实验二

计科21-2 2021011587 吴维皓

题目一

```
//多进程处理用户端请求
    for (hit = 1;; hit++)
    {
        length = sizeof(cli_addr);
        if ((socketfd = accept(listenfd, (struct sockaddr *)&cli_addr,
&length)) < 0)
            logger(ERROR, "system call", "accept", 0);

        pid_t pid = fork();

        if (pid == 0)
        {
            web(socketfd, hit); // 子进程处理网页请求
            exit(EXIT_SUCCESS);
        }
        else if (pid > 0)
        {
            close(socketfd);
        }
        else
        {
            perror("fork");
            exit(EXIT_FAILURE);
        }
    }

```

题目二

```
/* webserver.c*/
/*The following main code from https://github.com/ankushagarwal/nweb*, but
they are modified slightly*/

#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <errno.h>
#include <string.h>
#include <fcntl.h>
#include <signal.h>
#include <sys/types.h>
#include <sys/socket.h>
```

```c
#include <sys/wait.h>
#include <sys/time.h>
#include <sys/mman.h>
#include <sys/stat.h>
#include <sys/shm.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <time.h>
#include <semaphore.h>

#define SHARED_MEMORY_NAME "shared_memory" // 确保名称的唯一性
#define SEMAPHORE_NAME "semaphore"
#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 shared_data
{
    double total_time;
};
struct timeval start, end;

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}};

/* 日志函数，将运行过程中的提示信息记录到 webserver.log 文件中*/
void logger(int type, char *s1, char *s2, int socket_fd)
{
    int fd;
    char logbuffer[BUFSIZE * 2];

    time_t t = time(NULL);
    char time_now[30];
    strftime(time_now, sizeof(time_now), "%Y-%m-%d %H:%M:%S",
localtime(&t));
```

```
67
68      /*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*/ switch (type)
69      {
70      case ERROR:
71          (void)sprintf(logbuffer, "%s:ERROR: %s:%s Errno=%d exiting pid=%d", time_now, s1, s2, errno, getpid());
72          break;
73      case FORBIDDEN:
74          (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);
75          (void)sprintf(logbuffer, "%s:FORBIDDEN: %s:%s", time_now, s1, s2);
76          break;
77      case NOTFOUND:
78          (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);
79          (void)sprintf(logbuffer, "%s:NOT FOUND: %s:%s", time_now, s1, s2);
80          break;
81      case LOG:
82          (void)sprintf(logbuffer, "%s:INFO: %s:%s:%d", time_now, s1, s2, socket_fd);
83          break;
84      }
85      /* 将 logbuffer 缓存中的消息存入 webserver.log 文件*/
86      if ((fd = open("webserver.log", O_CREAT | O_WRONLY | O_APPEND, 0644)) >= 0)
87      {
88          (void)write(fd, logbuffer, strlen(logbuffer));
89          (void)write(fd, "\n", 1);
90          (void)close(fd);
91      }
92  }
93
94  /* 此函数完成了 WebServer 主要功能，它首先解析客户端发送的消息，然后从中获取客户端请求的文件名，然后根据文件名从本地将此文件读入缓存，并生成相应的 HTTP 响应消息；最后通过服务器与客户端的 socket 通道向客户端返回 HTTP 响应消息*/
95
96  void web(int fd, int hit)
97  {
98      int j, file_fd, buflen;
99      long i, ret, len;
100     char *fstr;
101     static char buffer[BUFSIZE + 1]; /* 设置静态缓冲区 */
102     ret = read(fd, buffer, BUFSIZE); /* 从连接通道中读取客户端的请求消息 */
103     if (ret == 0 || ret == -1)
104     { // 如果读取客户端消息失败，则向客户端发送 HTTP 失败响应信息
            logger(FORBIDDEN,"failed to read browser request","",fd);
105     }
106     if (ret > 0 && ret < BUFSIZE) /* 设置有效字符串，即将字符串尾部表示为 0 */
```

```
107         buffer[ret] = 0;
108     else
109         buffer[0] = 0;
110     for (i = 0; i < ret; i++) /* 移除消息字符串中的“CF”和“LF”字符*/
111         if (buffer[i] == '\r' || buffer[i] == '\n')
112             buffer[i] = '*';
113     logger(LOG, "request", buffer, hit);
114     /*判断客户端 HTTP 请求消息是否为 GET 类型，如果不是则给出相应的响应消息*/
115     if (strncmp(buffer, "GET ", 4) && strncmp(buffer, "get ", 4))
116     {
117         logger(FORBIDDEN, "Only simple GET operation supported", buffer,
    fd);
118     }
119     for (i = 4; i < BUFSIZE; i++)
120     { /* null terminate after the second space to ignore extra stuff */
121         if (buffer[i] == ' ')
122         { /* string is "GET URL " +lots of other stuff */
123             buffer[i] = 0;
124             break;
125         }
126     }
127     for (j = 0; j < i - 1; j++) /* 在消息中检测路径，不允许路径中出现“.” */
128         if (buffer[j] == '.' && buffer[j + 1] == '.')
129         {
130             logger(FORBIDDEN, "Parent directory (..) path names not
    supported", buffer, fd);
131         }
132     if (!strncmp(&buffer[0], "GET /\0", 6) || !strncmp(&buffer[0], "get
    /\0", 6))
133         /* 如果请求消息中没有包含有效的文件名，则使用默认的文件名 index.html */
    (void)strcpy(buffer, "GET /index.html");
134
135     /* 根据预定义在 extensions 中的文件类型，检查请求的文件类型是否本服务器支持 */
136     buflen = strlen(buffer);
137     fstr = (char *)0;
138     for (i = 0; extensions[i].ext != 0; i++)
139     {
140         len = strlen(extensions[i].ext);
141         if (!strncmp(&buffer[buflen - len], extensions[i].ext, len))
142         {
143             fstr = extensions[i].filetype;
144             break;
145         }
146     }
147     if (fstr == 0)
148         logger(FORBIDDEN, "file extension type not supported", buffer, fd);
149
150     if ((file_fd = open(&buffer[5], O_RDONLY)) == -1)
151     { /* 打开指定的文件名*/
152         logger(NOTFOUND, "failed to open file", &buffer[5], fd);
153     }
154     logger(LOG, "SEND", &buffer[5], hit);
155     len = (long)lseek(file_fd, (off_t)0, SEEK_END); /* 通过 lseek 获取文件长
    度*/
```

```
    (void)lseek(file_fd, (off_t)0, SEEK_SET);       /* 将文件指针移到文件首位置
*/
    /* Header + a blank line */
    (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);
    logger(LOG, "Header", buffer, hit);
    (void)write(fd, buffer, strlen(buffer));

    /* 不停地从文件里读取文件内容，并通过 socket 通道向客户端返回文件内容*/
    while ((ret = read(file_fd, buffer, BUFSIZE)) > 0)
    {
        (void)write(fd, buffer, ret);
    }
    sleep(1); /* sleep 的作用是防止消息未发出，已经将此 socket 通道关闭*/
    close(fd);
}

int main(int argc, char **argv)
{
    int i, port, 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
*/

    /*解析命令参数*/
    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:webserver 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))
```

```
201     {
202         (void)printf("ERROR: Bad top directory %s, see nweb -?\n",
argv[2]);
203         exit(3);
204     }
205     if (chdir(argv[2]) == -1)
206     {
207         (void)printf("ERROR: Can't Change to directory %s\n", argv[2]);
208         exit(4);
209     }
210 
211     /* 建立服务端侦听 socket*/
212     if ((listenfd = socket(AF_INET, SOCK_STREAM, 0)) < 0)
213         logger(ERROR, "system call", "socket", 0);
214     port = atoi(argv[1]);
215     if (port < 0 || port > 60000)
216         logger(ERROR, "Invalid port number (try 1->60000)", argv[1], 0);
217     serv_addr.sin_family = AF_INET;
218     serv_addr.sin_addr.s_addr = htonl(INADDR_ANY);
219     serv_addr.sin_port = htons(port);
220     if (bind(listenfd, (struct sockaddr *)&serv_addr, sizeof(serv_addr)) <
0)
221         logger(ERROR, "system call", "bind", 0);
222     if (listen(listenfd, 64) < 0) // 开始侦听socket连接，最大连接数为64
223         logger(ERROR, "system call", "listen", 0);
224 
225     shm_unlink(SHARED_MEMORY_NAME);
  // 删除共享内存对象，如果之前的程序是被强制退出，则共享内存中的数据会保留
226     int shm_fd = shm_open(SHARED_MEMORY_NAME, O_CREAT | O_RDWR, S_IRUSR |
S_IWUSR);                                       // 创建共享内存
227     ftruncate(shm_fd, sizeof(struct shared_data));
  // 设置共享内存大小
228     struct shared_data *sdata = mmap(NULL, sizeof(struct shared_data),
PROT_READ | PROT_WRITE, MAP_SHARED, shm_fd, 0); // 建立共享内存映射
229 
230     sem_unlink(SEMAPHORE_NAME); // 删除信号量对象
231     sem_t *mutex;               // 创建信号量
232     mutex = sem_open(SEMAPHORE_NAME, O_CREAT, S_IRUSR | S_IWUSR, 1);
233 
234     for (hit = 1;; hit++)
235     {
236         length = sizeof(cli_addr);
237         if ((socketfd = accept(listenfd, (struct sockaddr *)&cli_addr,
&length)) < 0)
238             logger(ERROR, "system call", "accept", 0);
239 
240         pid_t pid = fork();
241 
242         if (pid == 0)
243         {
244 
245             gettimeofday(&start, NULL);
246             web(socketfd, hit); // 子进程处理网页请求
247             gettimeofday(&end, NULL);
```

```
            double tfly = (end.tv_sec - start.tv_sec) * 1000.0 +
(end.tv_usec - start.tv_usec) / 1000.0;
            printf(" 进程:%d 执行时间:%.3f ms\n", getpid(), tfly);

            sem_wait(mutex); // 等待信号量
            sdata->total_time += tfly;
            printf("当前所有子进程消耗时间为:%.3f ms\n", sdata->total_time);
            sem_post(mutex); // 释放信号量

            exit(EXIT_SUCCESS);
        }
        else if (pid > 0)
        {
            close(socketfd);
        }
        else
        {
            perror("fork");
            exit(EXIT_FAILURE);
        }
    }
}

```

```
[linux1@bogon web]$ make
gcc -Wall -Wextra -O2 -o webserver webserver.c -lrt -pthread
[linux1@bogon web]$ ./webserver 8011 ../web
 进程:3412 执行时间:1003.582 ms
当前所有子进程消耗时间为:1003.582 ms
 进程:3413 执行时间:1062.058 ms
当前所有子进程消耗时间为:2065.640 ms
 进程:3414 执行时间:1003.661 ms
当前所有子进程消耗时间为:3069.301 ms
 进程:3415 执行时间:1048.142 ms
当前所有子进程消耗时间为:4117.443 ms
 进程:3420 执行时间:1001.447 ms
当前所有子进程消耗时间为:5118.890 ms
 进程:3422 执行时间:1005.260 ms
当前所有子进程消耗时间为:6124.150 ms
 进程:3423 执行时间:1057.254 ms
当前所有子进程消耗时间为:7181.404 ms
 进程:3424 执行时间:1000.797 ms
当前所有子进程消耗时间为:8182.201 ms
```

题目三:

http_load测试与分析

```
[linux1@bogon web]$ sudo /root/http_load-12mar2006/http_load -p 30 -s 25 url.txt
25 fetches, 30 max parallel, 9550 bytes, in 25.0025 seconds
382 mean bytes/connection
0.999899 fetches/sec, 381.962 bytes/sec
msecs/connect: 0.0818 mean, 0.202 max, 0.013 min
msecs/first-response: 12008.2 mean, 24017.3 max, 0.174 min
HTTP response codes:
  code 200 -- 25
```

- 单进程

```
[linux1@bogon web]$ sudo /root/http_load-12mar2006/http_load -p 30 -s 25 url.txt
3733 fetches, 30 max parallel, 1.01911e+06 bytes, in 25.0111 seconds
273 mean bytes/connection
149.254 fetches/sec, 40746.3 bytes/sec
msecs/connect: 0.0429354 mean, 0.699 max, 0.011 min
msecs/first-response: 2.42542 mean, 22.785 max, 0.347 min
HTTP response codes:
  code 200 -- 3733
```

- 多进程

分析：

1. 每秒响应数量提升：多进程模型下，每秒的响应数量是单进程的1.5倍。这表明多进程能够更有效地处理客户端的网页请求，实现并发处理，提高系统的响应速度。
2. 字节传输量大幅提升：每秒字节传输量是单进程的100倍。这说明多进程架构能够更有效地利用系统资源，提高数据传输效率，从而加速信息交换。
3. 建立请求连接的平均时间优化：多进程模型中，建立请求连接的平均时间比单进程快1倍。这表明多进程的并发处理能力有助于更迅速地建立客户端与服务器之间的连接，提高系统的连接响应速度。
4. 接受服务器第一个响应消息的平均时间显著优化：多进程模型中，接受服务器第一个响应消息的平均时间比单进程快6000倍。这巨大的性能提升表明多进程能够显著减少等待时间，快速响应客户端的首次请求，从而提升用户体验。

性能提升的原因在于多进程Web服务允许子进程并发处理客户端网页请求。在单进程模型中，每个循环只能完成一次网页访问，而多进程模型通过将处理任务交给子进程，实现并发处理。虽然父进程每次循环仍然只完成一次网页访问，但由于并发处理，循环的速度显著提高，无需等待上一次网页访问的完成就能迅速继续监听端口。这有效地提高了系统的吞吐量和响应速度。

性能瓶颈：

1. 子进程的创建和销毁：每次接受到新的连接时，都会创建一个子进程来处理请求。频繁地创建和销毁子进程可能会影响性能，尤其是在并发连接较多的情况下
2. 共享内存和信号量的使用：在多进程环境中，使用共享内存和信号量可能会引入竞争条件；共享内存和信号量的使用会带来一定的进程间通信开销

优化：

1. 使用进程池技术，每次启动的“子进程”从池子里拿，而不是父进程创建，init进程销毁
    - 进程池的作用主要有以下几点：
        - 提高性能：通过预先创建一定数量的进程，可以避免频繁地创建和关闭进程，从而减少系统资源的消耗，提高任务处理效率
        - 并发处理：进程池中的进程可以同时处理多个任务，从而提高了程序的并发处理能力
        - 资源复用：当任务处理完成后，进程可以被放回进程池中等待下一个任务的到来，从而实现了资源的复用
        - 系统稳定性：通过合理地管理和分配进程池中的进程，可以保证系统的稳定性和可靠性
2. 使用轻量级进程间通信方式，管道或消息队列
    - 管道通信优点：
        - 数据传输有序：管道通信可以保证数据的顺序传输（本题中一个进程写，接着立刻读即可）
        - 开销较低：管道通信不需要创建额外的数据结构，只需要使用系统提供的文件描述符即可