lab3

实验过程

4.3 线程调度实现

4.3.1 线程初始化

首先是task_init函数。其分为两个部分，一是idle的初始化，因为其代表了空闲，因此不需要设置pc和ra，priority和counter也设置为1即可。而对于地址空间的分配，我们只要调用mm.c中的kalloc函数，就能返回一个地址，我们将其强转成struct task就可以向内分配地址。同理，其他31个进程也可以这么设置，只不过ra需要设置在__dummy，sp指向kalloc返回的地址加上PGSIZE处。这样我们的task_init就设置完成了。

```
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 */
    // char *res = (char *)kalloc(10 * sizeof(char));
    uint64 pgNum = kalloc();
    idle = (struct task_struct*)pgNum;
    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

    /* YOUR CODE HERE */
    for (int i = 1; i < NR_TASKS; i++ ){
        pgNum = kalloc();
        task[i] = (struct task_struct*)pgNum;
        task[i]->state = TASK_RUNNING;
        task[i]->counter = 0;
        task[i]->priority = rand();
        task[i]->pid = i;
        task[i]->thread.ra = (uint64)__dummy;
        task[i]->thread.sp = pgNum + PGSIZE;
    }
    printk("...proc_init done!\n");
```

之后，我们将task_init和mm_init放入head.S中，将其初始化。

```
    # set sie[STIE] = 1
    addi t0, x0, 0b100000
    csrs sie, t0

    # init mm
    jal ra, mm_init

    # init task_struct
    jal ra, task_init

    # set first time interrupt
    jal ra, clock_set_next_event
```

之后，我们查看结果，可以看到成功初始化了。

```
Boot HART MEDELEG          : 0x000000000000b109
...mm_init done!
...proc_init done!
kernal is running!
kernal is running!
kernal [S] Supervisor Mode Timer Interrupt!
```

4.3.2 entry.S 添加 __dummy

由于创建一个新的线程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。 这个函数只要将sepc设置为__dummy，并使用sret返回即可。

如果是正常的进程切换，则是要将下一个函数的ra load出来，再返回switch_to到schedule再到do_timer最后返回时钟中断最后在_trap中设置sepc并使用sret返回，我们只是将这个过程提前而已。以下是实现的代码。

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

4.3.3 实现线程切换

首先是switch_to的实现。注意，当调用__switch_to转换reg的时候，记得task_struct的其他成员如pid等也要改变，不然就会出现reg变了，PID没变的情况，如下图所示。tmp也就是为了存储current现在的地址用于context switch.

```
next = 87ff9000
[PID = 0] is running. auto_inc_local_var = 1
[S] Supervisor Mode Timer Interrupt!
next = 87ff9000
[PID = 0] is running. auto_inc_local_var = 1
[S] Supervisor Mode Timer Interrupt!
next = 87ff9000
[PID = 0] is running. auto_inc_local_var = 1
[S] Supervisor Mode Timer Interrupt!
```

```
void switch_to(struct task_struct* next) {
    /* YOUR CODE HERE */
    if(current != next){
        struct task_struct *tmp = current;
        current = next;
        // here the cpu_switch_to: change reg
        __switch_to(tmp, next);
/*
```

之后是context switch的实现，也就是__switch_to。我们传入的是task_struct，距离thread_struct还要加一个offset，如下图所示。之后再将一个寄存器存入内存即可。

```
(gdb) x/16x next
0x87ffd000:     0x00000000      0x00000000      0x00000000      0x00000000
0x87ffd010:     0x00000000      0x00000000      0x00000004      0x00000000
0x87ffd020:     0x00000002      0x00000000      0x80200174      0x00000000
0x87ffd030:     0x87ffe000      0x00000000      0x00000000      0x00000000
```

```
    .globl __switch_to
__switch_to:
    # save state to prev process
    # YOUR CODE HERE
    li t0, 40
    add a0, a0, t0
    sd ra, 0(a0)
    sd sp, 8(a0)
    sd s0, 16(a0)
    sd s1, 24(a0)
    sd s2, 32(a0)
    sd s3, 40(a0)
    sd s4, 48(a0)
    sd s5, 56(a0)
    sd s6, 64(a0)
    sd s7, 72(a0)
    sd s8, 80(a0)
    sd s9, 88(a0)
    sd s10, 96(a0)
    sd s11, 104(a0)
```

```
    # restore state from next process
    # YOUR CODE HERE
    add a1, a1, t0

    ld ra, 0(a1)
    ld sp, 8(a1)
    ld s0, 16(a1)
    ld s1, 24(a1)
    ld s2, 32(a1)
    ld s3, 40(a1)
    ld s4, 48(a1)
    ld s5, 56(a1)
    ld s6, 64(a1)
    ld s7, 72(a1)
    ld s8, 80(a1)
    ld s9, 88(a1)
    ld s10, 96(a1)
    ld s11, 104(a1)

    ret
```

4.3.4 实现调度入口函数

我们要实现timer并在时钟中断中调用，实现并不难，但要注意时钟调用的位置，一定要clock_set_next_event之后再调用，因为第一次调用时不会返回do_timer，所以下一次时钟周期一定要提前设置好。

```
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{
        current->counter--;
        if(current->counter)
            return;
        else
            schedule();
    }
}
```

4.3.5 实现线程调度

4.3.5.1 短作业优先调度算法

SJF（shortest job first），简单明了，用min记录最小值，遍历线程指针数组task，TASK_RUNNING状态下counter最少的线程下一个执行。如果min没有变, 用rand()随机赋值counter,之后再重新进行调度。

```
void schedule(void) {
    /* YOUR CODE HERE */
    // switch_to(task[rand()%32]);
#ifdef SJF
    struct task_struct *next = idle;
    uint64 time_max_more = TIME_MAX + 1;
    uint64 min = time_max_more;
    while (1){
        for (int i = 1; i < NR_TASKS; i++){
            if (min > task[i]->counter && task[i]->counter && task[i]->state == TAS
K_RUNNING){
                next = task[i];
                min = task[i]->counter;
            }
        }
        if (min != time_max_more) break;
        if (min == time_max_more){
            for (int i = 1; i < NR_TASKS; i++){
                task[i]->counter = rand();
                printk("SET [PID = %d COUNTER = %d]\n", i, task[i]->counter);
            }
            // continue;
        }
    }
    // printk("next = %x\n", (uint64)next);
    switch_to(next);
#endif
```

```
SET [PID = 1 COUNTER = 10]
SET [PID = 2 COUNTER = 10]
SET [PID = 3 COUNTER = 5]
SET [PID = 4 COUNTER = 2]
next = 87ffb000
[PID = 4] is running. auto_inc_local_var = 1
current->counter = 2
[S] Supervisor Mode Timer Interrupt!
[PID = 4] is running. auto_inc_local_var = 2
current->counter = 1
[S] Supervisor Mode Timer Interrupt!
next = 87ffc000
[PID = 3] is running. auto_inc_local_var = 1
current->counter = 5
[S] Supervisor Mode Timer Interrupt!
[PID = 3] is running. auto_inc_local_var = 2
current->counter = 4
[S] Supervisor Mode Timer Interrupt!
[PID = 3] is running. auto_inc_local_var = 3
current->counter = 3
[S] Supervisor Mode Timer Interrupt!
[PID = 3] is running. auto_inc_local_var = 4
current->counter = 2
[S] Supervisor Mode Timer Interrupt!
[PID = 3] is running. auto_inc_local_var = 5
current->counter = 1
[S] Supervisor Mode Timer Interrupt!
next = 87ffe000
[PID = 1] is running. auto_inc_local_var = 1
current->counter = 10
```

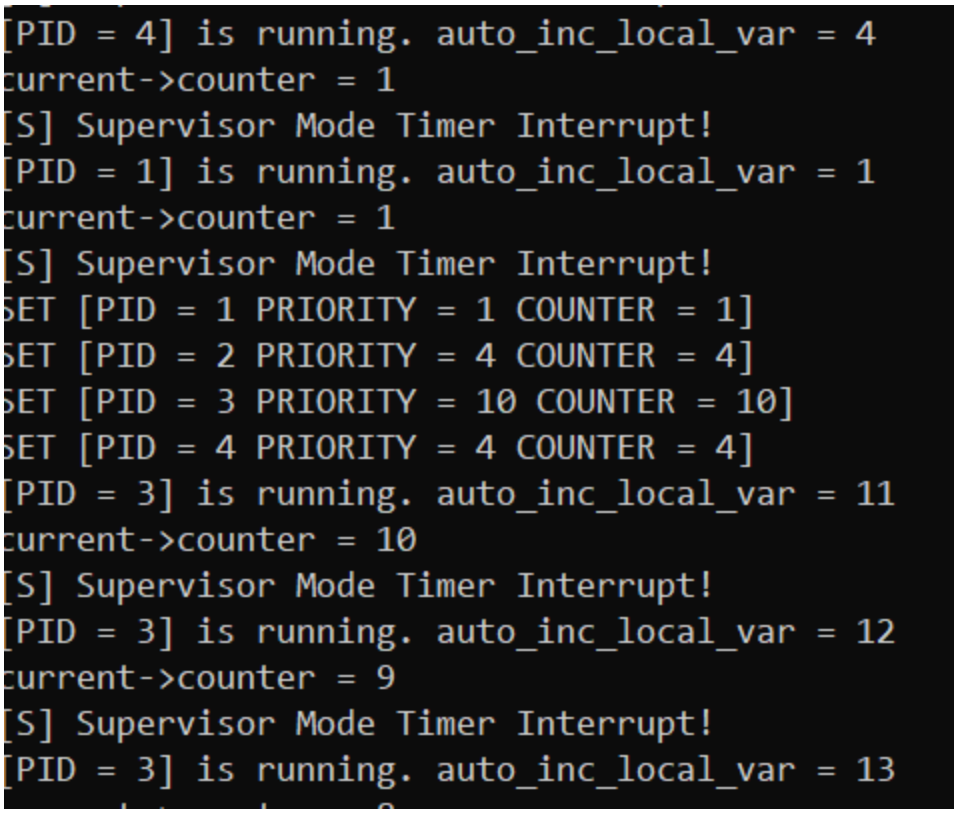
4.3.5.2 优先级调度算法

linux v0.11的做法，在最后counter赋值的时候变一下，用之前counter的一半加上priority即可。可以看到调度和set都是符合要求的。

```
#ifdef PRIORITY
    struct task_struct *next = idle;
    uint64 max = 0;
    while (1) {
        for (int i = 1; i < NR_TASKS; i++){
            if (max < task[i]->counter && task[i]->state == TASK_RUNNING){
                next = task[i];
                max = task[i]->counter;
            }
        }
        if (max) break;
        for (int i = 1; i < NR_TASKS; i++){
            task[i]->counter = (task[i]->counter >> 1) + task[i]->priority;
            printk("SET [PID = %d PRIORITY = %d COUNTER = %d]\n", i, task[i]->prior
ity, task[i]->counter);
        }
    }
    switch_to(next);
#endif
}
```

```
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]
[PID = 3] is running. auto_inc_local_var = 1
current->counter = 10
[S] Supervisor Mode Timer Interrupt!
[PID = 3] is running. auto_inc_local_var = 2
current->counter = 9
[S] Supervisor Mode Timer Interrupt!
[PID = 3] is running. auto_inc_local_var = 3
current->counter = 8
[S] Supervisor Mode Timer Interrupt!
[PID = 3] is running. auto_inc_local_var = 4
current->counter = 7
[S] Supervisor Mode Timer Interrupt!
[PID = 3] is running. auto_inc_local_var = 5
current->counter = 6
[S] Supervisor Mode Timer Interrupt!
[PID = 3] is running. auto_inc_local_var = 6
current->counter = 5
[S] Supervisor Mode Timer Interrupt!
[PID = 3] is running. auto_inc_local_var = 7
current->counter = 4
[S] Supervisor Mode Timer Interrupt!
[PID = 3] is running. auto_inc_local_var = 8
current->counter = 3
```

```
[PID = 3] is running. auto_inc_local_var = 10
current->counter = 1
[S] Supervisor Mode Timer Interrupt!
[PID = 2] is running. auto_inc_local_var = 1
current->counter = 4
[S] Supervisor Mode Timer Interrupt!
[PID = 2] is running. auto_inc_local_var = 2
current->counter = 3
[S] Supervisor Mode Timer Interrupt!
[PID = 2] is running. auto_inc_local_var = 3
current->counter = 2
[S] Supervisor Mode Timer Interrupt!
[PID = 2] is running. auto_inc_local_var = 4
current->counter = 1
[S] Supervisor Mode Timer Interrupt!
[PID = 4] is running. auto_inc_local_var = 1
current->counter = 4
[S] Supervisor Mode Timer Interrupt!
[PID = 4] is running. auto_inc_local_var = 2
current->counter = 3
[S] Supervisor Mode Timer Interrupt!
[PID = 4] is running. auto_inc_local_var = 3
current->counter = 2
[S] Supervisor Mode Timer Interrupt!
[PID = 4] is running. auto_inc_local_var = 4
current->counter = 1
[S] Supervisor Mode Timer Interrupt!
[PID = 1] is running. auto_inc_local_var = 1
```



思考题

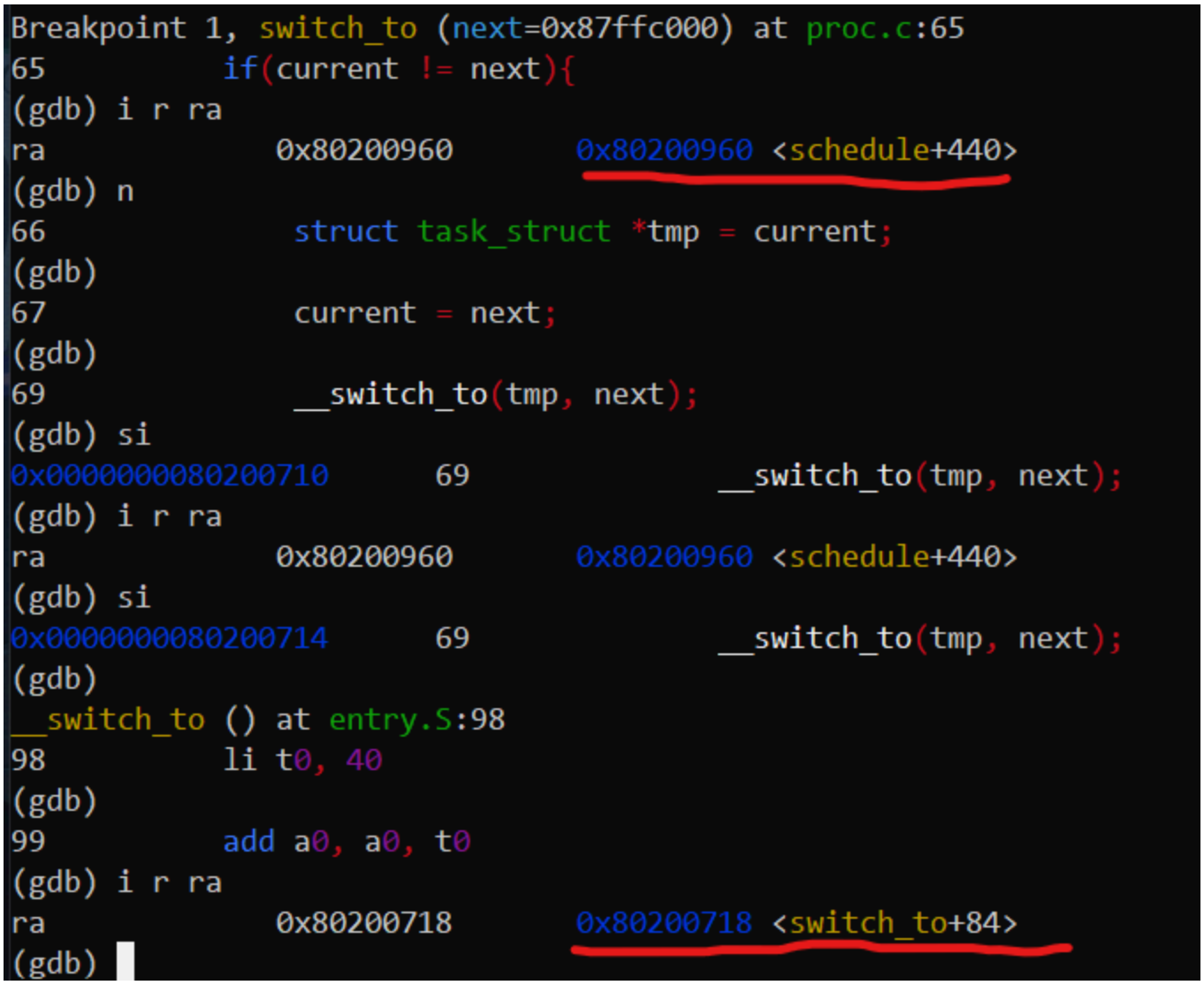
1. 在 RV64 中一共用 32 个通用寄存器, 为什么 context_switch 中只保存了14个?

因为context_swtich是被schedule调用的，因此其作为callee只需保存callee需要保存的寄存器即可。下图就展示了callee需要保存的reg，其中就有13个。而ra则是作为context_switch实现的核心，会随着进程改变而改变，因此旧值也需要保存。

Register	ABI Name	Description	Saver
x0	zero	Hard-wired zero	—
x1	ra	Return address	Caller
x2	sp	Stack pointer	Callee
x3	gp	Global pointer	—
x4	tp	Thread pointer	—
x5	t0	Temporary/alternate link register	Caller
x6–7	t1–2	Temporaries	Caller
x8	s0/fp	Saved register/frame pointer	Callee
x9	s1	Saved register	Callee
x10–11	a0–1	Function arguments/return values	Caller
x12–17	a2–7	Function arguments	Caller
x18–27	s2–11	Saved registers	Callee
x28–31	t3–6	Temporaries	Caller
f0–7	ft0–7	FP temporaries	Caller
f8–9	fs0–1	FP saved registers	Callee
f10–11	fa0–1	FP arguments/return values	Caller
f12–17	fa2–7	FP arguments	Caller
f18–27	fs2–11	FP saved registers	Callee
f28–31	ft8–11	FP temporaries	Caller

2. 用gdb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)

首先是第一次进入switch_to，可以看到ra是schedule的最后位置。再进入__switch_to，可以看到现在的ra是switch_to函数。

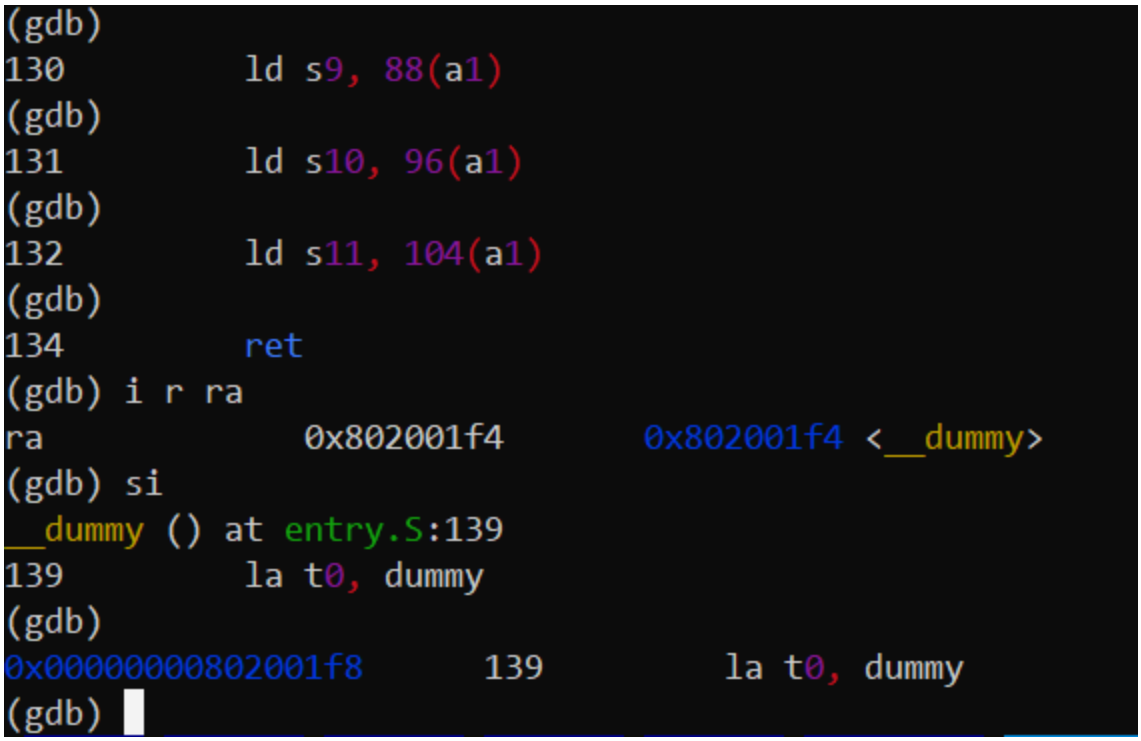


之后在__switch_to进行context switch之后，可以看到现在的ra就是新进程里的__dummy了。

经过__dummy之后, 由于sepc被设置为dummy, 并使用sret, pc就返回dummy循环, 直到下次时钟interrupt, 其从dummy中进入_traps, 这也是为什么下一次_traps存储的ra是dummy的原因.

sret返回sepc, ret返回pc.

从理论来说, __dummy中sepc设置成dummy, pc跳转到dummy, 相关的sp, a0/a1之类的也需要跟着改变, 但实际上由于dummy不需要参数, sp也是用的内核栈, 需要caller修改的寄存器都可以置0, 因此只需要修改pc即可.



再第二次进入switch_to，可以看到前一个进程的ra还在switch_to函数中。

```
0x0000000080200710      69              __switch_to(tmp, next);
(gdb) si
0x0000000080200714      69              __switch_to(tmp, next);
(gdb)
__switch_to () at entry.S:98
98          li t0, 40
(gdb)
99          add a0, a0, t0
(gdb)
100         sd ra, 0(a0)
(gdb)
101         sd sp, 8(a0)
(gdb)
102         sd s0, 16(a0)
(gdb)
103         sd s1, 24(a0)
(gdb)
104         sd s2, 32(a0)
(gdb)
105         sd s3, 40(a0)
(gdb)
106         sd s4, 48(a0)
(gdb) i r ra
ra             0x80200718       0x80200718 <switch_to+84>
(gdb)
```

之后进行完context switch之后，可以发现ra不是指向__dummy了，而是上次没做完的switch_to.

```
(gdb) i r ra
ra             0x80200718       0x80200718 <switch_to+84>
(gdb) si
switch_to (next=0x87ffd000) at proc.c:85
85      }
(gdb) si
0x000000008020071c      85      }
(gdb) si
0x0000000080200720      85      }
(gdb) i r ra
ra             0x80200960       0x80200960 <schedule+440>
(gdb) si
0x0000000080200724      85      }
```

switch_to出来后进入schedule的末尾，再进入do_timer，trap_handler,最后从_trap出来，load出下一个进程的ra,也就是dummy

```
0x0000000080200970      147     }
(gdb)
0x0000000080200790 in do_timer () at proc.c:99
99                  schedule();
(gdb) i r ra
ra             0x80200790       0x80200790 <do_timer+100>
(gdb) bt
```

```
0x000000008020079c      101     }
(gdb) i r ra
ra             0x80200ad8       0x80200ad8 <trap_handler+88>
(gdb) si
0x00000000802007a0      101     }
(gdb) si
0x00000000802007a4      101     }
(gdb) si
trap_handler (scause=9223372036854775813, sepc=2149582412) at trap.c:33
33      }
(gdb) i rra
Undefined info command: "rra".  Try "help info".
(gdb) i r ra
ra             0x80200ad8       0x80200ad8 <trap_handler+88>
(gdb) si
0x0000000080200adc      33      }
(gdb)
0x0000000080200ae0      33      }
(gdb)
0x0000000080200ae4      33      }
(gdb)
0x0000000080200ae8      33      }
(gdb)
_traps () at entry.S:52
52          ld t0, 256(sp)
(gdb) i r ra
ra             0x802000e4       0x802000e4 <_traps+152>
(gdb)
```

```
(gdb)
83          ld x29, 232(sp)
(gdb) i r ra
ra             0x802006c0       0x802006c0 <dummy+188>
(gdb) n
84          ld x30, 240(sp)
(gdb)
85          ld x31, 248(sp)
```

3. 运行课堂 demo 的 hello-lkm 代码, 回答下列问题:

a. 对运行结果进行截图, 展示同一进程内的线程哪些数据 share, 哪些不 share

- 首先，将其解压后，将路径修改成解压linux内核的路径：

```
all:
    make -C /home/drdg8/os22fall-stu/WSL2-Linux-Kernel-linux-msft-wsl-5.10.16.3 M=$
(PWD) modules


    # make -C /lib/modules/$(shell uname -r)/build M=$(PWD) clean
clean:
    make -C /home/drdg8/os22fall-stu/WSL2-Linux-Kernel-linux-msft-wsl-5.10.16.3 M=$
(PWD) clean
```

- 之后过程略过不谈，最后结果：

```
drdg8@LAPTOP-U2NC5HPH:~/os22fall-stu/hello-lkm-master$ sudo dmesg
[ 2808.989996] PROCESS   THREAD TSK=ffff888100230000    PID=1    STACK=ffffc9000000c000 COMM=init       MM=ffff888104fd8000     ACTIVE_MM=ffff888104fd8000     USER_SP=7ffc33cabfe8   USER_PC=206901
[ 2808.990330]           THREAD TSK=ffff888100a24880    PID=173  STACK=ffffc90000478000 COMM=init       MM=ffff888104fd8000     ACTIVE_MM=ffff888104fd8000     USER_SP=7f9042cd0e08   USER_PC=206901
[ 2808.990334] PROCESS   THREAD TSK=ffff888100230e80    PID=2    STACK=ffffc90000014000 COMM=kthreadd   MM=0    ACTIVE_MM=0     USER_SP=0      USER_PC=0
[ 2808.990352] PROCESS   THREAD TSK=ffff888100231d00    PID=3    STACK=ffffc90000040000 COMM=rcu_gp     MM=0    ACTIVE_MM=0     USER_SP=0      USER_PC=0
[ 2808.990371] PROCESS   THREAD TSK=ffff888100232b80    PID=4    STACK=ffffc90000048000 COMM=rcu_par_gp MM=0    ACTIVE_MM=0     USER_SP=0      USER_PC=0
[ 2808.990375] PROCESS   THREAD TSK=ffff888100233a00    PID=5    STACK=ffffc90000050000 COMM=kworker/0:0        MM=0    ACTIVE_MM=0     USER_SP=0      USER_PC=0
[ 2808.990393] PROCESS   THREAD TSK=ffff888100234880    PID=6    STACK=ffffc90000058000 COMM=kworker/0:0H       MM=0    ACTIVE_MM=0     USER_SP=0      USER_PC=0
[ 2808.990396] PROCESS   THREAD TSK=ffff888100236580    PID=8    STACK=ffffc90000068000 COMM=mm_percpu_wq       MM=0    ACTIVE_MM=0     USER_SP=0      USER_PC=0
[ 2808.990414] PROCESS   THREAD TSK=ffff8881002f0000    PID=9    STACK=ffffc90000070000 COMM=rcu_tasks_rude_    MM=0    ACTIVE_MM=0     USER_SP=0      USER_PC=0
[ 2808.990433] PROCESS   THREAD TSK=ffff8881002f0e80    PID=10   STACK=ffffc90000078000 COMM=rcu_tasks_trace    MM=0    ACTIVE_MM=0     USER_SP=0      USER_PC=0
[ 2808.990437] PROCESS   THREAD TSK=ffff8881002f1d00    PID=11   STACK=ffffc90000080000 COMM=ksoftirqd/0        MM=0    ACTIVE_MM=0     USER_SP=0      USER_PC=0
[ 2808.990440] PROCESS   THREAD TSK=ffff8881002f2b80    PID=12   STACK=ffffc90000088000 COMM=rcu_sched  MM=0    ACTIVE_MM=0     USER_SP=0      USER_PC=0
[ 2808.990442] PROCESS   THREAD TSK=ffff8881002f3a00    PID=13   STACK=ffffc90000090000 COMM=migration/0        MM=0    ACTIVE_MM=0     USER_SP=0      USER_PC=0
[ 2808.990461] PROCESS   THREAD TSK=ffff888100348000    PID=14   STACK=ffffc900000f0000 COMM=cpuhp/0    MM=0    ACTIVE_MM=0     USER_SP=0      USER_PC=0
[ 2808.990484] PROCESS   THREAD TSK=ffff888100348e80    PID=15   STACK=ffffc900000f8000 COMM=cpuhp/1    MM=0    ACTIVE_MM=0     USER_SP=0      USER_PC=0
[ 2808.990487] PROCESS   THREAD TSK=ffff888100349d00    PID=16   STACK=ffffc90000100000 COMM=migration/1        MM=0    ACTIVE_MM=0     USER_SP=0      USER_PC=0
[ 2808.990490] PROCESS   THREAD TSK=ffff88810034ab80    PID=17   STACK=ffffc90000108000 COMM=ksoftirqd/1        MM=0    ACTIVE_MM=0     USER_SP=0      USER_PC=0
[ 2808.990509] PROCESS   THREAD TSK=ffff88810034ba00    PID=18   STACK=ffffc90000110000 COMM=kworker/1:0        MM=0    ACTIVE_MM=0     USER_SP=0      USER_PC=0
[ 2808.990529] PROCESS   THREAD TSK=ffff88810034c880    PID=19   STACK=ffffc90000118000 COMM=kworker/1:0H       MM=0    ACTIVE_MM=0     USER_SP=0      USER_PC=0
[ 2808.990549] PROCESS   THREAD TSK=ffff88810034d700    PID=20   STACK=ffffc90000120000 COMM=cpuhp/2    MM=0    ACTIVE_MM=0     USER_SP=0      USER_PC=0
[ 2808.990553] PROCESS   THREAD TSK=ffff88810034e580    PID=21   STACK=ffffc90000128000 COMM=migration/2        MM=0    ACTIVE_MM=0     USER_SP=0      USER_PC=0
[ 2808.990556] PROCESS   THREAD TSK=ffff888100390000    PID=22   STACK=ffffc90000130000 COMM=ksoftirqd/2        MM=0    ACTIVE_MM=0     USER_SP=0      USER_PC=0
[ 2808.990575] PROCESS   THREAD TSK=ffff888100390e80    PID=23   STACK=ffffc90000138000 COMM=kworker/2:0        MM=0    ACTIVE_MM=0     USER_SP=0      USER_PC=0
[ 2808.990595] PROCESS   THREAD TSK=ffff888100391d00    PID=24   STACK=ffffc90000140000 COMM=kworker/2:0H       MM=0    ACTIVE_MM=0     USER_SP=0      USER_PC=0
[ 2808.990614] PROCESS   THREAD TSK=ffff888100392b80    PID=25   STACK=ffffc9000014c000 COMM=cpuhp/3    MM=0    ACTIVE_MM=0     USER_SP=0      USER_PC=0
[ 2808.990618] PROCESS   THREAD TSK=ffff888100393a00    PID=26   STACK=ffffc90000154000 COMM=migration/3        MM=0    ACTIVE_MM=0     USER_SP=0      USER_PC=0
[ 2808.990621] PROCESS   THREAD TSK=ffff888100394880    PID=27   STACK=ffffc9000015c000 COMM=ksoftirqd/3        MM=0    ACTIVE_MM=0     USER_SP=0      USER_PC=0
[ 2808.990640] PROCESS   THREAD TSK=ffff888100395700    PID=28   STACK=ffffc90000164000 COMM=kworker/3:0        MM=0    ACTIVE_MM=0     USER_SP=0      USER_PC=0
[ 2808.990660] PROCESS   THREAD TSK=ffff888100396580    PID=29   STACK=ffffc9000016c000 COMM=kworker/3:0H       MM=0    ACTIVE_MM=0     USER_SP=0      USER_PC=0
[ 2808.990679] PROCESS   THREAD TSK=ffff8881003d8000    PID=30   STACK=ffffc90000178000 COMM=cpuhp/4    MM=0    ACTIVE_MM=0     USER_SP=0      USER_PC=0
```

- 其中，看PID = 1/173的线程，TSK / PID / STACK / COMM是不共享的，MM与ACTIVE_MM是共享的。

b. 安装 lkm 和 remove lkm 的命令分别是什么? 对内核进行了哪些改动?

- 安装lkm和remove的命令：

```
drdg8@LAPTOP-U2NC5HPH:~/os22fall-stu/hello-lkm-master$ vim Makefile
drdg8@LAPTOP-U2NC5HPH:~/os22fall-stu/hello-lkm-master$ sudo insmod hello-lkm.ko
[sudo] password for drdg8:
```

```
master$ sudo rmmod hello_lkm
```

- 其将hello_lkm作为一个module insert into / remove from kernel

c. 使用哪个内核函数可以输出到内核 log?

- 使用dmesg，也就是debug message

```
drdg8@LAPTOP-U2NC5HPH:~/os22fall-stu/hello-lkm-master$ ls /proc/hello_lkm
/proc/hello_lkm
drdg8@LAPTOP-U2NC5HPH:~/os22fall-stu/hello-lkm-master$ sudo dmasg -c
sudo: dmasg: command not found
drdg8@LAPTOP-U2NC5HPH:~/os22fall-stu/hello-lkm-master$ sudo dmesg -C
drdg8@LAPTOP-U2NC5HPH:~/os22fall-stu/hello-lkm-master$ cat /proc/hello_lkm
drdg8@LAPTOP-U2NC5HPH:~/os22fall-stu/hello-lkm-master$ sudo dmasg
sudo: dmasg: command not found
drdg8@LAPTOP-U2NC5HPH:~/os22fall-stu/hello-lkm-master$ sudo dmesg
[ 2808.989996] PROCESS  THREAD TSK=ffff888100230000     PID=1    STACK=ffffc90000
```