Problem 7: Welcome to Recursion Hell [Email Submission]

By Ruijie Fang

1 Background: Interpreted vs. Native Languages

Mainstream programming languages largely fall into two categories: *interpreted*, or *native*. Languages such as Java, Python, Ruby, and Erlang are interpreted, meaning they are compiled into some intermediate bytecode that isn't directly executable on a CPU, and the bytecode is then executed on an *interpreter*, which can be seen as a "virtual machine" that runs the bytecode instructions line-by-line. For Java, the interpreter is called the Java Virtual Machine (JVM), which uses a stack-based architecture, meaning that it is register-free and uses a stack and push/pop operations to execute bytecode. Erlang has the BEAM virtual machine (in contrast to the JVM, BEAM uses a register-based architecture; a register-based architecture contains *virtual registers*, which, while making it more efficient and akin to a physical CPU, makes code generation harder since the compiler would have to solve the *register allocation problem*[1]); for Python and Ruby, the default implementation combines a source code-to-bytecode compiler and a bytecode interpreter, so that one can execute a Python/Ruby source file within one command. This is in contrast to *native* languages such as C, C++, Go, or Rust, which are compiled down to machine code and executed directly on the target CPU.

The benefits of an interpreted language is obvious: Once the input program is compiled into bytecode, it is portable between different platforms and across different CPU architectures, as long as an interpreter is available on that architecture. For instance, once we compile a Java source file down to a Java bytecode file, we can run it across different CPU architectures (x86, ARMv8, Sun SPARC, Itanium, MIPS) as long as there is a JVM interpreter implementation for that platform.

Java, in contrast to C/C++, is also generally considered more *debuggable*: for instance, at Princeton it is the language-of-choice in COS 126, instead of C or C++. One particularly interesting feature of the JVM is the `StackOverflowException`: Similar to other languages, JVM limits the recursion depth. When a recursion is too deep (e.g. `fib(1 << 26)` where `fib` is the recursive Fibonacci function, the JVM throws a `StackOverflowException` and a stack trace, telling you specifically where in your source code you made the deeply recursive function call.

[1] Prof. Andrew Appel, of Princeton COS, has made some important contributions to the solution of this problem.

C, on the other hand, is less friendly. If you compile a C file that contains a deep recursion, quite possibly running the resultant binary will result in a `segmentation fault` and no other information, making it difficult to debug where everything went wrong. I'm sure many of you can sympathize (cough, cough, COS 217).

We will consider the following two subproblems in the C programming language. First, given **only** the declaration of a deeply recursive function as a header file, write a *wrapper* function that calls it such that the deep recursion call can be performed correctly without inducing a segmentation fault. The challenge? *Do it without changing the actual implementation.* Second, for any general recursion, write a *wrapper* function such that when the recursion exceeds a given depth, output a line containing "StackOverflowException" instead of inducing a segmentation fault. **You might wonder if this is possible!** The answer is *yes. Doing any part gets you 15 points. Doing both parts correctly get you 30 points.*

2 Part A: Make a Deeply Recursive Function Run (15 Points)

You are given a deeply recursive function signature `unsigned long long BadF(unsigned long long n, unsigned long long p, unsigned long long k)` without access to its implementation. Write a C file containing the function `unsigned long long BadFHandler(unsigned long long n, unsigned long long p, unsigned long long k)` that performs some magic and returns the answer of calling `BadF(n,p,k)`, which by default results in a segmentation fault because it recurses too deep.

We'll compile your solution in `gcc-7.5.0` with `-fPIC -O0` flags and call `BadFHandler` with some test inputs to see if `BadFHandler(n,p,k)` yields the same result as `BadF(n,p,k)`. **To start,** write a C file named `PartA.c` with the following structure:

```
// declaration of the deeply recursive function.
unsigned long long BadF(unsigned long long n,
unsigned long long p, unsigned long long k);

// Your implementation of a Wrapper function.
unsigned long long BadFHandler(unsigned long long n, unsigned long long p,
    unsigned long long k) {
    // some code here that calculates the result of BadF(n,p,k) using magic...
    return BadF(n,p,k); // this will result in segfaults by default...
}
```

Assumptions: A grading machine will be configured for this problem. The target machine runs an x86-64 CPU and Ubuntu 16.04 Operating System. On this platform `unsigned/int` are 32 bits wide, `char` is a single byte and `unsigned long long` is 64-bits wide. Your code will be run using the `gcc 11.1.0` compiler with flags `-O0 -fPIC`. You are **not permitted to include any header files when submitting**, although we will help you include the following header files: `stdlib.h, memory.h, signal.h, unistd.h, setjmp.h, string.h, stdio.h`. *Usage of any other header file is strictly prohibited and will not make your code compile.* **Please read page 5 to look over the technical details of this setup — there will be some compiler-specific information that will help your submission.**

Grading and How to Submit

Email your completed file to `coscon21.interactive.grader@gmail.com` with attachment `PartA.c` as the *exact file name* and with `P7PartA` as the *exact email subject* and wait for a reply. Your email must also be sent from your Princeton address. Emailing multiple times is fine, although sending too many emails over a short period will result in a lockout and potential disqualification.

The mailbox is configured with an aggressive spam filter to filter out any non-princeton.edu email or email from any account sent too frequently over a short period. Note that inlining assembly is permitted (i.e. the use of `asm, volatile asm` keywords are fine).

Hint. A two-line solution in C exists for this problem. **Hint 2.** Feel free to use architecture-dependent assumptions. You may also assume the process is being run on a single thread. **Hint++.** A solution in assembly is probably safer, but our explanation in the end of this problem explains why you can also do it in C with `gcc-7.5.0` (and indeed, any old version of `gcc`).

3 Part B: Handle StackOverflow (15 Points)

The setup is same as Part A, except we introduce a new library function with signature

```
void StackOverflowException(unsigned long long n,
                            unsigned long long p, unsigned long long q)
```

Whenever naively calling `BadF(n,p,k)` will result in a segfault (due to a stack overflow), you should invoke the above function in `BadFHandler` to report the exception, and return 0. Otherwise, you should return the correct value of `BadF`. The template for Part B is as follows (next page):


```
// declaration of the deeply recursive function.
unsigned long long BadF(unsigned long long n,
                        unsigned long long p, unsigned long long k);

// If calling BadF(n,p,k) results in a stack overflow exception, call this
// function and use your code from part A to return the value of BadF(n,p,k).
void StackOverflowException(unsigned long long n,
                            unsigned long long p, unsigned long long k);

// Your implementation of a Wrapper function.
unsigned long long BadFHandler(unsigned long long n,
                               unsigned long long p, unsigned long long k) {
    // TODO: Decide when to call StackOverflowException(n, p, k).

    // some code here that calculates
    // the result of BadF(n,p,k) using magic from part A...

    return BadF(n,p,k); // without changes, this will result in a segfault
}
```

For this part, feel free to use Google and/or consult `man` pages if it helps. If you code up everything in C, the solution for this part is less compiler-dependant than the previous part: We tested our solution across four different CPUs (Intel Xeon x64, AMD EPYC x64, Intel Core i5 x64, and Apple M1 arm64) and at least four different clang/gcc versions, including gcc-7.5.0 on amd64, gcc-11.1.0 on Apple M1, gcc-9.3.0 on amd64 and x86.

Grading and How to Submit

Email your completed file to `coscon21.interactive.grader@gmail.com` with attachment `PartB.c` as the *exact file name* and with `P7PartB` as the *exact email subject* and wait for a reply. Your email must also be sent from your Princeton address. Emailing multiple times is fine, although sending too many emails over a short period will result in a lockout and potential disqualification.

Some Technical Notes

When writing up this problem, we found that newer versions of gcc, as well as clang, compile C99-style variable-length stack-allocated arrays inconsistently across different platforms. For instance, one might imagine the following program using gcc -O0:

```
int main()
{
int i = 16384 / 2 * 2;
char big[i]; return 0;
}
```

Compiles to assembly that executes constant number of instructions, where the program offsets `i` against `rsp` to allocate the space needed for `big`. However, we found that newer versions of gcc, as well as some versions of clang prefer to do it in a loop, extending `rsp` by a page boundary every iteration:

```
<truncated code above>
.L2:
cmpq %rdx, %rsp
je .L3
subq $4096, %rsp
orq $0, 4088(%rsp)
jmp .L2
.L3:
<truncated code below>
```

We reassure you that gcc-7.5.0 on our x86-64 Ubuntu 20.04 environment will not have the above behavior. Instead gcc-7.5.0 compiles the above C code to something like the following (beginning on next page). **If you're in doubt, maybe a solution utilizing inline assembly is better.** The next few pages will also contain detailed system information and gcc version information of our grading server to help facilitate your debugging (although likely it isn't needed). See the next page for what the above C code compiles to on our machine.


```
main:   pushq %rbp
      movq %rsp, %rbp
      subq $32, %rsp
      movq %fs:40, %rax
      movq %rax, -8(%rbp)
      xorl %eax, %eax
      movq %rsp, %rax
      movq %rax, %rcx
      movl $16384, -28(%rbp)
      movl -28(%rbp), %eax
      movslq %eax, %rdx
      subq $1, %rdx
      movq %rdx, -24(%rbp)
      movslq %eax, %rdx
      movq %rdx, %r8
      movl $0, %r9d
      movslq %eax, %rdx
      movq %rdx, %rsi
      movl $0, %edi
      cltq
      movl $16, %edx
      subq $1, %rdx
      addq %rdx, %rax
      movl $16, %edi
      movl $0, %edx
      divq %rdi
      imulq $16, %rax, %rax
      subq %rax, %rsp
      movq %rsp, %rax
      addq $0, %rax
      movq %rax, -16(%rbp)
      movl $0, %eax
      movq %rcx, %rsp
      movq -8(%rbp), %rdi
      xorq %fs:40, %rdi
      je .L3
```

`gcc -v` Output

```
Using built-in specs.
COLLECT_GCC=gcc-7
COLLECT_LTO_WRAPPER=/usr/lib/gcc/x86_64-linux-gnu/7/lto-wrapper
OFFLOAD_TARGET_NAMES=nvptx-none
OFFLOAD_TARGET_DEFAULT=1
Target: x86_64-linux-gnu
Configured with: ../src/configure -v --with-pkgversion='Ubuntu 7.5.0-6ubuntu2'
--with-bugurl=file:///usr/share/doc/gcc-7/README.Bugs
--enable-languages=c,ada,c++,go,brig,d,fortran,objc,obj-c++
--prefix=/usr --with-gcc-major-version-only --program-suffix=-7
--program-prefix=x86_64-linux-gnu- --enable-shared
--enable-linker-build-id --libexecdir=/usr/lib
--without-included-gettext --enable-threads=posix
--libdir=/usr/lib --enable-nls --enable-bootstrap
--enable-clocale=gnu --enable-libstdcxx-debug --enable-libstdcxx-time=yes
--with-default-libstdcxx-abi=new
--enable-gnu-unique-object --disable-vtable-verify
--enable-libmpx --enable-plugin --enable-default-pie
--with-system-zlib --with-target-system-zlib
--enable-objc-gc=auto --enable-multiarch --disable-werror
--with-arch-32=i686 --with-abi=m64
--with-multilib-list=m32,m64,mx32 --enable-multilib
--with-tune=generic --enable-offload-targets=nvptx-none
--without-cuda-driver --enable-checking=release
--build=x86_64-linux-gnu --host=x86_64-linux-gnu --target=x86_64-linux-gnu
Thread model: posix
gcc version 7.5.0 (Ubuntu 7.5.0-6ubuntu2)
```

Relevant OS Information

```
Distributor ID: Ubuntu
Description: Ubuntu 20.04.3 LTS
Release: 20.04
Codename: focal
```

Some Relevant CPU Information

Update: We had to switch the grading platform setup. The CPU for the grading platform is now either an Intel Xeon or an AMD EPYC CPU. Our solution does not depend on any Intel/AMD-specific instructions, nor assembly.