Project2 A Simple Kernel 设计文档

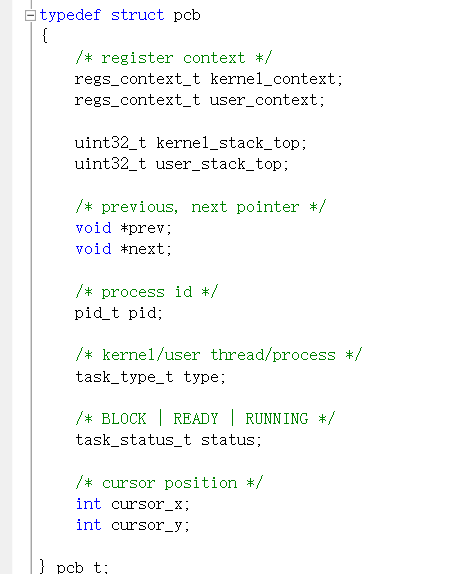
中国科学院大学

李昊宸

2017K8009929044

***任务启动与 Context Switching 设计流程***

1. PCB包含的信息



内核态的上下文

用户态的上下文

内核态的栈顶

用户态的栈顶

前驱

后继

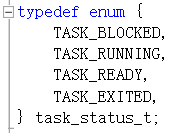
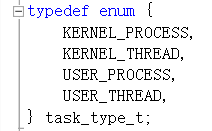
进程号

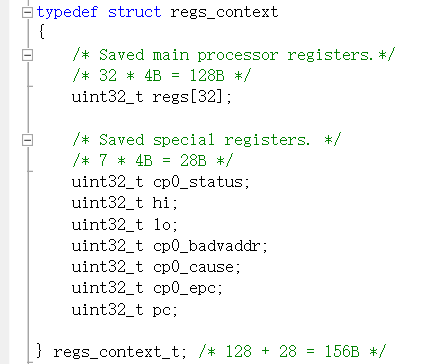
任务类型

任务状态

光标横坐标位置

光标纵坐标位置





协寄存器

处理器状态和控制寄存器

记录最近一次地址相关异常的地址

保存上一次异常原因

保存上一次异常时的程序计数器

1. 如何启动一个 task，例如如何获得 task 的入口地址，启动时需要设置哪些寄存器

非抢占式调度时，一个task启动是在main里调用do\_scheduler函数，进行任务调度。do\_scheduler函数第一步先保存现场（SAVE\_CONTEXT），先把目前current\_running指针（指向的是当前PCB的头地址）的地址赋给k0寄存器，因为k0寄存器是给操作系统保留的寄存器，不用保存里面的值。此时k0中存储的是当前PCB的头地址，对应头地址偏移量为kernel（为0）的部分是kernel\_context，将所有的寄存器（除去k0和k1）以及cp0协寄存器保存到对应的位置。然后调用scheduler函数，修改current\_running指针指向下一个PCB的头地址。之后再恢复现场（RESTORE\_CONTEXT），先通过当前的current\_running指针指向的地址恢复所有的寄存器（除了k0和k1寄存器）的内容，其中ra寄存器恢复的是任务的入口地址。这样，再通过一步jr ra指令就可以使pc跳转到切换PCB后的任务。

对于初始化PCB，需要将context部分全部清零（将寄存器全部清零），29 号 sp 寄存器初始化为分配的栈帧基址，31 号寄存器初始化为函数入口地址。具体实现方式见代码段。另外，考虑到第一次执行scheduler时会将current\_running指针修改，我们在PCB数组的最前端（0号）设置一个启动用PCB，将其状态设置为TASK\_RUNNING，这样可使第一次调用scheduler时自然的将current\_running指针设置为第一个task的PCB头地址。

1. context switching 时保存哪些寄存器，保存在内存什么位置，使得进程再切换回来后能 正常运行

上下文转换需要保存和恢复除了 k0 和 k1 的 30 个通用寄存器，以及 cp0\_status、hi、lo、cp0\_badvaddr、cp0\_cause 和 cp0\_cause 寄存器。

1. 设计、实现或调试过程中遇到的问题和得到的经验
   1. 这次出现的第一个问题是createimage的部分。由于之前没有考虑到p\_filesz与p\_memsz之间的关系，导致在这一部分面对较大的kernel时编译image出现了问题。查阅许多资料后发现，p\_filesz指文件在外存中的大小，p\_memsz指文件在内存中的大小。image最终是要被拷贝进入内存的，如果没有padding，实际运行中会把后续文件的空间覆盖掉。于是在书写createimage时需要首先拷贝p\_filesz部分，然后再把剩余的部分设置为0，直到padding到p\_memsz的大小为止。
   2. 每一个PCB第一次执行时道理上应该只把目标任务入口地址放进ra，随后跳转即可。为了达到同等的效果，我们在初始化PCB的时候把上下文（xx\_context）部分直接memset为0，仅给ra赋值入口地址，sp赋值栈顶基址，这样在第一次restore\_context时就可以把除了这两个寄存器之外的寄存器（除去k0和k1）全部初始化为0。
   3. 关于汇编.macro的宏定义，一般有 .macro 库名称 实参名,调用实参名的时候需要在前面加 \ ，用来与标号作区分。具体例子见部分代码。

另外，entry.S中定义的库函数在sched.c中被调用（do\_scheduler），因为在sched.c中全局区域声明了current\_running指针，所以在调用SAVE\_CONTEXT和RESTORE\_CONTEXT时可以直接使用current\_running这个标号，并且因为它只是一个标号，不是真实的实参，不需要声明引用文件（#include）

1. 部分代码
   1. init\_PCB

#define STACK\_MAX 0xa0f00000

#define STACK\_MIN 0xa0d00000

#define PCB\_STACK\_SIZE 0x10000 //process的大小

uint32\_t stack\_top = STACK\_MAX;

//uint32\_t initial\_cp0\_status;

//uint32\_t exception\_handler[32];

queue\_t ready\_queue, block\_queue[NUM\_MAX\_TASK];

static void init\_pcb()

{

int i, cur\_queue\_id;

queue\_init(&ready\_queue);

for(i = 0; i < NUM\_MAX\_TASK; i++)

queue\_init(&block\_queue[i]); //初始化阻塞队列

pcb[0].pid = process\_id++; //将队伍头初始化，以便第一次调用scheduler时使current\_running指向第一个任务

pcb[0].status = TASK\_RUNNING;

cur\_queue\_id = 1;

//scheduler1 task1

for(i = 0; i < num\_sched1\_tasks; i++, cur\_queue\_id++) //把等待队列初始化

{

memset(&pcb[cur\_queue\_id].kernel\_context,0, sizeof(pcb[cur\_queue\_id].kernel\_context));

memset(&pcb[cur\_queue\_id].user\_context,0, sizeof(pcb[cur\_queue\_id].user\_context));

pcb[cur\_queue\_id].pid = process\_id++;

// stack

pcb[cur\_queue\_id].kernel\_stack\_top = stack\_top; //具体每个进程可使用的栈空间都是0x10000

pcb[cur\_queue\_id].kernel\_context.regs[29] = stack\_top;

stack\_top -= PCB\_STACK\_SIZE;

pcb[cur\_queue\_id].kernel\_context.regs[31] = sched1\_tasks[i]->entry\_point;

pcb[cur\_queue\_id].kernel\_context.cp0\_status = 0x10008000;

pcb[cur\_queue\_id].kernel\_context.cp0\_epc = sched1\_tasks[i]->entry\_point;

pcb[cur\_queue\_id].user\_stack\_top = stack\_top;

pcb[cur\_queue\_id].user\_context.regs[29] = stack\_top;

stack\_top -= PCB\_STACK\_SIZE;

pcb[cur\_queue\_id].type = sched1\_tasks[i]->type;

pcb[cur\_queue\_id].status = TASK\_READY;

queue\_push(&ready\_queue, (void \*)&pcb[cur\_queue\_id]);

}

//lock1 task2

for(i = 0; i < num\_lock\_tasks; i++, cur\_queue\_id++)

{

memset(&pcb[cur\_queue\_id].kernel\_context,0, sizeof(pcb[cur\_queue\_id].kernel\_context));

memset(&pcb[cur\_queue\_id].user\_context,0, sizeof(pcb[cur\_queue\_id].user\_context));

pcb[cur\_queue\_id].pid = process\_id++;

//stack

pcb[cur\_queue\_id].kernel\_stack\_top = stack\_top;

pcb[cur\_queue\_id].kernel\_context.regs[29] = stack\_top;

stack\_top -= PCB\_STACK\_SIZE;

pcb[cur\_queue\_id].kernel\_context.regs[31] = lock\_tasks[i]->entry\_point;

pcb[cur\_queue\_id].kernel\_context.cp0\_status = 0x10008000;

pcb[cur\_queue\_id].kernel\_context.cp0\_epc = lock\_tasks[i]->entry\_point;

pcb[cur\_queue\_id].user\_stack\_top = stack\_top;

pcb[cur\_queue\_id].user\_context.regs[29] = stack\_top;

stack\_top -= PCB\_STACK\_SIZE;

pcb[cur\_queue\_id].type = lock\_tasks[i]->type;

pcb[cur\_queue\_id].status = TASK\_READY;

queue\_push(&ready\_queue, (void \*)&pcb[cur\_queue\_id]);

}

current\_running = &pcb[0];

}

* 1. enrty.S

.macro SAVE\_CONTEXT offset //offset为存储偏移量

// TODO save context

lw k0, current\_running

addi k0, k0, \offset //宏调用,不然会被当作标号

sw AT, OFFSET\_REG1(k0)

sw v0, OFFSET\_REG2(k0)

sw v1, OFFSET\_REG3(k0)

sw a0, OFFSET\_REG4(k0)

sw a1, OFFSET\_REG5(k0)

sw a2, OFFSET\_REG6(k0)

sw a3, OFFSET\_REG7(k0)

sw t0, OFFSET\_REG8(k0)

sw t1, OFFSET\_REG9(k0)

sw t2, OFFSET\_REG10(k0)

sw t3, OFFSET\_REG11(k0)

sw t4, OFFSET\_REG12(k0)

sw t5, OFFSET\_REG13(k0)

sw t6, OFFSET\_REG14(k0)

sw t7, OFFSET\_REG15(k0)

sw s0, OFFSET\_REG16(k0)

sw s1, OFFSET\_REG17(k0)

sw s2, OFFSET\_REG18(k0)

sw s3, OFFSET\_REG19(k0)

sw s4, OFFSET\_REG20(k0)

sw s5, OFFSET\_REG21(k0)

sw s6, OFFSET\_REG22(k0)

sw s7, OFFSET\_REG23(k0)

sw t8, OFFSET\_REG24(k0)

sw t9, OFFSET\_REG25(k0)

/\* $26(k0) $27(k1) will not be saved \*/

sw gp, OFFSET\_REG28(k0)

sw sp, OFFSET\_REG29(k0)

sw fp, OFFSET\_REG30(k0)

sw ra, OFFSET\_REG31(k0)

mfc0 k1, CP0\_STATUS

nop /\* mf act slow \*/ //因为mf指令执行的都很慢，为了不拖累

//流水,加入nop指令

sw k1,OFFSET\_STATUS(k0)

mfhi k1

nop

sw k1, OFFSET\_HI(k0)

mflo k1

nop

sw k1, OFFSET\_LO(k0)

mfc0 k1, CP0\_BADVADDR

nop

sw k1, OFFSET\_BADVADDR(k0)

mfc0 k1, CP0\_CAUSE

nop

sw k1, OFFSET\_CAUSE(k0)

mfc0 k1, CP0\_EPC

nop

sw k1, OFFSET\_EPC(k0)

.endm

.macro RESTORE\_CONTEXT offset

// TODO restore context

lw k0, current\_running

addi k0, k0, \offset

lw AT, OFFSET\_REG1(k0)

lw v0, OFFSET\_REG2(k0)

lw v1, OFFSET\_REG3(k0)

lw a0, OFFSET\_REG4(k0)

lw a1, OFFSET\_REG5(k0)

lw a2, OFFSET\_REG6(k0)

lw a3, OFFSET\_REG7(k0)

lw t0, OFFSET\_REG8(k0)

lw t1, OFFSET\_REG9(k0)

lw t2, OFFSET\_REG10(k0)

lw t3, OFFSET\_REG11(k0)

lw t4, OFFSET\_REG12(k0)

lw t5, OFFSET\_REG13(k0)

lw t6, OFFSET\_REG14(k0)

lw t7, OFFSET\_REG15(k0)

lw s0, OFFSET\_REG16(k0)

lw s1, OFFSET\_REG17(k0)

lw s2, OFFSET\_REG18(k0)

lw s3, OFFSET\_REG19(k0)

lw s4, OFFSET\_REG20(k0)

lw s5, OFFSET\_REG21(k0)

lw s6, OFFSET\_REG22(k0)

lw s7, OFFSET\_REG23(k0)

lw t8, OFFSET\_REG24(k0)

lw t9, OFFSET\_REG25(k0)

lw gp, OFFSET\_REG28(k0)

lw sp, OFFSET\_REG29(k0)

lw fp, OFFSET\_REG30(k0)

lw ra, OFFSET\_REG31(k0)

lw k1, OFFSET\_STATUS(k0)

mtc0 k1, CP0\_STATUS

nop

lw k1, OFFSET\_HI(k0)

mthi k1

nop

lw k1, OFFSET\_LO(k0)

mtlo k1

nop

lw k1, OFFSET\_BADVADDR(k0)

mtc0 k1, CP0\_BADVADDR

nop

lw k1, OFFSET\_CAUSE(k0)

mtc0 k1, CP0\_CAUSE

nop

lw k1, OFFSET\_EPC(k0)

mtc0 k1, CP0\_EPC

nop

.endm

* 1. scheduler方法

void scheduler(void)

{

// TODO schedule

// Modify the current\_running pointer.

pcb\_t \*next\_pcb;

next\_pcb = (pcb\_t \*)queue\_dequeue(&ready\_queue); //出栈

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

{

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

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

}

current\_running = next\_pcb;

current\_running->status = TASK\_RUNNING;

return ;

}

***Mutex lock 设计流程***

1. Pspin-lock 和 mutual lock 的区别，能获取到锁和获取不到锁时各自的处理流程

自旋锁的实现是设置一个变量，当一个线程要进入临界区的时候，首先检查这个变量，如果这个变量状态为临界区，并且此时没有其他冲突线程在访问这个变量，那么该线程就进入临界区，并且将这个变量置为有线程访问状态。如果这个变量状态为有其他冲突线程在访问，那么就不断的使用while 循环重试，直到可以进入临界区。自旋锁每次没获得锁，会进入忙等待，占用CPU时间，比较浪费资源，单线程处理器上如果不直接打断，就会一直等待下去。互斥锁的实现方法为一旦线程请求锁失败，那么该线程会自动被挂起到该锁的阻塞队列中，不会被调度器进行调度。直到占用该锁的线程释放锁之后，被阻塞的线程会被占用锁的线程主动的从阻塞队列中重新放到就绪队列，并获得锁。互斥锁可以节约CPU资源。

1. 被阻塞的 task 何时再次执行

互斥锁释放之后，会查询相应的阻塞队列，若不为空，则将第一个任务出队，修改任务为就绪状态，然后进入就绪队列，并把锁设为UNLOCKED状态。等待下次调度到这个任务时，该任务就可以顺利的获得锁并执行。

1. 设计、实现或调试过程中遇到的问题和得到的经验

在写完这部分的逻辑后，无论怎么上板卡发现问题都在执行到进入test\_lock的第一个任务时在开头停住。最后在助教的帮助下，发现Makefile中连接了错误的任务（test\_lock2），该任务在开头就出现中断，由于现在的任务还未进行到中断部分，程序便在此处停止。修改未正确的test\_lock1后，并接着修改打印光标的位置（从第1、2行打印修改到第3、4行打印），程序得以成功运行。

1. 部分代码
   1. block方法

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

{

// block the current\_running task into the queue

current\_running->status = TASK\_BLOCKED;

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

do\_scheduler();

}

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

{

// unblock the head task from the queue

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

block\_head->status = TASK\_READY;

queue\_push(&ready\_queue, block\_head); //这里不需要do\_scheduler

}

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;

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

}

}

* 1. mutex\_lock

int lock\_queue\_id = 0;

typedef struct mutex\_lock

{

lock\_status\_t status;

int lock\_id;

} mutex\_lock\_t;

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;

}