Communication & Synchronization

Sanity check

Do we need to? – really?

int i; (declare globally to multiple threads)
i++; if i > n {i=0;}

(int in one thread)
(int in another thread)

What’s the worst that can happen?

© 2020 Uwe R. Zimmer, The Australian National University page 257 of 758 (chapter 3: “Communication & Synchronization” up to page 369)

Do we need to? – really?

int i; (declare globally to multiple threads)
i++; if i > n {i=0;}

(int in one thread)
(int in another thread)

Unaligned manipulations on the main memory will usually not be atomic

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

© 2020 Uwe R. Zimmer, The Australian National University page 260 of 758 (chapter 3: “Communication & Synchronization” up to page 369)

Do we need to? – really?

int i; (declare globally to multiple threads)
i++; if i > n {i=0;}

(int in one thread)
(int in another thread)

Local caches might not be coherent

Yet perhaps they are.

© 2020 Uwe R. Zimmer, The Australian National University page 262 of 758 (chapter 3: “Communication & Synchronization” up to page 369)
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-sized atomic variable operating as a flag to indicate synchronization conditions.

![Image](https://example.com/image.png)

Sequence of operations:

\[ A \rightarrow Y \rightarrow Y \rightarrow B \]

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-sized atomic variable operating as a flag to indicate synchronization conditions.

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

- is not suitable for general mutual exclusion in critical sections!

- busy-waiting is required to pull the synchronization condition!

More powerful synchronization operations are required for critical sections.

More powerful synchronization operations are required for critical sections.

Towards synchronization

Condition synchronization by semaphores

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

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

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

- is not suitable for general mutual exclusion in critical sections!

- busy-waiting is required to pull the synchronization condition!

More powerful synchronization operations are required for critical sections.

More powerful synchronization operations are required for critical sections.

Towards synchronization

Condition synchronization by semaphores

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

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

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

- is not suitable for general mutual exclusion in critical sections!

- busy-waiting is required to pull the synchronization condition!

More powerful synchronization operations are required for critical sections.

More powerful synchronization operations are required for critical sections.

Towards synchronization

Condition synchronization by semaphores

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

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

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

- is not suitable for general mutual exclusion in critical sections!

- busy-waiting is required to pull the synchronization condition!

More powerful synchronization operations are required for critical sections.

More powerful synchronization operations are required for critical sections.
Towards synchronization

Malicious use of "queueless semaphores"

```ada
with Ada.Synchronous_Task_Control; use Ada.Synchronous_Task_Control;
X, Y : Suspension_Object;
task A; task body A is
begin
    Suspended_until_true (Y); Suspended_until_true (X);
    Set_true (Y); Suspended_until_true (Y);
    end B; end A;
end A;
```

or Will result in a deadlock (assuming no other `Set_true` calls)

Towards synchronization

Semaphores in POSIX

```c
int sem_init (sem_t *sem_location, int p, unsigned int value);
int sem_destroy (sem_t *sem_location);
int sem_post (sem_t *sem_location);
int sem_getvalue (sem_t *sem_location, int *value);
```

© 2020 Uwe R. Zimmer, The Australian National University

Towards synchronization

Distributed synchronization

Conditional Critical Regions

```
buffer : buffer;
resource critical_buffer_region : buffer;
```

© 2020 Uwe R. Zimmer, The Australian National University
Centralized synchronization

Monitors

(MODULE 3: HOARE — DJIKSTA, 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.

Monitors with condition synchronization

- Procedure take (var I : integer);
  - If buffer[I] is null: break
  - Else:
    - signal (spaceavailable);
    - buffer[I] := 0;
    - NumberInBuffer := NumberInBuffer - 1;
    - end

Wait
- Condition variables are implemented by semaphores (and Signal).

Basic idea:
- Collect all operations and data-structures
- Express them 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.

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.
- 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 with condition synchronization

(MODULE '74)

More efficient evaluation of the guards:
- Formulate all operations as procedures or functions.
- Collect all operations and data-structures.
- "protected" variables.
- "initialization" operations.

Monitors in POSIX ('C')

- procedure wait (s, r)
  - IF AWAITED (r) THEN SEND (r);
- procedure signal (s) return integer
  - IF AWAITED (s) THEN SEND (s);
Communication & Synchronization

Monitors in Java

Considerations:

1. Synchronized methods and code blocks:
   - In order to implement a monitor all methods in an object need to be synchronized.
   - or any other method can break a Java monitor and enter at any time.
   - Methods outside the monitor scope can synchronize at this object.
   - or it is impossible to synchronize a Java monitor locally. Enter lock accesses must exist on the system.
   - State information is shared between all objects of a class.
   - or access to state data need to be synchronized with all objects of a class.
   - Synchronize either in static synchronized block, synchronized (this.getClass()); (-)
   - or static methods public synchronized static <method> (-)

Centralized synchronization

Monitors in Java

(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 waiting on or all threads, which are waiting in the same synchronized object.

Centralized synchronization

Monitors in Java

Considerations:

2. Notification methods: wait, notify, and notifyAll
   - wait suspends the thread and releases the lock back to the monitor.
   - or notifyAll will keep all enclosing locks.
   - notify and notifyAll do not release the lock.
   - or 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
   - orlocks are not present at this level (in opposition to R3-Java).
   - Therefore, we need to implement our own solution with monitor or data
   - or notified threads need to wait for the lock to be released
   - and to re-evaluate its entry condition.

Centralized synchronization

Monitors in Java

Standard monitor solution:

• Declare the monitored datastructures private to the monitor object (non-static).

• Introduce a class ConditionVariable:

  ```java
  public class ConditionVariable {
    public boolean waitUntilDone = false;
  }
  ```

• Introduce synchronization-scopes in monitor methods:

  ```java
  public synchronized static <method> {…}
  ```

• 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 or by convention.

Monitors in Java

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

```java
public class ReaderWriters {
  private int readers = 0;
  private int waitingWriters = 0;
  private int waitingReaders = 0;
  private boolean writing = false;
  ConditionVariable OkToWrite = new ConditionVariable();
  ConditionVariable OkToRead = new ConditionVariable();

  public void StartWrite () throws InterruptedException {
    synchronized (OkToWrite) {
      waitingWriters++;               OkToWrite.wantToSleep = true;
      if (waitingWriters > 0) {
        OkToWrite.wait ();
      } else {
        if (OkToWrite.wantToSleep) OkToWrite.wait ();
        if (OkToRead.wantsToSleep) OkToRead.wait ();
        writing = false;
        OkToWrite.notifyAll (); // wakeup one writer
        OkToRead.notifyAll (); // wake up all readers
        writingReaders = waitingReaders;
        waitingReaders = 0;
      }
    }
  }

  public void StopWrite () {
    synchronized (OkToWrite) {
      if (writing) {
        OkToWrite.wait();
      } else {
        writing = false;
        OkToWrite.notifyAll (); // wake up all readers
        waitingReaders = writingReaders;
        writingReaders = 0;
      }
    }
  }

  public void StartRead () throws InterruptedException {
    synchronized (OkToRead) {
      if (!writing) {
        OkToRead.wait();
      } else {
        writing = true;
        OkToRead.notifyAll (); // wake up one reader
        writingReaders = waitingReaders;
        waitingReaders = 0;
      }
    }
  }

  public void StopRead () {
    synchronized (OkToRead) {
      if (!writing) {
        OkToRead.wait();
      } else {
        writing = false;
        OkToRead.notifyAll (); // wake up all readers
        waitingReaders = writingReaders;
        writingReaders = 0;
      }
    }
  }
}
```
Communication & Synchronization

Monitors in Java

```java
public void StopRead () { OkToRead() {
    synchronized (this) {
        if (waitingWriters > 0) {
            waitingReaders++;
            OkToRead.wantToSleep = true;
        }
        readers--;
    }
    if (OkToRead.wantToSleep) {
        OkToRead.wantToSleep = false;
    }
    if (OkToRead.wantToSleep) {
        OkToRead.wantToSleep = false;
    }
}
```

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.

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 discipline." (from The 2007 Computer Pioneer Award

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.
Centralized synchronization

Synchronization by protected objects

Condition synchronization is realized in the form of protected procedures combined with boolean predicates: 

```plaintext
G
entries
in
Ada:

Buffer : Bounded_Buffer;
```

Where `Buffer` is a `Bounded_Buffer` protected object. The `Bounded_Buffer` is a bounded buffer that can only hold a fixed number of items. The `entries` are the number of entries in the buffer.

Synchronization by protected objects

(Withdrawing entry calls)

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

1. When entering a protected procedure or entry
2. When leaving a protected function

Alternatively, an implementation may choose to evaluate barriers on different occasions:

- When calling a protected entry
- When ending a protected function

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

Condition synchronization is realized in the form of combined boolean predicates:

```plaintext
G
entries
in
Ada:
```

Where `Buffer` is a `Bounded_Buffer` protected object. The `Bounded_Buffer` is a bounded buffer that can only hold a fixed number of items. The `entries` are the number of entries in the buffer.

Synchronization by protected objects

(Withdrawing entry calls)

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

1. When entering a protected procedure or entry
2. When leaving a protected function

Alternatively, an implementation may choose to evaluate barriers on different occasions:

- When calling a protected entry
- When ending a protected function

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

Centralized synchronization

Synchronization by protected objects

(Withdrawing entry calls)

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

1. When entering a protected procedure or entry
2. When leaving a protected function

Alternatively, an implementation may choose to evaluate barriers on different occasions:

- When calling a protected entry
- When ending a protected function

Barriers are not evaluated while inside a protected object or on leaving a protected function.
Communication & Synchronization

Synchronization by protected objects
(entry families, request & private entry)

Conditional critical

Centralized synchronization

Synchronization by protected objects
(entry families, request & private entry)

Protected body Pre_Filter is
entry Reception (U : Urgency)
whenever Server (primary) count = 8 or else Server (secondary) count = 8
begin
if U is urgent and then Server (primary) count = 8 then
request Server (primary);
else
request Server (secondary);
end if;
end Pre_Filter;

Communication & Synchronization

Synchronization by protected objects
(entry families, request & private entry)

Entry families, requeue & private entries

Conditional critical

Centralized synchronization

Synchronization by protected objects
(entry families, request & private entry)

protected body Pre_Filter is
entry Reception (U : Urgency)
whenever Server (primary) count = 8 or else Server (secondary) count = 8
begin
if U is urgent and then Server (primary) count = 8 then
request Server (primary);
else
request Server (secondary);
end if;
end Pre_Filter;

Communication & Synchronization

Synchronization by protected objects
(entry families, request & private entry)

Entry families, requeue & private entries

Conditional critical

Centralized synchronization

Synchronization by protected objects
(entry families, request & private entry)

protected body Pre_Filter is
entry Reception (U : Urgency)
whenever Server (primary) count = 8 or else Server (secondary) count = 8
begin
if U is urgent and then Server (primary) count = 8 then
request Server (primary);
else
request Server (secondary);
end if;
end Pre_Filter;

Communication & Synchronization

Synchronization by protected objects
(entry families, request & private entry)

Entry families, requeue & private entries

Conditional critical

Centralized synchronization

Synchronization by protected objects
(entry families, request & private entry)

protected body Pre_Filter is
entry Reception (U : Urgency)
whenever Server (primary) count = 8 or else Server (secondary) count = 8
begin
if U is urgent and then Server (primary) count = 8 then
request Server (primary);
else
request Server (secondary);
end if;
end Pre_Filter;

Communication & Synchronization

Synchronization by protected objects
(entry families, request & private entry)

Entry families, requeue & private entries

Conditional critical

Centralized synchronization

Synchronization by protected objects
(entry families, request & private entry)

protected body Pre_Filter is
entry Reception (U : Urgency)
whenever Server (primary) count = 8 or else Server (secondary) count = 8
begin
if U is urgent and then Server (primary) count = 8 then
request Server (primary);
else
request Server (secondary);
end if;
end Pre_Filter;

Communication & Synchronization

Synchronization by protected objects
(entry families, request & private entry)

Entry families, requeue & private entries

Conditional critical

Centralized synchronization

Synchronization by protected objects
(entry families, request & private entry)

protected body Pre_Filter is
entry Reception (U : Urgency)
whenever Server (primary) count = 8 or else Server (secondary) count = 8
begin
if U is urgent and then Server (primary) count = 8 then
request Server (primary);
else
request Server (secondary);
end if;
end Pre_Filter;

Communication & Synchronization

Synchronization by protected objects
(entry families, request & private entry)

Entry families, requeue & private entries

Conditional critical

Centralized synchronization

Synchronization by protected objects
(entry families, request & private entry)

protected body Pre_Filter is
entry Reception (U : Urgency)
whenever Server (primary) count = 8 or else Server (secondary) count = 8
begin
if U is urgent and then Server (primary) count = 8 then
request Server (primary);
else
request Server (secondary);
end if;
end Pre_Filter;

Communication & Synchronization

Synchronization by protected objects
(entry families, request & private entry)

Entry families, requeue & private entries

Conditional critical

Centralized synchronization

Synchronization by protected objects
(entry families, request & private entry)

protected body Pre_Filter is
entry Reception (U : Urgency)
whenever Server (primary) count = 8 or else Server (secondary) count = 8
begin
if U is urgent and then Server (primary) count = 8 then
request Server (primary);
else
request Server (secondary);
end if;
end Pre_Filter;

Communication & Synchronization

Synchronization by protected objects
(entry families, request & private entry)

Entry families, requeue & private entries

Conditional critical

Centralized synchronization

Synchronization by protected objects
(entry families, request & private entry)

protected body Pre_Filter is
entry Reception (U : Urgency)
whenever Server (primary) count = 8 or else Server (secondary) count = 8
begin
if U is urgent and then Server (primary) count = 8 then
request Server (primary);
else
request Server (secondary);
end if;
end Pre_Filter;

Communication & Synchronization

Synchronization by protected objects
(entry families, request & private entry)

Entry families, requeue & private entries

Conditional critical

Centralized synchronization

Synchronization by protected objects
(entry families, request & private entry)

protected body Pre_Filter is
entry Reception (U : Urgency)
whenever Server (primary) count = 8 or else Server (secondary) count = 8
begin
if U is urgent and then Server (primary) count = 8 then
request Server (primary);
else
request Server (secondary);
end if;
end Pre_Filter;

Communication & Synchronization

Synchronization by protected objects
(entry families, request & private entry)

Entry families, requeue & private entries

Conditional critical

Centralized synchronization

Synchronization by protected objects
(entry families, request & private entry)

protected body Pre_Filter is
entry Reception (U : Urgency)
whenever Server (primary) count = 8 or else Server (secondary) count = 8
begin
if U is urgent and then Server (primary) count = 8 then
request Server (primary);
else
request Server (secondary);
end if;
end Pre_Filter;

Communication & Synchronization

Synchronization by protected objects
(entry families, request & private entry)

Entry families, requeue & private entries

Conditional critical

Centralized synchronization

Synchronization by protected objects
(entry families, request & private entry)

protected body Pre_Filter is
entry Reception (U : Urgency)
whenever Server (primary) count = 8 or else Server (secondary) count = 8
begin
if U is urgent and then Server (primary) count = 8 then
request Server (primary);
else
request Server (secondary);
end if;
end Pre_Filter;

Communication & Synchronization

Synchronization by protected objects
(entry families, request & private entry)

Entry families, requeue & private entries

Conditional critical

Centralized synchronization

Synchronization by protected objects
(entry families, request & private entry)

protected body Pre_Filter is
entry Reception (U : Urgency)
whenever Server (primary) count = 8 or else Server (secondary) count = 8
begin
if U is urgent and then Server (primary) count = 8 then
request Server (primary);
else
request Server (secondary);
end if;
end Pre_Filter;

Communication & Synchronization

Synchronization by protected objects
(entry families, request & private entry)

Entry families, requeue & private entries

Conditional critical

Centralized synchronization

Synchronization by protected objects
(entry families, request & private entry)

protected body Pre_Filter is
entry Reception (U : Urgency)
whenever Server (primary) count = 8 or else Server (secondary) count = 8
begin
if U is urgent and then Server (primary) count = 8 then
request Server (primary);
else
request Server (secondary);
end if;
end Pre_Filter;

Communication & Synchronization

Synchronization by protected objects
(entry families, request & private entry)

Entry families, requeue & private entries

Conditional critical

Centralized synchronization

Synchronization by protected objects
(entry families, request & private entry)

protected body Pre_Filter is
entry Reception (U : Urgency)
whenever Server (primary) count = 8 or else Server (secondary) count = 8
begin
if U is urgent and then Server (primary) count = 8 then
request Server (primary);
else
request Server (secondary);
end if;
end Pre_Filter;

Communication & Synchronization

Synchronization by protected objects
(entry families, request & private entry)

Entry families, requeue & private entries

Conditional critical

Centralized synchronization

Synchronization by protected objects
(entry families, request & private entry)

protected body Pre_Filter is
entry Reception (U : Urgency)
whenever Server (primary) count = 8 or else Server (secondary) count = 8
begin
if U is urgent and then Server (primary) count = 8 then
request Server (primary);
else
request Server (secondary);
end if;
end Pre_Filter;

Communication & Synchronization

Synchronization by protected objects
(entry families, request & private entry)

Entry families, requeue & private entries

Conditional critical

Centralized synchronization

Synchronization by protected objects
(entry families, request & private entry)

protected body Pre_Filter is
entry Reception (U : Urgency)
whenever Server (primary) count = 8 or else Server (secondary) count = 8
begin
if U is urgent and then Server (primary) count = 8 then
request Server (primary);
else
request Server (secondary);
end if;
end Pre_Filter;
Communication & Synchronization

Shared memory based synchronization

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

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

Conditional

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”, but 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.
- Not all algorithms break down into independent computation sites and so there is a need for memory integrity mechanisms in shared partitioned address spaces.

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

Message-based synchronization

Synchronous messages

Synchronous message

(sender waiting)

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

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
**Message-based synchronization**

**Message protocols**

**Asynchronous message** (simulated by synchronous messages)

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

**Synchronous message** (simulated by asynchronous messages)

- Introducing two asynchronous messages:
  1. Both processes voluntarily suspend themselves until the transaction is complete.
  2. As no immediate communication takes place, the processes are never actually blocked.
  3. The sender (but not the receiver) process knows that the transaction is complete.

**Remote invocation** (simulated by asynchronous messages)

- Simulate two synchronous messages
- Processes are never actually synchronized

**Remote invocation** (no results)

- Simulate one synchronous message
- Processes are never actually synchronized

### Synchronous vs. asynchronous communications

**Purpose**

- *Synchronous*: immediate communication
- *Asynchronous*: delayed communication

**Last message(s) only**

- *Synchronous*: last message
- *Asynchronous*: last message(s)

### Asynchronous message passing

- 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 synchronous message passing systems by introducing a 'buffer-task' (de-coupling sender and receiver as well as allowing for broadcasts).

### Direct versus indirect

- *Synchronous*: direct communication
- *Asynchronous*: indirect communication

**Addressing (name space)**

- *Synchronous*: process-name
- *Asynchronous*: send, receive

Asymmetrical addressing:

- *Send*: process-name
- *Receive*: process-name

- *Send*: message
- *Receive*: message

*Client-server paradigm*
Communication & Synchronization

Message-based synchronization

Addressing (name space)

Communication medium:

<table>
<thead>
<tr>
<th>Connections</th>
<th>Functionality</th>
</tr>
</thead>
<tbody>
<tr>
<td>one-to-one</td>
<td>broadcast</td>
</tr>
<tr>
<td>one-to-many</td>
<td>multicast</td>
</tr>
<tr>
<td>one-to-all</td>
<td>general network or bus-system</td>
</tr>
<tr>
<td>many-to-one</td>
<td>local event synchronization</td>
</tr>
<tr>
<td>many-to-many</td>
<td>distributed (packets)</td>
</tr>
</tbody>
</table>

• 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.

Pragmas (Ada, CHILL, Occam2):

- S'Read (Stream : access Ada.Streams.Root_Stream_Type'Class; data : in T) is a byte-level many-to-many message passing.
- S'Output (Stream : access Ada.Streams.Root_Stream_Type'Class; data : in T) is a byte-level many-to-many message passing.

Message-based synchronization in Occam2

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

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

```
CHAN OF INT SensorChannel:
PAR INT reading: SEQ i = 0 FOR 1000
SensorChannel ! reading
INT data: SEQ i = 0 FOR 1000
SensorChannel ? data
```

Message-based synchronization in CHILL

CHILL is the 'ECLIT High Level Language',
where ECLIT is the European Consultative International Telephonique et Telephonique.
The CHILL language development was started in 1973 and standardized in 1980.

- Strong support for concurrency, synchronization, and communication (monitors, buffered message passing, synchronous channels)
- Synchronous, memory-blocks, fully typed, one-to-one, remote invocation

Essential Occam2 keywords:

- DATA TYPE RECORD OFFSETOF PACKED
- INT BYTE REAL
- BOOLEAN BYTE INT REAL
- CASE IF ELSE FOR FROM WHILE
- ALTV INPUT OUTPUT DELAY STOP
- IF THEN ELSE WHEN
- SIGNAL SignalChannel + (int) to consumer
- receive SensorChannel (reading)
- send SensorBuffer (reading)
- send Buffer (in data)
- receive Buffer (in data)
Communication & 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 strongly supports concurrency, synchronization, and communication (channels, buffered message passing, synchronous channels).

dcl SensorBuffer buffer (32) int;

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