华中科技大学

DagasOS 内核设计文档

队员：冯业齐、卞曜洋、李诗阳

目　录

一、 使用 QEMU 启动并调试系统内核

1.1 目标

建立 DagasOS 仓库，设计工程结构，编写 Makefile 脚本。
实现 SBI 中 boot_jump 的功能，以 S 态进入系统内核。
使用 riscv-unknown-elf-qemu 运行并调试内核代码。

1.2 步骤

1.2.1 开始之前

1.2.1.1 建立 DagasOS 仓库

比赛要求选手使用 gitlab 仓库托管代码。
首先在 gitlab 上创建初始的仓库，接着拉取完成初始设置的 gitlab 仓库

```
git clone https://gitlab.eduxiji.net/T202410487992737/DagasOS.git
```

设置默认忽略追踪的文件

```
# Prerequisites
*.d
# Object files
*.o
*.ko
.....
#build
build/
#link：  riscv64
riscv-syscalls-testing/user/riscv64
#kernel
kernel-qemu
sbi-qemu
```

在每次修改完成后，使用 git 同步代码。

```
git add .
git commit -m "commit reason"
git pull
git push
```

1.2.1.2 设计工程结构

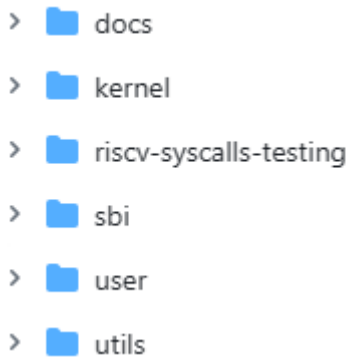


图 1.1 工程文件夹结构

docs：书写文档的部分，在其中记录在开发 DagasOS 过程中的想法，遇到的问题和解决方案。

kernel ：系统内核的核心代码，用于生成 kernel-qemu，在其中进行异常处理，内存分配，进程调度中大部分功能的实现。

riscv-syscalls-testing：比赛官方提供的测试数据集，生成的用户可执行文件会存入 sdcard.img 磁盘镜像，在这里编译其用于本地测试。

sbi：提供管理态二进制接口的底层工具，用于生成 sbi-qemu，能实现相同内核在不同机器环境下的兼容。

user：由我们自己编写的初始用户程序，生成的可执行文件将存放于引导盘镜像 initrd.img，在系统启动时会将 initrd.img 挂载，并且启动其中的用户程序去扫

描并运行根目录下的测试文件。

utils：辅助编译和调试的工具源代码，如生成 sbi 加载后的二进制文件工具 capture_elf.c

1.2.1.3 编写 Makefile 脚本

在开发初期，DagasOS 曾经考虑过使用单个 Makefile 文件，但如此做导致 makefile 文件过于冗长，且将中间文件与源代码存放在同一目录下，影响开发体验。

随着开发的进行，DagasOS 使用分层的 Makefile 文件来解决这一问题。

```
# DagasOS/Makefile

......

kernel:
	-rm kernel-qemu
	make -C $K all DDEBUG=$(DDEBUG)
	cp $(BUILD_DIR)/$K/kernel kernel-qemu

......

all : if_root sdcard.img kernel sbi initrd.img
```

```
# DagasOS/kernel/Makefile

SRC_C = $(wildcard *.c)
SRC_S = $(wildcard *.S)

K_BUILD_DIR = ../$(BUILD_DIR)/kernel

OBJS = $(patsubst %.S, $(K_BUILD_DIR)/%.o ,$(SRC_S) )
OBJS += $(patsubst %.c, $(K_BUILD_DIR)/%.o ,$(SRC_C) )

$(K_BUILD_DIR)/kernel: $(OBJS) kernel.ld $(SRC_C) $(SRC_S) print
	$(LD) $(LDFLAGS) -T kernel.ld -o $(K_BUILD_DIR)/kernel $(OBJS)

print:
	$(CC) -c $(CFLAGS) $(DDEBUG) -o $(K_BUILD_DIR)/print.o print.c
$(K_BUILD_DIR)/%.o : %.S
	@echo $@ $^
	$(CC) -M $(CFLAGS) -o $(patsubst %.o, %.d ,$@) $^
	$(CC) -c $(CFLAGS) -o $@ $^

$(K_BUILD_DIR)/%.o : %.c
	@echo $@ $^
	$(CC) -M $(CFLAGS) -o $(patsubst %.o, %.d ,$@) $^
	$(CC) -c $(CFLAGS) -o $@ $^

all :
	@if [ ! -d    "$(K_BUILD_DIR)" ]; \
		then mkdir $(K_BUILD_DIR); \
		fi;
	make $(K_BUILD_DIR)/kernel
```

我们使用 riscv64-unknown-elf-gcc 工具来编译项目，在编译过程中先解析依赖关系生成.d 中间文件，然后再生成.o 二进制文件，最后利用自己编写的链接脚本链接生成 kernel 文件。

1.2.2 实现 SBI

1.2.2.1 起初 DagasOS 在 M 态

机器模式（缩写为 M 模式，M-mode）是 RISC-V 中 hart（hardware thread，硬件线程）可以执行的最高权限模式。在 M 模式下运行的 hart 对内存，I/O 和一些对于启动和配置系统来说必要的底层功能有着完全的使用权。

机器模式最重要的特性是拦截和处理异常（不寻常的运行时事件）的能力。RISC-V 将异常分为两类。一类是同步异常(synchronous exception)，这类异常在指令执行期间产生，如访问了无效的存储器地址或执行了具有无效操作码的指令时。另一类是中断，它是与指令流异步的外部事件，比如鼠标的单击。RISC-V 中实现精确异常处理：保证异常之前的所有指令都完整地执行了，而后续的指令都没有开始执行（或等同于没有执行）。

M 模式下八个控制状态寄存器（CSR）是机器模式下异常处理的必要部分：

1. mtvec（Machine Trap Vector）它保存发生异常时处理器需要跳转到的地址。

2. mepc（Machine Exception PC）它指向发生异常的指令。

3. mcause（Machine Exception Cause）它指示发生异常的种类。

4. mie（Machine Interrupt Enable）它指出处理器目前能处理和必须忽略的中断。

5. mip（Machine Interrupt Pending）它列出目前正准备处理的中断。

6. mtval（Machine Trap Value）它保存了陷入（trap）的附加信息：地址例外中出错的地址、发生非法指令例外的指令本身，对于其他异常，它的值为 0。

7. mscratch（Machine Scratch）它暂时存放一个字大小的数据。

8. mstatus（Machine Status）它保存全局中断使能，以及许多其他的状态

1.2.2.2 总之得有一个系统栈

qemu 在 0x1000 处获取 mhartid 后直接跳转到 sbi 的起始地址 0x80000000 中，此时堆栈信息没有设置，我们需要根据其 mhartid 给其分配一个独立的堆栈。

```
auipc sp, KMEMORY / 4096
```

接着，DagasOS 会跳转到 SBI 的 start 函数中。

1.2.2.3 现在让 S 态来接管

start 函数主要承担第一次从 M 态过渡到 S 态的工作。

在 start 中我们需要设置以下寄存器

1、mstatus 中的 MPP 位，使 mret 后进入 S 态。

2、pmpaddr 和 pmpcfg，使得所有内存空间对系统合法

3、mepc 设置返回 S 态位置为 main 函数

4、MIP 和 SIP 完成部分中断的代理

最后使用 sret，进入 S 态的同时进入 main 函数。

1.2.2.4 初始的 SBI 完成了

初始的 SBI 到此就完成了，但这并不代表 DagasOS 可以直接在 QEMU 里使用我们生成的 ELF 文件，QEMU 加载 SBI 时并不会将其视为一个 ELF 文件，而是将其作为一段二进制数据载入 0x80000000 地址中，这里我们编写了一个 capture_elf 工具去模拟 SBI 的加载，生成对应的二进制文件。

同时我们需要编写 ld 脚本保证 entry 函数位于 0x80000000 位置。

```
OUTPUT_ARCH( "riscv" )
ENTRY( _entry )



SECTIONS
{
  . = 0x80000000;

  .text : {
    *(.entry)
    *(.text .text.*)
  }
  .....
}
```

现在 SBI 的 boot_jump 的功能终于完全实现了。

1.2.3 使用 riscv-unknown-elf-qemu 运行并调试内核代码

1.2.3.1 QEMU 参数解析

比赛官方提供的 qemu 选项为

```
qemu-system-riscv64 -machine virt -kernel kernel-qemu \
  -m 128M -nographic -smp 2 -bios sbi-qemu \
  -drive file=sdcard.img,if=none,format=raw,id=x0 \
  -device virtio-blk-device,drive=x0,bus=virtio-mmio-bus.0 \
-initrd initrd.img
```

其中 smp 指定的是 CPU 数量，m 指定内核大小，kernel 选择 kernel 文件，bios 选择 sbi 文件，drive file 指定磁盘镜像，initrd 指定引导盘镜像。

1.2.3.2 QEMU 调试

qemu 调试需要添加参数-S 令其等待 gdb 信号，然后指定 socket，gdb 与 qemu 之间传递信息是借助 socket 进行的。

1.3 调试

由于频繁测试，这里为 Makefile 添加了 run-gdb 功能

暂时取消了 gcc 的 O1 优化

经过 gdb 断点调试，发现成功进入 main 函数。

1.4 总结

万丈高楼，起于垒土。完成了 DagasOS 最初的工程创建，这表明团队有能力实现一个操作系统，现在一切的挑战即将开始。DagasOS 会在后续实现异常处理，内存管理，进程和线程的调度和文件系统。

二、异常处理

2.1 目标

1、 实现 kerneltrap 及其响应路径，处理内核中断。

2、 M 态的 mtvec 设置，处理设备中断。

3、 实现 usertrap 及其响应路径，处理用户中断。

2.2 步骤

2.2.1 kerneltrap

2.2.1.1 如何进入 stvec

1、start 中实现将中断交给 S-mode 代理(delegate)。

2、在 kernelvec 中实现上下文的保存并进入处理程序。

3、设置 CSR，以进入 stvec。

2.2.1.2 stvec 中实现了什么

目前 stvec 能处理的仅有时钟中断，且无须进行额外响应。
出现异常即代码出错直接停机即可。

2.2.2 M 态的 mtvec 设置

2.2.1.1 如何进入 mtvec

1、start 中没有将中断交给 S-mode 代理(delegate)。

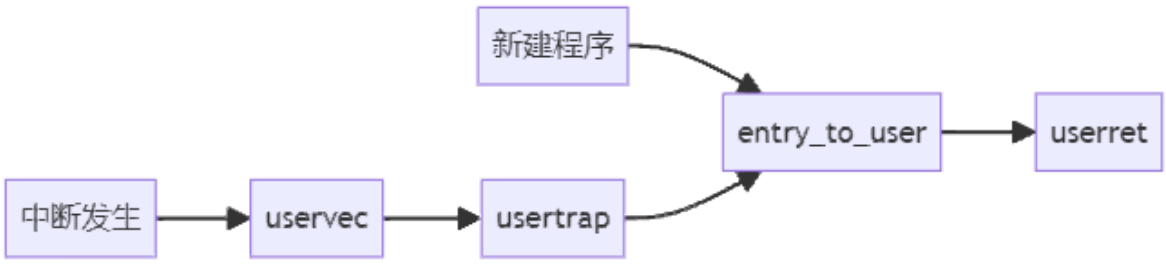
2.2.2.1 mtvec 中实现了什么

处理时钟中断

1. 首先预留五个 uint64 的空间来处理重置时钟的工作（[0..2]暂存寄存器，[3]存储上次触发时间，[4]存储间隔时间）。
2. 用 CSR 寄存器 mscratch 保存上述空间的地址。
3. 触发中断时先保存寄存器。
4. 将当前时间赋值给上次触发时间。
5. 当中断出现将上次触发时间加上间隔作为新的触发时间。
6. 将对应信息写入 CLINT 中的对应寄存器。
7. 引发 S-mode 的 2 号中断处理程序。
8. 恢复现场。
9. mret 进入 S-mode 的中断处理程序。

2.2.3 usertrap

2.2.1.1 如何进入 usertrap



1、start 中实现将中断交给 S-mode 代理(delegate)。

2、在 uservec 中实现上下文的保存，页表的切换并进入 usertrap。

3、设置 CSR，以进入 stvec。

2.2.1.2 uservec & userret

DagasOS 将这两个函数置于一个特殊的页， 这里称之为 trampoline_origin。在内核页表初始化和用户程序页表初始化时，会将 trampoline_origin 映射到一个相同的高位置虚拟空间 trampoline。由于拥有相同的虚拟地址，在 uservec 或 userret 中切换页表，仍然能正常寻找到下一个语句。

2.2.1.3 usertrap 中实现什么

实现对于用户缺页，时钟中断以及系统调用的处理。

用户如果缺失的页是采取 lazy_alloc 未分配的页，在中断处理程序中为其补上。

2.3 调试

设置 csr 中 stvec 的值为我们的中断跳转程序。在这里，我们遇到了一个困扰我们一个半小时的严重问题，就是 stvec 总是无法写入。

我们尝试给他随意赋值（例如 0x999）。然而，这样无论如何操作、在何处、以何种权限都无法成功写入。我们尝试了一些其他 csr 寄存器，有时可以写入，有时不行。

在尝试将 0x999 写入 原本赋值为 0x222 的 sie 时，我们却发现 sie 变为 0x000。这是令人震惊的。然后我们意识到在 S 态将 0x222 写入 sie 是成功的。因此我们怀疑是否是写入的值的问题。

在这里我们尝试将 0xffff0000 写入 stvec 令人震惊地，这居然成功了！我们立刻意识到，此前一直无法写入（转而会导致置零）的原因是，我们写入的值没有对 4 对齐，这使得我们本质上在修改一个 read-only 的部份。

2.4 总结

中断处理是系统的一个重要部分，中断连接了设备和用户之间的不平衡的需求，而系统处理中断的能力，解决了供需的不及时，不平衡。使得所有用户程序协调且高效的运行。

三、 内存管理

3.1 目标

1、 实现内核层面上对物理内存的虚拟化

2、 实现用户可见的虚拟内存

3、 维护虚拟内存空间

4、 设计虚拟内存和索引树

5、 分配多个页和不足一个页的写法

3.2 步骤

3.2.1 用户可用的内存

我们可以假装放一个很大的数组，并宣称它就是我们的用户可用的内存。我们只需要在链接脚本里指定这个内存应当处在的位置。

我们在链接脚本中使用 PROVIDE 命令标记内核占用的内存的末段，并将紧随其后的若干「页」指定为用户可用的内存。

一方面，我们定义变量：

```
extern char pmem_base[];
```

然后，我们在链接脚本的 SECTION 部份的末段指定：

```
PROVIDE(pmem_base = .)
```

这样将对应的地址导出成了符号。

当然，事实上，在这里我们还没有划分出真正的页。所以我们只是简单地约定一个物理内存的上限。事实上这就是我们一开始在编译选项中定义的 128 mb 内存。也就是说，我们将所有除了内核占用的几个段以外的内存都划成用户能访问的物理页。然后用一个链表维护这个物理页池即可。

同时，我们设计函数 palloc 和 pfree 用于分配和释放物理页。特别地，为了预防 double free 之类的 bug，我们每一次 free 以后会将页 memset 成 'U' ，alloc 以后会用 'N' 填充。

3.2.2 伙伴系统

朴素的链表在分配灵活性上存在一些问题，尤其是我们有时为了性能会希望分配一段物理意义上连续的内。这样可以在内核使用对应内存的时候免于重新映射对应的页表，而是直接用默认的全同映射就能完成。因此我们引入伙伴系统（Buddy System）用来管理大块、连续页的分配。

伙伴系统每一次只能分配 2 的次幂的页的内存。我们维护一棵线段树，线段树上每个节点代表一个能分配的区间，在这个节点处记录这个区间中最长的可分配 2 次幂区间。每一次分配的时候从上往下找一个恰好能分配的区间，然后将这个节点的可分配长度设 0 。

具体而言，我们如下做：

分配时，为了尽可能满足分配的对齐需求，我们先尝试右子树，再尝试左子树，直到找到一个节点满足这个区间整体未分配，且它的左右子区间都不够分配，就将这个区间整体分配出去，将当前区间的 m 值改为 0 ；

之后自下而上进行 m 值的更新，pa.m←max{ls.m,rs.m}。但有一个特例，如果左右区间均完全没有被分配，则 pa.m←ls.m+rs.m， 即将两个区间合并成一个更大的区间以供分配。

释放的时候，我们直接找到这段页的起始偏移量对应的线段树上的节点，然后恢复它，并往上合并零碎的节点。

我们实现的 Buddy System 提供了 palloc(_n) 和 p_free 函数，用来管理一开始用链表管理的页池。

3.2.3 slab

这里的 slab 几乎和 Linux 的 slab 一致，用于管理小块的（主要是不超过一页的）内存分配。将从伙伴系统中获取的整页地址进行合适拆分分配给内核。
具体实现是将页拆分成相同大小的内存段。每次 kalloc 取对应大小的内存。

3.2.4 虚拟内存

3.2.4.1 页表

通过类似树形的检索结构组织虚拟内存空间，通过三层页表形成一个动态的索引树。根索引目录下最多存放 512 个二级索引目录，每个二级目录下最多存放 512 个一级索引目录，每个一级索引目录下存放 512 个页表。

将虚拟地址划分为

- [0..11] 页内偏移(page_offset)
- [12..20] 0 级索引
- [21..29] 1 级索引
- [30..38] 2 级索引

最大支持 512GB 虚拟内存空间。

至于系统虚拟内存划分的部份，我们将 UART,kernel,PLIC,PMEM 的虚拟地址设置为与物理地址相同:避免因为在启用虚拟内存功能后，部分函数功能无法正常工作。这部分可以理解为一块全同的虚拟内存映射。

如果所有虚拟内存映射都借助系统去实现“翻译”，那么开销将是无比巨大的。故而 RISCV 借助 satp 寄存器辅助管理页表，实现在硬件层面对于虚拟内存的“翻译”。硬件层面会使用 tlb 等缓存技术，提高虚拟内存“翻译”的效率。

为了使硬件能使用我们设置的页表，我们根据 RISCV 手册去存储页表和设置寄存器。

3.2.4.2 RISCV 下内存映射寄存器

页表根目录寄存器 satp 的结构为

- [0..43] PNN (根索引目录的物理地址)
- [44..59] ASID (标识当前页表，使得机器能确认页表是否发生变化) (内核取 MAX_PROCESS+1 其他为进程号+1。 为什么+1：riscv64 对于 ASID = 0 有特殊处理)
- [60..63] mode (页表模式 8 for SV39)

注意 PNN 的偏移值在 satp 中为 0 而在 PTE 中为 10。

SV39 下 PTE 的结构为

- [0..9] 状态 VRWXUGAD
- [9..52] PNN (索引目标的物理地址)

指令 sfence.vma zero, zero 可以刷新 TLB

注意每次切换 satp pmp 寄存器前后需要刷新对应 TLB

刷新 TLB 后内核正式启用虚拟内存。

虚拟内存启用后，原先的各种代码仍然如原先般运行。但对应的代码涉及到内存的部份处理的不再是原来物理上的 page（块），而是从虚拟内存的堆空间取出一个可用的 page。

但依赖这些被修改的操作内存的函数的函数也能正常运行，因为他们也能“自动”通过页表“翻译”。

3.2.5 vm

vm 是管理虚拟内存段的结构体。每个 vm 拥有一个 pm_list 。

vm 可以管理对应虚拟内存段在 init, fork，exit，pagefault 时的行为（如何初始化，如何拷贝，是否释放，如何处理缺页）

每个进程拥有一个 vm_list。用于管理它对应的虚拟内存段。

3.2.6 地址约定

我们约定内存结构如下图：

kernel virtual memory space

name	base	size
CLINT0	0x02000000	PG_SIZE
PLIC0	0x0c000000	PG_SIZE
VIRTIO	0x10001000	PG_SIZE
KERNEL0	0x80000000	内核代码大小
PMEM0	KERNEL END	TO_MAX_PA
HEAP0	0x100000000	0x80000000
CORO_STACK0(tid)	0x20000f000 + tid * 0x10000	最大PG_SIZE
STACK0	0x300000000	0x80000000
TRAMPOLINE	0x3fffff000	PG_SIZE

user virtual memory space

name	base	size
CODE0	0x00000000	用户代码大小
STACK0	0x300000000	0x80000000
CORO_STACK0(tid)	0x20000f000 + tid * 0x10000	最大PG_SIZE
TRAMPOLINE	0x3fffff000	PG_SIZE

图 3-1 内存结构图

3.3 调试

在 mappages 时，代码发生 remmap(重复映射)。原因是 pmem（堆空间）的对齐未合理实现，一个较好的实现方式是在链接脚本中添加. = ALIGN(0x1000);（页大小为 4096）。

在启用 satp 代码后内核立刻出现异常，并且异常处理程序也无法工作。经过排查后发现是未给予 text 段执行权限(PTE_X)。

切换 satp 之后内存没有正常切换，经检验发现是没有及时调用 sfence 刷新寄存器。

在 sys_wait4 系统调用中，由于需要通过传指针来回传一个额外的返回值，而这个地址是跨越了两个不同的上下文的，故而需要使用 copy_to_va 来完成写入。

四、线程和进程的调度

4.1 目标

1、 正确分配进程和线程。

2、 切换线程运行。

3、 处理线程进程的状态切换和退出。

4.2 步骤

4.2.1 Process

4.2.1.1 资源管理的单位

我们认为进程是资源管理的单位，这样方便我们分配资源。

4.2.1.2 父进程与子进程

显然每个进程都需要存它的父进程。但它是否需要存它的子进程呢？

把一个进程的子进程都存下来有一些好处。例如，可以快速地遍历所有子进程。

但这样的空间开销也是不可忽视的，并且如果一个“关系”需要双向维护，那么保证“关系”的同步将会带来很大的代码实现难度。

在实现 wait 的时候，发现不记录进程子进程的代价是巨大的，每次 wait 需要遍历整个进程池。

而且在实现 wait 前，我们已经实现了 kmalloc，可以实现内核小段地址的分配，似乎保存自进程的开销也变得可以接受。

但多次 kmalloc 带来的时间代价依旧是难以接受的，所以我们在 process_t 中添加 child_list 和 prev,next 来保存线程树信息，在需要更新时获取 thread_pool 的 wait_lock 来更新。

4.2.1.3 fork

在多线程结构中，fork 理论上只会复制调用它的那个线程。（据说这是「POSIX 系统的历史包袱」）

或者说，我们不妨认为，多线程和子进程是不良兼容的，系统不保证你在多线程的过程中 fork 了一个子进程会发生什么。

4.2.1.4 wait

关于 wait ，在多进程系统中存在一个严重的问题。具体来说，一个进程和它的子进程，谁先结束，谁先运行到 wait 函数处，是没有办法保证的。

父进程要如何知道子进程是否结束，我们调查到两种方案。

一种方案是，子进程在结束以后其资源仍不释放，直到父进程使用 wait 函数监听了它的释放；这样的问题是可能会出现僵尸进程。也就是父进程一直不释放子进程的资源。

另一种方案则是，在父进程处维护一个类似消息队列的东西，子进程似了以后给父进程发一个消息。这样的问题是给进程增加了额外的信息储存。

真实情况是两种方案都无法确定在运行到 wait 时子进程的状态，所以 DagasOS 采用第一种方案解决 wait 前结束的子进程，采用第二种方法去解决 wait 后结束的进程

4.2.2 Thread

4.2.2.1 执行流控制的单位

我们认为线程是执行流控制的单位，这样方便我们控制代码运行。

我们考虑使用 scheduler_loop 去寻找一个处在就绪状态的 thread 然后运行之。即更换对应 proc 的页表，切换至用户程序的上下文，然后进入 S 态。

4.2.2.2 why kernel coroutine?

但当我们考虑到从用户程序返回后，内核即将切换 thread 时，我们发现此时内核只有可能处在中断处理程序，而我们无法获知在运行用户程序前 scheduler_loop 运行状态的信息，势必会导致潜在的内核内存泄漏问题（scheduler 在系统栈中的数据无法正常销毁）。在学习 xv6 时，我们注意到 xv6 中存在的

switch_context 函数。根据反复研读代码，认真分析，我们发现 xv6 隐式地为每一个 process 实现了一个“协程”(coroutine)，这里我们称呼其为管理协程；而 scheduler 隶属于 CPU，可以将其视为 CPU 的“主协程”（main coroutine）， xv6 在”协程“间可以自由切换。

在我们的 DagasOS 中，我们编写了协程库 coro.h，显式地将协程的切换逻辑表示出来。我们借鉴了 C 标准库中协程功能 setjump 和 longjump 功能的实现，根据 setjump 在切换前后返回值的不同，实现了 switch_coro 的功能，方便后续开发者理解和使用。这里我们将详细阐述，协程在进入用户程序前后的切换过程。

在进入用户态之前，系统首先需要从主协程切换到对应的管理协程，然后再进入用户程序。

那么，当从用户程序进入中断处理程序时，kernel 仍然在运行管理携程，如果需要切换用户程序，则先切换至主协程，而主协程仍然接着上次切换前的状态继续运行，选择出下一个合适的线程。

当主协程再次选择上述的 thread 继续运行时，管理协程将会在上次中断处理函数中继续运行，而中断处理程序会经过 entry_to_user ，然后运行 trapret 最后回到用户程序。

而辅助协程可以随时创造和销毁，由主协程中管理其内存空间，可以避免内存泄漏。

注意

为了避免协程出现内存泄漏，我们可以采取最简单的方法，在每次 entry_to_user 回到用户程序后，都可以重置协程的栈，这样可以避免每次协程因为生存时函数跳转流程不同，导致栈空间内存的泄漏。

4.2.3 线程和进程的关系

一个进程上允许存在多个线程，DagasOS 在初期使得多个线程**堆栈代码段资源完全相同**，这可能会导致**线程**之间的一些非法操作（如修改同一进程上其余线程的系统栈），但我们在系统层面上允许这种行为：综合考虑后，认为这种行为不具有较大的安全隐患，且用户可以避免并且在不了解系统的页表结构时很难触发。

并且在实现 fork 的时候，DagasOS 发现不同线程如果栈空间虚拟地址不同会带来 fork 之后无法访问涉及 fork 前栈空间信息的内容，于是在实现 fork&exec 的版本中，我们重新将用户程序系统栈的虚拟地址统一设置为 TSTACK0 不再与线程相关。

进程会记录附着的线程数（但不知道具体 tid：减小内存负担），当一个进程没有内存附着的时候，会允许其回收资源。

回收资源分为两步，第一步是释放掉 vm_list 记录的虚拟空间表中的虚拟空间，以及页表信息，并且令其子进程父进程变为 NULL，在此之后进程将会成为僵尸进程。

僵尸进程保留进程的基本信息，维持进程树上的关系，在其父进程 wait 其或其不存在父进程（父进程变为 NULL）时彻底回收剩下所有资源。

4.3 调试

4.3.1 在切换进程时仍然处于中断关闭状态

图 4.1 在切换进程时仍然处于中断关闭状态

原因：在某个版本修复 wait()时，sched 函数切换回时再次关闭中断，导致 usertrap 进入 sched 会导致中断仍然关闭。

解决方法：在 usertrap()中，无论时 sched 还是 ret_to_user 都应该打开中断。

4.3.2 休眠线程无法唤醒

图 4.2 休眠线程无法唤醒

原因：scheduler_loop 运行到休眠线程时，线程无法被唤醒。

解决方法：在发现线程休眠后主动归还锁。

4.4 总结

在线程调度中需要解决的是资源之间等待与竞争的关系。在系统中使用中断和锁解决。但每次关中断获取锁都必须及时释放，保证线程不会长时间占有资源，导致系统拥塞。

而进程管理记录线程使用的资源，线程需要依附进程才能运行，进程也需要线程的附着才能利用资源。

五、文件系统

5.1 概述

文件系统构建实验主要需要为操作系统内核提供对于文件系统的支持，使得操作系统能够获取外部磁盘的信息、同时又能将自己的修改持久化到磁盘中去。可以说文件系统是操作系统内核的一个关键组成部分。

具体来说，该实验主要需要实现读写磁盘，解析文件系统结构，对创建、读写文件操作等接口的封装，挂载镜像等功能。

在该部分中，我们首先需要创建一个供操作系统内核使用的磁盘镜像，并把内核运行时所需要的文件写入其中。其次需要对文件系统的结构进行划分，这里我们将文件系统按从底层到高层的顺序划分为**磁盘驱动层、磁盘块缓存层、具体文件系统层、虚拟文件系统层、文件层、文件描述符层**。然后我们要利用每一层的上一层的接口，去实现该层提供给下一层的接口。最后我们要基于这个文件系统接口实现供用户调用的和文件系统相关的系统调用。

5.2 创建外部磁盘镜像

首先我们需要创建一个空白的磁盘镜像，在 linux 中可以使用如下命令来创建：

```
dd if=/dev/zero of=sdcard.img bs=512k count=128
```

由于比赛的 SD 卡设备使用的是 FAT32 文件系统，所以我们此处也使用 FAT32 文件系统将该镜像格式化：

```
mkfs.vfat -F 32 sdcard.img
```

然后我们要把这个镜像挂载到主机的一个文件中，将内核运行过程中需要使用到的文件复制进去，具体代码如下：

```
sudo mount sdcard.img /mnt/sdcard
cp user_program /mnt/sdcard
sudo umount /mnt/sdcard
```

这样我们就得到了一个供内核使用的磁盘镜像 sdcard.img。

5.3 划分文件系统结构

对文件系统的结构进行划分是为了将文件系统部分的工作解耦以及为实现虚拟文件系统、缓存常用磁盘块、缓存常用文件等功能做准备。

首先我们要实现一个操作磁盘设备的**驱动层**，通过该驱动层我们能够直接发送操作命令给磁盘设备，控制对磁盘块进行读写。

将读取磁盘块内容后，我们需要一个数据结构来对这块内容进行管理，从而方便地将这部分内容进行缓存，以更高效地提供给程序的其他部分来使用，并在需要时将被修改的内容通过驱动层的接口进行写回。因此需要一个**磁盘块缓存层**。

读取磁盘内容后，便可以依据该磁盘的文件系统去解析磁盘的文件组织结构。该部分需要将文件系统的常用信息进行记录，并依据这些信息，按照对应文件系统的规范，实现对文件的查询、创建、读取、写入等操作。这里是对具体的文件系统进行操作，所以可以叫做**具体文件系统层**。但目前我们也只需要对 FAT32 提供支持，所以也可以叫做 **FAT32 层**。

为了让我们的操作系统能够统一地管理各种文件系统，我们需要抽象出一个**虚拟文件系统层**，让所有对具体文件系统的操作都能通过该虚拟文件系统提供的接口去实现。将虚拟文件系统作为一个代理去管理其背后的具体文件系统。

一般而言，用户操作的对象是文件，而并不会直接对文件系统进行操作。因此我们需要实现一个**文件层**，该层提供对于文件的操作，这些操作的内部实现是通过对其背后的文件系统进行操作而实现的。

在具体进程中对文件进行操作需要通过文件描述符来指定文件对象，这样在避免了由于用户直接操作文件对象而带来的越权以及其他难以预测的后果的同时，也降低了用户操作文件的难度。因此**文件描述符层**的实现是很有必要的。

5.4 磁盘驱动层

5.4.1 virtio 概述

qemu 指定内核使用 virtio 协议来与磁盘进行通讯，因此先要了解 virtio 协议的具体内容。

virtio 是半虚拟化的解决方案，对半虚拟化 Hypervisor 的一组通用 I/O 设备的抽象。它提供了一套上层应用与各 Hypervisor 虚拟化设备（KVM，Xen，VMware 等）之间的通信框架和编程接口。其具体的关系图如下图所示。

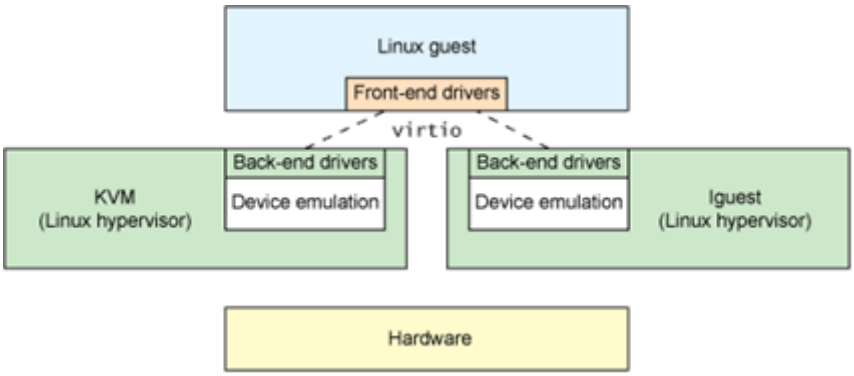


图 5-1 virtio 关系图

使用 qemu 命令运行操作系统内核时，该命令同时会创建一块虚拟磁盘设备 x0，并将 x0 连接到 virtio mmio 总线上。而我们驱动层就是通过这部分 mmio 总线来向该设备发送命令，该设备会接收命令、执行操作、然后同样通过 mmio 来返回结果。

5.4.2 磁盘驱动层的实现原理

磁盘驱动层初始化时，需要对磁盘设备进行一些配置。其中最关键的是将三个重要的数据结构：描述符数组（virtq_desc）、待处理指令队列（virtq_avali）、已完成指令队列（virtq_used）的内存地址发送给磁盘设备。

生成读写操作时，描述符数组会作为储存操作指令本身的内存区域。每条完整的操作指令由 3 个描述符结构体以链的方式构成，每个操作符通过其在描述符数组中的索引位置来指明链中的下一个描述符。

生成好读写操作指令后，需要将每条操作链头部描述符的索引放入待处理队列的队尾。完成这一系列操作后，即可将 VIRTIO_MMIO_QUEUE_NOTIFY 位置处设置为 0（待处理队列的标号，由于只有一个待处理队列，所以只需要设置为 0）来通知设备处理队列中的操作。

因此在实现磁盘读写接口时，仅需按照协议规范生成出对应的描述符链，将头部索引插入到待处理队列队尾、通知磁盘进行处理，并让当前线程进入休眠即可。

磁盘设备在完成某条操作后会发出一个外部中断信号，并将这条操作头部描述符的索引放入到已完成队列的队尾，并将处理结果（成功或失败）写到操作中所指定的地方。

由于磁盘设备是通过外部中断来通知内核操作处理完成的，所以我们需要实现一个磁盘中断处理程序来读取已完成的操作，并唤醒因磁盘操作而休眠的线程。

5.5 磁盘块缓存层

5.5.1 磁盘缓存块的需求

磁盘块缓存在使用时会和一块具体的磁盘块进行绑定。当内核操作该磁盘块时，会对其对应的缓存进行操作。因此每个缓存结构体都需要一块缓存区来存储磁盘块数据，同时需要记录其绑定的磁盘块的块号。当一个磁盘块没有被任何一个线程引用时，它就可以被释放掉，并在需要时被重新绑定到另一块磁盘块。因此缓存结构体还需要一个引用数变量来记录当前他被引用的次数。

同时我们还需要定义一个缓存块链表来组织所有的缓存块（空闲或已被占用的）。每次操作一块磁盘块时，先查找该队列是否已经缓存了该磁盘块，若是，则直接对该缓存块进行操作；否则选择一块空闲的缓存块将其绑定到该磁盘块、把磁盘块内容读取进缓存区，然后再对该缓存块进行操作。

当对缓存块进行修改后，我们并不马上将其写入到磁盘中去，而是给其打上一个脏标记。但该缓存块要被重新绑定或内核强制要求将所有更改同步到磁盘时，才将它的修改写入到磁盘中去。

同时缓存块还需要一个队列来记录当前在等待该块完成操作的线程，以便当操作完成时去唤醒这些线程。

5.5.2 磁盘缓存块的数据结构

综上可设计得磁盘块缓存结构体如下：

```
struct buf {
    int dirty;//0: clean, 1: dirty
    int valid;// has data been read from disk?
    int disk;// is disk operating the block?
    uint32 dev;
    uint32 block_id;
    uint32 refcnt;
    struct buf *prev;
    struct buf *next;
    uint8 data[BSIZE];
    wait_queue_t *wait_queue; // threads wait for this block
};
```

5.6 FAT32 层

5.6.1 FAT32 文件系统概述

首先我们要了解 FAT32 文件系统的组织结构。

简单来说，一个使用了 FAT32 文件系统的磁盘由引导区、FAT 表、数据区三个区域构成，整个磁盘被划分为固定大小的扇区，数据区被划分为固定大小的簇，并通过 32 位的簇号来定位。一个簇由若干个扇区组成。

其中磁盘的第一个扇区为文件系统引导块，会记录一些该 FAT32 的基本参数，如一个扇区的大小、根目录文件的簇号等。

紧接着引导块的是 FAT 表（文件分配表），FAT 表的键是簇号，值则是该簇在其所在的文件链中的下一个簇的簇号、结束号或者空闲号。

FAT 表后是数据区。数据区用于存储目录文件以及普通文件。目录文件的内容是该目录下的文件项。文件项会指明该项对应的文件内容的所存放在的第一个簇的位置。

在具体实现时，我们首先读取磁盘的引导扇区的内容，并从而获得 FAT 表的偏移、大小等数据。再把 FAT 表全部读取到内存中，这样便完成了文件系统块（superblock）对象的初始化。

5.6.2 FAT32 文件系统操作的实现

下面简单说明一些文件系统操作的实现方法。

查找文件：给定其父目录文件内容的起始簇号，依次获取其中的文件项，比较文件项的名字与查找的文件名是否相同，相同则返回该文件项数据结构，否则继续向后查找。

创建文件：给定父目录文件内容的起始簇号，找到第一个空闲位置，创建文件项，并分配一个空闲簇作为其内容区域的起始簇。

读取文件：给定文件内容的起始簇号，按顺序去读取。

写入（覆盖）文件：给定文件内容的起始簇号，按顺序写入。

要注意的是，对文件内容进行读取时，若当前簇读取完了，则要使用 FAT 表寻找下一个簇，对文件内容进行写入时，若当前已经是最后一个簇，则需通过 FAT 表查找到一个空闲簇接入到簇链尾。

可见 FAT 表会经常访问，这也是将 FAT 表全部读入内存的原因。

5.7 虚拟文件系统层

这里主要模仿了 linux 的设计，将每个文件项对应于一个 inode，所有对文件项的操作通过 inode 进行。

inode 由统一的 inode 链管理，获取某文件 inode 前会查询是否已存在该文件的 inode，有则直接使用，否则才返回，从而保证每个文件的 inode 最多只会存在一个，以免对文件的不同步修改。

由于具体文件系统不止一个，所以我们要为不同的具体文件系统分别实现一套虚拟文件系统接口的实现。为了编码的统一性，这里采用了类似于面向对象程序中多态的思想，每个具体文件系统初始化时会生成一个统一接口的 superblock 对象，并且把针对该文件系统实现的具体函数赋值给该 superblock 对象的函数指针。对 inode 进行操作时，会先获取 inode 对应的 superblock，然后用 superblock 中的函数指针来操作 inode。其中主要实现的函数接口如下：

```
int (*lookup_inode)(inode_t *dir, char *filename, inode_t *node);
int (*read_inode)(inode_t *node, int offset, int size, void
    *buffer);
int (*write_inode)(inode_t *node, int offset, int size, int cover, void *buffer);
void (*update_inode)(inode_t *node);
int (*create_inode)(inode_t *dir, char *filename, uint8 type, uint8
    major, inode_t *inode);
```

5.8 文件层

文件层主要是实现 file 对象。file 对象是对 inode 的引用，但同时又附加文件偏移，操作权限等和用户操作强相关的数据。

值得注意的是，可以存在多个 file 对象对应同一个 inode，这是因为用户可能以不同权限打开了同一个文件。

同时为了实现设备文件的功能，file 还有一个 major 字段来表时其表示的设备号。

综上，file 对象的结构体如下：

```
struct file_struct
{
    uint8 type;
    uint32 refcnt;
    uint8 readable;
    uint8 writable;
    uint32 flags;
    inode_t *node;
    uint32 off;
    uint8 major;
};
```

5.9 文件描述符层

文件描述符是直接暴露给用户的对文件的引用，它是每个进程间独立的，其具体对应的 file 对象由每个进程的文件描述符表记录。

当用户通过系统调用打开某个文件时，内核会在进程文件描述符表中找到第一个空的文件描述符，并把该描述符绑定给一个新生成的 file 对象。

而用户通过系统调用操作某个文件时则会通过进程的文件描述符表找到对应的 file 对象，再对 file 对象进行操作。

文件描述符简化了用户对文件的操作，同时隔离了进程间的文件控制，是保障文件系统在用户视角下方便、安全的重要一层。