

Concurrent and Parallel Programming, Part II

Programming Languages

CS 214



Cinderella took some pictures at the ball, and sent them off to a 24-hour print service. A few days later, she got an email from the service saying that due to a technical problem, there would be a delay in receiving her photos. What did Cinderella say?

“Someday, my prints will come!”

Distributed Synchronization

Semaphores, locks, condition variables, monitors, are *shared-memory* constructs, and so *only useful on a tightly-coupled multiprocessor*.

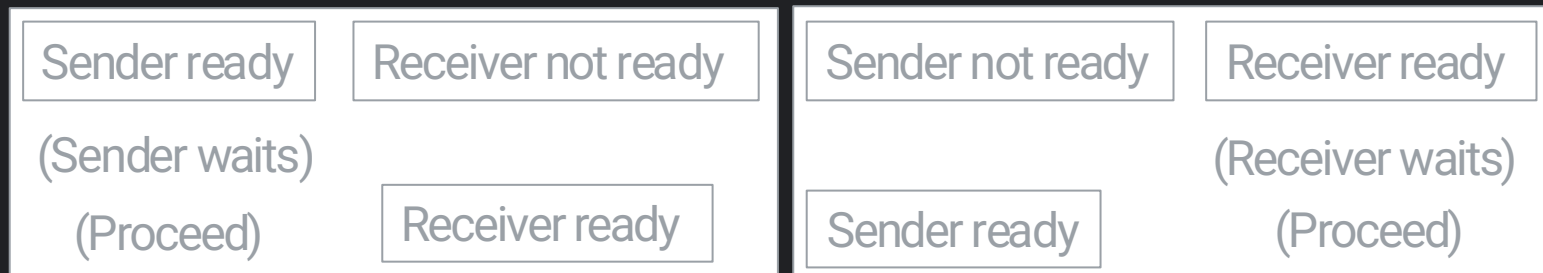
- They are of *no use* on a *distributed multiprocessor*

On a distributed multiprocessor, processes can communicate via *message-passing* – using *send()* and *receive()* primitives.

- If the message-passing system has *no storage*, then the send/receive operations must be *synchronized*:



Two scenarios...



Time



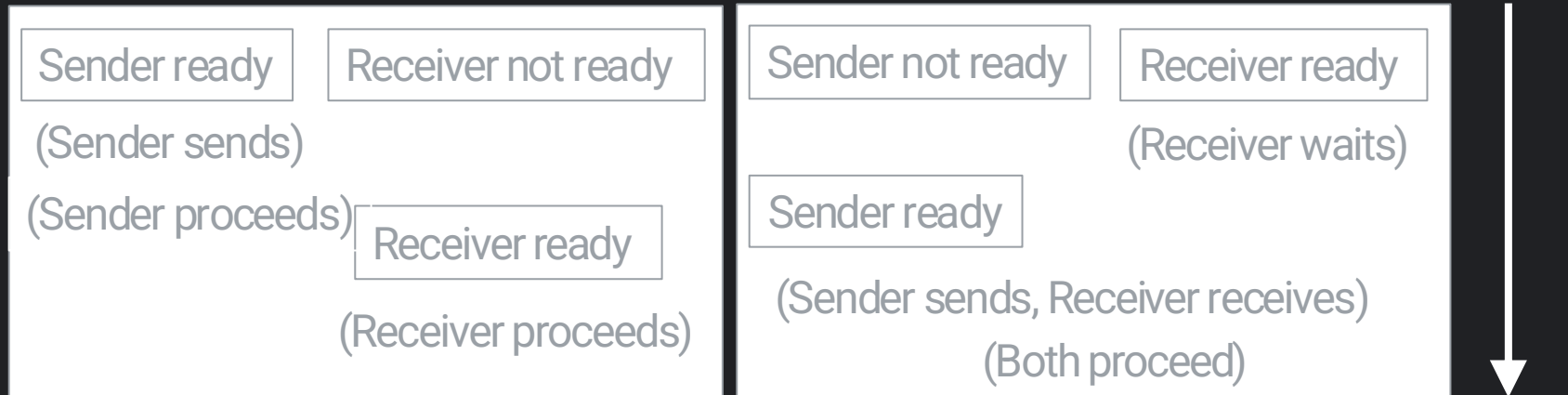
Asynchronous Communication

–If the message-passing system has *storage to buffer the message*, then the send/receive operations can proceed *asynch*



The receiver can then retrieve the message when it is ready...

Two scenarios...



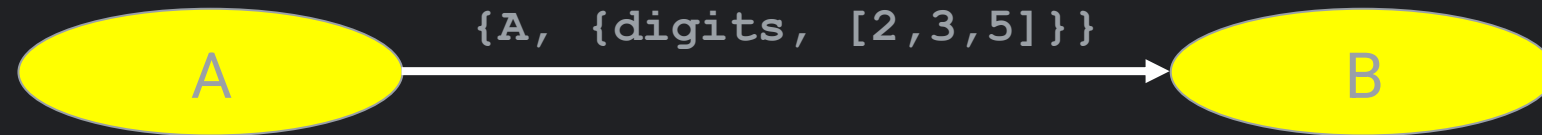
Message-buffering eliminates some (but not all) of the waiting.



Message-Passing Languages

Some languages support message-passing between

- **Erlang** is a functional language developed at Ericsson and used by Nortel, T-Mobile, Facebook (chat, WhatsApp), and 20+ others.



```
B!{self(),{digits, [2,3,5]}}
```

```
receive  
{A, {digits, nums}} ->  
  analyze(nums);  
end
```

- **Scala** is a hybrid OO+functional language used at Netflix, LinkedIn, Twitter, Tumblr, Foursquare, Sony, and other companies:

```
B! digits(2,3,5)
```

```
receive {  
  case digits(nums) =>  
    analyze(nums);  
}
```



An Ada Task

... has 3 characteristics:

- its own thread of control;
- its own execution state; and
- mutually exclusive subprograms (aka *entry procedures*)

Entry procedures are *self-synchronizing subprograms* that another task can invoke for task-to-task communication.

If task *t1* has an entry procedure *p*, then another task *t2* can execute:

t1.p(argument-list);

In order for *p* to execute, *t1* must execute:

accept p (parameter-list);

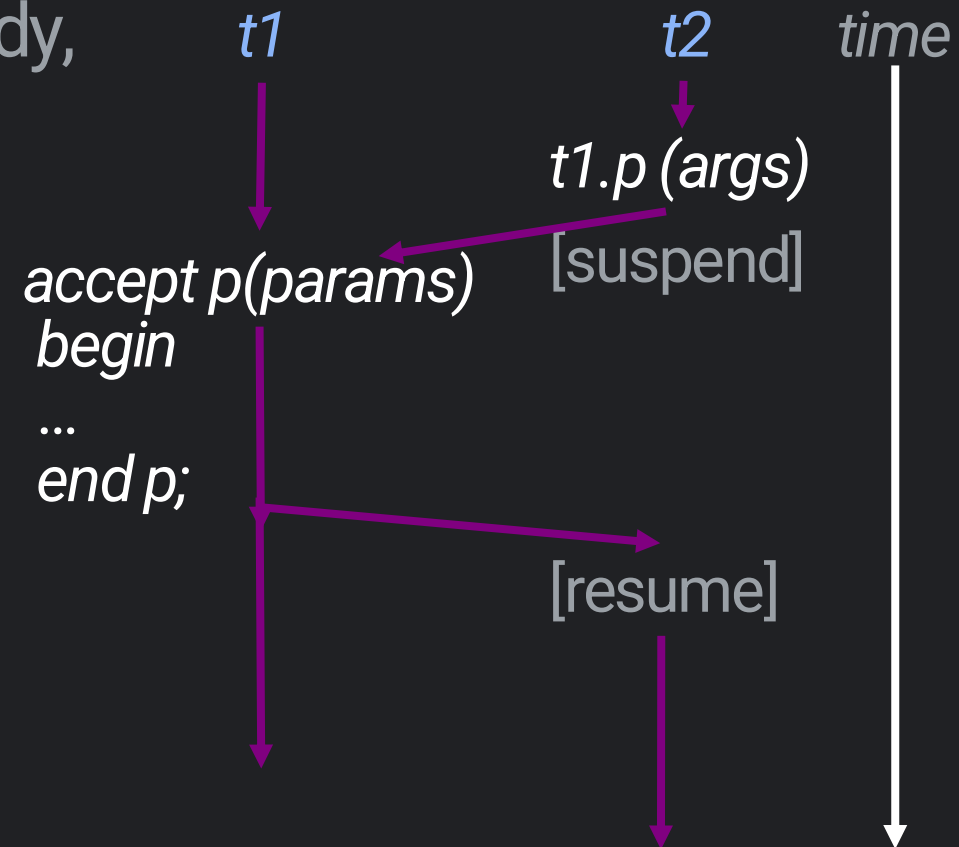
- If *t1* executes *accept p* and *t2* has not called *p*, *t1* must wait for *t2*;
- If *t2* calls *p* and *t1* has not done *accept p*, *t2* must wait for *t1*.



Rendezvous

When $t1$ and $t2$ are both ready,
 p executes:

- $t2$'s argument-list is evaluated and passed to $t1.p$'s parameters
- $t2$ suspends
- $t1$ executes the body of p , using its parameter values
- return-values (or *out* or *in out* parameters) are passed back to $t2$
- $t1$ continues execution;
 $t2$ resumes execution



This interaction is called a *rendezvous* between $t1$ and $t2$.

It does not depend on shared memory, so $t1$ and $t2$ can be on a uniprocessor, a tightly-coupled or a distributed multiprocessor.





Spring is just around the corner!

Ada Array Processing

How can we rewrite what's below to complete more quickly?

```
procedure sumArray is
  N: constant integer := 1000000;
  type RealArray is array(1..N) of float;
  anArray: RealArray;

  function sum(a: RealArray; first, last: integer)
    return float is
    result: float := 0.0;
  begin
    for i in first..last loop
      result := result + a(i);
    end loop;
    return result;
  end sum;

begin
  -- code to fill anArray with values omitted
  put( sum(anArray, 1, N) );
end sumArray;
```



Divide-And-Conquer via Tasks

```
procedure parallelSumArray is
  -- declarations of N, RealArray, anArray, Sum() as before ...
```

```
  task type ArraySliceAdder
    entry SumSlice(Start: in Integer; Stop: in Integer);
    entry GetSum(Result: out float);
  end ArraySliceAdder;

  task body ArraySliceAdder is
    i, j: Integer; Answer: Float;
  begin
    accept SumSlice(Start: in Integer; Stop: in Integer) do
      i:= Start; j:= Stop;                -- get inputs
    end SumSlice;

    Answer := Sum(anArray, i, j);         -- do the work

    accept GetSum(Result: out float) do
      Result := Answer;                   -- report outcome
    end GetSum;
  end ArraySliceAdder;
```

```
-- continued on next slide..
```



Divide-And-Conquer via Tasks (ii)

```
-- continued from previous slide ...

firstHalfSum, secondHalfSum: Integer;
T1, T2 : ArraySliceAdder;    -- T1, T2 start & wait on accept
begin
  -- code to fill anArray with values omitted

  T1.SumSlice(1, N/2);       -- start T1 on 1st half
  T2.SumSlice(N/2 + 1, N);   -- start T2 on 2nd half

  T1.GetSum( firstHalfSum ); -- get 1st half sum from T1
  T2.GetSum( secondHalfSum ); -- get 2nd half sum from T2

  put( firstHalfSum + secondHalfSum ); -- we're done!
end parallelSumArray;
```

Using two tasks T1 and T2, this *parallelSumArray* version requires roughly 1/2 the time required by *sumArray* (on a multiprocessor).

Using three tasks, the time will be roughly 1/3 the time of *sumArray*.

...



Producer-Consumer in Ada

To give the producer and consumer separate threads, we can define the behavior of one in the 'main' procedure:

and the behavior of the other in a separate task:

We can then build a Monitor-style *Buffer task* with *put()* and *get()* as (auto-synchronizing) entry procedures...

```
procedure ProducerConsumer is
  buf: Buffer;
  it: Item;

  task consumer;
  task body consumer is
    it: Item;
  begin
    loop
      buf.get(it);
      -- consume Item it
    end loop;
  end consumer;

begin -- producer task
  loop
    -- produce an Item in it
    buf.put(it);
  end loop;
end ProducerConsumer;
```



Capacity-1 Buffer

A single-value buffer is easy to build using an Ada *task-type*:

As a *task-type*, variables of this type (e.g., *buf*) will automatically have their own thread of execution.

The body of the task is a loop that accepts calls to *put()* and *get()* in strict alternation.

```
task type BoundedBuffer1 is
  entry get(it: out Item);
  entry put(it: in Item);
end BoundedBuffer1;
```

```
task body BoundedBuffer1 is
  myBuffer: Item;
begin
  loop
    accept put(it: in Item) do
      myBuffer := it;
    end put;

    accept get(it: out Item) do
      it := myBuffer;
    end get;
  end loop;
end BoundedBuffer1;
```

This causes *buf* to alternate between being empty and nonempty.



Capacity-N Buffer

An N-value buffer is a bit more work:

We can accept any call to *get()* so long as we are not empty, and any call to *put()* so long as we are not full.

Ada provides the *select-when* statement to guard an *accept*, and perform it if and only if a given condition is *true*

```
-- task declaration is as before ...  
task body BoundedBuffer is  
  N: constant integer := 1024;  
  package Buf is new Queue(N, Items);  
begin  
  loop  
    select  
      when not Buf.isFull =>  
        accept put(it: in Item) do  
          Buf.append(it);  
        end put;  
      or when not Buf.isEmpty =>  
        accept get(it: out Item) do  
          it := Buf.first;  
          Buf.delete;  
        end get;  
      end select;  
    end loop;  
end BoundedBuffer;
```



I was getting dressed this morning, and when I opened my wardrobe, there was a lion inside. I asked him, "What are you doing in there?" He answered, ...

... Narnia business.

MPI ...

... is the *Message Passing Interface*

... is an industry-standard library for distributed-memory parallel computing in C, C++, Fortran, with 3rd party bindings for Java, Python, R, ...

... was designed by a large consortium in 1994:

- 12 companies: *Cray, IBM, Intel, ...*
- 11 national labs: *ANL, LANL, LLNL, ORNL, Sandia, ...*
- representatives from 16 universities

... has “built in” support for many parallel design patterns

... continues to evolve (MPI 2.0 in 1997; 3.0 in 2012; ...)



Did you hear about that new band named *1023MB*?

They haven't had any gigs yet.

Summary

- *Concurrent computations* consist of multiple entities:
 - *Processes* in Smalltalk, MPI
 - *Tasks* in Ada
 - *Threads* in C/C++, C#, Java, Go, Python, Ruby, Scala, ...
- On a shared-memory multiprocessor:
 - The *Semaphore* was the first synchronization primitive
 - *Java* provides a *Semaphore* class for synchronizing threads
 - *Locks* and *condition variables* separate a semaphore's mutual-exclusion usage (locks) from its synchronization usage (c.v.s)
 - *Monitors* are higher-level, self-synchronizing objects
 - *Java* classes have an associated (simplified) monitor
- On a distributed system:
 - *Ada* tasks provide self-synchronizing *entry procedures*
 - *Erlang, Scala, MPI* support message-passing between processes



Summary (ii)

Comparing Monitors and Tasks/Threads (and Coroutines):

	Has Its Own Thread	Has Its Own Execution State
Monitor	No	No
Task/Thread	Yes	Yes
Coroutine	No	Yes

A *coroutine* (Simula, Lua) is two or more procedures that *share a single thread*, each exercising mutual control over the other:

```
procedure A;  
begin  
  -- do something  
  resume B;  
  -- do something  
  resume B;  
  -- do something  
  -- ...  
end A;
```

```
procedure B;  
begin  
  -- do something  
  resume A;  
  -- do something  
  resume A;  
  -- ...  
end B;
```

