Overcoming the XNU Process Stack Size Limit

Bilin Sun Hanchen Li

January 2023

Outline

Motivation
XNU Introduction

- The kernel of Apple's macOS (formerly branded Mac OS or OS X) is called XNU
- iOS, tvOS and watchOS all inherit the XNU kernel
- Some open source kernel projects are also based on XNU
- XNU was born as a hybrid of Mach and BSD
- The macOS version of XNU has been officially certified as POSIX-compliant since Mac OS X Leopard (10.5)
- The macOS version of XNU has always been open source, and the iOS version has been open source since iOS 12

Motivation
The Problem

- Older versions of `clang` and `llvm` have been ported to iOS
- We tried to bootstrap a newer version of `clang`
- Turned out the compiler runs into stack overflow when compiling the `clang` source!

Listing 1: terminal outputs

```
clang-10: error: unable to execute command: Illegal instruction: 4
clang-10: error: clang frontend command failed due to signal (use -
    v to see invocation)
```

Attempts

`ulimit`

- Needed to increase the stack size for newly created user processes
- `ulimit` is a utility required in the POSIX standard for setting limits on system resources
- `ulimit -Ss` sets a "soft limit" on the process stack size
- `ulimit -Hs` sets a "hard limit" on the process stack size
- The hard limit bounds the soft limit, but setting the hard limit needs root privilege
- Cannot increase hard limit beyond 1 MB on iOS even with root privilege!

Attempts

`setrlimit()`

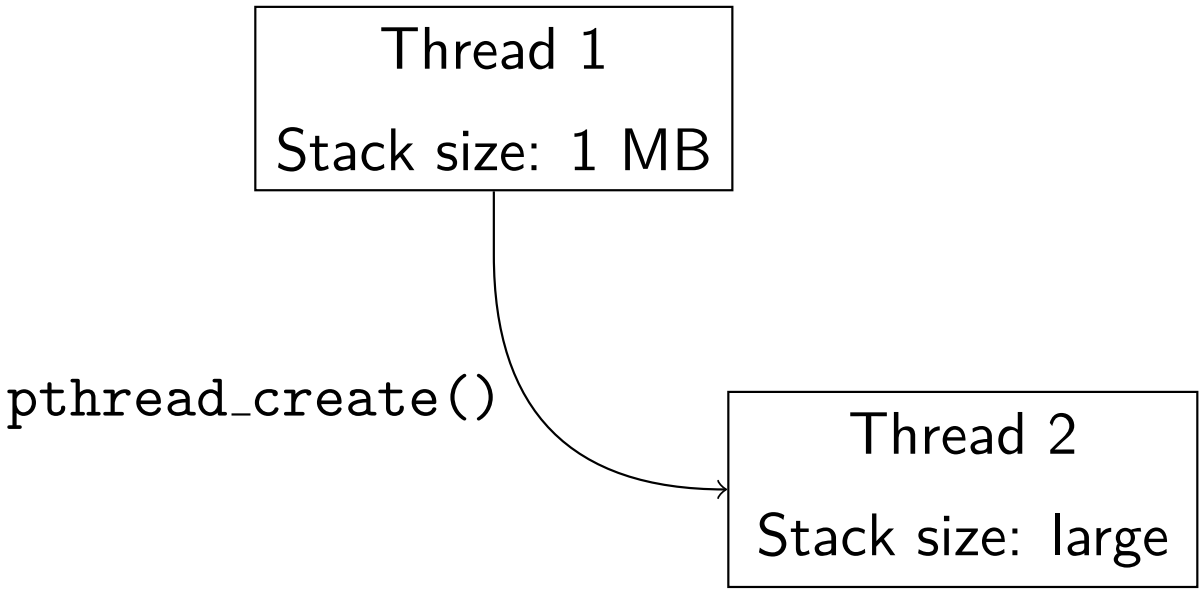
- `setrlimit()` in `sys/resource.h` is also required in the POSIX standard
- Neither does it work
- Digging the XNU source shows that there's a hard-coded limit in the kernel

Listing 2: /bsd/arm/vmparam.h

```
/* XXX stack size limit is a platform property: use getrlimit(2) */
#if (defined(TARGET_OS_OSX) && (TARGET_OS_OSX != 0)) || \
    (defined(KERNEL) && !defined(CONFIG_EMBEDDED) || \
    (CONFIG_EMBEDDED == 0))
#define MAXSSIZ (64*1024*1024)  /* max stack size */
#else
#define MAXSSIZ (1024*1024)     /* max stack size */
#endif /* TARGET_OS_OSX .. || XNU_KERNEL_PRIVATE .. */
```

Attempts
Thread Creation

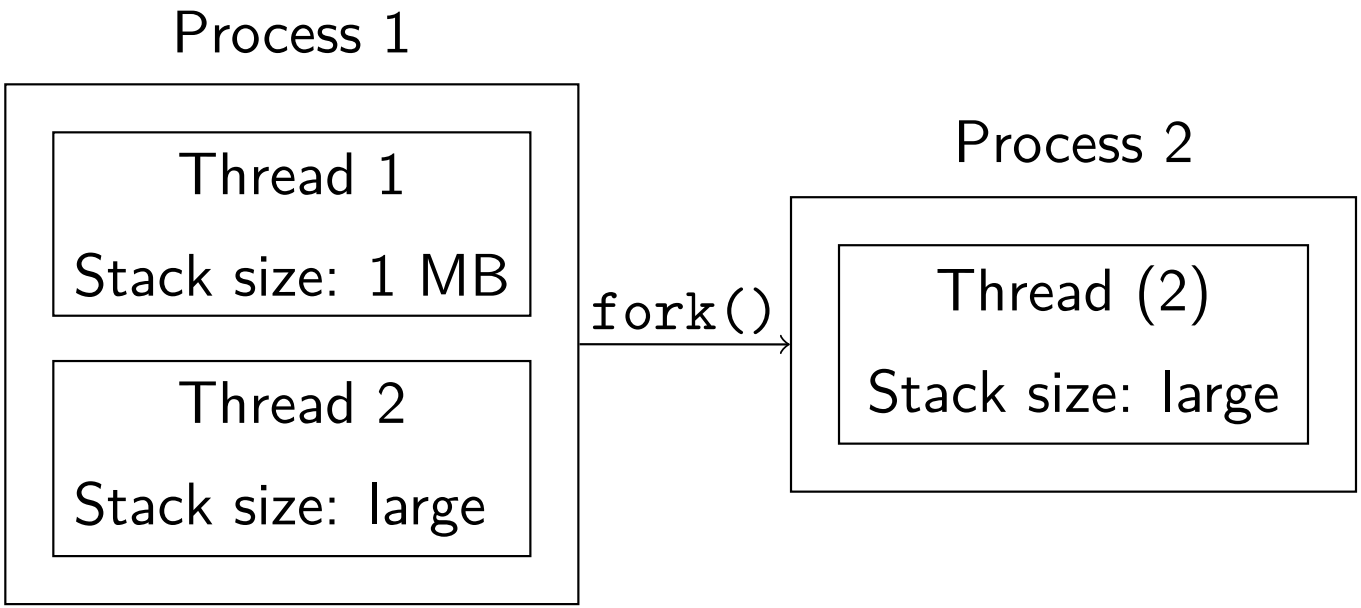
- Surprisingly, `pthread_create()` accepts a large stack size without failing



Attempts

`fork()` and `execv()`

- When run from a progress with more than one thread, `fork()` creates a new process with only one thread, namely the thread calling `fork()`.



- However, `fork()` does nothing to help us jump to the program we want to run
- The `exec` family functions reallocate the stack and so are useless

Resolution
Code Injection

- XNU provides a mechanism for injecting dynamic libraries into binaries being loaded, interacted with by setting the `DYLD_INSERT_LIBRARIES` environment variable
  - In fact, Mobile Substrate, a vital piece of iOS jailbreak infrastructure, relies exactly on this mechanism to enable the hooking of functions in the iOS system or Apps
- We can provide a `__attribute__((constructor))` function in our dynamic library and it will be automatically executed before the `main()` of the injected program
  - `__attribute__((constructor))` is an `gcc` extension to the C language; supported by `clang` as well

Resolution
Code Injection

- We then do the thread creation in this "constructor" and simply terminate the older thread with a smaller stack
- We need to resolve where `main()` (of the injected program) is loaded in the memory and manually jump there

Listing 3: terminal outputs

```
$ ./reportDefaultStackSize
The stack size is 1032192.
$ clang ./stackHack.c -isysroot
    ./iPhoneOS12.4.sdk -g -O0 -
    lpthread -dynamiclib -
    stackHack.c -o stackHack.
    dylib
$ DYLD_INSERT_LIBRARIES="
    $DYLD_INSERT_LIBRARIES:./
    stackHack.dylib"
$ ./reportDefaultStackSize
The stack size is 6565888.
```

Listing 4: stackHack.c

```c
#include<unistd.h>
#include<pthread.h>
#include<stdlib.h>
#include<dlfcn.h>

int mArgc;
char **mArgv;

pthread_t runbeforemain;

static void threadFunc(void)
{
        void *progHandle = dlopen(NULL, RTLD_NOW);
        int (*mainAddr)(int,char**) = dlsym(progHandle,"main");
        printf("%ul\n",(unsigned long)mainAddr);
        exit((*mainAddr)(mArgc,mArgv));
}
```

```
static void spawnThreadWithCustomStackSize(size_t sz)
{
    pthread_attr_t attr;
    if (pthread_attr_init(&attr) != 0)
        perror("pthread_attr_init␣failed");
    int r = pthread_attr_setstacksize(&attr, sz);
    if (r == 0)
    {
        pthread_t thread;
        if (pthread_create(&thread, &attr, threadFunc, &sz) == 0)
        {
            if (pthread_join(thread, NULL) != 0)
                    perror("pthread_join␣failed");
        }
        else perror("pthread_create␣failed");
    }
    else perror("pthread_attr_setstacksize␣failed");
}

__attribute__((constructor))
        static void runBeforeMain(int argc,const char **argv)
{
        mArgc=argc; mArgv=argv;
        runbeforemain=pthread_self();
        spawnThreadWithCustomStackSize(6553600);
}
```

Listing 5: reportDefaultStackSize.c

```c
#include<pthread.h>
#include<stdio.h>
int main(int argv, char **argc)
{
    size_t sz=pthread_get_stacksize_np(pthread_self());
    printf("The stack size is %d.", sz);
    return 0;
}
```

Conclusion

- Worked well for this simple demo program
- Didn't work for more complicated ones like `clang`
  - Perhaps due to certain initialization functions being interrupted

Conclusion
Further Research

- Try to take care of the initialization of more complex programs and make this hack universally applicable
- Alternatively, certain old program linked in specific ways may be edited to request a larger stack size on start
  - May no longer be applicable to newer programs
- The best things to do, however, is to make wise use of dynamically allocated memory (on the heap) and thus avoid writing programs that will require very deep recursions in the first place

End