Lecture 6: Interprocess Communication

CS178: Programming Parallel and Distributed Systems

February 12, 2001
Steven P. Reiss

I. Overview

A. So far we have covered

1. Multiple threads within a single process
2. Shared memory
3. Various types of synchronization

B. That is one way of getting performance

1. But it depends on specialized machines
2. And on local computation

C. Now we will move onto multiple processes

II. Multiple Process Programming

A. What do we mean here

1. Multiple processes on one or more machines
2. Examples

a) Multiplayer user game
b) Programming environment
c) Mail reader

B. What are the issues that arise here

1. Communications between the processes

a) How to do it
b) When to do it
c) Cost of doing it

2. Synchronization among the processes

a) Signaling one from another (notify, notifyAll)
b) Waiting for things to happen
c) Starting and stopping processes

3. **Detecting and dealing with failure**
    a) This is something new -- didn't occur in multithreaded
        (1) Perhaps it should have (exceptions in a thread)
    b) Types of failure
        (1) A process can fail (seg fault, ...)
        (2) A process can run very slowly (debugging, ...)
        (3) A communications link can fail

III. Interprocess Communication

A. UNIX was the first real impetus for IPC

1. **Model of doing things in small cooperating units**
2. **Seen in the set of Unix commands and in pipes**

B. Pipes

1. **Simple model of IPC**
2. **How they are used**
    a) From the shell
    b) From within an application
3. **Problems**
    a) Must be started from one process
    b) Limited to dealing with its subprocesses
    c) One-way communication per pipe
    d) Only works on the same machine

C. Sockets

1. **Basic socket is a 2-way pipe (can read/write)**
2. **Unix Domain sockets**
    a) Unix domain sockets are nodes in the file system
    b) Work only within a machine
    c) Similar to named pipes
    d) Faster (just buffered in the OS)
3. **IP Domain sockets**
    a) These are rooted at a host/port
    b) They work across machines

4. **UDP Sockets**
    a) Send messages unreliably
    b) Messages may be dropped, out of order, etc. (no bad status returned)
    c) Responsibility of receiver to accommodate
5. **TCP Sockets**
    a) Buffered message sending
    b) Reliablie -- in order and delivery ensured or failure returned
    c) Performance issues
6. **Master Sockets**
    a) Socket can listen for connections
    b) Other processes can connect to that socket
    c) That socket can accept connections
    d) An accepted connection results in a new socket in the server attached to the client socket

D. Shared Memory

1. **We'll cover this next week sometime**
2. **Problem is that it generally only works within a machine**

IV. Message Passing

A. Sockets provide the communications channel

1. **But what do you send over that channel**
2. **Data -- but what data**
3. **The easiest thing is to work in terms of messages**

B. What is a message

1. **Typically viewed as a command + arguments**
2. **Sent via sockets with header/length/body/trailer**
3. **Note that sockets are buffered (with fixed size)**
    a) Implementation problems
        (1) Reads can get more than one message
        (2) Reads can get partial messages

(3) Writes can send partial message

b) Implementing messages is non-trivial in general

C. Message Passing Architectures

1. Peer-to-peer

a) Similar to pipes

2. Client-server

a) Server sets up master socket

b) Clients connect to set up their own sockets

c) Server reads/writes from each of the clients as needed

d) Problem: how to find the server

3. Central message server

a) Clients send to central server

b) Central server resends to appropriate other clients

4. Central switchboard, local connections

D. Problems inherent to message passing

1. Accessing the central server

a) Single versus multiple servers

b) Known host/port versus locating host/port

2. What format to send data in

a) Text versus binary

b) Binary: byte order, size, etc. problems arise

c) How to send more complex objects (strings, structs, ...)

3. How to manage references

a) Pointers in one address space aren't valid in another

V. The FIELD Message Server

A. Problem definition

1. Want to build integrated programming environment for UNIX

a) Without recreating all the existing tools

b) Make it feel like the a single environment (VS)

2. Have the tools communicate via messages

a) Examples: debugger <-> editor

b) Messages can either be commands or informative

3. Want to allow arbitrary tools to run

a) Tools might not know who they are sending messages to

B. Solution

1. Provide a central message server

a) Tools register with message server when they start

(1) Provide patterns describing the mssages they are interested in

(2) Provide callbacks for handling those patterns

b) Tools send messages to the server

(1) It resends them to all other tools that have matching patterns

2. Message formats

a) All messages are text strings

(1) Patterns are string patterns

(2) Patterns indicate argument types which are decoded before the callback occurs.

b) Messages are stylized

(1) Who from/to; command; arguments; ...

(2) Standard form for argument types -- e.g. file names

3. Message types

a) Asynchronous messages (informative)

(1) Sent out; no wait for receiver, etc.; no reply, no ack

b) Synchronous messages (command)

(1) Sent out; sender gets first non-null reply

(2) This allows anonymous receivers

(3) Generally only one reply is relevant

c) Synchronous -- return value vs. callback

C. Overall architecture of FIELD

1. Notes:

a) Time to send a message

b) Ability to stream messages (Heapview)

2. **Identifying the proper message server at tool startup**
    a) Message groups
    b) By directory, by user choice, ...

D. Extensions

1. **Still using this architecture (TEA: Mint; Bloom: mince)**
2. **Extensions:**
    a) Use XML rather than strings
    b) Allow different types of synchronous messages
    c) Provide a high-level global registrar for message servers

VI. Remote Procedure Calls

A. If we are sending commands across sockets

1. **Why not think of these as procedure calls**
    a) They are issued in one process
    b) And executed in a second process
2. **This is the notion of remote procedure calls**

B. Remote objects

1. **As we move from procedural to OO programming**
2. **These should be method calls on remote objects**
3. **This is the basic idea in modern distributed message passing systems**

C. What are the issues that arise here

1. **Identifying the object to be called**
    a) Getting a handle on a remote object
    b) Classes versus interfaces
2. **Passing arguments to the routine/method**
    a) All the previous problems
    b) But objects are generally pointers, how to deal
3. **Synchronous vs. asynchronous calls**
4. **Handling failure**

D. These are the things we will cover next time and next week

1. **First with Java RMI, then with COM, CORBA, ...**