

Lab2 Report

All the source codes are available at <https://github.com/skyelves/Theseus>.

Part1

Overall:

The docs is easy to read and elaborates Rust detailedly. With the toy example in the docs, we can easily get into use of Rust.

When I am reading the docs, the most impressing thing is the safety which Rust guarantees. In C/C++, even the most proficient programmer can make some seemingly stupid problems, such as overflow and out-of-bounds accesses. However, in Rust, it prevents these problems in the compiling time and reduces the possibility of error in the runtime. Also Rust enforces the programmers to implicitly consider the memory allocations, objects copy and lifetime in an elegant way. With more efforts in the compiling time, the runtime can be much safer and the overhead can be much cheaper.

Another thing, which is also impressing, is Cargo, Rust's build system and package manager. Unlike C/C++ programs, we always suffer from compiling error due to the different APIs in different versions of dependencies. We can never build and install a massive open-source project at the first try. However, with the dependencies specified, it's much easier to transport the program from one platform to another platform or from one server to another server.

In a nutshell, It's very interesting to learn Rust and it's certainly promising to build network stack, database or even OS in Rust.

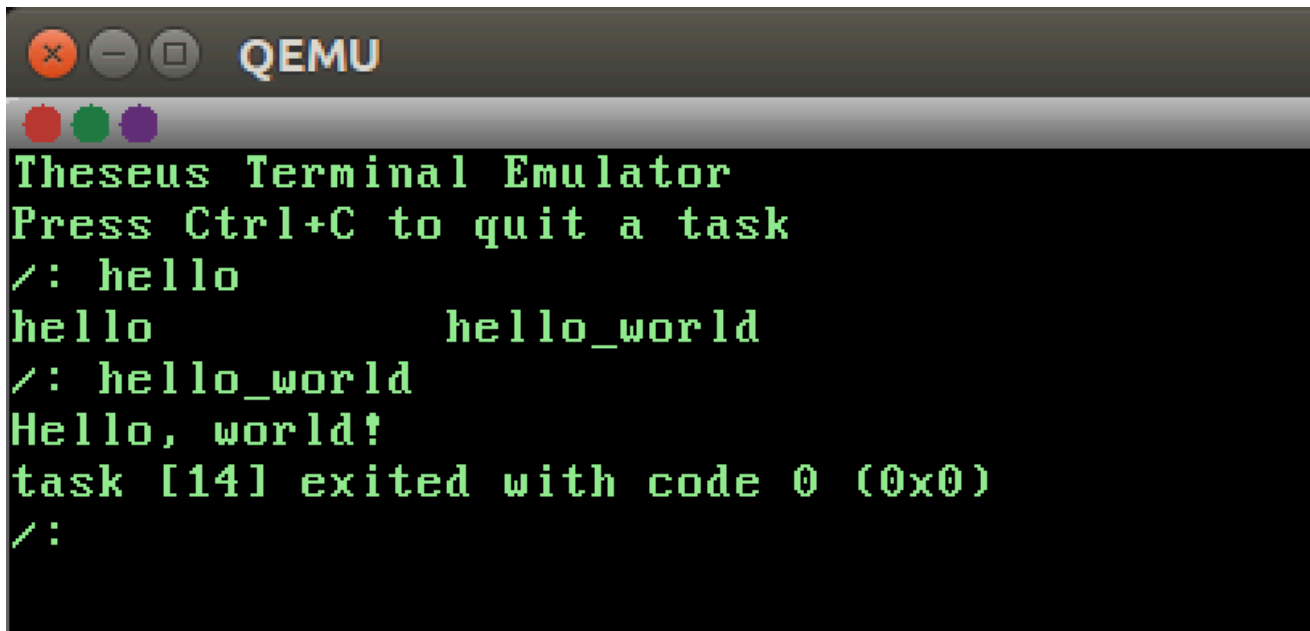
Part2

Overall:

patch the `part2.patch` and then run the following commands:

```
make run
# enter Theseus
$ hello_world # in Theseus
```

The results are as follows:



```
QEMU
Theseus Terminal Emulator
Press Ctrl+C to quit a task
/: hello
hello          hello_world
/: hello_world
Hello, world!
task [14] exited with code 0 (0x0)
/:
```

This part is rather simple since it's only a hello world program on Theseus. However, there are still some challenges when finishing this part.

The first challenge is to make and build the Theseus. It 's always onerous and error-prone to build systems written in C/C++. The dependencies versions or something are always making troubles. However, it's rather easy to make and build the Theseus system written in Rust. Actually, I succeeded in building the Theseus at my first try, which should be unbelievable for building systems written in C/C++.

The second challenge is to write the hello world program. It is a little bit confusing at the first sight reading the assignment: write a "Hello, World!" app for Theseus. Should I write a `helloworld` app on Theseus and then compile it using Rust on Theseus just like what we do in my Mac or something? Then it turns out we should just write a `helloworld` in the source codes of the Theseus and compile it when compile the system. We are not required to compile the `helloworld` when the Theseus is running.

One interesting thing is that the incremental compilation is well developed in Rust. After the first compilation, it takes only several seconds to rebuild the Theseus to test my modification. In lab1, it usually takes me about 5 minutes to rebuild the linux operating system. Though it may relate to the overall codebase amount of the two systems, but I think it should not matter much since I only changed the very limited files in the source codes.

Detail:

The source codes is shown in the part2.patch, our `helloworld` depends on the `terminal_print`. After incorporating this crate, we can use `println!()` to print `helloworld` string.

```

diff -Naur ../Theseus/applications/hello_world/Cargo.toml
lab2/Theseus/applications/hello_world/Cargo.toml
--- ../Theseus/applications/hello_world/Cargo.toml 1969-12-31 16:00:00.000000000
-0800
+++ lab2/Theseus/applications/hello_world/Cargo.toml 2021-11-15
19:03:09.148664752 -0800
@@ -0,0 +1,20 @@
+[package]
+name = "hello_world"
+version = "0.1.0"
+edition = "2018"
+authors = ["Ke Wang <ke.wang.kw754@yale.edu>"]
+build = "../../build.rs"
+
+## See more keys and their definitions at https://doc.rust-
lang.org/cargo/reference/manifest.html
+
+[dependencies]
+
+[dependencies.log]
+version = "0.4.8"
+
+
+[dependencies.terminal_print]
+path = "../../kernel/terminal_print"
+
+## [dependencies.application_main_fn]
+## path = "../../compiler_plugins"
\ No newline at end of file
diff -Naur ../Theseus/applications/hello_world/src/lib.rs
lab2/Theseus/applications/hello_world/src/lib.rs
--- ../Theseus/applications/hello_world/src/lib.rs 1969-12-31 16:00:00.000000000
-0800
+++ lab2/Theseus/applications/hello_world/src/lib.rs 2021-11-15
19:03:09.152664872 -0800
@@ -0,0 +1,12 @@
+#![no_std]
+// #![feature(plugin)]
+// #![plugin(application_main_fn)]
+
+// #[macro_use] extern crate log;
+#[macro_use] extern crate terminal_print;
+
+pub fn main() -> isize {
+    println!("Hello, world!");
+
+

```

```
+    0
+}
\ No newline at end of file
```

Part3

Overall:

patch the `part3.patch` and then run the following commands:

```
make run
# enter Theseus
$ memuse_test # in Theseus
```

The results are as follows:

```
/: memuse_test
tid:2, mem_type:Heap
memuse result: Ok(TaskRef(Task { name: "idle_task_core_1", id: 2, running_on: None, runstate: Runnable, pinned: None }))
tid:2, mem_type:CallStack
memuse result: Ok(TaskRef(Task { name: "idle_task_core_1", id: 2, running_on: None, runstate: Runnable, pinned: None }))
tid:100, mem_type:CallStack
memuse result: Err(TaskNotExist)
task [15] exited with code 0 (0x0)
/: █
```

Describe your experience and any interesting observations. Contrast the system call interface in Linux with the interface in Theseus. Explain why Theseus can export OS functions like that. Why can Theseus run applications in the same privilege level as the kernel? Why doesn't Theseus need a kernel stack?

It's interesting that in Theseus, there is almost no difference between developing kernel functions and developing application functions. It's even more convenient to develop kernel functions since we are allowed to use unsafe statements in kernel functions, though I don't use that in this part. While in Linux, it's much more onerous to build some kernel functions since we are not allowed to use userspace functions, which we are more familiar with, and we should be very careful with the error-prone parallelism, locking and unlocking explicitly, to prevent from things like use-after-free.

The major reason why Theseus can export OS functions like this stems from Rust's ownership design. In Rust, one process can only access the memory space it owns so there is no way an application process can access or modify the memory space of a kernel process. Since the different privileges are aimed to protect the kernel space from the user space. Since in Theseus, the application process can never access the kernel space, there is no need to have an extra privilege which could only add some overhead of privileges changing. Since the kernel no longer serves in a privilege level, there is no need to have a kernel stack.

The most difficult part of this part is to find out the exact function to determine whether the task exists or not. At first, I found `pub const TASKS_DIRECTORY_PATH: &str = "/tasks"`, which suggests all the tasks are listed under the `/tasks` directory. It works but it's not so elegant. Finally I found `get_task` encapsulating whether a task exists or not.

Another challenge is to learning how to refer to the function defined in some other crates and how to implement an enum struct in Rust since I am a true green hand of Rust.

Detail:

To implement memuse function, first we create a crate in `kernel/` directory.

```
cd kernel
cargo new --lib memuse
cd memuse
```

We have a directory `src` and a `Cargo.toml` under the `kernel/memuse` directory.

In `Cargo.toml`, we add three lines under `[package]`.

```
authors = ["Ke Wang <ke.wang.kw754@yale.edu>"]
description = "offers kernel crates the ability to get the physical memory usage of a certain process"
build = "../..../build.rs"
```

Our memuse function depends on the `task` crate, so we add the dependencies at the end of the `Cargo.toml`.

```
[dependencies.log]
version = "0.4.8"

[dependencies.task]
path = "../task"
```

Then we come to the major file of this part, `kernel/memuse/src/ib.rs`.

We first add the dependencies.

```
#[macro_use] extern crate log;
use task;
```

Then we define some helping structures such as `MemType`, `MemuseError` and the return type of the memuse function, `MemuseRes`.

```

#[derive(Debug)]
pub enum MemuseError {
    InvalidMemtype,
    TaskNotExist,
}

#[derive(Debug)]
pub enum MemType {
    Heap,
    CallStack,
}

pub type MemuseRes = Result<task::TaskRef, MemuseError>;

```

We implement the `mymemuse` function in the following way. We first check if the `mem_type` is valid or not. Then we check if the task exists or not. If the task doesn't exist, we return `Err(TaskNotExist)`. Otherwise, we return the `TaskRef` struct of this task.

```

pub fn mymemuse(tid: usize, mem_type: MemType) -> MemuseRes {
    // print to the kernel log
    info!("tid: {:?}, mem_type: {:?}", tid, mem_type);
    // check if the memory type is valid
    match mem_type {
        MemType::Heap => (),
        MemType::CallStack => (),
        _ => return Err(MemuseError::InvalidMemtype),
    }
    // check if the task exists
    let task_ref = task::get_task(tid).ok_or(MemuseError::TaskNotExist);

    task_ref
}

```

To test if our `mymemuse` works or not, we create a crate under `kernel/applications` named `memuse_test`.

```

cd kernel/applications
cargo new --lib memuse_test

```

We update the `[package]` info and add some dependencies. In our `memuse_test` program, we depends on `terminal_print` crate and `memuse` crate.

```

[package]
name = "memuse_test"
version = "0.1.0"
edition = "2018"
authors = ["Ke Wang <ke.wang.kw754@yale.edu>"]
build = "../../build.rs"

# See more keys and their definitions at https://doc.rust-
lang.org/cargo/reference/manifest.html

[dependencies]

[dependencies.log]
version = "0.4.8"

[dependencies.terminal_print]
path = "../../kernel/terminal_print"

[dependencies.memuse]
path = "../../kernel/memuse"

```

Then we add some tests in `memuse_test/src/lib.rs`. We first add some dependencies.

```

#[macro_use] extern crate terminal_print;
use memuse::mymemuse;

```

Then we set the `tid` and `memtype`, print them in the console and call the `mymemuse` function.

```

let tid = 2;
let mem_type = memuse::MemType::Heap;
println!("tid:{:?}, mem_type:{?}", tid, &mem_type);
let res = mymemuse(tid, mem_type);
println!("memuse result: {:?}", res);

```

Then we try some more tests, one for `Callstack` memory type and one for not existing tasks.

```
let tid = 2;
let mem_type = memuse::MemType::CallStack;
println!("tid:{:?}, mem_type:{:?}" , tid, &mem_type);
let res = mymemuse(tid, mem_type);
println!("memuse result: {:?}", res);

let tid = 100;
let mem_type = memuse::MemType::CallStack;
println!("tid:{:?}, mem_type:{:?}" , tid, &mem_type);
let res = mymemuse(tid, mem_type);
println!("memuse result: {:?}", res);
```

Now we are almost there! Test if that works. The results are shown at the beginning of this part.

Part4

Overall:

patch the `part4.patch` and then run the following commands:

```
make run
# enter Theseus
$ memuse_test # in Theseus
```

The results are as follows:


```
/: memuse_test
tid:14, mem_type:Heap
memuse result: Ok(90)
memuse result: Ok(170)
tid:14, mem_type:Heap
memuse result: Ok(528)
tid:14, mem_type:Heap
memuse result: Ok(768)
tid:14, mem_type:CallStack
memuse result: Ok(65536)
tid:14, mem_type:CallStack
memuse result: Ok(65536)
task [14] exited with code 0 (0x0)
```

Describe your experience and any interesting observations. Explain the pros and cons of per-process virtual address space (Linux) vs. one virtual address space for all (Theseus). Explain why Theseus could safely use a single address space for all. Explain the benefits of having tasks sharing the same heaps. Explain why Theseus implements one heap for each core.

One of the major differences between Theseus and linux is that Theseus uses a single global address space. Because the ownership in the Rust ensures the isolation between processes, one process can not access the memory not owned by it. One pages can be shared by multiple processes. For example, process A first allocates 8bytes. At this time, in Theseus process A only consumes 8 bytes rather than 4KB in linux. Then task B allocates 8bytes. in Theseus, process B also only consumes 8 bytes rather than another 4KB in linux. Also these two 8-byte can be in the same page.

The benefits of the same heap is that there is less overhead when task switches. Also, to share objects between tasks can be much more easier since they use the same heap.

However, with all tasks using the same heap, the heap can be the bottleneck of the applications with more and more cores in the modern processors. Actually, it's not rare that the heap allocation is the bottleneck especially in the emerging NVM allocators. Therefore, implementing one heap for each core ensure its scalability to multiple cores.

Linux

Pros: Per-process virtual memory gives the illusion to the process that they own the whole machine, which eases the programming. It also ensures the memory isolation between process and makes it easy to share code, so we don't have to keep several copies of the same code and can perform copy on write.

Cons: Virtual memory makes it hard to share data between processes. Every task switching requires to change the page table and syscall also asks for page table changing when entering kernel space.

Theseus

Pros: With single address space, the most significant advantage is the lower overhead of task switching. There is no need to change the page table. Also it doesn't need multiple privileges. Therefore, the context switching overhead is lower.

Cons: Unsafe code is a Necessity for the kernel which harms the security of the Theseus. Theseus also must trust the Rust compiler and its core/alloc libraries. If they are unreliable, with one single address space, one process can easily hack another process's data.

Experience:

Because it's the first time I write Rust, actually I don't get the convenience of `Result` and `Option`. Instead, investigating how to get the variable from them takes me a long time. Regardless of Rust language, it's much more convenient to implement `memuse` in Theseus. It's almost the same as writing an application function.

Interesting observations:

It seems there are memory leakage in Theseus, as discussed in Detail part. Also, Theseus maps all the stack pages to physical pages except the guard page when creating the stack, which is somewhat inefficient. Also since there is only one privilege, the `kstack` is somewhat confusing.

Though with these imperfections, it's a very interesting and even crazy system! I never imagine that the language could have such a big impact on the system design.

High level idea for part4

For heap memory, we can add a `heap_mem` field in the `Task` struct to maintain the heap size. We increase the `heap_mem` for every allocation and decrease the `heap_mem` for every deallocation.

For stack memory, As shown in the source code of [stack](#), it maps all the stack pages to physical pages except the guard page when creating the stack. Therefore, the stack memory usage is fixed, we can either record the fixed memory usage when creating the stack or calculate the difference between stack top and the bottom.

Detail:

Heap:

As illustrated above, we add a `heap_mem` field in the `Task`. We use the `MutexIrqSafe<usize>` to make sure it's mutable and atomic.

```
pub struct Task {
    /// The mutable parts of a `Task` struct that can be modified after task
    /// creation,
    /// excluding private items that can be modified atomically.
    ///
    /// We use this inner structure to reduce contention when accessing task
    /// struct fields,
    /// because the other fields aside from this one are primarily read, not
    /// written.
    ///
    /// This is not public because it permits interior mutability.
    inner: MutexIrqSafe<TaskInner>,

    pub heap_mem: MutexIrqSafe<usize>,
    ...
}
```

We increase the `heap_mem` for every allocation. Note that we can not simply increase the `heap_mem` by `layout.size` because there may be some requirements of the alignment. Therefore, we copy the alignment requirements of the `allocator.alloc()` to calculate the increased size and finally increase the `heap_mem` by `size`. Another thing notable is that we must first `lock()` the `heap_mem`, and then we can `deref_mut()` it to get its ownership. We can not write things like `a.heap_mem.lock().deref_mut()` because `a.heap_mem.lock()` is dropped after this line while `deref_mut()` requires the liveness of the it. At the end of it, we add some sentences to print the allocation info to the kernel log for debug.

```
fn increase_heap(layout: & Layout){
    let curr_task = task::get_my_current_task();
    // let tid = task::get_my_current_task_id();
    // let curr_task = task::get_task(tid);
    let mut heap_mem = match curr_task {
        Some(a) => a.heap_mem.lock(),
        None => return,
    };
    let heap_mem = heap_mem.deref_mut();
    // *heap_mem += layout.size();
    let mut size = 0;
```

```

let align = layout.align();
if align <= 8 && align <= layout.size() {
    size = size + layout.size();
} else {
    size += ((layout.size() - 1) / align) * align + align;
}
*heap_mem += &size;
let tid = task::get_my_current_task_id();
match tid {
    Some(a) => if a >= 14 {
        info!("task {:?} alloc mem_size: {:?}", a, &size);
    },
    None => return
}
}

```

The decrease logic is almost the same.

```

fn decrease_heap(layout: & Layout){
    let curr_task = task::get_my_current_task();
    // let pid = task::get_my_current_task_id();
    // let curr_task = task::get_task(pid);
    let mut heap_mem = match curr_task {
        Some(a) => a.heap_mem.lock(),
        None => return,
    };
    let heap_mem = heap_mem.deref_mut();
    // *heap_mem += layout.size();
    let mut size = 0;
    let align = layout.align();
    if align <= 8 && align <= layout.size() {
        size = size + layout.size();
    } else {
        size += ((layout.size() - 1) / align) * align + align;
    }
    *heap_mem -= &size;
    let tid = task::get_my_current_task_id();
    match tid {
        Some(a) => if a >= 14 {
            info!("task {:?} dealloc mem_size: {:?}", a, &size);
        },
        None => return
    }
}
}

```

Whenever there is an allocation/deallocation, we try to increase/decrease the accroding `heap_mem`.

```
unsafe impl GlobalAlloc for Heap {

    unsafe fn alloc(&self, layout: Layout) -> *mut u8 {
        increase_heap(&layout); // increase the heap_mem
        ...
    }

    unsafe fn dealloc(&self, ptr: *mut u8, layout: Layout) {
        decrease_heap(&layout); // decrease the heap_mem
        ...
    }

}
```

We implement `get_heap` in the `memuse` to get the heap physical memory.

```
fn get_heap(task_ref: task::TaskRef) -> usize{
    let heap_mem = task_ref.heap_mem.lock();
    *heap_mem
}
```

Stack:

As illustrated above, all the stack physical memory are mapped to physical pages when creating the stack.

The code path is `memuse` -> `Task` -> `TaskInner` -> `kstack` -> `pages`

Therefore, we implement a function `stack_size` in the `stack` struct.

```
pub fn stack_size(&self) -> usize{
    self.pages.size_in_bytes()
}
```

The `pages` in the `stack` represents the mapped pages of stack.

Then we implement a `stack_size` function in `Task`.

```
pub fn stack_size(&self) -> usize{
    let size = &self.inner.lock().kstack.stack_size();
    *size
}
```

We implement a `get_stack` in `memuse`.

```
fn get_stack(task_ref: task::TaskRef) -> usize{
    task_ref.stack_size()
}
```

Combination:

To implement `memuse`, we combine the `get_heap` and `get_stack` together.

Also we slightly change the `MemuseRes` from `pub type MemuseRes = Result<task::TaskRef, MemuseError>;`

to `pub type MemuseRes = Result<usize, MemuseError>;` since we are returning memory size now.

```
pub fn mymemuse(tid: usize, mem_type: MemType) -> MemuseRes {
    let mut res: usize = 0;
    // print to the kernel log
    info!("tid: {:?}, mem_type: {:?}", tid, mem_type);
    // check if the task exists
    let task_ref = match task::get_task(tid).ok_or(MemuseError::TaskNotExist) {
        Ok(a) => a,
        Err(e) => return Err(e),
    };
    // check if the memory type is valid
    match mem_type {
        MemType::Heap => res = get_heap(task_ref),
        MemType::CallStack => res = get_stack(task_ref),
        _ => return Err(MemuseError::InvalidMemtype),
    };

    Ok(res)
}
```

We generate some test in the `application/memuse_test/src/lib.rs`

```

fn recursive(depth: usize){
    if depth == 10{
        let tid = getpid();
        let mem_type = memuse::MemType::CallStack;
        println!("tid:{:?}, mem_type:{:?}", tid, &mem_type);
        let res = mymemuse(tid, mem_type);
        println!("memuse result: {:?}", &res);
        return;
    }
    recursive(depth + 1);
}

```

```

pub fn main() -> isize {
    let mut tid = getpid();
    let mut res;
    let mut res1;

    let mut mem_type = memuse::MemType::Heap;
    println!("tid:{:?}, mem_type:{:?}", tid, mem_type);
    res = mymemuse(tid, mem_type);

    let mut v = vec![0;20];

    tid = getpid();
    mem_type = memuse::MemType::Heap;
    res1 = mymemuse(tid, mem_type);
    println!("memuse result: {:?}", &res);
    println!("memuse result: {:?}", &res1);

    tid = getpid();
    mem_type = memuse::MemType::Heap;
    println!("tid:{:?}, mem_type:{:?}", tid, &mem_type);
    res = mymemuse(tid, mem_type);
    println!("memuse result: {:?}", &res);

    tid = getpid();
    mem_type = memuse::MemType::Heap;
    println!("tid:{:?}, mem_type:{:?}", tid, &mem_type);
    res = mymemuse(tid, mem_type);
    println!("memuse result: {:?}", &res);

    tid = getpid();
    mem_type = memuse::MemType::CallStack;
    println!("tid:{:?}, mem_type:{:?}", tid, &mem_type);
    res = mymemuse(tid, mem_type);
}

```

```
println!("memuse result: {:?}", &res);

    recursive(0);
0
}
```

We run the command `make run` terminal to run the Theseus and then run the command `memuse_test` in Theseus terminal.

The result is shown at the beginning of this part.

After allocating `vec![0;20]`, the heap size increases by 80 bytes as expected.

However, we notice that even without any explicit allocation, the heap size increases by 240 bytes after the function call. After investigating the reason, we find the `println!` seems to allocate some memory in heap but doesn't deallocate it.

Therefore, we take a look at the kernel log. It's obvious that there are much more allocation than deallocation. It seems there is memory leakage in Theseus.

[illegible]