①、高优先级任务和低优先级任务共享一个信号量 S，其计数初始化为 0.

根据 API 文档，创建信号量的函数如下

```
sem_t* sem_open ( const char *  name,
                  int           oflag,
                                ...
                )
```

创建或打开命名的计数信号量

该接口用于创建或打开命名的计数信号量。成功调用该接口之后，任务可以通过返回的信号 量ID来引用name关联的信号量，在 sem_wait()、sem_timedwait()、sem_trywait()、sem_post() 和sem_close()接口的调用中使用。

参数:

name 信号量对象的名字，用相同的name值调用sem_open()的任务会引用相同的信号量对象。

oflag 信号量对象的标志位，决定sem_open()是要创建还是仅仅访问这个信号量，oflag的取值包括：
O_CREAT 表示在信号量不存在时创建信号量。如果设置了O_CREAT标志且信号量已存在，那么 O_CREAT将不起作用，除非设置了O_EXCL标志；否则，sem_open( )会创建一个命名的信号量。
O_EXCL 在设置了 O_EXCL 和 O_CREAT的情况下，如果name指定名称的信号量存在的话，sem_open() 函数调用会失败。如果信号量不存在，检测信号量存在的操作和创建信号量的操作是原子性的， 以免其他任务以O_EXCL 和 O_CREAT的标志位来执行 sem_open()。
... 创建信号量还需要四个附加参数：
mode 类型为mode_t，表示信号量的权限位，本系统中忽略该值；
value 类型为unsigned，表示信号量创建时的初始值 ，合法的计数信号量初始值要小于等于SEM_VALUE_MAX ，合法的二进制信号量初始值只能为0或1；
sem_type 类型为int，表示信号量的类型，取值包括 SEM_BINARY（二进制信号量）和SEM_COUNTING（计数信号量）；
waitq_type 类型为WaitQ_Type，表示等待信号量资源的任务的排队策略，取值包括 PTHREAD_WAITQ_FIFO（先进先出策略）和PTHREAD_WAITQ_PRIO（优先级策略）。

返回:
信号量ID 函数执行成功，返回信号量的地址
SEM_FAILED 函数执行失败，并设定 errno 来指出错误

因此创建信号量如下

```
// 任务调度时间测试作用函数
void test_task_scheduling()
{
    pthread_t taskA, taskB;
    int ret;

    // 初始化信号量，初始值为0
    task_semid = sem_open("task_semid",O_CREAT, 0777, 0, SEM_BINARY, PTHREAD_WAITQ_PRIO);
    if (task_semid == SEM_FAILED)
    {
        printk("Failed to create semaphore\n");
        return NULL;
    }
```

②、接着创建出高优先级任务，但是这个任务的第一个指令为等待信号量 sem_wait，因此该任务将被挂起

同样的，在 API 文档中查找创建任务的方式

```
int pthread_create2 ( pthread_t *       pthread,
                      char *            name,
                      int               priority,
                      int               options,
                      int               stacksize,
                      void *(*)(void *) start_routine,
                      void *            argu
                    )
```

创建ReWorks任务

该接口以指定的任务名称、任务优先级和任务选项等信息创建一个新任务。任务名可以是最多32个ASCII字符组 成的字符串（含结束符），如果多于32个字节，只取前31个字节的内容，并添加结束符。任务栈由系统分配，参数stacksize指定其大小，栈 大小应该是个偶数。

参数:

pthread	存放任务标识的指针
name	任务名称，任务名可以是最多32个ASCII字符组成的字符串（含结束符），如果多于32个字节，只取前31个 字节的内容，并添加'\0'结束符。如果该参数为空，则新任务名为"pN"，其中N为新任务创建时递增的整数。
priority	任务优先级
options	任务选项，任务选项有RE_UNBREAKABLE、RE_FP_TASK、RE_NO_STACK_FILL和RE_NO_TIMESLICE四种。 RE_UNBREAKABLE表示任务忽略断点； RE_FP_TASK表示任务支持浮点运算； RE_NO_STACK_FILL表示任务栈不需要填写0XEE； RE_NO_TIMESLICE表示任务不支持时间片调度
stacksize	任务栈大小，最小为PTHREAD_STACK_MIN个字节，任务栈的内存从系统核心内存中分配。
start_routine	任务执行函数，不能为空
argu	任务执行函数的参数

返回:

0 函数执行成功
EAGAIN 系统缺少创建任务的资源，或超过了系统限制的任务总数
EINVAL priority超出范围，options的取值不合法，thread、start_routine是无效指针，或者stacksize小于0
EHOOKFAILED 钩子函数执行失败
ECALLEDINISR 当前处于中断上下文中，不能执行该操作

创建出高优先级任务，并等待信号量

```
#define TASK_TEST_COUNT 100000
#define HIGH_PRIORITY 120
#define LOW_PRIORITY 100
```

```
// 创建高优先级任务A
ret = pthread_create2(&taskA, "taskA", HIGH_PRIORITY, 0, 32 << 10,
                      taskA_func, NULL);
if (ret != 0)
{
    printk("Creating Task A failed\n");
    sem_close(task_semid);
    return NULL;
}
```

同样的，API 文档中也有等待信号量的函数

int sem_wait (sem_t * sem)

锁定信号量

该接口用来锁定sem指定的信号量。如果当前信号量的值为0，那么调用任务不会从sem_wait()中返回， 除非它成功锁定信号量或是被信号中断。如果该接口调用成功，信号量状态变为锁定状态，且保持到sem_post() 接口被成功调用。

参数:

sem 信号量标识

返回:

0 调用任务在 sem指定的信号量上成功执行了信号量锁定操作
-1 函数执行失败，设置 errno 指出错误，信号量的状态不发生改变。

错误号:

EINTR 函数被一个信号中断

EINVAL sem指定的信号量不合法

EOBJDELETED 因信号量对象被删除而终止阻塞等待

EMNOTINITED 信号量模块尚未初始化

ECALLEDINISR 接口不能在中断上下文中调用

ECANCELED 通过sem_flush解除的锁定

参见:

sem_trywait()
sem_timedwait()
sem_post()

示例:

example_sem_destroy.c、example_sem_flush.c及example_sem_wait.c.

```c
// 任务A：高优先级任务，等待信号量并记录时间
void* taskA_func(void* arg)
{
    while (task_test_count < TASK_TEST_COUNT)
    {
        sem_wait(task_semid); // 等待信号量
        freq = get_cntfrq();

        // 记录获取信号量的时间
        task_time2 = sys_timestamp();
//        printk("task_test_count is %d task_time2 is %10d\n",task_test_count,(int)task_time2);

        task_time_diff[task_test_count] = task_time2 - task_time1;
        printk("test count: %d, task_scheduling_time: %d\n", task_test_count, (int)(task_time_diff[task_test_count] * 1000000000.0 / freq));
//        printk("task_time_diff[%d] is %10d\n",task_test_count,(int)(task_time_diff[task_test_count]* 1000000.0 / freq));


        // 更新统计数据
        task_sum_time += task_time_diff[task_test_count];
        if (task_time_diff[task_test_count] > task_max_time)
        {
            task_max_time = task_time_diff[task_test_count];
        }
        if (task_time_diff[task_test_count] < task_min_time)
        {
            task_min_time = task_time_diff[task_test_count];
        }

        task_test_count++;
    }
    return NULL;
}
```

③、接着创建出低优先级任务，这个任务在执行的过程中释放信号量。高优先级任务得到信号量，结束挂起，并且将立马抢占低优先级任务

```
// 创建低优先级任务B
ret = pthread_create2(&taskB, "taskB", LOW_PRIORITY, 0, 32 << 10,
                      taskB_func, NULL);
if (ret != 0)
{
    printk("Creating Task B failed\n");
    pthread_cancel(taskA);
    sem_close(task_semid);
    return NULL;
}
```

int sem_post (sem_t * sem)

解除锁定信号量

该接口用来解锁sem指定的信号量。 如果从该接口的操作结果中得到的信号量的值是正数， 就表示没有任务在阻塞等待该信号量解锁，信号量的值只需要简单地递增；如果从该接口的 操作结果中得到的信号量的值是0，那么其中一个阻塞等待该信号量的任务将成功从sem_wait() 接口调用中返回。阻塞在信号量上的任务按优先级策略解除阻塞，即解除任务的阻塞时会选择 最高优先级且等待时间最长的任务。

参数:

sem 信号量标识

返回:

0 函数执行成功
-1 函数执行失败，设置 errno 指出错误

错误号:

EINVAL sem指定的信号量不合法

EMNOTINITED 信号量模块尚未初始化

注意:

阻塞在信号量上的任务按照优先级策略解除阻塞。

参见:

sem_wait()
sem_trywait()
sem_timedwait()

示例:

example_pthread_get_status_str.c、example_sem_destroy.c及example_sem_post.c.

```
// 任务B：低优先级任务，释放信号量并记录时间
void* taskB_func(void* arg)
{
    while (task_test_count < TASK_TEST_COUNT)
    {
        // 记录释放信号量的时间
        task_time1 = sys_timestamp();
        sem_post(task_semid); // 释放信号量

    }
    return NULL;
}
```

④、测量低优先级任务释放信号量到高优先级获取到信号量间的时间即可测量出任务间上下文切换时间

```c
// 任务A：高优先级任务，等待信号量并记录时间
void* taskA_func(void* arg)
{
    while (task_test_count < TASK_TEST_COUNT)
    {
        sem_wait(task_semid); // 等待信号量
        freq = get_cntfrq();

        // 记录获取信号量的时间
        task_time2 = sys_timestamp();
//        printk("task_test_count is %d task_time2 is %10d\n",task_test_count,(int)task_time2);

        task_time_diff[task_test_count] = task_time2 - task_time1;
        printk("test count: %d, task_scheduling_time: %d\n", task_test_count, (int)(task_time_diff[task_test_count] * 1000000000.0 / freq));
//        printk("task_time_diff[%d] is %10d\n",task_test_count,(int)(task_time_diff[task_test_count]* 1000000.0 / freq));


        // 更新统计数据
        task_sum_time += task_time_diff[task_test_count];
        if (task_time_diff[task_test_count] > task_max_time)
        {
            task_max_time = task_time_diff[task_test_count];
        }
        if (task_time_diff[task_test_count] < task_min_time)
        {
            task_min_time = task_time_diff[task_test_count];
        }

        task_test_count++;
    }
    return NULL;
}

// 任务B：低优先级任务，释放信号量并记录时间
void* taskB_func(void* arg)
{
    while (task_test_count < TASK_TEST_COUNT)
    {
        // 记录释放信号量的时间
        task_time1 = sys_timestamp();
        sem_post(task_semid); // 释放信号量

    }
    return NULL;
}
```

⑤、测量通过多次取平均的方法得到结果