Report of Lab3

1. 环境搭建

1.1 建立映射

利用前面实验的映射方法建立本地目录lab3与docker image内实验目录的映射

1.2 组织文件结构

组织文件结构如下：

```
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
```

2. 实验流程说明

- 简单来说，本次实验就是将lab2中trap_s内调用的`print_message()`修改为调用sched.c内的`do_timer()`函数（即每发生1e5次时钟中断，完成一次进程调度）。`do_timer()`作为进程调度的入口函数，在其内部执行如`schedule()`，`switch_to()`等操作完成整个调度过程。
- 实验流程梳理
  - 有关sepc寄存器保存的问题
    - lab2中实现了将时钟中断委托给S mode后，假设当前进程为prevoius task，则再次发生时钟中断后，将进入`trap_s`，发生时钟中断的异常地址会自动保存到`sepc`
    - `trap_s`前面部分处理时，会先把previous task的所有寄存器保存在它的栈上（此时的`sp`指向的是previous task的栈）
    - `trap_s`中间部分处理，会调用`do_timer()->schedule()->switch_to()`将进程由previous task调度为next task。需要注意的是，执行完`switch_to()`后，`sp`指向的是next task的栈
    - `trap_s`后部分的处理，会把next task栈中的所有寄存器恢复，其中就包括了先前保存过的next task中的sepc的值，这样`sret`后`pc`就会回到next task之前被切出去的位置

继续执行

这样 `trap_s` 里面保存的 `sepc` 和取出 `sepc` 的地方位于两个不同task的栈上

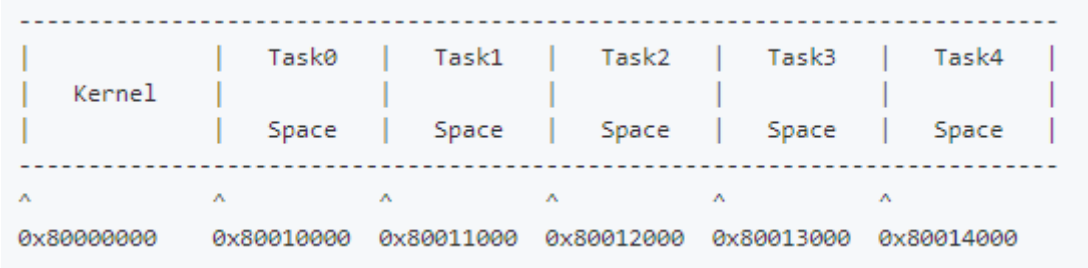
- 实现 `switch_to` 的过程如下
  - 所有task[1-4]在 `task_init()` 内被初始化后，会将每个task的 `task[i]->thread.ra` 指向一个“初始化函数”
  - 当在第一次 `switch_to()` 到某个next task(这里假设是 `task[1]`)，`switch_to()` 上下文切换过程如下:
    - `switch_to()` 的前面部分处理是将previous task的{ra, sp, s[0-11]}保存到 `previous task->thread` 的结构体中
    - `switch_to()` 的后面部分处理是将程序的{ra, sp, s[0-11]}从 `task[1]->thread` 中恢复。由于是第一次运行该task，这里恢复出来的 `ra` 是 `task_init()` 函数中保存的那个“初始化函数”地址
    - `switch_to()` 最后的处理是执行 `ret`，将 `ra` 赋值给 `pc`。
    - 这样第一次运行该task，此时便会执行到那个“初始化函数”，紧接着在初始化函数中，我们将 `dead_loop()` 的入口地址赋值给 `sepc` 寄存器，并通过 `sret` 返回，此时 `pc` 便等于 `sepc` 中保存在 `sepc` 中的 `dead_loop` 入口地址，便进入了死循环
  - 当下一次时钟中断产生，来到 `switch_to()` 时（即 `task[1]` 要切换到其他task）
    - `switch_to()` 的前半段仍旧是把previous task（此时为 `task[1]`）的{ra, sp, s[0-11]}分别保存到 `task[1]->thread` 结构体中。（此时保存的 `ra` 不再是 `task_init()` 函数保存的那个“初始化函数”地址，而是常规的函数调用关系所自动保存的 `ra` 的值）
    - ...
  - 若之后的某一次进程调度的next task为 `task[1]`，则先前保存到 `task[1]->thread` 中的那些寄存器的值，又会在 `switch_to()` 过程的后半段被恢复，最终通过 `ret` 将 `ra` 赋值给 `pc`，返回到 `trap_s` 中进行 `trap_s` 的后部分处理，即将next task栈中的所有寄存器恢复，其中就包括了先前保存过的 `task[1]` 的 `sepc` 的值，这样 `sret` 之后就会回到 `task[1]` 之前被切出去的位置继续执行。

3. 进程调度功能实现

因为要实现SJF与PRIORITY两种调度算法，因此我们在sched.c中需要使用 `#ifdef` 与 `#endif` 来控制调度的实现，并在 `lab3/Makefile` 中的 `CFLAG` 加上 `-DSJF / -DPRIORITY` 来指定采用的调度算法

3.1 task_init()的实现

`task_init()` 相对比较简单，就是将 `task[0-4]` 指向对应的内存空间地址（如下图），并对各task的数据成员进行赋值，此时task[1-4]的thread结构体内的 `ra` 均指向 `sepc_assign()` 这个“初始化函数”，在该函数内会将 `dead_loop` 函数的入口地址赋值给 `sepc`，并执行 `sret`



其中 `sepc_assign()` 的代码如下：

```
/* 将dead_loop赋值给sepc寄存器 */
void sepc_assign(void)
{
    unsigned long long temp = dead_loop;
    asm volatile("csrw sepc, %0" : : "r"(temp));
    asm volatile("sret": :);
}
```

3.2 do_timer()的实现

`do_timer()` 函数是在 `trap_s` 的时钟中断处理中被调用，相当于一个接口。

- 首先会将当前运行的进程 `current` 的运行时间减少一个单位
- 然后根据采用的调度算法分别进行一些调整
    - SJF算法
        - 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（直接return）
    - PRIORITY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
- 最后会执行 `schedule()` 去进行进程调度

对应的代码如下：

```
    // 2. schedule according to SJF or PRIORITY
    #ifdef SJF
    if(current->counter > 0)        // counter>0, continue this process
        return;
    current->counter = 0;

    #else
    if(current->counter == 0){
        switch(current->pid){
            case 1:
                current->counter = 7;
                break;
            case 2:
                current->counter = 6;
                break;
            case 3:
                current->counter = 5;
                break;
            case 4:
                current->counter = 4;
                break;
            default:
                //printf("ERROR:current->pid=0, counter=0\n");
                puts("ERROR: pid = 0\n");
        }
    }
    #endif
```

3.3 schedule()的实现

`schedule()` 是根据采用的算法选择下一个调度的进程。

- SJF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。
- PRIORITY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，优先被运行。
        - `优先级相同`，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`）

3.4 switch_to()的实现

`switch_to`负责执行上下文切换，保存与恢复相应的寄存器。需要注意的是，因为寄存器`s0`就是帧指针`fp`，因此若第一次执行`switch_to()`，在恢复阶段会导致`fp`被清零，也就使得`switch_to()`内部的局部变量与形参丢失，如`next`变量，这会导致严重的后果。因此我选择用汇编函数的方式在entry.S中实现了`__switch_to(prev, next)`来负责寄存器的保存与恢复，这样就不用考虑局部变量`next`的丢失。

- sched.c中的`switch_to()`函数

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

    struct task_struct* prev = current;
    current = next;
    __switch_to(prev, next);
    return;
}
```

- entry.S中的`__switch_to(prev, next)`函数（`a0`，`a1`为两个参数）

```
.globl __switch_to    #a0 = prev, a1 = next
.align 3
__switch_to:
    addi t0, a0, 40 # thread offset = 40B
    sd ra, 0 * WORD_SIZE(t0)
    sd sp, 1 * WORD_SIZE(t0)
    sd s0, 2 * WORD_SIZE(t0)
    sd s1, 3 * WORD_SIZE(t0)
    sd s2, 4 * WORD_SIZE(t0)
    sd s3, 5 * WORD_SIZE(t0)
    sd s4, 6 * WORD_SIZE(t0)
    sd s5, 7 * WORD_SIZE(t0)
    sd s6, 8 * WORD_SIZE(t0)
    sd s7, 9 * WORD_SIZE(t0)
    sd s8, 10 * WORD_SIZE(t0)
    sd s9, 11 * WORD_SIZE(t0)
    sd s10, 12 * WORD_SIZE(t0)
    sd s11, 13 * WORD_SIZE(t0)

    addi t0, a1, 40
    ld ra, 0 * WORD_SIZE(t0)
    ld sp, 1 * WORD_SIZE(t0)
    ld s0, 2 * WORD_SIZE(t0)
    ld s1, 3 * WORD_SIZE(t0)
    ld s2, 4 * WORD_SIZE(t0)
    ld s3, 5 * WORD_SIZE(t0)
    ld s4, 6 * WORD_SIZE(t0)
    ld s5, 7 * WORD_SIZE(t0)
    ld s6, 8 * WORD_SIZE(t0)
    ld s7, 9 * WORD_SIZE(t0)
    ld s8, 10 * WORD_SIZE(t0)
    ld s9, 11 * WORD_SIZE(t0)
    ld s10, 12 * WORD_SIZE(t0)
    ld s11, 13 * WORD_SIZE(t0)
    ret
```

最后执行`ret`将next task的`ra`赋值给`pc`，执行后续操作。

4. 实验结果

- SJF算法

```
oslab@37fa0a91c982:~/lab3$ make run
qemu-system-riscv64 -nographic -machine virt -kernel vmlinux
qemu-system-riscv64: warning: No -bios option specified. Not loading a firmware.
qemu-system-riscv64: warning: This default will change in a future QEMU release. Please use the -bios option to avoid breakages when this happens.
qemu-system-riscv64: warning: See QEMU's deprecation documentation for details.
ZJU OS LAB 3             GROUP-32
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: 5, counter: 1
[PID = 1] Context Calculation: counter = 1
[!] Switch from task 1 to task 4, prio: 5, counter: 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: 5, counter: 4
[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: 5, counter: 5
[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
[PID = 1] Reset counter = 5
[PID = 2] Reset counter = 5
[PID = 3] Reset counter = 5
[PID = 4] Reset counter = 2
[!] Switch from task 3 to task 4, prio: 5, counter: 2
[PID = 4] Context Calculation: counter = 2
[PID = 4] Context Calculation: counter = 1
[!] Switch from task 4 to task 3, prio: 5, counter: 5
```

- PRIORITY算法

```
oslab@37fa0a91c982:~/lab3$ make run
qemu-system-riscv64 -nographic -machine virt -kernel vmlinux
qemu-system-riscv64: warning: No -bios option specified. Not loading a firmware.
qemu-system-riscv64: warning: This default will change in a future QEMU release. Please use the -bios option to avoid breakages when this happens.
qemu-system-riscv64: warning: See QEMU's deprecation documentation for details.
ZJU OS LAB 3             GROUP-32
task init...
[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
tasks' priority changed
[PID = 1] counter = 7 priority = 1
[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
[!] 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
[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
```

5. 实验心得

本次实验我对调度算法从理论了解上升到了代码实现，并且对进程切换的具体实现也有了深刻的理解。希望实验手册中能对各个函数的具体实现内容能讲解的更清楚一点……