实验4­­——同步互斥

班级：硬件一班 姓名：于华宇 学号：171491120

## 预备知识

**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。

## 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？

设计必要性：由于有处理器调度的存在,且进程在访问某类资源暂时无法满足的情况下,进程会进入等待状态。这导致了多进程执行时序的不确定性和潜在执行结果的不确定性。

根据操作系统原理的知识,我们知道如果没有在硬件级保证读内存-修改值-写回内存的原子性,我们只能通过复杂的软件来实现同步互斥操作。但由于有定时器、屏蔽/使能中断、等待队列wait\_queue支持test\_and\_set\_bit等原子操作机器指令(在本次实验中没有用到)的存在,使得我们在实现进程等待、同步互斥上得到了极大的简化。

实验中实现的定时器、屏蔽/使能中断和等待队列。

* 定时器

在ucore中,时钟(timer)中断给操作系统提供了有一定间隔的时间事件,操作系统将其作为基本的调度和计时单位(我们记两次时间中断之间的时间间隔为一个时间片,timer splice)。

基于此时间单位,操作系统得以向上提供基于时间点的事件,并实现基于时间长度的睡眠等待和唤醒机制。在每个时钟中断发生时,操作系统产生对应的时间事件。应用程序或者操作系统的其他组件可以以此来构建更复杂和高级的进程管理和调度算法。

sched.h, sched.c定义的有关timer的各种相关接口主要包括:

typedef struct {......} timer\_t: 定义了timer\_t的基本结构,其可以用sched.h中的

timer\_init函数对其进行初始化。

typedef struct {

unsigned int expires;

struct proc\_struct \*proc;

list\_entry\_t timer\_link;

} timer\_t;

static inline timer\_t \* timer\_init(timer\_t \*timer, struct proc\_struct \*proc, int expires):对某定时器进行初始化,让它在expires时间片之后唤醒proc进程。

static inline timer\_t \* timer\_init(timer\_t \*timer, struct proc\_struct \*proc, int expires) {

timer->expires = expires;

timer->proc = proc;

list\_init(&(timer->timer\_link));

return timer;

}

void add\_timer(timer t \*timer): 向系统添加某个初始化过的timer\_t,该定时器在指定时间后被激活,并将对应的进程唤醒至runnable(如果当前进程处在等待状态)。

void add\_timer(timer\_t \*timer) {

bool intr\_flag;

local\_intr\_save(intr\_flag);

{

assert(timer->expires > 0 && timer->proc != NULL);

assert(list\_empty(&(timer->timer\_link)));

list\_entry\_t \* le = list\_next(&timer\_list);

while (le != &timer\_list) {

timer\_t \* next = le2timer(le, timer\_link);

if (timer->expires < next->expires) {

next->expires -= timer->expires;

break;

}

timer->expires -= next->expires;

le = list\_next(le);

}

list\_add\_before(le, &(timer->timer\_link));

}

local\_intr\_restore(intr\_flag);

}

void del\_timer(timer\_t \*time):向系统删除(或者说取消)某一个定时器。该定时器在取消后不会被系统激活并唤醒进程。

void del\_timer(timer\_t \*timer) {

bool intr\_flag;

local\_intr\_save(intr\_flag);

{

if (!list\_empty(&(timer->timer\_link))) {

if (timer->expires != 0) {

list\_entry\_t \*le = list\_next(&(timer->timer\_link));

if (le != &timer\_list) {

timer\_t \*next = le2timer(le, timer\_link);

next->expires += timer->expires;

}

}

list\_del\_init(&(timer->timer\_link));

}

}

local\_intr\_restore(intr\_flag);

}

void run\_timer\_list(void):更新当前系统时间点,遍历当前所有处在系统管理内的定时器,找出所有应该激活的计数器,并激活它们。该过程在且只在每次定时器中断时被调用。在ucore中,其还会调用调度器事件处理程序。

void run\_timer\_list(void) {

bool intr\_flag;

local\_intr\_save(intr\_flag);

{

list\_entry\_t \*le = list\_next(&timer\_list);

if (le != &timer\_list) {

timer\_t \*timer = le2timer(le, timer\_link);

assert(timer->expires != 0);

timer->expires --;

while (timer->expires == 0) {

le = list\_next(le);

struct proc\_struct \*proc = timer->proc;

if (proc->wait\_state != 0) {

assert(proc->wait\_state & WT\_INTERRUPTED);

}

else {

warn("process %d's wait\_state == 0.\n", proc->pid);

}

wakeup\_proc(proc);

del\_timer(timer);

if (le == &timer\_list) {

break;

}

timer = le2timer(le, timer\_link);

}

}

sched\_class\_proc\_tick(current);

}

local\_intr\_restore(intr\_flag);

}

一个timer\_t 在系统中的存活周期可以被描述如下:

1. timer\_t 在某个位置被创建和初始化,并通过 add\_timer加入系统管理列表中

2. 系统时间被不断累加,直到 run\_timer\_list 发现该 timer\_t到期。

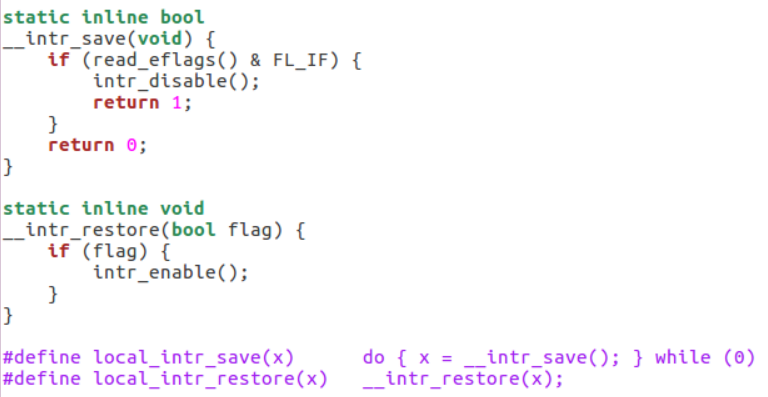
3. run\_timer\_list更改对应的进程状态,并从系统管理列表中移除该timer\_t。

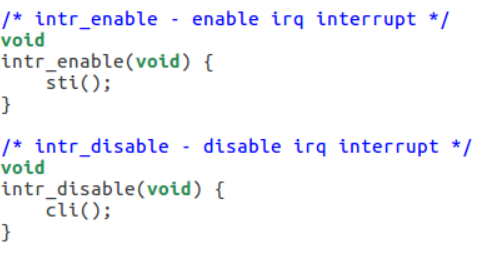
* 屏蔽与使能中断

在ucore中提供的底层机制包括中断屏蔽/使能控制等。kern/sync.c中实现的开关中断的控制函数local\_intr\_save(x)和local\_intr\_restore(x),它们是基于kern/driver文件下的intr\_enable()、intr\_disable()函数实现的。具体调用关系为:

关中断:local\_intr\_save --> \_\_intr\_save --> intr\_disable --> cli

开中断:local\_intr\_restore--> \_\_intr\_restore --> intr\_enable --> sti





最终的cli和sti是x86的机器指令,最终实现了关(屏蔽)中断和开(使能)中断,即设置了eflags寄存器中与中断相关的位。通过关闭中断,可以防止对当前执行的控制流被其他中断事件处理所打断。既然不能中断,那也就意味着在内核运行的当前进程无法被打断或被重新调度,即实现了对临界区的互斥操作。所以在单处理器情况下,可以通过开关中断实现对临界区的互斥保护,需要互斥的临界区代码的一般写法为:

local\_intr\_save(intr\_flag);

{

临界区代码

}

local\_intr\_restore(intr\_flag);

......

由于目前ucore只实现了对单处理器的支持,所以通过这种方式,就可简单地支撑互斥操作了。在多处理器情况下,这种方法是无法实现互斥的,因为屏蔽了一个CPU的中断,只能阻止本地CPU上的进程不会被中断或调度,并不意味着其他CPU上执行的进程不能执行临界区的代码。所以,开关中断只对单处理器下的互斥操作起作用。在本实验中,开关中断机制是实现信号量等高层同步互斥原语的底层支撑基础之一。

* 等待队列

用户进程或内核线程可以转入等待状态以等待某个特定事件,当该事件发生时这些进程能够被再次唤醒。内核实现这一功能的一个底层支撑机制就是等待队列wait\_queue,等待队列和每一个事件(睡眠结束、时钟到达、任务完成、资源可用等)联系起来。需要等待事件的进程在转入休眠状态后插入到等待队列中。当事件发生之后,内核遍历相应等待队列,唤醒休眠的用户进程或内核线程,并设置其状态为就绪状态(PROC\_RUNNABLE),并将该进程从等待队列中清除。

ucore在kern/sync/{wait.h,wait.c }中实现了等待项wait结构和等待队列wait queue结构以及相关函数),这是实现ucore中的信号量机制和条件变量机制的基础。进入wait queue的进程会被设为等待状态(PROC\_SLEEPING),直到他们被唤醒。

**数据结构：**

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;

typedef struct {

list\_entry\_t wait\_head; //wait\_queue的队头

}wait\_queue\_t;

le2wait(le, member) ； //实现wait\_t中成员的指针向wait\_t 指针的转化

**相关函数：**

与wait和wait queue相关的函数主要分为两层,底层函数是对wait queue的初始化、插入、删除和查找操作,相关函数如下:

void wait\_init(wait\_t \*wait, struct proc\_struct \*proc); //初始化wait结构

bool wait\_in\_queue(wait\_t \*wait); //wait是否在wait queue中

void wait\_queue\_init(wait\_queue\_t \*queue); //初始化wait\_queue结构

void wait\_queue\_add(wait\_queue\_t \*queue, wait\_t \*wait); //把wait前插到wait queue中

void wait\_queue\_del(wait\_queue\_t \*queue, wait\_t \*wait); //从wait queue中删除wait

wait\_t \*wait\_queue\_next(wait\_queue\_t \*queue, wait\_t \*wait);//取得wait的后一个链接指针

wait\_t \*wait\_queue\_prev(wait\_queue\_t \*queue, wait\_t \*wait);//取得wait的前一个链接指针

wait\_t \*wait\_queue\_first(wait\_queue\_t \*queue); //取得wait queue的第一个wait

wait\_t \*wait\_queue\_last(wait\_queue\_t \*queue); //取得wait queue的最后一个wait

bool wait\_queue\_empty(wait\_queue\_t \*queue); //wait queue是否为空

高层函数基于底层函数实现了让进程进入等待队列-- wait\_current\_set,以及从等待队列中唤醒进程-- wakeup\_wait ,相关函数如下:

//让wait与进程关联,且让当前进程关联的wait进入等待队列queue,当前进程睡眠

void wait\_current\_set(wait\_queue\_t \*queue, wait\_t \*wait, uint32\_t wait\_state);

//把与当前进程关联的wait从等待队列queue中删除

wait\_current\_del(queue, wait);

//唤醒与wait关联的进程

void wakeup\_wait(wait\_queue\_t \*queue, wait\_t \*wait, uint32\_t wakeup\_flags, bool del);

//唤醒等待队列上挂着的第一个wait所关联的进程

void wakeup\_first(wait\_queue\_t \*queue, uint32\_t wakeup\_flags, bool del);

//唤醒等待队列上所有的等待的进程

void wakeup\_queue(wait\_queue\_t \*queue, uint32\_t wakeup\_flags, bool del);

对于唤醒进程的函数wakeup\_wait,可以看到它会被各种信号量的V操作函数up 调用,并且它会调用wait\_queue\_del函数和wakup\_proc函数来完成唤醒进程的操作。

对于让进程进入等待状态的函数wait\_current\_set,可以看到它会被各种信号量的P操作函数`down调用,并且它会调用wait\_init完成对等待项的初始化,并进一步调用 wait\_queue\_add`来把与要处于等待状态的进程所关联的等待项挂到与信号量绑定的等待队列中。

1. 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。

信号量基本知识：PV操作是一种实现进程互斥与同步的有效方法。PV操作与信号量的处理相关，P表示通过的意思，V表示释放的意思。

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。

ucore中信号量参照上述原理描述,建立在开关中断机制和wait\_queue的基础上进行了具体实现。信号量的数据结构定义如下:

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;

semaphore\_t是最基本的记录型信号量(record semaphore)结构,包含了用于计数的整数值value,和一个进程等待队列wait\_queue,一个等待的进程会挂在此等待队列上。

在ucore中最重要的信号量操作是P操作函数down(semaphore\_t \*sem)和V操作函数up(semaphore\_t \*sem)。但这两个函数的具体实现是\_\_down(semaphore\_t \*sem, uint32\_t

wait\_state)函数和\_\_up(semaphore\_t \*sem, uint32\_t wait\_state)函数,二者的具体实现描述。

如下:

\_\_down(semaphore\_t \*sem, uint32\_t wait\_state, timer\_t \*timer):具体实现信号量的**P操作**,首先关掉中断,然后判断当前信号量的value是否大于0。如果是>0,则表明可以获得信号量,故让value减一,并打开中断返回即可;如果不是>0,则表明无法获得信号量,故需要将当前的进程加入到等待队列中,并打开中断,然后运行调度器选择另外一个进程执行。如果被V操作唤醒,则把自身关联的wait从等待队列中删除(**此过程需要先关中断,完成后开中断**)。

\_\_up(semaphore\_t \*sem, uint32\_t wait\_state):具体实现信号量的**V操作**,首先关中断,如果信号量对应的wait queue中没有进程在等待,直接把信号量的value加一,然后开中断返

回;如果有进程在等待且进程等待的原因是semophore设置的,则调用wakeup\_wait函数将

waitqueue中等待的第一个wait删除,且把此wait关联的进程唤醒,最后开中断返回(**此过程需要先关中断,完成后开中断**)。

比较后得出，Ucore信号量的具体实现采用了关中断的方式保证了对共享资源的互斥访问,通过等待队列让无法获得信号量的进程睡眠等待。

1. Ucore中的管程机制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。

在单处理器情况下,将会导致所有其它进程都无法进入临界区使得该条件Cond为真,该管程的执行将会发生死锁。为此,可引入条件变量(Condition Variables,简称CV)。

**条件变量CV**：可理解为一个进程的等待队列,队列中的进程正等待某个条件Cond变为真。每个条件变量关联着一个条件,如果条件Cond不为真,则进程需要等待,如果条件Cond为真,则进程可以进一步在管程中执行。需要注意当一个进程等待一个条件变量CV(即等待Cond为真),该进程需要退出管程,这样才能让其它进程可以进入该管程执行,并进行相关操作。

**管程数据结构**：

typedef struct monitor{

semaphore\_t mutex; //the mutex lock for going into the routines in monitor,

should be initialized to 1

// the next semaphore is used to

// (1) procs which call cond\_signal funciton should DOWN next sema after UP cv.

sema

//OR (2) procs which call cond\_wait funciton should UP next sema before DOWN cv.s ema

semaphore\_t next;

int next\_count; // the number of of sleeped procs which cond\_signal funcit on

condvar\_t \*cv; // the condvars in monitor

} monitor\_t;

管程中的条件变量的数据结构condvar\_t定义如下:

typedef struct condvar{

semaphore\_t sem; // the sem semaphore is used to down the waiting proc, and the signaling proc should up the waiting proc

int count; // the number of waiters on condvar

monitor\_t \* owner; // the owner(monitor) of this condvar

} condvar\_t;

ucore设计实现了条件变量 wait\_cv操作和signal\_cv操作对应的具体函数,即 cond\_wait函数和cond\_signal函数,此外还有cond\_init初始化函数。

**// Suspend calling thread on a condition variable waiting for condition Atomically unlocks**

**// mutex and suspends calling thread on conditional variable after waking up locks mutex. Notice: mp is mutex semaphore for monitor's procedures**

**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);**

**/\***

**\* cv.count ++;**

**\* if(mt.next\_count>0)**

**\* signal(mt.next)**

**\* else**

**\* signal(mt.mutex);**

**\* wait(cv.sem);**

**\* cv.count --;**

**\*/**

**cvp->count++;**

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

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

**else**

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

**down(&(cvp->sem));**

**cvp->count --;**

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

**}**

**// Unlock one of threads waiting on the condition variable.**

**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);**

**/\***

**\* cond\_signal(cv) {**

**\* if(cv.count>0) {**

**\* mt.next\_count ++;**

**\* signal(cv.sem);**

**\* wait(mt.next);**

**\* mt.next\_count--;**

**\* }**

**\* }**

**\*/**

**if(cvp->count>0) {**

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

**up(&(cvp->sem));**

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

**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);**

**}**

**// Initialize monitor.**

**void monitor\_init (monitor\_t \* mtp, size\_t num\_cv) {**

**int i;**

**assert(num\_cv>0);**

**mtp->next\_count = 0;**

**mtp->cv = NULL;**

**sem\_init(&(mtp->mutex), 1); //unlocked**

**sem\_init(&(mtp->next), 0);**

**mtp->cv =(condvar\_t \*) kmalloc(sizeof(condvar\_t)\*num\_cv);**

**assert(mtp->cv!=NULL);**

**for(i=0; i<num\_cv; i++){**

**mtp->cv[i].count=0;**

**sem\_init(&(mtp->cv[i].sem),0);**

**mtp->cv[i].owner=mtp;**

**}**

**}**

## 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。

//---------- philosophers problem using semaphore ----------------------

int state\_sema[N]; /\* 记录每个人状态的数组 \*/

/\* 信号量是一个特殊的整型变量 \*/

semaphore\_t mutex; /\* 临界区互斥 \*/

semaphore\_t s[N]; /\* 每个哲学家一个信号量 \*/

struct proc\_struct \*philosopher\_proc\_sema[N];

void phi\_test\_sema(i) /\* i：哲学家号码从0到N-1 \*/

{

if(state\_sema[i]==HUNGRY&&state\_sema[LEFT]!=EATING

&&state\_sema[RIGHT]!=EATING)

{

state\_sema[i]=EATING;

up(&s[i]);

}

}

void phi\_take\_forks\_sema(int i) /\* i：哲学家号码从0到N-1 \*/

{

down(&mutex); /\* 进入临界区 \*/

state\_sema[i]=HUNGRY; /\* 记录下哲学家i饥饿的事实 \*/

phi\_test\_sema(i); /\* 试图得到两只叉子 \*/

up(&mutex); /\* 离开临界区 \*/

down(&s[i]); /\* 如果得不到叉子就阻塞 \*/

}

void phi\_put\_forks\_sema(int i) /\* i：哲学家号码从0到N-1 \*/

{

down(&mutex); /\* 进入临界区 \*/

state\_sema[i]=THINKING; /\* 哲学家进餐结束 \*/

phi\_test\_sema(LEFT); /\* 看一下左邻居现在是否能进餐 \*/

phi\_test\_sema(RIGHT); /\* 看一下右邻居现在是否能进餐 \*/

up(&mutex); /\* 离开临界区 \*/

}

int philosopher\_using\_semaphore(void \* arg) /\* i：哲学家号码，从0到N-1 \*/

{

int i, iter=0;

i=(int)arg;

cprintf("I am No.%d philosopher\_sema\n",i);

while(iter++<TIMES)

{ /\* 无限循环 \*/

cprintf("Iter %d, No.%d philosopher\_sema is thinking\n",iter,i); /\* 哲学家正在思考 \*/

do\_sleep(SLEEP\_TIME);

phi\_take\_forks\_sema(i);

/\* 需要两只叉子，或者阻塞 \*/

cprintf("Iter %d, No.%d philosopher\_sema is eating\n",iter,i); /\* 进餐 \*/

do\_sleep(SLEEP\_TIME);

phi\_put\_forks\_sema(i);

/\* 把两把叉子同时放回桌子 \*/

}

cprintf("No.%d philosopher\_sema quit\n",i);

return 0;

}

1. 说明ucore中基于管程的哲学家就餐问题的实现机制。

monitor dp

{

enum {THINKING, HUNGRY, EATING} state[5];

condition self[5];

void pickup(int i) {

state[i] = HUNGRY; //轮到i吃饭时，将i状态设置为饥饿

test(i); //检查是否可以吃饭

if (state[i] != EATING) //如果i不可以吃饭，则i进入等待队列

self[i].wait\_cv();

}

void putdown(int i) { //放下餐具，状态变回思考

state[i] = THINKING; //检查左右的人是否需要吃饭

test((i+4) %5);

test((i+1) %5);

}

void test(int i) {

if ((state[(i+4) %5]!= EATING) &&(state[i] ==HUNGRY) &&

(state[(i+1) % 5]!= EATING)) { //i饥饿，且左右都没有在吃

state[i] = EATING; //i状态改为吃

self[i].signal\_cv(); //将i从等待队列中移出

}

} //若i不饥饿或左右有人在吃，则i状态不变

initialization code() {

for (int i = 0; i < 5; i++)

state[i] = THINKING;

}

}