任务内容

1. `syscall_clone` 支持 `CLONE_FS` 和 `CLONE_SIGHAND`
    i. `CLONE_FS` 需要做到父子进程之间工作目录共享，还需要后续对于 `chdir`， `umask` 的修改是共享的
    ii. `CLONE_SIGHAND`：继承信号处理函数，但是不继承掩码和等待信号集

功能支持

CLONE_FS (since Linux 2.0)
If **CLONE_FS** is set, the caller and the child process share the same filesystem information. This includes the root of the filesystem, the current working directory, and the umask. Any call to chroot(2), chdir(2), or umask(2) performed by the calling process or the child process also affects the other process.

If **CLONE_FS** is not set, the child process works on a copy of the filesystem information of the calling process at the time of the clone call. Calls to chroot(2), chdir(2), or umask(2) performed later by one of the processes do not affect the other process.

CLONE_FILES(since Linux 2.0)
如果设置了**CLONE_FILES**，则调用进程和子进程共享相同的文件描述符表。由调用进程或子进程创建的任何文件描述符在其他进程中也有效。同样，如果其中一个进程关闭文件描述符或更改其关联标志(使用**fcntl**(2)F_SETFD操作)，则另一个进程也会受到影响。如果共享文件描述符表的进程调用**execve**(2)，则其文件描述符表将被复制(未共享)。
如果未设置**CLONE_FILES**，则子进程将继承克隆调用时在调用进程中打开的所有文件描述符的副本。由调用进程或子进程执行的打开或关闭文件描述符或更改文件描述符标志的后续操作不会影响其他进程。但是请注意，子级中重复的文件描述符与调用过程中的相应文件描述符引用相同的打开文件描述，因此共享文件偏移量和文件状态标志(请参见**open**(2))。

- 共享与文件系统相关的信息：CLONE_FS

如果指定了该标志，那么父子进程将共享与文件系统相关的信息：权限掩码、根目录以及当前工作目录。也就是说无论在哪个进程中调用umask()、chdir()或者chroot()，都将影响到另一个进程。

```
772     /// 获取当前进程的工作目录
773     pub fn get_cwd(&self) -> String {
774         self.fd_manager.cwd.lock().clone()
775     }
776
777     /// Set the current working directory of the process
778     pub fn set_cwd(&self, cwd: String) {
779         *self.fd_manager.cwd.lock() = cwd;
780     }
781 } impl Process
```

修改代码

axprocess/src/process.rs

```
            let mut cwd_src = String::from("./").into();
            let mut mask_src = Arc::new(AtomicI32::new(0o022));
            if clone_flags.contains(CloneFlags::CLONE_FS) {
                cwd_src = Arc::clone(&self.fd_manager.cwd.lock());
                mask_src = Arc::clone(&self.fd_manager.umask);
            }
            // 若创建的是进程，那么需要新建进程
            // 由于地址空间是复制的，所以堆底的地址也一定相同
            let new_process = Arc::new(Process::new(
                process_id,
                self.get_stack_limit(),
                parent_id,
                new_memory_set,
                self.get_heap_bottom(),
                self.fd_manager.fd_table.lock().clone(),
                cwd_src,
                mask_src,
            ));
```

axprocess/src/process.rs

```
/// 获取当前进程的工作目录
pub fn get_cwd(&self) -> String {
    self.fd_manager.cwd.lock().clone().to_string()
}

/// Set the current working directory of the process
pub fn set_cwd(&self, cwd: String) {
    *self.fd_manager.cwd.lock() = cwd.into();
}
```

axprocess/src/fd_manager.rs

```
pub struct FdManager {
    /// 保存文件描述符的数组
    pub fd_table: Mutex<Vec<Option<Arc<dyn FileIO>>>>,
    /// 保存文件描述符的数组的最大长度
    pub limit: AtomicU64,
    /// 创建文件时的mode的掩码
    pub umask: Arc<AtomicI32>,
    pub cwd: Mutex<Arc<String>>,
}

impl FdManager {
    pub fn new(fd_table: Vec<Option<Arc<dyn FileIO>>>, cwd_src: Arc<String>,
mask_src: Arc<AtomicI32>, limit: usize) -> Self {
        Self {
            fd_table: Mutex::new(fd_table),
            limit: AtomicU64::new(limit as u64),
            umask: mask_src,
            cwd: Mutex::new(cwd_src),
        }
```

```
    }
```

测试用例

测试current_dir

```
#define _GNU_SOURCE
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sched.h>
#include <sys/mount.h>
#include <sys/types.h>
#include <sys/wait.h>

#define STACK_SIZE (1024 * 1024) // 1MB堆栈大小

// 子进程函数
int child_function(void *arg) {
    char *new_dir = "/tmp";

    // 打印子进程的当前工作路径
    char cwd[1024];
    if (getcwd(cwd, sizeof(cwd)) != NULL) {
        printf("Child process current working directory: %s\n", cwd);
    } else {
        perror("getcwd() error");
        return EXIT_FAILURE;
    }

    // 更改子进程的当前工作路径
    if (chdir(new_dir) == -1) {
        perror("chdir() error");
        return EXIT_FAILURE;
    }

    printf("Child process changed working directory to: %s\n", new_dir);

    // 再次打印子进程的当前工作路径
    if (getcwd(cwd, sizeof(cwd)) != NULL) {
        printf("Child process current working directory after change: %s\n",
cwd);
    } else {
        perror("getcwd() error");
        return EXIT_FAILURE;
    }

    return EXIT_SUCCESS;
}

int main() {
    int status;
```

```
    // Allocate stack for child process
    void *stack = malloc(STACK_SIZE);
    if (stack == NULL) {
        perror("malloc");
        exit(EXIT_FAILURE);
    }

    // 父进程当前工作路径
    char cwd[1024];
    if (getcwd(cwd, sizeof(cwd)) != NULL) {
        printf("Parent process current working directory: %s\n", cwd);
    } else {
        perror("getcwd() error");
        return EXIT_FAILURE;
    }

    // 使用clone创建子进程，并共享文件系统信息(CLONE_FS)
    pid_t pid = clone(child_function, (char *) stack + STACK_SIZE,
                    CLONE_VM | SIGCHLD|CLONE_FS | SIGCHLD, NULL);
    if (pid == -1) {
        perror("clone() error");
        return EXIT_FAILURE;
    }

    // 等待子进程退出
    if (waitpid(pid, &status, 0) == -1) {
        perror("waitpid() error");
        return EXIT_FAILURE;
    }

    // 打印父进程当前工作路径
    if (getcwd(cwd, sizeof(cwd)) != NULL) {
        printf("Parent process current working directory: %s\n", cwd);
    } else {
        perror("getcwd() error");
        return EXIT_FAILURE;
    }

    // 打印子进程退出状态
    if (WIFEXITED(status)) {
        printf("Child process exited with status %d\n", WEXITSTATUS(status));
    } else {
        printf("Child process did not exit normally\n");
    }

    // 释放堆栈空间
    free(stack);

    return EXIT_SUCCESS;
}
```

当前starry的工作目录有问题，需要将syscall_chidir的代码进行修改后测试

```
pub fn syscall_chdir(args: [usize; 6]) -> SyscallResult {
```

```
    let path = args[0] as *const u8;
    // 从path中读取字符串
    let path = if let Some(path) = deal_path(AT_FDCWD, Some(path), true) {
        path
    } else {
        return Err(SyscallError::EINVAL);
    };
    debug!("Into syscall_chdir. path: {:?}", path.path());
    match axfs::api::set_current_dir(path.path()) {
        Ok(_) => Ok(0),
        Err(_) => Err(SyscallError::EINVAL),
    }
}
```

```
Parent process current working directory: /
Child process current working directory: /
Child process changed working directory to: /tmp
Child process current working directory after change: /tmp/
Parent process current working directory: /tmp/
Child process exited with status 0
/tmp/ #
```

测试umask

```
#define _GNU_SOURCE
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sched.h>
#include <sys/mount.h>
#include <sys/types.h>
#include <sys/wait.h>

#define STACK_SIZE (1024 * 1024) // 1MB堆栈大小

void print_umask(const char *label) {
    mode_t current_umask = umask(0);  // 获取当前的 umask 值
    umask(current_umask);             // 恢复原来的 umask 值
    printf("%s: %04o\n", label, current_umask);

}

// 子进程函数
int child_function(void *arg) {
    print_umask("Child init umask");
    umask(027);
    print_umask("Child current umask");
    return EXIT_SUCCESS;
}

int main() {
    int status;

    // Allocate stack for child process
    void *stack = malloc(STACK_SIZE);
```

```
    if (stack == NULL) {
        perror("malloc");
        exit(EXIT_FAILURE);
    }

    print_umask("Parent init umask");
    // 使用clone创建子进程，并共享文件系统信息(CLONE_FS)
    pid_t pid = clone(child_function, (char *) stack + STACK_SIZE,
                    CLONE_VM | SIGCHLD|CLONE_FS | SIGCHLD, NULL);
    if (pid == -1) {
        perror("clone() error");
        return EXIT_FAILURE;
    }

    // 等待子进程退出
    if (waitpid(pid, &status, 0) == -1) {
        perror("waitpid() error");
        return EXIT_FAILURE;
    }

    print_umask("Parent current umask");
    // 打印子进程退出状态
    if (WIFEXITED(status)) {
        printf("Child process exited with status %d\n", WEXITSTATUS(status));
    } else {
        printf("Child process did not exit normally\n");
    }

    // 释放堆栈空间
    free(stack);

    return EXIT_SUCCESS;
}
```

```
Parent init umask: 0022
Child init umask: 0022
Child current umask: 0027
Parent current umask: 0027
Child process exited with status 0
/ #
```

其他说明

CLONE_SIGHAND已实现

测试

```
#define _GNU_SOURCE
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
```

```c
#include <sys/types.h>
#include <sys/wait.h>
#include <sys/mount.h>
#include <sched.h>
#include <signal.h>

#define STACK_SIZE (1024 * 1024) // 1MB

// 子进程函数
int child_func(void *arg) {


    printf("Child process: PID=%ld\n", (long)getpid());

    // 子进程循环等待信号
    while (1) {
        sleep(1); // 子进程持续运行，以便观察
    }

    return 0;
}

int main() {
    char *stack;
    char *stackTop;
    pid_t pid;

    // 为子进程分配栈空间
    stack = (char *)malloc(STACK_SIZE);
    if (stack == NULL) {
        perror("malloc");
        exit(EXIT_FAILURE);
    }
    stackTop = stack + STACK_SIZE;  // 栈向下增长
    printf("create pid.\n");

    // 使用clone创建子进程，并设置CLONE_SIGHAND标志
    pid = clone(child_func, stackTop,  CLONE_SIGHAND | SIGCHLD | CLONE_VM,
NULL);
    if (pid == -1) {
        perror("Parent: clone");
        exit(EXIT_FAILURE);
    }

    // 等待子进程启动
    sleep(1);

    // 定义信号处理函数
    struct sigaction act;
    act.sa_handler = SIG_IGN; // 忽略信号
    sigemptyset(&act.sa_mask);
    act.sa_flags = 0;

    // 设置父进程的信号处理器
    if (sigaction(SIGUSR1, &act, NULL) == -1) {
```

```
        perror("Parent: sigaction");
        exit(EXIT_FAILURE);
    }

    printf("Parent process: PID=%ld\n", (long)getpid());

    // 发送信号给子进程
    printf("Sending SIGUSR1 to child process\n");
    if (kill(pid, SIGUSR1) == -1) {
        perror("Parent: kill");
        exit(EXIT_FAILURE);
    }

    // 等待子进程退出
    int status;
    waitpid(pid, &status, 0);

    // 释放栈空间
    free(stack);

    // 检查子进程退出状态
    if (WIFEXITED(status)) {
        printf("Child exited with status %d\n", WEXITSTATUS(status));
    } else if (WIFSIGNALED(status)) {
        printf("Child terminated by signal %d\n", WTERMSIG(status));
    } else {
        printf("Unexpected status: %d\n", status);
    }

    return 0;
}
```

父进程设置信号处理方式为SIG_IGN，由于设置了CLONE_SIGHAND标志，子进程也同步修改为SIG_IGN

然后父进程给子进程发送关闭信号，子进程忽略掉该信号，因此僵死在等待状态中