchronization

### Synchronization by protected objects in Ada95

(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;
  
```

## Synchronization

### Synchronization by protected objects in Ada95

(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?

## Synchronization

### Synchronization by protected objects in Ada95

(restrictions applying to protected operations)

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!

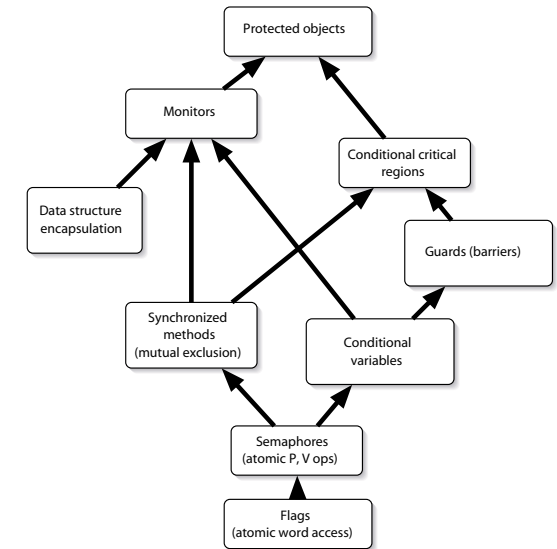
## Summary

### Shared memory based synchronization

#### General

Criteria:

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

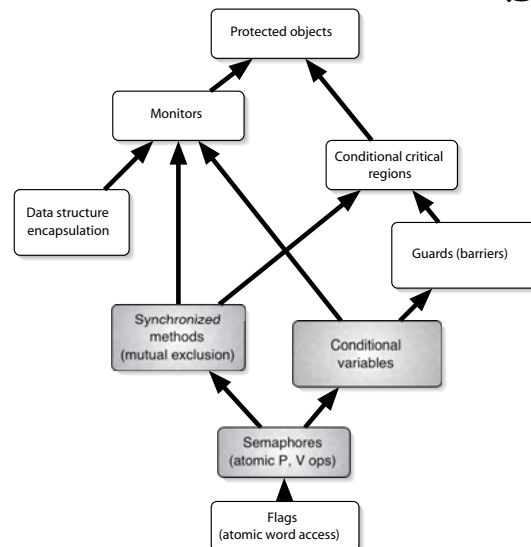


## Summary

### Shared memory based 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`).

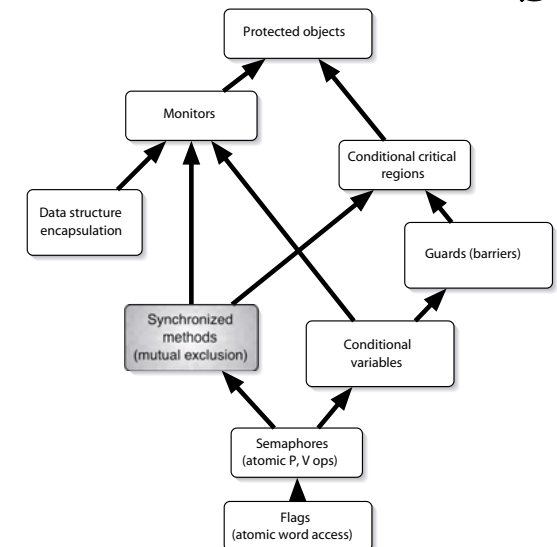


## Summary

### Shared memory based synchronization

#### Real-time 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.



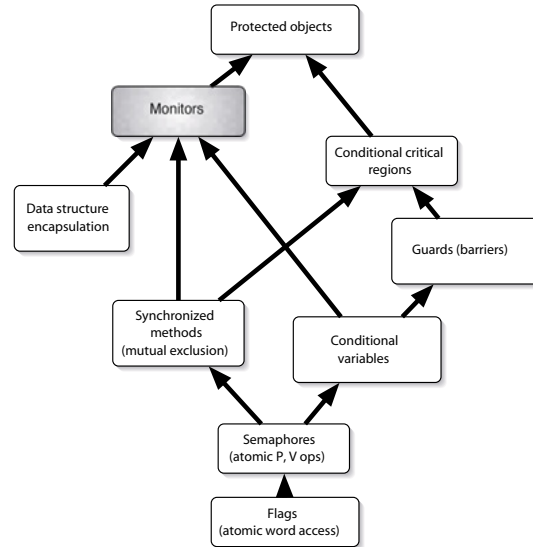


## Summary

### Shared memory based synchronization

#### Modula-1, CHILL

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

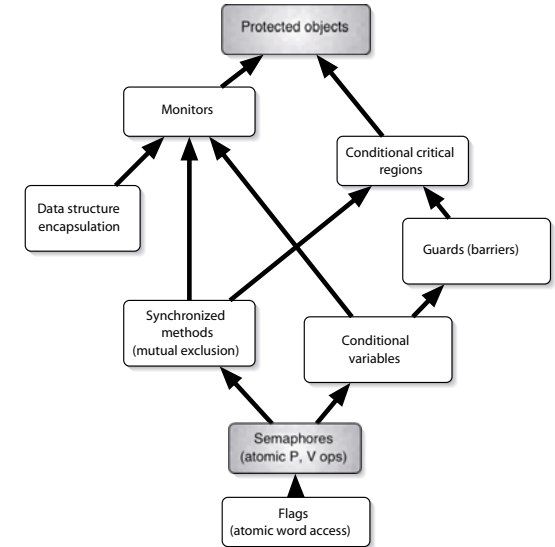


## Summary

### Shared memory based 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 than the programmer.
- (Ada95 is currently without any mainstream competitor in this field)



## Synchronization

### Message-based synchronization

- Synchronization model
  - Asynchronous
  - Synchronous
  - Remote invocation
- Addressing (name space)
  - direct communication
  - mail-box communication
- Message structure
  - arbitrary
  - restricted to 'basic' types
  - restricted to un-typed communications



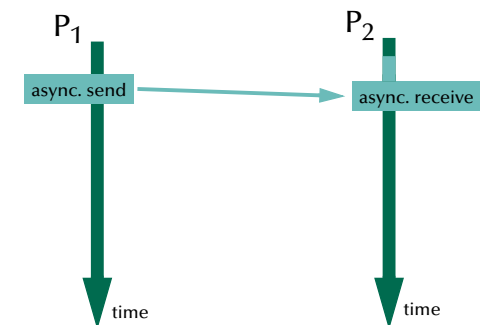
## Synchronization

### Message-based synchronization

#### Asynchronous messages

If there is a listener:

- ☞ send the message directly

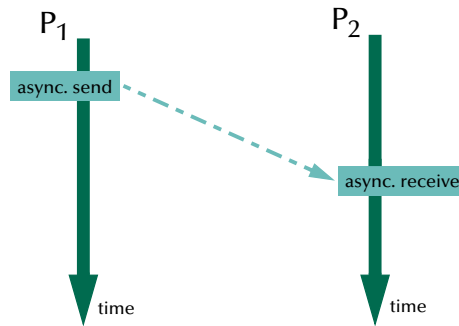




### Message-based synchronization

#### Asynchronous messages

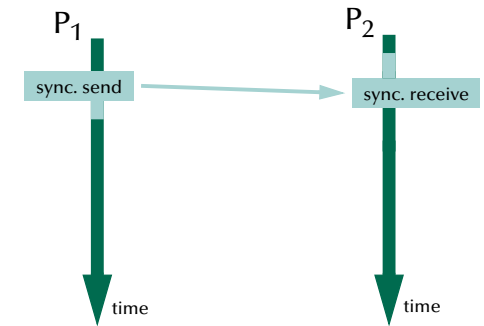
If the receiver becomes available at a later stage:  
 the message need to be buffered



### Message-based synchronization

#### Synchronous messages

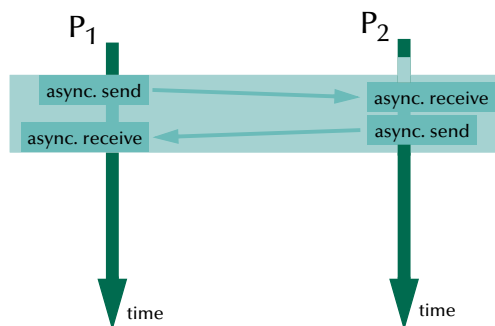
Delay the sender:  
 • until the receiver got the message



### Message-based synchronization

#### Synchronous messages

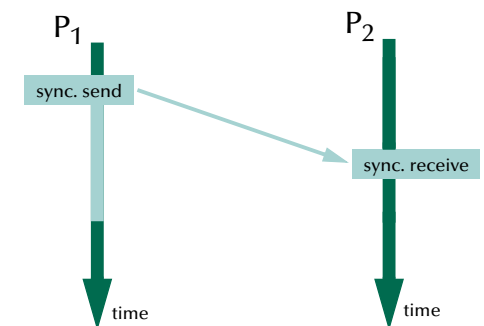
Delay the sender:  
 • until the receiver got the message  
 two asynchronous messages required



### Message-based synchronization

#### Synchronous messages

Delay the sender until:  
 • a receiver is available  
 • a receiver got the message



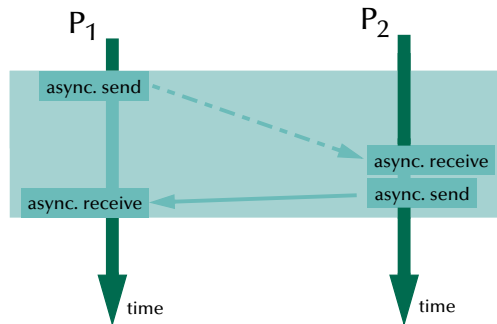


### Message-based synchronization

#### Synchronous messages

If the receiver becomes available at a later stage:

- ☞ messages need to be buffered

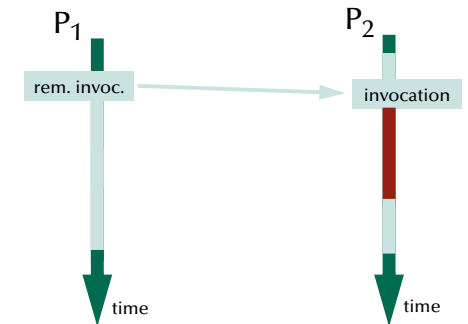


### Message-based synchronization

#### Remote invocation

Delay the sender, until:

- a receiver got the message
- a receiver executed an addressed routine

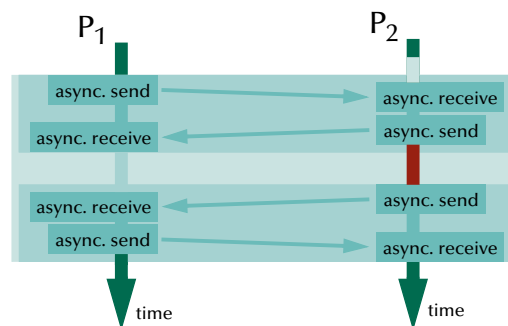


### Message-based synchronization

#### Remote invocation

Delay the sender, until:

- a receiver got the message
- a receiver executed an addressed routine

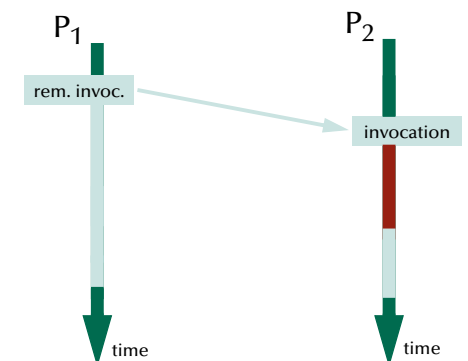


### Message-based synchronization

#### Remote invocation

Delay the sender, until:

- a receiver becomes available
- a receiver got the message
- a receiver executed an addressed routine



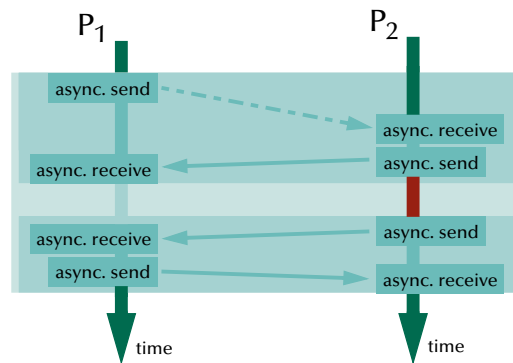


### Message-based synchronization

#### Remote invocation

Delay the sender, until:

- a receiver becomes available
- a receiver got the message
- a receiver executed an addressed routine

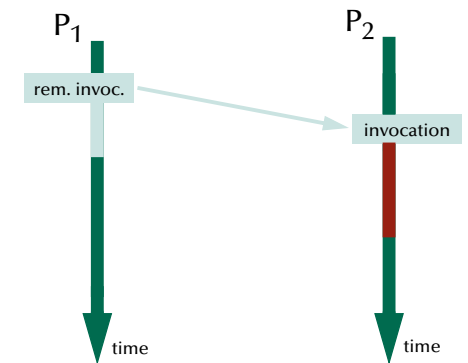


### Message-based synchronization

#### Asynchronous remote invocation

Delay the sender, until:

- a receiver becomes available
- a receiver got the message

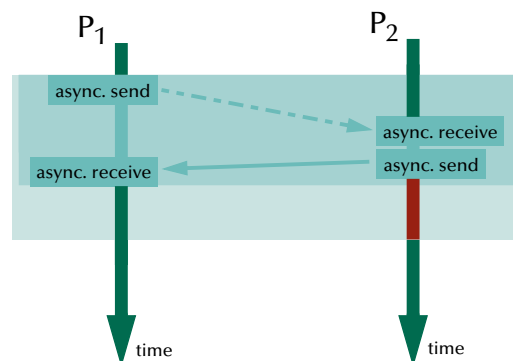


### Message-based synchronization

#### Asynchronous remote invocation

Delay the sender, until:

- a receiver becomes available
- a receiver got the message



### Synchronous vs. asynchronous communications

Purpose 'synchronization': ☞ synchronous messages / remote invocations

Purpose 'in-time delivery': ☞ asynchronous messages / asynchronous remote invocations

- ☞ 'Real' synchronous message passing in distributed systems requires hardware support.
- ☞ Asynchronous message passing requires the usage of (infinite?) buffers.

- Synchronous communications are emulated by a combination of asynchronous messages in some systems.
- Asynchronous communications can be emulated in synchronized message passing systems by introducing 'buffer-tasks' (de-coupling sender and receiver as well as allowing for broadcasts).



### Addressing (name space)

Direct vs. indirect:

```

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

```

Asymmetrical addressing:

```

send    <message> to ...
wait for <message>

```

☞ Client-server paradigm



### Addressing (name space)

Communication medium:

Connections	Functionality
one-to-one	buffer, queue, synchronization
one-to-many	multicast
one-to-all	broadcast
many-to-one	local server, synchronization
all-to-one	general server, synchronization
many-to-many	general network- or bus-system



### Message structure

- Machine dependent representations need to be taken care of in a distributed environment.
- Communication system is often outside the typed language environment.  
Most communication systems are handling streams (packets) of a basic element type only.

☞ Conversion routines for data-structures other than the basic element type are supplied ...

- ... manually (POSIX)
- ... semi-automatic (Real-time CORBA)
- ... automatic and are typed-persistent (Ada95)



### Message structure (Ada95)

```

package Ada.Streams is
  pragma Pure (Streams);
  type Root_Stream_Type is abstract tagged limited private;
  type Stream_Element is mod implementation-defined;
  type Stream_Element_Offset is range implementation-defined;
  subtype Stream_Element_Count is
    Stream_Element_Offset range 0..Stream_Element_Offset'Last;
  type Stream_Element_Array is
    array (Stream_Element_Offset range <>) of Stream_Element;
  procedure Read (...) is abstract;
  procedure Write (...) is abstract;

private
  ... -- not specified by the language
end Ada.Streams;

```



### Message structure (Ada95)

Reading and writing values of any type to a stream:

```

procedure S'Write(
  Stream : access Ada.Streams.Root_Stream_Type'Class; Item : in T);
procedure S'Class'Write(
  Stream : access Ada.Streams.Root_Stream_Type'Class; Item : in T'Class);
procedure S'Read(
  Stream : access Ada.Streams.Root_Stream_Type'Class; Item : out T);
procedure S'Class'Read(
  Stream : access Ada.Streams.Root_Stream_Type'Class; Item : out T'Class)
  
```

Reading and writing values, bounds and discriminants of any type to a stream:

```

procedure S'Output(
  Stream : access Ada.Streams.Root_Stream_Type'Class; Item : in T);
function S'Input(
  Stream : access Ada.Streams.Root_Stream_Type'Class) return T;
  
```



### Message-based synchronization

Practical message-passing systems:

POSIX:	“message queues”: ☞ <b>ordered indirect [asymmetrical   symmetrical] asynchronous byte-level many-to-many message passing</b>
CHILL:	“buffers”, “signals”: ☞ <b>ordered indirect [asymmetrical   symmetrical] [synchronous   asynchronous] typed [many-to-many   many-to-one] message passing</b>
Occam2:	“channels”: ☞ <b>indirect symmetrical synchronous fully-typed one-to-one message passing</b>
Ada95:	“(extended) rendezvous”: ☞ <b>ordered direct asymmetrical [synchronous   asynchronous] fully-typed many-to-one remote invocation</b>
Real-time Java:	no communication via messages available



### Message-based synchronization

Practical message-passing systems:

	ordered	symmetrical	asymmetrical	synchronous	asynchronous	direct	indirect	contents	one-to-one	many-to-one	many-to-many	method
POSIX:	*	*	*		*		*	bytes			*	message passing
CHILL:	*	*	*	*	*		*	typed		*	*	message passing
Occam2:		*		*			*	fully typed	*			message passing
Ada95:	*		*	*	*	*		fully typed		*		remote invocation
Real-time Java:	no communication via messages available											



### Message-based synchronization

Practical message-passing systems for synchronisation purposes:

	ordered	symmetrical	asymmetrical	synchronous	asynchronous	direct	indirect	contents	one-to-one	many-to-one	many-to-many	method
POSIX:	*	*	*		*		*	bytes			*	message passing
CHILL:	*	*	*	*	*		*	typed		*	*	message passing
Occam2:		*		*			*	fully typed	*			message passing
Ada95:	*		*	*	*	*		fully typed		*		remote invocation
Real-time Java:	no communication via messages available											



## Message-based synchronization in Occam2

Communication is ensured by means of a 'channel', which:

- can be used by one writer and one reader process only
- and is synchronous:

```
CHAN OF INT SensorChannel:
```

```
PAR
```

```
  INT reading:
```

```
  SEQ i = 0 FOR 1000
```

```
  SEQ
```

```
    -- generate reading
```

```
    SensorChannel ! reading
```

```
  INT data:
```

```
  SEQ i = 0 FOR 1000
```

```
  SEQ
```

```
    SensorChannel ? data
```

```
    -- employ data
```

tasks are synchronized  
at these points



## Message-based synchronization in CHILL

CHILL is the 'CCITT High Level Language',

where CCITT is the Comité Consultatif International Télégraphique et Téléphonique.

The CHILL language development was started in 1973 and standardized in 1979.

- ☞ strong support for concurrency, synchronization, and communication (monitors, buffered message passing, synchronous channels)

```
dc1 SensorBuffer buffer (32) int;
```

```
...
```

```
send SensorBuffer (reading);
```

```
receive case
```

```
  (SensorBuffer in data) : ...
```

```
esac;
```

```
signal SensorChannel = (int) to consumertype;
```

```
...
```

```
send SensorChannel (reading)
```

```
  to consumer
```

```
receive case
```

```
  (SensorChannel in data): ...
```

```
esac;
```



## Message-based synchronization in CHILL

CHILL is the 'CCITT High Level Language',

where CCITT is the Comité Consultatif International Télégraphique et Téléphonique.

The CHILL language development was started in 1973 and standardized in 1979.

- ☞ strong support for concurrency, synchronization, and communication (monitors, buffered message passing, synchronous channels)

```
dc1 SensorBuffer buffer (32) int;
```

```
...
```

```
send SensorBuffer (reading);
```

```
  --- asynchronous
```

```
receive case
```

```
  (SensorBuffer in data) : ...
```

```
esac;
```

```
signal SensorChannel = (int) to consumertype;
```

```
...
```

```
send SensorChannel (reading)
```

```
  to consumer
```

```
  ← synchronous
```

```
receive case
```

```
  (SensorChannel in data): ...
```

```
esac;
```



## Message-based synchronization in Ada95

Ada95 supports remote invocations ((extended) rendezvous) in form of:

- **entry points** in tasks
- **full set of parameter profiles** supported
  - If the local and the remote task are on different architectures, or if an intermediate communication system is employed:
- ☞ parameters incl. bounds and discriminants are 'tunnelled' through byte-stream-formats.

Synchronization:

- both tasks are synchronized at the beginning of the remote invocation (☞ 'rendezvous')
- the calling task is blocked until the remote routine is completed (☞ 'extended rendezvous')







## Selective waiting in Occam2

```
ALT
  Guard1
    Process1
  Guard2
    Process2
...
```

- Guards are referring to boolean expressions and/or channel input operations.
- The boolean expressions are local expressions, i.e. if none of them evaluates to true at the time of the evaluation of the ALT-statement, then the process is stopped.
- If all triggered channel input operations evaluate to false, the process is suspended until further activity on one of the named channels.
- Any Occam2 process can be employed in the ALT-statement
- The ALT-statement is non-deterministic (there is also a deterministic version: PRI ALT).



## Selective waiting in Occam2

```
ALT
  NumberInBuffer < Size & Append ? Buffer [Top]
    SEQ
      NumberInBuffer := NumberInBuffer + 1
      Top             := (Top + 1) REM Size
  NumberInBuffer > 0 & Request ? ANY
    SEQ
      Take ! Buffer [Base]
      NumberInBuffer := NumberInBuffer - 1
      Base           := (Base + 1) REM Size
```

- synchronization on input-channels only:
  - ☛ to initiate the sending of data (Take ! Buffer [Base]), a request need to be made first (Request ? ANY)

CSP (Hoare) also supports non-deterministic selective waiting



## Message-based selective synchronization in Ada95

Forms of selective waiting:

```
select_statement ::= selective_accept |
                  conditional_entry_call |
                  timed_entry_call |
                  asynchronous_select
... underlying concept: Dijkstra's guarded commands
```

selective\_accept implements ...

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



## Message-based selective synchronization in Ada95

selective\_accept in its full syntactical form in Ada95:

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

guard ::= when <condition> =>

selective_accept_alternative ::= accept_alternative |
                                delay_alternative |
                                terminate_alternative

accept_alternative ::= accept_statement [ sequence_of_statements ]
delay_alternative  ::= delay_statement [ sequence_of_statements ]
terminate_alternative ::= terminate;
```



## Synchronization

### Basic forms of selective synchronization

(select-or)

```

select
  accept ... do ...
end ...
or
  accept ... do ...
end ...
or
  accept ... do ...
end ...
or
  accept ... do ...
end ...
...
end select;

```

- If none of the named entries have been called, the task is suspended until one of the entries is addressed by another task.
- The selection of an accept is non-deterministic, in case that multiple entries are called.
- ☞ The selection can be controlled by means of the real-time systems annex.
- The select statement is completed, when at least one of the entries has been called and those accept-block has been executed.



## Synchronization

### Basic forms of selective synchronization

(guarded select-or)

```

select
  when <condition> =>
    accept ... do ...
end ...
or
  when <condition> =>
    accept ... do ...
end ...
or
  when <condition> =>
    accept ... do ...
end ...
...
end select;

```

- Analogue to Dijkstra's guarded commands
- all accepts closed will raise a Program\_Error
- ☞ set of conditions need to be complete



## Synchronization

### Basic forms of selective synchronization

(guarded select-or-else)

```

select
  [ when <condition> => ]
    accept ... do ...
end ...
or
  [ when <condition> => ]
    accept ... do ...
end ...
or
  [ when <condition> => ]
    accept ... do ...
end ...
else
  <statements>
...
end select;

```

- If none of the open entries can be accepted immediately, the else alternative is selected.
- There can be only one else alternative and it cannot be guarded.



## Synchronization

### Basic forms of selective synchronization

(guarded select-or-delay)

```

select
  [ when <condition> => ]
    accept ... do ...
end ...
or
  [ when <condition> => ]
    delay ...
    <statements>
or
  [ when <condition> => ]
    delay ...
    <statements>
...
end select;

```

- If none of the open entries has been called before the amount of time specified in the earliest open delay alternative, this delay alternative is selected.
- There can be multiple delay alternatives if more than one delay alternative expires simultaneously, either one may be chosen.
- `delay` and `delay until` can be employed.

## Synchronization

### Basic forms of selective synchronization

(guarded select-or-terminate)

```
select
  [ when <condition> => ]
    accept ... do ...
    end ...
or
  [ when <condition> => ]
    accept ... do ...
    end ...
or
  [ when <condition> => ]
    terminate;
...
end select;
```

The terminate alternative is chosen if none of the entries can ever be called again, i.e.:

- all tasks which can possibly call any of the named entries are terminated.
- or
- all remaining active tasks which can possibly call any of the named entries are waiting on selective terminate statements and none of their open entries can be called any longer.
- ☞ This task and all its dependent waiting-for-termination tasks are terminated together.

## Synchronization

### Basic forms of selective synchronization

(guarded select-or-else select-or-delay select-or-terminate)

```
select
  [ when <condition> => ]
    accept ... do ...
    end ...
or
  [ when <condition> => ]
    delay ...
    <statements>
...
end select;
select
  [ when <condition> => ]
    accept ... do ...
    end ...
or
  [ when <condition> => ]
    terminate;
...
end select;
```

```
select
  [ when <condition> => ]
    accept ... do ...
    end ...
or
  [ when <condition> => ]
    <statements>
...
end select;
```

else - delay - terminate alternatives cannot be mixed!

## Synchronization



### Non-determinism in selective synchronizations

- ☞ If equal alternatives are given, then the program correctness (incl. the timing specifications) must not be affected by the actual selection.
- If alternatives have different priorities, this can be expressed e.g. by means of the Ada real-time annex.
- Non-determinism in concurrent systems is or can be also introduced by:
  - non-ordered monitor or other queues
  - buffering / routing message passing systems
  - non-deterministic schedulers
  - timer quantization
  - ... any form of asynchronism

## Synchronization

### Conditional & timed entry-calls

```
conditional_entry_call ::=
  select
    entry_call_statement
    [sequence_of_statements]
  else
    sequence_of_statements
  end select;

timed_entry_call ::=
  select
    entry_call_statement
    [sequence_of_statements]
  or
    delay_alternative
  end select;

select
  Light_Monitor.Wait_for_Light;
  Lux := True;
else
  Lux := False;
end;
```

```
select
  Controller.Request (Medium)
  (Some_Item);
  -- process data
or
  delay 45.0;
  -- try something else
end select;
```

## Synchronization

### Conditional & timed entry-calls

```
conditional_entry_call ::=
  select
    entry_call_statement
    [sequence_of_statements]
  else
    sequence_of_statements
  end select;

timed_entry_call ::=
  select
    entry_call_statement
    [sequence_of_statements]
  or
    delay_alternative
  end select;

select
  Light_Monitor.Wait_for_Light;
  Lux := True;
else
  Lux := False;
end;
```

There is only **one entry call** and either one 'else' or one 'or delay'

## Synchronization

### Conditional & timed entry-calls

```
conditional_entry_call ::=
  select
    entry_call_statement
    [sequence_of_statements]
  else
    sequence_of_statements
  end select;

timed_entry_call ::=
  select
    entry_call_statement
    [sequence_of_statements]
  or
    delay_alternative
  end select;

select
  Light_Monitor.Wait_for_Light;
  Lux := True;
else
  Lux := False;
end;
```

The idea in both cases is to **withdraw a synchronization request** and **not** to implement polling or busy-waiting.

## Summary

### Synchronization

#### • Shared memory based synchronization

- Flags, condition variables, semaphores, ...  
... conditional critical regions, monitors, protected objects.
- Guard evaluation times, nested monitor calls, deadlocks, ...  
... simultaneous reading, queue management.
- Synchronization and object orientation, blocking operations and re-queuing.

#### • Message based synchronization

- Synchronization models, addressing modes, message structures
- Selective accepts, selective calls
- Indeterminism in message based synchronization