1. **Testing**

Testing is a complicated field with varying terminology and organizational structures. The Rust community divides testing into two categories: unit tests and integration tests. Unit tests are smaller and more concentrated, focusing on one module at a time and can test private interfaces. Integration tests are completely independent of a library and utilise the code in the same manner that any other external programs would, by just utilizing the public interface and possibly exercising many modules per test [1].

* 1. **Unit Testing**

The goal of unit tests is to isolate each unit of code from the rest of the code in order to rapidly identify where code is and not operating as intended. Unit tests will be placed in the **src** directory in each file that contains the code being tested. The convention is to include the test routines in a module named tests in each file and to annotate the module with **cfg (test)** [1].

#### **1.1.a. [The Tests Module and #[cfg(test)]](https://doc.rust-lang.org/book/ch11-03-test-organization.html" \l "the-tests-module-and-cfgtest)**

The #[cfg(test)] declaration on the tests component instructs Rust to compile and execute the test code only when cargo test is performed, not when cargo build is executed. When user merely wish to create the library, this reduces compilation time and space in the resultant generated artifact because the tests are not included. Because integration tests are stored in a separate directory, they do not require the #[cfg(test)] annotation. However, because unit tests are stored in the same files as the code, user will use #[cfg(test)] to tell the compiler not to include them in the built output [1].

#### **1.1.b. [Testing Private Functions](https://doc.rust-lang.org/book/ch11-03-test-organization.html" \l "testing-private-functions)**

There is disagreement in the testing profession about whether or not private functions should be tested properly, and some languages make testing private functions difficult or impossible. Whatever testing theory users follow, Rust's privacy rules make it possible to test private functions [1].

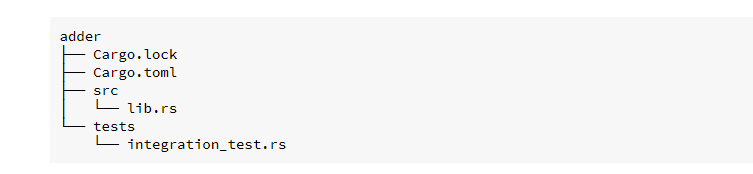
### **[Integration Tests](https://doc.rust-lang.org/book/ch11-03-test-organization.html" \l "integration-tests)**

Integration tests in Rust are completely independent of the library. They utilize the library in the same manner that any other programs would, which means they can only use functions that are part of the public API of any library. Their function is to ensure that numerous sections of the library work well together. Units of code that perform successfully on their own may fail when combined, therefore test coverage of the integrated code is also critical. A tests directory is required to be able to write integration tests [1].

#### **1.2.a. [The "tests" Directory](https://doc.rust-lang.org/book/ch11-03-test-organization.html" \l "the-tests-directory)**

Users add a tests directory alongside src at the top level of the project directory. Cargo knows to search in this directory for integration test files. Users can then create as many test files as they like, and Cargo will compile each one as a separate crate [1].

The below figure [1, Fig. 1] shows the structure of a directory.



*Figure 1: Structure of a Directory*

#### **1.2.b. [Submodules in Integration Tests](https://doc.rust-lang.org/book/ch11-03-test-organization.html" \l "submodules-in-integration-tests)**

As users add additional integration tests, users may want to create extra files in the tests directory to better organize them; for example, users may arrange the test functions by the feature they're evaluating. As previously stated, each file in the tests directory is created as its own independent crate, which is beneficial for generating various scopes to more precisely mimic how end users would use the crate. This implies that files in the tests directory do not behave the same way as files in the src directory [1].

The varied functionality of tests directory files is most obvious when user have a collection of helper calls to utilize in many integration test files and attempt to combine them into a common module [1].

#### **1.2.c. [Integration Tests for Binary Crates](https://doc.rust-lang.org/book/ch11-03-test-organization.html" \l "integration-tests-for-binary-crates)**

If the project is a binary crate with only a src/main.rs file and no src/lib.rs file, users can not construct integration tests in the tests directory and use statements to call functions specified in the src/main.rs file into scope. Only library crates expose functions for usage by other crates; binary crates are intended to function on their own [1].

One of the reasons why Rust projects that produce a binary have a simple src/main.rs file that calls logic in the src/lib.rs file. Using that structure, integration tests can validate the library crate's ability to provide critical functionality. If the critical feature works, the miniscule fraction of code in the src/main.rs file should also work, and that short amount of code does not need to be tested [1].

# **CPU Exceptions**

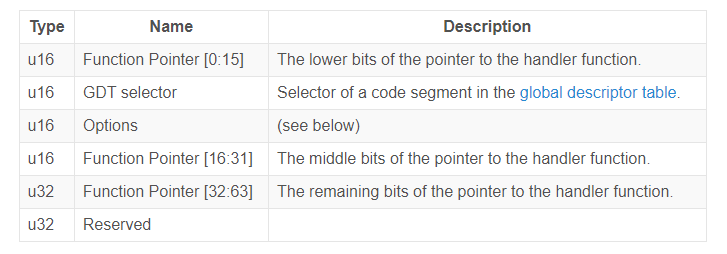
CPU exceptions arise in a variety of inappropriate scenarios, such as fetching an improper memory location or dividing by zero. To respond to them, we must create an interrupt descriptor table with handler routines [2].

On x86, there are around 20 different forms of CPU exceptions. The most significant are:

* **Page Fault** - Unauthorized memory accesses cause a page fault. For instance, if the current instruction attempts to read from an undiscovered page or writes to a read-only page [2].
* **Invalid Opcode - When users attempt to utilize new SSE instructions on an old CPU that does not support them, users get this error [2].**
* **General Protection Fault - This is the exception with the most diverse set of reasons. It can occur as a result of a variety of access breaches, such as attempting to execute an advantaged instruction in user-level code or writing restricted features in configuration records [2].**
* **Double Fault - When an exception occurs, the CPU attempts to invoke the appropriate handler code. If another exception exists even as the exception handler is being called, the CPU throws a double fault exception. This error also happens when no handler code for an exception is registered [2].**
* **Triple Fault - If an exception occurs while the CPU is attempting to execute the double fault handler code, a fatal triple fault is generated. Users can not  manage or recognize a triple fault. Most CPUs respond by restarting the operating system and resetting themselves [2].**

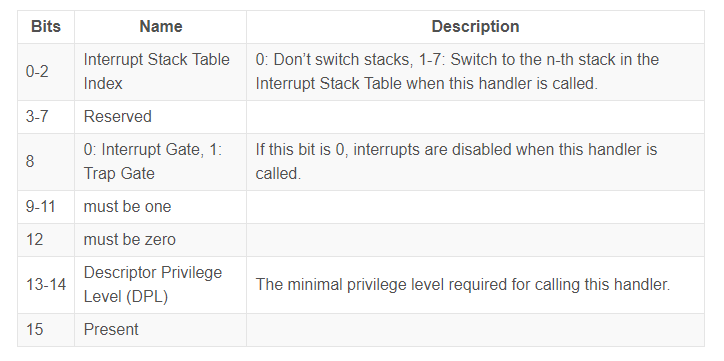
### **2.a. The Interrupt Descriptor Table**

To capture and handle exceptions, users must first create an Interrupt Descriptor Table (IDT). Users may define a handler function for each CPU exception in this table. Because the hardware relies on this table, users must adhere to a certain structure. Each item must have the 16-byte structure as shown below in [2, Fig. 2].



*Figure 2: The Interrupt Descriptor Table*

The format of the options field is as follows in [2, Fig. 3],



*Figure 3: The Option Field*

Each exception has its own IDT index. The erroneous opcode exception, for example, has table index 6, whereas the page fault exception has table index 14. As a result, for each exception, the hardware may automatically load the associated IDT item. The IDT indices of all exceptions are shown in the "Vector number." column of the OSDev wiki's Exception Table [2].

When an exception occurs, the CPU does the following roughly [2]:

1. Prompt some stack registers, such as the register and the RFLAGS register.
2. Examine the Interrupt Descriptor Table for the matching item (IDT). When a page fault occurs, for instance, the CPU examines the 14th item.
3. Verify the existence of the entry and, if not, issue a double fault.
4. If the entry is an interrupt entrance, restrict hardware interrupts (bit 40 not set).
5. Import the GDT selection supplied into the CS (code segment).
6. Navigate to the handler function given.

## **2.b. The Interrupt Calling Convention**

Exceptions are analogous to procedure calls: The CPU then performs the first instruction of the called function. Following that, the CPU jumps to the return address and resumes execution of the parent function [2].

Moreover, there is a significant distinction among exceptions and function calls: a function call is intentionally prompted by a compiler-inserted call instruction, but an exception can exist at any instruction. So need to explore at function calls in further depth to comprehend the implications of this discrepancy [2].

The specifics of a function call are specified by calling conventions. They indicate, for example, where function parameters are stored (e.g., in registers or on the stack) and how results are delivered [2]. For C functions (defined in the System V ABI), the following rules apply on x86 64 Linux:

* The first six integer inputs are supplied by registers rdi, rsi, rdx, rcx, r8, and r9.
* On the stack, further parameters are supplied.
* The results are returned in rax and rdx formats.

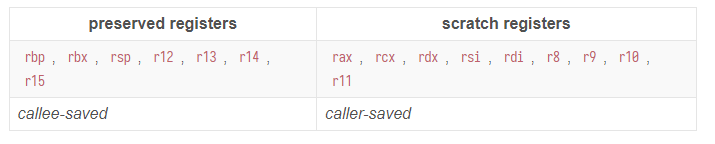
### **2.c. Preserved and Scratch Registers**

The calling convention separates registers into two categories: preserved registers and scratch registers [2].

The contents of preserved registers must not change between function calls. As a result, a called function (the "callee") may alter these registers only if it restores their original values before returning. Be a result, these registers are referred to as "callee-saved." It's usual practice to save these registers to the stack at the start of the function and reestablish them shortly before exiting [2].

A called function, on the other hand, has complete freedom to rewrite scratch registers. If the caller wishes to keep a scratch register's value across function calls, it must backup and restore it prior to the function call (e.g., by pushing it to the stack). As a result, the scratch registers are stored by the caller [2].

As seen in [2, Fig. 4] the C calling convention on x86 64 provides the following preserved and scratch registers:



*Figure 4: The Calling Convention*

Because the compiler is aware of these principles, it creates code in accordance with them. Most routines, for example, begin with a push rbp, which backups rbp on the stack (since it is a callee-saved register) [2].

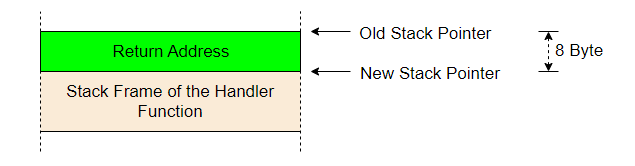
### **2.d. Preserving all Registers**

Exceptions, unlike function calls, can occur on any instruction. In most circumstances, users will not even know if the resultant code will throw an exception at compilation time. The compiler, for example, has no way of knowing if an instruction generates a stack overflow or a page fault [2].

Users can not backup any registers before an exception occurs since users will not know when it will occur. This implies users can not utilize a calling convention for exception handlers that relies on caller-saved registers. Instead, a calling convention that maintains all registers is required. As an example, the x86-interrupt calling convention ensures that all register values are restored to their original values upon function return [2].

### **2.e. The Interrupt Stack Frame**

The CPU transmits the return address before switching to the kernel function during a conventional function call (through the call instruction). The CPU releases this return address and moves to it on function return (via the ret instruction). As a result, the stack frame of a regular function call looks like this following figure [2, Fig. 5]:

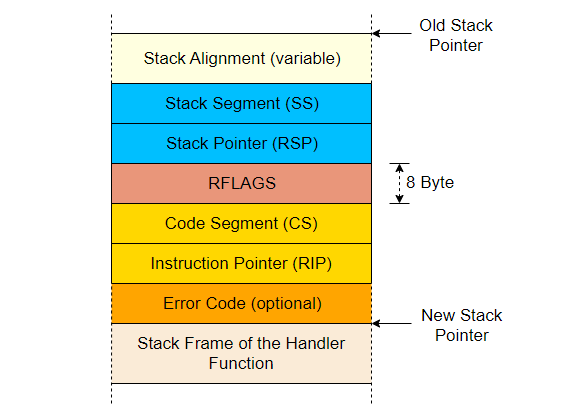


*Figure 5: The Interrupt Stack Frame of Normal Function*

Dragging a return address, on the other hand, would not sufficient for exception and interrupt handlers, because interrupt handlers sometimes execute in a distinct context (stack pointer, CPU flags, etc.). When an interrupt occurs, the CPU instead does the following [2]:

1. ****Saving the old stack pointer -** The CPU examines the contents of the stack pointer (rsp) and stack segment (ss) registers and stores them in an internal buffer.**
2. **Aligning the stack pointer - Because an interrupt can occur at any instruction, the stack pointer can also have any value. However, because some CPU instructions (for example, some SSE instructions) require the stack pointer to be positioned on a 16-byte boundary, the CPU executes this adjustment immediately after the interrupt.**
3. ****Switching stacks** (in some cases) -** When the CPU access transitions, such as when a CPU exception occurs in a user-mode application, a stack switch happens. The Interrupt Stack Table can also be used to configure stack switches for specific interruptions.
4. **Pushing the old stack pointer - The rsp and ss numbers from step 0 are pushed to the stack by the CPU. When heading back from an interrupt handler, this allows users to restore the original stack pointer.**
5. **Pushing and updating the**RFLAGS**register - Various control and status bits are stored in the RFLAGS register. When an interrupt occurs, the CPU modifies certain bits and transmits the previous value.**
6. **Pushing the instruction pointer - The CPU releases the instruction pointer (rip) and the code segment before proceeding to the interrupt handler procedure (cs). This is analogous to the return address push of a standard function call.**
7. ****Pushing an error code** (for some exceptions) -** The CPU sends an error code that identifies the reason of certain possible errors, including such page faults.
8. **Invoking the interrupt handler - The CPU receives the interrupt handler function's address and segment descriptor from the relevant field in the IDT. The handler is then called by placing the values into the rip and cs registers.**

**The below figure [2, Fig. 6] shows how looks like the interrupt stack frame is,**

****

*Figure 6: The Interrupt Stack Frame*

The InterruptStackFrame struct in the x86 64 crate represents the interrupt stack frame. It is supplied to interrupt handlers as &mut and can be used to get further information about the reason of the exception. Because only a few exceptions generate an error code, the struct lacks an error code field. These exceptions make use of the HandlerFuncWithErrCode function type, which has an extra error code parameter [2].

## 

# **Double Faults**

A double fault is a specific exception that happens when the CPU fails to call an exception handler. It happens, for example, when a page fault occurs but no page fault handler is registered in the Interrupt Descriptor Table (IDT). So it's comparable to catch-all blocks in exception-handling programming languages, such as catch(...) in C++ or catch(Exception e) in Java or C# [3].

A double fault works similarly to a standard exception. It has the vector number 8, and we may construct a standard handler function in the IDT for it. It is critical to provide a double fault handler because an unhandled double fault results in a deadly triple fault. Triple faults are not detectable, and most hardware responds with a system reset [3].

### **3.a. Triggering a Double Fault**

To write to the invalid address 0xdeadbeef, simply utilize unsecured. A page fault arises when the virtual address is not mapped to a physical address in the page tables. Because users did not register a page fault handler in the IDT, a double fault occurred [3].

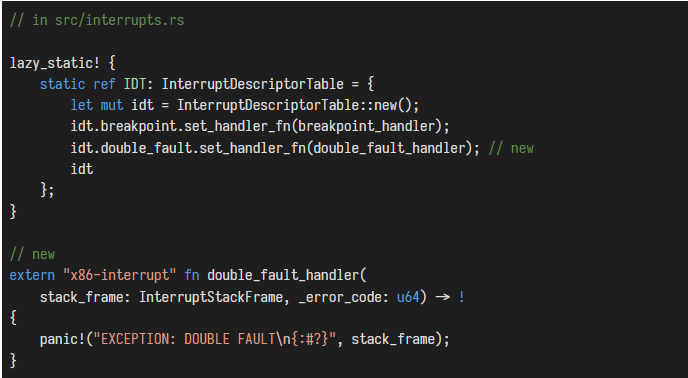
When users restart the kernel, users find that it goes into an infinite boot loop. The boot loop is caused by the following [3]:

1. The CPU attempts to write to 0xdeadbeef, resulting in a page fault.
2. The CPU examines the associated IDT item and notices that no handler function is given. As a result, it is unable to contact the page fault handler, resulting in a double fault.
3. The CPU examines the double fault handler's IDT entry, but this entrance, too, does not provide a handler function. As a result, a triple fault develops.
4. A triple error is deadly. QEMU responds to it in the same way that most genuine hardware does, by performing a system reset.

To avoid the triple fault, users must either provide a handler method for page faults or a double fault handler. To avoid triple faults in all circumstances,   just begin with a double fault handler that is called for all unhandled exception types [3].

## **3.b. A Double Fault Handler**

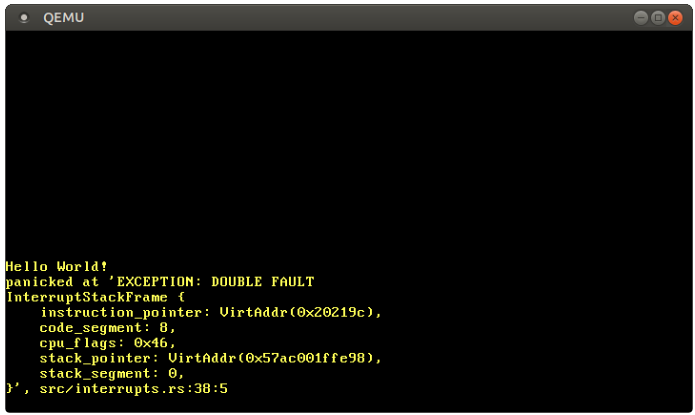
A double fault is a regular exception with an error code, therefore users can use the same handler method as previously did for the breakpoint [3]:



*Figure 7: A Double Fault Handler*

As illustrated in [3, Fig. 7], the handler sends a brief error message before dumping the exception stack frame. There is no purpose to report the error code of the double fault handler because it is always zero. The double fault handler differs from the breakpoint handler in that it is divergent. The x86 64 architecture does not support returning from a double fault exception [3].

When restart the kernel, users should observe the double fault handler being called as in [3, Fig. 8],



*Figure 8: Invoked Double Fault Handler*

What happened on above figure is [3],

1. The CPU attempts to write to 0xdeadbeef, resulting in a page fault.
2. The CPU, like previously, examines the appropriate element in the IDT and notices that no handler function is declared. As a result, a double fault arises.
3. The CPU advances to the now-present double fault handler.

Because the CPU may now invoke the double fault handler, the triple fault (and the boot-loop) no longer happens [3].

**References**

1. “The Rust Programming Language, ”. Accessed on: 29 October 2022. [Online]. Available: <https://doc.rust-lang.org/book/ch11-03-test-organization.html>
2. Philipp Oppermann, “Writing an OS in Rust, ” 17 June 2018. Accessed on: 01 November 2022. [Online]. Available: <https://os.phil-opp.com/cpu-exceptions/>
3. Philipp Oppermann, “Writing an OS in Rust, ” 18 June 2018. Accessed on: 01 November 2022. [Online]. Available: <https://os.phil-opp.com/double-fault-exceptions/>