Communication & Synchronization

Overview

ISO/IEC 8652:2006 (E)

Principles of Concurrent and Distributed Programming


Version 2.01

Draft — January 13, 2010

[ Gosling2005 ]

Communication & Synchronization

Side effects

Operations base side effects which are visible...

- either
- ...really only
- (uncontrolled by means, e.g., hardware environment)
- or
- ...outside the current process

...it side effects transcends the local process in all forms of access need to be synchronized.

Sanity check

Do we need to? – really?

```
int i;
{declare globally to multiple threads}

if i > n {i=0;}
{in one thread}

{in another thread}
```

... yet perhaps it is an 8-bit integer.

Conditional critical regions

Outside the current process

```
for p = 1 to n:
    {declare globally to multiple threads}
    a = a + b
        
    {in one thread}
    a = a + b
        
    {in another thread}
```

Asynchronous messages

Forms of access need to be synchronized.

Synchronous messages

```
end
```

```
end
```

Many schedulers interrupt threads irrespective of shared data operations

Local caches might not be coherent

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

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

```
XA < AY B
```

```
(68)
```

Condition synchronization by flags

Assuming further that there is an ownership between two processes

```
A Y B
```

```
(69)
```

```
(70)
```

```
(71)
```

```
(72)
```

```
(73)
```

Condition synchronization by flags

```
{Flag : boolean}
```

```
{Flag : boolean}
```

```
{Flag : boolean}
```

```
{Flag : boolean}
```

```
{Flag : boolean}
```

```
{Flag : boolean}
```

```
{Flag : boolean}
```

Sequence of operations:

```
A := B \[X < A \mid F \] \rightarrow [X ; Y ; B]
```

Sequence of operations:

```
A := B \[X < A \mid F \] \rightarrow [X ; Y ; B]
```

Sequence of operations:

```
A := B \[X < A \mid F \] \rightarrow [X ; Y ; B]
```

Sequence of operations:

```
A := B \[X < A \mid F \] \rightarrow [X ; Y ; B]
```

```
A := B \[X < A \mid F \] \rightarrow [X ; Y ; B]
```

```
A := B \[X < A \mid F \] \rightarrow [X ; Y ; B]
```

```
A := B \[X < A \mid F \] \rightarrow [X ; Y ; B]
```

```
A := B \[X < A \mid F \] \rightarrow [X ; Y ; B]
```

```
A := B \[X < A \mid F \] \rightarrow [X ; Y ; B]
```

```
A := B \[X < A \mid F \] \rightarrow [X ; Y ; B]
```

```
A := B \[X < A \mid F \] \rightarrow [X ; Y ; B]
```

```
A := B \[X < A \mid F \] \rightarrow [X ; Y ; B]
```

```
A := B \[X < A \mid F \] \rightarrow [X ; Y ; B]
```

```
A := B \[X < A \mid F \] \rightarrow [X ; Y ; B]
```

```
A := B \[X < A \mid F \] \rightarrow [X ; Y ; B]
```

```
A := B \[X < A \mid F \] \rightarrow [X ; Y ; B]
```

```
A := B \[X < A \mid F \] \rightarrow [X ; Y ; B]
```

```
A := B \[X < A \mid F \] \rightarrow [X ; Y ; B]
```

```
A := B \[X < A \mid F \] \rightarrow [X ; Y ; B]
```

```
A := B \[X < A \mid F \] \rightarrow [X ; Y ; B]
```

```
A := B \[X < A \mid F \] \rightarrow [X ; Y ; B]
```

```
A := B \[X < A \mid F \] \rightarrow [X ; Y ; B]
```

```
A := B \[X < A \mid F \] \rightarrow [X ; Y ; B]
```

```
A := B \[X < A \mid F \] \rightarrow [X ; Y ; B]
```

```
A := B \[X < A \mid F \] \rightarrow [X ; Y ; B]
```

```
A := B \[X < A \mid F \] \rightarrow [X ; Y ; B]
```

```
A := B \[X < A \mid F \] \rightarrow [X ; Y ; B]
```

```
A := B \[X < A \mid F \] \rightarrow [X ; Y ; B]
```

```
A := B \[X < A \mid F \] \rightarrow [X ; Y ; B]
```

```
A := B \[X < A \mid F \] \rightarrow [X ; Y ; B]
```

```
A := B \[X < A \mid F \] \rightarrow [X ; Y ; B]
```

```
A := B \[X < A \mid F \] \rightarrow [X ; Y ; B]
```

```
A := B \[X < A \mid F \] \rightarrow [X ; Y ; B]
```

```
A := B \[X < A \mid F \] \rightarrow [X ; Y ; B]
```

```
A := B \[X < A \mid F \] \rightarrow [X ; Y ; B]
```

```
A := B \[X < A \mid F \] \rightarrow [X ; Y ; B]
```

```
A := B \[X < A \mid F \] \rightarrow [X ; Y ; B]
```

```
A := B \[X < A \mid F \] \rightarrow [X ; Y ; B]
```

```
A := B \[X < A \mid F \] \rightarrow [X ; Y ; B]
```

```
A := B \[X < A \mid F \] \rightarrow [X ; Y ; B]
```

```
A := B \[X < A \mid F \] \rightarrow [X ; Y ; B]
```

```
A := B \[X < A \mid F \] \rightarrow [X ; Y ; B]
```

```
A := B \[X < A \mid F \] \rightarrow [X ; Y ; B]
```

```
A := B \[X < A \mid F \] \rightarrow [X ; Y ; B]
```

```
A := B \[X < A \mid F \] \rightarrow [X ; Y ; B]
```

```
A := B \[X < A \mid F \] \rightarrow [X ; Y ; B]
```

```
A := B \[X < A \mid F \] \rightarrow [X ; Y ; B]
```

```
A := B \[X < A \mid F \] \rightarrow [X ; Y ; B]
```

```
A := B \[X < A \mid F \] \rightarrow [X ; Y ; B]
```

```
A := B \[X < A \mid F \] \rightarrow [X ; Y ; B]
```

```
A := B \[X < A \mid F \] \rightarrow [X ; Y ; B]
```

```
A := B \[X < A \mid F \] \rightarrow [X ; Y ; B]
```

```
A := B \[X < A \mid F \] \rightarrow [X ; Y ; B]
```

```
A := B \[X < A \mid F \] \rightarrow [X ; Y ; B]
```

```
A := B \[X < A \mid F \] \rightarrow [X ; Y ; B]
```
Communication & Synchronization

Towards synchronization

Malicious use of “queueless semaphores”

with no, just ticks, etc., use max just ticks, etc., etc., etc., etc., etc., etc., etc., etc., etc., etc., etc., etc., etc., etc., etc., etc., etc., etc., etc., etc., etc., etc., etc., etc., etc., etc., etc., etc., etc., etc., etc., etc., etc., etc., etc., etc., etc., etc., etc., etc., etc., etc., etc., etc., etc., etc., etc., etc., etc., etc., etc., etc., etc., etc., etc., etc., etc., etc., etc., etc., etc., etc., etc., etc., etc., etc., etc., etc., etc., etc., etc., etc., etc., etc., etc., etc., etc., etc., etc., etc., etc., etc., etc., etc., etc., etc., etc., etc., etc., etc., etc., etc., etc., etc., etc., etc., etc., etc., etc., etc., etc., etc., etc., etc., etc., etc., etc., etc., etc., etc., etc., etc., etc., etc., etc., etc., etc., etc., etc., etc., etc., etc., etc., etc., etc., etc., etc., etc., etc., etc., etc., etc., etc., etc., etc., etc., etc., etc., etc., etc., etc., etc., etc., etc., etc., etc., etc., etc., etc., etc., etc., etc., etc., etc., etc., etc., etc., etc., etc., etc., etc., etc., etc., etc., etc., etc., etc., etc., etc., etc., etc., etc., etc., etc., etc., etc., etc., etc., etc., etc., etc., etc., etc., etc., etc., etc., etc., etc., etc., etc., etc., etc., etc., etc., etc., etc., etc., etc., etc., etc., etc., etc., etc., etc., etc., etc., etc., etc., etc., etc., etc., etc., etc., etc., etc., etc., etc., etc., etc., etc., etc., etc., etc., etc., etc., etc., etc., etc., etc., etc., etc., etc., etc., etc., etc., etc., etc., etc., etc., etc., etc., etc., etc., etc., etc., etc., etc., etc., etc., etc., etc., etc., etc., etc., etc., etc., etc., etc., etc., etc., etc., etc., etc., etc., etc., etc., etc., etc., etc., etc., etc., etc., etc., etc., etc., etc., etc., etc., etc., etc., etc., etc., etc., etc., etc., etc., etc., etc., etc., etc., etc., etc., etc., etc., etc., etc., etc., etc., etc., etc., etc., etc., etc., etc., etc., etc., etc., etc., etc., etc., etc., etc., etc., etc., etc., etc., etc., etc., etc., etc., etc., etc., etc., etc., etc., etc., etc., etc., etc., etc., etc., etc., etc., etc., etc., etc., etc., etc., etc., etc., etc., etc., etc., etc., etc., etc., etc., etc., etc., etc., etc., etc., etc., etc., etc., etc., etc., etc., etc., etc., etc., etc., etc., etc., etc., etc., etc., etc., etc., etc., etc., etc., etc., etc., etc., etc., etc., etc., etc., etc., etc., etc., etc., etc., etc., etc., etc., etc., etc., etc., etc., etc., etc., etc., etc., etc., etc., etc., etc., etc., etc., etc., etc., etc., etc., etc., etc., etc., etc., etc., etc., etc., etc., etc., etc., etc., etc., etc., etc., etc., etc., etc., etc., etc., etc., etc., etc., etc., etc., etc., etc., etc., etc., etc., etc., etc., etc., etc., etc., etc., etc., etc., etc., etc., etc., etc., etc., etc., etc., etc., etc., etc., etc., etc., etc., etc., etc., etc., etc., etc., etc., etc., etc., etc., etc., etc., etc., etc., etc., etc., etc., etc., etc., etc., etc., etc., etc., etc., etc., etc., etc., etc., etc., etc., etc., etc., etc., etc., etc., etc., etc., etc., etc., etc., etc., etc., etc., etc., etc., etc., etc., etc., etc., etc., etc., etc., etc., etc., etc., etc., etc., etc., etc., etc., etc., etc., etc., etc., etc., etc., etc., etc., etc., etc., etc., etc., etc., etc., etc., etc., etc., etc., etc., etc., etc., etc., etc., etc., etc., etc., etc., etc., etc., etc., etc., etc., etc., etc., etc., etc., etc., etc., etc., etc., etc., etc., etc., etc., etc., etc., etc., etc., etc., etc., etc., etc., etc., etc., etc., etc., etc., etc., etc., etc., etc., etc., etc., etc., etc., etc., etc., etc., etc., etc., etc., etc., etc., etc., etc., etc., etc., etc., etc., etc., etc., etc., etc., etc., etc., etc., etc., etc., etc., etc., etc., etc., etc., etc., etc., etc., etc., etc., etc., etc., etc., etc., etc., etc., etc,
Centralized synchronization

Centralized synchronization

Monitors with condition synchronization

Suggestions to overcome the multiple-tasks-in-monitor-problem:

• Sharing of mutexes and cond-objects
• Synchronization between POSIX threads

Monitors in Modula-2

Monitors in POSIX (C)

Monitors in POSIX (C)

Monitors in Visual C++

Monitors in Visual Basic

Monitors in Java

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

• Synchronized methods and code blocks:
  all methods and code blocks which are using the synchronized region are mutually exclusive with respect to the address class.

• Notification methods:
  wait, notify, and notifyAll can be used only in synchronized regions and are making any or all threads, which are waiting in the same synchronized object.
Monitors in Java

(Continued from previous page)

1. Synchronized methods and code blocks:

   - `public synchronized void someMethod() { ... }`
   - `public synchronized static <method> { ... }`

2. Notification methods `wait`, `notify`, and `notifyAll`:

   `public synchronized void someMethod() { ... synchronized (this) { ... notify(); ... } ... }`

3. Monitor synchronization:

   - Synchronization is enforced over read and write operations on shared data.
   - `public synchronized void someMethod() { ... synchronized (sharedObject) { ... } ... }`

4. Deadlock prevention:

   - Use synchronization to prevent deadlocks.

5. Monitor structure:

   - A monitor is an object that holds the lock.

6. Monitor use:

   - `public static void main(String[] args) { ... synchronized (sharedObject) { ... } ... }`

7. Monitor overview:

   - `public synchronized void someMethod() { ... synchronized (this) { ... } ... }`

8. Monitor conditions:

   - `public synchronized void someMethod() { ... synchronized (this) { ... notify(); ... } ... }`

9. Monitor semantics:

   - `public synchronized void someMethod() { ... synchronized (this) { ... notifyAll(); ... } ... }`

10. Monitor implementation:

    - `public synchronized void someMethod() { ... synchronized (this) { ... } ... }`

Centralized synchronization

Monitors in Java

(Continued from previous page)

1. Synchronized methods and code blocks:

   - `public synchronized void someMethod() { ... }`
   - `public synchronized static <method> { ... }`

2. Notification methods `wait`, `notify`, and `notifyAll`:

   - `public synchronized void someMethod() { ... synchronized (this) { ... notify(); ... } ... }`

3. Monitor synchronization:

   - Synchronization is enforced over read and write operations on shared data.

4. Deadlock prevention:

   - Use synchronization to prevent deadlocks.

5. Monitor structure:

   - A monitor is an object that holds the lock.

6. Monitor use:

   - `public static void main(String[] args) { ... synchronized (sharedObject) { ... } ... }`

7. Monitor overview:

   - `public synchronized void someMethod() { ... synchronized (this) { ... } ... }`

8. Monitor conditions:

   - `public synchronized void someMethod() { ... synchronized (this) { ... notify(); ... } ... }`

9. Monitor semantics:

   - `public synchronized void someMethod() { ... synchronized (this) { ... notifyAll(); ... } ... }`

10. Monitor implementation:

    - `public synchronized void someMethod() { ... synchronized (this) { ... } ... }`

Centralized synchronization

Monitors in Java

(Continued from previous page)

1. Synchronized methods and code blocks:

   - `public synchronized void someMethod() { ... }`
   - `public synchronized static <method> { ... }`

2. Notification methods `wait`, `notify`, and `notifyAll`:

   - `public synchronized void someMethod() { ... synchronized (this) { ... notify(); ... } ... }`

3. Monitor synchronization:

   - Synchronization is enforced over read and write operations on shared data.

4. Deadlock prevention:

   - Use synchronization to prevent deadlocks.

5. Monitor structure:

   - A monitor is an object that holds the lock.

6. Monitor use:

   - `public static void main(String[] args) { ... synchronized (sharedObject) { ... } ... }`

7. Monitor overview:

   - `public synchronized void someMethod() { ... synchronized (this) { ... } ... }`

8. Monitor conditions:

   - `public synchronized void someMethod() { ... synchronized (this) { ... notify(); ... } ... }`

9. Monitor semantics:

   - `public synchronized void someMethod() { ... synchronized (this) { ... notifyAll(); ... } ... }`

10. Monitor implementation:

    - `public synchronized void someMethod() { ... synchronized (this) { ... } ... }`

Centralized synchronization

Monitors in Java

(Continued from previous page)

1. Synchronized methods and code blocks:

   - `public synchronized void someMethod() { ... }`
   - `public synchronized static <method> { ... }`

2. Notification methods `wait`, `notify`, and `notifyAll`:

   - `public synchronized void someMethod() { ... synchronized (this) { ... notify(); ... } ... }`

3. Monitor synchronization:

   - Synchronization is enforced over read and write operations on shared data.

4. Deadlock prevention:

   - Use synchronization to prevent deadlocks.

5. Monitor structure:

   - A monitor is an object that holds the lock.

6. Monitor use:

   - `public static void main(String[] args) { ... synchronized (sharedObject) { ... } ... }`

7. Monitor overview:

   - `public synchronized void someMethod() { ... synchronized (this) { ... } ... }`

8. Monitor conditions:

   - `public synchronized void someMethod() { ... synchronized (this) { ... notify(); ... } ... }`

9. Monitor semantics:

   - `public synchronized void someMethod() { ... synchronized (this) { ... notifyAll(); ... } ... }`

10. Monitor implementation:

    - `public synchronized void someMethod() { ... synchronized (this) { ... } ... }`

Centralized synchronization

Monitors in Java

(Continued from previous page)

1. Synchronized methods and code blocks:

   - `public synchronized void someMethod() { ... }`
   - `public synchronized static <method> { ... }`

2. Notification methods `wait`, `notify`, and `notifyAll`:

   - `public synchronized void someMethod() { ... synchronized (this) { ... notify(); ... } ... }`

3. Monitor synchronization:

   - Synchronization is enforced over read and write operations on shared data.

4. Deadlock prevention:

   - Use synchronization to prevent deadlocks.

5. Monitor structure:

   - A monitor is an object that holds the lock.

6. Monitor use:

   - `public static void main(String[] args) { ... synchronized (sharedObject) { ... } ... }`

7. Monitor overview:

   - `public synchronized void someMethod() { ... synchronized (this) { ... } ... }`

8. Monitor conditions:

   - `public synchronized void someMethod() { ... synchronized (this) { ... notify(); ... } ... }`

9. Monitor semantics:

   - `public synchronized void someMethod() { ... synchronized (this) { ... notifyAll(); ... } ... }`

10. Monitor implementation:

    - `public synchronized void someMethod() { ... synchronized (this) { ... } ... }`
Shared memory based synchronization

POSIX

- All the kernel concepts are available
- Control over all the processes by means of communication
- Synchronized processes
- Synchronized communications
- Use of signals for communication
- User-level synchronization
- Protection by software

Java

- Mutual exclusion on the code level
- Communication using object-oriented mechanisms
- Use of coroutines for communication
- Use of JavaMail for communication
- Use of CORBA for communication

C++, C#

- Universal object-oriented mechanisms
- Use of coroutines for communication
- Use of JavaMail for communication
- Use of CORBA for communication

C++14

- Universal object-oriented mechanisms
- Use of coroutines for communication
- Use of JavaMail for communication
- Use of CORBA for communication

C"

- Universal object-oriented mechanisms
- Use of coroutines for communication
- Use of JavaMail for communication
- Use of CORBA for communication

C++

- Universal object-oriented mechanisms
- Use of coroutines for communication
- Use of JavaMail for communication
- Use of CORBA for communication

C++/CLI

- Universal object-oriented mechanisms
- Use of coroutines for communication
- Use of JavaMail for communication
- Use of CORBA for communication

C# :

- Universal object-oriented mechanisms
- Use of coroutines for communication
- Use of JavaMail for communication
- Use of CORBA for communication

C

- Universal object-oriented mechanisms
- Use of coroutines for communication
- Use of JavaMail for communication
- Use of CORBA for communication

C++11

- Universal object-oriented mechanisms
- Use of coroutines for communication
- Use of JavaMail for communication
- Use of CORBA for communication

C++14

- Universal object-oriented mechanisms
- Use of coroutines for communication
- Use of JavaMail for communication
- Use of CORBA for communication

C++17

- Universal object-oriented mechanisms
- Use of coroutines for communication
- Use of JavaMail for communication
- Use of CORBA for communication

C++20

- Universal object-oriented mechanisms
- Use of coroutines for communication
- Use of JavaMail for communication
- Use of CORBA for communication

C++22

- Universal object-oriented mechanisms
- Use of coroutines for communication
- Use of JavaMail for communication
- Use of CORBA for communication

C++23

- Universal object-oriented mechanisms
- Use of coroutines for communication
- Use of JavaMail for communication
- Use of CORBA for communication
Communication & Synchronization

Shared memory based synchronization

- Mutual exclusion
- Deadlock detection
- Resource starvation
- Conditional exclusion
- Concurrency control
- Resource allocation

Monitors

- High-level synchronization support
- Mutual exclusive in scopes
- Guards (barriers)
- Synchronized with locks to protect it

Shared memory based synchronization

- High Performance Computing (HPC) emphasizes on keeping as many CPU nodes busy as possible:
  - Rust
  - Modula-1, Chill, Parallel Pascal, ...
  - Dijkstra / Hoare monitor concept
  - Chapel, Fortress, X10.
  - Chapel offers only atomic blocks in unconditional and conditional form.
  - X10 offers only atomic blocks in unconditional and conditional form.
  - Chapel offers a variety of concurrent operations:
    - Parallel operations on data (e.g., concurrent array operations)
    - Parallel statements, and parallel I/O (if supported by the language)
    - Parallelism can also be explicitly limited by serializing or parallelizing
      with synchronization statements
    - Atomic blocks for communication and synchronization

Synchronization in Chapel

- Message-based synchronization
- Message protocols
- Synchronous message (synchronous)
- Asynchronous message (simulated by synchronous messages)
- Remote invocation (no results) (simulated by asynchronous messages)

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

- Simulate two synchronous messages
- Message structure
- Message protocols
- Addressing (name space)
- Direct invocation
- mailbox communication

Message-based synchronization

- Message protocols
- Synchronous message (sender waiting)
- Asynchronous message (receiver waiting)
- Remote invocation (no results)

Remote invocation (no results)

- Simulate asynchronous invocation (both sender and receiver)
- Remote invocation (no results) (simulated by asynchronous messages)
- Remote invocation (no results) (simulated by synchronous messages)
- Message-based synchronization
- Message protocols
- Asynchronous message (simulated by synchronous messages)
- Asynchronous message (simulated by asynchronous messages)
- Remote invocation (no results) (simulated by synchronous messages)

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

- Remote invocation (no results) (simulated by synchronous messages)
- Remote invocation (no results) (simulated by asynchronous messages)
- Message-based synchronization
- Message protocols
- Asynchronous message (simulated by synchronous messages)
- Asynchronous message (simulated by asynchronous messages)
- Remote invocation (no results) (simulated by synchronous messages)
- Remote invocation (no results) (simulated by asynchronous messages)

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

- Remote invocation (no results) (simulated by asynchronous messages)
- Message-based synchronization
- Message protocols
- Asynchronous message (simulated by synchronous messages)
- Asynchronous message (simulated by asynchronous messages)
- Remote invocation (no results) (simulated by synchronous messages)
- Remote invocation (no results) (simulated by asynchronous messages)
Synchronous vs. asynchronous communications

Synchronous communications
- require hardware support
- used in real-time systems
- guarantees delivery of messages

Asynchronous communications
- use buffers and overflow policies
- suitable for non-real-time systems
- does not guarantee delivery of messages

Message-based synchronization

Message structure (Ada)
- supports remote invocations
- extended rendezvous

Message-based synchronization in Occam2
- supports remote invocations
- extended rendezvous

Message-based synchronization in CHILL
- supports remote invocations
- extended rendezvous

Communication medium:
- Connection-oriented (stream)
- Datagram (message)

Message-passing systems examples:
- CHILL
- Ada
- Erlang
- Java
- MPI

Message-based synchronization in Ada

Communication system is often outside the typed language environment.

Asynchronous message passing requires the usage of buffers and overflow policies.

Asynchronous communications are emulated by a combination of asynchronous messages in some systems.

Ada supports remote invocations extended rendezvous in a model of:
- introspection tables
- call out operations and profiles supported
- typed and untyped remote function invocations, or an asynchronous communication system such as channels
- parameters in bounds and domains are handled through high-level formats.

Synchronization:
- Both sides synchronize at the beginning of the remote invocation in a model
- The safety check is triggered on the remote receiver completed a potential condition.

Synchronous message passing in distributed systems requires hardware support.

Connections Functionality:
- one-to-one buffer, queue, synchronization
- one-to-all broadcast
- asymmetrical addressing
- all-to-one general server, synchronization

Message-based synchronization in CHILL

CHILL is the X3T9 High-Level Language

The CHILL language development started in the 1970s and standardized in 1980.

Communication is modeled as a sequence of states:
- send SensorBuffer (reading);
- signal SensorChannel ? data
- accept SensorChannel ? data

Communication is controlled by events:
- send SensorBuffer (reading);
- signal SensorChannel ? data
- accept SensorChannel ? data

Asynchronous communications are modeled by a combination of asynchronous messages in some systems.
Message-based synchronization

Message-based synchronization in Ada

(Rendezvous)

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

synchronized

Message-based synchronization

Message-based synchronization in Ada

(Extended rendezvous)

accept <entry_name> [(index)] <parameter_profile>; ------ waiting for synchronization ------ waiting for synchronization ------ waiting for synchronization ------ waiting for synchronization ------ remote invocation ------ remote invocation ------ remote invocation ------ remote invocation end <entry_name>; ------ return results

Some things to consider for task-entries:

• In contrast to protected-object-entries, task-entry bodies can call all other blocking operations.
• An accept statement can use several handlers to be defined.
• An accept statement can use a dedicated exception handler to synchronize more than two tasks.
• Accept statements can have a dedicated exception handler (like any other code-block).
• Parameters can be direct 'access' parameters, but can be accessed through references.
• Parameters can be direct 'access' parameters, but can be accessed through references.
• Guards evaluated during the rendezvous phase are propagated to all involved tasks.
• A task's main task address is defined.
• Private entries are accessible to the main task which owns the entry.
• Private entries are accessible for internal tasks only supported.

Summary

• Shared memory based synchronization
  • Flags, condition variables, semaphores, ...
  • 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