卡通画

描述已自动生成

**操作系统实验报告**

**实验题目** 进程同步与信号量的实现

**学生姓名**  杨程锦

**学 号**  2021214710

**专业班级**  计科21-1班

**指导教师**  田卫东

**完成日期**  2023.12.27

**合肥工业大学 计算机与信息学院**

1. **实验目的和任务要求**

加深对进程同步与互斥概念的理解。

 掌握信号量的使用方法，并解决生产者—消费者问题。

 掌握信号量的实现原理。

1. **实验原理**

进程同步是为了解决多个进程之间的竞态条件和资源争夺问题，而信号量作为进程同步的一种机制，通过P（等待）和V（释放）操作来协调进程对共享资源的访问。P操作用于请求资源，若资源可用则继续执行，否则阻塞；V操作用于释放资源，同时唤醒可能在等待资源的其他进程。实验中，创建共享资源，通过信号量确保进程间有序访问该资源，验证信号量机制有效地解决了进程同步问题，避免了竞态条件的发生。这一过程旨在确保程序正确执行，提高系统的稳定性和可靠性。

1. **实验内容**

**3.1 准备实验**

使用浏览器登录平台领取本次实验对应的任务，从而在平台上创建个人项目（Linux 0.11 内核项目），然后使用VSCode将个人项目克隆到本地磁盘中。

**3.2 在内核中实现信号量的系统调用**

按照下面的内容在Linux 0.11内核项目中实现简易版的信号量。只需要修改include/unistd.h、kernel/system\_call.s和include/linux/sys.h三个文件、实现信号量的四个系统调用函数sem\_open、sem\_wait、sem\_post和sem\_unlink即可：

1. 在文件include/unistd.h中的第161行之后，定义四个新的系统调用号，如下：

#define \_\_NR\_sem\_open 87

#define \_\_NR\_sem\_wait 88

#define \_\_NR\_sem\_post 89

#define \_\_NR\_sem\_unlink 90

2. 在文件kernel/system\_call.s中的第73行，修改系统调用的总数：

nr\_system\_calls = 91

3. 在文件include/linux/sys.h中的第87行之后，添加系统调用内核函数的声明：

extern int sys\_sem\_open();

extern int sys\_sem\_wait();

extern int sys\_sem\_post();

extern int sys\_sem\_unlink();

在此文件的最后，向系统调用函数指针表sys\_call\_table[]中添加新系统调用函数的指针（注意，系统调用号必须与系统调用内核函数指针在系统调用函数表中的索引一一对应），如下：

fn\_ptr sys\_call\_table[] = {

……

sys\_sem\_open, //87

sys\_sem\_wait, //88

sys\_sem\_post, //89

sys\_sem\_unlink //90

};

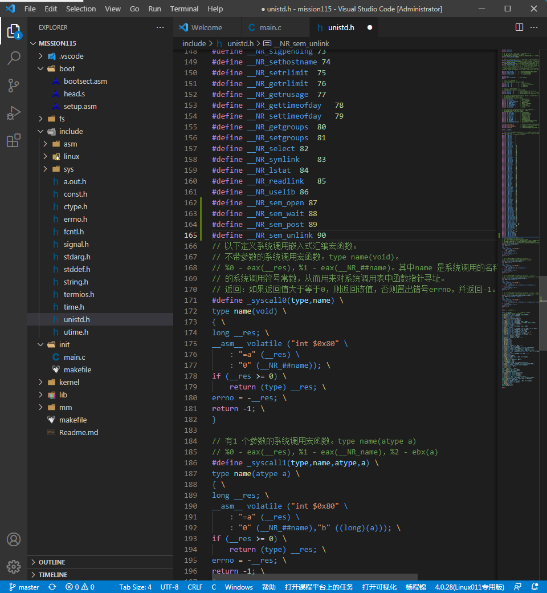


图 1 更改unistd.h

文本

描述已自动生成

图 2 更改system\_call.s

电脑萤幕画面

描述已自动生成 文本

描述已自动生成

图 3 更改sys.h

4. 打开“学生包”文件夹，在本实验对应的文件夹中找到“sem.c”文件。将此文件拖动到VSCode中释放，即可打开此文件。其中实现了信号量的四个系统调用函数。

5. 在VSCode的“文件资源管理器”窗口中，右键点击“kernel”文件夹节点，选择菜单“New File”新建一个名为“semaphore.c”的文件，并将步骤4中找到的“sem.c”中的代码复制到刚刚创建的kernel/semaphore.c文件中。

文本

描述已自动生成

图 4 semaphore.c

6. 生成项目，确保没有语法错误。

文本

描述已自动生成

图 5 生成项目

1. **实验的思考与问题分析**
2. 答：

在原始代码中，生产者和消费者的同步机制使用了信号量（Mutex、Empty、Full）。尽管基本上是标准实现，但存在潜在的问题。特别是，如果生产者在等待空闲缓存资源时被中断，可能导致死锁，因为互斥锁可能无法释放。

标准解决方案采用条件变量（Empty和Full）来改进。这种改进避免了潜在的死锁情况。通过等待和发出信号的方式，生产者和消费者能够更可靠地协同工作，确保缓存的正确使用，而不会出现可能导致死锁的问题。

2、答：

Producer.c:

#include <stdio.h>

#include <stdlib.h>

#include <unistd.h>

#include <sys/types.h>

#include <sys/ipc.h>

#include <sys/shm.h>

#define SHM\_SIZE 4 // 共享内存大小

int main() {

key\_t key = ftok("shared\_memory", 65); // 创建共享内存的key

int shmid = shmget(key, SHM\_SIZE, IPC\_CREAT | 0666); // 创建共享内存

if (shmid == -1) {

perror("shmget");

exit(EXIT\_FAILURE);

}

int \*shared\_memory = (int \*)shmat(shmid, NULL, 0);

while (1) {

// 生产者的代码逻辑

// ...

sleep(1); // 模拟生产时间

}

shmdt(shared\_memory); // 分离共享内存

shmctl(shmid, IPC\_RMID, NULL); // 删除共享内存

return 0;

}

Consumer.c:

#include <stdio.h>

#include <stdlib.h>

#include <unistd.h>

#include <sys/types.h>

#include <sys/ipc.h>

#include <sys/shm.h>

#define SHM\_SIZE 4 // 共享内存大小

int main() {

key\_t key = ftok("shared\_memory", 65); // 使用相同的key获取共享内存

int shmid = shmget(key, SHM\_SIZE, 0666); // 获取共享内存

if (shmid == -1) {

perror("shmget");

exit(EXIT\_FAILURE);

}

int \*shared\_memory = (int \*)shmat(shmid, NULL, 0);

while (1) {

// 消费者的代码逻辑

// ...

sleep(1); // 模拟消费时间

}

shmdt(shared\_memory); // 分离共享内存

return 0;

}

1. 答：

#include <linux/kernel.h>

#include <linux/module.h>

#include <linux/sched.h>

#include <linux/wait.h>

DECLARE\_WAIT\_QUEUE\_HEAD(wait\_queue); // 定义等待队列

struct semaphore {

int count;

struct wait\_queue\_head wait\_queue;

};

// 初始化信号量

void sema\_init(struct semaphore \*sem, int val) {

sem->count = val;

init\_waitqueue\_head(&sem->wait\_queue);

}

// P操作

void down(struct semaphore \*sem) {

while (1) {

if (sem->count > 0) {

sem->count--;

break;

} else {

// 阻塞当前进程

DEFINE\_WAIT(wait);

prepare\_to\_wait(&sem->wait\_queue, &wait, TASK\_INTERRUPTIBLE);

schedule(); // 让出CPU

finish\_wait(&sem->wait\_queue, &wait);

}

}

}

// V操作

void up(struct semaphore \*sem) {

sem->count++;

wake\_up(&sem->wait\_queue); // 唤醒等待队列中的一个进程

}

// 生产者代码

void producer(struct semaphore \*empty, struct semaphore \*full, struct semaphore \*mutex) {

while (1) {

down(empty); // 等待空闲缓存资源

down(mutex); // 进入临界区

// 生产产品

up(mutex); // 离开临界区

up(full); // 通知产品资源

}

}

// 消费者代码

void consumer(struct semaphore \*empty, struct semaphore \*full, struct semaphore \*mutex) {

while (1) {

down(full); // 等待产品资源

down(mutex); // 进入临界区

// 消费产品

up(mutex); // 离开临界区

up(empty); // 通知空闲缓存资源

}

}

使用sleep\_on和wake\_up函数来实现信号量的阻塞和唤醒需要定义等待队列，并在等待和唤醒的过程中合理使用相关函数。与之前直接使用sleep和wake\_up相比，这样的修改可能会引入更复杂的代码结构，但它更符合Linux内核中同步的实现方式。在实验中，1个生产者和4个消费者同步运行的结果可能会受到等待队列唤醒机制的影响，具体表现在进程调度和竞争条件上。

1. **总结和感想体会**

实验七深入探讨了进程同步与信号量的实现，提供了加深对这些核心概念理解的机会。通过实验，我了解到信号量是一个高级的同步机制，可以有效地解决进程间的竞争条件和协调问题。特别是通过在Linux 0.11内核中实现信号量的系统调用，我更加深入地理解了操作系统是如何在底层管理进程和资源的。实现信号量系统调用的过程中，我不仅学习了如何在内核级别编程，还学到了如何利用信号量解决实际问题，例如经典的生产者-消费者问题。这个实验强调了操作系统设计中的一些基本原则，比如进程同步、资源共享和原子操作的重要性。