



**CSCC 全国大学生
计算机系统能力大赛**



杭州电子科技大学
HANGZHOU DIANZI UNIVERSITY

FtidesOS

——基于 Xv6 内核改进实现



队伍名称：操作系统能不能简单点队

赛题方向：OS原理赛道——小型内核实现

ID: T202410336994293

目录

1	项目概述	3
2	系统调用实现	3
2.1	添加系统调用 getcpuid	3
2.2	添加系统调用 getSysCount	4
2.3	添加系统调用 countproc	7
2.4	添加系统调用 sigalarm 和 sigreturn	9
2.5	系统调用 sys_symlink 实现	13
2.6	系统调用 sys_open 的实现	14
3	进程调度	15
3.1	Lottery Scheduling (彩票调度)	16
3.1.1	概述	16
3.1.2	数据结构	17
3.1.3	彩票调度算法实现	18
3.1.4	集成 xv6 核心调度算法中	21
3.1.5	测试	24
3.2	MLFQ (多级反馈队列)	24
3.2.1	数据结构扩充—优先级队列	25
3.2.2	核心算法实现	25
4	文件系统板块	29
4.1	概述	29
4.2	混合索引文件优化	30
4.2.1	数据结构	31
4.3	算法思路与实现	32
4.4	文件共享优化	35
4.4.1	相应数据结构	36
4.4.2	算法和实现思路	36
5	内存管理板块	39
5.1	懒分配	40
5.1.1	数据结构	41
5.1.2	算法思路与实现	41
6	内核测试文档链接	44
7	总结和未来计划	45

1 项目概述

本项目是基于本校操作系统原理课程设计到的相关知识进行实践应用于 xv6 的操作系统的优化升级, 涉及系统调用添加、内核板块的优化升级 (进程调度算法、文件系统管理等), 在有限的时间内尽可能提升并扩展 xv6 操作系统的性能和应用场景! 下面将逐一介绍 xv6 操作系统优化升级的各个方面

2 系统调用实现

2.1 添加系统调用 getcpuid

1. 该系统调用的功能: 获取当前执行进程所在的 CPU 核心的 ID。在多核处理器系统中, 每个 CPU 核心都有一个唯一的 ID, 通过这个 ID, 可以识别当前执行的代码是在哪一个 CPU 核心上运行的。
2. `getcpuid` 的作用和实现的意义:
 - **多核处理器支持:** 实现 `getcpuid` 系统调用是支持多核处理器系统的第一步。通过获取 CPU ID, 操作系统能够更好地管理和调度进程, 提高系统性能和资源利用率。
 - **系统调用机制入门:** `getcpuid` 系统调用的实现相对简单, 是学习和掌握系统调用机制的理想起点。通过实现这个系统调用, 团队成员可以熟悉操作系统内核与用户空间之间的交互, 理解系统调用的基本流程。
 - **后续工作的基础:** `getcpuid` 系统调用为后续更加复杂的系统调用实现打下了基础。理解并实现这个系统调用后, 本团队在基础上实现更多功能, 如进程管理、内存管理和文件系统操作等。

下面是关于添加系统调用 `getcpuid` 整个过程

1. 首先, 在 `kernel/syscall.h` 文件中定义系统调用号 `SYS_getcpuid` 为 22:

```
1 // System call numbers
2 ...
3 #define SYS_getcpuid 22
```

2. 然后, 修改 `user/user.h`, 让应用程序能调用用户态入口函数 `getcpuid()`:

```
1 // system calls
2 ...
3 int getcpuid(void);
```

3. 接着, 在 `user/usys.S` 中定义用户态接口:

```
1 .global getcpuid
2 getcpuid:
3     li a7, SYS_getcpuid
4     ecall
5     ret
```

4. 在 `kernel/syscall.c` 文件中, 添加 `sys_getcpuid` 系统调用, 并将其添加到 `syscalls` 数组中:

```
1     extern uint64 sys_getcpuid(void);
2     // An array mapping syscall numbers from syscall.h
3     // to the function that handles the system call.
4     static uint64 (*syscalls[])(void) = {
5         ...
6         [SYS_getcpuid] sys_getcpuid,
7     };
```

5. 在 `kernel/sysproc.c` 文件中, 我添加了系统调用的实现

- 添加 `sys_getcpuid()`
- 返回 `getcpuid` 函数返回的值

```
1     uint64
2     ys_getcpuid()
3     {
4         return getcpuid();
5     }
```

6. 关于 `gecpuid` 函数在 `kernel/proc.c` 文件中实现:

```
1     int getcpuid()
2     {
3         return cpuid();
4     }
```

7. 在 `kernel/defs.h`

```
1     // proc.c
2     ...
3     int getcpuid(void);
```

可以点击[这里](#)跳转到该系统调用的测试文档查看详细的测试过程

2.2 添加系统调用 `getSysCount`

关于 `getSyscount` 的实现:

- **syntax:** `syscount <mask> command [args]`
- **mask** 是一个 32 位整数, 它指定了要计数的系统调用
- **command x** 是具体执行的命令
- **args** 是命令的参数

- 返回值: 系统调用返回掩码指定的系统调用在执行命令时被调用的次数
- 如果掩码无效, 系统调用将返回-1

下面是具体的逻辑实现过程

1. 首先, 我们在 **kernel/proc.h** 的 **proc** 结构中添加了一个数组, 以跟踪每个系统调用的调用次数

```

1      struct proc {
2          ...
3          // system call count
4          int syscallCount[32];
5          ...
6      };

```

2. 然后, 在 **kernel/sysproc.c** 文件中的每个 **sys_...** 函数中, 我们增加 **syscallCount** 数组中相应系统调用的计数。
3. 我们在 **kernel/proc.c** 文件的 **initproc** 函数和 **freeproc** 函数中对 **syscallCount** 数组添加清零的工作

```

1      void
2      initproc(void)
3      {
4          ...
5          for(int i=0;i<32;i++){
6              p->syscallCount[i]=0;
7          }
8          ...
9      }
10     ...
11     static void freeproc(struct proc *p) {
12         for(int i=0;i<32;i++){
13             p->syscallCount[i]=0;
14         }
15     }

```

4. 为了将子进程的系统计数添加到父进程中, 我们在 **kernel/proc.c** 文件的退出函数中添加了此计数

```

1      void
2      exit(int status)
3      {
4          ...
5          // 把子进程的各系统调用的次数分配给该进程
6          for(int i=0;i<32;i++){
7              p->parent->syscallCount[i] += p->syscallCount[i];}
8      }

```

5. 在 `kernel/syscall.h` 文件中，我定义了系统调用号 `getSysCount` 计数为 22
6. 在 `kernel/syscall.c` 文件中添加了系统调用的原型：`extern uint64 sys_getSysCount(void)` ；并将其添加到 `syscalls` 数组中
7. 在 `kernel/sysproc.c` 文件中，我添加了系统调用的实现
 - 我首先增加了系统调用 22
 - 然后我从参数中提取掩码，并将其传递给 `getSysCount` 函数
 - 返回 `getSysCount` 函数返回的值

```
1      uint64
2      sys_getSysCount(void){
3          myproc()->syscallCount[SYS_getSysCount]++;
4          int mask; // 这是系统调用的掩码
5          argint(0, &mask);
6          return getSysCount(mask);
7      }
```

8. 关于 `getSysCount` 函数在 `kernel/proc.c` 文件中实现：

```
1      // 定义一个函数，用于获取指定系统调用的计数
2      int getSysCount(int mask){
3          int getsysCallId = -1;
4          // 遍历位掩码的所有位，寻找第一个被设置的位
5          for(int i = 0; i < 32; i++){
6              // 如果第i位被设置（即不为0），则记录该位的位置
7              if((mask & (1 << i)) != 0){
8                  getsysCallId = i;
9                  break; // 找到后退出循环
10             }
11         }
12         // 如果没有找到被设置的位，返回-1表示错误
13         if(getsysCallId == -1){
14             return -1;
15         }
16         // 打印当前进程的PID
17         printf("my calling pid: %d\n", myproc()->pid);
18         // 获取当前进程对应系统调用ID的系统调用计数
19         int countSysCall = myproc()->syscallCount[getsysCallId];
20         // 返回系统调用计数
21         return countSysCall;
22     }
```

9. 在 `user/user.pl` 文件中定义了系统调用

```
1 entry("getSysCount");
```

10. 在 `user/user.h` 文件中添加了函数原型

```
1 int getSysCount(int mask);
```

11. 在 `kernel/defs.h` 文件中添加了函数声明

```
1 int getSysCount(int);
```

现在我们可以用户在用户空间中使用这个系统调用来获取系统计数通过在用户目录中制作程序 `syscount.c` 来调用。

[可以点击这里跳转到该系统调用的测试文档查看详细的测试过程](#)

2.3 添加系统调用 countproc

1. 该系统调用的功能: 用户空间的程序可以通过该系统调用获取当前系统中正在运行或睡眠状态的进程数量, 并且打印出每个进程的 PID、状态、内存使用情况以及文件描述符数目。

2. countproc 的作用和实现的意义:

- 进程状态监控: 该系统调用提供了对当前活跃进程的监控功能, 能够帮助开发者或运维人员快速了解系统中哪些进程处于活跃状态 (运行或睡眠), 并获取有关它们的资源使用情况 (如内存、文件描述符等)。
- 性能分析: 在操作系统的调试和优化过程中, 了解哪些进程正在消耗系统资源以及它们的资源分配情况是非常重要的。通过调用 countproc, 可以方便地进行性能分析, 找出可能的瓶颈。进程管理: countproc 提供了一个高效的方式来获取系统中活动进程的数量和相关信息, 从而帮助系统管理员或开发者更好地管理进程和系统资源, 尤其是在多任务环境下。
- 内存与文件描述符监控: 通过 countproc, 可以获取每个进程的内存使用量以及文件描述符的数量, 这对于系统的资源管理和审计尤为重要。内存泄漏或文件描述符泄漏是系统常见的性能问题, 通过该系统调用能够及时发现和修复这些问题。通过该系统调用, 开发者可以在用户空间获取系统运行状态, 进行必要的调试、性能分析和系统优化, 进而提升操作系统的整体运行效率。

关于 countproc 的实现:

- **syntax:countproc**
- 返回值: 系统调用返回的是目前系统中在运行的进程数量

下面是关于系统调用 countproc 实现的整个过程

- 首先我们在 `kernel/syscall.h` 文件中, 我们定义了系统调用号 `SYS_countproc` 计数为 23

- 然后，在 `kernel/syscall.c` 文件中，我添加了系统调用的原型，即 `extern uint64 sys_countproc(void);` 并将其添加到 `syscalls` 数组中
- 在 `kernel/sysproc.c` 文件中，我添加了系统调用的实现

```

1      uint64
2      sys_countproc(void)
3      {
4          return countproc();
5      }

```

- 关于 `countproc` 函数在 `kernel/proc.c` 文件中实现: 在这里需要说明的一点是，为了打印出每个进程打开的文件描述符的数量，还在这个文件里面构建了一个辅助函数进行查找每一个进程的文件描述符的打开数量

```

1      // 这里假设 get_fd_count 计算并返回每个进程打开的文件描述符数量
2      int get_fd_count(struct proc *p) {
3          int i, count = 0;
4          for(i = 0; i < NOFILE; i++) {
5              if(p->ofile[i] != 0) {
6                  count++;
7              }
8          }
9          return count;
10         /*上面是辅助函数，下面才是该系统调用的具体实现*/
11         int countproc(void) {
12             struct proc *p;
13             int count = 0;
14             // 遍历进程表
15             printf( "PID, State, Memory, FD, Count\n");
16             for(p = proc; p < &proc[NPROC]; p++) {
17                 if(p->state == RUNNING || p->state == SLEEPING) {
18                     count++;
19                     printf( "PID:%d, State:%d, Memory:%ld, FD count:%d\n",
20                         p->pid, p->state, p->sz, get_fd_count(p));
21                 }
22             }
23         }
24         return count;
25     }

```

- 在 `user/user.pl` 文件中定义了系统调用

```

1      entry("countproc");

```

- 在 `user/user.h` 文件中添加了函数原型


```
1      int countproc(void);
```

- 在 `kernel/defs.h` 文件中添加了函数声明

```
1      int countproc(void);
```

现在我们可以用户在用户空间中使用这个系统调用来获取系统计数通过在用户目录中制作程序 `countproc.c` 来调用。

[可以点击这里查看该系统调用的测试文档](#)

2.4 添加系统调用 `sigalarm` 和 `sigreturn`

1. `sigalarm` 和 `sigreturn` 的功能

- **`sigalarm`** 系统调用的功能: **`sigalarm`** 系统调用设置一个定时器, 指定进程在一定的时间(以 `ticks` 为单位)后接收到一个信号(通常是一个中断或警告信号), 并调用指定的信号处理程序(`handler`)。
- **`sigreturn`** 系统调用的功能: **`sigreturn`** 系统调用用于恢复进程在处理信号时的状态。当进程收到信号并执行信号处理程序后, 通过调用 **`sigreturn`** 可以恢复到信号触发前的执行状态。具体来说, 它会恢复进程的 `trapframe`, 重置定时器状态, 并返回到信号处理前的执行位置。

2. `sigalarm` 的作用和实现的意义:

- (a) 定时任务管理: **`sigalarm` 系统调用**允许用户为进程设置定时器, 可以在一定时间间隔后触发特定信号。该功能在定时任务、超时控制等场景中非常有用。例如, 可以通过 `sigalarm` 定时器来控制进程的超时执行, 避免因某些操作长时间不返回结果而导致系统卡死或资源浪费。
- (b) 信号机制与进程控制: **`sigalarm` 系统调用**实现了操作系统信号机制的一个应用, 允许用户自定义定时事件的处理。通过向进程发送 `SIGALRM` 信号, 进程可以在信号处理程序中实现特定的处理逻辑。这增强了操作系统对进程的控制能力, 使得进程可以响应外部事件或时间限制, 达到更精细的调度与管理。
- (c) 资源管理与超时控制: 在系统中, 尤其是在多任务环境下, 资源的合理分配和管理非常重要。**`sigalarm`** 提供了一种机制来控制进程的最大运行时间或等待时间, 避免由于进程阻塞导致系统资源被过多占用。此外, **`sigalarm`** 还可用于检测长时间未响应的进程并采取必要的措施, 如自动终止进程或报警。
- (d) 增强系统调度与响应能力: **`sigalarm` 系统调用**不仅能在进程中设定定时器, 并且可以与其他系统调用(如信号处理、进程控制等)结合使用, 增强操作系统对进程状态的响应能力。定时器的触发能够与进程的调度机制、信号处理机制等紧密配合, 优化系统性能和资源利用率。

3. `sigreturn` 的作用和实现的意义:

- (a) 信号处理恢复：在进程执行信号处理程序后，**sigreturn** 提供了一种机制使进程能够恢复到信号触发之前的状态。这对于信号处理后的状态恢复至关重要，确保进程能够继续执行原本的操作。
- (b) 进程控制：**sigreturn** 通过恢复进程的 **trapframe**，将进程从信号处理程序中恢复出来，使其可以继续执行。当进程在信号处理时发生状态变更时（如通过 **sigalarm** 设置了定时器），**sigreturn** 可以帮助清理和恢复这些变更，避免进程陷入不一致状态。
- (c) 定时器与信号机制：**sigreturn** 与 **sigalarm** 系统调用紧密配合，管理定时器的状态。当定时器触发时，信号处理程序会被执行，而 **sigreturn** 则确保进程在处理完信号后恢复原状态。通过这种机制，可以实现对时间和信号的精确控制，避免不必要的资源浪费。
- (d) 提升操作系统的稳定性：**sigreturn** 确保进程能够在处理信号后继续执行，避免信号处理导致的系统状态不一致问题。通过恢复 **trapframe** 和定时器状态，它帮助操作系统保持稳定的进程调度和管理。**xv6** 内核本身并没有完整的信号处理机制，这两个系统调用的实现实际上是在操作系统中引入了一种基础的信号机制，允许进程设置定时器，在指定时间触发信号，并能够通过信号处理程序进行响应。这为进程提供了以下功能：
 - **定时任务**：进程可以设置定时器，并在指定的时间点执行某个任务，类似于操作系统的 `sleep` 函数，但是更加灵活。进程无需手动轮询，内核会在定时器到期时自动触发信号并调用相应的处理程序。
 - **信号处理**：信号机制是 Unix 系统中一个核心的概念，允许进程响应外部事件（如定时器到期、外部中断等）。通过实现 `sigalarm` 和 `sigreturn`，**xv6** 增强了对信号的处理能力，使得进程可以在内核级别管理信号的发送和处理。

下面是这两个系统调用的具体实现

- 在 `kernel/syscall.h` 文件中做出更改，定义 **sigalarm** 和 **sigreturn** 的系统调用编号

```
1      #define SYS_sigalarm 24
2      #define SYS_sigreturn 25
```

- 在 `kernel/syscall.c` 文件中为系统调用添加了原型

```
1      extern uint64 sys_sigalarm(void);
2      extern uint64 sys_sigreturn(void);
```

- 在 `kernel/syscall.c` 文件的 **syscalls 数组**中添加了它们的条目

```
1      [SYS_sigalarm] sys_sigalarm,
2      [SYS_sigreturn] sys_sigreturn,
```

- 在用户空间的 `user/user.pl` 文件中添加了它们

```
1      entry("sigalarm");
2      entry("sigreturn");
```

- 在 `user/user.h` 文件中添加了函数原型

```
1      int sigalarm(uint64, void (*handler)());
2      int sigreturn(void);
```

- 在 `kernel/defs.h` 文件中添加了函数声明

```
1      int sigalarm(uint64, void (*handler)());
2      int sigreturn(void);
```

- 在 `kernel/sysproc.c` 文件中添加了 `sigalarm` 系统调用的实现

```
1      uint64
2      sys_sigalarm(void)
3      {
4          myproc()->syscallCount[SYS_sigalarm]++;
5          uint64 ticks;
6          uint64 handler;
7          argaddr(0, &ticks);
8          argaddr(1, &handler);
9          if(ticks < 0){
10             return -1;
11         }
12         return sigalarm(ticks, (void (*)(void))handler);
13     }
```

- 在 `kernel/sysproc.c` 文件中添加了 `sigreturn` 系统调用的实现

```
1      uint64
2      sys_sigreturn(void)
3      {
4          myproc()->syscallCount[SYS_sigreturn]++;
5          return sigreturn();
6      }
```

- 在 `kernel/proc.c` 文件中定义了 `sigalarm` 和 `sigreturn` 函数

```
1      int sigalarm(uint64 ticks, void (*handler)()) {
2          struct proc *process = myproc(); // 获取当前进程
3          process->sigretaddr = (uint64)handler; // 保存处理程序的地址
4          process->alarmticks = ticks; // 设置警告的时间刻度数
5          return 0; // 返回 0 表示成功
6      }
7      int sigreturn(void) {
8          struct proc *process = myproc(); // 获取当前进程
9          if (process->trapframebackup == 0) { // 检查是否存在备份的 trapframe
10             return -1; // 如果没有, 返回 -1 表示失败
```

```

11         }
12         process->alarmticks = process->passedticks; // 重置警告的时间刻度数
13         process->passedticks = 0; // 重置已通过的时间刻度数
14         memmove(process->trapframe, process->trapframebackup, sizeof(*process
            ->trapframe)); // 恢复保存的 trapframe
15         return process->trapframebackup->a0; // 返回恢复后的 trapframe 中的值}

```

- 在 `kernel/proc.c` 文件中的 `allocproc` 函数中初始化了 `alarmticks`、`passedticks` 和 `trapframebackup` 变量

```

1         // Sigalarm 和 sigreturn
2         p->alarmticks = 0; // 初始化 alarmticks
3         p->sigretaddr = 0; // 初始化 sigretaddr
4         p->passedticks = 0; // 初始化 passedticks
5         p->trapframebackup = 0; // 初始化 trapframebackup

```

- 在 `kernel/proc.h` 文件中将必要的变量添加到 `proc` 结构中：用于存储信号处理程序地址、警告时间刻度数、已通过时间刻度数和 `trapframe` 的备份。

```

1         struct proc {
2             ...
3             uint64 sigretaddr; // 信号处理程序的地址
4             uint64 alarmticks; // 设置的警告时间刻度数
5             uint64 passedticks; // 已经过的时间刻度数
6             struct trapframe *trapframebackup; // 当前 trapframe 的备份
7             ...
8         };

```

- 在 `kernel/trap.c` 文件中添加了处理警告信号的逻辑。`checkAlarm` 函数检查警告是否超时，`setEPC` 函数将 `EPC` 设置为处理程序地址，`resetAlarm` 函数重置警告，`backUpTrapFrame` 函数备份 `trapframe`，`logicForTicking` 函数处理时间刻度逻辑。

```

1         // 检查警告是否超时
2         int checkAlarm(struct proc *process){
3             if(process->alarmticks < process->passedticks){
4                 return 1;
5             }
6             return 0;
7         }
8         ///////////////////////////////////////////////////////////////////
9         // 如果警告超时，设置 EPC 为处理程序地址
10        void setEPC(struct proc *process){
11            process->trapframe->epc = process->sigretaddr; // 设置 EPC 为处理程序
                地址
12        }
13        ///////////////////////////////////////////////////////////////////

```

```

14 // 重置警告
15 void resetAlarm(struct proc *process){
16     process->passedticks = process->alarmticks; // 重置已经过的时间刻度数
17     process->alarmticks = 0; // 禁止警告
18 }
19 ///////////////////////////////////////////////////
20 // 备份 trapframe

```

```

1 void backUpTrapFrame(struct proc *process){
2     process->trapframebackup = (struct trapframe *) kalloc(); // 为备份的
    trapframe 分配内
3 存
4     memmove(process->trapframebackup, process->trapframe, sizeof(struct
    trapframe)); //
5 将当前 trapframe 复制到备份 trapframe
6 }
7 ///////////////////////////////////////////////////
8 void logicForTicking(struct proc *process){
9     if(process->alarmticks>0){
10         if(checkAlarm(process)){
11             resetAlarm(process); // 步骤 1
12             backUpTrapFrame(process); // 步骤 2
13             setEPC(process); // 步骤 3
14         }
15         else{
16             process->passedticks++;
17         }
18         return;
19     }
20 }

```

可以点击[这里](#)查看该系统调用的测试文档

2.5 系统调用 sys_symlink 实现

符号链接的实现需要通过系统调用来完成。以下是 **sys_symlink** 的实现步骤：

- 从用户态获取目标路径（即符号链接指向的文件或目录）和符号链接文件的路径。
- 调用 **create** 函数创建一个新的 **inode**，并将其类型设置为 **T_SYMLINK**。
- 将目标路径写入该 **inode** 的数据部分（通过 **writel** 实现）。
- 提交事务并释放 **inode**。

```

1 uint64
2 sys_symlink(void) {

```

```

3      char target[MAXPATH], path[MAXPATH]; // 目标路径和符号链接路径
4      struct inode* ip_path;
5      // 从寄存器中获取目标路径和符号链接路径
6      if(argstr(0, target, MAXPATH) < 0 || argstr(1, path, MAXPATH) < 0) {
7          return -1;
8      }
9      begin_op();
10     // 创建一个新的 inode, 类型为符号链接
11     ip_path = create(path, T_SYMLINK, 0, 0);

```

```

1      if(ip_path == 0) {
2          end_op();
3          return -1;
4      }
5      // 将目标路径写入符号链接 inode
6      if(writei(ip_path, 0, (uint64)target, 0, MAXPATH) < MAXPATH) {
7          iunlockput(ip_path);
8          end_op();
9          return -1;
10     }
11     iunlockput(ip_path);
12     end_op();
13     return 0;
14 }

```

可以[点击这里](#)查看该系统调用的测试文档

2.6 系统调用 sys_open 的实现

当打开文件时，如果文件是符号链接，**sys_open** 需要解析符号链接并打开其指向的目标文件。为避免符号链接形成循环，我们设置了最大访问深度 **MAX_SYMLINK_DEPTH**（设为 10），防止递归过深。

在打开符号链接时，我们逐层解析符号链接，直到找到实际的目标文件。如果达到最大深度，则返回错误。

```

1      uint64
2      sys_open(void) {
3          ...
4          // 处理符号链接
5          if(ip->type == T_SYMLINK && !(omode & O_NOFOLLOW)) {
6              for(int i = 0; i < 10; ++i) {
7                  // 读取符号链接指向的路径
8                  if(readi(ip, 0, (uint64)path, 0, MAXPATH) != MAXPATH) {
9                      iunlockput(ip);
10                     end_op();
11                     return -1;

```

```

12     }
13     iunlockput(ip);
14     ip = namei(path); // 获取路径对应的 inode
15     if(ip == 0) {
16         end_op();
17         return -1;
18     }
19     ilock(ip);
20     if(ip->type != T_SYMLINK)
21         break;}

```

```

1
2 // 如果递归超过最大深度, 返回错误
3     if(ip->type == T_SYMLINK) {
4         iunlockput(ip);
5         end_op();
6         return -1;
7     }
8 }
9 ...
10 return fd;
11 }

```

3 进程调度

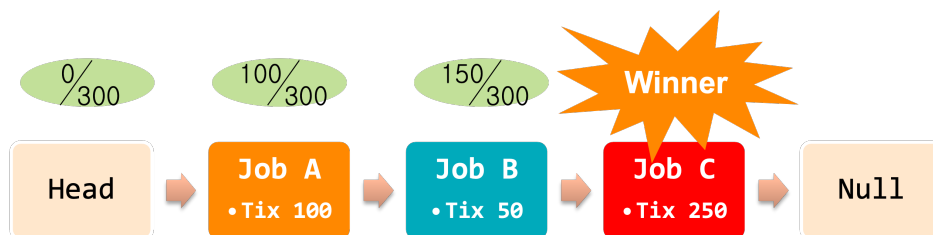
xv6 调度器实现了一个简单的调度策略, 它依次运行每个进程。这种策略被称为轮询调度 RR(round robin)

它存在明显的缺点:

- 不区分进程优先级: 轮询调度不考虑进程的优先级, 所有进程平等对待, 这可能导致重要进程的响应时间不够快。
- 上下文切换开销: 频繁的进程切换可能会导致较大的上下文切换开销, 尤其是在时间片很短的情况下。

但是在实际的操作系统当中会实现更加复杂的策略: **允许进程有优先级等**: 因此在本次内核优化过程中我们实现了两种不同的进程调度策略: 分别是 **Lottery scheduling** 和 **MLFQ**

3.1 Lottery Scheduling (彩票调度)



彩票调度是一种基于概率的调度算法，它为每个进程分配一定数量的“彩票”。调度器随机选择一个彩票来决定下一个运行的进程。进程获得的彩票数量与其优先级成正比。

3.1.1 概述

彩票调度是一种 **基于概率的进程调度算法**，通过给每个进程分配一定数量的“彩票” (numTickets)，随机抽取一张彩票以决定哪个进程获得 CPU 使用权。进程拥有的彩票越多，它被调度的概率就越高。这种调度机制以概率公平性为核心，同时允许通过分配更多的“彩票”优先考虑某些关键任务。

在 xv6 内核中，传统的调度器采用的是 **RR (Round-Robin, 时间片轮转) 算法**，按照固定的时间片循环分配 CPU 时间。相比之下，彩票调度通过概率机制实现了一种动态优先级模型，使得调度策略更加灵活。

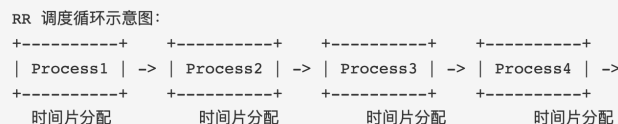
彩票调度在 xv6 中的优化点：

与时间片轮转 (RR) 调度相比，基于彩票的调度主要实现了以下优化：

- **动态优先级调度：**在 RR 调度中，所有进程无差别地按时间片循环分配 CPU，而彩票调度通过调整彩票票数实现动态优先级。票数越多的进程更有可能被调度，这样关键任务可以优先完成。
- **增强系统公平性：**每个进程的调度概率与其票数正相关。相较于 RR 的严格时间片分配，彩票调度提供了一种更灵活的公平性模型，适用于多用户、多任务场景。

下面是图例展示：

- **时间片轮转调度 (RR)：**所有进程按照循环顺序分配固定时间片 (timeSlice)，调度过程如下：



特点：

- 所有进程获得相同的调度机会。
- 优先级无法调整。

- **彩票调度 (LBS):** 每个进程根据票数获得一定的调度概率。以下示例中, Process1 具有 5 张彩票, Process2 具有 3 张彩票, Process3 具有 2 张彩票, 总票数为 10:

彩票分布图:

```

+-----+-----+-----+-----+
| Process1 | Process1 | Process1 | Process1 | Process1 |
+-----+-----+-----+-----+
| Process2 | Process2 | Process2 | Process3 | Process3 |
+-----+-----+-----+-----+

```

[抽中 Process1] -> Process1 获得调度权

特点:

- **概率驱动:** Process1 的调度概率为 50% (5/10), 而 Process3 的调度概率为 20% (2/10)。
- **动态优先级:** 可以通过调整彩票数量控制调度机会。

为了实现彩票调度算法我们在 proc 进程结构体中增加了结构体变量用于实现彩票调度, 然后实现了为进程分配彩票的逻辑, 以及处理彩票调度的逻辑。

3.1.2 数据结构

- 在 kernel/proc.c 文件中实现了 **彩票调度算法 (lottery scheduling)**
 - 首先, 我们在 kernel/proc.h 文件中的 **proc 结构体**中添加了必要的变量, 用于存储彩票票数。

```

1      struct proc {
2          ...
3          // 用于彩票调度的变量
4          int numTickets; // 进程的彩票票数
5          int timeOfArrival; // 进程到达时间
6          uint64 timeSlice; // 进程的时间片大小
7          ...
8      };

```

1. **numTickets:** 用于存储进程拥有的“彩票”数量。每个进程的“彩票”数量代表它被选中调度的概率, 票数越多, 被调度的可能性越大。这是实现彩票调度算法的**核心参数**。
2. **timeOfArrival:** 用于记录进程到达就绪队列的时间, 可以在某些调度策略中辅助实现公平性或用于调试分析。
3. **timeSlice:** 用于表示进程在被调度后可以运行的时间片大小。彩票调度通常随机选择一个进程执行, timeSlice 决定了它的执行时间。

下面是图例展示以下是这段代码在彩票调度算法中所起的作用的简要图示:

Process 1	numTickets: 5	
Process 2	numTickets: 3	
Process 3	numTickets: 2	

<-----随机选择----- Winning Ticket! |

		Running Process 1
		timeSlice: 4ms

1. 每个进程分配了一定的 **numTickets**。
2. 调度器会生成一个随机数，范围是所有进程的票数总和。
3. 根据随机数，找到拥有对应票数范围的进程（例如进程 1）。
4. 选中的进程获得调度机会，并运行 **timeSlice** 指定的时间。

通过这些字段的扩展，可以实现一个基于概率的调度机制—> 为后续的实现提供了必要条件，让进程获得一定的调度公平性，同时支持简单的优先级管理（通过调整票数）

3.1.3 彩票调度算法实现

初始化彩票调度机制变量

- 在 **kernel/proc.c** 文件中对 **allocproc** 函数进行扩展：以便实现用于初始化彩票调度机制中新增的变量

```

1      void
2      allocproc(void)
3      {
4          ...
5          // 用于彩票调度的初始化
6          if(p->parent == 0){
7              p->numTickets = 1 ; // 默认票数为 1
8          }
9          else{
10             p->numTickets = p->parent->numTickets; // 继承父进程的票数
11          }
12          #ifdef LBS
13             // p->numTickets = pseudoRandom(r_time(0),1000);
14             // printf("进程 %s 初始化时分配了 %d 张票\n", p->name, p->
                numTickets);
15          #endif
16          p->timeOfArrival = ticks; // 设置进程到达时间
17          p->timeSlice = 1; // 默认时间设为 1 个 tick
18          ...
19      }

```

下面是我们在 **allocproc** 函数中修改部分的解释:

1. `p->numTickets` 的初始化:

- 对于初始进程 (`p->parent == 0`), 分配默认票数为 1。
- 当然我们在实现的时候考虑到如果是子进程 (有父进程), 它能够继承父进程的票数。这种继承机制可以维持进程组的一致性, 也可以进一步扩展为动态调整票数策略。
- 在操作系统内核启用了宏 **LBS** 时, 可以用伪随机数生成逻辑 (已注释) 随机分配票数, 这样可以模拟更动态的调度概率分布。

2. `p->timeOfArrival`:

- 使用内核全局计时器 **ticks** 来记录进程创建时的到达时间。这个变量的意义是在我们编写测试程序的时候能够帮助度量调度性能, 比如周转时间或等待时间。

3. `p->timeSlice`: 我们首先为每个进程分配默认的时间片大小, 设置为 1 tick。

4. `#ifdef LBS`:

- 条件编译部分是在我们编写该调度算法时候调试用到的代码, 现在实现了加以注释!
- 注释的伪代码:

```
1 p->numTickets = pseudoRandom(r_time(0),1000);
```

这可以通过伪随机函数生成介于 `[1, 1000]` 之间的随机票数, 用于更复杂的调度场景。

下面是图例描述

```
allocproc()
+-----+
| Parent Process |
+-----+
| numTickets: 3   | <-- Parent process has 3 tickets
| timeOfArrival: t0 |
+-----+

      |
      | Fork Child Process
      V

+-----+
| Child Process   |
+-----+
| numTickets: 3   | <-- Inherits parent's tickets
| timeOfArrival: t1 | <-- Arrival time set to ticks
| timeSlice: 1    | <-- Default timeslice
+-----+
```

进程分配彩票逻辑

- 在 `kernel/proc.c` 文件中添加了逻辑，为进程分配彩票。

Listing 1: 使用伪随机数获取出事进程的彩票概率

```
1 //////////////////////////////////////////////////
2 // 使用伪随机数生成器获取彩票概率
3 uint64 pseudoRandom(uint64 seed,int max){
4     uint64 multiplier = 1664525;
5     uint64 increment = 1013904223;
6     uint64 modulus = 4294967296;
7     seed = (seed * multiplier + increment) % modulus;
8     if(seed < 0){
9         seed = seed * -1;
10    }
11    if(max == 0){
12        return 0;
13    }
14    seed = seed % max;
15    return seed;
16 }
17 //////////////////////////////////////////////////
```

Listing 2: 算法选择进程

```
1 // 根据彩票算法检查当前进程是否应被选中
2 int atomicLBSFind(int random,int total,struct proc* p){
3     if(random <= total+p->numTickets){ // total < random <= total+p->
4         numTickets
5         return 1;}
6     return 0;}
7 struct proc *processToRunLBS(uint64 totalTickets, uint64 randomNumber)
8 {
9     struct proc* processToRun = 0; // 要运行的进程
10    uint64 total = 0; // 当前进程之前的票数总和
11    for (struct proc *p = proc; p < &proc[NPROC]; p++) // 遍历进程列表
12    {
13        acquire(&p->lock);
14        if (p->state == RUNNABLE)
15        {
16            if(atomicLBSFind(randomNumber,total,p)){
17                processToRun = p;
18                release(&p->lock);
19                break;
20            }
21        }
22        total += p->numTickets;
23    }
```

```

21         release(&p->lock);
22     }
23     return processToRun;
24 }

```

Listing 3: 彩票算法的主调度函数

```

1     struct proc *lotteryScheduling(){
2         uint64 totalTickets = getTotalTickets();
3         if(totalTickets == 0){
4             return 0; // 没有进程可运行
5         }
6         uint64 randomNumber = pseudoRandom(r_time(0),totalTickets)+1; // 随
           机数
7         struct proc* processToRun = processToRunLBS(totalTickets,randomNumber)
           ;
8         if(processToRun == 0){
9             return 0; // 没有进程可运行
10        }
11        uint64 minArrivalTime = processToRun->timeOfArrival;
12        processToRun = checkArrival(minArrivalTime,processToRun);
13        return processToRun;
14    }

```

3.1.4 集成 xv6 核心调度算法中

本部分将基于彩票的调度逻辑集成到 xv6 的核心调度流程中。代码通过条件编译指令 `#ifdef LBS` 控制，启用时运行彩票调度器。具体可以在编译的时候使用 `make SCHEDULER=LBS` 进行选择操作系统使用哪种调度算法

1. 调用彩票调度算法

```

1     p = lotteryScheduling();
2     if(p == 0){
3         continue; // 如果没有可运行的进程，则跳过当前循环
4     }

```

调用 `lotteryScheduling()`，返回根据彩票算法选择的目标进程 `p`。如果没有可运行进程（如所有进程都处于 **非 RUNNABLE** 状态），返回 0，跳过本轮调度。

2. 获取进程锁并检查状态

```

1     acquire(&p->lock);
2     if(p->state != RUNNABLE){
3         release(&p->lock);
4         continue; // 如果状态不是 RUNNABLE，跳过本轮调度
5     }

```

使用 **acquire** 获取目标进程的锁，防止其他内核线程或 CPU 核心对该进程状态的竞争访问。检查进程状态是否为 **RUNNABLE**。如果不满足条件（如进程被阻塞或终止），释放锁并跳过调度。

3. 设置进程状态为 **RUNNING**

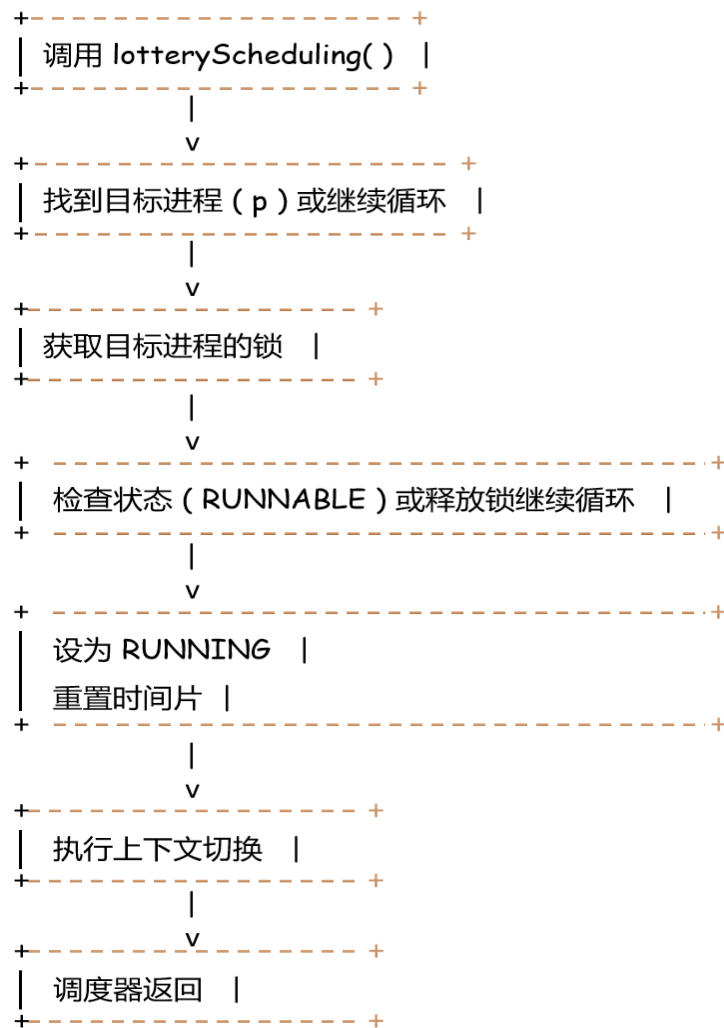
```
1         p->state = RUNNING;
2         c->proc = p;
3         p->timeSlice = 1; // 重置时间片
```

将目标进程的状态从 **RUNNABLE** 更改为 **RUNNING**，表明该进程已被调度，并将当前 CPU 的 **proc** 指针设置为当前调度的进程 **p**，最后重置进程的时间片为 1，用于限制进程的运行时长。

4. 上下文切换

```
1         swtch(&c->context, &p->context);
2         c->proc = 0;
3         release(&p->lock);
```

调用 **swtch** 进行上下文切换，从调度器切换到目标进程，上下文切换后，当目标进程运行结束或被中断时，返回调度器，最后清空当前 CPU 的 **proc** 指针，并释放目标进程的锁。



- 在 `kernel/proc.c` 文件的 `scheduler` 函数中添加了处理彩票调度的逻辑。

```

1  #ifdef LBS
2  // 彩票调度
3  p = lotteryScheduling();
4  if(p == 0){
5      continue;
6  }
7  acquire(&p->lock);
8  if(p->state != RUNNABLE){
9      release(&p->lock);
10     continue;
11 }
12 p->state = RUNNING;
13 c->proc = p;
14 p->timeSlice = 1; // 重置时间片
15 swtch(&c->context, &p->context);
16 c->proc = 0;
17 release(&p->lock);
18 #endif

```

- 在 `trap.c` 文件中添加了逻辑，处理进程时间片的递减。

```

1      #ifdef LBS
2      if(p!=0 && p->state == RUNNING){
3          p->timeSlice--; // 减少时间片
4          if(p->timeSlice <= 0){ // 如果时间片用完
5              p->timeSlice = 1; // 重置时间片
6              yield(); // 放弃 CPU
7          }
8      }
9      #endif

```

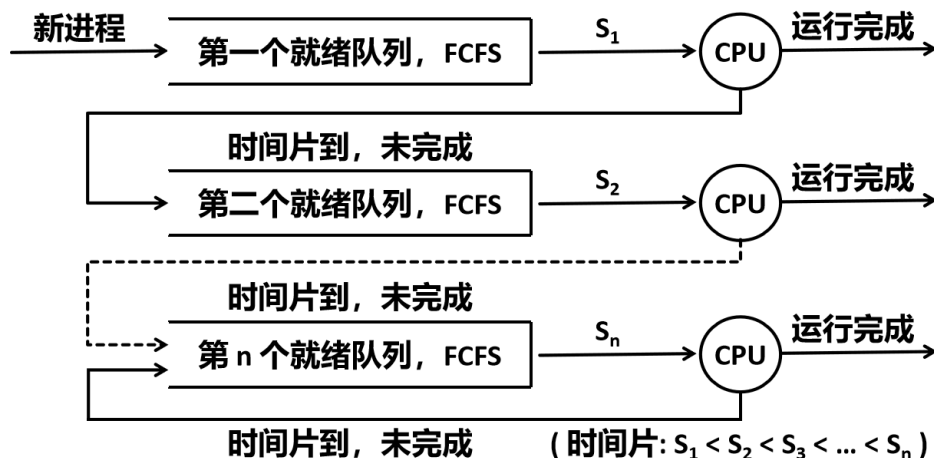
- 在 `kernel/proc.h` 文件中为 `proc` 结构体添加了变量，用于存储每个进程的彩票票数、到达时间和时间片。
- 在 `kernel/proc.c` 文件中实现了彩票调度的逻辑，包括伪随机数生成、进程选择和时间片管理。
- 在 `scheduler` 函数中整合了彩票调度算法。
- 在 `trap.c` 文件中管理时间片，并确保时间片到期时触发进程调度。

3.1.5 测试

- 我们还添加了一个系统调用 `settickets` 用于测试和验证。

具体的测试可以点击[这个链接](#)进行查看

3.2 MLFQ（多级反馈队列）



在基于 `xv6` 内核实现的多级反馈队列调度算法中，我们的目标是通过不同的优先级队列对进程进行管理，以实现更高效和更公平的进程调度。多级反馈队列调度（MLFQ）算法是一种常见的调度机制，它能够根据进程的行为动态调整其优先级，从而兼顾交互性任务的响应速度和长时间运行任务的公平性。

3.2.1 数据结构扩充—优先级队列

- 在 `kernel/proc.c` 文件中实现了多级反馈队列（MLFQ）调度算法。
 - 首先，我们在 `kernel/proc.h` 文件中的 `proc` 结构体中添加了必要的变量，用于存储优先级、时间片、进入队列的时间以及执行时间。

```
1      struct proc {
2          ...
3          int priorityLevel; // 进程的优先级
4          int slice; // 分配的时间片（以 ticks 为单位）
5          int entryTime; // 进程进入队列的时间，用于 MLFQ 调度
6          int execTime; // 进程上次进入队列后的累计执行时间
7          ...};
```

- 在 `kernel/proc.c` 文件中的 `allocproc` 函数中初始化这些变量。

```
1      void
2      allocproc(void)
3      {
4          p->priorityLevel=0; // 默认优先级为 0（最高优先级）
5          p->slice=_RESET_SLICE_MLFQ_; // 初始化时间片
6          p->entryTime=ticks; // 记录进入队列的时间
7          p->execTime=0; // 初始化执行时间
8          ...
9      }
```

- 在 `kernel/param.h` 文件中定义了一些用于 MLFQ 调度的宏和全局变量。

```
1      #define _MAXLEVEL_MLFQ_      4      // MLFQ 中的最大队列级别
2      #define _MAXPROCESS_MLFQ_    64      // MLFQ 中的最大进程数
3      #define _PRIORITY_BOOST_MLFQ_ 48 // 优先级提升的时间间隔
4      #define _MAXXXXXX_ 1000000000 // 一个很大的数（1e9）
5      #define _RESET_SLICE_MLFQ_ 1 // 时间片重置值
6
7      extern int mlfqSlice[_MAXLEVEL_MLFQ_];
```

3.2.2 核心算法实现

- 在 `kernel/proc.c` 文件中的 `scheduler` 函数中添加了 MLFQ 调度的逻辑，同时添加了一些辅助函数。

Listing 4: 根据 MLFQ 调度逻辑选择要运行的进程

```

1 void foundProcessToRun(int __FOUND_PROCESS_TO_RUN__, struct proc *processToRun,
2   struct cpu *c){
3   if(!__FOUND_PROCESS_TO_RUN__)
4   {
5       return;
6   }
7
8   processToRun->state = RUNNING; // 将进程状态设置为 RUNNING
9   c->proc = processToRun; // 将当前进程设置为要运行的进程
10  swtch(&c->context, &processToRun->context); // 切换上下文
11
12  c->proc = 0; // 重置当前 CPU 上的进程
13
14  release(&processToRun->lock); // 释放进程的锁
15 }

```

Listing 5: 原子检查是否有更优的进程可运行（根据优先级和到达时间）

```

1 int atomicCheckProcessToRunMLFQ(struct proc *p, struct proc *processToRun, uint
2   possiblePriority, uint lowestEntryTime){
3   if(p->priorityLevel < possiblePriority || (p->priorityLevel ==
4     possiblePriority && p->entryTime < lowestEntryTime))
5   {
6       return 1;
7   }
8   return 0;
9 }

```

```

1 void scheduler(void){
2   ...
3   #elif defined MLFQ
4   int __FOUND_PROCESS_TO_RUN__ = 0;
5   struct proc *processToRun = 0;
6   uint lowestEntryTime = _MAXXXXXX_; // 初始化为一个很大的数 (1e9)
7   uint possiblePriority = _MAXLEVEL_MLFQ_; // 初始化为最大优先级
8   for(p = proc; p < &proc[NPROC]; p++)
9   {
10      acquire(&p->lock);
11      if(p->state == RUNNABLE)
12      {
13          // 找到最高优先级（数值最小）的可运行进程
14          // 如果多个进程具有相同的优先级，选择等待时间最长的

```

Listing 6: 集成操作系统的进程调度 2

```

1      if(atomicCheckProcessToRunMLFQ(p,processToRun,possiblePriority,
2          lowestEntryTime))
3      {
4          if(processToRun != 0)
5          {
6              // 释放之前选择的进程的锁
7              release(&processToRun->lock);
8              processToRun = p;
9              possiblePriority = p->priorityLevel;
10             lowestEntryTime = p->entryTime;
11             __FOUND_PROCESS_TO_RUN__ = 1;
12         }
13     else
14     {
15         // 释放不符合条件的进程的锁
16         release(&p->lock);
17     }
18
19     }
20     else
21     {
22         // 释放不可运行的进程的锁
23         release(&p->lock);
24     }
25     }
26     // 如果找到了进程，就运行它
27     foundProcessToRun(__FOUND_PROCESS_TO_RUN__,processToRun,c);
28     #else
29     ...}

```

- 在 `kernel/proc.c` 文件中初始化了 `mlfqSlice` 全局变量。

```

1      int mlfqSlice[_MAXLEVEL_MLFQ_] = {1, 4, 8, 16}; // 每个优先级的时间片大小

```

- 在 `trap.c` 文件中添加了逻辑，用于递增优先级提升计数器（`__TICKS_BOOST__`），并在达到特定间隔时提升所有进程的优先级

Listing 7: 递增优先级提升的计数器

```

1 void incrementTicksBoost(){
2     if(__TICKS_BOOST__ >= _PRIORITY_BOOST_MLFQ_)
3     {
4         __TICKS_BOOST__ = 0; // 重置计数器
5     } else
6     {
7         __TICKS_BOOST__++; // 增加计数器
8     }
}

```

Listing 8: 在 MLFQ 调度中实现优先级提升

```

1 void priorityBoost(struct proc *p){
2     if(__TICKS_BOOST__ < _PRIORITY_BOOST_MLFQ_) // 如果尚未达到提升间隔
3     {
4         return;
5     }
6     // 达到提升条件时，提升优先级
7     p->priorityLevel = 0; // 将优先级重置为最高级
8     p->slice = _RESET_SLICE_MLFQ_; // 重置时间片
9     p->execTime = 0; // 重置执行时间
10    return; }

```

Listing 9: 用户中断关于 MLFQ 算法

```

1 void usertrap(void)
2 { ...
3     #ifdef MLFQ
4     p->execTime++; // 增加进程的累计执行时间
5     whenToYield(p); // 检查是否需要切换
6     incrementTicksBoost(); // 更新优先级提升计数器
7     for(int i=0; i<NPROC; i++){
8         if(proc[i].state == RUNNABLE){
9             acquire(&proc[i].lock); // 加锁
10            priorityBoost(&proc[i]); // 提升优先级
11            release(&proc[i].lock); // 解锁
12        }
13    }
14    #endif
}

```

调度器开始时初始化了一些变量，以记录当前找到的最佳候选进程；调度器通过遍历进程表中的所有进程，逐一检查每个进程的状态是否为 **RUNNABLE**，并依据优先级和进入队列时间选择最佳进程 → 这里我们主要实现是在构造了辅助函数 **atomicCheckProcessToRunMLFQ** 进行确认。

- 在 **kernel/trap.c** 文件中初始化了 **__TICKS_BOOST__** 变量。

```
1      int __TICKS_BOOST__ = 0;
```

- 在 `kernel/proc.c` 文件中的 `wakeup` 函数中设置了进程的进入时间，以确保 I/O 绑定的进程可以被及时调度。

```
1      void wakeup(void *chan)
2      {
3          struct proc *p;
4          for (p = proc; p < &proc[NPROC]; p++)
5          {
6              if (p != myproc())
7              {
8                  acquire(&p->lock);
9                  if (p->state == SLEEPING && p->chan == chan)
10                 {
11                     p->state = RUNNABLE;
12                     p->entryTime = ticks; // 设置进入时间为当前 ticks
13                 }
14                 release(&p->lock);
15             }
16         }
```

- 在 `kernel/proc.h` 文件中为 `proc` 结构体添加了变量，用于存储优先级、时间片、进入队列时间和累计执行时间。
- 在 `trap.c` 文件中添加了计数器和优先级提升逻辑。
- 在 `scheduler` 函数中实现了多级反馈队列调度逻辑，选择最高优先级进程，并在必要时提升优先级。
- 在 `wakeup` 函数中更新进程的进入时间，以确保队列的公平性。

具体的测试可以点击[这个链接](#)进行查看

4 文件系统板块

4.1 概述

xv6 的文件系统和许多其他系统的实现大体相同，只是在某些地方有简化，但存储文件的方式都是以 block 的形式。物理磁盘在读写时以扇区为单位，通常来讲，每个扇区是 512 个字节。但操作系统读取磁盘时，由于寻道的时间是很长的，读写数据的时间反而没那么久，因此会操作系统一般会读写连续的多个扇区，所使用的时间几乎一样。操作系统以多个扇区作为一个磁盘块，xv6 是两个扇区，即一个 block 为 1024 个字节。

磁盘只是以扇区的形式存储数据，但如果没有一个读取的标准，该磁盘就是一个生磁盘。磁盘需要按照操作系统读写标准来存储数据，格式如下：从中可以看到，磁盘中不同区域的数据块有不同的功

能。第 0 块数据块是启动区域，计算机启动就是从这里开始的；第 1 块数据是超级块，存储了整个磁盘的信息；然后是 log 区域，用于故障恢复；bit map 用于标记磁盘块是否使用；接下来是 inode 区域和 data 区域。

磁盘中主要存储文件的 block 是 inode 和 data。操作系统中，文件的信息是存放在 inode 中的，每个文件对应了一个 inode，inode 中含有存放文件内容的磁盘块的索引信息，用户可以通过这些信息来查找到文件存放在磁盘的哪些块中。inodes 块中存储了很多文件的 inode。

文件描述符、file 结构体、索引节点之间的关系如下图：在 xv6 中，文件系统设计相对简单，使用了一级索引（即直接块索引）来存储文件的内容，同时支持硬链接。虽然这种设计对于简单的操作系统来说具有一定的优势，但它也存在一些缺点：

1. 文件大小限制：

- **一级索引的限制：**xv6 使用的是直接块索引（即文件的索引直接指向数据块），每个文件最多只能有 12 个直接块。因为文件内容是通过这些直接块来存储的，所以一个文件最大只能有 12 个数据块。如果文件大小超过了这个限制，xv6 无法处理大文件。
- **缺乏间接块索引：**为了支持大文件，通常需要多级索引（如间接块），这使得文件系统可以通过间接块来引用更多的数据块，而 xv6 仅依赖一级索引，无法扩展到更大的文件。

2. 文件系统的灵活性不足：

- **硬链接限制：**虽然硬链接允许多个目录项指向同一个文件，但 xv6 中的硬链接实现也存在一些限制。由于硬链接共享相同的 inode，删除任何一个硬链接只会减少引用计数，只有当所有硬链接都被删除后，文件的数据块和 inode 才会被释放。这可能导致空间的浪费，尤其是当有大量不再需要的硬链接时，文件系统无法及时清理这些数据。
- **硬链接的管理复杂性：**硬链接的管理也会增加文件系统的复杂性。例如，删除或修改硬链接时，可能会导致一些难以追踪的错误或资源泄漏。

3. 性能瓶颈：

- **直接块的存取限制：**由于所有的文件数据都通过直接块进行存取，文件的访问速度可能会受到影响，尤其是当文件非常大时。对于更复杂的文件系统结构，多级索引（如一级、二级间接块索引）可以更好地平衡文件的存取性能。
- **硬链接访问效率：**硬链接的存在可能导致文件访问的效率降低，尤其是在有多个硬链接指向同一个文件时，文件的修改和删除可能变得更加复杂和低效。

在实际的操作系统当中，**二级索引**和**符号链接**的引入这是对原有文件系统设计的显著改进，增强了 xv6 文件系统的扩展性、灵活性和用户友好性。

一级索引示意图如下：

4.2 混合索引文件优化

在 xv6 的文件系统中，仅使用一级索引（即直接块索引）来管理文件的存储。这种设计虽然简单并适用于小规模文件系统，但随着文件系统的规模扩大或文件的增大，一级索引会暴露出一些显著的缺点：

- **文件大小受到严重限制：**xv6 中的每个文件通过 12 个直接块来存储数据。由于每个直接块仅能指向一个数据块，这意味着一个文件的大小最多只能达到 12 个数据块的大小。
- **空间利用率低下：**xv6 的文件系统使用固定大小的数据块。如果文件大小不能正好填充这些块，剩余部分就会被浪费。
- **文件存取效率差：**所有的文件数据都是通过 12 个直接块来存储的。如果一个文件的数据量较大，超出 12 个直接块的范围，就无法直接扩展。由于缺少间接块的支持，文件系统在存取这些大文件时的效率将会降低，尤其是在查找文件的过程中。

在 **FtidesOS** 中，实现了对文件系统的二级索引扩展，旨在支持更大规模的文件存储。原始的 xv6 文件系统仅通过直接索引和一级间接索引来管理文件数据块，文件的大小受限于直接索引的数量。为了突破这一限制，我们对 inode 结构进行了扩展，新增了二级间接索引的支持。以下是具体实现。

4.2.1 数据结构

1. **#define NDIRECT 11:** 这表示每个 **inode** 结构中直接指向数据块的最大数量。在 xv6 中，NDIRECT 是 11，意味着 inode 结构可以直接包含 11 个数据块的地址。
2. **#define NINDIRECT (BSIZE / sizeof(uint)):** NINDIRECT 定义了间接索引块 (Indirect Block) 的容量，它指明一个间接索引块可以容纳多少个指向数据块的地址。**BSIZE** 是磁盘块的大小，而 **sizeof(uint)** 是指针的大小（通常是 4 字节），因此，NINDIRECT 表示一个间接索引块可以包含多少个块地址。
3. **#define NDINDIRECT NINDIRECT * NINDIRECT:** NDINDIRECT 表示二级间接索引的容量，即一个二级索引块能容纳的指向一级间接索引块的指针数量。这个定义意味着每个二级索引块指向多个一级间接索引块，而每个一级间接索引块指向多个数据块。
4. **#define MAXFILE (NDIRECT + NINDIRECT + NDINDIRECT):** MAXFILE 表示一个文件可以包含的最大数据块数量。它是通过 11 个直接块 (NDIRECT)、一个一级间接索引块 (NINDIRECT) 和一个二级间接索引块 (NDINDIRECT) 加起来得出的。

结构体定义：

```

1 struct dinode {
2     short type;           // 文件类型：区分文件、目录和特殊文件
3     short major;         // 设备的主设备号（仅适用于设备文件）
4     short minor;         // 设备的次设备号（仅适用于设备文件）
5     short nlink;         // 连接数：引用此 inode 的目录条目数
6     uint size;           // 文件大小（字节）
7     uint addrs[NDIRECT+2]; // 数据块地址：保存文件内容的磁盘块的块号
8 };
9 addrs[NDIRECT+2]:

```

- 这是一个包含 13 个元素的数组 ($NDIRECT + 2 = 11 + 2$)，用于存储文件数据块的地址。

- 其中前 11 个地址是直接地址（指向数据块），接下来的 1 个是一级间接索引块的地址，最后 1 个是二级间接索引块的地址。

在二级索引中的作用：

- **直接索引：**前 11 个地址直接指向数据块。它们用于存储较小的文件，或者文件中的一部分数据。
- **一级间接索引：**第 12 个地址指向一个间接块，该块本身包含多个指向数据块的地址。一级间接索引扩展了文件能存储的数据块数量。
- **二级间接索引：**第 13 个地址指向一个二级间接索引块，该块包含多个一级间接索引块的地址。通过这种方式，二级间接索引可以进一步扩展文件的容量，允许存储大量的数据块。

图示说明

假设文件的大小超过了 11 个直接块的限制，但又不够大到需要更多的二级间接索引。文件的数据块地址管理将采用如下结构：

dinode结构体：

```
-----
| type | major | minor | nlink | size | addrs[0-12] |
-----
      ↓      ↓      ↓      ↓      ↓      ↓
addrs数组:  [0] [1] [2] ... [10] [11] [12]
              | Direct Blocks | Indirect Block | Double Indirect Block |
```

1. **直接数据块 (0-10):** 每个索引直接指向数据块。
2. **一级间接块 (11):** 一个间接块，包含指向多个数据块的地址。
3. **二级间接块 (12):** 一个二级间接块，包含多个指向一级间接块的地址。每个一级间接块再指向多个数据块。

4.3 算法思路与实现

- textbfkernel/fs.h 文件中减小 **NDIRECT** 的值，为二级索引留一个位置：

Listing 10: On-disk inode structure

```
1  #define NDIRECT 11
2  #define NINDIRECT (BSIZE / sizeof(uint))
3  #define NDINDIRECT NINDIRECT * NINDIRECT
4  #define MAXFILE (NDIRECT + NINDIRECT + NDINDIRECT)
5  struct dinode {
6      ...
7      uint size;           // Size of file (bytes)
8      uint addrs[NDIRECT+2]; // Data block addresses
9  };
```


- 在 `kernel/file.h` 中更改内存中的 `inode` 结构

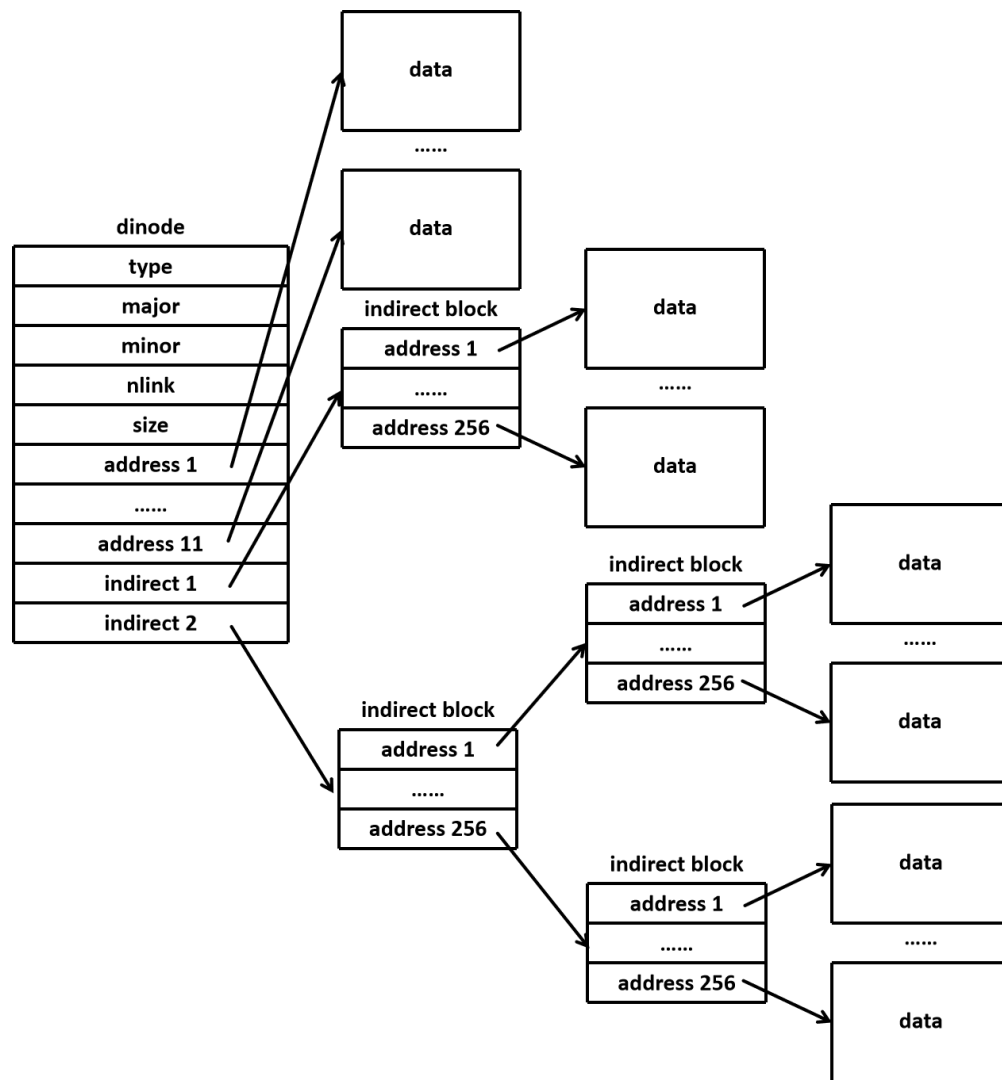
Listing 11: 二级索引节点结构修改

```
1      ...
2      uint size;
3      uint addrs[NDIRECT+2];
4  };
```

- 仿照一级索引写二级索引，在 `kernel/fs.c` 中添加代码：

```
1      static uint
2      bmap(struct inode *ip, uint bn)
3      {
4          // 二级索引
5          bn -= NINDIRECT;
6          if(bn < NDINDIRECT) {
7              if((addr = ip->addrs[NDIRECT+1]) == 0)
8                  ip->addrs[NDIRECT+1] = addr = balloc(ip->dev);
9              // 通过一级索引，找到下一级索引
10             bp = bread(ip->dev, addr);
11             a = (uint*)bp->data;
12             if((addr = a[bn/NINDIRECT]) == 0) {
13                 a[bn/NINDIRECT] = addr = balloc(ip->dev);
14                 log_write(bp);
15             }
16             brelse(bp);
17             // 重复上面的代码，实现二级索引
18             bp = bread(ip->dev, addr);
19             a = (uint*)bp->data;
20             if ((addr = a[bn%NINDIRECT]) == 0) {
21                 a[bn%NINDIRECT] = addr = balloc(ip->dev);
22                 log_write(bp);
23             }
24             brelse(bp);
25             return addr;
26         }
27         panic("bmap: out of range");
28     }
```

二级索引示意图如下：



在 `kernel/fs.c` 中，添加第二级索引的释放操作：将 `inode` 所指向的数据块（包括直接和间接索引块）全部释放，相当于删除文件

```

1 void
2 itrunc(struct inode *ip)
3 {
4     int i, j;
5     struct buf *bp;
6     uint *a;
7     for(i = 0; i < NDIRECT; i++){
8         if(ip->addrs[i]){
9             bfree(ip->dev, ip->addrs[i]);
10            ip->addrs[i] = 0;
11        }
12    }

```

```

13     if(ip->addr[NDIRECT]){
14         bp = bread(ip->dev, ip->addr[NDIRECT]);
15         a = (uint*)bp->data;
16         for(j = 0; j < NINDIRECT; j++){
17             if(a[j])
18                 bfree(ip->dev, a[j]);
19         }
20         brelse(bp);
21         bfree(ip->dev, ip->addr[NDIRECT]);
22         ip->addr[NDIRECT] = 0;
23     }
24     if(ip->addr[NDIRECT+1]) {
25         bp = bread(ip->dev, ip->addr[NDIRECT+1]);
26         a = (uint*)bp->data;
27         for(j = 0; j < NINDIRECT; j++) {
28             if(a[j]) {
29                 bp = bread(ip->dev, a[j]);
30                 a = (uint*)bp->data;
31                 for(i = 0; i < NINDIRECT; i++) {
32                     if(a[i]) bfree(ip->dev, a[i]);
33                 }
34                 brelse(bp);
35                 bfree(ip->dev, a[j]);
36                 a[j] = 0;
37             }
38         }
39         brelse(bp);
40         bfree(ip->dev, ip->addr[NDIRECT+1]);
41         ip->addr[NDIRECT] = 0;
42     }
43     ip->size = 0;
44     iupdate(ip);
45 }

```

具体结果参考测试文档。

4.4 文件共享优化

硬链接是指多个文件名指向同一个 inode 号码。有以下特点：

- 可以用不同的文件名访问同样的内容
- 对文件内容进行修改，会影响到所有文件名
- 删除一个文件名，不影响另一个文件名的访问

而软链接也是一个文件，但是文件内容指向另一个文件的 inode。打开这个文件时，会自动打开它指向的文件。在 xv6 文件系统中，只有硬链接而没有符号链接（软链接），这带来了一些限制和缺点：

- **灵活性差**：硬链接要求目标文件和链接必须位于同一个文件系统中，不能跨越不同的磁盘分区或文件系统，这限制了文件管理的灵活性
- **无法指向目录**：硬链接不能指向目录，因此不能创建目录的“快捷方式”，这对组织和管理文件系统中的目录结构造成了一定困难
- **删除困难**：硬链接和原始文件共享同一个 inode，因此删除一个硬链接并不会立即释放文件的存储空间，只有当所有硬链接都被删除后，文件空间才会被回收。这种机制可能导致一些不可见的存储占用，影响系统的空间管理
- **缺乏文件别名**：硬链接只是文件的不同名称，而无法提供更直观的别名功能（例如，符号链接），这使得文件和其引用之间的关系不够清晰

在 **FtidesOS** 中，我们在 xv6 文件系统中实现了符号链接（软链接）的功能。符号链接允许用户在文件系统中创建指向其他文件或目录的“快捷方式”，类似于 Windows 系统中的快捷方式。与硬链接不同，符号链接是一个独立的文件，它存储了指向目标文件路径的字符串，而不是直接指向数据块。

4.4.1 相应数据结构

我们在实现符号链接时，对 xv6 的数据结构做了必要的扩展。特别是，我们在 inode 结构中增加了对符号链接的支持。

- **inode 结构**：在原有的 inode 结构中增加了对符号链接的支持。inode 中的 `addrs` 数组继续用于存储数据块地址，但对于符号链接文件类型 (**T_SYMLINK**)，该数组将包含指向目标路径的地址。文件类型 **T_SYMLINK** 被定义为 4，表示该 inode 是一个符号链接。
- **文件类型定义**：**T_SYMLINK**（值为 4）被用来区分符号链接和其他文件类型（如普通文件、目录等）。
- **符号链接的存储**：符号链接的 inode 类型是 **T_SYMLINK**，并在 inode 的 `addrs` 数组中存储目标路径的字符串（通常是一个指向目标文件路径的字符数组）。

4.4.2 算法和实现思路

1. **符号链接创建**：通过系统调用 `sys_symlink`，创建符号链接文件。该文件将存储指向目标文件的路径字符串，而不是直接存储数据块地址。
2. **符号链接解析**：在文件打开过程中，如果遇到符号链接，系统将根据符号链接中的路径信息递归解析，直到找到最终的目标文件。为了避免死循环，我们对解析深度进行了限制。
3. **递归限制**：在解析符号链接时，设置了一个最大递归深度 (**MAX_SYMLINK_DEPTH**)，防止出现符号链接循环指向自身的问题。

添加提示中的相关定义，**T_SYMLINK** 以及 **O_NOFOLLOW**

```

1 // fcntl.h
2 #define O_NOFOLLOW 0x004
3 // stat.h
4 #define T_SYMLINK 4

```

在 `kernel/sysfile.c` 中实现 `sys_symlink`，这里需要注意的是 `create` 返回已加锁的 `inode`，此外 `iunlockput` 既对 `inode` 解锁，还将其引用计数减 1，计数为 0 时回收此 `inode`。先从寄存器中读取参数，然后开启事务，避免提交出错；为这个符号链接新建一个 `inode`；在符号链接的 `data` 中写入被链接的文件；最后，提交事务；

```

1 uint64
2 sys_symlink(void) {
3     char target[MAXPATH], path[MAXPATH]; // 目标和路径（软链接）
4     struct inode* ip_path;
5     // 从寄存器中取出用户态传入参数目标和被创建的软链接名
6     if(argstr(0, target, MAXPATH) < 0 || argstr(1, path, MAXPATH) < 0) {
7         return -1;
8     }
9
10    begin_op();
11    // 分配一个inode结点并设置类型为软链接，create返回锁定的inode
12    ip_path = create(path, T_SYMLINK, 0, 0);
13    if(ip_path == 0) {
14        end_op();
15        return -1;
16    }
17    // 向ip_path写入target路径
18    if(writei(ip_path, 0, (uint64)target, 0, MAXPATH) < MAXPATH) {
19        iunlockput(ip_path);
20        end_op();
21        return -1;
22    }
23
24    iunlockput(ip_path);
25    end_op();
26    return 0;
27 }

```

在打开文件时，如果遇到符号链接，直接打开对应的文件。这里为了避免符号链接彼此之间互相链接，导致死循环，设置了一个访问深度（我设成了 10），如果到达该访问次数，则说明打开文件失败。每次先读取对应的 `inode`，根据其中的文件名称找到对应的 `inode`，然后继续判断该 `inode` 是否为符号链接：

```

1      uint64
2      sys_open(void)
3      {
4          ...
5          if(ip->type == T_DEVICE && (ip->major < 0 || ip->major >= NDEV)){
6              ...
7          }
8          // 处理符号链接
9          if(ip->type == T_SYMLINK && !(omode & O_NOFOLLOW)) {
10             // 若符号链接指向的仍然是符号链接，则递归的跟随它
11             // 直到找到真正指向的文件
12             // 但深度不能超过MAX_SYMLINK_DEPTH 10
13             for(int i = 0; i < 10; ++i) {
14                 // 读出符号链接指向的路径
15                 if(readi(ip, 0, (uint64)path, 0, MAXPATH) != MAXPATH) {
16                     iunlockput(ip);
17                     end_op();
18                     return -1;
19                 }
20                 iunlockput(ip);
21                 ip = namei(path); // 输入路径名返回路径对应的inode
22                 if(ip == 0) {
23                     end_op();
24                     return -1;
25                 }
26                 ilock(ip);
27                 if(ip->type != T_SYMLINK)
28                     break;
29             }
30             // 超过最大允许深度后仍然为符号链接，则返回错误
31             if(ip->type == T_SYMLINK) {
32                 iunlockput(ip);
33                 end_op();
34                 return -1;
35             }
36         }
37
38         if((f = filealloc()) == 0 || (fd = fdalloc(f)) < 0){
39             ...
40         }
41         ...
42         return fd;
43     }

```

图示说明

以下是符号链接的工作原理的图示：

文件系统示意图：

符号链接 inode (symlink) -> [T_SYMLINK]

目标路径: "/path/to/target_file"

符号链接与目标文件的关系：

用户请求打开 "symlink" 文件：

1. 打开 "symlink" 文件，它是一个符号链接。
2. 读取符号链接存储的路径 "/path/to/target_file"。
3. 解析路径，找到目标文件 inode。
4. 如果目标文件不是符号链接，返回目标文件的 inode。

文件系统优化测试文档链接

5 内存管理板块

xv6 的内存管理模块主要负责进程的内存分配和管理。它采用了一些简化但有效的技术来实现内存分配。

1. **页表和分页**: xv6 使用分页机制来管理内存。每个进程都有自己的页表，用来将虚拟地址映射到物理地址。页表分为多级，以便支持更大的地址空间。
2. **内存分配**: xv6 采用了基于空闲列表的内存分配策略，核心部分是 **kalloc** 和 **kfree** 函数。
 - **kalloc**: 从空闲内存列表中分配一个物理页（大小为 4096 字节）。
 - **kfree**: 释放一个物理页，并将其返回到空闲列表。

在 xv6 中，由于没有实现懒分配，即在分配内存时会立即为每个页面分配物理内存，故存在以下缺点：

1. **内存浪费**：由于每次创建或分配新的页时，xv6 会立即分配物理内存，这可能导致大量不必要的内存分配。例如，在某些情况下，进程分配了内存页，但这些内存页实际上并没有被使用，这会浪费物理内存资源。
2. **性能低下**：懒分配的缺失使得系统每次需要访问一个新的虚拟页面时，都必须提前分配物理内存，这可能导致不必要的内存访问和初始化，尤其是在大文件或大数据结构的情况下。这不仅增加了内存的占用，还会影响系统的整体性能。
3. **无法处理稀疏内存需求**：如果一个进程仅使用虚拟内存的一部分区域，xv6 仍然会为整个虚拟内存区域分配物理页面。这对稀疏数据结构（如稀疏矩阵、稀疏数组等）来说是低效的，因为只有一部分内存会被使用。

4. **内存分配和释放的开销**：由于没有懒分配，xv6 必须为每个虚拟页即时分配物理内存，这增加了内存分配和释放的频率与开销。在复杂的内存操作中，这可能导致额外的 CPU 和内存管理开销。

5.1 懒分配

懒分配 (Lazy Allocation) 是一种内存管理技术，通过延迟实际的物理内存分配，直到内存真正被访问时才进行分配。通常，在进程请求内存（例如通过 `sbrk()` 系统调用）时，操作系统并不立即分配物理内存，而只是更新进程的虚拟地址空间。当进程首次访问这部分内存时，会触发**页面错误 (Page Fault)**，此时操作系统才分配物理内存并进行映射。懒分配具有以下优点：

- **提高性能**：减少初始开销，通过推迟内存分配，`sbrk()` 等系统调用可以更快速地完成，减少了内存分配的初始开销。
- **节省内存**：避免浪费，仅在内存真正被使用时才分配物理内存，避免了分配大量未使用内存的浪费情况。
- **优化资源利用**：允许操作系统更灵活地管理和分配内存资源，尤其是在内存紧张的情况下，提高了资源利用率。

懒分配技术通过延迟内存分配，优化了系统性能和内存利用率，是操作系统内存管理中的一种有效策略。



在本任务中，我们进行了以下两项内存管理的改进和扩展：

- **Eliminate allocation from `sbrk()`**：将 `sys_sbrk()` 系统调用中的内存分配改为仅在虚拟地址空间中增加地址，而不实际分配物理内存。

- **Lazy allocation:** 实现懒分配 (**Lazy Allocation**) 机制, 当进程访问未分配的虚拟内存页时, 触发缺页中断, 通过 `usertrap()` 处理程序动态分配并映射物理内存。

5.1.1 数据结构

在此任务中, 涉及到的主要数据结构包括:

1. 进程控制块 (`struct proc`)

- **sz:** 进程的虚拟地址空间大小, 表示进程的堆和栈的终点。
- **pagetable:** 每个进程的页表, 用于虚拟地址到物理地址的映射。
- **trapframe:** 保存中断或系统调用返回时的寄存器状态。

2. **页表项 (`pte_t`):** 存储虚拟地址和物理地址之间的映射关系。每个页表项包含标志位 (如有效位、读写权限、用户权限等)。

3. **内存页 (Page):** 页大小为 4KB, 懒分配机制会在进程访问一个未分配的页面时动态分配物理内存。

5.1.2 算法思路与实现

1. Eliminate allocation from `sbrk()`

在修改后的 `sys_sbrk()` 中, 当进程请求增加或减少虚拟内存时, 仅调整进程的虚拟地址空间大小, 而不实际分配物理内存。

- **内存增长:** 通过更新进程的 `sz` 字段来增加虚拟地址空间, 保持虚拟内存的连续性。
- **内存收缩:** 如果减少内存空间, 则调用 `uvmdealloc()` 释放虚拟内存范围内的物理页面。
- **地址越界检查:** 确保申请的虚拟地址空间不超过最大可用地址。

修改后的 `sys_sbrk()` 代码:

```
1      uint64
2      sys_sbrk(void)
3      {
4          myproc()->syscallCount[SYS_sbrk]++;
5          uint64 addr;
6          int n;
7          argint(0, &n);
8          struct proc *p = myproc();
9          addr = p->sz;
10         if (addr + n >= MAXVA || addr + n <= 0)
11             return addr;
12         p->sz = addr + n;
13         // addr = myproc()->sz;
14         // if (growproc(n) < 0)
15         //     return -1;
```

```

16         if(n < 0)
17             uvmdealloc(p->pagetable, addr, p->sz);
18         return addr;
19     }

```

2. **Lazy Allocation** 懒分配通过在进程访问未映射的虚拟内存时触发缺页中断来动态分配物理内存。具体步骤如下：

- **缺页中断**：当访问虚拟地址对应的物理内存未分配时，触发页错误中断（**Page Fault**），并进入 **usertrap()** 处理程序。
- **物理内存分配**：在 **usertrap()** 中，若虚拟地址有效并未超出进程地址空间范围，则分配新的物理内存页，并映射到虚拟地址空间。
- **内存映射**：通过 **mappages()** 将新的物理页映射到进程的虚拟地址空间中。
- **释放内存**：在内存收缩时（如通过 **sbrk()** 函数减少虚拟内存），需要释放未被访问的物理页面。

usertrap() 中的懒分配代码：

```

1     if(r_scause() == 13 || r_scause() == 15) { // 检查是否为缺页异常
2         uint64 va = r_stval(); // 获取虚拟地址
3         uint64 pa = (uint64)kalloc(); // 分配物理内存
4         if (pa == 0) {
5             p->killed = 1;
6         } else if (va >= p->sz || va <= PGROUNDDOWN(p->trapframe->sp)) {
7             kfree((void*)pa);
8             p->killed = 1;
9         } else {
10            va = PGROUNDDOWN(va);
11            memset((void*)pa, 0, PGSIZE); // 初始化物理内存
12            if (mappages(p->pagetable, va, PGSIZE, pa, PTE_W | PTE_U | PTE_R) != 0) {
13                kfree((void*)pa);
14                p->killed = 1;
15            }
16        }
17    }

```

懒分配的流程：

- 当进程访问虚拟地址时，**walkaddr()** 会检查该地址是否已经映射到物理内存。如果没有映射，则触发懒分配。
- 通过调用 **kalloc()** 分配新的物理内存页，并使用 **mappages()** 将其映射到虚拟地址空间中。
- 如果物理内存分配失败，进程会被杀死。

walkaddr() 的懒分配代码:

```
1      uint64 walkaddr(pagetable_t pagetable, uint64 va) {
2          pte_t *pte;
3          uint64 pa;
4          struct proc *p = myproc(); // 获取进程
5
6          if(va >= MAXVA)
7              return 0;
8
9          pte = walk(pagetable, va, 0); // 获取页表项
10         if(pte == 0 || (*pte & PTE_V) == 0) { // 未映射
11             if(va >= PGROUNDUP(p->trapframe->sp) && va < p->sz) {
12                 char *pa;
13                 if ((pa = kalloc()) == 0) { // 分配物理内存
14                     return 0;
15                 }
16                 memset(pa, 0, PGSIZE); // 初始化内存
17                 if (mappages(p->pagetable, PGROUNDDOWN(va), PGSIZE, (uint64) pa,
18                     PTE_W | PTE_R | PTE_U) != 0) {
19                     kfree(pa);
20                     return 0;
21                 }
22             } else {
23                 return 0;
24             }
25             if((*pte & PTE_U) == 0)
26                 return 0;
27             pa = PTE2PA(*pte);
28             return pa;
29         }
```

3. Fork 的懒分配支持

fork() 函数需要考虑父进程的虚拟内存空间向子进程的拷贝, 在此过程中, 懒分配会确保仅拷贝已经访问的虚拟页面, 而不是预先分配所有物理内存。

修改后的 **uvmcopy()**:

```
1      if((pte = walk(old, i, 0)) == 0)
2          continue; // 跳过未分配的页面
3      if((*pte & PTE_V) == 0)
4          continue; // 跳过无效的页表项
```

4. 内存释放处理

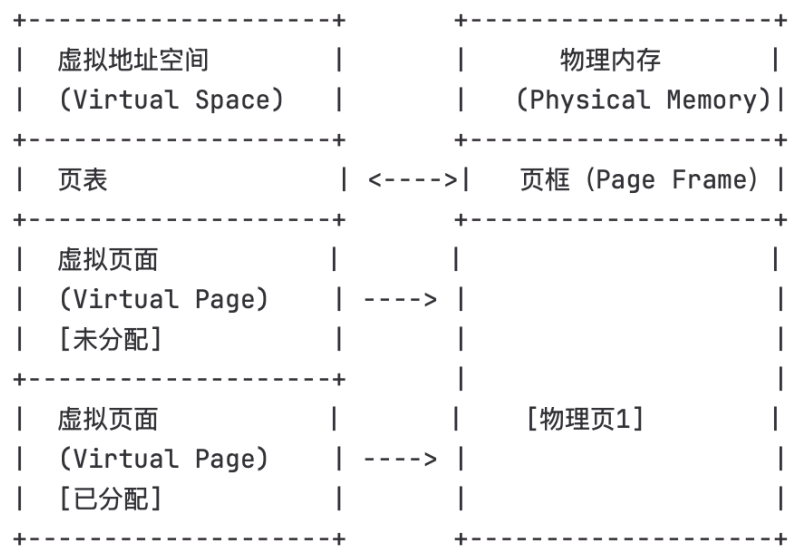
当进程的虚拟内存被缩小时, **uvmunmap()** 函数需要适应懒分配机制, 跳过未分配的虚拟页面。

修改后的 `uvmunmap()`

```
1     if((pte = walk(pagetable, a, 0)) == 0)
2         continue; // 跳过没有映射的虚拟地址
3     if((*pte & PTE_V) == 0)
4         continue; // 跳过无效的页表项
```

图示说明

以下是懒分配的示意图：



关于内存管理的测试文档链接

6 内核测试文档链接

在项目过程中在每一个板块的末尾都提供了链接，但是为了方便评委老师们查阅：在最后设置了一个板块统一提供测试文档的跳转链接

1. 可以点击[这里](#)跳转到 `getcpuid` 系统调用的测试文档查看详细的测试过程
2. 可以点击[这里](#)跳转到 `getSysCount` 系统调用的测试文档查看详细的测试过程
3. 可以点击[这里](#)查看 `countproc` 系统调用的测试文档
4. 可以点击[这里](#)查看 `sigalarm` 和 `sigreturn` 系统调用的测试文档
5. 可以点击[这里](#)查看 `sys_symlink` 系统调用的测试文档
6. 进程调度优化测试可以点击[这个链接](#)进行查看

- 7. [文件系统优化测试文档链接](#)
- 8. [关于内存管理的测试文档链接](#)

7 总结和未来计划

在本项目中，我们基于 xv6 操作系统进行了多方面的优化和升级，实现了下列举措：

- **增加了新系统调用：**
- **实现了较高效的进程调度算法：**能够替换 xv6 的 RR 算法，实现了多级反馈队列算法和彩票算法
- **文件系统完善：**实现了二级索引以及符号链接，优化了文件结构
- **内存管理和分配：**通过实现懒分配和 `sys_sbrk` 系统调用的优化，改进了内存分配、释放和规模管理。这不仅减少了内存消耗，还提高了运行效率和系统稳定性。

未来，我们计划对以下方面进行更多优化和升级：

1. **多核支持：**增强多核调度和文件分布。通过实现更高效的多核资源共享，提高系统性能和资源利用率。
2. **文件系统的分类与扩展：**增加根据文件类型和内容分类的支持，提升文件管理的便捷性和系统资源的优化使用。
3. **内存性能优化和监控：**集成全面的内存调试工具，优化内存管理的性能视图；利用基础资源和模式分析，提升内存使用效率。
4. **图形用户界面（GUI）支持与操作优化：**提供轻量化的图形界面和交互支持，提升用户操作的便捷性和系统适应性。

我们计划在实现更多功能和优化的同时，重点关注系统设计的简洁性和精简性，确保系统性能与开发效率的平衡。