COS 318: Operating Systems

Message Passing

Kai Li
Computer Science Department
Princeton University

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

- Average: 28.58
- Median: 28.5
- See suggested solutions online
Midterm Grading

- **Problem 1 (Aaron)**
  - Main issue: did not realize that monitors can cause deadlocks

- **Problem 2 (Yida)**
  - Main issue: definition of turnaround time

- **Problem 3 (Scott)**
  - Main issue: not thinking about device driver

- **Problem 4 (Yida & Aaron)**
  - Main issue: making multi multilock atomic can avoid deadlock but inefficient

- **Problem 5 (Me)**
  - Main issue: not knowing how to program with Mesa-style monitor

Look for graders outside the classroom
Revisit Idiom of Mesa-style Monitor

Acquire(mutex);
while ("condition not true")
  wait(mutex, cond);
... 
Release(mutex);

Acquire(mutex);
... 
Signal(cond);
/* or Broadcast(cond);
Release(mutex);
Today’s Topics

- Message passing
- Implementation issues
Big Picture

Sender

Process

Receiver

Process
Send and Receive Primitives

Many ways to design the message passing API
Synchronous Message Passing

Move data between processes
- Sender: when data is ready, send it to the receiver process
- Receiver: when the data has arrived and when the receive process is ready to take the data, move the data

Synchronization
- Sender: signal the receiver process that a particular event happens
- Receiver: block until the event has happened

Sender

…
Send(R, buf, n);
…

Receiver

…
Recv(S, &buf, &n);
…
Example: Producer-Consumer

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?

Producer()
...
while (1) {
    produce item;
    recv(Consumer, &credit);
    send(Consumer, item);
}

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

Implementation Issues

- Buffering messages
- Direct vs. indirect
- Unidirectional vs. bidirectional
- Asynchronous vs. synchronous
- Event handler vs. receive
- Handle exceptions
Buffering Messages

- **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 mailbox as the abstraction
  - Allow many-to-many communication
  - Require open/close a mailbox

- Buffering
  - A buffer, its mutex and condition variables should be at the mailbox

- Message size
  - Not necessarily. One can break a large message into packets

- Mailbox vs. pipe
  - A mailbox allows many to many communication
  - A pipe implies one sender and one receiver
Synchronous vs. Asynchronous: Send

<table>
<thead>
<tr>
<th>Synchronous</th>
<th>Asynchronous</th>
</tr>
</thead>
<tbody>
<tr>
<td>Block on if resource is busy</td>
<td>Block if resource is busy</td>
</tr>
<tr>
<td>Initiate data transfer</td>
<td>Initiate data transfer and return</td>
</tr>
<tr>
<td>Block until data is out of its source memory</td>
<td>Completion</td>
</tr>
<tr>
<td>• Require applications to check status</td>
<td>• Notify or signal the application</td>
</tr>
</tbody>
</table>

```c
send( dest, type, msg)
```

```c
msg transfer resource

status = async_send( dest, type, msg )
```

```c
if !send_complete( status )
    wait for completion;
```

```c
use msg data structure;
```

```c
```
Synchronous vs. Asynchronous: Receive

- **Synchronous**
  - Return data if there is a message
  - msg transfer resource
    - `recv(src, type, msg)`

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

- **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**

```c
void func(char* msg) {
    ...
}
...
... hrecv(src, type, msg, func)
...
```

```
while(1) {
    recv(src, type, msg);
    func(msg);
}
```
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?

```c
while (1) {
    P(s);
    Acquire(m);
    convert ... Release(m);
};
```
Exception: Process Termination

- 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, 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
Example: Sockets API

- Abstraction for TCP and UDP
- 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);
Summary

- **Message passing**
  - Move data between processes
  - Implicit synchronization
  - API design is important

- **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