

操作系统大作业 1 实验报告

智能工程学院 自动化 1 班 黄舒怡 17364029

○、 GitHub 上传情况

一共上传了 6 个代码，在所要求的的基础上多增加了 unlink.c。此代码于第二题——生产者消费者问题中使用，其作用为删掉创建的共享内存以及有名信号量。具体代码见第二题部分。

Makefile	1	100分的Makefile
cons.c	4	不能再高的prod和cons
dph.c	2	满分的dph
mycall.c	6	压轴出场的mycall
prod.c	3	不能再高的prod和cons
unlink.c	5	很有用的unlink

其中，Makefile 中包含 3 种命令，分别为 make all（同时产生上述文件对应的可执行文件）、make（产生单独的 bin 文件）、make clean（删除已生成的可执行文件）。其代码如下：

```
1  help:
2      @echo "Please input like 'make dph'"
3  dph :
4      @gcc dph.c -lpthread -o dph
5  prod :
6      @gcc prod.c -lm -lpthread -o prod
7  cons :
8      @gcc cons.c -lm -lpthread -o cons
9  unlink:
10     @gcc unlink.c -lpthread -o unlink
11 mycall :
12     @gcc mycall.c -o mycall
13 all : dph prod cons unlink mycall
14
15 clean :
16     @rm dph prod cons unlink mycall
17
```

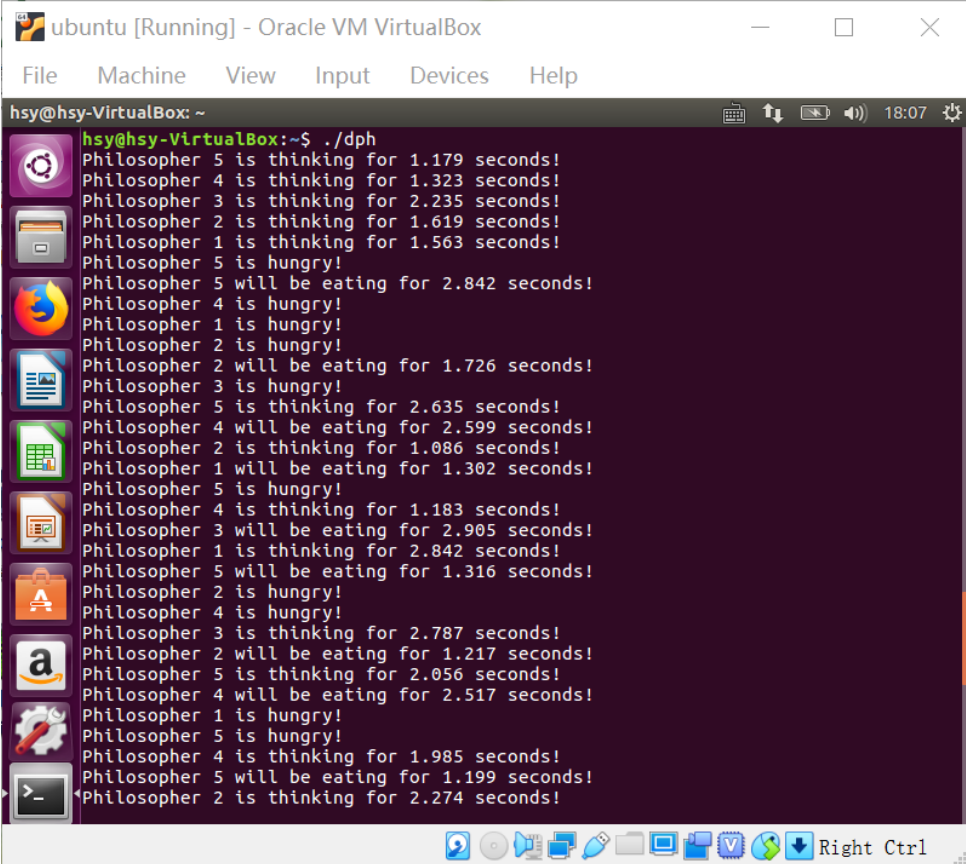
运行情况示例如下：

```
hsy@hsy-VirtualBox:~$ ls
cons.c  examples.desktop  mycall  prod.c  公共的  视频  文档  音乐
dph.c  Makefile             mycall.c  unlink.c  模板  图片  下载  桌面
hsy@hsy-VirtualBox:~$ make
Please input like 'make dph'!
hsy@hsy-VirtualBox:~$ make dph
hsy@hsy-VirtualBox:~$ ls
cons.c  dph.c  examples.desktop  mycall  prod.c  公共的  模板  图片  下载  桌面
dph     Makefile             mycall.c  unlink.c  prod.c  公共的  视频  文档  音乐
hsy@hsy-VirtualBox:~$ make all
prod.c: In function 'producer':
prod.c:45:9: warning: format '%d' expects argument of type 'int', but argument 3
      has type 'long int' [-Wformat=]
      printf("Produce: pid = %d, tid = %d, data = %d\n", getpid(), syscall(SYS_getti
      ^
cons.c: In function 'consumer':
cons.c:44:9: warning: format '%d' expects argument of type 'int', but argument 3
      has type 'long int' [-Wformat=]
      printf("Consume: pid = %d, tid = %d, data = %d\n", getpid(), syscall(SYS_getti
      ^
hsy@hsy-VirtualBox:~$ ls
cons  dph.c  examples.desktop  mycall  prod.c  公共的  图片  音乐
cons.c  dph     examples.desktop  mycall.c  unlink  模板  文档  桌面
dph     Makefile             mycall.c  unlink.c  unlink  模板  文档  下载
```

目录

一、哲学家就餐问题.....	3
二、生产者消费者问题.....	4
1. $\lambda_p=2000, \lambda_c=100$	4
2. $\lambda_p=100, \lambda_c=2000$	4
3. $\lambda_p=3000, \lambda_c=3000$	5
4. unlink.c.....	5
三、Linux 内核实验.....	6
a) complete fair scheduler (CFS) 内核代码阅读报告.....	6
1 Linux 进程的基本结构.....	6
1.1 进程基本属性.....	6
1.2 进程间关系.....	6
1.3 调度参数.....	7
1.4 文件和内存指针.....	8
1.5 信号.....	8
1.6 其它.....	8
2 进程的组织结构.....	8
2.1 双向链表.....	9
2.2 散列表.....	9
3 进程状态设置.....	9
4 Linux 的调度架构.....	10
4.1 从调度时机到调度器架构.....	10
4.1.1 调度函数.....	10
4.1.2 调度类.....	11
5 CFS 调度算法.....	11
5.1 思路.....	11
5.2 CFS 调度算法使用的主要结构.....	12
5.2.1 调度实体.....	12
5.2.2 就绪队列.....	12
5.3 CFS 调度算法的主要流程.....	13
5.4 Q & A.....	14
5.4.1 简述进程优先级、nice 值和权重之间的关系.....	14
5.4.2 vruntime 的基本思想是什么.....	14
5.4.3 vruntime 是如何计算的, 何时得到更新.....	14
5.4.4 min_vruntime 有什么作用.....	15
b) 添加系统调用: 用户层测试程序 mycall.c.....	16
1. 运行 screenfetch.....	16
2. 添加系统调用.....	16

一、哲学家就餐问题



The screenshot shows a terminal window titled "ubuntu [Running] - Oracle VM VirtualBox". The terminal output displays the execution of the `./dph` program, which simulates the Dining Philosophers problem. The output consists of a series of status messages for five philosophers (1-5), indicating their current state (thinking, hungry, or eating) and the time taken for each action. The messages are as follows:

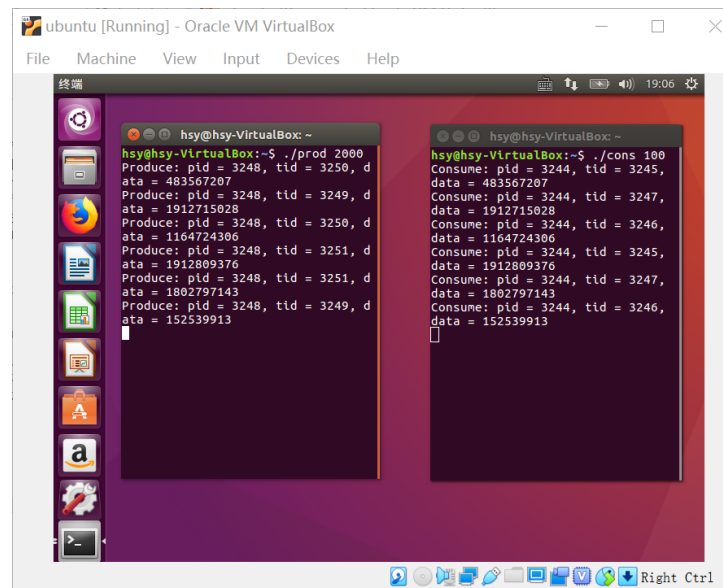
```
hsy@hsy-VirtualBox:~$ ./dph
Philosopher 5 is thinking for 1.179 seconds!
Philosopher 4 is thinking for 1.323 seconds!
Philosopher 3 is thinking for 2.235 seconds!
Philosopher 2 is thinking for 1.619 seconds!
Philosopher 1 is thinking for 1.563 seconds!
Philosopher 5 is hungry!
Philosopher 5 will be eating for 2.842 seconds!
Philosopher 4 is hungry!
Philosopher 1 is hungry!
Philosopher 2 is hungry!
Philosopher 2 will be eating for 1.726 seconds!
Philosopher 3 is hungry!
Philosopher 5 is thinking for 2.635 seconds!
Philosopher 4 will be eating for 2.599 seconds!
Philosopher 2 is thinking for 1.086 seconds!
Philosopher 1 will be eating for 1.302 seconds!
Philosopher 5 is hungry!
Philosopher 4 is thinking for 1.183 seconds!
Philosopher 3 will be eating for 2.905 seconds!
Philosopher 1 is thinking for 2.842 seconds!
Philosopher 5 will be eating for 1.316 seconds!
Philosopher 2 is hungry!
Philosopher 4 is hungry!
Philosopher 3 is thinking for 2.787 seconds!
Philosopher 2 will be eating for 1.217 seconds!
Philosopher 5 is thinking for 2.056 seconds!
Philosopher 4 will be eating for 2.517 seconds!
Philosopher 1 is hungry!
Philosopher 5 is hungry!
Philosopher 4 is thinking for 1.985 seconds!
Philosopher 5 will be eating for 1.199 seconds!
Philosopher 2 is thinking for 2.274 seconds!
```

```
Philosopher 3 is hungry!
Philosopher 3 will be eating for 2.155 seconds!
Philosopher 5 is thinking for 1.924 seconds!
Philosopher 1 will be eating for 1.729 seconds!
Philosopher 4 is hungry!
Philosopher 2 is hungry!
Philosopher 5 is hungry!
Philosopher 1 is thinking for 2.076 seconds!
Philosopher 5 will be eating for 2.067 seconds!
Philosopher 3 is thinking for 1.931 seconds!
Philosopher 2 will be eating for 1.399 seconds!
Philosopher 5 is thinking for 1.431 seconds!
Philosopher 4 will be eating for 1.617 seconds!
Philosopher 1 is hungry!
Philosopher 2 is thinking for 1.930 seconds!
Philosopher 1 will be eating for 2.107 seconds!
Philosopher 5 is hungry!
Philosopher 4 is thinking for 2.852 seconds!
Philosopher 3 is hungry!
Philosopher 3 will be eating for 2.550 seconds!
Philosopher 1 is thinking for 2.670 seconds!
Philosopher 5 will be eating for 2.694 seconds!
Philosopher 2 is hungry!
Philosopher 4 is hungry!
Philosopher 5 is thinking for 1.628 seconds!
Philosopher 3 is thinking for 2.658 seconds!
Philosopher 4 will be eating for 2.646 seconds!
Philosopher 2 will be eating for 2.066 seconds!
Philosopher 1 is hungry!
Philosopher 4 is thinking for 2.960 seconds!
Philosopher 2 is thinking for 1.181 seconds!
Philosopher 1 will be eating for 1.971 seconds!
Philosopher 3 is hungry!
Philosopher 3 will be eating for 2.802 seconds!
Philosopher 4 is hungry!
Philosopher 1 is thinking for 1.497 seconds!
Philosopher 2 is hungry!
Philosopher 3 is thinking for 2.110 seconds!
Philosopher 4 will be eating for 1.372 seconds!
Philosopher 2 will be eating for 2.553 seconds!
Philosopher 1 is hungry!
Philosopher 4 is thinking for 1.627 seconds!
Philosopher 2 is thinking for 2.357 seconds!
Philosopher 1 will be eating for 1.104 seconds!
Philosopher 3 is hungry!
Philosopher 3 will be eating for 1.253 seconds!
Philosopher 1 is thinking for 1.513 seconds!
Philosopher 3 is thinking for 2.028 seconds!
```

二、生产者消费者问题

1. $\lambda_p=2000, \lambda_c=100$

消费者的消费速度远大于生产者的生产速度——生产者每生成一个数据，就立刻被消费者消费，缓冲区始终为 empty 状态。

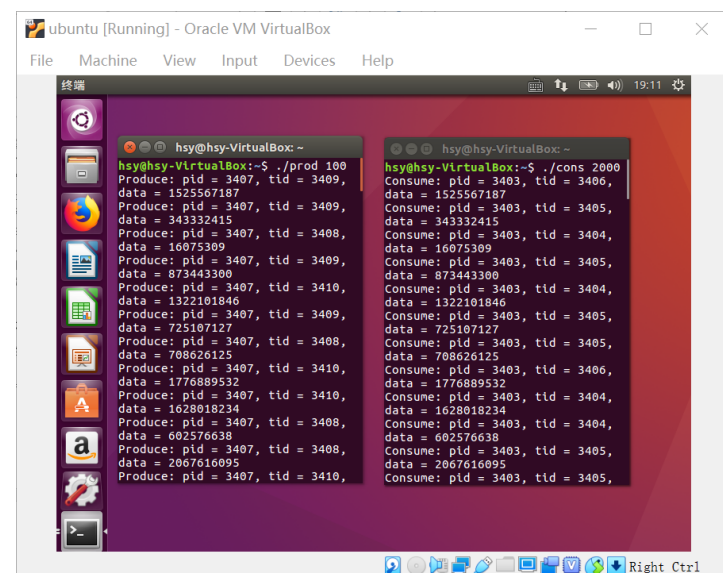


```
hsy@hsy-VirtualBox:~$ ./prod 2000
Produce: pid = 3248, tid = 3250, d
ata = 483567207
Produce: pid = 3248, tid = 3249, d
ata = 1912715028
Produce: pid = 3248, tid = 3250, d
ata = 1164724306
Produce: pid = 3248, tid = 3251, d
ata = 1912809376
Produce: pid = 3248, tid = 3251, d
ata = 1802797143
Produce: pid = 3248, tid = 3249, d
ata = 152539913

hsy@hsy-VirtualBox:~$ ./cons 100
Consume: pid = 3244, tid = 3245,
data = 483567207
Consume: pid = 3244, tid = 3247,
data = 1912715028
Consume: pid = 3244, tid = 3246,
data = 1164724306
Consume: pid = 3244, tid = 3245,
data = 1912809376
Consume: pid = 3244, tid = 3247,
data = 1802797143
Consume: pid = 3244, tid = 3246,
data = 152539913
```

2. $\lambda_p=100, \lambda_c=2000$

生产者的生产速度远大于消费者的消费速度——缓冲区始终为 full 状态，所以消费者当下消费的数据和生产者当下生产的数据始终相差 20 个。



```
hsy@hsy-VirtualBox:~$ ./prod 100
Produce: pid = 3407, tid = 3409,
data = 1525567187
Produce: pid = 3407, tid = 3409,
data = 343332415
Produce: pid = 3407, tid = 3408,
data = 16075309
Produce: pid = 3407, tid = 3409,
data = 873443300
Produce: pid = 3407, tid = 3410,
data = 1322101846
Produce: pid = 3407, tid = 3409,
data = 725107127
Produce: pid = 3407, tid = 3408,
data = 70806125
Produce: pid = 3407, tid = 3410,
data = 1776889532
Produce: pid = 3407, tid = 3410,
data = 1628018234
Produce: pid = 3407, tid = 3408,
data = 602576638
Produce: pid = 3407, tid = 3408,
data = 2067616095
Produce: pid = 3407, tid = 3410,

hsy@hsy-VirtualBox:~$ ./cons 2000
Consume: pid = 3403, tid = 3406,
data = 1525567187
Consume: pid = 3403, tid = 3405,
data = 343332415
Consume: pid = 3403, tid = 3404,
data = 16075309
Consume: pid = 3403, tid = 3405,
data = 873443300
Consume: pid = 3403, tid = 3404,
data = 1322101846
Consume: pid = 3403, tid = 3405,
data = 725107127
Consume: pid = 3403, tid = 3405,
data = 70806125
Consume: pid = 3403, tid = 3406,
data = 1776889532
Consume: pid = 3403, tid = 3404,
data = 1628018234
Consume: pid = 3403, tid = 3404,
data = 602576638
Consume: pid = 3403, tid = 3405,
data = 2067616095
```

```
hsy@hsy-VirtualBox: ~  
Produce: pid = 3407, tid = 3410, data = 371334813  
Produce: pid = 3407, tid = 3409, data = 280900240  
Produce: pid = 3407, tid = 3408, data = 1845823272  
Produce: pid = 3407, tid = 3410, data = 1733050315  
Produce: pid = 3407, tid = 3409, data = 1274289499  
Produce: pid = 3407, tid = 3408, data = 221633292  
Produce: pid = 3407, tid = 3410, data = 2122460924  
Produce: pid = 3407, tid = 3409, data = 40017244  
Produce: pid = 3407, tid = 3408, data = 2007115687  
Produce: pid = 3407, tid = 3410, data = 525759440  
Produce: pid = 3407, tid = 3409, data = 730589609  
^C  
hsy@hsy-VirtualBox:~$  
  
hsy@hsy-VirtualBox: ~  
Consume: pid = 3403, tid = 3406, data = 439250134  
Consume: pid = 3403, tid = 3404, data = 31595378  
Consume: pid = 3403, tid = 3406, data = 557839498  
Consume: pid = 3403, tid = 3404, data = 1485103186  
Consume: pid = 3403, tid = 3404, data = 864872343  
Consume: pid = 3403, tid = 3406, data = 995473402  
Consume: pid = 3403, tid = 3404, data = 1757974264  
Consume: pid = 3403, tid = 3405, data = 281976  
Consume: pid = 3403, tid = 3404, data = 764797543  
Consume: pid = 3403, tid = 3405, data = 2060432198  
Consume: pid = 3403, tid = 3404, data = 1773287530  
^C  
hsy@hsy-VirtualBox:~$
```

```
Produce: pid = 3407, tid = 3408, data = 439250134  
Produce: pid = 3407, tid = 3410, data = 31595378  
Produce: pid = 3407, tid = 3409, data = 557839498  
Produce: pid = 3407, tid = 3408, data = 1485103186  
Produce: pid = 3407, tid = 3410, data = 864872343  
Produce: pid = 3407, tid = 3409, data = 995473402  
Produce: pid = 3407, tid = 3408, data = 1757974264  
Produce: pid = 3407, tid = 3410, data = 281976  
Produce: pid = 3407, tid = 3409, data = 764797543  
Produce: pid = 3407, tid = 3408, data = 2060432198  
Produce: pid = 3407, tid = 3410, data = 1773287530  
Produce: pid = 3407, tid = 3409, data = 2134179013  
Produce: pid = 3407, tid = 3410, data = 1295514114  
Produce: pid = 3407, tid = 3409, data = 1973149876  
Produce: pid = 3407, tid = 3408, data = 114594618  
Produce: pid = 3407, tid = 3410, data = 2088653303  
Produce: pid = 3407, tid = 3408, data = 2075525967  
Produce: pid = 3407, tid = 3409, data = 1526048387  
Produce: pid = 3407, tid = 3408, data = 860225859  
Produce: pid = 3407, tid = 3410, data = 371334813  
Produce: pid = 3407, tid = 3409, data = 280900240  
Produce: pid = 3407, tid = 3408, data = 1845823272  
Produce: pid = 3407, tid = 3410, data = 1733050315  
Produce: pid = 3407, tid = 3409, data = 1274289499  
Produce: pid = 3407, tid = 3408, data = 2122460924  
Produce: pid = 3407, tid = 3409, data = 40017244  
Produce: pid = 3407, tid = 3408, data = 2007115687  
Produce: pid = 3407, tid = 3410, data = 525759440  
Produce: pid = 3407, tid = 3409, data = 730589609  
^C  
hsy@hsy-VirtualBox:~$
```

3. $\lambda p=3000$, $\lambda c=3000$

无明显规律。

```
hsy@hsy-VirtualBox: ~  
hsy@hsy-VirtualBox:~$ ./prod 3000  
Produce: pid = 3592, tid = 3595, data = 1537346326  
Produce: pid = 3592, tid = 3593, data = 1062418200  
Produce: pid = 3592, tid = 3595, data = 583502911  
Produce: pid = 3592, tid = 3593, data = 1278085430  
Produce: pid = 3592, tid = 3593, data = 1601929468  
Produce: pid = 3592, tid = 3594, data = 1624144060  
Produce: pid = 3592, tid = 3593, data = 1297479371  
Produce: pid = 3592, tid = 3594, data = 1953910889  
Produce: pid = 3592, tid = 3593, data = 1603268778  
^C  
hsy@hsy-VirtualBox:~$  
  
hsy@hsy-VirtualBox: ~  
hsy@hsy-VirtualBox:~$ ./cons 3000  
Consume: pid = 3587, tid = 3589, data = 1537346326  
Consume: pid = 3587, tid = 3590, data = 1062418200  
Consume: pid = 3587, tid = 3588, data = 583502911  
Consume: pid = 3587, tid = 3588, data = 1278085430  
Consume: pid = 3587, tid = 3589, data = 1601929468  
Consume: pid = 3587, tid = 3588, data = 1624144060  
Consume: pid = 3587, tid = 3588, data = 1297479371  
Consume: pid = 3587, tid = 3590, data = 1953910889  
Consume: pid = 3587, tid = 3589, data = 1603268778  
^C  
hsy@hsy-VirtualBox:~$
```

```
hsy@hsy-VirtualBox: ~  
data = 1558200475  
Produce: pid = 3592, tid = 3594, data = 1728901263  
Produce: pid = 3592, tid = 3595, data = 416502274  
Produce: pid = 3592, tid = 3595, data = 531468523  
Produce: pid = 3592, tid = 3595, data = 48507473  
Produce: pid = 3592, tid = 3594, data = 571803373  
Produce: pid = 3592, tid = 3593, data = 1984339213  
Produce: pid = 3592, tid = 3595, data = 1677630664  
Produce: pid = 3592, tid = 3593, data = 281050427  
Produce: pid = 3592, tid = 3593, data = 245955477  
Produce: pid = 3592, tid = 3594, data = 1363078761  
Produce: pid = 3592, tid = 3594, data = 1970447133  
^C  
hsy@hsy-VirtualBox:~$  
  
hsy@hsy-VirtualBox: ~  
data = 61204660  
Consume: pid = 3587, tid = 3590, data = 1477825023  
Consume: pid = 3587, tid = 3588, data = 1167797317  
Consume: pid = 3587, tid = 3589, data = 1275728169  
Consume: pid = 3587, tid = 3590, data = 522174386  
Consume: pid = 3587, tid = 3588, data = 129584988  
Consume: pid = 3587, tid = 3590, data = 1861953621  
Consume: pid = 3587, tid = 3590, data = 1558200475  
Consume: pid = 3587, tid = 3590, data = 1728901263  
Consume: pid = 3587, tid = 3590, data = 416502274  
Consume: pid = 3587, tid = 3590, data = 531468523  
Consume: pid = 3587, tid = 3590, data = 48507473  
^C  
hsy@hsy-VirtualBox:~$
```

4. unlink.c

```
1 #include <semaphore.h>  
2  
3 int main() {  
4     sem_unlink("/INIT");  
5     sem_unlink("/FULL");  
6     sem_unlink("/EMPTY");  
7     return 0;  
8 }
```

unlink 的作用:每次运行 prod 和 cons 前运行 unlink,删掉已创建的共享内存和有名信号量,防止历史数据对新一次的运行造成影响。所以, unlink 是有用的!

三、Linux 内核实验

a)complete fair scheduler (CFS) 内核代码阅读报告

1 Linux 进程的基本结构

Linux 内核使用 `task_struct` 结构体描述进程，该结构体定义在 `<include/linux/sched.h>` 中。`task_struct` 结构体是一个类似于 PCB 的结构，其中保存了进程的大量基本信息，主要内容大致可分为以下几类：

1.1 进程基本属性

```
volatile long state;    /* -1 unrunnable, 0 runnable, >0 stopped */
atomic_t usage;
pid_t pid;
pid_t tgid;
```

进程状态 state: 进程的状态码，其二进制的每位代表一个不同的状态。状态码分运行状态和退出状态两部分，对应位置为 1 时即标识进程处于该状态。

引用次数 usage: 结构体的被引用次数。只有当该变量为 0 时，结构体才能被销毁。

进程标识符 pid: 进程的唯一编号。

线程组标识符 tgid: 线程组的所有者编号，即其所属进程的唯一编号。由于在 Linux 内核中，线程也被实现为一个进程（轻量级进程），即也拥有一个 `task_struct` 结构体。因此，当该结构体标识一个进程时，`tgid` 即为进程编号 `pid`；结构体标识一个线程时，`tgid` 为拥有该线程的进程编号。

1.2 进程间关系

```
struct task_struct __rcu *real_parent; /* real parent process */
struct task_struct __rcu *parent; /* recipient of SIGCHLD, wait4() reports */
/*
 * children/sibling forms the list of my natural children
 */
struct list_head children; /* list of my children */
struct list_head sibling; /* linkage in my parent's children list */
struct task_struct *group_leader; /* threadgroup leader */
```


父进程 parent: 进程的父进程。

真实父进程 real_parent: 进程当前的父进程。在一些特殊情况下，进程当前的父进程可能并非其原始父进程，例如 gdb 调试某程序时，其 real_parent 会被设置为 gdb 进程，而其 parent 的值并不一定。

子进程列表 children: 维护其子进程的双向链表，便于对其子进程进行遍历操作。

兄弟进程列表 sibling: 维护其兄弟进程（拥有同一个父进程的进程集合）的双向链表，便于进行遍历操作。

组代表 group_leader: 即 tgid 标识的进程编号对应的 task_struct 结构体。若当前结构体 this 标识一个线程，那么通过 this->group_leader 可以访问拥有这个线程的进程。

1.3 调度参数

```
int prio, static_prio, normal_prio;
unsigned int rt_priority;
const struct sched_class *sched_class;
struct sched_entity se;
struct sched_rt_entity rt;
unsigned int policy;
int nr_cpus_allowed;
cpumask_t cpus_allowed;
```

优先级 prio、静态优先级 static_prio、常规优先级 normal_prio: 调度时可能使用的各种优先级信息。prio 为进程的动态优先级，一方面通过其取值范围可以区分进程类型（0~99 为实时进程，100~139 为普通进程），另一方面，其值越小，代表进程优先级越高，宏观上看更“容易”被调度到。static_prio 是自父进程继承的优先级，用于计算进程的初始时间片长度与动态优先级。normal_prio 是进程“应有”的动态优先级，某些时候，进程的动态优先级可能被暂时提升，提升过后应当恢复到的优先级即为 normal_prio。

实时进程优先级 rt_priority: 实时进程的优先级。

调度类 sched_class: 进程所使用的调度类。

调度实体 sched_entity: 进程所属的调度实体。

实时调度实体 sched_rt_entity: 进程所属的实时调度实体。

调度策略 policy: 进程使用的调度策略（抢占式/非抢占式）。

可用 cpu 数 nr_cpus_allowed: 指明该进程可在多少个 cpu 上进行调度。

可用 cpu 掩码 cpus_allowed: 指明该进程可在哪些 cpu 上进行调度。

1.4 文件和内存指针

```
struct mm_struct *mm, *active_mm;  
/* filesystem information */  
struct fs_struct *fs;  
/* open file information */  
struct files_struct *files;
```

地址空间 mm, **活跃地址空间 active_mm**: 分别指向进程地址空间和最近使用过的地址空间（利用时间局部性）。

文件系统指针 fs: 指向文件系统的信息结构体。

文件指针 files: 已打开的文件指针。

1.5 信号

```
struct signal_struct *signal;  
struct sighand_struct *sighand;  
  
sigset_t saved_sigmask; /* restored if set_restore_sigmask() was used */  
struct sigpending pending;
```

信号描述符 signal: 指向进程信号结构体的指针。

信号处理器 sighand: 指向进程信号处理器的指针。

信号掩码 sigmask: 表示进程可以接收到的信号。其二进制表示的每一位代表一个不同信号的接收使能，置 0 表示可以接收，置 1 表示不能接收。

挂起信号 pending: 保存所有已经接收但还未处理的信号。

1.6 其它

task_struct 结构体除了上述成员之外，还有非常多的其它成员，包括 trace 相关的成员、NUMA 相关的成员等。由于这些成员在以下部分不为重点（以及笔者对这些并不了解），故不再对它们做更多介绍。

2 进程的组织结构

为了对进程进行有效的管理，Linux 系统将进程组织成几种数据结构，便于应对不同的操

作。以下对有代表性的进程组织结构进行介绍：

2.1 双向链表

在 `task_struct` 结构体中有成员 `struct list_head tasks`，该成员为一双向链表的节点。Linux 系统将所有进程用一个双向链表连接，并定义了一个宏 `for_each_process`：

```
#define for_each_process(p) \
    for (p = &init_task ; (p = next_task(p)) != &init_task ; )
```

其中 `init_task` 为链表头节点。追踪其初始化过程，发现这个节点的初始化在 `init_task.c` 中最终完成。经过一系列曲折的探索与发现，可以找到其中初始化 `pid` 的数据路径为：

`INIT_TASK()->INIT_PID_LINK()->init_struct_pid->INIT_STRUCT_PID->ATOMIC_INIT(1)`。

2.2 散列表

有一些操作是通过进程号进行的（比如 `kill pid` 结束进程），因此我们还需要一个能快速将进程号转换成 `task_struct` 结构体地址的方法。在 Linux 中，进程以 `pid` 为关键词，通过散列函数组织成散列表，使用开散列方式解决冲突。当需要通过 `pid` 快速定位进程时，先通过散列函数获取 `pid` 对应的散列表下标，然后遍历放在该下标的开链表，即可快速定位进程。

```
/* PID/PID hash table linkage. */
```

```
struct pid_link pids[PIDTYPE_MAX];
```

在 `task_struct` 结构体中，`pids` 即为一个散列表的入口，其中 `struct pid_link` 包含两个成员：`hlist_node` 与 `pid` 结构体，前者为散列表的开链表节点，后者为该节点保存的 `pid` 值。

3 进程状态设置

```
#define TASK_RUNNING          0
#define TASK_INTERRUPTIBLE    1
#define TASK_UNINTERRUPTIBLE  2
#define __TASK_STOPPED        4
#define __TASK_TRACED          8
/* in task->exit_state */
#define EXIT_DEAD              16
#define EXIT_ZOMBIE            32
```

```

#define EXIT_TRACE      (EXIT_ZOMBIE | EXIT_DEAD)
/* in tsk->state again */
#define TASK_DEAD      64
#define TASK_WAKEKILL   128
#define TASK_WAKING     256
#define TASK_PARKED     512
#define TASK_NOLOAD     1024
#define TASK_STATE_MAX  2048

#define __set_task_state(tsk, state_value) \
    do { \
        (tsk)->task_state_change = _THIS_IP_; \
        (tsk)->state = (state_value); \
    } while (0)
#define set_task_state(tsk, state_value) \
    do { \
        (tsk)->task_state_change = _THIS_IP_; \
        smp_store_mb((tsk)->state, (state_value)); \
    } while (0)

```

在<include/linux/sched.h>中定义了进程的合法状态，其中数值较小的几个状态和熟知的 Linux 状态模型基本相符。此外，文件中还定义了设置进程状态的宏，通过搜索该宏的调用时机发现，内核代码大部分时候只会在将进程设置为 TASK_UNINTERRUPTIBLE 状态时才调用该宏，其余时候均直接使用赋值语句改变进程状态。

4 Linux 的调度架构

4.1 从调度时机到调度器架构

4.1.1 调度函数

通过课上的学习，我们大约知道调度程序何时工作。

- 在 3(b) 中做了这样一个实验：启动一个进程，在一个循环 10000 次的结构中调用 `usleep(1)`，然后调用新添加的系统调用查看进程信息，发现 `usleep()` 会引发一次**非抢占式**的上下文切换。搜索发现 `usleep` 调用了内核中的 `nanosleep`，在 <kernel/time/hrtimer.c>中找到了函数 `do_nanosleep()` 中的上下文切换部分：

```
set_current_state(TASK_INTERRUPTIBLE);
```

```

if (likely(t->task))
    freezable_schedule();

```

进一步查看 `freezable_schedule` 函数，发现其中主要调用了 `<kernel/sched/core.c>` 中的 `schedule` 函数，这是 Linux 调度器的主要函数之一。

在 3(b) 中还做了这样一个实验：启动一个进程，在循环 10 亿次的结构中累加一个变量，然后调用新添加的系统调用查看进程信息，发现进程被数次**抢占式**切换上下文。在 `sourceinsight` 中搜索关键词 `schedule`，在 `core.c` 中发现了函数 `scheduler_tick`。按照注释，这个函数由系统时钟定时触发，检查是否需要调度。

4.1.2 调度类

在 `scheduler_tick` 函数中有这样一条语句：

```
curr->sched_class->task_tick(rq, curr, 0);
```

而在 `schedule` 函数中，同样通过多层函数调用，最终调用了 `curr->sched_class` 成员中的 `dequeue_task` 函数。因此可以推断：在 Linux 系统中，调度器设计为两层结构，其中 `schedule` 和 `scheduler_tick` 两个调度程序负责进行 CPU 的上下文切换，而 `sched_class` 类中的成员函数负责进行进程的选择并维护调度需要的数据结构。

搜索关键词 `sched_class`，发现这只是一个未实现的调度类：

```
struct sched_class {  
    void (*enqueue_task) (struct rq *rq, struct task_struct *p, int flags);  
    void (*dequeue_task) (struct rq *rq, struct task_struct *p, int flags);  
    struct task_struct * (*pick_next_task) (struct rq *rq,  
                                             struct task_struct *prev);  
    void (*task_tick) (struct rq *rq, struct task_struct *p, int queued);  
};
```

因此，Linux 内核代码还应在别处实现了具体的调度类以及各种成员函数。在搜索中发现了在 `<kernel/sched/fair.c>` 中定义的 `fair_sched_class` 调度类，我们又回到了最初的起点。

5 CFS 调度算法

5.1 思路

CFS 调度算法和 RR 调度算法有一定的类似。RR 期望在一个调度周期内所有被调度的进程轮流运行且时间相同，而 CFS 期望在一个调度周期内所有被调度的**实体按某种策略运行**且**虚**

拟运行时间相同。它以 nice 值为依据设置不同进程的运行时间权重，通过实际运行时间和权重的综合计算得出虚拟运行时间，权重越大，虚拟运行时间就走得越“慢”。因此，若所有调度实体的虚拟运行时间相同，那么权重大的进程的实际运行时间就更长。

以下先介绍 CFS 调度算法的主要结构体，再介绍调度算法的主要流程和策略。

5.2 CFS 调度算法使用的主要结构

5.2.1 调度实体

```
struct sched_entity {
    struct load_weight  load;          /* for load-balancing */
    struct rb_node      run_node;
    struct list_head    group_node;
    unsigned int        on_rq;
    u64                 vruntime;
#ifdef CONFIG_FAIR_GROUP_SCHED
    struct sched_entity *parent;
    /* rq on which this entity is (to be) queued: */
    struct cfs_rq        *cfs_rq;
    /* rq "owned" by this entity/group: */
    struct cfs_rq        *my_q;
#endif
};
```

在CFS调度算法中，被调度的单位为调度实体，可以是一个进程，也可以是一组进程。各实体的虚拟运行时间相同，说的是实体内所有进程的总虚拟运行时间相同。

5.2.2 就绪队列

```
/* CFS-related fields in a runqueue */
struct cfs_rq {
    struct load_weight load;

    u64 exec_clock;
    u64 min_vruntime;
#ifdef CONFIG_64BIT
    u64 min_vruntime_copy;
#endif

    struct rb_root tasks_timeline;
    struct rb_node *rb_leftmost;
    struct sched_entity *curr, *next, *last, *skip;
```

```

#endif /* CONFIG_FAIR_GROUP_SCHED */
#endif /* CONFIG_SMP */

#ifdef CONFIG_FAIR_GROUP_SCHED
    struct rq *rq; /* cpu runqueue to which this cfs_rq is attached */
    struct task_group *tg; /* group that "owns" this runqueue */
#endif /* CONFIG_FAIR_GROUP_SCHED */
};

```

只有在就绪队列中的实体才可能被调度。就绪队列是一个优先队列，其内部使用了红黑树的数据结构，可以高效进行增删改查的操作。

5.3 CFS 调度算法的主要流程

```

static struct task_struct *
pick_next_task_fair(struct rq *rq, struct task_struct *prev)
{
    struct cfs_rq *cfs_rq = &rq->cfs;
    struct sched_entity *se;
    struct task_struct *p;

    cfs_rq = &rq->cfs;
    put_prev_task(rq, prev);

    do {
        se = pick_next_entity(cfs_rq, NULL);
        set_next_entity(cfs_rq, se);
        cfs_rq = group_cfs_rq(se);
    } while (cfs_rq);

    p = task_of(se);

    if (hrtick_enabled(rq))
        hrtick_start_fair(rq, p);

    return p;
}

```

我们知道：在调度算法中，选择换入的进程是最重要的部分之一。事实上，CFS 选择换入进程的方式一目了然：从就绪队列中选取虚拟运行时间最小的节点，若该节点为一个进程组，那么继续从该进程组的就绪队列中选取虚拟运行时间最小的节点。递归进行这一操作，直到选择的调度实体为一个进程。

5.4 Q & A

5.4.1 简述进程优先级、nice 值和权重之间的关系

进程优先级是内核用来表示进程执行迫切性的数值，其中 0~99 代表实时进程，100~139 代表普通进程。

nice 值范围在-20~+19 之间，每次进入调度序列时被设置，直接加在进程优先级上，用于修正进程的优先级，是一个形象的命名，表示一个进程“谦让”的程度。nice 值越高，进程在该次运行的优先级就越低，也就是更“谦让”。

权重是对应 nice 值的，实际运行时间到虚拟运行时间的转化比例。在 Linux 系统中，转化表设置在 prio_to_weight 中：

```
static const int prio_to_weight[40] = {
/* -20 */    88761,    71755,    56483,    46273,    36291,
/* -15 */    29154,    23254,    18705,    14949,    11916,
/* -10 */    9548,    7620,    6100,    4904,    3906,
/* -5 */     3121,    2501,    1991,    1586,    1277,
/* 0 */      1024,    820,    655,    526,    423,
/* 5 */       335,    272,    215,    172,    137,
/* 10 */      110,    87,    70,    56,    45,
/* 15 */       36,    29,    23,    18,    15,
};
```

相邻两级的权重差距都约为 25%。

5.4.2 vruntime 的基本思想是什么

正如上面所说，CFS 希望 nice 值高的实体获得更高的实际运行时间比例，因此将实际运行时间乘一个权重比例，转化成虚拟运行时间 vruntime。平衡各实体的 vruntime，实际上就平衡了各实体的加权运行时间，那么 nice 值高的实体就获得了更高的实际运行比例。

5.4.3 vruntime 是如何计算的，何时得到更新

同样如上面所说，vruntime 的计算方法即为实际运行时间乘一个权重比例，它的计算函数在<kernel/sched/fair.c>中：

```
static u64 __calc_delta(u64 delta_exec, unsigned long weight, struct load_weight *lw)
{
    u64 fact = scale_load_down(weight);
    int shift = WMULT_SHIFT;
```



```

__update_inv_weight(lw);

if (unlikely(fact >> 32)) {
    while (fact >> 32) {
        fact >>= 1;
        shift--;
    }
}

/* hint to use a 32x32->64 mul */
fact = (u64)(u32)fact * lw->inv_weight;

while (fact >> 32) {
    fact >>= 1;
    shift--;
}

return mul_u64_u32_shr(delta_exec, fact, shift);
}

```

计算公式为： $vruntime = time * NICE_0_LOAD / load$ ，其中 $NICE_0_LOAD$ 为 nice 值为 0 的时候对应的权重（即 1024）。由于除法运算速度慢，这里预先计算了 $2^{32}/prio_to_weight$ 的值 $prio_to_wmult$ ，这样所有运算都变成了乘法和位移操作，可以快速完成。

更新 $vruntime$ 的函数是 `fair.c` 的 `update_curr` 函数，它通过计算时间增量来计算 $vruntime$ 的增量。在进程创建、进出调度队列、周期调度器调度的时候，这个函数都会被调用。

5.4.4 min_vruntime 有什么作用

$min_vruntime$ 是 CFS 调度队列中 $vruntime$ 的最小值。一个新创建的进程进入队列，或是睡眠一段时间后刚被唤醒时，它的 $vruntime$ 和一直就绪的进程相比会有一定的落后，若此时维持 $vruntime$ 值不变，那么这个进程就会一直被调度，影响公平性。因此，这些情况发生时，进程的 $vruntime$ 会进行适当的调整，例如进程创建时，调度类的 `task_fork_fair` 函数会调用 `place_entity` 函数，将 $vruntime$ 先设为 $min_vruntime$ ，再在这个基础上进行适当的调整。在下面的实验中，我们可以观察到：进程的总运行时间和虚拟运行时间存在一个几乎相等的差值，这一定程度上也印证了进程创建时将 $vruntime$ 设为就绪队列的 $min_vruntime$ 这一策略。

另外，在多核处理器中，还涉及到一个新的需要调节 $vruntime$ 的场景，即任务迁移。

当进程从一个核心迁移到另一个核心时，由于两个核心的 `vruntime` 分布可能不同，若进程从 `vruntime` 小的核心迁移到较大的核心，就可能占便宜。CFS 应对这个问题的思想是：进程移动时，保留其“相对 `vruntime`”，具体做法是：进程离开队列时，减掉这个队列的 `min_vruntime`；进入队列时，加上这个队列的 `min_vruntime`，保留进程相对于一个队列的 `vruntime`，一定程度上保留了公平性。

b) 添加系统调用：用户层测试程序 `mycall.c`

1. 运行 `screenfetch`

```
hsy@hsy-VirtualBox:~$ screenfetch
      ./+o+-
      yyyyy- -yyyyyy+
      ://+///// -yyyyyyo
      .++ .:/++++++/- .+sss/`
      .:++o: /+++++++/!:-:/-
      o:++o:++ .`..`.-/oo+++++/
      .:++o:++o/.`      +sss0o+/
      .++/+:+oo+o:`      /sss0oo.
      /+++//+:`oo+o      /:--:.
      \+/+o+++`o+o      ++//.
      .++ .o+++oo+:`      /dddhhh.
      .+.o+oo:.`      `oddhhhh+
      \.++o+o`      .:ohdhhhh+
      :o+++`ohhhhhhhyo++os:
      .o:`syhhhhh/ .oo++o`
      /osyyyyyyo++ooo+++/
      +oo++o\
      `oo++.
```

```
hsy@hsy-VirtualBox
OS: Ubuntu 16.04 xenial
Kernel: x86_64 Linux 4.16.1
Uptime: 1h 33m
Packages: 1785
Shell: bash 4.3.48
Resolution: 800x600
DE: Unity 7.4.5
WM: Compiz
WM Theme: Ambiance
GTK Theme: Ambiance [GTK2/3]
Icon Theme: ubuntu-mono-dark
Font: Ubuntu 11
CPU: Intel Core i7-7500U CPU @ 2.904GHz
RAM: 675MiB / 1993MiB
```

2. 添加系统调用

syscall_64.tbl			
/usr/src/linux-4.16.1/arch/x86/entry/syscalls			
打开(O)			保存(S)
308	common	setns	sys_setns
309	common	getcpu	sys_getcpu
310	64	process_vm_readv	sys_process_vm_readv
311	64	process_vm_writev	sys_process_vm_writev
312	common	kcmp	sys_kcmp
313	common	finit_module	sys_finit_module
314	common	sched_setattr	sys_sched_setattr
315	common	sched_getattr	sys_sched_getattr
316	common	renameat2	sys_renameat2
317	common	seccomp	sys_seccomp
318	common	getrandom	sys_getrandom
319	common	memfd_create	sys_memfd_create
320	common	kexec_file_load	sys_kexec_file_load
321	common	bpf	sys_bpf
322	64	execveat	sys_execveat/ptregs
323	common	userfaultfd	sys_userfaultfd
324	common	membarrier	sys_membarrier
325	common	mlock2	sys_mlock2
326	common	copy_file_range	sys_copy_file_range
327	64	preadv2	sys_preadv2
328	64	pwritev2	sys_pwritev2
329	common	pkey_mprotect	sys_pkey_mprotect
330	common	pkey_alloc	sys_pkey_alloc
331	common	pkey_free	sys_pkey_free
332	common	statx	sys_statx
333	64	print_msg	sys_print_msg

打开(O) ▾

🔍

syscalls.h

/usr/src/linux-4.16.1/arch/x86/include/asm

保存(S)

```

/*
 * syscalls.h - Linux syscall interfaces (arch-specific)
 *
 * Copyright (c) 2008 Jaswinder Singh Rajput
 *
 * This file is released under the GPLv2.
 * See the file COPYING for more details.
 */

#ifndef _ASM_X86_SYSCALLS_H
#define _ASM_X86_SYSCALLS_H

#include <linux/compiler.h>
#include <linux/linkage.h>
#include <linux/signal.h>
#include <linux/types.h>

/* Common in X86_32 and X86_64 */
/* kernel/ioport.c */
asmlinkage long sys_ioperm(unsigned long, unsigned long, int);
asmlinkage long sys_iopl(unsigned int);
asmlinkage long sys_print_msg(void);
/* kernel/ldt.c */
asmlinkage long sys_modify_ldt(int, void __user *, unsigned long);

/* kernel/signal.c */
asmlinkage long sys_rt_sigreturn(void);

```

打开(O) ▾

🔍

sys.c

/usr/src/linux-4.16.1/kernel

保存(S)

```

        return 0;
}

asmlinkage long sys_print_msg(void){
#define PN(F) \
    printk("%-45s:%14ld.%06ld\n", #F, (long long)(p->F/1000000), (long)(p->F%1000000))
#define P(F) \
    printk("%-45s:%21ld\n", #F, (long long)p->F)
    struct task_struct *p=current;
    PN(se.exec_start);
    PN(se.vruntime);
    PN(se.sum_exec_runtime);
    P(se.nr_migrations);
    printk("%-45s:%21ld\n", "nr_switches", (long long)(p->nvcsw+p->nivcsw));
    printk("%-45s:%21ld\n", "nr_voluntary_switches", (long long)p->nvcsw);
    printk("%-45s:%21ld\n", "nr_involuntary_switches", (long long)p->nivcsw);
    P(se.load.weight);
    P(se.avg.load_sum);
    P(se.avg.util_sum);
    P(se.avg.load_avg);
    P(se.avg.util_avg);
    P(se.avg.last_update_time);
#undef PN
#undef P
    return 1;
}

```

2.1 先启动一个进程，在一个循环 10000 次的结构中调用 usleep(1)

```
打开(O)  mycall.c
~/

#include <stdio.h>
#include <linux/kernel.h>
#include <sys/syscall.h>
#include <unistd.h>

int main() {
    int i=0;
    for (i=0;i<=10000;i++)
        usleep(1);
    long a = syscall(333);
    printf("Syscall returns %d\n", a);
    return 0;
}
```

```
sy@hsy-VirtualBox:~$ ./mycall
syscall returns 1
sy@hsy-VirtualBox:~$ dmesg
6808.029829] se.exec_start : 6808029.813021
6808.029830] se.vruntime : 146.735237
6808.029831] se.sum_exec_runtime : 108.715921
6808.029831] se.nr_migrations : 0
6808.029832] nr_switches : 9967
6808.029833] nr_voluntary_switches : 9967
6808.029833] nr_involuntary_switches : 0
6808.029834] se.load.weight : 1048576
6808.029834] se.avg.load_sum : 7844
6808.029835] se.avg.util_sum : 7724892
6808.029835] se.avg.load_avg : 171
6808.029836] se.avg.util_avg : 164
6808.029836] se.avg.last_update_time : 6808029812736
```

2.2 先启动一个进程，在循环 10 亿次的结构中累加一个变量

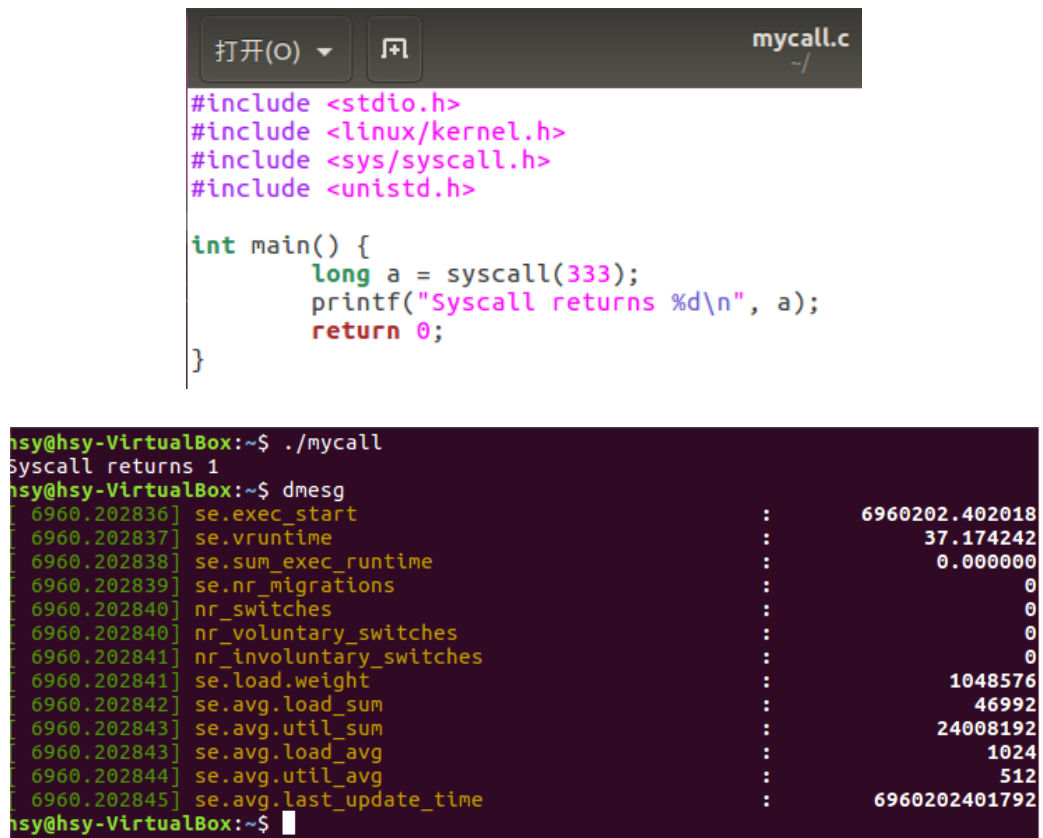
```
打开(O)  mycall.c
~/

#include <stdio.h>
#include <linux/kernel.h>
#include <sys/syscall.h>
#include <unistd.h>

int main() {
    int i=0;
    int j=0;
    for (i=0;i<=1000000000;i++)
        j++;
    long a = syscall(333);
    printf("Syscall returns %d\n", a);
    return 0;
}
```

```
sy@hsy-VirtualBox:~$ ./mycall
syscall returns 1
sy@hsy-VirtualBox:~$ dmesg
6648.301285] se.exec_start : 6648297.938901
6648.301287] se.vruntime : 2072.792164
6648.301288] se.sum_exec_runtime : 2036.152403
6648.301289] se.nr_migrations : 0
6648.301290] nr_switches : 57
6648.301290] nr_voluntary_switches : 0
6648.301291] nr_involuntary_switches : 57
6648.301291] se.load.weight : 1048576
6648.301292] se.avg.load_sum : 47727
6648.301292] se.avg.util_sum : 48798244
6648.301293] se.avg.load_avg : 1023
6648.301293] se.avg.util_avg : 1022
6648.301294] se.avg.last_update_time : 6648297937920
```

2.3 在调用前不执行任何指令



The image shows a code editor window titled 'mycall.c' with the following C code:

```
#include <stdio.h>
#include <linux/kernel.h>
#include <sys/syscall.h>
#include <unistd.h>

int main() {
    long a = syscall(333);
    printf("Syscall returns %d\n", a);
    return 0;
}
```

Below the code editor is a terminal window showing the execution of the program. The user runs './mycall' and the output is 'syscall returns 1'. Then, the user runs 'dmesg' to view kernel messages, which shows a list of statistics for the current process (PID 6960).

```
hsy@hsy-VirtualBox:~$ ./mycall
syscall returns 1
hsy@hsy-VirtualBox:~$ dmesg
6960.202836] se.exec_start           :          6960202.402018
6960.202837] se.vruntime             :          37.174242
6960.202838] se.sum_exec_runtime          :          0.000000
6960.202839] se.nr_migrations                :          0
6960.202840] nr_switches                      :          0
6960.202840] nr_voluntary_switches           :          0
6960.202841] nr_involuntary_switches         :          0
6960.202841] se.load.weight                  :        1048576
6960.202842] se.avg.load_sum                 :          46992
6960.202843] se.avg.util_sum                 :       24008192
6960.202843] se.avg.load_avg                 :          1024
6960.202844] se.avg.util_avg                 :           512
6960.202845] se.avg.last_update_time         :       6960202401792
hsy@hsy-VirtualBox:~$
```