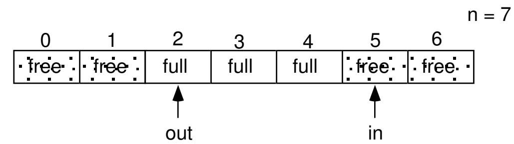
Cooperating Processes

- Processes can cooperate with each other to accomplish a single task.
- Cooperating processes can:
  - Improve performance by overlapping activities or performing work in parallel
  - Enable an application to achieve a better program structure as a set of cooperating processes, where each is smaller than a single monolithic program
  - Easily share information
- Issues:
  - How do the processes communicate?
  - How do the processes share data?

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 */                      /* consumer */
repeat forever                      repeat forever
    …                                   while (in == out)
    produce item nextp                      do nothing
    …                                   nextc = buffer[out]
    while (in+1 mod n == out)           out = out+1 mod n
        do nothing                      …
    buffer[in] = nextp                  consume item nextc
    in = in+1 mod n                     …
end repeat                          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 */                      /* consumer */
repeat forever                      repeat forever
    …                                   receive(producer,nextc)
    produce item nextp                  …
    …                                   consume item nextc
    send(consumer, nextp)               …
end repeat                          end repeat
```

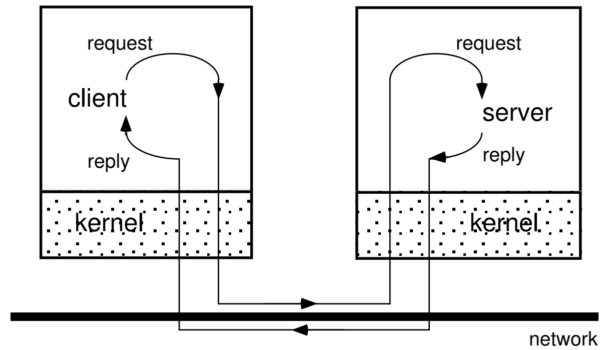

Direct vs. Indirect Communication

- Direct communication — explicitly name the process you're communicating with
  - send(*destination-process*, *message*)
  - receive(*source-process*, *message*)
  - Link is associated with exactly two processes
    - Between any two processes, there exists at most one link
    - The link may be unidirectional, but is usually bidirectional
- Indirect communication — communicate using mailboxes (owned by receiver)
  - send(*mailbox*, *message*)
  - receive(*mailbox*, *message*)
  - Link is associated with two or more processes that share a mailbox
    - Between any two processes, there may be a number of links
    - The link may be either unidirectional or bidirectional

Buffering

- Link may have some capacity that determines the number of message that can be temporarily queued in it
- Zero capacity: (queue of length 0)
  - No messages wait
  - Sender must wait until receiver receives the message — this synchronization to exchange data is called a *rendezvous*
- Bounded capacity: (queue of length *n*)
  - If receiver's queue is not full, 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)
  - Sender can always continue

Client / Server Model using Message Passing

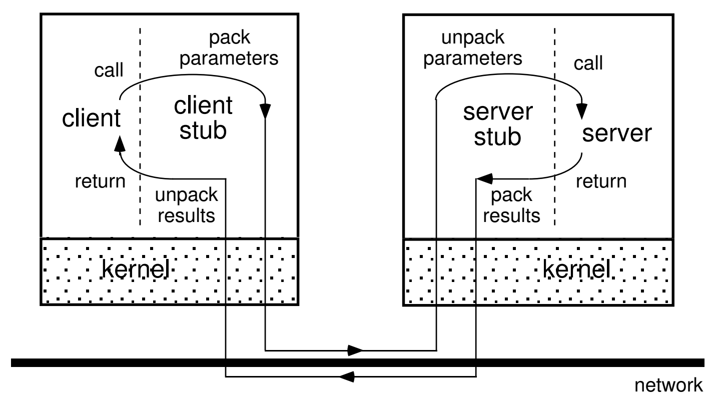


- Client / server model
  - *Server* = process (or collection of processes) that provides a *service*
    - Example: name service, file service
  - *Client* — process that uses the service
  - Request / reply protocol:
    - Client sends **request** message to server, asking it to perform some service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

RPC Invocation (More Detailed)

1. Client procedure calls the client stub
2. Client stub packs parameters into message and traps to the kernel
3. Kernel sends message to remote kernel
4. Remote kernel gives message to server stub
5. Server stub unpacks parameters and calls server
6. Server executes procedure and returns results to server stub
7. Server stub packs result(s) in message and traps to kernel
8. Remote kernel sends message to local kernel
9. Local kernel gives message to client stub
10. Client stub unpacks result(s) and returns them to client