2023秋《操作系统》课程实验报告

实验七

21301161 齐家昕

1. 实验步骤

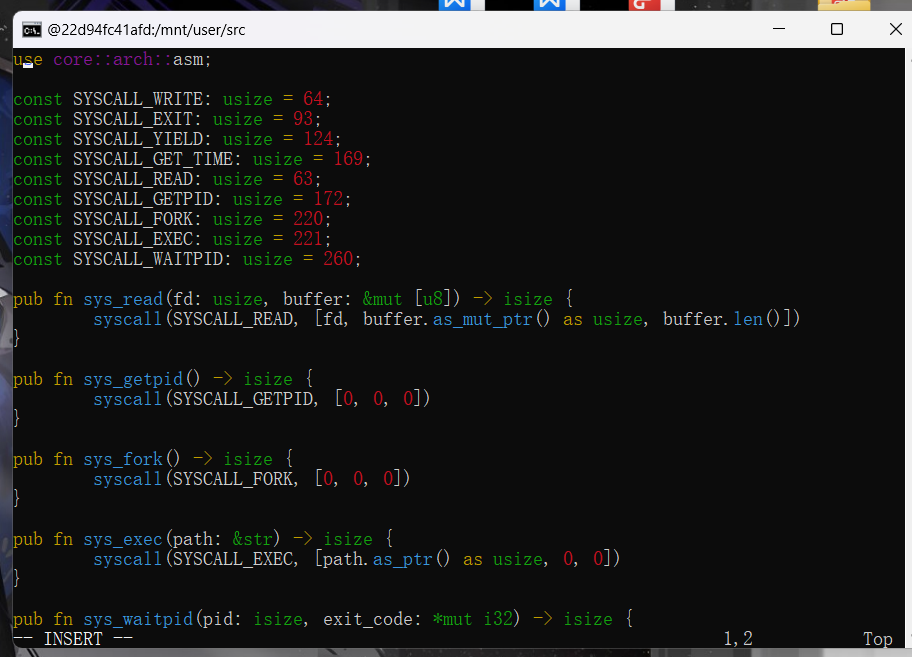
本实验的主要目的是实现进程及进程的管理。

1. 修改应用程序

（1）增加重要的系统调用

fork系统调用会创建一个新的进程；waitpid系统调用是的当前进程等待子进程结束，回收其资源并获得返回值；getpid系统调用获得当前进程的信息；exec系统调用将当前的进程地址空间清空并加载一个特定的可执行文件，然后返回用户态执行；read系统调用从文件中读取一段内容到缓冲区，主要目的是为了实现user shell。

首先，修改user/src/syscall.rs增加上述系统调用。



接着，在user/src/lib.rs封装系统调用为应用程序使用的形式。



（2）实现用户初始程序initproc

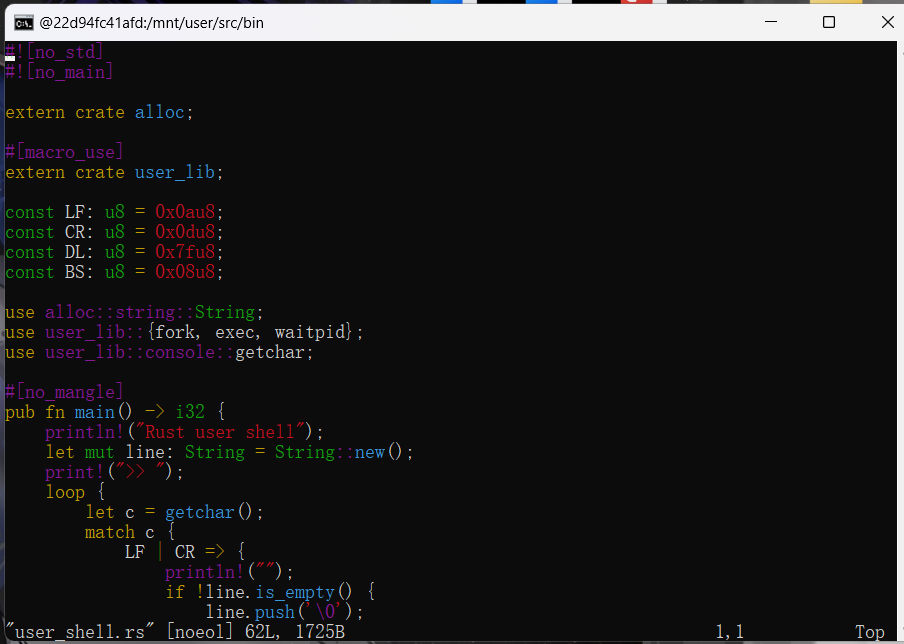


（3）实现shell程序

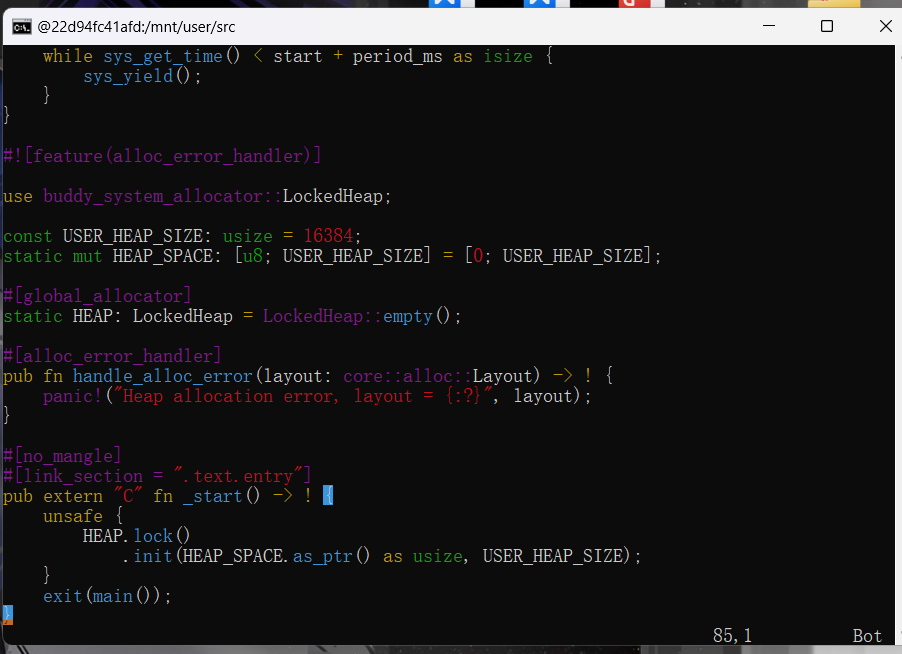
首先基于sys\_read系统调用封装能够从标准输入读取一个字符的函数getchar。



然后，实现user shell程序。

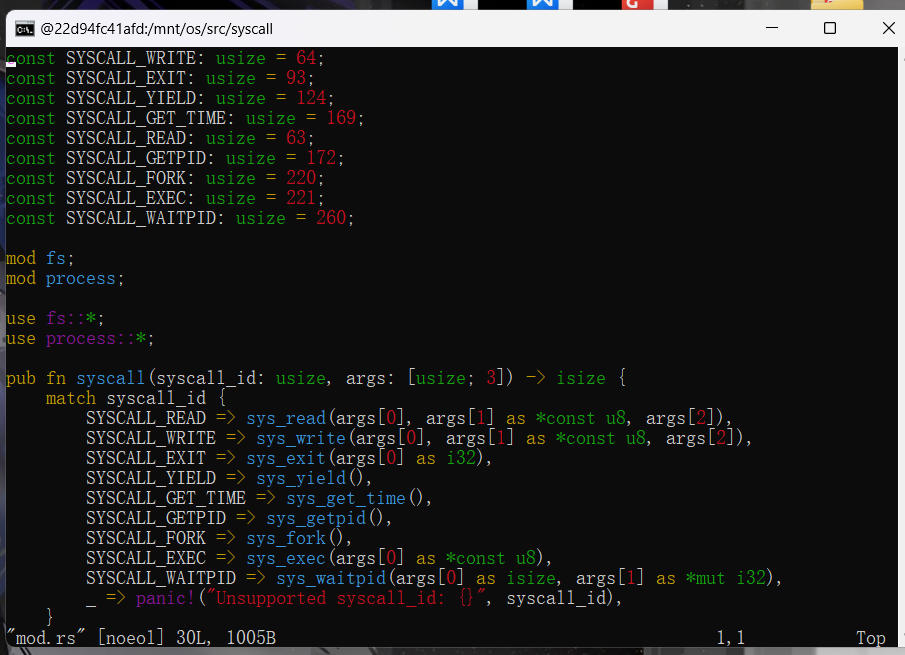


因为Rust的可边长字符串类型String基于动态内存分配，因此还需要在用户库user\_lib中支持动态内存分配。

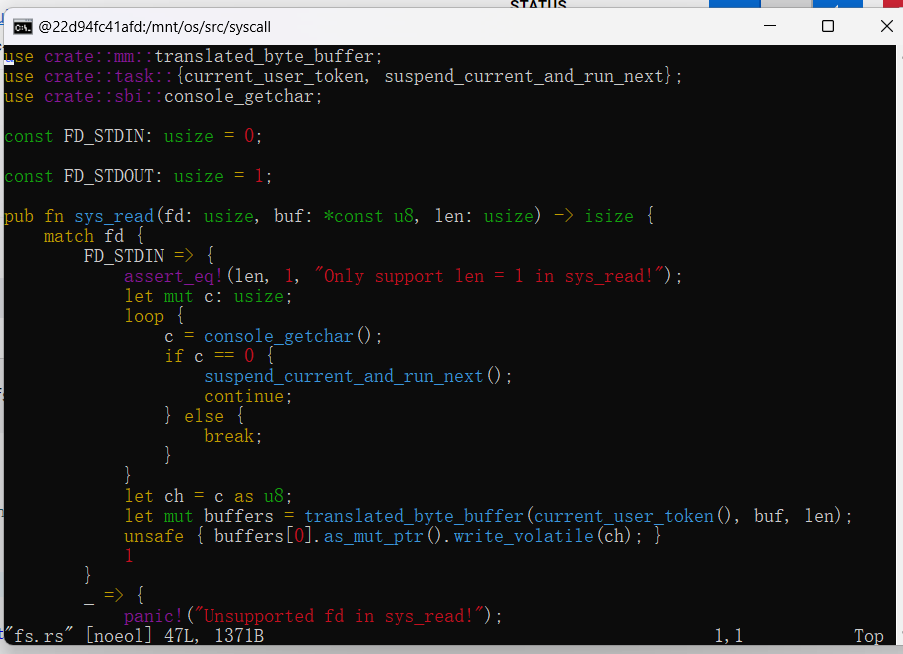


2. 在内核中增加系统调用

首先，修改os/src/syscall/mod.rs增加fork、waitpid、getpid、read系统调用。

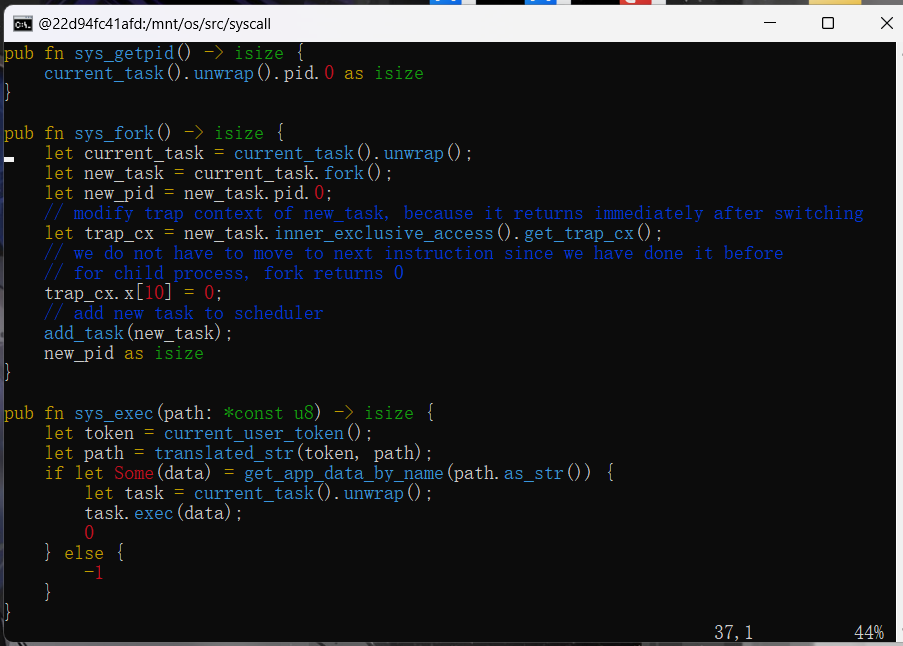


然后，修改os/src/syscall/fs.rs，实现sys\_read系统调用。



其中，suspend\_current\_and\_run\_next函数是暂停当前的任务并切换到下一个任务，具体实现将在后面介绍。然后，修改os/src/syscall/process.rs实现其他系统调用。

//os/src/syscall/process.rs

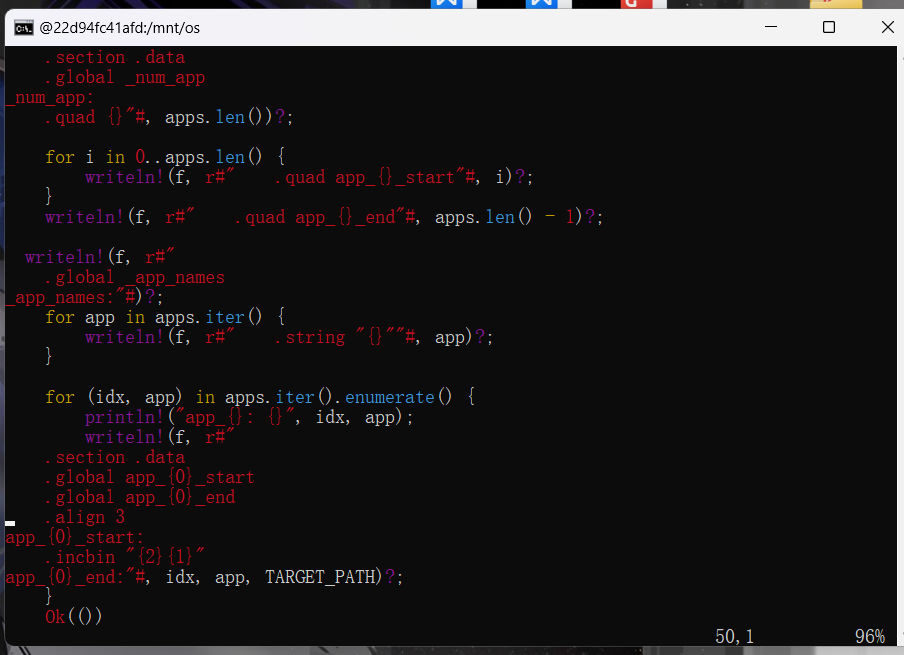


3. 应用的链接与加载

（1）基于名字的应用链接

因为实现exec系统调用需要根据应用程序的名字获取ELF格式的数据，因此需要修改链接和加载接口。

修改编译链接辅助文件os/build.rs。



（2）基于名字的应用加载

应用加载子模块loader.rs会用一个全局可见的只读向量APP\_NAMES按照顺序吧所有应用的名字保存在内存中。

//os/src/loader.rs



4. 进程标识符与内核栈

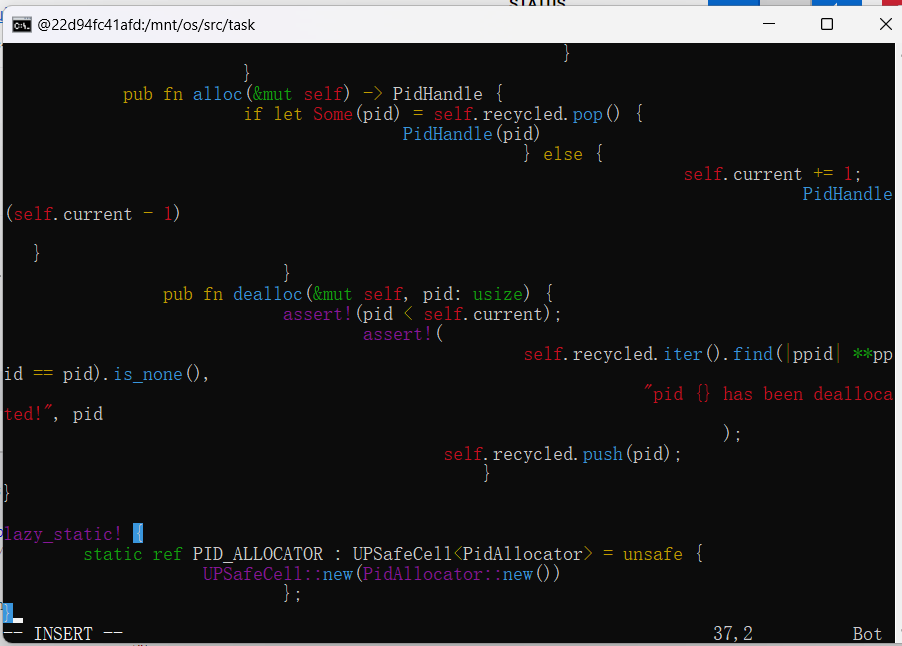
（1）实现进程标识符

进程标识应当是唯一的，我们将其抽象为一个PidHandle类型。

//os/src/task/pid.rs # 注意增加如下代码：

pub struct PidHandle(pub usize);

类似于之前的物理页帧的管理，我们实现一个进程标识符分配器PID\_ALLOCATOR。



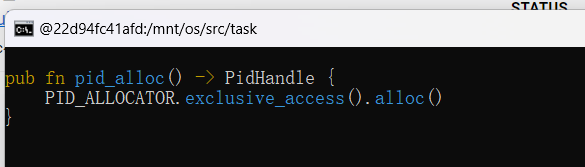
我们还需要封装一个全局的进程标识分配接口pid\_alloc。

//os/src/task/pid.rs

pub fn pid\_alloc() -> PidHandle {

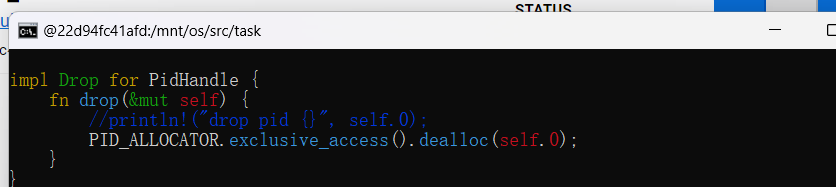
PID\_ALLOCATOR.exclusive\_access().alloc()

}



同时，为了允许资源的自动回收，还需要为PidHandle实现Drop Trait。

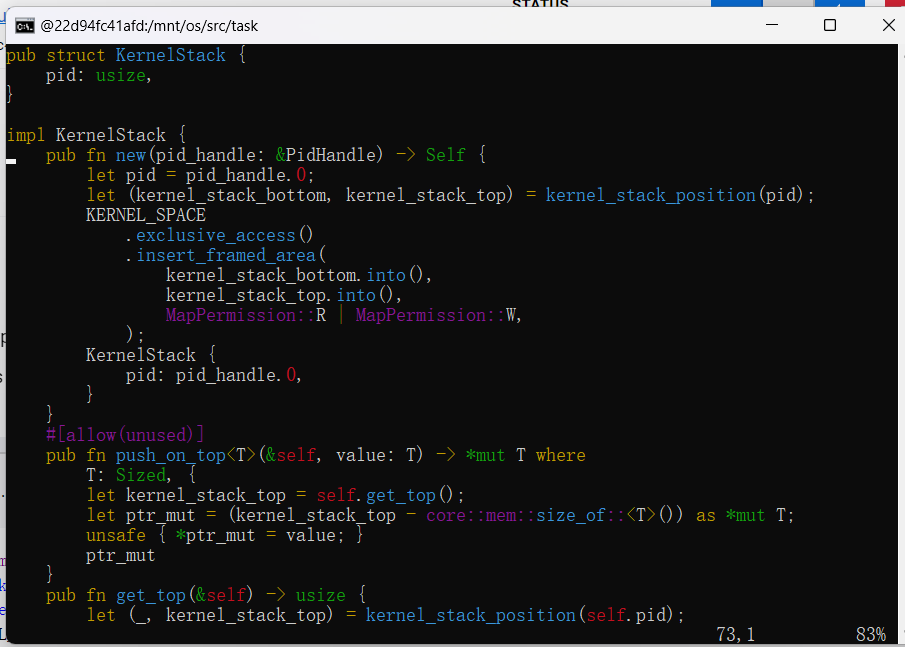
//os/src/task/pid.rs



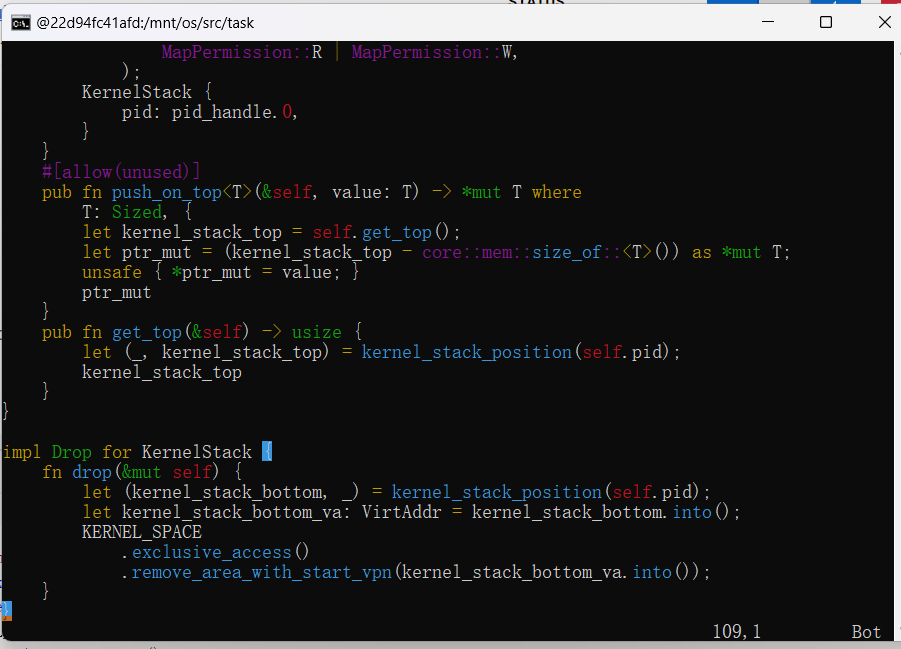
（2）在内核栈中保存进程标识符

重新定义内核栈。

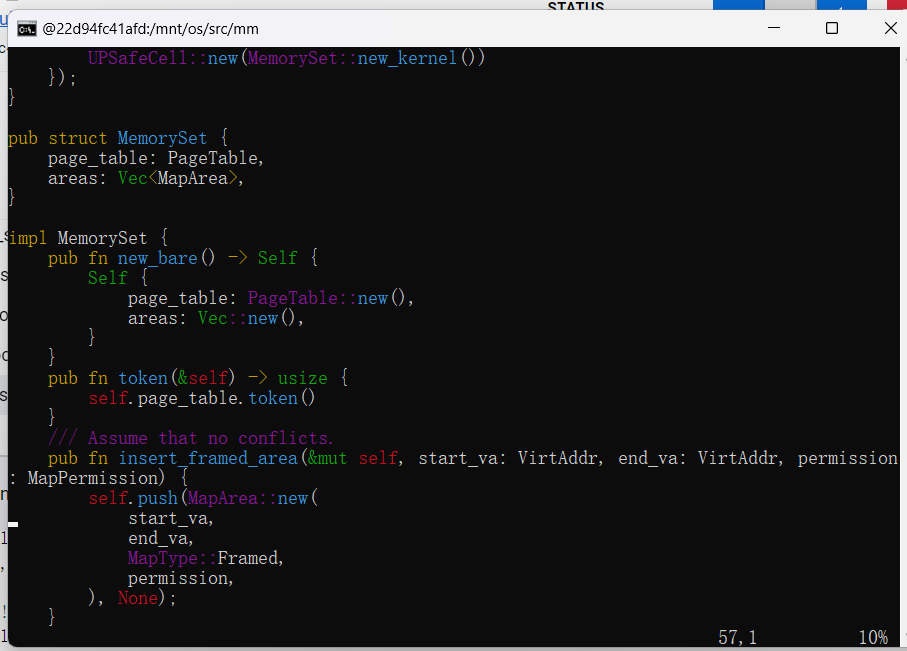
//os/src/task/pid.rs



同时也需要实现KernelStack 的Drop Trait以便KernelStack生命周期结束时回收相应的物理页帧。

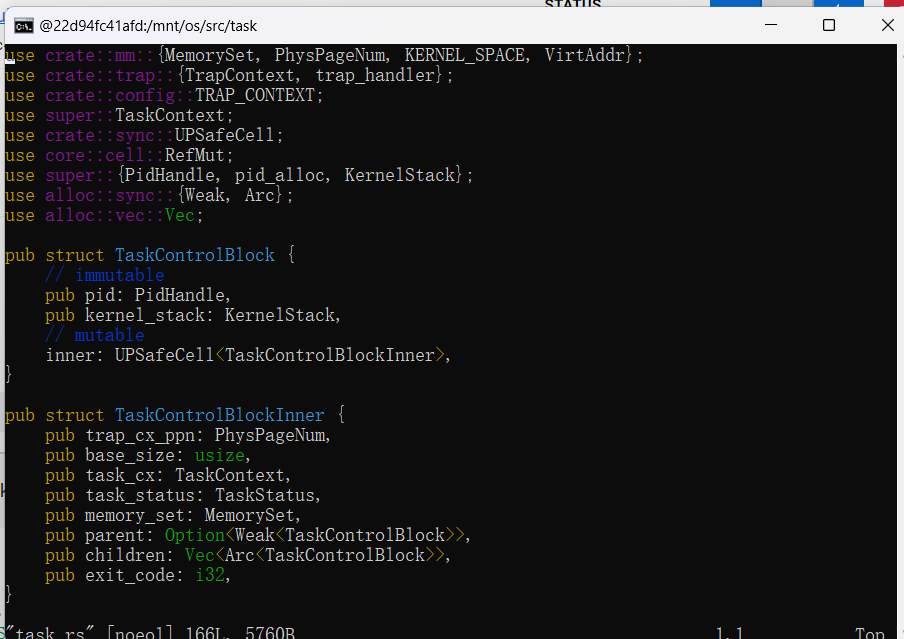


相应的，还需要修改os/src/mm/memory\_set.rs



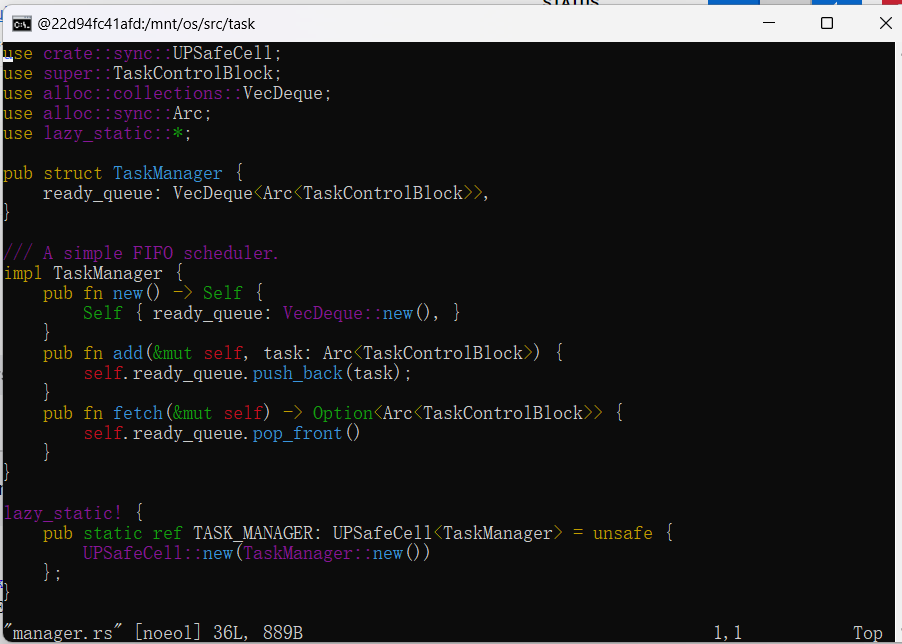
5. 修改实现进程控制块

修改原本的TaskControlBlock实现进程控制块的功能。



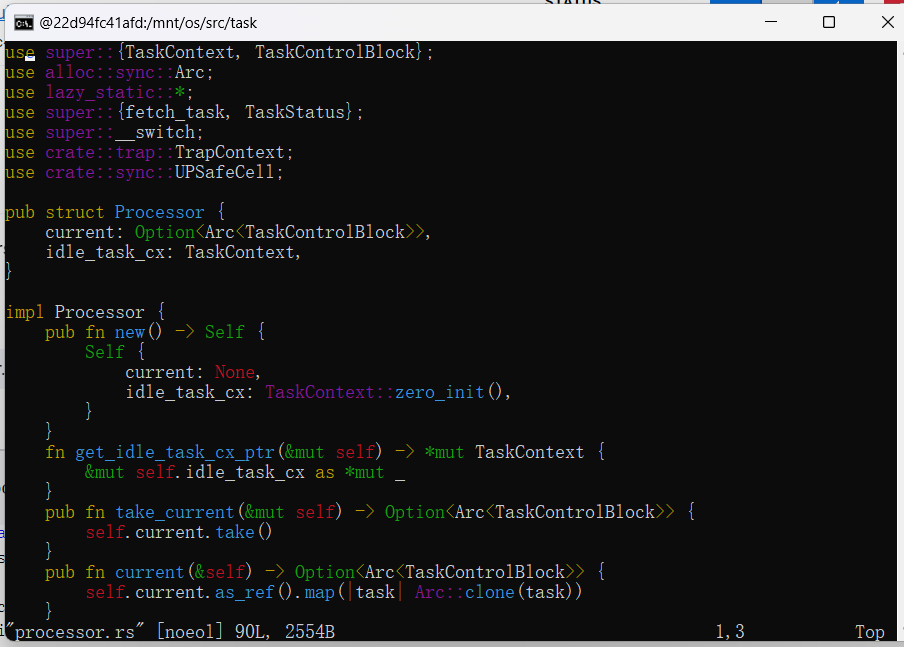
6. 实现任务管理器

修改任务管理器，将部分任务管理功能移到处理器管理中。



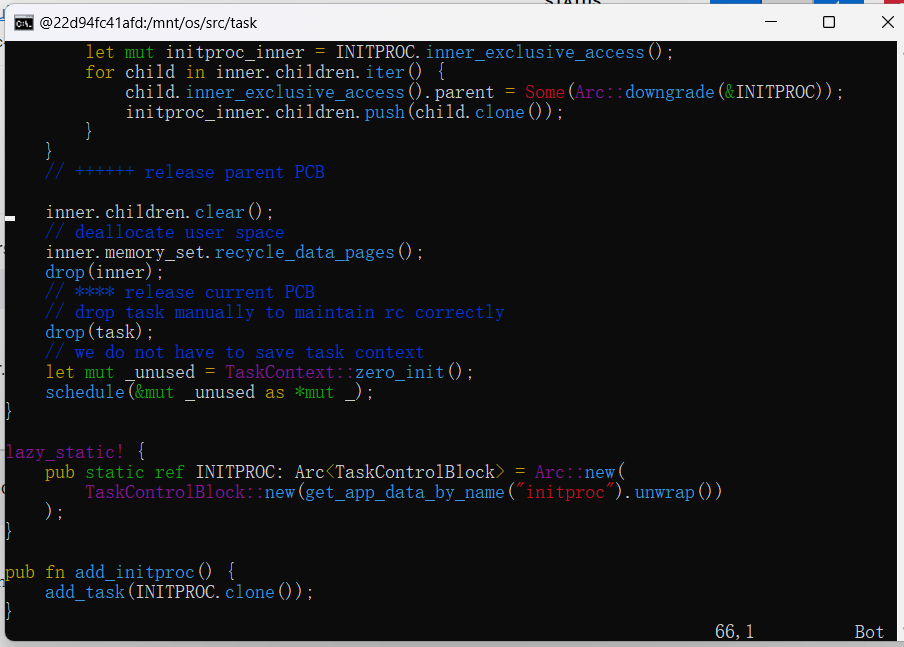
7. 增加处理器管理结构

实现处理器管理结构Processor，完成从任务管理器分离的维护CPU状态的部分功能。



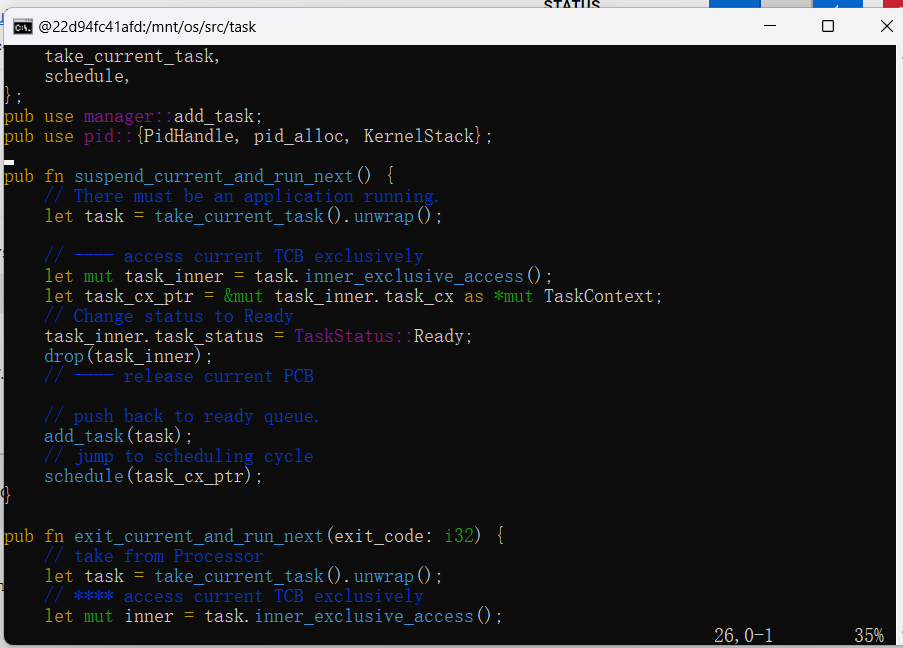
8. 创建初始进程

内核初始化完成后，将会调用task子模块的add\_initproc将初始进程initproc加入任务管理器。在这之前要初始化初始进程的进程控制块。

首先删除os/src/task/mod.rs中TaskManager和TaskManagerInner相关的实现。

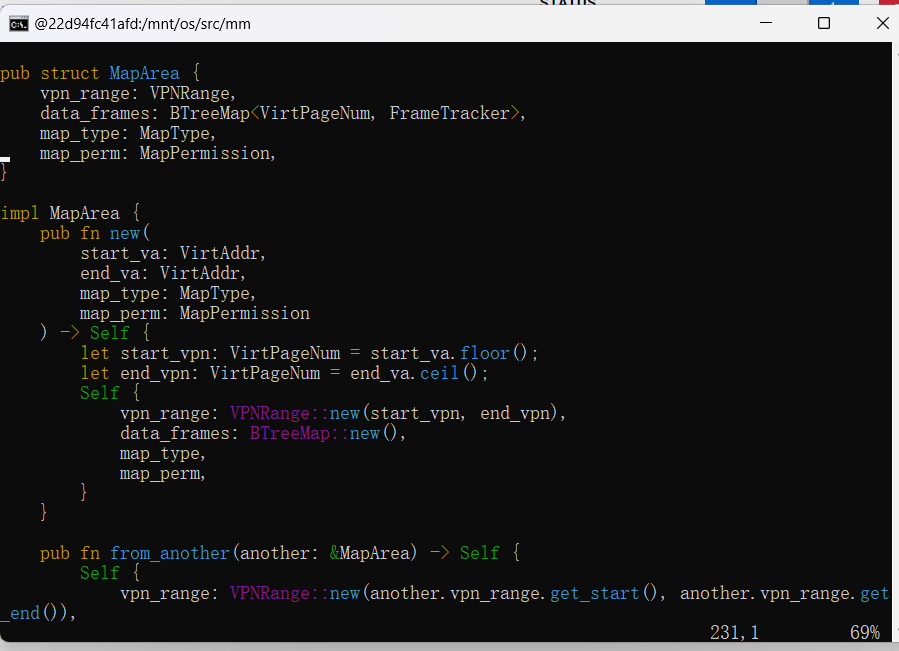
9. 进程调度机制

通过调用 task 子模块提供的 suspend\_current\_and\_run\_next 函数可以暂停当前任务并切换到另外一个任务。因为进程概念的引入，其实现需要更改。



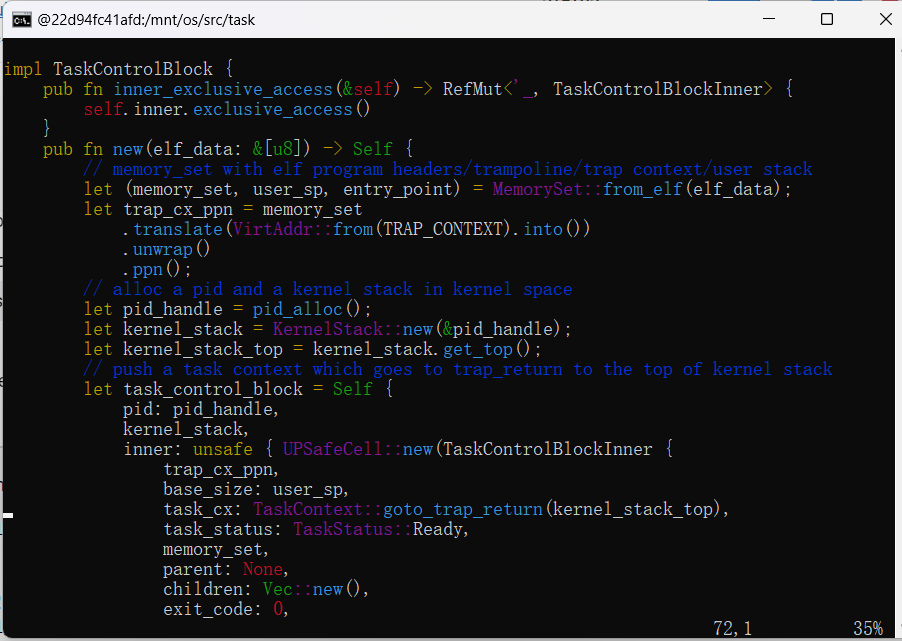
10. 进程的生成机制

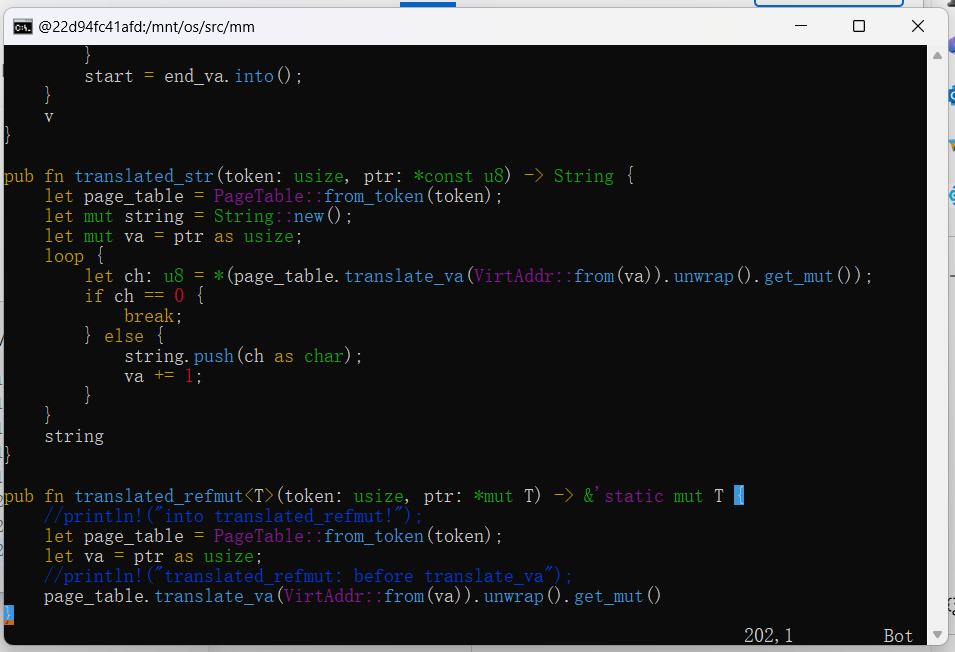
在内核中只有初始进程initproc是手动生成的，其他的进程由初始进程直接或间接fork出来，然后再调用exec系统调用加载并执行可执行文件。所以，进程的生成机制由fork和exec两个系统调用来完成。实现fork系统调用最关键的是为子进程创建一个和父进程几乎相同的地址空间。



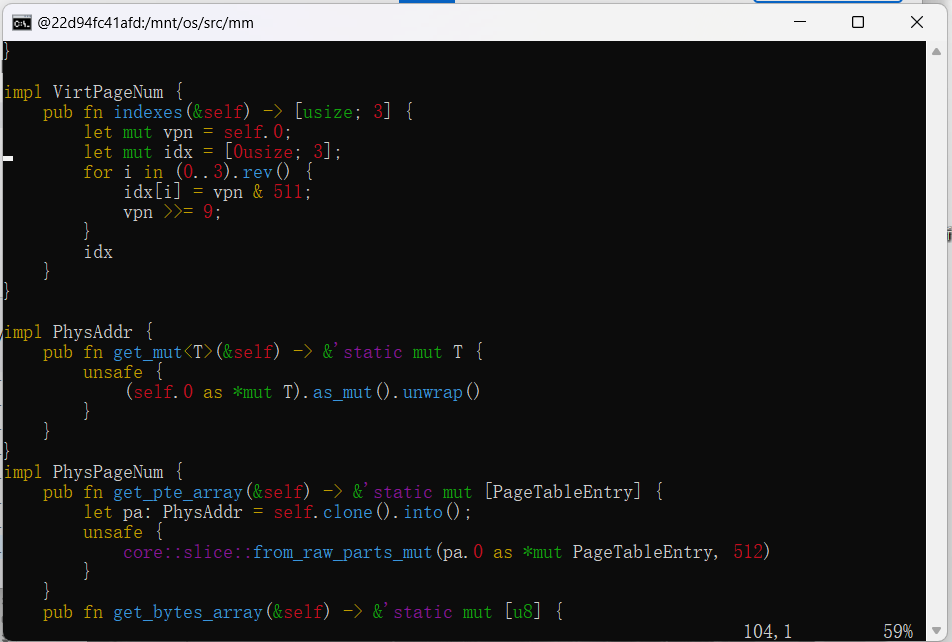
接着，实现 TaskControlBlock::fork 来从父进程的进程控制块创建一份子进程的控制块。

有了exec系统调用后，sys\_exec的实现就很容易理解了。Sys\_exec的实现还依赖于对页表的修改。

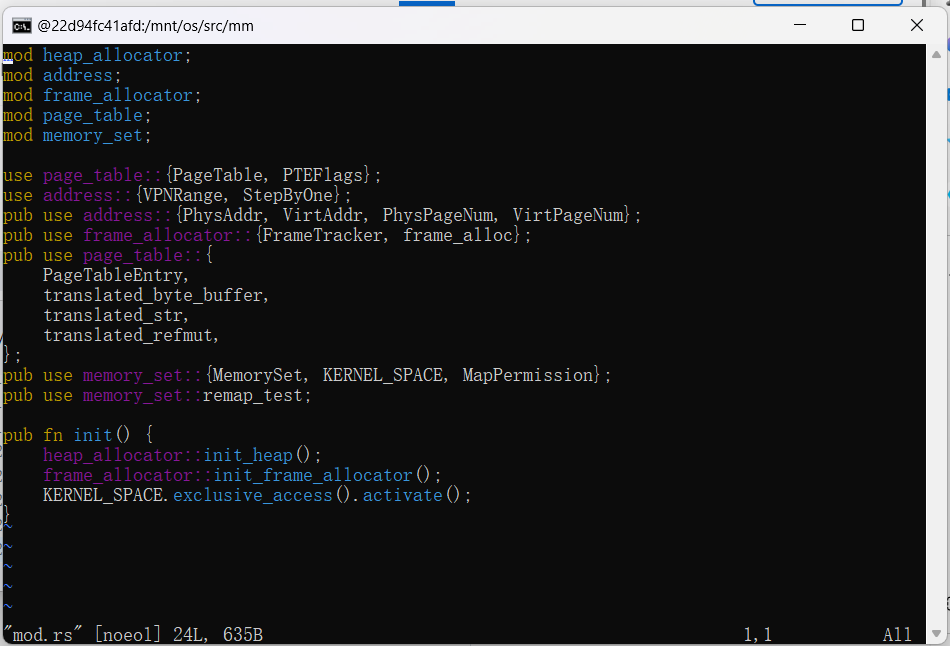




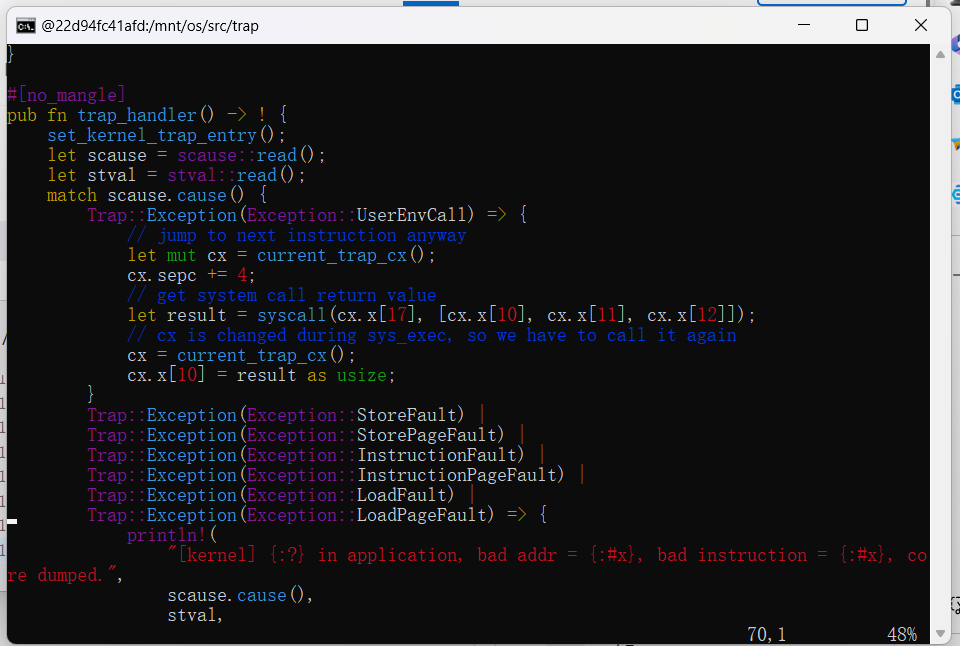
同时，需要修改os/src/mm/address.rs和os/src/mm/mod.rs的内容。



//os/src/mm/mod.rs

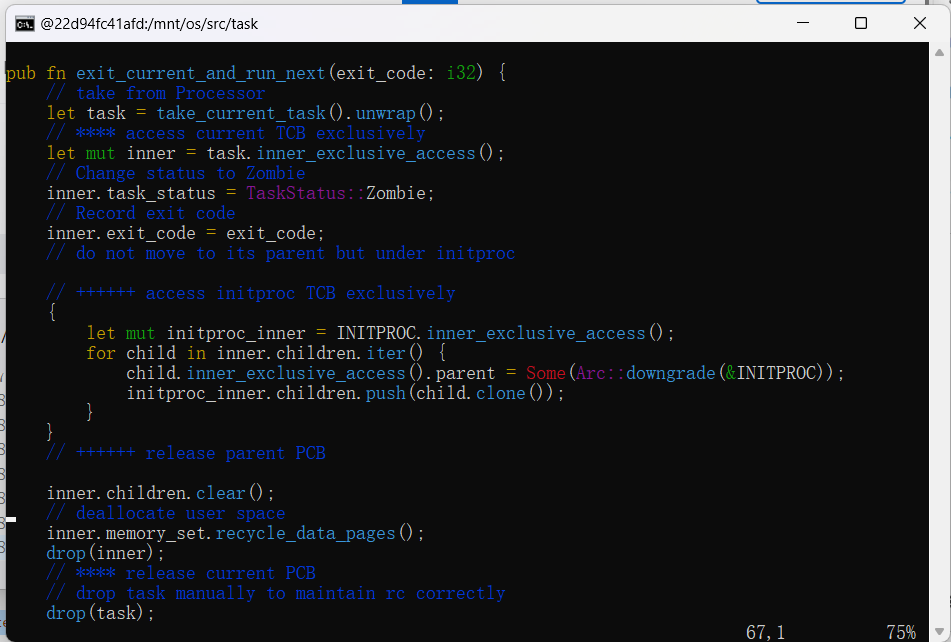


在sys\_exec系统调用后，trap\_handler原来的上下文cx失效了。为此，在syscall分发之后，还需要重新获取trap上下文。



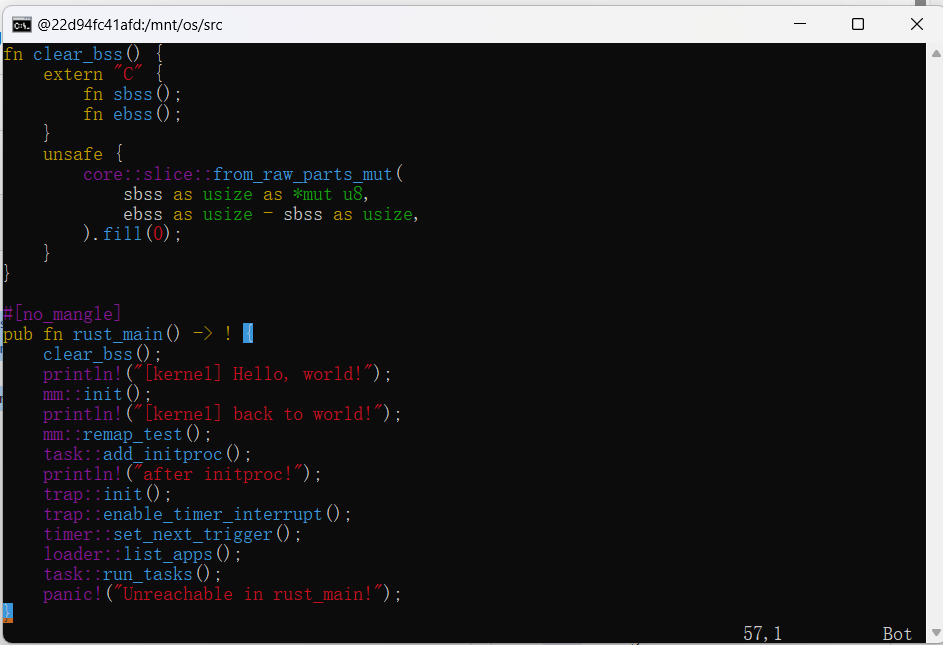
11. 进程资源回收机制

当应用调用 sys\_exit 系统调用主动退出或者出错由内核终止之后，会在内核中调用 exit\_current\_and\_run\_next 函数退出当前进程并切换到下一个进程。相比之前的实现，exit\_current\_and\_run\_next增加了一个退出码作为参数。

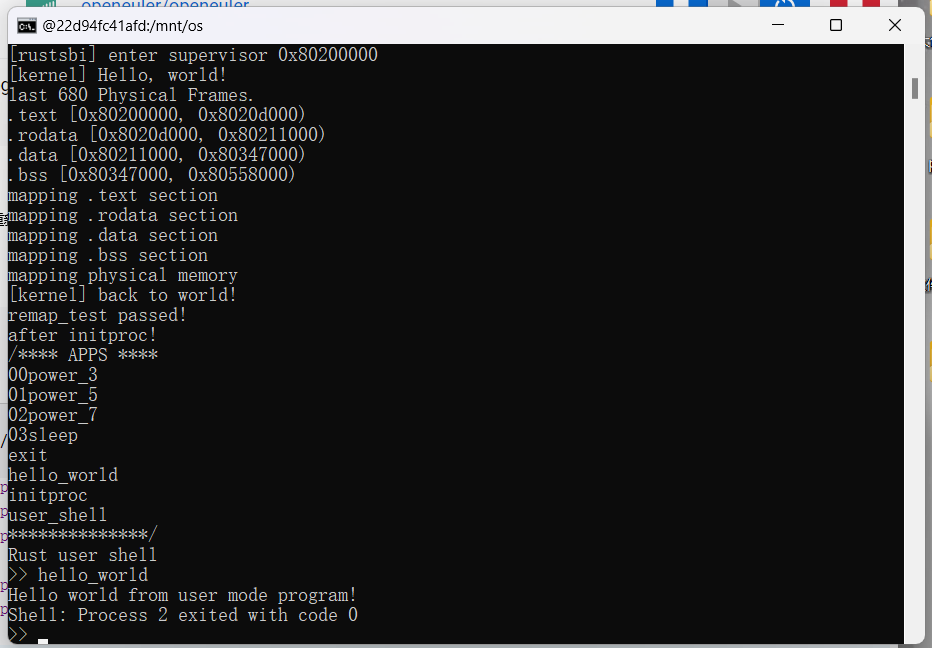


同时，父进程通过 sys\_waitpid 系统调用来回收子进程的资源并收集它的一些信息。

最后，修改main.rs。



至此，具有进程管理功能的操作系统实现完成。



1. 思考问题

（1）分析应用的链接与加载是如何实现的；

应用的链接与加载主要基于应用程序的名字进行，进程标识符的实现和内核栈的管理用于支持进程的唯一标识和内存管理。实现进程控制块、任务管理器、处理器管理结构以及进程调度机制完成了操作系统对进程的管理。

（2）分析进程标识符、进程控制块是如何设计和实现的；

通过定义PidHandle类型和实现相应的分配器，确保了进程标识符的唯一性和资源回收。对TaskControlBlock的修改和TaskStatus的引入有效管理了进程的状态。整体设计基于内核栈的重新定义，以及对任务管理器和处理器管理结构的调整，使得系统能够高效调度和管理进程。最终，通过创建初始进程、调度机制的实现以及进程资源回收机制，成功构建了具有进程管理功能的操作系统。

（3）分析任务管理是如何实现的；

在此实验中，通过修改应用程序和内核，成功引入关键系统调用，如fork、waitpid、getpid、exec和read，实现了进程的创建、等待、执行和文件读取等功能。通过对任务管理器、处理器管理结构和进程控制块的修改，建立了完备的进程调度机制。采用基于名字的应用链接和加载，实现了动态加载和执行不同的应用程序。通过进程标识符和内核栈的管理，确保了进程的唯一性和内存的有效使用。最终，成功创建初始进程、实现进程的生成机制和资源回收机制，为操作系统提供了完整的进程管理功能。

（4）分析进程的调度、生成、以及进程资源的回收是如何实现的。

通过修改应用程序和内核，引入了关键的系统调用，如fork、waitpid、getpid、exec和read，为进程的创建、等待、执行和文件读取提供了基础支持。通过任务管理器和处理器管理结构的调整，建立了有效的进程调度机制，通过suspend\_current\_and\_run\_next函数实现了任务的暂停和切换。基于名字的应用链接和加载确保了灵活的应用程序执行。进程标识符和内核栈的实现保障了进程的唯一性和内存的有效管理。通过TaskControlBlock和TaskStatus的修改，成功实现了进程的控制块和状态管理。最终，通过实现fork、exec和sys\_exit等系统调用，实现了进程的生成和资源回收，使得操作系统具备完整的进程管理功能。

1. Git提交截图

