XV6 进程模型 阅读报告

本次代码阅读主要对 XV6 中的进程模型进行调研，主要涉及的代码有：proc.h，proc.c，switch.S，除此之外还会涉及到的辅助代码有 vm.c，kalloc.c 等。下面我将逐个剖析，以阐释 XV6 的进程模型机制。

一、进程概览

首先，让我们来分析一下 proc.h 这个头文件，它包含了与 cpu 和进程相关的许多核心结构的声明与定义。

1.CPU 状态

struct cpu 包含了每个 CPU 的状态信息。

```
4   // Per-CPU state
5   struct cpu {
6     uchar id;                    // Local APIC ID; index into cpus[] below
7     struct context *scheduler;   // swtch() here to enter scheduler
8     struct taskstate ts;         // Used by x86 to find stack for interrupt
9     struct segdesc gdt[NSEGS];   // x86 global descriptor table
10    volatile uint started;       // Has the CPU started?
11    int ncli;                    // Depth of pushcli nesting.
12    int intena;                  // Were interrupts enabled before pushcli?
13
14    // Cpu-local storage variables; see below
15    struct cpu *cpu;
16    struct proc *proc;           // The currently-running process.
17  };
```

id 是一个无符号字符，它代表着当前 CPU 在全局 CPU 数组中的下标。

scheduler 是一个 context 结构的指针，指示 cpu 目前执行的进程的上下文结构。

ts 是一个 taskstate 结构，当中断发生时，x86 使用这个结构来寻找栈指针。这个结构定义在 mmu.h 中，其中包含了主要寄存器的值、权限级等内容。

```
150 struct taskstate {
151   uint link;
152   uint esp0;
153   ushort ss0;
154   ushort padding1;
155   uint *esp1;
156   ushort ss1;
157   ushort padding2;
158   uint *esp2;
159   ushort ss2;
160   ushort padding3;
161   void *cr3;
162   uint *eip;
163   uint eflags;
164   uint eax;
165   uint ecx;
166   uint edx;
167   uint ebx;
168   uint *esp;
169   uint *ebp;
170   uint esi;
171   uint edi;
172   ushort es;
173   ushort padding4;
174   ushort cs;
175   ushort padding5;
176   ushort ss;
177   ushort padding6;
178   ushort ds;
179   ushort padding7;
180   ushort fs;
181   ushort padding8;
182   ushort gs;
183   ushort padding9;
184   ushort ldt;
185   ushort padding10;
186   ushort t;
187   ushort iomb;
188 };
```

gdt 是段描述符数组，作为 x86 体系中的全局描述符表。segdesc 结构定义在 mmu.h 中，包含了段基址、段特权级等内容。GDT 表用于系统的中断处理，指示中断处理程序的入口位

置。

```
52  struct segdesc {
53    uint lim_15_0 : 16;  // Low bits of segment limit
54    uint base_15_0 : 16; // Low bits of segment base address
55    uint base_23_16 : 8; // Middle bits of segment base address
56    uint type : 4;       // Segment type (see STS_ constants)
57    uint s : 1;          // 0 = system, 1 = application
58    uint dpl : 2;        // Descriptor Privilege Level
59    uint p : 1;          // Present
60    uint lim_19_16 : 4;  // High bits of segment limit
61    uint avl : 1;        // Unused (available for software use)
62    uint rsv1 : 1;       // Reserved
63    uint db : 1;         // 0 = 16-bit segment, 1 = 32-bit segment
64    uint g : 1;          // Granularity: limit scaled by 4K when set
65    uint base_31_24 : 8; // High bits of segment base address
66  };
```

started 用于指明当前 CPU 是否已经启动。

ncli 指明 pushcli 操作的深度，intena 指明在 pushcli 操作之前是否恢复了中断响应。

*cpu 储存当前 cpu 的指针，*proc 储存指向当前正在运行的进程的结构的指针。这两个变量的赋值方式如下：

```
30  extern struct cpu *cpu asm("%gs:0");       // &cpus[cpunum()]
31  extern struct proc *proc asm("%gs:4");     // cpus[cpunum()].proc
```

asm 后缀指明了这两个指针所在的地址。其中，%gs 寄存器的值由 seginit 函数设置。第一条语句等价于对当前 CPU 在全局 CPU 数组中的位置取址；第二条语句等价于取到对应的 proc 成员值。

2.全局 CPU 数组

储存了整个系统的 CPU，NCPU 在 param.h 中，定义为 8。这两个全局变量在 proc.h 中声明，在 mp.c 中被定义和初始化。

```
19  extern struct cpu cpus[NCPU];
20  extern int ncpu;
```

3.进程上下文

context 结构储存了在上下文切换时必须保存的寄存器数值。其中包含%edi，%esi，%ebx，%ebp，%eip 的值。由于部分段寄存器在进程上下文中恒为常数，故不需要保存所有的段寄存器。此外，不需要保存%eax，%ecx，%edx，因为 x86 的管理是这些寄存器由调用者保存。context 结构中的寄存器值顺序与 switch.S 的操作顺序相同。%eip 由 allocproc()函数来操作。

```
44  struct context {
45    uint edi;
46    uint esi;
47    uint ebx;
48    uint ebp;
49    uint eip;
50  };
```

4.进程结构体

整个进程的所有信息保存在 proc 结构中，这个结构相当于 XV6 的 PCB 进程控制块。

```
54  // Per-process state
55  struct proc {
56    uint sz;
57    pde_t* pgdir;
58    char *kstack;
59    enum procstate state;
60    volatile int pid;
61    struct proc *parent;
62    struct trapframe *tf;
63    struct context *context;
64    void *chan;
65    int killed;
66    struct file *ofile[NOFILE];
67    struct inode *cwd;
68    char name[16];
69  };
```

sz 指明了当前进程的内存大小，以字节为单位。

pgdir 是 pde_t（实际上是 uint 的一个 typedef）的指针，指示页表的地址。

kstack 是一个 char 指针，指向进程对应的内核栈的底部。进程执行用户指令时，内核栈为空，仅使用用户栈；进程通过系统调用或者中断进入内核时，内核代码使用 kstack 指向的内核栈空间执行，而用户栈仍然保存原来的数据信息。内核栈不可以被用户代码访问。

state 是一个枚举变量，代表进程目前的状态。进程的状态一共有 6 种（未使用、创建、睡眠、可运行、运行、僵尸）。

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

pid 是进程的 id 号，作为进程唯一的标识符。

tf 是 trap frame 结构的指针。这个结构定义于 x86.h 中，用于系统调用，由硬件和 trapasm.S 共同生成，保存用户寄存器，其中，%esp 是栈指针，%cs 是段区，%eflags 指示硬件中断，具体内容如下：

```
150 struct trapframe {
151   // registers as pushed by pusha
152   uint edi;
153   uint esi;
154   uint ebp;
155   uint oesp;      // useless & ignored
156   uint ebx;
157   uint edx;
158   uint ecx;
159   uint eax;
160
161   // rest of trap frame
162   ushort gs;
163   ushort padding1;
164   ushort fs;
165   ushort padding2;
166   ushort es;
167   ushort padding3;
168   ushort ds;
169   ushort padding4;
170   uint trapno;
171
172   // below here defined by x86 hardware
173   uint err;
174   uint eip;
175   ushort cs;
176   ushort padding5;
177   uint eflags;
178
179   // below here only when crossing rings, such as from user to kernel
180   uint esp;
181   ushort ss;
182   ushort padding6;
183 };
```

context 为进程上下文结构指针，指向当前进程对应的进程上下文。当 swtch 进行进程切换时，会加载新进程的 context 到对应 cpu 中的 context。

chan 是一个指针，用于指明进程是否睡眠。如果这个指针非空，说明进程处于睡眠状态。

killed 表示进程是否被杀死。如果值不为 0，说明进程已经被杀死。

ofile 是一个 file 结构的指针数组，代表当前进程的打开文件表。NOFILE 宏定义在 param.h 中，值为 16。file 结构定义在 file.h 中，指明 XV6 文件中的必需信息。

cwd 是一个 inode 结构的指针，指明当前进程所在的目录。inode 结构定义在 file.h 中，作为文件的唯一标识，存储文件的元数据。

name 是一个字符数组，表示进程名称，用于 debug。

5.进程表

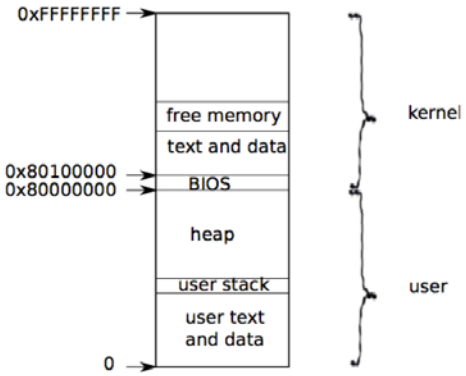
进程表 ptable 定义于 proc.c 中，储存了当前 cpu 创建的所有进程。最大个数为 NPROC，这个宏定义于 param.h 中，值为 64。除此之外，进程表还包含一个自旋锁 lock，用于处理竞争问题。

```
10  struct {
11    struct spinlock lock;
12    struct proc proc[NPROC];
13  } ptable;
```

6.进程地址空间

以上我们分析了 XV6 中 cpu 的内容、进程 PCB 的内容，以及整个系统的 CPU 数组和每个 CPU 的进程表。下面我们来看一下 XV6 中进程的内存地址布局。

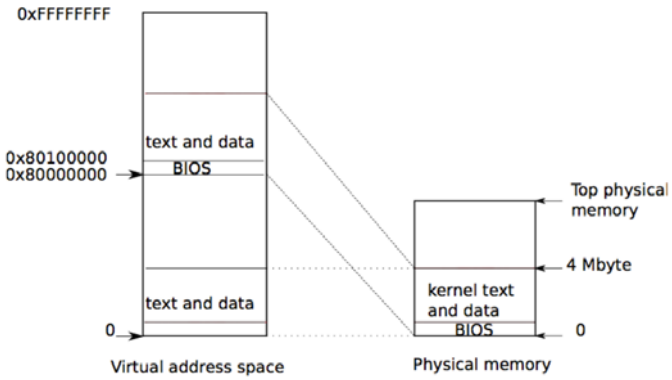
XV6 的地址空间由页表来管理（pgdir），进程的虚拟地址空间布局如下：



地址空间分为用户内存和内核内存两部分。用户内存从 0 地址开始，由低地址到高地址，内容依次为：用户指令、初始 data 段和 bss 数据段、固定大小的栈空间、可扩展的堆空间。内核地址空间包含 BIOS 区域、内核的指令段和数据段，以及内核的自由内存空间。

当 PC 开机时，boot loader 会把 XV6 内核从磁盘中载入。这时分页硬件还未开始运作，虚拟地址直接映射到物理地址上。boot loader 会把内核装载到 0x100000 处，以避免小型机器没有 0x80100000 这么大的内存，且 0xa000 到 0x100000 属于 I/O 设备。

接下来，entry 函数（位于 entry.S 中的汇编代码）设置页表，把 KERNBASE 地址（0x80000000）映射到物理地址 0x0 处。除此之外，还会设置页目录和栈指针，然后跳转到 main 函数开始运行。下图为虚拟地址与物理地址之间的映射。



二、进程的创建与运行

上面我们简要分析了与进程相关的数据结构，以及进程的地址空间，接下来我们以 main 函数创建的第一个进程为例，来分析一下进程的创建与运行过程。

1.main 函数创建第一个进程

main 函数在完成必要的初始化后，通过调用 userinit 函数来创建第一个进程。这个函数位于 proc.c 中，大致流程如下：

```
78  void
79  userinit(void)
80  {
81    struct proc *p;
82    extern char _binary_initcode_start[], _binary_initcode_size[];
83
84    p = allocproc();
85    initproc = p;
86    if((p->pgdir = setupkvm()) == 0)
87      panic("userinit: out of memory?");
88    inituvm(p->pgdir, _binary_initcode_start, (int)_binary_initcode_size);
89    p->sz = PGSIZE;
90    memset(p->tf, 0, sizeof(*p->tf));
91    p->tf->cs = (SEG_UCODE << 3) | DPL_USER;
92    p->tf->ds = (SEG_UDATA << 3) | DPL_USER;
93    p->tf->es = p->tf->ds;
94    p->tf->ss = p->tf->ds;
95    p->tf->eflags = FL_IF;
96    p->tf->esp = PGSIZE;
97    p->tf->eip = 0;  // beginning of initcode.S
98
99    safestrcpy(p->name, "initcode", sizeof(p->name));
100   p->cwd = namei("/");
101
102   p->state = RUNNABLE;
103 }
```

首先，userinit 调用 allocproc 函数来寻找是否还存在 UNUSED 状态的进程可以被用来分配，如果存在，allocproc 把进程状态切换为 EMBRO 并初始化进程，最后返回指向这个进程结构的指针，这个进程赋值给静态变量 initproc，作为所有进程的父进程。然后，userinit 调用 setupkvm 函数，为进程设置映射内核区的页表。这个函数在 vm.c 中，它调用 kalloc 为页表分配内存空间，然后用 memset 初始化这片内存，再对一些必要内容进行检查，没有错误则返回页表指针。

```
128  pde_t*
129  setupkvm(void)
130  {
131    pde_t *pgdir;
132    struct kmap *k;
133
134    if((pgdir = (pde_t*)kalloc()) == 0)
135      return 0;
136    memset(pgdir, 0, PGSIZE);
137    if (p2v(PHYSTOP) > (void*)DEVSPACE)
138      panic("PHYSTOP too high");
139    for(k = kmap; k < &kmap[NELEM(kmap)]; k++)
140      if(mappages(pgdir, k->virt, k->phys_end - k->phys_start,
141                  (uint)k->phys_start, k->perm) < 0)
142        return 0;
143    return pgdir;
144  }
```

分配并检查完页表指针后，userinit 函数调用 inituvm 函数，把 initcode.S 这段代码加载到内存起始地址。inituvm 为代码分配一页（PGSIZE）大小的内存空间，使用 memset 初始化，再使用 mappages 做内存映射、设置权限后，把代码内容移动到对应的内存空间中。_binary_initcode_start 和_binary_initcode_size 代表了这段二进制代码的起始地址和大小。

```
181  void
182  inituvm(pde_t *pgdir, char *init, uint sz)
183  {
184    char *mem;
185
186    if(sz >= PGSIZE)
187      panic("inituvm: more than a page");
188    mem = kalloc();
189    memset(mem, 0, PGSIZE);
190    mappages(pgdir, 0, PGSIZE, v2p(mem), PTE_W|PTE_U);
191    memmove(mem, init, sz);
192  }
```

接下来，函数为进程设置内存大小限制，PGSIZE 宏定义在 mmu.h 中，值为 4096。接然后函数初始化 trapframe 结构，并为一些段寄存器、栈指针、PC 值设置初始值。其中，%cs 保存段选择器，指向段 SEG_CODE 并设置特权级为 DPL_USER；%ds、%es、%ss 段选择器指向段 SEG_UDATA 并处于特权级 DPL_USER；%eflags 的 FL_IF 位被设置为允许硬件中断；栈指针%esp 被设置为进程的最大有效虚拟内存；PC 初值设置为 0，代表 initcode.S 的起始指令地址。为了便于调试，这个进程的名称被设置为 initcode。p->cwd 被设置为进程当前的工作目录。

所有的设置都完成以后，进程状态被设置为 RUNNABLE，表明这个进程可以准备运行了。

2.创建进程的一般方法：allocproc

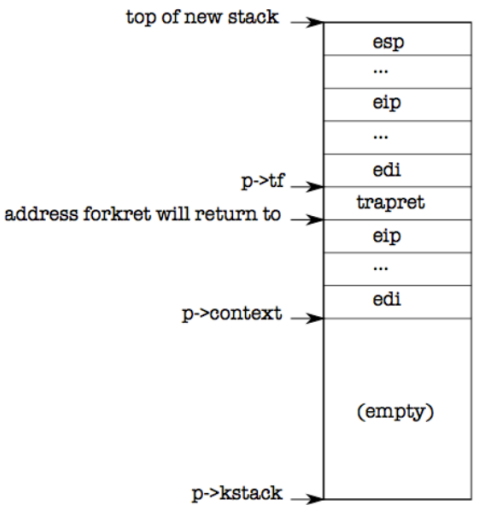
上面分析的 userinit 函数用于创建第一个进程，下面让我们来分析一下创建进程的一般方法，也就是 allocproc 函数。

```
34  static struct proc*
35  allocproc(void)
36  {
37    struct proc *p;
38    char *sp;
39
40    acquire(&ptable.lock);
41    for(p = ptable.proc; p < &ptable.proc[NPROC]; p++)
42      if(p->state == UNUSED)
43        goto found;
44    release(&ptable.lock);
45    return 0;
46
47  found:
48    p->state = EMBRYO;
49    p->pid = nextpid++;
50    release(&ptable.lock);
51
52    // Allocate kernel stack.
53    if((p->kstack = kalloc()) == 0){
54      p->state = UNUSED;
55      return 0;
56    }
57    sp = p->kstack + KSTACKSIZE;
58
59    // Leave room for trap frame.
60    sp -= sizeof *p->tf;
61    p->tf = (struct trapframe*)sp;
62
63    // Set up new context to start executing at forkret
64    // which returns to trapret.
65    sp -= 4;
66    *(uint*)sp = (uint)trapret;
67
68    sp -= sizeof *p->context;
69    p->context = (struct context*)sp;
70    memset(p->context, 0, sizeof *p->context);
71    p->context->eip = (uint)forkret;
72
73    return p;
74  }
```

首先，为了避免竞争问题，函数试图获取进程表的锁。获取成功后，函数在进程表中寻找一个未使用的进程结构，如果找到，进入进程初始化阶段，否则会释放锁并退出。

找到可分配的进程结构后，这个进程被设置为 EMBRYO 状态，表示正在初始化，然后为这个进程设置 PID。可以看到，PID 是依次递增的。

接下来，函数调用 kalloc 函数试图为进程分配内核栈空间，如果分配失败，则把进程状态重新设置为 UNUSED，并返回 0（相当于空指针）。如果分配成功，函数开始设置新进程的内核栈。sp 首先设置为栈顶地址，然后向低地址扩展 trap frame 大小，并以这个地址作为 trap frame 的起始地址，并赋值给 proc 的 tf 变量。然后，sp 指针继续向下扩展 4 字节，为返回地址保留空间。接下来的地址空间用于储存进程的 context 结构。函数把进程的 contet 指针赋值后，初始化这篇地址空间，并为 context 设置 PC 值。这个 PC 值指向 forkret，因此，新进程的内核线程会从 forkret 开始运行。这个函数进行一些初始化后，返回到当前栈底的地址，在这个栈布局下，这个地址对应的是 trapret 的值，因此接下来会执行 trapasm.S 中 trapret 标号下的代码，恢复用户寄存器。下图为内核栈空间的布局情况。可以看到，此时在栈顶端存放的便是中断异常处理时需要保存的全部内容。



3.第一个进程的运行

main 函数调用 userinit 后，会继续调用 mpmain，这个函数又会调用 scheduler 来运行进程。这个函数是每个 CPU 上的调度处理程序，它会找到一个 RUNNABLE 状态的进程，并将对应 cpu 的 proc 变量设置为这个进程的地址值，代表要运行这个进程。其具体内容如下：

```
57  void
58  scheduler(void)
59  {
60    struct proc *p;
61
62    for(;;){
63      // Enable interrupts on this processor.
64      sti();
65
66      // Loop over process table looking for process to run.
67      acquire(&ptable.lock);
68      for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
69        if(p->state != RUNNABLE)
70          continue;
71
72        // Switch to chosen process.  It is the process's job
73        // to release ptable.lock and then reacquire it
74        // before jumping back to us.
75        proc = p;
76        switchuvm(p);
77        p->state = RUNNING;
78        swtch(&cpu->scheduler, proc->context);
79        switchkvm();
80
81        // Process is done running for now.
82        // It should have changed its p->state before coming back.
83        proc = 0;
84      }
85      release(&ptable.lock);
86
87    }
88  }
```

选定进程后，函数调用 switchuvm 切换 TSS（任务状态段，指示硬件执行中断或系统调用）和 h/w 页表（用户的虚拟内存空间）至目标进程相符的内容，然后切换进程状态为 RUNNABLE，并调用 swtch 函数，切换进程上下文。swtch 函数是一段汇编代码，位于 swtch.S 中。它把调用者保存的寄存器（%ebp、%ebx、%esi、%edi）存储于栈中，把旧进程的栈切换为新进程的栈，然后加载新的调用者保存寄存器。汇编代码如下：

```
.globl swtch
swtch:
  movl 4(%esp), %eax
  movl 8(%esp), %edx

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

  # Switch stacks
  movl %esp, (%eax)
  movl %edx, %esp

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

接下来，scheduler 调用 switchkvm 切换内核的虚拟地址空间，这样，上下文切换完成，新的进程便可以准备运行了。

对于第一个进程，它会执行 initcode.S 中的代码。

```
#include "syscall.h"
#include "traps.h"


# exec(init, argv)
.globl start
start:
  pushl $argv
  pushl $init
  pushl $0  // where caller pc would be
  movl $SYS_exec, %eax
  int $T_SYSCALL

# for(;;) exit();
exit:
  movl $SYS_exit, %eax
  int $T_SYSCALL
  jmp exit

# char init[] = "/init\0";
init:
  .string "/init\0"

# char *argv[] = { init, 0 };
.p2align 2
argv:
  .long init
  .long 0
```

这个代码会触发 exec 系统调用，如果执行成功，它会运行 init.c 中的代码。init.c 中的 main 函数会在需要的情况下创建一个新的控制台设备文件，然后把它作为描述符 0，1，2 打开，接着它会不断循环，开启控制台 shell，处理没有父进程的僵尸进程，直到 shell 退出，然后它会不断循环，这样一个系统就创建完成了。

```c
char *argv[] = { "sh", 0 };

int
main(void)
{
  int pid, wpid;

  if(open("console", O_RDWR) < 0){
    mknod("console", 1, 1);
    open("console", O_RDWR);
  }
  dup(0);  // stdout
  dup(0);  // stderr

  for(;;){
    printf(1, "init: starting sh\n");
    pid = fork();
    if(pid < 0){
      printf(1, "init: fork failed\n");
      exit();
    }
    if(pid == 0){
      exec("sh", argv);
      printf(1, "init: exec sh failed\n");
      exit();
    }
    while((wpid=wait()) >= 0 && wpid != pid)
      printf(1, "zombie!\n");
  }
}
```

三、进程状态切换

从刚才进程创建和运行的过程，我们可以看到进程在不同状态之间的切换。XV6 中的进程一共有 6 个状态：

UNUESED：进程的初始状态

EMBRYO：表示进程创建初始时，正在被分配的状态

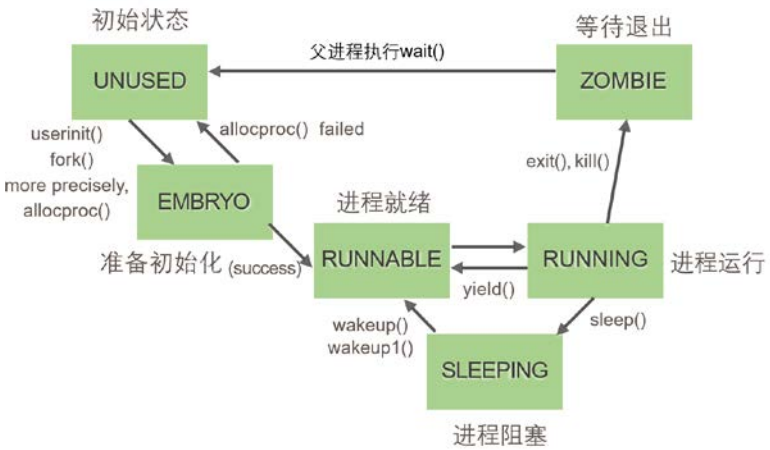
RUNNABLE：进程做好运行的准备，等待被调度

RUNNING：进程占用 CPU，准备执行

SLEEPING：进程遇到事件，被阻塞的状态

ZOMBIE：进程执行完毕后退出的状态

现在把它们之间的转换关系总结如下：



四、进程的操作

上面的转换图中还涉及到了进程的其他操作，下面我们来逐一分析一下。

1.fork

这个函数会创建一个新进程，它会复制父进程的进程内容。函数首先调用 allocproc 来创建进程 PCB，然后把父进程的页表和 trapframe 结构进行复制，并且把 parent 一项设为复制的进程，同时清空 trapframe 中的%eax，让子进程返回 0。然后，函数复制打开文件表和 cwd 结构，并为子进程设置状态为 RUNNABLE，复制进程名称，最后返回。对于父进程，会返回子进程的 pid，而由于子进程的 trapframe 结构中%eax 已经设置为 0，故子进程结束调用后会返回 0。

```
128  int
129  fork(void)
130  {
131    int i, pid;
132    struct proc *np;
133
134    // Allocate process.
135    if((np = allocproc()) == 0)
136      return -1;
137
138    // Copy process state from p.
139    if((np->pgdir = copyuvm(proc->pgdir, proc->sz)) == 0){
140      kfree(np->kstack);
141      np->kstack = 0;
142      np->state = UNUSED;
143      return -1;
144    }
145    np->sz = proc->sz;
146    np->parent = proc;
147    *np->tf = *proc->tf;
148
149    // Clear %eax so that fork returns 0 in the child.
150    np->tf->eax = 0;
151
152    for(i = 0; i < NOFILE; i++)
153      if(proc->ofile[i])
154        np->ofile[i] = filedup(proc->ofile[i]);
155    np->cwd = idup(proc->cwd);
156
157    pid = np->pid;
158    np->state = RUNNABLE;
159    safestrcpy(np->name, proc->name, sizeof(proc->name));
160    return pid;
161  }
```

2.exit

这个函数用于退出当前进程，不会返回。通过这个函数退出的进程会维持在 ZOMBIE 状态，直到它的父进程调用 wait 函数。在退出进程之前，首先要关闭打开文件表中所有的文件，并清空打开文件表的指针，同时清空 cwd 结构指针。然后进程会调用 wakeup1 函数来唤醒可能处于 wait 阻塞状态的父进程。对于当前进程的子进程，把它们的父进程设为 initproc。如果这些子进程中有 ZOMBIE 状态的进程，那么同时把 initproc 唤醒。最后，当前进程被设置为 ZOMBIE 状态，调用 sched 函数进行调度，不会再返回。

```
166  void
167  exit(void)
168  {
169    struct proc *p;
170    int fd;
171
172    if(proc == initproc)
173      panic("init exiting");
174
175    // Close all open files.
176    for(fd = 0; fd < NOFILE; fd++){
177      if(proc->ofile[fd]){
178        fileclose(proc->ofile[fd]);
179        proc->ofile[fd] = 0;
180      }
181    }
182
183    iput(proc->cwd);
184    proc->cwd = 0;
185
186    acquire(&ptable.lock);
187
188    // Parent might be sleeping in wait().
189    wakeup1(proc->parent);
190
191    // Pass abandoned children to init.
192    for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
193      if(p->parent == proc){
194        p->parent = initproc;
195        if(p->state == ZOMBIE)
196          wakeup1(initproc);
197      }
198    }
199
200    // Jump into the scheduler, never to return.
201    proc->state = ZOMBIE;
202    sched();
203    panic("zombie exit");
204  }
```

3.wait

这个函数用于等待当前进程的一个子进程退出，如果等待到，返回这个子进程的 pid，如果当前进程没有子进程，会返回-1。函数首先扫描进程表，寻找自己的僵尸子进程，如果找到，函数释放这个僵尸子进程的内核栈空间、页表，清空所有指针，并把这个进程的状态设置为 UNUSED，等待重新初始化，最后，返回这个进程的 pid。如果遍历进程表后，没找到子进程或者当前进程被杀死了，那么函数返回-1。如果上述条件都不满足，函数调用 sleep 把进程阻塞住，直到上述条件之一满足。

```
208  int
209  wait(void)
210  {
211    struct proc *p;
212    int havekids, pid;
213
214    acquire(&ptable.lock);
215    for(;;){
216      // Scan through table looking for zombie children.
217      havekids = 0;
218      for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
219        if(p->parent != proc)
220          continue;
221        havekids = 1;
222        if(p->state == ZOMBIE){
223          // Found one.
224          pid = p->pid;
225          kfree(p->kstack);
226          p->kstack = 0;
227          freevm(p->pgdir);
228          p->state = UNUSED;
229          p->pid = 0;
230          p->parent = 0;
231          p->name[0] = 0;
232          p->killed = 0;
233          release(&ptable.lock);
234          return pid;
235        }
236      }
237
238      // No point waiting if we don't have any children.
239      if(!havekids || proc->killed){
240        release(&ptable.lock);
241        return -1;
242      }
243
244      // Wait for children to exit.  (See wakeup1 call in proc_exit.)
245      sleep(proc, &ptable.lock);  //DOC: wait-sleep
246    }
247  }
```

4.yield

进程执行 yield 后，会把状态由 RUNNING 转换为 RUNNABLE，然后调用 sched 函数进行进程调度。这个函数允许进程主动释放 CPU，让位给其他进程。

```
311  void
312  yield(void)
313  {
314    acquire(&ptable.lock);  //DOC: yieldlock
315    proc->state = RUNNABLE;
316    sched();
317    release(&ptable.lock);
318  }
```

5.sleep

进程调用这个函数后，会把自己的状态转换为 SLEEPING 进入阻塞状态，并且设置 chan 变量等待被唤醒，然后调用调度函数切换到其他进程。唤醒后，清空 chan 变量，防止发生错误。

```
342  void
343  sleep(void *chan, struct spinlock *lk)
344  {
345    if(proc == 0)
346      panic("sleep");
347
348    if(lk == 0)
349      panic("sleep without lk");
350
351    // Must acquire ptable.lock in order to
352    // change p->state and then call sched.
353    // Once we hold ptable.lock, we can be
354    // guaranteed that we won't miss any wakeup
355    // (wakeup runs with ptable.lock locked),
356    // so it's okay to release lk.
357    if(lk != &ptable.lock){  //DOC: sleeplock0
358      acquire(&ptable.lock);  //DOC: sleeplock1
359      release(lk);
360    }
361
362    // Go to sleep.
363    proc->chan = chan;
364    proc->state = SLEEPING;
365    sched();
366
367    // Tidy up.
368    proc->chan = 0;
369
370    // Reacquire original lock.
371    if(lk != &ptable.lock){  //DOC: sleeplock2
372      release(&ptable.lock);
373      acquire(lk);
374    }
375  }
```

6.wakeup 与 wakeup1

这两个函数用于唤醒处于 chan 状态的进程。wakeup 调用 wakeup1，而 wakeup1 会遍历进程表，寻找 SLEEPING 状态且进程的 chan 变量与参数一致的进程，把状态转换为 RUNNABLE。

```
380  static void
381  wakeup1(void *chan)
382  {
383    struct proc *p;
384
385    for(p = ptable.proc; p < &ptable.proc[NPROC]; p++)
386      if(p->state == SLEEPING && p->chan == chan)
387        p->state = RUNNABLE;
388  }
```

```
391  void
392  wakeup(void *chan)
393  {
394    acquire(&ptable.lock);
395    wakeup1(chan);
396    release(&ptable.lock);
397  }
```

7.kill

这个函数会杀死进程号与传入参数相同的进程。实现方法时把进程的 killed 变量设置为 1，对于阻塞的进程，同时把它唤醒。当 killed 被设置为 1 后，从这个系统调用退出时，trap 函数会自动调用 exit()函数，终止进程。如果没找到进程号为 pid 的进程，那么返回-1；否则返回 0。

```
402  int
403  kill(int pid)
404  {
405    struct proc *p;
406
407    acquire(&ptable.lock);
408    for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
409      if(p->pid == pid){
410        p->killed = 1;
411        // Wake process from sleep if necessary.
412        if(p->state == SLEEPING)
413          p->state = RUNNABLE;
414        release(&ptable.lock);
415        return 0;
416      }
417    }
418    release(&ptable.lock);
419    return -1;
420  }
```

8.growproc

辅助函数，用于扩展当前进程的用户地址空间。成功返回 0，失败返回-1。

```
105  // Grow current process's memory by n bytes.
106  // Return 0 on success, -1 on failure.
107  int
108  growproc(int n)
109  {
110    uint sz;
111
112    sz = proc->sz;
113    if(n > 0){
114      if((sz = allocuvm(proc->pgdir, sz, sz + n)) == 0)
115        return -1;
116    } else if(n < 0){
117      if((sz = deallocuvm(proc->pgdir, sz, sz + n)) == 0)
118        return -1;
119    }
120    proc->sz = sz;
121    switchuvm(proc);
122    return 0;
123  }
```

9.procdump

用于调试，打印进程的信息到控制台上。用户输入 ctrl+P 时也会执行。

```
426  void
427  procdump(void)
428  {
429    static char *states[] = {
430    [UNUSED]    "unused",
431    [EMBRYO]    "embryo",
432    [SLEEPING]  "sleep ",
433    [RUNNABLE]  "runble",
434    [RUNNING]   "run   ",
435    [ZOMBIE]    "zombie"
436    };
437    int i;
438    struct proc *p;
439    char *state;
440    uint pc[10];
441
442    for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
443      if(p->state == UNUSED)
444        continue;
445      if(p->state >= 0 && p->state < NELEM(states) && states[p->state])
446        state = states[p->state];
447      else
448        state = "???";
449      cprintf("%d %s %s", p->pid, state, p->name);
450      if(p->state == SLEEPING){
451        getcallerpcs((uint*)p->context->ebp+2, pc);
452        for(i=0; i<10 && pc[i] != 0; i++)
453          cprintf(" %p", pc[i]);
454      }
455      cprintf("\n");
456    }
457  }
```

10.sched

进程调度相关函数。当前进程不是 RUNNING 状态时，会调用这个函数。函数检查当前进程的锁情况、进程状态、中断接受情况，然后调用 swtch 函数，把上下文切换到 cpu 对应的 scheduler 函数中，进行后续调度。

```
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;
}
```

五、小结

本次代码阅读调研了 XV6 的进程模型。XV6 系统启动时，进行初始化后，会首先调用 userinit 函数，创建第一个进程 initproc，并把它作为所有进程的父进程。创建完成后，会把它装载上 initcode.S 中的代码，进行后续的初始化，以完成整个系统的初始化。每个进程都有 6 种状态，从未分配，到分配，再到就绪、阻塞、运行，执行完毕后终止，每个过程都有对应的进程操作系统调用来实施。每个进程的信息都由 proc 这个结构来维护，修改进程状态等信息实际上就是对这个结构内的数据进行修改。每个 CPU 的状态由 cpu 这个数据结构来维护，每个 cpu 都维护着一个进程表，来管理挂载到这个 cpu 上的所有进程。XV6 中似乎没有涉及线程的概念，但每个进程都有内核态和用户态，相应地，每个进程也都有用户地址空间和内核地址空间。进程上下文切换由 swtch 汇编代码完成，主要是保存每个函数的调用者保存寄存器，进程调度由 sched 函数和 scheduler 函数控制，将在进程调度模型的代码阅读报告中进行进一步分析。