Overview

Synchronization methods

Shared memory based synchronization

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

Message based synchronization

- Asynchronous messages
- Synchronous messages
- Remote invocation, remote procedure call

Motivation

Side effects

Operations have side effects which are visible ... either

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

- ... outside the current process

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

References for this chapter

- [Ben-Ari06] M. Ben-Ari
  Principles of Concurrent and Distributed Programming

- [Barnes2006] Barnes, John
  Programming in Ada 2005

  The Java™ Language Specification
  - third edition
  2005

  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

Sanity check

Do we need to? – really?

int i; (declare globally to multiple threads)

i++; if i > n {i=0;}

(in one thread) (in another thread)

What's the worst that can happen?

- 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.

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!
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.

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:

  \[ \text{Flag := true;} \]

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.
Basic synchronization
by Semaphores

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’):
  
  \[ \text{P}(S) : (\text{as soon as } S > 0 \text{ then } S := S - 1) \]  
  
  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’):
  
  \[ \text{V}(S) : S := S + 1 \]  

aka ‘Signal’, ‘Set-True’, ‘sem_post’,...

\[ \text{then the variable } S \text{ is called a Semaphore.} \]

Towards synchronization
Condition synchronization by semaphores

\begin{align*}
\text{var } & \text{sync : semaphore := 0;} \\
\text{process } & \text{P1;} \\
& \text{statement } X; \\
& \text{wait (sync);} \\
& \text{statement } Y; \\
& \text{end } P1; \\
\text{process } & \text{P2;} \\
& \text{statement A;} \\
& \text{signal (sync);} \\
& \text{statement B;} \\
& \text{end } P2; \\
\end{align*}

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

Towards synchronization
Mutual exclusion by semaphores

\begin{align*}
\text{var } & \text{mutex : semaphore := 1;} \\
\text{process } & \text{P1;} \\
& \text{statement X;} \\
& \text{wait (mutex);} \\
& \text{statement Y;} \\
& \text{end P1;} \\
\text{process } & \text{P2;} \\
& \text{statement A;} \\
& \text{signal (mutex);} \\
& \text{statement B;} \\
& \text{end P2;} \\
\end{align*}

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

Towards synchronization
Semaphores in Ada

\begin{align*}
\text{package } & \text{Ada.Synchronous_Task_Control is} \\
& \text{type Suspension_Object is limited private;} \\
& \text{procedure Set_TRUE } (\text{S : in out Suspension_Object}); \\
& \text{procedure Set_FALSE } (\text{S : in out Suspension_Object}); \\
& \text{function Current_State } (\text{S : Suspension_Object}) \text{ return Boolean;} \\
& \text{procedure Suspend_Until_TRUE } (\text{S : in out Suspension_Object}); \\
& \text{private} \\
& \text{------ not specified by the language} \\
& \text{end Ada.Synchronous_Task_Control;} \\
\end{align*}

\begin{itemize}
  \item This is ‘queueless’ and can translate into a single machine instruction.
  \item only one task can be blocked at Suspend_Until_TRUE!
  \item (Program_Error will be raised with a second task trying to suspend itself)
  \item no queues! \item minimal run-time overhead
\end{itemize}
package Ada.Synchronous_Task_Control is
  type Suspension_Object is limited private;
  procedure Set_True (S : out Suspension_Object);
  procedure Set_False (S : out Suspension_Object);
  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!
(Progran_Error will be raised with a second task trying to suspend itself)
⇒ no queues! ⇒ minimal run-time overhead

© 2019 Uwe R. Zimmer, The Australian National University page 270 of 757 (chapter 3: “Communication & Synchronization” up to page 368)

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);
  Suspend_Until_True (X);
end B;

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

⇒ 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)

© 2019 Uwe R. Zimmer, The Australian National University page 271 of 757 (chapter 3: “Communication & Synchronization” up to page 368)

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);
  Suspend_Until_True (X);
  Suspend_Until_True (X);
end B;

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

Semaphores in POSIX

- `sem_getvalue`: Returns the semaphore value.
- `sem_init`: Initializes a semaphore.
- `sem_destroy`: Destroys a semaphore.
- `sem_wait` and `sem_post`: Wait for and post on the semaphore, respectively.
- `sem_trywait`: Tries to wait on the semaphore without blocking.
- `sem_timedwait`: Tries to wait on the semaphore for a specified time.

Semaphores in Java (since 2004)

- `acquire`: Acquires a semaphore.
- `acquireUninterruptibly`: Acquires a semaphore without waiting for interrupt.
- `tryAcquire`: Tries to acquire a semaphore.
- `availablePermits`: Returns the number of available permits.
- `reducePermits`: Reduces the number of outstanding permits.
- `drainPermits`: Drains all outstanding permits.
- `release`: Releases a semaphore.
- `getQueueLength`: Returns the length of the queue.
- `getQueuedThreads`: Gets the queue of threads.
- `isFair`: Checks if the semaphore is fair.

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.
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

- 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).)

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.
Centralized synchronization

Monitors

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

procedure take (var I : integer);
begin
end;

Centralized synchronization

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;

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!
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.

Monitors in Modula-1

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

Monitors in POSIX (‘C’)

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);...
Centralized synchronization

Monitors in POSIX ('C')

(types and creation)

Synchronization between POSIX-threads:
- typedef __ pthread_mutex_t;
types include:
- sharing of mutexes and
d condition variables between processes
- priority ceiling
- clock used for timeouts
- ... (operators)

- int pthread_mutex_init (      pthread_mutex_t     *mutex, const pthread_mutexattr_t *attr);
- int pthread_mutex_destroy (      pthread_mutex_t     *mutex);
- 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_init     (      pthread_cond_t      *cond, const pthread_condattr_t  *attr);
- int pthread_cond_destroy  (      pthread_cond_t      *cond);
- 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);
- ...
Centralized Synchronization

Monitors in POSIX ('C')

```c
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;
```

```c
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);
  }
  B->count ++;
  B->buf[B->count-1] = item;
  PTHREAD_COND_SIGNAL (&B->buffer_not_empty);
  return 0;
}
```

```c
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);
  *item = B->buf[B->count++];
  return 0;
}
```

Centralized Synchronization

Monitors in C#

```c
using System;
using System.Threading;

static long data_to_protect = 0;

static void Reader()
{
  try {
    Monitor.Enter (data_to_protect);
    while (true) {
      Monitor.Wait (data_to_protect);
      Console.WriteLine ("read out protected data");
    }
  }
  finally {
    Monitor.Exit (data_to_protect);
  }
}
```

```c
static void Writer()
{
  try {
    Monitor.Enter (data_to_protect);
    while (true) {
      data_to_protect ++;
      Monitor.Pulse (data_to_protect);
    }
  }
  finally {
    Monitor.Exit (data_to_protect);
  }
}
```

© 2019 Uwe R. Zimmer, The Australian National University page 293 of 757 (chapter 3: "Communication & Synchronization" up to page 368)
Centralized synchronization

Monitors in Java

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.
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> {...}

Centralized synchronization

Monitors in Java
(by means of language primitives)

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!
   - 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.

Standard monitor solution:

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

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

```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 ();
        }
    }
}
```

```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;  
                }
            }
        }
    }
}
```
Centralized synchronization

Monitors in Java

Per Brinch Hansen (1938-2007) in 1999:

Java’s most serious mistake was the decision to use the sequential part of the language to implement the run-time support for its parallel features. It strikes me as absurd to write a compiler for the sequential language concepts only and then attempt to skip the much more difficult task of implementing a secure parallel notation. This wishful thinking is part of Java’s unfortunate inheritance of the insecure C language and its primitive, error-prone library of threads methods.

“Per Brinch Hansen is one of a handful of computer pioneers who was responsible for advancing both operating systems development and concurrent programming from ad hoc techniques to systematic engineering disciplines.” (from his IEEE 2003 Computer Pioneer Award)

Centralized synchronization

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

- All provide lower-level primitives for the construction of monitors.
- All rely on convention rather than compiler checks.
- Visual C++, C+ & Visual Basic offer data-encapsulation and connection to the monitor.
- Java offers data-encapsulation (yet not with respect to a monitor).
- POSIX (being a collection of library calls) does not provide any data-encapsulation by itself.

- Extreme care must be taken when employing object-oriented programming and synchronization (incl. monitors)

Centralized synchronization

Nested monitor calls

Assuming a thread in a monitor is calling an operation in another monitor and is suspended at a conditional variable there:

- the called monitor is aware of the suspension and allows other threads to enter.
- the calling monitor is possibly not aware of the suspension and keeps its lock!
- the unjustified locked calling monitor reduces the system performance and leads to potential deadlocks.

Suggestions to solve this situation:
- Maintain the lock anyway: e.g. POSIX, Java
- Prohibit nested monitor calls: e.g. Modula-1
- Provide constructs which specify the release of a monitor lock for remote calls, e.g. Ada

Object-orientation and synchronization

Since mutual exclusion, notification, and condition synchronization schemes need to be designed and analyzed considering the implementation of all involved methods and guards:

- New methods cannot be added without re-evaluating the class!

Re-usage concepts of object-oriented programming do not translate to synchronized classes (e.g. monitors) and thus 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, yet they are complex and not offered in any concurrent programming language. Alternatively, inheritance can be banned in the context of synchronization (e.g. Ada).
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

Protected objects

- Mixture of low-level and high-level synchronization constructs.

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.

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): re-called entries in Ada:

Buffer_Size : constant Integer := 10;
type Index is mod Buffer_Size;
subtype Count is Natural range 0 .. Buffer_Size;
type Buffer_T is array (Index) of Data_Item;

protected type Bounded_Buffer is
  entry Get (Item : out Data_Item);
  entry Put (Item : Data_Item);
private
  First : Index := Index'First;
  Last : Index := Index'Last;
  Num : Count := 0;
  Buffer : Buffer_T;
end Bounded_Buffer;

Protected 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).
Centralized synchronization

Synchronization by protected objects

(Condition synchronization: entries & barriers)

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

Centralized synchronization

Synchronization by protected objects

(Withdrawing entry calls)

```vhdl
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;
```

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.
Centralized synchronization

Synchronization by protected objects

(Operations on entry queues)

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

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

- **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: external tasks are only considered for queuing on barriers once no internally requeued task can be progressed any further!

- **Private entries:**
  Protected entries which are not accessible from outside the protected object, but can be employed as destinations for requeue operations.

---

Centralized synchronization

Synchronization by protected objects

(Entry families, requeue & private entries)

Additional, essential primitives for concurrent control flows:

- **Entry families:**

  ```vhdl
  package Modes is
    type Mode_T is (Takeoff, Ascent, Cruising, Descent, Landing);
  protected body Block_Five is
    entry Proceed;
    when Proceed'count > 5 or Release is
      begin
      Release := Proceed'count > 0;
      end Proceed;
  end Block_Five;
  ```

- **Requeue facility:**

  ```vhdl
  private
    New_Message : Message;
    Arrived : Boolean := False;
  end Broadcast;
  ```

- **Private entries:**

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

---

Centralized synchronization

Synchronization by protected objects

(Entry families)

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

**Synchronization by protected objects**
(Entry families, requeue & private entries)

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

```plaintext
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;
```

---

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
**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

- **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

- **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

- **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?)
### 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.

#### 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)

#### 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.

### 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. 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.
Synchronization

Message-based synchronization

Synchronization model
• Asynchronous
• Synchronous
• Remote invocation

Message structure
• arbitrary
• restricted to ‘basic’ types
• restricted to un-typed communications

Addressing (name space)
• direct communication
• mail-box communication

Message protocols

Synchronous message
(sender waiting)

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

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.

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.
Message-based synchronization

Message protocols

Synchronous message
(receiver waiting)

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

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

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.

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.
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

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.

Remote invocation (simulated by asynchronous messages)

- Simulate one synchronous message
- Processes are never actually synchronized
**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?

**Addressing (name space)**

Direct versus indirect:
- send <message> to <process-name>
- send <message> to <mailbox>

Asymmetrical addressing:
- send <message> to ...

**Communication medium:**

<table>
<thead>
<tr>
<th>Connections</th>
<th>Functionality</th>
</tr>
</thead>
<tbody>
<tr>
<td>one-to-one</td>
<td>buffer, queue, synchronization</td>
</tr>
<tr>
<td>one-to-many</td>
<td>multicast</td>
</tr>
<tr>
<td>one-to-all</td>
<td>broadcast</td>
</tr>
<tr>
<td>many-to-one</td>
<td>local server, synchronization</td>
</tr>
<tr>
<td>all-to-one</td>
<td>general server, synchronization</td>
</tr>
<tr>
<td>many-to-many</td>
<td>general network- or bus-system</td>
</tr>
</tbody>
</table>
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)

Message-based synchronization

Message structure (Ada)

```ada
package Ada.Streams is
  pragma Pure (Streams);
  type Root_Stream_Type is abstract tagged limited private;
  type Stream_Element is mod 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  (…);
  procedure Write (…);

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

Message-passing systems examples:

- **POSIX**: "message queues";
  - ordered indirect [asymmetrical | symmetrical] asynchronous byte-level many-to-many message passing
- **MPL**: "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 [asymmetrical] [synchronous | asynchronous] fully-typed [many-to-many | many-to-one] message passing
- **Ada**: "(extended) rendezvous";
  - ordered direct [asymmetrical] [synchronous | asynchronous] fully-typed [many-to-many | many-to-one] remote invocation
- **Java**: no message passing system defined
Message-based synchronization

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

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

\[
\text{CHAN OF INT SensorChannel:}
\]

\[
\text{PAR INT reading: SEQ i = 0 FOR 1000 SEQ}
\]

-- generate reading

SensorChannel ! reading

\[
\text{INT data: SEQ i = 0 FOR 1000 SEQ}
\]

-- employ data

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.

\[\text{dcl SensorBuffer buffer (32) int;}\]

-- receive case

send SensorBuffer (reading); (SensorBuffer in data) : ..

\[\text{signal SensorChannel = (int) to consumer type;}\]

-- receive case
to consumer (SensorChannel in data) : ..

\[\text{Proc CHILL (parallelism and synchronization);} \]

concurrent entities are synchronized at these points
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.

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

```chill
dcl SensorBuffer buffer (32) int;
...
send SensorBuffer (reading) asynchronous (SensorBuffer in data) ;
...
receive case (SensorBuffer in data) :
  ...esac;

signal SensorChannel = (int) to consumertype;
...
send SensorChannel (reading) synchronous (SensorChannel in data) ;
...
receive case (SensorChannel in data) :
  ...esac;
```

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 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 (as ‘rendezvous’)
- The calling task is blocked until the remote routine is completed (as ‘extended rendezvous’)

```
<entry_name> [(index)] <parameters>
----- waiting for synchronization
----- waiting for synchronization
----- waiting for synchronization
----- waiting for synchronization
accept <entry_name> [(index)] <parameter_profile> ;

<entry_name> [(index)] <parameter_profile> do
----- blocked
----- blocked
----- blocked
----- blocked
accept <entry_name> [(index)] <parameter_profile> do
----- remote invocation
----- remote invocation
----- remote invocation
----- remote invocation
end <entry_name> ;
```
Message-based synchronization

Message-based synchronization in Ada

(Rendezvous)

accept \texttt{<entry_name> \texttt{[(index)]}} \texttt{<parameter_profile>};

----- waiting for synchronization
----- waiting for synchronization
----- waiting for synchronization

\texttt{<entry_name> \texttt{[(index)]} \texttt{<parameters>}} synchronized

time

(Extended rendezvous)

accept \texttt{<entry_name> \texttt{[(index)]}} \texttt{<parameter_profile>};

----- waiting for synchronization
----- waiting for synchronization
----- waiting for synchronization

\texttt{<entry_name> \texttt{[(index)]} \texttt{<parameters>}} synchronized

----- blocked
----- blocked
----- blocked
----- blocked

----- remote invocation
----- remote invocation
----- remote invocation
----- remote invocation

\texttt{<entry_name> \texttt{[(index)]} \texttt{<parameters>}} synchronized

----- return results

end \texttt{<entry_name>};

------ remote invocation

------ remote invocation

------ remote invocation

------ remote invocation

end

------ time

------ time

------ time

------ time

Some things to consider for task-entries:

- In contrast to protected-object-entries, task-entry bodies can call other blocking operations.
- Accept statements can be nested (but need to be different).
- Accept statements can have a dedicated exception handler (like any other code-block).
- Exceptions, which are not handled during the rendezvous phase are propagated to all involved tasks.
- Parameters cannot be direct ‘access’ parameters, but can be access-types.
- ‘count on task-entries is defined, but is only accessible from inside the tasks which owns the entry.
- Entry families (arrays of entries) are supported.
- Private entries (accessible for internal tasks) are supported.

Summary

Communication & 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
  - Examples