实现通关的补全的代码结构

```rust
// os/src/mm/memory_set

impl MemorySet {
    //body 到append_to函数截至都是自带的函数
    //新增了以下三个函数
    pub fn mmap(&mut self, start: VirtPageNum, end: VirtPageNum, flag: MapPermission) {}
    pub fn munmap(&mut self, start: VirtPageNum, end: VirtPageNum) -> Result<(), ()> {}
    pub fn is_overlap(&self, start: VirtPageNum, end: VirtPageNum) -> bool {}
}

// os/src/task/task.rs
pub struct TaskControlBlock {
    //body
    /// Program break
    pub program_brk: usize,

    /// 新增了syscall_times数组
    pub syscall_times: [usize; MAX_SYSCALL_NUM],
}

impl TaskControlBlock {
    //原来的函数
    pub fn new(elf_data: &[u8], app_id: usize) -> Self {
        //body1 未修改

        let task_control_block = Self {
            program_brk: user_sp, ////定位到这里
            //在new里面初始化syscall_times
            syscall_times: [0; MAX_SYSCALL_NUM],
        };

        //body2 未修改
    }
}

// os/src/task/mod.rs 修改的内容
/// mmap syscall
pub fn mmap(start: usize, len: usize, port: usize) -> isize {}

/// munmap syscall
pub fn munmap(start: usize, len: usize) -> isize {}

/// systrace
pub fn task_trace(_trace_request: usize, _id: usize, _data: usize) -> isize {
    trace!("kernel: sys_trace");
}

```

$$\text{可用页数} = \left\lfloor \frac{\text{MEMORY\_END}}{\text{PAGE\_SIZE}} \right\rfloor - \left\lceil \frac{\text{ekernel}}{\text{PAGE\_SIZE}} \right\rceil$$

以下是需要补全的代码任务

```rust
// os/src/syscall/process.rs

//! 进程管理系统调用
use crate::task::{change_program_brk, exit_current_and_run_next,
suspend_current_and_run_next};

#[repr(C)]
#[derive(Debug)]
pub struct TimeVal {
    pub sec: usize,     // 秒
    pub usec: usize,    // 微秒
}

/// 任务退出并提交退出码
pub fn sys_exit(_exit_code: i32) -> ! {
    trace!("kernel: sys_exit");
    exit_current_and_run_next();
    panic!("sys_exit 中不可达!");
}

/// 当前任务主动让出资源给其他任务
pub fn sys_yield() -> isize {
    trace!("kernel: sys_yield");
    suspend_current_and_run_next();
    0
}

/// 你的任务：获取时间（秒和微秒）
/// 提示：你可能需要通过虚拟内存管理重新实现它
/// 提示：如果 [`TimeVal`] 结构被跨页分割会怎样？
pub fn sys_get_time(_ts: *mut TimeVal, _tz: usize) -> isize {
    trace!("kernel: sys_get_time");
    -1
}

/// TODO: 完成 sys_trace 以通过测试用例
/// 提示：你可能需要通过虚拟内存管理重新实现它
pub fn sys_trace(_trace_request: usize, _id: usize, _data: usize) -> isize {
    trace!("kernel: sys_trace");
    -1
}

// 你的任务：实现 mmap
pub fn sys_mmap(_start: usize, _len: usize, _port: usize) -> isize {
    trace!("kernel: sys_mmap 尚未实现!");
    -1
}

// 你的任务：实现 munmap
```

```rust
pub fn sys_munmap(_start: usize, _len: usize) -> isize {
    trace!("kernel: sys_munmap 尚未实现!");
    -1
}

/// 修改数据段大小
pub fn sys_sbrk(size: i32) -> isize {
    trace!("kernel: sys_sbrk");
    if let Some(old_brk) = change_program_brk(size) {
        old_brk as isize
    } else {
        -1
    }
}
```

测试函数都在

/user/src/bin/ch3_xxx

这里

1.sys_get_time

```rust
pub fn sys_get_time(_ts: *mut TimeVal, _tz: usize) -> isize {
    //ts 时间结构 time structure 这里是TimeVal
    //tz 时区 timezone 未使用
    trace!("kernel: sys_get_time");
    let token = current_user_token();
    let _page_table = PageTable::from_token(token);

    // 将时间转换为虚拟地址
    let start: usize = _ts as usize;
    let len: usize = core::mem::size_of::<TimeVal>(); // TimeVal 的大小

    //防止end溢出
    let end: usize = match start.checked_add(len) {
        Some(val) => val,
        None => {
            return -1;
        }
    };

    // 检查是否跨页
    let start_vpn = VirtAddr::from(start).floor();
    let end_vpn = VirtAddr::from(end).floor();
    if start_vpn != end_vpn {
        return -1;
    }

    // 翻译虚拟地址并检查权限
    // 检查页表项是否存在且可读 Read仅仅需要readable

    //这里如果获取成功,就已经定义了pte这个变量
```

```
    if let Some(pte) = _page_table.translate(start_vpn) {
        if !pte.is_valid() || !pte.writable() {
            return -1;
        }

        // 获取当前时间（以微秒为单位）
        let time_ = get_time_us();

        // 将微秒转换为秒和微秒
        let curr_time = TimeVal {
            sec: time_ / 1_000_000,
            usec: time_ % 1_000_000,
        };
        let ppn = pte.ppn();
        let offset = VirtAddr::from(start).page_offset();

        //time_bytes是一个切片
        let time_bytes = unsafe {
            core::slice::from_raw_parts(
                //裸指针
                &curr_time as *const TimeVal as *const u8,
                //size大小
                len
            )
        };
        //把ppn返回物理地址,然后取出从[offset,offset+len]的范围,然后将源切片的数据复制到目标切片
        ppn.get_bytes_array()[offset..offset + len].copy_from_slice(time_bytes);
        // 写入时间到物理内存
        0 // 成功
    } else {
        -1 // 页面不存在
    }
}
```

2 sys_trace

```
pub fn sys_trace(_trace_request: usize, _id: usize, _data: usize) -> isize {
    trace!("kernel: sys_trace");
    // 获取当前任务的用户态token
    let token = current_user_token(); //satp 寄存器的值，包含页表物理地址和模式信息
    let _page_table = PageTable::from_token(token); //临时PageTable

    match _trace_request {
        // 读取操作 - 读取1字节内存
        0 => {
            let is_valid_id = |id: usize, len: usize| -> bool {
                const ADD_MAX: usize = 0x8000_0000;
                const PAGE_SIZE: usize = 4096; //4KB
                //end = id + len ,end是在增加之后的实际页码,但是是左闭右开区间[start,end)
                //所以下面需要 -1
                let end = match id.checked_add(len) {
                    Some(val) => val,
                    None => {
```

```
                    return false;
                }
            };
            if end > ADD_MAX {
                return false;
            }
            if len > PAGE_SIZE {
                let strat_page = id / PAGE_SIZE;
                let end_page = (end - 1) / PAGE_SIZE;
                if strat_page != end_page {
                    return false;
                }
            }
            true
        };
        //以下是判断代码
        let _vpn = VirtAddr::from(_id).floor();
        if !is_valid_id(_id, 1) {
            return -1;
        }
        let vpn = VirtAddr::from(_id).floor();

        // 检查页表项是否存在且可读 Read仅仅需要readable
        if let Some(pte) = _page_table.translate(vpn) {
            if !pte.is_valid() || !pte.readable() {
                return -1;
            }

            // 安全读取字节
            let ppn = pte.ppn();
            let offset = VirtAddr::from(_id).page_offset();
            let byte = ppn.get_bytes_array()[offset];
            byte as isize
        } else {
            -1
        }
    }

    // 写入操作 - 写入usize数据
    1 => {
        let vpn = VirtAddr::from(_id).floor();

        // 检查页表项是否存在且可写
        if let Some(pte) = _page_table.translate(vpn) {
            if !pte.is_valid() || !pte.writable() || !pte.readable() {
                return -1;
            }

            // 检查是否跨页
            let start = _id;
            let end = _id + core::mem::size_of::<usize>();
            if VirtAddr::from(start).floor() != VirtAddr::from(end - 1).floor() {
                return -1; // 不支持跨页写入
            }

            // 安全写入
            let ppn = pte.ppn();
            let offset = VirtAddr::from(_id).page_offset();
```

```
                let bytes = _data.to_ne_bytes();
                ppn.get_bytes_array()[
                    offset..offset + core::mem::size_of::<usize>()
                ].copy_from_slice(&bytes);
                0
            } else {
                -1
            }
        }

        // 无效请求
        _ => -1,
    }
}
```