

Tutorial 2: Operating Systems

Yonghao Lee

April 3, 2025

Definitions

Processes & I/O

A program is an executable file, whereas a process is an executing instance of a program to which system resources (CPU time, memory, etc.) are allocated. One of the main tasks of an operating system is scheduling these processes efficiently.

Input/Output (I/O) refers to operations that a process needs to perform but cannot execute independently. These operations require interaction with external devices or systems. The process requires an intermediary (typically the operating system) to facilitate communication with peripheral devices, allowing data transfer to and from these devices.

The Kernel

The kernel is the core component of the operating system and has complete control over everything that happens in the system. It manages system resources, provides essential services, and interfaces between hardware and software components.

The kernel is considered trusted software, while all other programs are treated as untrusted. This distinction is critical for security, as certain privileged instructions are too dangerous to be executed by untrusted programs.

Kernel Mode & User Mode

Modern CPUs implement different privilege levels, primarily:

- **Kernel Mode:** In this mode, the CPU can execute any instruction and access any memory location. The entire kernel, which is a controller of process executes only in kernel mode.
- **User Mode:** This is a non-privileged mode with restricted capabilities. Programs running in user mode:
 - Cannot execute certain privileged instructions (e.g., halt)
 - Can only access memory specifically allocated to them

- Cannot directly access hardware devices
- Must request services from the kernel through system calls when privileged operations are needed

This separation helps maintain system stability and security by preventing user programs from interfering with critical system functions or other processes.

Examples of Privileged Instructions (x86-64)

The following are examples of instructions that can only be executed in kernel mode:

- **HLT** (Halt): Instructs the CPU to halt all execution until an external interrupt is received. If arbitrary programs could halt the processor, they could effectively freeze the entire system.
- **Read/Write on Control Registers**: Control registers contain critical system configuration flags that affect fundamental CPU behavior. For example:
 - The control register contains the cache enable/disable bit, for example.
 - Allowing unprivileged access could permit malicious programs to disable memory protection or caching, severely degrading performance or compromising security
- **LIDT** (Load Interrupt Descriptor Table Register): Sets the IDTR register, which points to the Interrupt Descriptor Table (IDT). The IDT defines how interrupts are handled by mapping interrupt vectors to their handler routines. If user programs could modify this table, they could redirect system interrupts to malicious code, effectively hijacking the system.

In Practice: Protection Rings

In most x86 architectures, the CPU hardware supports a hierarchical protection scheme with 4 privilege levels, known as protection rings:

- **Ring 0** (Most Privileged): Where the kernel and core OS components operate. This ring has unrestricted access to all CPU instructions and hardware resources.
- **Ring 1 & Ring 2** (Intermediate Rings): Historically intended for device drivers and system services. In modern operating systems, these rings are rarely used, with most drivers now running either in Ring 0 (for performance) or Ring 3 (for security).
- **Ring 3** (Least Privileged): Where user applications run. Programs in this ring cannot execute privileged instructions and must use system calls to request kernel services.

This ring-based protection model creates concentric circles of privilege, with inner rings having access to more sensitive operations than outer rings. The CPU enforces these boundaries, preventing outer rings from accessing inner ring resources without proper authorization.

Current Privilege Level (CPL)

The Current Privilege Level (CPL) is a value maintained by the CPU that indicates which protection ring the processor is currently executing code in. The CPU constantly monitors this value to enforce access restrictions.

In x86 architecture:

- The CPL is stored in the two least significant bits (bits 0-1) of the CS (Code Segment) register. Recall that the current code segment in memory is where the CPU fetches program instructions from, this is where the CS register points to.
- These bits can represent values from 0 to 3, corresponding to the four protection rings.
- When a program attempts to execute a privileged instruction, the CPU compares the CPL against the required privilege level for that instruction.

OS-Process Interaction

The primary goal of an operating system is to manage processes by allocating resources they need to execute. This relationship between the OS and processes is facilitated through a structured interface.

When a user mode process (running with restricted privileges) needs to access a service provided by the kernel (such as file I/O, network operations, or hardware access), the system must temporarily switch to kernel mode. This controlled transition between privilege levels is fundamental to maintaining system security and stability.

The OS provides its services to user processes via **system calls**, which form the well-defined interface between user programs and the kernel.

System Calls

A system call is a programmatic way for a process to request a service from the kernel. Examples include:

- File operations (open, read, write, close)
- Process control (fork, exec, exit)
- And more

System calls provide the only controlled entry points into the kernel, allowing user programs to request privileged operations without compromising system security.

System Call Mechanism

Each system call has a unique identifier and follows a specific sequence to safely transition between user and kernel mode:

1. **Preparation:** The process prepares for the system call by:

- Loading the system call number into a designated register (RAX in x86-64)
 - Setting up arguments in specific registers (RDI for first argument, RSI for second, etc.)
2. **State Preservation:** The current execution state (register values, flags) is saved to allow for return to user mode later
 3. **Mode Transition:** The processor executes a special instruction (SYSCALL in x86-64) that:
 - Atomically switches from user mode (Ring 3) to kernel mode (Ring 0)
 - Changes the instruction pointer (equivalent to PC) to the kernel's system call handler
 4. **Kernel Execution:** The kernel verifies the system call number and arguments, then executes the requested service
 5. **Return to User Mode:** Once the kernel completes the requested operation:
 - It places any return value in the appropriate register
 - It restores the saved user-space context
 - It executes a return instruction (SYSRET in x86-64) to switch back to user mode with reduced privileges

This controlled mechanism ensures that user processes can access necessary system services while maintaining the security boundary between user and kernel space.

Note that Libc's functions are not system calls.

In high-level languages, e.g., C, we do not call system calls directly.

The C standard library wraps some of the systemcalls.

For example, the write function is part of the C standard library.

Trap's Overhead

Making a trap is extremely slow, switching from kernel mode to user mode is easy, yet the opposite is hard and consumes a lot of time.

The good news is that not all the kernel functions really need kernel mode. The OS decides what runs in which mode, think, for example, that we want to know the number of the processor on which the process runs on currently, we might not need the kernel mode.

The good news is that not all the kernel functions really need kernel mode. The OS decides what runs in which mode, think, for example, that we want to know the number of the processor on which the process runs on currently, we might not need the kernel mode.

Example - System Call `open`

When a program calls a high-level C function like `fopen()`, a series of operations occur that bridge user space and kernel space:

1. The program calls `fopen(...)` with a filename and mode
2. C standard library functions process this call by:
 - Validating parameters
 - Setting up data structures
 - Preparing arguments for the system call
3. The library sets the RAX register to the number of the 'open' system call
4. It calls a trap (interrupt 0x80, or `syscall` instruction)
5. The kernel's trap handler checks which system call was requested by examining RAX
6. The kernel locates the memory address of the `sys_open` function
7. The kernel executes `sys_open`, calling various kernel functions
8. When finished, the kernel saves the return value in the RAX register
9. Control returns to user space where the C library:
 - Processes the return value from RAX (a file descriptor or error code)
 - Creates a `FILE*` object if successful
 - Returns this object to the calling program

This process illustrates the layered architecture of system calls, where the C standard library provides a high-level, portable interface to low-level kernel functions. The RAX register serves as the critical interface point for passing the system call number to the kernel and receiving the result.

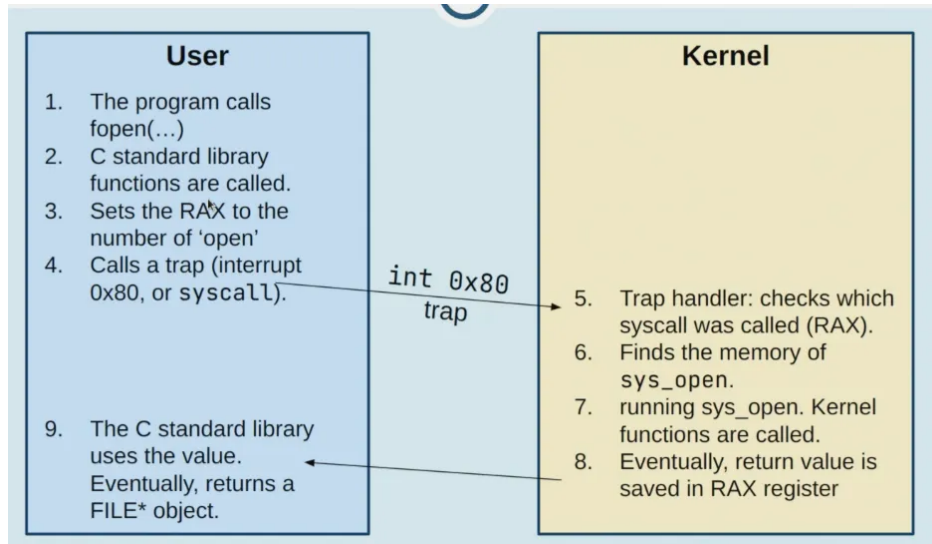


Figure 1: `fopen` example

Understanding Return Values and Error Handling

When using system calls directly or indirectly through library functions, proper error handling is essential:

- System calls like `open()` return an integer file descriptor on success
- A negative return value indicates failure
- The specific error is stored in the global variable `errno`
- The `perror()` function can be used to display human-readable error messages based on `errno`

```
int fd = open("/tmp/foo", ...);
if( fd < 0 ) {
    perror("Error opening file");
    /* Handle error */
}
```

Common error codes include:

- `ENOENT` (2): No such file or directory
- `EINTR` (4): Interrupted system call
- `EIO` (5): I/O error
- `EACCES` (13): Permission denied
- `EBUSY` (16): Device or resource busy

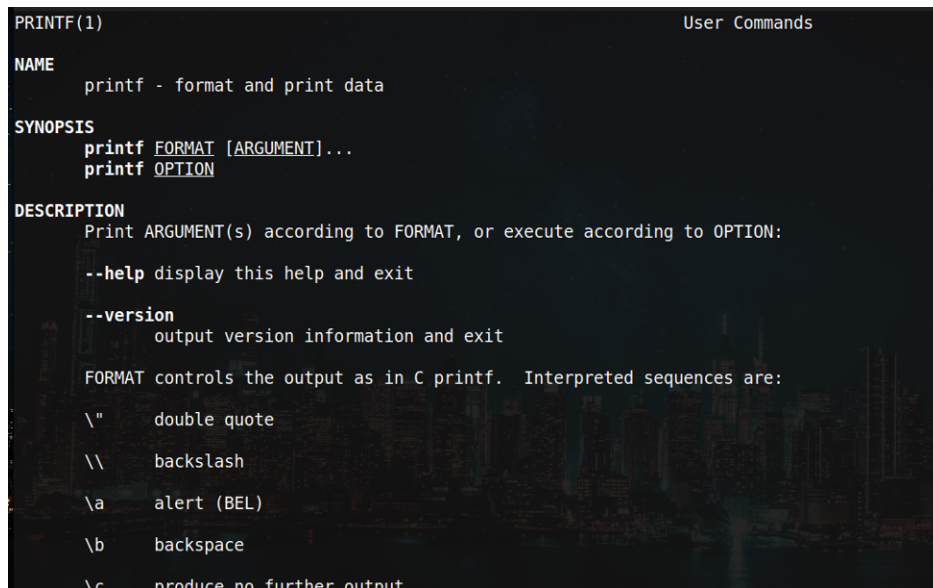
Using Man Pages for System Call Documentation

The `man` command provides detailed documentation for system calls and library functions:

- `man 1` - Shows documentation for executable programs and shell commands
- `man 2` - Shows documentation for system calls (kernel functions)
- `man 3` - Shows documentation for C library functions

For example:

- `man 2 open` will display details about the `open` system call
- `man 3 fopen` will display details about the `fopen` library function



```
PRINTF(1) User Commands
NAME
    printf - format and print data
SYNOPSIS
    printf FORMAT [ARGUMENT]...
    printf OPTION
DESCRIPTION
    Print ARGUMENT(s) according to FORMAT, or execute according to OPTION:
    --help display this help and exit
    --version output version information and exit
    FORMAT controls the output as in C printf. Interpreted sequences are:
    \" double quote
    \\ backslash
    \a alert (BEL)
    \b backspace
    \c produce no further output
```

Figure 2: `man 1 mkdir`

```
MKDIR(2) Linux Programmer's Manual

NAME
    mkdir, mkdirat - create a directory

SYNOPSIS
    #include <sys/stat.h>
    #include <sys/types.h>

    int mkdir(const char *pathname, mode_t mode);

    #include <fcntl.h>          /* Definition of AT_* constants */
    #include <sys/stat.h>

    int mkdirat(int dirfd, const char *pathname, mode_t mode);

    Feature Test Macro Requirements for glibc (see feature_test_macros(7)):

    mkdirat():
        Since glibc 2.10:
            _POSIX_C_SOURCE >= 200809L
        Before glibc 2.10:
            _ATFILE_SOURCE

DESCRIPTION
    mkdir() attempts to create a directory named pathname.
```

Figure 3: man 2 mkdir

Notice that the first and the second man commands give results for lookups on shell command and system call respectively.

Man pages typically include:

- Function prototype and required header files
- Description of parameters and return values
- Possible error codes
- Related functions

Using strace

strace is a powerful debugging utility that monitors and displays system calls made by a process. It provides insight into the interaction between user programs and the kernel.

Usage:

```
strace ./your_program [arguments]
strace -p PID # attach to running process
```

This tool is useful for:

- Debugging applications when source code is unavailable
- Understanding how programs interact with the operating system
- Identifying which system calls might be failing
- Analyzing performance issues related to I/O or other system operations

Example output for a simple file open operation might show:


```

open("/tmp/foo", O_RDONLY)      = 3
read(3, "hello\n", 4096)       = 6
close(3)                        = 0

```

This matches our earlier example of how `fopen()` ultimately calls the `open` system call, followed by operations on the resulting file descriptor.

Understanding strace Output

The following images illustrate the use of `strace`, a debugging utility for monitoring system calls:

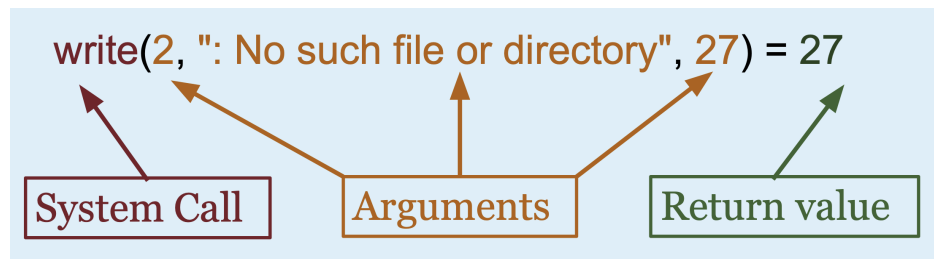


Figure 4: Basic strace output format

Figure 4 shows the basic output format of `strace`. Each line of output shows:

- The system call name (e.g., `write`)
- The arguments passed to the system call, shown inside parentheses
- The return value of the system call, shown after the equals sign

In the example, the `write` system call is invoked with three arguments:

- File descriptor 2 (standard error)
- The string `": No such file or directory"`
- The length 27 (number of bytes to write)

The return value of 27 indicates that all 27 bytes were successfully written.

Strace example: “strace ls /python/”

84

```
• open("/usr/share/locale/en_GB/LC_MESSAGES/coreutils.mo", O_RDONLY) =  
  -1 ENOENT (No such file or directory)  
• write(2, "ls: ", 4) = 4  
• write(2, "cannot access /python/", 22) = 22  
• open("/usr/share/locale/en_US/LC_MESSAGES/libc.mo", O_RDONLY) = -1  
  ENOENT (No such file or directory)  
• open("/usr/share/locale/en/LC_MESSAGES/libc.mo", O_RDONLY) = -1  
  ENOENT (No such file or directory)  
• open("/usr/share/locale/en_GB/LC_MESSAGES/libc.mo", O_RDONLY) = 3  
• fstat(3, {st_mode=S_IFREG|0644, st_size=1474, ...}) = 0  
• mmap(NULL, 1474, PROT_READ, MAP_PRIVATE, 3, 0) = 0x7f30b2df1000  
• close(3) = 0  
• write(2, ": No such file or directory", 27) = 27  
• write(2, "\n", 1) = 1  
• close(1) = 0
```

Figure 5: Output of running “strace ls /python/”

Figure 5 demonstrates running **strace** on the **ls** command when trying to list a non-existent directory. Key observations:

- Multiple unsuccessful **open** system calls trying to find localization files, all returning -1 with ENOENT error (No such file or directory)
- A successful **open** call (returning file descriptor 3)
- Memory mapping of the opened file with **mmap**
- The error message being written to standard error

This shows how even a simple command like **ls** makes multiple system calls to access files and handle error conditions.

Strace example: “strace wc sample2.in”

85

```
• stat("sample2.in", {st_mode=S_IFREG|0777, st_size=490, ...}) = 0
• open("sample2.in", O_RDONLY) = 3
• read(3, " The path of the righteous man "..., 16384) = 490
• open("/usr/lib/x86_64-linux-gnu/gconv/gconv-modules.cache", O_RDONLY) = 4
• fstat(4, {st_mode=S_IFREG|0644, st_size=26066, ...}) = 0
• mmap(NULL, 26066, PROT_READ, MAP_SHARED, 4, 0) = 0x7f81a4c88000
• close(4) = 0
• read(3, "", 16384) = 0
• fstat(1, {st_mode=S_IFCHR|0620, st_rdev=makedev(136, 4), ...}) = 0
• mmap(NULL, 4096, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0x7f81a4c87000
• write(1, " 1 96 490 sample2.in\n", 23) = 23
• close(3) = 0
• close(1) = 0
```

Figure 6: Output of running “strace wc sample2.in”

Figure 6 shows the sequence of system calls when running the `wc` (word count) command on a file. The sequence demonstrates a typical file processing pattern:

- `stat` to retrieve file metadata (size, permissions, etc.)
- `open` to obtain a file descriptor (3)
- `read` to actually read the file’s contents (490 bytes)
- Additional system calls for processing (including more `open`, `fstat`, and `mmap` calls)
- `write` to output the result to standard output (file descriptor 1)
- `close` to release the file descriptors

Interrupts

Motivation

Much of the functionality embedded inside a computer is implemented by hardware devices other than the processor. Since each device operates at its own pace, a method is needed for synchronizing the operation of the processor with these devices. Interrupts are the primary mechanism that allows hardware to communicate with software; they are how our keyboard, mouse, network interface card (NIC), and other peripherals signal the CPU that they need attention. Without interrupts, the processor would waste computational resources by constantly checking (polling) if devices need service, making interrupt-driven I/O significantly more efficient for system performance. Moreover, polling-based approaches risk losing data when the rate of data transfer from hardware devices is high. Since polling occurs at fixed

intervals, any data that arrives between polling cycles must be buffered by the hardware. If this buffer fills up before the next polling cycle, incoming data will be overwritten or discarded. This problem becomes more severe as data transfer rates increase, making polling inadequate for high-speed devices. Interrupts address this limitation by allowing devices to notify the CPU immediately when data is available, reducing the likelihood of buffer overflow and data loss.

On Interrupts

Instead of polling hardware devices to wait for their response, with interrupts, each device is responsible for notifying the processor about its current state. When a hardware device needs the processor's attention, it sends an electrical signal through a dedicated pin in the interrupt controller chip (located on the computer's motherboard). An interrupt is a signal to the CPU indicating that an event has occurred, and it results in changes in the sequence of instructions executed by the CPU.

Interrupts can also be characterized as asynchronous procedure calls (APCs). An asynchronous procedure call refers to the execution of code that occurs outside the normal sequential flow of a program and is triggered by an event rather than being explicitly called by the program. Unlike regular function calls where the program explicitly transfers control to a subroutine at a predetermined point, APCs happen in response to external events at unpredictable times. The interrupt is effectively invisible to the interrupted program since the program cannot control when the interrupt occurs. The operating system handles the interrupt, saves the current execution state, services the interrupt, and then restores the program's state, allowing it to continue execution as if nothing had happened.

Interrupts Categorization

Interrupts can be categorized in several ways: For example, by their source:

- Internal interrupts (software): These occur when the processor detects an error condition while executing an instruction (e.g., division by 0).
- External Interrupts (Hardware): Caused by an external event.

Note that traps are a special kind of internal interrupts. Another way of categorizing interrupts is by their maskability, which was covered in lecture 2.

Internal Interrupts Example

An internal interrupt occurs when the processor detects an error condition while executing an instruction. Unlike external interrupts, internal interrupts do not occur at random times during program execution. An important example is seg fault or page fault, which will be discussed later in the course.

Sometimes, not all the memory a program uses is currently in main memory; the data might be on disk and must be fetched to main memory before use. Suppose a program requests data that is not in main memory—an exception triggers the OS to fetch the data

from disk and load it into main memory. The program gets its data without even knowing that an exception has occurred. This is a page fault.

External Interrupts Example

This example illustrates what happens when you move your mouse while your computer is running a program:

- Assume we run the following program:

```
add r1, r2
sub r3, r4
and r5, r3
```

- As execution reaches this code, a user moves the mouse → an interrupt is triggered.
 1. Based on the time of the movement (in the middle of `sub`), hardware completes `add` and `sub`, but postpones `and` (for now).
 2. The handler starts:
 - The screen pointer (the little arrow) is moved
 3. The handler finishes
 4. Execution resumes with `and`.

Interrupt Handling

Handling an interrupt involves a coordinated sequence of hardware and operating system actions:

1. Interrupt is triggered:

- For internal interrupts (exceptions), the CPU detects conditions within itself or the running program (division by zero, invalid memory access, etc.)
- For external interrupts, peripheral devices send electronic signals through dedicated interrupt lines

2. State preservation:

- The processor automatically saves critical state information
- This includes the program counter (PC) value and processor status register
- Additional register contents may be saved by either hardware or the interrupt handler

3. Interrupt vector lookup:

- The processor uses the interrupt type to index into the interrupt vector table

- This table, stored at a predefined memory location, contains addresses of interrupt handler routines
- As shown in the diagram, the appropriate handler address is retrieved

4. **Switch to kernel mode:**

- The CPU switches from user mode to kernel mode (also called supervisor or privileged mode)
- This mode transition is essential because:
 - Interrupt handlers need privileged access to hardware resources
 - Security requires that user programs cannot directly access hardware or critical memory regions
 - Kernel mode allows execution of privileged instructions needed for system management
 - It protects the integrity of the system during interrupt handling

5. **Handler execution:**

- The program counter (PC) is loaded with the address from the interrupt vector
- Control transfers to the operating system's interrupt handler function
- The handler performs device-specific or exception-specific operations

6. **Return from interrupt:**

- After servicing the interrupt, the saved state is restored
- The CPU returns to user mode if the interrupted process was in user mode
- Execution resumes from the point of interruption

The kernel mode transition is a fundamental protection mechanism in modern computer systems. Without this transition, interrupt handling would create significant security vulnerabilities, as malicious user programs could potentially craft false interrupts to gain access to protected system resources or other users' memory spaces. The mode switch ensures that only trusted system code handles sensitive operations during interrupts.

Debug Breakpoint Mechanism - INT3

INT3 Instruction

The INT3 instruction is a special one-byte instruction (opcode `0xCC`) in x86 architecture that is designed specifically for debugging purposes:

- **Purpose:** Used by debuggers to set code breakpoints in a running program
- **Mechanism:** Debuggers temporarily replace an original instruction with INT3

- **Effect:** When executed, it generates a special interrupt that halts program execution
- **OS Role:** The operating system catches this interrupt and transfers control to the attached debugger

Implementation Details

When a debugger sets a breakpoint at address **X**:

1. The debugger saves the original byte at address **X**
2. It replaces this byte with 0xCC (the INT3 opcode)
3. When program execution reaches address **X**, the processor executes INT3
4. This triggers interrupt 3, causing the OS to stop the process
5. The debugger regains control, restores the original instruction for inspection, and allows debugging

Usage in C Programming

In C programs, you can manually trigger an INT3 breakpoint using inline assembly:

```
__asm__("int3"); // Triggers a breakpoint when this line executes
```

Example

Let's look at the following example to see what INT3 does.

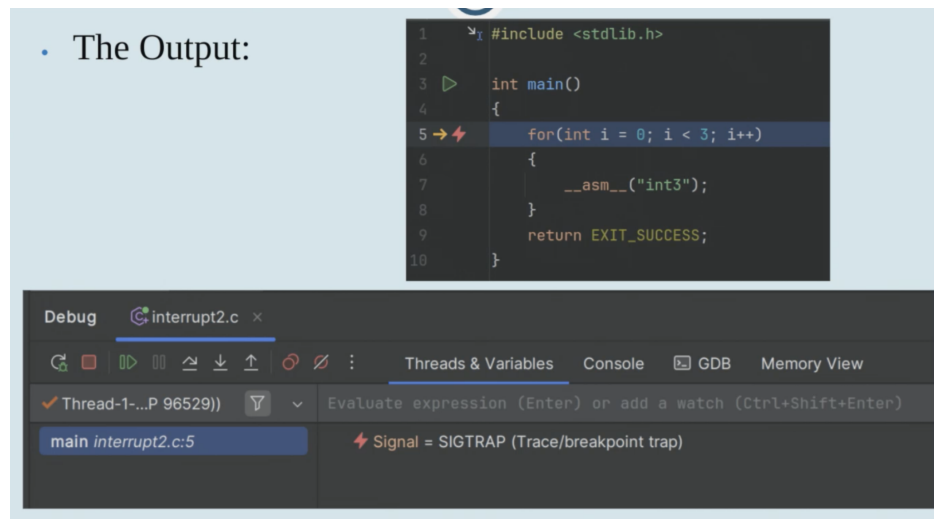


Figure 7: INT3 Example - GDB

The above figure demonstrates INT3 in action with the GNU Debugger (GDB). The code shows a simple C program that executes an INT3 instruction using inline assembly within a for loop. When this program runs under a debugger:

- The top panel shows the source code with a breakpoint at line 5 (indicated by the arrow and highlighted line).
- The bottom panel shows the debugger's interface, where execution has been paused after encountering the INT3 instruction.
- The signal "SIGTRAP (Trace/breakpoint trap)" confirms that the program was halted by a breakpoint interrupt.
- This example illustrates how the INT3 instruction allows debuggers to halt program execution at specific points, enabling developers to inspect program state and trace execution flow.