Sigaction 挑战性任务

本次挑战性任务需要同学们按照给定的要求实现Linux内核中的异步通信`sigaction`,在完成后即可提交代码进行自动化测试,获得挑战性任务分数。

任务要求

- 按照给定要求实现`sigaction`功能。
- 实现文档，**请务必详细地介绍你的实现。**

参考文档

- 文档1
- 文档2

sigaction简介

sigaction`是 Unix与类Unix操作系统中进程间或者内核与进程间的一种**异步通信**，用来提醒一个进程某一信号已经发生。

当一个信号被发送给一个进程时, 内核会中断进程的正常控制流，转而**执行与该信号相关的用户态处理函数**进行处理，在执行该处理函数前，会将该信号所设置的信号屏蔽集加入到进程的信号屏蔽集中，在执行完该用户态处理函数后，又会将恢复原来的信号屏蔽集;本次实验只需实现`[1,32]`普通信号，无需考虑`[33,64]`的实时信号。

任务描述

请将要求实现的函数的声明与结构体定义放在适当文件中，使得用户态程序通过`#include <lib.h>`即可调用要求实现的信号相关的函数，使用信号相关的结构体和宏定义等。

sigaction 结构体

`sigaction`结构体用于设置所需要处理的信号集及其对应的处理函数(在后续介绍的`sigaction`信号注册函数中被使用)。

`sigset_t`使用32位表示MOS所需要处理的[1,32]信号掩码，对应位为1表示阻塞，为0表示未被阻塞(使用后续介绍的信号掩码处理函数进行相应信号位的设置，用于`sigaction`结构体中表示处理该`sigaction`结构体对应的信号被处理时，需要被屏蔽的信号集)。

`sigset_t`与`sigaction`结构体定义如下:

```
typedef struct sigset_t {
    uint32_t sig;
} sigset_t;

struct sigaction {
    void     (*sa_handler)(int);
    sigset_t   sa_mask;
};
```

下面是对`sigaction`成员变量的具体说明:

- `sa_handler`: 信号的处理函数，当未屏蔽信号到达并且"处理时机"合适时，进程就会执行该函数(除去`SIGKILL`信号不能设置其他处理函数)。
- `sa_mask`: 存放了对应信号处理函数被执行时需要被阻塞的信号掩码，[1,32]对应位为1表示处理信号时需要屏蔽。

SIGNUM编号

如之前所说，这里仅考虑`[1,32]`内的signum编号,下面是你需要实现的信号:

信号名称	编号	描述	默认处理动作
SIGINT	2	中断信号	停止进程
SIGILL	4	非法指令	停止进程
SIGKILL	9	停止进程信号	强制停止该进程，不可被阻塞
SIGSEGV	11	访问地址错误，当访问[0, 0x003f_e000)内地址时	停止进程
SIGCHLD	17	子进程终止信号	忽略
SIGSYS	31	系统调用号未定义	忽略

其余`[1,32]`内的`signum`编号的默认处理动作为**忽略。**

> 注: - 不同普通信号优先级不同，编号更小的信号拥有更高的优先级，即当有两个不同的信号需要处理时，需要先处理编号较小的信号 - 同一普通信号在进程中最多只存在一个，也即如果有多个同样编号信号发送至某一进程，只取其中一个,**当一个信号被执行时，需要添加同类型信号的屏蔽，在结束执行后再恢复成原来的屏蔽集状态。** - **信号的被打断只与此时信号的屏蔽集有关**，优先级只会影响当有多种信号信号时，所需要执行信号的选择。 - 对于`SIGSEGV`信号，在原MOS中会进行panic，你需要取消该设置，改为发送SIGSEGV信号。 - 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。**
>
> 如果存在疑问可在讨论区提出。

需要实现的函数

下面的所有函数都位于用户态中，其中的某些函数的具体功能可能需要由系统调用实现。

信号注册函数

信号注册函数如下:

```
int sigaction(int signum, const struct sigaction *newact, struct sigaction
*oldact);
```

- `signum`:需要设置的信号编号。如之前所说你只需考虑`signum`小于或等于`32`的情况，当收到编号大于32的信号时直接返回异常码`-1`即可。
- `newact`:为`signum`设置的`sigaction`结构体,如果`newact`不为空。

- `oldact`: 将该信号之前的`sigaction`结构体其内容填充到oldact中(如果oldact不为空)。

信号发送函数

你可以使用`kill`函数向进程发送信号，`kill`函数的声明如下:

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

- 当`envid`为`0`时，代表向自身发送信号，否则代表向`envid`进程发送信号。
- 当`envid`对应进程不存在，或者`sig`不符合定义范围时，返回异常码`-1`。

> 需要注意一些信号通常并不通过`kill`函数发出，而是由内核发出，比如`SIGCHLD`,`SIGILL`,`SIGSYS`,`SIGSEGV`信号，在本次实验中，不需要考虑通过`kill`函数发出这些信号的情况。

信号集处理函数

信号集处理函数主要为对信号掩码的不同操作，下面是你需要实现的信号掩码处理函数:

> 注: 对于如下函数，如果参数非法，参考给定文档实现。

```
int sigemptyset(sigset_t *__set);
// 清空参数中的__set掩码，初始化信号集以排除所有信号。这意味着__set将不包含任何信号。(清0)

int sigfillset(sigset_t *__set);
// 将参数中的__set掩码填满，使其包含所有已定义的信号。这意味着__set将包括所有信号。(全为1)

int sigaddset(sigset_t *__set, int __signo);
// 向__set信号集中添加一个信号__signo。如果操作成功，__set将包含该信号。(置位为1)

int sigdelset(sigset_t *__set, int __signo);
// 从__set信号集中删除一个信号__signo。如果操作成功，__set将不再包含该信号。(置位为0)

int sigismember(const sigset_t *__set, int __signo);
// 检查信号__signo是否是__set信号集的成员。如果是，返回1；如果不是，返回0。

int sigisemptyset(const sigset_t *__set);
// 检查信号集__set是否为空。如果为空，返回1；如果不为空，返回0。

int sigandset(sigset_t *__set, const sigset_t *__left, const sigset_t *__right);
// 计算两个信号集__left和__right的交集，并将结果存储在__set中。

int sigorset(sigset_t *__set, const sigset_t *__left, const sigset_t *__right);
// 计算两个信号集__left和__right的并集，并将结果存储在__set中。

int sigprocmask(int __how, const sigset_t * __set, sigset_t * __oset);
// 根据__how的值更改当前进程的信号屏蔽字。__set是要应用的新掩码，__oset（如果非NULL）则保存旧的信号屏蔽字。__how可以是SIG_BLOCK（添加__set到当前掩码）、SIG_UNBLOCK（从当前掩码中移除__set）、或SIG_SETMASK（设置当前掩码为__set）。

int sigpending(sigset_t *__set);
```

```
// 获取当前被阻塞且未处理的信号集，并将其存储在__set中。
```

测试案例

SIGINT测试

```
#include <lib.h>

void sigint_handler(int sig) {
    debugf("capture SIGINT.\n");
    exit();
}

int main() {
    struct sigaction sa;
    sa.sa_handler = sigint_handler;
    sigemptyset(&sa.sa_mask);
    sigaction(SIGINT, &sa, NULL);
    debugf("sending SIGINT to myself\n");
    kill(0,SIGINT);
    debugf("If you see this on your screen, it means that the signal is not
handled correctly.\n");
    while(1);
    return 0;
}
```

该进程的控制条输出为:

```
sending SIGINT to myself
capture SIGINT.
#后续为进程结束输出内容，无需考虑
```

SIGILL测试

```
#include <lib.h>

void sigill_handler(int sig) {
    debugf("capture SIGILL signal.\n");
    exit();
}

int main() {
    struct sigaction sa;
    sa.sa_handler = sigill_handler;
    sigemptyset(&sa.sa_mask);
    sigaction(SIGILL, &sa, NULL);

    asm("\tmove $t0,$sp\r\n"
        "\tjr $t0\r\n");
    debugf("Hello, world!\n");
    return 0;
}
```

该进程的控制条输出为:

```
capture SIGILL signal.
#后续为进程结束内核输出内容，无需考虑
```

SIGSEGV测试

```
#include <lib.h>

void sigsegv_handler(int sig) {
    debugf("capture SIGSEGV signal.\n");
    exit();
}

int main() {
    struct sigaction sa;
    sa.sa_handler = sigsegv_handler;
    sigemptyset(&sa.sa_mask);
    sigaction(SIGSEGV, &sa, NULL);
    int* ptr = (int*)0x0;
    debugf("Accessing invalid memory address...\n");
    int val = *ptr;
    debugf("If you see this on your screen, it means that the signal is not
handled correctly.\n");
    return 0;
}
```

该进程输出为:

```
Accessing invalid memory address...
capture SIGSEGV signal.
#后续为进程结束内核输出内容，无需考虑
```

SIGCHLD测试

```
#include <lib.h>

void sigchld_handler(int sig) {
    debugf("capture SIGCHLD signal.\n");
    exit();
}

int main() {
    struct sigaction sa;
    sa.sa_handler = sigchld_handler;
    sigemptyset(&sa.sa_mask);
    sigaction(SIGCHLD, &sa, NULL);
    if (fork() == 0) {
        exit();
    }
    while (1);
   return 0;
```

```
}
```

该进程输出为：

```
capture SIGCHLD signal.
# 后续为进程结束内核输出内容，无需考虑
```

SIGSYS测试

```
#include <lib.h>

void sigsys_handler(int sig) {
    debugf("capture SIGSYS signal.\n");
    exit();
}

int main() {
    struct sigaction sa;
    sa.sa_handler = sigsys_handler;
    sigemptyset(&sa.sa_mask);
    sigaction(SIGSYS, &sa, NULL);

    asm("\t li $a0,0x2233\r\n"
        "\t syscall \r\n");

    debugf("If you see this on your screen, it means that the signal is not
handled correctly.\n");
    return 0;
}
```

该进程输出为：

```
capture SIGSYS signal.
# 后续为进程结束内核输出内容，无需考虑
```