CS 61A Lecture Notes Week 10

Topic: Client/server paradigm, Concurrency

Reading: Abelson & Sussman, Section 3.4

• Client/server programming paradigm

Before networks, most programs ran on a single computer. Today it's common for programs to involve cooperation between computers. The usual reason is that you want to run a program on your computer that uses data located elsewhere. A common example is using a browser on your computer to read a web page stored somewhere else.

To make this cooperation possible, *two* programs are actually required: the *client* program on your personal computer and the *server* program on the remote computer. Sometimes the client and the server are written by a single group, but often someone publishes a *standard* document that allows any client to work with any server that follows the same standard. For example, you can use Mozilla, Netscape, or Internet Explorer to read most web pages, because they all follow standards set by the World Wide Web Consortium.

For this course we provide a sample client/server system, implementing a simple Instant Message protocol. The files are available in

```
~cs61a/lib/im-client.scm
~cs61a/lib/im-server.scm
```

To use them, you must first start a server. Load `im-server.scm` and call the procedure `im-server-start`; it will print the IP (Internet Protocol) address of the machine you're using, along with another number, the *port* assigned to the server. Clients will use these numbers to connect to the server. Port numbers are important because there might be more than one server program running on the same computer, and also to keep track of connections from more than one client.

(Why don't you need these numbers when using "real" network software? You don't need to know the IP address because your client software knows how to connect to *nameservers* to translate the host names you give it into addresses. And most client/server protocols use fixed, *registered* port numbers that are built into the software. For example, web browsers use port 80, while the `ssh` protocol you may use to connect to your class account from home uses port 22. But our sample client/server protocol doesn't have a registered port number, so the operating system assigns a port to the server when you start it.)

To connect to the server, load `im-client.scm` and call `im-enroll` with the IP address and port number as arguments. (Details are in this week's lab assignment.) Then use the `im` procedure to send a message to other people connected to the same server.

This simple implementation uses the Scheme interpreter as its user interface; you send messages by typing Scheme expressions. Commercial Instant Message clients have a more ornate user interface, that accept mouse clicks in windows listing other clients to specify the recipient of a message. But our version is realistic in the way it uses the network; the IM client on your home computer connects to a particular port on a particular server in order to use the facility. (The only difference is that a large commercial IM system will have more than one server; your client connects to the one nearest you, and the servers send messages among themselves to give the illusion of one big server to which everyone is connected.)

In the news these days, client/server protocols are sometimes contrasted with another approach called *peer-to-peer* networking, such as file-sharing systems like Napster and Kazaa. The distinction is social rather than strictly technical. In each individual transaction using a peer-to-peer protocol, one machine is acting as a server and the other as a client. What makes it peer-to-peer networking is that any machine using the protocol can play either role, unlike the more usual commercial networking idea in which rich companies operate servers and ordinary people operate clients.

Internet primitives in STk

STk defines sockets, on systems which support them, as first class objects. Sockets permit processes to communicate even if they are on different machines. Sockets are useful for creating client-server applications.

`(make-client-socket hostname port-number)`

`make-client-socket` returns a new socket object. This socket establishes a link between the running application listening on port `port-number` of `hostname`.

`(socket?  socket)`

Returns `#t` if `socket` is a socket, otherwise returns `#f`.

`(socket-host-name socket)`

Returns a string which contains the name of the distant host attached to `socket`. If `socket` was created with `make-client-socket`, this procedure returns the official name of the distant machine used for connection. If `socket` was created with `make-server-socket`, this function returns the official name of the client connected to the socket. If no client has yet used the socket, this function returns `#f`.

`(socket-host-address socket)`

Returns a string which contains the IP number of the distant host attached to `socket`. If `socket` was created with `make-client-socket`, this procedure returns the IP number of the distant machine used for connection. If `socket` was created with `make-server-socket`, this function returns the address of the client connected to the socket. If no client has yet used the socket, this function returns `#f`.

`(socket-local-address socket)`

Returns a string which contains the IP number of the local host attached to `socket`.

`(socket-port-number socket)`

Returns the integer number of the port used for `socket`.

`(socket-input socket)`
`(socket-output socket)`

Returns the port associated for reading or writing with the program connected with `socket`. If no connection has been established, these functions return `#f`. The following example shows how to make a client socket. Here we create a socket on port 13 of the machine `kaolin.unice.fr`. [Port 13 is generally used for testing: making a connection to it returns the distant system's idea of the time of day.]

```
(let ((s (make-client-socket "kaolin.unice.fr" 13)))
  (format #t "Time is: ~A\n" (read-line (socket-input s)))
  (socket-shutdown s))
```

`(make-server-socket)`
`(make-server-socket port-number)`

`make-server-socket` returns a new socket object. If `port-number` is specified, the socket listens on the specified port; otherwise, the communication port is chosen by the system.

(socket-accept-connection socket)

`socket-accept-connection` waits for a client connection on the given socket. If no client is already waiting for a connection, this procedure blocks its caller; otherwise, the first connection request on the queue of pending connections is connected to socket. This procedure must be called on a server socket created with `make-server-socket`. The return value of `socket-accept-connection` is undefined. The following example is a simple server which waits for a connection on the port 1234. Once the connection with the distant program is established, we read a line on the input port associated to the socket and we write the length of this line on its output port. [Under Unix, you can simply connect to listening socket with the telnet command. With the given example, this can be achieved by typing the command

```
telnet localhost 1234
```

in a shell window.]

```
(let ((s (make-server-socket 1234)))
  (socket-accept-connection s)
  (let ((l (read-line (socket-input s))))
    (format (socket-output s) "Length is: ~A\n" (string-length l))
    (flush (socket-output s)))
  (socket-shutdown s))
```

(socket-shutdown socket)
(socket-shutdown socket close)

`Socket-shutdown` shuts down the connection associated to `socket`. `Close` is a boolean; it indicates if the socket must be closed or not, when the connection is destroyed. Closing the socket forbids further connections on the same port with the `socket-accept-connection` procedure. Omitting a value for `close` implies closing the socket. The return value of `socket-shutdown` is undefined. The following example shows a simple server: when there is a new connection on the port number 1234, the server displays the first line sent to it by the client, discards the others and goes back waiting for further client connections.

```
(let ((s (make-server-socket 1234)))
  (let loop ()
    (socket-accept-connections)
    (format #t "I've read: ~A\n" (read-line (socket-input s)))
    (socket-shutdown s #f)
    (loop)))
```

(socket-down? socket)

Returns `#t` if socket has been previously closed with socket-shutdown. It returns `#f` otherwise.

(socket-dup socket)

Returns a copy of `socket`. The original and the copy socket can be used interchangeably. However, if a new connection is accepted on one socket, the characters exchanged on this socket are not visible on the other socket. Duplicating a socket is useful when a server must accept multiple simultaneous connections. The following example creates a server listening on port 1234. This server is duplicated and, once two clients are present, a message is sent on both connections.

```
(define s1 (make-server-socket 1234))
(define s2 (socket-dup s1))
(socket-accept-connection s1)
(socket-accept-connection s2) ;; blocks until two clients are present
(display "Hello,\n" (socket-output s1))
(display "world\n" (socket-output s2))
(flush (socket-output s1))
(flush (socket-output s2))
```

(when-socket-ready socket handler)
(when-socket-ready socket)

Defines a handler for `socket`. The handler is a thunk which is executed when a connection is available on socket. If the special value `#f` is provided as `handler`, the current handler for socket is deleted. If a handler is provided, the value returned by `when-socket-ready` is undefined. Otherwise, it returns the handler currently associated to `socket`. This procedure, in conjunction with `socket-dup`, permits building multiple-client servers which work asynchronously. Such a server is shown below.

```
(define p (make-server-socket 1234))
(when-socket-ready p
  (let ((count 0))
    (lambda ()
      (set! count (+ count 1))
      (register-connection (socket-dup p) count))))
(define register-connection
  (let ((sockets '()))
    (lambda (s cnt)
      ;; Accept connection
      (socket-accept-connection s)
      ;; Save socket somewhere to avoid GC problems
      (set! sockets (cons s sockets))
      ;; Create a handler for reading inputs from this new connection
      (let ((in (socket-input s))
            (out (socket-output s)))
        (when-port-readable in
          (lambda ()
            (let ((l (read-line in)))
              (if (eof-object? l)
                  ;; delete current handler
                  (when-port-readable in #f)
                  ;; Just write the line read on the socket
                  (begin (format out "On #~A --> ~A\n" cnt l)
                         (flush out)))))))))
```

• Concurrency

To work with the ideas in this section you should first

```
(load "~cs61a/lib/concurrent.scm")
```

in order to get the necessary Scheme extensions.

Parallelism

Many things we take for granted in ordinary programming become problematic when there is any kind of parallelism involved. These situations include

- multiple processors (hardware) sharing data
- software multithreading (simulated parallelism)
- operating system input/output device handlers

This is the most important topic in CS 162, the operating systems course; here in 61A we give only a brief introduction, in the hope that when you see this topic for the second time it'll be clearer as a result.

To see in simple terms what the problem is, think about the Scheme expression

```
(set! x (+ x 1))
```

As you'll learn in more detail in 61C, Scheme translates this into a sequence of instructions to your computer. The details depend on the particular computer model, but it'll be something like this:

```
   lw   $8, x          ; Load a Word from memory location x
                       ; into processor register number 8.
   addi $8, 1          ; Add the Immediate value 1 to the register.
   sw   $8, x          ; Store the Word from register 8 back
                       ; into memory location x.
```

Ordinarily we would expect this sequence of instructions to have the desired effect. If the value of x was 100 before these instructions, it should be 101 after them.

But imagine that this sequence of three instructions can be interrupted by other events that come in the middle. To be specific, let's suppose that someone else is also trying to add 1 to x's value. Now we might have this sequence:

```
my process                             other process
----------                             -------------

lw   $8, x   [value is 100]
addi $8, 1   [value is 101]
                                       lw   $9, x   [value is 100]
                                       addi $9, 1   [value is 101]
                                       sw   $9, x   [stores 101]
sw   $8, x   [stores 101]
```

The ultimate value of x will be 101, instead of the correct 102.

The general idea we need to solve this problem is the *critical section*, which means a sequence of instructions that mustn't be interrupted. The three instructions starting with the load and ending with the store are a critical section.

Actually, we don't have to say that these instructions can't be interrupted; the only condition we must enforce is that they can't be interrupted by another process that uses the variable `x`. It's okay if another process wants to add 1 to `y` meanwhile. So we'd like to be able to say something like

```
reserve x
lw      $8, x
addi    $8, 1
sw      $8, x
release x
```

Levels of Abstraction

Computers don't really have instructions quite like `reserve` and `release`, but we'll see that they do provide similar mechanisms. A typical programming environment includes concurrency control mechanisms at three levels of abstraction:

SICP name	What's protected	Provided by
serializer	high level abstraction (procedure, object, ...)	programming language
mutex	critical section	operating system
test-and-set!	one atomic state transition	hardware

The serializer and the mutex are, in SICP, abstract data types. There is a constructor `make-serializer` that's implemented using a mutex, and a constructor `make-mutex` that's implemented using `test-and-set!`, which is a (simulated, in our case) hardware instruction.

Serializers

For now, let's look at how this idea can be expressed at a higher level of abstraction, in a Scheme program.

```
(define x-protector (make-serializer))

(define protected-increment-x (x-protector (lambda () (set! x (+ x 1)))))

> x
100
> (protected-increment-x)
> x
101
```

We introduce an abstraction called a *serializer.* This is a procedure that takes as its argument another procedure (call it `proc`). The serializer returns a new procedure (call it `protected-proc`). When invoked, `protected-proc` invokes `proc`, but only if the *same* serializer is not already in use by another protected procedure. `Proc` can have any number of arguments, and `protected-proc` will take the same arguments and return the same value.

There can be many different serializers, all in operation at once, but each one can't be doing two things at once. So if we say

```
(define x-protector (make-serializer))
(define y-protector (make-serializer))
```

```
(parallel-execute (x-protector (lambda () (set! x (+ x 1))))
                  (y-protector (lambda () (set! y (+ y 1)))))
```

then both tasks can run at the same time; it doesn't matter how their machine instructions are interleaved. But if we say

```
(parallel-execute (x-protector (lambda () (set! x (+ x 1))))
                  (x-protector (lambda () (set! x (+ x 1)))))
```

then, since we're using the same serializer in both tasks, the serializer will ensure that they don't overlap in time.

I've introduced a new primitive procedure, `parallel-execute`. It takes any number of arguments, each of which is a procedure of no arguments, and invokes them them, in parallel rather than in sequence. (This isn't a standard part of Scheme, but an extension for this section of the textbook.)

You may be wondering about the need for all those `(lambda ()...)` notations. Since a serializer isn't a special form, it can't take an expression as argument. Instead we must give it a procedure that it can invoke.

Programming Considerations

Even with serializers, it's not easy to do a good job of writing programs that deal successfully with concurrency. In fact, all of the operating systems in widespread use today have bugs in this area; Unix systems, for example, are expected to crash every month or two because of concurrency bugs.

To make the discussion concrete, let's think about an airline reservation system, which serves thousands of simultaneous users around the world. Here are the things that can go wrong:

• **Incorrect results.** The worst problem is if the same seat is reserved for two different people. Just as in the case of adding 1 to `x`, the reservation system must first find a vacant seat, then mark that seat as occupied. That sequence of reading and then modifying the database must be protected.

• **Inefficiency.** One very simple way to ensure correct results is to use a single serializer to protect the entire reservation database, so that only one person could make a request at a time. But this is an unacceptable solution; thousands of people are waiting to reserve seats, mostly not for the same flight.

• **Deadlock.** Suppose that someone wants to travel to a city for which there is no direct flight. We must make sure that we can reserve a seat on flight A and a seat on connecting flight B on the same day, before we commit to either reservation. This probably means that we need to use *two* serializers at the same time, one for each flight. Suppose we say something like

```
(serializer-A (serializer-B (lambda () ...))))
```

Meanwhile someone else says

```
(serializer-B (serializer-A (lambda () ...))))
```

The timing could work out so that we get serializer A, the other person gets serializer B, and then we are each stuck waiting for the other one.

• **Unfairness.** This isn't an issue in every situation, but sometimes you want to avoid a solution to the deadlock problem that always gives a certain process priority over some other one. If the high-priority process is greedy, the lower-priority process might never get its turn at the shared data.

Implementing Serializers

A serializer is a high-level abstraction. How do we make it work? Here is an *incorrect* attempt to implement serializers:

```
;;;;;                        In file cs61a/lectures/3.4/bad-serial.scm
(define (make-serializer)
  (let ((in-use? #f))
    (lambda (proc)
      (define (protected-proc . args)
        (if in-use?
            (begin
             (wait-a-while)                 ; Never mind how to do that.
             (apply protected-proc args))   ; Try again.
            (begin
             (set! in-use? #t)        ; Don't let anyone else in.
             (apply proc args)        ; Call the original procedure.
             (set! in-use? #f))))     ; Finished, let others in again.
      protected-proc)))
```

This is a little complicated, so concentrate on the important parts. In particular, never mind about the *scheduling* aspect of parallelism—how we can ask this process to wait a while before trying again if the serializer is already in use. And never mind the stuff about `apply`, which is needed only so that we can serialize procedures with any number of arguments.

The part to focus on is this:

```
        (if in-use?
            .......         ; wait and try again
            (begin
             (set! in-use #t)        ; Don't let anyone else in.
             (apply proc args)       ; Call the original procedure.
             (set! in-use #f)))      ; Finished, let others in again.
```

The intent of this code is that it first checks to see if the serializer is already in use. If not, we claim the serializer by setting `in-use` true, do our job, and then release the serializer.

The problem is that this sequence of events is subject to the same parallelism problems as the procedure we're trying to protect! What if we check the value of `in-use`, discover that it's false, and right at that moment another process sneaks in and grabs the serializer? In order to make this work we'd have to have another serializer protecting this one, and a third serializer protecting the second one, and so on.

There is no easy way to avoid this problem by clever programming tricks within the competing processes. We need help at the level of the underlying machinery that provides the parallelism: the hardware and/or the operating system. That underlying level must provide a *guaranteed atomic* operation with which we can test the old value of `in-use` and change it to a new value with no possibility of another process intervening. (It turns out that there is a very tricky software algorithm to generate guaranteed atomic test-and-set, but in practice, there is almost always hardware support for parallelism. Look up "Peterson's algorithm" in Wikipedia if you want to see the software solution.)

The textbook assumes the existence of a procedure called `test-and-set!` with this guarantee of atomicity. Although there is a pseudo-implementation on page 312, that procedure won't really work, for the same reason that my pseudo-implementation of `make-serializer` won't work. What you have to imagine is that `test-and-set!` is a single instruction in the computer's hardware, comparable to the Load Word instructions and so on that I started with. (This is a realistic assumption; modern computers do provide some such hardware mechanism, precisely for the reasons we're discussing now.)

The Mutex

The book uses an intermediate level of abstraction between the serializer and the atomic hardware capability, called a *mutex*. What's the difference between a mutex and a serializer? The serializer provides, as an abstraction, a protected operation, without requiring the programmer to think about the mechanism by which it's protected. The mutex exposes the sequence of events. Just as my incorrect implementation said

```
(set! in-use #t)
(apply proc args)
(set! in-use #f)
```

the correct version uses a similar sequence

```
(mutex 'acquire)
(apply proc args)
(mutex 'release)
```

By the way, all of the versions in these notes have another bug; I've simplified the discussion by ignoring the problem of return values. We want the value returned by `protected-proc` to be the same as the value returned by the original `proc`, even though the call to `proc` isn't the last step. Therefore the correct implementation is

```
(mutex 'acquire)
(let ((result (apply proc args)))
  (mutex 'release)
  result)
```

as in the book's implementation on page 311.