**Module Title: Operating Systems**

**Module Code: UFCFWK-15-2 Resit**

(Eyad, 21003020)

(Bitna 21065719)

(Farhan Zakir 22027036 )

**PintOS report**

Code repository: https://gitlab.uwe.ac.uk/f2-zakir/pintos\_student

Table of Contents

[**1** **PintOS Architecture** 3](#_Toc140924145)

[**1.1** **Kernal and User Space** 3](#_Toc140924146)

[**1.2** **Thread and Process Management** 3](#_Toc140924147)

[**1.3** **Memory Management in PintOS** 4](#_Toc140924148)

[**1.4** **File Systems** 4](#_Toc140924149)

[**2** **System Calls** 4](#_Toc140924150)

[**2.1** **System Call Design and Development** 5](#_Toc140924151)

[**3** **System Exploitation Summary** 5](#_Toc140924152)

[**4** **Countermeasures** 6](#_Toc140924153)

[**5** **References** 7](#_Toc140924154)

# **PintOS Architecture**

Pintos is an operating system framework, developed at Stanford university for educational purpose for the students in a way for them to engage the concepts and structure of found in the real operating systems(Stanford, 2019). Here in the report will be discussed on an overall structure of PintOS.

## **Kernal and User Space**

PintOS follows a rigid kernel architecture, where the code is in a single address space. The kernel is significant as it is the backbone of the OS, giving a direct access to the hardware resources and critical operations. On the other hand, where the developers work and run on the user space, isolated from the kernel for security and protection hence preventing user-level to access hardware which could hinder the system stability.

## **Thread and Process Management**

The basic units of execution in the Pintos operating system are threads, which stand for compact sets of instructions that can execute simultaneously on a CPU. Because they are formed within a process and share the same memory space, tasks may be completed quickly and the system is more responsive. A priority- based scheduler ensures equitable CPU time distribution by scheduling threads according to their priorities. Pintos offers synchronisation methods including locks, semaphores, and condition variables to stop data inconsistency and race situations. These aid in maintaining data integrity and managing shared resources. Pre-emption is another feature of Pintos that enables higher priority threads to interrupt lower priority ones as needed. The scheduler and events like I/O or synchronisation operations control the transitions between the various thread states, which include running, ready, blocked, and exited. For managing threads, Pintos provides a collection of kernel-level and system APIs that cover synchronisation, context switching, creation, and termination of threads. Additionally, it enables user- level threads, which offer freedom in design and execution control because user-space libraries manage the threads at this level(Stanford, 2019).

PintOS allows process creation, termination or scheduling, allowing users with managing the processes. Processes in PintOS are collection of threads that have the same executable code but separate memory spaces. The PCB (Process control block) stores information like stack, register and status.

## **Memory Management in PintOS**

Memory management in PintOS focuses on efficient allocation and deallocation of memory resources. Demand paging brings pages from disk into memory when they're needed, reducing memory overhead. Digital addresses are translated into physical addresses by a web page table maintained by each process. To cope with the digital-to-physical address mapping, PintOS uses a two-level web page table hierarchy along with a web page directory and page tables. A hierarchical approach reduces memory consumption memory. When a page fault occurs, PintOS fetches missing pages from disk and updates the web page table accordingly. There is a stack for each thread where a local variable can be stored and function returns can be addressed in PintOS.

## **File Systems**

Functional file systems are included in PintOS enabling operations by creating, opening, reading and closing files. It uses inodes to represent files and directories, also includes buffer cache which improves performance(Wikipedia, 2023).

# **System Calls**

User-level programmes can interface with system calls to ask for OS services. System calls in PintOS enable user programmes to communicate with the kernel. Process management (e.g., sys\_exec and sys\_exit), file operations (e.g., sys\_create and sys\_remove), and memory management (e.g., sys\_malloc and sys\_free) are all examples of common system calls in PintOS (The University of Texas at Austin, 2023).

## **System Call Design and Development**

The group decided to initialize 3 system calls during the meeting. SYS\_REMOVE, SYS\_CREATE, SYS\_EXEC. how it is implemented is that syscall\_handler function is initialized, where it retrieves system call number from the stack once it retrieves it, we move to switch statement where all the calls lie there.

* The exec function handle sys\_exec where it takes command line argument as in input and passed to process\_execute function.
* The create() function does handle sys\_create where it takes argument and name of the file and uses filesys\_create to new file
* The sys\_remove system call is used to delete a file from the file system adn filesys\_remove() under the syscall\_handler

The thing that is common is that they all use x86 architecture and after every string and the arguments passed onto stack, and the interrupt is responsible for syscall\_handler to call interrupt after the operation is done.

# **System Exploitation Summary**

* (Farhan Zakir 22027036 )(Using sys\_exec): utilized the shell hexacode for sys\_exec. Upon disassembling the code, I observed that strings and registers were pushed onto the stack, and I completed the process with interrupts. The result was the formation of a new user-level thread with a new command line named su.
* (Bit 21065719) (Using sys\_create): chose the shell hexacode for sys\_create. After disassembling the code, I discovered that I needed to push arguments onto the stack to create a new file. Once I did that (in the exploitation), I allowed the process to proceed until an interrupt halted it, successfully creating the new file.
* (Eyad, 21003020) (Using sys\_remove): For my part, I decided to work with the shell hexacode for sys\_remove. Upon disassembling the code, I noticed that I had to push the arguments, representing a new file, onto the stack before proceeding with the deletion process. I allowed the code to execute until an interrupt stopped it, resulting in the successful deletion of the specified file.

In this way, each team member used a different shell hexacode to achieve their individual goals - one created a new user-level thread with a new command line, another created a new file, and the third one successfully deleted a file.

# **Countermeasures**

텍스트, 폰트, 스크린샷, 화이트이(가) 표시된 사진

자동 생성된 설명

* For sys\_exec, the syscall file should be limited to authorized users or processes.

Make sure that executable files can only be located on trusted paths to prevent unauthorised programme execution.

텍스트, 폰트, 스크린샷이(가) 표시된 사진

자동 생성된 설명

* For sys\_create, we need to ensure that we give authroize file access to set control on who to read, write or execute the file.
* 텍스트, 폰트, 스크린샷, 대수학이(가) 표시된 사진

  자동 생성된 설명
* For sys\_remove, Only allow authorised users or processes to use sys\_remove. It should not be possible for unauthorised users to remove files at will.Make that the file being deleted is real and that the user has the right permissions to do so.

# **References**

Wikipedia (2023) Pintos available from: <https://en.wikipedia.org/wiki/Pintos> [accessed from 11th of July]

Stanford (2020) Pintos projects available from: <https://www.scs.stanford.edu/20wi-cs140/pintos/pintos.html> [accessed 12th of July]

The University of Texas at Austin (2023) Project documentation available from: <https://www.cs.utexas.edu/~ans/classes/cs439/projects/pintos/WWW/pintos_9.html> [accessed July 12th]