# OS-CA2

Kasra Noorbakhsh - Kourosh Sajjadi - Mehdi Jamalkhah

810100230 - 810100587 - 810100111

## Contents

# 1 Introduction

## 1.1 Libraries(Q1)

Because we use system calls frequently, we use a library that is a abstraction of lower levels of operating system. This library helps us to manage and change system calls more easily and this helps us to add system calls more efficiently.

## 1.2 Ways Of Access(Q2)

- **Interrupts**: Interrupts are hardware or software signals that can cause the CPU to interrupt its current execution and switch to a specific routine to handle the interrupt.
- **Exceptions**: Exceptions are similar to interrupts but are typically caused by the CPU in response to specific conditions such as divide-by-zero errors, page faults, or illegal instructions.
- **I/O Instructions**: Certain input/output instructions can only be executed in kernel mode. When a user-space program attempts to execute these instructions, a trap occurs, and control switches to the kernel, allowing the kernel to handle the I/O operation on behalf of the user-space program.
- **Software Interrupt**: They cause a context switch to kernel mode. These instructions are usually used for system-specific operations and are triggered by the int instruction in x86 assembly language.
- **Direct Memory Access (DMA)**: While DMA itself doesn't change the CPU's privilege level, it allows devices to access memory independently, which can be used to communicate with kernel-space buffers.

# 2 System Call Implementation

## 2.1 DPL-USER(Q3)

- **Critical System Operations**: Some traps correspond to critical system operations that must be executed with higher privileges to maintain the integrity and security of the operating system. For example, traps related to memory management (such as page faults) and hardware interrupts (such as timer interrupts)
- **Protection of Sensitive Resources**: Certain traps, such as those related to I/O operations or accessing sensitive hardware devices, require elevated privileges. Allowing user-level programs to directly access these resources without proper checks and controls would compromise system security and stability.

## 2.2 Stack(Q4)

This mechanism is part of the hardware-supported privilege level protection.

The Stack Segment (SS) and Stack Pointer (ESP) values are pushed onto the stack during a privilege level change for several reasons:

- **Security and Isolation**: Changing privilege levels is a critical operation that involves transitioning from less privileged code (user mode) to more privileged code (kernel mode) or vice versa. It's important to maintain security and prevent unauthorized access to kernel data structures. By pushing SS and ESP onto the stack during a privilege level change, the processor ensures that the kernel can safely execute without the risk of user mode processes tampering with kernel stack data.
- **Context Switching**: When a process switches from user mode to kernel mode (for system calls or exceptions), the kernel needs to save the user mode stack state before switching to the kernel stack. Similarly, when transitioning from kernel mode back to user mode, the kernel needs to restore the user mode stack state. By pushing SS and ESP during a privilege level change, the kernel can efficiently switch between user mode and kernel mode stack contexts.

# 3 High Level

## 3.1 Argptr(Q5)

- **argint**: Its a function that extracts an integer argument from the nth position in the user's argument list and stores it in the memory location pointed to by IP.
- **argptr**: Its a function that extracts a pointer argument from the nth position in the user's argument list and stores it in the memory location pointed to by pp.

the check for the validity of the address range is important to ensure that the user program does not try to access memory outside its allocated address space. This validation is crucial for security and stability reasons like buffer overflow:

A buffer overflow occurs when a program writes more data to a block of memory, or buffer, than it can hold. If a malicious user can control the data written beyond the buffer's boundaries, they can overwrite adjacent memory, including crucial data structures or function pointers. This can lead to unpredictable behavior, crashes, or, more dangerously, arbitrary code execution.

let's say a user program calls sys-read() to read data from a file into a buffer. The program assumes that the user input will fit into a buffer of a fixed size without performing proper bounds checking.

This can result in unpredictable behavior, crashes, or, in worst-case scenarios, the injected data could contain executable code, allowing the attacker to gain control over the program's execution.

# 4 GDB

After the boot of the OS we placed a break-point at line 130 in the syscall.c file Our program started and the gdb stopped at the break-point

At-last by calling the bt(back trace) we arrived at the outputs below

```c
#include "types.h"
#include "user.h"

int main(int argc, char* argv[]) {

    int pid = getpid();
    printf(1, "Our Process Id is %d\n", pid);
    exit();

}
```

Figure 1: user program

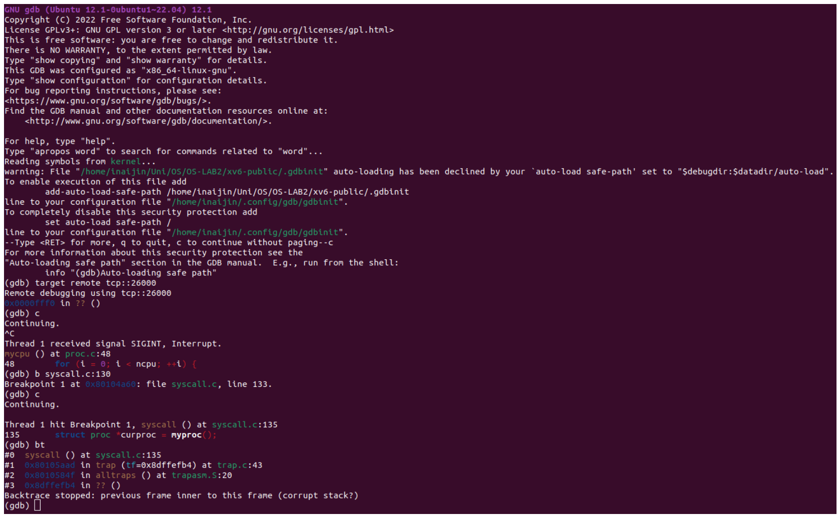


Figure 2: placing break-point

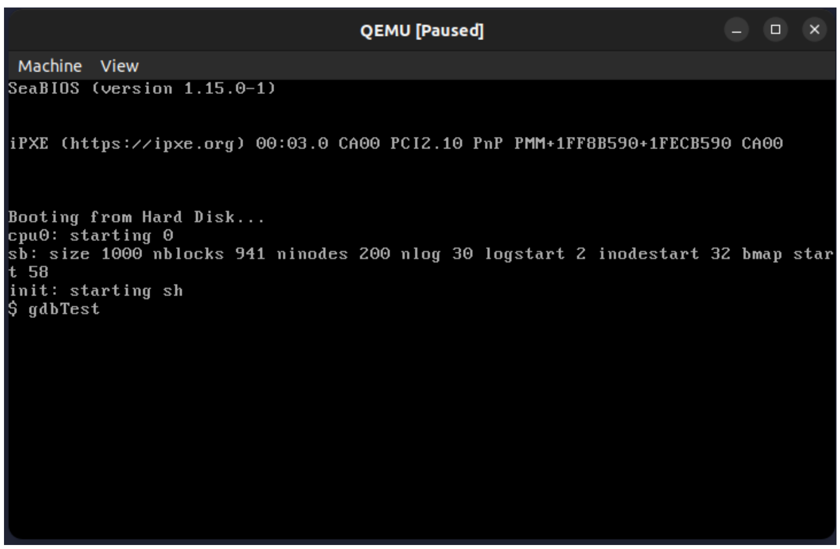


Figure 3: xv6 in gdb

- **bt**: Every thing happened after the break-point is pushed to stack. This command will print one line per frame for frames in the stack. By default, all stack frames are printed.
- **syscall.h**: A number is chosen for our system call.
- **user.h**: The id of our system call is written.
- **usys.s**: The system call definition in assembly.
- **all-traps**: First the trap frame gets built then it gets pushed in the stack then the trap in trap.c gets called.
- **syscall**: After it gets called in the field eax in trap frame current process it calls the function assigned to it.

With the use of the up we can go one frame backward.

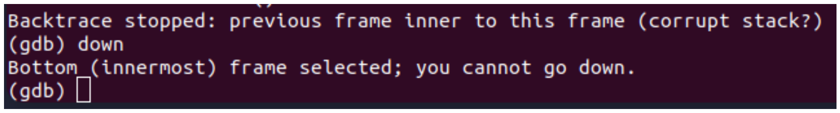


Figure 4: down

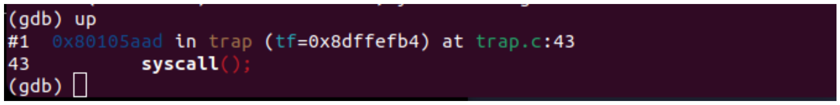


Figure 5: up

We know that getpid() system call's pid is 11 but the eax register shows 5 that isn't the pid of getpid()


Figure 6: eax wrong value

The reason for this is before getting to getpid system call we must call other system calls (like read...) by calling the c(continue) call in gdb this will get fixed.

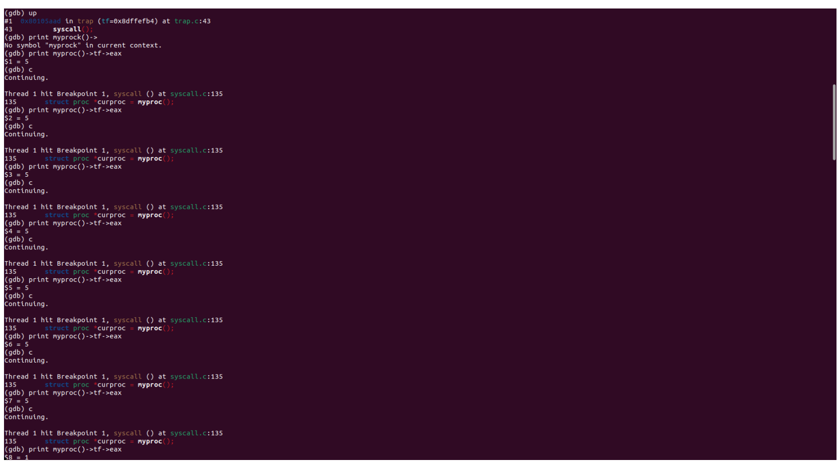


Figure 7: C + myproc().tf.eax

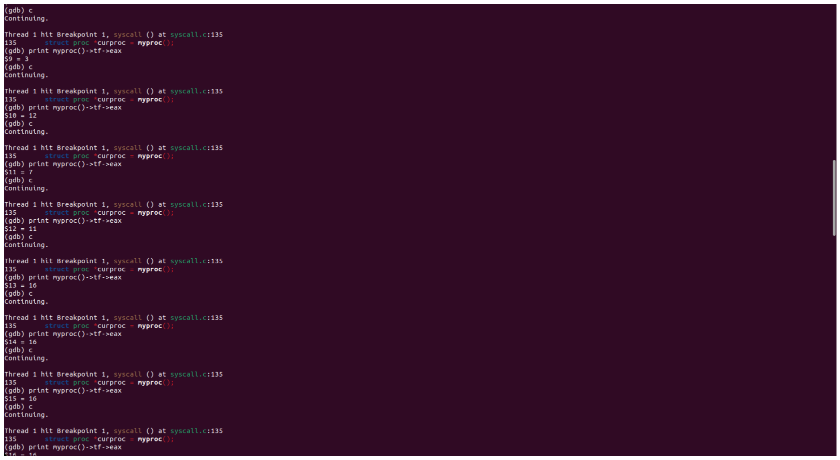


Figure 8: C + myproc().tf.eax

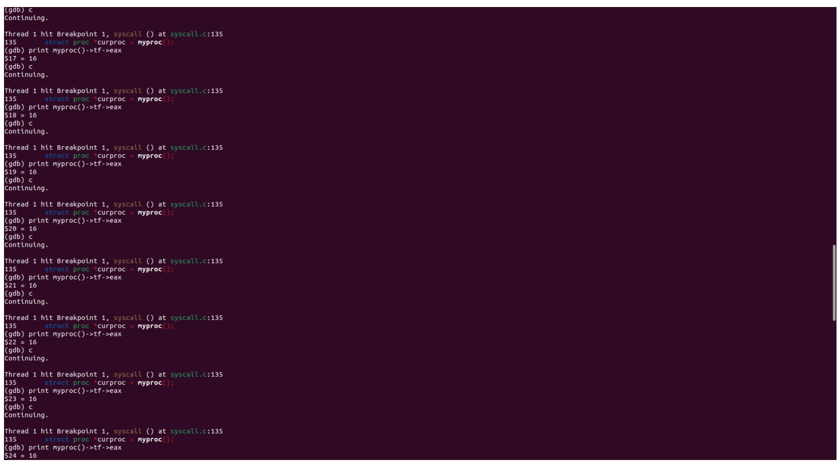


Figure 9: C + myproc().tf.eax

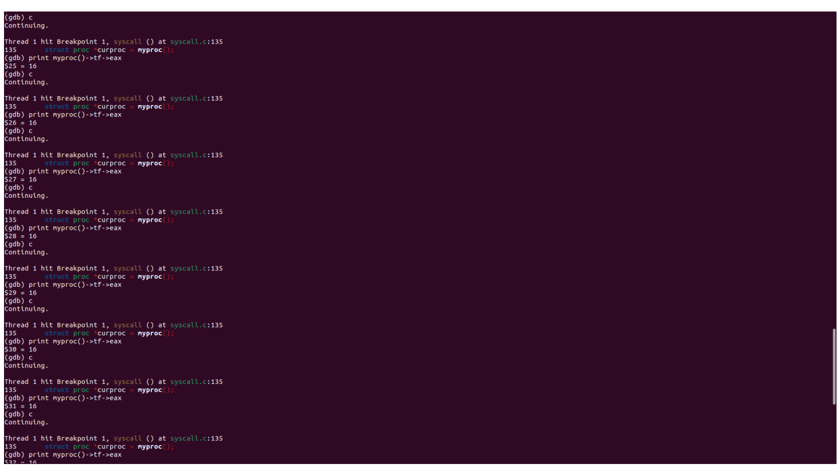


Figure 10: C + myproc().tf.eax


Figure 11: C + myproc().tf.eax

- **System-Call 5**: read()

  This system call gets called multiple times so that it reads the typed command in terminal.

- **System-Call 1**: fork()

  This system call gets called for making new processes.

- **System-Call 12**: sbrk()

  This system call gets called for allocating memory to processes.

- **System-Call 7**: exec()

  This system call gets called for running the pid program in the created processes.

- **System-Call 3**: wait()

  This system call is called in the father process and waits for the child to be executed.

- **System-Call 11**: getpid()

  This our own user program that is called in the terminal.

# 5 Commands

For adding new system calls to our OS, we must do certain things done. In defs.h and user.h and syscall.h header files we added our system calls prototype. In syscall.c sysproc.c proc.c and usys.s we added our system calls prototype again and in syscall.h header file we defined a new number for each system call. In some user programs we wrote the code for each of them to test them, just like as the project description.

```
SYSCALL(fork)
SYSCALL(exit)
SYSCALL(wait)
SYSCALL(pipe)
SYSCALL(read)
SYSCALL(write)
SYSCALL(close)
SYSCALL(kill)
SYSCALL(exec)
SYSCALL(open)
SYSCALL(mknod)
SYSCALL(unlink)
SYSCALL(fstat)
SYSCALL(link)
SYSCALL(mkdir)
SYSCALL(chdir)
SYSCALL(dup)
SYSCALL(getpid)
SYSCALL(sbrk)
SYSCALL(sleep)
SYSCALL(uptime)
SYSCALL(get_uncle_count)
SYSCALL(get_process_lifetime)
SYSCALL(copy_file)
SYSCALL(find_digital_root)
```

Figure 12: usys.s modifications

```
int get_uncle_count(int);
int get_process_lifetime(int);
int copy_file(const char*, const char*);
int find_digital_root(void);
```

Figure 13: user.h modifications

```
#define SYS_get_uncle_count 22
#define SYS_get_process_lifetime 23
#define SYS_copy_file 24
#define SYS_find_digital_root 25
```

Figure 14: syscall.h modifications

```c
int
get_uncle_count(int pid)
{
  struct proc *p;
  struct proc *parent = myproc();
  // Loop over process table looking for process to run.
    acquire(&ptable.lock);
    for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
      if(p->pid == pid) {
        parent = p->parent;
      }
    }
    release(&ptable.lock);
    return parent->parent->num_childs - 1;
}

int
get_process_lifetime(int pid)
{
  struct proc *p;

  acquire(&ptable.lock);
  for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
    if(p->pid == pid) {
      break;
    }
  }
  release(&ptable.lock);
  
  return ticks - p->birthday;
}
```

Figure 15: proc.c modifications

```c
int
sys_get_uncle_count(void)
{
  int pid;
  if(argint(0, &pid) < 0) {
    return -1;
  }
  return get_uncle_count(pid);
}

int
sys_get_process_lifetime(void)
{
  int pid;
  if(argint(0, &pid) < 0) {
    return -1;
  }
  return get_process_lifetime(pid);
}

int
sys_copy_file(void)
{
  char *src;
  char * dest;
  if(argstr(0, &src) < 0 || argstr(1, &dest) < 0) {
    return -1;
  }
  return copy_file(src, dest);

}

int
sys_find_digital_root(void)
{
  return find_digital_root(myproc()->tf->ebx);
}
```



Figure 16: sysproc.c modifications

```c
extern int sys_get_uncle_count(void);
extern int sys_get_process_lifetime(void);
extern int sys_copy_file(void);
extern int sys_find_digital_root(void);
```

Figure 17: sysproc.c modifications

```c
int             get_uncle_count(int);
int             get_process_lifetime(int);
int             copy_file(const char*, const char*);
int             find_digital_root(int);
```

Figure 18: defs.h modifications

## 5.1 Get-Uncle-Count

We add a number of children to our proc structure and initialize it in the allocproc function to 0 and we change our fork such that it changes the parent number of children and the created process number of children for the uncle we just return the number of grand parents children minus itself.

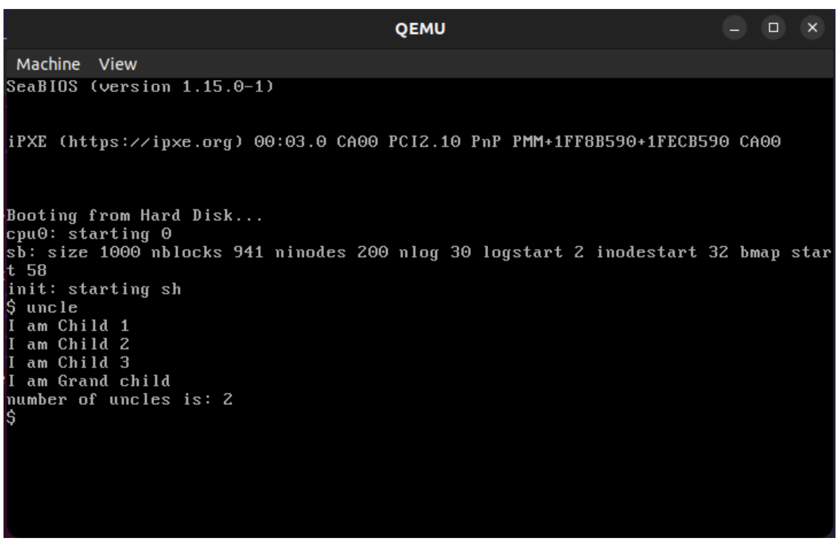


Figure 19: Get-Uncle-Count

## 5.2 Get-Lifetime

We first add a birthday variable to our proc structure then we initial it in allocproc function to our tick then for the lifetime we just subtract the birthday with the current tick.

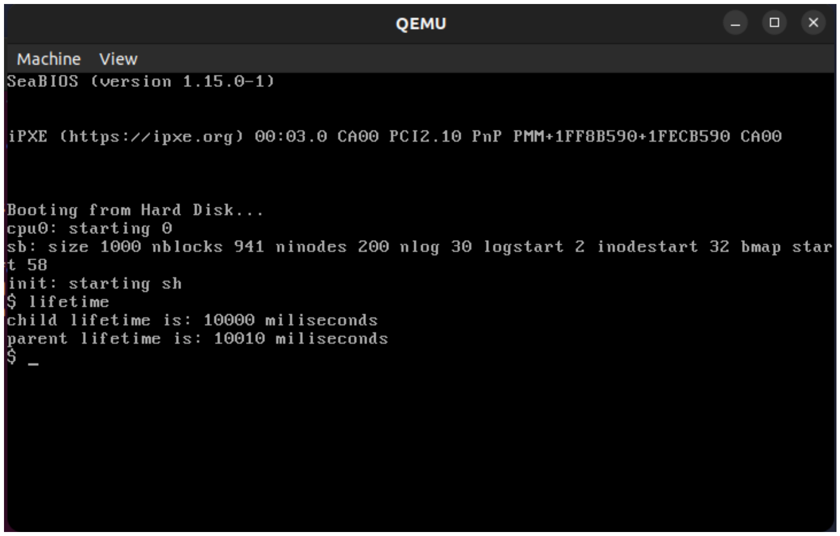


Figure 20: Get-Lifetime

## 5.3 Copy-File

We simply took the already available create and copy file codes and use them we also for the case that the names are similar we chose to cause an error.

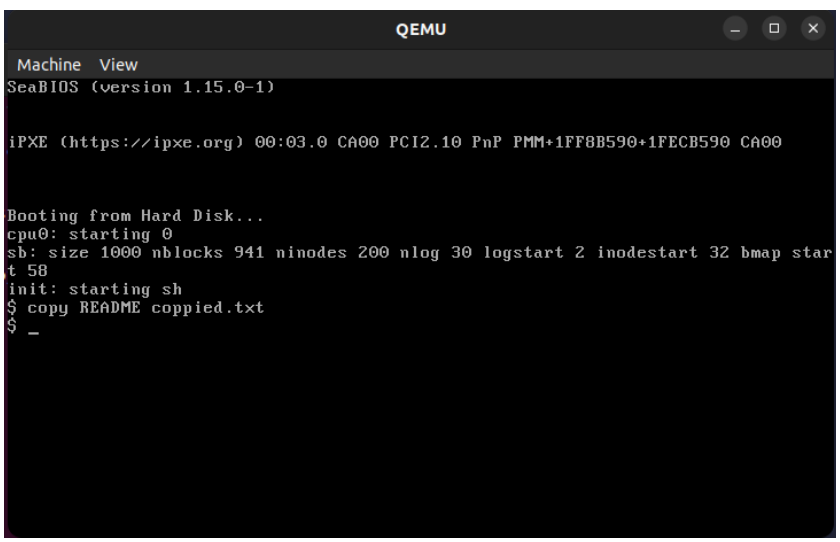


Figure 21: copy READ-ME

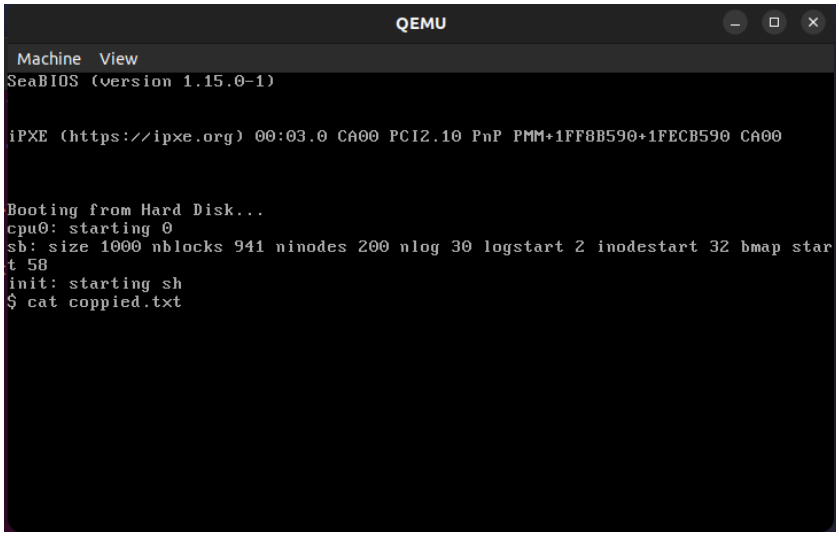


Figure 22: cat copied-file

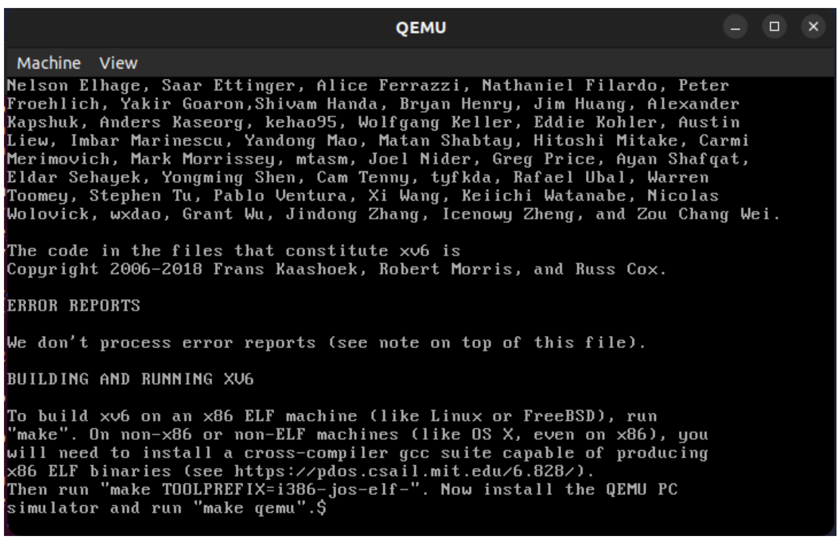


Figure 23: result

### 5.4 Find-Digital-Root

For this we first using assembly language store the ebx reg (can't use eax cause that saves the id) and we use that for our calculation then we restore it using assembly language.

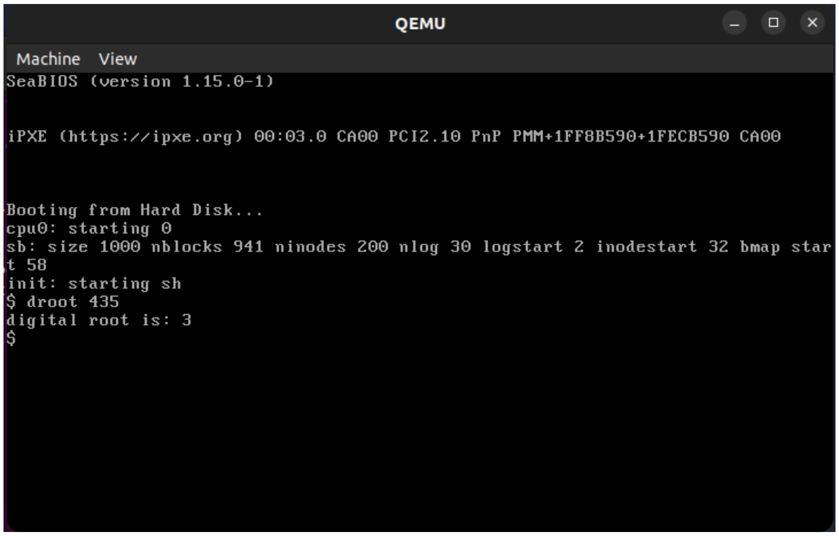


Figure 24: Find-Digital-Root


Figure 25: Find-Digital-Root