PURDUE UNIVERSITY

Principles of Concurrency

Lecture 15
Message Passing and Communicating Sequential Processes

Adapted from:

- Communicating Sequential Processes, Hoare (1978)

Issues

Given a collection of threads, how should they communicate information among one another?

Using message-passing, they communicate through *messages*, information that is directed from one thread to another.

- Sometimes the recipient may be anonymous
  channel-based communication
- Typically, the recipient is known

What should the sender do after the message is sent?

- wait until the recipient acknowledges receipt (synchronous)
- proceed regardless (asynchronous)

Communicating Sequential Processes

Landmark proposal by Hoare in 1978

Key components

- guarded commands
- dynamic thread creation
- synchronous message passing

  an input action in a guarded command causes the actions in the guard to block until the input action can be satisfied

  - there is a matching output action
- *No communication through global (shared) variables*

Parallel Commands

Defines eleven processes

Behavior of processes fork(0), ..., fork(4) specified by command FORK

- Bound variable i indicates identity of a particular fork

Similar structure for phil

```
room :: ROOM ||
fork(i:0..4) :: FORK ||
phil(i:0..4) :: PHIL
```

Input and Output Commands

`X ? (a,b)`

- inputs from process X a pair, binding the first element to a and the second to b

`Y ! (3 * a, b + 13)`

- outputs to process Y a pair, consisting of the values computed by the corresponding expressions within the environment in which the command takes place

`X(i)? V()`

- From the ith array of processes X, input a signal V()

`display(i-2) ! "A"`

- send to the i-2nd display the character "A"

Guarded Commands


```
x >= y --> m := x  []  y >=x m := y
```

Assign m to x if x >= y; assign m to y if y >= x. Do one or the other if x = y.

```
i := 0;
* [ i < size; content(i) <> n -> i := i + 1 ]
```

Scan the elements of the array contents incrementing counter i as long as n is not encountered and the end of the array is not reached.

Guarded Commands


```
X:: *[c:char, A?c ->
        [ c <> "*" --> B!c
        []
          c = "*" --> A?c;
                          [ c <> "*" --> B!"*"; B!c
                          []
                            c = "*" --> B!"#" ]
        ]  ]
```

What does this program do?
What assumptions does it make?

Bounded Buffer


```
X::
  buffer:(0..9) portion;
  in,out:integer, in:= 0; out := 0;
  *[in < out + 10; producer?buffer(in mod 10) --> in := in + 1
   []
    out < in; consumer?more() --> consumer ! buffer(out mod 10);
                                   out := out + 1
  ]
```

Consumer contains pairs of commands `X!more()` and `X?p`

Producer contains commands of the form `X!p`

Small Set of Integers


```
S::
  content:(0..99)integer, size:integer,size := 0;
*[  n:integer,X?has(n) --> SEARCH;X!(i<size)
    []
    n:integer; X?insert(n) --> SEARCH;
       [ i<size --> skip
         []
         i = size; size < 100 -->  content(size) := n; size := size + 1
]   ]
```

where SEARCH is:

```
  i:integer; i := 0;
   *[i < size; content(i) <> n --> i := i + 1 ]
```

Dining Philosophers

Five philosophers:

- Only eat and think
- Share a common dining room.
  - Shared bowl of spaghetti
  - Five forks
- Need two forks to eat (both right and left)
- After finishing eating, puts both forks down

Dining Philosophers


```
PHIL = *[ ... for ith philosopher ....
               THINK;
                room!enter( );
                fork(i)!pickup(); fork((i+1) mod 5)!pickup();
               EAT;
                 fork(i)!putdown(); fork((i+1) mod 5)!putdown();
                 room!exit()
              ]

FORK = *[phil(i)?pickup() --> phil(i)?putdown()
               | (phil(i - 1) mod 5)?pickup() --> phil((i-1) mod 5)?putdown()
              ]

ROOM = occupancy:integer; occupancy := 0;
               *[(i:0..4)phil(i)?enter() --> occupancy := occupancy + 1
               | (i:0..4)phil(i)?exit() --> occupancy := occupancy - 1
              ]

[room::ROOM || fork(i:0..4)::FORK || phil(i:0..4)::PHIL]
```

What happens if all five philosophers enter the room, and each picks up the left fork?
How would you adapt the algorithm to prevent this scenario?

Issues

Explicit naming of source and destination

- No first-class channels or ports

Fully synchronous

- How would you model asynchronous communication?

No unbounded number of processes

Fairness

```
[X::Y!stop() || Y::continue:boolean; continue := true;
                *[ continue; X?stop( ) --> continue := false
                 |  continue --> n := n + 1
               ]
]
```

Output guards

`Z:: [X!2 || Y!3]` could be expressed as: `Z::[X!2 --> Y!3 [] Y!3 --> X!2]`

Why does the following not work?

```
Z::[true --> X!2; Y!3 [] true --> Y!3; X!2]
```

Consider: `Y :: Z?y; X!go() || X:: Y?go(); Z?x`