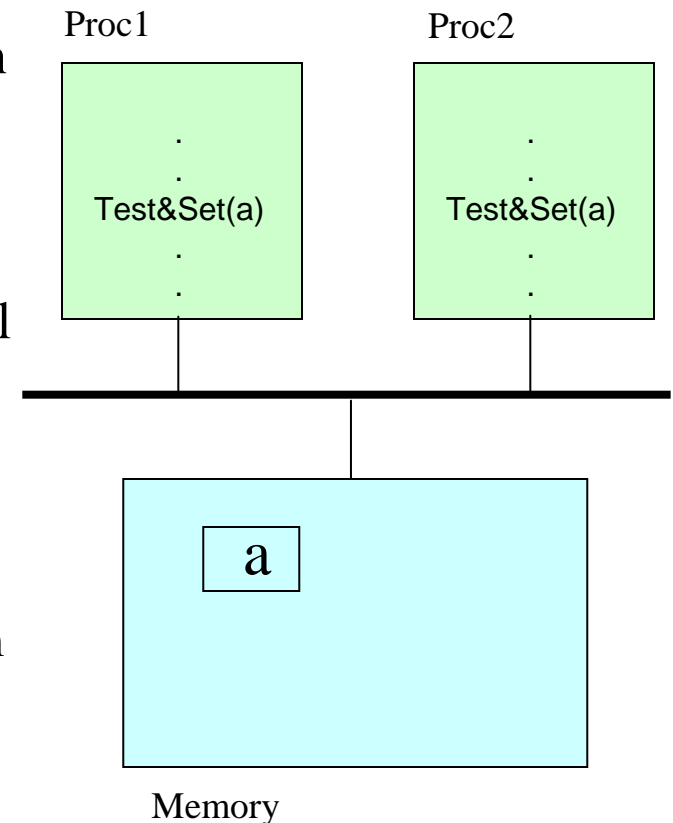


Process synchronization on distributed architectures

- We have so far studied process synchronization mechanisms for shared memory machines
 - implementing locks, semaphores, monitors requires that all processors have access to some shared variables
- A distributed system is a system built of several independent computers that:
 - have their own memory and run their own operating system
 - communicate through messages over a communication network
 - they share neither an address space nor a system bus nor a clock
- How is process synchronization dealt with on distributed systems?



Interprocess communication

- Process-to-process communication plays a fundamental role in distributed systems
- Data transport is most obvious aspect, but synchronization side effects are equally important
- Process communication takes different forms
 - Most relevant for our purposes: Message Passing, Remote Procedure Calls, Remote Method invocations
- We will start with message passing
 - historically: CSP, then ADA, then MPI

Communicating Sequential Processes (CSP)

- Key idea: input and output commands (a.k.a. send and receive) as synchronization primitives
- Processes communicate by *corresponding*, i.e.:
 - proc *a* executes an input command specifying proc *b* as source
 - proc *b* executes an output command specifying *a* as destination
 - *a*'s input variables matches the type of *b*'s output expression
- Input and output operations are synchronous
 - i.e. the command that happens to be executed first does not completes until the corresponding command is also executed

CSP notation

- I/O:
 - input: $\langle \text{source process id} \rangle ? \langle \text{target variable} \rangle$
 - output: $\langle \text{destination process id} \rangle ! \langle \text{expression} \rangle$
- Concurrency among processes:
 - $[\text{proc}_1 \parallel \text{proc}_2 \parallel \dots \parallel \text{proc}_n]$
- Guarded commands:
 - $G \rightarrow CL$ (where G is a *guard*, CL is a list of commands)
- Alternative command:
 - $G_1 \rightarrow CL_1 \square G_2 \rightarrow CL_2 \square \dots \square G_n \rightarrow CL_n$
- Repetitive command:
 - $*[G_1 \rightarrow CL_1 \square G_2 \rightarrow CL_2 \square \dots \square G_n \rightarrow CL_n]$

CSP examples

- Moving one character at a time from west to east
 - $* [c: \text{character}; \text{west} ? c \rightarrow \text{east} ! c]$
- Scanning an array to look for an element with value n :
 - $i := 0; * [i < \text{size}; \text{content}(i) \neq n \rightarrow i := i + 1]$

CSP: Producer-Consumer problem

Process bounded-buffer

pool: 0..9 of buffer;

in, out: integer

* [in < out + 10; producer?Pool(in **mod** 10) → in := in + 1; □
out < in; consumer?more() → consumer!Pool(out **mod** 10); out := out + 1]

Process Consumer

bounded-buffer!more(); bounded-buffer?*q*

Process Producer

bounded-buffer!*p*

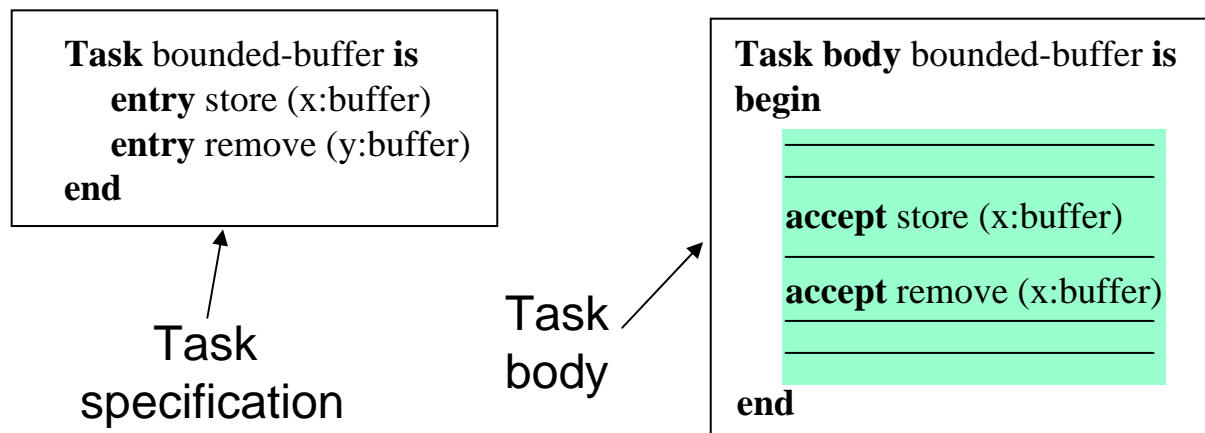
- Dummy procedure *more()* needed for proper synchronization
 - output commands are not allowed within guards

Notes on the CSP model

- Pros:
 - simple and flexible means of specifying concurrency and synchronization
 - it is easy to think in terms of sequential processes interacting only through explicit message passing
- Cons:
 - requires explicit naming of processes in I/O commands
 - synchronous I/O only (i.e. blocking, unbuffered operations) can be inefficient

Ada tasks

- Ada is a programming language that supports concurrent processes within a program
- Processes are called **tasks**
 - tasks provide services called **entries**, defined using an **accept** statement
 - tasks consist of a *Task Specification* and of a *Task Body*:



Trivial Producer-Consumer solution

```
task single-buffer is  
  entry store (x:buffer)  
  entry remove (y:buffer)  
end
```

```
task body single-buffer is  
temp: buffer;  
begin  
loop
```

```
  accept store (x:buffer)  
  temp := x;  
  end store;
```

```
  accept remove (y:buffer)  
  y := temp;  
  end remove;
```

```
end loop  
end single-buffer
```

Producer process:

```
...  
single-buffer.store(a);  
...
```

Consumer process:

```
...  
single-buffer.remove(b);  
...
```

Select statement

- The **select** statement allows Ada to deal with nondeterministic events
 - conceptually similar to the CSP *alternative* command
 - allows **accept**'s to be executed in arbitrary order
- Syntax:

```
select
  when <guard > => < accept statement >
or
  ...
else <statement>
end select
```

Complete Producer-Consumer solution

```
task bounded-buffer is  
  entry store (x:buffer)  
  entry remove (y:buffer)  
end
```

```
task body bounded-buffer is  
  ring: [0..9] of buffer;  
  head, tail: integer;  
begin  
loop  
  select
```

```
    when head < tail+10 =>
```

```
      accept store (x:buffer)
```

```
        ring[head] := x;
```

```
        head := head + 1 mod 10;
```

```
      end store;
```

```
    when tail < head =>
```

```
      accept remove (y:buffer)
```

```
        y := ring[tail];
```

```
        tail := tail + 1 mod 10;
```

```
      end remove;
```

```
    end select
```

```
  end loop
```

```
end bounded-buffer
```