ucoreOS操作系统实验——Lab6

问题发现与改进

- 信号量结构体中value在不同状态下有不同含义，>0时表示共享资源空闲数，=0等待队列为空，<0表示该信号量等待队列中的进程数

练习0

继续复制粘贴,并对 `trap.c` 做以下修改

```
static void trap_dispatch(struct trapframe *tf) {
    ++ticks;
    // sched_class_proc_tick(current);   //这一句已经被包含到run_timer_list中
    run_timer_list();
}
```

练习一

> **理解内核级信号量的实现和基于内核级信号量的哲学家就餐问题（不需要编码）**
>
> 1. 请在实验报告中给出内核级信号量的设计描述，并说明其大致执行流程。
> 2. 请在实验报告中给出给用户态进程/线程提供信号量机制的设计方案，并比较说明给内核级提供信号量机制的异同。

1. 信号量实现
    - 首先定义信号量和等待队列的数据结构

```
typedef struct {    //定义信号量数据结构  sem.h
    int value;    //计数值，用于后续PV原子操作
    wait_queue_t wait_queue;// 进程等待队列
} semaphore_t;
typedef struct {   //用于等待队列的结构  wait.h
    struct proc_struct *proc;  //存放了当前等待的线程PCB
    uint32_t wakeup_flags;   //唤醒原因
    wait_queue_t *wait_queue;//等待队列
    list_entry_t wait_link;   //还原结构体的等待队列标志
} wait_t;
```

    - `sem_init` : 对信号量进行初始化

```
void sem_init(semaphore_t *sem, int value) {
    sem->value = value;
    wait_queue_init(&(sem->wait_queue));
}
```

- __down: 对应P操作，表示请求一个该信号量对应的资源，采用禁用中断并保存eflag寄存器的值的方式来保证原子性，避免共享变量被多个线程同时修改
    - 通过判断计数值来知道是否存在多余的可分配的资源，大于0，则取出资源（计数值减1）并返回，之后当前进程便可以正常进行；
    - 如果没有可用的资源，计数值小于0，则表示已有其他线程访问临界区，则将该进程状态改为SLEEPING态，放入等待队列，并调用schedule来让出CPU，等到该进程被唤醒，再将它从当前进程取出并从等待队列删除，最后判断等待的进程是由于什么原因被唤醒
- __up: 对应V操作，表示释放了一个该信号量对应的资源，也采用禁用中断并保存eflag寄存器的值的方式来保证原子性
    - 判断等待队列是否为空，若为空，计数值+1并返回
    - 等待队列非空，说明还有线程在等待，取出等待队列的第一个线程将其唤醒，唤醒的过程中将其从等待队列删除
- try_down：不进入等待队列的P操作，即时是获取资源失败也不会堵塞当前进程

2. 用户态进程/线程提供信号量机制的设计方案，并比较异同
    - 将内核信号量机制迁移到用户态的最大困难是，用于保证操作原子性的禁用中断机制、以及CPU提供的Test and Set指令机制都只能在用户态下运行，软件方法的同步互斥没学也不会，所以没法在用户态下直接实现信号量机制。所以结合前面所学可以将信号量机制的实现放在OS内核中实现，最后使用系统调用的方法统一提供若干管理信号量的系统调用(但不可以直接使用信号量结构的指针作为参数)
        - 申请创建一个信号量的系统调用，可以指定初始值，返回一个信号量描述符(类似文件描述符)
        - 将指定信号量执行P操作
        - 将指定信号量执行V操作
        - 将指定信号量释放
    - 相同点
        - 提供信号量机制的代码实现逻辑是相同的
    - 不同点
        - 由于实现原子操作的中断禁用、Test and Set指令等均需要在内核态下运行，因此提供给用户态进程的信号量机制是通过系统调用来实现的，内核线程只需要直接调用相应的函数即可

练习二

> **完成内核级条件变量和基于内核级条件变量的哲学家就餐问题（需要编码）**
>
> 首先掌握管程机制，然后基于信号量实现完成条件变量实现，然后用管程机制实现哲学家就餐问题的解决方案（基于条件变量）

首先查看monitor，了解这里的管程实现，这里等待条件变量的进程的优先级更高

- 两个重要的结构：

```
typedef struct condvar{// 条件变量数据结构
    semaphore_t sem;  // 信号量，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;

typedef struct monitor{// 管程数据结构
    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;
```

- `monitor_init`：初始化管程

```c
void monitor_init (monitor_t * mtp, size_t num_cv) {
    int i;
    assert(num_cv>0);
    mtp->next_count = 0; // 睡在signal进程数 初始化为0
    mtp->cv = NULL;
    sem_init(&(mtp->mutex), 1); // 二值信号量 保护管程 使进程访问管程操作为互斥的
    sem_init(&(mtp->next), 0); // 条件同步信号量
    mtp->cv =(condvar_t *) kmalloc(sizeof(condvar_t)*num_cv);
  // 获取一块内核空间 放置条件变量
    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;
    }
}
```

- `cond_wait`：该函数的功能是令当前进程等待在指定信号量上
  - 条件变量的等待队列上的进程计数+1
  - 管程的next_count大于0，说明有进程睡在了signal操作上，我们需要将其唤醒
  - 管程的next_count小于0，说明当前没有进程睡在signal操作上，只需要释放互斥体
  - 之后自身阻塞将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

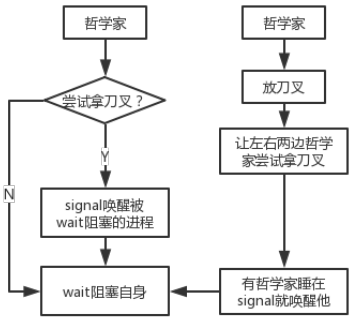
```c
void
cond_wait (condvar_t *cvp) {
    //LAB7 EXERCISE1: YOUR CODE
    cprintf("cond_wait begin:  cvp %x, cvp->count %d, cvp->owner->next_count %d\n", cvp, cvp->count, cvp->owner
   /*
    *         cv.count ++;
    *         if(mt.next_count>0)
    *            signal(mt.next)
    *         else
    *            signal(mt.mutex);
    *         wait(cv.sem);
    *         cv.count --;
    */
   cvp->count ++; // 修改等待在条件变量的等待队列上的进程计数
    if (cvp->owner->next_count > 0) { // 释放锁
        up(&cvp->owner->next);
    } else {
        up(&cvp->owner->mutex);
    }
    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      You, a few seconds ago • Uncommitted changes
    cprintf("cond_wait end:  cvp %x, cvp->count %d, cvp->owner->next_count %d\n", cvp, cvp->count, cvp->owner->
}
```

- `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
  - 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，没有则不需要进行任何操作，直接返回
  - 有正在等待的进程，则将其中的一个唤醒，这里的等待队列是使用了一个信号量来进行实现的，由于信号量中已经包括了对等待队列的操作，因此要进行唤醒只需要对信号量执行up操作即可
  - 同时设置条件变量所属管程的next_count+1，用来告诉wait有进程睡在signal操作上
  - 自身阻塞，等待条件同步被唤醒，next_count-1

```
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
  /*
   *      cond_signal(cv) {
   *          if(cv.count>0) {
   *             mt.next_count ++;
   *             signal(cv.sem);
   *             wait(mt.next);
   *             mt.next_count--;
   *          }
   *       }
   */
    if (cvp->count > 0) { // 判断条件变量的等待队列是否为空      You, a few seconds ago • Uncommitted changes
    cvp->owner->next_count ++; // 修改next变量上等待进程计数，跟下一个语句不能交换位置，为了得到互斥访问的效果，关键在于访问共
    up(&cvp->sem); // 唤醒等待队列中的某一个进程
    down(&cvp->owner->next); // 把自己等待在next条件变量上
    cvp->owner->next_count --; // 当前进程被唤醒，恢复next上的等待进程计数
    }
   cprintf("cond_signal end: cvp %x, cvp->count %d, cvp->owner->next_count %d\n", cvp, cvp->count, cvp->owner->
}
```

接下来解决哲学家就餐问题

整个问题的解决思路是



- `phi_take_forks_condvar`，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：
  - 给管程上锁，P操作进入管程
  - 将哲学家的状态修改为HUNGRY
  - 判断当前哲学家是否有足够的资源进行就餐（相邻的哲学家是否正在进餐）
  - 能进餐，将自己的状态修改成EATING，然后释放锁，并尝试唤醒因wait操作睡眠的进程，离开管程
  - 不能进餐，阻塞自己，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) {
    down(&(mtp->mutex));   //获取管程的锁P操作进入临界区
//--------into routine in monitor---------------
    // LAB7 EXERCISE1: YOUR CODE
    // I am hungry
    // try to get fork
    state_condvar[i] = HUNGRY; // 将自己设置为饥饿
    if (state_condvar[(i + 4) % 5] != EATING && state_condvar[(i + 1) % 5] != EATING)
    { // 判断当前叉子是否足够就餐       You, a few seconds ago • Uncommitted changes
        state_condvar[i] = EATING; // 就餐
    } else {
        cprintf("phi_take_forks_condvar: %d didn't get fork and will wait\n", i);
        cond_wait(mtp->cv + i); // 等待其他人释放资源
    }
//---------leave routine in monitor---------------
     if(mtp->next_count>0)
        up(&(mtp->next));
     else
        up(&(mtp->mutex));
}
```

- `phi_put_forks_condvar`，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，P操作进入管程
  - 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（使用cond_signal）
  - 释放锁，离开管程

```
void phi_put_forks_condvar(int i) {
    down(&(mtp->mutex));//获取管程的锁P操作进入临界区
//--------into routine in monitor---------------
    // LAB7 EXERCISE1: YOUR CODE
    // I ate over
    // test left and right neighbors
    state_condvar[i] = THINKING; // 停止就餐
    cprintf("phi_put_forks_condvar: %d finished eating\n", i);
    // 由于LAB7的评测脚本较弱，这是为了验证访问的互斥性而额外添加的注释性输出       You, a few seco
    phi_test_condvar((i + N - 1) % N); // 判断左右邻居的哲学家是否可以从等待就餐的状态中恢复过来
    phi_test_condvar((i + 1) % N);
//---------leave routine in monitor---------------
    if(mtp->next_count>0)
       up(&(mtp->next));
    else
       up(&(mtp->mutex));
}
```

由于限制了管程中在访问共享变量的时候处于RUNNABLE的进程只有一个，因此对进程的访问是互斥的。且由于每个哲学家只能占有所有需要的资源（刀叉）或者不占用资源，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，所以最终必定所有哲学将都能成功就餐

问题回答

1. 用户态进程/线程提供条件变量机制的设计方案，并比较说明给内核级提供条件变量机制的异同

- 这一lab管程的实现中互斥访问的保证是完全基于信号量的，即按照练习一，使用系统调用实现用户态的信号量的实现机制，就可以按照相同的逻辑在用户态实现管程机制和条件变量机制
- 当然也可以仿照用户态实现条件变量的方式，将对访问管程的操作封装成系统调用
- 异同点为：
  - 相同点：基本的实现逻辑相同
  - 不同点：最终在用户态下实现管程和条件变量机制，需要使用到操作系统使用系统调用提供一定的支持; 而在内核态下实现条件变量是不需要的

2. 能否不用基于信号量机制来完成条件变量？如果不能，请给出理由，如果能，请给出设计说明和具体实现
   - 能够基于信号量来完成条件变量机制
   - 事实上在本实验中就是这么完成的，只需要将使用信号量来实现条件变量和管程中使用的锁和等待队列即可

实验结果

```
cond_signal begin: cvp c03cc6c4, cvp->count 1, cvp->owner->next_count 0
cond_wait end:  cvp c03cc6c4, cvp->count 0, cvp->owner->next_count 1
Iter 4, No.3 philosopher_condvar is eating
cond_signal end: cvp c03cc6c4, cvp->count 0, cvp->owner->next_count 0
No.4 philosopher_condvar quit
No.1 philosopher_condvar quit
No.3 philosopher_condvar quit
pid 19 done!.
pid 25 done!.
pid 27 done!.
pid 32 done!.
pid 29 done!.
pid 23 done!.
pid 20 done!.
matrix pass.
all user-mode processes have quit.
kernel panic at kern/process/proc.c:859:
    assertion failed: nr_free_pages_store == nr_free_pages()
stack trackback:
ebp:0xc03cef8c eip:0xc0101edd args:0x00000001 0x00000000 0xc03cefbc 0xc03cefc0
    kern/debug/kdebug.c:350: print_stackframe+21
ebp:0xc03cefac eip:0xc01017d9 args:0xc0110504 0x0000035b 0xc0110531 0xc0110870
    kern/debug/panic.c:27: __panic+107
ebp:0xc03cefec eip:0xc010c418 args:0x00000000 0x00000000 0x00000010 0x0000000d
    kern/process/proc.c:859: init_main+361
Welcome to the kernel debug monitor!!
Type 'help' for a list of commands.
K>
```

```
badsegment:              (2.8s)
  -check result:                             OK
  -check output:                             OK
divzero:                 (2.8s)
  -check result:                             OK
  -check output:                             OK
softint:                 (3.0s)
  -check result:                             OK
  -check output:                             OK
faultread:               (1.7s)
  -check result:                             OK
  -check output:                             OK
faultreadkernel:         (1.7s)
  -check result:                             OK
  -check output:                             OK
hello:                   (2.9s)
  -check result:                             OK
  -check output:                             OK
testbss:                 (1.8s)
  -check result:                             OK
  -check output:                             OK
pgdir:                   (2.7s)
  -check result:                             OK
  -check output:                             OK
yield:                   (2.8s)
  -check result:                             OK
  -check output:                             OK
badarg:                  (2.7s)
  -check result:                            OK
  -check output:                            OK
exit:                    (2.8s)
  -check result:                            OK
  -check output:                            OK
spin:                    (3.2s)
  -check result:                            OK
  -check output:                            OK
waitkill:                (3.8s)
  -check result:                            OK
  -check output:                            OK
forktest:                (2.9s)
  -check result:                            OK
  -check output:                            OK
forktree:                (5.8s)
  -check result:                            OK
  -check output:                            OK
priority:                (15.6s)
  -check result:                            OK
  -check output:                            OK
sleep:                   (11.6s)
  -check result:                            OK
  -check output:                            OK
sleepkill:               (2.8s)
  -check result:                            OK
  -check output:                            OK
matrix:                  (21.5s)
  -check result:                           OK
  -check output:                          OK
Total Score: 190/190
```