Synchronization Methods in Message Passing Model

Message Passing Model

- Channel Specification
  - Specify how to communicate
- Synchronization Mechanisms
  - The synchronization mechanism between sender and receiver
  - Block/nonblock send, block/nonblock receive
  - Guarded communication

Channel Specification

- Direct naming
  - Simple but less powerful
  - Not suitable for specification of server-client systems
- Port naming
  - Server use a single port to receive client request
  - Simple, suitable for single server, multiple client systems
- Global naming
  - Suitable for multiple server, multiple client systems
  - Hard to implement for multiple processor systems

Direct naming: Channel dedicated to two specific nodes
Port naming: Port dedicated to one node, known to others (phone number)
Global naming: Channel can be used by any sender/receiver (bulletin board)
(In 1980s, VAX VMS offers this option)

Synchronization

- Blocked send, blocked receive (BSBR)
  - Easy to implement but less powerful
  - Example: Ada Rendezvous
- Nonblocked send, blocked receive (NSBR)
  - Most message passing systems use this model
  - need unbounded message queue to assure correctness
- Nonblocked send, nonblocked receive (NSNR)
  - More powerful than NSBR and BSBR
  - But difficult to reason the correctness of the system
    - The state of the system can be very different at the time the message is received

Producers-Consumer Problem

- Consider multiple producers and multiple consumers
  - Any producer's output can be processed by any consumer

Which naming scheme is most suitable for this problem?

- Use port naming
  - Each process has a port
  - Won't work since each producer needs to decide which consumer the produced item should be sent to $\Rightarrow$ Not desirable
- Use a buffer manager
  - Assume only port naming is available
  - The manager manages a shared buffer
  - It is like to let the user implement global naming

Producers-Consumer Problem

- Solution based on buffer manager and port naming
  - How to synchronize the producers and consumers with the bounded buffer
    - Buffer full $\Rightarrow$ Stop receiving messages from producers
    - Buffer empty $\Rightarrow$ Stop receiving messages from consumers
    - $\Rightarrow$ Need blocked send to achieve the control
  - $\Rightarrow$ Manager needs two ports, one for consumers, one for producers
    - Otherwise, cannot block only consumers or only producers
    - Still have problem: How to read from two ports
    - Solution 1: Use blocked receive and two threads
      - Thread creation is still an overhead
    - Solution 2: Use nonblocked receive
      - Poll two ports using BSNR
    - Any other possibility?

Producer-Consumer Problem

- Manager process ($P_{mamnager}$)
  - *reqport*: manager receives requests from consumers
  - *dataport*: manager receives data items from producers
  - BSNR
    - *Receive* returns immediately, it returns *true* if a message is ready in the port, otherwise, *false*
    - *Send* only returns after the message is received
  - *count*: # data items in $P_{mamnager}$'s buffer

BSBR Producer-Consumer Solution

Producer Process:

repeat
  produce *item*;
  send (*dataport*, *item*, *localport*);
  receive (*dataport*, *ack*);
until *false*;

Consumer Process:

repeat
  send (*reqport*, *localport*);
  receive (*localport*, *item*);
  consume *item*;
until *false*;

BSBR Producer-Consumer Solution

Manager Process:

```
repeat forever
  if count < N then   // can get item from producer
    { receive (dataport, item, port);
      put item in the queue;  count := count + 1;
      send (port, ack);
    }
  if count > 0 then   // can get request from consumer
    { receive (reqport, port);
      take item from queue;  count := count - 1;
      send (port, item);
    }
```

What problem will this cause?

NSNR Producer-Consumer Solution

Producer Process:

```
repeat
  produce item;
  send (dataport, item, localport);
  while not receive (dataport, ack) do nothing;
until false;
```

Consumer Process:

```
repeat
  send (reqport, localport);
  while not receive (localport, item) do nothing;
  consume item;
until false;
```

NSNR Producer-Consumer Solution

Manager Process:

repeat forever
  if *count* = 0 **then**   // have to get item from producer
    { **while not** *receive* (*dataport*, *item, port*) **do** nothing;
      put *item* in the queue; *count* := *count* + 1;
      send (*port*, *ack*);   }
  else if *count* = *N* **then**   // have to send item to consumer
    { **while not** *receive* (*reqport*, *port*) **do** nothing;
      take *item* from queue; *count* := *count* – 1;
      send (*port*, *item*);   }
  else   // can either produce or consume
    { **if** *receive* (*dataport*, *item, port*) **then**
        { put *item* in the queue; *count* := *count* + 1; *send* (*port, ack*); }
      if *receive* (*reqport*, *port*) **then**
        { take *item* from queue; *count* := *count* – 1; *send* (*port*, *item*); }
    }

Synchronous/Asynchronous IO

- IO can be synchronous or asynchronous
  - Asynchronous IO
    - More flexible, but more complex to program
  - Blocking IO (synchronous)
    - Won't work when we want to handle multiple IO ports
    - Create a new thread for each port ⇒ But thread has overhead
      - Especially if synchronization (lock, semaphore) is needed
    - Use thread pool, create N threads in advance, use the free threads
      - This is a commonly used method before the select function is offered in socket communication package
  - Consider IO multiplexing
    - Blocked IO, but allow waiting on multiple IO ports

The system gets interrupt from NIC and can tell which port it is for
⇒ This feature can be implemented readily
⇒ But the system needs to peek into the payload, not just the header

Guarded Command

- Guarded command
  - Guarded statement: $G \rightarrow$ S
    - When the guard G is true, execute statement S
  - Guarded command in CSP
    if $G_1 \rightarrow S_1 \parallel G_2 \rightarrow S_2 \parallel \ldots \parallel G_n \rightarrow S_n$ **fi**
    - When G_i is true, execute S_i
    - If more than one guard is true, then choose one of them arbitrarily and execute the corresponding statement
    - If none of the guards are true, then terminate the statement without doing anything

Guarded Communication

- Guarded communication:
  - Statements in a guarded command contain message passing operations
  - Message passing is based on BSBR
  - A guarded communication statement is ready when
    - The guard is true
    - For receive statement: The message to be received is ready
    - For send statement: The receiver of the message is ready
  - If none of the guards are true, then terminate the statement
  - If some guards are true and no message is ready then blocked

GC Producer-Consumer Solution

Manager Process:

```
repeat select
  when count < N   // can get item from producer
  { receive (dataport, item);
    put item in the queue;
    count := count + 1;
  }
  when count > 0   // can send item to consumer
  { receive (reqport, port);
    take item from queue;
    send (port, item);
    count := count – 1;
  }
until false;
```

Guarded Communication

- Now guarded communication is adopted
  - Offered in Ada

```
select
      when G1 => accept P1 do .... end;
      when G2 => accept P2 do .... end;
      …..
      else ....;
end select
```

  - A similar concept is implemented in socket
    - Select()
    - Not a full guarded communication, no guard
    - Better performance than thread based implementation
    - Implemented after socket package is released

Reading

- Chapter 5
  - Including synchronization in shared memory model and message passing model