COS 318: Operating Systems

Message Passing

(http://www.cs.princeton.edu/courses/cos318/)

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
- How to synchronize / schedule / communicate among processes that reside in different address spaces, and even on different machines?
  - Inter-process communication (IPC)
- Can we have a single set of primitives that are transparently extensible to the distributed environment?

Sending A Message

Within A Computer

P1
Send()
OS Kernel

P2
Recv()

Across A Network

Network
OS

COS461

P1 can send to P2, P2 can send to P1

API Issues

Generic API

send( dest, msg ), receive( src, msg )

- Destination or source
  - Direct address: node Id, process Id
  - Indirect address: mailbox, socket, channel, ...
- Message (msg)
  - Buffer (addr) and size
  - Message type, buffer and size
Issues/options

- Asynchronous vs. synchronous
- Event handler vs. simple receive
- How to match messages
- How to buffer messages
- Direct vs. indirect communication
- How to handle exceptions (when bad things happen?)

Synchronous vs. Asynchronous Send

- Synchronous
  - Will not return until data is out of its source memory
  - If a buffer is used for messaging and it is full, block
- Asynchronous
  - Return as soon as initiate send, regardless of whether data out of source memory
  - Completion
    - Applications must check status
    - Notify or signal the application
    - Block on full buffer

send( dest, msg)

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
  
recv( src, msg )

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

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

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

Buffering

- 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?

Indirect Addressing Example

Producer()
{
  while (1) {
    produce item;
    send(prodMbox, credit);
    send(consMbox, item);
  }
}

Consumer()
{
  for (i=0; i<N; i++)
    send(prodMbox, credit);
  while (1) {
    recv(consMbox, &item);
    consume item;
    send(prodMbox, credit);
  }
}

- 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 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

```
while (1) {
  Acquire(m);
  convert ...;
  Release(m);
}
```

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)
- Why ports?
  - Indirection: No need to know which process to communicate with
  - Updating software on one side wont affect another side
**Communication with Stream Sockets**

<table>
<thead>
<tr>
<th>Client</th>
<th>Server</th>
</tr>
</thead>
<tbody>
<tr>
<td>Create a socket</td>
<td>Create a socket</td>
</tr>
<tr>
<td>Bind to a port</td>
<td>Bind to a port</td>
</tr>
<tr>
<td>Listen on the port</td>
<td>Listen on the port</td>
</tr>
<tr>
<td>Connect to server</td>
<td>Connect to server</td>
</tr>
<tr>
<td>Send request</td>
<td>Send request</td>
</tr>
<tr>
<td>Receive response</td>
<td>Receive response</td>
</tr>
</tbody>
</table>

**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

**Message-Passing Implementation Issues**

- 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
  - 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
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)
- Server
- Request message including arguments
- Function execution w/ passed arguments
- Reply message including a return value
- Return (same as local calls)
- Compile time type checking and interface generation

RPC Mechanism

- Client program
- Server program
- Call
- Return
- Encode/marshall
- Decode/unmarshall
- Receive
- Send
- Client stub
- 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
- RPC
  - Remote execution like local procedure calls
  - With constraints in terms of passing data
- 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
Appendix:
Message Passing Interface (MPI)

Hello World using MPI

```c
#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)

```
MPI_Send(..., dest=1, tag=1, comm=X)

MPI_Recv(..., Source=0, tag=1, comm=X)
```

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)

```
MPI_Bsend(buf, ...)  

Buffer
```

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

```
MPI_Isend(...)  

Work to do

MPI_Wait(...)  

Work to do

MPI_Test(...)  

More work
```

Non-Blocking Recv

- 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, ierr, err)
    - Is there a matching message?

```
MPI_Irecv(...)  

Work to do

MPI_Wait(...)  

MPI_Test(...)  

More work

MPI_Probe(...)  

while ( flag == FALSE) {
  
}

MPI_Irecv(...)  

More work

or MPI_recv(...)  

More work
```