19271169-张东植-实验报告7

- **Gitlab repo: http://202.205.102.126:88/ZhangDongZhi/os-lab.git**

The main purpose of this experiment is to realize process and process management. Through this experiment can realize the operating system process and process management. Specifically, it includes adding system call, linking and loading of application, process identifier and kernel stack, modifying implementation process control block, implementing task manager, increasing processor management structure, creating initial process, process scheduling mechanism, process generation mechanism, and process resource recovery mechanism, etc.

1. 修改应用程序

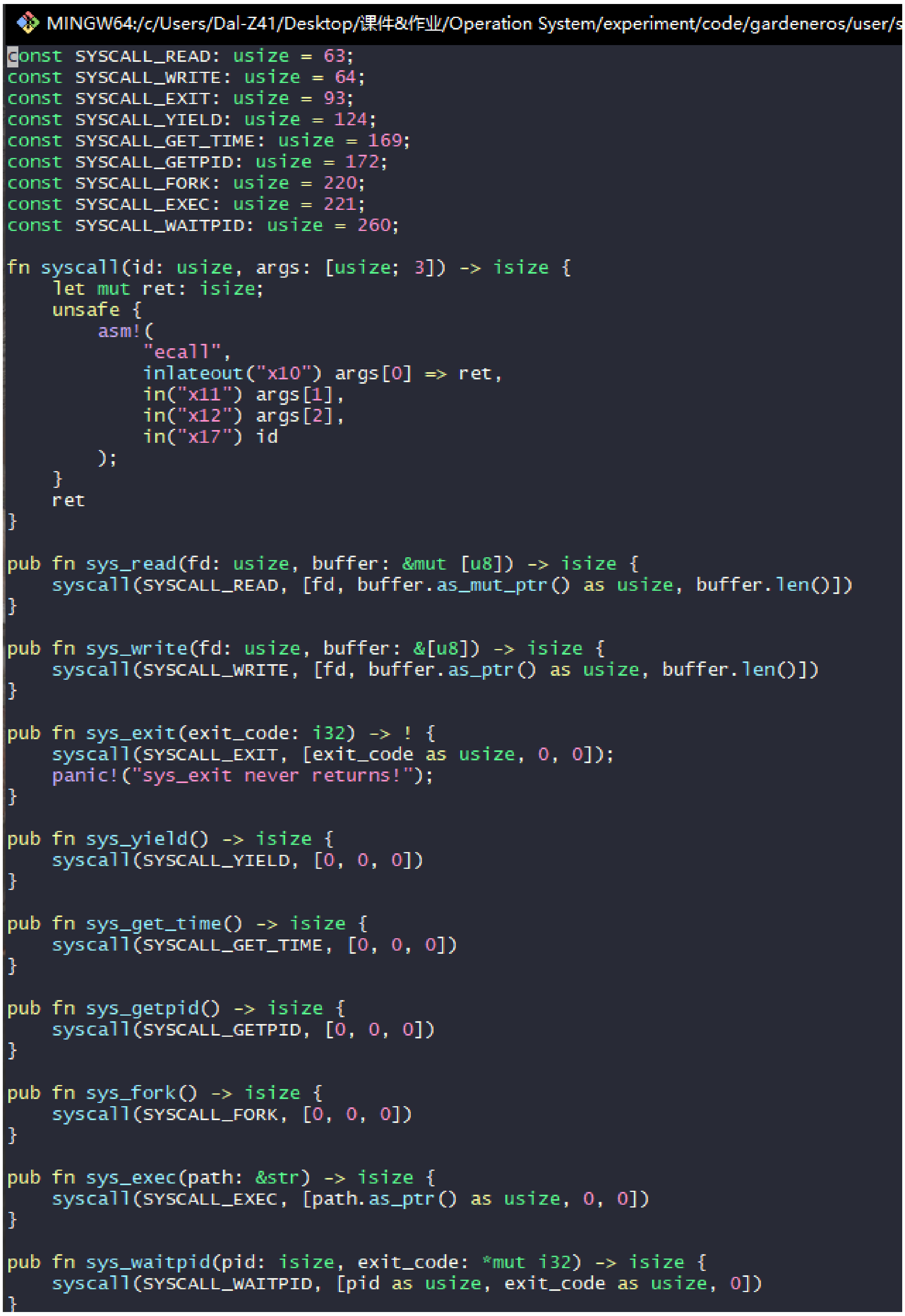
1.1 Add sys_call

The fork system call creates a new process; The **waitpid** system call causes the current process to wait for the child process to finish, reclaim its resources and get the return value; The **getpid** system call gets information about the current process; The **exec** system call clears the current process address space and loads a specific executable, then returns to user-mode execution; The **read** system call reads a piece of content from a file into a buffer, with the primary purpose of implementing the user shell.

1. First, modify "**user/ src /syscall.rs** "to add the above system call.

```
//user/src/syscall.rs
const SYSCALL_READ: usize = 63;
const SYSCALL_GETPID: usize = 172;
const SYSCALL_FORK: usize = 220;
```

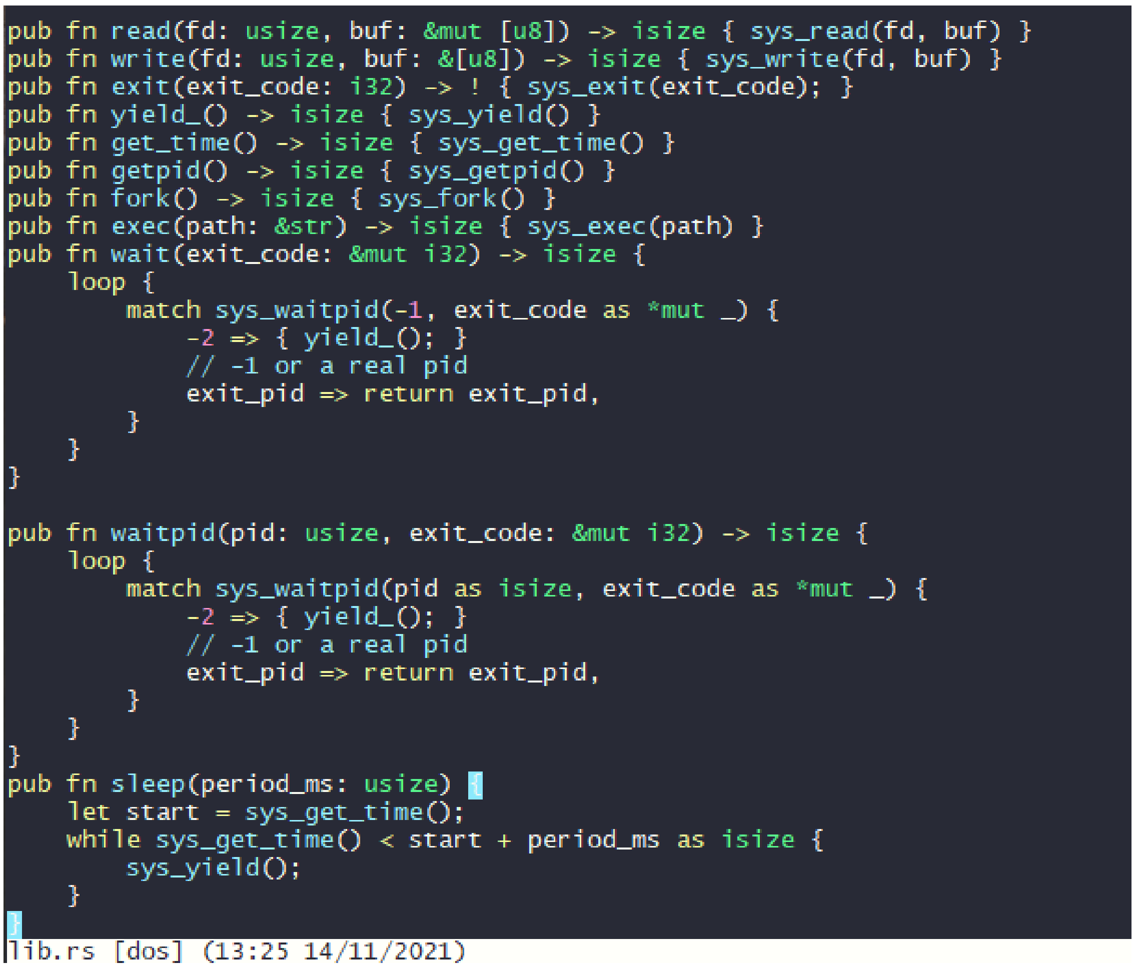
```rust
const SYSCALL_EXEC: usize = 221;
const SYSCALL_WAITPID: usize = 260;

pub fn sys_read(fd: usize, buffer: &mut [u8]) -> isize {
    syscall(SYSCALL_READ, [fd, buffer.as_mut_ptr() as usize, buffer.len()])
}

pub fn sys_getpid() -> isize {
    syscall(SYSCALL_GETPID, [0, 0, 0])
}

pub fn sys_fork() -> isize {
    syscall(SYSCALL_FORK, [0, 0, 0])
}

pub fn sys_exec(path: &str) -> isize {
    syscall(SYSCALL_EXEC, [path.as_ptr() as usize, 0, 0])
}

pub fn sys_waitpid(pid: isize, exit_code: *mut i32) -> isize {
    syscall(SYSCALL_WAITPID, [pid as usize, exit_code as usize, 0])
}
```


MINGW64:/c/Users/Dal-Z41/Desktop/课件&作业/Operation System/experiment/code/gardeneros/user/s

```
const SYSCALL_READ: usize = 63;
const SYSCALL_WRITE: usize = 64;
const SYSCALL_EXIT: usize = 93;
const SYSCALL_YIELD: usize = 124;
const SYSCALL_GET_TIME: usize = 169;
const SYSCALL_GETPID: usize = 172;
const SYSCALL_FORK: usize = 220;
const SYSCALL_EXEC: usize = 221;
const SYSCALL_WAITPID: usize = 260;

fn syscall(id: usize, args: [usize; 3]) -> isize {
    let mut ret: isize;
    unsafe {
        asm!(
            "ecall",
            inlateout("x10") args[0] => ret,
            in("x11") args[1],
            in("x12") args[2],
            in("x17") id
        );
    }
    ret
}

pub fn sys_read(fd: usize, buffer: &mut [u8]) -> isize {
    syscall(SYSCALL_READ, [fd, buffer.as_mut_ptr() as usize, buffer.len()])
}

pub fn sys_write(fd: usize, buffer: &[u8]) -> isize {
    syscall(SYSCALL_WRITE, [fd, buffer.as_ptr() as usize, buffer.len()])
}

pub fn sys_exit(exit_code: i32) -> ! {
    syscall(SYSCALL_EXIT, [exit_code as usize, 0, 0]);
    panic!("sys_exit never returns!");
}

pub fn sys_yield() -> isize {
    syscall(SYSCALL_YIELD, [0, 0, 0])
}

pub fn sys_get_time() -> isize {
    syscall(SYSCALL_GET_TIME, [0, 0, 0])
}

pub fn sys_getpid() -> isize {
    syscall(SYSCALL_GETPID, [0, 0, 0])
}

pub fn sys_fork() -> isize {
    syscall(SYSCALL_FORK, [0, 0, 0])
}

pub fn sys_exec(path: &str) -> isize {
    syscall(SYSCALL_EXEC, [path.as_ptr() as usize, 0, 0])
}

pub fn sys_waitpid(pid: isize, exit_code: *mut i32) -> isize {
    syscall(SYSCALL_WAITPID, [pid as usize, exit_code as usize, 0])
}
```

2. The system call is then encapsulated in "**user/ src/lib.rs**" in the form used by the application.

```
//user/src/lib.rs
pub fn read(fd: usize, buf: &mut [u8]) -> isize { sys_read(fd, buf) }

pub fn getpid() -> isize { sys_getpid() }
pub fn fork() -> isize { sys_fork() }
pub fn exec(path: &str) -> isize { sys_exec(path) }
pub fn wait(exit_code: &mut i32) -> isize {
    loop {
        match sys_waitpid(-1, exit_code as *mut _) {
                -2 => { yield_(); }
                // -1 or a real pid
                exit_pid => return exit_pid,
            }
        }
}

pub fn waitpid(pid: usize, exit_code: &mut i32) -> isize {
    loop {
        match sys_waitpid(pid as isize, exit_code as *mut _) {
            -2 => { yield_(); }
            // -1 or a real pid
            exit_pid => return exit_pid,
        }
    }
}

pub fn sleep(period_ms: usize) {
    let start = sys_get_time();
    while sys_get_time() < start + period_ms as isize {
        sys_yield();
    }
}
```

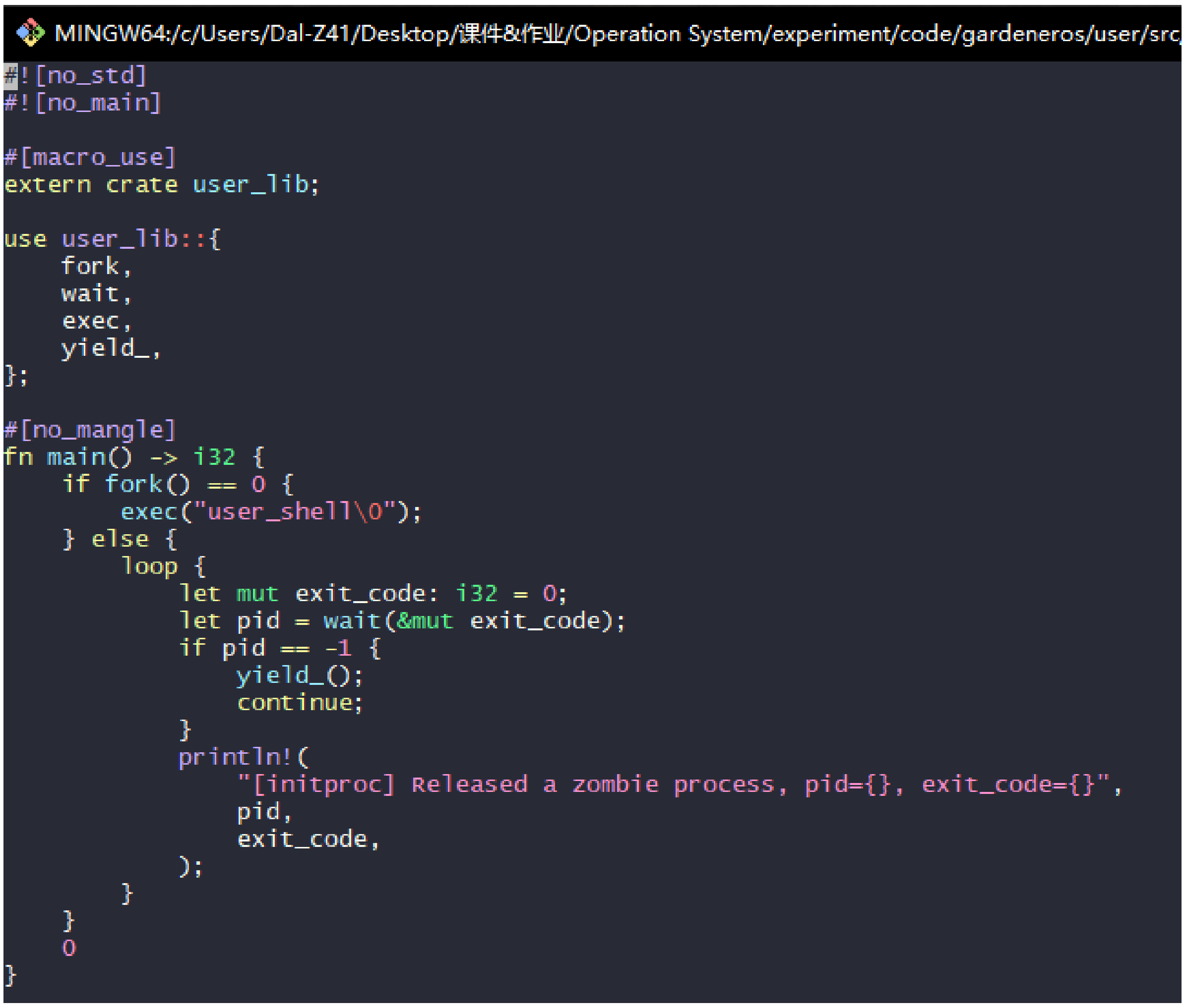


```
pub fn read(fd: usize, buf: &mut [u8]) -> isize { sys_read(fd, buf) }
pub fn write(fd: usize, buf: &[u8]) -> isize { sys_write(fd, buf) }
pub fn exit(exit_code: i32) -> ! { sys_exit(exit_code); }
pub fn yield_() -> isize { sys_yield() }
pub fn get_time() -> isize { sys_get_time() }
pub fn getpid() -> isize { sys_getpid() }
pub fn fork() -> isize { sys_fork() }
pub fn exec(path: &str) -> isize { sys_exec(path) }
pub fn wait(exit_code: &mut i32) -> isize {
    loop {
        match sys_waitpid(-1, exit_code as *mut _) {
            -2 => { yield_(); }
            // -1 or a real pid
            exit_pid => return exit_pid,
        }
    }
}

pub fn waitpid(pid: usize, exit_code: &mut i32) -> isize {
    loop {
        match sys_waitpid(pid as isize, exit_code as *mut _) {
            -2 => { yield_(); }
            // -1 or a real pid
            exit_pid => return exit_pid,
        }
    }
}
pub fn sleep(period_ms: usize) {
    let start = sys_get_time();
    while sys_get_time() < start + period_ms as isize {
        sys_yield();
    }
}
lib.rs [dos] (13:25 14/11/2021)
```

1.2 initproc.rs

```
//user/src/bin/initproc.rs
#![no_std]
#![no_main]
```

```rust
#[macro_use]
extern crate user_lib;

use user_lib::{
    fork,
    wait,
    exec,
    yield_,
};

#[no_mangle]
fn main() -> i32 {
    if fork() == 0 {
        exec("user_shell\0");
    } else {
        loop {
            let mut exit_code: i32 = 0;
            let pid = wait(&mut exit_code);
            if pid == -1 {
                yield_();rrr
                continue;
            }
            println!(
                "[initproc] Released a zombie process, pid={}, exit_code={}",
                pid,
                exit_code,
            );
        }
    }
    0
}
```

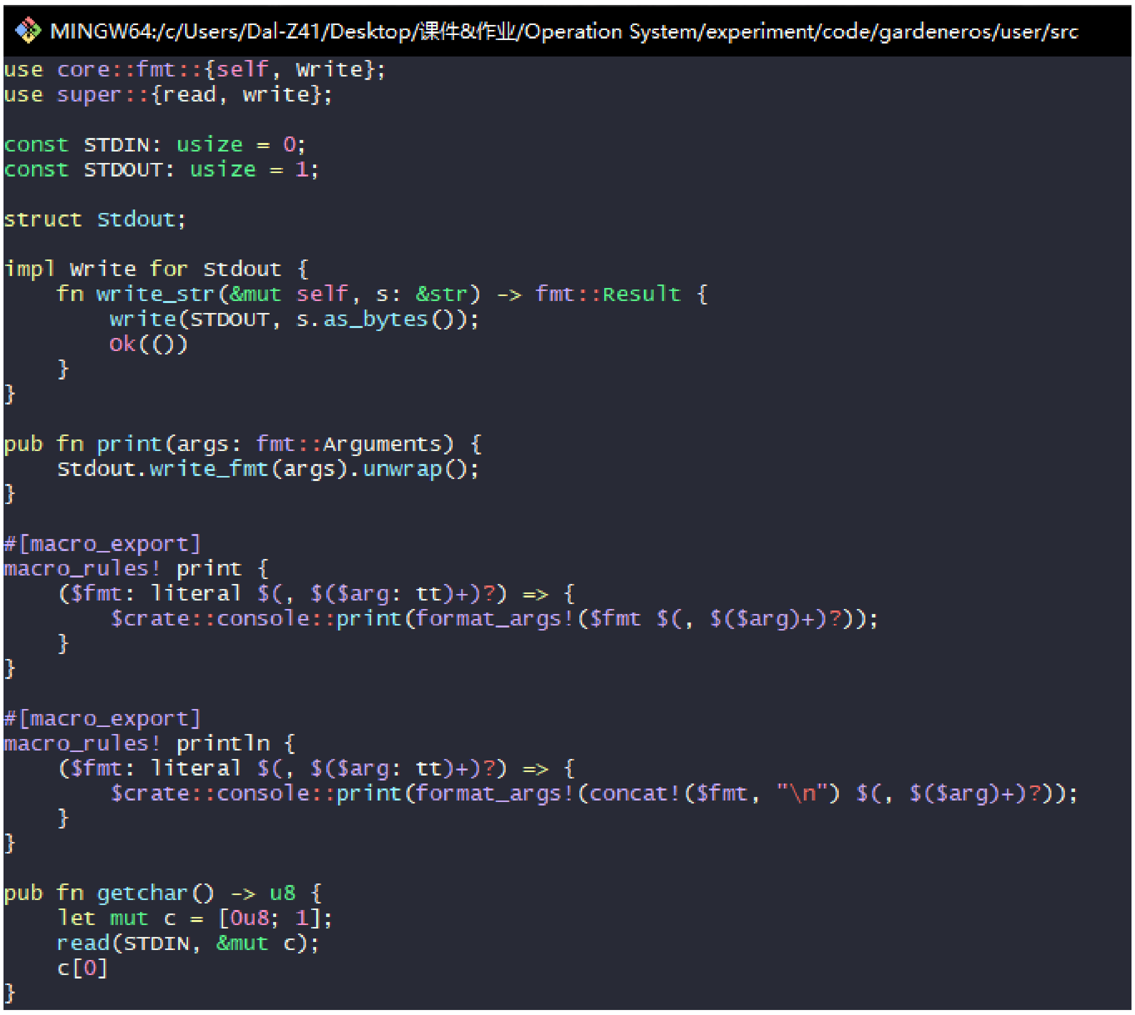

MINGW64:/c/Users/Dal-Z41/Desktop/课件&作业/Operation System/experiment/code/gardeneros/user/src

```
#![no_std]
#![no_main]

#[macro_use]
extern crate user_lib;

use user_lib::{
    fork,
    wait,
    exec,
    yield_,
};

#[no_mangle]
fn main() -> i32 {
    if fork() == 0 {
        exec("user_shell\0");
    } else {
        loop {
            let mut exit_code: i32 = 0;
            let pid = wait(&mut exit_code);
            if pid == -1 {
                yield_();
                continue;
            }
            println!(
                "[initproc] Released a zombie process, pid={}, exit_code={}",
                pid,
                exit_code,
            );
        }
    }
    0
}
```

1.3 Shell program

1. First, the" **sys_read**" system call encapsulates the function "**getchar**" that reads a character from standard input.

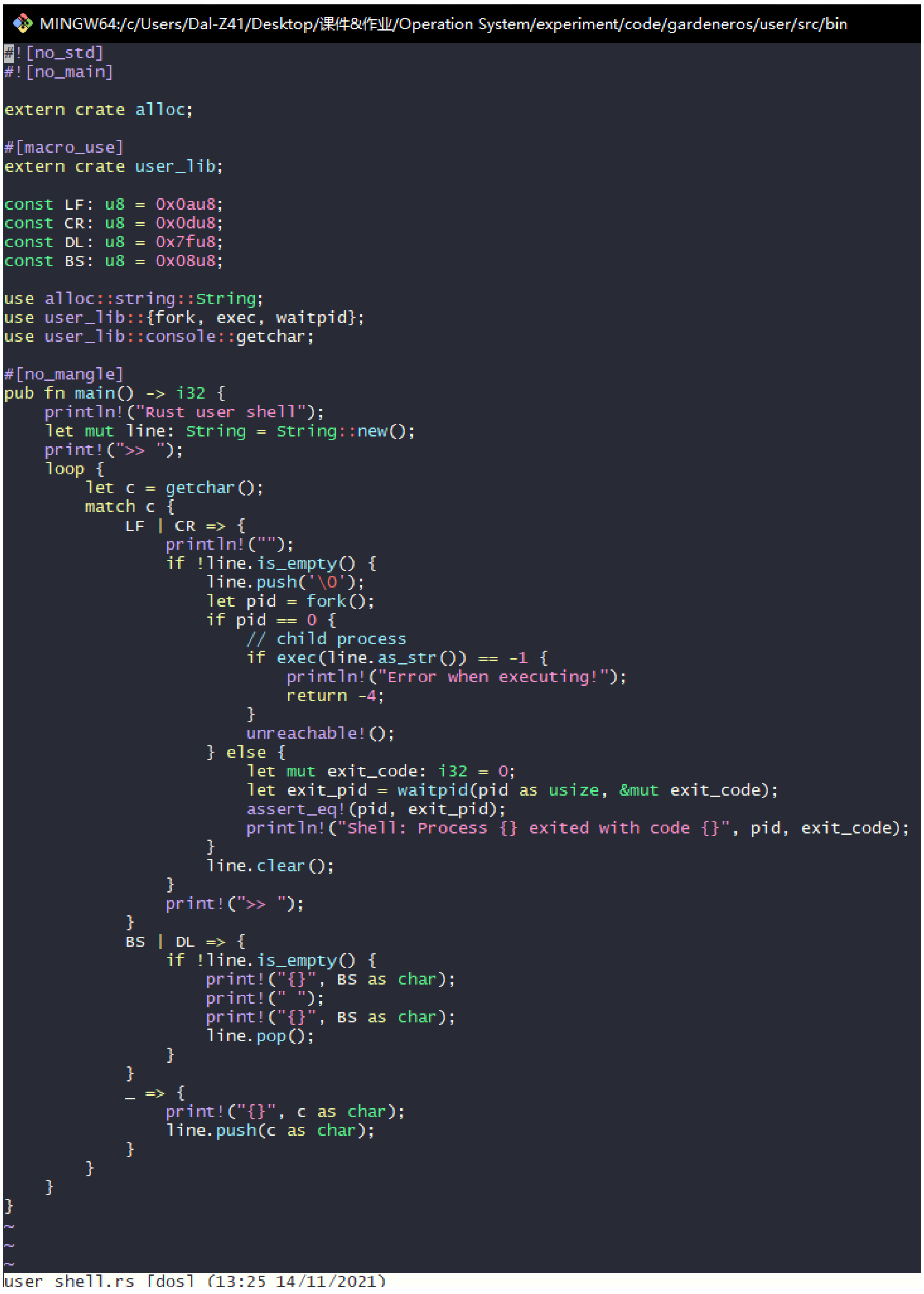
```
//user/src/console.rs
use super::read;

const STDIN: usize = 0;

pub fn getchar() -> u8 {
    let mut c = [0u8; 1];
    read(STDIN, &mut c);
    c[0]
}
```


MINGW64:/c/Users/Dal-Z41/Desktop/课件&作业/Operation System/experiment/code/gardeneros/user/src

```
use core::fmt::{self, Write};
use super::{read, write};

const STDIN: usize = 0;
const STDOUT: usize = 1;

struct Stdout;

impl Write for Stdout {
    fn write_str(&mut self, s: &str) -> fmt::Result {
        write(STDOUT, s.as_bytes());
        Ok(())
    }
}

pub fn print(args: fmt::Arguments) {
    Stdout.write_fmt(args).unwrap();
}

#[macro_export]
macro_rules! print {
    ($fmt: literal $(, $($arg: tt)+)?) => {
        $crate::console::print(format_args!($fmt $(, $($arg)+)?));
    }
}

#[macro_export]
macro_rules! println {
    ($fmt: literal $(, $($arg: tt)+)?) => {
        $crate::console::print(format_args!(concat!($fmt, "\n") $(, $($arg)+)?));
    }
}

pub fn getchar() -> u8 {
    let mut c = [0u8; 1];
    read(STDIN, &mut c);
    c[0]
}
```

2. Then, implement program **"user shell"**.

```
//user/src/bin/user_shell.rs
#![no_std]
#![no_main]

extern crate alloc;

#[macro_use]
extern crate user_lib;

const LF: u8 = 0x0au8;
const CR: u8 = 0x0du8;
const DL: u8 = 0x7fu8;
const BS: u8 = 0x08u8;

use alloc::string::String;
use user_lib::{fork, exec, waitpid};
use user_lib::console::getchar;

#[no_mangle]
pub fn main() -> i32 {
    println!("Rust user shell");
    let mut line: String = String::new();
    print!(">> ");
    loop {
        let c = getchar();
        match c {
            LF | CR => {
                println!("");
                if !line.is_empty() {
                    line.push('\0');
                    let pid = fork();
                    if pid == 0 {
                        // child process
                        if exec(line.as_str()) == -1 {
                            println!("Error when executing!");
                            return -4;
                        }
                        unreachable!();
                    } else {
                        let mut exit_code: i32 = 0;
                        let exit_pid = waitpid(pid as usize, &mut
exit_code);
                        assert_eq!(pid, exit_pid);
                        println!("Shell: Process {} exited with code {}",
pid, exit_code);
                    }
                    line.clear();
                }
                print!(">> ");
            }
            BS | DL => {
                if !line.is_empty() {
                    print!("{}", BS as char);
                    print!(" ");
                    print!("{}", BS as char);
                    line.pop();
                }
            }
            _ => {
                print!("{}", c as char);
                line.push(c as char);
            }
        }
    }
}
```

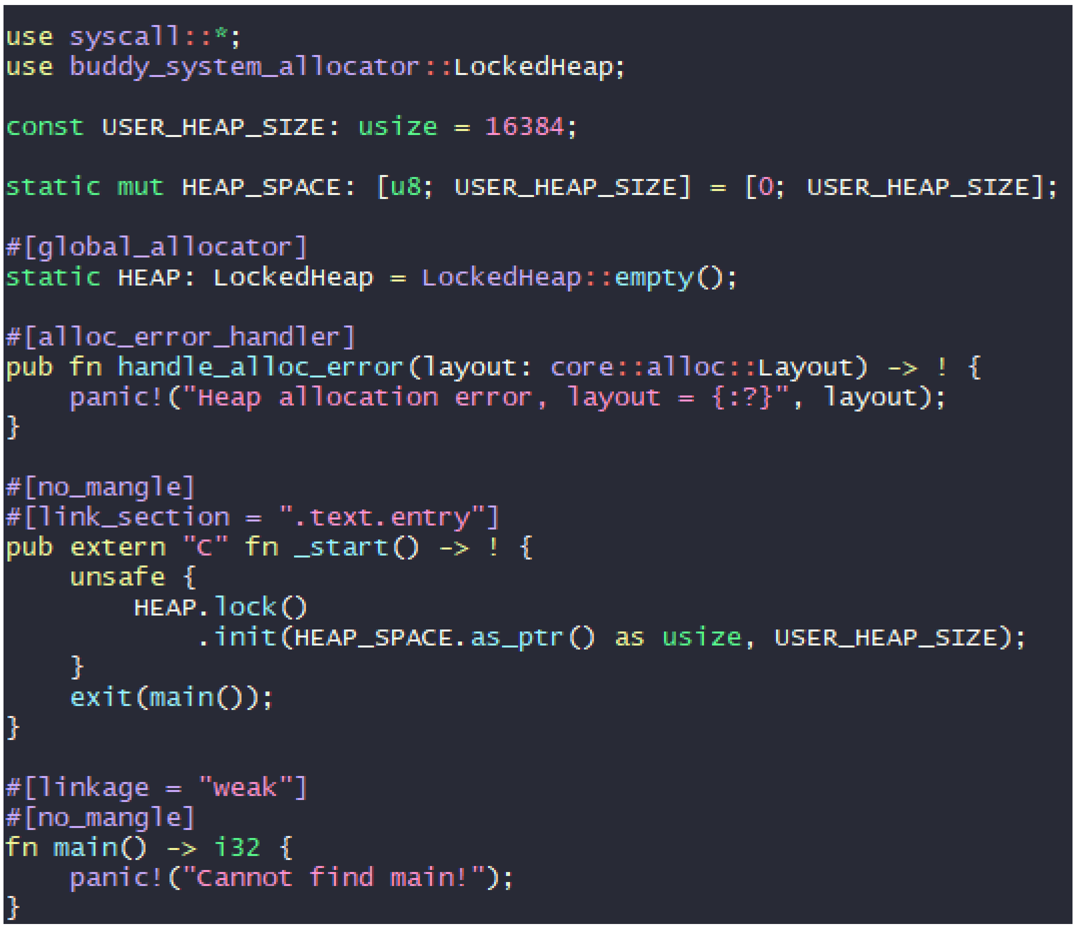

MINGW64:/c/Users/Dal-Z41/Desktop/课件&作业/Operation System/experiment/code/gardeneros/user/src/bin

```rust
#![no_std]
#![no_main]

extern crate alloc;

#[macro_use]
extern crate user_lib;

const LF: u8 = 0x0au8;
const CR: u8 = 0x0du8;
const DL: u8 = 0x7fu8;
const BS: u8 = 0x08u8;

use alloc::string::String;
use user_lib::{fork, exec, waitpid};
use user_lib::console::getchar;

#[no_mangle]
pub fn main() -> i32 {
    println!("Rust user shell");
    let mut line: String = String::new();
    print!(">> ");
    loop {
        let c = getchar();
        match c {
            LF | CR => {
                println!("");
                if !line.is_empty() {
                    line.push('\0');
                    let pid = fork();
                    if pid == 0 {
                        // child process
                        if exec(line.as_str()) == -1 {
                            println!("Error when executing!");
                            return -4;
                        }
                        unreachable!();
                    } else {
                        let mut exit_code: i32 = 0;
                        let exit_pid = waitpid(pid as usize, &mut exit_code);
                        assert_eq!(pid, exit_pid);
                        println!("Shell: Process {} exited with code {}", pid, exit_code);
                    }
                    line.clear();
                }
                print!(">> ");
            }
            BS | DL => {
                if !line.is_empty() {
                    print!("{}", BS as char);
                    print!(" ");
                    print!("{}", BS as char);
                    line.pop();
                }
            }
            _ => {
                print!("{}", c as char);
                line.push(c as char);
            }
        }
    }
}
~
~
~
user_shell.rs [dos] (13:25 14/11/2021)
```

3. Since Rust's edgeable String type String is based on dynamic memory allocation, you also need to support dynamic memory allocation in user library "**user_lib**".

```rust
//usr/src/lib.rs
#![feature(alloc_error_handler)]

use buddy_system_allocator::LockedHeap;

const USER_HEAP_SIZE: usize = 16384;
static mut HEAP_SPACE: [u8; USER_HEAP_SIZE] = [0; USER_HEAP_SIZE];

#[global_allocator]
static HEAP: LockedHeap = LockedHeap::empty();
```

```
#[alloc_error_handler]
pub fn handle_alloc_error(layout: core::alloc::Layout) -> ! {
    panic!("Heap allocation error, layout = {:?}", layout);
}

#[no_mangle]
#[link_section = ".text.entry"]
pub extern "C" fn _start() -> ! {
    unsafe {
        HEAP.lock()
            .init(HEAP_SPACE.as_ptr() as usize, USER_HEAP_SIZE);
    }
    exit(main());
}
```

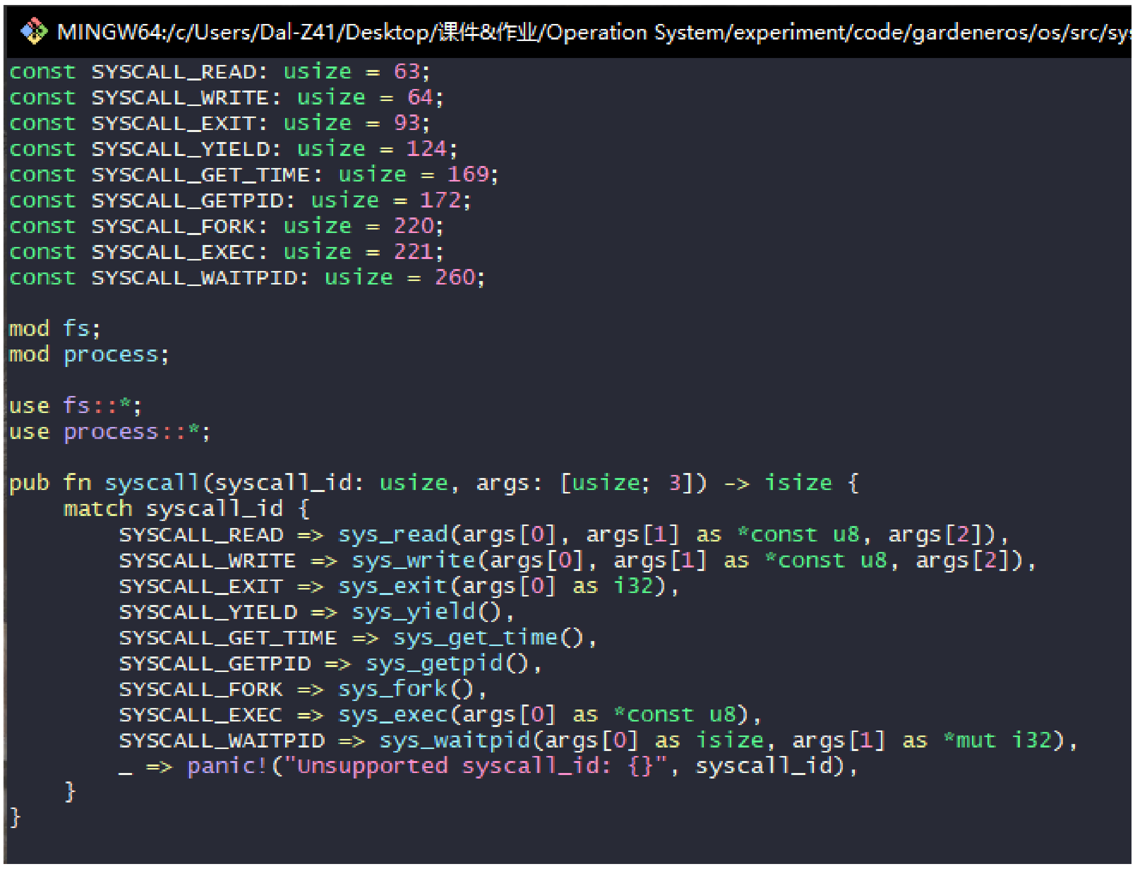


```
use syscall::*;
use buddy_system_allocator::LockedHeap;

const USER_HEAP_SIZE: usize = 16384;

static mut HEAP_SPACE: [u8; USER_HEAP_SIZE] = [0; USER_HEAP_SIZE];

#[global_allocator]
static HEAP: LockedHeap = LockedHeap::empty();

#[alloc_error_handler]
pub fn handle_alloc_error(layout: core::alloc::Layout) -> ! {
    panic!("Heap allocation error, layout = {:?}", layout);
}

#[no_mangle]
#[link_section = ".text.entry"]
pub extern "C" fn _start() -> ! {
    unsafe {
        HEAP.lock()
            .init(HEAP_SPACE.as_ptr() as usize, USER_HEAP_SIZE);
    }
    exit(main());
}

#[linkage = "weak"]
#[no_mangle]
fn main() -> i32 {
    panic!("Cannot find main!");
}
```

2. 内核增加系统调用

1. First, modify "**os/src/syscall/mod.rs**" to add fork, **waitpid**, **getpid**, **read** system calls.

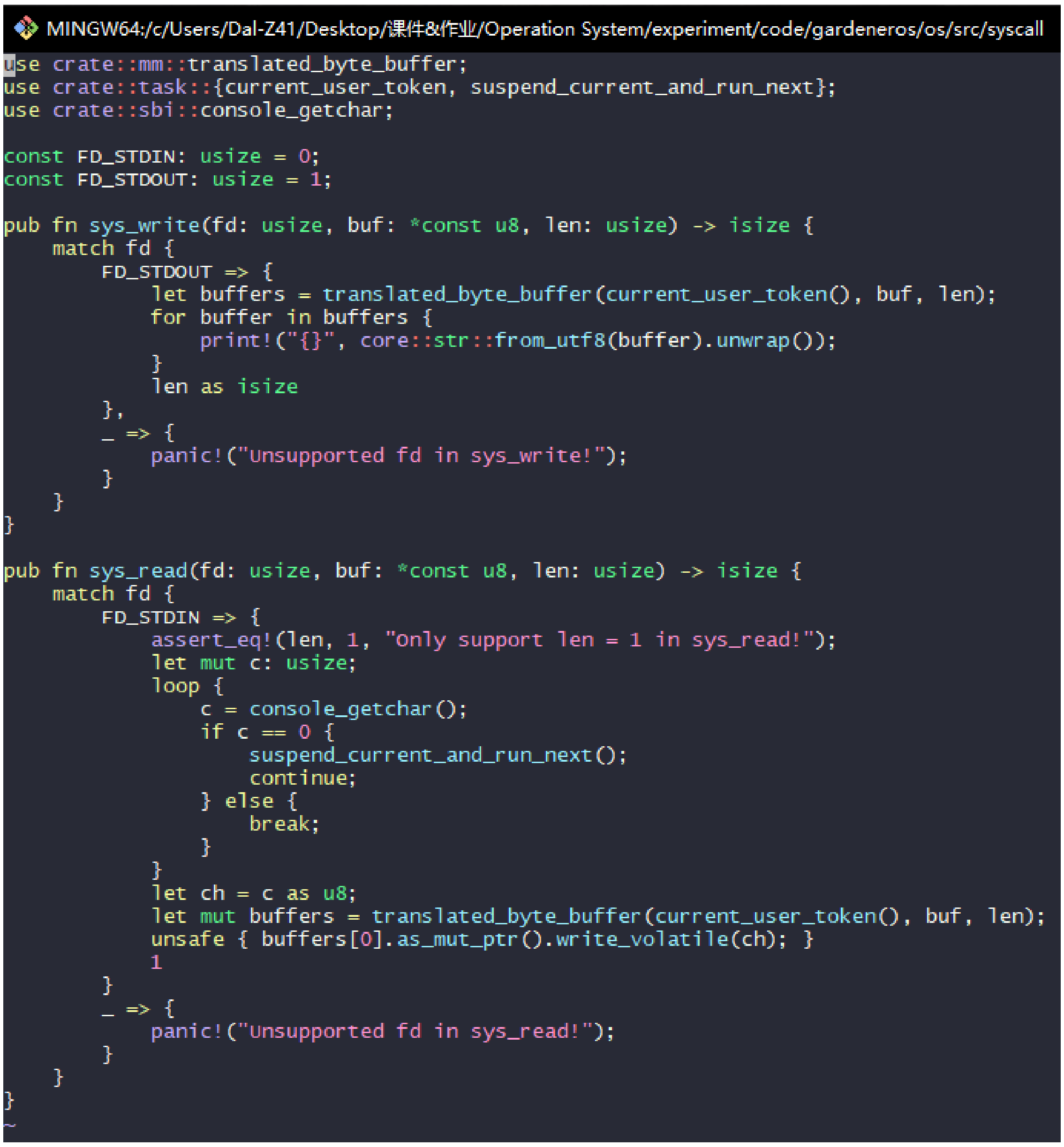
```
//os/src/syscall/mod.rs
const SYSCALL_READ: usize = 63;
const SYSCALL_GETPID: usize = 172;
const SYSCALL_FORK: usize = 220;
const SYSCALL_EXEC: usize = 221;
const SYSCALL_WAITPID: usize = 260;

mod fs;
mod process;

use fs::*;
use process::*;

pub fn syscall(syscall_id: usize, args: [usize; 3]) -> isize {
    match syscall_id {
        SYSCALL_READ => sys_read(args[0], args[1] as *const u8, args[2]),
        SYSCALL_WRITE => sys_write(args[0], args[1] as *const u8, args[2]),
        SYSCALL_EXIT => sys_exit(args[0] as i32),
        SYSCALL_YIELD => sys_yield(),
        SYSCALL_GET_TIME => sys_get_time(),
        SYSCALL_GETPID => sys_getpid(),
        SYSCALL_FORK => sys_fork(),
        SYSCALL_EXEC => sys_exec(args[0] as *const u8),
        SYSCALL_WAITPID => sys_waitpid(args[0] as isize, args[1] as *mut
i32),
        _ => panic!("Unsupported syscall_id: {}", syscall_id),
    }
}
```


MINGW64:/c/Users/Dal-Z41/Desktop/课件&作业/Operation System/experiment/code/gardeneros/os/src/sy

```
const SYSCALL_READ: usize = 63;
const SYSCALL_WRITE: usize = 64;
const SYSCALL_EXIT: usize = 93;
const SYSCALL_YIELD: usize = 124;
const SYSCALL_GET_TIME: usize = 169;
const SYSCALL_GETPID: usize = 172;
const SYSCALL_FORK: usize = 220;
const SYSCALL_EXEC: usize = 221;
const SYSCALL_WAITPID: usize = 260;

mod fs;
mod process;

use fs::*;
use process::*;

pub fn syscall(syscall_id: usize, args: [usize; 3]) -> isize {
    match syscall_id {
        SYSCALL_READ => sys_read(args[0], args[1] as *const u8, args[2]),
        SYSCALL_WRITE => sys_write(args[0], args[1] as *const u8, args[2]),
        SYSCALL_EXIT => sys_exit(args[0] as i32),
        SYSCALL_YIELD => sys_yield(),
        SYSCALL_GET_TIME => sys_get_time(),
        SYSCALL_GETPID => sys_getpid(),
        SYSCALL_FORK => sys_fork(),
        SYSCALL_EXEC => sys_exec(args[0] as *const u8),
        SYSCALL_WAITPID => sys_waitpid(args[0] as isize, args[1] as *mut i32),
        _ => panic!("Unsupported syscall_id: {}", syscall_id),
    }
}
```

2. Then, modify" **os/src/syscall/fs.rs**" to implement the **sys_read** system call. The suspend_current_AND_RUN_next function suspends the current task and switches to the next one.

```
//os/src/syscall/fs.rs
use crate::task::{current_user_token, suspend_current_and_run_next};
use crate::sbi::console_getchar;
const FD_STDIN: usize = 0;


pub fn sys_read(fd: usize, buf: *const u8, len: usize) -> isize {
    match fd {
        FD_STDIN => {
            assert_eq!(len, 1, "Only support len = 1 in sys_read!");
            let mut c: usize;
            loop {
                c = console_getchar();
                if c == 0 {
                    suspend_current_and_run_next();
                    continue;
                } else {
                    break;
                }
            }
            let ch = c as u8;
            let mut buffers = translated_byte_buffer(current_user_token(),
buf, len);
            unsafe { buffers[0].as_mut_ptr().write_volatile(ch); }
            1
        }
        _ => {
            panic!("Unsupported fd in sys_read!");
        }
    }
}
```

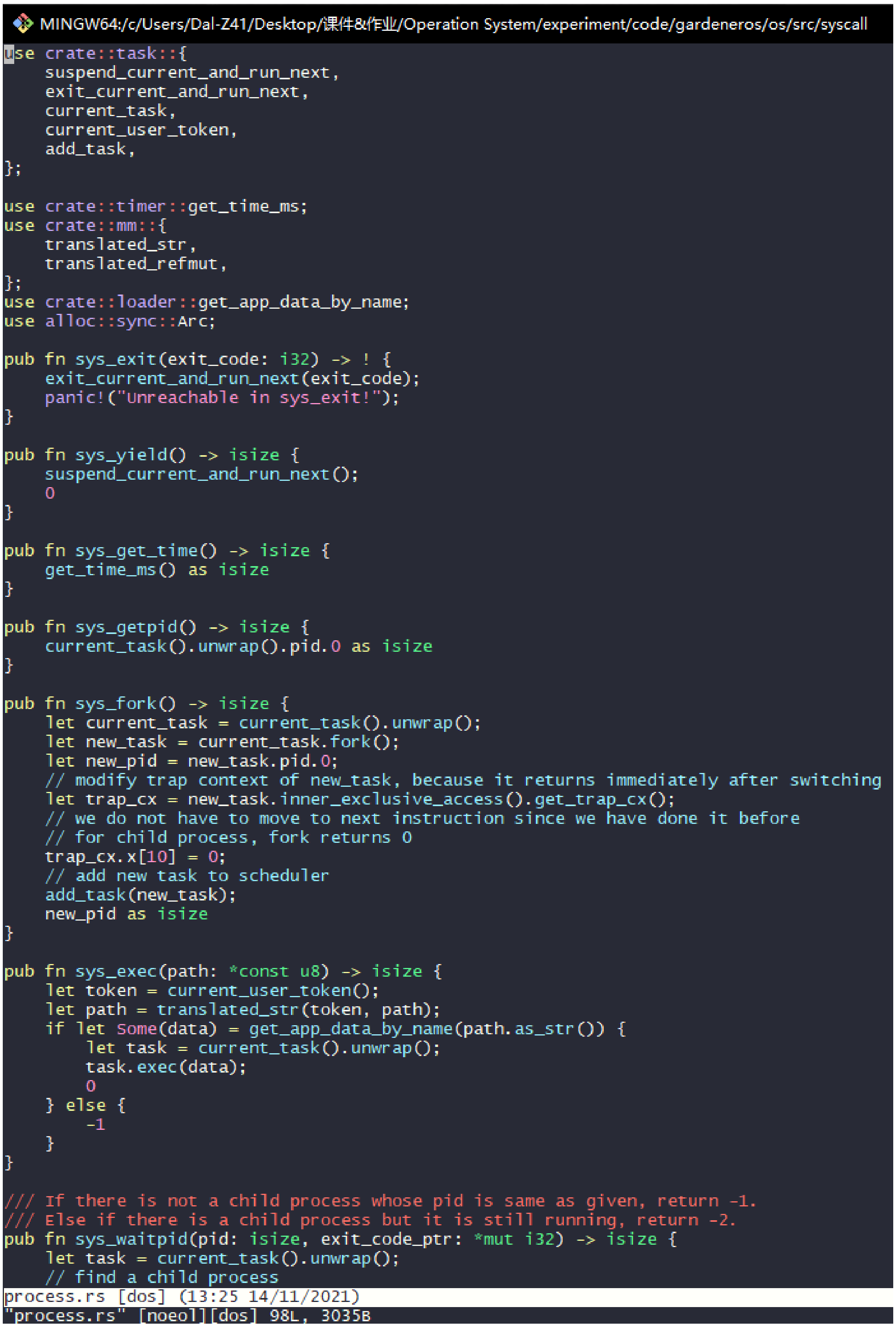


MINGW64:/c/Users/Dal-Z41/Desktop/课件&作业/Operation System/experiment/code/gardeneros/os/src/syscall

```
use crate::mm::translated_byte_buffer;
use crate::task::{current_user_token, suspend_current_and_run_next};
use crate::sbi::console_getchar;

const FD_STDIN: usize = 0;
const FD_STDOUT: usize = 1;

pub fn sys_write(fd: usize, buf: *const u8, len: usize) -> isize {
    match fd {
        FD_STDOUT => {
            let buffers = translated_byte_buffer(current_user_token(), buf, len);
            for buffer in buffers {
                print!("{}", core::str::from_utf8(buffer).unwrap());
            }
            len as isize
        },
        _ => {
            panic!("Unsupported fd in sys_write!");
        }
    }
}

pub fn sys_read(fd: usize, buf: *const u8, len: usize) -> isize {
    match fd {
        FD_STDIN => {
            assert_eq!(len, 1, "Only support len = 1 in sys_read!");
            let mut c: usize;
            loop {
                c = console_getchar();
                if c == 0 {
                    suspend_current_and_run_next();
                    continue;
                } else {
                    break;
                }
            }
            let ch = c as u8;
            let mut buffers = translated_byte_buffer(current_user_token(), buf, len);
            unsafe { buffers[0].as_mut_ptr().write_volatile(ch); }
            1
        }
        _ => {
            panic!("Unsupported fd in sys_read!");
        }
    }
}
~
```

3. Then, modify the "**os/src//syscall/process**". The rs implement other system calls.

```
//os/src/syscall/process.rs
use crate::task::{
    suspend_current_and_run_next,
    exit_current_and_run_next,
    current_task,
    current_user_token,
    add_task,
};

use crate::mm::{
    translated_str,
    translated_refmut,
};
use crate::loader::get_app_data_by_name;
use alloc::sync::Arc;

pub fn sys_getpid() -> isize {
    current_task().unwrap().pid.0 as isize
}

pub fn sys_fork() -> isize {
    let current_task = current_task().unwrap();
    let new_task = current_task.fork();
    let new_pid = new_task.pid.0;
    // modify trap context of new_task, because it returns immediately after
switching
    let trap_cx = new_task.inner_exclusive_access().get_trap_cx();
    // we do not have to move to next instruction since we have done it
before
    // for child process, fork returns 0
    trap_cx.x[10] = 0;
    // add new task to scheduler
    add_task(new_task);
    new_pid as isize
}

pub fn sys_exec(path: *const u8) -> isize {
    let token = current_user_token();
    let path = translated_str(token, path);
    if let Some(data) = get_app_data_by_name(path.as_str()) {
        let task = current_task().unwrap();
        task.exec(data);
        0
    } else {
        -1
    }
}

/// If there is not a child process whose pid is same as given, return -1.
/// Else if there is a child process but it is still running, return -2.
pub fn sys_waitpid(pid: isize, exit_code_ptr: *mut i32) -> isize {
    let task = current_task().unwrap();
    // find a child process

    // ---- access current TCB exclusively
    let mut inner = task.inner_exclusive_access();
    if inner.children
        .iter()
        .find(|p| {pid == -1 || pid as usize == p.getpid()})
        .is_none() {
        return -1;
        // ---- release current PCB
    }
    let pair = inner.children
        .iter()
        .enumerate()
        .find(|(_, p)| {
            // ++++ temporarily access child PCB lock exclusively
            p.inner_exclusive_access().is_zombie() && (pid == -1 || pid as
usize == p.getpid())
            // ++++ release child PCB
        });
    if let Some((idx, _)) = pair {
        let child = inner.children.remove(idx);
        // confirm that child will be deallocated after removing from
children list
        assert_eq!(Arc::strong_count(&child), 1);
        let found_pid = child.getpid();
        // ++++ temporarily access child TCB exclusively
        let exit_code = child.inner_exclusive_access().exit_code;
        // ++++ release child PCB
        *translated_refmut(inner.memory_set.token(), exit_code_ptr) =
exit_code;
        found_pid as isize
    } else {
        -2
    }
    // ---- release current PCB lock automatically
}
```



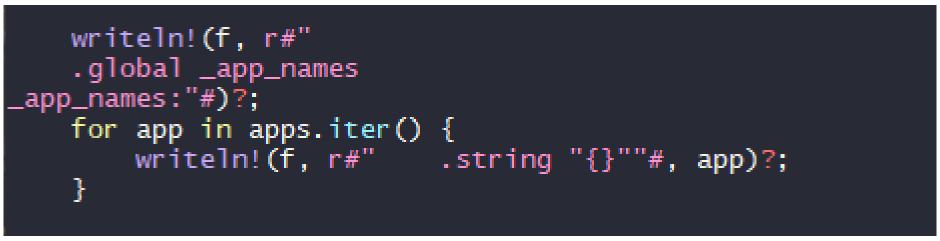
```
MINGW64:/c/Users/Dal-Z41/Desktop/课件&作业/Operation System/experiment/code/gardeneros/os/src/syscall
use crate::task::{
    suspend_current_and_run_next,
    exit_current_and_run_next,
    current_task,
    current_user_token,
    add_task,
};

use crate::timer::get_time_ms;
use crate::mm::{
    translated_str,
    translated_refmut,
};
use crate::loader::get_app_data_by_name;
use alloc::sync::Arc;

pub fn sys_exit(exit_code: i32) -> ! {
    exit_current_and_run_next(exit_code);
    panic!("Unreachable in sys_exit!");
}

pub fn sys_yield() -> isize {
    suspend_current_and_run_next();
    0
}

pub fn sys_get_time() -> isize {
    get_time_ms() as isize
}

pub fn sys_getpid() -> isize {
    current_task().unwrap().pid.0 as isize
}

pub fn sys_fork() -> isize {
    let current_task = current_task().unwrap();
    let new_task = current_task.fork();
    let new_pid = new_task.pid.0;
    // modify trap context of new_task, because it returns immediately after switching
    let trap_cx = new_task.inner_exclusive_access().get_trap_cx();
    // we do not have to move to next instruction since we have done it before
    // for child process, fork returns 0
    trap_cx.x[10] = 0;
    // add new task to scheduler
    add_task(new_task);
    new_pid as isize
}

pub fn sys_exec(path: *const u8) -> isize {
    let token = current_user_token();
    let path = translated_str(token, path);
    if let Some(data) = get_app_data_by_name(path.as_str()) {
        let task = current_task().unwrap();
        task.exec(data);
        0
    } else {
        -1
    }
}

/// If there is not a child process whose pid is same as given, return -1.
/// Else if there is a child process but it is still running, return -2.
pub fn sys_waitpid(pid: isize, exit_code_ptr: *mut i32) -> isize {
    let task = current_task().unwrap();
    // find a child process
process.rs [dos] (13:25 14/11/2021)
"process.rs" [noeol][dos] 98L, 3035B
```

3. 应用链接与加载

3.1 Link

Since implementing the EXEC system call requires retrieving data in ELF format based on the name of the application, you need to modify the link and load interface. Hence, we need to ,odify the build link auxiliary file "**os /build.rs**".

```
//os/build.rs
writeln!(f, r#"
    .global _app_names
_app_names:"#)?;
    for app in apps.iter() {
        writeln!(f, r#"    .string "{}""#, app)?;
    }
```

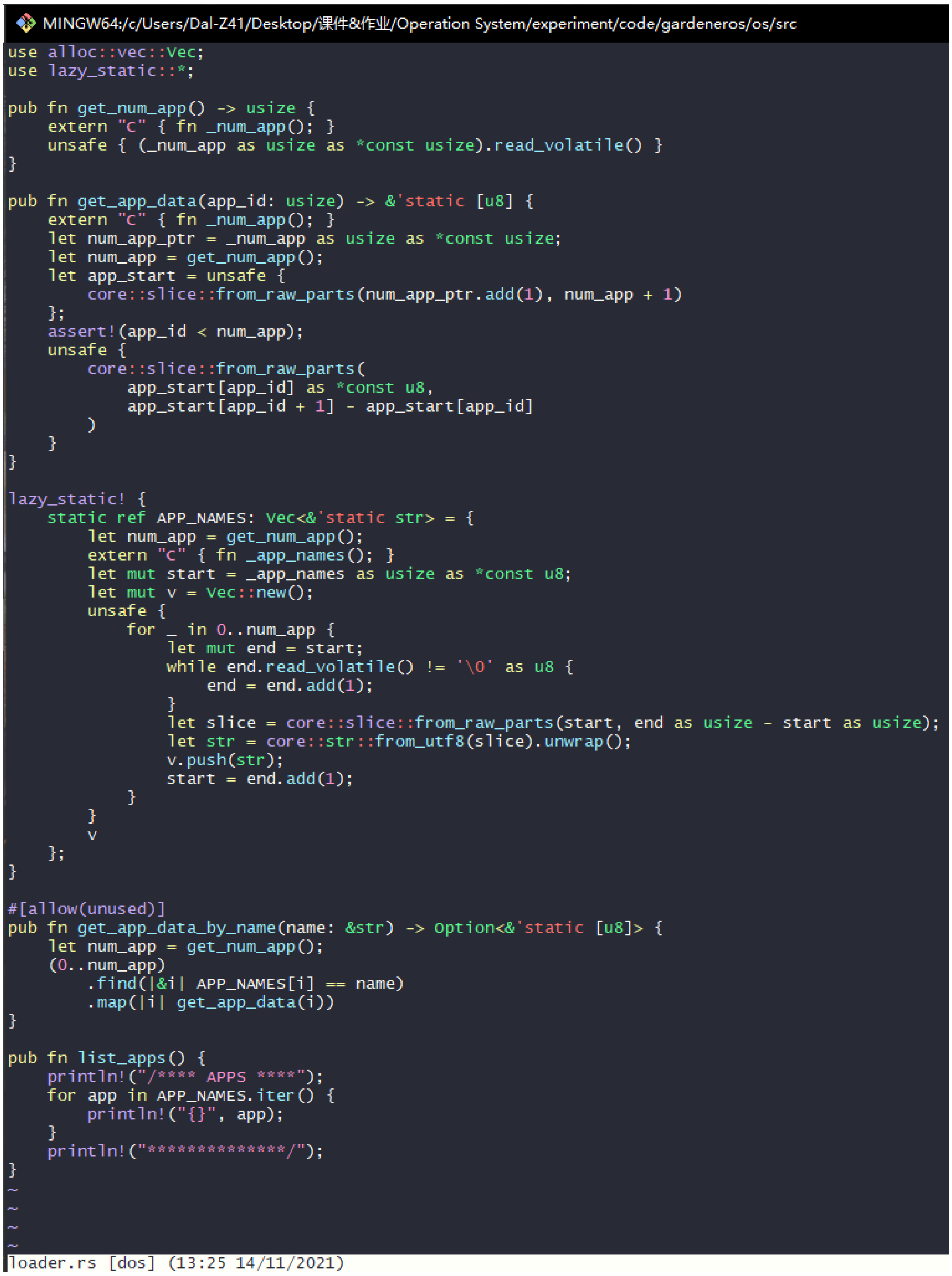


```
    writeln!(f, r#"
    .global _app_names
_app_names:"#)?;
    for app in apps.iter() {
        writeln!(f, r#"    .string "{}""#, app)?;
    }
```

3.2 Load

The application loading submodule "**loader.rs**" stores all application names in memory in order with a globally visible read-only vector APP_NAMES.

```
//os/src/loader.rs
use alloc::vec::Vec;
use lazy_static::*;

lazy_static! {
    static ref APP_NAMES: Vec<&'static str> = {
        let num_app = get_num_app();
        extern "C" { fn _app_names(); }
        let mut start = _app_names as usize as *const u8;
        let mut v = Vec::new();
        unsafe {
            for _ in 0..num_app {
                let mut end = start;
                while end.read_volatile() != '\0' as u8 {
                    end = end.add(1);
                }
                let slice = core::slice::from_raw_parts(start, end as usize -
start as usize);
                let str = core::str::from_utf8(slice).unwrap();
                v.push(str);
                start = end.add(1);
            }
        }
        v
    };
}

#[allow(unused)]
pub fn get_app_data_by_name(name: &str) -> Option<&'static [u8]> {
    let num_app = get_num_app();
    (0..num_app)
        .find(|&i| APP_NAMES[i] == name)
        .map(|i| get_app_data(i))
}

pub fn list_apps() {
    println!("/**** APPS ****");
    for app in APP_NAMES.iter() {
        println!("{}", app);
    }
    println!("**************/");
}
```



```
MINGW64:/c/Users/Dal-Z41/Desktop/课件&作业/Operation System/experiment/code/gardeneros/os/src
use alloc::vec::Vec;
use lazy_static::*;

pub fn get_num_app() -> usize {
    extern "C" { fn _num_app(); }
    unsafe { (_num_app as usize as *const usize).read_volatile() }
}

pub fn get_app_data(app_id: usize) -> &'static [u8] {
    extern "C" { fn _num_app(); }
    let num_app_ptr = _num_app as usize as *const usize;
    let num_app = get_num_app();
    let app_start = unsafe {
        core::slice::from_raw_parts(num_app_ptr.add(1), num_app + 1)
    };
    assert!(app_id < num_app);
    unsafe {
        core::slice::from_raw_parts(
            app_start[app_id] as *const u8,
            app_start[app_id + 1] - app_start[app_id]
        )
    }
}

lazy_static! {
    static ref APP_NAMES: Vec<&'static str> = {
        let num_app = get_num_app();
        extern "C" { fn _app_names(); }
        let mut start = _app_names as usize as *const u8;
        let mut v = Vec::new();
        unsafe {
            for _ in 0..num_app {
                let mut end = start;
                while end.read_volatile() != '\0' as u8 {
                    end = end.add(1);
                }
                let slice = core::slice::from_raw_parts(start, end as usize - start as usize);
                let str = core::str::from_utf8(slice).unwrap();
                v.push(str);
                start = end.add(1);
            }
        }
        v
    };
}

#[allow(unused)]
pub fn get_app_data_by_name(name: &str) -> Option<&'static [u8]> {
    let num_app = get_num_app();
    (0..num_app)
        .find(|&i| APP_NAMES[i] == name)
        .map(|i| get_app_data(i))
}

pub fn list_apps() {
    println!("/**** APPS ****");
    for app in APP_NAMES.iter() {
        println!("{}", app);
    }
    println!("**************/");
}
~
~
~
~
loader.rs [dos] (13:25 14/11/2021)
```

4. 进程标识符与内核栈

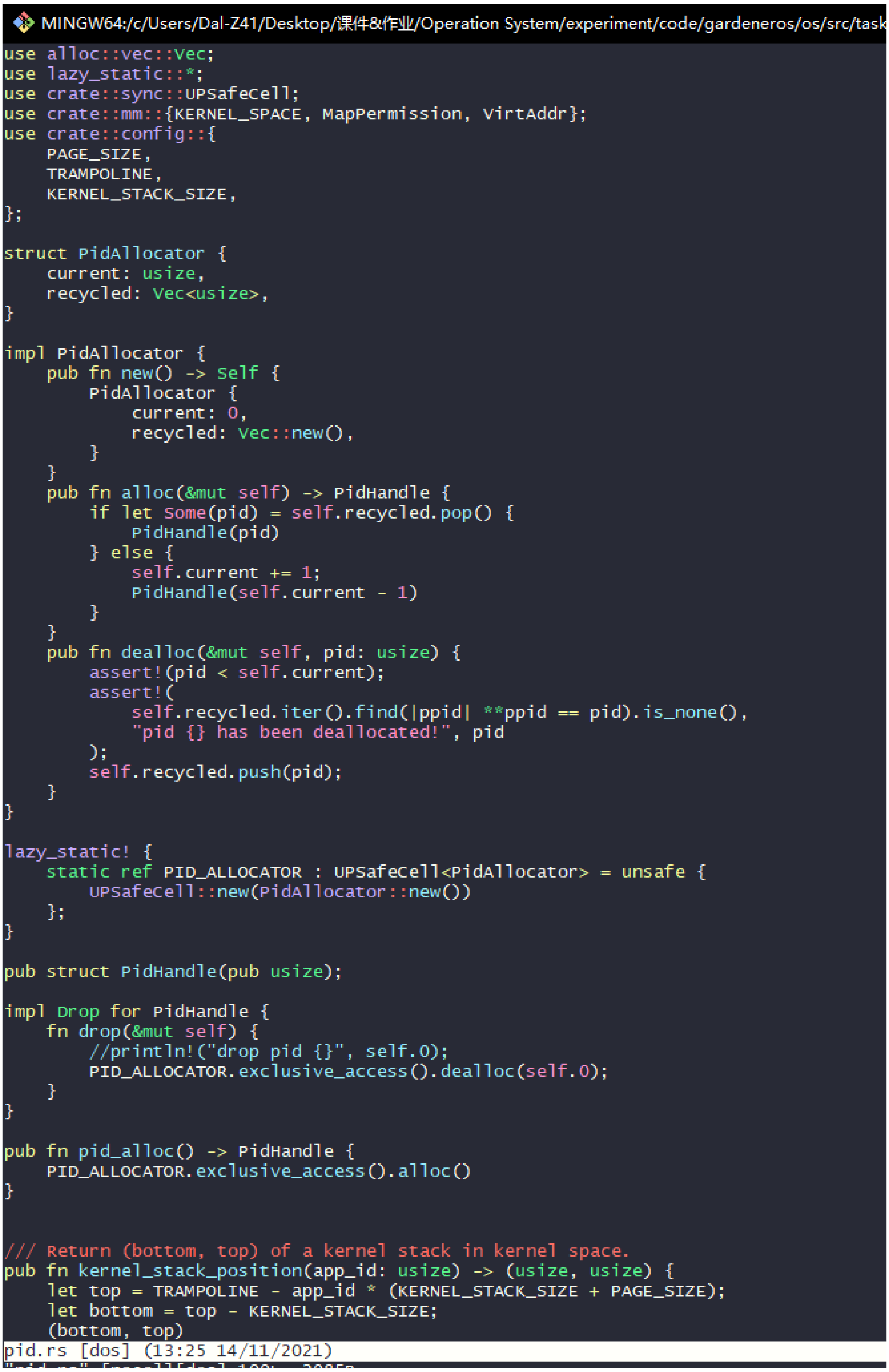
4.1 PID

The process id should be unique, which we abstracted as a **"PidHandle"** type. Similar to the previous management of physical page frames, we implement a process identifier allocator **"PID_ALLOCATOR"**.

```
//os/src/task/pid.rs
pub struct PidHandle(pub usize);

//os/src/task/pid.rs
struct PidAllocator {
    current: usize,
    recycled: Vec<usize>,
}

impl PidAllocator {
    pub fn new() -> Self {
        PidAllocator {
            current: 0,
            recycled: Vec::new(),
        }
    }
    pub fn alloc(&mut self) -> PidHandle {
        if let Some(pid) = self.recycled.pop() {
            PidHandle(pid)
        } else {
            self.current += 1;
            PidHandle(self.current - 1)
        }
    }
    pub fn dealloc(&mut self, pid: usize) {
        assert!(pid < self.current);
        assert!(
            self.recycled.iter().find(|ppid| **ppid == pid).is_none(),
            "pid {} has been deallocated!", pid
        );
        self.recycled.push(pid);
    }
}

lazy_static! {
    static ref PID_ALLOCATOR : UPSafeCell<PidAllocator> = unsafe {
        UPSafeCell::new(PidAllocator::new())
    };
}
```

Besides, we also need to encapsulate a global process identity assignment interface, **"pid_alloc"**.

```
//os/src/task/pid.rs
pub fn pid_alloc() -> PidHandle {
    PID_ALLOCATOR.exclusive_access().alloc()
}

//os/src/task/pid.rs
impl Drop for PidHandle {
    fn drop(&mut self) {
        //println!("drop pid {}", self.0);
        PID_ALLOCATOR.exclusive_access().dealloc(self.0);
    }
}
```


MINGW64:/c/Users/Dal-Z41/Desktop/课件&作业/Operation System/experiment/code/gardeneros/os/src/task

```
use alloc::vec::Vec;
use lazy_static::*;
use crate::sync::UPSafeCell;
use crate::mm::{KERNEL_SPACE, MapPermission, VirtAddr};
use crate::config::{
    PAGE_SIZE,
    TRAMPOLINE,
    KERNEL_STACK_SIZE,
};

struct PidAllocator {
    current: usize,
    recycled: Vec<usize>,
}

impl PidAllocator {
    pub fn new() -> Self {
        PidAllocator {
            current: 0,
            recycled: Vec::new(),
        }
    }
    pub fn alloc(&mut self) -> PidHandle {
        if let Some(pid) = self.recycled.pop() {
            PidHandle(pid)
        } else {
            self.current += 1;
            PidHandle(self.current - 1)
        }
    }
    pub fn dealloc(&mut self, pid: usize) {
        assert!(pid < self.current);
        assert!(
            self.recycled.iter().find(|ppid| **ppid == pid).is_none(),
            "pid {} has been deallocated!", pid
        );
        self.recycled.push(pid);
    }
}

lazy_static! {
    static ref PID_ALLOCATOR : UPSafeCell<PidAllocator> = unsafe {
        UPSafeCell::new(PidAllocator::new())
    };
}

pub struct PidHandle(pub usize);

impl Drop for PidHandle {
    fn drop(&mut self) {
        //println!("drop pid {}", self.0);
        PID_ALLOCATOR.exclusive_access().dealloc(self.0);
    }
}

pub fn pid_alloc() -> PidHandle {
    PID_ALLOCATOR.exclusive_access().alloc()
}


/// Return (bottom, top) of a kernel stack in kernel space.
pub fn kernel_stack_position(app_id: usize) -> (usize, usize) {
    let top = TRAMPOLINE - app_id * (KERNEL_STACK_SIZE + PAGE_SIZE);
    let bottom = top - KERNEL_STACK_SIZE;
    (bottom, top)
pid.rs [dos] (13:25 14/11/2021)
```

4.2 Kernel Stack

1. Redefine the kernel stack.

```
//os/src/task/pid.rs
use alloc::vec::Vec;
use lazy_static::*;
use crate::sync::UPSafeCell;
use crate::mm::{KERNEL_SPACE, MapPermission, VirtAddr};
use crate::config::{
    PAGE_SIZE,
    TRAMPOLINE,
    KERNEL_STACK_SIZE,
};

pub struct KernelStack {
    pid: usize,
}

/// Return (bottom, top) of a kernel stack in kernel space.
pub fn kernel_stack_position(app_id: usize) -> (usize, usize) {
    let top = TRAMPOLINE - app_id * (KERNEL_STACK_SIZE + PAGE_SIZE);
    let bottom = top - KERNEL_STACK_SIZE;
    (bottom, top)
}

impl KernelStack {
    pub fn new(pid_handle: &PidHandle) -> Self {
        let pid = pid_handle.0;
        let (kernel_stack_bottom, kernel_stack_top) =
kernel_stack_position(pid);
        KERNEL_SPACE
            .exclusive_access()
            .insert_framed_area(
                kernel_stack_bottom.into(),
                kernel_stack_top.into(),
                MapPermission::R | MapPermission::W,
            );
        KernelStack {
            pid: pid_handle.0,
        }
    }
    #[allow(unused)]
    pub fn push_on_top<T>(&self, value: T) -> *mut T where
        T: Sized, {
        let kernel_stack_top = self.get_top();
        let ptr_mut = (kernel_stack_top - core::mem::size_of::<T>()) as *mut
T;
        unsafe { *ptr_mut = value; }
        ptr_mut
    }
    pub fn get_top(&self) -> usize {
        let (_, kernel_stack_top) = kernel_stack_position(self.pid);
        kernel_stack_top
    }
}
```

2. Implement KernelStack's **Drop Trait** so that the corresponding physical page frame is reclaimed at the end of the KernelStack's life cycle.

```
impl Drop for KernelStack {
    fn drop(&mut self) {
        let (kernel_stack_bottom, _) = kernel_stack_position(self.pid);
        let kernel_stack_bottom_va: VirtAddr = kernel_stack_bottom.into();
        KERNEL_SPACE
            .exclusive_access()
            .remove_area_with_start_vpn(kernel_stack_bottom_va.into());
    }
}
```

3. Accordingly, "**os / src /mm/memory_set.rs**" needs to be modified.

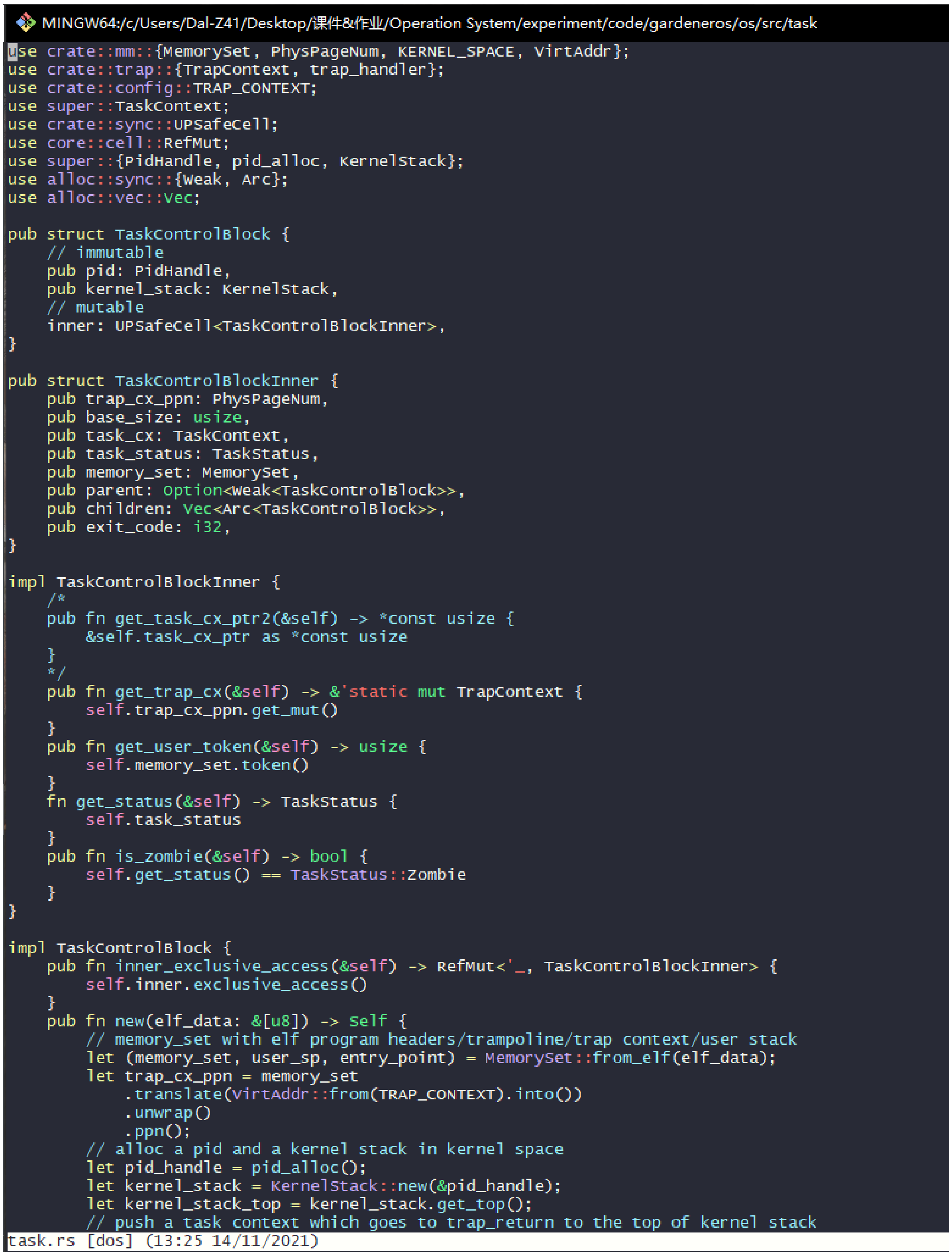
```
impl MemorySet {
    pub fn remove_area_with_start_vpn(&mut self, start_vpn: VirtPageNum) {
        if let Some((idx, area)) = self.areas.iter_mut().enumerate()
            .find(|(_, area)| area.vpn_range.get_start() == start_vpn) {
            area.unmap(&mut self.page_table);
            self.areas.remove(idx);
        }
    }
}
```

5. 修改实现PCB

Modified TaskControlBlock to implement process control block functionality. Change the TaskStatus at the same time.

```
//os/src/task/task.rs
pub struct TaskControlBlock {
    // immutable
    pub pid: PidHandle,
    pub kernel_stack: KernelStack,
    // mutable
    inner: UPSafeCell<TaskControlBlockInner>,
}

pub struct TaskControlBlockInner {
    pub trap_cx_ppn: PhysPageNum,
    pub base_size: usize,
    pub task_cx: TaskContext,
    pub task_status: TaskStatus,
    pub memory_set: MemorySet,
    pub parent: Option<Weak<TaskControlBlock>>,
    pub children: Vec<Arc<TaskControlBlock>>,
    pub exit_code: i32,
}

impl TaskControlBlockInner {
    /*
    pub fn get_task_cx_ptr2(&self) -> *const usize {
        &self.task_cx_ptr as *const usize
    }
    */
```

```rust
    pub fn get_trap_cx(&self) -> &'static mut TrapContext {
        self.trap_cx_ppn.get_mut()
    }
    pub fn get_user_token(&self) -> usize {
        self.memory_set.token()
    }
    fn get_status(&self) -> TaskStatus {
        self.task_status
    }
    pub fn is_zombie(&self) -> bool {
        self.get_status() == TaskStatus::Zombie
    }
}

impl TaskControlBlock {
    pub fn inner_exclusive_access(&self) -> RefMut<'_, TaskControlBlockInner> {
        self.inner.exclusive_access()
    }
    pub fn new(elf_data: &[u8]) -> Self {
        // memory_set with elf program headers/trampoline/trap context/user
stack
        let (memory_set, user_sp, entry_point) = MemorySet::from_elf(elf_data);
        let trap_cx_ppn = memory_set
            .translate(VirtAddr::from(TRAP_CONTEXT).into())
            .unwrap()
            .ppn();
        // alloc a pid and a kernel stack in kernel space
        let pid_handle = pid_alloc();
        let kernel_stack = KernelStack::new(&pid_handle);
        let kernel_stack_top = kernel_stack.get_top();
        // push a task context which goes to trap_return to the top of kernel
stack
        let task_control_block = Self {
            pid: pid_handle,
            kernel_stack,
            inner: unsafe { UPSafeCell::new(TaskControlBlockInner {
                trap_cx_ppn,
                base_size: user_sp,
                task_cx: TaskContext::goto_trap_return(kernel_stack_top),
                task_status: TaskStatus::Ready,
                memory_set,
                parent: None,
                children: Vec::new(),
                exit_code: 0,
            })},
        };
        // prepare TrapContext in user space
        let trap_cx = task_control_block.inner_exclusive_access().get_trap_cx();
        *trap_cx = TrapContext::app_init_context(
            entry_point,
            user_sp,
            KERNEL_SPACE.exclusive_access().token(),
            kernel_stack_top,
            trap_handler as usize,
        );
        task_control_block
    }
    pub fn exec(&self, elf_data: &[u8]) {
        // memory_set with elf program headers/trampoline/trap context/user
stack
        let (memory_set, user_sp, entry_point) = MemorySet::from_elf(elf_data);
        let trap_cx_ppn = memory_set
            .translate(VirtAddr::from(TRAP_CONTEXT).into())
            .unwrap()
            .ppn();

        // **** access inner exclusively
        let mut inner = self.inner_exclusive_access();
        // substitute memory_set
        inner.memory_set = memory_set;
        // update trap_cx ppn
        inner.trap_cx_ppn = trap_cx_ppn;
        // initialize trap_cx
        let trap_cx = inner.get_trap_cx();
        *trap_cx = TrapContext::app_init_context(
            entry_point,
            user_sp,
            KERNEL_SPACE.exclusive_access().token(),
            self.kernel_stack.get_top(),
            trap_handler as usize,
        );
        // **** release inner automatically
    }
    pub fn fork(self: &Arc<TaskControlBlock>) -> Arc<TaskControlBlock> {
        // ---- access parent PCB exclusively
        let mut parent_inner = self.inner_exclusive_access();
        // copy user space(include trap context)
        let memory_set = MemorySet::from_existed_user(
            &parent_inner.memory_set
        );
        let trap_cx_ppn = memory_set
            .translate(VirtAddr::from(TRAP_CONTEXT).into())
            .unwrap()
            .ppn();
        // alloc a pid and a kernel stack in kernel space
        let pid_handle = pid_alloc();
        let kernel_stack = KernelStack::new(&pid_handle);
        let kernel_stack_top = kernel_stack.get_top();
        let task_control_block = Arc::new(TaskControlBlock {
            pid: pid_handle,
            kernel_stack,
            inner: unsafe { UPSafeCell::new(TaskControlBlockInner {
                trap_cx_ppn,
                base_size: parent_inner.base_size,
                task_cx: TaskContext::goto_trap_return(kernel_stack_top),
                task_status: TaskStatus::Ready,
                memory_set,
                parent: Some(Arc::downgrade(self)),
                children: Vec::new(),
                exit_code: 0,
            })},
        });
        // add child
        parent_inner.children.push(task_control_block.clone());
        // modify kernel_sp in trap_cx
        // **** access children PCB exclusively
        let trap_cx = task_control_block.inner_exclusive_access().get_trap_cx();
        trap_cx.kernel_sp = kernel_stack_top;
        // return
        task_control_block
        // ---- release parent PCB automatically
        // **** release children PCB automatically
    }
    pub fn getpid(&self) -> usize {
        self.pid.0
    }
}

#[derive(Copy, Clone, PartialEq)]
pub enum TaskStatus {
    Ready,
    Running,
    Zombie,
}
```


MINGW64:/c/Users/Dal-Z41/Desktop/课件&作业/Operation System/experiment/code/gardeneros/os/src/task

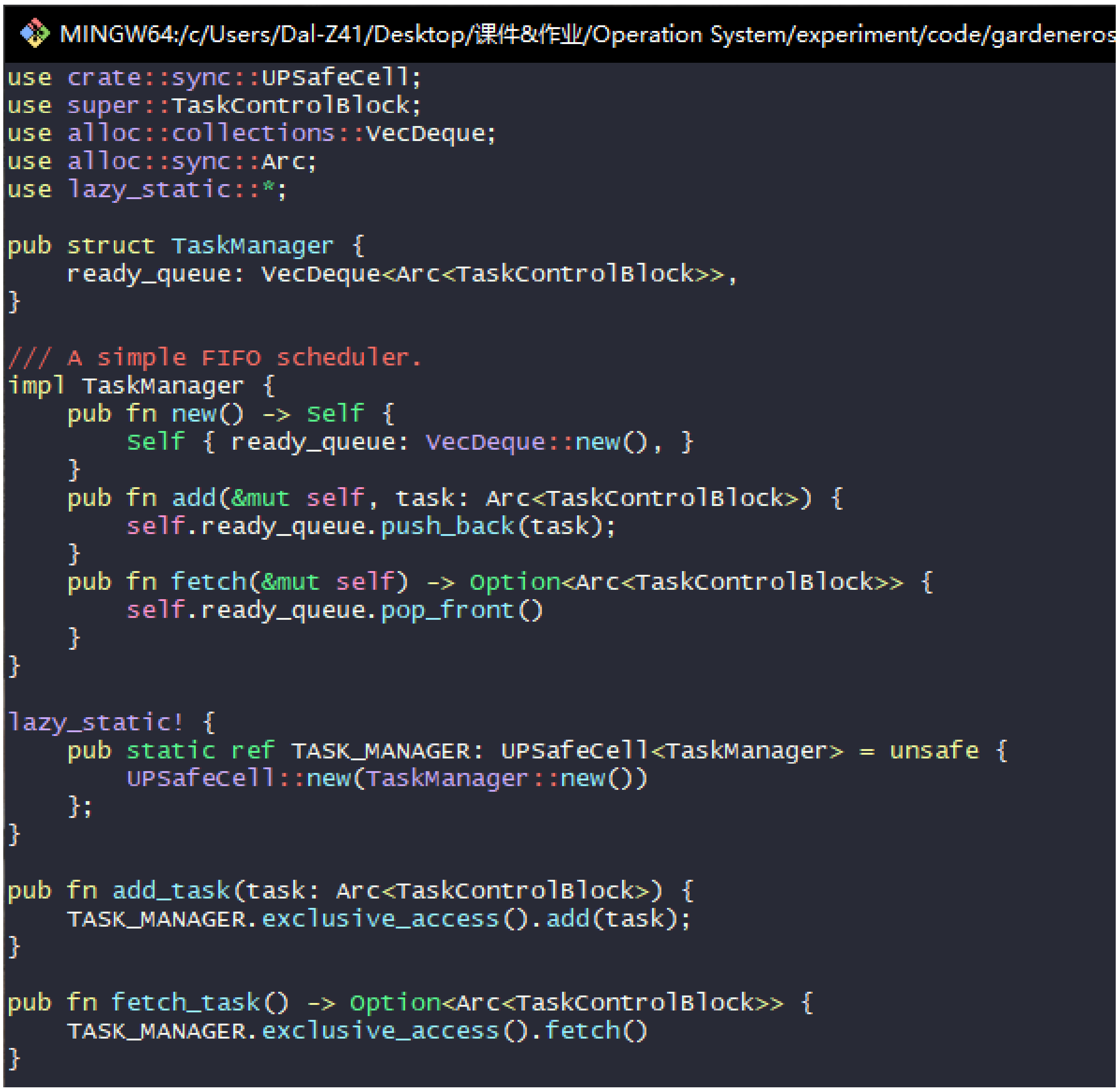
```rust
use crate::mm::{MemorySet, PhysPageNum, KERNEL_SPACE, VirtAddr};
use crate::trap::{TrapContext, trap_handler};
use crate::config::TRAP_CONTEXT;
use super::TaskContext;
use crate::sync::UPSafeCell;
use core::cell::RefMut;
use super::{PidHandle, pid_alloc, KernelStack};
use alloc::sync::{Weak, Arc};
use alloc::vec::Vec;

pub struct TaskControlBlock {
    // immutable
    pub pid: PidHandle,
    pub kernel_stack: KernelStack,
    // mutable
    inner: UPSafeCell<TaskControlBlockInner>,
}

pub struct TaskControlBlockInner {
    pub trap_cx_ppn: PhysPageNum,
    pub base_size: usize,
    pub task_cx: TaskContext,
    pub task_status: TaskStatus,
    pub memory_set: MemorySet,
    pub parent: Option<Weak<TaskControlBlock>>,
    pub children: Vec<Arc<TaskControlBlock>>,
    pub exit_code: i32,
}

impl TaskControlBlockInner {
    /*
    pub fn get_task_cx_ptr2(&self) -> *const usize {
        &self.task_cx_ptr as *const usize
    }
    */
    pub fn get_trap_cx(&self) -> &'static mut TrapContext {
        self.trap_cx_ppn.get_mut()
    }
    pub fn get_user_token(&self) -> usize {
        self.memory_set.token()
    }
    fn get_status(&self) -> TaskStatus {
        self.task_status
    }
    pub fn is_zombie(&self) -> bool {
        self.get_status() == TaskStatus::Zombie
    }
}

impl TaskControlBlock {
    pub fn inner_exclusive_access(&self) -> RefMut<'_, TaskControlBlockInner> {
        self.inner.exclusive_access()
    }
    pub fn new(elf_data: &[u8]) -> Self {
        // memory_set with elf program headers/trampoline/trap context/user stack
        let (memory_set, user_sp, entry_point) = MemorySet::from_elf(elf_data);
        let trap_cx_ppn = memory_set
            .translate(VirtAddr::from(TRAP_CONTEXT).into())
            .unwrap()
            .ppn();
        // alloc a pid and a kernel stack in kernel space
        let pid_handle = pid_alloc();
        let kernel_stack = KernelStack::new(&pid_handle);
        let kernel_stack_top = kernel_stack.get_top();
        // push a task context which goes to trap_return to the top of kernel stack
task.rs [dos] (13:25 14/11/2021)
```

6. 实现任务管理器

Modify task manager to move some task management functions to processor management.

```rust
//os/src/task/manager.rs
use crate::sync::UPSafeCell;
use super::TaskControlBlock;
use alloc::collections::VecDeque;
use alloc::sync::Arc;
use lazy_static::*;
```

```rust
pub struct TaskManager {
    ready_queue: VecDeque<Arc<TaskControlBlock>>,
}

/// A simple FIFO scheduler.
impl TaskManager {
    pub fn new() -> Self {
        Self { ready_queue: VecDeque::new(), }
    }
    pub fn add(&mut self, task: Arc<TaskControlBlock>) {
        self.ready_queue.push_back(task);
    }
    pub fn fetch(&mut self) -> Option<Arc<TaskControlBlock>> {
        self.ready_queue.pop_front()
    }
}

lazy_static! {
    pub static ref TASK_MANAGER: UPSafeCell<TaskManager> = unsafe {
        UPSafeCell::new(TaskManager::new())
    };
}

pub fn add_task(task: Arc<TaskControlBlock>) {
    TASK_MANAGER.exclusive_access().add(task);
}

pub fn fetch_task() -> Option<Arc<TaskControlBlock>> {
    TASK_MANAGER.exclusive_access().fetch()
}
```


MINGW64:/c/Users/Dal-Z41/Desktop/课件&作业/Operation System/experiment/code/gardeneros

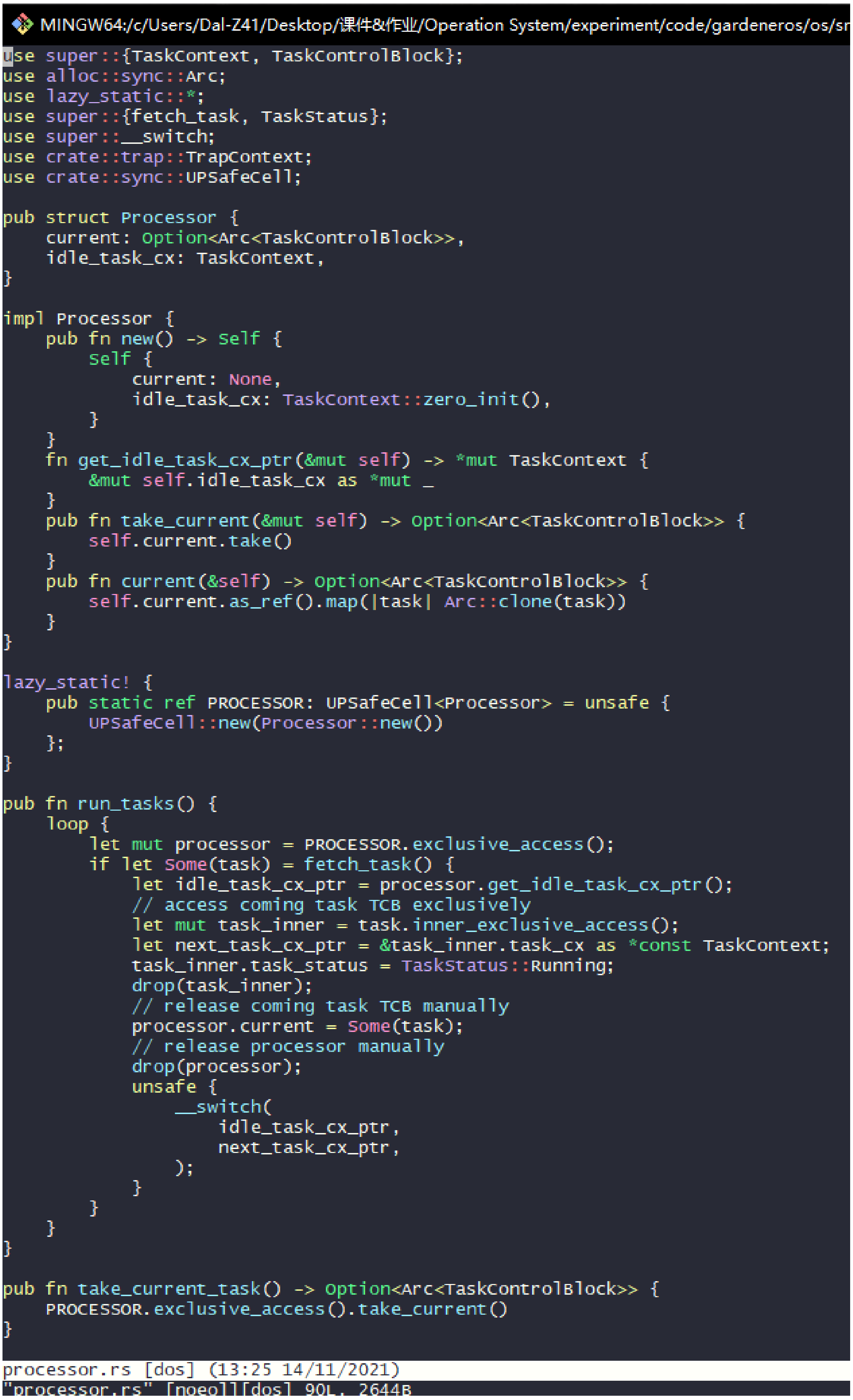
```rust
use crate::sync::UPSafeCell;
use super::TaskControlBlock;
use alloc::collections::VecDeque;
use alloc::sync::Arc;
use lazy_static::*;

pub struct TaskManager {
    ready_queue: VecDeque<Arc<TaskControlBlock>>,
}

/// A simple FIFO scheduler.
impl TaskManager {
    pub fn new() -> Self {
        Self { ready_queue: VecDeque::new(), }
    }
    pub fn add(&mut self, task: Arc<TaskControlBlock>) {
        self.ready_queue.push_back(task);
    }
    pub fn fetch(&mut self) -> Option<Arc<TaskControlBlock>> {
        self.ready_queue.pop_front()
    }
}

lazy_static! {
    pub static ref TASK_MANAGER: UPSafeCell<TaskManager> = unsafe {
        UPSafeCell::new(TaskManager::new())
    };
}

pub fn add_task(task: Arc<TaskControlBlock>) {
    TASK_MANAGER.exclusive_access().add(task);
}

pub fn fetch_task() -> Option<Arc<TaskControlBlock>> {
    TASK_MANAGER.exclusive_access().fetch()
}
```

7. 增加处理器管理结构

Implement the **Processor** management structure Processor, complete the separation from the task manager to maintain the CPU state of some functions.

```rust
//os/src/task/processor.rs
use super::{TaskContext, TaskControlBlock};
use alloc::sync::Arc;
use lazy_static::*;
use super::{fetch_task, TaskStatus};
use super::__switch;
use crate::trap::TrapContext;
use crate::sync::UPSafeCell;

pub struct Processor {
    current: Option<Arc<TaskControlBlock>>,
    idle_task_cx: TaskContext,
}

impl Processor {
    pub fn new() -> Self {
        Self {
            current: None,
            idle_task_cx: TaskContext::zero_init(),
        }
    }
    fn get_idle_task_cx_ptr(&mut self) -> *mut TaskContext {
        &mut self.idle_task_cx as *mut _
    }
    pub fn take_current(&mut self) -> Option<Arc<TaskControlBlock>> {
        self.current.take()
    }
    pub fn current(&self) -> Option<Arc<TaskControlBlock>> {
        self.current.as_ref().map(|task| Arc::clone(task))
    }
}

lazy_static! {
    pub static ref PROCESSOR: UPSafeCell<Processor> = unsafe {
        UPSafeCell::new(Processor::new())
    };
}

pub fn run_tasks() {
    loop {
        let mut processor = PROCESSOR.exclusive_access();
        if let Some(task) = fetch_task() {
            let idle_task_cx_ptr = processor.get_idle_task_cx_ptr();
            // access coming task TCB exclusively
            let mut task_inner = task.inner_exclusive_access();
            let next_task_cx_ptr = &task_inner.task_cx as *const TaskContext;
            task_inner.task_status = TaskStatus::Running;
            drop(task_inner);
            // release coming task TCB manually
            processor.current = Some(task);
            // release processor manually
            drop(processor);
            unsafe {
                __switch(
                    idle_task_cx_ptr,
                    next_task_cx_ptr,
                );
            }
        }
    }
}

pub fn take_current_task() -> Option<Arc<TaskControlBlock>> {
    PROCESSOR.exclusive_access().take_current()
}

pub fn current_task() -> Option<Arc<TaskControlBlock>> {
    PROCESSOR.exclusive_access().current()
}

pub fn current_user_token() -> usize {
    let task = current_task().unwrap();
    let token = task.inner_exclusive_access().get_user_token();
    token
}
```

```rust
pub fn current_trap_cx() -> &'static mut TrapContext {
    current_task().unwrap().inner_exclusive_access().get_trap_cx()
}

pub fn schedule(switched_task_cx_ptr: *mut TaskContext) {
    let mut processor = PROCESSOR.exclusive_access();
    let idle_task_cx_ptr = processor.get_idle_task_cx_ptr();
    drop(processor);
    unsafe {
        __switch(
            switched_task_cx_ptr,
            idle_task_cx_ptr,
        );
    }
}
```

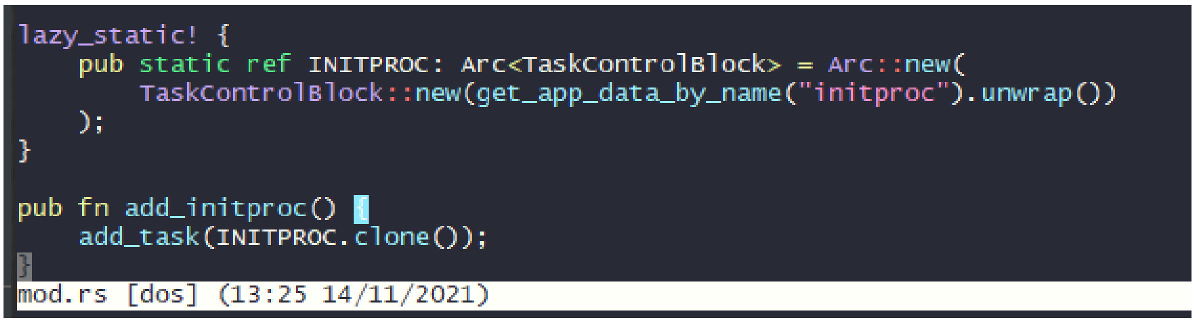

MINGW64:/c/Users/Dal-Z41/Desktop/课件&作业/Operation System/experiment/code/gardeneros/os/sr

```rust
use super::{TaskContext, TaskControlBlock};
use alloc::sync::Arc;
use lazy_static::*;
use super::{fetch_task, TaskStatus};
use super::__switch;
use crate::trap::TrapContext;
use crate::sync::UPSafeCell;

pub struct Processor {
    current: Option<Arc<TaskControlBlock>>,
    idle_task_cx: TaskContext,
}

impl Processor {
    pub fn new() -> Self {
        Self {
            current: None,
            idle_task_cx: TaskContext::zero_init(),
        }
    }
    fn get_idle_task_cx_ptr(&mut self) -> *mut TaskContext {
        &mut self.idle_task_cx as *mut _
    }
    pub fn take_current(&mut self) -> Option<Arc<TaskControlBlock>> {
        self.current.take()
    }
    pub fn current(&self) -> Option<Arc<TaskControlBlock>> {
        self.current.as_ref().map(|task| Arc::clone(task))
    }
}

lazy_static! {
    pub static ref PROCESSOR: UPSafeCell<Processor> = unsafe {
        UPSafeCell::new(Processor::new())
    };
}

pub fn run_tasks() {
    loop {
        let mut processor = PROCESSOR.exclusive_access();
        if let Some(task) = fetch_task() {
            let idle_task_cx_ptr = processor.get_idle_task_cx_ptr();
            // access coming task TCB exclusively
            let mut task_inner = task.inner_exclusive_access();
            let next_task_cx_ptr = &task_inner.task_cx as *const TaskContext;
            task_inner.task_status = TaskStatus::Running;
            drop(task_inner);
            // release coming task TCB manually
            processor.current = Some(task);
            // release processor manually
            drop(processor);
            unsafe {
                __switch(
                    idle_task_cx_ptr,
                    next_task_cx_ptr,
                );
            }
        }
    }
}

pub fn take_current_task() -> Option<Arc<TaskControlBlock>> {
    PROCESSOR.exclusive_access().take_current()
}
processor.rs [dos] (13:25 14/11/2021)
"processor.rs" [noeol][dos] 90L, 2644B
```

8. 创建初始进程

After kernel initialization, "**add_initproc**" of the **task** submodule is called to add initproc to the task manager. The process control block that initializes the process is initialized before doing so.

```
//os/src/task/mod.rs
use crate::loader::get_app_data_by_name;
use manager::add_task;

lazy_static! {
    pub static ref INITPROC: Arc<TaskControlBlock> = Arc::new(
        TaskControlBlock::new(get_app_data_by_name("initproc").unwrap())
    );
}

pub fn add_initproc() {
    add_task(INITPROC.clone());
}
```

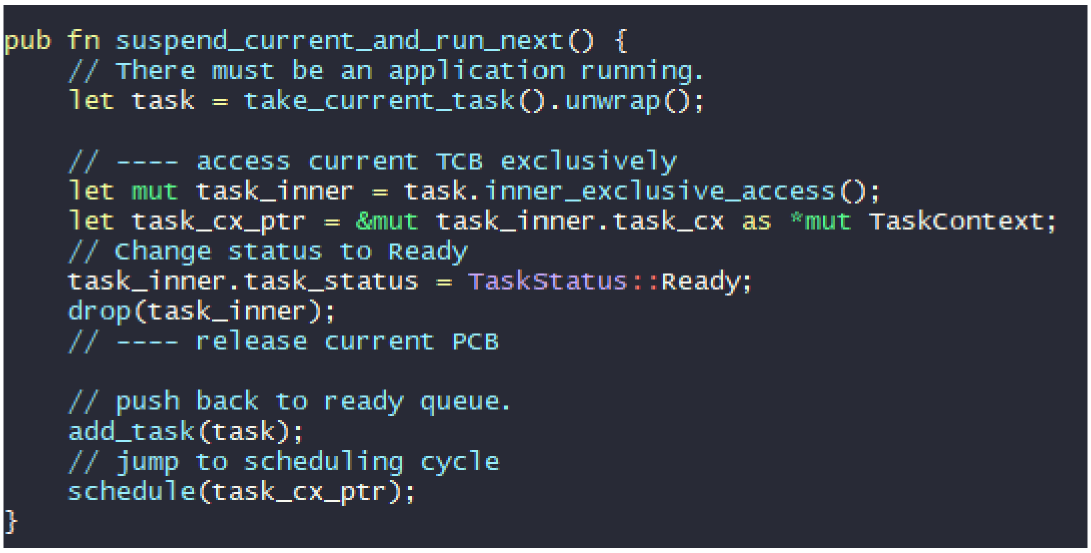


```
lazy_static! {
    pub static ref INITPROC: Arc<TaskControlBlock> = Arc::new(
        TaskControlBlock::new(get_app_data_by_name("initproc").unwrap())
    );
}

pub fn add_initproc() {
    add_task(INITPROC.clone());
}
mod.rs [dos] (13:25 14/11/2021)
```

9. 进程调度机制

Pause the current task and switch to another task by calling the "**suspend_current_and_run_next**" function provided by the **task** submodule. Because of the introduction of the process concept, its implementation needs to change.

```
//os/src/task/mod.rs
pub fn suspend_current_and_run_next() {
    // There must be an application running.
    let task = take_current_task().unwrap();

    // ---- access current TCB exclusively
    let mut task_inner = task.inner_exclusive_access();
    let task_cx_ptr = &mut task_inner.task_cx as *mut TaskContext;
    // Change status to Ready
    task_inner.task_status = TaskStatus::Ready;
    drop(task_inner);
    // ---- release current PCB

    // push back to ready queue.
    add_task(task);
    // jump to scheduling cycle
    schedule(task_cx_ptr);
}
```

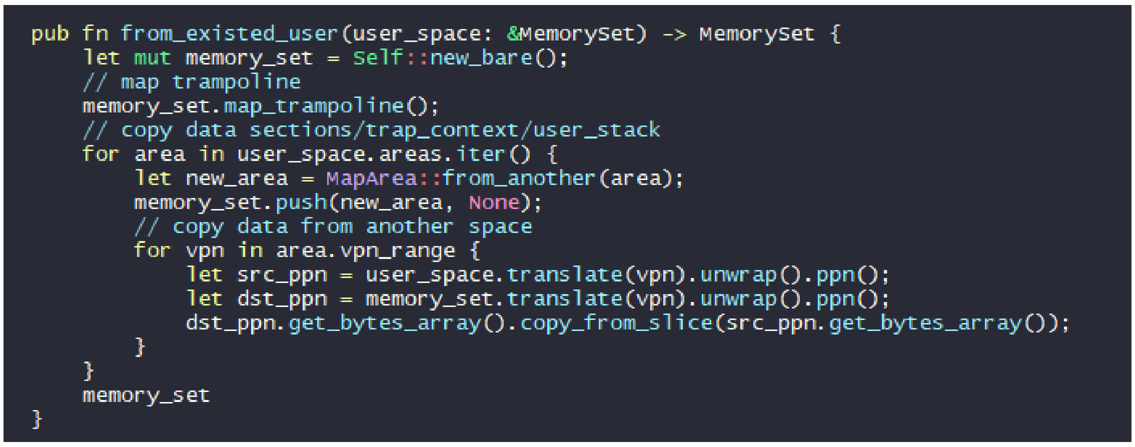


```
pub fn suspend_current_and_run_next() {
    // There must be an application running.
    let task = take_current_task().unwrap();

    // ---- access current TCB exclusively
    let mut task_inner = task.inner_exclusive_access();
    let task_cx_ptr = &mut task_inner.task_cx as *mut TaskContext;
    // Change status to Ready
    task_inner.task_status = TaskStatus::Ready;
    drop(task_inner);
    // ---- release current PCB

    // push back to ready queue.
    add_task(task);
    // jump to scheduling cycle
    schedule(task_cx_ptr);
}
```

10. 进程的生成机制

1. In the kernel, only the init process **initProc** is generated manually; the others are forked out directly or indirectly by the init process, which then calls the exec system call to load and execute the executable. Therefore, the process generation mechanism is accomplished by two system calls, **fork** and **exec**. The key to implementing fork is to create an address space for the child that is almost identical to that of the parent. The concrete implementation is as follows.

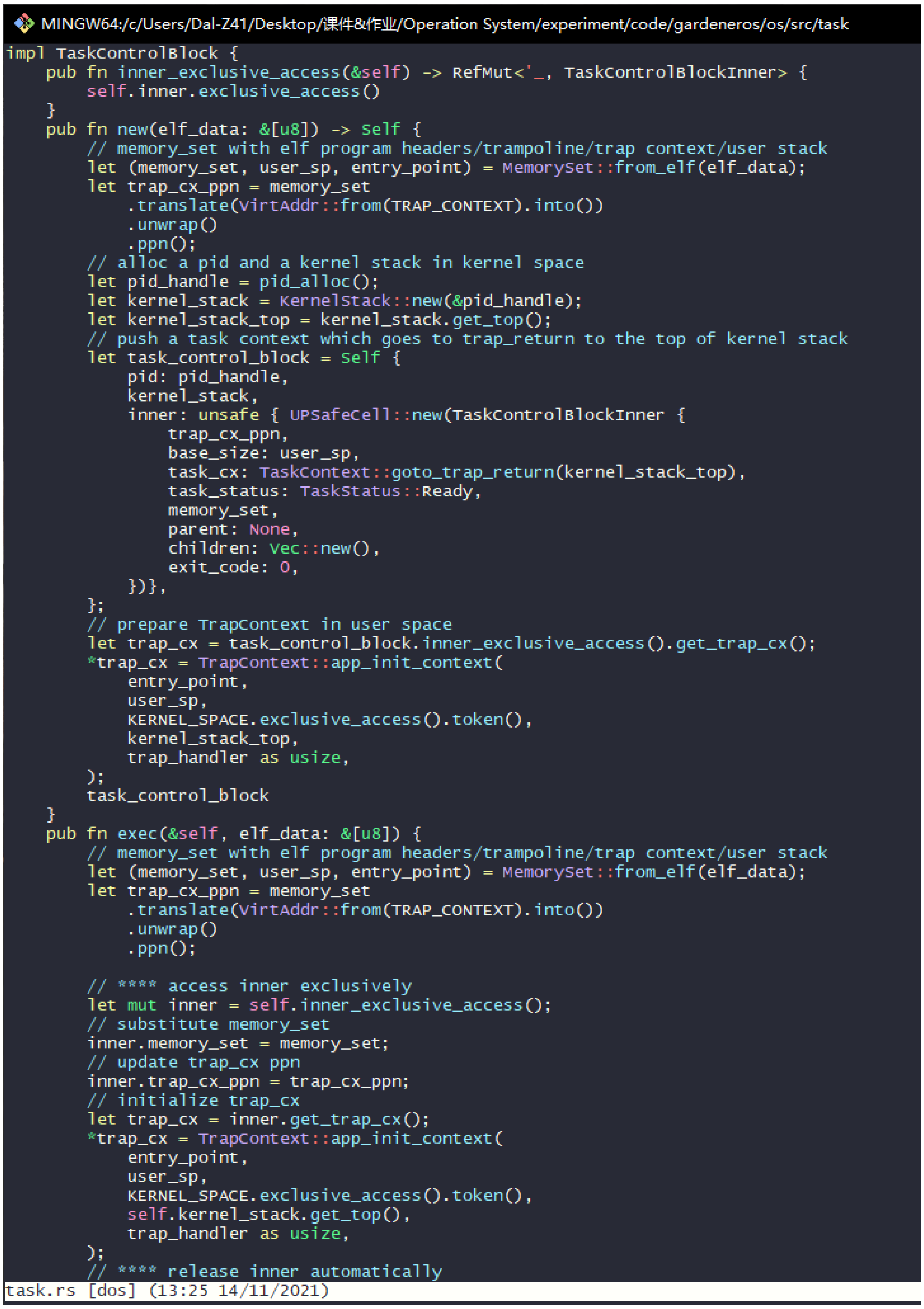
```
//os/src/mm/memory_set.rs
impl MapArea {
    pub fn from_another(another: &MapArea) -> Self {
        Self {
            vpn_range: VPNRange::new(another.vpn_range.get_start(),
another.vpn_range.get_end()),
            data_frames: BTreeMap::new(),
            map_type: another.map_type,
            map_perm: another.map_perm,
        }
    }
}

impl MemorySet {
    pub fn from_existed_user(user_space: &MemorySet) -> MemorySet {
        let mut memory_set = Self::new_bare();
        // map trampoline
        memory_set.map_trampoline();
        // copy data sections/trap_context/user_stack
        for area in user_space.areas.iter() {
            let new_area = MapArea::from_another(area);
            memory_set.push(new_area, None);
            // copy data from another space
            for vpn in area.vpn_range {
                let src_ppn = user_space.translate(vpn).unwrap().ppn();
                let dst_ppn = memory_set.translate(vpn).unwrap().ppn();

 dst_ppn.get_bytes_array().copy_from_slice(src_ppn.get_bytes_array());
            }
        }
        memory_set
    }
}
```



```
pub fn from_existed_user(user_space: &MemorySet) -> MemorySet {
    let mut memory_set = Self::new_bare();
    // map trampoline
    memory_set.map_trampoline();
    // copy data sections/trap_context/user_stack
    for area in user_space.areas.iter() {
        let new_area = MapArea::from_another(area);
        memory_set.push(new_area, None);
        // copy data from another space
        for vpn in area.vpn_range {
            let src_ppn = user_space.translate(vpn).unwrap().ppn();
            let dst_ppn = memory_set.translate(vpn).unwrap().ppn();
            dst_ppn.get_bytes_array().copy_from_slice(src_ppn.get_bytes_array());
        }
    }
    memory_set
}
```

2. Next, implement **TaskControlBlock::fork** to create a sub-process control block from the parent process control block.

```
//os/src/task/task.rs
impl TaskControlBlock {
        pub fn fork(self: &Arc<TaskControlBlock>) -> Arc<TaskControlBlock> {
        // ---- access parent PCB exclusively
        let mut parent_inner = self.inner_exclusive_access();
        // copy user space(include trap context)
        let memory_set = MemorySet::from_existed_user(
            &parent_inner.memory_set
        );
        let trap_cx_ppn = memory_set
            .translate(VirtAddr::from(TRAP_CONTEXT).into())
            .unwrap()
            .ppn();
        // alloc a pid and a kernel stack in kernel space
        let pid_handle = pid_alloc();
        let kernel_stack = KernelStack::new(&pid_handle);
        let kernel_stack_top = kernel_stack.get_top();
        let task_control_block = Arc::new(TaskControlBlock {
            pid: pid_handle,
            kernel_stack,
            inner: unsafe { UPSafeCell::new(TaskControlBlockInner {
                trap_cx_ppn,
                base_size: parent_inner.base_size,
                task_cx: TaskContext::goto_trap_return(kernel_stack_top),
                task_status: TaskStatus::Ready,
                memory_set,
                parent: Some(Arc::downgrade(self)),
                children: Vec::new(),
                exit_code: 0,
            })},
        });
        // add child
        parent_inner.children.push(task_control_block.clone());
        // modify kernel_sp in trap_cx
        // **** access children PCB exclusively
        let trap_cx =
task_control_block.inner_exclusive_access().get_trap_cx();
        trap_cx.kernel_sp = kernel_stack_top;
        // return
        task_control_block
        // ---- release parent PCB automatically
        // **** release children PCB automatically
    }
}
```

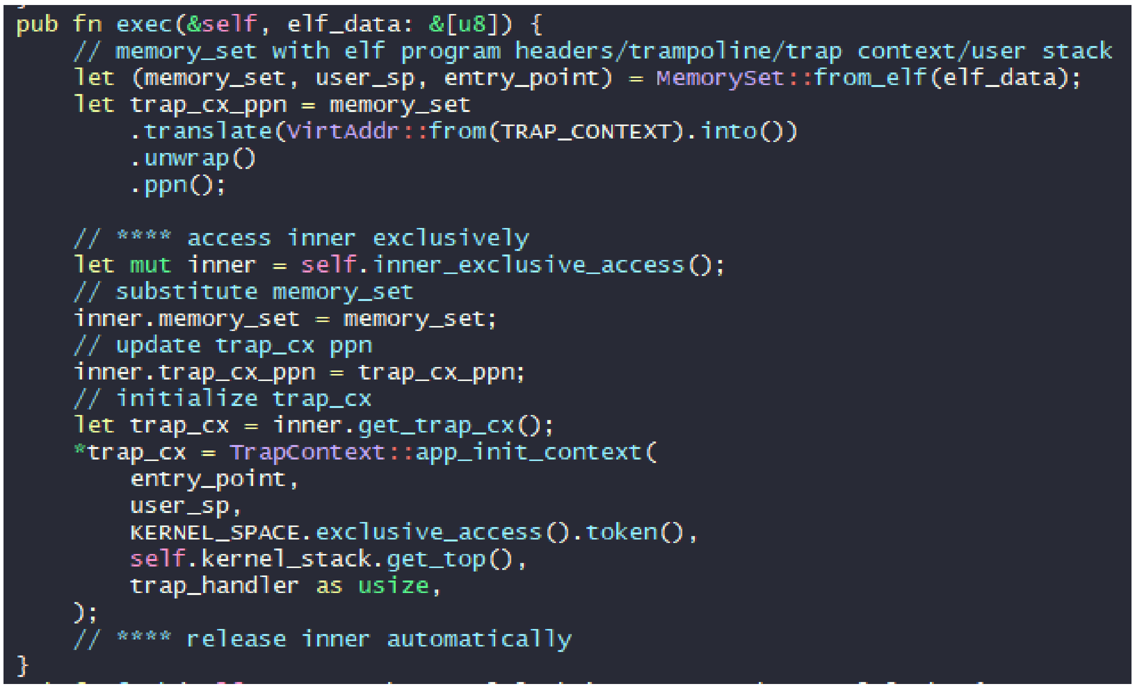


MINGW64:/c/Users/Dal-Z41/Desktop/课件&作业/Operation System/experiment/code/gardeneros/os/src/task

```
impl TaskControlBlock {
    pub fn inner_exclusive_access(&self) -> RefMut<'_, TaskControlBlockInner> {
        self.inner.exclusive_access()
    }
    pub fn new(elf_data: &[u8]) -> Self {
        // memory_set with elf program headers/trampoline/trap context/user stack
        let (memory_set, user_sp, entry_point) = MemorySet::from_elf(elf_data);
        let trap_cx_ppn = memory_set
            .translate(VirtAddr::from(TRAP_CONTEXT).into())
            .unwrap()
            .ppn();
        // alloc a pid and a kernel stack in kernel space
        let pid_handle = pid_alloc();
        let kernel_stack = KernelStack::new(&pid_handle);
        let kernel_stack_top = kernel_stack.get_top();
        // push a task context which goes to trap_return to the top of kernel stack
        let task_control_block = Self {
            pid: pid_handle,
            kernel_stack,
            inner: unsafe { UPSafeCell::new(TaskControlBlockInner {
                trap_cx_ppn,
                base_size: user_sp,
                task_cx: TaskContext::goto_trap_return(kernel_stack_top),
                task_status: TaskStatus::Ready,
                memory_set,
                parent: None,
                children: Vec::new(),
                exit_code: 0,
            })},
        };
        // prepare TrapContext in user space
        let trap_cx = task_control_block.inner_exclusive_access().get_trap_cx();
        *trap_cx = TrapContext::app_init_context(
            entry_point,
            user_sp,
            KERNEL_SPACE.exclusive_access().token(),
            kernel_stack_top,
            trap_handler as usize,
        );
        task_control_block
    }
    pub fn exec(&self, elf_data: &[u8]) {
        // memory_set with elf program headers/trampoline/trap context/user stack
        let (memory_set, user_sp, entry_point) = MemorySet::from_elf(elf_data);
        let trap_cx_ppn = memory_set
            .translate(VirtAddr::from(TRAP_CONTEXT).into())
            .unwrap()
            .ppn();

        // **** access inner exclusively
        let mut inner = self.inner_exclusive_access();
        // substitute memory_set
        inner.memory_set = memory_set;
        // update trap_cx ppn
        inner.trap_cx_ppn = trap_cx_ppn;
        // initialize trap_cx
        let trap_cx = inner.get_trap_cx();
        *trap_cx = TrapContext::app_init_context(
            entry_point,
            user_sp,
            KERNEL_SPACE.exclusive_access().token(),
            self.kernel_stack.get_top(),
            trap_handler as usize,
        );
        // **** release inner automatically
task.rs [dos] (13:25 14/11/2021)
```

3. Then, the **exec** system call is implemented.

```
//os/src/task/task.rs
impl TaskControlBlock {

    pub fn exec(&self, elf_data: &[u8]) {
        // memory_set with elf program headers/trampoline/trap context/user
stack
        let (memory_set, user_sp, entry_point) =
MemorySet::from_elf(elf_data);
        let trap_cx_ppn = memory_set
            .translate(VirtAddr::from(TRAP_CONTEXT).into())
            .unwrap()
            .ppn();

        // **** access inner exclusively
        let mut inner = self.inner_exclusive_access();
        // substitute memory_set
        inner.memory_set = memory_set;
        // update trap_cx ppn
        inner.trap_cx_ppn = trap_cx_ppn;
        // initialize trap_cx
        let trap_cx = inner.get_trap_cx();
        *trap_cx = TrapContext::app_init_context(
            entry_point,
            user_sp,
            KERNEL_SPACE.exclusive_access().token(),
            self.kernel_stack.get_top(),
            trap_handler as usize,
        );
        // **** release inner automatically
    }
}
```

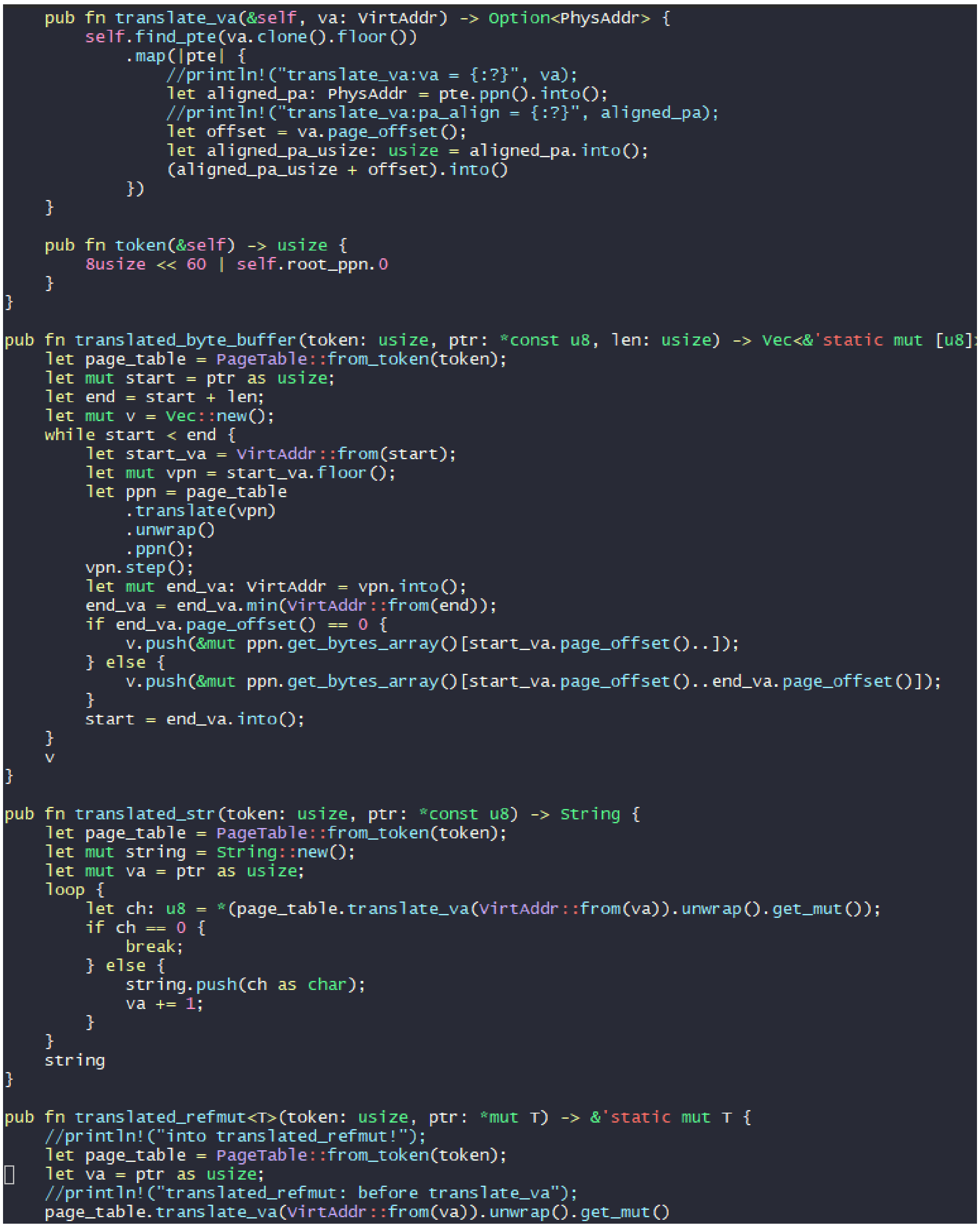


```
pub fn exec(&self, elf_data: &[u8]) {
    // memory_set with elf program headers/trampoline/trap context/user stack
    let (memory_set, user_sp, entry_point) = MemorySet::from_elf(elf_data);
    let trap_cx_ppn = memory_set
        .translate(VirtAddr::from(TRAP_CONTEXT).into())
        .unwrap()
        .ppn();

    // **** access inner exclusively
    let mut inner = self.inner_exclusive_access();
    // substitute memory_set
    inner.memory_set = memory_set;
    // update trap_cx ppn
    inner.trap_cx_ppn = trap_cx_ppn;
    // initialize trap_cx
    let trap_cx = inner.get_trap_cx();
    *trap_cx = TrapContext::app_init_context(
        entry_point,
        user_sp,
        KERNEL_SPACE.exclusive_access().token(),
        self.kernel_stack.get_top(),
        trap_handler as usize,
    );
    // **** release inner automatically
}
```

4. The implementation of "**sys_exec**" also relies on modifications to the page table.

```
//os/src/mm/page_table.rs
impl PageTable {
    pub fn translate_va(&self, va: VirtAddr) -> Option<PhysAddr> {
        self.find_pte(va.clone().floor())
```

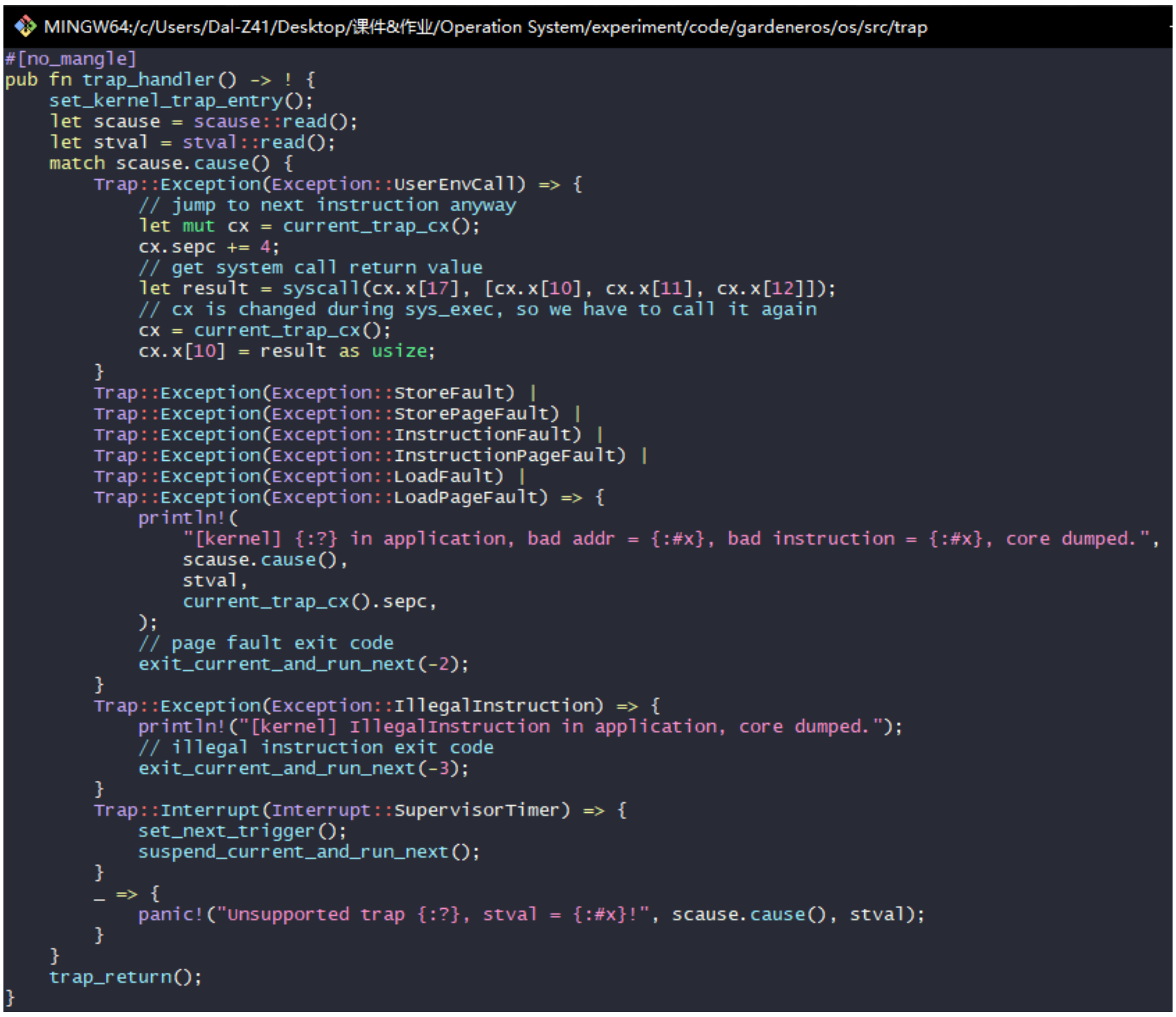
```
            .map(|pte| {
                //println!("translate_va:va = {:?}", va);
                let aligned_pa: PhysAddr = pte.ppn().into();
                //println!("translate_va:pa_align = {:?}", aligned_pa);
                let offset = va.page_offset();
                let aligned_pa_usize: usize = aligned_pa.into();
                (aligned_pa_usize + offset).into()
            })
    }
}

pub fn translated_str(token: usize, ptr: *const u8) -> String {
    let page_table = PageTable::from_token(token);
    let mut string = String::new();
    let mut va = ptr as usize;
    loop {
        let ch: u8 = *
(page_table.translate_va(VirtAddr::from(va)).unwrap().get_mut());
        if ch == 0 {
            break;
        } else {
            string.push(ch as char);
            va += 1;
        }
    }
    string
}

pub fn translated_refmut<T>(token: usize, ptr: *mut T) -> &'static mut T {
    //println!("into translated_refmut!");
    let page_table = PageTable::from_token(token);
    let va = ptr as usize;
    //println!("translated_refmut: before translate_va");
    page_table.translate_va(VirtAddr::from(va)).unwrap().get_mut()
}
```



```
    pub fn translate_va(&self, va: VirtAddr) -> Option<PhysAddr> {
        self.find_pte(va.clone().floor())
            .map(|pte| {
                //println!("translate_va:va = {:?}", va);
                let aligned_pa: PhysAddr = pte.ppn().into();
                //println!("translate_va:pa_align = {:?}", aligned_pa);
                let offset = va.page_offset();
                let aligned_pa_usize: usize = aligned_pa.into();
                (aligned_pa_usize + offset).into()
            })
    }

    pub fn token(&self) -> usize {
        8usize << 60 | self.root_ppn.0
    }
}

pub fn translated_byte_buffer(token: usize, ptr: *const u8, len: usize) -> Vec<&'static mut [u8]
    let page_table = PageTable::from_token(token);
    let mut start = ptr as usize;
    let end = start + len;
    let mut v = Vec::new();
    while start < end {
        let start_va = VirtAddr::from(start);
        let mut vpn = start_va.floor();
        let ppn = page_table
            .translate(vpn)
            .unwrap()
            .ppn();
        vpn.step();
        let mut end_va: VirtAddr = vpn.into();
        end_va = end_va.min(VirtAddr::from(end));
        if end_va.page_offset() == 0 {
            v.push(&mut ppn.get_bytes_array()[start_va.page_offset()..]);
        } else {
            v.push(&mut ppn.get_bytes_array()[start_va.page_offset()..end_va.page_offset()]);
        }
        start = end_va.into();
    }
    v
}

pub fn translated_str(token: usize, ptr: *const u8) -> String {
    let page_table = PageTable::from_token(token);
    let mut string = String::new();
    let mut va = ptr as usize;
    loop {
        let ch: u8 = *(page_table.translate_va(VirtAddr::from(va)).unwrap().get_mut());
        if ch == 0 {
            break;
        } else {
            string.push(ch as char);
            va += 1;
        }
    }
    string
}

pub fn translated_refmut<T>(token: usize, ptr: *mut T) -> &'static mut T {
    //println!("into translated_refmut!");
    let page_table = PageTable::from_token(token);
    let va = ptr as usize;
    //println!("translated_refmut: before translate_va");
    page_table.translate_va(VirtAddr::from(va)).unwrap().get_mut()
```

5. After the sys_exec system call, the original context cx of trap_handler is invalidated. To do this, you need to retrieve the trap context again after the syscall distribution.

```
//os/src/trap/mod.rs
#[no_mangle]
pub fn trap_handler() -> ! {
    set_kernel_trap_entry();
    let cx = current_trap_cx();
    let scause = scause::read();
    let stval = stval::read();
    match scause.cause() {
        Trap::Exception(Exception::UserEnvCall) => {
            // jump to next instruction anyway
            let mut cx = current_trap_cx();
            cx.sepc += 4;
            // get system call return value
            let result = syscall(cx.x[17], [cx.x[10], cx.x[11], cx.x[12]]);
            // cx is changed during sys_exec, so we have to call it again
            cx = current_trap_cx();
            cx.x[10] = result as usize;
        }
        Trap::Exception(Exception::StoreFault) |
        Trap::Exception(Exception::StorePageFault) |
        Trap::Exception(Exception::InstructionFault) |
        Trap::Exception(Exception::InstructionPageFault) |
        Trap::Exception(Exception::LoadFault) |
        Trap::Exception(Exception::LoadPageFault) => {
            println!(
                "[kernel] {:?} in application, bad addr = {:#x}, bad
instruction = {:#x}, core dumped.",
                scause.cause(),
                stval,
                current_trap_cx().sepc,
            );
            // page fault exit code
            exit_current_and_run_next(-2);
        }
        Trap::Exception(Exception::IllegalInstruction) => {
            println!("[kernel] IllegalInstruction in application, core
dumped.");
            // illegal instruction exit code
            exit_current_and_run_next(-3);
        }
        Trap::Interrupt(Interrupt::SupervisorTimer) => {
            set_next_trigger();
            suspend_current_and_run_next();
        }
        _ => {
            panic!("Unsupported trap {:?}, stval = {:#x}!", scause.cause(),
stval);
        }
    }
    trap_return();
}
```

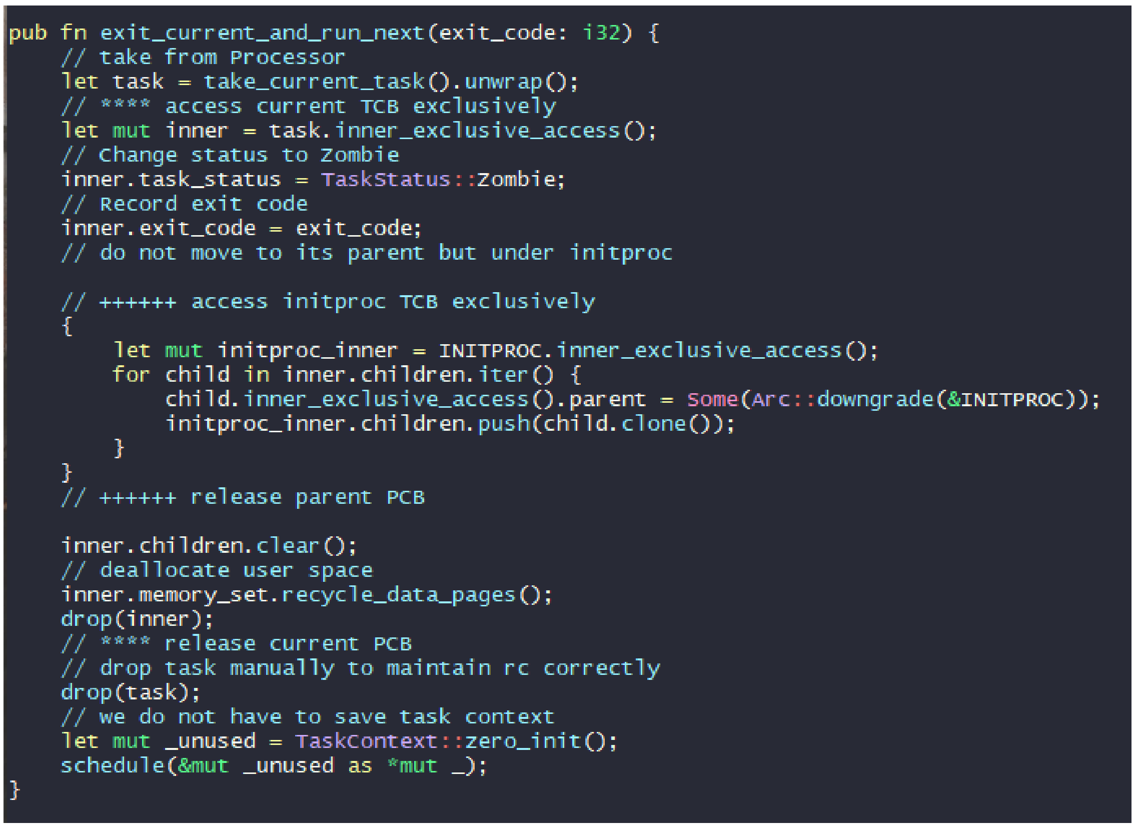

MINGW64:/c/Users/Dal-Z41/Desktop/课件&作业/Operation System/experiment/code/gardeneros/os/src/trap

```
#[no_mangle]
pub fn trap_handler() -> ! {
    set_kernel_trap_entry();
    let scause = scause::read();
    let stval = stval::read();
    match scause.cause() {
        Trap::Exception(Exception::UserEnvCall) => {
            // jump to next instruction anyway
            let mut cx = current_trap_cx();
            cx.sepc += 4;
            // get system call return value
            let result = syscall(cx.x[17], [cx.x[10], cx.x[11], cx.x[12]]);
            // cx is changed during sys_exec, so we have to call it again
            cx = current_trap_cx();
            cx.x[10] = result as usize;
        }
        Trap::Exception(Exception::StoreFault) |
        Trap::Exception(Exception::StorePageFault) |
        Trap::Exception(Exception::InstructionFault) |
        Trap::Exception(Exception::InstructionPageFault) |
        Trap::Exception(Exception::LoadFault) |
        Trap::Exception(Exception::LoadPageFault) => {
            println!(
                "[kernel] {:?} in application, bad addr = {:#x}, bad instruction = {:#x}, core dumped.",
                scause.cause(),
                stval,
                current_trap_cx().sepc,
            );
            // page fault exit code
            exit_current_and_run_next(-2);
        }
        Trap::Exception(Exception::IllegalInstruction) => {
            println!("[kernel] IllegalInstruction in application, core dumped.");
            // illegal instruction exit code
            exit_current_and_run_next(-3);
        }
        Trap::Interrupt(Interrupt::SupervisorTimer) => {
            set_next_trigger();
            suspend_current_and_run_next();
        }
        _ => {
            panic!("Unsupported trap {:?}, stval = {:#x}!", scause.cause(), stval);
        }
    }
    trap_return();
}
```

11. 进程资源回收机制

1. The "**exit_current_and_run_next**" function is called in the kernel to exit the current process and switch to the next process after the **"sys_exit"** system call is either initiated by the application or terminated by the kernel in error. Compared to the previous implementation, "**exit_current_and_run_next**" adds an exit code as an argument.

```
// os/src/mm/memory_set.rs
impl MemorySet {
    pub fn recycle_data_pages(&mut self) {
        self.areas.clear();
    }
}

//os/src/task/mod.rs
pub fn exit_current_and_run_next(exit_code: i32) {
    // take from Processor
    let task = take_current_task().unwrap();
    // **** access current TCB exclusively
    let mut inner = task.inner_exclusive_access();
    // Change status to Zombie
    inner.task_status = TaskStatus::Zombie;
    // Record exit code
    inner.exit_code = exit_code;
    // do not move to its parent but under initproc
    // ++++++ access initproc TCB exclusively
    {
        let mut initproc_inner = INITPROC.inner_exclusive_access();
        for child in inner.children.iter() {
            child.inner_exclusive_access().parent =
Some(Arc::downgrade(&INITPROC));
            initproc_inner.children.push(child.clone());
        }
    }
    // ++++++ release parent PCB

    inner.children.clear();
    // deallocate user space
    inner.memory_set.recycle_data_pages();
    drop(inner);
    // **** release current PCB
    // drop task manually to maintain rc correctly
    drop(task);
    // we do not have to save task context
    let mut _unused = TaskContext::zero_init();
    schedule(&mut _unused as *mut _);
}
```

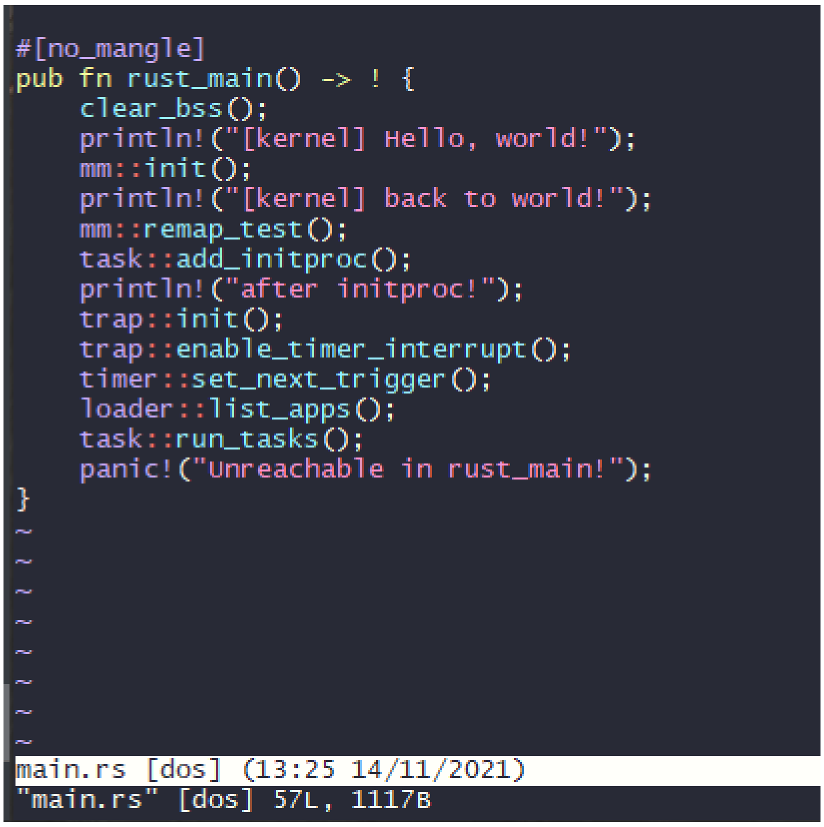


```
pub fn exit_current_and_run_next(exit_code: i32) {
    // take from Processor
    let task = take_current_task().unwrap();
    // **** access current TCB exclusively
    let mut inner = task.inner_exclusive_access();
    // Change status to Zombie
    inner.task_status = TaskStatus::Zombie;
    // Record exit code
    inner.exit_code = exit_code;
    // do not move to its parent but under initproc

    // ++++++ access initproc TCB exclusively
    {
        let mut initproc_inner = INITPROC.inner_exclusive_access();
        for child in inner.children.iter() {
            child.inner_exclusive_access().parent = Some(Arc::downgrade(&INITPROC));
            initproc_inner.children.push(child.clone());
        }
    }
    // ++++++ release parent PCB

    inner.children.clear();
    // deallocate user space
    inner.memory_set.recycle_data_pages();
    drop(inner);
    // **** release current PCB
    // drop task manually to maintain rc correctly
    drop(task);
    // we do not have to save task context
    let mut _unused = TaskContext::zero_init();
    schedule(&mut _unused as *mut _);
}
```

2. At the same time, the parent reclaims the child's resources and collects some of its information through the sys_waitpidsystem call.

3. Finally, modify "**main.rs.**"

```
//os/src/main.rs

#[no_mangle]
pub fn rust_main() -> ! {
    clear_bss();
    println!("[kernel] Hello, world!");
    mm::init();
    println!("[kernel] back to world!");
    mm::remap_test();
    task::add_initproc();
    println!("after initproc!");
    trap::init();
    trap::enable_timer_interrupt();
    timer::set_next_trigger();
    loader::list_apps();
    task::run_tasks();
    panic!("Unreachable in rust_main!");
}
```



```
#[no_mangle]
pub fn rust_main() -> ! {
    clear_bss();
    println!("[kernel] Hello, world!");
    mm::init();
    println!("[kernel] back to world!");
    mm::remap_test();
    task::add_initproc();
    println!("after initproc!");
    trap::init();
    trap::enable_timer_interrupt();
    timer::set_next_trigger();
    loader::list_apps();
    task::run_tasks();
    panic!("Unreachable in rust_main!");
}
~
~
~
~
~
~
~
~
main.rs [dos] (13:25 14/11/2021)
"main.rs" [dos] 57L, 1117B
```