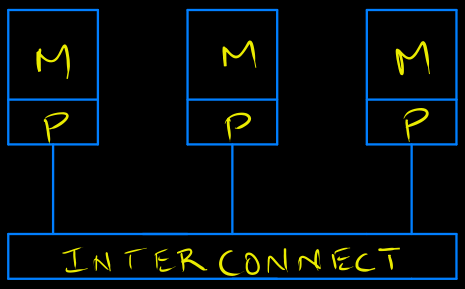
Message Passing Programming

- Most widely used approach
- Minimal requirements on the underlying hardware

Key characteristics

- Partitioned Address space
- Explicit Parallelization



⇒ Each data element must belong to one of the partitions
⇒ Needs explicit partitioning
⇒ Complexity in programming
⇒ But locality of access for high performance
⇒ All interactions need Cooperation of two processes
⇒ Process that has the data must participate even if it has no logical connection
⇒ But programmer is fully aware of non-local interactions and hence design to minimize
⇒ Can be efficiently implemented on a wide variety of architectures

Building Blocks : Send & Receive operations

Send (void *sendbuf, int n, int dest)

Receive (void *recvbuf, int n, int Src)

P_0

```
int a = 100 ;
Send ( &a, 1, 1);
a = 0 ;
```

P_1

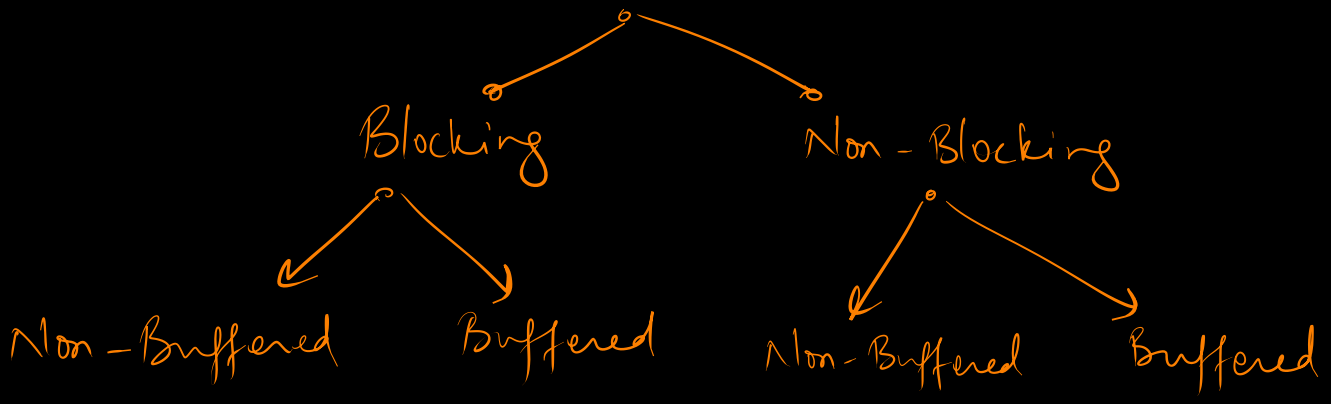
```
int a = 25;
Receive (&a, 1, 0);
printf ("%d", a);
```

⇒ Semantics seem obvious but implementations can differ

what if Send (..) returns after indicating &a to the network interface that can DMA?

what if Receive (..) returns after indicating &a to the network interface that can DMA?

⇒ Need to know different modes of operation



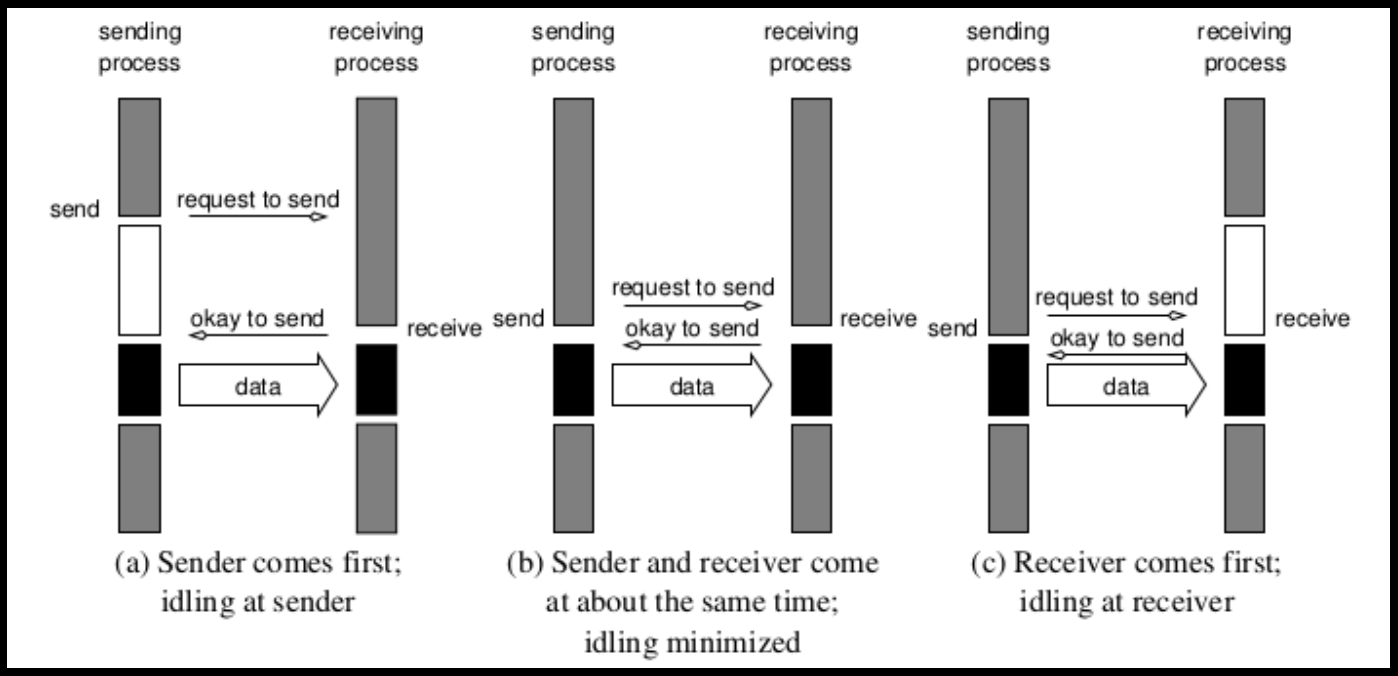
I. Blocking + Non-Buffered

→ Send does not return until matching receive is posted

→ Message sent & send returns on completion of the communication operation

→ Involves a handshake between sender & receiver

→ No buffers used at either end



(a) Sender comes first; idling at sender

(b) Sender and receiver come at about the same time; idling minimized

(c) Receiver comes first; idling at receiver

→ Process Idling

→ Best suited when send and receive are posted at roughly the same time

P_0

```
int a = 100, b = 200;
Send (&a, 1, 1);
receive (&b, 1, 1);
printf ("%d %d", a, b);
```

P_1

```
int a = 200, b = 100;
Send (&b, 1, 0);
receive (&a, 1, 0);
printf ("%d %d", a, b);
```

II. Blocking Buffered

→ Assume pre-allocated buffers at sender and/or receiver

→ send(...) copies the data into the designated buffer and returns

→ Sender can continue – data change cannot influence semantics of communication

→ Asynchronous communication capability of hardware

→ At receiver, data is copied into a buffer

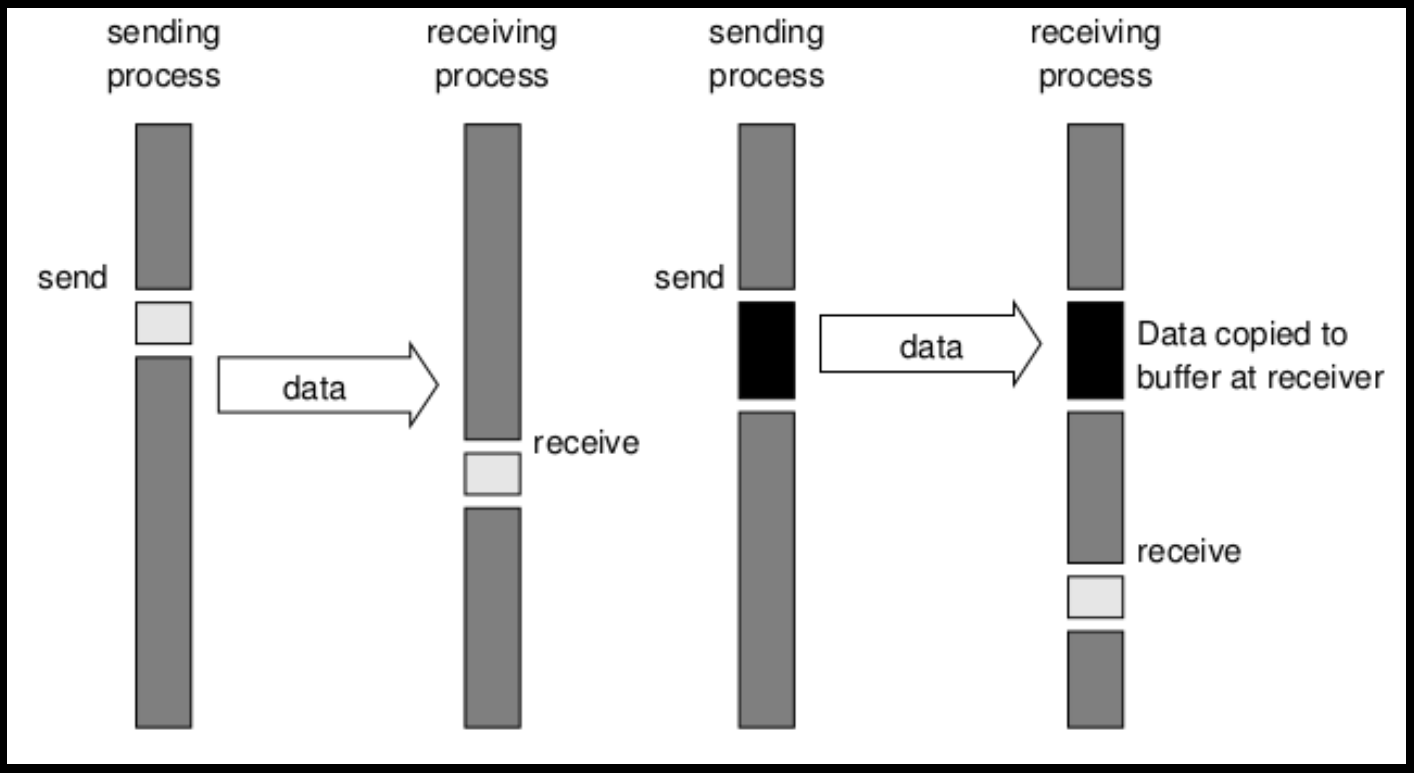
→ When the destination process calls receive(...), buffer is checked and if available, copies message to its buffer

→ Almost NO Idling

→ If such hardware is not available on both sides, some overhead can be avoided by using only one side buffers.

→ On encountering a send(...), the sender interrupts the receiver, both interact and message is deposited in a buffer at the receiver

→ When receiver calls receive(...), message is copied from the buffer into the target location



P_0

```
int a = 100, b = 200;
Send (&a, 1, 1);
receive (&b, 1, 1);
printf ("%d %d", a, b);
```

P_1

```
int a = 200, b = 100;
Send (&b, 1, 0);
receive (&a, 1, 0);
printf ("%d %d", a, b);
```

P_0

```
int a = 100, b = 200;
Receive (&a, 1, 1);
Send (&b, 1, 1);
printf ("%d %d", a, b);
```

P_1

```
int a = 200, b = 100;
Receive (&b, 1, 0);
Send (&a, 1, 0);
printf ("%d %d", a, b);
```

P_0

```
for( i=0; i<N; i++)
{
    produce_data (&a);
    Send (&a, 1, 1);
}
```

P_1

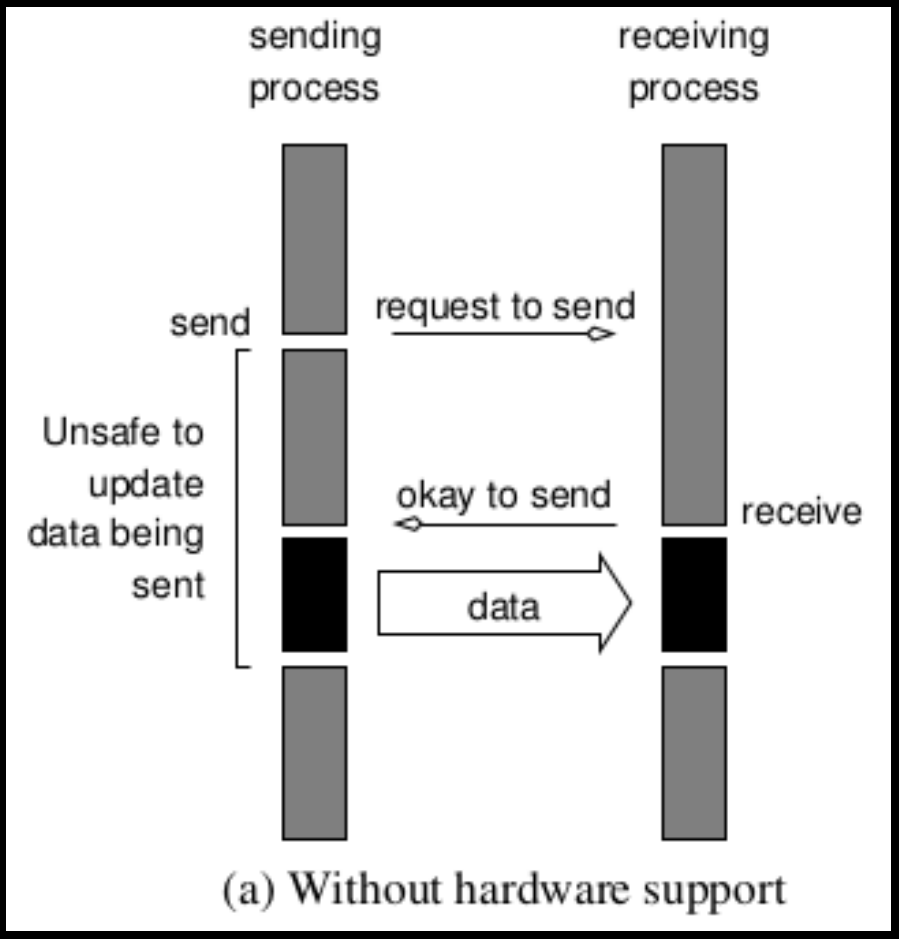
```
for( i=0; i<N; i++)
{
    receive( &a, 1, 0);
    Consume_data (&a);
}
```

What if P_1 reaches the loop much later?

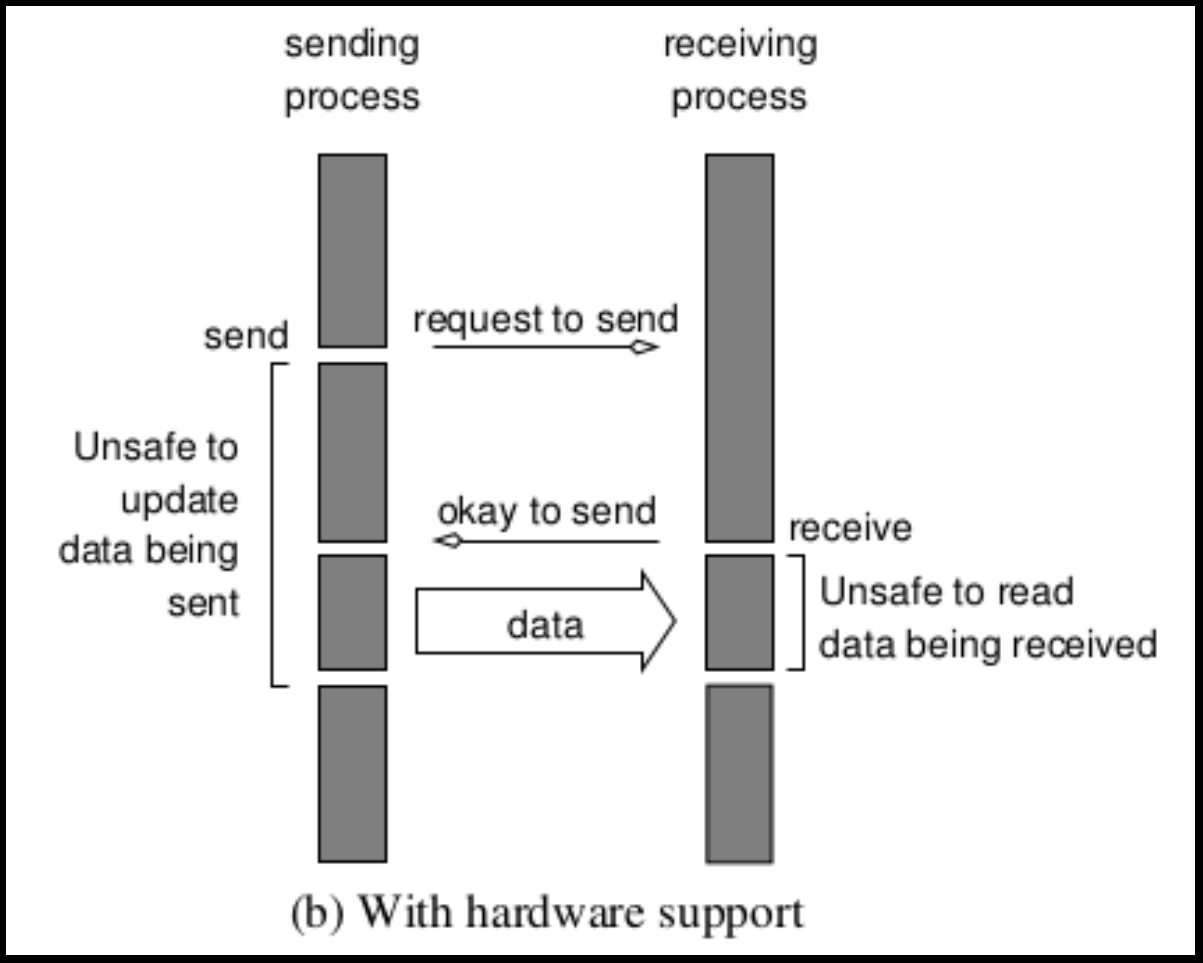
[In Blocking mode, Semantic Correctness is guaranteed, but paid in the form of idling or buffer management.]

III. Non-Blocking + Non-Buffered

→ Programmer to ensure Semantic Correctness
→ Provides fast send/receive with little overhead
→ Returns from send or receive before it is semantically safe to do so
→ Programmer to check status of completion
→ But program is free to continue after a quick send or receive



(a) Without hardware support



(b) With hardware support

IV. Non-Blocking + Buffered

→ Sender initiates DMA and returns

→ Data unsafe till DMA operation is complete

→ Receiver initiates transfer from sender's buffer to the receiver's target location

→ Reduces the time during which the data is unsafe

MPI : Message Passing Interface

- Standard library for Message passing
- Portable message passing in C or Fortran
- Defines syntax & semantics of the routines
- Almost all vendors provide their implementation
- > 125 functions
- Used in academia & industry
- Widely popular