

操作系统大作业1

陈政培 17363011 智能科学与技术 智科1班

操作系统大作业1

1. 哲学家就餐问题

原理和具体实现

死锁解决方案

程序运行结果

2. 生产者消费者问题

原理和具体实现

Buffer结构体

生产者和消费者逻辑（伪代码逻辑）

共享内存

踩坑

程序运行结果

3. Linux内核实验（Linux 4.0或以上）

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

Linux进程的基本结构

进程状态

state

进程表示符 PID

其他结构

进程调度

优先级

调度策略

结构体其他内容

判断标志

时间

信号处理

CPU调度架构

CPU调度准则

进程与调度相关数据结构

相关结构体

相关调度类

运行队列

优先级处理

调度初始化

周期调度器

其他

CFS调度算法

原理

CFS类的结构和算法

周期调度

再次排队的实现

选择最优进程使用CPU资源

其他

作业问题解答

进程优先级、nice值和权重的关系

CFS调度器中的vruntime的基本思想是什么？是如何计算的？何时得到更新？其中的

min_vruntime有什么作用？

基本思想：

计算与更新:

`min_vruntime` 的作用

b 添加系统调用: 用户层测试程序 `mycall.c`

实现步骤

结果展示

参考文献

1. 哲学家就餐问题

原理和具体实现

参考教材, 使用 `pthread_cond_t` 条件变量和 `pthread_mutex_t` 互斥量, 达到加锁的功能

将5根筷子作为5个互斥量, 记录筷子的使用情况

```
pthread_mutex_t chopstick[6] ;

int i;
for (i = 0; i < 5; i++)
    pthread_mutex_init(&chopstick[i], NULL);
```

5个哲学家对应五个线程, 每个线程拥有两个互斥量的掌控权

死锁解决方案

当一个哲学家感到饥饿, 将首先拿起左手的筷子, 如果此时右手的筷子并不处于空闲状态, 则将左手筷子放回桌子上并进入等待状态。等待状态中, 持续等待右手筷子出现, 当两只筷子都空闲时开始吃饭

```
if (pthread_mutex_trylock(&chopstick[right]) == EBUSY){ //拿起右手的筷子
    pthread_mutex_unlock(&chopstick[left]); //如果右边筷子被拿走放下左手的筷子
    printf("Philosopher %c is waiting.\n", phi);
    continue;
}
```

避免死锁的核心就是这个 `if` 判断语句, 如果注释掉, 则可以观察到死锁状态

程序运行结果

```
osc@ubuntu:~/test$ make dph
gcc -Wall -o dph dph.c -lpthread -lm -lrt
osc@ubuntu:~/test$ ./dph
Philosopher E is thinking. (for 1s)
Philosopher D is thinking. (for 1s)
Philosopher C is thinking. (for 1s)
Philosopher B is thinking. (for 1s)
Philosopher A is thinking. (for 1s)
Philosopher E is hungry.
Philosopher E is eating. (for 4s)
Philosopher D is hungry.
Philosopher D is waiting.
Philosopher C is hungry.
Philosopher C is eating. (for 5s)
Philosopher A is hungry.
Philosopher B is hungry.
Philosopher B is waiting.
Philosopher E is full.
Philosopher E is thinking. (for 3s)
Philosopher A is eating. (for 3s)
Philosopher C is full.
Philosopher C is thinking. (for 5s)
Philosopher D is eating. (for 5s)
Philosopher D is full.
Philosopher D is thinking. (for 2s)
Philosopher E is hungry.
Philosopher E is waiting.
Philosopher A is full.
Philosopher A is thinking. (for 5s)
Philosopher B is eating. (for 5s)
Philosopher E is eating. (for 3s)
Philosopher C is hungry.
Philosopher C is eating. (for 3s)
Philosopher B is full.
Philosopher B is thinking. (for 2s)
Philosopher E is full.
Philosopher E is thinking. (for 3s)
Philosopher A is hungry.
Philosopher A is eating. (for 3s)
Philosopher D is hungry.
Philosopher D is eating. (for 2s)
Philosopher B is hungry.
Philosopher B is eating. (for 5s)
^C
```

哲学家有等待或者立即吃饭的状态，并且没有出现死锁

2. 生产者消费者问题

原理和具体实现

结合教材提示，使用 `empty` 和 `full` 两个标准的计数信号量，并且通过 `mutex` 互斥量保护对缓冲区的插入和删除操作。并将仓库存放的数据 `value`、信号量、互斥量以及工作计数、当前队列指针统一封装为 `Buffer` 结构体

Buffer结构体

```
typedef struct Buffer
{
    int *ptr;
    int a;
    int counter;
    int current;
    sem_t *empty;
    sem_t *full;
    pthread_mutex_t mutex;
}Buffer;
```

生产者和消费者逻辑（伪代码逻辑）

```
produce {
    while( 1 ) {
        wait( space ); // 等待缓冲区有空闲位置
        wait( mutex );

        buffer.push( item, in ); // 加入新资源
        in = ( in + 1 ) % 20;

        signal( mutex );
        signal( items );
    }
}

consume {
    while( 1 ) {
        wait( items ); // 等待缓冲区有资源可以使用
        wait( mutex ); //

        buffer.pop( out ); // 将资源取走
        out = ( out + 1 ) % 10;

        signal( mutex );
        signal( space );
    }
}
```

共享内存

通过系统自带的 `shm` 相关头文件，创建一个统一名称统一大小的内存空间

```
const int BUFFER_SIZE = sizeof(int) * 20;
const char *name = "SharedMemory";

int shm_fd;
void *ptr;

shm_fd = shm_open(name, O_CREAT | O_RDWR, 0666);

ftruncate(shm_fd, BUFFER_SIZE);

ptr = mmap(0, BUFFER_SIZE, PROT_READ | PROT_WRITE, MAP_SHARED, shm_fd, 0);
if (ptr == MAP_FAILED) {
    printf("Map failed\n");
}
```

```
    return -1;
}
```

踩坑

共享内存申请时，`BUFFER_SIZE` 不能直接赋值20，申请的空间会在实际使用中会出现数组越界的情况。应当赋值为 `sizeof(int) * 20` 以此按照 `int` 类型的大小申请内存

程序运行结果

在 `main` 函数中定义了传递参数 `argv`，用来获得负指数分布的控制参数

```
int main(int argc, char **argv)
```

负指数分布获得时间间隔

```
y = ((double)rand()/RAND_MAX);
double a = buffer->a;
usleep(1000000*(log(y/a)/-a));
```

运行 `./prod 1` 和 `./cons 1`

```
osc@ubuntu:~/test$ make cons
gcc -Wall -o cons cons.c -lpthread -lm -lrt
osc@ubuntu:~/test$ ./prod 1
[pid: 3359] [tid: 3362] [value: 1714636915]
[pid: 3359] [tid: 3360] [value: 424238335]
[pid: 3359] [tid: 3362] [value: 1649760492]
[pid: 3359] [tid: 3361] [value: 1189641421]
[pid: 3359] [tid: 3360] [value: 1350490027]
[pid: 3359] [tid: 3362] [value: 1102520059]
[pid: 3359] [tid: 3362] [value: 1967513926]
[pid: 3359] [tid: 3361] [value: 1540383426]
[pid: 3359] [tid: 3362] [value: 1303455736]
[pid: 3359] [tid: 3360] [value: 521595368]
[pid: 3359] [tid: 3361] [value: 1726956429]
[pid: 3359] [tid: 3360] [value: 861021530]
[pid: 3359] [tid: 3361] [value: 233665123]
[pid: 3359] [tid: 3361] [value: 468703135]
[pid: 3359] [tid: 3361] [value: 1801979802]
[pid: 3359] [tid: 3362] [value: 635723058]
[pid: 3359] [tid: 3360] [value: 1125898167]
[pid: 3359] [tid: 3362] [value: 2089018456]
[pid: 3359] [tid: 3361] [value: 1656478042]
[pid: 3359] [tid: 3360] [value: 1653377373]
[pid: 3359] [tid: 3361] [value: 1914544919]
[pid: 3359] [tid: 3362] [value: 756898537]
[pid: 3359] [tid: 3360] [value: 1973594324]
[pid: 3359] [tid: 3362] [value: 2038664370]
[pid: 3359] [tid: 3361] [value: 184803526]
[pid: 3359] [tid: 3362] [value: 1424268980]
[pid: 3359] [tid: 3362] [value: 749241873]
[pid: 3359] [tid: 3361] [value: 42999170]
[pid: 3359] [tid: 3360] [value: 135497281]
[pid: 3359] [tid: 3361] [value: 2084420925]
[pid: 3359] [tid: 3361] [value: 1827336327]
[pid: 3359] [tid: 3362] [value: 1159126505]
[pid: 3359] [tid: 3360] [value: 1632621729]
[pid: 3359] [tid: 3361] [value: 1433925857]
[pid: 3359] [tid: 3362] [value: 84353895]
[pid: 3359] [tid: 3360] [value: 2001100545]
[pid: 3359] [tid: 3360] [value: 1548233367]
[pid: 3359] [tid: 3361] [value: 1585990364]
[pid: 3359] [tid: 3362] [value: 760313750]
[pid: 3359] [tid: 3361] [value: 356426808]
[pid: 3359] [tid: 3362] [value: 1889947178]
[pid: 3359] [tid: 3362] [value: 709393584]
[pid: 3359] [tid: 3360] [value: 1918502651]
[pid: 3359] [tid: 3361] [value: 1474612399]
[pid: 3359] [tid: 3361] [value: 1264095060]
[pid: 3359] [tid: 3361] [value: 1843993368]
[pid: 3359] [tid: 3360] [value: 1984210012]
osc@ubuntu:~/test$ ./cons 1
123[pid: 3363] [tid: 3366] [value: 1714636915]
[pid: 3363] [tid: 3364] [value: 424238335]
[pid: 3363] [tid: 3364] [value: 1649760492]
[pid: 3363] [tid: 3366] [value: 1189641421]
[pid: 3363] [tid: 3365] [value: 1350490027]
[pid: 3363] [tid: 3365] [value: 1102520059]
[pid: 3363] [tid: 3366] [value: 1967513926]
[pid: 3363] [tid: 3364] [value: 1540383426]
[pid: 3363] [tid: 3366] [value: 1303455736]
[pid: 3363] [tid: 3365] [value: 521595368]
[pid: 3363] [tid: 3366] [value: 1726956429]
[pid: 3363] [tid: 3364] [value: 861021530]
[pid: 3363] [tid: 3364] [value: 233665123]
[pid: 3363] [tid: 3364] [value: 468703135]
[pid: 3363] [tid: 3366] [value: 1801979802]
[pid: 3363] [tid: 3364] [value: 635723058]
[pid: 3363] [tid: 3365] [value: 1125898167]
[pid: 3363] [tid: 3366] [value: 2089018456]
[pid: 3363] [tid: 3365] [value: 1656478042]
[pid: 3363] [tid: 3365] [value: 1653377373]
[pid: 3363] [tid: 3364] [value: 1914544919]
[pid: 3363] [tid: 3365] [value: 756898537]
[pid: 3363] [tid: 3364] [value: 1973594324]
[pid: 3363] [tid: 3365] [value: 2038664370]
[pid: 3363] [tid: 3366] [value: 184803526]
[pid: 3363] [tid: 3364] [value: 1424268980]
[pid: 3363] [tid: 3364] [value: 749241873]
[pid: 3363] [tid: 3365] [value: 42999170]
[pid: 3363] [tid: 3364] [value: 135497281]
[pid: 3363] [tid: 3366] [value: 1827336327]
[pid: 3363] [tid: 3365] [value: 1159126505]
[pid: 3363] [tid: 3364] [value: 1632621729]
[pid: 3363] [tid: 3365] [value: 1433925857]
[pid: 3363] [tid: 3364] [value: 84353895]
[pid: 3363] [tid: 3364] [value: 2001100545]
[pid: 3363] [tid: 3366] [value: 1548233367]
[pid: 3363] [tid: 3366] [value: 1585990364]
[pid: 3363] [tid: 3365] [value: 760313750]
[pid: 3363] [tid: 3365] [value: 1889947178]
[pid: 3363] [tid: 3365] [value: 709393584]
[pid: 3363] [tid: 3364] [value: 1918502651]
[pid: 3363] [tid: 3366] [value: 1474612399]
[pid: 3363] [tid: 3366] [value: 1264095060]
[pid: 3363] [tid: 3366] [value: 1843993368]
^C
osc@ubuntu:~/test$
```

运行 `./prod 5` 和 `./cons 4`

```

[pid: 3384] [tid: 3386] [value: 1653377373]
[pid: 3384] [tid: 3385] [value: 1914544919]
[pid: 3384] [tid: 3387] [value: 756898537]
[pid: 3384] [tid: 3386] [value: 1973594324]
[pid: 3384] [tid: 3387] [value: 2038664370]
[pid: 3384] [tid: 3385] [value: 184803526]
[pid: 3384] [tid: 3387] [value: 1424268980]
[pid: 3384] [tid: 3387] [value: 749241873]
[pid: 3384] [tid: 3386] [value: 42999170]
[pid: 3384] [tid: 3385] [value: 135497281]
[pid: 3384] [tid: 3386] [value: 2084420925]
[pid: 3384] [tid: 3385] [value: 1827336327]
[pid: 3384] [tid: 3386] [value: 1159126505]
[pid: 3384] [tid: 3387] [value: 1632621729]
[pid: 3384] [tid: 3385] [value: 1433925857]
[pid: 3384] [tid: 3387] [value: 84353895]
[pid: 3384] [tid: 3386] [value: 2001100545]
[pid: 3384] [tid: 3385] [value: 154823367]
[pid: 3384] [tid: 3386] [value: 1585990364]
[pid: 3384] [tid: 3387] [value: 760313750]
[pid: 3384] [tid: 3386] [value: 356426808]
[pid: 3384] [tid: 3387] [value: 1889947178]
[pid: 3384] [tid: 3385] [value: 709393584]
[pid: 3384] [tid: 3387] [value: 1918502651]
[pid: 3384] [tid: 3386] [value: 1474612399]
[pid: 3384] [tid: 3386] [value: 1264095060]
[pid: 3384] [tid: 3387] [value: 1843993368]
[pid: 3384] [tid: 3385] [value: 1984210012]
[pid: 3384] [tid: 3386] [value: 1749698586]
[pid: 3384] [tid: 3387] [value: 1956297539]
[pid: 3384] [tid: 3385] [value: 463480570]
[pid: 3384] [tid: 3386] [value: 1975960378]
[pid: 3384] [tid: 3385] [value: 1892066601]
[pid: 3384] [tid: 3387] [value: 927612902]
[pid: 3384] [tid: 3385] [value: 603570492]
[pid: 3384] [tid: 3386] [value: 660260756]
[pid: 3384] [tid: 3387] [value: 485560280]
[pid: 3384] [tid: 3385] [value: 593209441]
[pid: 3384] [tid: 3386] [value: 894429689]
[pid: 3384] [tid: 3385] [value: 1947346619]
[pid: 3384] [tid: 3387] [value: 270744729]
[pid: 3384] [tid: 3386] [value: 1633108117]
[pid: 3384] [tid: 3387] [value: 2007905771]
[pid: 3384] [tid: 3386] [value: 822890675]
[pid: 3384] [tid: 3385] [value: 791698927]
[pid: 3384] [tid: 3387] [value: 498777856]
[pid: 3384] [tid: 3386] [value: 524872353]
[pid: 3384] [tid: 3385] [value: 1572276965]
[pid: 3384] [tid: 3387] [value: 1703964683]
[pid: 3384] [tid: 3386] [value: 1600028624]

[pid: 3380] [tid: 3383] [value: 749241873]
[pid: 3380] [tid: 3382] [value: 42999170]
[pid: 3380] [tid: 3381] [value: 135497281]
^C
osc@ubuntu:~/test$ ./cons 3
123[pid: 3388] [tid: 3391] [value: 1714636915]
[pid: 3388] [tid: 3389] [value: 424238335]
[pid: 3388] [tid: 3390] [value: 1649760492]
[pid: 3388] [tid: 3389] [value: 1189641421]
[pid: 3388] [tid: 3391] [value: 1350490027]
[pid: 3388] [tid: 3391] [value: 1102520059]
[pid: 3388] [tid: 3389] [value: 1967513926]
[pid: 3388] [tid: 3390] [value: 1540383426]
[pid: 3388] [tid: 3391] [value: 1303455736]
[pid: 3388] [tid: 3389] [value: 521595368]
[pid: 3388] [tid: 3390] [value: 1726956429]
[pid: 3388] [tid: 3391] [value: 861021530]
[pid: 3388] [tid: 3390] [value: 233665123]
[pid: 3388] [tid: 3389] [value: 468703135]
[pid: 3388] [tid: 3391] [value: 1801979002]
[pid: 3388] [tid: 3390] [value: 635723058]
[pid: 3388] [tid: 3389] [value: 1125898167]
[pid: 3388] [tid: 3391] [value: 2089018456]
[pid: 3388] [tid: 3389] [value: 1656478042]
[pid: 3388] [tid: 3390] [value: 1653377373]
[pid: 3388] [tid: 3389] [value: 1914544919]
[pid: 3388] [tid: 3389] [value: 756898537]
[pid: 3388] [tid: 3390] [value: 1973594324]
[pid: 3388] [tid: 3391] [value: 2038664370]
[pid: 3388] [tid: 3390] [value: 184803526]
[pid: 3388] [tid: 3389] [value: 1424268980]
[pid: 3388] [tid: 3391] [value: 749241873]
[pid: 3388] [tid: 3389] [value: 42999170]
[pid: 3388] [tid: 3390] [value: 135497281]
[pid: 3388] [tid: 3389] [value: 2084420925]
[pid: 3388] [tid: 3391] [value: 1827336327]
[pid: 3388] [tid: 3390] [value: 1159126505]
[pid: 3388] [tid: 3389] [value: 1632621729]
[pid: 3388] [tid: 3391] [value: 1433925857]
[pid: 3388] [tid: 3390] [value: 84353895]
[pid: 3388] [tid: 3389] [value: 2001100545]
[pid: 3388] [tid: 3390] [value: 154823367]
[pid: 3388] [tid: 3391] [value: 1585990364]
[pid: 3388] [tid: 3390] [value: 760313750]
[pid: 3388] [tid: 3389] [value: 356426808]
[pid: 3388] [tid: 3391] [value: 1889947178]
[pid: 3388] [tid: 3390] [value: 709393584]
[pid: 3388] [tid: 3391] [value: 1918502651]
[pid: 3388] [tid: 3389] [value: 1474612399]
[pid: 3388] [tid: 3390] [value: 1264095060]

```

运行 `./prod 1` 和 `./cons 1`


```

pid: 3384] [tid: 3385] [value: 1572276965]
pid: 3384] [tid: 3387] [value: 1703964683]
pid: 3384] [tid: 3386] [value: 1600028624]
c
osc@ubuntu:~/test$ ./prod 4
pid: 3393] [tid: 3396] [value: 1714636915]
pid: 3393] [tid: 3394] [value: 424238335]
pid: 3393] [tid: 3395] [value: 1649760492]
pid: 3393] [tid: 3396] [value: 1189641421]
pid: 3393] [tid: 3394] [value: 1350490027]
pid: 3393] [tid: 3395] [value: 1102520059]
pid: 3393] [tid: 3396] [value: 1967513926]
pid: 3393] [tid: 3395] [value: 1540383426]
pid: 3393] [tid: 3394] [value: 1303455736]
pid: 3393] [tid: 3396] [value: 521595368]
pid: 3393] [tid: 3395] [value: 1726956429]
pid: 3393] [tid: 3396] [value: 861021530]
pid: 3393] [tid: 3394] [value: 233665123]
pid: 3393] [tid: 3395] [value: 468703135]
pid: 3393] [tid: 3394] [value: 1801979802]
pid: 3393] [tid: 3396] [value: 635723058]
pid: 3393] [tid: 3395] [value: 1125898167]
pid: 3393] [tid: 3394] [value: 2089018456]
pid: 3393] [tid: 3396] [value: 1656478042]
pid: 3393] [tid: 3395] [value: 1653377373]
pid: 3393] [tid: 3396] [value: 1914544919]
pid: 3393] [tid: 3394] [value: 756898537]
pid: 3393] [tid: 3395] [value: 1973594324]
pid: 3393] [tid: 3394] [value: 2038664370]
pid: 3393] [tid: 3396] [value: 184803526]
pid: 3393] [tid: 3394] [value: 1424268980]
pid: 3393] [tid: 3394] [value: 749241873]
pid: 3393] [tid: 3396] [value: 42999170]
pid: 3393] [tid: 3395] [value: 135497281]
pid: 3393] [tid: 3396] [value: 2084420925]
pid: 3393] [tid: 3395] [value: 1827336327]
pid: 3393] [tid: 3396] [value: 1159126505]
pid: 3393] [tid: 3394] [value: 1632621729]
pid: 3393] [tid: 3395] [value: 1433925857]
pid: 3393] [tid: 3394] [value: 84353895]
pid: 3393] [tid: 3396] [value: 2001100545]
pid: 3393] [tid: 3396] [value: 1548233367]
pid: 3393] [tid: 3395] [value: 1585990364]
pid: 3393] [tid: 3394] [value: 760313750]
pid: 3393] [tid: 3395] [value: 356426808]
pid: 3393] [tid: 3394] [value: 1889947178]
pid: 3393] [tid: 3396] [value: 709393584]
pid: 3393] [tid: 3394] [value: 1918502651]
pid: 3393] [tid: 3395] [value: 1474612399]
pid: 3393] [tid: 3396] [value: 1264095060]
osc@ubuntu:~/test$ ./cons 5
123[pid: 3397] [tid: 3400] [value: 1714636915]
[pid: 3397] [tid: 3398] [value: 424238335]
[pid: 3397] [tid: 3399] [value: 1649760492]
[pid: 3397] [tid: 3398] [value: 1189641421]
[pid: 3397] [tid: 3400] [value: 1350490027]
[pid: 3397] [tid: 3400] [value: 1102520059]
[pid: 3397] [tid: 3398] [value: 1967513926]
[pid: 3397] [tid: 3398] [value: 1540383426]
[pid: 3397] [tid: 3399] [value: 1303455736]
[pid: 3397] [tid: 3400] [value: 521595368]
[pid: 3397] [tid: 3398] [value: 1726956429]
[pid: 3397] [tid: 3399] [value: 861021530]
[pid: 3397] [tid: 3398] [value: 233665123]
[pid: 3397] [tid: 3400] [value: 468703135]
[pid: 3397] [tid: 3399] [value: 1801979802]
[pid: 3397] [tid: 3398] [value: 635723058]
[pid: 3397] [tid: 3400] [value: 1125898167]
[pid: 3397] [tid: 3399] [value: 2089018456]
[pid: 3397] [tid: 3400] [value: 1656478042]
[pid: 3397] [tid: 3398] [value: 1653377373]
[pid: 3397] [tid: 3400] [value: 1914544919]
[pid: 3397] [tid: 3399] [value: 756898537]
[pid: 3397] [tid: 3398] [value: 1973594324]
[pid: 3397] [tid: 3400] [value: 2038664370]
[pid: 3397] [tid: 3398] [value: 184803526]
[pid: 3397] [tid: 3399] [value: 1424268980]
[pid: 3397] [tid: 3400] [value: 749241873]
[pid: 3397] [tid: 3399] [value: 42999170]
[pid: 3397] [tid: 3398] [value: 135497281]
[pid: 3397] [tid: 3399] [value: 2084420925]
[pid: 3397] [tid: 3400] [value: 1827336327]
[pid: 3397] [tid: 3398] [value: 1159126505]
[pid: 3397] [tid: 3399] [value: 1632621729]
[pid: 3397] [tid: 3400] [value: 1433925857]
[pid: 3397] [tid: 3398] [value: 84353895]
[pid: 3397] [tid: 3399] [value: 2001100545]
[pid: 3397] [tid: 3398] [value: 1548233367]
[pid: 3397] [tid: 3400] [value: 1585990364]
[pid: 3397] [tid: 3398] [value: 760313750]
[pid: 3397] [tid: 3399] [value: 356426808]
[pid: 3397] [tid: 3400] [value: 1889947178]
[pid: 3397] [tid: 3398] [value: 709393584]
[pid: 3397] [tid: 3400] [value: 1918502651]
[pid: 3397] [tid: 3399] [value: 1474612399]
[pid: 3397] [tid: 3398] [value: 1264095060]
[pid: 3397] [tid: 3399] [value: 1843993368]
[pid: 3397] [tid: 3400] [value: 1984210012]
[pid: 3397] [tid: 3398] [value: 1749698586]
[pid: 3397] [tid: 3400] [value: 1956297539]

```

速度更快的一方，会在全空或全满时等待仓库的变化

3. Linux内核实验（Linux 4.0或以上）

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

Linux进程的基本结构

Linux内核通过进程描述符 `task_struct` 结构体来管理进程，这个结构体包含了进程所需要的所有信息。结构体定义在 `include/linux/sched.h` 中，由于 `task_struct` 算是 linux 内核代码中最复杂的结构体之一，成员非常多，代码量500行不止，所以就选取部分相关度高的代码进行解析

进程状态

state

```

/* -1 unrunnable, 0 runnable, >0 stopped: */
volatile long      state;
int                exit_state;
int                exit_code;
int                exit_signal;

```

state成员的可能取值有

```

#define TASK_RUNNING            0
#define TASK_INTERRUPTIBLE     1
#define TASK_UNINTERRUPTIBLE   2
#define __TASK_STOPPED         4
#define __TASK_TRACED          8

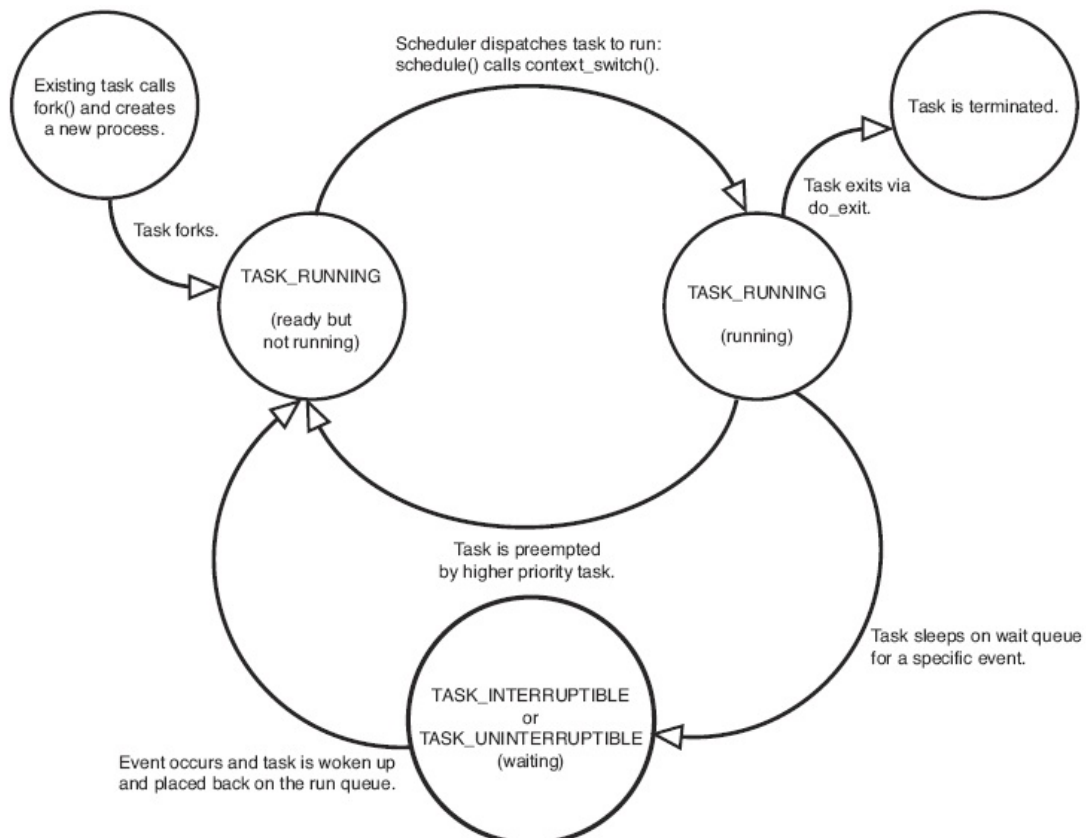
/* in tsk->exit_state */
#define EXIT_DEAD              16
#define EXIT_ZOMBIE            32
#define EXIT_TRACE              (EXIT_ZOMBIE | EXIT_DEAD)

/* in tsk->state again */
#define TASK_DEAD              64
#define TASK_WAKEKILL          128    /** wake on signals that are deadly */
#define TASK_WAKING            256
#define TASK_PARKED            512
#define TASK_NOLOAD            1024
#define TASK_STATE_MAX        2048

/* Convenience macros for the sake of set_task_state */
#define TASK_KILLABLE           (TASK_WAKEKILL | TASK_UNINTERRUPTIBLE)
#define TASK_STOPPED            (TASK_WAKEKILL | __TASK_STOPPED)
#define TASK_TRACED             (TASK_WAKEKILL | __TASK_TRACED)

```

- 其中 `TASK_RUNNING`、`TASK_INTERRUPTIBLE`、`TASK_UNINTERRUPTIBLE`、`__TASK_STOPPED`、`__TASK_TRACED` 是五个互斥的值，每个进程都必然处于其中一种
- 而 `exit_state`、`exit_code`、`exit_signal` 为 `state` 域另外的附加进程状态，一般当进程终止的时候，会出现这两种状态。包括 `EXIT_ZOMBIE`、`EXIT_DEAD`
- 进程状态的切换过程和原理



- `TASK_KILLABLE` 是除了 `TASK_INTERRUPTIBLE`、`TASK_UNINTERRUPTIBLE` 内核中实现的另外一种睡眠方式，原理和 `TASK_UNINTERRUPTIBLE` 类似但可以响应致命信号

进程表示符 PID

```
pid_t      pid;  
pid_t      tgid;
```

- linux 把不同的 pid 与系统中每个进程或轻量级线程关联，一个线程组所有线程与领头线程拥有相同的 pid
- 所以在使用系统调用 `sys_getpid()` 时，返回的是当前进程的 tgid 值

其他结构

- 进程内核栈

```
void      *stack;
```

- 进程标记

```
unsigned int      flags;
```

- Ptrace 系统调用

```
unsigned int      ptrace;
```

进程调度

优先级

```
int      prio;  
int      static_prio;  
int      normal_prio;  
unsigned int      rt_priority;
```

- prio 保存动态优先级
- static_prio 保存静态优先级，通过 nice 系统调用修改
- rt_priority 保存实时优先级
- normal_prio 由静态优先级和调度策略决定

调度策略

```
unsigned int      policy;  
  
const struct sched_class *sched_class;  
struct sched_entity se;  
struct sched_rt_entity rt;  
  
cpumask_t      cpus_allowed;
```

- policy 代表调度策略
- sched_class 调度类，se 为普通进程的调用实体，rt 为实时进程的调用实体，作业b中系统调用显示的大部分信息都是在实体 se 内的
- cpus_allowed 控制进程可以在什么处理器上运行

结构体其他内容

判断标志

```
int exit_code, exit_signal;
int pdeath_signal; /* The signal sent when the parent dies */
unsigned long jobctl; /* JOBCTL_*, siglock protected */

/* Used for emulating ABI behavior of previous Linux versions */
unsigned int personality;

/* scheduler bits, serialized by scheduler locks */
unsigned sched_reset_on_fork:1;
unsigned sched_contributes_to_load:1;
unsigned sched_migrated:1;
unsigned :0; /* force alignment to the next boundary */

/* unserialized, strictly 'current' */
unsigned in_execve:1; /* bit to tell LSMs we're in execve */
unsigned in_iowait:1;
```

时间

```
cputime_t utime, stime, utimescaled, stimescaled;
cputime_t gtime;
struct prev_cputime prev_cputime;
#ifdef CONFIG_VIRT_CPU_ACCOUNTING_GEN
seqcount_t vtime_seqcount;
unsigned long long vtime_snap;
enum {
    /* Task is sleeping or running in a CPU with VTIME inactive */
    VTIME_INACTIVE = 0,
    /* Task runs in userspace in a CPU with VTIME active */
    VTIME_USER,
    /* Task runs in kernelspace in a CPU with VTIME active */
    VTIME_SYS,
} vtime_snap_whence;
#endif
unsigned long nvcsw, nivcsw; /* context switch counts */
u64 start_time; /* monotonic time in nsec */
u64 real_start_time; /* boot based time in nsec */
/* mm fault and swap info: this can arguably be seen as either mm-specific or
thread-specific */
unsigned long min_flt, maj_flt;

struct task_cputime cputime_expires;
struct list_head cpu_timers[3];

/* process credentials */
const struct cred __rcu *real_cred; /* objective and real subjective task
                                     * credentials (COW) */
const struct cred __rcu *cred; /* effective (overridable) subjective task
                                * credentials (COW) */
char comm[TASK_COMM_LEN]; /* executable name excluding path
                           - access with [gs]et_task_comm (which lock
                           it with task_lock())
                           - initialized normally by setup_new_exec */

/* file system info */
struct nameidata *nameidata;
```

```

#ifdef CONFIG_SYSVIPC
/* ipc stuff */
struct sysv_sem sysvsem;
struct sysv_shm sysvshm;
#endif
#ifdef CONFIG_DETECT_HUNG_TASK
/* hung task detection */
unsigned long last_switch_count;
#endif

```

信号处理

```

struct signal_struct *signal;
struct sighand_struct *sighand;
1583
sigset_t blocked, real_blocked;
sigset_t saved_sigmask; /* restored if set_restore_sigmask() was used */
struct sigpending pending;
1587
unsigned long sas_ss_sp;
size_t sas_ss_size;

```

CPU调度架构

CPU调度准则

从CPU的吞吐量、周转时间、等待时间、响应时间等角度让CPU使用率尽可能高。在分时系统中，依靠进程优先级、进程分类等，依据进程调度策略优化调度

进程与调度相关数据结构

相关结构体

除了进程基本结构，还有 `sched_entity` 结构体

```

struct sched_entity {
    struct load_weight    load;
    unsigned long         runnable_weight;
    struct rb_node        run_node;
    struct list_head      group_node;
    unsigned int          on_rq;

    u64                   exec_start;
    u64                   sum_exec_runtime;
    u64                   vruntime;
    u64                   prev_sum_exec_runtime;

    u64                   nr_migrations;
    struct sched_statistics statistics;
}

```

- 其中 `load` 代表负载权重和重除
- `run_node` 表示红黑树上的节点
- `on_rq` 表示是否在调度队列上
- `exec_start`、`sum_exec_runtime`、`vruntime`、`prev_sum_exec_runtime` 则是有关调度的开始时间、执行总时间、虚拟运行时间、上次调度的执行总时间

```

struct load_weight {
    unsigned long    weight;
    u32              inv_weight;
};

```

- 为 sched_entity 结构体中使用到的 load_weight 结构体

相关调度类

```

struct sched_class {
    const struct sched_class *next;
    void (*enqueue_task) (struct rq *rq, struct task_struct *p, int flags);
    void (*dequeue_task) (struct rq *rq, struct task_struct *p, int flags);
    void (*yield_task)    (struct rq *rq);
    bool (*yield_to_task)(struct rq *rq, struct task_struct *p, bool preempt);
    void (*check_preempt_curr)(struct rq *rq, struct task_struct *p, int flags);
    /*
     * It is the responsibility of the pick_next_task() method that will
     * return the next task to call put_prev_task() on the @prev task or
     * something equivalent.
     *
     * May return RETRY_TASK when it finds a higher prio class has runnable
     * tasks.
     */
    struct task_struct * (*pick_next_task)(struct rq *rq,
                                           struct task_struct *prev,
                                           struct rq_flags *rf);
    void (*put_prev_task)(struct rq *rq, struct task_struct *p);
#ifdef CONFIG_SMP
    int (*select_task_rq)(struct task_struct *p, int task_cpu, int sd_flag, int
flags);
    void (*migrate_task_rq)(struct task_struct *p, int new_cpu);
    void (*task_woken)(struct rq *this_rq, struct task_struct *task);
    void (*set_cpus_allowed)(struct task_struct *p,
                             const struct cpumask *newmask);
    void (*rq_online)(struct rq *rq);
    void (*rq_offline)(struct rq *rq);
#endif
    void (*set_curr_task)(struct rq *rq);
    void (*task_tick)(struct rq *rq, struct task_struct *p, int queued);
    void (*task_fork)(struct task_struct *p);
    void (*task_dead)(struct task_struct *p);

    extern const struct sched_class stop_sched_class;
    extern const struct sched_class dl_sched_class;
    extern const struct sched_class rt_sched_class;
    extern const struct sched_class fair_sched_class;
    extern const struct sched_class idle_sched_class;
}

```

sched_class 类为 sched_entity 相关的功能实现，例如其中的CFS调度则使用的是 cfs_rq 结构体，在下一部分详细描述

运行队列

```

struct rq {

```

```

unsigned long nr_running;

#define CPU_LOAD_IDX_MAX 5
unsigned long cpu_load[CPU_LOAD_IDX_MAX];

struct load_weight load;

struct cfs_rq cfs;

unsigned long nr_uninterruptible;

struct task_struct *curr, *idle;
struct mm_struct *prev_mm;

u64 clock, prev_clock_raw;
u64 tick_timestamp;

#ifdef CONFIG_SMP
    struct sched_domain *sd;

    /* For active balancing */
    int active_balance;
    int push_cpu;

    struct task_struct *migration_thread;
    struct list_head migration_queue;
#endif
}

```

为整个调度框架的数据结构总入口，调度框架和实现函数只识别运行队列

优先级处理

```

#define MAX_USER_RT_PRIO    100
#define MAX_RT_PRIO        MAX_USER_RT_PRIO

#define MAX_PRIO            (MAX_RT_PRIO + 40)
#define DEFAULT_PRIO       (MAX_RT_PRIO + 20)

#define NICE_TO_PRIO(nice)  ((nice) + DEFAULT_PRIO)
#define PRIO_TO_NICE(prio)  ((prio) - DEFAULT_PRIO)
#define TASK_NICE(p)        PRIO_TO_NICE((p)->static_prio)

#define SCHED_LOAD_SHIFT    10
#define SCHED_LOAD_SCALE    (1L << SCHED_LOAD_SHIFT)
#define NICE_0_LOAD         SCHED_LOAD_SCALE
#define NICE_0_SHIFT        SCHED_LOAD_SHIFT

```

调度初始化

```

void __init sched_init(void)
{
    for_each_possible_cpu(i) {
        struct rt_prio_array *array;
        struct rq *rq;

        rq = cpu_rq(i);
    }
}

```

```

        init_cfs_rq(&rq->cfs, rq);

    }

    set_load_weight(&init_task);

    open_softirq(SCHED_SOFTIRQ, run_rebalance_domains, NULL);

    init_idle(current, smp_processor_id());
    current->sched_class = &fair_sched_class;
}

```

内核在系统启动时初始化调度框架的数据结构，调用的就是 `sched_init` 来初始化结构体 `rq` 并初始化第一个进程

周期调度器

```

void scheduler_tick(void)
{
    int cpu = smp_processor_id();
    struct rq *rq = cpu_rq(cpu);
    struct task_struct *curr = rq->curr;

    if (curr != rq->idle) /* FIXME: needed? */
        curr->sched_class->task_tick(rq, curr);

#ifdef CONFIG_SMP

    trigger_load_balance(rq, cpu);
#endif
}

```

周期调度器有 `scheduler_tick` 实现信息的统计和判断

其他

- CPU调度的其他内容包括主调度器
- 调度实体
- 调度标志
- 创建进程、唤醒进程的一些函数

CFS调度算法

原理

内核在实现 CFS 时，使用权重来描述进程应该分得多少CPU时间，因为 CFS 只负责非实时进程，所以权重与 nice 值密切相关。一般这个概念是，进程每降低一个 nice 值（优先级提升），则多获得 10% 的CPU时间，每升高一个 nice 值（优先级降低），则放弃 10% 的CPU时间

对于 CFS 调度类，内核不仅使用权重来衡量进程在一定CPU时间周期分得的时间配额，还是使用权重来衡量进程运行的虚拟时间的快慢 —— 权重越大，流失得越慢

计算相当公式：

- 当 `nice != NICE_0_LOAD` 虚拟时间 `vruntime = real_exec_time * NICE_0_LOAD / weight`
- 当 `nice == NICE_0_LOAD` 虚拟时间 `vruntime` 即使真实运行时间 `real_exec_time`

周期调度

```

static void task_tick_fair(struct rq *rq, struct task_struct *curr)
{
    struct cfs_rq *cfs_rq;
    struct sched_entity *se = &curr->se;

    for_each_sched_entity(se) {
        cfs_rq = cfs_rq_of(se);

        update_curr(cfs_rq);

        if (cfs_rq->nr_running > 1)
            check_preempt_tick(cfs_rq, curr);
    }
}

static void update_curr(struct cfs_rq *cfs_rq)
{
    struct sched_entity *curr = cfs_rq->curr;
    u64 now = rq_of(cfs_rq)->clock;
    unsigned long delta_exec;
    unsigned long delta_exec_weighted;

    delta_exec = (unsigned long)(now - curr->exec_start);

    curr->sum_exec_runtime += delta_exec;
    delta_exec_weighted = delta_exec;

    if (unlikely(curr->load.weight != NICE_0_LOAD)) {
        delta_exec_weighted = calc_delta_fair(delta_exec_weighted,
                                              &curr->load);
    }
    curr->vruntime += delta_exec_weighted;

    curr->exec_start = now;
}

static void check_preempt_tick(struct cfs_rq *cfs_rq, struct sched_entity *curr)
{
    unsigned long ideal_runtime, delta_exec;

    ideal_runtime = sched_slice(cfs_rq, curr);
    delta_exec = curr->sum_exec_runtime - curr->prev_sum_exec_runtime;
    if (delta_exec > ideal_runtime)
        resched_task(rq_of(cfs_rq)->curr);
}

```

CFS 类的 `task_tick_fair()` 方法利用 CFS 的分时是将降低当前运行进程对其他待调度进程造成的延迟为目标的调度算法，在一定的调度周期内，调度队列内所有的进程都被调度运行一次，即公平对待每个进程

再次排队的实现

```

static void put_prev_task_fair(struct rq *rq, struct task_struct *prev)

```

```

{
    struct sched_entity *se = &prev->se;
    struct cfs_rq *cfs_rq;

    for_each_sched_entity(se) {
        cfs_rq = cfs_rq_of(se);

        if (prev->se.on_rq) {

            update_curr(cfs_rq);

            __enqueue_entity(cfs_rq, &prev->se);
        }

        cfs_rq->curr = NULL;
    }
}

```

选择最优进程使用CPU资源

```

static inline struct task_struct *
pick_next_task(struct rq *rq, struct task_struct *prev)
{
    const struct sched_class *class;
    struct task_struct *p;

    if (likely(rq->nr_running == rq->cfs.nr_running)) {
        p = fair_sched_class.pick_next_task(rq);
        if (likely(p))
            return p;
    }

    class = sched_class_highest;
    for ( ; ; ) {
        p = class->pick_next_task(rq);
        if (p)
            return p;
        class = class->next;
    }
}

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

    if (unlikely(!cfs_rq->nr_running))
        return NULL;

    do {
        if (first_fair(cfs_rq)) {
            se = __pick_next_entity(cfs_rq);
            if (se->on_rq) {

                __dequeue_entity(cfs_rq, se);

                se->exec_start = rq_of(cfs_rq)->clock;
            }
        }
    } while (0);
}

```

```

        cfs_rq->curr = se;

        se->prev_sum_exec_runtime = se->sum_exec_runtime;
    }
}

cfs_rq = group_cfs_rq(se);
} while (cfs_rq);

return task_of(se);
}

```

其他

- 类比CPU调度，CFS也有其对应的创建、唤醒函数 `task_new_fair()`、`enqueue_task_fair()`

作业问题解答

进程优先级、nice值和权重的关系

内核在实现CFS时，使用权重来描述进程应该分得多少CPU时间，因为CFS只负责非实时进程，所以权重与nice值密切相关。一般这个概念是，进程每降低一个nice值（优先级提升），则多获得10%的CPU时间，每升高一个nice值（优先级降低），则放弃10%的CPU时间

对于CFS调度类，内核不仅使用权重来衡量进程在一定CPU时间周期分得的时间配额，还是使用权重来衡量进程运行的虚拟时间的快慢———权重越大，流失得越慢

CFS调度器中的vruntime的基本思想是什么？是如何计算的？何时得到更新？其中的min_vruntime有什么作用？

```

static inline s64 entity_key(struct cfs_rq *cfs_rq, struct sched_entity *se)
{
    return se->se->vruntime - cfs_rq->min_vruntime;
}

```

基本思想：

我们需要对每一个task添加一个线索字段用于追踪task的执行时间以确保完全加权公平这个字段就是"virtual runtime"。以执行时间不快不慢的task的绝对时间为基本归一化

计算与更新：

OK, 我们有了一个保证公平的线索, 在上例中, 就是上述的CPU2的基准时间流逝的值, 我们已经可以得到, 对于一个调度周期时间 T 内一个权重为 w_n 的task, 其执行的绝对时间为 T_n , 那么其 **virtual runtime** 就是:

$$V_n = T_n \times \frac{w_{base}}{w_n} = T \times \frac{w_n}{\sum_{i=1}^n w_i} \times \frac{w_{base}}{w_n} = T \times \frac{w_{base}}{W_{total}}$$

它已经和具体的权重 w_n 无关了, 如果按照 w_{base} 比如上例的 w_2 归一化, 即将 w_{base} 作为 **单位1**, 那么在一个调度周期内, 每一个task流逝的虚拟时间 **virtual runtime** 就是:

$$V_1 = V_2 = \dots = V_n = T \times \frac{1}{W_{total}}$$

事实上, 我们可以将任意task的权重做基准来计算基准流逝时间, 而不一定非要用中间的那个, 只所以用中间的那个, 只是计算时方便。

自然而然, 两次时钟tick时间差 ΔT 之间, 当前task的 **virtual runtime** 流逝必须是 $\Delta T \times \frac{1}{W_{total}}$ 才能保证所有task的 **virtual runtime** 趋向于一致, 保证完全公平!

那么, 我们最初的问题也就迎刃而解了, 当当前task的执行时间超过调度周期内的配额 T_n 时, 如何挑选下一个要执行的task呢?

- 挑选 **virtual runtime** 累加和最小的即可。

现在让我们更新一下算法:

- 一个进程投入运行时:
 $P.Runtime = now$
- 时钟tick时:
 $\Delta T = now - P.Runtime$
 $V_n = V_n + \Delta T \times \frac{1}{W_{total}}$
如果 ΔT 大于 $T \times \frac{w_n}{\sum_{i=1}^n w_i}$, 就切换到下一个task。
- 切换时:
将当前task以 V_n 为键值插入队列;
在队列中选取 V_i 最小的task投入运行。

以上这些就是Linux CFS的核心了。

<https://blog.csdn.net/armlinuxww>

min_vruntime 的作用

虚拟运行时间, 因为进程的优先级不一样加入了权值的因素。min_vruntime 就是最小虚拟运行时间, 其目的是为了创造一个相对的概念, 为了移除 cfs_rq 对 set_task_cpu() 中定义的函数的依赖性, 我们需要一个不变的标准对各个不同的参数, 即用一个定量衡量每一个变量

b 添加系统调用: 用户层测试程序 mycall.c

实现步骤

1. 环境安装

系统已经自带有 gcc 编译器、vim 编辑器和编译内核的 make

只需要额外安装 bison、flex、libssl-dev、libncurses5-dev, 这一串软件用来配置内核

```
apt-get install bison flex libssl-dev libncurses5-dev
```

2. 修改系统调用表

在目前修改的 linux-source-4.10.0 目录下

```
vim arch/x86/entry/syscalls/syscall_64.tbl
```

在文件末尾分配自己的系统调用号

3. 申明系统调用服务例程原型

```
vim include/linux/syscalls.h
```

```
asmlinkage long sys_crysr(void);
```

4. 实现系统调用服务

```
vim kernel/sys.c
```

此处由于工程量过大在 vim 编辑太不方便，所以使用了 vs code 连接虚拟机，并对 sys.c 进行修改但是由于 vs code 并没有权限直接修改系统文件，所以需要在虚拟机内修改 sys.c 的权限

```
sudo chown -R osc:osc ./sys.c
```

然后在 sys.c 末尾添加服务函数

```
SYSCALL_DEFINE0(crysr){
    printk("sys_call(crysr) success:\n");
    printk("se.exec_start      :   %llu\n",current->se.exec_start);
    printk("se.vruntime           :   %llu\n",current->
>se.vruntime);
    printk("se.nr_migrations         :   %llu\n",current->
>se.nr_migrations);
    printk("nr_switchs              :   %lu\n",current->nvcsw+current->nivcsw);
    printk("nr_voluntary_switches   :   %lu\n",current->nvcsw);
    printk("nr_involuntary_switches :   %lu\n",current->nivcsw);
    printk("se.load.weight          :   %lu\n",current->se.load.weight);
    printk("se.avg.load_sum         :   %llu\n",current->se.avg.load_sum);
    printk("se.avg.util_sum         :   %u\n",current->se.avg.util_sum);
    printk("se.avg.load_avg         :   %lu\n",current->se.avg.load_avg);
    printk("se.avg.util_avg         :   %lu\n",current->se.avg.util_avg);
    printk("se.last_update_time     :   %llu\n",current->
>se.avg.last_update_time);
    return 0;
}
```

相关原理：

- current 指针指向的是 include/linux/sched.h 中的 task_struct 结构体，而结构体内部包含有需要打印的信息
- 其中只有 nr_voluntary_switches 和 nr_involuntary_switches 所对应的信息，以及他们的加和 nr_switchs 是在结构体下的
- 其余的信息都在 task_struct 结构体内部的 sched_entity 结构体 se 中

5. 重新编译内核

清楚残留的 .config 和 .o 文件，并配置内核

```
make mrproper
make menuconfig
```

保存配置信息后编译内核

踩坑：

- 重新编译内核需要足够的硬盘存储空间和足够的内存空间，而一开始预设的空间是远远不够的，会在编译中途出现报错
- 在 VirtualBox 中将虚拟机内存分配到 2048MB 以上
- 将虚拟机虚拟介质分配到 30GB 以上，并通过 gparted-live-1.0.0-5-amd64.iso 磁盘控制管理工具，分配这些存储空间

编译内核、编译模块、安装模块、安装内核

```
make -j2
make modules
make modules_install
make install
```

其中 j2 和 j4 可以加快编译但是也同样会消耗更多的资源

6. 重启

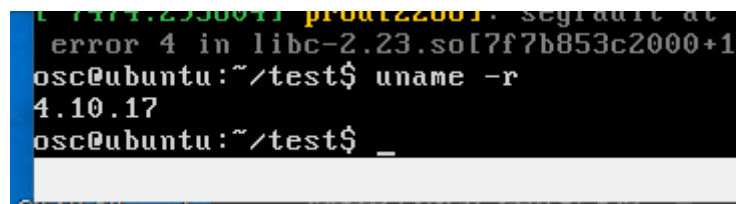
```
reboot
```

7. 编写用户层程序调用自定义的系统调用

新建 mycall.c 文件，调用 syscall(332)，编译运行

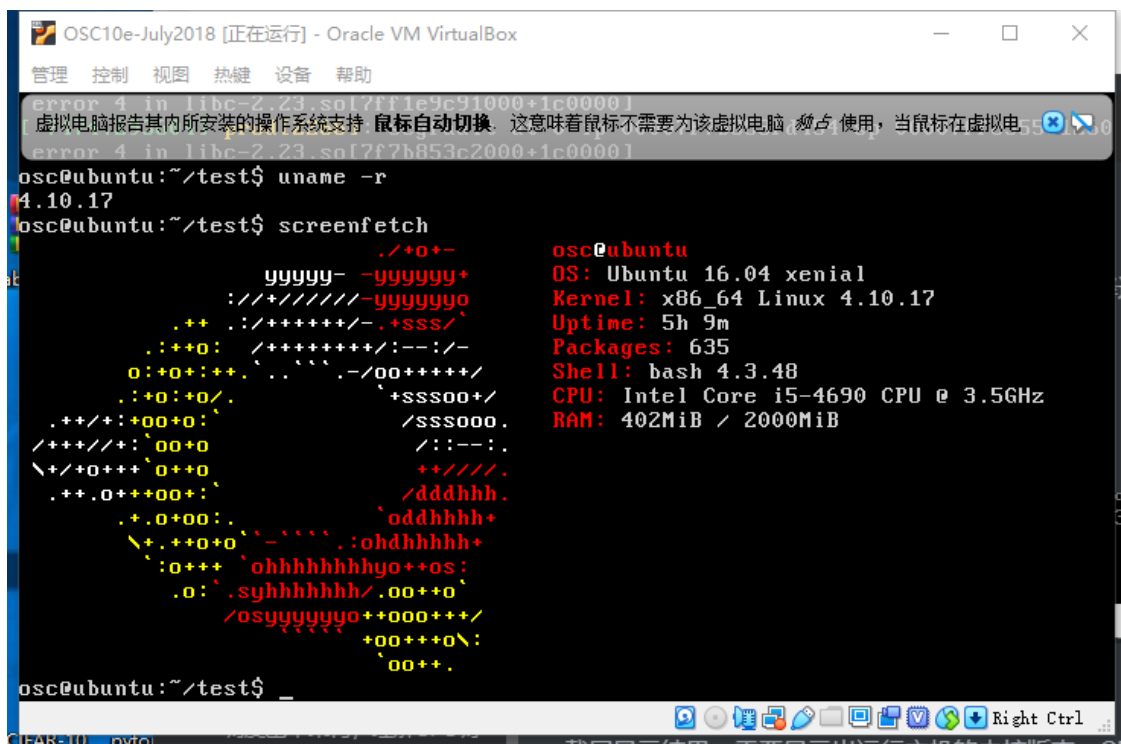
结果展示

1. 内核版本信息



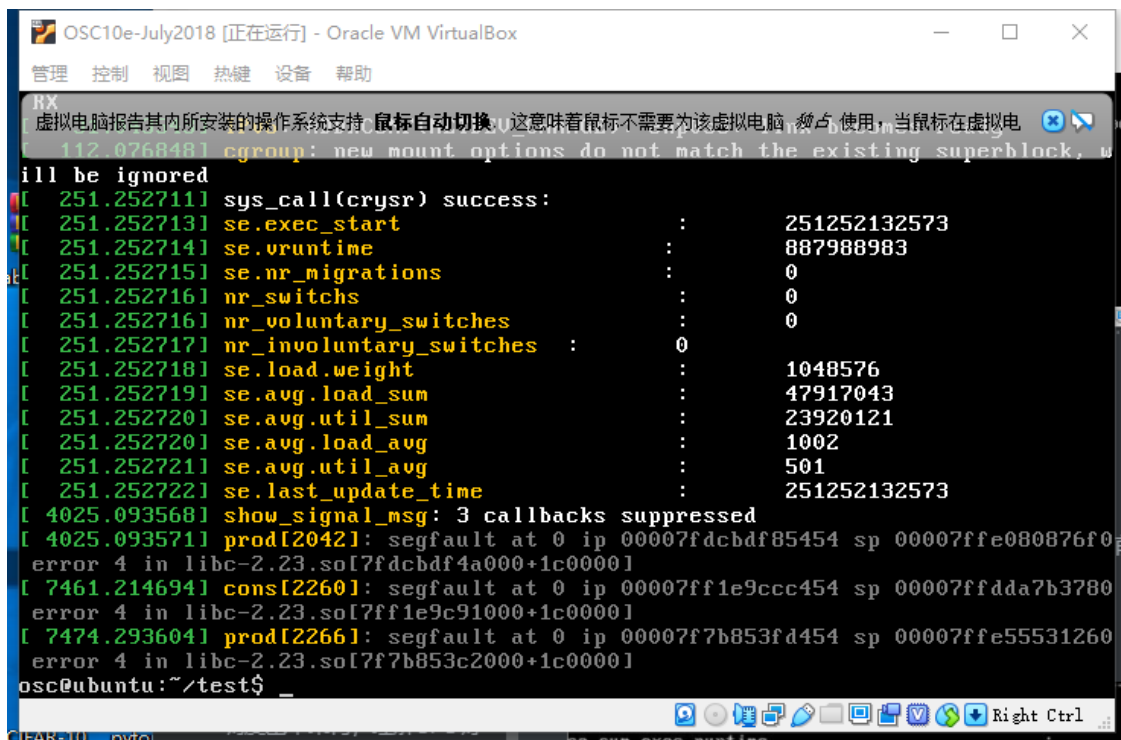
```
error 4 in libc-2.23.so[7f7b853c2000+1
osc@ubuntu:~/test$ uname -r
4.10.17
osc@ubuntu:~/test$ _
```

2. 其他信息 (screenfetch 命令获取)



```
error 4 in libc-2.23.so[7ff1e9c91000+1c00001
虚拟机报告其内所安装的操作系统支持 鼠标自动切换。这意味着鼠标不需要为该虚拟机，被点使用，当鼠标在虚拟机
error 4 in libc-2.23.so[7f7b853c2000+1c00001
osc@ubuntu:~/test$ uname -r
4.10.17
osc@ubuntu:~/test$ screenfetch
      .-/o+--
    yyyyy- -yyyyyy+
  ://+/////--yyyyyyo
.++ ://+++++/-+.sss/
.++o: /+++++/-:--:/-
o:+o+:+. ' ' ' -/oo++++/
.:+o:+o/. ' ' ' +sss00+/
.++/+:+oo+o: ' ' ' /sss000.
/+++//+:oo+o ' ' ' /:--:~
\+/+o+++o+o ' ' ' +++++//
.++.o+++oo+: ' ' ' /dddhhh.
.+.o+oo: ' ' ' 'oddhhhh+
\..++o+o ' ' ' 'ohdhhhh+
:o+++ 'ohhhhhhhhyo++os:
.o: 'syhhhhhhh/.oo++o
/osyyyyyyo++ooo+++/
      +oo++o\
      oo++.
```

3. 进程的调用信息（dmesg 命令获取）



```
OSC10e-July2018 [正在运行] - Oracle VM VirtualBox
管理 控制 视图 热键 设备 帮助
虚拟机报告其内所安装的操作系统支持 鼠标自动切换。这意味着鼠标不需要为该虚拟机，被点使用，当鼠标在虚拟机
112.0268481 cgroup: new mount options do not match the existing superblock, u
ill be ignored
[ 251.252711] sys_call(crysr) success:
[ 251.252713] se.exec_start : 251252132573
[ 251.252714] se.vruntime : 887988983
[ 251.252715] se.nr_migrations : 0
[ 251.252716] nr_switches : 0
[ 251.252716] nr_voluntary_switches : 0
[ 251.252717] nr_involuntary_switches : 0
[ 251.252718] se.load.weight : 1048576
[ 251.252719] se.avg.load_sum : 47917043
[ 251.252720] se.avg.util_sum : 23920121
[ 251.252720] se.avg.load_avg : 1002
[ 251.252721] se.avg.util_avg : 501
[ 251.252722] se.last_update_time : 251252132573
[ 4025.093568] show_signal_msg: 3 callbacks suppressed
[ 4025.093571] prod[2042]: segfault at 0 ip 00007fdcbdf85454 sp 00007ffe080876f0
error 4 in libc-2.23.so[7fdcbdf4a000+1c00001]
[ 7461.214694] cons[2260]: segfault at 0 ip 00007ff1e9ccc454 sp 00007ffdda7b3780
error 4 in libc-2.23.so[7ff1e9c91000+1c00001]
[ 7474.293604] prod[2266]: segfault at 0 ip 00007f7b853fd454 sp 00007ffe55531260
error 4 in libc-2.23.so[7f7b853c2000+1c00001]
osc@ubuntu:~/test$ _
```

参考文献

1. <https://blog.csdn.net/ruglcc/article/details/7814546/> Makefile经典教程(掌握这些足够)
2. https://blog.csdn.net/sinat_35297665/article/details/78467725 Linux经典问题一五哲学家就餐问题
3. <https://wenku.baidu.com/view/de5426edc1c708a1284a4469.html> 操作系统实验报告生产者消费者问题
4. <https://blog.csdn.net/yaozhiyi/article/details/7561759> OS: 生产者消费者问题(多进程+共享内存+信号量)
5. <https://blog.csdn.net/panxj856856/article/details/83014015> Linux内核源码解析 - CFS调度算法

6. https://blog.csdn.net/qg_41357569/article/details/83017667 操作系统实验一：linux内核编译及添加系统调用
7. <https://code.woboq.org/linux/linux/> Linux内核代码在线查找
8. https://blog.csdn.net/BigData_Mining/article/details/89014263 Ubuntu下vscode获得root权限
9. <https://blog.csdn.net/gatieme/article/details/51383272> Linux进程描述符task_struct结构体详解--Linux进程的管理与调度
10. <https://blog.csdn.net/CrazyHeroZK/article/details/84586537> Linux内核调度框架和CFS调度算法
11. <https://blog.csdn.net/u010173306/article/details/46646109> linux调度器第三代cfs(2)分解代码_vruntime和min_vruntime大概理解
12. <https://blog.csdn.net/armlinuxww/article/details/97242063> Linux CFS调度算法核心解析