Xv6 Lab Report 2021

Xv6课程实验报告

2251730 刘淑仪 操作系统课程设计

同济大学软件学院

2024 年 8 月 15 日

01

实验内容概述

Overview of experimental content

Lab1 : Xv6 and Unix utilities

实验目的：熟悉 xv6 及其系统调用。

Boot XV6 : 熟悉和理解 xv6 操作系统的启动过程以及基本操作。通过获取 xv6 源代码、编译并运行它，实验者将能够观察到操作系统从启动到提供基本命令行界面的整个过程。实验还涉及如何使用 Git 来管理代码的更改，从而帮助跟踪和记录实验进展。最终，实验者通过运行一些基本命令和查看进程信息，进一步加深对 xv6 内部工作的理解。

Sleep : 在 xv6 操作系统下实现一个基本的 sleep 程序。通过这个实验，用户将学会如何创建一个用户级应用程序，该应用程序能够暂停指定数量的时间片。实验涉及解析命令行参数、调用 xv6 提供的 sleep 系统调用，以及处理异常输入。利用 argc != 2 检查参数个数是否正确，若不正确使用 exit(1) 异常退出；正确则利用系统调用 sleep。注意其中需要将输入参数 argv[1] 通过 atoi 从字符串转换为 int，才可作为 sleep 的输入。需要注意的是 argv[0] 存储的是执行程序的文件名或路径。

Pingpong : 写一个 pingpong 程序，来学习进程间通信和进程同步的概念。具体来说，实验要求编写一个程序，利用 UNIX 系统调用通过一对管道在父进程和子进程之间传递一个字节。父进程向子进程发送一个字节，子进程收到后打印出收到的消息，并将字节回传给父进程，父进程再打印出收到的消息。通过这个实验，用户能够理解父子进程的关系、如何使用管道进行通信，以及如何通过系统调用确保进程在适当的时机进行同步。

Lab1 : Xv6 and Unix utilities

实验目的：熟悉 xv6 及其系统调用。

Primes：编写一个并发版本的素数筛选算法来学习进程间通信与并发编程。该算法使用 UNIX 系统调用，通过管道实现父进程和子进程之间的数据传递，递归地筛选素数。实验的核心在于理解如何利用管道和 fork 创建多个子进程，每个子进程通过管道接收前一个进程筛选后的数据，并筛选出素数后再传递给下一个进程。这种方法展示了进程同步与并发执行的概念，并帮助用户掌握进程间通信的机制和实现细节。

Find：编写一个简单版本的 UNIX find 命令，递归查找目录树中具有特定名称的所有文件。通过这个实验，用户将学习如何使用 xv6 操作系统中的系统调用和文件系统接口操作文件和目录，理解文件系统的结构，并应用递归算法进行目录树的遍历和文件查找。此外，实验还涵盖了字符串处理和错误处理的基本方法。

Xargs：编写一个简单版的 UNIX xargs 程序，从标准输入读取行并将其作为参数传递给指定命令执行。通过该实验，用户将学习如何处理命令行参数、从标准输入读取数据、并通过创建子进程和调用 exec 函数执行外部命令。同时，实验还涉及到进程管理和标准输入输出的处理，帮助用户理解和实现基本的 UNIX 命令操作。

Lab2 : System Calls

实验目的：本实验旨在帮助了解系统调用跟踪的实现，并演示如何修改 xv6 操作系统以添加新功能。实验任务是添加一个新的 trace 系统调用，该调用用于调试。具体来说，需要创建一个名为 trace 的系统调用，并将一个整数 "mask" 作为参数。"mask" 的各个位表示要跟踪的系统调用。通过本实验，将熟悉内核级编程，包括修改进程结构、处理系统调用以及管理跟踪掩码。

Lab2 : System Calls

System call tracing : 在 xv6 操作系统中实现一个系统调用追踪功能，通过创建一个新的 trace 系统调用，允许用户指定一个掩码来选择性地追踪特定的系统调用。通过实验，用户将学习如何在操作系统内核中添加新的系统调用，如何处理进程间的继承与状态管理，以及如何实现内核功能的调试输出。实验内容包括修改内核代码以支持新的系统调用、实现系统调用追踪的逻辑，以及通过命令行工具测试该功能的正确性。

Sysinfo : 在 xv6 操作系统中添加一个名为 sysinfo 的系统调用，用于收集系统的运行信息。通过该实验，用户将学习如何定义和实现一个新的系统调用，包括如何获取系统的可用内存和活动进程数量等信息，并返回给用户态程序。实验涉及到进程状态的遍历、内存管理的实现、以及在内核中处理并发访问时的加锁机制。用户还将学会如何调试和解决系统调用实现中的实际问题。

Lab3 : Page tables

旨在让学生理解和掌握操作系统中的分页机制和页表管理。

1. 理解虚拟内存管理： 实验深入探讨了分页机制，这是现代操作系统内存管理的核心技术之一。通过学习如何实现和操作页表，学生能够理解虚拟内存如何将进程的虚拟地址映射到物理地址，从而隔离进程并提高系统的安全性和稳定性。
2. 熟悉页表结构和操作： 实验中，学生需要操作和管理页表，包括创建页表、设置页表项、解析虚拟地址等操作。通过这些任务，学生能够熟悉操作系统如何利用页表管理内存，并理解页表的多级结构、权限控制和地址翻译过程。
3. 提高对操作系统机制的理解： 通过实现与页表相关的功能，如分页保护、地址转换、内存分配等，学生能够更深刻地理解操作系统在内存管理中的作用和重要性。

Lab3 : Page tables

Speed up system calls : 通过在 xv6 操作系统中优化 getpid() 系统调用，学习如何将页表映射到用户空间，以加速系统调用的执行。实验内容包括为每个进程分配一个只读的 usyscall 页面，将其映射到用户空间的特定地址，并在页面中存储当前进程的 PID。通过这种方式，用户态程序可以直接访问进程信息，而不必每次都进入内核，从而提高系统性能。实验还涉及到内存管理、页表映射以及虚拟地址到物理地址的转换等操作系统核心机制的实现与调试。

Print a page table : 通过实现 vmprint() 和 pgaccess() 系统调用，深入理解和操作 RISC-V 页表结构，以便可视化页表内容并检测页面访问情况。在实验的第一部分，通过编写 vmprint() 函数，我们能够输出并分析进程的页表结构，为后续调试提供支持。第二部分的实验中，通过实现 pgaccess() 系统调用，可以检测哪些页面在运行过程中被访问过，这对内存管理优化，如垃圾回收器的设计，具有实际应用意义。实验内容涉及页表遍历、标志位检测与清除、内存映射与管理等核心操作系统概念。

Lab4 ： Traps

关注操作系统中的中断和陷阱机制，这是理解操作系统如何处理硬件事件和系统调用的关键部分。实验的意义主要体现在以下几个方面：

1. 理解中断和陷阱机制： 通过实验，学生可以深入了解操作系统如何通过中断和陷阱机制处理外部硬件事件（如时钟中断、设备中断）和软件异常（如系统调用、非法操作）。这些机制是操作系统与硬件交互的核心。
2. 掌握陷阱处理流程： 实验指导学生实现并调试陷阱处理的各个步骤，包括保存和恢复进程的状态、识别陷阱的类型、调用适当的处理程序，以及在处理完成后返回用户态。这有助于理解操作系统如何在多任务环境中维持进程的正常执行。
3. 深入理解系统调用的实现： 实验中涉及到系统调用的处理流程，学生通过实验可以理解操作系统如何捕获系统调用请求，并将其转交给相应的内核函数处理。这对于理解操作系统的服务接口和内核功能的实现至关重要。
4. 提高异常处理的能力： 学习如何处理各种异常情况，如非法内存访问、除零错误等。这些异常处理机制是操作系统提供稳定性和安全性的基础。

Lab4 ： Traps

RISC-V assembly : 通过解析和理解 RISC-V 汇编代码，深入学习汇编语言的基本原理和RISC-V的指令集架构。实验内容包括生成可读的汇编代码文件并分析其中的函数调用、寄存器使用、地址定位和数据存储等细节。通过回答与汇编代码相关的具体问题，实验帮助学生掌握如何从汇编代码中识别函数参数的传递、函数的内联优化、返回地址的保存等关键概念，并理解小端和大端存储的差异以及由参数不匹配导致的未定义行为。这些知识对于理解底层计算机体系结构和编写高效的系统级代码非常重要。

Backtrace : 实现一个回溯（backtrace）功能，用于在操作系统内核发生错误时输出调用堆栈上的函数调用列表。这一功能对于调试和定位错误非常有帮助。实验内容包括在 riscv.h 文件中添加读取栈帧的函数 r_fp()，编写 backtrace() 函数以遍历和输出栈帧信息，并在关键位置（如 panic() 和 sys_sleep() 函数中）调用 backtrace()。深入理解 RISC-V 架构下的栈帧结构、函数调用过程以及如何在操作系统中实现和利用回溯功能来帮助调试复杂的内核问题。

Alarm : 实现一个定时提醒功能，通过添加 sigalarm 和 sigreturn 系统调用，使得用户进程能够在经过一定的 CPU 时间后执行指定的处理函数。实验内容包括在进程结构体中添加相关字段记录定时器状态，编写系统调用函数以设置和恢复定时处理函数，并在时钟中断处理中实现定时器逻辑。深入理解操作系统中断处理机制、用户态与内核态的转换过程以及如何处理复杂的状态管理与异常恢复。这一功能在处理需要周期性操作的任务中非常有用，并为进一步学习信号处理和异常处理机制打下基础。

Lab5 : Copy on-write

通过实现写时复制（COW）技术，优化 fork() 系统调用的内存使用效率。通过使父子进程在初始时共享物理内存页面，仅在写入时才进行实际的页面复制，这一实验旨在减少内存复制开销，提高系统性能，同时帮助学生深入理解操作系统中的内存管理机制和页表操作。

在xv6操作系统中实现写时复制（Copy-on-Write, COW）的 fork() 功能。传统的 fork() 调用会复制整个进程的内存空间，而COW通过让父子进程共享相同的物理内存页，只有在写入时才进行真正的复制，从而减少内存使用和提高系统性能。实验的主要内容包括修改页表映射、添加内存引用计数、处理页面错误等。通过这一实验，学生可以深入理解操作系统内存管理的优化技术以及如何高效地处理进程的创建和内存分配。

Lab6 : Multithreading

实现多线程支持，使得一个进程可以拥有多个线程来并发执行任务。实验的主要内容包括在内核中引入线程控制块（TCB），实现线程的创建、切换、同步和销毁功能，以及确保多线程环境下的资源共享和安全性。通过这一实验，学生可以深入理解操作系统中的线程管理机制，掌握多线程编程的基本原理和实现方法，从而提高并发编程的能力和对复杂系统的掌控力。

Lab6 : Multithreading

Uthread: switching between threads：设计和实现一个用户级线程系统的上下文切换机制，目标是能够在不同的用户线程之间切换执行。实验的主要步骤包括在uthread.c中添加线程的创建和上下文保存/恢复功能，编写thread_switch函数进行线程间的切换，并确保线程系统能够正确运行。通过这个实验，学习如何在用户态实现线程的上下文切换，掌握多线程的基本原理和实现方法。

Using threads：通过使用线程和锁来实现并行编程，并在多线程环境下处理哈希表。实验的目的是让学生学习如何使用线程库创建和管理线程，同时通过加锁机制来保护共享资源，确保哈希表在多线程环境下的正确性和性能。通过实验，学生可以理解多线程编程中数据一致性问题的解决方法，并通过优化锁的使用来提高程序的并发性能。

Barrier：通过实现一个线程屏障来加深对多线程编程中同步和互斥机制的理解。线程屏障是一种同步机制，用于确保多个线程在某个点上等待，直到所有线程都到达该点后才能继续运行。通过使用pthread条件变量和互斥锁，实验引导学生实现一个线程安全的屏障机制，解决多线程环境下的同步问题。这有助于理解多线程编程中的同步、竞争条件的处理以及并发程序的正确性和性能优化。

Lab7 ： Networking

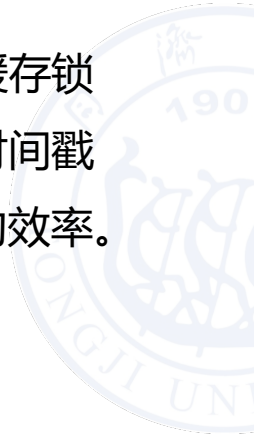
通过为xv6操作系统中的网络接口卡（NIC）编写设备驱动程序，深入理解网络通信的核心机制。实验包括实现以太网驱动程序、处理ARP请求和响应、实现IP数据包的发送和接收、处理ICMP Echo请求和响应、实现UDP数据报的发送和接收，以及编写一个简单的DHCP客户端。通过这些步骤，学生可以学习到网络协议栈的工作原理，理解如何在操作系统中实现网络通信功能，并解决网络编程中的实际问题。

Lab8 ： Lock

扩展文件系统以支持更大的文件尺寸。实验内容包括修改inode结构，将一个直接块号替换为双重间接块号，从而增加最大文件大小。实验过程中需要更改宏定义、更新inode结构，并修改如bmap()和itrunc()等函数以处理新的块结构。实验最后通过bigfile测试和usertests验证了修改的正确性。

Memory allocator：通过为每个CPU分配独立的自由列表（free list）和锁，减少多核系统中的锁争用，提升内存分配器的性能。具体内容包括修改内存分配器的设计，使每个CPU拥有自己的内存分配链表，并在链表耗尽时从其他CPU的链表中偷取部分内存页。通过这种改进，可以显著减少CPU之间的竞争，提高系统在多线程环境下的内存管理效率，从而增强操作系统的整体性能。

Buffer cache：通过对xv6操作系统中的缓冲区缓存（buffer cache）进行优化，以减少多个进程之间对缓冲区缓存锁的争用，进而提升系统的性能和并发能力。具体方法包括将原有的双向链表结构更改为哈希表结构，并使用基于时间戳的LRU算法管理缓存块。这些改动旨在减少锁的竞争，优化多线程环境下的缓冲区管理，从而提高整个操作系统的效率。



Lab9 : File System

扩展文件系统以支持更大的文件尺寸。实验内容包括修改inode结构，将一个直接块号替换为双重间接块号，从而增加最大文件大小。实验过程中需要更改宏定义、更新inode结构，并修改如bmap()和itrunc()等函数以处理新的块结构。实验最后通过bigfile测试和usertests验证了修改的正确性。

Large Files：通过为每个CPU分配独立的自由列表（free list）和锁，减少多核系统中的锁争用，提升内存分配器的性能。具体内容包括修改内存分配器的设计，使每个CPU拥有自己的内存分配链表，并在链表耗尽时从其他CPU的链表中偷取部分内存页。通过这种改进，可以显著减少CPU之间的竞争，提高系统在多线程环境下的内存管理效率，从而增强操作系统的整体性能。

Symbolic links：通过实现符号链接功能，通过修改文件系统的路径解析和锁机制，支持符号链接的创建、解析、以及循环检测，从而增强文件系统的灵活性和功能性。实验成功实现了符号链接的功能，并通过了所有测试验证。

Lab10 : Mmap

为 xv6 操作系统添加 mmap 和 munmap 系统调用，以实现文件内存映射功能，支持共享内存和将文件映射到进程地址空间等功能。通过实现这些系统调用，实验深入探讨了虚拟内存管理、页面错误处理、以及进程地址空间的管理。最终，实验成功通过了 mmap 测试，证明了内存映射功能的正确性和有效性。

02

实验中遇到的问题及解决

Problems encountered in the experiment and solutions

Lab1 : Util

Primes :

- 问题：父进程向管道中写入数据后，子进程可能会因为管道中的数据尚未完全写入而无法正确读取，导致素数筛选过程中的数据丢失或错误。
- 解决：父进程在将数字2到35写入管道后，关闭写端 `p[1]`，表示写入完成。子进程在读取数据时，正确处理读端口关闭的情况，通过检测读取返回值是否为0来判断管道是否已经关闭。

Lab2 : Syscall

Syscall tracing :

- 命名问题导致的错误：最初，我将 `tracemask` 命名为 `mask`，在实现过程中遇到了一些寄存器读取错误。由于在函数实现和调用过程中，`mask` 这个名字可能会与其他变量名冲突，导致读取和赋值操作出现错误。通过更改变量名为 `tracemask`，避免了这种命名冲突问题。

- 系统调用编号定义位置错误：在定义 `SYS_trace` 编号时，若未严格按照 `kernel/syscall.h` 文件中的顺序添加，可能导致系统调用编号错乱。因此，需要仔细检查和按照现有系统调用编号的顺序添加新的编号。

- `fork` 时 `tracemask` 继承问题：在实现子进程继承父进程 `tracemask` 的功能时，若未在 `fork` 函数中正确复制 `tracemask`，将导致子进程无法正确追踪系统调用。通过在 `kernel/proc.c` 文件中，添加 `np->tracemask = p->tracemask;` 解决了这个问题。

Lab2 : Syscall

Sysinfo :

> 在实现 `sysinfo` 系统调用时，出现了未定义引用 `sysinfo` 的链接错误。

确保在 `user/user.h` 中正确声明了 `sysinfo` 函数，并在 `user/usys.pl` 文件中正确添加了条目 `entry("sysinfo")`。

> 在获取可用内存大小时，访问链表节点可能导致竞争条件。

在遍历 `kmem.freelist` 链表时，加锁以确保线程安全。

> 在获取进程数量时，遍历进程表可能导致竞争条件。

在访问进程状态时，加锁以确保线程安全。

Lab3 : pgtbl

Speed up system calls :

- 问题- 在启动 xv6 内核时，遇到了以下错误：

```plaintext
xv6 kernel is booting
hart 2 starting
hart 1 starting
panic: freewalk: leaf
```

该错误通常与页表管理中的问题有关。具体来说，在处理页表释放时，可能存在对已经释放或未正确初始化的页表项进行操作的情况。

仔细检查页表的分配和释放逻辑，确保在分配页表时正确初始化各个页表项。最终发现问题出在 `proc_freepagetable()` 函数中，没有正确处理 `usyscall` 页面的释放。

- 页表映射问题：在实现 `mappages` 函数时，最初没有正确设置页面权限，导致 `usyscall` 页面无法正确映射。通过检查页表权限设置，我们确保了 `usyscall` 页面具有只读权限，从而解决了映射问题。

Lab3 : pgtbl

Detecting which pages have been accessed :

- 页表遍历中的错误：在实现 `sys_pgaccess` 时，最初遍历页表时漏掉了一些页，导致部分页面没有被正确检测到。这是由于在计算虚拟地址偏移时没有正确处理。通过仔细检查页表遍历逻辑，确保每个页面都被正确访问并检测到，解决了这个问题。

- 用户空间缓冲区的地址传递：最初实现时，没有正确处理用户空间缓冲区地址的传递和结果拷贝。通过使用 `copyout` 函数，将内核空间的结果拷贝到用户空间，解决了这个问题。

- 清除 `PTE_A` 标志位：在检测到页面被访问后，需要清除 `PTE_A` 标志位以便于后续的访问检测。最初实现时，忘记清除该标志位，导致重复检测到同一页面。通过在检测到访问后手动清除 `PTE_A` 标志位，解决了这个问题。

Lab4 : Traps

Backtrace :

- 获取上一级栈帧的终止条件：在实现过程中，需要考虑如何正确识别和处理栈帧的终止条件。通过使用 `PGROUNDDOWN()` 和 `PGROUNDUP()` 函数，计算栈帧所在页面的边界地址，确保循环在合理的边界内运行，避免越界访问。

- 栈帧指针的有效性检查：在遍历栈帧时，需要确保栈帧指针 `fp` 的有效性。通过检查 `fp` 是否在用户栈空间页面的范围内，确保访问的地址是合法的，避免出现异常访问和崩溃。

- 多级调用栈的处理：在输出栈帧信息时，需要正确处理多级调用栈。通过依次访问每一级栈帧并输出相应的返回地址，保证调用栈信息的完整性和准确性。

Lab4 ： Traps

Alarm :

`trapframe` 副本的创建与恢复：我们通过在时钟中断中直接修改进程的 `trapframe` 来实现定时提醒功能，但在运行测试时发现 `test1/test2()` 无法通过。这是因为在执行定时函数 `handler` 后，进程无法正确恢复到中断前的状态，导致程序继续执行时出现错误。为了解决这个问题，我们需要在进入定时函数前保存进程的 `trapframe` 状态，在定时函数执行完成后恢复该状态。为此，我们在进程结构体 `struct proc` 中添加了一个新的字段 `trapframecopy` 来存储 `trapframe` 的副本。

Lab5 ： Cow

一开始不是很理解 PHYSTOP 和 KERNBASE ，以及为什么 cow 记录页面引用个数的数组的大小如下定义 cow[(PHYSTOP - KERNBASE) >> 12] ，后来查阅了相关资料才知道，在xv6系统中，物理内存并不是从0开始的，而是从 KERNBASE 开始，其上才是地址空间，而其下用于一些关于硬件I/O的接口。然后我们看到 PHYSTOP 的定义#define PHYSTOP (KERNBASE + 128*1024*1024) ，这个说明物理地址的结束位置，一共有128MB的偏移量，再大的地址就是属于Unused了。

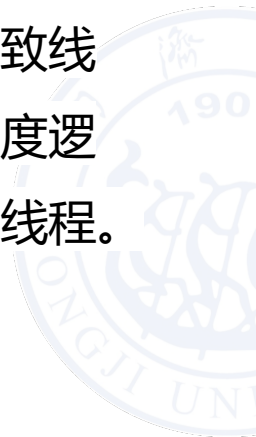
Lab6 ： Thread

Uthread: switching between threads :

1. 线程调度不正确：在 `thread_schedule()` 函数中，无法正确找到 RUNNABLE 状态的线程，导致线程无法正确调度。因此，在 `thread_schedule()` 函数中添加调试信息，验证线程数组的状态变化和调度逻辑的正确性。确保从当前线程的位置开始，正确地遍历线程数组，并找到下一个 `RUNNABLE` 状态的线程。

2. 上下文切换失败：在 `thread_switch()` 函数中，无法正确保存和恢复寄存器状态，导致线程切换失败或程序崩溃。因此，检查 `thread_switch` 的汇编代码，确保寄存器的保存和恢复操作正确无误。可以对照 `kernel/swtch.S` 中的 `swtch` 函数，逐行检查代码的正确性。此外，可以通过打印寄存器值或使用调试器，验证寄存器的正确保存和恢复。

3. 线程调度不正确：在 `thread_schedule()` 函数中，无法正确找到 `RUNNABLE` 状态的线程，导致线程无法正确调度。因此，在 `thread_schedule()` 函数中添加调试信息，验证线程数组的状态变化和调度逻辑的正确性。确保从当前线程的位置开始，正确地遍历线程数组，并找到下一个 `RUNNABLE` 状态的线程。



Lab6 ： Thread

Using Threads :

1. 线程安全问题：最初的哈希表在多线程环境下会导致数据丢失。通过为每个 `bucket` 添加互斥锁，我们成功地解决了这个问题。

2. 性能问题：在加锁过程中，需要确保锁的粒度尽可能小，以提高并发性能。通过将加锁范围缩小至 `insert()` 函数，我们减少了锁的争用，提升了程序性能。

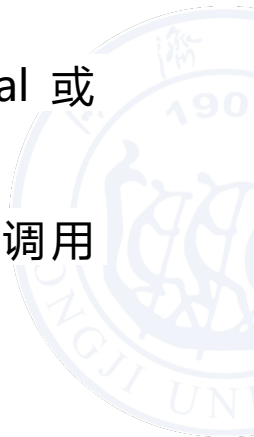
3. 锁的争用：当多个线程同时访问同一个 `bucket` 时，会导致锁的争用，从而影响性能。通过增大 `NBUCKET` 的值，减少了锁的争用，提高了并发性能。

Barrier :

对几个函数的理解：

pthread_cond_wait 函数用于将当前线程阻塞，并等待条件变量的信号。当其他线程调用 p thread_cond_signal 或 pthread_cond_broadcast 发送信号时，被阻塞的线程将被唤醒并继续执行

pthread_cond_wait 函数需要与互斥锁一起使用，以确保线程在等待条件变量时不会出现竞态条件。在调用 pthread_cond_wait 之前，通常需要先获取互斥锁，然后在等待期间释放互斥锁，以允许其他线程修改共享数据。



Lab7 : Net

XXXXXXX

Lab8 ： Lock

Memory Allocator:

1. 锁争用的显著减少：实验结果显示，通过为每个CPU设置独立的锁与自由列表，锁的争用情况有了明显改善。特别是在 `kalloctest` 测试中，`acquire()` 函数的调用次数大幅减少，自旋尝试获取锁的次数也降为零，这表明大部分内存分配操作在无锁竞争的情况下顺利进行。这一改进使得系统在高并发场景下能够更高效地管理内存分配，避免了不必要的等待和资源浪费。

2. 系统稳定性与兼容性：通过测试结果可以看到，经过修改后的内存分配器在运行 `kalloctest` 和 `usertests sbrkmuch` 测试时均表现出色，所有测试均通过。这表明我们在优化锁机制的同时，没有引入新的不稳定因素，系统能够稳定地运行并兼容现有的用户程序。这对于实际系统应用非常重要，因为它意味着改进不仅提高了性能，还保证了系统的可靠性。

3. 偷取物理页策略的有效性：我们实现了当某个CPU的自由列表为空时，从其他CPU的自由列表中偷取部分内存的策略。在实际测试中，该策略能够有效防止某个CPU因为缺乏空闲内存页而陷入停顿，从而保持系统的整体平衡与性能。这个策略在多核系统中尤其重要，因为不同的CPU可能会因为不同的负载情况而消耗不同数量的内存页，通过这样的动态调整，可以确保各个CPU之间的资源分配更加合理。

Lab8 ： Lock

Buffer Cache :

1. > 在 `xv6` 中执行 `bcachetest` 出现 `freeing free block` 的 `panic`

在实现基于哈希表的缓冲区管理机制时，当在某个 `bucket` 中找到可重用的缓存块时，如果错误地将其从 `bucket` 中移除，而不是仅仅更新或重新使用它，就可能导致缓存块在缓存系统中的丢失。具体表现为，系统在后续操作中试图访问该缓存块时，会检测到缓存块已被移除，而实际上这些块仍然应该在缓存中，从而引发 `freeing free block` 的 `panic` 错误。在本次实验中，当在目标 `bucket` 找到可重用的缓存块时错误地在 `bucket` 中移除了缓存块, 而此处不应该移除, 否则会导致缓存块的丢失。

2. > 在 `xv6` 中执行 `bcachetest` 通过, 而执行 `usertests` 在 `test manywrites` 中卡住

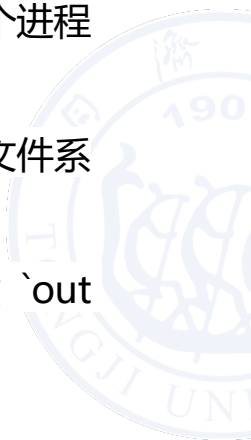
经过分析，此处的原因即将 `lock` 和 `hashlock` 二者作为同一个锁使用，造成了死锁。在新的缓冲区缓存管理机制中，`lock` 和 `hashlock` 分别用于不同的目的：

- `lock`：用于保护单个 `bucket` 中的操作，确保在多进程并发访问时，只有一个进程能够安全地操作该 `bucket`。
- `hashlock`：用于保护整个哈希表的全局操作，比如在需要遍历或操作多个 `bucket` 时，确保这些操作的原子性和一致性。

在某些情况下，如果将 `lock` 和 `hashlock` 误用为同一个锁，那么当一个进程持有 `hashlock` 并尝试获取 `lock` 时，另一个进程可能已经持有 `lock` 并尝试获取 `hashlock`，这就导致了死锁——两个进程都在等待对方释放锁，从而卡住了系统。

3. > `writebig` 测试在运行时触发了一个 `panic`，具体是 `balloc: out of blocks` 错误。这个错误通常意味着在分配新块时，文件系统已经没有可用的块了。

`xv6` 的文件系统在格式化时分配了一定数量的块供使用。如果在进行较大写入操作时，文件系统的可用块被耗尽，便会触发 `out of blocks` 错误。在 `param.h` 修改 `FSSIZE` 的大小为 10000。



Lab9 ： Fs

Large Files :

1. > 在修改 `bmap()` 函数时，由于需要处理二级间接块，索引的复杂性显著增加。一级间接块的处理相对简单，因为只需要一次索引即可获取目标数据块的地址。然而，二级间接块涉及两次索引：第一次索引获取一级间接块的地址，第二次索引获取实际数据块的地址。这要求在代码实现中准确管理这两个层次的索引关系，否则可能导致数据块访问错误或系统崩溃。

通过仔细设计 `bmap()` 函数的逻辑，确保在处理二级间接块时正确地获取和使用每个索引值。同时，在调试阶段，增加了对索引值的检查和验证，确保函数在不同情况下都能正确返回数据块的地址。

2. > 在 `itrunc()` 函数的实现中，由于需要释放的块数增加，释放过程变得更加复杂。除了释放直接块和一级间接块外，还需要遍历并释放二级间接块及其引用的一级间接块。这一过程中，任何遗漏或错误都可能导致内存泄漏或其他潜在问题。

通过增加多层次的循环来遍历和释放所有相关块，并严格管理每一层次的释放操作，确保所有分配的块在不再需要时都能被正确释放。同时，加入了更多的错误检查机制，以捕捉并处理可能出现的异常情况。

Lab9 ： Fs

Symbol links :

1. 符号链接递归深度问题：在实现符号链接时，我们引入了 `NSYMLINK` 常量来限制符号链接的递归深度。这是为了防止由于符号链接形成链式引用而导致的无限循环问题。然而，在实际测试中发现，当递归深度设定过低时，某些合法的深层符号链接无法正常解析，导致文件无法打开。为了解决这一问题，我们调整了 `NSYMLINK` 的值，使其能够满足大多数符号链接的使用需求，同时避免对系统性能的过度消耗。

2. 符号链接的成环检测：成环检测的实现最初采用了简单的数组记录已访问的 `inode` 编号，这种方式虽然实现简便，但在实际应用中，尤其是在处理大规模符号链接时，可能会引入性能瓶颈。为此，我们优化了成环检测算法，通过使用散列表（`hash table`）来加速 `inode` 编号的查找和比较，显著提升了检测效率。

3. 锁的管理与释放：在实验过程中，我们发现符号链接处理中的锁管理尤为复杂。尤其是在递归解析符号链接时，如何正确地管理 `inode` 的锁以及避免死锁成为一个关键问题。在多次调试后，我们确定了在 `follow_symlink()` 函数中对 `inode` 的锁定与释放顺序，并在多个关键操作点上加入了额外的检查与日志记录，以确保锁的管理能够正确进行，避免了潜在的死锁情况。

Lab10 : Mmap

1. 虚拟内存区域管理：在实现 `mmap` 系统调用时，需要确保在进程的虚拟地址空间中合理分配一个连续的虚拟内存区域，并记录该区域的信息。为了管理这些虚拟内存区域，我们引入了 `VMA` 结构体，并在 `proc` 结构体中添加了 `vma_pool` 字段。最初在设计 `VMA` 池的分配和释放机制时，如何高效地管理这些 `VMA` 条目成为一个难点。解决方案是在 `vma_pool` 中维护一个固定大小的数组，并通过标记条目的使用状态来分配和释放 `VMA`，这使得管理逻辑更加简单高效。

2. 页面错误处理与物理内存分配：实现 `mmap` 系统调用时，我们采用了延迟分配（`Lazy Allocation`）的策略，这意味着在调用 `mmap` 时只分配虚拟地址空间，并不立即分配物理内存。物理内存的分配是在发生页面错误时进行的。因此，在 `usertrap` 中添加页面错误处理逻辑变得至关重要。在实现过程中，最初的页面错误处理逻辑没有正确分配物理页面，导致进程访问 `mmap` 内存时出现段错误。通过调试和分析，我们确定了问题出在页面表的更新和物理内存分配的时机上。最终，通过修改 `vm.c` 中的 `uvmcopy` 和 `uvmunmap` 函数，确保在发生页面错误时正确分配物理内存并更新页面表，问题得以解决。