OSLab 实验报告

谢逸 171240536

2019 年 6 月 7 日

1 L1 部分

我完成了所有必做内容!

1.1 算法及实现

1.1.1 数据结构

我在每一个所用内存块的开头, 都加上了一个数据结构 block, 记录了这个内存块的信息. block 中有两个指针, 分别指向下一个和上一个 block, 构成链表; 此外还记录了 block 的起始位置, 是否空闲, 内存大小, 还有一个 magic number 来判断是否是一个 block 结构.

1.1.2 小内存的分配

我规定小于 1KB 的都是小内存. 小内存的分配尽量实现各处理器的独立, 所以我维护了 4 个链表, 每个都记录了当前处理器的所有直接可用和已经使用的内存 (由 free 变量来区分). 每次 alloc 一个小内存, 先遍历该处理器的链表, 寻找第一个大小足够且空闲的内存块. 如果没有这样的内存块, 就需要申请一块新的内存块 (我设置的分配 4KB), 从 heap 中低地址出申请内存. 对于找到或者新分配的内存块, 如果其大小比所需大小大得较多, 那就将这个内存块分为两个, 使用其中那个大小正好的.

1.1.3 大内存的分配

和小内存不同的是, 所有处理器大内存的分配共用了一个链表. 当链表中现有内存没有合适的时候, 从 heap 中高地址处申请内存 (和小内存就可以分开).

1.1.4 小内存的释放

释放某个内存的时候, 检查其 next 和 prev 内存块是否空闲且地址相邻, 如果空闲, 地址相邻且内存之和不太大, 那就合并为一个新的空闲内存块.

1.1.5 大内存的释放

类似与小内存.

1.1.6 锁的使用

大内存因为不同处理器共用一个链表, 所以需要上锁. 小内存因为不同的处理器使用不同的链表, 所以只有链表没有足够大的空闲内存块, 需要申请新的内存块的时候才需要上锁. 可以实现小内存分配的无锁化. 虽然我已经调好了我发现的因为小内存不上锁而引起的并发 bug, 也通过了我的测试, 但是由于不知道助教会使用什么神仙测试, 所以我在交之前还是上了锁.

1.1.7 测试

首先我尝试了每次分配内存立即释放, 和分配一堆内存最后释放. 然后参考了 yzy 同学的测试, 先 alloc100 次, 然后 free 且 alloc 同一个数组 1000 次, 然后 free 所有数组.

1.2 遇到的问题

测试中的数组如果开为局部数组并且大一点的话有可能会出现 qemu 窗口不时抖动的情况. 去掉一些 printf 后消失, 但是还是有各种不稳定的奇怪问题. 最终将数组改为全局的, 之后一切正常.

1.3 致谢

感谢欧助教帮我分析局部数组的玄学 bug, 也感谢 yzy 提供测试思路, 还感谢 zly 和我一起讨论局部数组的 bug.

2 L2 部分

我完成了所有必做内容, 通过了二核和四核的打印字符测试, 多生产者多消费者测试, echo task 测试, 暂时未发现 bug.

2.1 算法及实现

2.1.1 锁

几乎照搬了 xv6 的锁, 不得不说这个锁的很多 assert 帮助了我非常多.

2.1.2 信号量

信号量中比较重要的是维护了每个信号量的等待队列, 以及每个进程的状态 (见下一部分). wait 在信号量值为 0 的时候进入等待队列, 标记为不可调度然后 yield. signal 则是唤醒一个等待队列中的进程 (改标记).

2.1.3 进程调度

主要是维护了每个进程的状态: ready, waiting, running, wake-up(这种状态的存在意义见遇到的 bug), 每次从当前进程开始遍历寻找可调度进程.

2.1.4 os 模块

在 event 的注册时就是用数组维护了一个事件序列 (按 seq 序). 每次 trap, 遍历一遍事件序列, 碰到恰当的事件就调用对应的处理程序. 参考了 jyy 的实例代码, 将现场保存和进程切换看做事件, 这是一个非常棒的思路.

2.2 遇到的问题以及解决方案

1. 进程调度时, 每个 cpu 第一次切换进程时, 因为原来的进程 (除了 cpu0 都没有进程) 并没有被保存下来, 而我记录每个 cpu 当前跑的进程的数组仍为空, 所以遍历进程不能从当前进程开始. 起初因为空指针访问没有报错, 导致我遍历的起始进程号为负的, 当我在进程数组中遍历的时候, 取模结果也是负的, 就访问了 Task[-x], 导致可能出现到处乱切的情况.
   解决方法: 将每个 cpu 当前跑的进程的数组全初始化为 NULL, 在调

度的时候, 如果当前 cpu 所跑进程为 NULL, 那么特判后特定一个合法的遍历起始进程号.

2. 我遇到的最可怕的 bug: 信号量 sem wait 中, 在 yield 之前需要解锁, 解锁后可能会切换到别的进程, 如果切换到的进程正好调用 sem signal 唤醒的原来的进程, 就会导致 yield 之前进程的状态就是 ready, 触发我的 assert. 更可怕的是, 如果 yield 以后另一个 cpu 跑这个进程, 我记录当前 cpu 所跑进程的数组中会有两个都是记录的这个进程, 导致接下来的进程调度全乱.
   解决方法: 董杨静同学提议可以限定 wait 后醒来的进程只能在原来的 cpu 上跑, 我觉得这样有点浪费并发性, 所以加上了一个进程刚被唤醒的状态, 调度碰到这种进程, 如果我记录的 cpu 正在跑的进程数组中有这个进程, 那么只能原来的那个 cpu 跑, 否则随意.

3. 碰到了只有在 tcg 下有 while(1) _yield(); 进程的情况下出现多个 cpu 跑同一个进程的 bug. de 了很久找不到问题, 和李顶为讨论后, 决定直接改为不准跨核调度. 虽然变慢了, 但是 bug 的确没了.

2.3 致谢

1. 感谢李顶为同学碰到和我一样的问题 3 之后一起讨论很久 (虽然没找到问题, 换了实现方式)
2. 感谢张灵毓同学和我一起讨论了信号量的实现方法
3. 感谢董杨静同学提供了解决上面遇到的问题 2 的思路
4. 非常感谢 oxf 助教教会我用 gdb 调试 qemu 的方法, gdb 真是太强大了, 没有 gdb 我根本找不到 bug.

3 L3

注：之前因为 L1 出 bug 所以有更改 L1 实现 (基本上就是去掉了大内存的实现)!!! L1 的实验报告我并没有进行对应的修改。

请结合使用手册和下面的 shell 注意事项一起使用。

3.1 shell 使用手册

1. cd: cd 一个路径，可以改变 PWD。正常则会输出 PWD。一个合法的路径是这样的：./xxx(表示当前目录下的某个路径), ../xxx(表示上一级目录下的某个路径), /xxx(绝对路径). 不支持不以/或者./或者../开头的路径，不支持多个./或者../ 的叠加。下面的路径都是这样定义的。合法实例：cd ./Documents, cd /Documents, cd ../Documents

2. ls path: 同 linux。ls 无参数即路径为 PWD。path 支持相对路径，同 cd。

3. mkdir path：创建路径为 path 的文件夹。如果文件夹已经存在，那么什么都不做。同一目录下不支持和文件或者其他文件夹同名的文件夹。

4. rmdir：删除文件夹。同 linux，不支持删除非空的文件夹!!! 请先自行删除文件夹内文件。

5. touch：创建文件。如果文件已经存在，那么什么都不做。同一目录下不支持和文件或者其他文件夹同名的文件。6. cat path: 打开文件，读取前 100 字节并输出，然后关闭。读到不可显示字符或者文件结尾可能会使输出结束。

7. open path: 打开文件，同 linux 的 open 函数，返回 fd，打开后才能读写。

8. close fd: 同 linux 的 close 函数，关闭 fd 指向的文件，不再能读写。

9. read fd nbyte: 同 linux 的 read 函数，从文件的 offset 开始读取 nbyte 字节并且在中断输出。到达文件末尾或者不可显示字符会停止。

10. write fd str nbyte: 从文件的 offset 开始写入 str 的前 nbyte 字节。请保证 nbyte 小于等于 str 的长度。str 不能带空格。

11. lseek fd nbyte whence：改变文件的 offset。同 linux。whence 有三种，SEEK_CUR, SEEK_SET, SEEK_END.

12. access path：检查是否有这个路径的文件 (不支持对文件夹判断)。

13. link oldpath newpath: 同 linux 的 link。需要保证 oldpath 的文件存在，newpath 的文件不存在且上一级文件夹存在。link 的新旧文件共享 inode，因此对于一个的操作对于另一个也有效。link 对于文件夹无效 (在 linux 这是需要 sudo 的，既然没实现权限那就不允许吧)。

14. unlink path：删除文件。只要路径存在，就会在上一级目录中删除这个

文件的索引。当一个文件链接数为 1 且不被任何线程打开的时候，真正删除文件。(所以打开着的文件 unlink 了还能用)。
15. unmount path：只能 unmount /dev 或者 unmount /proc。不支持 unmount 根目录，因为 unmount 根目录之后 blkfs 就没了太危险了。例如 unmount /dev 后，/dev 文件夹就为空，之前打开的设备还能使用，但是不能再打开设备。/dev 文件夹留着，并且不像 blkfs 中的一样可以进行各种操作，只是留作以后 mount 用 (但是因为没有创建 filesystem 的函数，也没法再测试了，同理没有提供 mount 的 shell 命令，虽然实现了相关函数。)

3.2 注意事项

3.2.1 shell

1. 我的 shell 中每条命令的输出至多 128 字节，因此超过 100 字节的读被砍为了 100 字节 (主要针对 read)
2. 我的 shell 中每条命令的输入至多 128 字节，因此请不要输入超过 100 字节的 command(尤其是写入操作)
3. shell 中的命令和参数都是以单个空格隔开的，所以不支持参数中带有空格 (比如 write 的 str)，不支持多余的空格，不支持命令结尾是空格!
4. 请不要在一条命令没有执行完的时候就输入，否则可能输入无效。

3.2.2 blkfs

1. 文件大小至多 4000 字节
2. inode 总和不能超过 4000B，包括已经删除的
3. 每个文件夹中至多 64 个文件或文件夹, 包括已经删除的
4. 每个文件名不得超过 32 字节，路径不得超过 64 字节，文件名不支持空格，不支持/
5. 一个线程最好不要 open 多次同一个文件，如果这样做了，close 也需要同样的次数。
6. 不要 touch 一个已经存在的文件, 可能会导致引用计数错乱。
7. 不要对一个没有打开的 fd 进行操作 (后来我加了特判了，应该做了只是会失败而已)。
8. 不支持重定向、管道等，想要写文件就还是 open lseek write close 吧
9. 不要在/dev，/proc 下做不支持的操作，/dev 就只是设备系统，/proc 下

就是只是 proc 信息
10. /Documents/README 中有一个事先存好的话。

3.2.3 devfs

1. 设备列表：/dev/ramdisk0, /dev/ramdisk1, /dev/tty1 到/dev/tty4
2. tty 支持的操作有 read，cat，write, 而 ramdisk 支持的操作有 read，cat，write，除了 SEEK_END 的 lseek。
3. 尽量不要对于磁盘 1 进行操作，因为 blkfs 在这里面，如果乱写后果很严重。推荐用磁盘 0 进行测试，其中初始时就有连续的数字，适合用来读写.
4. 支持一个 tty read 或者 write 另一个打开了的 tty，但是顺序上可能会有小的错乱。例如 tty1 通过 read 读 tty2 的时候，tty2 第一次输入不会传到 tty1，第二次才会。这是因为 tty2 的 read 早就被调用了。
5. **绝对不要输错 dev 的名称!!! 绝对不要输错 dev 的名称!!! 绝对不要输错 dev 的名称!!!** 因为 jyy 的代码中的 dev_lookup 居然在没找到对应的设备的时候直接 panic("lookup device failed") 了，虽然完全不知道为什么要这样写，而且改起来也很方便，但是出于尊重我并没有擅自改他的代码。所以如果输错了 dev 的路径而 panic 了，这真的不是我的锅 (手动滑稽)。

3.2.4 procfs

1. 文件有/proc/0 到/proc/24, /proc/meminfo, /proc/cpuinfo.
2. 支持 cat，open，read，close，不支持 write，也不支持 lseek，不存 offset，每次读取的 offset 都是 0.
3. 对于/proc/x, 里面有 x 号进程的信息：task id，task name，task status. 如果这个 id 的 task 不存在，那么文件为空.
4. 对于/proc/cpuinfo，只有一个信息，就是 cpu 数量
5. 对于/proc/meminfo，里面有 pm_start,pm_end, 堆区总内存，已用内存，剩余空闲内存.

3.3 遇到的问题

个人认为解耦出 inode 反而给代码实现增加了难度，因此我并没有使用 inode 的那一套 API，在 vfs 调用 fsops 中的函数实现的。