继续阅读前，建议先阅读“xv6-内核初始化部分-代码详解”，先大概了解系统的初始化过程。

bio.c

`void binit(void)`

首先，初始化磁盘缓存锁，然后，将 10 块磁盘缓存组成链表，binit

`static struct buf* bget(uint dev, uint sector)`

首先，遍历磁盘缓存表，检测数据块是否已缓存，若已缓存，检测其是否正在被使用，若否则返回该缓存，若是则等待其使用完成，若数据块未被缓存，则从缓存表中找一个合适的缓存（不 B_BUSY，也不 B_DIRTY），对其进行设置后返回。

回到 bread 定义，接下来，查看该缓存是否已从磁盘读出，若否，则调用 iderw 将缓存从磁盘读出，最后，返回数据块的缓存，

`struct buf* bread(uint dev, uint sector)`

函数从指定磁盘（dev）的指定扇区（sector）获取数据块。首先调用 bget 获取数据块的缓存：

`void bwrite(struct buf *b)`

函数将指定缓存同步到磁盘。首先，检测缓存是否正在被使用，若是则 panic，设置缓存脏标志位，调用 iderw 将缓存同步到磁盘，

`void brelse(struct buf *b)`

函数释放指定缓存，并将其移到磁盘缓存表的头位置，最后叫醒等待该缓存的进程，

console.c

`static void printint(int xx, int base, int sign)`

首先，检测是否是负数，若是则将数转为正数，接下来，将数转为字符串（倒序），然后，如果是负数，在字符串最后添加一个负号，最后，调用 consputs 以倒序的方式打印出字符串。

`void cprintf(char *fmt, ...)`

函数打印格式化信息到终端（串口和本地显示器）。首先，获取格式化信息第一个本体的指针（本体是指除 fmt 字符串之外的，最终替换 fmt 字符串中形体的值），接下来依次解析 fmt 中的每个字符：

- 若不是字符'%'，则调用 consputc 将其直接打印；
- 若是字符'%'，则根据其后面一个字符（形体）分情况处理：
  - 若形体为'd'，则调用 printint 以 10 进制的形式打印对应的本体值；
  - 若形体为'x'或'p'，则调用 printint 以 16 进制的形式打印对应的本体值；
  - 若形体为's'，则调用 consputc 逐一打印本体（此时是一个字符串）的值；
  - 若形体为'%'，则打印形体（此时'%'就是本身，不代表形体的开头）；
  - 若是除以上几个字符外的字符，则打印'%'，表示无法处理；

`void panic(char *s)`

首先，关中断，然后，打印出错 cpu id 和出错信息，接下来，调用 getcallerpcs 获取函数调用栈信息，并将其打印出来，最后，置 panicked 值（会让其他 cpu 陆续入死循环），当前 cpu 进入死循环。

```
static void cgaputc(int c)
```

首先，获取当前光标的位置，先获得光标位置的高 8 位信息（光标位置高 8 位寄存器的 index offset--0xE（十进制是 14）写入 CRT 微控制器的 Data Register--port 0x3D4，然后从 CRT 微控制器的 Index Register--port 0x3D5 获取光标位置高 8 位寄存器的数值，详见 Display Hardware），然后，再获取光标位置的低 8 位信息（光标位置低 8 位寄存器的 index offset--0xF（十进制是 15）），接下来，若待打印的字符是换行符（'\n'），则将光标移到下一行的开头（设置的显示模式应该是 80x50），若待打印的字符是退格符（BACKSPACE），则将光标左移一个字符，其余情况，就将字符加上属性打印出来（xv6 使用的 CGA 显示，其映射到内存的地址是 0xB8000-0xBFFFF，其使用 2 个 byte 代表一个字符，低 8 位为字符的 ascii 码，高 8 位为显示属性，在此，高 8 位被设置为 0x07，意为前景是黑色，背景是白色，详见 Display Hardware），接下来，检测打印完当前字符后，是否需要 scroll（屏幕满了，必须要将所有显示内容整体上移一行），若是则将显存的第一行（头 80 个字符）删除，并将其余行前移一行，最后，刷新光标的位置并将光标当前位置显示为一个黑色的空格。

```
void consputc(int c)
```

函数将值c同时在串口和主机屏幕上打印出来。首先，检测系统是否已崩溃（panicked），若是则关中断进入死循环，接下来，检测待输出的字符是否为退格符（效果是向左删除一个字符，xv6 使用 ctrl+H 表示或者 delete 键表示删除一个字符，后面的章节会详细说明），若是则调用 uartputc 向串口输出'\b',' ','\b'三个字符产生左删的效果（为什么这样做，我没深入研究），若否则将向串口输出这个字符，接下来，调用 cgaputc 将字符在屏幕上打印出来。

```
void consoleintr(int (*getc)(void))
```

首先，调用 getc 逐个获取需要打印的值并解析：

- 若值是 ctrl+P，则调用 procdump 打印所有的进程信息；
- 若值是 ctrl+U，则删除当前输入缓存（input）中和当前已输出（串口和屏幕）的当前行；
- 若值是 ctrl+H 或者 0x7f（backspace 的 asic ii 码），则删除当前输入缓存（input）中和当前已输出（串口和屏幕）的前一个字符；
- 若是其他值，则调用 consputc 输出该值，然后，检测值是否为换行符或 EOF（'\n' 或者 ctrl+D），若是则唤醒等待当前输入缓存的进程；

```
int consoleread(struct inode *ip, char *dst, int n)
```

首先，查看是否还有未读的 buffer（input.w >input.r），若否则循环等待（等待过程中进程被杀则返回），若是，则将 buffer 读出，若读出的字符为 EOF（ctrl+D）或换行符（'\n'），则完成读取，返回读取的字符数。

```
int consolewrite(struct inode *ip, char *buf, int n)
```

首先解锁终端设备文件对应的 inode（iunlock 和 ilock 在以后的文件系统详解中一并解释），获取终端锁，将 buf 中的内容一个字节一个字节的通过 consputc 输出到终端，接下来，释放终端锁，给终端设备文件 inode 加锁，返回输出的字符数。

```
void consoleinit(void)
```

首先，初始化终端锁和输入锁，设置终端设备的读写函数，然后，使能终端锁（locking=1

为使能，并不是上锁)，最后，打开键盘中断（单 cpu 设备调用 picenable，多 cpu 设备调用 ioapicenable，并将中断绑定到 bsp--APIC ID 为 0)。

exec.c

```
int exec(char *path, char **argv)
```

继续之前，建议先阅读 xv6 book rev7--chapter 2。

首先，获取文件（path）的 inode，接下来，解析 inode 并将信息存储到 elf，然后，验证 elf 头，接下来，创建一个页目录，然后，将程序从磁盘一段一段的读取到内存（页目录），接下来，分配两个空白页，第一页设置为不可达（inaccessible，防止访问越界），第二页作为进程栈使用，然后，将程序的执行参数全部压入栈，接下来，将虚返回地址，参数数量，每个参数地址和第一个参数地址的地址（**argv）也压入栈，然后，将程序名保存到 proc 结构体中，接下来，设置 proc（页目录，程序入口地址，栈地址等），最后，载入程序镜像，释放旧镜像所占内存。

file.c

`void fileinit(void)`

初始化文件表（ftable，所有打开文件的列表）的锁。

`struct file* filealloc(void)`

从文件表中找到一个可用的成员（file 的 ref 成员为 0 的 file），将其 ref 成员置 1 并将其返回。

`struct file* filedup(struct file *f)`

将文件（f）的 ref 加 1，并将其返回。

`void fileclose(struct file *f)`

首先，将文件（f）的 ref 减 1，若其 ref 不为 0，则返回，接下来，置文件的 ref 为 0，置 type 为 0 (FD_NONE)，接下来，检测 ff (f 的复制品)，若 ff 的类型为管道 (FD_PIPE)，调用 pipeclose 将其 pipe 成员关闭，若 ff 的类型为 inode，将 inode 的引用减 1，并将改动写到磁盘（通过日志机制--××_trans)。

`int filestat(struct file *f, struct stat *st)`

调用 stati 将 f 的 inode 状态存入 st。

`int fileread(struct file *f, char *addr, int n)`

若文件类型是管道，调用 piperead 将 f 的 n 个字节读到 addr，若文件类型是 inode，调用 readi 将 f 文件偏移 off 开始的 n 个字节读到 addr，并更新 off 成员，函数返回读到的字节数。

```
int filewrite(struct file *f, char *addr, int n)
```

若文件类型是管道，调用 piperead 将 addr 的 n 个字节写到 f，若文件类型是 inode，通过日志机制调用 writei 将 addr 开始的 n 个字节写到 f 文件偏移 off 的位置。

fs.c

继续之前，建议先阅读 xv6 book rev7--chapter 6。

```
void readsb(int dev, struct superblock *sb)
```

首先，调用 bread 将指定设备的 superblock 信息（位于指定 dev 的第 2 个扇区--sector 1）读到缓存 bp，然后，将缓存中的 data 成员读到 sb 结构体，最后释放该缓存。

```
static void bzero(int dev, int bno)
```

首先，调用 bread 将指定设备的指定扇区读到缓存 bp，然后，将 bp 的数据清零，接下来，调用 log_write 将清零操作同步到磁盘，最后释放 bp。

```
static uint balloc(uint dev)
```

首先，调用 readsb 获取指定设备的 superblock 信息（sb），接下来，进入一个二层循环，第一层调用 bread 依次获取每块 bit map block （bit map blocks 位于 inode blocks 之后，详见 xv6 book rev7--chapter 6--Code: Block allocator），第二层依次查看 bit map block 中的每个 bit，找到一个为 0 的 bit，将其置 1（调用 log_write 同步到磁盘），将 bit 对应的 block 清零（调用 bzero），最后，返回该 block 在磁盘中的位置（b+bi）。

```
static void bfree(int dev, uint b)
```

首先，查找待释放块（b）对应的 bit（bit map 中的 bit）是哪个 bit map block 中的哪个 bit，然后将改 bit 置 0，最后将改动同步到磁盘。

```
void iinit(void)
```

初始化 inode 缓存的锁，

```
struct inode* ialloc(uint dev, short type)
```

首先，调用 readsb 获取指定设备的 superblock 信息（sb），接下来，遍历所有 inode，找到一个 free 的 inode，然后，将其数据结构清零，并设置其 type，并将改动同步到磁盘，最后，缓存该 inode 到内存并返回。

```
void iupdate(struct inode *ip)
```

首先，获取该 inode 的磁盘缓存（dinode），接下来，将该 inode 在内存中的复制（inode）同步到磁盘缓存中。

```
struct inode* iget(uint dev, uint inum)
```

首先，遍历 inode 缓存表，若在表中找到该 inode，则 ref 加 1，并返回该 inode，若未找到，则找一个空的 inode，将部分信息（dev 和 inum）填进去并返回。

```
struct inode* idup(struct inode *ip)
```

将对应的 inode 的 ref 加 1 并返回。

```
void ilock(struct inode *ip)
```

首先，等待 inode 状态变成不忙（I_BUSY），然后，将其置忙，接下来，查看其是否有效（I_VALID，已从磁盘读出到内存），若否，则将其从磁盘同步到缓存。

```
void ilock(struct inode *ip)
```

将 inode 置不忙，唤醒相关进程。

```
void iput(struct inode *ip)
```

首先，检测 inode 是否可以释放（ref，flag，nlink 值），若是，则释放其占用的资源（磁盘及缓存等），并唤醒相关的进程，若否，则 ref 减 1。

```
void iunlockput(struct inode *ip)
```

先解锁 inode，再 put。

```
static uint bmap(struct inode *ip, uint bn)
```

首先，若待创建的块是直接块（NDIRECT），将创建的块号直接存到 addrs 数组中，若是间接块（NINDIRECT），则首先分配一块 block 用来存储这些块号（addrs 数组的最后一个值也是一个块号，但是这个块里存的不是文件数据，而是其他块的块号），接下来，将创建的块号存到间接块号存储块（上面创建的那个专门存储间接块块号的块）里，并同步到磁盘，最后，返回块号。

```
static void itrunc(struct inode *ip)
```

释放 inode 里所有的直接块，间接块和存储间接块号的块。

```
void stati(struct inode *ip, struct stat *st)
```

获取 inode 相关的状态信息。

```
int readi(struct inode *ip, char *dst, uint off, uint n)
```

首先，检测 inode 类型是否是设备（T_DEV），若是，则调用指定设备（major）的 read 函数来读取数据到 dst，并返回读取到的字节数。接下来，将 inode 中指定的数据（off 开始的 n 字节）逐块（这里的块并不一定是 512 字节）拷贝到 dst 中。

```
int writei(struct inode *ip, char *src, uint off, uint n)
```

首先，检测 inode 类型是否是设备（T_DEV），若是，则调用指定设备（major）的 write 函数来从 src 读取数据到 inode。接下来，将 src 中的数据（n 字节）逐块（这里的块并不

一定是 512 字节）拷贝到 inode 缓存（inode 指向的数据块缓存）中，并同步到磁盘，最后，将 inode 的 metadata 部分也同步到磁盘。

```
int namecmp(const char *s, const char *t)
```

比较两个字符串是否相同。

```
struct inode* dirlookup(struct inode *dp, char *name, uint *poff)
```

遍历目录 inode 的文件数据（以 dirent 的 size 为单位，目录（T_DIR）类型的 inode 的文件数据里存的都是 dirent 结构体）中的所有目录入口（directory entry），找到指定文件名（name）对应的目录入口，接下来，告知调用者该目录入口在 dp 文件数据中的偏移（poff），最后，调用 iget 获取 inum 对应的 inode，并将其返回。

```
int dirlink(struct inode *dp, char *name, uint inum)
```

首先，确认该文件名在指定目录（dp）中不存在，接下来，从指定目录的中找到一个空的目录入口，然后，设置文件名和 inode 号（inum），最后，将改动同步到磁盘。

```
static char* skepelem(char *path, char *name)
```

分割路径名：返回第一层目录名和剩余部分。

```
struct inode* namex(char *path, int nameiparent, char *name)
```

首先，获取路径（path）对应的 inode（区分绝对路径和相对路径），然后，一层一层的往下找，如果是要找路径的父目录（nameiparent 为 1），则在倒数第二层返回 inode，否则，一直找到最后一层，返回 inode。

```
struct inode* namei(char *path)
```

返回路径（path）对应的 inode。

```
struct inode* nameiparent(char *path, char *name)
```

返回路径所在目录的 inode 和路径最底层的文件名（通过 name 返回，这里的文件也可能是一个目录）。

ide.c

继续阅读前，建议先浏览下 IO_devices 和 ATA PIO Mode 文档。

```
static int idewait(int checkerr)
```

首先，轮询磁盘状态寄存器，直到磁盘状态为 READY（磁盘状态寄存器的端口地址是 0x1F7，其中的 bit6 是 READY 位，bit7 是忙位，详见 IO_devices），若参数 checkerr 为 1，则查看磁盘状态是否有错误（磁盘状态寄存器的 bit0 和 bit5 是错误位），有错返回-1，其余返回 0。

```
void ideinit(void)
```

首先，初始化磁盘锁，接下来，使能磁盘中断（在多 cpu 设备上，将磁盘中断绑定到最后一个 cpu 上--cpu id 最大的那个 cpu），然后，调用 idewait 等待磁盘启动完成，接下来，检测设备是否存在第二块磁盘分区（disk0 是第一块，disk1 是第二块），将端口 0x1f6 置 0xf0（0xe0 | (1<<4)）来选择 disk1，等待 0x1f7 的值变化成非 0 值（非 0 表示 disk1 存在），若 0x1f7 依然是 0，则 disk1 不存在，最后恢复到 disk0 的选择（将端口 0x1f6 置 0xe0）。

```
static void idestart(struct buf *b)
```

首先，等待磁盘就绪，然后，使能中断（0 -> 0x3f6），接下来，设置扇区数（0x1f2，这个设置看不懂），设置扇区的地址（0x1f3，0x1f4，0x1f5，0x1f6 的低 4 位）和设备号（0x1f6 的 bit4），然后，查看是读还是写操作，写操作就将数据写到数据端口（写之前先设置 0x1f7），读操作就直接设置 0x1f7。

```
void ideintr(void)
```

首先，获取当前 ide 队列（idequeue）的第一个 buf（b），接下来，判断 buf 是否是待读状态（不是 B_DIRTY）,若是待读，就将 ide 数据端口的数据读到 buf，然后，更新 bug 的状态(可用,不脏),并唤醒等待 buf 的进程,最后,进入下一个 buf 的 ide 操作(idestart)。

```
void iderw(struct buf *b)
```

首先，将 buf 添加到 ide 队列的末尾，如果 ide 队列只有一个 buf 待处理，直接通知 ide 处理 buf（idestart），否则，进程进入 sleep 状态等待 buf 被处理完成。

ioapic.c

在继续阅读之前，建议先浏览 64-ia-32-architectures-software-developer-vol-3a-part-1-manual（以下简称 Manual） 的第 10.1 节及之后相关章节和 APIC-OSDev Wiki 的相关内容。

```
static uint ioapicread(int reg)
```

读指定寄存器中的数据（先选择寄存器，然后从中读取数据）。

```
static uint ioapicwrite(int reg)
```

往指定寄存器中写数据（先选择寄存器，然后往里写数据）

```
void ioapicinit(void)
```

首先，检测当前设备是否为多 cpu 设备，若否则退出，接下来获取 ioapic 的地址(IOAPIC，ioapic 被 map 到了内存的 0xfec00000)，然后，获取重定向入口的最大数量 maxintr（maximum amount of redirection entries，其位于 0x01 寄存器的 bits 16-23)，然后，获取 IO APIC 的 ID（其位于 0x0 寄存器的 bits 24-27)，然后检测获取的 IO APIC 的 ID 是否与之前 MP 配置表中获取的 ID 相同，接下来，设置每一个中断（设置中断向量--入口的低 8 位，关中断--入口的 bit16（之后会打开)，将入口的高 32 位全部清 0，意为将中断发给 APIC ID 为 0 的 cpu，就是自己--bsp，详见上述 Wiki 中的 IO APIC Registers 部分)。

```
void ioapicinit(int irq, int cpunum)
```

使能指定 cpu 的指定中断（设置中断向量--入口的低 8 位，选择转发 cpu--入口的高 8 位，详见上述 Wiki 中的 IO APIC Registers 部分)。

kalloc.c

`void kinit1(void *vstart, void *vend)`

首先，关闭 kmem 锁的使用（当前不使用 kmem 锁，后面会使用，注意不是释放锁，是关闭锁），然后，将指定范围的内存加入内核空闲内存表。

`void kinit2(void *vstart, void *vend)`

首先，将指定范围的内存加入内核空闲内存表，最后，开启 kmem 锁。

`void freerange(void *vstart, void* vend)`

首先调用 PGROUNDUP 获取 vstart 地址之后的第一个页对齐的地址（空闲内存表中必须都是页对齐的一页一页的地址），然后调用 kfree 将指定范围的内存块逐页添加到内核空闲内存表中。

`void kfree(char *v)`

首先将这一页都填上垃圾数据（每个字节都赋予 1），然后，将这一页地址加到内核空闲内存表的头部。

`char* kalloc(void)`

获取内核空闲内存表的头元素，更新内核空闲内存表，返回上述元素。

kbdgetc.c

在继续阅读之前，建议先浏览 The PC keyboard Scan Codes，PS_2 Keyboard - OSDev Wiki 和 Keyboard scancodes。

`int kbdgetc(void)`

首先，获取键盘状态（从端口 0x64--键盘状态寄存器），确认 input 缓存中有数据（没有就返回-1），然后，从缓存中获取数据（从端口 0x60--数据端口），接下来，判断读到的数据是否是 escape scancodes（0xe0，逃脱扫描码，作为一个扩展码，其只是起扩展扫描码的作用，详见上述文章），若是则做个记录（记在 shift 里）就返回，然后，判断读到的数据是否代表按键抬起（大于 0x80 的值都代表按键抬起，详见上述文章），若是，接下来，判断数据是否是扩展码（之前一个码是 0xe0，记录在 shift 里），若是，则不更改数据，若不是，则 mask 掉数据的高 1 位（这样做主要是因为 shift 键，alt 和 ctrl 键的左右区别只是有无扩展码，但左右 shift 键都没有扩展码，所以，将其减去 0x80，以更简洁的表示，详见上述文章），然后，将抬起按键对应的状态从 shift 变量 mask 掉（主要是 ctrl，alt，shift 键和扩展码）就返回，接下来，判断数据是否是扩展码，若是，则将数据 or 0x80（为了 map 的方便），并将扩展码从 shift 变量中 mask 掉，接下来，将 ctl，alt，shift or 进 shift 变量(如果这些键处于按下的状态)，然后，将 LOCK xor 进 shift(CAPSLOCK，NUMLOCK，SCROLLLOCK 开关)，接下来，从对应的 map 中获取按键数据（根据 ctl，shift 的按键状态对应不同的 map），然后，判断 CAPSLOCK 是否打开，若是，则将按键数据的大小写调换（若是字母数据的话），最后，返回最终的按键数据。

`void kbdintr(void)`

键盘中断触发时，调用 consoleintr 来输出键盘的按键。

lapic.c

在继续阅读之前，建议先浏览 Manual 的第 10.1 节及之后相关章节和 APIC-OSDev Wiki 的相关内容。

```
static void lapicw(int index, int value)
```

往本地 APIC 的指定寄存器（通过初始地址 lapic 偏移 index 选择）写值（value），然后，等待写操作结束（写操作结束后，才能读 APIC 的寄存器）后返回。

```
void lapicinit(void)
```

首先，检测本地 apic 基址是否存在，不存在就返回， 然后，设置伪中断向量寄存器 spurious interrupt vector register（SVR），使能 local APIC（SVR 的第 9 位--bit 8 置 1），并设置中断向量的数量（SVR 的低 8 位） （spurious-interrupt vector） （详细信息查看 Intel 64 and IA-32 Architectures Software Developer’ s Manual（以下简称 Manual）第 10.4.3 节和 APIC-OSDev Wiki 中的相关内容），接下来，设置计时器相关寄存器（详见 Manual 第 10.5.4 节内容）：

- 设置 Divide Configuration Register（TDCR），APIC 计时器的频率就是总线频率除以 TDCR 中的设置值（不是数值，有一个转换关系，详见 Manual 图 10-10），xv 设置的是 0xB（也就是 111，根据 Manual 中的转换关系转换后为 1，也就是 timer 的频率与总线频率相同）；
- 设置 LVT Timer Register（Timer），将计时器设置为周期性的（PERIODIC），这样一来，当 Current Timer Register（TCCR）递减到 0 时，会自动从 Initial Count Register（TICR）重新读取数值，将计时器的 Interrupt Vector 指向 IRQ_TIMER，这样一来，计时器计数到 0 时会产生一个指向 IRQ_TIMER 的中断；

- 设置 Initial Count Register（TICR）为 10000000，意为，以总线频率，起始值为 10000000 递减 TCCR，TCCR 到 0 时，产生一个指向 IRQ_TIMER 的中断，同时 TCCR 会自动从 Initial Count Register（TICR）重新读取数值（恢复为 10000000）；

接下来，关闭逻辑中断线，屏蔽 LINT0 和 LINT1，在多 CPU 设备中，LINT0 和 LINT1 不被用来将中断传给 CPU，而是将中断通过 IO APIC 传给 CPU，详见 Manual 第 8.7.13.4 节。然后，检测当前设备是否提供 Performance Counter Overflow Interrupts 入口，若提供，则屏蔽该 APIC 中断，详情可自行 google。然后，设置 IRQ_ERROR 中断向量。然后重置错误状态寄存器（Error Status Register，ESR），详见 Manual 第 10.5.3 节。接下来，重置中断结束寄存器（End Of Interrupt，EOI），中断程序在返回前必须设置 EOI，以便告知本地 APIC 中断处理已结束，便于其处理下一个中断，详见 Manual 第 10.8.5 节。然后，广播 Init Level De-assert 到所有本地 APICs 同步它们的 APIC ID 为仲裁 ID（Arbitration ID），详见 Manual 第 10.6.1 节和 10.7 节。最后，将任务优先寄存器值设置为最低（0），这样所有的中断都可以被 APIC 获取，详见 Manual 第 10.8.3.1 节。

```
int cpunum(void)
```

若在使能中断（FL_IF）的情况下，调用函数，则打印一些信息（为什么要这样做，没搞明白），接下来，将当前的本地 APIC ID（其实就是 cpu id，其位于本地 APIC ID 寄存器的高 8 位，详见 Manual 第 10.4.6 节）返回。

```
void lapiceoi(void)
```

在中断处理结束，中断例程返回之前，须调用该函数来通知本地 APIC 中断已处理完毕（这样一来，本地 APIC 就可以处理下一个中断）。

`void microdelay(int us)`

并没有实现该函数，用来做一个指定时间的延时。

`void lapicstartap(uchar apicid, uint addr)`

这是 intel 官方的 cpu 启动算法。(包括 bsp 和 ap，详见 CMOS Memory Map 和 Halting_Restarting a BSP - Intel Community 及 Intel Multi Processer Specification)。

log.c

继续之前，建议先阅读 xv6 book rev7（以下简称 book）--chapter 6。

`void initlog(void)`

实现获取磁盘 superblock 信息，接下来，根据 superblock 中的信息初始化日志结构体（日志块位于磁盘的尾部位置，详见 book 第 6 章），最后，从 log 中恢复文件系统（不一定真的恢复）。

`static void install_trans(void)`

遍历日志，将日志中的每一块数据缓存拷贝到它真正的目的块缓存，并同步到磁盘。

`static void read_head(void)`

首先，将日志头块（log header block）从磁盘读出，然后将其中的信息复制到其在日志结构体的镜像中（lh）。

`static void read_head(void)`

与 read_head 相反，将日志头从日志结构体中复制到日志头块。

`static void recover_from_log(void)`

首先，从磁盘读出日志头，然后，根据日志头恢复未同步的数据，最后，将日志清零（只是简单的设置了 lh.n），最后将日志头同步到磁盘。

`void begin_trans(void)`

只是获取日志的使用权（置忙）。

`void commit_trans(void)`

首先，将日志头第一次同步到磁盘（这时的日志头包含了需要拷贝的数据的信息），然后，根据日志头将日志中的数据块拷贝到其目的块，最后，清空日志，并将日志头第二次同步到磁盘（这时的日志头为空），接下来，归还日志的使用权（置闲），并唤醒等待使用日志的进程。

```
void log_write(struct buf *b)
```

首先，遍历日志头，检测待写入日志的数据块的目的块是否已存在于日志中（大概率存在一个 trans 中对同一个磁盘块进行多次写），接下来，将带写入日志的数据块写入到日志里。

main.c

包含了系统初始化的过程，在“xv6-内核初始化部分-代码详解”中有详细分析。

mp.c

包含了多 cpu 设备的 mp 初始化过程，在“xv6-内核初始化部分-代码详解”中有详细分析。

picirq.c

代码通过设置 io 端口来配置 pic，都是 8259A 的的标准流程，与 xv6 内核关系不大，故不深入讲解，有兴趣可以自行 google 其细节。

pipe.c

```
int pipealloc(struct file **f0, struct file **f1)
```

首先，创建两个文件，接下来，创建一页内核内存供管道使用，并初始化管道结构体，然后，设置 f0 为只读管道文件，设置 f1 为只写管道文件，二者共享这个管道。

```
void pipeclose(struct pipe *p, int writable)
```

首先,若是管道的写端文件关闭,则唤醒等待读管道的进程,若是管道的读端文件关闭，则唤醒等待写管道的进程，最后，若管道两端文件都关闭了，则释放管道占用的资源。

```
int pipewrite(struct pipe *p, char *addr, int n)
```

首先，循环查看管道是否已满（已写减已读等于空间上限），若管道已满，再查看读端文件是否还在，若否就返回，然后，唤醒待读管道的进程，进入睡眠，若管道不满，则将数据写入管道，最后，唤醒待读管道的进程，返回已写的字节数。

```
int piperead(struct pipe *p, char *addr, int n)
```

首先，循环查看管道是否空（已写字节数等于已读字节数），若管道为空，再查看写端文件是否还在，若否就返回，然后，进入睡眠，若管道不空，则将数据从管道读出，最后，唤醒待写管道的进程，返回已读的字节数。

proc.c

`static struct proc* allocproc(void)`

首先，从进程表中找一个空闲的进程结构体（struct　proc），找不到就返回 0，找到了就初始化它：

- 将进程状态设置为孵化态（正在初始化）；
- 设置进程号（pid）；
- 给进程分配内核栈，分配失败就返回 0；
- 在内核栈中设置 trapframe；
- 在内核栈中设置 trapret；
- 在内核栈中设置寄存器；
- 设置程序计数寄存器（eip）指向 forkret；

这样设置后，未来该进程被进程调度器启动时，会依次执行 forkret（跳转到 trapret），trapret（从 trapframe 中恢复用户寄存器），最后回到用户空间。

`void userinit(void)`

首先，调用 allocproc 创建一个进程句柄，接下来，给进程创建页目录（调用 setupkvm，创建并初始化其内核部分），然后，设置页表的用户部分（inituvm），接下来，设置进程占用的内存大小（sz），然后设置进程的 trapframe：

- 设置 tf 的 cs 的段类型为 SEG_UCODE；
- 设置 tf 的 ds，es，ss 的段类型为 SEG_UDATA；
- 打开 eflags 的中断使能（FL_IF）；
- 设置 esp（栈指针）指向程序镜像的末尾；

- 设置 eip（指令指针）指向程序镜像的开头；

接下来，设置进程名，进程的工作目录，最后，将进程的状态设置为 RUNNABLE（可被进程调度器调度执行）。

```
int growproc(int n)
```

首先，根据 n 的值来选择是给进程添加内存空间还是减少内存空间，最后，重新载入进程的内存信息。

```
int fork(void)
```

首先，给子进程分配一个进程槽（proc），然后，将父进程的页目录复制给子进程，若复制失败，释放进程槽占用的资源，并返回，接下来，设置进程的占用空间，父进程，和 trapframe，将子进程的返回值置 0（设置 trapframe 的 eax 寄存器值），然后，复制父进程所有打开的文件给子进程（实现了进程间通信），并复制父进程的当前目录给子进程，接下来，将子进程的状态改为可运行（RUNNABLE），最后，复制父进程的进程名给子进程，并返回子进程的进程号。

```
void exit(void)
```

首先，initproc 不能退出（没人收尸），然后，关闭进程所有打开的文件，释放目录文件引用（运行目录），接下来，遍历进程表，找到该进程的所有子进程，并交由 initproc 收养，顺便收个尸，接下来，将该进程状态改为已死（ZOMBIE），将 cpu 交还进程管理器。

```
int wait(void)
```

首先，遍历进程表，找到自己的孩子，若孩子已死，释放其占用的所有资源就返回，若自己没孩子，就返回，若自己孩子没一个死的，就等一个孩子死才返回。

`void scheduler(void)`

每一个 cpu 都运行有一个 scheduler，首先，允许中断，然后，从进程表中找到第一个状态为 RUNNABLE 的的进程，接下来，载入这个进程的虚拟内存（switchuvm），接下来设置进程状态为 RUNNING，然后，保存当前各寄存器值，载入进程的寄存器值（swtch），至此，进程开始执行，到时间片用完时，其主动放弃 cpu 占用，cpu 重新回到 scheduler，继续执行 switchkvm，重新载入内核页表，进入下一个调度循环。

`void sched(void)`

将 cpu 控制权交还给 scheduler。

`void yield(void)`

时间片用完，将 cpu 控制权交还给 scheduler，进程进入可执行状态等待下一次调度。

`void forkret(void)`

若是系统启动后第一次调用，则初始化 log。

`void sleep(void *chan, struct spinlock *lk)`

首先，将持有的锁（lk）释放掉，然后申请进程表锁（为什么要拿个锁睡觉，这块有点难以理解，是这样的，进程表锁是一个全局变量，整个内存中只有一个，其被所有 cpu 共享，各 cpu 的进程调度器工作的时候必须持有进程表锁--因为要改变进程状态，进程要将 cpu 控制权交还进程调度器，必须先持有进程表锁，只有这样，当进程调度器重新获权的时候才能与之前的状态一致--持有进程表锁，这有点违反谁上锁谁解锁的规则，有点黑客的感觉，但这很有效，总之，在进程调度这个世界里，进程表锁是可以跨进程操作的，但必须保持一个原则--我走的时候咋样，回来的时候必须还那样--进程切换，必须保证锁状态的一致

性)，接下来，设置唤醒条件（chan）和进程状态（SLEEPING)，然后，交权给进程调度器，被 wakeup 后，进程清理唤醒条件，并重新获取睡眠前持有的锁。

`static void wakeup1(void *chan)`

遍历进程表，将所有唤醒条件为 chan 的睡眠进程唤醒（设置为 RUNNABLE)。

`void wakeup(void *chan)`

首先获取进程表锁（因为接下来要改变进程状态)，然后唤醒符合条件的进程。

`int kill(int pid)`

遍历进程表，找到指定 pid 对应的进程，将其的 killed 值置为 1（进程返回用户空间后，看到该值会自行了断)，然后，将进程状态改为可执行（叫醒它，让它自杀)，最后，返回。

`void procdump（void）`

打印所有进程信息。

spinlock.c

包含了对锁的各项操作，在“xv6-内核初始化部分-代码详解”中有详细分析。

syscall.c

建议先阅读 xv6 book rev7（以下简称 book）--chapter 3。

`int fetchint(uint addr, int *ip)`

首先，检测地址是否越界（因为 addr 里存的应该是一个 int，所以这个 addr 及之后的 3 个地址都不能越界，进程的内存界限是 proc->sz-1），然后，从 addr 中取出 int 值。

`int fetchstr(uint addr, char **pp)`

首先，检测地址是否越界（先检测字符串第一个字符的地址），然后，检测其他字符是否越界，若都不越界，就返回 addr 中的字符串和字符串长度。

`int argint(int n, int *ip)`

获取该函数的第 n 个参数（应该是 int 型），并返回（这里的 esp+4+4*n，详见 book--chapter3--Code: System calls）。

`int argptr(int n, char **pp, int size)`

获取该函数的第 n 个参数（是一个指针，大小是 32bit，所以可以用 argint 来获取），该指针及之后的 size-1 个地址都不能越界，最后，返回该指针（详见 book--chapter3--Code: System calls）。

`int argstr(int n, char **pp)`

获取该函数的第 n 个参数（是一个指针，大小是 32bit，所以可以用 argint 来获取），然后，确认该指针合法（是一个 str，同时不越界）后返回。

`void syscall(void)`

首先，获取系统调用号，然后，找到调用号对应的系统调用函数，进入系统调用函数，从系统调用返回后，将返回值存入 eax，返回用户空间。

sysfile.c

`static int argfd(int n, int *pfd, struct file **pf)`

首先，获取该函数的第 n 个参数（是一个 file descriptor，大小是 32bit，所以可以用 argint 来获取），然后，返回 file descriptor 和 file 指针。

`static int falloc(struct file *f)`

将一个 file 指针存到进程的打开文件列表里（ofile）。

`int sys_dup(void)`

首先，获取第 0 个参数对应的 file 指针，然后，给这个 file 指针新建一个 file descriptor，最后，复制这个 file（调用 filedup），返回 file descriptor（注意，这两个 fp 指向同一个打开的文件，所有的操作都是相互可见的）。

`int sys_read(void)`

首先，获取 3 个参数，然后读文件（调用 fileread）。

`int sys_write(void)`

首先，获取 3 个参数，然后写文件（调用 filewrite）。

`int sys_close(void)`

首先，获取参数（fd），然后，将进程打开文件列表中该 fd 对应的值清 0（释放该 fd），最后关闭文件（调用 fileclose）。

`int sys_fstat(void)`

首先，获取 2 个参数，然后返回文件状态（调用 filestat）。

```
int sys_link(void)
```

首先,获取两个参数(src 路径和 dst 路径),然后,从 src 路径(old)获取对应的 inode,并确认 inode 不是目录型,接下来,更新 inode 的引用数(nlink)并更新到磁盘(log 区),然后,获取 dst 路径(new)的父目录 inode,并在该目录下创建一个文件链接(directory entry),最后将 log 区数据真正写到目的磁盘块(commit_trans)。

```
static int isdirempty(struct inode *dp)
```

遍历目录包含的所有 directory entry,所有 entry 都没有关联 inode,返回 1,否则返回 0。

```
int sys_unlink(void)
```

首先,获取参数(需要删除的文件路径名),并获取其父目录 inode,然后,确认该路径名确实对应着一个文件(dirlookup)并获取该文件对应的 inode,接下来,确认上述 inode 合法(nlink>0),若 inode 是目录,确认其下没有文件存在,然后,在父目录 inode 中删除 inode 对应的文件链接(directory entry),若 inode 是一个目录,则对其父目录 inode 的引用减 1(因为子目录的“..”链接指向父目录),接下来,将 inode 的引用减 1,并更新到磁盘(同样的,先更 log 块,再更目的块)。

```
static struct inode* create(char *path, short type, short major, short minor)
```

首先,获取路径的父目录 inode,然后,查看路径是否已存在,若是,进一步查看其文件类型是否与创建需求中的文件类型一致(都是 T_FILE 类型),若是,返回这个 inode,若否,返回 0,若路径当前不存在,则为其创建一个 inode,并对其进行设置(major,minor,nlink),接下来,查看是否是要创建一个目录,若是,则增加其父目录的引用,并将“.”链

接到自身，“. .”链接到父目录，最后，在父目录下创建这个文件。

`int sys_open(void)`

首先，获取参数（路径和打开方式），若打开方式包含创建操作，创建一个 inode，若不包含创建操作，获取路径对应 inode，若要对这个目录文件（如果 inode 是目录）执行非读操作，返回-1（错误），接下来，创建文件指针和文件描述符，最后，设置文件（type，ip，readable 等）。

`int sys_mkdir(void)`

直接调用 create 创建一个目录文件。

`int sys_mknod(void)`

直接调用 create 创建一个设备文件。

`int sys_chdir(void)`

获取路径对应的 inode，然后将进程的当前路径设置为 inode。

`int sys_exec(void)`

获取所有参数，调用 exec。

`int sys_pipe(void)`

首先，新建一个 pipe，然后，新建两个文件描述符。

sysproc.c

`int sys_sleep(void)`

首先，获取参数（睡 n 个 ticks），然后，循环 n 次以下过程：

- 睡觉；
- 过一个 ticks 后，被唤醒；

uart.c

在继续阅读之前，建议先浏览 Serial Uart--an in depth tutorial (以下简称 tutorial)，了解串口的相关信息。

```
void uartinit(void)
```

首先，关闭 FIFO（COM1 的端口地址是固定的 0x3F8，com1+2 是 Interrput Identification Register，其高 2 位置 0 表示不使用 FIFO，由于不同型号的串口的 FIFO 设置可能不同，会影响读写串口的机制，进而增加内核代码的复杂度，关闭 FIFO，就可以用同一套代码处理不同型号的串口，详见 tutorial)，接下来，置 DLAB 为 1（com1+3 是 Line Control Register，其 bit7 为 DLAB 开关，将其置为 1，是为了接下来写 DLL 和 DLM 来设置波特率，详见 tutorial)，然后，设置波特率为 9600（在 DLAB 置 1 时，com1+0 和 com1+1 分别是 DLL 和 DLM 寄存器，同时写这两个寄存器来设置串口的波特率，详见 tutorial)，接下来，置 DLAB 为 0，并设置字宽为 8 位（DLAB 置 0 是因为完成了波特率设置，接下来要使用 RBR 和 THR 等寄存器来收发数据，字宽为 8 位，意为每发送 8 个位就发间隔位，详见 tutorial)，然后，将 MCR 清零（Modem Control Register，用来与串口连接设备握手，详见 tutorial)，接下来，使能中断（com1+1 的 Interrupt Enable Register 置 1，意为 UART 可以主动发送中断给 cpu，详见 tutorial)，然后，检测串口是否存在（com1+5 的 Line Status Register 为 0xFF，意为串口不存在？)，不存在就返回，然后，清空 IIR 和 RBR，使能串口中断，最后打印一些信息到串口。

```
void uartputc(int c)
```

首先，等待输出缓存为空 (当前没有数据被输出)，然后，将数据写到 THR (Transmitter holding register)。

`void uartputc(int c)`

首先，确认输入缓存不为空，然后，将 RBR（receiver buffer register）中的数据返回。

`void uartintr(void)`

直接将 uartgetc 函数作为参数传给 consoleintr 来获取串口输入。

vm.c

在继续阅读以下内容之前，建议先浏览 Manual 的第 3.1 节及之后相关章节和“xv6-book”的 Appendix B，了解一下 x86 分段机制和 xv6 中对分段的处理。

接下来，内核调用 seginit 初始化段描述符，进入 seginit 的定义：

```
void seginit(void)
```

首先，调用 cpunum 获取当前 cpu 的 index，接下来，设置当前 cpu 的 gdt 各个段的描述信息（Segment Descriptor，xv6 对段的设置模式是 Basic Flat Model，也就是基本没有使用段的功能，详见 Manual 的 3.2.1 节，除了设置读，写，执行以及 DPL，其余都没有设置，详见 Maunal 图 3-8），内核和用户的代码段都是可执行和可读的，内核和用户的数据段都是可读可写（STA_W 的数值是 2 代表可读可写，详见 Manual 表 3-1）的，但内核段的 DPL（Descriptor Privilege Level）是最高的（0），而用户段的 DPL 是最低的（3），接下来，设置 CPU 段（SEG_KCPU）的描述信息，此段的 Base Address 指向当前 CPU 自身信息的数据结构（struct cpu，值得注意的是，此处设置的段长度为 8，因为这个段中不仅包含 cpu 的指针，还包含 proc 的指针，详见 struct cpu 的成员定义），然后，载入设置好的 gdt（lgdt），接下来，将当前 CPU 的段描述在 GDT 中的偏移载入到 gs 寄存器（loadgs），然后，给当前 cpu 数据结构指针（被规定是 gs 寄存器所指的段加上 0 偏移）赋值（就是上面初始化过的 c），最后，给当前 cpu 上运行的当前 proc（被规定是 gs 寄存器所指的段加上 4 偏移，因为 proc 指针在 cpu 结构体中紧随 cpu 指针之后，详见 struct cpu 的成员定义）赋值为 0（意为没有进程运行），这两行代码让每个 cpu 可以通过相同的代码获取自身信息（local cpu info）和运行于各个 cpu 上的进程信息（因为 seginit 会被每个 cpu 运行一次，每个 cpu 都将各自的 cpu 信息指针信息保存到各自的 gs 寄存器中，一定要注意，proc.h 中定义的 cpu 指针和 proc 指针在每个 cpu 上是不同的，对此 proc.h 明确注释道：

各个 cpu 的%gs:0 就是&cpus[cpunum()]，%gs:4 就是 cpus[cpunum()].proc，这样一来每个 cpu 只需通过 cpu 指针和 proc 指针就能获取各自的 cpu 信息和 proc 信息，而不需每次都调用 cpunum()来获取 cpu index 再获取 cpu 和 proc 信息，同一套代码在不同的 cpu 上产生不同的效果，减少了代码的冗余)。seginit 定义结束。

```
static pte_t* walkpgdir(pde_t *pgdir, const void *va, int alloc)
```

首先获取页虚拟地址 va 所在页表在页表目录中的位置 PDX（va)，然后在页表目录中获取该页表的入口信息(pde：页表入口信息=页表的地址+页表的状态)，&pgdir[PDX(va)]，然后，查看该页表是否存在（PTE_P)，若存在，则从页表的入口信息中提取页表的地址（PTE_ADDR(*pde))，若不存在。则调用 kalloc 分配一页内存作为页表，然后调用 memset 将新获得的空闲页内存清零，然后将 pgtab 的物理地址和一些标志位存到页表入口信息（pde)，最后，获得虚拟地址 va 在 pgtab 中的位置，并将位置对应在 pgtab 中的的页入口信息的地址返回。

```
static int mappages(pde_t *pgdir, void *va, uint size, uint pa, int perm)
```

首先调用 PGROUNDDOWN 来获得内核每个模块起始和结束虚拟地址所在的页的地址（页下对齐)，接下来调用 walkpgdir 获取每一页虚拟内存对应的物理内存在页表中的存储地址，返回 mappages 定义，检测获得的页入口信息的合法性，不合法就 panic，然后，设置每一页的页入口信息（包括物理页地址和标志位，权限等)。

```
pde_t* setupkvm(void)
```

首先调用 kalloc 获取一页内存用来存放页表目录，接下来检测这一页内存地址的合法性，然后调用 mappages 将内核每一页物理地址到虚拟地址的映射存到各个页表中，最后，返回生成的页表目录地址，setupkvm 定义结束。

`void kvmalloc(void)`

首先调用 setupkvm 新建一个内核页表目录，然后，调用 switchkvm 将内核页表目录地址载入页表目录基址寄存器。

`void switchkvm(void)`

调用 lcr3 载入页表目录物理地址。

`void switchuvm(struct proc *p)`

首先，设置任务状态段（SEG_TSS）,当进程被中断时，会使用设置的栈（栈的段位于 SEG_KDATA，栈的指针位于 kstack），然后，将 SEG_TSS 载入任务寄存器（ltr），最后，载入进程的页目录。

`void inituvm(pde_t *pgdir, char *init, uint sz)`

首先，分配一页内存用来装载 initcode（initcode 由 initcode.S 汇编而来）程序，接下来，将这页内存映射到虚拟地址 0x0（一个进程的程序镜像开始于进程虚拟地址 0x0），并将这个映射保存到页表（mappages），最后将 initcode 程序装载到内存页。inituvm 定义结束。

回到 userinit 定义，接下来，设置进程占用的内存大小（sz），然后设置进程的 trapframe：

- 设置 tf 的 cs 的段类型为 SEG_UCODE；
- 设置 tf 的 ds，es，ss 的段类型为 SEG_UDATA；
- 打开 eflags 的中断使能（FL_IF）；
- 设置 esp（栈指针）指向程序镜像的末尾；
- 设置 eip（指令指针）指向程序镜像的开头；

接下来，设置进程名，进程的工作目录，最后，将进程的状态设置为 RUNNABLE（可被进程调度器调度执行）。

```
int loaduvm(pde_t *pgdir, char *addr,struct inode *ip, uint offset, uint sz)
```

按照以下步骤逐页将程序的其中一段从磁盘载入内存：

- 获取程序（其中的一页）应该在内存中的页地址；
- 将程序（其中的一页）从文件中拷贝到内存的上述页中；

需要注意的是，这段程序中有两个虚拟地址，第一个是 addr，其是程序段的某一页在其进程地址空间中的虚拟地址（这个地址在程序编译链接完成后就确定了）；第二个是 p2v(pa)，先看 pa，pa 是当前进程使用 kalloc 分配的一页内存的物理地址，其在当前进程中的虚拟地址就是 p2v(pa)；这两个虚拟地址指向的是同一个物理地址，但其在不同的进程中有不同的虚拟地址。

```
int allocuvm(pde_t *pgdir, uint oldsz, uint newsz)
```

创建一定数量的页（（newsz-oldsz）/PGSIZE），并映射到 pgdir 中。

```
int allocuvm(pde_t *pgdir, uint oldsz, uint newsz)
```

释放一定数量的页（（oldsz- newsz）/PGSIZE），并将其从 pgdir 中删除。

```
void freevm(pde_t *pgdir)
```

首先，将 pgdir 中的用户内存空间全部释放，然后，释放全部的页表页（页上全是页表入口，只是用户空间的页表页，内核空间的页表没有 PTE_P），最后释放 pgdir 所占的页。

```
void clearpteu(pde_t *pgdir, char *uva)
```

将某一页设置为用户不可访问（主要用来防止访问越界）。

```
pde_t* copyuvm(pde_t *pgdir, uint sz)
```

首先，给子进程创建一个页目录，然后，创建同样大小的内存，并将父进程的所有数据拷贝到子进程的内存里，最后，按照父进程的页目录映射，生成子进程的页目录映射（父子相同的虚拟地址指向相同的数据，但这些数据不在一个物理地址上），最后，返回页目录。值得注意的是，xv6 中复制父进程时，会将父进程的所有数据复制给子进程，而现实中，很多操作系统只是复制可写的数据部分，而不会复制只读的代码部分。

```
char* uva2ka(pde_t *pgdir, char *uva)
```

首先，获取 pgdir（一般不是当前进程的 pgdir）中 uva 对应的物理地址，然后，获得该物理地址在当前进程中的虚拟地址。

```
int copyout(pde_t *pgdir, uint va, void *p, uint len)
```

首先，获取 va 在 pgdir（一般不是当前进程的 pgdir）中的物理地址，然后获取该物理地址在当前进程中的虚拟地址，然后，将 len 长度的以 p（是当前进程的虚拟内存地址）开始的数据拷贝到 va 在当前进程中的内存中。