Interprocess communication

Introduction

Cooperating processes need to exchange information, as well as synchronize with each other, to perform their collective task(s). The primitives discussed earlier can be used to synchronize the operation of cooperating processes, but they do not convey information between processes.

Methods for effective sharing of information among cooperating processes are collectively known as *inter-process communication (IPC)*. Two basic models are used:

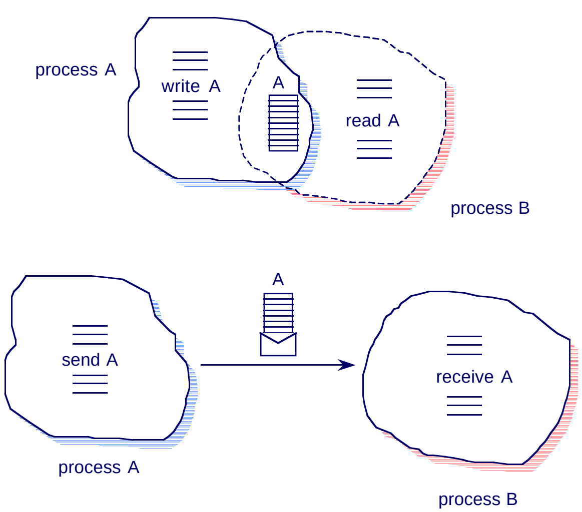
– *shared memory*—“shared data” are directly available to each process in their address spaces.

– *message passing*—“shared data” are explicitly exchanged.

Shared Memory versus Message Passing

shared memory
(e.g., UNIX pipes)

message passing
(e.g., UNIX sockets)



Terminology

A common approach in communication is where one process *sends* some information to another. The information exchanged among processes in this way is called a *message*. A message can be a structured (language) object, specified by its type, or it is specified by its size: fixed length or variable length.

There are two basic operations on messages:

– **send()**—transmission of a message.

– **receive()**—receipt of a message.

The OS component which implements these operations (and more!) is called a ‘‘*message passing*’’ system.

Fundamental questions

A message passing system should answer such questions as:

- When a message is sent, does the sender wait until the message is received or can it continue executing?
- What happens if a process executes a **receive()**, but no message has been sent?
- Can a message be sent to <u>one</u> or to <u>many</u> processes?
- Does a receiver identify the sender of the message or can it accept messages from <u>any</u> sender?
- Where are messages kept while in transit? Capacity?
- ...

Design Issues

Implementation of message passing systems may differ in a number of details that affect the functioning of the **send** and **receive** operations and their parameters. The following are the most important issues in the design and implementation of a message passing system:

- *Form of communication*—messages can be send <u>directly</u> to its recipient or <u>indirectly</u> through an intermediate named object.
- *Buffering*—how and where the messages are stored.
- *Error handling*—how to deal with exception conditions.

Direct communication

The sender and receiver can communicate in either of the following forms:

- *synchronous*—involved processes synchronize at every message. Both **send** and **receive** are *blocking* operations. This form is also known as a *rendezvous*.
- *asynchronous*—the **send** operation is almost always non-blocking. The **receive** operation, however, can have blocking (waiting) or non-blocking (polling) variants.

Direct communication continued

Processes must explicitly name the receiver or sender of a message (symmetric addressing).

- **send** (*P, message*). Send *message* to process *P*.
- **receive** (*Q, message*). Receive *message* from *Q*.

In a client-server system, the server does not have to know the name of a specific client in order to receive a message. In this case, a variant of the **receive** operation can be used (asymmetric addressing).

- **listen** (*ID*, *message*). Receive a pending (posted) *message* from any process; when a message arrives, *ID* is set to the name of the sender.

Direct communication continued

In this form of communication the interconnection between the sender and receiver has the following characteristics:

- A link is established automatically, but the processes need to know each other's identity.
- A unique link is associated with the two processes.
- Each pair of processes has only one link between them.
- The link is usually bi-directional, but it can be uni-directional.

Indirect communication

In case of indirect communication, messages are sent to *mailboxes*, which are special repositories. A message can then be retrieved from this repository.

– **send** (*A, message*). Send a *message* to mailbox *A*.
– **receive** (*A, message*). Receive a *message* from mailbox *A*.

This form of communication decouples the sender and receiver, thus allowing greater flexibility.

Generally, a mailbox is associated with many senders and receivers. In some systems, only one receiver is (statically) associated with a particular mailbox; such a mailbox is often called a *port*.

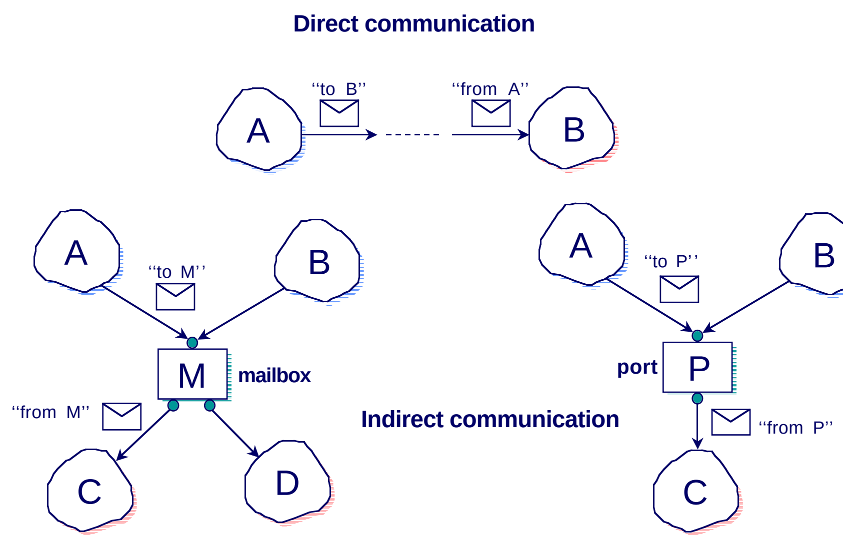
Indirect communication continued

A process that creates a mailbox is the owner (sender). Mailboxes are usually managed by the system.

The interconnection between the sender and receiver has the following characteristics:

- A link is established between two processes only if they "share" a mailbox.
- A link may be associated with more than two processes.
- Communicating processes may have different links between them, each corresponding to one mailbox.
- The link is usually bi-directional, but it can be uni-directional.

Message passing by "picture"



Buffering

Depending on the capacity of the link between the communicating processes, three types of messaging can be implemented:

- *Zero capacity*—used by synchronous communication.
- *Bounded capacity*—when the buffer is full, the sender must wait.
- *Indefinite capacity*—the sender never waits.

Note that, however, in the non-zero capacity cases (asynchronous), the sender is unaware of the status of the message it sends. Hence, additional mechanisms are needed to guarantee the delivery and receipt of a message.

Error handling

In distributed systems, message passing mechanisms extend inter-process communication beyond the machine boundaries. Consequently, messages are occasionally lost, duplicated, delayed, or delivered out of order. The following are the most common ''error'' conditions which requires proper handling:

- *Process terminates*—either a sender or a receiver may terminate before a message is processed.
- *Lost or delayed messages*—a message may be lost (or delayed) in the communications network.
- *Scrambled messages*—a message arrives in an unprocessible state.

Synchronization with messages

The primitives discussed earlier are not suitable for synchronization in distributed systems. For example, semaphores require global memory, whereas monitors require centralized control. Application of such *centralized* mechanisms to distributed environments is not usually practical.

However, message passing is a mechanism suitable not only for inter-process communication, but also for synchronization, in both centralized and distributed environments.

An example

```
typedef message {
   ...
}
const capacity = ... ;
message dummy = {};

int main()
{
   int i;

   create_mailbox( mayconsume );
   create_mailbox( mayproduce );
   for ( i = 0; i < capacity; i++ )
     send( mayproduce, dummy );
   producer();
   consumer();
}
```

```
producer()
{
  message pmsg;

  while ( true ) {
    receive( mayproduce, pmsg );

    < produce >

    send( mayconsume, pmsg );
  }
}
```

```
consumer()
{
  message cmsg;

  while ( true ) {
    receive( mayconsume, cmsg );

    < consume >

    send( mayproduce, cmsg );
  }
}
```

Note: In this example, both send() and receive() are blocking operations.

Other IPC mechanisms

The following are IPC mechanisms found in various flavors of UNIX:

- Pipes
- FIFOs (named pipes)
- Streams and Messages
- System V IPC
  - Message Queues
  - Semaphores
  - Shared Memory
- Sockets (BSD)
- Transport Level Interface (System V)

A case study—UNIX signals

A UNIX *signal*, a rudimentary form of IPC, is used to notify a process of an event. A signal is generated when the event first occurs and delivered when the process takes an action on that signal. A signal is pending when generated but not yet delivered. Signals, also called *software interrupts*, generally occur asynchronously.

A signal can be sent:

- by one process to another, including itself (in the latter case it is synchronous)
- by the kernel to a process

Sending a *signal*

```
kill(int pid, int sig);
```

sends a signal **sig** to the process **pid**. A process sends a signal to itself with

```
raise(int sig);
```

There is no operation to receive a signal. However, a process may declare a function to service a particular signal as:

```
signal(int sig, SIGARG func);
```

Whenever the specified signal **sig** is received, the process is interrupted and **func** is called immediately. In other words, the process catches the signal when it is delivered. On return from **func**, the process resumes its execution from where it was interrupted.

What to do with a *signal*?

Using the **signal()** system call, a process can:

- *Ignore the signal*—only two signals, **SIGKILL** (kill -9 PID) and **SIGSTOP** (Ctrl-Z) cannot be ignored.
- *Catch the signal*—tell the kernel to call a function whenever the signal occurs.
- *Let the default action apply*—depending on the signal, the default action can be:
  - **exit**—perform all activities as if the exit system call is requested.
  - **core**—first produce a core image on disk and then perform the exit activities.
  - **stop**—suspend the process.
  - **ignore**—disregard the signal.

UNIX signals—an example

```
#include < stdio.h >
#include < signal.h >
#include < unistd.h >
#include < stdlib.h >

int main ( void )
{
  int     i ;
  void    catch_signal( int ) ;

    if ( signal( SIGINT, catch_signal ) == SIG_ERR ) {
         perror( "SIGINT failed" ) ;
         exit ( 1 );
    }
    if ( signal( SIGQUIT, catch_signal ) == SIG_ERR ) {
         perror( "SIGQUIT failed" ) ;
         exit( 1 ) ;
    }
    for ( i = 0; ; i++ ) {      /* loop forever */
         printf( "%d\n", i ) ;
         sleep( 1 ) ;
    }
}

void catch_signal( int the_signal ) {
    signal( the_signal, catch_signal ) ;   /* catch again */
    printf( "\nSignal %d received.\n", the_signal ) ;
    if ( the_signal == SIGQUIT ) {
       printf( "Exiting...\n" ) ;
       exit( 3 );
    }
}
```

output

```
% a.out
0
1
2
^C
Signal 2 received.
3
^C
Signal 2 received.
4
5
6
^\
Signal 3 received.
Exiting...
%
```