Computação em Larga Escala

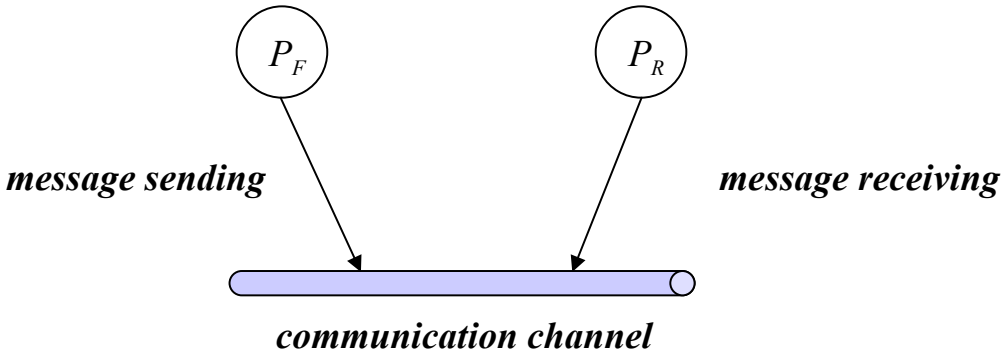
Message Passing

António Rui Borges

Summary

- *General principle*
- *Synchronization*
- *Types of addressing*
- *Type of communication*
- *Message format*
- *Problem of the producers / consumers*

General principle



Process communication through *message exchange* is a quite general way of communicating. It does not require sharing of the addressing space and its use, from the application programmer view point, is almost the same in single processor, multiprocessor and distributed processing environments.

It is based on very simple rules:

- whenever a process P_F, called the *forwarder*, wants to communicate with a process P_R, called the *recipient*, it sends to it a message through a communication channel that is established between them (*sending* operation)
- process P_R, in order to receive the message, needs to access the communication channel and wait for its arrival (*receiving operation*).

Synchronization - 1

Message exchange, however, will only lead to reliable communication between the *forwarder* and the *recipient* processes if some form of synchronization be ensured between them.

The degree of synchronization may be of two basic types

- *non-blocking synchronization* – the intervening processes are themselves responsible for synchronization; the *sending operation* forwards the message and returns without any information about message reception; the *receiving operation*, on the other hand, always returns independently of a message being received

```
/* sending operation */
void msgSendNB (unsigned int destid, MESSAGE msg);
/* receiving operation */
void msgReceiveNB (unsigned int srcid, MESSAGE *msg,
                   bool *msg_arrival);
```

Synchronization - 2

- *blocking synchronization* – the operations of sending and receiving contain in themselves elements which allow synchronization; the *sending operation* forwards the message and blocks until the message is actually received; the *receiving operation*, on the other hand, only returns when the message is received

```
/* sending operation */
void msgSend (unsigned int destid, MESSAGE msg);
/* receiving operation */
void msgReceive (unsigned int srcid, MESSAGE *msg);
```

Synchronization - 3

blocking synchronization can be divided into two types

- *rendez-vous* – the intervening processes must first arrive to the exchange point, only then message transfer takes place; it does not require the intermediate storage of the message and it is typical of dedicated communication channels (*point-to-point connections*);
- *remote* – the *sending operation* forwards the message and blocks, waiting for confirmation that the message was actually received by the recipient; it may, or may not, imply intermediate storage of the message and it is typical of shared communication channels.

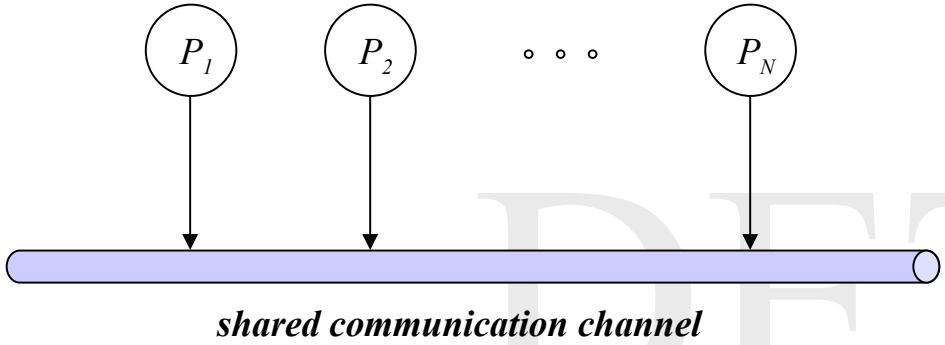
Types of addressing - 1

In order for the message to be exchanged between the forwarder and the recipient, it is fundamental that the *sending* and the *receiving* operations have a parameter that refers to the identity of the interlocutor process.

When the reference is explicit, the addressing is said to be *direct*. Otherwise, the communication channel that is being used, must be made explicit and the addressing is said to be *indirect*.

The communication channel may also allow the intermediate storage of the message, creating distributed data structures implementing queues where the chronological arrival order is in principle honored – the so called *mailboxes*.

Types of addressing - 2

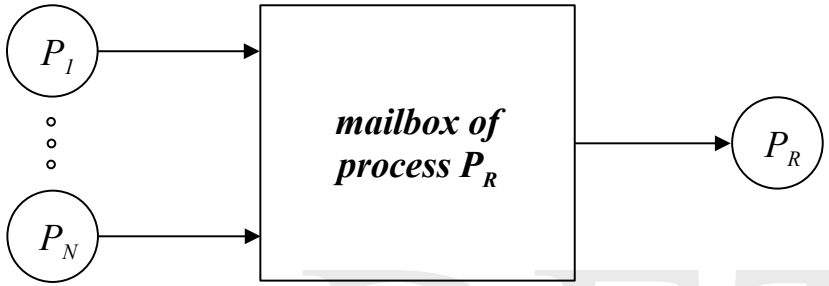


direct addressing – if based on process *id*

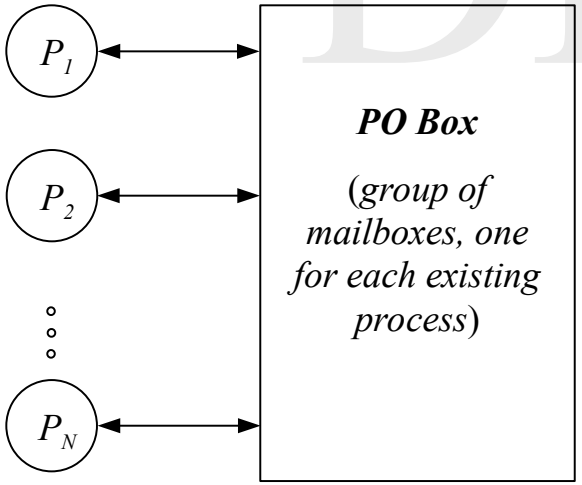
indirect addressing – if based on access *port*

indirect addressing

Types of addressing - 3



indirect addressing



indirect addressing

Type of communication - 1

Communication may occur in different contexts

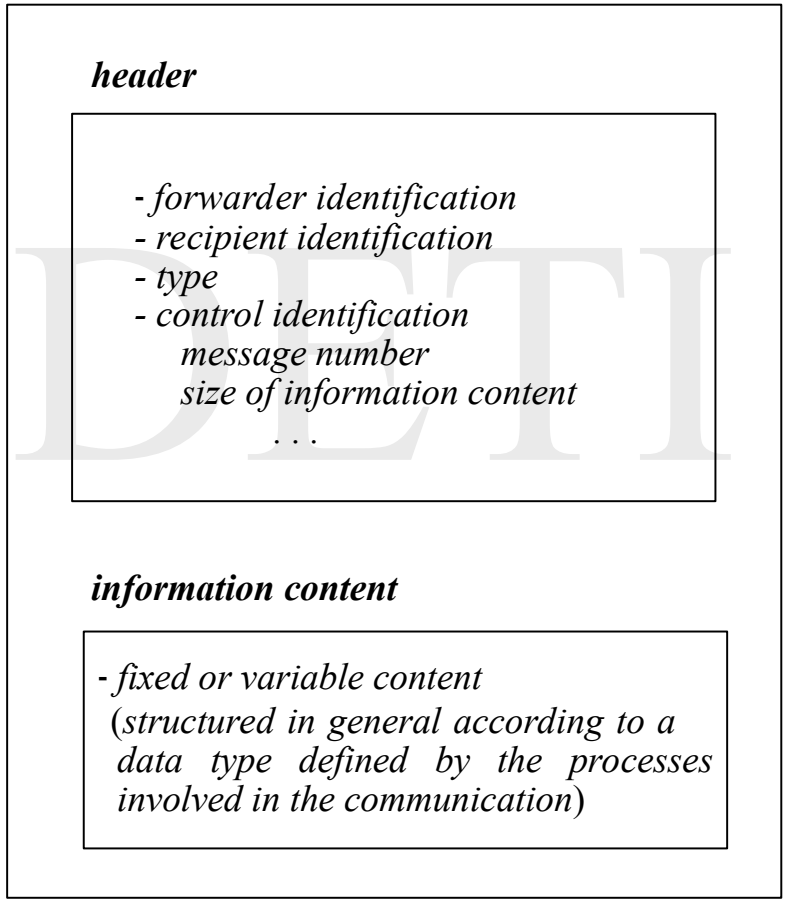
- *one-to-one communication* – the message is exchanged between two processes: a *forwarder* and a *recipient*
- *one-to-many communication* – the message is exchanged between a *forwarder* and many *recipients*; when the message is sent by one process to all the others in the application, is is called a *broadcast*; when the message is sent by one process to all the others belonging to the same group in the application, it is called a *multicast.*

Type of communication - 2

A composite message may be split into, or merged from, individual messages

- *scatter* – it is a one-to-many communication where the composite message is divided into mutual exclusive parts, each forwarded to a different *recipient*
- *gather* – it is a many-to-one communication where a composite message is formed by merging its parts, each received from a different *forwarder*.

Message format



Problem of the producers / consumers - 1

```
/* one assumes that a mailbox, with capacity for K messages, was created and
   can be accessed by all producer processes (as forwarders) and all consumer
   processes (as recipients)
*/
static unsigned int com;                              /* mailbox identification */
/* exchanged message */
typedef struct
        { DATA info;   /* there is solely a fixed size information content */
        } MESSAGE;
/* producer processes -  p = 0, 1, ..., N-1 */
void main (unsigned int p)
{
  MESSAGE msg;
  forever
  { produceValue (&msg.info);
    msgSendNB (com, msg);        /* the operation forwards the message and
                                 returns immediately, except if the mailbox
                                                    is full when it blocks */
    doSomeThingElse ();
  }
}
```

Problem of the producers / consumers - 2

```
/* one assumes that a mailbox, with capacity for K messages, was created and
   can be accessed by all producer processes (as forwarders) and all consumer
   processes (as recipients)
*/
static unsigned int com;                          /* mailbox identification */
/* exchanged message */
typedef struct
        { DATA info;   /* there is solely a fixed size information content */
        } MESSAGE;
/* consumer processes - c = N, N+1, ..., N+M-1 */
void main (unsigned int c)
{
  MESSAGE msg;
  forever
  { msgReceive (com, &msg);                    /* waits for message arrival */
    consumeValue (msg.info);
    doSomeThingElse ();
  }
}
```

Problem of the producers / consumers - 2

```
/* one assumes that a mailbox, with capacity for K messages, was created and
   can be accessed by all producer processes (as forwarders) and all consumer
   processes (as recipients)
*/
static unsigned int com;                          /* mailbox identification */
/* exchanged message */
typedef struct
        { DATA info;   /* there is solely a fixed size information content */
        } MESSAGE;
/* consumer processes - c = N, N+1, ..., N+M-1 */
void main (unsigned int c)
{
  MESSAGE msg;
  forever
  { msgReceive (com, &msg);                    /* waits for message arrival */
    consumeValue (msg.info);
    doSomeThingElse ();
  }
}
```