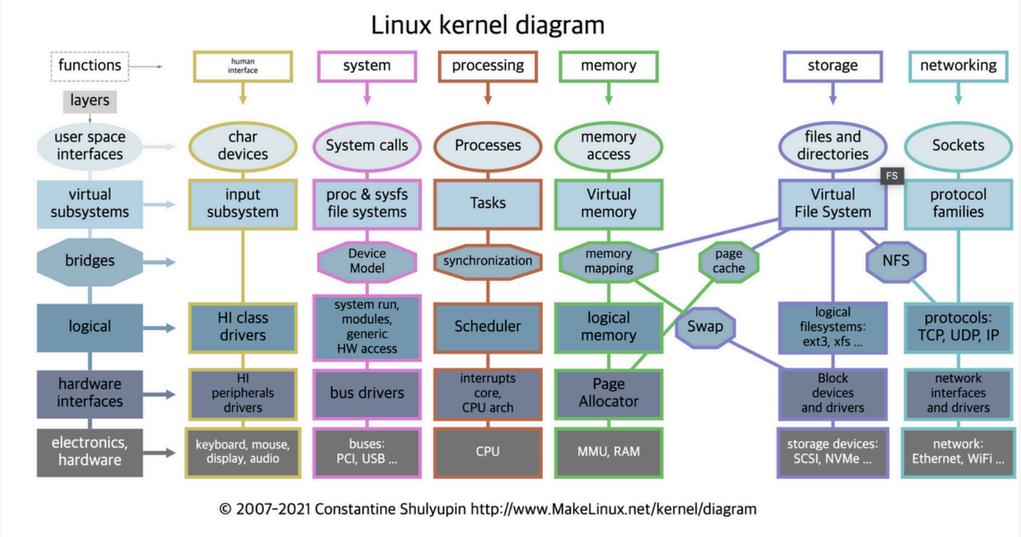
Linux Kernel Term Project

Replacing Kernel Data Structures for Performance : A Case Study on PID Allocation

[Team5] 김하람, 김도연, 이승화, 황민

How to improve the performance of my Linux Kernel?

- Linux Kernel optimizes performance by using appropriate data structures everywhere.
- However, even if it is “optimized”, there may be better data structures in certain environments or certain workloads.
- In particular, the PID management structure (PID allocation) is a performance sensitive point due to frequent process creation and deletion.

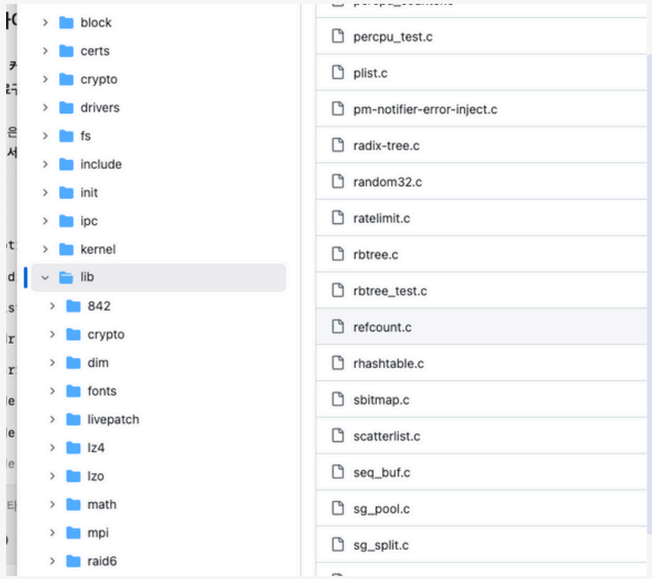


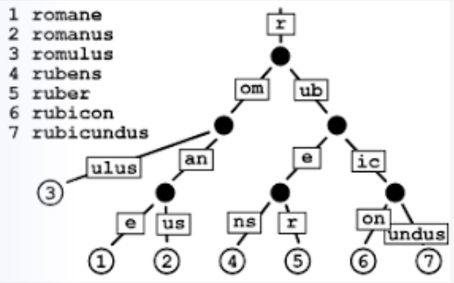
Data Structure Overview

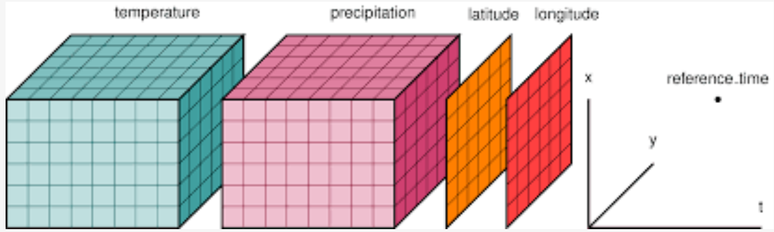
What kind of data structure does the kernel use?

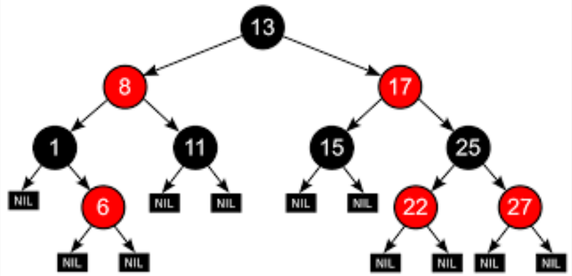
- Red Black Tree (VMA, regcahce, etc..)
- Radix Tree / XArray (page cache)
- List, hash table
- PID management structure is based on the Indexed Radix Tree

→ Insertion/Deletion is fast, but tree height increases and cache inefficiencies can occur









Linux Kernels incorporate a variety of data structures into their own ‘lib/’

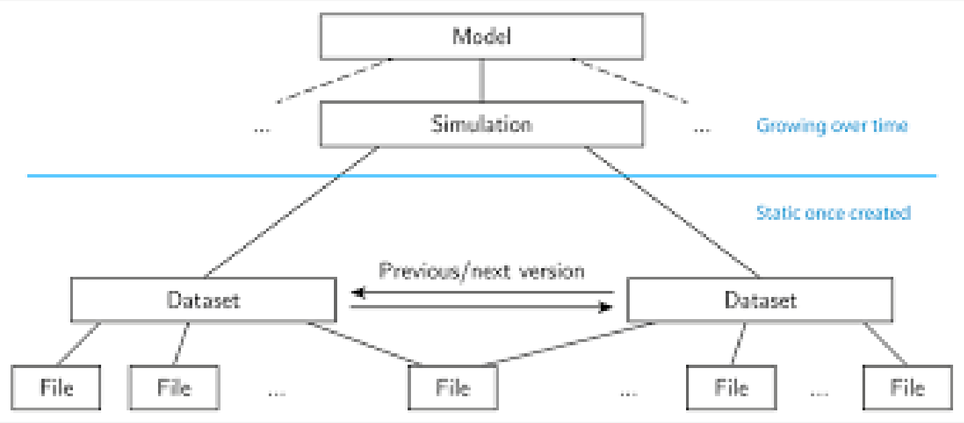
Target System：PID Management

PID System Discription

- The core structure of managing PIDs inside PID namespace is Radix Tree

alloc_pid() → PID slot assignment
free_pid() → return after PID is used

- In fork-heavy workload, PID assignment and release → Very frequent operations
- PID's operation pattern
  - Insertion
  - Search
  - Delete



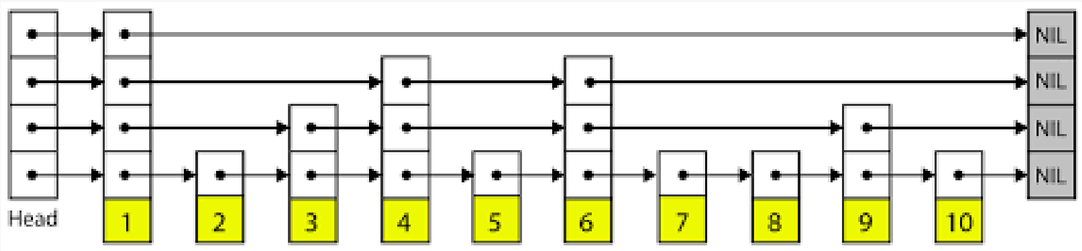
Skiplist / RB-Skiplist is a good pattern to optimize

Why Skiplist?

Why we choose SkipList?

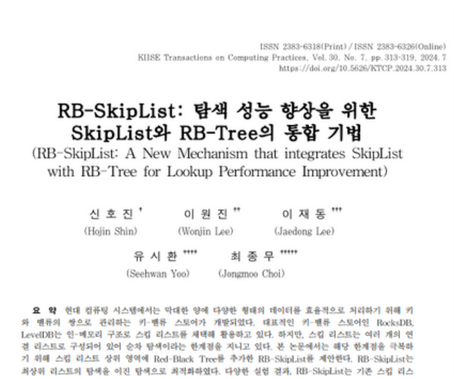
- O(log N) Performance
- Node Structure Simple (Cache - Friendly)
- No rebalancing when inserted/deleted
  → CPU cost less than RB-Tree
- Memory locality can be improved

Skip List



RB-SkipList Selection

RB- Skip List : Data structure that combines the cache affinity of SkipList and the balance of RB-Tree



ISSN 2383-6318(Print) / ISSN 2383-6326(Online)
KIISE Transactions on Computing Practices, Vol. 30, No. 7, pp. 313-319, 2024. 7
https://doi.org/10.5626/KTCP.2024.30.7.313

RB-SkipList: 탐색 성능 향상을 위한 SkipList와 RB-Tree의 통합 기법
(RB-SkipList: A New Mechanism that integrates SkipList with RB-Tree for Lookup Performance Improvement)

신 호 진 † (Hojin Shin) 이 원 진 †† (Wonjin Lee) 이 재 동 ††† (Jaedong Lee)
유 시 환 †††† (Seehwan Yoo) 최 종 무 ††††† (Jongmoo Choi)

요 약 현대 컴퓨팅 시스템에서는 막대한 양에 다양한 형태의 데이터를 효율적으로 처리하기 위해 키와 벨류의 방으로 관리하는 키-벨류 스토어가 개발되었다. 대표적인 키-벨류 스토어인 RocksDB, LevelDB는 인-메모리 구조로 스킵 리스트를 채택해 활용하고 있다. 하지만, 스킵 리스트는 여러 개의 연결 리스트로 구성되어 있어 순차 탐색이라는 한계점을 지니고 있다. 본 논문에서는 해당 한계점을 극복하기 위해 스킵 리스트 상위 영역에 Red-Black Tree를 추가한 RB-SkipList를 제안한다. RB-SkipList는 최상위 리스트의 탐색을 이진 탐색으로 최적화하였다. 다양한 실험 결과, RB-SkipList는 기존 스킵 리스

RB-SkipList: 탐색 성능 향상을 위한 SkipList와 RB-Tree의 통합 기법

RB-SkipList: A New Mechanism that integrates SkipList with RB-Tree for Lookup Performance Improvement

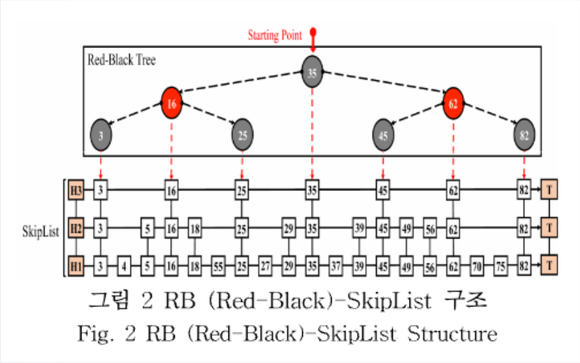


그림 2 RB (Red-Black)-SkipList 구조
Fig. 2 RB (Red-Black)-SkipList Structure

- Korean Researchers Reinforced SkipList Performance → Fast Concurrent SkipLists
- But no kernel application in the paper

→

- It was meaningful to apply it

Kernel Code Analysis - 1

Where and How will PID data structures be used?

kernel/pid.c - alloc_pid()

```
idr_preload(GFP_KERNEL);
spin_lock_irq(&pidmap_lock);

nr = idr_alloc_cyclic(&tmp->idr, NULL, pid_min,
 pid_max, GFP_ATOMIC);

spin_unlock_irq(&pidmap_lock);
idr_preload_end();
```

- PID number is first registered as a “NULL” reservation in the Radix Tree-based IDR structure using idr_alloc_cyclic().
- At this point, structure pid* is not yet connected

→ steps designed to prevent exposure to lookup.

- spin_lock_irq() → prevents concurrent writes
- idr_preload() is an optimization that prepares Radix tree node allocation in fastpath in advance.

kernel/pid.c - pid reveal

```
spin_lock_irq(&pidmap_lock);

for ( ; upid >= pid->numbers; --upid) {
   /* Make the PID visible to find_pid_ns. */
   idr_replace(&upid->ns->idr, pid, upid->nr);
   upid->ns->pid_allocated++;
}
spin_unlock_irq(&pidmap_lock);
```

- Step to write the actual structure pid* to the reserved NULL position created by idr_alloc_cyclic().
- From this point on, the PID can be inquired

find_pid_ns(nr, ns) → struct pid*

- Repeat for each level for multilevel PID namespace support.
- idr_replace() performs a leaf node entry replacement operation on XArray-based Radix Tree.

Kernel Code Analysis - 2

Where and How will PID data structures be used?

kernel/pid.c - free_pid()

```
spin_lock_irqsave(&pidmap_lock, flags);
for (i = 0; i <= pid->level; i++) {
    struct upid *upid = pid->numbers + i;
    struct pid_namespace *ns = upid->ns;
    idr_remove(&ns->idr, upid->nr);
}

spin_unlock_irqrestore(&pidmap_lock, flags);
```

- Run When the PID is no longer connected to any task.
- idr_remove() is a function that removes entries from Radix Tree.
- Repeatedly remove the PID entry by the stored level per PID namespace.
- Also adjusted the pid_allocated counter of PID namespace.
- After removal, freeing the structure is deferred to RCU call_rcu().

kernel/pid.c - find_pid_ns()

```
struct pid *find_pid_ns(int nr, struct
pid_namespace *ns)
{
    return idr_find(&ns->idr, nr);
}
```

- PID lookup is a key path that is very frequently called.
- idr_find() is internally:

```
xa_load()
└── Radix Tree-style indexed lookup
```

  - Lock-Free traversal based on RCU read lock
  - Fast, but there's a lot of pointer chasing
  - branch-heavy & multi-level tree approach
  - PID lookup allows performance bottlenecks as time builds

Kernel Code Analysis - 3

Where and How will PID data structures be used?

kernel/pid.c - find_pid_ns()

```
find_pid_ns()
 → idr_find()
  → xa_load()
   → Radix Tree node walk
```

- XArray is Radix Tree Successor and multi-level Array Tree.
- xa_load() is protected by RCU read lock on read-only path.
- Due to multi-level indexing:
  - Possible increase in cache miss
  - Low branch prediction
  - Disadvantage of branch prediction
  - In fork-heavy / syscall-heavy workload, accumulate overhead increase

Kernel Code Analysis - 4

Conclusion of Analysis

Problem of Radix Tree

- When creating a PID

→ idr_alloc_cyclic() & idr_replace()
→ Radix Tree Write lock + node expansion : cost increase

- When deleting the PID

→ idr_remove()
→ Radix Tree Entry Remove

- When looking up the PID

→ idr_find() → xa_load()
→ multi-level pinter chasing + branch-heavy

🎯 What to Expect 🎯
(Switching to SkipList / RBSkipList)

- continuous link-based pointer chain
- branch prediction-friendly
- No rebalancing when inserted/deleted
- Lookup path simplified

→ CPU cache efficiency increased

- Can actually improve fork-heavy workload

Implementation - 1

include/linux/pid_skiplist.h - Define SkipList Interface

```
struct pid_sl_node {
    int key;
    struct pid *pid;
    struct pid_sl_node **forward;
    u8 level;
    struct rcu_head rcu;
};
struct pid_skiplist {
    int level;
    struct pid_sl_node *header;
};
```

- PID SkipList Struct Define - (pid_sl_node, pid_skiplist)
- include ‘insert/lookup_rcu/remove function prototype
- CONFIG_PID_SKIPLIST conditional compiling activated only when on
- Designed to be called by the same API as Radix Tree on the outside

Modifying external dependencies

- Replace two external files using the idr API with a skip corresponding function.
  - arch/powerpc/../spufs/sched.c
  - fs/proc/loadavg.c

Modification for connecting pid_skiplist to kernel build system

init/Kconfig

```
config PID_SKIPLIST
    bool "Use skiplist for PID namespace"
    default n
```

- Add selectable options in menuconfig
- When ON, use SkipList allocator instead of Radix Tree

Implementation - 2

SkipList Implementation Summary

lib/pid_skiplist.c

```
struct pid_sl_node {
    int key;                // PID key value
    struct pid *pid;        //Pointer to the actual structure
    struct pid_sl_node **forward;
    u8 level;               //Node Level
    struct rcu_head rcu;
};
```

SkipList Node Structure

- Save PID as key
- Configure multiple levels with forward pointer array

→ Explore the mean O(log N)

- RCU-based memory release prevents race during lookup

Initialize

```
sl->level = 1;
sl->header = pid_sl_node_alloc(PID_SL_MAX_LEVEL, gfp | __GFP_NOFAIL);
sl->header->key = INT_MIN;
memset(sl->header->forward, 0, max * sizeof(...));
```

- IDR's idr_init() role
- Header nodes start at all levels
- Header sentinel required because skiplist is "aligned pointer chain"

Implementation - 3

SkipList Implementation Summary

lib/pid_skiplist.c

```
// 1. Save predecessors
for (i = sl->level - 1; i >= 0; i--)
    while (x->forward[i] && x->forward[i]->key < key)
        x = x->forward[i], update[i] = x;

// 2. Generate random level for the new node
lvl = pid_sl_random_level();
if (lvl > sl->level) sl->level = lvl;

// 3. Insert the new node into the skiplist
x = pid_sl_node_alloc(lvl, gfp);
x->key = key;
for (i = 0; i < lvl; i++) {
    x->forward[i] = update[i]->forward[i];
    update[i]->forward[i] = x;
}
```

Insertion - pid_skiplist_insert()

- Iterates through levels, advancing pointer as far as possible in each level before moving down to the next

- Random Level Based

→ Key Performance Elements of Skiplist
→ Geometric distribution

- Replace the idr_alloc_cyclic()

- Increased PID search efficiency as it is inserted in aligned locations

- insert/remove operates under pidmap_lock

→ keep the same synchronization model

Implementation - 4

SkipList Implmenation Summary

lib/pid_skiplist.c

```
for (i = sl->level - 1; i >= 0; i--)
    while (next = x->forward[i], next && next->key < key)
        x = next;

next = x->forward[0];
return (next && next->key == key) ? next->pid : NULL;
```

Lookup → pid_skiplist_lookup_rcu()

- Replace complete idr_find()
- Search by descending to “Up Level → Down Level”
- Branch prediction-friendly pattern, the biggest advantage of skiplist
- Lookup protects with RCU read lock → Increased synchronicity performance

```
for (i = sl->level - 1; i >= 0; i--)
    while (...) update[i] = x;

x = x->forward[0];
for (i = 0; i < sl->level; i++)
    if (update[i]->forward[i] == x)
        update[i]->forward[i] = x->forward[i];

call_rcu(&x->rcu, pid_sl_node_rcu_free);
```

Remove → pid_skiplist_remove()

- Role of idr_remove()
- O(log N) when the forward pointer is cut off
- Securely handle memory releases with call_rcu()

Implementation - 5

Kernel Connection Points Summary

include/linux/pid_namespace.h

```
struct pid_namespace {
   ...
#ifdef CONFIG_PID_SKIPLIST
   struct pid_skiplist slist;    // Skiplist 기반 PID 저장소
#else
   struct idr idr;               // 기존 Radix Tree 저장소
#endif
};
```

- Replace existing idr field with SkipList
- Runtime selectable with Kconfig option

```
nr = idr_alloc_cyclic(&ns->idr, NULL, pid_min, pid_max, GFP_ATOMIC);
...
idr_replace(&upid->ns->idr, pid, upid->nr);
```

↓

```
nr = pid_skiplist_insert(&ns->slist, pid_nr, pid, GFP_ATOMIC);
```

- Change the path called inside alloc_pid()

find_pid_ns(), find_ge_pid(), free_pid() are also changed in the same way →

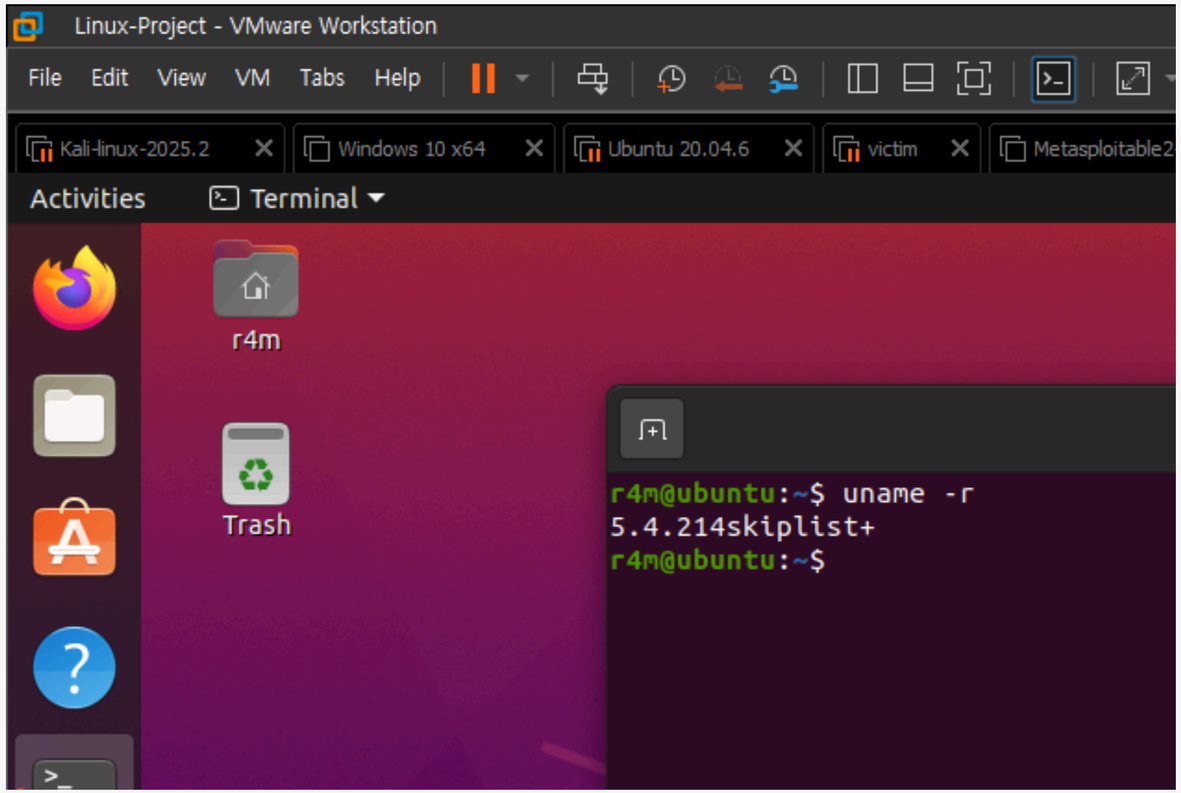
```
return pid_skiplist_lookup_rcu(&ns->slist, nr);
```

```
return pid_skiplist_find_ge_rcu(&ns->slist, nr);
```

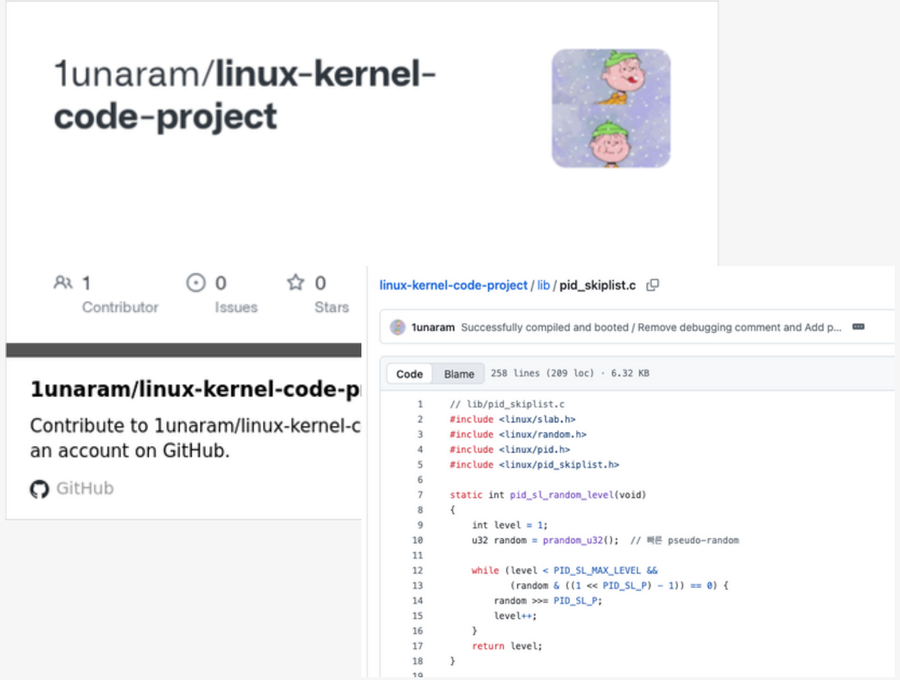
```
pid_skiplist_remove(&ns->slist, nr);
```

Implementation - 6

Kernel Compile Success



More about the implementation process in here.



Implementation - 7

RB-SkipList Implementation：Header

include/linux/pid_skiplist.h

```
struct pid_sl_node {
    int key;                    // PID key value
    struct pid *pid;            // Pointer to pid structure
    struct pid_sl_node **forward; // Skiplist forward pointers for
each level
    u8 level;                   // Number of levels this node spans
    struct rcu_head rcu;

   /*At the highest level, the node is also linked into RB-tree  */
    struct rb_node rb_node;
    bool in_rb_tree;
};
```

include/linux/pid_skiplist.h

```
struct pid_skiplist {
    int level;                  // Current maximum level of the skiplist
    struct pid_sl_node *header;   // Header node of the skiplist

    /* RB-tree used for the highest level search */
    struct rb_root top_rb_root;
};
```

Implementation - 8

RB-SkipList Implementation Summary

lib/pid_rbskiplist.c

```
while (level < PID_SL_MAX_LEVEL &&
      (random & ((1 << PID_SL_P) - 1)) == 0) {
    random >>= PID_SL_P;
    level++;
}
```

Level Decision : pid_sl_random_level()

Implementing a Skiplist Level Random Distribution

- prandom_u_32() + bitmask based
- Increasing Probabilistic Level → Maintatin a balanced skipplist

```
sl->level = 1;
sl->header = pid_sl_node_alloc(max, gfp);
sl->top_rb_root = RB_ROOT;
```

RB-tree + SkipList Initialize

- Header is at maximum level
- top-level RB-tree initializes to RB_ROOT

Implementation - 9

RB-SkipList Implementation : Insertion

lib/pid_rbskiplist.c

```
x = rb_find_le(sl, key);
```

```
for (i = sl->level - 1; i >= 0; i--) {
    while (x->forward[i] && x->forward[i]->key <
key)
        x = x->forward[i];
    update[i] = x;
}
```

```
if (lvl == sl->level && lvl == sl->header->level) {
    rb_insert_node(sl, x);
}
```

Insertion Key

Search start point with RB-tree
→ connect to skiplist

1) Find the maximum node smaller the key with RB-tree

2) Skiplist section navigation

3) If the level is at the top, update the RB-tree as well.

Implementation - 10

RB-SkipList Implementation : Lookup

lib/pid_rbskiplist.c

```
x = rb_find_le(sl, key);
```

```
for (i = sl->level - 1; i >= 0; i--) {
    while (next && next->key < key)
        x = next;
}
```

```
next = x->forward[0];
if (next && next->key == key)
    return next->pid;
```

Lookup Function

RB-tree start → skiplist finish

1) First find search start node in RB-tree

2) Explore subsequent skiplist steps

3) Check key at level 0

Implementation - 11

RB-SkipList Implementation : Remove

lib/pid_rbskiplist.c

```
if (x->in_rb_tree)
    rb_erase(&x->rb_node, &sl->top_rb_root);
```

```
while(new_level > 1 && !header->forward[new_level - 1])
    new_level--;
```

Remove Function

Remove all RB-tree and Skiplist

1) First remove RB-tree

2) After remove link in skiplist and adjust the level.

RB-SkipList Implementation Summary

Hybrid Design

- Skiplist for general O(log n) operations
- RB-tree accelerates top-level search
- Combines pointer-chaining locality + balanced-tree indexing

Key Techniques

- Random level generator
- RCU-safe remove → call_rcu() / RB-tree fastest-start search -> rb_find_le()
- Minimal modifications to PID allocator path

Why it matters

- Reduces pointer chasing
- Improves cache locality, Multi-Core results depend on lock contention optimizations

Benchmark Methods

Why Benchmark PID?

- PID Data Structure Replacement (IDR → SkipList / RB-SkipList)
  - To quantitatively verify how it affects actual process creation
- Benchmarked the system in two ways to measure cost

1 Micro benchmark(fork loop) → Direct measurement of PID alloc/free performance

2 System benchmark (UnixBench + stress-ng + perf) → Analysis of PID's Impact on Overall System Performance

Microbenchmark : fork() Loop

🎯 GOAL

- Directly hit PID allocate/free, the “hot path” of PID data structure changes
- Test that shows the purest effect of data structure change

```
for (i=0; i<10000; i++) {
    pid = fork();
    if (pid == 0) _exit(0);
    waitpid(pid, NULL, 0);
}
```

- 10,000 fork → instant wait
- Repeated measurement of the entire cycle of
  - PID allocate/lookup/delete
- Calculate the total time (ms) with CLOCK_MONOTONIC

System Benchmark : UnixBench, Perf

🎯 GOAL

- PID data structure changes
  - Evaluate how process-driven workloads affect overall performance

Test category

- Process Creation (fork)
  - task-clock, context-switches, page-faults, cycles, instructions
- Clone (with namespace)
  - CPU cycles, instructions executed, context-switches
- Context Switching
  - context-switches, cpu-migrations
- Detailed CPU Profiling
  - Sampling by ‘perf-record’, anlaysis by ‘perf report’
- Wait Performance
  - CPU cycles, instructions executed, context-switches

<< Test Environment >>

[Host]

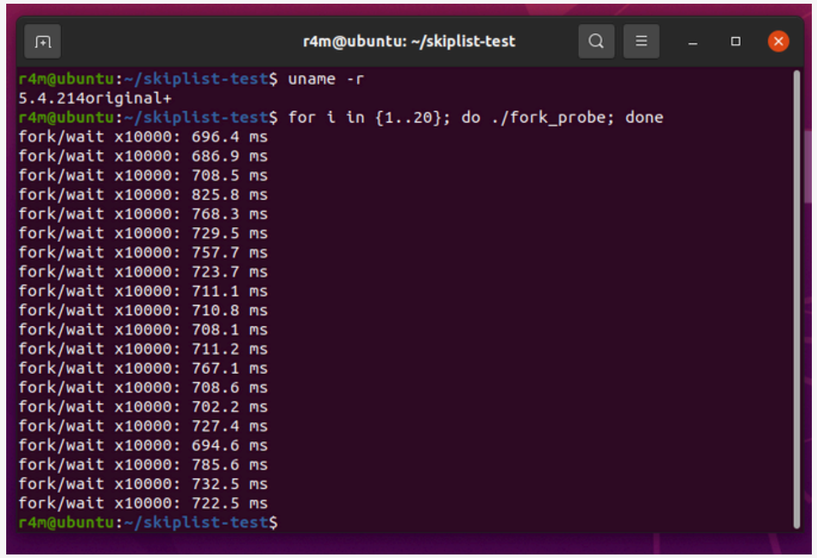
- CPU: Intel Core Ultra 7 155H
- # of processor cores: 16
- # of logical processors: 22
- Memory: 32GB

[Virtual Machine]

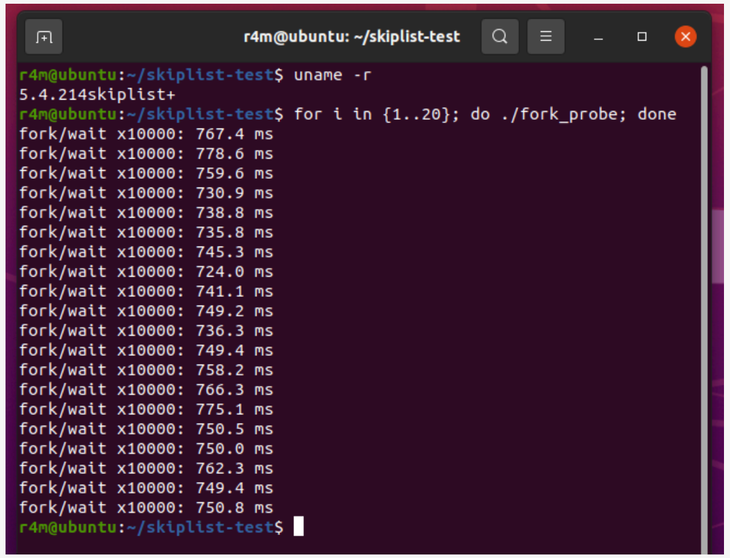
- # of processor cores: 12
- Memory: 8GB

Experiment Results (1)

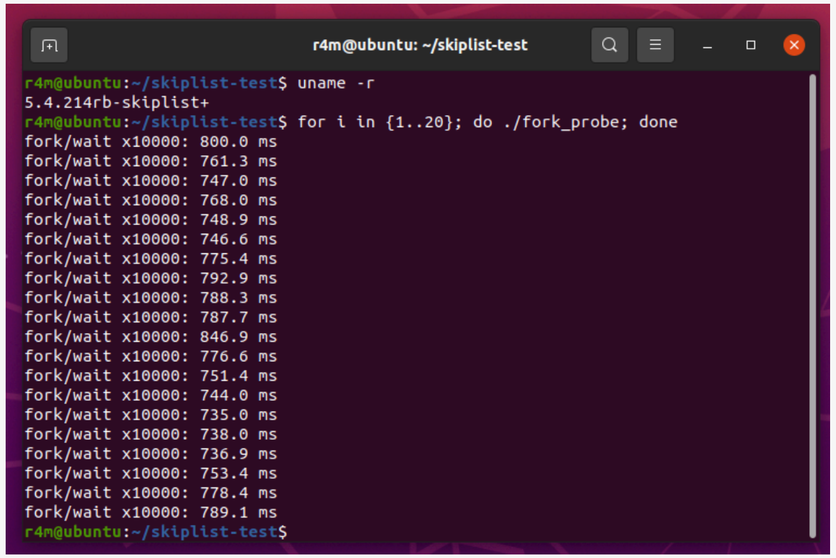
Fork Microbenchmark



5.4.214 original



SkipList

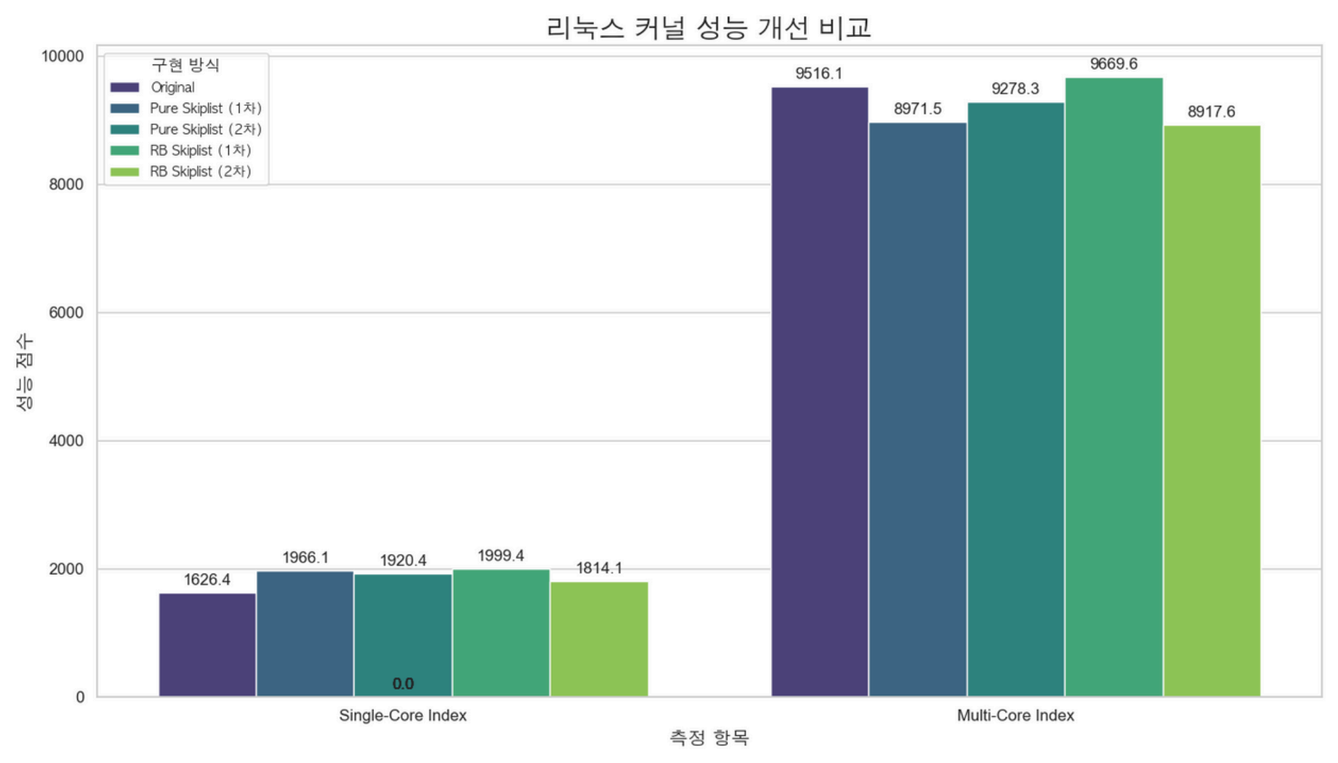


RB-SkipList

Experiment Results (2)

UnixBench Executive Summary

More Detailed Analysis Refer to Appendix Page 1~4



Key Findings

- **Single-Core Performance Improvement**
  - All skiplist-based implements showed distinct performance improvements of 10~20% or more over Original.
- **Difference in Multi-Core Performance**
  - Best performance : RB SkipList(1st) (+1.6%)
  - Improved Pure SkipList : Pure SkipList (-2.5%, 3.4% improvements)

→ In multicore, there is large variation by implementation, suggesting the importance of Concurrent Control Optimization.

- **High Performance of certain load Tests**
  - In the 'Process Creation' and 'System Call Overhead' categories, all Skiplist-based implementation showed an overwhelming performance

Experiment Results (3)

Perf Summary

Positive effects of implementing SkipList

- Significant Performance improvement on Clone and Context Switch

Analyzing the causes of Fork Performance degradation

- Additional data structure management overhead during fork for SkipList in SkipList/RB-SkipList
- Cost of managing skiplist level when PID allocation / release

Limitations of hybrid approach

- RB-Skiplist does not perform as expected
- Two data structure synchronization overheads cause performance degradation

Experiment Results (4)

Perf Summary

More Detailed Analysis Refer to Appendix Page 5~10

Workload Type	Recommended Kernel	Why?
fork - heavy	Original Kernel	Best Fork Performance
clone - heavy	SkipList or RB-SkipList	+12.8% improvement in clone operations
Context switching-heavy	SkipList	Fastest context switch throughput
Balanced Workload	SkipList	Overall +3.8% balanced Performance
Low variance requirements	RB-SkipList	Most Stable Performance Variance

Experiment Results (3)

Summary of pros and cons by Kernel

Original (Radix Tree)

- Adv : Best for Fork Performance, Minimum context Switch at wait calls, Minimum CPU Migration, Verified Stability
- Dis : Clone / Switch Performance is relatively low

SkipList

- Adv : 12.8% improvement in clone performance, Increase Context Switch Performance by 6%, Low-result variability and predictable performance, Minimum overhead per context switch (5.85 ns)
- Dis :3.2% lower Fork performance, Increase the number of context switches during wait (+37%)

RB-SkipList

- Adv : 12.9% improvement in clone performance, Minimum Clone test variability (1.75%), Original-like Fork performance
- Dis : Context Switch Performance degrades by 6.6%, Increasing CPU Migration

Discussion (1)

Why Single-Core Improved but Multi-Core Became Unstable?

Structural Reasons

- Skiplist has simple pointer chains → naturally strong for single-core
  - Simple forward-chain traversal → high cache locality
  - No rebalancing → predictable operation cost
- In Multi-Core, performance is dominated by concurrency control
  - PID alloc/free touches a shared structure
  - Skiplist is not lock-free and level maintencance adds overhead
- Radix Tree has decades of Linux kernel tuning
  - Radix tree benefits from deep kernel-level optimizations
  - Developed to minimize lock contention
  - Stronger-than-expected baseline for multi-core workloads

Discussion (2)

Why Single-Core Improved but Multi-Core Became Unstable?

Important Finding from Experiments

This behavior is also reported in academic papers.

- Skiplist-based structures excel in single-thread execution,
- but require specialized concurrency designs to scale on multi-threaded workloads

Conclusion

1. Replacing the PID data structure can actually improve performance

Single-Core → Clear, consistent improvement

- Skiplist / RB-Skiplist：18~23 %

Process Creation → Up to 24% improvement

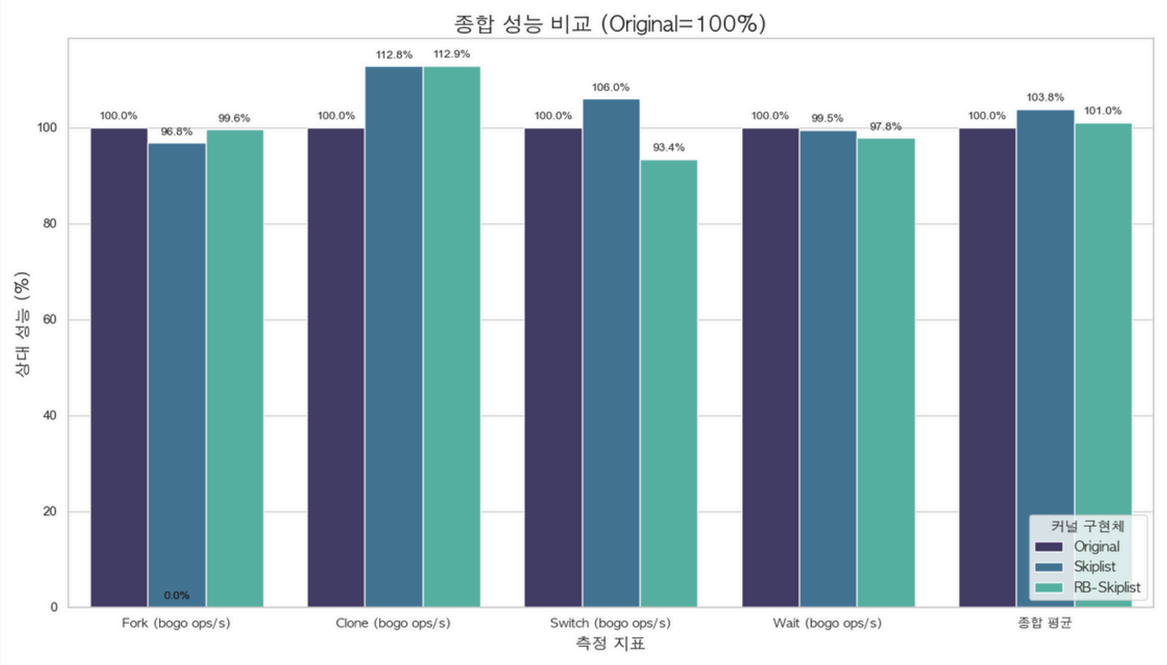
2. Multi-Core results reveal the need for concurrency optimization

Skiplist is structurally strong for single-core
But multi-core bottlenecks arise from lock contention & cache-line conflicts
RB-Skiplist has resarch value because no kernel-level implementation existed before

3. Research Contributions of This Project

Kernel-level data structure replacement is extremely challenging
Completing real benchmarking is a strong contribution

Conclusion



This project shows that the PID allocator has room for improvement - and proves it experimentally

Q&A

Thank You

Further Research

1. Where can we excel?

Ordered traversal in a designated group of PIDs is made ~35x faster

- Skiplist / RB-Skiplist : Can perform inserts, deletions, and searches in O(logN)

Unlike radix tree, moving to neighbor node is part of the basic search API

2. Scope?

Security / Policy enforcement: “walk PIDs in a range monitoring memory usage”
Researching OS: Further enables testing environments with numerical PID groupings
While higher level abstractions such as cgroups already exist, this new option still presents the plausibility of a new paradigm in “query-able ordered PID structures”

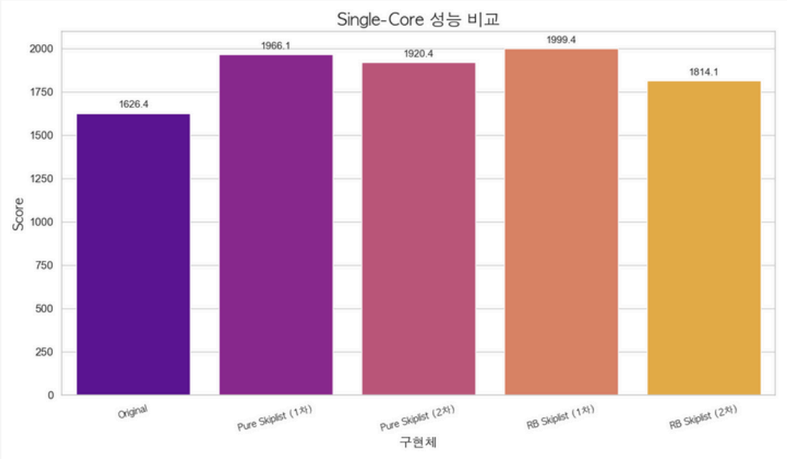
```
[ 1337.952639] rbskiplist_range_test: starting benchmark (PID_TEST_MAX=6000)
[ 1337.958605] rbskiplist_range_test: resolved pid_skiplist symbols OK
[ 1337.958853] rbskiplist bench: built skiplist (all threads), PID range [1, 476
3]
[ 1337.958854] Benchmarking 100000 random ranges, max width 5000
[ 1341.703136] RESULT: baseline(radix): 3644560624 ns total, avg 36445 ns/range
[ 1341.703137] RESULT: rbskiplist     : 99635293 ns total, avg 996 ns/range
```

simple test scanning PIDs across a random designated range

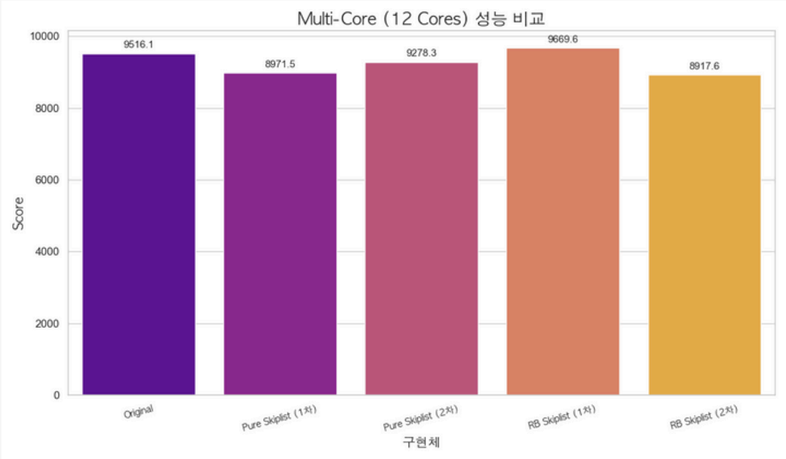
Detailed Analysis - UnixBench (1)

System Benchmarks Index Score

- **Single Core**
  - All Skiplist-based implementations outperform the original. For pure Skiplist, the first and second results are very stable, with a range of ±2.4%. RB Skipplist performed best in the first round, but drops slightly in the second round.
- **Multi Core (12 cores)**
  - Only RB SkipList exceeds the original
  - the second test of Pure Skiplist provides 3.4% improvement over the first order, suggesting that there is room for simultaneous control optimization.



Single Core

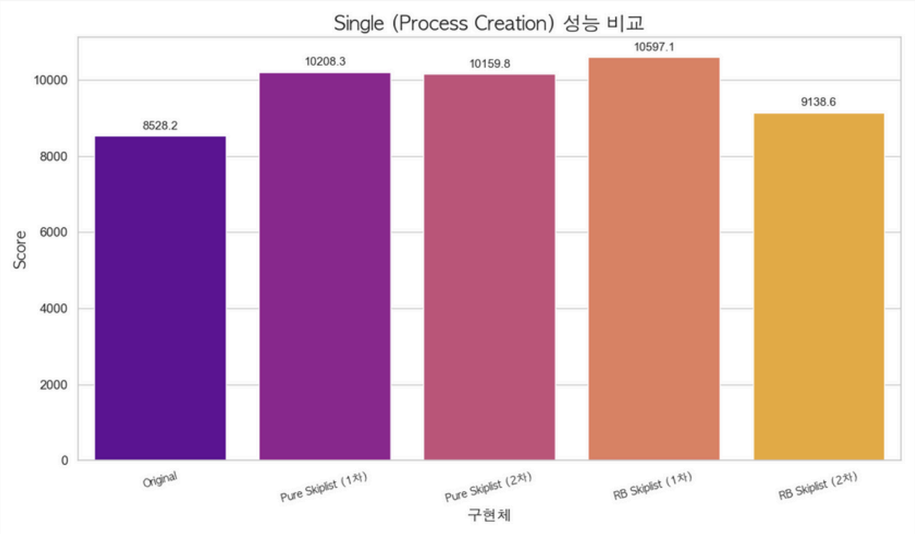


Multi Core

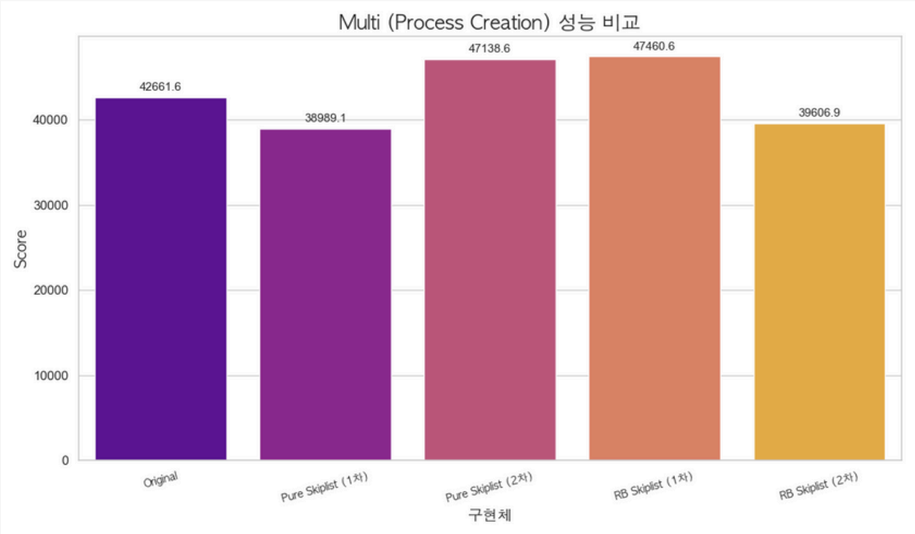
Detailed Analysis - UnixBench (2)

Process Creation

- **Multi-Core performance is significantly improved to +10.5% in the pure skiplist secondary test, which clearly demonstrates the effectiveness of the concurrency control optimization.**



Single Core

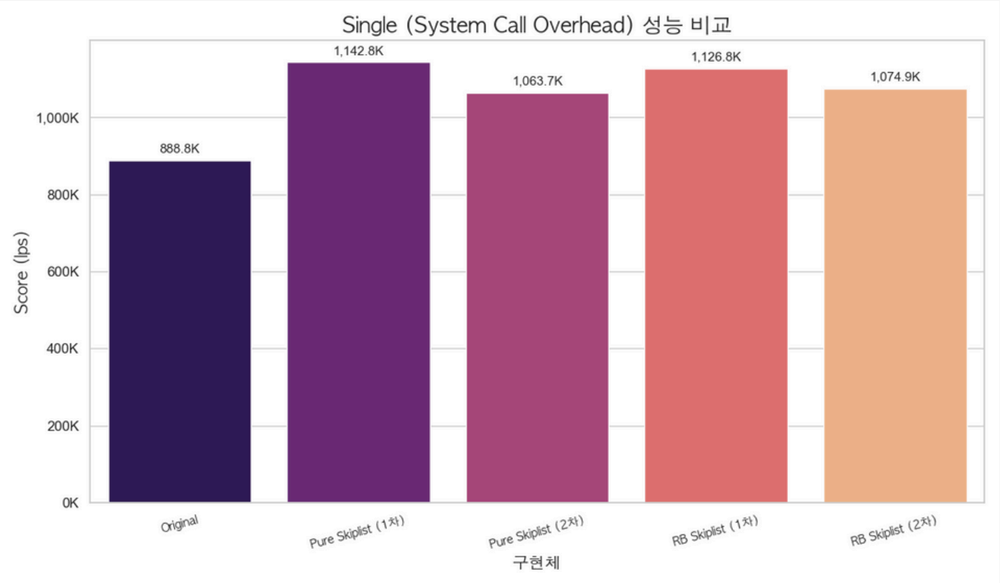


Multi Core

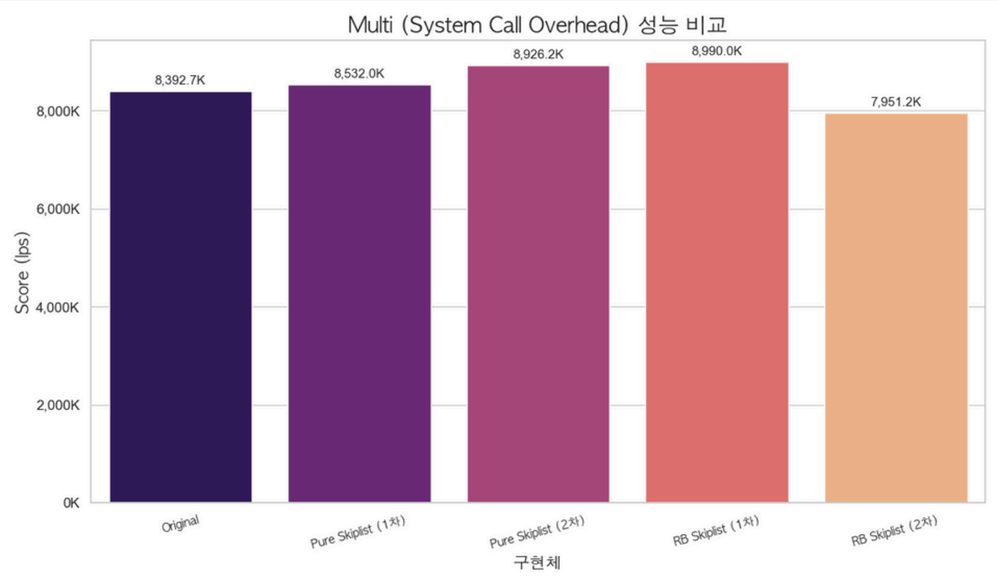
Detailed Analysis - UnixBench (3)

System Call Overhead

- **Multi-Core System Overhead is +4.7% better than 1st in Pure Skiplist 2nd, demonstrating optimization.**



Single Core



Multi Core

Detailed Analysis - UnixBench (4)

Single-Core Optimization Effect

- RadixTree's memory access pattern overhead : existing structure is a deep tree structure resulting in frequent cache misses
- SkipList's efficient search : Perform O(logn) search with less memory access
- Improve item assignment/release performance : Skiplist's simple node structure enables faster behavior

Multi-Core Performance Variability

- Difference between two tests(RB Skiplist 1st and 2nd) : Performance deviations of 2~10% in the same implementation are affected by environmental factors such as system load, cache status, etc

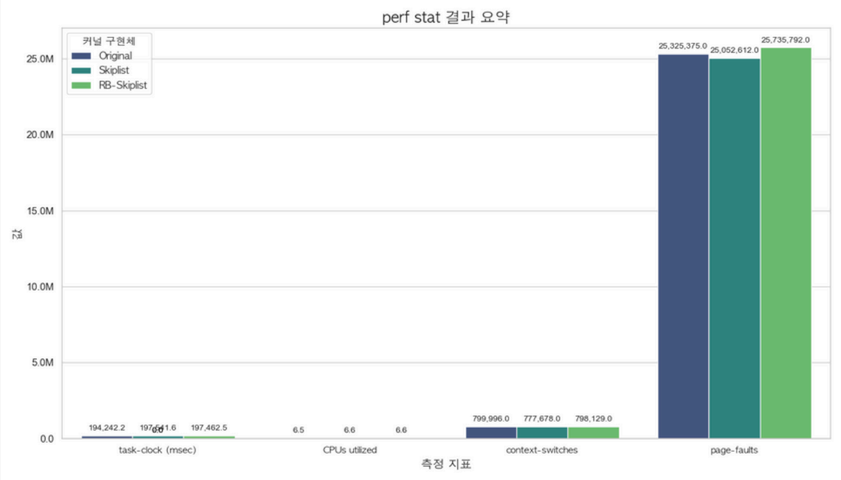
Powerful improvements to Process Creation and System Call Overhead

- Both indicators showing improvements of 20~30% or more strongly suggest that PID allocation/search logic was a major bottleneck.

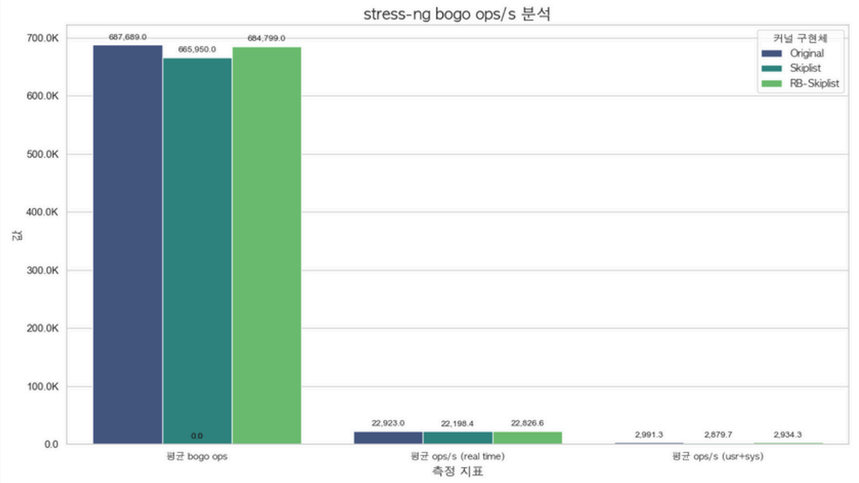
Detailed Analysis - Perf (1)

Fork test (Process Creation)

- Original kernel best in Fork Performance (approximately 3.3% higher throughput)
- Skiplist has the least number of context switches (-2.8%)
- Page Fault water has the lowest number of skiplists (-1.1%)
- RB-SkipList performs similarly to Original



Perf stat

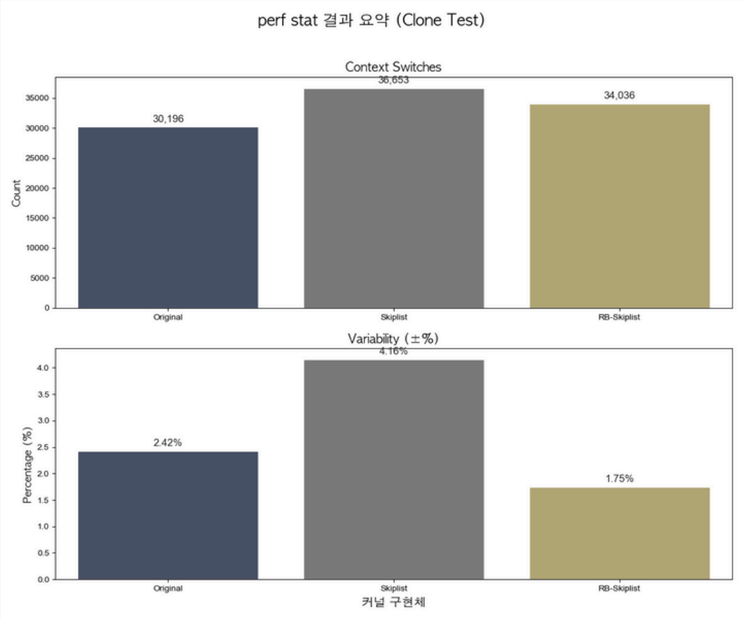


Stress-ng

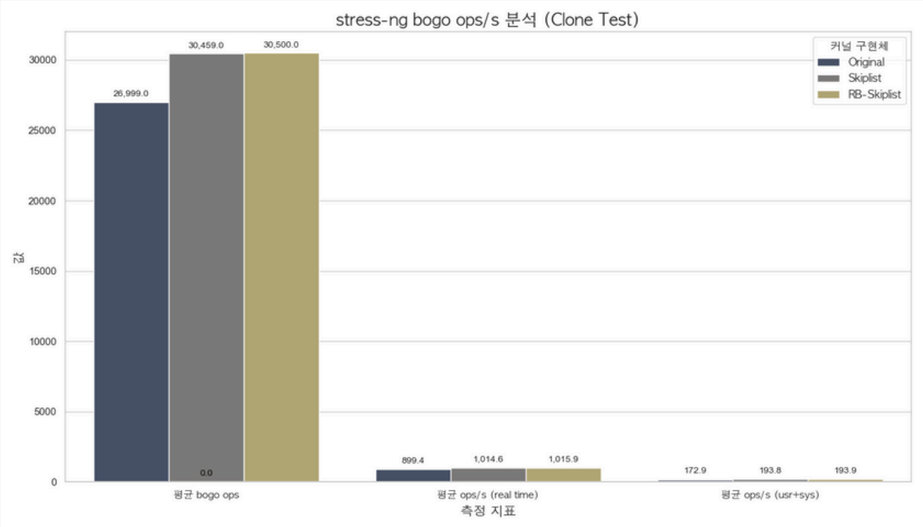
Detailed Analysis - Perf (2)

Clone Test (Thread Creation)

- Skiplist and RB-SkipList are 12.8% better at clone performance
- Original has the lowest number of context switches but low throughput
- RB-SkipList results in lowest volatility and stable



Perf stat

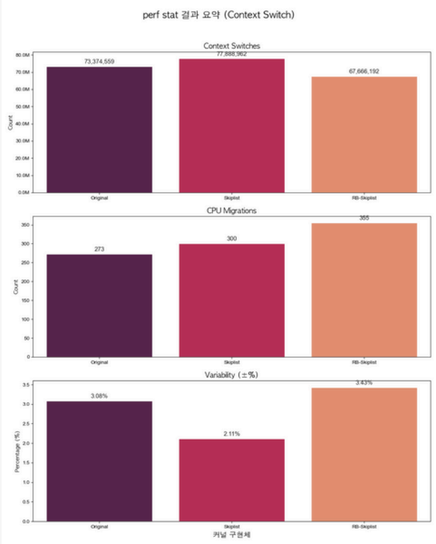


Stress-ng

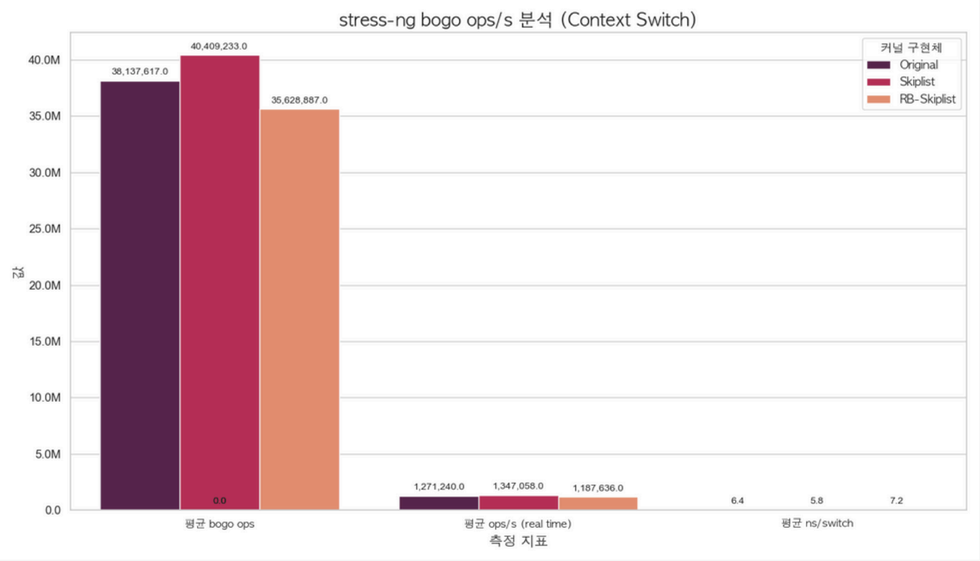
Detailed Analysis - Perf (3)

Context Switch Test

- Skiplist best in Context Switch Performance (+5.6% than Original)
- Skiplist has the shortest average time per context switch (5.85 ns)
- RB-Skiplist has the lowest performance (-6.6% than Original)



Perf stat

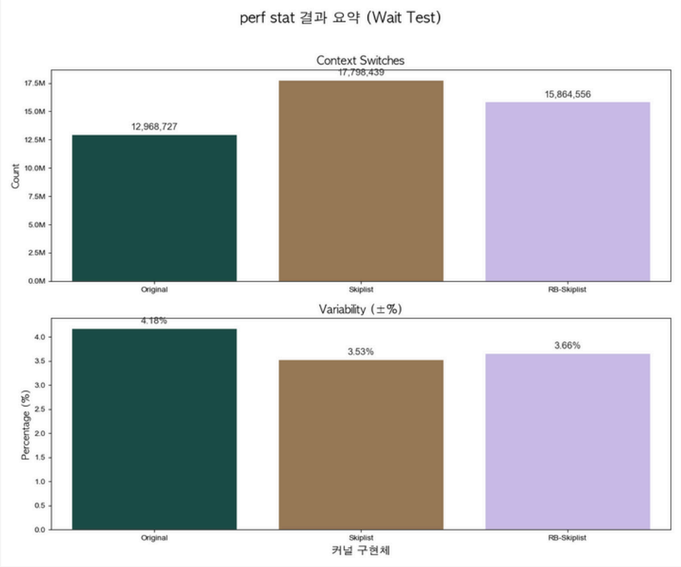


Stress-ng

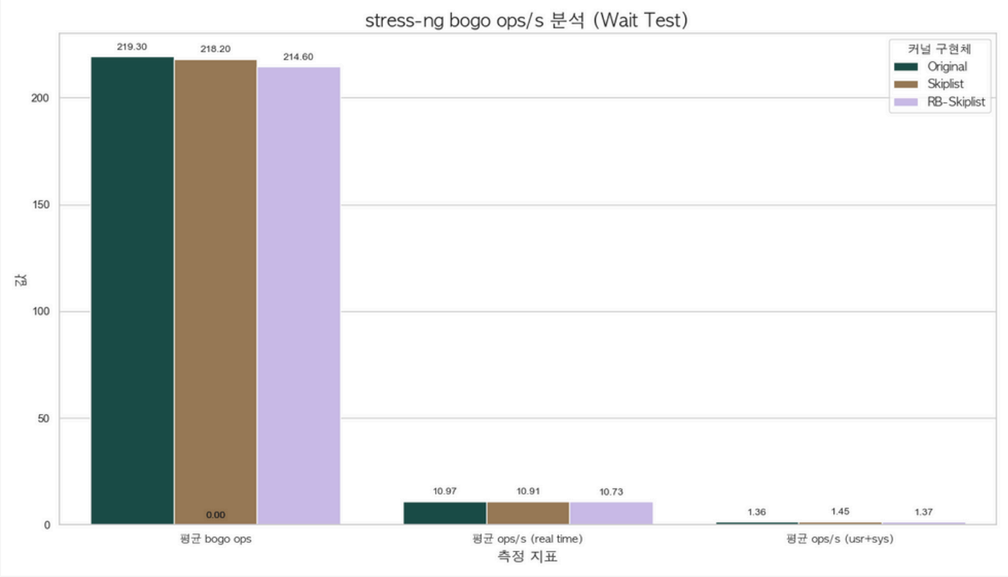
Detailed Analysis - Perf (4)

Wait Test (Waiting Process)

- Wait test shows minimal difference between three kernels.
  - Skiplist has the highest efficiency compared to usr+sys time (6.6%)
  - Original has the lowest number of context switches



Perf stat

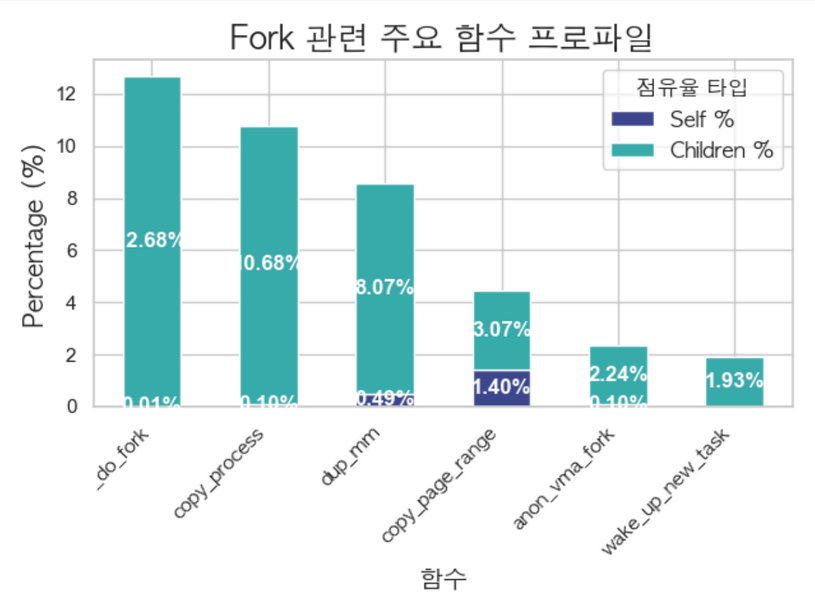


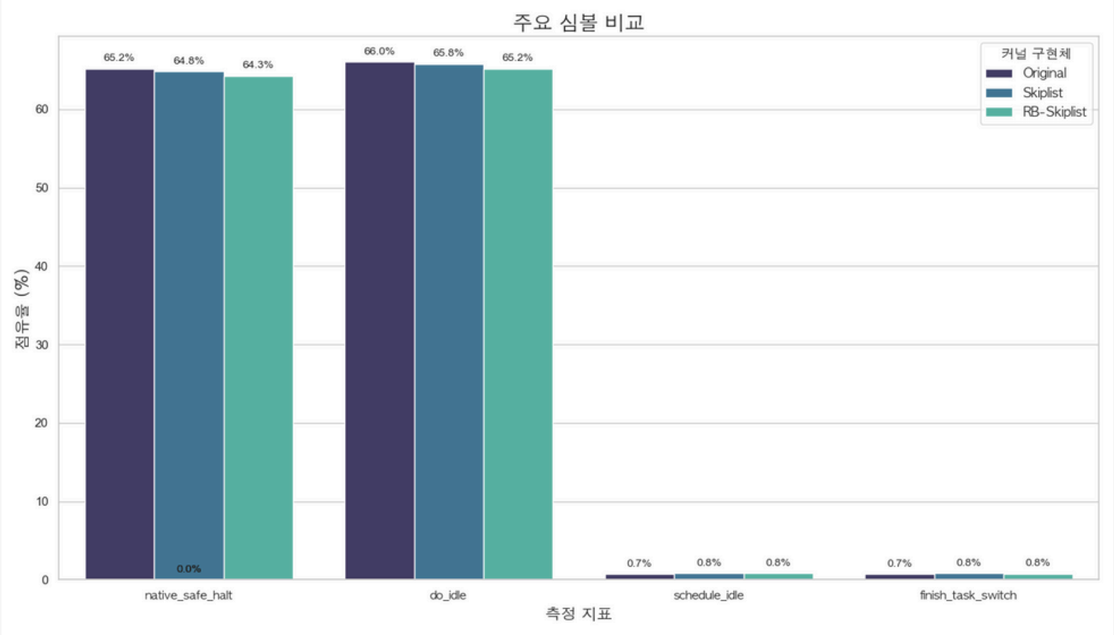
Stress-ng

Detailed Analysis - Perf (5)

CPU Profiling

- Comparsion of key symbols
  - Similar Idle status result in CPU utilzation
  - Slightly increased scheduling overhead in Skiplist/RB-Skiplist
- Key function Profiles for Fork





Another Implementation in RB-SkipList

Additionally, we implemented an alternative approach to evaluate whether a different structural design could further optimize PID operations.
→ This secondary implementation allowed us to compare multiple design strategies and assess their impact on overall system performance.

From here, we will present the implementation steps of this method and provide the benchmark (UnixBench) results obtained from our experiments.

<< Test Environment >>

[Host]

- CPU: AMD Ryzen 5 5600 6-Core Processor
- # of processor cores: 6
- # of logical processors: 12
- Memory: 32GB

[Virtual Machine]

- # of processor cores: 4
- Memory: 20GB

Another Version of RB-skipList

And we tried to implement RB-skiplist with the code of the team member's other methods.

Structure Overview

- Make RB-tree with nodes higher than level 3 in skiplist
- Skiplist contains all the nodes and provides read() even when write operation holds lock
- RB-tree provides a fast navigation

Node Structure

```
struct pid_sl_node {
    int key;
    int level;
    struct pid *pid;
    struct rcu_head rcu;

#ifdef CONFIG_PID_RB_SKIPLIST
    struct rb_node rb_node;
#endif
    struct pid_sl_node *forward[];
};
```

Tuning Parameters

```
#define PID_SL_MAX_LEVEL 16
#define PID_SL_P 2
#define PID_SL_RB_THRESHOLD 3
```

Appendix


```
 size_t size = sizeof(*node) + (level * sizeof(struct
pid_sl_node *));
  ...
 node = kzalloc(size, gfp);
```

```
if (lvl >= PID_SL_RB_THRESHOLD) {
   pid_sl_rb_insert(sl, new_node);
 }
 write_sequnlock(&sl->rb_seq);
```

```
#ifdef CONFIG_PID_RB_SKIPLIST
 write_seqlock(&sl->rb_seq);
#endif
```

```
seq = read_seqbegin(&sl->rb_seq);
rb_start = pid_sl_find_le_rb(sl, key);

if (read_seqretry(&sl->rb_seq, seq)) {
 rb_start = NULL;
}
x = rb_start ? rb_start : sl->header;
```

- Skiplist + RB-tree mix improves navigation performance
- Use ¼ probability for skiplist to increase search speed

- Calculate the size of node and pointer and allocate memory by using kzalloc

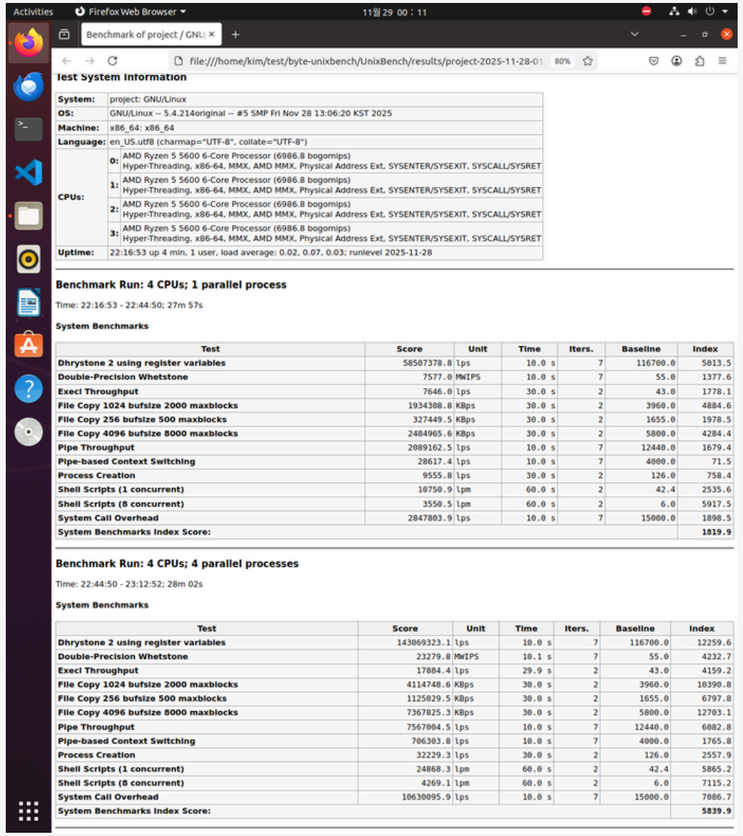
- Insert the node with level >= 3 to RB-Tree
- Maintaining the Skiplist default insertion logic

- Use seqlock to maintain RB-Tree

- Lookup starts from RB-Tree
- If there is lock at RB-Tree, core does not wait and start searching from header of skiplist
- Increased parallel lookup performance in multicore environments

UnixBenchmark Summary Screenshot

Original Benchmark



Test System Information

System:	project: GNU/Linux
OS:	GNU/Linux -- 5.4.214original -- #5 SMP Fri Nov 28 13:06:20 KST 2025
Machine:	x86_64: x86_64
Language:	en_US.utf8 (charmap="UTF-8", collate="UTF-8")
CPUs: 0:	AMD Ryzen 5 5600 6-Core Processor (6986.8 bogomips) Hyper-Threading, x86-64, MMX, AMD MMX, Physical Address Ext, SYSENTER/SYSEXIT, SYSCALL/SYSRET
CPUs: 1:	AMD Ryzen 5 5600 6-Core Processor (6986.8 bogomips) Hyper-Threading, x86-64, MMX, AMD MMX, Physical Address Ext, SYSENTER/SYSEXIT, SYSCALL/SYSRET
CPUs: 2:	AMD Ryzen 5 5600 6-Core Processor (6986.8 bogomips) Hyper-Threading, x86-64, MMX, AMD MMX, Physical Address Ext, SYSENTER/SYSEXIT, SYSCALL/SYSRET
CPUs: 3:	AMD Ryzen 5 5600 6-Core Processor (6986.8 bogomips) Hyper-Threading, x86-64, MMX, AMD MMX, Physical Address Ext, SYSENTER/SYSEXIT, SYSCALL/SYSRET
Uptime:	22:16:53 up 4 min, 1 user, load average: 0.02, 0.07, 0.03; runlevel 2025-11-28

Benchmark Run: 4 CPUs; 1 parallel process

Time: 22:16:53 - 22:44:50; 27m 57s

System Benchmarks

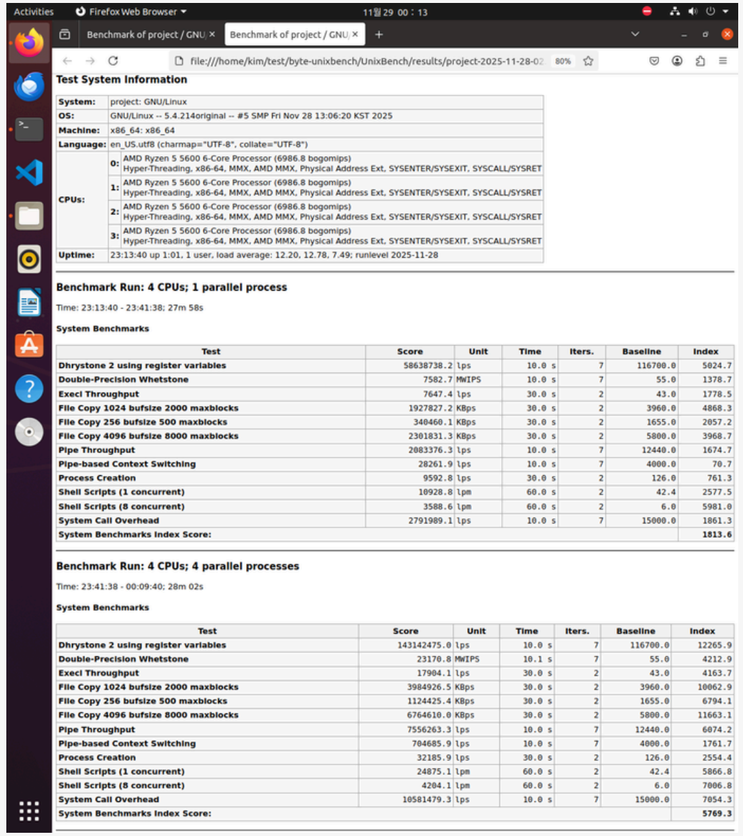
Test	Score	Unit	Time	Iters.	Baseline	Index
Dhrystone 2 using register variables	58507378.8	lps	10.0 s	7	116700.0	5013.5
Double-Precision Whetstone	7577.0	MWIPS	10.0 s	7	55.0	1377.6
Execl Throughput	7646.0	lps	30.0 s	2	43.0	1778.1
File Copy 1024 bufsize 2000 maxblocks	1934308.8	KBps	30.0 s	2	3960.0	4884.6
File Copy 256 bufsize 500 maxblocks	327449.5	KBps	30.0 s	2	1655.0	1978.5
File Copy 4096 bufsize 8000 maxblocks	2484965.6	KBps	30.0 s	2	5800.0	4284.4
Pipe Throughput	2089162.5	lps	10.0 s	7	12440.0	1679.4
Pipe-based Context Switching	28617.4	lps	10.0 s	7	4000.0	71.5
Process Creation	9555.8	lps	30.0 s	2	126.0	758.4
Shell Scripts (1 concurrent)	10750.9	lpm	60.0 s	2	42.4	2535.6
Shell Scripts (8 concurrent)	3550.5	lpm	60.0 s	2	6.0	5917.5
System Call Overhead	2847803.9	lps	10.0 s	7	15000.0	1898.5
System Benchmarks Index Score:						1819.9

Benchmark Run: 4 CPUs; 4 parallel processes

Time: 22:44:50 - 23:12:52; 28m 02s

System Benchmarks

Test	Score	Unit	Time	Iters.	Baseline	Index
Dhrystone 2 using register variables	143069323.1	lps	10.0 s	7	116700.0	12259.6
Double-Precision Whetstone	23279.8	MWIPS	10.1 s	7	55.0	4232.7
Execl Throughput	17884.4	lps	29.9 s	2	43.0	4159.2
File Copy 1024 bufsize 2000 maxblocks	4114748.6	KBps	30.0 s	2	3960.0	10390.8
File Copy 256 bufsize 500 maxblocks	1125029.5	KBps	30.0 s	2	1655.0	6797.8
File Copy 4096 bufsize 8000 maxblocks	7367825.3	KBps	30.0 s	2	5800.0	12703.1
Pipe Throughput	7567004.5	lps	10.0 s	7	12440.0	6082.8
Pipe-based Context Switching	706303.8	lps	10.0 s	7	4000.0	1765.8
Process Creation	32229.3	lps	30.0 s	2	126.0	2557.9
Shell Scripts (1 concurrent)	24868.3	lpm	60.0 s	2	42.4	5865.2
Shell Scripts (8 concurrent)	4269.1	lpm	60.0 s	2	6.0	7115.2
System Call Overhead	10630095.9	lps	10.0 s	7	15000.0	7086.7
System Benchmarks Index Score:						5839.9



Test System Information

System:	project: GNU/Linux
OS:	GNU/Linux -- 5.4.214original -- #5 SMP Fri Nov 28 13:06:20 KST 2025
Machine:	x86_64: x86_64
Language:	en_US.utf8 (charmap="UTF-8", collate="UTF-8")
CPUs: 0:	AMD Ryzen 5 5600 6-Core Processor (6986.8 bogomips) Hyper-Threading, x86-64, MMX, AMD MMX, Physical Address Ext, SYSENTER/SYSEXIT, SYSCALL/SYSRET
CPUs: 1:	AMD Ryzen 5 5600 6-Core Processor (6986.8 bogomips) Hyper-Threading, x86-64, MMX, AMD MMX, Physical Address Ext, SYSENTER/SYSEXIT, SYSCALL/SYSRET
CPUs: 2:	AMD Ryzen 5 5600 6-Core Processor (6986.8 bogomips) Hyper-Threading, x86-64, MMX, AMD MMX, Physical Address Ext, SYSENTER/SYSEXIT, SYSCALL/SYSRET
CPUs: 3:	AMD Ryzen 5 5600 6-Core Processor (6986.8 bogomips) Hyper-Threading, x86-64, MMX, AMD MMX, Physical Address Ext, SYSENTER/SYSEXIT, SYSCALL/SYSRET
Uptime:	23:13:40 up 1:01, 1 user, load average: 12.20, 12.78, 7.49; runlevel 2025-11-28

Benchmark Run: 4 CPUs; 1 parallel process

Time: 23:13:40 - 23:41:38; 27m 58s

System Benchmarks

Test	Score	Unit	Time	Iters.	Baseline	Index
Dhrystone 2 using register variables	58638738.2	lps	10.0 s	7	116700.0	5024.7
Double-Precision Whetstone	7582.7	MWIPS	10.0 s	7	55.0	1378.7
Execl Throughput	7647.4	lps	30.0 s	2	43.0	1778.5
File Copy 1024 bufsize 2000 maxblocks	1927827.2	KBps	30.0 s	2	3960.0	4868.3
File Copy 256 bufsize 500 maxblocks	340460.1	KBps	30.0 s	2	1655.0	2057.2
File Copy 4096 bufsize 8000 maxblocks	2301831.3	KBps	30.0 s	2	5800.0	3968.7
Pipe Throughput	2083376.3	lps	10.0 s	7	12440.0	1674.7
Pipe-based Context Switching	28261.9	lps	10.0 s	7	4000.0	70.7
Process Creation	9592.8	lps	30.0 s	2	126.0	761.3
Shell Scripts (1 concurrent)	10928.8	lpm	60.0 s	2	42.4	2577.5
Shell Scripts (8 concurrent)	3588.6	lpm	60.0 s	2	6.0	5981.0
System Call Overhead	2791989.1	lps	10.0 s	7	15000.0	1861.3
System Benchmarks Index Score:						1813.6

Benchmark Run: 4 CPUs; 4 parallel processes

Time: 23:41:38 - 00:09:40; 28m 02s

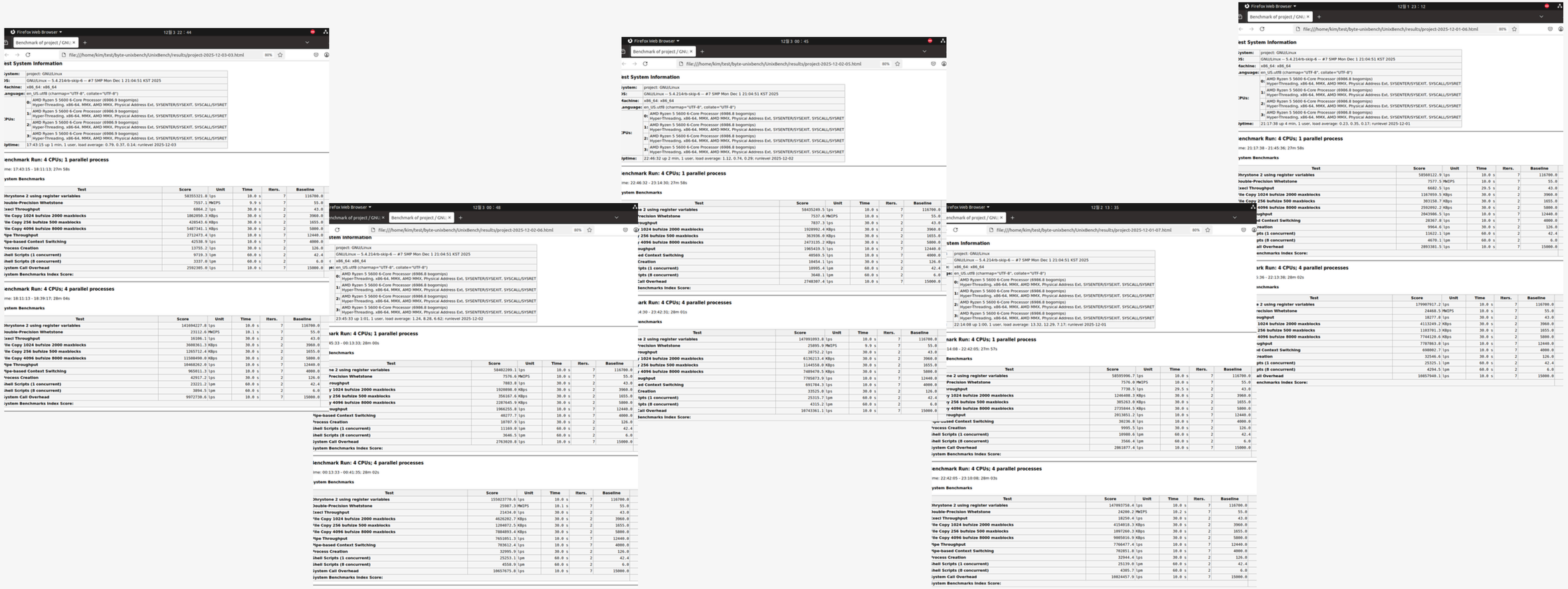
System Benchmarks

Test	Score	Unit	Time	Iters.	Baseline	Index
Dhrystone 2 using register variables	143142475.0	lps	10.0 s	7	116700.0	12265.9
Double-Precision Whetstone	23170.8	MWIPS	10.1 s	7	55.0	4212.9
Execl Throughput	17904.1	lps	30.0 s	2	43.0	4163.7
File Copy 1024 bufsize 2000 maxblocks	3984926.5	KBps	30.0 s	2	3960.0	10062.9
File Copy 256 bufsize 500 maxblocks	1124425.4	KBps	30.0 s	2	1655.0	6794.1
File Copy 4096 bufsize 8000 maxblocks	6764610.0	KBps	30.0 s	2	5800.0	11663.1
Pipe Throughput	7556263.3	lps	10.0 s	7	12440.0	6074.2
Pipe-based Context Switching	704685.9	lps	10.0 s	7	4000.0	1761.7
Process Creation	32185.9	lps	30.0 s	2	126.0	2554.4
Shell Scripts (1 concurrent)	24875.1	lpm	60.0 s	2	42.4	5866.8
Shell Scripts (8 concurrent)	4204.1	lpm	60.0 s	2	6.0	7006.8
System Call Overhead	10581479.3	lps	10.0 s	7	15000.0	7054.3
System Benchmarks Index Score:						5769.3

Appendix

UnixBenchmark Summary Screenshot

RB-SkipList 6th try



Limitations in Hybrid Architecture CPU

항목	Original	RB-Skiplist	Change Rate
System Index Score	9,516	6,048	▼ 36.4%
Process Creation (lps)	42,661	14,497	▼ 66.0%
Shell Scripts (8 con) (lpm)	7,917	4,794	▼ 39.4%
Pipe-based Context Switching	1,354,833	59,931	▼ 95.5%

<< Test Environment >>

[Host]

- CPU: Intel Core Ultra 7 155H
- # of processor cores: 16
- # of logical processors: 22
- Memory: 32GB

[Virtual Machine]

- # of processor cores: 12
- Memory: 8GB

Critical decrease in index score

- Hybrid architecture CPU use different types of cores (P-core, E-core, LP E-core)
- If E-core (slower than P-core) holds lock, P-core (faster core) has to wait until E-core finishes task
- Since level >= 3 nodes are inserted to RB-Tree, a lot of lock occurs
- P-core and E-core use separated cache system so that cache miss occurs frequently

UnixBenchmark Summary Description

1. The RB-Skiplist variants show competitive or improved performance compared to the Original RB-tree implement .

2. In particular, RB-skiplist 2,3 demonstrated clear gains in the System Benchmarks index Score, indicating the effectiveness of the hybrid skiplist design.

3. Both the single-process and multi-process runs exhibit consistent improvement trends, suggesting reduced PID lookup overhead

4. Performance variations across version appear to stem from different level distributions, but the overall trend validates the advantage of combining a skiplist with top-level RB indexing.