OS-lab8 report

刘泓尊 2018011446 计84

Chapter8 实现的内容

1. 在`mm`模块增加了获取剩余可分配物理页面数的接口，在可用内存不够时，终止一些需要分配新内存的syscall服务，如mmap, fork等
2. 增加了`sys_read`，`sys_write`，`sys_oepnat`，`sys_pipe`，`sys_{un}linkat`，`sys_fstat`等存在指针参数的syscall的指针地址合法性检查，避免程序读写非法位置。
3. 修正了`waitpid` syscall的bug, 使得该进程的子进程结束后依然由父进程管理（而不是INITPROC），父进程的waitpid调用不至于崩溃。
4. 可以通过`ch8_01`，`ch8_02`，`ch8_03`，`ch8_04`，`ch8_05`，`ch8_06`，`ch8_07`的测试。

编程作业

`ch8_01`

该测试程序总共会执行`fork` 1023次，会fork出大量相同的进程，如果采用教程中的实现，每次fork都会分配新的物理页面，这样会导致OS可分配内存急剧消耗完，从而导致OS崩溃或者无法响应。

解决办法：

- 实现fork的COW(Copy on Write)机制, 只有在子进程实际发生写入的时候再分配新的物理页面；这就需要OS对父子进程的地址空间和访问行为进行跟踪。但是这不能从根本上解决fork炸弹的问题，因为攻击者可以在每次fork后再写入一块区域来触发新的内存分配。
- 限制一个用户或一个OS可以并发运行的进程数量。这样用户就无法fork出大量的进程来消耗OS内存资源。但是也缺乏一定灵活性。在我的Ubuntu20.04系统上，每个用户最大进程数为`24958`, 通过执行`ulimit -a`可以获得该信息。
- 在可用物理内存不足时杀死申请新内存的进程，申请新内存的操作包括fork, spawn, mmap等。我采用了这一点进行改进，在 可用物理页面数 小于 当前进程占用页面数+INITPROC占用页面数 的时候，杀死该进程。引入INITPROC占用页面数是为了保证进程切换时有一定的可用内存，以避免OS崩溃。代码如下：

```
if usable_frames() < current_task.frames_used() + INITPROC.frames_used()
{
    exit_current_and_run_next(-1);
}
```

`ch8_02`

该程序使用`mmap`申请了大量的内存，总共申请`2048 * 65536 =  128MB`的内存，而我们的OS总共才有8MB，所以远远不够吗，可能会导致OS内存不足而崩溃。解决办法：

- 增加lazy的内存分配机制，在申请的时候并不实际分配物理内存，而推迟到实际访问的时候再分配。这样的话该程序一直执行`mmap`并不会消耗物理内存，避免OS崩溃。

- 增加swap机制，实现页替换算法，在需要新的物理内存时，将某些物理页面保存到磁盘，以解决内存紧张。
- 在可用物理内存不足时杀死申请新内存的进程。我采用了这一点的改进，在 可用物理页面数 小于 申请物理页面数+INITPROC占用页面数 的时候，杀死该进程。代码如下：

```
// task/mod.rs, map_virtual_pages(addr: usize, len: usize, port: usize)
-> isize
if usable_frames() < len / 4096 + INITPROC.frames_used() { // 可用内存不足
    return -1;
} else {
    ...
}
```

ch8_03

该程序执行 `sys_gettime` 时通过 `get_pc()` 访问了代码段 `.text` ，在执行 `sys_fstat` 时通过 `get_pc()` 访问了代码段 `.text` ,在执行 `sys_read` 时通过 `get_pc()` 访问了代码段 `.text` ;在执行 `sys_fstat` 是通过 `TRAP_CONTEXT` 访问了跳板代码 `.text.trampoline` 。而这几个syscall都涉及了对传入缓冲区的写入操作，所以都属于访问代码段 `.text` 时引起的权限错误。解决办法：

- 在 `sys_fstat` ， `sys_read` , `sys_gettime` 增加访问权限检查，权限不足杀死该进程，报告段错误(Segment Fault)。比如Linux中用户程序写0地址通常会触发该错误而终止。
- 在 `sys_fstat` ， `sys_read` , `sys_gettime` 增加访问权限检查，权限不足返回-1。在将虚拟地址翻译到物理地址时，检测该段是不是可写 `W` (代码段的权限只有 `RX`), 如果权限不足则返回失败。我采用了这种办法，我使用lab4中补充 `sys_write` 时的检查权限的函数 `virtual_addr_writable(token: usize, va: usize)->bool` , 主要了逻辑如下：

```
// mm/page_table.rs, virtual_addr_writable()
let va = VirtAddr::from(va);
let page_table = PageTable::from_token(token);
if let Some(pte) = page_table.translate_pte(va) {
    pte.is_valid() && pte.readable() && pte.writable()
} else {
    false
}
```

然后在对应syscall中加入如下代码, 权限不足返回-1即可

```
// syscall/fs.rs and process.rs, in sys_fstat, sys_read, sys_get_time
if !virtual_addr_writable(token, st as usize) {
    return -1 as isize;
} else {
    ...
}
```

ch8_04

该程序首先通过 `read` 读了 `stdout` 文件，但是这个文件是不可读的，所以会read败。之后 `write` 写了 `fd` 是65537的文件，这时该进程并没有该文件描述符，所以write失败。之后又通过 `read` 写了 `fd = 13513543` 的文件，这个文件也不存在，所以read失败。

之后通过 `close` 关闭了 `fd = 233` 的文件，因文件不存在而失败。之后通过 `close` 关闭了 `stdin`，`stdout` 和 `stderr`，均关闭成功，在这之后再次 `println` 就无法输出了。

执行 `set_priority` 设置优先级，因为 `-7` 不满足 `prio \in [0, isize::MAX]` 的条件而失败，但 `isize::MAX` 是成功的。

之后执行 `mail_write`，向100000进程的邮箱写入，因该进程不存在而失败；向133进程邮箱写入 `bug` 缓冲区，该进程如果存在就可以写入成功。之后向0号进程(INITPROC)的邮箱写入 `0x1ff0` 缓冲区对应的数据，可以预见这个地址是invalid的，所以发生权限错误而失败。

调用 `link`，因为 `nonono` 文件不存在而失败。之后 `link("fname0", "fname1");` `link("fname1", "fname0"); link("fname0", "fname0");` 三次执行都因为新的硬链接名字已经存在而失败。 `link("\0", "fname1");` 因文件名 `\0` 非法而失败（会被OS作为空串）。

之后调用 `sys_fstat` 试图写入0地址，显然会因为该地址非法而失败。调用 `sys_unlinkat(555, "❽❽❽❽❽❽", 1)` 会因为 `path ="❽❽❽❽❽❽"` 的不存在而失败；执行 `sys_linkat(0, "QAQ", 7, "❄❄❄❄❄❄", 0)` 也会因为 `QAQ` 和 `❄❄❄❄❄❄` 都不存在而失败（注意，之前测试框架保证 `dirfd =AT_FDCWD (-100)`，所以我忽略了对 `dirfd` 的检查）。

综合以上分析，我做出的修改如下：

- `read/write` 的时候如果对应 `fd` 为空，则返回-1表示失败，不进行任何操作。需要注意的是，因为测试框架引入了输出缓冲区 `ConsoleBuffer`，我发现如果println遇到stdout关闭的情形，返回-1会导致程序提前结束而不是忽略该调用。我还没有发现异常结束的原因，所以作为一个折衷，当stdout关闭的时候执行write, 会返回0，但依然什么都不做。
- 增加 `sys_read`，`sys_write`，`sys_fstat` 对读写缓冲区地址的权限检查，这个已经在解决 `ch8_03` 的时候得到解决。
- `mail_write` 的目标进程如果不存在，则返回-1，什么都不做。
- `link` 和 `unlink` 时检查源文件和目标文件路径是否合法且存在，合法且存在的情况下才实际执行操作。（当然也可以再加上对 `dirfd == -100` 的检查，我的实现是忽略它的）

ch8_05

调用了 `forktest`，这个函数主要是在循环中调用fork, 之后每次子进程执行函数 `func`，最后父进程会等待所有子进程结束，所以系统中会同时跑很多个进程，直到最初的进程结束。本程序的 `func` 是 `sleep` 一段时间，所以执行时间会比较长。而fork的进程数多也会引起内存资源耗尽的问题。

这个问题可以**通过 `ch8_01` 的解决方案解决**，我采用的是直接杀死进程，在此不再赘述。对于fork失败后被杀死的情形，子进程会被杀死并返回-1，父进程的 `children` 序列会一直保存子进程的返回值信息，直到父进程被杀死，这样父进程调用 `waitpid` 的时候不会出错，保证了正确性。

ch8_06

该程序一开始先申请了 `64KB` 的内存，之后同样调用了 `forktest`，此时的 `func` 执行的是写入或读取之前分配的内存的某个字节。因为进行了大量的 `fork`，所以之前申请的空间也被复制了很多份，同样存在内存资源耗尽的问题。

这个问题依然可以**通过 `ch8_01` 的解决方案解决**：在 `mmap` 或 `fork` 的时候如果可用内存不够就杀死该进程或返回失败。

- 如果在失败时杀死进程，后续不会出任何问题；我采用的是这种实现。
- 如果 `mmap` 在内存不足时返回-1（而不是杀死进程），那么后续对该段内存的读写地址在页表中就不存在映射(invalid), 导致读写异常（因为是直接裸指针操作，所以硬件会检测到异常，属于 `StorePageFault` 和 `LoadPageFault` ），之后进入OS异常处理程序并杀死该进程。

`ch8_07`

该程序创建了大量新文件，并随后执行 `unlink` ，因为每个文件的硬链接只有1个，所以 `unlink` 应该会删除该文件。如果不实现删除文件的逻辑，这个程序会在我们的OS上会消耗大量文件资源（毕竟申请创建了65536个文件），这显然小于 `fs.img` 的大小。而实现了删除逻辑之后，就可以通过 `unlink` 在文件层上实际删除该文件，OS并不会崩溃，只是执行时间比较长。另一方面，经过 `hash` 操作之后，创建的文件名有很多不合法的情况，比如带斜杠的 `\\\` 和 `///` ,以及全部为空格 的文件名等。这些文件名会给OS和用户都带来困扰，所以OS应该避免响应这些不合理的请求。解决办法：

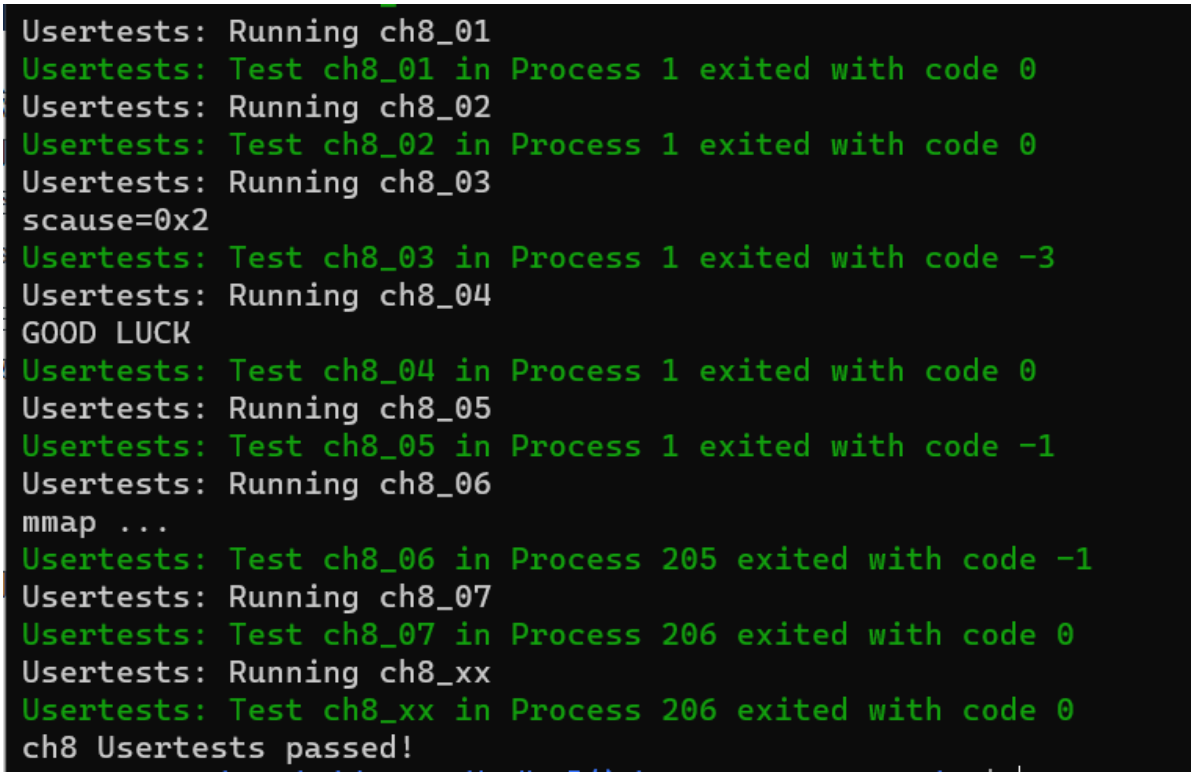
- check文件名是否合法，不合法则创建失败。我添加了该逻辑，在涉及文件名的syscall中，会先检查文件名的合法性，标准是第一个字母属于ASCII的 `0~z` ,后续字母属于ASCII可见字符且不是 `\` 和 `/` . 如果不合法就返回-1。判断的函数如下:

```rust
pub fn valid_file_name(path: &String) -> bool {
    if path.len() == 0 { return false; }
    // not include ! " #  $ % & ' ( ) * + - , - . /
    if (path.as_bytes()[0] >= 32 && path.as_bytes()[0] <= 47) ||
path.as_bytes()[0] > 122 {
        return false;
    }
    for ch in path.bytes() { // 不包含 '/' '\'
        if ch < 32 || ch > 126 || ch == 47 || ch == 92 {
            return false;
        }
    }
    return true;
}
```

- 在硬链接数 == 0的时候在磁盘中删除这个文件，避免可用磁盘资源耗尽。因为 `easy-fs` 原来不支持删除文件，所以修改比较繁琐。可以在删除的时候把对应data和目录项的bitmap清零，然后在下次分配inode的时候采用最先匹配，覆盖之前空出来的区域。也可以简单一些，在删除的时候直接将最后一个目录项换到被删除的位置，同时缩小该inode的size.
- 如果不删除文件，也可以在每次创建文件的时候遍历所有文件的硬链接，如果存在硬链接为0的情况，就直接在对应区域上覆盖，并修改对应的文件名，这样也可以避免磁盘空间耗尽的问题。

测试结果截图

经过测试，以上解决办法基本可以解决 `ch8_01~ch8_07` 的问题, 防止了OS的提前崩溃。程序运行结果如下:


Usertests: Running ch8_01
Usertests: Test ch8_01 in Process 1 exited with code 0
Usertests: Running ch8_02
Usertests: Test ch8_02 in Process 1 exited with code 0
Usertests: Running ch8_03
scause=0x2
Usertests: Test ch8_03 in Process 1 exited with code -3
Usertests: Running ch8_04
GOOD LUCK
Usertests: Test ch8_04 in Process 1 exited with code 0
Usertests: Running ch8_05
Usertests: Test ch8_05 in Process 1 exited with code -1
Usertests: Running ch8_06
mmap ...
Usertests: Test ch8_06 in Process 205 exited with code -1
Usertests: Running ch8_07
Usertests: Test ch8_07 in Process 206 exited with code 0
Usertests: Running ch8_xx
Usertests: Test ch8_xx in Process 206 exited with code 0
ch8 Usertests passed!