

苏州大学实验报告

院、系	计算机学院	年级专业	19 计科图灵	姓名	张昊	学号	1927405160
课程名称	操作系统课程实践					成绩	
指导教师	李培峰	同组实验者	无	实验日期	2022 年 2 月 24 日		

实验名称 实验 3 Linux 进程通信与进程同步

一. 实验目的

1. 理解进程间通信的概念和方法。
2. 掌握常用的 Linux 进程间通信的方法。
3. 加强对进程同步和互斥的理解，学会使用信号量解决资源共享问题。
4. 熟悉 Linux 进程同步原语。
5. 掌握信号量 wait/signal 原语的使用方法，理解信号量的定义、赋初值及 wait/signal 操作。

二. 实验内容

1. （实验 4.1: 两个进程相互通信）

- (1) 编写 C 程序，使用 Linux 中的 IPC 机制完成“石头、剪刀、布”游戏。
- (2) 修改上述 C 程序，使之能够在网络上运行。

2. （实验 4.2: 进程同步实验）

编写 C 程序，使用 Linux 操作系统中的信号量机制模拟解决经典的进程同步问题：生产者-消费者问题。假设有一个生产者和一个消费者，缓冲区可以存放产品，生产者不断生产产品并存入缓冲区，消费者不断从缓冲区中取出产品并消费。

三. 操作方法和实验步骤

1. （实验 4.1: 两个进程相互通信）

实验中创建三个进程，一个为裁判进程，另外两个为玩家进程。裁判进程负责通知玩家进程当前的游戏轮数、上一轮的胜负结果与对手出招，并接受玩家进程出拳信息；玩家进程根据当前的游戏轮数以及上一轮的胜负结果出招，向裁判发送本轮出招。比赛采取多轮（如 100 轮）定胜负机制，并由裁判宣布最后结果。每次出招由裁判限定时间，超时则判负。

- (1) 设计表示“石头”“剪刀”“布”的数据结构以及它们之间的大小规则。

定义了如下的数据结构来表示玩家的出招和结果：

```
/* game.h */
typedef enum { // 动作
    ACT_NONE, // 无动作
    ACT_ROCK, // 石头
    ACT_PAPER, // 布
    ACT_SCISSORS // 剪刀
} action_t;

typedef enum { // 结果
    RES_P1_VICTORY, // 玩家 1 胜利（玩家 2 失败）
    RES_DRAW, // 平局
    RES_P1_DEFEAT // 玩家 2 胜利（玩家 1 失败）
} result_t;
```

关于动作的胜负关系为：石头 > 剪刀 > 布 > 石头；若有玩家超时，则其动作为“ACT_NONE”。定义判定胜负的逻辑为：

```
/* game.c */
// 出招结果的判断
result_t result_announce(action_t action1, action_t action2) {
    if (action1 != action2) {
        if (action1 == ACT_SCISSORS && action2 == ACT_ROCK) return RES_P1_DEFEAT;
        if (action1 == ACT_SCISSORS && action2 == ACT_PAPER) return RES_P1_VICTORY;
        if (action1 == ACT_ROCK && action2 == ACT_SCISSORS) return RES_P1_VICTORY;
        if (action1 == ACT_ROCK && action2 == ACT_PAPER) return RES_P1_DEFEAT;
        if (action1 == ACT_PAPER && action2 == ACT_SCISSORS) return RES_P1_DEFEAT;
        if (action1 == ACT_PAPER && action2 == ACT_ROCK) return RES_P1_VICTORY;
        if (action1 == ACT_NONE) return RES_P1_DEFEAT;
        if (action2 == ACT_NONE) return RES_P1_VICTORY;
    }
    return RES_DRAW;
}
```

- (2) 使用一个结构体来存放比赛结果，同时在裁判进程中使用一个数组来保存各轮结果。

```
/* game.h */
typedef struct { // 历史记录
    int round; // 轮数
    action_t action1, action2; // 玩家 1、2 的动作
    result_t result; // 结果
} round_logs_t;
```

- (3) 选择的 IPC 方法是 System V 消息队列。使用了四个消息队列，分别负责裁判进程为两个玩家进程发送出招消息和接收玩家进程的出招消息。在主函数中创建对应的 IPC 资源，并创建玩家进程和运行裁判进程（注意玩家进程的创建要早于裁判进程的运行，待裁判的函数结束后要等待两玩家进程结束）：

```
/* game-ipc.c */
int main(int argc, const char **argv) {
    if (argc != 2) {
        fprintf(stderr, "请提供一个命令行参数作为比赛轮数! \n");
        exit(EXIT_FAILURE);
    }
    // 初始化消息队列：分别是两个玩家的提醒消息和出招消息
    int key11 = 0x1f, key12 = 0x2f, key21 = 0x3f, key22 = 0x4f;
    int msg_notice1, msg_notice2, msg_rec1, msg_rec2;
    if ((msg_notice1 = msgget(key11, IPC_CREAT | 0666)) == -1) {
        fprintf(stderr, "提醒消息队列 1 创建失败! \n");
        exit(EXIT_FAILURE);
    }
    if ((msg_notice2 = msgget(key12, IPC_CREAT | 0666)) == -1) {
        fprintf(stderr, "提醒消息队列 2 创建失败! \n");
        exit(EXIT_FAILURE);
    }
}
```

```

if ((msg_rec1 = msgget(key21, IPC_CREAT | 0666)) == -1) {
    fprintf(stderr, "出招消息队列 1 创建失败! \n");
    exit(EXIT_FAILURE);
}
if ((msg_rec2 = msgget(key22, IPC_CREAT | 0666)) == -1) {
    fprintf(stderr, "出招消息队列 2 创建失败! \n");
    exit(EXIT_FAILURE);
}

const int rounds = atoi(argv[1]); // 比赛轮数
printf("比赛轮数: %d\n", rounds);
fflush(stdout);
// 先生成玩家, 再运行裁判
run_player(1, msg_notice1, msg_rec1);
run_player(2, msg_notice2, msg_rec2);
run_judgement(msg_notice1, msg_notice2, msg_rec1, msg_rec2, rounds);
// 等待玩家退出
wait(NULL);
wait(NULL);
// 删除消息队列
if (msgctl(msg_notice1, IPC_RMID, 0) == -1) {
    fprintf(stderr, "提醒消息队列 1 删除失败! \n");
}
if (msgctl(msg_notice2, IPC_RMID, 0) == -1) {
    fprintf(stderr, "提醒消息队列 2 删除失败! \n");
}
if (msgctl(msg_rec1, IPC_RMID, 0) == -1) {
    fprintf(stderr, "接收消息队列 1 删除失败! \n");
}
if (msgctl(msg_rec2, IPC_RMID, 0) == -1) {
    fprintf(stderr, "接收消息队列 2 删除失败! \n");
}
return 0;
}

```

- (4) 使用 fork 来创建玩家进程。首先设置随机数种子, 之后在循环中不断接收来自裁判的出招消息 (从 msg_notice 消息队列取消息), 根据裁判通知的轮数随机出招 (向 msg_snd 信息队列发送消息)。

裁判发给玩家的内容格式为结构体 notice_t, 使用结构体 notice_msg 封装为 msg_notice 消息队列支持的格式:

```

/* game.h */
typedef struct { // 裁判发给玩家的信息
    int round; // 当前是第几轮
    action_t last_rival_action; // 上一轮对手的动作
    result_t last_result; // 上一轮结果
} notice_t;
typedef struct { // 送往消息队列的封装

```

```
notice_t notice;
long type;
} notice_msg;
```

玩家发给裁判出招信息的内容格式为结构体 `move_t`，使用结构体 `move_msg` 封装为 `msg_snd` 消息队列支持的格式：

```
/* game.h */
typedef struct { // 玩家发给裁判的信息
    int uid; // 用户 ID
    int round; // 第几轮的动作，Socket 中轮数<0 为请求轮数
    action_t action; // 出招动作
} move_t;
typedef struct { // 送往消息队列的封装
    move_t move;
    long type;
} move_msg;
```

特别地，当裁判通知轮数为负数时结束进程。

```
/* game-ipc.c */
// 玩家
void run_player(int uid, int msg_notice, int msg_snd) {
    pid_t pid = fork(); // 创建新进程
    if (pid < 0) {
        fprintf(stderr, "Fork player 失败! \n");
        exit(EXIT_FAILURE);
    }
    if (pid == 0) {
        // 设置随机数种子
        unsigned seed = (unsigned) (time(NULL)+msg_notice)*(msg_snd+2);
        if (uid % 2) { seed = -seed; } // 尽量拉开两玩家的差距
        srand(seed);
        for (;;) {
            fflush(stdout); // 刷新标准输出缓冲区
            notice_msg notice_m; // 玩家接收来自裁判的出招消息
            notice_msg_init(&notice_m);
            while(msgrcv(msg_notice,&notice_m,sizeof(notice_t),0,0)==-1);
            if (notice_m.notice.round < 0) { // 如果轮数为<0，玩家进程会退出
                printf("玩家%d 进程结束\n", uid);
                exit(EXIT_SUCCESS);
            }
            printf("玩家%d 收到第%d 轮出招信号\n",
                uid, notice_m.notice.round);
            // 玩家出招
            move_msg move_m;
            move_msg_init(&move_m, uid);
            move_m.move.round = notice_m.notice.round;
            move_m.move.action=random_action(&notice_m.notice); //随机出招

```

```

        printf("玩家%d 发送第%d 轮出招 %s\n", uid, move_m.move.round,
               action_string(move_m.move.action));
        if (msgsnd(msg_snd, &move_m, sizeof(move_t), 0) == -1) {
            fprintf(stderr, "玩家%d 发送第%d 轮出招信息失败! \n",
                    uid, move_m.move.round);
        }
    }
}
}

```

- (5) 裁判进程在主进程中执行。裁判进程中建立保存有历史记录的数据。从第一轮开始逐轮迭代，每一轮通知各玩家当前是第几轮，并告知上一轮对手的动作和结果（向消息队列 msg_noticel, msg_notice2 发消息）；接收玩家的出招结果（从消息队列 msg_rec1, msg_rec2 接收）。待收到两个玩家的结果或超时后，判定并记录结果。

裁判进程在每一轮中设置一个开始时间，记录本轮第一次收到玩家出招的时间。裁判进程使用非阻塞的方式从消息队列接收出招消息，消息队列为空且未达到设置的超时间隔时裁判进程忙等。当没有选手出招时，本轮不会超时；只有当收到了选手出招消息后，才开始判断是否超时。

裁判进程在接收选手出招信息时验证消息中轮数是否和当前一致，不一致则丢弃。

当裁判进程迭代结束后，会依次通知玩家进程退出（设置当前轮数为负数），并进行游戏数据统计。

```

/* game-ipc.c */
// 裁判
void run_judgement(int msg_noticel, // 通知玩家 1 的消息队列 id
                  int msg_notice2, // 通知玩家 2 的消息队列 id
                  int msg_rec1, // 获取玩家 1 出招结果的消息队列 id
                  int msg_rec2, // 获取玩家 2 出招结果的消息队列 id
                  const int rounds) {
    round_logs_t *round_logs = // 历史记录
        malloc((rounds + 1) * sizeof(round_logs_t));
    round_logs[0].action1 = round_logs[0].action2 = ACT_NONE;
    round_logs[0].result = RES_DRAW;
    for (int round = 1; round <= rounds; ++round) {
        fflush(stdout); // 刷新标准输出缓冲区
        // 通知各玩家当前是第 round 轮，并告知上一轮对手的动作和结果
        notice_msg noticel, notice2;
        notice_msg_init(&noticel);
        notice_msg_init(&notice2);
        noticel.notice.round = notice2.notice.round = round;
        // 保存上轮结果、对手出招
        noticel.notice.last_result = notice2.notice.last_result = \
            round_logs[round - 1].result;
        noticel.notice.last_rival_action = round_logs[round-1].action2;
        notice2.notice.last_rival_action = round_logs[round-1].action1;
        printf("裁判通知各玩家当前是第%d 轮\n", round);
        while(msgsnd(msg_noticel, &noticel, sizeof(notice_t), 0) == -1);
    }
}

```

```

while(msgsnd(msg_notice2, &notice2, sizeof(notice_t), 0)==-1);
// 接收玩家这一轮的出招结果
move_msg move1, move2;
int get1 = 0, get2 = 0; // 是否成功获取玩家出招信息
time_t start_time = 0; // 开始时间
while (!get1 || !get2) { // 接收玩家这一轮的出招结果
    // 非阻塞地接收数据
    if (!get1 && msgrcv(msg_rec1, &move1, sizeof(move_t),
                        0, IPC_NOWAIT) != -1) {
        if (move1.move.round != round) { // 错误轮次
            printf("裁判收到玩家 1 错误轮次的出招: "
                   "当前是第%d 轮, 玩家 1 发送的是第%d 轮\n",
                   round, move1.move.round);
        } else {
            get1 = 1;
            printf("裁判收到玩家 1 第%d 轮出招 %s\n",
                   round, action_string(move1.move.action));
            if (start_time == 0) {
                start_time = time(NULL); // 设定开始时间
            }
        }
    }
    if (!get2 && msgrcv(msg_rec2, &move2, sizeof(move_t),
                        0, IPC_NOWAIT) != -1) {
        if (move2.move.round != round) { // 错误轮次
            printf("裁判收到玩家 2 错误轮次的出招: "
                   "当前是第%d 轮, 玩家 2 发送的是第%d 轮\n",
                   round, move2.move.round);
        } else {
            get2 = 1;
            printf("裁判收到玩家 2 第%d 轮出招 %s\n",
                   round, action_string(move2.move.action));
            if (start_time == 0) {
                start_time = time(NULL); // 记录第一个发送的时间
            }
        }
    }
    if (start_time != 0 && (time(NULL) - start_time > INTERVAL)) {
        break; // 如果超时停止接收
    }
}
if (!get1) { // 覆盖掉错误的出招结果
    move_msg_init(&move1, 1);
    move1.move.round = round;
}

```

```

    if (!get2) { // 覆盖掉错误的出招结果
        move_msg_init(&move2, 2);
        move2.move.round = round;
    }
    // 判定并记录结果, 进行下一轮
    round_logs[round].round = round;
    round_logs[round].action1 = move1.move.action;
    round_logs[round].action2 = move2.move.action;
    round_logs[round].result = // 判定结果
        result_announce(move1.move.action, move2.move.action);
    show_round(&round_logs[round]); // 显示结果
}
// 循环结束, 通知玩家游戏结束
notice_msg notice1, notice2;
notice_msg_init(&notice1);
notice_msg_init(&notice2);
// 如果轮数为-1, 玩家进程会退出
notice1.notice.round = notice2.notice.round = -1;
printf("裁判通知玩家游戏结束\n");
while(msgsnd(msg_notice1, &notice1, sizeof(notice_t), 0) == -1);
while(msgsnd(msg_notice2, &notice2, sizeof(notice_t), 0) == -1);
// 统计, 保存结果的文件为 result.txt
statistics(round_logs, rounds, "result.txt");
free(round_logs);
}

```

为了能使得程序在网络上运行, 使用 UDP socket 替换原有的 System V 消息队列以完成进程间的通信, 同时将裁判进程和玩家进程的源码分为两个文件, 分别编译为服务端程序(裁判)和客户端程序(玩家)。

同时, 对进程间相应的通信机制做出一些改动。玩家进程不再被动接受来自裁判进程的轮数通知, 而是主动询问裁判进程当前是游戏第几轮(复用发送出招消息的结构体 `move_t`, 将当前轮数 `round` 设置为负数来实现询问当前轮数)。玩家接收裁判消息修改为非阻塞的; 为防止短时间发送大量询问请求, 若得不到裁判进程的回复或回复的轮数没有变化, 则进程休眠一段时间(根据重试次数调整, 数量级为几百毫秒, 且不超过超时间隔)。

修改后的两文件如下:

```

/* game-socket-server.c */
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <time.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include "game.h"
// 裁判
void run_judgement(int fd, const int rounds) {

```

```

// server socket 配置
struct sockaddr_in peer_addr;
socklen_t peer_size;
ssize_t recv_num, send_num; // 发送和接收的字节数
char recv_buff[BUFFER_SIZE]; // 接收缓冲区
// 历史记录
round_logs_t *round_logs =
    malloc((rounds + 1) * sizeof(round_logs_t));
round_logs[0].action1 = round_logs[0].action2 = ACT_NONE;
round_logs[0].result = RES_DRAW;
round_logs[1].action1 = round_logs[1].action2 = ACT_NONE;
int round = 1; // 当前游戏轮数
int client1_bye = 0, client2_bye = 0; // 是否通知玩家进程退出
int get1 = 0, get2 = 0; // 是否成功获取玩家出招信息
time_t start_time = 0; // 开始时间
for (;;) {
    fflush(stdout); // 刷新标准输出缓冲区
    peer_size = sizeof(peer_addr);
    bzero(recv_buff, sizeof(recv_buff)); // 清空接收缓冲区
    do { // 非阻塞地接收数据
        recv_num = recvfrom(fd, recv_buff, sizeof(recv_buff),
            MSG_DONTWAIT, (struct sockaddr *)&peer_addr,
            &peer_size);
        // printf("裁判收到%d字节数据: %s\n", recv_num, recv_buff);
    } while (recv_num <= 0 // 忙等, 直到成功接收或超时
        || (start_time != 0 && (time(NULL) - start_time > INTERVAL)));
    if (recv_num > 0) { // 处理数据
        move_t *move = (move_t *) recv_buff; // client 发送的是 move_t 结构体
        if (move->uid != 1 && move->uid != 2) { // 只认定 uid=1 或 2 的玩家
            printf("未知的 uid %d, 忽略\n", move->uid);
            continue;
        }
        if (move->round < 0) { // round<0, client 请求轮数信息
            notice_t notice; // 要发送的数据
            if (round <= rounds) {
                // 通知该玩家当前是第 round 轮, 并告知上一轮对手的动作和结果
                // printf("裁判通知玩家%d 当前是第%d 轮\n", move->uid, round);
                notice.round = round;
                // 保存上轮结果、对手出招
                notice.last_result = round_logs[round - 1].result;
                notice.last_rival_action = (move->uid == 1) ?
                    round_logs[round - 1].action2 :
                    round_logs[round - 1].action1;
            } else { // 游戏已经结束, 通知玩家进程退出
                printf("裁判通知玩家%d 退出\n", move->uid);
            }
        }
    }
}

```



```

        notice.round = -1; // 如果返回的轮数<0, 玩家进程会退出
        if (move->uid == 1) client1_bye = 1; // 已通知玩家进程退出
        else client2_bye = 1;
    }
    send_num = sendto(fd, &notice, sizeof(notice_t), 0,
                      (struct sockaddr *) &peer_addr, peer_size);
    // printf("裁判发送%d字节数据\n", send_num);
} else { // round>0, 接收玩家这一轮的出招结果
    if (move->round != round) { // 错误轮次
        printf("裁判收到玩家%d 错误轮次的出招: "
               "当前是第%d 轮, 玩家发送的是第%d 轮\n",
               move->uid, round, move->round);
    } else {
        if (move->uid == 1) {
            get1 = 1;
            round_logs[round].action1 = move->action;
        } else if (move->uid == 2) {
            get2 = 1;
            round_logs[round].action2 = move->action;
        }
        printf("裁判收到玩家%d 第%d 轮出招 %s\n", move->uid, round,
               action_string(move->action));
        if (start_time == 0) {
            start_time = time(NULL); // 记录第一个发送的时间
        }
    }
}
}

if (client1_bye && client2_bye) {
    break; // 已经通知所有玩家进程退出, 裁判可以停止接收数据了
}

// 收到了两个玩家出招结果, 或者超时, 判定结果, 进行下一轮
if ((get1 && get2)
    || (start_time != 0 && (time(NULL) - start_time > INTERVAL))) {
    round_logs[round].round = round;
    round_logs[round].result = // 判定结果
        result_announce(round_logs[round].action1, round_logs[round].action2);
    show_round(&round_logs[round]); // 显示结果
    round++; // 进行下一轮
    get1 = get2 = 0;
    start_time = 0;
    if (round <= rounds) {
        printf("裁判: 当前是第%d 轮\n", round);
        round_logs[round].action1 = ACT_NONE;
        round_logs[round].action2 = ACT_NONE;
    }
}

```

```

    }
}
}
// 统计, 保存结果的文件为 result-server.txt
statistics(round_logs, rounds, "result-server.txt");
free(round_logs);
}
int main(int argc, const char **argv) {
    if (argc != 2) {
        fprintf(stderr, "请提供一个命令行参数作为比赛轮数! \n");
        exit(EXIT_FAILURE);
    }
    // 创建 UDP Socket
    int fd = socket(AF_INET, SOCK_DGRAM, 0);
    if (fd < 0) {
        fprintf(stderr, "socket error\n");
        exit(EXIT_FAILURE);
    }
    struct sockaddr_in server_addr;
    bzero(&server_addr, sizeof(server_addr));
    server_addr.sin_family = AF_INET;
    server_addr.sin_port = htons(PORT); // 9000
    server_addr.sin_addr.s_addr = htonl(INADDR_ANY); // 0.0.0.0
    // 绑定监听端口
    printf("裁判: 监听 %d 端口\n", PORT);
    if(bind(fd, (struct sockaddr*)&server_addr, sizeof(server_addr)) < 0) {
        fprintf(stderr, "bind error\n");
        exit(EXIT_FAILURE);
    }
    const int rounds = atoi(argv[1]); // 比赛轮数
    printf("比赛轮数: %d\n", rounds);
    run_judgement(fd, rounds); // 运行裁判进程
    // 关闭 socket
    close(fd);
    return 0;
}

/* game-socket-client.c */
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <time.h>
#include <sys/socket.h>
#include <netinet/in.h>

```

```

#include "game.h"
// 玩家
void run_player(int uid, int fd) {
    // client socket 配置
    struct sockaddr_in server_addr;
    bzero(&server_addr, sizeof(server_addr));
    server_addr.sin_family = AF_INET;
    server_addr.sin_port = htons(PORT); // 9000
    server_addr.sin_addr.s_addr = htonl(INADDR_LOOPBACK); // 127.0.0.1
    ssize_t send_num, recv_num; // 发送和接收的字节数
    char recv_buff[BUFFER_SIZE]; // 接收缓冲区
    // 设置随机数种子
    unsigned seed = (unsigned) (time(NULL) + fd) * (uid + 2);
    if (uid % 2) { seed = -seed; } // 尽量拉开两玩家的差距
    srand(seed);
    int round; // 记录游戏轮数
    for (;;) {
        fflush(stdout); // 刷新标准输出缓冲区
        bzero(recv_buff, sizeof(recv_buff)); // 清空接收缓冲区
        move_t move; // 向 server 发送的数据
        move_init(&move, uid); // round = -1, 询问裁判当前轮数
        int retry = 0; // 未收到数据重发次数
        do { // 玩家询问裁判当前轮数
            send_num = sendto(fd, &move, sizeof(move_t), 0,
                              (struct sockaddr *) &server_addr,
                              sizeof(server_addr));
            // printf("玩家%d 询问裁判当前轮数, 发送%d 字节\n", uid, send_num);
            recv_num = recvfrom(fd, recv_buff, sizeof(recv_buff),
                                MSG_DONTWAIT, NULL, NULL);
            // printf("玩家%d 收到%d 字节\n", uid, recv_num);
            if (recv_num <= 0) { // 未收到数据重发请求, 中间暂时休眠
                usleep((100 + retry * 100) % (INTERVAL * 1000));
                retry++;
            } else {
                retry = 0; // 重置重发次数
            }
        } while (send_num <= 0 || recv_num <= 0); // 直到成功发送或接收
        // server 返回内容为 notice_t 结构体
        notice_t *notice = (notice_t *) recv_buff; // server 发的是 notice_t
        if (notice->round < 0) { // 如果轮数<0, 玩家进程会退出
            printf("玩家%d 进程即将结束\n", uid);
            break;
        }
        if (round != notice->round) { // 轮数更新, 玩家出招
            printf("玩家%d 收到第%d 轮出招信号\n", uid, notice->round);
        }
    }
}

```

```

        round = move.round = notice->round; // 更新游戏轮数
        move.action = random_action(notice); // 随机出招
        send_num = sendto(fd, &move, sizeof(move_t), 0,
                           (struct sockaddr *) &server_addr,
                           sizeof(server_addr));
        printf("玩家%d 发送第%d 轮出招 %s, 发送%d 字节\n", uid, move.round,
               action_string(move.action), send_num);
    } else { // 出招轮数不变, 展示休眠
        // printf("玩家%d: 出招轮数不变\n", uid);
        usleep(300);
    }
}
}

int main(int argc, const char **argv) {
    if (argc != 2) {
        fprintf(stderr, "请提供一个命令行参数作为玩家 ID! \n");
        exit(EXIT_FAILURE);
    }
    // UDP Socket
    int fd = socket(PF_INET, SOCK_DGRAM, 0);
    if (fd < 0) {
        fprintf(stderr, "socket error\n");
        exit(EXIT_FAILURE);
    }
    int uid = atoi(argv[1]); // 玩家 ID
    run_player(uid, fd); // 运行玩家进程
    // 关闭 socket
    close(fd);
    return 0;
}

```

2. (实验 4.2: 进程同步实验)

为了模拟解决生产者-消费者问题, 需要创建两个并发执行的进程, 即生产者进程和消费者进程, 并且要让这两个进程共享同一个缓冲区。简单起见, 使用线程来模拟进程的同步。代码中使用两个线程来模拟生产者和消费者, 并且使用 `pthread` 库提供的线程操作, 使用 POSIX 信号量机制实现线程的同步。设置缓冲区大小由宏 `SIZE` 确定, 总共的产品数量由宏 `COUNT` 确定, 生产者不断生产 `COUNT` 个产品, 消费者不断消费 `COUNT` 个产品。

```

/* sync.c */
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <semaphore.h>
#include <pthread.h>
#define SIZE 5 // 缓冲区大小
#define COUNT 10 // 总共生产的产品数量

```

```

typedef struct { // 产品
    int id;      // 编号
    int value;   // 属性
} product_t;

product_t *buffer[SIZE]; // 保存产品的缓冲区
int producer_ptr = 0, consumer_ptr = 0; // 生产指针和消费指针
// 生产指针指向缓冲区中下一个产品的保存位置，消费指针指向缓冲区中下一个取出的位置

sem_t empty; //定义同步信号量 empty: 空缓冲区数量
sem_t full;  //定义同步信号量 full: 满缓冲区数量
sem_t mutex; //定义互斥信号量 mutex
void semaphore_init(); // 初始化信号量
void semaphore_destroy(); // 删除信号量

void *producer() { //生产者
    for (int i = 0; i < COUNT; ++i) {
        sem_wait(&empty); // P(empty)
        sem_wait(&mutex); // P(mutex)
        // 生产产品
        product_t *product = malloc(sizeof(product_t));
        if (product == NULL) {
            fprintf(stderr, "生产产品失败\n");
            semaphore_destroy();
            exit(EXIT_FAILURE); // 进程退出
        } else {
            product->id = i;
            product->value = rand();
        }
        buffer[producer_ptr] = product;
        producer_ptr = (producer_ptr + 1) % SIZE;
        printf("生产产品 (id=%d, 属性: %d) \n", product->id, product->value);
        sem_post(&mutex); // V(mutex)
        sem_post(&full);  // V(full)
    }
    return NULL;
}

void *consumer() { //消费者
    for (int i = 0; i < COUNT; ++i) {
        sem_wait(&full); // P(full)
        sem_wait(&mutex); // P(mutex)
        // 从缓冲区中取出产品
        product_t *product = buffer[consumer_ptr];
        printf("消费产品 (id=%d, 属性: %d) \n", product->id, product->value);
        free(product);
    }
}

```

```

    buffer[consumer_ptr] = NULL;
    consumer_ptr = (consumer_ptr + 1) % SIZE;
    sem_post(&mutex); // V(mutex)
    sem_post(&empty); // V(empty)
}
return NULL;
}

int main() {
    srand(time(NULL));
    semaphore_init();
    pthread_t id_producer;
    pthread_t id_consumer;

    pthread_create(&id_producer, NULL, producer, NULL); //创建生产者线程
    pthread_create(&id_consumer, NULL, consumer, NULL); //创建消费者线程
    pthread_join(id_producer, NULL); //等待生产者线程结束
    pthread_join(id_consumer, NULL); //等待消费者线程结束
    semaphore_destroy();
    return 0;
}

void semaphore_init() {
    sem_init(&empty, 0, SIZE); // empty=SIZE
    sem_init(&full, 0, 0); // full=0
    sem_init(&mutex, 0, 1); // mutex=1
}

void semaphore_destroy() {
    sem_destroy(&empty);
    sem_destroy(&full);
    sem_destroy(&mutex);
}

```

四. 实验结果和分析

提供了 Makefile 文件，使用 make 工具编译本实验所有代码。

```

holger-405160@hao-zhang:~/codes/exp03$ ls
game.c  game-ipc.c          game-socket-server.c  sync.c
game.h  game-socket-client.c  Makefile
holger-405160@hao-zhang:~/codes/exp03$ make
gcc game.c game.h -c
gcc game.h game-ipc.c -c
gcc game.o game-ipc.o -o game
gcc game.h game-socket-server.c -c
gcc game.o game-socket-server.o -o game-server
gcc game.h game-socket-client.c -c
gcc game.o game-socket-client.o -o game-client
gcc sync.c -lpthread -c
gcc sync.o -lpthread -o sync
holger-405160@hao-zhang:~/codes/exp03$ ls
game          game.h.gch  game-server  game-socket-server.o  sync.o
game.c        game-ipc.c  game-socket-client.c  Makefile
game-client  game-ipc.o  game-socket-client.o  sync
game.h        game.o      game-socket-server.c  sync.c
holger-405160@hao-zhang:~/codes/exp03$

```

1. （实验 4.1: 两个进程相互通信）

运行可执行文件 game，提供一个命令行参数 20 作为游戏的轮数：

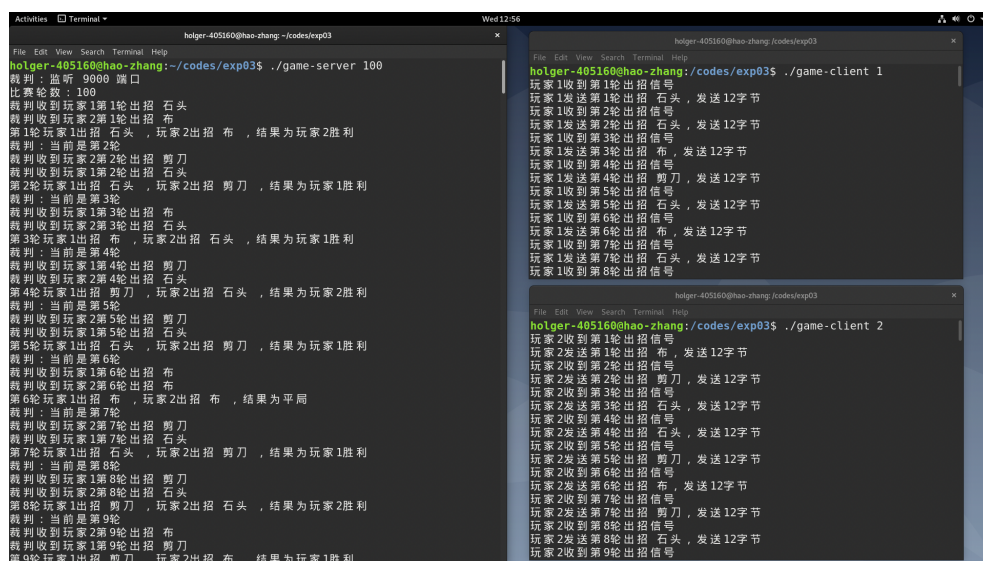
```
holger-405160@hao-zhang:~/codes/exp03$ ./game 20
比赛轮数：20
裁判通知各玩家当前是第1轮
玩家1收到第1轮出招信号
玩家1发送第1轮出招 石头
裁判收到玩家1第1轮出招 石头
玩家2收到第1轮出招信号
玩家2发送第1轮出招 石头
裁判收到玩家2第1轮出招 石头
第1轮玩家1出招 石头，玩家2出招 石头，结果为平局
裁判通知各玩家当前是第2轮
玩家1收到第2轮出招信号
玩家1发送第2轮出招 石头
玩家2收到第2轮出招信号
玩家2发送第2轮出招 布
裁判收到玩家2第2轮出招 布
裁判收到玩家1第2轮出招 石头
第2轮玩家1出招 石头，玩家2出招 布，结果为玩家2胜利
裁判通知各玩家当前是第3轮
玩家1收到第3轮出招信号
玩家1发送第3轮出招 剪刀
玩家2收到第3轮出招信号
玩家2发送第3轮出招 布
裁判收到玩家2第3轮出招 布
裁判收到玩家1第3轮出招 剪刀
第3轮玩家1出招 剪刀，玩家2出招 布，结果为玩家1胜利
```

result.txt 文件如下：

```
holger-405160@hao-zhang:~/codes/exp03$ cat result.txt
NO.1: 平局
NO.2: 玩家2胜利
NO.3: 玩家1胜利
NO.4: 玩家1胜利
NO.5: 玩家2胜利
NO.6: 玩家2胜利
NO.7: 玩家1胜利
NO.8: 平局
NO.9: 平局
NO.10: 玩家1胜利
NO.11: 平局
NO.12: 玩家2胜利
NO.13: 玩家2胜利
NO.14: 平局
NO.15: 玩家2胜利
NO.16: 玩家2胜利
NO.17: 玩家1胜利
NO.18: 玩家1胜利
NO.19: 玩家1胜利
NO.20: 玩家2胜利

玩家1胜利次数：7
玩家2胜利次数：8
平局次数：5
最终胜利：玩家2
holger-405160@hao-zhang:~/codes/exp03$
```

分别运行可执行文件 game-server(提供一个命令行参数 100 作为游戏的轮数),可执行文件 game-client（运行两次，分别提供命令行参数 1、2 作为玩家 ID）：



```
holger-405160@hao-zhang:~/codes/exp03$ ./game-server 100
裁判：监听 9000 端口
比赛轮数：100
裁判收到玩家1第1轮出招 石头
裁判收到玩家2第1轮出招 布
第1轮玩家1出招 石头，玩家2出招 布，结果为玩家2胜利
裁判：当前是第2轮
裁判收到玩家2第2轮出招 剪刀
裁判收到玩家1第2轮出招 石头
第2轮玩家1出招 石头，玩家2出招 剪刀，结果为玩家1胜利
裁判：当前是第3轮
裁判收到玩家1第3轮出招 布
裁判收到玩家2第3轮出招 石头
第3轮玩家1出招 布，玩家2出招 石头，结果为玩家1胜利
裁判：当前是第4轮
裁判收到玩家1第4轮出招 剪刀
裁判收到玩家2第4轮出招 石头
第4轮玩家1出招 剪刀，玩家2出招 石头，结果为玩家2胜利
裁判：当前是第5轮
裁判收到玩家2第5轮出招 剪刀
裁判收到玩家1第5轮出招 石头
第5轮玩家1出招 石头，玩家2出招 剪刀，结果为玩家1胜利
裁判：当前是第6轮
裁判收到玩家1第6轮出招 布
裁判收到玩家2第6轮出招 布
第6轮玩家1出招 布，玩家2出招 布，结果为平局
裁判：当前是第7轮
裁判收到玩家2第7轮出招 剪刀
裁判收到玩家1第7轮出招 石头
第7轮玩家1出招 石头，玩家2出招 剪刀，结果为玩家1胜利
裁判：当前是第8轮
裁判收到玩家1第8轮出招 剪刀
裁判收到玩家2第8轮出招 石头
第8轮玩家1出招 剪刀，玩家2出招 石头，结果为玩家2胜利
裁判：当前是第9轮
裁判收到玩家2第9轮出招 布
裁判收到玩家1第9轮出招 剪刀
第9轮玩家1出招 剪刀，玩家2出招 布，结果为玩家1胜利
```

```
holger-405160@hao-zhang:~/codes/exp03$ ./game-client 1
玩家1收到第1轮出招信号
玩家1发送第1轮出招 石头，发送12字节
玩家1收到第2轮出招信号
玩家1发送第2轮出招 石头，发送12字节
玩家1收到第3轮出招信号
玩家1发送第3轮出招 布，发送12字节
玩家1收到第4轮出招信号
玩家1发送第4轮出招 剪刀，发送12字节
玩家1收到第5轮出招信号
玩家1发送第5轮出招 石头，发送12字节
玩家1收到第6轮出招信号
玩家1发送第6轮出招 布，发送12字节
玩家1收到第7轮出招信号
玩家1发送第7轮出招 石头，发送12字节
玩家1收到第8轮出招信号
```

```
holger-405160@hao-zhang:~/codes/exp03$ ./game-client 2
玩家2收到第1轮出招信号
玩家2发送第1轮出招 布，发送12字节
玩家2收到第2轮出招信号
玩家2发送第2轮出招 剪刀，发送12字节
玩家2收到第3轮出招信号
玩家2发送第3轮出招 石头，发送12字节
玩家2收到第4轮出招信号
玩家2发送第4轮出招 石头，发送12字节
玩家2收到第5轮出招信号
玩家2发送第5轮出招 剪刀，发送12字节
玩家2收到第6轮出招信号
玩家2发送第6轮出招 布，发送12字节
玩家2收到第7轮出招信号
玩家2发送第7轮出招 剪刀，发送12字节
玩家2收到第8轮出招信号
玩家2发送第8轮出招 石头，发送12字节
玩家2收到第9轮出招信号
```

result-server.txt 文件如下:

```
holger-405160@hao-zhang:~/codes/exp03$ tail -n 20 result-server.txt
NO.86: 玩家1胜利
NO.87: 平局
NO.88: 平局
NO.89: 玩家1胜利
NO.90: 平局
NO.91: 玩家1胜利
NO.92: 玩家1胜利
NO.93: 玩家1胜利
NO.94: 玩家1胜利
NO.95: 玩家1胜利
NO.96: 玩家1胜利
NO.97: 平局
NO.98: 平局
NO.99: 玩家1胜利
NO.100: 平局

玩家1胜利次数: 29
玩家2胜利次数: 29
平局次数: 42
最终胜利: 平局
holger-405160@hao-zhang:~/codes/exp03$
```

2. (实验 4.2: 进程同步实验)

该实验代码编译时要特别注意添加 `-lpthread` 选项。

设置缓冲区大小为 5, 总共生产的产品数量为 10, 运行结果:

```
holger-405160@hao-zhang:~/codes/exp03$ ./sync
生产产品 (id=0, 属性: 2017566342)
生产产品 (id=1, 属性: 1683775475)
生产产品 (id=2, 属性: 1051093923)
生产产品 (id=3, 属性: 1933121522)
生产产品 (id=4, 属性: 286937760)
消费产品 (id=0, 属性: 2017566342)
消费产品 (id=1, 属性: 1683775475)
消费产品 (id=2, 属性: 1051093923)
消费产品 (id=3, 属性: 1933121522)
消费产品 (id=4, 属性: 286937760)
生产产品 (id=5, 属性: 357490684)
生产产品 (id=6, 属性: 190569857)
生产产品 (id=7, 属性: 635324392)
生产产品 (id=8, 属性: 1763867502)
生产产品 (id=9, 属性: 576005822)
消费产品 (id=5, 属性: 357490684)
消费产品 (id=6, 属性: 190569857)
消费产品 (id=7, 属性: 635324392)
消费产品 (id=8, 属性: 1763867502)
消费产品 (id=9, 属性: 576005822)
```

另一种可能的运行结果:

```
holger-405160@hao-zhang:~/codes/exp03$ ./sync
生产产品 (id=0, 属性: 1725925041)
生产产品 (id=1, 属性: 1506176354)
生产产品 (id=2, 属性: 638135532)
生产产品 (id=3, 属性: 488010073)
生产产品 (id=4, 属性: 156410177)
消费产品 (id=0, 属性: 1725925041)
生产产品 (id=5, 属性: 844281058)
消费产品 (id=1, 属性: 1506176354)
生产产品 (id=6, 属性: 2057420330)
消费产品 (id=2, 属性: 638135532)
生产产品 (id=7, 属性: 208856979)
消费产品 (id=3, 属性: 488010073)
生产产品 (id=8, 属性: 1360770368)
消费产品 (id=4, 属性: 156410177)
生产产品 (id=9, 属性: 779020279)
消费产品 (id=5, 属性: 844281058)
消费产品 (id=6, 属性: 2057420330)
消费产品 (id=7, 属性: 208856979)
消费产品 (id=8, 属性: 1360770368)
消费产品 (id=9, 属性: 779020279)
```

五. 讨论、心得

1. 本次实验的主要内容是进程的通信与同步。通过本次实验, 我进一步加深了对进程间通信的认

识，加强了进程同步和互斥的理解，熟悉并掌握了 Linux 进程间通信的方法以及同步原语，以及 socket 编程。

2. 第一个实验的代码中随机数的取值对于模拟“石头、剪刀、布”游戏很重要，如果取值不当（例如所有的基于 `time(NULL)` 的随机数种子都在 1 秒内产生），就可能出现大量平局的情况。因此，可以根据玩家 ID 以及消息队列 ID 或套接字文件描述符等额外信息对 `time(NULL)` 的结果进行处理（加减、数乘、溢出等），使得即使在同一时刻不同进程产生的随机数种子也不相同。
3. 比较 Linux 操作系统中的几种 IPC 机制，并说明它们各自适用于哪些场合。
 - a) 管道：无名管道是一种半双工的通信方式，数据只能单向流动，而且只能在具有亲缘关系（父子进程关系）的进程间使用；有名管道可以提供给任意关系的进程使用，长期存在于系统中。
 - b) 消息队列是消息的链表，存放在内核中并由消息队列标识符标识。消息队列克服了信号传递信息少、管道只能承载无格式字节流以及缓冲区大小受限等缺点，不再局限于父子进程，而允许任意进程通过共享消息队列来实现进程间通信。使用方便，不需要考虑同步问题，但是更消耗资源，不适用于操作频繁、信息量大的场合。
 - c) 共享内存是映射一段能被其他进程所访问的内存，这段共享内存由一个进程创建，但多个进程都可以访问。共享内存是最快的 IPC 方式，它不需要复制，是针对其他进程间通信方式运行效率低而专门设计的。它往往与其他通信机制（如信号量）配合使用，来实现进程间的同步和通信。由于内存实体存在于计算机系统中，所以只能由处于同一个计算机系统内的诸进程共享，不方便网络通信。
 - d) 信号量是用来解决进程间同步与互斥问题的一种进程间通信机制。程序对信号量的访问都是原子操作，且只允许对信号量进行 P 操作(`wait`)和 V 操作(`signal`)。信号量最主要的应用是实现共享内存方式的进程间通信。
 - e) 套接字(`socket`)也是一种进程间通信方式，但并不局限于同一台计算机中的进程。
4. 多线程并发与多进程并发有何不同与相同之处？
 - a) 如果处理机是单核的或不支持线程级的调度，多线程并发且线程属于同一个进程，进程的执行效率可以提高；但如果线程分属于不同进程，则退化为多进程之间的并发。
 - b) 如果处理机支持多线程调度，多进程并发就是多线程并发，而多线程并发不一定是多进程并发。
 - c) 如果处理机是多核的支持多线程调度，多线程还可能并行执行。