实验5 同步互斥

仔细阅读实验文档lab7同步互斥，完成以下练习

练习1: 了解信号量和管程的实现机制

1. 同步互斥的底层支持是如何实现的？
2. 对比原理课上学到的信号量和p，v操作，说明Ucore中信号量机制的实现。
3. Ucore中的信号量是基于信号量和条件变量实现的，请说明其中的数据结构和函数方法的设计。

同步互斥：一组并发进程中的一个或多个程序段，因为共享某一个公有资源而导致它们必须以一个不允许交叉执行的单位执行。即不允许两个以上的共享该资源的并发进程同时进入临界区称为互斥。

临界区(critical section)：每个进程中访问临界资源的那段程序（临界资源是一次仅允许一个进程使用的共享资源。）每次只准许一个进程进入临界区，进入后不允许其他进程进入。不论是硬件临界资源，还是软件临界资源，多个进程必须互斥地对它进行访问。

同步互斥是指某一资源同时只允许一个进程对其进行访问，具有唯一性和排它性，但互斥不用限制进程对资源的访问顺序，即访问可以是无序的。同步是指在进程间的执行必须严格按照规定的某种先后次序来运行，即访问是有序的，这种先后次序取决于要系统完成的任务需求。在进程写资源情况下，进程间要求满足互斥条件。在进程读资源情况下，可允许多个进程同时访问资源。

算法进入临界区的准则:有空让进、无空等待、择一而入、算法可行(等待有限)

如果有多个线程试图同时访问临界区，那么在有一个线程进入后其他所有试图访问此临界区的线程将被挂起，并一直持续到进入临界区的线程离开。临界区在被释放后，其他线程可以继续抢占，并以此达到用原子方式操作共享资源的目的。

信号量：它允许多个线程在同一时刻访问同一资源，但是需要限制在同一时刻访问此资源的最大线程数目。信号允许多个线程同时使用共享资源，这与操作系统中的P V操作相同。它指出了同时访问共享资源的线程最大数目。它允许多个线程在同一时刻访问同一资源，但是需要限制在同一时刻访问此资源的最大线程数目。信号量的关键之处在于它们原子地执行。必须确保没有两个进程能够同时对同一信号量执行wait和signal操作

P V 操作：P(sem)：表示申请一个资源 V (sem)：表示释放一个资源。信号量的初值应该大于等于0。P、V操作必须成对出现，有一个 P 操作就一定有一个 V 操作。

条件变量

    1.定义条件变量 pthread\_cond\_t cond;

    2.初始化 pthread\_cond\_init(&cond, NULL);  // 第二个条件是属性

    3.等待条件 pthread\_cond\_wait(&cond, &mutex);

        // 如果mutex没有在互斥环境，就形同虚设，在互斥环境中wait函数将其置为1(1表示打开锁，不管原来的值是多少)，

        执行结束后，wait返回，mutex恢复成原来的值。

    4.修改条件 pthread\_cond\_signal(&cond);  // 激活等待条件，相当于一个通知

    5.销毁条件变量 pthread\_cond\_destroy(&cond);

    多个生产者往缓冲区中放入东西的时候，很有可能放到同一块地方，因此要求生产者之间是互斥关系。同理消费者也是如此。

        这就需要加上互斥锁来保证每次只有一个线程来操作。

                生产者

        int pthread\_mutex\_lock(pthread\_mutex\_t \*mutex);

            生产

        int pthread\_cond\_signal(pthread\_cond\_t \*cond);

        int pthread\_mutex\_unlock(pthread\_mutex\_t \*mutex);

                消费者

        int pthread\_mutex\_lock(pthread\_mutex\_t \*mutex);

        if (没有可消费)

            int pthread\_cond\_wait(pthread\_cond\_t \*cond, pthread\_mutex\_t \*mutex);

        else

            消费

        int pthread\_mutex\_unlock(pthread\_mutex\_t \*mutex);

        分析：如果此时消费者发现没有可以消费的，于是就在等待生产者生产，但是此时生产者不能生产(ps:此时生产者由于加锁，进不去)。

        于是就产生了死锁。而pthread\_cond\_wait的第二个参数是个互斥锁，可以将生产者的锁解开。

    规范写法

        pthread\_mutex\_lock();

        while (条件不满足)

            pthread\_cond\_wait();

        pthread\_mutex\_unlock();

            分析为什么用while：这是因为wait函数是个阻塞函数，很有可能被信号打断，而wait函数返回有两种情况，一种是被signal函数通知，

        一种是被别的信号打断，设置成while后，如果是被信号打断，那么再次进入条件判断时，仍然是不满足的，不会执行下面的操作，

        继续等待。这种信号打断的方式也叫作假唤醒。

        pthread\_mutex\_lock();

        pthread\_cond\_signal();  // 如果没有线程等待，也就是说条件一直满足，那么信号将被丢弃。

            eg：假设条件是大于0，那么一开始大小为4时，满足条件，信号就不需要激活条件，执行完一次操作后，变成3仍然满足条件，信号

            也会被丢弃。

        pthread\_mutex\_unlock();

    生产者-消费者模型

            int \*p = malloc(sizeof(int));

            \*p = i;

            pthread\_create(&tid[i], NULL, producer, p);

        在线程中切记要释放掉开辟的堆内存

            int id = \*(int \*)arg;

            free(arg);

信号量    #include <semaphore.h>

    1.定义信号量 sem\_t sem;

    2.初始化信号量 int sem\_init(sem\_t \*sem, int pshared, unsigned int value);

                a. sem地址    b. 0表示本进程中多个线程间同步，非0表示可以跨进程的同步操作。c. 信号量初值(计数器的值)

    3.PV操作

        P   int sem\_wait(sem\_t \*sem);   // sem-1 如果小于0就阻塞

        V    int sem\_post(sem\_t \*sem);   // sem+1

    4.销毁 int sem\_destroy(sem\_t \*sem);

    仓库存放东西有限

        生产者

        while (1)

        {

            pthread\_mutex\_lock();

            sem\_wait(sem\_full);

            生产

            sem\_post(sem\_empty);

            pthread\_mutex\_unlock();

        }

        消费者

        while (1)

        {

            pthread\_mutex\_lock();

            sem\_wait(sem\_empty);

            消费

            sem\_post(sem\_full);

            pthread\_mutex\_unlock();

        }

    场景：http服务器是基于短连接的，那么在网页上的访问就会频繁地创建和销毁线程，这样开销是很大的。

线程池(生产者消费者模型)

    根据系统性能创建合适大小的线程池。

    将创建的线程存入在线程池中，需要线程时，就让它们建立联系，不想用了，就断开联系。

        1.线程池中有若干个线程

        2.用于执行大量相对短暂的任务

    计算密集型任务：大量的时间占用CPU进行运算。

        线程池中线程的个数等于CPU个数，避免线程切换。

    IO密集型任务：大量的时间占用CPU在阻塞等待IO。

        线程池中线程个数大于CPU个数。

    当任务增加时，可以动态增加线程池中线程的个数。当任务执行完成后，可以动态减少线程池中线程个数。

    需求：

        生产者线程向任务队列中添加任务，任务队列中有任务，如果线程池中有等待线程就唤醒它并执行任务，如果线程池中没有

    等待线程，并且没有达到上限，就添加新的线程到线程池。

        没有任务执行就等待，有任务则去执行。

        typedef struct condition

        {

            pthread\_mutex\_t mutex;

            pthread\_cond\_t cond;

        }condition\_t;

        // 任务队列

        typedef struct task

        {

            void \*(pfun)(void \*); // 任务队列的回调函数(管理任务的)

            void \*arg; // 回调函数传入的参数

            struct task \*next;

        }task\_t;

        typedef struct threadpool

        {

            condition\_t cond; // 同步与互斥

            task\_t \*first; // 任务队列的队头

            task\_t \*tail; // 任务队列的队尾

            int max\_thread; // 最大线程个数

            int idle; // 空闲线程个数   如果有空闲线程，此时可以signal通知下

            int counter; // 当前线程池中的线程个数

            int quit;    // 如果为1表示退出，为0表示不退出

        }threadpool\_t;

        // 初始化

        void threadpool\_init(threadpool\_t \*pool, int max);

        // 往线程池添加任务

        void threadpool\_add(threadpool\_t \*pool, void\*(\*pf)(void\*), void \*arg);

        // 销毁线程池

        void threadpool\_destroy(threadpool\_t \*pool);

        pthread\_cond\_broadcast();   // 唤醒所有

        pthread\_cond\_signal();   // 只唤醒一个

练习2: 了解基于信号量和管程的哲学家就餐问题

1. 说明ucore中基于信号量的哲学家就餐问题的实现机制。

信号量的核心用以下的简短代码可以大致表示：

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 \*/;

}

}

当多个 (>1)进程可以进行互斥或同步合作时，一个进程会由于无法满足信号量设置的某条件而在某一位置停止,直到它接收到一个特定的信号(表明条件满足了)。为了发信号，需要使用一个称作信号量的特殊变量。为通过信号量s传送信号,信号量的V操作采用进程可执行原语semSignal(s)； 为通过信号量s接收信号， 信号量的P操作采用进程可执行原语semWait(s)； 如果相应的信号仍然没有发送，则进程被阻塞或睡眠，直到发送完为止 。

首先定义了一些数据结构，包括信号量、等待队列等

// 定义信号量的数据结构

typedef struct {

int value; //信号量的当前值

wait\_queue\_t wait\_queue; //信号量对应的等待队列

} semaphore\_t;

// 用于等待队列，存放了当前等待的线程的PCB 和 唤醒原因 和 等待队列 和 用于还原结构体的等待队列标志

typedef struct {

struct proc\_struct \*proc; //等待进程的指针

uint32\_t wakeup\_flags; //进程被放入等待队列的原因标记

wait\_queue\_t \*wait\_queue; //指向此wait结构所属于的wait\_queue

list\_entry\_t wait\_link; //用来组织wait\_queue中wait节点的连接

} wait\_t;

信号量的计数器value 含义如下：

value>0， 表示共享资源的空闲数

vlaue<0， 表示该信号量的等待队列里的进程数

value=0， 表示等待队列为空

之后有相应的有关信号量的操作函数：

初始化函数 sem\_init:

// 对信号量进行初始化的函数

void sem\_init(semaphore\_t \*sem, int value)

{

//将value设为特定值

sem->value = value;

//将等待队列初始化

wait\_queue\_init(&(sem->wait\_queue));

}

实现P操作，请求一个信号量对应的资源：

//表示请求一个该信号量对应的资源

static \_\_noinline uint32\_t \_\_down(semaphore\_t \*sem, uint32\_t wait\_state) {

bool intr\_flag;

local\_intr\_save(intr\_flag);

//查询整型变量来了解是否存在多余的可分配的资源，

//如果有多余可分配资源，取出资源（整型变量减1），之后当前进程便可以正常进行；

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);

//如果没有可用的资源，整型变量不是正数，当前进程的资源需求得不到满足，

//因此将其状态改为SLEEPING态，然后将其挂到对应信号量的等待队列中，

wait\_t \_\_wait, \*wait = &\_\_wait;

wait\_current\_set(&(sem->wait\_queue), wait, wait\_state);

local\_intr\_restore(intr\_flag);

//调用schedule函数来让出CPU

schedule();

local\_intr\_save(intr\_flag);

//在资源得到满足，重新被唤醒之后，将自身从等待队列上删除掉；

wait\_current\_del(&(sem->wait\_queue), wait);

local\_intr\_restore(intr\_flag);

if (wait->wakeup\_flags != wait\_state) {

return wait->wakeup\_flags;

}

return 0;

}

实现V操作，释放一个信号量资源：

//表示释放一个该信号量对应的资源，

static \_\_noinline void \_\_up(semaphore\_t \*sem, uint32\_t wait\_state)

{

bool intr\_flag;

local\_intr\_save(intr\_flag);

{

wait\_t \*wait;

////查询等待队列是否为空，如果是空的话，给整型变量加1；

if ((wait = wait\_queue\_first(&(sem->wait\_queue))) == NULL) {

sem->value ++;

}

//如果等待队列非空，取出其中的一个进程唤醒；

else

{

assert(wait->proc->wait\_state == wait\_state);

wakeup\_wait(&(sem->wait\_queue), wait, wait\_state, 1);

}

}

local\_intr\_restore(intr\_flag);

}

1. 说明ucore中基于管程的哲学家就餐问题的实现机制。

实现管程所构建的数据结构 monitor 以及 condvar

// 管程的数据结构

typedef struct monitor{

semaphore\_t mutex; // 二值信号量 用来互斥访问管程

semaphore\_t next; // 用于条件同步 用于发出signal操作的进程等条件为真之前进入睡眠

int next\_count; // 记录睡在 signal 操作的进程数

condvar\_t \*cv; // 条件变量

} monitor\_t;

// 条件变量数据结构

typedef struct condvar{

semaphore\_t sem; // 用于条件同步 用于发出wait操作的进程等待条件为真之前进入睡眠

int count; // 记录睡在 wait 操作的进程数(等待条件变量成真)

monitor\_t \* owner; // 所属管程

} condvar\_t;

条件变量机制的实现主要是cond\_signal, cond\_wait两个函数中，分别表示提醒等待在这个条件变量上的进程恢复执行，以及等待在这个条件变量上，直到有其他进行将其唤醒为止

cond\_signal: 将指定条件变量上等待队列中的一个线程进行唤醒，并且将控制权转交给这个进程。该操作实现了对共享变量访问的互斥性；

void

cond\_signal (condvar\_t \*cvp) {

//LAB7 EXERCISE1: YOUR CODE

cprintf("cond\_signal begin: cvp %x, cvp->count %d, cvp->owner->next\_count %d\n", cvp, cvp->count, cvp->owner->next\_count);

//判断当前的条件变量的等待队列上是否有正在等待的进程，如果没有则不需要进行任何操作；

//如果有正在等待的进程

if(cvp->count>0)

{

//将其中的一个唤醒

up(&(cvp->sem));

//所属管程的next计数加1，表示当前进程会被等待者堵塞

cvp->owner->next\_count ++;

//阻塞，等待条件同步

down(&(cvp->owner->next));

//当前进程被唤醒，恢复next上的等待进程计数

cvp->owner->next\_count --;

}

cprintf("cond\_signal end: cvp %x, cvp->count %d, cvp->owner->next\_count %d\n", cvp, cvp->count, cvp->owner->next\_count);

}

总的来说：执行cond\_signal函数时，首先当前进程测试cv.count，如果不大于0，则表示当前没有执行cond\_wait而进入休眠状态的进程，函数直接返回。如果cv.count大于0，这表示当前有执行cond\_wait而进入休眠状态的进程，因此需要唤醒在cv.sem上等待出入休眠状态的进程。由于只允许一个进程在管程中执行，所以一旦当前进程唤醒了其他进程，自身就要进入休眠状态，即增加monitor.next\_count，让当前进程在信号量monitor.next上进入休眠状态，直到被唤醒时，减少monitor.next\_count。

cond\_wait：该函数的功能为将当前进程等待在指定信号量上，其操作过程为将等待队列的计数加1，然后释放管程的锁或者唤醒一个next上的进程来释放锁，然后把自己等在条件变量的等待队列上，直到有signal信号将其唤醒，正常退出函数；

void

cond\_wait (condvar\_t \*cvp) {

//LAB7 EXERCISE1: YOUR CODE

cprintf("cond\_wait begin: cvp %x, cvp->count %d, cvp->owner->next\_count %d\n", cvp, cvp->count, cvp->owner->next\_count);

cvp->count ++; // 修改等待在条件变量的等待队列上的进程计数

//当管程的 next\_count 大于0，说明有进程睡在了 signal 操作上我们将其唤醒

if (cvp->owner->next\_count > 0)

{ // 释放锁

up(&cvp->owner->next);

}

else //当前没有进程睡在 signal操作数 只需要释放互斥体

{

up(&cvp->owner->mutex);

}

//将自身阻塞，等待条件变量的条件为真，被唤醒后将条件不成立而睡眠的进程计数减1

down(&cvp->sem); // 将自己等待在条件变量上

cvp->count --; // 被唤醒，修正等待队列上的进程计数

cprintf("cond\_wait end: cvp %x, cvp->count %d, cvp->owner->next\_count %d\n", cvp, cvp->count, cvp->owner->next\_count);

}

因此，若某个进程执行了cond\_wait函数，表明该进程需因要的某个条件变量不满足需求而进入休眠状态。等待这个条件变量的休眠进程数cv.count增加。此时如果monitor.next\_count大于0，表示至少有1个进程执行cond\_signal函数后进入休眠状态，等待monitor.next信号量时，则需要唤醒等待该条件量的另一个进程，然后该进程在cv.sem上休眠。当进程A被唤醒时，减少cv.count，表示等待此条件变量的睡眠进程个数减少了一个，可继续执行。如果monitor.next\_count小于等于0，表明目前没有进程执行cond\_signal函数进入休眠，则需要唤醒的是由于互斥条件限制而无法进入管程的进程，即唤醒在monitor.mutex上休眠的进程。然后当前进程在cv.sem上休眠，直到被唤醒时，减少cv.count，表示等待此条件的睡眠进程个数减少，可继续执行。

实现上述函数后，管程基本已经实现，接下来分析本实验中基于条件变量和管程的哲学家就餐问题的实现:

实现哲学家问题主要在check\_sync.c中

这里创建了5个线程表示5个哲学家，哲学家尝试4次思考->拿叉子->吃饭->放下叉子。

具体分析实现:主要是phi\_take\_forks\_condvar函数以及phi\_put\_forks\_condvar函数

phi\_take\_forks\_condvar函数表示指定的哲学家尝试获得自己所需要进餐的两把叉子，如果不能获得则阻塞，具体实现流程为：

给管程上锁，将哲学家的状态修改为HUNGER；

判断相邻的哲学家是否正在进餐；

如果能够进餐，将自己的状态修改成EATING，然后释放锁，离开管程即可；

如果不能进餐，等待在自己对应的条件变量上，等待相邻的哲学家释放资源的时候将自己唤醒；

void phi\_take\_forks\_condvar(int i) {

//通过P操作进入临界区

down(&(mtp->mutex));

//记录下哲学家i是否饥饿，即处于等待状态拿叉子

state\_condvar[i]=HUNGRY;

phi\_test\_condvar(i);

while (state\_condvar[i] != EATING)

{

cprintf("phi\_take\_forks\_condvar: %d didn't get fork and will wait\n",i);

cond\_wait(&mtp->cv[i]);//如果得不到叉子就睡眠

}

//如果存在睡眠的进程则那么将之唤醒

if(mtp->next\_count>0)

up(&(mtp->next));

else

up(&(mtp->mutex));

}

phi\_put\_forks\_condvar函数表示释放当前哲学家占用的叉子，并且唤醒相邻的因为得不到资源而进入等待的哲学家：

首先获取管程的锁，将自己的状态修改成THINKING；

检查相邻的哲学家是否在自己释放了叉子的占用之后满足了进餐的条件，如果满足，将其从等待中唤醒

释放锁，离开管程

void phi\_put\_forks\_condvar(int i) {

//通过P操作进入临界区

down(&(mtp->mutex));

//记录进餐结束的状态

state\_condvar[i]=THINKING;

//看一下左边哲学家现在是否能进餐

phi\_test\_condvar(LEFT);

//看一下右边哲学家现在是否能进餐

phi\_test\_condvar(RIGHT);

//如果有哲学家睡眠就予以唤醒

if(mtp->next\_count>0)

up(&(mtp->next));

else

up(&(mtp->mutex));

}

由于每个哲学家只可能占有所有需要的资源或者完全不占用资源，因此不会出现部分占有资源的现象，从而避免了死锁的产生；最终必定所有哲学将都能成功就餐