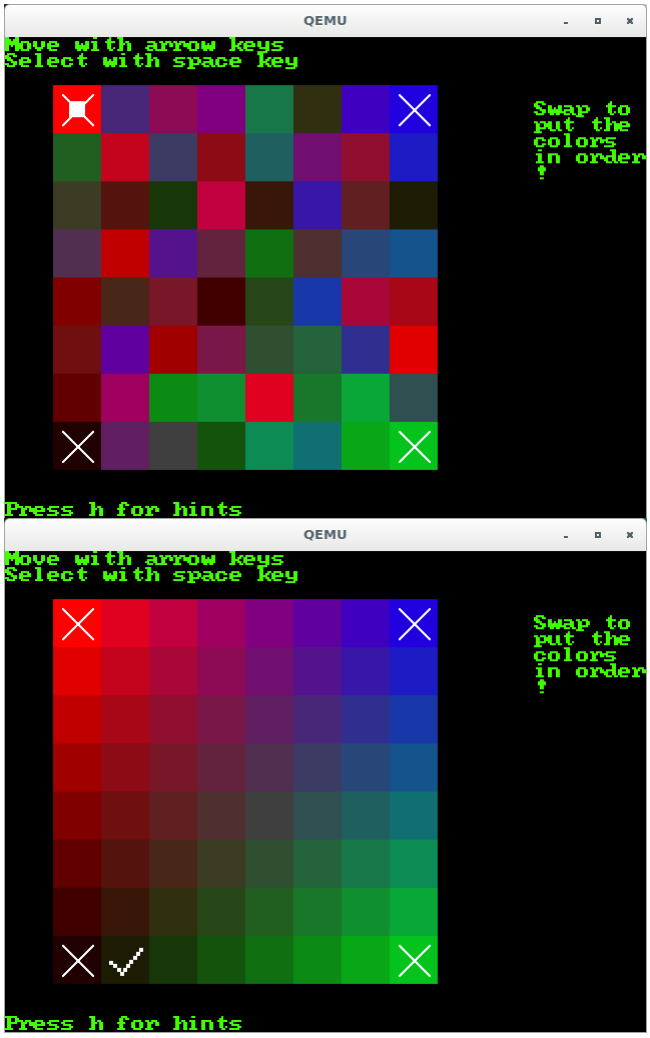
L0 直接运行在硬件上的小游戏(amgame)

游戏截图



游戏内容

游戏会随机*打乱GRID×GRID（在include/game.h中定义）个方块。其中，颜色为渐变色*。
玩家通过上下左右键移动，按下空格选中方块。选中两个方块后会自动交换。玩家要使界面内的颜色重新恢复原本的渐变色。

注1：随机指的是游戏文件中调用了srand(uptime())，基于rand函数的特性及uptime自身的不确定性，可以近似认为游戏达到了随机
注2：渐变色指，在初始的图中，同一行/列的方格，RGB值为三个等差数列

文件结构

```
src
├── draw.c
├── font.c
├── game.c
├── logic.c
└── shape.c
```

我的源代码分为5个文件。
game.c为游戏主循环，只包含主要的函数调用。
shape.c提供绘制基本图形的函数，如画叉、圆、箭头、打钩、光标及填充指定方格等。
font.c提供在图形界面上打印字符的函数，包含ASCII编码下所有字符的点阵，一个打印单个字符的函数（都是静态），打印字符串的函数（非静态）
draw.c提供较复杂的绘图的接口，包括初始化屏幕和在光标选中格打印光标或提示（由print_flag决定）
logic.c提供复杂逻辑的判定，包括游戏初始化、交换两个方格、计算颜色渐变值、获得下一个输入按键、对输入按键进行处理。

设计亮点

- 考虑到如果在主循环中进行判定while(uptime()< next_frame)，会导致按键的延迟，因此，我没有在主循环中使用时间相关的判定。只在少数需要体现出时间特性的部分单独加入了判断（如光标闪烁频率、长按按键的反馈）。这样可以做到，玩家按下按键后，第一时间*处理该按键信息，且长按时表现出若干毫秒处理一次的特性。
- 考虑到native和qemu的分辨率不同（同时还要考虑兼容其它分辨率），include/game.h 中 SIDE 被定义成 16会导致在不同分辨率下的效果差很多。所以我经过计算和尝试，将其改成w/40。使代码能够适应大多数分辨率。同理，draw_str中提供的size参数，我也使用了SIDE以实现自动适应屏幕的效果。但由于整型数精度问题，文字的适应效果有限。
- 考虑到游戏难度较大（作者自己都玩不过），我增加了Hint功能，按下h即可显示提示
- 为了节约空间，没有使用canvas数组，因此在打印时，为了表现出多层的效果，在进行判定时代码量稍多一点。
- 由于裸机本身不提供透明通道，alpha参数被我用来记录其它数值（该方格是否为固定方格）

L1 kalloc

实现原理简述

先将从pm_start到pm_end的堆区分为若干个8 KB大小的页面。
~~其中，使用类似buddy system和区间树的方法进行管理，管理方式如下，~~
（由于线段树方式存在bug，后来改用了链表）
有一个全局free_list链表和每个处理器自己的free list。当kalloc空间时，先上处理器单独的锁，若处理器自己有足够高的空间，则直接分配。
否则，调用global_alloc函数，此时需要上全局锁，然后从全局链表中获取一段内存。考虑到小内存分配

的频繁，如果处理其内部内存不足时，在调用global_alloc时，如果需要的内存小于PG_SIZE，则申请PG_SIZE的内存（否则多次小内存申请也会导致全局锁被阻塞）
kfree的时候，将内存归还给处理器独有的free list，因此，全局的free list可用内存会越来越少

具体规则为
先将需要的空间大小向上取到16的倍数（减少日后合并的碎片数量）
从free list的头开始往后搜寻，找到第一个空间充裕的块（first bit）。如果该块除了提供所需空间之外，还足够提供头部的空间，则在尾部产生一个header，header的size记录本次分配的内存大小，便于kfree释放。如果该块不足以为新的header提供空间，则在维护好链表的指向关系后，直接返回当前header的space
header声明如下

```
struct header{
    struct header *next;
    uintptr_t size;
    struct{}space;//space doesn't take any storage
    //directly return &space
}static free_list[4]={};
```

其中，next指向下一个free list的节点，size用于标记目前区域的大小。
在申请空间时，先将需要的空间向上变为16的倍数，如果该header对应的区域大小大于所需空间，则从header的尾部划出所需空间加sizeof(header)大小的空间，并将尾部转化为一个header，将size置为分配的实际大小以便free函数解析，然后返回尾部的space的地址

设计亮点：

- 在header的定义中，包含一个大小为0的space成员，通过取该成员的地址，可以直接得到分配的空间的起始地址（由于标准中说struct的成员顺序一定按照声明顺序，故在内存上排列方式一定如此），且不会浪费空间
- 每个处理器的free list的第一项是一个Sentinel，用于简化判断逻辑，由于每次free时会考虑该部分是否能和前后的free space互相merge，所以，每个处理器提供一个sentinel，由于分配的时间不同，Sentinel和free list里的free space必然不相连，Sentinel一定不会和要free的space互相merge，极大的简化了判断逻辑
- kalloc类似于一个装饰器，它调用kalloc_real，但可以在调用前后做一些必要的处理，简化某些逻辑(降低多个return维护难度)

测试代码:
（免责声明：本人与其余同学互相交流/分享了测试函数，因而可能在查重时会出现重合，可以通过git记录的时间证明本人并非抄袭）

```
void test(){
    void *space[100];
    int i;
    for(i=0;i<100;++i){
        space[i]=pmm->alloc(rand()%((1<<10)-1));
    }
    for(i=0;i<1000;++i){
        int temp=rand()%10;
        pmm->free(space[temp]);
        space[temp]=pmm->alloc(rand()&((1<<10)-1));
    }
    for(i=0;i<100;++i){
        pmm->free(space[i]);
    }
}
```

该段代码中有一个大小为100的指针数组space，初始化后，space的每个元素指向一个kalloc返回的空间。
然后，每次随机选一个元素，将其空间free，并重新申请一段新的大小随机的空间。
经过若干次free、alloc循环后，再将space里的每个元素free，然后打印free list，如果实现无误，free list应该只包含若干个很大的块，不应该存在碎片或者free list中互相重叠

```
static inline void fill(uint8_t *p,int a,int b,int len){
    for(int i=0;i<len;++i){
        p[i]=(a+=b);
    }
}
static inline void check(uint8_t *p,int a,int b,long long len){
    for(int i=0;i<len;++i){
        report_if(p[i]!=(uint8_t)(a+=b));
    }
}
void memory_test(void *dummy){
    int *a=pmm->alloc(MAXN*sizeof(int)),
        *b=pmm->alloc(MAXN*sizeof(int));
    int cpu_id=_cpu();
    long long len[MAXN];
    void *p[MAXN];
    for(int i=0;i<MAXN;++i){
        len[i]=(1<<((i&15)+5))+rand();
        a[i]=rand()+cpu_id;
        b[i]=rand()+cpu_id;
        p[i]=pmm->alloc(len[i]);
        fill(p[i],a[i],b[i],len[i]);
    }
    for(int j=0;j<MAXM;++j){
        int i=rand()%MAXN;
        check(p[i],a[i],b[i],len[i]);
        pmm->free(p[i]);
        int shift=13+(rand()&7);
        len[i]=rand()+((1<<shift)-1);
        if(i&1){
            len[i]+=1<<12;
        }
        a[i]=rand()+cpu_id;
        b[i]=rand()+cpu_id;
        p[i]=pmm->alloc(len[i]);
        fill(p[i],a[i],b[i],len[i]);
    }
    for(int i=0;i<MAXN;++i)
        pmm->free(p[i]);
    printf("[cpu%d] finish memory test.\n",_cpu());
}
```

类似前一个测试，这个测试增加的是，会向申请的内存里填充一段内容，并且由于fill的性质，内存被写入相同的序列的概率非常低。因此，能通过这个测试基本可以判定内存分配没有重叠

印象深刻的bug:

在nemu中，由于没有访问/读写权限的控制，如果声明了过大的局部变量，由于每个进程只分配了4KB的栈空间，因此，会导致stack overflow，修改到cpuinfo甚至更低地址的重要系统内存,因此，我在先使用小

的局部数组跑了上面的测试后，使用pmm->alloc声明了大内存，并将局部数组改为一个指向申请的空间的指针。

L2 内核多线程 (kthreads)

调度方案为随机选取进程池内可调度的进程。

设计亮点

代码构架

- 将pushcli和popcli打包为库函数intr_close和intr_open
- 将pthread_mutex_[lock|unlock|trylock]打包为库函数，只是简单的atomatic exchange，而操作系统提供的spinlock_t中，解决了reentrance等问题

运行效率

- 考虑到有可能没有可以调度的进程，于是给每个处理器准备了一个idle进程。而idle进程不放在进程池中，而是每个CPU在kmt_context_switch中，如果尝试足够多次都无法获取一个可被执行的进程时，则进入idles[_cpu()]。此设计方式可以提高效率（在有较多进程时，不会执行idle）
- 通过将ncli和intena保存在进程结构体中，实现了允许带锁自陷（带锁wait时自陷），带锁自陷后，在其它处理器恢复时，中断计数不会发生异常

（前提是程序逻辑没有问题，如果带锁自陷会导致死锁，属于测试代码的问题）

- kmt_context_switch是无锁实现，实现为，每个CPU要调度某进程时，使用trylock去锁那个进程的running成员，锁成功即为获得，多个CPU可以同时锁不同的进程而不互相干扰
- ~~累死累活地~~解决了跨核调度。

解决方案为，维护currents和lasts。在执行kmt_context_save前，lasts[_cpu()]存储的是当前CPU上上次运行的进程，currents[_cpu()]存储的是当前CPU上次运行的进程，在kmt_context_save中，将lasts[_cpu()]->running解开即可，大致思路为（某些细节处理较为复杂，详见源码）

```
unlock(lasts[_cpu()])
lasts[_cpu()]=currents[_cpu()]
currents[_cpu()]=kmt_context_switch(args)
```

代码组织架构

我在kernel/src下建了test文件夹，里面存放了针对不同部分（调度、信号量、自旋锁）的单元测试，并且通过相关的宏src/os.c:52-56，快速调用相关的测试。
为了防止过多测试函数影响可用内存大小，我增加了NO_TEST宏。在完成测试后，#define NO_TEST，则相关的测试代码和相关变量不会被生成，可以减少空间的占用

其中，context_test为最弱的测试，用于测试程序是否能够正确调度不使用自旋锁和信号量的进程
memory_test为后来给L1补充的测试
multithread_test为张天昀同学分享的信号量测试
printf_test为针对printf及其开关中断的一些测试
semaphore_test为针对信号量的测试
spin_test为针对自旋锁及其开关中断的一些测试
其中，加粗的两个测试是并发程度最高的测试
正确性本人基本可以保证，我的操作系统能够在同时运行加粗的两个测试的情况下，运行tty，并且向tty1和其它tty写入不会发生任何问题,只是由于某些问题，程序有时候突然卡顿，等待至多十余秒后即可恢复（同时运行**multithread_test, semaphore_test, tty_task**的展示可以git checkout L2并make run4，然后可以切换到tty[2|3|4]输入内容，再切回tty1会发现**multithread_test**确实在正常运行。）
（建议使用kvm，否则由于调度的进程过多，运行会较慢）（因为我能保证正确性，所以一点都不怕kvm233333）
（没有提供命令提示符）
（运行的时候电脑风扇会转啊转）

印象深刻的bug

之前在做的时候，发现task的属性（runable, running, slepping)的读、写都是与、或操作，于是使用lock add和lock or，希望能够略微提高效率（虽然影响不大，至少是难能可贵的尝试），但是发生的问题是，运行一段时间，就会有一些进程永远进入睡眠。
后来，我先花了一段时间重读了代码，删除一部分冗余代码，并且采用单元测试的思路，逐步缩小范围，最终消除了bug（但是并不知道究竟为什么之前的写法会有错）
学到的经验是：

- 单元测试比系统测试能够更好地定位bug所在，并且在有良好架构（如我写的那个宏）的情况下，单元测试的方便性非常之高。
- 遇到问题时，可以采用差量的方式来定位bug：我的信号量测试函数，是一个生产者和一个消费者互相唤醒，但总是运行一段时间之后就会发生死锁和$eip“飞走”的情况，于是，我开始删除一些会影响信号量正确性的代码，比如wait之后不维护信号量的list，也不sleep，发现这样不会发生死锁。于是再针对将进程放入信号量的池的部分的代码进行修改
- 我在kmt_create中加入了ignore_num变量，用来“屏蔽”最早创建的若干进程。设为2时，即屏蔽了tty相关task，使得我可以先对简单的测试进行模拟，同时不需要修改太多处代码（与直接注释相关kmt->create相比）

L3 虚拟文件系统

注：
以下的“与Linux相近”，指的是与我使用的Ubuntu 19.04, GNU bash 5.0.3的表现接近

代码解释

- 实现了三个文件系统
  - devfs
  - blkfs
  - procfs
- 由于一开始对讲义理解有误，实现有一定错误，现在已基本更正，但由于历史遗留原因，讲义中的file_t结构体被命名为vfile_t
- 命名规范
  - 某个fs的某操作op会被命名为fs_op且都为static函数, 如devfs_lookup, blkfs_init
  - 某个fs的fsops,命名为fs_ops, 如devfs_ops
  - 某个fs的inode的inodeops,命名为fs_iops, 如devfs_iops
  - 某个fs的inode（假定每个fs返回的inode只有一种inodeops）的某操作op会被命名为fs_iop且都为static函数,如devfs_iread, blkfs_iwrite。因为后来发现devfs的每个“设备”和devfs本身的根目录的各种函数逻辑相去甚远，而如果都使用特殊判定（见ad02d74fcee0b7b2f3ad917dbbdf2c69a78d047f），会影响代码可读性，因此，有的文件系统有多种inodeops,如devfs_rootiops
  - 而procfs更是如此，我分了三个inodeops，分别是procfs_rootiops, procfs_taskiops, procfs_iops
- 感觉inodeops里面的link和unlink的声明如果不增加一个inode_t*会很麻烦
  比如unlink时，如果调用根目录的inodeops->unlink(path)，那么除非在unlink里面增加递归的路径解析，否则无法调用其它inodes的unlink函数。而这样一来，代码耦合程度会变高。而如果先查看mount table，又难以避免类似/dev/../home/txt这样的路径。综合考虑我其他部分的代码后，我决定给link/unlink各加一个inode_t*参数
- 我抛弃了一个可能很多同学会用的假设
  - 假设：“文件系统的数据可以用指针找到”。也就是说，没有办法构造一个直接指向文件系统本身拥有的inode的指针。文件系统的inode，如blkfs，我需要在fs_init时通过read，将设备上的inode拷贝到内存的某个区域(fs->inodes)，或者某些不存在inode的文件系统（devfs和procfs），需要在fs_init时给所有可能要用的虚拟文件构造一个inode。我把fs_init时构造的inode存放在了fs->inodes里。然后就会涉及到更新的问题，更新需要同时在fs->inodes和设备上写入，因此，对于blkfs，我写了函数add_inode。别的文件系统我都将其作为只读，因此不需要过多考虑

设计亮点

- 实现了颜色改变
  - 可以通过color指令改变当前terminal颜色（四个终端颜色互不影响）
    - color 为恢复默认配色（我调整了默认配色，使其和ubuntu 18.04 gnome默认terminal一样）
    - color fg 为改变前景色
    - color bg 为改变背景色
    - color COLOR COLOR 为改变背景色和前景色（分别为fg和bg）

        - 由于color操作的对象是它的STDOUT，所以在tty1中执行color BG 000000 >/dev/tty2可以改变tty2的颜色配置，和Linux的\033[有些许相似之处
    - 也可以在代码中通过函数tty_set_color和tty_get_color来进行颜色相关操作
    - ls指令会通过isatty(STDOUT)得知输出是否为tty，并将结果保存在tty_mode中。以此决定输出的格式
        - 因此，ls和ls | cat的输出会不同
- 实现了错误提示
    - 在诸如cat /, cd txt这类命令执行错误之后，会打印出类似bash风格的错误提示信息，在shell测试命令中有若干命令会导致错误提示信息弹出
    - 实现方式为，vfs及各具体的文件系统遇到问题后，会将具体问题(如"Not a directory")打到当前进程的err成员中，并且返回值指出发生错误。上层的调用者发现后，会按照格式"当前进程名字: 出错的参数: 错误信息"向文件描述符2输出
    - 比如，cat /的输出为
    - cat /的输出为"shell1: cd: txt:Not a directory"
    - 这样有一个好处，比如mv FILE1 FILE2可以实现成
      cp FILE1 FILE2, unlink FILE1(直接调用mysh_[cp|unlink]）
      防止代码耦合。而由于进程的名字是mv，所以无论是cp还是unlink的过程中出错，提示信息都是"mv: "开头
        - 虽然mv用link和unlink可以提高效率，但是考虑到mv和cp都涉及到DEST为文件夹的情况，因此用这种方式来降低耦合程度
- 命令行解析部分移植了xv6的sh.c，并做了自己的修改（~~主要是增强~~）
    - 支持管道(如echo 123|echo 321或echo 123|cat)
        - 管道使用一个0x100的循环数组，在空/满时会对相应的操作（读/写）进行block，具体实现为_yield。在程序逻辑正确的情况下，是允许带有自旋锁的进程持有锁的情况下在向管道写入或读出时自陷的，由于我L2做的特殊考虑，该类进程即使自陷后从其它CPU上恢复，也不会产生中断计数等问题）
        - 前后的命令是并发执行的。当然，如上所言，如果后面的命令执行到一半，需要从stdin读取内容，而此时前面的命令还没有输出或者输出尚且不足，后面的命令会一直_yield，直到前面的命令产生了输出
        - PS: 其实管道的容量使用一个信号量更好，也可以防止block的进程频繁被无谓的唤醒
    - 支持后台执行指令(为了方便感受，我实现了sleep指令，可以观察sleep 5和sleep 5&的区别)，但此处可能招致bug, 详见“已知的bug一节”
    - 支持输入输出重定向
    - 支持列表指令
        - 如echo 123; echo 321
    - 有无空格都能正常parse, 如
        - echo>>txt
        - echo >> txt
    - xv6不支持" ",我修改了一定量的源码，使得能够支持

        - 也就是说echo Hello, world!会被认为有三个参数(参数包括"echo")
        - 而echo "Hello, world!"会被认为只有两个参数
        - 不支持\'或\"转义，但是由于"和'都可以表示字符串，所以有如下用法
        - echo "I'll kill you"
        - echo " a ' here" ' a " here"
        - 不支持跨行字符串，如果一行内的"或'没有闭合，会有错误提示
    - xv6的parse部分没有内存回收，cmd结构都是alloc之后不free，我通过在runcmd外面多打包一层，然后在递归执行命令的最后加上了递归free解决了这个问题
    - 不支持$VARIABLE
    - 不支持$(cmd)
- 使用非常接近fork-and-exec的方式执行shell命令(参考了xv6)
  准确地说，fork是通过kmt->create模拟的，wait是通过kmt_wait模拟的
    - shell读入一行命令后，创建一个进程"fork-and-run"，并将输入作为参数传给它，然后wait。（不管是不是后台执行）
      然后，fork-and-run先对输入进行parsing,形成复杂的cmd之间的关系（实际上可用下面的BNF范式表示），解析完成后（input内的数据就可以修改了，这也是为什么后台执行时前台能输入其他命令，也是为什么shell一定要wait）通过runcmd，开始递归地执行指令
    - $cmd := exec\_cmd \mid redir\_cmd \mid pipe\_cmd \mid back\_cmd$
    - $redir\_cmd := exec\_cmd > file | exec\_cmd < file | exec\_cmd < file > file$
    - $pipe\_cmd := cmd + "|" + cmd$
    - xv6实现的fancy之处在于，它的“后台”，实际上是fork-and-run再create一个新进程，然后自己exit，被最前台在wait的shell“捕获”
        - 这样做有一个问题，shell、管道命令等命令，在创建新进程后，会亲自执行free. 而此处的子进程创建之后，父进程就无法继续控制它，也就是说无法将其free
        - 我的解决方案为，给进程提供的exit()函数会查看TASK_NOWAIT标志位。如果为0，就将进程标为TASK_ZOMBIE，不再调度，然后立刻_yield()~~长眠~~. 如果该标志位为1，就将自己加入一个池中（自己free自己会导致free过后栈帧处于未分配的区域，对于多核多线程而言十分危险），通过os->on_irq注册函数kmt_context_clean，每次os->trap都会检查池子里是否有进程，若有，且该进程不在running（否则仍可能出现自己free自己，应该在该进程最后一次_yield并退出，该处理器换上了新的进程之后，再对之前的进程做free）则将其free
    - 为了证明我确实支持后台运行，我增加了sleep指令，执行sleep 10| echo Hello!&会在大约十秒后，在当前终端打印出"Hello!\n"，并且在sleep过程中，该终端可以继续执行其余指令
    - （如果指令过长写到"echo Hello!&"区域会导致bug）
      但是执行的命令都是buildin（内建）指令，只实现了一个假的vfs->execve。cd需特殊判定，然后直接执行，不进行fork-and-exec
- 实现了kmt_teardown, kmt_wait
- 实现了简单的EOF

在终端按下Ctrl-d后，终端的进程的下一次read将读到0个byte，但是再之后仍然可以读到数据，和Linux的EOF有些许不同，但相关buildin指令的行为与Linux较为接近
（我想到了实现EOF的方法，可以在某vfile_t被read返回0时，将其关闭，dev->read对于NULL会默认返回0）

- 自己构思了一个文件系统，取名为yls(Yan's Linked-list System)

简单来说，就是把所有信息（文件名或文件内容）以链表形式存储。metadata包括整个磁盘开头的bitmap（表示block占用情况）以及inode list

- 实现了较为完整的cd
  - 在实现cd过程中，我思考了几个问题
  - cd的..需要多次检查文件（夹）是否存在
  - cd可能在不同的文件系统之间跳来跳去
  - cd可能会在某个文件系统的根目录执行..
  - 综上考虑，为了提高效率，我给每个inodeops加入了一个find操作，用来寻找它（作为文件夹）所包含的文件中是否有我们要的文件，为了提高可读性，编码规范为，在fs_ifind的开头声明inode_t* next=NULL;最后一句为return next->op->find(next,path)，且此时path应该做好更新（跳到下一个'/'之前，但是并不略过'/',也即保证path[0]为'/'或'\0'，这是为了解耦合以及安全性检查，比如/test/和/test相比，前者要额外检查test是否是一个文件夹）
  - 这样一来，各个文件系统的find就可以不同，以devfs为例，我实现的devfs仅包含若干个device，不包含其它文件（夹），因此，在devfs中，寻找就是对所有设备的名字做一次strcmp即可。而判断一个inode是不是文件夹，只需要看它是否等于devfs.root就可以了，因此，只需要给devfs.root和devfs.inodes里面的其它inode分配不一样的inodeops就可以了！当然，blkfs的find就复杂得多，涉及到在硬盘上多次read和甚至write
  - 目录名(task_t.pwd)如何更新详见**kernel/src/dir.c:dir_cat**，下简述inode_t*如何更新
  - 比如cd **/dev/../../home/michael/**
    1. 不需要查询mount table，直接从/开始寻找
       先得到/对应的inode,记作cur， 调用cur->ops->find(cur,"**/dev/../../home/michael/**")得到一个inode next, path变为"**/../../home/michael/**"
       next是devfs的根目录，而next自带的inodeops中的find，便是devfs_rootifind
    2. 将cur更新, 继续调用cur->ops->find(cur,"**/../../home/michael/**")得到inode next
    3. 由于上述操作试图跳出devfs的根目录，因此我们会得到cur->fs->root_parent(在vfs_mount时维护), 也即/对应的inode
       现在，next是blkfs[0]的根目录(此时，由于inode的变化，inodeops也发生了对应的变化，自动完成了文件系统间的“跳跃”)
    4. 将cur更新为/对应的inode， 继续调用cur->ops->find(cur,"**/../home/michael/**")这里再次试图跳出一个文件系统(blkfs)，因此，我们会得到cur->fs->root_parent,值得注意的是，这个文件系统是被挂载在/，也就是vfs的根目录，因此在挂载时进行了特殊处理，fs->root_parent还是fs->root，因此，next仍然是cur

5. 将cur更新为/对应的inode， 继续调用cur->ops->find(cur,"**/home/michael/**")，后面一直都在blkfs[0]里面，具体细节不再赘言
    - 如果是相对路径，则第一步不是得到/对应的inode,而是从get_cur()->cur_dir这个inode_t*开始
    - 因此，我实现的cd对于相对路径而言，比Linux更加高效（渐进复杂度层面），但是降低了安全性（父目录被另一个shell删除后，当前进程试图cd ..或者其它操作）
    - 原本的版本是递归调用，也即每个fs_ifind最后一行都是return next->ops->find(next,path,flags)，后来，为了方便vfs控制及在某些层面进行干预（让open和mkdir的不同行为能复用这一流程）且防止爆栈，将其改为只执行一步就返回调用者。递归版本详见（git log 425058bbf5dce5de38740f5aa76d27dd764ba3f2）
- 可以cat /dev/里面的设备
  PS:cat ramdisk会得到人类难以理解的代码、cat非另一个的tty会导致另一个tty有时候不执行指令（被那个tty上运行的shell捕获了。这个都和Linux的行为相似
    - 默认的rd_read和rd_write没有边界检查，因此我修改了rd_read和rd_write，使得cat /dev/ramdisk[01]能够停止
- 增加了一些“虚拟设备”，实现代码均在src/devices/virtual内
    - /dev/null, 对其的读始终返回0，对其的写始终返回nbyte（与Linux相近
    - /dev/zero, 对其的读会读到许多0（通过memset实现，比较优雅），对其写始终返回nbyte（与Linux相近
- 对中断的巧妙处理
    - 我的shell及相关指令都可以在make run1的情况下正常运行（包括以下）
    - sleep 10在返回之前，可以通过alt+数字切换到别的终端，或者向这个终端继续输入内容（但是按下回车后也需要等sleep结束后才会运行，与Linux行为接近）
    - 如果手速够快，cat /dev/ramdisk0在打印还没结束时，可以切换到别的终端（并且输出还是在原终端）
- 修改了terminal的颜色
    - 默认配色参考了ubuntu 18.04 Gnome
- ramdisk.img里面有很多文件
    - 这么多文件显然不可能是手动编码的，我先把我的文件系统构思好，然后写了kernel/initrd/initrd.c，它的输出重定向后可以得到一个ramdisk.img。
    - 然后，在qemu中执行一些vfs的操作(echo, cat, touch, mkdir, rm)，再用gdb的命令 dump memory ramdisk.img &initrd.start &initrd.end将修改后的镜像导出
    - 在vim中，%!xxd阅读二进制编码，确定它符合我的文件系统的规范后（到后期可以跳过这步），再用其替换ramdisk.img

和Linux的不同

- Linux下可以ls 非文件夹的文件，会返回文件名。但是我的ls不允许，并且会返回错误提示

已知的bug

- 使用make run4打开，最开始的屏幕背景色渲染反而比make run1更慢
  - 原因应该是我L2设计不允许一个CPU连续两次调度同一个进程，除非中间通过idle进程过渡。单CPU时，进程不会被抢夺，多CPU时，屏幕渲染的相关进程受到CPU之间的抢夺影响速度。但渲染完毕之后shell运行速度令人满意，因此这个bug就暂且保留
- 内存泄漏
  - **（部分解决）** 在shell中，通过cat /dev/meminfo可以读取当前系统的剩余可用内存数量。在ad02d74fcee0b7b2f3ad917dbbdf2c69a78d047f这个刚完成的版本，每执行一条指令都会有5000字节左右的内存发生泄漏。由于为了使系统的适应性更强，我在很多地方用了动态内存，并且xv6的sh.c中，命令行解析部分，大量使用动态内存，且**从来不free**
  - 目前我解决了部分指令的内存泄漏，罗列如下
    - 简单的[cmd] [args]格式指令
    - [cmd] ; [cmd] (listcmd)
    - [cmd] [args] < [file] (redircmd)
  - 尚未解决的指令
    - [cmd] | [cmd] (pipecmd)内存泄漏352字节（管道没有做到自行回收）
    - [cmd] & (backcmd)内存泄漏16字节（主要在于原本xv6没有回收那个后台运行的子进程）
- shell输入共享
  - 这个应该属于xv6的parse部分的bug。在执行后台命令时，parse部分并没有将input备份，因此，此时input是可能被后续的指令写的，如sleep 10 | echo Hello &然后快速输入echo 12345678901234567890。
  - 我后来将pipe两端的指令改为同步执行（由于我的管道设计有阻塞性，能够自动让相关进程yield），因此这个bug较难复现，修改难度也比较大，就没有去修了
- 框架代码的bug
  - 有时候Alt或Ctrl的松开不会被检测到，导致后续输入什么都不会显示，只要再按一次相应按键即可恢复

shell的测试方法/使用手册

建议使用make run1和make run4分别测试
指令最后有(x)表示该指令不合法，是为了展示错误提示
前者是为了测试在单CPU的情况下，指令的执行并不会独占CPU，仍然能够在各个tty之间切换
后者是为了测试多个CPU的情况下，我的操作系统能够安全稳定地并发执行

```
echo Hello,      world!
```

期望输出Hello, world!\n（因为中间多个空格应该被认为是参数的分隔符，因此应该是echo接受了两个参数，空格是echo 自己加的，因此只应该有一个空格）

```
echo "Hello,     world!"
echo 'Hello,     world!'
echo "I'll always love you"
echo " a ' here " and ' a " here '
echo "Not closed will receieve error message(x)
```

展示"和'的解析

```
ls
ls /
ls .
ls txt(x)
```

展示ls对于多种不同输入的考虑

```
ls /dev
echo Hello > /dev/null
cat /dev/null
cat /dev/ramdisk0
ls /proc
ls /proc/1
cat /proc/1/pwd
cat /proc/2/name
cat /proc/meminfo
```

展示devfs和procfs实现的具体内容（第四条会出现看似乱码内容）

```
ls /dev/ramdisk0(x)
ls /proc/meminfo(x)
cat /proc/99(x)
ls /proc/2/pwd(x)
```

展示devfs和procfs的异常处理以及相关提示

```
cat txt
cat < txt
cat
```

展示cat及EOF（输入Ctrl-D即可终止第三条命令）

```
cat test | cat(x)
cat txt | cat
echo 123 | cat
test | cat
cat /dev/ramdisk0 | cat
cat /dev/ramdisk0 | wc
```

管道

```
cat > new_file
echo 123 > new_file
```

展示重定向及vfs_open带有O_CREAT选项的表现

```
sleep 10
sleep 10 &
```

展示……睡眠2333
（其实是展示单核的时候可以在sleep时切换到别的终端,或者让tty1执行sleep 30，然后跑到tty2执行cat /dev/tty1然后跑回tty1输入东西）

```
touch new_file
cat new_file
```

展示touch（其实没啥用，因为按照O_CREAT打开的话，会自动创建）

```
cd /
cd txt(x)
cd /dev/tty1(x)
cd /proc/meminfo(x)
cd /dev/../proc/../home
cd /../
cd /..////home//michael/
cd ..
```

展示cd（特别是多种特殊情况的处理）

- 如一次cd在多个文件系统中转换
- 多个/相连
- 在根目录..
- 向非文件夹跳转

还有关于EOF和文件开启的处理

```
cat > txt
Ctrl - D
cat txt
```

txt会变为空文件

```
echo 123 > new_file
echo 321 >> new_file
```

简单的O_APPEND

```
link test1 test2
link test2 test3
link test3 test1(x)
unlink test1
rm test1 (x) test2 test3
```

展示link, unlink, rm以及相关错误提示

```
mv txt txt.txt
mv txt.txt home////michael/
ls home/michael
cp home/michael/txt.txt txt
cp txt cp_txt
echo ADD>>txt
cat txt
cat cp_txt
mv txt cp_txt
ls
```

展示mv和cp（倒数第二条命令为覆盖）
(mv和cp不能用于文件夹)

```
mkdir home/another
rmdir home/another
mkdir txt/new_dir(x)
rmdir txt(x)
```

展示mkdir和rmdir(我的rmdir没有空文件夹检查，相当于默认-f)