Lab6

一、 实验目的

1. 理解操作系统核心功能–任务调度，以及调度系统的工作原理
2. 实现一个单进程支持多线程的抢占式调度系统
3. 实现一个单进程支持多线程的分时调度系统

二、 实验过程

1、实现调度系统的基础数据结构：双向链表

① 在include目录下创建lists_type.h文件，定义双向链表结构

② 在include目录下创建prt_list_external.h ，定义链表的各种相关操作

本文件中定义了一系列宏以及链表相关处理函数：

- ListLowLevelAdd：本函数用于在两个已存在的节点prev和next之间插入一个新的节点newNode。
- ListAdd：本函数将新的节点newNode添加到链表的头部，即listObject节点的后面。
- ListTailAdd：此函数将新的节点newNode添加到链表的尾部，即listObject节点的前面。
- ListLowLevelDelete：这是一个删除函数，用于删除prevNode和nextNode之间的节点。
- ListDelete：此函数删除指定的节点node，并将它从链表中移除。
- ListEmpty：本函数检查链表是否为空。通过判断链表的next和prev是否都指向它自己来判断是否为空
- OFFSET_OF_FIELD：这个宏用于获取结构体成员field在结构体type中的偏移量。
- COMPLEX_OF：这个宏通过给定的成员field的指针ptr，获取控制块的首地址。（利用成员field的指针ptr减去field在type类型的控制块中的偏移量）
- LIST_COMPONENT：这个宏是COMPLEX_OF宏的别名，根据成员地址prt获取控制块的首地址。
- LIST_FOR_EACH和LIST_FOR_EACH_SAFE：这两个宏用于在链表中遍历每一个元素，主要目的在于简化代码。

2、实现调度系统的任务控制块

此处直接按照指导书上在对应目录下创建相应名字的文件，然后复制粘贴即可，下面仅是代码部分片段的分析。

① 在include目录下创建文件prt_task.h，定义了一些相关宏定义以及任务创建时参数传递的结构体： struct TskInitParam。

② 在include目录下创建文件prt_task_external.h ，定义任务调度中最重要的数据结构——任务控制块 struct TagTskCb。

③ 最后在include目录下引入文件prt_amp_task_internal.h，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。

3、实现调度系统中任务的创建

在src/kernel/task目录下创建文件prt_task_init.c，用于实现任务创建代码（下文代码如果没有明确说明所在文件，均在此文件中，某些代码段开头有// src/core/kernel/task/prt_task_internal.h之类的注释，部分同学可能会误以为是代码对应的位置，导致最后程序报错并且难以找到原因。）

① 相关变量与函数声明

② 极简内存空间管理

③ 任务栈初始化

根据课程《操作系统》的相关学习，了解到当发生任务切换时，会产生上下文的恢复与保存，而当任务第一次执行时，内核栈是空的，无法进行上下文的恢复，因此我们需要先在内核栈中放置第一个任务的上下文内容，从而进行上下文恢复，此处最重要的是寄存器x30以及spsr（异常发生时的程序状态）的值。

④ 在include目录下的文件os_cpu_armv8.h加入任务上下文结构体TskContext的定义

此处原先应该是在bsp目录下的文件os_cpu_armv8.h中被定义，但是由于上一个实验在include目录下定义了同名文件os_cpu_armv8.h，此处将二者合并，将上下文结构体定义在include目录下的os_cpu_armv8.h中

⑤ 实现任务入口函数

在任务入口函数中，首先定义了该任务的TCB任务控制块，然后关闭中断，以防止在任务调度过程中产生中断，同时用intSave保存中断状态，待调度结束后恢复原来的中断状态，当调度完成后，调用OsTaskExit函数释放TCB资源。纵观整个调度系统，找不到类似OsTskEntry(taskId)这样的对 OsTskEntry进行的函数调用。事实是在通过 OsTskContextInit 函数进行栈初始化时传入，即任务第一次就绪运行时会进入OsTskEntry执行，这也符合任务入口函数的定义。

⑥ 实现任务创建

注意：这段代码的开头就是前文提到的可能致误的注释

对任务创建的函数进行分析：

- OsTaskCreateChkAndGetTcb 函数：检查空闲链表中是否还有任务块，如果有则从空闲链表 g_tskCbFreeList 中取出一个任务控制块，然后将该任务块从空闲链表中去除；
- OsCheckAddrOffsetOverflow：检查地址加上分配大小是否会导致地址溢出。
- OsTaskCreateRsrcInit函数：初始化任务资源。如果用户已经设置了任务栈，就使用用户配置的栈，否则通过 OsTskMemAlloc 为新建的任务分配堆栈空间。此函数返回任务栈的顶部地址和大小；
- OsTskCreateTcbInit：初始化任务控制块TCB。设置任务的栈指针，参数，栈的顶部地址和大小，优先级，入口函数等信息。
- 创建一个任务：首先获取一个空闲的任务控制块，然后初始化任务资源，初始化任务控制块，最后设置任务状态为挂起，并返回任务ID（并不激活此任务）
- OsTskContextInit 函数负责将栈初始化成刚刚发生过中断一样；
- PRT_TaskCreate：这是一个简单封装函数，调用OsTaskCreateOnly函数

⑦ 实现解挂任务

下面对此部分的函数进行分析：

- OsMoveTaskToReady: 此函数首先进行判断，如果任务处于可中断延迟状态并且超时，它会清除任务的延迟等待标志位，然后如果任务没有被阻塞，则将任务添加到就绪列表，同时检查UNI_FLAG中的OS_FLG_BGD_ACTIVE位是否被设置，若设置则引发调度程序。
- PRT_TaskResume: 此函数首先获取指定任务的控制块，然后检查任务是否正在使用。如果任务未创建或者正在运行且任务锁定，函数返回错误。如果任务没有被挂起或处于可中断延迟状态，函数同样返回错误。如果函数没有返回错误，则正常清除任务的挂起状态，并调用OsMoveTaskToReady将任务移到就绪列表。

⑧ 任务管理系统的初始化与启动

对该模块进行分析：

- OsTskAMPInit: 这个AMP任务初始化函数首先分配了4096字节的内存用于存储TCB任务控制块数组，每个TCB用于存储一个任务的信息，同时任务和TCB的一一映射关系也表明数组中的元素个数即为任务的最大个数（减2的目的是预留一个空闲任务块、一个无效任务块）。然后将所有的任务控制块加入到空闲列表，并设置每个任务控制块的初始状态和任务ID，同时给RUNNING_TASK的PID赋一个合法的无效值，防止在Trace使用时出现异常，然后增加OS_TSK_INUSE状态，使得在Trace记录的第一条信息状态为OS_TSK_INUSE。最后，初始化运行队列，并设置了当前运行任务的状态和优先级。

- OsTskInit: 本函数是一个封装函数，调用OsTskAMPInit进行AMP任务的初始化，如果初始化成功则返回OS_OK。
- OsTskIdleBgd: 本函数定义了Idle的操作：执行空循环，当系统没有其他任务需要运行时，会执行空闲任务IDLE。
- OsIdleTskAMPCreate: 本函数用于创建IDLE空闲任务，其首先设置了任务的参数，包括任务的入口函数、堆栈大小、优先级等，然后创建任务，并恢复（或解挂）这个任务。最后，将任务的ID保存到IDLE_TASK_ID。
- OsActivate: 本函数用于激活任务管理函数，其首先调用 OsIdleTskAMPCreate 函数创建IDLE任务，使系统在没有任务就绪时运行IDLE空闲任务，然后调用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，设置最高优先级的任务，同时标记背景任务正在运行，并开始执行多任务管理。（如果函数能够正常返回，说明任务调度出现了问题）

⑨ 在include目录下的文件prt_config.h 中加入空闲任务优先级定义

4、 任务状态转换

在 src/kernel/task中创建文件prt_task.c：

相关函数的分析:

- OsTskReadyAdd:本函数实现将一个任务添加到就绪队列中（与上文的另一个就绪添加函数相比，免去了一系列判断条件），它首先获取全局运行队列g_runQueue，然后设置任务的状态为就绪 (OS_TSK_READY)，并把任务添加到运行队列中，最后调用OsTskHighestSet()将g_highestTask 指针指向最高优先级任务（每当就绪队列中的任务发生变化时，要重新找到当前最高优先级的任务）。
- OsTskReadyDel:本函数实现将一个任务从就绪队列中移除，与添加函数相同，首先获取全局运行队列g_runQueue，然后清除任务的就绪状态 (OS_TSK_READY)，并从运行队列中移除该任务。最后，它同样调用OsTskHighestSet()将g_highestTask 指针指向最高优先级任务。
- OsTaskExit: 本函数实现任务退出，其首先锁定中断（防止退出过程引发中断），然后调用OsTskReadyDel()将任务从就绪队列中移除，最后调用OsTskSchedule()进行任务调度（因为一个任务运行结束之后，需要陷入操作系统来引发调度），最后恢复中断。

5、实现调度与切换

① 在src/kernel/sched目录下创建文件prt_sched_single.c

下面分析本文件中相关函数的功能:

- OsTskSchedule函数：本函数函数实现任务调度，首先调用OsTskHighestSet来设置g_highestTask 指针指向最高优先级任务，如果当前运行的任务不是最高优先级的任务，并且没有被锁定，就设置一个标志位，请求任务调度。如果当前不在中断上下文中，就调用OsTaskTrap来进行任务切换，否则就需要等待中断历程结束后才能进行任务切换。

- OsMainSchedule函数：本函数是任务调度的主入口，当有任务调度请求时（检查标志位OS_FLG_TSK_REQ），会保存当前运行的任务，清除任务调度请求标志位，然后更新任务的状态，并切换到最高优先级的任务，最后调用OsTskContextLoad加载新的任务上下文。
- OsFirstTimeSwitch函数：本函数是系统启动时的首次任务调度，首先调用OsTskHighestSet设置g_highestTask 指针指向最高优先级任务，然后设置当前运行的任务为最高优先级的任务，并设置任务状态为运行。最后，调用OsTskContextLoad加载新的任务上下文。

② 在src/include/prt_task_external.h 中定义内联函数OsTskHighestSet 函数

本函数在前文也已经使用，即遍历整个g_runQueue队列，查找最高优先级任务并将g_highestTask 指针指向该任务，此处定义为内联函数可以提高性能。

③ 在 src/bsp目录下创建文件prt_vector.S，实现OsTskContextLoad，OsContextLoad 和 OsTaskTrap。

④ 在 src/bsp目录下的文件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。

⑤ 最后在 src/kernel/task目录下的文件prt_sys.c中定义内核的各种全局数据，至此，调度系统构建完成

三、 测试及分析

运行测试抢占式调度系统的任务调度，发现正常进行：

四、 作业

实现分时调度系统

1. 在任务控制块中添加时间片属性

首先在`prt_config.h`中定义时间片初始值的宏:

```c
#define OS_TSK_DEFAULT_TIMESLICE 5
```

然后在`prt_task_external.h`的`TagTskCb`定义中添加时间片的属性:

```c
struct TagTskCb {
    /* 当前任务的SP */
    void *stackPointer;
    /* 任务的剩余时间片 */
    U32 timeSlice; // 这一行为需要添加的内容
    /* 任务状态,后续内部全改成U32 */
    U32 taskStatus;
    // 其他定义
}
```

注意：剩余时间片这个属性不能添加到结构体的开头，因为汇编代码OsTskContextLoad假定了结构体开头一定是任务的栈指针

然后在`prt_task_init.c`中为剩余时间片初始化:

```c
OS_SEC_L4_TEXT U32 OsTaskCreateOnly(TskHandle* taskPid,struct
TskInitParam* initParam) {
    // ...实验文档中提供的原代码...
    *taskPid = taskId;
    taskCb->timeSlice = OS_TSK_DEFAULT_TIMESLICE;
    OsIntRestore(intSave);
    return OS_OK;
}
```

2. 添加移动任务到空闲队列队尾的接口

在 `prt_task.c` 中实现一个将任务移动到就绪队列尾部的接口，把当前运行中的任务移动到队尾后再次调用优先级调度程序，会调用在队列相对头部的的同优先级任务，从而达到时间片轮转的效果。

```c
OS_SEC_L0_TEXT void OsTskMoveToBack(struct TagTskCb* task) {
    struct TagOsRunQue* rq = &g_runQueue;
    TSK_STATUS_CLEAR(task, OS_TSK_RUNNING);
    TSK_STATUS_SET(task, OS_TSK_READY);
    OS_TSK_DE_QUE(rq, task, 0);
    OS_TSK_EN_QUE(rq, task, 0);
    OsTskHighestSet();
    return;
}
```

将这个函数的声明添加在任务管理的接口文件 `prt_task_external.h` 中：

```c
extern void OsTskMoveToBack(struct TagTskCb *task);
```

3. 在中断处理函数中添加调度任务的逻辑

在处理中断的文件 `prt_tick.c` 中，添加对于时间片处理的逻辑，首先引用任务管理相关头文件，并且声明用到的外部函数 `OsGicIntClear` ：

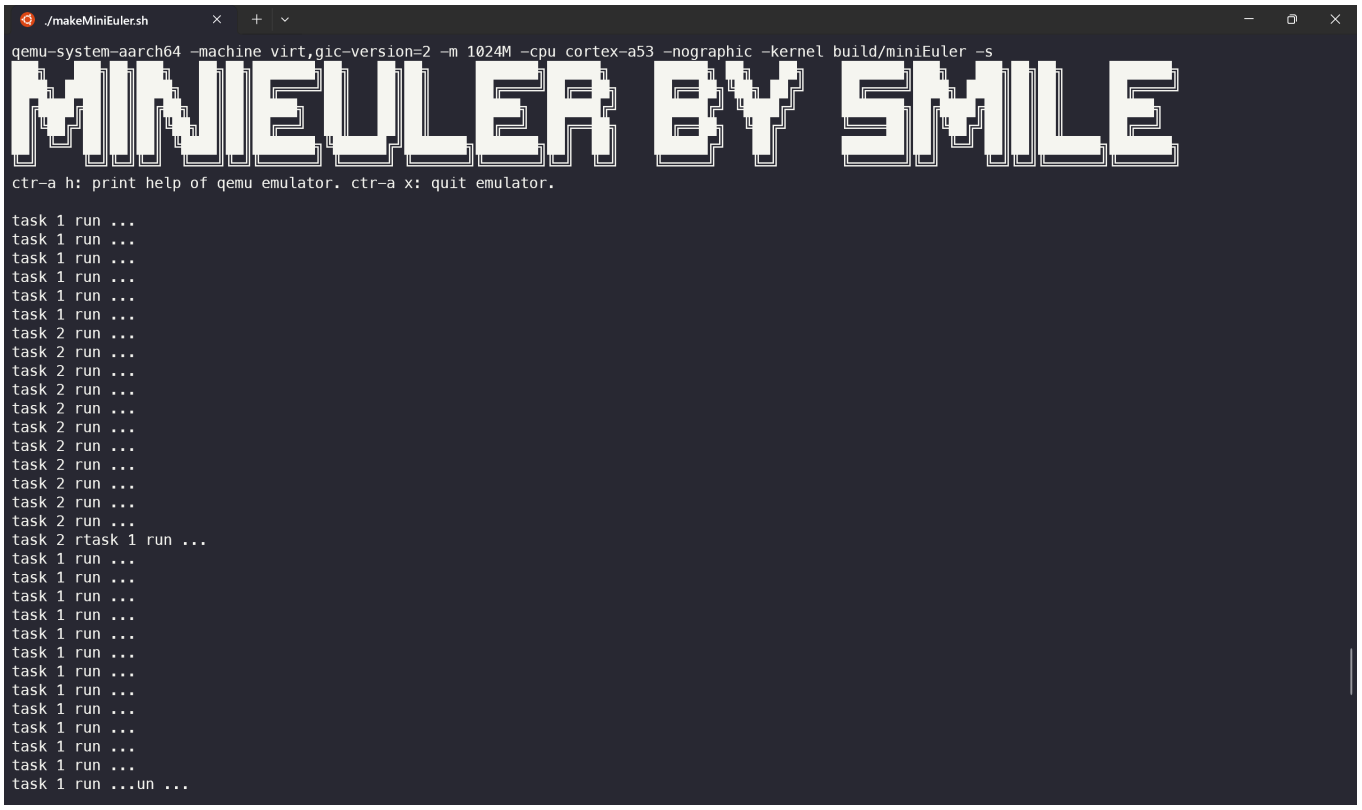
```c
#include "prt_task_external.h"
#include "prt_asm_cpu_external.h"

extern void OsGicIntClear(U32 value);
```

将`OsTickDispatcher`函数修改为如下：

```c
OS_SEC_TEXT void OsTickDispatcher(void) {
    uintptr_t intSave;
    intSave = OsIntLock();
    g_uniTicks++;
    OsIntRestore(intSave);
    U64 cycle = g_timerFrequency / OS_TICK_PER_SECOND;
    OS_EMBED_ASM("MSR CNTP_TVAL_EL0, %0" : : "r"(cycle) :
"memory",
                 "cc");  // 设置中断周期
    // 以下为新增的内容:
    // 调度任务前先禁用中断
    intSave = OsIntLock();
    if ((RUNNING_TASK != NULL) && (RUNNING_TASK->taskPid !=
IDLE_TASK_ID)){
        if (RUNNING_TASK->timeSlice > 0) {
            RUNNING_TASK->timeSlice--;
        }
        if (RUNNING_TASK->timeSlice == 0) {
            // 通知 GIC 本次中断已经完成，避免新任务运行时不触发中断
            OsGicIntClear(30);
            // 将旧任务的时间片复原
            RUNNING_TASK->timeSlice = OS_TSK_DEFAULT_TIMESLICE;
            // 将旧任务移动到队尾
            OsTskMoveToBack(RUNNING_TASK);
            // 申请调度任务
            OsTskSchedule();
        }
    }
    OsIntRestore(intSave);
}
```

结果如下图所示:



五、 心得体会

1. 对调度系统的工作原理有了更深刻的理解
2. 对分时调度系统的工作原理有进一步的理解
3. 更加熟悉调度系统的数据结构以及功能函数