Systems, Networks & Concurrency 2018

Communication & Synchronization

References for this chapter

Overview


M. Ben-Ari


• Synchronous messages
• Remote invocation, remote procedure call

Message-based synchronization

Performance

Operations have side effects which are visible …

• An atomic operation (aka 'Signal', 'Set-True', 'sem_post', …)
• The operation is performed by a privileged processor or the actual processor

Semaphore

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

Flag := true;

Memory flag method is ok for simple condition synchronization, but "… busy-waiting is required to poll the synchronization condition!"

Suspension_Object

Mutual exclusion by semaphores

Semaphore

Limited

private;

mutex : semaphore := 1;

Set_True (S : semaphore);

Sequence of operations:

A := R (X: A) := [F: X: Y: Z: B:]

Semaphore

Back synchronization by semaphores

Sequence of operations:

A := B (X := A) := [F: X: Y: Z: B:]

Communication & Synchronization

Condition synchronization by flags

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

A, B are processes

Assume processes agree on shared memory

Semaphore

Flag := true;

Sequence of operations:

A := B (X := A) := [F: X: Y: Z: B:]

Condition synchronization by flags

A := B (X := A) := [F: X: Y: Z: B:]

Semaphore

Back synchronization by semaphores

Sequence of operations:

A := B (X := A) := [F: X: Y: Z: B:]

Condition synchronization by semaphores

A := B (X := A) := [F: X: Y: Z: B:]

Semaphore

Back synchronization by semaphores

Sequence of operations:

A := B (X := A) := [F: X: Y: Z: B:]

Condition synchronization by semaphores

A := B (X := A) := [F: X: Y: Z: B:]

Semaphore

Back synchronization by semaphores

Sequence of operations:

A := B (X := A) := [F: X: Y: Z: B:]

Condition synchronization by flags

A := B (X := A) := [F: X: Y: Z: B:]

Semaphore

Back synchronization by flags

Sequence of operations:

A := B (X := A) := [F: X: Y: Z: B:]

Condition synchronization by flags

A := B (X := A) := [F: X: Y: Z: B:]

Semaphore

Back synchronization by flags

Sequence of operations:

A := B (X := A) := [F: X: Y: Z: B:]

Condition synchronization by flags

A := B (X := A) := [F: X: Y: Z: B:]

Semaphore

Back synchronization by flags

Sequence of operations:

A := B (X := A) := [F: X: Y: Z: B:]

Condition synchronization by flags

A := B (X := A) := [F: X: Y: Z: B:]

Semaphore

Back synchronization by flags

Sequence of operations:

A := B (X := A) := [F: X: Y: Z: B:]

Condition synchronization by flags

A := B (X := A) := [F: X: Y: Z: B:]

Semaphore

Back synchronization by flags

Sequence of operations:

A := B (X := A) := [F: X: Y: Z: B:]

Condition synchronization by flags

A := B (X := A) := [F: X: Y: Z: B:]

Semaphore

Back synchronization by flags

Sequence of operations:

A := B (X := A) := [F: X: Y: Z: B:]

Condition synchronization by flags

A := B (X := A) := [F: X: Y: Z: B:]

Semaphore

Back synchronization by flags

Sequence of operations:

A := B (X := A) := [F: X: Y: Z: B:]

Condition synchronization by flags

A := B (X := A) := [F: X: Y: Z: B:]

Semaphore

Back synchronization by flags

Sequence of operations:

A := B (X := A) := [F: X: Y: Z: B:]

Condition synchronization by flags

A := B (X := A) := [F: X: Y: Z: B:]

Semaphore

Back synchronization by flags

Sequence of operations:

A := B (X := A) := [F: X: Y: Z: B:]

Condition synchronization by flags

A := B (X := A) := [F: X: Y: Z: B:]

Semaphore

Back synchronization by flags

Sequence of operations:

A := B (X := A) := [F: X: Y: Z: B:]

Condition synchronization by flags

A := B (X := A) := [F: X: Y: Z: B:]

Semaphore
Semaphores in Ada

```
-- Only one task can be blocked at a time (occurs only with high efficiency semaphores which do not provide process queues)
-- when all tasks are blocked
```

Review of semaphores

```
define emun {low, high} priority_t;

typedef emun {low, high} priority_t;

int waiting; int busy;

sem_init (sem_t *sem_location, int pshared, unsigned int value);

int sem_wait (sem_t *sem_location);

int sem_getvalue (sem_t *sem_location, int pshared, unsigned int *sem_value);

int sem_post (sem_t *sem_location);

```

All concurrent languages and environments offer effi cient and higher-abstraction synchronization methods.

Synchronization

```
• Critical regions are a means to avoid interfering with the execution of other processes
• Semaphores are well known, efficient, easy to use
• Monitors are better suited for processes that need to communicate

Monitors with condition synchronization

```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
Synchronization between POSIX-threads:

- `pthread_mutex_t *mutex;`
- `int pthread_mutex_lock (pthread_mutex_t *mutex);`
- `int pthread_mutex_unlock (pthread_mutex_t *mutex);`
- `int pthread_mutex_init (pthread_mutex_t *mutex, const pthread_mutexattr_t *attr);`
- `int pthread_mutex_destroy (pthread_mutex_t *mutex);`
- `int pthread_mutexattr_init (pthread_mutexattr_t *attr);`
- `int pthread_mutexattr_destroy (pthread_mutexattr_t *attr);`
- `int pthread_mutexattr_settype (pthread_mutexattr_t *attr, int type);`
- `int pthread_mutexattr_gettype (pthread_mutexattr_t *attr, int *type);`

Monitors in Modula-1

```modula
MODULE Monitors

FUNCTION allocate
    BEGIN
    busy := FALSE;
    END;

FUNCTION deallocate
    BEGIN
    busy := TRUE;
    END;

PROCEDURE enter (PROCEDURE P;)
    BEGIN
    IF busy THEN
        signal
    END;

PROCEDURE exit
    BEGIN
    busy := TRUE;
    END;

```

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

• `SIGNAL`: Delays the caller until condition variable is true (`r`)
• `SEND`: Signal to exit the condition variable
• `RECEIVE`: Operation which unblocks another process

Monitors in POSIX ('C)

```c
typedef … pthread_mutex_t;
int pthread_mutex_init (pthread_mutex_t      *mutex,
                        const pthread_mutexattr_t *attr);

typedef … pthread_mutexattr_t;
int pthread_mutex_destroy (pthread_mutex_t      *mutex);

typedef … pthread_cond_t;
int pthread_cond_signal (pthread_cond_t      *cond);
int pthread_cond_broadcast (pthread_cond_t      *cond);

typedef … pthread_condattr_t;
int pthread_condattr_init (pthread_condattr_t      *attr);
int pthread_condattr_destroy (pthread_condattr_t      *attr);
int pthread_condattr_settype (pthread_condattr_t      *attr, int type);
int pthread_condattr_gettype (pthread_condattr_t      *attr, int *type);
```

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 synchronization are mutually exclusive with respect to the addressed class.
  - Notification methods: `notify(), notifyAll()` can be used only in synchronized methods and are making any or all threads which are waiting in the same synchronized object.

Monitors in Visual C++

```c++
using namespace System;
using namespace System::Threading;

public class Monitor
{
    private bool busy;
    private bool ready;

    public void allocate()
    {
        busy = false;
    }

    public void deallocate()
    {
        busy = true;
    }

    public void enter()
    {
        if (busy)
        {
            signal();
        }
    }

    public void exit()
    {
        busy = true;
    }

    public void synchronize()
    {
        lock
        {
            // Synchronized code block
        }
    }
}
```

Monitors in Visual Basic

```vbnet
Public Class Monitor
    Private busy As Boolean
    Private ready As Boolean

    Public Sub allocate()
    Busy = False
    End Sub

    Public Sub deallocate()
    Busy = True
    End Sub

    Public Sub enter()
    If Busy Then
        Signal
    End Sub

    Public Sub exit()
    Busy = True
    End Sub

    Public Sub synchronize()
    Lock
    {
        // Synchronized code block
    }
End Class
```

Monitors in C#

```csharp
using System;
using System.Threading;

public class Monitor
{
    private bool busy;
    private bool ready;

    public void allocate()
    {
        busy = false;
    }

    public void deallocate()
    {
        busy = true;
    }

    public void enter()
    {
        if (busy)
        {
            signal();
        }
    }

    public void exit()
    {
        busy = true;
    }

    public void synchronize()
    {
        lock
        {
            // Synchronized code block
        }
    }
}
```

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 synchronization are mutually exclusive with respect to the addressed class.
  - Notification methods: `notify(), notifyAll()` can be used only in synchronized methods and are making any or all threads which are waiting in the same synchronized object.
Monitors in Java

Considerations:

1. Synchronized methods and code blocks:
   - In order to implement an entire method in a synchronized block, or any other operation which is even to be monitored and only one instance of the object at a time.
   - Methods outside the monitor object appear to the outside observer to be in the local lock only.
   - It is compulsory to analyze the existence of local locks yourself, because the monitor object does not provide any local lock.
   - Static data shared between all instances of a class is not made thread-safe by synchronized methods. Implementations such as the monitor's behavior share information with all objects of a class.

2. Notification methods notify, notifyAll, and wait:
   - If a thread is waiting for the local lock only, it does not release the lock, and the thread will be suspended for a specified period of time.
   - If a thread is waiting for the global lock, it releases the lock, and the thread will be suspended for a specified period of time.

3. Nested monitor calls:
   - Nested synchronized methods call cannot be used if the calling method is not synchronized.
   - Nested synchronized blocks cannot be used if the calling block is not synchronized.

4. Protection of external variables:
   - Use of external variables requires synchronization.
   - Monitors are not intended for use in multiple threads.

5. Protection of static data:
   - Use of static data requires synchronization.
   - Monitors are not intended for use in multiple threads.

6. Protection of volatile data:
   - Use of volatile data requires synchronization.
   - Monitors are not intended for use in multiple threads.

7. Protection of shared data:
   - Use of shared data requires synchronization.
   - Monitors are not intended for use in multiple threads.

8. Protection of synchronized methods:
   - Use of synchronized methods requires synchronization.
   - Monitors are not intended for use in multiple threads.

9. Protection of synchronized blocks:
   - Use of synchronized blocks requires synchronization.
   - Monitors are not intended for use in multiple threads.

10. Protection of synchronized variables:
    - Use of synchronized variables requires synchronization.
    - Monitors are not intended for use in multiple threads.

11. Protection of synchronized fields:
     - Use of synchronized fields requires synchronization.
     - Monitors are not intended for use in multiple threads.

12. Protection of synchronized classes:
     - Use of synchronized classes requires synchronization.
     - Monitors are not intended for use in multiple threads.

13. Protection of synchronized objects:
     - Use of synchronized objects requires synchronization.
     - Monitors are not intended for use in multiple threads.

14. Protection of synchronized methods:
     - Use of synchronized methods requires synchronization.
     - Monitors are not intended for use in multiple threads.

15. Protection of synchronized blocks:
     - Use of synchronized blocks requires synchronization.
     - Monitors are not intended for use in multiple threads.

16. Protection of synchronized variables:
     - Use of synchronized variables requires synchronization.
     - Monitors are not intended for use in multiple threads.

17. Protection of synchronized fields:
     - Use of synchronized fields requires synchronization.
     - Monitors are not intended for use in multiple threads.

18. Protection of synchronized classes:
     - Use of synchronized classes requires synchronization.
     - Monitors are not intended for use in multiple threads.

19. Protection of synchronized objects:
     - Use of synchronized objects requires synchronization.
     - Monitors are not intended for use in multiple threads.

20. Protection of synchronized methods:
     - Use of synchronized methods requires synchronization.
     - Monitors are not intended for use in multiple threads.

21. Protection of synchronized blocks:
     - Use of synchronized blocks requires synchronization.
     - Monitors are not intended for use in multiple threads.

22. Protection of synchronized variables:
     - Use of synchronized variables requires synchronization.
     - Monitors are not intended for use in multiple threads.

23. Protection of synchronized fields:
     - Use of synchronized fields requires synchronization.
     - Monitors are not intended for use in multiple threads.

24. Protection of synchronized classes:
     - Use of synchronized classes requires synchronization.
     - Monitors are not intended for use in multiple threads.

25. Protection of synchronized objects:
     - Use of synchronized objects requires synchronization.
     - Monitors are not intended for use in multiple threads.

26. Protection of synchronized methods:
     - Use of synchronized methods requires synchronization.
     - Monitors are not intended for use in multiple threads.

27. Protection of synchronized blocks:
     - Use of synchronized blocks requires synchronization.
     - Monitors are not intended for use in multiple threads.

28. Protection of synchronized variables:
     - Use of synchronized variables requires synchronization.
     - Monitors are not intended for use in multiple threads.

29. Protection of synchronized fields:
     - Use of synchronized fields requires synchronization.
     - Monitors are not intended for use in multiple threads.

30. Protection of synchronized classes:
     - Use of synchronized classes requires synchronization.
     - Monitors are not intended for use in multiple threads.

31. Protection of synchronized objects:
     - Use of synchronized objects requires synchronization.
     - Monitors are not intended for use in multiple threads.

32. Protection of synchronized methods:
     - Use of synchronized methods requires synchronization.
     - Monitors are not intended for use in multiple threads.

33. Protection of synchronized blocks:
     - Use of synchronized blocks requires synchronization.
     - Monitors are not intended for use in multiple threads.

34. Protection of synchronized variables:
     - Use of synchronized variables requires synchronization.
     - Monitors are not intended for use in multiple threads.

35. Protection of synchronized fields:
     - Use of synchronized fields requires synchronization.
     - Monitors are not intended for use in multiple threads.

36. Protection of synchronized classes:
     - Use of synchronized classes requires synchronization.
     - Monitors are not intended for use in multiple threads.

37. Protection of synchronized objects:
     - Use of synchronized objects requires synchronization.
     - Monitors are not intended for use in multiple threads.

38. Protection of synchronized methods:
     - Use of synchronized methods requires synchronization.
     - Monitors are not intended for use in multiple threads.

39. Protection of synchronized blocks:
     - Use of synchronized blocks requires synchronization.
     - Monitors are not intended for use in multiple threads.

40. Protection of synchronized variables:
     - Use of synchronized variables requires synchronization.
     - Monitors are not intended for use in multiple threads.

41. Protection of synchronized fields:
     - Use of synchronized fields requires synchronization.
     - Monitors are not intended for use in multiple threads.

42. Protection of synchronized classes:
     - Use of synchronized classes requires synchronization.
     - Monitors are not intended for use in multiple threads.

43. Protection of synchronized objects:
     - Use of synchronized objects requires synchronization.
     - Monitors are not intended for use in multiple threads.

44. Protection of synchronized methods:
     - Use of synchronized methods requires synchronization.
     - Monitors are not intended for use in multiple threads.

45. Protection of synchronized blocks:
     - Use of synchronized blocks requires synchronization.
     - Monitors are not intended for use in multiple threads.

46. Protection of synchronized variables:
     - Use of synchronized variables requires synchronization.
     - Monitors are not intended for use in multiple threads.

47. Protection of synchronized fields:
     - Use of synchronized fields requires synchronization.
     - Monitors are not intended for use in multiple threads.

48. Protection of synchronized classes:
     - Use of synchronized classes requires synchronization.
     - Monitors are not intended for use in multiple threads.

49. Protection of synchronized objects:
     - Use of synchronized objects requires synchronization.
     - Monitors are not intended for use in multiple threads.

50. Protection of synchronized methods:
     - Use of synchronized methods requires synchronization.
     - Monitors are not intended for use in multiple threads.

51. Protection of synchronized blocks:
     - Use of synchronized blocks requires synchronization.
     - Monitors are not intended for use in multiple threads.

52. Protection of synchronized variables:
     - Use of synchronized variables requires synchronization.
     - Monitors are not intended for use in multiple threads.

53. Protection of synchronized fields:
     - Use of synchronized fields requires synchronization.
     - Monitors are not intended for use in multiple threads.

54. Protection of synchronized classes:
     - Use of synchronized classes requires synchronization.
     - Monitors are not intended for use in multiple threads.

55. Protection of synchronized objects:
     - Use of synchronized objects requires synchronization.
     - Monitors are not intended for use in multiple threads.

56. Protection of synchronized methods:
     - Use of synchronized methods requires synchronization.
     - Monitors are not intended for use in multiple threads.

57. Protection of synchronized blocks:
     - Use of synchronized blocks requires synchronization.
     - Monitors are not intended for use in multiple threads.

58. Protection of synchronized variables:
     - Use of synchronized variables requires synchronization.
     - Monitors are not intended for use in multiple threads.

59. Protection of synchronized fields:
     - Use of synchronized fields requires synchronization.
     - Monitors are not intended for use in multiple threads.

60. Protection of synchronized classes:
     - Use of synchronized classes requires synchronization.
     - Monitors are not intended for use in multiple threads.

61. Protection of synchronized objects:
     - Use of synchronized objects requires synchronization.
     - Monitors are not intended for use in multiple threads.

62. Protection of synchronized methods:
     - Use of synchronized methods requires synchronization.
     - Monitors are not intended for use in multiple threads.

63. Protection of synchronized blocks:
     - Use of synchronized blocks requires synchronization.
     - Monitors are not intended for use in multiple threads.

64. Protection of synchronized variables:
     - Use of synchronized variables requires synchronization.
     - Monitors are not intended for use in multiple threads.

65. Protection of synchronized fields:
     - Use of synchronized fields requires synchronization.
     - Monitors are not intended for use in multiple threads.

66. Protection of synchronized classes:
     - Use of synchronized classes requires synchronization.
     - Monitors are not intended for use in multiple threads.

67. Protection of synchronized objects:
     - Use of synchronized objects requires synchronization.
     - Monitors are not intended for use in multiple threads.

68. Protection of synchronized methods:
     - Use of synchronized methods requires synchronization.
     - Monitors are not intended for use in multiple threads.

69. Protection of synchronized blocks:
     - Use of synchronized blocks requires synchronization.
     - Monitors are not intended for use in multiple threads.

70. Protection of synchronized variables:
     - Use of synchronized variables requires synchronization.
     - Monitors are not intended for use in multiple threads.

71. Protection of synchronized fields:
     - Use of synchronized fields requires synchronization.
     - Monitors are not intended for use in multiple threads.

72. Protection of synchronized classes:
     - Use of synchronized classes requires synchronization.
     - Monitors are not intended for use in multiple threads.

73. Protection of synchronized objects:
     - Use of synchronized objects requires synchronization.
     - Monitors are not intended for use in multiple threads.

74. Protection of synchronized methods:
     - Use of synchronized methods requires synchronization.
     - Monitors are not intended for use in multiple threads.

75. Protection of synchronized blocks:
     - Use of synchronized blocks requires synchronization.
     - Monitors are not intended for use in multiple threads.

76. Protection of synchronized variables:
     - Use of synchronized variables requires synchronization.
     - Monitors are not intended for use in multiple threads.

77. Protection of synchronized fields:
     - Use of synchronized fields requires synchronization.
     - Monitors are not intended for use in multiple threads.

78. Protection of synchronized classes:
     - Use of synchronized classes requires synchronization.
     - Monitors are not intended for use in multiple threads.

79. Protection of synchronized objects:
     - Use of synchronized objects requires synchronization.
     - Monitors are not intended for use in multiple threads.

80. Protection of synchronized methods:
     - Use of synchronized methods requires synchronization.
     - Monitors are not intended for use in multiple threads.
Semaphores

Conditional Communication & Synchronization

Centralized synchronization

Synchronization by protected objects

(Condition synchronization: entries & barriers)

Withdrawing entry calls

(Withdrawing entry calls)

Barrier evaluation

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

• on Buffer.Get (Some_Data)
• on leaving a protected procedure or entry

-- meanwhile try something else

10.0;
delay
10.0;
-- do something after 10 s.

Get;

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

entry
when
is
select

select
;

then
abort
leaving a protected procedure or entry

else
all potentially altered barriers with tasks queued up on them are re-evaluated.

-- do something else

Put;

while inside a protected object or on leaving a protected function.

end

select

end

Bounded_Buffer;

324

Shared memory based synchronization

P.O.S.

• Off-the-shelf components available
• Cost-effective when the full software components are not necessary
• Can be reused over projects
• Design choices determined in advance at the software semantic level
• Causal
• Available

Java

• Object technology and modularity
• Concurrency code is more straightforward
• Shared data and code are easy to manage
• Universal object-oriented mechanism
• Advanced features implemented
• Memory management of object-oriented features and fine-grained resource protection

Cox ID

• Modular object-oriented approach
• Data structure manipulation and lifecycle management
• Condition dispatch related to object state transition
• Meta-object protocol to object association

© 2018 Uwe R. Zimmer, The Australian National University page 323 of 757 (chapter 3: "Communication & Synchronization" up to page 368)
**Communication & Synchronization**

### High Performance Computing

Synchronization in large scale concurrency

High Performance Computing (HPC) emphasizes on keeping as many CPU nodes busy as possible.

- Avoid contention on space memory.
- Data is migrated to individual processes rather than processes accessing data in data structures stored within the CPU cache to approximate “lock-free”, but there is still need to ensure consistent actions.

**Traditionally this has been implemented using the Message Passing Interface (MPI) with explicit message passing to address space.

- Avoid application implementation of specific contexts.
- Message passing is often preferred because it does not have the independence of computation and now there is a need for memory integrity mechanisms in shared memory parallel spaces.

---

### Message-based Synchronization

#### Message-based Synchronization

**Message protocols**

**Remote invocation**

- **Remote invocation (no results)**
  - **Simulate by asynchronous messages**
  - **Try different results from parallel back**
  - **Build in parallel processes or a bulk synchronous at the thread level**

**Synchronous message**

- **Synchronous message (sender waiting)**
  - **Delay the message until**
  - **Results become available**
  - **Send back new message:

---

### Synchronization

#### Synchronization in Chapel

**Message-based synchronization**

- **Message protocols**
  - **Asynchronous message (simulated by synchronous messages)**
  - **Introducing an intermediate process**
  - **Intermediate result to be asynchronously messages, and lines**
  - **Intermediate results to be used as the invocation cause**
  - **offer**
  - **While processes are the threads**

#### Message-based Synchronization

**Message protocols**

**Remote invocation**

- **Remote invocation (no results)**
  - **Simulate by asynchronous messages**
  - **Try different results from parallel back**
  - **Build in parallel processes or a bulk synchronous at the thread level**

---

### Communication & Synchronization

**Shared memory based synchronization**

#### Mutual and Shared Mutex

- Mutual and Shared Mutex
  - **Mutual exclusion in scopes**
  - **Encapsulation**
  - **Guards (barriers)**
  - **Low-Level semaphores for very special cases**
  - **Full implementation of the monitor (atomic P, V ops)**
  - **Combined with the message passing**
  - **Variations in the field of explicit concurrency.

---

### Communication & Synchronization

**Shared memory based synchronization**

#### Shared memory based synchronization

- **Conditional critical regions**
  - **High-level synchronization support**
  - **Avoid contention on sparse resources.**
  - **Data is assigned to individual processes rather than processes synchronizing on data.**
  - **Data structure encapsulation**
  - **monitor methods**
  - **Flags (atomic word access)**
  - **Atomic operations in X10**

---

### Communication & Synchronization

**Message-based synchronization**

#### Message-based synchronization

**Message protocols**

**Remote invocation**

- **Remote invocation (no results)**
  - **Simulate by asynchronous messages**
  - **Try different results from parallel back**
  - **Build in parallel processes or a bulk synchronous at the thread level**
Communication & Synchronization

Message-based synchronization

Synchronous vs. asynchronous communications

- Network technology
- Message exchange
- Control

- Direct versus indirect:
  - Message passing
  - Rendezvous

- Asynchronous communication requires high-speed and error-free networks.

- Can both communication modes emulate each other?
  - Synchronous communication can be emulated by a combination of asynchronous messages.
  - Asynchronous communication can be emulated in many-to-many general network or bus systems.

Communication medium:

- Direct: point-to-point
- Indirect: broadcast, multicast

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

- One-to-one buffer, queue, synchronization
- One-to-many multicast

Can both communication modes emulate each other?

- Synchronous communication can be emulated by a combination of asynchronous messages.
- Asynchronous communication can be emulated in many-to-many general network or bus systems.

Message-based synchronization

- Message structure
- Message-based synchronization (Ada)

- Message passing systems examples:

Communication & Synchronization

Message-based synchronization in CHILL

- CHILL is the VTH High Level Language
- CHILL is the VTH High Level Language

- Communications functions not be taken care of distributed environment.
- Communication is ensured by means of a 'channel'
- Communication is ensured by means of a 'channel'

- Message-based synchronization in Occam2

- Message-based synchronization in Ada

- Message-based synchronization in Erlang

- Message-based synchronization in Java

- Message-based synchronization in Go
Message-based synchronization in Ada

Message-based synchronization in Ada

(Message-based synchronization in Ada)

(Message-based synchronization in Ada)

(Message-based synchronization in Ada)

(Message-based synchronization in Ada)

(Message-based synchronization in Ada)

(Summary)

Communication & Synchronization

Some things to consider for task-entries:
- In contrast to protected-object entries, task-entry bodies can call all other blocking operations.
- Accept statements can call nested functions (but need to be different).
- Documentation includes more characteristic functions.
- Accept statements can call a dedicated exception handler similar to other code blocks.
- Parameters, which are checked during the rendezvous phase are propagated to all involved tasks.
- Parameters can return direct values of parameters, but only as constants.
- Synchronization and object orientation may use blocking operations and re-queuing.
- Message-based synchronization
  - Addressing modes
  - Message size
  - Examples

Communication & Synchronization

Shared memory based synchronization
- Flags, condition variables, semaphores,
  - Internal 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
    - Addressing modes
    - Message size
    - Examples