# 0. 前言

## 实验概述

进程第一部分。运行实验代码，了解新代码架构、Task与任务状态切换、进程PCB。

## 实验内容

1. 拉取实验所需环境与代码
2. 了解实验代码的新架构
3. 运行使用C语言编写的Hello World用户态程序
4. 了解OSTD中的Task，以及任务状态切换
5. 了解进程PCB
6. 了解Init process

# 1. 实验环境及代码

拉取实验所需环境（环境与lab4相同）以及代码：

```
podman pull glcr.cra.ac.cn/operating-systems/asterinas_labs/images/lab4:0.1.0
mkdir os-lab
podman run -it -v ./os-lab:/root/os-lab glcr.cra.ac.cn/operating-systems/asterinas_labs/images/lab4:0.1.0

git clone -b lab5 https://github.com/sdww0/sustech-os-lab.git
cd sustech-os-lab
```

运行实验代码:

```
cargo osdk run --scheme riscv --target-arch=riscv64
```

# 2. Hello World in C

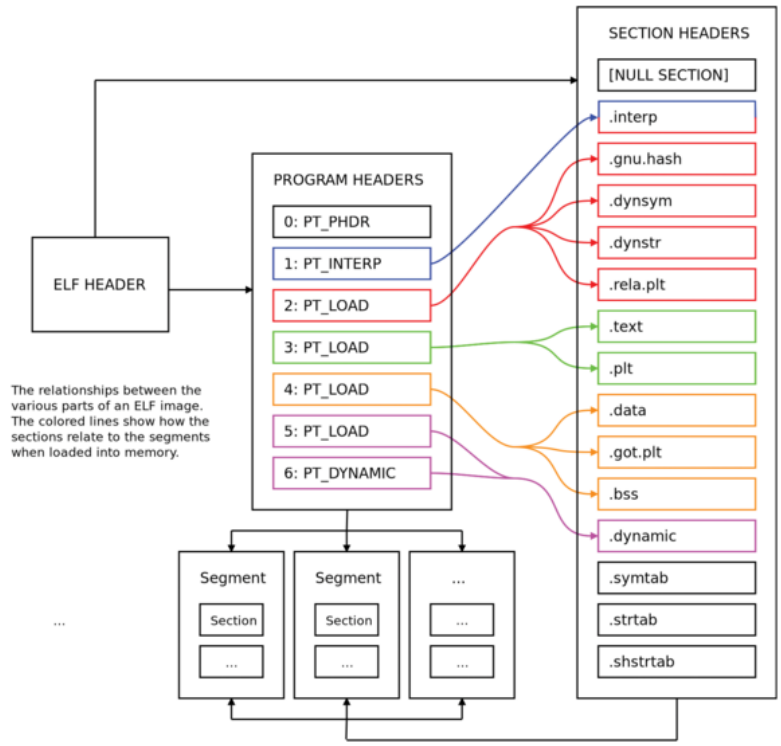
相比lab4需要编写汇编，lab5及以后的用户态应用程序代码可以使用GCC编译C文件来进行开发，这些C文件在user目录下。实验课代码已经编写好了自动编译的脚本`build.rs`，该脚本会进行三个步骤：（1）遍历user目录，获取C文件名；（2）执行编译，将编译好的文件放到`./target/user_prog/`目录下；（3）根据编译文件，生成`./src/fs/progs.rs`文件，该文件会通过`include_bytes_aligned`将编译文件直接链接到操作系统内。如要运行其他用户态程序，只需在`src/lib.rs`下将`INIT_PROCESS_NAME`更改为程序名即可。

与之前缺少标准库编译的汇编语言不同，使用标准库编译的C语言程序在进入到main之前会进行很多初始化操作，通过指定日志等级为debug可以看到这些初始化操作需要操作系统提供哪些服务：

```
cargo osdk run --scheme riscv --target-arch=riscv64 --kcmd-args="ostd.log_level=debug"
```

结果中会输出类似`[PID:  1] Syscall:xxx, args:xxxx`的语句，可以发现从程序开始运行，到输出Hello World之前程序还会进行若干系统调用，这就是为什么我们在正式支持使用标准库编译的C语言之前需要介绍系统调用的知识。代码已经对不可缺少的系统调用进行了基础支持，大家不需要关注这些系统调用，它们只是为了帮助我们更好学习其他内容的工具。

除此之外，这节实验课代码中正式加入了解析ELF文件的代码，之前映射固定地址的代码虽然简单，但不能应付稍微复杂一点的ELF文件，因此加入了加载ELF文件的代码。

> **ELF文件**
>
> ELF（Executable and Linkable Format)文件是由Section Header Table描述的一系列Section集合。一个程序从代码文件到可执行文件会生成若干个section，这些section的定义各不相同，比如.text是存放代码，.bss存放的是未初始化的全局变量，.rodata是只读数据。在ELF文件中就会通过一个结构来告诉加载器这些section在文件中的位置，以及该文件的一些信息：
>
> 
> 
>
> 图片来源于：[Linux magazine]

# 3. Process and Task

本次实验课代码中对进程进行了三层分级，它们的名称和功能分别是：

1. Task in OSTD：提供内核任务抽象，封装了内核任务所需要的必须资源如内核栈，CPU寄存器状态，以及一些调度信息。除此之外，它还提供了不同任务之间切换的接口。
2. Thread in OS：提供了线程抽象，使用到Task作为运行的工具，与进程的关系是线程：进程 = 多：一。
3. Process in OS：提供了进程抽象，里面会存储很多的信息以及辅助进程实现的域。

本章将会集中在OSTD的Task介绍和OS的Process介绍

## 3.1 Task

以`src/thread/mod.rs`中使用的Task作为入口点，我们可以追踪到到OSTD内的Task结构体：

```
1  /// A task that executes a function to the end.
```

```rust
///
/// Each task is associated with per-task data and an optional user space.
/// If having a user space, the task can switch to the user space to
/// execute user code. Multiple tasks can share a single user space.
pub struct Task {
    func: Box<dyn Fn() + Send + Sync>,
    data: Box<dyn Any + Send + Sync>,
    user_space: Option<Arc<UserSpace>>,
    ctx: UnsafeCell<TaskContext>,
    #[allow(dead_code)]
    kstack: KernelStack,

    schedule_info: TaskScheduleInfo,
}
```

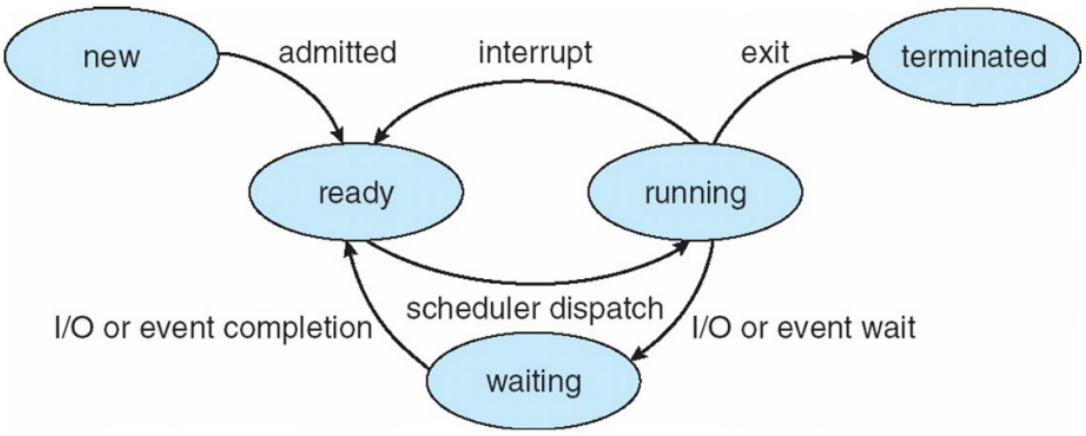
这些域的解释如下：

1. `func`：指定了当前Task在执行的入口函数。
2. `data`：存储了用户指定的数据，我们可以通过使用这个域与downcast来存放与获取自定义数据。
3. `user_space`：指定当前线程是内核线程还是用户线程，如果这个域为Some，则代表当前线程可以通过UserSpace进入到用户态空间，执行代码。
4. `ctx`：存放了当前线程的CPU寄存器信息，在进行线程切换时会使用到这里面的寄存器以进行上下文切换。
5. `kstack`：当前线程的内核栈，持有该域可以保证ctx存放的栈指针始终是有效且被当前线程拥有的。
6. `schedule_info`：一些调度信息。

除此之外，Task还提供了线程的基础API，其中有两个比较重要的API：`Task::run`和`Task::yield_now`分别代指运行该线程和当前线程放弃时间片，需要注意的是OSTD中的Task与调度器进行了绑定，`Task::run`实际上并不会立即运行当前任务。

`Task::run`会将当前任务加入到调度器中，并让调度器决定是否获取并执行下一个任务。当调度器需要执行下一个任务，则会调用`Task::yield_now`最终调用OSTD内部的API进行上下文切换。OSTD的调度器功能允许我们注册自定义的调度器，同时提供了一个默认的FIFO调度器，该调度器会在需要调度的时候，把当前的Task放在队尾，从队列中取出第一个可以运行的Task，切换到它运行。

同时，大家或许注意到了，Task提供的公开API中缺少一些线程基础的功能，包括：`switch_to`、`exit`、`change_status`等，这是因为OSTD维护了一套简易的线程状态切换。在理论课中学习到的进程状态切换表为：



那么这一套流程对外我们能看到什么呢？似乎只有`run`和`yield_now`来进行ready和running状态之间的切换。实际上，中间的各类状态转移已经在OSTD内部进行了，或者说要求我们通过OSTD提供的另外一些内容主动触发状态转移。

对于new, ready, running, terminated四种状态切换，OSTD内部进行了维护，切换时间点分别为：

1. `new -> ready`及`ready -> running`。当新建后调用run接口，就已经进入了ready状态，将任务添加到了调度器内部。我们执行`Task::run`时实际上并不会立即运行当前任务，而是将当前任务加入到调度器中，并让调度器获取并执行下一个任务，在这个流程的最后会调用OSTD内部的API进行上下文切换。
2. `running -> ready`。OSTD内部提供了抢占式调度的接口，未注册自定义调度器启用该功能时不会进行抢占式调度，但仍然存在这一条路径。
3. `running -> terminated`。Task在初始化后并不会直接进入用户指定的入口函数，而是会进入Task定义的一个函数，里面会间接调用该入口函数。在Task定义的函数的最后会调用`exit`函数，以完成资源回收

对于`running -> waiting`，`waiting -> ready`的状态切换，OSTD为我们提供了`WaitQueue`与`Waiter`，需要我们主动进行等待和唤醒的操作，具体的使用方法会在介绍`wait`系统调用时提及。

> **上下文切换的汇编**
>
> 上下文切换中需要将当前线程的CPU寄存器信息推入栈中，以及将下一个线程的CPU寄存器从栈中恢复，具体的汇编代码在`ostd/src/arch/riscv/task/switch.S`中。观察一下可以发现，在保存和恢复寄存器的过程中并没有涉及到所有的通用寄存器，这是源于编译器对于函数的处理。我们知道寄存器可以分为**调用者保存（caller-saved）寄存器**和**被调用者保存（callee-saved）寄存器**。 因为我们在一个函数中进行线程切换，所以编译器会自动生成保存和恢复调用者保存寄存器的代码。由此在进程切换过程中我们只需要保存被调用者保存寄存器即可。
>
> **调用者保存寄存器（caller saved registers）**
>
> 也叫**易失性寄存器**，在程序调用的过程中，这些寄存器中的值不需要被保存（即压入到栈中再从栈中取出），如果某一个程序需要保存这个寄存器的值，需要调用者自己压入栈；
>
> **被调用者保存寄存器（callee saved registers）**
>
> 也叫**非易失性寄存器**，在程序调用过程中，这些寄存器中的值需要被保存，不能被覆盖；当某个程序调用这些寄存器，被调用寄存器会先保存这些值然后再进行调用，且在调用结束后恢复被调用之前的值；

## 3.2 Thread

实验课代码中的TCB定义在`src/thread/mod.rs`中，其内容物较少：

```rust
pub struct Thread {
    tid: Tid,
    task: Once<Arc<Task>>,
    process: Weak<Process>,

    // Linux specific attributes.
    // https://man7.org/linux/man-pages/man2/set_tid_address.2.html
    set_child_tid: Mutex<Vaddr>,
    clear_child_tid: Mutex<Vaddr>,
}
```

其中最前面三个代表的是Thread的基础信息，包括线程ID，与底层任务的绑定，与上层应用的绑定。最后的两个是为了支持Linux ABI所包含的域，可以暂时忽略

## 3.3 Process

实验课代码中的PCB定义在`src/process/process.rs`中：

```
pub struct Process {
    // ======================== Basic info of process ==============================
    /// The id of this process.
    pid: Pid,
    /// Process state
    status: ProcessStatus,
    /// The name of this process, we use executable path for the user process.
    name: RwLock<String>,
    /// The threads of this process
    threads: Mutex<Vec<Arc<Thread>>>,

    // ======================== Memory-related fields ==============================
    /// The memory space of this process
    pub(super) memory_space: MemorySpace,
    /// The user space of this process contains CPU registers information.
    user_space: Option<Arc<UserSpace>>,
    /// The heap of the user process
    pub(crate) heap: UserHeap,

    // ======================== Process-tree fields ==================================
    /// Parent process.
    parent_process: Mutex<Weak<Process>>,
    /// Children process.
    children: Mutex<BTreeMap<Pid, Arc<Process>>>,
    /// The WaitQueue for a child process to become a zombie.
    wait_children_queue: WaitQueue,
    // TODO: more field of process, including fd table...
}
```

注释已经将这几个域进行了分类:

- 进程基础信息，包含pid，进程状态，进程名与绑定的线程。进程状态将会在之后的实验课进行讲解，绑定的线程没有什么实际的含义，只是展示进程和线程一对多的关系，实际使用中我们还是以一对一的绑定，方便进行开发。
- 内存相关信息，包含内存映射管理，用户态空间，用户堆。具体讲解将会在之后的实验课中介绍。
- 进程树相关信息，包含父子进程，以及一个等待子进程退出的队列。父子进程这两个域组成了课上所认知的进程树，对于等待子进程退出的队列同样会在之后的实验课进行讲解。

## 3.3 用户态进程的创建

`Process::new_user_process`会根据传入ELF文件的字节数组以及路径名新建一个用户态进程，里面会进行一系列操作：

1. 从PID分配器中获得一个PID
2. 根据`program`的字节数组，解析ELF文件并构建应用程序的页表映射以及CPU寄存器的初始值。对于CPU寄存器值，会将栈寄存器指定为分配好的用户栈最高地址，以及指令寄存器设为ELF文件指定的入口地址。
3. 设置name为路径名
4. 设置PCB中其它的域为默认值，并创建一个进程
5. 根据process新建一个线程，并加入到新建进程的线程队列
6. 设置当前进程为可运行状态

# 4. IDLE & Init process

有了进程抽象的支持，我们可以开始新建内核和用户进程了，内核在启动后会创建系统的第一个内核进程，又称为IDLE进程，该进程只会在内核执行，不会进入到用户态。

在IDLE进程中可以指定第一个需要运行的用户态进程（Init process），我们在这里指定了`hello_world`进程，那么内核便会根据这个名字，到文件系统中进行查找（当然，我们现在的文件系统只是一个简单的key-value存储），找到后便会开始运行该进程，直到初始进程退出变为Zombie之后再退出整个系统。

# 5. 上手练习

1. 尝试自己在user目录下创建一个C语言程序并编译内核，如果顺利会在控制台中看到添加进的程序名。如果没有看到，则需要清理一下编译缓存`cargo clean`再重新进行编译。
2. 试着使用标准库中的fork函数，观察系统出现了什么情况。

**更多的进程！**

1. 在`hello_world.c`中，使用`getppid`来获取并打印父进程pid，并在init thread中创建100个Hello world用户态进程并运行，观察pid的增长规律以及父进程的pid值。**注意需要使用`Process::set_parent_process`来设置父进程**
2. 基于步骤1，在init thread中等待100个Hello world用户态进程运行完后，再创建一个内核进程，并在其中创建100个Hello World用户态进程，观察pid的增长规律以及父进程的pid值。