《操作系统》

实验六报告

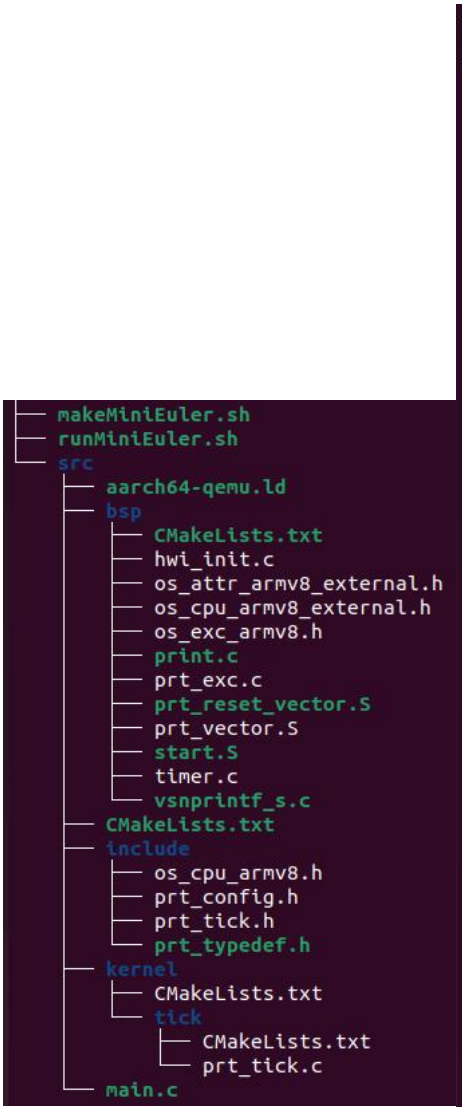
目录

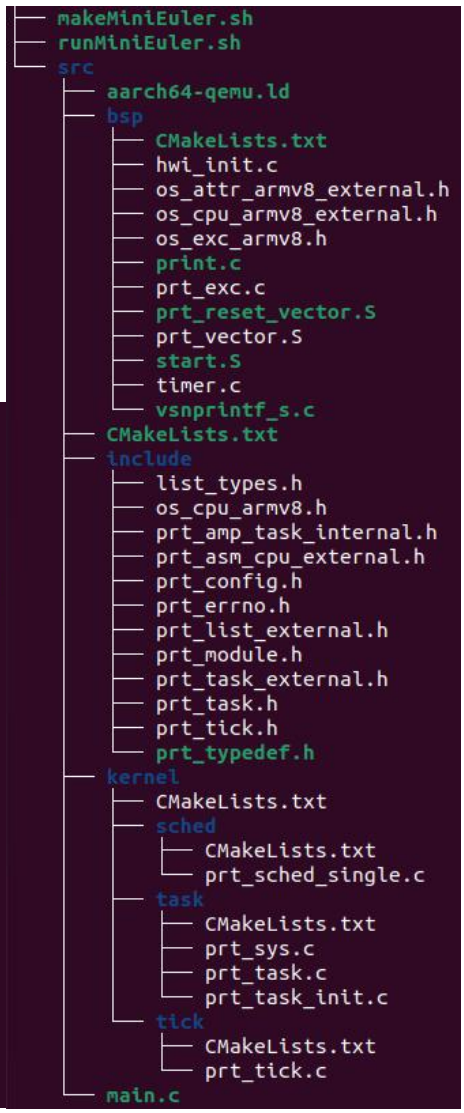
1 实验代码分析

1.1 实验目的

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。**本实验实现了任务调度。

1.2 代码结构





对应 lab5 的结构来看，在 include 文件夹下新增了 list_types.h、prt_list_external.h 用于链表定义，prt_module.h、prt_errno.h、prt_task.h、prt_task_external.h、prt_amp_task_internal.h 五个头文件与任务控制块相关，prt_asm_cpu_external.h 包含内核相关的一些状态定义。在 kernel 下新增文件夹 sched 和 task，任务创建的相关代码主要包含在 task 中，调度与切换的代码包含在 sched 中。修改了 src/include/os_cpu_armv8.h、prt_config.h 等和构建系统的多个文件。

1.3 实现双向链表

新建文件 src/include/list_types.h 和 src/include/prt_list_external.h，定义了链表结构和相关操作。

其中定义了几个宏以实现操作，LIST_COMPONENT 宏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；LIST_FOR_EAC 和 LIST_FOR_EACH_SAFE 用于遍历链表，主要是简化代码编写。

1.4 任务控制块/头文件引入

新建了五个与任务相关的头文件。

prt_module.h 中主要是一些模块 ID 的定义，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 中除有任务或任务控制块的相关宏的定义外，还定义了任务创建时参数传递的结构体： struct TskInitParam。包括入口函数、优先级、任务参数（最多四个）、栈的大小、任务名。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。

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**。包括当前任务的 SP、任务状态、运行优先级，栈配置标记、栈大小、栈顶，入口函数、信号量指针，任务参数、任务名、链表相关（信号量链表指针、任务延时链表指针、持有互斥信号量链表、记录条件变量的等待线程、等待队列指针），任务事件、任务事件掩码、任务记录的最后一个错误码、任务恢复时间点。简单起见，我们将任务运行队列结构 TagOsRunQue 直接定义为双向链表 TagListObject。

prt_amp_task_internal.h 中主要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。

1.5 任务创建

任务创建代码主要在 src/kernel/task/prt_task_init.c 中。

1.5.1 相关变量和函数声明

引入必要的头文件；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。

新建头文件 src/include/prt_asm_cpu_external.h，包含内核相关的一些状态定义。

1.5.2 极简内存空间管理

内核运行过程中需要动态分配内存。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。

1.5.3 任务栈初始化

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。但在任务第一次运行的时候无上下文，我们需要手动初始化。

struct TskContext 表示任务上下文，在 src/**include**/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。

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

注：这里实验指导书标注错误，os_cpu_armv8.h 文件在 include 中（可见于本报告 1.2 代码结构）。

1.5.4 任务入口函数

这个函数有几个有趣的地方。

（1）没有类似 OsTskEntry(taskId); 这样的对 OsTskEntry 的函数调用。**这是在通过**

OsTskContextInit 函数进行栈初始化时由参数的形式传入的（如下图），使得任务第一次就绪运行时会进入 OsTskEntry 执行。指定了函数入口。（建议这里指导书写详细一点，开始对这里理解错误导致后面完全弄混，在里面转圈转了一整天。）

```
12 /*
13 * 描述：初始化任务栈的上下文
14 */
15 void *OsTskContextInit(U32 taskID, U32 stackSize, uintptr_t *topStack, uintptr_t
    funcTskEntry)
16 {
17     (void)taskID;
```

（2）用户指定的 taskcb->taskEntry 不一定要求是 4 参数的，可以在 0~4 参数之间任意选定，需要从汇编层面去理解。

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。

OsTskEntry 是一个任务入口函数，它首先从全局变量中获取当前运行任务的 TCB，然后调用该任务的入口函数 taskCb->taskEntry。在任务执行完毕后，它确保中断状态被正确恢复，并通知操作系统任务已经完成。OsTskContextInit 函数负责这个任务栈的初始化工作，这个函数会设置栈的初始内容，包括任务的返回地址（Return Address），这个返回地址实际上是指向 OsTskEntry 函数的。当 RTOS 调度器切换到该任务时，它会根据栈中的这个返回地址跳转到 OsTskEntry 函数开始执行。

1.5.5 创建任务

该段函数实现了创建任务，分作不同的函数便于修改、阅读和维护。

我们从后往前看，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（任务句柄是在操作系统或嵌入式系统中用于唯一标识和管理任务的标识符）。

PRT_TaskCreate 函数会直接调用 OsTaskCreateOnly 函数进行实际的任务创建。

OsTaskCreateOnly 函数将：

1.通过 OsTaskCreateChkAndGetTcb 函数从空闲链表 g_tskCbFreeList 中取一个任务控制块；

2.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；

3.OsTskContextInit 函数负责将栈初始化成刚刚发生过中断一样；

4.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；

5.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**。

1.5.6 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 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。**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。

1.5.7 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。OsActivate 启动多任务系统。在 prt_config.h 中加入空闲任务优先级定义。

OsTskAMPInit 函数进行初始化时会报错：

```
24      // 初始化为全0
25      for(int i = 0; i < OS_MAX_TCB_NUM - 1; i++)
26          g_tskCbArray[i] = {0};
```

```
/home/guoruiling/lab6/src/kernel/task/prt_task_init.c: In function 'OsTskAMPInit':
/home/guoruiling/lab6/src/kernel/task/prt_task_init.c:354:27: error: expected expression bef
ore '{' token
  354 |         g_tskCbArray[i] = {0};
      |                           ^
gmake[2]: *** [kernel/task/CMakeFiles/task.dir/build.make:104: kernel/task/CMakeFiles/task.d
ir/prt_task_init.c.obj] Error 1
gmake[1]: *** [CMakeFiles/Makefile2:250: kernel/task/CMakeFiles/task.dir/all] Error 2
gmake: *** [Makefile:91: all] Error 2
```

查询后发现编译器不允许这种初始化方式，应使用 memset 或者强制类型转换，修改为如下方式后通过编译。

```
17      // 初始化为全0
16      for(int i = 0; i < OS_MAX_TCB_NUM - 1; i++)
15          g_tskCbArray[i] = (struct TagTskCb){0};
```

1.6 任务状态转换

新建 src/kernel/task/prt_task.c，其中声明了运行队列 g_runQueue，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。提供了任务结束退出的 OsTaskExit 函数，注意 OsTskEntry 中会调用 OsTaskExit 函数。**由于任务退出，需要进行调度，存在调度点，所以调用 OsTskSchedule 函数。**

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

1.7 调度与切换

新建 src/kernel/sched/prt_sched_single.c 文件，包括我们之前调用过很多次的 OsTskSchedule 函数，它实现了任务调度，切换到最高优先级任务；OsMainSchedule 是调度的主入口；OsFirstTimeSwitch 定义了系统启动时的首次任务调度。

OsTskHighestSet 函数查找队列中优先级最高的任务，它在 src/include/prt_task_external.h 中，被定义为了内联函数，可提高性能。

src/bsp/prt_vector.S 实现了 OsTskContextLoad，OsContextLoad 和 OsTaskTrap（调度处理函数）。

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的宏。

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

OsTskSchedule 函数，切换到最高优先级调度，调用 OsTaskTrap();切换，OsTaskTrap() 是调度处理函数，定义在 prt_vector.S 中，它调度了 OsMainSchedule，OsMainSchedule 是调度的主入口。

1.8 任务调度测试

修改 main 函数，新建了 task1 和 task2，进行任务调度测试。

修改，将新建文件加入构建系统。

根据实验五的构建系统规则进行构建（可见于本人实验五报告，此处附截图）。

include_directories()：这个命令告诉 CMake 在编译时应该在哪些目录中寻找头文件。

add_subdirectory()：这个命令告诉 CMake 在指定的子目录中查找另一个 CMakeLists.txt 文件，并处理其中的构建规则。这允许将项目分解为多个子目录，每个子目录都有自己的构建规则。例如，add_subdirectory(bsp)会告诉 CMake 在 bsp 子目录中查找 CMakeLists.txt 文件，并处理该文件中定义的任何目标（如库或可执行文件）、源文件、头文件目录等。

add_library()：这个命令添加一个库目标到构建系统中。命令的 OBJECT 类型是一个特殊的库类型，它允许将源文件（.c、.cpp 等）编译为对象文件（.o），但不将它们链接成一个库。这些对象文件可以在其他目标或库中被链接，用于在多个目标之间共享编译后的对象文件，提高编译效率。

list(APPEND obj ...) :这个命令将多个元素追加到名为 obj 的列表中。

我们将头文件 include 在总 CM 文件中包含进入编译目录 include_directories()，无需在 include 文件夹中使用 CM 文件；对于其他源文件，每个中都有一个 CM 文件。如果是底层目录，需要用 set 设置源文件列表并用 add_library 生成.o 目标文件并在总 CM 文件 list 中指定该目标，如果非底层目录，使用 add_subdirectory()在指定子目录中查找 CM 文件，并处理其中的构建规则。若在子目录中包含头文件，也需要被包含在 include_directories()命令中。

```
├── makeMiniEuler.sh
├── runMiniEuler.sh
└── src
    ├── aarch64-qemu.ld
    ├── bsp
    │   ├── CMakeLists.txt
    │   ├── hwi_init.c
    │   ├── os_attr_armv8_external.h
    │   ├── os_cpu_armv8_external.h
    │   ├── os_exc_armv8.h
    │   ├── print.c
    │   ├── prt_exc.c
    │   ├── prt_reset_vector.S
    │   ├── prt_vector.S
    │   ├── start.S
    │   ├── timer.c
    │   └── vsnprintf_s.c
    ├── CMakeLists.txt
    ├── include
    │   ├── list_types.h
    │   ├── os_cpu_armv8.h
    │   ├── prt_amp_task_internal.h
    │   ├── prt_asm_cpu_external.h
    │   ├── prt_config.h
    │   ├── prt_errno.h
    │   ├── prt_list_external.h
    │   ├── prt_module.h
    │   ├── prt_task_external.h
    │   ├── prt_task.h
    │   ├── prt_tick.h
    │   └── prt_typedef.h
    ├── kernel
    │   ├── CMakeLists.txt
    │   ├── sched
    │   │   ├── CMakeLists.txt
    │   │   └── prt_sched_single.c
    │   ├── task
    │   │   ├── CMakeLists.txt
    │   │   ├── prt_sys.c
    │   │   ├── prt_task.c
    │   │   └── prt_task_init.c
    │   └── tick
    │       ├── CMakeLists.txt
    │       └── prt_tick.c
    └── main.c
```

总 src/CMakeLists.txt：

```
include_directories(
    ${CMAKE_CURRENT_SOURCE_DIR}/include   # 增加 src/include 目录
    ${CMAKE_CURRENT_SOURCE_DIR}/bsp
)

add_subdirectory(bsp)
add_subdirectory(kernel) # 增加 kernel 子目录

list(APPEND OBJS $<TARGET_OBJECTS:bsp> $<TARGET_OBJECTS:tick> $<TARGET_OBJECTS:task> $<TARGET_OBJECTS:sched>) # 增加 $<TARGET_OBJECTS:tick> 目标
add_executable(${APP} main.c ${OBJS})
```

src/kernel/CMakeLists.txt：

```
add_subdirectory(tick)
add_subdirectory(task)
add_subdirectory(sched)
```

src/kernel/task/CMakeLists.txt：

```
set(SRCS prt_task.c prt_sys.c prt_task_init.c )
add_library(task OBJECT ${SRCS})  # OBJECT类型只编译生成.o目标文件，但不实际链接成库
```

src/kernel/sched/CMakeLists.txt：

```
set(SRCS prt_sched_single.c )
add_library(sched OBJECT ${SRCS})  # OBJECT类型只编译生成.o目标文件，但不实际链接成库
```

bsp 中无新增文件，CMakeLists.txt 无修改。

1.9 构建项目与执行结果

本实验逻辑为：

src/include/list_types.h 和 src/include/prt_list_external.h，定义链表结构和相关操作。

prt_module.h、prt_errno.h、prt_task.h、prt_task_external.h、prt_amp_task_internal.h 五个头文件与任务控制块相关。prt_module.h 中主要是一些模块 ID 的定义；prt_errno.h 主要是错误类型的相关定义；prt_task.h 中有一些相关宏定义和任务创建时参数传递的结构体；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，其中也包含 OsTskHighestSet 函数，将在 prt_sched_single.c 函数中被使用；prt_amp_task_internal.h 定义了三个内联函数和 prt_task.c 中使用的宏，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。

src/kernel/task 中包含任务创建的相关代码，prt_task_init.c 是主要部分，包括相关变量和函数声明、内存空间管理、任务栈初始化上下文（在 src/bsp/os_cpu_armv8.h 中补充了 struct TskContext 定义）、任务入口函数、创建任务、解挂任务、任务管理系统初始化与启动；prt_task.c 中包括运行队列、添加与移除队列、任务退出函数；prt_sys.c 定义了内核的各种全局数据。

src/kernel/sched 中 prt_sched_single.c 包含调度与切换的代码。

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

修改新增构建系统。

这里 task1 的 param.taskPrio 设置为 35，task2 的设置为 30，task2 先运行，运行结束后 task1 运行。通过任务结束退出函数 OsTaskExit 调用 OsTskSchedule 实现切换。

```
guoruiling@guoruiling-virtual-machine:~/lab6$ sh makeMiniEuler.sh
mkdir: cannot create directory ‘build’: File exists
-- The C compiler identification is GNU 11.2.1
-- The ASM compiler identification is GNU
-- Found assembler: /home/guoruiling/aarch64-none-elf/gcc-arm-11.2-2022.02-x86_64-aarch64-no
ne-elf/bin/aarch64-none-elf-gcc
-- Detecting C compiler ABI info
-- Detecting C compiler ABI info - done
-- Check for working C compiler: /home/guoruiling/aarch64-none-elf/gcc-arm-11.2-2022.02-x86_
64-aarch64-none-elf/bin/aarch64-none-elf-gcc - skipped
-- Detecting C compile features
-- Detecting C compile features - done
-- Configuring done
-- Generating done
-- Build files have been written to: /home/guoruiling/lab6/build
[  6%] Building C object kernel/sched/CMakeFiles/sched.dir/prt_sched_single.c.obj
[  6%] Built target sched
Scanning dependencies of target bsp
[ 13%] Building ASM object bsp/CMakeFiles/bsp.dir/start.S.obj
[ 20%] Building ASM object bsp/CMakeFiles/bsp.dir/prt_reset_vector.S.obj
[ 26%] Building C object bsp/CMakeFiles/bsp.dir/print.c.obj
[ 33%] Building C object bsp/CMakeFiles/bsp.dir/vsnprintf_s.c.obj
[ 40%] Building C object bsp/CMakeFiles/bsp.dir/prt_exc.c.obj
[ 46%] Building ASM object bsp/CMakeFiles/bsp.dir/prt_vector.S.obj
[ 53%] Building C object bsp/CMakeFiles/bsp.dir/hwi_init.c.obj
[ 60%] Building C object bsp/CMakeFiles/bsp.dir/timer.c.obj
[ 60%] Built target bsp
[ 66%] Building C object kernel/tick/CMakeFiles/tick.dir/prt_tick.c.obj
[ 66%] Built target tick
[ 73%] Building C object kernel/task/CMakeFiles/task.dir/prt_task.c.obj
[ 80%] Building C object kernel/task/CMakeFiles/task.dir/prt_sys.c.obj
[ 86%] Building C object kernel/task/CMakeFiles/task.dir/prt_task_init.c.obj
[ 86%] Built target task
[ 93%] Building C object CMakeFiles/miniEuler.dir/main.c.obj
[100%] Linking C executable miniEuler
[100%] Built target miniEuler
```

```
guoruiling@guoruiling-virtual-machine:~/lab6$ sh runMiniEuler.sh
qemu-system-aarch64 -machine virt,gic-version=2 -m 1024M -cpu cortex-a53 -nographic -kernel
build/miniEuler -s
------------ ------------    ------    -----------
 ************ ************   ********   ***********
----         ------------  ----------  ----    ---
************     ****     ****    **** *********
 ------------     ----     ------------ ---------
       *****     ****     ************ ****  ****
 ------------     ----     ----    ---- ----   ----
************     ****     ****    **** ****    ****
ctr-a h: print help of qemu emulator. ctr-a x: quit emulator.

task 2 run ...
task 2 run ...
task 2 run ...
task 2 run ...
task 2 run ...
task 2 run ...
task 1 run ...
task 1 run ...
task 1 run ...
task 1 run ...
task 1 run ...
task 1 run ...
```

2 实验任务

2.1 作业 1

题目：实现分时调度。分时调度的调度点存在于时钟 Tick 中断、任务结束等处。

先前在任务结束退出的 OsTaskExit 函数中已经添加了任务调度 OsTskSchedule()，这里无需修改。我们需要实现的是在时钟 Tick 中断处的分时调度，在这里我们实现轮转调度，以时间片为周期进行调度。

调度点：

1. 新任务进行初始化工作结束后，进行解挂任务时，通过调用 OsMoveTaskToReady 函数调用 OsTskSchedule()进行调度。**新的任务就绪为调度点之一（实现抢占式调度）。**

2. 任务结束时，直接调用 OsTskSchedule()调度。**任务结束为调度点之一。**

3. **时钟中断为调度点之一（本任务中）。原实验是：抢占式调度机制。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。main 函数在高优先级的 task2 结束后使用任务结束的调度点进入 task1.**

目标实现效果：

```
guoruiling@guoruiling-virtual-machine:~/lab6p2$ sh runMiniEuler.sh
qemu-system-aarch64 -machine virt,gic-version=2 -m 1024M -cpu cortex-a53 -nographic -kernel build
/miniEuler -s
------------ ------------    ------    -----------
 ************ ************   ********   ************
----         ------------  -----------  ----     ---
************     ****     ****    **** *********
 ------------     ----     ------------ ---------
       *****     ****     ************ ****  ****
 ------------     ----     ----    ---- ----   ----
************     ****     ****    **** ****    ****
ctr-a h: print help of qemu emulator. ctr-a x: quit emulator.

task 2 run ...
task 1 run ...
task 2 run ...
task 1 run ...
task 2 run ...
task 1 run ...
task 2 run ...
task 1 run ...
task 2 run ...
task 1 run ...
task 2 run ...
task 1 run ...
```

首先回顾 main 函数执行的主要过程，首先对 GIC 进行初始化，启用中断，对任务系统进行初始化；然后对任务 1、2 分别进行各种参数的设置，调用 PRT_TaskCreate()创建任务

并用 PRT_TaskResume 解挂；最终使用 OsActivate()函数启用调度。

OsActivate()是任务管理激活函数，它首先创建空闲任务以备没有任务就绪时运行，然后查找最高优先级任务并将 g_highestTask 指针指向它，设置好内核状态，调用 OsFirstTimeSwitch 函数加载 g_highestTask 的上下文并执行。OsFirstTimeSwitch 函数用于首次任务切换，它将最高优先级的任务设置为当前运行的任务，并设置该任务的运行状态，调用 OsTskContextLoad 函数加载上下文切换到该函数执行。

当中断时，会跳转到中断向量表，根据向量表中设置的异常处理函数进行处理，实现不同任务之间的切换（实验五）。我们需要修改异常处理函数以实现分时调度。

实验五中，修改后向量表将 IRQ 类型的异常（中断）指向 OsHwiDispatcher 函数处理，它调用了 OsHwiDispatch 函数，OsHwiDispatch 函数调用 OsHwiHandleActive 函数，OsHwiHandleActive 函数调用 OsTickDispatcher()函数，最终由 OsTickDispatcher()进行具体的中断处理。

- 将 prt_vector.S 中的 EXC_HANDLE 5 OsExcDispatch 改为 EXC_HANDLE 5 OsHwiDispatcher，表明我们将对 IRQ 类型的异常（即中断）使用 OsHwiDispatcher 处理。

OsHwiDispatcher 定义了一个中断或异常的分发器，它保存了异常发生时的上下文，调用了处理函数 OsHwiDispatch，并在处理完成后恢复了上下文，然后从异常返回。

对 OsHwiDispatch 的解释给出，它是中断处理的入口，调用处的外部已经关闭了中断，它首先使用 OsGicIntAcknowledge() 确认中断得到中断号，然后调用 OsHwiHandleActive 执行，最后使用 OsGicIntClear(irq_num|core_num) 清除中断。

```
20  OS_SEC_L0_TEXT void OsHwiDispatch( U32 excType, struct ExcRegInfo *e)
21  {
22      // 中断确认，相当于OsHwiNumGet()
23      U32 value = OsGicIntAcknowledge();
24      U32 irq_num = value & 0x1ff;
25      U32 core_num = value & 0xe00;
26
27      OsHwiHandleActive(irq_num);
28
29      // 清除中断，相当于 OsHwiClear(hwiNum);
30      OsGicIntClear(irq_num|core_num);
31  }
```

OsHwiHandleActive() 函数确认中断号为 30，再调用 OsTickDispatcher()函数。

```
OS_SEC_ALW_INLINE INLINE void OsHwiHandleActive(U32 irqNum)
{
    switch(irqNum){
        case 30:
            OsTickDispatcher();
            // PRT_Printf(".");
            break;
        default:
            break;
    }
}
```

OsTickDispatcher()：调用 OsIntLock() 函数关闭中断，并将返回的值存储在 intSave 中。增加全局变量 g_uniTicks 的值。调用 OsIntRestore() 恢复先前的中断状态。计算定时器每秒应该产生的中断次数，并使用一个内嵌的汇编指令来设置定时器的计数值，以便在适当的时候产生下一个中断。这里的 g_timerFrequency（从 CNTFRQ_EL0 读取的时钟频率）被除以 OS_TICK_PER_SECOND（操作系统每秒的节拍数）。结果是一个无符号 64 位整数 cycle，它表示每个操作系统节拍对应的时钟周期数。

```
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", "c

}
```

为了实现时间片轮转调度，我们需要修改时钟中断的部分（对中断计数）和调度的部分。

首先我们要知道，应在清除中断后再调度任务，否则后续时钟中断不起作用。当我们提到“完成中断再进行调度”时，这通常是指在中断处理完成并返回到原来的任务或主程序之前，系统会检查是否需要进行任务调度。

“清除中断”：

代表中断处理完成。也就是 OsHwiDispatch 中嵌套调用执行完毕，回到了 OsGicIntClear(irq_num|core_num) 的这一行，原代码在执行完这一行后结束。

“清除中断后再调度任务，否则后续时钟中断不起作用”：

有多条理由。1.**上下文保存与恢复**：当中断发生时，系统需要保存当前任务的上下文，以便在中断处理完成后能够恢复到正确的执行状态。如果在中断处理过程中直接进行任务调度，可能会导致上下文信息丢失或混淆，使得系统无法正确恢复之前任务的执行状态。2.**中断处理原子性**：某些中断处理可能需要一系列的操作来确保数据的完整性和一致性。如果在中断处理过程中进行任务调度，可能会打断这些操作，导致数据损坏或系统状态不一致。3.**优先级管理**：中断通常具有不同的优先级，用于区分紧急程度不同的外部事件。如果在中断处理过程中直接进行任务调度，可能会导致高优先级的中断被低优先级的任务阻塞，无法得到及时处理。

所以我们需要在 OsGicIntClear(irq_num|core_num) 的这一行代码后加入调度处理函数。

```
12 OS_SEC_L0_TEXT void OsHwiDispatch( U32 excType, struct ExcRegInfo *excRegs)
   mv8/common/hwi/prt_hwi.c
13 {
14     // 中断确认，相当于OsHwiNumGet()
15     U32 value = OsGicIntAcknowledge();
16     U32 irq_num = value & 0x1ff;
17     U32 core_num = value & 0xe00;
18
19     OsHwiHandleActive(irq_num);
20
21     // 清除中断，相当于 OsHwiClear(hwiNum);
22     OsGicIntClear(irq_num|core_num);
23
24     switch(irq_num){
25         case 30:
26                 OsTskSchedule();
27                 break;
28
29                 break;
30         }
31 }
```

设置时间片。

```
5 extern OS_SEC_L2_TEXT U64 PRT_getCycle(void);
6 extern U64 g_timerFrequency;
7
8 U64 TIME_SLICE = 0;
9 /*
10 * 描述：任务调度，切换到最高优先级任务
```

在中断的时候，修改时间片，每发生一次中断，就减小它，直到为零。

```
OS_SEC_TEXT void OsTickDispatcher(void)
{
    uintptr_t intSave;

    intSave = OsIntLock();
    volatile U64 cycle = PRT_getCycle();//+
    g_uniTicks++;
    if(TIME_SLICING !=0)    TIME_SLICING -= cycle;//+
    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

}

/*
* 描述：获取当前的tick计数
*/
OS_SEC_L2_TEXT U64 PRT_TickGetCount(void) //src/core/kernel/sys/prt_sys_time.c
{
    return g_uniTicks;
}

OS_SEC_L4_TEXT U64 PRT_getCycle(void)
{
    return g_timerFrequency;
}
```

调度时（调度策略）：若时间片为 0，重置时间片并切换至下一任务。

```
OS_SEC_ALW_INLINE INLINE void OsTskRR_Set()
{
                // 如果当前任务时间片耗尽且仍在就绪队列中（未退出）
        if(TIME_SLICING == 0 && TSK_STATUS_TST(g_RR_currentTask,OS_TSK_READY)){
                //切换到队列中下一任务
        g_RR_currentTask = LIST_NEXT(g_RR_currentTask, struct TagTskCb,pendList);
                // 如果是链表哑节点（g_runQueue），继续切换
        if(&(g_RR_currentTask->pendList) == &g_runQueue)
        LIST_NEXT(g_RR_currentTask, struct TagTskCb,pendList);
        TIME_SLICING = N*PRT_getCycle();
        return;
        }
}
```

```
OS_SEC_TEXT void OsTskSchedule(void)
{
    /* 外层已经关中断 */
    /* Find the highest task */
    OsTskRR_set();
    OsTskHighestSet();

    /* In case that running is not highest then reschedule */
    if ((g_highestTask != RUNNING_TASK) && (g_uniTaskLock == 0)) {
        UNI_FLAG |= OS_FLG_TSK_REQ;

        /* only if there is not HWI or TICK the trap */
        if (OS_INT_INACTIVE) { // 不在中断上下文中，否则应该在中断返回时切换
            OsTaskTrap();
            return;
        }
    }

    return;
}
```

思路是，由 OsHwiDispatcher 函数进入中断处理部分，在每次的中断处理 OsTickDispatcher()中削减时间片，结束中断处理，回到 OsHwiDispatch 函数，进入调度部分，若时间片为 0，重置时间片并切换至下一任务。回到任务执行。

需要进行时间片初始化。

```
18 * 描述：任务初始化
19 */
20 OS_SEC_L4_TEXT U32 OsTskInit(void)
21 {
22     TIME_SLICE=3*PRT_getCycle();
23     U32 ret;
24     ret = OsTskAMPInit();
25     if (ret != OS_OK) {
26         return ret;
27     }
28
29     return OS_OK;
30 }
```

参考了讨论区中廖志豪同学发布的思路，感恩。

（老师这个作业要自己写出来真的真的太难了，建议下一届在实验指导书给点提示吧）