Reference

一个操作系统的实现

minix

需求实现

- 实现系统调用`fork`，可在本进程基础上从当前上下文创建另一个线程
- 实现系统调用`get_pid`，用来观察不同的线程

技术难点

上下文的复制，线程组的标定.

fork

观察进程控制块结构

```
    struct Process {
        StackFrame regs;          /* process registers saved in stack frame */

        u16 ldt_sel;               /* gdt selector giving ldt base and limit */
        Descriptor ldts[LDT_SIZE]; /* local descriptors for code and data */

            int ticks;                 /* remained ticks */
            int priority;

        u32 pid;                   /* process id passed in from MM */
        char name[16];            /* name of the process */

        u32 state;

        ipc::Message* msg;
        u32 recvfrom;
        u32 sendto;

        int has_int_msg;           /**
                        * nonzero if an INTERRUPT occurred when
                        * the task is not ready to deal with it.
                        */

        Process* q_sending;   /**
                        * queue of procs sending messages to
                        * this proc
                        */
        Process* next_sending;/**
                        * next proc in the sending
                        * queue (q_sending)
```

```
                        */

        int nr_tty;

        u32 stack_size;

        char* stack;
    };
```

拷贝之后只需要改变

1. pid
2. stack
3. regs.esp

其他都不需要改变

创建系统调用`fork`

```
#pragma once

#include "proc.h"
#include "type.h"

constexpr u32 NR_SYS_CALL = 5;

extern system_call  sys_call_table[NR_SYS_CALL];

int     sys_write(u32 u1, u32 u2, char* buf, proc::Process* p_proc);
u8      sys_getchar(u32 u1, u32 u2, u32 u3, proc::Process* p_proc);
void    sys_fork(u32 u1, u32 u2, u32 u3, proc::Process* p_proc);
u32     sys_get_pid(u32 u1, u32 u2, u32 u3, proc::Process* p_proc);


extern "C" {
    void    write(const char* buf);
    u8      getch();
    u32     sendrec(int function, int src_dest, proc::ipc::Message* msg);
    void    fork();
    u32     get_pid();
}
```

所以创建`thread.h`，`thread.cpp`文件，创建`sys_clone`

```
void    sys_fork(u32 u1, u32 u2, u32 u3, proc::Process* p_proc)
{
    proc::clone(p_proc);
}
```

创建函数`clone`

```
    void clone(Process* p)
```

```
    {
        Process*    p_proc      = proc_table + proc_num;
        char*       p_task_stack    = proc_stack + kProcStackSize - used_proc_stack;

        memcpy(p_proc, p, sizeof(Process));

        p_proc->pid = proc_num;
        p_proc->stack = p_task_stack - p->stack_size;
        p_proc->regs.esp += (u32)(p_proc->stack - p->stack);

        memcpy(p_proc->stack, p->stack, p->stack_size);

        proc_num++;
    }
```

链表实现线程组，为信号量做准备

进程控制块有一下两个成员变量

```
        bool is_father; // 是否为父进程
        u32 thread_list; // 指向下一个本组链表
```

add_proc

```
        p_proc->is_father = true;
        p_proc->thread_list = proc_num;
```

clone

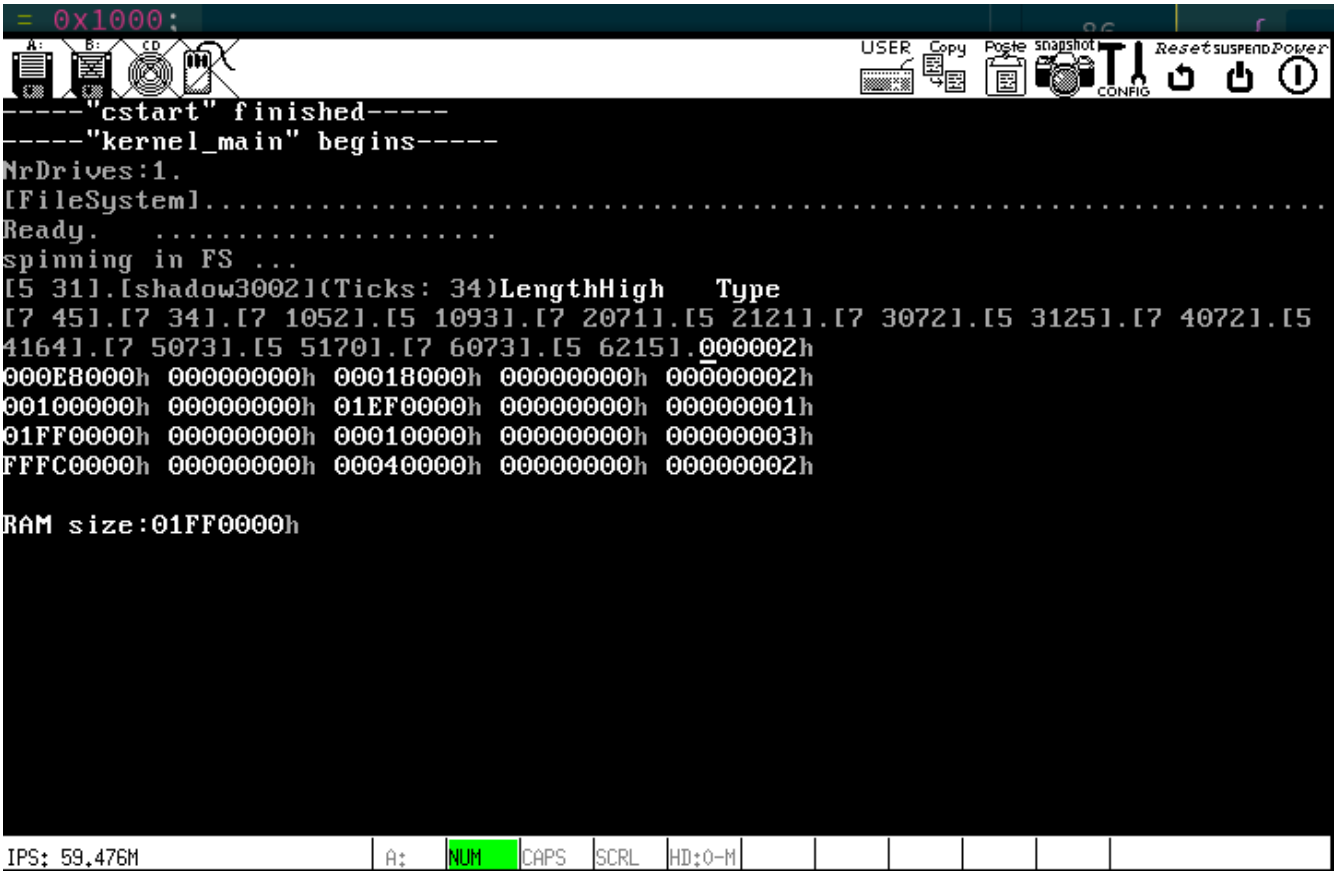
```
        p->is_father = true;
        p_proc->is_father = false;
        p_proc->thread_list = p->thread_list;
        p->thread_list = p_proc->pid;
```

实验展示

通过一个简单函数的`fork`，来观察

```
void TestB()
{
    int i = 0x1000;
    printf("[%d %d].", get_pid(), clock::get_ticks());
    fork();
    while(1){
        printf("[%d %d].", get_pid(), clock::get_ticks());
        clock::milli_delay(10000);
    }
}
```

实验结果



观察到确实创建了两个进程并发执行，而且每个进程还能调用系统调用.

实验心得

有个地方很容易忽略，就是`esp`的偏移更新，以及新栈的建立，以及栈的拷贝．总的来说本次实验比较简单．32位略微有些`dirty`，在考虑用`rust`重写64位系统.