【代码随想录知识星球】项目精讲-操作系统内核（CPP）

Mit6.s081 是公认的高质量的操作系统实践课。

我们做这个lab 不仅是要学习OS，也是要为简历上加一份项目经验

代码随想录知识星球 这次带大家来有目的性的做一做这个lab。

本项目文档将从如下几个方面，侧重求职面试来讲解这个lab

- 项目介绍
- 为什么要做mit6.s081?
- 前置知识
  - 编程语言
  - 操作系统理论知识
  - linux使用基础
- 相关网站
- 如何上手
  - 怎么配置环境
  - 怎么获取代码
  - 怎么调试
  - 常用命令
  - 各lab的难度、耗时以及建议
  - 可参考的答案
- 项目细节
- riscv
- 内存管理
  - 页表合并
  - 面试问题
- 进程管理
  - 面试问题
- 系统调用
  - 系统调用的流程
  - 面试问题
- 锁
  - 这个操作系统有哪些锁？怎么实现的?
  - 怎么降低锁竞争
  - 面试问题
- 文件系统
  - 文件缓存
  - 面试问题
- 其他

    - 面试问题
- 简历写法
- 项目拓展
    - 挑战练习
    - 参考linux
    - 用其他语言重写xv6

前言

项目介绍

Mit6.s081 是麻省理工学院面向本科生的操作系统课程，其课程实验是在教学用操作系统xv6上进行扩展和优化。Xv6 操作系统源代码只有一万余行，并且相对清晰和模块化，每个 lab 都带有自动评测功能，非常适合用于实践操作系统知识。

为什么要做mit6.s081?

- 操作系统是面试的考察重点
  操作系统本身就是面试的必考知识，尤其是面试后端开发、嵌入式开发等岗位时，更是如此。面试者如果拥有良好的操作系统基础，一方面可以提高自己在面试官心中的评级，另一方面也能消磨大量的时间，提高面试成功率。
- mit6.s081有丰富的参考资料
  首先是这门课程的官网，拥有详细的课程视频、讲义、参考书和实验指导等各种参考资料；其次互联网上，有这门课程的视频翻译、字幕翻译、参考书翻译和实验指导书翻译等各种汉化帮助；而且网上还有非常多该课程的学习记录可供参考、理解。这些种种，大大降低了我们学习掌握它的难度。
- 操作系统是最重要的计算机基础之一
  无论我们是打算求职还是继续深造，操作系统都是我们需要深刻理解的基础知识。就算抛开找工作的功利想法来说，深入学习操作系统的实践知识，对我们也有莫大好处。

前置知识

编程语言

既然是操作系统，那自然是c语言为主；虽然其中也有部分汇编语言，但汇编部分不对该项目的学习产生影响，所以不会汇编也可。

操作系统理论知识

具备一定的操作系统理论基础一方面可以让我们更好地学习 mit6.s081 的课程部分，另一方面，如果时间不够，可以让我们尝试跳过课程部分，直接做 lab 。理论知识可以看王道的操作系统考研课程或者《操作系统导论》。

linux使用基础

这个项目中，xv6 操作系统运行在qemu模拟的硬件上，我们需要一定的 linux 基础来配置 qemu 的运行环境、xv6 的编译环境以及开发环境，可以参考 “如何上手” 部分的配置教程。如果看不懂 “如何上手” 部分，可以先学一学linux的基础操作，如20分钟学会Linux的基本操作。

相关网站

- 官网
- 课程视频见官网 b站也有搬运
- 课程字幕翻译
- xv6参考书
- xv6参考书翻译
- 实验指导见官网顶栏的 "labs"
- 实验指导翻译

如何上手

怎么配置环境

- 编译环境：官方文档

ubuntu配置编译环境重点看下面这两部分即可

Installing via APT (Debian/Ubuntu)

Make sure you are running either "bullseye" or "sid" for your debian version (on ubuntu this can be checked by running `cat /etc/debian_version`), then run:

```
sudo apt-get install git build-essential gdb-multiarch qemu-system-misc gcc-riscv64-linux-gnu binutils-riscv64-linux-gnu
```

(The version of QEMU on "buster" is too old, so you'd have to get that separately.)

qemu-system-misc fix

At this moment in time, it seems that the package `qemu-system-misc` has received an update that breaks its compatibility with our kernel. If you run `make qemu` and the script appears to hang after
`qemu-system-riscv64 -machine virt -bios none -kernel kernel/kernel -m 128M -smp 3 -nographic -drive file=fs.img,if=none,format=raw,id=x0 -device virtio-blk-device,drive=x0,bus=virtio-mmio-bus.0`
you'll need to uninstall that package and install an older version:

```
$ sudo apt-get remove qemu-system-misc
$ sudo apt-get install qemu-system-misc=1:4.2-3ubuntu6
```

Testing your Installation

To test your installation, you should be able to check the following:

```
$ riscv64-unknown-elf-gcc --version
riscv64-unknown-elf-gcc (GCC) 10.1.0
...

$ qemu-system-riscv64 --version
QEMU emulator version 5.1.0
```

You should also be able to compile and run xv6:

```
# in the xv6 directory
$ make qemu
# ... lots of output ...
init: starting sh
$
```

To quit qemu type: Ctrl-a x.

- 可参考的开发环境：虚拟机ubuntu + ssh + vscode

> 即在ubuntu虚拟机里配置xv6的编译环境，vscode(mac或win下)通过ssh远程连接到虚拟机当中进行代码开发

可参考此文配置

怎么获取代码

参考此处

怎么调试

为了快速上手，可以先跳过这部分，需要调试时再学

使用gdb进行调试
文章
视频 26:10处

常用命令

- 运行 `make qemu`
- 退出 `Ctrl-a x` (先按Ctrl+a，再按x)
- 测试是否完成lab `make grade`
- 测试是否完成lab的子任务 `make GRADEFLAGS=<lab name> grade`
  - 如在util lab中，想测试是否完成子任务sleep，运行`make GRADEFLAGS=sleep grade`
- gdb调试
  - 一个终端执行 `make CPUS=1 qemu-gdb`
  - 在另一个终端执行 `riscv64-unknown-elf-gdb kernel/kernel`
  - 如果报错`bash: riscv64-unknown-elf-gdb: command not found`可参考此文解决

各lab的难度、耗时以及建议

实验难度出自官网，实验简述和实验耗时是主观体验

- util

  `简述`：使用几个常见的系统调用编程，熟悉开发环境

  `难度`：3 easy 3 moderate

  `耗时`：5h

  `建议`：如果没用过这几个系统调用，做起来可能会有些困难，但不用太灰心，这个lab和后面的lab没有太大关联，不好做直接看参考答案就行，主要是熟悉开发环境

- syscall

  `简述`：添加新的系统调用

  `难度`：2 moderate

  `耗时`：5h

  `建议`：这个lab不难，和后面的traps lab有一定关联

- pgtbl

  `简述`：学习xv6的页表机制，提高用户空间和内核空间之间传递数据的效率

  `难度`：1 easy 2hard 最难的一个lab

耗时： 21h

建议： 这是最难的一个lab，需要理解xv6的页表机制，需要阅读很多源码，但做完这个lab，再做后面的lab就得心应手了。建议通读`kernel/vm.c`。做完lab之后可以总结一下xv6的虚拟内存机制，面试常考题

- traps

  简述： 从系统调用入手梳理中断的全流程

  难度： 1 easy 1 moderate 1hard

  耗时： 7h

  建议： 做完lab后可以梳理总结一下系统调用的全流程，面试常考题

- lazy

  简述： 利用缺页故障实现内存分配的懒分配策略

  难度： 1 easy 2 moderate

  耗时： 8h

  建议： 可以总结一下使用懒分配策略的优缺点

- cow

  简述： 利用缺页故障实现fork系统调用的写时复制机制

  难度： 1 hard

  耗时： 2h

  建议： 本lab和lazy非常相似，完成lazy后，cow就不太费时间了。可以总结一下fork写时复制的优缺点

- thread

  简述: 实现用户级线程；优化并发程序；实现同步屏障

  难度: 3 moderate

  耗时: 3.5h

- lock

  简述: 分别在物理内存页管理和文件缓存页管理这两个场景中降低锁竞争，提高并发程度进而提升性能

  难度: 1 moderate 1 hard 难度仅次于pgtbl

  耗时: 20h

  建议: 在学习课程的时候琢磨这两个问题：1.怎么降低锁竞争？ 2.怎么处理死锁问题?

- fs

  简述: 让xv6支持大文件（原来只支持268MB以下的文件）；实现符号链接（软链接）

  难度: 2 moderate

  耗时: 7h

  建议: 借助这个lab梳理xv6的文件系统，面试常考题

- mmap

  简述: 实现内存映射功能（即mmap系统调用）

  难度: 1 hard

`耗时`: 6.5h

`建议`: 这个lab综合了系统调用、内存管理、文件系统、并发编程等各部分内容，算是对是否通过前面的这些lab掌握了这些知识的一个检验，在顺利通关了前面的lab后，这个lab并不难。

- network

  `简述`: 为网络接口卡（NIC）编写一个xv6设备驱动程序

  `难度`: 1 hard

  `耗时`: 3h

  `建议`: 面试很少被问到，时间不够可以不做

可参考的答案

> 此处主要是搜集到的一些博客

- 解析Ta
- Miigon
- 星見遥
- Doraemonzzz
- 林夕丶
- duguosheng

项目细节

> 水平有限，回答仅供参考
> 部分回答出自 ChatGPT4

riscv

- 1.riscv和arm架构的关系（中望）
  架构类型：
  RISC-V：RISC-V是一种基于精简指令集（RISC）的开放架构，它的指令集是开放的，可以自由使用和定制。RISC-V的设计是开放的，任何人都可以基于其规范来设计和制造RISC-V处理器。
  ARM：ARM是一家公司开发的精简指令集架构，它在嵌入式系统、移动设备和服务器领域广泛使用。ARM的设计是专有的，需要从Arm Limited获得许可才能使用。

开放性：
RISC-V：RISC-V是一种开放架构，其规范对任何人都是免费开放的。这种开放性使得RISC-V受到了广泛的欢迎，尤其在学术界和新兴市场领域。
ARM：ARM架构是专有的，使用ARM指令集需要获得相应的许可。这意味着ARM的使用通常会涉及许可费用。

应用领域：
RISC-V：RISC-V在各种领域中有潜力，包括嵌入式系统、物联网（IoT）、高性能计算和人工智能（AI）。由于其开放性和灵活性，它吸引了许多创新项目和初创公司。
ARM：ARM架构在移动设备、嵌入式系统、服务器和数据中心等领域具有强大的市场份额。它的生态系统和成熟度使得它成为许多传统领域的首选。

- 2.你这个项目是在物理板子上运行的吗？（联想）
  不是，xv6 运行在 qemu 模拟的硬件环境中。

- 3.riscv是几级流水？（联想）
  五级流水。
  取指（Fetch）：从内存中取出指令。
  译码（Decode）：解码指令，确定要执行的操作和操作数。
  执行（Execute）：执行指令的操作，可能涉及算术逻辑运算、内存访问等。
  访存（Memory）：如果指令需要访问内存（如加载或存储指令），则在这个阶段进行内存访问。
  写回（Write Back）：将执行阶段计算的结果写回寄存器文件。

- 3.1 五级流水会不会有pc减的情况？（联想）
  一些特殊情况下，可能会出现PC减的情况，例如：
  分支指令的处理：当处理器遇到条件分支指令时，如果条件不满足，需要跳转到另一个地址执行，这时PC可能会减少到跳转的目标地址。
  异常和中断处理：当处理器检测到异常或中断时，需要保存当前指令的地址（通常是将PC值保存到特定寄存器），然后跳转到异常或中断处理程序的地址。在这种情况下，PC的值会发生改变。
- 3.2 pc正常情况下每次加多少？（联想）
  正常情况下，PC会每次加4（32位系统）或每次加8（64位系统），这是因为指令通常以4字节（32位）或8字节（64位）的固定长度存储在内存中，并且PC的递增步长与指令的长度相匹配。

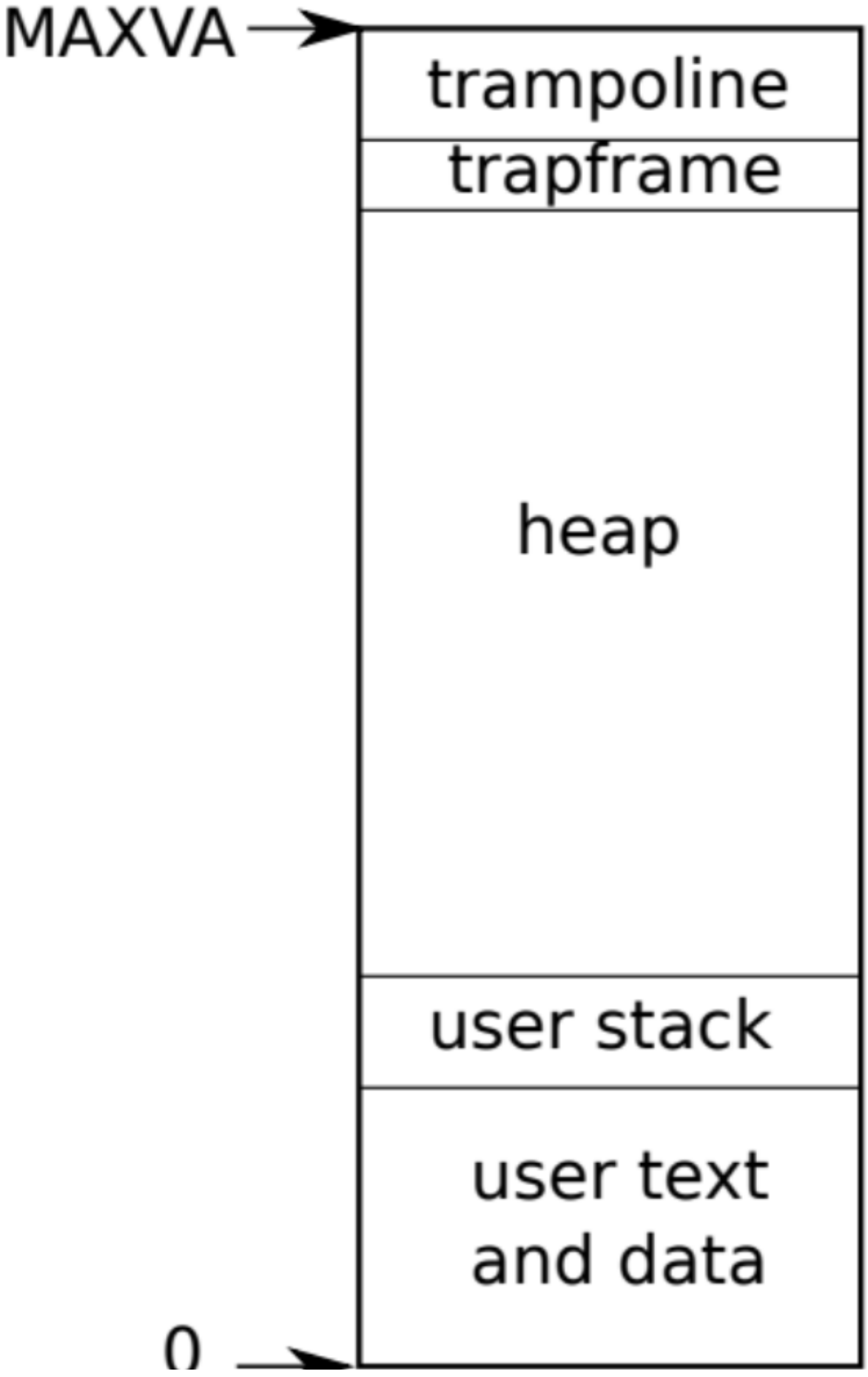
内存管理

建议通读`kernel/vm.c`

xv6通过三级页表管理虚拟内存，内核态页表和用户态页表分离，即程序处于用户态时使用自己的用户态页表，程序通过系统调用等方式陷入内核态时，会将页表切换为内核态页表。

用户态页表每个进程一个，页表地址保存在进程pcb(`kernel/proc.h proc`，以下都用pcb表示)上；内核态页表只有一个，所有进程共享，页表地址每个进程都会保存，在`pcb.trapframe.kernel_satp`上。

进程虚拟地址空间如下图，其中trampoline比较特殊，这块内存可以理解为一种共享内存，每个进程的trampoline都会映射到同一块物理内存当中，这块物理内存保存着`kernel/trampoline.S`，用于进程进出内核态时保存/恢复寄存器、切换栈、切换页表等操作。



页表合并

1. 为什么要合并页表?
   硬件mmu只能解析一个页表，当程序处于内核态时，mmu解析内核态页表，若此时想要访问用户态页表（比如访问一个用户态传入的指针），就没法通过硬件mmu来解析，只能通过软件方法进行解析，但软件方法的性能远低于硬件，故通过合并页表来使得该场景下仍可以使用硬件mmu，进而提高性能。
2. 两个页表不会冲突吗?
   不会。内核态页表采取了直接映射物理内存的方法，而物理地址0至0x80000000交由硬件进行统一编址，这部分地址没有对应的物理地址，内核态页表的这部分地址也就不会被使用到，刚好可以将用户态页表合并到这个位置，不与原有的内核态页表数据产生冲突。

面试问题

- 1.为什么要有虚拟内存？（中望）
  `内存隔离`：虚拟内存允许每个运行的程序都有自己独立的虚拟地址空间，这样不同程序之间的内存不会相互干扰。这提供了强大的内存隔离，防止一个程序的错误或崩溃影响其他程序。
  `更大的地址空间`：虚拟内存可以将物理内存和硬盘上的存储空间组合起来，使得每个程序看起来都拥有一个巨大的地址空间。这允许处理大型数据集或运行大型应用程序，而无需足够的物理内存来容纳所有数据。
  `内存映射和文件共享`：虚拟内存允许将文件映射到进程的地址空间中，这样可以轻松地读取和写入文件，而无需繁琐的文件操作。多个进程还可以共享同一个文件的虚拟内存副本，以实现高效的文件共享。
  `内存管理`：虚拟内存允许操作系统对物理内存的分配和管理进行优化。它可以将不活动的数据页移动到硬盘上，从而释放物理内存供其他程序使用。此外，虚拟内存还支持内存保护和内存权限控制，防止未经授权的访问。
  `交换和页面置换`：虚拟内存使得操作系统能够将不活动的内存页交换到硬盘上，从而在物理内存不足时为活动的程序提供足够的空间。这种页面置换策略有助于维护系统的稳定性和性能。
  `内存分配的简化`：虚拟内存允许程序使用连续的虚拟地址空间，而无需关心物理内存的碎片问题。这简化了内存分配和管理，减少了程序员的工作负担。
- 2.为什么能让所有虚拟内存之和超过物理内存？（中望）
  操作系统利用缺页故障，实现了fork写时复制、内存懒分配策略和磁盘交换等机制，为进程提供了许多 “虚假” 的虚拟内存，这些 “虚假” 的虚拟内存可能并没有对应的物理内存，也可能对应的物理内存被交换到了磁盘当中，从而提供了进程的视角中远超物理内存的虚拟内存量。
- 3.能使用磁盘交换的前提是什么？（中望）
  具有缺页故障机制
- 4.你认为操作系统内存管理要有哪些模块？（腾讯）
  `进程管理`：进程管理模块负责分配和释放内存给每个进程，以及记录每个进程的内存使用情况。这包括创建新的进程、终止进程、分配和回收内存空间。
  `虚拟内存管理`：虚拟内存管理模块允许进程使用虚拟内存，将虚拟地址映射到物理地址。它包括地址翻译、分页机制、页表管理、页面置换算法等。
  `内存分配与释放`：这个模块负责管理系统内存池，以便分配和释放内存块给进程。常见的内存分配算法包括首次适应、最佳适应、最差适应等。
  `页面置换`：当物理内存不足时，页面置换模块决定哪些页面应该被置换到磁盘上，以腾出空间供其他页面使用。常见的页面置换算法包括LRU、LFU、时钟算法等。
  `内存保护`：内存保护模块确保进程不能越界访问其他进程的内存，以及防止未经授权的内存访问。这包括权限位的设置和检查。

`共享内存`：共享内存模块允许多个进程共享相同的内存区域，以便它们可以轻松地进行通信和数据共享。
`内存映射文件`：内存映射文件模块允许文件直接映射到进程的地址空间，以实现高效的文件读写操作。
`页面回收与垃圾收集`：对于操作系统中不再需要的内存块，需要进行回收和垃圾收集，以便将内存资源归还给系统。
`内存监控与性能优化`：监控模块负责跟踪系统内存使用情况，以及性能分析和优化内存管理策略。

- 5.xv6 是怎么分配内存的？（腾讯）
  用户态程序通过malloc申请内存，malloc维护了一个内存池，当malloc无法提供合适的内存时，将调用sbrk系统调用向操作系统申请新的堆内存，sbrk系统调用正常情况下会在页表中做好标记，提供足量的虚拟内存，并通过kalloc申请足量的物理内存映射到新虚拟内存中，便可返回这些虚拟内存供malloc使用。其中物理内存的空闲内存维护通过链表进行，kalloc在该空闲链表上获取物理内存。
- 6.栈是什么样的？（腾讯）
  xv6为每个栈分配两页内存（一页为4KB），一页实际用于栈，另一页用于防止栈溢出时影响其他内存区域。函数调用时会在栈上产生栈帧，每一个栈帧都保存有返回地址、寄存器以及局部变量等信息，函数返回时会将栈帧弹出。
- 7.栈的大小是固定的吗？（腾讯）
  是的，在xv6中栈的大小固定为4KB
- 8.多个线程都使用这一个栈吗？（腾讯）
  xv6 不支持多线程，但在linux中，多个线程不会共用同一个栈，每个线程都有自己私有的线程栈。
- 9.每个线程释放后这个栈会被回收吗？（腾讯）
  一般来说，线程释放时，线程栈也会被回收，但也可以设置线程栈池，以便在需要时可以更快地分配栈内存，这些线程栈可以重复使用，则不一定会被回收。
- 10.假如频繁触发缺页故障，会影响性能吗？（地平线）
  会的，在linux上，可以通过一些配置来减少其对性能的影响。
  调整Swappiness：Swappiness是linux上的一个参数，它控制了Linux系统在内存不足时倾向于使用交换空间的程度。可以通过降低Swappiness值来减少频繁的交换，从而减少缺页故障的影响。
  关闭不必要的后台服务： 停止或禁用不必要的后台服务和进程，以释放内存资源，从而减少缺页故障的机会。
  调整页面大小： 如果系统支持，可以尝试调整页面大小以匹配工作负载需求。这需要深入了解系统和硬件的支持情况，并需要谨慎处理。页面大小通常通过`/proc/sys/vm/page_size`或其他相关参数进行配置。
- 11.讲讲页表的作用（腾讯）
  地址映射： 页表用于将虚拟地址（由应用程序生成）映射到物理地址（实际的内存地址）。每个虚拟页对应着一个物理页框，页表记录了这种映射关系，使得应用程序可以在不知道物理内存布局的情况下访问内存。
  内存隔离： 页表允许多个应用程序同时运行，并使它们的虚拟地址空间相互隔离。每个应用程序都有自己的页表，因此它们的虚拟地址不会相互干扰。
  虚拟内存管理： 页表是虚拟内存管理的核心。通过页表，操作系统可以将虚拟内存中的数据页映射到物理内存中，还可以将不活动的页交换到磁盘上以释放内存。
  内存保护： 页表允许操作系统设置不同的权限和保护位，以限制对内存的访问。例如，只读页表项可以防止写入特定的内存区域，从而提高系统的安全性。
- 11.1.三级页表相对于一级页表的优劣（腾讯）
  三级页表的优势：

1. **内存管理的灵活性：** 三级页表允许更灵活地管理大内存空间，因为它可以将地址空间划分成多个层次，每个层次可以根据需要分配。这样，即使系统具有大量物理内存，也可以有效地管理大型虚拟地址空间。
2. **内存空间的节省：** 三级页表可以根据需要分配，因此对于较小的虚拟地址空间，可以节省内存空间，而不必为整个地址空间分配一页表。这对于嵌入式系统或资源有限的环境很有用。
3. **地址映射的快速性：** 三级页表中的多级结构可以减少地址映射的时间复杂度，因为它将地址映射分成多个步骤，每个步骤都相对较小和高效。这可以提高内存访问的速度。

4. **更好的内存隔离：** 三级页表的多级结构使得更容易实现内存隔离，每个层次的页表可以分配给不同的进程，从而增加了安全性和隔离性。

三级页表的劣势：

1. **额外的内存开销：** 三级页表引入了额外的内存开销，因为每个层次都需要一个页表。对于小型系统或较小的地址空间，这可能会导致内存浪费。
2. **地址映射的复杂性：** 多级页表的实现和管理相对复杂，需要更多的硬件支持和操作系统的代码。这可能增加了系统的复杂性和维护难度。
3. **访问速度的折衷：** 尽管多级页表可以提高地址映射的速度，但它们也会引入额外的访问延迟，因为需要多次查找页表。这可能会对性能产生一定的负面影响。

- 12.讲一讲快表（腾讯）
  快表是页表的一种缓存，将虚拟地址翻译为物理地址时，可以优先访问快表，减少访问内存（页表存储于内存中）的次数，从而提高性能。
- 12.1快表应用了什么原理（腾讯）
  局部性原理

1. **时间局部性（Temporal Locality）：** 这种局部性表现为，如果一个内存地址被访问，那么在不久的将来它可能再次被访问。TLB利用时间局部性的原理，将最近访问的虚拟地址到物理地址的映射关系缓存在高速存储器中，以便快速访问。当处理器需要访问内存时，首先在TLB中查找，以确定是否存在地址映射，如果存在，就能够快速完成地址转换，从而减少了内存访问时间。
2. **空间局部性（Spatial Locality）：** 这种局部性表现为，如果一个内存地址被访问，那么与它相邻的内存地址也可能在不久的将来被访问。TLB通常以页为单位来存储地址映射，因此，当处理器访问某个虚拟页的地址时，TLB不仅会缓存这个地址的映射，还会缓存相邻虚拟页的地址映射，以利用空间局部性。

- 13.用户态分配的内存可以直接被内核态访问到吗？（联想）
  不能，在xv6中，无论是用户态程序还是内核态程序，都只能访问虚拟地址空间，对于某一个程序而言，它只能访问自己的虚拟地址空间，无法感知到其它程序的虚拟地址空间，故而内核态程序也不能直接访问用户态的内存。但内核态程序的虚拟地址空间采取了直接映射的方式，也就是说内核态程序的虚拟地址就是对应的物理地址，如果知道用户态下分配的虚拟地址对应的物理地址，那么内核态程序就可以访问到其内的数据。
- 14.缺页故障当中，如果物理内存不够，会发生什么（大华）
  xv6中将会直接触发panic，没有更多的处理。在其他操作系统中可能会采取一定的措施来获取物理内存，比如将部分物理页交换到磁盘中、触发OOM机制等。
- 15.堆和栈的区别是什么（科大讯飞）
  分配方式：
  - **栈（Stack）：** 栈是一种自动分配和释放内存的数据结构，通常由编译器或运行时系统管理。数据在栈上以线性方式分配，栈帧（函数调用的局部变量和参数）在栈上按照“先进后出”（LIFO）的原则进行管理。栈的分配和释放是隐式的，发生在函数的进入和退出时。
  - **堆（Heap）：** 堆是一种手动分配和释放内存的数据结构，通常由程序员来管理。数据在堆上分配的顺序和释放的顺序可以是任意的，需要程序员显式地分配内存，并在不再需要时显式地释放它。
    生命周期：
  - **栈：** 栈中的数据具有较短的生命周期，通常与包含它们的函数的执行周期相关。当函数返回时，栈帧中的数据会被销毁，因此栈中的数据不适合长期存储。
  - **堆：** 堆中的数据可以具有较长的生命周期，可以在不同函数之间共享和保持状态。数据在堆上分配后，会一直存在，直到程序员显式释放它。
    空间大小：
  - **栈：** 栈的大小通常有限，由操作系统或编译器指定。栈的大小取决于函数调用的嵌套深度和操作系统的

限制。
    - **堆：** 堆的大小通常较大，受系统内存限制，可以动态地分配和扩展，但需要小心管理以防止内存泄漏。
      访问速度：
    - **栈：** 栈上的数据访问速度较快，因为栈的分配和释放是隐式的，通常只需要增加或减少栈指针来访问数据。
    - **堆：** 堆上的数据访问速度较慢，因为需要通过指针来访问堆中的数据，还需要进行内存管理，如分配和释放。
      用途：
    - **栈：** 用于存储局部变量、函数调用的参数和控制流信息。栈的生命周期短，适用于临时数据。
    - **堆：** 用于存储动态分配的数据，如动态数组、对象实例等。堆的生命周期可以很长，适用于需要长期存储和共享的数据。
- 16.malloc的内存池找不到合适的内存块时，会怎么做，讲一讲操作系统底层的做法（美团）
  参考 “5 xv6 是怎么分配内存的？”

进程管理

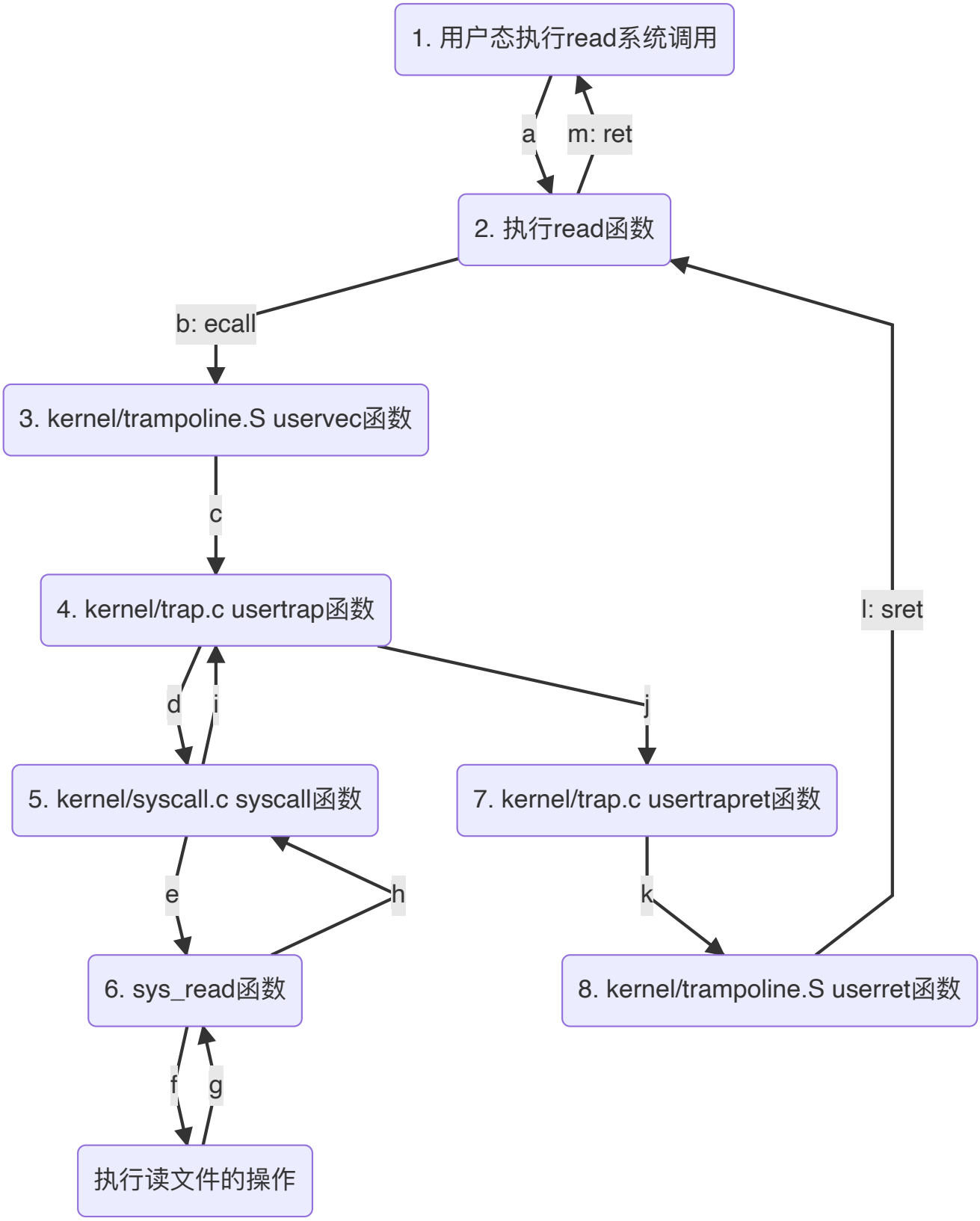
面试问题

- 1.进程、线程、协程的区别？（地平线 | 腾讯）
  进程是操作系统分配资源的最小单位，线程是在CPU上运行的最小单位，一个进程拥有多个线程，同一个进程下的多个线程共享这个进程的资源。协程可以理解为用户态线程，其上下文切换的过程在程序中进行，不由操作系统管理，在内核态的视角中无法察觉协程的存在。
- **2.讲讲你如何实现的用户态线程（腾讯*）**
  在用户态程序当中，将用户态线程用一个struct表示，其中只有线程栈（字符数组）、线程状态和线程上下文这三个成员，其中线程上下文也是一个struct，其中包含了十几个必须的寄存器值，用uint64表示。线程间进行切换时，修改线程状态，再切换线程上下文即可，线程栈可由上下文中的sp寄存器指引，整个切换过程都不会涉及内核态。

系统调用

系统调用的流程

在xv6中，以read系统调用为例，系统调用的实现流程大致如下



1. 用户态进程执行系统调用;
2. 由于系统调用是以函数形式暴露给进程的，所以此时会执行对应的函数。这些系统调用对应的函数在一个汇编文件中，每个函数都只有三条汇编指令:

```
read:
 li a7, SYS_read  ; 将对应的系统调用号放到a7寄存器中
 ecall  ; 执行中断隐指令进入内核
 ret  ; 返回到调用系统调用的下一条指令
```

ecall指令就是理论课上学习的中断隐指令，其作用如下

> a. 关中断
> b. 从用户模式升级为管理模式
> c. 设置寄存器以保存原pc、原模式并设置产生trap的原因
> d. 跳转到trampoline.S 中的uservec函数继续执行

3. 这里有个很有意思的点，**为什么是跳转到一个汇编文件中的函数而不是直接跳转到处理trap的函数？** 原因在于ecall指令为了保证灵活性，执行了尽可能少的工作，这导致页表没有切换到内核页表，内核栈也没有切换，此时无法执行内核的任何代码。所以，trampoline.S uservec函数的作用就是**切换到内核页表、切换内核栈以及保存断点** 也就是各种寄存器。**这里还有一个问题是，trampoline.S切换了页表，为什么系统没有崩溃？** 常理来说，由于pc寄存器是依据页表上的地址来执行指令的，一旦切换页表，由于新页表上的内容未知，pc寄存器指向了一个未知地址，那个地址的内容可能是不可执行的，可能是只读的，程序很快就会崩溃。这里trampoline.S没有崩溃的原因在于，**其在用户页表和内核页表的映射位置是固定且相同的，无论是在用户页表还是在内核页表中，trampoline.S都映射到了页表最顶部的一页，** 这使得切换页表后，pc指向的依旧是trampoline.S的代码，程序也就不会崩溃。从这里也可以推导出来，从内核返回到用户进程时，也会在trampoline.S里切换页表。
4. trampoline.S uservec函数最后会跳转到kernel/trap.c usertrap函数继续执行，此时其成功的进入了内核。usertrap函数会依据ecall在scause寄存器中设置的产生trap的原因来执行不同的操作，对于系统调用，usertrap函数会调用kernel/syscall.c syscall函数继续执行。
5. kernel/syscall.c syscall函数会依据ecall指令的前一条指令设置的系统调用号来确定要执行的函数，比如SYS_read宏定义为了数字5，其对应的函数就是sys_read函数，所以syscall将会调用sys_read函数。
6. sys_read函数会按照需求读取a0至a5中的寄存器，这些寄存器中保存的就是一开始用户执行系统调用的参数，对于read系统调用来说，就是文件描述符，读缓冲区，数据缓冲区的大小。读取完参数后，sys_read就会调用真正执行操作的函数，比如去读取文件，每个系统调用在此处执行的函数就不相同了。
7. 执行完操作后，会返回到sys_read函数，sys_read也会继续原路返回，sys_scall，usertrap。usertrap执行完后会调用usertrapret函数这个函数会设置部分寄存器的值来控制返回到用户进程的状态，比如设置spec寄存器可以控制返回到用户进程的位置，会设置sstatus寄存器以控制开关中断以及返回用户进程后的模式。
8. usertrapret函数会接着调用trampoline.S中的userret函数，trampoline.S文件中就只有两个函数，分别是前文提到的uservec函数和这里的userret函数，这两个函数执行相反的操作，userret函数会**切换到用户页表、切换内核栈以及恢复保存的断点**也就是各种寄存器，最后通过sret函数返回到ecall指令的下一条指令。
9. ecall指令的下一条指令是ret指令，通过ret指令再返回到调用系统调用的下一条指令，至此，系统调用执行结束。

面试问题

- 1.介绍一下上下文切换（宁德时代）
  在 xv6 中，程序的上下文指的是其通用寄存器和栈的状态，由于部分通用寄存器也会保存在栈中，所以上下文保存的是未保存于栈中的通用寄存器以及栈底指针sp。上下文切换主要发生在进程调度的时候，切换过程会保存当前运行进程的相关信息，并将另一进程的相关信息载入到相关寄存器中。
- 2.trampoline.S的作用是什么？（地平线）
  trampoline.S 中包含两个用户态和内核态间进行转换的必要函数，它们的作用很类似，以用户态向内核态转换的函数为例，其作用是保存用户态下的通用寄存器状态、载入之前保存的内核态通用寄存器状态、切换页

表、刷新快表以及切换函数栈，为了在切换后能够继续执行指令，trampoline.S 被保存在了共享内存中，即其物理地址会映射到每个进程虚拟地址空间的相同地址上。

- 3.系统调用的整个流程是什么？（美团 | 联想）
  参考上文。

锁

这个操作系统有哪些锁？怎么实现的?

自旋锁和睡眠锁

- 自旋锁
  见`kernel/spinlock.c`，由test_and_set原子指令实现
- 睡眠锁
  见`kernel/sleeplock.c`，由自旋锁+循环+sleep实现

怎么降低锁竞争

- 读写锁
- 无锁数据结构
- 将一把大锁分解为多把小锁，从而降低每把小锁的锁竞争

面试问题

- 1.说一下你是怎么降低锁争用的（Buffer cache实验）？（腾讯 | 美团 | 同为）
  在这个实验中，所有的缓存块通过一条链表连接，用一个锁来保证多进程访问这些缓存块的安全性，带来的问题就是这些进程只能串行访问文件缓存，性能不高。我的应对措施是将这些缓存块用多条链表连接，每条链表一个锁，从而降低每个锁的锁争用，提升性能。
- 2.讲一讲你了解过的锁（腾讯）
  在 xv6 中有两种锁，分别是自旋锁和睡眠锁。自旋锁由test_and_set原子指令实现，其特点是进程会持续获取锁，直到持有锁为止；睡眠锁由自旋锁加sleep机制实现，其特点是进程申请锁失败后会让出cpu，直到锁空闲才会被唤醒。
- 3.睡眠锁和自旋锁各自的应用场景是什么（腾讯）
  自旋锁适用于那些锁持有时间非常短的场景。因为在等待锁的过程中，线程会持续占用CPU进行循环（即“自旋”），这使得自旋锁在等待时间非常短的情况下非常高效。睡眠锁适用于那些锁持有时间较长的场景。当一个线程试图获得一个已被占用的锁时，它会进入睡眠状态，释放CPU给其他线程使用。
- 4.怎么处理死锁问题（美团 | 同为）
  预防死锁：
  - **避免多重锁定**：尽量避免一个线程同时获取多个锁。如果确实需要多重锁定，确保所有线程以相同的顺序获取锁。
  - **使用锁超时**：在尝试获取锁时使用超时机制。这允许线程在等待锁超过一定时间后放弃，从而防止永久阻塞。
  - **资源分配顺序**：确保所有线程以固定的顺序请求资源，这可以减少死锁的可能性。
    避免死锁：
  - **银行家算法**：这是一种避免死锁的算法，通过预先计算分配资源后是否会导致不安全状态来决定是否分配资源。
  - **资源分配图分析**：通过分析资源分配图检测是否有循环等待条件，从而避免死锁。
    检测死锁和恢复：

- **死锁检测**：定期检测资源分配图是否有循环等待的情况。这通常由操作系统或者特定的监控工具完成。
- **资源剥夺**：一旦检测到死锁，可以通过剥夺和重新分配资源来打破死锁。例如，中断一个线程，回滚其操作并释放其持有的资源。
  使用并发库和工具：
- **利用现代编程语言的并发库**：许多现代编程语言都提供了高级的并发和锁定机制，这些机制内部包含了避免死锁的策略。
- **使用不可重入锁**：不可重入锁（也称为互斥锁）确保同一线程不会重复获取同一锁，从而减少死锁的可能性。
  良好的设计和编程实践：
- **最小化锁的作用范围**：只在必要的最短时间内持有锁。
- **避免在持锁时调用外部代码**：尽量避免在持有锁的状态下调用可能再次获取锁或长时间运行的外部代码。
  测试和代码审查：
- **死锁检测工具**：使用工具对代码进行静态分析，以找出潜在的死锁风险。
- **代码审查和测试**：定期进行代码审查和并发测试来识别可能导致死锁的问题。

文件系统

文件缓存

由于涉及到了lru，面试官很可能会让你实现一下lru作为算法考察，建议多刷几遍146. LRU 缓存

面试问题

- 1.为什么xv6的单个文件限制是268KB而不是256KB这种2的n次方的数？（美团）
  xv6 文件系统中的 inode 在索引文件数据时混合使用了直接和间接的方法，inode 有12个直接索引和1个间接索引，每个直接索引指向一个数据块（1KB），共直接索引了12KB，间接索引也指向一个数据块，但数据块上存储了256个二次索引（每个索引4字节），每个二次索引指向一个数据块，共间接索引了256KB，加起来就是268KB。
- 2.为什么操作系统里要设置文件缓存（腾讯）
  主要的目的是减少磁盘I/O，进而提高数据访问速度

其他

面试问题

- 1.哪些部分是xv6原有的，哪些是你实现的？（腾讯）
  mit6.s081 已经提供了一个简易的可运行的 xv6 操作系统，我的工作只是在其上进行扩展和优化，具体来说，我所实现的各个部分都写在了简历上。
- 2.mit6.s081课程的学习形式是怎么样的？（腾讯）
  公开形式，mit6.s081 有专门的学习网站，他们提供课程视频、讲义、参考资料和实验指导等各种学习辅导，他们还提供了专门为教学而设计的简易操作系统内核xv6，我这个项目就是在xv6中进行各种扩展和优化。
- 3.你有读过linux的源码吗？（腾讯 | 科大讯飞）
- 4.介绍一下段错误的排查解决方式（宁德时代 | 地平线）
  使用gdb调试段错误产生的coredump，在产生段错误的位置会停下，此时用 bt（backtrace）命令查看函数调用栈，就可定位产生段错误的函数，再阅读修改对应源码，解决问题。

- 4.1 段错误除了数组越界和野指针之外，还有哪些情况会触发（地平线）
  栈溢出：
  - 当程序使用的栈内存超过了分配给栈的空间时，会发生栈溢出。这通常是由于过深的递归调用或在栈上分配了大量的局部变量。
    对只读内存的写操作：
  - 尝试修改存储在只读内存段（如文字常量）的数据时，会引发段错误。
    无效的动态内存操作：
  - 双重释放（double free）：对同一个内存地址调用两次 `free`。
  - 释放非动态分配的内存：试图释放不是通过 `malloc`、`calloc` 或 `realloc` 分配的内存。
  - 释放已经释放的内存：释放之后没有将指针设置为 NULL，导致再次释放相同的内存地址。
    不正确的类型转换或对齐：
  - 错误地将一个类型的指针转换为另一个类型的指针，并通过该指针进行读写操作，可能会导致段错误。
  - 对于某些类型的数据，如果它们的地址没有正确对齐，访问它们也可能导致段错误。
    内存映射文件错误：
  - 如果通过 `mmap` 等系统调用映射文件到内存，但访问的范围超出了映射区域，或者文件被意外关闭，也可能导致段错误。，
- 5.是否做过单片机开发或者内操作系统移植（宁德时代）
- 6.你在做mit6.s081项目的时候遇到的最大的难题是什么？（腾讯）
- 7.操作系统的前十行一般会做什么事情？（联想）
  xv6 的前十行是汇编指令，主要作用是配置C语言环境，再跳转到C语言中进行其他初始化操作。
- 8.你这个项目运行在哪里（大华）
  qemu 模拟的硬件上

简历写法

基本格式参照简历中【项目经验】这么写将项目经历拆分为项目描述、主要工作、个人收获和项目难点等部分。在主要工作部分，没必要将所有lab的每一个任务点都写上，可以挑出自己最熟悉、最能聊的任务点写上；项目难点部分可写可不写，但面试官大概率会问 “你自己认为这个项目里最难的点是什么？” 等问题，最好提前准备。

可以参考下列写法：

基于RISC-V指令集架构的操作系统内核 2023-01~2023-03

项目描述：本项目以基于RISC-V指令集架构的xv6操作系统为基础，对其系统调用、内存管理、进程管理、文件系统、中断等模块进行扩展与优化。

主要工作：

1. 理解xv6**系统调用**过程，增加新的系统调用；
2. 理解xv6**虚拟内存**机制，探索**页表**机制并修改，简化用户空间和内核空间之间传递数据的流程；
3. 理解xv6内中断处理流程，基于内中断机制实现alarm系统调用；
4. 利用**缺页故障**，在xv6上实现内存页面的**懒分配**和**写时复制**；
5. 理解xv6进程调度和上下文切换过程，以此为参照在用户态实现**协程**；
6. 更改数据结构和锁策略，降低内存空闲页分配与释放、磁盘块缓存块的使用过程中的锁争用；
7. 理解并修改xv6的文件系统，使单个文件的大小限制由268KB改进为65803KB；实现软链接；
8. 在xv6上实现内存映射文件，将文件映射到内存中，减少访问磁盘的次数。

项目难点：

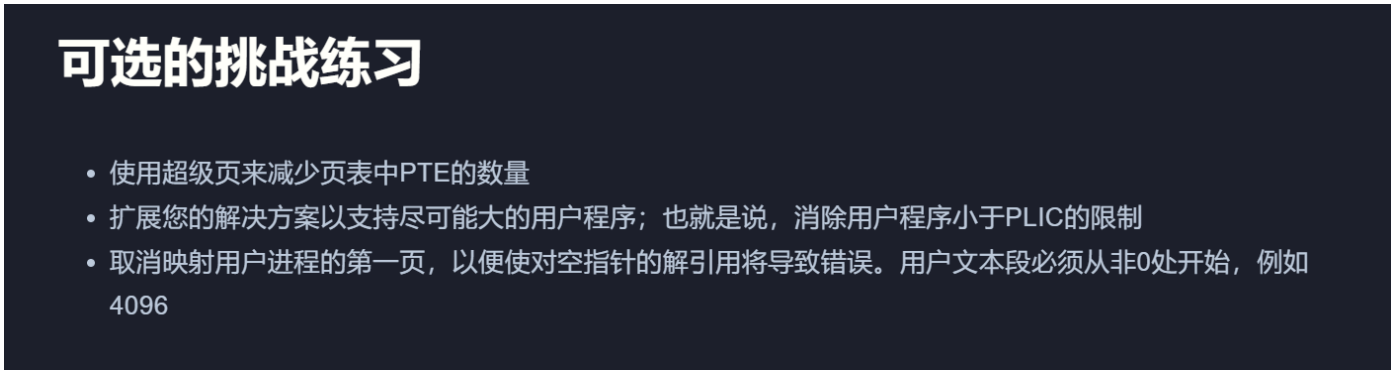
1. 为了简化内核和用户空间之间传递数据的流程，需要建立一个同时涵盖内核页表和用户页表的新页表；
2. 实现内存页面的懒分配和写时复制会增加内存管理的复杂性，需要修改和判断的部分较多；
3. 正确地使用锁，并降低锁争用很困难。

项目拓展

mit6.s081毕竟也有些类似WebServer烂大街的意味了，所以可以在其上做一些扩展来增加项目的独特性与含金量。

挑战练习

每个lab的最后都有挑战练习，可以从这些挑战练习着手扩展项目。如图



参考linux

xv6 毕竟是个教学操作系统，很多功能尚未实现，可以参考linux，在xv6上实现一些新功能，这样既能扩展项目，也能学一些linux源码，获得面试装13利器（手动狗头

用其他语言重写xv6

xv6 源码仅有一万余行，可以尝试用其他语言重写xv6，如[xv6-rust]