01-oppgave-beskrivelse.md

Operating Systems - Assignment 1

Task 1:

1. Describe user-mode and kernel-mode, and why.

A mode switch from kernel to user-mode must happen because user applications are not allowed to execute certain CPU instructions or access certain areas of memory directly. This is a security and stability measure to prevent user applications from interfering with the core functionality of the operating system.

2. Program "printx.c"

Manual References

Explanation: *Here you use `man 3 <function>` to find information about the functions you use in your code. The number 3 refers to the C library manual section.*

Manual References

- **`man 3 printf`**

```
  DESCRIPTION

  The  functions  in the printf() family produce output according to a format as described below.  The functions printf() and vprintf() write output to stdout, the standard
  output stream; fprintf() and vfprintf() write output to the given output stream; sprintf(), snprintf(), vsprintf(), and vsnprintf() write to the character string str.

  The function dprintf() is the same as fprintf() except that it outputs to a file descriptor, fd, instead of to a stdio(3) stream.

  The functions snprintf() and vsnprintf() write at most size bytes (including the terminating null byte ('\0')) to str.
  The functions vprintf(), vfprintf(), vdprintf(), vsprintf(), vsnprintf() are equivalent to the functions printf(), fprintf(), dprintf(),  sprintf(),  snprintf(),  respec-
  tively,  except  that  they  are  called with a va_list instead of a variable number of arguments.  These functions do not call the va_end macro.  Because they invoke the
  va_arg macro, the value of ap is undefined after the call.  See stdarg(3).

  All of these functions write the output under the control of a format string that specifies how subsequent arguments (or arguments accessed via the variable-length  argu-
  ment facilities of stdarg(3)) are converted for output.

  C99  and  POSIX.1-2001 specify that the results are undefined if a call to sprintf(), snprintf(), vsprintf(), or vsnprintf() would cause copying to take place between ob-
  jects that overlap (e.g., if the target string array and one of the supplied input arguments refer to the same buffer).  See CAVEATS.
```

- **`man 3 fgets`**

```
  DESCRIPTION
  fgetc() reads the next character from stream and returns it as an unsigned char cast to an int, or EOF on end of file or error.

  getc() is equivalent to fgetc() except that it may be implemented as a macro which evaluates stream more than once.

  getchar() is equivalent to getc(stdin).

  fgets()  reads  in at most one less than size characters from stream and stores them into the buffer pointed to by s.  Reading stops after an EOF or a newline.  If a new-
  line is read, it is stored into the buffer.  A terminating null byte ('\0') is stored after the last character in the buffer.

  ungetc() pushes c back to stream, cast to unsigned char, where it is available for subsequent read operations.  Pushed-back characters will be returned in reverse  order;
  only one pushback is guaranteed.

  Calls to the functions described here can be mixed with each other and with calls to other input functions from the stdio library for the same input stream.

  For nonlocking counterparts, see unlocked_stdio(3).
```

- **man 3 atoi**

  ```
    DESCRIPTION
    fgetc() reads the next character from stream and returns it as an
  unsigned char cast to an int, or EOF on end of file or error.

    getc() is equivalent to fgetc() except that it may be implemented as
  a macro which evaluates stream more than once.

    getchar() is equivalent to getc(stdin).

    fgets()  reads  in at most one less than size characters from stream
  and stores them into the buffer pointed to by s.  Reading stops after
  an EOF or a newline.  If a new-
    line is read, it is stored into the buffer.  A terminating null byte
  ('\0') is stored after the last character in the buffer.

    ungetc() pushes c back to stream, cast to unsigned char, where it is
  available for subsequent read operations.  Pushed-back characters will
  be returned in reverse  order;
    only one pushback is guaranteed.

    Calls to the functions described here can be mixed with each other
  and with calls to other input functions from the stdio library for the
  same input stream.

    For nonlocking counterparts, see unlocked_stdio(3).
  ```

- **man 3 sscanf**

  ```
    DESCRIPTION
    The  sscanf()  family of functions scans formatted input according to
  format as described below.  This format may contain conversion
  specifications; the results from such
    conversions, if any, are stored in the locations pointed to by the
  pointer arguments that follow format.  Each pointer argument must be of
  a type that is appropriate  for
    the value returned by the corresponding conversion specification.

    If  the  number of conversion specifications in format exceeds the
  number of pointer arguments, the results are undefined.  If the number
  of pointer arguments exceeds the
    number of conversion specifications, then the excess pointer
  arguments are evaluated, but are otherwise ignored.

    sscanf() These functions read their input from the string pointed to
  by str.
  ```

```
  The vsscanf() function is analogous to vsprintf(3).

  The format string consists of a sequence of directives which describe
how to process the sequence of input characters.  If processing of a
directive fails, no further in-
  put is read, and sscanf() returns.  A "failure" can be either of the
following: input failure, meaning that input characters were
unavailable, or matching failure,  mean-
  ing that the input was inappropriate (see below).

  A directive is one of the following:

  •      A sequence of white-space characters (space, tab, newline,
etc.; see isspace(3)).  This directive matches any amount of white
space, including none, in the input.

  •      An ordinary character (i.e., one other than white space or
'%').  This character must exactly match the next character of input.

  •      A  conversion specification, which commences with a '%'
(percent) character.  A sequence of characters from the input is
converted according to this specification,
          and the result is placed in the corresponding pointer
argument.  If the next item of input does not match the conversion
specification, the  conversion  fails—this
          is a matching failure.

  Each conversion specification in format begins with either the
character '%' or the character sequence "%n$" (see below for the
distinction) followed by:

  •      An  optional  '*'  assignment-suppression  character:
sscanf()  reads input as directed by the conversion specification, but
discards the input.  No corresponding
          pointer argument is required, and this specification is not
included in the count of successful assignments returned by scanf().

  •      For decimal conversions, an optional quote character (').
This specifies that the input number may include thousands' separators
as defined by the LC_NUMERIC cat-
          egory of the current locale.  (See setlocale(3).)  The quote
character may precede or follow the '*' assignment-suppression
character.

  •      An optional 'm' character.  This is used with string
conversions (%s, %c, %[), and relieves the caller of the need to
allocate a corresponding buffer to  hold  the
          input:  instead,  sscanf()  allocates  a buffer of sufficient
```

```
size, and assigns the address of this buffer to the corresponding
pointer argument, which should be a
          pointer to a char * variable (this variable does not need to
be initialized before the call).  The caller should subsequently
free(3) this buffer  when  it  is  no
          longer required.

  •      An optional decimal integer which specifies the maximum field
width.  Reading of characters stops either when this maximum is reached
or when a nonmatching charac-
          ter  is  found,  whichever happens first.  Most conversions
discard initial white space characters (the exceptions are noted
below), and these discarded characters
          don't count toward the maximum field width.  String input
conversions store a terminating null byte ('\0') to mark the end of the
input; the  maximum  field  width
          does not include this terminator.

  •      An  optional type modifier character.  For example, the l type
modifier is used with integer conversions such as %d to specify that
the corresponding pointer argu-
          ment refers to a long rather than a pointer to an int.

  •      A conversion specifier that specifies the type of input
conversion to be performed.

  The conversion specifications in format are of two forms, either
beginning with '%' or beginning with "%n$".  The two forms should not
be mixed in the same format string,
  except that a string containing "%n$" specifications can include %%
and %*.  If format contains '%' specifications, then these correspond
in order with successive pointer
  arguments.  In the "%n$" form (which is specified in POSIX.1-2001,
but not C99), n is a decimal integer that specifies that the converted
input should be  placed  in  the
  location referred to by the n-th pointer argument following format.
```

Task 2 memory and segments.

1. Describe the different memory segments in my code.

```
Address: 0x5a3b79218014; Value: 0
Address: 0x7ffd17f5c9ac; Value: 1
Address: 0x7ffd17f5c9b0; Address: 0x5a3ba69012a0; Value: 2
```

By looking at the print of the program 01-mem.bin we can come to this conclution:

`var1` is a global variable, in code it is allocated before the main function, and the adress is `0x5...` which is a low adress, so it is in the data segment.

`var2` is a local variable, it is allocated on the stack when the main function is called, and the address is `0x7...` which is a higher address, so it is in the stack segment.

`var3` is a pointer to a dynamically allocated memory on the heap, the pointer itself is stored on the stack (address `0x7...`), but it points to a memory location on the heap (address `0x5...`), so it is in the heap segment.

2. Describe the purpose of a process' address space, starting with high adresses to low adresses.

Each segment has its own purpose:

- **Stack segment:**, in this case `0x7...`, is used for local variables and function call management. With dynamic memory allocation.
- **Heap segment:**, in this case `0x5...`, is used for dynamic memory allocation outside of the stack, and remains allocated until it is explicitly freed. (e.g., using `malloc` and `free` in C). Example of this in the 01-mem.c code is the line `var3 = malloc(sizeof(int));`. For good practice, we should always free the memory we have allocated to the heap, bu using `free(var3);` when we are done using it.
- **Data segment:**, in this case `0x5...`, is used for global and static variables. This segment is divided into initialized and uninitialized sections. Initialized global and static variables are stored in the initialized data segment, while uninitialized ones are stored in the uninitialized data segment (BSS).
- **Text segment:**, not shown in the example, is used for the actual code of the program. This segment is typically read-only to prevent accidental modification of instructions.
- **0x0 - NULL pointer:**, the address `0x0` is reserved to represent a null pointer, which indicates that a pointer does not point to any valid memory location.

3. Differences between static, global and local variables.

Global variables:

- Declared outside of functions. *(including main)*

- Accessible from any function within the same file.
- Stored in the data segment of memory.
- **Can be accessed from other files using the `extern` keyword.**

Static variables:

- Declared with the `static` keyword.
- Can be declared inside or outside of functions.
- Functions like global variables, but their scope is limited to the file if declared outside a function, or to the function if declared inside a function.
- Also stored in the data segment of memory.
- **Cannot be accessed from other files.**
- Declared outside a function it acts much like a global variable, if we exclude `extern`.

Local variables:

- Declared inside a function.
- Only accessible within the function they are declared in.
- Stored on the stack.
- **Cannot be accessed from other functions or files.**
- Gets destroyed when the function exits.
- *Another way of describing it, is that local variables exist only while the function is executing, similar to how variables in `main` exist only during the program's execution. They are created when the function starts and destroyed when it ends. They are isolated and treat the function's stack frame as their own environment.*

Task 3: Program code:

1. Determine the sizes of text, data, bss.

```
   text    data     bss     dec     hex filename
   1799     616       8    2423     977 ./01-mem.bin
```

- **Text segment:** 1799 bytes
- **Data segment:** 616 bytes
- **BSS segment:** 8 bytes

2. Find the adress.

```
01-mem.bin:     file format elf64-x86-64
architecture: i386:x86-64, flags 0x00000150:
HAS_SYMS, DYNAMIC, D_PAGED
start address 0x00000000000010a0
```

Start address: `0x00000000000010a0`

3. Dissassemble compiled program & Capture output and find name of the function at the start address.

```
01-mem.bin:     file format elf64-x86-64


Disassembly of section .init:

0000000000001000 <_init>:
    1000:       f3 0f 1e fa             endbr64
    1004:       48 83 ec 08             sub    $0x8,%rsp
    1008:       48 8b 05 d9 2f 00 00    mov    0x2fd9(%rip),%rax        # 3fe
    100f:       48 85 c0                test   %rax,%rax
    1012:       74 02                   je     1016 <_init+0x16>
    1014:       ff d0                   call   *%rax
    1016:       48 83 c4 08             add    $0x8,%rsp
    101a:       c3                      ret

Disassembly of section .plt:

0000000000001020 <.plt>:
    1020:       ff 35 8a 2f 00 00       push   0x2f8a(%rip)        # 3fb0 <_G
    1026:       ff 25 8c 2f 00 00       jmp    *0x2f8c(%rip)        # 3fb8 <_
    102c:       0f 1f 40 00             nopl   0x0(%rax)
    1030:       f3 0f 1e fa             endbr64
    1034:       68 00 00 00 00          push   $0x0
    1039:       e9 e2 ff ff ff          jmp    1020 <_init+0x20>
    103e:       66 90                   xchg   %ax,%ax
    1040:       f3 0f 1e fa             endbr64
    1044:       68 01 00 00 00          push   $0x1
    1049:       e9 d2 ff ff ff          jmp    1020 <_init+0x20>
    104e:       66 90                   xchg   %ax,%ax
    1050:       f3 0f 1e fa             endbr64
    1054:       68 02 00 00 00          push   $0x2
    1059:       e9 c2 ff ff ff          jmp    1020 <_init+0x20>
    105e:       66 90                   xchg   %ax,%ax
```

```
Disassembly of section .plt.got:

0000000000001060 <__cxa_finalize@plt>:
    1060:       f3 0f 1e fa             endbr64
    1064:       ff 25 8e 2f 00 00       jmp    *0x2f8e(%rip)        # 3ff8 <_
    106a:       66 0f 1f 44 00 00       nopw   0x0(%rax,%rax,1)

Disassembly of section .plt.sec:

0000000000001070 <__stack_chk_fail@plt>:
    1070:       f3 0f 1e fa             endbr64
    1074:       ff 25 46 2f 00 00       jmp    *0x2f46(%rip)        # 3fc0 <_
    107a:       66 0f 1f 44 00 00       nopw   0x0(%rax,%rax,1)

0000000000001080 <printf@plt>:
    1080:       f3 0f 1e fa             endbr64
    1084:       ff 25 3e 2f 00 00       jmp    *0x2f3e(%rip)        # 3fc8 <p
    108a:       66 0f 1f 44 00 00       nopw   0x0(%rax,%rax,1)

0000000000001090 <malloc@plt>:
    1090:       f3 0f 1e fa             endbr64
    1094:       ff 25 36 2f 00 00       jmp    *0x2f36(%rip)        # 3fd0 <m
    109a:       66 0f 1f 44 00 00       nopw   0x0(%rax,%rax,1)

Disassembly of section .text:

00000000000010a0 <_start>:
    10a0:       f3 0f 1e fa             endbr64
    10a4:       31 ed                   xor    %ebp,%ebp
    10a6:       49 89 d1                mov    %rdx,%r9
    10a9:       5e                      pop    %rsi
    10aa:       48 89 e2                mov    %rsp,%rdx
    10ad:       48 83 e4 f0             and    $0xfffffffffffffff0,%rsp
    10b1:       50                      push   %rax
    10b2:       54                      push   %rsp
    10b3:       45 31 c0                xor    %r8d,%r8d
    10b6:       31 c9                   xor    %ecx,%ecx
    10b8:       48 8d 3d ca 00 00 00    lea    0xca(%rip),%rdi        # 1189
    10bf:       ff 15 13 2f 00 00       call   *0x2f13(%rip)        # 3fd8 <_
    10c5:       f4                      hlt
    10c6:       66 2e 0f 1f 84 00 00    cs nopw 0x0(%rax,%rax,1)
    10cd:       00 00 00

00000000000010d0 <deregister_tm_clones>:
    10d0:       48 8d 3d 39 2f 00 00    lea    0x2f39(%rip),%rdi        # 401
    10d7:       48 8d 05 32 2f 00 00    lea    0x2f32(%rip),%rax        # 401
    10de:       48 39 f8                cmp    %rdi,%rax
    10e1:       74 15                   je     10f8 <deregister_tm_clones+0x2
    10e3:       48 8b 05 f6 2e 00 00    mov    0x2ef6(%rip),%rax        # 3fe
    10ea:	48 85 c0             	test   %rax,%rax
    10ed:	74 09                	je     10f8 <deregister_tm_clones+0x2
    10ef:	ff e0                	jmp    *%rax
    10f1:	0f 1f 80 00 00 00 00 	nopl   0x0(%rax)
    10f8:	c3                   	ret
    10f9:	0f 1f 80 00 00 00 00 	nopl   0x0(%rax)

0000000000001100 <register_tm_clones>:
    1100:	48 8d 3d 09 2f 00 00 	lea    0x2f09(%rip),%rdi        # 401
    1107:	48 8d 35 02 2f 00 00 	lea    0x2f02(%rip),%rsi        # 401
    110e:	48 29 fe             	sub    %rdi,%rsi
    1111:	48 89 f0             	mov    %rsi,%rax
    1114:	48 c1 ee 3f          	shr    $0x3f,%rsi
    1118:	48 c1 f8 03          	sar    $0x3,%rax
    111c:	48 01 c6             	add    %rax,%rsi
    111f:	48 d1 fe             	sar    $1,%rsi
    1122:	74 14                	je     1138 <register_tm_clones+0x38>
    1124:	48 8b 05 c5 2e 00 00 	mov    0x2ec5(%rip),%rax        # 3ff
    112b:	48 85 c0             	test   %rax,%rax
    112e:	74 08                	je     1138 <register_tm_clones+0x38>
    1130:	ff e0                	jmp    *%rax
    1132:	66 0f 1f 44 00 00    	nopw   0x0(%rax,%rax,1)
    1138:	c3                   	ret
    1139:	0f 1f 80 00 00 00 00 	nopl   0x0(%rax)

0000000000001140 <__do_global_dtors_aux>:
    1140:	f3 0f 1e fa          	endbr64
    1144:	80 3d c5 2e 00 00 00 	cmpb   $0x0,0x2ec5(%rip)        # 401
    114b:	75 2b                	jne    1178 <__do_global_dtors_aux+0x
    114d:	55                   	push   %rbp
    114e:	48 83 3d a2 2e 00 00 	cmpq   $0x0,0x2ea2(%rip)        # 3ff
    1155:	00 
    1156:	48 89 e5             	mov    %rsp,%rbp
    1159:	74 0c                	je     1167 <__do_global_dtors_aux+0x
    115b:	48 8b 3d a6 2e 00 00 	mov    0x2ea6(%rip),%rdi        # 400
    1162:	e8 f9 fe ff ff       	call   1060 <__cxa_finalize@plt>
    1167:	e8 64 ff ff ff       	call   10d0 <deregister_tm_clones>
    116c:	c6 05 9d 2e 00 00 01 	movb   $0x1,0x2e9d(%rip)        # 401
    1173:	5d                   	pop    %rbp
    1174:	c3                   	ret
    1175:	0f 1f 00             	nopl   (%rax)
    1178:	c3                   	ret
    1179:	0f 1f 80 00 00 00 00 	nopl   0x0(%rax)

0000000000001180 <frame_dummy>:
    1180:	f3 0f 1e fa          	endbr64
    1184:	e9 77 ff ff ff       	jmp    1100 <register_tm_clones>

0000000000001189 <main>:
    1189:       f3 0f 1e fa             endbr64
    118d:       55                      push   %rbp
    118e:       48 89 e5                mov    %rsp,%rbp
    1191:       48 83 ec 20             sub    $0x20,%rsp
    1195:       64 48 8b 04 25 28 00    mov    %fs:0x28,%rax
    119c:       00 00
    119e:       48 89 45 f8             mov    %rax,-0x8(%rbp)
    11a2:       31 c0                   xor    %eax,%eax
    11a4:       c7 45 ec 01 00 00 00    movl   $0x1,-0x14(%rbp)
    11ab:       bf 04 00 00 00          mov    $0x4,%edi
    11b0:       e8 db fe ff ff          call   1090 <malloc@plt>
    11b5:       48 89 45 f0             mov    %rax,-0x10(%rbp)
    11b9:       48 8b 45 f0             mov    -0x10(%rbp),%rax
    11bd:       c7 00 02 00 00 00       movl   $0x2,(%rax)
    11c3:       8b 05 4b 2e 00 00       mov    0x2e4b(%rip),%eax        # 401
    11c9:       89 c2                   mov    %eax,%edx
    11cb:       48 8d 05 42 2e 00 00    lea    0x2e42(%rip),%rax        # 401
    11d2:       48 89 c6                mov    %rax,%rsi
    11d5:       48 8d 05 2c 0e 00 00    lea    0xe2c(%rip),%rax        # 2008
    11dc:       48 89 c7                mov    %rax,%rdi
    11df:       b8 00 00 00 00          mov    $0x0,%eax
    11e4:       e8 97 fe ff ff          call   1080 <printf@plt>
    11e9:       8b 55 ec                mov    -0x14(%rbp),%edx
    11ec:       48 8d 45 ec             lea    -0x14(%rbp),%rax
    11f0:       48 89 c6                mov    %rax,%rsi
    11f3:       48 8d 05 0e 0e 00 00    lea    0xe0e(%rip),%rax        # 2008
    11fa:       48 89 c7                mov    %rax,%rdi
    11fd:       b8 00 00 00 00          mov    $0x0,%eax
    1202:       e8 79 fe ff ff          call   1080 <printf@plt>
    1207:       48 8b 45 f0             mov    -0x10(%rbp),%rax
    120b:       8b 08                   mov    (%rax),%ecx
    120d:       48 8b 55 f0             mov    -0x10(%rbp),%rdx
    1211:       48 8d 45 f0             lea    -0x10(%rbp),%rax
    1215:       48 89 c6                mov    %rax,%rsi
    1218:       48 8d 05 01 0e 00 00    lea    0xe01(%rip),%rax        # 2020
    121f:       48 89 c7                mov    %rax,%rdi
    1222:       b8 00 00 00 00          mov    $0x0,%eax
    1227:       e8 54 fe ff ff          call   1080 <printf@plt>
    122c:       90                      nop
    122d:       48 8b 45 f8             mov    -0x8(%rbp),%rax
    1231:       64 48 2b 04 25 28 00    sub    %fs:0x28,%rax
    1238:       00 00
    123a:       74 05                   je     1241 <main+0xb8>
    123c:       e8 2f fe ff ff          call   1070 <__stack_chk_fail@plt>
    1241:       c9                      leave
    1242:       c3                      ret

Disassembly of section .fini:
```

```
0000000000001244 <_fini>:
    1244:       f3 0f 1e fa             endbr64
    1248:       48 83 ec 08             sub    $0x8,%rsp
    124c:       48 83 c4 08             add    $0x8,%rsp
    1250:       c3                      ret
```

Function at start address: `_start`

The start function executes **before** the main function, and set up the environment for the program to run. Such as setting up the stack, heap and other components. It also calls the main function, and when the main function returns, it handles the program's exit.

4. Run the program several times and see the address change.

The reason the address changes each time is because the address is randomized for each execution. This is simply a security feature, to make it harder for a bad actor to predict where certain parts of the program will be stored. Also known as `ASLR` *(Address Space Layout Randomization)*.

Task 4: The stack.

1. Compile the code with gcc.

Disclaimer: *Considering im using the newest version of ubuntu as operating system, some changes was made to the code to make it compile without errors. Using `@ %p"` instead of `@ 0x%08x` i do belive it should print the same result in this case.*

2. Determine default size of stack for your Linux system.

```
$ ulimit -s
8192
```

Default stack size: 8192 KB (8 MB)

3. Results from running the code:**

```
...
func() frame address @ 0x7ffded2277d0
func() frame address @ 0x7ffded2277b0
func() frame address @ 0x7ffded227790
Segmentation fault (core dumped)
```

This fault is simply the program **exceeding the stack limit**, by calling the function recursively without a base case to stop it.

4. What does grep func | wc -l *catch*?

This line `grep func` grabs lines containing the word *func*, and `wc -l` is a command that counts the number of lines.

So in this case we got:

```
261712
```

meaning that the function `func` was called 261712 times before the stack overflowed and caused a segmentation fault.

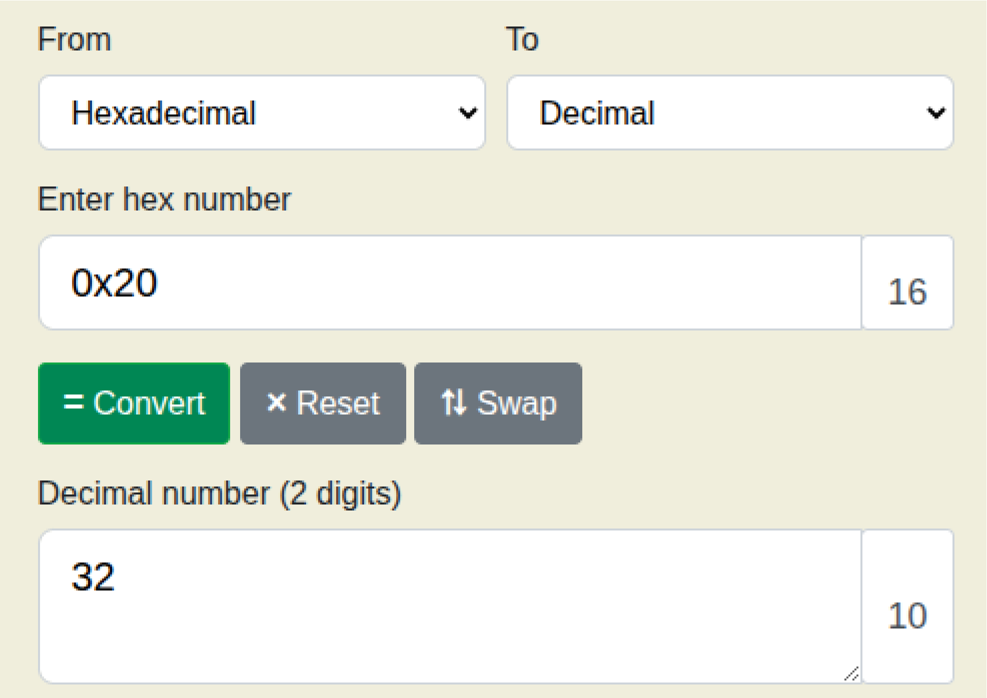
5. How much stack memory *bytes* does each recursive call use?

To calculate bytes of a stack we can simply take the difference between two consecutive frame addresses. It is importent to note that in stack we start from high adress to low adress. So we must subtract the lower adress from the higher adress. Which is uncommin in a mathematical context.

```
func() frame address @ 0x7ffe0a822920
func() frame address @ 0x7ffe0a822900
```

if we take `$7ffe0a822920 - $7ffe0a822900 = 0x20`

So each recursive call uses `0x20` bytes of memory, which is `32` bytes in decimal. This can easily be calculated by using a hex to decimal converter. In this case I used rapidtables



From: Hexadecimal

To: Decimal

Enter hex number: 0x20 16

Decimal number (2 digits): 32 10