练习1：了解信号量和管理的实现机制

1. 同步互斥的底层支持是如何实现的？
2. 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。但由于有开关中断和test\_and\_set\_bit等原子操作机器指令的存在，使得我们在实现同步互斥原语上可以大大简化。在atomic.c文件中实现的test\_and\_set\_bit等原子操作。
3. 在ucore中提供的底层机制包括中断开关控制和test\_and\_set相关原子操作机器指令。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：
4. 关中断：local\_intr\_save --> \_\_intr\_save --> intr\_disable --> cli
5. 开中断：local\_intr\_restore--> \_\_intr\_restore --> intr\_enable --> sti
6. 最终的cli和sti是x86的机器指令，最终实现了关中断和开中断，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从新调度，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：
7. local\_intr\_save(intr\_flag);
8. {
9. 临界区代码
10. }
11. local\_intr\_restore(intr\_flag);
12. ……
13. 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。
14. **等待队列**
15. 到目前为止，我们的实验中，用户进程或内核线程还没有睡眠的支持机制。在课程中提到用户进程或内核线程可以转入休眠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（runnable state），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wait结构和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的进程会被设为睡眠状态，直到他们被唤醒。
16. typedef struct {
17. struct proc\_struct \*proc; //等待进程的指针
18. uint32\_t wakeup\_flags; //进程被放入等待队列的原因标记
19. wait\_queue\_t \*wait\_queue; //指向此wait结构所属于的wait\_queue
20. list\_entry\_t wait\_link; //用来组织wait\_queue中wait节点的连接
21. } wait\_t;
22. typedef struct {
23. list\_entry\_t wait\_head; //wait\_queue的队头
24. } wait\_queue\_t;
25. le2wait(le, member) //实现wait\_t中成员的指针向wait\_t 指针的转化
26. 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：
27. void wait\_init(wait\_t \*wait, struct proc\_struct \*proc); //初始化wait结构
28. bool wait\_in\_queue(wait\_t \*wait); //wait是否在wait queue中
29. void wait\_queue\_init(wait\_queue\_t \*queue); //初始化wait\_queue结构
30. void wait\_queue\_add(wait\_queue\_t \*queue, wait\_t \*wait); //把wait前插到wait queue中
31. void wait\_queue\_del(wait\_queue\_t \*queue, wait\_t \*wait); //从wait queue中删除wait
32. wait\_t \*wait\_queue\_next(wait\_queue\_t \*queue, wait\_t \*wait);//取得wait的后一个链接指针
33. wait\_t \*wait\_queue\_prev(wait\_queue\_t \*queue, wait\_t \*wait);//取得wait的前一个链接指针
34. wait\_t \*wait\_queue\_first(wait\_queue\_t \*queue); //取得wait queue的第一个wait
35. wait\_t \*wait\_queue\_last(wait\_queue\_t \*queue); //取得wait queue的最后一个wait
36. bool wait\_queue\_empty(wait\_queue\_t \*queue); //wait queue是否为空
37. 高层函数基于底层函数实现了让进程进入等待队列，以及从等待队列中唤醒进程，相关函数如下：
38. //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
39. void wait\_current\_set(wait\_queue\_t \*queue, wait\_t \*wait, uint32\_t wait\_state);
40. //把与当前进程关联的wait从等待队列queue中删除
41. wait\_current\_del(queue, wait);
42. //唤醒与wait关联的进程
43. void wakeup\_wait(wait\_queue\_t \*queue, wait\_t \*wait, uint32\_t wakeup\_flags, bool del);
44. //唤醒等待队列上挂着的第一个wait所关联的进程
45. void wakeup\_first(wait\_queue\_t \*queue, uint32\_t wakeup\_flags, bool del);
46. //唤醒等待队列上所有的等待的进程
47. void wakeup\_queue(wait\_queue\_t \*queue, uint32\_t wakeup\_flags, bool del);

（2）对比原理课上学到的信号量和p，v操作，说明ucore中信号机制的实现

信号量是一种同步互斥机制的实现，普遍存在于现在的各种操作系统内核里。相对于spinlock 的应用对象，信号量的应用对象是在临界区中运行的时间较长的进程。等待信号量的进程需要睡眠来减少占用 CPU 的开销。参考教科书“Operating Systems Internals and Design Principles”第五章“同步互斥”中对信号量实现的原理性描述：

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 \*/;

}

}

基于上诉信号量实现可以认为，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）。具体实现如下所示：

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;

}

与\_\_down相关的调用和被调用函数关系图如下所示：

digraph "\_\_down" {

graph [bgcolor="#F7F5F3", fontname="Arial", fontsize="10", label="", rankdir="LR"];

node [shape="box", style="filled", color="blue", fontname="Arial", fontsize="10", fillcolor="white", label=""];

edge [color="#CC0044", fontname="Arial", fontsize="10", label=""];

graph [bgcolor="#F7F5F3"];

\_\_N1 [color="red", label="\_\_down"];

\_\_N2 [label="\_\_intr\_save"];

\_\_N3 [label="\_\_intr\_restore"];

\_\_N4 [label="wait\_current\_set"];

\_\_N5 [label="schedule"];

\_\_N6 [label="wait\_in\_queue"];

\_\_N7 [label="wait\_queue\_del"];

\_\_N8 [label="down"];

\_\_N9 [label="phi\_take\_forks\_sema"];

\_\_N10 [label="cond\_signal"];

\_\_N11 [label="phi\_put\_forks\_sema"];

\_\_N12 [label="cond\_wait"];

\_\_N13 [label="lock\_mm"];

\_\_N14 [label="phi\_take\_forks\_condvar"];

\_\_N15 [label="phi\_put\_forks\_condvar"];

\_\_N1 -> \_\_N2;

\_\_N1 -> \_\_N3;

\_\_N1 -> \_\_N4;

\_\_N1 -> \_\_N5;

\_\_N1 -> \_\_N6;

\_\_N1 -> \_\_N7;

\_\_N9 -> \_\_N8;

\_\_N10 -> \_\_N8;

\_\_N11 -> \_\_N8;

\_\_N12 -> \_\_N8;

\_\_N13 -> \_\_N8;

\_\_N14 -> \_\_N8;

\_\_N15 -> \_\_N8;

\_\_N8 -> \_\_N1;

}

● \_\_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。具体实现如下所示：

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 {

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

}

}

local\_intr\_restore(intr\_flag);

}

与\_\_up相关的调用和被调用函数关系图如下所示：

digraph "\_\_up" {

graph [bgcolor="#F7F5F3", fontname="Arial", fontsize="10", label="", rankdir="LR"];

node [shape="box", style="filled", color="blue", fontname="Arial", fontsize="10", fillcolor="white", label=""];

edge [color="#CC0044", fontname="Arial", fontsize="10", label=""];

graph [bgcolor="#F7F5F3"];

\_\_N1 [color="red", label="\_\_up"];

\_\_N2 [label="\_\_intr\_save"];

\_\_N3 [label="wait\_queue\_first"];

\_\_N5 [label="wakeup\_wait"];

\_\_N6 [label="\_\_intr\_restore"];

\_\_N7 [label="up"];

\_\_N8 [label="phi\_test\_sema"];

\_\_N9 [label="phi\_take\_forks\_sema"];

\_\_N10 [label="cond\_signal"];

\_\_N11 [label="phi\_put\_forks\_sema"];

\_\_N12 [label="cond\_wait"];

\_\_N13 [label="unlock\_mm"];

\_\_N14 [label="phi\_take\_forks\_condvar"];

\_\_N15 [label="phi\_put\_forks\_condvar"];

\_\_N1 -> \_\_N2;

\_\_N1 -> \_\_N3;

\_\_N1 -> \_\_N5;

\_\_N1 -> \_\_N6;

\_\_N8 -> \_\_N7;

\_\_N9 -> \_\_N7;

\_\_N10 -> \_\_N7;

\_\_N11 -> \_\_N7;

\_\_N12 -> \_\_N7;

\_\_N13 -> \_\_N7;

\_\_N14 -> \_\_N7;

\_\_N15 -> \_\_N7;

\_\_N7 -> \_\_N1;

}

对照信号量的原理性描述和具体实现，可以发现二者在流程上基本一致，只是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。另外，我们可以看出信号量的计数器value具有有如下性质：

* 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

（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。

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;

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 \*/;

}

}

练习二：了解基于信号量和管理的哲学家就餐问题

1. 说明ucore中基于信号量的哲学家就餐问题的实现机制

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 \*/;

}

}

基于上诉信号量实现可以认为，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;

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：

① \_\_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）。具体实现如下所示：

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);//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

local\_intr\_restore(intr\_flag);

schedule();

local\_intr\_save(intr\_flag);//当被唤醒时

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

local\_intr\_restore(intr\_flag);

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

return wait->wakeup\_flags;

}

return 0;

}

 ②\_\_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。具体实现如下所示：

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) {//取得wait queue的第一个wait

sem->value ++;

}

else {

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

}

}

local\_intr\_restore(intr\_flag);

}

我们可以看出信号量的计数器value具有有如下性质：

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

————————————————

版权声明：本文为CSDN博主「cs\_assult」的原创文章，遵循 CC 4.0 BY-SA 版权协议，转载请附上原文出处链接及本声明。

原文链接：https://blog.csdn.net/cs\_assult/article/details/46852833

2、说明ucore中基于管理的哲学家就餐问题的实现机制

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

down(&(mtp->mutex));//进入临界区

//--------into routine in monitor--------------

// LAB7 EXERCISE1: YOUR CODE

// I am hungry

// try to get fork

// I am hungry

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

// try to get fork

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]);//如果得不到叉子就阻塞

}

//--------leave routine in monitor--------------

if(mtp->next\_count>0)//如果阻塞则唤醒

up(&(mtp->next));

else

up(&(mtp->mutex));//离开临界区

}

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

down(&(mtp->mutex));//进入临界区

//--------into routine in monitor--------------

// LAB7 EXERCISE1: YOUR CODE

// I ate over

// test left and right neighbors

// I ate over

state\_condvar[i]=THINKING;//哲学家进餐结束

// test left and right neighbors

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

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

//--------leave routine in monitor--------------

if(mtp->next\_count>0)<span style="line-height: 25.6000003814697px; font-family: 'Open Sans', 'Clear Sans', 'Helvetica Neue', Helvetica, Arial, sans-serif;">//如果存在阻塞则唤醒</span>

up(&(mtp->next));

else

up(&(mtp->mutex));//离开临界区

}

运行结果：

————————————————

版权声明：本文为CSDN博主「cs\_assult」的原创文章，遵循 CC 4.0 BY-SA 版权协议，转载请附上原文出处链接及本声明。

原文链接：https://blog.csdn.net/cs\_assult/article/details/46852833