2024 操作系统 Lab4 串讲

2024 操作系统助教组

系统调用过程

系统调用接口

用户程序通过调用不同的 `syscall_*` 函数（定义在 `user/lib/syscall_lib.c` 中）请求内核完成相应的系统级操作。

但是 `syscall_*` 并非内核提供给用户程序的接口。真正的系统调用接口为 `msyscall` 函数。而 `syscall_*` 函数只是对 `msyscall` 的封装。

```
extern int msyscall(int, ...);
```

`msyscall` 函数以汇编形式编写，定义在 `user/lib/syscall_wrap.S` 中。其整体流程是：执行 `syscall` 指令使 CPU **陷入内核态**完成系统调用；在返回用户态继续运行此函数时，执行 `jr ra` 指令返回到 `syscall_*` 函数。

可以认为陷入就是一种异常，只不过陷入异常是由程序自行触发的。

系统调用过程

msyscall 的传参

```
    return msyscall(SYS_mem_map, srcid, srcva, dstid, dstva, perm);
```

内核在响应用户的系统调用请求时，需要获取系统调用的参数，例如系统调用号、用户进程 ID 等。观察 `syscall_*` 函数，我们将实现系统调用所需的参数作为 `msyscall` 的传入参数，`syscall_*` 在调用 `msyscall` 时，会自动将参数存入寄存器和用户栈，内核后续可以访问到这些参数，完成相应的系统调用。

```
    40134c: 8fc20030    lw  v0,48(s8)
    401350: 00000000    nop
    401354: afa20014    sw  v0,20(sp)  // 向用户栈指针($sp)+20的位置存入第6个参数
    401358: 8fc2002c    lw  v0,44(s8)
    40135c: 00000000    nop
    401360: afa20010    sw  v0,16(sp)  // 向用户栈指针($sp)+16的位置存入第5个参数
    401364: 8fc70028    lw  a3,40(s8)  // 向a3寄存器存入第4个参数
    401368: 8fc60024    lw  a2,36(s8)  // 向a2寄存器存入第3个参数
    40136c: 8fc50020    lw  a1,32(s8)  // 向a1寄存器存入第2个参数
    401370: 24040007    li  a0,7       // 向a0寄存器存入第1个参数
    401374: 0c10057c    jal 4015f0 <msyscall>
    401378: 00000000    nop
```

系统调用过程

异常入口与分发

用户态下执行 `syscall` 指令会触发异常，CPU 陷入内核态，从 `0x80000180`（即 `.exc_gen_entry` 所在位置）开始取指令。内核使用 `SAVE_ALL` 宏将用户进程的上下文运行环境保存在内核栈中，取出 `CP0_CAUSE` 寄存器中的异常码，系统调用对应的异常码为 `8`，以异常码作为索引在 `exception_handlers` 数组中找到对应的异常处理函数 `handle_sys`，转跳至 `handle_sys` 函数处理用户的系统调用请求。

`handle_sys` 函数的作用是处理用户的系统调用请求，其核心逻辑位于 `do_syscall` 函数。`handle_sys` 通过调用 `do_syscall` 实现处理系统调用。当内核完成处理系统调用请求，从 `do_syscall` 返回 `handle_sys` 后，`handle_sys` 调用 `ret_from_exception` 从内核态返回用户程序。

系统调用过程

内核获取用户传递的参数

`do_syscall` 函数在处理系统调用请求时需要获得由用户进程传递的参数。

根据 MIPS 调用规范，用户进程在调用 `msyscall` 时，已将这些参数存入 `a0` ， `a1` ， `a2` ， `a3` 寄存器和用户栈中。但内核处理系统调用请求时，CPU 处于内核态，通用寄存器等现场环境发生了变化，堆栈指针也已指向了内核栈。所以为了让内核获得这些参数，我们需要借助在内核栈中保存的用户进程上下文环境。

观察 `SAVE_ALL` ，在保存用户态现场时 `sp` 减去了一个 `Trapframe` 结构体的空间大小，此时我们将用户进程现场保存在内核栈中范围为 `[sp, sp + sizeof(TrapFrame))` 的这一空间范围内。当进入 `handle_sys` 函数时， `sp` 寄存器中保存的是 `Trapframe` 结构体的起始地址，将该起始地址存入 `a0` 寄存器作为 `do_syscall` 的传入参数，使得 `do_syscall` 函数可以访问到保存了用户进程现场的 `Trapframe` 结构体。

```
.macro SAVE_ALL
    // ...
    li      sp, KSTACKTOP
1:
    subu    sp, sp, TF_SIZE
    // ...
```

系统调用过程

do_syscall 函数

`do_syscall` 函数的主要流程如下：

- 从内核栈保存的用户进程现场中取出 `a0` 寄存器的值，即系统调用编号。以该编号为索引在 `syscall_table` 列表中找到对应的系统调用内核函数指针 `func`。

```
void *syscall_table[MAX_SYSNO] = {
    [SYS_putchar] = sys_putchar,
    [SYS_print_cons] = sys_print_cons,
    // ...
    [SYS_write_dev] = sys_write_dev,
    [SYS_read_dev] = sys_read_dev,
};
```

- 将内核栈中的 `epc` 寄存器的地址 +4。内核完成系统调用返回用户态时，将 `pc` 寄存器恢复到 `epc` 寄存器中设置的地址。这里 +4 的目的是使恢复到用户态后执行 `syscall` 指令的下一条指令，即 `jr ra`。

系统调用过程

do_syscall 函数

- 从 `Trapframe` 结构体中获取用户进程传递的第 2-6 个参数并赋给 `arg1`，`arg2` ，`arg3`，`arg4` ，`arg5`。

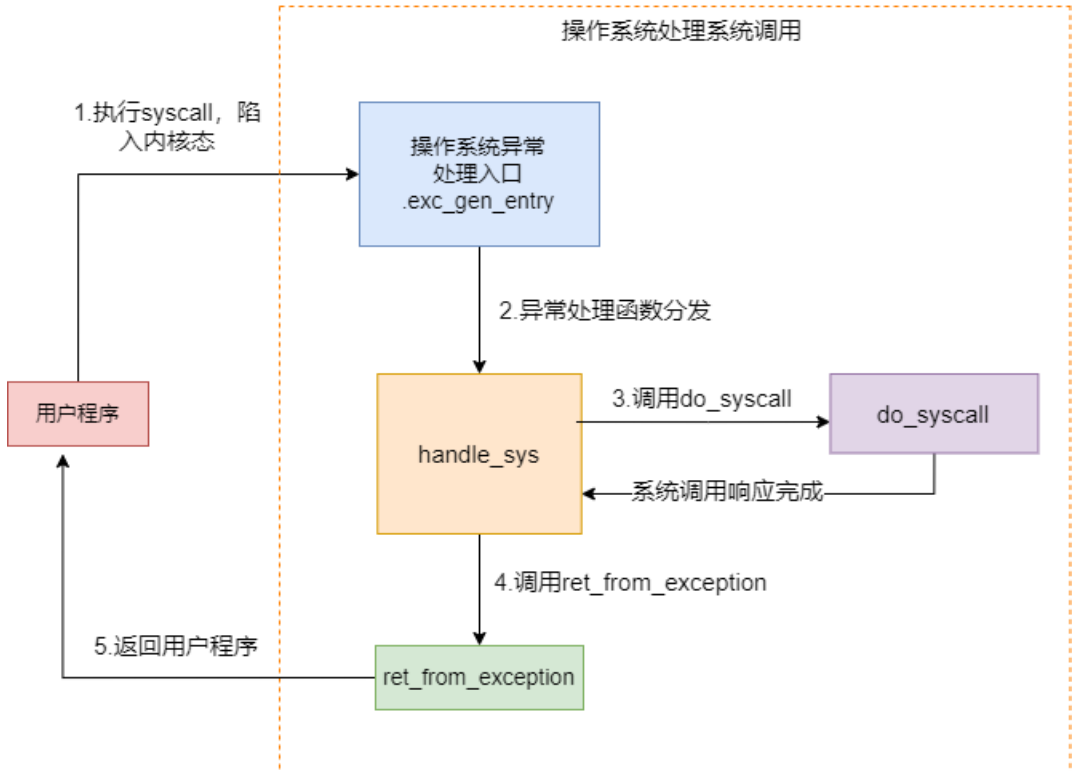
```
u_int arg1 = tf->regs[5];
u_int arg2 = tf->regs[6];
u_int arg3 = tf->regs[7];

u_int arg4, arg5;
// ...
```

- 通过函数指针 `func` 直接调用内核中相应的系统调用函数（即在 `kern/syscall_all.c` 中定义的 `sys_*` 函数）。由于内核中的实现函数都带有返回值，我们需要根据 MIPS 的调用规范，将调用 `func` 的返回值写入 `Trapframe` 结构体中保存的 `v0` 寄存器，使得在恢复用户进程上下文环境后，用户程序可以从该寄存器读取 `msyscall` 的返回值。

```
tf->regs[2] = func(arg1, arg2, arg3, arg4, arg5);
```

系统调用过程


操作系统处理系统调用
1.执行syscall，陷入内核态
操作系统异常处理入口
.exc_gen_entry
2.异常处理函数分发
用户程序
handle_sys
3.调用do_syscall
do_syscall
系统调用响应完成
4.调用ret_from_exception
5.返回用户程序
ret_from_exception

系统调用功能实现

envid2env 函数

`envid2env` 函数的作用是根据进程号 `envid` 返回对应的进程控制块。该函数在许多系统调用的实现中都有使用。

```
int envid2env(u_int envid, struct Env **penv, int checkperm);
```

重点关注：

- 需要从进程控制块数组 `envs` 中取出对应的进程控制块。使用 `ENVX` 将 `envid` 转化为下标。
- 参数 `checkperm` 表示在获得 `envid` 对应的进程控制块时是否对当前进程的权限进行验证。当 `checkperm` 为真时需要验证权限，若当前进程不能对 `envid` 表示的进程进行操作，则需要返回错误提示。

系统调用功能实现

系统调用：内存申请

`sys_mem_alloc` 函数的主要功能是为 `envid` 所对应的进程的虚拟地址空间 `va` 分配实际的物理页。

```
int sys_mem_alloc(u_int envid, u_int va, u_int perm);
```

- 其中，我们需要使用 `is_illegal_va` 函数判断 `va` 是否位于合法的用户地址空间中。
- 最后，使用 `page_alloc` 申请一个物理页，并使用 `page_insert` 建立进程 + 虚拟地址与物理页之间的映射关系。

系统调用功能实现

系统调用：内存映射

- `sys_mem_map` 函数用于将源进程虚拟地址空间中的相应内存映射到目标进程的相应虚拟地址空间的相应虚拟内存。实现的方式是将这两块地址映射到同一物理页面。

```
int sys_mem_map(u_int srcid, u_int srcva, u_int dstid, u_int dstva, u_int perm);
```

  需要将 `sys_mem_map` 和 `sys_mem_alloc` 区别开来。`sys_mem_map` 中要映射的物理页面已经存在，而 `sys_mem_alloc` 则需要额外申请。

- `sys_mem_unmap` 是 `sys_mem_map` 的逆过程，用于解除某个进程地址空间虚拟内存和物理内存之间的映射关系。

```
int sys_mem_unmap(u_int envid, u_int va);
```

系统调用内核实现

系统调用：进程让出

`sys_yield` 函数用于让用户进程主动让出执行权。调用该系统调用将会切换到其他进程，因此该系统调用是**无返回的**。我们用 `__attribute__((noreturn))` 指明这一点。

```
void __attribute__((noreturn)) sys_yield(void);
```

为实现让出执行权的功能，需要用到 Lab3 中实现的 `schedule` 函数，同时设定 `yield` 参数为真。

进程间通信机制

基本概念

进程间通信即不同进程间的数据传递。实现数据传递的方式有两种：

- 一是**借助内核空间**。内核空间并不区分不同进程，因而可以作为中介存放数据。发送方进程使用系统调用将传递的数据存放在进程控制块中，接收方进程同样使用系统调用在进程控制块中找到对应的数据，读取并返回。但是这种方法只能传递少量数据。

- 二是**共享内存**。我们知道，进程是资源分配的最小单位，各进程的虚拟地址空间相互隔离。但这不过是操作系统对上向用户进程提供的假象，如果我们选择让不同进程的两块虚拟地址空间映射到同一个物理页面，则通过存取这两块虚拟地址空间，就实现了不同进程间的通信。

进程间通信机制

进程控制块中 IPC 相关属性

```
struct Env {
    // lab 4 IPC
    u_int env_ipc_value; // data value sent to us
    u_int env_ipc_from; // envid of the sender
    u_int env_ipc_recving; // env is blocked receiving
    u_int env_ipc_dstva; // va at which to map received page
    u_int env_ipc_perm; // perm of page mapping received
};
```

- `env_ipc_value` 表示进程传递的数值。当进程间需要传递一个数值时，由发送进程通过系统调用将这个值写入接收进程控制块的env_ipc_value，接收进程访问自身控制块即可得到该值。
- `env_ipc_from` 表示发送方的进程 ID
- `env_ipc_recving` 表示进程的接收状态。其值为0或1。值为1时，表示等待接受数据中；值为0时，表示不可接受数据。
- `env_ipc_dstva` 表示进程接收到的物理页面需要与自身的哪个虚拟页面完成映射
- `env_ipc_perm` 表示传递的物理页面的权限位设置

进程间通信机制

系统调用：接收消息

```
int sys_ipc_recv(u_int dstva);
```

`sys_ipc_recv` 用于接收其他进程发送的数据。其基本流程如下：

- 如果进程需要通过共享内存接受数据（即 `dstva` 不为 0），则需检查接收虚拟地址 `dstva` 是否合法。
- 将当前进程控制块 `curenv` 中的接收状态 `env_ipc_recving` 设为真，表明当前进程准备接受发送方的消息。
- 将当前进程控制块属性 `env_ipc_dstva` 设为参数 `dstva` 的值，表明需要将物理页面映射到 `dstva` 处。
- 当前进程放弃执行，**进入阻塞状态**，等待发送进程传递的内容。操作系统调度其余进程运行。

需要注意，调用该系统调用后当前进程将不再处于运行状态，只有另一进程向本进程发送数据后本进程才会恢复运行。

进程间通信机制

系统调用：发送消息

```
int sys_ipc_try_send(u_int envid, u_int value, u_int srcva, u_int perm);
```

`sys_ipc_try_send` 用于尝试向目标进程发送数据。其基本流程如下：

- 如果进程需要通过共享内存接受数据（即 `dstva` 不为 0），则需检查接收虚拟地址 `dstva` 是否合法。
- 获取接收进程对应的进程控制块。（注意这里并不要求接收进程是当前进程的子进程，因此 `checkperm` 参数应为假）
- 检查接收进程的接收状态，若接收进程为可接受状态，则正常发送消息，否则返回 `-E_IPC_NOT_RECV`，表明接收进程未处于可接受状态。
- 清除接收进程的可接收状态，将参数 `value`、`envid` 和 `prem` 分别赋给接受进程控制块的对应字段。
- 设置接收进程的状态为 `ENV_RUNNABLE`，并将进程控制块加入调度队列 `env_sched_list`。
- 如需要通过共享内存接受数据，则使用 `page_lookup` 函数查找 `srcva` 对应的物理页，并使用 `page_insert` 将该物理页映射到接收进程的 `env_ipc_dstva` 地址处。

fork 的实现

fork 整体流程

`fork` 函数实现了子进程的复制，是 Lab4 内容中较为复杂的部分。

```
int fork(void);
```

`fork` 函数的整体流程如下：

- 父进程调用 `fork` 函数
  - 调用 `syscall_set_tlb_mod_entry` 设置当前进程 TLB Mod 异常处理函数
  - 调用 `syscall_exofork` 函数为子进程分配进程控制块
  - 将当前进程低于 `USTACKTOP` 的虚拟页标记为写时复制映射到子进程虚拟地址空间
  - 调用 `syscall_set_tlb_mod_entry` 设置子进程 TLB Mod 异常处理函数
  - 调用 `syscall_set_env_status` 进程使子进程处于就绪状态，并将子进程加入调度队列 `env_sched_list` 中

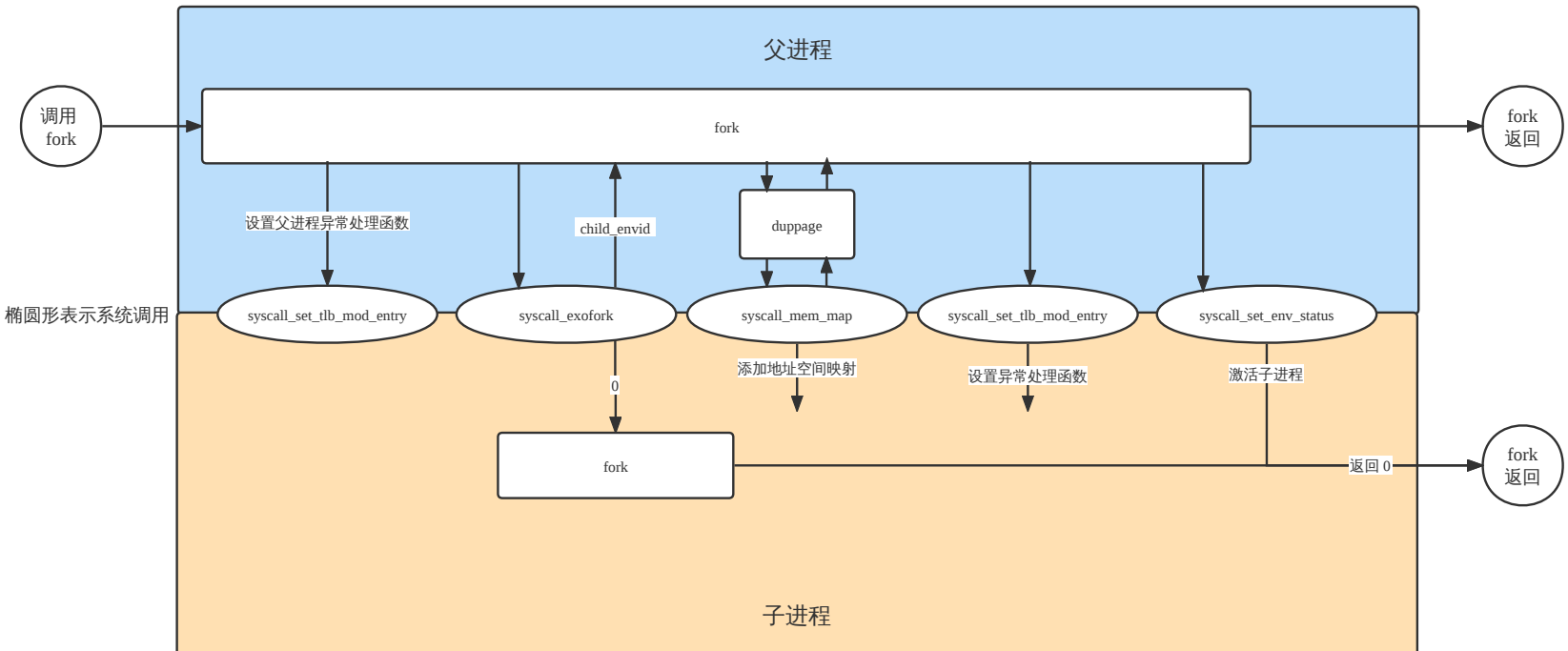
fork 的实现

fork 整体流程

- 子进程通过 `env_run` 函数启动运行
  - 最终通过调用 `ret_from_exception` 从内核栈的 `Trapframe` 中恢复上下文
  - 子进程从 `fork` 函数中的 `syscall_exofork` 调用处返回，返回值为 0
  - 子进程设置 `env` 变量，从 `fork` 中返回，返回值为 0

fork 的实现

fork 整体流程



fork 的实现

写时复制机制

通过 `fork` 函数创建的子进程是对父进程的复制，如果这时将父进程地址空间中的内容全部复制到新的物理页，将会消耗大量的物理内存。可以选择让父子进程暂时共享内存，当重新写入数据时才将该段内存分离。这就是**写时复制**。

当我们尝试写入的虚拟页对应的 TLB 项没有 `PTE_D` 标志（Dirty 位，表示是否可写入）时，会触发 TLB Mod 异常，此时会调用对应的异常处理函数。利用 TLB Mod 异常和其异常处理函数，我们实现写时复制机制。

如果我们希望两个进程共用一个虚拟页，我们可以将它们的虚拟页映射到同一个物理页。这样做可以让两个进程读到相同的内容。但是如果两个进程之一想要写入这个虚拟页，就会出现问题。因此，我们将 `PTE_D` 标志置 `0`。此时

- 当两个进程对该虚拟页进行**读**时，不会触发异常。
- 当两个进程对该虚拟页进行**写**时，就会触发 TLB Mod 异常。在异常处理函数中，我们将虚拟页映射到一个新的物理页，随后将旧的物理页的内容复制到新的物理页中。这样，两个进程的虚拟页便各自映射到了不同的物理页。

fork 的实现

写时复制机制

为了实现写时复制机制，我们将 `PTE_D` 标志置 `0`。但是此时还需区分两种情况：

- 虚拟页只读
- 虚拟页可写，但是是写时复制页

在这两种情况下，`PTE_D` 标志均为 `0`。因此，我们需要添加一个标志位 `PTE_COW`，用来区分这两种情况。

如果我们希望让一个虚拟页在两个进程中共享，只需要在两个进程中更新他们的页表项权限即可。因此，我们还需要实现的就是 TLB Mod 异常处理函数。

fork 的实现

写时复制机制

在 MOS 中，TLB Mod 异常的主要处理过程是在用户态运行的。内核态的处理函数 `do_tlb_mod` 会将当前上下文保存在用户态的**异常处理栈**中，并设置 `a0` 和 `epc` 寄存器的值，使得从异常恢复后能够跳转到 `env_user_tlb_mod_entry` 中所设置的用户异常处理函数的地址。

```
void do_tlb_mod(struct Trapframe *tf) {
    // ...

    if (curenv->env_user_tlb_mod_entry) {
        // ...
        tf->cp0_epc = curenv->env_user_tlb_mod_entry;
    } else {
        panic("TLB Mod but no user handler registered");
    }
}
```

fork 的实现

写时复制机制

用户态下的 TLB Mod 异常处理函数为 `cow_entry`。值得注意的是在 `cow_entry` 函数的最后，从异常处理中返回的方式。因为 `cow_entry` 已经处于用户态中，所以不能通过 `epc` 寄存器返回，这里的方法是使用系统调用 `syscall_set_trapframe` 恢复异常处理栈中保存的上下文。

```
static void __attribute__((noreturn)) cow_entry(struct Trapframe *tf) {
    // ...
    int r = syscall_set_trapframe(0, tf);
    user_panic("syscall_set_trapframe returned %d", r);
}
```

该函数的主要流程如下：

- 检查 `va` 对应的页是否为写时复制页面
- 使用系统调用 `syscall_mem_alloc` 分配一页物理页，暂时映射到虚拟地址 `UCOW` 处
- 将原页面的内容复制到新分配的页面，再将新分配的页面映射到虚拟地址 `va`，替换产生 TLB Mod 异常的页
- 取消到虚拟地址 `UCOW` 的临时映射，并使用系统调用 `syscall_set_trapframe` 恢复上下文

fork 的实现

系统调用：设置 Trapframe

从用户异常处理函数中返回的方式就是陷入一个新异常，在该异常中替换原本 `Trapframe` 的内容，从而在异常返回时根据替换后的上下文恢复现场。这就是系统调用 `sys_set_trapframe` 的基本原理。

```
int sys_set_trapframe(u_int envid, struct Trapframe *tf) {
    // ...
    if (env == curenv) {
        // ...
        *((struct Trapframe *)KSTACKTOP - 1) = *tf;
        return tf->regs[2];
    } else {
        // ...
    }
}
```

值得注意，在实现中我们使用 `*((struct Trapframe *)KSTACKTOP - 1) = *tf`，直接以结构体为单位进行复制。

fork 的实现

系统调用：设置 TLB Mod 异常处理函数

由于 TLB Mod 异常由用户态处理，因此需要一个系统调用为当前进程设置其 TLB Mod 异常处理函数。该系统调用即 `sys_set_tlb_mod_entry`。

```
int sys_set_tlb_mod_entry(u_int envid, u_int func);
```

该系统调用的流程较为简单：

- 通过进程的 `envid` 获取对应的进程控制块
- 在进程控制块中设置 TLB Mod 异常处理函数为 `func`

fork 的实现

系统调用：创建子进程

`sys_exofork` 系统调用实现了子进程的创建。该系统调用是父子进程真正“分支”的地方。实现了“一次调用，两次返回”。

```
int sys_exofork(void);
```

该函数的主要流程如下：

- 使用 `env_alloc` 创建子进程
- 将父进程的上下文复制到子进程中，同时将子进程上下文的返回值设置为 0
- 设置子进程为不可运行状态（`ENV_NOT_RUNNABLE`），并设置子进程的优先级与当前进程相同

由于当前子进程为不可运行状态，所以在调用 `sys_exofork` 后我们还可以对子进程做进一步的修改，如设置写时复制等等；又由于子进程的上下文是父进程的复制，所以子进程一旦开始运行，则同样会从 `sys_exofork` 处返回，和父进程一致。

fork 的实现

duppage 函数

在 `fork` 函数中我们遍历了父进程的页表

```
for (i = 0; i < PDX(UXSTACKTOP); i++) {
    if (vpd[i] & PTE_V) {
        for (u_int j = 0; j < PAGE_SIZE / sizeof(Pte); j++) {
            u_long va = (i * (PAGE_SIZE / sizeof(Pte)) + j) << PGSHIFT;
            if (va >= USTACKTOP) {
                break;
            }
            if (vpt[VPN(va)] & PTE_V) {
                duppage(child, VPN(va));
            }
        }
    }
}
```

对每个有效的页表项，我们使用 `duppage` 函数将其复制（duplicate）到子进程中。当然，由于写时复制的存在，本质上我们只是使用系统调用 `syscall_mem_map` 实现了对原有页的映射而已。

fork 的实现

duppage 函数

在 `duppage` 函数中，我们需要根据原页面权限设置新的映射权限，这样才能在子进程中通过 `PTE_D` 的设置触发 TLB Mod 异常。具体的权限设置规则如下：

- 原本便不可写、共享或就是写时复制的页面不需要更改其权限。
- 对可写、非共享且不是写时复制的页面，需要取消其可写位，设置写时复制位。然后将**其映射给子进程**，并**更新父进程这一页的权限**。

需要注意的是，在更新权限时，我们需要先将其映射给子进程，然后再更新父进程的权限。这是因为如果父进程先修改自己的权限位，则该页表就不再可写，这样的话可以会出现在映射到子进程之前页面便被写入的情况，此时父进程会触发写时复制，将该页面映射到新的物理页上，那么此时再映射到子进程的便是错误的页面。

fork 的实现

系统调用：设置进程状态

在 `fork` 函数的最后，我们使用 `sys_set_env_status` 系统调用设置子进程的运行状态为 `ENV_RUNNABLE` ，同时加入到调度队列中。之后，我们新创建的子进程便可以正常的按照调度流程运行了。

```
int sys_set_env_status(u_int envid, u_int status);
```

再次提醒，子进程此时的上下文依旧是父进程调用 `syscall_exofork` 时的上下文。因此子进程第一次运行时，也会从 `syscall_exofork` 之后继续执行。

Lab4 中若干次特殊的恢复

1. `fork` 创建的子进程的第一次启动：
   - 内核复制了父进程 `syscall_exofork` 时的上下文
   - 恢复后：从 `syscall_exofork` 返回
2. 触发写时复制：
   - 内核在 `do_tlb_mod` 中设置了调用 `cow_entry` 的上下文
   - 恢复后：如同 ”调用“ `cow_entry`
3. 用户态完成 TLB Mod 的处理
   - `cow_entry` 调用 `syscall_set_trapframe` 设置上下文为触发 TLB Mod 时的上下文
   - 恢复后：从触发写时复制的指令处继续执行

课下习题提示

- Exercise 4.1

  1. 使用 `syscall` 指令实现陷入
  2. 使用 `jr ra` 指令从 `msyscall` 函数中返回

- Exercise 4.2

  1. 注意并不存在 `a4`、`a5` 寄存器，`arg4` 和 `arg5` 存放在栈中
  2. 通过 `tf->regs[2]` 设置返回值

- Exercise 4.3

  使用 `ENVX` 宏获取 `envid` 对应的 `envs` 数组索引

- Exercise 4.4

  使用 `page_alloc` 申请物理页，使用 `page_insert` 创建映射

课下习题提示

- Exercise 4.5

  使用 `page_lookup` 查找源进程的源地址对应的物理页

- Exercise 4.6

  使用 `page_remove` 取消映射

- Exercise 4.7

  `schedule` 的 `yield` 参数设为真

- Exercise 4.8

  1. `sys_ipc_recv` 中 `curenv->env_ipc_dstva` 的值应设为 `dstva`
  2. `sys_ipc_try_send` 中使用 `e->env_ipc_recving` 判断目标进程是否在等待消息
  3. `sys_ipc_try_send` 中使用 `page_lookup` 和 `page_insert` 实现页面共享

课下习题提示

- Exercise 4.9

  直接使用赋值复制 `Trapframe` 结构体

- Exercise 4.10

  1. 使用 `vpn << PGSHIFT` 获得地址
  2. 使用 `vpt[vpn]` 获得表项，其中低 12 位为权限位
  3. 不**可写**（`PTE_D`）、**共享**（`PTE_LIBRARY`）或**写时复制**（`PTE_COW`）的页面不需要更改其权限。

- Exercise 4.11

  设置 `tf->cp0_epc` 的值为 `curenv->env_user_tlb_mod_entry`

- Exercise 4.12

  设置 `env->env_user_tlb_mod_entry` 的值为 `func`

课下习题提示

- Exercise 4.13

  1. 使用 `syscall_mem_alloc` 在 `UCOW` 处申请页面
  2. 使用 `memcpy` 复制数据，注意使用 `ROUNDDOWN` 使 `va` 对其
  3. 使用 `syscall_mem_map` 将 `UCOW` 处页面映射到 `va` 处
  4. 使用 `syscall_mem_unmap` 取消 `UCOW` 处的页面映射

- Exercise 4.14

  如果设为 `ENV_RUNNABLE` ，将进程插入调度队列尾部；如果设为 `ENV_NOT_RUNNABLE` ，将进程移出调度队列

- Exercise 4.15

  遍历页表时，先遍历页目录，如果页目录项有效，再遍历该页目录项对应的二级页表，对每个有效的页表项，调用 `duppage` 函数