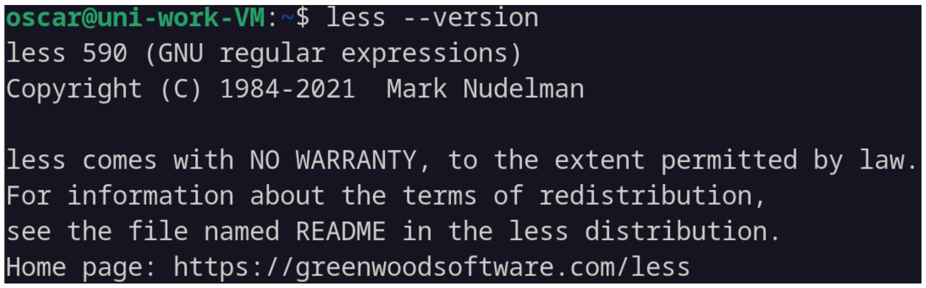
`less .bashrc` - A Deep Dive into a Simple Command

Oscar Topliss

June 2024

1 Table of Contents

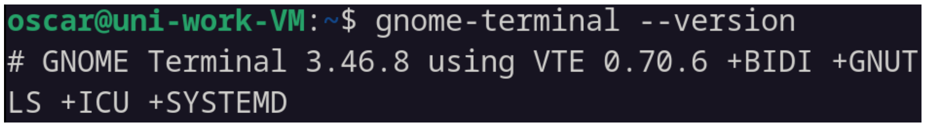
Contents



```
oscar@uni-work-VM:~$ less --version
less 590 (GNU regular expressions)
Copyright (C) 1984-2021  Mark Nudelman

less comes with NO WARRANTY, to the extent permitted by law.
For information about the terms of redistribution,
see the file named README in the less distribution.
Home page: https://greenwoodsoftware.com/less
```

Figure 1: Version number of `less`



```
oscar@uni-work-VM:~$ gnome-terminal --version
# GNOME Terminal 3.46.8 using VTE 0.70.6 +BIDI +GNUT
LS +ICU +SYSTEMD
```

Figure 2: Version number of Gnome Terminal

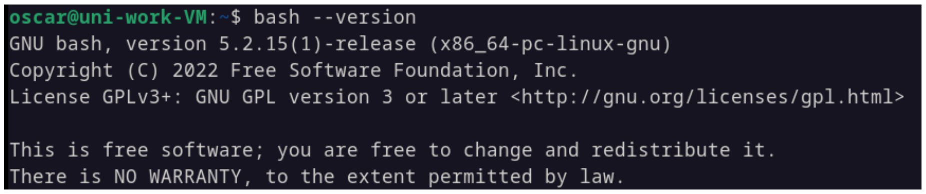
2 Introduction

2.1 Description

The aim of this report, as per the coursework brief, is to discuss what occurs between an unprivileged user entering the command `less .bashrc` into their terminal, and the resulting process exiting. This will involve analysis of how the process interacts with the underlying operating system, including how files, memory, and process data are managed.

2.1.1 Software versions

Throughout the report I will be using Debian 12 Stable, and version 590 of `less` which comes with it (see figure 1). It is assumed that the user is entering the command in a graphical terminal window, in this case Gnome Terminal version 3.46.8 (see figure 2), using bash version 5.2.15(1) (see figure 3)
Any conclusions may not apply to other operating systems or versions of `less`



```
oscar@uni-work-VM:~$ bash --version
GNU bash, version 5.2.15(1)-release (x86_64-pc-linux-gnu)
Copyright (C) 2022 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>

This is free software; you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.
```

Figure 3: Version number of bash

3 Report

3.1 Initiation

3.1.1 Context of the command

The `PATH` environment varible defines where to search for the executables which correspond to commands(Debian 2023*b*). When the `less` command is entered, the bash terminal searches using this environment variable to find the `less` executable, and execute it. The bash terminal likely gains access to this environment variable via the DisplayManager(Wiki 2023), which would be `gdm3` in the case of Debian 12 running Gnome(Debian 2023*a*).

3.2 VFS

3.2.1 What VFS is

VFS stands for **Virtual Filesystem**. It is a part of the linux kernel which acts to present files and directories in a consistent way to processes and users, regardless of what type of filesystem each file is actually stored in(Bovet & Marco 2006, p. 456).

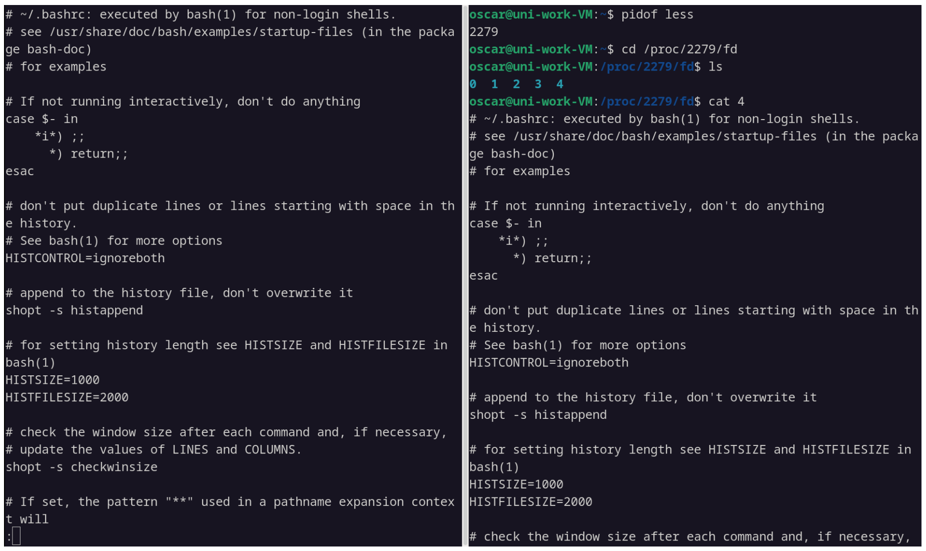
3.2.2 Common file model

The Common File Model is the abstract representation used by VFS for managing files. It splits files and their uses into a number of object types, which enable different processes to access the same files at the same time.

3.2.2.1 Superblocks Each superblock represents a particular file system, and the collection of files/directories stored within it.(Senofsky et al. 2022)

3.2.2.2 Inodes Inodes ("Index nodes"(Senofsky et al. 2022)) represent the files themselves (including directories as the Common File Model considers directories to be files(Bovet & Marco 2006, p. 459)). Inodes include data relating to the file, including who can access the file and in what way (`i_mode` variable), and a pointer referencing which supberblock the file belongs to (`i_sb`)(Bovet & Marco 2006, p. 467).

3.2.2.3 Dentry objects Dentry objects are used to link human-understandable path/resource names to the inodes which they represent(Bovet & Marco 2006, p. 460). There can be multiple dentry objects referring to the same inode in cases where multiple hard links exist.(Bovet & Marco 2006, p. 460). (hard links being direct references to an inode, as opposed to soft links which refer to the inode's location at the time of the soft links creation(GeeksForGeeks 2022)). Dentry objects refer to individual files including directories, and as such a path made up of multiple directories such as `/home/user/Desktop` will consist of chained-together dentry objects for `/`, `home` etc.(Bovet & Marco 2006, p. 475)



```
# ~/.bashrc: executed by bash(1) for non-login shells.
# see /usr/share/doc/bash/examples/startup-files (in the packa
ge bash-doc)
# for examples

# If not running interactively, don't do anything
case $- in
    *i*) ;;
      *) return;;
esac

# don't put duplicate lines or lines starting with space in th
e history.
# See bash(1) for more options
HISTCONTROL=ignoreboth

# append to the history file, don't overwrite it
shopt -s histappend

# for setting history length see HISTSIZE and HISTFILESIZE in
bash(1)
HISTSIZE=1000
HISTFILESIZE=2000

# check the window size after each command and, if necessary,
# update the values of LINES and COLUMNS.
shopt -s checkwinsize

# If set, the pattern "**" used in a pathname expansion contex
t will
:
```

```
oscar@uni-work-VM:~$ pidof less
2279
oscar@uni-work-VM:~$ cd /proc/2279/fd
oscar@uni-work-VM:/proc/2279/fd$ ls
0  1  2  3  4
oscar@uni-work-VM:/proc/2279/fd$ cat 4
# ~/.bashrc: executed by bash(1) for non-login shells.
# see /usr/share/doc/bash/examples/startup-files (in the packa
ge bash-doc)
# for examples

# If not running interactively, don't do anything
case $- in
    *i*) ;;
      *) return;;
esac

# don't put duplicate lines or lines starting with space in th
e history.
# See bash(1) for more options
HISTCONTROL=ignoreboth

# append to the history file, don't overwrite it
shopt -s histappend

# for setting history length see HISTSIZE and HISTFILESIZE in
bash(1)
HISTSIZE=1000
HISTFILESIZE=2000

# check the window size after each command and, if necessary,
```

Figure 4: Two terminal windows, with the left showing the output of the `less .bashrc` command and the right showing the list of files in that processes `proc/PID/fd` directory.

To achieve this, dentry objects have pointers to both their parent dentry object (`d_parent` variable), and a list of sub-directory dentry objects (`d_subdirs`)(Bovet & Marco 2006, p. 475).

3.2.2.4 File objects File objects represent interactions between an inode and a process. They include useful variables such as `f_pos` (aka the file pointer), which specify where the process is in the file.(Bovet & Marco 2006, p. 471)

3.2.2.5 Process data Processes are associated with `fs_struct` data structures, which contain information such as the working directory of the process (`pwd`).(Bovet & Marco 2006, p. 478)
Processes are also associated with a data structure called `files_struct`, which contains information relating to the files being manipulated and used by it.(Bovet & Marco 2006, p. 479)
`files_struct` contains a pointer to an array known as the `fd_array`, which points to the files in use by the process. As standard, a user-facing process will begin with `stdin`, `stdout`, and `stderr` loaded as files `0`, `1` and `2` in this array.(College n.d.)(Bovet & Marco 2006, p. 479)
Looking at the `proc/PID/fd` directory representing the files accessed by the `less .bashrc` process(Khurana 2022) shows 5 entries. I'm unsure as to what file 3 represents, but file 4 shows `.bashrc`, as shown by figure 4

3.2.3 Finding, opening, and closing the file.

3.2.3.1 Pathname lookup In order to open the `.bashrc` file, a pointer to that file must be resolved. If the pathname doesn't begin with a `/`, as is the case in the report's example, then it is not an absolute path and searching will begin from the working directory associated with the calling process(Bovet & Marco 2006, p. 497) found in the `fs_struct` instance associated with the process as discussed in section 3.2.2.5.

3.2.3.1.1 `link_path_walk()` and verifying file access rights A pointer to the file is then found by running a method called `link_pathname_walk()`. This method recursively searches for the file in question by splitting the path of the desired file into its component parts. For each of these component parts, the restrictions on accessing that directory/file are compared against the permissions of the process which which is attempting to find the file. If the process has insufficient permissions, an error is returned.(Bovet & Marco 2006, p. 498). If no error is returned and all other stages of lookup are successful, the inode and dentry objects of the desired file are stored and the `link_path_walk()` method returns a "no error" signal `0`.(Bovet & Marco 2006, p. 502).

3.2.3.2 `open()` function `open()` is used to open files, and takes the path to the file the user/process wishes to open as an argument.(Bovet & Marco 2006, p. 506) The open function then finds the corresponding file of this path via a pathname lookup as described in section 3.2.3.1(Bovet & Marco 2006, p. 495) If the `open()` function is successful, a new item is added to the `fd_array` of open files (see section 3.2.2.5), and the function returns its index, returning `-1` if the function failed.

3.2.3.3 `read()` The `read()` function is used to read data from an open file within the process. It takes as an argument how much data should be read in bytes, starting from the processes current position in the file specified by `f_pos` (see section 3.2.2.4)(Bovet & Marco 2006, p. 508)

3.2.3.4 `close()` The `close()` function removes the pointer(Bovet & Marco 2006, p. 506) to a specified file from the `fd_array` (see section 3.2.2.5).(Bovet & Marco 2006, p. 509), and sets that entry to null. The file object itself is also deleted(Bovet & Marco 2006, p. 509)

3.2.4 Usage of VFS by the `less` utility

The way that `less` finds, opens, reads and closes the `.bashrc` file in regards to VFS can be broken down most easily into four stages

3.2.4.1 Finding the file in VFS In the first stage, the `open()` function is called, which in turn calls the `link_path_walk()` function to find the dentry and

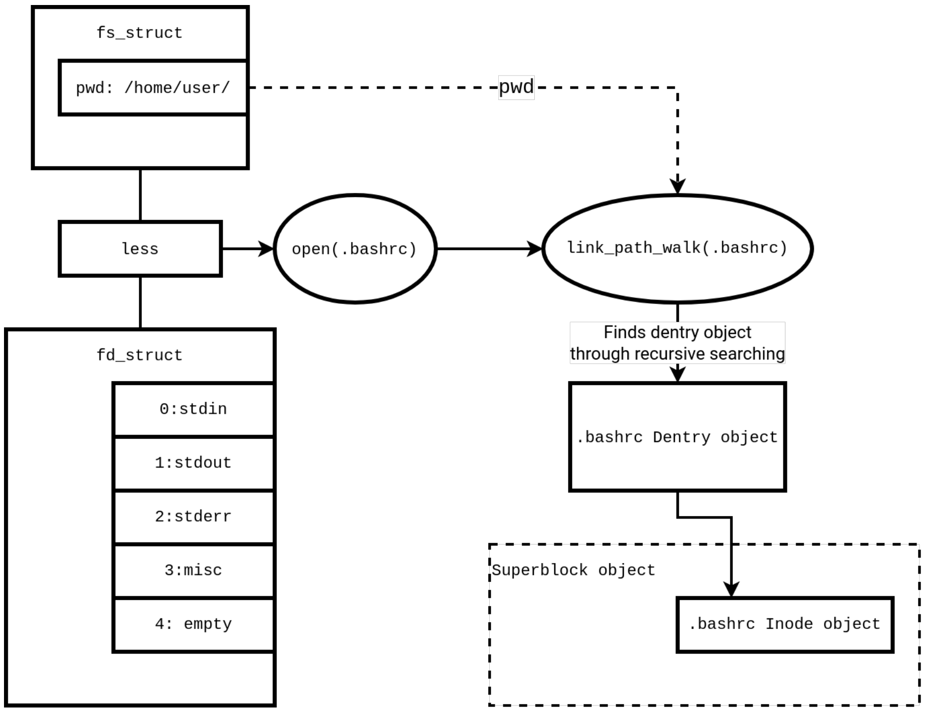


Figure 5: The `open()` function is called, and the correct dentry and inode objects for `.bashrc` are found.

inode objects of `.bashrc`. This process is shown in figure 5, partially inspired by a diagram in *Understanding the Linux Kernel.*(Bovet & Marco 2006, p. 460)

3.2.4.2 Creating the file object In the second stage, this information is returned to the `open()` method, and a new file object is created and placed in the `fd_struct` array of file objects. Figure 6 shows a diagram of this process.

3.2.4.3 Reading from the file object In the third stage, information is read from the file object. Figure 7 shows the process of using the `read()` function(Bovet & Marco 2006, p. 508-509), as discussed in section 3.2.3.3.

3.2.4.4 Closing the file In the fourth stage, the file is closed and removed from the `fd_struct` array. In the case of `less .bashrc` the file will be closed as the `less` process is closed. Figure 8 shows this process as discussed in section 3.2.3.4.

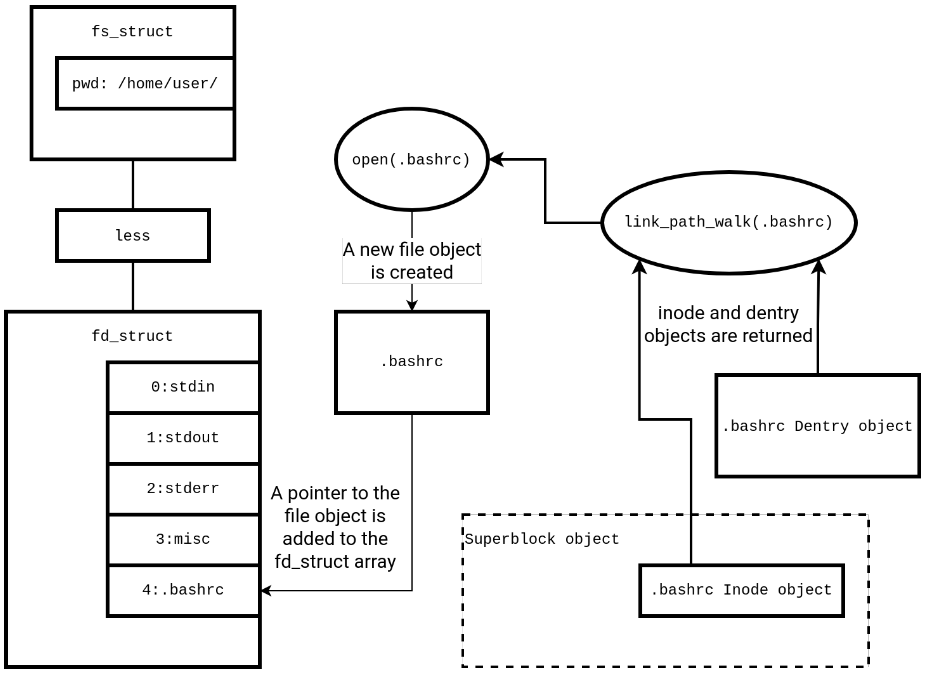


Figure 6: With the correct dentry and inode objects now found, a new file object is created. A pointer to this file object is stored in the `fd_struct` array.

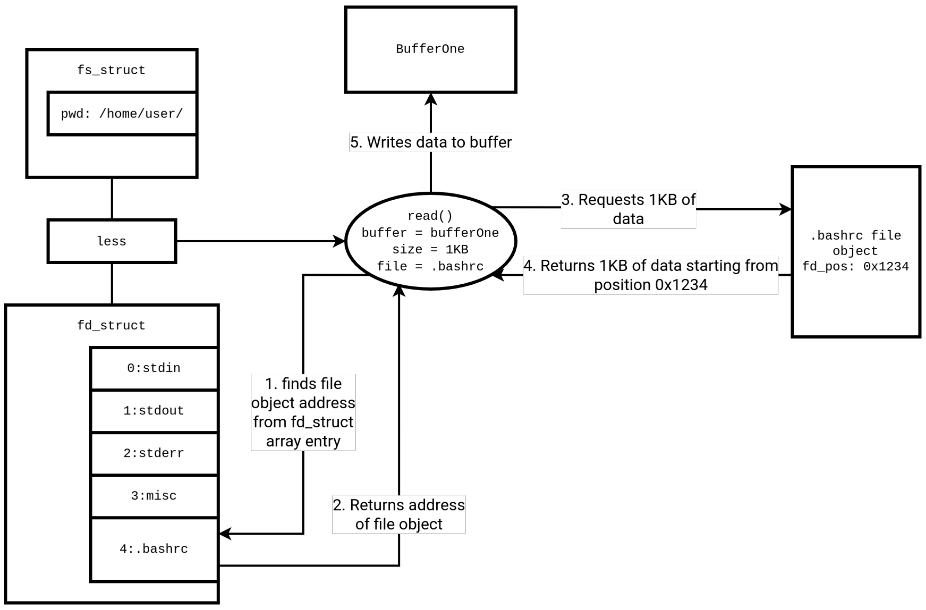


Figure 7: The process of reading from an opened file using the `read()` function.

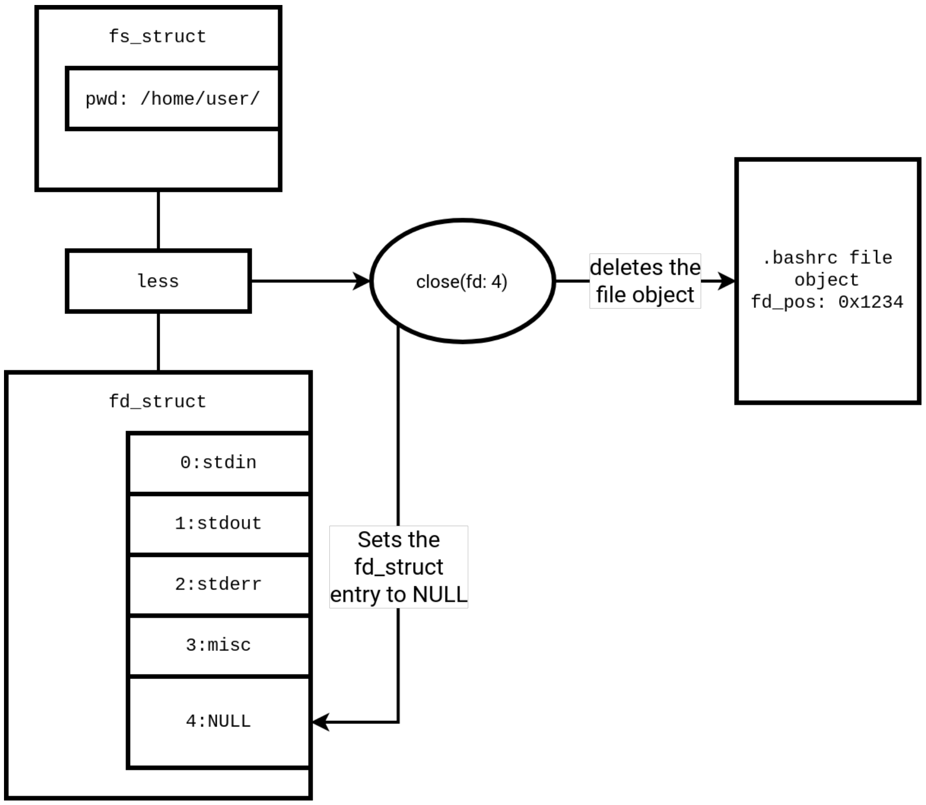


Figure 8: Closing an open file, deleting the file object, and removing its pointer from the array in `fd_struct`

3.3 Process and memory management

3.3.1 Virtual address spaces

processes have access to a virtual allocation of memory which does not reflect how the data is actually stored in RAM or secondary storage.(Chapter 13 Arpaci-Dusseau & Arpaci-Dusseau 2023, p. 4)(Kerrisk 2010, p. 121) This block of virtual memory is called a virtual address space.

3.3.1.1 Process isolation Presenting an abstracted memory containing only a process' own data to each process has the major benefit of preventing processes from accessing each other's data.(Chapter 13 Arpaci-Dusseau & Arpaci-Dusseau 2023, p. 5-6)(Kerrisk 2010, p. 121) This adds a level of defence against malicious processes.

3.3.1.2 Virtual address space sections Process data in Linux is split into 4 main sections:

- `text` contains instructions
- `data` contains statically-defined variables
- The `stack` contains function calls
- The `heap` contains dynamic memory

(Kerrisk 2010, p. 31)

3.3.1.3 Paging In order that a program doesn't take up more memory than it needs to at a given time, it's virtual address space is split up into equally-sized **pages**.(Kerrisk 2010, p. 119)
A **page table** is associated with each process which is used to translate these virtual pages to locations in real memory.(Kerrisk 2010, p. 120) The pages in actual memory are not necessarily unique to a single process, and may be referenced using pages in the virtual memory of multiple processes.(Kerrisk 2010, p. 120)
page faults can occur if a process attempts to access a page which has not been loaded into RAM. If the page exists on the disk, it can be loaded before the program continues.(Kerrisk 2010, p. 119) Page table entries include a reference to the page's location, and whether or not it's in RAM ,alongside other pieces of information about the page.(GeeksForGeeks 2023) If the page is in RAM, then the page table entry contains a reference to that memory location. If the page isn't in RAM, the page table entry contains a **swap entry** which points to the page's location in secondary storage. —In the event of a page fault the page is found using the `do_swap_page()` function.(Linux-Kernel-Community n.d.) Figure 9 shows the process of resolving data from a virtual page.

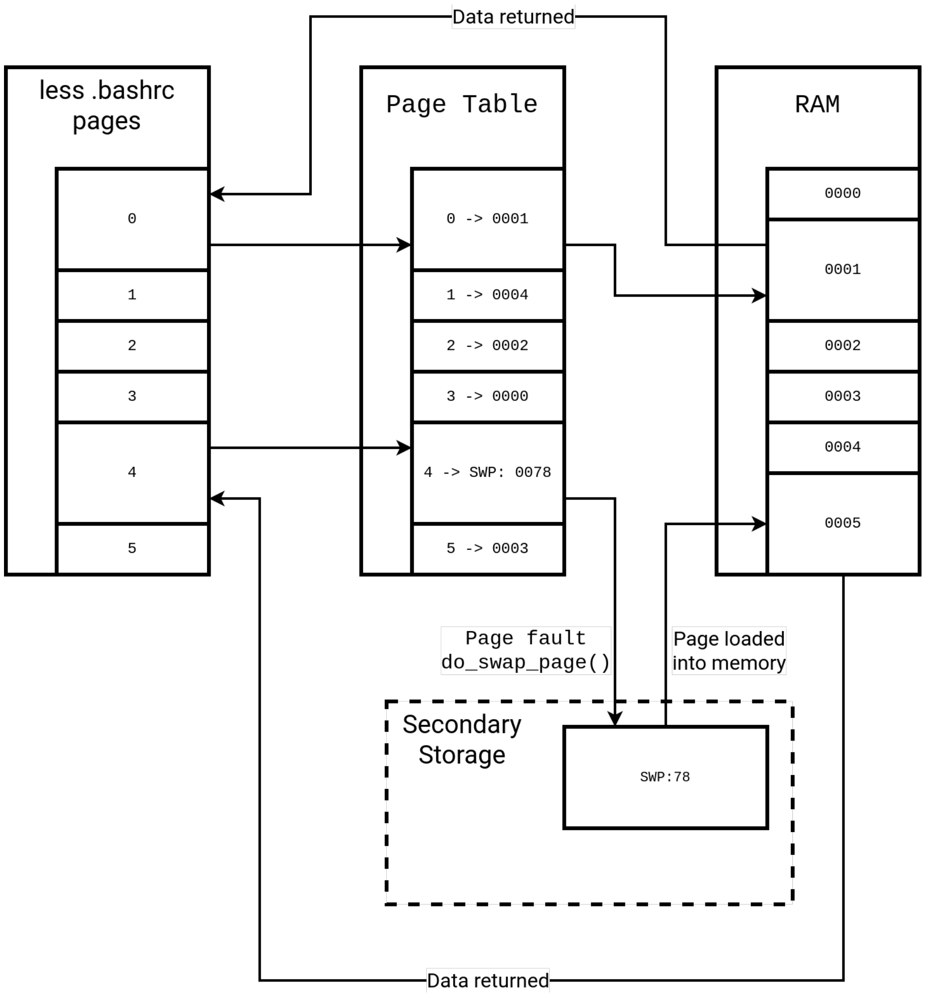
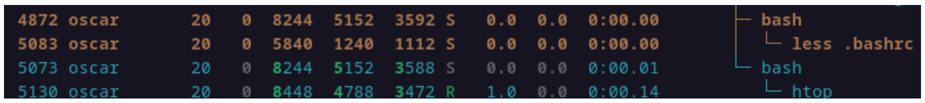


Figure 9: Diagram showing how a page address table finds the real memory addresses of virtual memory pages. The diagram includes an example of a page fault where the page exists in secondary storage but not in RAM.



```
4872 oscar       20   0  8244  5152  3592 S   0.0  0.0  0:00.00          ├ bash
5083 oscar       20   0  5840  1240  1112 S   0.0  0.0  0:00.00          │ └ less .bashrc
5073 oscar       20   0  8244  5152  3588 S   0.0  0.0  0:00.01          └ bash
5130 oscar       20   0  8448  4788  3472 R   1.0  0.0  0:00.14            └ htop
```

Figure 10: Part of a process tree seen using the `htop` utility. The `less .bashrc` process is shown to be a child of the `bash` process.

3.3.2 Running `less` in a child process

In order to run the `less` utility, the bash process must first create a copy of itself.(Bovet & Marco 2006, p. 114) This is done with some variation of the `fork()` system call which creates a new process with a copy of the original process' address space.(Kerrisk 2010, p. 31-32) The child process can then execute `execve()` syscall to load a different program,(Kerrisk 2010, p. 31) replacing its inherited address space(Bovet & Marco 2006, p. 114)
Evidence of this occurring can be seen in figure 10, after `less .bashrc` had been entered.

3.3.2.1 Waiting for `less` to finish executing By waiting for the `less` process to finish executing (see section 3.3.2.2), for example with a variation of the `wait()` system call,(Chapter 5 Arpaci-Dusseau & Arpaci-Dusseau 2023, p. 4)(Matthew & Richard 1997, p. 352) the terminal can return to the bash prompt when the `less` process is finished.

3.3.2.2 `less` termination Processes can either exit intentionally using a variation of the `_exit()` method, or be terminated due to a signal.(Kerrisk 2010, p. 32) If the program runs correctly and exits in a way it expects, which in the case of `less` could be by using the "q" key to quit the program, then the child process will return `0` to any `wait()` syscall made by its parent process.(Kerrisk 2010, p. 32) If the process is terminated by a signal, such as the `SIGINT` signal generated by a keyboard interrupt (`CTRL-C`)(Bovet & Marco 2006, p. 412), a different value will be returned.(Kerrisk 2010, p. 32)
A diagram showing the ways in which the `less` process is created and terminated can be seen in figure 11.

3.4 User interaction

As discussed in section 3.2.2.5, process start with `stdin`, `stdout` and `stderr` file objects representing their input, as well as standard and error-related output. As `less` is a terminal program, the input taken is keyboard input from the user.(Kerrisk 2010, p. 30)(Vona 2019, Chapter Standard Input - STDIN)
Similarly, `stdout` is generally sent to the terminal to be outputted to the user(Vona 2019, Chapter Standard Output - STDOUT)
Despite being a text-based terminal program, `less` is able to accept mouse commands such as scrolling. I'm not completely sure as to how this is achieved using

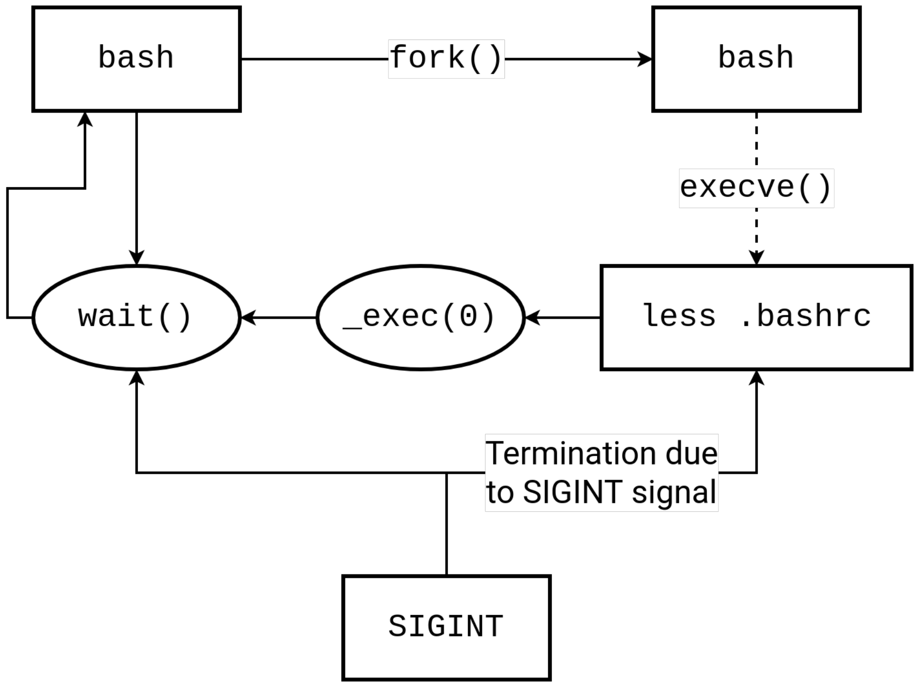
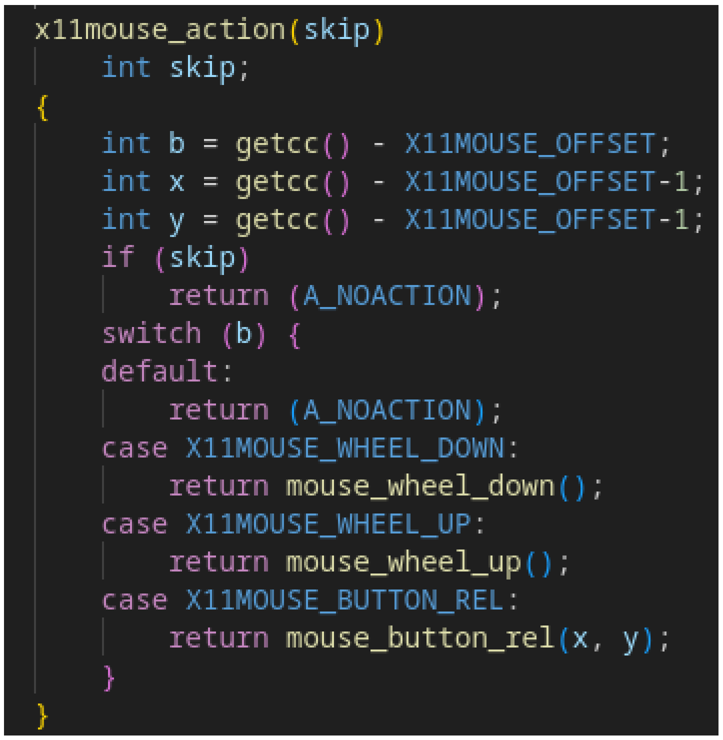


Figure 11: The `bash` process clones itself with `fork()`, before the cloned process switches to `less .bashrc` with an `execve()`-type command. The `less .bashrc` process is then terminated with either a successful exit, or a SIGINT signal, with the result being returned via a `wait()` syscall to the original `bash` process.



```
x11mouse_action(skip)
    int skip;
{
    int b = getcc() - X11MOUSE_OFFSET;
    int x = getcc() - X11MOUSE_OFFSET-1;
    int y = getcc() - X11MOUSE_OFFSET-1;
    if (skip)
        return (A_NOACTION);
    switch (b) {
    default:
        return (A_NOACTION);
    case X11MOUSE_WHEEL_DOWN:
        return mouse_wheel_down();
    case X11MOUSE_WHEEL_UP:
        return mouse_wheel_up();
    case X11MOUSE_BUTTON_REL:
        return mouse_button_rel(x, y);
    }
}
```

Figure 12: References to mouse inputs under the X11 display manager standard in the `less` version 590 source code

the Wayland desktop manager run by default on Debian 12, but I have found reference to mouse input under the X11 display manager in utility's source code. (see figure 12 It's possible that there is some form of emulation re-creating these X11 style inputs.

3.5 System calls

Several functions described in the execution and termination of `less .bashrc`, such as the `read()` function in section 3.2.3.3, are **system calls**. System calls enable a process running with low privileges, such as `less .bashrc` in this case, to request the use of certain restricted functions.(Chapter 6 Arpaci-Dusseau & Arpaci-Dusseau 2023, p. 3) In the case of `read()`, preventing processes from running it arbitrarily means that the permissions of the file being read can be checked against the permissions of the process, throwing an error if the process should not be able to read the file.(Bovet & Marco 2006, p. 509)

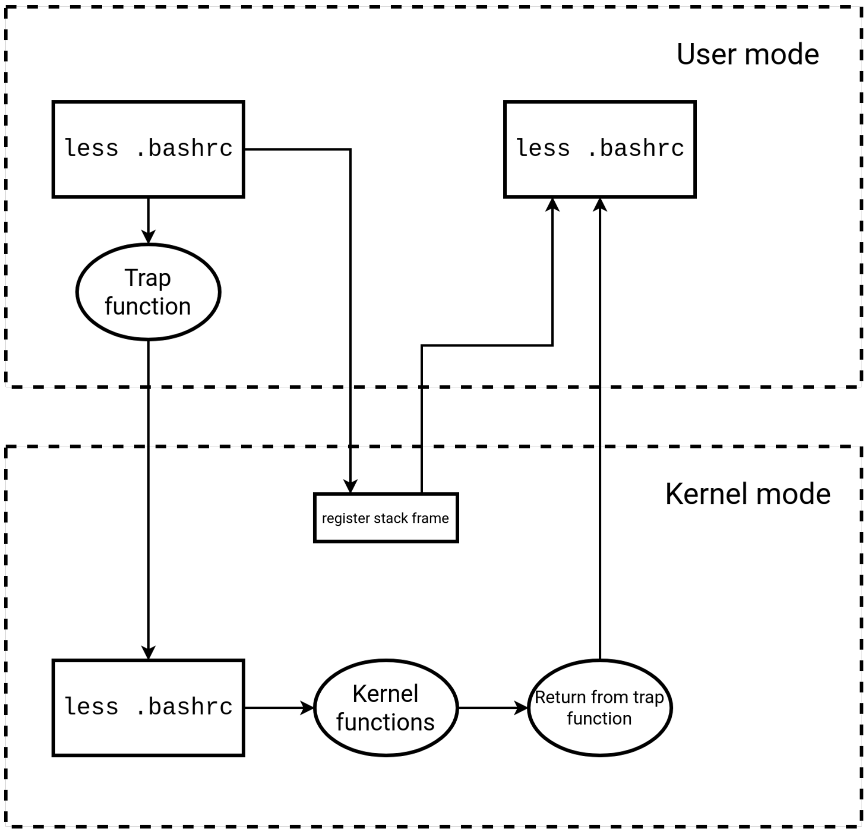


Figure 13: Usage of trap functions, allowing temporary execution of kernel functions in kernel mode

3.5.1 usermode, kernel mode, and traps

System calls rely on two modes of operation, **user mode** and **kernel mode**.
Normal processes run in user mode, and are unable to execute functions which could damage the operating system or other files.
Processes running in kernel mode can execute any function without restrictions.
Trap functions are used to execute system calls. A trap function enters the kernel at a pre-defined point (after saving the original process' registers to **kernel stack**, in order to return to normal execution later) and executes commands in kernel mode, before executing a "return-from-trap" function, returning execution in user mode to the original process.(Chapter 6 Arpaci-Dusseau & Arpaci-Dusseau 2023, p. 4)
Figure 13 shows a diagram of how trap functions are used to enable system calls.

4 Conclusion

Exeuting the command `less .bashrc` requires many intertwining processes to locate data and allocate resources. Although the command is innocuous, some functions in the process of starting the `less` process could prove damaging if abused by a malicious process. To ensure that this is mitigated, a large number of security checks are completed to limit potential damage.

5 List of References

References

Arpaci-Dusseau, R. H. & Arpaci-Dusseau, A. C. (2023), *Operating Systems: Three Easy Pieces*, 1.10 edn, Arpaci-Dusseau Books.

Bovet, D. & Marco, C. (2006), *Understanding the Linux Kernel*, 3 edn, O'Reilly.

College, S. (n.d.), 'File descriptors'.
URL: *https://www.cs.swarthmore.edu/ kwebb/cs31/s15/bucs/file_descriptors.html*

Debian (2023*a*), 'Debian displaymanagers'.
URL: *https://wiki.debian.org/DisplayManager*

Debian (2023*b*), 'environ(7)'.
URL: *https://manpages.debian.org/bookworm/manpages/environ.7.en.html*

GeeksForGeeks (2022), 'Soft and hard links in unix/linux'.
URL: *https://www.geeksforgeeks.org/soft-hard-links-unixlinux/*

GeeksForGeeks (2023), 'Page table entries in page table'.
URL: *https://www.geeksforgeeks.org/page-table-entries-in-page-table/*

Kerrisk, M. (2010), *The Linux Programming Interface*, No Starch Press, Incorporated.
URL: *https://ebookcentral.proquest.com/lib/warw/detail.action?docID=1137549*

Khurana, S. (2022), 'Developer diaries: Processes, files, and file descriptors in linux'.
URL: *https://medium.com/geekculture/developer-diaries-processes-files-and-file-descriptors-in-linux-ebf007fb78f8*

Linux-Kernel-Community (n.d.), 'Chapter 3 page table management'.
URL: *https://www.kernel.org/doc/gorman/html/understand/understand006.html*

Matthew, N. & Richard, S. (1997), *Beginning Linux Programming*, Wrox Press Limited.

Senofsky, M., Dietl, P. & Schwalm, K. (2022), 'Introduction to the linux virtual filesystem (vfs): A high-level tour'.
URL: *https://www.starlab.io/blog/introduction-to-the-linux-virtual-filesystem-vfs-part-i-a-high-level-tour*

Vona, S. (2019), 'Linux fundamentals - i/o, standard streams, and redirection.'.
URL: *https://www.putorius.net/linux-io-file-descriptors-and-redirection.html#*

Wiki, D. (2023).
URL: *https://wiki.debian.org/EnvironmentVariables*