

Operating system

Lab 2



Author: 吴杭

Date: 2024/10/17

Autumn-Fall 2024-2025 Semester

Table of Contents

Chapter 1: 实验流程	3
1.1) 初始化工程	3
1.2) 线程调度功能实现	4
1.2.1) 线程初始化	4
1.2.2) dummy 与 _dummy 的实现	5
1.2.3) 实现线程的切换	5
Declaration	6

Chapter 1: 实验流程

1.1) 初始化工程

从仓库中 clone 相关的代码到本地，由于 lab2 是在 lab1 的基础上开发的，所以我们需要把 lab1 已经完成的代码和 lab2 合并。最后得到结果如下

```
→ lab2 git:(main) X tree .
.
├── Makefile
├── arch
│   └── riscv
│       ├── Makefile
│       ├── include
│       │   ├── defs.h
│       │   ├── mm.h
│       │   ├── proc.h
│       │   └── sbi.h
│       └── kernel
│           ├── Makefile
│           ├── clock.c
│           ├── entry.S
│           ├── head.S
│           ├── mm.c
│           ├── proc.c
│           ├── sbi.c
│           ├── trap.c
│           └── vmlinux.lds
├── fw_jump.bin
├── include
│   ├── printk.h
│   ├── stddef.h
│   ├── stdint.h
│   ├── stdlib.h
│   └── string.h
├── init
│   ├── Makefile
│   ├── main.c
│   └── test.c
└── lib
    ├── Makefile
    ├── printk.c
    ├── rand.c
    └── string.c

8 directories, 28 files
```

Figure 1: 工程结构

然后根据实验文档，为了让 `kalloc` 能正常分配内存，我们需要在 `defs.h` 中添加相应的宏。然后需要在 `_start` 的适当位置调用 `mm_init`，这里我们选择在 `call task_init`（task init 相关部分会在报告后续内容提及）之前调用 `mm_init`。

```
# initialize the memory management
call mm_init

# init the tasks before starting the kernel
call task_init
```

1.2) 线程调度功能实现

1.2.1) 线程初始化

线程初始化的时候，需要给每个线程都分配一个 4KiB 的物理页。然后需要初始化一些用于记录线程运行信息的数据结构。而第一个我们需要初始化的特殊线程就是 `idle` 线程（也就是我们运行的操作系统本身）。实现的思路和实验框架给出的注释一致，当然下面展示的代码的注释也已经展示了设计的思路。具体的实现代码如下

```
// allocate a physical page for idle
idle = (struct task_struct*)kalloc();

// set the state as TASK_RUNNING
idle->state = TASK_RUNNING;

// set the counter and priority
idle->counter = idle->priority = 0;

// set the pid as 0
idle->pid = 0;

// set the current and task[0]
current = idle;
task[0] = idle;
```

接下来我们需要初始化其他的线程，而相对于 `idle` 线程不同的是，其他的线程需要参与调度，所以我们需要为他们分配一个随机的优先级。然后由于调度过程中，需要保存 PC 返回的地址，所以我们需要在他们的 `thread_struct` 中设置好 `ra` 寄存器。而对于还没有开始运行过的进程，本实验中我们把他们的 `ra` 设置为一个特殊的地址 `__dummy`，当他们被调度的时候，就会从 `__dummy` 开始运行，接着进入他们各自的程序。

为了能够分离地管理这些线程的内存分配资源，我们需要为他们独立分配栈空间，所以我们也需要在他们的 `thread_struct` 中记录下他们的栈空间的高地址。相应的代码如下：

```
for (int i = 1; i < NR_TASKS; i++) {
    struct task_struct* new_task = (struct task_struct*)kalloc();
    new_task->state = TASK_RUNNING;
    new_task->pid = i;

    // set counter and priority using rand
    new_task->counter = 0;
    new_task->priority = rand() % (PRIORITY_MAX-PRIORITY_MIN+1) + PRIORITY_MIN;

    // set the ra and sp
    new_task->thread.ra = (uint64_t)__dummy;
```

```

    new_task->thread.sp = (uint64_t)new_task + PGSIZE; // notice new_task is also an
address of the struct (the bottom of the PAGE)

    task[i] = new_task;
}

```

上面的代码中要注意的是，`new_task` 是指向这个 `task` 的指针，同时也是这个 `task` 被分配到的内存空间的低地址，而整个 `task` 被分配到的空间的大小为 `PGSIZE` (也就是 4KiB)，所以每个 `task` 的 `sp` 的值就是 `(uint_64t)new_task + PGSIZE`。

1.2.2) dummy 与 __dummy 的实现

本实验中，所有的 `task` (`idle` 除外) 都运行同一段代码 `dummy`。

上面我们提到了，现成第一次调度的时候，需要提供一个特殊的地址 `__dummy`，根据实验文档中的设计，我们设计这个函数的代码如下

```

__dummy:
    la t0, dummy # load the address of dummy into the t0 register
    csrw sepc, t0
    sret # the program will return to the address of dummy

```

1.2.3) 实现线程的切换

通过一个 `switch_to` 函数，来实现线程的切换。这个函数接受一个参数，为指向下一个要切换的线程的指针 `next`，然后判断 `next` 和当前的线程的指针 `current` 是否为同一个线程 (通过比较两者的 `pid` 得到)，如果是同一个，那么就不需要调度，如果不是同一个，那么就调用 `__switch_to` 函数进行调度。

```

void switch_to(struct task_struct *next) {
    // check if the current task is the same as the next one using their unique pid
    if (next->pid != current->pid) {
        // if they are not the same, call __switch_to
        __switch_to(current, next);
    }
}

```

`__switch_to` 函数的任务就是储存当前的运行的程序的上下文 `ra`, `sp`, `s0-s11` 到之前用来储存线程信息的数据结构 (`task_struct` 中的 `thread_struct`) 中，然后再把下一个要切换的程序的上下文 load 到 `ra`, `sp`, `s0-s11` 这些寄存器中。这里要注意的是我们传给 `__switch_to` 的是指向 `task_struct` 的指针，而 `thread_struct` 储存在 `task_struct` 中的起始位置在第 32 个 byte，所以我们 store 和 load 的时候需要在 `a0` 的第 32 个 byte 开始。

代码如下：

```

__switch_to:
    # save states to prev process
    # a0 is the base address of task_struct prev, and the content of thread starts from
32(a0)
    sd ra, 32(a0)
    ...

```

```
ld ra, 32(a1)
...
# restore state from next process
ret
```

Declaration

I hereby declare that all the work done in this lab 1 is of my independent effort.