Outline for Today

- Objective:
  - Continue with *message-passing* style interprocess communication (IPC)
  - Get specific with UNIX system calls
- Administrative details:

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.

Example: Time Service

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 can query 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

What if servicing request results in waiting?

Hiding Message-Passing: RPC

The request/response communication is a basis for the remote procedure call (RPC) model:

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

Remote Procedure Call - RPC

- *Looks* like a nice familiar procedure call

\[ \text{P}_0 \]
\[ \text{result} = \text{foo}(\text{param}); \]

\[ \text{P}_1 \]
\[ \text{Receive} \]

Remote Procedure Call - RPC

- *Looks* like a nice familiar procedure call

\[ \text{P}_0 \]
\[ \text{result} = \text{foo}(\text{param}); \]

\[ \text{P}_1 \]
\[ \text{Receive} \]
Remote Procedure Call - RPC

- Looks like a nice familiar procedure call

\[ \text{result} = \text{foo(param);} \]

Please do foo for P_0 with param

\[ \text{Receive} \]
\[ \text{r} = \text{foo(param);} \]

// actual call

Blocked here

\[ \text{returning} \]
\[ \text{r} \text{ to P_0} \]

Reply

\[ \text{result} = \text{foo(param);} \]

Receive
\[ \text{r} = \text{foo(param);} \]

// actual call
5DP via RPC with Fork Manager

- Looks like a nice familiar procedure call

```
Philosopher0
result = PickupForks(0);
```

Example: Time Service via RPC

What's Really Going On

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 does not support transactions.

2. Complex systems may be structured in the usual way as interacting modules, with processes imposing protection boundaries crossed using RPC.
   Interacting processes and 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
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.

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 a single 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 unauthorized 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 services in /etc/services listen at well-known (promoted) 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, …)
3. Connection made
   accept(sfd, …)
4. Exchange data
   read(cfd, …)
5. Exchange data
   write(cfd, …)
6. Done: close(cfd);
   close(sfd);

Client
1. Create unnamed socket syscalls:
   cfd = socket(…)
2. Ask for connection
   err=connect(cfd, ptr, …)
3. Create unnamed socket
   & ask for connection
   cfd=socket(…)
   err=connect(cfd, ptr, …)
4. Connection made
   and continue
   listen(cfd, …)
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 queuing 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.

**Unix Process Model**

- Simple and powerful primitives for process creation and initialization.
  - `fork` syscall creates a child process as (initially) a clone of the parent
  - parent program runs in child process to set it up for `exec`
  - child can `exit`, parent can `wait` for child to do so.
- Rich facilities for controlling processes by asynchronous signals.
  - notification of internal and/or external events to processes or groups
  - the look, feel, and power of interrupts and exceptions
  - default actions: stop process, kill process, dump core, no effect
  - user-level handlers

**Unix Process Control**

```
#include <unistd.h>

int pid;
int status = 0;
if (pid = fork()) {
   /* parent */
   //...
   pid = wait(&status);
} else {
   /* child */
   //...
   exit(status);
}
```

Parent uses `wait` to sleep until the child exits; wait returns child pid and status.
Wait variants allow wait on a specific child, or notification of stops and other signals.
Child process passes status back to parent on `exit`, to report success/failure.

The `fork` syscall returns a zero to the child and the child process ID to the parent.
`Fork` creates an exact copy of the parent process.

**Child Discipline**

- After a `fork`, the parent program (not process) has complete control over the behavior of its child process.
- The child inherits its execution environment from the parent...but the parent program can change it.
  - sets bindings of file descriptors with `open`, `close`, `dup`
  - `pipe` sets up data channels between processes
- Parent program may cause the child to execute a different program, by calling `exec` in the child context.
Fork/Exit/Wait Example

- Child process starts as clone of parent: increment refcounts on shared resources.
- Parent and child execute independently: memory states may diverge.
- On exit, release memory and decrement refcounts on shared resources.
- Child enters zombie state: process is dead and most resources are released, but process descriptor remains until parent reaps exit status via wait.
- Parent sleeps in wait until child stops or exits.

Exec, Execve, etc.

- Children should have lives of their own.
- Exec* "boots" the child with a different executable image.
  - parent program makes exec* syscall (in forked child context) to run a program in a new child process
  - exec* overlays child process with a new executable image
  - restarts in user mode at predetermined entry point (e.g., crt0)
  - no return to parent program (it's gone)
  - arguments and environment variables passed in memory
  - file descriptors etc. are unchanged

Join Scenarios

- Several cases must be considered for join (e.g., exit/wait).
  - What if the child exits before the parent does the wait?
    - "Zombie" process object holds child status and stats.
  - What if the parent continues to run but never joins?
    - Danger of filling up memory with zombie processes?
    - Parent might have specified it was not going to wait or that it would ignore its child's exit. Child status can be discarded.
  - What if the parent exits before the child?
    - Orphans become children of init (process 1).
    - What if the parent can't afford to get "stuck" on a join?
      - Asynchronous notification (we'll see an example later).
Unix Signals

- Signals notify processes of internal or external events.
  - the Unix software equivalent of interrupts/exceptions
  - only way to do something to a process “from the outside”
  - Unix systems define a small set of signal types
- Examples of signal generation:
  - keyboard ctrl-c and ctrl-z signal the foreground process
  - synchronous fault notifications, syscall errors
  - asynchronous notifications from other processes via kill
  - IPC events (SIGPIPE, SIGCHLD)
  - alarm notifications

Process Handling of Signals

1. Each signal type has a system-defined default action.
   - abort and dump core (SIGSEGV, SIGBUS, etc.)
   - ignore, stop, exit, continue
2. A process may choose to block (inhibit) or ignore some signal types.
3. The process may choose to catch some signal types by specifying a (user mode) handler procedure.
   - specify alternate signal stack for handler to run on
   - system passes interrupted context to handler
   - handler may munge and/or return to interrupted context

Predefined Signals (a Sampler)

<table>
<thead>
<tr>
<th>Name</th>
<th>Default Action</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SIGINT</td>
<td>quit</td>
<td>interrupt</td>
</tr>
<tr>
<td>SIGI</td>
<td>dump</td>
<td>illegal instruction</td>
</tr>
<tr>
<td>SIGKILL</td>
<td>quit</td>
<td>kill (can not be caught, blocked, or ignored)</td>
</tr>
<tr>
<td>SIGSEGV</td>
<td>dump</td>
<td>out of range addr</td>
</tr>
<tr>
<td>SIGALRM</td>
<td>quit</td>
<td>alarm clock</td>
</tr>
<tr>
<td>SIGCHLD</td>
<td>ignore</td>
<td>child status change</td>
</tr>
<tr>
<td>SIGTERM</td>
<td>quit</td>
<td>termination sent by kill</td>
</tr>
</tbody>
</table>
User’s View of Signals

```c
int alarmflag = 0;
alarmHandler () {
    printf("An alarm clock signal was received\n");
    alarmflag = 1;
}

main()
{
signal (SIGALRM, alarmHandler);
    alarm(3); printf("Alarm has been set\n");
    while (!alarmflag) pause ();
    printf("Back from alarm signal handler\n");
}
```

Sets up signal handler

Instructs kernel to send SIGALRM in 3 seconds

Suspends caller until signal

User’s View of Signals II

```c
main()
{
    int (*oldHandler) ();
    printf("I can be control-c'ed\n");
    sleep (3);
    oldHandler = signal (SIGINT, SIG_IGN);
    printf("I'm protected from control-c'\n");
    sleep(3);
    signal (SIGINT, oldHandler);
    printf("Back to normal\n");
    sleep(3); printf("bye\n");
}
```

Yet Another User’s View

```c
main(argc, argv)
int argc; char* argv[]
{
    int pid;
    signal (SIGCHLD, childhandler);
    pid = fork ();
    if (pid == 0) /*child*/{
        execvp (argv[2], &argv[2]);
    } else
    {
        sleep (5); printf("child too slow\n");
        kill (pid, SIGINT);
    }
}
```

childhandler()
{
    int childPid, childStatus;
    childPid = wait (&childStatus);
    printf("child done in time\n");
    exit;}

SIGCHLD sent by child on termination;
if SIG_IGN, dezombie

What does this do?

Files (& everything else)

• Descriptors are small unsigned integers used as handles to manipulate objects in the system, all of which resemble files.
• open with the name of a file returns a descriptor
• read and write, applied to a descriptor, operate at the current position of the file offset. lseek repositions it.
• Pipes are unnamed, unidirectional I/O stream created by pipe.
• Devices are special files, created by mknod, with ioctl used for parameters of specific device.
• Sockets introduce 3 forms of sendmsg and 3 forms of recvmsg syscalls.
File Descriptors

- Unix processes name I/O and IPC objects by integers known as file descriptors.
  - File descriptors 0, 1, and 2 are reserved by convention for standard input, standard output, and standard error.
  - "Conforming" Unix programs read input from stdin, write output to stdout, and errors to stderr by default.
  - Other descriptors are assigned by syscalls to open/create files, create pipes, or bind to devices or network sockets.
    - pipe, socket, open, creat
  - A common set of syscalls operate on open file descriptors independent of their underlying types.
    - read, write, dup, close

File Sharing Between Parent/Child

```c
main(int argc, char *argv[]) {
    for (;;) {
        if (read(fdrd, &c, 1) != 1) exit(0);
        write(fdwt, &c, 1);
    }
}
```

Sharing Open File Instances

```
parent
```

```
child
```

```
process objects
```

```
process file descriptors
```

```
system open file table
```

```
shared file
```

Bach
Producer/Consumer Pipes

Pipes support a simple form of parallelism with built-in flow control.

```c
char inbuffer[1024];
char outbuffer[1024];
while (inbytes != 0) {
    inbytes = read(stdin, inbuffer, 1024);
    outbytes = process data from inbuffer to outbuffer;
    write(stdout, outbuffer, outbytes);
}
```

Example:
```
sort <grades | grep Dan | mail mark
```
### Setting Up a Child in a Pipeline

```c
int setup_child(int rfd) {
    int pfd[2] = {0, 0}; /* pfd[0] is read, pfd[1] is write */
    int i, wfd;
    pipe(pfd); /* create right-hand pipe */
    wfd = pfd[1]; /* this child's write side */
    if (fork()) { /* parent */
        close(wfd); close(rfd);
    } else { /* child */
        close(pfd[0]); /* close far end of right pipe */
        close(i); /* stdin */
        close(i); /* stdout */
        dup(rfd); /* takes fd 0 */
        dup(wfd); /* takes fd 1 */
        close(rfd); close(wfd);
        ... /* execs nth stage of pipeline */
    }
    return(pfd[0]);
}
```

### Sockets for Client-Server Message Passing

**Server**
1. Create a named socket
   - syscalls: `socket(...)`, `bind(...)`, `listen(...)`
2. Listen for clients
3. Continue listening
4. Exchange data
5. Close
6. Done

**Client**
3. Create unnamed socket
   - syscalls: `socket(...)`, `connect(...)`
4. Exchange data
5. Close
6. Done

In a child process of server

### 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