Lab-6

1. 原理：使用条件变量之后，分别从线程和操作系统的角度而言读取字符的系统调用是阻塞的还是非阻塞的？

   对于线程而言，是阻塞的，因为在等待有效输入之前线程都会暂停。但对于操作系统而言，等待输入的时间完全分配给了其他线程，所以对于操作系统来说是非阻塞的。

2. 设计：如果要让用户线程能够使用 `Vec` 等，需要做哪些工作？如果要让用户线程能够使用大于其栈大小的动态分配空间，需要做哪些工作？

   使用Vec就意味着使用堆内存，就要向之前内核中做的一样，搞一个#[global_allocator]

   要实现动态分配内存的系统调用，像linux的malloc。

3. 实验：实现 `get_tid` 系统调用，使得用户线程可以获取自身的线程 ID。

   在内核syscall.rs中实现系统调用，然后在user中将sbi_call封装为相应的syscall函数，然后用户程序中调用即可

```
MEDELEG : 0x000000000000b109
PMP0    : 0x0000000080000000-0x000000008001ffff (A)
PMP1    : 0x0000000000000000-0xffffffffffffffff (A,R,W,X)
mod memory initialized
mod interrupt initialized
mod driver initialized
.
..
hello_world
notebook
mod fs initialized
Hello world from user mode program! id = 1
thread 1 exit with code 0
src/process/processor.rs:87: 'all threads terminated, shutting down'

# root @ bdb20918c35c in /home/rCore/rCore-Tutorial/os on git:lab-4 x [14:31:57
$
```

4. 实验：基于你在实验四（上）的实践，实现 `sys_fork` 系统调用。该系统调用复制一个进程，并为父进程返回 1（目前没有引入进程 ID，也可以自行补充为进程 ID），而为子进程返回 0。

   相比于实验四，你可能需要额外注意文件描述符的复制。

   要做的：

   1. 改造Process，添加id，使用Arc包裹inner

2. 实现线程的fork函数（抄lab4）和进程的fork函数，进程的fork函数只需要修改下id，clone下inner就可以，这样两个线程就共享了页表和文件描述符。
3. 实现fork系统调用，fork进程，fork线程并使用新fork的进程。设置子线程context中的x10为0，返回子进程的id

由于当前的进程和线程实际是用来分离CPU资源和其他资源的，所以是一对一关系，如果以后进程和线程是一对多关系，这种实现就不可行了。一对多关系下fork函数应当只创建调用fork函数的线程。

```
PMP1    : 0x0000000000000000-0xffffffffffffffff (A,R,W,X)
mod memory initialized
mod interrupt initialized
mod driver initialized
.
..
hello_world
notebook
mod fs initialized
我是父进程
thread 1 exit with code 0
我是子进程
thread 2 exit with code 0
src/process/processor.rs:87: 'all threads terminated, shutting down'

# root @ bdb20918c35c in /home/rCore/rCore-Tutorial/os on git:lab-4 x [15:53:37]
```

5. 实验：将一个文件打包进用户镜像，并让一个用户进程读取它并打印其内容。需要实现 `sys_open`，将文件描述符加入进程的 `descriptors` 中并返回，然后通过 `sys_read` 来读取。
    1. 编写sys_open系统调用。由于sys_read是通过Vec的序号读文件的，所以sys_open应当返回Vec的序号。该函数只需要调用ROOT_INODE的find方法查找文件并添加到进程的文件描述符Vec中就可以了。
    2. 编写syscall_handler，传来的参数是指针和长度，所以要用from_raw_parts函数转为string，但是由于后面还会用到context，所以暂时不能让rust回收String，因此又调用into_raw_parts，延长args[0]和args[1]生命周期。

        ```
        SYS_OPEN => {
                    let s = unsafe { String::from_raw_parts(args[0] as *mut u8, args[1], args[1]) };
                    let ret = sys_open(s.as_str());
                    //不能让String被回收
                    s.into_raw_parts();
                    ret
                },
        ```

    3. 最后在user 文件夹的build/disk中添加一个测试文件，并在hello_world程序中调用封装的系统调用

```
PMP0    : 0x0000000080000000-0x000000008001ffff (A)
PMP1    : 0x0000000000000000-0xffffffffffffffff (A,R,W,X)
mod memory initialized
mod interrupt initialized
mod driver initialized
.
..
hello_world
notebook
test_file
mod fs initialized
共 11 个字符: "hello world"
thread 1 exit with code 0
src/process/processor.rs:87: 'all threads terminated, shutting down'

# root @ bdb20918c35c in /home/rCore/rCore-Tutorial/os on git:lab-4 x [17:14:33]
```

6. 挑战实验：实现 `sys_pipe`，为进程添加并返回两个文件描述符，分别为一个管道的读和写端。用户进程调用完 `sys_pipe` 后调用 `sys_fork`，父进程写入管道，子进程可以读取。读取时尽量避免忙等待。

```
/// 用户进程样例pub fn main() -> usize {
    let (mut write_fd, mut read_fd) = sys_pipe();
    if sys_fork() {
        // 父进程        sys_close(read_fd); // 不一定需要实现        sys_write(write_fd,
"hello_world".as_bytes());
    } else {
        // 子进程        sys_close(write_fd); // 不一定需要实现        let mut buffer = [0u8; 64];
        let len = sys_read(read_fd, &mut buffer);
        println!("{}", core::str::from_utf8(&buffer));
    }
}
```

虽说功能实现了，但是还不完善，由于rcore-fs目前没有提供管道文件，在当前代码框架内实现貌似要改动已有的许多API，所以目前实现的并不能算是一个管道，但是还勉强能用

```
Runtime SBI Version    : 0.2

MIDELEG : 0x0000000000000222
MEDELEG : 0x000000000000b109
PMP0    : 0x0000000080000000-0x000000008001ffff (A)
PMP1    : 0x0000000000000000-0xffffffffffffffff (A,R,W,X)
mod memory initialized
mod interrupt initialized
mod driver initialized
.
..
hello_world
notebook
test_file
mod fs initialized
thread 1 exit with code 0
hello_world
thread 2 exit with code 0
src/process/processor.rs:87: 'all threads terminated, shutting down'
```

```
    let mut fd = [0;2];
    let res =  sys_pipe(&mut fd);
    if res < 0 {
        panic!("创建管道失败")
    }
    let write_fd = fd[0];
    let read_fd = fd[1];
    if sys_fork() != 0 {
        // 父进程
        sys_write(read_fd, "hello_world".as_bytes());
    } else {
        // 子进程
        let mut buffer = [0u8; 64];
        let res = sys_read(read_fd, &mut buffer);
        if res < 0 {
            panic!("读取错误")
        }
        println!("{}", core::str::from_utf8(&buffer).unwrap());
    }
    0
```