Lab5 - Std C program & Multiprocess

0. 前言

实验概述

为操作系统支持标准库环境下的C语言程序，了解多进程系统与进程生命周期，并为系统添加Clone（Fork）系统调用支持

实验环境安装

在原有的sustech-os-lab中运行`./lab5-setup.sh`：

```
$ git pull && bash ./lab5-setup.sh
```

或重新克隆仓库并运行：

```
$ sudo apt install -y git
$ git clone https://github.com/sdww0/sustech-os-lab.git
$ cd sustech-os-lab && bash ./lab5-setup.sh
```

实验代码

本次代码新增：

1. **Makefile**，用于构建用户态程序，并运行系统，使用`make run`运行程序
2. prog目录，用于将要运行的文件链接到os的binary中
3. mm目录，构建运行用户态程序所需的内存空间，比如存储了用户栈，用户代码的信息
4. process/elf，elf文件解析，将elf文件的内容进行解析，并构建用户内存空间
5. process/heap，用户态程序堆空间
6. syscall目录下的若干文件，用于更精准的处理系统调用
7. error.rs，指示系统调用错误码
8. logger.rs，格式化日志输出，添加颜色

本次实验课代码没有加入自动退出QEMU的功能，请输入`Ctrl + A + X`来手动退出QEMU

1. Hello World in C

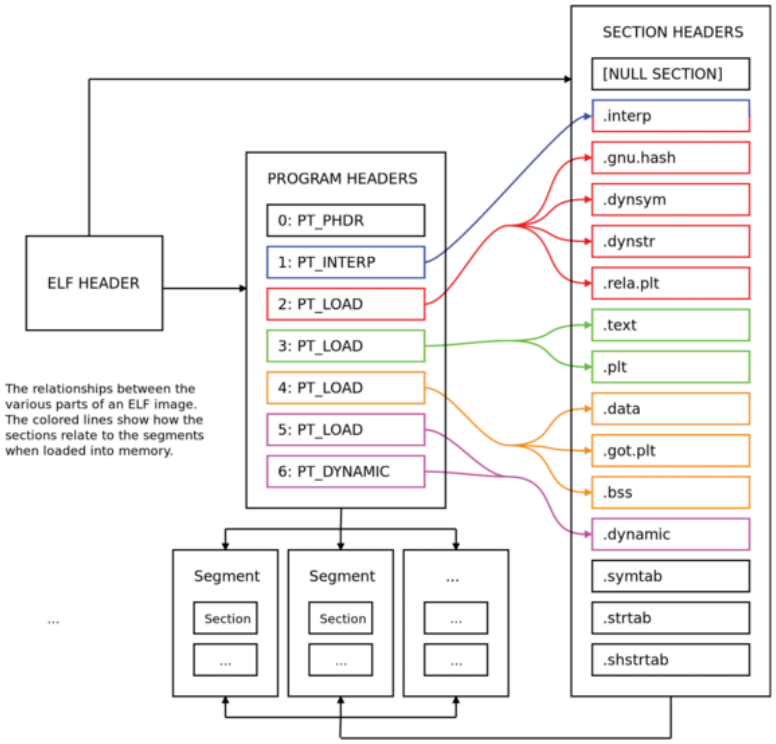
在之前的课程中我们成功运行了非标准库环境下，Hello World的汇编语言程序，但这对于我们之后的开发和测试不友好，我们需要直接编写汇编语言，且没法使用到标准库提供的丰富功能，为了解决这个问题，我们将针对标准库环境下的C语言程序，进行开发和支持。

1.1 用户空间：ELF文件解析与用户栈构建

为支持标准库环境下的C语言程序，我们需要进一步扩展之前的用户空间构建代码，回顾一下一个基础程序运行所需要的资源有：代码，静态数据，**堆和栈**。针对前两个，我们将会进行**ELF文件解析**以进行加载各个字段，针对后一个，我们会进行**用户栈构建**，但堆需要在系统调用中进行支持。

ELF文件解析

ELF（Executable and Linkable Format)文件是由Section Header Table描述的一系列Section集合。ELF文件会通过一个固定结构来告诉加载器，该程序的一些信息（比如程序架构，程序大小端，**程序入口地址**等），以及程序运行所需的各个字段，如.text，.rodata在文件和内存中的位置。加载器需要解析这些信息，并将文件中的指定内容复制到内存中，最后运行程序。下图是一个典型ELF文件的布局：



在之前的代码中我们会将指定地址作为用户态程序的入口地址，这很显然是个dummy实现，不能应付稍微复杂一点的用户态程序，因此我们需要加入针对用户态程序分布的解析，即ELF文件解析。

用户栈构建

ELF文件并不会告诉我们用户栈的地址，因此对于用户栈，我们需要寻找一个特定的高地址，分配一些内存后，将用户的栈指针设置为该地址的高位，在代码中我们的设置为：

```
pub const USER_STACK_SIZE: usize = 8192 * 1024; // 8MB
let stack_low = 0x40_0000_0000 - 10 * 4096 - USER_STACK_SIZE;
let stack_high = stack_low + USER_STACK_SIZE
user_cpu_state.set_stack_pointer(0x40_0000_0000 - 10 * 4096 - 32);
```

1.2 C语言标准库支持

标准库环境下的C语言程序会在进入main函数之前进行一系列的初始化操作。这些初始化操作可以通过运行系统来略窥一二，本次代码的Makefile添加了`LOG_LEVEL`的设置，我们可以通过`make run LOG_LEVEL=debug`来将系统的日志等级设置为`debug`，这样就能看到内核在输出Hello World之前还会接受若干的用户态系统调用请求，如：

```
[INFO ] [pid: 1] syscall num: 261, args: [0, 3, 0, 3fffff5f38, 56b98, 3]
```

根据Linux对于这些系统调用的定义（参考： RISC-V Linux Syscall Table 以及对应的manual），我们可以找出这些系统调用号所对应的功能，并加以支持，通过一步步试错，我们定位出了支持Hello World应用程序所需要的最小系统调用集合：

- `newuname`：获取当前系统的基础信息，比如系统版本，系统名称等，我们常用的`uname`指令就是会执行该系统调用，标准库会判断当前系统版本，如过低则会直接退出，因此我们需要fake一个值以让程序正常运行
- `mprotect`：当前无作用，但需要我们返回0 （0在系统调用中通常代表成功执行）
- `brk`：扩展用户态程序的**堆**大小

有了这三个基础的系统调用支持，我们便可以运行标准库下的C语言程序，使得它进入main函数输出Hello World。

> **RISC-V浮点**
>
> 在`src/process/mod.rs::create_user_task`中有一段代码，会在用户态程序出现IllegalInstruction时跳过该指令:
>
> ```
> let exception = user_context.take_exception().unwrap();
> if exception.cpu_exception() == Exception::IllegalInstruction {
>     // The illegal instructions can include the floating point instructions
>     // if the FPU is not enabled. Here we just skip it.
>     user_context
>         .set_instruction_pointer(user_context.instruction_pointer() + 2);
> } else {
>     early_println!(
>         "Process {} killed by exception: {:#x?}   at instruction {:#x}",
>         process.pid,
>         exception,
>         user_context.instruction_pointer()
>     );
>     exit_qemu(QemuExitCode::Success);
> }
> ```
>
> 这么做的原因是，OSTD目前没有针对浮点寄存器进行测试，使得用户态程序不能使用浮点相关的寄存器操作，如`fsd`指令，因此会触发Illegal Instruction异常，这里为了workaround直接跳过了这几条指令，但我们也可以通过在进入用户态时暂时启用浮点来解决该问题，比如在`ostd/src/arch/riscv/trap/trap.S`中，将135行的`trap_return`改为：
>
> ```
>     LOAD_SP t0, 32          # t0 = sstatus
>     LOAD_SP t1, 33          # t1 = sepc
>     csrw sepc, t1           # load sepc
>     li t1, 3 << 13
>     or t1, t1, t0           # sstatus.FS = Dirty (3)
>     csrw sstatus, t1
> ```
>
> 就会在进入用户态时开启浮点指令，此时不会触发到Illegal Instruction（注意需要clean后重新run），具体原因可见`sstatus`寄存器中的FS域

2. 多进程系统

在之前的代码中，我们已经实现了用户态程序加载，用户态-内核态切换，以及系统调用服务三个功能，为运行单个进程做了支持，但只有这些功能，我们进程的功能依旧十分受限，集中在缺少多进程的支持，这会让我们的操作系统不能同时运行多个用户程序

为了实现多进程系统，我们需要支持两大类功能：

1. 内核多线程，提供**内核线程的资源隔离和内核线程之间的切换**，在代码中为OSTD中的`Task`

2. 多进程管理，如进程的退出时的资源回收，进程的创建，以及父-子进程的管理，在代码中为`Process`

2.1 Task

我们自定义PCB中的Task会负责我们内核线程之间的切换和调度，提供基础的内核多线程支持，使得我们可以基于此进行进程的构建，以`Process::task`作为入口点，我们可以追踪到到OSTD内的Task结构体：

```rust
/// A task that executes a function to the end.
///
/// 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.
#[derive(Debug)]
pub struct Task {
    #[expect(clippy::type_complexity)]
    func: ForceSync<Cell<Option<Box<dyn FnOnce() + Send>>>>,

    data: Box<dyn Any + Send + Sync>,
    local_data: ForceSync<Box<dyn Any + Send>>,

    ctx: SyncUnsafeCell<TaskContext>,
    /// kernel stack, note that the top is SyscallFrame/TrapFrame
    kstack: KernelStack,

    /// If we have switched this task to a CPU.
    ///
    /// This is to enforce not context switching to an already running task.
    /// See [`processor::switch_to_task`] for more details.
    switched_to_cpu: AtomicBool,

    schedule_info: TaskScheduleInfo,
}
```

在Task中会存储一些线程特定的信息：

1. 用于指定线程入口函数：`func`
2. 用于存储用户指定的线程数据：`data` & `local_data`，其中`data`主要会用于存储`Arc<Process>`，以获取当前进程
3. 用于存储线程运行所需的信息（同样为线程特定，且是线程运行的基础）：`ctx`&`kstack`，存储了寄存器与栈信息，其中寄存器信息会在**上下文切换**时使用
4. 其他信息：`switch_to_cpu`&`schedule_info`，为OSTD其他模块使用的数据，我们不会使用到

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，以及当前线程放弃时间片以切换至下一个线程

> **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；

2.2 Process

本次实验课代码对Process进一步添加了一些域，这些域支撑起我们对于标准库C程序以及多进程的支持，PCB定义在`src/process/mod.rs`中：

```
pub struct Process {
    // ======================== Basic info of process ==========================
    /// The id of this process.
    pid: Pid,
    /// Process state
    status: ProcessStatus,
    /// The thread of this process
    task: Once<Arc<Task>>,

    // ======================== Memory management ===============================
    memory_space: MemorySpace,
    // Heap
    heap: UserHeap,

    // ======================== Process-tree fields
=================================
    /// Parent process.
    parent_process: Mutex<Weak<Process>>,
    /// Children process.
    children: Mutex<BTreeMap<Pid, Arc<Process>>>,
}
```

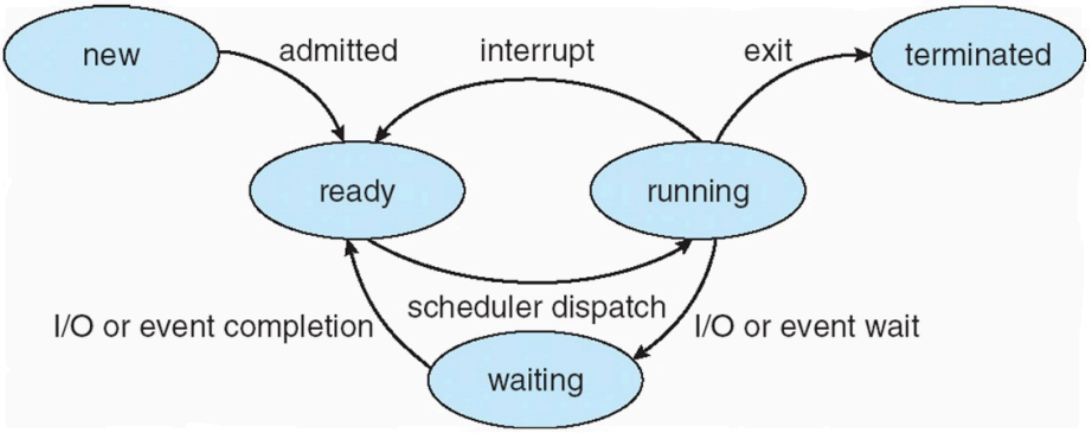
对比之前的PCB，我们进行了如下修改：

1. 基础信息中，将status扩展为一个结构体，封装了进程状态管理，并支持exit code
2. 内存管理部分将原有的`Arc<VmSpace>`修改为了`MemorySpace`和`UserHeap`，两者协同来完成对用户态应用程序的支持
3. 新增进程树的区域，用于管理父-子进程

> **Thread**
>
> OSTD中的`Task`并不能满足我们未来对于用户态多线程的支持，但因为现在我们的设计比较简单，所以暂时无需Thread的支持，有一些系统调用我们并未实现，如set_tid_address

3. 进程生命周期

内核对于进程生命周期进行了分离，有部分状态切换在`Task`中，有部分则在`Process`，回顾一个进程生命周期图：



首先我们来看Task能提供给我们什么状态切换，只包括了`run`和`yield_now`来进行ready和running状态之间的切换：

1. `Task::run`：完成了`ready -> running`的状态切换，运行线程
2. `Task::yield_now`：完成了`running -> ready`的状态切换，该函数被调用时，就会进行将当前线程重新push到运行队列中，并切换到下一个可运行的内核线程

但只有这些还不够，图里还有两类的状态没有涉及到，即`waiting`与`new & Terminated`，分别由OSTD中的`Waiter & Waker`，内核的`Process`来进行管理：

- `Waiter & Waker` / `WaitQueue`：为OSTD提供的一组用于 **等待-唤醒** 的结构，当内核线程需要等待某个事件发生时，就需要使用到这两个结构，只有当超时或者事件发生时等待才会结束，我们将会在之后使用它来进行父进程对子进程退出的等待
- `Process`：我们定义了`ProcessStatus`来进行`new -> ready`与`ready -> terminated`的状态切换，当进程初始化完毕时，便会将状态设置为`ready`（代码中为Runnable），当进程退出时，便会将状态设置为`terminated`（代码中为Zombie）

4. 上手练习

1. `SYS_PRLIMIT64`系统调用完善。目前我们的`SYS_PRLIMIT64`实现并不完善，只是返回了一个dummy的值，请你用Debug日志等级运行系统，查看用户态程序传入的参数，然后查看Linux的manual网页，以针对用户态程序的初始化流程，完善该系统调用的支持，代码中全局搜索`TODO-1`
2. `Clone`(Fork)系统调用支持。目前我们虽然声明有`Fork`系统调用支持，但里面并没有干任何事情，请你完善`sys_clone`函数，为内核添加Fork系统调用的支持，代码中全局搜索`TODO-2`。以下是提示：
    1. 目前不需要关注`sys_clone`传入的参数，只进行Fork即可
    2. `TODO-2`中已经包含了一些步骤提示
    3. `MemorySpace`的复制可以参考`MemorySpace::duplicate`，`UserContext`直接调用`.clone()`