XV6进程线程源代码-部分阅读

types.h

```
//定义了几个变量的别称
//所以叫做types.h
//uint ushort uchar pde_t

typedef unsigned int   uint;
typedef unsigned short ushort;
typedef unsigned char  uchar;
typedef uint pde_t;

//pde_t与uint是一样的，pde_t用在页表。
```

param.h

```
//parameter.h 设置一些XV6操作系统中的参数


#define NPROC        64  // maximum number of processes 最大的进程数
#define KSTACKSIZE 4096  // size of per-process kernel stack 每个进程内核栈的最大空间是4096字节
#define NCPU          8  // maximum number of CPUs XV6操作系统支持的最大的CPU数是8个
#define NOFILE       16  // open files per process 每个进程打开的最大文件数为16个
#define NFILE       100  // open files per system 整个操作系统打开的最大文件数为100个
#define NBUF         10  // size of disk block cache 磁盘上的缓存大小
#define NINODE       50  // maximum number of active i-nodes 文件系统中最大的i-node数（index_node 文件索引）
#define NDEV         10  // maximum major device number 最多10个驱动设备
#define ROOTDEV       1  // device number of file system root disk 文件系统的根目录放在1号驱动设备上
#define MAXARG       32  // max exec arguments  最大的执行参数的个数是32个，argv中元素的个数
#define LOGSIZE      10  // max data sectors in on-disk log 日志文件的=最大为10扇区 5120字节

```

memlayout.h

```
// Memory layout
//内存布局的情况

#define EXTMEM  0x100000            // Start of extended memory
#define PHYSTOP 0xE000000           // Top physicla memory
#define DEVSPACE 0xFE000000         // Other devices are at high addresses

// Key addresses for address space layout (see kmap in vm.c for layout)
//kmap表
#define KERNBASE 0x80000000         // First kernel virtual address
//从80000000地方开始，可以参考32位机器的elf格式
#define KERNLINK (KERNBASE+EXTMEM)  // Address where kernel is linked

//下面的都是虚拟地址和物理地址之间的转换
//v2p 虚拟地址转换为物理地址
//p2v 物理地址转换为虚拟地址
//为什么要这么转换?
//启动分页了，但是页表还没有设置好，所以这个时候虚拟地址和物理地址之间的转换是线性的转换……
#ifndef __ASSEMBLER__

static inline uint v2p(void *a) { return ((uint) (a))  - KERNBASE; }
static inline void *p2v(uint a) { return (void *) ((a) + KERNBASE); }

#endif

#define V2P(a) (((uint) (a)) - KERNBASE)
#define P2V(a) (((void *) (a)) + KERNBASE)

#define V2P_WO(x) ((x) - KERNBASE)    // same as V2P, but without casts
#define P2V_WO(x) ((x) + KERNBASE)    // same as V2P, but without casts

```

defs.h

定义了各种声明和全局函数，我这里只截取了进程相关的。

```
// proc.c
//常见的进程的定义函数
struct proc*    copyproc(struct proc*);
void            exit(void);
```

```
int             fork(void);
int             growproc(int);//给进程分配更多的内存资源
int             kill(int);
void            pinit(void);
/*
//初始化
void
pinit(void)
{
    //主要是锁初始化
  initlock(&ptable.lock, "ptable");
}
*/
void            procdump(void);//将所有的进程打印出来（避免死锁的打印）
void            scheduler(void) __attribute__((noreturn));//调度函数
void            sched(void);
void            sleep(void*, struct spinlock*);
void            userinit(void);//第一个进程设置好
int             wait(void);
void            wakeup(void*);
void            yield(void);//从running状态跳转到runnable状态

```

x86.h

```
// Routines to let C code use special x86 instructions.
// 采用了内嵌汇编的方式 让C语言能够使用汇编

//介绍一些内嵌汇编的基本格式
//第一个：后面的是输出，第二个：后面的是输入
//第一个：前面的是指令

//涉及到asm中的in指令和out指令
//in指令读外设端口内容
//out指令向外设端口输出
//%1代表edx=port %0代表eax=data
static inline uchar
inb(ushort port)
{
  uchar data;

  asm volatile("in %1,%0" : "=a" (data) : "d" (port));
  return data;
}

static inline void
insl(int port, void *addr, int cnt)
{
  asm volatile("cld; rep insl" :
```

```
                 "=D" (addr), "=c" (cnt) :
                 "d" (port), "0" (addr), "1" (cnt) :
                 "memory", "cc");
}

static inline void
outb(ushort port, uchar data)
{
  asm volatile("out %0,%1" : : "a" (data), "d" (port));
}

static inline void
outw(ushort port, ushort data)
{
  asm volatile("out %0,%1" : : "a" (data), "d" (port));
}

static inline void
outsl(int port, const void *addr, int cnt)
{
  asm volatile("cld; rep outsl" :
                 "=S" (addr), "=c" (cnt) :
                 "d" (port), "0" (addr), "1" (cnt) :
                 "cc");
}

//这个代码用来写入地址用的，就是修改addr的值
//https://docs.microsoft.com/zh-cn/cpp/intrinsics/stosb?view=vs-2015
static inline void
stosb(void *addr, int data, int cnt)
{
  asm volatile("cld; rep stosb" :
                 "=D" (addr), "=c" (cnt) :
                 "0" (addr), "1" (cnt), "a" (data) :
                 "memory", "cc");//cc c compiler
}

static inline void
stosl(void *addr, int data, int cnt)
{
  asm volatile("cld; rep stosl" :
                 "=D" (addr), "=c" (cnt) :
                 "0" (addr), "1" (cnt), "a" (data) :
                 "memory", "cc");
}

struct segdesc;//一个数据结构，定义在mmu.h中
//各种段的描述符数据结构

//加载全局/中断描述符表格寄存器
//load
```

```c
static inline void
lgdt(struct segdesc *p, int size)
{
  volatile ushort pd[3];

  pd[0] = size-1;
  pd[1] = (uint)p;
  pd[2] = (uint)p >> 16;

  asm volatile("lgdt (%0)" : : "r" (pd));
}

struct gatedesc;

static inline void
lidt(struct gatedesc *p, int size)
{
  volatile ushort pd[3];

  pd[0] = size-1;
  pd[1] = (uint)p;
  pd[2] = (uint)p >> 16;

  asm volatile("lidt (%0)" : : "r" (pd));
}

//装载任务状态段寄存器TR
//在任务内发生特权级变换时堆栈也随着自动切换，外层堆栈指针保存在内层堆栈中，而内层堆栈指针存放在当前任务的TSS中。
// 所以，在从外层向内层变换时，要访问TSS(从内层向外层转移时不需要访问TSS，而只需内层栈中保存的栈指针)。

static inline void
ltr(ushort sel)
{
  asm volatile("ltr %0" : : "r" (sel));
}

static inline uint
readeflags(void)
{
  uint eflags;
  asm volatile("pushfl; popl %0" : "=r" (eflags));
  return eflags;
}

//linux内核用于GS访问cpu特定的内存
//GS段寄存器

static inline void
loadgs(ushort v)
{
```

```
  asm volatile("movw %0, %%gs" : : "r" (v));
}

//CLI汇编指令全bai称为Clear Interupt，该指令的作用是禁止中断发生。
//
//STI汇编指令全称为Set Interupt，该指令的作用是允许中断发生。

static inline void
cli(void)
{
  asm volatile("cli");
}

static inline void
sti(void)
{
  asm volatile("sti");
}

//交换两个寄存器的值

static inline uint
xchg(volatile uint *addr, uint newval)
{
  uint result;

  // The + in "+m" denotes a read-modify-write operand.
  asm volatile("lock; xchgl %0, %1" :
               "+m" (*addr), "=a" (result) :
               "1" (newval) :
               "cc");
  return result;
}

//读CR2寄存器
//CR2是页故障线性地址寄存器，保存最后一次出现页故障的全32位线性地址。

static inline uint
rcr2(void)
{
  uint val;
  asm volatile("movl %%cr2,%0" : "=r" (val));
  return val;
}
//CR3是页目录基址寄存器，保存页目录表的物理地址，页目录表总是放在以4K字节为单位的存储器边界上，因此，它的地址的低12位总为0
/*
 * //////CR0中包含了6个预定义标志，0位是保护允许位PE(Protedted Enable)，用于启动保护模式，
 * 如果PE位置1，则保护模式启动，如果PE=0，则在实模式下运行。
 * 1位是监控协处理位MP(Moniter coprocessor)，它与第3位一起决定：当TS=1时操作码WAIT是否产生一个“协处理器不能使用”的出错信号。
```

```
 * 第3位是任务转换位(Task Switch)，当一个任务转换完成之后，自动将它置1。随着TS=1，就不能使用协处理器。
 * CR0的第2位是模拟协处理器位 EM (Emulate coprocessor)，如果EM=1，则不能使用协处理器，如果EM=0，则允许使用协处理器。
 * 第4位是微处理器的扩展类型位ET(Processor Extension Type)，其内保存着处理器扩展类型的信息，
 * 如果ET=0，则标识系统使用的是287协处理器，如果 ET=1，则表示系统使用的是387浮点协处理器。
 * ////////CR0的第31位是分页允许位(Paging Enable)，它表示芯片上的分页部件是否允许工作。
 */
static inline void
lcr3(uint val)
{
  asm volatile("movl %0,%%cr3" : : "r" (val));
}

//PAGEBREAK: 36
// Layout of the trap frame built on the stack by the
// hardware and by trapasm.S, and passed to trap().

//用户态陷入内核态的数据结构

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 defined by x86 hardware
  uint err;
  uint eip;
  ushort cs;
  ushort padding5;
  uint eflags;

```

```
  // below here only when crossing rings, such as from user to kernel
  uint esp;
  ushort ss;
  ushort padding6;
};

```

asm.h

这个基本上完全看不懂，只弄懂了边界地址一定是4K的整数倍……

```
//
// assembler macros to create x86 segments
//

//汇编程序宏来创建x86段

// 汇编的宏定义

#define SEG_NULLASM                                             \
        .word 0, 0;                                             \
        .byte 0, 0, 0, 0

// The 0xC0 means the limit is in 4096-byte units
//4096字节正好就是4k，无论是虚拟地址空间还是物理内存，都是按照4k划分的
// and (for executable segments) 32-bit mode.

//find in path 全局搜索
//生成elf中各种段
//看不懂……
#define SEG_ASM(type,base,lim)                                  \
        .word (((lim) >> 12) & 0xffff), ((base) & 0xffff);      \
        .byte (((base) >> 16) & 0xff), (0x90 | (type)),         \
                (0xC0 | (((lim) >> 28) & 0xf)), (((base) >> 24) & 0xff)

#define STA_X     0x8       // Executable segment
#define STA_E     0x4       // Expand down (non-executable segments)
#define STA_C     0x4       // Conforming code segment (executable only)
#define STA_W     0x2       // Writeable (non-executable segments)
#define STA_R     0x2       // Readable (executable segments)
#define STA_A     0x1       // Accessed
```

```
//对应elf中的各种段......
```

mmu.h

定义了一些寄存器的值，一些数据结构，x86 memory management。

```
// This file contains definitions for the
// x86 memory management unit (MMU).
// 内存管理单元，进程地址空间详细数据结构

// Eflags register

//详细的定义了eflags这个寄存器
//32位的 下面正好也32位
#define FL_CF           0x00000001      // Carry Flag
#define FL_PF           0x00000004      // Parity Flag
#define FL_AF           0x00000010      // Auxiliary carry Flag
#define FL_ZF           0x00000040      // Zero Flag
#define FL_SF           0x00000080      // Sign Flag
#define FL_TF           0x00000100      // Trap Flag
#define FL_IF           0x00000200      // Interrupt Enable
#define FL_DF           0x00000400      // Direction Flag
#define FL_OF           0x00000800      // Overflow Flag
#define FL_IOPL_MASK    0x00003000      // I/O Privilege Level bitmask
#define FL_IOPL_0       0x00000000      //   IOPL == 0
#define FL_IOPL_1       0x00001000      //   IOPL == 1
#define FL_IOPL_2       0x00002000      //   IOPL == 2
#define FL_IOPL_3       0x00003000      //   IOPL == 3
#define FL_NT           0x00004000      // Nested Task
#define FL_RF           0x00010000      // Resume Flag
#define FL_VM           0x00020000      // Virtual 8086 mode
#define FL_AC           0x00040000      // Alignment Check
#define FL_VIF          0x00080000      // Virtual Interrupt Flag
#define FL_VIP          0x00100000      // Virtual Interrupt Pending
#define FL_ID           0x00200000      // ID flag

// Control Register flags
//定义了32位的CR0寄存器
//CR0中含有控制处理器操作模式和状态的系统控制标志
#define CR0_PE          0x00000001      // Protection Enable
#define CR0_MP          0x00000002      // Monitor coProcessor
#define CR0_EM          0x00000004      // Emulation
#define CR0_TS          0x00000008      // Task Switched
#define CR0_ET          0x00000010      // Extension Type
#define CR0_NE          0x00000020      // Numeric Errror
#define CR0_WP          0x00010000      // Write Protect
#define CR0_AM          0x00040000      // Alignment Mask
#define CR0_NW          0x20000000      // Not Writethrough
```

```
#define CR0_CD          0x40000000      // Cache Disable
#define CR0_PG          0x80000000      // Paging

#define CR4_PSE         0x00000010      // Page size extension


//定义了elf格式中的各种段
#define SEG_KCODE 1  // kernel code
#define SEG_KDATA 2  // kernel data+stack
#define SEG_KCPU  3  // kernel per-cpu data
#define SEG_UCODE 4  // user code
#define SEG_UDATA 5  // user data+stack
#define SEG_TSS   6  // this process's task state

//PAGEBREAK!
#ifndef __ASSEMBLER__// if not define
// Segment Descriptor 段描述符
//看到这个数据结构，我终于对32位机器上的地址生成有了直观的感受……
struct segdesc {
  uint lim_15_0 : 16;  // Low bits of segment limit
  uint base_15_0 : 16; // Low bits of segment base address
  uint base_23_16 : 8; // Middle bits of segment base address
  uint type : 4;       // Segment type (see STS_ constants)
  uint s : 1;          // 0 = system, 1 = application
  uint dpl : 2;        // Descriptor Privilege Level
  uint p : 1;          // Present
  uint lim_19_16 : 4;  // High bits of segment limit
  uint avl : 1;        // Unused (available for software use)
  uint rsv1 : 1;       // Reserved
  uint db : 1;         // 0 = 16-bit segment, 1 = 32-bit segment
  uint g : 1;          // Granularity: limit scaled by 4K when set
  uint base_31_24 : 8; // High bits of segment base address
};

```

elf.h

看名字就知道了，ELF的头文件。

```
// Format of an ELF executable file
// 定义了可执行文件的ELF格式

#define ELF_MAGIC 0x464C457FU  // "\x7FELF" in little endian
//elf中叫做魔数的概念
//最开始的4个字节是所有ELF文件都必须相同的标识码，分别为0x7F、0x45、0x4c、0x46
//为什么要倒着写？小端模式……

// File header
```

```c
struct elfhdr {
  uint magic;  // must equal ELF_MAGIC
  uchar elf[12];
  ushort type;
  ushort machine;
  uint version;
  uint entry;
  uint phoff;
  uint shoff;
  uint flags;
  ushort ehsize;
  ushort phentsize;
  ushort phnum;
  ushort shentsize;
  ushort shnum;
  ushort shstrndx;
};

// Program section header

//定义的这两个数据结构是有差别的
//https://baike.baidu.com/pic/ELF/7120560/0/279759ee3d6d55fb68df1ab16f224f4a20a4dd08?fr=lemma&ct=single
struct proghdr {
  uint type;
  uint off;
  uint vaddr;
  uint paddr;
  uint filesz;
  uint memsz;
  uint flags;
  uint align;
};

// Values for Proghdr type
#define ELF_PROG_LOAD           1

// Flag bits for Proghdr flags
//定义了一些标志位
#define ELF_PROG_FLAG_EXEC      1
#define ELF_PROG_FLAG_WRITE     2
#define ELF_PROG_FLAG_READ      4
```

proc.h

```c
//声明了cpu、进程、进程上下文等数据结构
//声明了proc.c中要使用的数据结构
//proc.h中没有实现线程结构
// Segments in proc->gdt.
#define NSEGS     7

```

```
// Per-CPU state
//定义了一个CPU结构
struct cpu {
  uchar id;                    // Local APIC ID; index into cpus[] below//extern struct cpu cpus[NCPU];//索引号
  struct context *scheduler;   // switch() here to enter scheduler//调度的时候保存上下文环境
  struct taskstate ts;         // Used by x86 to find stack for interrupt//任务状态 用在完成中断之后恢复自己的状态
  struct segdesc gdt[NSEGS];   // x86 global descriptor table//全局描述符表……
  volatile uint started;       // Has the CPU started?
  //https://blog.csdn.net/qq_25426415/article/details/54631192
  //关闭终端，记录关中断的次数，与锁有关……
  int ncli;                    // Depth of pushcli nesting.
  int intena;                  // Were interrupts enabled before pushcli?

  // Cpu-local storage variables; see below
  struct cpu *cpu;//定义了一个自己的指针
  //当前跑在这个CPU上面的进程
  struct proc *proc;           // The currently-running process.
};

//所以说，XV6是可用的，它支持多CPU
extern struct cpu cpus[NCPU];//结构数组
extern int ncpu; //定义了CPU的个数

// Per-CPU variables, holding pointers to the
// current cpu and to the current process.
// The asm suffix tells gcc to use "%gs:0" to refer to cpu
// and "%gs:4" to refer to proc.  seginit sets up the
// %gs segment register so that %gs refers to the memory
// holding those two variables in the local cpu's struct cpu.
// This is similar to how thread-local variables are implemented
// in thread libraries such as Linux pthreads.
//通过指针cpu和proc，能够准确引用当前所在CPU的cpu结构体
//https://blog.csdn.net/qq_25426415/article/details/54619488
extern struct cpu *cpu asm("%gs:0");       // &cpus[cpunum()]
extern struct proc *proc asm("%gs:4");     // cpus[cpunum()].proc

//PAGEBREAK: 17
// Saved registers for kernel context switches.
// Don't need to save all the segment registers (%cs, etc),
// because they are constant across kernel contexts.
// Don't need to save %eax, %ecx, %edx, because the
// x86 convention is that the caller has saved them.
// Contexts are stored at the bottom of the stack they
// describe; the stack pointer is the address of the context.
```

```c
// The layout of the context matches the layout of the stack in swtch.S
// at the "Switch stacks" comment. Switch doesn't save eip explicitly,
// but it is on the stack and allocproc() manipulates it.
//保存上下文的寄存器 在上下文切换的时候用得着
//为内核上下文切换保存的寄存器。
////不需要保存所有的段寄存器
struct context {
  uint edi;
  uint esi;
  uint ebx;
  uint ebp;
  uint eip;
};

//进程的六种状态
//未使用，胚胎(初期)，睡眠，可运行，正在运行，僵尸
enum procstate { UNUSED, EMBRYO, SLEEPING, RUNNABLE, RUNNING, ZOMBIE };

// Per-process state
//定义了进程的结构
struct proc {
  uint sz;                     // Size of process memory (bytes)//进程内存的大小（字节）
  pde_t* pgdir;                // Page table//页表的指针
  char *kstack;                // Bottom of kernel stack for this process//内核的最低位置在哪里
  enum procstate state;        // Process state //进程的状态 六种之一
  volatile int pid;            // Process ID //进程ID
  struct proc *parent;         // Parent process //父进程
  struct trapframe *tf;        // Trap frame for current syscall //当该进程启动系统调用的时候应该保存的信息
  struct context *context;     // swtch() here to run process //进程切换应该保存的寄存器信息
  void *chan;                  // If non-zero, sleeping on chan //阻塞位的标志
  int killed;                  // If non-zero, have been killed //被kill的标志位
  struct file *ofile[NOFILE];  // Open files //打开的文件表
  struct inode *cwd;           // Current directory //当前目录
  char name[16];               // Process name (debugging) //程序名字
};

//讲了一下进程地址空间
// Process memory is laid out contiguously, low addresses first:
//   text
//   original data and bss
```

```
//   fixed-size stack
//   expandable heap

```

proc.c

```
//一些进程创建 退出 等待的具体函数
#include "types.h"
#include "defs.h"
#include "param.h"
#include "memlayout.h"
#include "mmu.h"
#include "x86.h"
#include "proc.h"
#include "spinlock.h"

struct {
  struct spinlock lock;//互斥锁
  struct proc proc[NPROC];//NPROC=64
} ptable;//进程表
////进程索引表，64个进程以数组的形式记录

static struct proc *initproc;//初始进程，userinit调用中赋值

int nextpid = 1;
//forkret函数的函数体在后面
extern void forkret(void);
//陷入返回
extern void trapret(void);

static void wakeup1(void *chan);

//初始化
void
pinit(void)
{
    //主要是锁初始化
  initlock(&ptable.lock, "ptable");
}

//PAGEBREAK: 32
// Look in the process table for an UNUSED proc.
// If found, change state to EMBRYO and initialize
// state required to run in the kernel.
// Otherwise return 0.
static struct proc*
allocproc(void)
{
  struct proc *p;
  char *sp;

```

```
  //锁住&ptable
  acquire(&ptable.lock);
  for(p = ptable.proc; p < &ptable.proc[NPROC]; p++)
    if(p->state == UNUSED)//找到未被使用的进程
      goto found;
  release(&ptable.lock);//解锁
  //如果没有找到，连一个unused的进程都找不到
  return 0;//返回0

found:
  p->state = EMBRYO;//将未被使用的进程设置为胚胎模式，初期模式……反正就这么叫……
  p->pid = nextpid++;
  release(&ptable.lock);

  // Allocate kernel stack.分配内核栈
  if((p->kstack = kalloc()) == 0){
      //*kalloc()函数的作用是分配一个4096B大小的空间，而在param.h中规定了一个进程的内核栈大小为4096*/
    p->state = UNUSED;//分配失败，状态变回UNUSED
    return 0;
  }
  sp = p->kstack + KSTACKSIZE;//栈指针移动，因为已经分配好了内核栈

  // Leave room for trap frame.
  //不理解……
  //? ? ?
  sp -= sizeof *p->tf;
  p->tf = (struct trapframe*)sp;

  // Set up new context to start executing at forkret,
  // which returns to trapret.
  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;

  return p;
}

//PAGEBREAK: 32
// Set up first user process.
//设置第一个用户进程
void
userinit(void)
{
  struct proc *p;
  extern char _binary_initcode_start[], _binary_initcode_size[];
```

```

  p = allocproc();//设置好了内核应该设置的东西
  initproc = p;//将initproc作为第一个进程，所有进程的父亲
  if((p->pgdir = setupkvm()) == 0)
    panic("userinit: out of memory?");
  inituvm(p->pgdir, _binary_initcode_start, (int)_binary_initcode_size);//分配页表空间
  p->sz = PGSIZE;
  memset(p->tf, 0, sizeof(*p->tf));
  p->tf->cs = (SEG_UCODE << 3) | DPL_USER;//做DPl的分配 CS段中有DPL=3 代表用户态
  p->tf->ds = (SEG_UDATA << 3) | DPL_USER;
  p->tf->es = p->tf->ds;
  p->tf->ss = p->tf->ds;
  p->tf->eflags = FL_IF;
  p->tf->esp = PGSIZE;
  p->tf->eip = 0;  // beginning of initcode.S

  safestrcpy(p->name, "initcode", sizeof(p->name));
  p->cwd = namei("/");////分配目录 cwd Current directory 当前分配的是根目录 /

  p->state = RUNNABLE;
}

// Grow current process's memory by n bytes.
// Return 0 on success, -1 on failure.
//给进程空间增加内存
int
growproc(int n)
{
  uint sz;

  sz = proc->sz;
  if(n > 0){
    if((sz = allocuvm(proc->pgdir, sz, sz + n)) == 0)
      return -1;
  } else if(n < 0){
    if((sz = deallocuvm(proc->pgdir, sz, sz + n)) == 0)
      return -1;
  }
  proc->sz = sz;
  switchuvm(proc);
  return 0;
}

// Create a new process copying p as the parent.
// Sets up stack to return as if from system call.
// Caller must set state of returned proc to RUNNABLE.

//创建一个新进程，将p复制为父进程。
//设置堆栈以使其好像从系统调用中返回一样。
```

```
//调用者必须将返回的proc的状态设置为RUNNABLE。
int
fork(void)
{
  int i, pid;
  struct proc *np;

  // Allocate process.
  if((np = allocproc()) == 0)//allocproc
    return -1;

  // Copy process state from p.
  if((np->pgdir = copyuvm(proc->pgdir, proc->sz)) == 0){
    kfree(np->kstack);
    np->kstack = 0;
    np->state = UNUSED;
    return -1;
  }
  np->sz = proc->sz;
  np->parent = proc;//父子关系
  *np->tf = *proc->tf;////理解为所有进程公用一个内核栈tf（我不太理解……）

  // Clear %eax so that fork returns 0 in the child.
  np->tf->eax = 0;

  for(i = 0; i < NOFILE; i++)
    if(proc->ofile[i])
      np->ofile[i] = filedup(proc->ofile[i]);//复制打开的文件表
  np->cwd = idup(proc->cwd);

  pid = np->pid;
  np->state = RUNNABLE;//fork出来的子进程 状态设置为可运行
  safestrcpy(np->name, proc->name, sizeof(proc->name));
  return pid;
}

// Exit the current process.  Does not return.
// An exited process remains in the zombie state
// until its parent calls wait() to find out it exited.
void
exit(void)
{
  struct proc *p;
  int fd;

  if(proc == initproc)//如果是initproc被exit painc
      /*void
      panic(char *s)
      {
          printf(2, "%s\n", s);
          exit();
```

```
      }*/
    panic("init exiting");

  // Close all open files.
  //关闭所有的文件
  for(fd = 0; fd < NOFILE; fd++){
    if(proc->ofile[fd]){
      fileclose(proc->ofile[fd]);
      proc->ofile[fd] = 0;
    }
  }

  iput(proc->cwd);
  proc->cwd = 0;

  acquire(&ptable.lock);

  // Parent might be sleeping in wait().
  //唤醒它的父进程
  wakeup1(proc->parent);

  // Pass abandoned children to init.
  //将它的子进程给init，防止孤儿进程，僵尸进程的出现
  for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
    if(p->parent == proc){
      p->parent = initproc;
      if(p->state == ZOMBIE)
        wakeup1(initproc);
    }
  }

  // Jump into the scheduler, never to return.
  //将自己设置为僵尸进程
  proc->state = ZOMBIE;
  //exit之后就到了调度时机
  sched();
  panic("zombie exit");
}

// Wait for a child process to exit and return its pid.
// Return -1 if this process has no children.
//父进程等待，等待子进程的pid
//扫描所有进程，如果进程没有子进程，则返回 - 1；如果有子进程且找到一个已进入
//zombie状态，则先对其资源进行回收，然后返回其pid；如果有子进程但都没有进入zombie状
//态，则进入sleeping状态。其中：
int
wait(void)
{
  struct proc *p;
  int havekids, pid;
```

```

  acquire(&ptable.lock);//锁住
  for(;;){
    // Scan through table looking for zombie children.
    havekids = 0;
    for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
      if(p->parent != proc)
        continue;
      havekids = 1;
      if(p->state == ZOMBIE){
        // Found one.
        pid = p->pid;
        kfree(p->kstack);
        p->kstack = 0;
        freevm(p->pgdir);
        p->state = UNUSED;
        p->pid = 0;
        p->parent = 0;
        p->name[0] = 0;
        p->killed = 0;
        release(&ptable.lock);
        return pid;
      }
    }

    // No point waiting if we don't have any children.
    if(!havekids || proc->killed){
      release(&ptable.lock);
      return -1;
    }

    // Wait for children to exit.  (See wakeup1 call in proc_exit.)
    sleep(proc, &ptable.lock);  //DOC: wait-sleep
  }
}

//PAGEBREAK: 42
// Per-CPU process scheduler.
// Each CPU calls scheduler() after setting itself up.
// Scheduler never returns.  It loops, doing:
//  - choose a process to run
//  - swtch to start running that process
//  - eventually that process transfers control
//      via swtch back to the scheduler.
//-选择要运行的进程
//-swtch开始运行该进程
//-最终该过程转移了控制权
//通过swtch返回调度程序。
void
scheduler(void)
{
```

```
  struct proc *p;

  for(;;){
    // Enable interrupts on this processor.
    sti();

    // Loop over process table looking for process to run.
    acquire(&ptable.lock);
    for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
      if(p->state != RUNNABLE)//简单的遍历 找到第一个runnable......这么直白的吗......
        continue;

      // Switch to chosen process.  It is the process's job
      // to release ptable.lock and then reacquire it
      // before jumping back to us.
      proc = p;
      switchuvm(p);
      p->state = RUNNING;
      swtch(&cpu->scheduler, proc->context);
      switchkvm();

      // Process is done running for now.
      // It should have changed its p->state before coming back.
      proc = 0;
    }
    release(&ptable.lock);

  }
}

// Enter scheduler.  Must hold only ptable.lock
// and have changed proc->state.
void
sched(void)
{
  int intena;

  if(!holding(&ptable.lock))
    panic("sched ptable.lock");
  if(cpu->ncli != 1)
    panic("sched locks");
  if(proc->state == RUNNING)
    panic("sched running");
  if(readeflags()&FL_IF)
    panic("sched interruptible");
  intena = cpu->intena;
  swtch(&proc->context, cpu->scheduler);
  cpu->intena = intena;
}

```

```
// Give up the CPU for one scheduling round.
//yield过程将running转变成runnable
void
yield(void)
{
  acquire(&ptable.lock);  //DOC: yieldlock
  proc->state = RUNNABLE;
  sched();//CPU空出来了就要执行调度了
  release(&ptable.lock);
}

// A fork child's very first scheduling by scheduler()
// will swtch here.  "Return" to user space.
void
forkret(void)
{
  static int first = 1;
  // Still holding ptable.lock from scheduler.
  release(&ptable.lock);

  if (first) {
    // Some initialization functions must be run in the context
    // of a regular process (e.g., they call sleep), and thus cannot
    // be run from main().
    first = 0;
    initlog();
  }

  // Return to "caller", actually trapret (see allocproc).
}

// Atomically release lock and sleep on chan.
// Reacquires lock when awakened.
void
sleep(void *chan, struct spinlock *lk)
{
  if(proc == 0)
    panic("sleep");

  if(lk == 0)
    panic("sleep without lk");

  // Must acquire ptable.lock in order to
  // change p->state and then call sched.
  // Once we hold ptable.lock, we can be
  // guaranteed that we won't miss any wakeup
  // (wakeup runs with ptable.lock locked),
  // so it's okay to release lk.
  if(lk != &ptable.lock){  //DOC: sleeplock0
    acquire(&ptable.lock);  //DOC: sleeplock1
```

```
    release(lk);
  }

  // Go to sleep.
  proc->chan = chan;
  proc->state = SLEEPING;
  sched();

  // Tidy up.
  proc->chan = 0;

  // Reacquire original lock.
  if(lk != &ptable.lock){  //DOC: sleeplock2
    release(&ptable.lock);
    acquire(lk);
  }
}

//PAGEBREAK!
// Wake up all processes sleeping on chan.
// The ptable lock must be held.
static void
wakeup1(void *chan)
{
  struct proc *p;

  for(p = ptable.proc; p < &ptable.proc[NPROC]; p++)//很简单的循环
    if(p->state == SLEEPING && p->chan == chan)
      p->state = RUNNABLE;//改变状态到runnable
}

// Wake up all processes sleeping on chan.
//为什么要设置两个这样的函数，wakeup和wakeup1?
void
wakeup(void *chan)
{
  acquire(&ptable.lock);
  wakeup1(chan);//调用wakeup1
  release(&ptable.lock);
}

// Kill the process with the given pid.
// Process won't exit until it returns
// to user space (see trap in trap.c).
int
kill(int pid)
{
  struct proc *p;

  acquire(&ptable.lock);//锁
  for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
```

```
    if(p->pid == pid){
      p->killed = 1;//killed设置为1
      // Wake process from sleep if necessary.
      //也可以在这里将阻塞进程变成可运行
      //为什么要在这里设置这样一个功能？
      //CPU空出来了？
      if(p->state == SLEEPING)
        p->state = RUNNABLE;
      release(&ptable.lock);//解锁
      return 0;
    }
  }
  release(&ptable.lock);
  return -1;
}

//PAGEBREAK: 36
// Print a process listing to console.  For debugging.
// Runs when user types ^P on console.
// No lock to avoid wedging a stuck machine further.
//将所有的进程打印在控制台上
//没有死锁
void
procdump(void)
{
  static char *states[] = {
  [UNUSED]    "unused",
  [EMBRYO]    "embryo",
  [SLEEPING]  "sleep ",
  [RUNNABLE]  "runble",
  [RUNNING]   "run   ",
  [ZOMBIE]    "zombie"
  };
  int i;
  struct proc *p;
  char *state;
  uint pc[10];

  for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
    if(p->state == UNUSED)
      continue;
    if(p->state >= 0 && p->state < NELEM(states) && states[p-
>state])
      state = states[p->state];
    else
      state = "???";
    cprintf("%d %s %s", p->pid, state, p->name);
    if(p->state == SLEEPING){
      getcallerpcs((uint*)p->context->ebp+2, pc);
      for(i=0; i<10 && pc[i] != 0; i++)
        cprintf(" %p", pc[i]);
    }
```

```
    cprintf("\n");
  }
}



```

swtch.S

```
# Context switch
//上下文切换
#
#   void swtch(struct context **old, struct context *new);
#
# Save current register context in old
# and then load register context from new.

.globl swtch
swtch:
  movl 4(%esp), %eax//利用栈来传递传参数，32位机器
  movl 8(%esp), %edx

  # Save old callee-save registers
  pushl %ebp
  pushl %ebx
  pushl %esi
  pushl %edi

  # Switch stacks
  movl %esp, (%eax)//一个打了括号 一个不打括号 修改指针本身的内容 间接寻址
  movl %edx, %esp

  # Load new callee-save registers
  popl %edi
  popl %esi
  popl %ebx
  popl %ebp
  ret

```