第四部分展示

王琮 陈修齐 胡睿贤 曹琬璐（报告人曹琬璐）

第四部分：

main/init.c->fork/usys.S->alltraps/trapasm.S->trap/trap.c->syscall/syscall.c->sys_fork/sysproc.c->for, wait, sleep/proc.c ->sched/proc.c->swtch/swtch.S

其中 alltraps, trap, syscall 简要介绍下执行流程即可。

重点关注的问题：

- 用户态程序 main/init.c 调用的 fork 的定义在哪里？和一般的函数定义有什么不同？
- 为什么子进程和父进程一样都会返回 main/init.c？
- 子进程是如何从 RUNNABLE 转换到 RUNNING 状态的？
- main/init.c 调用 fork 后，是父进程先返回还是子进程先返回？
- 对于父进程和子进程，fork 返回的 pid 相同么？为什么？
- 子进程返回后，加载的程序是什么程序？
- wait 系统调用的功能？

进阶题：

xv6 和 Linux 中调度器如何选择下一个要执行的进程？可选取一个 Linux 调度算法针对代码详细分析。

流程

1. 用户态fork定义：

在usys.S，用一行SYSCALL(fork)定义

```
SYSCALL(fork)

#这个定义会定义好的宏替换为下面的代码
.globl fork
fork:
movl $SYS_fork, %eax  #1，fork的系统调用号
int $T_SYSCALL        #64，系统调用对应的中断号
ret
```

int例外处理过程：

- 启用当前进程（父进程）内核栈（切换ss，esp），压参数（ss，esp，中断号等）
- 跳到trapasm，继续保存现场后转入trap：
- 通过中断号启动syscall：
- 通过系统调用号找到处理函数，选用sysfork，sysfork中调用了内核态fork函数，执行完毕后把返回值放入tf的eax

2. 内核态fork执行过程：

（创建子进程的过程

- 先调用allocproc：内核栈：tf空间：trapret；context：forkret（代码见下
- 从alloc返回后（一个拷贝父进程的过程：代码见下

  copy父进程：pgdir，tf，ofile[]，cwd（**tf除了eax之外都相同，context也相同，所以都会返回init运行**

  自己填充：np->tf->eax = 0;（返回值是0）state

  （而父进程的eax是fork函数的返回值，即子进程pid（就是这里实现了pid返回不一样，因为二者有不同的任务，接下来要进不同的分支

```
static struct proc* allocproc(void){
  struct proc *p;
  char *sp;
  acquire(&ptable.lock);

  for(p = ptable.proc; p < &ptable.proc[NPROC]; p++)     //遍历进程结构表找一个空位
    if(p->state == UNUSED)
      goto found;
  release(&ptable.lock);
  return 0;

found:
  p->state = EMBRYO;
  p->pid = nextpid++;    //分配pid

  release(&ptable.lock);
  if((p->kstack = kalloc()) == 0){   //分配内核栈
```

```
    p->state = UNUSED;
    return 0;
  }
  sp = p->kstack + KSTACKSIZE;  //内核栈顶

  // Leave room for trap frame.
  sp -= sizeof *p->tf;
  p->tf = (struct trapframe*)sp;   //记录tf结构的地址

  sp -= 4;
  *(uint*)sp = (uint)trapret;  //填trapret

  sp -= sizeof *p->context;    //分配context
  p->context = (struct context*)sp;  //记录context地址
  memset(p->context, 0, sizeof *p->context);//初始化置零
  p->context->eip = (uint)forkret;   //放forkret（fork返回

  return p;
}

int fork(void){
  int i, pid;
  struct proc *np;
  struct proc *curproc = myproc();  //cpu当前的进程

  if((np = allocproc()) == 0){
    return -1;
  }

// Copy父进程：

  //页表
  if((np->pgdir = copyuvm(curproc->pgdir, curproc->sz)) == 0){
    kfree(np->kstack);
    np->kstack = 0;
    np->state = UNUSED;
    return -1;
  }
  //sz, tf内容
  np->sz = curproc->sz;
  *np->tf = *curproc->tf;    //填入父进程的tf（在自己的位置
  //文件信息和cwd
  for(i = 0; i < NOFILE; i++)
    if(curproc->ofile[i])
      np->ofile[i] = filedup(curproc->ofile[i]);
  np->cwd = idup(curproc->cwd);
  //name
  safestrcpy(np->name, curproc->name, sizeof(curproc->name));
```

```
  //填写父进程、返回值、pid
  np->parent = curproc;
  np->tf->eax = 0;  //子进程返回值


  pid = np->pid;
  //修改状态
  acquire(&ptable.lock);
  np->state = RUNNABLE;
  release(&ptable.lock);
  return pid;
  //进程构建完毕，返回proc结构check
}
```

3. fork调用完毕：

一层层返回

然后sysfork调用返回，子进程pid作为返回值放入np->tf->eax，trap返回，trapret，iret（ip，cs，esp，标志等）——**然后又返回父进程中断前的现场**

```
void syscall(void){
  int num;
  struct proc *curproc = myproc();
  num = curproc->tf->eax;
  if(num > 0 && num < NELEM(syscalls) && syscalls[num]) {
    curproc->tf->eax = syscalls[num]();    //父进程返回值放入tf->eax
  }
}
```

4. 回到父进程现场：wait

fork执行结束后（代码

（有可能会被时钟中断切走

```
int main(void){
  ...

  for(;;)
  {
    printf(1, "init: starting sh\n");
    pid = fork();
    //返回
```

```
    if(pid < 0){
      printf(1, "init: fork failed\n");
      exit();
    }
    if(pid == 0){
      exec("sh", argv);
      printf(1, "init: exec sh failed\n");
      exit();
    }
    //父进程
    //不考虑时间片，父进程会先接着往下运行：wait回收子进程
    while((wpid=wait()) >= 0 && wpid != pid)
      //wait循环：回收子进程（见下
      //子进程回收完毕后，跳出这两句，继续下一个大循环
      printf(1, "zombie!\n");
  }
}

int wait(void){
  struct proc *p;
  int havekids, pid;
  struct proc *curproc = myproc();
  acquire(&ptable.lock);


//wait等待子进程退出，完成其退出后的扫尾工作（回收页表等），并返回其pid

  for(;;)  //循环遍历ptable
  {
    havekids = 0;
    for(p = ptable.proc; p < &ptable.proc[NPROC]; p++)
    {
      if(p->parent != curproc)   //寻找自己的子进程
        continue;
      havekids = 1;
      if(p->state == ZOMBIE){      //子进程已标记退出，则释放页表，清空proc结构，返回
        // Found one.
        pid = p->pid;
        kfree(p->kstack);
        p->kstack = 0;
        freevm(p->pgdir);
        p->pid = 0;
        p->parent = 0;
        p->name[0] = 0;
        p->killed = 0;
        p->state = UNUSED;
        release(&ptable.lock);
        return pid;           //返回退出的进程的pid
      }//继续遍历
```

```
    }
    //一次遍历结束，如果没有子进程了，则返回-1，退出
    if(!havekids || curproc->killed){
      release(&ptable.lock);
      return -1;
    }
    //否则就是有子进程，但是还没退出，就睡一会，醒了再来循环找
    // Wait for children to exit.  (See wakeup1 call in proc_exit.)
    sleep(curproc, &ptable.lock);  //DOC: wait-sleep
  }
}
```

5. 子进程加载：

- 某次调度的时候（scheduler函数），可能启动子进程

  （比如时间中断，进入trap，调用yield，通过swtch回到scheduler上下文，继续调度）
- 调度完毕，用子进程的信息恢复上下文

  因为上下文中保存的位置都是直接从父进程copy来的，所以子进程会从init中fork之后接着运行
- 返回值是0，进入0分支

  exec加载sh对应的可执行文件

```
void scheduler(void){
  struct proc *p;
  struct cpu *c = mycpu();
  c->proc = 0;

  for(;;){
    sti();
    acquire(&ptable.lock);
    for(p = ptable.proc; p < &ptable.proc[NPROC]; p++)
    {
      if(p->state != RUNNABLE)
        continue;
      c->proc = p;
      switchuvm(p);
      p->state = RUNNING;

      swtch(&(c->scheduler), p->context);
      //上一次调度，就是通过swtch函数切到用户进程的上下文
      //时钟中断中yield调用swtch，就又会切回到这里，接着向下走
      switchkvm(); //换内核页表
      c->proc = 0; //切走当前进程，然后回到内循环开始处，继续查ptable找一个runnable进程
    }
    release(&ptable.lock);
  }
}
```

```
//子进程加载上之后
int main(void){
  ...

  for(;;)
  {
    printf(1, "init: starting sh\n");
    pid = fork();
    //恢复现场
    if(pid < 0){
      printf(1, "init: fork failed\n");
      exit();
    }
    if(pid == 0){    //子进程从这里接着运行
      exec("sh", argv);    //加载sh可执行文件，运行
      printf(1, "init: exec sh failed\n");
      exit();
    }
    while((wpid=wait()) >= 0 && wpid != pid)
      printf(1, "zombie!\n");
  }
}

//exec:
//切换sh的页表和各种程序信息
//tf的eip放上sh的entry，中断返回后直接进入sh运行（不会再返回init中main函数

//shell：壳（相对内核），命令解释器
//shell接收用户输入，解析，处理（调用内核），反馈结果
```

gdb

断点：

- fork函数处，ni转进allocproc，跟踪pcb初始化过程
- fork停下后加上wait和exec处的断点，继续运行可以看到先进了wait，即父进程，然后是带着sh参数exec，即子进程

图1. 内核栈：

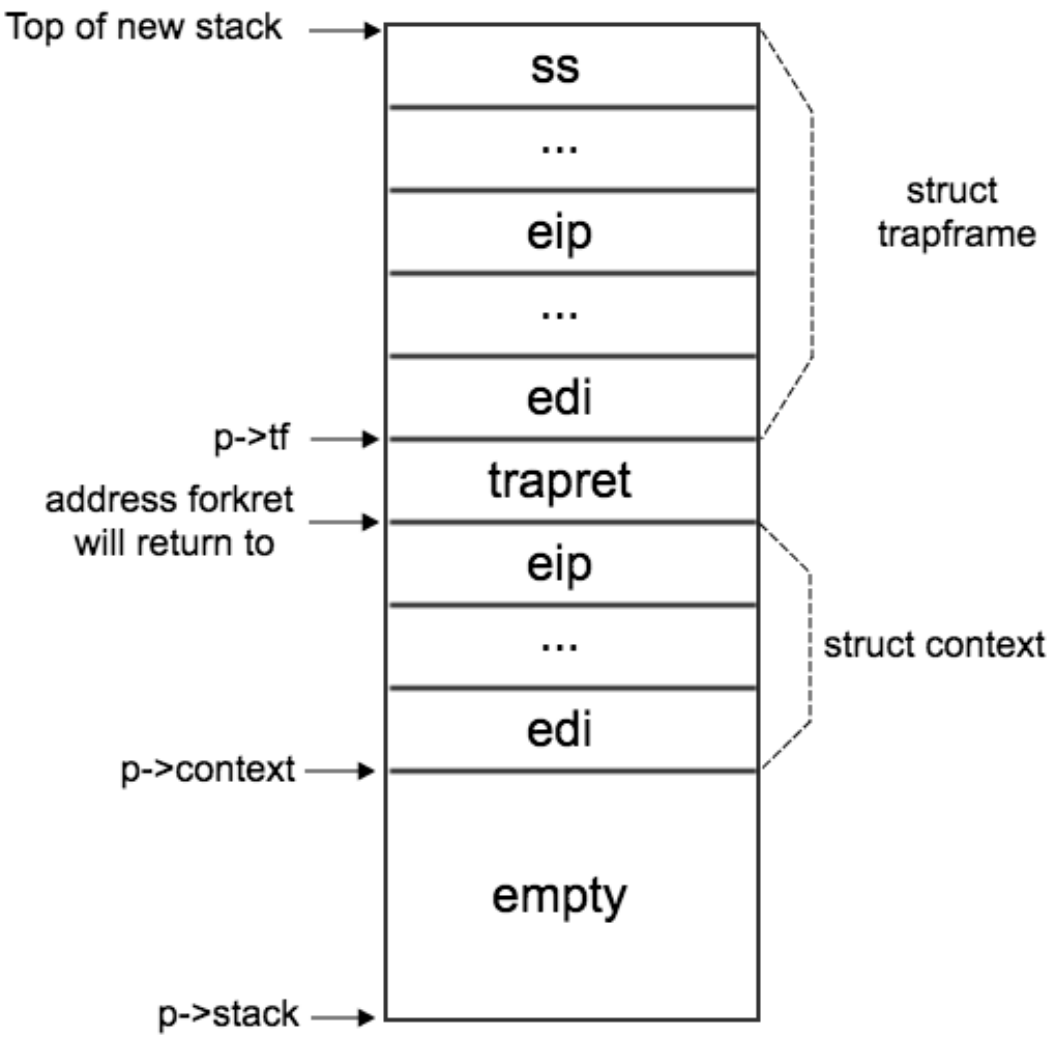
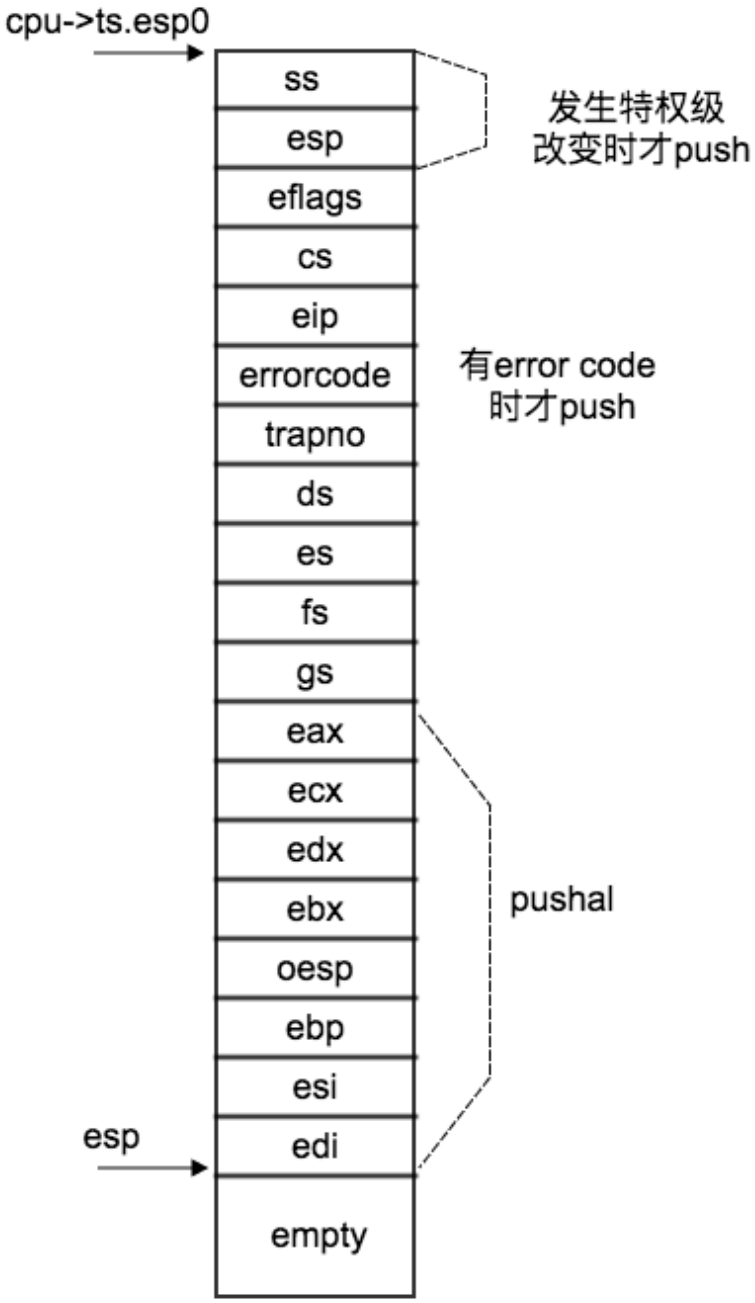


图2. trap frame



示例代码：context和proc

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

struct proc {
  uint sz;                     // Size of process memory (bytes)
```

```
  pde_t* pgdir;                // Page table
  char *kstack;                // a
  enum procstate state;        // a
  int pid;                     // a
  struct proc *parent;         // Parent process
  struct trapframe *tf;        // a
  struct context *context;     // a
  void *chan;                  // If non-zero, sleeping on chan
  int killed;                  // If non-zero, have been killed
  struct file *ofile[NOFILE];  // Open files
  struct inode *cwd;           // Current directory
  char name[16];               // Process name (debugging)
};
```