# 山东大学 计算机科学与技术 学院

## 操作系统 课程实验报告

| 学号:201800130005 | 姓名：张畅 | 班级：2018 计科 3 班 |
|---|---|---|
| 实验题目：死锁问题实验 | | |
| 实验学时：2 | 实验日期：2020.12.4 | |

实验目的：

通过本实验观察死锁产生的现象，考察觉接思索问题的方法。从而进一步加深对于死锁问题的理解。掌握解决死锁问题的几种算法和编程及调试技术。练习怎样构造管程来较好地避免死锁和饥饿问题的产生。

实验过程中遇到和解决的问题：

（记录实验过程中遇到的问题，以及解决过程和实验结果。可以适当配以关键代码辅助说明，但不要大段贴代码。）

本实验的程序设计围绕着单向车道的双向通行问题，并且有三个要求：需要考虑车道上同时允许的最大行驶车辆数目、撞车问题和长时间等待(饥饿)问题。利用共享内存和信号量机制保证了要求一，利用锁机制保证了要求二，但是饥饿问题无法像前两者一样依靠简单的调用机制来解决，为了解决饥饿问题，对代码经过了多次的修改，并且针对饥饿问题的解决与助教进行了交流，最终实现了较好的效果。

在程序设计之初，将防止饥饿的机制设计成：如果与当前行驶向相逆的方向存在大于同向行车数的等待车辆，当前方向的车辆不再允许进入行驶道路，而是阻塞直至行驶的车辆为零，将路线让给对向行驶。

但是经过测试之后，发现存在一定的问题，由于车辆产生的速率远小于车辆过桥的速度，很容易出现等待一侧队列过长导致长期等待的问题。

思考之后，将阈值设置为同向行车数的一半，经过测试之后，明显提高了系统处理的效率，系统能够较好地均衡两侧车辆的负载，保持比较高的并行效率。

与助教进行讨论之后，发现这种设计并不完美，仍然存在饿死的可能。一旦等待侧的车辆过少，始终小于同向行车数的一半，就会导致行车侧的车辆源源不断进入，最终饿死等待侧。

经过考虑和测试，选择了加入时间概念，每 10 秒为一个周期，定期检测是否存在长期饥饿的进程(车辆)，如果存在则停止形式侧驶入车辆，转向等待侧通行。改进后的系统进行测试的过程中，效率明显较高，且始终没有出现饥饿的问题，保证了系统设计的正确性和高效性。

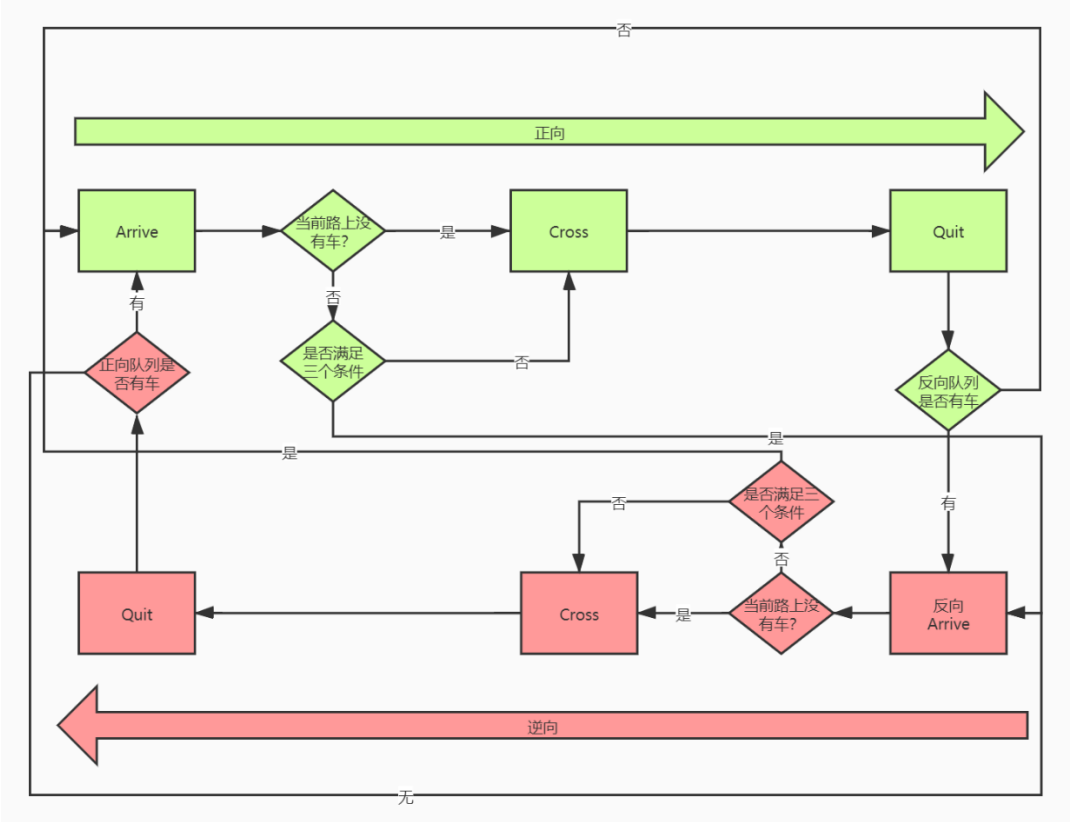
实验步骤与内容：

1、通过实例程序掌握锁的用法

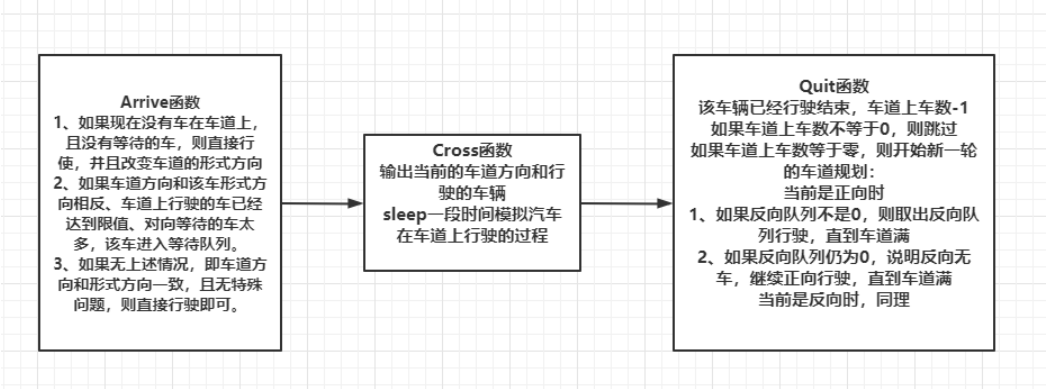
本实验的核心是围绕着程序运行中的锁问题与管程，两者都是在解决同步互斥问题中重要的元素。在课程上了解了锁的基本原理和管程的大致实现，但是并没有进行实际的设计。通过阅读示例程序，结合前面两个实验中有关共享内存、信号量和消息队列的知识，基本理解了锁的用法。同时学习了示例程序中的管程写法，利用锁和信号量结构来完成数据的保护和修改，

注意点：在管程的类方法中，如果需要修改共享的数据，在修改之前需要执行

上锁，在修改完成之后需要执行解开锁，只有这样才能保证数据的安全性。
2、设计实验的基本逻辑结构
本实验要解决的问题是一个单向车道的通行问题，需要考虑车道上同时允许的最大行驶车辆数目、撞车问题和长时间等待(饥饿)问题。
经过思考和设计，基本确定了分别解决上述问题的一些途径：
最大的行驶车辆数目问题可以使用共享内存解决，在车辆驶入之前，查看系统内的车辆数目，如果已经达到上限，则需要进入等待队列，直到当前系统内的车辆驶出一部分之后再驶入。
撞车问题可以使用锁机制较好的解决，确定下当前的行驶方向之后，后面保证每次修改形式方向时，都使用锁机制进行保护，即可保证不会撞车。
长时间等待问题即饥饿是一个在解决进程问题时常常出现的问题，一般无法采用某种特定的数据结构解决。本系统中采用限定车辆进入辅以强制转换行驶方向转换的机制。具体即：在道路上有车辆行驶时，如果逆方向没有车辆等待，则新来的同方向车辆可以加入到道路上；否则新来的同方向车辆必须加入阻塞进入等待状态。根据以上的解决逻辑，可以绘制逻辑图如下所示：

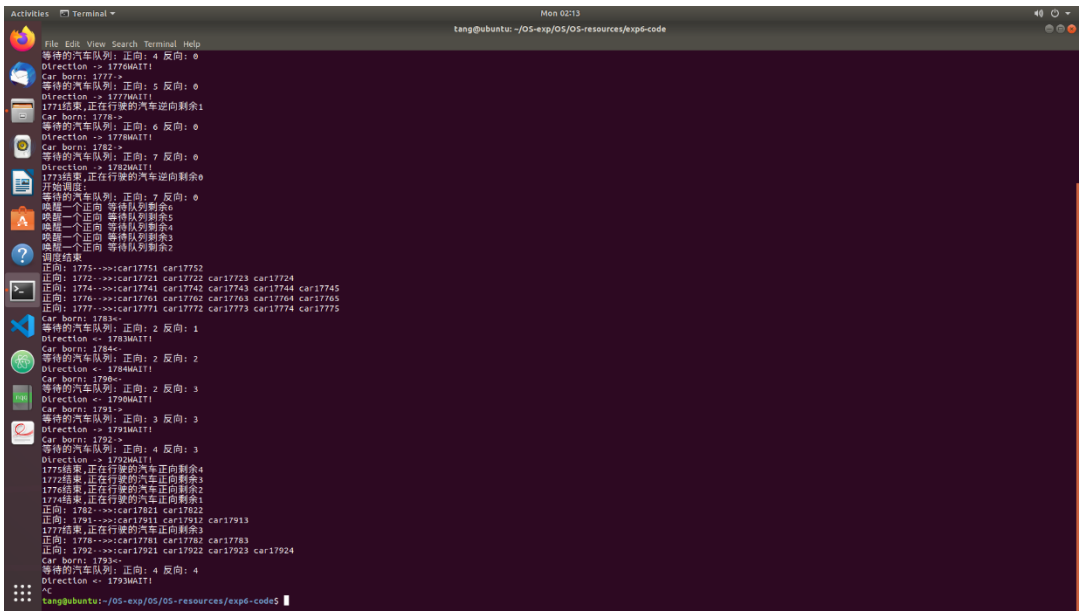



3、设计函数细节实现逻辑设计
在上述的设计逻辑中，将一辆车(一个子进程)的执行分成了三个阶段：Arrive、Cross、Quit。系统每隔一段时间，执行 fork 生成一个子进程，每个子进程执行完成三个函数即完成了生成(等待)-通过-结束的过程。函数的设计逻辑如下：




4、运行代码并测试结果：

按照上述的代码逻辑，实现了程序，并且进行了测试，达到了实验的要求，并且保证了系统的工作效率较高。测试结果如下所示：




实验总结：

通过本次实验的实例程序和自己设计的程序，加深了对锁机制和管程应用的理解，对课本上学到的理论知识有了更加实际的认识：

1、锁机制：在多线程编程中，操作系统引入了锁机制。通过锁机制，能够保证在多核多线程环境中，在某一个时间点上，只能有一个线程进入临界区代码，从而保证临界区中操作数据的一致性。所谓的锁，可以理解为内存中的一个整型数，拥有两种状态：空闲状态和上锁状态。加锁时，判断锁是否空闲，如果空闲，修改为上锁状态，返回成功；如果已经上锁，则返回失败。解锁时，则把锁状态修改为空闲状态。

2、管程(monitor)只是保证了同一时刻只有一个进程在管程内活动,即管程内定义的操作在同一时刻只被一个进程调用(由编译器实现).但是这样并不能保证进程以设计的顺序执行,因此需要设置 condition 变量,让进入管程而无法继续执行的进程阻塞自己。

3、通过死锁问题的编写和调试，意识到了在死锁解决的过程中，也存在性能问题。保证系统正确运行的同时，提高系统的效率是至关重要的。此外，在解饥饿问题时，要考虑普遍情况和特殊情况，才能保证系统的鲁棒性优秀。

附录：程序源代码

Dp.h

```
#include <iostream>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#include <sys/sem.h>
#include <sys/msg.h>
#include <sys/wait.h>

using namespace std;
// 信号灯控制用的共同体
typedef union semuns {
	int val;
} Sem_uns;

//管程中使用的信号量
class Sema {
private:
	int sem_id; //信号量标识符
public:
	Sema(int id) {
		sem_id = id;
	}
	~Sema() {}

    //信号量加 1
	int down() {
		struct sembuf buf;
		buf.sem_op = -1;
		buf.sem_num = 0;
		buf.sem_flg = SEM_UNDO;
		if ((semop(sem_id, &buf, 1)) < 0) {
			perror("down error ");
			exit(EXIT_FAILURE);
		}
		return EXIT_SUCCESS;
	}
	//信号量减 1
	int up() {
```

```
        Sem_uns arg;
        struct sembuf buf;
        buf.sem_op = 1;
        buf.sem_num = 0;
        buf.sem_flg = SEM_UNDO;
        if ((semop(sem_id, &buf, 1)) < 0)
        {
            perror("up error ");
            exit(EXIT_FAILURE);
        }
        return EXIT_SUCCESS;
    }
};

//管程中使用的锁
class Lock {
private:
    Sema *sema; //锁使用的信号量
public:
    Lock(Sema *s) {
        sema = s;
    }
    ~Lock(){ }

    //上锁
    void close_lock() { sema->down(); }

    //开锁
    void open_lock() { sema->up(); }
};

class Condition {
public:
    Condition(Sema *semax1, Sema *semax2) {
        sema0 = semax1;
        sema1 = semax2;
    }
    ~Condition(){};
    // 过路条件不足时阻塞 看看是否能通过
    void Wait(Lock *lock, int direction) {

        if (direction == 0) {
            lock->open_lock();
            cout <<"Direction -> "<< getpid() << "WAIT!" << "\n";//正向->
```

```
            sema0->down();
            lock->close_lock();
        }
        else if (direction == 1) {
            lock->open_lock();
            cout <<"Direction <- "<< getpid() << "WAIT!" << "\n";//反向<-
            sema1->down();
            lock->close_lock();
        }
    }

    //唤醒相反方向阻塞车辆
    int Signal(int direction) {
        int sig_ren;
        if (direction == 0) { //唤醒一个方向
            sig_ren = sema0->up();
        }
        else if (direction == 1) {
            sig_ren = sema1->up();
        }
        return sig_ren;
    }
private:
    Sema *sema0; // 一个方向阻塞队列
    Sema *sema1; // 另一方向阻塞队列
    Lock *lock;  // 进入管程时获取的锁
};

class OneWay  {
private:
    // 调用函数
    int get_ipc_id(char *proc_file, key_t key);
    int set_sem(key_t sem_key, int sem_val, int sem_flag);
    char *set_shm(key_t shm_key, int shm_num, int shm_flag);

    int MaxCars;              // 最大同向车数
    int *NumCars;             // 当前正在通过的车辆数
    int *CurrentDirect;   // 当前通过的车辆的方向
    int *Frontward;          // 正向等待人数
    int *Backward;           // 反向等待人数
    int *SumPassedCars;   // 已经通过的车辆总数
    Condition *condition;// 通过单行道的条件变量
    Lock *lock;               // 单行道管程锁
public:
```

```
    OneWay(int CarLimit);
    ~OneWay() { delete condition; }
    // 车辆准备上单行道,direction 为行车方向
    void Arrive(int direction,int limit);
    // 车辆正在通过单行道
    void Cross(int direction);
    // 车辆通过了单行道
    void Quit(int direction);
};

OneWay::OneWay(int CarLimit) {
    int ipc_flg = IPC_CREAT | 0644;
    MaxCars = CarLimit;

    // 共享内存
    NumCars = (int *)set_shm(201, 1, ipc_flg);        //当前方向上通过的总的车辆数
    CurrentDirect = (int *)set_shm(301, 1, ipc_flg); //当前方向 0 正向 1 反向
    SumPassedCars = (int *)set_shm(401, 1, ipc_flg);//已经通过的车辆总数
    Frontward = (int *)set_shm(501, 1, ipc_flg);// 等待人数
    Backward = (int *)set_shm(502, 1, ipc_flg);// 等待人数
    //初始化
    *NumCars = 0;
    *CurrentDirect = 0;
    *SumPassedCars = 0;
    *Frontward = 0;
    *Backward = 0;
    // 信号量
    int sema0_id = set_sem(601, 0, ipc_flg);
    int sema1_id = set_sem(602, 0, ipc_flg);
    int semaLock_id = set_sem(701, 1, ipc_flg);
    //锁和信号量
    lock = new Lock(new Sema(semaLock_id));
    condition = new Condition(new Sema(sema0_id), new Sema(sema1_id));
}

void OneWay::Arrive(int direction,int limit) {
    lock->close_lock();
    if(direction==0){
        cout<<"Car born: "<<getpid()<<"->"<<endl;
    }else{
        cout<<"Car born: "<<getpid()<<"<-"<<endl;
    }
    //没有等待的车 --> 到达的车直接通行
    if (*NumCars == 0 && *Frontward == 0 && *Backward == 0) *CurrentDirect = direction;
```

```
    //方向不对->等待防止撞车 || 在路上的车太多->防止超过最大行车数量 || 某边等待大于队列长度的一半 && 路上有车
    if (*CurrentDirect != direction || *NumCars >= MaxCars || ((*Frontward > limit/2 || *Backward > limit/2) && *NumCars > 0)) {
        if (direction == 0){
            *Frontward += 1;
        } else{
            *Backward += 1;
        }
        cout << "等待的汽车队列: " <<"正向: "<<*Frontward<< " " <<"反向: "<< *Backward << endl;
        condition->Wait(lock, direction);
    }

    *NumCars = *NumCars + 1;
    *CurrentDirect = direction;

    lock->open_lock();
}

void OneWay::Cross(int direction) {
    lock->close_lock();
    if(direction==0){
        cout<<"正向: "<<getpid()<<"-->>:";
    }else{
        cout<<"反向: "<<getpid()<<"<<--:";
    }
    int num=1;
    for (int i = 1; i <= *NumCars; ++i) {
        cout<<"car"<<getpid()<<num++<<" ";
    }
    cout << endl;
    lock->open_lock();
    sleep(10);
}

void OneWay::Quit(int direction) {
    lock->close_lock();
    *NumCars -= 1;
    if(direction==0){
        cout<<getpid()<<"结束,正在行驶的汽车正向剩余" << *NumCars << endl;
    }else
    {
        cout<<getpid()<<"结束,正在行驶的汽车逆向剩余" << *NumCars << endl;
```

```
    }


    if (*NumCars == 0) {
        int max = MaxCars - *NumCars;
        cout <<"开始调度:"<<endl;
        cout << "等待的汽车队列: " <<"正向: "<<*Frontward<< " " <<"反向: "<< *Backward
<< endl;
        if (direction == 0) {
            if (*Backward > 0) while (max && (*Backward > 0)) {
                condition->Signal(1);
                *Backward = *Backward - 1;
                max--;
                cout << "唤醒一个逆向 等待队列剩余" << *Backward << endl;
            }
            else while (max && (*Frontward > 0)) {
                condition->Signal(0);
                *Frontward = *Frontward - 1;
                max--;
                cout << "唤醒一个正向 等待队列剩余" << *Frontward << endl;
            }
        }
        else if (direction == 1) {
            if (*Frontward > 0) while (max && (*Frontward > 0)) {
                condition->Signal(0);
                *Frontward = *Frontward - 1;
                max--;
                cout << "唤醒一个正向 等待队列剩余" << *Frontward << endl;
            }
            else while (max && (*Backward > 0)) {
                condition->Signal(1);
                *Backward = *Backward - 1;
                max--;
                cout << "唤醒一个逆向 等待队列剩余" << *Backward << endl;
            }
        }
        cout << "调度结束"<<endl;
    }
    lock->open_lock();
}


int OneWay::get_ipc_id(char *proc_file, key_t key) {
```

```
#define BUFSZ 256
    FILE *pf;
    int i, j;
    char line[BUFSZ], colum[BUFSZ];
    if ((pf = fopen(proc_file, "r")) == NULL) {
        perror("Proc file not open");
        exit(EXIT_FAILURE);
    }
    fgets(line, BUFSZ, pf);
    while (!feof(pf)) {
        i = j = 0;
        fgets(line, BUFSZ, pf);
        while (line[i] == ' ') i++;
        while (line[i] != ' ') colum[j++] = line[i++];
        colum[j] = '\0';
        if (atoi(colum) != key) continue;
        j = 0;
        while (line[i] == ' ') i++;
        while (line[i] != ' ') colum[j++] = line[i++];
        colum[j] = '\0';
        i = atoi(colum);
        fclose(pf);
        return i;
    }
    fclose(pf);
    return -1;
}

char *OneWay::set_shm(key_t shm_key, int shm_num, int shm_flg) {
    int i, shm_id;
    char *shm_buf;
    //测试由 shm_key 标识的共享内存区是否已经建立
    if ((shm_id = get_ipc_id("/proc/sysvipc/shm", shm_key)) < 0) {
        //shmget 新建 一个长度为 shm_num 字节的共享内存
        if ((shm_id = shmget(shm_key, shm_num, shm_flg)) < 0) {
            perror("shareMemory set error");
            exit(EXIT_FAILURE);
        }
        //shmat 将由 shm_id 标识的共享内存附加给指针 shm_buf
        if ((shm_buf = (char *)shmat(shm_id, 0, 0)) < (char *)0) {
            perror("get shareMemory error");
            exit(EXIT_FAILURE);
        }
        for (i = 0; i < shm_num; i++) shm_buf[i] = 0; //初始为 0
```

```
    }
    //共享内存区已经建立,将由 shm_id 标识的共享内存附加给指针 shm_buf
    if ((shm_buf = (char *)shmat(shm_id, 0, 0)) < (char *)0) {
        perror("get shareMemory error");
        exit(EXIT_FAILURE);
    }
    return shm_buf;
}
int OneWay::set_sem(key_t sem_key, int sem_val, int sem_flg) {
    int sem_id;
    Sem_uns sem_arg;
    //测试由 sem_key 标识的信号量是否已经建立
    if ((sem_id = get_ipc_id("/proc/sysvipc/sem", sem_key)) < 0) {
        //semget 新建一个信号灯,其标号返回到 sem_id
        if ((sem_id = semget(sem_key, 1, sem_flg)) < 0) {
            perror("semaphore create error");
            exit(EXIT_FAILURE);
        }
    }
    //设置信号量的初值
    sem_arg.val = sem_val;
    if (semctl(sem_id, 0, SETVAL, sem_arg) < 0) {

        perror("semaphore set error");
        exit(EXIT_FAILURE);
    }
    return sem_id;
}
Main.c
#include "dp.h"
#include <iostream>

int main(int argc, char *argv[]) {
    int limit;
    if (argv[1] != NULL) limit = atoi(argv[1]);
    else limit = 5;
    OneWay oneWay(limit); // 最大十辆车
    int pid = fork();
    while (pid != 0) { //创建子进程
        sleep(rand() % 5);
        pid = fork();
    }
    srand(time(NULL));
    int direct = rand() % 2; //决定东西方向
```

```
        oneWay.Arrive(direct,limit); //进入
        oneWay.Cross(direct); //通过
        oneWay.Quit(direct); //离开
        return EXIT_SUCCESS;
}
```