Contents

[Component 3 1](#_Toc156238816)

[part 1 1](#_Toc156238817)

[Part 2 2](#_Toc156238818)

[part 3 3](#_Toc156238819)

[Component 4 4](#_Toc156238820)

[Exploitation 4](#_Toc156238821)

[References 8](#_Toc156238822)

[Appendix 9](#_Toc156238823)

[Appendix 1 9](#_Toc156238824)

[Appendix 2 11](#_Toc156238825)

[Gitlab link 13](#_Toc156238826)

# Component 3

## part 1

Linux: is an open-source operating system that shares many of the same system calls as Unix because it is founded on the Unix concept. Calls for file operations, process management, and other things are included in this (Babayo and Haruna, 2023).

Windows: This Microsoft-developed operating system features a unique collection of system calls that are a component of the Windows API (WinAPI). A distinct user and developer ecosystem is the focus of the design concepts and philosophy (Babayo and Haruna, 2023).

Linux: Ensures backward compatibility by using a consistent system call interface that remains unchanged across versions. The man pages contain extensive documentation on the system calls (Babayo and Haruna, 2023).

Windows: Versions of Windows can affect system calls, which are frequently accessed through the WinAPI. Backward compatibility is preserved through a number of techniques, including Windows Compatibility Mode (Babayo and Haruna, 2023).

Linux: runs on a Monolithic Kernel. In a monolithic kernel architecture, every core system service, including drivers, file system management, and system server processes, are integrated into one large kernel space. This means they all have direct access to the hardware and can execute processes efficiently. However, this tight integration can lead to increased complexity and potential stability issues, as a bug in one service can affect the entire system (GeeksforGeeks, 2024).

Windows: runs on a Microkernel. A microkernel architecture takes a minimalistic approach. In this setup, only the most essential services, such as memory management, process scheduling, and inter-process communication, are included in the kernel. These core components are designed to be small and efficient. All other system services, like device drivers, file systems, and network interfaces, run in user space as separate processes. This separation enhances system stability and security since a failure in one of these user-space services doesn't necessarily compromise the entire system. Additionally, it can lead to easier maintenance and updating of individual components without affecting the kernel directly. However, the trade-off is often in performance, as interactions between user space and kernel space can introduce overhead (GeeksforGeeks, 2024).

## Part 2

The Following table contains some of the system calls Linux uses in the commands cp, rm, and cat, they were extracted from the Linux terminal using the strace command which gives the command’s intercepts and records any system calls.

|  |  |  |  |
| --- | --- | --- | --- |
| Linux System Calls | | | |
| System calls | cp | rm | cat |
| execve | Y | Y | Y |
| brk | Y | Y | Y |
| access | Y | Y | Y |
| mmap2 | Y | Y | Y |
| open | Y | Y | Y |
| fstat64 | Y | Y | Y |
| read | Y | Y | Y |
| munmap | Y | Y | Y |
| write | Y | Y | N |
| \_llseek | Y | Y | N |
| exit\_group | Y | Y | N |
| rt\_sigaction | Y | N | N |
| rt\_sigprocmask | Y | N | N |
| ugetrlimit | Y | N | N |
| uname | Y | N | N |
| statfs64 | Y | N | N |
| open | Y | N | N |
| mmap2 | Y | N | N |
| geteuid32 | Y | N | N |
| mprotect. | Y | N | N |
| set\_tid\_address | Y | N | N |
| set\_robust\_list. | Y | N | N |
| close | N | Y | Y |
| mprotect | N | Y | Y |
| set\_thread\_area | N | Y | Y |
| ioctl | N | Y | N |
| fstat64 | N | N | Y |
| fadvise64\_64 | N | N | Y |
| SIGWINCH | N | N | Y |
| read | N | N | Y |

Table 1: Linux System Calls

execve: This system call is used to execute a new program. It replaces the current process image with a new process image, loading the program into memory and starting its execution (man7.org, n.d.-a).

brk and mmap2: These are related to memory management. brk changes the location of the end of the process's data segment (effectively resizing the heap). mmap2 maps files or devices into memory, providing a method for file I/O (man7.org, n.d.-b), (man7.org, n.d.-c).

open and close: Used to open and close a file descriptor. open is used to open files, devices, or other resources, and close is used to free the file descriptor for reuse (man7.org, n.d.-d), (man7.org, n.d.-e).

fstat64: Retrieves information about a file, based on its file descriptor. This can include details like file size, permissions, and timestamps (linux.die.net, n.d.).

read and write: Fundamental for reading from and writing to file descriptors. This can include files, devices, pipes, etc (man7.org, n.d.-f), (man7.org, n.d.-g).

munmap: Unmaps a region of memory that was previously mapped by mmap2, effectively freeing that portion of memory (man7.org, n.d.-h).

exit\_group: Exits all threads in a process. This is typically used to end multi-threaded processes (man7.org, n.d.-i).

## part 3

To show the deference between Linux and Windows, the equivalent commands in Windows are copy, del, and type, and compared the system calls as shown in the tables below.

|  |  |  |  |
| --- | --- | --- | --- |
| Windows System Calls | | | |
| System calls | copy | del | type |
| CreateFile | Y | Y | Y |
| CloseHandle | Y | Y | Y |
| WriteFileEx | Y | N | N |
| FlushFileBuffers | Y | N | N |
| ReadFileEx | Y | N | N |
| opyFileEx | Y | N | N |
| FindNextFile | N | Y | N |
| GetLastError | N | Y | N |
| RemoveDirectory | N | Y | N |
| CreateFile | N | Y | Y |
| DeleteFile | N | Y | Y |
| SetFileAttributes | N | Y | Y |
| ReadFile | N | N | Y |
| WriteConsole | N | N | Y |
| SetFilePointer | N | N | Y |
| GetConsoleMode | N | N | Y |

Table 2: Windows System Calls

The table below shows the Linux equivalent for each system call.

|  |  |
| --- | --- |
| Windows System Calls | Linux System Calls |
| CreateFile | open |
| CloseHandle | close |
| ReadFile | read |
| WriteFileEx | write |
| FlushFileBuffers | fsync |
| CopyFileEx | [No direct equivalent, uses cp command] |
| RemoveDirectory | rmdir |
| GetLastError | [No direct equivalent, Linux returns error codes directly] |
| DeleteFile | unlink |
| SetFileAttributes | chmod |
| ReadFileEx | pread64 |
| WriteConsole | write (to console file descriptor) |
| SetFilePointer | lseek |
| GetConsoleMode | ioctl |
| execve | execve |
| brk | brk |
| mmap2 | mmap |
| fstat64 | fstat |
| munmap | munmap |
| \_llseek | lseek |
| exit\_group | exit |
| rt\_sigaction | rt\_sigaction |
| rt\_sigprocmask | rt\_sigprocmask |
| ugetrlimit | getrlimit |
| uname | uname |
| statfs64 | statfs |
| mprotect | mprotect |
| set\_tid\_address | set\_tid\_address |
| set\_robust\_list | set\_robust\_list |
| ioctl | ioctl |
| fadvise64\_64 | posix\_fadvise |

Table 3: Linux Equivalent System Calls

# Component 4

## Exploitation

In this example Buffer Overflow have been used and will be demonstrated, but to understand Buffer Overflow, there should be a basic knowledge on how the memory works so the following figure shows the ram memory.

A black and white diagram

Description automatically generated with medium confidence

Figure 1: RAM Memory

Figure 1 illustrates the data organization within RAM memory, specifically highlighting the growth directions of the Heap and Stack. While both structures are integral to memory management, this example concentrates on the Stack, particularly in the context of buffer overflow vulnerabilities. It's important to note that buffer overflows can occur in various memory areas, but this example emphasizes their manifestation in the Stack.

A screen shot of a computer screen

Description automatically generated

Figure 2: Stack Diagram

Figure 2 Illustrates what is inside the stack, the Stack shows the Buffer where the Buffer Overflow happens, Buffer Size differs based on the program coded.

The cp command was chosen to exploit two versions of the code was produced one of them was made in which it cannot be exploited using buffer overflow, the code uses limit\_size to check for the limit size, the read and write system calls are used with the buffer size as one of the parameters. This ensures that no more than buffer\_size bytes are read from the source file and written into the buffer in each iteration of the loop. Thus, the data copied into the buffer is controlled and should not exceed its capacity, and the code checks the number of bytes read with n\_read = read(src\_fd, buffer, buffer\_size). If read returns a value greater than the buffer size, it indicates an error. In this code, since read is called with buffer\_size, it will not read more bytes than the buffer can hold.

This is the first version of the code can be found in Appendix 1.

The second version, the protection parameters have been removed so that the code can be exploited using buffer overflow, and a fixed buffer has been introduced so that buffer cannot be controlled by the program, the fixed buffer has been initialized with a smaller buffer size, so it does not need a large file for the program to be exploited.

This is the second version of the code can be found in Appendix 2.

The second version of the code was run in Linux using the Linux gdb debugger, the gdb debugger shows all the details of the code when is being ran, so the segmentation faults can be spotted, a segmentation fault is what the CPU will send back when the user is trying to access something the user should not in the memory, so in that case we can overwrite the memory, and since gdb is being used the user can see the machine code of the written script using disas main which views the code for the main.

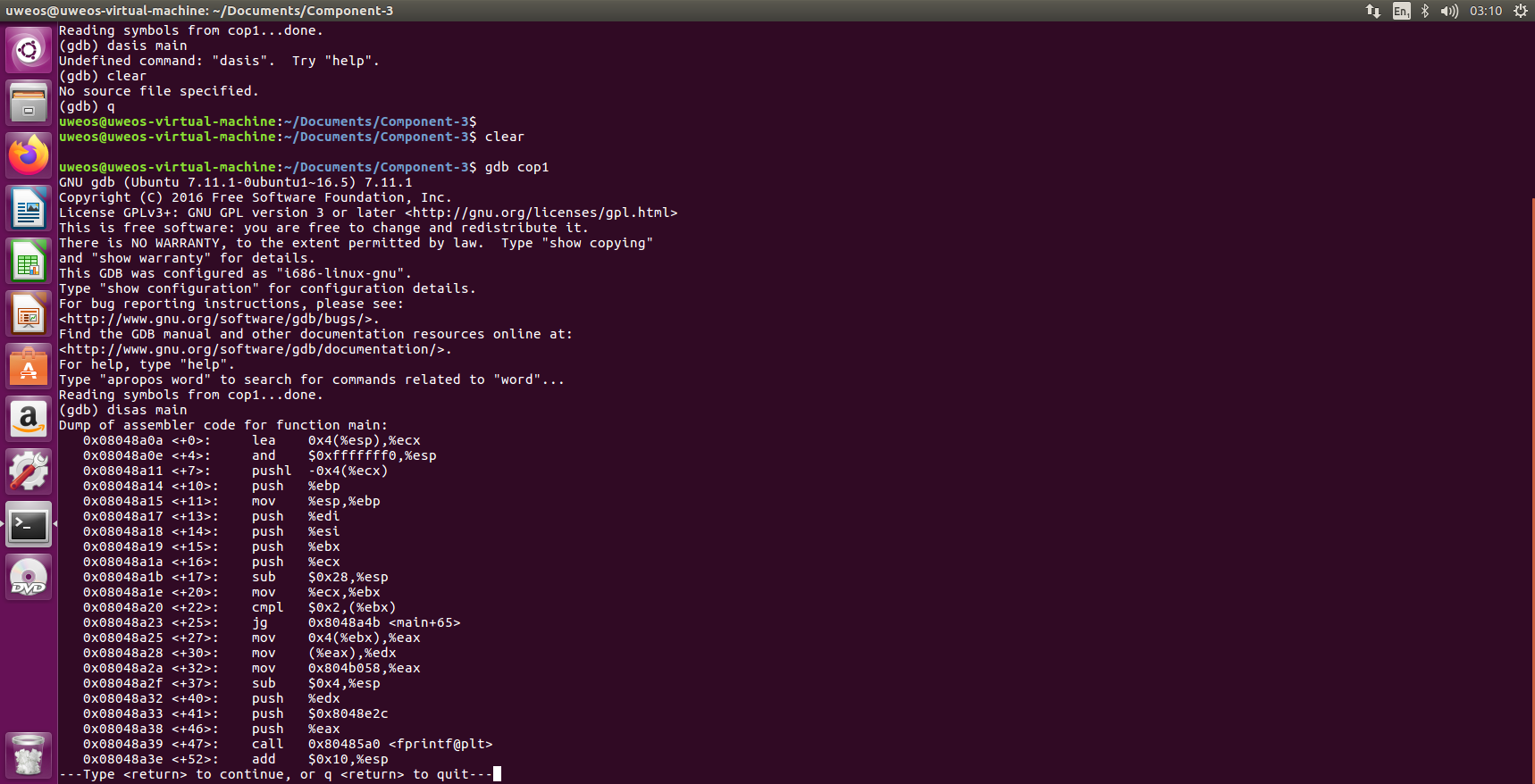


Figure 3: Buffer size allocation

The selected line in Figure 3 shows the buffer size allocation in memory which is initialized in the code in Appendix 2 at 256 buffer size that means that the buffer will move down towards the return for it overflow or anything else can be done.

A screen shot of a computer

Description automatically generated

Figure 4: gdb Normal Run

This Figure 4 Demonstrates how a normal run would look like using gdb debugger as shown in the figure the program has exited normally and no errors were shown, and no additional detail was shown, since the program has run successfully within the buffer range.

A screenshot of a computer screen

Description automatically generated

Figure 5: What happens in the memory in buffer overflow.

Figure 5 illustrates how the data flows, if anything that crosses the buffer in the return it will return a segmentation error and break the code, and that is where the exploitation starts.

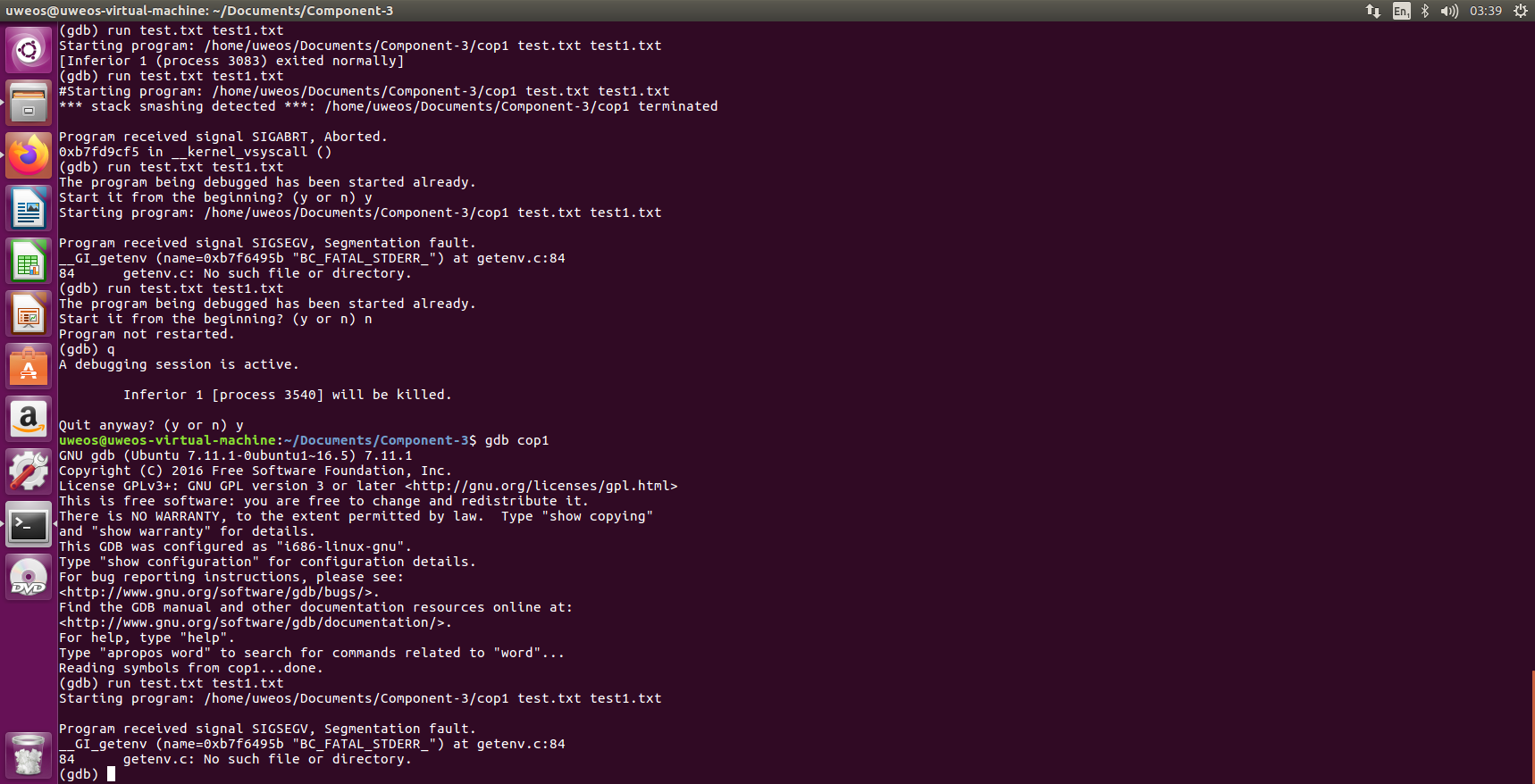


Figure 6: Segmentation Fault

Figure 6 shows the segmentation fault when a larger file is used, the segmentation fault is shown in the getenv() function, which is a system library function, used to retrieve values from the environment variable list, which could lead to a buffer overflow if there is a virus code in the getenv() function.

The way it can be done if we have a payload that could be introduced, which then it will return something the program should not and could lead to root access in the operating system, by running a shell code into this it could manipulate the program into, so it will run the file with Malicious code that will eventually run back into it.

A diagram of a bar graph

Description automatically generated

Figure 7: How Malicious code works

Figure 7 illustrates how the code would run back into the malicious code after the return function has been overflown and redirected into the malicious code inserted by the user which could lead to root access, or any other method of hack and exploitation.

# References

BABAYO, B. and HARUNA, A.S.E., 2023. AN IN-DEPTH ANALYSIS OF CURRENT OPERATING SYSTEMS. Journal of Systematic and Modern Science Research.

linux.die. n.d. fstat64(2) - Linux manual page. Available from: https://linux.die.net/man/2/fstat64 [Accessed 10 January 2024]

man7.org. n.d. execve(2) - Linux manual page. Available from: https://man7.org/linux/man-pages/man2/execve.2.html [Accessed 10 January 2024].

man7.org. n.d. brk(2) - Linux manual page. Available from: https://man7.org/linux/man-pages/man2/brk.2.html [Accessed 10 January 2024].

man7.org. n.d. mmap2(2) - Linux manual page. Available from: https://man7.org/linux/man-pages/man2/mmap2.2.html [Accessed 10 January 2024].

man7.org. n.d. open(2) - Linux manual page. Available from: https://man7.org/linux/man-pages/man2/open.2.html [Accessed 10 January 2024].

man7.org. n.d. close(2) - Linux manual page. Available from: https://man7.org/linux/man-pages/man2/close.2.html [Accessed 10 January 2024].

man7.org. n.d. read(2) - Linux manual page. Available from: https://man7.org/linux/man-pages/man2/read.2.html [Accessed 10 January 2024].

man7.org. n.d. write(2) - Linux manual page. Available from: https://man7.org/linux/man-pages/man2/write.2.html [Accessed 10 January 2024].

man7.org. n.d. mmap(2) - Linux manual page. Available from: https://man7.org/linux/man-pages/man2/mmap.2.html [Accessed 10 January 2024].

man7.org. n.d. exit\_group(2) - Linux manual page. Available from: https://man7.org/linux/man-pages/man2/exit\_group.2.html [Accessed 10 January 2024].

GeeksforGeeks . (2024). 'Monolithic Kernel and Microkernel: Key Differences - Overview', GeeksforGeeks Summary, [online]. Available at: https://www.geeksforgeeks.org/monolithic-kernel-and-key-differences-from-microkernel/ [Accessed: 10/01/2024].

# Appendix

## Appendix 1

#include <stdio.h>  
#include <stdlib.h>  
#include <fcntl.h>  
#include <unistd.h>  
#include <sys/stat.h>  
#include <string.h>  
#include <stdbool.h>  
#include <time.h>  
  
#define DEFAULT\_BUFFER\_SIZE 500  
  
void copy\_file(const char \*src, const char \*dest, int buffer\_size, bool verbose, bool append, bool overwrite, bool copy\_if\_newer, size\_t limit\_size) {  
    int src\_fd, dest\_fd, n\_read;  
    char buffer[buffer\_size];  
    struct stat src\_stat, dest\_stat;  
    bool dest\_exists = false;  
  
    src\_fd = open(src, O\_RDONLY);  
    if(src\_fd < 0) exit(1);  
  
    if(stat(src, &src\_stat) < 0) exit(1);  
  
    if(stat(dest, &dest\_stat) == 0) {  
        dest\_exists = true;  
        if(overwrite == false) {  
            printf("Destination file exists. Overwrite? (y/n): ");  
            char ch;  
            scanf(" %c", &ch);  
            if(ch != 'y' && ch != 'Y') exit(0);  
        }  
        if(copy\_if\_newer && src\_stat.st\_mtime <= dest\_stat.st\_mtime) {  
            if(verbose) printf("Source is not newer than destination. Skipping copy.\n");  
            exit(0);  
        }  
    }  
  
    int flags = O\_WRONLY | O\_CREAT;  
    if(append) flags |= O\_APPEND; else if(!dest\_exists || overwrite) flags |= O\_TRUNC;  
  
    dest\_fd = open(dest, flags, 0666);  
    if(dest\_fd < 0) exit(1);  
  
    size\_t total\_copied = 0;  
    while((n\_read = read(src\_fd, buffer, buffer\_size)) > 0 && (limit\_size == 0 || total\_copied < limit\_size)) {  
        size\_t to\_write = (limit\_size != 0 && total\_copied + n\_read > limit\_size) ? limit\_size - total\_copied : n\_read;  
        write(dest\_fd, buffer, to\_write);  
        total\_copied += to\_write;  
        if(verbose) printf("Copied %lu bytes\n", total\_copied);  
    }  
  
    close(src\_fd);  
    close(dest\_fd);  
    if(verbose) printf("File copy completed.\n");  
}  
  
int main(int argc, char \*argv[]) {  
    if(argc < 3) {  
        fprintf(stderr, "Usage: %s source\_file destination\_file [options]\n", argv[0]);  
        exit(EXIT\_FAILURE);  
    }  
  
    int buffer\_size = DEFAULT\_BUFFER\_SIZE;  
    bool verbose = false;  
    bool append = false;  
    bool overwrite = true;  
    bool copy\_if\_newer = false;  
    size\_t limit\_size = 0;  
  
    for(int i = 3; i < argc; i++) {  
        if(strcmp(argv[i], "--verbose") == 0 || strcmp(argv[i], "-v") == 0) {  
            verbose = true;  
        } else if(strcmp(argv[i], "--append") == 0 || strcmp(argv[i], "-a") == 0) {  
            append = true;  
        } else if(strcmp(argv[i], "--no-overwrite") == 0 || strcmp(argv[i], "-n") == 0) {  
            overwrite = false;  
        } else if(strcmp(argv[i], "--if-newer") == 0 || strcmp(argv[i], "-i") == 0) {  
            copy\_if\_newer = true;  
        } else if(strncmp(argv[i], "--buffer=", 9) == 0) {  
            buffer\_size = atoi(argv[i] + 9);  
        } else if(strncmp(argv[i], "--limit=", 8) == 0) {  
            limit\_size = atol(argv[i] + 8);  
        } else {  
            fprintf(stderr, "Unknown option: %s\n", argv[i]);  
            exit(EXIT\_FAILURE);  
        }  
    }  
  
    copy\_file(argv[1], argv[2], buffer\_size, verbose, append, overwrite, copy\_if\_newer, limit\_size);  
  
    return EXIT\_SUCCESS;  
}

## Appendix 2

#include <stdio.h>  
#include <stdlib.h>  
#include <fcntl.h>  
#include <unistd.h>  
#include <sys/stat.h>  
#include <string.h>  
#include <stdbool.h>  
#include <time.h>  
  
#define DEFAULT\_BUFFER\_SIZE 1024  
#define FIXED\_BUFFER\_SIZE 256 // Intentionally small buffer size for demonstration  
  
void copy\_file(const char \*src, const char \*dest, int buffer\_size, bool verbose, bool append, bool overwrite, bool copy\_if\_newer, size\_t limit\_size) {  
    int src\_fd, dest\_fd, n\_read;  
    char buffer[FIXED\_BUFFER\_SIZE]; // Fixed-size buffer  
    struct stat src\_stat, dest\_stat;  
    bool dest\_exists = false;  
  
    src\_fd = open(src, O\_RDONLY);  
    if(src\_fd < 0) exit(1);  
  
    if(stat(src, &src\_stat) < 0) exit(1);  
  
    if(stat(dest, &dest\_stat) == 0) {  
        dest\_exists = true;  
        if(overwrite == false) {  
            printf("Destination file exists. Overwrite? (y/n): ");  
            char ch;  
            scanf(" %c", &ch);  
            if(ch != 'y' && ch != 'Y') exit(0);  
        }  
        if(copy\_if\_newer && src\_stat.st\_mtime <= dest\_stat.st\_mtime) {  
            if(verbose) printf("Source is not newer than destination. Skipping copy.\n");  
            exit(0);  
        }  
    }  
  
    int flags = O\_WRONLY | O\_CREAT;  
    if(append) flags |= O\_APPEND; else if(!dest\_exists || overwrite) flags |= O\_TRUNC;  
  
    dest\_fd = open(dest, flags, 0666);  
    if(dest\_fd < 0) exit(1);  
  
    size\_t total\_copied = 0;  
    while((n\_read = read(src\_fd, buffer, buffer\_size)) > 0 && (limit\_size == 0 || total\_copied < limit\_size)) {  
        size\_t to\_write = (limit\_size != 0 && total\_copied + n\_read > limit\_size) ? limit\_size - total\_copied : n\_read;  
        write(dest\_fd, buffer, to\_write);  
        total\_copied += to\_write;  
        if(verbose) printf("Copied %lu bytes\n", total\_copied);  
    }  
  
    close(src\_fd);  
    close(dest\_fd);  
    if(verbose) printf("File copy completed.\n");  
}  
  
int main(int argc, char \*argv[]) {  
    if(argc < 3) {  
        fprintf(stderr, "Usage: %s source\_file destination\_file [options]\n", argv[0]);  
        exit(EXIT\_FAILURE);  
    }  
  
    int buffer\_size = DEFAULT\_BUFFER\_SIZE;  
    bool verbose = false;  
    bool append = false;  
    bool overwrite = true;  
    bool copy\_if\_newer = false;  
    size\_t limit\_size = 0;  
  
    for(int i = 3; i < argc; i++) {  
        if(strcmp(argv[i], "--verbose") == 0 || strcmp(argv[i], "-v") == 0) {  
            verbose = true;  
        } else if(strcmp(argv[i], "--append") == 0 || strcmp(argv[i], "-a") == 0) {  
            append = true;  
        } else if(strcmp(argv[i], "--no-overwrite") == 0 || strcmp(argv[i], "-n") == 0) {  
            overwrite = false;  
        } else if(strcmp(argv[i], "--if-newer") == 0 || strcmp(argv[i], "-i") == 0) {  
            copy\_if\_newer = true;  
        } else if(strncmp(argv[i], "--buffer=", 9) == 0) {  
            buffer\_size = atoi(argv[i] + 9);  
        } else if(strncmp(argv[i], "--limit=", 8) == 0) {  
            limit\_size = atol(argv[i] + 8);  
        } else {  
            fprintf(stderr, "Unknown option: %s\n", argv[i]);  
            exit(EXIT\_FAILURE);  
        }  
    }  
  
    copy\_file(argv[1], argv[2], buffer\_size, verbose, append, overwrite, copy\_if\_newer, limit\_size);  
  
    return EXIT\_SUCCESS;  
}

# Gitlab link

<https://gitlab.uwe.ac.uk/a5-hasan/operating-systems>