

The Producer-Consumer Problem

- One process is a producer of information; another is a consumer of that information
- Processes communicate through a bounded (fixed-size) circular buffer

```
var buffer: array[0..n-1] of items; /* circular array */
in = 0
out = 0
```

```
/* producer */
repeat forever
...
produce item nextp
...
while (in+1 mod n == out)
do nothing
buffer[in] = nextp
in = in+1 mod n
end repeat

/* consumer */
repeat forever
while (in == out)
do nothing
nextc = buffer[out]
out = out+1 mod n
...
consume item nextc
...
end repeat
```

Message Passing using Send & Receive

- Blocking send:
 - `send(destination-process, message)`
 - Sends a message to another process, then *blocks* (i.e., gets suspended by OS) until message is received
- Blocking receive:
 - `receive(source-process, message)`
 - Blocks until a message is received (may be minutes, hours, ...)

Producer-Consumer problem:

```
/* producer */
repeat forever
...
produce item nextp
...
send(consumer, nextp)
end repeat

/* consumer */
repeat forever
receive(producer, nextc)
...
consume item nextc
...
end repeat
```

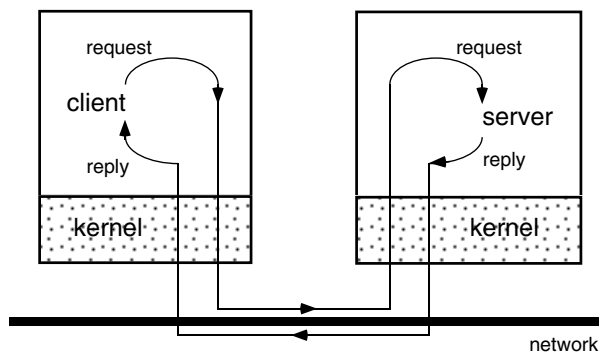
Direct vs. Indirect Communication

- Direct communication — explicitly name the process you're communicating with
 - `send(destination-process, message)`
 - `receive(source-process, message)`
 - Variation: receiver may be able to use a "wildcard" to receive from any source
 - Receiver can not distinguish between multiple "types" of messages from sender
- Indirect communication — communicate using mailboxes (owned by receiver)
 - `send(mailbox, message)`
 - `receive(mailbox, message)`
 - Variation: ... "wildcard" to receive from any source into that mailbox
 - Receiver can distinguish between multiple "types" of messages from sender
 - Some systems use "tags" instead of mailboxes

Buffering

- Link may be able to temporarily queue some messages during communication
- Zero capacity: (queue of length 0)
 - Blocking send operation
 - Sender must wait until receiver receives the message — this synchronization to exchange data is called a *rendezvous*
- Bounded capacity: (queue of length n)
 - Blocking send operation
 - If receiver's queue is has free space, new message is put on queue, and sender can continue executing immediately
 - If queue is full, sender must block until space is available in the queue
- Unbounded capacity: (infinite queue)
 - Non-blocking send operation
 - Sender can always continue

Client / Server Model using Message Passing



Client / server model

- *Server* = process (or collection of processes) that provides a *service*
 - Example: name service, file service
- *Client* — process that uses the service
- Request / reply protocol:
 - Client sends **request** message to server, asking it to perform some service
 - Server performs service, sends **reply** message containing results or error code

Remote Procedure Call (RPC)

RPC mechanism:

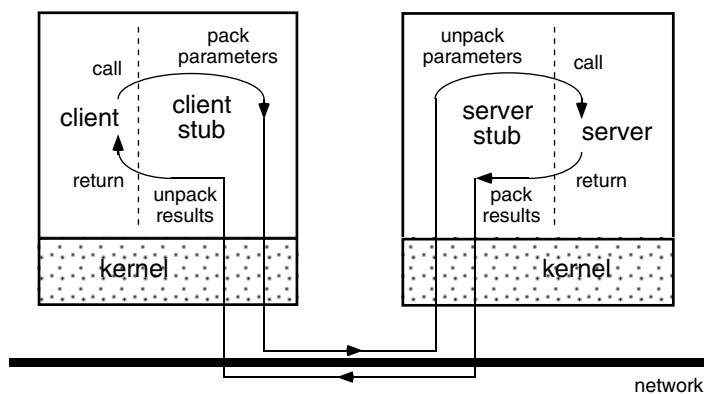
- Hides message-passing I/O from the programmer
- Looks (almost) like a procedure call — but client invokes a procedure on a server

RPC invocation (high-level view):

- Calling process (client) is suspended
- Parameters of procedure are passed across network to called process (server)
- Server executes procedure
- Return parameters are sent back across network
- Calling process resumes

- Invented by Birrell & Nelson at Xerox PARC, described in February 1984 *ACM Transactions on Computer Systems*

Client / Server Model using Remote Procedure Calls (RPCs)



- Each RPC invocation by a client process calls a *client stub*, which builds a message and sends it to a *server stub*
- The server stub uses the message to generate a local procedure call to the server
- If the local procedure call returns a value, the server stub builds a message and sends it to the client stub, which receives it and returns the result(s) to the client

Remote Method Invocation (RMI)

RMI mechanism:

- A Java mechanism similar to RPCs
- Allows a Java program on one machine to invoke a method on a remote object
- Client *stub* creates a *parcel*, sends to *skeleton* on the server side

