任务管理实验设计

第五组

1 实验目的

1. 深入了解任务控制块TCB的属性以及任务管理相关函数，并学会如何使用`Liteos`相关函数调用、处理任务属性值
2. 在`Liteos`任务管理函数的基础上，通过对三个任务执行挂起、阻塞、删除、恢复等操作，研究任务管理有意义的状态间变化。

2 实验环境

2.1 鸿蒙软件环境

- OpenHarmony-v1.0-release

2.2 开发环境

- Windows11
- MobaXterm_Portable_v12.2
- Vscode

2.3 硬件环境

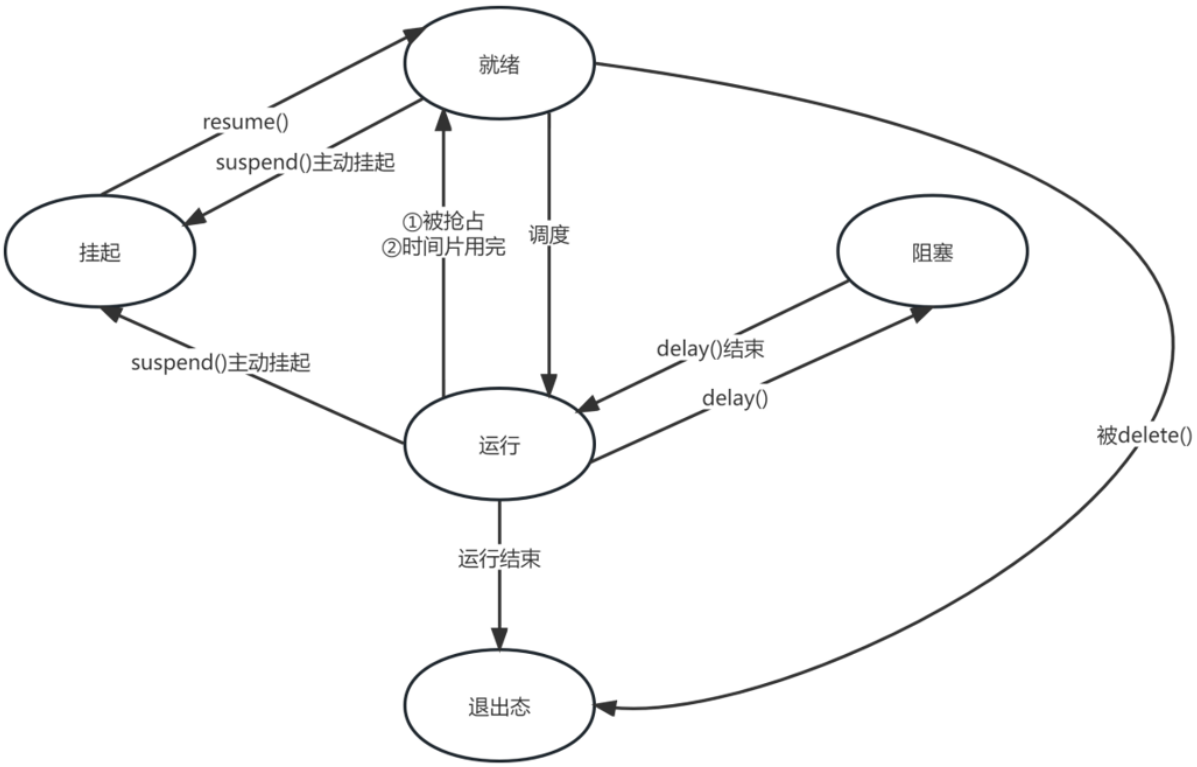
- Ubuntu 18.04

2.4 硬件环境

- IMX6ULL MINI 开发板

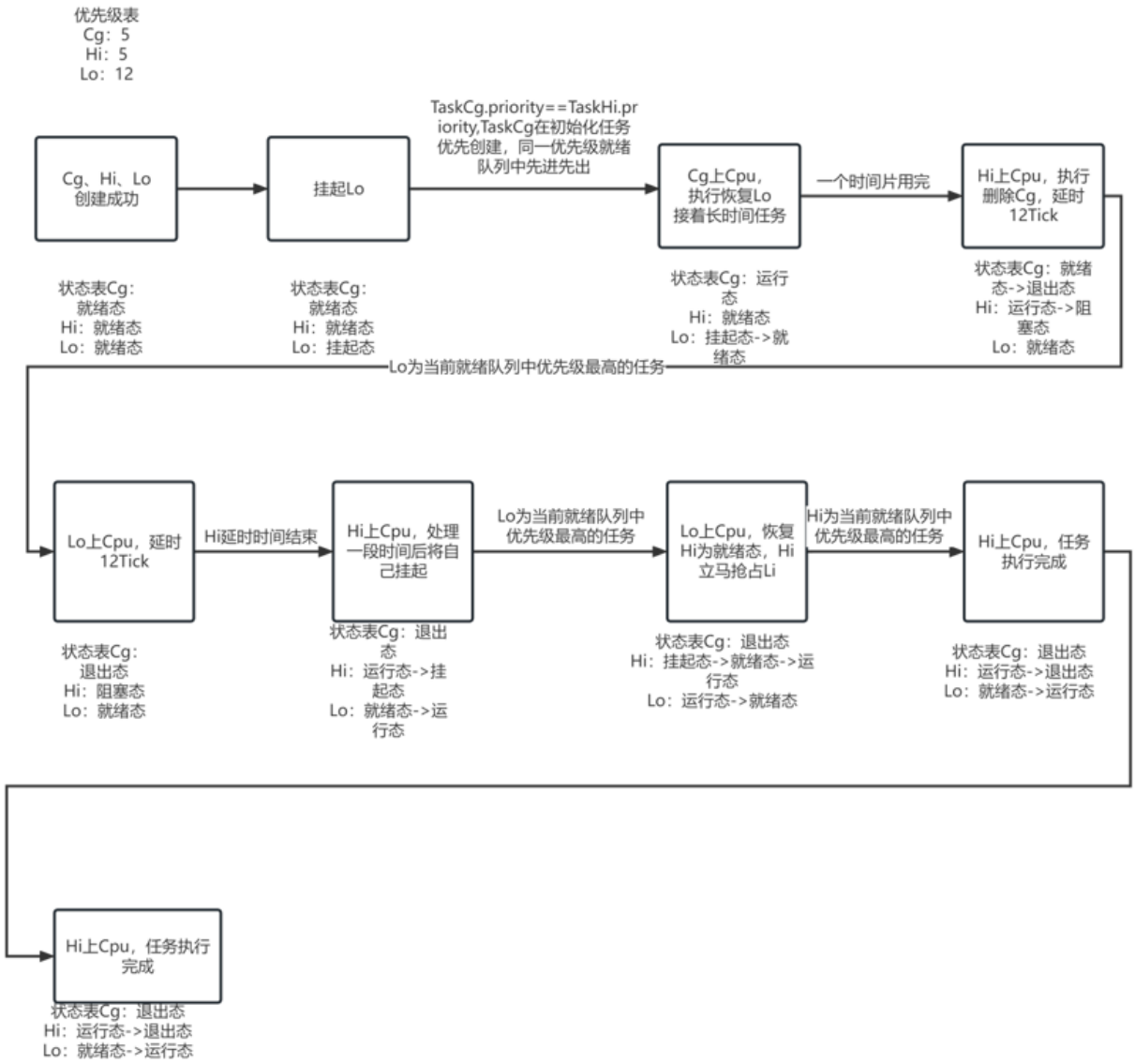
3 实验要求

编写程序创建三个任务，通过调用Liteos相关函数实现如图1所示的所有状态转换，对于关键的状态转换，调用TCB相关参数在命令行窗口输出任务的状态、优先级等信息。本实验的要求在于对任务状态转变的流程以及条件有清晰的认识，通过三个任务之间的相互关系来实现任务状态的切换。



4　实验状态图与状态对照图

实验状态图：为了方便对实验流程的理解，以及了解实验中三个任务是如何交互的，展示一张状态图来展示任务状态变化的过程，其中方块中的为事件，横线上为转化的原因。借助状态图可以清楚的了解实验是如何进行的。



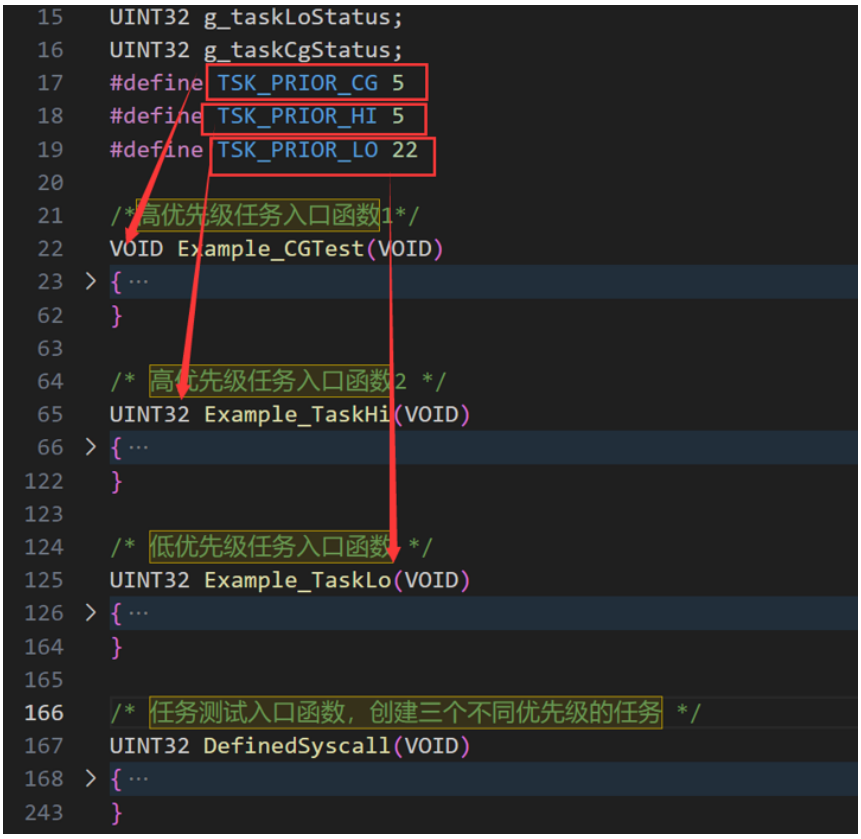
状态对照图：在实验中我们调用任务体输出任务的状态码和优先级码，状态对照表可以让我们清楚的映射相应的码对应的状态，方便查看实验结果。

```
#define OS_TASK_STATUS_INIT          0x0001U
#define OS_TASK_STATUS_READY         0x0002U
#define OS_TASK_STATUS_RUNNING       0x0004U
#define OS_TASK_STATUS_SUSPEND       0x0008U
#define OS_TASK_STATUS_PEND          0x0010U
#define OS_TASK_STATUS_DELAY         0x0020U
#define OS_TASK_STATUS_TIMEOUT       0x0040U
#define OS_TASK_STATUS_PEND_TIME     0x0080U
#define OS_TASK_STATUS_EXIT          0x0100U
#define OS_TASK_STATUS_UNUSED        0x0200U
#define OS_TASK_FLAG_PTHREAD_JOIN    0x0400U
#define OS_TASK_FLAG_DETACHED        0x0800U
#define OS_TASK_FLAG_IDLEFLAG        0x1000U
#define OS_TASK_FLAG_SYSTEM_TASK     0x2000U
```

5 实验步骤

5.1 创建自定义任务

创建三个Task，以及一个任务测试入口函数。人为设定此3个Task的优先级。Example_CGTest()和Example_TaskHi()的优先级都是5，处于高优先级，将两个任务的优先级设置为同一优先级便于研究**同优先级就绪队列的调度情况**，Example_TaskLo()的优先级为22，处于低优先级。通过DefinedSyscall()函数依此创建这创建三个任务。



5.2 任务创建约束

在调用任务创建函数时进行**锁任务调度**，防止有更高优先级的任务发生抢占，以保证任务可以被成功创建。

```
UINT32 DefinedSyscall(VOID)
{
    UINT32 ret;
    TSK_INIT_PARAM_S initParam;

    /* 锁任务调度，防止新创建的任务比本任务高而发生调度 */
    LOS_TaskLock();
    PRINTK("\nLOS_TaskLock() 成功，防止新创建的任务比本任务高而发生调度!\n");
```

InitParam设置创建任务所必须的参数，我们把它的优先usTaskPrio设置为TSK_Prior_CG(为5)，然后调用LOS_TaskCreate()创建任务。注意uwResved我们设置为LOS_TASK_STATUS_DETACHED。

```
initParam.pfnTaskEntry = (TSK_ENTRY_FUNC)Example_CGTest;
initParam.usTaskPrio = TSK_PRIOR_CG;
initParam.pcName = "TaskCg";
initParam.uwStackSize = 2048;
initParam.uwResved   = LOS_TASK_STATUS_DETACHED;
ret = LOS_TaskCreate(&g_taskCgId, &initParam);
```

将uwResved设置为LOS_TASK_STATUS_DETACHED即自删除状态，设置成自删除状态的任务会在运行完成时进行自删除动作。而此处设置自删除状态，将会影响任务状态的参数值，例如**就绪态的参数值为0X002H**，加入自删除状态后，**就绪态的参数值为0X802H**，此处在后面结果输出时也有所体现。

```
#if (LOSCFG_KERNEL_LITEIPC == YES)
    LOS_ListInit(&(taskCB->msgListHead));
    (VOID)memset_s(taskCB->accessMap, sizeof(taskCB->accessMap), 0, sizeof(taskCB->accessMap));
#endif
    taskCB->policy = (initParam->policy == LOS_SCHED_FIFO) ? LOS_SCHED_FIFO : LOS_SCHED_RR;
    taskCB->taskStatus = OS_TASK_STATUS_INIT;
    if (initParam->uwResved & OS_TASK_FLAG_DETACHED) {
        taskCB->taskStatus |= OS_TASK_FLAG_DETACHED;
    } else {
        LOS_ListInit(&taskCB->joinList);
        taskCB->taskStatus |= OS_TASK_FLAG_PTHREAD_JOIN;
    }
```

此处自删除状态的参数值为0X800，与任务的任何状态做或操作都将修改状态码的原始值。

```
#define LOS_TASK_STATUS_DETACHED                0x0800U

/**
 * @ingroup los_task
 * Task error code: Insufficient memory for task creation.
 *
 * Value: 0x03000200
 *
 * Solution: Allocate bigger memory partition to task creation.
 */
```

5.3 实现状态转换：就绪态➡挂起态（调用挂起接口方法）

任务全部创建完成后，三个任务都将成为就绪态，状态码为0X802，此处我们调用LOS_TaskSuspend(g_taskLoId)将优先级为22的任务挂起，即为实现任务由就绪态转换为挂起态，挂起态的状态码为0X008，即此处输出的状态码为0X808，实验结果如下：

```
OHOS #
LOS_TaskLock() 成功，防止新创建的任务比本任务高而发生调度!
TaskCg TaskHi TaskLo 创建成功!
TaskCg 状态: 802  TaskHi 状态: 802  TaskLo 状态: 802
TaskLo:就绪态->挂起态!
TaskCg 状态: 802  TaskHi 状态: 802  TaskLo 状态: 808
```

5.4 实现状态转换：就绪态➡运行态（系统调度方法）

任务创建完成后，解锁任务调度，操作系统会将优先级最高的任务调度到CPU执行。**注意：**此处有TaskHi和TaskCg优先级均为5，此时将用到**相同优先级就绪队列**的知识，采用尾插法以及先进先出的原则，即首先调度最先创建的TaskCg任务执行。

5.5 实现状态转换：挂起态➡就绪态（调用恢复接口方法）

TaskCg任务体中，首先将挂起的TaskLo任务恢复，为之后实验能够对任务二继续进行相关的操作，即实现主动的任务状态由挂起态到就绪态，实现结果如下：

```
---------------Enter TaskCg Handler.---------------
TaskCg:就绪态->运行态! TaskCg 状态: 804  TaskHi 状态: 802  TaskLo 状态: 808
TaskLo:挂起态->就绪态! TaskCg 状态: 804  TaskHi 状态: 802  TaskLo 状态: 802
```

5.6 实现状态转换：运行态➡就绪态（单位时间片结束）

我们在TaskCg任务中设置了while(1){} (如图2),使此任务不断在CPU上运行直到时间片结束。注意：此处时间片结束后，TaskCg任务将会添加到5优先级就绪队列的末尾(如图3)，所以系统再此调度时，运转CPU上的任务为TaskHi (如图4)

```
PRINTK("TaskLo:挂起态->就绪态! ");
ret = LOS_TaskInfoGet(g_taskCgId, &stateCg);
PRINTK("TaskCg 状态: %x  ", stateCg.usTaskStatus);
ret = LOS_TaskInfoGet(g_taskHiId, &stateHi);
PRINTK("TaskHi 状态: %x  ", stateHi.usTaskStatus);
ret = LOS_TaskInfoGet(g_taskLoId, &stateLo);
PRINTK("TaskLo 状态: %x\n", stateLo.usTaskStatus);

while(1){}
```

TaskCg任务体

```
} « end OsTaskSchedQueueDequeue »

STATIC INLINE VOID OsSchedTaskEnqueue(LosProcessCB *processCB, LosTaskCB *taskCB)
{
    if (((taskCB->policy == LOS_SCHED_RR) && (taskCB->timeSlice != 0)) ||
        ((taskCB->taskStatus & OS_TASK_STATUS_RUNNING) && (taskCB->policy == LOS_SCHED_FIFO))) {
        OS_TASK_PRI_QUEUE_ENQUEUE_HEAD(processCB, taskCB);
    } else {
        OS_TASK_PRI_QUEUE_ENQUEUE(processCB, taskCB);
    }
    taskCB->taskStatus |= OS_TASK_STATUS_READY;
}
```

就绪队列函数

```
Cg执行完初始时间片长度的时间,尾插到就绪队列尾部
--------------Enter TaskHi Handler.---------------
TaskCg:运行态->就绪态
TaskHi:就绪态->运行态
```

Hi执行 Cg被挂起

5.7 实现状态转换：运行态➡阻塞态（任务延时方法）

TaskCg任务时间片用尽后开始执行优先级相同的TaskHi任务，在TaskHi任务中我们调用ret=LOS_TaskDelete(g_taskCgId)方法，将TaskCg删除，防止因为高优先级影响我们后续的实验效果。在TaskHi任务体中我们主动调用TaskDelay() 将任务延时，使其进入阻塞态，实验结果如图所示：

```c
/* 延时12个Tick，延时后该任务会挂起，TaskHi进入挂起态*/
ret = LOS_TaskDelay(12);
if (ret != LOS_OK) {
    PRINTK("Delay Task Failed.\n");
    return LOS_NOK;
}
```

```
TaskCg 状态: 802  TaskHi 状态: 804  TaskLo 状态: 802
TaskHi主动调用删除接口，删除TaskCg任务
TaskHi延时2Tick
TaskHi:运行态->阻塞态
```

Hi被阻塞

5.8 实现状态转换：阻塞态➡就绪态（任务延时结束方法）

TaskCg任务被删除，TaskHi任务被延时，TaskLo被执行，在TaskLo中同样执行TaskDelay()操作且延时时间和TaskHi相同，所以此时TaskHi的延时结束，TaskHi由阻塞态转化为就绪态，而此时并没有任务和TaskHi抢夺，TaskHi继续执行转化为运行态，实验结果如图6所示：

```c
UINT32 Example_TaskLo(VOID)
{
    UINT32 ret;
    PRINTK("TaskHi延时2Tick\n");
    PRINTK("TaskHi:运行态->阻塞态\n");
    TSK_INFO_S stateHi,stateLo;
    ret = LOS_TaskInfoGet(g_taskHiId, &stateHi);
    PRINTK("---------------Enter TaskLo Handler.---------------\n");
    PRINTK("TaskLo:就绪态->运行态\n");
    PRINTK("TaskHi 状态: %x  ", stateHi.usTaskStatus);
    ret = LOS_TaskInfoGet(g_taskLoId, &stateLo);
    PRINTK("TaskLo 状态: %x\n", stateLo.usTaskStatus);


    /* 延时12个Tick，延时后该任务会挂起，执行剩余任务中最高优先级的任务(背景任务) */
    ret = LOS_TaskDelay(12);
    if (ret != LOS_OK) {
        PRINTK("Delay TaskLo Failed.\n");
        return LOS_NOK;
    }
```

```
--------------Enter TaskLo Handler.--------------
TaskLo:就绪态->运行态
TaskHi 状态: 820  TaskLo 状态: 804
TaskLo延时2Tick
TaskLo:运行态->阻塞态
--------------Enter TaskHi Handler.--------------
TaskHi:阻塞态->运行态
TaskLo:运行态->阻塞态
TaskHi 状态: 804  TaskLo 状态: 820
```

Hi重新运行 Lo阻塞

5.9 实现状态转换：运行态➡挂起态（调用挂起接口方法）

继续执行TaskHi的任务体，TaskHi任务体主动调用LosSuspend()，将TaskHi主动挂起，任务挂起后，TaskLo任务将继续执行，实现结果如图：

```
/* 挂起自身任务 */
ret = LOS_TaskSuspend(g_taskHiId);
if (ret != LOS_OK) {
    PRINTK("Suspend TaskHi Failed.\n");
    return LOS_NOK;
}
```

```
TaskHi调用suspend接口
TaskHi:运行态->挂起态
--------------Enter TaskLo Handler.--------------
TaskHi 状态: 808  TaskLo 状态: 804
```

Hi主动挂起

5.10 实现状态转换：挂起态➡就绪态➡运行态

TaskLo任务中调用LosResume()方法恢复TaskHi，TaskHi由挂起态转化为就绪态，如图8。注意：此处TaskLo不会继续执行以后的任务体而是因为TaskHi任务已经恢复，因为Liteos的抢断机制，TaskHi会重新获得CPU的执行，会继续执行TaskHi的任务体。

```
/* 恢复被挂起的任务g_taskHiId */
ret = LOS_TaskResume(g_taskHiId);
if (ret != LOS_OK) {
    PRINTK("Resume TaskHi Failed.\n");
    return LOS_NOK;
}
```

Lo主动恢复Hi

因为Liteos的抢断机制，当任务被抢断后，TaskHi将是运行态，TaskLo将是就绪态，结果如图9所示。

```
TaskHi被TaskLo恢复为就绪态，TaskHi抢占TaskLo.
---------------Enter TaskHi Handler.---------------
TaskHi:挂起态->就绪态->运行态
TaskLo:运行态->就绪态
TaskHi 状态: 804  TaskLo 状态: 802
```

Lo被Hi抢占

5.11 任务退出

TaskHi抢占了TaskLo，所以TaskHi会先获得CPU的执行权力，所以结果中TaskHi优先于TaskLo结束

```
TaskHi执行完成，自删除成功.
---------------Enter TaskLo Handler.---------------
TaskLo执行完成，自删除成功.
```

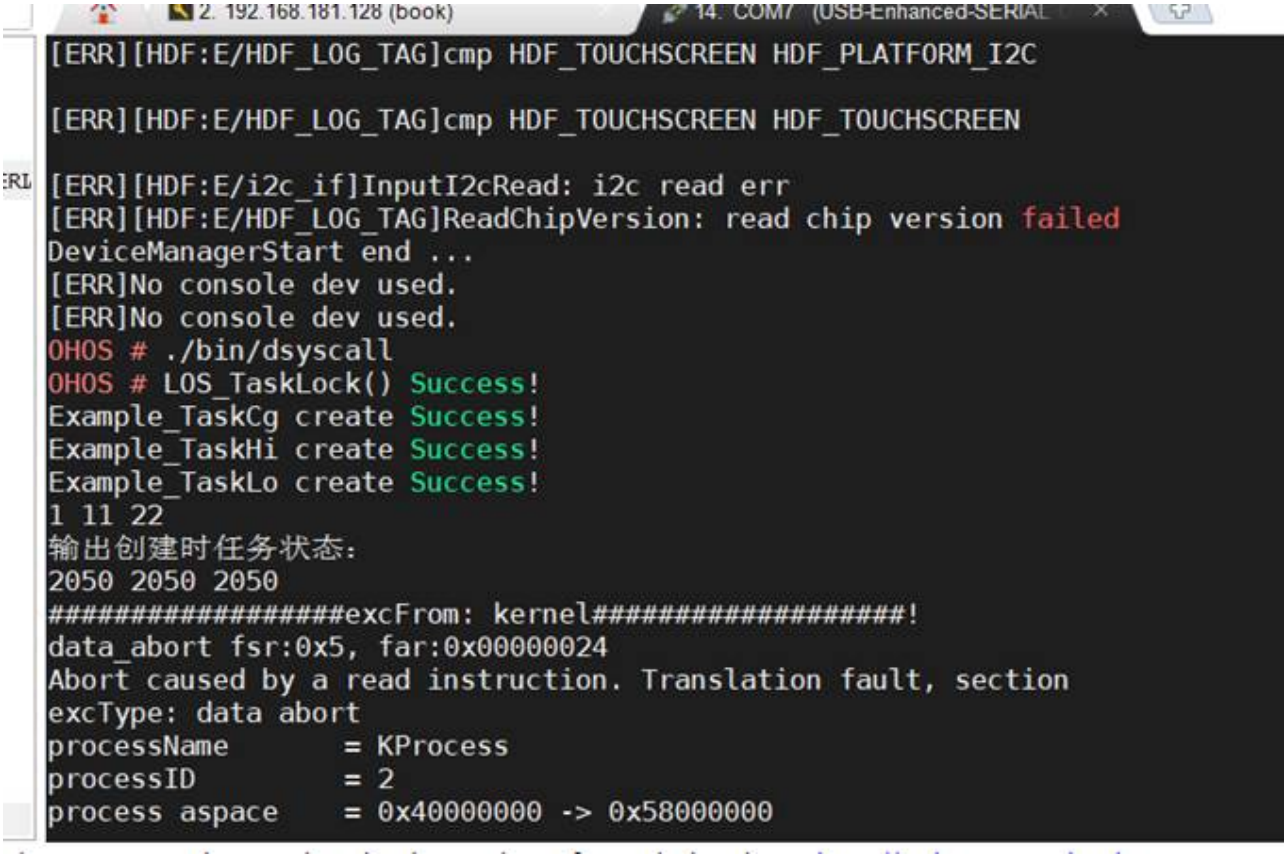
6 实验结果

任务中调用主创建任务作为起始点，按调度顺序执行三个任务，最后输出的结果如图所示，结果所示即反应了任务调度的情况。

```
OHOS #
LOS_TaskLock() 成功，防止新创建的任务比本任务高而发生调度!
TaskCg TaskHi TaskLo 创建成功!
TaskCg 状态: 802  TaskHi 状态: 802  TaskLo 状态: 802
TaskLo:就绪态->挂起态!
TaskCg 状态: 802  TaskHi 状态: 802  TaskLo 状态: 808
三个任务优先级分别为: 5 5 22
TaskCg.priority==TaskHi.priority,TaskCg在初始化任务优先创建，同一优先级就绪队列
中先进先出
---------------Enter TaskCg Handler.---------------
TaskCg:就绪态->运行态! TaskCg 状态: 804  TaskHi 状态: 802  TaskLo 状态: 808
TaskLo:挂起态->就绪态! TaskCg 状态: 804  TaskHi 状态: 802  TaskLo 状态: 802
Cg执行完初始时间片长度的时间,尾插到就绪队列尾部
---------------Enter TaskHi Handler.---------------
TaskCg:运行态->就绪态
TaskHi:就绪态->运行态
TaskCg 状态: 802  TaskHi 状态: 804  TaskLo 状态: 802
TaskHi主动调用删除接口，删除TaskCg任务
TaskHi延时2Tick
TaskHi:运行态->阻塞态
---------------Enter TaskLo Handler.---------------
TaskLo:就绪态->运行态
TaskHi 状态: 820  TaskLo 状态: 804
TaskLo延时2Tick
TaskLo:运行态->阻塞态
---------------Enter TaskHi Handler.---------------
TaskHi:阻塞态->运行态
TaskLo:运行态->阻塞态
TaskHi 状态: 804  TaskLo 状态: 820
TaskHi调用suspend接口
TaskHi:运行态->挂起态
---------------Enter TaskLo Handler.---------------
TaskHi 状态: 808  TaskLo 状态: 804
TaskHi被TaskLo恢复为就绪态，TaskHi抢占TaskLo.
---------------Enter TaskHi Handler.---------------
TaskHi:挂起态->就绪态->运行态
TaskLo:运行态->就绪态
TaskHi 状态: 804  TaskLo 状态: 802
TaskHi执行完成，自删除成功.
---------------Enter TaskLo Handler.---------------
TaskLo执行完成，自删除成功.
```

7 实验难点

1. IMX6ULL-MINI板子的局限性，本实验意图研究任务的亲核性问题，但是板子本身的局限性让我们无法进行本次实验
2. IMX6ULL-MINI板子的自身驱动欠佳，许多实验相关的驱动需要自己编码
3. 操作系统代码维护问题，HarmonyOS作为新生操作系统，版本更新迭代快，如果不锁定其中相应的版本，会出现函数调用出问题，相关功能确实等情况
4. 在实验实现时出现了系统BUG，系鸿蒙1.0的系统BUG，在升级版本以及修复。报错问题情况，以及官方回应如下：


2. 192.168.181.128 (book)
14. COM7 (USB-Enhanced-SERIAL
[ERR][HDF:E/HDF_LOG_TAG]cmp HDF_TOUCHSCREEN HDF_PLATFORM_I2C
[ERR][HDF:E/HDF_LOG_TAG]cmp HDF_TOUCHSCREEN HDF_TOUCHSCREEN
[ERR][HDF:E/i2c_if]InputI2cRead: i2c read err
[ERR][HDF:E/HDF_LOG_TAG]ReadChipVersion: read chip version failed
DeviceManagerStart end ...
[ERR]No console dev used.
[ERR]No console dev used.
OHOS # ./bin/dsyscall
OHOS # LOS_TaskLock() Success!
Example_TaskCg create Success!
Example_TaskHi create Success!
Example_TaskLo create Success!
1 11 22
输出创建时任务状态:
2050 2050 2050
####################excFrom: kernel####################!
data_abort fsr:0x5, far:0x00000024
Abort caused by a read instruction. Translation fault, section
excType: data abort
processName = KProcess
processID = 2
process aspace = 0x40000000 -> 0x58000000


开源项目 > OpenHarmony && 其他开源 > 操作系统
OpenHarmony / kernel_liteos_a
代码
Issues 33
Pull Requests 1
Wiki
统计
流水线
服务
Issues / 详情
3.0 LTS，在QEMU执行时，报错，不出现鸿蒙命令行
已完成
#I4EIWU
缺陷
Jianwei Mao
创建于 2021-10-19 17:51
【问题】
3.0 LTS，在QEMU执行时，报错，不出现鸿蒙命令行
【命令行输出】
Welcome
Processor : Cortex-A7
Run Mode : UP
GIC Rev : GICv2
build time : Oct 19 2021 16:14:59
Kernel : Huawei LiteOS 2.0.0.37/debug
main core booting up...
cpu 0 entering scheduler
mem dev init ...

8 实验心得

通过本次实验我们小组了解到了，想要设计一个有效的实验必须先将基础的代码全部搞懂，因为你设计一个实验时，是一个从上到下的过程，例如在我们的实验中创建顺序也会影响任务执行顺序，我们可以简单的通过实验得出，但这不能反应问题的本质，真正问题的要求是通过实验设计促进对代码以及知识的理解程度。

并且本次实验中从无到有的过程，也让我增加搜索知识以及信息的能力，完成一个良好的实验设计，需要我们对网上的信息进行这边甄别。

通过这次实验，对liteos-a的任务管理机制有了更深入的了解。在设计实验时，学习了LiteOS-A的任务调度算法和状态切换的各个函数，这些知识对于理解操作系统的任务管理机制非常有帮助。通过LiteOS-A的任务管理实验设计，我不仅加深了对任务管理机制的理解，还提升了实验设计和操作底层代码的能力。这次实验让我更加熟悉操作系统的实际应用，对我的学习和研究有着积极的影响。