Washington State University
Vancouver

CS 563

Concurrent Programming

Lecture 6: Communicating Sequential Processes (CSP)

Communication Sequential Processes

- History
  - CSP is a formal language for describing concurrent systems
  - It was introduced by C.A.R. Hoare in 1978
  - An implementation of CSP (OCCAM) was used in the T9000 Transputer

Syntax

- Destination!port(e1, ... , en)

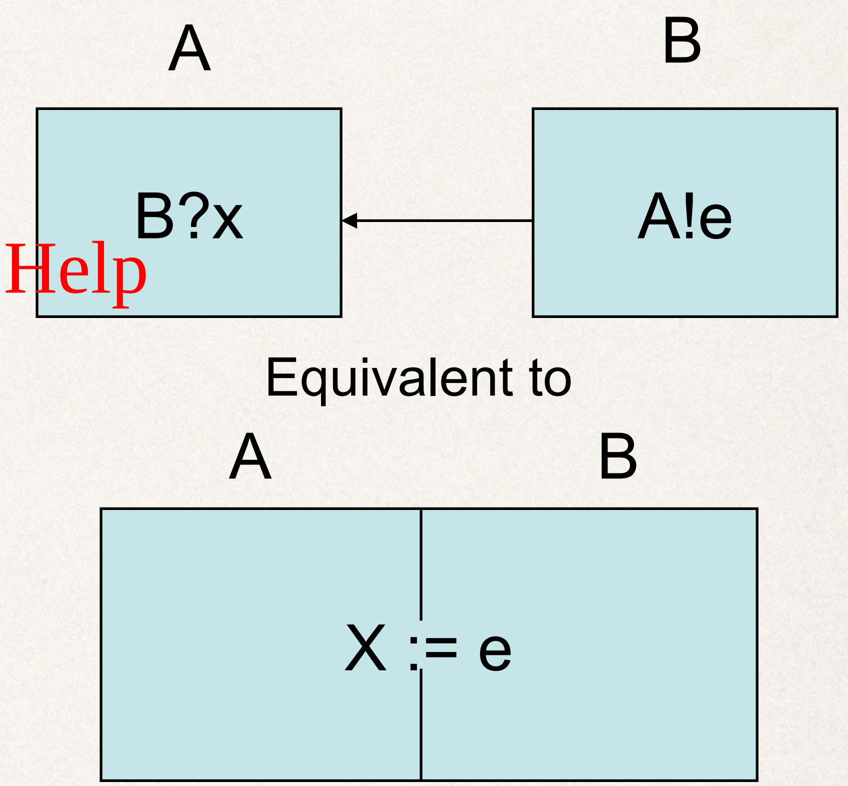
- Source?port(x1, ..., xn)

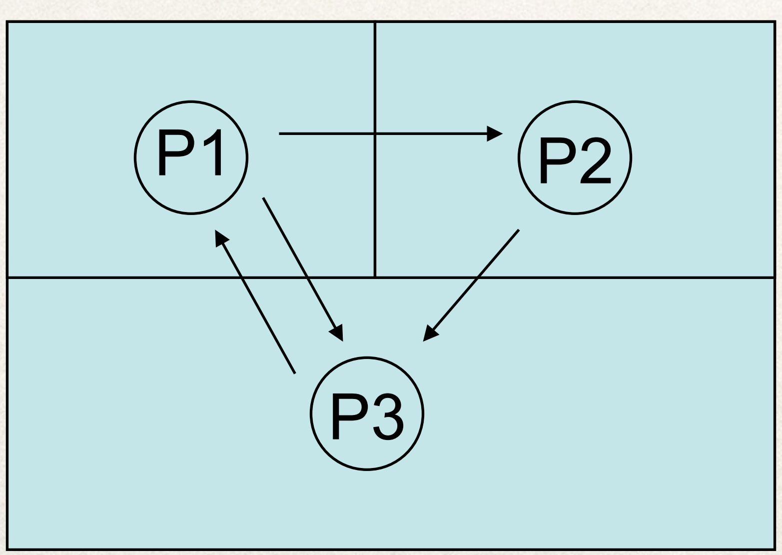
Example: Copy


```
process Copy {
   char c;
   do true ->
      West?c; # input char from West
      East!c;   # output char to East
   od
}
```

CSP: The Main Idea

- Something similar to input and output can be used to allow processes to communicate
- Multiple communicating processes can be present in both a single machine and across multiple machines





CSP Process Interaction

- Processes interact via synchronous message passing

- When a process gets to a send, it has to wait until the receiving process is ready to receive

- When a process gets to a receive, it has to wait until the sending process sends
- Processes have to rendezvous at a point, or else process is blocked
- Processes have to be named explicitly

Example: GCD

Any Source

Nondeterministic Choice

Destination

```
process GCD {
    int id, x, y;
    do true ->
        Client[*] ? args(id, x, y);    # input a "call"
        # repeat the following until x == y
        do x > y -> x = x - y;
        [ ] x < y -> y = y - x;
        od
        Client[id] ! result(x); # return the result
    od
}
```

Guarded Communication

- CSP is partially based on a programming construct proposed by Dijkstra to indicate the concurrent execution of processes and non-determinism

- Syntax: B; C->S;

- Example:

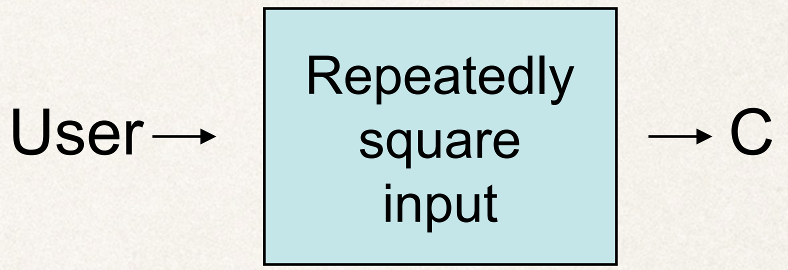
```
process Copy {
   char c;
   do West?c -> East!c; od
}
```

Guarded Communication Example

- Define a process that repeatedly accepts input from the user, squares it and sends it to process C

- Without guards:
  *[x:integer; user?x; C!x*x]

- With guards:
  *[x:integer; user?x ->C ! x*x]



User ? X is the guard
C ! x*x is the guarded command

Guarded Outcomes

- The guard succeeds when the Boolean expression is true, and (if the guard includes I/O), the I/O does not block
- The guard fails if the Boolean is false
- The guard is neither true or false if the Boolean is true and the I/O of the guard does block

Specifying Alternative Commands

*[<guard$_1$> -> <guarded commands$_1$>
[] <guard$_2$> -> <guarded commands$_2$>
[] <guard$_3$> -> <guarded commands$_3$>
.
.
.
[] <guard$_n$> -> <guarded commands$_n$>
]

Cases:

1. If all guards fail, the result is an error.
2. If one guard succeeds, it executes its command (or command list).
3. If more than 1 guard succeeds, one of the commands (whose guard was true) is non-deterministically chosen and executed
4. If none succeed, but not all fail, wait.

Buffering I

```
process Copy {
    char c1, c2;
    West ? c1;
    do West ? c2 -> East ! c1; c1 = c2;
    [ ] East  !  c1 -> West ? c1;
    od
}
```

Buffering II

```
process Copy {
    char buffer[10];
    int front = 0, rear = 0, count = 0;
    do count < 10; West ? buffer[rear] ->
            count = count+1; rear = (rear + 1) mod 10;
    [ ] count > 0; East ! buffer[front] ->
            count = count - 1; front = (front + 1) mod 10;
    od
}
```

Resource Allocation

```
process Allocator {
    int avail = MAXUNITS;
    set units = initial values;
    int index, unitid;
    do avail > 0; Client[*] ? acquire (index) ->
        avail--; remove (units, unitid);
        Client [index] ! reply (unitid);
    [ ] Client[*] ? release (index, unitid) ->
        avail++; insert (units, unitid);
    od
}
```

Sieve of Erastosthenes

```
process Sieve[1] {
  int p = 2;
  for [i = 3 to n by 2]
    Sieve[2]!i;   # pass odd numbers to Sieve[2]
}
process Sieve[i = 2 to L] {
  int p, next;
  Sieve[i-1]?p;              # p is a prime
  do Sieve[i-1]?next ->      # receive next candidate
    if (next mod p) != 0 ->  # if it might be prime,
      Sieve[i+1]!next;       #   pass it on
    fi
  od
}
```