# 实验指导

首先我先移除 rcore中的user文件夹 重新建立一个

按照rCore中的指导建立配置与基础文件

**定义Makefile**:

build: dependency

# 编译

@cargo build

@echo Targets: $(patsubst $(SRC\_DIR)/%.rs, %, $(SRC\_FILES))

# 移除原有的所有文件

@rm -rf $(OUT\_DIR)

@mkdir -p $(OUT\_DIR)

# 复制编译生成的 ELF 至目标目录

@cp $(BIN\_FILES) $(OUT\_DIR)

# 使用 rcore-fs-fuse 工具进行打包

@rcore-fs-fuse --fs sfs $(IMG\_FILE) $(OUT\_DIR) zip

# 将镜像文件的格式转换为 QEMU 使用的高级格式

@qemu-img convert -f raw $(IMG\_FILE) -O qcow2 $(QCOW\_FILE)

# 提升镜像文件的容量（并非实际大小），来允许更多数据写入

@qemu-img resize $(QCOW\_FILE) +1G

其中SRC\_DIR=src/bin SRC\_FILES：$(wildcard $(SRC\_DIR)/\*.rs)= 获取所有 src/bin/\*.rs文件

Targets就是将这些文件的rs后缀去掉 最后建立build/disk.img镜像文件

**xmas-elf 解析器**

定义readall读取整个文件

在memory\_set文件中定义读取elf的函数,大致流程为下：

首先建立MemorySet 并将其映射到内核态的空间，读取文件头一直到load

获取每个字段的属性，最后建立映射复制数据

**思考题：我们在为用户程序建立映射时，虚拟地址是 ELF 文件中写明的，那物理地址是程序在磁盘中存储的地址吗？这样做有什么问题吗？**

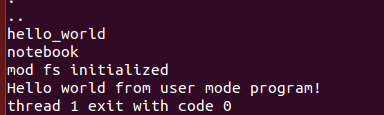
我觉得不是，因为虚拟地址不一定能够映射到磁盘中的地址，这就会为之后的编程带来困难。

**思考题：对于一个页面，有其物理地址、虚拟地址和待加载数据的地址。此时，是不是直接从待加载数据的地址拷贝到页面的虚拟地址，如同 memcpy 一样就可以呢**？

在目前的框架中，只有当线程将要运行时，才会加载其页表。因此，除非我们额外的在每映射一个页面之后，就更新一次页表并且刷新 TLB，否则此时的**虚拟地址**是无法访问的。

但是，我们通过分配器得到了页面的**物理地址**，而这个物理地址实际上已经在内核的线性映射当中了。所以，这里实际上用的是**物理地址**来写入数据。

**运行HelloWorld:**



**实现系统调用：**

我们只需要实现最简单的stdout与stdin即可

首先要把haddler.rs中的外部中断打开，这在之前的实验其实已经打开了

**条件变量：**

wait：当前线程开始等待这个条件变量

notify\_one：让某一个等待此条件变量的线程继续运行

notify\_all：让所有等待此变量的线程继续运行

条件变量与互斥锁的区别：

互斥锁解铃还须系铃人，但条件变量可以由任何来源发出 notify 信号

互斥锁的一次 lock 一定对应一次 unlock，但条件变量多次 notify 只能保证 wait 的线程执行次数不超过 notify 次数

首先需要调整调度器，增加休眠区

定义结构体Condval 使用队列存储所有等待它的变量

**思考**：**如果多个线程同时等待输入流会怎么样？有什么解决方案吗？**

用调度算法解决这个问题就行，选取优先级或者衡量指标中最符合的线程接受

# 习题

1. **原理：使用条件变量之后，分别从线程和操作系统的角度而言读取字符的系统调用是阻塞的还是非阻塞的？**

首先我们先来理解阻塞和非阻塞的区别，简单说来

阻塞就是干不完不准回来，     
非阻塞就是你先干，我现看看有其他事没有，完了告诉我一声

从这个角度来看，对于线程来说是阻塞的，对于操作系统是非阻塞的，因为线程干不完，就得休眠继续等

1. **设计：如果要让用户线程能够使用 Vec 等，需要做哪些工作？如果要让用户线程能够使用大于其栈大小的动态分配空间，需要做哪些工作？**

在用户线程中实现alloc即可。

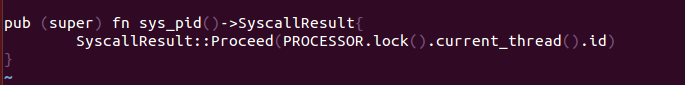
分配一个堆给用户线程

1. **实验：实现 get\_tid 系统调用，使得用户线程可以获取自身的线程 ID。**

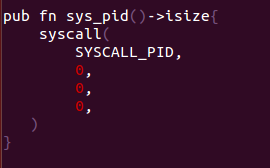
首先在os中定义 SYS\_PID=101；也就是对应的系统调用编号



在match中实现匹配，定义调用函数



接下来需要在用户态中实现



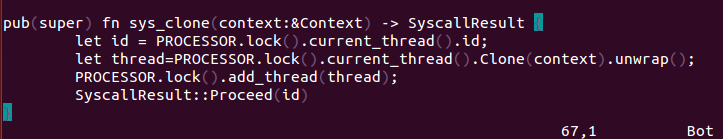
在hello\_world中打印出pid，接下来让我们进行测试



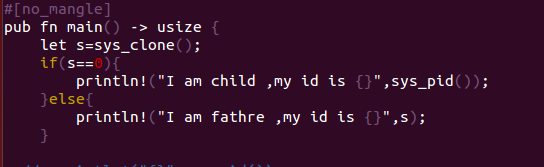
1. **实验：将你在实验四（上）实现的 clone 改进成为 sys\_clone 系统调用，使得该系统调用为父进程返回自身的线程 ID，而为子线程返回 0。**

在用户态中定义 SYS\_CLONE=102；

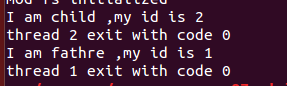
利用写好的Clone函数进行复制



在user中这么定义：



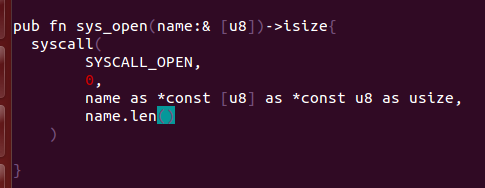
结果:

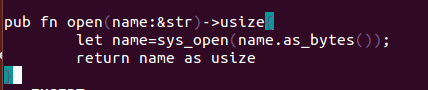


1. **实验：将一个文件打包进用户镜像，并让一个用户进程读取它并打印其内容。需要实现 sys\_open，将文件描述符加入进程的 descriptors 中，然后通过 sys\_read 来读取。**

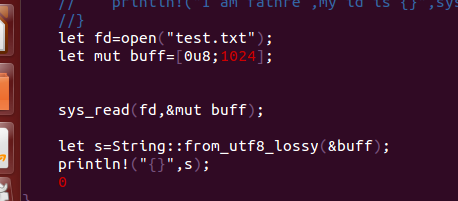
定义 SYS\_OPEN=62；

然后在用户态中定义





然后在主函数中调用



这里的test.txt是跟它在同一目录下的

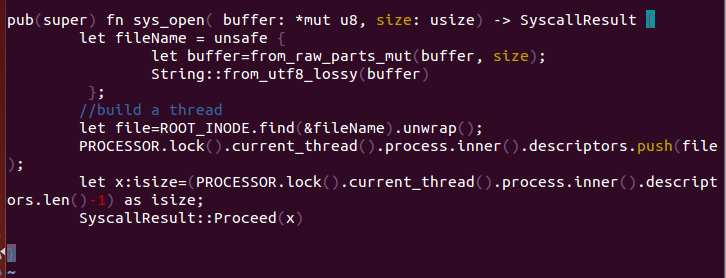
在Makefile中打包





Make build之后就可进入内核中设置

设置接口



结果如下：

