Motivation

Side effects
Operations have side effects which are visible ...

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

  or

- outside the current process

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

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

What’s the worst that can happen?

Condition synchronization by flags

Towards synchronization
Condition synchronization by flags

Assumption: word-access atomicity:

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

\[ x = x \oplus \bar{x} \]

will result in either \[ x = 0 \] or \[ x = 510 \] - and no other value is ever observable

References for this chapter

- Chapel 1.11.0 Language Specification Version 0.97
- ISO/IEC 8652:201x (E)
- Chapel 1.11.0 Language Specification Version 0.97
- C, POSIX — DiPietro
- Ada — Pilutti
- POSIX
- Modiato, L. — DiPietro, Hoare, ...
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 shared-size atomic variable operating as a flag to indicate synchronization conditions.
- the variable $S$ is a "flag" to indicate synchronization conditions:

``` ADA
var Flag : boolean := false;
process P1; 
end P1;
```

Sequence of operations: $A \rightarrow B_2$ or $A \rightarrow Y_2$ are not suitable for general mutual exclusion in critical sections!

More powerful synchronization operations are required for critical sections!

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 shared-size atomic variable operating as a flag to indicate synchronization conditions.
- Memory flag method is ok for single condition synchronization, but ...
  - ... is not suitable for general mutual exclusion in critical sections!
  - ... busy-waiting is required to poll the synchronization condition!

Towards synchronization

Basic synchronization by Semaphores

Basic definition (Dijkstra 1968)

- set of processes agree on a variable
- an atomic operation

``` ADA
signal (sync);
```

Sequence of operations: $A \rightarrow B_1$ or $A \rightarrow Y_1$ are not suitable for general mutual exclusion in critical sections!

Towards synchronization

Mutual exclusion by semaphores

``` ADA
var sync : semaphore := B;
process P1; 
process P2; 
end P1; 
end P2; 
```

Sequence of operations: $A \rightarrow B_1$ or $A \rightarrow Y_1$ are not suitable for general mutual exclusion in critical sections!

Towards synchronization

Semaphores in Ada

package Ada.Synchronous_Task_Control is 
  type Suspension_Object is limited private;
  procedure Set_True (S : in out Suspension_Object);
  procedure Set_False (S : in out Suspension_Object);
  function Get_Current_State (S : in Suspension_Object) return Boolean;
  function Suspended (S : in Suspension_Object) return Boolean;
private
  -- not specified by the language
  end Ada.Synchronous_Task_Control;
```

This is a potentially delaying operation

```
end Ada.Synchronous_Task_Control;
```

Program_Error will be raised with a second task trying to suspend itself

```
end P1; 
end P2; 
```

only one task can be blocked at Suspend_Until_True!

```
end Ada.Synchronous_Task_Control;
```

```
begin 
end
```

no queues! minimal run-time overhead
Towards synchronization

Malicious use of "queueless semaphores"

with Ada.Synchronous.Task_Control; use Ada.Synchronous.Task_Control;

X, Y : Suspension.Object;

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

\[ \text{Will result in a deadlock (assuming no other Set_TRUE calls.)} \]

Semaphores in POSIX

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

busy = 0;

if sem_post (&cond[P]);

\[ \text{Will potentially result in a deadlock (with general semaphores).} \]

sem_wait (&cond[P]);

int sem_getvalue (sem_t *sem_location, int *value);

Semaphores in Java (since 2004)

void acquire            (int permits)          void acquireUninterruptibly (int permits)          boolean tryAcquire             ()          boolean tryAcquire             (int permits, long timeout, TimeUnit unit)

void acquireUninterruptibly (int permits)          boolean tryAcquire             ()          boolean tryAcquire             (int permits, long timeout, TimeUnit unit)

Semaphores are
well formed synchronization blocks and synchronization conditions.

Well-formed synchronization blocks and synchronization conditions.

all concurrent languages and environments offer efficient and higher-abstraction synchronization methods)

Processes are prohibited from entering a critical region, when

Potential livelocks

Processes are prohibited from entering a critical region, when

Potential livelocks

Processes are prohibited from entering a critical region, when

Potential livelocks

Processes are prohibited from entering a critical region, when

Potential livelocks

Processes are prohibited from entering a critical region, when

Potential livelocks

Processes are prohibited from entering a critical region, when

Potential livelocks
Monitors with condition synchronization

Basic idea:
- Collect all operations and data-structures shared in critical regions in one place, the monitor.
- Functionality is encapsulated.
- Access to the monitor is controlled.
- Prevent access to data-structures other than by the monitor procedures and functions.
- Access is realized through guarded operations.

**Hoare-monitors**:

```plaintext
var protect

begin

end
```

Condition variables are implemented by semaphores (wait and signal).

Basic idea:
- Queues for tasks suspended on condition variables are realized.
- A suspended task releases its lock on the monitor, enabling another task to enter.
- Collect all operations and data-structures shared in critical regions in one place, the monitor.
- The signalling and the waiting process are both active in the monitor!

**Program:**

```plaintext
begin
  BUF := array[...]
of integer;
  NumberInBuffer := 0;
  top := 0;
  base := 0;
end
```

Prohibit access to data-structures, other than by the monitor procedures and functions.

Assure mutual exclusion of all monitor procedures and functions.

A suspended task releases its lock on the monitor, enabling another task to enter.
Monitors in Java
(by means of language primitives)

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

1. Synchronized methods and code blocks:
   - all methods and code blocks which are using the synchronized tag are mutually exclusive with respect to the addressed class.

2. Notification methods:
   - wait, notify, and notifyAll can be used only in synchronized regions and are waking any or all threads, which are waiting in the same synchronized object.

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 standard method can break a Java monitor and enter at any time.
   - Methods outside the monitor-object can synchronize at this object.
   - Static data shared between all objects of a class or access to static data needs to be synchronized with all objects of a class.
   - Synchronizes either static synchronized blocks: synchronized (this.getClass()), or static methods public synchronized static method( ).

2. Notification methods: wait, notify, and notifyAll
   - wait suspends the thread and releases the local lock only or meted wait calls will keep all enclosing locks.
   - notify and notifyAll do not release the lock!
   - Java does not require any specific release order like acquire-for-wait-suspended threads or threads are not passed at this level (as in opposition to RT-Java).
   - There are no explicit conditional variables associated with the monitor or data, or notified threads need to wait for the lock to be released, and to re-evaluate its entry condition.

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

public class ReadersWriters {
    private int readers = 0;
    private int writers = 0;
    private boolean writing = false;

    private ConditionVariable OkToRead = new ConditionVariable();
    private ConditionVariable OkToWrite = new ConditionVariable();

    public void reader() throws InterruptedException {
        synchronized (OkToRead) {
            if (readers > 0) {
                writing = true;
                OkToWrite.waitsleep = true;
            } else {
                writing = true;
                OkToWrite.waitsleep = false;
            }
            OkToRead.notifyAll(); // wake up one writer
            writing = false;
            OkToRead.notifyAll(); // wake up all readers
            readers = waitingReaders;
            waitingReaders = 0;
        }
    }
}

public class ReadersWriters {
    private int readers = 0;
    private int writers = 0;
    private boolean writing = false;

    private ConditionVariable OkToRead = new ConditionVariable();
    private ConditionVariable OkToWrite = new ConditionVariable();

    public void reader() throws InterruptedException {
        synchronized (OkToRead) {
            if (readers > 0) {
                writing = true;
                OkToWrite.waitsleep = true;
            } else {
                writing = true;
                OkToWrite.waitsleep = false;
            }
            OkToRead.notifyAll(); // wake up one writer
            writing = false;
            OkToRead.notifyAll(); // wake up all readers
            readers = waitingReaders;
            waitingReaders = 0;
        }
    }

    public void writer() throws InterruptedException {
        synchronized (OkToWrite) {
            if (writers > 0) {
                writing = false;
                OkToWrite.notifyAll(); // wake up one writer
                writing = true;
                OkToRead.waitsleep = true;
            } else {
                writing = true;
                OkToWrite.waitsleep = false;
            }
            OkToWrite.notifyAll(); // wake up all writers
            writers = waitingWriters;
            waitingWriters = 0;
        }
    }
}
Monitors in Java

Multiple-readers-one-writer example usage of external conditional variable:

```java
public void StopRead () {
    synchronized (OkToWrite) {
        synchronized (this) {
            if (writing | waitingWriters > 0) {
                waitingReaders++;
            } else {
                waitingWriters--;
            }
            OkToWrite.notify ();
        }
        readers--;
        OkToRead.wantToSleep = true;
    }
    if (OkToRead.wantToSleep) OkToRead.wait ();
}
```

Monitors in Java

Centralized synchronization

Criticism of monitors

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

Synchronization by protected objects

Combine

- **the encapsulation feature of monitors**
- the coordinated entries of conditional critical regions

- **Protected objects**
  - All accessed data and operations are encapsulated
  - Operations are mutually exclusive (with exceptions for read-only operations)
  - Guards (predicates) are syntactically attached to entries
  - No protected data is accessible (other than by the defined operations)
  - Semantics inside operations is guaranteed by queuing (according to their priorities)
  - Semantics across all operations is guaranteed by the "internal progress first" rule
  - Re-blocking provided by re-queuing to entries (no internal condition variables).

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 heritage of the insecure C language and its primitive, error prone library of threads methods.

"Per Brinch Hansen was one of a handful of computer pioneers who was responsible for advancing both operating system development and concurrent programming from all his techniques in systems engineering disciplines." (from his IEEE 2002 Computer Pioneer Award)
**Centralized synchronization**

**Synchronization by protected objects**

(Condition synchronization: entries & barriers)

Condition synchronization is realized in the form of protected procedures combined with boolean predicates. An entry family, or call entry family, can be defined as an entry family (see entry family: requeue & private entries).

```ada
package Bounded_Buffer is
  entry Get (Item: out Data_Item) when Num > 0 is
    begin
      Item := Buffer (First);
      First := First + 1;
      Num := Num - 1;
    end;
end;
```

```ada
package Body_Bounded_Buffer is
  entry Put (Some_Data: in Data_Item) when Num < Buffer_Size is
    begin
      Last := Last + 1;
      Buffer (Last) := Some_Data;
      Num := Num + 1;
    end;
end;
```

(Withdrawing entry calls)

```ada
package Body_Bounded_Buffer is
  entry Probe when Probe'count > 0 is
    begin
      -- try to enter for 10 s.
    end;
end;
```

**Synchronization by protected objects**

(Condition synchronization: entries & barriers)

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

```ada
protected body Broadcast is
  procedure Send (Message: in Broadcast.Message) is
    private
      New_Message : Broadcast.Message;
    begin
      New_Message := Message;
      -- do something else
    end;
end;
```

**Synchronization by protected objects**

(Barrier evaluation)

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

- on creating a protected object.
- on having a protected procedure or entry.

All barriers are evaluated according to the initial values of the internal, protected data.

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

- on creating a protected object.
- on having a protected procedure or entry.

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

**Synchronization by protected objects**

(Operations on entry queues)

```ada
package Body_Broadcast is
  procedure Send (Message: in Broadcast.Message) is
    private
      New_Message : Broadcast.Message;
    begin
      New_Message := Message;
      -- do something else
    end;
end;
```

**Synchronization by protected objects**

(Operations on entry queues)

```ada
package Modes is
  type Mode_T is (Takeoff, Ascent, Cruising, Landing);
protected body Modes is
  procedure Set_Mode (Mode: Modes.Mode_T) is
    private
      Current_Mode : Modes.Mode_T := Takeoff;
      End_Mode : Modes.Mode_T := Mode;
    begin
      if Current_Mode /= Mode then
        -- do something else
      end;
    end;
end;
```

**Synchronization by protected objects**

(Entry families)

```ada
package body Modes is
  type Mode_T is (Takeoff, Ascent, Cruising, Landing);
protected body Modes is
  procedure Set_Mode (Mode: Modes.Mode_T) is
    private
      Current_Mode : Modes.Mode_T := Takeoff;
      End_Mode : Modes.Mode_T := Mode;
    begin
      if Current_Mode /= Mode then
        -- do something else
      end;
    end;
end;
```
Centralized synchronization

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

Conditional
Semaphores

Centralized synchronization

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

All code inside a protected procedure, function or entry is bound to non-blocking operations. Thus the following operations are prohibited:

- Entry call statements
- `if` statements
- `begin` statements
- `end` statements
- `= 0` statements
- `= 0` statements

Example:

```cpp
type Urgency is (urgent, not_so_urgent);
type Server_Farm is (primary, secondary);
protected Pre_Filter is
  entry Reception (0 : Urgency);
private
  entry Server (Server_Farm) (0 : Urgency);
end Pre_Filter;

If U = urgent and Server (primary)'count = 0 then
  requeue Server (primary);
else
  requeue Server (secondary);
end if;

select
  protected Pre_Filter;
end;

entry Server (Server_Farm) (0 : Urgency) when True is
begin
  -- Might try something even more useful
end Server;

end Pre_Filter;
```

The `Pre_Filter` entry releases the current lock!
Communication & Synchronization

Shared memory based synchronization

- Full implementation of the
  - Ada/No Erlang concept

The term *monitor* appears in many other concurrent languages, yet it is usually not associated with an actual language primitive.

Current developments

Atomic operations in X10

- Unconditional atomic blocks are guaranteed to be non-blocking, which means that they cannot be blocked and need to be implemented using call-backs
- Conditional atomic blocks can also be used as a pure notification system (similar to the Java/Java methods)
- Parallel statements incl. parallel i.e. unaligned top-of-the-stack
- 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

Chapel offers a variety of concurrent primitives:

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

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

Message-based synchronization

Synchronous message

- Synchronous message (sender waiting)
- Synchronous message (receiver waiting)

Asynchronous message

- Asynchronous message (sender waiting)
- Asynchronous message (receiver waiting)
Communication & Synchronization

Message-based synchronization

Asyncynchronous message (simulated by asynchronous messages)

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

Synchronous message (simulated by asynchronous messages)

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

Remote invocation

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

Remote invocation (no results)

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

Remote invocation (no results)

Simulate one synchronous message
- Processes are never actually synchronized

Remote invocation (no results)

Simulate two asynchronous messages
- Processes are never actually synchronized

Synchronous vs. asynchronous communications

Purpose: 'synchronization'
- Synchronous messages / remote invocations
- Synchronous pass coupling only
- Synchronous message passing in distributed systems requires hardware support.
- Asynchronous messages 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).

Addressing (name space)

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

Asymmetrical addressing:
- send <message> to "<process-name>" wait for <message> from "<mailbox>"
- 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>direct</td>
</tr>
<tr>
<td>one-to-many</td>
<td>multicast</td>
</tr>
<tr>
<td>many-to-one</td>
<td>broadcast</td>
</tr>
<tr>
<td>many-to-many</td>
<td>general network</td>
</tr>
</tbody>
</table>

Communications systems are often outside the typed language environment.

Message structure (Ada)

```ada
package Ada.Streams is
  pure (Streams);
  abstract tagged limited private;
  private
    -- Communication system is often outside the typed language environment.
    type Stream_Element is mod implementation-defined;
    Connections Functionality
    type Stream_Element_Array is array <> of Stream_Element;
    type Stream_Element_Offset is range 0..Stream_Element_Offset'Last;
    one-to-many multicast
    procedure Read (…)
    one-to-all broadcast
    procedure is abstract
    many-to-one local server, synchronization
    procedure
    many-to-many general network- or bus-system
    …
end Ada.Streams;
```

Message-passing systems examples:

- OCCAM: "messages-queue" as ordered indirect (asymmetrical | symmetrical) synchronous
- Java: no message passing system defined
- Erlang: no message passing system defined
- CHILL: "buffer" | "pipe"
- C++: "channel"

Reading and writing values of any subtype `S` of a specific type to a stream:

- procedure `S_Write` (Stream : access Ada.Streams.Root_Stream_Type'Class; Item : in T)
- procedure `S_Write` (Stream : access Ada.Streams.Root_Stream_Type'Class; Item : in T'Class)
- procedure `S_Read` (Stream : access Ada.Streams.Root_Stream_Type'Class; Item : out T)
- procedure `S_Read` (Stream : access Ada.Streams.Root_Stream_Type'Class; Item : out T'Class)

Reading and writing values, bounds and discriminants of any subtype `S` to a stream:

- procedure `S_Output` (Stream : access Ada.Streams.Root_Stream_Type'Class;)
- function `S_Input` (Stream : access Ada.Streams.Root_Stream_Type'Class) return T

Message-based synchronization in OCCAM2

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

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

```occam2
CHAN OF INT SensorChannel:
PAR   INT reading:   SEQ i = 0 FOR 1000      SEQ
INT data:   SEQ i = 0 FOR 1000      SEQ
send SensorChannel (reading);   -- generate reading
SensorChannel ? data
-- employ data
```

Message-based synchronization in CHILL

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

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

```chill
CHAN of INT SensorChannel:
int SensorChannel;   -- generate reading
SensorChannel ? data
int SensorChannel;   -- employ data
```

Essential OCCAM2 keywords

- `ALT` parallelism, nondeterminism
- `PAR` parallelism
- `SEQ` sequentialism
- `CASE` selective execution
- `IF` conditional execution
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.

Key points:
- Strong support for concurrency, synchronization, and communication (monitors, buffered message passing, synchronous channels).
- Message-based synchronization in CHILL is the "CCITT High Level Language", where CCITT is the Comité Consultatif International Télégraphique et Téléphonique.

CHILL development began in 1973 and was standardized in 1979.

Message-based synchronization in Ada

Ada supports remote invocations ("extended rendezvous") in form of:
- Entry points in tasks
- Full set of parameter profiles supported

- If the local and the remote task are on different architectures, or if an intermediate communication system is employed then parameters incl. bounds and discriminants are "tunnelled" through byte-stream formats.

Synchronization:
- Both tasks are synchronized at the beginning of the remote invocation ("extended rendezvous")
- The calling task is blocked until the remote routine is completed ("extended rendezvous")

Message-based synchronization in Ada (Rendezvous)

- Call other blocking operations.
- Guard evaluation times, nested monitor calls, deadlocks.
- Synchronization and object orientation, blocking operations and re-queuing.

Message-based synchronization in Ada (Extended rendezvous)

- Contained in Ada 95 and later.
- Parameters cannot be direct 'count' parameters, but can be access-types.
- Private entries (accessible for internal tasks) are supported.

Summary

- Shared memory based synchronization
  - Flags, condition variables, semaphores, conditional critical regions, monitors, protected objects.
  - Guard evaluation times, re-detonation calls, deadlock, simultaneous reading, spanner management.
  - Synchronization and object orientation, blocking operations and re-squeuing.

- Message based synchronization
  - Synchronization models
  - Addressing modes
  - Message structures
  - Examples...