Lecture 5 (part2) : “Interprocess communication”

- reasons for process cooperation
- types of message passing
- direct and indirect message passing
- buffering
- client/server communication
- remote procedure calls
- remote method invocation

Cooperating processes

- Processes can cooperate with each other to accomplish a single task.
- Cooperating processes can:
  - Improve performance by overlapping activities or performing work in parallel
  - Enable an application to achieve a better program structure as a set of cooperating processes, where each is smaller than a single monolithic program
  - Easily share information
- Issues:
  - How do the processes communicate?
  - How do the processes share data?

Message passing

- syntax:
  - send(*destination-process*, *message*)
  - receive(*source-process*, *message*)
- the communicating processes can be equal (*peer to peer*) or some process can solicit certain services from another (*client-server*)
- process can:
  - block until the message is sent/received (*blocking*) - safer, easier to think about, slower
  - proceed immediately (*non-blocking*) - faster, harder to code, riskier, requires additional OS support
- process can:
  - block until the message it sent is received (*synchronous*) - easier to code, deadlock prone, slower
  - proceed without receipt confirmation (*asynchronous*) - faster, requires separate message confirming receipt
- process knows its party (*direct*) or does not know it as long as the service it requests are performed (*indirect*)

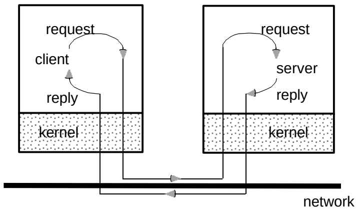
Direct vs. indirect communication

- Direct communication — explicitly name the process you’re communicating with
    - send(*destination-process*, *message*)
    - receive(*source-process*, *message*)
  - Link is associated with exactly two processes
    - Between any two processes, there exists at most one link
    - The link may be unidirectional, but is usually bidirectional
- Indirect communication — communicate using mailboxes (ports) owned by receiver
    - send(*mailbox*, *message*)
    - receive(*mailbox*, *message*)
  - Link is associated with two or more processes that share a mailbox
    - Between any two processes, there may be a number of links
    - The link may be either unidirectional or bidirectional

Buffering

- Link may have some capacity that determines the number of message that can be temporarily queued in it
- Zero capacity: (queue of length 0)
  - No messages wait
  - Sender must wait until receiver receives the message — this synchronization to exchange data is called a *rendezvous*
- Bounded capacity: (queue of length *n*)
  - If receiver’s queue is not full, new message is put on queue, and sender can continue executing immediately
  - If queue is full, sender must block until space is available in the queue
- Unbounded capacity: (infinite queue)
  - Sender can always continue

Client-server communication using message passing



Server = process (or collection of processes) that provides a *service*

Example: name service, file service, web service

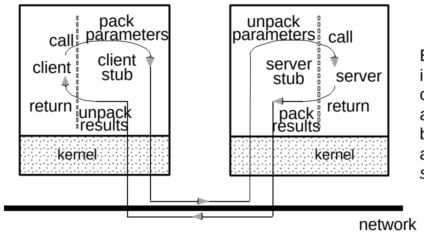
Client — process that uses the service

- Request / reply protocol:
  - Client sends request message to server, asking it to perform some service
  - Server performs service, sends reply message containing results or error code

Remote procedure call (RPC)

- RPC idea:
  - hide message-passing I/O from the programmer
  - look (almost) like a procedure call — but client invokes a procedure on a server
- RPC invocation (high-level view):
  - calling process (client) is suspended
  - parameters of procedure are passed across network to called process (server)
  - server executes procedure
  - return parameters are sent back across network
  - calling process resumes

Client / Server Model using Remote Procedure Calls (RPCs)



Each RPC invocation by a client process calls a *client stub*, which builds a message and sends it to a *server stub*

- The server stub uses the message to generate a local procedure call to the server
- If the local procedure call returns a value, the server stub builds a message and sends it to the client stub, which receives it and returns the result(s) to the client

RPC invocation step by step

1. Client procedure calls the client stub
2. Client stub packs parameters into message and traps to the kernel
3. Kernel sends message to remote kernel
4. Remote kernel gives message to server stub
5. Server stub unpacks parameters and calls server
6. Server executes procedure and returns results to server stub
7. Server stub packs result(s) in message and traps to kernel
8. Remote kernel sends message to local kernel
9. Local kernel gives message to client stub
10. Client stub unpacks result(s) and returns them to client

Generating stubs

- C/C++ may not be descriptive enough to allow stubs to be generated automatically

```
typedef struct {
        double item1;
        int item2;
        char *annotation;
} tuple;
```

```
char add(int key, tuple value);
char remove(int key, tuple value);
int query(int key, int number, tuple values[ ]);
```

  - Which are in, in-out, and out parameters?
  - Exactly what size are parameters (e.g., integers, arrays)?
  - What does it mean to pass a pointer?
- Using OSF’s DCE Interface Definition Language (IDL) to specify procedure signatures for stub generation:

```
inerface db
{
typedef struct {
        double item1;
        long item2;
        [string, ptr]
          ISO_LATIN_1
          *annotation;
} tuple;
```

```
boolean add (
  [in]  long  key,
  [in]  tuple value
);

boolean remove (
  [in]  long  key,
  [in]  tuple value
);
```

```
long query (
  [in]  long  key,
  [in]  long  number,
  [out, size_is(number)]
        tuple values[ ]
);
```

Remote Method Invocation

- Java allows a process to invoke a method of a remote object
- done transparently to the application programmers
- unlike RPC
  - can invoke methods on remote objects
  - can pass objects as parameters
- example: client executes statement

  `boolean val = Server.someMethod(par1, par2);`
- *stub* – proxy for the remote object on the client that marshals parameters into a *parcel* consisting of name of method to be invoked + serialized object parameters, unmarshals the return value
- *skeleton* – server side “stub”
- local objects are passed by copy-return, remote objects are passed by reference (which allows the server to invoke remote objects via RMI)