一、实验名称

任务调度

二、实验目的

实现任务调度。
以下是UniProton中任务调度机制：
任务调度是操作系统的核心功能之一。 UniProton实现的是一个单进程支持多线程的操作系统。在UniProton中，一个任务表示一个线程。UniProton中的任务为抢占式调度机制，而非时间片轮转调度方式。高优先级的任务可打断低优先级任务，低优先级任务必须在高优先级任务挂起或阻塞后才能得到调度。

三、实验任务

基础数据结构：双向链表

双向链表结构在 src/include/list_types.h 中定义。

```
#ifndef _LIST_TYPES_H

#define _LIST_TYPES_H

struct TagListObject {

    struct TagListObject *prev;//前指针

    struct TagListObject *next;//后指针

};
#endif  /* end _LIST_TYPES_H */
```

此外，在 src/include/prt_list_external.h 中定义了链表各种相关操作。

```
#ifndef PRT_LIST_EXTERNAL_H
#define PRT_LIST_EXTERNAL_H

// 引入必要的类型定义
#include "prt_typedef.h"
#include "list_types.h"
```

```
// 定义一个初始化链表对象的宏，将其 prev 和 next 都指向自身，表示空链表
#define LIST_OBJECT_INIT(object) { \
        &(object), &(object)       \
    }

// 初始化链表对象的宏，适用于结构体指针形式
#define INIT_LIST_OBJECT(object)   \
    do {                           \
        (object)->next = (object); \
        (object)->prev = (object); \
    } while (0)

// 获取链表的最后一个元素
#define LIST_LAST(object) ((object)->prev)
// 获取链表的第一个元素
#define LIST_FIRST(object) ((object)->next)
// 获取链表的第一个元素（别名）
#define OS_LIST_FIRST(object) ((object)->next)

/*
 * 链表底层插入操作
 * newNode：待插入的新节点
 * prev：前一个节点
 * next：后一个节点
 */
OS_SEC_ALW_INLINE INLINE void ListLowLevelAdd(struct TagListObject *newNode,
struct TagListObject *prev,
                                              struct TagListObject *next)
{
    newNode->next = next;
    newNode->prev = prev;
    next->prev = newNode;
    prev->next = newNode;
}

/*
 * 插入新节点到链表头部（头插法）
 * newNode：待插入的新节点
 * listObject：链表头结点
 */
OS_SEC_ALW_INLINE INLINE void ListAdd(struct TagListObject *newNode, struct
TagListObject *listObject)
{
```

```
    ListLowLevelAdd(newNode, listObject, listObject->next);
}

/*
 * 插入新节点到链表尾部（尾插法）
 * newNode：待插入的新节点
 * listObject：链表头结点
 */
OS_SEC_ALW_INLINE INLINE void ListTailAdd(struct TagListObject *newNode,
struct TagListObject *listObject)
{
    ListLowLevelAdd(newNode, listObject->prev, listObject);
}

/*
 * 链表底层删除操作
 * prevNode：要删除节点的前一个节点
 * nextNode：要删除节点的后一个节点
 */
OS_SEC_ALW_INLINE INLINE void ListLowLevelDelete(struct TagListObject
*prevNode, struct TagListObject *nextNode)
{
    nextNode->prev = prevNode;
    prevNode->next = nextNode;
}

/*
 * 删除指定节点
 * node：要删除的节点
 * 删除后将其前后指针置空
 */
OS_SEC_ALW_INLINE INLINE void ListDelete(struct TagListObject *node)
{
    ListLowLevelDelete(node->prev, node->next);

    node->next = NULL;
    node->prev = NULL;
}

/*
 * 判断链表是否为空
 * 返回 true 表示为空，false 表示非空
 */
```

```
OS_SEC_ALW_INLINE INLINE bool ListEmpty(const struct TagListObject
*listObject)
{
    return (bool)((listObject->next == listObject) && (listObject->prev ==
listObject));
}

// 计算结构体中某个字段的偏移量
#define OFFSET_OF_FIELD(type, field) ((uintptr_t)((uintptr_t)(&((type
*)0x10)->field) - (uintptr_t)0x10))

// 根据结构体中字段的地址计算出结构体的首地址
#define COMPLEX_OF(ptr, type, field) ((type *)((uintptr_t)(ptr) -
OFFSET_OF_FIELD(type, field)))

/*
 * 从结构体成员指针获取其宿主结构体指针
 * ptrOfList：结构体中成员字段的地址
 * typeOfList：结构体类型
 * fieldOfList：字段名
 */
#define LIST_COMPONENT(ptrOfList, typeOfList, fieldOfList)
COMPLEX_OF(ptrOfList, typeOfList, fieldOfList)

/*
 * 遍历链表的宏定义（不安全版本）
 * posOfList：用于遍历的临时变量
 * listObject：链表头节点
 * typeOfList：结构体类型
 * field：结构体中的链表字段
 */
#define LIST_FOR_EACH(posOfList, listObject, typeOfList, field)
\
    for ((posOfList) = LIST_COMPONENT((listObject)->next, typeOfList,
field); &(posOfList)->field != (listObject); \
        (posOfList) = LIST_COMPONENT((posOfList)->field.next, typeOfList,
field))

/*
 * 遍历链表的宏定义（安全版本，支持在遍历过程中删除节点）
 */
#define LIST_FOR_EACH_SAFE(posOfList, listObject, typeOfList, field)
\
```

```
    for ((posOfList) = LIST_COMPONENT((listObject)->next, typeOfList,
field);       \
        (&(posOfList)->field != (listObject))&&((posOfList)->field.next !=
NULL);  \
        (posOfList) = LIST_COMPONENT((posOfList)->field.next, typeOfList,
field))

#endif /* PRT_LIST_EXTERNAL_H */
```

这里面最有意思的是 LIST_COMPONENT 宏，其作用是根据成员地址得到控制块首地址, ptr成员地址, type控制块结构, field成员名。
在内核风格链表中，链表节点（如 `TagListObject`）常作为其他结构体的一个成员字段。这样就可以由链表节点推知其结构体的首地址。

LIST_FOR_EACH 和 LIST_FOR_EACH_SAFE 用于遍历链表，主要是简化代码编写。

任务控制块

任务相关的头文件主要包括 src/include/prt_task.h [下载] 和 src/include/prt_task_external.h [下载]两个头文件。此外还会用到 src/include/prt_module.h [下载] 和 src/include/prt_errno.h [下载] 两个头文件。 prt_module.h中主要是一些模块ID的定义，而 prt_errno.h 主要是错误类型的相关定义，引入这两个头文件主要是为了保持接口与原版 UniProton 相一致。

prt_task.h 中除了一些相关宏定义外，还定义了任务创建时参数传递的结构体： struct TskInitParam。

```
/*

* 任务创建参数的结构体定义。

*

* 传递任务创建时指定的参数信息。

*/

struct TskInitParam {

    /* 任务入口函数 */

    TskEntryFunc taskEntry;

    /* 任务优先级 */
```

```
    TskPrior taskPrio;
    U16 reserved;
    /* 任务参数，最多4个 */
    uintptr_t args[4];
    /* 任务栈的大小 */
    U32 stackSize;
    /* 任务名 */
    char *name;
    /*
    * 本任务的任务栈独立配置起始地址，用户必须对该成员进行初始化，
    * 若配置为0表示从系统内部空间分配，否则用户指定栈起始地址
    */
    uintptr_t stackAddr;
};
```

prt_task_external.h 中定义了任务调度中最重要的数据结构——任务控制块 struct TagTskCb。

```
#define TagOsRunQue TagListObject //简单实现

/*
* 任务线程及进程控制块的结构体统一定义。
*/
struct TagTskCb {
```

```
    /* 当前任务的SP */

    void *stackPointer;

    /* 任务状态,后续内部全改成U32 */

    U32 taskStatus;

    /* 任务的运行优先级 */

    TskPrior priority;

    /* 任务栈配置标记 */

    U16 stackCfgFlg;

    /* 任务栈大小 */

    U32 stackSize;

    TskHandle taskPid;


    /* 任务栈顶 */

    uintptr_t topOfStack;



    /* 任务入口函数 */

    TskEntryFunc taskEntry;

    /* 任务Pend的信号量指针 */

    void *taskSem;



    /* 任务的参数 */

    uintptr_t args[4];
```

```
#if (defined(OS_OPTION_TASK_INFO))
    /* 存放任务名 */
    char name[OS_TSK_NAME_LEN];
#endif
    /* 信号量链表指针 */
    struct TagListObject pendList;
    /* 任务延时链表指针 */
    struct TagListObject timerList;
    /* 持有互斥信号量链表 */
    struct TagListObject semBList;
    /* 记录条件变量的等待线程 */
    struct TagListObject condNode;
#if defined(OS_OPTION_LINUX)
    /* 等待队列指针 */
    struct TagListObject waitList;
#endif
#if defined(OS_OPTION_EVENT)
    /* 任务事件 */
    U32 event;
    /* 任务事件掩码 */
    U32 eventMask;
#endif
```

```
    /* 任务记录的最后一个错误码 */
    U32 lastErr;
    /* 任务恢复的时间点(单位Tick) */
    U64 expirationTick;

#if defined(OS_OPTION_NUTTX_VFS)
    struct filelist tskFileList;
#if defined(CONFIG_FILE_STREAM)
    struct streamlist ta_streamlist;
#endif
#endif
};
```

简单起见，我们还将任务运行队列结构 TagOsRunQue 直接定义为双向链表 TagListObject（见上面代码）。

最后我们引入了 src/include/prt_amp_task_internal.h

```
#ifndef PRT_AMP_TASK_INTERNAL_H
#define PRT_AMP_TASK_INTERNAL_H

#include "prt_task_external.h"
#include "prt_list_external.h"

#define OS_TSK_EN_QUE(runQue, tsk, flags) OsEnqueueTaskAmp((runQue), (tsk))
```

```
#define OS_TSK_EN_QUE_HEAD(runQue, tsk, flags)
OsEnqueueTaskHeadAmp((runQue), (tsk))

#define OS_TSK_DE_QUE(runQue, tsk, flags) OsDequeueTaskAmp((runQue), (tsk))


extern U32 OsTskAMPInit(void);

extern U32 OsIdleTskAMPCreate(void);


OS_SEC_ALW_INLINE INLINE void OsEnqueueTaskAmp(struct TagOsRunQue *runQue,
struct TagTskCb *tsk)

{

    ListTailAdd(&tsk->pendList, runQue);

    return;

}

OS_SEC_ALW_INLINE INLINE void OsEnqueueTaskHeadAmp(struct TagOsRunQue
*runQue, struct TagTskCb *tsk)

{

    ListAdd(&tsk->pendList, runQue);

    return;

}

    OS_SEC_ALW_INLINE INLINE void OsDequeueTaskAmp(struct TagOsRunQue
*runQue, struct TagTskCb *tsk)

{

    ListDelete(&tsk->pendList);

    return;
```

```
}

#endif /* PRT_AMP_TASK_INTERNAL_H */
```

该头文件中主要是定义了三个内联函数，用于将任务控制块加入运行队列或从运行队列中移除任务控制块。

任务创建

任务创建代码主要在 src/kernel/task/prt_task_init.c 中。 代码比较多，我们分几个部分分别介绍。

相关变量与函数声明

首先是引入必要的头文件。

然后声明了 1 个全局双向链表，并通过 LIST_OBJECT_INIT 宏进行初始化。
g_tskCbFreeList 链表是空闲的任务控制块链表。

最后声明了3个外部函数。

```
#include "list_types.h"

#include "os_attr_armv8_external.h"

#include "prt_list_external.h"

#include "prt_task.h"

#include "prt_task_external.h"

#include "prt_asm_cpu_external.h"

#include "os_cpu_armv8_external.h"

#include "prt_config.h"




/* Unused TCBs and ECBs that can be allocated. */

OS_SEC_DATA struct TagListObject g_tskCbFreeList =
LIST_OBJECT_INIT(g_tskCbFreeList);
```

```
extern U32 OsTskAMPInit(void);//初始化 AMP 模式下的任务管理（Task）模块。

extern U32 OsIdleTskAMPCreate(void);//为每个处理器核心创建 AMP 模式下的空闲任务（Idle Task）

extern void OsFirstTimeSwitch(void);//执行首次上下文切换（Context Switch），也就是从系统初始化阶段切换到第一个正式运行的任务。
```

其中头文件 src/include/prt_asm_cpu_external.h [下载] 包含内核相关的一些状态定义。

```
/*
 * Copyright (c) 2009-2022 Huawei Technologies Co., Ltd. All rights reserved.
 *
 * UniProton is licensed under Mulan PSL v2.
 * You can use this software according to the terms and conditions of the Mulan PSL v2.
 * You may obtain a copy of Mulan PSL v2 at:
 *          http://license.coscl.org.cn/MulanPSL2
 * THIS SOFTWARE IS PROVIDED ON AN "AS IS" BASIS, WITHOUT WARRANTIES OF ANY KIND,
 * EITHER EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO NON-INFRINGEMENT,
 * MERCHANTABILITY OR FIT FOR A PARTICULAR PURPOSE.
 * See the Mulan PSL v2 for more details.
 * Create: 2009-10-05
 * Description: 核架构相关的汇编宏头文件
 */
#ifndef PRT_ASM_CPU_EXTERNAL_H
#define PRT_ASM_CPU_EXTERNAL_H

/* 有名空间不支持时，从核不能从有名空间获取 */
#define GLB_NS_ADDR     0
#define GLB_NSM_START   0

#define OS_EXC_REGINFO_OFFSET 160
#define OS_EXC_MAGIC_WORD 0xEDCAACDC
#define OS_SYS_STACK_MAGIC_WORD 0xCACACACA /* 系统栈魔术字 */

/* 内核状态定义 */
#define OS_FLG_HWI_ACTIVE 0x0001
```

```
#define OS_FLG_BGD_ACTIVE 0x0002
#define OS_FLG_TICK_ACTIVE 0x0008
#define OS_FLG_SYS_ACTIVE 0x0010
#define OS_FLG_EXC_ACTIVE 0x0020
#define OS_FLG_TSK_REQ 0x1000
#define OS_FLG_TSK_SWHK 0x2000 /* 任务切换时是否调用切换入口函数 */

#endif /* PRT_ASM_CPU_EXTERNAL_H */
```

极简内存空间管理

内核运行过程中需要动态分配内存。我们实现了一种极简的内存管理，该内存管理方法仅支持4K大小，最多256字节对齐空间的分配。

```
//简单实现OsMemAllocAlign

/*

* 描述：分配任务栈空间

* 仅支持4K大小，最多256字节对齐空间的分配

*/

uint8_t stackMem[20][4096] __attribute__((aligned(256))); // 256 字节对齐，20 个 4K 大小的空间

uint8_t stackMemUsed[20] = {0}; // 记录对应 4K 空间是否已被分配

OS_SEC_TEXT void *OsMemAllocAlign(U32 mid, U8 ptNo, U32 size, U8 alignPow)

{

    // 最多支持256字节对齐

    if (alignPow > 8)

        return NULL;

    if (size != 4096)

        return NULL;
```

```
    for(int i = 0; i < 20; i++){
        if (stackMemUsed[i] == 0){
            stackMemUsed[i] = 1; // 记录对应 4K 空间已被分配
            return &(stackMem[i][0]); // 返回 4K 空间起始地址
        }
    }
    return NULL;
}

/*
* 描述：分配任务栈空间
*/
OS_SEC_L4_TEXT void *OsTskMemAlloc(U32 size)
{
    void *stackAddr = NULL;
        stackAddr = OsMemAllocAlign((U32)OS_MID_TSK, (U8)0, size,
                                    /* 内存已按16字节大小对齐 */
                                    OS_TSK_STACK_SIZE_ALLOC_ALIGN);
    return stackAddr;
}
```

任务栈初始化

在理论课程中，我们知道当发生任务切换时会首先保存前一个任务的上下文到栈里，然后从栈中恢复下一个将运行任务的上下文。可是当任务第一次运行的时候怎么恢复上下文，之前从来没有保存过上下文?

答案就是我们手工制造一个就可以了。下面代码中 stack->x01 到 stack->x29 被初始化成很有标志性意义的值，其他他们的值不重要。比较重要的是 stack->x30 和 stack->spsr 等处的值。

struct TskContext 表示任务上下文，放在 src/bsp/os_cpu_armv8.h 中定义。在我们的实现上它与中断上下文 struct ExcRegInfo (在 src/bsp/os_exc_armv8.h 中定义)没有区别。在UniProton中，它们的定义有一些差别。

```
/*

* 描述: 初始化任务栈的上下文

*/

void *OsTskContextInit(U32 taskID, U32 stackSize, uintptr_t *topStack,
uintptr_t funcTskEntry)

{

    (void)taskID;

    struct TskContext *stack = (struct TskContext *)((uintptr_t)topStack +
stackSize);



    stack -= 1; // 指针减，减去一个TskContext大小



    stack->x00 = 0;

    stack->x01 = 0x01010101;

    stack->x02 = 0x02020202;

    stack->x03 = 0x03030303;

    stack->x04 = 0x04040404;
```

```
    stack->x05 = 0x05050505;

    stack->x06 = 0x06060606;

    stack->x07 = 0x07070707;

    stack->x08 = 0x08080808;

    stack->x09 = 0x09090909;

    stack->x10 = 0x10101010;

    stack->x11 = 0x11111111;

    stack->x12 = 0x12121212;

    stack->x13 = 0x13131313;

    stack->x14 = 0x14141414;

    stack->x15 = 0x15151515;

    stack->x16 = 0x16161616;

    stack->x17 = 0x17171717;

    stack->x18 = 0x18181818;

    stack->x19 = 0x19191919;

    stack->x20 = 0x20202020;

    stack->x21 = 0x21212121;

    stack->x22 = 0x22222222;

    stack->x23 = 0x23232323;

    stack->x24 = 0x24242424;

    stack->x25 = 0x25252525;

    stack->x26 = 0x26262626;
```

```
    stack->x27 = 0x27272727;

    stack->x28 = 0x28282828;

    stack->x29 = 0x29292929;

    stack->x30 = funcTskEntry;   // x30:  lr(link register)设置返回地址为
`funcTskEntry`（就是 `OsTskEntry`）

    stack->xzr = 0;



    stack->elr = funcTskEntry;//当从异常返回时，会跳转到这个地址执行任务

    stack->esr = 0;

    stack->far = 0;

    stack->spsr = 0x305;    // EL1_SP1 | D | A | I | F

    return stack;

}
```

字段	对应寄存器	用途
x00 ~ x28	通用寄存器	保存参数、局部变量等
x29	FP（帧指针）	调用栈追踪
x30	LR（链接寄存器）	异常/函数返回地址（此处设为 `funcTskEntry`）
elr	ELR_EL1	异常返回地址（也设为 `funcTskEntry`）
spsr	SPSR_EL1	状态寄存器，控制返回后 CPU 的状态
xzr	XZR/zero reg	通常不保存
esr、far	异常相关	初始化为 0，用不到

stack->spsr = 0x305 补充说明

DAIF = 1: 禁用 Debug、SError、IRQ、FIQ 中断。初始禁用中断，防止调度前任务中断触发。

M[3:0] = 0101 = EL1h (Exception Level 1, handler stack)，任务将以 EL1 模式（内核模式）执行。

`OsTskContextInit` 相当于 **手工构造出一套 CPU 在中断返回时要恢复的上下文，骗过 CPU 以为它“中断返回后要执行 OsTskEntry”，而实际上是任务的开始。**

在 src/bsp/os_cpu_armv8.h 中加入 struct TskContext 定义。

```
/*

* 任务上下文的结构体定义。

*/

struct TskContext {

    /* *< 当前物理寄存器R0-R12 */

    uintptr_t elr;                // 返回地址

    uintptr_t spsr;

    uintptr_t far;

    uintptr_t esr;

    uintptr_t xzr;

    uintptr_t x30;

    uintptr_t x29;

    uintptr_t x28;

    uintptr_t x27;

    uintptr_t x26;

    uintptr_t x25;

    uintptr_t x24;

    uintptr_t x23;
```

```
    uintptr_t x22;

    uintptr_t x21;

    uintptr_t x20;

    uintptr_t x19;

    uintptr_t x18;

    uintptr_t x17;

    uintptr_t x16;

    uintptr_t x15;

    uintptr_t x14;

    uintptr_t x13;

    uintptr_t x12;

    uintptr_t x11;

    uintptr_t x10;

    uintptr_t x09;

    uintptr_t x08;

    uintptr_t x07;

    uintptr_t x06;

    uintptr_t x05;

    uintptr_t x04;

    uintptr_t x03;

    uintptr_t x02;
```

```
    uintptr_t x01;

    uintptr_t x00;

};
```

```
In file included from /home/xiaoye/OSlab/lab6/src/bsp/prt_exc.c:2:
/home/xiaoye/OSlab/lab6/src/bsp/os_cpu_armv8.h:6:5: error: unknown type name 'uintptr_t'
    6 |     uintptr_t elr;              // ???
      |     ^~~~~~~~~
/home/xiaoye/OSlab/lab6/src/bsp/os_cpu_armv8.h:7:5: error: unknown type name 'uintptr_t'
    7 |     uintptr_t spsr;
      |     ^~~~~~~~~
```

在原代码的基础上，添加**include <stdint.h>**，`uintptr_t` 是 C 标准中定义的无符号整型类型，用来存储指针值的整数表示，需要依赖这个头文件。

任务入口函数

这个函数有几个有趣的地方。（1）你找不到类似 OsTskEntry(taskId); 这样的对 OsTskEntry 的函数调用。这实际上是在通过 OsTskContextInit 函数进行栈初始化时传入的，也就意味着当任务第一次就绪运行时会进入 OsTskEntry 执行。（2）用户指定的 taskcb->taskEntry 不一定要求是 4 参数的，可以是 0~4 参数之间任意选定，这个需要你在汇编层面去理解。

采用 OsTskEntry 的好处是在用户提供的 taskCb->taskEntry 函数的基础上进行了一层封装，比如可以确保调用taskCb->taskEntry执行完后调用 OsTaskExit。

```
/*

* 描述：所有任务入口

*/

OS_SEC_L4_TEXT void OsTskEntry(TskHandle taskId)

{

    struct TagTskCb *taskCb;

    uintptr_t intSave;



    (void)taskId;
```

```
    taskCb = RUNNING_TASK;



    taskCb->taskEntry(taskCb->args[OS_TSK_PARA_0], taskCb-
>args[OS_TSK_PARA_1], taskCb->args[OS_TSK_PARA_2],
                      taskCb->args[OS_TSK_PARA_3]);



    // 调度结束后会开中断，所以不需要自己添加开中断
    intSave = OsIntLock();



    OS_TASK_LOCK_DATA = 0;



    /* PRT_TaskDelete不能关中断操作，否则可能会导致它核发SGI等待本核响应时死等 */
    OsIntRestore(intSave);



    OsTaskExit(taskCb);
}
```

补充说明

任务函数本身 `taskCb->taskEntry` 并不是直接调度执行的，而是被包裹在 `OsTskEntry` 中执行，形成统一入口。
在汇编实现的任务上下文初始化中，任务的 0～3 号参数-写入AArch64 架构中的 X0~X3。 这意味着 taskEntry 无论是 0 参数、1 参数或 4 参数的函数调用形式，系统都可以通过调用规范正确传参。

创建任务

创建任务的代码看上去还是比较多，但已经不是很复杂了。我们从后面的代码往前面看，首先是接口函数 PRT_TaskCreate 函数根据传入的 initParam 参数创建任务返回任务句柄 taskPid。

PRT_TaskCreate 函数会直接调用 OsTaskCreateOnly 函数实际进行任务创建。
OsTaskCreateOnly 函数将：

- 通过 OsTaskCreateChkAndGetTcb 函数从空闲链表 g_tskCbFreeList 中取一个任务控制块；
- 在 OsTaskCreateRsrcInit 函数中，如果用户未提供堆栈空间，则通过 OsTskMemAlloc 为新建的任务分配堆栈空间；
- OsTskContextInit 函数负责将栈初始化成刚刚发生过中断一样；
- OsTskCreateTcbInit 函数负责用 initParam 参数等初始化任务控制块，包括栈指针、入口函数、优先级和参数等；
- 最后将任务的状态设置为挂起 Suspend 状态。这意味着 PRT_TaskCreate 创建任务后处于 Suspend 状态，而不是就绪状态。

```
// 获取一个空闲任务控制块，如果空闲链表为空则返回错误
OS_SEC_ALW_INLINE INLINE U32 OsTaskCreateChkAndGetTcb(struct TagTskCb
**taskCb)
{
    if (ListEmpty(&g_tskCbFreeList)) { // 判断空闲任务控制块链表是否为空
        return OS_ERRNO_TSK_TCB_UNAVAILABLE; // 如果为空，返回错误码
    }

    *taskCb = GET_TCB_PEND(OS_LIST_FIRST(&g_tskCbFreeList)); // 从链表中获取第
一个空闲任务控制块
    ListDelete(OS_LIST_FIRST(&g_tskCbFreeList)); // 将该任务控制块从空闲链表中移
除

    return OS_OK; // 返回成功
}

// 检查地址加法是否发生溢出，防止内存访问越界
OS_SEC_ALW_INLINE INLINE bool OsCheckAddrOffsetOverflow(uintptr_t base,
size_t size)
{
    return (base + size) < base; // 如果加法结果小于原地址，说明发生了溢出
}

// 初始化任务资源，如栈空间分配等
OS_SEC_L4_TEXT U32 OsTaskCreateRsrcInit(U32 taskId, struct TskInitParam
*initParam, struct TagTskCb *taskCb,
```

```
                                                    uintptr_t **topStackOut,
uintptr_t *curStackSize)
{
    U32 ret = OS_OK;
    uintptr_t *topStack = NULL;

    if (initParam->stackAddr != 0) { // 用户传入了栈地址
        topStack = (void *)(initParam->stackAddr); // 使用用户提供的栈
        taskCb->stackCfgFlg = OS_TSK_STACK_CFG_BY_USER; // 标记为用户配置
    } else { // 用户未传入，系统自动分配
        topStack = OsTskMemAlloc(initParam->stackSize); // 动态分配栈空间
        if (topStack == NULL) {
            ret = OS_ERRNO_TSK_NO_MEMORY; // 分配失败，返回错误
        } else {
            taskCb->stackCfgFlg = OS_TSK_STACK_CFG_BY_SYS; // 成功则标记为系统
配置
        }
    }

    *curStackSize = initParam->stackSize; // 记录当前栈大小

    if (ret != OS_OK) {
        return ret; // 如果前面分配失败，直接返回
    }

    *topStackOut = topStack; // 输出栈顶指针
    return OS_OK; // 返回成功
}

// 初始化任务控制块（TCB）的字段
OS_SEC_L4_TEXT void OsTskCreateTcbInit(uintptr_t stackPtr, struct
TskInitParam *initParam,
    uintptr_t topStackAddr, uintptr_t curStackSize, struct TagTskCb *taskCb)
{
    taskCb->stackPointer = (void *)stackPtr; // 设置栈指针
    taskCb->args[OS_TSK_PARA_0] = (uintptr_t)initParam->args[OS_TSK_PARA_0];
// 初始化任务参数
    taskCb->args[OS_TSK_PARA_1] = (uintptr_t)initParam->args[OS_TSK_PARA_1];
    taskCb->args[OS_TSK_PARA_2] = (uintptr_t)initParam->args[OS_TSK_PARA_2];
    taskCb->args[OS_TSK_PARA_3] = (uintptr_t)initParam->args[OS_TSK_PARA_3];
    taskCb->topOfStack = topStackAddr; // 栈顶地址
    taskCb->stackSize = curStackSize; // 栈大小
    taskCb->taskSem = NULL; // 初始化信号量为空
```

```
    taskCb->priority = initParam->taskPrio; // 设置任务优先级
    taskCb->taskEntry = initParam->taskEntry; // 设置任务入口函数

#if defined(OS_OPTION_EVENT)
    taskCb->event = 0;
    taskCb->eventMask = 0;
#endif

    taskCb->lastErr = 0; // 上一次错误清零

    INIT_LIST_OBJECT(&taskCb->semBList); // 初始化链表结构
    INIT_LIST_OBJECT(&taskCb->pendList);
    INIT_LIST_OBJECT(&taskCb->timerList);

    return;
}

// 描述：仅创建任务但不激活，主要用于静态场景
OS_SEC_L4_TEXT U32 OsTaskCreateOnly(TskHandle *taskPid, struct TskInitParam
*initParam)
{
    U32 ret;
    U32 taskId;
    uintptr_t intSave;
    uintptr_t *topStack = NULL;
    void *stackPtr = NULL;
    struct TagTskCb *taskCb = NULL;
    uintptr_t curStackSize = 0;

    intSave = OsIntLock(); // 关中断，进入临界区

    ret = OsTaskCreateChkAndGetTcb(&taskCb); // 获取一个空闲TCB
    if (ret != OS_OK) {
        OsIntRestore(intSave); // 出错恢复中断
        return ret;
    }

    taskId = taskCb->taskPid; // 获取任务ID

    ret = OsTaskCreateRsrcInit(taskId, initParam, taskCb, &topStack,
&curStackSize); // 初始化任务资源
    if (ret != OS_OK) {
        ListAdd(&taskCb->pendList, &g_tskCbFreeList); // 分配失败，TCB重新加入
```

```
空闲链表
        OsIntRestore(intSave); // 恢复中断
        return ret;
    }

    stackPtr = OsTskContextInit(taskId, curStackSize, topStack,
(uintptr_t)OsTskEntry); // 栈初始化，构造上下文

    OsTskCreateTcbInit((uintptr_t)stackPtr, initParam, (uintptr_t)topStack,
curStackSize, taskCb); // 初始化TCB

    taskCb->taskStatus = OS_TSK_SUSPEND | OS_TSK_INUSE; // 设置任务状态为挂起
+已使用

    *taskPid = taskId; // 输出任务ID

    OsIntRestore(intSave); // 恢复中断

    return OS_OK; // 返回成功
}

// 对外接口：创建一个任务但不进行激活
OS_SEC_L4_TEXT U32 PRT_TaskCreate(TskHandle *taskPid, struct TskInitParam
*initParam)
{
    return OsTaskCreateOnly(taskPid, initParam); // 调用内部实现
}
```

解挂任务

PRT_TaskResume 函数负责解挂任务，即将 Suspend 状态的任务转换到就绪状态。PRT_TaskResume 首先检查当前任务是否已创建且处于 Suspend 状态，如果处于 Suspend 状态，则清除 Suspend 位，然后调用 OsMoveTaskToReady 将任务控制块移到就绪队列中。

OsMoveTaskToReady 函数将任务加入就绪队列 g_runQueue，然后通过 OsTskSchedule 进行任务调度和切换（稍后描述）。 由于有新的任务就绪，所以需要通过OsTskSchedule 进行调度。这个位置一般称为调度点。对于优先级调度来说，找到所有的调度点并进行调度非常重要。

```
/*
* 描述：将一个处于挂起状态的任务恢复（解挂）到就绪状态
*/
```

```
OS_SEC_L2_TEXT U32 PRT_TaskResume(TskHandle taskPid)
{
    uintptr_t intSave; // 用于保存中断标志
    struct TagTskCb *taskCb = NULL; // 任务控制块指针

    // 获取 taskPid 对应的任务控制块（TCB）
    taskCb = GET_TCB_HANDLE(taskPid);

    intSave = OsIntLock(); // 关中断，进入临界区

    if (TSK_IS_UNUSED(taskCb)) { // 如果任务控制块处于未使用状态（尚未创建）
        OsIntRestore(intSave); // 恢复中断
        return OS_ERRNO_TSK_NOT_CREATED; // 返回任务未创建错误码
    }

    // 若任务正在运行，且系统任务锁不为 0（表示禁止任务切换）
    if (((OS_TSK_RUNNING & taskCb->taskStatus) != 0) && (g_uniTaskLock !=
0)) {
        OsIntRestore(intSave); // 恢复中断
        return OS_ERRNO_TSK_ACTIVE_FAILED; // 返回任务激活失败错误码
    }

    /*
     * 如果任务既没有被挂起（SUSPEND），也不在可中断的延时中
（DELAY_INTERRUPTIBLE），
     * 则认为任务没有处于挂起状态，不能 resume。
     */
    if (((OS_TSK_SUSPEND | OS_TSK_DELAY_INTERRUPTIBLE) & taskCb->taskStatus)
== 0) {
        OsIntRestore(intSave); // 恢复中断
        return OS_ERRNO_TSK_NOT_SUSPENDED; // 返回任务未挂起错误码
    }

    // 清除挂起状态标志（SUSPEND）
    TSK_STATUS_CLEAR(taskCb, OS_TSK_SUSPEND);

    // 如果任务不是处于阻塞状态，则将其移动到就绪队列中
    OsMoveTaskToReady(taskCb);

    OsIntRestore(intSave); // 恢复中断

    return OS_OK; // 返回成功
```

```
}
```

任务管理系统初始化与启动

OsTskInit 函数通过调用 OsTskAMPInit 函数完成任务管理系统的初始化。主要包括：

> - 为任务控制块分配空间，由于我们只实现了简单的内存分配算法，所以支持的任务控制块数目为：4096 / sizeof(struct TagTskCb) - 2; 减去2是因为预留了 1 个空闲任务，1 个无效任务。
> - 将所有分配的任务控制块加入空闲任务控制块链表 g_tskCbFreeList， 并对所有控制块进行初始化。
> - 任务就绪链表 g_runQueue 通过 INIT_LIST_OBJECT 初始化为空。
> - RUNNING_TASK 目前指向无效任务。

OsActivate 启动多任务系统。

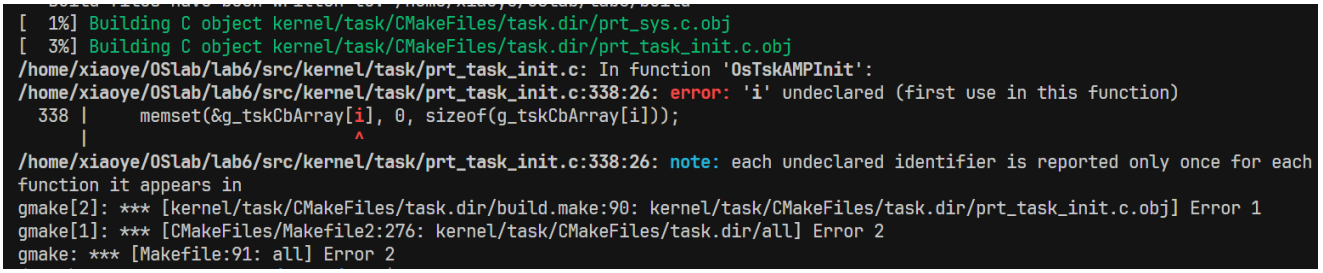
> - 首先通过 OsIdleTskAMPCreate 函数创建空闲任务，这样当系统中没有其他任务就绪时就可以执行空闲任务了。
> - OsTskHighestSet 函数在就绪队列中查找最高优先级任务并将 g_highestTask 指针指向该任务。
> - UNI_FLAG 设置好内核状态
> - OsFirstTimeSwitch 函数将会加载 g_highestTask 的上下文后执行（稍后描述）。

```
/*
* 描述：AMP（对称多处理）任务初始化
*/
extern U32 g_threadNum;
extern void *OsMemAllocAlign(U32 mid, U8 ptNo, U32 size, U8 alignPow);

OS_SEC_L4_TEXT U32 OsTskAMPInit(void)
{
    uintptr_t size;
    U32 idx;

    // 简单处理，分配 4096 字节，用于存储 OS_MAX_TCB_NUM 个任务控制块（TCB）
    // #define OS_MAX_TCB_NUM  (g_tskMaxNum + 1 + 1)  // 1 个 IDLE 任务 + 1
个无效任务
    g_tskCbArray = (struct TagTskCb *)OsMemAllocAlign((U32)OS_MID_TSK, 0,
4096, OS_TSK_STACK_SIZE_ALLOC_ALIGN);
    if (g_tskCbArray == NULL) {
        return OS_ERRNO_TSK_NO_MEMORY; // 分配失败返回内存不足
    }
```

```
    // 根据分配大小计算最大任务数量，减去 2 个特殊任务（Idle + 无效）
    g_tskMaxNum = 4096 / sizeof(struct TagTskCb) - 2;

    // threadNum 增加任务总数（含Idle任务）
    g_threadNum += (g_tskMaxNum + 1);

    // 将所有TCB初始化为0
    for (int i = 0; i < OS_MAX_TCB_NUM - 1; i++)
        g_tskCbArray[i] = (struct TagTskCb){0}; // C99结构体清零语法

    g_tskBaseId = 0; // 基础任务ID为0

    // 初始化空闲任务链表 g_tskCbFreeList，并将所有TCB添加进去
    INIT_LIST_OBJECT(&g_tskCbFreeList);
    for (idx = 0; idx < OS_MAX_TCB_NUM - 1; idx++) {
        g_tskCbArray[idx].taskStatus = OS_TSK_UNUSED; // 初始状态为未使用
        g_tskCbArray[idx].taskPid = (idx + g_tskBaseId); // 设置任务ID
        ListTailAdd(&g_tskCbArray[idx].pendList, &g_tskCbFreeList); // 加入空
闲列表
    }

    /* 设置运行中的任务为一个合法的“僵尸”任务，防止Trace系统访问非法指针 */
    RUNNING_TASK = OS_PST_ZOMBIE_TASK;
    RUNNING_TASK->taskPid = idx + g_tskBaseId;

    // 初始化运行队列
    INIT_LIST_OBJECT(&g_runQueue);

    // 设置任务初始状态为正在使用+正在运行
    RUNNING_TASK->taskStatus = (OS_TSK_INUSE | OS_TSK_RUNNING);
    RUNNING_TASK->priority = OS_TSK_PRIORITY_LOWEST + 1; // 设置最低优先级+1

    return OS_OK;
}

/*
* 描述：任务初始化（调用 AMP 初始化）
*/
OS_SEC_L4_TEXT U32 OsTskInit(void)
{
    U32 ret;
    ret = OsTskAMPInit(); // 调用AMP版本初始化
```

```
    if (ret != OS_OK) {
        return ret; // 初始化失败
    }

    return OS_OK; // 成功
}

/*
* 描述：Idle 背景任务（空循环任务）
*/
OS_SEC_TEXT void OsTskIdleBgd(void)
{
    while (TRUE); // 死循环，不做任何处理
}

/*
* 描述：创建 Idle 背景任务（必须存在才能启用调度）
*/
OS_SEC_L4_TEXT U32 OsIdleTskAMPCreate(void)
{
    U32 ret;
    TskHandle taskHdl;
    struct TskInitParam taskInitParam = {0};
    char tskName[OS_TSK_NAME_LEN] = "IdleTask";

    // 设置任务入口为 Idle 任务函数
    taskInitParam.taskEntry = (TskEntryFunc)OsTskIdleBgd;
    taskInitParam.stackSize = 4096; // 分配 4KB 栈空间
    // taskInitParam.name = tskName; // 名称可选，注释掉
    taskInitParam.taskPrio = OS_TSK_PRIORITY_LOWEST; // 设置最低优先级
    taskInitParam.stackAddr = 0; // 使用系统自动分配栈地址

    // 创建任务，但不会立即激活
    ret = PRT_TaskCreate(&taskHdl, &taskInitParam);
    if (ret != OS_OK) {
        return ret; // 创建失败
    }

    // 激活 Idle 任务
    ret = PRT_TaskResume(taskHdl);
    if (ret != OS_OK) {
        return ret; // 激活失败
    }
```

```
    IDLE_TASK_ID = taskHdl; // 记录 Idle 任务ID

    return ret;
}

/*
* 描述：启动任务管理（激活调度）
*/
OS_SEC_L4_TEXT U32 OsActivate(void)
{
    U32 ret;

    // 创建并激活 Idle 任务
    ret = OsIdleTskAMPCreate();
    if (ret != OS_OK) {
        return ret; // Idle 创建失败
    }

    OsTskHighestSet(); // 设置当前最高优先级任务

    // 设置系统标志，表示后台任务已激活并且有任务请求
    UNI_FLAG |= OS_FLG_BGD_ACTIVE | OS_FLG_TSK_REQ;

    // 启动多任务调度（第一次任务切换）
    OsFirstTimeSwitch();

    // 如果程序继续运行到这里说明启动失败（理论上不会执行到此）
    return OS_ERRNO_TSK_ACTIVE_FAILED;
}
```



结构体赋值语法错误 `{0}`。改为：

```
for(int i = 0; i < OS_MAX_TCB_NUM - 1; i++)

        memset(&g_tskCbArray[i], 0, sizeof(g_tskCbArray[i]));
```

加一个头文件：

```
#include <string.h>
```

在 prt_config.h 中加入空闲任务优先级定义。

```
#define OS_TSK_PRIORITY_LOWEST 63
```

任务状态转换

在 src/kernel/task/prt_task.c 中，

> - 声明了运行队列 g_runQueue， 注意我们之前已经将其定义为双向队列。
> - 提供了将任务添加到就绪队列的 OsTskReadyAdd 函数和从就绪队列中移除就绪队列的 OsTskReadyDel 函数。
>   - OsTskReadyAdd 会设置任务为就绪态
>   - OsTskReadyDel 会清除任务的就绪态
> - 提供了任务结束退出 OsTaskExit 函数，注意 OsTskEntry 中会调用 OsTaskExit 函数。由于任务退出，因此需要进行调度，即存在调度点，所以调用 OsTskSchedule 函数。

```
#include "prt_task_external.h"
#include "prt_typedef.h"
#include "os_attr_armv8_external.h"
#include "prt_asm_cpu_external.h"
#include "os_cpu_armv8_external.h"
#include "prt_amp_task_internal.h"

// 核的局部运行队列，全局变量，用于调度就绪任务
OS_SEC_BSS struct TagOsRunQue g_runQueue;

/*
* 描述：将任务添加到就绪队列。
* 调用者必须确保当前操作不会发生核切换，并且已经加锁了运行队列。
*/
OS_SEC_L0_TEXT void OsTskReadyAdd(struct TagTskCb *task)
{
    struct TagOsRunQue *rq = &g_runQueue; // 获取当前核的运行队列指针
    TSK_STATUS_SET(task, OS_TSK_READY);   // 设置任务状态为“就绪”

    OS_TSK_EN_QUE(rq, task, 0);           // 将任务加入运行队列（0 表示普通优先级操作）
```

```
    OsTskHighestSet();                      // 更新当前核的最高优先级任务

    return;
}

/*
* 描述：将任务从就绪队列中移除。
* 注意：此操作的调用者必须在外部关中断以确保线程安全。
*/
OS_SEC_L0_TEXT void OsTskReadyDel(struct TagTskCb *taskCb)
{
    struct TagOsRunQue *runQue = &g_runQueue; // 获取当前核的运行队列指针
    TSK_STATUS_CLEAR(taskCb, OS_TSK_READY);   // 清除任务的“就绪”状态标志

    OS_TSK_DE_QUE(runQue, taskCb, 0);         // 将任务从运行队列中删除
    OsTskHighestSet();                        // 更新当前核的最高优先级任务

    return;
}

// src/core/kernel/task/prt_task_del.c
/*
* 描述：任务退出函数，当任务主动结束时调用此函数。
* 步骤包括从就绪队列删除该任务并进行任务调度。
*/
OS_SEC_L4_TEXT void OsTaskExit(struct TagTskCb *tsk)
{
    uintptr_t intSave = OsIntLock();   // 关中断，保存中断状态，进入临界区

    OsTskReadyDel(tsk);                // 从就绪队列中移除该任务
    OsTskSchedule();                   // 立即触发任务调度，切换到下一个合适的任务

    OsIntRestore(intSave);             // 恢复中断状态，退出临界区
}
```

其中，OS_TSK_EN_QUE 和 OS_TSK_DE_QUE 宏在 src/include/prt_amp_task_internal.h 定义。

调度与切换

src/kernel/sched/prt_sched_single.c

```
#include "prt_task_external.h"
#include "os_attr_armv8_external.h"
#include "prt_asm_cpu_external.h"
#include "os_cpu_armv8_external.h"

/*
* 描述：任务调度函数，判断是否需要任务切换，并进行上下文切换
* 调用场景：由系统主动触发，已确保外部已关中断
*/
OS_SEC_TEXT void OsTskSchedule(void)
{
    /* 外层已经关闭中断，避免并发 */

    /* 设置当前核上的最高优先级任务 */
    OsTskHighestSet();

    /* 如果当前运行任务不是最高优先级任务，且任务调度未被锁定，则触发任务调度请求 */
    if ((g_highestTask != RUNNING_TASK) && (g_uniTaskLock == 0)) {
        UNI_FLAG |= OS_FLG_TSK_REQ;  // 设置调度请求标志位

        /* 如果当前不是在中断或系统Tick上下文中，则可以立即进行调度切换 */
        if (OS_INT_INACTIVE) {  // OS_INT_INACTIVE 宏判断是否处于中断上下文中
            OsTaskTrap();       // 执行任务上下文切换
            return;
        }
    }

    return;  // 若无调度需求或当前处于中断中，则不执行调度
}

/*
* 描述：调度的主入口函数，执行任务切换
* 通常在系统检测到调度请求标志时由中断返回路径调用
*/
OS_SEC_L0_TEXT void OsMainSchedule(void)
{
    struct TagTskCb *prevTsk;

    // 如果调度请求标志被设置
    if ((UNI_FLAG & OS_FLG_TSK_REQ) != 0) {
        prevTsk = RUNNING_TASK;  // 记录当前运行任务

        // 清除调度请求标志位，表示即将处理调度
```

```
        UNI_FLAG &= ~OS_FLG_TSK_REQ;

        // 当前任务取消运行状态
        RUNNING_TASK->taskStatus &= ~OS_TSK_RUNNING;

        // 设置最高优先级任务为运行状态
        g_highestTask->taskStatus |= OS_TSK_RUNNING;

        // 切换当前任务指针
        RUNNING_TASK = g_highestTask;
    }

    // 如果没有任务调度请求，仍然跳转回当前任务上下文
    OsTskContextLoad((uintptr_t)RUNNING_TASK);  // 加载任务上下文并恢复执行
}

/*
* 描述：系统启动阶段的首次任务调度，启动第一个任务
*/
OS_SEC_L4_TEXT void OsFirstTimeSwitch(void)
{
    OsTskHighestSet();                  // 获取当前系统中最高优先级任务
    RUNNING_TASK = g_highestTask;      // 设置为当前运行任务
    TSK_STATUS_SET(RUNNING_TASK, OS_TSK_RUNNING);  // 设置任务状态为“运行中”
    OsTskContextLoad((uintptr_t)RUNNING_TASK);      // 启动任务上下文执行

    // 此处永远不应返回，一旦返回则说明调度失败或异常
    return;
}
```

其中，OsTskHighestSet 函数在 src/include/prt_task_external.h 中被定义为内联函数，提高性能。

```
/*

* 模块内内联函数定义

*/

OS_SEC_ALW_INLINE INLINE void OsTskHighestSet(void)

{
```

```
    struct TagTskCb *taskCb = NULL;

    struct TagTskCb *savedTaskCb = NULL;
    // 遍历g_runQueue队列，查找优先级最高的任务

    LIST_FOR_EACH(taskCb, &g_runQueue, struct TagTskCb, pendList) {

        // 第一个任务，直接保存到savedTaskCb

        if(savedTaskCb == NULL) {

            savedTaskCb = taskCb;

            continue;

        }

        // 比较优先级，值越小优先级越高

        if(taskCb->priority < savedTaskCb->priority){

            savedTaskCb = taskCb;

        }

    }
    g_highestTask = savedTaskCb;

}
```

在 src/bsp/prt_vector.S 实现 OsTskContextLoad，OsContextLoad 和 OsTaskTrap

```
/*
* 描述：恢复任务上下文并跳转执行（首次调度或上下文切换时调用）
* 原型：void OsTskContextLoad(uintptr_t stackPointer)
*/
    .globl OsTskContextLoad             // 声明为全局符号
    .type OsTskContextLoad, @function   // 声明符号类型为函数
    .align 4

OsTskContextLoad:
    ldr    X0, [X0]        // X0 是传入的 &stackPointer，先读取真正的栈指针值
```

```
    mov     SP, X0          // 设置 SP = X0，恢复当前任务的堆栈指针

OsContextLoad:
    ldp     x2, x3, [sp], #16   // 弹出 x2 = elr_el1（返回地址），x3 = spsr_el1
（程序状态）
    add     sp, sp, #16         // 跳过异常相关寄存器（ESR_EL1 和 FAR_EL1），系统
启动时这些无意义

    msr     spsr_el1, x3        // 恢复 saved program status register
    msr     elr_el1, x2         // 恢复异常返回地址（实际执行的代码位置
OsTskEntry）

    dsb     sy                  // 数据同步屏障，确保前面设置的寄存器生效
    isb                         // 指令同步屏障

    RESTORE_EXC_REGS            // 宏：恢复异常现场寄存器（通用寄存器、浮点寄存器
等）

    eret                        // 从异常返回（进入任务执行）

/*
* 描述：任务调度处理函数，保存当前任务上下文，跳转执行调度主流程
* 参数：X0 为 g_runningTask（当前任务控制块指针）
*/
    .globl OsTaskTrap
    .type OsTaskTrap, @function
    .align 4

OsTaskTrap:
    LDR     x1, =g_runningTask  // 获取当前运行任务的指针地址，x1 =
&g_runningTask
    LDR     x0, [x1]            // x0 = g_runningTask（即当前任务控制块）

    SAVE_EXC_REGS               // 宏：保存所有通用寄存器和系统上下文到任务栈

    /*
    * 保存 CPSR 状态：
    * - DAIF 表示中断屏蔽位（IRQ、FIQ、SError、Debug）
    * - NZCV 是条件码标志
    * - 拼接得到当前 CPSR 伪值
    */
    mrs     x3, DAIF            // 获取中断屏蔽标志位
    mrs     x2, NZCV            // 获取条件码
```

```
    orr    x3, x3, x2          // 合并为 CPSR 基础部分
    orr    x3, x3, #(0x1U << 2) // 设置当前 Exception Level 位（假设为 EL1）
    orr    x3, x3, #(0x1U)     // 设置使用 SP_ELX 栈标志（1 表示使用 SP_EL1）

    mov    x2, x30             // 保存返回地址 x30（LR）作为异常返回点
    sub    sp, sp, #16         // 栈指针向下移动，预留 esr_el1, far_el1 空间
    stp    x2, x3, [sp, #-16]! // 将 x2（elr）和 x3（spsr）压栈保存

    // 将当前任务栈顶地址保存到任务控制块中（g_runningTask->sp = sp）
    mov    x1, sp
    str    x1, [x0]            // 存储栈指针到 g_runningTask->sp

    B      OsMainSchedule      // 跳转调用调度主函数，进行任务切换

loop1:
    B      loop1               // 死循环（理论上不会执行到此）
```

在 src/bsp/os_cpu_armv8_external.h 加入 OsTaskTrap 和 OsTskContextLoad 的声明和关于栈地址和大小对齐宏。

```
#define OS_TSK_STACK_SIZE_ALIGN  16U

#define OS_TSK_STACK_SIZE_ALLOC_ALIGN 4U //按2的幂对齐，即2^4=16字节

#define OS_TSK_STACK_ADDR_ALIGN  16U



extern void OsTaskTrap(void);

extern void OsTskContextLoad(uintptr_t stackPointer);
```

最后在 src/kernel/task/prt_sys.c 定义了内核的各种全局数据。

```
#include "prt_typedef.h"

#include "os_attr_armv8_external.h"

#include "prt_task.h"
```

```
OS_SEC_L4_BSS U32 g_threadNum;



/* Tick计数 */

extern U64 g_uniTicks; // 把 lab5 中在 src/kernel/tick/prt_tick.c 定义的
g_uniTicks 移到此处则取消此行的注释


/* 系统状态标志位 */

OS_SEC_DATA U32 g_uniFlag = 0;



OS_SEC_DATA struct TagTskCb *g_runningTask = NULL;



// src/core/kernel/task/prt_task_global.c

OS_SEC_BSS TskEntryFunc g_tskIdleEntry;




OS_SEC_BSS U32 g_tskMaxNum;

OS_SEC_BSS struct TagTskCb *g_tskCbArray;

OS_SEC_BSS U32 g_tskBaseId;



OS_SEC_BSS TskHandle g_idleTaskId;

OS_SEC_BSS U16 g_uniTaskLock;

OS_SEC_BSS struct TagTskCb *g_highestTask;
```

```
[100%] Linking C executable miniEuler
/home/xiaoye/OSlab/aarch64-none-elf/bin/../lib/gcc/aarch64-none-elf/11.2.1/../../../../aarch64-none-elf/bin/ld: kernel/task/CMak
eFiles/task.dir/prt_sys.c.obj:/home/xiaoye/OSlab/lab6/src/kernel/task/prt_sys.c:9: multiple definition of `g_uniTicks'; kernel/t
ick/CMakeFiles/tick.dir/prt_tick.c.obj:/home/xiaoye/OSlab/lab6/src/kernel/tick/prt_tick.c:9: first defined here
```

补充说明

在C语言中，若在两个 `.c` 文件中重复定义同名全局变量，会导致**链接阶段的多重定义错误**。
上图的注释很清楚的说明：在prt_tick.c、prt_sys.c都需要定义变量g_uniTicks。
所以，为了正确使用，在prt_sys.c中声明其为外部引用extern。
不要使用static，因为这个变量需要全局共享。

任务调度测试

```
#include "prt_typedef.h"

#include "prt_tick.h"

#include "prt_task.h"



extern U32 PRT_Printf(const char *format, ...);

extern void PRT_UartInit(void);

extern void CoreTimerInit(void);

extern U32 OsHwiInit(void);

extern U32 OsActivate(void);

extern U32 OsTskInit(void);



void Test1TaskEntry()

{

    PRT_Printf("task 1 run ...\n");



    U32 cnt = 5;
```

```
    while (cnt > 0) {

        // PRT_TaskDelay(200);

        PRT_Printf("task 1 run ...\n");

        cnt--;

    }

}



void Test2TaskEntry()

{

    PRT_Printf("task 2 run ...\n");



    U32 cnt = 5;

    while (cnt > 0) {

        // PRT_TaskDelay(100);

        PRT_Printf("task 2 run ...\n");

        cnt--;

    }

}



S32 main(void)

{
```

```
    // 初始化GIC

    OsHwiInit();

    // 启用Timer

    CoreTimerInit();

    // 任务系统初始化

    OsTskInit();



    PRT_UartInit();



    PRT_Printf("            _       _ _____      _             _
_   _ _   _ _   _           \n");

    PRT_Printf("  _ __ ___ (_)_ __ (_) ____|   _| | ___ _ __  | |__  _   _
 | | | | \\ | | | | ___ _ __ \n");

    PRT_Printf(" | '_ ` _ \\| | '_ \\| |  _|| | | | |/ _ \\ '__| | '_ \\| |
| | | |_| |  \\| | | | |/ _ \\ '__|\n");

    PRT_Printf(" | | | | | | | | | | |__| |_| | |  __/ |    | |_) | |_| |
|  _  | |\\  | |_| |  __/ |   \n");

    PRT_Printf(" |_| |_| |_|_|_| |_|_|_____\\__,_|_|\\___|_|    |_.__/ \\__,
| |_| |_|_| \\_|\\___/ \\___|_|   \n");

    PRT_Printf("                                                |___/
                          \n");



    PRT_Printf("ctr-a h: print help of qemu emulator. ctr-a x: quit
emulator.\n\n");
```

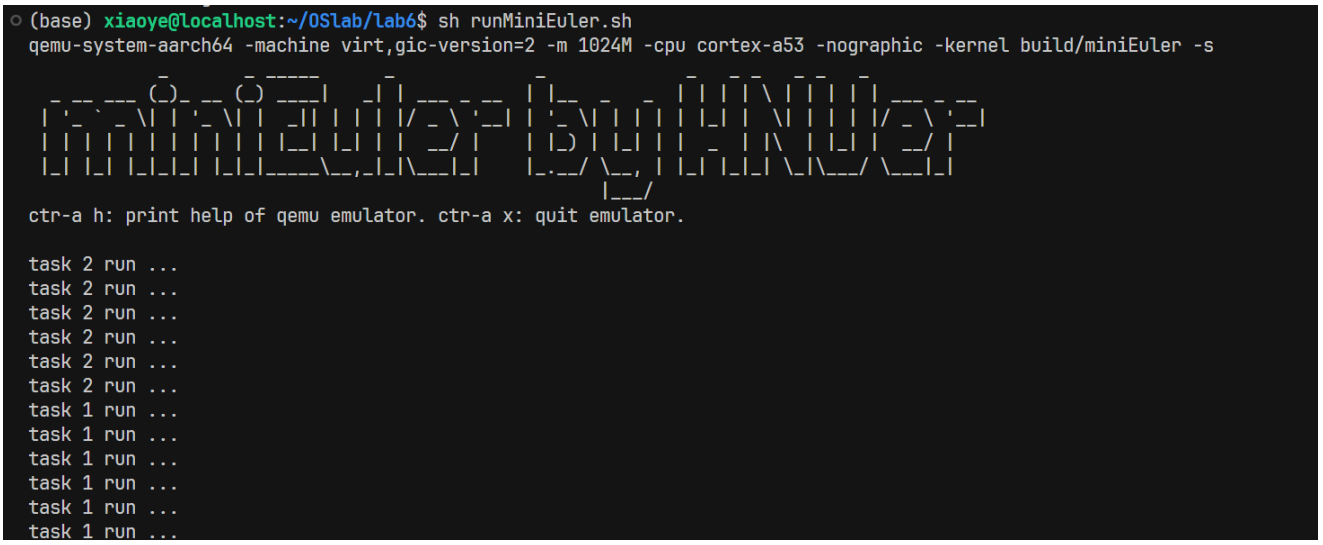
```
    U32 ret;
    struct TskInitParam param = {0};


    // task 1
    // param.stackAddr = 0;
    param.taskEntry = (TskEntryFunc)Test1TaskEntry;// 设置任务入口函数
    param.taskPrio = 35;// 设置任务优先级（值越小优先级越高）
    // param.name = "Test1Task";
    param.stackSize = 0x1000; //固定4096，参见prt_task_init.c的
OsMemAllocAlign


    TskHandle tskHandle1;
    ret = PRT_TaskCreate(&tskHandle1, &param);// 创建任务并获取其句柄
    if (ret) {
        return ret;
    }


    ret = PRT_TaskResume(tskHandle1);// 将任务从创建态切换到就绪态
    if (ret) {
        return ret;
    }
```

```
    // task 2
    // param.stackAddr = 0;
    param.taskEntry = (TskEntryFunc)Test2TaskEntry;
    param.taskPrio = 30;
    // param.name = "Test2Task";
    param.stackSize = 0x1000; //固定4096，参见prt_task_init.c的
OsMemAllocAlign



    TskHandle tskHandle2;
    ret = PRT_TaskCreate(&tskHandle2, &param);
    if (ret) {
        return ret;
    }



    ret = PRT_TaskResume(tskHandle2);
    if (ret) {
        return ret;
    }



    // 启动调度
    OsActivate();
```

```
    // while(1);

    return 0;



}
```

提示：将新建文件加入构建系统

加入之后成功运行。



作业1

实现分时调度。

提示：分时调度的调度点存在于时钟Tick中断、任务结束等处。

第一次尝试

lab6_1
1.修改 prt_tick.c：

```
extern void OsTimerInterrupt(void);//通过extern声明了一个外部实现的定时器中断处理函数
```

2.新建 src/kernel/sched/prt_sched_rr.c

```
/*------------------ 头文件与宏定义 ------------------*/
#include "prt_task_external.h"        // 任务管理外部头文件
```

```
#include "os_attr_armv8_external.h"  // ARMv8架构属性定义
#include "prt_asm_cpu_external.h"    // CPU汇编相关定义
#include "os_cpu_armv8_external.h"   // ARMv8 CPU操作接口

#define OS_TASK_TIME_SLICE_TICKS 50  // 定义时间片长度为50个时钟滴答

extern void OsgicIntClear(U32 value); // 声明中断清除函数
extern U64 PRT_TickGetCount(void);    // 声明获取系统滴答计数的函数

U64 taskStartTick = 0;               // 记录任务启动时的滴答值

#include "prt_amp_task_internal.h"    // 包含任务管理内部实现

/*------------------ 队列操作函数 ------------------*/
/* 从队列取出首元素并插入末尾 */
#define LIST_FIRST_ELEM(list, type, field) \
    (type*)((list)->next)  // 通过链表指针获取首元素

OS_SEC_ALW_INLINE void ListPopAndPushFirst(struct TagList *listObject) {
    struct TagTskcb *taskCb = LIST_FIRST_ELEM(listObject, struct TagTskcb,
pendList);
    OsDequeueTaskAMP(&g_runQueue, taskCb); // 从运行队列移除任务
    OsEnqueueTaskAMP(&g_runQueue, taskCb); // 将任务重新插入队列末尾
}

/*------------------ FIFO任务调度设置 ------------------*/
OS_SEC_ALW_INLINE void OsTskFIFOSet(void) {
    struct TagTskcb *nextTaskCb = LIST_FIRST_ELEM(&g_runQueue, struct
TagTskcb, pendList);

    // 如果下个任务是空闲任务，则将其移到队列末尾
    if (nextTaskCb->taskPid == g_idleTaskId) {
        ListPopAndPushFirst(&g_runQueue);
        nextTaskCb = LIST_FIRST_ELEM(&g_runQueue, struct TagTskcb,
pendList);
    }
    g_highestTask = nextTaskCb; // 设置为最高优先级任务
}

/*------------------ 主调度函数 ------------------*/
OS_SEC_TEXT void OsTskSchedule(void) {
    /* 注意：外层已关闭中断 */
    OsTskFIFOSet(); // 确定下一个要运行的任务
```

```
    // 当最高优先级任务非当前任务且任务锁未启用时
    if ((g_highestTask != RUNNING_TASK) && (g_uniTaskLock == 0)) {
        UNI_FLAG |= OS_FLG_TSK_REQ; // 设置任务切换请求标志

        // 如果当前不在中断上下文中，触发任务陷阱切换
        if (OS_INT_INACTIVE) {
            OsTaskTrap();
        }
    }
}

/*------------------ 定时器中断处理 ------------------*/
OS_SEC_TEXT void OsTimerInterrupt(void) {
    OsgicIntClear(30); // 清除定时器中断（硬件特定操作）

    // 检查后台任务未激活时直接返回
    if ((OS_FLG_BGD_ACTIVE & UNI_FLAG) == 0) {
        return;
    }

    // 时间片耗尽且当前任务处于就绪状态
    if ((PRT_TickGetCount() - taskStartTick) > OS_TASK_TIME_SLICE_TICKS) {
        if ((RUNNING_TASK->taskStatus & OS_TSK_READY) != 0) {
            ListPopAndPushFirst(&g_runQueue); // 轮转当前任务到队列末尾
            OsTskSchedule(); // 触发任务调度
        }
    }
}

/*------------------ 调度主入口 ------------------*/
OS_SEC_L4_TEXT void OsMainSchedule(void) {
    struct TagTskcb *prevTsk;

    // 当检测到任务切换请求标志时
    if ((UNI_FLAG & OS_FLG_TSK_REQ) != 0) {
        prevTsk = RUNNING_TASK;
        UNI_FLAG &= ~OS_FLG_TSK_REQ; // 清除请求标志

        // 更新任务状态：旧任务退出运行态，新任务进入运行态
        RUNNING_TASK->taskStatus &= ~OS_TSK_RUNNING;
        g_highestTask->taskStatus |= OS_TSK_RUNNING;
```

```
        RUNNING_TASK = g_highestTask;           // 切换当前运行任务
        taskStartTick = PRT_TickGetCount();     // 更新任务开始Tick值

        OsTskContextLoad((uintptr_t)RUNNING_TASK); // 加载新任务上下文
    }
}

/*------------------ 首次任务切换 ------------------*/
OS_SEC_L4_TEXT void OsFirstTimeSwitch(void) {
    OsTskFIFOSet(); // 初始化运行队列

    RUNNING_TASK = g_highestTask;               // 设置当前任务为最高优先级任务。
    taskStartTick = PRT_TickGetCount();         // 更新任务开始Tick值
    TSK_STATUS_SET(RUNNING_TASK, OS_TSK_RUNNING); // 设置任务状态为运行中

    OsTskContextLoad((uintptr_t)RUNNING_TASK); // 加载任务上下文
    return; // 理论上不会执行到此（上下文切换后不会返回）
}
```

新建 rr 调度机制文件，实现新添加的定时器中断处理函数 OsTimerInterrupt()。

关键点说明：

- OsTskFIFOSet：structTagTskCb * nextTaskCb=NULL;：定义一个指向TagTskCb结构体的指针变量nextTaskCb，并初始化为NULL。使用宏LISTFIRSTELEM从运行队列g_runQueue中获取第一个任务的控制块指针。检查获取到的第一个任务是否是空闲任务（通过比较任务的PID）。如果第一个任务是空闲任务，则调用ListPopAndPushFirst函数将该任务移到队列末尾。再次获取队列中的第一个任务。将最高优先级任务设置为从队列中获取到的第一个任务。
- OsTskSchedule：调用OsTskFIFOSet函数查找下一个任务。如果最高优先级任务不是当前运行任务，并且任务锁没有被锁定，则设置调度请求标志。如果不在中断上下文中，则调用OsTaskTrap进行任务切换。
- OsTimerInterrupt：清除中断。如果后台任务标志没有激活，则直接返回。如果当前Tick计数减去任务开始Tick值大于任务时间片（也就是说时间片耗尽），并且当前任务是就绪状态，则将当前任务放到队列最后，并进行任务调度。
- 调度主入口和首次上下文切换见注释。

> **补充说明**
>
> 为了避免函数重复定义的错误，同时支持多种调度策略（如时间片轮转 `RR` 和单任务调度 `Single`），需通过 **条件编译** 隔离不同策略的代码。

在 `CMakeLists.txt` 中配置编译选项，确保同一时间只编译一种调度策略：
当然啦，完全可以新建一个副本，其中只包括一种调度策略。

整段代码通过定时器中断来监控任务的执行时间，当一个任务的时间片（设定为50个时钟周期）用尽时，系统会将该任务移到队列的尾部，并根据FIFO调度算法选择下一个任务执行。如果当前任务是空闲任务，则继续查找直到找到非空闲任务。定时器中断处理函数清除中断标志，检查并切换任务，通过上下文切换机制确保新任务能够正确执行，从而实现系统内多个任务的公平分时调度。

第二次尝试

lab6.0

1.在OsTskHighwstSet函数中增加一项priority++的操作。由于每次调度都会调用该函数，这就使得每一次调度完成之后对应任务的优先级都减一。
2.将两个任务的初始优先级均设置为最高（priority越小优先级越高）。
3.修改两个任务，将时钟信号转换为电平信号。每隔一段固定的时间就触发高电平，引发调度，实现分时共享。

```
OS_SEC_ALW_INLINE INLINE void OsTskHighestSet(void)

{

    struct TagTskCb *taskCb = NULL;

    struct TagTskCb *savedTaskCb = NULL;
    // 遍历g_runQueue队列，查找优先级最高的任务

    LIST_FOR_EACH(taskCb, &g_runQueue, struct TagTskCb, pendList) {

        // 第一个任务，直接保存到savedTaskCb

        if(savedTaskCb == NULL) {

            savedTaskCb = taskCb;

            continue;

        }

        // 比较优先级，值越小优先级越高

        if(taskCb->priority < savedTaskCb->priority){
```

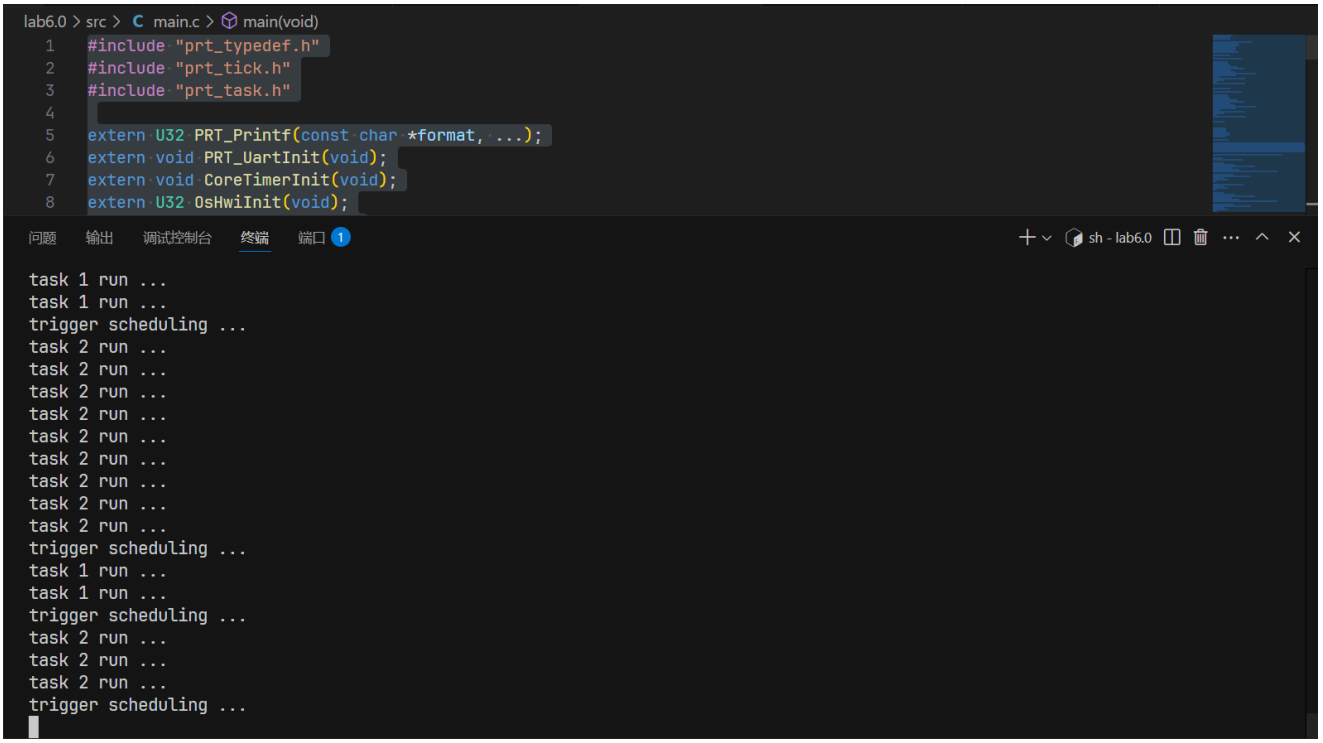
```
            savedTaskCb = taskCb;

        }

    }

    savedTaskCb->priority++;
    g_highestTask = savedTaskCb;

}
```

```
#include "prt_typedef.h"

#include "prt_tick.h"

#include "prt_task.h"



extern U32 PRT_Printf(const char *format, ...);

extern void PRT_UartInit(void);

extern void CoreTimerInit(void);

extern U32 OsHwiInit(void);

extern U32 OsActivate(void);

extern U32 OsTskInit(void);

extern volatile U32 g_uniTicks;//注意声明外部引用变量 g_uniTicks



void Test1TaskEntry()

{

    PRT_Printf("task 1 run ...\n");
```

```
    //U32 cnt = 5;
    U32 cnt = 1000;
    while (cnt > 0) {
        // PRT_TaskDelay(200);
        U32 tick = PRT_TickGetCount();
        if (tick>=1){
            g_uniTicks = 0;
            PRT_Printf("trigger scheduling ... \n");
            OsTskSchedule();
        }
        PRT_Printf("task 1 run ...\n");
        cnt--;
    }
   PRT_Printf("\ntask 1 done ...\n\n");
}

void Test2TaskEntry()
{
    PRT_Printf("task 2 run ...\n");

    //U32 cnt = 5;
```

```
    U32 cnt = 1000;

    while (cnt > 0) {

        // PRT_TaskDelay(100);

        U32 tick = PRT_TickGetCount();

        if (tick>=1){

            g_uniTicks = 0;

            PRT_Printf("trigger scheduling ... \n");

            OsTskSchedule();

        }

        PRT_Printf("task 2 run ...\n");

        cnt--;

    }

    PRT_Printf("\ntask 2 done ...\n\n");

}
```

运行结果如下：



逻辑解释

ticks是操作系统中的基础时间单位。在我们的代码中，1tick=1ms。这里的分时逻辑是，每次达到占用CPU的时间大于等于1ms时，主动让出CPU，从就绪队列中开始调度。
而为了避免高优先级的抢占，我们将两个任务都初始化为最高的优先级。并且在原先的代码OsTskHighestSet（）中加入savedTaskCb->priority++（值越高，优先级越低），即每次调度优先级都减去1。这样下一次，它调度选择的就是高优先级的任务（另一个没有被调度的任务）。逻辑可类比于动态优先级轮转调度。

第三次尝试

lab6_2

通过给每个进程设置时间片进行轮转调度

- 1.首先修改Dispatch调度函数，在中断清除之后新增设调度的情况，即当满足轮转调度或者优先级调度（当某个任务运行结束后，将最高优先级的任务调度运行）的条件，并且相应调度队列不为空时，调用OsTskSchedule函数发起调度；
- 2.接着仿照优先级调度函数中找到运行队列中的最高优先级任务OsTskhighest_Set函数，定义轮转调度函数中寻找下一运行任务的函数OsTskRR_Set，其中定义了每一个任务的时间片长度为100个时钟中断的时间；
- 3.在main函数中对轮转调度队列进行初始化；
- 4.最后修改任务1、2，延长运行的时间，便于更加直观地看到轮转运行