浙江大学实验报告

一、实验内容

准备工程

根据实验指导，从git仓库中将需要的代码拷贝至本地，并在需要改动的文件内增加相应的代码。

`proc.h`数据结构定义

该部分没有需要写代码的部分，阅读理解即可。

线程调度功能实现

线程初始化

代码如下，根据实验指导要求以及注释的要求写代码即可。分配物理页通过调用`kalloc()`完成；进程的信息设置通过给结构体的成员赋值即可完成；要记得将当前进程`current`设置为`idle`。

```
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
    idle = (struct task_struct *)kalloc(PGSIZE);

    idle->state = TASK_RUNNING;
    idle->counter = 0;
    idle->priority = 0;
    idle->pid = 0;
    current = idle;
    task[0] = idle;
    // 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
    for (int i = 1; i < NR_TASKS; i++)
    {
        task[i] = (struct task_struct *)kalloc(PGSIZE);
        task[i]->state = TASK_RUNNING;
        task[i]->counter = 0;
        task[i]->priority = rand() % PRIORITY_MAX + PRIORITY_MIN;
        task[i]->pid = i;
        task[i]->thread.ra = (uint64)__dummy;
        task[i]->thread.sp = (uint64)task[i] + PGSIZE;
    }
```

```
    printk("...proc_init done!%d\n", NR_TASKS);
}
```

`__dummy`与`dummy`介绍

根据实验指导要求，`__dummy`要做的事情非常简单：通过`la`得到`dummy()`的地址，并将其设置为`sepc`。从中断退出时，进程就会进入执行`dummy()`函数。由于`__dummy`只在进程被第一次调度时使用（每一个进程只使用一次该函数），第一次调度时进程还没有上下文，因此`__dummy`不需要恢复上下文。

当一个进程被第一次调度时，由于我们给进程的`ra`赋予的初始值是`__dummy`的地址，在中断中（S态）系统将进程结构体中的`ra`载入`ra`，使得系统在进程切换后进入`__dummy`。`__dummy`修改了`sepc`后执行`sret`，从S态的中断返回至U态的`dummy()`。由此完成进程切换。

```
__dummy:
    la t0, dummy
    csrw sepc, t0
    sret
```

实现线程切换

根据实验指导，`switch_to()`用c语言实现，并调用汇编语言实现的`__switch_to`完成线程切换。如果调度算法给出的下一个线程即是当前的线程，则不必作出任何动作；否则，修改当前线程的值，并进行线程切换。

```
void switch_to(struct task_struct *next)
{
    struct task_struct *p = current;
    // current process is the process to be switched
    if (current == next)
    {
        return;
    }
    else
    {
        // change the current process
        current = next;
        // switch to the next process
        __switch_to(p, next);

        return;
    }
}
```

根据实验指导，保存并恢复上下文即可。

```
.global __switch_to
__switch_to:
    # save the context of the last trap
    sd ra, 40(a0)
    sd sp, 48(a0)
    sd s0, 56(a0)
```

```
    sd s1, 64(a0)
    sd s2, 72(a0)
    sd s3, 80(a0)
    sd s4, 88(a0)
    sd s5, 96(a0)
    sd s6, 104(a0)
    sd s7, 112(a0)
    sd s8, 120(a0)
    sd s9, 128(a0)
    sd s10, 136(a0)
    sd s11, 142(a0)
    # load the context of the next trap
    ld ra, 40(a1)
    ld sp, 48(a1)
    ld s0, 56(a1)
    ld s1, 64(a1)
    ld s2, 72(a1)
    ld s3, 80(a1)
    ld s4, 88(a1)
    ld s5, 96(a1)
    ld s6, 104(a1)
    ld s7, 112(a1)
    ld s8, 120(a1)
    ld s9, 128(a1)
    ld s10, 136(a1)
    ld s11, 142(a1)
    ret
```

实现调度入口函数

进行调度即调用`schedule()`。根据注释和实验指导的要求写代码即可。

需要注意的是，这里的要求暗含了一个特性：当一个线程被执行，它在执行完成（`counter`变为0）之前是不会被切换到其他线程的。

```
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 */
    if (current == idle)
    {
        schedule();
        return;
    }
    else
    {
        if (--current->counter)
            return;
        else
        {
            schedule();
            return;
        }
    }
```

```
}
```

实现线程调度

SJF的调度实现如下。根据实验指导要求，被调度的线程必须是还有剩余执行时间的线程（即`counter`不为0）。因此我们首先检查是否有线程还有剩余执行时间，如果否则给所有线程重新随机分配执行时间；如果是则选出剩余执行时间最短的进程执行。这里需要尤其注意，在选出剩余执行时间最短的进程时，要确保被选中的进程的剩余执行时间不为0。

```
void schedule(void)
{
    struct task_struct *p = task[1];
    int zero_flag = 0, i;
    for (i = 1; i < NR_TASKS; i++)
    {
        if (task[i]->counter)
        {
            // there is a process can be switched
            break;
        }
    }
    if (i == NR_TASKS)
    {
        // reassign values to counters
        for (i = 1; i < NR_TASKS; i++)
        {

            task[i]->counter = rand() % 10 + 1;
            // printk("task[%d]->counter = %d  ", i, task[i]->counter);
            // printk("task[%d]->priority = %d\n", i, task[i]->priority);
            printk("SET [PID = %d PRIORITY = %d COUNTER = %d]\n", i, task[i]-
>priority, task[i]->counter);
        }
    }
    for (i = 2; i < NR_TASKS; i++)
    {
        // use SJF to pick out the next process
        if (p->counter == 0 || p->state != TASK_RUNNING)
            p = task[i];
        else if (task[i]->counter < p->counter && task[i]->state == TASK_RUNNING
&& task[i]->counter != 0)
            p = task[i];
    }
    switch_to(p);
    return;
}
```

优先级的调度实现如下。根据实验指导要求，被调度的线程必须是还有剩余执行时间的线程（即`counter`不为0）。因此我们首先检查是否有线程还有剩余执行时间，如果否则给所有线程重新随机分配执行时间；如果是则选出优先级最高的进程执行。这里需要尤其注意，在选出进程时，要确保被选中的进程的剩余执行时间不为0。

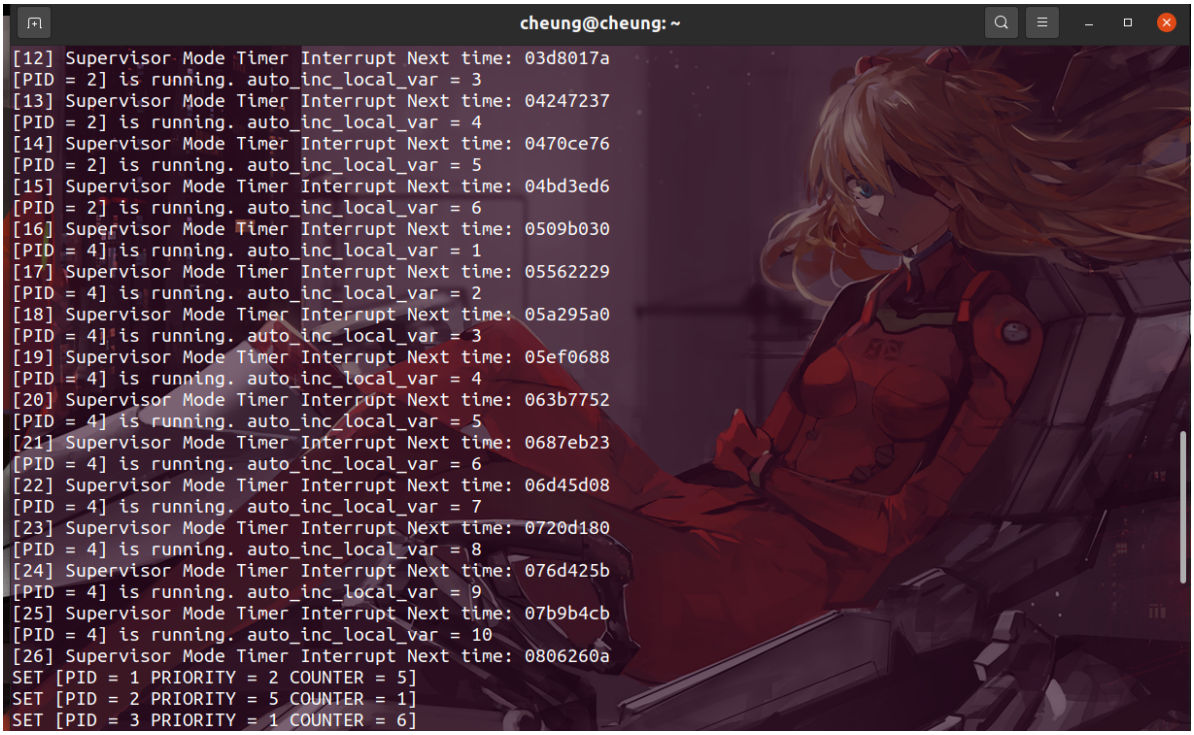
```
void schedule(void)
{
```

```
    struct task_struct *p = task[1];
    int zero_flag = 0, i;
    for (i = 1; i < NR_TASKS; i++)
    {
        if (task[i]->counter)
        {
            // there is a process can be switched
            break;
        }
    }
    if (i == NR_TASKS)
    {
        // reassign values to counters
        for (i = 1; i < NR_TASKS; i++)
        {
            task[i]->counter = rand() % 10 + 1;
            // printk("task[%d]->counter = %d  ", i, task[i]->counter);
            // printk("task[%d]->priority = %d\n", i, task[i]->priority);
            printk("SET [PID = %d PRIORITY = %d COUNTER = %d]\n", i, task[i]-
>priority, task[i]->counter);
        }
    }
    for (i = 2; i < NR_TASKS; i++)
    {
        // use priority to pick out the next process
        if (p->counter == 0 || p->state != TASK_RUNNING)
            p = task[i];
        else if (task[i]->priority >= p->priority && task[i]->state ==
TASK_RUNNING && task[i]->counter != 0)
            p = task[i];
    }

    switch_to(p);
    return;
}
```

编译及测试

运行程序，可以看到正常输出。首先测试SJF调度。可以看到第一次调度时，剩余时间最短的第一个进程被调度，然后是第三个进程，第三个进程结束后是第五个进程……由以下两个图可以看到，线程执行的顺序与剩余执行时间的升序一致。需要注意的是，由于一开始的进程是`idle`，根据实验指导要求，此时直接进行调度，`schedule()`会被调用并且第一个线程会有输出。



当所有线程的剩余执行时间被重新赋值时，情况与第一次调度略有不同。这里的第一个线程和第四个线程没有产生输出，是因为实验指导要求先自减再判断剩余时间。第一个线程和第四个线程的剩余时间自减后为0，则没有调用`schedule()`，没有产生输出。

再测试优先级的调度。由以下两张图可以看出，线程的执行顺序与优先级的逆序一致。

到了第二次调度，可以看到，这里第一个线程和第四个线程没有输出的原因与SJF调度的原因是一致的。除此之外，可以看到输出均符合预期。

二、思考题

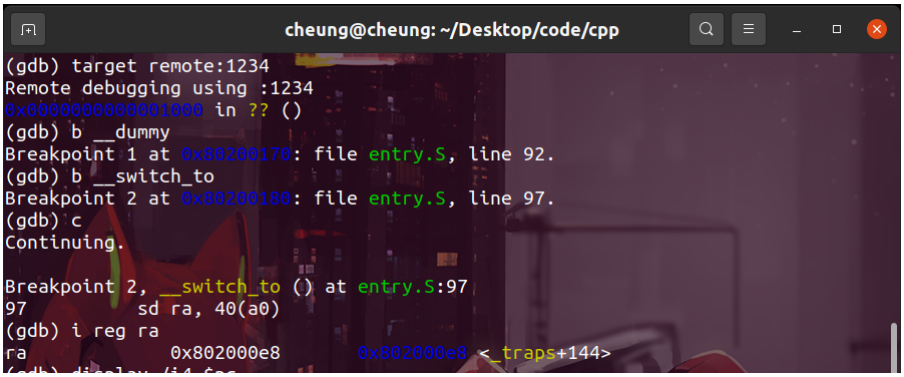
1. 在 RV64 中一共用 32 个通用寄存器， 为什么 `context_switch` 中只保存了14个?

首先，我们需要清楚PC的变化顺序。首先，时钟中断触发，PC进入`_traps`，再进入`trap_handler()`。`trap_handler()`调用一系列函数，完成线程切换。每个线程的全部寄存器，其实在触发时钟中断的时候已经全部保存了一次；只保存14个寄存器更类似于函数调用的保存，`trap_handler()`调用了`__switch_to`，后者将trap的14个callee-save寄存器保存好。

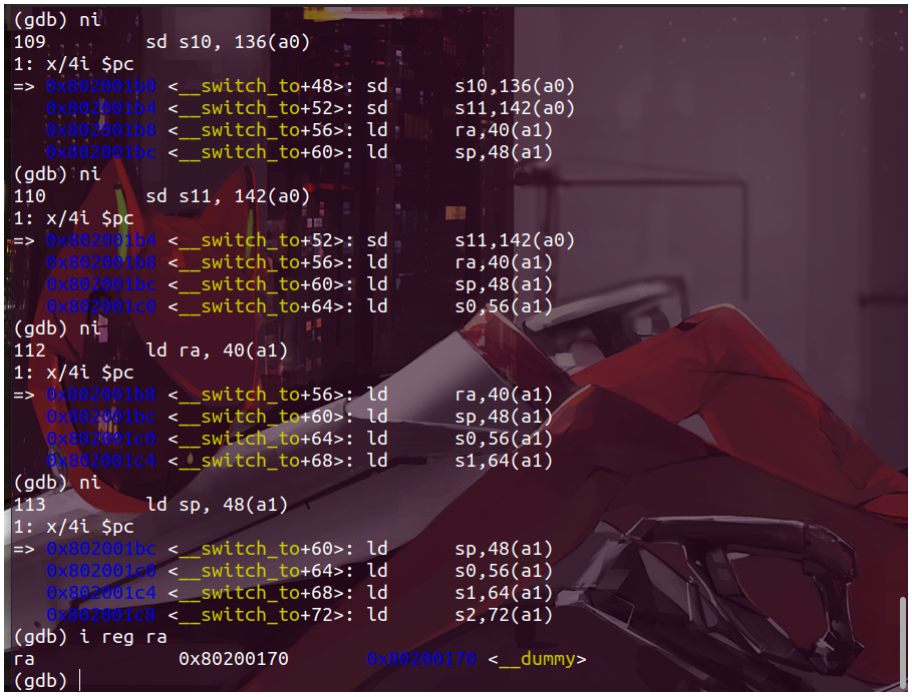
2. 当线程第一次调用时， 其`ra`所代表的返回点是`__dummy`。那么在之后的线程调用中`context_switch`中， `ra`保存/恢复的函数返回点是什么呢？ 请同学用gdb尝试追踪一次完整的

线程切换流程， 并关注每一次 `ra` 的变换。

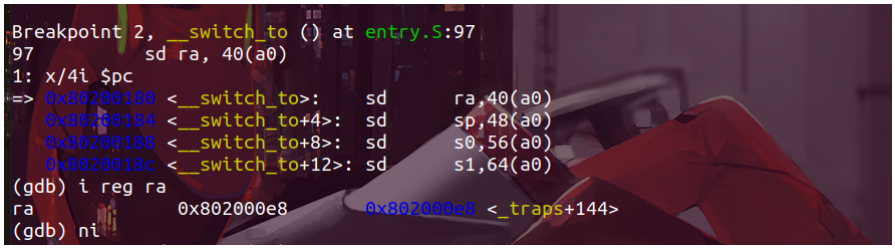
第一次调度在刚进入`__switch_to`时，可以看到`ra`的地址是`call trap_handler`的下一条指令。



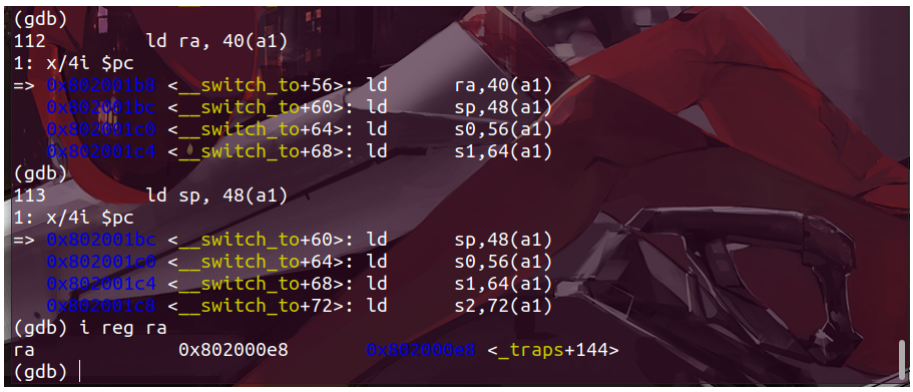
在线程切换，载入新的`ra`后，可以看到此时`ra`的值是`__dummy`的首地址。符合设计要求。



完成第一次调度之后，（对于同一个线程）第二次进入`__switch_to`时，可以看到`ra`的地址仍然是`call trap_handler`的下一条指令。



跟踪至载入新的`ra`后，可以看到此时`ra`的值同样是`call trap_handler`的下一条指令。这个结果稍微有些意外，我们对其进行了探究，探究过程在讨论心得部分。

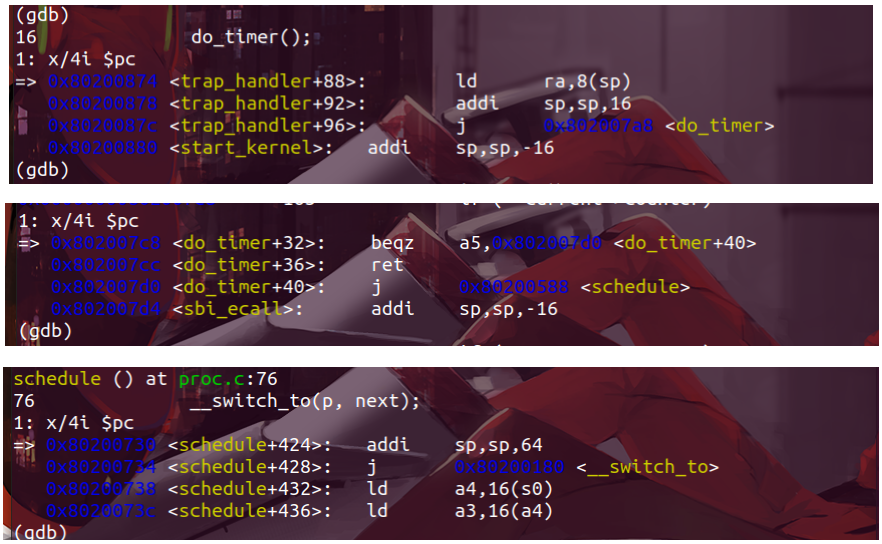


三、讨论心得

本次实验总体来说难度不大，实验过程中我们最大的困惑点在于整个线程切换的流程。我们的讨论结果是：

1. 时钟中断触发，系统由U态进入S态
2. 由 `_traps` 进入 `trap_handler`，再进入 `do_timer` 等一系列函数
3. 在 `switch_to` 内调用 `__switch_to`，此时 `ra` 内保存的是 `switch_to` 内调用 `__switch_to` 的指令的下一条指令的地址
4. `__switch_to` 完成线程切换，进入 `__dummy` 或回到 `_traps`，最后由 `sret` 进入U态

但是实际上，恢复的 `ra` 的值同样是 `call trap_handler` 的下一条指令。因此层层跟踪调用，可以发现 `trap_handler` 调用其余一些列函数时使用的是 `j` 而不是 `jal`：



并且，汇编代码直接忽略掉了 `switch_to` 这一层函数，直接由 `schedule` 调用 `__switch_to`（即 `switch_to` 的函数体被整合到了 `schedule` 内）；多层函数的 `return` 命令都被略去，只保留了 `__switch_to` 的 `ret`。我们没能查到编译器这么优化的具体原因，但我们猜测是因为：

1. 这一系列由我们编写的函数内，在调用其他函数后都是直接接上 `return`，并无其他指令需要执行
2. 多个返回语句简化保留至一个返回语句，可以节省运行成本

编译器优化之后的程序与我们编写的程序有着相当的出入，虽然最后结果一样，但是其过程有许多微妙的不同。