操作系统原理实验

实验八（Ucore Lab7） - 同步互斥

实验准备

> 主机环境：macOS X 10.14.5 Mojave
>
> 虚拟机环境：Linux Ubuntu 14.04 LTS
>
> 虚拟机搭载软件：Parallels Desktop.app
>
> 命令行终端：Linux 下 Terminal
>
> 终端shell：zsh
>
> 之前的第一次 ucore 实验中，实验用虚拟机硬盘文件.vdi搭建的虚拟机环境存在兼容性不好，卡顿严重，分辨率也不高等问题。故从这次实验开始，在Github上将原实验项目clone下来，自己按照指导书第一章讲的环境配置，安装所需支持，配置实用工具进行实验。
>
> 为满足实验要求，将命令行用户名字段临时指定为姓名。（需要在zsh主题文件中修改显示。因为中文姓名在命令行终端会影响显示效果，故用我的中大NetID作为署名标记：许滨楠 - xubn。）

实验目的

> - 理解操作系统的同步互斥的设计实现；
> - 理解底层支撑技术：禁用中断、定时器、等待队列；
> - 在 ucore 中理解信号量 (semaphore) 机制的具体实现；
> - 了解经典进程同步问题，并能使用同步机制解决进程同步问题。

实验内容

> - 实验7 (lab6) 完成了用户进程的调度框架和具体的调度算法、可调度运行多个进程。如果多个进程需要协同操作或访问共享资源，则存在如何同步和有序竞争的问题。本次实验主要是熟悉 ucore 的进程同步机制 - 信号量 (semaphore) 机制，以及基于信号量的哲学家就餐问题解决方案。在本次实验中，在 kern/sync/check_sync.c 中提供了一个基于信号量的哲学家就餐问题解法。
> - 哲学家就餐问题描述如下：有五个哲学家，他们的生活方式是交替地进行思考和进餐。哲学家们公用一张圆桌，周围放有五把椅子，每人坐一把。在圆桌上有五个碗和五根筷子，当一个哲学家思考时，他不与其他人交谈，饥饿时便试图取其左、右最靠近他的筷子，但他可能一根都拿不到。只有在他拿到两根筷子时，方能进餐，进餐完后，放下筷子又继续思考。

练习

练习0

> 填写已有实验。

利用 meld 工具，将之前完成的 lab1/2/3/4/5/6 代码 merge 到 lab7 的代码中。因为之前已经将 lab1/2/3/4/5 的代码整合到 lab6 ，所以这次直接将 lab6 和 lab7 的代码在 meld 中进行目录比较，合并已经完成的代码。主要在之前编写过代码的文件中做比较，补充填入已完成的代码。由于之前在 merge 的时候错把一些新的 labcode 在与旧的 code merge 的时候去掉了，导致重要内容缺失，后续的测试总是出错，所以只需要把前面的 lab 练习中要求补充完整的函数 merge 到新的 lab 中。具体需要填写的文件和函数如下：

- Lab 1
  - kdebug.c : print_stackframe
- Lab 2
  - default_pmm.c : default_init
  - default_pmm.c : default_init_memmap
  - default_pmm.c : default_alloc_pages
  - default_pmm.c : default_free_pages
  - pmm.c : get_pte
  - pmm.c : page_remove_pte
- Lab 3
  - vmm.c : do_pgfault
  - swap_fifo.c : __fifo_map_swappable
  - swap_fifo.c : __fifo_swap_out_victim
- Lab 4 / 5
  - proc.c : alloc_proc
  - proc.c : do_fork
  - trap.c : idt_init
- Lab 6
  - kern/schedule/*

先前操作的文件进行比对合并时，发现其中在 /kern/trap/trap.c 中的函数 trap_dispatch 中有一处代码需要添加内容，以方便 Lab7 实验验证。根据注释提示，要完成 Lab7 的功能测试，需要在之前频繁改动的 ticks 时间计数的时候调用关键函数 run_timer_list，修改代码如下：

```
case IRQ_OFFSET + IRQ_TIMER:
    /* LAB7 YOUR CODE */
    /* you should upate you lab6 code
     * IMPORTANT FUNCTIONS:
     * run_timer_list
     */
    ++ticks;
    assert(current != NULL);
    run_timer_list();
    break;
```

另外，由于这里对 run_timer_list 函数进行了调用，而在 /kern/schedule/sched.c 中可以看到 run_time_list 函数内部调用了 sched_class_proc_tick 函数，该函数位于同文件中，类型为 static，如果不将 static 的前缀去掉，会导致 make qemu 失败，所以此处也把 sched_class_proc_tick 函数的 static 前缀去除。

练习1

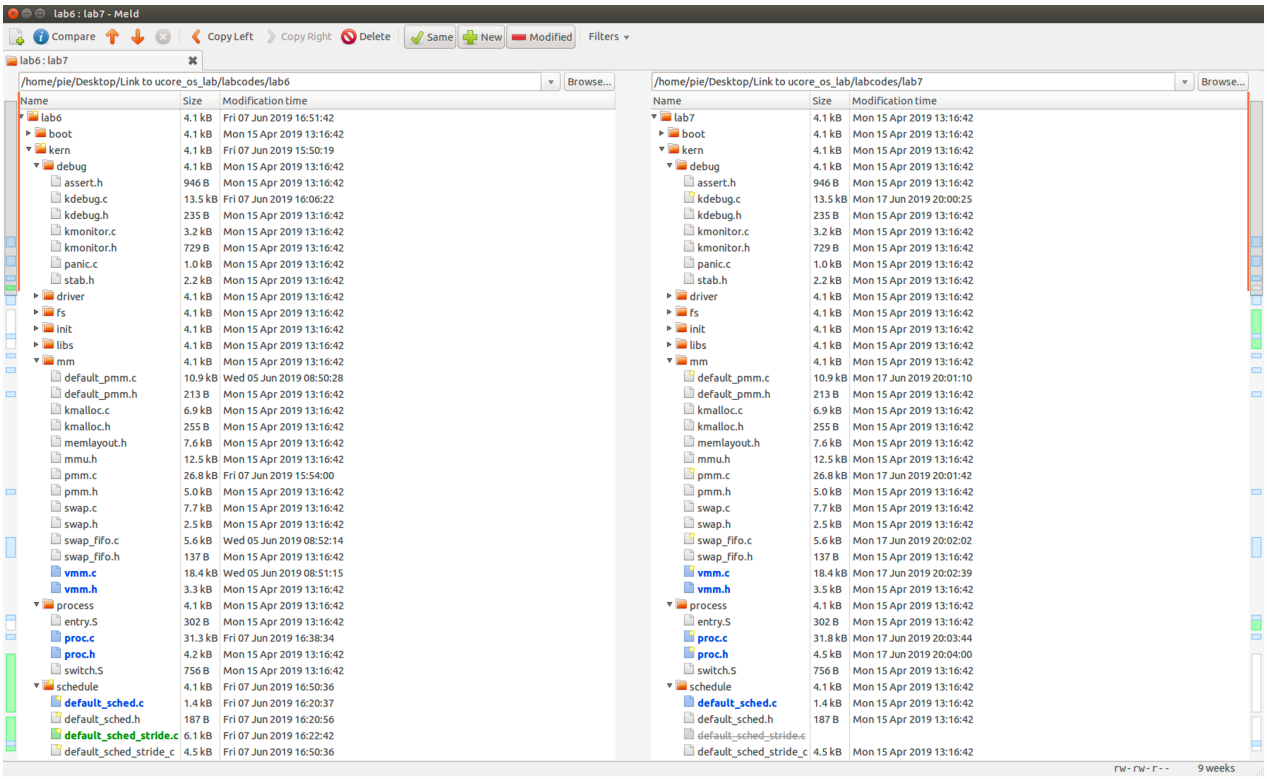
> 理解内核级信号量的实现和基于内核级信号量的哲学家就餐问题。

练习 0 测试

> 完成练习 0 后，比较 lab6 和练习 0 完成后的刚修改的 lab7 之间的区别，分析了解 lab7 采用信号量的执行过程。执行 make grade，查看测试样例的通过情况。

区别比较

用 meld diff 比较软件对之前完成的 lab6 和练习 0 中完成的 lab7：



合并过后一些基本的之前实验中的模块和前面的实验保持一致。上面列举的需要将之前的 lab 中实现的代码加入的板块之外，还有许多新增的代码模块，用于支持同步互斥问题中的实现。新增的内容主要有：

- /kern/trap/trap.c 中，对 trap_dispatch 的修改，调度了位于 /kern/schedule/sched.c 中的 run_time_list() 函数，用于支持定时器机制；
- /kern/sync/ 中添加了许多相关的源代码文件，用于支持 ucore 的内核信号量和同步互斥实现；
- 加入了系统对 sleep 系统调用的支持；

lab7 对比 lab6，在执行过程上主要是添加了 sync 相关内容，在函数开始调度之前先调用了 check_sync 函数，这个函数的主要内容是模拟执行了基于信号量和管程的哲学家就餐问题，实验中我们先关注基于信号量实现的部分：

```
1  void check_sync(void){
2      int i;
3      //check semaphore
4      sem_init(&mutex, 1);
5      for(i=0;i<N;i++){
6          sem_init(&s[i], 0);
7          int pid = kernel_thread(philosopher_using_semaphore, (void *)i,
   0);
8          if (pid <= 0) {
9              panic("create No.%d philosopher_using_semaphore failed.\n");
10         }
11         philosopher_proc_sema[i] = find_proc(pid);
12         set_proc_name(philosopher_proc_sema[i], "philosopher_sema_proc");
13     }
14 }
```

首先，利用 sem_init 函数初始化了互斥的信号量，互斥信号量的数据结构位于 /kern/sync/sem.h ，如下：

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

其中包含 value 表示互斥信号量的值，在理论课上我们已经学习过，这个值可以理解为可用资源的份数，正数表示系统中该资源仍有可用的单位，进程在遇到信号量为正的时候，其资源请求可以直接得到满足；0 或负数表示资源目前没有直接可用量，而且负数表示仍有其他请求者在等待中，则进程遇到这种情况会加入等待队列，也就是数据结构中的 wait_queue ，等待目标资源被释放，然后系统会从等待队列中唤醒一个等待者，满足其资源需要。所以信号量的初始化就是创建这样一个数据结构来完成信号量的记录管理工作。回到 check_sync 函数执行过程，接下来因为源文件中宏定义 #define N 5 所以循环创建了 5 个内核线程，代表 5 个哲学家， 每个线程内容如下：

```
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;
}
```

线程的函数中主要实现了哲学家思考（挂起）、用餐（需要资源）的行为。其中与信号量相关的重点内容是 phi_take_forks_sema 和 phi_put_forks_sema 两个函数，其具体内容如下：

```
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 */
}
```

依然是对哲学家行为的模拟，其中与信号量操作相关的是 up 和 down 两个函数，其具体实现位于 /kern/sync/sem.c 中，又调用了两个底层 __up 和 __down 函数，主要实现如下：

```
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);
}

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;
}
```

其实 up 和 down 相关函数分别对应的是理论课上信号量的 signal (V) 和 wait (P) 两种操作。

实现 signal 操作的 up 函数，通过 local_intr_save 函数关闭系统中断，然后如果该信号量的等待队列标识为无进程正在等待，则将信号量的值 + 1；如果等待队列有进程在等待，则调用 wakeup_wait 函数将等待队列中的队首元素弹出并唤醒，完成调度。最后都会通过 local_intr_restore 函数重新恢复中断。

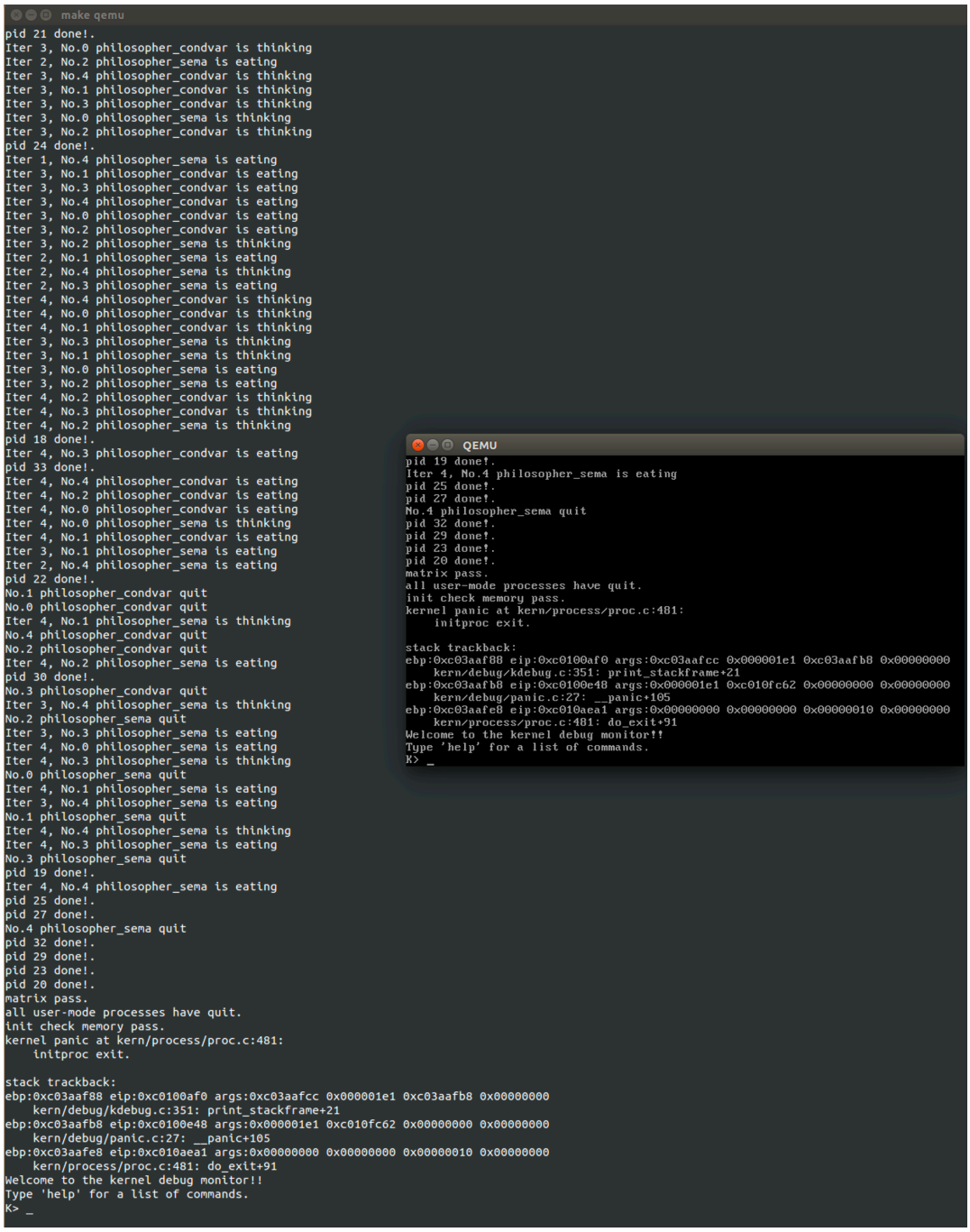
实现 wait 操作的 down 函数，依然是先关闭中断，然后判断当前信号量的值是否为正，是则表示当前所请求的资源可用或者有多余的份数可用，可以获得资源的使用权，此时将信号量的值 - 1 用以表示一份资源被占用，接着恢复中断并返回；如果信号量的值不为正，则表示资源暂时无法直接使用，便将当前进程加入等待队列，然后开中断，接着调用 schedule 函数进行调度，选择另外一个进程执行，进入等待唤醒的状态，被唤醒后就会再次关中断——从等待队列中弹出——开中断。

分析至此，已经了解了 ucore 中提供信号量机制的实现中的重要接口，包括信号量的设置、初始化，以及重要的 V P 操作。结合理论课上早已提及的信号量使用的基本框架和规则，lab7 中利用信号量机制实现哲学家就餐问题的实现就不难理解了：函数定义哲学家的行为并且不断循环，当行为进行到需要访问资源——筷子（叉子）的时候借用信号量实现同一只筷子每次只能一位哲学家使用的要求。

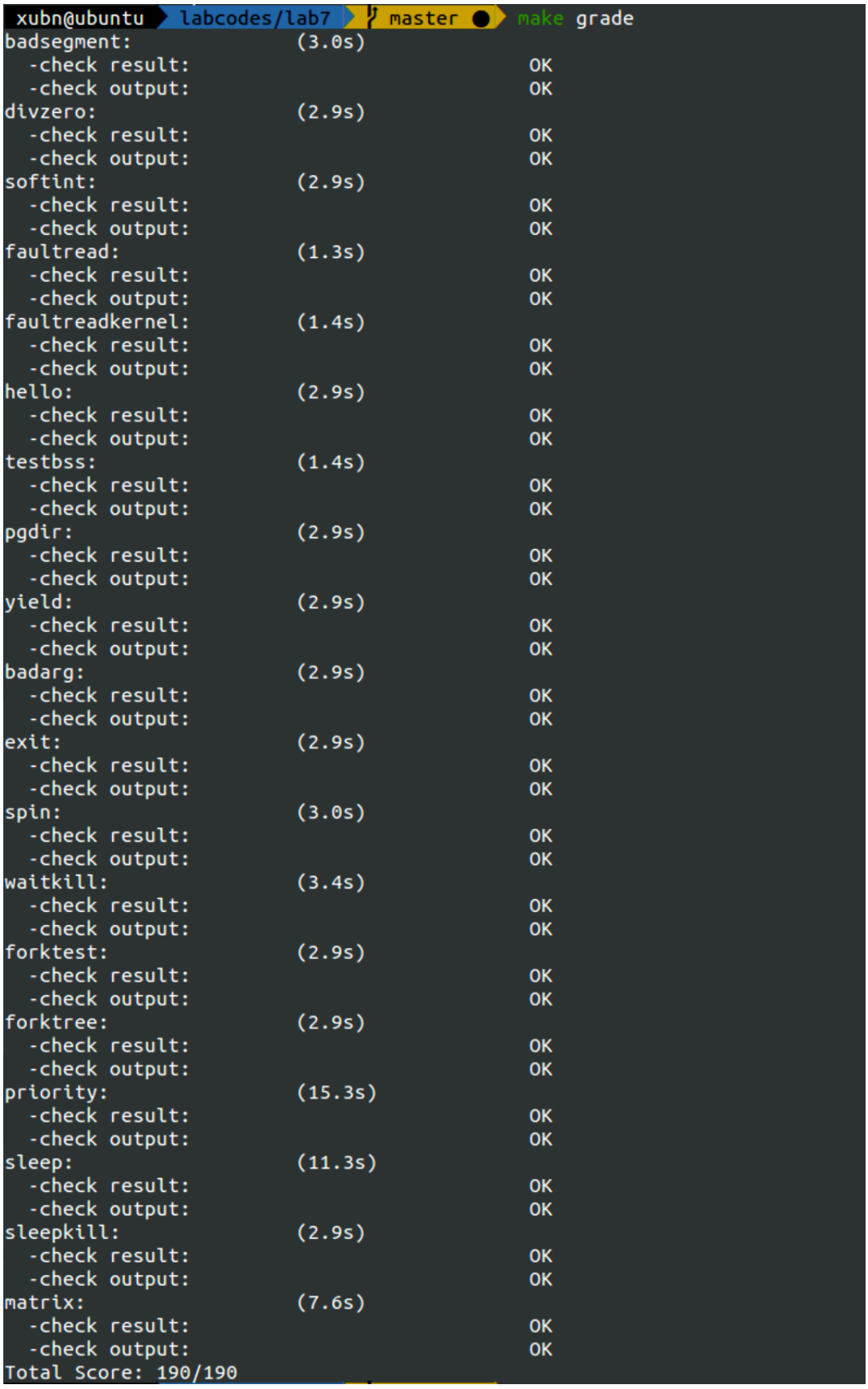
结果验证

接下来对代码的 merge 和 update 结果进行验证。

先执行 make qemu 查看系统能否正常运行起来：



调试、调整 merge 过程中出错的地方，改动完毕后系统可以正常运行。此时运行 make grade 进行测试，等待测试样例跑完后查看测试结果：



在之前实验都已经通过测试的情况下，将之前实验内容合并到新的实验中，与新实验中的新功能完成正常的耦合之后，当前的系统实现已经可以通过所有测试样例。

内核信号量设计

> 给出内核信号量的设计描述，并说明其大致的执行流程。

内核信号量的设计模式基本是理论课上的互斥信号量以及其对应的 V P 操作配合实现互斥控制的框架。其中的具体设计、相关函数、执行流程和简单的调用栈已经在上面对 check_sync 函数中实现的哲学家就餐问题中做了探讨和比较详细的说明。

用户态进程/线程信号量机制

> 给出用户态进程/线程信号量机制的设计方案。

信号量机制的实现的重点是要保证对于信号量的两种操作，V P 即 signal 和 wait 操作的原子性。在内核级的信号量实现中可以通过直接关闭中断，避免抢占破坏执行过程中的操作的原子性。所以用户态进程/线程信号量机制的实现重点也是需要解决这个问题，其他包括数据结构和相关操作都可以与内核级信号量机制一样，用 value 和 wait_queue 完成对信号量相应资源和对其请求的管理。重点是要完成 V P 操作的实现，可以考虑利用系统调用，用系统中已经完成的内核级信号量实现机制来完成，即通过系统调用来关中断——管理信号量中信息——开中断，实现信号量操作的原子性。这样就可以实现用户态下的信号量机制，支持用户态进程/线程的同步互斥了。

两者异同对比

> 比较说明给用户态进程/线程提供信号量机制和给内核级进程/线程提供信号量机制的异同。

用户态进程/线程依赖的信号量机制和内核级进程/线程依赖的信号量机制的相同之处在于：

- 两者依赖的信号量机制逻辑是一致的：
  - 都通过信号量这一数据结构管理一个类似计数器的值和等待队列；
  - 都通过关中断——维护——开中断的流程对信号量进行更新管理；

两者的不同之处在于：

- 实现信号量操作原子性的机制不同：
  - 系统内核级的信号量操作实现通过直接对中断开关进行操作实现；
  - 用户态下信号量操作没有足够的权限管理中断，所以需要通过系统调用实现。

实验总结

分析与区别

- 本次实验未进行编码相关内容。

重要知识点

- 同步互斥相关知识、信号量机制的原理和其在操作系统中的实现方法；
- 哲学家就餐的同步互斥情景问题；

补充知识点

同步互斥的其他实现，比如管程，在这次实验修改后没有涉及，接下来的大作业可能会涉及到。