**2024.10 – 2025.01 基于RISC-V 架构的 xv6 操作系统 个人项目**

**技术栈： C语言、页表机制、系统调用、进程调度、陷阱机制、文件系统**

**项目描述：**xv6是MIT基于RISC-V架构开发的类UNIX操作系统，其运行在QEMU模拟的硬件环境下。本项目旨在深入理解操作系统核心机制，通过对xv6操作系统多个模块进行扩展与优化，提升其系统性能与功能完整性。

**工作亮点：**

* 修改xv6的**页表机制**，使得内核直接解引用用户虚拟地址，提升系统调用性能；
* 实现xv6的**周期性用户级中断机制**，允许用户进程注册中断处理程序，支持定期触发与正确恢复执行；
* 实现xv6的**延迟分配**和**COW策略**，减少物理内存冗余分配，提升进程创建效率
* 优化xv6的内存分配器和块缓存管理机制，实现多核环境下的高效内存管理和多进程文件系统访问的并发性能；
* 扩展xv6文件系统支持**软链接**功能，并通过引入二级间接块扩展单个文件的最大容量；

修改xv6的页表机制

**实验基础：**

xv6有一个单独的用于在内核中执行程序时的内核页表。内核页表直接映射（恒等映射）到物理地址，也就是说内核虚拟地址x映射到物理地址仍然是x。Xv6还为每个进程的用户地址空间提供了一个单独的页表，只包含该进程用户内存的映射，从虚拟地址0开始。因为内核页表不包含这些映射，所以用户地址在内核中无效。因此，当内核需要使用在系统调用中传递的用户指针（例如，传递给write()的缓冲区指针）时，内核必须首先将指针转换为物理地址，再根据物理地址获取内核虚拟地址，最后再通过虚拟地址进行数据读写。该实验的目标是允许内核直接解引用用户指针。

**实验思路：**

1修改内核来让每一个进程在内核中执行时使用它自己的内核页表的副本。

2修改struct proc来为每一个进程维护一个内核页表。

3修改调度程序使得切换进程时也切换内核页表。

4将用户空间的映射添加到每个进程的内核页表。

**具体做法：**

1. 修改proc结构体，添加新的字段unit64 kernelpt，用于保存进程的内核页表

2. 参考原代码中全局内核页表的创建函数kvmmap()和kvminit()函数，然后，添加自己的方法proc\_kpt\_init()初始化进程的内核页表，在进程的内核页表中建立物理地址和虚拟地址的映射，初始化的映射关系与全局内核页表一样

3. 进程的内核页表应该随进程的用户页表一起创建，也就是在allocproc()函数中，调用proc\_kpt\_init()函数。继续在allocproc()中为每个进程分配一个内核栈，并将内核栈映射到内核页表中。

4. 修改进程调度器scheduler()，在进程切换时，加载进程的内核页表到satp寄存器，切换后恢复全局内核页表，保证进程使用自己的内核页表。

5. 进程结束时，应该要释放内核页表，他应该和用户页表一起释放，所以在freeproc（）函数中释放。释放进程的内核页表时递归清除页表项（proc\_freekernelpt函数），并解除内核栈映射（uvmunmap），对共享的物理页（如设备内存）的映射会被清除，但物理页不会被释放。

6. 将用户空间的映射添加到进程的内核页表，目前，已经有了一个初始化的内核页表和用户页表。在内核更改进程的用户映射的每一处 （`fork()`, `exec()`, 和`sbrk()`）,都复制一份到进程的内核页表。

7. 替换掉原有的`copyin()`和`copyinstr()`，让需要解引用用户页表时，使用内核页表。

**Linux的页表机制，系统调用传递指针的原理**

实现xv6的周期性用户级中断机制

**实验基础**：

usertrap、proc结构体、**RISC-V陷阱机制处理流程**、**陷阱帧中常见寄存器的作用**

**实验思路：**

新增sys\_sigalarm和sys\_sigreturn系统调用，结合定时器中断触发用户处理函数，并通过陷阱帧保存/恢复机制避免状态破坏。

**具体做法：**

为了支持周期性用户级中断，我在 proc 结构体中新增了 alarm\_interval（报警触发周期）、alarm\_handler（报警处理函数指针）、ticks\_count（周期内时钟滴答计数）、is\_alarming（是否正在执行报警处理）以及 alarm\_trapframe（保存报警前的寄存器状态），并在 allocproc 和 freeproc 中完成了内存分配与回收（主要回收alarm\_trapframe），保证进程资源的完整管理。

我实现了新的系统调用 sigalarm(interval, handler)，允许用户注册定时器处理程序，并设置报警周期。在 usertrap 函数中处理时钟中断，当滴答数达到设定的报警周期且当前未处于报警处理中时，我首先把当前的 trapframe 备份到 alarm\_trapframe 中保存所有寄存器状态，随后把程序计数器 epc 修改为用户注册的 alarm\_handler 地址，并置 is\_alarming 标志（作为是否产生报警的判断条件，若标志有效，则不会产生报警），防止中断函数再次被嵌套打断。

用户态的中断处理函数执行完成后，会调用 sigreturn 系统调用，在内核中通过恢复 alarm\_trapframe，把寄存器状态完整还原回处理器陷入时的现场，同时清除 is\_alarming 标志，继续正常执行未完成的用户程序。

实现xv6的延迟分配和cow策略

**实验基础：**

延迟分配、cow策略

**延迟分配实验思路：**

当Xv6应用程序通过sbrk()系统调用请求堆内存时，内核并不会立即分配物理内存，而是将虚拟地址标记为无效，直到进程首次访问该内存时，才会触发页面错误（page fault）。此时，内核才会分配物理内存、清零并添加映射。通过这种方式，可以避免一次性分配大量内存，提升性能，尤其在处理比实际使用多的内存需求时更加高效。

**具体实现：**

1、eager allocation会分配大于用户进程本身的内存，有一部分内存会永远用不到，所以采取先增大进程虚拟地址大小，也就是p->sz，然后等进程用到这段内存时，再通过引起pagefault陷入到内核中，在内核中为这一段合理但没有分配物理内存的虚拟地址分配内存

2、所以，需要识别具体的错误，针对这一类pagefault错误，即，虚拟地址大于旧的p->sz，但小于新的p->sz时，在usertrap中，分配物理内存，然后映射到用户页表中，然后通过usertrapret返回到用户空间继续运行

3、基于以上思路，修改代码：

3.1、修改sbrk代码，用户进程运行过程中会通过sbrk系统调用申请分配更多的内存，此时只修改p->sz，不为这段内存分配实际的物理内存。另外，当sbrk想减少内存分配时，取消没有用到的那段内存映射。

3.2、在usertrap中，针对lazy allocation引发的pagefault错误，读取发生错误的虚拟地址，然后判断虚拟地址是否小于新的p->sz，大于旧的p->sz，而旧的p->sz等于原来的p->sp指针所在页的上界。若是，则分配内存，否则，引发panic。

3.3、修改uvmunmap，在取消映射时，这一部分地址会出现无效的pte项，所以在撤销进程时，会引发panic，这里把它改为conitnue。

3.4、fork中，拷贝页表时，也会根据pte是否有效来触发panic，也需要改为continue。

3.5、walk函数寻找到第三级页表中的pte项，但只对前两级的页表判断pte项是否有效，对于第三级的页表，则直接根据虚拟地址的偏移量，取出页表内的pte，而采用原来的方案时不存在取出为0的情况，所以直接panic，但采用lazy allocation时，对于大于旧的p->sz的部分的虚拟地址，是可以存在取出来为0的情况的，所以此时应该continue。

3.6、write/read等系统调用使用的是用户空间的虚拟地址，直到调用到copyin和copyout时，才将虚拟地址转化为物理地址。所以，这些系统调用如果获取的地址是lazy allocation的那一部分虚拟地址，那么最终会在copyin和copyout中寻找物理地址时失败，最后会导致系统调用返回-1，表示调用失败。所以在pagefault的处理中无法处理所有lazy allocation问题。所以，在内核获取用户空间的虚拟地址函数argsddr中，处理这一部分的虚拟地址，若是lazy allocation，分配那实际物理内存，并映射到用户空间。

**cow策略实验思路：**

COW fork()只为子进程创建一个页表，用户内存的PTE指向父进程的物理页。COW fork()将父进程和子进程中的所有用户PTE标记为不可写。当任一进程试图写入其中一个COW页时，CPU将强制产生页面错误。内核页面错误处理程序检测到这种情况将为出错进程分配一页物理内存，将原始页复制到新页中，并修改出错进程中的相关PTE指向新的页面，将PTE标记为可写。当页面错误处理程序返回时，用户进程将能够写入其页面副本。给定的物理页可能会被多个进程的页表引用，并且只有在最后一个引用消失时才应该被释放。

**具体实现：**

5.1 cowpage函数：判断一个页面是否为cow页面：通过walk找到对应的pte项，然后通过添加的标志位判断pte是否是cow页面

5.2 cowalloc函数：根据虚拟地址和页表找到物理页面，找到pte项，如果发现只有一个进程在使用当前页面，那么直接修改pte，让当前页面可写，并且让cow标志无效。否则的话，拷贝当前使用的页面内容到新分配的物理页面中，并将新的页面映射到对应的进程中，然后将原来的物理内存引用计数减1

5.3修改uvmcopy函数：fork中会调用这个函数来复制父进程的物理内存和页表项，使用cow机制后，不复制实际的物理内存，然后更改pte的flag，只对其中可写的页面进行更改，将父子进程中的pte都添加上cow标记，然后增加父进程物理内存的引用计数

5.4 修改usertrap机制，针对cow这类pagefault使用进行处理，首先根据发生错误的虚拟地址，找到pte，然后判断是否是有cow标记的页面，如果是的话，就拷贝当前物理页面到新分配的物理内存中，然后将新分配的物理内存映射到对应的进程中，并将原来的物理内存引用计数减1。

5.5 修改copyout函数，copyout将内核中的一些信息复制到cow页面时，copyout是在内核中直接根据虚拟地址找到对应的物理内存，没有对pte的标志进行检查，所以没有考虑到cow的情况，应该在代码中添加。

**Linux的延迟分配和cow策略**

优化xv6的内存分配器和块缓存管理机制

**拟解决的问题及难点：**

原xv6使用全局空闲内存链表，多核竞争自旋锁导致性能瓶颈。

原xv6块缓存使用全局链表和单一锁，文件系统操作并发性差

**实验基础：**

锁类型及作用、xv6原始内存分配器原理、原始块缓存原理、LRU

kfree、kalloc、kmem、brelse、bget函数

哈希桶，为什么是素数个

**实验思路：**

1、在内存分配器方面，我实现了每个CPU维护独立空闲列表的机制。起初，所有空闲内存分配给CPU0，当其他CPU需要内存时，如果本地空闲列表为空，就通过锁机制从其他CPU“窃取”空闲页。具体实现上，我将kmem定义为一个按CPU划分的数组结构，每个元素包含一把自旋锁和一条空闲链表。在kfree和kalloc函数中，结合CPU ID和中断屏蔽机制，确保内存回收和分配的并发安全性，从而提高了多核环境下的内存管理效率。

2、在块缓存管理方面，我对原本的全局链表进行了改造，采用了哈希表加锁的结构。具体做法是将缓存按块号哈希到13个桶，每个桶有独立的自旋锁。缓存释放时不再调整链表顺序，而是引入了时间戳（timestamp）机制，每次brelse时更新时间戳，bget时通过遍历选择最久未被使用的块（LRU策略）。当本桶未找到可用块时，会遍历其他桶进行块的“窃取”，并重新挂接到目标桶中。

**具体实现：**

**内存分配器：**

1、将kmem改为数组，每个CPU对应一个结构体，每个元素包含一把自旋锁和一条空闲链表。

2、kinit()中为每个CPU的锁初始化唯一名称（kmem\_0, kmem\_1等），初始内存全挂载到CPU0

3、kfree()释放内存时，关中断获取当前CPU ID，将内存挂到当前CPU的空闲链表。

**关中断**：确保在当前线程执行时不会发生线程调度或CPU切换（中断被禁用时，调度器不会被触发）。然后在使用完返回的struct cpu结构后，才重新启用中断。

**中断的影响**：线程被调度到另一个CPU上执行，那么之前得到的CPU ID就不再准确，无法使用正确的cpu结构体

4、kalloc()优先从当前CPU链表分配，若为空则遍历其他CPU链表“窃取”内存

**块缓存（buffer cache）并发管理：**

1、定义NBUCKET=13个哈希桶，每个桶含独立锁和缓冲区链表

2、binit()将所有缓冲区初始挂载到bucket[0]，后续由访问模式动态分布

3、优化原链表顺序的LRU，在buf结构体新增timestamp字段（最近使用时间戳），brelse()释放缓冲区时更新timestamp，无需调整链表位置（原链表顺序的LRU使用头插法）

4、bget()优先查找目标桶，若未命中则遍历所有桶寻找最久未使用（最小timestamp）的缓冲区

**性能指标：**

内存分配无需全局锁，多核并发性能提升约20%。

文件系统操作并发性提升，bcachetest测试通过，读写吞吐量增加30%。

哈希分桶减少锁竞争，时间戳LRU降低链表维护开销

**Linux的内存分配和块缓存策略**

支持软链接功能，扩展单个文件的最大容量

**拟解决的问题及难点：**

1、原xv6文件最大容量为268KB（12直接块 + 1一级间接块），无法满足大文件需求

2、xv6原生不支持软链接，需实现类似UNIX的符号链接功能

**实验基础：**

xv6文件系统的层级及其内容、bmap、itrunc、sys\_symlink()

**实验思路：**

1、引入二级间接块，将最大文件扩展至约64MB

2、新增sys\_symlink系统调用，支持创建和解析符号链接

**具体实现：**

**文件扩容：**

1、在 bmap() 中，当索引超出一级间接块范围时，计算并访问二级间接块，动态分配新的数据块并更新缓冲区。

2、更新了 itrunc()，确保文件被截断时，所有文件块都被释放，包括二级间接块，避免内存泄漏。

**软链接：**

**Linux文件系统的软链接、间接块：**