

Worked Example: Web Server

Networked Systems Architecture 3
Lecture 19



UNIVERSITY
of
GLASGOW

Lecture Outline

- Review of laboratory exercise
- Examination

Review of Lab Work


- Four laboratory tasks:
 - Simple web download
 - Web server – single request
 - Web server – multiple requests, sequential
 - Web server – multiple requests, concurrent
- Will review operation of web server

Basic Operation

```
int
main()
{
    int                sfd;
    int                cfd;
    struct sockaddr_in caddr;
    socklen_t          caddr_len = sizeof(caddr);

    if ((sfd = create_socket()) == -1) {
        return 1;
    }

    while (!shouldExit) {
        if ((cfd = accept(sfd, (struct sockaddr *) &caddr, &caddr_len)) == -1) {
            perror("Unable to accept connection");
            shouldExit = 1;
        } else {
            process_request(cfd);
        }
    }
    close(sfd);
    return 0;
}
```



Address of client

Basic logic:
1) accept new connection
2) process request from that connection

Creating a Socket

```
static int
create_socket(void)
{
    int                fd;
    struct sockaddr_in  addr;

    fd = socket(AF_INET, SOCK_STREAM, 0);    Create a TCP/IP socket
    if (fd == -1) {
        Error...
    }

    addr.sin_family    = AF_INET;
    addr.sin_port      = htons(8080);        Bind to port 8080
    addr.sin_addr.s_addr = INADDR_ANY;      Any available interface
    if (bind(fd, (struct sockaddr *) &addr, sizeof(addr)) == -1) {
        Error...
    }

    if (listen(fd, 4) == -1) {               Listen for connections
        Error...
    }
    return fd;
}
```

Processing Requests

```
static void
process_request(int fd)
{
    while (1) {
        char    buf[BUFLen];
        char    *headers    = malloc(1);
        unsigned headerLen = 0;
        char    filename[1024];
        int     rlen;
        int     inf;

        headers[0] = '\0';
        while (strstr(headers, "\r\n\r\n") == NULL) {
            rlen = read(fd, buf, BUFLen);
            if (rlen == 0) {          // Connection closed by client
                Cleanup and exit
            }
            if (rlen == -1) {
                Error...
            }
            headerLen += rlen;
            headers = realloc(headers, headerLen + 1);
            strncat(headers, buf, rlen);
        }
    }
    ...
}
```

Retrieve the request – note that this reads until a blank line is received (signalled by two end of line markers “\r\n\r\n” in a row)

Space is allocated for arbitrary length headers – inefficient, since existing headers are copied each time realloc() is called

Processing Requests

```
...
// Parse the HTTP request, to determine the requested filename.
// Note that we specify a maximum field width, to avoid buffer
// overflow attacks when parsing long filenames.
if (sscanf(headers, "GET /%1023s HTTP/1.1", filename) != 1) {
    Error...
}

if (!hostname_matches(headers)) {
    send_response_404(fd, filename);
    free(headers);
    break;
}

if ((inf = open(filename, O_RDONLY, 0)) == -1) {
    send_response_404(fd, filename);
} else {
    send_response_200(fd, filename, inf);
};
close(inf);
free(headers);
}
close(fd);
};
```

A malicious client may send us an arbitrary length filename

Checking the Hostname

```
static int
hostname_matches(char *headers)
{
    char    *host;
    char    *colonpos;
    char    hostname[256];
    char    myhostname[256];
    char    domainname[256];

    // Parse the HTTP headers, to find and validate the "Host:" header.
    // Note that we search for a newline followed by "Host:", to avoid
    // matching other headers that end in "Host:".
    host = strstr(headers, "\nHost:");
    if ((host == NULL) || (sscanf(host, "\nHost: %255s\n", hostname) != 1)) {
        printf("Cannot parse HTTP Host: Header\n");
        return 0;
    }

    // When running on a non-standard port, browsers include a colon
    // and the port number in the "Host:" header. Strip this out.
    if ((colonpos = strchr(hostname, ':')) != NULL) {
        *colonpos = '\0';
    }
    ...
}
```


Checking the Hostname

```
...
gethostname(myhostname, 256);
if (strcmp(hostname, myhostname) != 0) {
    // The hostname in the request didn't match the return from gethostname().
    // There are three possible reasons for this:
    // 1) The hostname in the request doesn't match our hostname
    // 2) The hostname in the request doesn't include the domain name, but
    //    gethostname() does (gethostname() works this way on MacOS X)
    // 3) The hostname in the request might include the full domain name,
    //    while gethostname() on this machine returns only the host part
    //    (this is how gethostname() works on Linux)
    // Cases (2) and (3) are okay, and should be accepted, so check for these now.
    char    myNameDom[512];
    char    reNameDom[512];

    getdomainname(domainname, 256);
    sprintf(myNameDom, "%s.%s", myhostname, domainname);
    sprintf(reNameDom, "%s.%s", hostname, domainname);
    if ((strcmp(hostname, myNameDom) != 0) && (strcmp(reNameDom, myhostname) != 0)){
        return 0;
    }
}
return 1;
}
```

Sending Responses

```
send_response(int fd, char *data)
{
    write(fd, data, strlen(data));
}

static void
send_response_404(int fd, char *filename)
{
    // Requested file doesn't exist, send an error
    send_response(fd, "HTTP/1.1 404 File Not Found\r\n");
    send_response(fd, "Content-Type: text/html\r\n");
    send_response(fd, "Content-Length: 105\r\n");
    send_response(fd, "\r\n");
    send_response(fd, "<html>\r\n");
    send_response(fd, "<head>\r\n");
    send_response(fd, "<title> 404 File Not Found </title>\r\n");
    send_response(fd, "</head>\r\n");
    send_response(fd, "<body>\r\n");
    send_response(fd, "<p> File not found </p>\r\n");
    send_response(fd, "</body>\r\n");
    send_response(fd, "</html>\r\n");

    printf("404 %s\n", filename);
}
```

Sending Responses

```
static void
send_response_200(int fd, char *filename, int inf)
{
    // File exists, send OK response:
    struct stat      fs;
    char             *extn;
    char             buf[BUFLen];
    int              rlen;

    send_response(fd, "HTTP/1.1 200 OK\r\n");

    extn = strrchr(filename, '.');           Generate and send Content-Type: based on the extension
    if (extn == NULL) {
        // No extension on the requested filename
        send_response(fd, "Content-Type: application/octet-stream\r\n");
    } else if (strcmp(extn, ".html") == 0) {
        send_response(fd, "Content-Type: text/html\r\n");
    } else if ...
        ...
    } else {
        // Unknown extension
        send_response(fd, "Content-Type: application/octet-stream\r\n");
    }
    ...
}
```

Sending Responses

```
...
// Find file size, generate and send Content-Length:
fstat(inf, &fs);
sprintf(buf, "Content-Length: %d\r\n", (int) fs.st_size);
send_response(fd, buf);

// Blank line indicates end of headers
send_response(fd, "\r\n");

// Send the requested file
while ((rlen = read(inf, buf, BUFLen)) > 0) {
    write(fd, buf, rlen);
}

printf("200 %s (%d bytes)\n", filename, (int) fs.st_size);
}
```

Functions from <stdio.h> could also be used

Multiple Connections

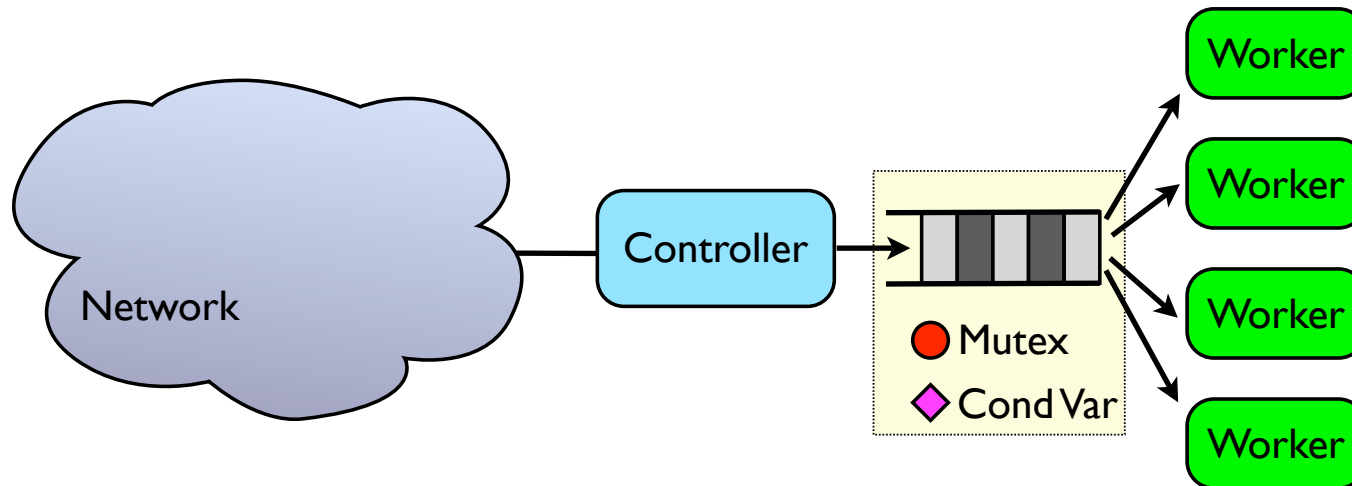
```
static void *
response_thread(void *arg)
{
    int fd = * ((int *) arg);
    free(arg);

    printf("[fd=%02d] Connection opened\n", fd);
    while (1) {
        ...as process_request() before
    }
}

int main()
{
    pthread_t  t;
    ...
    if ((cfd = accept(sfd, (struct sockaddr *) &caddr, &caddr_len)) == -1) {
        ...
    } else {
        int *arg = malloc(sizeof(int));
        *arg = cfd;
        pthread_create(&t, NULL, response_thread, arg);
    }
    ...
}
```

*Simplest concurrent approach: create a thread
for each request; requires only minimal changes
relative to the single connection case*

Thread Pool



- Single controller thread – accepts connections and adds to work queue
- Pool of worker threads – take connections from queue, generate responses
- Access to work queue protected by a mutex; workers wait on a condition variable

The Work Queue

```
struct work_queue_elem {  
    int                fd;  
    struct work_queue_elem *next;  
};
```

Each piece of work is an accepted file descriptor, from which a request should be read

```
struct work_queue {  
    pthread_mutex_t    lock;  
    struct work_queue_elem *head;  
    int                should_exit;  
    int                worker_waiting;  
    pthread_cond_t     worker_cv;  
};
```

*Mutex lock to synchronise access to the queue
Single-linked list of work items*

*Number of workers waiting
Condition variable, on which workers wait*

```
int  
main()  
{  
    struct work_queue *wq = malloc(sizeof(struct work_queue));  
    ...  
    wq->head          = NULL;  
    wq->should_exit    = 0;  
    wq->worker_waiting = 0;  
    pthread_mutex_init(&wq->lock, NULL);  
    pthread_cond_init(&wq->worker_cv, NULL);  
}
```

Setup the work queue

Work Queue Initialisation

```
pthread_t          threads[NUM_THREADS];

// Setup the work queue...
wq->head           = NULL;
wq->should_exit     = 0;
wq->worker_waiting  = 0;
pthread_mutex_init(&wq->lock, NULL);
pthread_cond_init(&wq->worker_cv, NULL);

// Create the worker threads...
for (i = 0; i < NUM_THREADS; i++) {
    pthread_create(&threads[i], NULL, response_thread, wq);
}
```


Adding Work to the Queue

```
if ((cfd = accept(sfd, (struct sockaddr *) &caddr, &caddr_len)) == -1) {
    perror("Unable to accept connection");
    break;
} else {
    struct work_queue_elem *wqe;

    pthread_mutex_lock(&wq->lock);

    wqe = malloc(sizeof(struct work_queue_elem));
    wqe->fd = cfd;
    wqe->next = wq->head;

    wq->head = wqe;

    if (wq->worker_waiting) {
        pthread_cond_signal(&wq->worker_cv);
    }

    pthread_mutex_unlock(&wq->lock);
}
```

Lock the work queue

Create and populate struct representing new work item (an accepted file descriptor)

Add to head of work queue

Signal workers that there's a new work item available

Unlock the work queue

Accepting a Work Item

```
pthread_mutex_lock(&wq->lock);
```

Lock the work queue

```
while (wq->head == NULL) {  
    if (wq->should_exit) {  
        pthread_mutex_unlock(&wq->lock);  
        pthread_exit(NULL);  
    }  
  
    wq->worker_waiting++;  
    pthread_cond_wait(&wq->worker_cv, &wq->lock);  
    wq->worker_waiting--;  
}
```

*Loop, waiting for new work
Time to exit?*

```
    wq->worker_waiting++;  
    pthread_cond_wait(&wq->worker_cv, &wq->lock);  
    wq->worker_waiting--;  
}  
wqe = wq->head;  
wq->head = wqe->next;
```

Nothing to do, wait for next request...

Take work item from queue

```
pthread_mutex_unlock(&wq->lock);
```

Unlock the work queue

Must loop waiting on the condition variable, since another thread may take the work

Laboratory Exercise: Wrap Up

- Aimed to introduce socket programming in C, showing operation of a typical Internet server
 - Should understand how to use Berkeley Sockets API
 - Should understand design pattern for single threaded and concurrent server
 - Should understand basic concepts in parsing text-based protocols – and difficulty getting it correct!

Examination

- Assessment: 100% examination
- Format of exam:
 - Two hours; answer all 3 questions
 - Laboratory material will be explicitly examined
- Past papers in the library
 - The course website has the 2007 exam and a worked answer – *strongly recommend trying the questions before looking at the worked answer*

Questions?