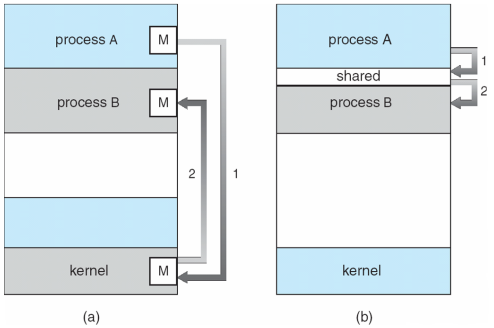
Interprocess Communication

- Processes within a system may be *independent* or *cooperating*
- Cooperating process can affect or be affected by other processes.
- Reasons for cooperating processes:
  - Information sharing, e.g. shared files
  - Computation speed-up (sometimes, depending on hardware)
  - Modularity
  - Convenience
- Cooperating processes need some mechanism of interprocess communication (IPC), which is provided by their OS.
- Two models of IPC
  - Shared memory
  - Message passing

Communication Models: Message Passing and shared Memory



- a – Message passing via the kernel.
- b – Use of shared memory: more efficient, since less (or no) context switching, though complicated by shared data concurrency issues.

Producer-Consumer Problem: Bounded-Buffer Solution

- Useful problem for understanding issues of processes that cooperated via shared memory.
  - producer process produces information into some buffer that is consumed by some consumer process.
- Note, this example does not take into consideration important concurrency issues, which we will explore in a later lecture.

```
#define BUFFER_SIZE 10
// Define the thing we wish to store.
typedef struct {
  ...
} Item;

// Define a buffer of items with two indexes: in and out.
// Buffer is empty when in==out;
// full when ((in+1)%BUFFER_SIZE)==out
Item buffer[BUFFER_SIZE];
int in = 0;
int out = 0;
```


Producer Process Code

```
// Produce items until the cows come home.
while (TRUE) {
  // Produce an item.
  Item next_produced = [some new item to add to buffer];

  // Wait one place behind the next item to be consumed - so we don't
  // write to items that have yet to be consumed.
  while (((in + 1) % BUFFER_SIZE) == out); // <- Spin on condition

  // Store the new item and increment the 'in' index.
  buffer[in] = next_produced;
  in = (in + 1) % BUFFER_SIZE;
}
```

Consumer Process Code

```
while (true) {
  // Wait until there is something to consume.
  while (in == out); // <- Spin on condition
  // Get the next item from the buffer
  Item next_item = buffer[out];
  [process next_item]

  // Increment the out index.
  out = (out + 1) % BUFFER_SIZE;
}
```


IPC via Message Passing

- Mechanism for processes to communicate and to synchronize their actions
- Message system - processes communicate with each other without resorting to shared variables
- IPC facility in the OS provides two operations:
  - `send(message)`
  - `receive(message)`
- If two processes wish to communicate, they need to:
  - establish a communication link between them
  - exchange messages via send/receive
- Implementation of communication link/channel
  - physical (e.g., shared memory, hardware bus) - we will not worry about the physical implementation here.
  - logical (e.g., abstract channel that may utilise one of many physical technologies, such ethernet, wireless, and several protocol layers, such as IP/UDP|TCP)

Implementation Questions

- How are links established?
- Can a link be associated with more than two processes?
- How many links can there be between every pair of communicating processes?
- What is the capacity of a link?
- Is the size of a message that the link can accommodate fixed or variable?
- Is a link unidirectional (one way) or bi-directional (two way)?

Direct Communication

- Under direct communication, processes must *name each other explicitly*:
  - `send (P, message)` - send a message to process *P*
  - `receive(Q, message)` - receive a message from process *Q*
- (But, we *could* also only name the recipient)
- Properties of communication link
  - Links are established automatically
  - A link is associated with exactly one pair of communicating processes
  - Between each pair there exists exactly one link
  - The link may be unidirectional, but is usually bi-directional
- Either way, we suffer a lack of modularity from processes communicating this way

Indirect Communication

- More flexible, and so more common communication mechanism.
- Messages are sent to and received from mailboxes (a.k.a ports)
  - Each mailbox has a unique id
  - Processes can communicate only if they share a mailbox (*i.e.* if they know the ID of the mailbox)
- Properties of communication link
  - Link established only if processes share a common mailbox
  - A link may be associated with many processes
  - Each pair of processes may share several communication links
  - Link may be unidirectional (send but not receive) or bi-directional (send and receive)

Indirect Communication

- Operations
  - create a new mailbox
  - send and receive messages through mailbox
  - destroy a mailbox
- Primitives are defined as:
  - `create()` - returns the ID of a new mailbox
  - `send(A, message)` - send a message to mailbox A
  - `receive(A, message)` - receive a message from mailbox A
  - `destroy(A)` - destroy mailbox A

Indirect Communication

- Mailbox sharing
  - P_1, P_2, and P_3 share mailbox A
  - P_1, sends; P_2 and P_3 receive
  - Who gets the message - P_2 or P_3?
- Solutions
  - Allow a link to be associated with at most two processes
  - Allow only one process at a time to execute a receive operation
  - Allow the system to select arbitrarily the receiver. Sender is notified who the receiver was.

Synchronisation

- Message passing may be either *blocking* or *non-blocking*
- Blocking is considered *synchronous*
  - Blocking send has the sender block until the message is received
  - Blocking receive has the receiver block until a message is available
- Non-blocking is considered *asynchronous*
  - Non-blocking send has the sender send the message and continue
  - Non-blocking receive has the receiver receive a valid message or null
- If both send and receive are implemented as blocking, we get a rendezvous

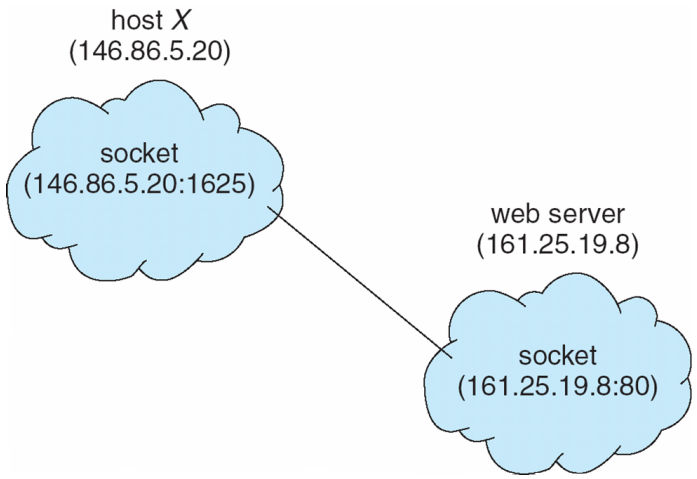
Buffering

- Queue of messages attached to the link; implemented in one of three ways
  - Zero capacity - 0 messages (a.k.a rendezvous)
    - Sender must wait for receiver
  - Bounded capacity - finite length of n messages
    - Sender must wait if link full
  - Unbounded capacity - infinite length
    - Sender never waits

Sockets

- A socket is defined as an abstract endpoint for communication, and is named as the concatenation of IP address and port
  - It is abstract in the sense the the particular network medium is hidden from the application programmer (*e.g.* wireless, wired, network protocols, *etc.*)
- The socket `161.25.19.8:1625` refers to port `1625` on host `161.25.19.8`
- Communication happens between a pair of sockets
- Applications read from and write to sockets.

Socket Communication



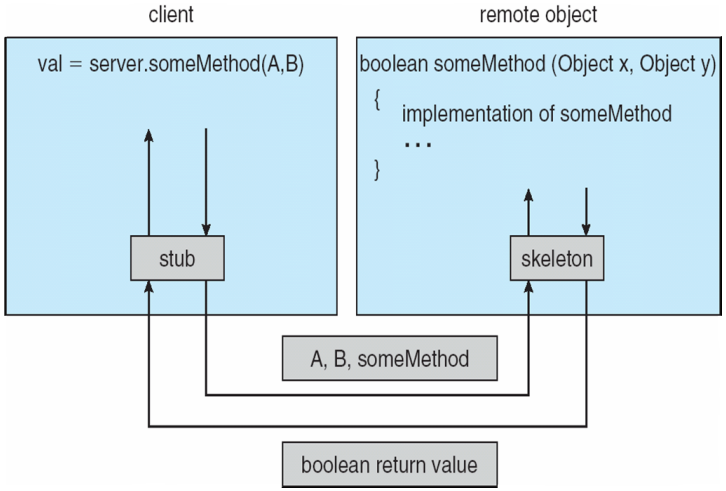
Remote Procedure Calls

- Remote procedure call (RPC) abstracts procedure calls between processes on networked systems
  - RPC is built on top of some message-based communication channel (*e.g.* Sockets)
- Obviously, one process cannot call a function directly on another process (they are in a different address space), especially a remote one, so there is a trick to this:
  - Client uses a *stub* - a client-side proxy for the actual procedure on the server
  - The client-side stub locates the server and marshalls the parameters (*e.g.* native datatypes of the caller) into some universally understood form (*e.g.* XML, big-endian, little-endian, *etc.*)
  - The server-side *skeleton* (the reciprocal of the stub) receives this message, unpacks the marshalled parameters, and executes the requested procedure on the server

Remote Procedure Calls

- The whole point of RPC is to reduce the complexity of network programming by giving the illusion that we can simply call methods on remote machines, without worrying about how to structure the low-level messages.
- And since message structuring is automated by the RPC middleware (a fancy name for software that tries to hide complexity of networking), it is very easy to extend a distributed system by adding new procedures and datatypes.

RPC Architecture



RPC: Invocation Issues

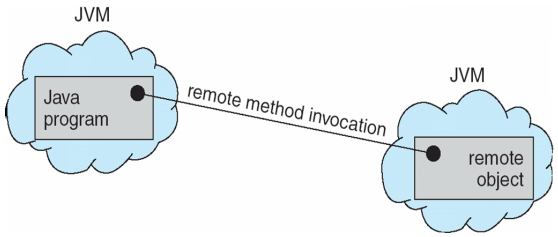
- The idea of RPC is all well and good, but there are some issues to deal with that arise from hiding details of lower-level messaging from the programmer.
- What happens if the request message gets lost in the network, so doesn't reach the server?
  - Perhaps we can resend it a number of times until we get a reply.
  - But then what if the server executes the function several times (*e.g.* mattsBankAccount.decrement_balance(£10))?
  - What if the server gets the request and executes the function but the server's response message gets lost?

RPC: Invocation Semantics

- So the following semantics have been defined, based on particular requirements of communication within the app:
  - *Maybe*: The function will execute once or not at all - useful if we can tolerate some loss in communication and efficiency is important (*e.g.* real-time multiplayer shooting game)
  - *At least once*: the function will execute one or multiple times - useful if efficiency is important and functions can be called multiple times with no ill-effect (*i.e.* functions that do not alter global state)
  - *Exactly once*: the function must execute exactly once - useful for accurate interaction but requires more effort to implement. Remove the risk that the server won't receive the request—implement *at most once* but ACK the receipt and execution of each RPC call. Client resends until ACK received. This is the standard approach of Java RMI.

Remote Method Invocation

- Remote Method Invocation (RMI) is a Java mechanism similar to RPC
- RMI allows a Java program on one machine to invoke a method on a remote object
- Since this is like RPC but instead with objects, there is the possibility to move objects transparently and at run-time from host to host for reasons load balancing, hardware maintenance, *etc.*



Example of POSIX (*i.e.* UNIX-like) Shared Memory IPC

- Process first creates shared memory segment:
  - `segment_id = shmget(IPC_PRIVATE, size, S_IRUSR | S_IWUSR);`
- Process wanting access to that shared memory must attach to it:
  - `shared_memory = (char *) shmat(segment_id, NULL, 0);`
- Now the process could write to the shared memory:
  - `sprintf(shared_memory, "Writing to shared memory");`
- When done, a process can detach the shared memory from its address space:
  - `shmdt(shared_memory);`
- We will have a go with this in a later lecture.

Examples of IPC Systems - Mach (used in Mac OS X)

- Mach, a micro-kernel architecture used within Mac OS X, where communication is message based
  - Even system calls are messages
  - Each task gets two mailboxes at creation, `Kernel` and `Notify`, to communicate with the kernel.
  - Only three system calls needed for message transfer
    - `msg_send()`, `msg_receive()`, `msg_rpc()`
  - The OS may allocate mailboxes in shared memory to reduce inefficient double copying of writes and reads between processes.
  - Mailboxes needed for communication, created via `port_allocate()`

Examples of IPC Systems - Windows XP/Vista

- OS provides support for multiple operating environments (*i.e.* system call APIs) for different types of processes (*e.g.* Windows Processes, MS-DOS processes, POSIX processes, *etc.*) using an easily extended subsystem architecture.
- Message-passing centric, via local procedure call (LPC) facility, so works only between processes on the same system
  - Uses ports (like mailboxes) to establish and maintain communication channels
  - The client (user process) opens a handle to the subsystem's (think server) connection port object, which is well known to all processes.
  - The client sends a connection request
  - The server creates two private communication ports and returns the handle to one of them to the client
  - The client and server use the corresponding port handle to send messages and to listen for replies or receive callbacks

Local Procedure Call in WindowsXP

- Since the size of message ports is limited, for communication of large data types the client and server processes can establish a segment of shared memory called a section object.