Project2 A Simple Kernel 设计文档

中国科学院大学

李昊宸

2017K8009929044

***时钟中断、系统调用与 blocking sleep 设计流程***

1. 时钟中断处理的流程

时钟中断处理的流程：首先，是触发例外【第一种可能：CP0\_COUNT寄存器与CP0\_COMPARE寄存器内的值相等时，会触发一个时钟中断例外。第二种可能：系统调用】，进行第一级例外处理（硬件自动完成）到达中断处理函数入口exception\_handle\_entry，首先关中断（CLI）,然后保存用户上下文（SAVE\_CONTEXT(USER)），随后进行第二级例外处理（对CP0\_CAUSE寄存器中CAUSE\_EXCCODE字段的数值分析，确定跳转到哪个中断处理程序（exception\_handler[32]），如果是时钟中断就跳转到exception\_handler[0]（handle\_int），系统调用就跳转到exception\_handler[8]（handle\_syscall），其他的例外就跳转到handle\_other。在中断处理程序入口，经过第三季判断是否为时钟中断；在系统调用处理程序入口，判断具体是什么类型的系统调用。处理完例外后，进行内核调度（SAVE\_CONTEXT(KERNEL)，do\_schedule，RESTORE(KERNEL)），恢复用户上下文（RESTORE(USER)），开中断（STI），eret退出时钟中断。

1. 你所实现的时钟中断的处理流程中，如何处理 blocking sleep 的任务，你如何决定何时 唤醒 sleep 的任务

被do\_sleep的任务将被放入sleep\_queue睡眠队列里。每次进入时钟中断说明已经过了一个时间片的时间。这时中断会把计时器time\_elapsed增加一个时间片的长度，记录经过了多少个时钟周期。中断期间会进行任务调度，每次scheduler之前都会检查睡眠队列，把睡眠时长到达要求的睡眠时间的任务重新放回准备队列。

1. 你实现的时钟中断处理流程和系统调用处理流程有什么相同步骤，有什么不同步骤

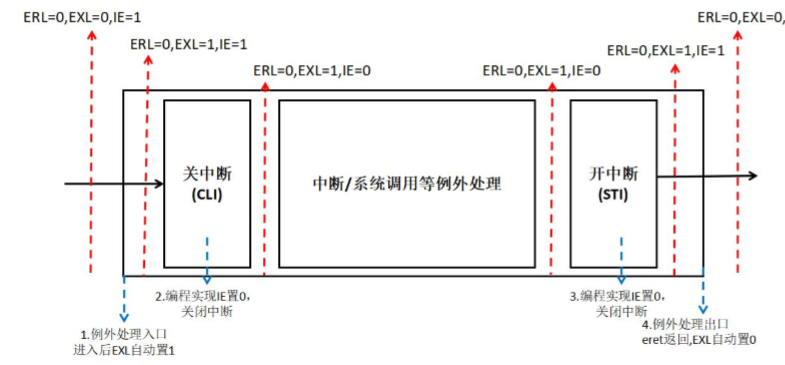
相同的步骤：硬件跳到相同的例外处理入口，然后关中断、保存用户上下文，根据例外码分级处理，之后会恢复用户上下文和开中断， eret 返回。

不同的步骤：时钟中断会进行任务调度，保存打印光标位置，重置 compare 和 count 寄存器，。系统调用会将 epc 加 4，调用其他的内核函数完成操作。

1. priority-based scheduler 的设计思路，包括在你实现的调度策略中优先级是怎么定义的， 何时给 task 赋予优先级，测试结果如何体现优先级的

两个优先级参数都定义在了PCB的结构体中：task\_priority和priority。task\_priority是任务优先级，由任务的性质决定，是一个定值，在初始化PCB的时候被从task\_info中的task\_priority赋用（在测试函数中人为的将所有的任务优先级赋值为1）。priority是总优先级，它的值与任务优先级和等待时间的和相关。在每一次进行任务调度和do\_block\_release的时候，都要将队列中所有的PCB的priority加1。入队的时候重新书写一个priority\_queue\_push函数，按照优先级从高到低的顺序插入到队列之中。这样的入队方式可以保证每次出队的时候的任务都是优先级最高的任务。测试时，调整task\_priority的值，可以发现任务优先级高的任务运行次数会比低的任务按比例增多，当所有任务优先级都是1的时候，相同规模的任务运行次数接近于相等。

1. 设计、实现或调试过程中遇到的问题和得到的经验
   1. 对于时钟中断，需要对内核态与用户态执行的任务是什么有清楚的理解。之前在写任务一和任务二时，直接把执行的测试程序放到了内核中执行。在后两个任务中，需要在用户的上下文部分执行测试程序，当通过时钟中断进入内核态时，做任务调度，中断处理，随后再返回用户态。也就是说，任务一和任务二执行的是非抢占式调度，任务的调度需要程序自己交出控制权。任务三和任务四实现的是任务本身没有调度权，程序运行完一个时间片后，会自动发起中断，进行任务之间的调度。
   2. 在实现系统调用时，起初不清楚CP0\_EPC寄存器的工作机制，导致运行时会一直在系统调用处死循环。查阅相关资料和学长交流后明白，需要在系统调用期间将EPC寄存器中的值加4，使任务返回时执行下一行命令。（此处需要注意系统调用是不是在转移指令延迟槽中）
   3. 时间片的大小。发现如下规律：如果时间片太小，会导致完整的时间不足以做程序本身的处理。如：时间片设置的过小，仅仅和系统调用和保存、恢复现场的总时间相似的话，将没有时间用于处理用户程序本身，导致用户程序运行会被卡死。如果时间片太长的话，会导致一个程序被长时间执行，而其他的程序始终处于未被执行的状态。通过一些测试，目前代码中测试的时间片是能在板卡上运行良好的时间片。
   4. 另外，为了避免产生bug，以及尽可能满足同一性，在做任务四的时候对任务三的重置COUNT和COMPARE寄存器的部分做了一定的修改：之前的设计是在irq\_timer()模块内使用SET\_CP0\_COUNT和SET\_CP0\_COMPARE宏函数来重置这两个寄存器，kernel\_context的ra寄存器放的是exception\_handler\_finish的地址。现在把重置寄存器模块直接放在exception\_handler\_entry后的frist\_run\_handle处，并把kernel\_context的ra寄存器的值也初始化为这里，这样可以在第一次执行程序时显式的重置两个寄存器的值。
   5. 优先级调度的实现过程中，起初想不到一个好的算法去衡量等待时间与起始优先级的关系。后来在与学长交流的过程中借鉴了这样的算法，问题确实得到了解决。
   6. 在初始化寄存器CP0\_STATUS时，不太确定初始化的值，因为涉及到第一次调度时的问题。



IE=1

这张图很好的解释了这一点。在初始化PCB时，我们要有这样的观念：最早运行的是main中的while循环，当时间片用完，时钟中断例外发生后，此时在内核中做的事情是把current\_running指针转向准备队列中优先级最高的那个PCB。转移完成后，需要恢复内核上下文，这个时候对应的是图中【中断/系统调用等例外处理】方块内部的过程，所以在初始化PCB的kernel\_context时需要把kernel\_context.cp0\_status初始化为0x10008002。然后中断处理完成，想从这个方块退出时，返回的位置应该是frist\_handle\_entry，所以需要把kernel\_context.reg[31]的值赋为这个标号的地址。返回例外处理程序的末尾，随后需要恢复用户上下文，此时对应的是右侧第三个红色虚线的位置，故初始化PCB的user\_context.cp0\_status 初始化为0x10080002，user\_context.cp0\_epc应该为测试函数的入口，开中断,eret，随后进入测试程序中运行。这样一来就完成了首次运行任务时，【模拟是从内核态退出】的过程。

* 1. 一定要记得，在初始化全部之后，SET\_CP0\_STATUS开中断！之前调试的时候怎么都发现程序一运行就陷入死循环，用gdb调试很多次后才发现，init\_exception部分关闭了中断之后再也没有开启。QAQ
  2. 另外，调试时候总是出一些莫名其妙的bug？？？比如说编译器在编译汇编指令时，会默认你在entry.S中调用c函数的时候已经把参数空间留了出来（比如说有两个参数，那么它默认从栈基址开始下数8个字节都是参数空间），这样导致如果不手动addiu sp， sp，-8的话会使栈访问越界到上一个PCB块的区域中，导致莫名其妙的错误（比如说任务跑一会之后就从任务队列中消失了），通过修改两处这样的c语言调用之后问题得到解决。

1. 部分代码

代码的总体框架与任务一任务二类似，这里只展示部分函数

* 1. 例外处理函数

NESTED(exception\_handler\_entry, 0, sp)

exception\_handler\_begin:

// TODO close interrupt

CLI

SAVE\_CONTEXT(USER)

// jmp exception\_handler[i] which decided by CP0\_CAUSE

// Leve2 exception Handler.

mfc0 k0, CP0\_CAUSE

nop

andi k0, k0, CAUSE\_EXCCODE

la k1, exception\_handler

add k0, k0, k1

lw k0, 0(k0)

jr k0

exception\_handler\_end:

END(exception\_handler\_entry)

NESTED(handle\_int, 0, sp)

// interrupt handler

// Leve3 exception Handler.

mfc0 a0, CP0\_STATUS

nop

mfc0 a1, CP0\_CAUSE

nop

addiu sp, sp, -8

jal interrupt\_helper

addiu sp, sp, 8 //十分重要！因为编译器在调用c函数时默认

已经留出来传参空间

first\_run\_handle:

//每次发起中断，完成调度之后都给用户重新分配10000的时间

//reset count and compare reg

mtc0 zero, CP0\_COUNT

nop

li k0, 100000

mtc0 k0, CP0\_COMPARE

nop

j exception\_handler\_finish

END(handle\_int)

NESTED(handle\_syscall, 0, sp)

// system call handler

add a3, a2, zero

add a2, a1, zero

add a1, a0, zero

add a0, v0, zero

addiu sp, sp, -16

jal system\_call\_helper

addiu sp, sp, 16 //同样要留出参数空间

j exception\_handler\_finish

END(handle\_syscall)

NESTED(handle\_other, 0, sp)

// other exception handler

j exception\_handler\_finish

END(handle\_other)

LEAF(exception\_handler\_finish)

RESTORE\_CONTEXT(USER)

STI

eret

END(exception\_handler\_finish)

* 1. 系统调用函数

LEAF(invoke\_syscall)

// TODO syscall entry

add v0, a0, zero

add a0, a1, zero

add a1, a2, zero

add a2, a3, zero

syscall //自动跳转到例外处理入口

jr ra

END(invoke\_syscall)

//调Ì用syscall时，v0是系统调用号

//a0,a1,a2是参数

void system\_call\_helper(int fn, int arg1, int arg2, int arg3)

{

// syscall[fn](arg1, arg2, arg3)

current\_running->user\_context.cp0\_epc = current\_running->user\_context.cp0\_epc + 4;

//特别重要！epc会保存出现例外时的pc值，此时保存的是syscall的地址，不+4会陷入死循环

syscall[fn](arg1, arg2, arg3);

}

* 1. 初始化函数

static void init\_exception\_handler()

{

int i;

exception\_handler[0] = (uint32\_t)&handle\_int;

for(i = 1; i < 32; i++)

exception\_handler[i] = (uint32\_t)&handle\_other;

exception\_handler[8] = (uint32\_t)&handle\_syscall;

}

static void init\_exception()

{

init\_exception\_handler();

// 1. Get CP0\_STATUS

initial\_cp0\_status = GET\_CP0\_STATUS();

// 2. Disable all interrupt

initial\_cp0\_status |= 0x10008001;

initial\_cp0\_status ^= 0x1;

SET\_CP0\_STATUS(initial\_cp0\_status);

initial\_cp0\_status |= 0x1;

// 3. Copy the level 2 exception handling code to 0x80000180

memcpy((void \*)0x80000180, exception\_handler\_entry, exception\_handler\_end-exception\_handler\_begin);

memcpy((void \*)0xbfc00380, exception\_handler\_entry, exception\_handler\_end-exception\_handler\_begin);

// 4. reset CP0\_COMPARE & CP0\_COUNT register

SET\_CP0\_COUNT(0);

SET\_CP0\_COMPARE(100000);

}

static void init\_syscall(void)

{

// init system call table.

int i;

for(i = 0; i < NUM\_SYSCALLS; i++)

syscall[i] = (int (\*)())&sys\_other; //int (\*syscall[NUM\_SYSCALLS])();

syscall[SYSCALL\_SLEEP] = (int (\*)())&do\_sleep;

syscall[SYSCALL\_BLOCK] = (int (\*)())&do\_block;

syscall[SYSCALL\_UNBLOCK\_ONE] = (int (\*)())&do\_unblock\_one;

syscall[SYSCALL\_UNBLOCK\_ALL] = (int (\*)())&do\_unblock\_all;

syscall[SYSCALL\_WRITE] = (int (\*)())&screen\_write;

syscall[SYSCALL\_CURSOR] = (int (\*)())&screen\_move\_cursor;

syscall[SYSCALL\_REFLUSH] = (int (\*)())&screen\_reflush;

syscall[SYSCALL\_MUTEX\_LOCK\_INIT] = (int (\*)())&do\_mutex\_lock\_init;

syscall[SYSCALL\_MUTEX\_LOCK\_ACQUIRE] = (int (\*)())&do\_mutex\_lock\_acquire;

syscall[SYSCALL\_MUTEX\_LOCK\_RELEASE] = (int (\*)())&do\_mutex\_lock\_release;

}

* 1. 中断处理函数

static void irq\_timer()

{

// TODO clock interrupt handler.

// scheduler, time counter in here to do, emmmmmm maybe.

screen\_reflush();

time\_elapsed += 100000;

current\_running->cursor\_x = screen\_cursor\_x;

current\_running->cursor\_y = screen\_cursor\_y;

do\_scheduler();

screen\_cursor\_x = current\_running->cursor\_x;

screen\_cursor\_y = current\_running->cursor\_y; //系统中断调度

return;

}

void interrupt\_helper(uint32\_t status, uint32\_t cause)

{

// TODO interrupt handler.

// Leve3 exception Handler.

// read CP0 register to analyze the type of interrupt.

uint32\_t interrupt\_kind = status & cause & 0x0000ff00;

if(interrupt\_kind & 0x00008000)//时钟中断?

irq\_timer();

else

other\_exception\_handler();

return;

}

void other\_exception\_handler()

{

// TODO other exception handler

time\_elapsed += 100000;

return;

}

* 1. 调度

static void check\_sleeping()

{

pcb\_t \*pcb\_sleep\_head = sleep\_queue.head; //取出睡眠队列的头

uint32\_t time = get\_timer();

if(pcb\_sleep\_head != NULL) //一直检测到队尾

{

//如果休眠时间已到

if(time - pcb\_sleep\_head->begin\_time >= pcb\_sleep\_head->sleep\_time)

{

pcb\_t \*out\_queue = (pcb\_t \*)queue\_remove(&sleep\_queue, (void \*)pcb\_sleep\_head);

/\* remove this item and return next item \*/

//注意，是&sleep\_queue不是sleep\_queue！

pcb\_sleep\_head->sleep\_time = 0;

pcb\_sleep\_head->status = TASK\_READY;

priority\_queue\_push(&ready\_queue, (void \*)pcb\_sleep\_head);

pcb\_sleep\_head = out\_queue;

}

else

pcb\_sleep\_head = pcb\_sleep\_head->next;

}

}

void scheduler(void)

{

// TODO schedule

// Modify the current\_running pointer.

check\_sleeping();

pcb\_t \*next\_pcb, \*p;

if(queue\_is\_empty(&ready\_queue))

next\_pcb = current\_running;

else

next\_pcb = (pcb\_t \*)queue\_dequeue(&ready\_queue); //当队列为空时，反复运行当前pcb

if(current\_running->status != TASK\_BLOCKED && next\_pcb != current\_running)

{

current\_running->status = TASK\_READY;//如果不是被阻塞，就回到队列中

if(current\_running->pid != 1) //不让启动pcb回到队列

{

priority\_queue\_push(&ready\_queue, current\_running);

}

}

current\_running = next\_pcb;

current\_running->status = TASK\_RUNNING;

current\_running->priority = current\_running->task\_priority;//优先级重置

p = (pcb\_t \*)ready\_queue.head;

while(p != NULL)

{

p->priority += 1; //基于等待时间提升优先级

p = p->next;

}

return ;

}

void do\_sleep(uint32\_t sleep\_time)

{

// TODO sleep(seconds)

current\_running->status = TASK\_BLOCKED;

current\_running->begin\_time = get\_timer();

current\_running->sleep\_time = sleep\_time;

queue\_push(&sleep\_queue,(void \*)current\_running);

do\_scheduler();

}

void do\_block(queue\_t \*queue)

{

// block the current\_running task into the queue

current\_running->status = TASK\_BLOCKED;

priority\_queue\_push(queue, (void \*)current\_running);

do\_scheduler();

}

void do\_unblock\_one(queue\_t \*queue)

{

// unblock the head task from the queue

pcb\_t \*p = (pcb\_t \*)(queue->head);

while(p != NULL)

{

p->priority += 1;//基于等待时间提升优先级

p = p->next;

}

pcb\_t \*block\_head = (pcb\_t \*)queue\_dequeue(queue);

block\_head->status = TASK\_READY;

priority\_queue\_push(&ready\_queue, block\_head);

}

void do\_unblock\_all(queue\_t \*queue)

{

// unblock all task in the queue

pcb\_t \*block\_list;

while(!queue\_is\_empty(queue))

{

block\_list = (pcb\_t \*)queue\_dequeue(queue);

block\_list->status = TASK\_READY;

priority\_queue\_push(&ready\_queue, block\_list);

}

}

void priority\_queue\_push(queue\_t \*queue, void \*item)

{

item\_t \*\_item = (item\_t \*)item;

/\* queue is empty \*/

if (queue->head == NULL)

{

queue->head = item;

queue->tail = item;

\_item->next = NULL;

\_item->prev = NULL;

}

else

{

if(((item\_t \*)(queue->head))->priority < \_item->priority) //队头优先级低于新加入的优先级

{

\_item->next = queue->head;

\_item->prev = NULL;

((item\_t \*)(queue->head))->prev = item;

queue->head = item;

}

else

{

item\_t \*pos = queue->head;

while(pos->next != NULL && ((item\_t \*)(pos->next))->priority >= \_item->priority)

{

pos = pos->next; } //直到找到比新加入的优先级还低的PCB

if(pos->next == NULL) //如果没有

queue->tail = item;

\_item->next = pos->next;

\_item->prev = pos;

pos->next = item;

}

}

}

***Bonus 设计思路***

1. 如何处理一个进程获取多把锁

对于每一把锁，在init的时候都赋给它一个id号，对应这个id号会有一个阻塞队列。试图获取这把锁的任务，如果该锁锁住，就进入这个id的阻塞队列。释放该锁的时候，也是从这个阻塞队列中取出优先级最高的加入准备队列。一个任务获取多把锁不可能同时进行，所以如果都能获得就直接执行，有哪个不能获得就进入这个任务的阻塞队列。

1. 如何处理多个进程获取一把锁

多个进程获取一把锁就是只有一个进程能够获得，其他的进程都会进入对应的等待队列，并且按照优先级排序。等锁释放之后，最高优先级的任务 unblock，获得锁。同时，为了避免低优先级的任务无法获得到锁，一直阻塞的状态，在每次 unblock 的时候，会对相应的阻塞队列的任务的优先级全部加 1，实现优先级的改变，避免低优先级任务阻塞死。

1. 你的测试用例和结果介绍

测试例子中有三个任务：task\_lock1是先获取锁1，占用锁运行并打印，然后获取锁2，运行并打印，然后释放锁2，打印，最后释放锁1。task\_lock2是获取锁1，打印，释放锁1，然后获取锁2，打印，释放锁2。task\_lock3只有获取锁1，打印，释放锁1.

结果和预期相同，第一个任务先获得锁1后，后两个任务被阻塞；任务一用完锁1锁2释放后，任务二随后获取锁1，如此下去。不过测试的时候由于时间片太大，运行速度太快，需要将时间片适度调小之后才能看见正确的结果。

1. 设计、实现或调试过程中遇到的问题和得到的经验

虽然完成了锁的设计，但是不太了解怎么设计测试程序。从学长处获得帮助之后，成功运行了测试用例。

1. 部分代码
   1. lock方法（具体的do函数见上部分）

void do\_mutex\_lock\_init(mutex\_lock\_t \*lock)

{

lock->status = UNLOCKED;

lock->lock\_id = lock\_queue\_id++;

}

void do\_mutex\_lock\_acquire(mutex\_lock\_t \*lock)

{

if(lock->status == LOCKED)

{

do\_block(&block\_queue[lock->lock\_id]);

}

lock->status = LOCKED;

}

void do\_mutex\_lock\_release(mutex\_lock\_t \*lock)

{

if(!queue\_is\_empty(&block\_queue[lock->lock\_id]))

{

do\_unblock\_one(&block\_queue[lock->lock\_id]);

lock->status = UNLOCKED; //先把阻塞队列中的第一个进入准备队列，然后把锁解开，这样准备队列

//轮到它执行时才能把锁拿到

}

else

lock->status = UNLOCKED;

}

* 1. 测试用例（为了阅读方便就不更改字体了）

void lock\_task1(void)

{

int print\_location = 3, j = 1;

while (1)

{

j++;

int i;

if (!is\_init)

{

is\_init = TRUE;

#ifdef SPIN\_LOCK

spin\_lock\_init(&spin\_lock);

#endif

#ifdef MUTEX\_LOCK

mutex\_lock\_init(&mutex\_lock\_o);

mutex\_lock\_init(&mutex\_lock\_s);

#endif

}

sys\_move\_cursor(1, print\_location);

printf("%s", blank);

sys\_move\_cursor(1, print\_location);

printf("> [TASK1] Applying for a lock1.(%d)\n", j);

#ifdef SPIN\_LOCK

spin\_lock\_acquire(&spin\_lock);

#endif

#ifdef MUTEX\_LOCK

mutex\_lock\_acquire(&mutex\_lock\_o);

#endif

for (i = 0; i < 30; i++)

{

sys\_move\_cursor(1, print\_location);

printf("> [TASK1] Has acquired lock1 and running.(%d)\n", i);

}

sys\_move\_cursor(1, print\_location);

printf("%s", blank);

sys\_move\_cursor(1, print\_location);

printf("> [TASK1] Applying for a lock2.(%d)\n", j);

#ifdef SPIN\_LOCK

spin\_lock\_acquire(&spin\_lock);

#endif

#ifdef MUTEX\_LOCK

mutex\_lock\_acquire(&mutex\_lock\_s);

#endif

for (i = 0; i < 100; i++)

{

sys\_move\_cursor(1, print\_location);

printf("> [TASK1] Has acquired lock2 and running.(%d)\n", i);

}

sys\_move\_cursor(1, print\_location);

printf("%s", blank);

sys\_move\_cursor(1, print\_location);

printf("> [TASK1] Has acquired lock2 and exited.(%d)\n", j);

#ifdef SPIN\_LOCK

spin\_lock\_release(&spin\_lock);

#endif

#ifdef MUTEX\_LOCK

mutex\_lock\_release(&mutex\_lock\_s);

#endif

sys\_move\_cursor(1, print\_location);

printf("%s", blank);

sys\_move\_cursor(1, print\_location);

printf("> [TASK1] Has acquired lock1 and exited.(%d)\n", j);

#ifdef SPIN\_LOCK

spin\_lock\_release(&spin\_lock);

#endif

#ifdef MUTEX\_LOCK

mutex\_lock\_release(&mutex\_lock\_o);

#endif

}

}

void lock\_task2(void)

{

int print\_location = 4, j = 1;

while (1)

{

j++;

int i;

if (!is\_init)

{

is\_init = TRUE;

#ifdef SPIN\_LOCK

spin\_lock\_init(&spin\_lock);

#endif

#ifdef MUTEX\_LOCK

mutex\_lock\_init(&mutex\_lock\_o);

mutex\_lock\_init(&mutex\_lock\_s);

#endif

}

sys\_move\_cursor(1, print\_location);

printf("%s", blank);

sys\_move\_cursor(1, print\_location);

printf("> [TASK2] Applying for a lock1.(%d)\n", j);

#ifdef SPIN\_LOCK

spin\_lock\_acquire(&spin\_lock);

#endif

#ifdef MUTEX\_LOCK

mutex\_lock\_acquire(&mutex\_lock\_o);

#endif

for (i = 0; i < 20; i++)

{

sys\_move\_cursor(1, print\_location);

printf("> [TASK2] Has acquired lock1 and running.(%d)\n", i);

//do\_scheduler();

}

sys\_move\_cursor(1, print\_location);

printf("%s", blank);

sys\_move\_cursor(1, print\_location);

printf("> [TASK2] Has acquired lock1 and exited.(%d)\n", j);

#ifdef SPIN\_LOCK

spin\_lock\_release(&spin\_lock);

#endif

#ifdef MUTEX\_LOCK

mutex\_lock\_release(&mutex\_lock\_o);

#endif

//do\_scheduler();

sys\_move\_cursor(1, print\_location);

printf("%s", blank);

sys\_move\_cursor(1, print\_location);

printf("> [TASK2] Applying for a lock2.(%d)\n", j);

//do\_scheduler();

#ifdef SPIN\_LOCK

spin\_lock\_acquire(&spin\_lock);

#endif

#ifdef MUTEX\_LOCK

mutex\_lock\_acquire(&mutex\_lock\_s);

#endif

for (i = 0; i < 20; i++)

{

sys\_move\_cursor(1, print\_location);

printf("> [TASK2] Has acquired lock2 and running.(%d)\n", i);

//do\_scheduler();

}

sys\_move\_cursor(1, print\_location);

printf("%s", blank);

sys\_move\_cursor(1, print\_location);

printf("> [TASK2] Has acquired lock2 and exited.(%d)\n", j);

#ifdef SPIN\_LOCK

spin\_lock\_release(&spin\_lock);

#endif

#ifdef MUTEX\_LOCK

mutex\_lock\_release(&mutex\_lock\_s);

#endif

//do\_scheduler();

}

}

void lock\_task3(void)

{

int print\_location = 5, j = 1;

while (1)

{

j++;

int i;

if (!is\_init)

{

is\_init = TRUE;

#ifdef SPIN\_LOCK

spin\_lock\_init(&spin\_lock);

#endif

#ifdef MUTEX\_LOCK

mutex\_lock\_init(&mutex\_lock\_o);

mutex\_lock\_init(&mutex\_lock\_s);

#endif

}

sys\_move\_cursor(1, print\_location);

printf("%s", blank);

sys\_move\_cursor(1, print\_location);

printf("> [TASK3] Applying for a lock1.(%d)\n", j);

//do\_scheduler();

#ifdef SPIN\_LOCK

spin\_lock\_acquire(&spin\_lock);

#endif

#ifdef MUTEX\_LOCK

mutex\_lock\_acquire(&mutex\_lock\_o);

#endif

for (i = 0; i < 20; i++)

{

sys\_move\_cursor(1, print\_location);

printf("> [TASK3] Has acquired lock1 and running.(%d)\n", i);

//do\_scheduler();

}

sys\_move\_cursor(1, print\_location);

printf("%s", blank);

sys\_move\_cursor(1, print\_location);

printf("> [TASK3] Has acquired lock1 and exited.(%d)\n", j);

#ifdef SPIN\_LOCK

spin\_lock\_release(&spin\_lock);

#endif

#ifdef MUTEX\_LOCK

mutex\_lock\_release(&mutex\_lock\_o);

#endif

//do\_scheduler();

}

}