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
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

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-qnueabihf
```

• 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

• An application that uses write!() works successfully
Stack overflow occurs

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

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

- Workaround

```rust
#![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   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()`

```rust
// [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

```c
#include<pthread.h>

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]
The hello application run with no errors

- Finally it worked

- But memory leak occurs
Comparison with write!()

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

• 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
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!