sigaction 实验报告

写在开头：

强烈建议下一届能够更加详细地说明一下挑战性任务的一些具体实现要求，网站上的说明文档写的有点过于简略了，我花了好长时间阅读往届博客、阅读参考文档，请教室友等等，最后才对整个任务有了基本的了解，在debug的时候，也经常出现因为忽略了某一些要求而出现问题。不过最后能完成，真是太好了~~~

任务要求

这里参考了助教在讨论区的补充说明。

我们需要完成sigaction的基本操作，这是一种进程之间的异步通信。这种信号可以由进程主动向其它进程发出（使用kill），也可以是进程在运行过程中产生的，比如出现非法指令的时候，会向自己发送SIGILL信号。

信号的处理过程如下：

可以看出，信号是在进程从内核态返回用户态的时候被处理的，这一点非常关键。

从内核态返回用户态”时，如果有多个信号待处理，则需要选择优先级最高的进行处理。

实现过程

信号的存储

很显然，进程中一定要保存着该进程中与sigaction相关的信息，所以首先，我们应当在进程控制块中添加一些内容。

首先，建立`include/signal.h`头文件，定义一些结构体和宏：

```
#ifndef __SIGNAL_H__
#define __SIGNAL_H__

typedef struct sigset_t {
    uint32_t sig;  //代表是否屏蔽32个不同的信号
} sigset_t;

struct sigaction {
    void     (*sa_handler)(int);  //信号处理函数
    sigset_t   sa_mask;  //掩码
};
//几种特殊的信号
#define SIGINT 2
#define SIGILL 4
#define SIGKILL 9
#define SIGSEGV 11
#define SIGCHLD 17
#define SIGSYS 31
//sigprocmask函数中__how的不同值
#define SIG_BLOCK 0
#define SIG_UNBLOCK 1
#define SIG_SETMASK 2
```

```
#endif
```

然后，在`include/env.h`中的Env结构体中加入一些和信号相关的成员属性：

```
struct Env {
    //...

    // sigaction
    struct sigaction env_sigaction[33];  //1-32代表32个信号的处理函数和掩码
    sigset_t env_sa_mask;  //代表当前进程屏蔽的信号
    u_int env_sig_recv;  //32位，分别代表当前进程是否接受到该信号
    u_int env_sig_entry;  //信号处理入口

    u_int env_protect[256];//防止内存泄漏
};
```

当然，也要在`kern/env.c`中进行初始化：

```
    for (u_int i = 1; i <= 32; i++) {
        e->env_sigaction[i].sa_handler = 0;
    }
    e->env_sig_recv = 0;
    e->env_sig_entry = 0;
```

信号的操作函数

包括了需要在内核中进行处理的信号注册函数`sigaction`、信号发送函数`kill`、两个信号集处理函数`sigprocmask`和`sigpending`，以及另外8个不需要在内核中处理的信号集处理函数。这些函数都在`user/lib/libos.c`中实现，相应的函数声明在`user/include/lib.h`。

添加系统调用

首先回顾一下添加系统调用的方法：

1. 在`user/include/lib.h`中添加：
    - `void user_func(u_int envid, ......);`
    - `void syscall_func(u_int envid, ......);`
2. 在`user/lib/syscall_lib.c`中添加：

    ```
    int syscall_func(u_int envid, ......) {
        return msyscall(SYS_func, envid, ......);
    }
    ```

3. 在`user/lib`中实现`user_func`函数的文件中编写具体代码（其中会调用`syscall_func`函数）。
4. 在`include/syscall.h`中的`enum`的`MAX_SYSNO`前面加上`SYS_func`，（注意有逗号）。
5. 在`kern/syscall_all.c`的`void *syscall_table[MAX_SYSNO]`的最后加上`[SYS_func] = sys_func`，（注意有逗号）。
6. 在`kern/syscall_all.c`的`void *syscall_table[MAX_SYSNO]`的前面具体实现函数`void syscall_func(u_int envid, ......);`。

5个在内核中进行处理的函数

```
// user/lib/libos.c
int sigaction(int signum, const struct sigaction *newact, struct sigaction
*oldact) {
    if (signum < 1 || signum > 32) {
        return -1;
    }
    if (syscall_get_sig_act(0, signum, oldact) != 0) {
        return -1;
    }
    return syscall_set_sig_act(0, signum, newact);
}
int kill(u_int envid, int sig) {
    return syscall_kill(envid, sig);
}
int sigprocmask(int __how, const sigset_t * __set, sigset_t * __oset) {
    return syscall_sigprocmask(0, __how, __set, __oset);
}
int sigpending(sigset_t *__set) {
    return syscall_sigpending(0, __set);
}

// kern/syscall_lib.c
int sys_get_sig_act(u_int envid, int signum, struct sigaction *oldact) {
    struct Env *env;
    if (envid2env(envid, &env, 0) < 0) {
        return -1;
    }
    if (oldact != NULL) {
        oldact->sa_handler = env->env_sigaction[signum].sa_handler;
        oldact->sa_mask = env->env_sigaction[signum].sa_mask;
    }
    return 0;
}
int sys_set_sig_act(u_int envid, int signum, struct sigaction *act) {
    struct Env *env;
    if (envid2env(envid, &env, 0) < 0) {
        return -1;
    }
    if (act == NULL) {
        return -1;
    }
    env->env_sigaction[signum].sa_handler = act->sa_handler;
    env->env_sigaction[signum].sa_mask = act->sa_mask;
    return 0;
}
int sys_kill(u_int envid, int sig) {
    //printk("sys_kill start: signum=%d\n", sig);
    if (sig < 1 || sig > 32) {
        return -1;
    }
    struct Env *env;
    if (envid2env(envid, &env, 0) < 0) {
        return -1;
```

```
    }
    env->env_sig_recv |= (1 << (sig - 1));
    return 0;
    //printk("sys_kill end\n");
}
int sys_sigprocmask(u_int envid, int how, const sigset_t *set, sigset_t *oset) {
    struct Env *env;
    if (envid2env(envid, &env, 0) < 0) {
        return -1;
    }
    if (oset != NULL) {
        oset->sig = env->env_sa_mask.sig;
    }
    if (set == NULL) {
        return -1;
    }
    if (how == SIG_BLOCK) {
        env->env_sa_mask.sig |= set->sig;
    } else if (how == SIG_UNBLOCK) {
        env->env_sa_mask.sig &= (~set->sig);
    } else if (how == SIG_SETMASK) {
        env->env_sa_mask.sig = set->sig;
    } else {
        return -1;
    }
    return 0;
}
int sys_sigpending(u_int envid, sigset_t *set) {
    struct Env *env;
    if (envid2env(envid, &env, 0) < 0) {
        return -1;
    }
    if (set == NULL) {
        return -1;
    }
    set->sig = env->env_sa_mask.sig & env->env_sig_recv & ~(1 << 8);  // 因为
SIGKILL不可被阻塞，所以要特判一下
}
```

8个不需要在内核中处理的信号集处理函数

```
int sigaddset(sigset_t *__set, int __signo) {
    if (__set == NULL) {
        return -1;
    }
    if (__signo < 1 || __signo > 32) {
        return -1;
    }
    __set->sig |= SIGSET(__signo);
    return 0;
}
int sigdelset(sigset_t *__set, int __signo) {
    if (__set == NULL) {
        return -1;
    }
```

```
    if (__signo < 1 || __signo > 32) {
        return -1;
    }
    __set->sig &= (~SIGSET(__signo));
    return 0;
}
int sigismember(const sigset_t *__set, int __signo) {
    if (__set == NULL) {
        return -1;
    }
    if (__signo < 1 || __signo > 32) {
        return -1;
    }
    return ((SIGSET(__signo) & __set->sig) != 0);
}
int sigisemptyset(const sigset_t *__set) {
    if (__set == NULL) {
        return -1;
    }
    return (__set->sig == 0);
}
int sigandset(sigset_t *__set, const sigset_t *__left, const sigset_t *__right) {
    if (__left == NULL || __right == NULL || __set == NULL) {
        return -1;
    }
    __set->sig = __left->sig & __right->sig;
    return 0;
}
int sigorset(sigset_t *__set, const sigset_t *__left, const sigset_t *__right) {
    if (__left == NULL || __right == NULL || __set == NULL) {
        return -1;
    }
    __set->sig = __left->sig | __right->sig;
    return 0;
}
```

需要注意的是，这些函数中的输入参数的各种非法形式都要考虑到，不然评测的时候很可能会出bug。

用户态异常处理函数

上一过程只是完成了sigaction中使用到的一些工具函数，现在我们要完成处理信号的过程。过程大致仿造tlb_mod异常处理思路。

前面说过，信号的处理是在内核态返回用户态的时候进行的，具体过程是`kern/genex.S`在返回的时候，进程会跳转到`kern/tlbex.c`中的`do_signal`函数，在这里，进程会判断是否要进入`sig_entry`中处理信号。

异常处理函数的跳转

在`kern/genex.S`中我们知道，进程从内核态恢复到用户态的时候，都要通过`ret_from_exception`汇编函数，因此在这个函数里面加入跳转到`do_signal`的功能。

```
FEXPORT(ret_from_exception)
    # jump to do_signal
    move    a0, sp
    addiu   sp, sp, -24
    jal     do_signal
    nop
    addiu   sp, sp, 24

    RESTORE_ALL
    eret
```

`do_signal`函数的功能是：判断是否接下来要跳转到`sig_entry`用户态处理函数。

```
void do_signal(struct Trapframe *tf) {
    //printk("do_signal start\n");
    u_int sig_recv = curenv->env_sig_recv;
    u_int signum = 0;
    for(int i = 1; i <= 32; i++) { //遍历找到优先级最高，也就是编号最小的待处理信号
        if((sig_recv & 1 == 1) && (((1 << (i - 1)) & curenv->env_sa_mask.sig) ==
0)) {
            signum = i;
            break;
        }
        sig_recv = sig_recv >> 1;
    }
    if (curenv->env_sig_recv & (1 << 8)) { //如果有SIGKILL信号，就需要立刻处理
        signum = 9;
        curenv->env_sa_mask.sig = 0xffffffff;
    }
    if (signum == 0) { //如果没有需要处理的，就直接返回，不进入sig_entry
        return;
    }
    curenv->env_sig_recv &= (~(1 << (signum - 1)));
    if (signum == 31 && !curenv->env_sigaction[signum].sa_handler) { //对于
SIGSYS，注意epc+4的问题
        tf->cp0_epc += 4;
    }

    //接下来就像do_tlb_mod一样，设置好参数，调用sig_entry函数
    struct Trapframe tmp_tf = *tf;
    if (tf->regs[29] < USTACKTOP || tf->regs[29] >= UXSTACKTOP) {
        tf->regs[29] = UXSTACKTOP;
    }
    tf->regs[29] -= sizeof(struct Trapframe);
    *(struct Trapframe *)tf->regs[29] = tmp_tf;

    if (curenv->env_sig_entry) {
        tf->regs[4] = tf->regs[29];
        tf->regs[5] = signum;
        tf->regs[29] -= sizeof(tf->regs[4]);
        tf->regs[29] -= sizeof(tf->regs[5]);
        tf->cp0_epc = curenv->env_sig_entry;
    } else {
        panic("sig but no user handler registered");
```

```
    }
}
```

异常处理函数的定义

`sig_entry`函数定义在`user/lib/fork.c`中，在这里对信号进行了处理，实现类似于`cow_entry`。

```
static void __attribute__((noreturn)) sig_entry(struct Trapframe *tf, int signum)
{
    struct sigaction signal = env->env_sigaction[signum];
    sigset_t set, oset;
    set.sig = signal.sa_mask.sig | (1 << (signum - 1));
    sigprocmask(SIG_BLOCK, &set, &oset);  // 将要处理信号的屏蔽掩码加入到进程的掩码中。
    //debugf("signum:%d\n", signum);
    if (signum == SIGKILL) {  //对于SIGKILL信号，该信号不可被阻塞，任何对其处理函数进行修改都是无效的，其处理动作只会是结束进程。
        exit();
    }
    else if (signal.sa_handler != 0) {
        signal.sa_handler(signum);
    } else {
        if (signum == SIGINT || signum == SIGILL || signum == SIGKILL || signum == SIGSEGV) {
            exit();
        }
    }
    sigprocmask(SIG_SETMASK, &oset, NULL);  // 恢复掩码
    int r = syscall_set_sig_trapframe(0, tf);
    user_panic("syscall_set_sig_trapframe returned %d", r);
}
```

设置异常处理函数入口地址

最后一个问题，在`do_signal`中，我们最后是跳转到了`curenv->env_sig_entry`，这个地方应该是`sig_entry`函数，那它是什么时候设置的呢。

在`user/lib/fork.c`中，我定义了一个`set_sig_entry`函数来设置入口。

```
int set_sig_entry() {
    try(syscall_set_sig_entry(0, sig_entry));
    try(syscall_set_tlb_mod_entry(0, cow_entry));
    return 0;
}
```

在`user/lib/libos.c`中的`libmian`函数中，我调用这个函数来设置入口，这样就能对每一个新的进程设置`sig_entry`，同时，也需要为进程设置`cow_entry`函数入口地址，防止在后续写时复制的时候出现`panic("TLB Mod but no user handler registered");`异常。

```
void libmain(int argc, char **argv) {
    // set env to point at our env structure in envs[].
    env = &envs[ENVX(syscall_getenvid())];

    set_sig_entry();

    //...
}
```

通过以上的步骤，sigaction的整个处理过程大致就搭建好了，接下来，是针对于几种会影响进程正常运行的信号，需要在内核中进行处理。

需要特殊处理的信号

SIGILL

该信号在进程遇到非法指令时发给自己。处理sigill的方法就是在`kern/genex.S`中，仿照处理其它异常的操作，添加一个`BUILD_HANDLER ri  do_send_sigill`。当出现sigill异常的时候，会跳转到`do_send_sigill`函数进行处理，该函数我们在`kern/traps`中进行实现。

```
// 注意要仿照其它的异常处理函数来写
extern void handle_int(void);
extern void handle_tlb(void);
extern void handle_sys(void);
extern void handle_mod(void);
extern void handle_reserved(void);
extern void handle_ri(void);

void (*exception_handlers[32])(void) = {
    [0 ... 31] = handle_reserved,
#if !defined(LAB) || LAB >= 4
    [1] = handle_mod,
    [8] = handle_sys,
    [10] = handle_ri,
#endi

void do_send_sigill(struct Trapframe *tf) { // 处理方法就是向自己发送SIGILL信号，直接修改env或者用sys_kill都可以。
    curenv->env_sig_recv |= (1 << (4-1));
    schedule(-1);
}
```

SIGSECV

该信号是访问地址出错的时候发给自己。处理方法就是在`kern/tlbex.c`的`passive_alloc`函数中，将对应的`panic("address too low");`改成发送信号`sys_kill(0, SIGSEGV);`

SIGCHLD

该信号在子进程结束时发送给父进程。在`kern/env.c`中的`env_free`函数中加入相应操作即可：

```
if (e->env_parent_id) {
    struct Env *parent;
    envid2env(e->env_parent_id, &parent, 0);
    parent->env_sig_recv |= 1 << (17 - 1); // SIGCHLD
}
```

SIGSYS

该信号在系统调用号出错的时候发送给自己。在`kern/syscall_all.c`中的`do_syscall`中修改相应操作：

```
if (sysno < 0 || sysno >= MAX_SYSNO) {
    tf->regs[2] = -E_NO_SYS;
    curenv->env_sig_recv |= 1 << (31 - 1); // SIGSYS
    schedule(-1);
}
```

总结

挑战性任务的结束也标志着os课程的结束。os的实验和理论课程相辅相成，我在这其中学到了很多，一路走来，os的代码对我来说也从晦涩难懂变得逐渐熟练起来。对于挑战性任务，我花了三四天才完成，虽然debug的时候非常痛苦，但看到程序成功输出了正确的结果之后喜悦之情溢于言表，对于操作系统的认识也上了一个新的台阶。