BUAA OS Lab4-2 课上测试

Extra部分

（一）题目

由于原题目比较长，笔者在这里只做简单的概括，原题目可见最后附录。

在Extra部分，我们需要实现信号机制，包括以下三种信号：

信号	编号	来源	描述	默认处理动作
SIGTERM	15	用户程序	用于终止进程，但允许目标进程通过信号处理函数拦截	退出
SIGSEGV	11	操作系统	用户程序访问了页表中未映射且地址严格小于 0x10000 的虚拟页	退出
SIGCHLD	18	操作系统	进程的某个子进程退出	忽略

以及在user/ipc.c中实现两个函数（以及其他必要函数等）：

```
void kill(u_int envid, int sig);
```

该函数向 envid 为 envid 的进程发送编号为 sig 的信号。 若 envid 为 0 或当前进程的 envid，则向当前进程自身发送信号。kill只会发送编号为15的信号。

```
void signal(int sig, void (*handler)(int));
```

该函数为当前用户进程注册一个专门处理sig信号的处理函数handler。当handler为0时，取消之前注册的处理函数；如果之前已经注册过了，则会覆盖。处理时，sig要作为参数传递给handler。

实现提示（概括）：

1. 为使用户进程进入处理函数以及从处理函数中返回，可能需要新增系统调用
2. 注册处理函数可以模仿COW机制中用户注册异常处理函数(syscall_set_pgfault_handler)。
3. 可以通过修改进程的上下文（Trapframe）使进程从原来流程跳转到处理函数中。需要注意该进程的上下文保存在哪，以及修改的寄存器。
4. SIGSEGV的原行为是mm/pmap.c中的TOO LOW，需要修改。
5. 不要修改Env结构体。
6. 可以模仿异常向量表进行信号的分发。

(二) 解题思路

笔者在课上光是读懂题就花了好久，整理思路又花了好久，不知不觉17点就到来了（coding的时间总是飞速流逝~）。于是，笔者在课下重新读题，按照课上未完成的思路继续前进，最终整理出了一点思绪，通过了课下的测试，在这里和大家分享（当然，和题目给的实现提示有一定的差异，也希望能和大家一起讨论其他的实现方法）。

笔者认为本题的难点有三个：

1. **如何使一个进程知道自己收到了信号**
2. **如何使一个进程跳转到处理函数**
3. **如何使一个进程经过处理函数后能够恢复原来的现场**

此外，还需要分清何时处于内核态，何时处于用户态。

1. 如何使一个进程知道自己收到了信号

一个进程一旦开始跑起来了，如果没有外界（如操作系统）的中断，它只会按照自己的逻辑一直跑下去，即使其他进程给它发了信号，它也不知道。因此，我们需要内核来“通知”进程接收信号。

这就意味着，发送方（另一个平平无奇的进程）不能只依靠自己的力量来发送信号，它必须通过系统调用，请求内核帮它发送信号。因此，如果一个进程想发送信号，那么它就需要调用相应的系统调用，陷入内核态，由操作系统来完成。

对此，笔者增加了新的系统调用（这里只先简单地声明）

```
int sys_send_sig(int sysno, int envid, int sig);
// envid为发送的目标 sig为信号编号
```

2. 如何使一个进程跳转到处理函数

使进程从原来的流程跳转到处理函数，需要暴力修改PC，但是让进程自己修改显然不太可能（因为一个进程本身根本无法知道自己是否收到了信号）。这时候，又需要我们强大的内核出场了。

这里我们可以把目标进程（即接收信号的进程）分为两类。

1. 向自己发送信号的进程（发送方 == 接收方）
2. 被别人发送信号的进程（发送方 != 接收方）

由前面的分析，我们知道一个进程想发送信号，需要通过系统调用陷入内核态。因此，对于第一种情况（自己给自己发信号），此时已经处于内核态了，那么根据异常处理的流程（系统调用也属于一种异常），我们知道**该进程的上下文保存在`KERNEL_SP - sizeof(struct Trapframe)`的位置，而从异常返回时，也将用这个位置的Trapframe恢复上下文。**于是，我们在内核态下**直接修改这个位置上下文的EPC为处理函数地址**，那么从异常返回时，将上下文中的EPC赋值给PC，就跳转到处理函数了。

而对于第二类（被别人发送信号的），此时该进程一定处于未被调度的状态（因为当前调度的是发送方）。根据env_run的内容，我们知道进程被调度时，**会用Env结构体中的Trapframe来恢复上下文**，因此我们**可以直接修改Env结构体中上下文的PC为处理函数地址（注意，这里是PC而不是EPC，因为恢复时直接用PC恢复PC）**，这样下次该进程被调度时，自然就跳转到了处理函数。

```
struct Trapframe* old; // 保存的上下文
if (envid == curenv->env_id) {
    /* 发送方==接收方
        上下文保存在KERNEL_SP - sizeof(struct Trapframe)
    */
```

```
    old = (struct Trapframe*)(KERNEL_SP - sizeof(struct Trapframe));
    /* ... */
    old->cp0_epc = handler;
    /* ... */
}
else {
    /* 发送方!=接收方
        上下文保存在Env结构体中
    */
    struct Env* e;
    envid2env(envid, &e, 0);
    old = &(e->env_tf);
    /* ... */
    old->pc = handler;
    /* ... */
}
```

3. 如何使一个进程处理完后恢复现场

首先我们需要明确，由谁来恢复现场？用户程序还是内核？当然，两种方法都可以，不过笔者选择的是**由内核来恢复**，毕竟内核中的异常返回`ret_from_exception`正是一个绝佳的恢复现场的操作。（用户态下的__asm_pgfault_handler同样有恢复现场的操作可以参考）

这里的难点就在于，用户处理函数执行完毕后，怎么让内核恢复现场。（这也是笔者在课上思维最混乱的地方）

笔者的想法是，处理函数执行完毕后，进行一个系统调用，陷入内核态，由内核来恢复现场。**但是处理函数并不是我们能写的（它在远端的评测机里~），意味着我们不能在处理函数的末尾加上一个**`syscall_xxx()`。因此，我们需要寻找别的方法来使用户进程在处理函数执行完毕后，跳转到某个用户空间中我们能够coding的地方进行一个系统调用。

这个地方既要是用户空间的，又要是我们"能够写"的，那么就是user/libos.c中了（虽然这个文件经常被忽视hhh）。我们在其中加入一个函数来进行系统调用，让内核帮忙完成上下文恢复：

```
void restore() {
    syscall_restore(); // syscall_resotre的实现后文说明
}
```

现在我们面临了一个新的问题，**如何让用户程序在执行完处理函数**`handler`**后跳到我们刚刚写的这个**`restore()`**呢?** 这时候，计组留下的宝贵知识就起作用了。每个函数编译后，最后都得来条`jr $ra`，那如果我们提前在**内核态中**把进程上下文的`$ra`改成`restore()`的地址呢？这时，恢复上下文后，当执行完`handler`，return时用户进程就乖乖地跳到`restore()`来了。

```
struct Trapframe* old;
/* ... 经过前面的一番操作，old指向了保存的上下文 ... */
old->regs[31] = restore_addr;
```

好了，我们又遇到了新的问题。这个`restore()`是用户空间的函数，内核并不知道该函数的地址。不过这个问题我们在COW机制中用户处理函数`__asm_pgfault_handler`也遇到过，可以通过系统调用来注册，告诉内核。

于是，我们又增加了一个系统调用`syscall_set_restore_addr(int envid, int restore_addr)`来告诉内核用户空间中`restore()`函数的地址。

到这里，我们整理一下思路：

1. 某个进程想要发送信号，它需要通过系统调用`syscall_send_sig`让内核帮它发送
2. 内核需要修改接收方上下文中的pc，使接收方在恢复上下文后跳转到`handler`.其中，两类接收方存储上下文的地方不一样，修改的EPC或PC也不一样
3. 内核还需要修改接收方上下文中的`$ra`为用户空间中的`restore()`,使用户程序在完成`handler`后，`return`到`restore()`中发出系统调用`syscall_restore`，由内核来恢复其原来的上下文
4. 当然，在第3步之前，进程需要通过系统调用`syscall_set_restore_addr`来告诉内核，用户空间中`restore()`的地址

有了大致思路，我们开始进行具体的实现。

（三）具体实现

这里只讲系统调用的主体部分的实现，具体增加系统调用的流程可以参考lab4-1的内容。

1. sys_set_sig_handler 注册handler，告诉内核handler的地址

内核中需要有地方保存每个进程的handler，那我们就开个大数组来保存handler的地址吧。

```
// lib/syscall_all.c

int handlers[3 * NENV]; // 每个进程可以有3个类型的handler
// 规定handlers中，[0-2]为第一个进程的11、15、18号handler [3-5]为第二个进程的11、15、18号handler ...

int get_handler_index(int envid, int sig) {
    // 返回对于envid的进程的sig类型的handler的index
    if (envid == 0) envid = curenv->env_id;
    int index = ENVX(envid);
    index = index * 3;
    if (sig == 11) {
        index = index + 0;
    }
    else if (sig == 15) {
        index = index + 1;
    }
    else if (sig == 18) {
        index = index + 2;
    }
    else panic("wrong sig num");
    return index;
}
```

有了地方存handler，我们就可以加系统调用来进行注册了。

```
// lib/syscall_all.c

int sys_set_sig_handler(int sysno, int envid, int sig, int handler) {
    if (envid == 0) envid = curenv->env_id;
    int index = get_handler_index(envid, sig);
    handlers[index] = handler; // 根据题意直接覆盖(NULL时也直接覆盖)
    return 0;
}
// user/ipc.c
void signal(int sig, void (*handler)(int)) {
    int addr = (int) handler;
    syscall_set_sig_handler(sig, handler);
}
```

2. sys_set_restore_addr 注册restore，告诉内核restore的地址

同理，我们在内核中开个大数组。

```
// lib/syscall_all.c

int restore_addrs[NENV];
int sys_set_restore_addr(int sysno, int envid, int addr) {
    if (envid == 0) envid = curenv->env_id;
    int index = ENVX(envid);
    restore_addrs[index] = addr;
    return 0;
}
```

那么，用户进程需要在什么时候通过这个系统调用来告诉内核`restore`的地址呢？如果是一般进程（不是fork出来的），那么在进入`umain`前调用就行（**即在user/libos.c中`libmain()`中进入`umain`之前调用**）。

而对于fork出来的子进程，那么需要父进程帮它设置。因此**需要在fork函数中调用`syscall_set_restore_addr`由父进程为子进程设置。**

3. sys_send_sig 发送信号（不仅仅是发送信号...）

在笔者的实现中，恢复现场的操作是由内核完成，因此内核在修改上下文之前，需要先保存下接收方的原始上下文，以便于最后恢复用户程序原来的流程。我们同样开个大数组来保存原始上下文。

```
struct Trapframe trapframes[NENV]; // 保存原始上下文（原来进程的正常流程）的大数组
```

经过前面的分析，我们知道，`sys_send_sig`的主要功能有：

1. 保存接收方的原始上下文（复制到大数组中）
2. 修改接收方上下文的`pc`为`handler`地址（如果有的话）
3. 修改接收方上下文的`$ra`为`restore`地址
4. 修改其他必要的寄存器（如`$a0`传参给handler）

```
int sys_send_sig(int sysno, int envid, int sig) {
    if (envid == 0) envid = curenv->env_id;
    int env_index = ENVX(envid);
```

```
    int handler_index = get_handler_index(envid, sig);
    struct Trapframe* old = NULL; // 原来保存的上下文
    struct Env* e = NULL;
    int r;

    if (envid == curenv->env_id) { // 发送方 == 接收方
        old = (struct Trapframe*)(KERNEL_SP - sizeof(struct Trapframe));
        e = curenv;
    }
    else { // 发送方 != 接收方
        r = envid2env(envid, &e, 0);
        if (r < 0 && sig == 18) return 0; // sig == 18 且父进程已经退出 不必发送
SIGCHLD
        if (r < 0) panic("cannot find env");
        old = &(e->env_tf);
    }

    int handler = handlers[handler_index]; // 获取handler地址
    int restore_addr = restore_addrs[env_index]; // 获取restore地址
    if (handler == 0) { // 没有注册handler 默认处理
        if (sig == 11 || sig == 15) env_destroy(e); // 直接退出
        else return 0; // sig == 18 忽略 啥也不干
    }
    trapframes[env_index] = *old; // 复制原始上下文到大数组中
    old->regs[4] = sig; // 上下文的$a0设置为传参 sig
    old->regs[31] = restore_addr; // 上下文的$ra设置restored的地址
    if (envid == curenv->env_id) {
        old->cp0_epc = handler; // 上下文的epc设置为handler地址
        //此时发送方==接收方 是通过系统调用进入的 异常返回时用epc恢复
    }
    else old->pc = handler; // 上下文的pc设置为handler地址
    //此时发送方!=接收方 接收方未调度 进程调度时用pc恢复
    return 0;
}
```

4. sys_restore 用户进程处理完毕后，内核帮忙恢复上下文

由`sys_send_sig`的实现我们知道，进程的原始上下文保存在大数组`trapframes`中。而从系统调用返回时，又会用`KERNEL_SP - sizeof(struct Trapframe)`处的上下文恢复。因此，我们**只要把大数组`trapframes`中的上下文复制到`KERNEL_SP - sizeof(struct Trapframe)`处，然后`ret_from_exception`就会帮我们完成恢复工作了。**

```
int sys_restore(int sysno, int envid) {
    if (envid == 0) envid = curenv->env_id;
    int index = ENVX(envid);
    struct Trapframe* old = &(trapframes[index]);
    bcopy(old, KERNEL_SP - sizeof(struct Trapframe), sizeof(struct Trapframe));
    return 0;
}
```

5. sys_copy_handler 子进程需要继承父进程的handler

题目明确指出，子进程需要继承父进程的`handler`。而`handler`的信息我们保存在了内核中，因此需要父进程发起一个系统调用，把父进程的`handler`复制给子进程。

```
int sys_copy_handler(int sysno, int envid, int penvid) {
    // envid为子进程id penvid为父进程envid
    if (envid == 0) envid = curenv->env_id;
    int i,j;
    i = get_handler_index(envid, 11);
    j = get_handler_index(penvid, 11);
    handlers[i] = handlers[j];
    i = get_handler_index(envid, 15);
    j = get_handler_index(penvid, 15);
    handlers[i] = handlers[j];
    i = get_handler_index(envid, 18);
    j = get_handler_index(penvid, 18);
    handlers[i] = handlers[j];
    return 0;
}
```

父进程需要在fork中通过这一系统调用把`handler`传递给子进程。

6. 用户空间的restore

```
// user/libos.c
void restore() {
    syscall_restore();
}
```

7. SIGTERM相关的函数

```
void kill(u_int envid, int sig) {
    syscall_send_sig(envid, sig);
}
```

8. SIGCHLD相关的函数

由于子进程在退出时，操作系统需要向父进程发送信号，因此我们可以在用户空间的`exit`函数中，发起一个系统调用，向父进程发送信号。

```
void
exit(void)
{
    //close_all();
    if (env->env_parent_id != 0) {
        syscall_send_sig(env->env_parent_id, 18); // 通知父进程
    }
    syscall_env_destroy(0);
}
```

9. SIGSEGV相关的函数

由于原来的内核实现中，访问地址严格小于0x10000的非法地址会出现无比熟悉的TOO LOW，因此我们需要修改mm/pmap.c中的pageout函数，把原来的panic改成发送相应的信号。

```
// mm/pmap.c
void pageout(int va, int context) {
    // ... //
    if (va < 0x10000) {
        sys_send_sig(0, 0, 11); // 向当前进程发送编号11的信号（第一个参数为系统调用号，无所谓；第二个参数为envid）
        set_sp(KERNEL_SP - sizeof(struct Trapframe)); // 设置sp寄存器
        ret_from_exception(); // 直接从异常中返回
        return;
    }
    // ... //
}

// mm/tlb_asm.S
LEAF(set_sp)
nop
addu sp, a0, zero
jr ra
nop
END(set_sp)
```

其中，由于笔者对do_refill的具体内容的理解还不够透彻，不知道为什么触发tlb异常时，sp不会指向`KERNEL_SP - sizeof(struct Trapframe)`，因此写了一个汇编函数来直接设置sp的值，同时也直接调用`ret_from_exception`来强制恢复上下文，跳转到`handler`中。

以上就是本次Extra的思路分享，关于笔者遇到的tlb异常处理时do_refill中sp不指向`KERNEL_SP - sizeof(struct Trapframe)`的问题，真诚地希望了解的同学能够不吝赐教！如果上述思路及实现有任何错误或可以改进的地方，也麻烦大家指正，谢谢！