Cooperating processes can communicate using:
shared memory
or
message-passing
Using shared memory (producer/consumer)

```c
#define N 10
item buffer[N];
int in = 0;
int out = 0;

While (1) {
    nextProduced = Produce();
    while ((((in+1)%N) == out) ; /* do nothing */
        buffer[in] = nextProduced;
        in = (in + 1) % N;
}

While (1) {
    while ( in == out ) ; /* do nothing */
        nextConsumed = buffer[out];
        out = (out + 1) % N;
        Consume( nextConsumed );
}
```
Message-Passing

• 2 primitive operations:
  • Send(...)  
  • Receive(...)  

• Issues:
  • Naming: direct or indirect  
  • Buffering  
  • Synchronization  
  • Copying  

• Can be used for network communications (e.g., client-server)
Naming: Direct Communication

- Names the process to send to (receive from)
  - Send ( P, msg ); // sends the msg to P
  - Receive ( Q, msg ); // receives a message from Q
- Processes need to know each other's names
- Asymmetric version:
  - Send ( P, msg )
  - Receive ( id, msg ); // returns id of sending process
Naming: Indirect Communication

- Messages sent to (received from) a mailbox or port
- Each mailbox has an id
- Process can use possibly many mailboxes
- Processes must share a mailbox to communicate
- Primitives:
  - Send ( A, msg ); // send msg to mailbox A
  - Receive ( A, msg ); // receive msg from mailbox A
- Possibly many senders, but receivers?...
Synchronization

- **Blocking** = synchronous
- **Nonblocking** = asynchronous
- **Blocking Send**: sender blocks until msg received (at process or mailbox)
- **Nonblocking Send**: sender resumes immed.
- **Blocking Receive**: receiver blocks until a message is available
- **Nonblocking Receive**: receiver is returned a message or NULL
- **Rendezvous** = blocking send + blocking receive
Buffering

- Messages sent but not received can temporarily be kept in a queue
  - Zero capacity: no messages queued, sender must block until receiver receives.
  - Bounded capacity: queue holds up to n messages. Send() and queue is not full, the message is queued and sender returns. If Send and queue is full, sender must block.
  - Unbounded capacity: infinite queue, sender never blocks. (?)
A message can be passed by value, requiring a copy to be made (slow)

Passing a message by reference can avoid the need to copy (fast), but requires some form of shared memory.
Client-server communication

- Used to access resources at a remote computer (the server)
- No shared memory possible, use messages over a network
- Various failures possible
- Typical protocol:
  - Client sends a request message to server, and blocks awaiting reply
  - Server receives request and processes it
  - Server sends reply message to client, and blocks awaiting next request
  - Client continues
Sockets

- Socket = communications endpoint
- Socket “name” = IP address + port number
- Port numbers < 1024: standard services (e.g. 23=telnet, 80=http)
- Port numbers >= 1024: applications
- Connection-oriented sockets (TCP) give an error-corrected, unstructured stream of bytes.
- Connectionless sockets (UDP) provide datagram communications (possibly lost, duplicated, or reordered packets).
Remote Procedure Calls (RPC)

- Uses a request-reply protocol to mimic typical procedure call semantics
- Client calls a local "stub" procedure
- Stub *marshalls* arguments, sends request message to server, blocks awaiting reply
- Request messages contain the desired procedure's number, and actual arguments
- Marshalling gathers args into a form for transmission, *external data representation* (XDR).
Remote Method Invocation (RMI)

- Supports OOP: call a method on a remote object
- A stub object, with same interface as the remote one, is used by the client.
- As with RPC, the stub marshalls the args and send to remote server.
- Server has a “skeleton” to unmarshall args, call method, marshall result, send to client stub.
- ex) Java, CORBA, SOAP