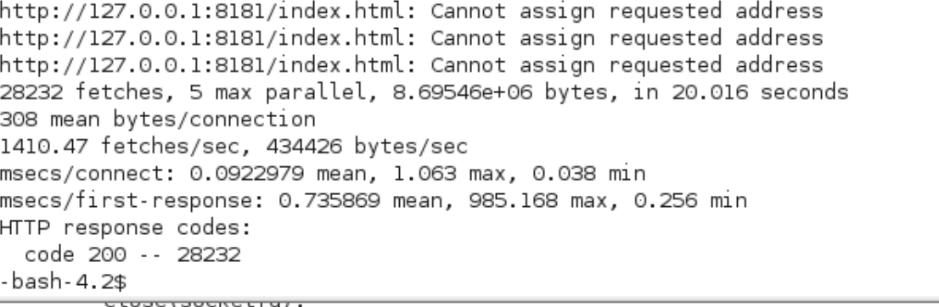
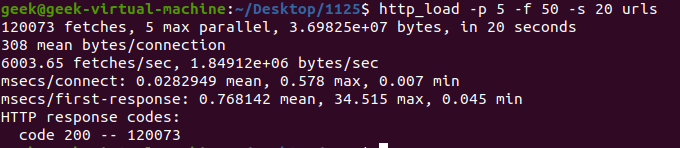
操作系统第三次课程设计（1125）

2021.12.8

2.3多线程模型

题目一：（1）将上述多线程Web 服务器与实验 2 中多进程模型的Web 服务器性能进行对比，说明它俩各具有什么优缺点。



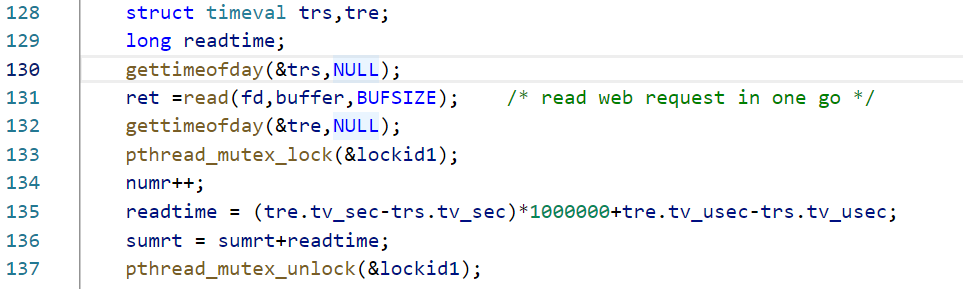
（多进程测试）

(多线程测试)

如上图所示，通过http\_load的结果，多进程在20s内模拟5个客户端发送数据量是8.69\*10^6，而多线程则是3.69\*10^7，线程大概是进程的4倍，从下面的code 200也能看出，线程的性能优于进程，大概快了4倍。原因是线程是更小的执行单位，cpu执行线程时候空闲时间小于进程，且线程的创建和销毁都比进程快，所以线程性能更好。

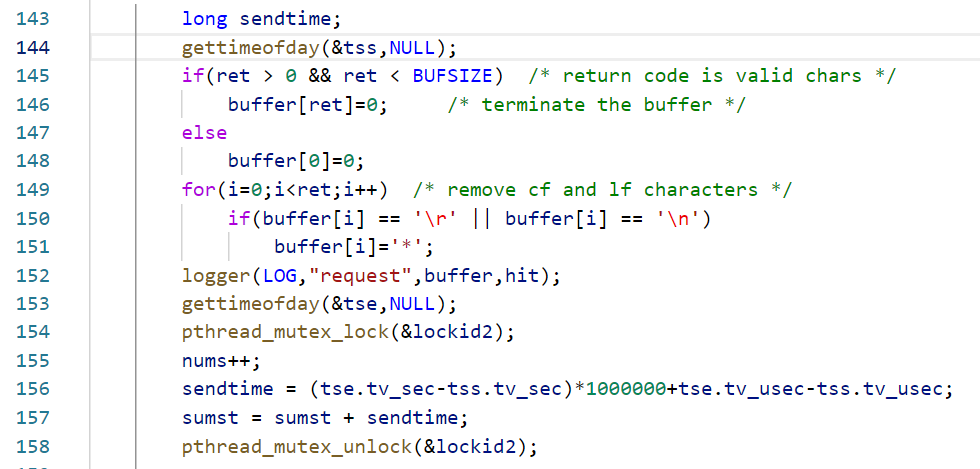
（2）

对比进程和线程模型的socket数据读取、发送、网页文件读取和日志文件写入四个操作，分别计时并算平均值。这里面线程的四个操作计时我加了互斥锁，以保证每个线程的每个操作运行时间能加到共享资源里，所以在web函数里有四个互斥锁，锁加的地方就是线程内资源访问共享资源的地方。如下图：

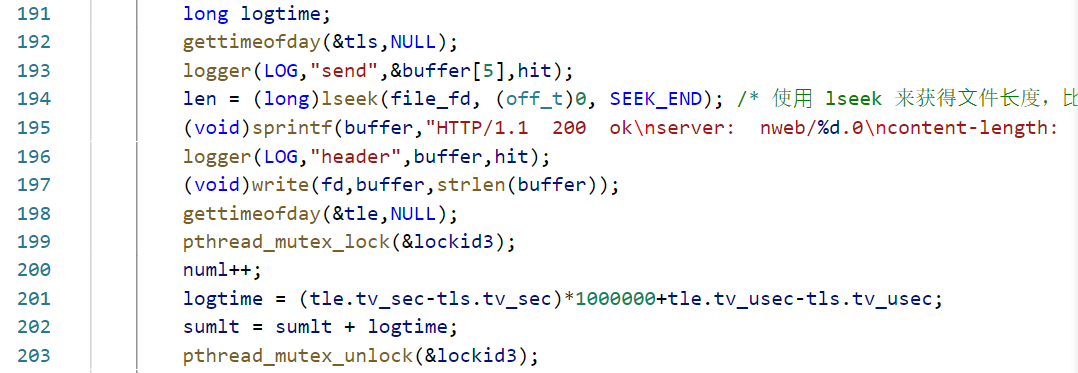


131行是单个线程socket读取的时间，起始点前用trs(time reading start)，结束是tre(time reading end)。互斥锁加在了133-137行，当有线程进入时，条件变量和互斥锁让单个线程进入，其他线程等待在pthread\_mutex\_lock()，一个线程执行完了，unlock线程锁，后面的线程抢先进来一个。readtime是单个线程执行的时间，sumrt是总的执行时间，这是全局变量，用于共享的线程资源。

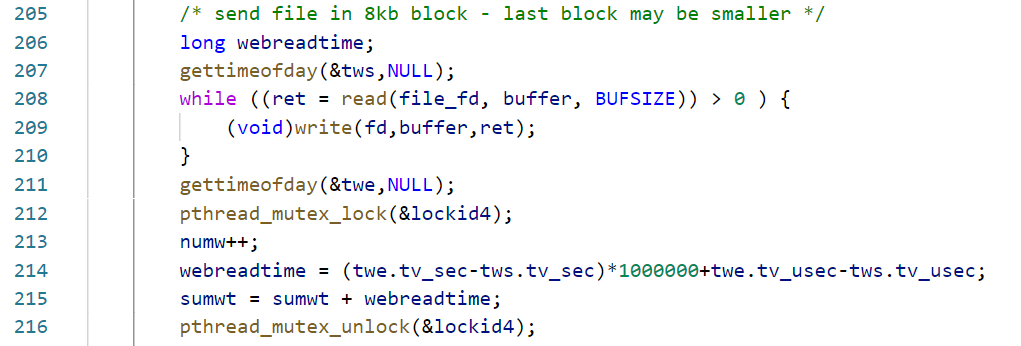
之后的数据发送，网页读取和日志写入，计时方法也同理。每个操作的计时区间如下：



（数据发送）

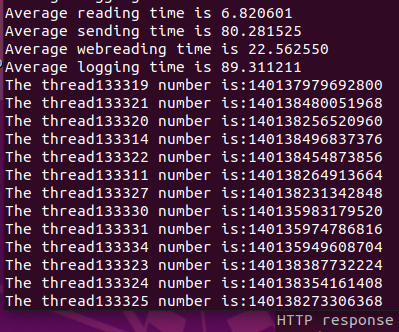


（日志文件写入）



（网页文件读取）

socket数据读取是刚进入web函数后的read；socket数据发送是返回给客户端接收请求；日志文件写入是logger函数，当然要成功访问的日志写入才有效，所以选择logger（LOG…..）之类的；网页文件读取是最后面的ret=read()，当前面没有异常的时候，读取网页文件。



（线程的操作计时）







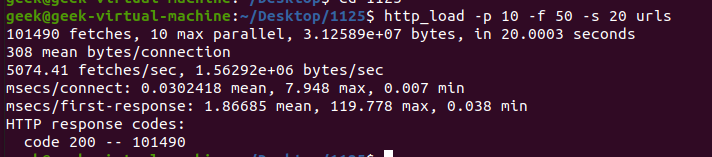


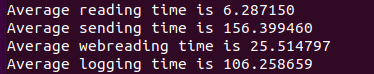
（进程的四个操作计时）

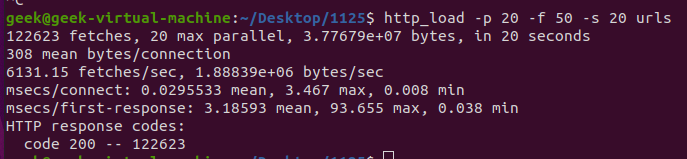
上图是四个操作中线程和进程的分别计时。进程的读取socket数据和发送数据的时间大概是线程的二倍，而网页文件读取的时间大概是四倍，日志文件写入时间几乎相同。结合上面的进程性能比线程低的结论，可知进程之所以在相同条件下发送接收数据量少，因为socket相关操作和网页操作耗时更多，从而降低了性能。

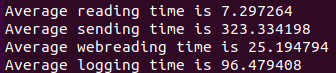
无论是多进程还是多线程，socket发送和日志文件写入，都是很耗时间的，而多进程比线程更多一项的是网页文件的读取。

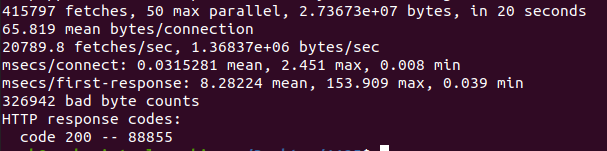
题目二：调整http\_load 命令参数，增加其并发访问线程数量，会发现随着并发访问数 量在达到一定数量后，再增多会导致多线程Web 服务进程的性能出现下降的现象。试分析 产生上述现象的原因是什么？

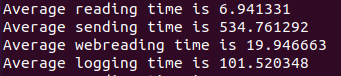


（模拟10个用户请求）



（20个）



（50个）

通过改变p的值，表示并发访问数不同，然后观察bytes和codes的值，可发现20个线程时候code值还是12万多，传输字节数也是最高的；而到了50个线程，传输字节数下降了，code值也降到了8万多。我通过前面做过的检测不同web函数中的任务，来看不同数量用户对各个子程序执行时间的影响，结果如上图所示，socket数据发送的时间随着用户数增多基本上呈现线性增长，10个是156us，20个323us，50个534us。几乎呈线性增长，性能下降的原因我认为是由于socket数据发送引起的。代码段详见上一题截图。

2.4 线程池模型

题目1-4：必要的代码已添加，在附录会给出。

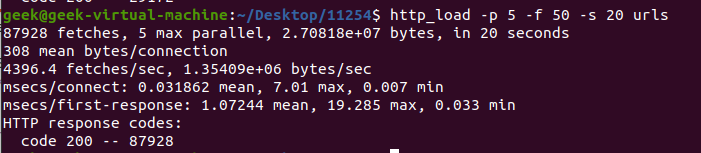
我从线程池中有五十个线程开始，依次累加。观察code(成功访问次数)、bytes和fetches，下面性能均以code值为基准。50个是87928，100个达到13万多，200个线程达到15万，而400个却下降了，所以我在这个区间找最大值，测试了300个线程和250个的情况。结果都没有200个的时候高，由此可知web服务并发性能最优时候，线程个数在200个左右。

比较线程池和多线程：最上面的图相比，同样的模拟5个用户，多线程模型有12万多次成功访问，而线程池开五个线程模拟五个用户，就只有8万次成功访问。但随着线程池中线程数量的增多，性能会先慢慢增强，会达到一个最大值。多线程会随着用户数量增多而性能下降。而线程池中线程数量有一个最优值。

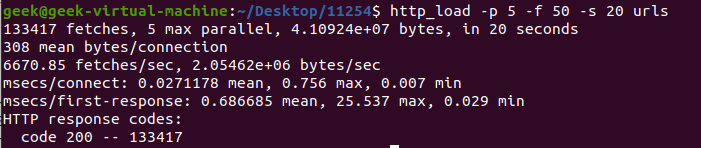
也就是说，二者都会随着线程数量增多而最终使得性能下降，但方式不同，线程池是跟池中线程数有关，而多线程则跟用户数有关。二者可以相互取长补短，从而达到性能最优，具体方法例如使用线程池模型并且增加线程数量，在某个值时候会比多线程同等情况下性能更强。



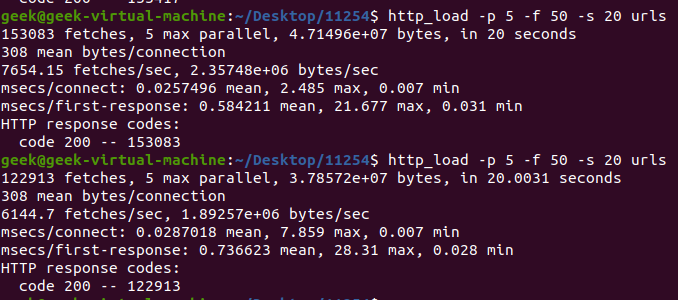
（531行，参数是线程池中线程数量）



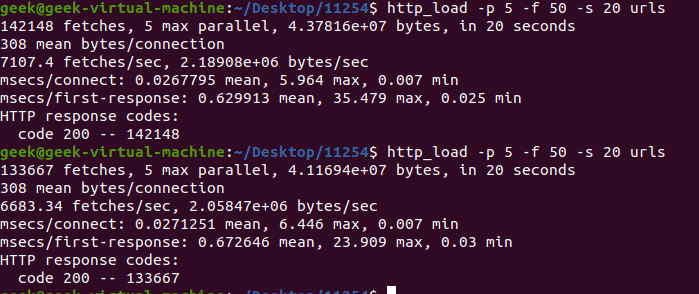
（50个）



（100个）



（200，400个）



(300,250)

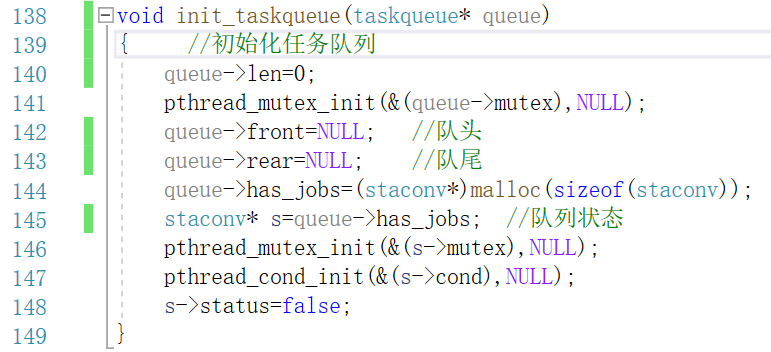
题目1. 添补相应的程序代码到上面函数中“……”位置处。

题目2. 完成函数push\_taskqueue，take\_taskqueue，init\_taskqueue 和destory\_taskqueue。

题目3. 添加必要的程序代码，以最终完成线程池。

题目4. 利用实现的线程池，替换实验3 中Web 服务的多线程模型。

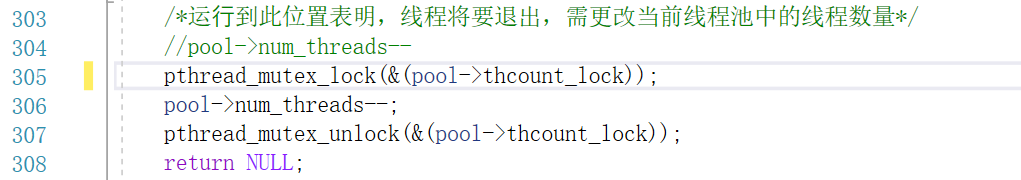
实现init\_taskqueue



实现push\_taskqueue



工作线程数减一，访问临界资源先上锁

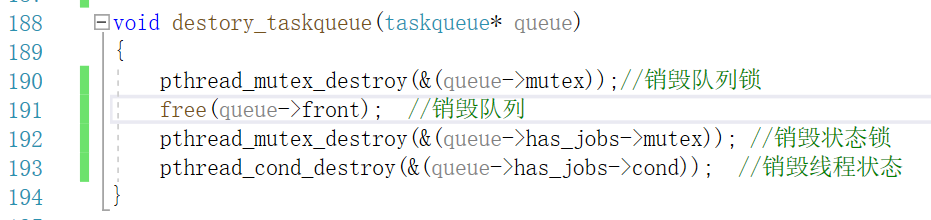


等待当前任务执行完的实现和函数进入条件

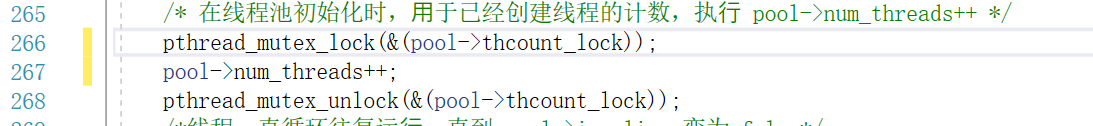


247：销毁线程池中的线程；248：销毁线程池锁；249：销毁线程的条件变量

destroy\_taskqueue实现



线程池中的线程数属于临界资源，上锁后执行。

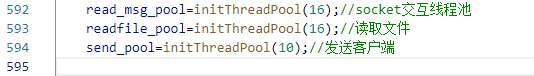


线程执行体的相关实现和说明

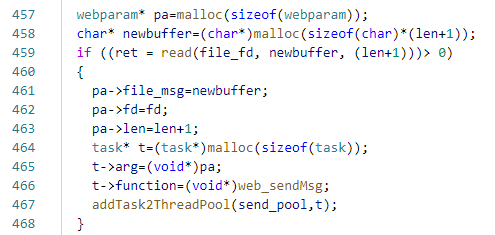


2.5业务分割模型

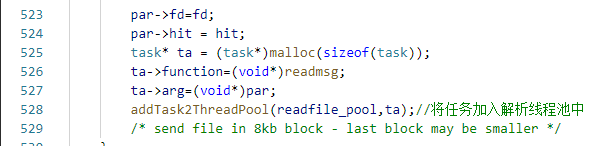
将服务器接收访问的分成三个，即socket交互信息、读取文件和发送客户端，从而创建三个线程池来处理任务。下面代码是三个线程池初始化的代码。



业务分割就是把原先执行的web函数分成三个部分，这三部分间是层级调用关系，web调用readmsg()，readmsg()调用sendmsg()。



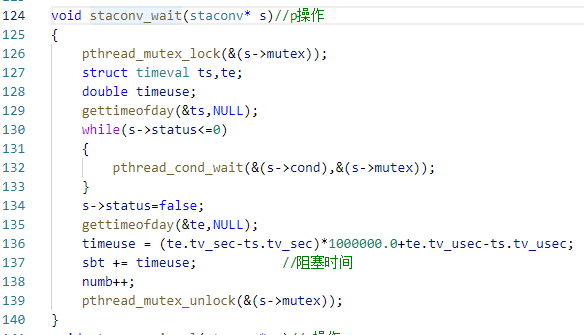
（read调用send）



（web调用read）

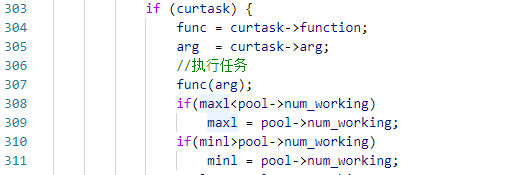
题目二：求性能参数并打印，下面说我的各个性能参数从哪里获得

线程平均阻塞时间在staconv\_wait()函数里，如下图：

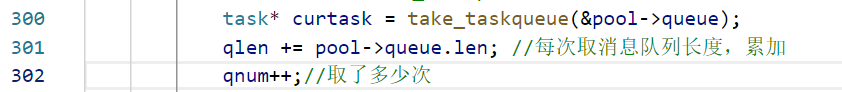




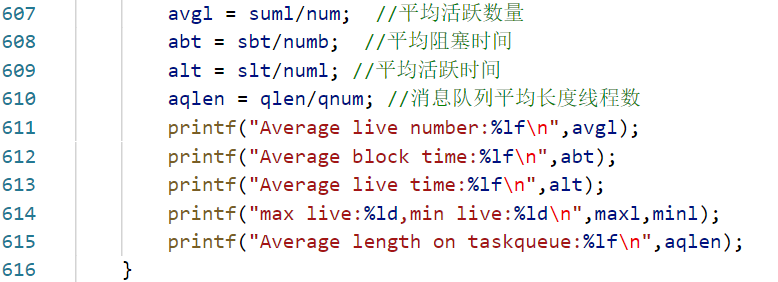
线程平均活跃时间来源上图所示，在thread\_do（）函数里。



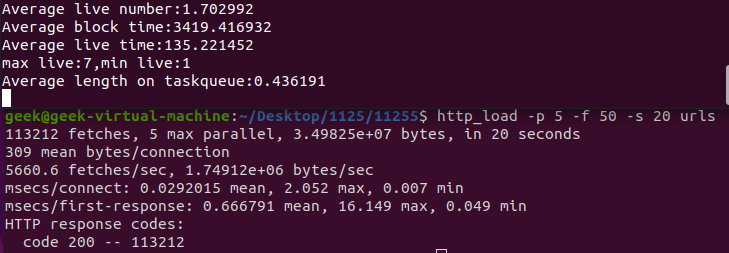
线程最高最低活跃数在活跃线程里寻找，用找最大最小值的方法求得，平均活跃数用每次算的活跃线程数加和，再计数算了多少次活跃线程数，然后相除。



消息队列的消息长度计算，在thread\_do函数中。

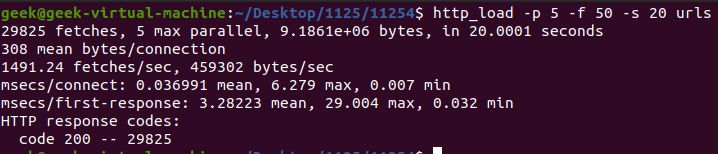


回到main函数中，由于前面开的都是全局变量，是线程的共享资源，所以在main里计算平均数，并输出。我采用http\_load进行测试，实验结果取最后一次，结果下图所示



（16个线程，业务分割）

可见，平均活跃线程数为1.69个；平均阻塞时间3419us；平均活跃时间135us。线程池最大线程数为7，最小为1；平均队列长度0.436。



（16个线程，不分割）

题目三：和线程池的模型相比，开16个线程，业务分割的成功访问次数（code）能达到11万多，是线程池的3倍左右。这说明业务分割有助于增强系统性能。现在程序的瓶颈在于线程阻塞时间远高于活跃时间。故应该控制线程池中线程数量和消息队列长度来优化。

附 源代码：（最终的业务分割版本）

#include <stdio.h>

#include <stdlib.h>

#include <unistd.h>

#include <sys/time.h>

#include <errno.h>

#include <signal.h>

#include <string.h>

#include <fcntl.h>

#include <sys/types.h>

#include <sys/socket.h>

#include <sys/stat.h>

#include <sys/types.h>

#include <sys/mman.h>

#include <semaphore.h>

#include <netinet/in.h>

#include <sys/wait.h>

#include <arpa/inet.h>

#include <time.h>

#include <pthread.h>

#include <sys/prctl.h>

#include <stdbool.h>

#define VERSION 23

#define BUFSIZE 8096

#define ERROR      42

#define LOG        44

#define FORBIDDEN 403

#define NOTFOUND  404

#ifndef SIGCLD

#define SIGCLD SIGCHLD

#endif

struct {

    char \*ext;

    char \*filetype;

}extensions [] = {

{"gif", "image/gif" },

{"jpg", "image/jpg" },

{"jpeg","image/jpeg"},

{"png", "image/png" },

{"ico", "image/ico" },

{"zip", "image/zip" },

{"gz",  "image/gz"  },

{"tar", "image/tar" },

{"htm", "text/html" },

{"html","text/html" },

{0,0}

};

pthread\_mutex\_t lockid1 = PTHREAD\_MUTEX\_INITIALIZER;

pthread\_cond\_t cond1 = PTHREAD\_COND\_INITIALIZER;

pthread\_mutex\_t lockid2 = PTHREAD\_MUTEX\_INITIALIZER;

pthread\_cond\_t cond2 = PTHREAD\_COND\_INITIALIZER;

pthread\_mutex\_t lockid3 = PTHREAD\_MUTEX\_INITIALIZER;

pthread\_cond\_t cond3 = PTHREAD\_COND\_INITIALIZER;

pthread\_mutex\_t lockid4 = PTHREAD\_MUTEX\_INITIALIZER;

pthread\_cond\_t cond4 = PTHREAD\_COND\_INITIALIZER;

typedef struct{

    int fd;

    int hit;

    int file\_fd;

    long len;

    char\* file\_name;

    char\* file\_msg;

}webparam;

/\* queue status and conditional variable\*/

typedef struct staconv {

    pthread\_mutex\_t mutex;

    pthread\_cond\_t  cond;  /\*⽤于阻塞和唤醒线程池中线程\*/

    bool status;            /\*表示任务队列状态：false 为⽆任务；true 为有任务\*/

} staconv;

/\*Task\*/

typedef struct task {

    struct task\* next;            /\* 指向下⼀任务   \*/

    void  (\*function)(void\* arg);  /\*  函数指针   \*/

    void\* arg;                    /\*  函数参数指针  \*/

} task;

/\*Task Queue\*/

typedef struct taskqueue {

    pthread\_mutex\_t mutex;/\* ⽤于互斥读写任务队列 \*/

    task\* front;     /\* 指向队⾸\*/

    task\* rear;      /\* 指向队尾\*/

    staconv\* has\_jobs;    /\* 根据状态，阻塞线程\*/

    int  len;     /\* 队列中任务个数\*/

} taskqueue;

/\* Thread \*/

typedef struct thread {

    int id; /\* 线程 id\*/

    pthread\_t pthread;  /\* 封装的 POSIX 线程\*/

    struct threadpool\* pool;    /\* 与线程池绑定\*/

} thread;

/\*Thread Pool\*/

typedef struct threadpool {

    thread\*\* threads;    /\* 线程指针数组\*/

    volatile int num\_threads;    /\*线程池中线程数量\*/

    volatile int num\_working;  /\*⽬前正在⼯作的线程个数\*/

    pthread\_mutex\_t  thcount\_lock;          /\*线程池锁⽤于修改上⾯两个变量 \*/

    pthread\_cond\_t  threads\_all\_idle;     /\*⽤于销毁线程的条件变量\*/

    taskqueue  queue; /\*任务队列\*/

        volatile bool is\_alive;   /\* 表示线程池是否还存活\*/

}threadpool;

void staconv\_wait(staconv\* s);

void staconv\_signal(staconv\* s);

void init\_taskqueue(taskqueue\* queue);

task\* take\_taskqueue(taskqueue\* queue);

void push\_taskqueue(taskqueue\* queue,task\* curtask);

void destroy\_taskqueue(taskqueue\* queue);

struct threadpool\* initTheadPool(int num\_threads);

void addTask2ThreadPool(threadpool\* pool,task\* curtask);

void waitThreadPool(threadpool\* pool);

void destoryThreadPool(threadpool\* pool);

int getNumofThreadWorking(threadpool\* pool);

int create\_thread(struct threadpool\* pool, struct thread\* pthread, int id);

void\* thread\_do(struct thread\* pthread);

double sbt=0,abt,slt=0,alt,numb=0,numl=0,num=0,suml=0,avgl;

long maxl=0,minl=10000000;

void staconv\_wait(staconv\* s)//p操作

{

    pthread\_mutex\_lock(&(s->mutex));

    struct timeval ts,te;

    double timeuse;

    gettimeofday(&ts,NULL);

    while(s->status<=0)

    {

        pthread\_cond\_wait(&(s->cond),&(s->mutex));

    }

    s->status=false;

    gettimeofday(&te,NULL);

    timeuse = (te.tv\_sec-ts.tv\_sec)\*1000000.0+te.tv\_usec-ts.tv\_usec;

    sbt += timeuse;           //阻塞时间

    numb++;

    pthread\_mutex\_unlock(&(s->mutex));

}

void staconv\_signal(staconv\* s)//v操作

{

    pthread\_mutex\_lock(&(s->mutex));

    s->status=true;

    pthread\_cond\_signal(&(s->cond));

    pthread\_mutex\_unlock(&(s->mutex));

}

void init\_taskqueue(taskqueue\* queue)

{

    queue->len=0;

    pthread\_mutex\_init(&(queue->mutex),NULL);

    queue->front=NULL;

    queue->rear=NULL;

    queue->has\_jobs=(staconv\*)malloc(sizeof(staconv));

    staconv\* s=queue->has\_jobs;

    pthread\_mutex\_init(&(s->mutex),NULL);

    pthread\_cond\_init(&(s->cond),NULL);

    s->status=false;

}

task\* take\_taskqueue(taskqueue\* queue)

{

    pthread\_mutex\_lock(&(queue->mutex));

    task\* t=queue->front;

    if(queue->len==1)

    {

        queue->front=NULL;

        queue->rear=NULL;

        queue->len=0;

    }

    else if(queue->len>1)

    {

        queue->front=t->next;

        queue->len-=1;

        staconv\_signal(queue->has\_jobs);

    }

    pthread\_mutex\_unlock(&(queue->mutex));

    return t;

}

void push\_taskqueue(taskqueue\* queue,task\* curtask)

{

    pthread\_mutex\_lock(&(queue->mutex));

    curtask->next=NULL;

    if(queue->len==0)

    {

        queue->rear=curtask;

        queue->front=curtask;

    }

    else

    {

        queue->rear->next=curtask;

        queue->rear=curtask;

    }

    queue->len++;

    staconv\_signal(queue->has\_jobs);

    pthread\_mutex\_unlock(&(queue->mutex));

}

void destory\_taskqueue(taskqueue\* queue)

{

    pthread\_mutex\_destroy(&(queue->mutex));

    free(queue->front);

    pthread\_mutex\_destroy(&(queue->has\_jobs->mutex));

    pthread\_cond\_destroy(&(queue->has\_jobs->cond));

}

/\*线程池初始化函数\*/

struct threadpool\* initThreadPool(int num\_threads)

{

    //创建线程池空间

    threadpool\* pool;

    pool = (threadpool\*)malloc(sizeof(struct threadpool));

    pool->num\_threads = 0;

    pool->num\_working=0;

    pool->is\_alive=true;

    //初始化互斥量和条件变量

    pthread\_mutex\_init(&(pool->thcount\_lock), NULL);

    pthread\_cond\_init(&pool->threads\_all\_idle, NULL);

    //初始化任务队列

    //\*\*\*\*需实现\*\*\*\*\*

    init\_taskqueue(&pool->queue);

    //创建线程数组

    pool->threads=(struct thread \*)malloc(num\_threads\*sizeof(struct thread\*));

    //创建线程

    for (int i = 0; i < num\_threads; i++) {

        create\_thread(pool,&(pool->threads[i]),i);  //i 为线程 id,

    }

    //等等所有的线程创建完毕,在每个线程运⾏函数中将进⾏ pool->num\_threads++ 操作

    //因此，此处为忙等待，直到所有的线程创建完毕，并⻢上运⾏阻塞代码时才返回。

    while(pool->num\_threads!=num\_threads) {}

    return pool;

}

/\*向线程池中添加任务\*/

void addTask2ThreadPool(threadpool\* pool,task\* curtask)

{

    //将任务加⼊队列

    push\_taskqueue(&pool->queue,curtask);

}

/\*等待当前任务全部运⾏完\*/

void waitThreadPool(threadpool\* pool){

    pthread\_mutex\_lock(&pool->thcount\_lock);

    while (pool->queue.len || pool->num\_working) {

        pthread\_cond\_wait(&pool->threads\_all\_idle, &pool->thcount\_lock);

    }

    pthread\_mutex\_unlock(&pool->thcount\_lock);

}

/\*销毁线程池\*/

void destoryThreadPool(threadpool\* pool){

    //如果当前任务队列中有任务，需等待任务队列为空，并且运⾏线程执⾏完任务后

    if((pool->queue).has\_jobs->status==true)

        waitThreadPool(pool);

    //销毁任务队列

    //\*\*\*\*需实现\*\*\*\*\*

    destory\_taskqueue(&pool->queue);

    //销毁线程指针数组,并释放所有为线程池分配的内存

    free(pool->threads);

    pthread\_mutex\_destroy(&(pool->thcount\_lock));

    pthread\_cond\_destroy(&(pool->threads\_all\_idle));

}

/\*获得当前线程池中正在运⾏线程的数量\*/

int getNumofThreadWorking(threadpool\* pool)

{

    return pool->num\_working;

}

double qlen=0,qnum=0,aqlen;

/\*线程运⾏的逻辑函数\*/

void\* thread\_do(struct thread\* pthread)

{

    /\* 设置线程名字 \*/

    struct timeval ts,te;

    double timeuse;

    char thread\_name[128] = {0};

    sprintf(thread\_name, "thread-pool-%d", pthread->id);

    prctl(PR\_SET\_NAME, thread\_name);

    /\* 获得线程池\*/

    threadpool\* pool = pthread->pool;

    /\* 在线程池初始化时，⽤于已经创建线程的计数，执⾏ pool->num\_threads++ \*/

    pthread\_mutex\_lock(&(pool->thcount\_lock));

    pool->num\_threads++;

    pthread\_mutex\_unlock(&(pool->thcount\_lock));

    /\*线程⼀直循环往复运⾏，直到 pool->is\_alive 变为 false\*/

    while(pool->is\_alive)

    {

        /\*如果任务队列中还要任务，则继续运⾏，否则阻塞\*/

        staconv\* s=(pool->queue).has\_jobs;

        staconv\_wait(s);

        if (pool->is\_alive){

            /\*执⾏到此位置，表明线程在⼯作，需要对⼯作线程数量进⾏计数\*/

            pthread\_mutex\_lock(&(pool->thcount\_lock));

            pool->num\_working++;

            pthread\_mutex\_unlock(&(pool->thcount\_lock));

            /\* 从任务队列的队⾸提取任务，并执⾏\*/

            gettimeofday(&ts,NULL);

            void (\*func)(void\*);

            void\*  arg;

            //take\_taskqueue 从任务队列头部提取任务，并在队列中删除此任务

            //\*\*\*\*需实现 take\_taskqueue\*\*\*\*\*

            task\* curtask = take\_taskqueue(&pool->queue);

            qlen += pool->queue.len; //每次取消息队列长度，累加

            qnum++;//取了多少次

            if (curtask) {

                func = curtask->function;

                arg  = curtask->arg;

                //执⾏任务

                func(arg);

                if(maxl<pool->num\_working)

                    maxl = pool->num\_working;

                if(minl>pool->num\_working)

                    minl = pool->num\_working;

                suml += pool->num\_working;

                num++;

                //释放任务

                free(curtask);

            }

            gettimeofday(&te,NULL);

            timeuse = (te.tv\_sec-ts.tv\_sec)\*1000000.0+te.tv\_usec-ts.tv\_usec;

            slt += timeuse;           //活跃时间

            numl++;

            /\*执⾏到此位置，表明线程已经将任务执⾏完成，需更改⼯作线程数量\*/

           //此处还需注意，当⼯作线程数量为 0，表 示 任 务 全 部 完 成 ，要 让 阻 塞 在 waitThreadPool 函数上的线程继续运⾏

           pthread\_mutex\_lock(&(pool->thcount\_lock));

            pool->num\_working--;

            if(pool->num\_working==0)

                pthread\_cond\_signal(&(pool->threads\_all\_idle));

            pthread\_mutex\_unlock(&(pool->thcount\_lock));

        }

    }

    /\*运⾏到此位置表明，线程将要退出，需更改当前线程池中的线程数量\*/

    //pool->num\_threads--

    pthread\_mutex\_lock(&(pool->thcount\_lock));

    pool->num\_threads--;

    pthread\_mutex\_unlock(&(pool->thcount\_lock));

    return NULL;

}

/\*创建线程\*/

int create\_thread (struct threadpool\* pool, struct thread\* pthread, int id)

{

    //为 thread 分配内存空间

    pthread = (struct thread\*)malloc(sizeof(struct thread));

    if (pthread == NULL){

        perror("creat\_thread(): Could not allocate memory for thread\n");

        return -1;

    }

    //设置这个 thread 的属性

    (pthread)->pool = pool;

    (pthread)->id = id;

    //创建线程

    pthread\_create(&(pthread)->pthread, NULL, (void \*)thread\_do, (pthread));

    pthread\_detach((pthread)->pthread);

    return 0;

}

unsigned long get\_file\_size(const char \*path) {

    unsigned long filesize = -1; struct stat statbuff;

    if(stat(path, &statbuff) < 0)

        return filesize;

    else

        filesize = statbuff.st\_size;

    return filesize;

}

void logger(int type, char \*s1, char \*s2, int socket\_fd)

{

  int fd ;

  struct timeval t1,t2;

  double timeuse;

  char logbuffer[BUFSIZE\*2];

  time\_t timep;

  struct tm \*p;

  time(&timep);

  p = gmtime(&timep);

  /\*根据消息类型，将消息放入logbuffer缓存，或直接将消息通过socket通道返回给客户端\*/

  gettimeofday(&t1,NULL);

  switch (type) {

  case ERROR:

    (void)sprintf(logbuffer,"date:%d\\%d\\%d %d:%d:%d\nERROR: %s:%s Errno=%d exiting pid=%d",(1900+p->tm\_year),(1+p->tm\_mon),p->tm\_mday,p->tm\_hour+8,p->tm\_min,p->tm\_sec,s1, s2, errno,getpid());

    break;

  case FORBIDDEN:

    (void)write(socket\_fd, "HTTP/1.1 403 Forbidden\nContent-Length: 185\nConnection: close\nContent-Type: text/html\n\n<html><head>\n<title>403 Forbidden</title>\n</head><body>\n<h1>Forbidden</h1>\n The requested URL, file type or operation is not allowed on this simple static file webserver.\n</body></html>\n",271);

    (void)sprintf(logbuffer,"date:%d\\%d\\%d %d:%d:%d\nFORBIDDEN: %s:%s",(1900+p->tm\_year),(1+p->tm\_mon),p->tm\_mday,p->tm\_hour+8,p->tm\_min,p->tm\_sec,s1, s2);

    break;

  case NOTFOUND:

    (void)write(socket\_fd,  "HTTP/1.1  404  Not  Found\nContent-Length:  136\nConnection: close\nContent-Type:  text/html\n\n<html><head>\n<title>404  Not Found</title>\n</head><body>\n<h1>Not  Found</h1>\nThe  requested  URL  was  not  found  on  this server.\n</body></html>\n",224);

    (void)sprintf(logbuffer,"date:%d\\%d\\%d %d:%d:%d\nNOT FOUND: %s:%s",(1900+p->tm\_year),(1+p->tm\_mon),p->tm\_mday,p->tm\_hour+8,p->tm\_min,p->tm\_sec,s1, s2);

    break;

  case LOG:

    (void)sprintf(logbuffer,"date:%d\\%d\\%d %d:%d:%d\nINFO: %s:%s:%d",(1900+p->tm\_year),(1+p->tm\_mon),p->tm\_mday,p->tm\_hour+8,p->tm\_min,p->tm\_sec,s1, s2,socket\_fd);

    break;

  }

  gettimeofday(&t2,NULL);

  timeuse = (t2.tv\_sec-t1.tv\_sec)\*1000000.0+t2.tv\_usec-t1.tv\_usec;

  /\*printf("Time spent on putting in loggbuffer cache：%lf\n",timeuse);\*/

  /\* 将logbuffer缓存中的消息存入webserver.log文件\*/

  gettimeofday(&t1,NULL);

  if((fd = open("nweb.log", O\_CREAT| O\_WRONLY | O\_APPEND,0644)) >= 0) {

    (void)write(fd,logbuffer,strlen(logbuffer));

    (void)write(fd,"\n",1);

    (void)close(fd);

  }

  gettimeofday(&t2,NULL);

  timeuse = (t2.tv\_sec-t1.tv\_sec)\*1000000.0+t2.tv\_usec-t1.tv\_usec;

  /\*printf("The time of loggbuffer sending to webserver.log：%lf\n",timeuse);\*/

}

threadpool\* read\_msg\_pool; //初始化线程池300个线程容量，且300个线程开始等待任务队列的进来

threadpool\* readfile\_pool;

threadpool\* send\_pool;

void\* web\_sendMsg(void\* data)

{

    webparam\* param=data;

    char\* buffer=param->file\_msg;

    (void)write(param->fd,buffer,param->len);

    close(param->fd);

    free(param->file\_msg);

    free(param);

}

void readmsg(void \*data)

{

     webparam\* param=data;

  int fd;

  int hit,file\_fd,buflen;

  long ret,len,i;

  char\* fstr;

  char\* buffer=param->file\_name;

  buflen=strlen(buffer);

  fd=param->fd;

  hit=param->hit;

  fstr = (char \*)0;

  for(i=0;extensions[i].ext != 0;i++) {

    len = strlen(extensions[i].ext);

    if( !strncmp(&buffer[buflen-len], extensions[i].ext, len)) {

      fstr =extensions[i].filetype;

      break;

    }

  }

  if(fstr == 0) logger(FORBIDDEN,"file extension type not supported",buffer,fd);

  if(( file\_fd = open(&(buffer[5]),O\_RDONLY)) == -1) {  /\* open the file for reading \*/

    logger(NOTFOUND, "failed to open file",&(buffer[5]),fd);

  }

  logger(LOG,"SEND",buffer,hit);

  len = (long)lseek(file\_fd, (off\_t)0, SEEK\_END); /\* lseek to the file end to find the length \*/

        (void)lseek(file\_fd, (off\_t)0, SEEK\_SET); /\* lseek back to the file start ready for reading \*/

          (void)sprintf(buffer,"HTTP/1.1 200 OK\nServer: nweb/%d.0\nContent-Length: %ld\nConnection: close\nContent-Type: %s\n\n", VERSION, len, fstr); /\* Header + a blank line \*/

  logger(LOG,"Header",buffer,hit);

  (void)write(fd,buffer,strlen(buffer));

  webparam\* pa=malloc(sizeof(webparam));

  char\* newbuffer=(char\*)malloc(sizeof(char)\*(len+1));

  if ((ret = read(file\_fd, newbuffer, (len+1)))> 0)

  {

    pa->file\_msg=newbuffer;

    pa->fd=fd;

    pa->len=len+1;

    task\* t=(task\*)malloc(sizeof(task));

    t->arg=(void\*)pa;

    t->function=(void\*)web\_sendMsg;

    addTask2ThreadPool(send\_pool,t);

  }

  free(param);

  close(file\_fd);

}

void \*web(void \*data)

{

    webparam\* par = (webparam\*)malloc(sizeof(webparam));

    par->file\_name = (char\*)malloc(sizeof(char)\*(BUFSIZE+1));

    int fd;

    int hit;

    int j, file\_fd, buflen;

    long i, ret, len;

    char \* fstr;

    struct timeval tss,tse,tws,twe,tls,tle;

    char\* buffer=par->file\_name; /\* static so zero filled \*/

    webparam \*param=(webparam\*) data;

    fd=param->fd; hit=param->hit;

    struct timeval trs,tre;

    long readtime;

    gettimeofday(&trs,NULL);

    ret =read(fd,buffer,BUFSIZE);    /\* read web request in one go \*/

    gettimeofday(&tre,NULL);

    if(ret == 0 || ret == -1) {  /\* read failure stop now \*/

        logger(FORBIDDEN,"failed to read browser request","",fd);

    }

    else{

        if(ret > 0 && ret < BUFSIZE)  /\* return code is valid chars \*/

            buffer[ret]=0;     /\* terminate the buffer \*/

        else

            buffer[0]=0;

        for(i=0;i<ret;i++)  /\* remove cf and lf characters \*/

            if(buffer[i] == '\r' || buffer[i] == '\n')

                buffer[i]='\*';

        logger(LOG,"request",buffer,hit);

        if( strncmp(buffer,"GET ",4) && strncmp(buffer,"get ",4) ) {

            logger(FORBIDDEN,"only simple get operation supported",buffer,fd);

        }

        for(i=4;i<BUFSIZE;i++) { /\* null terminate after the second space to ignore extra stuff \*/

            if(buffer[i] == ' ') { /\* string is "get url " +lots of other stuff \*/

                buffer[i] = 0;

                break;

            }

        }

        for(j=0;j<i-1;j++)    /\* check for illegal parent directory use .. \*/

            if(buffer[j] == '.' && buffer[j+1] == '.') {

                logger(FORBIDDEN,"parent directory (..) path names not supported",buffer,fd);

            }

        if( !strncmp(&buffer[0],"GET /\0",6) || !strncmp(&buffer[0],"get /\0",6) ) /\* convert no filename to index file\*/

            (void)strcpy(buffer,"GET /index.html");

        /\* work out the file type and check we support it \*/

        par->fd=fd;

        par->hit = hit;

        task\* ta = (task\*)malloc(sizeof(task));

        ta->function=(void\*)readmsg;

        ta->arg=(void\*)par;

        addTask2ThreadPool(readfile\_pool,ta);//将任务加入解析线程池中

        /\* send file in 8kb block - last block may be smaller \*/

    }

    free(param);

    /\*printf("The thread%d number is:%lu\n",hit,pthread\_self());\*/

}

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

{

    int i, port, pid, listenfd, socketfd, hit;

    socklen\_t length;

    static struct sockaddr\_in cli\_addr; /\* static = initialised to zeros \*/

    static struct sockaddr\_in serv\_addr; /\* static = initialised to zeros \*/

    sem\_t\* psem;

    int show;

    /\*if((show=sem\_init(psem,0, 1))!=0){

        perror("create semaphore error");

        exit(1);

    }\*/

    if( argc < 3  || argc > 3 || !strcmp(argv[1], "-?") ) {

        (void)printf("hint: nweb Port-Number Top-Directory\t\tversion %d\n\n"

                     "\tnweb is a small and very safe mini web server\n"

                    "\tnweb only servers out file/web pages with extensions named below\n"

                    "\t and only from the named directory or its sub-directories.\n"

                    "\tThere is no fancy features = safe and secure.\n\n"

                    "\tExample: nweb 8181 /home/nwebdir &\n\n"

                    "\tOnly Supports:", VERSION);

        for(i=0;extensions[i].ext != 0;i++)

            (void)printf(" %s",extensions[i].ext);

        (void)printf("\n\tNot Supported: URLs including \"..\", Java, Javascript, CGI\n"

                    "\tNot Supported: directories / /etc /bin /lib /tmp /usr /dev /sbin \n"

                    "\tNo warranty given or implied\n\tNigel Griffiths nag@uk.ibm.com\n");

        exit(0);

    }

    if( !strncmp(argv[2],"/"    ,2 ) || !strncmp(argv[2],"/etc", 5 ) ||

        !strncmp(argv[2],"/bin",5 ) || !strncmp(argv[2],"/lib", 5 ) ||

        !strncmp(argv[2],"/tmp",5 ) || !strncmp(argv[2],"/usr", 5 ) ||

        !strncmp(argv[2],"/dev",5 ) || !strncmp(argv[2],"/sbin",6) ){

          (void)printf("ERROR: Bad top directory %s, see nweb -?\n",argv[2]); exit(3);

    }

    if(chdir(argv[2]) == -1){

        (void)printf("ERROR: Can't Change to directory %s\n",argv[2]);

        exit(4);

    }

    /\* Become deamon + unstopable and no zombies children (= no wait()) \*/

    pid\_t pro = fork();

    (void)signal(SIGCLD, SIG\_IGN); /\* ignore child death \*/

    (void)signal(SIGHUP, SIG\_IGN); /\* ignore terminal hangups \*/

    logger(LOG,"nweb starting",argv[1],getpid());

    /\* setup the network socket \*/

    if((listenfd = socket(AF\_INET, SOCK\_STREAM,0)) <0)

        logger(ERROR, "system call","socket",0);

    port = atoi(argv[1]);

    if(port < 0 || port >60000)

        logger(ERROR,"Invalid port number (try 1->60000)",argv[1],0);

    serv\_addr.sin\_family = AF\_INET;

    serv\_addr.sin\_addr.s\_addr = htonl(INADDR\_ANY);

    serv\_addr.sin\_port = htons(port);

    if(bind(listenfd, (struct sockaddr \*)&serv\_addr,sizeof(serv\_addr)) <0)

        logger(ERROR,"system call","bind",0);

    if( listen(listenfd,64) <0)

        logger(ERROR,"system call","listen",0);

    /\*printf("The child process number is:%d,its father is%d\n",getpid(),getppid());\*/

    read\_msg\_pool=initThreadPool(16);//socket交互线程池

    readfile\_pool=initThreadPool(16);//读取文件

    send\_pool=initThreadPool(10);//发送客户端

    for(hit=1; ;hit++) {

        length = sizeof(cli\_addr);

        if((socketfd = accept(listenfd, (struct sockaddr \*)&cli\_addr, &length)) < 0)

            logger(ERROR,"system call","accept",0);

        webparam \*param=malloc(sizeof(webparam));

        param->hit=hit;

        param->fd=socketfd;

        task\* t = (task\*)malloc(sizeof(task));

        t->function=(void\*)web;

        t->arg=(void\*)param;

        addTask2ThreadPool(read\_msg\_pool,t);//将任务加入解析线程池中

        avgl = suml/num;  //平均活跃数量

        abt = sbt/numb;  //平均阻塞时间

        alt = slt/numl; //平均活跃时间

        aqlen = qlen/qnum; //消息队列平均长度线程数

        printf("Average live number:%lf\n",avgl);

        printf("Average block time:%lf\n",abt);

        printf("Average live time:%lf\n",alt);

        printf("max live:%ld,min live:%ld\n",maxl,minl);

        printf("Average length on taskqueue:%lf\n",aqlen);

    }

}