Administrivia

• Homework
  • HW3 is due this Friday June 2nd
Recap: OS roles

- Abstraction
- Virtualization
- Isolation and access control
- What about collaboration?
Inter-Process Communication

- **Exchange of data, coordinate operations, and request services** between two or more separate, independent processes.

- Operating systems provide **facilities/resources** for inter-process communications (IPC), such as **shared memory**, **semaphores**, **message queues**, etc.

- Similar concept exists for network communications (processes that reside on different physical hosts).
Inter-Process Communication (cont.)

- Two main types
  - Shared memory (discussed earlier, minimal kernel involvement)
  - Message queues (heavy kernel involvement)
Recap: shared memory

• Once page table mapping setup, no syscall or additional copying necessary
Ring buffer
Ring buffer (cont.)

- Read pointer
- Write pointer
- Can be lock free
Message queues (1)
Message queues (2)

- Two basic operations:
  - \texttt{send}(message) and \texttt{receive}(message)
  - Message contents can be anything mutually comprehensible
    - Data, remote procedure calls, executable code etc.
  - Usually contains standard fields, e.g.,
    - Destination process ID, sending process ID for any reply
    - Message length
    - Data type, data etc.
Message queues (3)

- UNIX
  - System V Message Queues
  - UNIX domain socket, local TCP/UDP socket
Network communication

• By in large similar to lPC in terms of the interface
  • `send`(message) and `receive`(message)
  • Message contents can be anything mutually comprehensible
    • Data, remote procedure calls, executable code etc.
  • Standard fields
    • Destination server IP, sending client IP
    • Message length
    • Data type, data etc.
Client-Server communication

1. Client connects to the server (routing)
2. Client sends a request to the server
3. Server performs some action
4. Server sends a response back
OS support for network

- OS includes implementations of network protocols
  - For example: TCP, UDP, ICMP, etc.
- **Abstraction**: socket
- **Operations**: `connect`, `send`, `receive`, `listen`, `accept`
- OS automatically construct the message and talk to the hardware, applications do not need to worry about the details
Messages: a raw abstraction

- Hand-coding messages gets tiresome
  - Need to worry about message formats
  - Have to pack and unpack data from messages
  - Servers have to decode and dispatch messages to handlers
  - Messages are often asynchronous
- Messages are not a very natural programming model (still heavily used nevertheless)
  - Think about web browsing
Remote procedure calls (1)

- Procedure calls are a more natural way to communicate
  - Every language supports them
  - Semantics are well-defined and understood
  - Natural for programmers to use

- **Idea**: have "servers" export a set of procedures that can be called by client programs
  - For example: `GetWeather(Date, City, State)`
  - Similar to module interfaces, class definitions, etc
Remote procedure calls (2)

• Clients just do a procedure call as if they were directly "linked" with the server
  • Under the covers, the procedure call is converted into a message exchange with the server
• Remote Procedure Call (RPC) is used both by operating systems and applications
  • NFS is implemented as a set of RPCs
  • DCOM, CORBA, Java RMI, etc., are all basically just RPC
Remote procedure calls (3)

• Lots of challenges
  • How do we hide the details from the programmer?
  • What are the semantics of parameter passing?
  • How do we (locate, connect) to servers?
  • How do we support heterogeneity (OS, arch, language)?
  • How do we make it perform well?
RPC model

• A server defines the server’s interface using an interface definition language (IDL)
  • The IDL specifies the names, parameters, and types for all client-callable server procedures
RPC model (cont.)

- A stub compiler reads the IDL and produces two stub procedures for each server procedure (client and server)
  - The server programmer implements the server procedures and links them with the server-side stubs
  - The client programmer implements the client program and links it with the client-side stubs
  - The stubs are responsible for managing all details of the remote communication between client and server
**RPC example**

- If the server were just a library, then Add would just be a procedure call.

```c
Client Program:
...
sum = server->Add(3,4);
...
```

```c
Server Interface:
int Add(int x, int y);
```

```c
Server Program:
int Add(int x, int, y) {
    return x + y;
}
```
**RPC example: call**

**Client Program:**
```c
sum = server->Add(3,4);
```

**Client Stub:**
```c
int Add(int x, int y) {
    Alloc message buffer;
    Mark as "Add" call;
    Store x, y into buffer;
    Send message;
}
```

**RPC Runtime:**
```c
Send message to server;
```

**Server Program:**
```c
int Add(int x, int, y) {};
```

**Server Stub:**
```c
AddStub(Message) {
    Remove x, y from buffer
    r = Add(x, y);
}
```

**RPC Runtime:**
```c
Receive message;
Dispatch, call Add_Strip;
```
RPC example: return

Client Program:
sum = server->Add(3,4);

Client Stub:
Int Add(int x, int y) {
    Create, send message;
    Remove r from reply;
    return r;
}

RPC Runtime:
Return reply to stub;

Server Program:
int Add(int x, int, y) {} 

Server Stub:
Add_Stub(Message) {
    Remove x, y from buffer
    r = Add(x, y);
    Store r in buffer;
}

RPC Runtime:
Send reply to client;
RPC summary

- RPC is the most common model for communication in distributed applications
  - "Cloaked" as DCOM, CORBA, Java RMI, etc.
- RPC is language support for distributed programming
- RPC relies upon a stub compiler to automatically generate client/server stubs from the IDL server descriptions
  - These stubs do the marshalling/unmarshalling, message sending/receiving/replying
Summary

• Communication enables collaboration between processes

• Two main types
  • Shared memory
    • Ring buffer
  • Message queue

• RPC
Next class ...

- Security