Concurrent and Parallel Programming, Part II

Programming Languages

CS 214

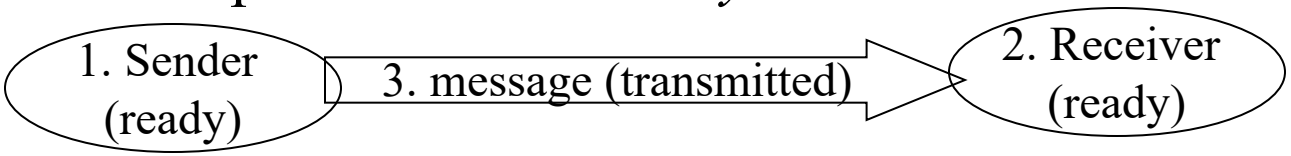
Distributed Synchronization

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

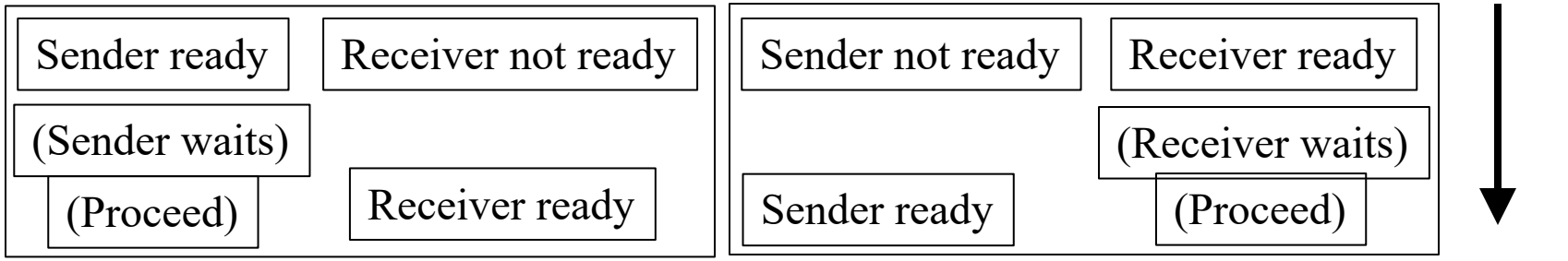
- They are of *no use* on a *distributed multiprocessor*

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*:

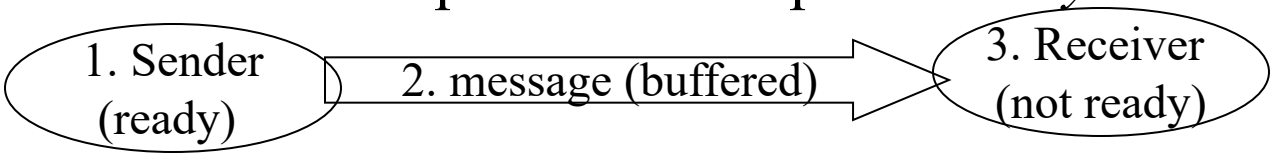


Two scenarios...



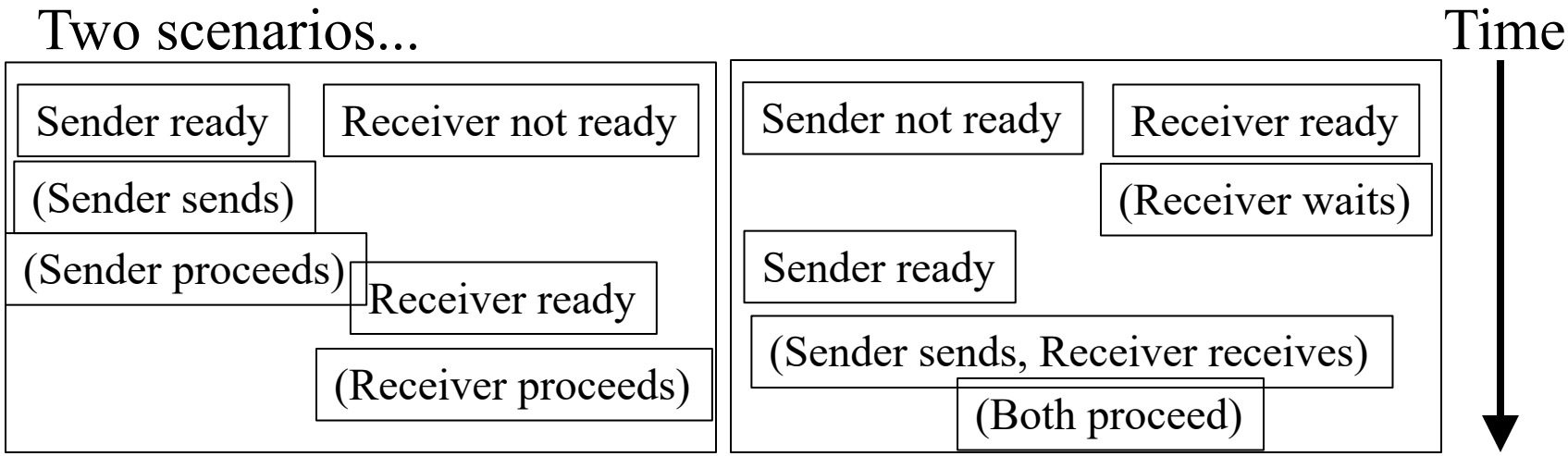
Asynchronous Communication

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



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

Two scenarios...

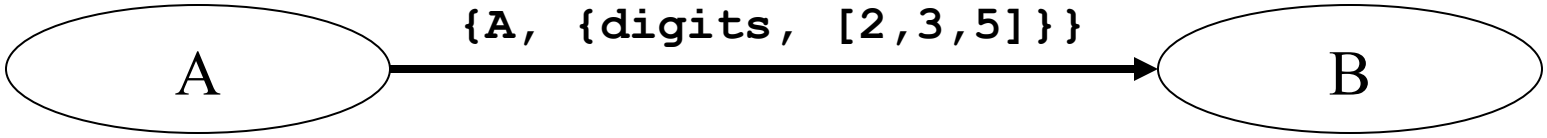


Message-buffering eliminates some (but not all) of the waiting.

Message-Passing Languages

Some languages support message-passing between _____:

- _____ is a functional language developed at Ericcson 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
```

- _____ 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 are _____ that another task can invoke for task-to-task communication.

If task __ has an entry procedure __, then another task __ can execute:

_____*(argument-list);*

In order for __ to execute, __ must execute:

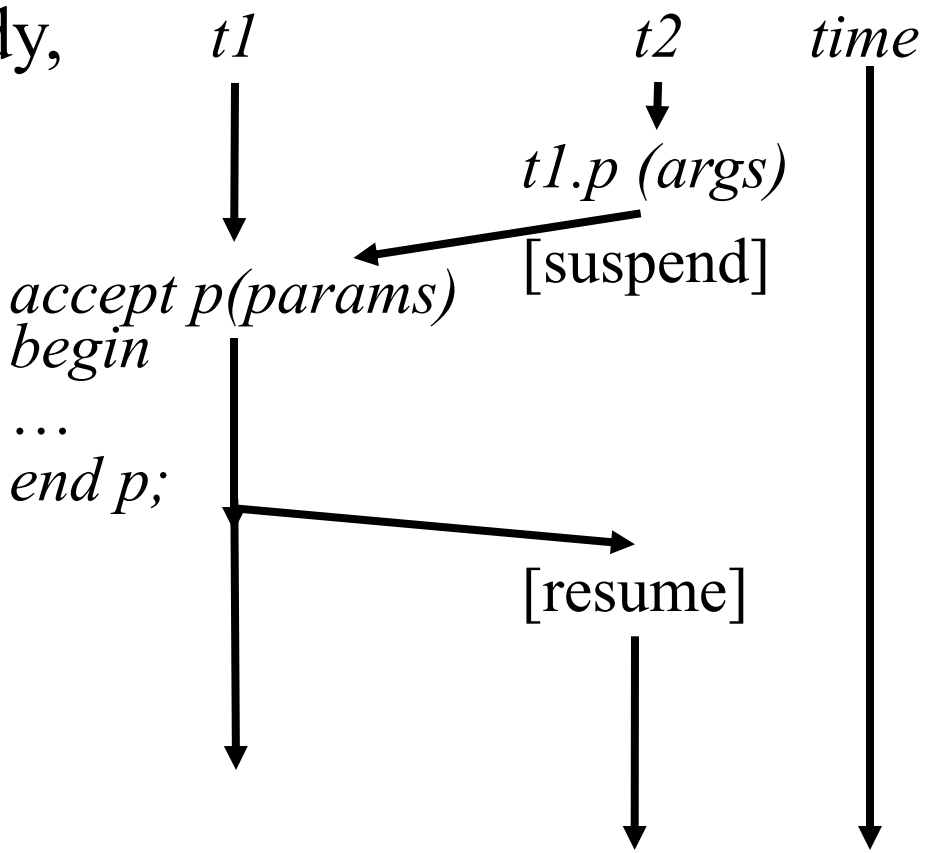
_____ *(parameter-list);*

- If __ executes _____ and __ has not called __, __ must wait for __;
- If __ calls __ and __ has not done _____, __ must wait for __.

Rendezvous

When __ and __ are both ready, __ executes:

- __*'s argument-list* is evaluated and passed to ____'s parameters
- __ suspends
- __ executes the body of __, using its parameter values
- return-values (or ___ or ____ parameters) are passed back to __
- __ continues execution; __ resumes execution



This interaction is called a _____ between __ and __.
It does not depend on shared memory, so __ and __ can be on a uniprocessor, a tightly-coupled or a distributed multiprocessor.

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 __ and __, this _____ version requires roughly 1/2 the time required by _____ (on a multiprocessor).

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

…

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 _____ with _____ and _____ 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 _____:

As a _____, variables of this type (e.g., ___) will automatically have their own thread of execution.

The body of the task is a loop that accepts calls to ____ and ____ 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 ___ 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 _____ so long as we are not empty, and any call to _____ so long as we are not full.

Ada provides the_____ statement to *guard* an _____, 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;
```

MPI …

… is the _____

… is an industry-standard library for distributed-memory parallel computing in _____, 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; …)

Summary

- _____ consist of multiple entities:
  - _____ in Smalltalk, MPI
  - _____ in Ada
  - _____ in C/C++, C#, Java, Go, Python, Ruby, Scala, …
- On a shared-memory multiprocessor:
  - The _____ was the first synchronization primitive
    - _____ provides a _____ class for synchronizing threads
  - _____ and _____ separate a semaphore's mutual-exclusion usage (locks) from its synchronization usage (c.v.s)
  - _____ are higher-level, self-synchronizing objects
    - _____ classes have an associated (simplified) monitor
- On a distributed system:
  - _____ tasks provide self-synchronizing _____
  - _____ 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 _____ (Simula, Lua) is two or more procedures that *share a single thread*, each exercising mutual control over the other: