“less .bashrc”: a process lifecycle, and a dive into operating system functionality

Peter Walker — u5539006

May 2024

Contents

1 Introduction

This is a report that follows the lifecycle of a process, triggered from the command line: “less .bashrc”. It covers the initialisation of the process, and follows the entire process through to the end, covering CPU scheduling and states, I/O requests, and security measures to provide a clear overview of the process, while maintaining a level of technical detail. “less” is an executable that allows the user to view the contents of a file in an interactive window, with a lazy-loading functionality to decrease performance issues when compared to traditional methods that parse the entire file to be read before displaying.

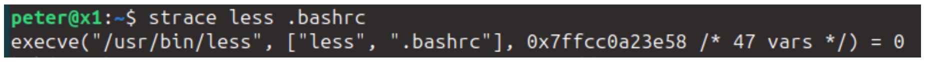
2 Process Initialisation

When a command is entered at the command line, it has to be parsed by the terminal before it can be executed. Recognition of command arguments use "token recognition": a set of rules that govern the standard form of commands [1]. When a user enters the command "less .bashrc" the terminal will read the input through a system call, then recognise the first argument ("less") as the executable (the command) and ".bashrc" as the first argument, and in the case of "less", as a filename. This can be seen by checking the system calls for the command, which includes all arguments separated, as in figure 1.

The terminal then looks for the executable "less" using the PATH environmental variable, and the current directory. By default, the pre-packaged binary is located at "/usr/bin/less", and the parent folder is usually located in the PATH: see figure 2. If not, the absolute path to the executable binary would need to be specified: "/usr/bin/less .bashrc". All folders contained in the PATH variable are checked for executables when commands are inputted, after checking the current working directory.

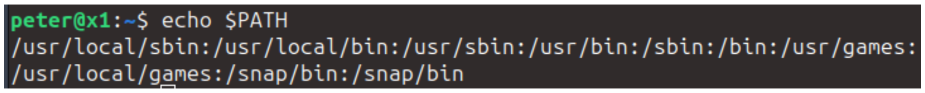
Before an executable can be executed, the current user, or effective user, needs to have the relevant permissions. Specifically, the execute permission bit needs to be set for the current user, or for a group that the current user is in. By default, all standard users have executable permission for "/usr/bin/less", as demonstrated in figure 3. It is clear that the current user has executable permission due to the "x" in the 4th position of the permissions of the executable. This "x" also appears in the 7th and 10th positions, which means any user can run the executable [2].

The parent process, the terminal, forks to create a child which will run the



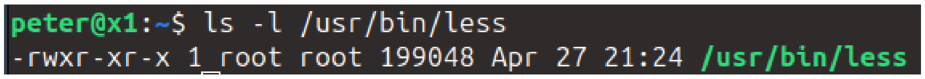
```
peter@x1:~$ strace less .bashrc
execve("/usr/bin/less", ["less", ".bashrc"], 0x7ffcc0a23e58 /* 47 vars */) = 0
```

Figure 1: The arguments of the command "less .bashrc" can be seen identified and split up using token recognition, as ["less", ".bashrc"], when tracing the system calls of the command.



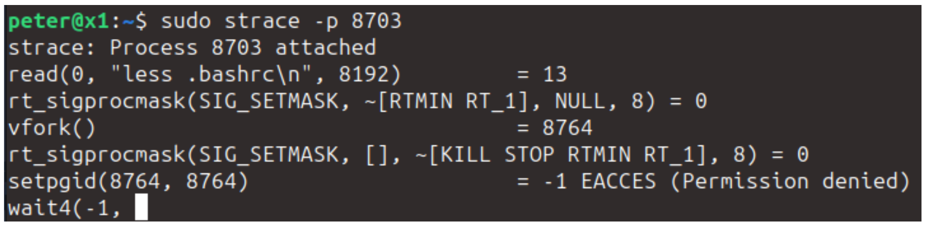
```
peter@x1:~$ echo $PATH
/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin:/usr/games:
/usr/local/games:/snap/bin:/snap/bin
```

Figure 2: The PATH variable on a standard Ubuntu installation. "/usr/bin" is the fourth section of the variable.



```
peter@x1:~$ ls -l /usr/bin/less
-rwxr-xr-x 1 root root 199048 Apr 27 21:24 /usr/bin/less
```

Figure 3: The default permissions for "/usr/bin/less" on a standard Ubuntu installation.



```
peter@x1:~$ sudo strace -p 8703
strace: Process 8703 attached
read(0, "less .bashrc\n", 8192)         = 13
rt_sigprocmask(SIG_SETMASK, ~[RTMIN RT_1], NULL, 8) = 0
vfork()                                 = 8764
rt_sigprocmask(SIG_SETMASK, [], ~[KILL STOP RTMIN RT_1], 8) = 0
setpgid(8764, 8764)                     = -1 EACCES (Permission denied)
wait4(-1,
```

Figure 4: The system call trace of the "less .bashrc" command which shows a syscall of "vfork()": a more secure variation of "fork()", and the "wait4()" syscall.

executable. The system call (syscall) "vfork()" is used to create a child process (see figure 4) [3]. The parent process then calls the "wait4()" syscall (figure 4) which attaches to a certain child process and returns additional resource usage information compared to "waitpid()" [4].

The first syscall of the child process is an "execve()" which executes a program, and replaces the calling process (see figure 1). This replaces the bash terminal child process (because it is initially identical to the parent process) with the program code of the "less" executable [5]. This overwrites the memory instructions, libraries, and re-initialises the stack and heap sections of the address space for this process. The arguments shown in figure 1 (in this case, ".bashrc") are passed to this new executable.

3 Process Management

CFS (Completely Fair Scheduler) is likely the scheduler used for the duration of this process. This is a scheduler that still prioritises certain tasks, but ensures all tasks have fair amounts of time to execute. Each process, as it runs, accumulates virtual runtime (vruntime). The amount of vruntime the process accumulates is proportional to the priority of the process. (Higher priority processes accumulate less vruntime per unit time). Within a certain amount of time, called the *target latency*, every process will be executed, for a minimum time equal to the *minimum granularity* of the operating system. Tasks which have accumulated the least vruntime are prioritised, and accordingly placed on the left-hand side of the red-black tree: a self-balancing, logarithmic-searchable data structure used to store processes according to their vruntime [6, 7].

Any process currently executing has a process state: running, ready or blocked. Only one process has the "running" state, which is the process currently executing. Any process with the "ready" state is waiting to use the CPU and it not waiting on any other operation to complete. "Blocked" is the state for processes that are waiting for another operation to complete or event to happen before they can execute again, such as if they're waiting to receive a response for an I/O request: waiting for keyboard input, or to read the contents of a file, for example. A process can have the state "completed" briefly before it is killed and its resources are freed, once its execution has concluded. See figure 6.

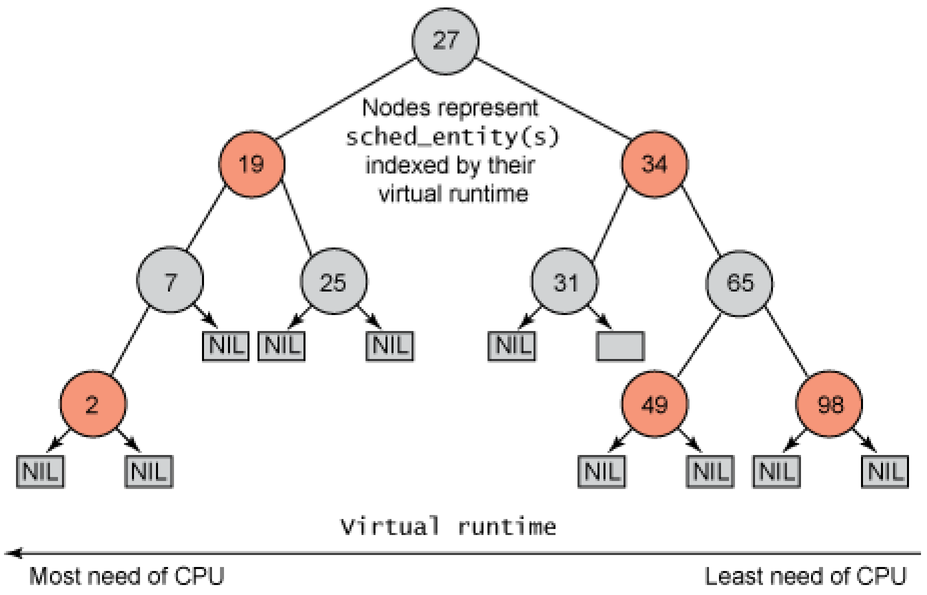


Figure 5: An example of a red-black tree, used for CFS to store processes according to vruntime. Source: IBM [8]

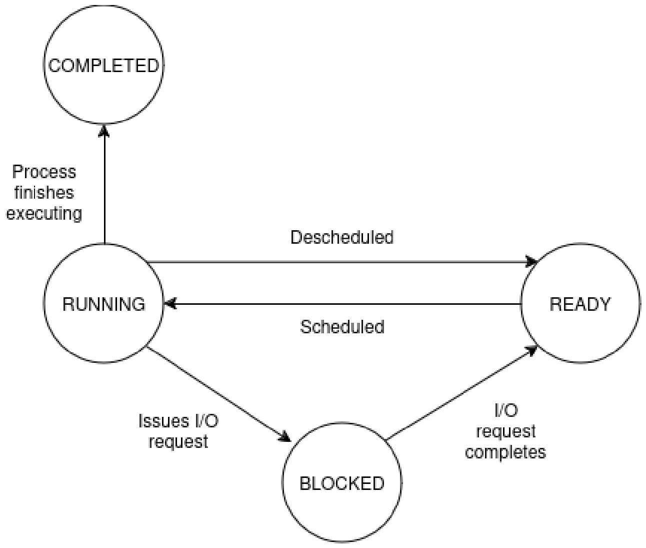
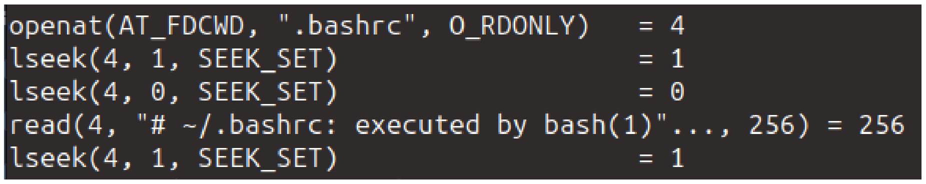


Figure 6: A diagram illustrating the transition between process states.



```
openat(AT_FDCWD, ".bashrc", O_RDONLY)   = 4
lseek(4, 1, SEEK_SET)                   = 1
lseek(4, 0, SEEK_SET)                   = 0
read(4, "# ~/.bashrc: executed by bash(1)"..., 256) = 256
lseek(4, 1, SEEK_SET)                   = 1
```

Figure 7: The "openat()" syscall traced through the execution of "less .bashrc".

Any command that needs to be run by the "less .bashrc" process has to be scheduled for execution by the CPU, using CFS. For example, this could be a "openat()" syscall, to access subsequent pages of the file that is visible to the user, as seen in figure 7. This syscall returns a file descriptor of the given file. The CPU will issue an I/O request (Input/Output request) to the file system to read the metadata of the filesystem. For the duration of this request, the process will be in an "I/O request" state, until the file is returned. The CPU will, once this process is scheduled, read the metadata to identify the physical location of the file. Another I/O request will be issued to retrieve the permissions of the file. Once this completes, and the process is executed again, the permissions will be checked to determine whether the current user can access that file. If so, a descriptor for the file is returned to the "less" executable [9].

4 Memory Management

Each process has a certain amount of allocated memory: its address space. This contains the program code for the process, the necessary libraries, a stack and a heap. The process isn't allowed to access any memory locations beyond its address space, and the bounds register on the CPU is used to ensure that no process can read or write outside its allocated memory.

A process' address space can have ASLR (Address Space Layout Randomisation) enabled, which causes the relative locations of program code, libraries, stack and heap inside the address space to be reordered and placed non-consecutively.

The memory of the device is split into physical page frames: fixed-size chunks of data, and the address space for processes are split across multiple pages (each of which fills a page frame). The OS stores a table, per-process, of the pages used for the address space, and their physical memory addresses. This allows the address space to be split up into non-continuous page frames which helps reduce wasted allocated memory [10].

Depending on the size of the address spaces, and the number of necessary pages, several bits will be allocated to identify both the number of the page and the location of the necessary data within a given page (the offset). This means that any virtual address supplied by the process can be accurately converted to a physical memory address by the CPU [10].

In the context of "less .bashrc", the main use of memory is to store some of the content of the given file. This data is stored in a buffer, allocated using the "malloc()" function, which creates a buffer of the specified size on the heap. The heap is used to store user-allocated data structures.

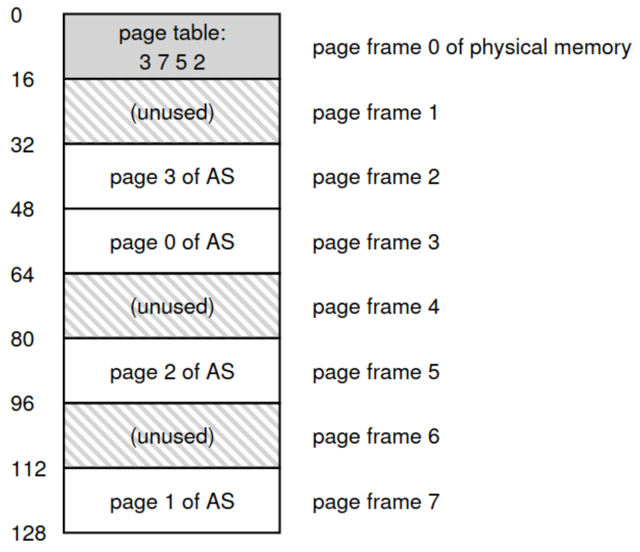


Figure 8: An example of paging for an address space. Source: Operating Systems: Three Easy Pieces [10].

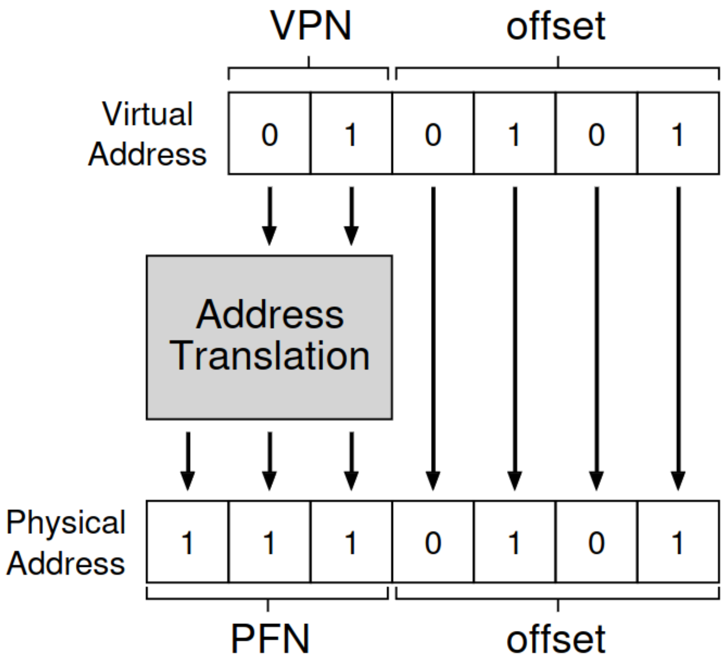


Figure 9: An example of address translation with pages. Source: Operating Systems: Three Easy Pieces [10]

5 “less .bashrc”: an overview of runtime

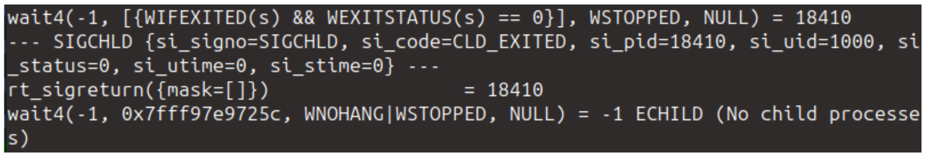
The command is ran and is initialised as given in Section 2. In addition, a buffer is allocated to store partial contents of the file, and the internal state of “less” is initialised, including a pointer to the current read position of the file, and the dimensions of the terminal window, which are stored as LINES and COLUMNS shell variables [11].

The “less” binary uses the syscall “openat()” to return a file descriptor, as is laid out in Section 3. Afterwards, the syscall “lseek()” is used to change the offset of the file descriptor to move it to a relevant location within the file. Finally the “read()” syscall is used to read the content of the file. This data is stored in the buffer allocated when the program is initialised [11].

The contents of the buffer containing the first portion of the file, as much as can be displayed in the window, are written into the terminal using the “write()” syscall. Afterwards, a loop is started where the process waits for keyboard input to control the process; using keyboard shortcuts to navigate the file. For example, a user could use the shortcut *Ctrl+G* to jump to the end of the file.

For the duration of the user-input loop, signal handlers are in place to catch exceptions and increase interactivity. If the “less” process receives the “SIGINT” signal, which is usually if the user presses *Ctrl+C* to force exit the process, the process will not exit but instead jump to the input prompt, allowing the user to enter the exit command if they want to. If the process receives the “SIGWINCH” signal, then the terminal has been resized. The “write()” syscall is called again to format the data into the resized terminal [11].

When the user presses “q” in the “less” process, it will quit. This causes the memory buffer storing the file to be freed with the “free()” function, which causes the given memory to be unallocated. After completing, the parent process, the shell, will receive a “SIGCHLD” signal (see figure 10 which informs it that the child process has exited. Since the shell was waiting for the child process to conclude, the “wait4()” syscall will be satisfied, and the user will be able to access the terminal for commands again [11].



```
wait4(-1, [{WIFEXITED(s) && WEXITSTATUS(s) == 0}], WSTOPPED, NULL) = 18410
--- SIGCHLD {si_signo=SIGCHLD, si_code=CLD_EXITED, si_pid=18410, si_uid=1000, si
_status=0, si_utime=0, si_stime=0} ---
rt_sigreturn({mask=[]})                 = 18410
wait4(-1, 0x7fff97e9725c, WNOHANG|WSTOPPED, NULL) = -1 ECHILD (No child processe
s)
```

Figure 10: A trace of the command’s execution including the “SIGCHLD” signal and the “wait4()” syscall being satisfied.

6 User and Kernel Space

System calls (syscalls) are functions that allow the user space to interface and talk to the kernel space. The user space is where the user is allowed to execute commands and user applications can run, in a limited-access environment. On the other hand, the kernel space is an unrestricted environment, which is managed by the operating system. Syscalls are a means of asking the operating system to perform certain predefined functions to perform critical operations on system files or devices, without giving user applications the unrestricted ability to do so.

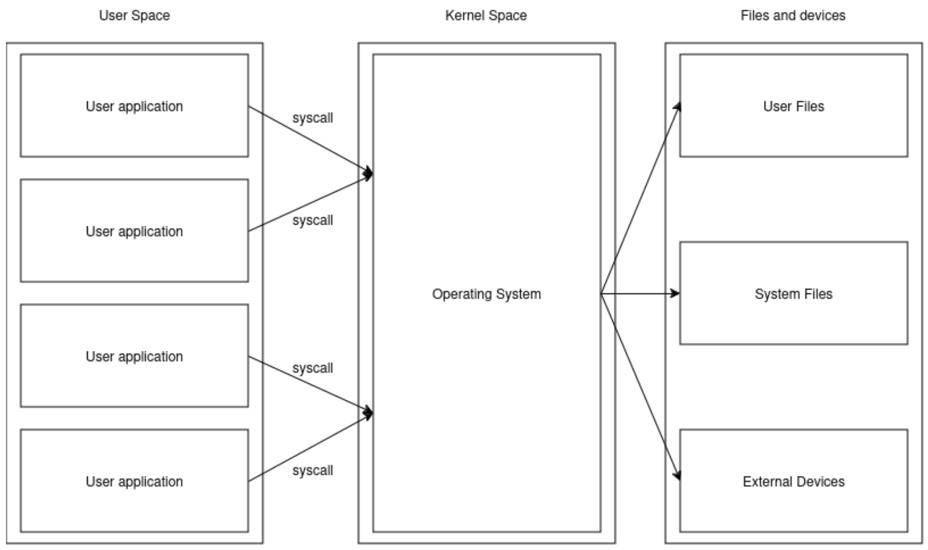


Figure 11: The user space is only allowed to access files and devices by going through the operating system which operates in the kernel space.

7 Security

ASLR can be implemented on systems to move the components of a process around randomly within its address space. This helps obscure where critical parts of the program are stored when the executable is running, which makes it harder for hackers to run successful overflow attacks.

The implementation of user space and kernel space allows the OS to manage critical operations and applications to perform as intended without compromising security by giving applications direct access to files or devices.

File permissions stop unauthorised changes being made to the "less" binary, being replaced by a malicious executable, or one being placed earlier in the PATH. This is because non-superusers do not have write permission to binary folders that aren't per-user.

The file to be read has to be loaded into the heap. It is therefore recommended to ensure that the Non-eXecutable bit is enabled for the heap, as this will stop code loaded from the file being potentially executed by a malicious attacker. Ideally, this should be enabled on all memory pages that don't contain executable code.

The "less" binary should be compiled with PIE (position independent executable) enabled, as this will allow the randomisation of executable code and shared libraries within the address space. Without this, ASLR can only randomise locations of other components such as the stack and the heap [12].

8 Conclusion

In conclusion, the command “less .bashrc” and the process that it triggers, provide a responsive and secure interface for the user, and interfaces with the OS in a standard and reliable manner. This report provided a clear technical overview of the process and memory management of this process and as part of the overall system.

References

[1] Shell Command Language — The Single Unix. Available at: https://pubs.opengroup.org/onlinepubs/7908799/xcu/chap2.html#tag_001_003 (Accessed: 14 May 2024).

[2] Gupta, U. (2023) File permissions and Access Control lists — LinkedIn. Available at: https://www.linkedin.com/pulse/file-permissions-access-control-lists-unnati-gupta- (Accessed: 14 May 2024).

[3] vfork (2) - Linux manual page — man7. Available at: https://man7.org/linux/man-pages/man2/vfork.2.html (Accessed: 14 May 2024).

[4] wait4 (2) - Linux manual page — man7. Available at: https://man7.org/linux/man-pages/man2/wait4.2.html (Accessed: 14 May 2024).

[5] execve (2) - Linux manual page — man7. Available at: https://man7.org/linux/man-pages/man2/execve.2.html (Accessed: 14 May 2024).

[6] Rajam, M.A. (2018) Scheduling in Linux, INFLIBNET Centre Gandhinagar. Available at: https://ebooks.inflibnet.ac.in/csp3/chapter/scheduling-in-linux/ (Accessed: 14 May 2024).

[7] CFS Scheduler — The Linux Kernel Archives. Available at: https://www.kernel.org/doc/Documentation/scheduler/sched-design-CFS.txt (Accessed: 14 May 2024).

[8] Jones, M. (2018) Inside the Linux 2.6 Completely Fair Scheduler, IBM developer. Available at: https://developer.ibm.com/tutorials/l-completely-fair-scheduler/ (Accessed: 14 May 2024).

[9] openat(2) - Linux manual page — man7. Available at: https://man7.org/linux/man-pages/man2/openat2.2.html (Accessed: 15 May 2024).

[10] Arpaci-Dusseau, R.H. Paging — Operating Systems: Three Easy Pieces. Available at: https://pages.cs.wisc.edu/ remzi/OSTEP/vm-paging.pdf (Accessed: 15 May 2024).

[11] less (1) - Linux manual page — man7. Available at: https://man7.org/linux/man-pages/man1/less.1.html (Accessed: 15 May 2024).

[12] PIE (2021) — Binary Exploitation. Available at: https://ir0nstone.gitbook.io/notes/types/stack/pie (Accessed: 15 May 2024).

A Appendix: Earliest Eligible Virtual Deadline First Scheduling

"Earliest Eligible Virtual Deadline First Scheduling" is a scheduling algorithm, first designed and documented in 1995 by Ion Stoica and Hussein Abdel-Wahab at the Old Dominion University in Virginia, USA.

This algorithm is being adopted as the scheduler in recent versions of Linux, with kernel versions ¿= 6.6.0, including Arch, Fedora, Tumbleweed and the new LTS Ubuntu distribution: 24.04 (as of April 25th 2024!)

It is an algorithm that bears resemblance to CFS but also includes the implementation of virtual deadlines.

Each process is assigned a weight that reflects the priority of the process, and all processes request CPU time based on their weight and accumulated virtual runtime. Virtual runtime is accumulated with respect to weight, as in CFS. Each process is also assigned a virtual deadline: a time by which the request would be completed in an ideal system. This calculation factors in the weight and previously accrued virtual runtime of the process. The scheduler then prioritises requests based on virtual deadlines, and time quantums (CPU time slots) are allocated to requests with a closer virtual deadline or those falling behind their CPU time allocation.

It uses augmented binary search trees to store the processes, which keeps the search complexity at O(log n). The type of tree is not explicitly stated, but they are very similar to the CFS red-black trees, so they could be the same.

I thought it appropriate to include this information since this is the scheduling algorithm included in new versions of Linux. However, there is no extensive documentation or reports on it yet, so it is included as an appendix.

A.1 References

Stoica, I. and Abdel-Wahab, H., 1995. Earliest eligible virtual deadline first: A flexible and accurate mechanism for proportional share resource allocation. Old Dominion Univ., Norfolk, VA, Tech. Rep. TR-95-22.

Vaughan-Nichols, S. (2023) Linux kernel 6.6 is the next long-term support release, ZDNET. Available at: https://www.zdnet.com/article/linux-kernel-6-6-is-the-next-long-term-support-release/ (Accessed: 15 May 2024).