第一次实例分析第一部分

本部分涉及到的部分数据结构

struct cpu/proc.h

```
struct cpu {
uchar apicid;                    // Local APIC ID
struct context *scheduler;  // swtch() here to enter scheduler
struct taskstate ts;             // Used by x86 to find stack for interrupt
struct segdesc gdt[NSEGS];  // x86 global descriptor table
volatile uint started;           // Has the CPU started?
int ncli;                   // Depth of pushcli nesting.
int intena;                 // Were interrupts enabled before pushcli?
struct proc *proc;          // The process running on this cpu or null
};
```

本部分涉及到的部分数据结构

```
struct cpu {
uchar apicid;                    // Local APIC ID
struct context *scheduler;   // swtch() here to enter scheduler
struct taskstate ts;             // Used by x86 to find stack for interrupt
struct segdesc gdt[NSEGS];   // x86 global descriptor table
volatile uint started;           // Has the CPU started?
int ncli;                        // Depth of pushcli nesting.
int intena;                      // Were interrupts enabled before pushcli?
struct proc *proc;               // The process running on this cpu or null
};
```

scheduler:指向一个将要切换的进程的上下文信息

本部分涉及到的部分数据结构

```
struct cpu {
uchar apicid;                    // Local APIC ID
struct context *scheduler;   // swtch() here to enter scheduler
struct taskstate ts;             // Used by x86 to find stack for interrupt
struct segdesc gdt[NSEGS];   // x86 global descriptor table
volatile uint started;           // Has the CPU started?
int ncli;                        // Depth of pushcli nesting.
int intena;                      // Were interrupts enabled before pushcli?
struct proc *proc;           // The process running on this cpu or null
};
```

ts:它是一个struct taskstate的成员属性。表示当前CPU正在运行的任务所处的状态。taskstate中包含了大量寄存器的信息。当中断或者异常发生时，CPU能够通过ts找到当前任务内核栈的位置

本部分涉及到的部分数据结构

```
struct cpu {
uchar apicid;                    // Local APIC ID
struct context *scheduler;  // swtch() here to enter scheduler
struct taskstate ts;             // Used by x86 to find stack for interrupt
struct segdesc gdt[NSEGS];  // x86 global descriptor table
volatile uint started;           // Has the CPU started?
int ncli;                        // Depth of pushcli nesting.
int intena;                      // Were interrupts enabled before pushcli?
struct proc *proc;               // The process running on this cpu or null
};
```

gdt：gdt是一个struct segdesc类型的成员属性，即segment descriptor, 该数据结构用于界定不同内存区域的特征。。

本部分涉及到的部分数据结构

struct context/proc.h

```
struct context {
    uint edi;   //目的变址寄存器，用于存放目的数据单元在段中的偏移
    uint esi;   //源变址寄存器，用于存放源操作数的数据单元在段内的偏移量
    uint ebx;   //基地址寄存器
    uint ebp;   //栈基址寄存器
    uint eip;   //程序计数器，用于保存指令的地址
};
```

context表示了当前的CPU重要寄存器的状态

本部分涉及到的部分数据结构

struct proc/proc.h

```
struct proc {
    uint sz;                     // 该进程加载到内存中所需要的大小
    pde_t* pgdir;                // 内存页表
    char *kstack;                // 内核栈的栈底地址
    enum procstate state;        // 该进程的当前状态
    int pid;                     // 进程号
    struct proc *parent;         // 该指针指向该进程的父进程
    struct trapframe *tf;        // 指向一个陷阱帧的指针
    struct context *context;     // 指向一个context的的指针
    void *chan;                  // chan是一个与进程sleep相关的void指针
    int killed;                  // killed表示此进程是否被杀死，为1表示已经被杀了
    struct file *ofile[NOFILE];  // 该进程打开文件的列表
    struct inode *cwd;           // 该进程当前所处的文件夹目录
    char name[16];               // 该进程的名字
};
```

用于表示一个进程的相关状态和数据信息

本部分涉及到的部分数据结构

struct proc/proc.h

```
struct proc {
    uint sz;                     // 该进程加载到内存中所需要的大小
    pde_t* pgdir;                // 内存页表
    char *kstack;                // 内核栈的栈底地址
    enum procstate state;        // 该进程的当前状态
    int pid;                     // 进程号
    struct proc *parent;         // 该指针指向该进程的父进程
    struct trapframe *tf;        // 指向一个陷阱帧的指针
    struct context *context;     // 指向一个context的的指针
    void *chan;                  // chan是一个与进程sleep相关的void指针
    int killed;                  // killed表示此进程是否被杀死，为1表示已经被杀了
    struct file *ofile[NOFILE];  // 该进程打开文件的列表
    struct inode *cwd;           // 该进程当前所处的文件夹目录
    char name[16];               // 该进程的名字
};
```

pid：表示该进程的进程号，pid在proc.c中的allocproc函数中被全局变量nextpid赋值,nextpid仅在proc.c中使用，表示当前已经使用了的进程数目，初始值为1

本部分涉及到的部分数据结构

struct proc/proc.h

```
struct proc {
    uint sz;                     // 该进程加载到内存中所需要的大小
    pde_t* pgdir;                // 内存页表
    char *kstack;                // 内核栈的栈底地址
    enum procstate state;        // 该进程的当前状态
    int pid;                     // 进程号
    struct proc *parent;         // 该指针指向该进程的父进程
    struct trapframe *tf;        // 指向一个陷阱帧的指针
    struct context *context;     // 指向一个context的的指针
    void *chan;                  // chan是一个与进程sleep相关的void指针
    int killed;                  // killed表示此进程是否被杀死，为1表示已经被杀了
    struct file *ofile[NOFILE];  // 该进程打开文件的列表
    struct inode *cwd;           // 该进程当前所处的文件夹目录
    char name[16];               // 该进程的名字
};
```

tf：struct trapframe是一种因为中断，自陷，异常等情况进入内核后，在堆栈上形成的一种数据结构。关于struct trapframe的定义在x86.h中

本部分涉及到的部分数据结构

struct trapframe/x86.h

```
struct trapframe {
    // registers as pushed by pusha
    uint edi;
    uint esi;
    uint ebp;
    uint oesp;      // useless & ignored
    uint ebx;
    uint edx;
    uint ecx;
    uint eax;

    // rest of trap frame
    ushort gs;
    ushort padding1;
    ushort fs;
    ushort padding2;
    ushort es;
    ushort padding3;
    ushort ds;
    ushort padding4;
    uint trapno;

    // below here on from user to kernel
    uint esp;
    ushort ss;
    ushort padding6;

    // below here de
    uint err;
    uint eip;
    ushort cs;
    ushort padding5;
    uint eflags;
};
```

在中断，异常和系统调用时，需要陷入内核态，在陷入内核态之前需要保存现场，在从内核态恢复到用户态的时候需要恢复现场，trapframe结构体就是用于保存现场的数据结构

本部分涉及到的部分数据结构

struct proc/proc.h

```
struct proc {
    uint sz;                     // 该进程加载到内存中所需要的大小
    pde_t* pgdir;                // 内存页表
    char *kstack;                // 内核栈的栈底地址
    enum procstate state;        // 该进程的当前状态
    int pid;                     // 进程号
    struct proc *parent;         // 该指针指向该进程的父进程
    struct trapframe *tf;        // 指向一个陷阱帧的指针
    struct context *context;     // 指向一个context的的指针
    void *chan;                  // chan是一个与进程sleep相关的void指针
    int killed;                  // killed表示此进程是否被杀死，为1表示已经被杀了
    struct file *ofile[NOFILE];  // 该进程打开文件的列表
    struct inode *cwd;           // 该进程当前所处的文件夹目录
    char name[16];               // 该进程的名字
};
```

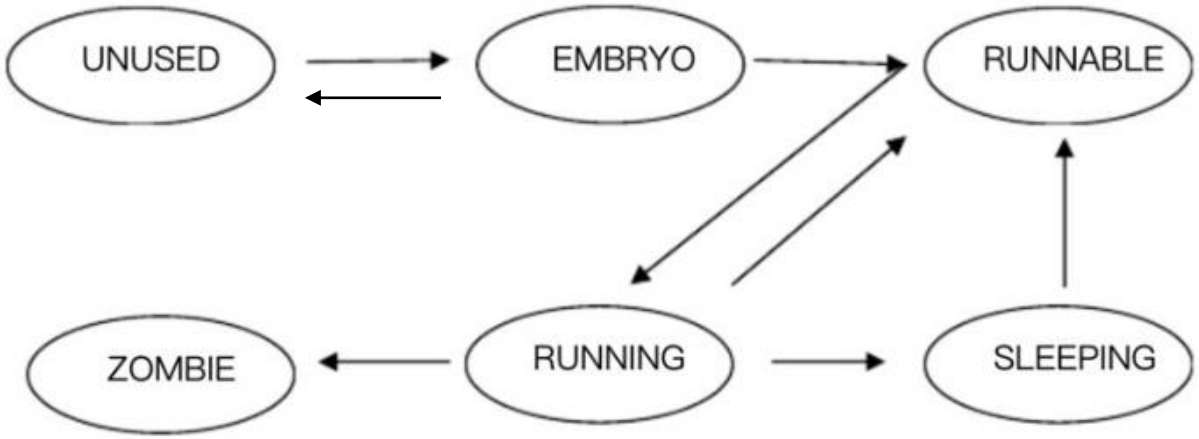
state：state成员用于表示该进程的当前状态，进程的状态由枚举类型procstate给出，它定义在proc.h中

本部分涉及到的部分数据结构

enum procstate/proc.h

```
enum procstate { UNUSED, EMBRYO, SLEEPING, RUNNABLE,
RUNNING, ZOMBIE };
```

在allocproc中创建进程时，state先被初始化为EMBRYO，当该进程申请内核栈失败时，会将该进程的状态设置为UNUSED，表明这次进程创建失败
此外，在fork函数中，若复制创建的子进程在拷贝父进程的页表时失败，则会将子进程的状态设置为UNUSED，否则会在一系列必要的初始化工作完成之后将状态设置为RUNNABLE:
进程状态的切换由于涉及到很多其他的模块内容，这里不再细说。下图大致表示了xv6操作系统中进程状态的切换情况:



第一个进程的启动

大致流程：

- main/main.c
- userinit/proc.c
- allocproc/proc.c
- kalloc/proc.c
- mpmain/main.c
- scheduler/proc.c
- swtch/swtch.S
- forkret/proc.c
- trapret/trapasm.S

main/main.c

kernel的主函数main在运行阶段做的事情如下

```
int main(void)
{
    kinit1(end, P2V(4*1024*1024)); // phys page
allocator
    kvmalloc();      // kernel page table
    mpinit();        // detect other processors
    ...
    kinit2(P2V(4*1024*1024), P2V(PHYSTOP)); // must
come after startothers()
    userinit();      // first user process
    mpmain();        // finish this processor's setup
}
```

kinit1与kinit2是对分配器进行初始化；
kvmalloc用于创建并切换到一个拥有内核运行所需的KERNBASE以上映射的页表；
mpinit用于检测CPU个数并将检测到的CPU存入一个全局的数组中；
userinit用于创建第一个用户进程；
最后进入mpmain函数

userinit/proc.c

- userinit函数用于创建第一个用户进程。它仅仅在main.c中被调用一次，用于创建第一个用户进程。后面的进程创建由allocproc来完成。
- 在userinit中，首先调用allocproc创建一个进程控制块，然后对页表，中断帧中的一些寄存器进行初始化。最后将该进程设置为RUNNABLE
- 其中要留意的部分是这几个

```
p = allocproc();  p->tf->eip = 0;
```

allocproc/proc.c

- allocproc函数用于创建一个进程并返回指向该进程PCB的一个指针。
- 开始时，在pcb 数组ptable.proc中找到一个处于未使用状态的进程控制块然后为它分配一个进程id

```
    for(p = ptable.proc; p < &ptable.proc[NPROC]; p++)
        if(p->state == UNUSED)
            goto found; //如果找到一个没用过的proc，就跳转到found标签，否则退出
...
found:
    p->state = EMBRYO; //标记为EMBRYO
    p->pid = nextpid++;
```

allocproc/proc.c

- allocproc调用kalloc为该进程分配栈空间。
- 如果分配失败，该进程的状态将被重新设置为UNUSED，表示进程创建失败;
- 如果内核栈分配成功，allocproc函数会将内核栈的基址增加KSTACKSIZE大小，并通过将sp指针上移struct trapframe大小的方法为中断帧预留出空间

```
sp = p->kstack + KSTACKSIZE;
// Leave room for trap frame.
sp -= sizeof *p->tf;
p->tf = (struct trapframe*)sp;
```

allocproc/proc.c

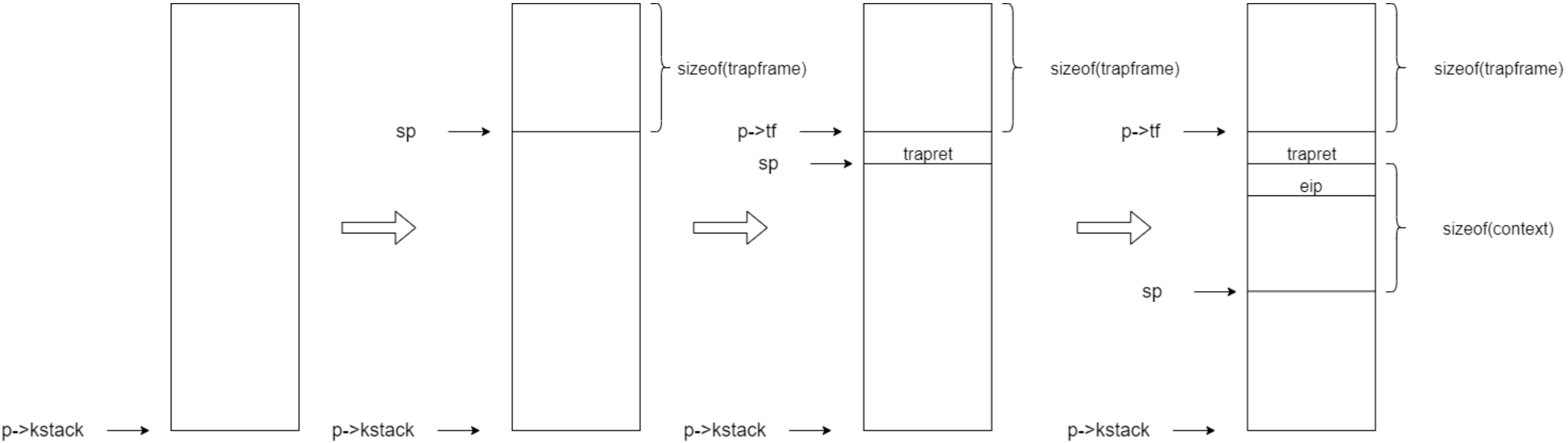
- 同时，sp指针移动4byte，预留出trapret的地址，该地址将用于作为forkret的返回地址；
- sp再上移context的大小字节数目，留出用于保存context的相应空间

```
sp -= 4;
*(uint*)sp = (uint)trapret;
sp -= sizeof *p->context;
p->context = (struct context*)sp;
memset(p->context, 0, sizeof *p->context);
p->context->eip = (uint)forkret;
```

```
struct context {
    uint edi;
    uint esi;
    uint ebx;
    uint ebp;
    uint eip;
};
```

allocproc/proc.c

上述指令的动态过程如下所示

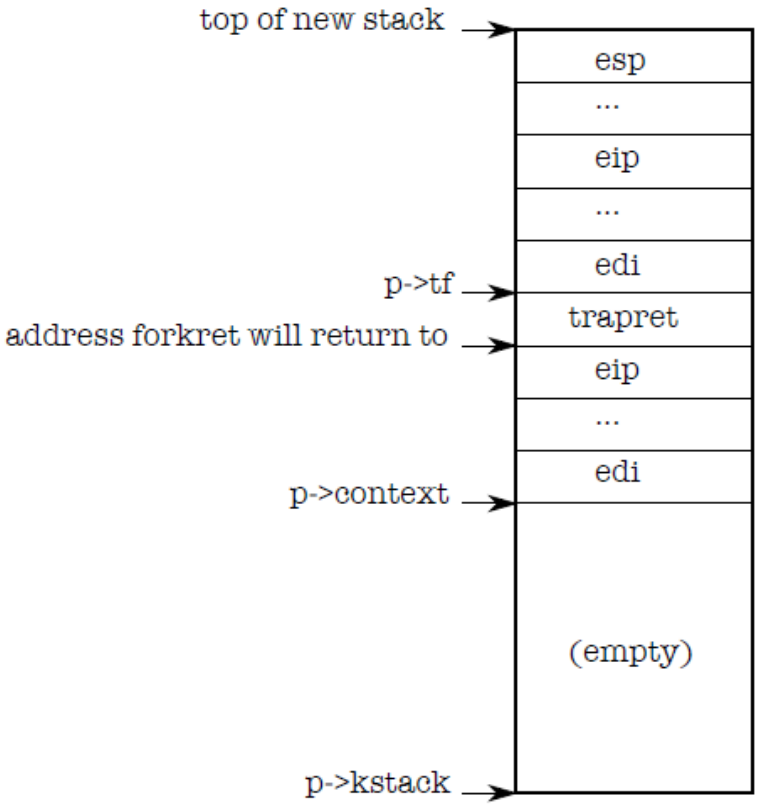


kalloc/kalloc.c

- allocproc使用到了此函数
- kalloc函数的作用是为进程创建时分配栈空间，因此需要谈到xv6的内存分配:
- xv6中用于内存分配的数据结构是free list，组成该free list的元素是一个被称为struct run的数据结构
- 在形式上，该数据结构类似于一个链表，它记录了当前的物理内存地址是否被占用，如果没有被占用，它会将free list队首的一个run结构体从链表中移除出来，表示该物理内存被占用，而将该被移除出来的run结构体的地址设置为返回值，表示为栈分配的内存地址

allocproc/proc.c

在完成了这些指令之后，该proc对应的进程控制块的内存布局如下所示



mpmain/main.c

```
cprintf("cpu%d: starting %d\n", cpuid(), cpuid());
idtinit();
xchg(&(mycpu()->started), 1);
scheduler();
```

- 使用idtinit进行中断机制中IDT表的初始化
- 使用xchg将CPU的已启动标志置1
- 调用scheduler开始进行进程调度

scheduler/proc.c

- 寻找一个状态为RUNNABLE的进程

```
// Loop over process table looking for process to run.
acquire(&ptable.lock);  //获得锁
for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
    if(p->state != RUNNABLE)
        continue;
```

scheduler/proc.c

- 调用switchuvm通知硬件开始使用该进程的页表
- 将找到的进程的状态改为RUNNING，调用swtch函数
- 最后再调用switchkvm函数设置cr3寄存器的值为kpgdir首地址

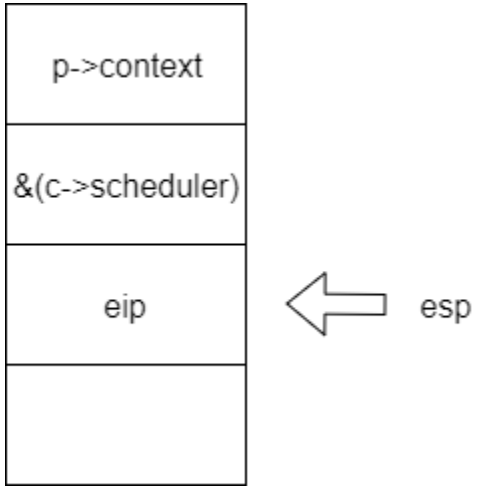
```
c->proc = p;
switchuvm(p);          //切换到该进程的页表
p->state = RUNNING; //切换为running状态

swtch(&(c->scheduler), p->context);//切换到该进程中运行
switchkvm();//设置cr3寄存器的值为kpgdir首地址

c->proc = 0;
```

swtch/swtch.S

- 根据scheduler调用swtch的语句 `swtch(&(c->scheduler), p->context);`
- c->scheduler的地址和p->context作为swtch的参数。前者最终指向正在执行的 scheduler函数，后者则是目标函数，
- 根据C的默认参数调用约定__cdecl，函数实参在线程栈上按照从右至左的顺序依次压栈。



而swtch函数所起到的作用就是将控制权由正在执行的函数转移到目标函数。由于当前的上下文并非是进程的，而是一个特殊的调度器Per-CPU的上下文。因此当前上下文会被保存在Per-CPU的cpu->scheduler这个域里面

swtch/swtch.S

- 将**old的地址写入eax;
- 将目标函数的地址赋给edx
- 当前上下文入栈保存
- 将栈指针的位置赋给*old
- 将目标函数的地址赋给栈指针
- 返回

```
movl 4(%esp), %eax
movl 8(%esp), %edx
```

```
pushl %ebp
pushl %ebx
pushl %esi
pushl %edi
```

```
movl %esp, (%eax)
movl %edx, %esp
```

```
popl %edi
popl %esi
popl %ebx
popl %ebp
```

```
ret
```

swtch/swtch.S

- 在此之前allocproc函数将p->context->eip设置为了forkret的入口地址
- 根据ret指令的定义，它会返回eip所保存的位置，也就是进入forkret函数

```
p->context->eip = (uint)forkret;
```

forkret/proc.c

- 此时为forkret第一次执行，first= 1
- forkret在第一次执行的时候，会调用一些初始化函数。iinit函数用于读取Superblock，初始化inode相关的锁；initlog函数是执行与日志相关的初始化。

```
first = 0;
iinit(ROOTDEV);
initlog(ROOTDEV);
```

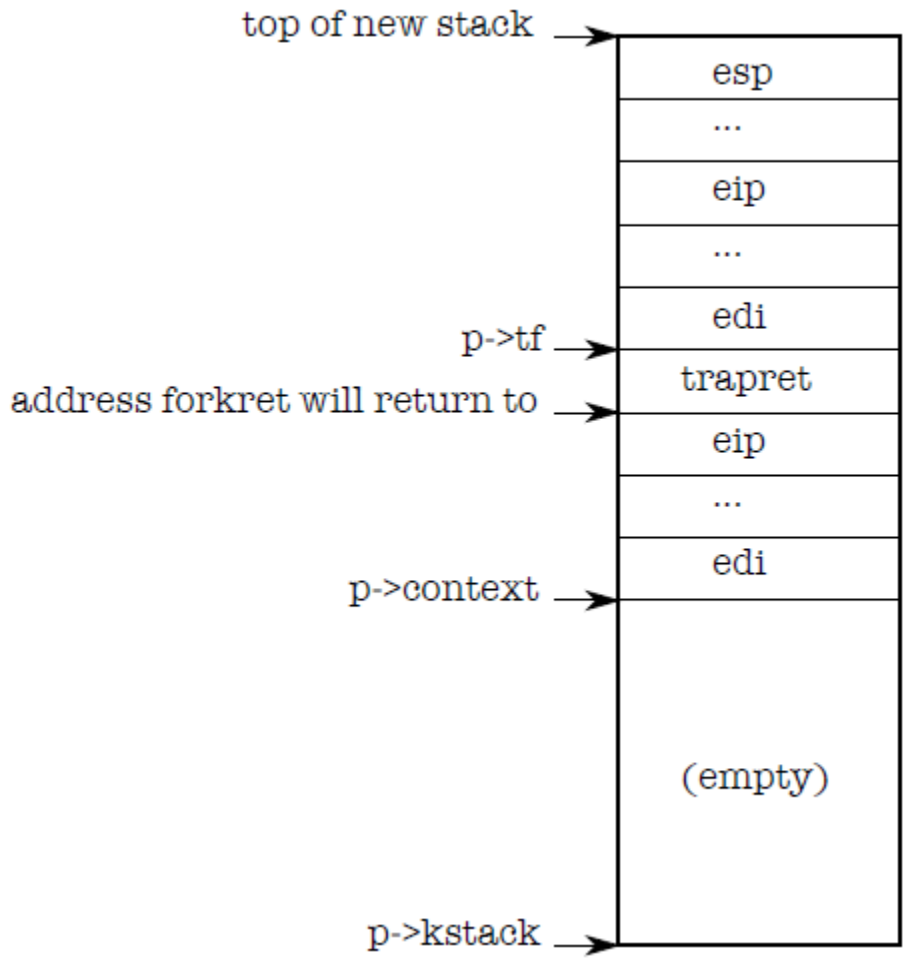
forkret/proc.c

- forkret执行完毕之后，返回当初读取eip的位置。而在之前allocproc函数在某个时刻将sp所处的位置写入了trapret的地址

```
*(uint*)sp = (uint)trapret;
```

forkret/proc.c

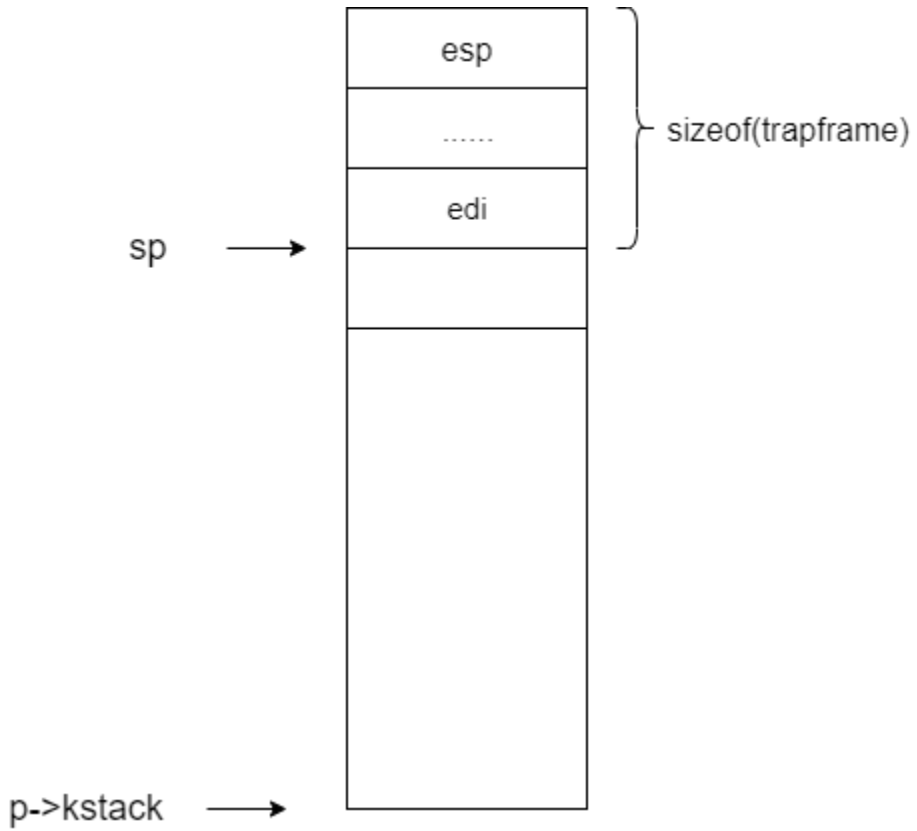
- 这使得trapret的入口地址正好在eip上面。
- 由于调用forkret函数的时候不是用正常的call方法，因此它的地址不会被压栈，
- 但是forkret的返回行为是正常的函数返回行为。
- 等到forkret执行完毕之后，栈指针指向了eip之后的内存地址，系统以为这个位置存储的是forkret在调用时候的eip值，于是会将这个位置的内容从栈中弹出，赋给eip，因此从forkret返回之后就会进入trapret了



trapret/trapasm.S

- 在之前allocproc函数里面，已经在trapret上面分配出来了一个trapframe大小的内容

```
sp -= sizeof *p->tf;
```



trapret/trapasm.S

- 弹出所有通用寄存器对应的栈空间的信息
- 将四个段寄存器所对应的内存区段的内容按照trapframe定义的顺序弹出
- 使用addl指令让esp增长8字节，以跳过trapno和err两个寄存器对应的区段
- 执行iret指令

```
popal
```

```
popl %gs
popl %fs
popl %es
popl %ds
addl $0x8, %esp  # trapno and errcode
```

```
iret
```

struct trapframe/x86.h

```
struct trapframe {
    // registers as pushed by pusha
    uint edi;
    uint esi;
    uint ebp;
    uint oesp;      // useless & ignored
    uint ebx;
    uint edx;
    uint ecx;
    uint eax;

    // rest of trap frame
    ushort gs;
    ushort padding1;
    ushort fs;
    ushort padding2;
    ushort es;
    ushort padding3;
    ushort ds;
    ushort padding4;
    uint trapno;

    // below here de
    uint err;
    uint eip;
    ushort cs;
    ushort padding5;
    uint eflags;

    // below here on
from user to kernel
    uint esp;
    ushort ss;
    ushort padding6;
};
```

trapret/trapasm.S

- iret：interrupt return
- 根据Intel提供的x86指令手册
- 如果iret跳转到了另一个特权级，那么也会将esp和ss从栈里面弹出来。
- 由于xv6将控制权在用户和内核之间切换是通过中断机制实现的，因此需要在进程的内核栈上面保存用户寄存器
- 而ret不会弹出这些内容

As with a real-address mode interrupt return, the IRET instruction pops the return instruction pointer, return code segment selector, and EFLAGS image from the stack to the EIP, CS, and EFLAGS registers, respectively, and then resumes execution of the interrupted program or procedure. If the return is to another privilege level, the IRET instruction also pops the stack pointer and SS from the stack, before resuming program execution. If the return is to virtual-8086 mode, the processor also pops the data segment registers from the stack.

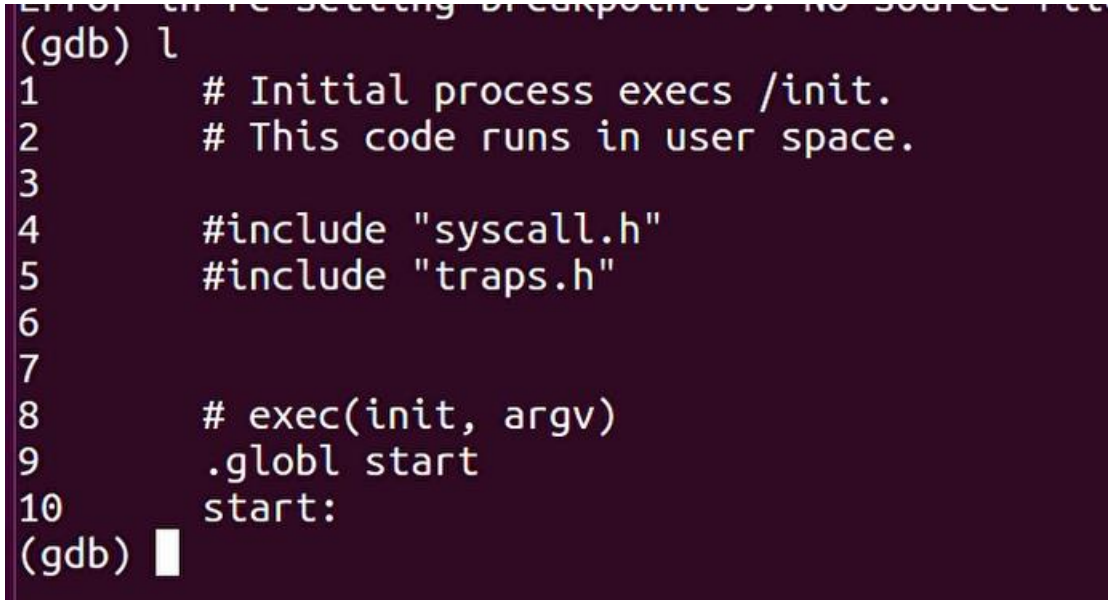
trapret/trapasm.S

- Userinit已经将trapframe的内容初始化过了，eip也不例外
- 执行iret之后，这里的0就会被赋给eip，使得系统退出trapret之后进入了虚拟地址为0x0的地方

```
p->tf->eip = 0;  // beginning of initcode.S
```

trapret/trapasm.S

- 经实际验证，此处为initcode.S的第一条指令



Linux进程相关的数据结构

- linux中与进程相关的数据结构主要位于kernel\sched\sched.h
- 与xv6相比，linux的PCB中加入了以下结构

Linux进程相关的数据结构

涉及进程调度的：

```
int prio, static_prio, normal_prio;  /* 表示进程的运行优先级 */
/*
    prio: 用于保存动态优先级
    static_prio: 用于保存静态优先级，可以通过nice系统调用来修改
    normal_prio: 它的值取决于静态优先级和调度策略
    priort_priority: 用于保存实时优先级
*/
const struct sched_class *sched_class  /* 调度器的指针 */
struct sched_entity se; /* 调度器 实例化的对象 */
struct sched_rt_entity rt; /* 实时调度器的一个对象 */

unsigned int policy;  /* 进程调度策略 */
cpumask_t     cpus_allowed;  /* 表示某个cpu可用的标志 */
```

Linux进程相关的数据结构

涉及进程之间关系的:

```
struct task_struct *real_parent;
struct task_struct *parent; /*   指向父进程的指针   */

struct list_head children;  /* 孩子进程形成的链表   */
struct list_head sibling;   /* 兄弟进程形成的链表   */

struct task_struct *group_leader  /* 指向该进程所在进程组的领头进程   */
```

Linux进程相关的数据结构

- 以thread_info为例

```
struct thread_info {
    struct pcb_struct   pcb;        /* palcode state */

    struct task_struct  *task;      /* main task structure */
    unsigned int        flags;      /* low level flags */
    unsigned int        ieee_state; /* see fpu.h */

    mm_segment_t        addr_limit; /* thread address space */

    struct exec_domain  *exec_domain;

    unsigned        cpu;        /* current CPU */
    int         preempt_count; /* 0 => preemptable, <0 => BUG */
    unsigned int        status;     /* thread-synchronous flags */

    int bpt_nsaved;
    unsigned long bpt_addr[2];      /* breakpoint handling  */
    unsigned int bpt_insn[2];
};
```

Linux进程相关的数据结构

- 以thread_info为例
- Thread_info反映了当前正在CPU上运行的线程的情况
- 在Linux中，内核会将一个进程的内核栈与thread_info放在一起，以union的方式实现：

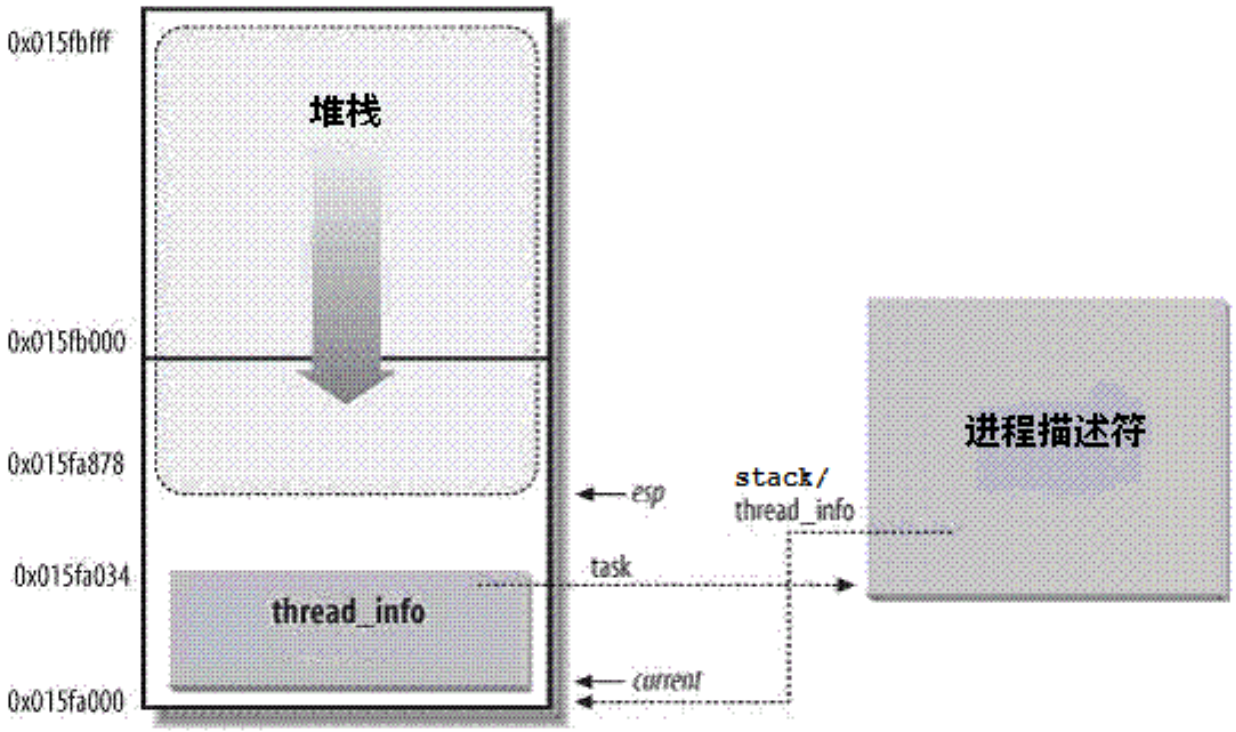
```
union thread_union
{
    struct thread_info thread_info;
    unsigned long stack[THREAD_SIZE/sizeof(long)];
};
```

Linux进程相关的数据结构

- 内核栈的空间大小被定义为THREAD_SIZE/sizeof(long)

```
#define THREAD_SIZE (2*PAGE_SIZE)
```

- THREAD_SIZE = 8192，实际上它的值根据体系结构的不同会有所差别



Linux进程相关的数据结构

- thread_info的第二个成员属性是指向task_struct 的指针
- task_struct也有一个对应的thread_info成员，以实现互相引用
- task_struct里面含有一个进程的大量资源信息，而thread_info则反映了该进程中正在运行的线程的信息

```
struct thread_info {
    struct pcb_struct   pcb;        /* palcode state */

    struct task_struct  *task;      /* main task structure */
    unsigned int        flags;      /* low level flags */
    unsigned int        ieee_state; /* see fpu.h */

    mm_segment_t        addr_limit; /* thread address space */

    struct exec_domain  *exec_domain;

    unsigned        cpu;        /* current CPU */
    int         preempt_count; /* 0 => preemptable, <0 => BUG */
    unsigned int        status;     /* thread-synchronous flags */

    int bpt_nsaved;
    unsigned long bpt_addr[2];      /* breakpoint handling  */
    unsigned int bpt_insn[2];
};
```

Linux进程相关的数据结构

- 在Linux系统中，通过current_thread_info函数得到当前运行线程的地址，如果引用thread_info中的task_struct指针就能得到进程描述符(task_struct)的相关信息。

```
static inline __attribute_const__ struct thread_info *current_thread_info(void)
{
    register unsigned long sp asm("sp");
    return (struct thread_info *)(sp & ~(THREAD_SIZE - 1));
}
```

Linux进程相关的数据结构

- exec_domain表明了当前进程的运行程序属于哪一种规范，不同体系结构的可运行程序规范差异存放在exec_domain变量中。

```
struct thread_info {
    struct pcb_struct   pcb;        /* palcode state */

    struct task_struct  *task;      /* main task structure */
    unsigned int        flags;      /* low level flags */
    unsigned int        ieee_state; /* see fpu.h */

    mm_segment_t        addr_limit; /* thread address space */

    struct exec_domain  *exec_domain;

    unsigned        cpu;        /* current CPU */
    int         preempt_count; /* 0 => preemptable, <0 => BUG */
    unsigned int        status;     /* thread-synchronous flags */

    int bpt_nsaved;
    unsigned long bpt_addr[2];      /* breakpoint handling  */
    unsigned int bpt_insn[2];
};
```

Linux进程相关的数据结构

- addr_limit通常是一个unsigned long 变量，用于表示地址空间的限制
- bpt_addr提供了断点处理的功能

```
struct thread_info {
    struct pcb_struct   pcb;        /* palcode state */

    struct task_struct  *task;      /* main task structure */
    unsigned int        flags;      /* low level flags */
    unsigned int        ieee_state; /* see fpu.h */

    mm_segment_t        addr_limit; /* thread address space */

    struct exec_domain  *exec_domain;

    unsigned        cpu;        /* current CPU */
    int         preempt_count; /* 0 => preemptable, <0 => BUG */
    unsigned int        status;     /* thread-synchronous flags */

    int bpt_nsaved;
    unsigned long bpt_addr[2];      /* breakpoint handling  */
    unsigned int bpt_insn[2];
};
```

参考资料

[1] https://github.com/ranxian/xv6-chinese

[2] https://qiming.info/Xv6%E5%AD%A6%E4%B9%A0%E5%B0%8F%E8%AE%A12/

[3] https://zh.wikipedia.org/wiki/X86%E8%B0%83%E7%94%A8%E7%BA%A6%E5%AE%9A

[4] Intel® 64 and IA-32 Architectures Software Developer's Manual Volume 2 (2A, 2B, 2C & 2D): Instruction Set Reference, A-Z. September 2016

[5] 陈莉君. 深入分析Linux内核源代码. 人民邮电出版社. 2002-8

[6] http://abcdxyzk.github.io/blog/2014/05/06/kernel-mm-stack/

[7] https://www.cnblogs.com/linhaostudy/p/9557808.html