CIS 415: Operating Systems
IPC and RPC

Prof. Kevin Butler
Spring 2012
Today’s Lecture

• Inter-process communication
• Remote procedure calls
• Distributed RPC (MapReduce & Hadoop)

• Reminders
  ‣ Assignment 1 due April 23
  ‣ Project 1 due April 25
Process State

• What do we need to track about a process?
  ‣ how many processes?
  ‣ what’s the state of each of them?

• Process table: kernel data structure tracking processes on system

• Process control block: structure for tracking process context
Scheduling Processes

- Processes transition among *execution states*
Process States

• Running
  ‣ Running == in processor and in memory with all resources

• Ready
  ‣ Ready == in memory with all resources, waiting for dispatch

• Waiting
  ‣ Waiting == waiting for some event to occur
State Transitions

- **New Process ==> Ready**
  - Allocate resources
  - End of process queue

- **Ready ==> Running**
  - Head of process queue
  - Scheduled

- **Running ==> Ready**
  - Interrupt (Timer)
  - Back to end of process queue
State Transitions: Page Fault Handling

• Running ==> Waiting
  ‣ Page fault exception (similar for syscall or I/O interrupt)
  ‣ Wait for event

• Waiting ==> Ready
  ‣ Event has occurred (page fault serviced)
  ‣ End of process queue (or head?)

• Ready ==> Running
  ‣ As before…
State Transitions: Other Issues

• Priorities
  ‣ Can provide policy indicating which process should run next
    • More when we discuss scheduling…

• Yield
  ‣ System call to give up processor
  ‣ For a specific amount of time (sleep)

• Exit
  ‣ Terminating signal (Ctrl-C)
Process Control Block

- State of running process
- Linked list of process control information
Per Process Control Info

- Process state
  - Ready, running, waiting (momentarily)
- Links to other processes
  - Children
- Memory Management
  - Segments and page tables
- Resources
  - Open files
- And Much More…
/proc File System

- Linux and Solaris
  - `ls /proc`
  - A directory for each process
- Various process information
  - `/proc/<pid>/io` -- I/O statistics
  - `/proc/<pid>/environ` -- Environment variables (in binary)
  - `/proc/<pid>/stat` -- process status and info
Context Switch

• OS switches from one execution context to another
  ‣ One process to another process
  ‣ Interrupt handling
  ‣ Process to kernel (*mode transition*, not context switch)

• Current Process to New Process
  ‣ Save the state of the current process
    • *Process control block*: describes the state of the process in the CPU
  ‣ Load the saved context for the new process
    • Load the new process’s process control block into OS and registers
  ‣ Start the new process

• Does this differ if we are running an interrupt handler?
Context Switch

```
<table>
<thead>
<tr>
<th>process $P_0$</th>
<th>operating system</th>
<th>process $P_1$</th>
</tr>
</thead>
<tbody>
<tr>
<td>executing</td>
<td>interrupt or system call</td>
<td></td>
</tr>
<tr>
<td></td>
<td>save state into PCB$_0$</td>
<td></td>
</tr>
<tr>
<td></td>
<td>...</td>
<td></td>
</tr>
<tr>
<td>idle</td>
<td>reload state from PCB$_1$</td>
<td></td>
</tr>
<tr>
<td>executing</td>
<td>interrupt or system call</td>
<td></td>
</tr>
<tr>
<td></td>
<td>save state into PCB$_1$</td>
<td></td>
</tr>
<tr>
<td></td>
<td>...</td>
<td></td>
</tr>
<tr>
<td>idle</td>
<td>reload state from PCB$_0$</td>
<td></td>
</tr>
</tbody>
</table>
```

Context Switch

• No useful work is being done during a context switch
  ‣ Speed it up and limit system calls to things that can’t be done in user mode

• Hardware support
  ‣ Multiple register sets (Sun UltraSPARC)

• However, hardware optimization may conflict
  ‣ TLB flush is necessary
  ‣ Different virtual to physical mappings on different processes
Process Communication

• Processes need to share information

• Process model is a useful way to isolate running programs (separate resources, state, etc)
  ‣ Can simplify programs (no need to worry about other processes running)
  ‣ But processes don’t always work in isolation

• Discuss a variety of ways
  ‣ Doesn’t include regular files and signals
Process communication

• When is communication necessary?

• Lots of examples in operating systems
  ‣ threads with access to same data structures
  ‣ kernel/OS access to user process data
  ‣ processes sharing data via shared memory
  ‣ processes sharing data via system calls
  ‣ processes sharing data via file system

• And in general computer science
  ‣ DB transactions, P/L parallelism issues
IPC Mechanisms

• Two fundamental methods
• Shared memory
  ‣ Pipes, shared buffer
• Message Passing
  ‣ Mailboxes, Sockets
• Which one would you use and why?
Shared Memory

• Two processes share a memory region
  ‣ One writes: Producer
  ‣ One reads: Consumer

• Producer action
  ‣ While buffer not full
  ‣ Add stuff to buffer

• Consumer actions
  ‣ When stuff in buffer
  ‣ Read it

• Must manage where new stuff is in the buffer…
item nextProduced;

while (1) {
    while (((in + 1) % BUFFER_SIZE) == out)
        ; /* do nothing */
    buffer[in] = nextProduced;
    in = (in + 1) % BUFFER_SIZE;
}
Shared Memory -- Consumer

item nextConsumed;

while (1) {
    while (in == out)
        ; /* do nothing */
    nextConsumed = buffer[out];
    out = (out + 1) % BUFFER_SIZE;
}

Shared Memory

- Communicate by reading/writing from a specific memory location
  - Setup a shared memory region in your process
  - Permit others to attach to the shared memory region

- `shmget` -- create shared memory segment
  - Permissions (read and write)
  - Size
  - Returns an identifier for segment

- `shmat` -- attach to existing shared memory segment
  - Specify identifier
  - Location in local address space
  - Permissions (read and write)

- Also, operations for detach and control
Pipes

• **Producer-Consumer mechanism**
  ‣ `prog1 | prog2`
  ‣ The output of `prog1` becomes the input to `prog2`
  ‣ More precisely,
    • The standard output of `prog1` is connected to the standard input of `prog2`

• **OS sets up a fixed-size buffer**
  ‣ System calls: `pipe`, `dup`, `popen`

• **Producer**
  ‣ Write to buffer, if space available

• **Consumer**
  ‣ Read from buffer if data available
Pipes

• Buffer management
  ‣ A finite region of memory (array or linked-list)
  ‣ Wait to produce if no room
  ‣ Wait to consume if empty
  ‣ Produce and consume complete items

• Access to buffer
  ‣ Write adds to buffer (updates end of buffer)
  ‣ Reader removes stuff from buffer (updates start of buffer)
  ‣ Both are updating buffer state

• Issues
  ‣ What happens when end is reached (e.g., in finite array)?
  ‣ What happens if reading and writing are concurrent?
Shared Memory Machines

- SGI UV 1000 (Pitt SC)
  - 256 blades, each with 2 8-core Xeon processors
  - Each core has 8 GB RAM = 128 GB per blade

- Coherent shared-memory machine = all memory accessible to the machine
  - 32 TB of RAM

- Why? Certain problems hard to chunk up (eg graphs)
IPC -- Message Passing

• Establish communication link
  ‣ Producer sends on link
  ‣ Consumer receives on link

• IPC Operations
  ‣ Y: Send(X, message)
  ‣ X: Receive(Y, message)

• Issues
  ‣ What if X wants to receive from anyone?
  ‣ What if X and Y aren’t ready at same time?
  ‣ What size message can X receive?
  ‣ Can other processes receive the same message from Y?
IPC -- Synchronous Messaging

• Direct communication from one process to another

• Synchronous send
  ‣ Send(X, message)
  ‣ Producer must wait for the consumer to be ready to receive the message

• Synchronous receive
  ‣ Receive(id, message)
  ‣ Id could be X or anyone
  ‣ Wait for someone to deliver a message
  ‣ Allocate enough space to receive message

• Synchronous means that both have to be ready!
IPC -- Asynchronous Messaging

• Indirect communication from one process to another

• Asynchronous send
  ‣ Send(M, message)
  ‣ Producer sends message to a buffer M (like a mailbox)
  ‣ No waiting (modulo busy mailbox)

• Asynchronous receive
  ‣ Receive(M, message)
  ‣ Receive a message from a specific buffer (get your mail)
  ‣ No waiting (modulo busy mailbox)
  ‣ Allocate enough space to receive message

• Asynchronous means that you can send/receive when you’re ready
  ‣ What are some issues with the buffer?
IPC -- Sockets

• Communication end point
  ‣ Connect one socket to another (TCP/IP)
  ‣ Send/receive message to/from another socket (UDP/IP)

• Sockets are named by
  ‣ IP address (roughly, machine)
  ‣ Port number (service: ssh, http, etc.)

• Semantics
  ‣ Bidirectional link between a pair of sockets
  ‣ Messages: unstructured stream of bytes

• Connection between
  ‣ Processes on same machine (UNIX domain sockets)
  ‣ Processes on different machines (TCP or UDP sockets)
  ‣ User process and kernel (netlink sockets)
Files and file descriptors

• Remember open, read, write, and close?
  ‣ POSIX system calls for interacting with files
  ‣ open( ) returns a file descriptor
    • an integer that represents an open file
    • inside the OS, it’s an index into a table that keeps track of any state associated with your interactions, such as the file position
    • you pass the file descriptor into read, write, and close
Networks and sockets

• UNIX likes to make all I/O look like file I/O
  ‣ the good news is that you can use `read()` and `write()` to interact with remote computers over a network!
  ‣ just like with files....
    • your program can have multiple network channels open at once
    • you need to pass `read()` and `write()` a `file descriptor` to let the OS know which network channel you want to write to or read from
  ‣ a file descriptor used for network communications is a `socket`
Examples of sockets

- HTTP / SSL
- email (POP/IMAP)
- ssh
- telnet
IPC: Sockets

**Client X**
- 10.12.3.4:5544
- 10.12.3.4

**Client Y**
- 44.1.19.32:7113
- 44.1.19.32

**Web Server**
- 128.95.4.33:80
- 128.95.4.33

**Internet**

Connections:
- **Socket** 10.12.3.4:5544 to 44.1.19.32:7113
- **Socket** 128.95.4.33:80 to 128.95.4.33:80
**OS’s descriptor table**

<table>
<thead>
<tr>
<th>File descriptor</th>
<th>Type</th>
<th>Connected to?</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>pipe</td>
<td>stdin (console)</td>
</tr>
<tr>
<td>1</td>
<td>pipe</td>
<td>stdout (console)</td>
</tr>
<tr>
<td>2</td>
<td>pipe</td>
<td>stderr (console)</td>
</tr>
<tr>
<td>3</td>
<td>TCP socket</td>
<td>local: 128.95.4.33:80 remote: 44.1.19.32:7113</td>
</tr>
<tr>
<td>5</td>
<td>file</td>
<td>index.html</td>
</tr>
<tr>
<td>8</td>
<td>file</td>
<td>pic.png</td>
</tr>
<tr>
<td>9</td>
<td>TCP socket</td>
<td>local: 128.95.4.33:80 remote: 102.12.3.4:5544</td>
</tr>
</tbody>
</table>

128.95.4.33

**Web server**

- fd 5
- fd 8
- fd 9
- fd 3

Index.html

Pic.png

Internet

10.12.3.4:5544

44.1.19.32:7113
Types of sockets

• Stream sockets
  ‣ for connection-oriented, point-to-point, reliable bytestreams
    • uses TCP, SCTP, or other stream transports

• Datagram sockets
  ‣ for connection-less, one-to-many, unreliable packets
    • uses UDP or other packet transports

• Raw sockets
  ‣ for layer-3 communication (raw IP packet manipulation)
Stream sockets

• Typically used for client / server communications
  ‣ but also for other architectures, like peer-to-peer

• Client
  ‣ an application that establishes a connection to a server

• Server
  ‣ an application that receives connections from clients

1. establish connection
2. communicate
3. close connection
Datagram sockets

• Used less frequently than stream sockets
  ‣ they provide no flow control, ordering, or reliability

• Often used as a building block
  ‣ streaming media applications
  ‣ sometimes, DNS lookups
IPC -- Sockets

• Issues

• Communication semantics
  ‣ Reliable or not

• Naming
  ‣ How do we know a machine’s IP address? DNS
  ‣ How do we know a service’s port number?

• Protection
  ‣ Which ports can a process use?
  ‣ Who should you receive a message from?
    • Services are often open -- listen for any connection

• Performance
  ‣ How many copies are necessary?
  ‣ Data must be converted between various data types
Remote Procedure Calls

• IPC via a procedure call
  ‣ Looks like a “normal” procedure call
  ‣ However, the called procedure is run by another process
    • Maybe even on another machine

• RPC mechanism
  ‣ Client stub
  ‣ “Marshall” arguments
  ‣ Find destination for RPC
  ‣ Send call and marshalled arguments to destination (e.g., via socket)
  ‣ Server stub
  ‣ Unmarshalls arguments
  ‣ Calls actual procedure on server side
  ‣ Return results (marshall for return)
Remote Procedure Calls

- User calls kernel to send RPC message to procedure \( X \)
- Kernel sends message to matchmaker to find port number
- Kernel places port \( P \) in user RPC message
- Kernel sends RPC
- Kernel receives reply, passes it to user
- From: client
  - To: server
  - Port: matchmaker
  - Re: address for RPC \( X \)
- Matchmaker receives message, looks up answer
- Matchmaker replies to client with port \( P \)
- From: server
  - To: client
  - Port: kernel
  - Re: RPC \( X \)
  - Port: \( P \)
- Daemon listening to port \( P \) receives message
- Daemon processes request and processes send output
- From: RPC
  - Port: \( P \)
  - To: client
  - Port: kernel
  - <output>
Remote Procedure Calls

• Supported by systems
  ‣ Java RMI
  ‣ CORBA

• Issues
  ‣ Support to build client/server stubs and marshalling code
  ‣ Layer on existing mechanism (e.g., sockets)
  ‣ Remote party crashes… then what?

• Performance versus abstractions
  ‣ What if the two processes are on the same machine?
Remote Procedure Calls

• Marshalling

```
val = server.someMethod(A, B)

boolean someMethod(Object x, Object y)
{
  implementation of someMethod
  ...
}
```

Diagram:

- Client
  - `val = server.someMethod(A, B)`
  - Stub
    - `A, B, someMethod`
- Remote object
  - `boolean someMethod(Object x, Object y)`
  - Skeleton
    - `boolean return value`
public class RmiServer extends UnicastRemoteObject
    implements RmiServerIntf {
    public static final String MESSAGE = "Hello world";

    public RmiServer() throws RemoteException {
    }

    public String getMessage() {
        return MESSAGE;
    }

    public static void main(String args[]) {
        System.out.println("RMI server started");

        // Create and install a security manager
        if (System.getSecurityManager() == null) {
            System.setSecurityManager(new RMISecurityManager());
            System.out.println("Security manager installed.");
        } else {
            System.out.println("Security manager already exists.");
        }

        try {
            // Instantiate RmiServer
            RmiServer obj = new RmiServer();

            // Bind this object instance to the name "RmiServer"
            Naming.rebind("//localhost/RmiServer", obj);

            System.out.println("PeerServer bound in registry");
        } catch (Exception e) {
            System.err.println("RMI server exception:" + e);
            e.printStackTrace();
        }
    }
}
import java.rmi.Remote;
import java.rmi.RemoteException;

public interface RmiServerIntf extends Remote {
    public String getMessage() throws RemoteException;
}
import java.rmi.Naming;
import java.rmi.RemoteException;
import java.rmi.RMISecurityManager;

public class RmiClient {
    // "obj" is the reference of the remote object
    RmiServerIntf obj = null;

    public String getMessage() {
        try {
            obj = (RmiServerIntf) Naming.lookup("//localhost/RmiServer");
            return obj.getMessage();
        } catch (Exception e) {
            System.err.println("RmiClient exception: " + e);
            e.printStackTrace();
        }
        return e.getMessage();
    }

    public static void main(String args[]) {
        // Create and install a security manager
        if (System.getSecurityManager() == null) {
            System.setSecurityManager(new RMISecurityManager());
        }

        RmiClient cli = new RmiClient();

        System.out.println(cli.getMessage());
    }
}
MapReduce

• Distributed computing framework for working on large data sets on compute clusters

• Divide data into subset that are “mapped” to each node involved in computation

• Collect all subproblem answer and “reduce” to form the final output

• Uses:
  ‣ distributed sort and grep
  ‣ graph reversal and search
  ‣ statistical analysis and web analytics, bioinformatics
MapReduce: Word Count

Input: Deer Bear River, Car Car River

Split: Deer Bear River, Car Car River, Deer Car Bear

Map: Deer, 1; Bear, 1; River, 1; Car, 2; Car, 1; Deer, 1; Deer, 1; River, 1; River, 1

Shuffle: Bear, 1; Bear, 1; Car, 2; Car, 1; Deer, 1; Deer, 1; River, 1; River, 1

Reduce: Bear, 2; Car, 3; Deer, 2; Deer, 2; Bear, 2; Car, 3; Deer, 2; River, 2

Final: Bear, 2; Car, 3; Deer, 2; River, 2
MapReduce

```java
void map(String name, String document):

    // name: document name
    // document: document contents
    for each word w in document:
        EmitIntermediate(w, "1");

void reduce(String word, Iterator partialCounts):
    // word: a word
    // partialCounts: a list of aggregated partial counts
    int sum = 0;
    for each pc in partialCounts:
        sum += ParseInt(pc);
    Emit(word,AsString(sum));
```

Concepts come from functional programming (pay attention in CIS 425!)
package org.myorg;
import java.io.IOException;
import java.util.*;
import org.apache.hadoop.*;

public class WordCount {
    public static class Map extends MapReduceBase implements Mapper<LongWritable, Text, Text, IntWritable> {
        private final static IntWritable one = new IntWritable(1);
        private Text word = new Text();
        public void map(LongWritable key, Text value, OutputCollector<Text, IntWritable> output, Reporter reporter) throws IOException {
            String line = value.toString();
            StringTokenizer tokenizer = new StringTokenizer(line);
            while (tokenizer.hasMoreTokens()) {
                word.set(tokenizer.nextToken()); /* splits lines into words */
                output.collect(word, one);
            }
        }
    }
    public static class Reduce extends MapReduceBase implements Reducer<Text, IntWritable, Text, IntWritable> {
        public void reduce(Text key, Iterator<IntWritable> values, OutputCollector<Text, IntWritable> output, Reporter reporter) throws IOException {
            int sum = 0;
            while (values.hasNext()) {
                sum += values.next().get(); /* sums all the collected words */
            }
            output.collect(key, new IntWritable(sum));
        }
    }
}
public static void main(String[] args) throws Exception {
    JobConf conf = new JobConf(WordCount.class);
    conf.setJobName("wordcount");
    conf.setOutputKeyClass(Text.class);
    conf.setOutputValueClass(IntWritable.class);
    conf.setMapperClass(Map.class);
    conf.setCombinerClass(Reduce.class); /* collects all values together */
    conf.setReducerClass(Reduce.class);
    conf.setInputFormat(TextInputFormat.class);
    conf.setOutputFormat(TextOutputFormat.class);
    FileInputFormat.setInputPaths(conf, new Path(args[0]));
    FileOutputFormat.setOutputPath(conf, new Path(args[1]));
    JobClient.runJob(conf);
}

Scalable framework: works on single-node machine, “pseudo-distributed” (single machine, multiple processes), or fully distributed cluster (depending on how Hadoop installation is set up)
IPC Summary

• Lots of mechanisms
  ‣ Pipes
  ‣ Shared memory
  ‣ Sockets
  ‣ RPC

• Trade-offs
  ‣ Ease of use, functionality, flexibility, performance

• Implementation must maximize these
  ‣ Minimize copies (performance)
  ‣ Synchronous vs Asynchronous (ease of use, flexibility)
  ‣ Local vs Remote (functionality)
Summary

• Process
  ‣ Execution state of a program

• Process Creation
  ‣ fork and exec
  ‣ From binary representation

• Process Description
  ‣ Necessary to manage resources and context switch

• Process Scheduling
  ‣ Process states and transitions among them

• Interprocess Communication
  ‣ Ways for processes to interact (other than normal files)
• Next time: Threads