**郑州大学国际学院**

**计算机科学与技术专业**

## 《操作系统》实验设计作业

**专业班级 计算机科学与技术一班**

**学 号 202284160112**

**姓 名 廖正**

**任课老师 申丰山**

**报告日期 5月24日**

**------------------------------------------------------------------------------------------**

**本实验报告为实验成绩评定的主要依据，需要提交纸质版和电子版存档。实验报告要求：**

**（1）在第14周（5月25日）之前完成并提交纸质版和电子版；**

**（2）给出源程序（默认编程语言为C语言，批处理脚本为Linux Shell脚本）和运行方法以及运行结果。添加必要的注释；**

**（3）实验报告应完备，包含准备答辩、演示所需完整信息；**

**（4）不按默认或规定语言、脚本编程的实验报告为不合格报告，成绩为不及格；**

**（5）目前程序运行界面为字符界面，而非图形用户界面。在截图中给出程序及文字类运行结果者为不合格报告，成绩为不及格；**

**（6）排版整洁、规范，单倍行距，五号字体，双面打印（勿改）。内容组织条理清晰；如果文中有图表，则图表中文字字号不应大于正文字号，图、表应分别有图号图名、表号表名，图框长宽要与其中的文字数量相适宜，具体参考书本、课本、论文中的图表格式。**

**（7）纸质版封皮要手签姓名，电子版封皮信息要填写完整；**

**（8）雷同者为不合格报告，成绩为不及格；**

**（9）**请保证质量完成，成绩将根据质量确定；

**（10）此实验报告相当于考试，只有一次提交机会，不遵守上述要求者，实验成绩按零分计算。若有疑问，可在提交前咨询老师。**

----------------------------------------------------------------------------------------------

1、编写分析rCore操作系统源程序结构的Shell脚本，完成如下两个任务。（30分）

（1）从每个源程序文件中提取函数签名即函数头和文件中的全局变量定义（即不在结构体和函数中的变量定义），例如loader.rs中的函数头fn get\_sp(&self) -> usize和结构体类型名称struct UserStack以及其它全局简单变量定义，产生类似如下格式的结果：

@startuml

set namespaceSeparator none

……

class "3.loader.rs" as 3loader {

{field} struct KernelStack

{field} struct UserStack

{field} static KERNEL\_STACK: [KernelStack; MAX\_APP\_NUM]

{field} static USER\_STACK: [UserStack; MAX\_APP\_NUM]

{method} 3.1 KernelStack::fn get\_sp(&self) -> usize

{method} 3.2 pub fn push\_context(&self, trap\_cx: TrapContext) -> usize

{method} 3.3 UserStack:: fn get\_sp(&self) -> usize

{method} 3.4 fn get\_base\_i(app\_id: usize) -> usize

{method} 3.5 pub fn get\_num\_app() -> usize

{method} 3.6 pub fn load\_apps()

{method} 3.7 pub fn init\_app\_cx(app\_id: usize) -> usize

}

……

class "11.trap/context.rs" as 11trap\_context {

{field} pub struct TrapContext

{method} 11.1 pub fn TrapContext::set\_sp(&mut self, sp: usize)

{method} 11.2 pub fn TrapContext::app\_init\_context(entry: usize, sp: usize) -> Self

}

……

A --> B: B.F2

C \*-- D: define

3loader --> 11trap\_context:set\_sp

11trap\_context --> 3loader:push\_context

11trap\_context --> 3loader: init\_app\_cx

@enduml

//代码如下

#!/bin/bash

# 将此处SRC\_DIR的值改为自己文件的路径

SRC\_DIR="/home/oslab/rCore-Tutorial-v3"

OUTPUT\_FILE="rCore-Class-ch2.txt"

echo "@startuml" > "$OUTPUT\_FILE"

echo "set namespaceSeparator none" >> "$OUTPUT\_FILE"

echo "skinparam classAttributeIconSize 0" >> "$OUTPUT\_FILE"

echo "skinparam ArrowColor Black" >> "$OUTPUT\_FILE"

echo "skinparam classBackgroundColor #FDF6E3" >> "$OUTPUT\_FILE"

declare -A class\_id\_map

counter=1

# 构建类图

for file in $(find "$SRC\_DIR" -name "\*.rs"); do

relative\_path=$(realpath --relative-to="$SRC\_DIR" "$file")

class\_base=$(echo "$relative\_path" | sed 's|.rs$||; s|/|\_|g')

class\_name="${counter}${class\_base//./\_}"

class\_id\_map["$file"]="$class\_name"

echo "class \"$relative\_path\" as ${class\_name} {" >> "$OUTPUT\_FILE"

# 结构体

grep -E "^\s\*(pub\s+)?struct\s+[A-Za-z0-9\_]+" "$file" | while read -r line; do

struct\_name=$(echo "$line" | awk '{print $2}')

echo " struct $struct\_name" >> "$OUTPUT\_FILE"

done

# 静态变量

grep -E "^\s\*static\s+[A-Za-z0-9\_]+\s\*:" "$file" | while read -r line; do

var\_name=$(echo "$line" | awk '{print $2}')

echo " static $var\_name" >> "$OUTPUT\_FILE"

done

# 函数签名

grep -E "^\s\*(pub\s+)?fn\s+[A-Za-z0-9\_]+\(" "$file" | while read -r line; do

func\_sig=$(echo "$line" | sed 's/^ \*//' | sed 's/{$//')

echo " $func\_sig" >> "$OUTPUT\_FILE"

done

echo "}" >> "$OUTPUT\_FILE"

((counter++))

done

# 添加类之间的调用关系（箭头 + 方法名）

declare -A call\_map

for caller\_file in "${!class\_id\_map[@]}"; do

caller\_class=${class\_id\_map[$caller\_file]}

grep -hEo '[A-Za-z0-9\_:]+::[A-Za-z0-9\_]+\(' "$caller\_file" | sort -u | while read -r call\_line; do

method\_name=$(echo "$call\_line" | sed -E 's/.\*::([a-zA-Z0-9\_]+)\(/\1/')

for callee\_file in "${!class\_id\_map[@]}"; do

if [ "$caller\_file" == "$callee\_file" ]; then continue; fi

if grep -qE "fn\s+$method\_name(\s\*<|[\s(])" "$callee\_file"; then

callee\_class=${class\_id\_map[$callee\_file]}

echo "$caller\_class --> $callee\_class : $method\_name" >> "$OUTPUT\_FILE"

fi

done

done

done

echo "@enduml" >> "$OUTPUT\_FILE"

echo "UML 类图描述已生成: $OUTPUT\_FILE"

//结果如下

@startuml

set namespaceSeparator none

class "os\_build" as os\_build {

{field} static TARGET\_PATH:

}

class "os\_src\_sbi" as os\_src\_sbi {

{method} fn fn

{method} fn fn

{method} fn fn

}

class "os\_src\_trap\_context" as os\_src\_trap\_context {

{field} struct struct

{method} fn fn

{method} fn fn

}

class "os\_src\_trap\_mod" as os\_src\_trap\_mod {

{method} fn fn

{method} fn fn

}

class "os\_src\_logging" as os\_src\_logging {

{field} static LOGGER:

{method} fn fn

}

class "os\_src\_lang\_items" as os\_src\_lang\_items {

}

class "os\_src\_sync\_up" as os\_src\_sync\_up {

{field} struct struct

{method} fn fn

}

class "os\_src\_sync\_mod" as os\_src\_sync\_mod {

}

class "os\_src\_main" as os\_src\_main {

{method} fn fn

}

class "os\_src\_batch" as os\_src\_batch {

{field} static KERNEL\_STACK:

{field} static USER\_STACK:

{method} fn fn

{method} fn fn

{method} fn fn

{method} fn fn

{method} fn fn

{method} fn fn

{method} fn fn

}

class "os\_src\_console" as os\_src\_console {

{method} fn fn

}

class "os\_src\_syscall\_process" as os\_src\_syscall\_process {

{method} fn fn

}

class "os\_src\_syscall\_fs" as os\_src\_syscall\_fs {

{method} fn fn

}

class "os\_src\_syscall\_mod" as os\_src\_syscall\_mod {

{method} fn fn

}

class "user\_src\_lang\_items" as user\_src\_lang\_items {

}

class "user\_src\_console" as user\_src\_console {

{method} fn fn

}

class "user\_src\_syscall" as user\_src\_syscall {

{method} fn fn

{method} fn fn

}

class "user\_src\_bin\_01store\_fault" as user\_src\_bin\_01store\_fault {

}

class "user\_src\_bin\_03priv\_inst" as user\_src\_bin\_03priv\_inst {

}

class "user\_src\_bin\_04priv\_csr" as user\_src\_bin\_04priv\_csr {

}

class "user\_src\_bin\_02power" as user\_src\_bin\_02power {

}

class "user\_src\_bin\_00hello\_world" as user\_src\_bin\_00hello\_world {

}

class "user\_src\_lib" as user\_src\_lib {

{method} fn fn

{method} fn fn

}

os\_src\_main --> os\_src\_trap\_mod: init

os\_src\_main --> os\_src\_trap\_mod: init

os\_src\_main --> os\_src\_trap\_mod: init

os\_src\_main --> os\_src\_logging: init

os\_src\_main --> os\_src\_logging: init

os\_src\_main --> os\_src\_logging: init

os\_src\_main --> os\_src\_batch: run\_next\_app

os\_src\_main --> os\_src\_batch: init

os\_src\_main --> os\_src\_batch: init

os\_src\_main --> os\_src\_batch: init

os\_src\_main --> os\_src\_batch: run\_next\_app

os\_src\_batch --> os\_src\_trap\_context: app\_init\_context

os\_src\_batch --> os\_src\_sync\_up: new

os\_src\_console --> os\_src\_sbi: console

os\_src\_console --> os\_src\_sbi: console

user\_src\_console --> os\_src\_sbi: console

user\_src\_console --> os\_src\_sbi: console

@enduml

（2）在产生上述结果的过程中也要捕捉如下信息。

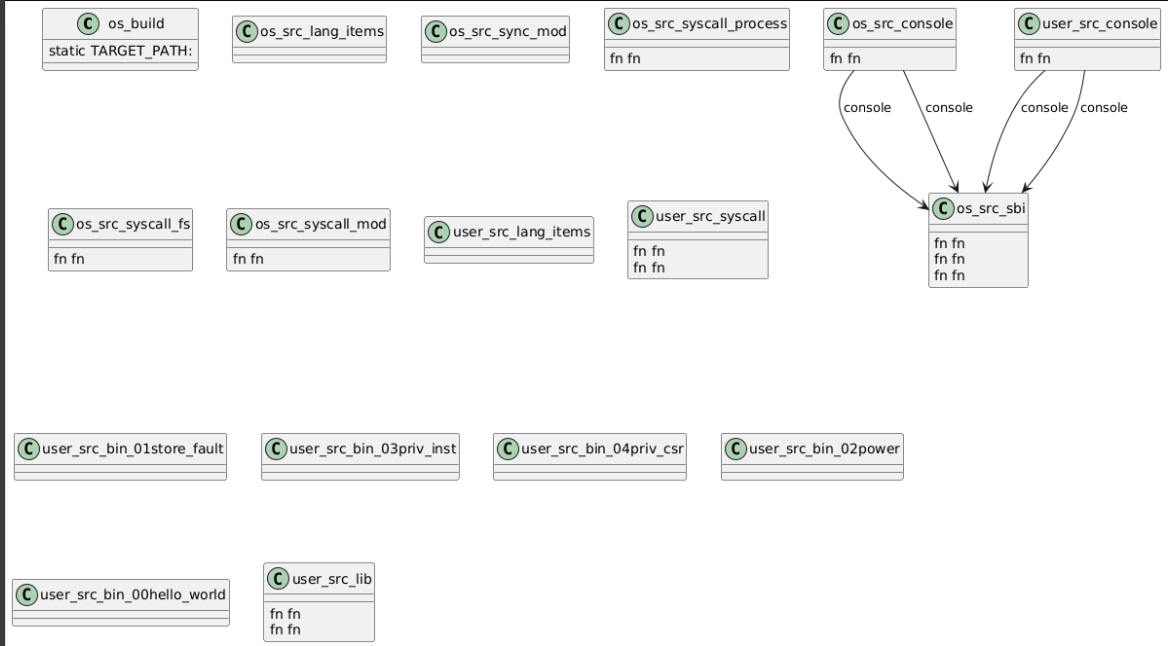
情况1，如果文件A.rs中的函数F1调用了文件B.rs中的函数F2，则在上述结果的末尾、@enduml之前追加A --> B: B.F2。

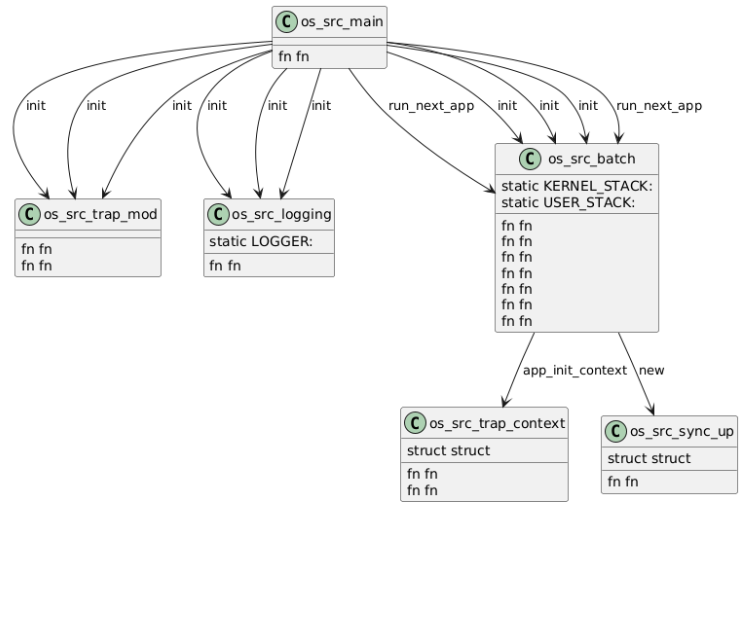
情况2，如果上述结果中的类C包含成员类D，则在上述结果的末尾、@enduml之前追加C \*-- D: define。

将上述结果保存为文件rCore-Class-ch2.txt。

说明：上述处理把每个源文件分别当作一个类，源文件名称及路径当作类名称，前面加上class，把变量当作类的属性，前面加上{field}，把函数当作类的方法，前面加上{method}，再在存在函数调用关系或者存在整体部分关系的类之间添加连线生成plantUML类图脚本文件。打开<https://plantuml.com/zh/sitemap>，点击“入门指南”，点击“[在线服务器](https://www.plantuml.com/plantuml)”，将生成的文件内容粘贴到窗口中，即可看到生成的UML类图，验证类图是否与rCore源程序结构一致。将结果保存为图片。如下是一个示例结果截图。完成该项目后，用自己的结果代替下面的示例截图。

点击“PNG”可保存生成的类图：生成的类图如下





1. 用Linux C语言编写程序实现如下功能：有3个生产者P1、P2、P3和4个消费者C1、C2、C3、C4访问包含6个缓冲区的环形缓冲池，每个生产者循环生产4次，每个消费者循环消费3次，规定：每个生产者和消费者各循环一次后才能进入下一轮循环，即每个进程或线程的速度不能比其它进程或线程的速度快很多。（25分）

//代码如下

#include <stdio.h>

#include <stdlib.h>

#include <pthread.h>

#include <semaphore.h>

#include <unistd.h>

#include <signal.h>

#define BUFFER\_SIZE 6

#define PRODUCER\_COUNT 3

#define CONSUMER\_COUNT 4

int buffer[BUFFER\_SIZE];

int in = 0, out = 0;

sem\_t empty, full;

pthread\_mutex\_t mutex;

volatile sig\_atomic\_t running = 1; // 控制运行状态

void\* producer(void\* arg) {

int id = \*(int\*)arg;

while (running) {

sem\_wait(&empty);

if (!running) break;

pthread\_mutex\_lock(&mutex);

buffer[in] = rand() % 1000;

printf("生产者 P%d 生产数据 %d 到缓冲区[%d]\n", id + 1, buffer[in], in);

in = (in + 1) % BUFFER\_SIZE;

pthread\_mutex\_unlock(&mutex);

sem\_post(&full);

usleep(200000); // 模拟延时

}

pthread\_exit(NULL);

}

void\* consumer(void\* arg) {

int id = \*(int\*)arg;

while (running) {

sem\_wait(&full);

if (!running) break;

pthread\_mutex\_lock(&mutex);

int data = buffer[out];

printf("消费者 C%d 消费数据 %d 从缓冲区[%d]\n", id + 1, data, out);

out = (out + 1) % BUFFER\_SIZE;

pthread\_mutex\_unlock(&mutex);

sem\_post(&empty);

usleep(300000); // 模拟延时

}

pthread\_exit(NULL);

}

int main() {

pthread\_t producers[PRODUCER\_COUNT], consumers[CONSUMER\_COUNT];

int i;

int producer\_ids[PRODUCER\_COUNT], consumer\_ids[CONSUMER\_COUNT];

sem\_init(&empty, 0, BUFFER\_SIZE);

sem\_init(&full, 0, 0);

pthread\_mutex\_init(&mutex, NULL);

for (i = 0; i < PRODUCER\_COUNT; i++) {

producer\_ids[i] = i;

pthread\_create(&producers[i], NULL, producer, &producer\_ids[i]);

}

for (i = 0; i < CONSUMER\_COUNT; i++) {

consumer\_ids[i] = i;

pthread\_create(&consumers[i], NULL, consumer, &consumer\_ids[i]);

}

// 主线程监听输入

printf("程序运行中，按回车键结束...\n");

getchar();

running = 0;

// 防止线程卡在sem\_wait

for (i = 0; i < PRODUCER\_COUNT; i++) sem\_post(&empty);

for (i = 0; i < CONSUMER\_COUNT; i++) sem\_post(&full);

for (i = 0; i < PRODUCER\_COUNT; i++) {

pthread\_join(producers[i], NULL);

}

for (i = 0; i < CONSUMER\_COUNT; i++) {

pthread\_join(consumers[i], NULL);

}

sem\_destroy(&empty);

sem\_destroy(&full);

pthread\_mutex\_destroy(&mutex);

printf("程序已退出。\n");

return 0;

}

//将以上保存为c文件，编译后在命令行中运行即可

//运行结果如下

程序运行中，按回车键结束...

生产者 P1 生产数据 383 到缓冲区[0]

生产者 P2 生产数据 886 到缓冲区[1]

消费者 C3 消费数据 383 从缓冲区[0]

消费者 C4 消费数据 886 从缓冲区[1]

生产者 P3 生产数据 777 到缓冲区[2]

消费者 C1 消费数据 777 从缓冲区[2]

生产者 P2 生产数据 915 到缓冲区[3]

生产者 P1 生产数据 793 到缓冲区[4]

消费者 C2 消费数据 915 从缓冲区[3]

生产者 P3 生产数据 335 到缓冲区[5]

消费者 C3 消费数据 793 从缓冲区[4]

消费者 C4 消费数据 335 从缓冲区[5]

生产者 P3 生产数据 386 到缓冲区[0]

生产者 P2 生产数据 492 到缓冲区[1]

生产者 P1 生产数据 649 到缓冲区[2]

消费者 C1 消费数据 386 从缓冲区[0]

消费者 C2 消费数据 492 从缓冲区[1]

消费者 C4 消费数据 649 从缓冲区[2]

生产者 P3 生产数据 421 到缓冲区[3]

消费者 C3 消费数据 421 从缓冲区[3]

生产者 P1 生产数据 362 到缓冲区[4]

生产者 P2 生产数据 27 到缓冲区[5]

消费者 C1 消费数据 362 从缓冲区[4]

生产者 P1 生产数据 690 到缓冲区[0]

生产者 P2 生产数据 59 到缓冲区[1]

生产者 P3 生产数据 763 到缓冲区[2]

消费者 C2 消费数据 27 从缓冲区[5]

消费者 C4 消费数据 690 从缓冲区[0]

消费者 C3 消费数据 59 从缓冲区[1]

生产者 P1 生产数据 926 到缓冲区[3]

生产者 P2 生产数据 540 到缓冲区[4]

生产者 P3 生产数据 426 到缓冲区[5]

消费者 C1 消费数据 763 从缓冲区[2]

消费者 C2 消费数据 926 从缓冲区[3]

消费者 C4 消费数据 540 从缓冲区[4]

消费者 C3 消费数据 426 从缓冲区[5]

生产者 P2 生产数据 172 到缓冲区[0]

生产者 P1 生产数据 736 到缓冲区[1]

生产者 P3 生产数据 211 到缓冲区[2]

消费者 C1 消费数据 172 从缓冲区[0]

生产者 P2 生产数据 368 到缓冲区[3]

生产者 P1 生产数据 567 到缓冲区[4]

消费者 C2 消费数据 736 从缓冲区[1]

生产者 P3 生产数据 429 到缓冲区[5]

消费者 C4 消费数据 211 从缓冲区[2]

消费者 C3 消费数据 368 从缓冲区[3]

生产者 P2 生产数据 782 到缓冲区[0]

生产者 P3 生产数据 530 到缓冲区[1]

生产者 P1 生产数据 862 到缓冲区[2]

消费者 C1 消费数据 567 从缓冲区[4]

消费者 C2 消费数据 429 从缓冲区[5]

消费者 C4 消费数据 782 从缓冲区[0]

消费者 C3 消费数据 530 从缓冲区[1]

生产者 P2 生产数据 123 到缓冲区[3]

生产者 P1 生产数据 67 到缓冲区[4]

生产者 P3 生产数据 135 到缓冲区[5]

消费者 C1 消费数据 862 从缓冲区[2]

生产者 P2 生产数据 929 到缓冲区[0]

生产者 P3 生产数据 802 到缓冲区[1]

消费者 C2 消费数据 123 从缓冲区[3]

生产者 P1 生产数据 22 到缓冲区[2]

消费者 C4 消费数据 67 从缓冲区[4]

消费者 C3 消费数据 135 从缓冲区[5]

消费者 C1 消费数据 929 从缓冲区[0]

生产者 P2 生产数据 58 到缓冲区[3]

生产者 P1 生产数据 69 到缓冲区[4]

生产者 P3 生产数据 167 到缓冲区[5]

程序已退出。

3、火车票余量是一种临界资源（共享变量），同时被许多购票进程访问。现在有100张车票可售。用Linux C语言编写套接字通信程序，模拟旅客群体通过自己的终端（客户端）访问售票服务器购票的过程。注意防止多名旅客买到同一张车票。多名旅客同时购票很常见，在有多张车票可售的情况下，让旅客排成一个队列顺序购票显然并不高效。在程序中考虑支持多个旅客同时访问不同的车票，同时完成购票的方案。（45分）

以下为server.c代码

#include <stdio.h>

#include <stdlib.h>

#include <unistd.h>

#include <string.h>

#include <pthread.h>

#include <netinet/in.h>

#include <arpa/inet.h>

#define PORT 8888

#define MAX\_TICKETS 100

int tickets[MAX\_TICKETS]; // 0 = 未售, 1 = 已售

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

void\* handle\_client(void\* arg) {

int client\_sock = \*(int\*)arg;

free(arg);

int ticket\_number = -1;

pthread\_mutex\_lock(&ticket\_mutex);

for (int i = 0; i < MAX\_TICKETS; i++) {

if (tickets[i] == 0) {

tickets[i] = 1;

ticket\_number = i + 1;

break;

}

}

pthread\_mutex\_unlock(&ticket\_mutex);

if (ticket\_number == -1) {

char\* msg = "票已售罄，请下次再来。\n";

send(client\_sock, msg, strlen(msg), 0);

} else {

char msg[64];

sprintf(msg, "购票成功，您购买的是第 %d 张票。\n", ticket\_number);

send(client\_sock, msg, strlen(msg), 0);

}

close(client\_sock);

return NULL;

}

int main() {

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

if (server\_sock < 0) {

perror("socket");

exit(1);

}

// 设置端口重用，避免“Address already in use”，第二次调试时遇到问题，在网上找到了解决方案

int opt = 1;

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

struct sockaddr\_in server\_addr = {

.sin\_family = AF\_INET,

.sin\_port = htons(PORT),

.sin\_addr.s\_addr = INADDR\_ANY,

};

if (bind(server\_sock, (struct sockaddr\*)&server\_addr, sizeof(server\_addr)) < 0) {

perror("bind");

exit(1);

}

listen(server\_sock, 20);

printf("售票服务器已启动，监听端口 %d...\n", PORT);

while (1) {

struct sockaddr\_in client\_addr;

socklen\_t len = sizeof(client\_addr);

int\* client\_sock = malloc(sizeof(int));

\*client\_sock = accept(server\_sock, (struct sockaddr\*)&client\_addr, &len);

if (\*client\_sock < 0) {

perror("accept");

continue;

}

pthread\_t tid;

pthread\_create(&tid, NULL, handle\_client, client\_sock);

pthread\_detach(tid);

}

close(server\_sock);

return 0;

}

以下为client.c代码

#include <stdio.h>

#include <stdlib.h>

#include <unistd.h>

#include <string.h>

#include <netinet/in.h>

#include <arpa/inet.h>

#define SERVER\_IP "127.0.0.1"

#define SERVER\_PORT 8888

int main() {

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

if (sock < 0) {

perror("socket");

exit(1);

}

struct sockaddr\_in server\_addr = {

.sin\_family = AF\_INET,

.sin\_port = htons(SERVER\_PORT),

.sin\_addr.s\_addr = inet\_addr(SERVER\_IP),

};

if (connect(sock, (struct sockaddr\*)&server\_addr, sizeof(server\_addr)) < 0) {

perror("connect");

exit(1);

}

char buffer[128];

int n = read(sock, buffer, sizeof(buffer) - 1);

if (n > 0) {

buffer[n] = '\0';

printf("服务器回复：%s", buffer);

}

close(sock);

return 0;

}

如何使用

编译

gcc -o server server.c -lpthread

gcc -o client client.c

运行

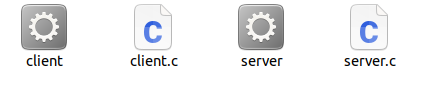
# 开一个终端运行服务器

./server

# 开多个终端模拟多个旅客购票

./client

编译，运行结果如下



//此为server端

oslab@oslab-virtual-machine:~/vvv$ ./server

售票服务器已启动，监听端口 8888...

//此为第一个client端

oslab@oslab-virtual-machine:~/vvv$ ./client

服务器回复：购票成功，您购买的是第 1 张票。

//此为第二个client端

oslab@oslab-virtual-machine:~/vvv$ ./client

服务器回复：购票成功，您购买的是第 2 张票。