

合肥工业大学

XV6-RISCV 设计文档

参赛队名：_____某某队

队伍成员：_____马合坤 张博文 于思琪

指导老师：_____田卫东 周红鹃

全国大学生计算机系统能力大赛

操作系统赛

模块实验创新赛道

2024 年 12 月

目 录

一、 Our-XV6-RISCV	1
1、介绍.....	1
2、整体架构.....	2
二、 Our-xv6-riscv 的设计与实现	6
1、系统调用跟踪.....	6
2、当前空闲内存/进程数	10
3、进程追踪.....	12
4、自旋锁性能优化.....	14
5、多级反馈队列调度器（MLFQ）	17
三、 总结.....	24
1、页表更新与 TLB 刷新：	24
2、系统调用的规范性设计：	24
3、多核并发调试：	24
4、 用户态内存管理：	24
5、 硬件相关操作的谨慎性：	24
四、未来计划.....	25
1、进程抢占支持.....	25
2、进程的内存隔离和共享.....	28
3、 增加文件权限支持.....	34
5、条件变量与读写锁支持.....	38
6、支持用户态线程库.....	38
7、支持动态内存分页.....	39
8、实现内核页缓存.....	39
9、 Shell 扩展	39
10. 网络支持.....	39
12. 崩溃恢复与日志系统.....	39

一、 Our-XV6-RISCV

1、介绍

实现进程管理和调度：

- 在 C 语言的框架下，我们实现了 **多级反馈队列调度器 (MLFQ)**，能够动态调整进程优先级，并通过时间片和队列调度逻辑实现了高效的进程调度。
- 支持 **进程抢占功能**，通过时钟中断机制强制切换当前运行的进程，保障了调度器的公平性。
- 增强进程的 **内存隔离和共享** 功能，设计并实现 `shm_alloc` 和 `shm_map` 系统调用，支持共享内存区域的创建和映射，从而为进程间通信 (IPC) 提供了基础支持。

内存管理：

- 我们基于 `xv6` 的原有架构，完善了虚拟内存的管理功能，实现了 **写时拷贝 (COW)** 的优化，降低了进程创建的内存开销。
- 实现用户态与内核态的 **堆栈分离和安全管理**，保障了用户态对内存操作的合法性和内核态的稳定性。

文件系统增强：

- 基于原有的文件系统，希望实现 **文件权限支持**，引入 POSIX 风格的文件权限字段（读、写、执行权限），并对系统调用（如 `open`、`chmod`）进行扩展。
- 部分实现 **日志结构文件系统 (LFS)**，通过事务日志的方式保障文件系统的崩溃一致性，并减少了磁盘写入次数，从而提升性能。

多核与并发优化：

- 为多核环境优化了自旋锁，设计并实现了 **Ticket Lock** 替代简单的自旋锁，同时引入了 **指数退避机制**，减少了总线争用和 CPU 资源浪费。
- 增强了多核调度的稳定性，使得调度器可以在多个核心上并行

调度进程。

用户态支持:

- 我们完成了用户态系统调用接口的设计与实现，支持用户态程序通过系统调用与内核交互。
- 基于用户态的扩展，我们设计了多个用户程序（如 `trace`、`ticketlocks...`），通过实际测试验证了操作系统各模块的功能性和稳定性。

2、整体架构

一、进程调度与管理优化

1. 多级反馈队列调度器（MLFQ）

·目标：实现一个更复杂的进程调度器，支持时间片动态调整，优先处理交互型进程。

·修改点：

扩展 `proc` 结构，增加优先级字段和时间片计数。

修改调度器的 `scheduler()` 实现，支持多队列优先级。

·效果：提高交互型进程的响应速度，同时保障 CPU 密集型进程能被公平调度。

·主要涉及文件：`proc.c`、`proc.h`、`param.h`、`trap.c`。

2. 进程抢占支持

·目标：允许内核中高优先级任务抢占低优先级任务的 CPU。

·实现思路：在每次时钟中断（`clockintr`）中，判断是否有更高优先级的进程需要运行。

·效果：提高实时任务的响应速度。

·主要涉及文件：`proc.c`、`trap.c`。

3. 进程的内存隔离和共享

·优化方向：

提供 `mmap` 功能：支持进程间共享内存段。

改进进程内存分配策略，优化 `uvmmalloc` 和 `uvmdealloc`。

·主要涉及文件：vm.c、sysproc.c。

二、文件系统增强

1. 增加文件权限支持

·目标：为文件增加权限管理（读/写/执行权限）。

·修改点：

修改 inode 结构，增加 mode 字段存储权限信息。

修改 fs.c 和 sysfile.c，根据权限检查操作是否合法。

·效果：支持更细粒度的文件权限管理。

·主要涉及文件：bio.c、fs.c

2. 日志结构文件系统（LFS）

·目标：优化文件系统写性能，通过日志实现写操作的快速提交。

·实现思路：

增加一个日志区域，所有写操作先写入日志，再提交到实际磁盘块。

效果：提高磁盘写性能，增强崩溃恢复能力。

·主要涉及文件：bio.c、fs.c。

3. 扩展文件系统 API

·扩展的接口：

truncate：支持动态缩小文件大小。

fchmod：修改文件权限。

getcwd：获取当前工作目录。

·主要涉及文件：sysfile.c、fs.c。

三、用户态与内核态交互优化

1. 用户态系统调用库优化

·目标：为用户态提供更友好的系统调用接口。

·实现：

在 ulib.c 中扩展接口，如提供更安全的字符串操作函数。

优化 printf 支持更多格式化类型。

·主要涉及文件：ulib.c、usertests.c。

2. 用户态信号机制

·目标：实现一个简单的信号机制（类似 POSIX 的 signal）。

·修改点：

扩展 proc 结构，增加信号处理函数的指针。

修改 trap.c，在 trap 返回时检查是否有信号需要处理。

·效果：支持用户态的异步事件处理。

·主要涉及文件：proc.c、trap.c、sysproc.c。

四、多核与并发优化

1. 自旋锁性能优化

·目标：为多核环境优化自旋锁，减少自旋时间。

·改进方向：

使用 ticket lock 替代简单的 spinlock。

引入指数退避策略减少总线争用。

·主要涉及文件：spinlock.c。

2. 条件变量与读写锁支持

·目标：在内核中实现条件变量和读写锁。

·实现：

条件变量：结合自旋锁和睡眠锁实现。

读写锁：允许多个读进程同时访问，写进程独占。

·主要涉及文件：sleeplock.c。

3. 支持用户态线程库

·目标：实现轻量级线程（LWP）。

·实现：

为用户态进程增加线程控制块（TCB）。

提供线程创建、销毁、锁等基础接口。

·主要涉及文件：sysproc.c、ulib.c。

五、内存管理优化

1. 支持动态内存分页

·目标：支持按需分配物理内存。

·实现思路：

在 `trap.c` 的缺页异常处理中调用 `kalloc` 分配新页。

修改 `uvmalloc` 支持按需分配。

·主要涉及文件：`vm.c`、`trap.c`。

2. 实现内核页缓存

·目标：为内核增加页缓存，提升磁盘读取性能。

·实现：

在内核中维护一个 LRU 缓存。

修改 `bread` 和 `bwrite`，优先从缓存中读取数据。

·主要涉及文件：`bio.c`。

六、扩展用户态功能

1. 实现更多用户态程序

·新增的用户态工具：

`top`：显示进程状态。

`df`：显示文件系统使用情况。

`grep`：实现简单的字符串匹配。

·主要涉及文件：`usertests.c`、`ulib.c`。

2. Shell 扩展

·目标：为用户提供更强大的交互能力。

·改进方向：

支持管道 (`|`) 和重定向 (`>`, `<`)。

支持基本的脚本功能（如 `for`、`if` 等）。

·主要涉及文件：`init.c`。

七、其他扩展方向

1. 网络支持

·目标：为 xv6 增加基础网络栈（如 TCP/IP 协议）。

·实现思路：

模拟一个虚拟网卡设备。

提供基本的 socket 接口。

·学习涉及文件：net.c。

2. 崩溃恢复与日志系统

·目标：增强文件系统的可靠性。

·实现：

实现一个简单的事务日志系统。

修改文件系统的元数据更新流程，支持崩溃恢复。

·主要涉及文件：fs.c、log.c。

二、 Our-xv6-riscv 的设计与实现

1、系统调用跟踪

在 xv6 中添加一个系统调用跟踪功能，该功能可帮助我们在以后的实验中调试程序。创建一个新的 trace 系统调用来控制跟踪。它应该有一个参数，一个整数“mask”，其位指定要跟踪的系统调用。例如，为了跟踪 fork 系统调用，程序调用 trace (1<<SYS_fork)，其中 SYS_fork 是 kernel/syscall.h 中的 syscall 编号。如果在掩码中设置了系统调用的编号，则必须修改 xv6 内核，以便在每个系统调用即将返回时输出一行。该行应包含进程 id、系统调用的名称和返回值；不需要输出系统调用参数。trace 系统调用应启用对调用它的进程及其随后派生的任何子进程的跟踪，但不应影响其他进程。

函数实现：

①在 proc 结构体中添加一个 trace_mask 字段，之后在创建子进程的 fork 函数中复制该字段到新进程：

```
// Per-process state
struct proc {
```



```

struct spinlock lock;

// p->lock must be held when using these:
enum procstate state;          // Process state
void *chan;                    // If non-zero, sleeping on chan
int killed;                    // If non-zero, have been killed
int xstate;                    // Exit status to be returned to parent's
wait
int pid;                       // Process ID
int priority;                  //***new

// wait_lock must be held when using this:
struct proc *parent;           // Parent process

// these are private to the process, so p->lock need not be held.
uint64 kstack;                 // Virtual address of kernel stack
uint64 sz;                     // Size of process memory (bytes)
pagetable_t pagetable;        // User page table
struct trapframe *trapframe;  // data page for trampoline.S
struct context context;       // swtch() here to run process
struct file *ofile[NOFILE];   // Open files
struct inode *cwd;            // Current directory
char name[16];                // Process name (debugging)
uint trace_mask;
};

```

②在 fork 函数中增加 np->trace_mask=p->trace_mask; 复制 mask。

③系统调用 sys_trace 的实现：在 sysproc.c 中添加函数 uint64

sys_trace(void)，该函数通过 argint 函数读取参数赋值给 mask 变量，然后与 trace_mask 字段进行位或即可。

```

uint64
sys_trace(void)
{
    uint mask;
    argint(0, (int*)&mask);
    if(mask<0)
        return -1;
    struct proc *p = myproc();
    p->trace_mask |= mask;
    return 0;
}

```

④修改 syscall 函数，当系统调用号和 trace_mask 匹配时输出相关信息：

```

161 void
162 syscall(void)
163 {
164     int num;
165     struct proc *p = myproc();
166
167     num = p->trapframe->a7;
168     if(num > 0 && num < NELEM(syscalls) && syscalls[num]) {
169         uint64 ret = syscalls[num]();
170         p->trapframe->a0 = ret;
171         if((1 << num) & p->trace_mask) {
172             printf("%d: syscall %s -> %d\n", p->pid, syscall_name[num], ret);
173         }
174     } else {
175         printf("%d %s: unknown sys call %d\n",
176             p->pid, p->name, num);
177         p->trapframe->a0 = -1;
178     }
179 }

```

⑤在 kernel/syscall.h 中定义系统调用号；在 kernel/syscall.c 的 syscalls 函数指针数组中添加对应的函数；

⑥编写应用工具：

```

#include "kernel/param.h"
#include "kernel/types.h"
#include "kernel/stat.h"
#include "user/user.h"

int
main(int argc, char *argv[ ])
{
    int i;
    char *nargv[MAXARG] ;

    if(argc < 3 || (argv[1][0] < '0' || argv[1][0] > '9')){
        fprintf(2, "Usage: %s mask command\n", argv[0]);
        exit(1);
    }

    if (trace(atoi(argv[1])) < 0) {
        fprintf(2, "%s: trace failed\n", argv[0]);
        exit(1)
    }

    for(i = 2; i < argc && i < MAXARG; i++){
        nargv[i-2] = argv[i];
    }
}

```

```
}  
exec(nargv[0], nargv);  
exit(0)  
}
```

⑥修改 Makefile,将\$U/_trace 添加到 Makefile 的 UPROGS 中；在 user/user.h 中添加系统调用函数的定义；在 user/usys.pl 中添加入口 entry("trace")；这样系统调用的用户空间的存根（stubs）便设置好了。

⑦Makefile 调用 perl 脚本 user/usys.pl，它生成 user/usys.S，即实际的系统调用存根，它使用 RISC-V ecall 指令转换到内核。

usys.S 中出现下图中.global trace 开始的 5 行代码：

```
103 .global uptime  
104 uptime:  
105  li a7, SYS_uptime  
106  ecall  
107  ret  
108 .global trace  
109 trace:  
110  li a7, SYS_trace  
111  ecall  
112  ret
```

在该汇编程序中，每个函数有五行，其中有三条指令，其功能是将系统调用号通过 li(load imm)存入 a7 寄存器，之后使用 ecall 指令进入内核态，最后返回。

⑧make qemu，然后测试

```
$ trace 32 grep hello README
trace pid: 3
3: syscall read -> 1023
3: syscall read -> 966
3: syscall read -> 70
3: syscall read -> 0
$ trace 2147483647 grep hello README
trace pid: 4
4: syscall trace -> 0
4: syscall exec -> 3
4: syscall open -> 3
4: syscall read -> 1023
4: syscall read -> 966
4: syscall read -> 70
4: syscall read -> 0
4: syscall close -> 0
$ grep hello README
$ trace 2 usertests forkforkfork
```

2、当前空闲内存/进程数

增加一个 `sysinfo` 系统调用，它收集有关运行系统的信息。

该系统调用有一个参数，即指向结构 `sysinfo` 的指针（参见 `kernel/sysinfo.h`）。内核为该结构的各个字段赋值：设置 `freemem` 字段为可用内存的字节数，设置 `nproc` 字段为状态是非 `UNUSED` 的进程数。

① `freemem()` 函数的实现：由 `kalloc` 和 `kfree` 两个函数可知，`kmem.freelist` 是一个保存了当前空闲内存块的链表，因此只需要统计这个链表的长度再乘以 `PGSIZE` 就可以得到空闲内存（注意在 `defs.h` 中声明 `freemem` 函数）。

```
// get free memory
uint64
freemem(void)
{
    uint64 counter = 0;
    struct run *r;
    acquire(&kmem.lock); // 上锁
    r = kmem.freelist;    // 空闲块链表
    while(r){             // 遍历空闲块链表，统计空闲块数
        r = r->next;
        ++counter;
    }
    release(&kmem.lock); // 释放自旋锁
    return counter * PGSIZE; // 返回空闲存储空间大小，单位字节（B）
}
```

②nproc()函数的实现：xv6 的进程结构体保存在 proc[NPROC]数组中。而 proc->state 字段保存了进程的当前状态，有 UNUSED、SLEEPING、RUNNABLE、RUNNING、ZOMBIE 五种状态。因此只需要遍历这个数组，统计 state 不是 UNUSED 状态的就行了：

```
// get number of proc
uint64
nproc(void)
{
    uint64 counter = 0;
    struct proc *p;
    // 遍历进程控制块，即 OS 课程中的 PCB
    for(p = proc; p < &proc[NPROC]; p++) {
        acquire(&p->lock);
        if(p->state != UNUSED) {
            ++counter;
        }
        release(&p->lock);
    }
    return counter;
}
```

③系统调用 sys_sysinfo 的实现：该函数主要通过 freemem 和 nproc 两个函数来统计空闲内存量和进程数。

```
// get sysinfo
uint64
sys_sysinfo(void)
{
    uint64 info; // user pointer
    struct sysinfo kinfo;
    struct proc *p = myproc();
    if(argaddr(0, &info) < 0){
        return -1;
    }
    kinfo.freemem = freemem();
    kinfo.nproc = nproc();
    if(copyout(p->pagetable, info, (char*)&kinfo, sizeof(kinfo)) < 0){
        return -1;
    }
    return 0;
}
```

④编写应用工具 sysinfo.c:

```

#include "kernel/param.h"
#include "kernel/types.h"
#include "kernel/sysinfo.h"
#include "user/user.h"

int
main(int argc, char *argv[])
{
    // param error
    if (argc != 1)
    {
        fprintf(2, "Usage: %s need not param\n", argv[0]);
        exit(1);
    }

    struct sysinfo info;
    sysinfo(&info);
    // print the sysinfo
    printf("free space: %d\nused process: %d\n", info.freemem,
info.nproc);
    exit(0);
}

```

⑤修改 Makefile,将\$U/_sysinfotest 添加到 Makefile 的 UPROGS 中。

⑥make qemu, 然后测试。

3、进程追踪

在 xv6 系统中, 不自带 Linux 下的 ps 命令, 不能直观查看各个进程的状态。为此, 需要新增一个系统调用 sys_cps(), 通过它查看进程, 将进程名, pid 和优先级打印出来, 为之后的调度算法实现做准备。

①首先在进程结构体 struct proc 中加入一个表示优先级的成员。(在 proc.h 中) 引入 priority 表示优先级, 在创建一个进程的时候, 可以默认将其优先级设定为 10。最终的 proc 如下:

```

// Per-process state
struct proc {
    uint sz;                // Size of process memory (bytes)
    pde_t* pgdir;          // Page table

```

```

char *kstack;           // Bottom of kernel stack for this
process
enum procstate state;   // Process state
int pid;                // Process ID
struct proc *parent;    // Parent process
struct trapframe *tf;   // Trap frame for current syscall
struct context *context; // swtch() here to run process
void *chan;             // If non-zero, sleeping on chan
int killed;             // If non-zero, have been killed
struct file *ofile[NOFILE]; // Open files
struct inode *cwd;      // Current directory
char name[16];          // Process name (debugging)
int priority;           // (0-20)
};

```

②实现的函数 `cps()` 功能即为遍历进程池，打印每个进程的信息，具体实现如下：

```

int
cps(void)
{
    struct proc *p;
    sti(); // Enable interrupts
    acquire(&ptable.lock);
    cprintf("name \t pid \t state \t \t priority \n");
    for(p = ptable.proc; p < &ptable.proc[NPROC]; p++)
    {
        if(p->state == SLEEPING)
            cprintf("%s \t %d \t SLEEPING \t %d\n", p->name, p->pid,
p->priority);
        else if(p->state == RUNNING)
            cprintf("%s \t %d \t RUNNING \t %d\n", p->name, p->pid,
p->priority);
        else if(p->state == RUNNABLE)
            cprintf("%s \t %d \t RUNNABLE \t %d\n", p->name, p->pid,
p->priority);
    }
    release(&ptable.lock);
    return 22;
}

```

最后，`cps()` 的返回值为 22。因为 `cps()` 最终需要变成一个系统调用，而每个系统调用函数的返回值都是 `int`，代表系统调用号。因此需要制定它的返回编号，在后续操作中，`sys_cps` 的系统调用号即为 22。

③在内核中注册该函数：包括系统调用编号、syscalls[]中的注册等。

4、自旋锁性能优化

多核环境中，自旋锁的性能优化可以显著减少总线争用，提高系统的并发性能。我们将实现 **ticket lock** 替代简单的 **spinlock**，并引入 **指数退避策略** 来减少无效自旋。

①工作原理：

通过维护两个计数器来保证公平性：

- **next_ticket**: 每个线程在进入临界区前获取一个“票号”。
- **current_ticket**: 指向当前正在使用锁的“票号”。

工作流程：

1. 每个线程在试图获取锁时，递增 **next_ticket** 并保存自己的票号。
2. 线程自旋，直到 **current_ticket** 等于自己的票号。
3. 线程释放锁时，递增 **current_ticket**。

②引入指数退避策略：

当一个线程发现锁不可用时：等待时间逐渐增加（如：2、4、8、16 微秒），避免线程频繁占用总线资源。

③具体实现：

在 **spinlock.h** 中定义 **ticket lock** 的必要字段：

```
// spinlock.h

struct spinlock {
    uint locked;           // Is the lock held? (for backward compatibility)
    char *name;           // Name of lock.
    struct cpu *cpu;      // The CPU holding the lock.

    // Ticket lock fields
    uint next_ticket;      // The next ticket to acquire.
    uint current_ticket;   // The current ticket being served.
};
```

初始化 **next_ticket** 和 **current_ticket**:

```
void
```



```
initlock(struct spinlock *lk, char *name)
{
    lk->locked = 0;
    lk->name = name;
    lk->cpu = 0;

    // Initialize ticket lock fields
    lk->next_ticket = 0;
    lk->current_ticket = 0;
}
```

修改 acquire、release、holding 函数来适应 ticket lock。

④创建一个测试程序，验证 ticket lock 的公平性和正确性：

在 user/ticketlocktest.c 中定义独立的 spinlock 结构和相关函数，用于模拟锁的行为：

```
#include "kernel/types.h"
#include "user/user.h"

#define NUM_THREADS 10

// 用户态自定义的 spinlock
struct spinlock {
    uint locked; // 0: unlocked, 1: locked
    char *name; // Lock name
};

// 初始化锁
void initlock(struct spinlock *lk, char *name) {
    lk->locked = 0;
    lk->name = name;
}

// 获取锁
void acquire(struct spinlock *lk) {
    while (__sync_lock_test_and_set(&lk->locked, 1)) {
        ; // 自旋等待锁释放
    }
}

// 释放锁
void release(struct spinlock *lk) {
    __sync_lock_release(&lk->locked);
}
```

将上述 `spinlock` 实现与测试程序集成，修改后的代码如下：

```
// user/ticketlocktest.c
#include "kernel/types.h"
#include "user/user.h"

#define NUM_THREADS 10

// 自定义的 spinlock
struct spinlock lock;
volatile int counter = 0;

void thread_func(void *arg) {
    for (int i = 0; i < 1000; i++) {
        acquire(&lock); // 获取锁
        counter++;       // 临界区
        release(&lock);  // 释放锁
    }
    exit(0);
}

int main() {
    initlock(&lock, "test_lock"); // 初始化锁

    for (int i = 0; i < NUM_THREADS; i++) {
        if (fork() == 0) { // 创建多个线程
            thread_func(0);
        }
    }

    for (int i = 0; i < NUM_THREADS; i++) {
        wait(0); // 等待子进程结束
    }

    // 验证最终计数器的值
    printf("Final counter value: %d (expected: %d)\n", counter,
NUM_THREADS * 1000);

    exit(0);
}
```

⑤在 Makefile 的 UPROGS 部分添加 `ticketlocktest`:

```
UPROGS=\
```

```
_cat\
_echo\
...\
_ticketlocktest\
```

5、多级反馈队列调度器（MLFQ）

多级反馈队列调度器（MLFQ）是一个复杂的调度器，可以根据进程的行为动态调整其优先级，从而提高系统的交互性和公平性。

以下是我们实现多级反馈队列调度器的步骤：

① 修改 proc.h 添加必要的字段

在 struct proc 中添加以下字段，用于支持 MLFQ 调度：

```
// proc.h
#define NQUEUE 5 // Number of priority levels in MLFQ

struct proc {
    ...
    int queue; // Current queue level (0 is highest priority)
    int ticks[NQUEUE]; // Time slices consumed in each queue
    int time_slice; // Remaining time slice for the current level
};
```

② 定义每个队列的时间片大小

在 param.h 和 proc.c 中定义时间片大小数组 queue_time_slices[]：

```
#define NQUEUE 5 // 多级反馈队列数量

int queue_time_slices[NQUEUE] = {1, 2, 4, 8, 16}; // 每个队列的时间片大小
```

③ 修改 proc.c 初始化 MLFQ 队列

在 proc.c 中添加一个全局 MLFQ 数据结构，用于管理各个队列的进程：

```
// proc.c
#include "param.h"

struct {
    struct proc *queue[NQUEUE][NPROC]; // Multi-level queues
    int head[NQUEUE]; // Head pointers for each queue
    int tail[NQUEUE]; // Tail pointers for each queue
};
```

```
} mlfq;
```

在 procinit() 中初始化队列结构:

```
void
procinit(void)
{
    ...
    for (int i = 0; i < NQUEUE; i++) {
        mlfq.head[i] = 0;
        mlfq.tail[i] = 0;
    }
}
```

④ 添加队列操作函数

MLFQ 的入队和出队操作:

```
// Add a process to the specified queue
void enqueue(int queue, struct proc *p) {
    if (mlfq.tail[queue] < NPROC) {
        mlfq.queue[queue][mlfq.tail[queue]++] = p;
    }
}

// Remove and return the next process in the specified queue
struct proc* dequeue(int queue) {
    if (mlfq.head[queue] < mlfq.tail[queue]) {
        return mlfq.queue[queue][mlfq.head[queue]++];
    }
    return 0;
}
```

⑤ 修改调度器逻辑

在 scheduler() 中实现 MLFQ 调度逻辑:

```
void
scheduler(void)
{
    struct proc *p;
    struct cpu *c = mycpu();

    c->proc = 0;

    for(;;){
```

```

// Enable interrupts on this CPU.
intr_on();

// 遍历优先级队列，从最高优先级（0）开始寻找 RUNNABLE 的进程
for(int level = 0; level < NQUEUE; level++) { // 遍历每个优先级队
列
    acquire(&p->lock); // 假设 `ptable.lock` 是全局锁保护进程队列

    for(p = proc; p < &proc[NPROC]; p++) { // 遍历进程表
        if(p->state == RUNNABLE && p->priority == level) { // 找到对
应队列的 RUNNABLE 进程
            // 切换到选中的进程
            p->state = RUNNING;
            c->proc = p;

            // 切换到进程上下文并执行
            swtch(&c->context, &p->context);

            // 当进程切回到调度器时，可能是因为时间片耗尽、进程退出或被阻塞
            c->proc = 0;

            // 如果时间片耗尽，降低优先级
            if(p->ticks[level] >= p->time_slice) {
                p->ticks[level] = 0; // 重置当前队列的时间片计数
                if(p->priority < NQUEUE - 1) { // 如果当前不在最低优先级队
列
                    p->priority++; // 降低优先级
                    p->time_slice = queue_time_slices[p->priority]; // 更新
时间片
                }
            }
        }
    }
    release(&p->lock);
}
}
}

```

这个函数需要根据进程的优先级队列来调度，并在每次调度时更新进程的时间片和优先级。

主要逻辑：

- 多级反馈队列的遍历：

使用 `for (int level = 0; level < NQUEUE; level++)` 遍历从高到低的优先级队列。

优先级队列的顺序确保高优先级的进程被优先调度。

- 进程的时间片管理：

每个优先级队列有不同的时间片大小（由 `queue_time_slices[level]` 定义）。

当进程在当前队列的时间片耗尽时，将其优先级降低到下一队列。

- 锁的使用：

使用全局 `ptable.lock` 来保护进程表的访问，避免多核并发调度时的竞争问题。

- 进程的优先级更新：

如果时间片用尽且进程不是最低优先级队列，则将其优先级降低一级。

- CPU 运行的进程上下文切换：

调用 `swtch(&c->context, &p->context)` 切换到选定的进程上下文。

当进程切换回来时，可能是因为时间片耗尽、被阻塞或主动退出。

⑥ 在时钟中断中实现优先级调整

在 `trap.c` 的时钟中断处理函数中，添加逻辑以处理时间片消耗和队列调整：

```
void
clockintr()
{
    if (cpuid() == 0) {
        acquire(&tickslock);
        ticks++;
        wakeup(&ticks);
        release(&tickslock);
    }

    struct proc *p = myproc();
    if (p && p->state == RUNNING) {
        if (--p->time_slice <= 0) {
            // Time slice expired, move process to a lower queue
            if (p->queue < NQUEUE - 1) {
                p->queue++;
            }
            p->time_slice = (1 << p->queue); // Exponential time slice
            yield(); // Preempt the current process
        }
    }
}
```

```
}
}
```

⑦ 实现队列提升

通过定时器实现老化机制，定期提升低优先级队列中的进程：

```
void
boost_queues()
{
    for (int i = 1; i < NQUEUE; i++) { // Skip the highest-priority queue
        while (mlfq.head[i] < mlfq.tail[i]) {
            struct proc *p = dequeue(i);
            p->queue = 0; // Move to the highest-priority queue
            p->time_slice = (1 << p->queue);
            enqueue(0, p);
        }
    }
}
```

在定时器中定期调用：

```
void
clockintr()
{
    ...
    if (ticks % 1000 == 0) { // Every 1000 ticks
        boost_queues();
    }
}
```

⑧ 初始化进程的优先级

在 allocproc() 中为新进程初始化优先级和时间片：

```
struct proc*
allocproc(void)
{
    ...
    p->queue = 0; // Start in the highest-priority queue
    p->time_slice = 1; // Initial time slice
    memset(p->ticks, 0, sizeof(p->ticks)); // Clear tick counters
    ...
}
```

⑨ 测试与调试

创建一个新的测试程序文件：user/mlfqtest.c。注意，我们并没有在 usertest.c 文件中直接实现 mlfqtest 函数，而是单独创建一个文件。这个测试程序创建多个子进程，并让它们打印信息以模拟不同的工作负载，测试多级反馈队列的行为。

```
#include "types.h"
#include "stat.h"
#include "user.h"

#define N 5 // Number of processes to test priority scheduling

void spin(int ticks) {
    uint start = uptime();
    while (uptime() - start < ticks);
}

int main(void) {
    printf("MLFQ Test: Creating %d processes\n", N);
    for (int i = 0; i < N; i++) {
        int pid = fork();
        if (pid < 0) {
            printf("fork failed\n");
            exit(1);
        }
        if (pid == 0) {
            printf("Child %d started\n", i);
            for (int j = 0; j < 5; j++) {
                printf("Child %d: Tick %d\n", i, j);
                spin(10); // Simulate workload
            }
            exit(0);
        }
    }
    for (int i = 0; i < N; i++) {
        wait(0);
    }
    printf("MLFQ Test: Finished\n");
    exit(0);
}
```

在 Makefile 中添加 mlfqtest 在 UPROGS 变量中添加 mlfqtest:

```
UPROGS=\
```



```
$U/_cat\  
$U/_echo\  
$U/_forktest\  
$U/_grep\  
$U/_init\  
$U/_kill\  
$U/_ln\  
$U/_ls\  
$U/_mkdir\  
$U/_mlfqtest\  
$U/_rm\  
$U/_sh\  
$U/_stressfs\  
$U/_usertests\  
$U/_grind\  
$U/_wc\  
$U/_zombie\  

```

重新编译和创建磁盘映像 运行以下命令重新编译代码并生成磁盘映像：

```
make clean  
make fs.img
```

运行 QEMU：

```
make qemu
```

进入 QEMU shell 后运行测试程序：

```
sh  
mlfqtest
```

观察程序输出是否符合预期。

在程序运行过程中，可以通过以下方式进一步验证 MLFQ 的调度行为：

输出进程信息 在 `proc.c` 的调度函数中（`scheduler`），添加打印当前正在运行的进程及其队列信息：`printf("Running process %d in queue %d\n", p->pid, p->queue);`

监控队列迁移 确保进程按照预期在不同的优先级队列间迁移，例如高优先级耗尽时间片后降级，低优先级长时间等待后升级；

检查时间片 修改 `ticks` 或 `time_slice` 的设置，确认进程的调度是否符合预期。

三、 总结

1、页表更新与 TLB 刷新：

- 在每次更新页表后，必须及时刷新 TLB，否则会引发难以排查的内存访问错误。
- 这在实现 `exec` 和 `fork` 等涉及地址空间切换的系统调用时尤为重要。

2、系统调用的规范性设计：

- 在设计 `exec` 调用时，严格遵循 POSIX 标准，确保栈的初始化包括辅助变量、环境变量及命令行参数的规范布局。

3、多核并发调试：

- 多核环境下调试难度较大，通过插桩和日志打印，可以有效追踪进程状态及锁的使用情况，避免死锁等问题。

4、 用户态内存管理：

- 在实现共享内存等功能时，必须对用户态传入的地址进行严格检查，确保内核态操作不会因非法地址导致崩溃。
- 注意内存大小问题，在我们的工作过程中，对于内存方面的改善出现 `batchsize` 无法整除内存的情况导致的报错。

5、 硬件相关操作的谨慎性：

- 手写汇编代码时，应严格检查关键寄存器（如 `sp`、`tp` 等）的初始化和赋值情况，否则可能导致意外的运行错误。

四、未来计划

1、进程抢占支持

进程调度是基于时间片的非抢占式调度，也就是当前运行的进程只有在时间片耗尽或者主动让出 CPU 时，才会切换到其他进程。而抢占式调度的目的是允许高优先级的进程可以打断正在运行的低优先级进程，以实现更高的系统响应能力。

抢占式调度需要结合 时钟中断 和 调度器逻辑，在每次时钟中断时检查当前进程是否需要被抢占。

① 修改时钟中断处理逻辑

时钟中断（clockintr）是实现进程抢占的关键。它每隔一段时间（通常为 10ms 或 100ms）触发中断，检查是否需要切换当前进程。

- 在 trap.c 中的 usertrap() 和 kerneltrap() 函数内处理时钟中断。
- 检查当前运行的进程是否已经耗尽了时间片或者是否有更高优先级的进程需要运行。

```
void
usertrap(void)
{
    // ... 省略现有代码 ...

    // 如果是时钟中断
    if(which_dev == 2) {
        struct proc *p = myproc();

        // 增加当前进程的运行时间计数
        p->ticks[p->priority]++;

        // 如果时间片耗尽，标记需要抢占
        if(p->ticks[p->priority] >= p->time_slice) {
            p->ticks[p->priority] = 0; // 重置时间片计数
            p->state = RUNNABLE;       // 将当前进程置为可运行
            yield();                    // 让出 CPU，进入调度器
        }
    }

    // ... 省略现有代码 ...
}
```

```
}

```

- `yield()` 会强制当前进程进入调度器，切换到其他进程。
- 时间片耗尽后，通过 `yield()` 将 CPU 让给其他进程。

② 调整调度器逻辑

调度器需要能够及时响应被抢占的进程，重新选择适合的高优先级进程运行。

在 `scheduler()` 中，确保总是优先选择高优先级进程：

```
void
scheduler(void)
{
    struct proc *p;
    struct cpu *c = mycpu();

    c->proc = 0;

    for(;;){
        intr_on();

        // 遍历优先级队列，从高到低寻找可运行的进程
        for(int level = 0; level < NQUEUE; level++) {
            acquire(&p->lock);

            for(p = proc; p < &proc[NPROC]; p++) {
                if(p->state == RUNNABLE && p->priority == level) {
                    // 切换到选中的进程
                    p->state = RUNNING;
                    c->proc = p;

                    swtch(&c->context, &p->context); // 切换到进程

                    // 调度器切回后，进程已被抢占或主动退出
                    c->proc = 0;

                    release(&p->lock);
                    break;
                }
            }
            release(&p->lock);
        }
    }
}
```

}

③ 优先级动态调整

添加优先级提升机制：

- 当某个低优先级进程长时间未被调度时，提升其优先级。
- 可在 `trap.c` 的时钟中断中定期扫描所有进程，根据 `ticks` 或等待时间动态调整优先级。

```
void
adjust_priorities(void)
{
    struct proc *p;
    acquire(&p->lock);
    for(p = proc; p < &proc[NPROC]; p++) {
        if(p->state == RUNNABLE && p->priority > 0) {
            p->priority--; // 提升优先级
        }
    }
    release(&p->lock);
}
```

在时钟中断中定期调用 `adjust_priorities()`，实现优先级动态调整。

④ 原理解析：

- 时钟中断驱动调度：

每次时钟中断都会检查当前进程的时间片。

如果时间片耗尽或有更高优先级的进程等待，就触发抢占。

- 进程状态管理：

抢占式调度需要动态调整进程的状态：

时间片耗尽的进程状态从 `RUNNING` 切换到 `RUNNABLE`。

调度器会选择其他进程运行。

- 动态优先级调整：

避免低优先级进程“饥饿”。

定期提升等待时间较长的进程优先级，使其有机会被调度。

- 抢占实现核心：

结合时钟中断和调度器逻辑，确保高优先级进程总是优先获得 CPU。

使用 `yield()` 强制当前进程放弃 CPU。

2、进程的内存隔离和共享

在操作系统中，内存隔离和内存共享是两个重要的特性：

内存隔离：每个进程都应该有独立的虚拟地址空间，保证进程之间互不干扰。这是操作系统保护进程免受其他进程恶意行为的重要措施。

内存共享：有些情况下，多个进程需要协作，彼此访问共享的内存区域来进行高效通信。比如，使用共享内存可以减少数据在进程间传递的开销。

其中**内存隔离**在 `xv6` 中已经通过页表机制实现。每个进程都有一个独立的页表，确保虚拟地址到物理地址的映射互不干扰。

主要代码实现：

- 为每个进程创建独立的页表：

在 `proc.c` 中的 `proc_pagetable()` 函数为每个进程创建页表。

映射进程的用户内存（代码段、数据段、堆和栈）以及必要的内核区域（如 `TRAMPOLINE` 和 `TRAPFRAME`）。

- 销毁进程的页表：

在 `proc.c` 中的 `proc_freepagetable()` 函数，销毁进程页表和其映射的内存。

- 访问控制：

通过页表权限位（如 `PTE_R`、`PTE_W`、`PTE_U`）控制内存访问权限，防止越权访问。

共享内存可以让多个进程在某个虚拟地址范围内访问相同的物理内存页，从而实现进程间高效的数据共享。

设计方案：

- 共享内存段：

为共享内存段定义一个全局的共享内存结构 `shared_mem`，用来管理物理页的分配和映射。

每个共享内存段可以被多个进程映射。

- 系统调用接口：

定义两个新的系统调用：

`shm_alloc(int key, int size)`：分配或获取共享内存段。

`shm_map(int key, uint64 addr)`: 将共享内存段映射到调用进程的虚拟地址空间。

• 数据结构:

```
#define NSHARED 16    // 最大支持的共享内存段数量
#define MAXSHARED_PAGES 64 // 每个共享内存段最大页数

struct shared_mem {
    int key;           // 共享内存段的唯一标识
    int ref_count;     // 当前使用该共享内存段的进程数
    void *pages[MAXSHARED_PAGES]; // 物理页数组
    int size;          // 共享内存段的大小（页数）
};

struct shared_mem shared_mem_segments[NSHARED];
```

①初始化共享内存段管理结构

在系统启动时初始化 `shared_mem_segments` 数组。

```
void
shm_init(void)
{
    for (int i = 0; i < NSHARED; i++) {
        shared_mem_segments[i].key = -1; // -1 表示未使用
        shared_mem_segments[i].ref_count = 0;
        shared_mem_segments[i].size = 0;
        memset(shared_mem_segments[i].pages, 0,
sizeof(shared_mem_segments[i].pages));
    }
}
```

②shm_alloc 分配或获取共享内存段:

检查 `key` 是否已经存在:

- 如果存在, 则返回该共享内存段的标识。
- 如果不存在, 则分配新的共享内存段, 并分配对应的物理页。

```
• uint64
• shm_alloc(int key, int size)
• {
•     acquire(&shm_lock);
•
•     // 检查是否已有相同 key 的共享内存段
•     for (int i = 0; i < NSHARED; i++) {
```

```

    if (shared_mem_segments[i].key == key) {
        shared_mem_segments[i].ref_count++;
        release(&shm_lock);
        return key; // 返回已有的共享内存段
    }
}

// 分配新的共享内存段
for (int i = 0; i < NSHARED; i++) {
    if (shared_mem_segments[i].key == -1) {
        shared_mem_segments[i].key = key;
        shared_mem_segments[i].ref_count = 1;
        shared_mem_segments[i].size = size;

        // 分配物理页
        for (int j = 0; j < size; j++) {
            shared_mem_segments[i].pages[j] = kalloc();
            if (!shared_mem_segments[i].pages[j]) {
                release(&shm_lock);
                return -1; // 分配失败
            }
            memset(shared_mem_segments[i].pages[j], 0,
PGSIZE);
        }

        release(&shm_lock);
        return key;
    }
}

release(&shm_lock);
return -1; // 无法分配新的共享内存段
}

```

③shm_map 将共享内存段映射到进程的虚拟地址空间

```

uint64
shm_map(int key, uint64 addr)
{
    acquire(&shm_lock);

    // 查找共享内存段
    struct shared_mem *shm = 0;
    for (int i = 0; i < NSHARED; i++) {

```



```

        if (shared_mem_segments[i].key == key) {
            shm = &shared_mem_segments[i];
            break;
        }
    }

    if (!shm) {
        release(&shm_lock);
        return -1; // key 不存在
    }

    struct proc *p = myproc();
    for (int i = 0; i < shm->size; i++) {
        if (mappages(p->pagetable, addr + i * PGSIZE, PGSIZE,
            (uint64)shm->pages[i], PTE_W | PTE_R | PTE_U) < 0) {
            release(&shm_lock);
            return -1; // 映射失败
        }
    }

    release(&shm_lock);
    return addr; // 映射成功
}

```

④释放共享内存：当一个进程调用 `exit` 或 `shm_unmap` 时，减少共享内存段的引用计数。如果引用计数为 0，则释放共享内存段的物理页。

```

void
shm_free(int key)
{
    acquire(&shm_lock);

    for (int i = 0; i < NSHARED; i++) {
        if (shared_mem_segments[i].key == key) {
            shared_mem_segments[i].ref_count--;
            if (shared_mem_segments[i].ref_count == 0) {
                // 释放物理页
                for (int j = 0; j < shared_mem_segments[i].size; j++) {
                    kfree(shared_mem_segments[i].pages[j]);
                }
                shared_mem_segments[i].key = -1; // 重置
            }
            break;
        }
    }
}

```

```

    }

    release(&shm_lock);
}

```

⑤编写测试程序

```

// 测试共享内存的分配和映射。
#include "kernel/types.h"
#include "kernel/stat.h"
#include "user/user.h"

#define SHM_KEY 1234
#define SHM_SIZE 4

int
main(void)
{
    int key = SHM_KEY;
    int size = SHM_SIZE;
    void *addr = (void *)0x40000000;

    if (shm_alloc(key, size) < 0) {
        printf("shm_alloc failed\n");
        exit(1);
    }

    if (shm_map(key, addr) < 0) {
        printf("shm_map failed\n");
        exit(1);
    }

    printf("Shared memory mapped at %p\n", addr);
    char *p = (char *)addr;
    p[0] = 'A'; // 写入共享内存
    printf("Written to shared memory: %c\n", p[0]);

    exit(0);
}

```

⑥这里列举添加系统调用的具体步骤，往后函数不在列举：

- 在 kernel/syscall.h 中，为新的系统调用定义系统调用号：

```
#define SYS_shm_alloc 23
```

```
#define SYS_shm_map 24
```

• 在 kernel/syscall.c 中，将新系统调用与它们对应的处理函数绑定，在 syscalls[] 数组中添加以下条目：

```
extern uint64 sys_shm_alloc(void);
extern uint64 sys_shm_map(void);

static uint64 (*syscalls[])(void) = {
    // ... 已有的系统调用
    [SYS_shm_alloc] sys_shm_alloc,
    [SYS_shm_map] sys_shm_map,
};
```

• 在 kernel/sysproc.c 中实现 sys_shm_alloc 和 sys_shm_map 的处理逻辑：

```
extern uint64 shm_alloc(int key, int size);
extern uint64 shm_map(int key, uint64 addr);

// `shm_alloc` 系统调用的处理函数
uint64
sys_shm_alloc(void)
{
    int key, size;

    // 从系统调用参数中获取 key 和 size
    if(argint(0, &key) < 0 || argint(1, &size) < 0)
        return -1;

    return shm_alloc(key, size);
}

// `shm_map` 系统调用的处理函数
uint64
sys_shm_map(void)
{
    int key;
    uint64 addr;

    // 从系统调用参数中获取 key 和 addr
    if(argint(0, &key) < 0 || argaddr(1, &addr) < 0)
        return -1;

    return shm_map(key, addr);
}
```

```
}
```

- 在 kernel 目录中新增一个文件 shm.c，并实现共享内存的核心功能：
- 在 defs.h 中定义 shm.c 的函数：

```
// shm.c
void          shm_init(void);
uint64        shm_alloc(int, int);
uint64        shm_map(int, uint64);
```

- 在 kernel/main.c 的 main() 函数中，调用 shm_init()：

```
int
main(void)
{
    // 其他初始化逻辑...
    shm_init(); // 初始化共享内存管理
    scheduler();
}
```

- 在 user/usys.pl 文件中添加用户程序的系统调用接口：

```
entry("shm_alloc");
entry("shm_map");
```

运行 make clean && make，重新生成 user/usys.S 文件。

- 在 user/user.h 文件中，声明系统调用：

```
int shm_alloc(int key, int size);
int shm_map(int key, void *addr);
```

- 在 user 目录中创建一个测试程序 shmtest.c，测试共享内存的分配和映射。

3、 增加文件权限支持

- ①在文件系统的头文件中增加权限字段：

修改 struct dinode：

```
// On-disk inode structure
struct dinode {
    short type;           // File type
    short major;          // Major device number (T_DEVICE only)
    short minor;          // Minor device number (T_DEVICE only)
    short nlink;          // Number of links to inode in file system
    uint size;            // Size of file (bytes)
    uint addrs[NDIRECT+1]; // Data block addresses
    uint mode;            // File permissions (e.g., rwx bits)
};
```

②在 fs.h 中添加以下宏定义，表示文件权限::

```
#define MODE_R  0x4  // Read permission
#define MODE_W  0x2  // Write permission
#define MODE_X  0x1  // Execute permission
```

③修改 stat.h 文件:

在 kernel/stat.h 中添加文件权限字段:

```
struct stat {
    int dev;      // File system's disk device
    uint ino;     // Inode number
    short type;   // Type of file
    short nlink;  // Number of links to file
    uint size;    // Size of file in bytes
    uint mode;    // File permissions (rwx bits)
};
```

④在文件系统操作中添加对权限的支持: 更新 iupdate 和 ilock。将权限字段写入磁盘或从磁盘读取。

```
void
iupdate(struct inode *ip)
{
    struct buf *bp;
    struct dinode *dip;

    bp = bread(ip->dev, IBLOCK(ip->inum, sb));
    dip = (struct dinode*)bp->data + ip->inum%IPB;
    dip->type = ip->type;
    dip->major = ip->major;
    dip->minor = ip->minor;
    dip->nlink = ip->nlink;
    dip->size = ip->size;
    dip->mode = ip->mode; // 更新权限信息
    memmove(dip->addrs, ip->addrs, sizeof(ip->addrs));
    log_write(bp);
    brelse(bp);
}

void
ilock(struct inode *ip)
{
    struct buf *bp;
```

```

struct dinode *dip;

if(ip == 0 || ip->ref < 1)
    panic("ilock");

acquiresleep(&ip->lock);

if(ip->valid == 0){
    bp = bread(ip->dev, IBLOCK(ip->inum, sb));
    dip = (struct dinode*)bp->data + ip->inum%IPB;
    ip->type = dip->type;
    ip->major = dip->major;
    ip->minor = dip->minor;
    ip->nlink = dip->nlink;
    ip->size = dip->size;
    ip->mode = dip->mode; // 从磁盘读取权限信息
    memmove(ip->addrs, dip->addrs, sizeof(ip->addrs));
    brelse(bp);
    ip->valid = 1;
    if(ip->type == 0)
        panic("ilock: no type");
}
}

```

⑤在 sysfile.c 中添加 check_permission 函数，用于验证用户对文件的权限：

```

static int
check_permission(struct inode *ip, int perm)
{
    if((ip->mode & perm) != perm) {
        return -1; // 没有所需权限
    }
    return 0; // 权限验证通过
}

```

⑥修改文件操作系统调用：文件操作进行权限验证，例如在 sys_read 和 sys_write 中

```

uint64
sys_read(void)
{
    struct file *f;
    int n;

```

```
uint64 p;

argaddr(1, &p);
argint(2, &n);
if(argfd(0, 0, &f) < 0)
    return -1;

ilock(f->ip);
if(check_permission(f->ip, MODE_R) < 0) { // 检查读权限
    iunlock(f->ip);
    return -1;
}
iunlock(f->ip);

return fileread(f, p, n);
}

uint64
sys_write(void)
{
    struct file *f;
    int n;
    uint64 p;

    argaddr(1, &p);
    argint(2, &n);
    if(argfd(0, 0, &f) < 0)
        return -1;

    ilock(f->ip);
    if(check_permission(f->ip, MODE_W) < 0) { // 检查写权限
        iunlock(f->ip);
        return -1;
    }
    iunlock(f->ip);

    return filewrite(f, p, n);
}
```

⑦加 chmod 系统调用

⑧编写测试程序:

```
#include "kernel/types.h"
#include "kernel/stat.h"
```

```
#include "user/user.h"

int
main(int argc, char *argv[])
{
    if(argc != 3){
        printf("Usage: chmod <path> <mode>\n");
        exit(1);
    }

    const char *path = argv[1];
    int mode = atoi(argv[2]);

    if(chmod(path, mode) < 0) {
        printf("chmod failed\n");
        exit(1);
    }

    printf("chmod success\n");
    exit(0);
}
```

5、条件变量与读写锁支持

- 目标：在内核中实现条件变量和读写锁。
- 实现：
 - 条件变量：结合自旋锁和睡眠锁实现。
 - 读写锁：允许多个读进程同时访问，写进程独占。
- 主要涉及文件：sleeplock.c。

6、支持用户态线程库

- 目标：实现轻量级线程（LWP）。
- 实现：
 - 为用户态进程增加线程控制块（TCB）。
 - 提供线程创建、销毁、锁等基础接口。
- 主要涉及文件：sysproc.c、ulib.c。

7、支持动态内存分页

- 目标：支持按需分配物理内存。
- 实现思路：
 - 在 `trap.c` 的缺页异常处理中调用 `kalloc` 分配新页。
 - 修改 `uvmmalloc` 支持按需分配。
- 主要涉及文件：`vm.c`、`trap.c`。

8、实现内核页缓存

- 目标：为内核增加页缓存，提升磁盘读取性能。
- 实现：
 - 在内核中维护一个 LRU 缓存。
 - 修改 `bread` 和 `bwrite`，优先从缓存中读取数据。
- 主要涉及文件：`bio.c`。

9、Shell 扩展

- 目标：为用户提供更强大的交互能力。
- 改进方向：
 - 支持管道 (`|`) 和重定向 (`>`, `<`)。
 - 支持基本的脚本功能（如 `for`、`if` 等）。
- 主要涉及文件：`init.c`。

10. 网络支持

- 目标：为 `xv6` 增加基础网络栈（如 `TCP/IP` 协议）。
- 实现思路：
 - 模拟一个虚拟网卡设备。
 - 提供基本的 `socket` 接口。
- 学习涉及文件：`net.c`。

12. 崩溃恢复与日志系统

- 目标：增强文件系统的可靠性。

·实现:

实现一个简单的事务日志系统。

修改文件系统的元数据更新流程，支持崩溃恢复。

·主要涉及文件：fs.c、log.c。