Inter Process Communication
Message Passing

Cooperating processes interact with one another, and a concurrent execution environment provides synchronization and communication facilities. Processes need to be synchronized to enforce mutual exclusion; cooperating processes may need to exchange information. One approach to providing both of these functions is message passing. Message-passing systems come in many forms. Here we describe a general introduction and discuss features typically found in such systems. The actual function of message passing is normally provided in the form of a pair of primitives and is the minimum set of operations needed for processes to engage in message passing:

```
send (destination, message)
receive (source, message)
```

A link is established automatically between every pair of processes that want to communicate. The processes need to know only each other's identity to communicate. A process sends information in the form of a message to another process designated by a destination. A process receives information by executing the receive primitive, indicating the source and the message.

Message Passing and Synchronization

The communication of a message between two processes implies some level of synchronization between the two: The receiver cannot receive a message until it has been sent by another process. In addition, we need to specify what happens to a process after it issues a send or receive primitive. When a send primitive is executed in a process, there are two possibilities: Either the sending process is blocked until the message is received, or it is not blocked. Similarly, when a process issues a receive primitive, there are two possibilities, in the first case if a message has previously been sent, the message is received and execution continues. In the second case if there is no waiting message, then either (a) the process is blocked until a message arrives, or (b) the process continues to execute, abandoning the attempt to receive.

Thus, both the sender and receiver can be blocking or nonblocking. Three combinations are common, although any particular system will usually have only one or two combinations implemented:

- *Blocking send, blocking receive*: Both the sender and receiver are blocked until the message is delivered; this is sometimes referred to as a rendezvous. This combination allows for tight synchronization between processes.
- *Nonblocking send, blocking receive*: Although the sender may continue on, the receiver is blocked until the requested message arrives. This is probably the most useful combination. It allows a process to send one or more messages to a variety of destinations as quickly as possible. A process that must receive a message before it can do useful work needs to be blocked until such a message arrives. An example is a server process that exists to provide a service or resource to other processes.

- *Nonblocking send, nonblocking receive*: Neither party is required to wait.

The nonblocking send is more natural for many concurrent programming tasks. For example, if it is used to request an output operation, such as printing, it allows the requesting process to issue the request in the form of a message and then carry on. For the receive primitive, the blocking version appears to be more natural for many concurrent programming tasks. Generally, a process that requests a message will need the expected information before proceeding.

Naming (Addressing) in Message Passing

Processes that want to communicate must have a way to refer to each other. It is necessary to specify in the send primitive which process is to receive the message. Similarly, most implementations allow a receiving process to indicate the source of a message to be received. Specifying processes in send and receive primitives fall into two categories: direct addressing and indirect addressing.

With direct addressing, each process that wants to communicate must explicitly name the recipient or sender of the communication. In this scheme, the send() and receive() primitives are defined as:

- send(X, message); Send a message to process X
- receive(Y, message); Receive a message from process Y

Here the send and receive primitives includes a specific identifier of the destination and source process, respectively and show a symmetry in addressing/naming. In this case a process explicitly designates a sending process. Thus, the process must know ahead of time from which process a message is expected, which is effective for cooperating concurrent processes. In situations, it is impossible to specify the anticipated source process. An example is a printer server process, which will accept a print request message from any other process. For such applications, asymmetric approach of addressing is employed and only the sender names the recipient; the recipient is not required to name the sender. In this scheme, the send() and receive() primitives are defined as follows:

- send(X, message); Send a message to process X.
- receive(id, message); Receive a message from any process.

In this case, the source parameter of the receive primitive possesses a value returned is the name of the process with which communication has taken place.

In indirect addressing, messages are not sent directly from sender to receiver but rather are sent to a shared data structure consisting of queues that can temporarily hold messages. Such queues are generally referred to as *mailboxes*. Thus, for two processes to communicate, one process sends a message to the appropriate mailbox and the other process picks up the message from the mailbox. By decoupling the sender and receiver, it

allows greater flexibility in the use of messages. A process can communicate with another process via a number of different mailboxes, but two processes can communicate only if they have a shared mailbox. The send() and receive() primitives are defined as follows:

- send(M, message); Send a message to mailbox M.
- receive(M, message); Receive a message from mailbox M.

The relationship between senders and receivers can be one to one, many to one, one to many, or many to many. A one-to-one relationship allows a private communications link to be set up between two processes. This insulates their interaction from erroneous interference from other processes. A many-to-one relationship is useful for client/server interaction; one process provides service to a number of other processes. In this case, the mailbox is often referred to as a port. A one-to-many relationship allows for one sender and multiple receivers; it is useful for applications where a message or some information is to be broadcast to a set of processes. A many-to-many relationship allows multiple server processes to provide concurrent service to multiple clients.
The association of processes to mailboxes can be either static or dynamic. Ports are often statically associated with a particular process; that is, the port is created and assigned to the process permanently. Similarly, a one-to-one relationship is typically defined statically and permanently. When there are many senders, the association of a sender to a mailbox may occur dynamically. Primitives such as connect and disconnect may be used for this purpose.
Mailbox has owner and users. In the case of a port, it is typically owned by and created by the receiving process. Thus, when the process is destroyed, the port is also destroyed. For the general mailbox case, the OS may offer a create-mailbox service. Such mailboxes can be viewed either as being owned by the creating process, in which case they terminate with the process, or as being owned by the OS, in which case an explicit command will be required to destroy the mailbox.

Message Format

The format of the message depends on the objectives of the messaging facility and whether the facility runs on a single computer or on a distributed system. For some operating systems, designers have preferred short, fixed-length messages to minimize processing and storage overhead. If a large amount of data is to be passed, the data can be placed in a file and the message then simply references that file. A more flexible approach is to allow variable-length messages. The message is divided into two parts: a header, which contains information about the message, and a body, which contains the actual contents of the message. The header may contain an identification of the source and intended destination of the message, a length field, and a type field to discriminate among various types of messages. There may also be additional control information, such as a pointer field so that a linked list of messages can be created; a sequence number, to keep track of the number and order of messages passed between source and destination and a priority field.

Message Passing for Mutual Exclusion

Message passing can be used to enforce mutual exclusion among cooperating processes accessing shared resources, due to blocking and nonblocking characteristics of receive and send primitives, respectively. We can use indirect addressing (mailbox) to solve critical section problem for n processes. A set of concurrent processes share a mailbox, which can be used by all processes to send and receive. The code for mutual exclusion for n processes is listed below:

```
/* program mutual_exclusion */

const int n =              /* number of process */
void main()
  create mailbox (box);
  send (box, null);
void P(int i) {
   message msg;
   while (true) {
       receive (box, msg);
       ; /* critical section */
       send (box, msg);
      /* remainder */;
  }
}
```

The mailbox is initialized to contain a single message with null content. A process wishing to enter its critical section first attempts to receive a message. If the mailbox is empty, then the process is blocked. Once a process has acquired the message, it performs its critical section and then places the message back into the mailbox. Thus, the message functions as a token that is passed from process to process. The solution assumes that if more than one process performs the receive operation concurrently, then, if there is a message, it is delivered to only one process and the others are blocked, or if the message queue is empty, all processes are blocked; when a message is available, only one blocked process is activated and given the message.

Message Passing for Synchronization

Since receive() primitive has blocking characteristics, send() and receive() primitives can easily be used for synchronization problems. We describe producer-consumer problem with bounded buffer using indirect message passing primitives. In this case, send and receive() primitives are used to pass data, and signals. Two mailboxes *mayconsume* and *mayproduce* are used. Initially, the mailbox mayproduce is filled with a number of null messages equal to the capacity of the buffer. Initialization of shared variables and mailboxes is listed bellow:

```
const int
capacity = /* buffering capacity */ ;
null =    /* empty message */ ;
int i;
void main() {
        create_mailbox (mayproduce);
        create_mailbox (mayconsume);
        for (int i = 1; i<= capacity; i++)
                send (mayproduce, null);
}
```

As the producer generates data, it is sent as a message to the mailbox mayconsume. The code of producer process is listed below:

```
void producer() {
  message pmsg;
  while (true) {
     receive (mayproduce, pmsg);
     pmsg = produce();
     send (mayconsume, pmsg);
  }
}
```

As long as there is at least one message in that mailbox, the consumer can consume. The code of consumer process is listed bellow:

```
void consumer() {
   message cmsg;
   while (true) {
     receive (mayconsume,cmsg);
     consume (cmsg);
     send (mayproduce,null);
   }
}
```

Here mayconsume serves as the buffer; the data in the buffer are organized as a queue of messages. The number of messages in mayproduce shrinks with each production and grows with each consumption.

Indirect message passing approach is quite flexible and can be used for classical synchronization problems. Here we describe reader-writer problem with reader's priority. Solution of reader-writer problem with reader's priority using semaphores is described in semaphores-Reader-Writer lecture notes. Here we will use the same solution and relate the similarities between operations on semaphores and send and receive primitives of indirect message passing. There are similarities between receive(msg) primitive and wait(x) operation of semaphore, both block the process on a condition. Likewise, send(msg) and signal(x) operation of semaphore are similar.

We create two mailboxes rwsyn and mutex and initialized them by sending a null msg to both of these mailboxes and a variable read_count is initialized to 0.

```
const int
null =    /* empty message */ ;
int read-count = 0;
create_mailbox (mutex); send (mutex, null);
create_mailbox (rwsyn); send (rwsyn, null);
```

The code for the reader is listed below:

```
Reader()
message msg, wmsg;
Receive (mutex, msg);
read_count++;
if (read_count == 1)
       receive (rwsyn, wmsg);
send(mutex, msg);
<Reader Unit>;
Receive (mutex, msg);
read_count--;
if (read_count==0)
     send((rwsyn, wmsg);
send(mutex, msg);
```

The code for the writer process is listed below:

```
Writer()

message wmsg;
receive (rwsyn, wmsg);
   <Write Unit>;
send (rwsyn, wmsg);
```

The mutex mailbox (initialized to one null msg) is used to exclusively update variable read_count. The read_count variable keeps track of how many processes are currently reading the data. The mailbox rwsyn is common to both reader and writer processes. The mailbox rwsyn is to ensure exclusive writing function by the writers. It is also used by the first or last reader that enters or exits the critical section (reading data). When first reader comes, it will receive msg from mutex mailbox (thus emptying the mailbox), increment the read_count (set it '1') and will receive wmsg from rwsyn mailbox (thus emptying the mailbox) and after that send msg to mutex mailbox (now mutex mailbox has again one null msg) and start reading. Now another reader comes, it will first receive msg from from mutex mailbox (thus emptying the mailbox), increment the read_count (set it '2'). Now the values of read_count is '2', so it will invoke receive, rather it will send a msg to mutex mailbox and start reading. So far multiple readers can read. Now we assume a writer comes, writer will invoke receive on rwsyn mailbox, and will be blocked, since

rwsyn mailbox is empty. When the last reader will leave, it sends a wmsg to rwsyn mailbox and with the availability of a message in the mailbox, writer process will be unblocked and receive wmsg and start writing. With completion of the receive operation on rwsyn mailbox, the mailbox is again empty, so another writer will not be able to perform write operation and is blocked on rwsyn mailbox. You can explore different sequences of reader and writer processes to verify that solution of reader-writer problem with reader's priority using receive and send primitives works.

Carefully examine solution of reader-writer problem with reader's priority using semaphores and this solution. You can easily identify how wait and signal operations in the solution of classical problems using semaphores can be replaced by send and receive primitives of indirect message passing.