



操作系统设计文档

项目组长：陈自航

指导教师：鲁强

2022 年 8 月

目录

1. 概述	3
2. 系统整体架构	4
3. 系统硬件层次	5
4. 软件系统层次	6
5. 基础模块	6
5.1 进程管理模块	6
5.2 分配器模块	8
6. 非基础性模块	10
6.1 虚拟内存和页表管理	10
6.2 加载器模块	12
6.3 文件描述符模块	13
6.4 信号管理模块	15
6.5 多线程管理	17
6.5 文件系统模块	18
6.6 串口设备模块	20
6.7 网络模块	21

1. 概述

本项目是 2022 全国大学生计算机系统能力大赛内核赛参赛项目，该项目在 Sipeed Maixdock 和 QEMU 等硬件平台上实现了一个小型操作系统内核，并顺利通过了初赛和决赛阶段大部分功能点的测试。操作系统内核本质上是一个屏蔽硬件细节，为用户进程提供服务的系统级软件，其设计目标我认为有两个，一个是功能性强，一个是系统性能好，而前者是设计的首要目的。具体来讲，在初赛过程中，我们的系统支持 SD 卡读写和串口读写，这两个部分的实现就是功能性设计，而增加文件缓存，使用 DMA 方式进行数据传输来提高读写效率等工作则是属于性能设计的范畴。内核的设计当然更重要的还是功能性设计，只有实现了对应功能之后，才能够考虑如何优化以使其效率更高。

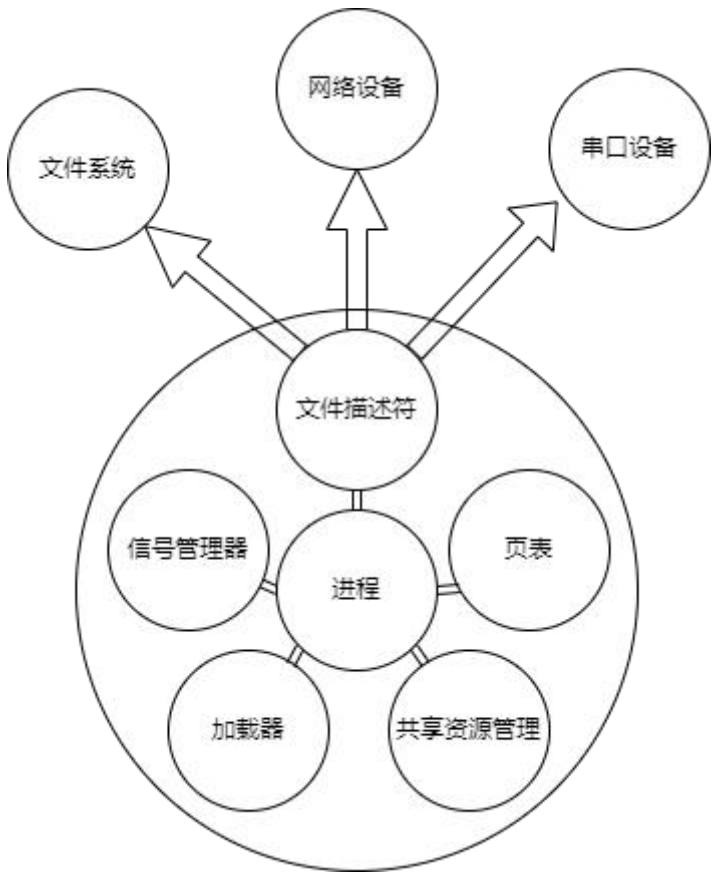
内核的设计本身又是一个细节性非常强的事情，这种细节性和硬件本身特性紧密相关，可以说操作系统内核和硬件是紧密联系的整体，内核的正常，稳定，按照预定的逻辑运行紧密依赖于硬件本身的机制，而硬件特性的充分发挥，又紧密地依赖于内核代码精妙的逻辑设计。本文档可能无法将这些细节性的东西一一用语言论述出来，但是这些东西比操作系统设计本身更加重要，我认为这些东西是构成操作系统的非常基础的东西，没有他们，系统很难正常运行。本文档只能从宏观上对系统的设计架构，系统的各个功能模块进行详细论述，对于那些更细节的东西，就不进行过多的论述了。

我们系统的设计中有一个非常核心的概念：进程。所谓进程，就是一个能够被 CPU 选择执行的，能够启动，运行，暂停，终止的一个上下文环境。系统的所有设计都围绕着进程展开。例如，一个用户进程想要访问 SD 卡中的文件，那么它会发出系统调用，使得进程进入内核状态，然后通过访问文件模块来完成这一次的系统调

用，最后返回用户态。在此过程中，进程可以看做是一个中心体，文件模块是一个边缘部分。进程被 CPU 所执行，并根据进程的需求访问不同的模块，完成后返回。多个进程环境下，只需要对对应的模块进行一些访问控制，例如资源不可用的时候，进程进入睡眠状态等待。这可以说是系统运行的一个大致机理了。

2. 系统整体架构

系统中有非常多的模块，这些模块都是围绕着进程展开的，有些模块和进程本身是耦合在一起的，而有些相对具有独立性，下面一幅图尽可能清晰地展示这些模块之间的关系。



图中的大环形区域代表进程及和进程本身耦合度较高的模块，包括文件描述符，信号管理器，页表，加载器和共享资源管理模块。文件描述符是描述一个文件或设

备的抽象模块，用户通过文件描述符号找到对应的设备或文件进行操作，而无需关心具体实现细节。页表用来表示进程的虚拟地址空间。信号管理器用来管理进程收到及发出的信号。加载器负责向进程虚拟地址空间中载入内容，可执行文件部分和动态链接库部分都有相应的加载器。共享资源管理主要是指多线程，当两个或多个进程需要共享页表，文件描述符，信号管理信息的时候需要用到这个部分。文件描述符所连接的设备模块和进程模块本身是松耦合的，系统中的文件描述符所指向的设备主要有文件系统，网络设备，串口设备。

此外，还有一些零零散散的模块没有在图中列出来，这些模块之间的耦合度也都比较低，和进程本身耦合度也比较低，例如内存分配器模块。我们把模块分成基础模块和非基础模块，一般地经常被其他模块所调用的模块称为基础模块，否则为非基础性模块，后面会按这两大类来对系统中的所有模块进行论述。

3. 系统硬件层次

RISCV 架构将 CPU 分成 3 个不同的特权级别，权力由高到低分别为 M，S，U 级别。我们的内核始终工作在 M 模式，并规定用户程序运行在 U 模式。CPU 的特权级切换需要跨过 S 态而直接在 M 态和 U 态之间进行切换。下面一张图反映了这种层次关系。



4. 软件系统层次

从软件层面上来看，操作系统位于软件层次的最底层，需要注意的是，多数 RISC-V 架构系统的软件层次都是以 SBI 作为最底层，这和我们系统是不太一样的。操作系统层之上则是属于用户的层次，目前系统已经实现了与 musl 和 glibc 库多数主要接口的对接，在 musl 和 glibc 层次之上则是用户的程序。此外，我们的内核采用的结构偏向于宏内核结构，基本上不把属于内核层次的东西放置在用户层次，所以内核直接和用户态的标准库进行对接。下面这张图展示这种层次结构。



5. 基础模块

5.1 进程管理模块

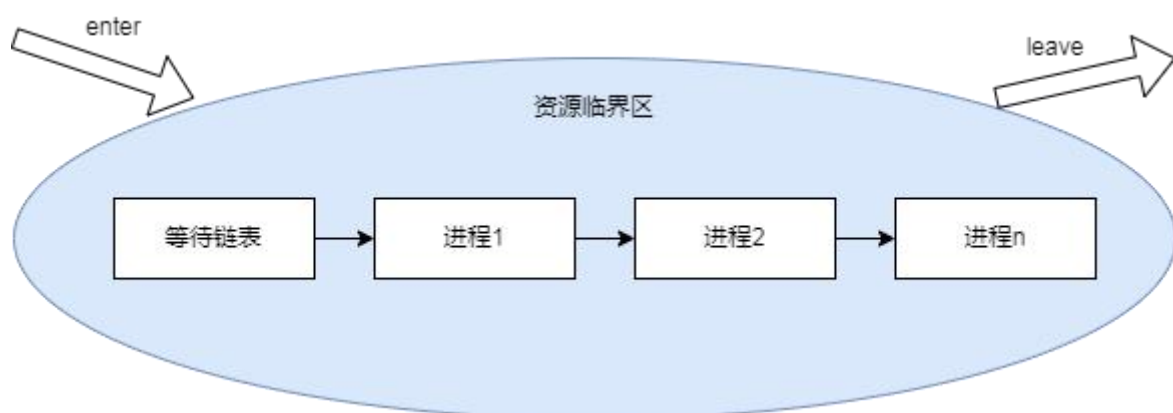
进程管理是系统中最基础也是最重要的模块，粗略可以划分成进程调度器部分，进程创建和销毁部分，进程睡眠阻塞部分，以及进行进程父子关系管理部分。我们主要创建了 3 个公共链表用来管理系统中的进程，分别为：

1. Runnable_list
2. Zombie_list
3. All_list

这三个链表分别代表可执行的进程列表，僵尸进程列表，所有进程的列表。调度器模块所做的工作很简单，就是从可执行进程列表中选择一个开始执行，可执行的进程进入可执行列表的方式是可以指定的，目前系统中加入可执行列表的方式遵循先入先出原则，系统也提供了先入先出和后进先出两种方式供调用方选择以何种方式被调度。

进程创建即进程控制块的创建，最开始是通过预留一定大小的空间用于进行进程控制块内存的分配，后来随着进程控制块大小的不断增大，这种方式显得越来越笨重，所以改进后的方法分配一个页面大小的内存作为进程控制块，并将进程控制块物理内存的地址除以页面大小的值作为进程 ID，这种方式能够保证进程能够拥有全局唯一的 ID，而且不会带来较多的开销，唯一的问题就是进程控制块的实际地址暴露了出来，可能会带来一些安全隐患。进程任务结束后并不是直接进行销毁，而是等待父进程或 1 号进程回收该进程，在回收该进程的同时销毁进程。

进程等待某一资源时往往需要进入睡眠状态以释放 CPU，并等到资源可用后被唤醒。目前系统的进程睡眠和唤醒机制和 xv6 的设计没有太大的区别，唯一的改动就是我们使用链表结构来管理等待某个资源的进程。



当进程访问某一种公共资源时，首先进入资源的临界区，查看资源是否可用，

若不可用，则将进程本身挂到资源的等待链表中，进入睡眠状态并离开临界区。待其他进程释放资源时，进入临界区，唤醒等待链表中的进程，这就是进程睡眠和唤醒的基本方式。

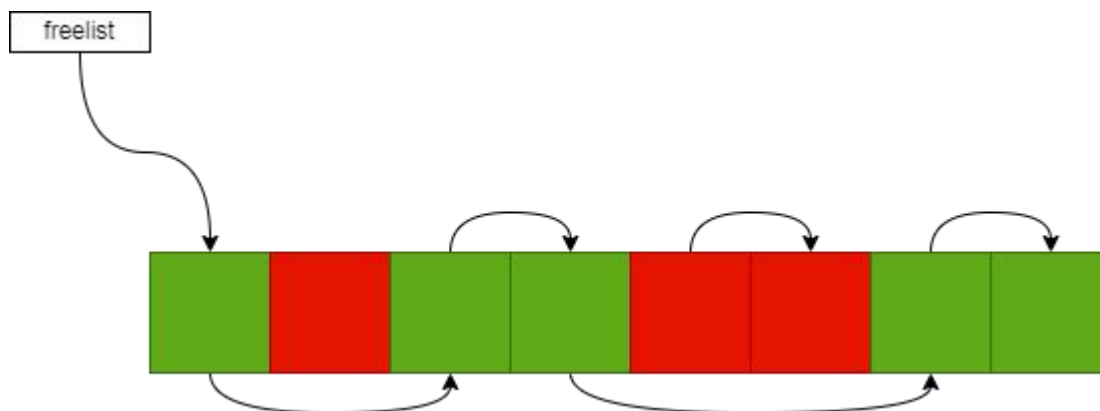
进程父子关系的管理也是采用链式管理，每一个进程控制块中都有 `child` 和 `sib` 属性，使用左孩子右兄弟的方法形成进程父子关系树。很多上层功能都依赖于进程父子关系的管理，例如进程调用 `wait` 方法来等待一个进入僵尸状态的孩子进程时，需要访问自己的孩子进程，`getppid` 等系统调用更是要求对进程父子关系直接的访问。

5.2 分配器模块

系统中的分配器设计更多考虑了系统的简洁性，我们系统运行在一个较为简单的硬件平台上，应当以实现主要功能为分配器设计的前提，而不能盲目地设计非常复杂的分配器系统。对于我们的系统而言，大多数内存分配的操作都比较简单，且很多时候内存的分配都是页面大小的整数倍，所以分配器设计也综合了各种因素，当前实现的分配器是一种以页面为最小粒度的连续内存分配器，其分配的复杂度为 $O(n)$ ，释放的复杂度为 $O(1)$ ，除此之外没有其他的额外开销。系统对于可分配页面的管理采用的是数组化管理方式，在系统启动时，内核尾部一段内存空间（其大小取决于物理内存总大小）被用作页面管理数组，每个元素的定义如下

```
struct page{  
    int refcnt;  
    struct page *link;  
};
```

每个元素都对应了物理内存中的一个页面，`refcnt` 为 0 代表页面未被使用，此外，可用的页面元素通过一个 `freelist` 链表连接起来，以便进行空闲页面的查找。下图显示了某一时刻数组的状态，其中绿色代表空闲，红色代表不空闲。



空闲的块（绿色标识）都连接到了 **freelist** 链表中，而不空闲的块（红色标识）不在 **freelist** 链表中。但是，为了记录分配出去的内存长度，对于连续多个页面的分配，需要使用链表将这些连续的元素连接起来以表示连续页面的长度。也就是说 **page** 结构体中的 **link** 属性是一个复用字段，当页面空闲时，用来进行 **freelist** 的连接，页面非空闲时，用来进行连续页面的连接。

元素中的 **refcnt** 属性表示块的引用次数，这在某些地方会用到，例如进程虚拟地址空间中的页面是写时复制页面时，通过 **refcnt** 来表示该公共页面的引用次数。释放某一个页面时，分配器模块会查看页面引用次数，直到达到 0 才会真正释放。

最后，系统还对于内存不够的极端情形进行了一些设计，当调用分配器的分配函数分配内存时，分配器先检查是否还有可分配的内存，如果没有了，则通过内存 **callback** 机制来召回一些被分配的内存，并再次检查。缓存的召回目前考虑了主要有两个方向，一个是向文件系统的缓存索要，一个是向虚拟地址空间已映射的页面索要（交换区机制），这两个方法都曾经有过尝试，且后者在决赛第一阶段的一个极端测例上采用交换区机制完成了测试点。

6. 非基础性模块

6.1 虚拟内存和页表管理

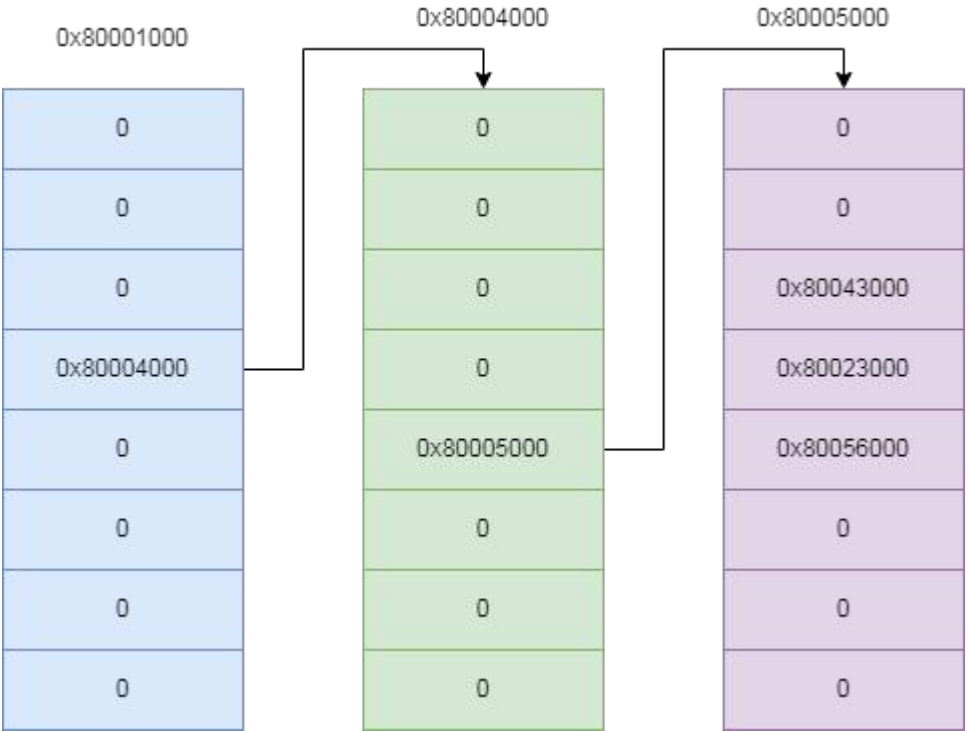
系统中每一个用户进程都有一个对应的页表，在不考虑多线程的情况下，该页表就是进程所独有的，它表示了用户的虚拟地址空间。我们根据需求对用户的虚拟地址空间进行了划分，其空间布局如下：



当前来讲系统的虚拟地址空间（256G）远大于物理地址空间（6M），所以很多设计都没有考虑虚拟地址重复利用的问题，例如在动态库映射这一部分我们仅设置一个指针值代表动态库的可映射起点，该值一直递增直到超出动态库区域，而一般的设计可能需要设置一个 bitmap 结构来记录页面的映射情况，见缝插针地使用虚拟地址空间，而对于我们这种虚拟地址空间远大于物理地址空间的情况设计起来就感觉比较自由，没有太大的约束。

页表管理没有太复杂的内容，基本上就是往虚拟地址空间中写入和从中读取内容，这些东西使用 xv6 原本自带的一些函数就能够实现，而且效率也很好。但是 XV6 在虚拟地址空间整体的复制和删除等操作上并没有太好的支持，虚拟地址空间中的

页面映射情况是具有不确定性的，对于虚拟地址空间整体的操作需要整体遍历页表才能完成操作，基于这个考虑，我们在原本的基础上设计了一种效率相对较好的遍历方法，能够让其遍历整个虚拟地址空间进行操作的同时兼顾遍历效率。这种遍历会对高层页表中的页表项进行判断，若为 0 则不进行更深一层的遍历，从而提高遍历效率，以 SV39 页表为例：



假设进程的页表第一级地址为 `0x80001000`，遍历程序遍历全部的 256 个页表项（图中没有全部画出），对于页表项不为零的再进行下一级的遍历，图中只有一个页表项非零，指向 `0x80004000`，从而进行第二级遍历，第二级页表中也只有一项，指向 `0x80005000`，指向第三级，再进行第三级遍历。最终我们遍历次数为第一级的 256 个页表项，第二级的 256 个页表项以及第三级的 256 个页表项，这比从虚拟地址 0 开始逐页面进行遍历的效率要好得多。

此外，系统中允许有一个页面被多个页表同时引用的情况，通常来讲，属于只

读类型的页面内容不会发生改变，这种页面多个页表共享并没有什么坏处，反而能够节省不少内存。此外，fork 写时复制机制的实现也依赖于页面共享，在进程使用 fork 之后，对于可写类型的页面将它标记为只读的写时复制页面，该页面不被修改时，没有任何问题，当有一方试图修改该页面时，会触发页面异常，并分配新的页面，复制原来页面的内容，重新标记为可写。

6.2 加载器模块

加载器模块是将进程可执行文件的内容加载到进程虚拟地址空间的一个功能模块，最开始加载器模块是包含在进程模块内部的，因为最开始进程的加载和执行非常简单，只需要将进程可执行文件的镜像复制到虚拟地址空间中执行就可以了，但是随着需求的不断增加，系统的越来越复杂，加载器越来越大，逐渐与进程模块相分离。

系统中的加载器对于可执行文件（包括动态库）的加载都是采用的懒加载的方式，即在程序运行前标记对应的页表项，程序发生缺页错误时再补充映射对应页面的内容。在执行程序之前，系统先将可执行文件的 program-header 信息提取出来保存到一个数组中，待发生缺页错误时，判断对应的页面是否位于待加载的区域，并和数组中所记录的加载信息进行比较，最后加载对应的页面。

目前系统有两个加载器，一个是可执行文件（.elf）的加载器，一个是动态库文件（.so）的加载器。后者只是用于动态链接解释器的加载，而可执行程序所依赖的动态链接库会被动态链接解释器自动加载，内核不需进行关心。

加载器模块以及前面提到的写时复制都会带来一个严重的问题，那就是用户在请求某个系统调用服务时，如果传入的地址值恰好是写时复制页面位置或者是懒加

载页面的位置，该如何处理？为了解决这个问题，我们每次在内核中接收到一个用户传来的地址后，先对其进行检查，看对应的位置是否为写时复制或懒加载类型的页面，并进行相应的处理后，才进行内存的操作。

6.3 文件描述符模块

文件描述符是操作系统为用户提供的访问硬件设备的统一接口，用户通过文件描述符代码来对文件或设备进行便捷的操作。除了串口，SD 卡，网卡等实际设备以外，系统中还有三个虚拟设备也被文件描述符所支持，分别为 NULL 设备，虚拟文件和管道。具体来讲，系统中的文件描述符有下面几类

```
FDESC_CONSOLE = 1,
FDESC_FILE,
FDESC_PIPE,
FDESC_NULL,
FDESC_SOCK,
FDESC_VIRTUAL
```

FDESC_CONSOLE 是串口设备的文件描述符，FDESC_FILE 用来描述一个 SD 卡上的文件，FDESC_PIPE 为管道，主要用于进程间通信，FDESC_NULL 为空设备，输入和输出都是 0，FDESC_SOCK 用来描述一个套接字，用于网络编程，FDESC_VIRTUAL 描述一个虚拟的文件（不在 SD 卡中）。

根据 linux 系统一切皆文件的设计宗旨，系统中的一切文件描述符都应当有尽可能统一的操作接口，例如 read write 等调用可以操作于几乎所有类型的文件描述符，但是，实际中并不是所有类型的描述符的所有操作接口都一样，下面的表格列出了当前本系统对于不同文件描述符类型及操作的适用关系：

	CONSOLE	FILE	PIPE	NULL	VIRTUAL	SOCKET
dup	√	√	√	√	√	√

dup2	✓	✓	✓	✓	✓	✓
openat		created			special	
close	✓	✓	✓	✓	✓	✓
read/readv	✓	✓	✓	✓	✓	
write/writev	✓	✓	✓	✓	✓	
sendfile	receive	send/receive	send/receive			
fstat		✓				
getdents		✓				
lseek		✓				
mkdir		✓				
mmap		✓				
pipe			created			
unlink		✓				
renameat		✓				
utimensat		✓				
sendto						✓
recvfrom						✓

从上表中可以看到，系统中有很多和文件描述符相关的操作（尚未全部列出），但很难实现理想中的使用一套完全统一的接口来实现对所有类型描述符进行访问的模式，甚至如 fstat, unlink 等方法只适用于文件类型的描述符，而完全不适用于其他类型，linux 系统试图屏蔽所有硬件的区别，所有设备都抽象成文件，提供统一的接口给用户，这个目标还是很难达到的。我认为更多时候我们需要在异构性和同

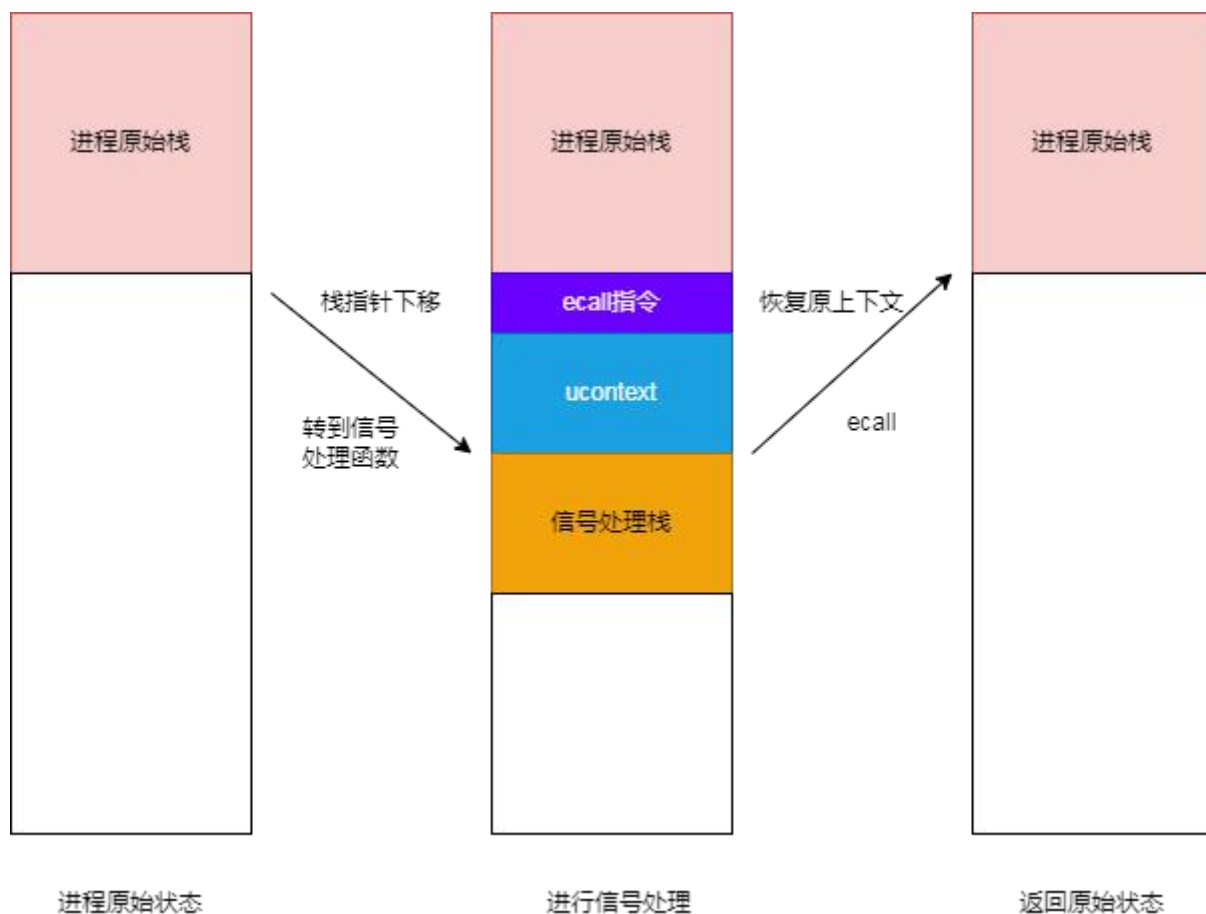
一性之间寻找一个较好的平衡点，而不能单纯地偏向于某一个方向。

6.4 信号管理模块

信号机制相对于其他模块来讲使用地不是很频繁，在我们系统中对它的重视程度也不是很高，我们尽可能把它设计地简单，逻辑不出错就可以了。

信号模块分成信号的检查以及信号的处理，系统中信号的检查时机位于进程从内核态即将切换到用户态之前。信号检查要做的就是对发往该进程的信号进行主动检查，并根据信号屏蔽位来决定是否对该信号进行响应。

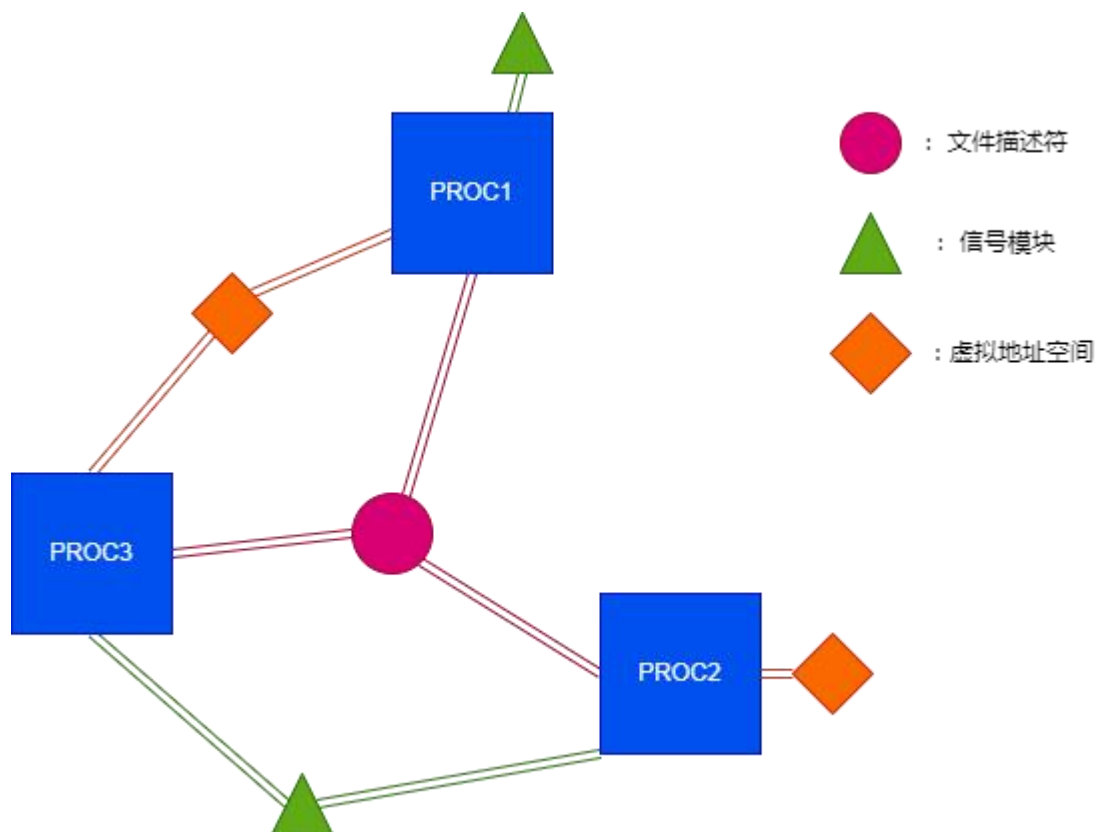
信号处理的过程分成下面三个步骤，第一步将进程的栈指针下移一定长度，下移留出的空间用来存放一个 `ecall` 指令和一个 `ucontext` 结构体，第二步切换用户进程上下文进行信号处理，最后，用户通过 `ecall` 系统调用告知内核信号处理已经完成，切换回原本的上下文。



具体说来，在检测到信号并决定要进行信号处理之后，进程将原本的上下文（寄存器组）备份到另一个特定的位置，然后将当前上下文的栈指针下移一个 `ecall` 指令及一个 `ucontext` 结构体的长度，内核将填充该 `ucontext` 结构体以向用户传递必要的信息，接下来，内核将上下文的 `pc` 指针指向信号处理函数的位置，并将 `a0` 寄存器设置为信号的 `num`，`a1` 寄存器设置为 `ucontext` 的地址，`ra` 寄存器设置为 `ecall` 指令的地址，这样，进程进入信号处理函数时，得到的参数列表就是信号 `num`，`ucontext` 地址，并且处理函数的返回地址是 `ecall` 指令所在的位置，执行 `ecall` 指令后内核即认为进程的信号处理已经结束，将之前备份好的原上下文恢复，即可继续执行进程。

6.5 多线程管理

文件描述符，信号模块，虚拟地址空间这三个部分中的一个或多个可以被若干个进程所共享，我们系统中不仅实现了传统意义上的多线程即虚拟地址空间的共享，还实现了文件描述符和信号模块的共享，这些共享关系可以是错综复杂的，下图显示了三个进程共享关系的某种可能情况。



这个图表示，三个进程共享了同一个文件描述符资源，PROC1 和 PROC3 共享同一个虚拟地址空间（页表），PROC2 和 PROC3 共享同一个信号处理模块，并且 PROC1 有独立的信号模块，PROC2 有独立的虚拟地址空间。

在进程控制块中分别使用 `gvm`, `gfd`, `gsig` 这三个字段表示页表，文件描述符，信号的共享情况，这和 `CLONE_VM`, `CLONE_FILES`, `CLONE_SIGHAND` 分别对应。当 `gvm`, `gfd`, `gsig` 这三个字段中某字段的值非 0 时，代表对应资源是和一个或多个进程

所共享的。在资源共享的情况下，进程控制块中指向对应资源的指针就是指向共享资源的指针，例如 gfd 非 0 时，进程控制块中 fdesc 字段指向共享的文件描述符资源地址。

对于共享资源的访问控制，我们针对不同的情况设计了不同的方式，首先针对页表共享和信号共享，采用读写锁的方式进行控制，例如当一个进程想要读取共享的页表时，首先拿到该共享页表的读者锁，如果当前有一个进程正在写入内容，则进入睡眠状态直到被写者进程唤醒，当一个进程想要写入共享的页表时，首先拿到该共享页表的写者锁，如果当前有一个进程正在写入内容，或者有至少一个进程正在读取内容，则进入睡眠状态直到被读者或写者进程唤醒。

对于共享文件描述符资源的管理，采用大粒度的锁似乎不太合适，因为系统中的文件描述符资源中有上百个文件描述符槽，采用整体加锁就意味着对这几百个槽同时加锁，非常不合理，况且文件描述符资源被使用的频率还是相当高的。所以我们采用了另一种方式，我们在每个文件描述符槽中添加一个 busy 标识，代表这个槽正在被使用，并使用一个自旋锁保护所有描述符槽的 busy 状态。基本操作有两个，拿到槽和释放槽，拿到槽的操作会原子性地设置槽的状态为 busy，此后，任何进程都不能试图访问这个槽都会导致进程睡眠等待，进程释放槽之后，会原子性地修改槽 busy 的值，并唤醒所有的睡眠进程。这样，锁的粒度就降低到了单个的描述符槽，而不是整个文件描述符资源。

6.5 文件系统模块

我们把文件系统分成了两个层次，一个是表示文件节点及节点间关系的层次，一个表示是节点在文件系统中的具体存在的层次。具体来讲，文件系统分成 FS 层和

FAT32（本系统简称为 f32）层，其中 FS 层维护文件系统中文件节点之间的父子关系，我们采用树结构进行表示，F32 层对于 FS 层中的每一个节点进行详细的解释，即该节点在 F32 文件系统中处于哪个扇区，哪个簇，簇链的情况等等，是一些非常具体的文件节点本身的信息。

这种分层结构逻辑上比较清晰，且和内核中的文件描述符模块能够方便地对接，即文件描述符模块不需要考虑文件的具体存在形式，也不需要知道具体的文件系统，而只需要知道系统中有这样一个文件，能够通过标准的接口（本系统中有 `fsread`, `fswrite`, `fsfind`, `fscreate`, `fsdelete` 等）对文件和文件系统进行操作，屏蔽了底层的细节。

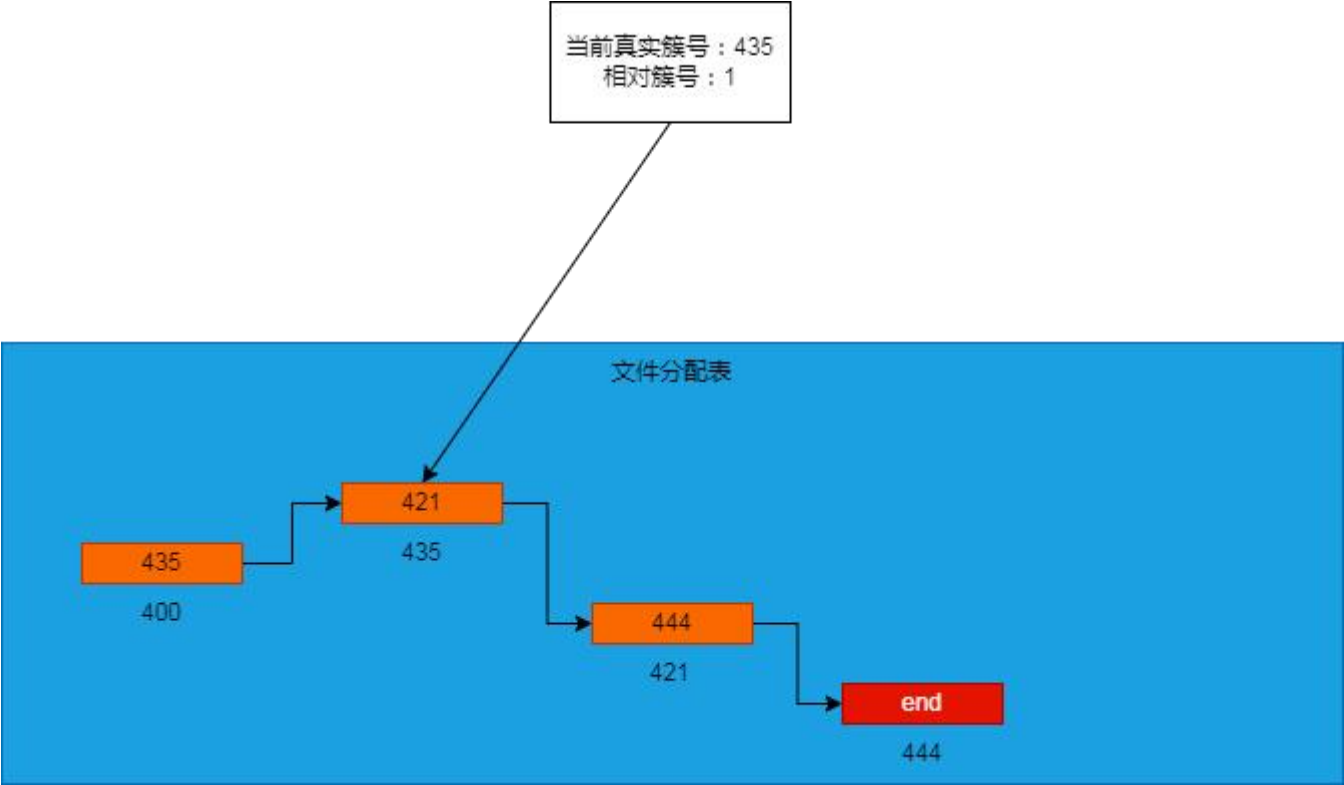
上层请求到达 fs 层后，fs 层会和 f32 层进行对接，例如调用了 `fswrite` 函数后，`fswrite` 会首先锁住该文件节点及该文件的父节点（考虑 F32 的特性），然后调用 f32 层的 `f32write`，传入该节点及父节点信息，最后对该节点和其父节点进行解锁，从而完成文件写入的请求。

我们系统中采用基数树缓存结合扇区缓存的方式来构建文件系统的缓存系统，基数树缓存位于 f32 层，而扇区缓存相对独立一些，不处于某个特定层次。基数树缓存针对文件进行缓存，把文件按照页面大小进行划分并进行缓存。基数树的深度为 4，每层 256 个索引项，8 比特，基数树共使用 32 位页面号作为索引，最大可表示文件 16384GB。

基数树缓存用于单个文件具体内容的缓存，而扇区缓存是对 SD 卡上实际的扇区进行缓存，后者主要用于一些特殊数据区域如 FAT 分配表区域的访问和缓存。

除了缓存机制，系统还引入了一定的机制以加快 F32 文件的读写效率。由于 F32 是以簇链来表示文件连续性的，访问某一个文件的一个特定位置往往需要顺着簇链

进行迭代，直到找到对应位置的簇号，时间复杂度为 $O(N)$ 。在我们的设计中加入了当前正在访问的簇的信息，包括真实簇号，相对簇号，可以方便地从任意位置开始遍历簇链，从而降低簇定位的时间。



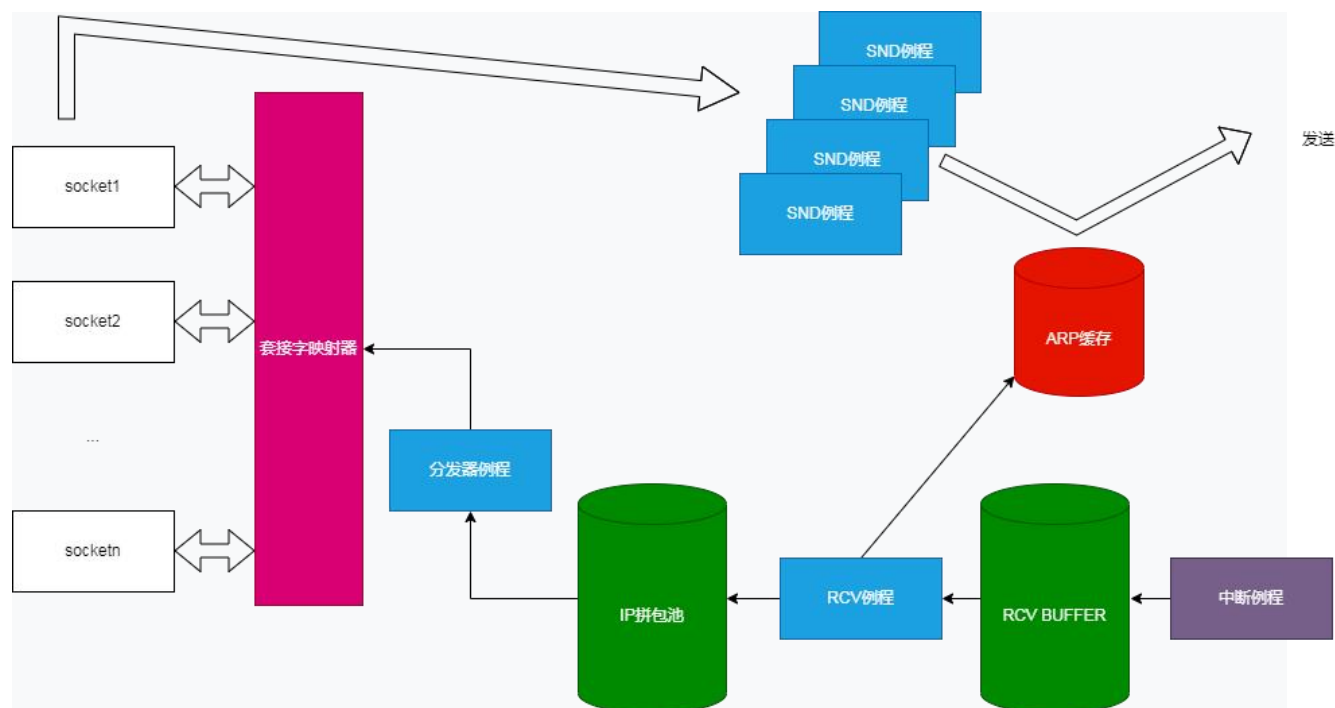
相对簇号即为簇相对于文件起始簇的相对值，真实簇号即实际意义上的簇号，通过这两个属性可以很方便地从当前位置直接顺着簇链进行簇定位，这在大多数顺序读取或写入的情况下能够起到非常好的效果。

6.6 串口设备模块

串口设备模块和 XV6 的原始版本没有太多的区别，其基本思路即构建接受和发送两个相互独立的通道，并构建一定长度的接受缓冲区。串口设备发出中断后，系统会检查串口设备输入的情况，并将输入字符添加到接收缓冲区。用户请求串口读取时，从接收缓冲区中读取数据；串口写入时，直接输出到串口设备。

6.7 网络模块

截至撰写此文档，网络协议栈尚未完全完成，但一些基本功能已经实现，UDP 栈的部分已经基本完成，能够支持 `gethostbyname` 方法。这里只对当前网络栈的架构进行简要的说明。



单个进程很难独立完成整个网络功能的实现，所以我们又增加了若干内核例程来共同配合完成网络协议栈。网卡设备发出中断时，中断处理程序会检查网卡是否发来了数据包，并将数据包放入 RCV 缓存（接收缓存）中，RCV 例程负责将数据包从 RCV 缓存中取出，进行分类处理，目前我们支持 ARP 包和 IP 包的处理，对于 ARP 包，视情况修改 ARP 表，对于 IP 包，则将 IP 包进行解析，与拼包池中已经存在（若不存在则新建）的 IP 包进行合并和拼包，拼包完成后包被标记为 ready。分发器例程从拼包池中取出已经拼包完成的 IP 包，解析其传输层部分，并根据 socket 映射表将包分发给不同的 socket。Socket 发送一个报文时，将报文封装好后创建 SND 例程发送该报文，SND 会对目的 IP 进行 ARP 解析，并进行报文拆分和发送。