


北京大学
PEKING UNIVERSITY

星绽 MM：虚拟内存管理的统一抽象设计

CortenMM: A High-Performance Memory System with Synchronization Correctness through Unified Abstraction

0. 总览

汇报本小组在安全内核项目下，针对虚拟内存管理问题的初步设计、实验和结论。

相比 Linux：
- 星绽MM大幅提升了虚拟内存系统调用和缺页中断的多核可扩展性；
- 并从机制上确保了并发安全性。

汇报大纲：
1. 问题背景与动机
2. 系统设计与实现
3. 实验评估
4. 总结和讨论

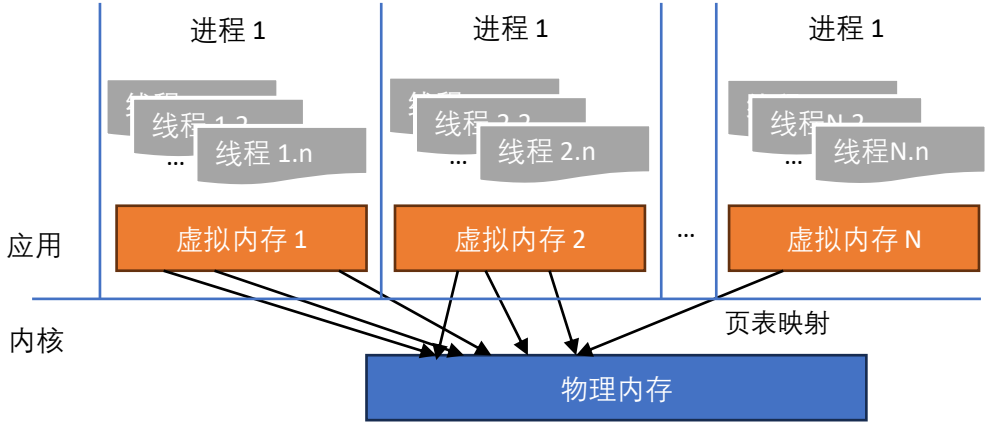
1.问题背景-虚拟内存管理（应用视角）

操作系统内核为应用提供虚拟内存抽象:

每个进程拥有独立的虚拟地址空间，进程内的所有线程共享这个虚拟地址空间。

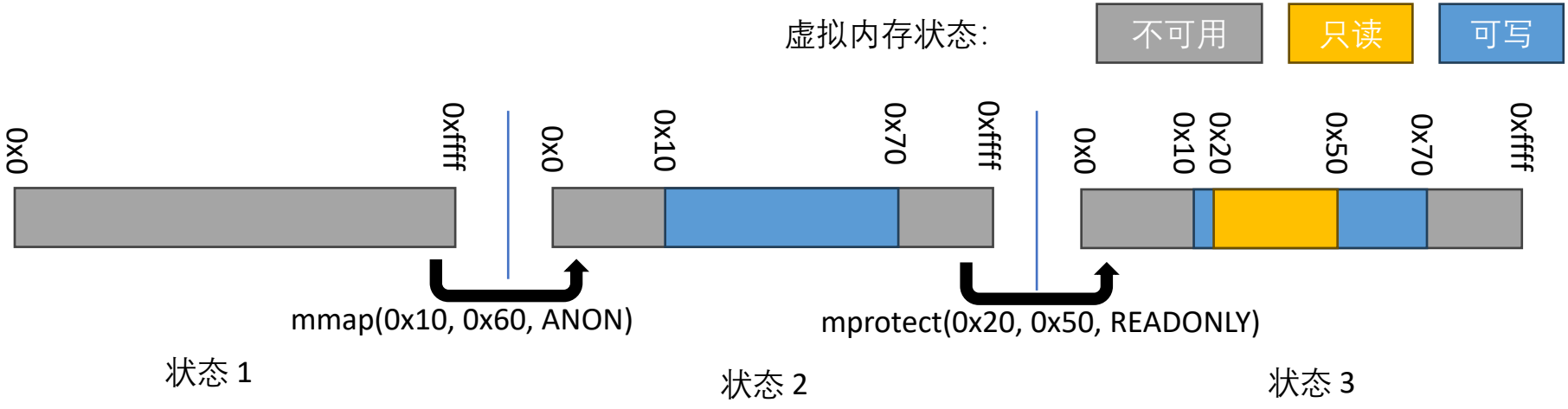
应用程序可以通过 POSIX 系统调用操作其虚拟内存的状态

- **mmap(ANON)：**
  将一段内存地址范围设为可用
- **mmap(FILE)：**
  将一段内存地址范围映射至文件
  读操作可获得文件内容，
  写操作将写入文件
- **munmap：**
  将一段内存地址范围设为不可用
- **mprotect：**
  改变一段地址范围的保护属性



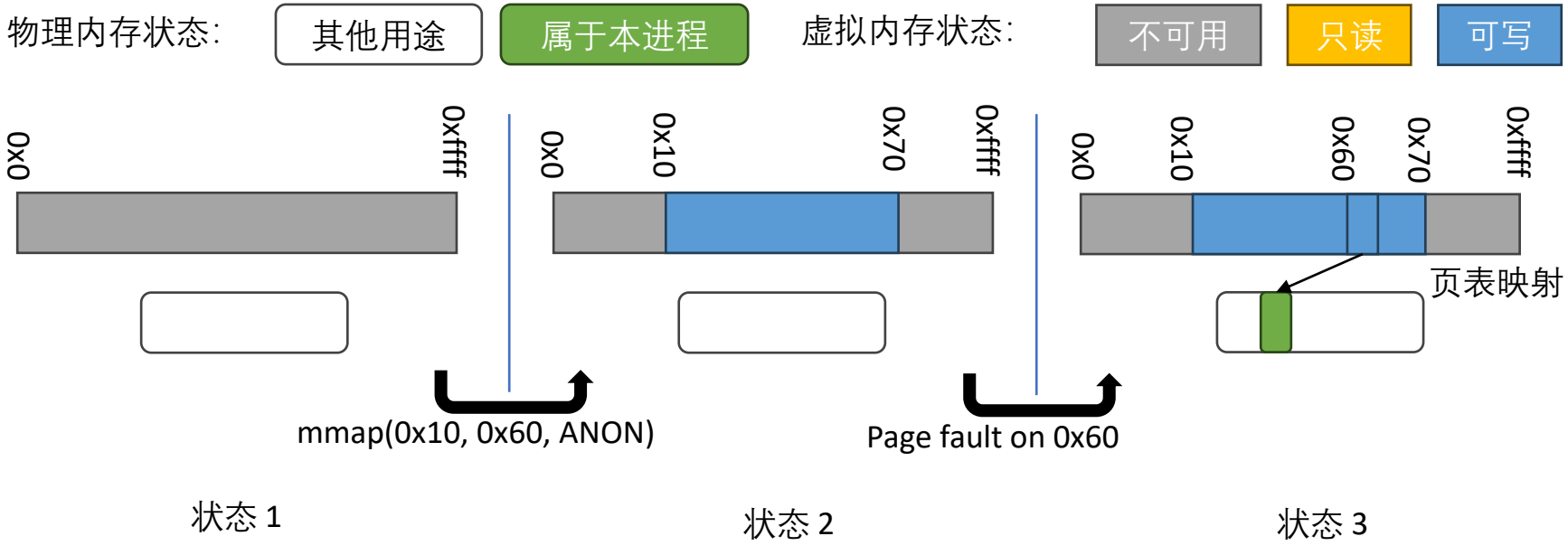
1.问题背景-虚拟内存管理（应用视角）

应用程序通过**mmap**设定可访问的内存区域、通过**mprotect**改变访问模式



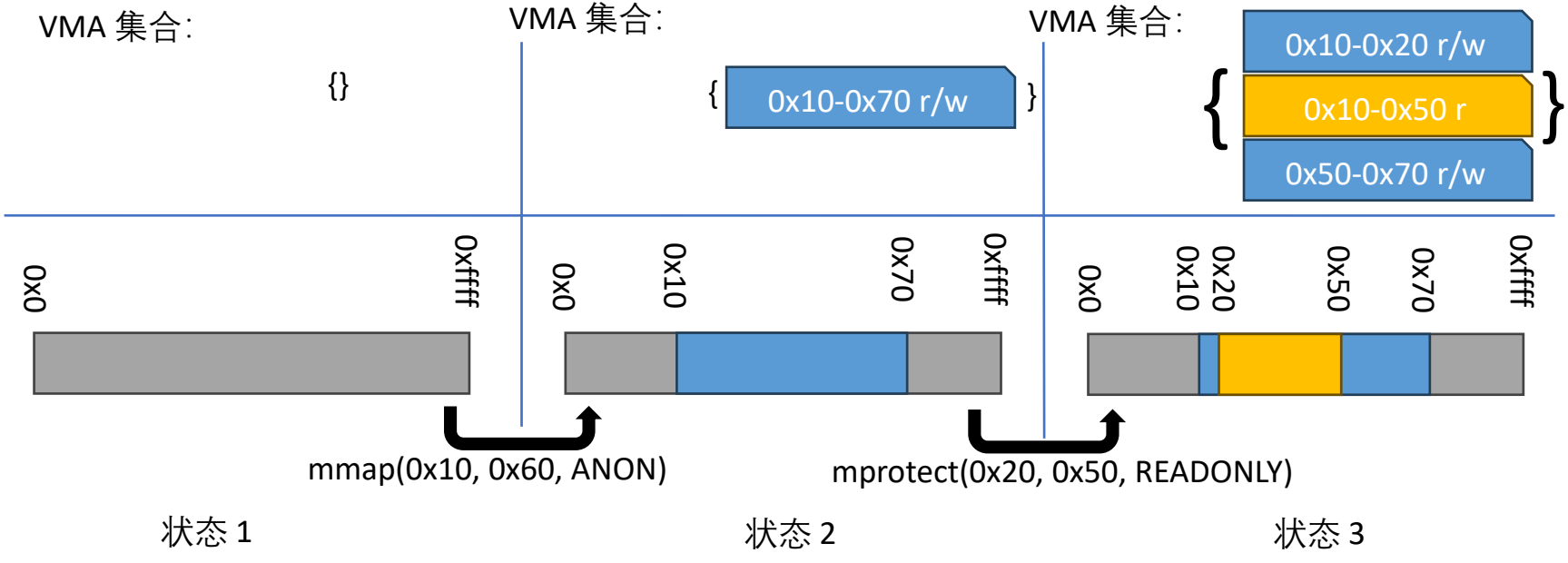
1.问题背景-按需分页（应用视角）

按需分页（On-demand-paging）技术可以使内核无需为 **mmap** 的内存区域真正分配内存资源，当用户访问 mmap区域的地址，触发缺页中断，再真正分配物理内存并通过页表映射



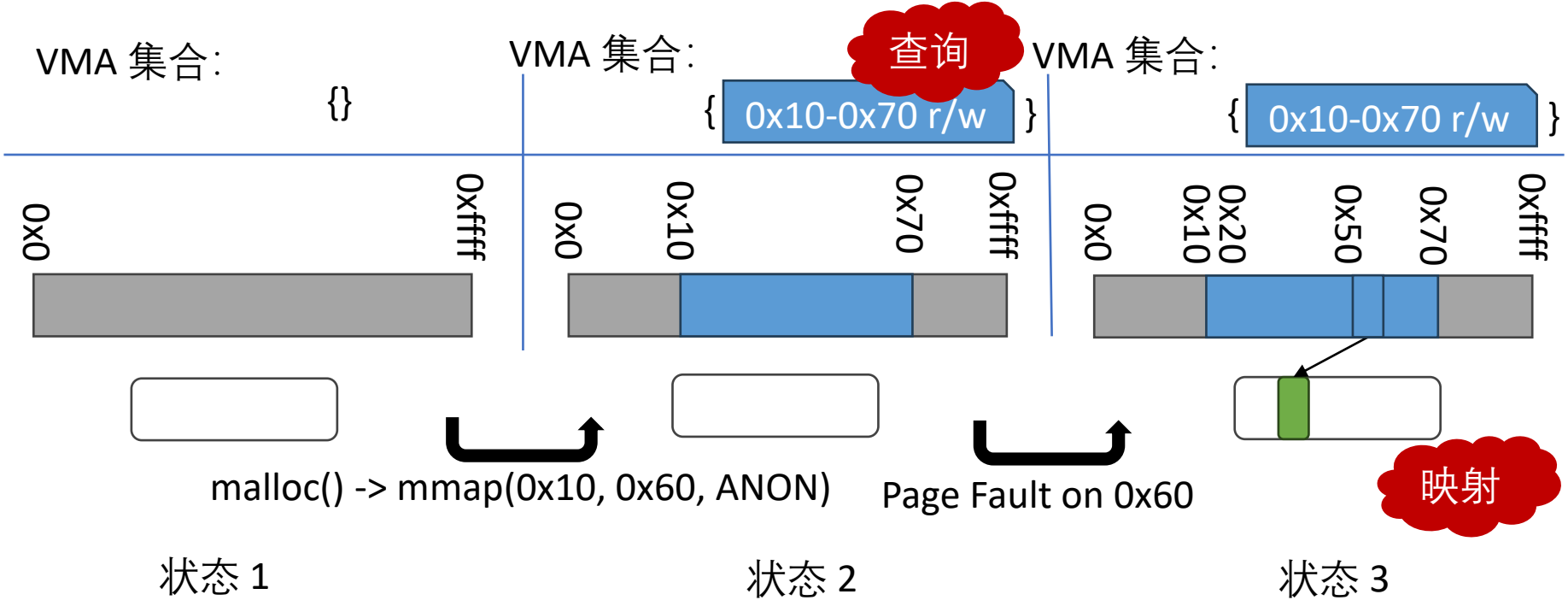
1.问题背景-虚拟内存区域（VMA）

BSD 4.4 (1995) 引入 vm_map_entry 结构记录用户 mmap 的区域；Linux 沿用了这一机制，称为 VMA (virtual memory area)。一段连续区域用一个 VMA 结构表示。



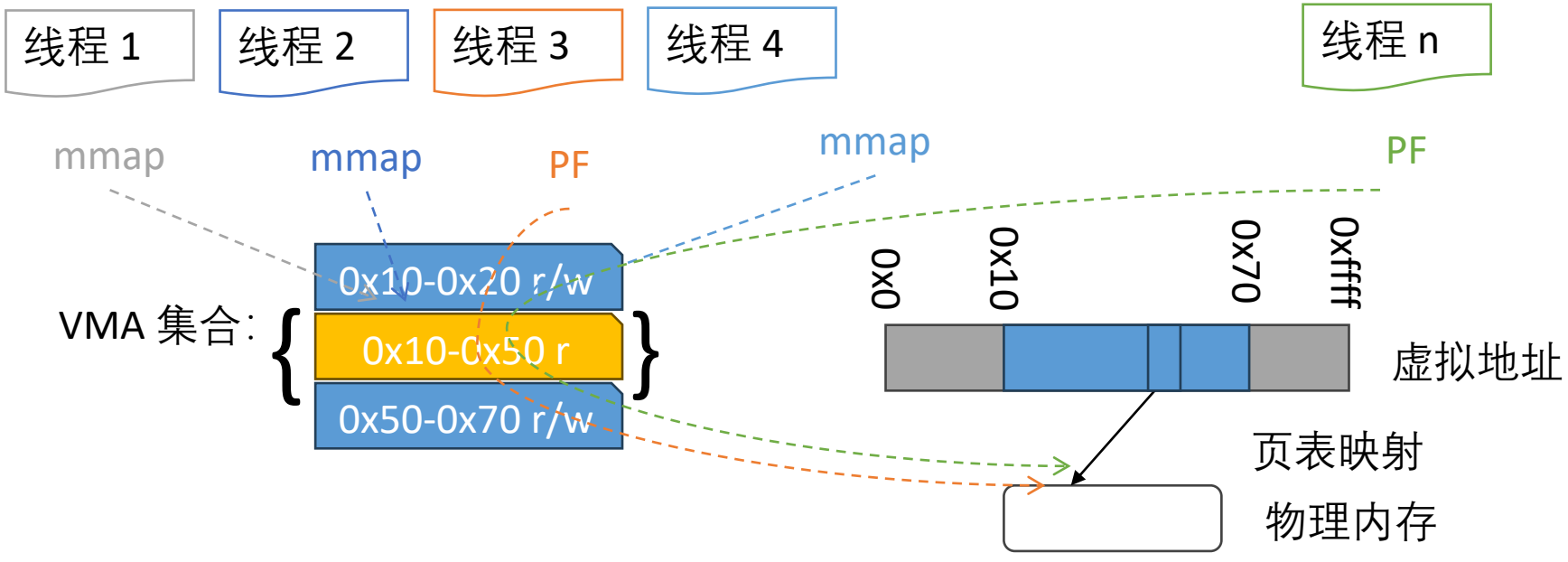
1.问题背景-基于VMA维护页表

- 在首次访问mmap区域中的地址时触发 page fault
- Page fault 处理程序需要通过查询 VMA 获得访问属性来更新页表映射物理页面



1.问题背景-多核可扩展性问题

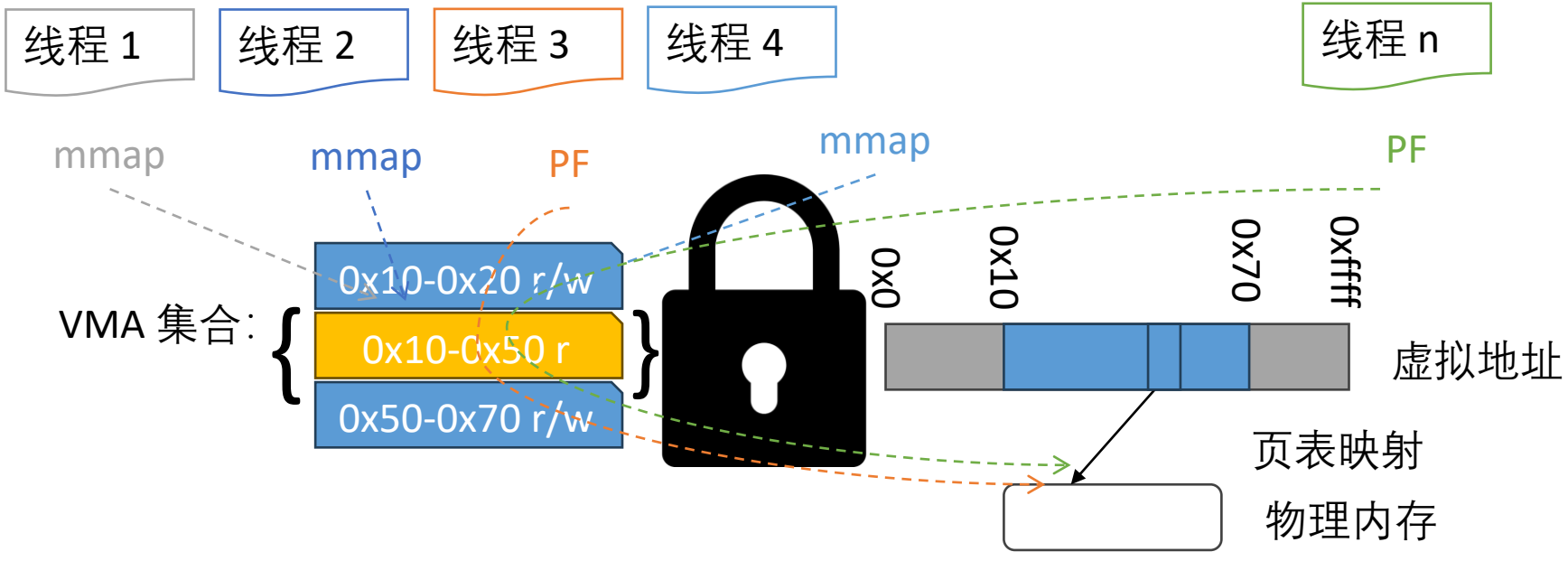
在现代多核/多线程场景下，需要考虑 mmap/PF 并发的问题



1.问题背景-多核可扩展性问题

Linux 5.x：所有需要读/写 VMA 集合或页表的操作都需要拿一个锁

可扩展性问题严重：多核并行不会比单核串行更快!

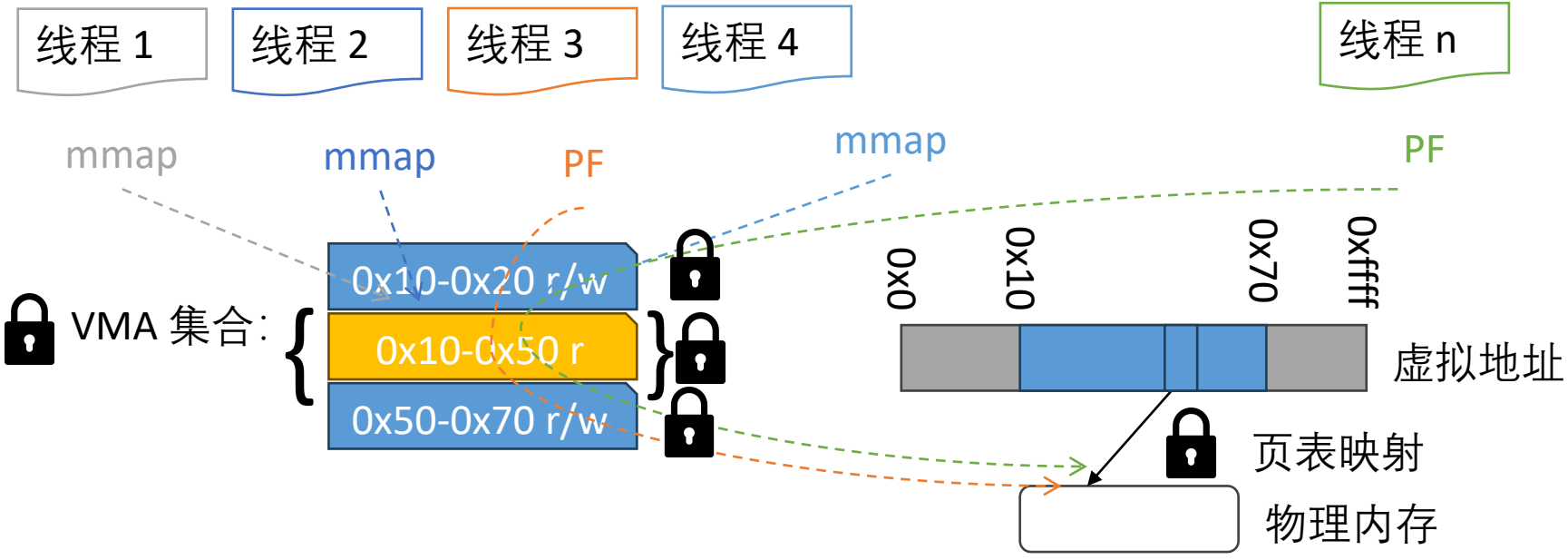


1.问题背景-多核可扩展性问题

Linux 6.3+ 解决扩展性的方法：
- VMA 集合有对应的读写锁、
- 每一个 VMA 有对应的读写锁、
- 页表有对应的读写锁

不同的操作该如何拿锁？拿哪个锁？

mmap 拿集合写锁，PF 拿集合读锁和 VMA 读锁；如果不需要回收/展开页表只需要拿住任意 VMA 锁；文件映射缩短可以不拿 VMA 锁；回收页表需要拿所有锁……



1.问题背景-多核扩展导致并发安全问题

在Linux 6.3+内核操作中如何拿锁？拿哪个锁？规则非常复杂!

https://www.kernel.org/doc/html/next/mm/process_addrs.html

Examining all valid lock states:

mmap lock	VMA lock	rmap lock	Stable?	Read?	Write most?	Write all?
-	-	-	N	N	N	N
-	R	-	Y	Y	N	N
-	-	R/W	Y	Y	N	N
R/W	-/R	-/R/W	Y	Y	N	N
W	W	-/R	Y	Y	Y	N
W	W	W	Y	Y	Y	Y

Warning:

While it's possible to obtain a VMA lock while holding an mmap read lock, attempting to do the reverse is invalid as it can result in deadlock - if another task already holds an mmap write lock and attempts to acquire a VMA write lock that will deadlock on the VMA read lock.

All of these locks behave as read/write semaphores in practice, so you can obtain either a read or a write lock for each of these.

繁多的状态组合

随处可见的边界情况

```
->i_mmap_rwsem                  (truncate_pagecache)
  ->private_lock                (__free_pte->block_dirty_folio)
    ->swap_lock                 (exclusive_swap_page, others)
      ->i_pages lock

->i_rwsem
  ->invalidate_lock             (acquired by fs in truncate path)
    ->i_mmap_rwsem              (truncate->unmap_mapping_range)

                                lock or pte_lock    (various, mainly in memory.c)
                                                    (arch-dependent flush_dcache_mmap_lock)

                                                    (filemap_fault)
    ->lock_page                 (filemap_fault, access_process_vm)

->i_rwsem                       (generic_perform_write)
  ->mmap_lock                   (fault_in_readable->do_page_fault)

bdi->wb.list_lock
  sb_lock                       (fs/fs-writeback.c)
  ->i_pages lock                (__sync_single_inode)

->i_mmap_rwsem
  ->anon_vma.lock               (vma_merge)

->i_pages lock                  (try_to_unmap_one)
  ->lruvec->lru_lock            (follow_page_mask->mark_page_accessed)
  ->lruvec->lru_lock            (check_pte_range->folio_isolate_lru)
  ->private_lock                (folio_remove_rmap_pte->set_page_dirty)
  ->i_pages lock                (folio_remove_rmap_pte->set_page_dirty)
  bdi.wb->list_lock             (folio_remove_rmap_pte->set_page_dirty)
  ->inode->i_lock               (folio_remove_rmap_pte->set_page_dirty)
  bdi.wb->list_lock             (zap_pte_range->set_page_dirty)
  ->inode->i_lock               (zap_pte_range->set_page_dirty)
  ->private_lock                (zap_pte_range->block_dirty_folio)
```

When **installing** page table entries, the mmap or VMA lock must be held to keep the VMA stable. We explore why this is in the page table locking details section below.

Warning:

Page tables are normally only traversed in regions covered by VMAs. If you want to traverse page tables in areas that might not be covered by VMAs, heavier locking is required. See `walk_page_range_novma()` for details.

Freeing page tables is an entirely internal memory management operation and has special requirements (see the page freeing section below for more details).

Warning:

When **freeing** page tables, it must not be possible for VMAs containing the ranges those page tables map to be accessible via the reverse mapping.

The `free_pgtables()` function removes the relevant VMAs from the reverse mappings, but no other VMAs can be permitted to be accessible and span the specified range.

Locking Implementation Details

Warning:

Locking rules for PTE-level page tables are significantly more lax than those from locking rules for page tables at other levels.

Page table locking details

In addition to the locks described in the terminology section above, we have additional locks dedicated to page tables:

- **Higher level page table locks** - Higher level page tables, that is PGD, P4D and PUD each make use of the process address space granularity `mm->page_table_lock` lock when modified.

Locking rules

We establish basic locking rules when interacting with page tables:

- When changing a page table entry the page table lock for that page table **must** be held, except if you can safely assume nobody can access the page tables concurrently (such as on invocation of `free_pgtables()`).
- Reads from and writes to page table entries must be appropriately atomic. See the section on atomicity below for details.
- Populating previously empty entries requires that the mmap or VMA locks are held (read or write), doing so with only rmap locks would be dangerous (see the warning below).
- As mentioned previously, zapping can be performed while simply keeping the VMA stable, that is holding any one of the mmap, VMA or rmap locks.

either kept within the folios describing `LIT_PTLOCKS` is set. The PMD spin lock if a 32-bit system) and carefully locked

Warning:

Populating previously empty entries is dangerous as, when unmapping VMAs, `vms_clear_ptes()` has a window of time between zapping (via `unmap_vmas()`) and freeing page tables (via `free_pgtables()`), where the VMA is still visible in the rmap tree. `free_pgtables()` assumes that the zap has already been performed and removes PTEs unconditionally (along with all other page tables in the freed range), so installing new PTE entries could leak memory and also cause other unexpected and dangerous behaviour.

...

1.问题背景-多核扩展导致并发安全问题

复杂的锁协议导致并发安全性问题：

CVE-2023-3269: A race between stack expansion and maple tree walking leads to UAF
CVE-2023-4611: A race between mbind() and VMA-locked page fault
CVE-2024-1312: Race condition leads to UAF during VMA lock in lock_vma_under_rcu
CVE-2024-27022: Race condition leads to accessing uninitialized VMA
CVE-2024-47676: UAF of VMA in HugeTLB fault pathway
CVE-2024-50066: A race in mremap() and move_page_tables()
…

1.问题背景-多核可扩展性问题解决了吗?

Linux 6.3+：VMA 集合有对应的锁、每一个 VMA 有对应的读写锁、页表有对应的锁

复杂的锁协议导致并发安全性问题（假设所有的安全问题都解决了）

用了这么细粒度的锁，多核可扩展性真的解决了吗?

…没有！

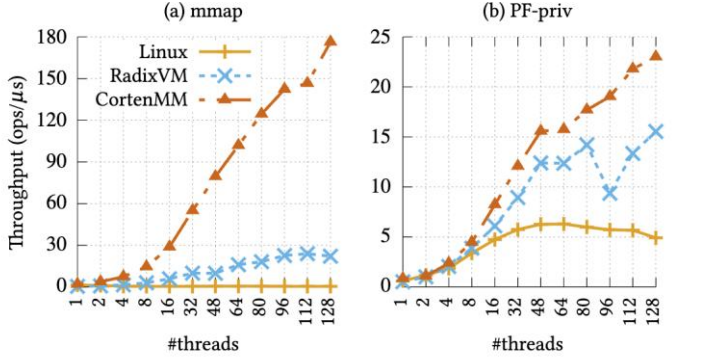


Figure 1: The multicore performance of two other memory systems and CortenMM with two basic memory system operations: (a) each thread `mmap()` a private region. (b) each thread page faults on a private region.

Linux
mmap 仍需全局锁
PF 仍需可扩展性差的读锁

星绽能做到413/4.7倍提升

1. 问题背景与研究动机-总结

对于虚拟内存管理问题：

多核可扩展性 和 并发安全性

均是还未解决的问题！

Linux Kernel中采用的添砖加瓦式的解决路线

通过大量细粒度锁实现多核可扩展性

反而让并发安全性问题变得更糟！

星绽要实现**多核可扩展性强**、**并发安全性高**的虚拟内存管理机制！

2. 系统设计与实现-根因分析与解决思路

传统操作系统设计同时使用 VMA 和页表对地址空间进行抽象（两层抽象）

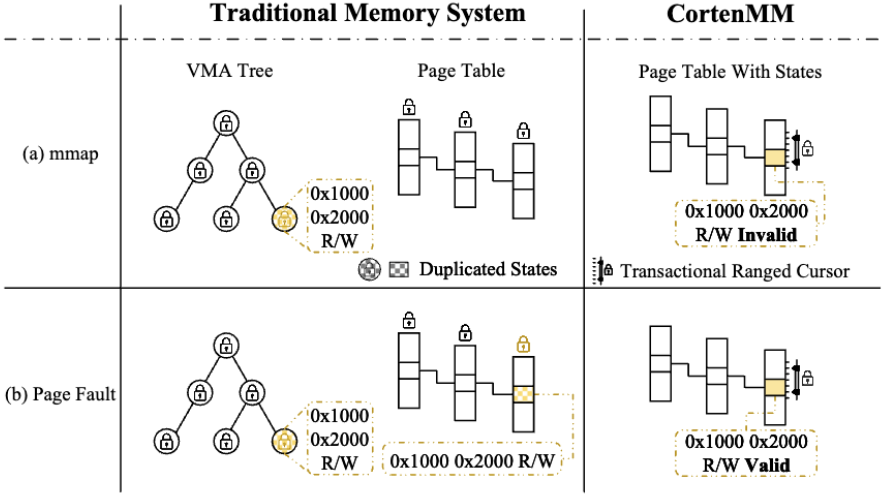
存在重复的状态、存在不合法的状态（如果用不同锁保护，则可能有不合法的瞬间）

星绽 MM 将状态合二为一，仅有虚拟页的状态，记录在页表项（以及附属记录）中

VMA 状态：
是否映射/可写/文件…

页表状态：
是否映射/可写/地址…

不合法状态：
VMA 未映射/PT 已映射



页表状态：
已标记可映射/
已映射/
已换出/
写时复制

2. 系统设计与实现-状态编码

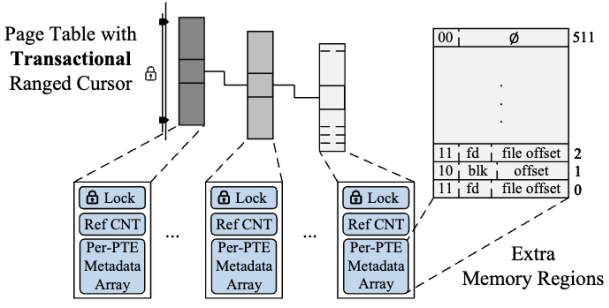
页表项中记录虚拟页的（核心）状态：已标记可映射/已映射/已换出/写时复制

已标记可映射对应存在 VMA 但没有页表映射的状态
- 已标记可映射分为：匿名映射/文件映射

已标记可映射的文件映射需要记录：
- 映射的文件 FD 和映射偏移量

这两个信息无法记录在页表项中！

对于需要文件映射的情况，为对应的页表页分配一个页大小的元数据数组，存储页表项的额外信息

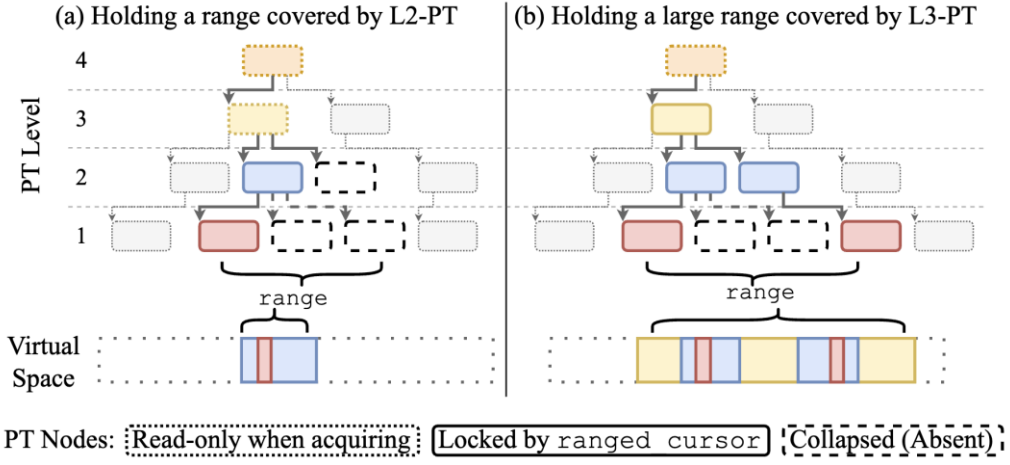


2. 系统设计与实现-多核可扩展的锁协议

我们通过一个细粒度的页表锁协议来实现多核可扩展的内存操作

对于每一个页表页，我们使用一个锁保护。修改该页表页需要获得该页表的锁。

采用二阶段锁协议（two-phase-locking）：首先锁住所有需要操作的节点，接着操作这些节点，最后放掉所有节点。



末级页表对应一个地址空间上的范围；高层页表对应它所有孩子的范围

规定：锁住一个能够覆盖范围 R 的页表及其子树中所有与 R 相交的页表，即锁住了地址范围 R；所有后续在范围 R 上的操作均被这**一组锁**保护

2. 系统设计与实现-事务性保证

所有内存系统调用均可视为在虚拟地址空间范围上的事务性操作：
锁住一个范围 R，操作范围 R 上虚拟页的属性，释放范围 R。

- 利用 Rust 语言的类型系统，可以将锁的持有时间与一个类型（RCursor）的生命周期绑定，并强制所有地址范围上的操作都需要通过一个 RCursor 实例，保证 RCursor 的原子性。

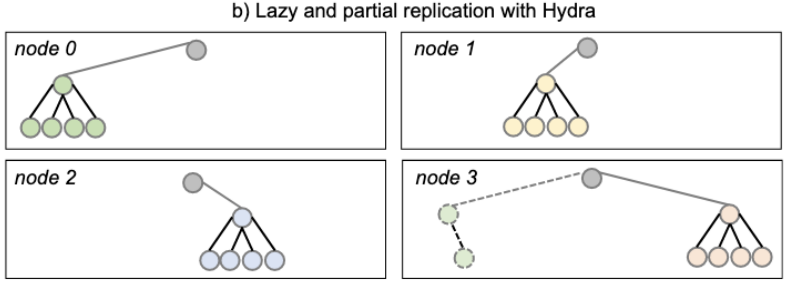
- 为了确保事务性，每次内存管理操作必须由一个 RCursor 完成，RCursor 的操作不可依赖其他 RCursor 读取的状态。

- 使用 Rust 的零尺寸对象模拟一次性权限，保证仅会为每次系统调用/缺页异常/重试产生一个一次性权限，用于获取 RCursor

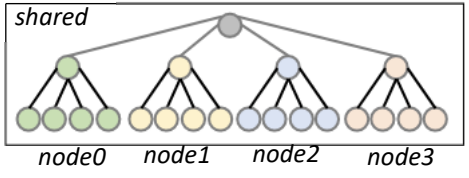
```rust
fn page_fault_handler(
    faulting_addr: Vaddr, reason: PFReason, t: Token,
)-> Result<()> {
    // Consume the token and create an RCursor.
    let mut rcursor = this_addr_space!(t).lock(fault_range)?;
    match rcursor.query(faulting_addr) {
        // The `Status` is mapped by `RCursor::map` when `mmap`.
        Status::VirtuallyAllocatedAnon(perm) => {
            // ... SEGFAULT checks omitted.
            rcursor.map(faulting_addr, alloc_zeroed(), perm);
        }
        Status::Mapped(&mut page, &mut perm) => {
            // Use the first unused bit as "copy-on-write".
            if reason.is_write() && perm.contains(COW) {
                // No need to COW if the page is not shared.
                if page.meta().map_count() == 1 {
                    perm -= COW; perm |= WRITE;
                    return Ok(()); // Automatic TLB shootdown.
                }
                rcursor.map(alloc_copied(&page), *perm | WRITE);
            } else { return Err(SEGFAULT); }
        }
        Status::Invalid => { return Err(SEGFAULT); }
        // ... other states (e.g., swapped, file-backed) omitted
    } // The function is atomic under `rcursor`.
}
```

2. 系统设计与实现-用户地址分配优化

传统工作通过 NUMA 复制（Node Replication）来增加页表操作可扩展性[1]



我们发现，可以通过在 mmap 阶段根据不同核分配分离的地址，间接达成“不同核访问私有的页表”这一目的



[1] B. Gao, Q. Kang, H.-W. Tee, K. T. N. Chu, A. Sanaee, and D. Jevdjic, “Scalable and Effective Page-table and {TLB} management on {NUMA} Systems,” presented at the 2024 USENIX Annual Technical Conference (USENIX ATC 24), 2024, pp. 445–461.

2. 系统设计与实现-星绽内核中的实现

星绽内核最初的版本借鉴 Linux 的双层抽象，但也如 Linux 发现了诸多问题

星绽 MM 分为两层五个模块，框架层包括物理页元数据和页表管理，其实现包含unsafe Rust 代码，对外封装安全接口。

大部分功能（如系统调用、用户空间管理、物理内存管理等）均强制由安全代码实现。

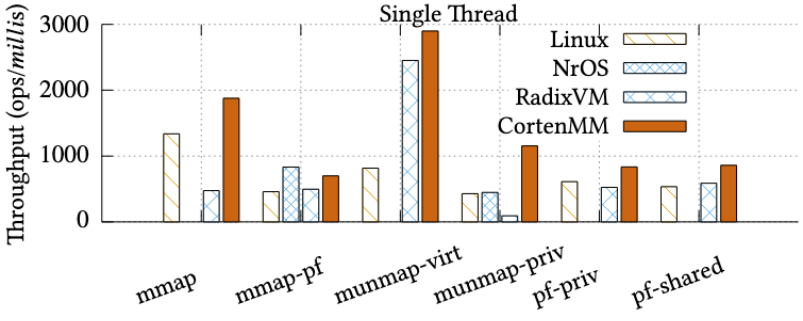
	PT	AddrSpace	Phys Page	Syscall	Userspace
LoC	1527	271	777	328	2904
unsafe	91	0	31	0	0

3. 实验评估 -微基准测试程序与单核性能

- **微基准测试**：编写了六个微基准测试程序，用于测试星绽 MM 的**系统调用/操作**性能

Name	Description
`mmap`	Creates an anonymous VMA.
`mmap-PF`	Creates a single-page VMA and then faults on it.
`unmap-virt`	Unmaps a virtually allocated VMA in a private region.
`unmap-priv`	Unmaps a private region in a shared VMA.
`PF-priv`	Page faults on a private region in a shared VMA.
`PF-shared`	Random accesses in a shared VMA.

- **单核性能：**微基准测试实验表明，单层抽象设计不会带来额外开销，并因为维护的数据结构数量减少，能相对获得更高的性能。



3. 实验评估 - 多核可扩展性/微基准测试

多核性能评估分为基准测试和应用测试：

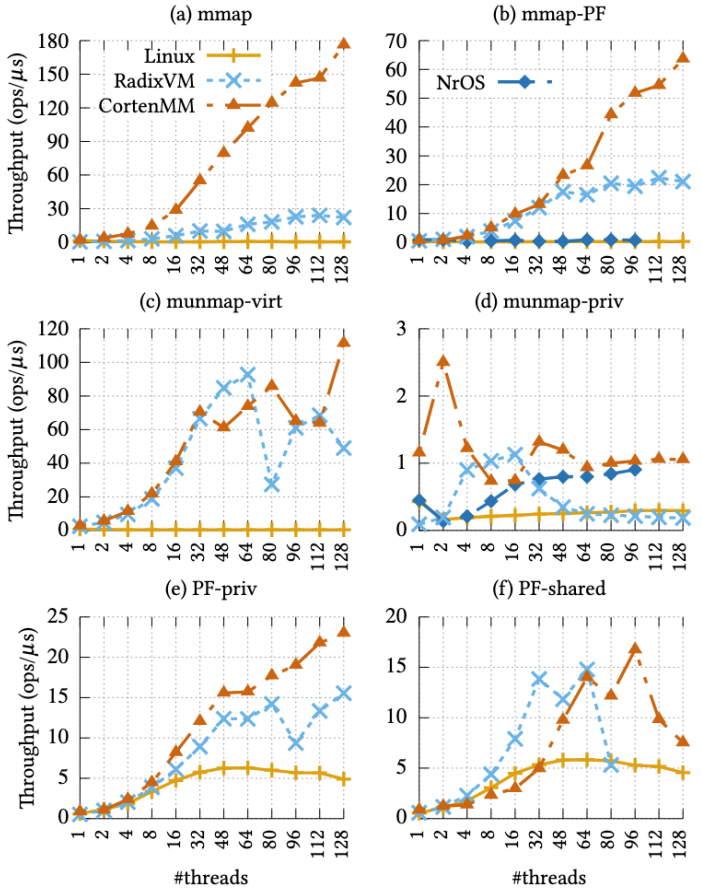
基准测试使用同样的六个微基准测试程序

Name	Description
mmap	Creates an anonymous VMA.
mmap-PF	Creates a single-page VMA and then faults on it.
unmap-virt	Unmaps a virtually allocated VMA in a private region.
unmap-priv	Unmaps a private region in a shared VMA.
PF-priv	Page faults on a private region in a shared VMA.
PF-shared	Random accesses in a shared VMA.

参与基准测试对照比较技术的包括：

- NrOS（基于 NUMA 复制）的内存系统、
- RadixVM（基于基数树的 VMA 管理）
- Linux（6.xx）

- 星绽相比 Linux 在 mmap 上获得最多410倍的提升
- 在操作的内存区域在大粒度上不重叠的情况下，星绽 MM 具有很好地多核扩展性
- 在小粒度上重叠不可避免的情况下（如 munmap / PF-shared），星绽 MM 的扩展性至少与Linux相当



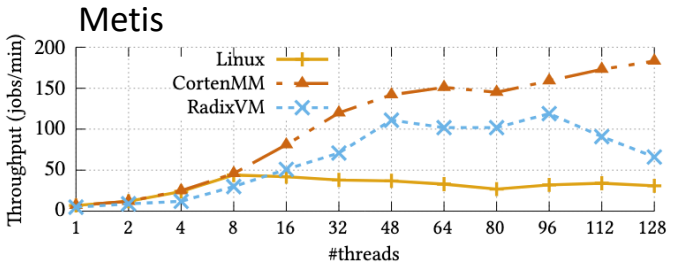
3. 实验评估 - 多核可扩展性/应用测试

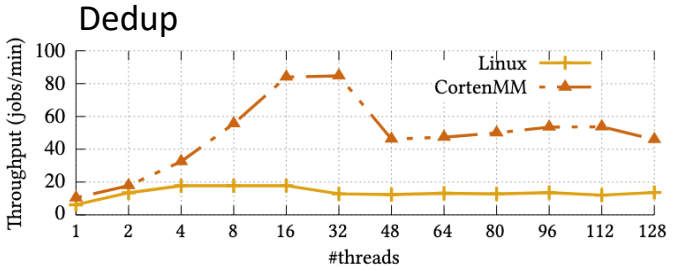
在应用测试中选取可三个典型应用：

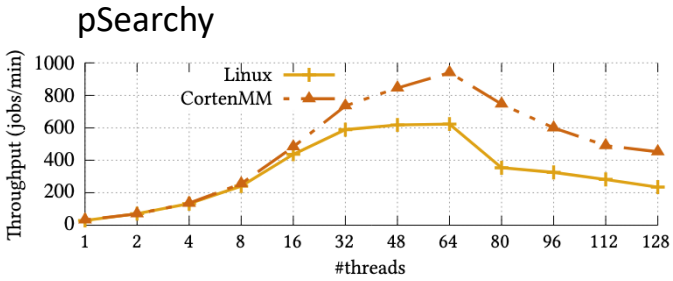
- Metis (MOSBENCH)，Map-reduce 框架
- Dedup (PARSEC)，流水线数据处理程序
- pSearchy (MOSBENCH)，多核文件索引程序

- Metis 表现为在内核的内存子系统中会发生严重的争用情况。相对于Linux，星绽MM在Metis 上获得了最高5.88 倍性能提升。

- 受限于 Dedup 和 pSearchy 应用本身的原因（包括用户空间内存分配器中的竞争现象），扩展到 32/64 核以上性能反而下降，但星绽MM仍比Linux表现更好。

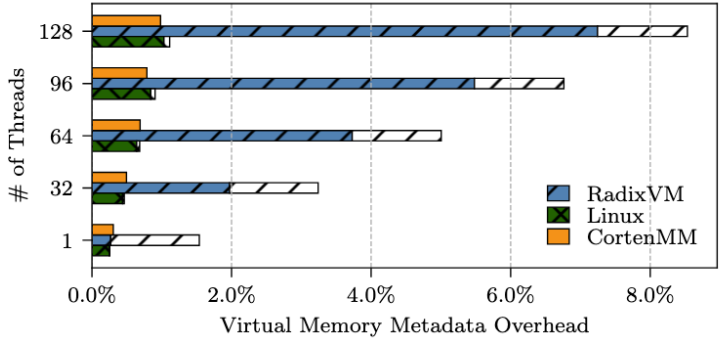






3. 实验评估 -内存开销

- 星绽 MM 在带来更好性能的同时并不带来额外的内存开销
- 相比将页表复制到所有核的 RadixVM、将页表重复到 NUMA 节点上的 NrOS，星绽 MM 没有重复页表带来的开销。
- 得益于按需分配的页表项元数据存储，星绽 MM 所占用的页表内存和 Linux 的相近



4. 总结与讨论 -兼容性和未来工作

- 星绽 MM 对不同体系结构、新型硬件特性（如 MPK）的兼容较为容易（前提是目标体系结构使用多层页表进行虚拟内存管理），相比与Linux，星绽MM在兼容实现中无需修改 VMA 抽象。

- 星绽 MM 对内存子系统的基础功能：如按需分页、文件映射、文件换出等功能，在实现没有障碍，且预期能更好地提升实现的安全性。

- 可能存在的问题包括：
  - 由于在 星绽 MM 不维护VMA数据结构，对于依赖 VMA 的功能实现（如 rmap），星绽 MM 无法直接借鉴 Linux 的实现方式
  - 对于直接依赖 VMA 数据结构的监控功能和相关应用，可能存在兼容问题
  - 在极端应用场景下，所有线程需要在相邻的虚拟地址上发生缺页异常时，星绽MM的表现有可能不如 Linux的内存子系统

北京大学
PEKING UNIVERSITY

感谢关注