

# 19271169-张东植-实验报告4

---

## 19271169-张东植-实验报告4

### 1. 实现应用程序

1.1 Settings

1.2 Yield

1.3 Test

### 2. 多道程序加载

### 3. 任务设计与实现

3.1 Context Switch

3.2 Task condition & TCB

3.3 Task Switch

3.4 Task Manager

### 4. sys\_yield & sys\_exit

### 5. 完善程序

### 6. 运行结果

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

The main purpose of this experiment is to implement an operating system that supports multi-program and cooperative scheduling. Through this experiment, we can realize an operating system that supports multi-program and cooperative scheduling. It includes application program placement, multi-channel program loading, task design and implementation, system call like sys\_yield and sys\_exit implementation, etc.

## 1. 实现应用程序

---

The application implementation of multiprogram operating system is basically the same as that of batch operating system. The main difference is where the application is loaded into memory. At the same time, in a multiprogram operating system, an application can voluntarily give up the CPU to switch to another application. Collaborative scheduling means that the application program actively cedes the CPU during THE I/O operation so that the CPU can execute other applications and ultimately improve the CPU efficiency.

### 1.1 Settings

---

In a batch operating system, the memory location for application loading is the same. However, in a multiprogram operating system, each application loads at a different location. This is why the BASE\_ADDRESS of "linker.ld" in the linker script is different. To do this, we need to write a script called "build.py" that implements custom link scripts for each application.

```
// user/build.py
import os

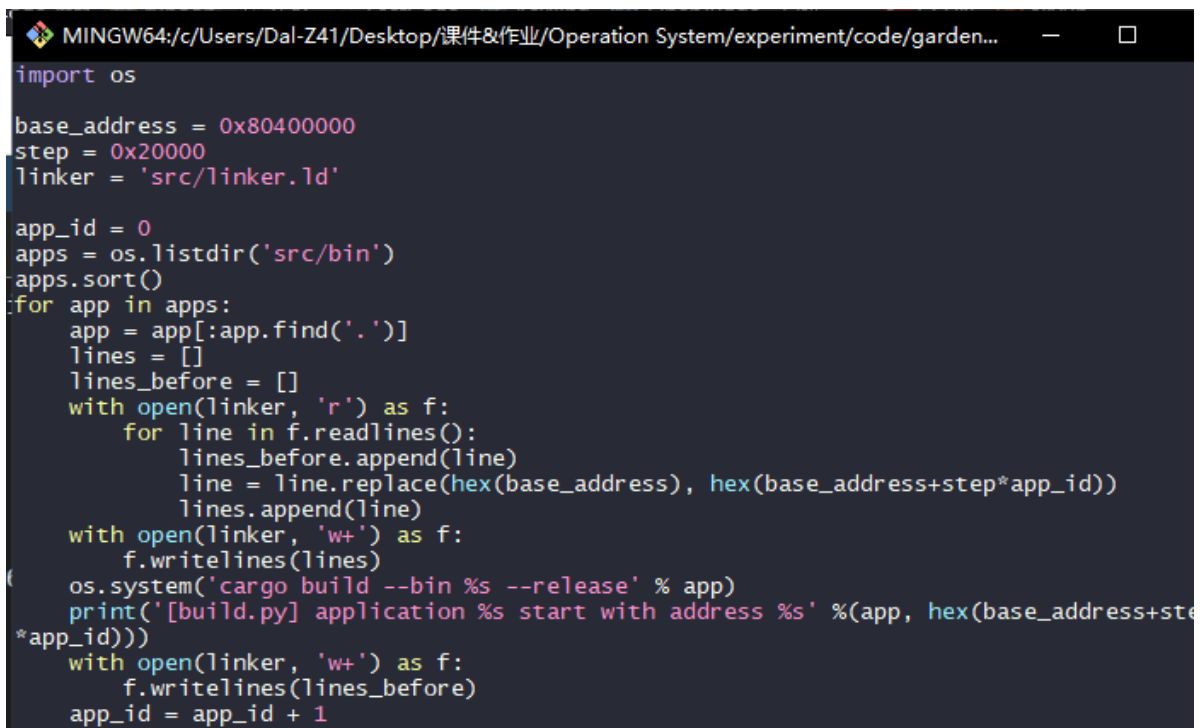
base_address = 0x80400000
step = 0x20000
linker = 'src/linker.ld'

app_id = 0
```

```

apps = os.listdir('src/bin')
apps.sort()
for app in apps:
    app = app[:app.find('.')]
    lines = []
    lines_before = []
    with open(linker, 'r') as f:
        for line in f.readlines():
            lines_before.append(line)
            line = line.replace(hex(base_address),
hex(base_address+step*app_id))
            lines.append(line)
    with open(linker, 'w+') as f:
        f.writelines(lines)
    os.system('cargo build --bin %s --release' % app)
    print('[build.py] application %s start with address %s' %(app,
hex(base_address+step*app_id)))
    with open(linker, 'w+') as f:
        f.writelines(lines_before)
    app_id = app_id + 1

```



```

import os

base_address = 0x80400000
step = 0x20000
linker = 'src/linker.ld'

app_id = 0
apps = os.listdir('src/bin')
apps.sort()
for app in apps:
    app = app[:app.find('.')]
    lines = []
    lines_before = []
    with open(linker, 'r') as f:
        for line in f.readlines():
            lines_before.append(line)
            line = line.replace(hex(base_address), hex(base_address+step*app_id))
            lines.append(line)
    with open(linker, 'w+') as f:
        f.writelines(lines)
    os.system('cargo build --bin %s --release' % app)
    print('[build.py] application %s start with address %s' %(app, hex(base_address+step*app_id)))
    with open(linker, 'w+') as f:
        f.writelines(lines_before)
    app_id = app_id + 1

```

## 1.2 Yield

The procedures performed by the application are usually computations and IO intermittently executed. If the program still occupies CPU during the IO execution, it will cause a waste of CPU resources. Multiprogramming allows applications to actively cede CPU privileges to other applications while performing IO operations. Yield system calls are system calls that enable an application to voluntarily surrender CPU privileges.

```
// sys_yield
const SYSCALL_YIELD: usize = 124;

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

// lib.rs
pub fn yield_() -> isize { sys_yield() }
```

```
MINGW64:/c/Users/Dal-Z41/Desktop/课件&作业/Operation System/experiment/code/garden...
const SYSCALL_WRITE: usize = 64;
const SYSCALL_EXIT: usize = 93;
const SYSCALL_YIELD: usize = 124;

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

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) -> isize {
    syscall(SYSCALL_EXIT, [exit_code as usize, 0, 0])
}

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

```

MINGW64:/c/Users/Dal-Z41/Desktop/课件&作业/Operation System/experiment/code/gar
#![no_std]
#![feature(asm)]
#![feature(linkage)]
#![feature(panic_info_message)]

#[macro_use]
pub mod console;
mod syscall;
mod lang_items;

fn clear_bss() {
    extern "C" {
        fn start_bss();
        fn end_bss();
    }
    (start_bss as usize..end_bss as usize).for_each(|addr| {
        unsafe { (addr as *mut u8).write_volatile(0); }
    });
}

#[no_mangle]
#[link_section = ".text.entry"]
pub extern "C" fn _start() -> ! {
    clear_bss();
    exit(main());
    panic!("unreachable after sys_exit!");
}

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

use syscall::*;

pub fn write(fd: usize, buf: &[u8]) -> isize { sys_write(fd, buf) }
pub fn exit(exit_code: i32) -> isize { sys_exit(exit_code) }
pub fn yield_() -> isize { sys_yield() }

~
~
~
~
~
~
~
~
~
~
lib.rs [unix] (15:54 12/11/2021)

```

## 1.3 Test

Then, we can now execute the make build command in the user directory to compile the multiprogram operating system application.

- 00write\_a.rs :

```

MINGW64:/c:/Users/Dal-Z41/Desktop/课件&作业/Operation System/experiment/c
#![no_std]
#![no_main]

#[macro_use]
extern crate user_lib;

use user_lib::yield_;

const WIDTH: usize = 10;
const HEIGHT: usize = 5;

#[no_mangle]
fn main() -> i32 {
    for i in 0..HEIGHT {
        for _ in 0..WIDTH { print!("A"); }
        println!(" [{}]/{}", i + 1, HEIGHT);
        yield_();
    }
    println!("Test write_a OK!");
    0
}

```

- 01write\_b.rs :

```

MINGW64:/c:/Users/Dal-Z41/Desktop/课件&作业/Operation System/experiment/co
#![no_std]
#![no_main]

#[macro_use]
extern crate user_lib;

use user_lib::yield_;

const WIDTH: usize = 10;
const HEIGHT: usize = 5;

#[no_mangle]
fn main() -> i32 {
    for i in 0..HEIGHT {
        for _ in 0..WIDTH { print!("B"); }
        println!(" [{}]/{}", i + 1, HEIGHT);
        yield_();
    }
    println!("Test write_b OK!");
    0
}
~

```

- 02write\_c.rs :

```

MINGW64:/c:/Users/Dal-Z41/Desktop/课件&作业/Operation System/exp
#![no_std]
#![no_main]

#[macro_use]
extern crate user_lib;

use user_lib::yield_;

const WIDTH: usize = 10;
const HEIGHT: usize = 5;

#[no_mangle]
fn main() -> i32 {
    for i in 0..HEIGHT {
        for _ in 0..WIDTH { print!("c"); }
        println!(" [{}]/[{}]", i + 1, HEIGHT);
        yield_();
    }
    println!("Test write_c OK!");
    0
}

```

- make build :

```

root@izuf69gjp4acy5gbxody49Z:~# cd user
root@izuf69gjp4acy5gbxody49Z:~/user# make build
   Compiling user_lib v0.1.0 (/root/user)
   Finished release [optimized] target(s) in 2.77s
src/bin/02write_c.rs src/bin/00write_a.rs src/bin/01write_b.rs
target/riscv64gc-unknown-none-elf/release/02write_c target/riscv64gc-unknown-none-elf/release/00write_a target/riscv64gc-unknown-none-elf/release/01write_b
target/riscv64gc-unknown-none-elf/release/02write_c.bin target/riscv64gc-unknown-none-elf/release/00write_a.bin target/riscv64gc-unknown-none-elf/release/01write_b.bin
rust-objcopy --binary-architecture=riscv64 target/riscv64gc-unknown-none-elf/release/02write_c --strip-all -O binary target/riscv64gc-unknown-none-elf/release/02write_c.bin; rust-objcopy --binary-architecture=riscv64 target/riscv64gc-unknown-none-elf/release/00write_a --strip-all -O binary target/riscv64gc-unknown-none-elf/release/00write_a.bin; rust-objcopy --binary-architecture=riscv64 target/riscv64gc-unknown-none-elf/release/01write_b --strip-all -O binary target/riscv64gc-unknown-none-elf/release/01write_b.bin;

```

## 2. 多道程序加载

In batch operating systems, the loading and execution of applications is handled by the **batch** submodule. In a multiprogram operating system, the loading and execution of application programs are divided into two modules. The **loader** submodule is responsible for loading applications, and the **task** submodule is responsible for executing and switching applications. Also, unlike batch operating systems, the applications used by multiprogram operating systems are loaded together into memory when the kernel is initialized.

1. First, separate some constants into "**config.rs**".

```

pub const USER_STACK_SIZE: usize = 4096 * 2;
pub const KERNEL_STACK_SIZE: usize = 4096 * 2;
pub const MAX_APP_NUM: usize = 4;
pub const APP_BASE_ADDRESS: usize = 0x80400000;
pub const APP_SIZE_LIMIT: usize = 0x20000;

```

```

MINGW64:/c/Users/Dal-Z41/Desktop/课件&作业/Operation System/experiment/code/gardene..
pub const USER_STACK_SIZE: usize = 4096 * 2;
pub const KERNEL_STACK_SIZE: usize = 4096 * 2;
pub const MAX_APP_NUM: usize = 4;
pub const APP_BASE_ADDRESS: usize = 0x80400000;
pub const APP_SIZE_LIMIT: usize = 0x20000;

```

2. Reuse batch submodule kernel stack and user stack code. Note that the kernel stack context information has been added to the task context information.

```

use crate::trap::TrapContext;
use crate::task::TaskContext;
use crate::config::*;

#[repr(align(4096))]
#[derive(Copy, Clone)]
struct KernelStack {
    data: [u8; KERNEL_STACK_SIZE],
}

#[repr(align(4096))]
#[derive(Copy, Clone)]
struct UserStack {
    data: [u8; USER_STACK_SIZE],
}

static KERNEL_STACK: [KernelStack; MAX_APP_NUM] = [
    KernelStack { data: [0; KERNEL_STACK_SIZE], },
    MAX_APP_NUM
];

static USER_STACK: [UserStack; MAX_APP_NUM] = [
    UserStack { data: [0; USER_STACK_SIZE], },
    MAX_APP_NUM
];

impl KernelStack {
    fn get_sp(&self) -> usize {
        self.data.as_ptr() as usize + KERNEL_STACK_SIZE
    }
    pub fn push_context(&self, trap_cx: TrapContext, task_cx: TaskContext) -
    > &'static mut TaskContext {
        unsafe {
            let trap_cx_ptr = (self.get_sp() - core::mem::size_of::
            <TrapContext>()) as *mut TrapContext;
            *trap_cx_ptr = trap_cx;
            let task_cx_ptr = (trap_cx_ptr as usize - core::mem::size_of::
            <TaskContext>()) as *mut TaskContext;
            *task_cx_ptr = task_cx;
            task_cx_ptr.as_mut().unwrap()
        }
    }
}

impl UserStack {
    fn get_sp(&self) -> usize {
        self.data.as_ptr() as usize + USER_STACK_SIZE
    }
}

```

```
}  
}
```

### 3. Load the program.

```
fn get_base_i(app_id: usize) -> usize {  
    APP_BASE_ADDRESS + app_id * APP_SIZE_LIMIT  
}  
  
pub fn get_num_app() -> usize {  
    extern "C" { fn _num_app(); }  
    unsafe { (_num_app as usize as *const usize).read_volatile() }  
}  
  
pub fn load_apps() {  
    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)  
    };  
    // clear i-cache first  
    unsafe { asm!("fence.i"); }  
    // load apps  
    for i in 0..num_app {  
        let base_i = get_base_i(i);  
        // clear region  
        (base_i..base_i + APP_SIZE_LIMIT).for_each(|addr| unsafe {  
            (addr as *mut u8).write_volatile(0)  
        });  
        // load app from data section to memory  
        let src = unsafe {  
            core::slice::from_raw_parts(app_start[i] as *const u8,  
            app_start[i + 1] - app_start[i])  
        };  
        let dst = unsafe {  
            core::slice::from_raw_parts_mut(base_i as *mut u8, src.len())  
        };  
        dst.copy_from_slice(src);  
    }  
}
```



```
use crate::trap::TrapContext;
use crate::task::TaskContext;
use crate::config::*;

#[repr(align(4096))]
#[derive(copy, clone)]
struct KernelStack {
    data: [u8; KERNEL_STACK_SIZE],
}

#[repr(align(4096))]
#[derive(copy, clone)]
struct UserStack {
    data: [u8; USER_STACK_SIZE],
}

static KERNEL_STACK: [KernelStack; MAX_APP_NUM] = [
    KernelStack { data: [0; KERNEL_STACK_SIZE], },
    MAX_APP_NUM
];

static USER_STACK: [UserStack; MAX_APP_NUM] = [
    UserStack { data: [0; USER_STACK_SIZE], },
    MAX_APP_NUM
];

impl KernelStack {
    fn get_sp(&self) -> usize {
        self.data.as_ptr() as usize + KERNEL_STACK_SIZE
    }
    pub fn push_context(&self, trap_cx: TrapContext, task_cx: TaskContext) -> &'static mut TaskContext {
        unsafe {
            let trap_cx_ptr = (self.get_sp() - core::mem::size_of:::<TrapContext>()) as *mut TrapContext;
            *trap_cx_ptr = trap_cx;
            let task_cx_ptr = (trap_cx_ptr as usize - core::mem::size_of:::<TaskContext>()) as *mut TaskContext;
            *task_cx_ptr = task_cx;
            task_cx_ptr.as_mut().unwrap()
        }
    }
}

impl UserStack {
    fn get_sp(&self) -> usize {
        self.data.as_ptr() as usize + USER_STACK_SIZE
    }
}

fn get_base_i(app_id: usize) -> usize {
    APP_BASE_ADDRESS + app_id * APP_SIZE_LIMIT
}

loader.rs [dos] (16:30 12/11/2021) 1,1 Top
```

We can see that the application is loaded at the physical address calculated by `get_base_i`.

## 3. 任务设计与实现

Multiprogramming enables applications to voluntarily surrender CPU usage. We call a computational execution a task. Switching a task from one application to another is called task switching. The contents such as registers and stacks that need to be saved during the task switchover are called task lead-in files. Different from Trap switching, task switching does not involve privilege switching.

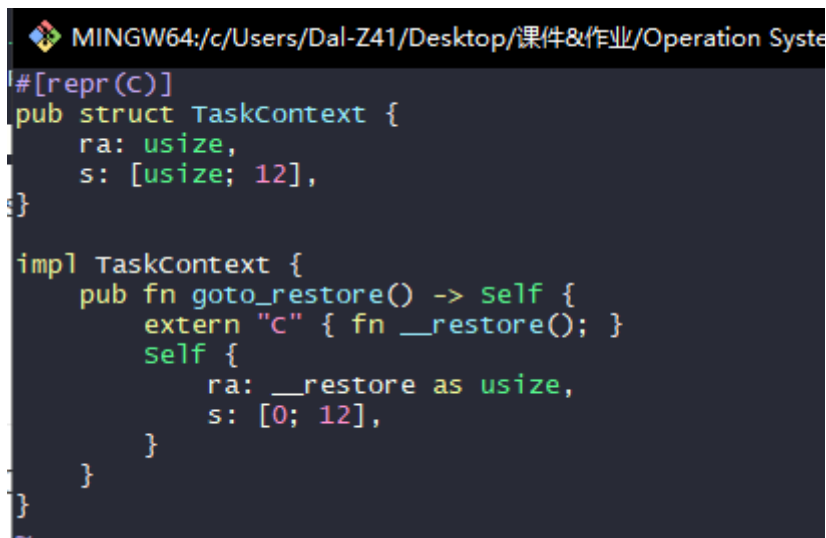
### 3.1 Context Switch

Task switching needs to be supported by task context. We use the **TaskContext** data structure to record the context information of the task.

```
#[repr(c)]
```

```
pub struct TaskContext {
    ra: usize,
    s: [usize; 12],
}

impl TaskContext {
    pub fn goto_restore() -> Self {
        extern "C" { fn __restore(); }
        Self {
            ra: __restore as usize,
            s: [0; 12],
        }
    }
}
```



```
MINGW64:/c/Users/Dal-Z41/Desktop/课件&作业/Operation System
#[repr(C)]
pub struct TaskContext {
    ra: usize,
    s: [usize; 12],
}

impl TaskContext {
    pub fn goto_restore() -> Self {
        extern "C" { fn __restore(); }
        Self {
            ra: __restore as usize,
            s: [0; 12],
        }
    }
}
```

## 3.2 Task condition & TCB

To support task switching, we need to maintain the running state of the task in the kernel. The kernel also needs to store more information in the task control block. A task context address pointer `task_cx_ptr` is maintained in the task control block.

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

#[derive(Copy, Clone)]
pub struct TaskControlBlock {
    pub task_cx_ptr: usize,
    pub task_status: TaskStatus,
}

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

```

MINGW64/c/Users/Dal-Z41/Desktop/课件&作业/Operation System/experiment
#[derive(Copy, Clone)]
pub struct TaskControlBlock {
    pub task_cx_ptr: usize,
    pub task_status: TaskStatus,
}

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

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

```

### 3.3 Task Switch

The main process of task switching is that after saving the context of a task, the task enters the pause state. At the same time, restore the context of another task and let it continue on the CPU.

```

.altmacro
.macro SAVE_SN n
    sd s\n, (\n+1)*8(sp)
.endm
.macro LOAD_SN n
    ld s\n, (\n+1)*8(sp)
.endm
.section .text
.globl __switch
__switch:
    # __switch(
    #     current_task_cx_ptr2: &*const TaskContext,
    #     next_task_cx_ptr2: &*const TaskContext
    # )
    # push TaskContext to current sp and save its address to where a0 points to
    addi sp, sp, -13*8
    sd sp, 0(a0)
    # fill TaskContext with ra & s0-s11
    sd ra, 0(sp)
    .set n, 0
    .rept 12
        SAVE_SN %n
        .set n, n + 1
    .endr
    # ready for loading TaskContext a1 points to
    ld sp, 0(a1)
    # load registers in the TaskContext
    ld ra, 0(sp)
    .set n, 0
    .rept 12
        LOAD_SN %n
        .set n, n + 1
    .endr

```

```
# pop TaskContext
addi sp, sp, 13*8
ret
```

```
MINGW64:/c/Users/Dal-Z41/Desktop/课件&作业/Operation System/experiment/code/gardene...
.altmacro
.macro SAVE_SN n
    sd s\, (\n+1)*8(sp)
.endm
.macro LOAD_SN n
    ld s\, (\n+1)*8(sp)
.endm
.section .text
.globl __switch
__switch:
# __switch(
#     current_task_cx_ptr2: &const TaskContext,
#     next_task_cx_ptr2: &const TaskContext
# )
# push TaskContext to current sp and save its address to where a0 points to
addi sp, sp, -13*8
sd sp, 0(a0)
# fill TaskContext with ra & s0-s11
sd ra, 0(sp)
.set n, 0
.rept 12
    SAVE_SN %n
    .set n, n + 1
.endr
# ready for loading TaskContext a1 points to
ld sp, 0(a1)
# load registers in the TaskContext
ld ra, 0(sp)
.set n, 0
.rept 12
    LOAD_SN %n
    .set n, n + 1
.endr
# pop TaskContext
addi sp, sp, 13*8
ret
```

To make it easier to call `__switch`, you also need to wrap it as a function of Rust.

```
global_asm!(include_str!("switch.S"));

extern "C" {
    pub fn __switch(
        current_task_cx_ptr2: *const usize,
        next_task_cx_ptr2: *const usize
    );
}
```

```
MINGW64:/c/Users/Dal-Z41/Desktop/课件&作业/Operation Sys
global_asm!(include_str!("switch.S"));

extern "C" {
    pub fn __switch(
        current_task_cx_ptr2: *const usize,
        next_task_cx_ptr2: *const usize
    );
}
```

## 3.4 Task Manager

We need a global task manager to manage the application described by **task control**.

```
mod context;
mod switch;
mod task;

use crate::config::MAX_APP_NUM;
use crate::loader::{get_num_app, init_app_cx};
use core::cell::RefCell;
use lazy_static::*;
use switch::__switch;
use task::{TaskControlBlock, TaskStatus};

pub use context::TaskContext;

pub struct TaskManager {
    num_app: usize,
    inner: RefCell<TaskManagerInner>,
}

struct TaskManagerInner {
    tasks: [TaskControlBlock; MAX_APP_NUM],
    current_task: usize,
}

unsafe impl Sync for TaskManager {}

lazy_static! {
    pub static ref TASK_MANAGER: TaskManager = {
        let num_app = get_num_app();
        let mut tasks = [
            TaskControlBlock { task_cx_ptr: 0, task_status: TaskStatus::UnInit
};
            MAX_APP_NUM
        ];
        for i in 0..num_app {
            tasks[i].task_cx_ptr = init_app_cx(i) as * const _ as usize;
            tasks[i].task_status = TaskStatus::Ready;
        }
        TaskManager {
            num_app,
            inner: RefCell::new(TaskManagerInner {
                tasks,
                current_task: 0,
            }),
        }
    };
}

impl TaskManager {
    fn run_first_task(&self) {
        self.inner.borrow_mut().tasks[0].task_status = TaskStatus::Running;
        let next_task_cx_ptr2 = self.inner.borrow().tasks[0].get_task_cx_ptr2();
        let _unused: usize = 0;
        unsafe {
            __switch(
                &_unused as *const _,
            )
        }
    }
}
```

```

        next_task_cx_ptr2,
    );
}

fn mark_current_suspended(&self) {
    let mut inner = self.inner.borrow_mut();
    let current = inner.current_task;
    inner.tasks[current].task_status = TaskStatus::Ready;
}

fn mark_current_exited(&self) {
    let mut inner = self.inner.borrow_mut();
    let current = inner.current_task;
    inner.tasks[current].task_status = TaskStatus::Exited;
}

fn find_next_task(&self) -> Option<usize> {
    let inner = self.inner.borrow();
    let current = inner.current_task;
    (current + 1..current + self.num_app + 1)
        .map(|id| id % self.num_app)
        .find(|id| {
            inner.tasks[*id].task_status == TaskStatus::Ready
        })
}

fn run_next_task(&self) {
    if let Some(next) = self.find_next_task() {
        let mut inner = self.inner.borrow_mut();
        let current = inner.current_task;
        inner.tasks[next].task_status = TaskStatus::Running;
        inner.current_task = next;
        let current_task_cx_ptr2 = inner.tasks[current].get_task_cx_ptr2();
        let next_task_cx_ptr2 = inner.tasks[next].get_task_cx_ptr2();
        core::mem::drop(inner);
        unsafe {
            __switch(
                current_task_cx_ptr2,
                next_task_cx_ptr2,
            );
        }
    } else {
        panic!("All applications completed!");
    }
}

pub fn run_first_task() {
    TASK_MANAGER.run_first_task();
}

fn run_next_task() {
    TASK_MANAGER.run_next_task();
}

fn mark_current_suspended() {
    TASK_MANAGER.mark_current_suspended();
}

```

```

}

fn mark_current_exited() {
    TASK_MANAGER.mark_current_exited();
}

pub fn suspend_current_and_run_next() {
    mark_current_suspended();
    run_next_task();
}

pub fn exit_current_and_run_next() {
    mark_current_exited();
    run_next_task();
}

```

The above code also calls **init\_app\_cx** of the Loader submodule. Therefore, you also need to add code in "loader.rs".

```

// os/src/loader.rs

pub fn init_app_cx(app_id: usize) -> &'static TaskContext {
    KERNEL_STACK[app_id].push_context(
        TrapContext::app_init_context(get_base_i(app_id),
    USER_STACK[app_id].get_sp()),
        TaskContext::goto_restore(),
    )
}

```

```

pub fn init_app_cx(app_id: usize) -> &'static TaskContext {
    KERNEL_STACK[app_id].push_context(
        TrapContext::app_init_context(get_base_i(app_id), USER_STACK[app_id].get_sp()),
        TaskContext::goto_restore(),
    )
}

```

loader.rs [dos] (16:30 12/11/2021) 89,1 Bot

Analyzing this part of the code shows that KernelStack presses a Trap context and then a task context. The TaskContext is constructed by **TaskContext::goto\_restore()**.

## 4. sys\_yield & sys\_exit

To implement the two system calls, we need to modify file "process.rs" as below.

```

use crate::task::{
    suspend_current_and_run_next,
    exit_current_and_run_next,
};

pub fn sys_exit(exit_code: i32) -> ! {
    println!("[kernel] Application exited with code {}", exit_code);
    exit_current_and_run_next();
    panic!("Unreachable in sys_exit!");
}

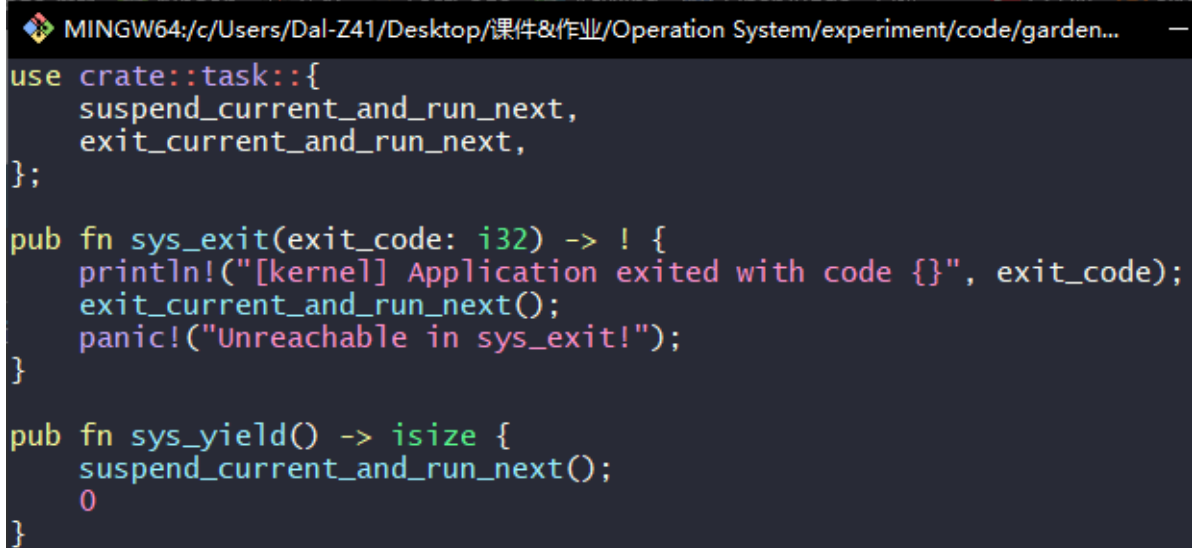
pub fn sys_yield() -> isize {

```

```

suspend_current_and_run_next();
0
}

```



```

MINGW64;C:/Users/Dal-Z41/Desktop/课件&作业/Operation System/experiment/code/garden...
use crate::task::{
    suspend_current_and_run_next,
    exit_current_and_run_next,
};

pub fn sys_exit(exit_code: i32) -> ! {
    println!("[kernel] Application exited with code {}", exit_code);
    exit_current_and_run_next();
    panic!("Unreachable in sys_exit!");
}

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

```

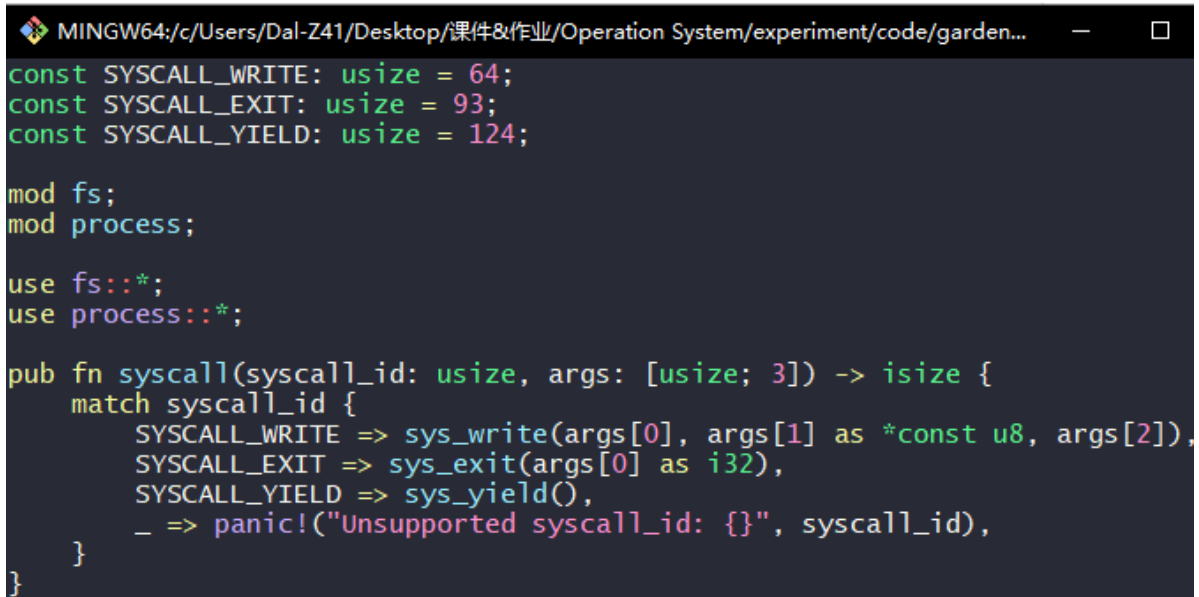
Besides, we need to change to **"mod.rs"** to add `sys_yield` to the handling of system calls. Both system calls are implemented based on the interface provided by the `Task` submodule. We need to add some codes to it.

```

const SYSCALL_YIELD: usize = 124;

SYSCALL_YIELD => sys_yield(),

```



```

MINGW64;C:/Users/Dal-Z41/Desktop/课件&作业/Operation System/experiment/code/garden...
const SYSCALL_WRITE: usize = 64;
const SYSCALL_EXIT: usize = 93;
const SYSCALL_YIELD: usize = 124;

mod fs;
mod process;

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

pub fn syscall(syscall_id: usize, args: [usize; 3]) -> isize {
    match syscall_id {
        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(),
        _ => panic!("Unsupported syscall_id: {}", syscall_id),
    }
}

```

## 5. 完善程序

We need to comment out the `run_next_app()` part of the `trap` submodule. Also, comment out `mv sp, a0` in `__restore` in `trap.S`. Then, file `"main.rs"` is as below.



```

MINGW64:/c/Users/Dal-Z41/Desktop/课件&作业/Operation System/experiment/code/garden.
#![no_std]
#![no_main]
#![feature(asm)]
#![feature(global_asm)]
#![feature(panic_info_message)]

#[macro_use]
mod console;
mod lang_items;
mod sbi;
mod syscall;
mod trap;
mod loader;
mod config;
mod task;

global_asm!(include_str!("entry.asm"));
global_asm!(include_str!("link_app.S"));

fn clear_bss() {
    extern "C" {
        fn sbss();
        fn ebss();
    }
    (sbss as usize..ebss as usize).for_each(|a| unsafe { (a as *mut u8).write_volatile(0) });
}

#[no_mangle]
pub fn rust_main() -> ! {
    clear_bss();
    println!("[Kernel] Hello, world!");
    trap::init();
    loader::load_apps();
    task::run_first_task();
    panic!("Unreachable in rust_main!");
}
~
~
~
~
~
~
~
~
main.rs [unix] (15:52 12/11/2021)

```

## 6. 运行结果

```

root@iZuf6bff8a2k5qecbdmwnyZ:~/user# make build
Compiling user_lib v0.1.0 (/root/user)
Finished release [optimized] target(s) in 1.83s
src/bin/02write_c.rs src/bin/00write_a.rs src/bin/01write_b.rs
target/riscv64gc-unknown-none-elf/release/02write_c target/riscv64gc-unknown-none-elf/release/00write_a target/riscv64gc-unknown-none-elf/release/01write_b
target/riscv64gc-unknown-none-elf/release/02write_c.bin target/riscv64gc-unknown-none-elf/release/00write_a.bin target/riscv64gc-unknown-none-elf/release/01write_b.bin
rust-objcopy --binary-architecture=riscv64 target/riscv64gc-unknown-none-elf/release/02write_c --strip-all -O binary target/riscv64gc-unknown-none-elf/release/02write_c.bin; rust-objcopy --binary-architecture=riscv64 target/riscv64gc-unknown-none-elf/release/00write_a --strip-all -O binary target/riscv64gc-unknown-none-elf/release/00write_a.bin; rust-objcopy --binary-architecture=riscv64 target/riscv64gc-unknown-none-elf/release/01write_b --strip-all -O binary target/riscv64gc-unknown-none-elf/release/01write_b.bin;

```

```
root@iZuf69gjp4acy5gbxody49Z:~/os# cargo build
  Updating `ustc` index
  Updating git repository `https://github.com/rcore-os/riscv`
  Compiling memchr v2.4.1
  Compiling semver-parser v0.7.0
  Compiling regex-syntax v0.6.25
  Compiling lazy_static v1.4.0
  Compiling log v0.4.14
  Compiling cfg-if v1.0.0
  Compiling bit_field v0.10.1
  Compiling os v0.1.0 (/root/os)
  Compiling spin v0.5.2
  Compiling bitflags v1.3.2
  Compiling semver v0.9.0
  Compiling rustc_version v0.2.3
  Compiling aho-corasick v0.7.18
  Compiling bare-metal v0.2.5
  Compiling regex v1.5.4
  Compiling riscv-target v0.1.2
  Compiling riscv v0.6.0 (https://github.com/rcore-os/riscv#b6c469f0)
  Finished dev [unoptimized + debuginfo] target(s) in 34.64s
root@iZuf69gjp4acy5gbxody49Z:~/os# cargo run
  Finished dev [unoptimized + debuginfo] target(s) in 0.02s
  Running `target/riscv64gc-unknown-none-elf/debug/os`
```