

Systems, Networks & Concurrency 2020



3

Communication & Synchronization

Uwe R. Zimmer - The Australian National University



Communication & Synchronization

References for this chapter

[Ben-Ari06]

M. Ben-Ari
Principles of Concurrent and Distributed Programming
2006, second edition, Prentice-Hall, ISBN 0-13-711821-X

[Barnes2006]

Barnes, John
Programming in Ada 2005
Addison-Wesley, Pearson education, ISBN-13 978-0-321-34078-8, Harlow, England, 2006

[Gosling2005]

Gosling, James, Joy, B, Steele, Guy & Bracha, Gilad
The Java™ Language Specification
- third edition
2005

[AdaRM2012]

Ada Reference Manual - Language and Standard Libraries;
ISO/IEC 8652:201x (E)

[Chapel 1.11.0 Language Specification Version 0.97]

see course pages or <http://chapel.cray.com/spec/spec-0.97.pdf> released on 2. April 2015

[Saraswat2010]

Saraswat, Vijay
Report on the Programming Language X10
Version 2.01
Draft — January 13, 2010



Communication & Synchronization

Overview

Synchronization methods

Shared memory based synchronization

- Semaphores
- Conditional critical regions
- Monitors
- Mutexes & conditional variables
- Synchronized methods
- Protected objects
- Atomic blocks

- ▣ C, POSIX — Dijkstra
- ▣ Edison (experimental)
- ▣ Modula-1, Mesa — Dijkstra, Hoare, ...
- ▣ POSIX
- ▣ Java, C#, ...
- ▣ Ada
- ▣ Chapel, X10

Message based synchronization

- Asynchronous messages
 - Synchronous messages
 - Remote invocation, remote procedure call
- ▣ e.g. POSIX, ...
 - ▣ e.g. Ada, CHILL, Occam2, ...
 - ▣ e.g. Ada, ...



Communication & Synchronization

Motivation

Side effects

Operations have side effects which are visible ...

either

▣ ... locally only

(and protected by runtime-, os-, or hardware-mechanisms)

or

▣ ... outside the current process

▣ If side effects transcend the local process then all forms of access need to be synchronized.



Communication & Synchronization

Sanity check

Do we need to? – really?

```
int i; {declare globally to multiple threads}
    i++;
    {in one thread}
    if i > n {i=0;}
    {in another thread}
```

What's the worst that can happen?



Communication & Synchronization

Sanity check

Do we need to? – really?

```
int i; {declare globally to multiple threads}
    i++;
    {in one thread}
    if i > n {i=0;}
    {in another thread}
```

- ☞ Handling a 64-bit integer on a 8- or 16-bit controller will not be atomic
 - ... yet perhaps it is an 8-bit integer.
- ☞ Unaligned manipulations on the main memory will usually not be atomic
 - ... yet perhaps it is a aligned.
- ☞ Broken down to a load-operate-store cycle, the operations will usually not be atomic
 - ... yet perhaps the processor supplies atomic operations for the actual case.
- ☞ Many schedulers interrupt threads irrespective of shared data operations
 - ... yet perhaps this scheduler is aware of the shared data.
- ☞ Local caches might not be coherent
 - ... yet perhaps they are.



Communication & Synchronization

Sanity check

Do we need to? – really?

```
int i; {declare globally to multiple threads}
    i++;
    {in one thread}
    if i > n {i=0;}
    {in another thread}
```

- ☞ Handling a 64-bit integer on a 8- or 16-bit controller will not be atomic

Unaligned manipulation on the main memory will usually not be atomic
 Broken down to a load-operate-store cycle, the operations will usually not be atomic
 Many schedulers interrupt threads irrespective of shared data operations

Even if all assumptions hold:

How to expand this code?

... yet perhaps the processor supplies atomic operations for the actual case.

... yet perhaps this scheduler is aware of the shared data.

- ☞ Local caches might not be coherent

... yet perhaps they are.



Communication & Synchronization

Sanity check

Do we need to? – really?

```
int i; {declare globally to multiple threads}
    i++;
    {in one thread}
    if i > n {i=0;}
    {in another thread}
```

- ☞ 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 usually rare, yet often disastrous.)

- ☞ On assembler level on very simple CPU architectures: synchronization by employing knowledge about the atomicity of CPU-operations and interrupt structures is nevertheless possible and utilized in practice.

In anything higher than assembler level on single core, predictable μ -controllers:

- ☞ **Measures for synchronization are required!**



Communication & Synchronization

Towards synchronization

Condition synchronization by flags

Assumption: word-access atomicity:

i.e. assigning two values (not wider than the size of a 'word') to an aligned memory cell concurrently:

$$x := 0 \quad | \quad x := 500$$

will result in either $x = 0$ or $x = 500$ – and no other value is ever observable



Communication & Synchronization

Towards synchronization

Condition 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:



Communication & Synchronization

Towards synchronization

Condition synchronization by flags

```

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

var Flag : boolean := false;

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

```

Sequence of operations: $A \rightarrow B; [X \mid A] \rightarrow Y; [X, Y \mid B]$



Communication & Synchronization

Towards synchronization

Condition 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:

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

☞ ... is not suitable 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



Communication & Synchronization

Basic synchronization by Semaphores

Basic definition (Dijkstra 1968)

Assuming the following three conditions on a shared memory cell between processes:

- a set of processes agree on a variable S operating as a flag to indicate synchronization conditions
- an atomic operation P on S — for 'passeren' (Dutch for 'pass'):
 $P(S): [as\ soon\ as\ S > 0\ then\ S := S - 1]$ \Rightarrow this is a potentially delaying operation
 aka: 'Wait', 'Suspend_Until_True', 'sem_wait', ...
- an atomic operation V on S — for 'vrygeven' (Dutch for 'to release'):
 $V(S): [S := S + 1]$
 aka 'Signal', 'Set-True', 'sem_post', ...

\Rightarrow then the variable S is called a **Semaphore**.



Communication & Synchronization

Towards synchronization Condition synchronization by semaphores

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

```
process P1;                               process P2;
  statement X;                             statement A;
  wait (sync);                             signal (sync);
  statement Y;                             statement B;
end P1;                                    end P2;
```

Sequence of operations: $A \rightarrow B; [X | A] \rightarrow Y; [X, Y | B]$



Communication & Synchronization

Towards synchronization Mutual exclusion by semaphores

```
var mutex : semaphore := 1;
```

```
process P1;                               process P2;
  statement X;                             statement A;
  wait (mutex);                             wait (mutex);
  statement Y;                             statement B;
  signal (mutex);                          signal (mutex);
  statement Z;                             statement C;
end P1;                                    end P2;
```

Sequence of operations:

$A \rightarrow B \rightarrow C; X \rightarrow Y \rightarrow Z; [X, Z | A, B, C]; [A, C | X, Y, Z]; \neg[B | Y]$



Communication & Synchronization


Towards synchronization Semaphores in Ada

```
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;
```

\Rightarrow This is "queueless" and can translate into a single machine instruction.

only one task can be blocked at Suspend_Until_True!
 (Program_Error will be raised with a second task trying to suspend itself)

\Rightarrow no queues! \Rightarrow minimal run-time overhead



Communication & Synchronization

Towards synchronization

Semaphores in Ada

```


package Ada.Synchronous_Task_Control is
  type Suspension_Object is limited private;
  procedure Set_True (S : in out Suspension_Object);
  for special cases only ... otherwise:
  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!
(Program_Error will be raised with a second task trying to suspend itself)

no queues! minimal run-time overhead

© 2020 Uwe R. Zimmer, The Australian National University page 270 of 758 (chapter 3: "Communication & Synchronization" up to page 369)



Communication & Synchronization

Towards synchronization

Malicious use of "queueless semaphores"

```

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


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

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

```

Could raise a Program_Error as multiple tasks potentially suspend on the same semaphore
(occurs only with high efficiency semaphores which do not provide process queues)

© 2020 Uwe R. Zimmer, The Australian National University page 271 of 758 (chapter 3: "Communication & Synchronization" up to page 369)



Communication & Synchronization

Towards synchronization

Malicious use of "queueless 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);
  Set_True (X);
  ...
end B;

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

```

Will result in a deadlock (assuming no other Set_True calls)

© 2020 Uwe R. Zimmer, The Australian National University page 272 of 758 (chapter 3: "Communication & Synchronization" up to page 369)



Communication & Synchronization

Towards synchronization

Malicious use of "queueless 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;

```

Will potentially result in a deadlock (with general semaphores)
or a Program_Error in Ada.

© 2020 Uwe R. Zimmer, The Australian National University page 273 of 758 (chapter 3: "Communication & Synchronization" up to page 369)

Communication & Synchronization

Towards synchronization

Semaphores in POSIX

pshared is actually a Boolean indicating whether the semaphore is to be shared between processes

```
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);
```

*value indicates the number of waiting processes as a negative integer in case the semaphore value is zero

© 2020 Uwe R. Zimmer, The Australian National University page 274 of 758 (chapter 3: "Communication & Synchronization" up to page 369)

Communication & Synchronization

Towards synchronization

Semaphores in POSIX

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

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]);
        }
    }
    sem_post (&mutex);
} }
```

Deadlock?
Livelock?
Mutual exclusion?

© 2020 Uwe R. Zimmer, The Australian National University page 275 of 758 (chapter 3: "Communication & Synchronization" up to page 369)

Communication & Synchronization

Towards synchronization

Semaphores in Java (since 2004)

```
Semaphore (int permits, boolean fair)
void acquire ()
void acquire (int permits)
void acquireUninterruptibly (int permits)
boolean tryAcquire ()
boolean tryAcquire (int permits, long timeout, TimeUnit unit)
int availablePermits ()
protected void reducePermits (int reduction)
int drainPermits ()
void release ()
void release (int permits)
protected Collection <Thread> getQueuedThreads ()
int getQueueLength ()
boolean hasQueuedThreads ()
boolean isFair ()
String toString ()
```

wait

check and manipulate

signal

administration

© 2020 Uwe R. Zimmer, The Australian National University page 276 of 758 (chapter 3: "Communication & Synchronization" up to page 369)


Communication & Synchronization

Towards synchronization

Review of semaphores

- Semaphores are **not bound to any resource or method or region**
 - ☞ Compiler has no idea what is supposed to be protected by a semaphore.
- Semaphores are **scattered all over the code**
 - ☞ Hard to read and highly error-prone.
 - ☞ Adding or deleting a single semaphore operation usually stalls a whole system.
- ☞ Semaphores are generally considered inadequate for non-trivial systems.
 - (all concurrent languages and environments offer efficient and higher-abstraction synchronization methods)
 - ☞ Special (usually close-to-hardware) applications exist.

© 2020 Uwe R. Zimmer, The Australian National University page 277 of 758 (chapter 3: "Communication & Synchronization" up to page 369)



Communication & Synchronization


Distributed synchronization

Conditional Critical Regions

Basic idea:

- Critical regions are a *set of associated 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 to false, the process is suspended / delayed.
- Generally, no access order can be assumed ☞ potential livelocks

© 2020 Uwe R. Zimmer, The Australian National University page 278 of 758 (chapter 3: "Communication & Synchronization" up to page 369)



Communication & Synchronization

Distributed 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;

```

© 2020 Uwe R. Zimmer, The Australian National University page 279 of 758 (chapter 3: "Communication & Synchronization" up to page 369)



Communication & Synchronization


Distributed synchronization

Review of Conditional Critical Regions

- Well formed synchronization blocks and synchronization conditions.
- Code, data and synchronization primitives are associated (known to compiler and runtime).
- 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
- Condition synchronisation inside the critical code sections requires to leave and re-enter a critical region.
- As with semaphores the conditional critical regions are distributed all over the code.
 - ☞ on a larger scale: same problems as with semaphores.

(The language Edison (Per Brinch Hansen, 1981) uses conditional critical regions for synchronization in a multiprocessor environment (each process is associated with exactly one processor).)

© 2020 Uwe R. Zimmer, The Australian National University page 280 of 758 (chapter 3: "Communication & Synchronization" up to page 369)



Communication & Synchronization

Centralized 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 and functions.
- Assure mutual exclusion of all monitor-procedures and functions.

© 2020 Uwe R. Zimmer, The Australian National University page 281 of 758 (chapter 3: "Communication & Synchronization" up to page 369)

Communication & Synchronization

Centralized 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 implement
conditional synchronization?

© 2020 Uwe R. Zimmer, The Australian National University page 282 of 758 (chapter 3: "Communication & Synchronization" up to page 369)

Communication & Synchronization

Centralized synchronization

Monitors with condition synchronization

(Hoare '74)

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.

© 2020 Uwe R. Zimmer, The Australian National University page 283 of 758 (chapter 3: "Communication & Synchronization" up to page 369)

Communication & Synchronization

Centralized synchronization

Monitors with condition synchronization

```

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

```

© 2020 Uwe R. Zimmer, The Australian National University page 284 of 758 (chapter 3: "Communication & Synchronization" up to page 369)

Communication & Synchronization

Centralized synchronization

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!

© 2020 Uwe R. Zimmer, The Australian National University page 285 of 758 (chapter 3: "Communication & Synchronization" up to page 369)



Communication & Synchronization

Centralized synchronization

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.



Communication & Synchronization

Centralized synchronization

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.



Communication & Synchronization

Centralized 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.
```



Communication & Synchronization

Centralized synchronization

Monitors in POSIX ('C')

(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 );
...
```



Communication & Synchronization

Centralized synchronization

Monitors in POSIX ('C')

(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);
...
```

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
- ...



Communication & Synchronization

Centralized synchronization

Monitors in POSIX ('C')

(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);
```

pthread_cond_wait
pthread_cond_timedwait
pthread_cond_signal
pthread_cond_broadcast
unlocks 'at least one' thread

pthread_mutex_lock
pthread_mutex_trylock
pthread_mutex_timedlock
unlocks all threads



Communication & Synchronization

Centralized synchronization

Monitors in POSIX ('C')

(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);
...
```

Undefined while locked

Undefined while threads are waiting



Communication & Synchronization

Centralized synchronization

Monitors in POSIX ('C')

(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);
```

pthread_mutex_unlock
pthread_cond_wait
pthread_cond_timedwait
pthread_mutex_t *mutex; if called 'out of order' pthread_mutex_t *mutex, i.e. mutex is not locked pthread_mutex_t *mutex;



Communication & Synchronization

Centralized synchronization

Monitors in POSIX (C)

```

...
pthread_mutex_lock
pthread_mutex_trylock
pthread_mutex_timedlock
pthread_mutex_unlock
pthread_cond_wait
pthread_cond_timedwait
pthread_cond_signal
pthread_cond_broadcast
  
```

(operators)

```

pthread_mutex_t *mutex;
pthread_mutex_t *mutex;
pthread_mutex_t *mutex;
const struct timespec *abstime);
pthread_mutex_t *mutex);
pthread_cond_t *cond,
pthread_mutex_t *mutex);
pthread_cond_t *cond,
const struct timespec *abstime);
pthread_cond_t *cond);
pthread_cond_t *cond);
  
```

can be called

- any time
- anywhere
- multiple times



Communication & Synchronization

Centralized synchronization

```

#define BUFF_SIZE 10
typedef struct { pthread_mutex_t mutex;
pthread_cond_t buffer_not_full;
pthread_cond_t buffer_not_empty;
int count, first, last;
int buf [BUFF_SIZE],
} buffer;

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



Communication & Synchronization

Centralized synchronization

```

#define BUFF_SIZE 10
typedef struct { pthread_mutex_t mutex;
pthread_cond_t buffer_not_full;
pthread_cond_t buffer_not_empty;
int count, first, last;
int buf [BUFF_SIZE],
} buffer;

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

need to be called with a locked mutex

better to be called after unlocking all mutexes (as it is itself potentially blocking)



Communication & Synchronization

Centralized synchronization

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);
}
}
  
```



Communication & Synchronization

Centralized synchronization

Monitors in Visual C++

```
using namespace System;
using namespace System.Threading;
private: integer data_to_protect;

void Reader()
{ try {
    Monitor::Enter (data_to_protect);
    Monitor::Wait (data_to_protect);
    ... read out protected data
}
finally {
    Monitor::Exit (data_to_protect);
}
};

void Writer()
{ try {
    Monitor::Enter (data_to_protect);
    ... write protected data
    Monitor::Pulse (data_to_protect);
}
finally {
    Monitor.Exit (data_to_protect);
}
};
```



Communication & Synchronization

Centralized synchronization

Monitors in Visual Basic

```
Imports System
Imports System.Threading
Private Dim data_to_protect As Integer = 0

Public Sub Reader
    Try
        Monitor.Enter (data_to_protect)
        Monitor.Wait (data_to_protect)
        ... read out protected data
    Finally
        Monitor.Exit (data_to_protect)
    End Try
End Sub

Public Sub Writer
    Try
        Monitor.Enter (data_to_protect)
        ... write protected data
        Monitor.Pulse (data_to_protect)
    Finally
        Monitor.Exit (data_to_protect)
    End Try
End Sub
```



Communication & Synchronization

Centralized synchronization

Monitors in Java

```
Monitor mon = new Monitor();
Monitor.Condition Condvar = mon.new Condition();

public void reader
throws InterruptedException {
    mon.enter();
    Condvar.await();
    ... read out protected data
    mon.leave();
}

public void writer
throws InterruptedException {
    mon.enter();
    ... write protected data
    Condvar.signal();
    mon.leave();
}
```

... the Java library monitor connects data or condition variables to the monitor by convention only!



Communication & Synchronization

Centralized synchronization

Monitors in Java

(by means of language primitives)

Java provides two mechanisms to construct a monitors-like structure:

- **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.



Communication & Synchronization

Centralized synchronization

Monitors in Java

(by means of language primitives)

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 a Java monitor and enter at any time.
 - Methods outside the monitor-object can synchronize at this object.
 - ☞ it is impossible to analyse a Java 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.

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



Communication & Synchronization

Centralized synchronization

Monitors in Java

(by means of language primitives)

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 *adequate* 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 or interfering with this monitor-object in any way ☞ by convention.



Communication & Synchronization

Centralized synchronization

Monitors in Java

(by means of language primitives)

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` do not release the lock!
 - Java 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 associated with the monitor or data.
 - ☞ notified threads need to wait for the lock to be released **and** to re-evaluate its entry condition.



Communication & Synchronization

Centralized synchronization

Monitors in 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 ();
    ...
}
```



Communication & Synchronization

Centralized synchronization

Monitors in Java

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



Communication & Synchronization

Centralized synchronization

Monitors in Java

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



Communication & Synchronization

Centralized synchronization

Monitors in Java

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



Communication & Synchronization

Centralized synchronization

Monitors in Java

(multiple-readers-one-writer-example: usage of external conditional variables)

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




Communication & Synchronization

Centralized synchronization Criticism of monitors

- Mutual exclusion is solved elegantly and safely.
- Conditional synchronization is on the level of semaphores still
 - ☞ all criticism about semaphores applies inside the monitors
- ☞ Mixture of low-level and high-level synchronization constructs.



Communication & Synchronization

Centralized synchronization

Synchronization by protected objects

- Combine
- with
- the encapsulation feature of monitors
- the coordinated entries of conditional critical regions
- to:
- ☞ Protected objects
 - All controlled data and operations are **encapsulated**.
 - Operations are **mutual 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 their 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).



Communication & Synchronization

Centralized synchronization

Synchronization by protected objects

(Simultaneous read-access)

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

```
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 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 Ada.



Communication & Synchronization

Centralized synchronization

Synchronization by protected objects

(Condition synchronization: entries & barriers)

Condition synchronization is realized in the form of **protected procedures** combined with boolean predicates (**barriers**): ☞ 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;
```




Communication & Synchronization

Centralized synchronization

Synchronization by protected objects

(Condition synchronization: entries & 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 : Data_Item) when Num < Buffer_Size is
    begin
      Last := Last + 1;
      Buffer (Last) := Item;
      Num := Num + 1;
    end Put;
end Bounded_Buffer;
```

© 2020 Uwe R. Zimmer, The Australian National University

page 318 of 758 (chapter 3: "Communication & Synchronization" up to page 369)



Communication & Synchronization

Centralized synchronization

Synchronization by protected objects

(Withdrawing entry calls)

```
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;
```

© 2020 Uwe R. Zimmer, The Australian National University

page 319 of 758 (chapter 3: "Communication & Synchronization" up to page 369)



Communication & Synchronization

Centralized synchronization

Synchronization by protected objects

(Withdrawing entry calls)

```
Buffer : Bounded_Buffer;

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

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

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

© 2020 Uwe R. Zimmer, The Australian National University

page 320 of 758 (chapter 3: "Communication & Synchronization" up to page 369)



Communication & Synchronization

Centralized synchronization

Synchronization by protected objects

(Barrier evaluation)

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

- on *creating a protected object*, all barrier 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.

Alternatively an implementation may choose to evaluate barriers on those two occasions:

- on *calling a protected entry*, the one associated barrier is evaluated.
- on *leaving a protected procedure or entry*, all potentially altered barriers with tasks queued up on them are re-evaluated.

Barriers are not evaluated *while inside* a protected object or on *leaving a protected function*.

© 2020 Uwe R. Zimmer, The Australian National University

page 321 of 758 (chapter 3: "Communication & Synchronization" up to page 369)



Communication & Synchronization

Centralized synchronization

Synchronization by protected objects

(Operations on entry queues)

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

```
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;
```



Communication & Synchronization

Centralized synchronization

Synchronization by protected objects

(Entry families, request & private entries)

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.
- **Request facility:**
Protected operations can use 'request' 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 requested 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 request operations.



Communication & Synchronization

Centralized synchronization

Synchronization by protected objects

(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: Message);
private
  New_Message : Message;
  Arrived : Boolean := False;
end Broadcast;

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;
```



Communication & Synchronization

Centralized synchronization

Synchronization by protected objects

(Entry families)

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



Communication & Synchronization

Centralized synchronization

Synchronization by protected objects

(Entry families, `request` & private entries)

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;

```



Communication & Synchronization

Centralized synchronization

Synchronization by protected objects

(Entry families, `request` & private entries)

```

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
      request Server (primary);
    else
      request Server (secondary);
    end if;
  end Reception;
  entry Server (for S in Server_Farm) (U : Urgency) when True is
  begin null; -- might try something even more useful
  end Server;
end Pre_Filter;

```



Communication & Synchronization

Centralized synchronization

Synchronization by protected objects

(Restrictions for protected operations)

All code inside a protected procedure, function or entry is bound to non-blocking operations.

Thus the following operations are prohibited:

- entry call statements
 - delay statements
 - task creations or activations
 - select statements
 - accept statements
 - ... as well as calls to sub-programs which contain any of the above
- ⚠ The `request` facility allows for a potentially blocking operation, and releases the current lock!



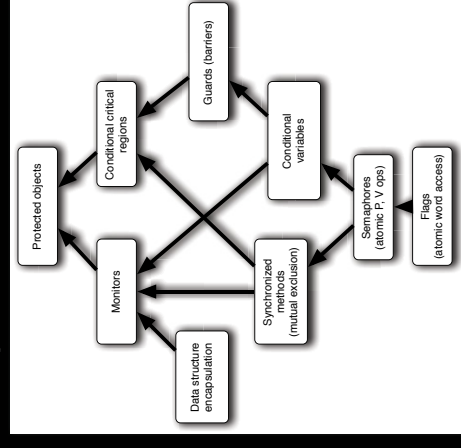
Communication & Synchronization

Shared memory based synchronization

General

Criteria:

- Levels of abstraction
- Centralized versus distributed
- Support for automated (compiler based) consistency and correctness validation
- Error sensitivity
- Predictability
- Efficiency



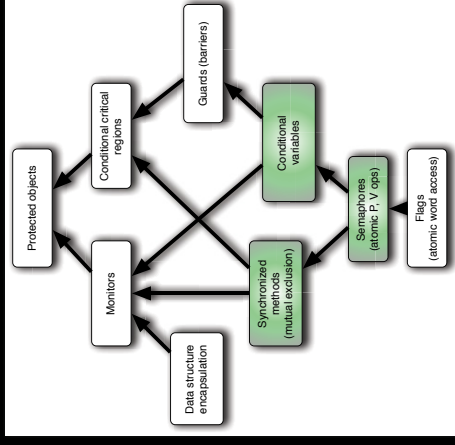


Communication & Synchronization

Shared memory based synchronization

POSIX

- All low level constructs available
- Connection with the actual data-structures by means of convention only
- Extremely error-prone
- Degree of non-determinism introduced by the 'release some' semantic
- 'C'-based
- Portable

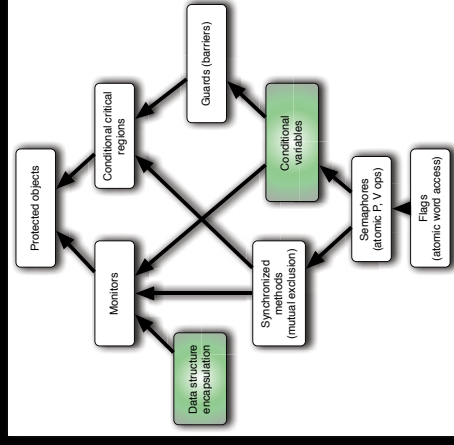


Communication & Synchronization

Shared memory based synchronization

C#, Visual C++, Visual Basic

- Mutual exclusion via library calls (convention)
- Data is associated with the locks to protect it
- Condition variables related to the data protection locks
- Mixture of high-level object oriented features and low level concurrency primitives

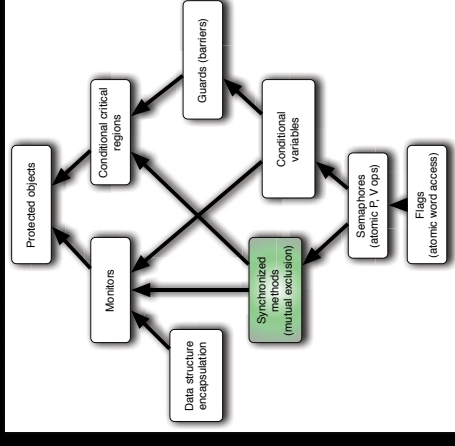


Communication & Synchronization

Shared memory based synchronization

Java

- Mutual exclusion available.
- General notification feature (not connected to other locks, hence not a conditional variable)
- Universal object orientation makes local analysis hard or even impossible
- Mixture of high-level object oriented features and low level concurrency primitives

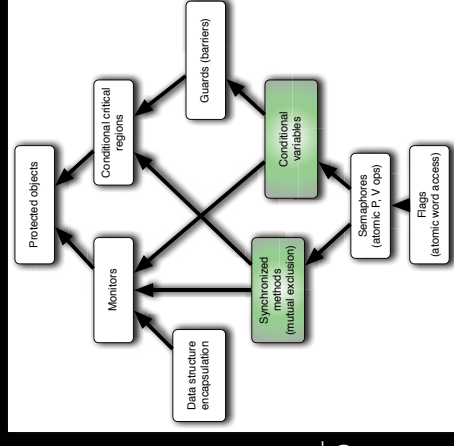


Communication & Synchronization

Shared memory based synchronization

C++14

- Mutual exclusion in scopes
- Data is not strictly associated with the locks to protect it
- Condition variables related to the mutual exclusion locks
- Set of essential primitives without combining them in a syntactically strict form (yet?)



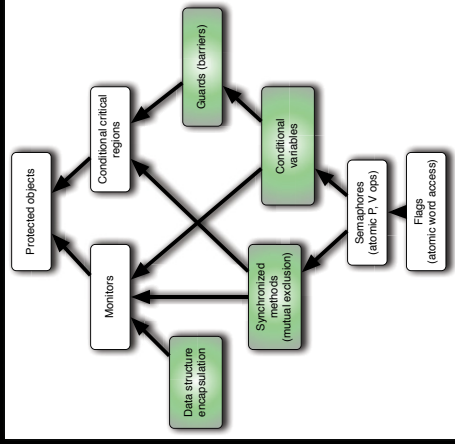


Communication & Synchronization

Shared memory based synchronization

Rust

- *Mutual exclusion in scopes*
- *Data is strictly associated with locks to protect it*
- *Condition variables related to the mutual exclusion locks*
- *Combined with the message passing semantics already a power set of tools.*
- *Concurrency features migrated to a standard library.*

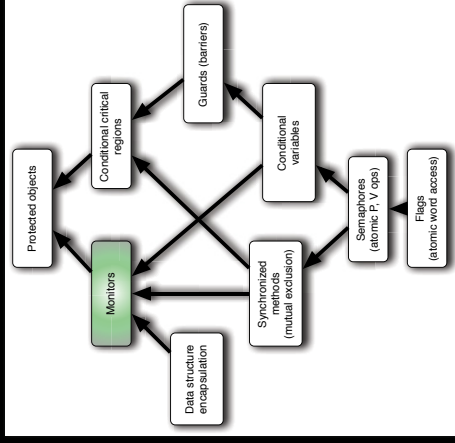


Communication & Synchronization

Shared memory based synchronization

Modula-1, Chill, Parallel Pascal, ...

- *Full implementation of the Dijkstra /Hoare monitor concept*
- *The term monitor appears in many other concurrent languages, yet it is usually not associated with an actual language primitive.*

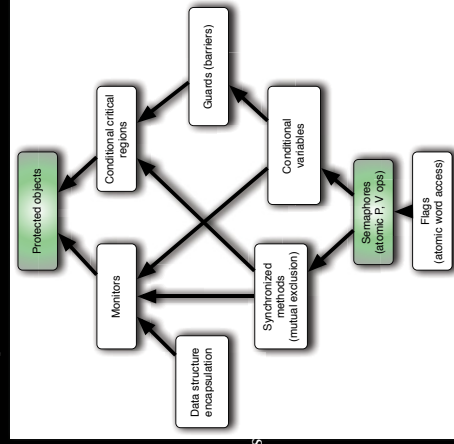


Communication & Synchronization

Shared memory based synchronization

Ada

- *High-level synchronization support which scales to large size projects.*
 - *Full compiler support incl. potential deadlock analysis*
 - *Low-Level semaphores for very special cases*
- Ada has still
no mainstream competitor
in the field of explicit concurrency.
(2018)



Communication & Synchronization

High Performance Computing

Synchronization in large scale concurrency

- High Performance Computing (HPC) emphasizes on keeping as many CPU nodes busy as possible:
- Avoid contention on sparse resources.
 - Data is assigned to individual processes rather than processes synchronizing on data.
 - Data integrity is achieved by keeping the CPU nodes in approximate "lock-step", yet there is still a need to re-sync concurrent entities.
- Traditionally this has been implemented using the Message Passing Interface (MPI) while implementing separate address spaces.
- Current approaches employ partitioned address spaces, i.e. memory spaces can overlap and be re-assigned. (e.g. Chapel, Fortress, X10).
 - Not all algorithms break down into independent computation slices and so there is a need for memory integrity mechanisms in shared/partitioned address spaces.



Communication & Synchronization

Current developments

Atomic operations in X10

- X10 offers only atomic blocks in unconditional and conditional form.
- Unconditional atomic blocks are guaranteed to be non-blocking, which means that they cannot be nested and need to be implemented using roll-backs.
 - Conditional atomic blocks can also be used as a pure notification system (similar to the Java notify method).
 - Parallel statements (incl. parallel, i.e. unrolled 'loops').
 - Shared variables (and their access mechanisms) are not defined.
 - The programmer does not specify the scope of the locks (atomic blocks) but they are managed by the compiler/runtime environment.
 - Code analysis algorithms are required in order to provide efficiently, otherwise the runtime environment needs to associate every atomic block with a *global* lock.



Communication & Synchronization

Current developments

Synchronization in Chapel

Chapel offers a variety of concurrent primitives:

- Parallel operations on data (e.g. concurrent array operations)
- Parallel statements (incl. parallel, i.e. unrolled 'loops')
- Parallelism can also be explicitly limited by serializing statements
- Atomic blocks for the purpose to construct atomic transactions
- Memory integrity needs to be programmed by means of synchronization statements (waiting for one or multiple control flows to complete) and/or atomic blocks

Further Chapel semantics are still forthcoming ... so there is still hope for a stronger shared memory synchronization / memory integrity construct.



Communication & Synchronization

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



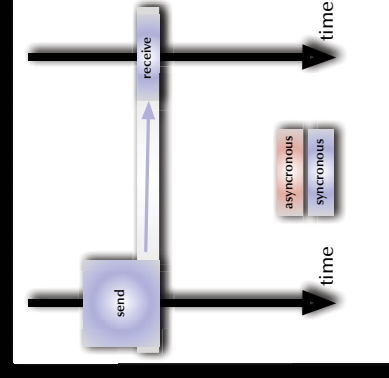
Communication & Synchronization

Message-based synchronization

Message protocols

Synchronous message (sender waiting)

- Delay the sender process until
- Receiver becomes available
 - Receiver acknowledges reception



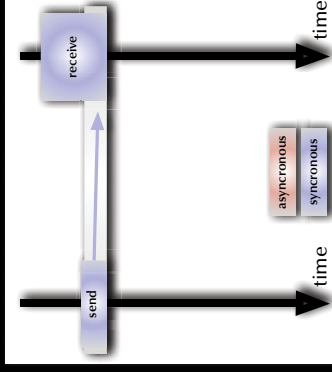


Communication & Synchronization

Message-based synchronization Message protocols

Synchronous message (receiver waiting)

- Delay the receiver process until
- Sender becomes available
- Sender concludes transmission



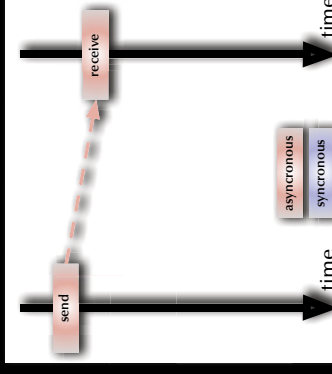
Communication & Synchronization

Message-based synchronization Message protocols

Asynchronous message

Neither the sender nor the receiver is blocked:

- Message is not transferred directly
- A buffer is required to store the messages
- Policy required for buffer sizes and buffer overflow situations



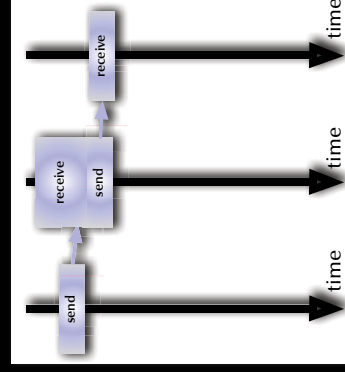
Communication & Synchronization

Message-based synchronization Message protocols

Asynchronous message (simulated by synchronous messages)

Introducing an intermediate process:

- Intermediate needs to be accepting messages at all times.
 - Intermediate also needs to send out messages on request.
- ⚠ While processes are blocked in the sense of synchronous message passing, they are not actually delayed as the intermediate is always ready.



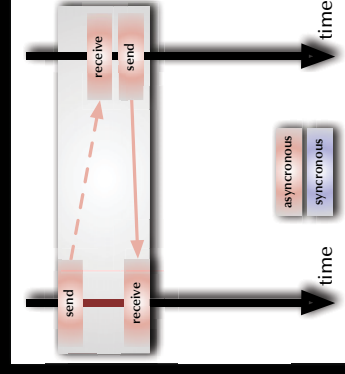
Communication & Synchronization

Message-based synchronization Message protocols

Synchronous message (simulated by asynchronous messages)

Introducing two asynchronous messages:

- Both processes voluntarily suspend themselves until the transaction is complete.
- As no immediate communication takes place, the processes are never actually synchronized.
- The sender (but not the receiver) process knows that the transaction is complete.



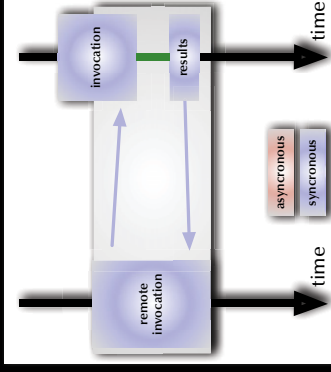


Communication & Synchronization

Message-based synchronization Message protocols

Remote invocation

- Delay sender or receiver until the first rendezvous point
- Pass parameters
- Keep sender blocked while receiver executes the local procedure
- Pass results
- Release both processes out of the rendezvous

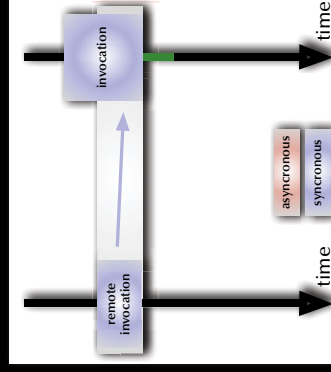


Communication & Synchronization

Message-based synchronization Message protocols

Remote invocation (no results)

- Shorter form of remote invocation which does not wait for results to be passed back.
- Still both processes are actually synchronized at the time of the invocation.

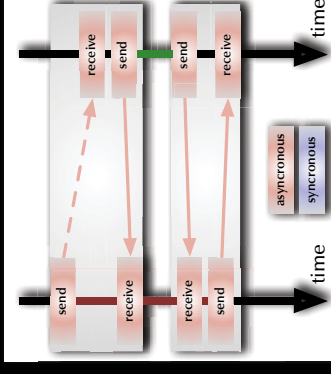


Communication & Synchronization

Message-based synchronization Message protocols

Remote invocation (simulated by asynchronous messages)

- Simulate two synchronous messages
- Processes are never actually synchronized

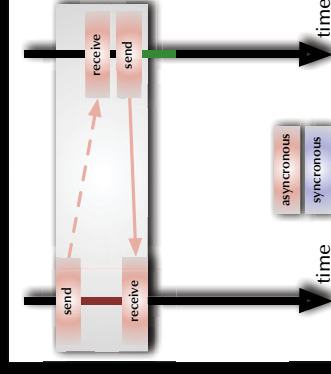


Communication & Synchronization

Message-based synchronization Message protocols

Remote invocation (no results) (simulated by asynchronous messages)

- Simulate one synchronous message
- Processes are never actually synchronized





Communication & Synchronization

Message-based synchronization

Synchronous vs. asynchronous communications

Purpose 'synchronization': ☞ synchronous messages / remote invocations
 Purpose 'last message(s) only': ☞ asynchronous messages

- ☞ Synchronous message passing in distributed systems requires hardware support.
- ☞ Asynchronous message passing requires the usage of buffers and overflow policies.

Can both communication modes emulate each other?



Communication & Synchronization

Message-based synchronization

Synchronous vs. asynchronous communications

Purpose 'synchronization': ☞ synchronous messages / remote invocations
 Purpose 'last message(s) only': ☞ asynchronous messages

- ☞ Synchronous message passing in distributed systems requires hardware support.
- ☞ Asynchronous message passing requires the usage of buffers and overflow policies.

Can both communication modes emulate each other?

- Synchronous *communications* are emulated by a combination of asynchronous messages in some systems (not identical with hardware supported synchronous communication).
- Asynchronous *communications* can be emulated in synchronized message passing systems by introducing a 'buffer-task' (de-coupling sender and receiver as well as allowing for broadcasts).



Communication & Synchronization

Message-based synchronization

Addressing (name space)

Direct versus 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



Communication & Synchronization

Message-based synchronization

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



Communication & Synchronization

Message-based synchronization 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, C)
 - ... semi-automatic (CORBA)
 - ... automatic (compiler-generated) and typed-persistent (Ada, CHILL, Occam2)



Communication & Synchronization

Message-based synchronization Message structure (Ada)

```
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;
```



Communication & Synchronization

Message-based synchronization Message structure (Ada)

Reading and writing values of any subtype S of a specific type T 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 subtype S of a specific type T 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;
```



Communication & Synchronization

Message-based synchronization Message-passing systems examples:

POSIX: "message queues":
 ☞ ordered indirect [asymmetrical | symmetrical] asynchronous
 byte-level many-to-many message passing

MPI: "message passing":
 ☞ ordered [direct | indirect] [asymmetrical | symmetrical] asynchronous memory-block-level [one-to-one | one-to-many | many-to-one | many-to-many] message passing

CHILL: "buffers"/"signals":
 ☞ ordered indirect [asymmetrical | symmetrical] [synchronous | asynchronous] typed [many-to-many | many-to-one] message passing

Occam2: "channels":
 ☞ ordered indirect symmetrical synchronous fully-typed one-to-one message passing

Ada: "(extended) rendezvous":
 ☞ ordered direct asymmetrical [synchronous | asynchronous] fully-typed many-to-one remote invocation

Java: no message passing system defined



Communication & Synchronization

Message-based synchronization

Message-passing systems examples:

	ordered	symmetrical	asymmetrical	synchronous	asynchronous	direct	indirect	contents	one-to-one	many-to-one	many-to-many	method
POSIX:	✓	✓	✓	✓	✓	✓	✓	byte-stream	✓	✓	✓	message queues
MPI:	✓	✓	✓	✓	✓	✓	✓	memory-blocks	✓	✓	✓	message passing
CHILL:	✓	✓	✓	✓	✓	✓	✓	basic types	✓	✓	✓	message passing
Occam2:	✓	✓	✓	✓	✓	✓	✓	fully typed	✓	✓	✓	message passing
Ada:	✓	✓	✓	✓	✓	✓	✓	fully typed	✓	✓	✓	remote invocation
Go:	✓	✓	✓	✓	✓	✓	✓	fully typed	✓	✓	✓	channels
Erlang:	✓	✓	✓	✓	✓	✓	✓	fully typed	✓	✓	✓	message passing

Java: ¹⁸ no message passing system defined



Communication & Synchronization

Message-based synchronization

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 SensorChannel1:
PAR
  INT reading:
  SEQ i = 0 FOR 1000
  SEQ
    -- generate reading
    SensorChannel1 ! reading
  INT data:
  SEQ i = 0 FOR 1000
  SEQ
    SensorChannel1 ? data
    -- employ data
  
```

concurrent entities are
synchronized at these points



Communication & Synchronization

Message-based synchronization

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 SensorChannel1:
PAR
  INT reading:
  SEQ i = 0 FOR 1000
  SEQ
    -- generate reading
    SensorChannel1 ! reading
  INT data:
  SEQ i = 0 FOR 1000
  SEQ
    SensorChannel1 ? data
    -- employ data
  
```

Essential Occam2 keywords

```

ALT PAR SEQ PRI
ANY CHAN OF
DATA TYPE RECORD OFFSETOF PACKED
BOOL BYTE INT REAL
CASE IF ELSE FOR FROM WHILE
FUNCTION RESULT PROC IS
PROCESSOR PROTOCOL TIMER
SKIP STOP VALOF
  
```

¹⁸ Concurrent, distributed, real-time programming language!



Communication & Synchronization

Message-based synchronization

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.

¹⁸ strong support for concurrency, synchronization, and communication (monitors, buffered message passing, synchronous channels)

```

dc1 SensorBuffer buffer (32) int;

...
send SensorBuffer (reading);
signal SensorChannel1 = (int) to consumertype;
...
send SensorChannel1 (reading)
to consumer

receive case
(SensorBuffer in data) : ...
esac;

receive case
(SensorChannel1 in data) : ...
esac;
  
```



Communication & Synchronization

Message-based synchronization

Message-based synchronization in CHILL

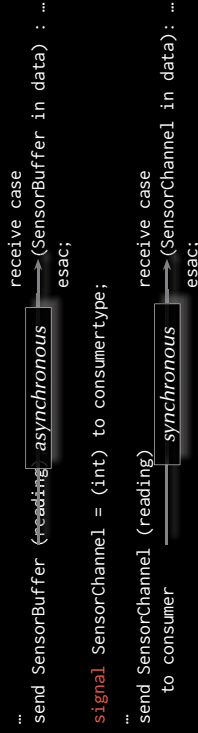
CHILL is the 'CCITT High Level Language',

where CCITT is the Comité Consultatif International Télégraphique et Téléphonique.

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

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

decl. `SensorBuffer buffer (32) int;`

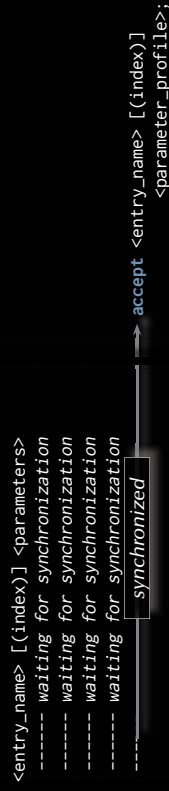


Communication & Synchronization

Message-based synchronization

Message-based synchronization in Ada

(Rendezvous)



time



Communication & Synchronization

Message-based synchronization

Message-based synchronization in Ada

Ada 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 then:

Parameters incl. bounds and discriminants are 'tunnelled' through byte-stream-formats.

Synchronization:

- Both tasks are synchronized at the beginning of the remote invocation (i.e. 'rendezvous')
- The calling task is blocked until the remote routine is completed (i.e. 'extended rendezvous')

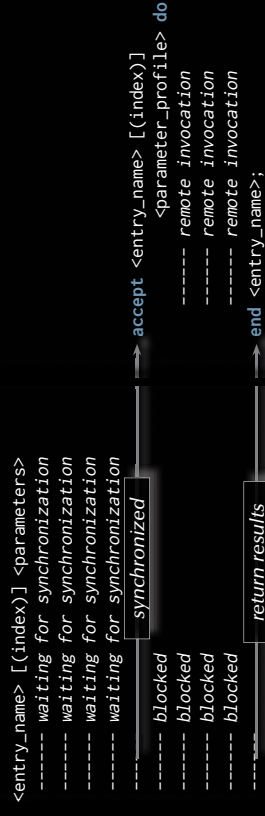


Communication & Synchronization

Message-based synchronization

Message-based synchronization in Ada

(Extended rendezvous)



time

