考试说明

考试时间：14:00 ~ 16:30

本次题面以 pdf 和 html 两种格式下发，需要复制代码的同学请使用 html 格式的题面。

lab4-2-exam

创建并切换分支

```
git checkout lab4
git add .
git commit --allow-empty -m "save my lab4"
git checkout -b lab4-2-exam
```

题目描述

请你在 `user/fork.c` 中实现下面这个用户函数，并在 `user/lib.h` 中添加相应的声明，使用户程序能使用该函数：

```
// user/lib.h
int make_shared(void *va);
```

- 该函数将当前进程中虚拟地址 va 所属的虚拟页标记为**共享页**，并返回其映射到的物理页的物理地址。
- 在该进程后续执行 fork 时，其共享页应与子进程共享，使得两个进程的地址空间中该页映射到同一个物理页。
- fork **不应**对共享页进行 COW 保护。若父进程或子进程修改了共享页中的数据，随后另一进程读取该页时也会读取到修改后的数据。

实现要求

- 若当前进程的页表中不存在该虚拟页，该函数应首先分配一页物理内存，并将该虚拟页映射到新分配的物理页，使当前进程能够**读写**该虚拟页。若无法分配新的物理页，该函数应返回 -1 表示失败。
- 若 va 不在用户空间中（大于或等于 UTOP），或者当前进程的页表中已存在该虚拟页，但进程对其没有写入权限，则该函数应返回 -1 表示失败，不产生任何影响。
- 除了失败的情况，该函数都应返回 va 所在的虚拟页所映射到的物理页的物理地址。
- 若虚拟页 va 已经为共享页，该函数仍应成功，直接返回对应的物理地址。
- 评测保证调用 `make_shared` 之前，虚拟页 va 没有被 COW 保护。

请注意：

- 进程的共享页作为进程的状态，在执行 fork 后创建的子进程中仍应保持。即：若虚拟页 va 是父进程中的共享页，则在子进程中 va 仍然是子进程的共享页。若子进程再执行一次 fork，父进程、子进程、子进程的子进程都能通过 va 共享同一个物理页。
- 作为参数传入 `make_shared` 的 va 不一定是页对齐的，但**返回的物理地址一定是页对齐的。**

实现提示

- 在用户函数中，你可以读取当前进程的页表，也可以调用已有的 `syscall_` 开头的系统调用函数，但不能直接调用内核中的函数。
- 你可以回顾指导书中 `duppage` 函数的相关说明，利用其中已实现的机制完成页面共享。

本地测试

测试程序

```
// user/shmtest.c
#include "lib.h"

void umain() {
    volatile u_int *a = (volatile u_int *) 0x23333334;
    make_shared((void *) a);
    *a = 233;
    if (fork() == 0) {
        u_int ch = syscall_getenvid();
        *a = ch;
        while (*a == ch)
            syscall_yield();
        writef("parent is %x\n", *a);
    } else {
        while (*a == 233)
            syscall_yield();
        writef("child is %x\n", *a);
        *a = syscall_getenvid();
    }
}
```

测试流程

1. 将以上的用户程序保存为 `user/shmtest.c`
2. 修改 `user/Makefile`，在构建目标 `all:` 后添加 `shmtest.x` 和 `shmtest.b`
3. 将 `init/init.c` 中的 `mips_init()` 函数修改为以下内容：

```
void mips_init() {
    printf("init.c:\tmips_init() is called\n");
    mips_detect_memory();
    mips_vm_init();
    page_init();
    env_init();

    ENV_CREATE(user_shmtest);

    trap_init();
    kclock_init();
    while(1);
    panic("init.c:\tend of mips_init() reached!");
}
```

4. 构建并运行：

```
make clean && make
/OSLAB/gxemul -E testmips -C R3000 -M 64 gxemul/vmlinux
```

参考输出

```
child is c01
parent is 400
```

此处略去了输出中的空行，以及 `pageout`、`main is start`、`i am killed` 等内核输出。

提交评测

```
git add .
git commit -m "finish exam"
git push origin lab4-2-exam:lab4-2-exam
```

lab4-2-Extra

创建并切换分支

```
git checkout lab4
git checkout -b lab4-2-Extra
```

题目背景

> 信号（英语：Signals）是Unix、类Unix以及其他POSIX兼容的操作系统中进程间通讯的一种有限制的方式。它是一种异步的通知机制，用来提醒进程一个事件已经发生。当一个信号发送给一个进程，操作系统会打断进程正常的控制流程，此时，任何非原子操作都将被打断。如果进程注册了信号的处理函数，那么它将被执行，否则就执行默认的处理函数。
>
> 信号的机制类似于硬件中断（异常），不同之处在于中断由处理器发出并由内核处理，而信号由内核发出并由用户程序处理。除了进程通过系统调用向另一进程（或它自身）发出的信号，内核还可以将发生的中断通过信号通知给引发中断的进程。如果说系统调用是一种用户程序通知内核的机制，那么信号就是内核通知用户程序的机制。
>
> —— Unix信号，维基百科（有删改）

以上材料仅用于帮助理解概念，具体的实现要求请以下文为准。

题目描述

请你在 `user/ipc.c` 中实现以下两个用户函数（以及可能需要的辅助函数和变量），并在 `user/lib.h` 中相应声明：

发送信号

```
// user/lib.h
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。

注册信号处理函数

```
// user/lib.h
void signal(int sig, void (*handler)(int));
```

该函数为当前进程注册一个信号处理函数。注册完成后，每当该进程收到一个编号为 `sig` 的信号，该进程将打断原有的流程，开始执行信号处理函数 `handler`，同时信号编号 `sig` 会作为 `handler` 的参数传入。

若 `handler` 为空指针，该函数取消已为信号 `sig` 注册的处理函数，如同未曾注册过该信号的处理函数。

信号列表

本题中定义的信号有且仅有以下几种：

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

如上表所示，除了用户程序通过 `kill` 发送的 SIGTERM 信号，你还需要修改内核中原有的实现，在以上的后两种情况发生时，向进程发送相应的信号。

实现说明

- 进程注册的信号处理函数应视作进程状态的一部分，即进程调用 `fork()` 后创建的子进程中仍应注册有 `fork()` 前父进程注册的处理函数。
- 信号处理函数返回后，进程应继续执行被信号打断的流程。当然，信号处理函数也可能不返回，如调用 `exit()` 使进程退出。
- 若收到信号的进程没有为该信号注册处理函数，其行为须与上表中的默认处理动作一致。
- 若收到信号的进程注册了该信号的处理函数，上表中的默认处理动作不再生效。即，如果处理函数正常返回，即使该信号的默认处理动作是退出，进程也会回到原流程继续执行。
- 如果子进程退出时父进程也已退出，则不会产生 SIGCHLD 信号。
- 本题中的“子进程”仅指 `fork()` 产生的直接子进程，即在进程 A 的子进程 B 的子进程 C 退出时，只有进程 B 可能收到 SIGCHLD 信号，进程 A 不会收到。
- SIGSEGV 信号的处理函数返回后，进程应恢复到用户程序中进行非法访存的指令处，这意味着该处理函数可能被反复执行。
- 进程使用 `kill()` 向自身发送的信号也应是同步处理的，即 `kill()` 返回时该信号的处理函数（或默认动作）应已执行完毕。
- 除上述的 SIGSEGV 和向自身发送信号的情况外，信号的收发可能是异步的，即收到信号的进程在哪条指令被打断可能依赖于调度策略，评测对此没有特定要求。

评测保证：

- 参数中传入的 `sig` 均为本题的信号列表中出现的编号。对于 `kill()`，`sig` 只会为 15。
- 向目标进程发送信号时，目标进程尚未退出，且不在前一个信号的处理过程中。

    - 这包括调用 `kill()` 的时刻，以及触发操作系统信号的事件发生的时刻。此时要么未向目标进程发送过任何信号，要么目标进程已从向它发送的前一个信号的处理函数中返回到原流程。
    - 这意味着你实现的信号处理不需要是可重入的。
- 调用 `signal()` 时，`handler` 要么为空指针，要么指向用户空间中定义的函数。
- 除了信号列表中对 SIGSEGV 的描述，用户程序不会进行其他的非法内存访问。
- 每次评测创建的总进程数量不超过 20。

实现提示

- 为使进程进入信号处理函数以及从信号处理过程返回，你可能需要引入新的系统调用。
- 你可以参照实现 COW 机制时用户程序注册异常处理函数的过程，将信号处理过程在用户空间中的入口告知内核。
- 为使进程从原流程跳转到信号处理过程，你可以参照 `sys_env_alloc` 的实现，在内核中修改进程保存的上下文（Trapframe）。
    - 根据目标进程为当前发起系统调用的进程还是其他进程，为该进程保存上下文的地址可能不同，需要修改的寄存器也可能不同。
    - 为了向信号处理函数传递参数，你可能需要修改上下文中保存的通用寄存器。
    - 为使进程能返回到原流程，可能需要将信号产生前进程原本的上下文保存到内核或用户空间中。
- 根据信号编号分发信号的过程可以参考内核中异常和系统调用的分发，使用向量表来实现。
- 你可以定义一个用户空间中的信号处理入口函数，进行信号的分发，并在处理函数返回后恢复原本的上下文。或者，也可以在内核中直接分发。
- 如果你需要在系统调用中恢复上下文，请注意从系统调用返回时会被覆盖的通用寄存器。
- 在原实现中，访问非法的内存地址可能导致内核输出 TOO LOW 并 panic，因此实现 SIGSEGV 时需要修改此行为。
- 为了保证内存映射的正确性，请避免修改结构体 `struct Env` 的定义。你可以使用 `ENVX` 宏等方式获取进程控制块在 `envs` 中的下标，并使用独立的静态数组为每个进程在内核中维护必要的信息。
- 你可能还需要阅读或修改以下文件：
    - `user/libos.c`
    - `lib/env.c`
    - `lib/genex.S` 和 `mm/pmap.c`
    - 关于系统调用的文件：
        - `include/unistd.h`
        - `lib/syscall.S`
        - `lib/syscall_all.c`
        - `user/lib.h`
        - `user/syscall_lib.c`

以上提示仅供参考，在满足题目要求，正确实现三种信号的前提下，你可以不遵循以上提及的实现细节。

评测说明

评测时，以下文件可能会被替换为标准版本：

- `Makefile`
- `user/Makefile`
- `init/init.c`

评测由 5 个 Part 组成，各个 Part 独立评测，分值均为 20 分。每个 Part 使用的用户程序满足相应的约束：

- Part 1：信号只涉及 SIGTERM，不会调用 `signal()` 函数。
- Part 2：信号只涉及 SIGTERM，使用 `signal()` 注册的所有信号处理函数最后都会调用 `exit()` 函数退出进程。
- Part 3：信号只涉及 SIGTERM。
- Part 4：信号只涉及 SIGTERM 和 SIGSEGV。
- Part 5：信号只涉及 SIGTERM 和 SIGCHLD。

本地测试

```
// user/sigtest.c
#include "lib.h"

int flag = 0;

void handle(int sig) {
    flag += 1;
    if (sig == 15)
        writef("%x: kill me baby!\n", syscall_getenvid());
    else if (sig == 18)
        writef("%x: child exited\n", syscall_getenvid());
}

void handle_segv(int sig) {
    writef("%x: segmentation fault\n", syscall_getenvid());
    exit();
}

void umain() {
    signal(15, handle);
    signal(18, handle);
    signal(11, handle_segv);

    int parent = syscall_getenvid();
    int ch = fork();
    if (ch == 0) {
        kill(parent, 15);
        ipc_recv(0, 0, 0);
        *(int *) NULL = 233;
    } else {
        while (!flag)
            syscall_yield();
        ipc_send(ch, 0, 0, 0);
        while (flag < 2)
            syscall_yield();
    }
```

```
    signal(15, 0);
    kill(0, 15);
    writef("this is unreachable");
}
```

1. 将以上程序保存为 `user/sigtest.c`
2. 在 `user/Makefile` 中添加构建目标 `sigtest.b sigtest.x`
3. 在 `init/init.c` 中添加 `ENV_CREATE(user_sigtest)` 并删除其他 `ENV_CREATE`
4. 构建并运行

参考输出

```
400: kill me baby!
[ warning: LOW reference: vaddr=0x00000000, exception TLBS, pc=0x004001ec <(no
symbol)> ]
c01: segmentation fault
400: child exited
```

此处略去了空行，以及 pageout、`i am killed` 等内核输出。gxemul 所产生 warning 中的 PC 也是不确定的。

提交评测

```
git add .
git commit -m "finish Extra"
git push origin lab4-2-Extra:lab4-2-Extra
```