CSCE 613 Final Project - Extended Rust Micro Kernel

Name: Yanze Li UIN: 326006687

Project Description

This goal of this project is using Rust programming language to implement a micro-kernel prototype. The intended features includes **Frame Allocator**, **Heap Allocator**, **Thread**, and **Process**.

The starting point of the project is a minimal bootable kernel with interrupt and VGA text mode support, plus a very limited frame allocator.

Therefore, in my project, I re-implemented the frame allocator from scratch and build a kernel heap allocator based on it. For the thread/process implementation, it is only required to support single-threaded process, therefore, the key difference between the thread and process in my project whether having a separate memory space.

By the time this report was written, the frame allocator and heap allocator has been fully implemented, with some passed test cases available. The heap allocator can successfully replace the global heap allocator, so we can use `Box` and `vec!` in the rest implementation. The implementation of thread/process is only partly done. I'm working on implementing the rest, and hopefully this will be done by the review session.

References

For frame allocation and heap allocation, following materials are referred:

1. Writing an OS in Rust (First Edition)
2. OS Dev
3. Understanding the Linux Virtual Memory Manager
4. CSCE 611 Machine Problem code

For thread/process:

1. Redox Source code (a Unix-like OS written in Rust)
2. Some documentations about X64 registers, such as (this)[https://docs.microsoft.com/en-us/windows-hardware/drivers/debugger/x64-architecture] and (this)[https://stackoverflow.com/questions/18024672/what-registers-are-preserved-through-a-linux-x86-64-function-call]

In addition to the materials mentioned above, I also discussed the design and some implementation difficulties with my lambastes (who are also in the class).

Design

Frame Allocation

My original design was directly taking my work from CSCE 611, which used a bitmap to manage the whole physical memory and scan the bitmap to allocate a continuous chunk of frames. The implementation ended up being ugly and and inefficient.

After some discussion with classmates, I decide to re-implement a buddy based frame allocator. To enable separate memory spaces, I manage each separate physical memory space as `Region` (similar to the concept of `frame_pool` in CSCE 611). This design is inspired by the `MemoryRegion` structure inside `BootInfo` crate. The definition of `Region` is as follows:

```
pub struct Region {
    // each node in the linked list denote a buddy
    // the content of each node is the start index of the frame
    // we can use this index to access FrameInfo object
    free_lists: [LinkedList<usize>; LEVEL_NUM],
    size: usize,
    free_frame_num: usize,
    // the base frame index of the region
    base_frame_idx: usize,
    // the start (usable) frame number of the region
    start_frame_idx: usize,
    frame_map: *mut FrameInfo,
}
```

The `free_lists` is a set of linked list that store buddies at different level. `size` denotes the total frame number of the region, `free_frame_num` denotes the number of frames that are free for allocation. We need to take some frames out from the region to store the meta-information of this region, which is stored in `frame_map`. The meta-information for each frame, namely `FrameInfo`, is stored here contiguously.

The design of `FrameInfo` is as follows:

```
pub struct FrameInfo {
    flgs: FrameFlags,
    count: u16,
    direct_access: usize,
    level: u32,
    // this is the global index
    index: usize,
}
```

The `flgs` is the same bit flag we used in CSCE 611. We take 2-bits to represent the usage of each frame, with `0x00` as FREE, `0x10` as TAKEN, and `0x11` as HEAD. `count` record the time of this frame being mapped by a virtual address. `level` indicates the level of buddy this frame belongs to.

A different field here is `direct_access`. We have this field because the blog chooses to use a completely mapped physical memory to manipulate data in physical memory (instead of using recursive page look up). Therefore, the `direct_access` here is the virtual address we can directly access the frame.

To make the frame allocator simpler, the actual frame allocation function is defined in each `region`, the frame allocator, i.e. `SimpleFrameAllocator`, only manages a set of regions and request or return the frames to them.

The design is as follows:

```
impl Region{
  pub fn request_frames(&mut self, frame_num: usize) -> Option<&'static mut
FrameInfo> {}

  pub fn retrieve_frame(&mut self, frame_info: &mut FrameInfo) {}
}

pub struct SimpleFrameAllocator {
    regions: [Region; MAX_REGION_NUM],
    region_num: usize,
}

impl SimpleFrameAllocator {
  pub fn register_region(&mut self, region: Region) {}

  pub fn alloc_frames(&mut self, frame_num: usize) -> Option<&'static mut
FrameInfo> {}

  pub fn dealloc_frame(&mut self, frame_info: &mut FrameInfo) {}
}
```

Heap Allocation

In CSCE 611, we take advantage of handling page fault to map the page table entries with physical frames. So that we can treat the virtual memory just as physical memory.

Considering in this kernel we already have a complete map between virtual memory and physical memory, we can simplify our implementation. To further make the kernel heap allocation more efficient, I leveraged some existing design in Linux kernel to pre-allocate some physical memory. Every time we free those heap memory, we re-collect them into a free list instead of actually free them.

The pre-allocate heap memories are also divided into several levels. In my design, it starts from 32 Byte to 4MB, which should suffice our use in the kernel space.

The overall design is shown below:

```
pub struct KernelHeapAllocator {
    pre_alloc_memory: [LinkedList<usize>; LEVEL_NUM],
    frame_allocator: SimpleFrameAllocator,
}

impl KernelHeapAllocator {
  // this function initialize the pre-allocated memory
  fn init(&mut self, pre_alloc_frame_num: usize) {}

  pub fn malloc(&mut self, layout: Layout) -> *mut u8 {}

  pub fn free(&mut self, ptr: *mut u8, layout: Layout) {}
}
```

Thread/Process

Since the thread and process have very few difference in our context, so I refer both of them to **process**.

Comparing to the thread design in CSCE 611, the major feature of process is **having a new memory space**. In other words, we need to reload `CR3` register every time we switch processes (so we have a different page table for each process).

Other than that, the design of process is identical to thread.

The structure of `Context` is as follows:

```
pub struct Context {
    cr3: usize, // we need to store CR3 as part of our context
    rflags: usize,
    rbx: usize,
    r12: usize,
    r13: usize,
    r14: usize,
    r15: usize,
    rbp: usize,
    // stack pointer
    rsp: usize,
    // stack content
    stack: Vec<u8>,
}

impl Context {
  pub fn set_stack(&mut self, addr: usize) {}

  pub unsafe fn push_stack(&mut self, value: usize) {}

  pub unsafe fn pop_stack(&mut self) -> usize {}
```

```
  pub fn set_cr3(&mut self, addr: usize) {}

  pub fn get_cr3(&self) -> usize {}

  pub unsafe fn switch_to(&mut self, next: &mut Context) {}
}
```

The design of process is as follows:

```
pub struct Process {
    init: bool,
    pid: usize,
    context: Context,
}

impl Process {
  fn init_page_table() -> usize {}

  pub fn dispatch_to(process: &mut Process) {}
}
```

As I haven't fully implement the process, the design of process may change in the future.

Implementation

Before implementing any concrete functionality, a simple data structure is needed. Here I implement a `LinkedList<T>` to store a list of data. The linked list has three basic operation, which are `append`, `pop` and `remove`. This structure is sufficient for implementing memory management.

The implementation of linked list without runtime support is actually painful. We have to heavily rely on raw pointer and unsafe operation, which undermined the security provided by Rust.

The key parts buddy-based frame allocation is how to initialize the physical memory and how to dynamic split and merge buddies.

1. For the initialization, we separate the whole physical memory into the highest level buddy. If we have a small piece left in the end, we put it to the second-highest level.
2. Every time we make an allocation, we find the level that fit into the requested space best. Then we recursively split the higher level buddies until we get a target level buddy.
3. Every time we free a buddy, we check if this buddy can be merged with the next contiguous buddy (they can be merged if they are of the same level). We do the merge recursively, and put the merge buddy into its corresponding free list.

The heap allocator is very simple. I define a fixed pre-allocation space for each level. The pre-allocated objects are kept in their corresponding free list. Every time we request a heap memory, we find the best fit and return an object (actually its virtual address) from the free list.

Once the heap allocator is implemented, we can replace the global allocator with our implementation using following code:

```
// Types for which it is safe to share references between threads.
unsafe impl Sync for KernelHeapAllocatorWrap {}
unsafe impl Send for KernelHeapAllocatorWrap {}

unsafe impl GlobalAlloc for KernelHeapAllocatorWrap {
    unsafe fn alloc(&self, layout: Layout) -> *mut u8 {
        let ptr = (*self.kernel_heap_allocator).malloc(layout);
        ptr
    }

    unsafe fn dealloc(&self, ptr: *mut u8, layout: Layout) {
        (*self.kernel_heap_allocator).free(ptr, layout);
    }
}

#[global_allocator]
static mut GLOBAL_ALLOCATOR: KernelHeapAllocatorWrap =
KernelHeapAllocatorWrap::new();

#[alloc_error_handler]
fn alloc_error_handler(layout: Layout) -> ! {
    println!("{:?}", layout);
    panic!("Allocation Error");
}
```

Then we can initialize the allocator in the `kernel_main` as follows:

```
// in kernel_main(boot_info: &'static BootInfo)
unsafe { memory::init(boot_info.physical_memory_offset) };
    unsafe { PHYSICAL_MEMORY_OFFSET = boot_info.physical_memory_offset as
usize };

    let mut frame_allocator = SimpleFrameAllocator::new();
    frame_allocator.init(&boot_info.memory_map);

    let mut heap_allocator = KernelHeapAllocator::new(frame_allocator, 1024);

    unsafe { GLOBAL_ALLOCATOR.init(&mut heap_allocator) };
```

Then we are able to enjoy programming with `Box` or `vec!`.

The implementation of context switch highly rely on inlined assembly. Most of my implementation for inlined assembly are modified from **redox OS**.

Conclusion & Future Work

Most of my design and implementation are simple and immature.

The future work can be done in following direction:

1. We can allocate frames with a combination of different buddy levels.
2. We can also assign levels to memory region, so that maybe we can separate kernel region and user region.
3. If we have multiple physical memory regions, we can request a large chunk of frames that cross regions.
4. Implement the function of map a physical address to a page table entry (through page fault).
5. Implement a scheduler for thread/process scheduling.
6. Other stuff like multi-core, filesystem, device driver, etc.

Using Rust to implement a kernel is a new experience. Before I finish implementing the memory management, using Rust is painful, as we have to rely on a lot of raw pointers. In general, the type system and the borrow checker provided by Rust make it less likely to produce bug, I believe it is promising to develop kernel using Rust.