COS 318: Operating Systems
Message Passing
Motivation

- Locks, semaphores, monitors are good but they only work under the shared-address-space model
  - Threads in the same process
  - Processes that share an address space

- We have assumed that processes/threads communicate via shared data (counter, producer-consumer buffer, …)

- How to synchronize and communicate among processes with different address spaces?
  - Inter-process communication (IPC)

- Can we have a single set of primitives that work for all cases: single machine OS, multiple machines same OS, multiple machines multiple OS, distributed?
With No Shared Address Space

- No need for explicit mutual exclusion primitives
  - Processes cannot touch the same data directly
- Communicate by sending and receiving explicit messages: Message Passing
- Synchronization is implicit in message passing
  - No need for explicit mutual exclusion
  - Event ordering via sending and receiving of messages
- More portable to different environments, though lacks some of the convenience of a shared address space
- Typically, communication in consummated via a send and a matching receive
Sending A Message

Within A Computer

P1
Send()

P2
Recv()

OS Kernel

Across A Network

P1 can send to P2, P2 can send to P1
Simple Send and Receive

send ( dest, data ), receive ( src, data )

- Send “data” specifies where the data are in sender’s address space
-Recv “data” specifies where the incoming message data should be put in receiver’s address space
Simple API

send( dest, data ), receive( src, data )

- Destination or source
  - Direct address: node Id, process Id
  - Indirect address: mailbox, socket, channel, …

- Data
  - Buffer (addr) and size
  - Anything else that specifies the source or destination data
Simple Semantics

- Send call does not return until data have been copied out of source data structure
  - Could be to destination process/machine, or to OS buffer, … (there are variants depending on this)
  - At that point, source data structure can be safely overwritten without changing the data carried by the message

- Receive does not return until data have been received and copied into destination data structure

- This is called Synchronous Message Passing
  - Makes synchronization implicit and easy
  - But processes wait around a lot for send and receive calls to return, so can hurt performance
Issues/options

- Asynchronous vs. synchronous
- Buffering of messages
- Matching of messages
- Direct vs. indirect
- Data alone, or function invocation?
- How to handle exceptions (when bad things happen)?
Synchronous vs. Asynchronous Send

- **Synchronous**
  - Send will not return until data are copied out of source data structure
  - If a buffer is used for messaging and it is full, block

- **Asynchronous**
  - Return before data are copied out of source data structure
  - Completion of send or of copy:
    - Applications must check status
    - Notify or signal the application
  - Block on full buffer

```c
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 in incoming buffer
  - If not, buffer

- **Asynchronous**
  - Return data if there is a message
  - Return status if there is no message (probe)

```c
recv(src, msg);
/* consume if available, or move on */
status = async_recv(src, msg);
if (status == SUCCESS)
  consume msg;
/* At some point, wait for message */
while (probe(src) != haveMSG)
  wait for msg arrival;
recv(src, msg);
consume msg;
```
Buffering

Buffer holds message contents so sender and receiver can be decoupled

- No buffering
  - Sender must wait until the receiver receives message
  - Rendezvous on each msg

- Finite buffer
  - Sender blocks on buffer full
Synchronous Send/Recv Within a System

Synchronous send:
- Call send system call with M
- Send system call:
  - No buffer in kernel: block
  - Copy M to kernel buffer

Synchronous recv:
- Call recv system call
- Recv system call:
  - No M in kernel: block
  - Copy to user buffer

How to manage kernel buffer?

On distributed machines/OSes, buffers at one/both ends
What if Buffers Fill Up?

- Make processes wait (can be hard to do when they are on different machines)
- Drop messages
- Don’t send fast enough to fill up buffers: flow control
- Credits
  - Receivers provide credits based on space availability
  - Senders don’t send unless they have the credits to do so
Direct Addressing Example

Q: Does this work?
- 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?
Indirect Addressing Example

- 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?
Indirect Communication

- **Names**
  - mailbox, socket, channel, …

- **Properties**
  - Some allow one-to-one (e.g. pipe)
  - Some allow many-to-one or one-to-many communications (e.g. mailbox)
Mailbox Message Passing

- Message-oriented 1-way communication
- Data structure
  - Mutex, condition variable, buffer for messages
- Operations
  - Init, open, close, send, receive, ...
- Does the sender know when receiver gets a message?

```
mbox_send(M)
```

```
mbox_recv(M)
```
Example: Keyboard Input

- **Interrupt handler**
  - Get the input characters and give to device thread

- **Device thread**
  - Generate a message and send it to mailbox of an input process

```c
V(s);
...
while (1) {
P(s);
    Acquire(m);
    convert ...
    Release(m);
};
```

![Diagram of interrupt handler and device thread](image-url)
Sockets

- Sockets
  - Bidirectional (unlike mailbox)
  - Unix domain sockets (IPC)
  - Network sockets (over network)
  - Same APIs

- Two types
  - Datagram Socket (UDP)
    - Collection of messages
    - Best effort
    - Connectionless
  - Stream Socket (TCP)
    - Stream of bytes (like pipe)
    - Reliable
    - Connection-oriented
Network Socket Address Binding

- A network socket binds to
  - Host: IP address
  - Protocol: UDP/TCP
  - Port:
    - Well known ports (0..1023), e.g. port 80 for Web
    - Unused ports available for clients (1025..65535)

**Q**: Why do we need ports?
- Indirection: No need to know which process to communicate with
- Updating software on one side won’t affect another side
Communication with Stream Sockets

Client

Create a socket
Connect to server
Send request
Receive response

Server

Create a socket
Bind to a port
Listen on the port
Accept connection
Receive request
Send response

Establish connection
request
reply
Sockets API

- Create and close a socket
  - sockid = socket(af, type, protocol);
  - sockerr = close(sockid);

- Bind a socket to a local address
  - sockerr = bind(sockid, localaddr, addrlen);

- Negotiate the connection
  - listen(sockid, length);
  - accept(sockid, addr, length);

- Connect a socket to destination
  - connect(sockid, destaddr, addrlen);

- Message passing
  - send(sockid, buf, size, flags);
  - recv(sockid, buf, size, flags);
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
What if things go bad?

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

- S sends a message to R, but R has terminated
  - 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 for each msg
  - 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: Message Loss, contd.

- **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
  - Piggy-back acks in send messages
Exception: Message Corruption

- **Detection**
  - Compute a checksum over the entire message and send the checksum (e.g. CRC code) as part of the message
  - Recompute a checksum on receive and compare with the checksum in the message

- **Correction**
  - Trigger retransmission
  - Use correction codes to recover
Message Passing Interface (MPI)

- A message-passing library for parallel machines
  - Implemented at user-level for high-performance computing
  - Portable

- Basic (6 functions)
  - Works for most parallel programs

- Large (125 functions)
  - Blocking (or synchronous) message passing
  - Non-blocking (or asynchronous) message passing
  - Collective communication

- References
  - http://www.mpi-forum.org/
Remote Procedure Call (RPC)

- Make remote procedure calls
  - Similar to local procedure calls
  - Examples: SunRPC, Java RMI

- Restrictions
  - Call by value
  - Call by object reference (maintain consistency)
  - Not call by reference

- Different from mailbox, socket or MPI
  - Remote execution, not just data transfer

- References
RPC Model

Caller (Client)  

RPC call

Request message including arguments

Function execution w/ passed arguments

Reply message including a return value

Server

Return (same as local calls)

Compile time type checking and interface generation
RPC Mechanism

Client program

- Call
- Encode/marshall
- Send
- Receive

Server program

- Call
- Return
- Decode/unmarshall
- Send
- Receive

Client stub

RPC runtime

Server stub

RPC runtime

ClientId
RPCId
Call
Args

Reply
Results
Summary

- **Message passing**
  - Move data between processes
  - Implicit synchronization
  - Many API design alternatives (Socket, MPI)
  - Indirection is helpful

- **Implementation and Semantics**
  - Synchronous method is most common
  - Asynchronous method provides overlapping, but required careful design and implementation decisions
  - Indirection makes implementation flexible
  - Exception needs to be carefully handled

- **RPC**
  - Remote execution like local procedure calls
  - With constraints in terms of passing data
Appendix:
Message Passing Interface (MPI)
Hello World using MPI

#include "mpi.h"
#include <stdio.h>

int main( int argc, char *argv[] )
{
    int rank, size;
    MPI_Init( &argc, &argv );
    MPI_Comm_rank( MPI_COMM_WORLD, &rank );
    MPI_Comm_size( MPI_COMM_WORLD, &size );
    printf( "I am %d of %d\n", rank, size );
    MPI_Finalize();
    return 0;
}
Blocking Send

- MPI_Send(buf, count, datatype, dest, tag, comm)
  - buf  address of send buffer
  - count # of elements in buffer
  - datatype data type of each send buffer element
  - dest rank of destination
  - tag message tag
  - comm communicator

- This routine may block until the message is received by the destination process
  - Depending on implementation
  - But will block until the user source buffer is reusable

- More about message tag later
Blocking Receive

- **MPI_Recv(buf, count, datatype, source, tag, comm, status)**
  - **buf** address of receive buffer (output)
  - **count** maximum # of elements in receive buffer
  - **datatype** datatype of each receive buffer element
  - **source** rank of source
  - **tag** message tag
  - **comm** communicator
  - **status** status object (output)

- Receive a message with the specified tag from the specified comm and specified source process
- **MPI_Get_count(status, datatype, count)** returns the real count of the received data
More on Send & Recv

- Can send from source to destination directly
- Message passing must match
  - Source rank (can be MPI_ANY_SOURCE)
  - Tag (can be MPI_ANY_TAG)
  - Comm (can be MPI_COMM_WORLD)
Buffered Send

- **MPI_Bsend(buf, count, datatype, dest, tag, comm)**
  - `buf` address of send buffer
  - `count` # of elements in buffer
  - `Datatype` type of each send element
  - `dest` rank of destination
  - `tag` message tag
  - `comm` communicator

- May buffer; user can use the user send buffer right away
- **MPI_Buffer_attach(), MPI_Buffer_detach** creates and destroy the buffer

- **MPI_Ssend**: Returns only when matching receive posted. No buffer needed.
- **MPI_Rsend**: assumes received posted already (programmer’s responsibility)
Non-Blocking Send

- **MPI_Isend(buf, count, datatype, dest, tag, comm, *request)**
  - `request` is a handle, used by other calls below

- **Return as soon as possible**
  - Unsafe to use buf right away

- **MPI_Wait(*request, *status)**
  - Block until send is done

- **MPI_Test(*request, *flag, *status)**
  - Return the status without blocking

```c
MPI_Isend(...)

Work to do

MPI_Wait(...)

MPI_Isend(...)  

Work to do

MPI_Test(..., flag,...);
while ( flag == FALSE) {

More work
}
```
Non-BlockingRecv

- `MPI_Irecv(buf, count, datatype, dest, tag, comm, *request, ierr)`
- Return right away
- `MPI_Wait()`
  - Block until finishing receive
- `MPI_Test()`
  - Return status
- `MPI_Probe(source, tag, comm, flag, status, ierror)`
  - Is there a matching message?

```c
while ( flag == FALSE) {
    MPI_Irecv(…)
    or MPI_recv(…)
}
More work
```