INTER PROCESS AND THREAD COMMUNICATION: DESIGN SPACE

Instructor: Prasun Dewan (FB 150, dewan@unc.edu)
POINTS VS. DESIGN SPACE

Dimensions are issues
Value are approaches

Pipes
RMI
Serialized Streams
Xinu
NIO
Sync
Sockets
OS Example

Scheduling

Synchronization

Memory Management
Why IPC Design Space?

- Many Examples
  - Xinu
  - Socket Channels
  - Pipes
  - Sockets
  - Web Services
  - RMI

- Common mechanism and issues
- Abstract model to cover all of them
- Implementation of the model
Distributed program

→ Inter-process coupling

→ inter-process information communication

→ Communication among lightweight or heavyweight processes
UNIFYING BASIS?

Are there forms of IPC that are fundamentally different from these?
**Intra Process Shared Memory**

- **Thread** writes to a shared object.
- **Thread** reads from the shared object.

**Process**

- **Write**: Sends information, read receives it.
- **e.g.:** Shared object could be a result of matrix communication.
- **How does information consumer get notified about new information?**
  - Thread synchronization mechanisms (semaphores, conditions, ...)
  - Receiving thread must wait.
- **Threads in different processes?**
**Inter Process Shared Memory**

- **Thread**
  - Write

- **Shared Object**
  - Read

- **Shared file/database/memory**

- **If not on same computer, then need process around it**

- **Need some other mechanism to communicate with the process**
**SHARED MEMORY PROS AND CONS**

- **Familiar model**
- **Not sufficient when threads on different computers**
- **Global variables considered bad**
- **Threads other than the writer and reader can access them**
- **Receiving thread must wait for or poll for information (unless some notification mechanism added)**
- **Inter-process notification method?**
SOFTWARE INTERRUPTS (SIGAMS)

Information receiving process registers software interrupt number and handlers with OS
register (interrupt#, handler)

Information sender interrupts (signals) receiving process by naming process and handler#
interrupt (process id, interrupt#)

OS calls software interrupt handler in process, interrupting its current activity, just as hardware calls hardware-interrupt handler in CPU.

New thread created?
Uses stack of current thread
OS-PROCESS COMMUNICATION

Operating System

register (1, handler₁)

User wishes to kill process (CTRL-C)

An alarm set by process goes on

Some limit such as file size or virtual time expired

Process (P²)

handler₁()

handler₂()
SOFTWARE INTERRUPTS PRO AND CONS

React to OS Event

Familiar interrupt model for IPC

Communicates only event id, not parameters of the event

Assumes processes share an operating system thus not suitable when processes on different computers

Process ids (usually) make no sense on other computers

Receiver cannot delay processing information (Dual of shared memory problem)

No way of queuing signals
MESSAGE PASSING

Alternative to Shared Memory and Software Interrupts

Many Examples

- Sockets
- Socket Channels
- Pipes
- HTTP
- Web Services
- RMI

Traditionally: RPC not considered message passing

Common mechanism and issues

Abstract model to cover all of them

GIPC (Generalized IPC) based on abstract model
UNIFYING BASIS?

- Xinu
- Pipes
- NIO
- Sync
- RMI
- Serialized Streams
- Sockets
MESSAGE PASSING: COMMON BASIS

Process/Thread

Message Queue

Port (Mailbox)

Processes /threads send each other messages
Sender deposits message in port queue (of varying size)
Receiver gets message from queue
Issues?

message ³
message ²
message ¹

message ³
message ²
message ¹
DISTINGUISHING ISSUES?

- Xinu
- Pipes
- NIO
- RMI
- Sync
- Sockets
- Serialized Streams
ISSUES IN MESSAGE PASSING

Process/Thread

Message Queue

Port (Mailbox)

Process/Thread

Reliable message delivery?

In-order message delivery?

Port access (and message routing)?

Operations?

Synchronous vs asynchronous?

Blocking vs non blocking?

Buffering of messages at queue?

Location of communicating threads?
RELIABLE VS. UN-RELIABLE

- **Process/Thread**
- **Message Queue**
- **Port (Mailbox)**
- **Process/Thread**

**Reliable**: every message sent to a port is receivable
- Noise, disconnection can cause unreliability
- When reliability needed, programmer does not need to implement it
- Reliable not always needed
- Spam, load averages, telepointers
- Reliable is less space efficient (redundancy) and time efficient (reliability algorithm)
- Networking handles this issue

Message Queue:
- Message \(^1\)
- Message \(^2\)
- Message \(^3\)
**In-Order vs. Not In-Order Delivery**

- **Process/Thread**
- **Message Queue**
- **Port (Mailbox)**

Order: message received in order in which they are sent

Different routes can take different times

When order needed, programmer does not need to implement it

Order not always needed

Commuting operations

In-order is less space efficient (seq #) and time efficient (in-order algorithm)

Networking handles this issue, but relevant to distributed computing as we see later

- message<sup>2</sup>
- message<sup>3</sup>
- message<sup>1</sup>
ACCESS/ROUTING

Process/Thread

Port (Mailbox)

Process/Thread

Set of senders and receivers?
**Simplex Bound Port**

- **Sender**
- **Port (Mailbox)**
- **Receiver**

**Communication between a pair of processes/threads**

**One is the sender and another the receiver**

**Example?**

A port can represent an opened file in a file server - File server sends file bytes to the file client.
**Duplex Bound Port**

- **Sender/Receiver**
  - Communication between a pair of processes/threads
  - Either can send or receive
  - Example?
    - HTTP connection, two-player game
  - > 1 sender, receiver?
**Simplex Input Port**

A single receiver, called server

Arbitrary number of senders, called clients

Set of simplex ports?

Messages from all clients queued together (not a collection of simplex bound ports)

Example?

Print Server

Few examples where no information comes back from server
(Replying) Duplex Input Port

Like simplex equivalent except server can also send messages to clients, each with separate queue.

In replying duplex input port, server can only reply back to messages, cannot initiate sends.

- Security, simpler API
- Example of replying?
  - Http Server, File Server (open, close)
- Example of general?
  - Session manager, relayer
- Dual of input port?
MESSAGE PASSING: COMMON BASIS (REVIEW)

Processes /threads send each other messages
Sender deposits message in port queue (of varying size)
Receiver gets message from queue
Issues?

Message Queue

message
message
message

Port (Mailbox)

Process/Thread

message
message
message
DUPLEX BOUND PORT (REVIEW)

Communication between a pair of processes/threads

Either can send or receive

Example?

HTTP connection, two–player game

> 1 sender, receiver?
(Replying) Duplex Input Port (Review)

- Like simplex equivalent except server can also send messages to clients, each with separate queue
- In replying duplex input port, server can only reply back to messages, cannot initiate sends
  - Security, simpler API

Example of replying?
- Http Server, File Server (open, close)

Example of general?
- Session manager, relayer

Dual of input port?
**Simplex Output Port**

- **Client**
- **Port (Mailbox)**
  - message³
  - message¹
- **Server**
- **Server**

**A single sender, called client**

**Arbitrary number of receivers, called servers, sharing a single queue of received messages**

**Broadcast?**

**A message goes to a single server – it is not broadcast**

**Example?**

**Coupon distribution**
**Duplex Output Port**

Like simplex output port, except servers can send messages to client.

- Example?
- SETI computation

Few examples of one client with many servers.
SIMPLEX FREE PORT

Arbitrary number of senders, called clients, to a single message queue

Arbitrary number of receivers, called servers, sharing a single queue of received messages

Examples?

Load distribution among servers

Shared pool of print servers
**Duplex Free Port**

- **Client**
- **Client**
- **Port (Mailbox)**
- **Server**
- **Server**

**Diagram Description:**
- Like simplex free port, except servers can send messages to client.
- Unlike servers, clients do not share a common message queue.
- General scalable solution.
- Less efficient when clients on different computers.
- Which machine has the queue?

**Example:**
- Web search server.
Queue Location in Input Port and Message Latency

Client  

Port (Mailbox)  

Server  

Message Queue  
message$^3$  
message$^2$  

Client  

Unique server has queue  
Message sent from client to server - one hop
**Duplex Output Port**

- **Client**
  - Message Queue
    - message³
    - message²
  - Port (Mailbox)
    - Server
    - Server

- Unique client, has queue
- Message fetched by servers from client - one hop
**Duplex Free Port**

- No distinguished client or server
- In general queue is remote from both a client and server
- Message goes from client to queue machine, and from queue machine to server
**Output/Free Port**

A message goes to a single server – it is not broadcast.
**Multicast/Broadcast Port (Routing)**

A message from client goes to multiple receivers.

In theory could have a single sender.

A message from server goes to specific client.

Example?

Fault tolerance.
SESSION PORT

Port (Mailbox)

Process

Message Queue

message³

message²

message¹

message¹

message¹

message¹

Process

A message from a process goes to every one else

“Reply” goes to sender

Example?

Multi-Player Game, Replicated File System
ACCESS/ROUTING

Set of senders and receivers?

- Bound Port
- Input Port
- Output Port
- Free Port
- Broadcast Port
- Session Port

Simplex or (Replying) Duplex

Port (Mailbox)

Process/Thread

Process/Thread
DISTINCTION BASED ON ACCESS?

- Xinu
- Pipes
- Sockets
- NIO
- RMI
**XINU**

- send (<thread_id>, <int expression>)
- int receive ()
- int recvclr ()
**Xinu: Simplex Input Port**

- Each thread (process) has a single built-in input port.
- A la each Java monitor having a single built in condition.
- Less flexible.
- No need for port creation operations.
- Thread id is the port id.
- Not duplex, but sender can be sent back a message on its input port.
Pipes: Simplex Bound Port?

Sender

Port (Mailbox)

Receiver

man 2 pipe | more

Command interpreter connects output of a process to input of another process.
**Duplex Free Port**

Some process calls pipe() to create the port

Children of pipe creator can send and receive messages

Usually one child writes and another reads
Socket/NIO Access: Many Kinds of Sockets

- Process
- Server Socket
- Stream Socket
- Datagram Socket
- Process

Socket
**Server Socket: Duplex Input Port**

- **Server socket is used to create regular sockets**
- **Represent a non-stream input port handling multiple clients**
- **Each client connects to it to create a dedicated stream duplex port**

Diagram:
- **Client**
- **Server Socket**
- **Server**
STREAM SOCKET: DUPLEX BOUND PORT?
Regular Stream Socket: Duplex/Free Port

- Regular socket represents a duplex port
- Children processes share descriptors, so actually free port
DATAGRAM SERVER SOCKET: SIMPLEX INPUT PORT

- No need to create ServerSocket
- Each client also creates a simplex input port to receive replies
- Address of client simplex port sent with each message
RMI Access?

Client Object

Server Proxy

m(p₁, … pᴺ)

Server Object

m(p₁, … pᴺ)
RMI: Replying Duplex Input Port

Can imagine simplex, bound, output, free port semantics for RMI
**Issues in Message Passing**

- **Process/Thread**
  - Message Queue
  - Message 1
  - Message 2
  - Message 3
  - Message 4

- **Port (Mailbox)**
  - Reliable message delivery?
  - In-order message delivery?
  - Port access?
  - Semantics?
  - Synchronous vs asynchronous?
  - Blocking vs non-blocking?
  - Buffering of messages at queue?
  - Location of communicating threads?
RMI vs. Xinu?

Client Object

Server Proxy

Server Object

send (<thread_id>, <int expression>)

int receive ()

int recvclr ()
What does it mean for a process/thread to consume a message produced by another process/thread?
**Remote Assignment (Data Communication)**

- **Port (Mailbox)**
  - `<expression>`
  - **Message is data to be remotely assigned to a variable**

- **Process/Thread**
  - `<expression>`
  - **The result of send is to assign a sent expression to be assigned to some variable (typically called buffer) in the remote process/thread**

- **Process/Thread**
  - `<expression>`
  - **Type of expression depends on mechanisms and whether language support is provided to define primitives**

- **CSP (Hoare)**
  - `P!<Expression>`
  - `Q?<Var>`
RMI vs. Xinu? (Review)

Client Object

Server Proxy

m(p_1, \ldots , p_N)

Server Object

m(p_1, \ldots , p_N)

send (<thread_id>, <int expression>)

int receive ()

int recvclr ()
**Send Semantics? (Review)**

- **Process/Thread**
  - Message production
  - Port (Mailbox)
    - What does it mean for a process/thread to consume a message produced by another process/thread
  - Message consumption
  - Process/Thread
The result of send is to assign a sent expression to be assigned to some variable (typically called buffer) in the remote process/thread.

Type of expression depends on mechanisms and whether language support is provided to define primitives.

Message is data to be remotely assigned to a variable.
**Remote Procedure Call (RPC)**

Message is request to be remotely executed

The result of a send is to invoke some method in the remote receiving thread/process

Usually built on top of object communication
VARIATIONS OF REMOTE ASSIGNMENT

- Socket/NIO Stream
- Socket/NIO Datagram
- Socket Object Stream
Byte Data Communication

Process/Thread

Provided by OS interface
Assume expression is a byte sequence
It is assigned to a variable that can hold such a sequence

bytes

Port (Mailbox)

bytes

bytes

bytes

Process/Thread

bytes
Byte Communication Communication

Process/Thread

Assume expression is a single byte

Byte blocks can be communicated through libraries

These blocks are not known to port queue

Port (Mailbox)

Socket/NIO Stream

<var>byte</var>
**Block Communication**

- **Process/Thread**
  - Assume expression is a variable sized byte sequence
- **Port (Mailbox)**
  - Port queue is in terms of byte blocks
- **Process/Thread**
  - Socket/NIO Datagram?
**Object Data Communication**

1. **Process/Thread**
   - Provided by OO Language
   - Assume expression is a location-independent object
   - It is assigned to a variable that can hold such an object
   - Typically implemented on top of stream or block communication
   - Typically port queue does not understand objects

2. **Port (Mailbox)**
   - <expression>

3. **Process/Thread**
   - <expression>
Can one simulate one with the other?
**Simulating Remote Procedure Call**

1. **Goal is to call a method with params**
2. **Send expressions encoding method and parameters**
3. **These are assigned to corresponding remote variables**
4. **Side effect of assignment is to call method with parameters**
5. **Efficiency?**
6. **Several simulating calls for one simulated call – less efficient if OS involved**

Diagram:
- Process/Thread
  - `<method>`
  - `<params>`
- Port (Mailbox)
  - `<method>`
  - `<params>`
- Process/Thread
  - `<method>`
  - `<params>`
  - `<method>`
  - `( <params> )`
**Simulating Remote Assignment**

- **Process/Thread**
  - `assign (<expression>)`

- **Port (Mailbox)**
  - `assign (<expression>)`

- **Process/Thread**
  - `<expression>`
  - `assign (<expression>)`

**Goal is to assign some expression to a buffer variable**

- Define special assign method with single formal parameter
- Call this method with the expression to be assigned to the buffer variable
- Side effect of method call is to assign actual parameter to buffer variable

**Efficiency?**

- More expensive RPC to do RA.
**Remote Procedure Call vs. Remote Assignment**

- **Remote Procedure Call**
  - Suitable when requests must be serviced
  - E.g. join a session

- **Remote Assignment**
  - Can simulate remote assignment
  - Simulation awkward and not efficient

- **Remote assignment suitable when data must be sent**
  - E.g. Next command in simulation

- **Remote assignment can implement RPC**
  - However, simulation awkward and not as efficient
Other operations can vary

Executed by process that sends expression or calls method

Every system will have a send operation

Nature depends on semantics and system

<port><expression>

send(<port>, <expression>)

<port><method><params>

call(<port>, <method>, <params>)

Multiple or single method may be associated with port

Assuming multiple; in later examples, a port names a single method

<var>

Process/Thread

Process/Thread

<method>(<params>)
Language/Compiler Support: Type Checking and Special Syntax

- send(<port>, <expression>)
  
  Allows typed <expression> of arbitrary type to be assigned to <var> of arbitrary type

- call(<port>, <method>, <params>)
  
  Allows safe method calls, where name is port is associated with (one or more) signatures

- send <port>(<typed expression>)
- send load(5.2)
- send echoer(“hello”)
- send <port/method> (<actual params>);
- send add(5,3);
- send open(“rpc.doc”);

In example, single signature, add(int, int), associated with port, and procedure name is port name
Explicit Receive (Language Support)

**send** `<port>(<expr>)`

**send** `load(2.2)`

**Port (Mailbox)**

**receive** `<port>(<var>)`

**receive** `load (newVal)`

**Special call by receiver to indicate willingness to receive**

**send** `<port>(<actual parms>)`

**send** `add (5, 3)`

**<method> (<params>)**

**receive** `<type> <port>(<formal param declarations>) {<body>}`

**receive** `int add(int p1,int p2){
  return p1 + p2 
}`
System provides way to automatically assign some known matching variable or call some known matching procedure when message arrives.

These could have been registered by a special register call.

```
register <type> <port>(<formal param declarations>) {<body>}
register int add(int p1, int p2){
    return p1 + p2 }
```
**Reply?**

- `send <port>(<expr>)`
- `send <port>(<actual parms>)`
- `reply (<expr>)`
- `reply (<actual parms>)`

**Syntax and semantics?**

- Last sender is implicit port
- Could reply multiple times according to this definition
Issues in Message Passing

Process/Thread

Message Queue

Port (Mailbox)

Reliable message delivery?

In-order message delivery?

Port access?

Semantics?

Synchronous vs asynchronous?

Blocking vs non-blocking?

Buffering of messages at queue?

Location of communicating threads?
**Synchronous vs. Asynchronous**

- **Synchronous**: Operation invoker waits until the operation finishes.
- **Asynchronous**: Operation invoker does not wait until completion.
  - Some other operation (e.g., software interrupt) needed to wait for result or completion status.
Synchronous vs. Asynchronous vs. Blocking Operations

operation(<parms>)

read(file)

send(loadAvgPort, 1.2)

Blocking: Operation invoker waits, unblocking possibly before, until, or after operation completion

Synchronous is always blocking

Blocking is not always synchronous

Logical blocking times?
Synchronous vs. Asynchronous vs. Blocking Operations (Review)

Blocking: Operation invoker waits, unblocking possibly before, until, or after operation completion

Synchronous is always blocking

Blocking is not always synchronous

Logical blocking times?

<table>
<thead>
<tr>
<th>operation(&lt;parms&gt;)</th>
</tr>
</thead>
<tbody>
<tr>
<td>read(file)</td>
</tr>
<tr>
<td>send(loadAvgPort, 1.2)</td>
</tr>
</tbody>
</table>
Blocking (Logical) Times? a la Binding Time

Binding time: When is some property bound to an entity?

- E.g. when variable bound to to an address, value

Program writing, compilation, link, load, runtime

Tied to phases in the lifetime of a program

Logical times for blocking?

Phases in the lifetime of a message?

Communication pipe line?
**Blocking Times?: Message Pipe Line**

- **send(<port>, <expression>)**
- **call(<port>, <method>, <params>)**

**Operation started**

- **Message in Source System Buffer**
- **Message in Destination System Buffer**

**Destination thread/process starts operation**

**Destination thread/process finishes operation**
# Blocking Times Pros

<table>
<thead>
<tr>
<th>Event</th>
<th>Description</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>Operation started</td>
<td>No waiting, most concurrency (without using extra thread)</td>
<td>Sending load average</td>
</tr>
<tr>
<td>Message in Source System Buffer</td>
<td>Waits until buffer available, prevents flooding</td>
<td>Sending tele-pointers</td>
</tr>
<tr>
<td>Message in Destination System Buffer</td>
<td>Sender knows message did not get lost in network</td>
<td>Sending load average on unreliable port</td>
</tr>
<tr>
<td>Destination thread/process starts operation</td>
<td>Sender knows receiving process did not fail or ignore message</td>
<td>Remote animation started</td>
</tr>
<tr>
<td>Destination thread/process finishes operation</td>
<td>Sender knows operation finished</td>
<td>Reservation made by airline, file received by Dropbox</td>
</tr>
</tbody>
</table>
**Late Blocking and Buffering**

- **send(<port>, <expression>)**
  - Sender buffer

- **call(<port>, <method>, <params>)**
  - Buffer (stack)

- **Message in Source System Buffer**
  - Operation started

- **Message in Destination System Buffer**
  - Destination thread/process starts operation

- **Destination thread/process finishes operation**

- **No flooding!**
  - System buffer

- **Flush(port)**

- **Coalescing more important in byte communication**
  - Byte communication cannot allow sender to wait until data in system buffer

- **Size of source and destination system buffers is an issue**
  - If sender waits, message can be kept in user buffer until it is in destination system buffer

- **However, multiple messages from thread cannot be coalesced into one network message**

- **Operation started**
  - Thread

- **Thread**

- **Send<port>**

- **System buffer**
**Using Sender Buffer**

Async NIO has direct buffer to prevent copying

Programmer can use select to determine when buffer is available
**Synchronous vs. Asynchronous Receive (Semantics)**

Synchronous: Operation invoker waits until the operation finishes

Synchronous receive: Receive blocks until remote assignment or procedure call finishes

Asynchronous: Operation invoker initiates operation and does not wait until completion

Asynchronous receive: Receive provides buffers to receive <expression> or <method> call

Some other operation (e.g. software interrupt) needed to wait for result or completion status
**Implicit vs. Async Receive (Semantics)?**

System provides way to automatically assign some known matching variable or call some known matching procedure when message arrives.

These could have been registered by a special register call for all receives.

In async receive, a special call made for each message.

```
register <port>(<var>)

register load (newVal)
```

```
send <port>(<expr>)

send load (2.2)

send add (5, 3)
```

```
(method)

register <type> <port>(<formal param declarations>) {<body>}

register int add(int p1, int p2) {
  return p1 + p2
}
**Blocking Times in Sockets**

```java
socket.connect(
    new InetSocketAddress(host, port));

Socket socket =
    serverSocket.accept();

Block until server accepts
connection to server socket

Block until next client tries to
contact the server socket

outputStream = socket.getOutputStream();
outputStream.write(buf, offset, length);

Block until in system buffer

inputStream = socket.getInputStream();
int retVal =
    inputStream.read(buf, offset, length);

Block until <= length >=1 bytes received
```
**Asynchronous RPC?**

- Asynchronous RPC semantics?
- Starting remote thread
- Example?
- A remote animation
**Rendezvous in Sync RPC Send, Receive**

- **Synchronous port send**
  - wait
  - Receive code executed

- **Synchronous RPC port receive**
  - Rendezvous
  - Receive code executed

- **Synchronous port send**
  - wait

- **Synchronous RPC port receive**
  - wait
  - Receive code executed

- **Receiver may make the sender wait**
- **Sender may make the receiver wait**

*(Ada) Rendezvous: when RPC is executing – both sender and receiver are waiting*
### Synchronous vs. Asynchronous Receive (Pros/Cons)

<table>
<thead>
<tr>
<th><strong>Sync:</strong> no need to have separate operation to determine when operation finishes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sometimes receiver does not need notification, shared memory model</td>
</tr>
<tr>
<td>Alpha/beta search optimization parameters, when receiver next looks at them, the value may have been set</td>
</tr>
<tr>
<td>Receiver can block on a port on which a message does not arrive</td>
</tr>
</tbody>
</table>
**WAITING ON MULTIPLE RECEIPTS**

- **receive**: `<port1>(<var>)`

New construct for single thread and sync receive?

A single thread/process can expect messages on multiple ports

Synchronous receive → fork a thread for each receive (inefficient)

Asynchronous receive → a single thread can wait on multiple receives
**Abstract Sync Select with Sync Receive**

- **select**
  - receive ... <port<sub>1</sub>> ...  
  - receive ... <port<sub>n</sub>> ...  
- **end**

Select operation waits until a matching send arrives for one of the receives.

- It completes when the receive completes.
- If more than one matching send?
  - Pick one non deterministically.
**Usually Receive All Requests**

```
loop
  select
    receive ... <port^1> ...
    receive ... <port^n> ...
  end
end
```

Select is typically in a loop

Each receive is executed atomically
GUARDED RECEIVE

A receive is matched if a matching send arrives and the guard evaluates to true

Ensure preconditions, block synchronous sender
loop

select {
    when size < MAX_SIZE:
        receive void put(ElementType element) {
            buffer[nextIn] = element;
            nextIn = (nextIn + 1) % MAX_SIZE;
            size++;
        }
    when size > 0:
        receive ElementType get() {
            ElementType retVal = 
                (ElementType) buffer[nextOut];
            nextOut = (nextOut + 1) % MAX_SIZE;
            size--;
            return retVal;
        }
}
**Thread/Process vs. Monitor**

- **Monitor**
  - Method 1
  - Method 2

- **Process/Thread**
  - Call
  - Send

- **Process/Thread**
  - Call
  - Send
**Shared Memory/Message Passing Duality**

```java
loop
    select {
        when size < MAX_SIZE:
            receive void put(ElementType element) {
                buffer[nextIn] = element;
                nextIn = (nextIn + 1) % MAX_SIZE;
                size++;
            }
        when size > 0:
            receive ElementType get() {
                ElementType retVal =
                    (ElementType) buffer[nextOut];
                nextOut = (nextOut + 1) % MAX_SIZE;
                size--;
                return retVal;
            }
    }
}
```

```java
public synchronized void put(ElementType element) {
    while (size >= MAX_SIZE) {
        nonFull.condWait();
    }
    buffer[nextIn] = element;
    nextIn = (nextIn + 1) % MAX_SIZE;
    size++;
    nonEmpty.condBroadcastSignal();
}
```

```java
public synchronized ElementType get() {
    while (size == 0) {
        nonEmpty.condWait();
    }
    ElementType retVal =
        (ElementType) buffer[nextOut];
    nextOut = (nextOut + 1) % MAX_SIZE;
    size--;
    nonFull.condSignal()
    return retVal;
}
```

---

**Synchronous RPC port receive**

**Procedure**

**Synchronous RPC port send**

**Guard**

**Entry procedure call**

**Condition wait**

**Signal and return**

**Entry procedure**

**Loop and select 1 receive at a time**
**Flexibility Comparison**

- Guard evaluated once at start
- One signal, which is a return
- Multiple waits in entry procedure
- Multiple signals
- Do we need the extra monitor flexibility?
  - Probability not
**Explicit Sync vs. Implicit Receive (Pros, Cons)**

- Can be used to make sender wait when sync send
- Necessary to simulate monitors
- Sender has to wait
- Loop, select, receive needed

If other mechanisms available to make sender wait, then implicit receive better

Usually implicit receive for RPC, and explicit receive for remote assignment (data communication)
LOCATION OF COMMUNICATING THREADS?
LOCATION OF COMMUNICATING THREADS?
INTRA-PROCESS (ADDRESS SPACE) COMMUNICATION

Intra-Process: Avoids global memory

Rationale?
INTRA-COMPUTER MESSAGE PASSING

Inter-Process: Cooperating processes

Is | more
INTER-COMPUTER MESSAGE PASSING

Inter-Computer: Cooperating remote processes

File, web, ..., servers

Inter-Computer \(\rightarrow\) Inter-Process
Location: Applications

- **Intra-Process:** Avoids global memory
- **Inter-Process:** Cooperating processes
  - `ls | more`
- **Inter-Computer:** Cooperating remote processes
  - File, web, ..., servers

More flexible and distant the message passing, the more complex the API and implementation.
HOW TO NAME PORT?
Intra-Process (Address Space) Communication

- In same process, shared memory
- A single port variable can be used by all threads

Xinu, CSP
**Intra-Computer Message Passing**

- **Port**
  - Process-specific local handle
  - Process-specific local handle

**Similar to naming shared file**

**With different processes, on same OS, different resource (file) descriptors**
name common port

**These can be inherited from common parent process (like standard I/O)**
or they can have external name like a file name

**Pipes**
GENERAL MESSAGE PASSING: EXTERNAL NAME

Remote Handle

Process

Cannot inherit handle from common parent

Port

Local Handle

Process

Need an external name on which both parties agree
# Alternatives

<table>
<thead>
<tr>
<th>Datagram Socket</th>
</tr>
</thead>
<tbody>
<tr>
<td>Can use external name for each message, specifying machine address each time</td>
</tr>
<tr>
<td>Not compatible if underlying communication mechanism requires handshake and thus connection</td>
</tr>
<tr>
<td>Access control needed on each message</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Stream Socket</th>
</tr>
</thead>
<tbody>
<tr>
<td>External name may be bound to handle, which may result in handshake between two machines and authentication and access control at connection time</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Java RMI</th>
</tr>
</thead>
<tbody>
<tr>
<td>An external name server keeps name to handle binding</td>
</tr>
</tbody>
</table>
Connection vs. Non Connection

Non connection based IPC can use any of these mechanisms.

Connection based ipc uses handles, not involving external name server, representing specific connection that defines data associated with connection such as offset in stream and rights.
IPC DESIGN SPACE

Shared Memory

Software Interrupts

Message Passing (Multiple Dimensions)

- Reliable?
- In-order?
- Access?
- RPC or RA
- Byte, Block or Object
- Buffer sizes?
- Send blocking times
- Language support?
- Explicit receive?
- Receive Blocking times?
- Select?
- Location of communicating threads?
- Reply?
- Reply blocking times?
- Naming?