中山大學

SUN YAT-SEN UNIVERSITY

操作系统实验 4

用户程序与系统调用

姓　　名：　王宇

学　　号：　22330111

教学班号：　计科 1 班

专　　业：　计算机科学与技术

院　　系：　计算机学院

2023 学年第二学期

一. 用户程序

1. 编译用户程序

用户态库被定义在 `pkg/lib` 中，在用户程序中，编辑 `Cargo.toml`，使用如下方式引用用户库：

```
[dependencies]
lib = { path="../../lib", package="yslib"}
```

一个简单的用户程序示例如下所示，同样存在于 app/hello/src/main.rs 中：

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

use lib::*;

extern crate lib;

fn main() -> isize {
    println!("Hello, world!!!");

    233
}
entry!(main);
```

下面是在 `app/hello` 下面执行 `cargo build` 的结果。可以看到能够正常编译一个用户程序。

```
PS C:\Users\wy106\repository\YSOS2\0x04\pkg\app\hello> cargo build --release
warning: `C:\Users\wy106\scoop\persist\rustup\.cargo\config` is deprecated in favor of `config.toml`
note: if you need to support cargo 1.38 or earlier, you can symlink `config` to `config.toml`
   Compiling core v0.0.0 (C:\Users\wy106\scoop\persist\rustup\.rustup\toolchains\nightly-x86_64-pc-windows-msvc\lib\rustlib\src\rust\library\core)
   Compiling compiler_builtins v0.1.108
   Compiling syn v2.0.48
   Compiling num_enum_derive v0.7.2
   Compiling rustc-std-workspace-core v1.99.0 (C:\Users\wy106\scoop\persist\rustup\.rustup\toolchains\nightly-x86_64-pc-windows-msvc\lib\rustlib\src\rust\library\rustc-std-workspace-core)
   Compiling alloc v0.0.0 (C:\Users\wy106\scoop\persist\rustup\.rustup\toolchains\nightly-x86_64-pc-windows-msvc\lib\rustlib\src\rust\library\alloc)
   Compiling num_enum v0.7.2
   Compiling ysos_syscall v0.4.0 (C:\Users\wy106\repository\YSOS2\0x04\pkg\syscall)
   Compiling yslib v0.4.0 (C:\Users\wy106\repository\YSOS2\0x04\pkg\lib)
   Compiling ysos_hello v0.1.0 (C:\Users\wy106\repository\YSOS2\0x04\pkg\app\hello)
    Finished `release` profile [optimized] target(s) in 26.43s
```

2. 加载程序文件

为了存储用户程序的相关信息，在 `pkg\boot\src\lib.rs` 中，定义一个 App 结构体。由于 `ElfFile` 中含有一个需要显示标注生命周期的 `&'a[u8]` 字段，所以我们需要在 `App` 结构体中加入这个生命周期标注。当然，实际上我们只需要关心 `'static` 生命周期即可，因为加载进来的文件所持有的生命周期就是 `'static` 。

```rust
const APP_LEN:usize = 16;
/// App information
pub struct App<'a> {
    /// The name of app
    pub name: ArrayString<APP_LEN>,
    /// The ELF file
    pub elf: ElfFile<'a>,
}

pub type AppList = ArrayVec<App<'static>, APP_LEN>;

pub type AppListRef<'a> = Option<&'static ArrayVec<App<'a>, APP_LEN>>;
```

并添加 `loaded_apps` 字段到 `BootInfo` 结构体中。

```rust
/// This structure represents the information that the bootloader passes to the kernel.
pub struct BootInfo {
    /// The memory map
    pub memory_map: MemoryMap,

    /// The offset into the virtual address space where the physical memory is mapped.
    pub physical_memory_offset: u64,

    /// UEFI SystemTable
    pub system_table: SystemTable<Runtime>,

    // Loaded apps
    pub loaded_apps: Option<AppList>,
}
```

在 `pkg/boot/src/fs.rs` 中，补全 `load_apps` 函数如下。**代码中标记了 `FIXME` 的部分是需要补充的内容。**

```rust
/// Load apps into memory, when no fs implemented in kernel
///
/// List all file under "APP" and load them.
pub fn load_apps(bs: &BootServices) -> AppList {
    let mut root = open_root(bs);
    let mut buf = [0; 8];
    let cstr_path = uefi::CStr16::from_str_with_buf("\\APP\\", &mut buf).unwrap();
    // FIXME: get handle for \APP\ dir
    let mut handle = {
        let handle = root
            .open(cstr_path, FileMode::Read, FileAttribute::empty())
            .expect("Failed to open APP");
        match handle.into_type().expect("Failed to into_type") {
            FileType::Dir(dir) => dir,
            _ => panic!("Invalid file type which is not a directory"),
        }
    };
```

```
let mut apps = ArrayVec::new();
let mut entry_buf = [0u8; 0x100];

loop {
    let info = handle
        .read_entry(&mut entry_buf)
        .expect("Failed to read entry");
    debug!("Entry is {:?}", info);

    match info {
        Some(entry) => {
            if entry
                .file_name()
                .as_slice_with_nul()
                .starts_with(&[Char16::try_from('.').unwrap()])
            {
                continue;
            }
            // FIXME: get handle for app binary file
            let mut file = {
                handle
                    .open(entry.file_name(), FileMode::Read, entry.attribute())
                    .expect("Failed to open file")
                    .into_regular_file()
                    .expect("Failed to into_regular_file")
            };
            if file.is_directory().unwrap_or(true) {
                continue;
            }

            let elf = {
                // FIXME: load file with `load_file` function
                let input = load_file(bs, &mut file);
                // FIXME: convert file to `ElfFile`
                ElfFile::new(input).unwrap()
            };

            let mut name = ArrayString::<16>::new();
            entry.file_name().as_str_in_buf(&mut name).unwrap();

            apps.push(App { name, elf });
        }
        None => break,
    }
}

info!("Loaded {} apps", apps.len());
```

```
    apps
}
```

在 `boot/src/main.rs` 中，`main` 函数中加载好内核的 `ElfFile` 之后，根据配置选项按需加载用户程序，并将其信息传递给内核：

```
// ...

let apps = if config.load_apps {
    info!("Loading apps...");
    Some(load_apps(system_table.boot_services()))
} else {
    info!("Skip loading apps");
    None
};

// ...
```

```
// construct BootInfo
let bootinfo = BootInfo {
    // ...
    loaded_apps: apps,
};
```

修改 `ProcessManager` 的定义与初始化逻辑，将 `AppList` 添加到 `ProcessManager` 中：

```
pub struct ProcessManager {
    // ...
    app_list: boot::AppListRef,
}
```

最后修改 `kernel/src/proc/mod.rs` 的 `init` 函数：

```
/// init process manager
pub fn init(boot_info: &'static boot::BootInfo) {
    // ...
    let app_list = boot_info.loaded_apps.as_ref();
    manager::init(kproc, app_list);
}
```

之后，在 `kernel/src/proc/mod.rs` 中，定义一个 `list_app` 函数，用于列出当前系统中的所有用户程序和相关信息：

```
pub fn list_app() {
    x86_64::instructions::interrupts::without_interrupts(|| {
        let app_list = get_process_manager().app_list();
        if app_list.is_none() {
            println!("[!] No app found in list!");
            return;
```

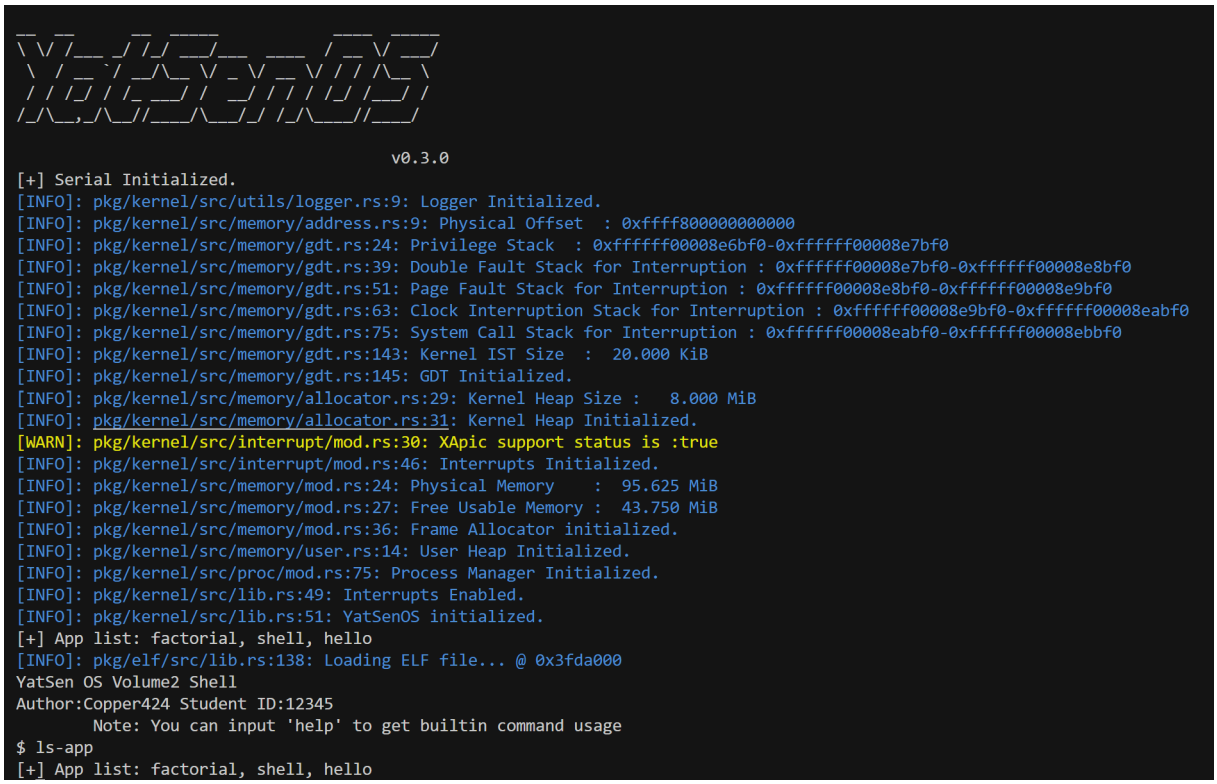
```
        }

        let apps = app_list
            .unwrap()
            .iter()
            .map(|app| app.name.as_str())
            .collect::<Vec<&str>>()
            .join(", ");

        // TODO: print more information like size, entry point, etc.

        println!("[+] App list: {}", apps);
    });
}
```

下面是使用 `list_app` 函数列出当前可用用户态程序的测试结果。



3. 生成用户程序

在 `kernel/src/proc/mod.rs` 中，添加 `spawn` 和 `elf_spawn` 函数，将 ELF 文件从列表中取出，并生成用户程序：

```
pub fn spawn(name: &str) -> Option<ProcessId> {
    let app = x86_64::instructions::interrupts::without_interrupts(|| {
        let app_list = get_process_manager().app_list()?;
        app_list.iter().find(|&app| app.name.eq(name))
```

```
    })?;

    elf_spawn(name.to_string(), &app.elf)
}

pub fn elf_spawn(name: String, elf: &ElfFile) -> Option<ProcessId> {
    let pid = x86_64::instructions::interrupts::without_interrupts(|| {
        let manager = get_process_manager();
        let process_name = name.to_lowercase();
        let parent = Arc::downgrade(&manager.current());
        let pid = manager.spawn(elf, name, Some(parent), None);

        debug!("Spawned process: {}#{}", process_name, pid);
        pid
    });

    Some(pid)
}
```

接下来，在 `pkg\kernel\src\proc\manager.rs` 中，为 `ProcessManager` 实现 `spawn` 函数。

```
 pub fn spawn(
        &self,
        elf: &ElfFile,
        name: String,
        parent: Option<Weak<Process>>,
        proc_data: Option<ProcessData>,
    ) -> ProcessId {
        let kproc = self.get_proc(&KERNEL_PID).unwrap();
        let page_table = kproc.read().clone_page_table();
        let proc = Process::new(name, parent, page_table, proc_data);
        let pid = proc.pid();

        // FIXME: load elf to process pagetable
        proc.write().load_elf(elf);
        // debug!("loading elf to process pagetable");
        // FIXME: alloc new stack for process
        let entry = VirtAddr::new(elf.header.pt2.entry_point());
        proc.write()
            .init_stack_frame(entry, VirtAddr::new(STACK_INIT_TOP));
        // FIXME: mark process as ready
        proc.write().pause();

        trace!("New {:#?}", &proc);

        // FIXME: something like kernel thread
        self.add_proc(pid, proc);
        self.push_ready(pid);
        pid
    }
```

为了将 `ElfFile` 的内容加载进内存，在 `pkg\kernel\src\proc\process.rs` 中实现 `load_elf` 函数。此函数主要解析 ELF 的文件格式，按照其格式的要求，将其正确地映射到内存中。

```
pub fn load_elf(&mut self, elf: &ElfFile) {
        let frame_allocator = &mut *get_frame_alloc_for_sure();
        let mut page_table_mapper = self
            .page_table
            .as_ref()
            .expect("page table did not exist!\n")
            .mapper();
        // map elf segments to new frames
        if let Err(e) = elf::load_elf(
            elf,
            PHYSICAL_OFFSET.get().cloned().unwrap(),
            &mut page_table_mapper,
            frame_allocator,
            true,
        ) {
            debug!("Failed to load ELF: {:?}", e);
        }
        // map and allocate stack
        if let Err(e) = elf::map_range(
            STACK_INIT_BOT,
            STACK_DEF_PAGE,
            &mut page_table_mapper,
            frame_allocator,
            true,
        ) {
            debug!("Failed to map stack: {:?}", e);
        }
        const STACK_INIT_END: u64 = STACK_INIT_BOT + STACK_DEF_SIZE;
        let stack_segment = PageRange {
            start: Page::containing_address(VirtAddr::new(STACK_INIT_BOT)),
            end: Page::containing_address(VirtAddr::new(STACK_INIT_END)),
        };
        let code_segment: Vec<PageRange> = elf
            .program_iter()
            .filter(|p| p.get_type().unwrap() == elf::program::Type::Load)
            .map(|segment_header| {
                                                        let    start    =
Page::containing_address(VirtAddr::new(segment_header.virtual_addr()));
                let end = Page::containing_address(VirtAddr::new(
                    segment_header.virtual_addr() + segment_header.mem_size(),
                ));
                PageRange { start, end }
            })
            .collect();
        self.proc_data.as_mut().unwrap().stack_pages = stack_segment.count();
        self.proc_data.as_mut().unwrap().code_pages = code_segment
            .iter()
```

```
            .map(|code_segment| code_segment.count())
            .sum();
        self.proc_data.as_mut().unwrap().stack_segment = Some(stack_segment);
        self.proc_data.as_mut().unwrap().code_segment = Some(code_segment);
    }
```

二. 系统调用的实现

1. 调用约定

在 `src/interrupt/syscall/mod.rs` 中，根据系统调用号来调用对应的处理函数。

```
pub fn dispatcher(context: &mut ProcessContext) {
    let args = super::syscall::SyscallArgs::new(
        Syscall::from(context.regs.rax),
        context.regs.rdi,
        context.regs.rsi,
        context.regs.rdx,
    );

    match args.syscall {
        // fd: arg0 as u8, buf: &[u8] (ptr: arg1 as *const u8, len: arg2)
        Syscall::Read => {
            context.set_rax(sys_read(&args));
        }
        // fd: arg0 as u8, buf: &[u8] (ptr: arg1 as *const u8, len: arg2)
        Syscall::Write => {
            context.set_rax(sys_write(&args));
        }

        // None -> pid: u16
        Syscall::GetPid => {
            context.set_rax(crate::proc::get_pid().0 as usize);
        }

        // path: &str (ptr: arg0 as *const u8, len: arg1) -> pid: u16
        Syscall::Spawn => {
            context.set_rax(spawn_process(&args));
        }
        // ret: arg0 as isize
        Syscall::Exit => {
            exit_process(&args, context);
        }
        // pid: arg0 as u16 -> status: isize
        Syscall::WaitPid => {
```

```
            context.set_rax(service::waitpid(&args) as usize);
        }
        // pid: arg0 as u16
        Syscall::Kill => {
            sys_kill(&args, context);
        }

        // None
        Syscall::Stat => {
            list_process();
        }
        // None
        Syscall::ListApp => {
            list_app();
        }

        // ------------------------------------------------------
        // NOTE: following syscall examples are implemented
        // ------------------------------------------------------

        // layout: arg0 as *const Layout -> ptr: *mut u8
        Syscall::Allocate => context.set_rax(sys_allocate(&args)),
        // ptr: arg0 as *mut u8
        Syscall::Deallocate => sys_deallocate(&args),
        // Unknown
        Syscall::Unknown => warn!("Unhandled syscall: {:x?}", context.regs.rax),
    }
}
```

相应的系统调用实现函数位于 `pkg\kernel\src\interrupt\syscall\service.rs` 。这些函数也一并给出如下。

```
use core::alloc::Layout;

use crate::proc::*;

use super::SyscallArgs;

pub fn spawn_process(args: &SyscallArgs) -> usize {
    // FIXME: get app name by args
    //       - core::str::from_utf8_unchecked
    //       - core::slice::from_raw_parts
    // FIXME: spawn the process by name
    // FIXME: handle spawn error, return 0 if failed
    // FIXME: return pid as usize
    let name = unsafe {
        core::str::from_utf8_unchecked(core::slice::from_raw_parts(
            args.arg0 as *const u8,
            args.arg1,
        ))
```

```
    };
    if let Some(pid) = spawn(name) {
        return pid.0 as usize;
    }
    0
}

pub fn sys_write(args: &SyscallArgs) -> usize {
    // FIXME: get handle by fd
    // FIXME: handle read from fd & return length
    //       - core::slice::from_raw_parts
    // FIXME: return 0 if failed
    let handle_num = args.arg0 as u8;
    if let Some(resource) = crate::proc::handle(handle_num) {
       let buf = unsafe { core::slice::from_raw_parts(args.arg1 as *const u8, args.arg2) };
        if let Some(len) = resource.write(buf) {
            return len;
        }
    }
    0
}

pub fn sys_read(args: &SyscallArgs) -> usize {
    // FIXME: just like sys_write
    let handle_num = args.arg0 as u8;
    if let Some(resource) = crate::proc::handle(handle_num) {
      let buf = unsafe { core::slice::from_raw_parts_mut(args.arg1 as *mut u8, args.arg2) };
        if let Some(len) = resource.read(buf) {
            return len;
        }
    }
    0
}

pub fn exit_process(args: &SyscallArgs, context: &mut ProcessContext) {
    // FIXME: exit process with retcode
    crate::proc::exit(args.arg0 as isize, context);
}

pub fn list_process() {
    // FIXME: list all processes
    print_process_list();
}

pub fn waitpid(args: &SyscallArgs) -> isize {
    let pid = ProcessId(args.arg0 as u16);
    crate::proc::waitpid(pid)
}

pub fn sys_allocate(args: &SyscallArgs) -> usize {
```

```
    let layout = unsafe { (args.arg0 as *const Layout).as_ref().unwrap() };

    if layout.size() == 0 {
        return 0;
    }

    let ret = crate::memory::user::USER_ALLOCATOR
        .lock()
        .allocate_first_fit(*layout);

    match ret {
        Ok(ptr) => ptr.as_ptr() as usize,
        Err(_) => 0,
    }
}

pub fn sys_deallocate(args: &SyscallArgs) {
    let layout = unsafe { (args.arg1 as *const Layout).as_ref().unwrap() };

    if args.arg0 == 0 || layout.size() == 0 {
        return;
    }

    let ptr = args.arg0 as *mut u8;

    unsafe {
        crate::memory::user::USER_ALLOCATOR
            .lock()
            .deallocate(core::ptr::NonNull::new_unchecked(ptr), *layout);
    }
}

pub fn sys_kill(args: &SyscallArgs, context: &mut ProcessContext) {
    // kill process according to the given PID
    let pid = ProcessId(args.arg0 as u16);
    crate::proc::kill(pid, context);
}
```

2. 软中断处理

在 `src/interrupt/syscall/mod.rs` 中，实现 `register_idt` 函数，以便正确地设置其在中断中的行为。

```
pub unsafe fn register_idt(idt: &mut InterruptDescriptorTable) {
    // FIXME: register syscall handler to IDT
    //        - standalone syscall stack
    //        - ring 3
    idt[super::consts::Interrupts::Syscall as u8]
```

```
        .set_handler_fn(syscall_handler)
        .set_stack_index(SYSCALL_IST_INDEX as u16)
        .set_privilege_level(PrivilegeLevel::Ring3);
}

pub extern "C" fn syscall(mut context: ProcessContext) {
    x86_64::instructions::interrupts::without_interrupts(|| {
        super::syscall::dispatcher(&mut context);
    });
}
```

三. 用户态库的实现

1. 动态内存分配

在 `src/memory/user.rs` 中，实现 `init` 函数与 `init_user_heap` 函数。`init` 函数负责在内核初始化时启用(用户)堆内存分配机制，其内部其实是调用了 `init_user_heap` 函数来完成这一点。`init_user_heap` 需要映射堆的内存位置，并且正确地设置相应的权限。最终的堆内存管理用到了外部 `linked_list_allocator` crate 中的 `LockHeap`。

```
// NOTE: export mod user / call in the kernel init / after frame allocator
pub fn init() {
    init_user_heap().expect("User Heap Initialization Failed.");
    info!("User Heap Initialized.");
}

pub fn init_user_heap() -> Result<(), MapToError<Size4KiB>> {
    // Get current pagetable mapper
    let mapper = &mut PageTableContext::new().mapper();
    // Get global frame allocator
    let frame_allocator = &mut *super::get_frame_alloc_for_sure();

    // FIXME: use elf::map_range to allocate & map
    //        frames (R/W/User Access)
    elf::map_range(
        USER_HEAP_START as u64,
        USER_HEAP_PAGE as u64,
        mapper,
        frame_allocator,
        true,
    )?;
    debug!(
        "Init the User heap    : 0x{:16x} - 0x{:16x}",
        USER_HEAP_START,
```

```
        USER_HEAP_START + USER_HEAP_SIZE
    );
    unsafe {
        USER_ALLOCATOR
            .lock()
            .init(USER_HEAP_START as *mut u8, USER_HEAP_SIZE);
    }

    Ok(())
}
```

2. 标准输入输出

为了简化实验，这里的实现并不需要考虑 I/O 缓冲区和批处理的问题。我们实现的 `sys_read` 系统调用只会一次读取 4 个字节。这样在现实的操作系统中会有性能问题，这里出于简单而没有进行相应的考量。

2.1. 标准输出

这里涉及到内核以及用户态库的多个文件，现在以标准输入函数为例，说明从一个用户态程序调用 `sys_read` 函数到操作系统内核中执行的全部过程。

在 `pkg\lib\src\syscall.rs` 中提供了用户态库的 `sys_read` 实现。该函数利用 `syscall` 宏来选择对应的函数实现，由于 `sys_read` 函数有三个参数，所以是选择了 `syscall3` 函数。`syscall3` 函数通过 `int 0x80` 汇编指令触发系统调用，并通过调用惯例中的 `rdi`，`rsi`，`rdx` 寄存器传递参数。

```
#[inline(always)]
pub fn sys_read(fd: u8, buf: &mut [u8]) -> Option<usize> {
    let ret = syscall!(
        Syscall::Read,
        fd as u64,
        buf.as_ptr() as u64,
        buf.len() as u64
    ) as isize;
    if ret.is_negative() {
        None
    } else {
        Some(ret as usize)
    }
}

#[macro_export]
macro_rules! syscall {
    ($n:expr) => {
        $crate::macros::syscall0($n)
```

```
    };
    ($n:expr, $a1:expr) => {
        $crate::macros::syscall1($n, $a1 as usize)
    };
    ($n:expr, $a1:expr, $a2:expr) => {
        $crate::macros::syscall2($n, $a1 as usize, $a2 as usize)
    };
    ($n:expr, $a1:expr, $a2:expr, $a3:expr) => {
        $crate::macros::syscall3($n, $a1 as usize, $a2 as usize, $a3 as usize)
    };
}

#[doc(hidden)]
#[inline(always)]
pub fn syscall3(n: Syscall, arg0: usize, arg1: usize, arg2: usize) -> usize {
    let ret: usize;
    unsafe {
        asm!(
            "int 0x80", in("rax") n as usize,
            in("rdi") arg0, in("rsi") arg1, in("rdx") arg2,
            lateout("rax") ret
        );
    }
    ret
}
```

在中断处理函数中，经过判断系统调用号，会选中位于 `pkg\kernel\src\interrupt\syscall\service.rs` 中的 `sys_read` 函数。该函数内部调用了 `Resource` 类型上面的 `read` 方法。

```
pub fn sys_read(args: &SyscallArgs) -> usize {
    // FIXME: just like sys_write
    let handle_num = args.arg0 as u8;
    if let Some(resource) = crate::proc::handle(handle_num) {
      let buf = unsafe { core::slice::from_raw_parts_mut(args.arg1 as *mut u8, args.arg2) };
        if let Some(len) = resource.read(buf) {
            return len;
        }
    }
    0
}
```

在 `pkg\kernel\src\utils\resource.rs` 中， `Resource` 类型的 `read` 方法实现如下。主要利用了之前代码实现过的 `try_get_key` 函数来从内核的输入缓冲区中读取字符，并将其编码进给出的 `buf` 缓冲区中。

```
pub fn read(&self, buf: &mut [u8]) -> Option<usize> {
        match self {
            Resource::Console(stdio) => match stdio {
```

```
                &StdIO::Stdin => {
                    // FIXME: just read from kernel input buffer
                    if let Some(key) = crate::drivers::input::try_get_key() {
                        if buf.len() < 4 {
                            return None;
                        }
                        let s = key.encode_utf8(buf);
                        return Some(s.len());
                    }
                    Some(0)
                }
                _ => None,
            },
            Resource::Null => Some(0),
        }
    }
```

至此，我们看到了整个 `sys_read` 系统调用从用户态库到内核中实现的全过程。从用户态到内核态的桥梁是 `int 0x80` 汇编指令，它促使系统进入基于中断的系统调用处理函数中，提升了特权权限，加载了内核页表。

2.2. 标准输入

对于标准输入(利用 sys_write 系统调用实现)，我们只关心 `pkg\kernel\src\utils\resource.rs` 中，`Resource` 类型的 `write` 方法实现即可。从用户态到内核态的切换过程是类似的。下面是其实现的代码展示。对于输入的 `buf` ，我们利用 `String::from_utf8_lossy` 函数来将其转换成 `String` 格式以打印。该函数会在遇到不可解析的 UTF-8 字符的时候，将其转换成 `U+FFFD REPLACEMENT CHARACTER` 或者 `` 而不是返回 `Err(_)` ，避免了一些错误处理。

```
pub fn write(&self, buf: &[u8]) -> Option<usize> {
        match self {
            Resource::Console(stdio) => match *stdio {
                StdIO::Stdin => None,
                StdIO::Stdout => {
                    print!("{}", String::from_utf8_lossy(buf));
                    Some(buf.len())
                }
                StdIO::Stderr => {
                    warn!("{}", String::from_utf8_lossy(buf));
                    Some(buf.len())
                }
            },
            Resource::Null => Some(buf.len()),
        }
    }
```

2.3. 按行读取的 `read_line`

在 `pkg\lib\src\io.rs` 中，为用户态库实现按行读取输入内容的 `read_line` 函数。该函数借助 `sys_read` 系统调用来一个字符一个字符地读取内容。该函数能够对一些特殊都控制字符做出响应，例如，当输入换行符的时候结束读取，当输入 `\x04` (通常对应于 Ctrl + D) 的时候清除之前的输入，结束读取，当输入退格字符‘x08’以及‘x7F’的时候，向标准输出写入“"x08x20x08”” (一个退格，一个空格，再一个退格)，来退格并将原来位置上的字符替换成空格(只发送退格只会使得光标回退，而不会清除原来位置上面的显示字符)。

```
pub fn read_line(&self) -> String {
        // FIXME: allocate string
        let mut buf = String::new();
        // FIXME: read from input buffer
        //       - maybe char by char?
        let mut char_buf = [0u8; 4];
        while let Some(len) = sys_read(0, &mut char_buf) {
            if len > 0 {
                let ch = String::from_utf8_lossy(&char_buf[..len])
                    .chars()
                    .next()
                    .unwrap();
                // FIXME: handle backspace / enter...
                match ch {
                    '\n' | '\r' => {
                        stdout().write("\n");
                        break;
                    }
                    '\x04' => {
                        buf.clear();
                        buf.push(ch);
                        break;
                    }
                    '\x08' | '\x7F' => {
                        if !buf.is_empty() {
                            io::print!("\x08\x20\x08");
                            buf.pop();
                        }
                    }
                    // ignore other control character
                    '\x00'..='\x1F' => {}
                    c => {
                        buf.push(ch);
                        // echo the input character
                        io::print!("{}", c);
                    }
                }
            } else {
                // len == 0
                continue;
```

```
            }
        }
        // FIXME: return string
        buf
    }
```

3. 进程的退出与杀死

3.1. 进程的退出

用户程序通过主动调用 `exit` 系统调用来通知内核释放资源。由于该系统调用是通过中断触发的，而硬件会自动地将中断前栈帧信息压入中断栈中。从而，在系统调用中，操作系统获知了用户程序的 CPU 上下文，并且可以像时钟中断一样，控制退出中断时的 CPU 上下文，从而实现进程的切换。

在 `pkg\kernel\src\proc\mod.rs` 中，添加 `exit` 函数。进程的退出过程主要包括清理进程的资源，并切换到下一个可以执行的进程。

```rust
pub fn exit(ret: isize, context: &mut ProcessContext) {
    x86_64::instructions::interrupts::without_interrupts(|| {
        let manager = get_process_manager();
        manager.kill_current(ret); // FIXME: implement this for ProcessManager
        manager.switch_next(context);
    })
}
```

在 `pkg\kernel\src\proc\manager.rs` 中，为 `ProcessManager` 添加 `kill` 以及 `kill_current` 方法。`kill_current` 是 `kill` 的一个特例，专门用于杀死当前进程。`kill` 方法在判断了进程存在且不处于 `ProgramStatus::Dead` 的情况下，调用 `Process` 的 `kill` 方法来杀死进程。在 lab3 中，已经为 `Process` 实现过 `kill` 方法，这里直接进行代码复用。

```rust
pub fn kill_current(&self, ret: isize) {
    self.kill(processor::get_pid(), ret);
}
pub fn kill(&self, pid: ProcessId, ret: isize) {
        let proc = self.get_proc(&pid);

        if proc.is_none() {
            warn!("Process #{} not found.", pid);
            return;
        }

        let proc = proc.unwrap();

        if proc.read().status() == ProgramStatus::Dead {
```

```
            warn!("Process #{} is already dead.", pid);
            return;
        }

        trace!("Kill {:#?}", &proc);

        proc.kill(ret);
    }
```

3.2. 进程的杀死

注:此部分内容没有在本节的实验文档中直接给出，是我个人自行补充的内容。

与进程的退出类似，进程的杀死(sys_kill)也是一个常见的用于结束进程的系统系统调用。不过，进程的退出强调当前进程主动地退出，而进程的杀死则是根据进程号杀死给定的进程。进程的杀死是由某个进程触发，作用到这个进程本身或者其它的进程的一个操作。两者的辩证关系有点类似于 01 信号量与互斥锁之间的关系。

在 `pkg\syscall\src\lib.rs` 中，添加 `kill` 的系统调用号。

```
#[repr(usize)]
#[derive(Clone, Debug, FromPrimitive)]
pub enum Syscall {
    // ...
    Kill = 62,
    // ...
}
```

在 `pkg\lib\src\syscall.rs` 中，添加调用 `syscall` 宏的用户态库函数 `sys_kill` 。

```
#[inline(always)]
pub fn sys_kill(pid: u16) {
    syscall!(Syscall::Kill, pid);
}
```

经过类似于标准输入中的用户态到内核态的转换操作，我们会调用位于 `pkg\kernel\src\interrupt\syscall\service.rs` 的 `sys_kill` 函数。此函数将系统调用传递过来的参数(进程号)，转换成 `ProcessId` ，并借助于 `crate::proc::kill` 函数来杀死给定的进程。

```
pub fn sys_kill(args: &SyscallArgs, context: &mut ProcessContext) {
    // kill process according to the given PID
    let pid = ProcessId(args.arg0 as u16);
    crate::proc::kill(pid, context);
}
```

在 `pkg\kernel\src\proc\mod.rs` 中实现 `crate::proc::kill` 函数。该函数首先判断要结束的进程是不是当前进程。若不是，就调用 `ProcessManager` 的 `kill` 方法来结束进程; 否则，则要结束

的进程就是当前进程，我们需要采取类似于 `exit` 系统调用的做法来结束进程，并切换到下一个进程。

```
pub fn kill(pid: ProcessId, context: &mut ProcessContext) {
    x86_64::instructions::interrupts::without_interrupts(|| {
        let manager = get_process_manager();
        if pid == processor::get_pid() {
            manager.kill_current(-1);
            manager.switch_next(context);
        } else {
            manager.kill(pid, -1);
        }
    })
}
```

4. 进程的创建与等待

4.1. 进程的创建

在本实验中，进程的创建实现为 `spawn` 系统调用。类似于之前的系统调用，我们需要实现用户态库的内容，并在系统调用的中断处理函数中选择对应于 `spawn` 的中断处理函数。在用户态库中会判断需要执行的程序是否存在，如不存在会给出错误。接着，中断处理函数会选中 `pkg/kernel/src/proc/mod.rs` 中的 `spawn` 函数并进一步地调用 `elf_spawn` 函数。接下来，`ProcessManager` 的 `spawn` 方法将 ELF 文件从列表中取出，并生成 (spawn) 用户程序。

```
pub fn spawn(name: &str) -> Option<ProcessId> {
    let app = x86_64::instructions::interrupts::without_interrupts(|| {
        let app_list = get_process_manager().get_app_list()?;
        app_list.iter().find(|&app| app.name.eq(name))
    })?;
    elf_spawn(name.to_string(), &app.elf)
}

pub fn elf_spawn(name: String, elf: &ElfFile) -> Option<ProcessId> {
    let pid = x86_64::instructions::interrupts::without_interrupts(|| {
        let manager = get_process_manager();
        let process_name = name.to_lowercase();
        let parent = Arc::downgrade(&manager.current());
        let pid = manager.spawn(elf, name, Some(parent), None);

        debug!("Spawned process: {}#{}", process_name, pid);
        pid
    });

    Some(pid)
}
```

在关闭中断之后，进入到 `pkg/kernel/src/proc/manager.rs` 中的 `spawn` 方法的实现。该函数，先克隆内核的页表，创建一份新的进程数据块。再利用 `load_elf` 文件根据给出的 `ElfFile` 将其内容从 ELF 文件格式映射到内存中。接着分配好新的进程的栈空间，将其置于就绪状态，添加到进程表以及就绪队列中。

```
pub fn spawn(
    &self,
    elf: &ElfFile,
    name: String,
    parent: Option<Weak<Process>>,
    proc_data: Option<ProcessData>,
) -> ProcessId {
    let kproc = self.get_proc(&KERNEL_PID).unwrap();
    let page_table = kproc.read().clone_page_table();
    let proc = Process::new(name, parent, page_table, proc_data);
    let pid = proc.pid();

    // FIXME: load elf to process pagetable
    proc.write().load_elf(elf);
    // debug!("loading elf to process pagetable");
    // FIXME: alloc new stack for process
    let entry = VirtAddr::new(elf.header.pt2.entry_point());
    proc.write()
        .init_stack_frame(entry, VirtAddr::new(STACK_INIT_TOP));
    // FIXME: mark process as ready
    proc.write().pause();

    trace!("New {:#?}", &proc);

    // FIXME: something like kernel thread
    self.add_proc(pid, proc);
    self.push_ready(pid);
    pid
}

pub fn load_elf(&mut self, elf: &ElfFile) {
        let frame_allocator = &mut *get_frame_alloc_for_sure();
        let mut page_table_mapper = self
            .page_table
            .as_ref()
            .expect("page table did not exist!\n")
            .mapper();
        // map elf segments to new frames
        if let Err(e) = elf::load_elf(
            elf,
            PHYSICAL_OFFSET.get().cloned().unwrap(),
            &mut page_table_mapper,
            frame_allocator,
            true,
        ) {
```

```
            debug!("Failed to load ELF: {:?}", e);
        }
        // map and allocate stack
        if let Err(e) = elf::map_range(
            STACK_INIT_BOT,
            STACK_DEF_PAGE,
            &mut page_table_mapper,
            frame_allocator,
            true,
        ) {
            debug!("Failed to map stack: {:?}", e);
        }
        const STACK_INIT_END: u64 = STACK_INIT_BOT + STACK_DEF_SIZE;
        let stack_segment = PageRange {
            start: Page::containing_address(VirtAddr::new(STACK_INIT_BOT)),
            end: Page::containing_address(VirtAddr::new(STACK_INIT_END)),
        };
        let code_segment: Vec<PageRange> = elf
            .program_iter()
            .filter(|p| p.get_type().unwrap() == elf::program::Type::Load)
            .map(|segment_header| {
                                                    let    start    =
Page::containing_address(VirtAddr::new(segment_header.virtual_addr()));
                let end = Page::containing_address(VirtAddr::new(
                    segment_header.virtual_addr() + segment_header.mem_size(),
                ));
                PageRange { start, end }
            })
            .collect();
        self.proc_data.as_mut().unwrap().stack_pages = stack_segment.count();
        self.proc_data.as_mut().unwrap().code_pages = code_segment
            .iter()
            .map(|code_segment| code_segment.count())
            .sum();
        self.proc_data.as_mut().unwrap().stack_segment = Some(stack_segment);
        self.proc_data.as_mut().unwrap().code_segment = Some(code_segment);
    }
```

4.2. 进程的等待

在本实验中，进程的等待实现为 `wait_pid` 系统调用。

在 `pkg/lib/src/syscall.rs` 中，实现用户态库的 `wait_pid` 系统调用部分。在用户态中，`wait_pid` 不停的轮询内核态系统调用的结果。这里，我们约定返回值大于等于零就是已经退出。

```
#[inline(always)]
pub fn sys_wait_pid(pid: u16) -> isize {
    // FIXME: try to get the return value for process
    //        loop & halt until the process is finished
```

```
    loop {
        let ret = syscall!(Syscall::WaitPid, pid as usize) as isize;
        if ret >= 0 {
            return ret;
        }
    }
}
```

经过一些用户态到内核态的转换，以及中断处理函数的选择。在 `pkg/kernel/src/proc/manager.rs` 中，实现 `waitpid` 。`waitpid` 函数调用了另外一个 `ProcessManager` 上面的 `get_proc_exit_code` 方法，这个方法在进程死亡时，将进程的退出值返回，否则返回一个 `None` 。从而在 `waitpid` 中，当 `get_proc_exit_code` 返回为空的时候，我们赋予其一个默认值 `-1` 标志着该进程还未退出。

```
    pub fn waitpid(&self, pid: ProcessId) -> isize {
        self.get_proc_exit_code(pid).unwrap_or(-1)
    }

    pub fn get_proc_exit_code(&self, pid: ProcessId) -> Option<isize> {
        if let Some(proc) = self.processes.read().get(&pid) {
            let proc = proc.read();
            if proc.status() == ProgramStatus::Dead {
                return proc.exit_code();
            }
        }
        None
    }
```

四. 运行 Shell

Shell 是操作系统内核与人交互的一种重要方式。在本实验中，Shell 主要实现下面的一些功能:

- 列出帮助信息
- 进程的创建以及杀死
- 列出所有可用的用户程序
- 打印进程列表

并且，在输入不符合任何命令格式时，原样输出。下面是 Shell 的具体实现代码。

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

use lib::{vec::Vec, *};
```

```rust
extern crate lib;
mod help;
mod proc;

const NAME: &str = "Copper424";
const STUDENT_ID: &str = "12345";

fn main() -> isize {
    println!("YatSen OS Volume2 Shell");
    println!("Author:{} Student ID:{}", NAME, STUDENT_ID);
    println!("Note: You can input \'help\' to get builtin command usage");

    loop {
        print!("$ ");
        let line = lib::stdin().read_line();
        let line_arr: Vec<&str> = line.split(' ').collect();
        match *line_arr.first().unwrap() {
            "help" => {
                help::print_help_infomation();
            }
            // "\x04" stands for ^D or "ctrl + d"
            "exit" | "\x04" => {
                println!("exit the shell. Bye~");
                break;
            }
            "ps" => {
                lib::sys_print_process_list();
            }
            "ls-app" => {
                lib::sys_list_app();
            }
            "exec" => {
                proc::spawn(&line_arr);
            }
            "nohup" => {
                proc::nohup(&line_arr);
            }
            "kill" => {
                proc::kill(&line_arr);
            }
            s => {
                if s.is_empty() {
                    print!("\n");
                    continue;
                }
                println!("You said: \"{}\"", s);
            }
        }
    }
    0
```

```
}

entry!(main);
```

其中，`proc` 模块中实现了 `spawn` 函数和 `kill` 函数，其内容如下:

```
use lib::{print, println, sys_wait_pid, vec::Vec};

pub fn spawn(line_arr: &Vec<&str>) {
    if line_arr.len() != 2 {
        println!("cannot find the app name for execution. Usage: exec <app name>");
        return;
    }
    let pid = lib::sys_spawn(line_arr[1]);
    let _exit_code = sys_wait_pid(pid);
    // println!("The exit code of PID {} is {}", pid, _exit_code);
}

pub fn nohup(line_arr: &Vec<&str>) {
    if line_arr.len() != 2 {
        println!("cannot find the app name for execution. Usage: nohup <app name>");
        return;
    }
    let _pid = lib::sys_spawn(line_arr[1]);
    // println!("Process {} is running in the background", _pid);
}

pub fn kill(line_arr: &Vec<&str>) {
    if line_arr.len() != 2 {
        println!("The PID of the process to be killed is not found. Usage: kill <PID>");
        return;
    }
    match line_arr[1].parse::<u16>() {
        Ok(pid) => {
            lib::sys_kill(pid);
            print!("\n");
        }
        Err(e) => println!("failed to parse PID:{}", e),
    }
}
```

`help` 模块中负责打印帮助信息，其内容如下:

```
extern crate lib;


pub fn print_help_infomation() {
    use ::lib::println;
    println!("YatSen OSv2 Shell");
```

```
    println!("Builtin Command Usage:");

    println!("\thelp: print this help message");
    println!("\tls-app: list all available apps");
    println!("\tps: show process list");
    println!("\texec: execute given app");
    println!("\tnohup: execute given app in the background");
    println!("\tkill: kill process referring to PID");
    println!("\texit: exit shell");
}
```

下面的 图 1 展示了在系统启动后， Shell 的初始化页面以及输入 `help` 内建指令后打印输出的帮助信息。

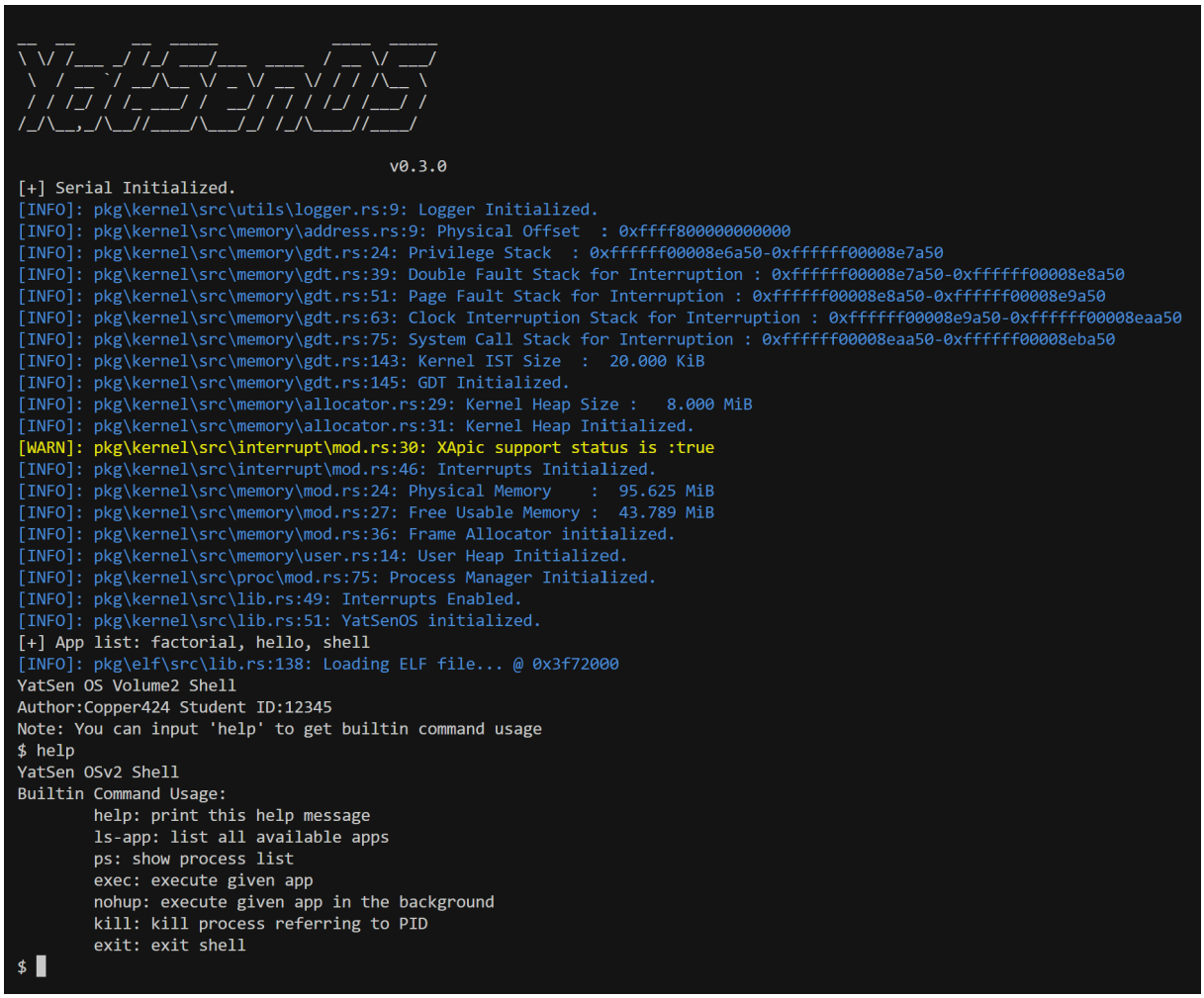


图 1　shell 初始化页面以及帮助信息的输出

下面的 图 2 展示了 `ls-app` 命令、 `ps` 命令、 `exec` 命令以及 `exit` 命令的输出。

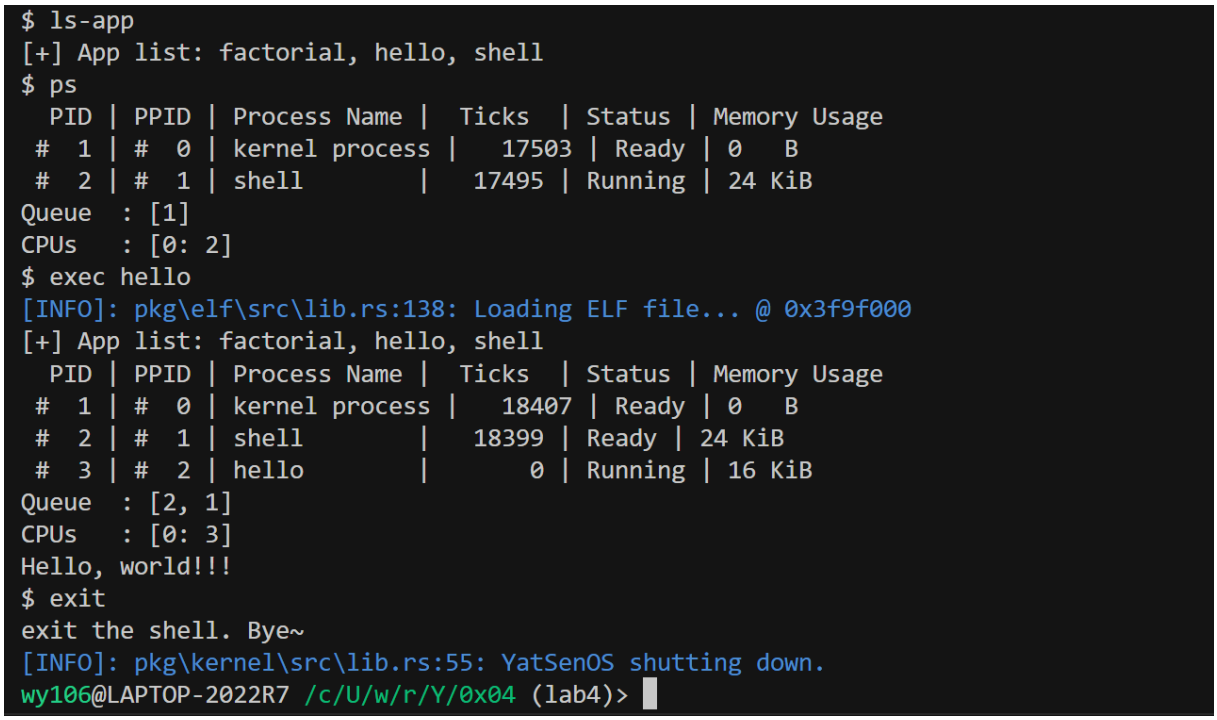


图 2　ls-app,ps,exec 以及 exit 命令的输出

1. 斐波那契数列测试程序

下面这个是用于计算斐波那契数列的程序。我们将利用其测试 `nohup` 命令以及 `kill` 命令。

```
const MOD: u64 = 1000000007;

fn factorial(n: u64) -> u64 {
    if n == 0 {
        1
    } else {
        n * factorial(n - 1) % MOD
    }
}

fn main() -> isize {
    print!("Input n: ");

    let input = lib::stdin().read_line();

    // prase input as u64
    let n = input.parse::<u64>().unwrap();

    if n > 1000000 {
        println!("n must be less than 1000000");
        return 1;
    }
```

```
    // calculate factorial
    let result = factorial(n);

    // print system status
    sys_stat();

    // print result
    println!("The factorial of {} under modulo {} is {}.", n, MOD, result);

    0
}

entry!(main);
```

下面是测试结果。

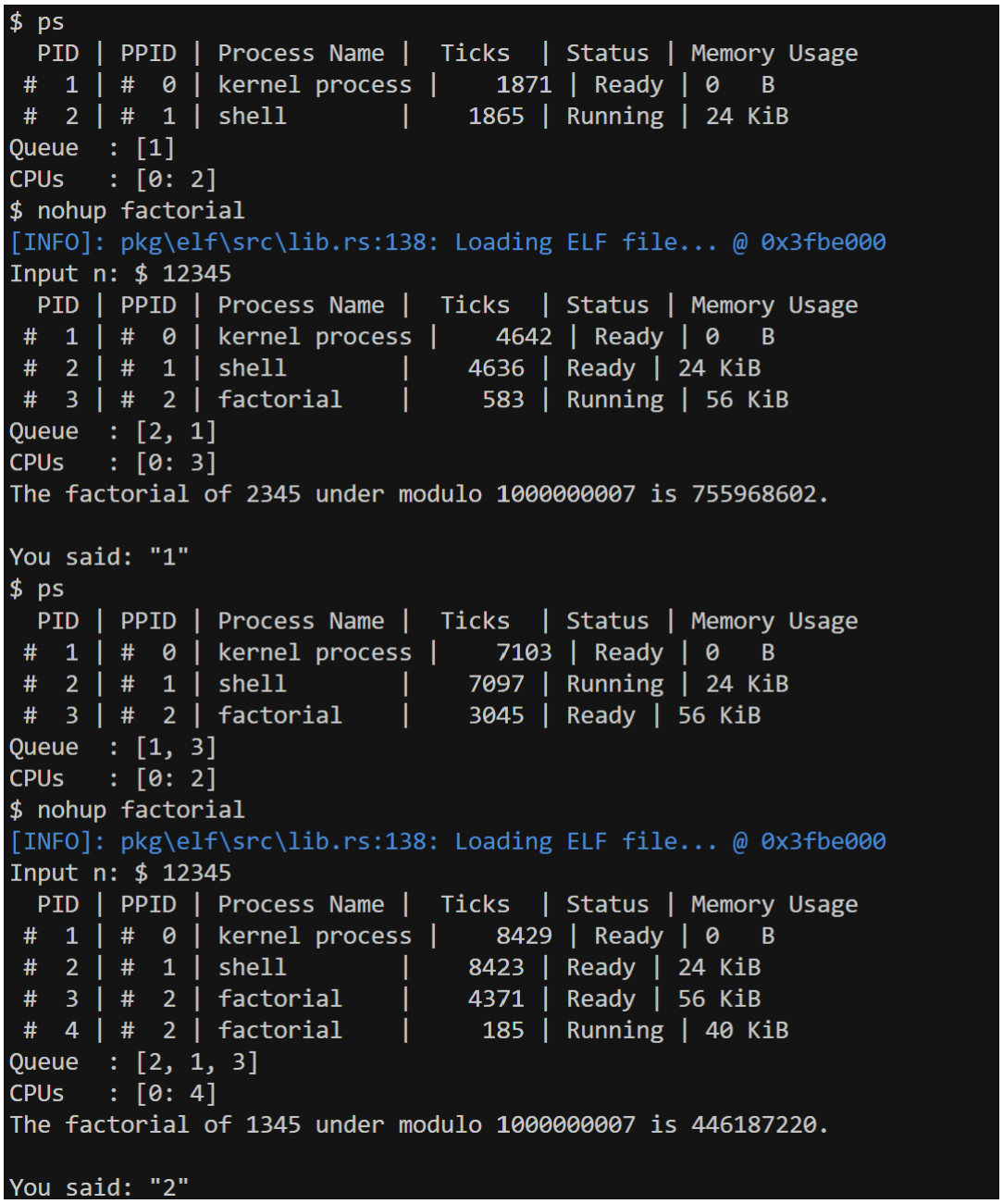


图 3　使用 nohup 创建进程

```
$ kill 3

$ ps
  PID | PPID | Process Name |  Ticks   | Status | Memory Usage
 #  1 | #  0 | kernel process |   10270 | Ready | 0   B
 #  2 | #  1 | shell        |   10264 | Running | 24 KiB
 #  4 | #  2 | factorial    |    2027 | Ready | 40 KiB
Queue  : [1, 4]
CPUs   : [0: 2]
$ kill 4

$ ps
  PID | PPID | Process Name |  Ticks   | Status | Memory Usage
 #  1 | #  0 | kernel process |   11295 | Ready | 0   B
 #  2 | #  1 | shell        |   11289 | Running | 24 KiB
Queue  : [1]
CPUs   : [0: 2]
$
```

图 4　使用 kill 结束进程

2. 堆内存分配测试程序

在本实验中，`sys_allocate` 系统调用实现了 C 语言中 `malloc` 函数提供的堆内存分配功能；`sys_deallocate` 系统调用实现了 C 语言中 `free` 函数提供的释放堆内存功能。下面是用于测试的堆内存分配测试程序。

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

use lib::*;
use core::alloc::Layout;
extern crate lib;
fn main() -> isize{
    let layout = Layout::new::<[usize;4096]>();
    println!("{:#?}", layout);
    let ptr = lib::sys_allocate(&layout);
    println!("{:0x?}", ptr);
    for idx in 0..5{
        unsafe {
            ptr.add(idx).write(idx as u8);
        }
    }
    for idx in 0..5{
        unsafe {
            println!("{}", ptr.add(idx).read());
        }
    }
    lib::sys_deallocate(ptr, &layout);
    0
```

```
}

entry!(main);
```

下面图 5 展示了上面这个程序的运行结果。我们可以看到，按照要求分配了从 `ptr` 开始的 4096 个 usize(u64) 类型的堆空间。我们可以借助指针 `ptr` 来访问分配的堆内存空间，向其写入数据，并读取出来。

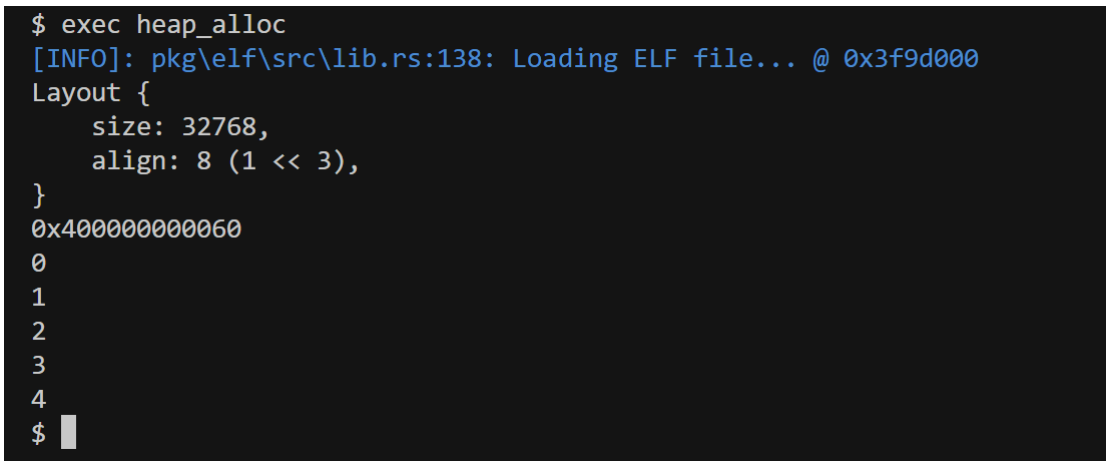


图 5　堆内存分配的测试结果

五. 思考题

1. 是否可以在内核线程中使用系统调用？并借此来实现同样的进程退出能力？分析并尝试回答。

可以在内核线程中使用系统调用，但是在内核线程中，本来就有内核的访问能力，所以使用系统调用会浪费一些性能。可以利用系统调用来实现相同的进程退出能力。

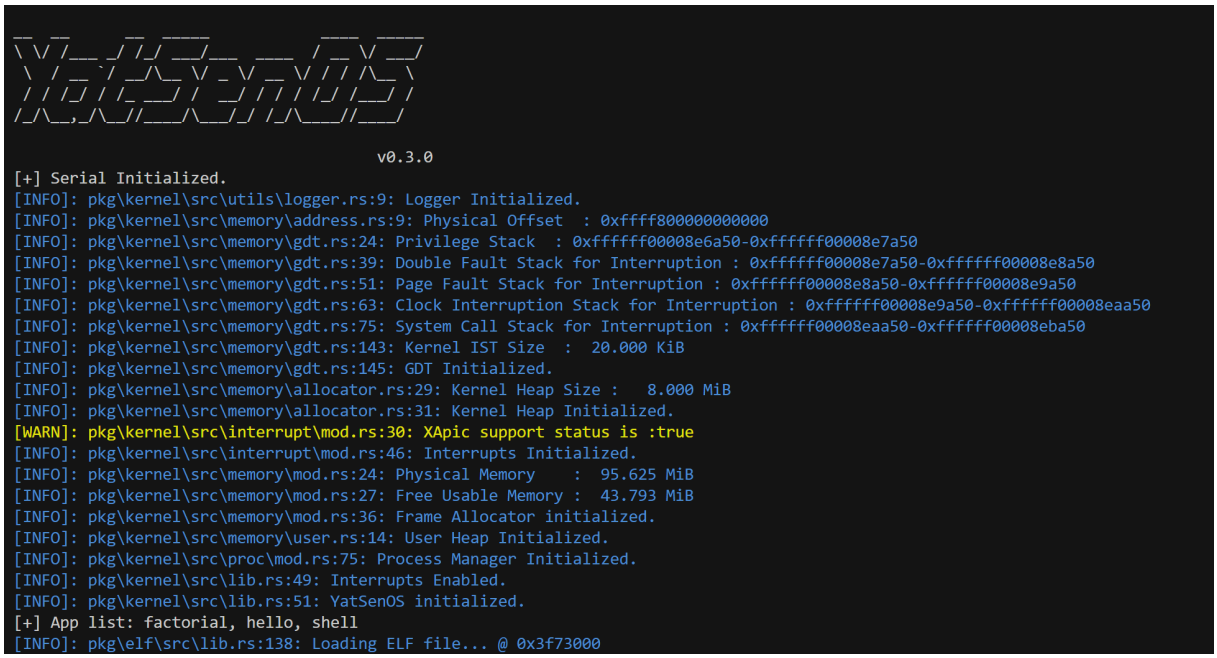
2. 为什么需要克隆内核页表？在系统调用的内核态下使用的是哪一张页表？用户态程序尝试访问内核空间会被正确拦截吗？尝试验证你的实现是否正确。

出于安全性以及方便性的考虑，需要克隆内核页表。一方面，需要防止用户进程直接访问内核空间的内容，另外一方面，用户进程持有自己的页表可以使得每个进程都有着相同的栈虚拟地址空间，而无需考虑与其它进程的栈空间的冲突。

系统调用的内核态下使用着内核页表。

用户态程序访问内核空间会被正确拦截。下面是一个测试。

首先从下图中可以看到，我们注意到特权级栈位于 `0xffffff00008e6a50`



接着利用这个下面这个程序来访问特权级栈，从而触发安全保护。

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

use lib::*;

extern crate lib;

fn main() -> isize {

    let pid = sys_get_pid();
    let ptr = 0xffffff00008e6a50 as *mut u8;
    unsafe {
        *ptr = 1;
    }
    sys_wait_pid(pid as u16);
    pid as isize
}

entry!(main);
```

下面是触发的安全保护。

```
[WARN]: pkg\kernel\src\interrupt\exceptions.rs:81: Page fault was caused by Process #3

[WARN]: pkg\kernel\src\interrupt\exceptions.rs:74: EXCEPTION: PAGE FAULT, ERROR_CODE: PageFaultErrorCode(PROTECTION_VIOLATION | CAUSED_BY_WRITE |
 USER_MODE)

Trying to access: 0xffffff00008e6a50
InterruptStackFrame {
    instruction_pointer: VirtAddr(
        0x11110000105c,
    ),
    code_segment: SegmentSelector {
        index: 5,
        rpl: Ring3,
    },
    cpu_flags: RFlags(
        IOPL_HIGH | IOPL_LOW | INTERRUPT_FLAG | PARITY_FLAG | 0x2,
    ),
    stack_pointer: VirtAddr(
        0x3fffffffff80,
    ),
    stack_segment: SegmentSelector {
        index: 6,
        rpl: Ring3,
    },
}
```

3. 为什么在使用 `still_alive` 函数判断进程是否存活时，需要关闭中断？在不关闭中断的情况下，会有什么问题？

在不关闭中断的情况下，执行 `still_alive` 函数的时候，可能会出现竞争条件，多个处理器核心同时读取 `still_alive` 的返回值。执行过程中来一个中断，新的中断中修改了储存进程的数据结构，有可能导致 `still_alive` 调用的返回值不一致，影响其功能正确性。

4. 对于如下程序，使用 gcc 直接编译：

```c
#include <stdio.h>

int main() {
    printf("Hello, World!\n");
    return 0;
}
```

从本次实验及先前实验的所学内容出发，结合进程的创建、链接、执行、退出的生命周期，参考系统调用的调用过程（可以仅以 Linux 为例），解释程序的运行。

假设在 Linux 执行此程序。默认情况下，gcc 使用动态链接，所以生成的二进制可执行文件(a.out)是动态链接的。该程序的启动流程大致如下:

- 首先，从 `init` 进程 fork 出一个子进程 A，初始化 A 的进程控制块的相关内容，例如按照 ELF 格式将文件映射到内存中，设置好程序计数器的值，初始化页表。
- 接着，由于程序是动态链接的，将交由动态链接的 ld 来去相应位置加载动态链接库进内存。
- 在动态链接库加载完成后，进程等待操作系统的调度来获取执行的时间片。
- 当需要进行系统调用时，会进行用户态与内核态的切换，以提升权限的内核态来执行指令，访问内核页表。

- 当执行结束的时候，调用 `exit` 系统调用，通知操作系统当前进程已经执行结束，回收其使用和持有的资源。

5. `x86_64::instructions::hlt` 做了什么？为什么这样使用？为什么不可以在用户态中的 `wait_pid` 实现中使用？

它将处理器暂停到下一次中断到来，中止了当前处理器正在执行的进程。这样使用可以避免处理器中的忙等待空转，降低能耗，自然而然地等待时钟中断来切换进程。用户态中使用 `x86_64::instructions::hlt` 会触发异常，在下面的 代码 1 中显示了详细的报错信息。归根结底，`hlt` 指令是一个特权级指令(PrivilegeLevel 0)，无法在用户态(PrivilegeLevel 3) 中使用。

```
[ERROR]: pkg\kernel\src\utils\macros.rs:76: ERROR: panic!

PanicInfo {
    payload: Any { .. },
    message: Some(
        EXCEPTION: GENERAL PROTECTION FAULT, ERROR_CODE: 0x0000000000000000

        InterruptStackFrame {
            instruction_pointer: VirtAddr(
                0x111100001070,
            ),
            code_segment: SegmentSelector {
                index: 5,
                rpl: Ring3,
            },
            cpu_flags: RFlags(
                IOPL_HIGH | IOPL_LOW | INTERRUPT_FLAG | SIGN_FLAG | PARITY_FLAG | 0x2,
            ),
            stack_pointer: VirtAddr(
                0x3fffffffff80,
            ),
            stack_segment: SegmentSelector {
                index: 6,
                rpl: Ring3,
            },
        },
    ),
    location: Location {
        file: "pkg\\kernel\\src\\interrupt\\exceptions.rs",
        line: 88,
        col: 5,
    },
    can_unwind: true,
    force_no_backtrace: false,
}
```

代码 1　General Protection Fault 异常

6. 有同学在某个回南天迷蒙的深夜遇到了奇怪的问题：

只有当进行用户输入（触发了串口输入中断）的时候，会触发奇怪的 Page Fault，然而进程切换、内存分配甚至 `fork` 等系统调用都很正常。经过近三个小时的排查，发现他将 TSS 中的 `privilege_stack_table` 相关设置注释掉了。请查阅资料，了解特权级栈的作用，实验说明这一系列中断的触发过程，尝试解释这个现象。

- **可以使用 `intdbg` 参数，或 `ysos.py -i` 进行数据捕获。**
- **留意 `0x0e` 缺页异常和缺页之前的中断的信息。**
- **注意到一个不应当存在的地址……？**

或许你可以重新复习一下 Lab 2 的相关内容：double-fault-exceptions

根据“https://os.phil-opp.com/double-fault-exceptions/#the-ist-and-tss”中关于特权级栈的说明，我们了解到，当 CPU 在用户态模式下触发了一个异常时，CPU 通常会进行特权提升。相应地，CPU 也会根据 TSS 中关于特权级栈的信息加载相应的特权级栈。

```
$ [WARN]: pkg\kernel\src\interrupt\exceptions.rs:74: EXCEPTION: PAGE FAULT, ERROR_CODE: PageFaultErrorCode(CAUSED_BY_WRITE)

Trying to access: 0xfffffffffffffff8
InterruptStackFrame {
    instruction_pointer: VirtAddr(
        0x11110000216e,
    ),
    code_segment: SegmentSelector {
        index: 5,
        rpl: Ring3,
    },
    cpu_flags: RFlags(
        IOPL_HIGH | IOPL_LOW | INTERRUPT_FLAG | ZERO_FLAG | PARITY_FLAG | 0x2,
    ),
    stack_pointer: VirtAddr(
        0x3fffffffda0,
    ),
    stack_segment: SegmentSelector {
        index: 6,
        rpl: Ring3,
    },
}
[WARN]: pkg\kernel\src\interrupt\exceptions.rs:81: Page fault was caused by Process #2
```

图 6　注释掉特权级栈导致串口输入中断触发缺页异常

然而，由于现在没有在 TSS 中手动注册特权级栈的信息，内核会尝试访问 `0xfffffffffffffff8` 这个不应当存在的地址。这个看起来比较奇怪的地址与 `x86_64` crate 中初始化 TSS 的特权级栈部分的内容有关。创建 `TaskStateSegment` 的默认构造函数如下 代码 2 所示。可以看到特权级栈有默认值 `0`。默认的栈顶位置是最高地址(0x0)往下移动一个字长(在 x86_64 中，就是 8 个字节)，也就正好是 `0xfffffffffffffff8` (考虑到补码运算，这个值是所谓的 “−8”)。

```
pub const fn new() -> TaskStateSegment {
        TaskStateSegment {
            privilege_stack_table: [VirtAddr::zero(); 3],
            interrupt_stack_table: [VirtAddr::zero(); 7],
            iomap_base: size_of::<TaskStateSegment>() as u16,
            reserved_1: 0,
            reserved_2: 0,
            reserved_3: 0,
            reserved_4: 0,
        }
}
```

代码 2　TaskStateSegment 的默认构造函数

六. 加分项

1. 尝试在 `ProcessData` 中记录代码段的占用情况，并统计当前进程所占用的页面数量，并在打印进程信息时，将进程的内存占用打印出来。

在 `pkg\kernel\src\proc\manager.rs` 中，修改 `print_process_list` 函数，增加一列用以显示 `Memory Usage` 。

```
pub fn print_process_list(&self) {
    let mut output = String::from("  PID | PPID | Process Name |  Ticks  | Status | Memory Usage\n");

    for (_, p) in self.processes.read().iter() {
        if p.read().status() != ProgramStatus::Dead {
            output += format!("{}\n", p).as_str();
        }
    }

    // TODO: print memory usage of kernel heap

    output += format!("Queue  : {:?}\n", self.ready_queue.lock()).as_str();

    output += &processor::print_processors();

    print!("{}", output);
}
```

在 `pkg\kernel\src\proc\process.rs` 中，修改为 `Process` 实现的 `Display` trait。为了方便显示一个人类可读的内存大小，需要利用 `memory::humanized_size` 函数来格式化字节数。

```
impl core::fmt::Display for Process {
    fn fmt(&self, f: &mut core::fmt::Formatter) -> core::fmt::Result {
        let inner = self.inner.read();
        let (memory_usage, memory_unit) = inner
            .proc_data
            .as_ref()
            .map(|d| {
                let total_pages = d.stack_pages + d.code_pages;
                let total_size = total_pages as u64 * PAGE_SIZE;
                memory::humanized_size(total_size)
            })
            .unwrap_or((0f64, "B"));
        write!(
            f,
            " #{:-3} | #{:-3} | {:12} | {:7} | {:?} | {:<} {}",
            self.pid.0,
            inner.parent().map(|p| p.pid.0).unwrap_or(0),
```

```
            inner.name,
            inner.ticks_passed,
            inner.status,
            memory_usage,
            memory_unit,
        )?;
        Ok(())
    }
}
```

```
$ ps
  PID | PPID | Process Name |  Ticks   | Status | Memory Usage
 #  1 | #  0 | kernel process |   12711 | Ready | 0   B
 #  2 | #  1 | shell        |   12701 | Running | 24 KiB
Queue  : [1]
CPUs   : [0: 2]
```

图 7　使用 ps 命令打印带有内存信息的进程列表

2. 尝试在 `kernel/src/memory/frames.rs` 中实现帧分配器的回收功能 `FrameDeallocator`，作为一个最小化的实现，你可以在 `Allocator` 使用一个 `Vec` 存储被释放的页面，并在分配时从中取出。

在 `BootInfoFrameAllocator` 添加 `released_pages` 的字段。并相应地修改初始化函数 `init` 。

```
pub struct BootInfoFrameAllocator {
    // ...
    released_pages: Vec<PhysFrame>,
}
/// Create a FrameAllocator from the passed memory map.
///
/// This function is unsafe because the caller must guarantee that the passed
/// memory map is valid. The main requirement is that all frames that are marked
/// as `USABLE` in it are really unused.
pub unsafe fn init(memory_map: &MemoryMap, size: usize) -> Self {
    BootInfoFrameAllocator {
        size,
        frames: create_frame_iter(memory_map),
        used: 0,
        released_pages: Vec::new(),
    }
}
```

修改 `allocate_frame` 函数以便重用释放出去的内存。

```rust
unsafe impl FrameAllocator<Size4KiB> for BootInfoFrameAllocator {
    fn allocate_frame(&mut self) -> Option<PhysFrame> {
        if !self.released_pages.is_empty() {
            self.released_pages.pop()
        } else {
            self.used += 1;
            self.frames.next()
        }
    }
}
```

添加 `deallocate_frame` 函数的一个简单实现如下。

```rust
impl FrameDeallocator<Size4KiB> for BootInfoFrameAllocator {
    unsafe fn deallocate_frame(&mut self, _frame: PhysFrame) {
        // TODO: deallocate frame (not for lab 2)
        self.released_pages.push(_frame);
    }
}
```

3. 基于帧回收器的实现，在 `elf` 中实现 `unmap_range` 函数，从页表中取消映射一段连续的页面，并使用帧回收器进行回收。之后，在合适的地方，结合 `ProcessData` 中存储的页面信息，利用这个函数实现进程栈的回收。其他进程资源（如页表、代码段、数据段等）的回收将会在后续实验中实现，目前暂时不需要考虑。

在 `pkg\elf\src\lib.rs` 中添加 `unmap_range` 函数如下。

```rust
/// Unmap a range of memory
///
/// unmap frames and deallocate if needed
pub fn unmap_range(
    addr: u64,
    count: u64,
    page_table: &mut impl Mapper<Size4KiB>,
    frame_deallocator: &mut impl FrameDeallocator<Size4KiB>,
    deallocate_flag: bool,
) -> Result<(), UnmapError> {
    let range_start = Page::containing_address(VirtAddr::new(addr));
    let range_end = range_start + count;
    trace!("Unmapping a range of memory");
```

```
    trace!(
        "Page Range: {:?}({})",
        Page::range(range_start, range_end),
        count
    );
    trace!(
        "Map hint: {:#x} -> {:#x}",
        addr,
        page_table
            .translate_page(range_start)
            .unwrap()
            .start_address()
    );

    for page in Page::range(range_start, range_end) {
        let (frame, changed_page) = page_table.unmap(page)?;
        unsafe {
            if deallocate_flag {
                frame_deallocator.deallocate_frame(frame);
            }
        }
        changed_page.flush();
    }

    Ok(())
}
```

4. 尝试利用 `UefiRuntime` 和 `chrono` crate，获取当前时间，并将其暴露给用户态，以实现 sleep 函数。

`UefiRuntime` 的实现，它可能需要使用锁进行保护：

```
pub struct UefiRuntime {
    runtime_service: &'static RuntimeServices,
}

impl UefiRuntime {
    pub unsafe fn new(boot_info: &'static BootInfo) -> Self {
        Self {
            runtime_service: boot_info.system_table.runtime_services(),
        }
    }

    pub fn get_time(&self) -> Time {
        self.runtime_service.get_time().unwrap()
    }
}
```

这里提供一个可能的 `sleep` 函数实现：

```
pub fn sleep(millisecs: i64) {
    let start = sys_time();
    let dur = Duration::milliseconds(millisecs);
    let mut current = start;
    while current - start < dur {
        current = sys_time();
    }
}
```

当前实现是纯用户态、采用轮询的，这种实现是很低效的。在现代操作系统中，进程会被挂起，并等待对应事件触发后重新被调度。虽然不是最好，但是在目前的需求下，这已经足够了。在实现后，写一个或更改现有用户程序，验证你的实现是否正确，尝试输出当前时间并使用 `sleep` 函数进行等待。

在 `pkg\kernel\src\utils\uefi_runtime.rs` 中，创建 `UefiRuntime` 类型，使用 `once_mutex` 宏来声明一个静态变量 `UEFI_RUNTIME` 。

```
use boot::{BootInfo, RuntimeServices, Time};

once_mutex!(pub UEFI_RUNTIME:UefiRuntime);

pub fn init(boot_info: &'static BootInfo) {
    unsafe {
        init_UEFI_RUNTIME(UefiRuntime::new(boot_info));
    }
}

pub struct UefiRuntime {
    runtime_service: &'static RuntimeServices,
}

impl UefiRuntime {
    pub unsafe fn new(boot_info: &'static BootInfo) -> Self {
        Self {
            runtime_service: boot_info.system_table.runtime_services(),
        }
    }

    pub fn get_time(&self) -> Time {
        self.runtime_service.get_time().unwrap()
    }
}
```

在 `pkg\kernel\src\interrupt\syscall\service.rs` 中，添加 `sys_time` 中断处理函数。该函数的返回值是以毫秒为单位的时间戳。

```
pub fn sys_time() -> usize {
    // get current time
```

```
    let time = crate::utils::uefi_runtime::UEFI_RUNTIME
        .get()
        .unwrap()
        .lock()
        .get_time();
    let datetime =
       chrono::NaiveDate::from_ymd_opt(time.year() as i32, time.month() as u32, time.day()
as u32)
            .unwrap()
            .and_hms_nano_opt(
                time.hour() as u32,
                time.minute() as u32,
                time.second() as u32,
                time.nanosecond() as u32,
            )
            .unwrap();
    datetime.and_utc().timestamp_millis() as usize
}
```

在 `pkg\lib\src\syscall.rs` 添加用户态库的 `sys_time` 以及 `sleep` 函数。

```
#[inline(always)]
pub fn sys_time() -> usize {
    syscall!(Syscall::Time) as usize
}
#[inline(always)]
pub fn sleep(millisecs: i64) {
    let start = sys_time();
    let dur = Duration::from_millis(millisecs as u64);
    let mut current = start;
    while current - start < dur.as_millis() as usize{
        current = sys_time();
    }
}
```