Washington University
Department of Computer Science

CS422: Operating Systems

Programming Project
Due Tuesday, December 4^{th}, 1997

1 Overview

In this assignment, you will implement a thread-safe *multiplexed stream* abstraction. This mechanism makes it possible to have multiple client threads simultaneously send and receive requests to a multi-threaded server *over a single socket connection*. This feature is useful since it reduces the number of connections open to a server from each client.

To test your program, you can use the multiplexed stream abstraction to reimplement your Naming Service client application. Instead of just having one client thread making socket calls over TCP to the Name Server, you can spawn multiple client threads that will invoke methods on the Name Server. All client invocations to the same Name Server host/port tuples will be multiplexed via the same TCP connection.

2 Design and Implementation Issues

2.1 Client Functionality

Once you've determined the IP address and port number where your Name Server is listening for TCP connections, the client will use `thr_create` to spawn off *n* threads. Each thread will use the `Muxed_SOCK_Connector` shown below to obtain a `Muxed_SOCK_Stream`.

```
typedef int HANDLE;
// Make UNIX look like Win32...

class Muxed_SOCK_Connector
  // = TITLE
  //     Defines an active connection factory for
  //     multiplexed connections.
{
public:
  // = Initialization routines.
  Muxed_SOCK_Connector (void);
  // Default constructor.

  int connect (Muxed_SOCK_Stream *&new_stream,
               const INET_Addr &remote_sap);
  // Actively connect and produce a <new_stream> if things go well.
  // The <remote_sap> is the address that we are trying to connect
  // with.  If there's already a connection open to the <remote_sap>,
  // the <new_stream> passed back will be shared by all clients in
  // this process.  Otherwise, a dynamically allocated <new_stream>
  // will be passed back.  The <timeout> is the amount of time to wait to
  // connect. If it's 0 then we block indefinitely.

private:
  // ...
};
```

Each `Muxed_SOCK_Stream` references an endpoint of communication that is shared by all client threads that are connected to the same IP address and port number tuple. The `Muxed_SOCK_Stream recv` method also handles memory allocation to improve performance and reduce data copying.

```
class Muxed_SOCK_Stream
  // = TITLE
  //     Defines the methods for the multiplexed socket I/O
  //     routines (e.g., send/recv).
{
  friend class Muxed_SOCK_Connector;
public:
  ssize_t send (const void *buffer,
                size_t n,
                u_long sequence_number) const;
  // Send an <n> byte buffer to the multiplexed socket.  This
  // implementation should use <writev(2)> with an <iovec> array of
  // size 3.  The length is sent in <iovec[0]>, the sequence number is
  // sent in <iovec[1]>, and the data is sent in <iovec[2]>.

  ssize_t recv (void *&buffer,
                long &sequence_number) const;
  // Receive <buffer> from the multiplexed socket.  This call will
  // block until a <buffer> of type <sequence_number> arrives from the
  // server.  If <sequence_number> == -1 then it will return the first
  // <buffer> that arrives and set <sequence_number> to the value that
  // is associated with the <buffer>.  The <recv> method will allocate
  // the <buf> and return the size of <buffer> as the return value, 0 if
  // EOF is reached, or -1 if a failure occurs.

  int close (void);
  // Close down the socket.  This decrements the reference count
  // and frees up resources when it goes to 0.

  HANDLE get_handle (void) const;
  // Return the underlying socket handle.

  void set_handle (HANDLE);
  // Assign the underlying socket handle.

private:
  Muxed_SOCK_Stream (void);
  // Initialize data members.

  u_int ref_count_;
  // Reference count of the number of clients that
  // are multiplexing us.

  HANDLE handle_;
  // Connected socket handle.

  // ... You fill in here ...
};
```

Implementing the `Muxed_SOCK_Stream` is tricky. For sending, you'll just need to make sure that no other thread is sending while you're calling `writev(2)` to send the data on the socket connection. This will require a Muxed to serialize the use of the connection.

Receiving from the `Muxed_SOCK_Stream` is more complicated since (1) you can't allow multiple threads to call `recv` on the same connected `HANDLE` and (2) you'll want to minimize the number of context switches incurred by receiving threads. Here are the steps you'll need to consider to resolve these forces:

- *The first-thread-in blocks on* `select` – If a thread calls `recv` on a `Muxed_SOCK_Stream`, and there are no other threads already calling `recv` on this muxed stream, then your thread will wait in `select(3c)` for this `HANDLE` to become "ready to recv." When `select` returns the `recv` method will read the length of the data from the socket `HANDLE` (`recv(3n)` and then read the sequence number and data buffer (`writev(2)`. If the sequence number of the data is for the thread that called `select` you can return to the caller. Otherwise, you'll need to locate the thread that is waiting for the sequence number, wake it up (`cond_signal(3T)`), and let it return to its caller while you go back to select.

- *Subsequent threads block in* cond_wait – If there's already a thread waiting in select then all subsequent threads must block on a condition variable (cond_wait(3T)) waiting for one of two conditions: (1) the thread blocked in select will read their reply message and signal them via cond_signal or (2) the thread blocked in select will read its own reply and signal another thread to come and block in select.

2.2 Server Functionality

The server-side implements the Muxed_SOCK_Acceptor, which is a factory that creates a Muxed_SOCK_Stream to receive client requests and send replies back to the clients. Here's the interface:

```
typedef int HANDLE;
// Make UNIX look like Win32...

class Muxed_SOCK_Acceptor
  // = TITLE
  //     Defines a passive connection factory for
  //     multiplexed connections.
{
public:
  // = Initialization routines.
  Muxed_SOCK_Acceptor (const INET_Addr &local_addr);
  // Establish a listening socket.

  int accept (Muxed_SOCK_Stream &new_stream);
  // Passively connect and produce a <new_stream> if things go well.

private:
  // ...
};
```

3 Client/Server Driver Programs

3.1 Client Driver

The following illustrates a multi-threaded Name Server client application that allows client threads to lookup "employees" by their unique numerical ID. If there is a match, the name is returned to the client. Each client request to the server is run in parallel, but all requests for the same server share a single connection.

```
#include "Muxed_SOCK_Connector.h"

static int MAX_ITERATIONS = 100;

// Function called when a new thread is created.
// This function is passed the name to lookup
// from the server.

static void *
lookup_id (void *a)
{
  INET_Addr *addr = (INET_Addr *) a;

  // Active-mode connector object.
  Muxed_SOCK_Connector connector;
  Muxed_SOCK_Stream muxed_stream;

  // Accept a muxed connection from a client.
  if (server.connect (muxed_stream, *addr) == -1)
    return 0;

  // Note, this isn't portable to platforms other than Solaris.
  long sequence_number = thr_self ();
  char *buffer;

  for (int i = 0; i < MAX_ITERATIONS; i++)
```

```
    {
      char id[BUFSIZ];

      // Randomly generate an id between 1 .. 10.

      ...

      // Send the request to the server.
      if (muxed_stream.send (id,
                             strlen (id) + 1, // Include the '\0'.
                             sequence_number) == -1)
        return 0;
      // Block waiting for the reply to come for our sequence number.
      else if (muxed_stream.recv (buffer, sequence_number) == -1)
        return 0;

      fprintf (stderr, "found %s for id %s with sequence %d\n",
               buffer, id, sequence_number);

      delete [] buffer;
    }

  // Shutdown the stream.
  muxed_stream.close ();

  return 0;
}

// Default port number.
static const int default_port = 5000;

static const char *default_server = "localhost";

int
main (int argc, char *argv[])
{
  // Port number of server.
  u_short port = argc > 1
    ? atoi (argv[1]) : default_port;

  // Host name of the server.
  char *server = argc > 2
    ? argv[2] : default_server;

  // Internet address of server.
  INET_Addr addr (port, server);

  for (int i = 0; i < max_threads; i++) {
    // Spawn off a thread-per client request.
    if (thr_create (0, 0,
                    lookup_id,
                    (void *) &addr,
                    THR_NEW_LWP,
                    &t_id) != 0)
      perror ("thr_create");
  }

  // Wait for all the threads to exit.
  while (thr_join (0, 0, 0) == 0)
    continue;

  // NOTREACHED
  return 0;
}
```

3.2 Server Driver

The following illustrates a multi-threaded Name Server that allows a client to lookup "employees" by their unique numerical ID. If there is a match, the name is returned to the client. Each client request to the server is run in parallel, but all requests from the same client share a single connection. Note that this implementation only supports one client...

```
#include "Muxed_SOCK_Acceptor.h"

struct Args
{
```

```
  Args (char *r, long s)
    : recvline_ (r), sequence_number (s) {}

  ~Args (void) { delete [] recvline_; }

  char *recvline_;
  // Request from the client.

  long sequence_number_;
  // Sequence number of the client's request.
};

// Function called when a new thread is created.
// This function is passed arguments used to receive
// a Name Service lookup request from a client.

static void *
lookup_name (void *a)
{
  enum {
    EMPNAMELEN = 512
    // Maximum size of employee name.
  };

  // = Simple read-only name service database.
  static struct {
    int emp_id;
    const char emp_name[EMPNAMELEN];
  } emp_db[] =
    {
      1, "George Strait",
      2, "George Bush",
      3, "Boy George",
      4, "George Burns",
      5, "George Washington",
      6, "George of the Jungle",
      7, "George Michael",
      8, "George Plimpton",
      9, "George Hamilton",
      10, "George Clooney",
      0, ""
    };

  Args *args = (Args *) a;

  char sendline[EMPNAMELEN];
  int emp_id = atoi (args->recvline_);
  int found = 0;
  ssize_t n;

  for (int index = 0;
       found == 0 && emp_db[index].emp_id;
       index++)
    if (emp_id == emp_db[index].emp_id) {
      found = 1;
      n = sprintf (sendline, "%s",
                   emp_db[index].emp_name);
    }

  if (found == 0)
    n = sprintf (sendline, "%s", "ERROR");

  // Send the reply back to the client.
  muxed_stream->send (sendline,
                      n + 1, // Include the '\0'.
                      args->sequence_number_);

  delete args;
  return 0;
}

// Default port number.
static const int default_port = 5000;

int
main (int argc, char *argv[])
{
  // Port number of server.
  u_short port = argc > 1
```

```
    ? atoi (argv[1]) : default_port;

  // Internet address of server.
  INET_Addr addr (port);

  // Passive-mode acceptor object.
  Muxed_SOCK_Acceptor server (addr);

  Muxed_SOCK_Stream muxed_stream;

  // Accept a muxed connection from a client.
  if (server.accept (muxed_stream) == -1)
    return -1;

  for (;;) {
    char *recvline;

    // We'll take any buffer containing any
    // sequence number that comes along.
    long sequence_number = -1;

    // Receive any new request.
    ssize_t n = muxed_stream.recv (recvline,
                                   sequence_number);
    if (n <= 0)
      return n;

    Args args = new Args (recvline,
                          sequence_number);
    thread_t t_id;

    // Spawn off a thread-per client request.
    if (thr_create (0, 0,
                    lookup_name,
                    (void *) args,
                    THR_NEW_LWP | THR_DETACHED,
                    &t_id) != 0)
      perror ("thr_create");
  }

  // NOTREACHED
  return 0;
}
```

4 Concluding Remarks

This assignment is hard. I recommend that you start early.