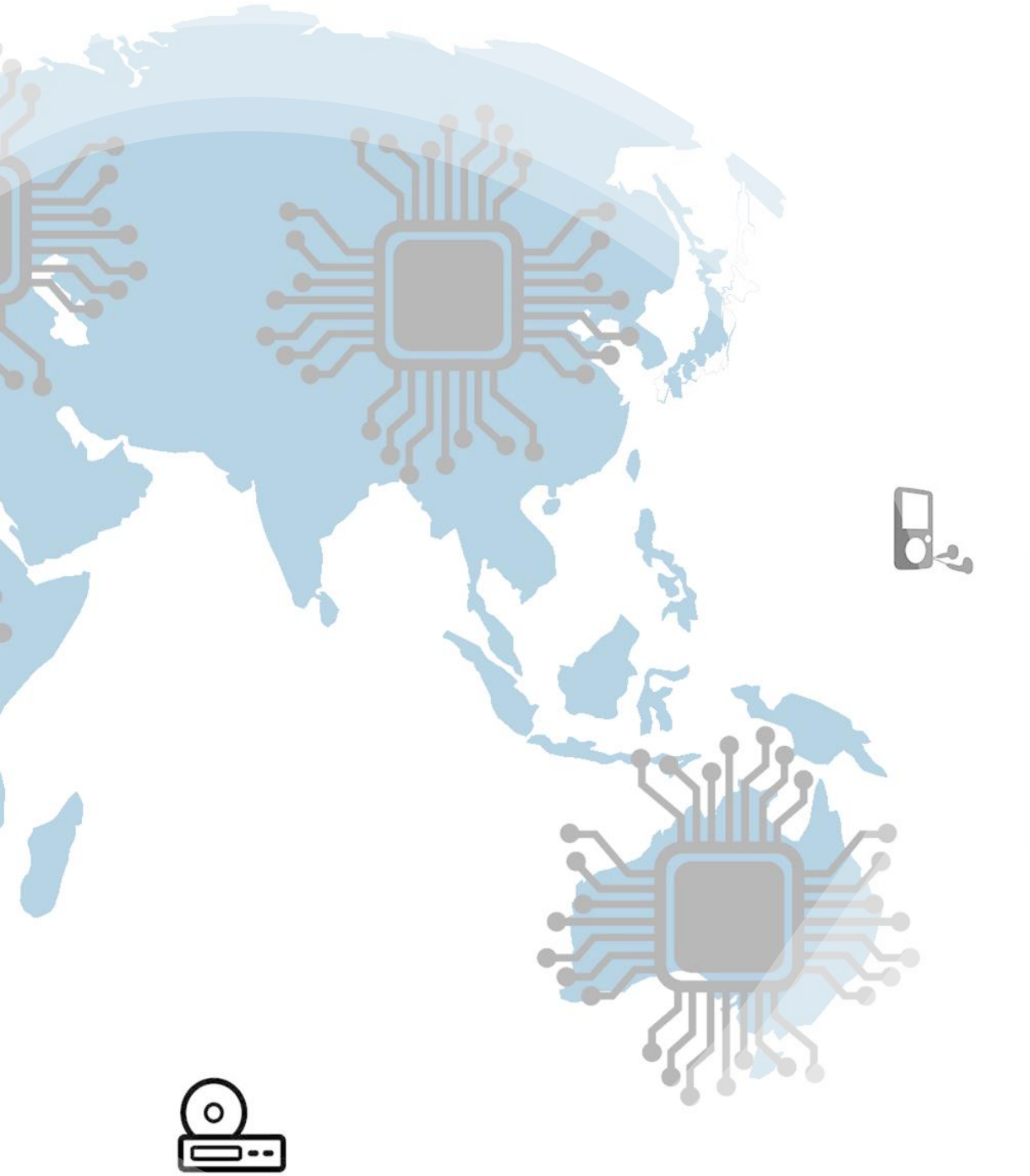


Porting the Rust libstd to NuttX/Cortex-M4F and prototyping a simple web server

Sony Home Entertainment & Sound Products Inc.

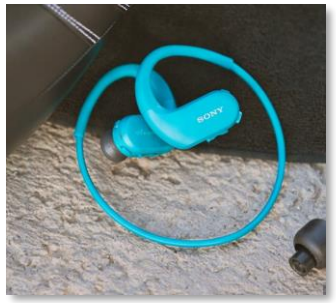
Yoshinori.Sugino@sony.com

- **August 15-16 2020**

NuttX Online Workshop

About me

- Software engineer
  - Digital voice recorder
  - Digital music player WALKMAN



- Rust.Tokyo 2019 speaker
- My mentor: Masayuki Ishikawa
  - NuttX contributor
  - Arm TechCon 2016, Embedded Linux Conference 2017-2019, NuttX 2019 speaker

GitHub profile picture

Yoshinori Sugino

sgysh

Agenda

- What is Rust?
- Objectives
- Using the Rust standard library (libstd) on NuttX
  - Using println! macro
  - Using std::thread
  - Using std::net
  - Using std::fs

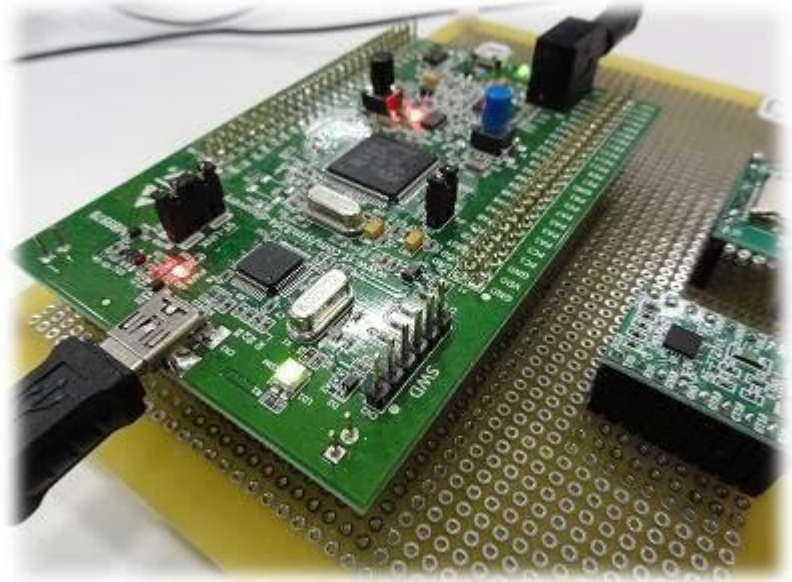
What is Rust?

- Open-source systems programming language
  - Compiled language
  - focuses on speed, memory safety and parallelism
  - Sponsored by Mozilla
- "most loved programming language“
  - in the Stack Overflow Developer Survey for 5 years in a row

https://insights.stackoverflow.com/survey/2020#technology-most-loved-dreaded-and-wanted-languages-loved

Objectives

- Using Rust in embedded systems
  - Not bare metal, but on RTOS
  - Using the Rust standard library (libstd)
    - println!
    - std::vec
    - std::thread
    - std::net
    - std::fs
- Identifying issues for using Rust on NuttX
  - By running examples on NuttX
    - TRPL(The Rust Programming Language book) examples
    - RBE(Rust by Example) examples
  - NuttX + STM32F4Discovery



Approaches

- Link Rust library and built-in application
  - Create .a file from Rust code
  - Link the .a file and existing built-in application
    - Use the hello application this time

- NOTE
  - Chose the way that changes the source code as little as possible
    - Perhaps, it's not the best way
  - This prototyping was done around April 2019

Create custom target

- Create custom target based on thumbv7em-none-eabi

```
% rustc --print target-list | grep thumbv7
thumbv7a-pc-windows-msvc
thumbv7em-none-eabi
thumbv7em-none-eabihf
thumbv7m-none-eabi
thumbv7neon-linux-androideabi
thumbv7neon-unknown-linux-gnueabihf
```

- Thought that there were many changes needed to add a definition for NuttX
- Decided to reuse the settings for Linux which seems most similar to NuttX
  - "os": "linux"
  - "target-family": "unix“

Create built-in application

- Modify the hello application
  - Call Rust function from the hello application
  - Stack size: 8192 bytes
- Create a Rust library that says hello
  - println!(“Hello, world!!”);
  - channel: nightly
  - crate-type = ["staticlib"]
  - Dependencies: libstd

Build errors

- Many undefined reference errors
  - Remove unnecessary sections with a linker option --gc-sections
    - Each function was placed in a separate .text section
  - Change symbol names with objcopy --redefine-sym
    - __errno_location
    - __xpg_strerror_r
  - Implemented by myself
    - posix_memalign
    - pthread_condattr_setclock ← defined this function but did not fully implemented

Runtime problems

- Building succeeded but …
  1. stack overflow occurs
  2. The hello application hung

```
#[no_mangle]
pub fn rust_hello() {
    println!("Hello, world!!");
}
```

- An application that uses write!() works successfully

```
use std::fs::File;
use std::io::Write;
use std::os::unix::io::FromRawFd;

#[no_mangle]
pub fn rust_hello() {
    let mut f = unsafe { File::from_raw_fd(1) };
    write!(&mut f, "Hello, world!\n").unwrap();
}
```

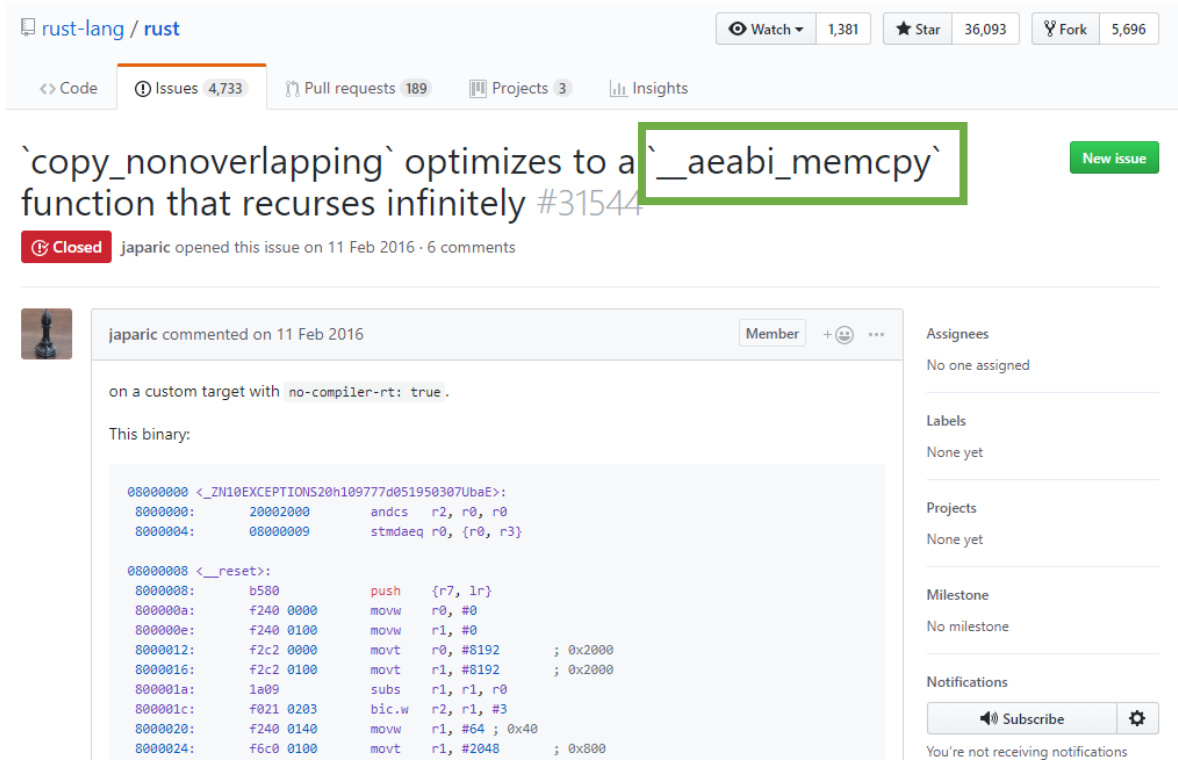
Stack overflow occurs

- Investigation
  - Set a watchpoint at the end of stack
  - Strangely, __aeabi_memcpy4 was called recursively

```
(gdb) bt
#0  0x08017d88 in __aeabi_memcpy4 (
    dest=0x10001c68 "ª\336ª\336ª\336ª\336h\034", src=0x10000a64 "", n=4)
    at /home/ysugino/.cargo/registry/src/github.com-1ecc6299db9ec823/compiler_built
ins-0.1.12/src/arm.rs:143
#1  0x08017e8a in core::intrinsics::copy_nonoverlapping::he1ce10561b48cb87 (
    src=0x10000a64, dst=0x10001c68, count=1)
    at /home/ysugino/my_rust_toolchains/nightly-x86_64-unknown-nuttx/lib/rustlib/sr
c/rust/src/libcore/intrinsics.rs:1422
#2  0x08017ebc in core::ptr::read::h74acc211aa6b5938 (src=0x10000a64)
    at /home/ysugino/my_rust_toolchains/nightly-x86_64-unknown-nuttx/lib/rustlib/sr
c/rust/src/libcore/ptr.rs:575
#3  0x08017dae in __aeabi_memcpy4 (
    dest=0x10001cf8 "ª\336ª\336ª\336ª\336\370\034", src=0x10000a64 "",
    n=4)
    at /home/ysugino/.cargo/registry/src/github.com-1ecc6299db9ec823/compiler_built
ins-0.1.12/src/arm.rs:144
#4  0x08017e8a in core::intrinsics::copy_nonoverlapping::he1ce10561b48cb87 (
    src=0x10000a64, dst=0x10001cf8, count=1)
    at /home/ysugino/my_rust_toolchains/nightly-x86_64-unknown-nuttx/lib/rustlib/sr
c/rust/src/libcore/intrinsics.rs:1422
#5  0x08017ebc in core::ptr::read::h74acc211aa6b5938 (src=0x10000a64)
    at /home/ysugino/my_rust_toolchains/nightly-x86_64-unknown-nuttx/lib/rustlib/sr
c/rust/src/libcore/ptr.rs:575
#6  0x08017dae in __aeabi_memcpy4 (
    dest=0x10001d88 "ª\336ª\336ª\336ª,\035", src=0x10000a64 "", n=4)
    at /home/ysugino/.cargo/registry/src/github.com-1ecc6299db9ec823/compiler_built
ins-0.1.12/src/arm.rs:144
```

Stack overflow occurs

- Investigation
  - I found the following issue on GitHub
  - According to the report, it seems to be a linker mis-optimization



rust-lang / rust

Watch 1,381 | Star 36,093 | Fork 5,696

Code | Issues 4,733 | Pull requests 189 | Projects 3 | Insights

`copy_nonoverlapping` optimizes to a `__aeabi_memcpy` function that recurses infinitely #31544

New issue

Closed japaric opened this issue on 11 Feb 2016 · 6 comments

japaric commented on 11 Feb 2016 Member

on a custom target with `no-compiler-rt: true`.

This binary:

```
08000000 <_ZN10EXCEPTIONS20h109777d051950307UbaE>:
 8000000:       20002000        andcs   r2, r0, r0
 8000004:       08000009        stmdaeq r0, {r0, r3}

08000008 <__reset>:
 8000008:       b580            push    {r7, lr}
 800000a:       f240 0000       movw    r0, #0
 800000e:       f240 0100       movw    r1, #0
 8000012:       f2c2 0000       movt    r0, #8192       ; 0x2000
 8000016:       f2c2 0100       movt    r1, #8192       ; 0x2000
 800001a:       1a09            subs    r1, r1, r0
 800001c:       f021 0203       bic.w   r2, r1, #3
 8000020:       f240 0140       movw    r1, #64 ; 0x40
 8000024:       f6c0 0100       movt    r1, #2048       ; 0x800
```

Assignees: No one assigned

Labels: None yet

Projects: None yet

Milestone: No milestone

Notifications: Subscribe

You're not receiving notifications

Stack overflow occurs

- Workaround

```
#![no_builtins]

#[no_mangle]
pub fn rust_hello() {
    println!("Hello, world!!");
}

#[no_mangle]
pub unsafe extern "aapcs" fn __aeabi_memcpy4(dest: *mut u8, src: *const u8, size: usize) {
    __aeabi_memcpy(dest as *mut u8, src as *const u8, size);
}

#[no_mangle]
pub unsafe extern "aapcs" fn __aeabi_memcpy(dest: *mut u8, src: *const u8, size: usize) {
    let mut i = 0;
    while i < size {
        *dest.offset(i as isize) = *src.offset(i as isize);
        i += 1;
    }
}
```

The hello application hung

- Investigation
  - The hello application hung on a semaphore

```
NuttShell (NSH)
nsh> hello &
hello [3:100]
nsh> ps
  PID PRI POLICY   TYPE    NPX STATE    EVENT     SIGMASK   STACK   USED  FILLED COMMAND
    0   0 FIFO     Kthread N-- Ready              00000000 000000 000000   0.0%  Idle Task
    2 100 FIFO     Task    --- Running            00000000 002028 001984  97.8%! init
    3 100 RR       Task    --- Waiting  Semaphore 00000000 008172 001136  13.9%  hello
nsh>
```

  - Hung in pthread_mutex_lock()

The hello application hung

- Investigation
  - Confirm correct behaviors on Linux
    1. libstd on NuttX should work in the same way as on Linux because the same libstd is used on NuttX
    2. Set breakpoints at pthread_mutex_init() and pthread_mutex_lock()
    3. I found that pthread_mutex_init() was not called before pthread_mutex_lock() was called

The hello application hung

- Investigation
  - It turned out that PTHREAD_MUTEX_INITIALIZER is used instead of pthread_mutex_init()

```
#[allow(dead_code)] // sys isn't exported yet
impl Mutex {
    pub const fn new() -> Mutex {
        // Might be moved to a different address, so it is better to avoid
        // initialization of potentially opaque OS data before it landed.
        // Be very careful using this newly constructed `Mutex`, reentrant
        // locking is undefined behavior until `init` is called!
        Mutex { inner: UnsafeCell::new(libc::PTHREAD_MUTEX_INITIALIZER) }
    }
    #[inline]
    pub unsafe fn init(&mut self) {
```

The hello application hung

- Investigation
  - NuttX and Rust have different PTHREAD_MUTEX_INITIALIZER data structures

```
#if defined(CONFIG_PTHREAD_MUTEX_TYPES) && !defined(CONFIG_PTHREAD_MUTEX_UNSAFE)
#  define PTHREAD_MUTEX_INITIALIZER {NULL, SEM_INITIALIZER(1), -1, \
                                     __PTHREAD_MUTEX_DEFAULT_FLAGS, \
                                     PTHREAD_MUTEX_DEFAULT, 0}
#elif defined(CONFIG_PTHREAD_MUTEX_TYPES)
#  define PTHREAD_MUTEX_INITIALIZER {SEM_INITIALIZER(1), -1, \
                                     PTHREAD_MUTEX_DEFAULT, 0}
#elif !defined(CONFIG_PTHREAD_MUTEX_UNSAFE)
#  define PTHREAD_MUTEX_INITIALIZER {NULL, SEM_INITIALIZER(1), -1,\
                                     __PTHREAD_MUTEX_DEFAULT_FLAGS}
#else
#  define PTHREAD_MUTEX_INITIALIZER {SEM_INITIALIZER(1), -1}
#endif
```

NuttX
include/pthread.h

```
align_const! {
    pub const PTHREAD_MUTEX_INITIALIZER: pthread_mutex_t = pthread_mutex_t {
        size: [0; __SIZEOF_PTHREAD_MUTEX_T],
    };
    pub const PTHREAD_COND_INITIALIZER: pthread_cond_t = pthread_cond_t {
        size: [0; __SIZEOF_PTHREAD_COND_T],
    };
    pub const PTHREAD_RWLOCK_INITIALIZER: pthread_rwlock_t = pthread_rwlock_t {
        size: [0; __SIZEOF_PTHREAD_RWLOCK_T],
    };
}
```

Rust bindings to libc
src/unix/notbsd/linux/mod.rs

The hello application hung

- Solution
  - Modify the same value as defined in NuttX
    - Modify __SIZEOF_PTHREAD_MUTEX_T to 12
    - Modify PTHREAD_MUTEX_INITIALIZER to [0, 0, 0, 0, 1, 0, 0xff, 0xff, 1, 0, 0, 0]

```
% git diff
diff --git a/src/unix/notbsd/linux/mod.rs b/src/unix/notbsd/linux/mod.rs
index 3d9ccad..9701ce1 100644
--- a/src/unix/notbsd/linux/mod.rs
+++ b/src/unix/notbsd/linux/mod.rs
@@ -1066,7 +1066,7 @@ pub const TCP_MD5SIG: ::c_int = 14;

 align_const! {
     pub const PTHREAD_MUTEX_INITIALIZER: pthread_mutex_t = pthread_mutex_t {
-        size: [0; __SIZEOF_PTHREAD_MUTEX_T],
+        size: [0, 0, 0, 0, 1, 0, 0xff, 0xff, 1, 0, 0, 0],
     };
     pub const PTHREAD_COND_INITIALIZER: pthread_cond_t = pthread_cond_t {
         size: [0; __SIZEOF_PTHREAD_COND_T],
diff --git a/src/unix/notbsd/linux/other/b32/mod.rs b/src/unix/notbsd/linux/other/b32/mod.rs
index d078f75..2d77de2 100644
--- a/src/unix/notbsd/linux/other/b32/mod.rs
+++ b/src/unix/notbsd/linux/other/b32/mod.rs
@@ -300,7 +300,7 @@ pub const EPOLL_CLOEXEC: ::c_int = 0x80000;
 pub const EFD_CLOEXEC: ::c_int = 0x80000;

 pub const __SIZEOF_PTHREAD_CONDATTR_T: usize = 4;
-pub const __SIZEOF_PTHREAD_MUTEX_T: usize = 24;
+pub const __SIZEOF_PTHREAD_MUTEX_T: usize = 12;
 pub const __SIZEOF_PTHREAD_RWLOCK_T: usize = 32;
 pub const __SIZEOF_PTHREAD_MUTEXATTR_T: usize = 4;
 pub const __SIZEOF_PTHREAD_RWLOCKATTR_T: usize = 8;
@@ -310,24 +310,21 @@ align_const! {
     pub const PTHREAD_RECURSIVE_MUTEX_INITIALIZER_NP: ::pthread_mutex_t =
         pthread_mutex_t {
             size: [
-                0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0,
-                0, 0, 0,
+                0, 0, 0, 0, 1, 0, 0xff, 0xff, 1, 0, 0 ,0
             ],
         };
     #[cfg(target_endian = "little")]
     pub const PTHREAD_ERRORCHECK_MUTEX_INITIALIZER_NP: ::pthread_mutex_t =
         pthread_mutex_t {
             size: [
-                0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 2, 0, 0, 0, 0, 0, 0, 0,
-                0, 0, 0,
+                0, 0, 0, 0, 1, 0, 0xff, 0xff, 1, 0, 0 ,0
             ],
         };
     #[cfg(target_endian = "little")]
     pub const PTHREAD_ADAPTIVE_MUTEX_INITIALIZER_NP: ::pthread_mutex_t =
         pthread_mutex_t {
             size: [
-                0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 3, 0, 0, 0, 0, 0, 0, 0,
-                0, 0, 0,
+                0, 0, 0, 0, 1, 0, 0xff, 0xff, 1, 0, 0 ,0
             ],
         };
     #[cfg(target_endian = "big")]
```

The hello application run with no errors

- Finally it worked

```
NuttShell (NSH)
nsh> hello
Hello, world!!
nsh>
```

- But memory leak occurs

```
NuttShell (NSH)
nsh> free
              total       used       free    largest
Umem:        192608       7568     185040     124880
nsh> hello
Hello, world!!
nsh> free
              total       used       free    largest
Umem:        192608       8832     183776     124880
nsh> hello
Hello, world!!
nsh> free
              total       used       free    largest
Umem:        192608       8928     183680     124880
nsh>
```

Comparison with write!()

- Memory leak does not occur by the application using write!()

```
NuttShell (NSH)
nsh> free
              total       used       free    largest
Umem:        192640       7568     185072     124912
nsh> hello
Hello, world!
nsh> free
              total       used       free    largest
Umem:        192640       7568     185072     124912
nsh>
```

Memory leaks using println! macro

- Investigation
  1. Set breakpoints at malloc and free
  2. Cannot find free for thread_local! macro
  3. Memory leak occurred in the simple application using thread_local! macro
  4. It turned out that pthread_key_create in libc of NuttX ignores **the destructor argument**
- Solution
  - Support the destructor pthread_key_create
- Result
  - Memory leak at first execution decreased slightly
    - 1264 bytes leak → 1200 bytes leak
  - The amount of memory leak at second execution does not change
    - 96 bytes leak

Memory leaks using println! macro

- Investigation
  1. Notice that pthread_key_create is not called and there are uninitialized variables at second execution
  2. It turned out that "global variables are initialized only once when the system powers up†".

     †https://cwiki.apache.org/confluence/display/NUTTX/Linux+Processes+vs+NuttX+Tasks
- Solution
  - Insert the start code for the built-in application
    - .data, .bss, .ctors, .init_array, .dtors, .fini_array, …
- Result
  - Memory leaks 1200 bytes each time
  - But no memory leaks for a simple application that uses thread-local variables

Memory leaks using println! macro

- Investigation
  1. Found 1024 bytes malloc for struct Lazy
  2. It turned out that **cleanup function for struct Lazy was not called**
- Solution
  - Call the cleanup function just before end of process
- Result
  - Memory leaks do not occur by using println!(“Hello, world!”)

std::thread

- Undefined reference errors occur when std::thread is used

- sigaltstack
- munmap
- pthread_self
- pthread_getattr_np
- pthread_attr_getguardsize
- dlsym

Undefined reference errors occur when std::thread is used

- Investigation
  - Read the source code of the Rust standard library (libstd)
  - Undefined reference symbols are found in functions for stack overflow detection
- Solution
  - Remove the functions because it takes a lot of time to implement
- Result
  - Link without any errors
  - But runtime error occurs when thread is created

Runtime error occurs when thread is created

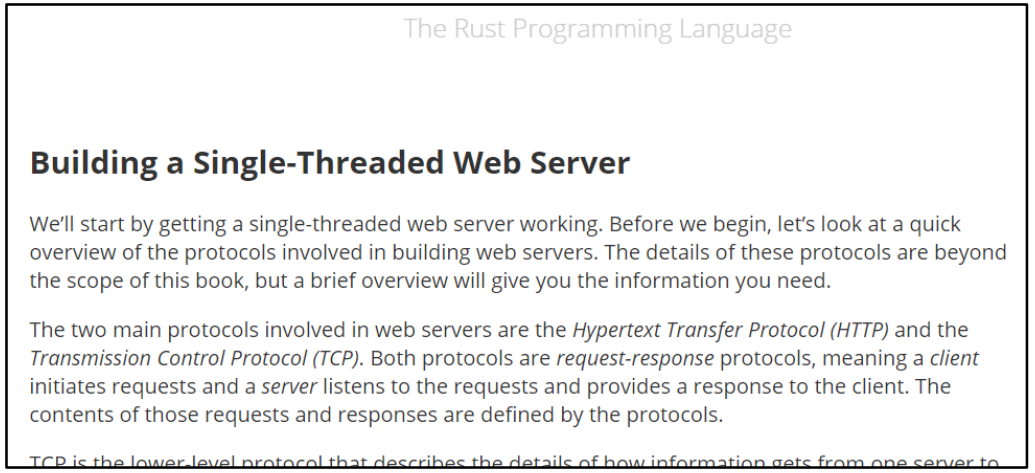
- Investigation
  - Return ENOMEM in the memory allocation for stack
- Solution
  - Modify stack size from 2MiB to 4KiB
- Result
  - Thread and channel examples in RBE worked
  - But memory leak occurs → under investigation

```
NuttShell (NSH)
nsh> hello
this is thread number 0
this is thread number 1
this is thread number 2
this is thread number 3
this is thread number 4
this is thread number 5
this is thread number 6
this is thread number 7
this is thread number 8
this is thread number 9
nsh>
```

```
NuttShell (NSH)
nsh> hello
thread 0 finished
thread 1 finished
thread 2 finished
[Ok(0), Ok(1), Ok(2)]
nsh>
```

std::net

- Try to run a simple single thread web server written in TRPL
  1. socket()
  2. bind()
  3. listen()
  4. accept()
  5. read()/write()
  6. close()


The Rust Programming Language

Building a Single-Threaded Web Server

We'll start by getting a single-threaded web server working. Before we begin, let's look at a quick overview of the protocols involved in building web servers. The details of these protocols are beyond the scope of this book, but a brief overview will give you the information you need.

The two main protocols involved in web servers are the *Hypertext Transfer Protocol (HTTP)* and the *Transmission Control Protocol (TCP)*. Both protocols are *request-response* protocols, meaning a *client* initiates requests and a *server* listens to the requests and provides a response to the client. The contents of those requests and responses are defined by the protocols.

TCP is the lower-level protocol that describes the details of how information gets from one server to

https://doc.rust-lang.org/book/ch20-01-single-threaded.html

- Change to RNDIS configuration in order to use USB Ethernet
- Remove the accept4() that caused undefined reference error
  - Use accept() instead

Linking succeeded but runtime error occurs

- Investigation
  - Error occurs in std::net::TcpListener::bind()
  - **SOCK_CLOEXEC** is used (Linux-specific, **NuttX does not support**)
- Solution
  - Return EINVAL when unsupported types(SOCK_*) is used
- Result
  - Error still occurs in std::net::TcpListener::bind()

std::net::TcpListener::bind() caused runtime errors

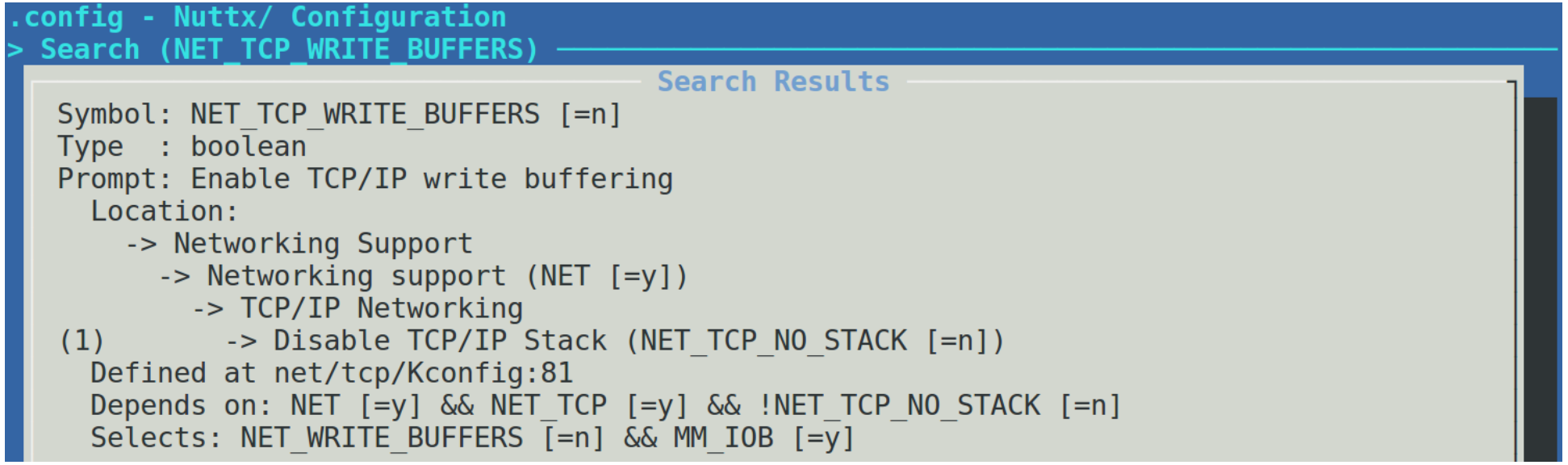
- Investigation
  - **FIOCLEX** is used, but **NuttX does not support**
- Workaround
  - Ignore FIOCLEX
  - Tried to use fcntl with F_SETFD and FD_CLOEXEC instead, but F_SETFD is not implemented
- Result
  - Error still occurs in std::net::TcpListener::bind()

std::net::TcpListener::bind() caused runtime errors

- Investigation
  - Some constants such as SOL_SOCKET and SO_REUSEADDR have different value between NuttX and Rust
- Solution
  - Change to the same value as defined in NuttX
- Result
  - std::net::TcpListener::bind() succeeded
  - can read requests when wget runs
  - But a response with HTML does not reach host PC

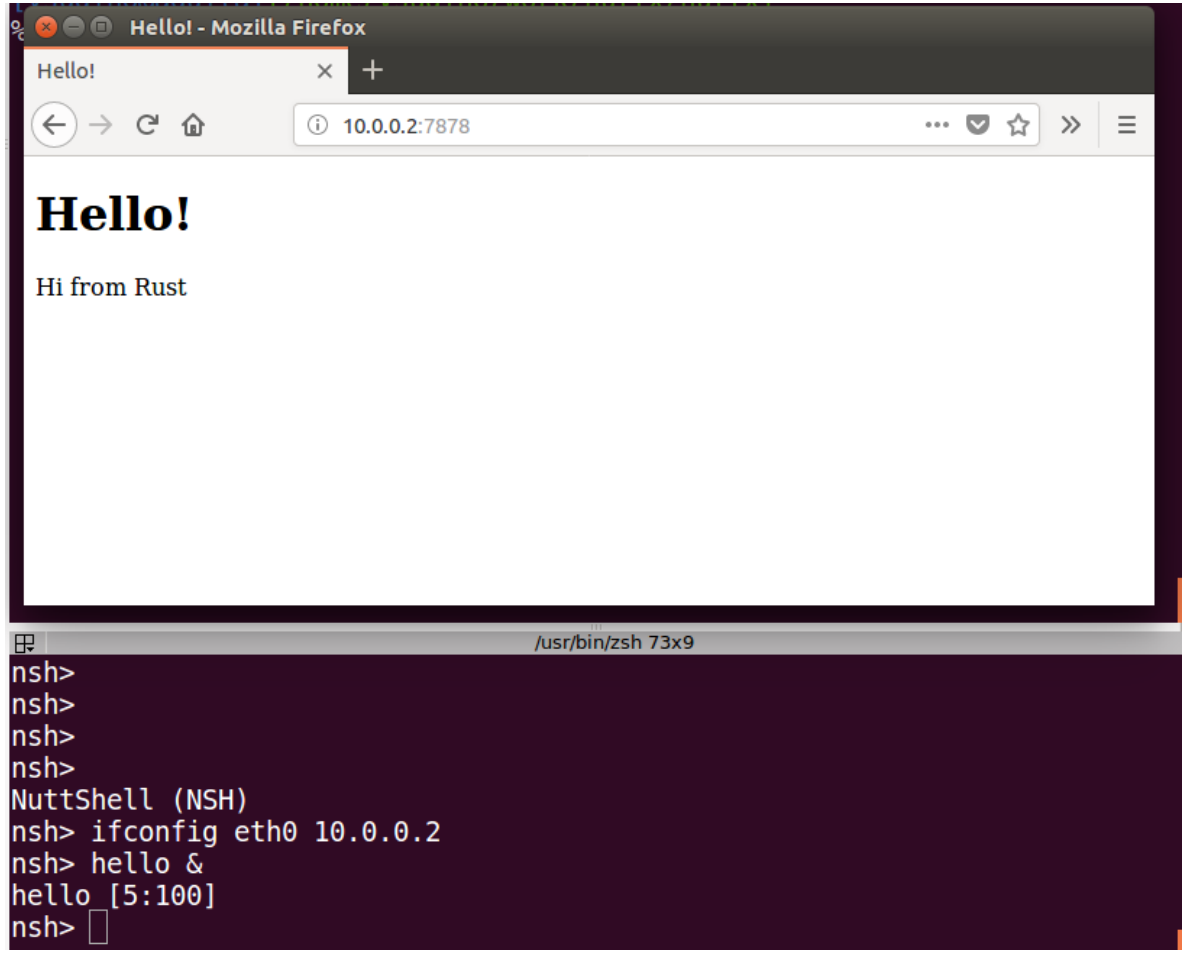
A response with HTML does not reach the host PC

- Workaround
  - Disable CONFIG_NET_TCP_WRITE_BUFFERS of NuttX
- Result
  - 200 OK
  - Response was received but wget did not exit



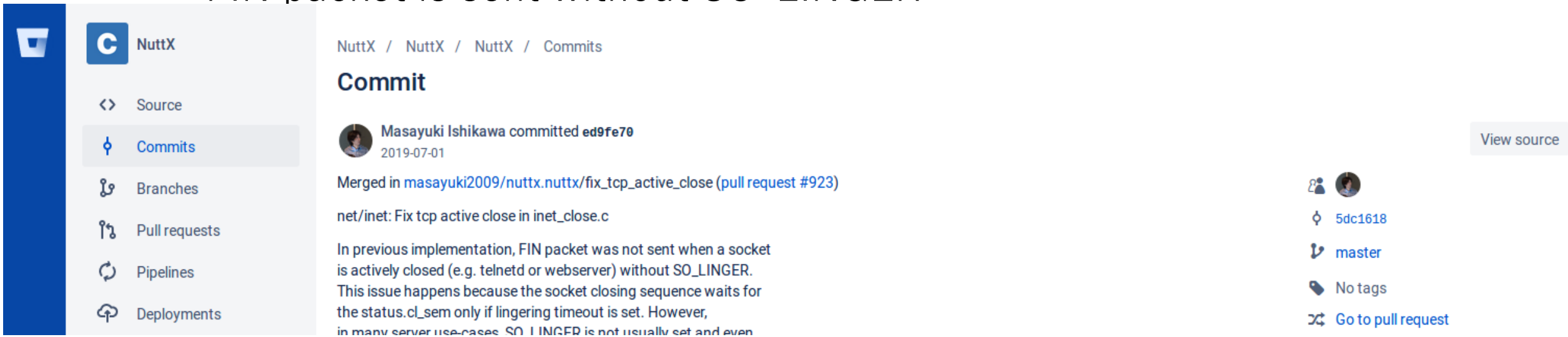
Response was received but wget did not exit

- Investigation
  - Response was received but wget did not exit
  - **FIN packet is not sent** when a socket is closed
- Workaround
  - Add ‘Connection: close’ and ‘Content-Length’ to response header
  - (Or SO_LINGER is set)
- Result
  - wget exits successfully
  - Firefox shows successfully



The network stack bugs were fixed

- Fixed by the following commit
  - A response reaches host PC if CONFIG_NET_TCP_WRITE_BUFFERS is enabled
  - FIN packet is sent without SO_LINGER



https://bitbucket.org/nuttx/nuttx/commits/ed9fe700242909851b6ef4049aa8fea13fa67699

std::fs

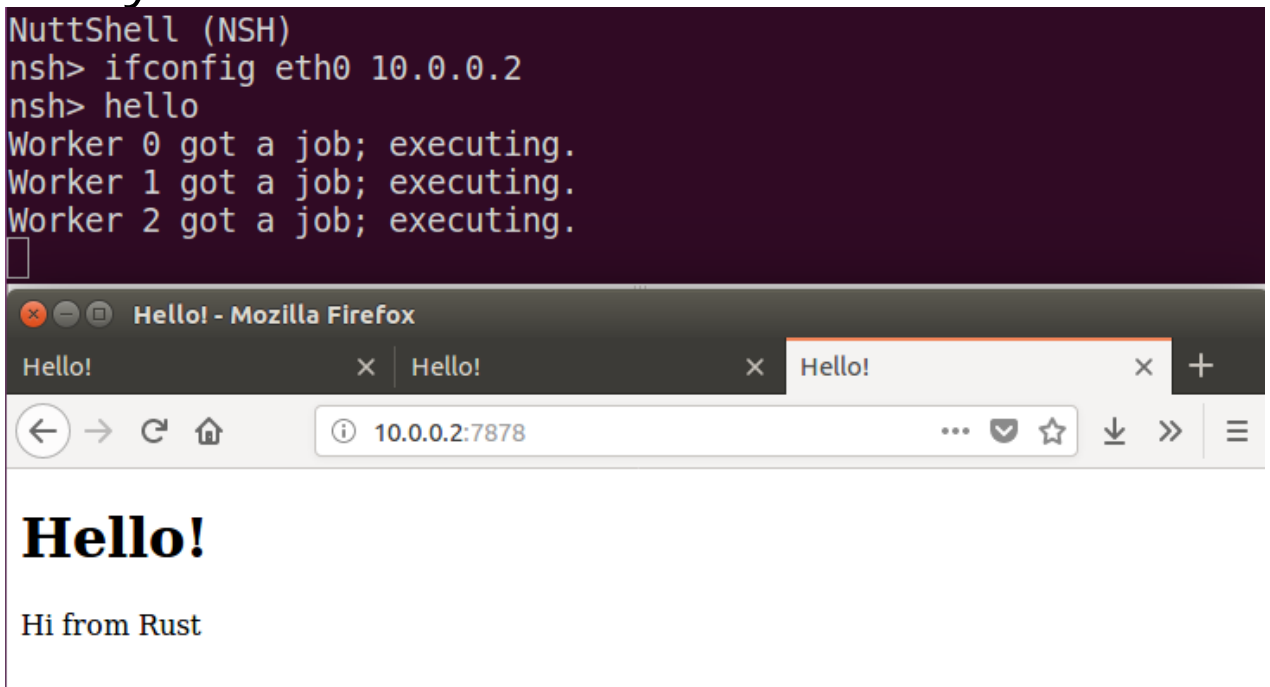
- Try to use std::fs
  - Read a file and show its contents

- Error and solution
  - Undefined reference errors occur
    - Modify open64 and fstat64 to open and fstat
  - Remove a function using F_SETFD
    - that is not supported on NuttX

- Result
  - Can read a file without any memory leaks

Run a multithreaded web server

- Try to run a multithreaded web server based on TRPL implementation
  - Reading from romfs ("/rom/hello.html")
- Run without any errors and Firefox shows successfully



Issues and future work

- Issues about using the Rust standard library on NuttX
  - Different constants and different signatures
    - It cannot be detected at link time
  - Memory leaks
  - Unimplemented features on NuttX
  - Network stack bugs
- Future work
  - Investigate memory leak when std::thread is used


NuttX Online
Workshop
Thank you!
THE
APACHE
SOFTWARE FOUNDATION