《操作系统课程设计》任务书

目录

一．环境搭建

1．希冀平台

希冀平台提供了基于 Web 的云环境，相关的实验必须在该环境中测试、提交。该云环境操作受限，只能利用其预安装的各种工具，无法使用集成开发环境。

代码编辑

只能使用字符界面下的编辑器 Vim，你需要具备基础的 Vim 使用能力。Vim 包含三种工作模式，要点如下：
1）命令模式：最初的模式，可执行撤销、重做、复制、剪切、粘贴之类的操作，命令自学。
2）输入模式：即编辑模式，随着光标录入、删除。
3）底部命令行模式：用于保存、退出之类，命令自学。
上述模式之间的切换方法自学。
特殊：如何选中一片复制粘贴？命令模式下，光标移至目标区起点，按 v 进入视图模式，移动光标完成选取，然后 d 剪切，y 复制，p 粘贴。

程序编译

云环境中提供了 xv6 的源程序，并提供 Makefile 指导程序的编译。如果我们增、删了程序文件，就需要对 Makefile 做相应调整。
Makefile 本身比较复杂，有兴趣的人可以自学，本实验要求你能粗略推测 Makefile 中各部分的作用，并可根据增、删程序文件的情况修改 Makefile。

程序调试

由于没有集成开发环境，程序的调试只能在字符界面下开展，直接使用 GDB 命令。希冀平台的作业中提供了一些讲解，更系统的学习需要借助互联网自学。

与本地电脑的文件传输

云环境允许我们从中下载文件，或上传本地的文件，页面上提供了相关操作的菜单。
云环境并未对文件传输开放整个文件系统，只是开放了/mnt/cgshare 文件夹，且不支持文件夹的下载。因此，文件传输必须借助/mnt/cgshare 文件夹中转，必要时须对文件打包压

缩。云环境提供了 tar 命令，使用方法如下：
打包后压缩为 gz 格式：tar -czvf archive.tar.gz file1 file2 directory
解压缩 gz 格式后解包：tar -xzvf archive.tar.gz
本地端，Windows、MacOS、Linux 都提供 tar 命令。

2．本地环境搭建

以 Ubuntu 22.04 为例，搭建过程如下（https://zhuanlan.zhihu.com/p/501901665）。

准备 Linux 环境

不再赘述

下载 xv6 源代码

https://github.com/mit-pdos/xv6-riscv
git clone https://github.com/mit-pdos/xv6-riscv.git

安装编译工具链

需要的工具如下：
- make
- gcc
- perl
- qemu-system-riscv64
- riscv64-unknown-elf-gcc 或者 riscv64-linux-gnu-gcc
- riscv64-unknown-elf- 或者 riscv64-linux-gnu-
    - ld
    - objcopy
    - objdump
    - ...（这套东西一般叫 binutils-riscv64-.....）
- gdb-multiarch

以 ubuntu 为例：
```
sudo apt install binutils-riscv64-linux-gnu
sudo apt install gcc-riscv64-linux-gnu
sudo apt install gdb-multiarch
```

`sudo apt install qemu-system-misc opensbi u-boot-qemu qemu-utils`
实际上，你只需要跟着后续 `make qemu` 命令的报错，需要什么装什么就行了。

编译运行

在 xv6 源代码文件夹中（与 Makefile 同一个目录下）执行 `make qemu`，正常的话就进入了 xv6。然后退出 qemu：
`first press Ctrl + A (A is just key a, not the alt key),`
`then release the keys,`
`afterwards press X.`

配置 GDB 调试

在 xv6-riscv 目录下，执行 `make qemu-gdb`，进程会阻塞。

```
delta@ubuntukvm ~/xv6-riscv (riscv)> make qemu-gdb
*** Now run 'gdb' in another window.
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 -S -gdb tcp::26000
```

另开一个终端，在 xv6-riscv 目录下，执行 `gdb-multiarch kernel/kernel`
可能出现以下问题：

```
delta@ubuntukvm ~/xv6-riscv (riscv)> gdb-multiarch kernel/kernel
GNU gdb (Ubuntu 9.2-0ubuntu1~20.04.1) 9.2
Copyright (C) 2020 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.
Type "show copying" and "show warranty" for details.
This GDB was configured as "x86_64-linux-gnu".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>.
Find the GDB manual and other documentation resources online at:
    <http://www.gnu.org/software/gdb/documentation/>.

For help, type "help".
Type "apropos word" to search for commands related to "word"...
Reading symbols from kernel/kernel...
warning: File "/home/delta/xv6-riscv/.gdbinit" auto-loading has been declined by your `auto-load safe-path' set
 to "$debugdir:$datadir/auto-load".
To enable execution of this file add
        add-auto-load-safe-path /home/delta/xv6-riscv/.gdbinit
line to your configuration file "/home/delta/.gdbinit".
To completely disable this security protection add
        set auto-load safe-path /
line to your configuration file "/home/delta/.gdbinit".
For more information about this security protection see the
"Auto-loading safe path" section in the GDB manual.  E.g., run from the shell:
        info "(gdb)Auto-loading safe path"
(gdb)
```

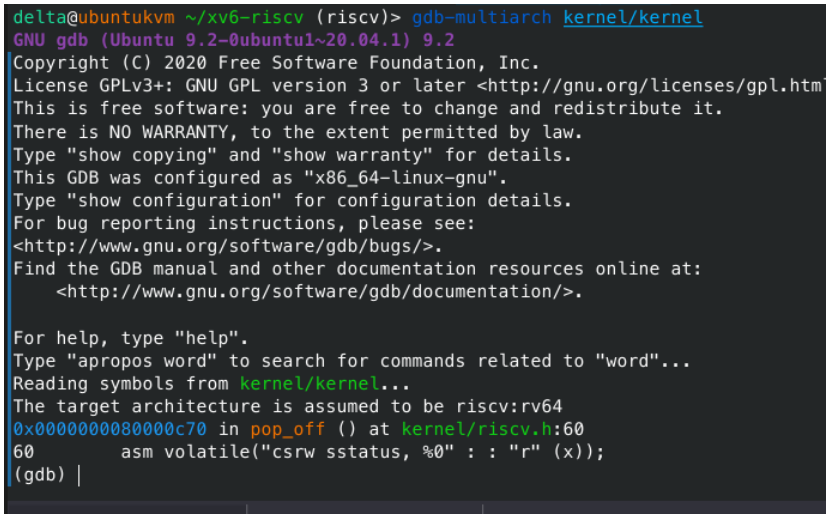
`make qemu-gdb` 在当前目录生成了`.gdbinit`，具体见 Makefile 第 166 行。gdb 会默认首先执行当前目录下的`.gdbinit`，上图中这一步被阻止了。

`warning: File "/home/username/xv6-riscv/.gdbinit" auto-loading ...`
依照提示操作即可，将

```
add-auto-load-safe-path /home/username/xv6-riscv/.gdbinit
```

这一行加到/home/username/.gdbinit 这个文件里。先退出 GDB（快捷键 ctrl+d），然后执行 echo "add-auto-load-safe-path /home/username/xv6-riscv/.gdbinit" >> "/home/username/.gdbinit"，或者 Vim 手动编辑也行。

之后再次运行 gdb-multiarch kernel/kernel，应该一切正常了。



安装配置 VSCode

先安装 VSCode，并在其中安装“C/C++”插件（微软发布）。

在 xv6-riscv 目录下，创建.vscode 文件夹，并在其中创建以下两个文件：

launch.json

```
// xv6-riscv/.vscode/launch.json
{
   "version": "0.2.0",
   "configurations": [
      {
         "name": "xv6debug",
         "type": "cppdbg",
         "request": "launch",
         "program": "${workspaceFolder}/kernel/kernel",
         "stopAtEntry": true,
         "cwd": "${workspaceFolder}",
         "miDebuggerServerAddress": "127.0.0.1:26000",
         "miDebuggerPath": "/usr/bin/gdb-multiarch",
         "MIMode": "gdb",
         "preLaunchTask": "xv6build"
```

```
      }
   ]
}
```

其中 miDebuggerServerAddress 见.gdbinit 中：target remote xxxx:xx

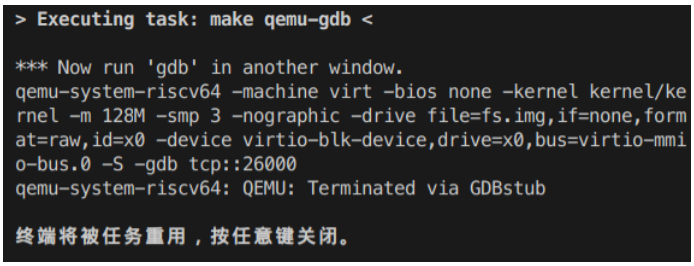
tasks.json

```
// xv6-riscv/.vscode/tasks.json
{
   "version": "2.0.0",
   "tasks": [
      {
         "label": "xv6build",
         "type": "shell",
         "isBackground": true,
         "command": "make qemu-gdb",
         "problemMatcher": [
            {
               "pattern": [
                  {
                     "regexp": ".",
                     "file": 1,
                     "location": 2,
                     "message": 3
                  }
               ],
               "background": {
                  "beginsPattern":  ".*Now  run  'gdb'  in  another
window.",
                  "endsPattern": "."
               }
            }
         ]
      }
   ]
}
```

其中 beginsPattern 要对应编译成功后，echo 的内容，此处对应 Makefile 第 170 行。

之后即可按 F5 开始调试。可能会发现报错如下：

或者



这是因为.gdbinit 中有 target remote 127.0.0.1:26000，这个文件会被 GDB 最先执行一遍，此外，我们在 launch.json 中指定了

"miDebuggerServerAddress": "127.0.0.1:26000"

同一个 remote address 被配置了两次。只需要把.gdbinit 中的注释掉即可：

```
...
@REM target remote 127.0.0.1:26000
...
```

其中@REM 就是.gdbinit 的注释符号。将来如果你要使用命令行 GDB，记得再改回来。至此配置完成，可以在 VSCode 中设置断点，单步调试了。

二．希冀平台 xv6 实验

1．xv6 入门

本实验通过一系列操作让学生熟悉xv6-RISC-V的基本环境。并通过增加几个xv6上的应用，使学生熟悉 xv6 上的系统调用原理和使用。

实验步骤

1）熟悉 xv6-riscv 编译与运行环境
2）实验准备
3）对 xv6-riscv 内核进行调试
4）为 xv6-riscv 系统增加 sleep 命令
5）为 xv6-riscv 系统增加 find 命令
6）为 xv6-riscv 系统增加 xargs 命令
7）为 xv6-riscv 系统增加带首/尾星号通配符的 findx 命令

重点理解

user 文件夹下各文件的功能、依赖关系

2．系统调用

本实验通过让学生增加一个系统调用，熟悉 XV6 Riscv 操作系统系统调用的原理与实现。

实验步骤

1）实验准备
2）在内核中为 xv6-riscv 增加一个系统调用接口
3）为 xv6-riscv 增加系统调用的实现
4）为 xv6-riscv 增加使用以上系统调用的命令
5）测试新增的系统调用

3．内存分配

本实验要求使用伙伴内存管理系统来分配和释放内核中的文件描述结构，这样的话，xv6-riscv 就能拥有超过 NFILE 限制的打开文件描述符。更进一步地要求，在用户内存管理中，使用延后的方法分配内存。

xv6 只有一个页分配器，不能动态分配小于一个页的对象。为了绕过这个限制，xv6 将小于一个页的对象声明为静态的。例如，xv6 声明了一个文件结构体数组（file structs），一个进程结构体数组（proc structures）等。因此，系统可以打开的文件数量受到静态声明的文件数组大小的限制，这个数组包含 NFILE 个条目（参见 kernel/file.c 和 kernel/param.h）。

解决方案是采用 buddy 分配器,我们已经将其添加到 xv6 中,对应的文件是 kernel/buddy.c 和 kernel/list.c。

实验步骤

1）实验准备
2）伙伴算法内存分配（一）
3）伙伴算法内存分配（二）
4）内存页的延迟分配（一）
5）内存页的延迟分配（二）
6）内存页的延迟分配（三）

重点理解

1）用户内存空间的分布，proc 结构体 sz 成员的含义，sbrk()系统调用的功能
2）发生页错误时，scause 寄存器值为 13 或 15，stval 寄存器存放引发错误的虚拟地址

4．fork 调用的写时拷贝

本实验为 fork 调用实现内存的写时拷贝。当系统执行 fork 调用时，如果将父进程的所有用户态内存拷贝到子进程空间，不但费时，也有可能造成内存消耗过多。本实验在要求在内核中实现内存管理的写时拷贝，即 fork 时只增加对父进程用户态内存的引用，只有当对内存写时，才进行拷贝。本实验自带的 cowtest 程序首先分配大量内存，然后进行 fork 调用。如果不进行写时拷贝(COW)优化处理，则执行 cowtest 不会通过。当实验者在内存管理系统中实现了写时拷贝后，cowtest 测试程序可以通过。

总体思路

fork 调用写时拷贝的目的是延迟分配与拷贝子进程的物理内存页直到确实需要拷贝的时候。具有 COW 功能的 fork 调用在 fork 时仅为子进程创建一个页表，此页表中用户态内存的 PTE 项指向父进程的物理内存。同时将父进程和子进程的用户态 PTE 项都标记为不可写。
当父进程/子进程试图写其中的 COW 页时，CPU 会触发一个页错误。内核的页错误处理机制检测到这个问题，为引发错误的进程分配物理内存，将原页面的内容拷贝到新的页面，并且将相关 PTE 指向这个新的页，此时将 PTE 设置为可写。当页错误处理机制返回时，用户态进程就可以对它的拷贝进行写了。
具有 COW 功能的 fork 的让实现用户态内存的物理页释放机制变得复杂。一个物理页可能被多个进程的页表所引用，只有当最后一个引用消失时，才能真正释放该页。

实验步骤

1）实验准备
2）初步尝试执行 cowtest
3）调整 fork 时用户态内存拷贝机制
4）增加物理页面的引用计数功能
5）实现页错误捕获/修正机制
6）在 copyout 函数中处理 COW

重点理解

引用计数信息放在哪里？设置一全局数组，每元素对应一个物理页面的引用计数。内核内存空间布局变为：内核代码 - 内核全局数据 - end - 引用计数数组 - kalloc 管理的区域

5．文件系统

本实验的目的是通过让实验者修改 XV6-Riscv 内核增加对大文件的支持，以及增加对符号链接文件的支持，熟悉和掌握文件块的索引和读写对于文件系统的重要地位，熟悉和掌握文件系统按照路径名寻找文件的过程。

实现大文件支持总体思路

本步骤及一下要求实验者增加 xv6-riscv 系统支持文件的最大尺寸。当前，xv6 文件的最大文件尺寸限制在 268 块，或 268*BSIZE 字节（xv6 中一个 BSIZE 大小为 1024 字节）。

以上的限制来自 xv6 inode 包含 12 个直接块和一个“single-indirect”块编号，该编号指向一个拥有 256 个更多块编号的块，所以就是 12+256=268 个块。

系统带的 bigfile 命令尽可能地生成最大的文件，并且报告这个文件的大小：

```
$ bigfile
...
wrote 268 blocks
bigfile: file is too small
```

以上测试失败了，因为大文件期望能够生成一个具有 65803 个块的文件，但是未经修改的 xv6 内核直支持最多 268 个块的文件。

需要实验者更改 xv6-riscv 文件系统代码以在每个 inode 上支持一个“doubly-indirect”块，其中包含 256 个“singly-indirect”块的地址，而每个“singly-indirect”块包含最多 256 个数据块的地址。所以修改后，一个文件就能包含 65803 个块，或 256*256+256+11 个块。

以上的 11 而不是 12，是因为我们牺牲了一个直接块编号用作“double-indirect”块。

符号链接技术准备

本步骤要求实验者在 xv6-riscv 中增加符号链接支持。符号链接（或软链接），指的是一个通过路径名链接的文件。

当一个符号链接被打开时，内核跟随链接到被指向的文件。符号链接类似硬链接，但是硬链接仅限于指向同一磁盘的文件，而符号链接可以跨磁盘设备。

虽然 xv6-riscv 不支持多磁盘设备，但是实验者可以通过实现这个系统调用来了解路径名查找是如何工作的。

实验步骤

1）实验准备

2）对 bmap 函数进行修改

3）符号链接实现

4）文件系统同步与并发调试

重点理解

磁盘 inode 的格式在 fs.h 的 struct dinode 定义。

实验者可能对 NDIRECT，NINDRIECT，MAXFILE 等常量以及 struct dinode 的 addrs[]元素感兴趣，并希望了解它们。

用来查询一个文件的数据的代码在 fs.c 的 bmap 函数中实现。实验者可以学习这个函数是如何实现文件读写的。
当进行写操作时，bmap 函数会按需分配新的块来存放写的内容，也会分配一个间接块来存放块地址。
bmap 函数处理两种块编号。bn 参数的意思是一个：“逻辑块号”，一个文件中的块号，与文件的起始相对应。而在 ip->addrs[]中的块编号以及传送给 bread 的参数是磁盘块编号。
可以把 bamp 看成把逻辑块号对磁盘块号的映射。

6．mmap 系统调用

本实验要求实验者通过修改 XV6-Riscv 内核增加对 mmap 系统调用以及 munmap 系统调用的支持。
mmap 以及 munmap 系统调用让 UNIX 程序可以对它们的地址空间进行精细的控制。这些系统调用可以：
1）让进程共享内存
2）把文件映射到进程地址空间
3）为用户态内存缺页异常方案的一部分（例如垃圾回收算法）发挥作用
实验者不需要实现所有 mmap 和 munmap 调用的功能，只需要让 maptest 程序能够工作即可系统，其他的功能无需实现。
本实验主要让实验者关注内存映射文件（memory-mapped file）。

实验步骤

1）实验准备
2）查看 Unix/Linux 的 mmap 系统调用手册
3）查看 Unix/Linux 的 munmap 系统调用手册
4）mmap 和 munmap 实现

7．锁

本实验可以让实验者通过重新设计代码来提高并发性。多核机器的一个通用的问题是高的锁冲突。提高并发性包括改变数据结构和锁策略来减少冲突。本实验为内存分配机制和块缓冲机制设计锁机制。

实验步骤

1）实验准备
2）学习 XV6 指导书第 6 章：锁
3）内存分配的锁冲突测试
4）为内存分配消除锁冲突
5）磁盘块缓存的锁冲突测试
6）减轻磁盘块缓存的锁冲突

8．网络

本实验需要实验者为 xv6-riscv 操作系统内核完善网卡驱动，并且为 xv6-riscv 操作系统添加 UDP 网络套接字支持。

知识背景准备

可以针对 xv6 教学指导书的第四章：Traps and system calls，第五章：Interrupts and device drivers 以及第 8.18 节：File descriptor layer 进行学习。还可阅读 the lecture notes on networking 中对网络相关知识的介绍。
本实验会用到一个名为 E1000 虚拟网络设备来处理通讯。对于 xv6 操作系统（以及我们添加的驱动而言），E1000 像是联到以太局域网（LAN）的一个真实设备。
事实上，实验者进行对话的是由 qemu 模拟的一个设备，其联到一个同样是 qemu 模拟的局域网上。在这个局域网上，xv6（来宾机）具有 ip 地址 10.0.2.15。而对端的 ip 地址是 10.0.2.2。当 xv6 使用 E1000 来向 10.0.2.2 发送报文时，qemu 把该报文递交给在宿主机上运行的应用。
本实验使用 qemu user mode 网络栈，其不需要管理员权限。运行 qemu 以及模拟网络的命令已写在更新了的 Makefile 中。
本实验配置了 QEMU 对网络栈的报文捕获。捕获的报文在 xv6-riscv 的 packets.pcap 文件。实验者可以通过 tcpdump 命令来查看。
最后，本实验的初始代码提供了对以太网，IP，UPD，ARP 等报文头的解析和处理。实验者能够在 kernel/net.c 和 kernel/net.h 文件中看到。
本实验使用了一个用于管理和存储报文的抽象数据结构 mbuf。

实验步骤

1）实验准备
2）网络设备驱动
3）网络设备驱动注意事项

4）网络套接字
5）网络套接字实现的注意事项
6）测试与测评

9．简单的 xv6 shell

实验者的任务是为 xv6 写一个简单的 shell。该 shell 可以运行带参数的命令，处理输入和输出重定向，并且创建双端管道。实验者实现的 shell 的行为与 xv6 的 shell（sh）类似，例如一下这些简单的命令：

```
echo hello there
echo something > file.txt
ls | grep READ
grep lion < data.txt | wc > count
echo echo hello | nsh
find . b | xargs grep hello
```

实验步骤

1）实验准备
2）简单 shell 的具体实现
3）简单 shell 实现的注意事项

10．用户态线程和闹钟

本实验通过实现用户态线程以及闹钟让实验者熟悉在上下文切换和系统调用过程中，线程状态是如何保留和恢复的。同时让实验者熟悉将类中断的事件递交程序的方法。
本实现的任务是通过补充代码，完成线程的状态保存（主要是寄存器等信息），最终完成用户态线程的创建以及切换。通过补充代码，增加设置闹钟的系统调用，以完成闹钟的按时唤醒功能。

实验步骤

1）实验准备
2）RISC-V 汇编语言预热
3）用户态线程切换
4）用户态线程切换实现注意事项

5）设计闹钟

6）闹钟的实现说明

7）测试程序说明 test0：对闹钟处理机制的调用

8）测试程序说明 test1：从被中断的代码中恢复

9）内核其他部分测评

重点理解

定时器（及其它通过内核发起的）回调函数的调用机制和返回机制

三．xv6 的裁剪与拼接

本实验包含若干阶段性任务，每个阶段要求同学们从 xv6 的源代码中裁剪出有用的部分添加进来，使我们的系统逐阶段的扩充、丰富。
各阶段的裁剪、拼接过程中，xv6 原有的代码结构（即哪些变量、函数在哪个文件中，哪个文件在哪个文件夹中）不应改变，以方便验收检查。

1．启动至 S 模式

本阶段目标：让系统启动，经过 M 模式的必要初始化，切换到 S 模式，然后每个 CPU 输出一句提示语，之后所有 CPU 陷入一个死循环。

主线

RiscV 启动后自动进入 M 模式，在 xv6 中，CPU 首先执行 entry.S，然后调用 start.c 中的 start()，start()经过一番设置，最后退出 M 模式进入 S 模式，并跳转到 main.c 中的 main()。
我们的目标系统也遵循这个主线，不过进入 main()后只需要输出提示语即可，不必开展进一步的系统启动。main()如下：

```
void main()
{
   if(cpuid() == 0) {
      // 此处为调用 printf()执行必要的初始化
      printf("cpu %d is booting!\n", cpuid());
      __sync_synchronize();
      started = 1;
   } else {
      while(started == 0);
      __sync_synchronize();
      printf("cpu %d is booting!\n", cpuid());
   }
   while (1);
}
```

kernel.ld

内核连接的指导，须结合 xv6 源程序理解。重点理解以下内容：
1）ENTRY(_entry)与 entry.S 中的_entry
2）. =
3）PROVIDE(etext = .)与 vm.c 中的 extern char etext[];
4）PROVIDE(end = .)与 kalloc.c 中的 extern char end[];
我们的程序可沿用 xv6 的 kernel.ld，但由于本阶段未涉及跳板页，应把跳板页相关的内容（从_trampoline 行到 ASSERT 行）注释掉。

start.c

涉及 RiscV 的各种初始化，涉及 riscv.h。

main.c

为了向屏幕输出文字，涉及 printf.c。

printf.c

xv6 中，printf()向控制台输出，涉及 console.c，而控制台又涉及文件设备、输入输出、缓冲区、同步异步等，最终涉及串口操作 uart.c。
本阶段任务中 printf()不涉及复杂的控制台，仅通过串口做简单的同步输出，直接引用 uart.c。
启动阶段每个 CPU 都要执行 printf()输出，容易导致输出内容交错，为此须引入自旋锁，涉及 spinlock.h、spinlock.c。
spinlock 相关程序涉及 proc.c 中的 cpuid()和 mycpu()。

关于串口 UART 的工作机制
1）CPU 与串口之间通过高速系统总线连接，串口与外界（显示器或键盘）则采用逐 bit 串行传输，二者速度差异大。
2）接收时，串口逐位接收，每收齐一个字符，会向 CPU 发起中断。
3）发送时，CPU 每发送一个字符到串口，须等待一阵，确保串口完成发送，才可向串口送出下一个字符。
4）串口完成一个字符的发送，其状态寄存器的“空闲”位会变成 1，同时串口还会向 CPU 发起中断。

关于 uart.c 中同步传输的解释
1）接收无所谓同步异步，在中断服务程序中处理，uartintr()的前半部分即执行接收。

2）uartputc_sync()负责同步的发送，即 CPU 不断读取串口的状态寄存器并检测其“空闲”位，确认空闲了再发送字符。
3）xv6 的内核中的 printf()最终使用的是同步发送。
也就是说，同步发送期间会存在忙等，但函数返回时，字符一定交到串口了。

关于 uart.c 中异步传输的解释
1）异步发送被用户态的 printf()使用，本阶段用不到，以下提到的内容都可以删掉。
2）程序中准备了全局的缓冲区 uart_tx_buf[]，uartputc()用于启动异步发送，实际上只是把待发送字符放入了缓冲区，并调用 uartstart()，如果程序想要发送 10 个字符，就会调用 10 次 uartputc()。
3）uartstart()检查串口是否空闲（非循环检测，仅检查一次），不空闲就啥也不做，如果空闲就把缓冲区的下一个字符送到串口。
4）串口的中断可能因收到字符，也可能因发送字符完成，处理函数 uartintr()并不清楚原因，所以分成上下两半，前半部分试着读字符，后半部分调用 uartstart()试着发字符。
也就是说，异步发送期间不存在忙等，但 uartputc()只是确保“发送业务被受理”，真正的发送是在中断的驱使下推进的。

关于 console
console 是夹在内核和驱动程序（即 uart.c）之间的模块，比较复杂，向上要对接内核的 printf()和用户的系统调用，中间要跟文件系统对接以营造“一切皆文件”的局面，向下要对接 uart.c 中的同步、异步发送和中断处理。
本阶段没有用户态程序，没有文件系统，所以不应保留 console.c。
1）内核的 printf()应直接调用 uart.c 的同步发送。
2）串口的中断处理函数 uartintr()接收到字符后应直接调用同步发送，把收到的字符显示到屏幕上。

Makefile

沿用 xv6 的 Makefile，但去掉所有用户态的内容，去掉所有文件系统和磁盘相关的内容，修改 kernel 目录下的 o 文件列表。
注意：kernel.ld 规定程序从 0x80000000 开始，RiscV 要求入口程序出现在 0x80000000，所以必须确保 entry.S 的程序被放在程序开头。为此，Makefile 中务必将$K/entry.o 排在最前面。

其它

各种 typedef、常量、函数声明等，涉及 defs.h、param.h、types.h、memlayout.h。

2．物理内存管理及内核页表

本阶段目标：在上一阶段的基础上，实现对物理内存的管理，对内核构建页表，并让所有 CPU 启用页表。本阶段不产生额外的输出，但启用页表后，只要程序执行不错乱，仍可把提示语输出到屏幕上，就基本说明本阶段任务正确完成。

物理内存管理

将空闲的物理内存按页组织成链表，并提供分配、释放功能，可完全沿用 xv6 的 kalloc.c。

内核页表

构建内核页表，涉及 vm.c。
xv6 中，内核页表映射了串口、磁盘、中断管理器、内核程序、内核数据、空闲物理内存、跳板页、所有进程的内核栈。
本阶段任务中不涉及磁盘、跳板页、进程内核栈，只需要映射串口、中断管理器（本阶段用不到，但第三阶段任务马上就要用到，顺手解决）、内核程序、内核数据、空闲物理内存。
创建页表需要分配物理页面后清 0，涉及 string.c 中的 memset()。

main.c

遵循 xv6 的做法，上述空闲物理页面链表的构建（初始化）、内核页表的构建、启用，在 main() 中调用。

其它

各种 typedef、常量、函数声明等，涉及 defs.h、param.h、types.h、memlayout.h。
各种与页表有关的计算、操作的宏，涉及 riscv.h。

3．内核态下的中断处理

本阶段目标：在上一阶段的基础上，增加对定时器和串口的中断处理。
定时器按照 xv6 的方式配置，即周期性的产生中断，两次中断的间隔时间设置为 1000000 个时钟周期（大约 0.1 秒）。出于测试目的，本阶段要求定时器每经过 30 次中断，就向屏幕（串口）输出一个“T”字符。
串口的中断实际上是由键盘触发，本阶段要求处理该中断，获取键盘发来的字符，并立即将

该字符输出到屏幕（串口）上。
从效果上看，系统完成启动后，一方面，每隔大约 3 秒，屏幕上出现一个“T”字符，另一方面，用户敲击键盘，即可在屏幕上看到相应的字符。

RiscV 相关基础知识

中断/异常的处理，第一要务是弄清楚原因。RiscV 提供了 scause 寄存器，读取该寄存器即可了解本次中断/异常被触发的原因。以下展示了 scause 寄存器的格式和编码含义。

Interrupt	Exception Code	Description
1	0	*Reserved*
1	1	Supervisor software interrupt
1	2–4	*Reserved*
1	5	Supervisor timer interrupt
1	6–8	*Reserved*
1	9	Supervisor external interrupt
1	10–15	*Reserved*
1	≥16	*Designated for platform use*
0	0	Instruction address misaligned
0	1	Instruction access fault
0	2	Illegal instruction
0	3	Breakpoint
0	4	Load address misaligned
0	5	Load access fault
0	6	Store/AMO address misaligned
0	7	Store/AMO access fault
0	8	Environment call from U-mode
0	9	Environment call from S-mode
0	10–11	*Reserved*
0	12	Instruction page fault
0	13	Load page fault
0	14	*Reserved*
0	15	Store/AMO page fault
0	16–23	*Reserved*
0	24–31	*Designated for custom use*
0	32–47	*Reserved*
0	48–63	*Designated for custom use*
0	≥64	*Reserved*

以下讲解的内容，应结合上述图片，及 xv6 源代码 trap.c 中 kerneltrap()和 usertrap()理解。

中断与异常

中断分为三种：外部中断、定时器中断、软件中断。异常包括各种因指令执行导致的错误、缺页、系统调用等。

M、S、U 模式下有各自的中断/异常处理程序（当然，U 模式下的几乎没人用）。处理程序应读取 mcause（或 scause、ucause）寄存器，以分辨中断/异常的类型，进而分门别类处理。

中断与异常的委托

默认情况下，RiscV 的所有中断和异常都在 M 模式下处理。对于操作系统，M 模式只是启动之初的承上启下，完成其初始化的使命后就应退出历史舞台，我们希望所有中断都在 S 模式下处理。为此，RiscV 允许将所有外部中断及异常委托到 S 模式下处理，xv6 在 start()中即做了相关配置。

定时器中断

定时器不是外部设备，每个 CPU 都集成了自己的定时器，其中断不是外部中断。按照 RiscV 的规定，定时器的设定和中断必须在 M 模式下处理，不得委托。这一规定是为了确保 M 模式拥有高于 S 模式的权限，即无论 S 模式下的程序如何操作，定时器始终被 M 模式的程序掌握在手中。因此，我们必须为定时器准备 M 模式下的中断处理程序。

注：上述说法来自 xv6 的官方文档（September 5, 2022 版，5.4 节），但似乎并不正确，新版本的 xv6 中就启用了 S 模式下的定时器中断，因而处理起来更简洁。

软件中断

针对中断，有 mip 寄存器（machine interrupt pending），每一位对应某一类中断源（例如第 1 位对应 S 模式的软件中断），表示该类型的中断目前已发生待处理。类似的，也有 sip 和 uip 寄存器，分别用于 S 模式和 U 模式，含义同 mip，但允许访问的位更少。

软件中断不是“软中断指令”，而是通过程序设置 sip 寄存器的第 1 位（对应 S 模式的软件中断），然后 CPU 会真实发生 S 模式下的中断，跳转到 S 模式下的中断处理程序。

外部中断

外部中断通过 PLIC 控制器管理。中断处理程序通过读取 mcause（或 scause、ucause）寄存器，可分辨中断的类型，如果发现是外部中断，则须进一步访问 PLIC 控制器（plic_claim()）以获取 IRQ 编号。

PLIC 控制器与外部设备是一对多的关系，多个设备发来中断请求时，PLIC 负责仲裁谁被优先处理。PLIC 控制器与 CPU 核心也是一对多的关系，PLIC 可能把同一个中断请求发送给多个 CPU，当中断处理程序执行 plic_claim()，既是为了获取 IRQ 编号，也是为了认领该中断，晚来的 CPU 只会获取到 0。中断处理结束后，还要再次访问 PLIC（plic_complete()），告知本次中断处理结束。

定时器中断

由于 RiscV 的硬件限制，我们必须在 M 模式下处理定时器中断，但处理程序中仅做简单基础的处理，然后设置 sip 寄存器以触发软件中断。也就是说，S 模式下的软件中断其实是定时

器中断触发的。如此一来，我们即可在 S 模式下（即系统内核）通过处理软件中断实质性的处理定时器中断。
关于 M 模式下的初始化和中断处理，涉及 start.c 和 kernelvec.S 中的 timervec。
关于 S 模式下软件中断的处理，涉及 main.c、kernelvec.S 中的 kernelvec、trap.c。其中 trap.c 中的 kerneltrap()和 devintr()居于核心调度的地位。
注：如上一小节所述，新版本的 xv6 中启用了 S 模式下的定时器中断，因而没这么繁琐。

串口中断

串口中断处理的主线与定时器中断大体一致，仍然围绕 trap.c 中的 kerneltrap()和 devintr()开展。额外的处理包括中断的委托、PLIC 的初始化、外部中断的处理框架、串口的中断处理和读操作，涉及 start.c、plic.c、trap.c、uart.c。

其它

须注意，trap.c 中有一些与用户态中断/异常相关的内容，以及一些与进程管理、磁盘相关的内容，现阶段都应删掉。
不要忘记本阶段的要求，出于测试目的，定时器每经过 30 次中断，就向屏幕（串口）输出一个“T”字符。
main()的末尾要调用 intr_on()开放中断。
各种 typedef、常量、函数声明等，涉及 defs.h、param.h、types.h、memlayout.h。

4．启动首个进程

本阶段目标：在上一阶段的基础上，启动首个进程，该进程执行两个简单的系统调用，随后进入死循环即可。即，完成启动后，0 号 CPU 陷在首进程用户态的死循环中，其余 CPU 陷在 main()末尾的死循环中。
本阶段不实现：进程状态、多进程、时间片、有实际功能的系统调用、文件系统。
从效果上看，屏幕上首先看到启动过程中每个 CPU 的输出提示，随后看到进程的两次系统调用导致的输出，完成启动后，可响应定时器和键盘的中断，每隔大约 3 秒，屏幕上出现一个“T”字符，并可显示键盘的输入。

xv6 的工作机制

xv6 定义了 64 个元素的 PCB 数组，即最多可创建 64 个进程。启动之初，main()调用 kvminit()时，就为 64 个进程开辟并映射了内核栈。然后 main()调用 procinit()，为这 64 个进程的

PCB 指定了内核栈的虚拟地址。

在启动的最后阶段，main()调用 userinit()用于启动第一个进程，userinit()执行如下：

1）调用 allocproc()，执行最初的初始化

1.1）allocproc()找到空闲的 PCB，设置 PID 和状态

1.2）为新进程开辟 trapframe

1.3）调用 proc_pagetable()，创建页表，并映射 trampoline 页和 trapframe 页

1.4）为新进程设置 context，包括内核栈顶、返回地址

2）调用 uvmfirst()，为首进程载入代码并映射代码页

3）为新进程设置 trapframe，包括用户栈顶、返回地址

4）将新进程设为就绪态

特别说明：

1）以上步骤 1.4 提及 context，即切换到该进程后将恢复到 CPU 的各寄存器。xv6 在此设置返回地址（ra 寄存器）为 forkret()的地址，意味着将来切换到该进程时，执行切换的函数 swtch()最后执行 ret 返回指令，将返回到 forkret()执行。也就是说，每个进程最初获得 CPU 时，都是从 forkret()开始，forkret()又将调用 usertrapret()，从而跳转到进程用户态。

2）以上步骤 2 要载入首进程（1 号进程）的程序，但此时文件系统尚未准备好（因为文件系统的初始化本身就要依托进程的阻塞能力，实际上上面提到的 forkret()就负责触发文件系统的初始化），因此，首进程的程序无法从磁盘获取。xv6 的做法是，把首进程的程序（用汇编语言编写，极简，仅触发 exec()系统调用启动著名的 init 进程）事先编译成可执行文件，然后把机器码程序直接当作字节数组导出，再把该字节数组定义在 proc.c 的 initcode[]数组中。也就是说，首进程的程序被当作一份全局数组编译在内核中。

3）由于首进程极简，执行过程不需要栈，所以 userinit()没有为首进程创建用户栈。

4）以上 userinit()只是把进程在内存中准备好并设置为就绪态，并未切换到首进程，而是返回 main()，main()执行到最后，会进入一个死循环，在其中执行调度程序。调度程序自然会发现就绪的首进程，并切换过去。

trampoline.S 和 swtch.S

这两个汇编程序文件可以直接从 xv6 引用，幸运的是，一行都不用改。

特别注意，我们在第一阶段从 kernel.ld 中删掉了 trampoline 页相关的内容，本阶段务必恢复，以保证 trampoline 的程序是页对齐的，将来才有可能将 trampoline 单独映射为跳板页。

相应的，在 vm.c 的 kvmmake()中，要增加对 trampoline 跳板页的映射。

数据结构

本阶段任务涉及 proc.h 中以下结构体的定义：cpu、context、trapframe、proc。
其中前三者都与 xv6 保持一致，proc 则需要简化，删掉与进程状态相关、与自旋锁相关、与文件相关的成员。

首进程相关

遵循 xv6 的机制，创建首进程主要涉及 proc.c 的 procinit()、userinit()，以及由此嵌套调用、访问的各函数、数据。

PCB 全局数组简化为单个全局结构体

xv6 在 proc.c 中定义了 PCB 的全局数组，本阶段任务用不到，可将其改为单个 PCB 全局结构体。相应的，

1）procinit()原本为 64 个进程的 PCB 指定内核栈地址，改为设定唯一的 PCB 结构。

2）allocproc()原本从 PCB 数组中寻找空闲的，并按自增原则设置 PID，均可简化。

3）allocproc()原本在创建 trapframe 和用户页表时出错会调用 freeproc()清理页表、trapframe 和 PCB，对于现阶段的首进程，调用 freeproc()没啥意义。

4）如前述，main()调用 vm.c 中的 kvminit()，kvminit()调用 kvmmake()构建内核页表，kvmmake()原本在其末尾调用 proc_mapstacks()为 64 个进程开辟并映射了内核栈，此处改为直接针对唯一的进程开辟映射内核栈。

首进程程序

根据本阶段任务的需要，我们不能直接使用 xv6 自带的首进程代码，应使用如下等效代码：

initcode.c

```
int main() {
    syscall(12); // 封装 12 号系统调用，a7 寄存器装入 12 后执行 ecall 指令
    syscall(12); // 再次发起相同系统调用
    while(1) ;   // 死循环
    return 0;
}
```

上述代码不能直接使用，老师已将其编译并导出为字节数组。上述代码的细节，以及如何编译、导出为字节数组，本阶段可忽略。字节数组如下：

```
uchar initcode[] = {
    0x13, 0x01, 0x01, 0xff, 0x23, 0x34, 0x11, 0x00,
    0x23, 0x30, 0x81, 0x00, 0x13, 0x04, 0x01, 0x01,
    0x13, 0x05, 0x40, 0x01, 0x97, 0x00, 0x00, 0x00,
    0xe7, 0x80, 0x80, 0x01, 0x13, 0x05, 0x40, 0x01,
    0x97, 0x00, 0x00, 0x00, 0xe7, 0x80, 0xc0, 0x00,
```

```
    0x6f, 0x00, 0x00, 0x00, 0x93, 0x08, 0xc0, 0x00,
    0x73, 0x00, 0x00, 0x00, 0x67, 0x80, 0x00, 0x00
};
```

将以上数组复制到 proc.c 的 initcode[]数组中，代替原数组。

须注意，uvmfirst()为首进程载入代码并映射代码页时，假定代码不超过一页，我们须严格遵守这一点。不过对于首进程来说，4KB 本来就绰绰有余了。

首进程用户数据页

如前述，xv6 的首进程极为简单，没有用到用户栈。我们的首进程程序稍复杂，将来阶段还可能更复杂，我们要在 userinit()的末尾创建全局数据区（2 个页）和用户栈（1 个页）并在用户页表中建立映射。

也就是说，用户空间的第 0 页是程序，第 1、2 页是全局数据，第 3 页是栈。未来如使用堆，则堆从第 4 页开始向高地址增长。

特别注意，全局数据区需要初始化为 0！应用程序指望你这样做，如果你不做，会发生奇怪的错误，而我们的进程程序是以机器码字节数组的方式加载的，无法调试。

相应的，我们要把用户态的栈顶指针（p->trapframe->sp）设置在第 3 页的末尾（即第 4 页的起始地址）。

切换跳转相关

除了前面已经引入项目的 trampoline.S 和 swtch.S，进程的切换跳转还涉及 trap.c 中的 usertrap()、usertrapret()。

usertrap()原本对于系统调用会调用 syscall()进行分发处理。本阶段简化处理，将系统调用的处理程序改为如下：

```
...
if (r_scause() == 8)
{
    // system call
    printf("get a syscall from proc %d\n", myproc()->pid);
    p->trapframe->epc += 4;
    intr_on();
}
else
...
```

此外，usertrap()的末尾有一些与进程状态有关的内容，应删去。

如前述，xv6 中 allocproc()把进程的返回地址设置为 forkret()，本阶段我们尚未构建起

fork()系统调用，没有 forkret()，因此应该把返回地址修改为 usertrapret()。
此外，我们说过，xv6 中 userinit()只是把首进程设置为就绪态，main()运行到最后执行调度程序，从而切换到首进程。现阶段我们还没有进程状态，也没有准备好调度程序，不能指望 main()实现切换。为此，我们要在 userinit()的末尾调用 swtch()，强行切换到首进程。

其它

本阶段没有进程结束功能，main()调用 userinit()将导致 CPU 切换到进程用户态再也回不来，因此对 userinit()的调用须放在 started 置 1 之后，否则会阻碍其它 CPU 的启动。
内存的简单批量操作，涉及 string.c。
各种 typedef、常量、函数声明等，涉及 defs.h、param.h、types.h、memlayout.h。
各种与页表有关的计算、操作的宏，涉及 riscv.h。

5．系统调用机制和用户堆管理

本阶段目标：在上一阶段的基础上，将系统调用的机制完善起来，并提供 sbrk()系统调用，实现用户堆内存的增长、收缩。完善系统调用机制，不是要求实现所有系统调用，而是指搭建起具备扩展能力的代码框架。由于尚未实现进程状态和文件系统，绝大多数系统调用都不具备实现的条件，因此本阶段只能实现 sbrk()系统调用。
本阶段仍然只能启动首个进程，但首进程的程序比上一阶段复杂一些，会在其 main()函数中执行一些堆分配和释放的测试，随后进入死循环。
从效果上看，屏幕上首先看到启动过程中每个 CPU 的输出提示，随后看到进程堆操作触发的若干次系统调用导致的输出，完成启动后，可响应定时器和键盘的中断，每隔大约 3 秒，屏幕上出现一个“T”字符，并可显示键盘的输入。

系统调用机制的搭建

上一阶段，我们把 trap.c 中 usertrap()中与系统调用相关的分支做了简化处理。本阶段须恢复 xv6 在此处的原貌。
相应的，要把 xv6 的 syscall.h 和 syscall.c 引入项目。其中包含了核心的 syscall()函数，并声明了 xv6 中所有系统调用的处理函数。本阶段我们只打算实现 12 号（SYS_sbrk）系统调用，其它无关的应屏蔽掉。
syscall.c 中还提供了处理系统调用时重要的支撑函数，即参数获取函数。这些函数又进一步涉及了 vm.c 中的 copyout()、copyin()、copyinstr()等函数。

用户堆的基本原理

在 xv6 中，用户内存空间从低地址向高地址分布如下：
1）程序页面，从 0 开始，可执行文件中会声明需要多大
2）全局数据区，可执行文件中会声明需要多大
3）栈，固定为一个页，4KB
4）堆，大小不固定，最初为 0
5）空闲未用的区域，堆可根据需要侵占或回缩
6）Trapframe、Trampoline
也就是说，从 0 开始，直到堆的顶部，是一整片有效的、投入使用的、构建了映射的连续页面。对于系统来说，当前堆的顶部的地址也就是该进程的内存占用量，记录在 PCB 的 sz 字段。系统只记录管理堆的顶部边界，并确保顶部以下所有页面被映射。
堆的使用由用户程序做主。如何使用，用什么数据结构组织管理，哪些部分正被使用，哪些部分空闲，都不需要向系统汇报。只有当用户程序发现堆的容量不够，希望把顶部边界向前推，或者发现堆的顶部存在大量空闲，希望把顶部边界向回收缩，这时才需要发起 sbrk()系统调用，调整边界和页面映射。

sbrk()系统调用的实现

该系统调用的处理函数是 sysproc.c 中的 sys_sbrk()。由此出发，涉及 proc.c 中的 growproc()，vm.c 中的 uvmalloc()、uvmdealloc()、walkaddr()。

其它

各种 typedef、常量、函数声明等，涉及 defs.h、param.h、types.h、memlayout.h。

用户首进程的处理

根据本阶段任务的需要，我们需要设计更加复杂的首进程代码。老师已经准备好了程序，大致结构如下：
initcode.c

```
...


void free(void *ap) {
    ...    // 本函数释放堆内存，必要时调用 add_heaptop()
}
```

```
static Header *add_heaptop(int nu) {
    ...    // 本函数调用 sbrk()
}

void *malloc(uint nbytes) {
    ...    // 本函数分配堆内存，必要时调用 add_heaptop()
}

int main() {
    int *p[10];
    for (int i = 0; i < 10; i++)
        p[i] = malloc((i + 1) * 1000);
    free(p[0]);
    ...    // 连续调用 free()
    free(p[1]);

    while (1) ;
    return 0;
}
```

上述代码中用到了 sbrk()，该函数须使用汇编语言编写，实现方式借鉴 xv6 的方法：

1）事先准备好 usys.pl 脚本，编译时用 perl 命令执行该脚本，生成 usys.S 汇编程序

2）用 gcc 对 initcode.c 和 usys.S 分别编译，生成两个.o 文件

3）依照老师提供的 user.ld 进行连接，生成 initcode.out（ELF 规范的可执行文件）

4）用老师提供的 ELF 文件解析工具，分析 initcode.out，产生 initcode.txt

针对以上步骤老师已准备好 Makefile，执行 make init，即可自动执行直至产生 initcode.txt。

打开 initcode.txt，即可看到 initcode.out 的文件结构和内容，大致如下：

```
ELF Header:
    ...
    entry: 0x2e0
    phoff: 0x40
    ...
    phnum: 0x4
    ...
```

```
Program section header 0:
   ...

Program section header 1:
   type: 0x1
   flags: 0x7
   off: 0x120
   vaddr: 0x0
   paddr: 0x0
   filesz: 0x3b0
   memsz: 0x1000
   align: 0x4

   Content:
   0x13, 0x01, 0x01, 0xfe, 0x23, 0x3c, 0x11, 0x00,
   0x23, 0x38, 0x81, 0x00, 0x23, 0x34, 0x91, 0x00,
   0x13, 0x04, 0x01, 0x02, 0x93, 0x04, 0x05, 0xff,
   ...

Program section header 2:
   type: 0x1
   flags: 0x6
   off: 0x4d0
   vaddr: 0x1000
   paddr: 0x1000
   filesz: 0x0
   memsz: 0x20
   align: 0x8

   Content:
   [None: 0 size]

Program section header 3:
   ...
```

其中可以看到程序编译后导出的字节数组，复制到 proc.c 的 initcode[]数组中，代替原数组。

注意，上述 initcode.c 及其配套的 usys.pl、user.ld、Makefile、ELF 分析工具等不是系统内核程序的一部分，而是一个独立的、辅助性的项目，唯一的目的就是产生 initcode[]数组。

此外，main()函数并不一定位于代码区的开头，initcode.txt 的开头 ELF Header 段落有 entry 字段，即 main()函数的起始虚拟地址。在 proc.c 的 userinit()的末尾，执行切换前，应把 main()的起始地址设置到 p->trapframe->epc。

6．进程状态与多进程调度

本阶段目标：在上一阶段的基础上，实现进程的多种状态，以及多进程的调度切换。相应的，实现以下系统调用：getpid()、uptime()、fork()、sleep()、wait()、exit()、kill()。此外，为了实现用户态的 printf()，对串口增加异步传输的支持，并相应增加 write()系统调用。

本阶段仍然只能启动首个进程，但首进程的程序更加复杂，包含 3 个子进程，从而执行上述所有系统调用的测试，随后进入死循环。程序由老师提供。

从效果上看，屏幕上首先看到启动过程中每个 CPU 的输出提示，随后看到各个子进程的执行输出。第一个子进程简单死循环，没有任何输出；第二个子进程执行堆操作触发若干次系统调用导致输出，随后退出；第三个子进程执行三次延迟等待后输出，并在退出前向第一个子进程执行 kill()；主进程则循环 wait()等待子进程结束。完成启动后，可响应定时器和键盘的中断，每隔大约 3 秒，屏幕上出现一个“T”字符，并可显示键盘的输入。

进程状态与多进程

我们在前面的任务中，曾经在 proc.h 中注释掉 proc 结构体中与状态有关的几个成员，包括 state、chan、killed、xstate，这些都应该恢复。结构体中还有一个自旋锁和父进程，也应恢复。

proc.c 中，应按照 xv6 的做法，恢复 PCB 的全局数组，恢复动态分配 PID 的 allocpid()及相关的数据，此外还要恢复自旋锁 wait_lock（你能说出这个自旋锁的作用吗？）。

vm.c 中，我们曾修改 kvmmake()末尾为进程分配内核栈的代码，现在也应恢复。

进程调度及相关系统调用

进程的状态改变、调度涉及多个函数，须厘清脉络，可分为以下几个模块：

1）sched()

基础的函数，被多处调用，用于从当前进程切换到 main()上下文（也就是调度器）。

2）唤醒进程

wakeup()，另一个基础的函数，被多处调用，使符合条件的阻塞进程进入就绪态。

3）创建进程

核心是 allocproc()，被 userinit()和 fork()调用，使进程进入就绪态。

4）时间片结束

yield()，使本进程进入就绪态，并进而调用 sched()让出 CPU。

yield()因定时器中断而触发，但并不在中断处理函数 clockintr()中被调用，而是在 usertrap()和 kerneltrap()中分别被调用（你能说出原因吗？）。

说到 clockintr()，该函数调用了 wakeup()，唤醒那些上了闹钟的进程。

5）阻塞

sleep()，使本进程进入阻塞态，并进而调用 sched()让出 CPU。

在 xv6 中，sleep()被多处调用，但大多数都与外设、文件、终端有关。仅考虑进程调度，则只有 wait()调用了 sleep()。

6）退出

exit()，使本进程进入僵尸态，并进而调用 sched()让出 CPU。

一个进程的退出需要其父进程执行 wait()，所以 exit()还要调用 wakeup()唤醒父进程。

一个进程的退出会导致其子进程被 init 进程领养，如果此时子进程已经变成僵尸态，则子进程没有机会唤醒 init 进程，所以 exit()还要调用 wakeup()唤醒 init 进程，不管三七二十一让它先执行一轮 wait()。

与进程调度相关的函数中，kill()、setkilled()将导致进程退出，但这两个函数并不直接调用 exit()，而是设置 PCB 的 killed 成员，之后随着进程的调度执行，来到 usertrap()中，将根据 killed 成员调用 exit()。

7）调度器

scheduler()，在 main()的末尾被调用，一个死循环，寻找就绪进程，设为执行态，并切换过去。

上述函数，有些在之前的任务中就已引入，但曾做过某些修改，本阶段须根据需要恢复其面貌。特别注意，之前我们在 allocproc()中把进程的返回地址修改为 usertrapret()，现在务必恢复为 forkret()，并删去 forkret()中初始化文件系统的程序。请观察 forkret()在调用 usertrapret()前做了什么，想明白为什么要这样做。

如前述，我们要实现 6 个进程调度相关的系统调用，涉及 sysproc.c 中的 sys_getpid()、sys_uptime()、sys_fork()、sys_sleep()、sys_wait()、sys_exit()、sys_kill()。

用户态的 printf()

在 xv6 中，printf()向“标准输出”发送内容，所以与文件系统有关。printf()是一个用户态的库函数，针对 1 号文件描述符触发了 write()系统调用。

在 xv6 的内核中，对于 write()系统调用，会根据文件描述符分情况讨论。文件描述符的背

后，可能是终端，可能是管道，可能是文件。其中对于终端，最终通过 console.c 中的 consolewrite()完成输出。
本阶段，我们还没有文件系统。出于方便考虑，我们直接引用 xv6 的 user 目录下 printf.c 中的 printf()函数，同时构造一个简化的 write()系统调用。在 write()系统调用中，我们忽略文件描述符，直接调用 consolewrite()。

串口的异步传输

consolewrite()最终调用串口的输出函数。需要注意的是，xv6 中对于串口的输出提供了同步和异步两种方式。内核中的 printf()使用的是同步方式，而用户态的 printf()通过 consolewrite()，调用的是异步方式 uartputc()。
串口的异步输出涉及 uart.c 中的 uartputc()、uartstart()、uartintr()，以及由此引出的数据、初始化等。

其它

各种 typedef、常量、函数声明等，涉及 defs.h、param.h、types.h、memlayout.h。

首进程程序

首进程程序沿用上一阶段所描述的处理方式。老师已经准备好了程序，大致结构如下：
initcode.c

```
// malloc 相关
...
void free(void *ap) {
    ...    // 本函数释放堆内存，必要时调用 add_heaptop()
}
static Header *add_heaptop(int nu) {
    ...    // 本函数调用 sbrk()
}
void *malloc(uint nbytes) {
    ...    // 本函数分配堆内存，必要时调用 add_heaptop()
}

// printf 相关
...
```

```
static void putc(int fd, char c) {
    write(fd, &c, 1);
}
static void printint(int fd, int xx, int base, int sgn) {
    ...    // 本函数输出整数，调用 putc()
}
static void printptr(int fd, uint64 x) {
    ...    // 本函数输出地址，调用 putc()
}
void vprintf(int fd, const char *fmt, va_list ap)
{
    ...    // 本函数综合处理，根据情况调用 putc()或 printint()或 printptr()
}
void printf(const char *fmt, ...) {
    ...
    vprintf(1, fmt, ap);
}

// 测试
int main() {
    int pid, to_be_killed;
    pid = fork();    // child 1: 100% CPU
    ...
    if (pid == 0) {
        while (1) ;
        exit(0);
    }
    to_be_killed = pid;
    pid = fork();    // child 2: dynamic memory allocation
    ...
    if (pid == 0) {
        int *p[10];
        for (int i = 0; i < 10; i++)
            p[i] = malloc((i + 1) * 1000);
        free(p[0]);
        ...    // 连续调用 free()
```

```
        free(p[1]);
        exit(0);
    }
    pid = fork();    // child 3: sleep, then kill child 1
    ...
    if (pid == 0) {
        sleep(10);
        printf("[pid: %d] sleep test - 1s. uptime: %d\n", ...);
        ...
        kill(to_be_killed);
        exit(0);
    }
    while (1) {    // init process: wait
        int pid, xstate;
        pid = wait(&xstate);
        ...
    }
    ...
    return 0;
}
```

此外，usys.pl 要根据本阶段新增的系统调用做出调整。

7．磁盘盘块读写

本阶段目标：在上一阶段的基础上，实现磁盘在 QEMU 虚拟机上的加载，以及系统对盘块的读写能力。为了方便测试，需要对 write()系统调用做一些关于盘块读写测试的扩展。

本阶段仍然只能启动首个进程，在上一阶段的基础上，对 child 1 子进程增加了盘块读写测试的调用。程序由老师提供。

从效果上看，屏幕上的输出内容保留了上一阶段的所有输出，期间增加了对某些磁盘盘块的内容输出。

文件系统概述

从程序的业务逻辑上看，文件系统自底向上被分为五层：

1）磁盘驱动程序，主要涉及 virtio.h 和 virtio_disk.c

2）缓冲/缓存层，主要涉及 buf.h 和 bio.c

3）日志事务层，主要涉及 log.c
4）inode 和相关的盘块组织层，主要涉及 fs.h 和 fs.c
5）文件抽象层，向下对接的除了 inode，也可能是终端、管道等，主要涉及 file.h 和 file.c
6）系统调用接口层，主要涉及 sysfile.c
本阶段，我们完成底部的两层。

sleeplock

磁盘访问过程中不可避免出现获取互斥资源不得而陷入阻塞的情况，为此 xv6 中设计了 sleeplock 结构体及其相关程序。涉及 sleeplock.h 和 sleeplock.c，程序简短，不难懂，须引入项目。

虚拟磁盘和 mkfs.c

mkfs 目录下的 mkfs.c 可以原封不动的引入项目，但其程序与磁盘文件系统密切相关，仍应读懂，并弄清楚该程序与 xv6 程序之间的关系。

来龙去脉
QEMU 虚拟机需要挂载一个虚拟的磁盘，观察 Makefile 不难发现，虚拟磁盘是以 fs.img 文件的形式提供的。因此，fs.img 的内容就是扇区的数组。如此说来，只要保证 fs.img 的文件尺寸是 512B 的整数倍，即可担任合格的虚拟磁盘。但在我们的项目中，系统假定磁盘上已经构建了文件系统，所以我们还要对磁盘做“格式化”，而执行格式化的程序，就是 mkfs.c。注意观察 Makefile，会发现 mkfs.c 的编译使用的是普通的 gcc，而非交叉编译器。也就是说，mkfs.c 不是在 QEMU 虚拟机中运行的 RiscV 程序，而是先于 QEMU 虚拟机运行的，在 Linux 下编译运行以生成 fs.img 文件的程序。梳理一下 Makefile，过程如下：
1）交叉编译器编译 xv6 的内核程序；
2）交叉编译器编译 xv6 的应用程序；
3）普通 gcc 编译器编译 mkfs.c，生成 mkfs 可执行文件；
4）执行 mkfs 可执行文件，产生了 fs.img 文件，该文件内部已按文件系统规范做格式化，并已将 README 和前述 xv6 的应用程序复制到了虚拟磁盘的根目录下；
5）启动 QEMU 虚拟机，执行内核程序，并将 fs.img 作为虚拟磁盘挂载到虚拟机上。

磁盘布局
xv6 的文件系统是对 Linux 的 ext 文件系统的简化。磁盘的最小访问单位是扇区（sector），512B，文件系统的最小组织单位是盘块（block），1024B。虚拟磁盘共包含 2000 个 block，从前往后依次分成几个段落如下：
1）boot block，占据 1 个 block，xv6 并不保存在虚拟磁盘上，因此该 block 没用；

2）super block，占据 1 个 block，对文件系统的整体描述；
3）log 区，存放日志事务，占据 30 个 block；
4）inode 区，xv6 仅支持 200 个 inode，每个 inode 占 64B，共占据 13 个 block，1 号 inode 对应根目录；
5）空闲位图区，占据 1 个 block，可表达 8192 个 block 的空闲情况，而我们的磁盘仅包含 2000 个 block，足够了；
6）数据区，占据剩余所有的 block。
其中数据区的第 1 个 block 存放根目录的内容；数据区的第 2 个及后续 block 依次存放 README 和前述 xv6 的应用程序文件。数一下，发现 README 保存在 47-49 号 block（从 0 开始计）。

缓冲/缓存

涉及 buf.h 和 bio.c，可以原封不动的引入项目，但其程序仍应读懂。
每个 buf 结构体可对应一个 block，既做缓冲区也做缓存。当我们需要读写某个 block，首先要获取一个空闲的（引用数为 0）buf 结构体，将其绑定到 block 盘块号，之后 buf 结构体中的 data 数组就成为该 block 的读写缓冲区。完成读写后，我们要释放该 buf 结构体使其恢复空闲（引用数清 0），但并不撤销其与 block 的绑定关系，也不清空缓冲区数据，因此该 buf 结构体就成为 block 的缓存。
bio.c 中定义了 bcache 全局结构，其中包含了 30 个 buf 结构体构成的数组，这 30 个 buf 结构体又通过各自的 prev 和 next 指针构建了双向链表环，链表按照 most recently used 规则排列，即刚被释放的 buf 结构体总被插到最前面，而获取空闲 buf 结构体时总是从链表末尾（即最近最久未使用的缓存块）向前寻找。
下层的磁盘驱动程序提供接口 virtio_disk_rw()函数，该函数被本层的 bread()和 bwrite()函数调用。

磁盘驱动程序

涉及 virtio.h 和 virtio_disk.c，可以原封不动的引入项目，其程序涉及较多硬件细节（QEMU 虚拟机的外设遵循 virtio 规范，预定义了各种寄存器、常量等），除非花较多时间另外阅读相关手册，否则很难完全读懂，但其对数据结构的组织管理仍值得读一读，从而领会今天的接口电路（即课堂上讲的控制器）是如何在 DMA 机制的帮助下与主机（即 CPU、内存）交互工作的。

核心数据结构：struct virtq_desc（描述符）的数组
磁盘控制器要求软件系统（即驱动程序）在内存中准备该数组，并把该数组的起始物理地址写入控制器的特定寄存器。作为比较，回顾 CPU 要求软件系统在内存中准备页表，并把页表

起始物理地址写入 CPU 的特定寄存器。
描述符内部包含 next 字段（数组下标），因而可在数组内构成链表。
一次读写传输需要三个描述符，这三个描述符在数组中并不需要连续，而是通过 next 字段构成“三节点链”，第一个描述符指向 struct virtio_blk_req 结构，说明操作类型（读/写）、盘块号，第二个描述符指向读写的数据，第三个描述符指向一个状态字节变量，读写完成后 DMA 控制器会在此留下状态。

核心数据结构：struct virtq_avail 结构体
磁盘控制器要求软件系统（即驱动程序）在内存中准备该结构体，并把其起始物理地址写入控制器的特定寄存器。
如前述，每个传输请求对应描述符数组内的“三节点链”，只要提供其首节点在描述符数组中的下标，即可顺藤摸瓜找到所有三个节点。
本结构体表示有待传输的请求队列。结构体内包含一个数组，每个元素存放一个“三节点链”的首节点下标，即对应一个传输请求。

核心数据结构：struct virtq_used 结构体
磁盘控制器要求软件系统（即驱动程序）在内存中准备该结构体，并把其起始物理地址写入控制器的特定寄存器。
本结构体表示完成传输的请求队列。结构体内包含一个数组，每个元素存放一个“三节点链”的首节点下标，即对应一个传输请求。

驱动程序概述
以上提及的核心数据结构均为磁盘控制器硬件要求提供并可识别处理的数据结构。至于这些数据结构放在内存的什么地方、相互如何关联管理，则是驱动程序的自由。
对于上层发来的读写请求，驱动程序最终转化为对上述三个核心数据结构的修改。读写过程中进程阻塞，读写完成后进程被中断唤醒，会发现上述核心数据结构已被磁盘控制器留下了痕迹，从而做相应的后续处理。

virtio_disk_init()
初始化磁盘控制器，启动时被 main()调用。将上述三种核心数据结构的起始物理地址写入寄存器。涉及硬件细节多，复杂，看不懂的可以跳过。

virtio_disk_rw()
执行盘块读写的核心函数，也是对上层（缓冲/缓存层）开放的接口。工作过程如下：
1）分配描述符，设定描述符
2）将 buf 结构体的 disk 成员（表示该结构已经提交给磁盘，正在读写过程中）置为 1
3）将本次请求的“三节点链”的首描述符下标填入 struct virtq_avail 结构体的数组
4）通知磁盘控制器有待办任务，然后阻塞于 buf 结构体，直至其 disk 成员变为 0
5）苏醒后释放描述符

virtio disk intr()

中断处理程序，读写完成时会发生中断，此时会发现 struct virtq_used 结构体中被磁盘驱动器添加了新的已完成请求。

本函数扫描 struct virtq_used 结构体的数组，获取每个已完成传输请求的“三节点链”的首描述符的下标，根据下标找到 buf 结构体，将 disk 成员置为 0，并针对 buf 结构体唤醒进程。

测试程序

write()系统调用

上一阶段任务中，我们让 write()系统调用直接调用 consolewrite()。本阶段做个扩展，对于文件描述符参数小于 3 的情况，仍然直接调用 consolewrite()；对于文件描述符参数为 3 的情况则执行盘块读写测试。

盘块读写测试包含以下过程：

1）读取 super block，并将其内容输出到屏幕上

2）读取 README 文件的第一个盘块（如“虚拟磁盘和 mkfs.c”一节末尾所述，应为 47 号 block），并将其内容输出到屏幕上

3）将上述第一个盘块中的第一段文字保留，后续全部清 0，并写回磁盘

4）再次读取 README 文件的第一个盘块，并将其内容输出到屏幕上

相关程序如下：

```
int blocktest()
{
    struct buf *b;
    // read superblock
    b = bread(ROOTDEV, 1);
    struct superblock *sb = (struct superblock *)(b->data);
    printf("Super Block info:\n");
    printf("\tmagic: %x\n", sb->magic);
    printf("\tsize: %d\n", sb->size);
    printf("\tnblocks: %d\n", sb->nblocks);
    printf("\tninodes: %d\n", sb->ninodes);
    printf("\tnlog: %d\n", sb->nlog);
    printf("\tlogstart: %d\n", sb->logstart);
    printf("\tinodestart: %d\n", sb->inodestart);
    printf("\tbmapstart: %d\n\n", sb->bmapstart);
    brelse(b);
```

```
    // read first file data block
    b = bread(ROOTDEV, 47);
    char *c = (char *)(b->data);
    c[BSIZE - 1] = '\0';
    printf("README (1KB):\n%s\n\n", c);

    // modify first file data block
    int i;
    for (i = 0; i < BSIZE - 1; i++)
    {
        if (c[i] == '\n' && c[i + 1] == '\n')
            break;
    }
    if (i < BSIZE - 1)
    {
        for (; i < BSIZE; i++)
            c[i] = 0;
    }
    bwrite(b);
    brelse(b);

    // confirm first file data block
    b = bread(ROOTDEV, 47);
    c = (char *)(b->data);
    c[BSIZE - 1] = '\0';
    printf("README (modified):\n%s\n\n", c);
    brelse(b);

    return 0;
}

uint64 sys_write(void)
{
    ...
    int fd_test;
```

```
    argint(0, &fd_test);
    if (fd_test < 3)
        return consolewrite(1, p, n);
    else if (fd_test == 3)
        return blocktest();
    return 0;
}
```

首进程程序

首进程的程序与上一阶段相比仅做一处修改。在 main()函数中，对于 child 1 子进程段落，

```
    if (pid == 0) {
        while (1) ;
        exit(0);
    }
```

改为

```
    if (pid == 0) {
        write(3, 0, 0); // 3 is testcode for block r/w, see sys_write()
        while (1) ;
        exit(0);
    }
```

其它

引入上述 xv6 程序后，还须引入以下被#include 的头文件：fs.h、stat.h。

main.c，main()中要增加对缓冲/缓存的初始化，以及对磁盘控制器的初始化。

plic.c，plicinit()和 plicinithart()中要增加针对磁盘的内容。

trap.c，devintr()中要增加针对磁盘的内容。

vm.c，kvmmake()中要增加针对磁盘控制器的映射。

各种 typedef、常量、函数声明等，涉及 defs.h、param.h、types.h、memlayout.h。

Makefile，须增加与磁盘、mkfs、fs.img 相关的内容。

8．log 层、inode 层与 exec()

本阶段目标:在上一阶段的基础上,实现文件系统中的日志事务层和 inode 层,并实现 exec()系统调用。

本阶段对用户态程序做出调整，initcode 中的部分子进程被改写为独立的应用程序，并被 initcode 首进程通过 exec()系统调用启动。用户态程序由老师提供。

从效果上看，本阶段的执行输出与上一阶段完全一样。

日志事务层

日志事务层的内容在理论课上少有涉及，同学们普遍感到陌生。这部分涉及 log.c，可以原封不动的引入项目，但其程序仍应读懂。

基本原理

文件系统的数据结构是相互关联的，例如一次写文件操作就可能涉及磁盘的空闲位图区、inode 区、数据区，因此磁盘的写操作需要以“事务”的方式开展，即要么全套执行，维系方方面面的数据关联，要么啥也别做。日志事务层的目的是防范写到一半时出错，造成文件系统的内在一致性被破坏而崩溃。

如上一阶段“虚拟磁盘和 mkfs.c”一节所述，磁盘上设立有专门的 log 区，存放日志事务。所有的写磁盘操作，都分为上下半场，上半场先登记为一个“事务”并写到 log 区，写完后，下半场再把该事务写到真正的目标盘块（即空闲位图区、inode 区、数据区）。

如果在上半场向 log 区写入事务的过程中出错，那么这个事务就未能完整出现在 log 区，也就是说这个事务丢失了。这对于发起写操作的进程当然是不可挽回的损失，但日志事务层的目的是维护文件系统的内在一致性，是为了防范文件系统的崩溃，不是为了把每次写操作都贯彻落实。这一轮写操作尚未开启下半场，没有对空闲位图区、inode 区、数据区产生任何变更，因而保证了安全。

如果在下半场向真正的目标盘块执行写操作的过程中出错，那么因为 log 区还有完整的事务记录，系统重启过程中会发现 log 区还有未完成事务，于是可以重新执行写操作。

读磁盘操作不涉及日志事务层。

代码概述

磁盘的 super block 记录了 log 区的起始盘块和大小。log 区的首盘块用作 log header，记录本事务涉及的盘块数量和所有盘块号。相应的，log.c 中即创建了全局的 log 结构体，其中又包含 lh（即 logheader）结构体。

日志事务层的存在，要求所有的磁盘写操作都要遵循如下基本套路：

```
begin_op();  // 申请参与一个事务
...
log_write(struct buf *b);    // 一次或多次执行 log_write()，登记要写的盘块
...
end_op();    // 结束事务登记，启动向 log 区写盘块，然后向真正的目标盘块写
```

说明如下：

1）begin_op()，用于申请参与一个事务。

1.1）log 区只能容纳一个事务，如果该事务已完成登记并正在执行磁盘的写操作（无论

是上半场写入 log 区还是下半场写入真正的目标盘块），那么需要阻塞等待；

1.2）一个事务并不一定归属于单个进程的一次写操作，如果多个进程同时试图执行写操作，可以让多个进程的写操作加入同一个事务。但是 log 区容量有限，如果参与的进程太多可能造成 log 区容不下，则阻塞等待；

1.3）所谓参与一个事务，只是把登记的参与进程数加 1。

2）log_write(struct buf *b)，并不执行任何写操作，只是把盘块缓冲 b 所绑定的盘块号登记到 log.lh 中，需要写多个盘块，就调用 log_write()多次。

3）end_op()，完成登记，登记的参与进程数减 1，如果发现参与进程数减为 0 了，表示所有参与的进程都登记完了，就执行 commit()。

3.1）首先执行 write_log()（注意不是 log_write），把所有登记的盘块写入 log 区；

3.2）接下来执行 write_head()，把 log header 写入 log 区，至此整个事务真正被记录到磁盘上；

3.3）接下来执行 install_trans()，把事务写到真正的目标盘块；

3.4）最后把 log 区 log header 的盘块数清 0，至此磁盘的 log 区清空。

其它还有 recover_from_log()，系统启动阶段被 initlog()调用，重新执行未完成下半场的事务。

inode 和相关的盘块组织层

涉及 fs.c，可以原封不动的引入项目。该文件是 xv6 所有源文件中最长的之一，但其内容完全是对理论课 inode 相关知识的实现，应读懂。

代码读起来并不愉快，脑中要牢记磁盘的布局，深刻理解并熟练读懂缓冲/缓存层（struct buf 相关）和日志事务层（log 相关）的重要函数，此外一些过于简短的函数名/变量名/宏名也时常构成干扰。

需注意程序中有两个与 inode 相关的结构体：struct dinode 和 struct inode，前者与磁盘上保存的 inode 完全一致，后者定义于 file.h，在前者基础上增加了锁、盘块号、引用次数之类的成员以便系统管理和操作。

main()中要相应增加对 iinit()的调用。

exec()系统调用

涉及 exec.c 和 elf.h，可以原封不动的引入项目，程序应读懂。

前几个阶段我们都对 initcode.c 编译后的可执行文件进行过解析，对 elf 文件规范已有一定的了解，因而此处阅读 exec()和 loadseg()中的相关程序应该不难。

exec()中对 argc、argv 入栈和传参的处理略繁琐，先把 argv 的所有字符串入栈，再把 argv 指针数组（每个元素指向一个字符串）入栈，最后通过 a0、a1 寄存器传递 argc 和 argv。

其它

当前路径

进程 PCB 中要记录“当前路径”，之前的阶段未准备好文件系统，因此删去了相关内容，本阶段要恢复。proc.h 中，struct proc 要恢复 cwd 成员，相应的，proc.c 中 userinit()、fork()、exit()都有 cwd 相关的操作，要恢复。

forkret()

proc.c 中 allocproc()把进程的返回地址设置为 forkret()，而 forkret()要求首进程跳回用户态之前先以内核态执行文件系统的初始化 fsinit()。在第 6 阶段我们删除了 forkret()对文件系统的初始化，本阶段要恢复。

系统调用

为了增加 exec()系统调用，须修改 syscall.c、sysfile.c、user/usys.pl。

defs.h

此外，defs.h 自不必说，配合其它模块的增长，增加必要的声明。

测试程序

在第 7 阶段，我们曾经把首进程的 main()函数中 child 1 子进程段落改为：

```
    if (pid == 0) {
        write(3, 0, 0); // 3 is testcode for block r/w, see sys_write()
        while (1) ;
        exit(0);
    }
```

本阶段，我们把该段落改为：

```
    if (pid == 0) {
        char *argv[] = {"execchild1", 0};
        exec("execchild1", argv);
        exit(0);
    }
```

相应的，我们在 user 文件夹下创建 execchild1.c，程序如下：

```
extern int exit(int) __attribute__((noreturn));
extern int write(int, const void *, int);
int main() {
    write(3, 0, 0); // 3 is testcode for block r/w, see sys_write()
```

```
    while (1) ;
    exit(0);
}
```

execchild1.c 将被编译为 execchild1 可执行文件，进而被复制到虚拟磁盘的根目录下。我们终于需要恢复 Makefile 中与用户程序相关的内容。

用户态程序的重新整理

既然我们有了 exec()系统调用和可执行文件，也恢复了 Makefile 中与用户程序相关的内容，那就把用户态程序重新整理一下。但我们强烈建议先把前一节的“测试程序”跑通，毕竟那个程序仅执行一次 exec()系统调用，更利于测试。

前一节的“测试程序”的主要问题是 printf()和 malloc()这样的基础功能都被集成在 initcode.c 中，未被提炼为函数库。我们仿照 xv6 对 user 文件夹的组织方式，对程序做出如下整理：

1）把 printf()和 malloc()分别提炼到 printf.c 和 umalloc.c 中；

2）构建 user.h，对所有系统调用 API 和库函数做集中的声明；

3）user.ld 和 usys.pl 不变；

4）initcode.c 中的三个子进程，前两个分别提炼为 execchild1.c 和 execchild2.c，第三个保持不变。

以上重新整理过的程序由老师提供。Makefile 做出相应调整，如此，最终的虚拟磁盘上除了 README 文本文件，还将有 execchild1 和 execchild2 两个可执行文件，并将被首进程启动执行。

当然，initcode.c 仍须用 user/Makefile 事先编译处理，以获取首进程程序的二进制代码和入口地址，从而复制到 proc.c 中。user/Makefile 中也要增加 printf.c 相关的编译连接命令，以满足 initcode.c 对 printf()的调用。

如果前一节的“测试程序”能跑通，重新整理过的程序也应该没啥问题。

9. sh

本阶段目标：在上一阶段的基础上，实现简易的 sh 程序作为命令行界面。

我们并未实现文件系统的全部功能，应用程序不具备打开文件读写的能力，所有与此相关的内容都要删减。

从效果上看，系统启动后终于拥有了一个命令行界面，可以启动执行用户程序了。

console.c

是时候引入 console.c 了。之前的阶段，我们只是简单测试键盘输入，本阶段我们要让键盘输入的内容被暂存下来，并识别回车符，一旦检测到回车符就要唤醒 sh 进程，并允许 sh 进程读取暂存的键盘输入内容。console.c 承担的正是上述职责。

解除与文件系统的对接

xv6 中，控制台要与文件系统对接，经过 init 进程的初始化，控制台被挂载到/console，于是用户程序可以利用文件系统的 API，“打开”控制台，并开展“读”、“写”。调用关系如下：

1）open()系统调用 -> sys_open() -> 磁盘上查找/console 并在全局打开文件表中打开

2）read()系统调用 -> sys_read() -> fileread() -> devsw[CONSOLE].read() -> consoleread()

3）write()系统调用 -> sys_write() -> filewrite() -> devsw[CONSOLE].write() -> consolewrite()

本阶段我们不提供文件的打开、读写功能，也就不需要把控制台与文件系统对接，我们期望的调用关系如下：

1）用户态不需要 open

2）用户态 printf() –> write()系统调用 -> sys_write() -> consolewrite()

3）用户态 gets() -> read()系统调用 -> sys_read() -> consoleread()

因此做出如下调整：

1）去掉 consoleinit()末尾关于 devsw[CONSOLE]的两行代码。

2）对于 write()系统调用，本阶段不用对 sys_write()做任何修改，第 6、7 阶段已经改好了，即删去对 filewrite()的调用，改为根据文件描述符，若其小于 3 则直接调用 consolewrite(1, p, n)，若等于 3 则执行盘块读写测试 blocktest()。

3）对于 read()系统调用，删去对 fileread()的调用，直接调用 consoleread(1, p, n)。

其它适配

内核的 printf()原本调用控制台的 consputc()，在第 1 阶段我们改为直接调用串口的 uartputc_sync()，现在有了控制台，应恢复。

串口的中断处理函数 uartintr()中，对于收到的字符，原本调用 consoleintr()交由控制台做后续处理，第 1 阶段我们改为直接调用串口的 uartputc_sync()，现在有了控制台，应恢复。

main()中，原本第一步执行 consoleinit()（该函数会调用串口初始化 uartinit()），第 1 阶段我们改为直接调用串口的 uartinit()，现在有了控制台，应恢复。

项目中引入了新的源程序，Makefile 做相应修改，不必多说。

chdir()系统调用

增加 chdir()系统调用，sh 程序执行 cd 命令时要用到。从 xv6 源代码中引入即可，不需要任何修改。

启动 sh

xv6 启动 sh 程序的流程

在 xv6 中，启动过程涉及 3 份用户程序：initcode.S、init.c、sh.c。

我们在第 4 阶段提到最初的用户进程程序经编译后将机器码以字符数组形式定义在 proc.c 的 initcode[]数组中。initcode.S 就是这个最初的用户程序，极简的汇编代码，运行过程中甚至不需要栈。该程序只是简单的执行 exec()系统调用，将自己变身为 init 进程。

init.c 和 sh.c 是最终要出现在虚拟磁盘上的两个可执行程序。init 进程 fork()一个子进程，然后让子进程执行 exec()系统调用变身为 sh，主进程则进入死循环，不断的执行 wait()系统调用。sh 进程是界面进程，同样在一个死循环中读取键盘输入的命令并执行。

initcode

从第 4 阶段以来，我们一直对 proc.c 中的 initcode[]数组进行修改，并在 userinit()的末尾为首进程创建全局数据区和用户栈，使其承载我们的测试程序。本阶段我们终于要回归 xv6 的原始路线，让 proc.c 中的 initcode[]数组恢复 xv6 的原貌，并删除 userinit()末尾的临时程序。这样，系统启动后将自动从虚拟磁盘上读取执行 init 程序。

init.c

从 xv6 的源程序中引入 init.c。可以看到 main()函数开头有打开“console”并两次执行 dup()的操作，上一小节提到过 init 进程的初始化使得控制台被挂载到/console，便是在这里实现的。本阶段当然应该把这一整段程序都删掉。

其它

此前我们一直需要先行编译首进程程序并获得其机器码，因此在 user 目录下准备了 Makefile 文件和 parse_elf 程序。现在终于不再需要了，删掉它们吧。

相应的，项目 Makefile（即 user 目录外的那个 Makefile 文件）要恢复 initcode 相关的内容。观察 xv6 的 Makefile，除了有专门的“$U/initcode”段落，要注意“$K/kernel”和“clean”段落中也提到了 initcode。此外“UPROGS”段落中要增加$U/_init。

sh.c

引入 sh.c，会发现其中大量引用了 ulib.c 中的函数，于是把 ulib.c 也引入。

ulib.c 中应删去 stat()，因为该函数要打开并访问文件，而我们也用不到这个函数。还有个_main()，其注释说明了该函数的目的，对照观察 xv6 的 user.ld 中的 ENTRY()，即可明白其工作原理。我们可以把_main()删掉，只要确保所有应用程序都通过 exit()系统调用实

现退出即可；当然也可以保留该函数，那就要把我们的 user.ld 的 ENTRY()与 xv6 的 user.ld 保持一致。

前面增加了两个系统调用，此处又引入了 ulib.c，user.h 也要随之增加相关函数的声明。

最后聚焦到 sh.c。读一读，万万没想到做到第 9 个阶段，眼看大功告成了，在这里吃了瘪，太难读。这里简单说说其中的要点：

1）程序开头有 cmd 结构体和各种 xxxcmd 结构体，基本上可以理解为抽象父类和各种子类的关系，程序中经常使用父类（cmd 结构）指针指向子类对象。

2）键盘录入的命令经过解析，从宏观到微观形成各种 cmd 结构体的链式结构，例如：

```
$ cmd_a > out_a | cmd_b < in_b
```

宏观上看是个管道命令 pipecmd，该管道命令有两个成员，left 和 right，分别指向管道左边和右边的命令；左边是个重定向命令 redircmd，其 file 成员指向字符串"out_a"，cmd 成员指向一个 execcmd（即 cmd_a）；管道右边同理。

3）读程序的难点是命令的解析，先把 peek()和 gettoken()这两个函数的功能看懂并牢记于心，然后再读各种 parsexxx()函数，会感觉好一点。

本阶段我们的系统不支持进程访问文件，所以诸如管道命令、重定向命令无法实现，应删减。

一不做二不休，干脆把列表命令 listcmd（即用分号连接两个命令）、后台命令 backcmd（即命令末尾用&符号表示后台运行）也去掉，把对命令中括号的支持也去掉，最后留下的只有普通的 execcmd。老师已经准备好了精简后的程序。

项目的 Makefile 做相应的修改，ULIB 要恢复 xv6 的原始样貌，“UPROGS”段落中要增加 $U/_sh。

应用程序

上一阶段的测试程序被命名为 initcode.c，被用做首进程，因而在其 main()的末尾设置了一个死循环确保首进程永不结束。

这一阶段，这个测试程序变成了普通的可执行程序，首先改名为 test.c，然后把 main()末尾的死循环去掉。相应的，Makefile 中，“UPROGS”段落中要增加$U/_test。

其它

trap.c 中，我们曾在定时器中断处理 clockintr()中增加周期性的“T”字符输出。现在有了更美观的界面，就不要让老东西干扰我们的美丽新生活了。

同之前的每个阶段一样，defs.h 须配合其它内核模块的增长，增加必要的声明。

针对本阶段增加的系统调用，usys.pl 和 syscall.c 要做配套修改。

结束

完成所有修改并调试通过，我们终于有了可以交互的界面。

注意此时虚拟磁盘上有以下文件：README、init、sh、test、execchild1、execchild2，其中 test 会通过 exec()系统调用启动 execchild1 和 execchild2。由于没有访问文件的能力，所以我们并未实现 ls 命令，只能直接敲击可执行程序的名称。你可以敲击“test”或“execchild2”，不过不要敲击“execchild1”，因为这个程序里面有死循环。

做到这个阶段，我们距离完整的 xv6 系统只缺文件访问功能了。最后这块拼图就不再当作实验任务了，留给同学们自行阅读相关程序吧，因为你做完后得到的基本就是原始的完整的 xv6 源程序，已经没有验收检查的价值。