OS 大实验结题汇报

2024.6.15 · 致理-信计11 游宇凡

主要工作

LearningOS/osbiglab-2024s-verifyingkernel

- 了解了 OS 形式化验证的相关工作
  - 复现了 HyperKernel 的运行与验证
  - 学习了 Verus 工具的使用，编写了一些小练习，给 Verus 提的 4 个 PR 被 merge
- 构建出了内存相关组件经验证的 ArceOS
  - 编写并验证了 **memory allocator**、memory addr 组件
  - 将 Matthias Brun 的 verified page table 接入 ArceOS

HyperKernel

- 阅读了论文 Hyperkernel: Push-Button Verification of an OS Kernel
- 搭建了用于复现的 Docker 运行环境，能够运行 kernel 以及验证

Verus 环境配置

- 使用 Git submodule + direnv 在 repo 内配置了 Verus 运行环境
- 配置了 GitHub Actions，会在 CI 上自动运行 Verus 验证，以及运行 ArceOS apps

学习 Verus

- 基础教程：Verus Tutorial and Reference
- 论文 Verus: Verifying Rust Programs using Linear Ghost Types
- 示例代码
- 标准库 vstd 文档
- 其他现有项目的代码
- Zulip 提问，与作者取得联系

Linear Ghost Type (permission token)

这个功能在目前的 tutorial 中还是 todo，缺少相关介绍。但它对于验证指针操作、内存分配等功能非常有用，论文 Atmosphere: Towards Practical Verified Kernels in Rust 称其为“the true power of Verus”，值得学习。

Linear Ghost Type (permission token)

这个功能在目前的 tutorial 中还是 todo，缺少相关介绍。但它对于验证指针操作、内存分配等功能非常有用，论文 Atmosphere: Towards Practical Verified Kernels in Rust 称其为“the true power of Verus”，值得学习。

variable mode：exec、tracked、ghost

Linear Ghost Type (permission token)

这个功能在目前的 tutorial 中还是 todo，缺少相关介绍。但它对于验证指针操作、内存分配等功能非常有用，论文 Atmosphere: Towards Practical Verified Kernels in Rust 称其为“the true power of Verus”，值得学习。

variable mode：exec、tracked、ghost

- *tracked*、ghost：ghost code，仅用于验证，没有运行时开销 （exec：正常的可执行代码中的变量）
- exec、*tracked*：受 ownership rule 制约，不能随意复制，有唯一 owner （ghost：可以随意复制）

Linear Ghost Type (permission token)

这个功能在目前的 tutorial 中还是 todo，缺少相关介绍。但它对于验证指针操作、内存分配等功能非常有用，论文 Atmosphere: Towards Practical Verified Kernels in Rust 称其为“the true power of Verus”，值得学习。

variable mode：exec、tracked、ghost

- *tracked*、ghost：ghost code，仅用于验证，没有运行时开销 （exec：正常的可执行代码中的变量）
- exec、*tracked*：受 ownership rule 制约，不能随意复制，有唯一 owner （ghost：可以随意复制）

tracked：既没有运行时开销，又有唯一 owner，可以用来作为 permission token 辅助验证。

Linear Ghost Type (permission token)

这个功能在目前的 tutorial 中还是 todo，缺少相关介绍。但它对于验证指针操作、内存分配等功能非常有用，论文 Atmosphere: Towards Practical Verified Kernels in Rust 称其为“the true power of Verus”，值得学习。

variable mode：exec、tracked、ghost

- *tracked*、ghost：ghost code，仅用于验证，没有运行时开销 （exec：正常的可执行代码中的变量）
- exec、*tracked*：受 ownership rule 制约，不能随意复制，有唯一 owner （ghost：可以随意复制）

tracked：既没有运行时开销，又有唯一 owner，可以用来作为 permission token 辅助验证。

例如：`PointsToRaw` 表示拥有若干内存空间，可以作为 `PPtr` 相关函数的参数来获取指针访问内存的权限，而正常情况下需要通过 `alloc` 获取 `PointsToRaw` 类型的实例（或者通过 `new` 获取 `PointsTo<T>`），这保证了拥有 `PointsToRaw` 即意味着有权限访问一段内存，而权限拥有者（内存访问者）永远是唯一的。

练习

- 计算 Fibonacci 数列
- 还是计算 Fibonacci 数列，但是参照论文《Verified Paging for x86-64 in Rust》的证明结构，通过 state machine refinement 的结构证明
- bump allocator，最简单的 memory allocator，验证了其正确性（alloc 满足参数要求，分配出的内存不会相交，运行过程中不会发生算术溢出等错误），并实现了 `GlobalAlloc` trait，可以直接在 `no_std` 下使用

Pull Requests

向 Verus 提交的 4 个 PR 被 merge

- Use `bit_vector` instead of `external_body` verifier for `set_bit64_proof`
- Add `insert` and `take` methods for `PointsToRaw` (inplace `join` / `split`)
- Add `no-std` and `no-alloc` build tests in CI
- Use `OnceLock` instead of `lazy_static` for `cfg_verify_core`

ArceOS modules

对整个 OS 进行形式化验证工作量较大

ArceOS 是组件化的 OS

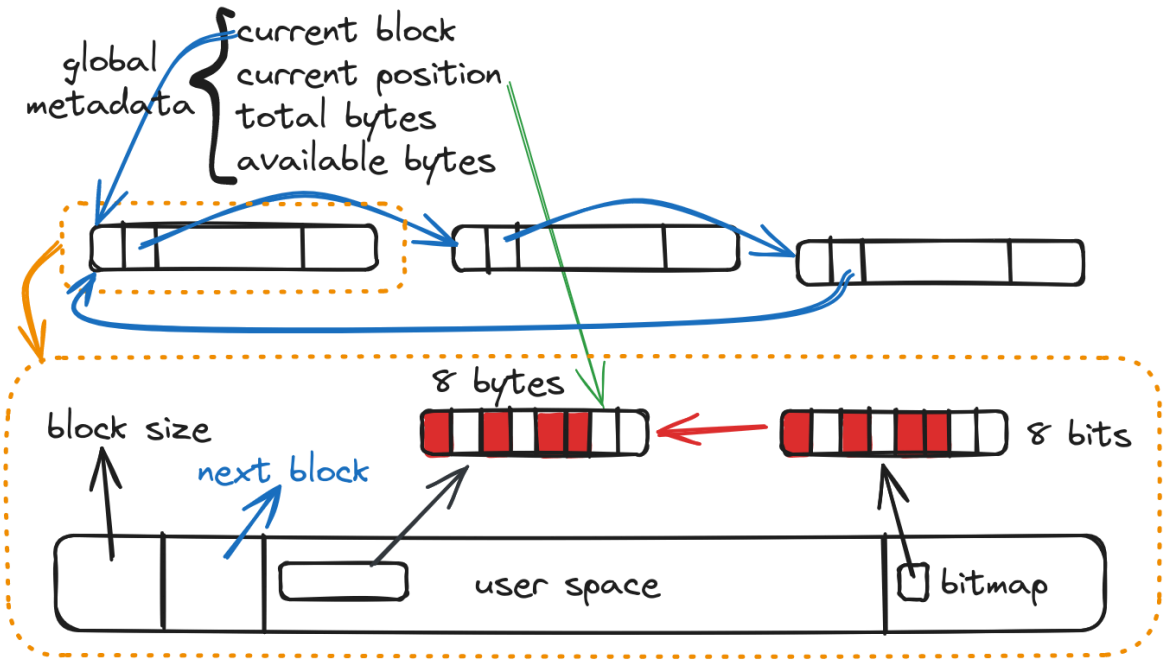
可以对一部分组件进行验证，替换掉 ArceOS 原有的组件

- memory allocator
- page table

内存相关组件经验证的 ArceOS

Verified Bitmap Allocator

编写并验证了 bitmap allocator 的正确性，是本大实验最核心的成果



示意图使用 https://excalidraw.com/ 绘制

结构设计

Verified Bitmap Allocator

- API 支持添加不连续的 memory region，所以设计上把每次添加进来的 memory region 独立作为一个 block。
- 每个 block 内部通过 bitmap 记录内存分配情况，即一个 block 的空间分为 metadata、bitmap、user space 三部分，其中 bitmap 的每个 bit 表示 user space 中对应的一个 byte 是否 allocated。
- 各个 block 之间通过循环链表相连，metadata 中存储 block size 和链表中下一个 block 的地址。
- 在全局的 allocator 中，还记录了当前 block 以及当前 byte 的位置，相当于实现了 next fit，每次从上次搜索结束的位置开始往后搜索可用的空闲内存，以提升运行效率。

实现细节

Verified Bitmap Allocator

模块划分：

- bitmask：位运算
- block：单个 block
- allocator：整体的 allocator

实现细节

Verified Bitmap Allocator

模块划分：

- bitmask：位运算
- block：单个 block
- allocator：整体的 allocator

---

bitmask：主要验证了 bset、bclr、bext 这三个位运算的相关性质，即设置、清除、查询一个 bit。这些位运算以及相关引理在 bitmap 相关操作中被调用。

实现细节：block

Verified Bitmap Allocator

```
pub tracked struct BitmapBlock {
    size_pt: PointsTo<usize>,
    next_pt: PointsTo<usize>,
    user_pt: PointsToRaw,
    mask_pt_map: Map<int, PointsTo<usize>>,
}
```

实现细节：block

Verified Bitmap Allocator

```
pub tracked struct BitmapBlock {
    size_pt: PointsTo<usize>,
    next_pt: PointsTo<usize>,
    user_pt: PointsToRaw,
    mask_pt_map: Map<int, PointsTo<usize>>,
}
```

- `new`：需要传入对应于这个 block memory 的 permission token，然后会将这块 block memory 拆解成小块的 size、next、user space、bitmap 的 permission token，并写入各个字段以及清空 bitmap，完成初始化。

实现细节：block

Verified Bitmap Allocator

```
pub tracked struct BitmapBlock {
    size_pt: PointsTo<usize>,
    next_pt: PointsTo<usize>,
    user_pt: PointsToRaw,
    mask_pt_map: Map<int, PointsTo<usize>>,
}
```

- `new`：需要传入对应于这个 block memory 的 permission token，然后会将这块 block memory 拆解成小块的 size、next、user space、bitmap 的 permission token，并写入各个字段以及清空 bitmap，完成初始化。
- `alloc`：遍历 bitmap，找到足够大的连续空闲内存，然后更新 bitmap，并删去 `user_pt` 中相应的部分，收集起来作为返回值，以证明 allocate 出来的这些内存可以安全地转移给 caller 使用。

实现细节：block

Verified Bitmap Allocator

```
pub tracked struct BitmapBlock {
    size_pt: PointsTo<usize>,
    next_pt: PointsTo<usize>,
    user_pt: PointsToRaw,
    mask_pt_map: Map<int, PointsTo<usize>>,
}
```

- `new`：需要传入对应于这个 block memory 的 permission token，然后会将这块 block memory 拆解成小块的 size、next、user space、bitmap 的 permission token，并写入各个字段以及清空 bitmap，完成初始化。
- `alloc`：遍历 bitmap，找到足够大的连续空闲内存，然后更新 bitmap，并删去 `user_pt` 中相应的部分，收集起来作为返回值，以证明 allocate 出来的这些内存可以安全地转移给 caller 使用。
- `dealloc`：需要传入相应的 permission token，会更新 bitmap，并将传入的 permission token 再放回原处。

实现细节：block

Verified Bitmap Allocator

为了辅助证明，会证明整个 block 一直保持着一些不变量（即每个函数的 precondition 要求满足这些性质，postcondition 证明仍然满足这些性质），包括：

- block 地址加上整个 block 的大小不超过 `usize::MAX`
- block 地址对齐到 `usize` 的 alignment
- 各个 field 的 permission token 对应的地址位于它们该在的地方，且已赋值（initialized）
- user space permission token 与 bitmap 的取值是一致的，即 bitmap 中是 1 当且仅当对应的 permission token 存在

实现细节：allocator

Verified Bitmap Allocator

```
pub struct BitmapAllocator {
    current_block_addr: Option<usize>,
    current_pos: usize,
    total_bytes: usize,
    available_bytes: usize,
    block_seq: Ghost<Seq<usize>>,
    block_map: Tracked<Map<usize, BitmapBlock>>,
}
```

实现细节：allocator

Verified Bitmap Allocator

```
pub struct BitmapAllocator {
    current_block_addr: Option<usize>,
    current_pos: usize,
    total_bytes: usize,
    available_bytes: usize,
    block_seq: Ghost<Seq<usize>>,
    block_map: Tracked<Map<usize, BitmapBlock>>,
}
```

- `add_memory`：首先检查是否有内存相交，然后创建新的 block，更新链表指针以及全局维护的相关信息。

实现细节：allocator

Verified Bitmap Allocator

```
pub struct BitmapAllocator {
    current_block_addr: Option<usize>,
    current_pos: usize,
    total_bytes: usize,
    available_bytes: usize,
    block_seq: Ghost<Seq<usize>>,
    block_map: Tracked<Map<usize, BitmapBlock>>,
}
```

- `add_memory`：首先检查是否有内存相交，然后创建新的 block，更新链表指针以及全局维护的相关信息。
- `alloc`：从当前 block 和 byte position 开始尝试调用各个 block 的 `alloc`，成功则更新全局信息并返回。

实现细节：allocator

Verified Bitmap Allocator

```
pub struct BitmapAllocator {
    current_block_addr: Option<usize>,
    current_pos: usize,
    total_bytes: usize,
    available_bytes: usize,
    block_seq: Ghost<Seq<usize>>,
    block_map: Tracked<Map<usize, BitmapBlock>>,
}
```

- `add_memory`：首先检查是否有内存相交，然后创建新的 block，更新链表指针以及全局维护的相关信息。
- `alloc`：从当前 block 和 byte position 开始尝试调用各个 block 的 `alloc`，成功则更新全局信息并返回。
- `dealloc`：寻找地址位于哪个 block，然后调用这个 block 的 `dealloc`。

实现细节：allocator

Verified Bitmap Allocator

```
pub struct BitmapAllocator {
    current_block_addr: Option<usize>,
    current_pos: usize,
    total_bytes: usize,
    available_bytes: usize,
    block_seq: Ghost<Seq<usize>>,
    block_map: Tracked<Map<usize, BitmapBlock>>,
}
```

- `add_memory`：首先检查是否有内存相交，然后创建新的 block，更新链表指针以及全局维护的相关信息。
- `alloc`：从当前 block 和 byte position 开始尝试调用各个 block 的 `alloc`，成功则更新全局信息并返回。
- `dealloc`：寻找地址位于哪个 block，然后调用这个 block 的 `dealloc`。

其中，`add_memory` 和 `dealloc` 需要传入 permission token，而为了能在 Verus 外使用，还提供了 `unsafe_add_memory` 和 `unsafe_dealloc`，它们不需要传入 permission token，而是会内部通过 `unsafe_obtain_pt_for_region` 函数获取到需要的 permission token，需要 caller 保证调用正确（只不过 `add_memory` 还是会检查是否有 memory region 相交的情况）。

实现细节：allocator

Verified Bitmap Allocator

不变量主要包括：

实现细节：allocator

Verified Bitmap Allocator

不变量主要包括：

- `block_seq` 正确：
  - 记录的当前 block 地址是 seq 的第一个元素（或者 allocator 为空，一个 block 都没有）
  - seq 中每个元素的链表指针指向 seq 中的下一个元素，最后一个元素指向第一个元素
  - seq 中没有重复元素

实现细节：allocator

Verified Bitmap Allocator

不变量主要包括：

- `block_seq` 正确：
  - 记录的当前 block 地址是 seq 的第一个元素（或者 allocator 为空，一个 block 都没有）
  - seq 中每个元素的链表指针指向 seq 中的下一个元素，最后一个元素指向第一个元素
  - seq 中没有重复元素
- `block_map` 正确：
  - `block_map` 的定义域等于 `block_seq`
  - block 的地址区间均不相交
  - 每个 block（`BitmapBlock`）满足其不变性质

实现细节：allocator

Verified Bitmap Allocator

不变量主要包括：

- `block_seq` 正确：
  - 记录的当前 block 地址是 seq 的第一个元素（或者 allocator 为空，一个 block 都没有）
  - seq 中每个元素的链表指针指向 seq 中的下一个元素，最后一个元素指向第一个元素
  - seq 中没有重复元素
- `block_map` 正确：
  - `block_map` 的定义域等于 `block_seq`
  - block 的地址区间均不相交
  - 每个 block（`BitmapBlock`）满足其不变性质
- 全局信息正确：
  - 当前 byte position 不超过当前 block size
  - available bytes 不超过 total bytes

正确性证明

Verified Bitmap Allocator

最终证明的正确性如下：

正确性证明

Verified Bitmap Allocator

最终证明的正确性如下：

- `add_memory`：报错时 allocator 没有发生改变，没有报错时这段 memory region 真的作为一个 block 添加进了 allocator，且不变量保证了各个 memory region 不相交。
- `alloc`：报错时 allocator 没有发生改变，没有报错时返回的地址满足对齐要求，且返回的 permission token 保证了 allocator 有权将这段内存转移给 caller 使用。
- `dealloc`：传入的这段内存要么不位于任何一个 block（正确调用时不应发生），要么被正确释放。

正确性证明

Verified Bitmap Allocator

最终证明的正确性如下：

- `add_memory`：报错时 allocator 没有发生改变，没有报错时这段 memory region 真的作为一个 block 添加进了 allocator，且不变量保证了各个 memory region 不相交。
- `alloc`：报错时 allocator 没有发生改变，没有报错时返回的地址满足对齐要求，且返回的 permission token 保证了 allocator 有权将这段内存转移给 caller 使用。
- `dealloc`：传入的这段内存要么不位于任何一个 block（正确调用时不应发生），要么被正确释放。

此外，permission token 避免了使用 `unsafe`，指针访问都是经验证的；不会发生算术溢出等错误。

代码量

Verified Bitmap Allocator

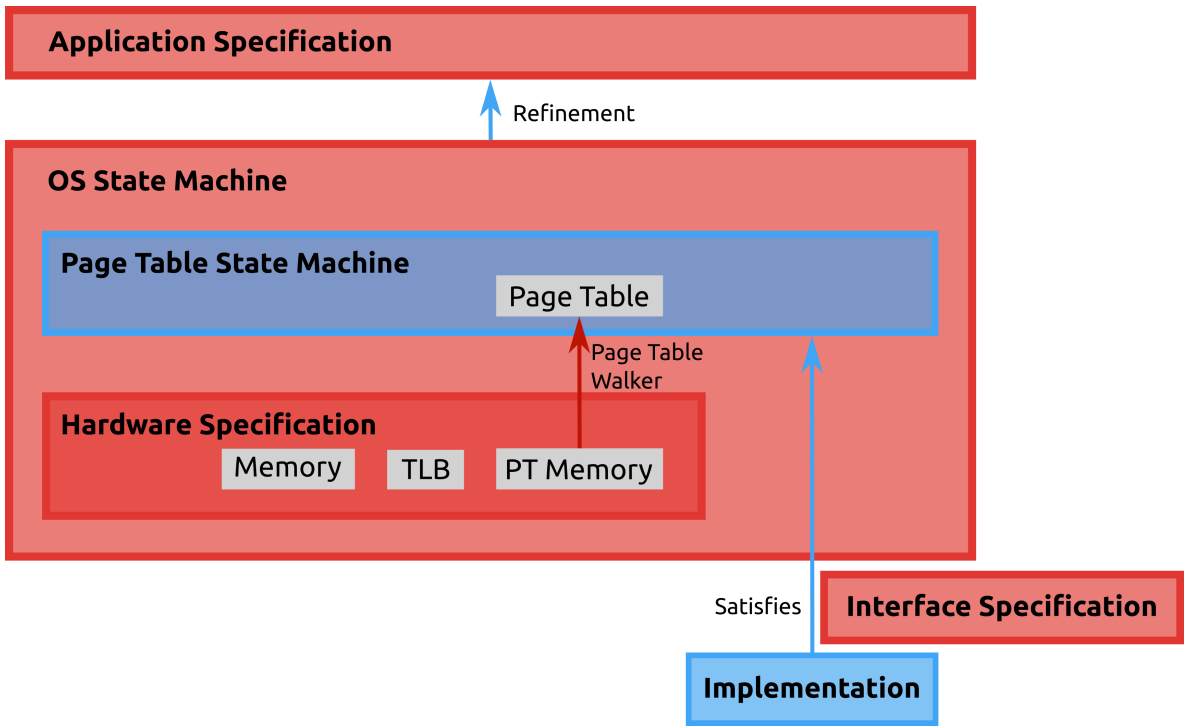
验证代码约占总行数的 70%

file	Spec	Proof	Exec	Proof+Exec	Comment	Layout	unaccounted	Directives
`lib.rs`	0	0	6	0	0	0	15	1
`bitmap/allocator.rs`	133	160	130	9	36	4	35	1
`bitmap/bitmask.rs`	45	20	12	0	0	0	21	3
`bitmap/block.rs`	162	70	125	20	0	0	37	0
total	340	250	273	29	36	4	113	5

使用 Verus 官方的 `line_count` 工具自动统计

Verified Page Table

将 Matthias Brun 的 verified page table 接入 ArceOS



接入 ArceOS

Verified Page Table

1. 将 verified-nrkernel 中的代码复制出来，它可以用最新版 Verus 进行验证，但需要进行一些修改才能编译。
2. 添加使用外部 page allocator 来 alloc/dealloc page 的接口。
3. 添加设置 PTE 的 disable cache flag 的接口并补充相关验证代码（这个 flag 在 ArceOS 中被 MMIO 使用）。
4. 添加用于接入 ArceOS 的上层接口。
5. 修复 bug：verified page table 之前只支持 lower half (user space) virtual address，但 ArceOS 用的是 higher half (kernel space) virtual address，直接传过去就会出错。调了很久才找到错误的原因，但修起来很简单，因为 PTE index 看的是 virtual address 的低 48 位（lower half 的高 17 位、higher half 的高 17 位分别是固定的），只需要取低 48 位传给 verified page table 即可。

最终可以通过 Verus 验证，并成功编译运行了启用了 paging feature 的 shell、httpserver、httpclient 等 app。

Verified `memory_addr`

验证了 ArceOS 的 `memory_addr` crate，主要是证明了相关位运算确实能达到期望的地址对齐效果。

Q & A

感谢各位老师、助教以及同学们！