Lab 3：RV64 内核线程调度

1 实验目的

- 了解线程概念，并学习线程相关结构体，并实现线程的初始化功能。
- 了解如何使用时钟中断来实现线程的调度。
- 了解线程切换原理，并实现线程的切换。
- 掌握简单的线程调度算法，并完成两种简单调度算法的实现。

2 实验环境

- Docker in Lab0

3 背景知识

3.1 进程与线程

`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`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`。同时一个`进程`可以对应一个或多个`线程`，`线程`之间往往具有相同的代码，共享一块内存，但是却有不同的CPU执行状态。

在本次实验中，为了简单起见， 我们采用 `single-threaded process` 模型， 即`一个进程`对应`一个线程`，进程与线程不做明显区分。

3.1 线程相关属性

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

- `线程ID`：用于唯一确认一个线程。
- `运行栈`：每个线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`：当线程不在执行状态时，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。
- `运行时间片`：为每个线程分配的运行时间。
- `优先级`：在优先级相关调度时，配合调度算法，来选出下一个执行的线程。

3.2 线程切换流程图

- 在每次处理时钟中断时，操作系统首先会将当前线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线程来执行。
- 在进程调度时，操作系统会遍历所有可运行的线程，按照一定的调度算法选出下一个执行的线程。最终将选择得到的线程与当前线程切换。
- 在切换的过程中，首先我们需要保存当前线程的执行上下文，再将将要执行线程的上下文载入到相关寄存器中，至此我们就完成了线程的调度与切换。

4 实验步骤

4.1 准备工程

- 此次实验基于 lab2 同学所实现的代码进行。
- 从 `repo` 同步以下代码: `rand.h/rand.c`, `string.h/string.c`, `mm.h/mm.c`。并按照以下步骤将这些文件正确放置。其中 `mm.h\mm.c` 提供了简单的物理内存管理接口， `rand.h\rand.c`提供了 `rand()` 接口用以提供伪随机数序列， `string.h/string.c`提供了`memset` 接口用以初始化一段内存空间。

```
.
├── arch
│   └── riscv
│       ├── include
│       │   └── mm.h
│       └── kernel
│           └── mm.c
├── include
│   ├── rand.h
│   └── string.h
└── lib
    ├── rand.c
    └── string.c
```

- 在 lab3 中我们需要一些物理内存管理的接口，在此我们提供了 `kalloc` 接口 (见 `mm.c`) 给同学。同学可以用 `kalloc` 来申请 4KB 的物理页。由于引入了简单的物理内存管理，需要在 `_start` 的适当位置调用 `mm_init`, 来初始化内存管理系统，并且在初始化时需要用一些自定义的宏，需要修改 `defs.h`, 在 `defs.h` 添加 如下内容：

```
#define PHY_START 0x0000000080000000
#define PHY_SIZE  128 * 1024 * 1024 // 128MB,  QEMU 默认内存大小
#define PHY_END   (PHY_START + PHY_SIZE)

#define PGSIZE 0x1000 // 4KB
#define PGROUNDUP(addr) ((addr + PGSIZE - 1) & (~(PGSIZE - 1)))
#define PGROUNDDOWN(addr) (addr & (~(PGSIZE - 1)))
```

- 请在添加/修改上述文件代码之后，确保工程可以正常运行，之后再开始实现 `lab3` (有可能需要同学自己调整一些头文件的引入)。
- 在 lab3 中需要同学需要添加并修改 `arch/riscv/include/proc.h` `arch/riscv/kernel/proc.c` 两个文件。
- 本次实验需要实现两种不同的调度算法， 如何控制代码逻辑见 `4.4`

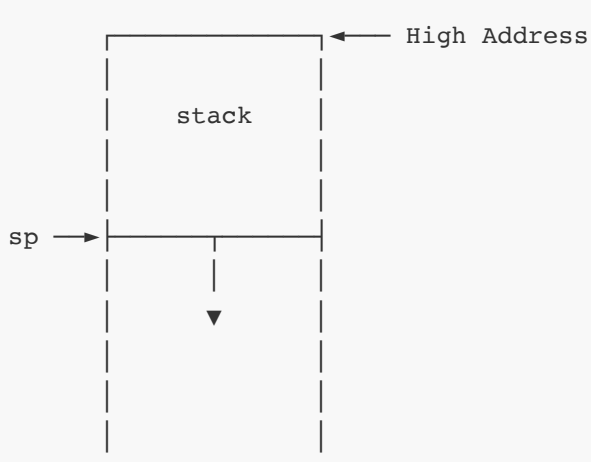
4.2 `proc.h` 数据结构定义

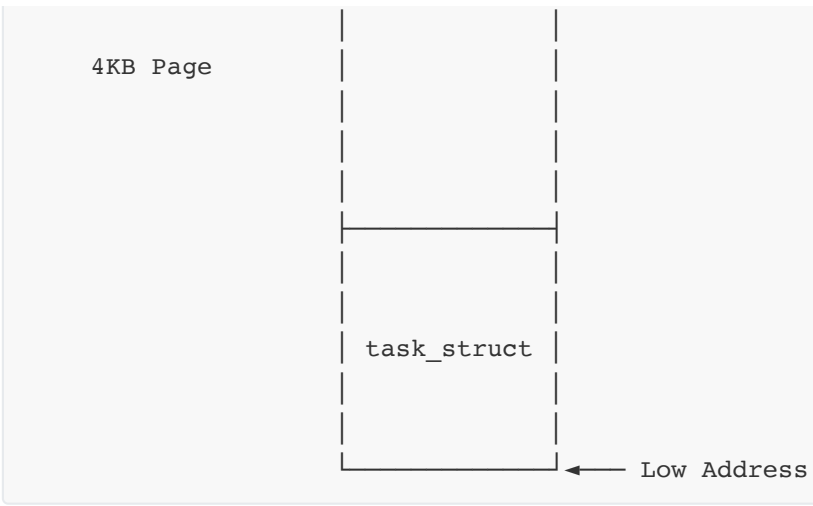
```
// arch/riscv/include/proc.h

#include "types.h"

#define NR_TASKS  (1 + 31) // 用于控制 最大线程数量 （idle 线程 + 31 内核线程）

#define TASK_RUNNING    0 // 为了简化实验，所有的线程都只有一种状态

#define PRIORITY_MIN 1
#define PRIORITY_MAX 10

/* 用于记录 `线程` 的 `内核栈与用户栈指针` */
/* (lab3中无需考虑，在这里引入是为了之后实验的使用) */
struct thread_info {
    uint64 kernel_sp;
    uint64 user_sp;
};

/* 线程状态段数据结构 */
struct thread_struct {
    uint64 ra;
    uint64 sp;
    uint64 s[12];
};

/* 线程数据结构 */
struct task_struct {
    struct thread_info* thread_info;
    uint64 state;    // 线程状态
    uint64 counter;  // 运行剩余时间
    uint64 priority; // 运行优先级 1最低 10最高
    uint64 pid;      // 线程id

    struct thread_struct thread;
};

/* 线程初始化 创建 NR_TASKS 个线程 */
void task_init();

/* 在时钟中断处理中被调用 用于判断是否需要进行调度 */
void do_timer();

/* 调度程序 选择出下一个运行的线程 */
void schedule();

/* 线程切换入口函数*/
void switch_to(struct task_struct* next);

/* dummy funciton: 一个循环程序，循环输出自己的 pid 以及一个自增的局部变量*/
void dummy();
```

4.3 线程调度功能实现

4.3.1 线程初始化

- 在初始化线程的时候，我们参考Linux v0.11中的实现为每个线程分配一个 4KB 的物理页，我们将 `task_struct` 存放在该页的低地址部分， 将线程的栈指针 `sp` 指向该页的高地址。具体内存布局如下图所示：





- 当我们的 OS run 起来时候，其本身就是一个线程 `idle 线程`，但是我们并没有为它设计好 `task_struct`。所以第一步我们要为 `idle` 设置 `task_struct`。并将 `current`, `task[0]` 都指向 `idle`。
- 为了方便起见，我们将 `task[1]` ~ `task[NR_TASKS - 1]`, 全部初始化， 这里和 `idle` 设置的区别在于要为这些线程设置 `thread_struct` 中的 `ra` 和 `sp`.
- 在 `_start` 适当的位置调用 `task_init`

```
//arch/riscv/kernel/proc.c

extern void __dummy();

struct task_struct* idle;           // idle process
struct task_struct* current;        // 指向当前运行线程的 `task_struct`
struct task_struct* task[NR_TASKS]; // 线程数组，所有的线程都保存在此

void task_init() {
    // 1. 调用 kalloc() 为 idle 分配一个物理页
    // 2. 设置 state 为 TASK_RUNNING;
    // 3. 由于 idle 不参与调度 可以将其 counter / priority 设置为 0
    // 4. 设置 idle 的 pid 为 0
    // 5. 将 current 和 task[0] 指向 idle

    /* YOUR CODE HERE */

    // 1. 参考 idle 的设置，为 task[1] ~ task[NR_TASKS - 1] 进行初始化
    // 2. 其中每个线程的 state 为 TASK_RUNNING, counter 为 0, priority 使用 rand() 来设置, pid 为该线程在线程数组中的下标。
    // 3. 为 task[1] ~ task[NR_TASKS - 1] 设置 `thread_struct` 中的 `ra` 和 `sp`,
    // 4. 其中 `ra` 设置为 __dummy （见 4.3.2）的地址， `sp` 设置为 该线程申请的物理页的高地址

    /* YOUR CODE HERE */

    printk("...proc_init done!\n");
}
```

> Debug 提示：
>
> 1. 修改 `proc.h` 中的 `NR_TASKS` 为一个比较小的值, 比如 5， 这样 除去 `task[0]` (idle)，只需要初始化 4 个线程，方便调试。
> 2. 注意以上的修改只是为了在做实验的过程中方便调试，最后一定记住要修改回去！！！

4.3.2 `__dummy` 与 `dummy` 介绍

- `task[1]` ~ `task[NR_TASKS - 1]`都运行同一段代码 `dummy()` 我们在 `proc.c` 添加 `dummy()`:

```
// arch/riscv/kernel/proc.c

void dummy() {
    uint64 MOD = 1000000007;
    uint64 auto_inc_local_var = 0;
    int last_counter = -1;
    while(1) {
        if (last_counter == -1 || current->counter != last_counter) {
            last_counter = current->counter;
            auto_inc_local_var = (auto_inc_local_var + 1) % MOD;
            printk("[PID = %d] is running. auto_inc_local_var = %d\n", current->pid, auto_inc_local_var);
        }
    }
}
```

> Debug 提示： 可以修改 printk 打印更多的信息

- 当线程在运行时，由于时钟中断的触发，会将当前运行线程的上下文环境保存在栈上 (lab2 中实现的 `_traps`)。当线程再次被调度时，会将上下文从栈上恢复，但是当我们创建一个新的线程，此时线程的栈为空，当这个线程被调度时，是没有上下文需要被恢复的，所以我们需要为线程`第一次调度`提供一个特殊的返回函数 `__dummy`
- 在 `entry.S` 添加 `__dummy`
  - 在`__dummy` 中将 sepc 设置为 `dummy()` 的地址, 并使用 `sret` 从中断中返回。
  - `__dummy` 与 `_traps`的 `restore`部分相比， 其实就是省略了从栈上恢复上下文的过程 （但是手动设置了 `sepc` ）。

```
# arch/riscv/kernel/entry.S

    .global __dummy
__dummy:
    # YOUR CODE HERE
```

4.3.3 实现线程切换

- 判断下一个执行的线程 `next` 与当前的线程 `current` 是否为同一个线程，如果是同一个线程，则无需做任何处理，否则调用 `__switch_to` 进行线程切换。

```
// arch/riscv/kernel/proc.c

extern void __switch_to(struct task_struct* prev, struct task_struct* next);

void switch_to(struct task_struct* next) {
    /* YOUR CODE HERE */
}
```

- 在 `entry.S` 中实现线程上下文切换 `__switch_to`：
  - `__switch_to` 接受两个 `task_struct` 指针作为参数
  - 保存当前线程的 `ra`，`sp`，`s0~s11` 到当前线程的 `thread_struct` 中
  - 将下一个线程的 `thread_struct` 中的相关数据载入到 `ra`，`sp`，`s0~s11` 中。

```
# arch/riscv/kernel/entry.S

    .globl __switch_to
__switch_to:
    # save state to prev process
    # YOUR CODE HERE

    # restore state from next process
    # YOUR CODE HERE

    ret
```

> Debug 提示： 可以尝试是否可以从 idle 正确切换到 process 1

4.3.4 实现调度入口函数

- 实现 `do_timer()`，并在 `时钟中断处理函数` 中调用。

```
// arch/riscv/kernel/proc.c

void do_timer(void) {
    /* 1. 如果当前线程是 idle 线程 直接进行调度 */
    /* 2. 如果当前线程不是 idle 对当前线程的运行剩余时间减 1
          若剩余时间任然大于0 则直接返回 否则进行调度 */

    /* YOUR CODE HERE */
}
```

4.3.5 实现线程调度

本次实验我们需要实现两种调度算法：1.短作业优先调度算法，2.优先级调度算法。

4.3.5.1 短作业优先调度算法

- 当需要进行调度时按照一下规则进行调度：
  - 遍历线程指针数组 `task`(不包括 `idle` ，即 `task[0]`)，在所有运行状态 (`TASK_RUNNING`) 下的线程运行剩余时间 `最少` 的线程作为下一个执行的线程。
  - 如果 `所有` 运行状态下的线程运行剩余时间都为0，则对 `task[1]` ~ `task[NR_TASKS-1]` 的运行剩余时间重新赋值 (使用 `rand()`)，之后再重新进行调度。

```
// arch/riscv/kernel/proc.c

void schedule(void) {
    /* YOUR CODE HERE */
}
```

> Debug 提示： 将 `NR_TASKS` 改为较小的值，调用 `printk` 将所有线程的信息打印出来。

4.3.5.2 优先级调度算法

- 参考 Linux v0.11 调度算法实现 实现。

```
// arch/riscv/kernel/proc.c

void schedule(void) {
    /* YOUR CODE HERE */
}
```

4.4 编译及测试

- 由于加入了一些新的 .c 文件，可能需要修改一些Makefile文件，请同学自己尝试修改，使项目可以编译并运行。
- 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 （短作业优先调度）。
  - DPRIORITY （优先级调度）。
  - 在 `proc.c` 中使用 `#ifdef` , `#endif` 来控制代码。 修改Makefile中的 `CFLAG = ${CF} ${INCLUDE} -DSJF / -DPRIORITY` (作业提交的时候 `Makefile` 选择任意一个都可以)
- 短作业优先调度输出示例 (为了便于展示，这里一共只初始化了 4 个线程) 同学们最后提交时需要 保证 NR_TASKS 为 32 不变

```
OpenSBI v0.9

  ____                    _____ ____ _____
 / __ \                  / ____|  _ \_   _|
| |  | |_ __   ___ _ __ | (___ | |_) || |
| |  | | '_ \ / _ \ '_ \ \___ \|  _ < | |
| |__| | |_) |  __/ | | |____) | |_) || |_
 \____/| .__/ \___|_| |_|_____/|____/_____|
       | |
       |_|

...

Boot HART MIDELEG         : 0x0000000000000222
Boot HART MEDELEG         : 0x000000000000b109

...mm_init done!
...proc_init done!
Hello RISC-V
idle process is running!

SET [PID = 1 COUNTER = 10]
SET [PID = 2 COUNTER = 10]
SET [PID = 3 COUNTER = 5]
SET [PID = 4 COUNTER = 2]

switch to [PID = 4 COUNTER = 2]
[PID = 4] is running. auto_inc_local_var = 1
[PID = 4] is running. auto_inc_local_var = 2
```

```
switch to [PID = 3 COUNTER = 5]
[PID = 3] is running. auto_inc_local_var = 1
.....
[PID = 3] is running. auto_inc_local_var = 5

switch to [PID = 2 COUNTER = 10]
[PID = 2] is running. auto_inc_local_var = 1
...
[PID = 2] is running. auto_inc_local_var = 10

switch to [PID = 1 COUNTER = 10]
[PID = 1] is running. auto_inc_local_var = 1
...
[PID = 1] is running. auto_inc_local_var = 10

SET [PID = 1 COUNTER = 9]
SET [PID = 2 COUNTER = 4]
SET [PID = 3 COUNTER = 4]
SET [PID = 4 COUNTER = 10]

switch to [PID = 3 COUNTER = 4]
[PID = 3] is running. auto_inc_local_var = 6
...
[PID = 3] is running. auto_inc_local_var = 9
```

- 优先级调度输出示例

```
OpenSBI v0.9
   ____                   _____ ____ _____
  / __ \                 / ____|  _ \_   _|
 | |  | |_ __   ___ _ __ | (___ | |_) || |
 | |  | | '_ \ / _ \ '_ \ \___ \|  _ < | |
 | |__| | |_) |  __/ | | |____) | |_) || |_
  \____/| .__/ \___|_| |_|_____/|____/_____|
        | |
        |_|

...

Boot HART MIDELEG         : 0x0000000000000222
Boot HART MEDELEG         : 0x000000000000b109

...mm_init done!
...proc_init done!
Hello RISC-V
idle process is running!

SET [PID = 1 PRIORITY = 1 COUNTER = 1]
SET [PID = 2 PRIORITY = 4 COUNTER = 4]
SET [PID = 3 PRIORITY = 10 COUNTER = 10]
SET [PID = 4 PRIORITY = 4 COUNTER = 4]

switch to [PID = 3 PRIORITY = 10 COUNTER = 10]
[PID = 3] is running. auto_inc_local_var = 1
...
[PID = 3] is running. auto_inc_local_var = 10

switch to [PID = 4 PRIORITY = 4 COUNTER = 4]
[PID = 4] is running. auto_inc_local_var = 1
...
[PID = 4] is running. auto_inc_local_var = 4

switch to [PID = 2 PRIORITY = 4 COUNTER = 4]
[PID = 2] is running. auto_inc_local_var = 1
...
[PID = 2] is running. auto_inc_local_var = 4

switch to [PID = 1 PRIORITY = 1 COUNTER = 1]
[PID = 1] is running. auto_inc_local_var = 1

SET [PID = 1 PRIORITY = 1 COUNTER = 1]
SET [PID = 2 PRIORITY = 4 COUNTER = 4]
SET [PID = 3 PRIORITY = 10 COUNTER = 10]
SET [PID = 4 PRIORITY = 4 COUNTER = 4]

switch to [PID = 3 PRIORITY = 10 COUNTER = 10]
[PID = 3] is running. auto_inc_local_var = 11
...
```

思考题

1. 在 RV64 中一共用 32 个通用寄存器， 为什么 `context_switch` 中只保存了14个？
2. 当线程第一次调用时， 其 `ra` 所代表的返回点是 `__dummy`。 那么在之后的线程调用中 `context_switch` 中， `ra` 保存/恢复的函数返回点是什么呢？ 请同学用gdb尝试追踪一次完整的线程切换流程， 并关注每一次 `ra` 的变换。

作业提交

同学需要提交实验报告以及整个工程代码。在提交前请使用 `make clean` 清除所有构建产物。