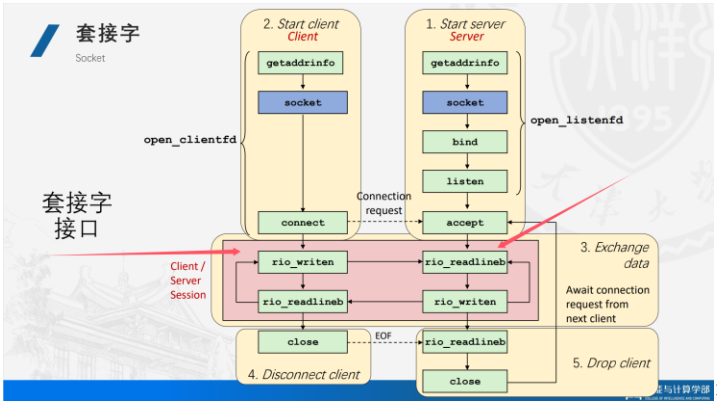
函钩　2025.8.22 proxylab 实验

proxy 意为代理，即作为 web 浏览器和终端服务器之间的中间人。“浏览器不直接连接终端服务器获取网页，而是连接代理，代理将请求转发给终端服务器。当终端服务器回复代理时，代理将回复转发给浏览器。”（来自 README.md）实验要求我们编写 HTTP 代理用于缓存 web 对象。作业总分为 70 分，作业满分要求我们的程序做到“基本正确性、并发性、缓存”基本正确性即为实现 proxy 程序基本逻辑的正确，并发性即为通过线程思想实现并发，缓存即为先查缓存，再代理，再写缓存。以下为程序的实现思路。

在实验附带的小程序“tiny”中，我们可以看到课本中作为案例对一个服务端的实现方式。proxy 的本质既是客户端又是服务器，其要作为服务器接受客户端的请求，又要作为客户端向目标服务器发送请求。



按照课本中的结构示例，proxy 在基本逻辑上要实现双方的功能。tiny.c 已经实现了服务端的基本功能，所以我们可以对 tiny.c 中的 main 函数和 doit 函数进行进一步的修改。

```c
int main(int argc, char **argv)
{
    int listenfd, connfd;
    char hostname[MAXLINE], port[MAXLINE];
    socklen_t clientlen;
    struct sockaddr_storage clientaddr;

    /* Check command line args */
    if (argc != 2) {
    fprintf(stderr, "usage: %s <port>\n", argv[0]);
    exit(1);
    }

    listenfd = Open_listenfd(argv[1]);
    while (1) {
    clientlen = sizeof(clientaddr);
    connfd = Accept(listenfd, (SA *)&clientaddr, &clientlen); //line:netp:tiny:accept
        Getnameinfo((SA *) &clientaddr, clientlen, hostname, MAXLINE,
                    port, MAXLINE, 0);
        printf("Accepted connection from (%s, %s)\n", hostname, port);
    doit(connfd);                                     //line:netp:tiny:doit
    Close(connfd);                                    //line:netp:tiny:close
    }
}
```

先来阅读一下示例的代码。示例代码创建监听套接字文件描述符，连接套接字文件描述符，域名和端

口，客户端地址。根据输入的命令行，使用 open_listenfd 启动监听。在 while(1)的无限循环中，示例代码使用了 clientlen=sizeof(clientaddr)来创建地址缓冲区。connfd=Accept(……)即为接受客户端连接，建立连接后使用 getnameinfo 来输出地址端口号等信息，存入 hostname 和 port 缓冲区。doit 函数为处理 http 连接的核心函数，在处理完一个客户端请求后关闭连接，并处理下一个客户端的请求。tiny 的核心对于本实验的要求来说有一个缺陷，就是单次只能处理一个客户端的请求，无法做到多线程并发处理。为了完成实验，我们需要先了解一下 linux 的多线程编程思想。

在 pthread.h 中包含了 linux 绝大多数关于线程的函数。在 csapp.h 中包含了一个线程创建的包装函数（在 pthread_create 基础上添加了错误判断等简单逻辑）。

```
int main(int argc, char** argv){
    rw = Malloc(sizeof(struct rwlock_t));
    printf("%s", user_agent_hdr);
    pthread_t tid;
    int listenfd, connfd;
    char hostname[MAXLINE], port[MAXLINE];
    socklen_t clientlen;
    struct sockaddr_storage clientaddr;

    rwlock_init();

    if (argc != 2){
        fprintf(stderr, "usage: %s <port>\n", argv[0]);
        exit(0);
    }
    listenfd = Open_listenfd(argv[1]);
    while (1){
        clientlen = sizeof(clientaddr);
        connfd = Accept(listenfd, (SA*)&clientaddr, &clientlen);
        Getnameinfo((SA*)&clientaddr, clientlen, hostname, MAXLINE, port,
MAXLINE, 0);
        printf("Accepted connection from (%s %s)\n", hostname, port);
        Pthread_create(&tid, NULL, thread, (void*)&connfd);
    }
    return 0;
}
```

基于 tiny 中的 main 函数，实验中的 main 函数使用了 Pthread_create 函数，并将处理 http 的核心函数 doit 放置在了 thread 函数中，这样在软件运行时，程序可以并发地处理多个客户端请求了。&tid 为线程 id，记录每个新线程的句柄；NULL 为使用默认线程属性，thread 是线程的入口函数，&connfd 是传给线程的参数：客户端套接字。

```
void thread(void* v){
    int fd = *(int*)v;              //这一步是必要的,v 是 main 中 connfd 的地址，
后续可能被改变，所以必须要得到一个副本
                                    //还要注意不能锁住，因为允许多个一起读
    Pthread_detach(pthread_self()); //设置线程，结束后自动释放资源
    doit(fd);
    close(fd);
}
```

这个是线程入口函数 thread，此处使用了 Pthread_detach 函数，当线程结束后该自动回收线程资源。在每个线程中用 doit 函数实现对 http 请求的处理。相当于将 tiny 代码中 main 函数的 doit 部分放进了一个线程中。

对 main 函数的另一个更改就是添加了读写锁(rwlock)来保护共享缓存。添加了锁结构体：

```
struct rwlock_t{
    sem_t lock;          //基本锁
    sem_t writelock;     //写者所
    int readers;         //读者个数
};
```

和对读写锁结构体的初始化函数。

```
void rwlock_init(){
    rw->readers = 0;
    sem_init(&rw->lock, 0, 1);
    sem_init(&rw->writelock, 0, 1);
}
```

结构体包含三个量：基本锁，写者锁和读者个数。读写锁主要在 doit 函数中为 readcache 函数和 writecache 函数服务。lock 为互斥信号量专门用于原子地维护当前读者数量，writelock 为互斥信号量，负责真正的读写互斥。任何线程读缓存时，先通过 lock 把读者数+1，若发现自己是第一个读者便立即抢占 writelock 以挡住写者，后续读者可并行进入，读完后递减计数，当最后一个读者退出时释放 writelock 允许写者进入；写者则直接一次性获取 writelock ，独占缓存进行替换或更新。该机制实现了 读并行、写独占 的并发策略，既保证数据一致性，又最大化读吞吐量。

先来浏览 tiny.c 中对 doit 函数的实现方式

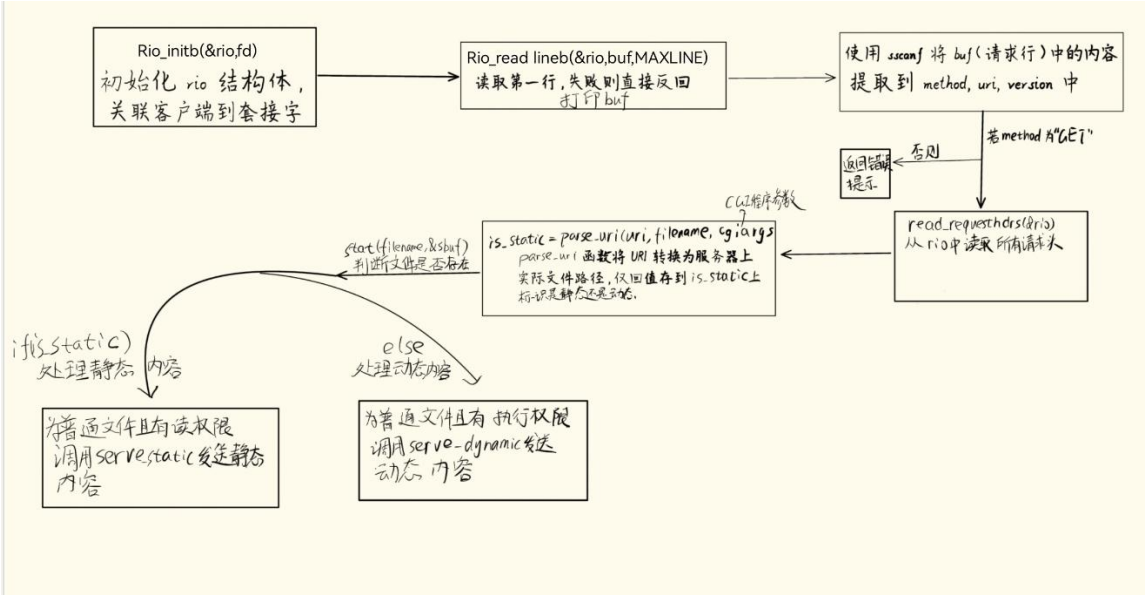
```c
/* $begin doit */
void doit(int fd)
{
    int is_static;
    struct stat sbuf;
    char buf[MAXLINE], method[MAXLINE], uri[MAXLINE], version[MAXLINE];
    char filename[MAXLINE], cgiargs[MAXLINE];
    rio_t rio;

    /* Read request line and headers */
    Rio_readinitb(&rio, fd);
    if (!Rio_readlineb(&rio, buf, MAXLINE))  //line:netp:doit:readrequest
        return;
    printf("%s", buf);
    sscanf(buf, "%s %s %s", method, uri, version);       //line:netp:doit:parserequest
    if (strcasecmp(method, "GET")) {                     //line:netp:doit:beginrequesterr
        clienterror(fd, method, "501", "Not Implemented",
                    "Tiny does not implement this method");
        return;
    }                                                    //line:netp:doit:endrequesterr
    read_requesthdrs(&rio);                              //line:netp:doit:readrequesthdrs

    /* Parse URI from GET request */
    is_static = parse_uri(uri, filename, cgiargs);       //line:netp:doit:staticcheck
    if (stat(filename, &sbuf) < 0) {                     //line:netp:doit:beginnotfound
    clienterror(fd, filename, "404", "Not found",
            "Tiny couldn't find this file");
    return;
    }                                                    //line:netp:doit:endnotfound

    if (is_static) { /* Serve static content */
    if (!(S_ISREG(sbuf.st_mode)) || !(S_IRUSR & sbuf.st_mode)) { //line:netp:doit:readable
        clienterror(fd, filename, "403", "Forbidden",
            "Tiny couldn't read the file");
        return;
    }
    serve_static(fd, filename, sbuf.st_size);        //line:netp:doit:servestatic
    }
    else { /* Serve dynamic content */
    if (!(S_ISREG(sbuf.st_mode)) || !(S_IXUSR & sbuf.st_mode)) { //line:netp:doit:executable
        clienterror(fd, filename, "403", "Forbidden",
            "Tiny couldn't run the CGI program");
        return;
    }
    serve_dynamic(fd, filename, cgiargs);            //line:netp:doit:servedynamic
    }
}
/* $end doit */
```

doit 函数的核心作用是通过接收一个文件描述符 fd（已连接的客户端套接字），然后读取并解析 HTTP 请求，最后根据请求类型返回相应的内容。解析 HTTP 的流程如图所示：



进行流程图梳理之后可以知道 tiny 的 doit 函数主要是通过解析命令行，判断静态或动态内容，然后进行相应处理。tiny 是一个需要响应客户端请求的服务器，根据实验要求，proxy 需要将客户端的请求原样转发给目标服务器，所以不需要进行动态或者静态内容的处理。

根据实验要求，我们可以设想 doit 函数的执行流程：1、关联客户端到套接字 2、解析浏览器发来的 URL 3、把请求行、Host 头等信息重新拼装成新的 HTTP 报文 4、把报文发给真正的目标服务器 5、把目标服务器的响应逐字节读回来，再逐字节写给浏览器

1、关联客户端到套接字的方法和 tiny 大同小异，都是先用 Rio_readinitb()函数初始化 rio_t 结构体，然后再用 Rio_readlineb()从该结构体管理的缓冲区(buf)中读取 HTTP 请求数据。

2、解析 URL 需要仿照 tiny 中的 parse_uri 来写一个自己的函数 parse_url。

```
void parse_url(char* url, struct UrlData* u){
    char* hostpose = strstr(url, "//");
    if (hostpose == NULL){
        char* pathpose = strstr(url, "/");
        if (pathpose != NULL)
            strcpy(u->path, pathpose);
        strcpy(u->port, "80");
        return;
    } else{
        char* portpose = strstr(hostpose + 2, ":");
        if (portpose != NULL){
            int tmp;
```

```
            sscanf(portpose + 1, "%d%s", &tmp, u->path);
            sprintf(u->port, "%d", tmp);
            *portpose = '\0';

        } else{
            char* pathpose = strstr(hostpose + 2, "/");
            if (pathpose != NULL){
                strcpy(u->path, pathpose);
                strcpy(u->port, "80");
                *pathpose = '\0';
            }
        }
        strcpy(u->host, hostpose + 2);
    }
    return;
}
```

函数需要判断多种不同的 url 情况，例如：/home.html　是目前大部分的形式，仅由路径构成，Host 首部字段藏在 HTTP 请求头里 ，由浏览器自动生成并放在请求报文的第二行（第一行是请求行），端口默认 80。又如：http://www.abc.com:8080/home.html 在这种情况下，Host 首部字段为空，我们需要解析域名 www.abc.com，端口：8080，路径：/home.html。

函数 strstr 可以确定某字符串在目标字符串的起始位置，使用 char *hostpose=strstr(char *url ,"//");来确定 url 中的域名位置。若无域名，则定义 char *pathpose 确定"/"的位置，第一个斜线的位置就是路径的位置，并且将路径储存到结构体 UrlData* u 中，端口为默认值 80。

```
/*关于url信息的结构体*/
struct UrlData{
    char host[MAXLINE]; //hostname
    char port[MAXLINE]; //端口
    char path[MAXLINE]; //路径
};
```

若包含了域名，则需确定冒号位置。冒号位置之后就是 port 和路径位置。当存储完 port 和 path 之后，可以使用一个小技巧，把冒号的位置（*portpose）替换为'\0'（结束符）这样 *hostpose+2 指向的剩余字符串刚好就是域名。若未填写端口，则默认端口为 80，确定路径方式同上。

3、把请求行、Host 头等信息重新拼装成新的 HTTP 报文的功能在 change_httpdata()函数中实现。

```
void change_httpdata(rio_t* rio, struct UrlData* u, char* new_httpdata){
    static const char* Con_hdr = "Connection: close\r\n";
    static const char* Pcon_hdr = "Proxy-Connection: close\r\n";
    char buf[MAXLINE];
```

```
    char Reqline[MAXLINE], Host_hdr[MAXLINE], Cdata[MAXLINE];//分别为请求
行，Host 首部字段，和其他不东的数据信息
    sprintf(Reqline, "GET %s HTTP/1.0\r\n", u->path);   //获取请求行
    while (Rio_readlineb(rio, buf, MAXLINE) > 0){
        /*读到空行就算结束，GET 请求没有实体体*/
        if (strcmp(buf, "\r\n") == 0){
            strcat(Cdata, "\r\n");
            break;
        }
        else if (strncasecmp(buf, "Host:", 5) == 0){
            strcpy(Host_hdr, buf);
        }

        else if (strncasecmp(buf, "Connection:", 11) != 0 &&
        strncasecmp(buf, "Proxy-Connection:", 17) != 0 && strncasecmp(buf,
"User-agent:", 11) != 0){
            strcat(Cdata, buf);
        }
    }
    if (!strlen(Host_hdr)){
        /*如果 Host_hdr 为空，说明该 host 被加载进请求行的 URL 中，我们格式读从 URL
中解析的 host*/
        sprintf(Host_hdr, "Host: %s\r\n", u->host);
    }

    sprintf(new_httpdata, "%s%s%s%s%s%s", Reqline, Host_hdr, Con_hdr,
Pcon_hdr, user_agent_hdr, Cdata);
    return;
}
```

这个函数是 proxy 中负责重新构造 HTTP 请求报文的核心函数。该函数将客户端发送的原始 HTTP 请求转换为适合发送给目标服务器的格式。假设用户向代理服务器发送如下内容:

GET http://www.example.com:8080/index.html HTTP/1.1

User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64) Chrome/90.0.4430.93

Accept: text/html,application/xhtml+xml

在函数处理后，我们期望得到

GET /index.html HTTP/1.0　　（单独的路径请求）

Host: www.example.com　　（将 HOST 地址单独摘出来）

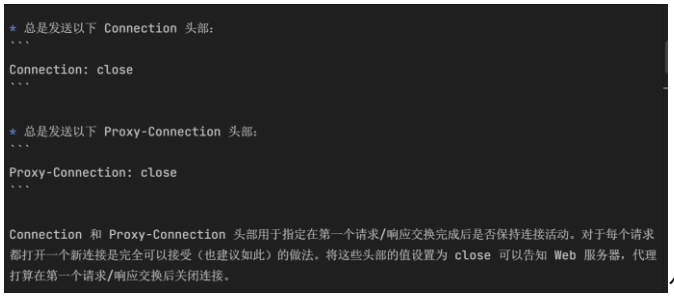
Connection: close　　（指定在第一个请求/响应交换完成后关闭连接活动）

Proxy-Connection: close

User-Agent: Mozilla/5.0 (X11; Linux x86_64; rv:10.0.3) Gecko/20120305 Firefox/10.0.3 （实验可以始终发送这个 User-Agent 来应对所有情况）

Accept: text/html,application/xhtml+xml　（保留原来的内容）

在 README.md 中有提到，



代理服务器需要总是发送 Connection 头部和 Proxy-Connection 头部，所以在函数的开头定义了两个字符指针用于在拼接报文之后输出”Connection : close”和”Proxy-Connection: close”。

在完成准备工作后，首先要做的就是构建新的请求行。请求行由“GET 路径 HTTP/1.0”构成，使用 u->path 直接作为路径即可。接下来在 while 循环内使用 Rio_readlineb 逐行读取 HTTP 请求头，1.当读到空行（”\r\n”）时表示 HTTP 请求头结束，将空行添加到 Cdata 中然后结束循环。2.若读到 Host 头部时，将其储存到 Host_hdr 中。3.对于 Connection:，Proxy-Connection:，User-Agent:以外的特殊字段，将其追加到 Cdata 末尾。请求头读取结束后若 Host_hdr 仍为空，则说明该 host 被加载进请求行的 URL 中（即 GET http://www.example.com:8080/index.html HTTP/1.1 这种格式），使用 sprintf 将 u->host 储存到 Host_hdr 中。最后再使用格式读将所有要素按顺序拼接起来形成新的更符合格式的 HTTP 报文。以上就是 change_httpdata()函数的作用原理。

4、把报文发送给真正的服务器

```
    int server_fd = Open_clientfd(u.host, u.port);
    size_t n;

    Rio_readinitb(&server_rio, server_fd);
    Rio_writen(server_fd, new_httpdata, strlen(new_httpdata));
```

先通过 Open_clientfd(u.host, u.port)函数与目标服务器的 TCP 连接，然后初始化(Rio_readinitb 函数)一个 rio_t 结构体 server_rio，用于后续从服务器读取响应数据。最后通过 Rio_writen(server_fd, new_httpdata, strlen(new_httpdata))函数将 change_httpdata 重新构造的 HTTP 请求报文 new_httpdata 发送到目标服务器。

5、把目标服务器的响应逐字节读回来，再逐字节写给浏览器

```
    char cache[MAX_OBJECT_SIZE];
    int sum = 0;
```

```
    while((n = Rio_readlineb(&server_rio, buf, MAXLINE)) != 0){
        Rio_writen(fd, buf, n);
        sum += n;
        strcat(cache, buf);
    }
    printf("proxy send %ld bytes to client\n", sum);
    if (sum < MAX_OBJECT_SIZE)
        writecache(cache, urltemp); //如果可以的话，读入缓存
    close(server_fd);
    return;
```

为了读取服务器的响应内容，我们需要创建一个缓冲区来暂时储存服务器响应。sum 是字节计数器，在循环中不断累加，求和得到相应得到的总字符数。

Rio_readlineb 的返回值是每一行的字数，当读取到 EOF 时 n=0，则循环停止。在每次循环中，将读取到的循环重新转发给客户端（即之前用 fd 绑定的客户端）。sum 统计累积转发的字节数，然后再将接收到的字节拼接到 cache 后，最后显示出总共转发了多少字节。如果总转发字节小于最大对象限制，则可以将响应写入缓存供后续使用。结束后关闭与目标服务器的连接。**doit 函数的功能解释完成，函数完整代码如下:**

```
void doit(int fd){
    char buf[MAXLINE], method[MAXLINE], url[MAXLINE], version[MAXLINE];
    char new_httpdata[MAXLINE], urltemp[MAXLINE];
    struct UrlData u;
    rio_t rio, server_rio;
    Rio_readinitb(&rio, fd);
    Rio_readlineb(&rio, buf, MAXLINE);

    sscanf(buf, "%s %s %s", method, url, version);
    strcpy(urltemp, url);   //赋值 url 副本以供读者写者使用，因为在解析 url 中，
url 可能改变

    /*只接受 GEI 请求*/
    if (strcmp(method, "GET") != 0){
        printf ("The proxy can not handle this method: %s\n", method);
        return;
    }

    if (readcache(fd, urltemp) != 0)    //如果读者读取缓存成功的话，直接返回
        return;

    parse_url(url, &u);     //解析 url
```

```
    change_httpdata(&rio, &u, new_httpdata);    //修改 http 数据，存入
new_httpdata 中

    int server_fd = Open_clientfd(u.host, u.port);
    size_t n;

    Rio_readinitb(&server_rio, server_fd);
    Rio_writen(server_fd, new_httpdata, strlen(new_httpdata));

    char cache[MAX_OBJECT_SIZE];
    int sum = 0;
    while((n = Rio_readlineb(&server_rio, buf, MAXLINE)) != 0){
        Rio_writen(fd, buf, n);
        sum += n;
        strcat(cache, buf);
    }
    printf("proxy send %ld bytes to client\n", sum);
    if (sum < MAX_OBJECT_SIZE)
        writecache(cache, urltemp); //如果可以的话，读入缓存
    close(server_fd);
    return;
}
```

关于 writecache()和 readcache()函数

```
void writecache(char* buf, char* key){
    sem_wait(&rw->writelock);   //需要等待获得写者锁
    int index;
    /*利用时钟算法，当前指针依次增加，寻找 used 字段为 0 的对象*/
    /*如果当前 used 为 1，则设置为 0，最坏情况下需要 O(N)时间复杂度*/
    while (cache[nowpointer].used != 0){
        cache[nowpointer].used = 0;
        nowpointer = (nowpointer + 1) % MAX_CACHE;
    }

    index = nowpointer;

    cache[index].used = 1;
    strcpy(cache[index].key, key);
    strcpy(cache[index].value, buf);
    sem_post(&rw->writelock);   //释放锁
    return;
```

```
}
```

writecache 函数采用了类似 LRU 最近最少使用算法，以及读写锁机制。 在程序的开头我们定义了读写锁结构体，一直到读写 cache 这两个函数，读写锁才真正发挥作用。sem_wait 是一个等待函数，用于对信号量进行 P 操作(减少/等待操作)，函数等待直到获得写者锁（rw->writelock==1），当信号量值为 0 时则线程阻塞等待，如果大于 0，则将信号量值减 1 并继续执行。类似的，函数结尾处的 sem_pos(&rw->writelock)是 V 操作(释放/增加操作)，用于释放锁并将信号量值加 1。综上所述，rw->writelock==1 时是可写的，==0 时是不可写的，>1 时是异常情况，不应当出现。

当该线程可写时，使用时钟算法来寻找可用的缓存槽来存储新的 HTTP 响应数据，实现缓存替换策略。我们定义一个 int index 来记录缓存位置。变量 nowpointer 是一个 int 类型全局变量。cache 的结构体定义如下

```
struct Cache{
    int  used;          //最近被使用过则为 1，否则为 0
    char key[MAXLINE];  //URL索引
    char value[MAX_OBJECT_SIZE];    //URL所对应的缓存
};
```

，该结构体含有一个 used 标记位用来表示最近是否被使用过。在 while 循环中，cache[nowpointer].used!=0 作为了一个判断条件，若最近使用过就将其更改为 0，并继续查看下一个 cache 是否最近使用过。 nowpointer = (nowpointer + 1) % MAX_CACHE;通过对 cache 最大数量取余，实现了近似对缓存的循环查找。当最终找到空闲 cache 时，index=nowpointer，将现在这个空闲块的 used 位标记为 1（表示我们现在已经用过了/正在用）。

Cache 结构体的 key 用于储存 HTTP 请求的 URL，用作缓存的查找键。key 和 value 是一个键值对，key 存储 URL 索引，value 存储 buf 中的内容（这个 buf 不是单行 buf，而是 doit 传入的参数 cache。这个 cache 不是结构体，而是 doit 中的局部变量字符串，储存了一次请求的所有单行 buf）。函数最后使用 sem_post 释放写锁。

```
int readcache(int fd, char* url){
    sem_wait(&rw->lock);            //读者等待并获取锁(因为要修改全局变量，可能是线程不安全的，所以要锁)
    if (rw->readers == 0)           //如果没有读者的话，说明可能有写者在写，必须等待并获取写者锁
        sem_wait(&rw->writelock);   //读者再读时，不允许有写着
    rw->readers++;
    sem_post(&rw->lock);            //全局变量修改完毕，接下来不会进入临界区，释放锁给更多读者使用
    int i, flag = 0;

    /*依次遍历找到缓存，成功则设置返回值为 1*/
    for (i = 0; i < MAX_CACHE; i++){
```

```
        //printf ("Yes! %d\n",cache[i].usecnt);
        if (strcmp(url, cache[i].key) == 0){
            Rio_writen(fd, cache[i].value, strlen(cache[i].value));
            printf("proxy send %d bytes to client\n",
strlen(cache[i].value));
            cache[i].used = 1;
            flag = 1;
            break;
        }
    }

    sem_wait(&rw->lock);    /*进入临界区，等待并获得锁*/
    rw->readers--;
    if (rw->readers == 0)   /*如果没有读者了，释放写者锁*/
        sem_post(&rw->writelock);
    sem_post(&rw->lock);    /*释放锁*/
    return flag;
}
```

在 doit 函数中，readcache 的功能是读取之前在 writecache 中可能写入过的相同内容。doit 函数中存在这样一个判断行

if (readcache(fd, urltemp) != 0)　　//如果读者读取缓存成功的话，直接返回

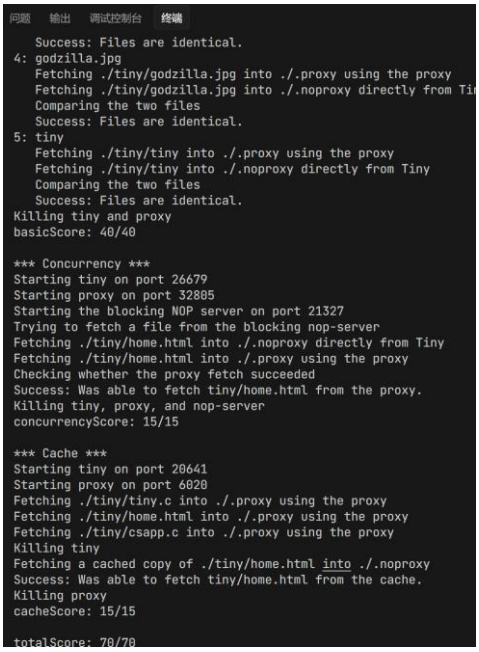
return;

在收到请求时，如果能在缓存中找到已经弄好的请求，会节约很多资源。

rw->lock 是基本锁，主要用于保护对 readers 计数器的并发访问。在多线程并发的情况下，多个读者可以同时访问/读取同一个缓存，但是当正在读取缓存的时候，其他线程是不可以写入缓存的，所以使用 rw->lock 进行一个基本锁。使用 sem_wait(&rw->lock)等待可读。若此时 rw->readers==0(无读者)则 sem_wait(&rw->writelock)，这样做是给写操作上锁，防止其他线程在读操作时进行写。当有读者的时候不用操作，因为最初的读者已经给写操作上锁了，防止冲突。然后给 readers++,表示自己正在读。sem_post(&rw->lock)释放基本锁。基本锁只保证在读者计数时，不会被写操作打断，所以使用的方式不是在函数的开头和结尾分别进行一次等待和释放，而是“等待→读者数+1→释放→读取 cache→等待→读者数-1→释放→函数结束”。读取操作通过遍历 cache，使用 strcmp 比较传入的 url 参数和缓存中 cache[i]的 key，若找到了目标的 cache[i]，就使用 Rio_writen 将 cache[i].value 发送给服务器，同时打印输出字节数量，将最近使用标记位标记为 1(cache[i].used=1),flag=1.

flag 是 readcache 函数的返回值，用于表示是否找到了目标 cache。

至此 proxy lab 的编程就基本结束了。编译运行，使用 driver.sh 进行案例测试，得分70 分满分。



实验感悟：

实验是我在暑假做完的，由于网络这一章是课本的最后一章，所以很少有自媒体有这种毅力能把课讲到最后一章，网上教学资源也挺少，照着卡耐基梅隆大学的课看也看的似懂非懂。最终还是靠着 ai 解析一点一点硬把实验啃下来了。不得不吐槽一点，linux 的网络编程实在是繁琐，但是非常符合教科书的体系，适合学生深入了解不同设备建立起链接的方式。确实，我在搞完这个实验之后彻底搞明白了什么叫套接字，什么叫 buf，什么叫文件描述符。本来天书一样的这个实验，在写完实验报告之后也是在脑中构建起一个完整的图谱来了。受益匪浅!