WASHINGTON STATE UNIVERSITY
VANCOUVER

CS 563

Concurrent Programming

Lecture 5: Message Passing (2)

Client-Server

- Two-way interaction pattern

Client-Server

(a) with procedures

- client does:


```
call(args)
```


- server is:

```
procedure(formals)
  body
end
```

Client-Server

(b) with message passing


```
chan request(...), reply(...)
```



```
"caller"
 send request(args)
          ...
 receive reply(vars)
```

```
"server"
 while(true) {# std server loop
     receive request(vars)
     body
     send reply(results)
 }
```

Client-Server

```
"caller"
 send request(args)
          ...
 receive reply(vars)
```

```
"server"
 while(true) {# std server loop
      receive request(vars)
      body
      send reply(results)
 }
```


- (c) with message passing and multiple clients
  - Suppose there are multiple clients (and one server). What has to change in (b)?

Client-Server

```
chan request(int clientID, types of input values);
chan reply[n](types of results);
process Server {
  int clientID;
  declarations of other permanent variables;
  initialization code;
  while (true) {    ## loop invariant W
    receive request(clientID, input values);
    code from body of operation op;
    send reply[clientID](results);
  }
}
process Client[i = 0 to n-1] {
  send request(i, value arguments);      # "call" op
  receive reply[i](result arguments);    # wait for reply
}
```

syntax for creating multiple processes

how processes know who they are

Client-Server (Multiple Operations)

```
type op_kind = enum(op1, ..., opn);
type arg_type = union(arg1, ..., argn);
type result_type = union(res1, ..., resn);

chan request(int clientID, op_kind, arg_type);
chan reply[n](res_type);

process Server {
  int clientID; op_kind kind; arg_type args;
  res_type results; declarations of other variables;
  initialization code;
  while (true) {     ## loop invariant MI
    receive request(clientID, kind, args);
    if (kind == op1)
      { body of op1; }
    ...
    else if (kind == opn)
      { body of opn; }
    send reply[clientID](results);
  }
}

process Client[i = 0 to n-1] {
  arg_type myargs; result_type myresults;
  place value arguments in myargs;
  send request(i, opj, myargs);     # "call" opj
  receive reply[i](myresults);      # wait for reply
}
```

Resource Allocation Using Message Passing


```
process Client[i = 0 to n-1] {
  int unitID;
  send request(i, ACQUIRE, 0)     # "call" request
  receive reply[i](unitID);
  # use resource unitID, then release it
  send request(i, RELEASE, unitID);
    ...
}
```

Resource Allocation Using Message Passing

```
type op_kind = enum(ACQUIRE, RELEASE);
chan request(int clientID, op_kind kind, int unitid);
chan reply[n](int unitID);

process Allocator {
  int avail = MAXUNITS; set units = initial values;
  queue pending;  # initially empty
  int clientID, unitID; op_kind kind;
  declarations of other local variables;
  while (true) {
    receive request(clientID, kind, unitID);
    if (kind == ACQUIRE) {
       if (avail > 0) {  # honor request now
          avail--; remove(units, unitID);
          send reply[clientID](unitID);
       } else   # remember request
          insert(pending, clientID);
    } else {   # kind == RELEASE
       if empty(pending) {  # return unitID to units
          avail++; insert(units, unitid);
       } else {  # allocate unitID to a waiting client
          remove(pending, clientID);
          send reply[clientID](unitID);
       }
    }
  }
}
```

Interacting Peers

- Used to share data, combine data, make decisions

- Exchanging values problem:
  - There are n processes; each has a value; want every process to learn every value
  - This type of exchange occurs in many places: n-body simulation, etc.

Min-Max Problem

```
chan values(int), results[n](int smallest, int largest);
process P[0] {   # coordinator process
  int v;   # assume v has been initialized
  int new, smallest = v, largest = v;  # initial state
  # gather values and save the smallest and largest
  for [i = 1 to n-1] {
    receive values(new);
    if (new < smallest)
      smallest = new;
    if (new > largest)
      largest = new;
  }
  # send the results to the other processes
  for [i = 1 to n-1]
    send results[i](smallest, largest)
}
process P[i = 1 to n-1] {
  int v;   # assume v has been initialized
  int smallest, largest;
  send values(v);
  receive results[i](smallest, largest);
}
```


Centralized

Min-Max Problem

```
chan values[n](int);
process P[i = 0 to n-1] {
  int v;   # assume v has been initialized
  int new, smallest = v, largest = v;  # initial state
  # send my value to the other processes
  for [j = 0 to n-1 st j != i]
    send values[j](v);
  # gather values and save the smallest and largest
  for [j = 1 to n-1] {
    receive values[i](new);
    if (new < smallest)
      smallest = new;
    if (new > largest)
      largest = new;
  }
}
```

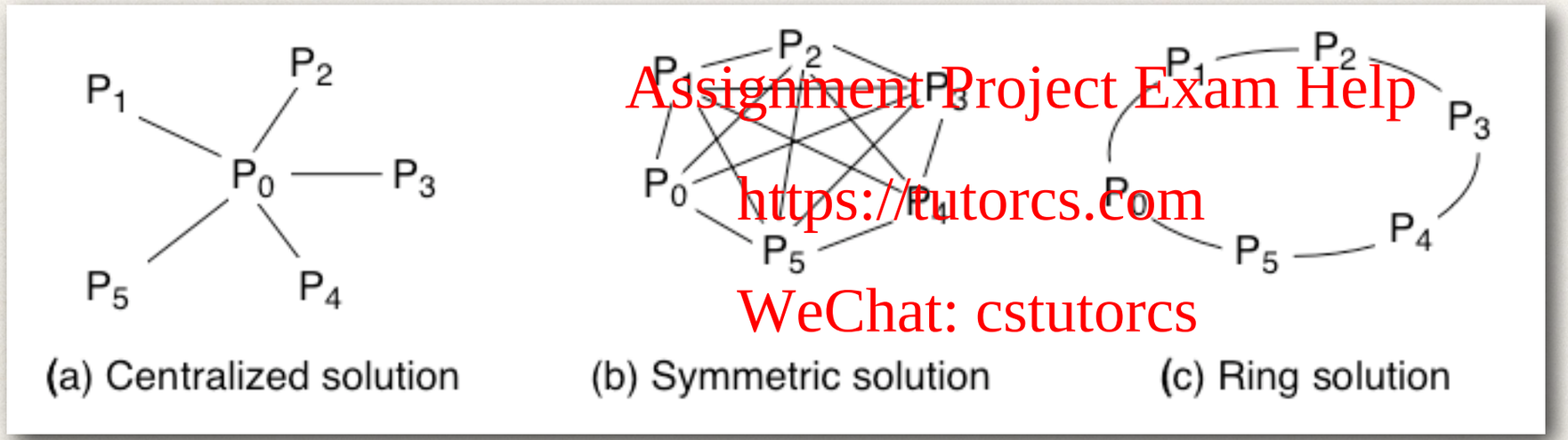

Symmetric

Min-Max Problem

```
chan values[n](int smallest, int largest);
process P[0] {  # initiates the exchanges
  int v;   # assume v has been initialized
  int smallest = v, largest = v;  # initial state
  # send v to next process, P[1]
  send values[1](smallest, largest);
  # get global smallest and largest from P[n-1] and
  #   pass them on to P[1]
  receive values[0](smallest, largest);
  send values[1](smallest, largest);
}
process P[i = 1 to n-1] {
  int v;   # assume v has been initialized
  int smallest, largest;
  # receive smallest and largest so far, then update
  #   them by comparing their values to v
  receive values[i](smallest, largest)
  if (v < smallest)
      smallest = v;
  if (v > largest)
      largest = v;
  # send the result to the next processes, then wait
  # to get the global result
  send values[(i+1) mod n](smallest, largest);
  receive values[i](smallest, largest);
}
```


Ring

Communication Patterns



(a) Centralized solution (b) Symmetric solution (c) Ring solution

Message counts:

a. 2(n-1)

b. n(n-1)

c. 2n

Synchronous Message Passing

- `synch_send name(expr1, ..., exprN)`
  - Types and number of fields must match
  - Effect:
    - Evaluate the expressions and produce a message M
    - Atomically append M to the end of the named channel
      - ➡ sender is blocked until the message is received (synchronous)

Producer Consumer

```
chan values(int);

process Producer{
        int data[n];
        for [i=0 to n-1]{
           do some computation;
           synch_send values(data[i]);
        }
}
```

```
process Consumer{
int results[n];
for [i=0 to n-1]{
   receive
   values(results[i]);
   do some computation;
}
}
```


Advantage: bound on the size of channels

Disadvantage: concurrency is reduced

anything else?

```
chan in1(int), in2(int);

process P1{
        int value1=1, value2;
        synch_send in2(value1);
        receive in1(value2);
}

process P2{
        int value1, value2=2;
        synch_send in1(value2);
        receive in2(value1);
}
```


Disadvantage: more prone to deadlock