用fork()模拟多个客户端同时访问url，测试网站在压力下工作的性能。

主函数进行必要的准备工作，进入bench开始压测

bench函数使用fork模拟出多个客户端，调用socket并发请求，每个子进程记录自己的访问数据，并切入管道

父进程从管道读取子进程的输出信息。

使用alarm函数进行时间控制，到时间后会产生SIGALRM信号，调用信号处理函数使子进程停止

最后只留下父进程将所有子进程的输出数据汇总计算，输出到屏幕上。

工作流程图：

Webbench主要两个源文件socket.c和webbench.c

socket.c主要是封装的一个socket模块，

webbench.c是主要文件，完成网站测压的整个过程。

## socket.c

// socket描述符，主要以host和clientPort构成一对TCP的套接字（host支持域名），创建失败返回-1，成功返回一个

int Socket(const char \*host, int clientPort)

{

int sock;

unsigned long inaddr;

struct sockaddr\_in ad;

struct hostent \*hp;

memset(&ad, 0, sizeof(ad));

ad.sin\_family = AF\_INET;

//若字符串有效，则将字符串转换为32位二进制。

//网络字节序的IPV4地址，否则为INADDR\_NONe

inaddr = inet\_addr(host);

if (inaddr != INADDR\_NONE)

memcpy(&ad.sin\_addr, &inaddr, sizeof(inaddr));

else

{

// 返回对应于给定主机名的包含主机名字和地址信息的hostent结构指针

hp = gethostbyname(host);

if (hp == NULL)

return -1;

memcpy(&ad.sin\_addr, hp->h\_addr, hp->h\_length);

}

// 将一个无符号短整型的主机数值转换为网络字节顺序

ad.sin\_port = htons(clientPort);

// 创建socket套接字

sock = socket(AF\_INET, SOCK\_STREAM, 0);

if (sock < 0)

return sock;

// 连接到相应的主机

if (connect(sock, (struct sockaddr \*)&ad, sizeof(ad)) < 0)

return -1;

return sock;

}

## Webbench.c

从mian，其整体流程为：

main——>对命令行进行参数解析——>调用build\_request函数构建HTTP的“Get”请求头——>调用bench测试函数（其中子进程调用benchcore函数进行压力测试），之后主进程从管道读取消息，并输出到标准输出上即可。

首先是main函数，mian函数首先进行参数解析，

然后执行build\_request函数，构建HTTP请求头，

最后调用核心函数bench()执行网站测压测试。

// 主函数

int main(int argc, char \*argv[])

{

// getopt\_long的返回字符

int opt=0;

// getopt\_long的第五个参数，一般为0

int options\_index=0;

char \*tmp=NULL;

if(argc==1)

{

usage();

return 2;

}

// 用getopt\_long函数读取命令行参数，来设置所涉及到的全局变量的值。

// getopt\_long()支持长选项的命令行解析，其声明如下：

// int getopt\_long(int argc, char \*const argv[], const char \*optstring, const struct option \*long\_options, int \*longindex)

while((opt=getopt\_long(argc,argv,"912Vfrt:p:c:?h",long\_options,&options\_index))!=EOF )

{

switch(opt)

{

case 0 : break;

case 'f': force=1;break;

case 'r': force\_reload=1;break;

case '9': http10=0;break;

case '1': http10=1;break;

case '2': http10=2;break;

case 'V': printf(PROGRAM\_VERSION"\n");exit(0);

case 't': benchtime=atoi(optarg);break; // -t 后跟压力测试时间，optarg返回，使用atoi转换成整数

case 'p':

/\* proxy server parsing server:port \*/

tmp=strrchr(optarg,':');

proxyhost=optarg;

if(tmp==NULL)

{

break;

}

if(tmp==optarg)

{

fprintf(stderr,"Error in option --proxy %s: Missing hostname.\n",optarg);

return 2;

}

if(tmp==optarg+strlen(optarg)-1)

{

fprintf(stderr,"Error in option --proxy %s Port number is missing.\n",optarg);

return 2;

}

// 获取代理地址

\*tmp='\0';

proxyport=atoi(tmp+1);break; // 获取代理端口

case ':':

case 'h':

case '?': usage();return 2;break;

case 'c': clients=atoi(optarg);break; // 并发数目 -c N

}

}

//扫描参数选项时，optind标识下一个选项的索引；扫描结束后，标识第一个非选项参数索引；

//如果optind=argc，说明非选项参数即服务器URL缺失。此变量是系统定义的。

// optind返回第一个不包含选项的命令名参数，此处为URL值

if(optind==argc)

{

fprintf(stderr,"webbench: Missing URL!\n");

usage();

return 2;

}

// 此处多做一次判断，可预防BUG，因为上文并发数目用户可能写0

if(clients==0) clients=1;

// 压力测试时间默认为30s，如果用户写成0，则默认为60s

if(benchtime==0) benchtime=60;

/\* Copyright \*/

fprintf(stderr,"Webbench - Simple Web Benchmark "PROGRAM\_VERSION"\n"

"Copyright (c) Radim Kolar 1997-2004, GPL Open Source Software.\n"

);

// 调用build\_request函数构建完整的HTTP请求头，HTTP request存储在全局变量char request[REQUEST\_SIZE]

build\_request(argv[optind]); // 参数为URL值

/\* print bench info \*/

// 在屏幕上打印测试的信息，如HTTP协议，请求方式，并发个数，请求时间等

printf("\nBenchmarking: ");

switch(method)

{

case METHOD\_GET:

default:

printf("GET");break;

case METHOD\_OPTIONS:

printf("OPTIONS");break;

case METHOD\_HEAD:

printf("HEAD");break;

case METHOD\_TRACE:

printf("TRACE");break;

}

printf(" %s",argv[optind]);

switch(http10)

{

case 0: printf(" (using HTTP/0.9)");break;

case 2: printf(" (using HTTP/1.1)");break;

}

printf("\n");

if(clients==1) printf("1 client");

else

printf("%d clients",clients);

printf(", running %d sec", benchtime);

if(force) printf(", early socket close");

if(proxyhost!=NULL) printf(", via proxy server %s:%d",proxyhost,proxyport);

if(force\_reload) printf(", forcing reload");

printf(".\n");

// 调用bench函数，开始压力测试，bench()为压力测试核心代码

return bench();

}

下面是build\_request函数分析

//此函数主要目的是要把类似于http GET请求的信息全部存储到全局变量request[REQUEST\_SIZE]中，其中换行操作使用"\r\n"。

//其中应用了大量的字符串操作函数。

//创建url请求连接，HTTP头，创建好的请求放在全局变量request中

void build\_request(const char \*url)

{

char tmp[10];

int i;

bzero(host,MAXHOSTNAMELEN);

bzero(request,REQUEST\_SIZE);

// 协议适配

if(force\_reload && proxyhost!=NULL && http10<1) http10=1;

if(method==METHOD\_HEAD && http10<1) http10=1;

if(method==METHOD\_OPTIONS && http10<2) http10=2;

if(method==METHOD\_TRACE && http10<2) http10=2;

switch(method)

{

default:

case METHOD\_GET: strcpy(request,"GET");break;

case METHOD\_HEAD: strcpy(request,"HEAD");break;

// 请求方法相应的不能缓存

case METHOD\_OPTIONS: strcpy(request,"OPTIONS");break;

case METHOD\_TRACE: strcpy(request,"TRACE");break;

}

// 追加空格

strcat(request," ");

if(NULL==strstr(url,"://")) // strstr(str1, str2)用于判断str2是否是str1的子串

{

fprintf(stderr, "\n%s: is not a valid URL.\n",url);

exit(2);

}

if(strlen(url)>1500)

{

fprintf(stderr,"URL is too long.\n");

exit(2);

}

if(proxyhost==NULL)

// 未使用代理服务器的情况下，只允许HTTP协议

if (0!=strncasecmp("http://",url,7)) // 比较前7个字符串

{ fprintf(stderr,"\nOnly HTTP protocol is directly supported, set --proxy for others.\n");

exit(2);

}

/\* protocol/host delimiter \*/

// 指向"://"后的第一个字母

i=strstr(url,"://")-url+3;

/\* printf("%d\n",i); \*/

// URL后必须的'/'

if(strchr(url+i,'/')==NULL) //url + i 指向http://后第一个位置

{

fprintf(stderr,"\nInvalid URL syntax - hostname don't ends with '/'.\n");

exit(2);

}

// 如果未使用代理服务器，就表示肯定是HTTP协议

if(proxyhost==NULL)

{

/\* get port from hostname \*/

// 如果是server:port形式，解析主机和端口

if(index(url+i,':')!=NULL &&

index(url+i,':')<index(url+i,'/')) // 判断url中是否指定了端口号

{

strncpy(host,url+i,strchr(url+i,':')-url-i); // 取出主机地址

bzero(tmp,10);

strncpy(tmp,index(url+i,':')+1,strchr(url+i,'/')-index(url+i,':')-1);

/\* printf("tmp=%s\n",tmp); \*/

// 目标端口

proxyport=atoi(tmp); // 端口号转换为int

if(proxyport==0) proxyport=80;

} else{

strncpy(host,url+i,strcspn(url+i,"/"));

}

// printf("Host=%s\n",host);

strcat(request+strlen(request),url+i+strcspn(url+i,"/"));

} else{

// printf("ProxyHost=%s\nProxyPort=%d\n",proxyhost,proxyport);

// 如若使用代理服务器

strcat(request,url);

}

if(http10==1)

strcat(request," HTTP/1.0");

else if (http10==2)

strcat(request," HTTP/1.1");

// 完成如 GET/HTTP1.1后，添加"\r\n"

strcat(request,"\r\n");

if(http10>0)

strcat(request,"User-Agent: WebBench "PROGRAM\_VERSION"\r\n");

if(proxyhost==NULL && http10>0)

{

strcat(request,"Host: ");

strcat(request,host);

strcat(request,"\r\n");

}

// force\_reload=1和存在代理服务器，则不缓存

if(force\_reload && proxyhost!=NULL)

{

strcat(request,"Pragma: no-cache\r\n");

}

// 如果为HTTP1.1，则存在长连接，应将Connection置位close

if(http10>1)

strcat(request,"Connection: close\r\n");

/\* add empty line at end \*/

// 最后不要忘记在请求后添加“\r\n”

if(http10>0) strcat(request,"\r\n");

// printf("Req=%s\n",request);

}

下面是bench函数解析，开始先进行一次socket连接，确认能连接以后，才进行后续步骤；

调用pipe函数初始化一个管道，用于子进程想父进程汇总测试数据。

而子进程是主进程通过fork函数复制出来的；之后每隔子进程都调用benchcore函数进行测试，并将结果输出到管道，供父进程读取。

父进程负责收集所有子进程的测试数据，并进行汇总输出显示即可。

static int bench(void)

{

int i,j,k;

pid\_t pid=0;

FILE \*f;

/\* check avaibility of target server \*/

// 进行socket连接，调用了Socket.c文件中的函数，主要是为了测试远程主机是否能够连通

i=Socket(proxyhost==NULL?host:proxyhost,proxyport);

if(i<0)

{

// 错误处理

fprintf(stderr,"\nConnect to server failed. Aborting benchmark.\n");

return 1;

}

close(i);

/\* create pipe \*/

// 创建管道，管道用于子进程想父进程汇报数据

if(pipe(mypipe))

{

// 错误处理

perror("pipe failed.");

return 3;

}

/\* not needed, since we have alarm() in childrens \*/

/\* wait 4 next system clock tick \*/

/\*

cas=time(NULL);

while(time(NULL)==cas)

sched\_yield();

\*/

/\* fork childs \*/

// 根据clients大小fork出来足够的子进程进行测试

for(i=0;i<clients;i++)

{

pid=fork();

if(pid <= (pid\_t) 0) // pid=0 ->子进程.pid < 0 -> error

{

/\* child process or error\*/

// 注意这里子进程sleep(1)

sleep(1); /\* make childs faster \*/

break; // 子进程跳出循环orfork出错父进程跳出循环

}

}

if( pid< (pid\_t) 0) // fork出错

{

// 错误处理

fprintf(stderr,"problems forking worker no. %d\n",i);

perror("fork failed.");

return 3;

}

// 如果是子进程，调用benchcore进行测试

if(pid== (pid\_t) 0) // 子进程

{

/\* I am a child \*/

// 子进程执行请求，尽可能多的发送请求，直到超时返回为止

if(proxyhost==NULL)

benchcore(host,proxyport,request);

else

benchcore(proxyhost,proxyport,request);

/\* write results to pipe \*/

// 子进程将测试结果输出到管道

f=fdopen(mypipe[1],"w");

// 错误处理

if(f==NULL)

{

perror("open pipe for writing failed.");

return 3;

}

/\* fprintf(stderr,"Child - %d %d\n",speed,failed); \*/

// 子进程将speed failed bytes写进管道

fprintf(f,"%d %d %d\n",speed,failed,bytes);

fclose(f);

// 子进程完成任务，返回退出

return 0;

}

else

{

// 父进程从管道读取子进程输出，并做汇总，然后输出显示

f=fdopen(mypipe[0],"r"); // mypipe[0]与标准流相结合

// 错误处理

if(f==NULL)

{

perror("open pipe for reading failed.");

return 3;

}

// \_IONBF（无缓冲）：直接从流中读入数据或直接向流中写入数据，而没有缓冲区

setvbuf(f,NULL,\_IONBF,0); // 设置无缓冲区

// 虽然子进程不能污染父进程的这几个变量，但用前重置一下，在这里是个好习惯

speed=0;

failed=0;

bytes=0;

// 从管道读取数据，fscanf为阻塞式函数

// 从管道中读取每个子进程的任务执行请求，并计数

while(1)

{

// 通过f从管道读取数据，注意fscanf为阻塞式函数

pid=fscanf(f,"%d %d %d",&i,&j,&k);

// 错误处理

if(pid<2)

{

fprintf(stderr,"Some of our childrens died.\n");

break;

}

// 父进程利用管道负责统计子进程的三种数据和

speed+=i;

failed+=j;

bytes+=k;

/\* fprintf(stderr,"\*Knock\* %d %d read=%d\n",speed,failed,pid); \*/

// 用于记录已经读取了多少个子进程的数据，读完就退出

if(--clients==0) break;

}

fclose(f);

// 最后将结果打印到屏幕上

printf("\nSpeed=%d pages/min, %d bytes/sec.\nRequests: %d susceed, %d failed.\n",

(int)((speed+failed)/(benchtime/60.0f)),

(int)(bytes/(float)benchtime),

speed,

failed);

}

return i;

}

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

18

19

20

21

22

23

24

25

26

27

28

29

30

31

32

33

34

35

36

37

38

39

40

41

42

43

44

45

46

47

48

49

50

51

52

53

54

55

56

57

58

59

60

61

62

63

64

65

66

67

68

69

70

71

72

73

74

75

76

77

78

79

80

81

82

83

84

85

86

87

88

89

90

91

92

93

94

95

96

97

98

99

100

101

102

103

104

105

106

107

108

109

110

111

112

113

114

115

116

117

118

119

120

121

122

123

124

125

126

127

128

129

由于bench函数子进程调用了benchcore函数，而benchcore函数是测试函数，它通过使用SIGALARM信息来控制时间，alarm函数设置了多少时间之后产生SIGALRM信号，一旦产生此信息，将运行alam\_handler函数，是的timerexpired=1，这样之后可以通过判断timerexpired值来退出程序。此外，全局变量force表示是否发出请求后需要等待服务器的相应结果。

// benchcore函数是子进程进行压力测试的函数，被每个子进程调用。其函数中参数信息如下：

// host：地址

// port：端口

// req：http格式方法

void benchcore(const char \*host,const int port,const char \*req)

{

int rlen;

// 记录服务器相应请求所返回的数据

char buf[1500];

int s,i;

struct sigaction sa;

/\* setup alarm signal handler \*/

// 当程序执行到指定的秒数之后，发送SIGALRM信号，即设置alam\_handler函数为信号处理函数

sa.sa\_handler=alarm\_handler;

sa.sa\_flags=0;

// sigaction成功则返回0，失败则返回-1，超时会产生信号SIGALRM，用sa指定函数处理

if(sigaction(SIGALRM,&sa,NULL))

exit(3);

// 开始计时

alarm(benchtime);

rlen=strlen(req);

// 无限执行请求，直到收到SIGALRM信号将timerexpired设置为1时

nexttry:while(1)

{

// 一旦超时，则返回

if(timerexpired)

{

if(failed>0)

{

/\* fprintf(stderr,"Correcting failed by signal\n"); \*/

failed--;

}

return;

}

// 连接远程服务器，通过调用Socket函数建立TCP连接

s=Socket(host,port);

// 连接失败，failed数加一

if(s<0) { failed++;continue;}

// 发出请求，header大小与发送的不相等，则失败

if(rlen!=write(s,req,rlen)) {failed++;close(s);continue;}

// 针对http0.9做的特殊处理，则关闭socket的写操作，成功返回0，错误返回-1

if(http10==0)

if(shutdown(s,1)) { failed++;close(s);continue;}

// 全局变量force表示是否要等待服务器返回的数据

// 如果等待数据返回，则读取响应数据，计算传输的字节数

// 发出请求后需要等待服务器的响应结果 force=0表示等待从Server返回的数据

if(force==0)

{

/\* read all available data from socket \*/

while(1)

{

if(timerexpired) break; // timerexpired默认为0，在规定时间内读取当为1时表示定时结束

// 从socket读取返回数据

i=read(s,buf,1500);

/\* fprintf(stderr,"%d\n",i); \*/

if(i<0)

{

failed++;

close(s);

goto nexttry;

}

else

if(i==0) break;

else

bytes+=i;

}

}

// 关闭连接

if(close(s)) {failed++;continue;}

// 成功完成一次请求，并计数，继续下一次相同的请求，直到超时为止

speed++;

}

}

关于getopt getopt\_long参数处理函数

简述

在程序中一般都会用到命令行选项，我们可以使用getopt和getopt\_long函数来解析命令行参数

getopt

getopt主要用于处理短命令行选项，例如 ./test -v 中-v就是一个短命令行选项。

使用该函数需要引入头文件<unistd.h>,以下是getopt函数的定义

#include <unistd.h>

extern char \*optarg; //选项的参数指针，存放选项对应的输入参数

extern int optind; //下一次调用getopt时，从optind存储的位置处重新开始检查选项。

extern int opterr; //当opterr=0时，getopt不向stderr输出错误信息。

extern int optopt; //当命令行选项字符不包括在optstring中或者最后一个选项缺少必要的参数时，该选项存储在optopt中，getopt返回'？’

int getopt(int argc, char \* const argv[], const char \* optstring);

其中argc和argv是main函数中传递参数和内容，optstring用来指定可以处理哪些选项，

字符串optstring可以下列元素

单个字符，表示选项，没有参数，optarg=NULL.

单个字符后接一个冒号：表示该选项后必须跟一个参数。参数紧跟在选项后或者以空格隔开。该参数的指针赋给optarg。

单个字符后跟两个冒号，表示该选项后必须跟一个参数。参数必须紧跟在选项后不能以空格隔开。该参数的指针赋给optarg。（这个特性是GNU的扩张）。

下面是optstring的一个实例：

"a:bc::"

该示例表明程序可以接受三个选项：-a -b -c，其中a后面的:表示该选项后面要跟一个参数，例如./test -a text的形式，c后面的::表示该选项后面要跟一个参数且中间不准有空格，例如./test -ctext，选项后面跟的参数会被保存到optarg变量中。下面一段代码是该函数的使用实例

#include <stdio.h>

#include <unistd.h>

int main(int argc,char \* argv[])

{

int ch;

while((ch = getopt(argc, argv, "a:bc::")) != -1){

switch (ch) {

case 'a':

printf("option a: %s\n",optarg);

break;

case 'b':

printf("option b: %s\n",optarg);

break;

case 'c':

printf("option c: %s\n",optarg);

break;

case 'd':

printf("unknown option\n");

break;

default:

printf("unknown option\n");

break;

}

}

return 0;

}

getopt\_long

getopt\_long() 是同时支持长选项和短选项的 getopt() 版本。它可以根据输入的option是单横线还是双横线开头来区分是短选项还是长选项。

以下是getopt\_long的声明：

#include <getopt.h>

struct option{ //长选项表

const char \*name; //选项名，前面没有短横线，help,verbose之类

int has\_arg; //描述选项是否需要选项参数，no\_argument 0 表示选项没有参数，required\_argument 1 表示需要参数，optional\_argument 2 选项参数可选

int \*flag; //如果这个指针为NULL，那么getopt\_long()返回该结构val字段中的数值。如果该指针不为NULL，getopt\_long()会使得它所指向的变量中填入val字段中的数值，并且getopt\_long()返回0

int val;

};

// 每个长选项在长选项表中都有一个单独条目，该条目里需要填入正确的数值。数组中最后的元素的值应该全是0。数组不需要排序，getopt\_long()会进行线性搜索。

int getopt\_long(int argc, char \* const argv[], const char \*optstring, const struct option \*longopts, int \*longindex);

在webbench.c中充分体现了对getopt\_long的使用