OS_lab4实验报告

思考题

Thinking 4.1 系统调用的实现

- 陷入内核之后，内核会调用 SAVE_ALL 来保存用户进程的运行现场，包括32个通用寄存器的值。
- 系统陷入内核调用后如果$a0~$a3的值没有被修改，则可以直接调用。
- 在调用msyscall的时候，将前4个参数按顺序放入$a0~$a3中，然后，将最后两个参数按顺序存入内核中栈的相应位置中。内核在调用 sys_* 的时候也会读取相应的参数。
- 改变了v0寄存器的值，就是系统调用函数的返回值，可以根据返回值来判断系统调用是否成功；同时，还将EPC+4，确保还原之后，用户进程能够继续执行下一条指令，防止陷入循环。

Thinking 4.2 envid2env 的实现

确保通过 ENVX 索引到的进程控制块是这个 envid 对应的，如果不判断的话，可能会导致返回错误的进程控制块。

Thinking 4.3 mkenvid 函数细节

`mkenvid` 不会返回0，说明不会有进程的envid为0，那么在 `envid2env` 中，我们就可以使用0来代表当前的进程，而不会产生冲突。

Thinking 4.4 fork 的返回结果

C，`fork` 虽然只在父进程中调用一次，但在父进程和子进程中都有返回值。

Thinking 4.5 用户空间的保护

映射的地址肯定都在kuseg中。用户空间中ULIM到USTACKTOP之间的部分，存储了页表信息以及异常处理栈等，这部分内容由所有进程共享，所以只需要映射USTACKTOP以下的地址。

映射的时候只需要映射可写的，也就是PTE_V为1的页面。

Thinking 4.6 vpt 的使用

```
#define vpt ((const volatile Pte *)UVPT)
#define vpd ((const volatile Pde *)(UVPT + (PDX(UVPT) << PGSHIFT)))
```

- vpt是用户页表的地址，vpd是用户页目录的地址。可以根基这两个地址找到需要的页面。
- vpd采取了页目录自映射机制，等价于 `vpd = UVPT | UVPY >> 10` 即页目录自映射保证了可以通过访问UVPT访问页目录即vpd，同时访问UVPT可以访问所有页表。
- 不能，页表是由内核态维护的，用户态必须通过中断异常陷入内核才能修改页表，不能直接修改。

Thinking 4.7 页写入异常-内核处理

- 缺页异常处理函数是自定义的，通过将EPC设置成curenv->env_user_tlb_mod_entry来调用指定的处理函数，后续其他的自定义异常处理函数可能会产生tlb_mod异常，发生异常重入；

- 异常处理函数在用户态进行，访问不到内核空间，将异常的现场Trapframe复制到用户空间，才能知道异常发生使得状态。

Thinking 4.8 页写入异常-用户处理-1

在内核态写入异常的时候，如果产生失误，可能会使操作系统崩溃，用户态产生失误的时候则不会影响其它的进程。同时在用户态写入也能避免对内核空间的不必要修改。

Thinking 4.9 页写入异常-用户处理-2

- 因为要先放置好页面写入异常的异常处理函数地址，可能在duppage中也会发生页写入异常。
- 可能在duppage的时候有页写入异常，但是没有异常处理的情况。

难点分析

本次实验主要分为了三个部分：

- 系统调用
- 进程间通信
- `fork`函数

系统调用

以用户函数`dubugf`为例，在函数运行的时候，经历了这样的函数调用过程：`debugf -> vdebugf -> vprintfmt -> debug_output -> debug_flush -> syscall_print_cons`。最终，为了向屏幕输出字符，用户程序还是使用了系统调用`syscall_print_cons`。

```
// user/lib/syscall_lib.c
int syscall_print_cons(const void *str, u_int num) {
    return msyscall(SYS_print_cons, str, num);
}
```

在用户程序中，所有的系统调用均为`syscall_*`的形式，所有的系统调用都定义在 user/lib/syscall_lib.c 中。可以看到，所有的系统调用函数都调用了函数`msycall`，该函数的第一个参数代表了不同的系统调用类型，后面的参数是相应系统调用所需要的。

```
LEAF(msyscall)
    // Just use 'syscall' instruction and return.
    syscall
    jr  ra
END(msyscall)
```

`msyscall`函数也不复杂，主要就是使用了`sysscall`指令，该指令使程序产生一个系统异常，之后程序就能陷入内核态，进行异常处理。之后就是lab3的异常分发过程，这中间使用了`SAVE_ALL`宏保存了发生异常时的现场，存储在了trap frame中。最后调用了`do_syscall`函数。

`do_syscall`函数的功能是查找并调用内核中相应的`sys_*`函数。函数的参数可以从用户进程的trap frame中查到，前4个参数可以从寄存器a0~a3查到，最后两个可以用栈指针寄存器sp找到。根据第一个参数可以从`syscall_table`查到相应的`sys_*`函数，最后将函数的返回值写入寄存器v0。除此之外`do_syscall`还要将EPC寄存器加4，使得系统调用结束之后，用户程序能够执行下一条指令。

进程间通信

由于进程之间共享一个内核空间，因此进程之间的通信可以通过内核空间来实现。以下是进程控制块之间与进程通信有关的属性：

```
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
};
```

我们主要实现两个系统调用 `syscall_ipc_try_send` 和 `syscall_ipc_recv`：

- `syscall_ipc_try_send`：
  - 检查虚拟地址是否处于用户空间。
  - 设置进程控制块的字段，`env_ipc_recving` 和 `env_ipc_dstva`。
  - 阻塞当前进程，将该进程从调度队列中移出。
  - 将返回值设置为0，调用 `schedule` 函数进行进程切换。
- `syscall_ipc_recv`：
  - 判断地址是否正确。并通过 envid 获取进程控制块。
  - 检查 `env_ipc_recving`，这一字段在信息接收时设置。
  - 传输一些信息，并将 `env_ipc_recving` 重新置0。
  - 取消接收进程的阻塞状态。
  - 将当前进程的一个页面共享到接收进程。使得接收进程才能通过该页面获得发送进程发送的一些信息。

fork函数

`fork` 是创建进程的基本方法。

设置TLB Mod异常处理函数

```
/* Step 1: Set our TLB Mod user exception entry to 'cow_entry' if not done yet.
*/
    if (env->env_user_tlb_mod_entry != (u_int)cow_entry) {
        try(syscall_set_tlb_mod_entry(0, cow_entry));
    }
```

`fork` 函数首先会设置TLB Mod异常处理函数，该过程使用了系统调用 `env_user_tlb_mod_entry`，设置了进程控制块中的 `env_user_tlb_mod_entry` 参数。

写时复制技术（Copy on Write, COW）

`fork` 会根据复制调用进程来创建一个新进程。可如果每创建一个新的进程就要在内存中复制一份相同的数据，开销就太大了。所以在创建子进程时只是让子进程映射到和父进程相同的物理页。这样如果父进程和子进程只是读取其中的内容，就可以共享同一片物理空间。当有进程需要修改内存中的数据时，再将这块物理空间复制一份，让想要修改的进程只修改属于自己的数据。在`fork`中，我们使用`cow_entry`函数来实现写时复制技术。

```
perm = vpt[VPN(va)] & 0xfff;
if (!(perm & PTE_COW)) {
    user_panic("perm doesn't have PTE_COW");
}
perm = (perm & ~PTE_COW) | PTE_D;
```

首先，先使用`vpt`获取当前虚拟地址的页表项，并查看是否有PTE_COW权限，之后再重新设置该页表项的权限。

```
syscall_mem_alloc(0, (void *)UCOW, perm);
memcpy((void *)UCOW, (void *)ROUNDDOWN(va, PAGE_SIZE), PAGE_SIZE);
```

然后，使用系统调用来申请新的物理页面，并将原先的页面内容复制到新的物理页面上。

```
panic_on(syscall_mem_map(0, (void *)UCOW, 0, (void *)ROUNDDOWN(va, PAGE_SIZE),
perm));
panic_on(syscall_mem_unmap(0, (void *)UCOW));
int r = syscall_set_trapframe(0, tf);
user_panic("syscall_set_trapframe returned %d", r);
```

最后，只需要取消`va`到原物理页的映射，将`va`映射到新申请的物理页，然后调用`syscall_set_trapframe`来恢复异常处理之前的现场。

创建子进程

```
child = syscall_exofork();
if (child == 0) {
    env = envs + ENVX(syscall_getenvid());
    return 0;
}
```

`fork`函数的第二步是使用`syscall_exofork`创建子进程，在此调用之后的一条语句，我们就通过不同返回值实现了父子进程的不同流程，对于子进程来说，我们设置了 `env` 的值为当前进程，然后直接返回0，而父进程继续执行。

```
for (i = 0; i < VPN(USTACKTOP); i++) {
    if((vpd[i >> 10] & PTE_V) && (vpt[i] & PTE_V)) {
        duppage(child, i);
    }
}
try(syscall_set_tlb_mod_entry(child, cow_entry));
try(syscall_set_env_status(child, ENV_RUNNABLE));
```

父进程首先循环调用duppge函数将USTACKTOP下所有有效的页进行复制，然后设置子进程的异常处理函数和状态。

心得体会

lab4完成了系统调用的部分，在此基础上完成了ipc和fork。这次的实验还是有一定难度的，需要考虑到用户态和内核态的切换。想要更好的理解实验内容，对于系统调用，fork函数的实现过程以及相应调用的函数需要有比较清晰的认识。虽然完成代码过程中的debug很痛苦，但是在这个过程中我又将代码重读了很多遍，对代码理解也更加深刻了。