CS 422/522 Design & Implementation of Operating Systems

Lecture 12: Message Passing

Zhong Shao
Dept. of Computer Science
Yale University

Acknowledgement: some slides are taken from previous versions of the CS422/522 lectures taught by Prof. Bryan Ford and Dr. David Wolinsky, and also from the official set of slides accompanying the OSPP textbook by Anderson and Dahlin.

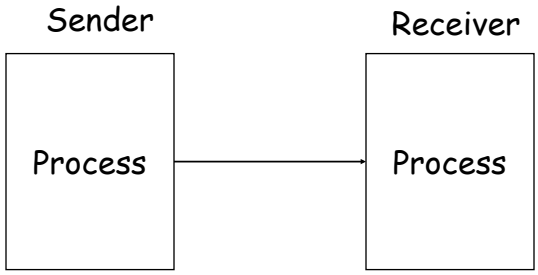
Motivation

- Locks, semaphores, monitors are good but they only work under the shared-memory model
- How to synchronize / schedule / communicate between processes that reside in different address spaces / different machines ?
- Can we have a single set of primitives that are transparently extensible to the distributed environment ?

Interprocess communication (IPC)

- Mechanism for processes to communicate and to synchronize their actions.
- Message system – processes communicate with each other without resorting to shared variables.
- IPC facility provides two operations:
  - **send** a *message* – message size fixed or variable
  - **receive** a *message*
- If *P* and *Q* wish to communicate, they need to:
  - establish a *communication link* between them
  - exchange messages via send/receive
- Implementation of communication link
  - physical (e.g., shared memory, hardware bus)
  - logical (e.g., logical properties)

The big picture

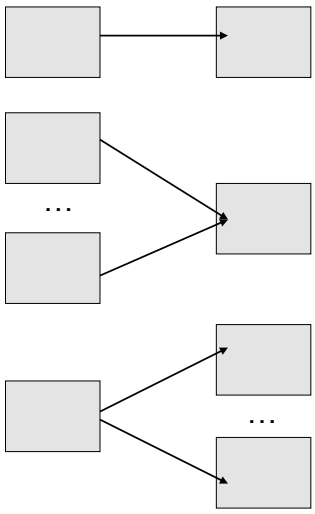


Message passing API

- Generic API
  - `send( dest, msg ), receive( src, msg )`
- What should the "`dest`" and "`src`" be?
  - pid
  - file: e.g. a pipe
  - port: network address, pid, etc
  - no src: receive any message
  - src combines both specific and any
- What should "msg" be?
  - Need both buffer and size for a variable sized message

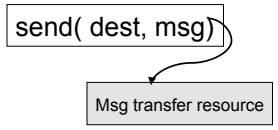
Implementation issues

- Asynchronous vs. synchronous
- Event handler vs. receive
- How to buffer messages?
- Direct vs. indirect
- 1-to-1 vs. 1-to-many vs. many-to-one vs. many-to-many
- Unidirectional vs. bidirectional
- What is the size of a message?
- How to handle exceptions (when bad things happen)?



Synchronous vs. asynchronous: send

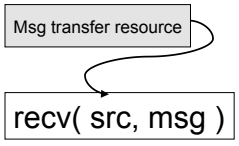
- Synchronous
  - Will not return until data is out of its source memory
  - Block on full buffer
- Asynchronous
  - Return as soon as initiating its hardware
  - Completion
    - * Require applications to check status
    - * Notify or signal the application
  - Block on full buffer




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

Synchronous vs. asynchronous: receive

- Synchronous
  - Return data if there is a message
  - Block on empty buffer
- Asynchronous
  - Return data if there is a message
  - Return status if there is no message (probe)

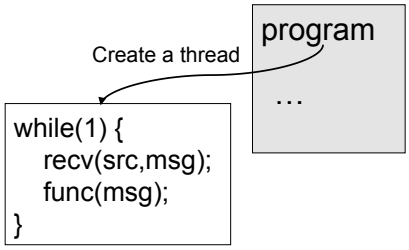



```
status = async_recv( src, msg );
if ( status == SUCCESS )
   consume msg;

while ( probe(src) != HaveMSG )
   wait for msg arrival
recv( src, msg );
consume msg;
```

Event handler vs. receive

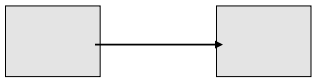
- hrecv(src, msg, func)
  - msg is an arg of func
  - Execute "func" on a message arrival
- Which one is more powerful?
  - Recv with a thread can emulate a Handler
  - Handler can be used to emulate Recv by using Monitor
- Pros and Cons
  - Handler is better for event-based applications (no need to think about threads), but concurrent executions require more thoughts
  - Recv with a thread require thread context switches but can run concurrently

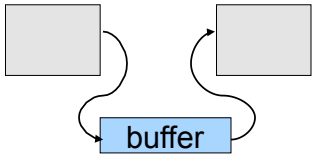
```
void func( char * msg ) {
   …
}

…
hrecv( src, msg, func)
…
```



Buffering

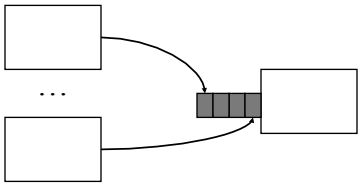
- No buffering
  - Sender must wait until the receiver receives the message
  - Rendezvous on each message
- Bounded buffer
  - Finite size
  - Sender blocks on buffer full
  - Use mesa-monitor to solve the problem
- Unbounded buffer
  - "Infinite" size
  - Sender never blocks



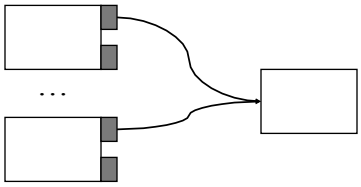


Direct communication

- A single buffer at the receiver
  - More than one process may send messages to the receiver
  - To receive from a specific sender, it requires searching through the whole buffer

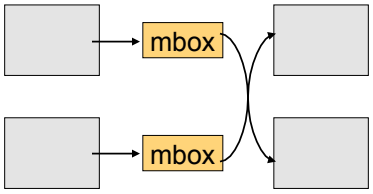


- A buffer at each sender
  - A sender may send messages to multiple receivers
  - To get a message, it also requires searching through the whole buffer



Indirect communication

- Use a "mailbox" to allow many-to-many communication
  - Requires open/close a mailbox before using it
- Where should the buffer be?
  - A buffer, its mutex and condition variables should be at the mailbox
- Fixed sized messages?
  - Not necessarily. One can break a large message into packets
- Are there any differences between a mailbox and a pipe?
  - A mailbox allows many to many communication
  - A pipe implies one sender and one receiver

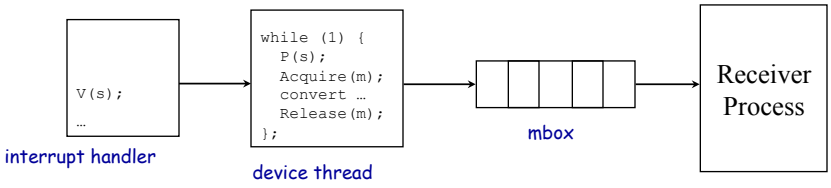


Indirect communication (cont' d)

- Mailbox sharing
  - P_1, P_2, and P_3 share mailbox A.
  - P_1, sends; P_2 and P_3 receive.
  - Who gets the message?
- Solutions
  - Allow a link to be associated with at most two processes.
  - Allow only one process at a time to execute a receive operation.
  - Allow the system to select arbitrarily the receiver. Sender is notified who the receiver was.

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?



Example: Sockets API

- Abstraction for TCP and UDP
  - Learn more about internetworking in the future
- Addressing
  - IP address and port number (2^{16} ports available for users)
- Create and close a socket

```
sockid = socket (af, type, protocol);
sockerr = close(sockid);
```

- Bind a socket to a local address

```
sockerr = bind(sockid, localaddr, addrlength);
```

- Negotiate the connection

```
listen(sockid, length);
accept(sockid, addr, length);
```

- Connect a socket to destination

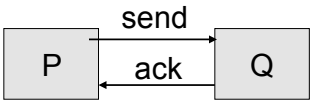
```
Connect(sockid, destaddr, addrlength);
```

Unix pipes

- An output stream connected to an input stream by a chunk of memory (a queue of bytes).
- Send (called write) is non-blocking
- Receive (called read) is blocking
- Buffering is provided by OS
- Message boundaries erased while reading

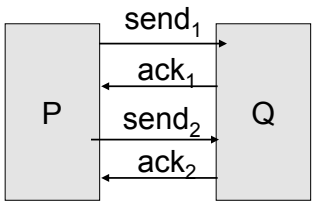
Exception: losing messages

- Use ack and timeout to detect and retransmit a lost message
  - Require the receiver to send an ack message for each message
  - Sender blocks until an ack message is back or timeout
    status = send(dest, msg, timeout);
  - If timeout happens and no ack, then retransmit the message
- Issues
  - Duplicates
  - Losing ack messages



Exception: losing messages (cont' d)

- 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



Summary

- Message passing
  - Move data between processes
  - Implicit synchronization
- Implementation issues
  - Synchronous method is most common
  - Asynchronous method provides overlapping but requires careful design considerations
  - Indirection makes implementation flexible
  - Exception needs to be carefully handled