****

****

**Pintos Project 2 设计方案**

**目录**

1. 工作总结

1.1 小组成员信息

1.2 工作总结

1.3 GitHub工作截图

1.4 测试点情况

1. 需求分析

2.1 Process Termination Messages

2.2 Argument Passing

2.3 System Calls

2.4 Denying Writes to Executables

1. 设计方案

3.1 Process Termination Messages

3.1.1 原理

3.1.2 具体实现

3.2 Argument Passing

3.2.1 原理

3.2.2 具体实现

3.2.3 初步测试

3.3 System Calls

3.3.1 原理

3.3.2 具体实现

3.4 Denying Writes to Executables

1. 工作总结

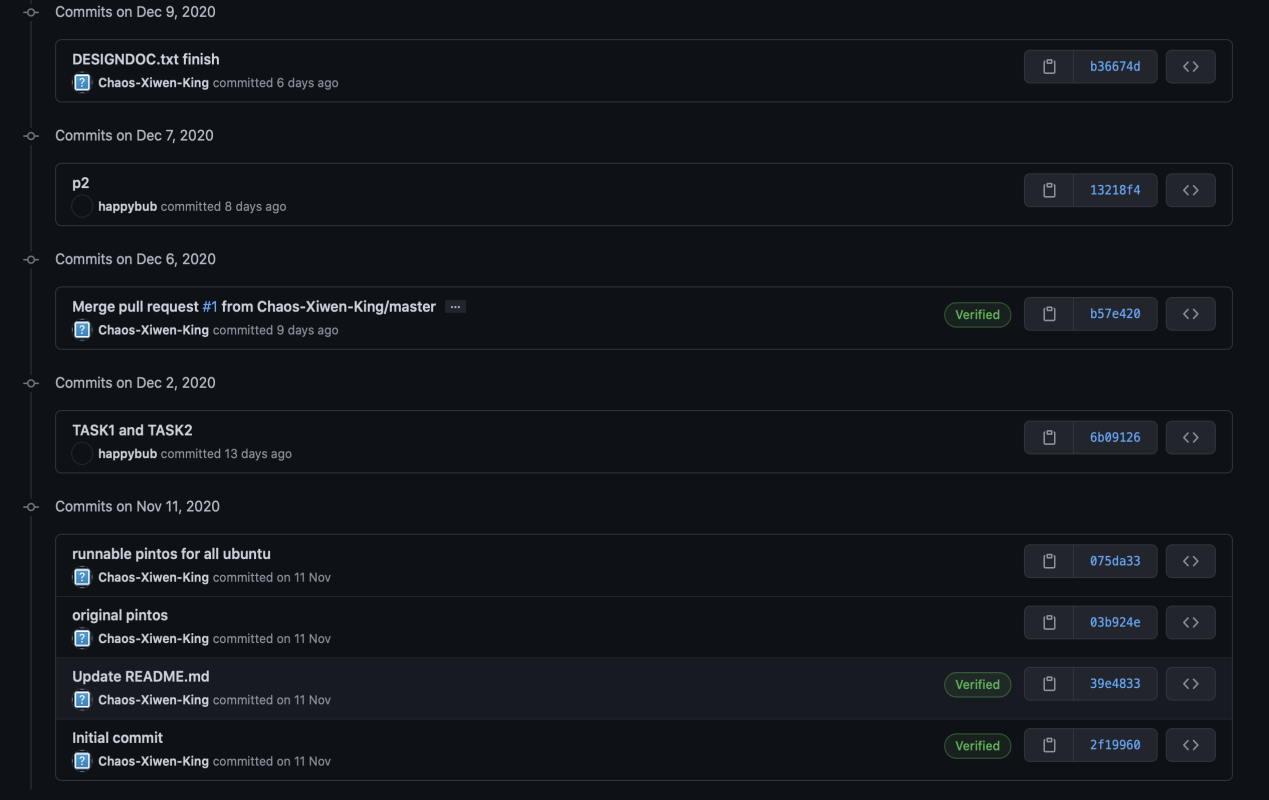
1.1 小组成员信息

|  |  |  |
| --- | --- | --- |
| 姓名 | 学号 | 贡献 |
| 康曦文 | 18373607 | 1.15 |
| 郭凌宇 | 18373638 | 1.15 |
| 陈宇畅 | 18373609 | 0.85 |
| 赵致远 | 18373669 | 0.85 |

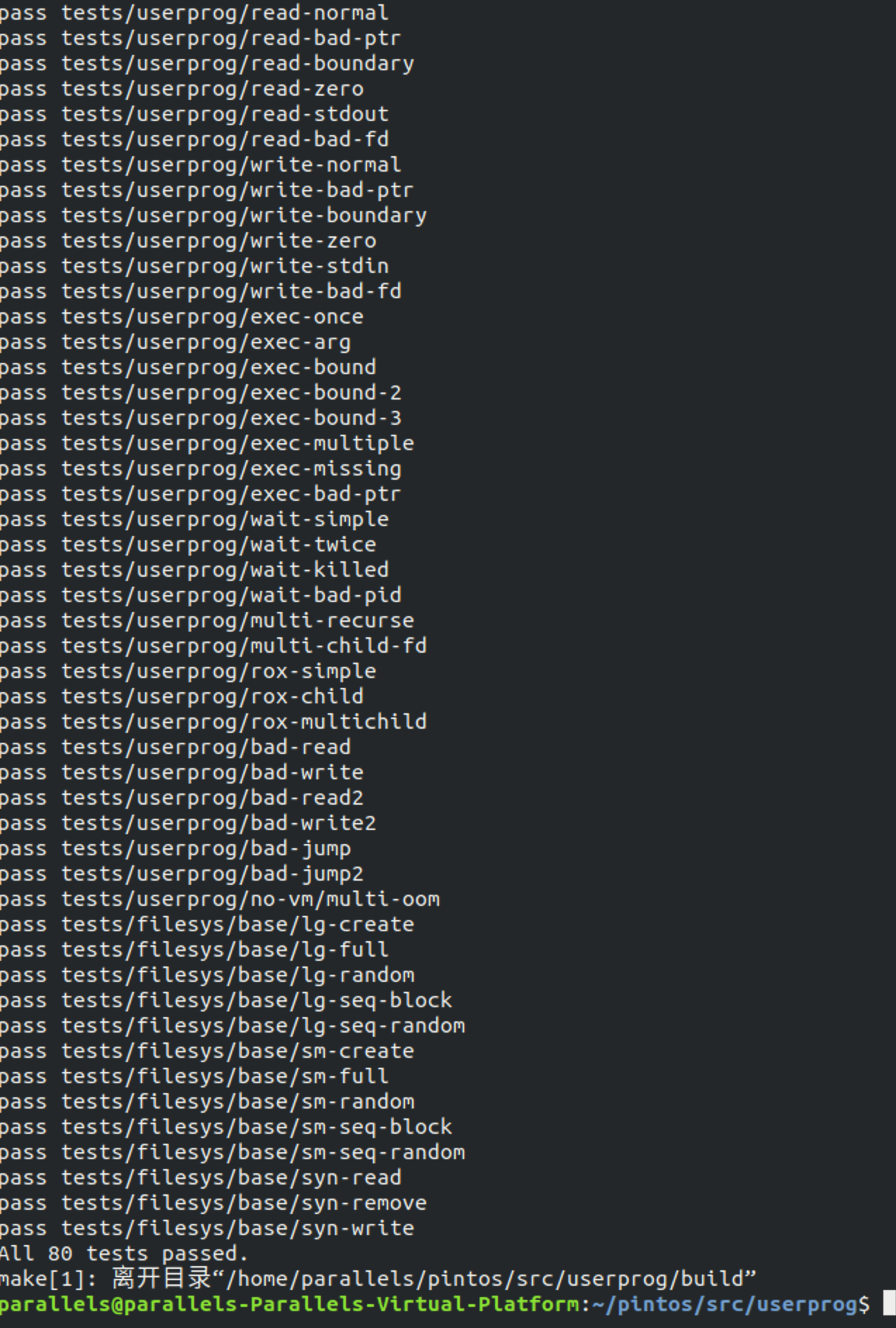
1.2 工作总结

|  |  |
| --- | --- |
| 姓名 | 工作内容 |
| 康曦文 | Task 3部分系统调用实现、Projec2 DESIGNDOC、Project2 设计文档 |
| 郭凌宇 | Task1~4 主要代码实现、Project2 设计文档 |
| 陈宇畅 | 搜集资料 |
| 赵致远 | 搜集资料 |

1.3 GitHub工作截图



1.4 测试点情况



二、需求分析

2.1 Process Termination Messages

任务的要求是在进程（线程）终止的时候，在终端打印一条结束信息。格式是printf ("%s: exit(%d)\n",...)，其中包含一个字符串和一个整数，分别是这个线程的文件名（命令）和返回值。同时，任务还要求这个文件名是通过指令参数分离提取得到的文件名。

2.2 Argument Passing

在这个任务中，我们需要对TASK1中分离的参数进行入栈处理。使得函数process\_execute()能够向新的用户进程传递参数。

2.3 System Calls

在这个任务中，我们需要实现13个系统调用，在调用的同时，需要注意父子进程的同步问题以及用户传入非法指针参数问题的处理，尤其是当多个线程都需要同时对文件系统进行操作的时候，需要确保线程间同步。

2.4 Denying Writes to Executables  
这个任务相对较为简单，要求我们拒绝对于执行文件的写入操作，因为这样会导致许多不可知的后果。

三、设计方案

3.1 Process Termination Messages

3.1.1 原理

在Pintos文档中说明了头文件lib/user/syscall.h下是用户进程使用的函数，我们对该头文件下的源代码进行了分析。发现用户进程通过调用exit(int status)函数来终止当前进程。而exit()函数将函数的返回值和退出进程的系统调用编号（SYS\_EXIT，被定义为1）压入进程的栈中，并触发了30号中断。userprog/syscall.c中的函数syscall\_init()将30号中断处理例程设置为函数syscall\_handler()。我们在这个函数中根据系统调用编号，执行1号系统调用，并获取到exit()函数压入到栈中的返回值，打印线程的名字和这个返回值。

参数的分离和Argument Passing任务紧密相关，因此我们将参数的分离部分放在Argument Passing实现。

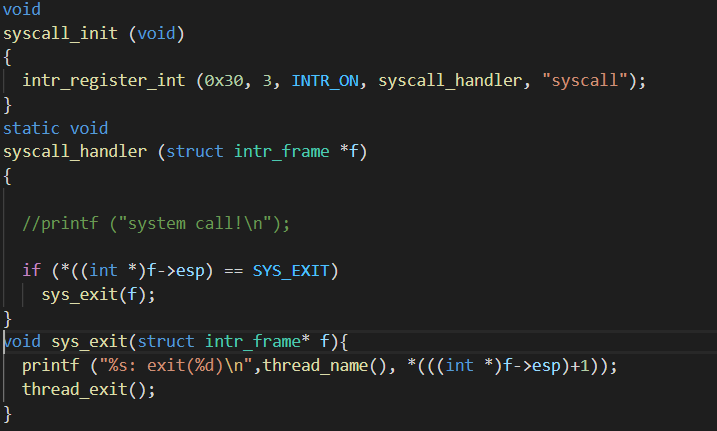
3.1.2 具体实现

（1）新增的数据结构

该任务没有添加新的数据结构。

（2）新增的代码

新增代码如下图所示。

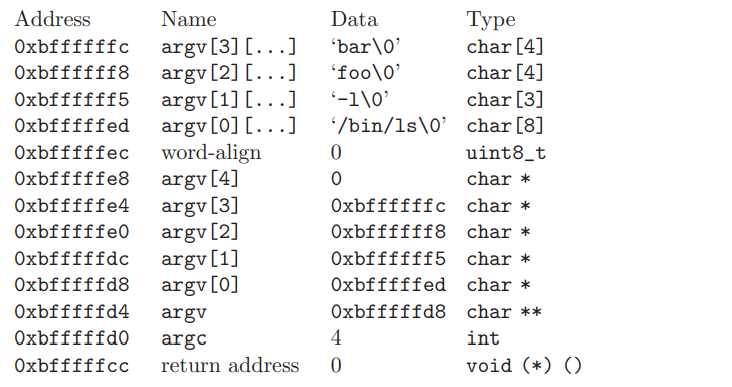


在后续任务的实现当中，我们对以上部分代码进行了补充，但是基本原理没有改变。为了体现迭代开发的过程，我们展示的是最初实现的版本。

3.2 Argument Passing

3.2.1 原理

在Pintos文档中说明了参数在栈中的格式。我们按照这个格式进行了入栈的实现。



在process\_execute()函数中，我们利用自带的函数strtok \_ r()对文件名和参数进行了分离和存储。注意到process\_execute()只是简单的调用了thread\_create()函数创建了这个线程，并没有分配栈空间。这个线程会被放到ready\_list中等待调度。这个内核线程真正被调度的时候执行start\_process()函数，这才是真正分配内存的地方。因此我们选择在star\_process()函数中实现参数的入栈。

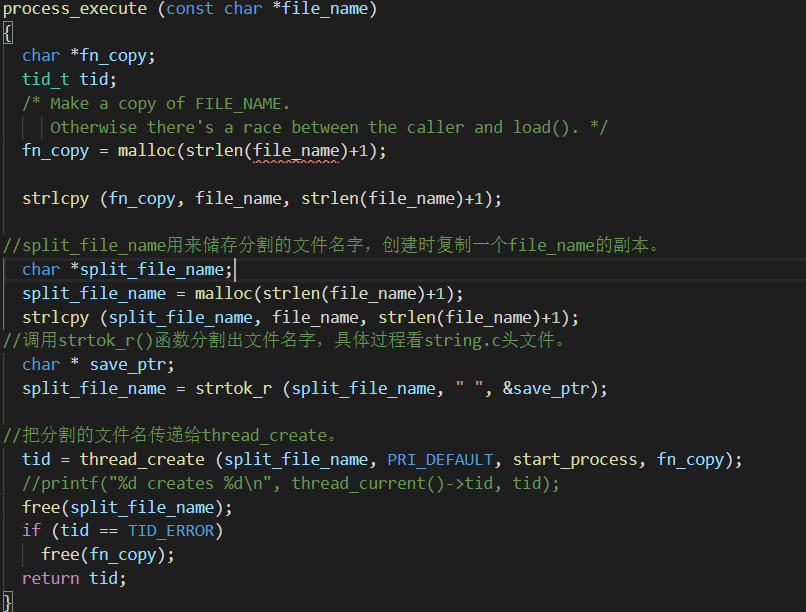
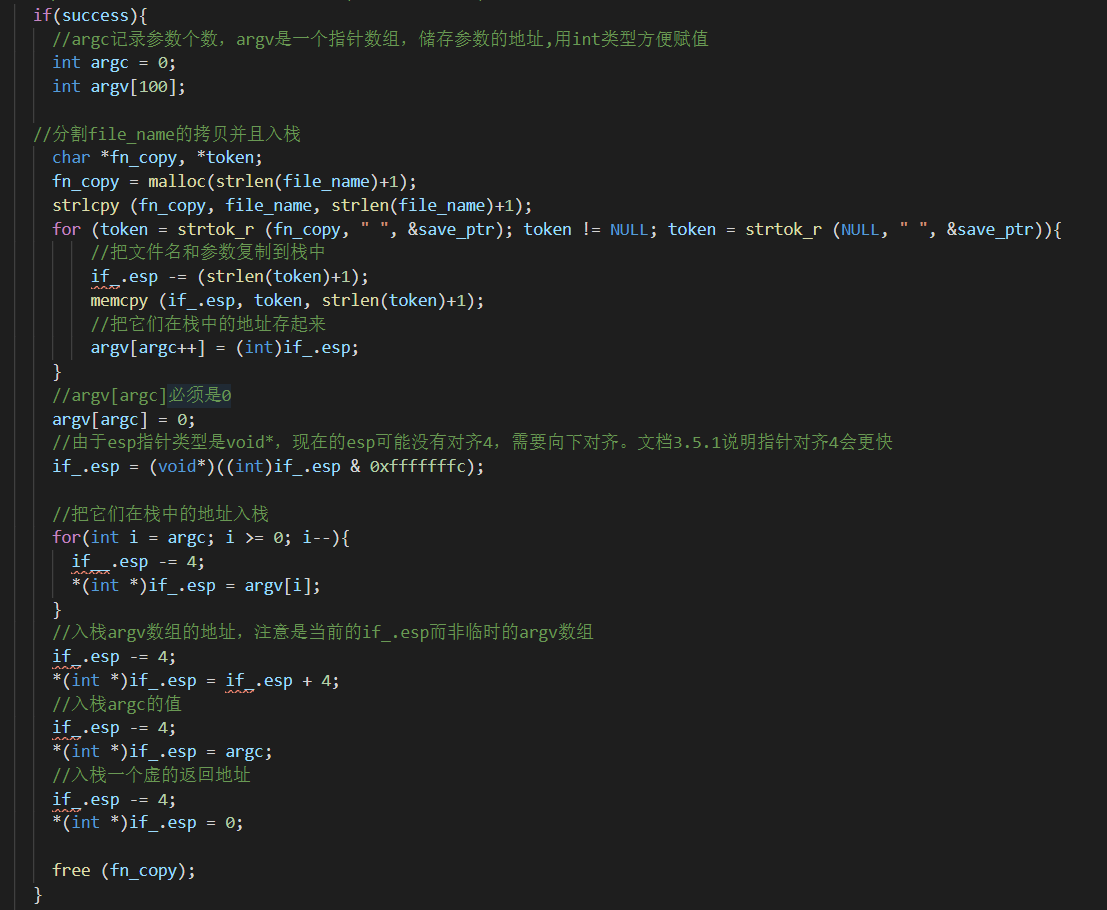
3.2.2 具体实现

（1）新增的数据结构

该任务没有添加新的数据结构。

（2）新增的代码

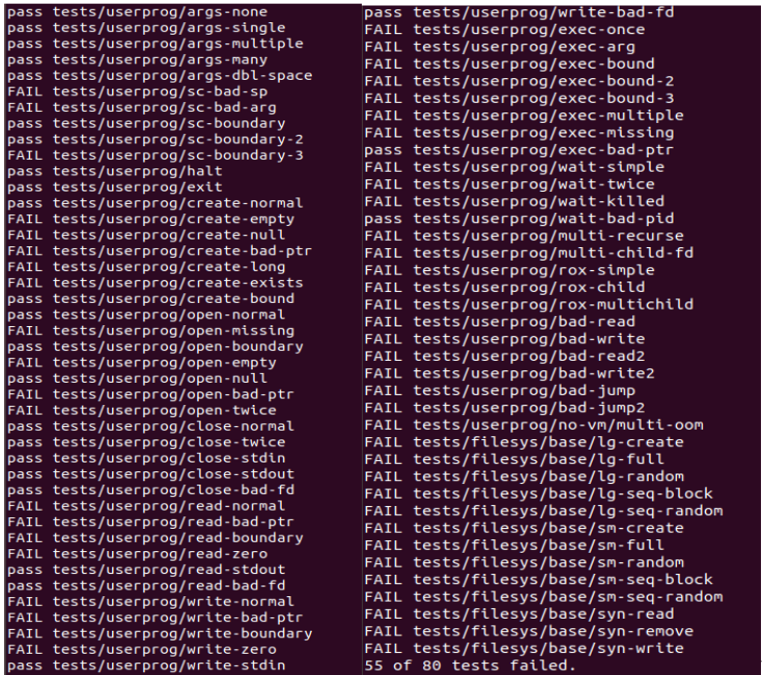
新增代码如下图所示。

3.2.3 初步测试

为了检查在前两个任务中的实现是否准确，我们实现了9号系统调用（测试文件频繁调用write()触发9号系统调用来试图向终端输出测试信息），并且在函数process\_wait()中加入了一个1千万的大数据循环拖延时间（pintos线程通过调用process\_wait()来等待测试线程，我们需要等待测试进程结束，在实现同步性之前只能通过这个方法）。

测试结果如下:



成功通过了25个测试点。

3.3 System Calls

3.3.1 原理

我们首先实现了父进程等待子进程的函数process\_wait()。父进程通过调用process\_wait()等待某个特定的子进程调用thread\_exit()结束。由于父进程调用process\_wait()函数时，子进程可能已经结束并且释放了内存空间，因此为了获取子进程的返回值，我们在子进程创建的时候就分配了一小块额外的空间（并不在子进程的空间里），当子进程退出的时候，向这块空间写入返回值和其他父进程需要知道的参数。子进程的结束和空间释放并不影响这一块额外空间。父进程得到子进程结束的信号量之后，从这块空间获取到该子进程的返回值，并且释放这一块空间。

同时，我们还设置了另一个信号量用于process\_execute()函数和start\_process()函数的同步。在start\_process()函数结束内存分配之后，process\_execute()函数才会返回。这么做的目的是如果start\_process()函数分配内存失败，process\_execute()可以获取到这一信息。

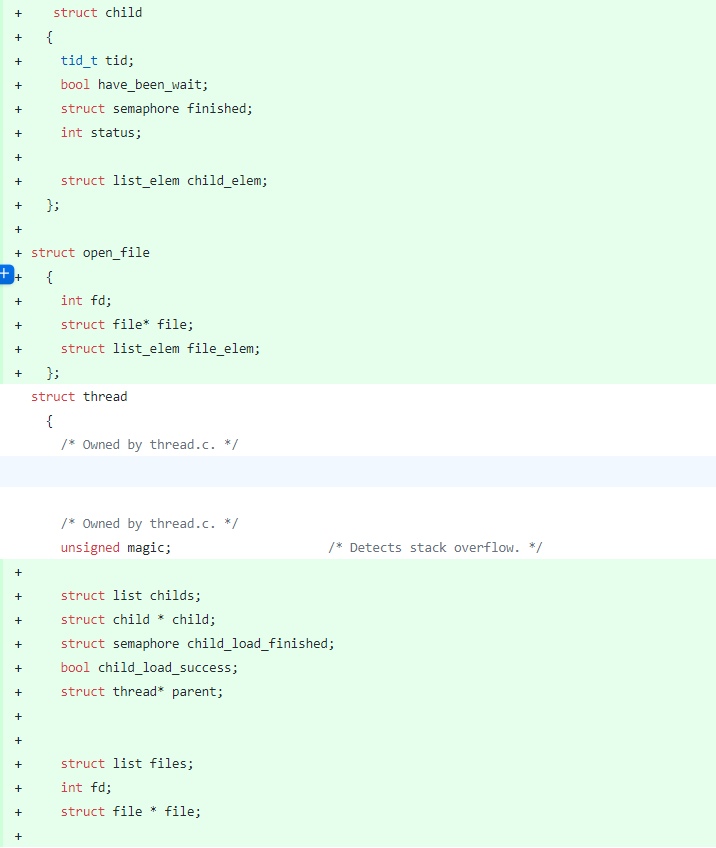
对于文件系统，我们设置了一个同步锁，用于控制多个线程试图同时操作文件系统的情况。某个线程调用文件系统的函数前，需要获得这个同步锁，函数结束后释放这个锁。与此同时，我们用了一个列表来存储进程打开的所有文件，当进程结束时关闭这些文件，避免用户进程没有关闭文件的操作。

对于指针的合法性检查，我们实现了函数assert\_is\_vaddr(const void \*vaddr,int offset)，用于确保系统调用函数中访问区域的合法性。

3.3.2 具体实现

（1）新增的数据结构

如下图所示：



其中，

struct child是一个结构体，是子进程向父进程传递返回值的媒介。

open\_file是一个结构体，进程用其来记录打开的文件。

childs是一个列表，连接所有子进程的child结构体。

child\_load\_finished和child\_load\_success用于process\_execute()函数和start\_process()函数的同步，前者是一个信号量，后者记录内存的分配是否成功。

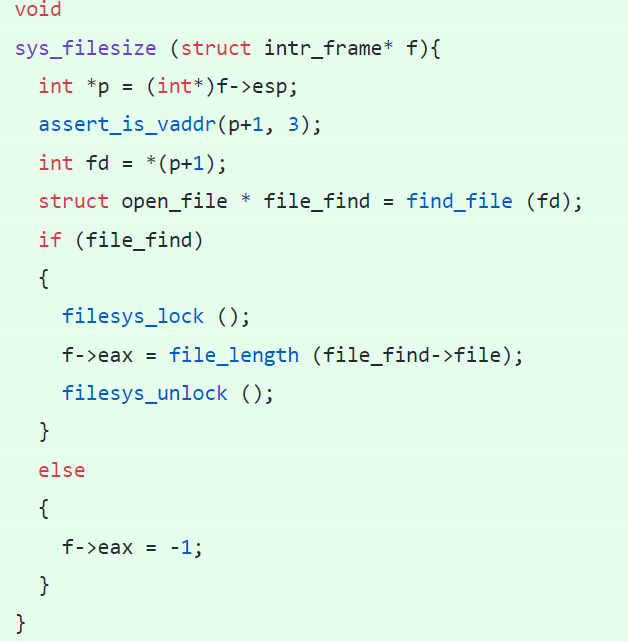
parent是指向父结构体的指针。

files是一个列表，连接所有打开的文件。

fd和file存储该进程是因为执行哪个文件而创建的，用于进程退出关闭文件。

（2）新增的代码

由于代码量较大，我们不做全部的展示。仅展示一个系统调用的典型实现。



这个系统调用返回一个文件的大小，其中文件fd作为参数。首先确保文件的fd位于一个合法的位置，接着获取文件系统锁，接着算出文件的大小，然后释放这个锁。由于lib/user/syscall.h下的函数的返回值是通过eax获取的，我们还需要修改eax的值作为用户进程系列函数的返回值。

3.4 Denying Writes to Executables  
 该部分实现较为简单，按照pintos说明文档里给出的描述指示，只要在有需要拒绝写入需求的地方调用函数file\_deny\_write() 即可实现该功能。

在userprog / process.c的load() 函数中我们调用了一次该函数，load() 函数作用是将需要执行的文件绑定到线程上去执行，那么此时应当确保该文件不能被其他线程修改，否则会出现同步错误，故在此调用该函数。

