COS 318: Operating Systems

Message Passing

Jaswinder Pal Singh
Computer Science Department
Princeton University

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

- In class, here, Monday after break at 11 am
- All topics covered before break
  - OS structure, processes and threads
  - Synchronization
  - Scheduling
  - Deadlocks
  - I/O devices
  - Other
- Closed book and notes
  - No book, no notes, no cheat sheet
  - No devices and no online access
Today’s Topics

- Message passing
Sending A Message

Within A System

Send() \rightarrow \text{OS Kernel} \rightarrow \text{Recv()}

Across A Network

\text{Send()} \rightarrow \text{Network} \rightarrow \text{Recv()}

\text{COS461}
Synchronous Message Passing
(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?
API Issues

- **Message**
  - Buffer and size
  - Message type, buffer and size

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

```
send(dest, msg)
```

```
recv(src, msg)
```
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
  - Like real mailbox: letters/messages, not sure about receiver

- 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 a mailbox of an input process

```
V(s);
...

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

mbox

getchar()
```
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 again)?
  - No need to know which process to communicate with
  - Updating software on one side won't affect another side
Communication with Stream Sockets

Client

1. Create a socket
2. Connect to server
3. Send request
4. Receive response

Server

1. Create a socket
2. Bind to a port
3. Listen on the port
4. Accept connection
5. Receive request
6. Send response
7. Repeat

Establish connection
Request
Reply
Sockets API

- Create and close a socket
  - `sockid = socket(AF_INET, SOCK_STREAM, 0);`
  - `sockerr = close(sockid);`
- Bind a socket to a local address
  - `sockerr = bind(sockid, localaddr, addrlen);`
- Negotiate the connection
  - `listen(sockid, backlog);`
  - `accept(sockid, addr, addrlen);`
- Connect a socket to destination
  - `connect(sockid, destaddr, addrlen);`
- Message passing
  - `send(sockid, buf, size, flags);`
  - `recv(sockid, buf, size, flags);`
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
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;
}
```

- Initialize MPI environment
- Return my rank
- Last call to clean up
- Return # of processes
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
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, ierror)
  - Is there a matching message?

MPI_Irecv(…)

MPI_Wait(…)

MPI_Probe(…)

while (flag == FALSE) {
  More work
}

MPI_Irecv(…)

or MPI_recv(…)

Work to do

More work
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

RPC call

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

Server program
- Call
- Return
- Decode unmarshall
- Encode/marshall
- Receive
- Send

Client stub
Server stub
RPC runtime
RPC runtime

ClientId | RPCId | Call | Args
--- | --- | --- | ---

Reply | Results
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, 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
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
Summary

◆ Message passing
  ● Move data between processes
  ● Implicit synchronization
  ● Many API design alternatives (Socket, MPI)
  ● Indirections are helpful

◆ RPC
  ● Remote execution like local procedure calls
  ● With constraints in terms of passing data

◆ Issues
  ● Synchronous method is most common
  ● Asynchronous method provides overlapping
  ● Exception needs to be carefully handled