$Id: asg4-cxi-cxid-code.mm,v 1.60 2022-02-19 18:25:14-08 - - $
/afs/cats.ucsc.edu/courses/cse111-wm/Assignments/asg4-client-server
https://www2.ucsc.edu/courses/cse111-wm/:/Assignments/asg4-client-server/

Some code has been provided in several modules.

1. `debug.{h,cpp}`

As in previous assignments.

2. `logstream.h`

Is a wrapper for an **`ostream`** that can be used to print debugging information, but prefixes each output line with the execname and the process id. Since all code is in the header file, a **`cpp`** files is not needed.

Usage : Declare a variable such as **`logstream outlog (cout);`**.
Then use it in the same way as **`cout`** :

```
  outlog << ..... << endl;
```

The execname and process id (pid) via **`getpid`**(2) is printed followed by whatever else is to be printed, represented by the ..... above.

The operator

```
  template <typename T>
  ostream& operator<< (const T& obj)
```

is a template so that its right operand can be of any type that **`ostreeam::operator<<`** can accept as a right operand.

The field

```
  ostream& out_;
```

is a reference, not a value because objects of type **`ostream`** can not be copied. Note also that the ctor initializes it by using a field intializer. It can not be initialized by **`operator=`** inside the body of the ctor.

3. **`protocol.{h,cpp}`**

Implements the protocol that interacts between client and server. The **`cxi_header`** is precisely defined, along with the enum class **`cxi_commands`** to be used in communication.

(a) **`static_assert`**
verifies the exact size of the header, and **`uint32_t`** specifies that the count field is exactly 32 bits. This limits packet size to 4,294,967,296 bytes, but that is more than large enough for a student project.

`static_assert` is determined at compile time and causes the compilation to be rejected if its argument is false. An **`assert`**, on the other hand, is a macro whose argument is tested at runtime.

(b) **`send_packet`**
expects a socket and a buffer and loops using the socket's send until the entire message is sent.

(c) **`recv_packet`**
uses socket's recv and receive a message. It is expected that the buffer thus passed in is large enough to accept the message.

(d) **`operator<<`**
is for debug printing the header and checking for network or host byte order errors in the header.

Warning: Some strange errors will occur that will possibly lead to deadlock between the client and the server, or other strange occurrences, if you forget to use **`ntohl`** and **`htonl`** when communicating between the client and the server.

4. `socket.{h,cpp}` — `base_socket`

Is a wrapper around the more cumbersome standard C interface to sockets. It wraps a socket file descriptor. `base_socket` defines all of the communication functions. No inheritance is used, but it is convenient to separate the user interfaces into separate purposes.

Most of its functions are `protected` and may only be used by the subclasses. Since the ctor and dtor are protected, users are prohibited from declaring a `base_socket`. Program code in the client, daemon, and server only declare objects of the subclasses.

The field values are :

(a) `int socket_fd`
file descriptor (small integer) returned by the `socket` system call.

(b) `sockaddr_in`
containing the connection information for communication. Its fields are :
- `sa_family_t sin_family` : address family : `AF_INET`
- `in_port_t sin_port` : net byte order `uint16_t` port number
- `struct in_addr sin_addr` : net byte order `uint32_t` internet address

Since these have access to operating system facilities, the copiers and movers have been declared as `=delete`.

The following functions are inherited by the derived classes and may be used by them. As memtioned previously, the user may not instantiate this class directly. The available functions are :

(a) `void close();`
Close the socket. The destructor automatically closes a socket, sot this may be little redundant.

(b) `ssize_t send (const void* buffer, size_t bufsize);`
Send a message to the socket, returning the number of bytes actually sent.

(c) `ssize_t recv (void* buffer, size_t bufsize);`
Receive a message from a buffer of the given size, reporting the number of bytes actually received.

(d) `friend string to_string (const base_socket& sock);`
A debugging format function.

5. `socket.{h,cpp}` — user interface

The user interface functions specialize the uses of a base socket for particular purposes. Only the constructors differ in the way the socket is created. Usage for all of the is by send and receive.

(a) **`server_socket`**
is used by the daemon, and perhaps should have been named the daemon socket instead.
- Create a socket for communication on a specific server port.
- Bind the socket to the specific part.
- Listen for a connection request from a client, which is then accepted.

(b) **`accepted_socket`**
- Created by a server accepting a call from a client and used by the server sub-process to communicate with a client.

(c) **`client_socket`**
is used by a client in an attempt to connect to a server.
- Creates a socket to request the connection.
- Attempts to connect to a server, given a particular host and port.

6. `cxi.cpp`

is the client code. It reads commands from the input and interprets each command as needed by sending a packet to the server. It has the client's main loop and calls a separate function for each of the client commands.

(a) **`cxi_ls`**
example function which shows to the client interacts with the server. Implements the **`ls`** command.

(b) **`make_unique`**
The statement
```
  auto buffer = make_unique<char[]> (host_nbytes + 1);
```
creates a **`unique_ptr`** pointing to a buffer of the appropriate size. This buffer is automatically freed when the pointer goes out of scope. The size of the buffer is known in advance, having been received in the header. In the case of loading a file, the **`stat`**(2) system call can determine the size of a file.

(c) **`stat`**(2)
```
  struct stat stat_buf;
  int status = stat (filename, &stat_buf);
```
will enquire as the the status of the given filename. If the result is 0, then **`stat_buf.st_size`** contains the number of bytes in the file. Note that **`filename`** is a **`const char*`**, not a **`string`**.

(d) If a system call fails, an error is reported to the standard error in the usual manner, and the exit status for the program is set to **`EXIT_FAILURE`**.
execname: filename: reason
- *execname* is always the **`basename`**(3) of the program being run (derived from **`argv[0]`**).
- *filename* is the name of the file or other object that could not be accessed.
- *reason* is the explanation of the problem from **`strerror(errno)`**.

7. `cxid.cpp` — daemon

Is both the daemon and server code in one program, but multiple processes. The daemon creates a **`server_socket`** which is used to listen for a client connection. When a client connects, **`fork`**(2) is used to create a child process to execute the server code. It also reaps zombie processes which are servers from which a client has disconnected. There is a separate server process for each client that connects.

(a) **`main`**
The main function sets up a signal action and a server socket to listen for a client. Whenever a client connects, it forks a child process to preform the server task. It is an infinite loop and will run until killed at the terminal.

(b) **`reap_zombies`**
when a child process exits, it becomes a zombie until the parent waits for it or until the parent exits. But the daemon is in an infinite loop, so it uses **`wait-pid`**(2) to report the status of the child process, thus removing the child server from the process table.

(c) **`fork_cxiserver`**
Use **`fork`**(2) to run the server. It must close the unused sockets (the server in the child and the accepted socket in the parent).

8. `cxid.cpp` — server

contains the run_server process, which loops reading commands from the client, acting on those commands, then sending back a reply. It runs in an infinite loop, exiting only when the client disconnects.

(a) **`run_server`**
is the child process that waits for a command from the client and then replies as appropriate. It continues until the client disconnects, at which time the server exits.

(b) **`reply_ls`**
is an example of how to reply to a client. It uses **`popen`**(3) to create a pipe to **`ls`**(1) to list the files in the server's directory. Then it sends the information to the client.