实验四 内核线程管理实验



struct proc\_struct {

enum proc\_state state; // Process state

int pid; // Process ID

int runs; // the running times of Proces

uintptr\_t kstack; // Process kernel stack

volatile bool need\_resched; // need to be rescheduled to release CPU?

struct proc\_struct \*parent; // the parent process

struct mm\_struct \*mm; // Process's memory management field

struct context context; // Switch here to run process

struct trapframe \*tf; // Trap frame for current interrupt

uintptr\_t cr3; // the base addr of Page Directroy Table(PDT)

uint32\_t flags; // Process flag

char name[PROC\_NAME\_LEN + 1]; // Process name

list\_entry\_t list\_link; // Process link list

list\_entry\_t hash\_link; // Process hash list

};

static struct proc\_struct \*

alloc\_proc(void) {

struct proc\_struct \*proc = kmalloc(sizeof(struct proc\_struct));

if (proc != NULL) {

//LAB4:EXERCISE1 YOUR CODE

/\*

\* below fields in proc\_struct need to be initialized

\* enum proc\_state state; // Process state

\* int pid; // Process ID

\* int runs; // the running times of Proces

\* uintptr\_t kstack; // Process kernel stack

\* volatile bool need\_resched; // bool value: need to be rescheduled to release CPU?

\* struct proc\_struct \*parent;

\* struct mm\_struct \*mm;

\* struct context context;

\* struct trapframe \*tf;

\* uintptr\_t cr3;

\* uint32\_t flags;

\* char name[PROC\_NAME\_LEN + 1];

proc->state = PROC\_UNINIT;

proc->pid = -1;

proc->runs = 0;

proc->kstack = 0;

proc->need\_resched = 0;

proc->parent = NULL;

proc->mm = NULL;

memset(&(proc->context), 0, sizeof(struct context));//进程上下文初始化

proc->tf = NULL;

proc->cr3 = boot\_cr3;//设置内核页目录表的基址

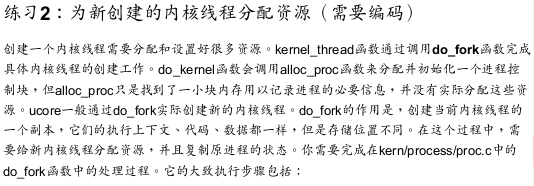
proc->flags = 0; //标志位初始化

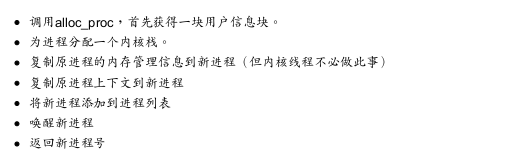
memset(proc->name, 0, PROC\_NAME\_LEN); //进程名初始化

}

return proc;

}







int

do\_fork(uint32\_t clone\_flags, uintptr\_t stack, struct trapframe \*tf) {

int ret = -E\_NO\_FREE\_PROC;

struct proc\_struct \*proc;

if (nr\_process >= MAX\_PROCESS) {

goto fork\_out;

}

ret = -E\_NO\_MEM;

/\*

\* Some Useful MACROs, Functions and DEFINEs, you can use them in below implementation.

\* MACROs or Functions:

\* alloc\_proc: create a proc struct and init fields (lab4:exercise1)

\* setup\_kstack: alloc pages with size KSTACKPAGE as process kernel stack

\* copy\_mm: process "proc" duplicate OR share process "current"'s mm according clone\_flags

\* if clone\_flags & CLONE\_VM, then "share" ; else "duplicate"

\* copy\_thread: setup the trapframe on the process's kernel stack top and

\* setup the kernel entry point and stack of process

\* hash\_proc: add proc into proc hash\_list

\* get\_pid: alloc a unique pid for process

\* wakup\_proc: set proc->state = PROC\_RUNNABLE

\* VARIABLES:

\* proc\_list: the process set's list

\* nr\_process: the number of process set

\*/

bool intr\_flag;

local\_intr\_save(intr\_flag);

{

proc->pid = get\_pid();

hash\_proc(proc);

list\_add(&proc\_list, &(proc->list\_link));

nr\_process ++;

}

local\_intr\_restore(intr\_flag);

wakeup\_proc(proc);

ret = proc->pid;

fork\_out:

return ret;

bad\_fork\_cleanup\_kstack:

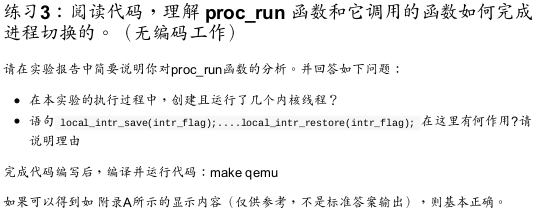
put\_kstack(proc);

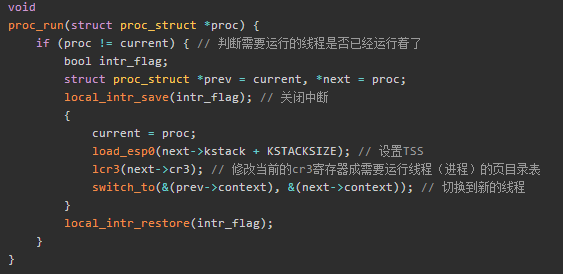
bad\_fork\_cleanup\_proc:

kfree(proc);

goto fork\_out;

}





问题回答：

1. 总共创建了两个内核线程，分别为：

* idleproc: 最初的内核线程，在完成新的内核线程的创建以及各种初始化工作之后，进入死循环，用于调度其他线程；
* initproc: 被创建用于打印"Hello World"的线程；

1. 语句 local\_intr\_save(intr\_flag);....local\_intr\_restore(intr\_flag);

的作用

* 这个语句的左右是关闭中断，使这个语句块内的内容不会被中断打断，是一个原子操作；
* 这使得一些关键的代码不会被打断，从而引起不必要的错误；
* 如在proc\_run函数中，将current指向了要切换到的线程，但是此时还没有真正将控制权转移过去，如果在这个时候出现中断打断这些操作，就会出现current中保存的并不是正在运行的线程的中断控制块，从而出现错误；

编译make qemu

