Send and Receive Operations

send (void *sendbuf, int nelems, int dest)

receive (void *recvbuf, int nelems, int source)

1. sendbuf points to a buffer that stores the data to be sent,
2. recvbuf points to a buffer that stores the data to be received,
3. nelems is the number of data units to be sent and received,
4. dest is the identifier of the process that receives the data,
5. source is the identifier of the process that sends the data.

```
1       P0                      P1
2
3       a = 100;                  receive(&a, 1, 0)
4       send(&a, 1, 1);             printf("%d\n", a);
5       a=0;
```

- In this simple example, process P0 sends a message to process P1 which receives and prints the message.
- The important thing to note is that process P0 changes the value of a to 0 immediately following the send.
- The semantics of the send operation require that the value received by process P1 must be 100 as opposed to 0.
- That is, the value of a at the time of the send operation must be the value that is received by process P1.

Most message passing platforms have additional hardware support for sending and receiving messages. They may support **DMA (direct memory access) and asynchronous message transfer (independent of the CPU) using network interface hardware**.

Network interfaces allow the transfer of messages from buffer memory to desired location without CPU intervention.

Similarly, DMA allows copying of data from one memory location to another (e.g., communication buffers) without CPU support (once they have been programmed).

As a result, if the send operation programs the communication hardware and returns before the communication operation has been accomplished, process P1 might receive the value 0 in a instead of 100!

Blocking Message Passing Operations

A simple solution to the dilemma presented in the code fragment above is for the send operation to return only when it is semantically safe to do so.

This is not the same as saying that the send operation returns only after the receiver has received the data.

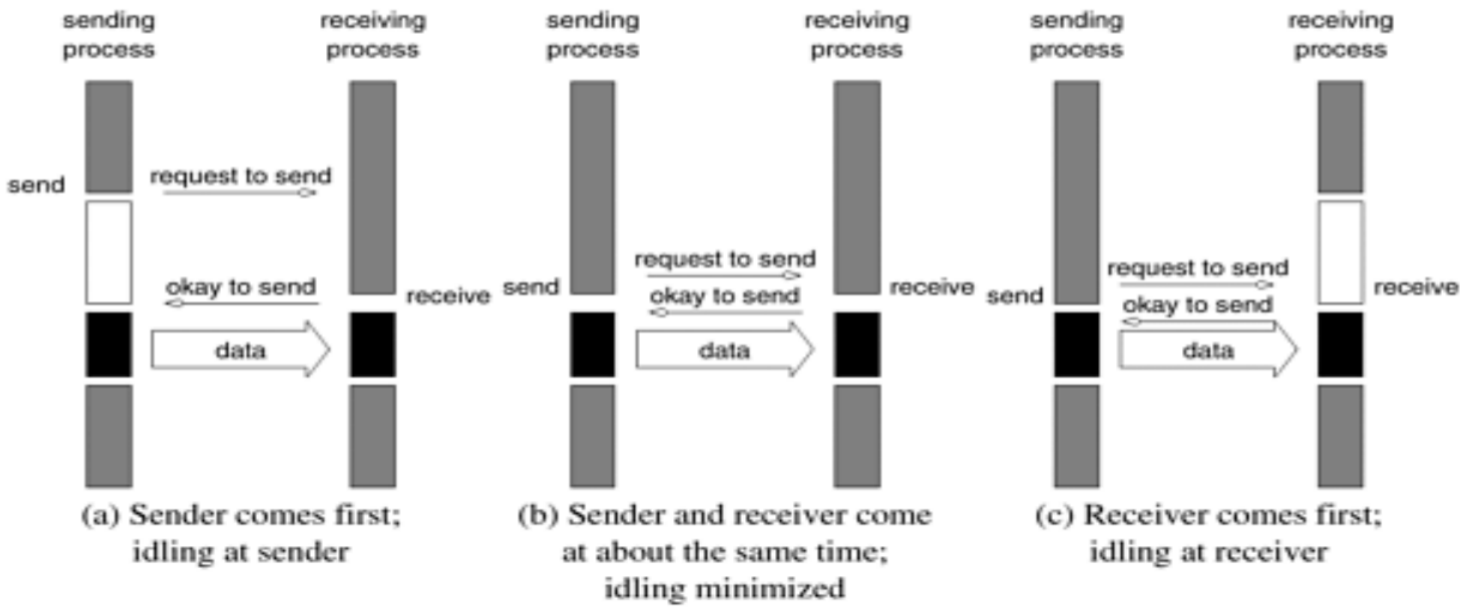
It simply means that the sending operation blocks until it can guarantee that the semantics will not be violated on return irrespective of what happens in the program subsequently.

There are two mechanisms by which this can be achieved:

1. Blocking Non-Buffered Send/Receive

- In the first case, the send operation does not return until the matching receive has been encountered at the receiving process.
- When this happens, the message is sent and the send operation returns upon completion of the communication operation.
- Typically, this process involves a handshake between the sending and receiving processes. The sending process sends a request to communicate to the receiving process.
- When the receiving process encounters the target receive, it responds to the request.
- The sending process upon receiving this response initiates a transfer operation.
- Since there are no buffers used at either sending or receiving ends, this is also referred to as a non-buffered blocking operation.

Figure 6.1. Handshake for a blocking non-buffered send/receive operation. It is easy to see that in cases where sender and receiver do not reach communication point at similar times, there can be considerable idling overheads.



Idling Overheads in Blocking Non-Buffered Operations

In Figure 6.1, we illustrate three scenarios in which the send is reached before the receive is posted, the send and receive are posted around the same time, and the receive is posted before the send is reached.

In cases (a) and (c), we notice that there is considerable idling at the sending and receiving process. It is also clear from the figures that a blocking non-buffered protocol is suitable when the send and receive are posted at roughly the same time.

However, in an asynchronous environment, this may be impossible to predict.

This idling overhead is one of the major drawbacks of this protocol

Deadlocks in Blocking Non-Buffered Operations

Consider the following simple exchange of messages that can lead to a deadlock:

```
1       P0                      P1
2
3       send(&a, 1, 1);             send(&a, 1, 0);
4       receive(&b, 1, 1);          receive(&b, 1, 0);
```

The code fragment makes the values of a available to both processes P0 and P1.

However, if the send and receive operations are implemented using a blocking non-buffered protocol, the send at P0 waits for the matching receive at P1 whereas the send at process P1 waits for the corresponding receive at P0, resulting in an infinite wait.

As can be inferred, deadlocks are very easy in blocking protocols and care must be taken to break cyclic waits of the nature outlined.

In the above example, this can be corrected by replacing the operation sequence of one of the processes by a receive and a send as opposed to the other way around.

Blocking Buffered Send/Receive

A simple solution to the idling and deadlocking problem outlined above is to rely on buffers at the sending and receiving ends.

We start with a simple case in which the sender has a buffer pre-allocated for communicating messages.

On encountering a send operation, the sender simply copies the data into the designated buffer and returns after the copy operation has been completed.

The sender process can now continue with the program knowing that any changes to the data will not impact program semantics.

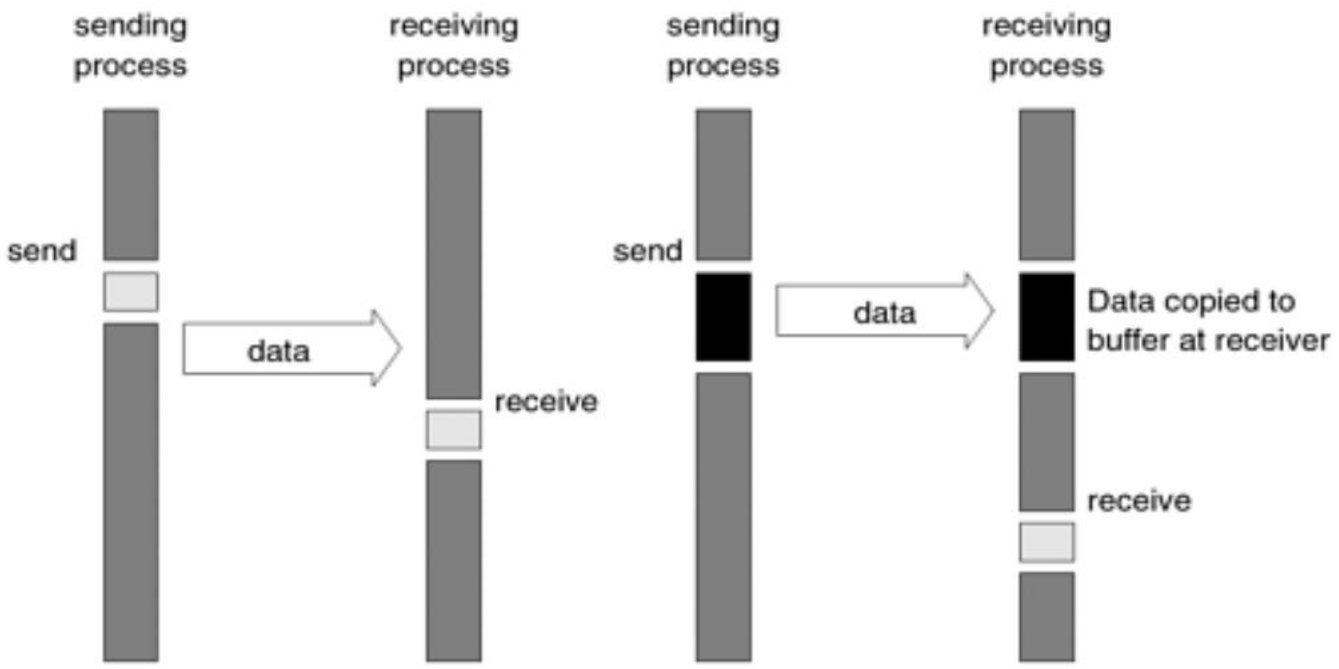
The actual communication can be accomplished in many ways depending on the available hardware resources.

If the hardware supports asynchronous communication (independent of the CPU), then a network transfer can be initiated after the message has been copied into the buffer.

Note that at the receiving end, the data cannot be stored directly at the target location since this would violate program semantics. Instead, the data is copied into a buffer at the receiver as well.

When the receiving process encounters a receive operation, it checks to see if the message is available in its receive buffer. If so, the data is copied into the target location.

Figure 6.2. Blocking buffered transfer protocols: (a) in the presence of communication hardware with buffers at send and receive ends; and (b) in the absence of communication hardware, sender interrupts receiver and deposits data in buffer at receiver end.



In the protocol illustrated above, buffers are used at both sender and receiver and communication is handled by dedicated hardware.

Sometimes machines do not have such communication hardware.

In this case, some of the overhead can be saved by buffering only on one side.

For example, on encountering a send operation, the sender interrupts the receiver, both processes participate in a communication operation and the message is deposited in a buffer at the receiver end.

When the receiver eventually encounters a receive operation, the message is copied from the buffer into the target location.

This protocol is illustrated in Figure 6.2(b).

It is not difficult to conceive a protocol in which the buffering is done only at the sender and the receiver initiates a transfer by interrupting the sender.

It is easy to see that buffered protocols alleviate idling overheads at the cost of adding buffer management overheads.

In general, if the parallel program is highly synchronous (i.e., sends and receives are posted around the same time), non-buffered sends may perform better than buffered sends. However, in general applications, this is not the case and buffered sends are desirable unless buffer capacity becomes an issue.

Impact of finite buffers in message passing

```
2                  P0                          P1
3       for (i = 0; i < 1000; i++) {    for (i = 0; i < 1000; i++) {
4         produce_data(&a);                 receive(&a, 1, 0);
5         send(&a, 1, 1);                 consume_data(&a);
6       }                               }
```

In this code fragment, process P0 produces 1000 data items and process P1 consumes them.

However, if process P1 was slow getting to this loop, process P0 might have sent all of its data.

If there is enough buffer space, then both processes can proceed; however, if the buffer is not sufficient (i.e., buffer overflow), the sender would have to be blocked until some of the corresponding receive operations had been posted, thus freeing up buffer space.

This can often lead to unforeseen overheads and performance degradation. In general, it is a good idea to write programs that have bounded buffer requirements.

Deadlocks in Buffered Send and Receive Operations

```
1       P0                      P1
2
3       receive(&a, 1, 1);          receive(&a, 1, 0);
4       send(&b, 1, 1);             send(&b, 1, 0);
```

While buffering alleviates many of the deadlock situations, it is still possible to write code that deadlocks.

This is due to the fact that as in the non-buffered case, receive calls are always blocking (to ensure semantic consistency).

Thus, a simple code fragment such as the following deadlocks since both processes wait to receive data but nobody sends it

Non-Blocking Message Passing Operations

In blocking protocols, the overhead of guaranteeing semantic correctness was paid in the form of idling (non-buffered) or buffer management (buffered).

Often, it is possible to require the programmer to ensure semantic correctness and provide a fast send/receive operation that incurs little overhead.

This class of non-blocking protocols returns from the send or receive operation before it is semantically safe to do so.

Consequently, the user must be careful not to alter data that may be potentially participating in a communication operation.

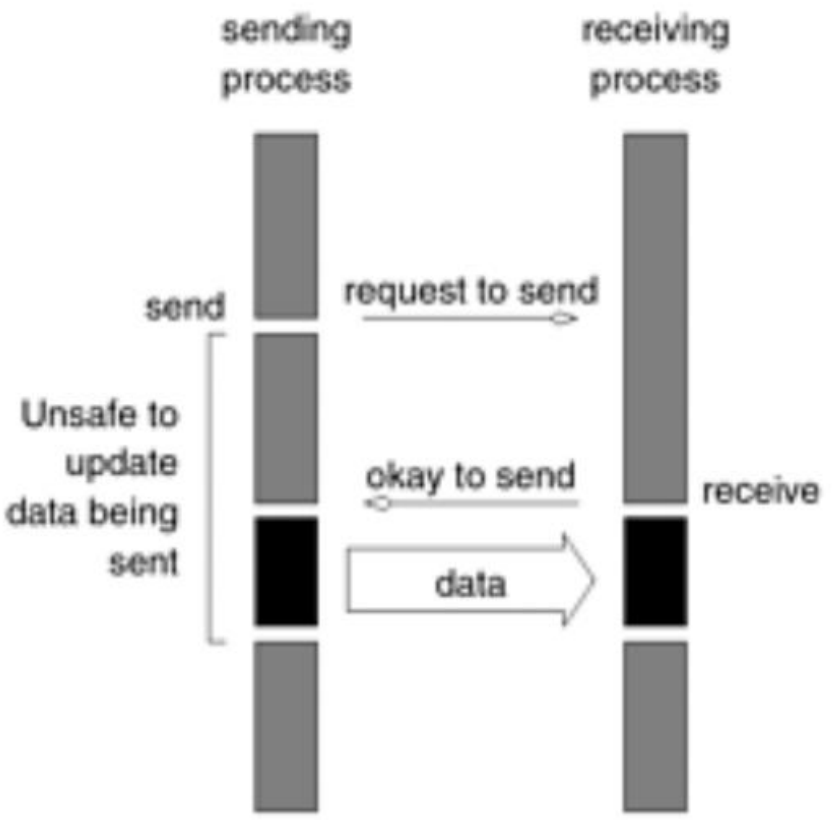
Non-blocking operations are generally accompanied by a check-status operation, which indicates whether the semantics of a previously initiated transfer may be violated or not.

Upon return from a non-blocking send or receive operation, the process is free to perform any computation that does not depend upon the completion of the operation.

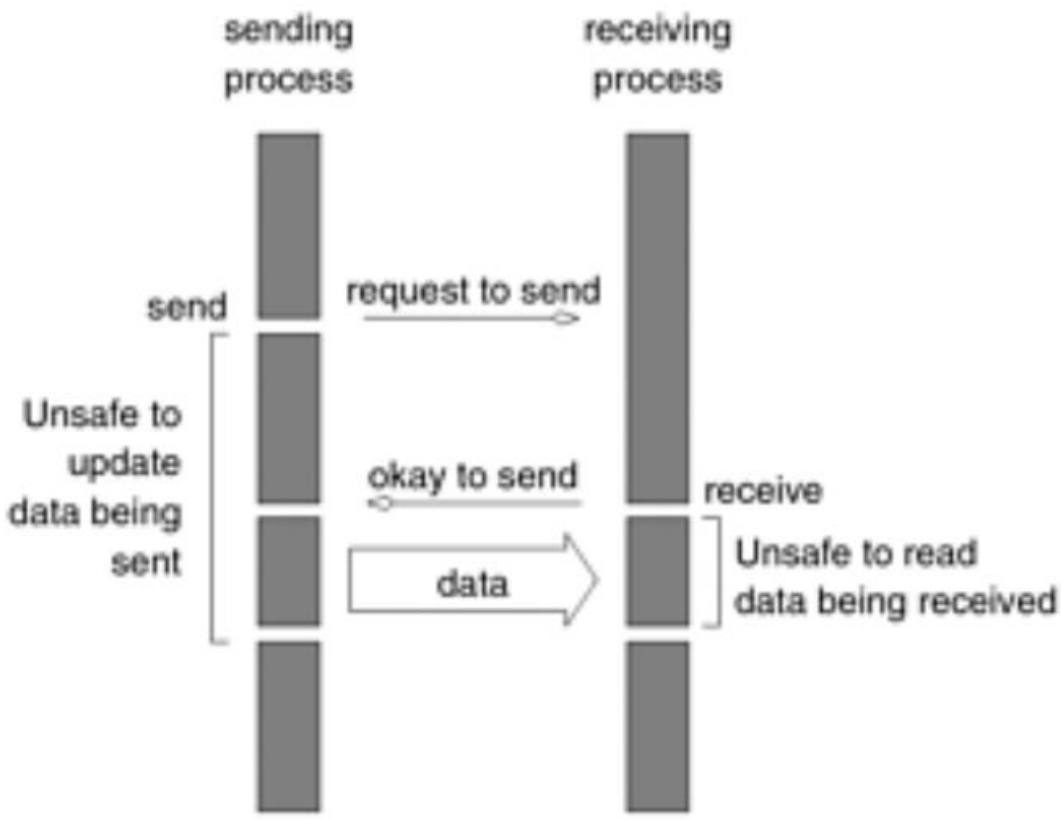
Later in the program, the process can check whether or not the non-blocking operation has completed, and, if necessary, wait for its completion.

Non-blocking operations can themselves be buffered or non-buffered.

In the non-buffered case, a process wishing to send data to another simply posts a pending message and returns to the user program. The program can then do other useful work. At some point in the future, when the corresponding receive is posted, the communication operation is initiated. When this operation is completed, the check-status operation indicates that it is safe for the programmer to touch this data.



(a) Without hardware support



(b) With hardware support

Non-blocking operations can also be used with a buffered protocol.

In this case, the sender initiates a DMA operation and returns immediately.

The data becomes safe the moment the DMA operation has been completed.

At the receiving end, the receive operation initiates a transfer from the sender's buffer to the receiver's target location.

Using buffers with non-blocking operation has the effect of reducing the time during which the data is unsafe.

Typical message-passing libraries such as Message Passing Interface (MPI) and Parallel Virtual Machine (PVM) implement both blocking and non-blocking operations.

Blocking operations facilitate safe and easier programming and non-blocking operations are useful for performance optimization by masking communication overhead.

One must, however, be careful using non-blocking protocols since errors can result from unsafe access to data that is in the process of being communicated.

	Blocking Operations	Non−Blocking Operations
Buffered	Sending process returns after data has been copied into communication buffer	Sending process returns after initiating DMA transfer to buffer. This operation may not be completed on return
Non−Buffered	Sending process blocks until matching receive operation has been encountered	
	Send and Receive semantics assured by corresponding operation	Programmer must explicitly ensure semantics by polling to verify completion