Outline for 9/12

• Objective:
  – Message Passing
• Administrative details:
  – Sign up for demo slots on Demo Scheduler
  – Check for demo location with grader
  – Submit details will be posted on the newsgroup
  – “Freeze” your code at midnight in a subdir you won’t touch until the demo (same as submitted)

Interprocess Communication – Messages (API-level)

• Assume no explicit sharing of data elements in the address spaces of processes wishing to cooperate/communicate.
• Essence of message-passing is copying (although implementations may avoid actual copies whenever possible).
• Problem-solving with messages - has a feel of more active involvement by participants.
Issues

• System calls for sending and receiving messages with the OS(s) acting as courier.
  – Variations on exact semantics of primitives and in the definition of what comprises a message.
• Naming - direct (to/from pids), indirect (to distinct objects - e.g., mailboxes, ports, sockets)
  – How do unrelated processes “find” each other?
• Buffering - capacity and blocking semantics.
• Guarantees - in-order delivery? no lost messages?

Send and Receive

A common and useful IPC abstraction: Generalized message send and receive primitives.

A messaging interface allows a process to send messages to a particular destination, e.g.,:

```c
thread->send(data);
currentThread->receive(data);
```

Messages for a given destination are stored in a queue pending delivery.

Send and receive are typically system calls, with message queues maintained by the kernel.
5 DP – Direct Send/Receive Message Passing Between Philosophers

Philosopher 0 (thinking)
Philosopher 1
Philosopher 2
Philosopher 3 (eating)
Philosopher 4

Fork please?

Umm. Oh yeah.
Philosopher 0 (thinking)

Philosopher 1

Philosopher 2

Philosopher 3 (eating)

Philosopher 4

Fork please?

I’ll ignore that request until I’m done
Client / Server

One common style of messaging is for a server process to provide services to client processes on demand, using request/response message exchanges.

```
Thread* client;

server-> end(request);
response = currentThread->receive();
...
```

```
while(systemActive) {
  currentThread->receive(request);
  handle the request
  requester->send(response);
}
```
Example: Time Service (kernel-based)

A time service could be packaged as a library, using time-related system calls provided by the underlying kernel.

Example: Time Service (via Messages)

The time service may be packaged as a server; clients pause or request time by sending a message to the server and waiting for a response. The clients trust the time server to provide the service correctly, just as they trust the kernel.
Client / Server with Threads

while (true) {
    SendRequest();
    RevReply();
}

Note the synergy with threads:
1. Client blocks until a reply is received.
   - Threads allow a client to issue concurrent requests.
2. Server waits for a request to arrive.
   - Threads allow a server to handle concurrent requests.

Hiding Message-Passing: RPC

The request/response communication is a basis for the \textit{remote procedure call} (RPC) model.

- Think of a server as a module (data + methods).
- Think of a request message as a \textit{call} to a server method.
  Each request carries an identifier for the desired method; the rest of the message contains the arguments.
- Think of the reply message as a \textit{return} from a server method.
  Each reply carries an identifier for the matching call; the rest of the message contains the result.

\textit{With a little extra glue, the messaging communication can be hidden to look “just like a procedure call” to one using it.}
Remote Procedure Call - RPC

- *Looks* like a nice familiar procedure call

\[ P_0 \]
\[ result = foo(param); \]

\[ P_1 \]
Receive

Remote Procedure Call - RPC

- *Looks* like a nice familiar procedure call

\[ P_0 \]
\[ result = foo(param); \]

\[ P_1 \]
Receive

Please do foo for \( P_0 \) with \( param \)

Blocked here
Remote Procedure Call - RPC

• *Looks* like a nice familiar procedure call

```
result = foo(param);
```

Please do foo for $P_0$ with param

```
Receive r = foo(param);
// actual call
```

<table>
<thead>
<tr>
<th>$P_0$</th>
<th>$P_1$</th>
</tr>
</thead>
<tbody>
<tr>
<td>blocked here</td>
<td>Receive r to $P_0$</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>$P_0$</th>
<th>$P_1$</th>
</tr>
</thead>
</table>
| result = foo(param); | Receive r = foo(param);
// actual call |

Reply
Remote Procedure Call - RPC

• *Looks* like a nice familiar procedure call

\[
\begin{align*}
\text{P}_0 & \quad \text{result} = \text{foo}(\text{param}); \\
\text{P}_1 & \quad \text{Receive} \\
& \quad r = \text{foo}(\text{param}); \\
& \quad // \text{ actual call} \\
& \quad \text{Reply} \\
\end{align*}
\]
5DP via RPC with Fork Manager

- *Looks* like a nice familiar procedure call

```
result = PickupForks(0);
```

Example: Time Service via RPC

RPC *stubs* are library routines that handle the details of interacting with the server/client. They may be generated by the system automatically from an abstract description of the service (e.g., a module header file).
RPC Issues

1. RPC is a syntactically friendly communication/interaction model built *above* basic messaging or other IPC primitives.
   
   RPC is a nice model, but it is constrained and not fully transparent; not everyone likes it, and it more-or-less assumes threads.

2. Complex systems may be structured in the usual way as interacting modules, with processes imposing *protection boundaries* crossed using RPC.
   
   Interacting processes/modules may fail independently (?).

3. The RPC paradigm extends easily to distributed systems, but a variety of optimizations may be employed in the local cases.
   
   *e.g.*, research systems and NT’s *IPC* pass arguments in shared memory

4. The RPC model also extends naturally to object-based systems and object-based distributed systems.
   
   *e.g.*, research systems, CORBA, *Java Remote Method Invocation*...there is an entire subculture out there

---

Practice Break

Larry, Moe, and Curly are planting seeds. Larry digs the holes. Moe then places a seed in each hole. Curly then fills the hole up.

There are several synchronization constraints:

- Moe cannot plant a seed unless at least one empty hole exists, but Moe does not care how far Larry gets ahead of Moe.
- Curly cannot fill a hole unless at least one hole exists in which Moe has planted a seed, but the hole has not yet been filled. Curly does not care how far Moe gets ahead of Curly.
- Curly *does* care that Larry does not get more than MAX holes ahead of Curly. Thus, if there are MAX unfilled holes, Larry has to wait.
- There is only one shovel with which both Larry and Curly need to dig and fill the holes, respectively.

Sketch out the pseudocode for the 3 threads which represent Larry, Curly, and Moe using whatever synchronization/communication method you like.
Naming Destinations for Messages: Ports

It may be useful for a given process to manage multiple communication endpoints - often called ports - with messages sent to ports rather than processes.

```
Port* svc;
    ....
    svc->send(request);
    replyport->receive(response);
    ....
while(systemActive) {
    svc->receive(request);
    ....
    replyport->send(response);
}
```

Advantages of Ports

1. Ports decouple IPC endpoints from processes and threads.
   A thread may send to a port without knowing the identity of the process/thread that receives on that port.
   Different threads may listen/service the same port, possibly at different times.

2. A thread may listen to multiple ports, separating the message streams designated for different ports.
   E.g., assign different ports to different objects or virtual services.

3. Ports are a convenient granularity to control message flow.
   E.g., Selectively enable/disable ports independently, or assign different priorities or access control to different ports.
Port Issues

1. *Asynchrony and notification.* How does a thread know when a message arrives on a port?
   - How to receive from multiple ports, without blocking on an idle port while incoming messages are queued on another?

2. *Naming and binding.* How do threads name the ports to send to or receive from (listen)?
   - How do threads find the names, e.g., for services they want to use?

3. *Protection and access control.*
   - How does the system know if a thread/process has a “right” to send to or listen on a particular port? E.g., how can we prevent untrusted programs from masquerading as a legitimate service?

Examples of Ports in Real Systems

1. Unix sockets and TCP/IP communication.
   - Common primitives/protocols for local messaging and network communication.
   - TCP/IP defines a fixed space of port numbers per node.
     - System calls to send/listen to a particular port.
   - Some ports are reserved to processes running with superuser (root) privilege.
     - Standard servers in /etc/services listen at well-known protected ports.

2. Mach supplies a rich set of port/messaging primitives.
   - Open ports (*port rights*) are kernel object handles.
   - Port rights may be passed in messages among processes.
     - The only way to get a send/receive right is for some other process to pass it to you! This is a system-wide basis for protection.
Sockets for Client-Server Message Passing

Server
1. Create a named socket syscalls:
   sfd = socket(…)
   bind (sfd, ptr,…)
2. Listen for clients listen(sfd, …)
4. Connection made and continue listening
   cfd=accept(sfd, …)
5. Exchange data write(cfd, …)
6. Done: close(cfd); close(sfd);

Client
3. Create unnamed socket & ask for connection syscalls:
   cfd=socket(…)
   err=connect(cfd, ptr, …)
5. Exchange data read(cfd, …)
6. Done: close(cfd);

Notification of Pending Messages

Communication-oriented systems face an important problem:

How does a client or server know what to do next?

- Servers in networks or server-structured systems might service many clients, possibly on different ports.
- The server must handle messages as they arrive, without blocking to receive on an empty port while others have pending messages.

Option 1: Use blocking primitives with lots of threads.
Leave the scheduling to the thread scheduler.

Option 2: Introduce nonblocking primitives or provide notifications or combined queueing of incoming messages.
A wide variety of mechanisms have been used: nonblocking polling, Unix select, Mach port groups, event queues, etc.
**Polling: Select**

A thread/process with multiple network connections or open files can initiate nonblocking I/O on all of them.

The Unix `select` system call supports such a polling model:
- pass a bitmask for which descriptors to query for readiness
- returns a bitmask of descriptors ready for reading/writing
- reads and/or writes on these descriptors will not block

---

**Immediate Notification: Upcalls**

**Problem:** what if an event requires a more “immediate” notification?
- What if a high-priority event occurs while we are executing the handler for a low-priority event?
- What about exceptions relating to the handling of an event?

We need some way to preemptively “break in” to the execution of a thread and notify it of events.

- `upcalls`
- `example`: NT Asynchronous Procedure Calls (APCs)
- `example`: Unix signals

Preemptive event handling raises synchronization issues similar to interrupt handling.
Advantages of Server “Isolation” Afforded by Message Passing

Like the kernel, the server is protected from its clients.

- Address space isolation is preserved, so the client cannot corrupt the server’s data.
- The only way a client can cause code to run in the server is to send a message. The server decides how to validate and interpret each message.
- The client is also protected from the server, although it must rely on it to correctly perform the service.

(Unlike the kernel, the server cannot access client memory.)

Protected servers may coordinate interactions among processes, manage system-critical data, or otherwise assume roles “typically” reserved for the operating system kernel.

Reconsidering the Kernel Interface and OS Structure

The kernel can be thought of as nothing more than a server; it is special only in that it runs in a protected hardware mode.

- Many of the services traditionally offered by the kernel can be supported outside of the kernel, in servers or in libraries.
- What features must be implemented in the kernel? Could we implement (say) the entire Unix interface as an application?
- Why would we want to do such a thing?

What are the advantages of supporting some OS feature in a server rather than directly in the kernel? What are the costs?

- How would we design a kernel interface that is powerful enough to implement multiple OS “personalities” as servers?

The kernel interface is not the programming interface!
Servers and Microkernels

A number of systems have been structured as collections of servers running above a minimal kernel ("microkernel").

- Microkernel provides, e.g., basic threads and scheduling, IPC, virtual address spaces, and device I/O primitives.
  - Kernel is hoped to be smaller, more reliable, and more secure.
  - Policies (e.g., security) may be implemented outside of the kernel.
- Operating system “personalities” (e.g., Unix or Windows) may be implemented as servers.
  - OS may have multiple personalities and policies, with new OS features and APIs added on-the-fly.
- The performance of server-structured systems is determined largely by the efficiency of the messaging primitives.

Microkernel with “User-Level” OS Server Processes
End-to-End Argument

• Application-level *correctness* requires checking at the endpoints to ensure that the message exchange accomplished its purpose
  – Application semantics involved
  – Notification of successful delivery (UPS tracking) is not as good as a direct response (thank you note) from the other end.

• Reliability guarantees in the message-passing subsystem provide *performance* benefits (short-circuiting corrective measures).
  – Re-transmitting packet may save re-transferring whole file.