**Minix3.3 shell安装、测试、编译过程说明文档**

10215501412 彭一珅

[一、使用Vmware workstation安装minix3.3 1](#_Toc129281984)

[·实验环境 1](#_Toc129281985)

[二、Shell代码实现过程 2](#_Toc129281986)

[·实验工具 2](#_Toc129281987)

[·实验目的 2](#_Toc129281988)

[·实验过程 3](#_Toc129281989)

[实现shell主体 3](#_Toc129281990)

[Shell内置命令 6](#_Toc129281991)

[Shell程序命令 14](#_Toc129281992)

[三、程序测试与心得体会 21](#_Toc129281993)

[·实验前的准备与学习 21](#_Toc129281994)

[·Execvp无法读取环境变量 21](#_Toc129281995)

[·管道实现时遇到的问题 22](#_Toc129281996)

[·后台运行时遇到的问题 22](#_Toc129281997)

[·mytop指令实现时遇到的问题 23](#_Toc129281998)

[四、程序运行结果 24](#_Toc129281999)

[·cd命令 24](#_Toc129282000)

[·program命令（以ls、vi为例） 24](#_Toc129282001)

[·程序输入重定向到文件 26](#_Toc129282002)

[·以文件为输入执行命令（以grep为例） 26](#_Toc129282003)

[·带有管道符的命令 26](#_Toc129282004)

[·带有后台运行符的命令 26](#_Toc129282005)

[·mytop命令 26](#_Toc129282006)

[·history命令 27](#_Toc129282007)

[·exit命令 27](#_Toc129282008)

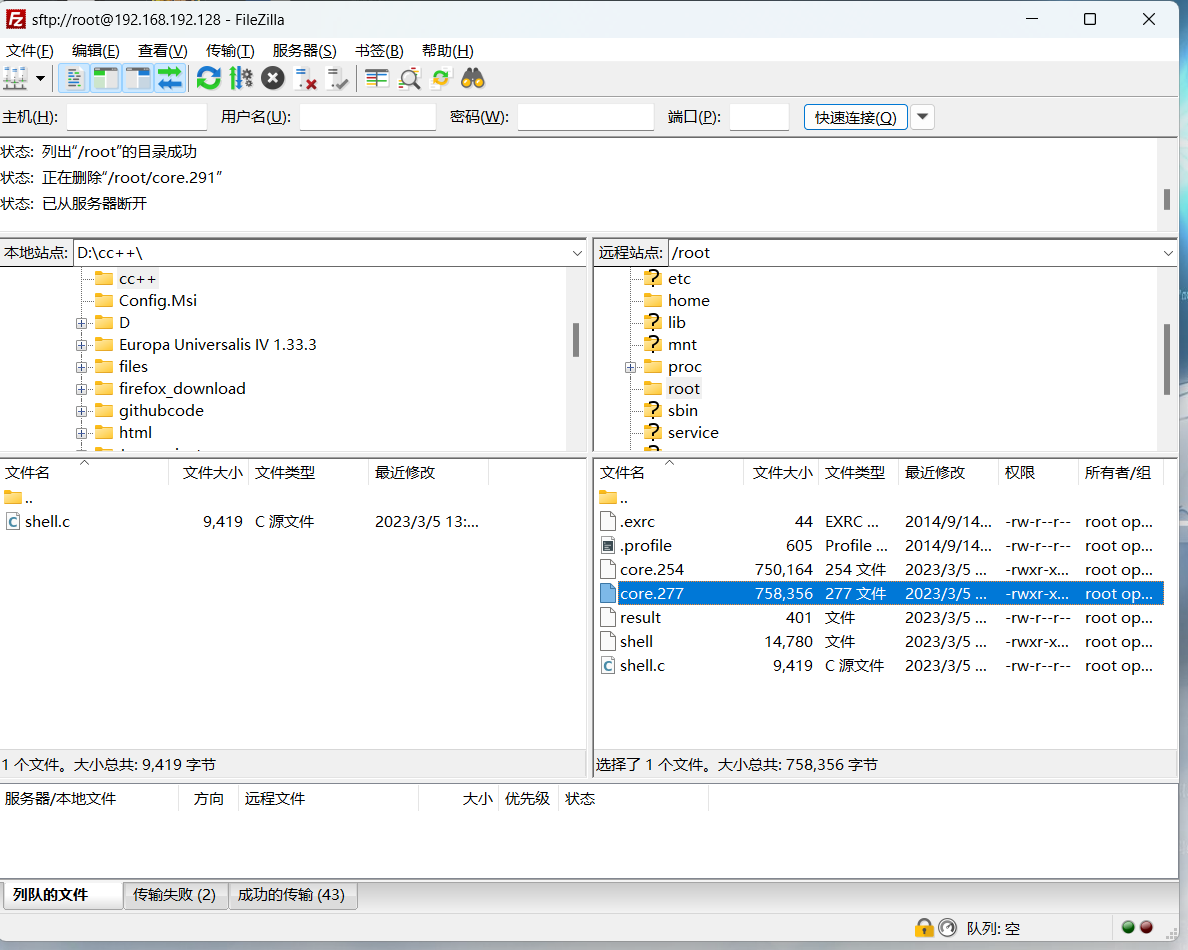
# 一、使用Vmware workstation安装minix3.3

·实验环境

·Windows11物理机

下载Vmware Workstation pro 16.1，并下载3.3.0版本minix镜像文件，将其解压。在Vmware中启动minix虚拟机，注册账户，用pkgin命令安装开发环境所需要的库。通过Filezilla软件将虚拟机与物理机进行网络连接。

连接后得到如下界面：



即可访问虚拟机中的文档。

# 二、Shell代码实现过程

·实验工具

·Vscode C语言开发环境

·Clang编译工具

·minix3.3虚拟机

·实验目的

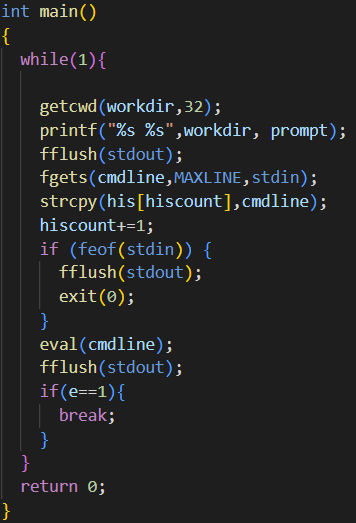
·实现一个基本的shell，可以在虚拟机中运行，需要包含的功能如下：

1. 带参数的程序运行功能
2. 使用> >> <符号实现重定向功能
3. 使用管道符号|在程序间传递数据
4. 使用后台符号&将命令以后台方式执行
5. 工作路径移动命令cd
6. 程序运行统计mytop
7. Shell退出命令exit
8. history n显示最近执行的n条命令

·实验过程

实现shell主体

首先我以一个while循环的形式实现了shell的主体：



前两行输出命令行的提示符：当前的工作目录和提示符“$”：



随后用fgets从标准输入文件读取用户输入的命令行，并将输入的命令行保存在his数组当中。his数组的定义如图所示：



相关的宏定义分别表示：MAXNUM最多输入的命令数目、MAXLINE每条命令最多的字符数。

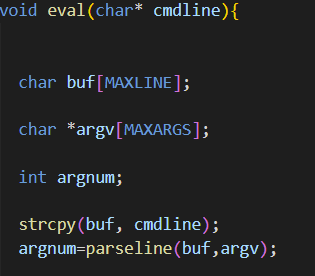
当读入的标准输入文件是结束符时，清空输入行并退出主程序。

将读入的命令行加入缓冲区buf数组（buffer），传入eval函数。

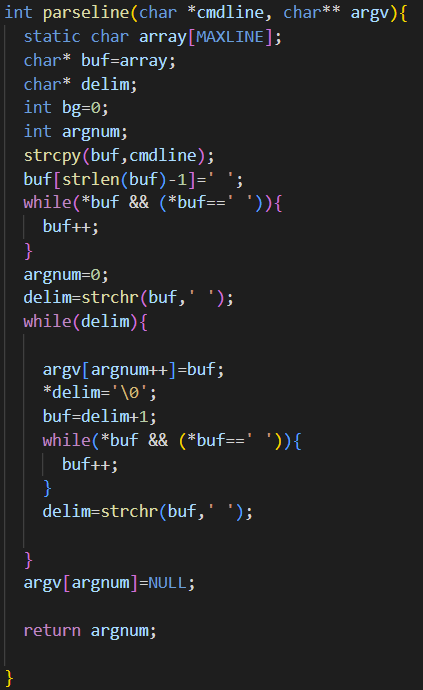
然后设置程序退出的条件。全局变量e=0作为exit内置命令的标识，exit将e置为1，此时主程序将跳出循环并返回：



随后是eval函数的实现。这个函数对命令行进行评估（evaluate），并执行相应的命令。首先用parseline函数将命令行解析：



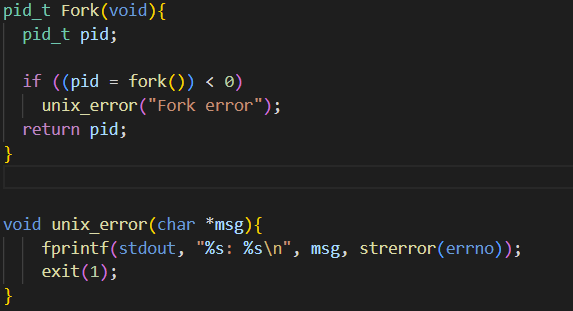
其中argv数组的每一个元素都是一个指向字符串的数组，将指向buf命令行用空格分割的每一个命令参数。元素数MAXARGS是由宏命令所限定的最多能解析的参数数目。



Parseline函数将命令按照空格分割，并存入argv当中。首先将命令行拷贝到array数组中，用指针buf进行遍历。将命令行最后的换行符“\n”替换为空格；将命令开头出现的空格忽略，buf指针向前直到出现非空格的字符；argnum用于记录命令行中的参数个数，delim（delimiter）指针指向命令行中参数与参数之间的空格，通过strchr指向到第一个空格所在的位置。

随后的while循环分为以下五个步骤：1.记录buf所指向的参数2.将空格替换为结束符“\0”3.buf指向空格后的下一个字符4.将buf向前移动直到指向的字符不再是空格5.delim指向下一个空格出现的位置。

最后对unix自带的fork命令进行错误处理的包装，使其调用时可以输出错误信息：



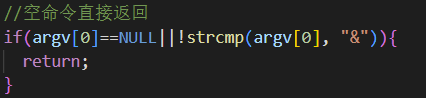
至此，shell的主体已经编写完毕。

Shell内置命令

本次实验要求实现的内置命令分别是“cd”、“history”、“exit”、“mytop”，可用strcmp函数直接进行判别。下图注释中的标号是命令在课程设计一讲解ppt中出现的顺序：

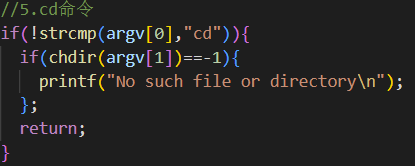


忽略空命令（可能是只有空格或只输入了回车）和只有一个&符号的命令：



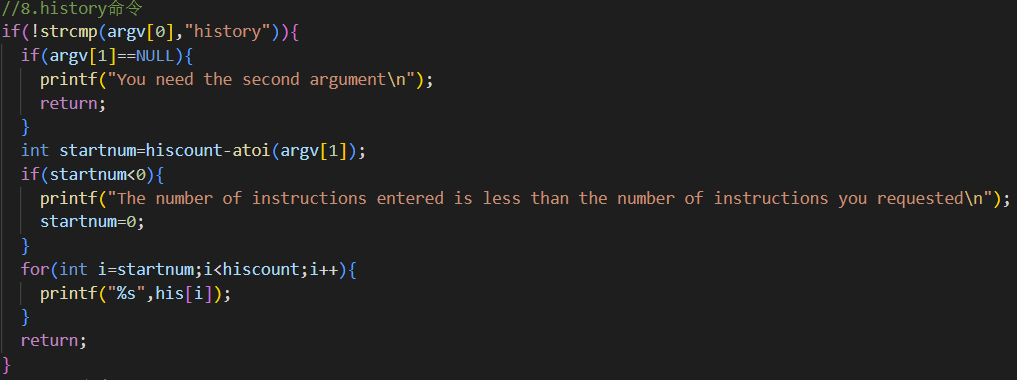
·cd命令

首先，cd命令可直接使用系统调用int \_\_cdecl chdir(const char \*)进行实现，加上对错误的处理，即可写出如下代码：



·history命令

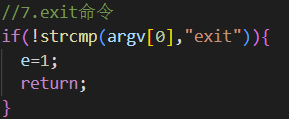
接下来是history命令，直接打印全局变量his数组中的内容即可：



其中加入了两种错误情况处理。history命令如果没有第二个参数则留下错误信息，直接返回。如果要求的命令数目多于从运行程序开始输入的命令总数目，（此时用于计数的startnum为负）则先输出错误信息，再将所有输入过的命令进行输出。

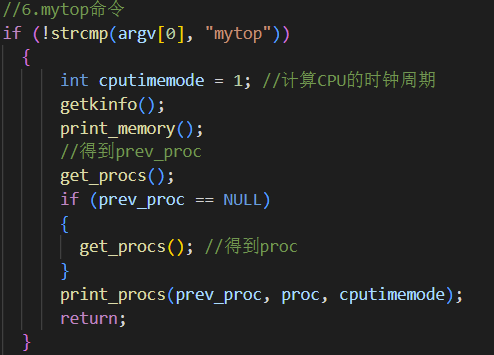
·exit命令

将全局变量e置为负，返回即可：

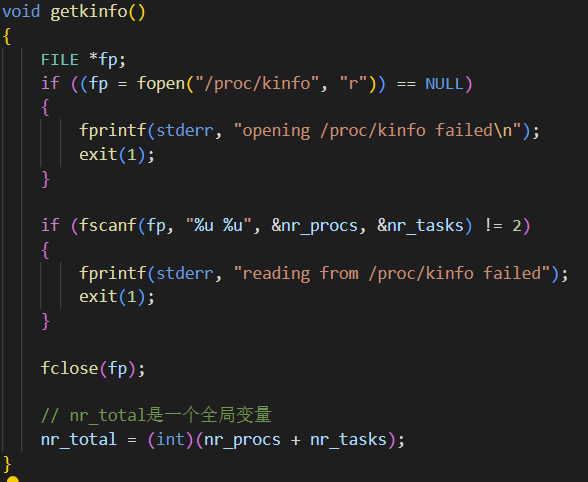


·mytop命令

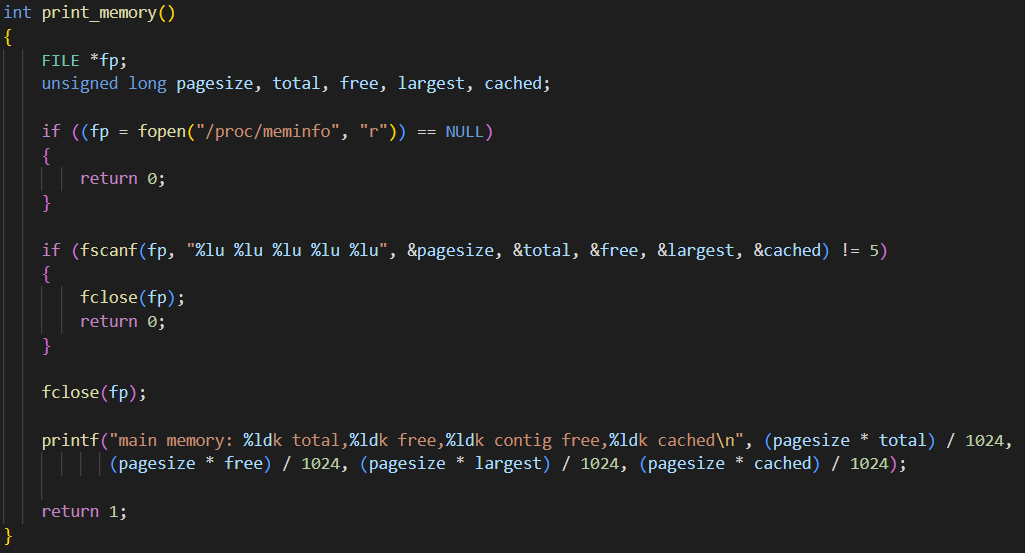
mytop命令需要输出内存使用情况和cpu使用情况，因此实现过程包括两部分。Print\_memory函数获取并输出内存使用情况，getkinfo\get\_procs\print\_procs获取并输出cpu占用情况：



先通过getkinfo，读取文件/proc/kinfo获取总的进程和任务数，存入全局变量nr\_total：

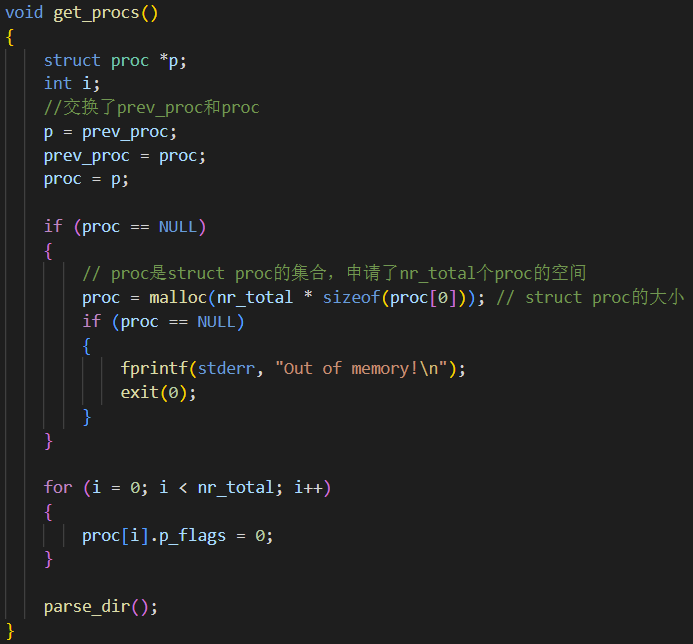


然后print\_memory函数打开并读取/proc/meminfo文件，读取其中的五个数据：页面大小，总页面数，空闲页数，最大页数，缓存页面数，从而计算出内存占用情况：

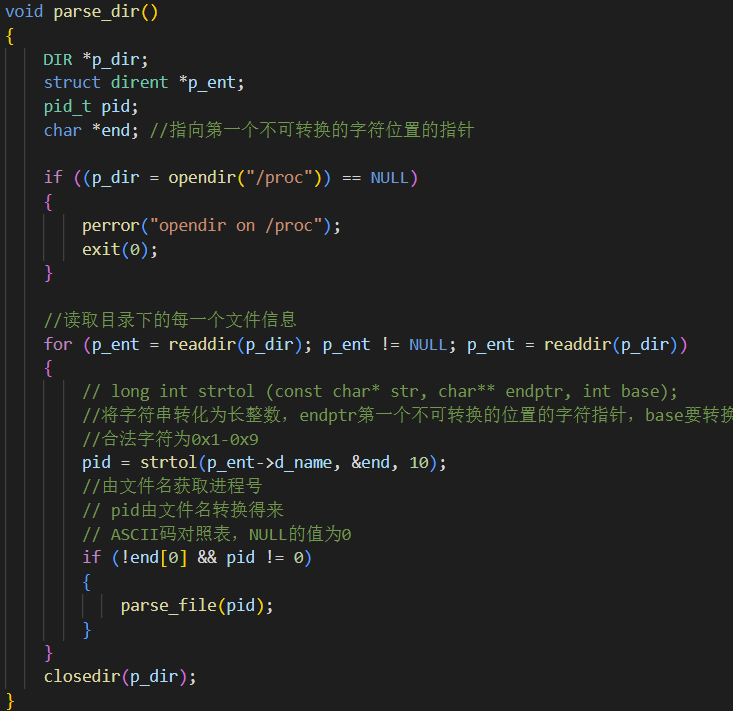


可以看到，get\_procs函数开始处交换了prev\_proc和proc两个全局指针指向的首位地址：

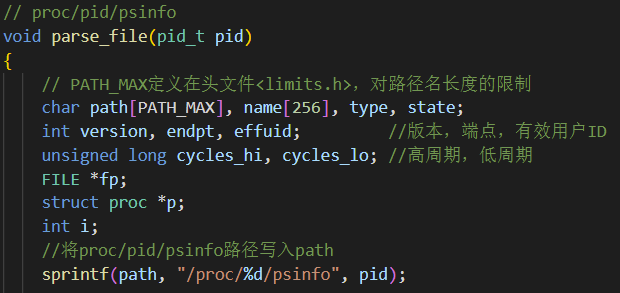




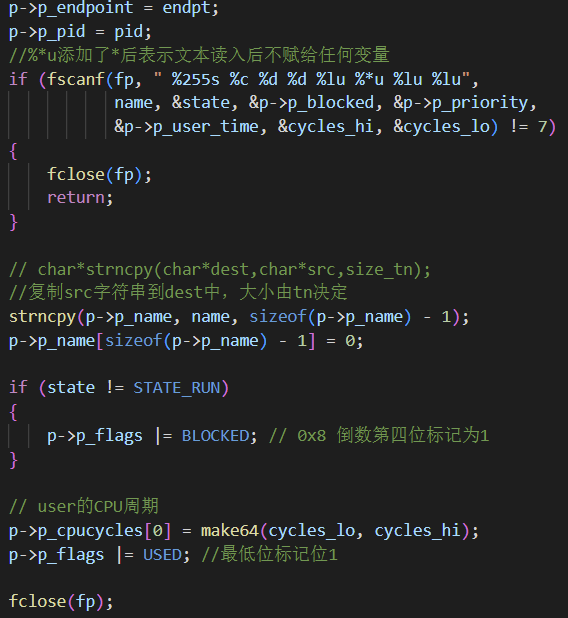
然后为proc指向的进程指针分配了保存nr\_total个进程结构体所需的内存，将每个内存的flag初始化为0，接下来使用parse\_dir函数读取/proc文件夹下的文件信息：



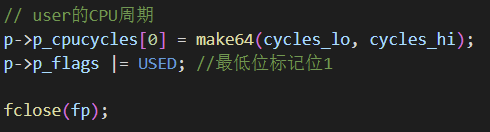
将读取到的文件夹名字存入变量pid当中，并用parse\_file函数读取每个进程信息所存储的以pid命名的文件夹：



然后打开文件，用fscanf将数据按顺序读取，赋给proc结构体p的各个属性，其中与mytop实现有关的内容如下：

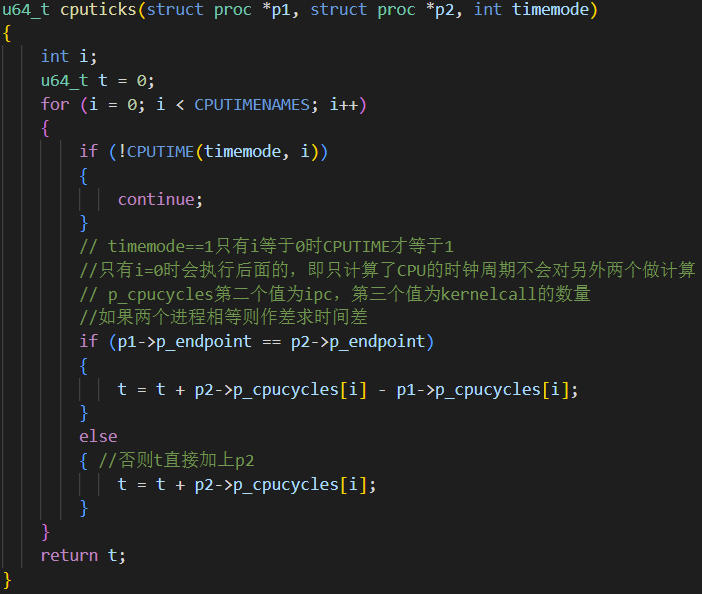


进程结构体的pid记录文件夹的名称，读取高周期、低周期；



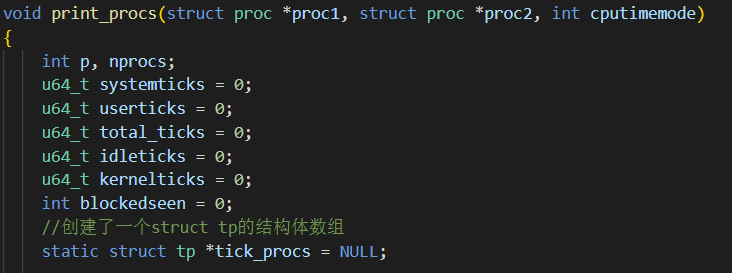
通过make64函数，将cycles\_lo的数值作为cpucycles的高32位，cycles\_hi的数值作为低32位，然后将proc数组指向的该位置标为USED，表示该位置已经保存了一个进程的内容。

然后用cputicks计算进程运行的时间：

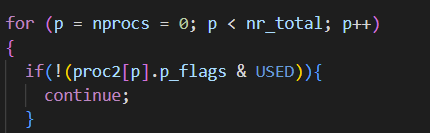


如果两个进程相等则对prev\_proc和proc两个进程的cpu转数作差，如果prev\_proc中没有保存，就是新加入的进程，直接使用最新的数据作为ticks即可。

输出cpu使用率由print\_procs来实现：



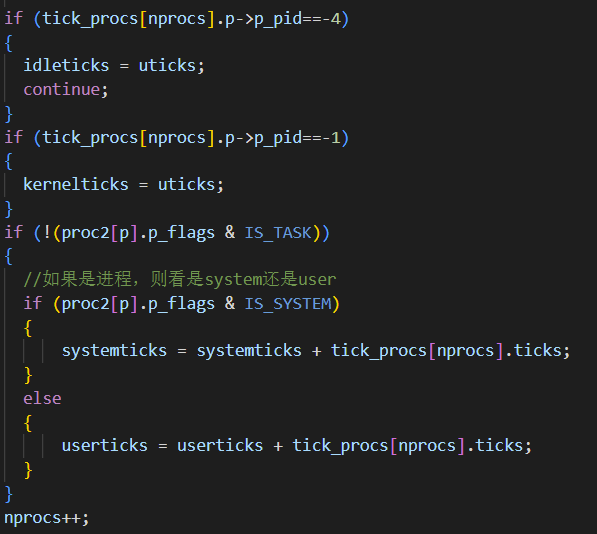
遍历proc数组，如果该位置没有保存进程数据则跳过此循环：



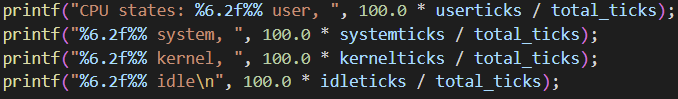
调用cputicks计算每个进程的ticks：



如果进程的pid是-4则代表进程是minix设定的idle进程，只在没有其他进程运行时运行；如果pid是-1则是内核进程，这些pid为负数的文件夹都代表任务，即kinfo文件中的第二个数，在minix中数值是5，具体表现为/proc文件夹下-1、-2、-3、-4、-5五个任务文件夹；如果是pid为正数的进程，则分为用户U和系统S两种，分别计算利用率即可：

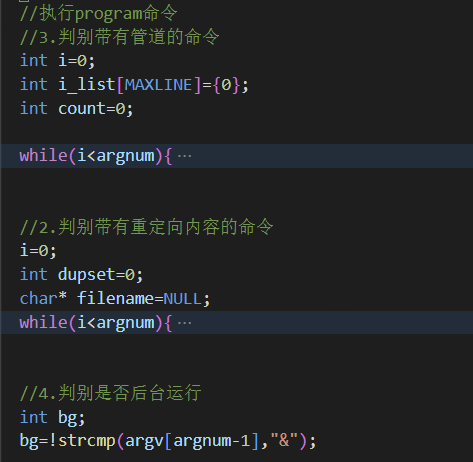


最后输出百分比：



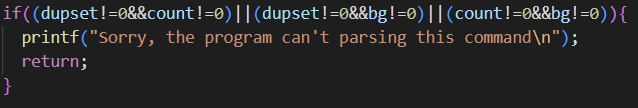
Shell程序命令

Shell要实现的程序命令有四种情况：带有管道、带有重定向符号、带有后台运行符号&。先对这三种情况进行判别：

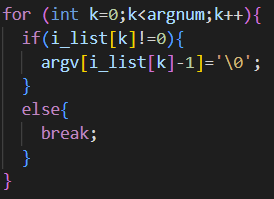


基本思路是一致的，遍历argv数组，利用strcmp函数对每个参数进行判别，用变量记录特殊符号出现的位置。i\_list数组记录管道连接的不同命令的起始位置，filename记录重定向的文件名称。

如果以上三种符号同时出现了两个，shell无法解析这种命令，则输出错误信息并返回：

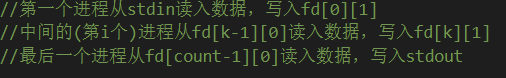


将特殊符号替换成结束符“\0”，便于执行指令。

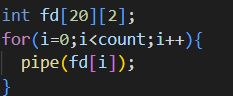


·执行管道指令

管道指令的实现有三个步骤：



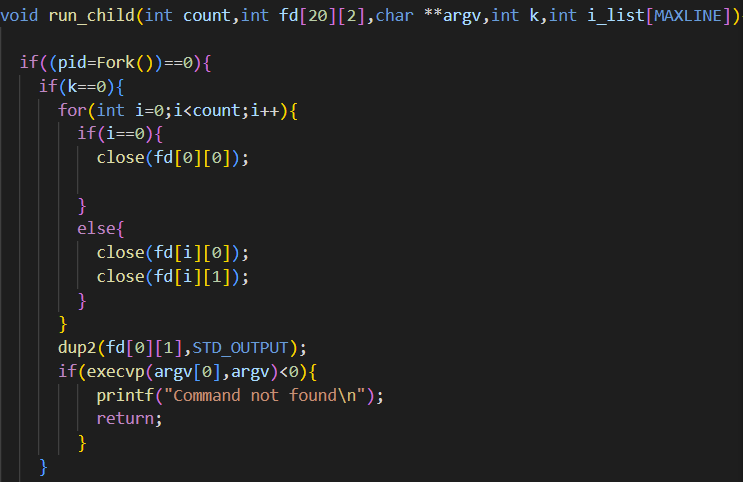
先用系统调用pipe新建n-1个管道：



然后新建子进程，并递归调用函数run\_child：

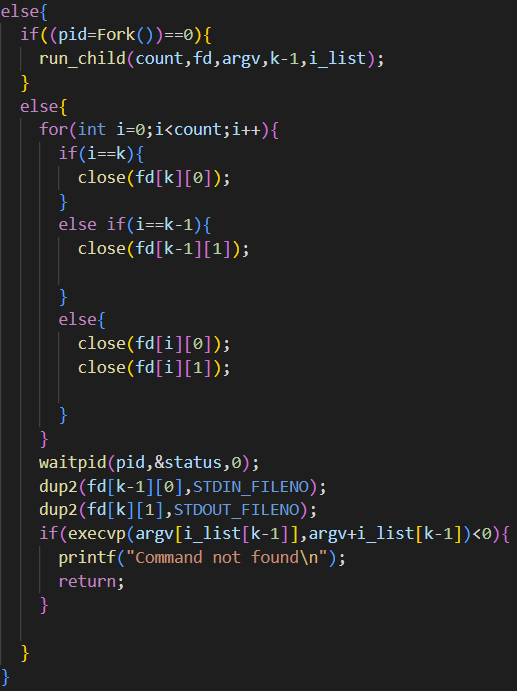


首先父进程关闭管道，子进程执行第n条指令（这条指令与函数内的n-2条指令不同，因为输出端不是管道而是标准输出），同时再创建子进程调用run\_child：



函数的返回条件是k=0，即执行第1条指令，这条指令与剩下n-2条不同，输入端是stdin标准输入文件。所以用for循环关闭其他管道，用dup2将程序输出定位到管道，执行第一条指令。

另一种情况，k不为0时，执行中间的n-2条命令：



基本结构与调用该函数的部分相似，在父进程中关闭多余的管道并运行第k条指令，子进程调用函数，函数里执行第k-1条指令，以此类推。

调用函数时本身就创建了子进程，这使函数原本所在的进程可以等待子进程结束，以使管道里的内容按顺序传递。

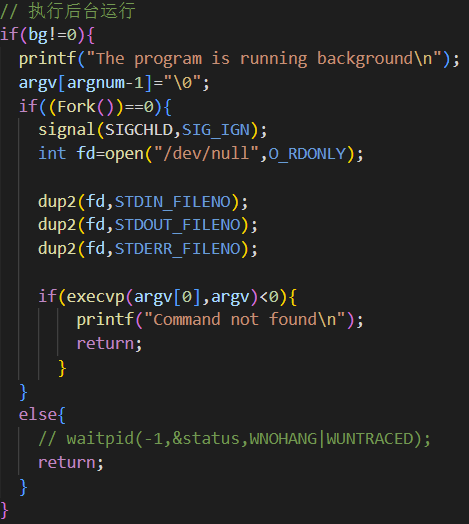
·执行重定向到文件

重定向到文件需要打开文件，用dup、close函数将标准输入（输出）文件关闭，并重定向到打开的文件中。此时使用dup2也是可以的。另外两种重定向符号的实现与下图基本类同：

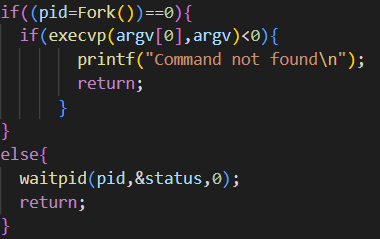


·执行后台运行

将进程进行后台运行，需要把输入和输出导入到/dev/null文件中，并向父进程发送一个SIGCHLD信号：



最后，实现普通的前台运行，只需用execvp将program载入到子进程即可：



# 三、程序测试与心得体会

在本次实验中，我将shell.c程序通过filezilla传输到minix虚拟机，并使用clang编译，做了很多不同的尝试，对代码进行了数次纠错和改进。下面将简述我在实验过程中遇到的问题和解决方法。

·实验前的准备与学习

为了理解pipe、dup、dup2、close、open、chdir等系统调用命令的使用方法，我参阅了《unix环境高级编程》的相关章节，对dup创建管道的原理和阻塞读取写入的性质、exec类型函数的不同点、O\_RDONLY/STDIN\_FILENO等宏定义的意义有了相应的理解。在代码中尝试了不同的实现方式，例如：

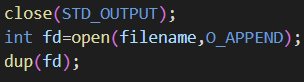




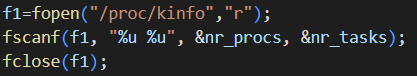
其中STD\_OUTPUT是我定义的宏：

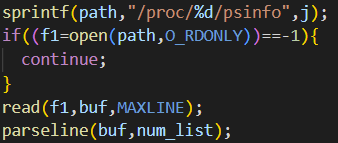


也尝试使用dup2和dup实现同样的功能：



用fscanf类比open+read系统调用，实现读文件操作：

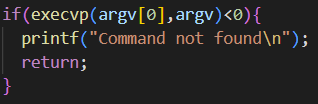




其中，fscanf可以占用较少的内存。

·Execvp无法读取环境变量

起初，我对execvp函数进行包装，以便于输出错误信息，如图所示：



但是无法使用父进程原有的环境变量，即想要执行ls就需要输入完整路径/bin/ls。于是把包装好的函数替换成未封装的状态，分别输出错误信息，就可以正常执行命令了。

·管道实现时遇到的问题

在实现管道的递归时，我对子进程与父进程分别执行的内容、开启子进程的时机等做了仔细的推理。起初是设计由主程序开启n个子进程，分别执行每个指令，将输入或输出重定向到管道中，但是由于pipe函数只能由有亲子关系的进程互相传递，只能再次进行修改，最终得出递归式结构。

在递归结构中，每一个子进程都是“干净地”执行的，即这些子进程在执行时没有被关闭任何管道，所有[n-1]个管道都是打开的。在创建子进程之后，每个父进程才会通过for循环关闭其他管道，留下所需要的管道。在调试这段代码时，我遇到了两个问题。

首先，执行ls | grep e时会发生grep程序执行不结束的现象，此时命令行不再输出“[path]$”的输入提示符。这是由于最原始的只执行waitpid的父进程（也即shell所在的进程）没有关闭grep程序接收输入所用的管道，所以grep始终在等待管道中的输入，而waitpid使得父进程不会在grep所执行的程序结束前结束，程序就无法结束了。

此外，用ls | grep e | grep l测试的时候，发现了grep执行输出完毕后直接返回的现象，即不再输出输入提示符，直接退出shell进程。这是因为没有创建子进程而在shell进程中进行execvp，将shell进程覆盖掉了。

·后台运行时遇到的问题

测试后台运行功能时，我使用了sleep 3命令，起初前台运行的功能还正常，可以停止3秒再打印输入提示符。但是当输入一条sleep 3 &后，任何命令都变成后台运行了，输入提示符总是在执行的程序结束之前就被打印。后来将前台运行的：

waitpid(-1,&status,0);

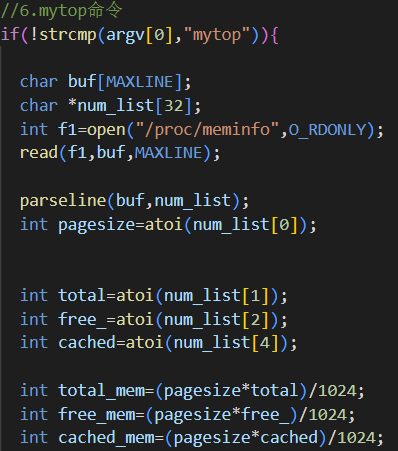
命令改为：

waitpid(pid,&status,0);

就可以正常执行了。这是因为如果将第一个参数设为-1，那么等待集合的子进程包含了原先后台运行的程序，程序发送的信号就使得父进程不再等待子进程的结束，直接返回了。

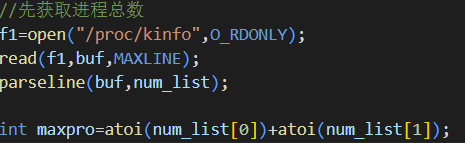
·mytop指令实现时遇到的问题

mytop命令要求输出：总体内存大小、空闲内存大小、缓存大小、CPU使用百分比。因此起初，我先读取/proc/meminfo中的信息，计算出前三个数：

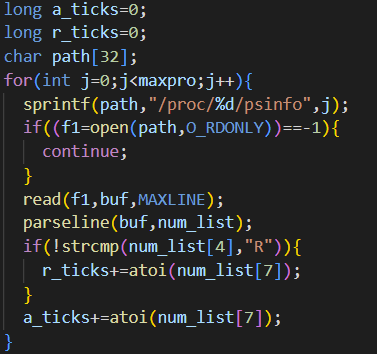


此时使用系统调用open、read，读取到的文件内容存在buf数组中，使用parseline函数进行解析，num\_list指向文件中的每一个数字。

然后计算cpu使用百分比。首先读取/proc/kinfo中的信息，将两个数加起来得到进程总数maxpro:



然后遍历这些程序的pid，访问/proc/[pid]/psinfo文件，获取每个进程的信息。将ticks累加得到a\_ticks；将状态为R（run）的进程ticks累加得到r\_ticks：



根据对ls /proc对文件夹内容的观察，如果没有以pid命名的文件夹则跳过本次循环。最后将r\_ticks/a\_ticks即得到cpu占用率。

但是通过阅读源码，我发现这种做法是有问题的。经过对系统文件的查看，我发现idle进程和kernal进程的user\_time都是0。因此学习top源代码中的实现方法，使用cpu\_cycles计算使用时间，从而得到cpu使用占比。

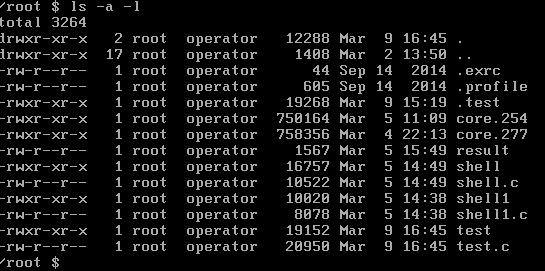
# 四、程序运行结果

·cd命令

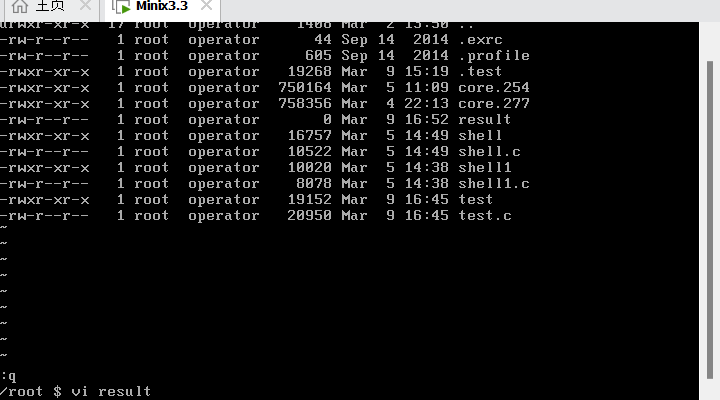


命令输入提示符前显示的是shell程序的工作路径。

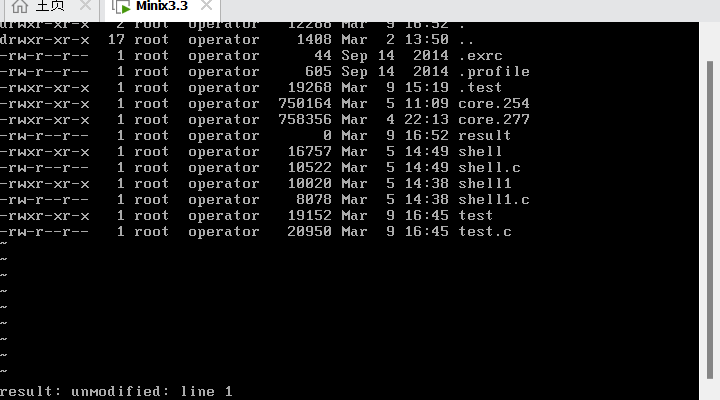
·program命令（以ls、vi为例）



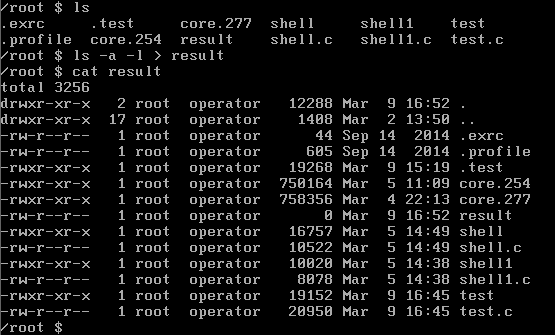
Vi运行前：



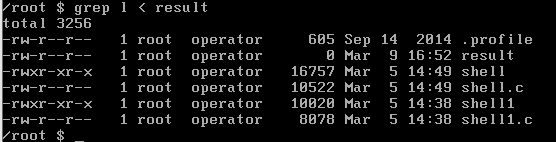
vi运行后：



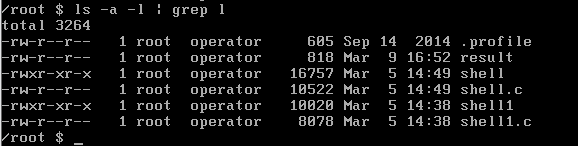
·程序输入重定向到文件



·以文件为输入执行命令（以grep为例）



·带有管道符的命令

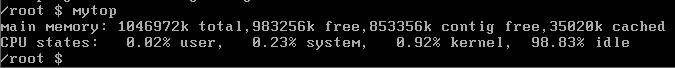




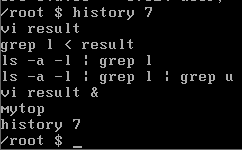
·带有后台运行符的命令



·mytop命令



·history命令



·exit命令

