OS开发中阶训练营第一周课后作业

预备

1. Fork ArceOS的工程，clone到本地。工程链接如下

```
git@github.com:rcore-os/arceos.git
```

2. 在main分支下，创建并切换到新的分支week1，执行

```
git checkout -b week1
```

后面的实验都在该分支下进行。

第一节课课后作业

支持HashMap数据类型。以apps/memtest为测试应用。

本次作业的主要目的为带领大家了解 arceos 的接口层次，主要修改内容在 ulib/axstd 文件夹中。

首先修改 apps/memtest/src/main.rs，把 BTreeMap 替换为 HashMap ，如下：

```
 use rand::{rngs::SmallRng, RngCore, SeedableRng};
-use std::collections::BTreeMap;
+use std::collections::HashMap;
 use std::vec::Vec;

 fn test_vec(rng: &mut impl RngCore) {
@@ -22,9 +22,9 @@ fn test_vec(rng: &mut impl RngCore) {
     println!("test_vec() OK!");
 }

-fn test_btree_map(rng: &mut impl RngCore) {
+fn test_hashmap_map(rng: &mut impl RngCore) {
     const N: usize = 50_000;
-    let mut m = BTreeMap::new();
+    let mut m = HashMap::new();
     for _ in 0..N {
         let value = rng.next_u32();
         let key = format!("key_{value}");
@@ -35,7 +35,7 @@ fn test_btree_map(rng: &mut impl RngCore) {
             assert_eq!(k.parse::<u32>().unwrap(), *v);
         }
     }
-    println!("test_btree_map() OK!");
+    println!("test_hashmap_map() OK!");
```

```
 }

 #[cfg_attr(feature = "axstd", no_mangle)]
@@ -44,7 +44,7 @@ fn main() {

     let mut rng = SmallRng::seed_from_u64(0xdead_beef);
     test_vec(&mut rng);
-    test_btree_map(&mut rng);
+    test_hashmap_map(&mut rng);

     println!("Memory tests run OK!");
 }
```

然后，尝试编译运行，`make A=apps/memtest ARCH=riscv64 run`，此时会报错，因为我们目前不支持HashMap类型。

```
➜  arceos git:(os_camp) make A=apps/memtest ARCH=riscv64 run
    Building App: memtest, Arch: riscv64, Platform: riscv64-qemu-virt, App type:
rust
cargo build --target riscv64gc-unknown-none-elf --target-dir
/home/hky/workspace/rcore-os/arceos/target --release  --manifest-path
apps/memtest/Cargo.toml --features "axstd/log-level-warn"
   Compiling arceos-memtest v0.1.0 (/home/hky/workspace/rcore-
os/arceos/apps/memtest)
error[E0432]: unresolved import `std::collections::HashMap`
 --> apps/memtest/src/main.rs:9:5
  |
9 | use std::collections::HashMap;
  |     ^^^^^^^^^^^^^^^^^^^^^^^^^ no `HashMap` in `collections`

For more information about this error, try `rustc --explain E0432`.
error: could not compile `arceos-memtest` (bin "arceos-memtest") due to 1 previous
error
make: *** [scripts/make/build.mk:36: _cargo_build] Error 101
```

要求：在ulib/axstd中支持HashMap类型

预期输出：执行`make A=apps/memtest ARCH=riscv64 run`

```
arch = riscv64
platform = riscv64-qemu-virt
target = riscv64gc-unknown-none-elf
smp = 1
build_mode = release
log_level = warn

Running memory tests...
test_vec() OK!
```

```
test_hashmap_map() OK!
Memory tests run OK!
```

提示 ：

1. 参考官方rust标准库中的HashMap实现，把涉及的代码拷过来，做一下修改。只需要满足memtest的测试需要即可。

（ 在 axstd 中使用 rust 标准库中的代码可能需要给 axstd 增加一些 feature，可能需要增加这些： ）

```
// ulib/axstd/src/lib.rs
#![cfg_attr(all(not(test), not(doc)), no_std)]
#![feature(doc_cfg)]
#![feature(doc_auto_cfg)]
#![feature(hashmap_internals)]
#![feature(extend_one)]
#![feature(hasher_prefixfree_extras)]
#![feature(error_in_core)]
#![feature(try_reserve_kind)]
#![feature(thread_local)]
#![feature(const_hash)]
```

2. 注意：官方std与ArceOS的axstd的区别。官方rust标准库主要是基于Linux/Windows这些内核，为应用提供的用户库。官方std的支持后端是libc+syscall；而ArceOS是单特权级，没有syscall一说，axstd直接通过一系列function-call调用底层的功能。

3. HashMap之所以没有像其他collections类型一样放到alloc库中实现，主要是因为它需要随机数的支持，而随机数的产生机制是平台相关的。类似的代码可以在 std/src/hash/randoms.rs 中对 `RandomState` 的初始化代码中找到。 大家做实验可以简单点，用一个软实现的随机数函数来产生。比如

```
use spinlock::SpinNoIrq;
use crate::time;

static PARK_MILLER_LEHMER_SEED: SpinNoIrq<u32> = SpinNoIrq::new(0);
const RAND_MAX: u64 = 2_147_483_647;

pub fn random() -> u128 {
    let mut seed = PARK_MILLER_LEHMER_SEED.lock();
    if *seed == 0 {
        *seed = time::current_ticks() as u32;
    }

    let mut ret: u128 = 0;
    for _ in 0..4 {
        *seed = ((u64::from(*seed) * 48271) % RAND_MAX) as u32;
        ret = (ret << 32) | (*seed as u128);
    }
    ret
}
```

第二节课课后作业

在第一节课的基础上，针对字节内存分配，新增一个简单的内存分配算法。

本次作业的主要目的为带领大家了解arceos的内存分配接口，主要修改内容在 crates/allocator 文件夹中。

准备：阅读 crates/allocator/src/lib.rs，了解 arceos 目前支持的内存分配算法，思考它们的区别与适用场景。

要求：在 crates/allocator 中支持一个新的内存分配算法，传递 feature 为 "new"。

```
// crates/allocator/src/lib.rs
#[cfg(feature = "bitmap")]
mod bitmap;
#[cfg(feature = "bitmap")]
pub use bitmap::BitmapPageAllocator;

#[cfg(feature = "buddy")]
mod buddy;
#[cfg(feature = "buddy")]
pub use buddy::BuddyByteAllocator;

#[cfg(feature = "slab")]
mod slab;
#[cfg(feature = "slab")]
pub use slab::SlabByteAllocator;

#[cfg(feature = "new")]
mod new;
#[cfg(feature = "new")]
pub use new::YourNewAllocator;
```

注意修改 allocator 的 Cargo.toml

```
[features]
default = []
full = ["bitmap", "tlsf", "slab", "buddy", "allocator_api", "new"]

bitmap = ["dep:bitmap-allocator"]

tlsf = ["dep:rlsf"]
slab = ["dep:slab_allocator"]
buddy = ["dep:buddy_system_allocator"]
new = ["dep:your_dep_if_needed"]
```

提示：

~~可以使用 https://github.com/rust-osdev/linked-list-allocator~~

推荐使用 https://crates.io/crates/talc

也可以去 crates.io 里面寻找更多的合适的 allocator

包括 linked-list-allocator 在内大部分现成的 allocator 都没有现成的 `add_memory` 功能实现（可能需要您自己修改）

但是 arceos 的 allocator crate 定义的 `BaseAllocator Trait` 中的 `add_memory` 接口要求实现类似功能

```
impl BaseAllocator for YourNewByteAllocator {
    fn init(&mut self, start: usize, size: usize) {
        /// Your implementation.
    }

    fn add_memory(&mut self, _start: usize, _size: usize) -> AllocResult {
        /// Your implementation.
    }
}
```

为什么 arceos 的 `BaseAllocator Trait` 要求实现类似的动态拓展堆空间的功能？

观察定义在 `modules/axalloc/src/lib.rs` 中的 `GlobalAllocator struct`

```
/// The global allocator used by ArceOS.
///
/// It combines a [`ByteAllocator`] and a [`PageAllocator`] into a simple
/// two-level allocator: firstly tries allocate from the byte allocator, if
/// there is no memory, asks the page allocator for more memory and adds it to
/// the byte allocator.
///
/// Currently, [`TlsfByteAllocator`] is used as the byte allocator, while
/// [`BitmapPageAllocator`] is used as the page allocator.
///
/// [`TlsfByteAllocator`]: allocator::TlsfByteAllocator
pub struct GlobalAllocator {
    balloc: SpinNoIrq<DefaultByteAllocator>,
    palloc: SpinNoIrq<BitmapPageAllocator<PAGE_SIZE>>,
}
```

这是一个 "two-level allocator", 当 byte allocator 中的堆空间不足时，会从 page allocator 中申请更多的内存空间并塞到 byte allocator 里面

具体的实现细节可以查看 `GlobalAllocator` 的 `alloc` 方法：

```
impl GlobalAllocator {
    /// Allocate arbitrary number of bytes. Returns the left bound of the
    /// allocated region.
    ///
```

```
    /// It firstly tries to allocate from the byte allocator. If there is no
    /// memory, it asks the page allocator for more memory and adds it to the
    /// byte allocator.
    ///
    /// `align_pow2` must be a power of 2, and the returned region bound will be
    ///  aligned to it.
    pub fn alloc(&self, layout: Layout) -> AllocResult<NonNull<u8>> {
        // simple two-level allocator: if no heap memory, allocate from the page
allocator.
        let mut balloc = self.balloc.lock();
        loop {
            if let Ok(ptr) = balloc.alloc(layout) {
                return Ok(ptr);
            } else {
                let old_size = balloc.total_bytes();
                let expand_size = old_size
                    .max(layout.size())
                    .next_power_of_two()
                    .max(PAGE_SIZE);
                let heap_ptr = self.alloc_pages(expand_size / PAGE_SIZE,
PAGE_SIZE)?;
                debug!(
                    "expand heap memory: [{:#x}, {:#x})",
                    heap_ptr,
                    heap_ptr + expand_size
                );
                balloc.add_memory(heap_ptr, expand_size)?;
            }
        }
    }
}
```

由于 `apps/memtest/src/main.rs` 的测例中 `test_vec` 方法开了一个 3_000_000 大小的 u32 Vec，堆空间大概率是会不够的，还是需要动态往里塞点内存页（~~也可以选择一开始把堆变大一些~~）

注意:

参考 ulib/axstd/Cargo.toml 中关于内存管理算法的 features 增加一个新的 feature 选项。

```
# ulib/axstd/Cargo.toml
# Memory
alloc = ["arceos_api/alloc", "axfeat/alloc", "axio/alloc"]
alloc-tlsf = ["axfeat/alloc-tlsf"]
alloc-slab = ["axfeat/alloc-slab"]
alloc-buddy = ["axfeat/alloc-buddy"]
alloc-new = ["axfeat/alloc-new"]
```

参考 api/axfeat/Cargo.toml 中关于内存管理算法的 features 增加一个新的 feature 选项。

```
# api/axfeat/Cargo.toml
# Memory
alloc = ["axalloc", "axruntime/alloc"]
alloc-tlsf = ["axalloc/tlsf"]
alloc-slab = ["axalloc/slab"]
alloc-buddy = ["axalloc/buddy"]
alloc-new = ["axalloc/new"]
```

修改 modules/axalloc/Cargo.toml 中关于内存管理算法的 features 增加一个新的 feature 选项。

```
# modules/axalloc/Cargo.toml
[features]
default = ["tlsf"] # 默认采用 tlsf 内存分配算法
tlsf = ["allocator/tlsf"]
slab = ["allocator/slab"]
buddy = ["allocator/buddy"]
new = ["allocator/new"]
```

重要：

修改 `modules/axalloc/src/lib.rs` 第26行的 cfg_if 代码块，添加 `#[cfg(feature = "new")]` 条件下使用新增的 allocator 作为 `DefaultByteAllocator`

注意添加的顺序要在 `#[cfg(feature = "tlsf")]` 之前，否则由于 axalloc 的 default feature 为 `"tlsf"`，若 tlsf 放在前面 axalloc 还是会启用 `TlsfByteAllocator` (或者注释掉 `modules/axalloc/Cargo.toml` 中的 `default = ["tlsf"]` 也行)

```
cfg_if::cfg_if! {
    if #[cfg(feature = "slab")] {
        use allocator::SlabByteAllocator as DefaultByteAllocator;
    } else if #[cfg(feature = "buddy")] {
        use allocator::BuddyByteAllocator as DefaultByteAllocator;
    } else if #[cfg(feature = "new")] {
        use allocator::YourNewByteAllocator as DefaultByteAllocator;
    } else if #[cfg(feature = "tlsf")] {
        use allocator::TlsfByteAllocator as DefaultByteAllocator;
    }
}
```

同样，修改 `pub const fn name(&self) -> &'static str` 方法，为您新增的 allocator 起一个名字

```
impl GlobalAllocator {
    /// Returns the name of the allocator.
    pub const fn name(&self) -> &'static str {
        cfg_if::cfg_if! {
            if #[cfg(feature = "slab")] {
```

```
                "slab"
            } else if #[cfg(feature = "buddy")] {
                "buddy"
            } else if #[cfg(feature = "new")] {
                "Your new allocator"
            } else if #[cfg(feature = "tlsf")] {
                "TLSF"
            }
        }
    }
}
```

预期输出：执行`make A=apps/memtest ARCH=riscv64 run FEATURES=alloc-new`，结果应该与第一次作业一致。

```
arch = riscv64
platform = riscv64-qemu-virt
target = riscv64gc-unknown-none-elf
smp = 1
build_mode = release
log_level = warn

Running memory tests...
test_vec() OK!
test_hashmap_map() OK!
Memory tests run OK!
```

多说一句：arceos 目前的 `crates/allocator` 里面写好了一些现成的bench，可以进入 `crates/allocator` 目录使用 `cargo bench` 命令测试不同内存分配算法的性能，大家加上新的allocator之后也可以选择加到bench里面跟现有的allocator做一个对比。

作业提交

1、题目做完以后, 使用 git diff 将本次作业修改的内容提取成补丁文件,命名为 用户名-题号.patch

例如:

```
git diff main > pengzechen-lesson1.patch
```

2、提交之前**删除根目录的target目录**, 将剩余文件(包括上一步生成的patch文件)打包成**zip压缩包**，使用固定邮箱发送到指定邮箱: 173701980@qq.com, **邮箱标题**命名为 **学员名称-小组名称-lesson{题号}**，没有加入小组可命名为 **单人**。

（例如：第一题课后作业邮箱标题为 "唐翰文-测试小组-lesson1" 或 "唐翰文-单人-lesson1"）

注意：

第一周的作业主要是为了让大家熟悉 arceos 的代码结构，作业本身难度并不高，只是需要有限的工作量。

作业的预期输出比较简单，老师与助教会手动查看代码实现，**希望大家不要通过修改输出结果来欺骗自动评测脚本**。