实验 3：RISC-V64简单的进程调度实现

1. 实验简介

- 结合课堂所学习的相关内容，在上一实验实现中断的基础上进一步地实现简单的进程调度

2. 实验环境

- Docker Image

3. 背景知识

3.1 什么是进程

源代码经编译器一系列处理（编译、链接、优化等）后得到的可执行文件，我们称之为程序（Program）。而通俗地说，进程（Process）就是正在运行并使用计算机资源的程序。进程与程序的不同之处在于，进程是一个动态的概念，其不仅需要将其运行的程序的代码/数据等加载到内存空间中，还需要拥有自己的运行栈。

3.2 进程的表示

在不同的操作系统中，为每个进程所保存的信息都不同。在这里，我们提供一种基础的实现，每个进程会包括：

- 进程ID：用于唯一确认一个进程。
- 运行时间片：为每个进程分配的运行时间。
- 优先级：在调度时，配合调度算法，来选出下一个执行的进程。
- 运行栈：每个进程都必须有一个独立的运行栈，保存运行时的数据。
- 执行上下文：当进程不在执行状态时，我们需要保存其上下文（其实就是状态寄存器的值），这样之后才能够将其恢复，继续运行。

3.3 进程调度与切换的过程

- 在每次时钟中断处理时，操作系统首先会将当前进程的运行剩余时间减少一个单位。之后根据调度算法来确定是继续运行还是调度其他进程来执行。
- 在进程调度时，操作系统会对所有可运行的进程进行判断，按照一定规则选出下一个执行的进程。如果没有符合条件的进程，则会对所有进程的优先级和运行剩余时间相关属性进行更新，再重新选择。最终将选择得到的进程与当前进程切换。
- 在切换的过程中，首先我们需要保存当前进程的执行上下文，再将将要执行进程的上下文载入到相关寄存器中，至此我们完成了进程的调度与切换。

4. 实验步骤

4.1 环境搭建

4.1.1 建立映射

同lab2的文件夹映射方法，目录名为lab3。

4.1.2 组织文件结构

```
lab3
├── arch
│   └── riscv
│       ├── include
│       │   ├── put.h
│       │   └── sched.h
│       ├── kernel
│       │   ├── entry.S
│       │   ├── head.S
│       │   ├── Makefile
│       │   ├── sched.c
│       │   ├── strap.c
│       │   └── vmlinux.lds
│       └── Makefile
├── include
│   ├── put.h
│   ├── rand.h
│   └── test.h
├── init
│   ├── main.c
│   ├── Makefile
│   └── test.c
├── lib
│   ├── Makefile
│   ├── put.c
│   └── rand.c
└── Makefile
```

在之前的实验代码的基础上，提供rand.h, rand.c, sched.h给同学们。将lab2中实现的Makefile放置到对应目录下，并在修改相关Makefile，将新增文件纳入到整个编译的工程管理中。

4.2 rand.h rand.c介绍

- rand.h中定义SEED。
- rand.c中实现了一个随机数迭代器，（产生一系列随机数，用来初始化和更新进程的运行时间块与优先级。保证SEED相同时随机数序列是一样的）。

4.3 sched.h数据结构定义

```
#ifndef _SCHED_H
#define _SCHED_H

#define TASK_SIZE   (4096)
#define THREAD_OFFSET  (5 * 0x08)

#ifndef __ASSEMBLER__

/* task的最大数量 */
#define NR_TASKS    64

#define FIRST_TASK  (task[0])
#define LAST_TASK   (task[NR_TASKS-1])

/* 定义task的状态，Lab3中task只需要一种状态。*/
#define TASK_RUNNING                0
// #define TASK_INTERRUPTIBLE       1
// #define TASK_UNINTERRUPTIBLE     2
// #define TASK_ZOMBIE              3
// #define TASK_STOPPED             4

#define PREEMPT_ENABLE  0
#define PREEMPT_DISABLE 1

/* Lab3中进程的数量以及每个进程初始的时间片 */
#define LAB_TEST_NUM        4
#define LAB_TEST_COUNTER    5

/* 当前进程 */
extern struct task_struct *current;

/* 进程指针数组 */
extern struct task_struct * task[NR_TASKS];

/* 进程状态段数据结构 */
struct thread_struct {
    unsigned long long ra;
    unsigned long long sp;
    unsigned long long s0;
    unsigned long long s1;
    unsigned long long s2;
    unsigned long long s3;
    unsigned long long s4;
    unsigned long long s5;
    unsigned long long s6;
    unsigned long long s7;
    unsigned long long s8;
    unsigned long long s9;
    unsigned long long s10;
    unsigned long long s11;
};

/* 进程数据结构 */
struct task_struct {
     long state;    // 进程状态 Lab3中进程初始化时置为TASK_RUNNING
     long counter;  // 运行剩余时间
     long priority; // 运行优先级 1最高 5最低
     long blocked;
     long pid;      // 进程标识符
    // Above Size Cost: 40 bytes

    struct thread_struct thread; // 该进程状态段
};

/* 进程初始化 创建四个dead_loop进程 */
void task_init(void);

/* 在时钟中断处理中被调用 */
void do_timer(void);

/* 调度程序 */
void schedule(void);

/* 切换当前任务current到下一个任务next */
void switch_to(struct task_struct* next);

/* 死循环 */
void dead_loop(void);

#endif

#endif
```

4.4 sched.c进程调度功能实现

4.4.1 实现 task_init()

- 本实验中，我们使用了多个进程，需要对物理内存区域进行划分。此次实验中我们手动做内存分配，把物理地址空间划分成多个帧(frame)。即，从0x80010000地址开始，连续地给此次实验的 4 个 Task [1-4]以及内核栈 Task[0]做内存分配，我们以 4KB 为粒度，按照每个 task 一帧的形式进行分配。**（请同学按照下图的内存空间分配地址，不要随意修改，否则有可能影响到最终的实验结果）**
- 为方便起见，我们要求Task [1-4] 进程均为dead_loop。
- 可将Task [1-4] 看作是从 0 号进程(Task [0]) fork 出的子进程，后续实验的实现将主要考虑如何对这四个子进程进行调度。

```
--------------------------------------------------------------------------
|           |  Task0   |  Task1   |  Task2   |  Task3   |  Task4   |
|   Kernel  |          |          |          |          |          |
|           |  Space   |  Space   |  Space   |  Space   |  Space   |
--------------------------------------------------------------------------
^           ^          ^          ^          ^          ^
0x80000000  0x80010000 0x80011000 0x80012000 0x80013000 0x80014000
```

- 初始化current与task[0]
  - 设置current指向Task0 Space的基地址。
  - 设置task[0]为current。
  - 初始化task[0]中的成员变量
    - state = TASK_RUNNING
    - counter = 0
    - priority = 5
    - blocked = 0
    - pid = 0
  - 设置task[0]的thread中的sp指针为 Task0 Space基地址 + 4KB的偏移。
- 参照task[0]的设置，对task[1-4]完成初始化设置
  - 短作业优先非抢占式算法
    - counter = rand()（task[1-4]的初始剩余运行时间均采用rand()得到）
    - priority = 5（task[1-4]初始优先级均为5）
  - 优先级抢占式算法
    - counter = 7, 6, 5, 4（分别对应 task[1-4]的运行时长）
    - priority = 5（task[1-4]初始优先级均为5）

4.4.2 实现 do_timer()

- 将当前所运行进程的运行剩余时间减少一个单位（counter--）
- 短作业优先非抢占式
  - 如果当前进程运行剩余时间已经用完，则进行调度，选择新的进程来运行，否则继续执行当前进程。
- 优先级抢占式算法
  - 每次do_timer都进行一次抢占试优先级调度。当在do_timer中发现当前运行进程剩余运行时间为0（即当前进程已运行结束）时，需重新为该进程分配其对应的运行时长。相当于重启当前进程，即重新设置每个进程的运行时间长度和初始化的值一致。

4.4.3 实现 schedule()

本次实验我们需要实现两种调度算法：1.短作业优先非抢占式算法，2.优先级抢占式算法。

4.4.3.1 短作业优先非抢占式算法

- 当需要进行调度时按照一下规则进行调度：
  - 遍历进程指针数组task，从LAST_TASK至FIRST_TASK(不包括FIRST_TASK，即Task[0])，在所有运行状态(TASK_RUNNING)下的进程运行剩余时间最小的进程作为下一个执行的进程。
  - 如果所有运行状态下的进程运行剩余时间都为0，则对这些进程的运行剩余时间重新随机赋值（以模拟有新的不同运行时间长度的任务生成），之后再重新进行调度。

4.4.3.2 优先级抢占式算法

- 遍历进程指针数组task，从LAST_TASK至FIRST_TASK(不包括FIRST_TASK)，调度规则如下：
  - 高优先级的进程在运行剩余时间不为0的情况下，优先被运行。
  - 优先级相同且优先级相同的各进程都未运行完毕，则选择运行剩余时间少的进程（若运行剩余时间也相同，则遍历的顺序优先选择）。
- 每次schedule，实现随机更新Task[1-4]进程的priority = rand()（模拟动态优先级变化）

4.4.4 实现 switch_to(struct task_struct* next)

- 判断下一个执行的进程next与当前的进程current是否为同一个进程，如果是同一个进程，则无需做任何处理。
- 实现切换进程的过程
  - 保存当前进程的ra, sp, s0~s11到当前进程的进程状态段（thread）中。
  - 将下一个进程的进程状态段(thread)的相关数据载入到ra, sp, s0~s11中。

4.5 编译及测试

仿照lab2进行调试，预期的实验结果如下：(请对test.c做修改，确保输出自己的组号，例第4组修改XX为04)

- 短作业优先非抢占式

```
ZJU OS LAB 3             GROUP-XX
task init...
[PID = 1] Process Create Successfully! counter = 1
[PID = 2] Process Create Successfully! counter = 4
[PID = 3] Process Create Successfully! counter = 5
[PID = 4] Process Create Successfully! counter = 4
[PID = 0] Context Calculation: counter = 0
[!] Switch from task 0 to task 1, prio: 4, counter: 1 // 由于task[1]的剩余运行时间最少，故选择task[1]
[PID = 1] Context Calculation: counter = 1
[!] Switch from task 1 to task 4, prio: 4, counter: 4 // task[1]运行完成之后，当前task[2, 4]的剩余运行时间最少，按照遍历的顺序（由后向前）选择task[4]
[PID = 4] Context Calculation: counter = 4
[PID = 4] Context Calculation: counter = 3
[PID = 4] Context Calculation: counter = 2
[PID = 4] Context Calculation: counter = 1
[!] Switch from task 4 to task 2, prio: 4, counter: 4 // 选择task[2]
[PID = 2] Context Calculation: counter = 4
[PID = 2] Context Calculation: counter = 3
[PID = 2] Context Calculation: counter = 2
[PID = 2] Context Calculation: counter = 1
[!] Switch from task 2 to task 3, prio: 4, counter: 5 // 选择task[3]
[PID = 3] Context Calculation: counter = 5
[PID = 3] Context Calculation: counter = 4
[PID = 3] Context Calculation: counter = 3
[PID = 3] Context Calculation: counter = 2
[PID = 3] Context Calculation: counter = 1
task init... //当task[1-4]都运行完成，重新初始化task[1-4]
[PID = 1] Process Create Successfully! counter = 5
[PID = 2] Process Create Successfully! counter = 5
[PID = 3] Process Create Successfully! counter = 5
[PID = 4] Process Create Successfully! counter = 2
[!] Switch from task 0 to task 4, prio: 4, counter: 2
[PID = 4] Context Calculation: counter = 2
[PID = 4] Context Calculation: counter = 1
[!] Switch from task 4 to task 3, prio: 4, counter: 5
```

- 优先级抢占式算法

```
ZJU OS LAB 3             GROUP-XX
[PID = 1] Process Create Successfully! counter = 7 priority = 5
[PID = 2] Process Create Successfully! counter = 6 priority = 5
[PID = 3] Process Create Successfully! counter = 5 priority = 5
[PID = 4] Process Create Successfully! counter = 4 priority = 5
[!] Switch from task 0 to task 4, prio: 5, counter: 4 // 此时所有进程的优先级相同，选择剩余运行时间最少的 task[4]
tasks' priority changed
[PID = 1] counter = 7 priority = 1 // 此时task[1]的优先级最高
[PID = 2] counter = 6 priority = 4
[PID = 3] counter = 5 priority = 5
[PID = 4] counter = 4 priority = 4
[!] Switch from task 4 to task 1, prio: 1, counter: 7
tasks' priority changed
[PID = 1] counter = 7 priority = 5
[PID = 2] counter = 6 priority = 5
[PID = 3] counter = 5 priority = 5
[PID = 4] counter = 3 priority = 2 // 此时task[4]的优先级最高
[!] Switch from task 1 to task 4, prio: 2, counter: 3
tasks' priority changed
[PID = 1] counter = 6 priority = 4
[PID = 2] counter = 6 priority = 4
[PID = 3] counter = 5 priority = 4 // 此时task[1 2 3]的优先级最高且相同，由于task[3]的剩余运行时间最少，故选择task[3]
[PID = 4] counter = 3 priority = 5
!] Switch from task 4 to task 3, prio: 4, counter: 5
tasks' priority changed
[PID = 1] counter = 6 priority = 5
[PID = 2] counter = 6 priority = 5
[PID = 3] counter = 5 priority = 4
[PID = 4] counter = 2 priority = 2
[!] Switch from task 3 to task 4, prio: 2, counter: 2
tasks' priority changed
[PID = 1] counter = 6 priority = 5
[PID = 2] counter = 6 priority = 3
[PID = 3] counter = 4 priority = 3 // 此时task[2 3]的优先级最高且相同，由于task[3]的剩余运行时间最少，故选择task[3]
[PID = 4] counter = 2 priority = 4
[!] Switch from task 4 to task 3, prio: 3, counter: 4
...
```

5. 实验任务与要求

请仔细阅读背景知识，确保理解进程调度与进程切换过程，并按照实验步骤完成实验，撰写实验报告，需提交实验报告以及整个工程的压缩包。

- 由于本次实验需要完成两个调度算法，因此需要两种调度算法可以使用gcc –D选项进行控制。
  - -DSJF （短作业优先式）。
  - -DPROPRITY （优先级抢占式）。
    - 在sched.c中使用#ifdef,#endif语句来控制进程调度的代码实现。
  - 修改lab3/Makefile中的CFLAG = ${CF} ${INCLUDE} -DSJF / -DPROPRITY(作业提交的时候这个随机确定一个即可。)
- 实验报告：
  - 各实验步骤截图以及结果分析
  - 实验结束后心得体会
  - 对实验指导的建议（可选）
- 工程文件
  - 所有source code（确保make clean）
- 最终目录
  - 将Lab3_319010XXXX目录压缩并打包（若分组，则一组只需要一位同学提交）

```
Lab3_319010XXXX
├── arch
│   └── riscv
│       ├── include
│       │   ├── put.h
│       │   └── sched.h
│       ├── kernel
│       │   ├── entry.S
│       │   ├── head.S
│       │   ├── Makefile
│       │   ├── sched.c
│       │   ├── strap.c
│       │   └── vmlinux.lds
│       └── Makefile
├── include
│   ├── put.h
│   ├── rand.h
│   └── test.h
├── init
│   ├── main.c
│   ├── Makefile
│   └── test.c
├── lib
│   ├── Makefile
│   ├── put.c
│   └── rand.c
├── Makefile
└── report
    └── 319010XXXX.pdf
```

本文贡献者

王星宇（背景知识，实验步骤 wangxingyu@zju.edu.cn）
张文龙（背景知识，实验步骤 2968829696@qq.com）