任务管理阅读心得

我们小组负责的是鸿蒙系统代码的任务管理模块，我们将阅读代码与设计部分分为4块：任务控制块、任务管理、任务状态切换以及实验设计，具体分工如下：

阅读代码：

- 任务控制块——胡翼翔
- 任务管理——黄勖
- 任务状态切换——陈伊彬

实验设计（排序不分先后）：石宇昊、邹慧、曾鸿勇

以下是我们阅读代码的总结与心得收获：

1 基本概念

- 任务是竞争系统资源的最小运行单元。任务可以使用或等待CPU、使用内存空间等系统资源，并独立于其它任务运行
- 任务与线程
  - 在 LiteOS 中，一个任务可以表示一条线程。
- Huawei LiteOS的任务一共有 **32** 个优先级(0-31)，**最高**优先级为**0**，最低优先级为31。
- **任务状态**
  - 就绪 （Ready）
  - 运行（Running）
  - 运行（Running）
  - 退出态（Dead）
- **任务ID**
  - 可通过任务ID获取任务句柄
    - `***任务句柄*** = (((LOS_TASK_CB *)g_pstTaskCBArray) + (TaskID))`
- **任务控制块TCB**
  - **TCB** 包含了任务上下文栈指针（stack pointer）、任务状态、任务优先级、任务ID、任务名、任务栈大小等信息。**TCB** 可以反映出每个任务运行情况。（**TCB** 其实就是一个**袜子**）
- **任务栈**
  - 每一个任务都拥有一个独立的栈空间，称为任务栈。
- **任务上下文切换** （个人把上下文分开理解）
  - ** Huawei LiteOS** 在任务由运行态转为其它状态时会将本任务的上文信息，保存在自己的任务栈里面，也称压栈或入栈；
  - 当任务切换到运行态时，把保存到任务栈中的上文信息加载到CPU寄存器中，即可恢复该任务的运行，称为出栈。新的任务信息也就是下文。
  - 以上流程就是 **上文压栈 --> 下文出栈**。
- Huawei LiteOS任务管理模块提供
  - 任务创建
  - 任务删除
  - 任务延时
  - 任务挂起

  - 任务恢复
  - 更改任务优先级
  - 锁定任务调度
  - 解锁任务调度
  - 根据任务控制块查询任务ID
  - 根据ID查询任务查询任务控制块信息功能。

- 任务创建时，如果 OS 的系统可用空间少于任务，则创建失败，反之亦然。

- 用户创建任务时，系统会将任务栈进行初始化，预置上下文。

  - **任务入口函数** 也放到了相应的位置，在任务第一次执行时便可执行 **任务入口函数**。

2 任务控制块

任务控制块(LosTaskCB)的数据结构如下：

```
typedef struct {
    VOID            *stackPointer;      /**< Task stack pointer */ //内核态栈指
针，SP位置，切换任务时先保存上下文并指向TaskContext位置
    UINT16          taskStatus;         /**< Task status */   //各种状态标签，可
以拥有多种标签，按位标识
    UINT16          priority;           /**< Task priority */  //任务优先级
[0:31]，默认是31级
    UINT16          policy;    //任务的调度方式(三种 .. LOS_SCHED_RR )
    UINT16          timeSlice;          /**< Remaining time slice *///剩余时间片
    UINT32          stackSize;          /**< Task stack size */  //非用户模式下
栈大小
    UINTPTR         topOfStack;         /**< Task stack top */  //非用户模式下的
栈顶 bottom = top + size
    UINT32          taskID;             /**< Task ID */    //任务ID，任务池本质
是一个大数组，ID就是数组的索引，默认 < 128
    TSK_ENTRY_FUNC  taskEntry;          /**< Task entrance function */ //任务执
行入口函数
    VOID            *joinRetval;        /**< pthread adaption */ //用来存储join
线程的返回值
    VOID            *taskSem;           /**< Task-held semaphore */ //task在等
哪个信号量
    VOID            *taskMux;           /**< Task-held mutex */  //task在等哪把
锁
    VOID            *taskEvent;         /**< Task-held event */  //task在等哪个
事件
    UINTPTR         args[4];            /**< Parameter,  of which the maximum
number is 4 */ //入口函数的参数 例如 main (int argc, char *argv[])
    CHAR            taskName[OS_TCB_NAME_LEN]; /**< Task name */ //任务的名称
    LOS_DL_LIST     pendList;           /**< Task pend node */  //如果任务阻塞时
就通过它挂到各种阻塞情况的链表上，比如OsTaskWait时
    LOS_DL_LIST     threadList;         /**< thread list */   //挂到所属进程的线
程链表上
```

```
    SortLinkList    sortList;           /**< Task sortlink node */ //挂到cpu
core 的任务执行链表上
    UINT32          eventMask;          /**< Event mask */   //事件屏蔽
    UINT32          eventMode;          /**< Event mode */   //事件模式
    UINT32          priBitMap;          /**< BitMap for recording the change of
task priority,  //任务在执行过程中优先级会经常变化，这个变量用来记录所有曾经变化
                                            the priority can not be greater
than 31 */   //过的优先级，例如 ..01001011 曾经有过 0, 1, 3, 6 优先级
    INT32           errorNo;            /**< Error Num */
    UINT32          signal;             /**< Task signal */ //任务信号类型,
(SIGNAL_NONE, SIGNAL_KILL, SIGNAL_SUSPEND, SIGNAL_AFFI)
    sig_cb          sig;    //信号控制块，这里用于进程间通讯的信号，类似于 linux
singal模块
#if (LOSCFG_KERNEL_SMP == YES)
    UINT16          currCpu;            /**< CPU core number of this task is
running on */ //正在运行此任务的CPU内核号
    UINT16          lastCpu;            /**< CPU core number of this task is
running on last time */ //上次运行此任务的CPU内核号
    UINT16          cpuAffiMask;        /**< CPU affinity mask,  support up to
16 cores */ //CPU亲和力掩码，最多支持16核，亲和力很重要，多核情况下尽量一个任务在一
个CPU核上运行，提高效率
    UINT32          timerCpu;           /**< CPU core number of this task is
delayed or pended */ //此任务的CPU内核号被延迟或挂起
#if (LOSCFG_KERNEL_SMP_TASK_SYNC == YES)
    UINT32          syncSignal;         /**< Synchronization for signal
handling */ //用于CPU之间 同步信号
#endif
#if (LOSCFG_KERNEL_SMP_LOCKDEP == YES) //死锁检测开关
    LockDep         lockDep;
#endif
#if (LOSCFG_KERNEL_SCHED_STATISTICS == YES) //调度统计开关，显然打开这个开关性能
会受到影响，鸿蒙默认是关闭的
    SchedStat       schedStat;          /**< Schedule statistics */ //调度统计
#endif
#endif
    UINTPTR         userArea;   //使用区域，由运行时划定，根据运行态不同而不同
    UINTPTR         userMapBase;  //用户模式下的栈底位置
    UINT32          userMapSize;        /**< user thread stack size , real size
: userMapSize + USER_STACK_MIN_SIZE */
    UINT32          processID;          /**< Which belong process *///所属进程ID
    FutexNode       futex;    //实现快锁功能
    LOS_DL_LIST     joinList;           /**< join list */ //联结链表，允许任务之
间相互释放彼此
    LOS_DL_LIST     lockList;           /**< Hold the lock list */ //拿到了哪些
锁链表
    UINT32          waitID;             /**< Wait for the PID or GID of the
child process */ //等待孩子的PID或GID进程
```

```
    UINT16          waitFlag;           /**< The type of child process that is
waiting,  belonging to a group or parent,
                                            a specific child process,  or any
child process */
#if (LOSCFG_KERNEL_LITEIPC == YES)
    UINT32          ipcStatus;    //IPC状态
    LOS_DL_LIST     msgListHead;  //消息队列头结点，上面挂的都是任务要读的消息
    BOOL            accessMap[LOSCFG_BASE_CORE_TSK_LIMIT];//访问图，指的是task之
间是否能访问的标识，LOSCFG_BASE_CORE_TSK_LIMIT 为任务池总数
#endif
} LosTaskCB;
```

结构体还是比较复杂，虽一一都做了注解，但还是不够清晰，没有模块化。这里把它分解成以下六大块逐一分析:

2.1 多核CPU相关块

```
#if (LOSCFG_KERNEL_SMP == YES) //多CPU核支持
    UINT16          currCpu;            /**< CPU core number of this task is
running on */ //正在运行此任务的CPU内核号
    UINT16          lastCpu;            /**< CPU core number of this task is
running on last time */ //上次运行此任务的CPU内核号
    UINT16          cpuAffiMask;        /**< CPU affinity mask,  support up to
16 cores */ //CPU亲和力掩码，最多支持16核，亲和力很重要，多核情况下尽量一个任务在一
个CPU核上运行，提高效率
    UINT32          timerCpu;           /**< CPU core number of this task is
delayed or pended */ //此任务的CPU内核号被延迟或挂起
#if (LOSCFG_KERNEL_SMP_TASK_SYNC == YES)
    UINT32          syncSignal;         /**< Synchronization for signal
handling */ //用于CPU之间 同步信号
#endif
#if (LOSCFG_KERNEL_SMP_LOCKDEP == YES) //死锁检测开关
    LockDep         lockDep;
#endif
#if (LOSCFG_KERNEL_SCHED_STATISTICS == YES) //调度统计开关，显然打开这个开关性能
会受到影响，鸿蒙默认是关闭的
    SchedStat       schedStat;          /**< Schedule statistics */ //调度统计
#endif
#endif
```

鸿蒙内核支持多CPU，谁都知道多CPU当然好，效率高，快嘛，但凡事有两面性，在享受一个东西带来好处的同时，也得承担伴随它一起带来的麻烦和风险。多核有哪些的好处和麻烦，这里不展开说，后续有专门的文章和视频说明。任务可叫线程，或叫作业。CPU就是做作业的，多个CPU就是有多个能做作业的，一个作业能一鼓作气做完吗?

答案是:往往不行，因为现实不允许，作业可以有N多，而CPU数量非常有限，所以经常做着A作业被老板打断让去做B作业。这老板就是调度算法。A作业被打断回来接着做的还会是原来那个CPU吗?

答案是:不一定。 变量cpuAffiMask叫CPU亲和力，它的作用是可以指定A的作业始终是同一个CPU来完成， 也可以随便，交给调度算法，分到谁就谁来，这方面可以不挑。

2.2 栈空间

```
    VOID            *stackPointer;      /**< Task stack pointer */  //内核态栈指针，SP位置，切换任务时先保存上下文并指向TaskContext位置。
    UINT32          stackSize;          /**< Task stack size */     //内核态栈大小
    UINTPTR         topOfStack;         /**< Task stack top */      //内核态栈顶 bottom = top + size

    UINTPTR         userArea;       //使用区域，由运行时划定，根据运行态不同而不同
    UINTPTR         userMapBase;    //用户态下的栈底位置
    UINT32          userMapSize;    /**< user thread stack size , real size : userMapSize + USER_STACK_MIN_SIZE */
```

进程分内核态进程和用户态进程，这个区别表现在线程(任务)层面上就是

- 内核态进程下创建的任务只有内核态的栈空间，`OsTaskStackAlloc`负责内核态栈空间的分配。`OsTaskStackInit`负责对内核态栈的初始化。

```
//任务栈初始化，非常重要的函数，返回任务上下文
LITE_OS_SEC_TEXT_INIT VOID *OsTaskStackInit(UINT32 taskID,  UINT32 stackSize, VOID *topStack,  BOOL initFlag)
{
    UINT32 index = 1;
    TaskContext *taskContext = NULL;

    if (initFlag == TRUE) {
        OsStackInit(topStack,  stackSize);
    }
    taskContext = (TaskContext *)(((UINTPTR)topStack + stackSize) - sizeof(TaskContext));//上下文存放在栈的底部

    /* initialize the task context */ //初始化任务上下文
#ifdef LOSCFG_GDB
    taskContext->PC = (UINTPTR)OsTaskEntrySetupLoopFrame;
#else
    taskContext->PC = (UINTPTR)OsTaskEntry;//程序计数器，CPU首次执行task时跑的第一条指令位置
#endif
    taskContext->LR = (UINTPTR)OsTaskExit;  /* LR should be kept,  to distinguish it's THUMB or ARM instruction */
```

```
    taskContext->resved = 0x0;
    taskContext->R[0] = taskID;                 /* R0 */
    taskContext->R[index++] = 0x01010101;    /* R1,  0x01010101 : reg initialed
magic word */ //0x55
    for (; index < GEN_REGS_NUM; index++) {//R2 - R12的初始化很有意思，为什么要这
么做?
        taskContext->R[index] = taskContext->R[index - 1] + taskContext->R[1];
/* R2 - R12 */
    }//R[2]=R[2]<<1=0xAA

#ifdef LOSCFG_INTERWORK_THUMB // 16位模式
    taskContext->regPSR = PSR_MODE_SVC_THUMB; /* CPSR (Enable IRQ and FIQ
interrupts,  THUMNB-mode) */
#else //用于设置CPSR寄存器
    taskContext->regPSR = PSR_MODE_SVC_ARM;   /* CPSR (Enable IRQ and FIQ
interrupts,  ARM-mode) */
#endif

#if !defined(LOSCFG_ARCH_FPU_DISABLE)
    /* 0xAAA0000000000000LL : float reg initialed magic word */
    for (index = 0; index < FP_REGS_NUM; index++) {
        taskContext->D[index] = 0xAAA0000000000000LL + index; /* D0 - D31 */
    }
    taskContext->regFPSCR = 0;
    taskContext->regFPEXC = FP_EN;
#endif

    return (VOID *)taskContext;
}
```

可以看到，初始化了任务上下文(TaskContext)，并将任务上下文放在了栈底，初始化任务上下文目的是为了在运行阶段先初始化R0~R15，CPSR寄存器的值。 保存上下文和恢复上下文都是针对寄存器值而言的。这个工作是在内核态的栈中完成的，也就是说一个任务的上下文就是保存在任务的内核态栈中。 `OsTaskStackInit`的返回值将赋给`stackPointer`，即寄存器SP

- 用户态进程下创建的任务除了有内核态的栈空间外，还有用户态栈空间。

```
//用户任务使用栈初始化
LITE_OS_SEC_TEXT_INIT VOID OsUserTaskStackInit(TaskContext *context,
TSK_ENTRY_FUNC taskEntry,  UINTPTR stack)
{
    LOS_ASSERT(context != NULL);

#ifdef LOSCFG_INTERWORK_THUMB
    context->regPSR = PSR_MODE_USR_THUMB;
#else
    context->regPSR = PSR_MODE_USR_ARM;//工作模式:用户模式 + 工作状态:arm
#endif
```

```
    context->R[0] = stack;//栈指针给r0寄存器
    context->SP = TRUNCATE(stack,  LOSCFG_STACK_POINT_ALIGN_SIZE);//异常模式所专用的堆栈 segment fault 输出回溯信息
    context->LR = 0;//保存子程序返回地址 例如 a call b , 在b中保存 a地址
    context->PC = (UINTPTR)taskEntry;//入口函数
}
```

2.3 资源竞争/同步

```
    VOID            *taskSem;           /**< Task-held semaphore */ //task在等哪个信号量
    VOID            *taskMux;           /**< Task-held mutex */  //task在等哪把锁
    VOID            *taskEvent;         /**< Task-held event */  //task在等哪个事件
    UINT32          eventMask;          /**< Event mask */   //事件屏蔽
    UINT32          eventMode;          /**< Event mode */   //事件模式
    FutexNode       futex;    //实现快锁功能
    LOS_DL_LIST     joinList;           /**< join list */ //联结链表，允许任务之间相互释放彼此
    LOS_DL_LIST     lockList;           /**< Hold the lock list */ //拿到了哪些锁链表
    UINT32          signal;             /**< Task signal */ //任务信号类型,(SIGNAL_NONE, SIGNAL_KILL, SIGNAL_SUSPEND, SIGNAL_AFFI)
    sig_cb          sig;
```

公司的资源是有限的，CPU自己也是公司的资源，除了它还有其他的设备，比如做作业用的黑板，用户A，B，C都可能用到，狼多肉少，咋搞?

互斥量(taskMux，futex)能解决这个问题，办事前先拿锁，拿到了锁的爽了，没有拿到的就需要排队，在lockList上排队，注意lockList是个双向链表，它是内核最重要的结构体，开篇就提过，没印象的看 (双向链表篇)，上面挂都是等锁进房间的西门大官人。这是互斥量的原理，解决任务间资源紧张的竞争性问题。

另外一个是用于任务的同步的信号量(sig_cb)，任务和任务之间是会有关联的，现实生活中公司的A，B用户之间本身有业务往来的正常，CPU在帮B做作业的时候发现前置条件是需要A完成某项作业才能进行，这时B就需要主动让出CPU先办完A的事。这就是**信号量**的原理，解决的是任务间的同步问题。

2.4 任务调度

前面说过了作业N多，做作业的只有几个人，单核CPU等于只有一个人干活。那要怎么分配CPU，就需要调度算法。

```
    UINT16          taskStatus;         /**< Task status */   //各种状态标签，可
以拥有多种标签，按位标识
    UINT16          priority;           /**< Task priority */  //任务优先级
[0:31]，默认是31级
    UINT16          policy;    //任务的调度方式(三种 .. LOS_SCHED_RR )
    UINT16          timeSlice;          /**< Remaining time slice *///剩余时间片
    CHAR            taskName[OS_TCB_NAME_LEN]; /**< Task name */ //任务的名称
    LOS_DL_LIST     pendList;           /**< Task pend node */  //如果任务阻塞时
就通过它挂到各种阻塞情况的链表上，比如OsTaskWait时
    LOS_DL_LIST     threadList;         /**< thread list */   //挂到所属进程的线
程链表上
    SortLinkList    sortList;           /**< Task sortlink node */ //挂到cpu
core 的任务执行链表上
```

是简单的先来后到(FIFO)吗？ 当然也支持这个方式。鸿蒙内核用的是抢占式调度(policy)，就是可以插队，比优先级(priority)大小，[0，31]级，数字越大的优先级越低，跟考试一样，排第一才是最牛的。

鸿蒙排0的最牛! 想也想得到内核的任务优先级都是很高的，比如资源回收任务排第5，定时器任务排第0。够牛了吧。普通老百姓排多少呢？默认28级，惨!!!

另外任务有时间限制timeSlice，叫时间片，默认20ms，用完了会给你重置，发起重新调度，找出优先级高的执行，阻塞的任务(比如没拿到锁的，等信号量同步的，等读写消息队列的)都挂到pendList上，方便管理。

2.5 任务间通讯

```
#if (LOSCFG_KERNEL_LITEIPC == YES)
    UINT32          ipcStatus;   //IPC状态
    LOS_DL_LIST     msgListHead;  //消息队列头结点，上面挂的都是任务要读的消息
    BOOL            accessMap[LOSCFG_BASE_CORE_TSK_LIMIT];//访问图，指的是task之
间是否能访问的标识，LOSCFG_BASE_CORE_TSK_LIMIT 为任务池总数
#endif
```

这个很重要，解决任务间通讯问题，要知道进程负责的是资源的管理功能，什么意思？就是它不并负责内容的生产和消费，它只负责管理确保你的内容到达率和完整性。生产者和消费者始终是任务。进程管了哪些东西系列篇有专门的文章，请自行翻看。

liteipc是鸿蒙专有的通讯消息队列实现。简单说它是基于文档的，而传统的ipc消息队列是基于内存的。有什么区别也不在这里讨论，已有专门的文章分析。

2.6 辅助工具

要知道任务对内核来说太重要了，是任务让CPU忙里忙外的，那中间出差错了怎么办，怎么诊断你问题出哪里了，就需要一些工具，比如死锁检测，比如占用CPU，内存监控 如下:

```
#if (LOSCFG_KERNEL_SMP_LOCKDEP == YES) //死锁检测开关
    LockDep         lockDep;
#endif
#if (LOSCFG_KERNEL_SCHED_STATISTICS == YES) //调度统计开关，显然打开这个开关性能会受到影响，鸿蒙默认是关闭的
    SchedStat       schedStat;          /**< Schedule statistics */ //调度统计
#endif
```

3 任务管理

从系统的角度看，线程是竞争系统资源的最小运行单元。线程可以使用或等待CPU、使用内存空间等系统资源，并独立于其它线程运行。

鸿蒙内核每个进程内的线程独立运行、独立调度，当前进程内线程的调度不受其它进程内线程的影响。

鸿蒙内核中的线程采用抢占式调度机制，同时支持时间片轮转调度和FIFO调度方式。

鸿蒙内核的线程一共有32个优先级(0-31)，最高优先级为0，最低优先级为31。

当前进程内高优先级的线程可抢占当前进程内低优先级线程，当前进程内低优先级线程必须在当前进程内高优先级线程阻塞或结束后才能得到调度。

那么任务怎么管理呢?

3.1 任务池

前面已经说了任务是内核调度层面的概念，调度算法保证了task有序的执行，调度机制详见其他姊妹篇的介绍。
如此多的任务怎么管理和执行？管理靠任务池和就绪队列，执行靠调度算法。

代码如下：

```
LITE_OS_SEC_TEXT_INIT UINT32 OsTaskInit(VOID)
{
    UINT32 index;
    UINT32 ret;
    UINT32 size;

    g_taskMaxNum = LOSCFG_BASE_CORE_TSK_LIMIT;//任务池中最多默认128个，可谓铁打的任务池流水的线程
    size = (g_taskMaxNum + 1) * sizeof(LosTaskCB);//计算需分配内存总大小
    /*
     * This memory is resident memory and is used to save the system resources
     * of task control block and will not be freed。
     */
    g_taskCBArray = (LosTaskCB *)LOS_MemAlloc(m_aucSysMem0,  size);//任务池 常驻内存，不被释放
    if (g_taskCBArray == NULL) {
```

```
        return LOS_ERRNO_TSK_NO_MEMORY;
    }
    (VOID)memset_s(g_taskCBArray,  size,  0,  size);

    LOS_ListInit(&g_losFreeTask);//空闲任务链表
    LOS_ListInit(&g_taskRecyleList);//需回收任务链表
    for (index = 0; index < g_taskMaxNum; index++) {
        g_taskCBArray[index]。taskStatus = OS_TASK_STATUS_UNUSED;
        g_taskCBArray[index]。taskID = index;//任务ID最大默认127
        LOS_ListTailInsert(&g_losFreeTask,  &g_taskCBArray[index]。pendList);//
都插入空闲任务列表
    }//注意:这里挂的是pendList节点，所以取TCB要通过 OS_TCB_FROM_PENDLIST 取。

    ret = OsPriQueueInit();//创建32个任务优先级队列，即32个双向循环链表
    if (ret != LOS_OK) {
        return LOS_ERRNO_TSK_NO_MEMORY;
    }

    /* init sortlink for each core */
    for (index = 0; index < LOSCFG_KERNEL_CORE_NUM; index++) {
        ret = OsSortLinkInit(&g_percpu[index]。taskSortLink);//每个CPU内核都有一
个执行任务链表
        if (ret != LOS_OK) {
            return LOS_ERRNO_TSK_NO_MEMORY;
        }
    }
    return LOS_OK;
}
```

g_taskCBArray 就是个任务池，默认创建128个任务，常驻内存，不被释放。
g_losFreeTask是空闲任务链表，想创建任务时来这里申请一个空闲任务，用完了就回收掉，继续给后面的申请使用。
g_taskRecyleList是回收任务链表，专用来回收exit 任务，任务所占资源被确认归还后被彻底删除，就像员工离职一样，得有个离职队列和流程，要归还电脑，邮箱，有没有借钱要还的 等操作。

对应张大爷的故事：用户要来场馆领取表格填节目单，场馆只准备了128张表格，领完就没有了，但是节目表演完了会回收表格，这样多了一张表格就可以给其他人领取了，这128张表格对应鸿蒙内核这就是任务池，简单吧。

3.2 就绪队列

CPU执行速度是很快的，鸿蒙内核默认一个时间片是 10ms， 资源有限，需要在众多任务中来回的切换，所以绝不能让CPU等待任务，CPU就像公司最大的领导，下面很多的部门等领导来审批，吃饭。只有大家等领导，哪有领导等你们的道理，所以工作要提前准备好，每个部门的优先级又不一样，所以每个部门都要有个任务队列，里面放的是领导能直接处理的任务，没准备好的不要放进来，因为这是给CPU提前准备好的粮食！

这就是就绪队列的原理，一共有32个就绪队列，进程和线程都有，因为线程的优先级是默认32个， 每个队列中放同等优先级的task。

```
#define OS_PRIORITY_QUEUE_NUM 32
LITE_OS_SEC_BSS LOS_DL_LIST *g_priQueueList = NULL;//队列链表
LITE_OS_SEC_BSS UINT32 g_priQueueBitmap;//队列位图 UINT32每位代表一个优先级，共32个优先级
//内部队列初始化
UINT32 OsPriQueueInit(VOID)
{
    UINT32 priority;

    /* system resident resource *///常驻内存
    g_priQueueList = (LOS_DL_LIST *)LOS_MemAlloc(m_aucSysMem0,
(OS_PRIORITY_QUEUE_NUM * sizeof(LOS_DL_LIST)));//分配32个队列头节点
    if (g_priQueueList == NULL) {
        return LOS_NOK;
    }

    for (priority = 0; priority < OS_PRIORITY_QUEUE_NUM; ++priority) {
        LOS_ListInit(&g_priQueueList[priority]);//队列初始化，前后指针指向自己
    }
    return LOS_OK;
}
```

注意看`g_priQueueList`的内存分配，就是`32`个`LOS_DL_LIST`，还记得`LOS_DL_LIST`的妙用吗，不清楚去翻双向链表篇。

对应张大爷的故事：就是门口那些排队的都是至少有一个节目单是符合表演标准的，资源都到位了，没有的连排队的资格都木有，就慢慢等吧。

3.3 任务栈

每个任务都是独立开的，任务之间也相互独立，之间通讯通过IPC，这里的“独立”指的是每个任务都有自己的运行环境 —— 栈空间，称为任务栈，栈空间里保存的信息包含局部变量、寄存器、函数参数、函数返回地址等等
但系统中只有一个CPU，任务又是独立的，调度的本质就是CPU执行一个新task，老task在什么地方被中断谁也不清楚，是随机的。那如何保证老任务被再次调度选中时还能从上次被中断的地方继续玩下去呢?

答案是：任务上下文，CPU内有一堆的寄存器，CPU运行本质的就是这些寄存器的值不断的变化，只要切换时把这些值保存起来，再还原回去就能保证task的连续执行，让用户毫无感知。鸿蒙内核给一个任务执行的时间是 20ms ，也就是说有多任务竞争的情况下，一秒钟内最多要来回切换50次。

对应张大爷的故事：就是碰到节目没有表演完就必须打断的情况下，需要把当时的情况记录下来，比如小朋友在演躲猫猫的游戏，一半不演了，张三正在树上，李四正在厕所躲，都记录下来，下次再回来你们上次在哪就会哪呆着去，就位了继续表演。这样就接上了，观众就木有感觉了。
任务上下文(TaskContext)是怎样的呢？还是直接看源码

```
/* The size of this structure must be smaller than or equal to the size
specified by OS_TSK_STACK_ALIGN (16 bytes). */
typedef struct { //参考OsTaskSchedule来理解
#if !defined(LOSCFG_ARCH_FPU_DISABLE) //支持浮点运算
    UINT64 D[FP_REGS_NUM]; /* D0-D31 */
    UINT32 regFPSCR;       /* FPSCR */
    UINT32 regFPEXC;       /* FPEXC */
#endif
    UINT32 R4;
    UINT32 R5;
    UINT32 R6;
    UINT32 R7;
    UINT32 R8;
    UINT32 R9;
    UINT32 R10;
    UINT32 R11;

    /* It has the same structure as IrqContext */
    UINT32 reserved2; /**< Multiplexing registers, used in interrupts and
system calls but with different meanings */
    UINT32 reserved1; /**< Multiplexing registers, used in interrupts and
system calls but with different meanings */
    UINT32 USP;       /**< User mode sp register */
    UINT32 ULR;       /**< User mode lr register */
    UINT32 R0;
    UINT32 R1;
    UINT32 R2;
    UINT32 R3;
    UINT32 R12;
    UINT32 LR;
    UINT32 PC;
    UINT32 regCPSR;
} TaskContext;
```

发现基本都是`CPU`寄存器的恢复现场值， 具体各寄存器有什么作用大家可以去网上详查，后续也有专门的文章来介绍。这里说其中的三个寄存器 `SP`， `LR`， `PC`

`LR`
用途有二，一是保存子程序返回地址，当调用`BL`、`BX`、`BLX`等跳转指令时会自动保存返回地址到`LR`；二是保存异常发生的异常返回地址。

`PC (Program Counter)`
为程序计数器，用于保存程序的执行地址，在ARM的三级流水线架构中，程序流水线包括取址、译码和执行三个阶段，`PC`指向的是当前取址的程序地址，所以`32`位`ARM`中，译码地址（正在解析还未执行的程序）为`PC-4`，执行地址（当前正在执行的程序地址）为`PC-8`， 当突然发生中断的时候，保存的是`PC`的地址。

`SP`
每一种异常模式都有其自己独立的`r13`，它通常指向异常模式所专用的堆栈，当ARM进入异常模式的时候，程序就可以把一般通用寄存器压入堆栈，返回时再出栈，保证了各种模式下程序的状态的完整性。

3.3.1 任务栈初始化

任务栈的初始化就是任务上下文的初始化，因为任务没开始执行，里面除了上下文不会有其他内容，注意上下文存放的位置在栈的底部。初始状态下 `sp`就是指向的栈底， 栈顶内容永远是`0xCCCCCCCC` "烫烫烫烫"，这几个字应该很熟悉吗？ 如果不是那几个字了，那说明栈溢出了，后续篇会详细说明这块，大家也可以自行去看代码，很有意思。

```
/// 内核态任务运行栈初始化
LITE_OS_SEC_TEXT_INIT VOID *OsTaskStackInit(UINT32 taskID, UINT32 stackSize,
VOID *topStack, BOOL initFlag)
{
    if (initFlag == TRUE) {
        OsStackInit(topStack, stackSize);
    }
    TaskContext *taskContext = (TaskContext *)(((UINTPTR)topStack + stackSize)
- sizeof(TaskContext));//上下文存放在栈的底部
    /* initialize the task context */ //初始化任务上下文
#ifdef LOSCFG_GDB
    taskContext->PC = (UINTPTR)OsTaskEntrySetupLoopFrame;
#else
    taskContext->PC = (UINTPTR)OsTaskEntry;//内核态任务有统一的入口地址.
#endif
    taskContext->LR = (UINTPTR)OsTaskExit;  /* LR should be kept, to
distinguish it's THUMB or ARM instruction */
    taskContext->R0 = taskID;               /* R0 */
#ifdef LOSCFG_THUMB
    taskContext->regCPSR = PSR_MODE_SVC_THUMB; /* CPSR (Enable IRQ and FIQ
interrupts, THUMNB-mode) */
#else //用于设置CPSR寄存器
    taskContext->regCPSR = PSR_MODE_SVC_ARM;   /* CPSR (Enable IRQ and FIQ
interrupts, ARM-mode) */
#endif
#if !defined(LOSCFG_ARCH_FPU_DISABLE)
    /* 0xAAA0000000000000LL : float reg initialed magic word */
    for (UINT32 index = 0; index < FP_REGS_NUM; index++) {
        taskContext->D[index] = 0xAAA0000000000000LL + index; /* D0 - D31 */
```

```
    }
    taskContext->regFPSCR = 0;
    taskContext->regFPEXC = FP_EN;
#endif
    return (VOID *)taskContext;
}
```

任务创建后，内核可以执行锁任务调度，解锁任务调度，挂起，恢复，延时等操作，同时也可以设置任务优先级，获取任务优先级。任务结束的时候，则进行当前任务自删除操作。

Huawei LiteOS 系统中的任务管理模块为用户提供下面几种功能。

接口名	描述
LOS_TaskCreateOnly	创建任务，并使该任务进入suspend状态，并不调度。
LOS_TaskCreate	创建任务，并使该任务进入ready状态，并调度。
LOS_TaskDelete	删除指定的任务。
LOS_TaskResume	恢复挂起的任务。
LOS_TaskSuspend	挂起指定的任务。
LOS_TaskDelay	任务延时等待。
LOS_TaskYield	显式放权，调整指定优先级的任务调度顺序。
LOS_TaskLock	锁任务调度。
LOS_TaskUnlock	解锁任务调度。
LOS_CurTaskPriSet	设置当前任务的优先级。
LOS_TaskPriSet	设置指定任务的优先级。
LOS_TaskPriGet	获取指定任务的优先级。
LOS_CurTaskIDGet	获取当前任务的ID。
LOS_TaskInfoGet	设置指定任务的优先级。
LOS_TaskPriGet	获取指定任务的信息。
LOS_TaskStatusGet	获取指定任务的状态。
LOS_TaskNameGet	获取指定任务的名称。
LOS_TaskInfoMonitor	监控所有任务，获取所有任务的信息。
LOS_NextTaskIDGet	获取即将被调度的任务的ID。

3.4 创建任务的全过程

创建任务之前先了解另一个结构体 tagTskInitParam

```
typedef struct tagTskInitParam {//Task的初始化参数
    TSK_ENTRY_FUNC  pfnTaskEntry;  /**< Task entrance function */ //任务的入口函数
    UINT16          usTaskPrio;    /**< Task priority */ //任务优先级
    UINT16          policy;        /**< Task policy */  //任务调度方式
    UINTPTR         auwArgs[4];    /**< Task parameters,  of which the maximum number is four */ //入口函数的参数，最多四个
    UINT32          uwStackSize;   /**< Task stack size */ //任务栈大小
```

```
    CHAR            *pcName;        /**< Task name */  //任务名称
#if (LOSCFG_KERNEL_SMP == YES)
    UINT16          usCpuAffiMask; /**< Task cpu affinity mask         */ //任务cpu亲和力掩码
#endif
    UINT32          uwResved;      /**< It is automatically deleted if set to LOS_TASK_STATUS_DETACHED。
                                        It is unable to be deleted if set to 0。 */ //如果设置为LOS_TASK_STATUS_DETACHED，则自动删除。如果设置为0，则无法删除
    UINT16          consoleID;     /**< The console id of task belongs  */ //任务的控制台id所属
    UINT32          processID; //进程ID
    UserTaskParam   userParam; //在用户态运行时栈参数
} TSK_INIT_PARAM_S;
```

这些初始化参数是外露的任务初始参数，`pfnTaskEntry 对java来说就是你new进程的run()`，需要上层用户提供。 看个例子吧:shell中敲 ping 命令看下它创建的过程

```
u32_t osShellPing(int argc,  const char **argv)
{

    int ret;
    u32_t i = 0;
    u32_t count = 0;
    int count_set = 0;
    u32_t interval = 1000; /* default ping interval */
    u32_t data_len = 48; /* default data length */
    ip4_addr_t dst_ipaddr;
    TSK_INIT_PARAM_S stPingTask;
    // 。。。省去一些中间代码
    /* start one task if ping forever or ping count greater than 60 */
    if (count == 0 || count > LWIP_SHELL_CMD_PING_RETRY_TIMES) {

        if (ping_taskid > 0) {

            PRINTK("Ping task already running and only support one now\n");
            return LOS_NOK;
        }
        stPingTask.pfnTaskEntry = (TSK_ENTRY_FUNC)ping_cmd;//线程的执行函数
        stPingTask.uwStackSize =
LOSCFG_BASE_CORE_TSK_DEFAULT_STACK_SIZE;//0x4000 = 16K
        stPingTask.pcName = "ping_task";
        stPingTask.usTaskPrio = 8; /* higher than shell 优先级高于10，属于内核态线程*/
        stPingTask.uwResved = LOS_TASK_STATUS_DETACHED;
        stPingTask.auwArgs[0] = dst_ipaddr。addr; /* network order */
        stPingTask.auwArgs[1] = count;
        stPingTask.auwArgs[2] = interval;
```

```
        stPingTask.auwArgs[3] = data_len;
        ret = LOS_TaskCreate((UINT32 *)(&ping_taskid),  &stPingTask);
    }
 // ...
    return LOS_OK;
ping_error:
    lwip_ping_usage();
    return LOS_NOK;
}
```

发现`ping`的调度优先级是`8`，比`shell` 还高，那`shell`的是多少？答案是:看源码是 `9`

```
LITE_OS_SEC_TEXT_MINOR UINT32 ShellTaskInit(ShellCB *shellCB)
{

    CHAR *name = NULL;
    TSK_INIT_PARAM_S initParam = {
   0};
    if (shellCB->consoleID == CONSOLE_SERIAL) {

        name = SERIAL_SHELL_TASK_NAME;
    } else if (shellCB->consoleID == CONSOLE_TELNET) {

        name = TELNET_SHELL_TASK_NAME;
    } else {

        return LOS_NOK;
    }
    initParam.pfnTaskEntry = (TSK_ENTRY_FUNC)ShellTask;
    initParam.usTaskPrio   = 9; /* 9:shell task priority */
    initParam.auwArgs[0]   = (UINTPTR)shellCB;
    initParam.uwStackSize  = 0x3000;
    initParam.pcName       = name;
    initParam.uwResved     = LOS_TASK_STATUS_DETACHED;
    (VOID)LOS_EventInit(&shellCB->shellEvent);
    return LOS_TaskCreate(&shellCB->shellTaskHandle,  &initParam);
}
```

前置条件了解清楚后，具体看任务是如何一步步创建的，如何和进程绑定，加入调度就绪队列，还是继续看源码

```
//创建Task
LITE_OS_SEC_TEXT_INIT UINT32 LOS_TaskCreate(UINT32 *taskID,  TSK_INIT_PARAM_S
*initParam)
{
    UINT32 ret;
    UINT32 intSave;
    LosTaskCB *taskCB = NULL;
```

```
    if (initParam == NULL) {
        return LOS_ERRNO_TSK_PTR_NULL;
    }

    if (OS_INT_ACTIVE) {
        return LOS_ERRNO_TSK_YIELD_IN_INT;
    }

    if (initParam->uwResved & OS_TASK_FLAG_IDLEFLAG) {//OS_TASK_FLAG_IDLEFLAG
是属于内核 idle进程专用的
        initParam->processID = OsGetIdleProcessID();//获取空闲进程
    } else if (OsProcessIsUserMode(OsCurrProcessGet())) {//当前进程是否为用户模
式
        initParam->processID = OsGetKernelInitProcessID();//不是就取"Kernel"进程
    } else {
        initParam->processID = OsCurrProcessGet()->processID;//获取当前进程 ID赋
值
    }
    initParam->uwResved &= ~OS_TASK_FLAG_IDLEFLAG;//不能是
OS_TASK_FLAG_IDLEFLAG
    initParam->uwResved &= ~OS_TASK_FLAG_PTHREAD_JOIN;//不能是
OS_TASK_FLAG_PTHREAD_JOIN
    if (initParam->uwResved & LOS_TASK_STATUS_DETACHED) {//是否设置了自动删除
        initParam->uwResved = OS_TASK_FLAG_DETACHED;//自动删除，注意这里是 = ，也
就是说只有 OS_TASK_FLAG_DETACHED 一个标签了
    }

    ret = LOS_TaskCreateOnly(taskID,  initParam);//创建一个任务，这是任务创建的实
体，前面都只是前期准备工作
    if (ret != LOS_OK) {
        return ret;
    }
    taskCB = OS_TCB_FROM_TID(*taskID);//通过ID拿到task实体

    SCHEDULER_LOCK(intSave);
    taskCB->taskStatus &= ~OS_TASK_STATUS_INIT;//任务不再是初始化
    OS_TASK_SCHED_QUEUE_ENQUEUE(taskCB,  0);//进入调度就绪队列，新任务是直接进入
就绪队列的
    SCHEDULER_UNLOCK(intSave);

    /* in case created task not running on this core,
       schedule or not depends on other schedulers status。 */
    LOS_MpSchedule(OS_MP_CPU_ALL);//如果创建的任务没有在这个内核上运行，是否调度取
决于其他调度程序的状态。
    if (OS_SCHEDULER_ACTIVE) {//当前CPU核处于可调度状态
        LOS_Schedule();//发起调度
    }
```

```
    return LOS_OK;
}
```

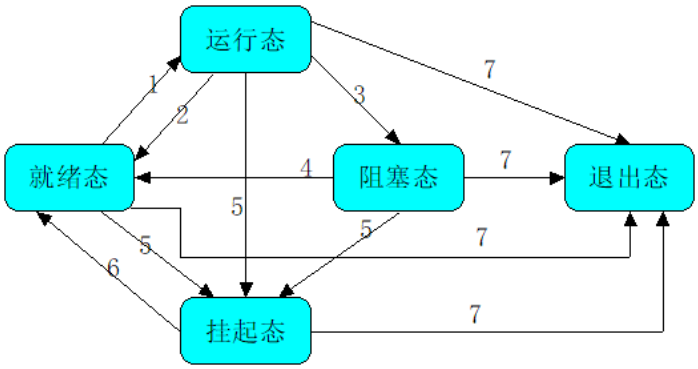
对应张大爷的故事：就是节目单要怎么填，按格式来，从哪里开始演，要多大的空间，王场馆好协调好现场的环境。这里注意 在同一个节目单只要节目没演完，王场馆申请场地的空间就不能给别人用，这个场地空间对应的就是鸿蒙任务的栈空间，除非整个节目单都完了，就回收了。把整个场地干干净净的留给下一个人的节目单来表演。

至此的创建已经完成，已各就各位，源码最后还申请了一次`LOS_Schedule()`;因为鸿蒙的调度方式是抢占式的，如何本次`task`的任务优先级高于其他就绪队列，那么接下来要执行的任务就是它了！

4　任务状态切换

任务状态： Huawei LiteOS系统中的任务有多种运行状态。系统初始化完成后，创建的任务就可以在系统中竞争一定的资源，由内核进行调度。任务状态通常分为以下五种：

- 就绪（Ready）：该任务在就绪队列中，只等待CPU。
- 运行（Running）：该任务正在执行。
- 阻塞（Blocked）：该任务不在就绪队列中。包含任务被挂起（suspend状态）、任务被延时（delay状态）、任务正在等待信号量、读写队列或者等待事件等。
- 挂起（Suspend）：该任务被用户挂起。
- 退出态（Dead）：该任务运行结束，等待系统回收资源。



任务状态迁移说明：

- 1. 就绪态→运行态：任务创建后进入就绪态，发生任务切换时，就绪队列中最高优先级的任务被执行，从而进入运行态，但此刻该任务依旧在就绪队列中。
- 2. 运行态→就绪态：有更高优先级任务创建或者恢复后，会发生任务调度，此刻就绪队列中最高优先级任务变为运行态，那么原先运行的任务由运行态变为就绪态，依然在就绪队列中。
- 3. 运行态→阻塞态：正在运行的任务发生阻塞（延时、读信号量等）时，该任务会从就绪队列中删除，任务状态由运行态变成阻塞态，然后发生任务切换，运行就绪队列中最高优先级任务。

• 4. 阻塞态→就绪态：任务也有可能在就绪态时被阻塞，此时任务状态由就绪态变为阻塞态，该任务从就绪队列中删除，不会参与任务调度，直到该任务被恢复。

• 5 其他状态→挂起态：其他状态的任务调用挂起接口，任务状态由其他状态变为挂起态。

• 6. 挂起态→就绪态：挂起的任务被恢复后，此时被恢复的任务会被加入就绪队列，从而由挂起态变成就绪态；此时如果被恢复任务的优先级高于正在运行任务的优先级，则会发生任务切换，该任务由就绪态变成运行态。

• 7. 其他状态→退出态：运行中的任务运行结束，或调用删除接口，任务状态由其他状态变为退出态。退出态包含任务运行结束的正常退出状态以及Invalid状态。

4.1 就绪态→运行态

LOS_Schedule->OsSchedResched

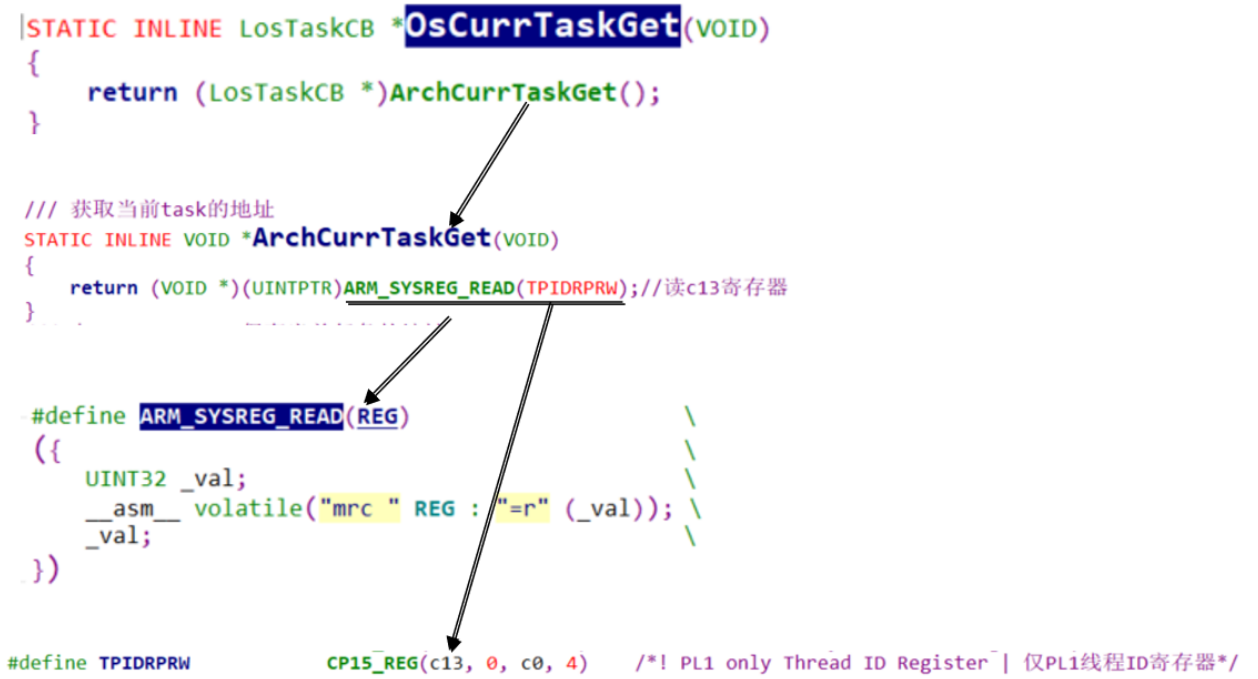
```
VOID OsSchedResched(VOID)
{
    LOS_ASSERT(LOS_SpinHeld(&g_taskSpin));
    SchedRunqueue *rq = OsSchedRunqueue();
#ifdef LOSCFG_KERNEL_SMP
    LOS_ASSERT(rq->taskLockCnt == 1);
#else
    LOS_ASSERT(rq->taskLockCnt == 0);
#endif

    rq->schedFlag &= ~INT_PEND_RESCH;
    LosTaskCB *runTask = OsCurrTaskGet();
    LosTaskCB *newTask = TopTaskGet(rq);
    if (runTask == newTask) {
        return;
    }

    SchedTaskSwitch(rq, runTask, newTask);
}
```

1.获得当前CPU的就绪队列，保存到rq指针中

2.获取当前正在CPU上运行的任务的TCB，保存在runTask指针中

3.获取当前CPU就绪队列中优先级最高的任务TCB，保存在newTask指针中

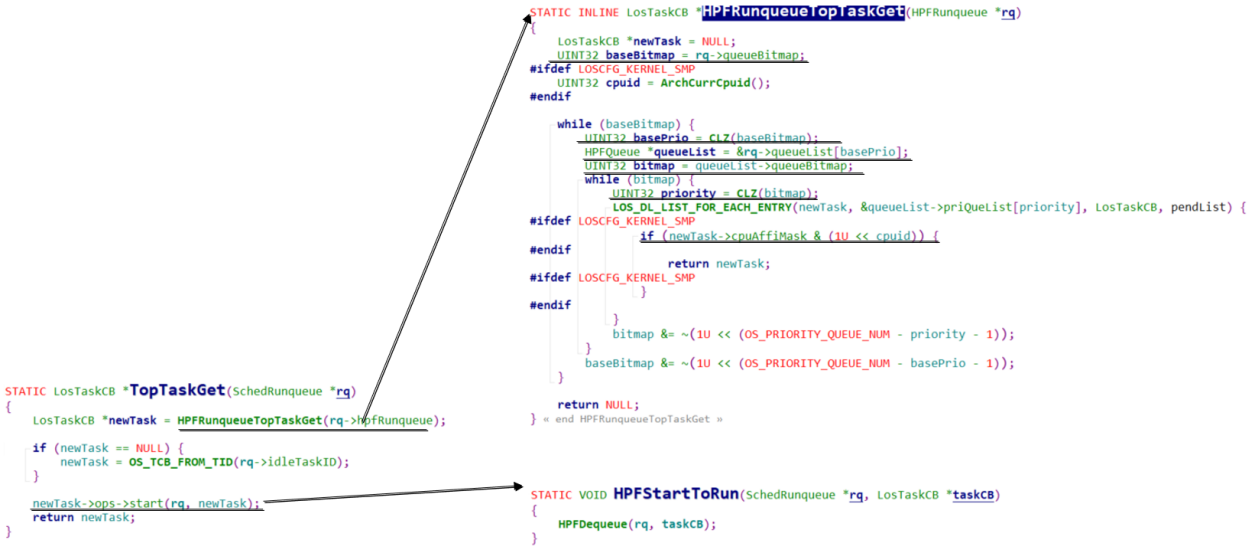
4.调用SchedTaskSwitch完成新旧任务的“上下文切换”。



OsCurrTaskGet的功能是获得正在运行任务的TCB地址。

该函数调用ArchCurrTaskGet函数，从这个函数的名称（Arch开头）可见，是一个与体系结构相关的函数。

通过ARM_SYSREG_READ宏，执行mrc汇编指令，读取c13寄存器，该寄存器中保存着当前运行任务TCB的地址。

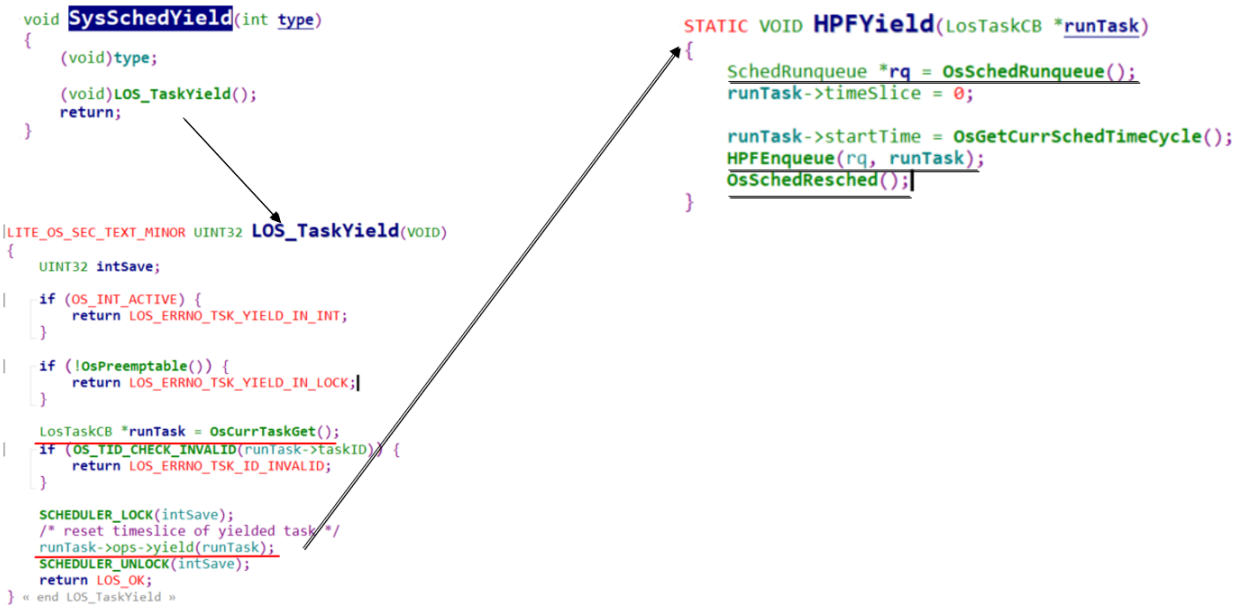


HPFRunqueueTopTaskGet仅负责找到就绪队列中优先级最高的任务。主要是通过HPFRunqueue中的Bitmap，利用CLZ函数进行。

对多核情况，还要检查找到的任务与当前CPU的“亲核性”。

如果找到了新的就绪任务，则调用HPFDequeue函数将该任务从就绪队列中移除。

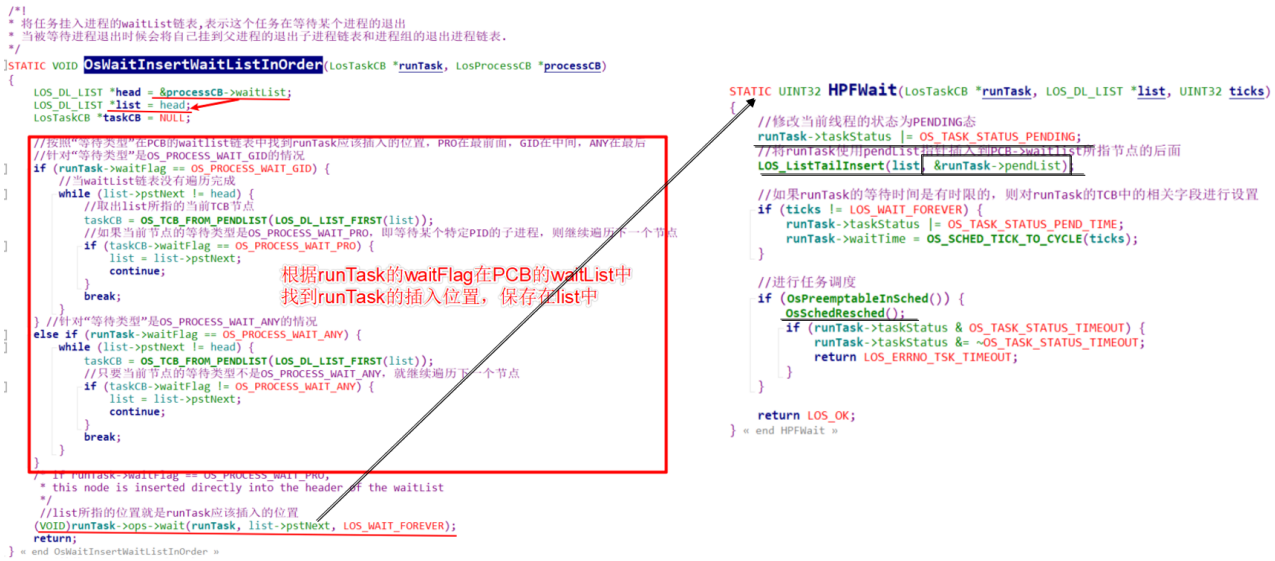
4.2 运行态→就绪态



yield系统调用是在很多操作系统中都存在的一种系统调用，它的作用是让当前正在执行的进程**主动放弃 CPU 的使用权**，让其他进程有机会执行。

4.3 运行态→阻塞态

SysWaitid->LOS_Waitid->OsWait->OsWaitInsertWaitListInOrder



4.4 阻塞态→就绪态

OsProcessNaturalExit->OsWaitCheckAndWakeParentProcess->OsWaitWakeTask

```c
/*! 唤醒等待wakePID结束的任务 */
VOID OsWaitWakeTask(LosTaskCB *taskCB, UINT32 wakePID)
{
    taskCB->waitID = wakePID;
    taskCB->ops->wake(taskCB);
#ifdef LOSCFG_KERNEL_SMP
    LOS_MpSchedule(OS_MP_CPU_ALL);//向所有cpu发送调度指令
#endif
}
```

```c
STATIC VOID HPFWake(LosTaskCB *resumedTask)
{
    LOS_ListDelete(&resumedTask->pendList);
    resumedTask->taskStatus &= ~OS_TASK_STATUS_PENDING;

    if (resumedTask->taskStatus & OS_TASK_STATUS_PEND_TIME) {
        OsSchedTimeoutQueueDelete(resumedTask);
        resumedTask->taskStatus &= ~OS_TASK_STATUS_PEND_TIME;
    }

    if (!(resumedTask->taskStatus & OS_TASK_STATUS_SUSPENDED)) {
#ifdef LOSCFG_SCHED_DEBUG
        resumedTask->schedStat.pendTime += OsGetCurrSchedTimeCycle() - resumedTask->startTime;
        resumedTask->schedStat.pendCount++;
#endif
        HPFEnqueue(OsSchedRunqueue(), resumedTask);
    }
}
```

首先从进程PCB的waitList中拿下被唤醒任务的TCB，然后将其加入就绪队列。

4.5 其他态→挂起态

```c
///外部接口，对OsTaskSuspend的封装
LITE_OS_SEC_TEXT_INIT UINT32 LOS_TaskSuspend(UINT32 taskID)
{
    UINT32 intSave;
    UINT32 errRet;

    if (OS_TID_CHECK_INVALID(taskID)) {
        return LOS_ERRNO_TSK_ID_INVALID;
    }

    LosTaskCB *taskCB = OS_TCB_FROM_TID(taskID);
    if (taskCB->taskStatus & OS_TASK_FLAG_SYSTEM_TASK) {
        return LOS_ERRNO_TSK_OPERATE_SYSTEM_TASK;
    }

    SCHEDULER_LOCK(intSave);
    errRet = OsTaskSuspend(taskCB);
    SCHEDULER_UNLOCK(intSave);
    return errRet;
}
```

```c
LITE_OS_SEC_TEXT STATIC UINT32 OsTaskSuspend(LosTaskCB *taskCB)
{
    UINT32 errRet;
    UINT16 tempStatus = taskCB->taskStatus;
    if (tempStatus & OS_TASK_STATUS_UNUSED) {
        return LOS_ERRNO_TSK_NOT_CREATED;
    }

    if (tempStatus & OS_TASK_STATUS_SUSPENDED) {
        return LOS_ERRNO_TSK_ALREADY_SUSPENDED;
    }

    if ((tempStatus & OS_TASK_STATUS_RUNNING) && //如果参数任务正在运行，注意多Cpu core
        !OsTaskSuspendCheckOnRun(taskCB, &errRet)) {//很有可能是别的CPU core在跑的任务
        return errRet;
    }

    return taskCB->ops->suspend(taskCB);
}
```

```c
STATIC UINT32 HPFSuspend(LosTaskCB *taskCB)
{
    if (taskCB->taskStatus & OS_TASK_STATUS_READY) {
        HPFDequeue(OsSchedRunqueue(), taskCB);
    }

    SchedTaskFreeze(taskCB);

    taskCB->taskStatus |= OS_TASK_STATUS_SUSPENDED;
    OsHookCall(LOS_HOOK_TYPE_MOVEDTASKTOSUSPENDEDLIST, taskCB);
    if (taskCB == OsCurrTaskGet()) {
        OsSchedResched();
    }
    return LOS_OK;
```

4.6 挂起态→就绪态

进行调度的方式有两种：

第一种是显示，即直接调用OsSchedResched函数。

第二种是设置needSched标记为True，在适当的时机调用OsSchedResched函数进行调度。

```c
///恢复挂起的任务，使该任务进入ready状态
LITE_OS_SEC_TEXT_INIT UINT32 LOS_TaskResume(UINT32 taskID)
{
    UINT32 intSave;
    UINT32 errRet;
    BOOL needSched = FALSE;

    if (OS_TID_CHECK_INVALID(taskID)) {
        return LOS_ERRNO_TSK_ID_INVALID;
    }

    LosTaskCB *taskCB = OS_TCB_FROM_TID(taskID);
    SCHEDULER_LOCK(intSave);

    /* clear pending signal */
    taskCB->signal &= ~SIGNAL_SUSPEND;//清楚挂起信号

    if (taskCB->taskStatus & OS_TASK_STATUS_UNUSED) {
        errRet = LOS_ERRNO_TSK_NOT_CREATED;
        OS_GOTO_ERREND();
    } else if (!(taskCB->taskStatus & OS_TASK_STATUS_SUSPENDED)) {
        errRet = LOS_ERRNO_TSK_NOT_SUSPENDED;
        OS_GOTO_ERREND();
    }

    errRet = taskCB->ops->resume(taskCB, &needSched);
    SCHEDULER_UNLOCK(intSave);

    LOS_MpSchedule(OS_MP_CPU_ALL);
    if (OS_SCHEDULER_ACTIVE && needSched) {
        LOS_Schedule();
    }

    return errRet;
```

```c
STATIC UINT32 HPFResume(LosTaskCB *taskCB, BOOL *needSched)
{
    *needSched = FALSE;

    SchedTaskUnfreeze(taskCB);

    taskCB->taskStatus &= ~OS_TASK_STATUS_SUSPENDED;
    if (!OsTaskIsBlocked(taskCB)) {
        HPFEnqueue(OsSchedRunqueue(), taskCB);
        *needSched = TRUE;
    }

    return LOS_OK;
}
```

```c
STATIC INLINE BOOL OsTaskIsBlocked(const LosTaskCB *taskCB)
{
    return ((taskCB->taskStatus & (OS_TASK_STATUS_SUSPENDED | OS_TASK_STATUS_PENDING | OS_TASK_STATUS_DELAY)) != 0);
}
```

4.7 其他态→退出态

回忆“初始化内核栈”中，OsTaskEntry是内核任务的入口函数，当内核任务执行完毕后，将继续后面的执行，即运行OsRunningTaskToExit函数。此时任务由“运行态”->“退出态”。

```
LITE_OS_SEC_TEXT_INIT VOID OsTaskEntry(UINT32 taskID)
{
    LOS_ASSERT(!OS_TID_CHECK_INVALID(taskID));

    /*
     * task scheduler needs to be protected throughout the whole process
     * from interrupt and other cores. release task spinlock and enable
     * interrupt in sequence at the task entry.
     */
    LOS_SpinUnlock(&g_taskSpin);//释放任务自旋锁
    (VOID)LOS_IntUnLock();//恢复中断

    LosTaskCB *taskCB = OS_TCB_FROM_TID(taskID);
    taskCB->joinRetval = taskCB->taskEntry(taskCB->args[0], taskCB->args[1],//调用任务的入口函数
                                           taskCB->args[2], taskCB->args[3]); /* 2 & 3: just for args array index */
    if (!(taskCB->taskStatus & OS_TASK_FLAG_PTHREAD_JOIN)) {
        taskCB->joinRetval = 0;//结合数为0
    }

    OsRunningTaskToExit(taskCB, 0);
} « end OsTaskEntry »
```

```
LITE_OS_SEC_TEXT VOID OsRunningTaskToExit(LosTaskCB *runTask, UINT32 status)
{
    UINT32 intSave;
    //所退出的线程是主线程，则
    if (OsProcessThreadGroupIDGet(runTask) == runTask->taskID) {
        OsProcessThreadGroupDestroy();
    }

    OsHookCall(LOS_HOOK_TYPE_TASK_DELETE, runTask);

    SCHEDULER_LOCK(intSave);
    if (OsProcessThreadNumberGet(runTask) == 1) { /* 1: The last task of the process exits *
        SCHEDULER_UNLOCK(intSave);

        OsTaskResourcesToFree(runTask);
        OsProcessResourcesToFree(OS_PCB_FROM_PID(runTask->processID));

        SCHEDULER_LOCK(intSave);

        OsProcessNaturalExit(OS_PCB_FROM_PID(runTask->processID), status);
        OsTaskReleaseHoldLock(runTask);
        OsTaskStatusUnusedSet(runTask);
    } else if (runTask->taskStatus & OS_TASK_FLAG_PTHREAD_JOIN) {
        OsTaskReleaseHoldLock(runTask);
    } else {
        SCHEDULER_UNLOCK(intSave);

        OsTaskResourcesToFree(runTask);

        SCHEDULER_LOCK(intSave);
        OsInactiveTaskDelete(runTask);
        OsEventWriteUnsafe(&g_resourceEvent, OS_RESOURCE_EVENT_FREE, FALSE, NULL);
    }

    OsSchedResched();
    SCHEDULER_UNLOCK(intSave);
    return;
} « end OsRunningTaskToExit »
```

退出的是主线程，则摧毁整个线程组，进程退出

如果是最后一个退出的线程，则退出进程

针对退出一个普通的没有Join的线程，
同时不是线程组的最后一个线程的情况

```
LITE_OS_SEC_TEXT VOID OsInactiveTaskDelete(LosTaskCB *taskCB)
{
    UINT16 taskStatus = taskCB->taskStatus;

    OsTaskReleaseHoldLock(taskCB);

    taskCB->ops->exit(taskCB);
    if (taskStatus & OS_TASK_STATUS_PENDING) {
        LosMux *mux = (LosMux *)taskCB->taskMux;
        if (LOS_MuxIsValid(mux) == TRUE) {
            OsMuxBitmapRestore(mux, NULL, taskCB);
        }
    }

    OsTaskStatusUnusedSet(taskCB);

    OsDeleteTaskFromProcess(taskCB);

    OsHookCall(LOS_HOOK_TYPE_TASK_DELETE, taskCB);
} « end OsInactiveTaskDelete »
```

```
STATIC VOID HPFExit(LosTaskCB *taskCB)
{
    if (taskCB->taskStatus & OS_TASK_STATUS_READY) {
        HPFDequeue(OsSchedRunqueue(), taskCB);
    } else if (taskCB->taskStatus & OS_TASK_STATUS_PENDING) {
        LOS_ListDelete(&taskCB->pendList);
        taskCB->taskStatus &= ~OS_TASK_STATUS_PENDING;
    }

    if (taskCB->taskStatus & (OS_TASK_STATUS_DELAY | OS_TASK_STATUS_PEND_TIME)) {
        OsSchedTimeoutQueueDelete(taskCB);
        taskCB->taskStatus &= ~(OS_TASK_STATUS_DELAY | OS_TASK_STATUS_PEND_TIME);
    }
}
```

```
VOID OsDeleteTaskFromProcess(LosTaskCB *taskCB)
{
    LosProcessCB *processCB = OS_PCB_FROM_PID(taskCB->processID);

    LOS_ListDelete(&taskCB->threadList);
    processCB->threadNumber--;
    OsTaskInsertToRecycleList(taskCB);
}
```

5 心得体会

通过深入学习 LiteOS 任务管理源码并完成相关实验，我们对任务控制块属性和任务管理函数的较为深刻理解。实验中，我们通过对多个任务执行挂起、阻塞、删除和恢复等操作，深入研究了任务管理状态的有意义变化。这使我们掌握了 LiteOS 任务管理函数的使用，还深化了对实时系统中任务管理和状态切换的认识。