Outline for Today

- Objective:
  - Continue with “Classic Problems”
  - Introduce message passing
  - Unix process-oriented system calls
- Administrative details:
  - Program 2 extension until Friday 11:59pm
  - Schedule of demos (sheet going around)
  - Making up missed classes???

No Plan?

- Each GROUP (one designated member) send me email indicating “yes”, “no”, “maybe” to the following proposed makeup sessions. Or propose your own.
  - Friday 9/29
    - 8-9 am
    - 9-10 am
    - 5:30 – 6:30 pm
    - Other time on this date?
  - Wednesday 10/4
    - 9-10 am
    - 5:30 – 6:30 pm
    - Other time on this date?
  - Other dates and times?

Only heard from 2 groups so far!!!

EventBarrier

EventBarrier::Wait()

If the EventBarrier is not in the signaled state, wait for it.

EventBarrier::Signal()

Signal the event, and wait for all waiters/arrivals to respond.

EventBarrier::Complete()

Notify EventBarrier that caller’s response to the event is complete.

Block until all threads have responded to the event.

The Moat Problem

- Travelers, knights, and troubadours arrive at the castle.
- The castle guard decides when to lower the bridge to allow the arrivals into the castle.
- If the bridge is down, new arrivals may enter immediately without waiting.
- The guard doesn’t raise the bridge if there are people on it.

This can be solved easily using EventBarrier.
The Moat Problem with EventBarrier

EventBarrier gate;
/* Called by knights etc. */
void EnterCastle() {
  gate.Wait(); /* wait for gate to open (if necessary) */
  CrossBridge();
  gate.Complete(); /* tell the guard it’s OK to close gate */
}

void GuardThread() {
  while (TRUE) {
    /* twiddle thumbs */
    /* watch for arriving travelers & decide when to open gate */
    WaitForOrderToOpenGate();
    gate.Signal(); /* open gate, wait for travelers to cross, close gate */
    /* gate is closed */
  }
}

Semaphore Solution with Writer Priority

int readCount = 0, writeCount = 0;
semaphore mutex1 = 1, mutex2 = 1;
semaphore readBlock = 1;
semaphore writePending = 1;
semaphore writeBlock = 1;

Reader()
{
  while (TRUE) {
    other stuff;
    P(writePending);
    P(readBlock);
    P(mutex1);
    readCount = readCount +1;
    if (readCount == 1)
      P(writeBlock);
    V(mutex1); V(readBlock); V(writePending);
    access resource
    P(mutex1);
    readCount = readCount -1;
    if(readCount == 0)
      V(writeBlock);
    V(mutex1);
  }
}

Writer()
{
  while (TRUE) {
    other stuff;
    P(mutex2);
    writeCount = writeCount +1;
    if (writeCount == 1)
      P(readBlock);
    V(mutex2);
    P(writeBlock);
    access resource
    P(mutex1);
    writeCount = writeCount -1;
    if(writeCount == 0)
      V(readBlock);
    V(mutex1);
  }
}

Template for Readers/Writers

Reader()
{
  while (true) {
    read
    startRead();
    while (true) {
      other stuff;
      P(writePending);
      P(readBlock);
      P(mutex1);
      readCount = readCount +1;
      if (readCount == 1)
        P(writeBlock);
      V(mutex1); V(readBlock); V(writePending);
      access resource
      P(mutex1);
      readCount = readCount -1;
      if(readCount == 0)
        V(writeBlock);
      V(mutex1);
    }
    endRead();
    startRead();
  }
}

Writer()
{
  while (true) {
    write
    startWrite();
    while (true) {
      other stuff;
      P(mutex2);
      writeCount = writeCount +1;
      if (writeCount == 1)
        P(readBlock);
      V(mutex2);
      P(writeBlock);
      access resource
      P(mutex1);
      writeCount = writeCount -1;
      if(writeCount == 0)
        V(readBlock);
      V(mutex1);
    }
    endWrite();
    startWrite();
  }
}
Reader()
while (TRUE) {
    other stuff;
P(writePending);
P(mutex1);
    readCount = readCount + 1;
if (readCount == 1)
P(writeBlock);
V(mutex2);
access resource;
P(mutex1);
    readCount = readCount - 1;
if (readCount == 0)
V(writeBlock);
V(mutex1);}
}

Writer()
while (TRUE) {
    other stuff;
P(mutex2);
writeCount = writeCount + 1;
if (writeCount == 1)
P(readBlock);
V(mutex2);
access resource;
P(mutex2);
    writeCount = writeCount - 1;
if (writeCount == 0)
V(readBlock);
V(mutex1);}
}

Assume the writePending semaphore was omitted. What would happen?

Guidelines for Choosing Lock Granularity

2. Limit lock overhead. Keep to a minimum the number of times mutexes are acquired and released.

   Note tradeoff between contention and lock overhead.
3. Use as few mutexes as possible, but no fewer.
   Choose lock scope carefully: if the operations on two different data structures can be separated, it may be more efficient to synchronize those structures with separate locks.
   Add new locks only as needed to reduce contention. “Correctness first, performance second!”
Interprocess Communication - Messages

- 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.

Send and Receive

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

- 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?

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?
Philosopher 0 (thinking)

Philosopher 1

Philosopher 2 (eating)

Philosopher 3 (eating)

Philosopher 4

Umm. Oh yeah.

Philosopher 0 (thinking)

Philosopher 1

Philosopher 2 (eating)

Philosopher 3 (eating)

Philosopher 4

Fork please?

Philosopher 0 (thinking)

Philosopher 1

Philosopher 2 (eating)

Philosopher 3 (eating)

Philosopher 4

Fork please?

I'll ignore that request until I'm done.

Philosopher 0 (thinking)

Philosopher 1

Philosopher 2 (eating)

Philosopher 3 (eating)

Philosopher 4

Fork please?

Fork please?
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.

Client

server

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 pause or request time by sending a message to the server and waiting for a response. The client trusts the time server to provide the service correctly, just as they trust the kernel.

Client / Server with Threads

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

*With a little extra glue, the messaging communication can be be hidden and made to look "just like a procedure call" to both the client and the server.*

**Remote Procedure Call - RPC**

- *Looks* like a nice familiar procedure call

```plaintext
P₀

result = foo(param);
```

Please do foo for P₀ with param.

```
P₁
Receive
```

```plaintext
P₁

result = foo(param);
```

// actual call

Please do foo for P₁ with param.

```
P₁
Receive
```

blocked here
Remote Procedure Call - RPC

• *Looks* like a nice familiar procedure call

```
P0
  result = foo(param);
  \[ \text{blocked here} \]
```

```
P1
  Receive r = foo(param);
  // actual call

  returning r to P0
  Reply
```

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 may or may not change thread
2. Complex systems may be structured in the exact way as interacting modules, with processes imposing protection boundaries crossed using RPC. Interacting processes/modules may fail independently/fail.
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 N's LPC 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 architecture out there

Naming Destinations for Messages: Ports

Advantages of 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.

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 unique 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 do threads find the names, e.g., for services they want to use?

2. Naming and binding. How do threads name the ports to send to or receive from (listen)?

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.
   - System calls to bind a port 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.

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, and others have pending messages.

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

Option 2: Introduce non-blocking 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 bitmap of which descriptors to query for readiness
- Returns a bitmap of descriptors ready for reading/writing
- Reads and/or writes on those descriptors will not block.

Select has tremendous scaling limitations in counting, tracing, and inspecting the bitmaps.
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 thought to be smaller, more reliable, and more secure.
- Operating system “personalities” (e.g., Unix or Windows) may be implemented as servers.
- Policies (e.g., security) may be implemented outside of the kernel.

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
- `fork` creates an exact copy of the parent process.
- Child process passes control to `exec`.
- Process manager takes control and child exit.
- Parent uses `wait` to sleep until the child exits; wait returns child pid and status.
- Child process passes status back to parent on `exit`, to report success/failure.
- Performance benefits (short-circuiting corrective measures).
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.

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., crtl)
  - no return to parent program (it’s gone)
  - arguments and environment variables passed in memory
  - file descriptors etc. are unchanged

Fork/Exit/Wait Example

Fork/Exec/Exit/Wait Example
Join Scenarios

- Several cases must be considered for join (e.g., exit/wait).
  - What if the child exits before the parent joins?
    - "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

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");
}
```

- Suspends caller until signal
- Sets up signal handler
- Instructs kernel to send SIGALRM in 3 seconds
- Suspends caller until signal
Yet Another User’s View
main(argc, argv)
{ int argc, char* argv[ ];
    pid = fork();
    if (pid == 0) /*child*/
        execvp (argv[2], & argv[2]);
    else
    { sleep(5);
        printf("child too slow 
");
        kill (pid, SIGINT);
    }
}

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

SIGCHLD sent
by child on termination;
if SIG_IGN, dezombie
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. seek 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 indepenent of their underlying types.
  • read, write, dup, close

File System Calls
• Files are named to by an integer file descriptor.
• Standard descriptors (0, 1, 2) for
  input, output, error messages (stdin, stdout, stderr).
  • File System Calls
  – read, write, dup, close
  – Standard descriptors (0, 1, 2) for
    standard input, output, error
  – Standard calls (write, dup, close, etc.)
main(int argc, char *argv[]) {
    char c;
    int fdrd, fdwt;
    if ((fdrd = open(argv[1], O_RDONLY)) == -1)
        exit(1);
    if ((fdwt = creat(argv[2], 0666)) == -1)
        exit(1);
    fork();
    for (;;) {
        if (read(fdrd, &c, 1) != 1)
            exit(0);
        write(fdwt, &c, 1);
    }
}

Producer/Consumer Pipes

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

while (inbytes != 0) {
    inbytes = read(stdin , inbuffer , 1024);
    outbytes = process data from inbuffer to outbuffer;
    writestdout , outbuffer , outbytes);
}

Setting Up Pipelines

```c
int pfd[2] = {0, 0}; /* pfd[0] is read, pfd[1] is write */
int in, out; /* pipeline entrance and exit */
pipe(pfd); /* create pipeline entrance */
out = pfd[0];
in = pfd[1];
/* loop to create a child and add it to the pipeline */
for (i = 1; i < procCount; i++) {
    out = setup_child(out);
    out = setup_child(out);
    if (pfd[2] = {0, 0}); /* pfd[2] is read, pfd[0] is write */
    if (in = out, out = pfd[0], out = pfd[1], out = pfd[2], out = pfd[1],
    out = setup_child(out);
}
```

e.g.: `sort < grades | grep Dan | mail nadia`
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(0); /*stdin*/ close(1); /*stdout*/
        dup(rfd); /*takesfd 0 */ dup(wfd); /*takes fd1 */
        close(rfd); close(wfd);
    }
    return(pfd[0]);
}

Sockets for Client-Server Message Passing

**Server**

1. Create a named socket sysscalls:
   `socket(...)`
2. Listen for clients `listen(...)`
3. Connection made and continue listening `cfd=accept(...)`
4. Exchange data `write(...)`
5. Exchange data `read(cfd ...)`
6. Done: close(cfd); close(sfd);

**Client**

1. Create unnamed socket & ask for connection sysscalls:
   `socket(...)`
2. Connect to server `connect(...)`
3. Exchange data `read(cfd ...)`
4. Exchange data `write(...)`
5. Close connection `close(...)`
6. Done: close(sfd);