

PR #49675 完整报告

vllm-project/vllm

[Bugfix][Core] Stop zero-progress preemption cascades for deferred KV frees

合并时间: 2026-09-11 08:40

原文链接: <http://prhub.com.cn/vllm-project/vllm/pull/49675>

执行摘要

- 一句话: 延迟释放栅栏未推进时停止抢占重试, 消除零进展级联
- 推荐动作: 值得精读, 尤其适合负责 v1 调度器、KV connector 与异步调度的工程师。建议重点看三处: (1) 谓词抽取的位置与复用方式——把“释放是否被延迟”的判断收敛成 `_request_blocks_can_be_freed()`, 使释放路径与抢占路径不会分叉; (2) `break` 被放在所有 `mutation` (队列删除、`budget` 回滚、状态迁移、抢占计数) 之前, 这是“零进展不产生副作用”的关键, 也是后续任何零增量修复都可以照搬的模板; (3) 测试设计——用一次性故障注入 + 参数化 FCFS/PRIORITY 复现“栅栏未推进 → 零抢占”与“栅栏推进 → 恢复抢占”两阶段, 并用 PRIORITY 的 `failed_call=2` 覆盖 #49206 的 `victim_index < req_index`。同时注意作者显式接受的 `stop-and-wait` 权衡与未来策略 RFC, 不要在未读该权衡的前提下把它当作通用性能优化。

功能与动机

issue #49674 把这一定位为「scheduler progress/control-flow defect」, 触发条件有四个: `max_concurrent_batches > 1`、配置了 `consumer` 型 KV connector (含原生 CPU KV offload)、运行中请求的 `allocate_slots()` 失败、且策略选出的 `victim` 满足 `last_sched_seq > processed_step_seq`。此时「每次抢占都能立即扩大空闲 block 池」的隐含假设不再成立: 请求状态变了, 却没有一个 block 变得可用, 一次分配失败会沿着 `running` 队列级联到多个 `victim`。issue 明确要求的最小改动是「Check the existing policy-selected victim before running-queue removal, budget rollback, or request-status mutation」, 同时保留 #45357 引入的 `deferred-free` 安全栅栏与 #49206 的 PRIORITY 游标修正, 并且在延迟释放关闭时保持惰性。PR body 也强调其非目标: 不改变 FCFS/PRIORITY 的 `victim` 选择策略, 不处理 `shared reference` 或 `connector pin` 造成的其他零增量原因。

实现拆解

1. 抽出共享谓词, 统一“能否立即回收”的语义: 在 `vllm/v1/core/sched/scheduler.py` 新增 `_request_blocks_can_be_freed(request)`, 把原本内联在 `_free_request_blocks()` 中的条件 `not self.defer_block_free or request.last_sched_seq <= self.processed_step_seq` 提取为单一谓词, 并让 `_free_request_blocks()` 改为调用它。这样“释放是否需要延迟”只有一处定义, 抢占路径与释放路径不会出现判定漂移。
2. 抢占 `victim` 选择改为“先窥视、后变更”: `schedule()` 里运行中请求的分配重试中, PRIORITY 仍用 `max(key=(priority, arrival_time))` 选出 `victim`, 但先不执行 `del`

self.running[...]; FCFS 由 self.running.pop() 改为只读的 self.running[-1]。随后插入 ifnot self._request_blocks_can_be_freed(preempted_req): break, 被挡时直接退出本轮分配重试, 不删除队列元素、不回滚 token/budget 预算、不改请求状态、不计入抢占统计, 也不会把触发请求放进本步 SchedulerOutput。

3. 保留既有策略与游标修正: 真正执行抢占的分支保持原样 —— PRIORITY 的 victim_index = self.running.index(...), del self.running[victim_index] 与 if victim_index < req_index: req_index -= 1 (#49206 的修正) 只在通过栅栏检查后运行; FCFS 分支也在检查通过后才 self.running.pop()。因此 victim 顺序、FCFS/PRIORITY 语义与 #49206 的行为均不变。
4. 补齐测试基础设施与确定性回归: tests/v1/core/utils.py 的 create_scheduler() 新增 scheduling_policy: SchedulerPolicy = "fcfs" 参数并透传到 SchedulerConfig(policy=...) ; tests/v1/core/test_deferred_block_free.py 新增 _fail_one_allocation() (只让第 N 次 allocate_slots 返回 None, 其余调用透传真分配器) 和参数化用例 test_allocation_retry_waits_for_fence_then_succeeds (FCFS 失败点 1、PRIORITY 失败点 2), 断言“栅栏未推进时零抢占、触发请求不被调度”与“处理输出推进栅栏后, 抢占与分配恢复且触发请求被调度”。
5. 异步调度侧的行为覆盖与门禁绕过: tests/v1/core/test_async_scheduler.py 的 test_kv_pressure_preempt_mid_handoff 从单维 kv_role 扩展为 (kv_role, defer_free) 三维 (producer/False、consumer/False、consumer/True), 并用手动置位 scheduler.defer_block_free 覆盖 CPU 上被强制关闭的生产门禁 (门禁在真实环境依赖异步调度或 PP 提供的重叠 batch)。
6. 配套与兼容性: 没有 schema、配置默认值、部署脚本或模型输出语义的改动; 门禁关闭时新谓词恒为 True, 行为与修改前完全一致, 属于纯 Python 调度器侧的窄修复。

关键文件:

- vllm/v1/core/sched/scheduler.py (模块 调度器; 类别 source; 类型 core-logic; 符号 _request_blocks_can_be_freed, _free_request_blocks, schedule): 唯一的源码改动文件: 新增 _request_blocks_can_be_freed() 谓词并在分配重试中于任何 running 队列变更前检查策略选中的 victim, 被 deferred-free 栅栏挡住时直接停止本轮重试 (停在 break, 不改状态、不回滚预算); 同时让 _free_request_blocks() 复用同一谓词, 保证释放与抢占语义一致。
- tests/v1/core/test_deferred_block_free.py (模块 调度器测试; 类别 test; 类型 test-coverage; 符号 _create_deferring_scheduler, _fail_one_allocation, allocate, test_allocation_retry_waits_for_fence_then_succeeds): 新增 _fail_one_allocation() 一次性分配失败注入器与参数化用例 test_allocation_retry_waits_for_fence_then_succeeds (FCFS 失败点 1 / PRIORITY 失败点 2), 用真实 allocator 与真实 update_from_output() 验证“栅栏未推进时零抢占”与“栅栏推进后抢占恢复”; _create_deferring_scheduler() 增加 scheduling_policy 形参以覆盖两种策略。
- tests/v1/core/test_async_scheduler.py (模块 异步调度测试; 类别 test; 类型 test-coverage; 符号 test_kv_pressure_preempt_mid_handoff): test_kv_pressure_preempt_mid_handoff 由单维 kv_role 扩展为 (kv_role, defer_free), 新增 consumer + fenced free 分支: 断言 victim 保持 RUNNING、不被抢占, 随后从

in-flight 输出正常完成，直接覆盖本 PR 改变的行为。

- tests/v1/core/utils.py (模块 测试工具; 类别 test; 类型 test-coverage; 符号 create_scheduler) : create_scheduler() 新增 scheduling_policy: SchedulerPolicy = "fcfs" 形参并透传到 SchedulerConfig(policy=...), 使 FCFS/PRIORITY 可以在构造阶段声明, 而不必像早期提交那样在构造后手工改写调度器状态。

关键符号: _request_blocks_can_be_freed, _free_request_blocks, Scheduler.schedule, _preempt_request, _drain_deferred_frees, _create_deferring_scheduler, _fail_one_allocation, test_allocation_retry_waits_for_fence_then_succeeds, test_kv_pressure_preempt_mid_handoff, create_scheduler

关键源码片段

vllm/v1/core/sched/scheduler.py

唯一的源码改动文件: 新增 `_request_blocks_can_be_freed()` 谓词并在分配重试中于任何 running 队列变更前检查策略选中的 victim, 被 deferred-free 栅栏挡住时直接停止本轮重试 (停在 `break`, 不改状态、不回滚预算); 同时让 `_free_request_blocks()` 复用同一谓词, 保证释放与抢占语义一致。

vllm/v1/core/sched/scheduler.py 中新增的谓词与释放路径的复用 (安全性语义完全沿用 #45357 的栅栏):

```
def _request_blocks_can_be_freed(self, request: Request) -> bool:
    # 谓词含义: 这些 block 现在能不能立刻还给 block pool。
    # 关闭延迟释放时恒为 True, 对既有部署完全惰性;
    # 否则只有当请求最后一次被调度的 step 已经被处理
    # (last_sched_seq <= processed_step_seq) 时, 才说明没有 in-flight 的 GPU 写
    # 会落到这些 block 上。释放路径与抢占路径共用这一个判定, 避免语义分叉。
    return not self.defer_block_free or (
        request.last_sched_seq <= self.processed_step_seq
    )

def _free_request_blocks(self, request: Request):
    """Free the request's KV blocks, deferring the return to the block
    pool when an in-flight GPU step may still write them.
    """
    if self._request_blocks_can_be_freed(request):
        # 没有 in-flight 写 (普通 finish 请求走这里): 走原有同步释放路径。
        self.kv_cache_manager.free(request)
        return
    # 否则先摘掉 KV 管理器记账, 保证请求可以立刻被安全重调度;
    # block 本体挂进 deferred_frees, 等 _drain_deferred_frees()
    # 看到 processed_step_seq 越过 fence 后再归还 block pool。
    blocks = self.kv_cache_manager.pop_blocks_for_free(request)
    if blocks:
        self.deferred_frees.append((self.sched_step_seq, blocks))
```

分配重试中的核心修复：检查发生在一切 mutation 之前。

```
# schedule(): running 请求的 KV 分配重试循环
while True:
    new_blocks = self.kv_cache_manager.allocate_slots(
        request,
        num_new_tokens,
        num_lookahead_tokens=self.num_lookahead_tokens,
    )

    if new_blocks is not None:
        break # 分配成功，按正常路径继续调度该请求

    # 分配失败：先按现行策略「选出」victim，此刻不要动 running 队列。
    # PRIORITY 取 (priority, arrival_time) 最低者；
    # FCFS 由原来的 self.running.pop() 改成只读窥视 self.running[-1]。
    if self.policy == SchedulingPolicy.PRIORITY:
        preempted_req = max(
            self.running,
            key=lambda r: (r.priority, r.arrival_time),
        )
    else:
        preempted_req = self.running[-1]

    # 关键修复点：如果 victim 的 block 释放仍被 deferred-free 栅栏挡住，
    # 抢占它也不会给本次分配带来任何可用容量，属于零进展的状态变更。
    # 因此在删除 running 元素、回滚 token/budget、改请求状态、记抢占计数
    # 之前直接退出本轮重试；等 update_from_output() 推进 processed_step_seq
    # 后，下一次调度自然回到普通抢占 / 分配路径。
    if not self._request_blocks_can_be_freed(preempted_req):
        break

    # 以下与原逻辑一致：只有 victim 的 block 确实能立即回收时才真正抢占。
    if self.policy == SchedulingPolicy.PRIORITY:
        victim_index = self.running.index(preempted_req)
        del self.running[victim_index]
        # 保留 #49206 的游标修正，避免后续请求被静默跳过。
        if victim_index < req_index:
            req_index -= 1
    else:
        preempted_req = self.running.pop()
    # ... (_preempt_request(), preempted_reqs.append() 等后续逻辑不变)
```

tests/v1/core/test_deferred_block_free.py

新增 `_fail_one_allocation()` 一次性分配失败注入器与参数化用例

`test_allocation_retry_waits_for_fence_then_succeeds` (FCFS 失败点 1 / PRIORITY 失败点 2)，用真实 allocator 与真实 `update_from_output()` 验证“栅栏未推进时零抢占”与“栅栏推进后抢占恢复”；`_create_deferring_scheduler()` 增加 `scheduling_policy` 形参以覆盖两种策略。

tests/v1/core/test_deferred_block_free.py 中的故障注入器与两阶段回归用例:

```
def _fail_one_allocation(scheduler, call_number: int):
    """只让第 call_number 次 allocate_slots 返回 None, 其余调用透传真分配器。"""
    real_allocate = scheduler.kv_cache_manager.allocate_slots
    calls = 0

    def allocate(*args, **kwargs):
        nonlocal calls
        calls += 1
        # 仅命中的那一次返回 None (模拟分配失败),
        # 其余调用必须委托给真实 allocator ——
        # 这样断言的调用次数才反映调度器真实的重试次数,
        # 也让“栅栏推进后会不会恢复”变成可观测事实而不是 mock 行为。
        if calls == call_number:
            return None
        return real_allocate(*args, **kwargs)

    return patch.object(
        scheduler.kv_cache_manager,
        "allocate_slots",
        side_effect=allocate,
    )

@pytest.mark.parametrize(
    ("policy", "failed_call"),
    [
        ("fcfs", 1), # FCFS: 第 1 次分配即失败, victim 是 running 队尾
        ("priority", 2), # PRIORITY: 第 2 次失败, 构造 victim_index < req_index
    ],
)
def test_allocation_retry_waits_for_fence_then_succeeds(policy, failed_call):
    scheduler = _create_deferring_scheduler(policy)
    requests = create_requests(
        num_requests=3,
        num_tokens=NUM_PROMPT_TOKENS,
        max_tokens=10,
        stop_token_ids=[STOP_TOKEN_ID],
    )
    for request in requests:
        scheduler.add_request(request)
    out0 = scheduler.schedule()
    out1 = scheduler.schedule() # 制造 in-flight step, 形成 deferred fence

    worst, trigger, tail = requests
    worst.priority, trigger.priority, tail.priority = 9, 0, 1

    # 阶段一: 栅栏尚未推进, 策略选中的 victim 不能立即回收,
```

```

# 因此本次 schedule() 必须在任何队列变更前停下:
# 不产生 preempted_req_ids, 触发请求也不进入本步调度结果。
with _fail_one_allocation(scheduler, failed_call) as allocate:
    blocked = scheduler.schedule()
assert allocate.call_count == failed_call
assert not blocked.preempted_req_ids
retry_trigger = requests[failed_call - 1]
assert retry_trigger.request_id not in blocked.num_scheduled_tokens

# 阶段二: 把构造出的 model runner 输出喂回真实的 update_from_output(),
# 让 processed_step_seq 前移、deferred fence 落地。
for output in (out0, out1, blocked):
    if output.total_num_scheduled_tokens:
        scheduler.update_from_output(
            output,
            _make_model_runner_output(output),
        )

# 阶段三: 同样注入一次失败, 此时普通抢占与分配应能恢复。
expected_victim = worst if policy == "priority" else tail
with _fail_one_allocation(scheduler, failed_call) as allocate:
    resumed = scheduler.schedule()

assert allocate.call_count > failed_call
assert expected_victim.request_id in resumed.preempted_req_ids
assert retry_trigger.request_id in resumed.num_scheduled_tokens
if policy == "priority":
    assert tail.request_id in resumed.num_scheduled_tokens

```

评论区精华

本次 review 没有行内评论 (review_comments_count = 0), 有效信息集中在 approve 说明与作者的分阶段验证更新上。

1. 维护者直接改名并补注释: njhill 在 APPROVED 时写道 “@LearningMachine621 thank you for this fix and sorry for the delay in reviewing! I made a small modification to change the method name for clarity.”, 并推入 commit [5e9f8154](#) (“rename method, add comment”) 与 [4977ccb8](#) (“fix”)。对照 PR body 中仍写作 [_should_defer_request_block_free](#) 的示例可知, 最终落地名 [_request_blocks_can_be_freed](#) 来自维护者这次改名: 语义从“是否要延迟释放”反转为“现在能不能释放”, 与 [_free_request_blocks\(\)](#) 的复用点更一致, 也让 [schedule\(\)](#) 里的 [break](#) 条件读起来是正向判断。
2. 作者主动锁定“窄修复”边界: PRbody用“Scope”“Non-goals”两节反复声明不改安全栅栏、不改模型输出语义、不改 victim 选择策略, 并把替代 victim 扫描、容量感知准入等列为 future-policy RFC; 还给出 deterministic red/green 结果 (fix 树 15 passed; pinned base + 测试树 2 failed / 13 passed) 与 allocator 调用次数对比 (FCFS 3 → 1, PRIORITY 4 → 2)。

3. 合并流程事务：mergify 机器人先报冲突 (“This pull request has merge conflicts ... Please rebase”)，作者 rebase 后更新验证数据；njhill 连续三次 `/ci run`，触发 Buildkite #88187 / #88198 / #88214，最终以 `36e0b41` 合入。claude[bot] 的自动 review 因 fork 被禁用，未产生实质技术意见。
- 谓词改名与可读性：由维护者直接改代码而非提评论 (design)：接受维护者改名与补注释，谓词在释放路径与抢占路径共用，语义从“延迟判定”翻转为正向的“可释放判定”。
- 修复边界：只停在当前 victim，接受 stop-and-wait 权衡 (design)：以“保留现有 victim 顺序、做窄的正确性修复”为决策结论，性能代价被显式记录并单独测量 (P50 TTFT +20.3 ms, P99 TTFT +0.41%，吞吐 +1.90%)。
- 测试如何覆盖在 CPU 上被强制关闭的生产门禁 (testing)：接受“手动置位标志”的折中，换取在无 GPU 环境下也能确定性覆盖两条路径；代价是测试与生产门禁条件之间需要人工保持同步。
- 独立复现与 CI：pinned base 对照实验 + 持续推进合并 (other)：冲突解决与 CI 通过后合入；上游未对该修复的机制提出反对意见或要求额外改动。

风险与影响

- 风险：
 1. stop-and-wait 的显式性能权衡（最主要风险）：被挡时直接 break，本次调度 pass 中排在其后的 running 请求不再被尝试，即使另一个请求的 block 本可立即回收。作者在 PR body 中承认 “Stopping can leave the remainder of one scheduling pass unused even when a different request might be immediately reclaimable”，代价是队头延迟与瞬时欠利用；实测 P50 TTFT 由 995.702 ms 升至 1016.034 ms (+20.3 ms)，虽然 P99 TTFT 仅 +0.41%、吞吐 +1.90%。
 2. 缺少替代 victim 扫描：实现刻意停在策略选中的 victim 上，不寻找其他候选。当 FCFS 队尾或 PRIORITY 最低优先级请求长期被 fence 时，可能反复在同一处停下，公平性与饥饿问题被推迟到 future-policy RFC。
 3. 覆盖面有限：只处理 deferred-free 这一种“抢占后零增量”的原因，shared reference、connector pin 等其他零增量路径仍未处理，issue 场景之外仍可能观察到抢占抖动。
 4. 测量结论不可泛化：34.4x 抢占下降、+1.90% 吞吐均来自单一重建的冻结负载（80/80 请求、OSL 1792, n=5），作者自己标注为 “descriptive workload-specific thresholds, not confidence intervals”，且配对 P99 TTFT 效应区间为 -11.11% ~ +9.21%，不同负载 / 模型下的延迟表现需重新评估。
 5. 测试与生产门禁的同步成本：生产门禁依赖重叠 batch（异步调度或 PP），而 CPU 测试平台会强制关闭异步调度，因此测试直接置位 scheduler.defer_block_free。若未来门禁条件变更，测试不会自动失败，存在覆盖漂移风险。
 6. 核心热路径改动：schedule() 是每步都执行的调度主循环，FCFS 由 pop() 改为 [-1] 窥视 + 检查通过后再 pop()，改动虽小但位于关键路径；好在所有 mutation 都在 guard 之后，逻辑上不会出现“已出队但未抢占”的中间态。- 影响：用户侧：使用 async scheduling / PP 叠加 KV offload、PD 分离等 consumer 型 KV connector 的部署（正是 issue #49674 标注的 kimi/k3 类长上下文高并发场景）将从每轮数千次无效抢占中解脱，请求重算与无效状态突变显著减少；作者实测同一负载下抢占由 1099.8 ± 48.721

降到 32.0 ± 0.000 ，输出吞吐 +1.90%，P99 E2EL -1.86%，P50 TTFT 有约 +20 ms 的代价。未启用 deferred free 的部署完全不受影响（谓词恒真）。系统侧：调度器新增了一个可复用的“能否立即回收”判定入口，抢占路径与释放路径共享同一语义；schedule() 的 victim 选择改为“先窥视、后变更”，为未来引入候选扫描或容量感知准入留出了结构位置。团队 / 工程侧：tests/v1/core/utils.py 新增的 scheduling_policy 参数与 _fail_one_allocation() 一次性故障注入器可被后续调度器测试复用，降低了写抢占 / 预算回滚类回归用例的成本。

- 风险标记：核心调度路径变更，stop-and-wait 可能造成队头延迟，不扫描替代 victim，测试与生产门禁依赖手动同步，性能结论仅限单一冻结负载

关联脉络

- PR #45357 引入 deferred-free 安全栅栏 (PR body / issue 引用) : issue #49674 与 PR body 均声明该栅栏是有意为之且保持不变；本 PR 只是把它的判定条件抽成 _request_blocks_can_be_freed() 复用到抢占路径，安全性前提未变。
- PR #49206 PRIORITY 抢占 victim_index / req_index 游标修正 (PR body 引用) : 本 PR 把 del self.running[victim_index] 与 req_index -= 1 移到栅栏检查之后，语义保持不变；新增测试的 PRIORITY 参数 (failed_call=2) 正是为覆盖 victim_index < req_index 场景。
- PR #53723 默认关闭的 LCF victim-policy 提案 (PR body 引用) : 与本 PR 同属“victim 怎么选”的问题域。本 PR 明确不改动 FCFS/PRIORITY 顺序、不做替代 victim 扫描，把策略选择留给该提案与 future-policy RFC。
- PR #25538 抢占抖动 (preemption churn) 相关报告 (PR body 引用) : 报告了相关的抢占 churn 现象，但不涉及 consumer 侧 fence 机制，属于同一现象域的不同成因。
- PR #53250 MoRIIO 抢占后远程 KV 重载路径 (PR body 引用) : 同样出现在抢占与 KV 交付的交叉点，但走的是不同的重新加载路径；PR body 用它划清本修复的边界。
- PR #55531 [KV Connector] Support symmetric DCP disagg for hybrid mamba models: 同仓库近期 KV connector / 分离部署线的代表 PR (改动 nixl pull_worker、base_worker 与 KV 相关配置)，与本 PR 共享“异步 KV 交付 + 重叠 batch”的背景，但位于不同模块，可与本修复一起理解 connector 侧的演进节奏。