CSE333 – Section 4

fread vs. read
Client-side Network Code

Cody A. Schroeder

April 19, 2012

fread/fwrite

- Standard C library functions.
- Takes a FILE stream and reads/writes `nmemb` elements of data, each `size` bytes long, into/from the given buffer (`ptr`).
- Function will block until all `nmemb` elements are read/written, until an error occurs, or until an EOF is reached.
- Returns the number of elements successfully read/written.
    - EOF and errors can be checked with feof(3) and ferror(3).
    - Errors/EOFs may result in anywhere between 0 to `nmemb` elements being read/written.

```
#include <stdio.h>
size_t fread(void *ptr, size_t size, size_t nmemb,
             FILE *stream);
size_t fwrite(const void *ptr, size_t size, size_t nmemb,
              FILE *stream);
```

read/write

- POSIX system call (very low level).
- Takes a file descriptor (socket, file, etc.) and reads/writes **up to** `count` bytes into/from the given buffer.
- Returns the number of bytes read/written or -1 on an error.
  - 0 bytes can be written on a success.
  - 0 bytes read signals an EOF.
  - On an error, errno is set appropriately to an error (i.e. EAGAIN, EINTR, EBADF, EFAULT, etc.)

```c
#include <unistd.h>
ssize_t read(int fd, void *buf, size_t count);
ssize_t write(int fd, const void *buf, size_t count);
```

Comparison

- What are the primary differences between fread and read?
  - Library function vs. System call
  - FILE streams vs. File descriptors
  - Blocking for EOF or N bytes vs. You get whatever is available
  - Buffering
  - Return Values and Errors
- Which is better...?
  - **IT DEPENDS!!!**
  - for reading from the filesystem?
    - usually fread (Why?)
  - for network IO?
    - usually read (Why?)

Basic Idea

Client-side Network Code

- DNS Resolve a Name (e.g. `www.uw.edu` to 140.142.16.69)
- Create a Socket
- Connect the Socket to the Resolved Address
- Use `read`/`write` on the Resulting File Descriptor
- Close the Socket

Network Address Translation

LookupName

```
bool LookupName(char *name,
                unsigned short port,
                struct sockaddr_storage *ret_addr,
                size_t *ret_addrlen) {
  struct addrinfo hints, *results;
  int retval;

  memset(&hints, 0, sizeof(hints));
  hints.ai_family = AF_UNSPEC;
  hints.ai_socktype = SOCK_STREAM;

  // Do the lookup by invoking getaddrinfo().
  if ((retval = getaddrinfo(name, NULL, &hints, &results)) != 0) {
    cerr << "getaddrinfo failed: ";
    cerr << gai_strerror(retval) << endl;
    return false;
  }
  assert(results != NULL);

  // .......
```

Network Address Translation (cont.)

LookupName (cont.)

```
    // .......

    // Set the port in the first result.
    if (results->ai_family == AF_INET) {
      struct sockaddr_in *v4addr = (struct sockaddr_in *) results->ai_addr;
      v4addr->sin_port = htons(port);
    } else if (results->ai_family == AF_INET6) {
      struct sockaddr_in6 *v6addr = (struct sockaddr_in6 *) results->ai_addr;
      v6addr->sin6_port = htons(port);
    } else {
      cerr << "getaddrinfo failed to provide an IPv4 or IPv6 address";
      cerr << endl;
      return false;
    }

    // Return the first result.
    memcpy(ret_addr, results->ai_addr, results->ai_addrlen);
    *ret_addrlen = results->ai_addrlen;

    // Clean up.
    freeaddrinfo(results);
    return true;
}
```

Initiate a Socket Connection

Connect

```
bool Connect(const struct sockaddr_storage &addr,
             const size_t &addrlen,
             int *ret_fd) {
  // Create the socket.
  int socket_fd = socket(addr.ss_family, SOCK_STREAM, 0);
  if (socket_fd == -1) {
    cerr << "socket() failed: " << strerror(errno) << endl;
    return false;
  }

  // Connect the socket to the remote host.
  int res = connect(socket_fd,
                    reinterpret_cast<const sockaddr *>(&addr),
                    addrlen);
  if (res == -1) {
    cerr << "connect() failed: " << strerror(errno) << endl;
    return false;
  }

  *ret_fd = socket_fd;
  return true;
}
```

- Write a C++ program that will
  - connect to a server and port (given by the command-line),
  - read a line of input from the user (using cin),
  - send that line of data to the server (including the '\n'),
  - read a line of data back from the server (terminated by a '\n'),
  - print that line to stdout (using cout),
  - repeat the read/write loop until cin has reached an EOF.
- Grab the template code from:

  http://www.cs.washington.edu/education/courses/cse333/12sp/sections/sec4/echoClient.cc
  - You'll get the lecture code we talked about and a basic main.
- I've setup a few test servers to try out your code:
  - An echo server:

    ~~./echoClient attu1.cs.washington.edu 9442~~
  - A **FUN** server:

    ~~./echoClient attu1.cs.washington.edu 9443~~
  - An ELIZA server:

    ~~./echoClient attu1.cs.washington.edu 9444~~

Don't forget to turn your code into Catalyst!!!