Outline

• 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)
  – Check for changes on the website (in schedule, problem sets, etc)

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=recv(data);
```

Messaging combines synchronization and data transfer.

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

Fork please?

Philosopher 0 (thinking)

Philosopher 0 (thinking)

Philosopher 4

Philosopher 3 (eating)

Philosopher 1

Philosopher 2

Umm. Oh yeah.

Philosopher 4

Philosopher 3 (eating)

Philosopher 1

Philosopher 2
Philosopher 0 (thinking)

Philosopher 1

Philosopher 2

Philosopher 3 (eating)

Philosopher 4

Fork please?

Philosopher 0 (thinking)

Philosopher 1

Philosopher 2

Philosopher 3 (eating)

Philosopher 4

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

Philosopher 1

Philosopher 2

Philosopher 3 (eating)

Philosopher 4

Fork please?

Philosopher 0 (thinking)

Philosopher 1

Philosopher 2

Philosopher 3 (eating)

Philosopher 4

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.

```plaintext
Thread* client;
server->send(request);
response = currentThread->receive();
```

Example: Time Service (kernel-based)

A time service can 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 send or request time by sending a message to the server and waiting for a response. This clients send the time server to provide the service centrally, just as they trust the kernel.
Client / Server with Threads

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 contains 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.”

Remote Procedure Call - RPC
- Looks like a nice familiar procedure call

\[ P_0 \downarrow \]
\[ \text{result} = \text{foo}(\text{param}); \]

\[ P_1 \downarrow \]
\[ \text{Receive} \]
Remote Procedure Call - RPC

• Looks like a nice familiar procedure call

```c
P_0
  ↓
result = foo(param); → please do foo for P_0 with param
blocked here

P_1
  ↓
Receive
```

Receive blocked here

```
P_0
  ↓
result = foo(param); → please do foo for P_0 with param

P_1
  ↓
Receive
  ↓
r = foo(param);
  // actual call
```

Reply

```
P_0
  ↓
result = foo(param); → please do foo for P_0 with param
blocked here

P_1
  ↓
Receive
  ↓
r = foo(param);
  // actual call

returning r to P_0
```
Remote Procedure Call - RPC

- Looks like a nice familiar procedure call

```
P_0
      ↓
result = foo(param);
```

```
P_1
      ↓
Receive
      ↓
r = foo(param);
      ↓
// actual call
      ↓
Reply
```

5DP via RPC with Fork Manager

- Looks like a nice familiar procedure call

```
Philosopher_0
      ↓
result = PickupForks (0);
```

```
Fork Server
      ↓
Receive
      ↓
r = proc(param);
      ↓
// explicit queuing when necessary
      ↓
Reply
```
Example: Time Service via RPC

RPC Issues

1. RPC is a syntactically friendly communication interaction model built above basic messaging or other RPC primitives.
   RPC is a new model, but it is consistent and not fully transparent not everyone likes it, and it is not a one-size-fits-all model.
2. Complex systems may be structured in the usual way as interacting modules, with processes improving protection boundaries across using RPC. Internally, processes may fail independently.
3. The RPC model extends easily to distributed systems, but a variety of optimizations may be employed in the local case, e.g., regular systems and NT’s RPC pan arguments in shared memory.
4. The RPC model also extends naturally to object-based systems and object-based distributed systems, e.g., CORBA, Java Remote Method Invocation...there is an entire alphabet 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.
Larry ()
{while (TRUE) {
P(holes);
P(shovel);
dig;
V(shovel);
V(empty);
}
}

Curly ()
{while (TRUE) {
P(seeded);
P(shovel);
fill hole;
V(shovel);
V(holes);
}
}

Moe ()
{while (TRUE) {
P(empty);
seed empty hole;
V(seeded);
}
}

semaphore holes = MAX;
semaphore shovel = 1;
semaphore empty, seeded = 0;

Garden Monitor
Lock garden_gate;
Condition shovel_free, empty_holes, filled_holes, not_too_far_ahead;

Void allowdigging() {
garden_gate.Acquire();
while (holes >= MAX)
not_too_far_ahead.Wait(garden_gate);
while (!shovel)
shovel_free.Wait(garden_gate);
holes ++; shovel = FALSE;
garden_gate.Release();
}

Void donedigging() {
garden_gate.Acquire();
empty++; shovel=TRUE;
empty_holes.Signal(garden_gate);
shovel_free.Signal(garden_gate);
garden_gate.Release();
}

Void allowseeding() {
garden_gate.Acquire();
while (empty == 0)
empty_holes.Wait(garden_gate);
garden_gate.Release();
}

Void doneseeding() {
garden_gate.Acquire();
empty--; filled++;
filled_holes.Signal(garden_gate);
garden_gate.Release();
}

Garden Monitor
Lock garden_gate;
Condition shovel_free, empty_holes, filled_holes, not_too_far_ahead;

Void allowdigging() {
garden_gate.Acquire();
while (holes >= MAX)
not_too_far_ahead.Wait(garden_gate);
while (!shovel)
shovel_free.Wait(garden_gate);
holes ++; shovel = FALSE;
garden_gate.Release();
}

Void donedigging() {
garden_gate.Acquire();
empty++; shovel=TRUE;
empty_holes.Signal(garden_gate);
shovel_free.Signal(garden_gate);
garden_gate.Release();
}

Void allowseeding() {
garden_gate.Acquire();
while (empty == 0)
empty_holes.Wait(garden_gate);
garden_gate.Release();
}

Void doneseeding() {
garden_gate.Acquire();
empty--; filled++;
fill hole;
empty_holes.Signal(garden_gate);
shovel_free.Signal(garden_gate);
garden_gate.Release();
}
Void allowfilling() {
    garden_gate.Acquire();
    while (filled==0)
        filled_holes.Wait(garden_gate);
    while (!shovel)
        shovel_free.Wait(garden_gate);
    filled--; shovel = FALSE;
    garden_gate.Release();}

Void donefilling() {
    garden_gate.Acquire();
    holes--; shovel=TRUE;
    not_too_far_ahead.Signal(garden_gate);
    shovel_free.Signal(garden_gate);
    garden_gate.Release();}

Garden Server
(using threads and monitor)

While (1){
    ReceiveRequest(msg);
    thread->fork(handler, msg);
}

Void handler (msg){
    garden->msg.function();
    SendReply(msg.requester);
    exit();
}

Larry (){
    while (TRUE) {
        garden_server->RPC(allowdigging);
        dig;
        garden_server->RPC(donedigging);
    }
}

Moe (){
    while(TRUE){
        garden_server->RPC(allowseeding);
        seed empty hole;
        garden_server->RPC(doneseeding);
    }
}

Curly (){
    while(TRUE){
        garden_server-> RPC(allowfilling);
        fill hole;
        garden_server->RPC(donefilling);
    }
}