

电子科技大学

UNIVERSITY OF ELECTRONIC SCIENCE AND TECHNOLOGY OF CHINA

网络编程大作业文档



题 目：Web 服务器的实现与测试

学院名称：信息与软件工程学院

学生姓名：李晓旺

学号：202122090622

指导教师：任立勇

时 间：2021 年 12 月 05 日

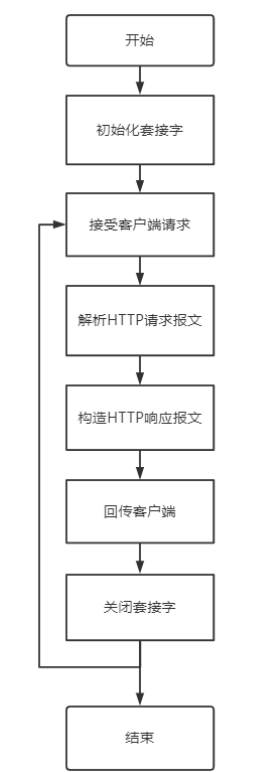
1 引言

本大作业文档主要涉及多种体系结构 Web 服务器的程序设计说明与测试结果。本文档对 Web 服务器进行需求分析，对迭代、多进程、多线程、I/O 复用、线程池、流水线等多种技术实现的 Web 服务器进行设计与实现，最后利用 Siege（Web 压力测试和评测工具）对各种 Web 服务器进行测试对比。

本大作业文档主要由系统设计与代码实现、测试对比两大部分组成。

2 系统设计与代码实现

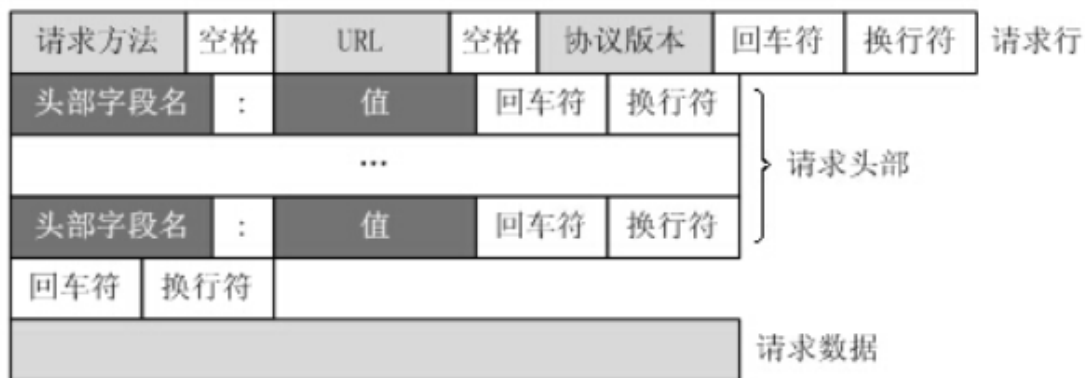
2.1 常规 Web 服务器处理流程图



如上图,是常规 Web 服务器处理的流程图。服务器初始化套接字,调用 `accept` 接受客户端请求,从客户端传入的数据中按照 HTTP 协议格式解析 HTTP 请求报文,再按照 HTTP 协议格式构造 HTTP 响应报文,将响应报文回传给客户端,关闭该套接字,服务器重新等待客户端请求或者关闭服务器程序。该 Web 服务器流程结束。

2.2 HTTP 协议解析与响应

2.2.1 HTTP 请求解析



如上图是 HTTP 请求报文的格式，在该项目中实现简易 Web 文件服务器，通过解析客户端传入的请求行中的请求方法、URL、协议版本，主要利用 URL 来确定请求的文件名，并把该文件按照 HTTP 格式返回给客户端。这部分代码被封装在 helper.c 的 http_request 函数中。

```
// http请求处理 这里只关注请求行
void http_request(const char* request, int cfd){
    // 拆分http请求行
    char method[12], path[1024], protocol[12];
    sscanf(request, "%[^ ] %[^ ] %[^ ]", method, path, protocol);

    // 转码 将不能识别的中文乱码 -> 中文
    decode_str(path, path);

    char* file = path+1; // 去掉path中的/ 获取访问文件名

    // 如果没有指定访问的资源，默认显示资源目录中的内容
    if(strcmp(path, "/") == 0) {
        file = "./";
    }

    // 获取文件属性
    struct stat st;
    int ret = stat(file, &st);
    if(ret == -1) {
        send_error(cfd, 404, "Not Found", "NO such file or direntry");
        return;
    }

    // 判断是目录还是文件
    // 如果是目录，则读取该目录下所有目录项并发送给客户端
    if(S_ISDIR(st.st_mode)) {
        send_respond_head(cfd, 200, "OK", get_file_type(".html"), -1);
        send_dir(cfd, file);
    }
    // 如果是文件，则读取该文件所有内容并发送给客户端
    else if(S_ISREG(st.st_mode)) {
        send_respond_head(cfd, 200, "OK", get_file_type(file), st.st_size);
        send_file(cfd, file);
    }
}
```

如上图，通过解析请求行中的 method、path、protocol，使用 path 去掉 ‘/’ 来得到文件名，判断该文件名对应的文件是否存在，如果文件不存在，则回传一个 404 的错误页面，如果文件存在，则进一步判断该文件名对应的是文件还是目录从而构造不同的响应报文发送给客户端。

2.2.2 HTTP 构造响应



如上图是 HTTP 响应报文的格式，在该项目中实现简易的 Web 文件服务器，通过上述的解析已经得到了需要传输的文件，只需要按照格式构造响应报文即可，即依次填充状态行、响应头部、空行、响应正文四个部分即可。对应的代码如下。

```
// 发送响应头
void send_respond_head(int cfd, int no, const char* desp, const char* type, long len){
    char buf[1024] = {0};
    // 状态行
    sprintf(buf, "http/1.1 %d %s\r\n", no, desp);
    send(cfd, buf, strlen(buf), 0);
    // 消息报头
    sprintf(buf, "Content-Type:%s\r\n", type);
    sprintf(buf+strlen(buf), "Content-Length:%ld\r\n", len);
    send(cfd, buf, strlen(buf), 0);
    // 空行
    send(cfd, "\r\n", 2, 0);
}
```

如上图是构造响应头的代码，通过 sprintf 来拼接字符串，发送协议版本，状态码，状态码描述三个字段构成的状态行给客户端，响应头部中有两个比较重要的字段分别是 Content-Type 和 Content-Length，对应响应正文的类型和大小，这里可以通过文件的 stat 来获取。

如下是发送文件的代码，首先通过 open 函数以只读方式打开文件，可以通过返回值判断文件是否成功打开，异常需要返回错误给客户端。然后通过循环读取文件中的内容并调用 send 发送给客户端。这里需要对 send 函数的返回值进行判断，在项目编写过程中，由于中断导致文件发送过程打断造成的错误，需要重新继续发送。在项目编写过程中，发送 MP3 和视频时经常产生 connection reset by peer 的问题，经查阅资料是发送的内容和 Content-Length 长度不一致造成的，多发了数据但对端关闭就会产生此问题。所以此时应跳出循环，发送文件过程结束。

发送目录的过程需要读取该目录的目录项并填写到 HTML 页面中返回，大致

逻辑与发送文件类似，不详细阐述，可以阅读源代码中的 send_dir 函数。

```
// 发送文件
void send_file(int cfd, const char* filename){
    // 打开文件
    int fd = open(filename, O_RDONLY);
    if(fd == -1) {
        send_error(cfd, 404, "Not Found", "NO such file or direntry");
        exit(1);
    }

    // 循环读文件
    char buf[4096] = {0};
    int len = 0, ret = 0;
    while( (len = read(fd, buf, sizeof(buf))) > 0 ) {
        // 发送读出的数据
        ret = send(cfd, buf, len, 0);
        if (ret == -1) {
            if (errno == EAGAIN || errno == EINTR) {
                continue;
            }else if(errno == ECONNRESET){
                break;
            }
            else {
                perror("sesssnd error:");
                exit(1);
            }
        }
    }
    if(len == -1) {
        perror("read file error");
        exit(1);
    }

    close(fd);
}
```

如下是发送错误的过程，需要从状态行开始构造，填入状态和状态码描述，然后填充 Content-Type、Content-Length、Connection 等响应头部，最后拼接一个错误的页面，涉及部分 html 的知识。

```
// 发送错误信息
void send_error(int cfd, int status, char *title, char *text){
    char buf[4096] = {0};

    sprintf(buf, "%s %d %s\r\n", "HTTP/1.1", status, title);
    sprintf(buf+strlen(buf), "Content-Type:%s\r\n", "text/html");
    sprintf(buf+strlen(buf), "Content-Length:%d\r\n", -1);
    sprintf(buf+strlen(buf), "Connection: close\r\n");
    send(cfd, buf, strlen(buf), 0);
    send(cfd, "\r\n", 2, 0);

    memset(buf, 0, sizeof(buf));

    sprintf(buf, "<html><head><title>%d %s</title></head>\n", status, title);
    sprintf(buf+strlen(buf), "<body bgcolor=\"#cc99cc\"><h2 align=\"center\">%d %s</h4>\n", status, title);
    sprintf(buf+strlen(buf), "%s\n", text);
    sprintf(buf+strlen(buf), "<hr>\n</body>\n</html>\n");
    send(cfd, buf, strlen(buf), 0);

    return ;
}
```

2.3 各体系 Web 服务器设计与实现

2.3.1 迭代服务器

```
// 提示使用方式 获取服务器端口和映射目录
if (argc != 3){
    fprintf(stderr, "usage: %s <port> <path>\n", argv[0]);
    exit(1);
}
int port = atoi(argv[1]);

ret = chdir(argv[2]);
if(ret == -1){
    perror("chdir error");
    exit(1);
}
```

如上图是程序启动时参数的判断，IP 地址是由服务器调用 `INADDR_ANY` 获取的，所以不需要手动传入。需要传入的两个参数分别是 `port` 和 `path`。`port` 是服务器监听的端口地址，`path` 是映射本地的文件路径。

```
// 建立listenfd套接字
listenfd = socket(AF_INET, SOCK_STREAM, 0);
if(listenfd == -1){
    perror("socket error");
    exit(1);
}

// 设置属性
bzero(&servaddr, sizeof(servaddr));
servaddr.sin_family = AF_INET;
servaddr.sin_addr.s_addr = htonl(INADDR_ANY);
servaddr.sin_port = htons(port);

// 设置地址重用
int opt = 1;
setsockopt(listenfd, SOL_SOCKET, SO_REUSEADDR, (const char*)&opt, sizeof(opt));

// bind
ret = bind(listenfd, (struct sockaddr *) &servaddr, sizeof(servaddr));
if(ret == -1){
    perror("bind error");
    exit(1);
}

// listen
ret = listen(listenfd, 256);
if(ret == -1){
    perror("listen error");
    exit(1);
}
```

如上图是初始化套接字的过程，依次调用 `socket`、`bind`、`listen` 等函数接口，获取初始化的套接字。

```

//迭代服务器
while(1){
    //accept
    clilen = sizeof(cliaddr);
    connfd = accept(listenfd, (struct sockaddr *) &cliaddr, &clilen);
    if (connfd == -1) {
        perror("accept error");
        exit(1);
    }
    //打印发起连接的客户端信息
    char str[INET_ADDRSTRLEN];
    printf("New Client: IP %s, PORT %d\n",
        inet_ntop(AF_INET, &cliaddr.sin_addr, str, sizeof(str)),
        ntohs(cliaddr.sin_port));

    do_process(connfd);

    close(connfd);
}

close(listenfd);

return 0;
}

```

如上图是迭代服务器的组织形式，调用 `accept` 函数接受客户端的请求，打印客户端的 IP 和 Port 信息，调用 `do_process` 函数进行 http 的解析与响应，处理完后调用 `close` 关闭对应的套接字，然后再回到循环开始等待客户端请求。

2.3.2 多进程服务器

```

while(1) {
    clilen = sizeof(cliaddr);
    connfd = accept(listenfd, (struct sockaddr *) &cliaddr, &clilen);
    if (connfd == -1) {
        perror("accept error");
        exit(1);
    }

    // 创建进程，根据返回值确定父子进程
    pid = fork();
    // 子进程负责进行处理
    if(pid == 0){
        // 关闭listenfd套接字
        close(listenfd);

        // 打印发起连接的客户端信息
        char str[INET_ADDRSTRLEN];
        printf("New Client: IP %s, PORT %d\n",
            inet_ntop(AF_INET, &cliaddr.sin_addr, str, sizeof(str)),
            ntohs(cliaddr.sin_port));

        do_process(connfd);
        close(connfd);
        return 0;
    }
    // 父进程负责接受请求
    else if(pid > 0){
        close(connfd);
        signal(SIGCHLD, sig_chld);
    }
    else{
        perror("fork error");
        exit(1);
    }
}

```

如上图是多进程服务器，主进程负责调用 `accept` 函数接受客户端的连接，子进程负责进行相关的业务功能。在调用 `accept` 之后调用 `fork` 函数生成子进程，子进程关闭 `listenfd` 套接字，然后打印客户端的信息，调用 `do_process` 函数进行 `http` 的解析与响应，后调用 `close` 关闭套接字。父进程需要调用 `signal` 函数处理函数来回收子进程，打印子进程退出的信息，还可以避免僵尸进程的产生。

```
void sig_chld(int signo) {
    pid_t  pid;
    int     stat;

    while((pid = waitpid(-1, &stat, WNOHANG)) > 0){
        printf("child %d terminated\n", pid);
    }
    return;
}
```

如上图是 `sig_chld` 函数，该函数调用 `waitpid` 来回收子进程，打印子进程退出的信息。

2.3.3 多线程服务器

```
while(1) {
    clilen = sizeof(cliaddr);
    connfd = accept(listenfd, (struct sockaddr *) &cliaddr, &clilen);
    if (connfd == -1) {
        perror("accept error");
        exit(1);
    }

    ts[i].cliaddr = cliaddr;
    ts[i].connfd = connfd;

    pthread_create(&tid, NULL, do_work, (void*)&ts[i]);
    pthread_detach(tid); // 子线程分离，避免僵尸进程。
    i++;
}
}
```

如上图是多线程服务器的主体框架，调用 `accept` 函数后用 `ts` 数组存储客户端信息，用于传递信息给子线程。调用 `pthread_create` 函数来创建线程，传入参数，调用 `do_work` 函数，`do_work` 函数在后续会展示，调用 `pthread_detach` 来进行父子线程分离，避免产生资源忘记回收的现象。线程计数 `i` 自增。


```

void * do_work(void *arg){
    struct cl_info *ts = (struct cl_info*)arg;

    // 打印发起连接的客户端信息
    char str[INET_ADDRSTRLEN];
    printf("New Client: IP %s, PORT %d\n",
        inet_ntop(AF_INET, &(*ts).cliaddr.sin_addr, str, sizeof(str)),
        ntohs((*ts).cliaddr.sin_port));

    do_process(ts->connfd);

    close(ts->connfd);

    return (void *)0;
}

```

如上图是子线程执行的主体框架，先打印对端客户端的 IP 和端口信息，然后调用 do_process 函数进行 http 的解析和响应，调用 close 关闭套接字。

2.3.4 IO 复用（Select）服务器

```

maxfd = listenfd;

// 将所有的client初值设置为-1
maxi = -1;
for(i = 0; i < FD_SETSIZE; i++)
    client[i] = -1;

// 将listenfd添加到allset中
FD_ZERO(&allset);
FD_SET(listenfd, &allset);

```

如上图，是 Select 服务器中，将 listenfd 加入到监听套接字列表。Maxfd 记录目前 select 数组中使用的最大套接字对应的数值，通过使用该变量来减少遍历的范围。然后将未使用的 client 数组中的元素置位为-1，表示未使用。调用 FD_ZERO 函数将 allset 清空，调用 FD_SET 将 listenfd 添加到 allset 中。

```

while(1){
    rset = allset;
    nready = select(maxfd+1, &rset, NULL, NULL, NULL);
}

```

如上图，是循环开始时，需要给 rset 重新赋值，因为 Select 调用返回后会改变原先的状态，所以需要重新赋值。调用 select 对套接字可读状态监听。

```

//如果就绪描述符是listenfd
if(FD_ISSET(listenfd,&rset)){
    clilen = sizeof(cliaddr);
    connfd = accept(listenfd,(struct sockaddr *)&cliaddr,&clilen);
    if (connfd == -1) {
        perror("accept error");
        exit(1);
    }

    //打印发起连接的客户端信息
    char str[INET_ADDRSTRLEN];
    printf("New Client: IP %s, PORT %d\n",
        inet_ntop(AF_INET, &cliaddr.sin_addr, str, sizeof(str)),
        ntohs(cliaddr.sin_port));

    // 为connfd找到一个存放的位置
    for(i = 0; i < FD_SETSIZE; i++){
        if(client[i]<0){
            client[i] = connfd;
            break;
        }
    }

    if(i == FD_SETSIZE){
        perror("too many clients");
        exit(1);
    }

    FD_SET(connfd,&allset);
    if(connfd > maxfd)
        maxfd = connfd;

    if(i > maxi)
        maxi = i;

    if(--nready <= 0)
        continue;
}

```

如上图，调用 FD_ISSET 来判断套接字是否已经就绪，如果 listenfd 就绪，则处理客户端的请求信息，打印客户端的 IP 和 Port，然后为该 connfd 在 client 数组中寻找一个合适的位置进行存放，更新 maxfd 和 maxi 的信息。

```

for(i = 0; i <= maxi; i++){
    if((sockfd = client[i]) < 0)
        continue;

    // 如果就绪描述符是客户描述符
    if(FD_ISSET(sockfd,&rset)){
        do_process(sockfd);

        FD_CLR(sockfd,&allset);

        if(--nready <= 0)
            break;
    }
}

```

如上图，遍历 client 数组，依次比对套接字是否就绪，对于就绪的套接字，调用 do_process 函数进行 http 的解析和响应构造，在执行完后需要调用 FD_CLR 函数将该套接字从 allset 中剔除。

2.3.5 流水线服务器

```
// 流水线式Web服务器
int main(int argc, char **argv)
{
    int port = parse_argc(argc,argv);

    int listenfd = init_server(port);

    while(1){
        int connfd = do_accept(listenfd);

        do_process(connfd);

        disconnect(connfd);
    }
    close(listenfd);

    return 0;
}
```

流水线服务器即将各个功能进行函数化，依次调用函数进行处理，这里主要是将迭代服务器修改为流水线式服务器，parse_argc 对应于迭代服务器中输入参数的处理，init_server 对应于迭代服务器中的 socket、bind、listen 等函数，do_accept 对应于获取客户端连接并打印客户端的 IP 和 Port，do_process 进行 http 的解析与响应，disconnect 对应于 close 关闭客户端套接字。

```
int do_accept(int listenfd){
    int connfd;
    socklen_t clilen;
    struct sockaddr_in cliaddr;

    //accept
    clilen = sizeof(cliaddr);
    connfd = accept(listenfd, (struct sockaddr *) &cliaddr, &clilen);
    if (connfd == -1) {
        perror("accept error");
        exit(1);
    }
    //打印发起连接的客户端信息
    char str[INET_ADDRSTRLEN];
    printf("New Client: IP %s, PORT %d\n",
        inet_ntop(AF_INET, &cliaddr.sin_addr, str, sizeof(str)),
        ntohs(cliaddr.sin_port));

    return connfd;
}
```

2.3.6 线程池服务器

```
// 线程池式Web服务器
int main(int argc, char **argv)
{
    int port = parse_argc(argc,argv);

    int listenfd = init_server(port);

    // 创建线程池，池里最小10个线程，最大100，队列最大100
    threadpool_t *thp = threadpool_create(10,100,100);
    printf("ThreadPool Initied\n");

    // 主进程负责接受并处理请求，任务交给线程池中的线程进行处理
    while(1){
        int connfd = do_accept(listenfd);

        //向线程池中添加任务
        threadpool_add(thp, process, (void*)&connfd);

        // connfd会在线程池中被关闭，这里不要调用close()
    }

    close(listenfd);
    // 销毁线程池
    threadpool_destroy(thp);

    return 0;
}
```

如上图是线程池式服务器，主要由流水线式进行修改而来。在调用 `init_server` 后调用 `threadpool_create` 创建线程池，在调用 `do_accept` 之后调用 `threadpool_add` 函数将任务添加到线程池中进行处理，在服务器关闭时应该关闭 `listenfd` 和调用 `threadpool_destroy` 函数销毁线程池。

```
struct threadpool_t {
    pthread_mutex_t lock; /* 用于锁住本结构体 */
    pthread_mutex_t thread_counter; /* 记录忙状态线程个数de值 -- busy_thr_num */

    pthread_cond_t queue_not_full; /* 当任务队列满时，添加任务的线程阻塞，等待此条件变量 */
    pthread_cond_t queue_not_empty; /* 任务队列不为空时，通知等待任务的线程 */

    pthread_t *threads; /* 存放线程池中每个线程的tid，数组 */
    pthread_t adjust_tid; /* 管理线程tid */
    threadpool_task_t *task_queue; /* 任务队列(数组首地址) */

    int min_thr_num; /* 线程池最小线程数 */
    int max_thr_num; /* 线程池最大线程数 */
    int live_thr_num; /* 当前存活线程个数 */
    int busy_thr_num; /* 忙状态线程个数 */
    int wait_exit_thr_num; /* 要销毁的线程个数 */

    int queue_front; /* task_queue队头下标 */
    int queue_rear; /* task_queue队尾下标 */
    int queue_size; /* task_queue中实际任务数 */
    int queue_max_size; /* task_queue队列可容纳任务数上限 */

    int shutdown; /* 标志位，线程池使用状态，true或false */
};
```

线程池被封装到 threadpool.c 中，主要参数可以从上图中看到，该线程池是一个可变长的线程池。通过计算忙状态和存活状态线程的比例来进行扩容和缩容。但是不会超过或少于定义的最大/最小线程数。

```
/* 创建新线程 算法： 任务数大于最小线程池个数，且存活的线程数少于最大线程个数时 如：30>=10 && 40<100*/
if (queue_size >= MIN_WAIT_TASK_NUM && live_thr_num < pool->max_thr_num) {
    pthread_mutex_lock(&(pool->lock));
    int add = 0;

    /* 一次增加 DEFAULT_THREAD 个线程*/
    for (i = 0; i < pool->max_thr_num && add < DEFAULT_THREAD_VARY
        && pool->live_thr_num < pool->max_thr_num; i++) {
        if (pool->threads[i] == 0 || !is_thread_alive(pool->threads[i])) {
            pthread_create(&(pool->threads[i]), NULL, threadpool_thread, (void *)pool);
            add++;
            pool->live_thr_num++;
        }
    }

    pthread_mutex_unlock(&(pool->lock));
}

/* 销毁多余的空闲线程 算法：忙线程*2 小于 存活的线程数 且 存活的线程数 大于 最小线程数时*/
if ((busy_thr_num * 2) < live_thr_num && live_thr_num > pool->min_thr_num) {

    /* 一次销毁DEFAULT_THREAD个线程，随机10个即可 */
    pthread_mutex_lock(&(pool->lock));
    pool->wait_exit_thr_num = DEFAULT_THREAD_VARY; /* 要销毁的线程数 设置为10 */
    pthread_mutex_unlock(&(pool->lock));

    for (i = 0; i < DEFAULT_THREAD_VARY; i++) {
        /* 通知处在空闲状态的线程，他们会自行终止*/
        pthread_cond_signal(&(pool->queue_not_empty));
    }
}
}
```

线程池的运作方式是一个生产者-消费者模型，线程池服务器属于生产者，将 http 的解析与响应任务放入到一个缓冲区（消息队列）中，线程池中的线程属于消费者，将 http 的解析与响应任务消费掉，在无任务时，线程池中的线程阻塞在队列为空的条件下。满任务时，服务器生产阻塞在队列为满的条件下。

```
/* 从任务队列里获取任务，是一个出队操作*/
task.function = pool->task_queue[pool->queue_front].function;
task.arg = pool->task_queue[pool->queue_front].arg;

pool->queue_front = (pool->queue_front + 1) % pool->queue_max_size; /* 出队，模拟环形队列 */
pool->queue_size--;

/* 通知可以有新的任务添加进来*/
pthread_cond_broadcast(&(pool->queue_not_full));

/* 任务取出后，立即将 线程池资源 释放*/
pthread_mutex_unlock(&(pool->lock));

/* 执行任务*/
printf("thread 0x%x start working\n", (unsigned int)pthread_self());
pthread_mutex_lock(&(pool->thread_counter)); /* 忙状态线程数变量锁*/
pool->busy_thr_num++; /* 忙状态线程数+1*/
pthread_mutex_unlock(&(pool->thread_counter));

(*(task.function))(task.arg); /* 执行回调函数任务*/
//task.function(task.arg); /* 执行回调函数任务*/

/* 任务结束处理*/
printf("thread 0x%x end working\n", (unsigned int)pthread_self());
pthread_mutex_lock(&(pool->thread_counter));
pool->busy_thr_num--; /* 处理掉一个任务，忙状态线程数-1*/
pthread_mutex_unlock(&(pool->thread_counter));
}
```

3 测试对比

3.1 Web 服务器功能测试

```
→ webservers gcc -c helper.c
→ webservers gcc -c IterativeServer.c
→ webservers gcc -c MultiProcessServer.c
→ webservers gcc -c MultiThreadServer.c -pthread
→ webservers gcc -c PipelineServer.c
→ webservers gcc -c SelectServer.c
→ webservers gcc -c threadpool.c
→ webservers gcc -c ThreadPoolServer.c
→ webservers
```

首先需要对源代码进行编译，生成.o 文件

```
→ webservers gcc -o IterativeServer IterativeServer.o helper.o
→ webservers gcc -o MultiProcessServer MultiProcessServer.o helper.o
→ webservers gcc -o MultiThreadServer MultiThreadServer.o helper.o -pthread
→ webservers gcc -o PipelineServer PipelineServer.o helper.o
→ webservers gcc -o SelectServer SelectServer.o helper.o
→ webservers gcc -o ThreadPoolServer ThreadPoolServer.o helper.o threadpool.o -pthread
→ webservers
```

然后通过-o 选项生成可执行文件，这里需要注意涉及到线程时，需要添加-pthread 选项。

```
→ webservers ./MultiProcessServer 29999 /home/lxw/resources
```

如上图，以多进程服务器为例，图上是运行程序的方式，其它服务器同理，表示在 29999 端口上进行监听，将本地的“/home/lxw/resources”路径映射到 web 服务器上。

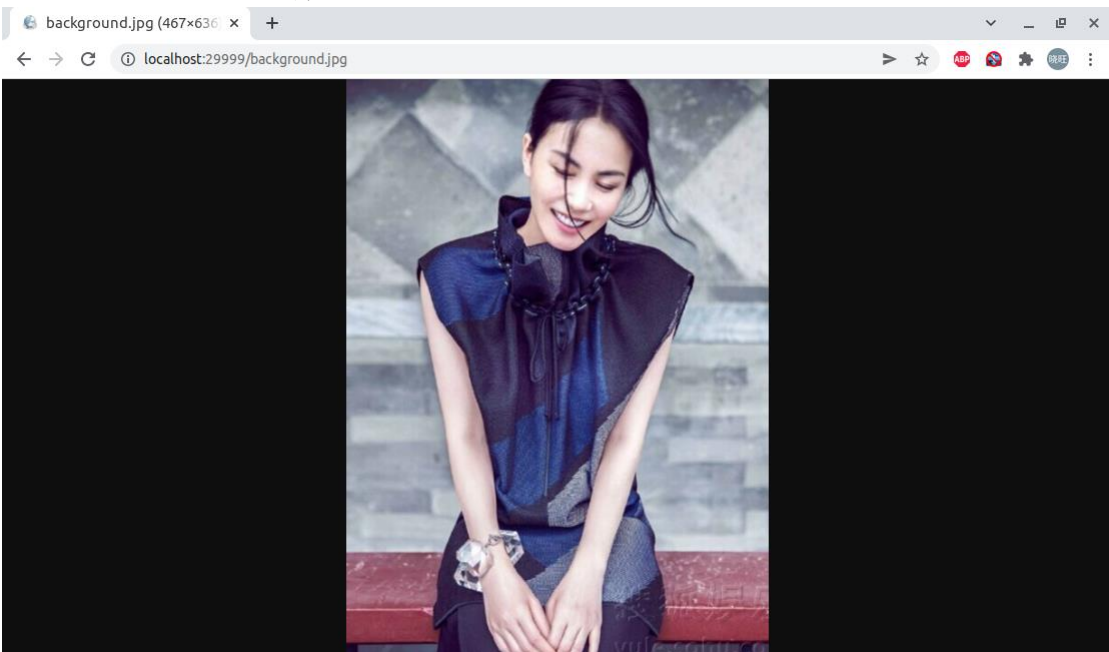
```
New Client: IP 127.0.0.1, PORT 48330
New Client: IP 127.0.0.1, PORT 48334
===== 请求头 =====
请求行数据: GET / HTTP/1.1
===== Process Finish =====
child 206193 terminated
===== 请求头 =====
请求行数据: GET /favicon.ico HTTP/1.1
===== Process Finish =====
child 206194 terminated
```

如上图，在客户端访问 <http://localhost:29999/> 时，服务器接收到两个请求，一个是客户端请求的路径，另一个是浏览器请求的图标信息，服务器回传对应的文件给客户端，这里 “/” 默认是传输映射地址的根目录给客户端。

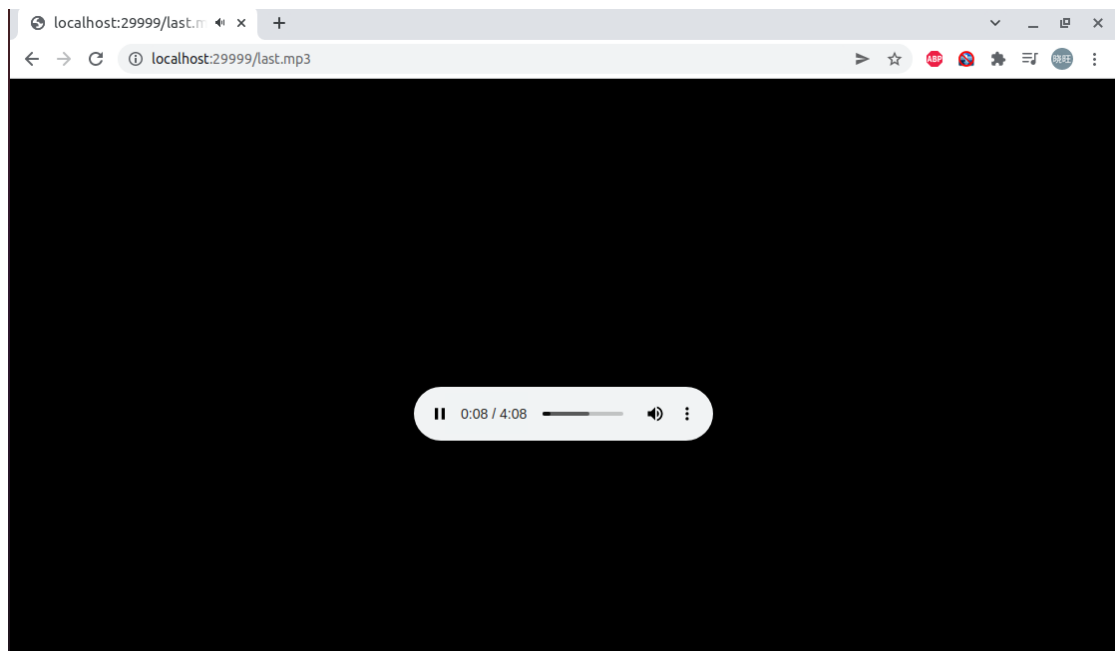


如下图是通过浏览器访问对应地址得到的响应信息，由于缓存的原因，图上显示颜色存在差异。当我们点击对应的文件时，服务器就会传输对应的文件给客户端，由于遵从 html 格式，所以可以直接在浏览器端浏览。

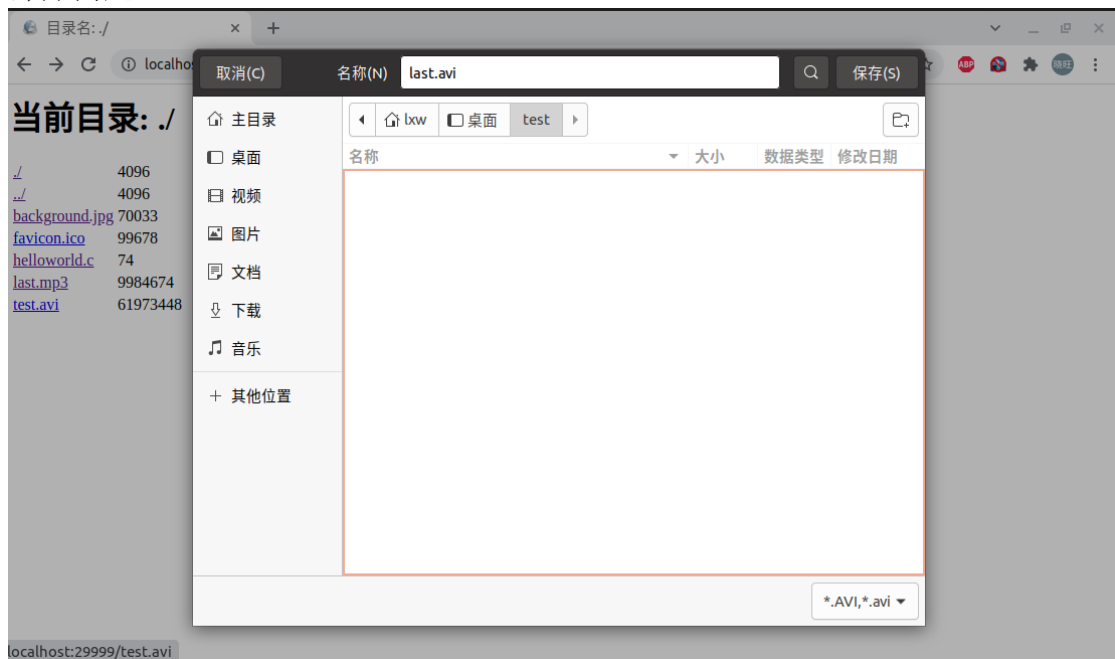
如下图是，客户端请求图片的效果。



从图中可以看出客户端可以正常得到服务器发送的图片并进行响应渲染到浏览器上。



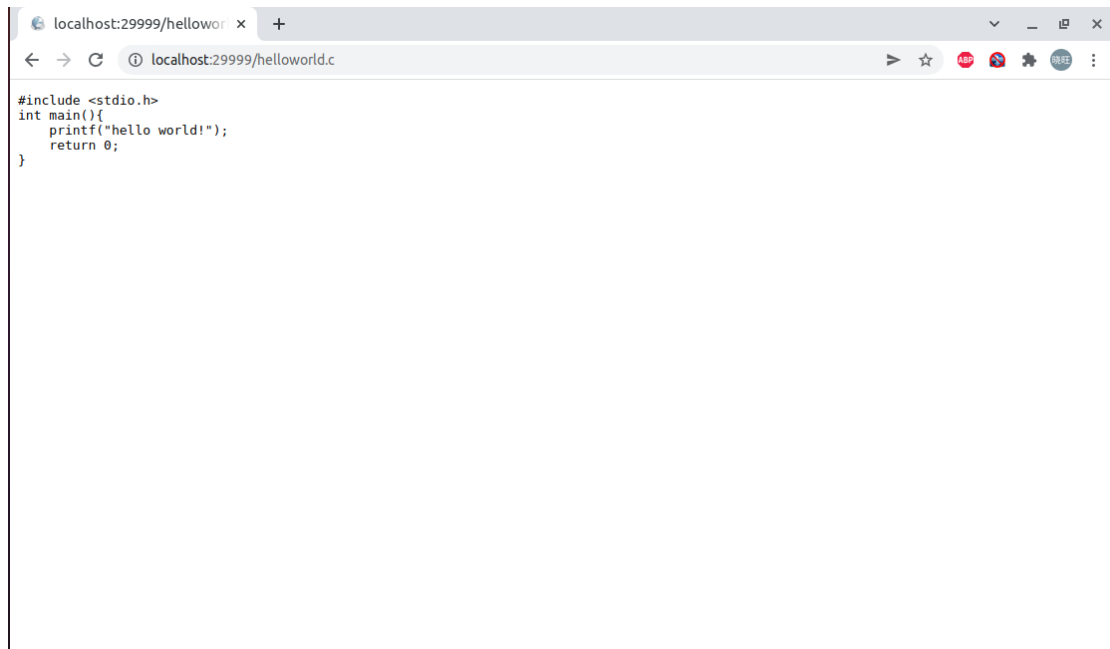
如上图，是客户端请求 MP3 文件时，浏览器会得到一个音乐播放器，经测试音乐可以正常播放，但由于未设置相关组件的属性，所以组件的使用上可能存在部分问题。



如上图，是客户端请求 Avi 文件时，浏览器会自动触发下载界面，经测试把视频下载之后可以正常播放。

如下图，是客户端请求文本文件时，浏览器可以直接展示对应的文本，如下是浏览器请求了一个 helloworld 程序。

经测试其余几个体系结构的 Web 服务器都可以正常使用，这里不再进行演示。

A screenshot of a web browser window. The address bar shows 'localhost:29999/helloworld.c'. The page content displays a C program source code:

```
#include <stdio.h>
int main(){
    printf("hello world!");
    return 0;
}
```

3.2 Web 服务器性能测试

对各种 Web 体系架构服务器做性能测试使用的是 Siege 程序，Siege 是一个压力测试和评测工具，设计用于 WEB 开发这评估应用在压力下的承受能力：可以根据配置对一个 WEB 站点进行多用户的并发访问，记录每个用户所有请求过程的相应时间，并在一定数量的并发访问下重复进行。

这里的性能测试分别在两台虚拟机上进行，一台虚拟机执行服务器程序，另一台虚拟机调用 Siege 程序访问服务器的服务进行测试。

-C,或-config在屏幕上打印显示出当前的配置,配置是包括在他的配置文件\$HOME/.siegerc中,可以编辑里面的参数,这样每次siege都会按照它运行.

-v 运行时能看到详细的运行信息

-c n,或-concurrent=n模拟有n个用户在同时访问,n不要设得太大,因为越大,siege消耗本地机器的资源越多

-i,-internet 随机访问urls.txt中的url列表项,以此模拟真实的访问情况(随机性),当urls.txt存在是有效

-d n,-delay=n hit每个url之间的延迟,在0-n之间

-r n,-reps=n 重复运行测试n次,不能与 -t同时存在

-t n,-time=n 持续运行siege 'n'秒(如10S),分钟(10M),小时(10H)

-l 运行结束,将统计数据保存到日志文件中siege.log,一般位于/usr/local/var/siege.log中,也可在.siegerc中自定义

-R SIEGERC,-rc=SIEGERC 指定用特定的siege配置文件来运行,默认的为\$HOME/.siegerc

-f FILE, -file=FILE 指定用特定的urls文件运行siege ,默认为urls.txt,位于siege安装目录下的etc/urls.txt

-u URL,-url=URL 测试指定的一个URL,对它进行"siege",此选项会忽略有关urls文件的设定

urls.txt文件: 是很多行待测试URL的列表以换行符断开,格式为:

如上图是该性能测试程序支持的部分参数。

Transactions://完成多少次处理

Availability: //成功率%

Elapsed time://总共使用时间secs

Data transferred://总数据传输M（不包含头数据）

Response time://平均响应时间secs

Transaction rate://平均每秒完成多少次处理trans/sec

Throughput://平均每秒传送数据MB/sec

Concurrency: //实际最高并发连接数

Successful transactions://成功处理次数

Failed transactions: //失败处理次数

Longest transaction://满足一个请求所需最长时间

Shortest transaction://满足一个请求所需最短时间

如上图是该性能测试程序返回的测试结果与解释。

```
[root@node1 siege-4.1.1]# siege -c 2 -r 25 -d 1 -f site.url
[alert] Zip encoding disabled; siege requires zlib support to enable it
** SIEGE 4.1.1
** Preparing 2 concurrent users for battle.
The server is now under siege...
http/1.1 200      0.03 secs: 984674 bytes ==> GET /last.mp3
http/1.1 200      0.03 secs: 33 bytes ==> GET /background.jpg
http/1.1 200      0.00 secs: 4 bytes ==> GET /helloworld.c
http/1.1 200      0.00 secs: 33 bytes ==> GET /background.jpg
http/1.1 200      0.00 secs: 4 bytes ==> GET /helloworld.c
http/1.1 200      0.06 secs: 984674 bytes ==> GET /last.mp3
http/1.1 200      0.09 secs: 984674 bytes ==> GET /last.mp3
http/1.1 200      0.00 secs: 33 bytes ==> GET /background.jpg
http/1.1 200      0.00 secs: 4 bytes ==> GET /helloworld.c
http/1.1 200      0.00 secs: 33 bytes ==> GET /background.jpg
```

如上图是迭代服务器的测试条件，表示 2 个客户端，每隔 1s 进行一次 url 访问，每一个客户端进行 25 次访问，即总共访问 50 次。

```
Transactions:          50 hits
Availability:         100.00 %
Elapsed time:          14.50 secs
Data transferred:     15.96 MB
Response time:         0.02 secs
Transaction rate:      3.45 trans/sec
Throughput:            1.10 MB/sec
Concurrency:           0.06
Successful transactions: 50
Failed transactions:   0
Longest transaction:   0.09
Shortest transaction:  0.00
```

如上图是测试结果，即总用时 14.5s，传输了 15.96M，平均响应时间 0.02s，

每秒处理 3.45 次请求，平均每秒传输 1.1MB，实际最高并发连接数为 0.06，满足一个请求最长时间为 0.09s，最短时间为 0.00s。

为了做对比分析，所以将传输相同的文件，进行测试。测试条件如下图：

```
[root@node1 siege-4.1.1]# siege -c 2 -r 25 -d 2 -f site.url
[alert] Zip encoding disabled; siege requires zlib support to enable it
** SIEGE 4.1.1
** Preparing 2 concurrent users for battle.
The server is now under siege...
http/1.1 200      0.05 secs:  984674 bytes ==> GET  /last.mp3
http/1.1 200      0.07 secs:  984674 bytes ==> GET  /last.mp3
http/1.1 200      0.04 secs:  984674 bytes ==> GET  /last.mp3
http/1.1 200      0.05 secs:  984674 bytes ==> GET  /last.mp3
http/1.1 200      0.08 secs:  984674 bytes ==> GET  /last.mp3
http/1.1 200      0.05 secs:  984674 bytes ==> GET  /last.mp3
http/1.1 200      0.02 secs:  984674 bytes ==> GET  /last.mp3
http/1.1 200      0.04 secs:  984674 bytes ==> GET  /last.mp3
```

迭代服务器

```
Transactions:                49 hits
Availability:                98.00 %
Elapsed time:                 30.12 secs
Data transferred:            46.01 MB
Response time:                0.04 secs
Transaction rate:             1.63 trans/sec
Throughput:                   1.53 MB/sec
Concurrency:                  0.07
Successful transactions:      49
Failed transactions:          1
Longest transaction:          0.09
Shortest transaction:         0.02
```

多进程服务器

```
Transactions:                50 hits
Availability:                100.00 %
Elapsed time:                 25.08 secs
Data transferred:            46.95 MB
Response time:                0.04 secs
Transaction rate:             1.99 trans/sec
Throughput:                   1.87 MB/sec
Concurrency:                  0.08
Successful transactions:      50
Failed transactions:          0
Longest transaction:          0.07
Shortest transaction:         0.02
```

多线程服务器

```
Transactions:          50 hits
Availability:          100.00 %
Elapsed time:          27.10 secs
Data transferred:      46.95 MB
Response time:         0.04 secs
Transaction rate:      1.85 trans/sec
Throughput:            1.73 MB/sec
Concurrency:           0.08
Successful transactions: 50
Failed transactions:    0
Longest transaction:   0.09
Shortest transaction:   0.02
```

I0 复用服务器

```
Transactions:          48 hits
Availability:          96.00 %
Elapsed time:          27.10 secs
Data transferred:      45.07 MB
Response time:         0.04 secs
Transaction rate:      1.77 trans/sec
Throughput:            1.66 MB/sec
Concurrency:           0.08
Successful transactions: 48
Failed transactions:    2
Longest transaction:   0.12
Shortest transaction:   0.02
```

流水线服务器

```
Transactions:          47 hits
Availability:          94.00 %
Elapsed time:          34.98 secs
Data transferred:      44.14 MB
Response time:         0.04 secs
Transaction rate:      1.34 trans/sec
Throughput:            1.26 MB/sec
Concurrency:           0.05
Successful transactions: 47
Failed transactions:    3
Longest transaction:   0.10
Shortest transaction:   0.02
```

线程池服务器

```
Transactions:          48 hits
Availability:          96.00 %
Elapsed time:          29.07 secs
Data transferred:      45.07 MB
Response time:         0.04 secs
Transaction rate:      1.65 trans/sec
Throughput:            1.55 MB/sec
Concurrency:           0.07
Successful transactions: 48
Failed transactions:    2
Longest transaction:   0.07
Shortest transaction:   0.02
```

对于上述结果，从每秒处理的请求数来看。

服务器	迭代	多进程	多线程	I0 复用	流水线	线程池
每秒请求数	1.63	1.99	1.85	1.77	1.34	1.65

即流水线 < 迭代 < 线程池 < I0 复用 < 多线程 < 多进程，表明 I0 复用和线程池虽理论上应占优势，但是当并发数不多的场景下反而由于复杂的设计导致性能不如简易的多进程和多线程服务器。