浙江大学

本科实验报告

课程名称：	操作系统
实验名称：	RV64 内核线程调度
姓　　名：	蔡雨谦
学　　院：	计算机学院
系：	计算机系
专　　业：	计算机科学与技术
学　　号：	3210102466
指导教师：	寿黎但

2023 年　　10 月　　26 日

浙江大学实验报告

一、 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。

二、 实验环境

- Ubuntu 22.04.2 LTS Windows Subsystem for Linux 2

三、 操作方法与实验步骤

1. 准备工程

从课程提供的 github 仓库获取实验代码框架，准备进行后续实验。本次实验需要修改或完成的文件有：

- defs.h
- proc.c
- proc.h
- trap.c
- head.S
- entry.S
- 若干 Makefile 文件

在 lab2 中需要一些物理内存管理的接口，可以用实验提供的 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)))
```

为了让程序正确运行，还需对 makefile 文件及部分头文件的引用进行调整，部分调整如下：

向部分 lab1 的代码中引入 lab2 的头文件，如 trap.c。

```
// trap.c
#include "clock.h"
#include "printk.h"
#include "proc.h"
```

修改 makefile 文件以编译新添加的文件。

```
all:
    ${MAKE} -C kernel all
    ${LD} -T kernel/vmlinux.lds kernel/*.o ../../init/*.o ../../lib/*.o ../../test/*.o -o ../../vmlinux
    $(shell test -d boot || mkdir -p boot)
    ${OBJCOPY} -O binary ../../vmlinux ./boot/Image
    nm ../../vmlinux >  ../../System.map

clean:
```

-D DSJF 表示使用短作业优先算法

-D DPRIORITY 表示使用优先级调度算法（实际上使用的 Linux v0.11 中的调度算法并不是纯粹的优先级调度）

```
INCLUDE = -I $(shell pwd)/include -I $(shell pw
CF = -march=$(ISA) -mabi=$(ABI) -mcmodel=medany
CFLAG = ${CF} ${INCLUDE} -D DSJF
# CFLAG = ${CF} ${INCLUDE} -D DPRIORITY
```

2. RV64 内核线程调度

2.1 task_init

该函数用于为线程分配物理空间并设置各变量的值。根据提示完成代码即可。

```
    // 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 */
    uint64 idlePage = kalloc();
    idle = (struct task_struct *)idlePage;
    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, 此外，为了单元测试的需要，counter 和 priority 进行如下赋值：
//      task[i].counter  = task_test_counter[i];
//      task[i].priority = task_test_priority[i];
// 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 */
for (int i = 1; i < NR_TASKS; i++)
{
    uint64 taskPage = kalloc();
    task[i] = (struct task_struct *)taskPage;
    task[i]->state = TASK_RUNNING;
    task[i]->counter = task_test_counter[i];
    task[i]->priority = task_test_priority[i];
    task[i]->pid = i;
    task[i]->thread.ra =(uint64)__dummy;
    task[i]->thread.sp = (uint64)task[i] + PGSIZE;
}
```

在 head.S 文件中添加跳转到 task_init 的代码：

```
_start:

    # 内核引导部分的代码

    la sp, boot_stack_top

    #lab2-----------
    jal ra, mm_init

    jal ra, task_init
    #---------------
```

2.2 dummy 与__dummy

每个线程在第一次调度时都会进入 dummy 函数进行一次 schedule_test，之后在里面一直循环并打印线程信息。

```
// arch/riscv/kernel/proc.c
void dummy()
{
    printk("dummy\n");
    schedule_test();
    uint64 MOD = 1000000007;
    uint64 auto_inc_local_var = 0;
    int last_counter = -1;
    while (1)
    {
        if ((last_counter == -1 || current->counter != last_counter) && current->counter != 0)
        {
            if (current->counter == 1)
            {
                --(current->counter); // forced the counter to be zero if this thread is going to be scheduled
            }                         // in case that the new counter is also 1, leading the information not printed.
            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);
        }
    }
    //printk("\n");
}
```

Task_init 中用于初始化线程的 ra 的__dummy 是 dummy 函数的地址：

```
task[i]->thread.ra =(uint64)__dummy;
```

```
    .extern dummy
    .globl __dummy
__dummy:
    # YOUR CODE HERE
    la t0, dummy
    csrw sepc, t0
    sret
```

2.3 switch_to 与__switch_to

Switch_to 函数用来切换线程。参数为目标线程。当前运行的线程与想要切换至的目标线程相同时无需进行切换；需要切换时调用__switch_to 函数，参数为切换前后的线程。

```c
void switch_to(struct task_struct *next)
{
    if (next != current)
    {
        //printk("1");
        struct task_struct *prev = current;
        //printk("2");
        current = next;
        //printk("3");
        __switch_to(prev, next);
        //printk("4");
        return;
    }
    else
    {
        return;
    }
    //printk("\nswitch_to end\n");
}
```

__switch_to 函数先将切换前的线程的信息保存下来(prev)，再将切换后的线程的信息加载进来(next)。

```asm
    .global __switch_to
__switch_to:
    # save state to prev process
    # YOUR CODE HERE
    #la a0, switch_to
    sd ra,48(a0)
    sd sp,56(a0)
    sd s0,64(a0)
    sd s1,72(a0)
    sd s2,80(a0)
    sd s3,88(a0)
    sd s4,96(a0)
    sd s5,104(a0)
    sd s6,112(a0)
    sd s7,120(a0)
    sd s8,128(a0)
    sd s9,136(a0)
    sd s10,144(a0)
    sd s11,152(a0)
```

```
    # restore state from next process
    # YOUR CODE HERE
    ld ra,48(a1)
    ld sp,56(a1)
    ld s0,64(a1)
    ld s1,72(a1)
    ld s2,80(a1)
    ld s3,88(a1)
    ld s4,96(a1)
    ld s5,104(a1)
    ld s6,112(a1)
    ld s7,120(a1)
    ld s8,128(a1)
    ld s9,136(a1)
    ld s10,144(a1)
    ld s11,152(a1)

    ret
```

线程结构如下，thread_info,state,counter,priority,pid 存放在 caller 寄存器中，随函数调用自动保存，共 48bytes。Thread_struct 结构中 s0 到 s11 是 callee 寄存器，需要在汇编文件中保存，从寄存器的地址加 48 字节处开始保存。

```
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
    /*size = 6 * 8 = 48*/

    struct thread_struct thread;
};
```

2.4 do_timer

该函数在时钟中断时运行，用于判断是否需要进行调度。根据提示填写代码即可。

```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 若剩余时间仍然大于0 则直接返回 否则进行调度

    /* YOUR CODE HERE */
    if (current == idle)
    {
        schedule();
    }
    else
    {
        if (current->counter > 0)
        {
            current->counter--;
        }
        if (current->counter == 0)
        {
            schedule();
        }
    }
}
```

```c
    // YOUR CODE HERE
    if (scause >> 63) // 最高位为1就是interrupt
    {
        unsigned long rest = scause - 0x8000000000000000;
        if (rest == 5)// 剩余位表示5时是timer interrupt
        {
            //printk("\nkernel is running!\n[S] Supervisor Mode Timer Interrupt\n");
            clock_set_next_event();
            //lab2
            do_timer();
        }
    }
}
```

2.5 调度算法

2.5.1 DSJF

短作业优先算法，选出全部线程中 counter 最小的线程作为下一个执行的线程。当线程的 counter 全为 0 时，将线程的 counter 随机赋值再次调度。

```c
    int min_counter = 0xFFFFFFFF;
    struct task_struct *next;
    int all0 = 1;
    for (int i = 1; i < NR_TASKS; i++)
    {
        if (task[i]->state == TASK_RUNNING && task[i]->counter != 0)
        {
            all0 = 0;
            if (task[i]->counter < min_counter)
            {
                min_counter = task[i]->counter;
                next = task[i];
            }
        }
    }
```

```c
if (all0)
{
    printk("\n");
    for (int i = 1; i < NR_TASKS; i++)
    {
        task[i]->counter = rand();
        printk("SET [PID = %d COUNTER = %d]\n", task[i]->pid, task[i]->counter);
    }
    schedule();
}
else
{
    if (next)
    {
        printk("\n\nswitch to [PID = %d COUNTER = %d]\n\n", next->pid, next->counter);
        switch_to(next);
    }
}
```

2.5.2 DPRIORITY

参考 Linux v0.11 调度算法实现。选择线程中 counter 最大的线程作为下一个执行的线程。当 counter 全为 0 时重新赋值为(counter>>1 + priority)。

```c
int max = 0;
struct task_struct *next;
int all0 = 1;
for (int i = NR_TASKS-1; i > 0; i--)
{
    if (task[i]->state == TASK_RUNNING && task[i]->counter != 0)
    {
        all0 = 0;
        if (task[i]->counter > max)
        {
            max = task[i]->counter;
            next = task[i];
        }
    }
}
```

```c
if (all0)
{
    printk("\n");
    for (int i = NR_TASKS-1; i > 0; i--)
    {
        task[i]->counter = (task[i]->counter>>1)+task[i]->priority;
        //task[i]->priority = rand();
        printk("SET [PID = %d PRIORITY = %d COUNTER = %d]\n", task[i]->pid, task[i]->priority, task[i]->counter);
    }
    schedule();
}
else
{
    if (next)
    {
        printk("\n\nswitch to [PID = %d PRIORITY = %d COUNTER = %d]\n\n", next->pid, next->priority, next->counter);
        switch_to(next);
    }
}
```

3. 测试

3.1 DSJF

NR_TASKS = 4

```
dummy

C
BBBBDDDDDDDDC
C
BBBBDDDDDDDDCC
C
BBBBDDDDDDDDCCC
C
BBBBDDDDDDDDCCCC
C
BBBBDDDDDDDDCCCCC
C
BBBBDDDDDDDDCCCCCC
C
BBBBDDDDDDDDCCCCCCC
C
BBBBDDDDDDDDCCCCCCCC
C
BBBBDDDDDDDDCCCCCCCCC
NR_TASKS = 4, SJF test passed!
```

NR_TASKS = 8

```
dummy

F
HHBBBBEEEEEDDDDDDDDCCCCCCCCCGGGGGGGGGGF
F
HHBBBBEEEEEDDDDDDDDCCCCCCCCCGGGGGGGGGGFF
F
HHBBBBEEEEEDDDDDDDDCCCCCCCCCGGGGGGGGGGFFF
F
HHBBBBEEEEEDDDDDDDDCCCCCCCCCGGGGGGGGGGFFFF
F
HHBBBBEEEEEDDDDDDDDCCCCCCCCCGGGGGGGGGGFFFFF
F
HHBBBBEEEEEDDDDDDDDCCCCCCCCCGGGGGGGGGGFFFFFF
F
HHBBBBEEEEEDDDDDDDDCCCCCCCCCGGGGGGGGGGFFFFFFF
F
HHBBBBEEEEEDDDDDDDDCCCCCCCCCGGGGGGGGGGFFFFFFFF
F
HHBBBBEEEEEDDDDDDDDCCCCCCCCCGGGGGGGGGGFFFFFFFFF
F
HHBBBBEEEEEDDDDDDDDCCCCCCCCCGGGGGGGGGGFFFFFFFFFF
F
HHBBBBEEEEEDDDDDDDDCCCCCCCCCGGGGGGGGGGFFFFFFFFFFF
F
HHBBBBEEEEEDDDDDDDDCCCCCCCCCGGGGGGGGGGFFFFFFFFFFFF
NR_TASKS = 8, SJF test passed!
```

NR_TASKS = 16

```
dummy

F
IHHOOLLLNNNBBBBMMMMEEEEEPPPPPPJJJJJJJJDDDDDDDDCCCCCCCCCGGGGGGGGGGKKKKKKKKKKF
F
IHHOOLLLNNNBBBBMMMMEEEEEPPPPPPJJJJJJJJDDDDDDDDCCCCCCCCCGGGGGGGGGGKKKKKKKKKKFF
F
IHHOOLLLNNNBBBBMMMMEEEEEPPPPPPJJJJJJJJDDDDDDDDCCCCCCCCCGGGGGGGGGGKKKKKKKKKKFFF
F
IHHOOLLLNNNBBBBMMMMEEEEEPPPPPPJJJJJJJJDDDDDDDDCCCCCCCCCGGGGGGGGGGKKKKKKKKKKFFFF
F
IHHOOLLLNNNBBBBMMMMEEEEEPPPPPPJJJJJJJJDDDDDDDDCCCCCCCCCGGGGGGGGGGKKKKKKKKKKFFFFF
F
IHHOOLLLNNNBBBBMMMMEEEEEPPPPPPJJJJJJJJDDDDDDDDCCCCCCCCCGGGGGGGGGGKKKKKKKKKKFFFFFF
F
IHHOOLLLNNNBBBBMMMMEEEEEPPPPPPJJJJJJJJDDDDDDDDCCCCCCCCCGGGGGGGGGGKKKKKKKKKKFFFFFFF
F
IHHOOLLLNNNBBBBMMMMEEEEEPPPPPPJJJJJJJJDDDDDDDDCCCCCCCCCGGGGGGGGGGKKKKKKKKKKFFFFFFFF
F
IHHOOLLLNNNBBBBMMMMEEEEEPPPPPPJJJJJJJJDDDDDDDDCCCCCCCCCGGGGGGGGGGKKKKKKKKKKFFFFFFFFF
F
IHHOOLLLNNNBBBBMMMMEEEEEPPPPPPJJJJJJJJDDDDDDDDCCCCCCCCCGGGGGGGGGGKKKKKKKKKKFFFFFFFFFF
F
IHHOOLLLNNNBBBBMMMMEEEEEPPPPPPJJJJJJJJDDDDDDDDCCCCCCCCCGGGGGGGGGGKKKKKKKKKKFFFFFFFFFFF
F
IHHOOLLLNNNBBBBMMMMEEEEEPPPPPPJJJJJJJJDDDDDDDDCCCCCCCCCGGGGGGGGGGKKKKKKKKKKFFFFFFFFFFFF
NR_TASKS = 16, SJF test passed!
```

3.2 DPRIORITY

NR_TASKS = 4

```
dummy

B
CCCCCCCCCDDDDDDDDB
B
CCCCCCCCCDDDDDDDDBB
B
CCCCCCCCCDDDDDDDDBBB
B
CCCCCCCCCDDDDDDDDBBBB
NR_TASKS = 4, PRIORITY test passed!
```

NR_TASKS = 8

```
dummy

H
FFFFFFFFFFFFGGGGGGGGGGCCCCCCCCCDDDDDDDDEEEEEBBBBH
H
FFFFFFFFFFFFGGGGGGGGGGCCCCCCCCCDDDDDDDDEEEEEBBBBHH
NR_TASKS = 8, PRIORITY test passed!
```

NR_TASKS = 16

```
H
FFFFFFFFFFFFKKKKKKKKKKGGGGGGGGGGCCCCCCCCDDDDDDDDJJJJJJJPPPPPPEEEEEMMMMBBBBNNNLLLOOH
H
FFFFFFFFFFFFKKKKKKKKKKGGGGGGGGGGCCCCCCCCDDDDDDDDJJJJJJJPPPPPPEEEEEMMMMBBBBNNNLLLOOHH

switch to [PID = 8 PRIORITY = 25 COUNTER = 1]

dummy

I
FFFFFFFFFFFFKKKKKKKKKKGGGGGGGGGGCCCCCCCCDDDDDDDDJJJJJJJPPPPPPEEEEEMMMMBBBBNNNLLLOOHHI
NR_TASKS = 16, PRIORITY test passed!
```

四、思考题

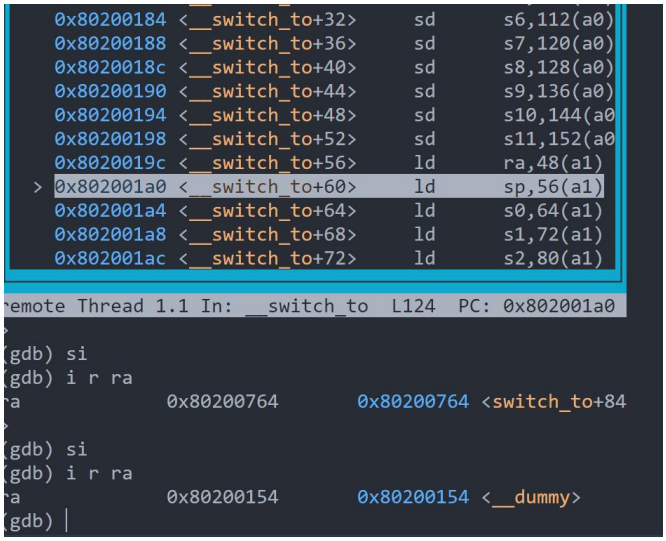
1. 在 RV64 中一共用 32 个通用寄存器，为什么 context_switch 中只保存了 14 个?

__switch_to 函数使用的 s0 到 s11 寄存器是 callee 寄存器，不会随函数调用自动保存，所以要手动保存，其他的寄存器(caller 寄存器)会自动被编译器保存。而 ra 和 sp 寄存器用于跟踪线程运行情况和栈的地址，也需要进行保存。

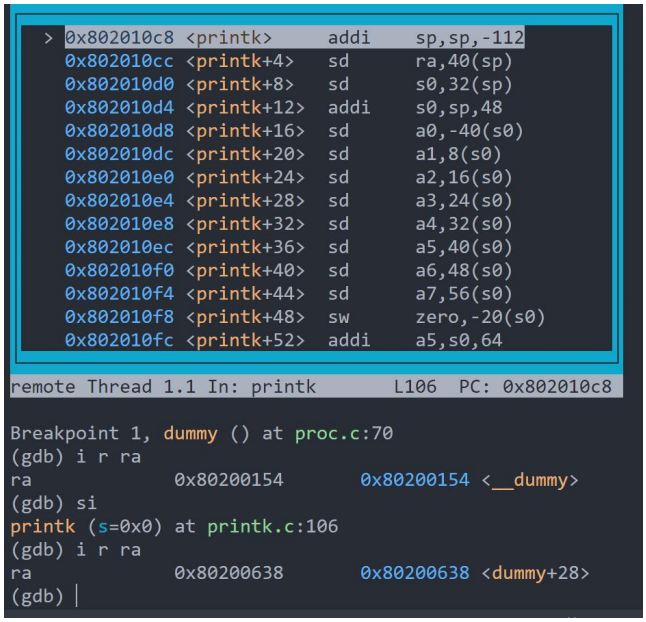
2. 当线程第一次调用时，其 ra 所代表的返回点是 __dummy。那么在之后的线程调用中 context_switch 中，ra 保存/恢复的函数返回点是什么呢? 请同学用 gdb 尝试追踪一次完整的线程切换流程，并关注每一次 ra 的变换（需要截图)。

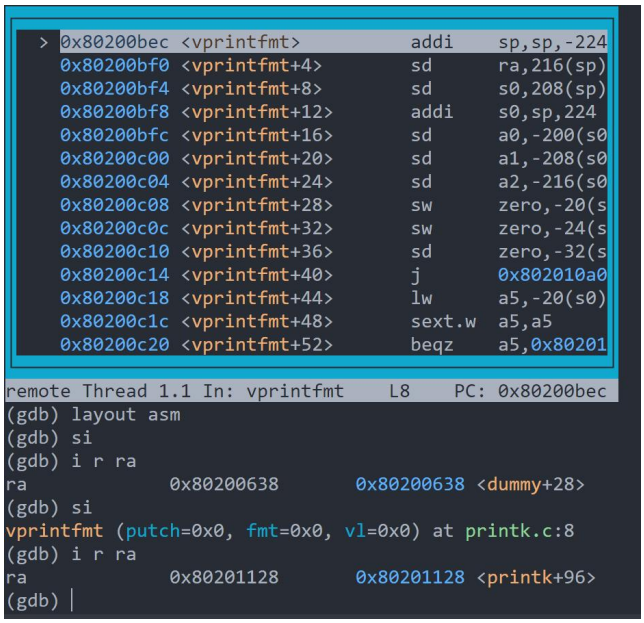
线程第一次调用与切换过程：

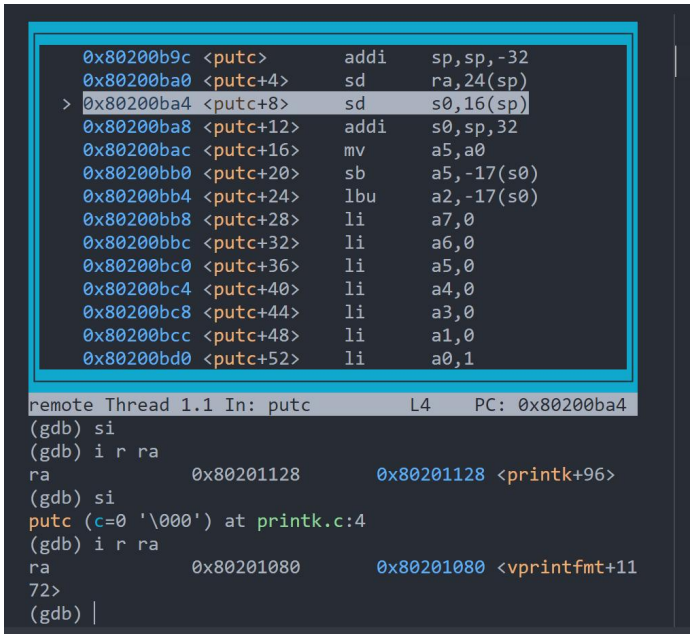
如图为一次线程切换的开始，__switch_to 函数调用时，ra 中保存的是 switch_to 函数的地址，该地址在__switch 函数中又被保存到 a0 寄存器（参数 prev）中。之后将 a1（参数 next）的“ra”（第一次调用时是__dummy,因为此时还没有“上文”）保存在 ra 中 。

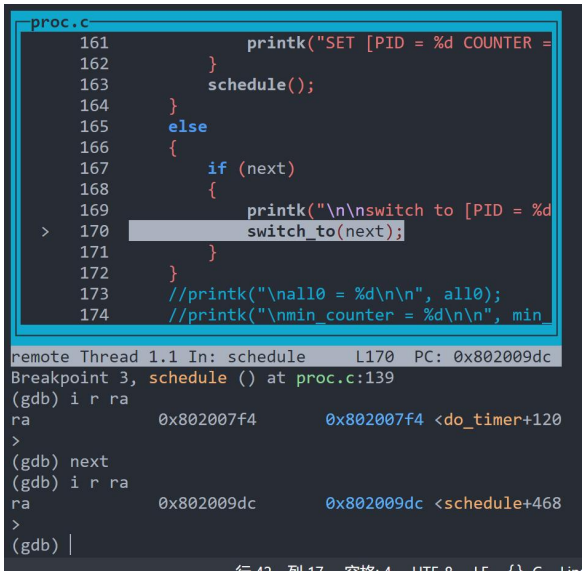


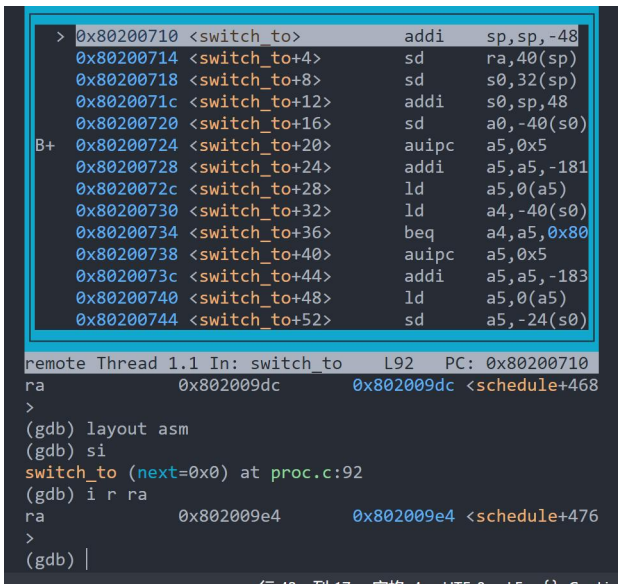
在打印信息时接连调用了许多函数，每次调用函数时都将当前的地址保存在 ra 中，调用完后回到 ra 中的地址。如 dummy 函数调用 printk 函数时 ra 的值变为调用处的地址<dummy+28>，printk 调用 vprintfmt 时 ra 变为<printk+96> 等等。


> 0x802010c8 <printk> addi sp,sp,-112
0x802010cc <printk+4> sd ra,40(sp)
0x802010d0 <printk+8> sd s0,32(sp)
0x802010d4 <printk+12> addi s0,sp,48
0x802010d8 <printk+16> sd a0,-40(s0)
0x802010dc <printk+20> sd a1,8(s0)
0x802010e0 <printk+24> sd a2,16(s0)
0x802010e4 <printk+28> sd a3,24(s0)
0x802010e8 <printk+32> sd a4,32(s0)
0x802010ec <printk+36> sd a5,40(s0)
0x802010f0 <printk+40> sd a6,48(s0)
0x802010f4 <printk+44> sd a7,56(s0)
0x802010f8 <printk+48> sw zero,-20(s0)
0x802010fc <printk+52> addi a5,s0,64
remote Thread 1.1 In: printk L106 PC: 0x802010c8
Breakpoint 1, dummy () at proc.c:70
(gdb) i r ra
ra 0x80200154 0x80200154 <__dummy>
(gdb) si
printk (s=0x0) at printk.c:106
(gdb) i r ra
ra 0x80200638 0x80200638 <dummy+28>
(gdb)

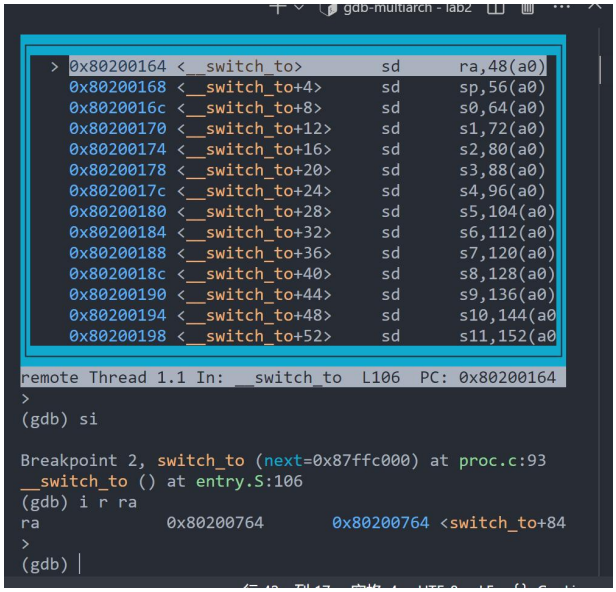

> 0x80200bec <vprintfmt> addi sp,sp,-224
0x80200bf0 <vprintfmt+4> sd ra,216(sp)
0x80200bf4 <vprintfmt+8> sd s0,208(sp)
0x80200bf8 <vprintfmt+12> addi s0,sp,224
0x80200bfc <vprintfmt+16> sd a0,-200(s0
0x80200c00 <vprintfmt+20> sd a1,-208(s0
0x80200c04 <vprintfmt+24> sd a2,-216(s0
0x80200c08 <vprintfmt+28> sw zero,-20(s
0x80200c0c <vprintfmt+32> sw zero,-24(s
0x80200c10 <vprintfmt+36> sd zero,-32(s
0x80200c14 <vprintfmt+40> j 0x802010a0
0x80200c18 <vprintfmt+44> lw a5,-20(s0)
0x80200c1c <vprintfmt+48> sext.w a5,a5
0x80200c20 <vprintfmt+52> beqz a5,0x80201
remote Thread 1.1 In: vprintfmt L8 PC: 0x80200bec
(gdb) layout asm
(gdb) si
(gdb) i r ra
ra 0x80200638 0x80200638 <dummy+28>
(gdb) si
vprintfmt (putch=0x0, fmt=0x0, vl=0x0) at printk.c:8
(gdb) i r ra
ra 0x80201128 0x80201128 <printk+96>
(gdb)


0x80200b9c <putc> addi sp,sp,-32
0x80200ba0 <putc+4> sd ra,24(sp)
> 0x80200ba4 <putc+8> sd s0,16(sp)
0x80200ba8 <putc+12> addi s0,sp,32
0x80200bac <putc+16> mv a5,a0
0x80200bb0 <putc+20> sb a5,-17(s0)
0x80200bb4 <putc+24> lbu a2,-17(s0)
0x80200bb8 <putc+28> li a7,0
0x80200bbc <putc+32> li a6,0
0x80200bc0 <putc+36> li a5,0
0x80200bc4 <putc+40> li a4,0
0x80200bc8 <putc+44> li a3,0
0x80200bcc <putc+48> li a1,0
0x80200bd0 <putc+52> li a0,1
remote Thread 1.1 In: putc L4 PC: 0x80200ba4
(gdb) si
(gdb) i r ra
ra 0x80201128 0x80201128 <printk+96>
(gdb) si
putc (c=0 '\000') at printk.c:4
(gdb) i r ra
ra 0x80201080 0x80201080 <vprintfmt+1172>
(gdb)


proc.c
161 printk("SET [PID = %d COUNTER =
162 }
163 schedule();
164 }
165 else
166 {
167 if (next)
168 {
169 printk("\n\nswitch to [PID = %d
> 170 switch_to(next);
171 }
172 }
173 //printk("\nall0 = %d\n\n", all0);
174 //printk("\nmin_counter = %d\n\n", min_
remote Thread 1.1 In: schedule L170 PC: 0x802009dc
Breakpoint 3, schedule () at proc.c:139
(gdb) i r ra
ra 0x802007f4 0x802007f4 <do_timer+120>
(gdb) next
(gdb) i r ra
ra 0x802009dc 0x802009dc <schedule+468>
(gdb)

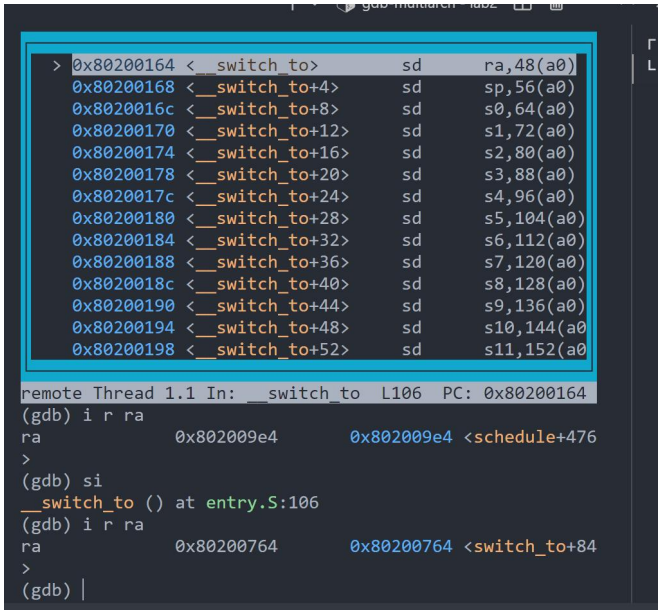

> 0x80200710 <switch_to> addi sp,sp,-48
0x80200714 <switch_to+4> sd ra,40(sp)
0x80200718 <switch_to+8> sd s0,32(sp)
0x8020071c <switch_to+12> addi s0,sp,48
0x80200720 <switch_to+16> sd a0,-40(s0)
B+ 0x80200724 <switch_to+20> auipc a5,0x5
0x80200728 <switch_to+24> addi a5,a5,-181
0x8020072c <switch_to+28> ld a5,0(a5)
0x80200730 <switch_to+32> ld a4,-40(s0)
0x80200734 <switch_to+36> beq a4,a5,0x80
0x80200738 <switch_to+40> auipc a5,0x5
0x8020073c <switch_to+44> addi a5,a5,-183
0x80200740 <switch_to+48> ld a5,0(a5)
0x80200744 <switch_to+52> sd a5,-24(s0)
remote Thread 1.1 In: switch_to L92 PC: 0x80200710
ra 0x802009dc 0x802009dc <schedule+468>
(gdb) layout asm
(gdb) si
switch_to (next=0x0) at proc.c:92
(gdb) i r ra
ra 0x802009e4 0x802009e4 <schedule+476>
(gdb)

至此又准备切换至另一线程，上一线程的调度暂停。

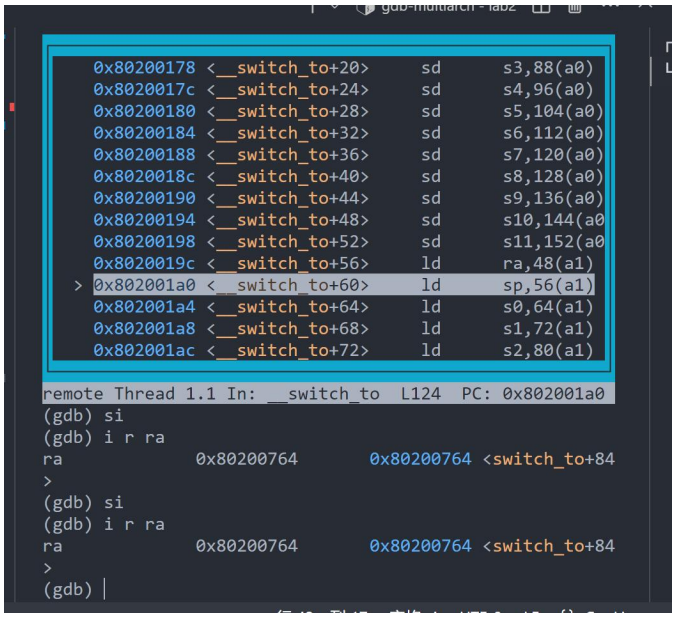


第一次调用之后的切换过程：

这次准备切换的线程(next)是第二次被调用，a0(prev)保存当前线程的运行地址<switch_to+84>，这和第一次调用是一样的。



而在将 a1（参数 next）切换为当前线程时，可以看到此时 a1 的 ra 变成了<switch_to+84>而不是__dummy，这是因为该线程在第一次调用结束准备切换至其他线程时将那时的地址<switch_to+84>保存在了 ra 中。



五、讨论心得

1、在实现实验里提到的优先级调度算法时，我一开始是参照课程上的介绍来编写的，即优先度高的任务优先执行，主要逻辑如下：

```
// for (int i = 1; i < NR_TASKS; i++)
// {
//     if (task[i]->state == TASK_RUNNING && task[i]->counter != 0 && task[i]->priority != 0)
//     {
//         all0 = 0;
//         if (task[i]->priority > max_p)
//         {
//             max_p = task[i]->priority;
//             next = task[i];
//         }
//     }
// }
```

但是发现测试样例却并没有将优先度最高的‘C’（priority = 88）排在第一位：

```
char priority_16[] = "FFFFFFFFFFFFKKKKKKKKKKGGGGGGGGGGCCCCCCCCDDDDDDDDJJJJJJJPPPPPPEEEEEMMMMBBBBNNNLLLOOHHI";
```

阅读 Linux v0.11 的调度算法发现该算法与纯粹的优先级优先并不一样。按照 linux v0.11 的算法，如果 counter 初始都为 0,那么将用 priority 对 counter 进行赋值，再选择 counter 最大的线程进行调度，这样也确实时优先级优先。但是本实验的测试初始都给 counter 赋了值，导致第一次调度不是按照优先级调度而是最长时间调度。 这应该是实验设置上的一个失误。

2、通过 gdb 调试，直观感受了 ra 寄存器的作用，即用于调用函数前的地址保存与调用函数后的地址恢复。

3、学习到了 CFLAG = ${CF} ${INCLUDE} -D <宏名>的用法。在 makefile 控制

的文件中使用#ifdef、#else、#endif 可以通过 makefile 决定想要进行编译的代码。

4、意外发现在 gdb 调试中如果打开了 asm 窗口，输入 f 可以将汇编代码变成对应位置的文件，这样调试会方便一些。