WIKIPEDIA

Segmentation fault

In computing, a **segmentation fault** (often shortened to **segfault**) or **access violation** is a fault, or failure condition, raised by hardware with memory protection, notifying an operating system (OS) the software has attempted to access a restricted area of memory (a memory access violation). On standard x86 computers, this is a form of general protection fault. The OS kernel will, in response, usually perform some corrective action, generally passing the fault on to the offending process by sending the process a signal. Processes can in some cases install a custom signal handler, allowing them to recover on their own,[1] but otherwise the OS default signal handler is used, generally causing abnormal termination of the process (a program crash), and sometimes a core dump.

Segmentation faults are a common class of error in programs written in languages like C that provide low-level memory access. They arise primarily due to errors in use of pointers for virtual memory addressing, particularly illegal access. Another type of memory access error is a bus error, which also has various causes, but is today much rarer; these occur primarily due to incorrect *physical* memory addressing, or due to misaligned memory access – these are memory references that the hardware *cannot* address, rather than references that a process is not *allowed* to address.

Many programming languages may employ mechanisms designed to avoid segmentation faults and improve memory safety. For example, the Rust programming language employs an 'Ownership'[2] based model to ensure memory safety.[3] Other languages, such as Lisp and Java, employ garbage collection,[4] which avoids certain classes of memory errors that could lead to segmentation faults.[5]

Contents

Overview

Causes

Handling

Examples

Writing to read-only memory

Null pointer dereference

Buffer overflow

Stack overflow

See also

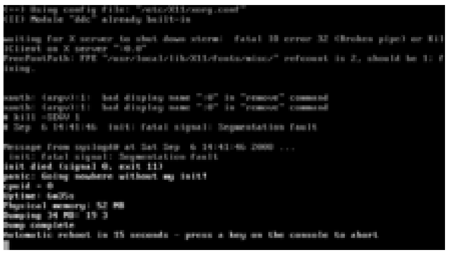
References

External links

Overview

A segmentation fault occurs when a program attempts to access a memory location that it is not allowed to access, or attempts to access a memory location in a way that is not allowed (for example, attempting to write to a read-only location, or to overwrite part of the operating system).

The term "segmentation" has various uses in computing; in the context of "segmentation fault", a term used since the 1950s, it refers to the address space of a *program.* With memory protection, only the program's own address space is readable, and of this, only the stack and the read/write portion of the data segment of a program are writable, while read-only data and the code segment are not writable. Thus attempting to read outside of the program's address space, or writing to a read-only segment of the address space, results in a segmentation fault, hence the name.



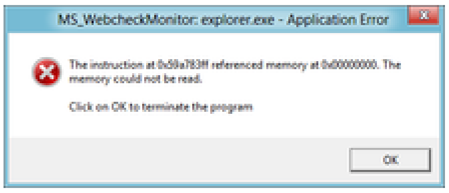
```
(--) Using config file: "/etc/X11/xorg.conf"
(II) Module "ddc" already built-in

waiting for X server to shut down xterm: fatal IO error 32 (Broken pipe) or Kil
lClient on X server ":0.0"
FreeFontPath: FPE "/usr/local/lib/X11/fonts/misc/" refcount is 2, should be 1; f
ixing.

xauth: (argv):1: bad display name ":0" in "remove" command
xauth: (argv):1: bad display name ":0" in "remove" command
# kill -SEGV 1
# Sep 6 14:41:46 init: fatal signal: Segmentation fault

Message from syslogd@ at Sat Sep 6 14:41:46 2008 ...
 init: fatal signal: Segmentation fault
init died (signal 0, exit 11)
panic: Going nowhere without my init!
cpuid = 0
Uptime: 6m35s
Physical memory: 52 MB
Dumping 34 MB: 19 3
Dump complete
Automatic reboot in 15 seconds - press a key on the console to abort
```

Example of human generated signal



MS_WebcheckMonitor: explorer.exe - Application Error

The instruction at 0x59a783ff referenced memory at 0x00000000. The memory could not be read.

Click on OK to terminate the program

A null pointer dereference on Windows 8

On systems using hardware memory segmentation to provide virtual memory, a segmentation fault occurs when the hardware detects an attempt to refer to a non-existent segment, or to refer to a location outside the bounds of a segment, or to refer to a location in a fashion not allowed by the permissions granted for that segment. On systems using only paging, an invalid page fault generally leads to a segmentation fault, and segmentation faults and page faults are both faults raised by the virtual memory management system. Segmentation faults can also occur independently of page faults: illegal access to a valid page is a segmentation fault, but not an invalid page fault, and segmentation faults can occur in the middle of a page (hence no page fault), for example in a buffer overflow that stays within a page but illegally overwrites memory.

At the hardware level, the fault is initially raised by the memory management unit (MMU) on illegal access (if the referenced memory exists), as part of its memory protection feature, or an invalid page fault (if the referenced memory does not exist). If the problem is not an invalid logical address but instead an invalid physical address, a bus error is raised instead, though these are not always distinguished.

At the operating system level, this fault is caught and a signal is passed on to the offending process, activating the process's handler for that signal. Different operating systems have different signal names to indicate that a segmentation fault has occurred. On Unix-like operating systems, a signal called SIGSEGV (abbreviated from *segmentation violation*) is sent to the offending process. On Microsoft Windows, the offending process receives a STATUS_ACCESS_VIOLATION exception.

Causes

The conditions under which segmentation violations occur and how they manifest themselves are specific to hardware and the operating system: different hardware raises different faults for given conditions, and different operating systems convert these to different signals that are passed on to processes. The proximate cause is a memory access violation, while the underlying cause is generally a software bug of some sort. Determining the root cause – debugging the bug – can be simple in some cases, where the program will consistently cause a segmentation fault (e.g., dereferencing a null pointer), while in other cases the bug can be difficult to reproduce and depend on memory allocation on each run (e.g., dereferencing a dangling pointer).

The following are some typical causes of a segmentation fault:

- Attempting to access a nonexistent memory address (outside process's address space)
- Attempting to access memory the program does not have rights to (such as kernel structures in process context)
- Attempting to write read-only memory (such as code segment)

These in turn are often caused by programming errors that result in invalid memory access:

- Dereferencing a null pointer, which usually points to an address that's not part of the process's address space
- Dereferencing or assigning to an uninitialized pointer (wild pointer, which points to a random memory address)
- Dereferencing or assigning to a freed pointer (dangling pointer, which points to memory that has been freed/deallocated/deleted)
- A buffer overflow
- A stack overflow
- Attempting to execute a program that does not compile correctly. (Some compilers will output an executable file despite the presence of compile-time errors.)

In C code, segmentation faults most often occur because of errors in pointer use, particularly in C dynamic memory allocation. Dereferencing a null pointer will always result in a segmentation fault, but wild pointers and dangling pointers point to memory that may or may not exist, and may or may not be readable or writable, and thus can result in transient bugs. For example:

```
char *p1 = NULL;           // Null pointer
char *p2;                  // Wild pointer: not initialized at all.
char *p3  = malloc(10 * sizeof(char));  // Initialized pointer to allocated memory
                                        // (assuming malloc did not fail)
free(p3);                  // p3 is now a dangling pointer, as memory has been freed
```

Now, dereferencing any of these variables could cause a segmentation fault: dereferencing the null pointer generally will cause a segfault, while reading from the wild pointer may instead result in random data but no segfault, and reading from the dangling pointer may result in valid data for a while, and then random data as it is overwritten.

Handling

The default action for a segmentation fault or bus error is abnormal termination of the process that triggered it. A core file may be generated to aid debugging, and other platform-dependent actions may also be performed. For example, Linux systems using the grsecurity patch may log SIGSEGV signals in order to monitor for possible intrusion attempts using buffer overflows.

Examples

Writing to read-only memory

Writing to read-only memory raises a segmentation fault. At the level of code errors, this occurs when the program writes to part of its own code segment or the read-only portion of the data segment, as these are loaded by the OS into read-only memory.

Here is an example of ANSI C code that will generally cause a segmentation fault on platforms with memory protection. It attempts to modify a string literal, which is undefined behavior according to the ANSI C standard. Most compilers will not catch this at compile time, and instead compile this to executable code that will crash:

```
int main(void)
{
    char *s = "hello world";
    *s = 'H';
}
```

When the program containing this code is compiled, the string "hello world" is placed in the rodata section of the program executable file: the read-only section of the data segment. When loaded, the operating system places it with other strings and constant data in a read-only segment of memory. When executed, a variable, *s*, is set to point to the string's location, and an attempt is made to write an *H* character through the variable into the memory, causing a segmentation fault. Compiling such a program with a compiler that does not check for the assignment of read-only locations at compile time, and running it on a Unix-like operating system produces the following runtime error:

```
$ gcc segfault.c -g -o segfault
$ ./segfault
Segmentation fault
```

Segmentation fault on an EMV keypad

Backtrace of the core file from GDB:

```
Program received signal SIGSEGV, Segmentation fault.
0x1c0005c2 in main () at segfault.c:6
6               *s = 'H';
```

This code can be corrected by using an array instead of a character pointer, as this allocates memory on stack and initializes it to the value of the string literal:

```
char s[] = "hello world";
s[0] = 'H';  // equivalently, *s = 'H';
```

Even though string literals should not be modified (this has undefined behavior in the C standard), in C they are of `static char []` type,[6][7][8] so there is no implicit conversion in the original code (which points a `char *` at that array), while in C++ they are of `static const char []` type, and thus there is an implicit conversion, so compilers will generally catch this particular error.

Null pointer dereference

In C and C-like languages, null pointers are used to mean "pointer to no object" and as an error indicator, and dereferencing a null pointer (a read or write through a null pointer) is a very common program error. The C standard does not say that the null pointer is the same as the pointer to memory address 0, though that may be the case in practice. Most operating systems map the null pointer's address such that accessing it causes a segmentation fault. This behavior is not guaranteed by the C standard. Dereferencing a null pointer is undefined behavior in C, and a conforming implementation is allowed to assume that any pointer that is dereferenced is not null.

```
int *ptr = NULL;
printf("%d", *ptr);
```

This sample code creates a null pointer, and then tries to access its value (read the value). Doing so causes a segmentation fault at runtime on many operating systems.

Dereferencing a null pointer and then assigning to it (writing a value to a non-existent target) also usually causes a segmentation fault:

```
int *ptr = NULL;
*ptr = 1;
```

The following code includes a null pointer dereference, but when compiled will often not result in a segmentation fault, as the value is unused and thus the dereference will often be optimized away by dead code elimination:

```
int *ptr = NULL;
*ptr;
```

Buffer overflow

Stack overflow

Another example is recursion without a base case:

```
int main(void)
{
    main();
    return 0;
}
```

which causes the stack to overflow which results in a segmentation fault.[9] Infinite recursion may not necessarily result in a stack overflow depending on the language, optimizations performed by the compiler and the exact structure of a code. In this case, the behavior of unreachable code (the return statement) is undefined, so the compiler can eliminate it and use a tail call optimization that might result in no stack usage. Other optimizations could include translating the recursion into iteration, which given the structure of the example function would result in the program running forever, while probably not overflowing its stack.

See also

- Core dump
- General protection fault
- Page fault
- Storage violation
- Guru Meditation

References

1. *Expert C programming: deep C secrets* By Peter Van der Linden, page 188
2. The Rust Programming Language - Ownership (http://doc.rust-lang.org/book/ownership.html)
3. Fearless Concurrency with Rust - The Rust Programming Language Blog (http://blog.rust-lang.org/2015/04/10/Fearless-Concurrency.html)
4. McCarthy, John (April 1960). "Recursive functions of symbolic expressions and their computation by machine, Part I" (http://www-formal.stanford.edu/jmc/recursive.html). *Communications of the ACM*. **4** (3): 184–195. Retrieved 2018-09-22.

5. Dhurjati, Dinakar; Kowshik, Sumant; Adve, Vikram; Lattner, Chris (1 January 2003). "Memory Safety Without Runtime Checks or Garbage Collection" (http://llvm.org/pubs/2003-05-05-LCTES03-CodeSafety.pdf) (PDF). *Proceedings of the 2003 ACM SIGPLAN Conference on Language, Compiler, and Tool for Embedded Systems*. ACM: 69–80. doi:10.1145/780732.780743 (https://doi.org/10.1145%2F780732.780743). ISBN 1581136471. Retrieved 2018-09-22.
6. "6.1.4 String literals". *ISO/IEC 9899:1990 - Programming languages -- C*.
7. "6.4.5 String literals". *ISO/IEC 9899:1999 - Programming languages -- C*.
8. "6.4.5 String literals". *ISO/IEC 9899:2011 - Programming languages -- C* (http://www.iso-9899.info/n1570.html#6.4.5p6).
9. What is the difference between a segmentation fault and a stack overflow? (https://stackoverflow.com/questions/2685413/what-is-the-difference-between-a-segmentation-fault-and-a-stack-overflow/2685434#2685434) at Stack Overflow

External links

- Process: focus boundary and segmentation fault (https://www.encious.com/process)
- A FAQ: User contributed answers regarding the definition of a segmentation fault (http://www.faqs.org/qa/qa-673.html)
- A "null pointer" explained (http://c-faq.com/null/null1.html)
- Answer to: NULL is guaranteed to be 0, but the null pointer is not? (http://c-faq.com/null/varieties.html)
- The Open Group Base Specifications Issue 6 signal.h (http://www.opengroup.org/onlinepubs/009695399/basedefs/signal.h.html)