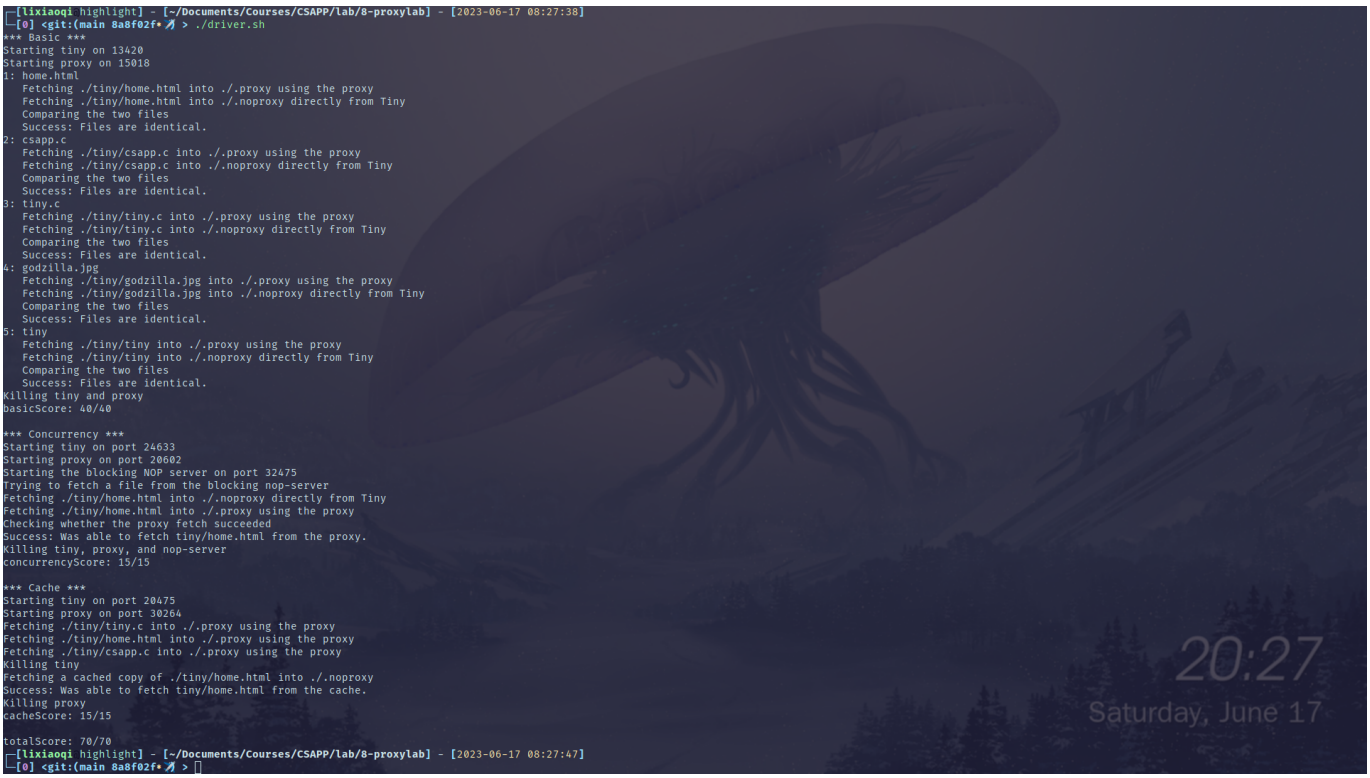
proxylab报告

快览



详解

1、part1

第一部分，需要利用socket编程，实现客户端和服务端之间的代理，在教材分别有关于客户端与服务端的实现，代理端其实就是作为客户端的服务器、服务端的客户，通过转发数据，连接起两方的通信即可，除基本的转发之外还需要填写Host、User-Agent等字段。

这里给出为每个连接服务的 `do_it` 函数的代码：

```
void doit(int client_fd)
{
    char buf[MAXLINE], method[MAXLINE], uri[MAXLINE], version[MAXLINE];
    rio_t client_rio, server_rio;

    /* Read request line and headers */
    Rio_readinitb(&client_rio, client_fd);
    if (!Rio_readlineb(&client_rio, buf, MAXLINE))
        return;
    sscanf(buf, "%s %s %s", method, uri, version);
    printf("%s %s %s\n", method, uri, version);
    if (strcasecmp(method, "GET")) {
        clienterror(client_fd, method, "501", "Not Implemented",
                    "proxy does not implement this method");
        return;
    }
    /* Open Server Connection */
    struct URI_INFO uri_info;
    int server_fd = connnect_server(uri, &uri_info);
    if (server_fd < 0)
        return;
    Rio_readinitb(&server_rio, server_fd);

    /* requst line */
    sprintf(buf, "GET /%s HT\rTP/1.0\r\n", uri_info.filename);
    Rio_writen(server_fd, buf, strlen(buf));
    /* Host */
    sprintf(buf, "Host: %s\r\n", uri_info.hostname);
    Rio_writen(server_fd, buf, strlen(buf));
    /* request header */
    forward_requesthdrs(&client_rio, &server_rio);
    puts("");
    /* response */
    forward_response(&client_rio, &server_rio);

    Close(server_fd);
    Close(client_fd);
}
```

2、part2

在part1中，代理是串行的，一个请求必须等待上一个连接被释放，才会被服务。在part2的测评脚本中，代理服务器将先被指定连接到一个不会发出回应的 `nop_server` ，然后测试向 `tiny_server` 的代理转发。在串行实现中，代理会被阻塞在 `nop_server` 处，part2要求实现代理服务器的线程并行。

这里采用最简单的形式：每一个新连接都产生一个子线程。

主线程循环：

```
while (1) {
    clientlen       = sizeof(clientaddr);
    int *client_fd_ptr = Malloc(sizeof(int));
    *client_fd_ptr    = 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, doit_thread, client_fd_ptr);
}
```

线程函数：

```
void *doit_thread(void *client_fd_ptr)
{
    int client_fd = *(int *)client_fd_ptr;
    Pthread_detach(pthread_self());
    Free(client_fd_ptr);
    doit(client_fd);
    return NULL;
}
```

唯一存在的竞争状态就是`client_fd`的写入，如果主线程使用相同的内存做fd分配，那么就存在可能：上一个子线程还未读取完毕，主线程就接受了一个新的连接并写入了`client_fd`。所以这里为每一个线程单独开辟一块内存，保存对应的fd参数。

> 这里的实现总归是toy example，对于malloc失败、线程创建失败等情况，标准的服务器应该至少满足：
>
> - 避免服务器崩溃
> - 总结并反馈错误信息
> - 释放已分配资源，避免内存泄漏
>
> 由于评测中不对这部分做考察，我就简单地调用了`csapp.h`中的封装函数，出错会直接导致服务器退出。

3、part3

这部分要求实现简单的LRU缓存，测试脚本将首先开启tiny服务器，通过代理读取3个文件，然后关闭tiny服务器，再次读取，只有实现了缓存才能成功发送。

为了最大化利用缓存空间，应该使用链表存储缓存空间的指针，实现变长的块缓存，但这也会带来内存空间不连续的情况，需要结合循环内存地址甚至地址映射，太过复杂。所以这里采用了简单的固定块大小实现：

```
struct block
{
    char uri[MAXLINE];
    char object[MAX_OBJECT_SIZE];
    size_t size;

    int LRU;
    int is_used;

    int reader_cnt;
    sem_t mutex, w;
};
struct
{
    struct block blocks[MAX_OBJECT];
} cache;
```

在每次收到GET请求时，首先判断申请的URI是否已缓存，如果是，读取完剩余请求头后直接返回缓存内容，否则正常发出请求，并在大小不超限的情况下进行新缓存。

这部分需要小心实现的点在于共享变量读写的互斥。根据文档要求，我们使用读者优先模型：

```
int read_cnt;
sem_t mutex, w;

void reader(void)
{
   while(1){
      P(&mutex);
      readcnt++;
      if(readcnt==1)
         P(&w);
      V(&mutex);

      P(&mutex);
      readcnt--;
      if(readcnt==0)
         V(&w);
      V(&mutex);
   }
}

void writer(void)
{
   while(1){
      P(&w);

      ...

      V(&w)
   }
}
```

在 `struct block` 中，定义的两个锁 `mutex` 和 `w` 就是与上述相同的语义：

- 锁w用来区分读者、写者，读者群组在读取时、写者在写时，都要持有锁w
- 锁mutex用来保证锁w的互斥，以及block的状态、内容具有互斥的写操作

至于LRU的部分，由于我们希望细粒度的并行，writer检查时不能阻塞掉全部block，所以不能精准实现LRU。在准备缓存新数据时，寻找block的代码如下：

```
struct block *write_ptr;
for (int i = 0; i < MAX_OBJECT; i++) {
    struct block *p = &cache.blocks[i];
    reader_P(p);
    if (p->is_used == 0) {
        write_ptr = p;
        reader_V(p);
        break;
    }
    if (p->LRU < minLRU) {
        minLRU    = p->LRU;
        write_ptr = p;
    }
    reader_V(p);
}
```

我们无法保证最后写入的`write_ptr`一定是最低频率访问的block、或者在执行V操作后不会出现其他writer抢先执行了P，所以这个LRU是报告中的`something reasonably close`。