任务 1：多 tasks 启动与 context switch

中国科学院大学 操作系统研讨课

2017.09.27

1. 介绍

本次作业主要目的是了解多 *task*(指代进程或者线程)在内核中的实现以及运行方式。这里 *task* 的调度采用非抢占式式调度。

本次任务主要分为两大块：实现一个非抢占式调度器，以及实现 *task* 启动与 context switch。做此开发之前，需要完善作业一的 createimage.

1.1. 需要了解的部分

- 非抢占式调度
- context switch
- *task*运行需要维护的信息
- 进程和线程
- *task*执行有关的寄存器，以及这些寄存器的功能

1.2. 注意

本任务中的进程、线程和传统意义上的进程线程存在差别。本作业中所提到的进程和线程都是本次定义的进程和线程，请和传统的分别开来。下面对此进行介绍。

进程：虽然有自己的保护域，但是并没有独立的地址空间，即所有进程和线程平分一块地址空间，实现的时候需要规划每个进程（线程）自己的地址范围。

线程：本次作业中的线程指的是内核线程，和内核分享相同的地址空间，并只在特权模式下执行。内核线程是内核的一部分，可以直接访问内核的数据、调

用内核函数，进程只有通过系统调用才可以达到此目的。

2. 初始代码

2.1. 文件介绍

- Makefile: 编译文件。
- bootblock.s: 内核启动程序，**请使用作业一中自己写的代码**。
- Createimage.c: 生成内核镜像的Linux工具，**请使用作业一中自己写的代码**。
- entry_mips.S: 代码中*task* context switch时需要进行的操作，在本次任务中需要完成scheduler_entry和save_pcb。本文件也提供了获取时间函数。
- kernel.c: 内核最先执行的文件，放在内核的起始处。在本次任务中需要完成多tasks的初始化
- lock.c: 实现一个互斥锁，本次任务中不用修改。
- scheduler.c: 调度器，实现*task*的调度，本次任务中需要完成多个tasks轮流运行，以及yield和exit操作。
- syslib.S: 系统调用函数，进程通过系统调用进入内核，本次任务需要了解。
- queue.c: 队列操作函数，在本次任务中，如果非抢占调度器用队列实现，可以直接使用该文件内函数。
- task.c: 将本次任务中所有*task*通过一个task_info的结构体引用。初始化*task*时就可以通过该文件提供的task数组，做本次任务时task数组的值不需要改变。完成不同任务时，请给task数组赋不同的task值，用于不同的测试目的。文件中的注释部分表明每一个任务需要测试的进程或者线程（将注释部分去掉就可以直接使用）。
- process1/2/3.c：定义了测试用例进程，可以分别用于不同的任务。
- th1/2/3/4.c：定义了测试用例内核线程，可以分别用于不同的任务。
- util.c: 提供了一些print函数，可以用于调试以及显示信息。请在本任务开始前了解该文件。

- *.h: 相应.c/.S文件的头文件。

2.2. 获取：

课程网站。

2.3. 运行

createimage 为提供的可执行文件，当 createimage.c 实现完成后，将 Makefile 中的 createimage 项去掉注释。

make 命令编译文件

make clean 对编译产生的文件进行清除

sudo dd if=image of=/dev/sdb 将产生的 image 写进 SD 卡中

在 minicom 中执行 *loadboot* 运行程序

2.4. 注意

在 Project 1 中内核就指的是 kernel.c 编译后生成的 kernel 文件，但是在 Project 2 中，kernel 文件（内核）是由若干个文件编译连接生成的，kernel.c 只是存放在 kernel 文件开始的地方，即内核运行时，kernel.c 的内容最先执行。Process1, Process2, Process3 是不和 kernel 一起编译的，但是需要 createimage 将其也写入到 image 中。最终 bootblock 只需要读一次 SD 卡(image)，将 kernel 以及所有的进程线程都加载到内存中。以 Process1 为例, Makefile 的编译选项中指出 Process1 的-Ttext 为 0xffffffffa0810000, createimage 就需要根据此信息将 Process1 写到指定的位置，使得 Process1 加载到内存的位置为 0xa0810000(这里为 32 位寻址)。

例如 kernel 被加载到内存的 0xa0800200 处，在 SD 卡的第二个扇区。而 Process1 需要加载的地址为 0xa081000,与 kernel 相差的空间为 0xFE00 字节，在 SD 卡上相差的扇区数即为 127，所以 Process1 就需要写在 SD 卡的第 129 个扇区处，即需要在 image 中填充数据，使得 Process1 在 image 的第 128 个扇区处。

可以自行修改 Process 的位置，但需要同时修改 Makefile 和 task.c，因为 Makefile 指定 Process 要被存放的位置（这里 createimage 使用），task.c 则告诉

操作系统进程在内存中的位置（调用进程时使用），该进程执行时就从这个地址执行。

3. 任务

3.1. 设计和评审

帮助学生发现设计的错误，及时完成任务。学生需要对这次的作业进行全面考虑，在实现代码之前有清晰的思路。学生讲解设计思路时可以用不同的形式，如伪代码、流程图等，建议使用 PPT。

3.1.1. 设计问题考虑

- 进程控制块（PCB），PCB里面存储什么信息，以及tasks启动时初始化那些内容？
- *task* context switch：怎样保存以及加载*task*的上下文？对于系统运行的第一个*task*有什么不同的地方吗？
- 进程和线程在本作业中应该怎样区别对待？

3.2. 开发

这次任务主要有两个部分，一部分为 *task* 启动，另一部分为 *task* context switch。

3.2.1. 要求

- **设计进程控制块(PCB, kernel.h)**

该结构体记录 *task* 当前的状态，当 *task* 切换时，将 *task* 当前的状态(寄存器值)记录到 PCB 中，当 *task* 开始重新执行时，通过 PCB 得到 *task* 最后执行的信息，还原 *task* 的执行现场。

- **设计栈的位置(kernel.h)**

设置栈的最大、最小值以及每个 task 的栈大小

- **初始化*task*调度器以及PCB块(kernel.c)**

task 调度器可以采用队列的方式，例如先进先出。PCB 初始化需要设置 *task* 栈、*task* 起始执行地址等。注意：本任务中需要初始化多个 tasks。

- **实现*task*启动与context switch(entry.S)**

scheduler_entry(entry.S)需要调用 scheduler 选取下一步要执行的 *task*，再通过被选取 *task* 的 PCB 信息还原该 *task* 执行（即将 PCB 的内容加载到寄存器中）。save_pcb 需要保存 *task* 最后执行信息到 PCB（即将程序执行的有关寄存器的内容存储在 PCB 中）。

- **实现*task*调度**

do_yield(scheduler.c)保存正在执行的 *task* 信息，找到下一个需要执行的 *task*，并加载该 *task*。do_exit 将当前 *task* 标志为结束，找到下一个需要执行的 *task* 并加载。

3.2.2.注意事项

内核可以有多个内核线程，内核线程都运行在特权级别，它的代码是内核的一部分，和内核公用地址空间。但是每一个内核线程之间不共享寄存器和栈。

进程可以运行在两个地方：用户态和内核。对于进程的栈就有两种实现方式：一种为在用户态和内核态都建立栈，即在用户态运行的时候使用用户态栈，在内核中运行的时候使用内核栈；另一种只建立用户栈，因为进程在内核态运行的数据并不需要进行保存。

内核启动时首先执行 kernel.c 中的_stat()函数，该函数对内核运行进行一些初始化。每个 *task* 的 PCB 结构都在该函数中进行初始化，建议 PCB 结构体空间分配采用静态的方式，即直接声明 pcb_t pcbs[NUM_TASKS], NUM_TASKS 为此次运行的 *task* 的总个数，在 tasks.c 中定义。

栈设置时可以分配一块连续的地址空间，每个 *task* 分配固定大小，可以使用 common.h:ASSERT([assertion])函数来确保所有 *task* 的栈空间在 STACK_MIN 和 STAKC_MAX 之间，每个 *task* 栈大小为 STACK_SIZE,这些参数可以在 kernel.h 中进行设置。

本次作业为非抢占调度，所以 *task* 在执行中间会自己调用调度函数。

进程执行时需要执行内核中的函数，访问内核的数据，但是进程并不是内核的一部分，不能直接进行访问。所以本次作业中定义了一个 ENTRY_POINT, 该变

量的值即为内核中系统调用处理函数的地址，进程系统调用时通过该值来访问内核对应的系统调用处理函数（可以参考 syslib.S: kernel_entry 函数）。

线程的 context switch：

do_yield (scheduler.c)->scheduler_entry(entry.S)->scheduler(scheduler.c)

do_exit(scheduler.c)->scheduler_entry(entry.S)->scheduler(scheduler.c)

进程的 context switch：

yield(syslib.S) -> kernel_entry(entry.S) -> do_yield(scheduler.c)

exit (syslib.S)-> kernel_entry(entry.S) -> do_exit(Scheduler.c)

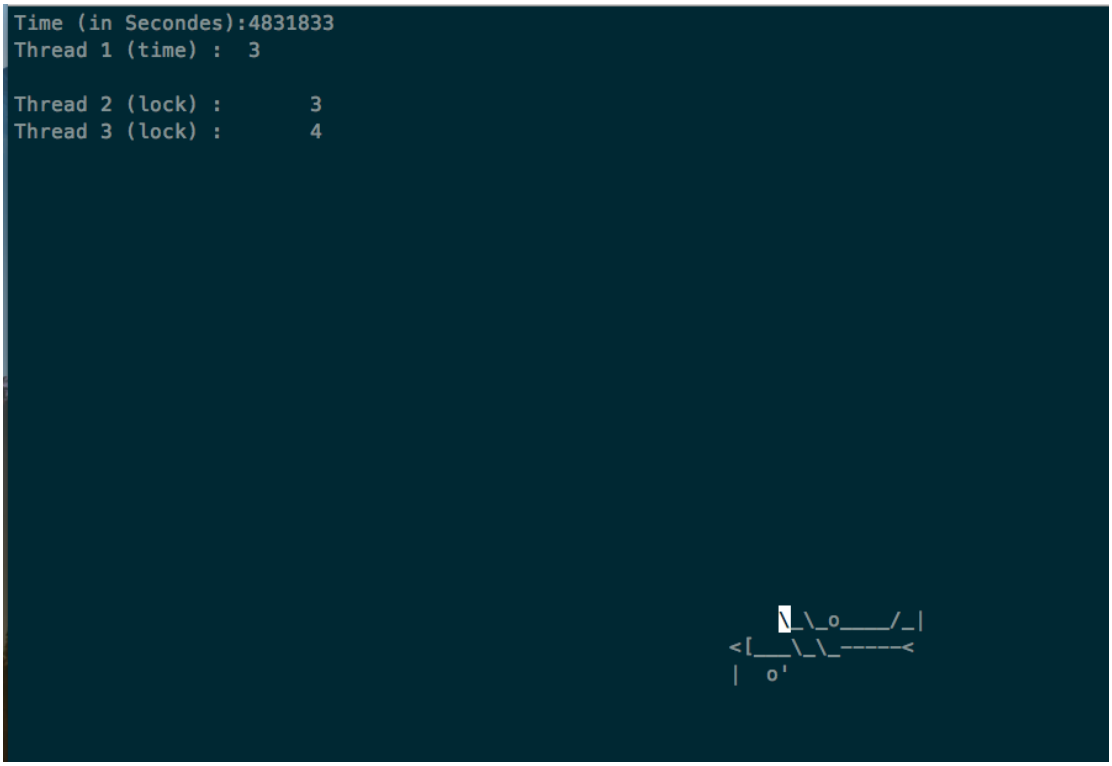
3.2.3.代码实现中的问题

本次作业中有以下几个地方可能会因为代码实现方式或者编译环境发生改变，如遇到请自即调整：

- bootblock.S中kernel_main的值为内核开始执行第一条指令的地址，请确保此处一致。
- syslib.S中的宏定义SYSCALL跳转的地址为kernel_entry函数的地址，请确保一致。
- 本系统输出为串口输出，可能会因为minicom窗口大小不同显示结果不同（代码的输出位置），这里可以忽略或者自己调整和设计
- do_yield()和block()函数都需要首先执行save_pcb函数,在设计save_pcb时需要考虑这两个函数可能对sp指针的不同影响。

4.测试

本次任务成功完成后，可以看到一架飞机在屏幕上不断移动，Thread 1,2,3 和 Process 1 在交替执行。如下图：


Time (in Secondes):4831833
Thread 1 (time) : 3
Thread 2 (lock) : 3
Thread 3 (lock) : 4