Assignment 10

(1) **(1 pt)** Explain how this version of **getenv** achieve **thread-safe** according to the code above?

Answer: The base version of getenv is not thread safe, the reason being that no matter which thread calls getenv, it will store its data in the same static buffer, shared by all threads.

This version is thread safe due to the following reasons:

a) It initializes thread-specific data and associates getenv data with a single thread using `pthread_key_create`, and uses `pthread_once` to ensure initialization is done once per thread.
b) after locking the mutex for environment access to prevent other threads from accessing the `key` variable, we use `pthread_getspecific` to get the value for the key in the current thread.
c) The crucial step, if the environment buffer is not `NULL`, we designate it as thread specific data using `pthread_setspecific`, associating it with the intialized key.
d) After this we simply use the thread specific `envbuf` to store the contents of the environment variable we are searching for, if it is found, and return it. We unlock the mutex before returning to make sure other threads can use the function.
e) Otherwise, we simply unlock the mutex and return `NULL`, because the environment variable is not found.

(2) **(1 pt)** Is it possible to make this getenv function **async-signal safety** by just temporarily blocking signals at the beginning of the function and then restoring the previous signal mask before the function returns? Explain.

Answer: If all signals are blocked at the beginning of the function, it should be possible to provide async-signal safety to the function since we are disabling all incoming signals for the whole function, and `pthread_sigmask` will atomically make sure that the mask change will work on all threads accurately. The only possible way this method will not be successful is if the signal mask is changed inside the function after blocking all signals in the beginning of thef unction.

(3) **(2 pt)** Please run assignment10.c on FreeBSD and see if it can run successfully. If not, try to explain what happened. You can use gdb to help

you find out why and where it crashes. (Github:https://github.com/Jia-WeiFang/Advanced-UNIX-Programming_Student. git)

Answer: The given code is using the thread-safe `getenv` from question 1. Running it with `time` provides the following info:

```
$ time ./a.out &
(2 minutes pass..)
$ time: command terminated abnormally
      125.55 real         1.90 user         8.06 sys

[1]   Segmentation fault      time ./a.out
```

The program runs for 2 minutes and terminates abnormally.

The first discovery after trying comments on the source code is that commenting line 31 un-hangs the code:

```
envbuf = malloc(MAXSTRINGSZ);
```

this indicates a potential problem stemming from what malloc performs in its body.

Using gdb to start through the code we get an immediate problem:

```
(gdb) b main
Breakpoint 1 at 0x400eb8: file assignment10.c, line 55.
(gdb) run
Starting program: /home/raghu/imato/assignment10/a.out

Program received signal SIGSEGV, Segmentation fault.
Address not mapped to object.
memset () at /usr/src/contrib/cortex-strings/src/aarch64/
memset.S:136
136     /usr/src/contrib/cortex-strings/src/aarch64/memset.S: No
such file or directory.
(gdb)
```

the issue seems to happen *before* the main function. D The problem here may be that malloc itself is calling thread unsafe code that attempts to access data outside the thread, and fails. This may be causing the segmentation fault that we are seeing. inspecting the crash in gdb:

```
#11 0x00000000405e9da8 in malloc_conf_init
(sc_data=0xffffc0000b88, bin_shard_sizes=0xffffc0000af8) at /
usr/obj/usr/src/arm64.aarch64/lib/libc/jemalloc_jemalloc.c:1449
#12 malloc_init_hard_a0_locked () at /usr/obj/usr/src/
```

```
arm64.aarch64/lib/libc/jemalloc_jemalloc.c:1509
#13 0x00000000405ebf5c in malloc_init_hard () at /usr/obj/usr/
src/arm64.aarch64/lib/libc/jemalloc_jemalloc.c:1754
#14 0x00000000405e33b0 in malloc_init () at /usr/obj/usr/src/
arm64.aarch64/lib/libc/jemalloc_jemalloc.c:227
#15 imalloc_init_check (sopts=<optimized out>, dopts=<optimized
out>) at /usr/obj/usr/src/arm64.aarch64/lib/libc/
jemalloc_jemalloc.c:2233
#16 imalloc (sopts=<optimized out>, dopts=<optimized out>) at /
usr/obj/usr/src/arm64.aarch64/lib/libc/jemalloc_jemalloc.c:2264
#17 __je_malloc_default (size=4096) at /usr/obj/usr/src/
arm64.aarch64/lib/libc/jemalloc_jemalloc.c:2293
#18 0x0000000000400d60 in getenv (name=0x404c27fc "MALLOC_CONF")
at assignment10.c:30
#19 0x00000000405ea1f8 in jemalloc_secure_getenv
(name=<optimized out>) at /usr/obj/usr/src/arm64.aarch64/lib/
libc/jemalloc_jemalloc.c:725
#20 obtain_malloc_conf (which_source=3, buf=0xffffc00045bc "")
at /usr/obj/usr/src/arm64.aarch64/lib/libc/jemalloc_jemalloc.c:
1007
#21 malloc_conf_init_helper (sc_data=0x0, bin_shard_sizes=0x0,
initial_call=<optimized out>,
opts_cache=opts_cache@entry=0xffffc00049c0, buf=<optimized out>,
buf@entry=0xffffc00045bc "")
  at /usr/obj/usr/src/arm64.aarch64/lib/libc/
jemalloc_jemalloc.c:1042
#22 0x00000000405e9da8 in malloc_conf_init
(sc_data=0xffffc0002c08, bin_shard_sizes=0xffffc0002b78) at /
usr/obj/usr/src/arm64.aarch64/lib/libc/jemalloc_jemalloc.c:1449
#23 malloc_init_hard_a0_locked () at /usr/obj/usr/src/
arm64.aarch64/lib/libc/jemalloc_jemalloc.c:1509
#24 0x00000000405ebf5c in malloc_init_hard () at /usr/obj/usr/
src/arm64.aarch64/lib/libc/jemalloc_jemalloc.c:1754
#25 0x00000000405e33b0 in malloc_init () at /usr/obj/usr/src/
arm64.aarch64/lib/libc/jemalloc_jemalloc.c:227
#26 imalloc_init_check (sopts=<optimized out>, dopts=<optimized
out>) at /usr/obj/usr/src/arm64.aarch64/lib/libc/
jemalloc_jemalloc.c:2233
#27 imalloc (sopts=<optimized out>, dopts=<optimized out>) at /
usr/obj/usr/src/arm64.aarch64/lib/libc/jemalloc_jemalloc.c:2264
#28 __je_malloc_default (size=4096) at /usr/obj/usr/src/
arm64.aarch64/lib/libc/jemalloc_jemalloc.c:2293
#29 0x0000000000400d60 in getenv (name=0x404c27fc "MALLOC_CONF")
at assignment10.c:30
```

The stack calls indicate that our getenv function is being called recursively inside malloc before `pthread_once` is called, creating an infinite loop. Changing the name of `getenv` to `getenv_m` lets us run the program with correct results.

A potential way to fix this without renaming the function is to allow initialization of thread safe data elsewhere, shifting the code for malloc outside the `getenv` function.