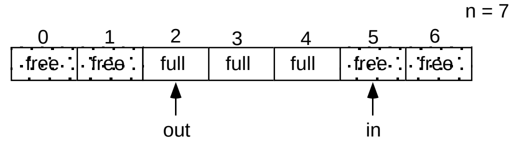
Communication Models in Distributed Systems

- Peer-to-peer
  - Producer / consumer
- Client / server
  - Clients ask dedicated server to perform some specific service
- Central coordinator (many-to-one)
  - Nodes send information to coordinator; coordinator makes decision
  - Central point of failure
- Distributed consensus (one-to-many)
  - Nodes send information to each other; group as a whole reaches a consensus
  - Large amount of communication required

The Producer-Consumer Problem

- One process is a producer of information; another is a consumer of that information
- Solution when the two processes have a shared memory in common:

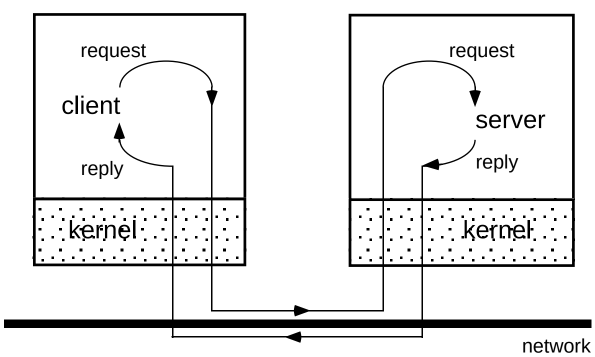
```
var buffer:  array[0..n-1] of items;     /* circular array */
in = 0
out = 0
```




```
/* 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
```


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

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
```

Buffering

- Link may be able to temporarily queue some messages during communication
- Zero capacity: (queue of length 0)
  - Blocking communication
  - 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)
  - Non-blocking communication
  - Sender can always continue

Non-blocking Send & Receive

- Non-blocking send:
  - Sends, then goes on to next instruction without waiting for an acknowledgment
  - Advantage: sending process can execute in parallel with message transmission
  - Problem: must avoid modifying message buffer until message has been received (but how do you know?)
    1. Copy message from user space to kernel space, then let process continue
    2. Keep message in user space, have kernel send interrupt when message has been received (difficult to program)
- Non-blocking receive:
  - Receive returns with buffer, but doesn't know if there's a message there or not
    - Must poll or receive interrupt when message is ready and process should perform a receive (difficult to program)

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 <u>can not</u> 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 <u>can</u> distinguish between multiple "types" of messages from sender
  - Some systems use "tags" instead of mailboxes

LAM / MPI

- MPI = Message Passing Interface
- LAM = Local Area Multicomputer
  - Implementation of MPI from the OSC that "simulates" a multicomputer
- See AOS class web page
  - "Using LAM/MPI in the KSU MCS Dept."
  - "MAN T&EC MPI Tutorial"
  - Other information is optional
- MPI uses the SPMD (Same Program, Different Data) programming model
  - Same program runs on all machines
  - May choose to have one program run "master" code, and others run "worker / slave" code