Concurrent
• things happen at the same time
Parallel
• several computation units in one computer

Benefits of concurrency:
– speed-up of computation (serializable operation sequences)
– expressive power (concurrent nature of the problem)

Concurrency at different levels:
– machine instructions
– programming language statements
– processes (independent subroutines)
– applications

Virtual and real concurrency
– logical computing units (processes)
– physical computing units (processors)

Serializable:
\[
\begin{align*}
a &= 1; \\
b &= 2; \\
a &= 1; \\
b &= a \times 2;
\end{align*}
\]

Not serializable:
\[
\begin{align*}
a &= 1; \\
b &= 2; \\
a &= 1; \\
b &= a \times 2;
\end{align*}
\]
Processes and threads

• Process
  – provided by operating system
  – enables programs to run concurrently
  – separate address space

• Thread
  – lightweight process
  – sequence of instructions that does not depend on other threads
    • each thread has an own program counter and stack
  – threads belonging to the same process share the same address space

• Interaction
  – shared memory or message passing

Different languages have different terminology
Synchronization

• Control of the mutual execution order of the processes
• Co-operation synchronization
  – process A waits until process B terminates; A cannot run before that
    • e.g. producer-consumer – problem
• Competition synchronization
  – processes need the same resource that can be used only by one process at a time
    • e.g. writing to shared memory
Coroutines

- The oldest concurrency mechanism in programming languages
  - Simula67, Modula-2

- Quasi-concurrency
  - single processor
  - processor switches from one process to another are described explicitly
  - programmer acts as a scheduler

- Becoming popular again (Python generators)
MODULE Program;
FROM SYSTEM IMPORT PROCESS, NEWPROCESS, TRANSFER, ..;
VAR v1, v2, main: PROCESS;

PROCEDURE P1;
BEGIN
    ...
    TRANSFER ( v1, v2 );
    ...
END P1;

PROCEDURE P2;
BEGIN
    ...
    TRANSFER ( v2, v1 );
    ...
    TRANSFER ( v1, v2 );
    ...
END P2;

BEGIN
    NEWPROCESS ( P1, ..., v1 );
    NEWPROCESS ( P2, ..., v2 );
    TRANSFER ( main, v1 );
END;

Coroutines in Modula-2
Semaphores

- Enable mutual exclusion
- Integer variables with operations P (wait) and V (signal):

P ( S ): \[\text{if } S > 0 \text{ then } \]
\[ S := S - 1 \]
\[\text{else} \]
\[\text{set this process to wait } S \]
\[ V ( S ): \text{if some process is waiting for } S \text{ then } \]
\[\text{let one continue (its execution)} \]
\[\text{else} \]
\[ S := S + 1 \]

- P and V are atomic operations
- General or binary semaphore
Example on semaphores

Buf: array [ 1..SIZE ] of Data
NextIn, NextOut: Integer := 1, 1
Mutex: semaphore := 1
EmptySlots, FullSlots: semaphore := SIZE, 0

procedure Insert ( d: Data )
P ( EmptySlots )
P ( Mutex )
{ Buf [ NextIn ] := d
  NextIn := NextIn mod SIZE + 1
  V ( Mutex )
  V ( FullSlots )
}

function Remove: Data
P ( FullSlots )
P ( Mutex )
{ d : Data := Buf [ NextOut ]
  NextOut := NextOut mod SIZE + 1
  V ( Mutex )
  V ( EmptySlots )
  return d
}
Monitors

• More advanced controllers of common data than semaphores
• Make use of module structure
  – encapsulation / information hiding
  – abstract data types
• Encapsulation mechanism
  – mutual exclusion of operations
  – only one process at a time can execute the operations of the module (process has the lock of the monitor)
• Synchronization (co-operation)
  – enabled with signals (conditions)
• Wait set of monitor
  – consists of processes that want to execute the operations of the monitor
Monitors

- Monitor type (module)
- Signal type has the following operations:

<table>
<thead>
<tr>
<th>Operation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>wait (S)</code></td>
<td>set the calling process to wait for <code>S</code></td>
</tr>
<tr>
<td><code>release monitor</code></td>
<td>exit from the monitor routine (and release the monitor)</td>
</tr>
<tr>
<td><code>continue (S)</code></td>
<td>let some other process to continue</td>
</tr>
</tbody>
</table>

- Shared data inside a monitor
  - the semantics of monitor type prevents parallel access to the data structures

- Programmer takes care of the co-operation synchronization
monitor BufferType
imports Data, SIZE
exports insert, remove

var Buf: array [ 1 .. SIZE ] of Data
Items: Integer = 0
NextIn, NextOut: Integer := 1, 1
FullSlots, EmptySlots: signal

procedure entry Insert ( d: Data )
  if Items = SIZE then wait ( EmptySlots )
  Buf [ NextIn ] := d
  NextIn := NextIn mod SIZE + 1
  Items := Items + 1
  continue ( FullSlots )

function entry Remove: Data
  if Items = 0 then wait ( FullSlots )
  d : Data := Buf [ NextOut ]
  NextOut := NextOut mod SIZE + 1
  Items := Items – 1
  continue ( EmptySlots )
  return d
Threads (Java)

- Concurrency is enabled by
  - inheriting the class `Thread`
  - implementing the interface `Runnable`
  - thread for main program is created automatically

- Thread execution
  - the functionality of the thread is written in `run` operation
  - execution begins by calling `start`, when the system calls `run`

- Other operations of `Thread`
  - `sleep`: locks the thread (for milliseconds)
  - `yield`: thread gives up the rest of its execution time
Synchronization (Java)

- Every object has a lock
  - prevents synchronized operations from being executed at the same time
- When a thread calls a synchronized operation of an object
  - thread takes control of the lock of the object when
    - other threads cannot call any synchronized operation of the object
  - thread releases the lock when
    - operation is finished or the thread waits for a resource
Ways of synchronization in Java

- **Competition synchronization**
  - synchronized operation is executed completely before starting to execute any other operation

- **Co-operation synchronization**
  - waiting for the access to execute (*wait*)
  - notifying the other threads that the event they have been waiting has happened (*notify* or *notifyAll*)
Creation of threads

- Co-begin
- Parallel loops
- Launch-at-elaboration
- Fork-join
- Others:
  - implicit receipt, early-reply
Co-begin

non-deterministic:  parallel:

begin
  a := 3,
  b := 4
end

par begin
  a := 3,
  b := 4
end

par begin
  p (a, b, c),
  begin
    d := q(e, f);
    r(d, g, h)
  end,
  s(i, j)
end

p(a, b, c) d := q(e, f) s(i, j)
r(d, g, h)

TUT Pervasive Computing

Principles of programming languages
Maarit Harju / Matti Kintala / Henri Hansen

Algol68
Occam
Parallel loops:

SR: \[
\begin{align*}
\text{co} & (i := 5 \text{ to } 10) \rightarrow\\
p & (a, b, i)\\
\text{oc}
\end{align*}
\]

Occam: \[
\begin{align*}
\text{par} & i = 5 \text{ for } 6\\
p & (a, b, i)
\end{align*}
\]

Fortran95: \[
\begin{align*}
\text{forall} & (i = 1 : n - 1)\\
A & (i) = B (i) + C (i)\\
A & (i + 1) = A (i) + A (i + 1)\\
\text{end forall}
\end{align*}
\]

Launch-at-elaboration:

Ada: \[
\begin{align*}
\text{procedure} & P \text{ is}\\
& \text{task} T \text{ is}\\
& \begin{align*}
& \text{...}\\
& \text{end } T;\\
& \text{begin} \quad \text{-- } P\\
& \quad \text{...}\\
& \text{end } P;
\end{align*}
\end{align*}
\]
Previous ways

nested structure

Fork-join

fork

join

more general structure
Fork-join

**Ada:**

```ada
task type T is
    ...
begin
    ...
end T;

pt: access T := new T;
```

**Modula-3:**

```modula-3
t := Fork ( c );
    ...
Join ( t );
```

**Java:**

```java
class myThread extends Thread {
    ...
    public void myThread ( ... ) { ... }
    public void run ( ) { ... }
}
...
myThread t = new myThread ( ... );
```

```java
t.start ( );

t.join ( );
```
Message passing

• No shared memory
  – e.g. distributed systems
• Nondeterminism enables fairness
  – Dijkstra’s guarded commands
• Synchronized message passing
  – sender is waiting for the response
    • Ada83
• Asynchronized message passing
  – sender continues its processing without waiting
    • Ada95
Tasks in Ada

- Resembles packages (syntactically)
  - specification and body
    - specification has no private part
  - active unit (unlike a package)

- Task interaction
  - based on rendezvous (meeting) that is enabled with entries
    - entry defines the services of the task
    - server/client
  - synchronized message passing
    - waiting is passive
Entries and rendezvous (Ada)

- Meeting (rendezvous) is enabled by `accept` clause
  - a statement, although it looks like a procedure
- Meeting is not symmetric:
  - caller must know the entry
  - task having the entry does not know the caller
- Task (entry) call creates a place for the meeting
  - actor (caller, client) waits for a certain task
  - server (task having the entry) waits for any task
  - normal parameter passing (in, out, in out)
Selective rendezvous (Ada)

- **Entry holder**
  - lack of clients at one entry must not prevent service at other entries
  - choose an entry with clients waiting

- **Entry caller**
  - waiting for a certain service may not be reasonable
  - if the entry holder is not ready, do something else

```plaintext
select
   accept P1 ... end P1;
   ... statements ...
or
   accept P2 ... end P2;
   ... statements ...
or
   ...
end select;
```

```plaintext
select
   P (...);
   ... statements ...
or
   ... statements ...
end select;
```
Conditions in accept clause (Ada)

```ada
task body Buffer is
  N: constant := 10;
  Contents: array ( 1..N ) of Integer;
  NextIn, NextOut: Integer range 1..N;
  Items: Integer range 0..N;
begin
  NextIn := 1; NextOut := 1; Items := 0;
loop
  select
    when Items < N =>
      accept Insert ( X: in Integer ) do
        Contents ( NextIn ) := X;
      end Insert;
      NextIn := ( NextIn mod N ) + 1;
      Items := Items + 1;
    or
    when Items > 0 =>
      accept Remove ( X: out Integer ) do
        X := Contents ( NextOut );
      end Remove;
      NextOut := ( NextOut mod N ) + 1;
      Items := Items - 1;
  end select;
  end loop;
end Buffer;
```

task Buffer is
  entry Insert ( X: in Integer );
  entry Remove ( X: out Integer );
end Buffer;
```
Ways of synchronization in Ada

- **Competition synchronization**
  - semantics of `accept` clause
  - statements following `accept` clause
    - executed after the actor has left rendezvous
    - before serving the next actor

- **Co-operation synchronization**
  - guard (`when` condition)
    - several conditions can hold true at the same time
  - c.f. Dijkstra’s guarded commands
Ada95

- Changes to Ada83:
  - protected objects
    - due to the complexity of rendezvous and its inefficient implementation
    - not tasks but rather monitors
  - asynchronic message passing

```
select
  -- entry call or delay statement
  -- Ada code
then abort
  -- Ada code
end select;
```
Asynchronous message passing (Ada95)

Entry call as a triggering statement:

Delay statement as a triggering statement:

```ada
loop
  select
    Terminal.Wait_For_Interrupt;
    Put_Line ("Interrupted");
  then abort
    -- This will be abandoned upon
    -- terminal interrupt
    Put_Line ("-> ");
    Get_Line ( Command, Last );
    Process_Command ( Command ( 1..Last ) );
  end select;
end loop;

select
delay 5.0;
  Put_Line ("Calculation does not converge");
then abort
  -- This calculation should finish in 5 seconds,
  -- if not, it is assumed to diverge
  Horribly_Complicated_Function ( X, Y );
end select;
```
Semantics of asynchonous accept clause (Ada95)

- Triggering part and abortable part are executed in parallel
  - triggering statement starts its execution, and abortable part follows immediately
- If abortable part completes, triggering part is cancelled
- If triggering statement completes (other than due to cancellation), abortable part is aborted and other statements of the triggering part are executed
Futures and promises

- Upon parallel call, how to pass a value to the caller that does not wait?

- *Future* represents a value that can be received later

- If the value of the future is read/used, the reader will have to wait

- Future must report, if the result is available

- *Promise* is a single assignment container (or function), which sets the value of the future
Futures (C++11)

```cpp
#include <future>
using namespace std;

int factorial ( int n ) {
    int res = 1;
    for ( int i = n; i > n; i-- )
        res *= i;
    return res;
}

int main ( ) {
    int x;
    std::future<int> fu = std::async ( factorial, 4 );
    x = fu.get ( ); // can be called only once
    return 0;
}
```
#include <future>

```cpp
using namespace std;

int factorial ( std::future<int>& f ) {
    int res = 1;
    int n = f.get ( );  // waits if needed
    for ( int i = n; i > n; i-- )
        res *= i;
    return res;
}

int main ( ) {
    int x;
    std::promise<int> p;
    std::future<int> f = p.get_future ( );
    std::future<int> fu = std::async ( factorial, std::ref ( f ) );
    p.set_value ( 4 );  // keeping the promise
    x = fu.get ( );
    return 0;
}
```