CS551
Warm-up Project #1
Bill Cheng

http://merlot.usc.edu/cs551-f12
typedef struct tagReqMsg {
    unsigned short MsgType;
    unsigned int Offset;
    unsigned char ServerDelay;
    unsigned int DataLen;
    char *Data;
} ReqMsg;

int SendReq(int n_socket) {
    ReqMsg request;

    memset(&request, 0, sizeof(ReqMsg));
    /* fill up the request data structure */
    if (write(n_socket, &request, sizeof(ReqMsg)) == sizeof(ReqMsg)) {
        return 0;
    }
    switch (errno) {
    case EINTR: ...
    default:
        fprintf(stderr, "Unrecognized errno %1d in SendReq()\n", errno);
        break;
    }
    return (-1);
}
Is `sizeof(ReqMsg)` 11?

```c
typedef struct tagReqMsg {
    unsigned short MsgType;
    unsigned int Offset;
    unsigned char ServerDelay;
    unsigned int DataLen;
    char *Data;
} ReqMsg;
```

Filling the data structure

```c
unsigned short usAddrReqMsgType = (unsigned short)0xfe10;

request.MsgType = usAddrReqMsgType;
request.Offset = 0;
request.ServerDelay = 0;
request.DataLen = strlen("www.google.com");
request.Data = argv[3];
```

This is incorrect

"www.google.com"
Memory Layout (Cont...)

- stream abstraction of TCP

```c
int msg_buf_sz=10+strlen("www.google.com")+1;
char *msg_buf=(char*)malloc(msg_buf_sz);

if (msg_buf == NULL) { fprintf(stderr, "malloc() failed\n"); ...
memset(msg_buf, 0, msg_buf_sz);
memcpy(msg_buf, &usAddrReqMsgType, 2); /* is this right? */
memcpy(&msg_buf[2], &request.Offset, 4); /* is this right? */
msg_buf[6] = &request.ServerDelay, 1;
memcpy(&msg_buf[7], &request.DataLen, 4); /* is this right? */
strcpy(&msg_buf[11], request.Data); /* is this right? */
...
free(msg_buf);
```

- need to call htons()/htonl() before sending and
  ntohss()/ntohl() after receiving
- in order to make sure a data object is 2/4 bytes long, you
can use uint16_t/uint32_t
- there is really no difference between signed and unsigned
  - except in the context of negative numbers, then you
    need to watch out for sign extension
TCP’s Stream Abstraction

1st write:  (Note: Assuming you can write up to 2048 bytes at a time)

<table>
<thead>
<tr>
<th>0 1 2 3 4 5 6 7 8 9 10 11</th>
<th>...</th>
<th>2047</th>
</tr>
</thead>
<tbody>
<tr>
<td>MT1</td>
<td>O1</td>
<td>DL1</td>
</tr>
</tbody>
</table>

2nd write:

<table>
<thead>
<tr>
<th>0 1 2 3 4 5 6 7 8 9 10 11</th>
<th>...</th>
<th>2047</th>
</tr>
</thead>
<tbody>
<tr>
<td>MT2</td>
<td>O2</td>
<td>DL2</td>
</tr>
</tbody>
</table>

3rd write:

<table>
<thead>
<tr>
<th>0 1 2 3 4 5 6 7 8 9 10 11</th>
<th>...</th>
<th>910</th>
</tr>
</thead>
<tbody>
<tr>
<td>MT3</td>
<td>O3</td>
<td>DL3</td>
</tr>
</tbody>
</table>

Receiver *concatenates* all bytes received

| MT1 | O1 | DL1 | D1 | MT2 | O2 | DL2 | D2 | MT3 | O3 | DL3 | D3 |

Need:  

| MT | O | DL | D1 | D2 | D3 |
TCP’s Stream Abstraction (Cont...)

- for warmup #1 (and warmup #1 only), you must read and write *one byte at a time*
  - this means that if you call `send()` or `write()` with the first argument being a socket descriptor, the 3rd argument must be 1
Warmup Project #1

- 3 request types
  - ADDR → ADR_REQ
  - FILESIZE → FSZ_REQ
  - GET → GET_REQ

- 3 reply types
  - ADDR_RPLY
  - FSZ_RPLY
  - GET_RPLY

- other
  - ADR_FAIL
  - FSZ_FAIL
  - GET_FAIL
  - ALL_FAIL

Client program commandline
- client {adr|fsz|get} [-d delay] [-o offset] \\
  [-m] hostname:port string

Message format

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | ...
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Type</td>
<td>Offset</td>
<td>DataLength</td>
<td>Data</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- for requests, Data came from string in commandline

Server program commandline
- server [-t seconds] [-m] port
Examples

ADDR
client adr nunki.usc.edu:6001 www.cs.usc.edu
<TAB>ADDR = 128.125.3.104

FILE_SIZE
client fsz nunki.usc.edu:6001 /etc/passwd
<TAB>FILESIZE = 1030

GET
client get nunki.usc.edu:6001 /bin/less
<TAB>FILESIZE = 104908, MD5 = f27df2e0...

client get -o 123 nunki.usc.edu:6001 /bin/less
<TAB>FILESIZE = 104785, MD5 = eccfd764...

openssl md5 /bin/less
MD5(/bin/less) = f27df2e0...
Many Requirements

Please read the spec yourself for details

- Ex:
  - separate compilation
  - buffer size limit
  - reading and writing one byte at a time

Be careful with binary data

- binary file contains binary data
- MD5 buffer contains binary data
- write a function to print binary data correctly
  - if you use "\%x" in `printf()`, the corresponding data is assume to be a signed integer
  - if the most significant bit is 1, will cause sign-extension
Some Major Requirements for *All* Projects

- Severe penalty for failing `make`
- Severe penalty for using large memory buffers
- Severe penalty for any segmentation fault -- you must test your code well
- Severe penalty for not using separate compilation or for having all your source code in header files -- you must learn to plan how to write your program

- Never do *busy-wait*
  - run "`top`" on nunki
  - don’t stay in a tight loop and poll
    - just sleep for 50-100 milliseconds before poll again
  - use blocking I/O and sockets
Separate Compilation

- Break up your code into *modules*
  - *compile the modules separately*, at least one rule per module per rule in the Makefile
  - a separate rule to *link* all the modules together
    - if your program requires additional libraries, add them to the link stage

To receive full credit for separate compilation

- to create an executable, at a minimum, you must run the compiler at least *twice* and the linker *once*
- for warmup #1, there are two executables, they can share modules
Don’t design your program "procedurally"

You need to learn how to write functions!
- a function has a well-defined interface
  - what are the meaning of the parameters
  - what does it suppose to return
- pre-conditions
  - what must be true when the function is entered
  - you assume that these are true
    - you can verify it if you want
- post-conditions
  - what must be true when the function returns
- you design your program by making designing a sequence of function calls
Sticky Issues

→ Your server must shutdown **gracefully**
  → wait for all child threads/processes to terminate before the server terminates itself
  ○ must not kill child threads/processes abruptly
  ○ send signals to child threads/processes
    ◇ a child thread/process must be prepared to handle this and self-terminates
    ◇ a child thread/process should react as soon as possible
  ○ since we are read the socket one byte at a time, you should check if it’s time to quit after reading a byte or if `select()` times out (after ~100ms)
  ○ since we are writing to the socket one byte at a time, you should check if it’s time to quit after writing out a byte
Sticky Issues (Cont...)

Your server must shutdown *gracefully* (cont...)

- in order to do this, the server needs to know which child thread/process has terminated
  - keep a list of child thread/process IDs
  - more trickly if you use child processes
    - should handle SIGCHLD explicitly (i.e., need to reap child processes)
    - call `waitpid()` in SIGCHLD handler
    - watch out for a *race condition*
Race Condition

Race condition (only if you use `fork()`)

- SIGCHLD handler:

  ```c
  void sigchld_handler(...) {
    for (;;) {
      pid = waitpid((pid_t)(-1), &status, WNOHANG);
      if (pid == 0) break; /* == 0 for Linux, <= for Solaris */
      remove_from_list(pid);
    }
  }
  ```

- server infinite loop:

  ```c
  for (;;) {
    newsockfd = accept(nSocket, ...);
    if (newsockfd > 0) {
      int pid=fork();
      if (pid == 0) {
        close(nSocket);
        child_processing(newsockfd);
        exit(0);
      }
      close(newsockfd);
      add_to_list(pid);
    }
  }
  ```

- **what if `remove_from_list()` happens first?**
Race Condition (Cont...)

Fix for the race condition (only if you use fork())
- block SIGCHLD until add_to_list() is finished

```c
for (;;) {
    newsockfd = accept(nSocket, ...);
    if (newsockfd > 0) {
        sigprocmask(SIG_BLOCK, ...);
        int pid=fork();

        if (pid == 0) {
            close(nSocket);
            sigprocmask(SIG_UNBLOCK, ...);
            child_processing(newsockfd);
            exit(0);
        }
        close(newsockfd);
        add_to_list(pid);
        sigprocmask(SIG_UNBLOCK, ...);
    }
}
```

Maybe it’s easier just to use pthread and mutex
- also warms you up for warmup project #2
- but you need to learn how to deliver signal to a specific thread - see beginning of Warmup Project #2 slides