Lab8 POSIX

一、 实验概述

本次实验，我们学习使用POSIX标准系统接口和POSIX进程间通信，包括自旋锁、互斥锁、条件变量、信号量、消息队列、共享内存。

并且学习在ucore中实现信号量的方式。

二、实验目的

1. POSIX Pthread
2. spin_lock vs mutex
3. semaphore
4. condition variables
5. shared memory
6. message queue
7. 掌握在ucore中信号量机制的具体实现

三、实验项目整体框架概述

// Lab8
├── kern
│ ├── debug
│ ├── driver
│ ├── fs
│ ├── init
│ ├── libs
│ ├── mm
│ ├── process
│ ├── schedule
│ ├── **sync**
│ │ ├── **check_sync.c** //哲学家算法
│ │ ├── **sem.c** //信号量
│ │ ├── **sem.h**
│ │ ├── sync.h
│ │ ├── **wait.c** //等待队列
│ │ └── **wait.h**
│ ├── syscall
│ └── trap
├── libs
├── Makefile
├── tools
└── user

四、实验内容

阅读并运行样例代码:

Step 1. POSIX Pthread

Function	Description
pthread_create	create a new thread
pthread_exit	terminate calling thread
pthread_cancel	send a cancellation request to a thread
pthread_join	wait for the specified thread
pthread_detach	detach a thread

Sample code:

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/stat.h>

int a = 0;

void *add()
{
    for (int i = 0; i < 3; i++)
    {
        int b = a;
        b++;
        a = b;
        printf("%d\n", a);
    }
}

int main(int argc, char *argv[])
{
    pthread_t p1, p2;

    // Create two threads (both run func)
    pthread_create(&p1, NULL, add, NULL);
    pthread_create(&p2, NULL, add, NULL);

    // Wait for the threads to end.
    pthread_join(p1, NULL);
    pthread_join(p2, NULL);
}
```

> **Race Condition —— The too much milk problem**

> Mom and Dad are all used to checking the fridge when they arrive home. If milk run out, he or she will leave home to buy milk. The fridge is small in your home that only one bottle of milk can be put in it at a time. Mom and Dad always arrive home at different time.
>
> | Time | Dad | Mom |
> | --- | --- | --- |
> | 3:00 | Arrive Home | |
> | 3:05 | Look in fridge, no milk | |
> | 3:10 | Leave for supermarket | |
> | 3:15 | | Arrive Home |
> | 3:20 | Arrive at Supermarket | Look in fridge, no milk |
> | 3:25 | Buy Milk | Leave for supermarket |
> | 3:30 | Arrive home, put milk into fridge | |
> | 3:35 | | Arrive at Supermarket |
> | 3:40 | | Buy milk |
> | 3:45 | | Arrive home, put milk in fridge |
> | 3:50 | | Oh ..... |
>
> **Compile and run `milk.c`**
>
> ```
> gcc milk.c -o milk -pthread
> ./milk
> ```

You may found the outcome of an execution depends on a particular order in which the shared resource is accessed.

Step 2. spin lock vs mutex

Function	Description
pthread_spin_init	initialize a spin lock
pthread_spin_lock	lock a spin lock
pthread_spin_unlock	unlock a spin lock
pthread_mutex_lock	lock a mutex
pthread_mutex_unlock	unlock a mutex
pthread_mutex_trylock	try to lock a mutex
pthread_spin_trylock	try to lock a spin lock

Sample code——spin lock:

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/stat.h>
int a = 0;
pthread_spinlock_t spinlock;
```

```
void *add()
{
    for (int i = 0; i < 3; i++)
    {
        pthread_spin_lock(&spinlock);
        int b = a;
        b++;
        a = b;
        printf("%d\n", a);
        pthread_spin_unlock(&spinlock);
    }
}

int main(int argc, char *argv[])
{
    pthread_t p1, p2;
    pthread_spin_init(&spinlock, 0);

    // Create two threads (both run func)
    pthread_create(&p1, NULL, add, NULL);
    pthread_create(&p2, NULL, add, NULL);

    // Wait for the threads to end.
    pthread_join(p1, NULL);
    pthread_join(p2, NULL);

    pthread_spin_destroy(&spinlock);
}
```

> **The too much milk problem**
>
> Try to fix the too much milk problem by spin_lock(busy waiting). When Dad go to buy milk, he will lock the fridge. If Mom found the fridge locked, she will spin before the fridge until it unlocked. Vice versa.

Sample code——mutex lock:　　闲锁，没有计算资源，要通过切换才能获得计算资源

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/stat.h>
int a = 0;
pthread_mutex_t mutex;

void *add()
{
    for (int i = 0; i < 3; i++)
    {
        pthread_mutex_lock(&mutex);
        int b = a;
        b++;
        a = b;
        printf("%d\n", a);
        pthread_mutex_unlock(&mutex);
    }
```

```
}

int main(int argc, char *argv[])
{
    pthread_t p1, p2;
    pthread_mutex_init(&mutex, NULL);

    // Create two threads (both run func)
    pthread_create(&p1, NULL, add, NULL);
    pthread_create(&p2, NULL, add, NULL);

    // Wait for the threads to end.
    pthread_join(p1, NULL);
    pthread_join(p2, NULL);

    pthread_mutex_destroy(&mutex);
}
```

> **The too much milk problem**
>
> Try to fix the too much milk problem by mutex. When Dad go to buy milk, he will lock the fridge. If Mom found the fridge locked, she will sleep until it unlocked. Vice versa.
>
> **Better way**
>
> Try trylock to realize:
>
> - Mom see the fridge is empty.
> - Mom leave a note and then go buy milk.
> - Dad open the fridge and see the note.
> - Dad just go away.

Step 3. semphores——POSIX IPC(Inter-Process Communication)

同时保证几个进程同时访问

How to realize `semaphore`:

```
struct semaphore {
       int count;
       queueType queue;
};
void semWait(semaphore s)
{
       s.count--;
       if (s.count < 0) {
         /* place this process in s.queue */;
         /* block this process */;
       }
}
void semSignal(semaphore s)
{
       s.count++;
       if (s.count<= 0) {
         /* remove a process P from s.queue */;
         /* place process P on ready list */;
       }
}
```

APIs:

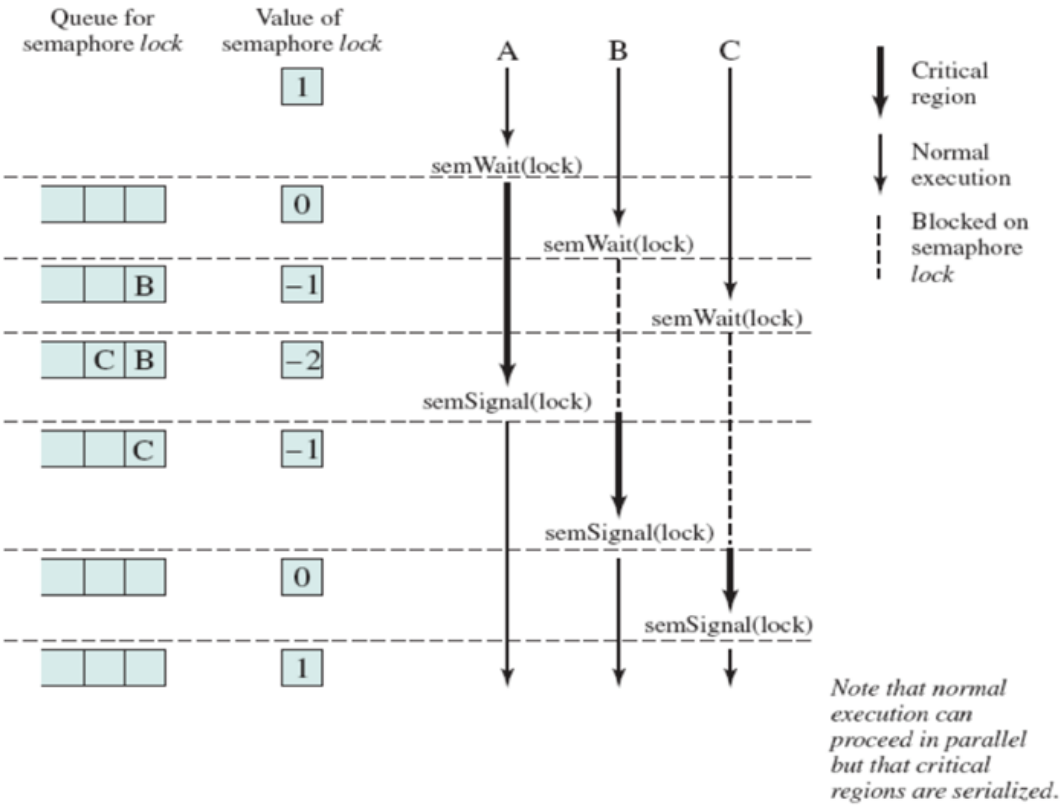
Function	Description
sem_open	opens/creates a named semaphore for use by a process **有名信号量**
sem_wait	lock a semaphore
sem_post	unlock a semaphore
sem_close	deallocates the specified named semaphore
sem_unlink	removes a specified named semaphore
sem_getvalue	get semaphore value

Sample code:

```
#include <fcntl.h>           /* For O_* constants */
#include <sys/stat.h>        /* For mode constants */
#include <semaphore.h>
#include <stdio.h>

int main(int argc, char * argv[]){
    char * name = "my_semaphore";
    int VALUE = 2;
    sem_t * sema;
    //If semaphore with name does not exist, then create it with VALUE
    printf("Open or Create a named semaphore, %s, its init value is %d\n",
name,VALUE);
    sema = sem_open(name, O_CREAT, 0666, VALUE);
    //wait on semaphore sema and decrease it by 1
    sem_wait(sema);
    sem_getvalue(sema, &VALUE);
    printf("Decrease semaphore by 1, now the value is %d\n", VALUE);
    //add semaphore sema by 1
    sem_post(sema);
    sem_getvalue(sema, &VALUE);
    printf("Add semaphore by 1, now the value is %d\n", VALUE);
    //Before exit, you need to close semaphore and unlink it, when all
processes have
    //finished using the semaphore, it can be removed from the system using
sem_unlink
    sem_close(sema);
    sem_unlink(name);
    return 0;
}
```

How to use:



> **The too much milk problem**
>
> Now your family bought a new fridge, it has space to store 2 bottles of milk. And you began to buy milk, there are three people in your family who buy milk. A person buys only one bottle of milk at a time.
>
> Try to use semaphore to solve this problem.

Step 4. condition variable

Function	Description
pthread_cond_wait	release lock, put thread to sleep until condition is signaled; when thread wakes up again, re-acquire lock before returning.
pthread_cond_signal	Wake up at least one of the threads that are blocked on the specified condition variable; If more than one thread is blocked on a condition variable, the scheduling policy shall determine the order in which threads are unblocked.

Sample code:

```
//Producers and Consumers.
//Two producers vs two consumers
//At any time, only one person can access count

#include <pthread.h>
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
```

```c
int count = 0;
pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t cond = PTHREAD_COND_INITIALIZER;
void *producer(void *arg)
{
    int i = 5;
    while (i--)
    {
        pthread_mutex_lock(&mutex);
        printf("producer add lock\n");
        count++;
        printf("in producer count is %d\n", count);
        if (count > 0)
        {
            pthread_cond_signal(&cond);
        }
        printf("producer release lock\n");
        pthread_mutex_unlock(&mutex);
    }
    return NULL;
}

void *consumer(void *arg)
{
    int i = 5;
    while (i--)
    {
        pthread_mutex_lock(&mutex);
        printf("consumer add lock\n");
        if (count <= 0)
        {
            printf("begin wait\n");
            pthread_cond_wait(&cond, &mutex);
            printf("end wait\n");
        }
        count--;
        printf("in consumer count is %d\n", count);
        pthread_mutex_unlock(&mutex);
        printf("consumer release lock\n");
    }
    return NULL;
}

int main()
{
    pthread_t producethread1, producethread2, consumethread1, consumethread2;
    pthread_create(&consumethread1, NULL, consumer, NULL);
    pthread_create(&consumethread2, NULL, consumer, NULL);
    pthread_create(&producethread1, NULL, producer, NULL);
    pthread_create(&producethread2, NULL, producer, NULL);
    pthread_join(producethread1, NULL);
    pthread_join(consumethread1, NULL);
    pthread_join(producethread2, NULL);
    pthread_join(consumethread2, NULL);
    return 0;
}
```

The too much milk problem——new problem

- Your family buy a new big fridge, which can put in 100 bottles of milk.
- Dad and you always take milk but never buy.
- Mom and sister is very frequently checking the fridge is empty or not.
- If fridge is empty, she will go buy milk. Otherwise do nothing.

The problem can be solved like:

Dad and you(two threads)

```
while (1){
 lock      闲锁
 int num=check_fridge()
 if(num>0)
     take milk
 else  cond_signal      发信号
 unlock
}
```

Mom and sister(two threads)

```
while (1){
 lock//lock mutex
 while(check_fridge()>0)//question 2
     cond_wait  //wait for cond_signal and unlock mutex
 go buy milk
 unlock
}
```

Try to realize the above solution by condition variable, and think about:

1. How to wake up a thread which entered cond_wait?
2. What will happen if change while to if?
3. Why cond_signal first rather than unlock mutex first?

Step 5. shared memory——POSIX IPC

Function	Description
shm_open	create/open POSIX shared memory objects
mmap	map files or devices into memory
munmap	unmap files or devices into memory
shm_unlink	unlink POSIX shared memory objects
ftruncate	truncate a file to a specified length

Sample code:

```
/****** headers.h ******/
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <fcntl.h>
#include <sys/shm.h>
#include <sys/stat.h>
#include <sys/mman.h>

/***** Makefile ********/
all:
    gcc -o producer producer.c -lrt
    gcc -o consumer consumer.c -lrt

/******* producer.c *******/
#include "headers.h"

int main()
{
    const char *name = "OS";
    const char *message = "Learning operating system is fun!";
    int shm_fd;
    void *ptr;
    /* create the shared memory segment */
    shm_fd = shm_open(name, O_CREAT | O_RDWR, 0666);

    /* configure the size of the shared memory segment */
    ftruncate(shm_fd, 4096);

    /* now map the shared memory segment in the address space of the process */
    ptr = mmap(0, 4096, PROT_READ | PROT_WRITE, MAP_SHARED, shm_fd, 0);
    if (ptr == MAP_FAILED)
    {
        printf("Map failed\n");
        return -1;
    }
    sprintf(ptr, "%s", message);
}

/********* consumer.c ***********/
#include "headers.h"
int main()
{
    const char *name = "OS";
    int shm_fd; // file descriptor, from shm_open()
    char *ptr;  // base address, from mmap()
    /* open the shared memory segment as if it was a file */
    shm_fd = shm_open(name, O_RDONLY, 0666);
    if (shm_fd == -1)
    {
        printf("Shared memory failed\n");
        exit(1);
    }
    /* map the shared memory segment to the address space of the process */
    ptr = mmap(0, 4096, PROT_READ, MAP_SHARED, shm_fd, 0);
    if (ptr == MAP_FAILED)
    {
        printf("Map failed\n");
```

```
        exit(1);
    }
    /* Read data */
    printf("%s", ptr);
    /* remove the named shared memory object*/
    shm_unlink(name);
}
```

Step 6. message queue——POSIX IPC

Function	Description
mq_open	open a message queue
mq_close	close a message queue descriptor
mq_getattr	get message queue attributes
mq_setattr	set message queue attributes
mq_send	send a message to a message queue
mq_receive	receive a message from a message queue

Sample code:

```
//gcc -lrt
#include <errno.h>
#include <string.h>
#include <unistd.h>
#include <fcntl.h>
#include <mqueue.h>
#include <stdio.h>
#include <stdlib.h>


int main()
{
    mqd_t mqID;
    mqID = mq_open("/mq", O_RDWR | O_CREAT | O_EXCL, 0666, NULL);

    if (fork() == 0)
    {
        struct mq_attr mqAttr;
        mq_getattr(mqID, &mqAttr);

        char buf[mqAttr.mq_msgsize];

        for (int i = 1; i <= 5; ++i)
        {
            if (mq_receive(mqID, buf, mqAttr.mq_msgsize, NULL) < 0)
            {
                printf("receive message failed. \n");
                continue;
            }
```

```
            printf("receive message %d:%s\n",i,buf);
        }
        exit(0);
    }

    char msg[] = "haha";
    for (int i = 1; i <= 5; ++i)
    {
        if (mq_send(mqID, msg, sizeof(msg), i) < 0)
        {
            printf("send message %d failed.\n",i);
            continue;
        }
        printf("send message %d success.\n",i);

        sleep(1);
    }
    mq_close(mqID);
}
```

Step 7. 实现ucore信号量

相关代码在Lab8.zip中。

屏蔽使能中断（已经用到）

内核在执行的过程中可能会被外部的中断打断，我们实现的ucore是不可抢占的系统，所以操作系统进行某些同步互斥的操作的时候需要先禁用中断，等执行完之后再打开中断。这样保证了操作系统在执行临界区的时候不会被打断，也就实现了简单的同步互斥。trap包括中断和异常，异常是无法屏蔽的。

在ucore中经常可以看到下面这样的代码：

```
.......
local_intr_save(intr_flag);
{
  临界区代码
}
local_intr_restore(intr_flag);
......
```

这段代码就是使用屏蔽使能中断处理内核同步互斥问题的例子。

信号量

信号量（semaphore）是一种同步互斥的实现。semaphore一词来源于荷兰语，原来是指火车信号灯。想象一些火车要进站，火车站里有n个站台，那么同一时间只能有n辆火车进站装卸货物。当火车站里已经有了n辆火车，信号灯应该通知后面的火车不能进站了。当有火车出站之后，信号灯应该告诉后面的火车可以进站。

这个问题放在操作系统的语境下就是有一个共享资源只能支持n个线程并行的访问，信号量统计目前有多少进程正在访问，当同时访问的进程数小于n时就可以让新的线程进入，当同时访问的进程数为n时想要访问的进程就需要等待。

现在我们需要实现ucore中的信号量。

在信号量中，一般用两种操作来刻画申请资源和释放资源：P操作申请一份资源，如果申请不到则等待；V操作释放一份资源，如果此时有进程正在等待，则唤醒该进程。在ucore中，我们使用`down`函数实现P操作，`up`函数实现V操作。

结构体

首先是信号量结构体的定义：

```
typedef struct {
    int value;
    wait_queue_t wait_queue;
} semaphore_t;
```

其中的`value`表示信号量的值，其正值表示当前可用的资源数量，负值表示正在等待资源的进程数量。`wait_queue`即为这个信号量相对应的等待队列。

等待队列

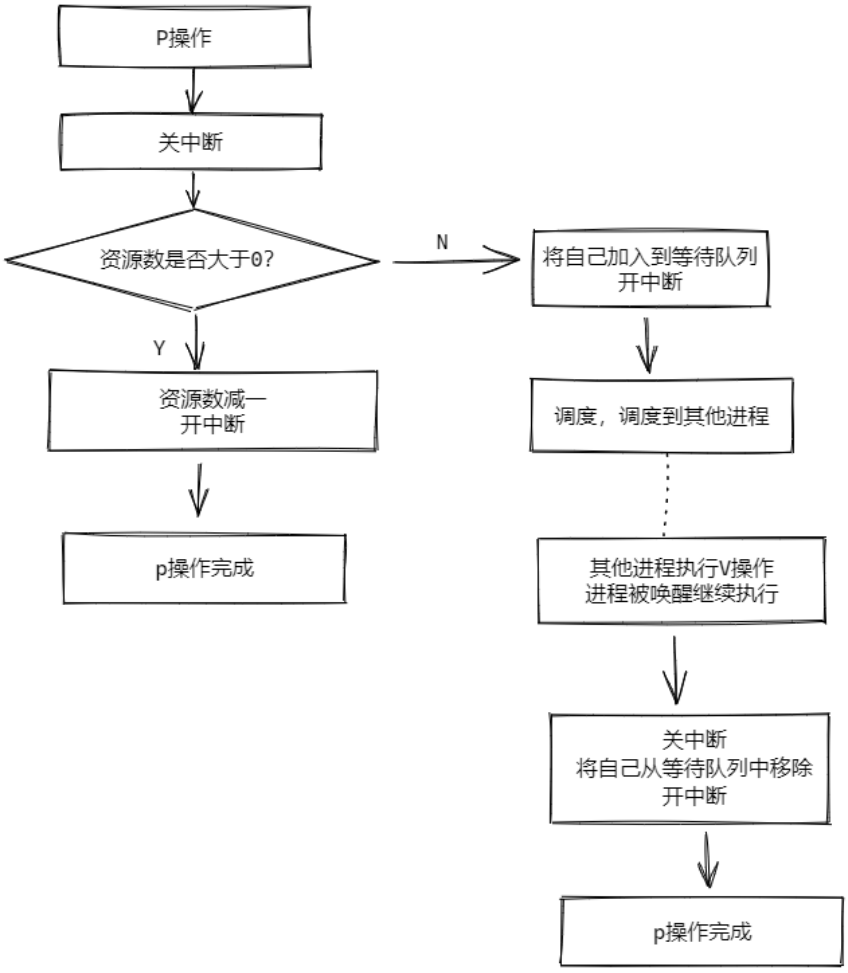
等待队列是操作系统提供的一种事件机制。一些进程可能会在执行的过程中等待某些特定事件的发生，这个时候进程进入睡眠状态。操作系统维护一个等待队列，把这个进程放进他等待的事件的等待队列中。当对应的事件发生之后，操作系统就唤醒相应等待队列中的进程。

等待队列的实现用到了前面实验中经常用到的双向链表，主要数据结构和方法是定义在`kernel\sync\wait.c`中。请大家自行阅读此部分代码。

P操作

`down`函数对应的是P操作。首先关闭中断，然后判断信号量的值是否为正。如果是正值说明进程可以获得信号量，将信号量的值减一，打开中断然后函数返回即可。否则表示无法获取信号量，将自己的进程保存进等待队列，打开中断，调用`schedule`函数进行调度。

等到V操作唤醒等待队列中的进程的时候，进程会从`schedule`函数后面开始执行，这时候将自身从等待队列中删除（此过程需要关闭中断）并返回即可。



具体的实现如下：

```
static __noinline uint32_t __down(semaphore_t *sem, uint32_t wait_state) {
    bool intr_flag;
    local_intr_save(intr_flag);
    if (sem->value > 0) {
        sem->value --;
        local_intr_restore(intr_flag);
        return 0;
    }
    wait_t __wait, *wait = &__wait;
    wait_current_set(&(sem->wait_queue), wait, wait_state);
    local_intr_restore(intr_flag);

    schedule();

    local_intr_save(intr_flag);
    wait_current_del(&(sem->wait_queue), wait);
    local_intr_restore(intr_flag);

    if (wait->wakeup_flags != wait_state) {
        return wait->wakeup_flags;
    }
    return 0;
}
```

V操作

`up`函数实现了V操作。首先关闭中断，如果释放的信号量没有进程正在等待，那么将信号量的值加一，打开中断直接返回即可。如果有进程正在等待，那么唤醒这个进程，把它放进就绪队列，打开中断并返回。具体的实现如下：

```
static __noinline void __up(semaphore_t *sem, uint32_t wait_state) {
    bool intr_flag;
    local_intr_save(intr_flag);
    {
        wait_t *wait;
        if ((wait = wait_queue_first(&(sem->wait_queue))) == NULL) {
            sem->value ++;
        }
        else {
            assert(wait->proc->wait_state == wait_state);
            wakeup_wait(&(sem->wait_queue), wait, wait_state, 1);
        }
    }
    local_intr_restore(intr_flag);
}
```

六、本节知识点回顾

在本次实验中，你需要了解以下知识点：

1. spin lock, mutex, condition variable
2. POSIX IPC
3. 如何在ucore实现信号量

七、下一实验简单介绍

下一次实验，我们将介绍ucore中的内存管理。