Message Passing

Loïc Guégan

Adapted from P. Ha @ UiT, K. Li @ Princeton,
A. S. Tanenbaum @ 2008, A. Silberschatz @ 2009 and B. Wilkinson @ 2004

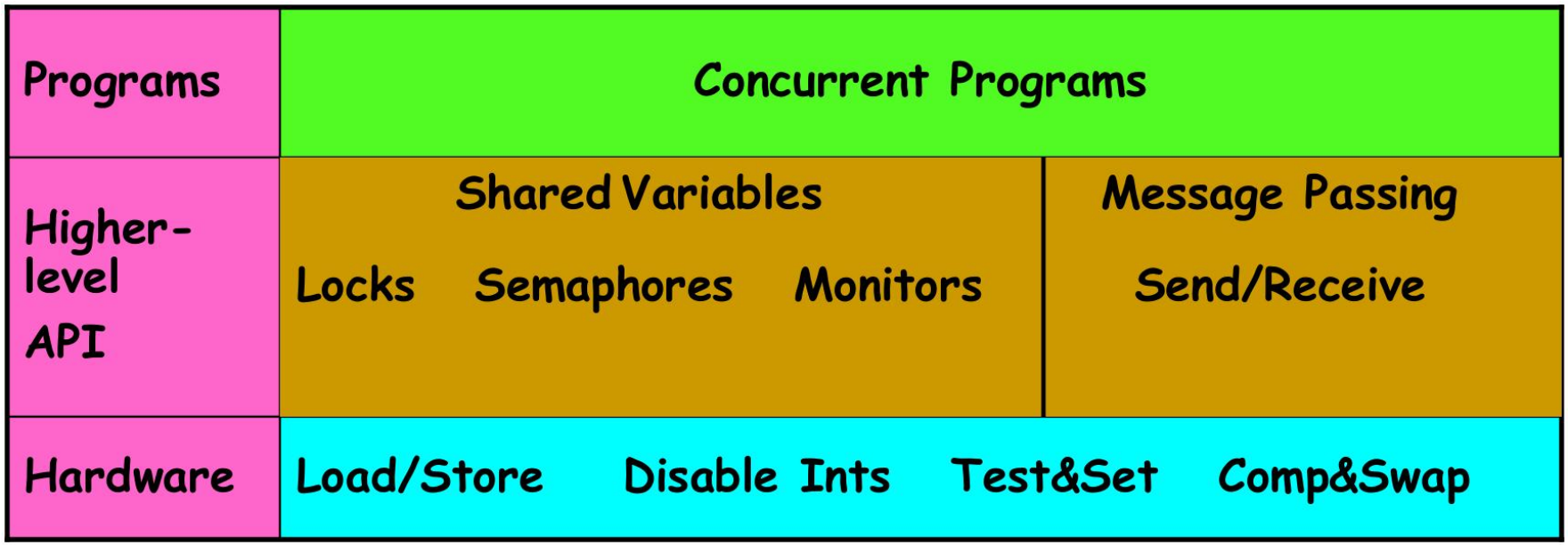
UiT The Arctic University of Norway

Spring - 2024

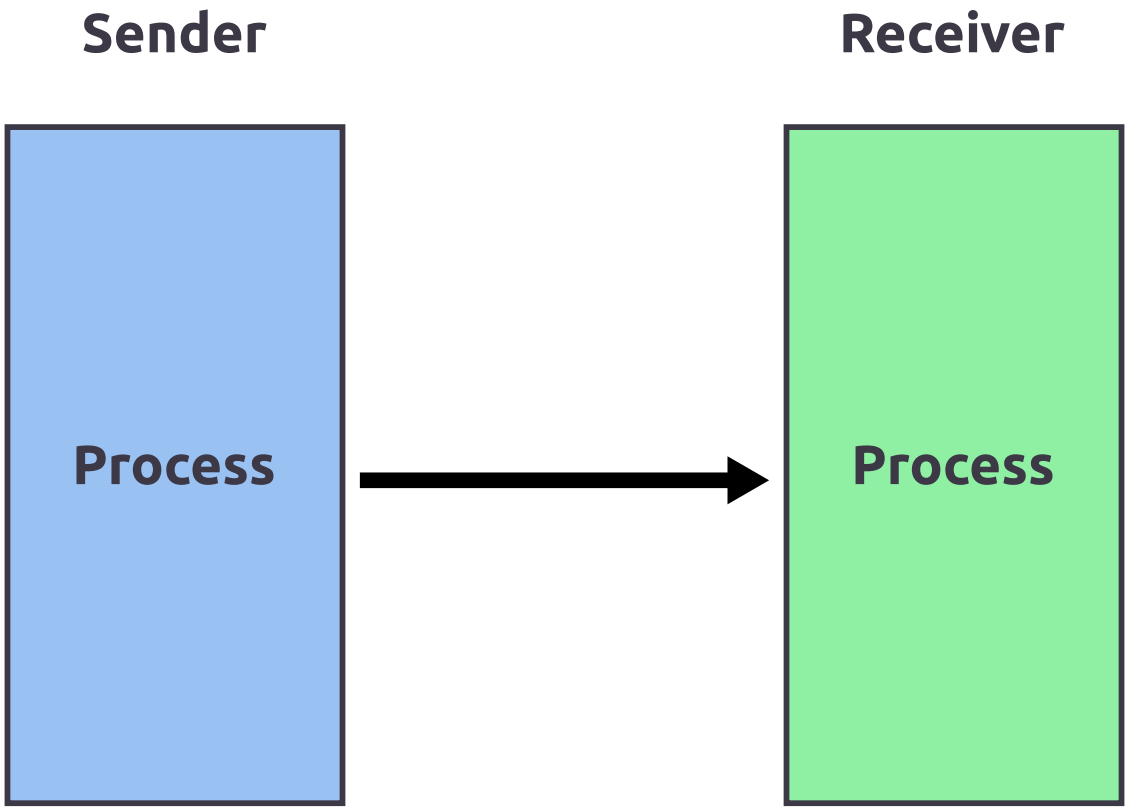
Outline

- What is message passing ?
  - Semantics
  - How to use
- How to implement message passing?
  - Implementation issues
- Examples of message passing systems

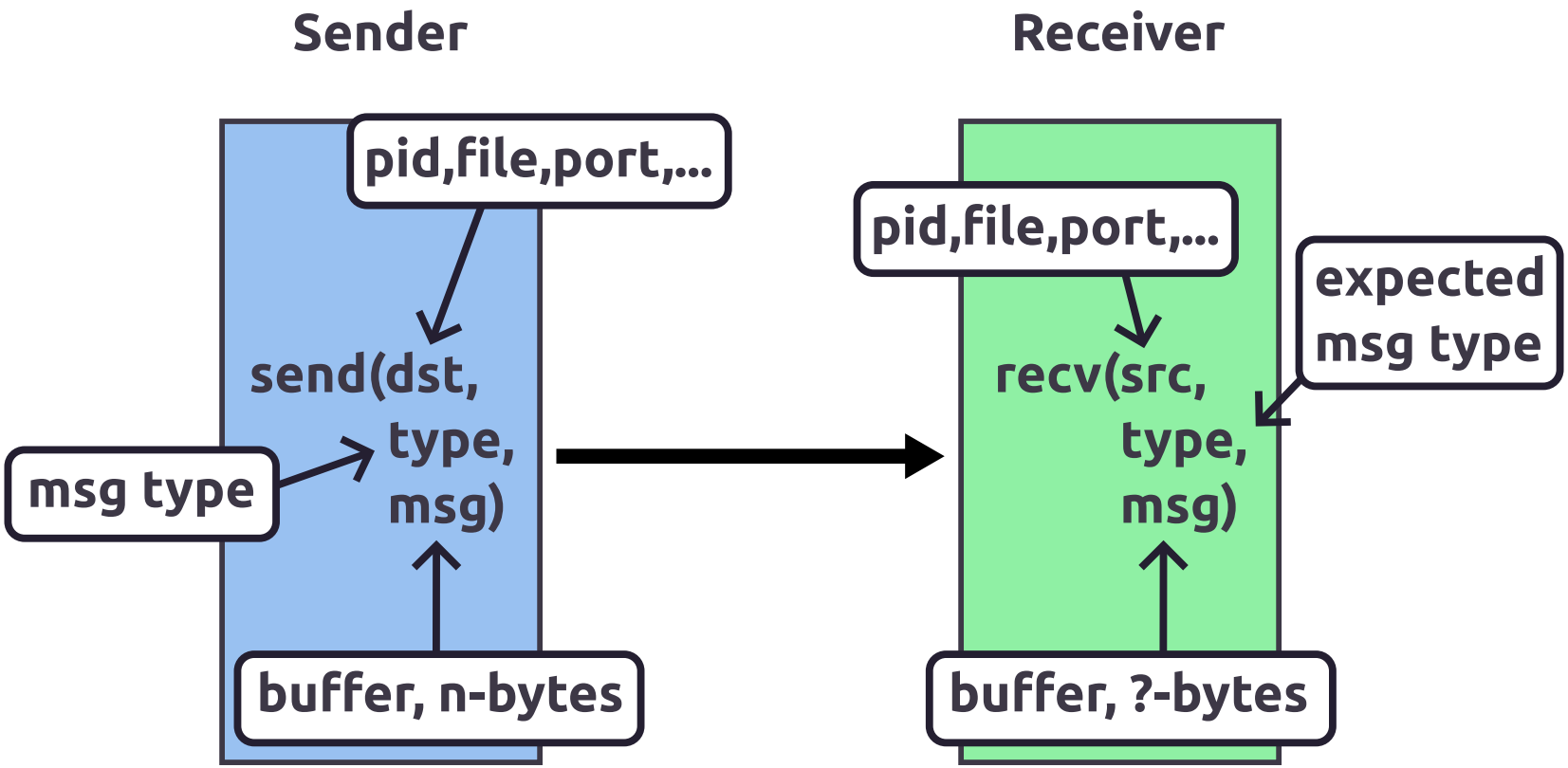
Overview of communication mechanisms



Big picture



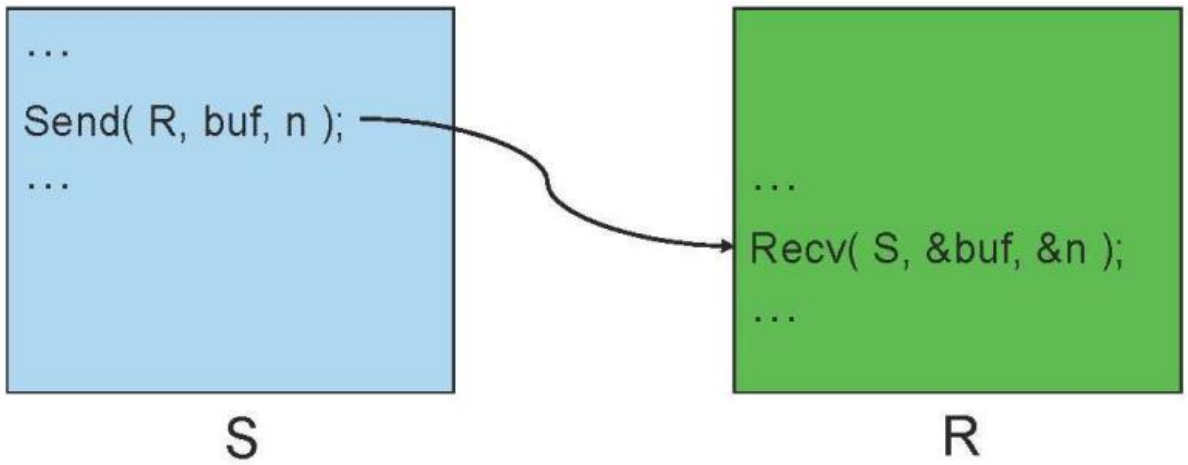
Send and Receive primitives



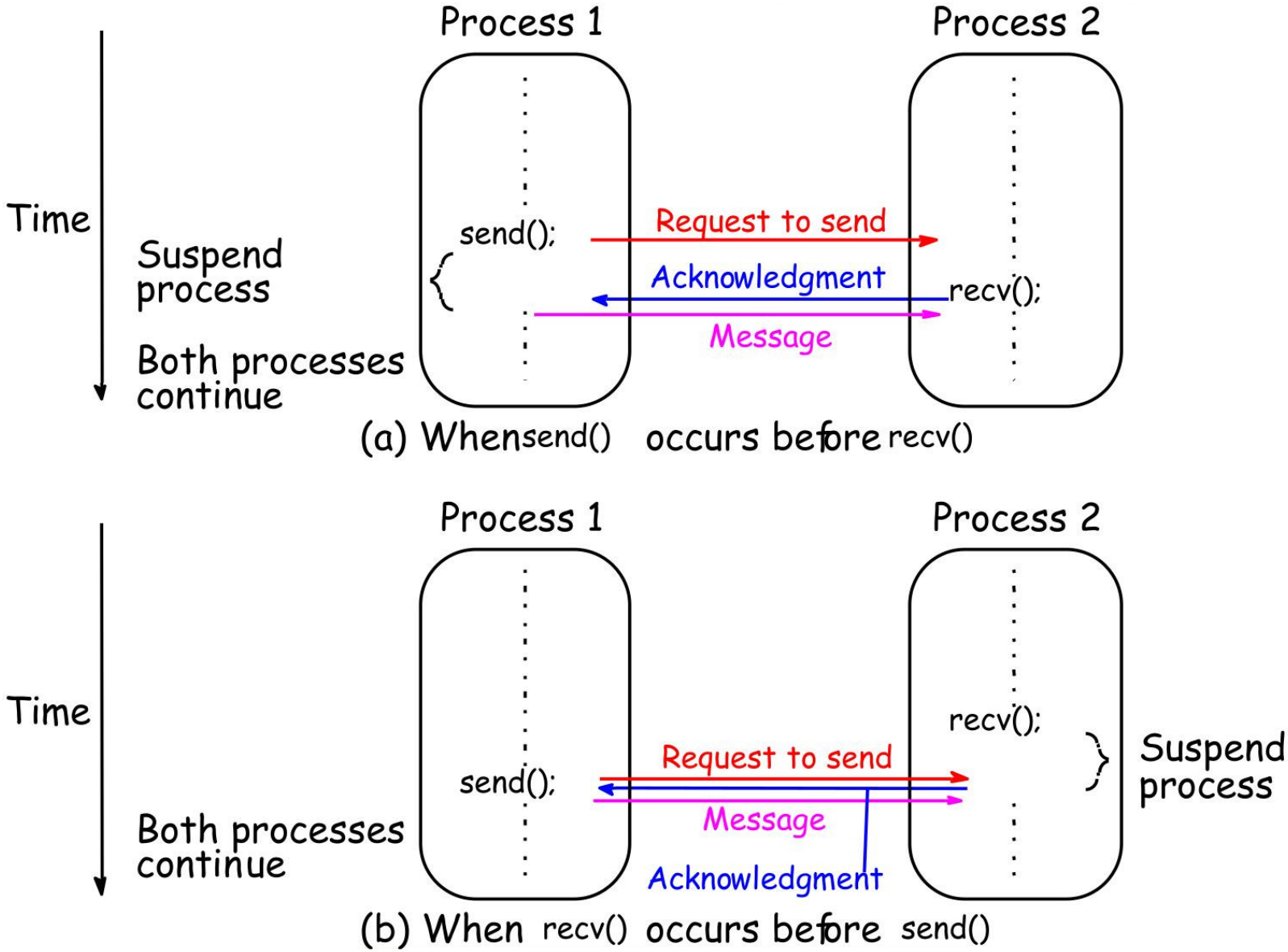
Many ways to design the message passing API

Synchronous Message Passing

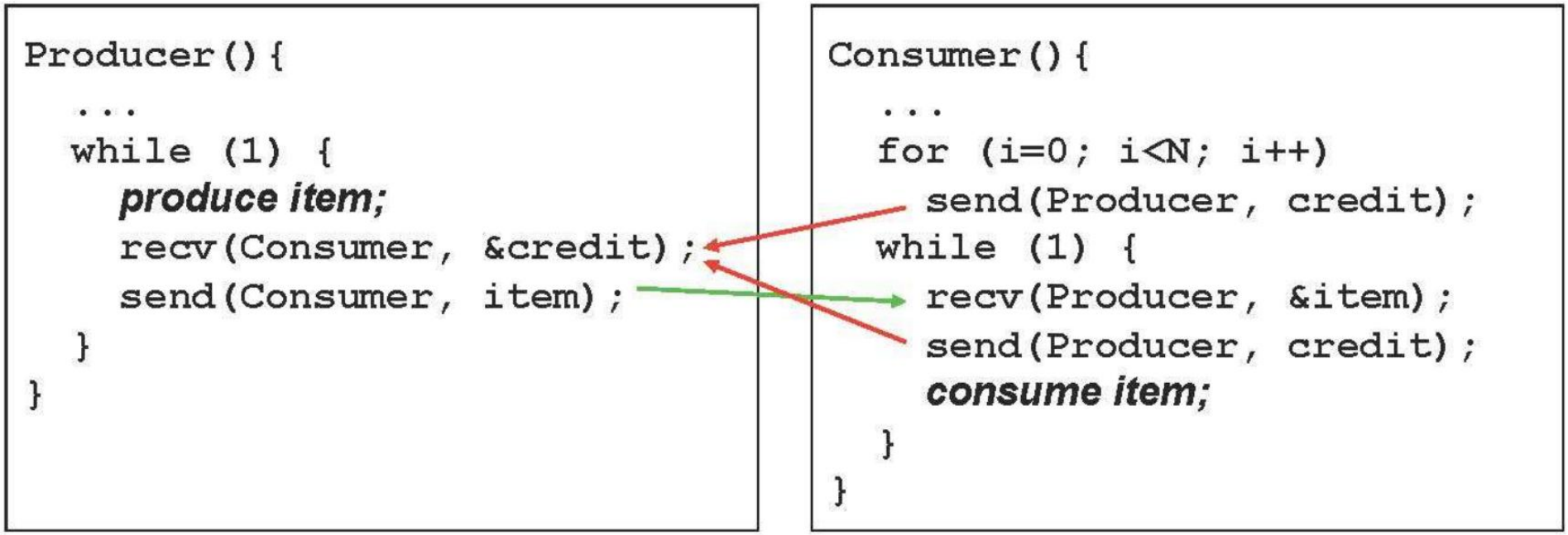
- Synchronizing processes
  - Sender: signal the receiver process that a particular event happens
  - Receiver: blocked until the event has happened
- Moving data between processes
  - Sender: send data to the receiver when data is ready
  - Receiver: collect data when the data has arrived and process is ready



Synchronous send() and recv() using 3-way protocol



Example: Producer-Consumer



- Questions
  - Would it work with multiple producers and 1 consumer?
  - Would it work with 1 producer and multiple consumers?
  - What about multiple producers and multiple consumers?

Summary

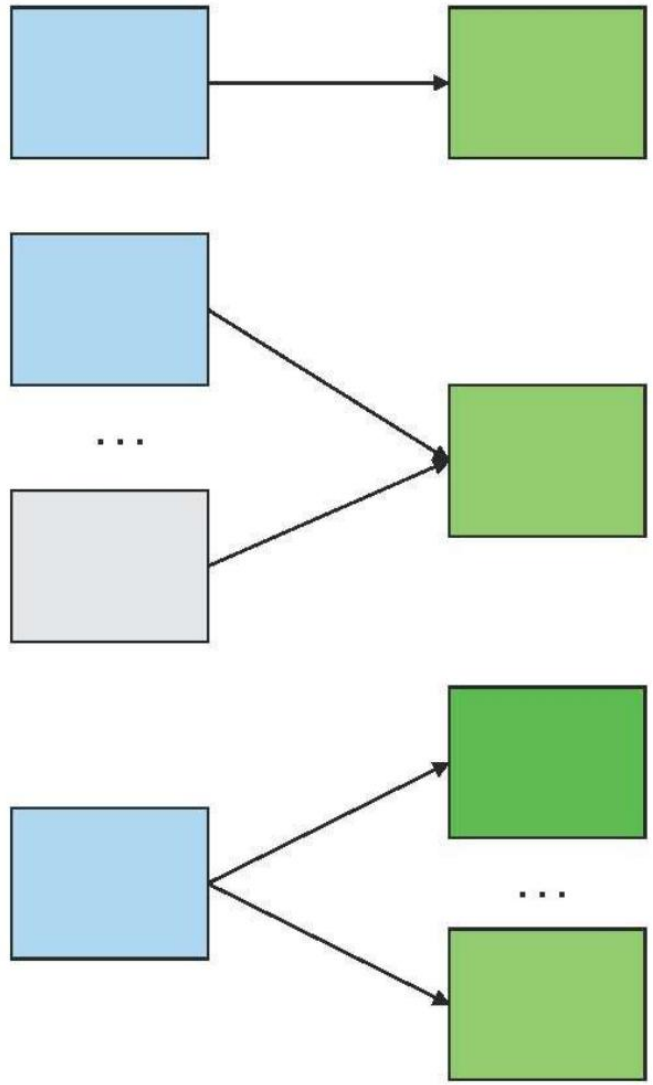
- Mechanism for processes to communicate and to synchronize their actions
- Allows to avoid using shared variables
- If P and Q wish to communicate:
  - Establish a communication link between them
  - Exchange messages via **send**/**receive**
- Layers:
  1. physical (e.g. main memory, hardware bus)
  2. logical (e.g. logical properties)
     - ★ Direct vs. indirect communication
     - ★ Synchronous vs. asynchronous communication
     - ★ Automatic vs. explicit buffering

Outline

- Message Passing
- Implementation issues
- Examples of message-passing systems

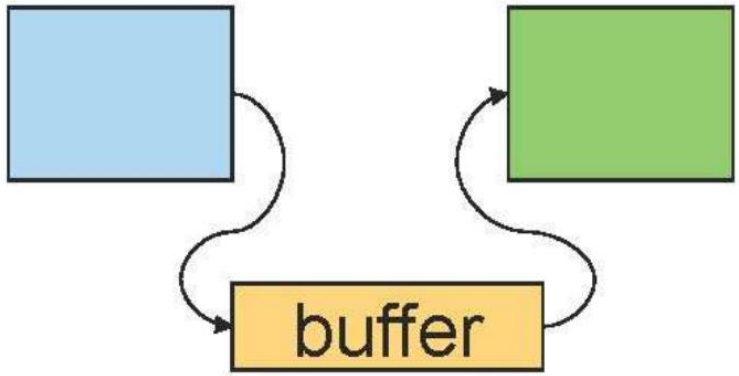
Implementation issues

- Buffering messages
- Direct vs. indirect
- Asynchronous vs. synchronous
- How to handle exceptions?



Buffering messages

- No buffering
  - Sender must wait until the receiver receives the message
  - Rendezvous on each message
- Bounded buffer
  - Finite size
  - Sender is blocked on buffer full
- Unbounded buffer
  - "Infinite" size
  - Sender is never blocked

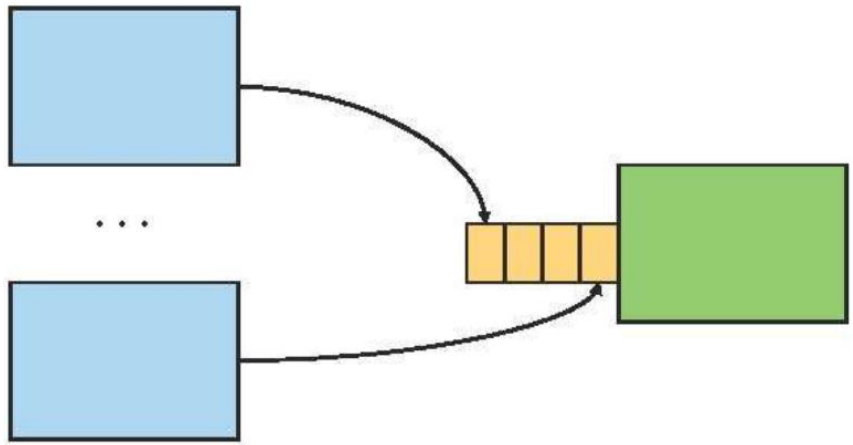


Direct communication

- Processes must name each other explicitly:
  - **send**(P,message) - send a message to process P
  - **receive**(Q,message) - receive a message from process Q
- Properties of communication link
  - A link is associated with exactly one pair of communicating processes
  - Links are established automatically

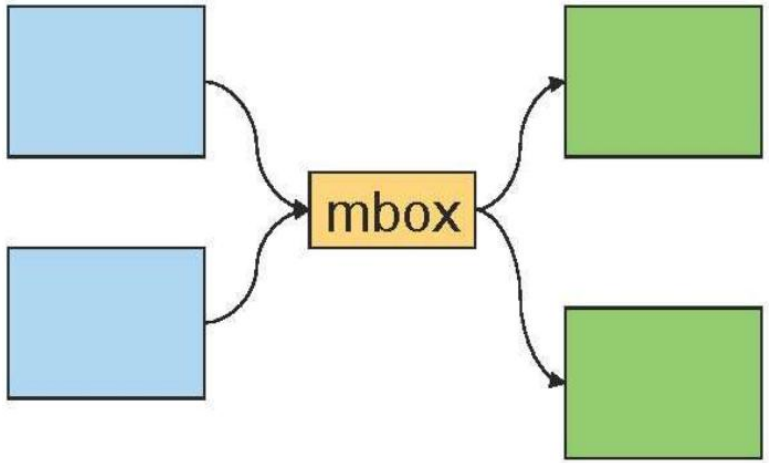
Direct communication with buffer

- A single buffer at the receiver
  - More than one process may send messages to the receiver
  - recv(ANY,msg)
  - To broadcast a message, a sender must send separate messages to receivers
- A buffer at each sender
  - A sender may send a message to multiple receivers
  - To get a message, it requires searching through the whole buffer
- What if process id/name changes?



Indirect communication

- Use mailbox as the abstraction
  - **send**(A, message) - send a message to mailbox A
  - **recv**(A, message) - receive a message from mailbox A
  - Require create/destroy a mailbox
  - Allow many-to-many communication
- Mailbox sharing
  - P1, P2, and P3 share mailbox A. P1 sends, P2 and P3 receive.
  - Who gets the message?
- Mailbox vs. pipe
  - A mailbox allows many to many communication
  - A pipe implies one sender and one receiver

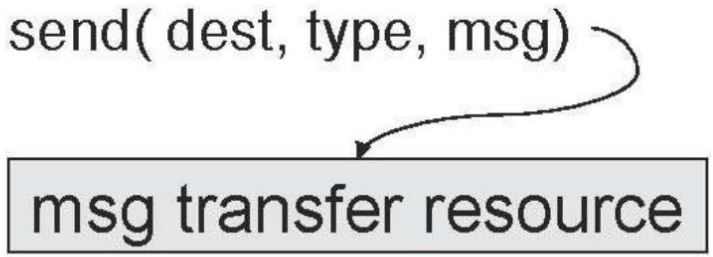


Outline

- Message Passing
- Implementation issues
  - Buffering
  - Indirection
  - Blocking vs. nonblocking
  - Exceptions

Blocking vs. Nonblocking: Send

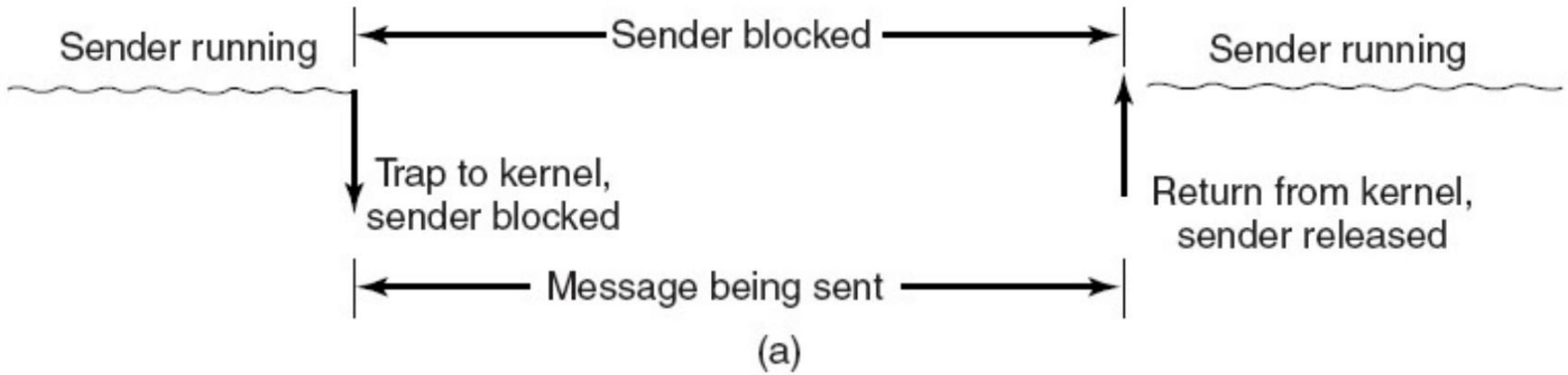
- Blocking
  - Initiate data transfer
  - Either:
    1. Block until data is out of its source memory (buffering)
    2. Block until data is received (no buffering)



- Nonblocking
  - Initiate data transfer and return
  - Completion
    - ★ Require applications to check status
    - ★ Notify or signal the application

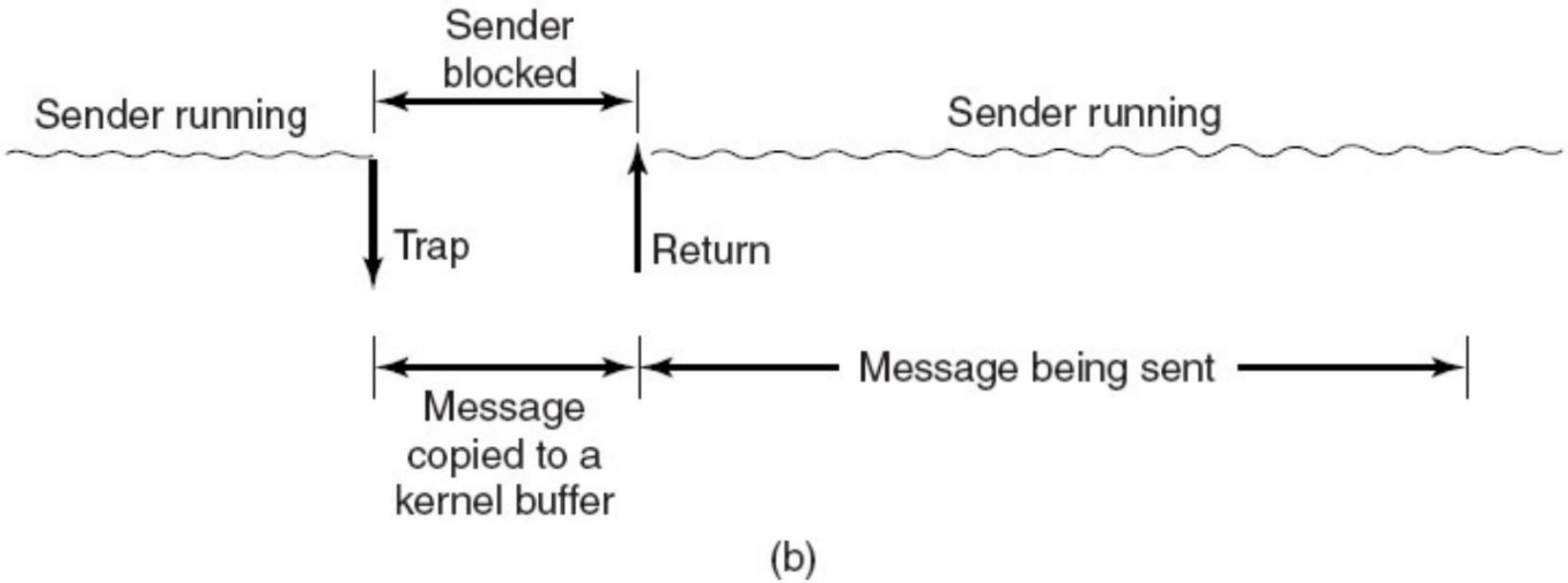
```
status = async_send( dest, type, msg )
...
if !send_complete( status )
    wait for completion;
...
use msg data structure;
...
```

Blocking versus Nonblocking: Send (2)



A blocking send call

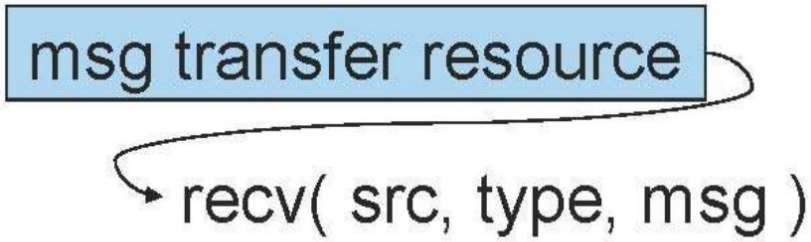
Blocking versus Nonblocking: Send (3)



A nonblocking send call

Blocking vs. Nonblocking: Receive

- Blocking
  - Block until a message is available
- Nonblocking
  - Return either a valid message or a null
  - Null: probe(src)



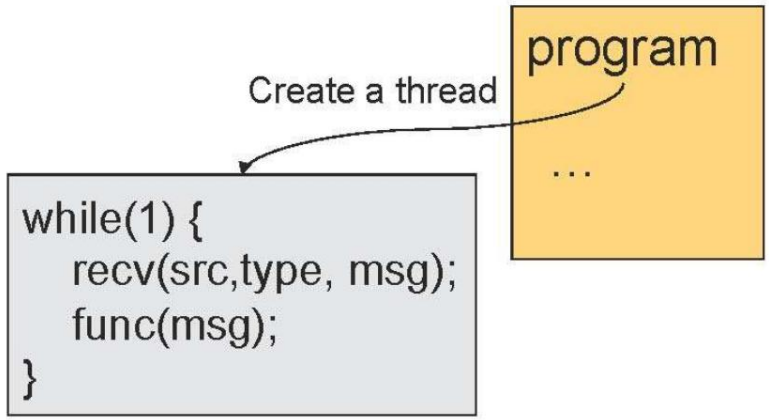

```
status = async_recv( src, type, msg );
if ( status == SUCCESS )
    consume msg;
...
while ( probe(src) != HaveMSG )
    wait for msg arrival
recv( src, type, msg );
consume msg;
```

Event Handler vs. Receive

- hrecv(src, type, msg, func)
  - msg is an arg of func
  - Execute "func" on a message arrival

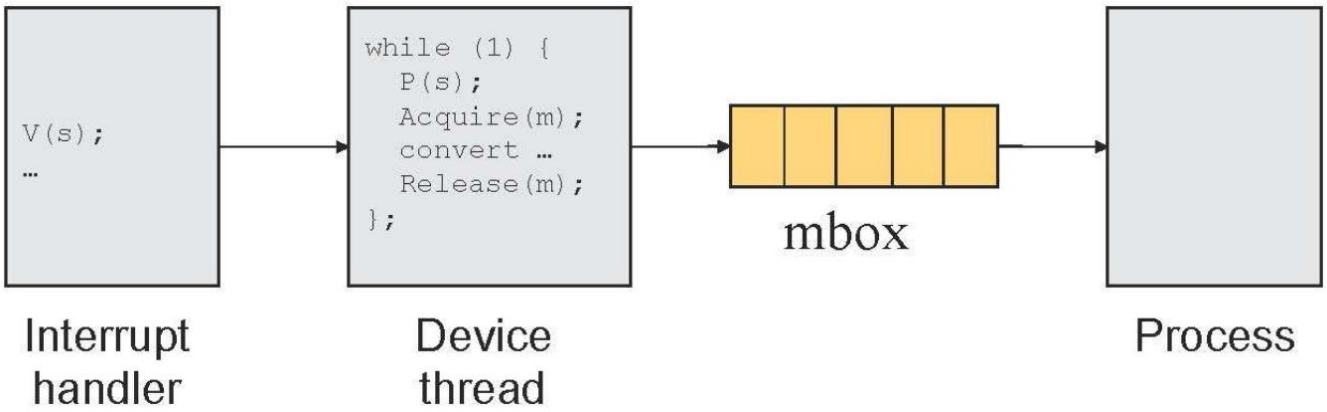
```
void func( char * msg ) {
  ...
}

...
hrecv( src, type, msg, func)
...
```

- Which one is more powerful?
  - Recv with a thread can emulate a Handler
  - Handler can be used to emulate recv by using Monitor



Example: Keyboard Input

- How do you implement keyboard input?
  - Need an interrupt handler
  - Generate a mbox message from the interrupt handler
- Suppose a keyboard device thread converts input characters into an mbox message
  - How would you synchronize between the keyboard interrupt handler and device thread?
  - How can a device thread convert input into mbox messages?

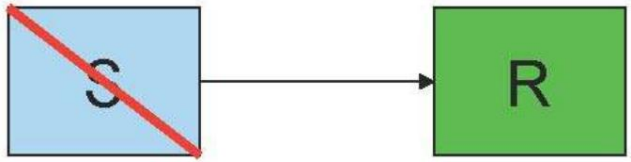


Outline

- Message Passing
- Implementation issues
  - Buffering
  - Indirection
  - Blocking vs. nonblocking
  - Exceptions

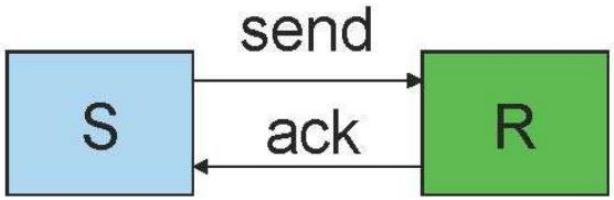
Exception: Process Termination

- R waits for a message from S, but S has terminated
  - Problem: R may be blocked forever



- S sends a message to R, but R has terminated
  - Problem: S has no buffer and will be blocked forever

Exception: Message Loss



- Use ack and timeout to detect and retransmit a lost message
  - Receiver sends an ack message for each message
  - Sender is blocked until an ack message is back or timeout
    status = send(dest, msg, timeout);
  - If timeout happens and no ack, sender retransmits the message

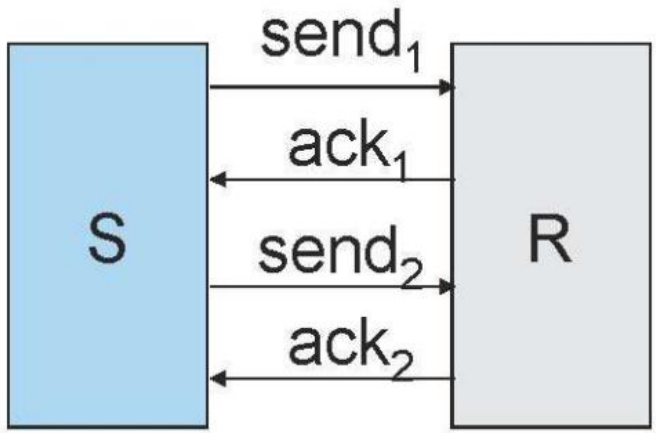
- Issues
  - Losing ack messages
  - Duplicates

Exception: Message Loss (2)

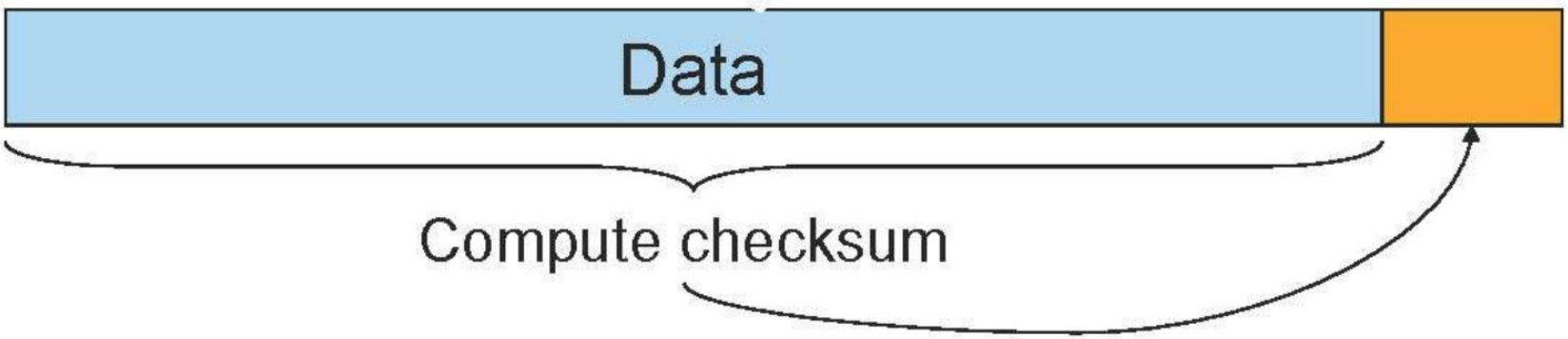
- Retransmission must handle
  - Duplicate messages on receiver side
  - Out-of-sequence ack messages on sender side

- Retransmission
  - Use sequence number for each message to identify duplicates
  - Remove duplicates on receiver side
  - Sender retransmits on an out-of-sequence ack

- Reduce ack messages
  - Bundle ack messages
  - Receiver sends NOACK messages: can be complex
  - Piggy-back acks in send messages



Exception: Message Corruption



- Detection
  - Compute a checksum over the entire message and send the checksum (e.g. CRC) as part of the message
  - Recompute a checksum at receiver and compare with the checksum in the message
- Correction
  - Trigger retransmission
  - Use correction codes (e.g. ECC) to recover

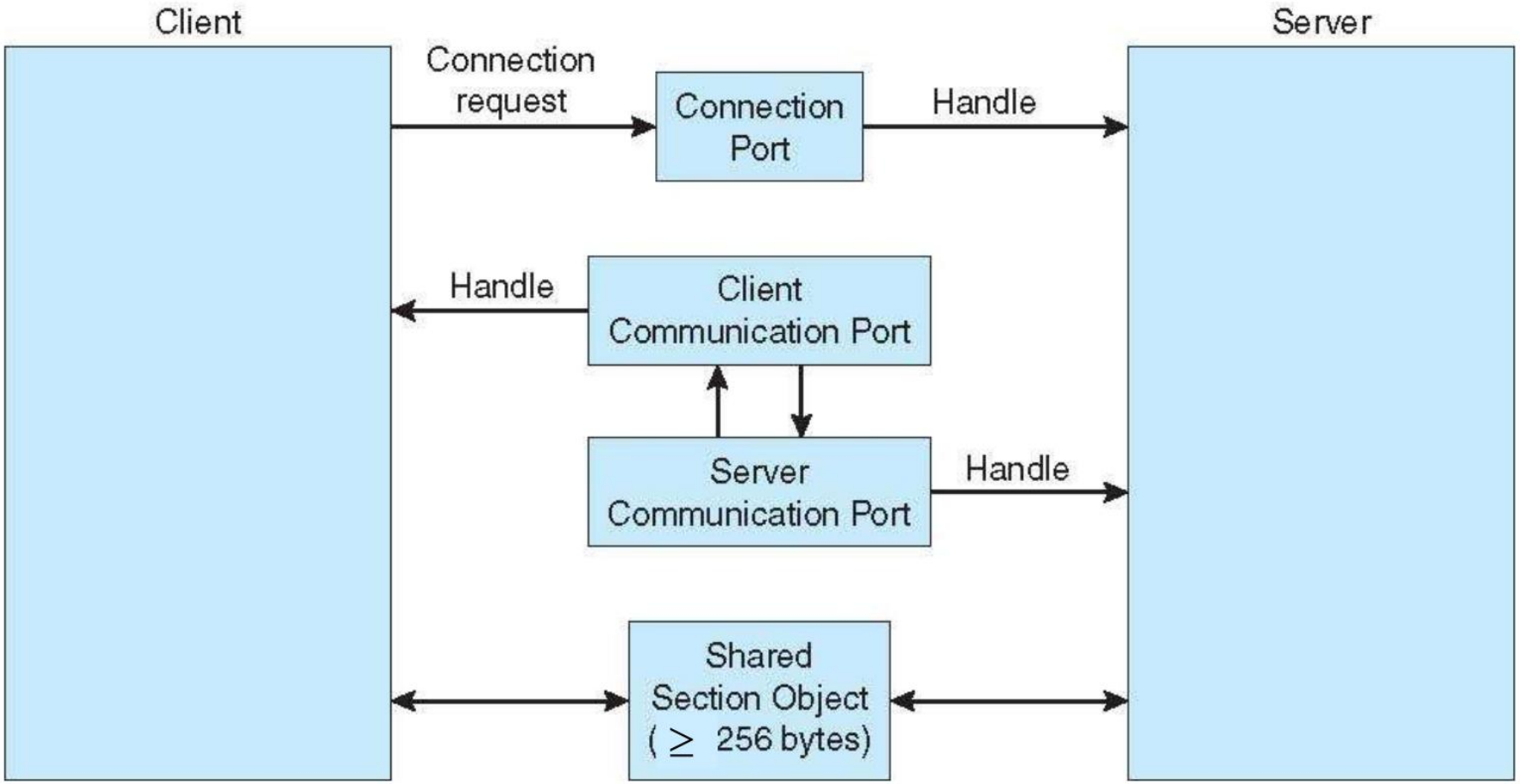
Outline

- Message Passing
- Implementation issues
- Examples of message passing systems

Mach microkernel (Mac OS X)

- Mach communication is message based
  - System calls are messages
  - Each task gets two mailboxes at creation- Kernel and Notify
  - Only three system calls needed for message transfer `msg_send()`, `msg_receive()`, `msg_rpc()`
  - Mailboxes needed for communication, created via `port_allocate()`

Local Procedure Calls in Windows XP



References

- A. S. Tanenbaum, Modern Operating Systems.
- A. Silberschatz et. al., Operating System Concepts.
- B. Wilkinson et. al., Parallel Programming: Techniques and Applications Using Networked Workstations and Parallel Computers.
- B. Barney, "Message Passing Interface (MPI)", Livermore Computing.

Thanks for your attention!

Questions?