

## 1. 进程控制块

在之前我们没有提进程的概念，都是以任务的概念来说的，从这章开始正式进入进程，在内核中一个TaskControlBlock即代表了一个进程，系统中同一时间存在的每个进程都被一个不同的 **进程标识符** (PID, Process Identifier) 所标识。在内核初始化完毕之后会创建一个进程——即 **用户初始进程** (Initial Process)，它是目前在内核中以硬编码方式创建的唯一一个进程。其他所有的进程都是通过一个名为 **fork** 的系统调用来创建的。

因此我们首先更新一下TCB的内容：

C

```
typedef struct TaskControlBlock
{
    TaskState task_state;      //任务状态
    int pid;                   // Process ID
    struct TaskControlBlock* parent; //Parent
    process
    TaskContext task_context;  //任务上下文
    u64 trap_cx_ppn;          //Trap 上下文所在物理地址
    u64 base_size;            //应用数据大小
    u64 kstack;                //应用内核栈的虚拟地址
    u64 ustack;                //应用用户栈的虚拟地址
    u64 entry;                 //应用程序入口地址
    PageTable pagetable;       //应用页表所在物理页
}TaskControlBlock;
```

新增了pid以及父进程的TCB指针，每个进程都需要为其分配一个pid的值，我将初始的第一个进程的pid值设置成了1，当有新的进程被创建时，相应的pid值加一就行，代码实现如下：

```

int nextpid = 1;
int allocpid()
{
    int pid;
    pid = nextpid;
    nextpid =
nextpid + 1;
    return pid;
}

```

如果使用`app_init`函数进行进程初始化的话就去调用此函数为进程分配一个`pid`号：

```

void app_init(size_t app_id)
{
    TrapContext* cx_ptr = tasks[app_id].trap_cx_ppn;
    reg_t sstatus = r_sstatus();
    // 设置 sstatus 寄存器第8位即SPP位为0 表示为U模式
    sstatus &= (0U << 8);
    w_sstatus(sstatus);
    // 设置程序入口地址
    cx_ptr->sepc = tasks[app_id].entry;
    //
    cx_ptr->sstatus = sstatus;
    // 设置用户栈虚拟地址
    cx_ptr->sp = tasks[app_id].ustack;
    // 设置内核页表token
    cx_ptr->kernel_satp = kernel_satp;
    // 设置内核栈虚拟地址
    cx_ptr->kernel_sp = tasks[app_id].kstack;
    // 设置内核trap_handler的地址
    cx_ptr->trap_handler = (u64)trap_handler;

    /* 构造每个任务任务控制块中的任务上下文，设置 ra 寄存器为 trap_return 的入口地址*/
    tasks[app_id].task_context = tcx_init((reg_t)tasks[app_id].kstack);
    // 初始化 TaskStatus 字段为 Ready
    tasks[app_id].task_state = Ready;
    /* 分配pid值 */
    tasks[app_id].pid = allocpid();
}

```

顺便这里修复一个bug，在调用`tcx_init`函数初始化任务上下文时，传入的参数应该是该进程的内核栈的地址才对，用于更新`sp`之前传入错了。

## 2. fork的实现

`fork`函数的功能直观体现如下：

```

int main()
{
    // 在父进程中创建子进程
    pid_t pid = fork();
    printf("当前进程fork()的返回值: %d\n", pid);
    if(pid > 0)
    {
        // 父进程执行的逻辑
        printf("我是父进程, pid = %d\n", getpid());
    }
    else if(pid == 0)
    {
        // 子进程执行的逻辑
        printf("我是子进程, pid = %d, 我爹是: %d\n", getpid(),
getppid());
    }
    else // pid == -1
    {
        // 创建子进程失败了
    }

    // 不加判断, 父子进程都会执行这个循环
    for(int i=0; i<5; ++i)
    {
        printf("%d\n", i);
    }

    return 0;
}

```

父进程在调用fork函数后会生成一个新的子进程，此子进程和父进程的代码一样，执行逻辑是不一样的，当父进程fork成功后，fork返回的是子进程的pid号，而子进程中返回的是0代表子进程创建成功

在实现 fork 的时候，分为两个步骤：

- 新建一个进程，为其分配页表，分配trap页，映射跳板页
- 拷贝父进程的数据和子进程的一致

父进程和子进程的虚拟地址空间是完全相同的，但是由于页表不一样，所以映射到的实际物理内存是不一样的。

首先来实现第一步，为子进程创建页表，分配与映射trap页，映射跳板页：

C

```
struct TaskControlBlock* allocproc()
{
    struct TaskControlBlock* p;
    for(p = tasks; p < &tasks[MAX_TASKS]; p++)
    {
        if(p->task_state == UnInit)
        {
            goto found;
        }
    }
    return 0;

found:
    p->pid = allocpid();
    p->task_state = Ready;
    // 为每个应用程序分配一页内存用与存放trap, 同时初始化任务
    上下文
    proc_trap(p);
    // 为用户程序创建页表, 映射跳板页和trap上下文页
    proc_pagetable(p);
    return p;
}
```

- 去进程数组里查找未初始化的进程, 找到了则跳转到found处
- 为此进程分配pid, 将其任务状态设置成Ready, 这样就可以进入调度了
- 调用proc\_trap函数, 为应用程序分配一页内存用与存放trap, 同时初始化任务上下文
- 调用proc\_pagetable函数, 为用户程序创建页表, 映射跳板页和trap上下文页

在此之前, 实现了一个内核支持进程的初始化函数, 将所有进程状态设置成了UnInit

C

```
void procinit()
{
    struct TaskControlBlock *p;
    for(p = tasks; p <
&tasks[MAX_TASKS]; p++)
    {
        p->task_state = UnInit;
    }
}
```

这样当创建新的进程时就去tasks数组里查找未初始化的。

然后来实现第二步，为子进程创建一个和父进程几乎完全相同的应用地址空间，定义了 `uvmcopy` 函数，此函数的入参为父进程的根页表和子进程的根页表和一个 `sz` 的参数，我们对父进程的页表进行索引，去查找父进程的虚拟地址空间中那些页是映射了的，如果此虚拟地址被映射了，则为子进程创建相同的映射，不同的是需要为子进程分配进行映射的物理地址页。

C

```
int uvmcopy(PageTable* old, PageTable* new, u64 sz)
{
    PageTableEntry* pte;
    u64 pa, i;
    u8 flags;

    for (i = 0; i < sz; i+=PAGE_SIZE)
    {
        VirtPageNum vpn = floor_virts(virt_addr_from_size_t(i));
        pte = find_pte(old,vpn);

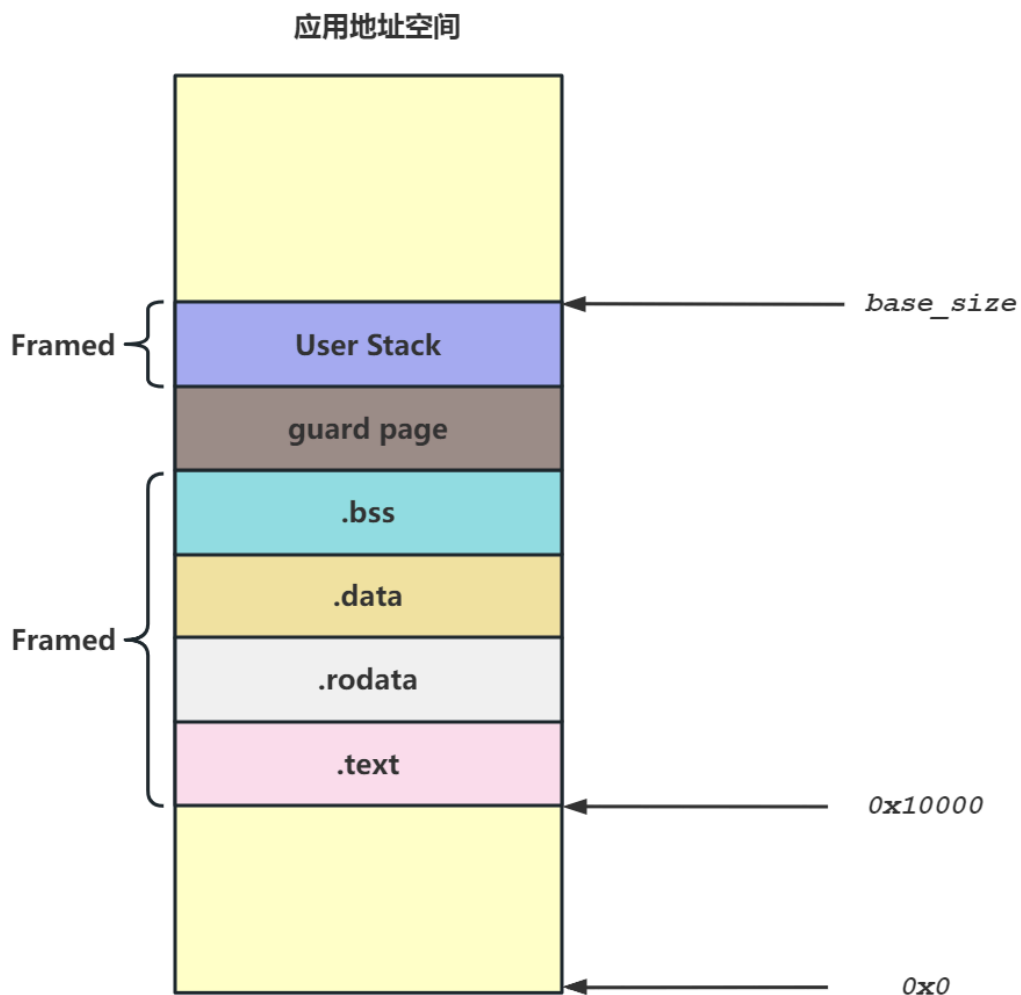
        if (pte != 0)
        {
            /* 将PTE 转换为物理地址*/
            u64 phyaddr = PTE2PA(pte->bits);
            /* 得到PTE的映射 flags */
            flags = PTE_FLAGS(pte->bits);
            /* 分配一页内存 */
            PhysPageNum ppn = kalloc();
            u64 paddr = phys_addr_from_phys_page_num(ppn).value;
            /* 拷贝内存 */
            memcpy((void*)paddr,(void*)phyaddr,PAGE_SIZE);

            /* 映射内存 */
            PageTable_map(new,virt_addr_from_size_t(i), \
                phys_addr_from_size_t(paddr),PAGE_SIZE,flags);
        }
    }
}
```

这个 `sz` 的值是父进程所占的虚拟地址空间的最大值，是在 `load_app` 函数中进行赋值的：

```
// 映射应用程序用户栈开始地址
proc->ustack = 2 * PAGE_SIZE + PGROUNDUP(proc->ustack);
PhysPageNum ppn = kalloc();
u64 paddr = phys_addr_from_phys_page_num(ppn).value;
PageTable_map(&proc->pagetable,virt_addr_from_size_t(proc->ustack - PAGE_SIZE),phys_addr_from_s
    PAGE_SIZE, PTE_R | PTE_W | PTE_U);
proc->base_size=proc->ustack;
```

如下图：



我们从`0x0`开始依次遍历，通过`find_pte(old, vpn);`函数去一页一页遍历直到遍历到`base_size`大小，如果此页未被映射，则`find_pte(old, vpn);`会返回0，反之如果此页被映射了，则会返回映射的`pte`的值，我们此时就能对子进程进行映射了，同时根据父进程的`pte`找到存储父进程应用数据的物理页，拷贝数据到子进程的物理页中

这里`find_pte`函数有个小bug:

```

PageTableEntry* find_pte(PageTable* pt, VirtPageNum vpn)
{
    // 拿到虚拟页号的三级索引，保存到idx数组中
    size_t idx[3];
    indexes(vpn, idx);
    //根节点
    PhysPageNum ppn = pt->root_ppn;
    //从根节点开始遍历，如果没有pte，就返回空
    for (int i = 0; i < 3; i++)
    {
        //拿到具体的页表项
        PageTableEntry* pte = &get_pte_array(ppn)[idx[i]];
        //如果此项页表为空
        if (!PageTableEntry_is_valid(pte)) {
            return NULL;
        }
        if (i == 2) {
            return pte;
        }
        //取出进入下级页表的物理页号
        ppn = PageTableEntry_ppn(pte);
    }
}

```

将判断页表为空的这一步提到了返回pte的前面。

有了上面这两部我们再来看\_\_sys\_fork()的实现：

- 父进程就是当前正在执行的进程，通过current\_proc()函数拿到了当前进程的PCB指针
- 创建一个新的子进程
- 拷贝父进程的内存数据，创建一个和父进程相同的虚拟地址空间
- 拷贝父进程的trap页的数据
- 将子进程的trap返回值设置为0，然后复制父进程的TCB的信息
- 设置子进程的返回地址和内核栈
- 将\_top++
- \_\_sys\_fork()的返回值为子进程的pid号

```

int __sys_fork()
{
    struct TaskControlBlock* np;
    struct TaskControlBlock* p = current_proc();
    // Allocate process.
    if((np = allocproc()) == 0){
        return -1;
    }

    // 拷贝父进程的内存数据，根据页表查找物理页拷贝
    uvmcopy(&p->pagetable,&np->pagetable,p->base_size);

    // 拷贝父进程的trap页数据
    memcpy((void*)np->trap_cx_ppn,(void*)p->trap_cx_ppn,PAGE_SIZE);

    // 子进程返回值为0
    TrapContext* cx_ptr = np->trap_cx_ppn;
    cx_ptr->a0 = 0;
    // 复制TCB的信息
    np->entry = p->entry;
    np->base_size = p->base_size;
    np->parent = p;
    np->ustack = p->ustack;
    // 设置子进程返回地址和内核栈
    np->task_context.ra = trap_return;
    np->task_context.sp = np->kstack;

    _top++;
    return np->pid;
}

```

我们在子进程内核栈上压入一个初始化的任务上下文，使得内核一旦通过任务切换到该进程，就会跳转到 `trap_return` 来进入用户态。而在复制地址空间的时候，子进程的 Trap 上下文也是完全从父进程复制过来的，这可以保证子进程进入用户态和其父进程回到用户态的那一瞬间 CPU 的状态是完全相同的。而两个进程的应用数据由于地址空间复制的原因也是完全相同的，

我们再来看看从父进程trap进内核执行fork的过程：

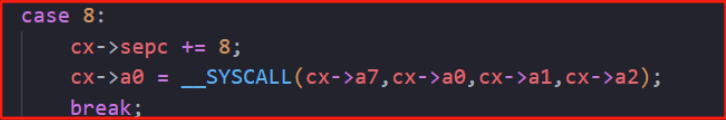
首先父进程执行fork函数进入内核的trap\_handler函数进行分发，这里将cx->sepc += 8;向前移动到了sys\_call的上一步，这也是一个小bug。因为在\_\_sys\_fork()内部会对父进程的trap页数据进行拷贝，为了保证子进程从内核返回后能正确返回到调用fork函数的下一指令开始执行，所以需要先修改trap页数据



```

switch (cause_code)
{
/* U模式下的syscall */
case 8:
    cx->sepc += 8;
    cx->a0 = __SYSCALL(cx->a7, cx->a0, cx->a1, cx->a2);
    break;
default:
    printk("undefined exception scause:%x\n", scause);
}

```



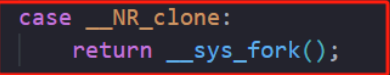
然后执行`__sys_fork()`函数，此函数执行完毕后，内核中就多了一个子进程，父进程就从`trap`返回了，此时返回到用户态的值就是`__sys_fork()`的返回值即子进程的`pid`值

```

uint64_t __SYSCALL(size_t syscall_id, reg_t arg1, reg_t arg2, reg_t arg3) {
    switch (syscall_id)
    {

        case __NR_write:
            __sys_write(arg1, arg2, arg3);
            break;
        case __NR_read:
            __sys_read(arg1, arg2, arg3);
        case __NR_sched_yield:
            __sys_yield();
            break;
        case __NR_gettimeofday:
            return __sys_gettime();
        case __NR_clone:
            return __sys_fork();
        default:
            printk("Unsupported syscall id:%d\n", syscall_id);
            break;
    }
}

```



然后进行调度，当调度到子进程时会从内核态返回到用户态，由于在`__sys_fork()`函数中我们将子进程的`trap`页的`a0`寄存器的值设置成了0，所以用户态接收到的值就是0

### 3. 测试

首先修改一下`main`函数，初始化所有进程

```
void os_main()
{
    printk("hello timer os!\n");

    // 内存分配器初始化
    frame_allocator_init();
    //初始化内存
    kvminit();
    //初始化进程
    procinit();
    //加载进程
    load_app(0);
    app_init(0);
    load_app(1);
    app_init(1);
    //映射内核
    kvminithart();
    //trap初始化
    set_kernel_trap_entry();

    get_app_names();

    //初始化时钟
    timer_init();

    run_first_task();
}
```

然后修改应用程序，我直接修改time.c

C

```
#include
<timeros/types.h>
#include
<timeros/syscall.h>
#include
<timeros/string.h>
int main()
{
    // 在父进程中创建子进
    程
    int pid = sys_fork();
    while (1)
    {
        if(pid > 0)
        {
            // 父进程执行
            的逻辑

            printf("father\n");
        }
        else if(pid == 0)
        {
            // 子进程执行
            的逻辑

            printf("child\n");
        }
        else // pid == -1
        {
            // 创建子进程
            失败了
        }
    }
    return 0;
}
```

在app.c中新建一个系统调用函数, #define \_\_NR\_clone 220

C

```
int sys_fork()
{
    return
    syscall(__NR_clone,0,0,0);
}
```

编译运行, 先编译time应用程序, 再编译内核:



可以看见来回打印`father`和`child`，测试成功

## 参考链接

---

- 进程管理机制的设计实现 - rCore-Tutorial-Book-v3 3.6.0-alpha.1 文档 (rcore-os.cn)
- xv6 学习：进程管理B fork&exec - 知乎 (zhihu.com)
- [xv6 fork的实现 | Blurred code](<https://www.blurredcode.com/2020/11/xv6fork的实现/#:~:text=fork的实现> 在xv6中的fork的实现是 `int fork(void) { int child_pid %3D,0 for child_process fork () return child_pid%3B }`)

文章作者: Timer

文章链接: <https://yanglianoo.github.io/2023/10/08/sys-fork的实现/>

版权声明: 本博客所有文章除特别声明外, 均采用 CC BY-NC-SA 4.0 许可协议。转载请注明来自 TimerのBlog!

相关推荐