操作系统实习大作业结题报告

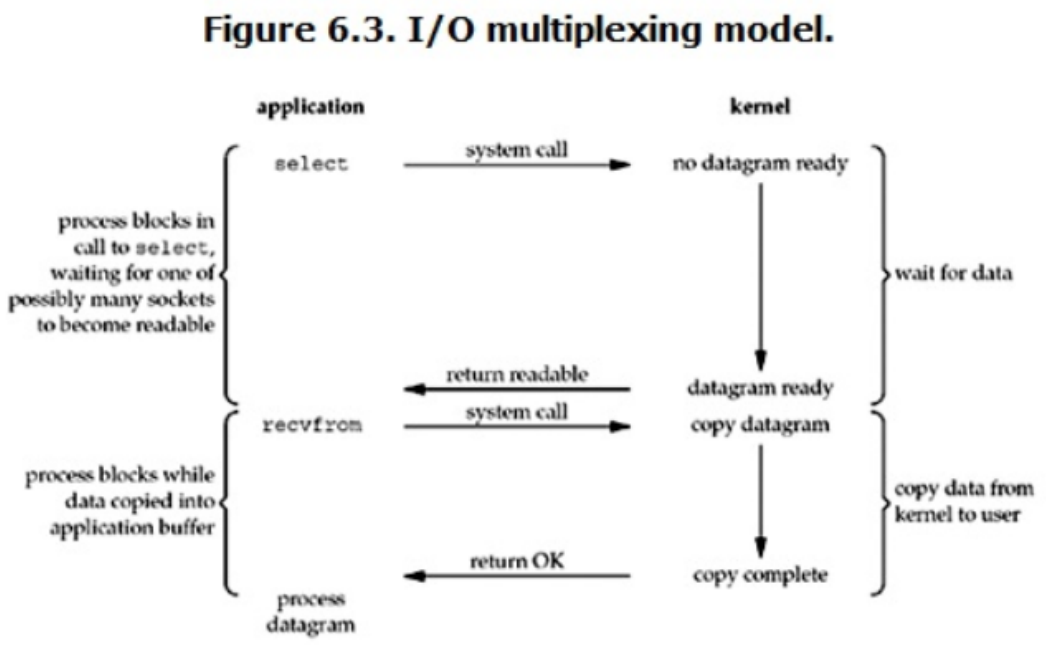
1. 前言

本次操作系统实习大作业，我选择的主题是proxylab，主要进行了两方面工作：其一，研究了用IO多路复用方法实现并发代理服务器的方法，在成功实现的基础上，对多路复用方法所需要的操作进行了必要的封装，并且编写了一些测试工具；其二，对proxylab课程测试本身的检测流程进行了一些改良，添加了用真实网页自动化对提交的proxy进行鲁棒性和速度测试的过程。

2. IO多路复用

多线程是使得代理服务器支持并发的标准方法，但并不是唯一方法。IO多路复用使得我们可以用一个进程处理多个客户端请求。

阻塞的IO是导致单一进程不能处理并发请求的根本原因，但如果只是简单地改IO非阻塞，我们又必须通过轮询的手段检测数据是否已经准备好，从而浪费大量的CPU时间。IO多路复用的基本思路是在保持单个IO接口依然阻塞的条件下，通过Select函数让内核检测一组IO接口的可读写状态，并且在任一个接口可读或者可写时返回。注意Select本身还是阻塞的，因此我们使用的多路复用是一个**阻塞异步IO**模型。下面以Select为例，介绍该方法的模型：



I. 准备好想要监视的IO端口集合S
II. 执行Select系统调用，进程阻塞，让内核代替进程自己监视整个IO集合读写状态
III. 某些IO端口可读写，Select返回
IV. 处理对应端口的工作，此时一定不会被阻塞
V. 执行 I

以上循环中，我们实际上相当于阻塞在了多个IO上，注意Select本身是不能明确区分哪些端口处于可读写状态的，因此Select返回后，我们需要对端口集合进行手动检查。

相比于多线程，IO多路复用机制由于不再需要上下文切换以及维护多线程的工作信息，节约了系统开销。特别是在处理端口数量巨大，无法承担创建线程的开销时，IO多路复用模型是一个很好的选择。

3. 用IO多路复用实现proxy

根据教材提示，我们选择使用Select函数来实现多路复用。Select的具体用法在CSAPP教材上讲的比较清楚，在此不再赘述，但是要满足我们的需要，我们必须能检测可读和可写状态两种状态，原因是为了防止进程在连接到目标服务器时被阻塞（后面详细叙述）。

简易web proxy的主要工作就是接受客户端发来的请求连接，解析client的http请求，再向真正的目标服务器发起连接，取得内容，然后写回给客户端即可。基于多线程的proxy主要是使用主线程接受客户端的请求，然后为一份客户端的工作创建一个子线程，由子线程去完成具体工作。现在我们不再使用多线程，因此既需要能够通过Select同时监测多个连接的可读写情况，还必须在返回后能够有序地处理不同的工作。因此，我们选择以**一个客户端请求所蕴含的工作**为基本元素，构建为Select服务、完成具体任务的工作池pool：

```
/* Multi-I/O work pool */
typedef struct {
    int maxfd;
    int nready;
    fd_set read_set;
    fd_set write_set;
    fd_set ready_read_set;
    fd_set ready_write_set;
    int maxi;
    int socket_oldoption[MAX_WORKBUF_SIZE];
    int fail_time[MAX_WORKBUF_SIZE];
    int state[MAX_WORKBUF_SIZE];
    int client_fd[MAX_WORKBUF_SIZE];
    int server_fd[MAX_WORKBUF_SIZE];
    struct addrinfo *listp[MAX_WORKBUF_SIZE];
    struct addrinfo *p[MAX_WORKBUF_SIZE];
    char *end_http_header_p[MAX_WORKBUF_SIZE];
    rio_t clientrio[MAX_WORKBUF_SIZE];
    rio_t serverrio[MAX_WORKBUF_SIZE];

} pool;
```

我们在池中维护Select自己所需要的最大文件描述符、读写监测集合，并且将具体任务组织为工作槽，每个槽中元素都拥有自己完成工作所**必须**的数据结构。由于连接到目标服务器可能不会一次成功，proxy必须有能力搁置结果不确定的当前工作，转而去服务其他client，并适时回来继续工作，因此对每个元素，保存工作过程中的一些数据都是必要的。

下面介绍程序的主要逻辑：

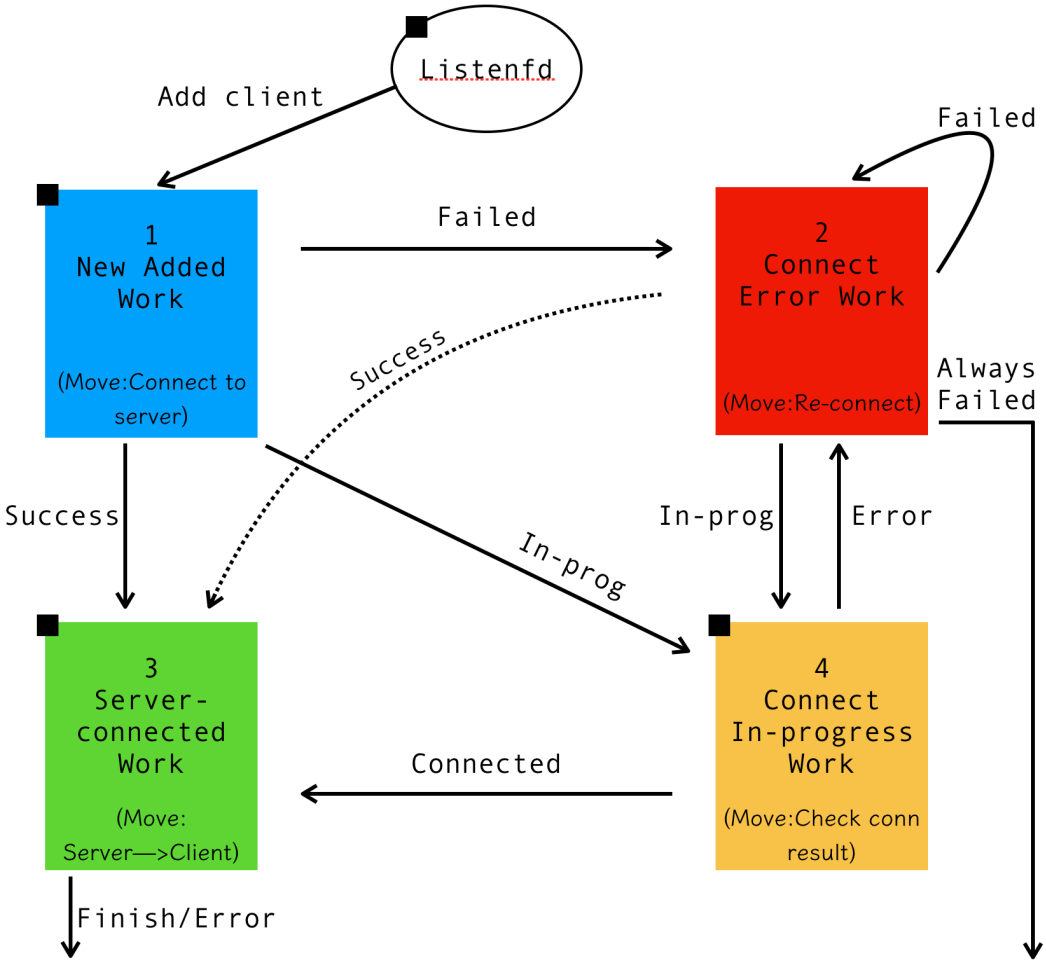
I. 将监听描述符加入Select监测集合（监听描述符应当被一直监测）
II. 更新监测集合
III. 对给定好的监测集合，执行Select，挂起当前进程
IV. 从Select返回
V. 若有，把监听描述符的连接加入工作槽
VI. 遍历工作槽，处理具体工作，相应地增删改监测集合元素
VII. 执行 II

从上述逻辑可以得知：工作槽中的所有元素都来自于监听描述符的添加；完成工作后的工作槽元素应该被清除；工作槽中的元素在某一时间总是处于某个工作阶段之中；遍历工作槽的过程应当能够推进每个元素在工作流程中前进。针对于proxy的具体任务，我们为每一个元素设置了工作状态量，并将其构建为简单的**有穷自动机。**这样，处理每个元素就可以依照自动机的逻辑进行。在这里，我们为每个工作元素设置了4个状态：

STATE 1： 新加入
STATE 2： 连接失败
STATE 3： 已经连接到目标服务器
STATE 4： 正在连接/连接结果未知

非常值得一提的是，状态4的设置对于基于Select的proxy是非常必要的。注意，我们的proxy现在只拥有一个进程，默认连接到目标服务器的connect操作是默认阻塞的，而TCP三路握手时间最多可以长达75s。如此长的阻塞时间是不可接受的。因此我们必须使用非阻塞的连接方式，在调用connect之后立刻返回。此时的连接有可能立刻成功/失败，但更多情况下，状态为**正在连接（EINPROGRESS）**。一个正在连接的socket端口是不可写的。我们也用Select来对其进行监测。只有当连接有了确切的结果时，才去处理这个工作元素。由此，我们实现了连接某一服务器与处理其他工作的**异步**。

下面是4状态的有穷自动机，处理每种状态要进行不同的Move，并且根据Move的执行结果，决定此工作元素变为哪一种状态。带黑色方块的状态表示只有当Select检测到此元素处在可读/可写集合中时，才进行处理，否则直接跳过。注意处于状态2的元素没有这一要求，这意味着我们每次检查工作槽时都会将连接错误的元素进行重连。正如pool的定义，我们会记录一个元素的连接失败次数，当其超过一定阈值之后，此工作元素将被从槽中清除，以防止其长期占用资源。



对Select监听描述符集合的处理非常重要，并且与状态的处理关系密切。简单来说，状态1时，被监听的描述符是其实连接client的socket，其可读意味着client已经把http请求写到端口上，我们可以去读；状态3时，监听的是连接server的socket，其可读意味着server已经把client要的内容写到端口，我们应该读取然后写到client的端口上。特别地，前两个是在监听是否可读，而状态4则监测尝试连接server的socket是否可写。但是，一旦可写，也**不能**确定连接成功，还需进一步检查套接字的error option来区分连接错误和连接已建立。

另外遇到的一个非常有意思的问题，是针对状态3中server传来的数据，我们应该如何写回client。最直接的想法就是不断读取socket上的内容随即写回client，直到读完所有。这种做法一般不会有严重问题，但是实质上**不利于**proxy支持并发。其一，如果某些不正常甚至恶意的client通过proxy去访问非常大的资源，那么代理服务器将在处理此工作元素上花费大量的时间，这段时间内别的client得不到服务，也不能响应新的连接。其二，proxy对待不同的client应该秉持**公平**原则，不应当有client占用过多的时间。目前我个人的处理方法是，对监测到可读的状态3的元素，每次处理只从server读取一行并写入到client。直到读完了server的全部内容，才把其描述符从监测集合中移除。这样可以保证，我们每次检查工作槽时，都会处理尚未读完的状态为3的工作元素。

多线程的实现中，每个子线程都可以读取server的内容直到结束，这是因为不同线程之间的并发，是由**操作系统**进行**调度**的结果；由于现在只有一个进程，公平对待不同任务就成为实现proxy需要考虑的问题。上面所说的处理方法，相当于按照**轮转法**进行调度。

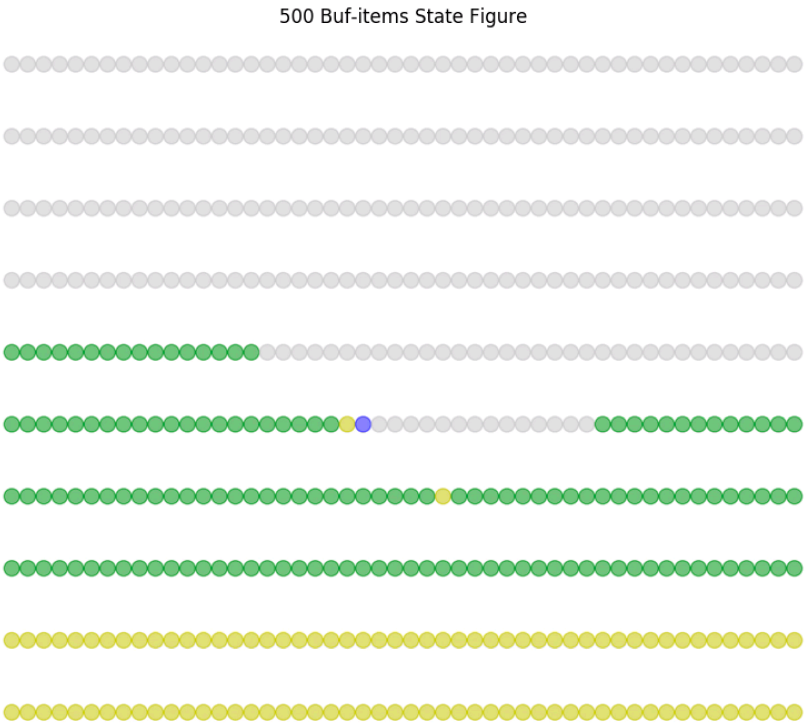
4. 其他事项以及整合到lab

不同于多线程，在使用多路复用实现proxy时，必须妥善处理各种异常情况，不能武断地直接退出程序。CSAPP包装的Rio系列函数在遇到IO错误时会直接导致进程退出，因此应当全部使用未包装的rio系列函数，并且谨慎处理错误返回值。

针对于调试工作，我们不再遇到多线程时的恼人的并发错误。由于程序的控制流在自动机中转换，每个工作元素的状态含有关键信息。因此，我实现了一个基于python可视化的debug工具vis_buf.py：在proxy中每次检查工作槽后，将前500个元素的状态值写入到文件中；外部使用python脚本循环刷新，读取文件并将状态可视化为散点图。通过动态可视化，我们可以清晰地看到工作槽中的元素是如何发生状态转换，直至最后被清除的。

此外，还实现了一个实用的调试工具test_client，其可以取得给定网址的http内容；还可以在指定并发数量后，使用多线程向指定端口建立连接然后发送请求，可以用来测试proxy的鲁棒性和性能。具体的工作方式由参数确定。

下图是test_client和vis_buf.py搭配使用的效果图：



这部分针对lab本身的工作是对操作Select工作池状态、处理池中工作元素、处理Select监测集合等完成proxy所必需的过程以及调试函数进行了封装，封装所在C文件与头文件一

并打包分发给学生，最终链接到proxy之中（封装的细节在后面叙述）。我们希望学生可以深入思考IO多路复用的实现方法，但是并不强制。上面提到的两个辅助工具也会随附，以供方便。事实上，在探究实现的过程中，对proxy最主要的检验方法就是访问现有网络的http资源。因此，真实页面仍然是最推荐的测试手段。

此外，为了降低学生理解的困难，分发给学生的proxy.c中包含好了程序运行的主要逻辑，学生只需要完成检查工作槽时，针对不同状态的工作元素的处理函数即可。我们按照以上思路实现的多路复用proxy源代码及可执行文件，则被添加到了lab的src文件夹中，以供用于对学生提交的测试工作。

5. 对lab测试的改进

另外一部分工作是改进对学生提交的测试。原有proxylab的测试方法是在本地运行tiny服务器，透过proxy去访问资源，以此来检测正确性和功能实现情况。对于真实页面，则需要人工筛查。借助前面实现的测试工具test_client以及实现的多路复用proxy，我在脚本中添加了三个访问真实网页的测试，并且调整了分数的分配。

Real Basic测试提交proxy访问真实页面的基本功能。通过test_client向proxy和标准proxy发送请求，将两边的返回内容写入文件然后比对，一致则得分。

```
*** Real Basic ***
Starting proxy on port 7400
Starting std_proxy on port 27673
Fetching pkunews.pku.edu.cn using the proxy
Fetching pkunews.pku.edu.cn using the std_proxy
   Comparing the two files
   Real Basic Success: Files are identical.
Killing proxy and stdproxy, clear tmp files
Real Basic: 15 / 15
*** Real Basic End ***
```

Robust测试提交proxy对异常IO情况的鲁棒性。首先执行多线程的test_client向proxy发送请求，随后立即kill该进程，以制造IO异常。在此之后重复Real Basic的测试并计分。

```
*** Robust ***
Starting proxy on port 1688
Starting std_proxy on port 4004
Fetching pkunews.pku.edu.cn using the proxy

Making bad client.
Bad client finished.
Fetching pkunews.pku.edu.cn using the std_proxy
   Comparing the two files
   Robust_Success: Files are identical.
Killing proxy and stdproxy, clear tmp files
Robust: 10 / 10
*** Robust End ***
```

Speed粗略测试提交proxy处理较多任务的性能。通过多线程test_client，对同一资源，向proxy和标准proxy发送相同并发数量的请求，并分别计时，然后比较。由于真实的网络环境可能比较复杂，测试的结果并不稳定，比较粗略。此外，该测试里标准proxy的实现方法并没有追求性能上的最优，理论上并不会使得学生难以通过测试。

```
*** Speed ***
Starting proxy on port 7036
Performance testing pkunews.pku.edu.cn using the proxy

Starting std_proxy on port 10027
Performance testing pkunews.pku.edu.cn using the stdproxy

Proxy time: 4.17
Std_proxy time: 3.51
Speed: 0 / 5
*** Speed End***
```

以上的工作通过修改driver.sh进行，为了保证测试效果，后三种测试对学生并不可见。对proxy的真实测试操作都使用了超时机制，以防止脚本运行时间过长。

下面是在Autolab上的部分测试结果：

You have 31 submissions left.

VER	FILE	SUBMISSION DATE	BASIC (40.0)	CONCURRENCY (15.0)	CACHING (15.0)	REAL PAGES (20.0)	STYLE (10.0)	CORRECTNESS DEDUCTIONS (0.0)	LATE PENALTY (PENALTY LATE DAYS)	TOTAL SCORE
1	1600012757@pku.edu.cn_1_proxylab-handin.tar	2018-12-26 00:55:16 -0500	40.0	15.0	0.0	--	--	--		--

Page loaded in 0.014987166 seconds

Autolab Project · Contact · GitHub · Facebook · Logout

6. 感想与收获

本质上说，探究多路复用的proxy，本质上是在研究**IO模型**和**并发问题**。必须承认，此模型虽然没有多线程的问题，但是程序的复杂性比较高，构建思路和调试工作并不是很容易。Select是最经典的IO多路复用的实现函数，也是CSAPP教材提及的内容，使用它可以让我们以最朴素、最直接的方式去理解IO多路复用的思想。封装的工作过程涉及到为别人构造接口，必然要涉及到API的规范问题，以及符合既有编程惯例的要求。如何编写才能使得可靠性和复用性高，也是必须要考虑的一个问题。

为了实现对测试的改进，我学习了shell语言并且认知到其强大。前期调试脚本有些困难，随后越来越熟练。自我构思测试方法的过程还是非常有趣的。还有一点重要的收获就是使用git进行版本控制的一些经验。

研究IO模型的长远意义在于构建高性能服务器。Select本身的能力其实相当有限，性能上也不够优秀，但是其可移植性最好。Linux上已有诸如epoll等更高效的实现方法，事件驱动模型的框架和库也相对比较丰富，进一步调研、学习这些方法和库，是我将来非常想要进行的工作。但是无论如何变化发展，这些模型的基本思想总是一致的。

7. 附：封装工作的细节

前面已经介绍了pool工作池的实现，下面是一些相关封装的简单介绍：

- 对Select监测集合操作的封装，会更新池中关于监测的文件描述符的记录值

```
void AddtoSet(pool *p,int fd,fd_set *set);
void RemovefromSet(pool *p,int fd,fd_set *set);
int IsinSet(int fd, fd_set *set);
```

- 对工作池及其元素操作的封装，包含初始化池、增删改元素，都会更新相关记录值

```
void Clear_pool_item(pool *po, int i);
void Init_pool(int listenfd, pool *po);
void Add_client(int connfd, pool *p);
void Place_endserver(int connfd,pool *p,int i);
void Stop_work(pool *p, int i);
void Terminate_work(pool *p, int i);
```

- 实现非阻塞connect功能的封装，屏蔽底层细节

```
int Check_nb_soc_state(int fd);
void Set_serfd_to_block(pool *po, int i);
void Close_nb_serverfd(pool *po, int i);
int Get_socket_list(char *hostname,char *port,pool *po, int i);
int Nonblock_try_connect(int i, pool *po, int *waiting_fd);
int New_try_open_serverfd(int i, pool *po, int *endserverfd);
```

- 完成工作的辅助封装，考虑任务流程而设计，可能不必需

```
void Save_http_header(pool *po, int i, char *http_hd);
void Free_http_header(pool *po,int i);
```

- 搭配vis_buf.py可视化脚本使用，调试函数

```
int Make_file_record(pool *p);
```

注意，以上所列封装都是为了方便使用IO多路复用的模型实现proxy而设计的，多数需要有Select工作池支持，而且全部没有考虑线程安全问题。