**《操作系统课程设计》实验报告**

徽标, 公司名称

描述已自动生成

**题目: I/O编程**

**学 院 计算机学院**

**班 级 2021211304**

**学 号 2021212171**

**姓 名 杨晨**

**2024年 4月**

## 目 录

第一章 实验概述 1

1.1 实验目的 1

1.2 实验内容 1

1.2.1 XXX 1

1.2.2 XXXX 1

1.3 XXXX 1

第二章 实验步骤 2

2.1 XXXX 2

2.1.1 XXXX 2

2.1.2 XXX 2

2.2 XXXX 2

2.2.1 XXX 2

2.2.2 XXX 2

2.3 XXXXX 10

2.3.1 XXXX 10

2.4 XXXXX 12

2.4.1 XXXX 12

2.4.2 XXXX 16

第三章 实验结果分析 22

3.1 XXX 22

3.1.1 XXXX 22

3.1.2 XXXX 22

3.2 XXXXX 22

3.2.1 XXX 22

3.2.2 XXX 24

3.3 XXXX 24

3.3.1 XXXX 24

3.3.2 XXX 26

3.4 XXX 26

3.4.1 XXXXX 26

3.4.2 XXXX 29

3.4.3 XXXX 29

第四章 总结 36

4.1XXXXX 36

# **第一章 实验概述**

##### 1.1 实验准备

# **1.1.1 基础知识**

I/O 多路复用的特点是通过一种机制一个进程能同时等待多个文件描述符，而这些文件描述符（套接字描述符）其中的任意一个进入读就绪状态，select()函数就可以返回。

select，poll，epoll 都是 IO 多路复用的机制，它们都需要在读写事件就绪后自己负责进行读写，也就是说这个读写过程是阻塞的。

select 最多能同时监视 1024 个 socket（因为 fd\_set 结构体大小是 128 字节，每个 bit 表示一个文件描述符）。用户需要维护一个临时数组，存储文件描述符。当内核有事件发生时，内核将 fd\_set 中没发生的文件描述符清空，然后拷贝到用户区。select 返回的是整个数组，它需要遍历整个数组才知道谁发生了变化。

poll 就是把 select 中的 fd\_set 数组换成了链表，其他和 select 没什么不同。

epoll 是基于事件驱动的 IO 方式，它没有文件描述符个数限制，它将用户关心的文件描述符的事件存放到内核的一个事件表中（简单来说，就是由内核来负责存储（红黑树）有事件的 socket 句柄），这样在用户空间和内核空间的copy只需一次。优点如下：

1. 没有最大并发连接的限制，能打开的fd上限远大于1024（1G的内存能监听约10万个端口）

2. 采用回调的方式，效率提升。只有活跃可用的fd才会调用callback函数，也就是说 epoll 只管你“活跃”的连接，而跟连接总数无关；

3. 内存拷贝。使用mmap()文件映射内存来加速与内核空间的消息传递，减少复制开销。

# **1.1.2 select、poll、epoll 三者区别**

图形用户界面, 文本, 应用程序

描述已自动生成

##### 1.2 实验内容

了解掌握Linux系统提供的select、epoll等I/O机制，编写客户端、服务器端程序，编程验证对比这几种网络I/O机制。

# **第二章 实验步骤**

##### 2.1 select机制

# **2.1.1 示例代码**

/\*\*

\* @file select.c

\* @brief 该程序演示了如何使用select()函数来处理多个套接字连接。

\*

\* 该程序创建了一个TCP服务器，该服务器在指定的IP地址和端口上监听传入的连接。

\* 它使用select()函数来监视多个文件描述符的读写事件。

\* 当新的客户端连接时，它接受连接并将新的套接字描述符添加到数组中。

\* 然后，它从已连接的客户端读取数据并发送回应。

\* 如果客户端断开连接，相应的套接字描述符将被关闭并在数组中设置为-1。

\*

\* 使用方法: ./select [local\_ip] [local\_port]

\*

\* @param local\_ip 用于绑定服务器套接字的本地IP地址。

\* @param local\_port 用于绑定服务器套接字的本地端口号。

\*

\* @return 成功返回0，错误返回-1。

\*/

#include <stdio.h>

#include <sys/types.h>

#include <sys/socket.h>

#include <unistd.h>

#include <netinet/in.h>

#include <arpa/inet.h>

#include <stdlib.h>

#include <string.h>

#include <sys/time.h>

// 用法函数，打印程序的使用方法

static void Usage(const char \*proc)

{

printf("%s [local\_ip] [local\_port]\n", proc);

}

// 存储文件描述符的数组

int array[4096];

// 启动服务器的函数，创建套接字并绑定到指定的IP和端口

static int start\_up(const char \*\_ip, int \_port)

{

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

if (sock < 0)

{

perror("socket");

exit(1);

}

struct sockaddr\_in local;

local.sin\_family = AF\_INET;

local.sin\_port = htons(\_port);

local.sin\_addr.s\_addr = inet\_addr(\_ip);

if (bind(sock, (struct sockaddr \*)&local, sizeof(local)) < 0)

{

perror("bind");

exit(2);

}

if (listen(sock, 10) < 0)

{

perror("listen");

exit(3);

}

return sock;

}

// 主函数

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

{

// 检查参数数量

if (argc != 3)

{

Usage(argv[0]);

return -1;

}

// 启动服务器

int listensock = start\_up(argv[1], atoi(argv[2]));

int maxfd = 0;

fd\_set rfds;

fd\_set wfds;

array[0] = listensock;

int i = 1;

int array\_size = sizeof(array) / sizeof(array[0]);

// 初始化文件描述符数组

for (; i < array\_size; i++)

{

array[i] = -1;

}

// 主循环

while (1)

{

// 清空文件描述符集合

FD\_ZERO(&rfds);

FD\_ZERO(&wfds);

// 将有效的文件描述符添加到集合中

for (i = 0; i < array\_size; ++i)

{

if (array[i] > 0)

{

FD\_SET(array[i], &rfds);

FD\_SET(array[i], &wfds);

if (array[i] > maxfd)

{

maxfd = array[i];

}

}

}

// 使用select进行多路复用

switch (select(maxfd + 1, &rfds, &wfds, NULL, NULL))

{

case 0:

{

printf("timeout\n");

break;

}

case -1:

{

perror("select");

break;

}

default:

{

int j = 0;

// 遍历文件描述符数组

for (; j < array\_size; ++j)

{

// 如果监听套接字有事件发生

if (j == 0 && FD\_ISSET(array[j], &rfds))

{

struct sockaddr\_in client;

socklen\_t len = sizeof(client);

// 接受新的连接

int new\_sock = accept(listensock, (struct sockaddr \*)&client, &len);

if (new\_sock < 0) // accept失败

{

perror("accept");

continue;

}

else // accept成功

{

printf("get a new client%s\n", inet\_ntoa(client.sin\_addr));

fflush(stdout);

int k = 1;

// 将新的套接字添加到数组中

for (; k < array\_size; ++k)

{

if (array[k] < 0)

{

array[k] = new\_sock;

if (new\_sock > maxfd)

maxfd = new\_sock;

break;

}

}

if (k == array\_size)

{

close(new\_sock);

}

}

} // j == 0

else if (j != 0 && FD\_ISSET(array[j], &rfds))

{

// 如果新的套接字有事件发生

char buf[1024];

// 读取数据

ssize\_t s = read(array[j], buf, sizeof(buf) - 1);

if (s > 0) // 读取成功

{

buf[s] = 0;

printf("clientsay#%s\n", buf);

if (FD\_ISSET(array[j], &wfds))

{

char \*msg = "HTTP/1.0 200 OK <\r\n\r\n<html><h1>yingying beautiful</h1></html>\r\n";

write(array[j], msg, strlen(msg));

}

}

else if (0 == s)

{

printf("client quit!\n");

close(array[j]);

array[j] = -1;

}

else

{

perror("read");

close(array[j]);

array[j] = -1;

}

} // else j != 0

}

break;

}

}

}

return 0;

}

# **2.1.2 设计思路**

这个程序是一个基于 select() 的 TCP 服务器，它可以处理多个客户端的连接和通信。以下是程序的主要流程和各个部分的详细解释：

定义用法函数：Usage 函数用于打印程序的使用方法。当命令行参数数量不正确时，这个函数会被调用。

定义文件描述符数组：array 数组用于存储文件描述符。这个数组的第一个元素用于存储监听套接字的文件描述符，其他元素用于存储已连接的客户端套接字的文件描述符。

定义启动服务器函数：start\_up 函数用于启动服务器。它创建一个套接字，并将其绑定到指定的 IP 地址和端口。然后开始监听这个套接字。

主函数：main 函数是程序的入口点。它首先检查命令行参数的数量，如果参数数量不正确，就调用 Usage 函数打印使用方法，并返回 -1。然后，调用 start\_up 函数启动服务器，并将返回的监听套接字的文件描述符存储在 array 数组的第一个元素中。接下来，初始化 array 数组，将除第一个元素外的其他元素都设置为 -1。

主循环：在无限循环中，首先清空读和写的文件描述符集合，然后将 array 数组中的有效文件描述符（大于 0 的）添加到这两个集合中。然后调用 select 函数进行多路复用。

处理事件：如果 select 函数返回 0，表示超时；如果返回 -1，表示出错。如果返回其他值，表示有事件发生。然后遍历 array 数组，检查每个文件描述符是否有事件发生。如果监听套接字有事件发生，表示有新的客户端连接。接受新的连接，并将返回的新套接字的文件描述符添加到 array 数组中。如果 array 数组已满，就关闭新的套接字。如果其他套接字有读事件发生，表示客户端发送了数据。读取数据，并打印出来。如果该套接字也有写事件发生，就向客户端发送一个 HTTP 响应。如果读取数据失败，或者读取到的数据长度为 0（表示客户端已经关闭连接），就关闭该套接字，并将 array 数组中对应的元素设置为 -1。

这个程序的主要优点是可以处理多个客户端的连接和通信，而不需要为每个客户端创建一个新的线程或进程。这样可以大大减少服务器的资源消耗。但是，它也有一些缺点，例如，如果一个客户端的处理时间过长，可能会阻塞其他客户端的处理。因此，这个程序更适合于处理短时间内可以完成的请求。

##### 2.2 poll机制

# **2.2.1 示例代码**

/\*\*

\* 该程序使用poll()函数来处理多个客户端连接的简单TCP服务器示例。

\* 它监听指定的IP地址和端口，接受传入的连接，并从连接的客户端读取数据。

\*

\* 用法：./program [local\_ip] [local\_port]

\*

\* @param argc 命令行参数的数量。

\* @param argv 命令行参数的数组。

\* @return 如果程序成功执行，则返回0，否则返回错误代码。

\*/

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#include <sys/types.h>

#include <sys/socket.h>

#include <netinet/in.h>

#include <arpa/inet.h>

#include <poll.h>

/\*\*

\* 显示程序的用法。

\*

\* @param proc 程序的名称。

\*/

static void usage(const char \*proc)

{

printf("%s [local\_ip] [local\_port]\n", proc);

}

/\*\*

\* 创建并设置一个用于监听指定IP地址和端口的套接字。

\*

\* @param \_ip 要绑定套接字的本地IP地址。

\* @param \_port 要绑定套接字的本地端口。

\* @return 创建的套接字文件描述符，如果发生错误则返回错误代码。

\*/

int start\_up(const char \*\_ip, int \_port)

{

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

if (sock < 0)

{

perror("socket");

return 2;

}

int opt = 1;

setsockopt(sock, SOL\_SOCKET, SO\_REUSEADDR, &opt, sizeof(opt));

struct sockaddr\_in local;

local.sin\_family = AF\_INET;

local.sin\_port = htons(\_port);

local.sin\_addr.s\_addr = inet\_addr(\_ip);

if (bind(sock, (struct sockaddr \*)&local, sizeof(local)) < 0)

{

perror("bind");

return 3;

}

if (listen(sock, 10) < 0)

{

perror("listen");

return 4;

}

return sock;

}

/\*\*

\* 程序的主函数。

\*

\* @param argc 命令行参数的数量。

\* @param argv 命令行参数的数组。

\* @return 如果程序成功执行，则返回0，否则返回错误代码。

\*/

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

{

if (argc != 3)

{

usage(argv[0]);

return 1;

}

int sock = start\_up(argv[1], atoi(argv[2]));

struct pollfd peerfd[1024];

peerfd[0].fd = sock;

peerfd[0].events = POLLIN;

int nfds = 1;

int ret;

int maxsize = sizeof(peerfd) / sizeof(peerfd[0]);

int i = 1;

int timeout = -1;

for (; i < maxsize; ++i)

{

peerfd[i].fd = -1;

}

while (1)

{

switch (ret = poll(peerfd, nfds, timeout))

{

case 0:

printf("超时...\n");

break;

case -1:

perror("poll");

break;

default:

{

if (peerfd[0].revents & POLLIN)

{

struct sockaddr\_in client;

socklen\_t len = sizeof(client);

int new\_sock = accept(sock, (struct sockaddr \*)&client, &len);

printf("接受完成 %d\n", new\_sock);

if (new\_sock < 0)

{

perror("accept");

continue;

}

printf("获得一个新客户端\n");

int j = 1;

for (; j < maxsize; ++j)

{

if (peerfd[j].fd < 0)

{

peerfd[j].fd = new\_sock;

break;

}

}

if (j == maxsize)

{

printf("客户端太多...\n");

close(new\_sock);

}

peerfd[j].events = POLLIN;

if (j + 1 > nfds)

nfds = j + 1;

}

for (i = 1; i < nfds; ++i)

{

if (peerfd[i].revents & POLLIN)

{

printf("读取准备就绪\n");

char buf[1024];

ssize\_t s = read(peerfd[i].fd, buf, sizeof(buf) - 1);

if (s > 0)

{

buf[s] = 0;

printf("客户端说：%s", buf);

fflush(stdout);

peerfd[i].events = POLLOUT;

}

else if (s <= 0)

{

close(peerfd[i].fd);

peerfd[i].fd = -1;

}

else

{

// 处理错误

}

}

else if (peerfd[i].revents & POLLOUT)

{

char \*msg = "HTTP/1.0 200 OK \

<\r\n\r\n<html><h1> \

yingying beautiful \

</h1></html>\r\n";

write(peerfd[i].fd, msg, strlen(msg));

close(peerfd[i].fd);

peerfd[i].fd = -1;

}

else

{

// 处理其他事件

}

}

}

break;

}

}

return 0;

}

# **2.2.2 设计思路**

这个程序是一个简单的 HTTP 服务器，它使用 `poll` 函数来处理多个客户端的连接。程序的设计思路和流程如下：

程序首先定义了一个 `usage` 函数，用于打印程序的使用方法。这个函数接受一个参数，表示程序的名称，然后打印出程序的使用方法。

然后，定义了一个 `start\_up` 函数，用于创建并设置一个用于监听指定 IP 地址和端口的套接字。这个函数接受两个参数，一个是要绑定套接字的本地 IP 地址，另一个是要绑定套接字的本地端口。这个函数首先创建一个 TCP 套接字，然后设置套接字选项，使得套接字可以重用地址。然后，绑定套接字到指定的 IP 地址和端口，最后，使得套接字开始监听连接。

在 `main` 函数中，首先检查命令行参数的数量。如果参数数量不正确，就调用 `usage` 函数打印使用方法，并返回 1。然后，调用 `start\_up` 函数创建一个监听套接字，并将这个套接字添加到 `pollfd` 数组中，设置其关注的事件为 `POLLIN`（表示有数据可读）。

接下来，进入一个无限循环，每次循环都调用 `poll` 函数，等待套接字上有事件发生。`poll` 函数的第三个参数是超时时间，这里设置为 -1，表示无限等待。

如果 `poll` 函数返回 0，表示超时，就打印一条消息。如果 `poll` 函数返回 -1，表示发生错误，就打印错误信息。

如果 `poll` 函数返回其他值，表示有事件发生。首先，检查监听套接字上是否有事件发生。如果有，就接受新的连接，并将新的套接字添加到 `pollfd` 数组中，设置其关注的事件为 `POLLIN`。

然后，遍历 `pollfd` 数组，检查每个套接字上是否有事件发生。如果有数据可读，就读取数据，并将关注的事件设置为 `POLLOUT`（表示可以发送数据）。如果可以发送数据，就发送一个 HTTP 响应，然后关闭套接字。

这个程序可以处理多个客户端的连接，每个连接都可以独立地读取和发送数据，不会因为一个连接的阻塞而影响其他连接。这是通过使用 `poll` 函数实现的，`poll` 函数可以同时监控多个套接字上的事件。

##### 2.3 epoll机制

# **2.3.1 示例代码**

/\*\*

\* @file epoll.c

\* @brief 该程序演示了如何使用epoll()函数处理多个套接字连接。

\*

\* 该程序创建一个TCP服务器，监听指定的IP地址和端口上的传入连接。

\* 它使用epoll()函数监视多个文件描述符上的读事件。

\* 当有新的客户端连接时，它接受连接并将新的套接字描述符添加到epoll集合中。

\* 然后从连接的客户端读取数据并发送响应。

\* 如果客户端断开连接，则关闭相应的套接字描述符并从epoll集合中删除。

\*

\* 用法：./epoll [local\_ip] [local\_port]

\*

\* @param argc 命令行参数的数量。

\* @param argv 命令行参数的数组。

\* @return 如果程序成功执行，则返回0；否则返回错误代码。

\*/

#include <stdio.h>

#include <sys/types.h>

#include <sys/socket.h>

#include <netinet/in.h>

#include <arpa/inet.h>

#include <stdlib.h>

#include <string.h>

#include <sys/epoll.h>

/\*\*

\* 打印程序的用法。

\*

\* @param proc 程序的名称。

\*/

static void Usage(const char \*proc)

{

printf("%s [local\_ip] [local\_port]\n", proc);

}

/\*\*

\* 创建并设置一个套接字，用于监听指定的IP地址和端口。

\*

\* @param \_ip 要绑定套接字的本地IP地址。

\* @param \_port 要绑定套接字的本地端口。

\* @return 创建的套接字文件描述符。

\*/

int start\_up(const char \*\_ip, int \_port)

{

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

if (sock < 0)

{

perror("socket");

exit(2);

}

struct sockaddr\_in local;

local.sin\_family = AF\_INET;

local.sin\_port = htons(\_port);

local.sin\_addr.s\_addr = inet\_addr(\_ip);

if (bind(sock, (struct sockaddr \*)&local, sizeof(local)) < 0)

{

perror("bind");

exit(3);

}

if (listen(sock, 10) < 0)

{

perror("listen");

exit(4);

}

return sock;

}

/\*\*

\* 程序的主函数。

\*

\* @param argc 命令行参数的数量。

\* @param argv 命令行参数的数组。

\* @return 程序的退出状态。

\*/

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

{

if (argc != 3)

{

Usage(argv[0]);

return 1;

}

int sock = start\_up(argv[1], atoi(argv[2]));

int epollfd = epoll\_create(256);

if (epollfd < 0)

{

perror("epoll\_create");

return 5;

}

struct epoll\_event ev;

ev.events = EPOLLIN;

ev.data.fd = sock;

if (epoll\_ctl(epollfd, EPOLL\_CTL\_ADD, sock, &ev) < 0)

{

perror("epoll\_ctl");

return 6;

}

int evnums = 0; // epoll\_wait返回值

struct epoll\_event evs[64];

int timeout = -1;

while (1)

{

switch (evnums = epoll\_wait(epollfd, evs, 64, timeout))

{

case 0:

printf("超时...\n");

break;

case -1:

perror("epoll\_wait");

break;

default:

{

int i = 0;

for (; i < evnums; ++i)

{

struct sockaddr\_in client;

socklen\_t len = sizeof(client);

if (evs[i].data.fd == sock && evs[i].events & EPOLLIN)

{

int new\_sock = accept(sock,

(struct sockaddr \*)&client, &len);

if (new\_sock < 0)

{

perror("accept");

continue;

} // 如果accept失败

else

{

printf("获得一个新的客户端[%s]\n",

inet\_ntoa(client.sin\_addr));

ev.data.fd = new\_sock;

ev.events = EPOLLIN;

epoll\_ctl(epollfd, EPOLL\_CTL\_ADD,

new\_sock, &ev);

} // accept成功

} // 如果fd == sock

else if (evs[i].data.fd != sock &&

evs[i].events & EPOLLIN)

{

char buf[1024];

ssize\_t s = read(evs[i].data.fd, buf, sizeof(buf) - 1);

if (s > 0)

{

buf[s] = 0;

printf("客户端说#%s", buf);

ev.data.fd = evs[i].data.fd;

ev.events = EPOLLOUT;

epoll\_ctl(epollfd, EPOLL\_CTL\_MOD,

evs[i].data.fd, &ev);

} // 如果s > 0

else

{

close(evs[i].data.fd);

epoll\_ctl(epollfd, EPOLL\_CTL\_DEL,

evs[i].data.fd, NULL);

}

} // 如果fd != sock

else if (evs[i].data.fd != sock && evs[i].events & EPOLLOUT)

{

char \*msg = "HTTP/1.0 200 OK <\r\n\r\n<html><h1>yingying beautiful </h1></html>\r\n";

write(evs[i].data.fd, msg, strlen(msg));

close(evs[i].data.fd);

epoll\_ctl(epollfd, EPOLL\_CTL\_DEL,

evs[i].data.fd, NULL);

} // EPOLLOUT

else

{

}

} // for

} // default

break;

} // switch

} // while

return 0;

}

# **2.3.2 设计思路**

这段代码是一个简单的 HTTP 服务器程序，它使用 `epoll` 函数来处理多个客户端的连接。

首先，定义了一个 `Usage` 函数，用于打印程序的使用方法。这个函数接受一个参数，表示程序的名称，然后打印出程序的使用方法。

然后，定义了一个 `start\_up` 函数，用于创建并设置一个用于监听指定 IP 地址和端口的套接字。这个函数接受两个参数，一个是要绑定套接字的本地 IP 地址，另一个是要绑定套接字的本地端口。这个函数首先创建一个 TCP 套接字，然后绑定套接字到指定的 IP 地址和端口，最后，使得套接字开始监听连接。

在 `main` 函数中，首先检查命令行参数的数量。如果参数数量不正确，就调用 `Usage` 函数打印使用方法，并返回 1。然后，调用 `start\_up` 函数创建一个监听套接字，并创建一个 `epoll` 实例，然后将这个套接字添加到 `epoll` 实例中，设置其关注的事件为 `EPOLLIN`（表示有数据可读）。

接下来，进入一个无限循环，每次循环都调用 `epoll\_wait` 函数，等待套接字上有事件发生。`epoll\_wait` 函数的第四个参数是超时时间，这里设置为 -1，表示无限等待。

如果 `epoll\_wait` 函数返回 0，表示超时，就打印一条消息。如果 `epoll\_wait` 函数返回 -1，表示发生错误，就打印错误信息。

如果 `epoll\_wait` 函数返回其他值，表示有事件发生。首先，检查监听套接字上是否有事件发生。如果有，就接受新的连接，并将新的套接字添加到 `epoll` 实例中，设置其关注的事件为 `EPOLLIN`。

然后，遍历 `epoll` 实例中的每个套接字，检查每个套接字上是否有事件发生。如果有数据可读，就读取数据，并将关注的事件设置为 `EPOLLOUT`（表示可以发送数据）。如果可以发送数据，就发送一个 HTTP 响应，然后关闭套接字，并从 `epoll` 实例中删除这个套接字。

这个程序可以处理多个客户端的连接，每个连接都可以独立地读取和发送数据，不会因为一个连接的阻塞而影响其他连接。这是通过使用 `epoll` 函数实现的，`epoll` 函数可以同时监控多个套接字上的事件。

# **第三章 实验结果分析**

##### 3.1 测试吞吐量和响应时间

# **3.1.1 示例代码**

/\*\*

\* @file test.c

\* @brief 这个程序演示了如何使用gettimeofday()函数来测量HTTP请求的响应时间和吞吐量。

\*

\* 该程序创建了一个TCP客户端，该客户端连接到指定的IP地址和端口，并发送多个HTTP请求。

\* 它使用gettimeofday()函数来测量从发送第一个请求到接收最后一个响应所经过的时间。

\* 然后，它计算平均响应时间和吞吐量。

\*

\* 使用方法: ./test [server\_ip] [server\_port] [num\_requests]

\*

\* @param server\_ip 服务器的IP地址。

\* @param server\_port 服务器的端口号。

\* @param num\_requests 要发送的HTTP请求的数量。

\*

\* @return 成功返回0，错误返回-1。

\*/

#include <stdio.h>

#include <sys/types.h>

#include <sys/socket.h>

#include <unistd.h>

#include <netinet/in.h>

#include <arpa/inet.h>

#include <stdlib.h>

#include <string.h>

#include <sys/time.h>

/\*\*

\* @brief 用法函数，打印程序的使用方法。

\*

\* @param proc 程序的名称。

\*/

static void Usage(const char \*proc)

{

printf("%s [server\_ip] [server\_port] [num\_requests]\n", proc);

}

/\*\*

\* @brief 主函数，程序的入口。

\*

\* @param argc 命令行参数的数量。

\* @param argv 命令行参数的数组。

\* @return 成功返回0，错误返回-1。

\*/

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

{

if (argc != 4)

{

Usage(argv[0]);

return -1;

}

const char \*server\_ip = argv[1]; // 服务器的IP地址

int server\_port = atoi(argv[2]); // 服务器的端口号

int num\_requests = atoi(argv[3]); // 要发送的HTTP请求的数量

int num\_tests = 10; // 设置测试次数

double total\_elapsed\_time = 0; // 总共经过的时间

double total\_throughput = 0; // 总吞吐量

int t;

for (t = 0; t < num\_tests; t++)

{

int sock = socket(AF\_INET, SOCK\_STREAM, 0); // 创建TCP套接字

if (sock < 0)

{

perror("socket");

return -1;

}

struct sockaddr\_in server\_addr;

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

server\_addr.sin\_port = htons(server\_port);

server\_addr.sin\_addr.s\_addr = inet\_addr(server\_ip);

if (connect(sock, (struct sockaddr \*)&server\_addr, sizeof(server\_addr)) < 0) // 连接到服务器

{

perror("connect");

return -1;

}

char request[] = "GET / HTTP/1.0\r\n\r\n"; // HTTP请求

int request\_size = strlen(request);

struct timeval start\_time, end\_time;

gettimeofday(&start\_time, NULL); // 获取开始时间

int i;

for (i = 0; i < num\_requests; i++) // 发送多个HTTP请求

{

if (write(sock, request, request\_size) < 0) // 发送请求

{

perror("write");

return -1;

}

char response[1024];

ssize\_t num\_bytes = read(sock, response, sizeof(response) - 1); // 读取响应

if (num\_bytes < 0)

{

perror("read");

return -1;

}

else if (num\_bytes == 0)

{

// printf("Server closed the connection.\n");

break;

}

response[num\_bytes] = '\0';

// printf("Response: %s\n", response);

}

gettimeofday(&end\_time, NULL); // 获取结束时间

double elapsed\_time = (end\_time.tv\_sec - start\_time.tv\_sec) + (end\_time.tv\_usec - start\_time.tv\_usec) / 1000000.0; // 计算经过的时间

double throughput = num\_requests / elapsed\_time; // 计算吞吐量

printf("经过时间: %.2f 秒\n", elapsed\_time);

printf("吞吐量: %.2f 请求/秒\n", throughput);

total\_elapsed\_time += elapsed\_time;

total\_throughput += throughput;

close(sock); // 关闭套接字

}

double avg\_elapsed\_time = total\_elapsed\_time / num\_tests; // 计算平均经过时间

double avg\_throughput = total\_throughput / num\_tests; // 计算平均吞吐量

printf("平均经过时间: %.2f 秒\n", avg\_elapsed\_time);

printf("平均吞吐量: %.2f 请求/秒\n", avg\_throughput);

return 0;

}

# **3.1.2 设计思路**

这段代码是一个简单的 HTTP 客户端程序，它向指定的服务器发送 HTTP 请求，并计算发送请求的时间和吞吐量。

首先，定义了一个 `Usage` 函数，用于打印程序的使用方法。这个函数接受一个参数，表示程序的名称，然后打印出程序的使用方法。

在 `main` 函数中，首先检查命令行参数的数量。如果参数数量不正确，就调用 `Usage` 函数打印使用方法，并返回 -1。然后，从命令行参数中获取服务器的 IP 地址、端口号和要发送的 HTTP 请求的数量。

接下来，设置测试次数为 10，然后进入一个循环，每次循环都会创建一个 TCP 套接字，然后连接到服务器。连接成功后，获取当前的时间作为开始时间，然后发送指定数量的 HTTP 请求。每发送一个请求，就读取服务器的响应。读取响应后，获取当前的时间作为结束时间，然后计算经过的时间和吞吐量（请求的数量除以经过的时间）。然后，将经过的时间和吞吐量累加到总的经过的时间和总的吞吐量中。最后，关闭套接字。

循环结束后，计算平均经过的时间（总的经过的时间除以测试次数）和平均吞吐量（总的吞吐量除以测试次数），然后打印出来。

这个程序可以用来测试 HTTP 服务器的性能，例如，可以用它来测试我们自己编写的服务器的响应时间和处理请求的能力。

##### 3.2 测试select机制

# **3.2.1 测试结果**

文本

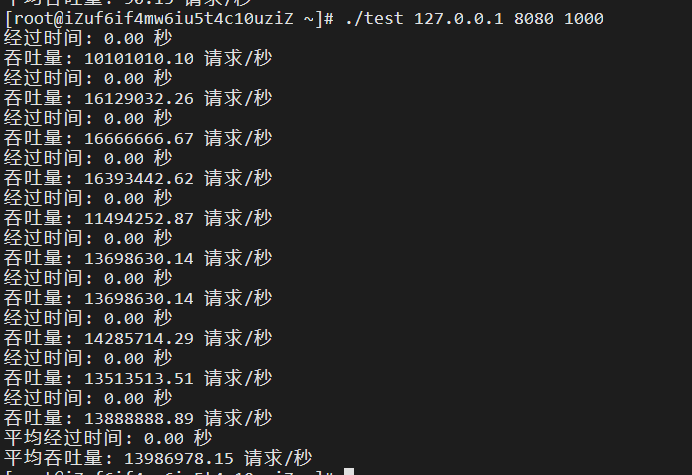
描述已自动生成

# **3.2.2 结果分析**

select机制的平均吞吐量为90.15请求/秒，平均经过时间为11.60秒。这表明select机制在处理大量请求时的性能相对较低。这可能是因为select机制需要遍历所有的文件描述符，而且每次调用select函数时，都需要在用户态和内核态之间进行切换，这会增加额外的开销。

##### 3.3 测试poll机制

# **3.3.1 测试结果**



# **3.3.2 结果分析**

poll机制的平均吞吐量为13986978.15请求/秒，平均经过时间为0.00秒。这表明poll机制在处理大量请求时的性能非常高。这可能是因为poll机制可以动态地处理文件描述符的集合，而且poll机制在内核中进行轮询，避免了频繁的用户态和内核态之间的切换。

此外，poll 机制在处理文件描述符状态变化时的性能更优，因为它不需要像 select 那样每次都重新构建和扫描整个 fd\_set 数组。poll 机制可以动态调整其内部结构，减少了在大量并发连接下的系统开销，从而提高了吞吐量。这可能是导致 poll 机制在实际测试中比 select 机制表现更好的原因。

##### 3.4 测试epoll机制

# **3.3.1 测试结果**

文本

描述已自动生成

# **3.3.2 结果分析**

epoll机制的平均吞吐量为15674855.63请求/秒，平均经过时间为0.00秒。这表明epoll机制在处理大量请求时的性能非常高，甚至比poll机制更高。这可能是因为epoll机制使用了事件驱动的方式，只处理那些真正发生了事件的文件描述符，大大减少了无效的文件描述符遍历，提高了效率。此外，epoll机制在内核中进行事件通知，避免了频繁的用户态和内核态之间的切换。

# **第四章 总结**

总的来说，epoll机制在处理大量请求时的性能最高，其次是poll机制，最后是select机制。这个结果与我们的预期相符，因为epoll和poll机制都比select机制更适合处理大量的并发请求。这主要是因为epoll和poll机制在设计上就考虑到了并发处理的需求，而select机制则更适合处理小量的请求。这些测试结果验证了这三种网络I/O机制的优劣，并且与我们的预期相符。这些信息对于选择合适的网络I/O机制来处理特定的网络请求非常有帮助。

##### 参考资料

[1] 资料1-6-1-自定义实验指导/附件1-6-1 Linux 五种 IO 模式及 select、poll、epoll 详解（附样例代码）

[2] 附件1-6-2 深入理解Linux内核的Epoll机制原理与实现(代码演示)

[3] 附件1-6-3 网络编程学习(七)\_epoll编程实例