2.1 苹果 橘子 问题

#include <stdio.h>

#include <pthread.h>

#include <semaphore.h>

#define N 5

sem\_t mutex, apple, orange;

int plate = 0; //盘子状态，0表示空盘子，1表示有苹果，2表示有桔子

void\* father(void\* arg) {

while (1) {

sem\_wait(&mutex); //获取互斥锁

if (plate == 0) { //盘子空，可以放苹果

printf("Father puts an apple into the plate\n");

plate = 1; //修改盘子状态为有苹果

sem\_post(&apple); //释放放苹果的信号量

}

sem\_post(&mutex); //释放互斥锁

}

}

void\* mother(void\* arg) {

while (1) {

sem\_wait(&mutex); //获取互斥锁

if (plate == 0) { //盘子空，可以放桔子

printf("Mother puts an orange into the plate\n");

plate = 2; //修改盘子状态为有桔子

sem\_post(&orange); //释放放桔子的信号量

}

sem\_post(&mutex); //释放互斥锁

}

}

void\* son(void\* arg) {

while (1) {

sem\_wait(&orange); //获取放桔子的信号量

sem\_wait(&mutex); //获取互斥锁

if (plate == 2) { //有桔子，可以吃

printf("Son eats the orange\n");

plate = 0; //修改盘子状态为空盘子

}

sem\_post(&mutex); //释放互斥锁

sem\_post(&mutex); //释放放苹果的信号量

}

}

void\* daughter(void\* arg) {

while (1) {

sem\_wait(&apple); //获取放苹果的信号量

sem\_wait(&mutex); //获取互斥锁

if (plate == 1) { //有苹果，可以吃

printf("Daughter eats the apple\n");

plate = 0; //修改盘子状态为空盘子

}

sem\_post(&mutex); //释放互斥锁

sem\_post(&orange); //释放放桔子的信号量

}

}

int main() {

sem\_init(&mutex, 0, 1); //互斥锁，初值为1

sem\_init(&apple, 0, 0); //放苹果的信号量，初值为0

sem\_init(&orange, 0, 0); //放桔子的信号量，初值为0

pthread\_t tid\_father, tid\_mother, tid\_son, tid\_daughter;

pthread\_create(&tid\_father, NULL, father, NULL);

pthread\_create(&tid\_mother, NULL, mother,

2.1.1 伪代码

semaphore plate = 1;

semaphore apple = 0;

semaphore orange = 0;

process Father() {

while (true) {

// 父亲向盘子里放苹果

sem\_wait(&plate);

// 放苹果

printf("Father puts an apple into the plate.\n");

// 通知儿子盘子里有水果了

sem\_post(&apple);

}

}

process Mother() {

while (true) {

// 母亲向盘子里放桔子

sem\_wait(&plate);

// 放桔子

printf("Mother puts an orange into the plate.\n");

// 通知女儿盘子里有水果了

sem\_post(&orange);

}

}

process Son() {

while (true) {

// 儿子等待盘子中出现桔子

sem\_wait(&apple);

// 取出桔子吃掉

printf("Son takes an orange from the plate and eats it.\n");

// 通知盘子已经空了

sem\_post(&plate);

}

}

process Daughter() {

while (true) {

// 女儿等待盘子中出现苹果

sem\_wait(&orange);

// 取出苹果吃掉

printf("Daughter takes an apple from the plate and eats it.\n");

// 通知盘子已经空了

sem\_post(&plate);

}

}

四个进程分别为父亲、母亲、儿子、女儿。使用了三个信号量：**plate**表示盘子的状态，1表示盘子空，0表示盘子满；**apple**表示苹果的状态，1表示盘子里有苹果，0表示盘子里没有苹果；**orange**表示桔子的状态，1表示盘子里有桔子，0表示盘子里没有桔子。

父亲和母亲分别等待盘子空闲，当盘子空闲后就向盘子中放入一个水果，并通知对应的子进程。儿子和女儿则分别等待对应的水果在盘子中出现，取出水果后吃掉，并通知盘子已经空闲。在以上过程中，使用了信号量来控制进程的同步和互斥。

2.2.1 用wait single 操作实现 get copy put

#define BUFFER\_SIZE 10 // 缓冲区大小

int buffer[BUFFER\_SIZE]; // 缓冲区

int in = 0; // 缓冲区输入指针

int out = 0; // 缓冲区输出指针

sem\_t empty; // 缓冲区空信号量

sem\_t full; // 缓冲区满信号量

sem\_t mutex; // 互斥信号量

// 初始化信号量

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

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

sem\_init(&mutex, 0, 1);

// 进程A代码

void get\_data() {

int data;

sem\_wait(&full); // 等待缓冲区中有数据

sem\_wait(&mutex); // 申请访问缓冲区的互斥信号量

data = buffer[out];

out = (out + 1) % BUFFER\_SIZE; // 指针移动到下一个位置

sem\_post(&mutex); // 释放缓冲区的互斥信号量

sem\_post(&empty); // 通知缓冲区有空位

// 对数据进行处理...

}

// 进程B代码

void put\_data(int data) {

sem\_wait(&empty); // 等待缓冲区有空位

sem\_wait(&mutex); // 申请访问缓冲区的互斥信号量

buffer[in] = data;

in = (in + 1) % BUFFER\_SIZE; // 指针移动到下一个位置

sem\_post(&mutex); // 释放缓冲区的互斥信号量

sem\_post(&full); // 通知缓冲区有数据

}

// 主函数

int main() {

// 创建进程A和B并分别执行get\_data()和put\_data()函数

2.2.2 试从物理概念上说明记录型信号量的wait和signal。

wait操作：如果信号量的值为0，那么就将当前线程（或进程）加入到等待队列中，让它睡眠，直到该信号量的值变为1。等待期间，线程将不会占用任何CPU资源。当信号量的值变为1时，就会唤醒一个等待的线程，使它从睡眠状态中醒来，并继续执行wait之后的代码。如果信号量的值本来就为1，则什么也不做，直接返回。

signal操作：将信号量的值加1。如果此时有等待的线程（或进程），则会唤醒其中的一个，使它从睡眠状态中醒来，并开始执行wait之后的代码。如果没有等待的线程，则什么也不做，直接返回。

从物理上理解，可以将记录型信号量看做一个开关，其值为0时表示关，值为1时表示开。wait操作相当于将线程挂在开关上等待开关打开；signal操作相当于打开开关，如果有线程在等待则会将其中的一个线程唤醒。

2.2.3

// wait 操作

void wait(semaphore \*s) {

s->value--; // value 减 1

if (s->value < 0) {

// 阻塞当前进程，并将其加入等待队列

add\_to\_waiting\_queue(s->waiting\_queue, current\_process);

// 切换上下文，执行其他进程

switch\_to\_another\_process();

}

}

// signal 操作

void signal(semaphore \*s) {

s->value++; // value 加 1

if (s->value <= 0) {

// 唤醒等待队列中的一个进程

wakeup\_one\_process(s->waiting\_queue);

// 切换上下文，执行被唤醒的进程

switch\_to\_wakeup\_process();

}

}

在 wait 操作中，首先将信号量的 value 减 1。如果 value 变成了负数，说明资源已经被其他进程占用，当前进程需要被阻塞，并将其加入等待队列。然后，切换上下文，执行其他进程。

在 signal 操作中，首先将信号量的 value 加 1。如果 value 大于 0，说明此时没有进程在等待该资源，直接返回。否则，唤醒等待队列中的一个进程，并切换上下文，执行被唤醒的进程。

2.3.1 为什么进程在进入临界区之前应先执行“进入区”代码？而在退出前又要执行“退出区”代码

进程在进入临界区之前应先执行“进入区”代码，这是为了保证在进程进入临界区之前，能够判断该进程是否有访问临界资源的权利，如果没有则需要等待。进入区代码通常是执行P操作（wait）操作，该操作会将信号量的值减一，当信号量的值小于0时，表示当前进程需要等待。这样能够保证每个进程在访问临界区之前，都必须获取到访问临界区的权限，从而避免了多个进程同时访问临界资源的问题。

而在退出临界区之前又要执行“退出区”代码，这是为了保证在进程退出临界区后，能够释放对临界资源的访问权限，以便其他进程能够访问该资源。退出区代码通常是执行V操作（signal）操作，该操作会将信号量的值加一，当信号量的值大于0时，表示当前有进程在等待，可以唤醒等待的进程继续执行。这样能够保证每个进程在访问临界区之后，都必须释放对临界资源的访问权限，从而避免了资源的浪费和其他进程无法访问临界资源的问题。