Inter-process Communication: An Overview

Dr. Yong Guan

Department of Electrical and Computer Engineering
& Information Assurance Center
Iowa State University
Outline for Today’s Talk

- Inter-process Communication: An Introduction
  - General Issues
  - IPC on the same host
Readings for Today’s Lecture

- Chapter 4 of “Distributed Systems: Principles and Paradigms”
- Chapter 14 & Chapter 15 of “Advanced Programming in the UNIX Environment”
- Other online resources
Inter-process Communication

- Primitives
- Message Passing: issues
- Communication Schemes
Interprocess Communication (IPC)

Primitives for interprocess communication

- **message passing**
  - the RISC among the IPC primitives

- **remote procedure call (RPC)**
  - process interaction at language level
  - type checking

- **transactions**
  - support for operations and their synchronization on shared objects
Message Passing

The primitives:

send expression_list to destination_identifier;
receive variable_list from source_identifier;

Variations:

guarded receive:

receive variable_list from source_id when B;

selective receive:

select
  receive var_list from source_id1;
| receive var_list from source_id2;
| receive var_list from source_id3;
end
Semantics of Message-Passing Primitives

- blocking vs. non-blocking
- buffered vs. unbuffered
- reliable vs. unreliable
- fixed-size vs. variable-size messages
- direct vs. indirect communication
## Blocking vs. Non-Blocking Primitives

<table>
<thead>
<tr>
<th></th>
<th>blocking</th>
<th>non-blocking</th>
</tr>
</thead>
<tbody>
<tr>
<td>send</td>
<td>Returns control to user only after message has been sent, or until acknowledgment has been received.</td>
<td>Returns control as soon as message queued or copied.</td>
</tr>
<tr>
<td>receive</td>
<td>Returns only after message has been received.</td>
<td>Signals willingness to receive message. Buffer is ready.</td>
</tr>
<tr>
<td>problems</td>
<td>• Reduces concurrency.</td>
<td>• Need buffering:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• still blocking</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• deadlocks!</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Tricky to program.</td>
</tr>
</tbody>
</table>
Buffered vs. Unbuffered Primitives

- **Asynchronous** `send` is never delayed
  - may get arbitrarily ahead of `receive`.
- However: messages need to be buffered.
- If no buffering available, operations become blocking, and processes are **synchronized** on operations: `rendezvous`.

```
invoke entry
    copy input parms
    rendezvous
    copy output parms
accept on entry

invoke entry
    copy input parms
    rendezvous
    copy output parms
accept on entry
```
Reliable vs. Unreliable Primitives

- Transmission problems:
  - corruption
  - loss
  - duplication
  - reordering
- Recovery mechanism: Where?
- Reliable transmission: acknowledgments

- At-least-one vs. exactly-one semantics

```
send                      send                      send                      send
<table>
<thead>
<tr>
<th>time-out</th>
<th>time-out</th>
<th>time-out</th>
<th>time-out</th>
</tr>
</thead>
<tbody>
<tr>
<td>receive</td>
<td>receive</td>
<td>receive</td>
<td>receive</td>
</tr>
</tbody>
</table>

inc(A)                     inc(A)                     inc(A)                     inc(A)
| A = 0                    | A = 0                    | A = 0                    | A = 0                    |
| A = 1                    | A = 1                    | A = 1                    | A = 1                    |
| A = 2                    | A = 1                    | dejá-vu!                 |
```

request table
Direct vs. Indirect Communication

- Direct communication:

  ```
  send(P, message)
  receive(Q, message)
  ```

- Variation thereof:

  ```
  send(P, message)
  receive(var, message)
  ```

```
C1
```

```
C2
```

```
server
```

```
S
```

```
send(S, msg1)
receive(&client_id, &msg)
```

```
send(S, msg2)
receive(&client_id, &msg)
```
Indirect communication:
- Treat communication paths as first-class objects.

Mailboxes:
Direct vs. Indirect Communication (cont.)

- Indirect communication (cont)
- Ports:
  - example: Accent (CMU)

- multiple senders
- only one receiver
- access to port is passed between processes in form of capabilities
Communication Schemes

- one-to-one unicast
- one-to-many multicast
- many-to-one
- many-to-many
Case Study: IPC on the Same Host

Ways of Inter-process Communication

- Signal
- Passing file descriptor between parent and child processes
- UNIX IPC
  - Pipes
  - FIFOs
  - Stream Pipes
  - Named Stream Pipes
  - Message Queues
  - Semaphores
  - Shared Memory
Case Study: IPC on the Same Host (cont.)

- Pipes
  - Half-duplex
  - Only used between processes that have a common ancestor, e.g., parent and child processes.
FIFO (also called named pipe)

- Half-duplex
- Can be used between unrelated processes (not necessarily between parent and child processes).
Case Study: IPC on the Same Host (cont.)

- **Message Queues**
  - A linked list of messages stored in the kernel and identified by msg queue id.
  - Not necessarily first-in first-out order
  - Can fetch messages based on type
  - Bi-directional

write mesg: msgsnd()

Receive mesg: msgrcv()
Case Study: IPC on the Same Host (cont.)

Semaphore

- Not really a form of IPC as pipe, FIFOs, and message queues
- A counter used to provide access to a shared data object for multiple processes

1. Test the semaphore that controls the resource
2. If the value is positive, the process can use the resource and the value of semaphore decrements by one.
3. If the value is 0, the process goes to sleep until the semaphore value is greater than 0.
Case Study: IPC on the Same Host (cont.)

- Shared Memory
  - Allow two or more processes to share a given region of memory.
  - Fastest IPC mechanism
  - Synchronization access
Stream pipes
- Allow passing open file descriptors between processes (parent and a child)
- Bi-directional

Similar to FIFO, we have named Stream Pipe
Questions?

Thanks and See you next time