Operating Systems Project: Topic 1

Modify Linux Kernel Scheduler

Boyu Liu

225040138@link.cuhk.edu.cn

Mengxi Yang

225040130@link.cuhk.edu.cn

2026.3.10

Contribution

Mengxi Yang（60%）

Contribution In Detail：

- Data Structure Analysis
- WRR Design
- GUI Implementation
- GUI Demo Make
- Slides Making
- Presentation

Boyu Liu（40%）

Contribution In Detail：

- Environment Construction
- Lottery Design
- System Verification And Testing
- Slides Making
- Presentation

Outline

Outline

Objectives

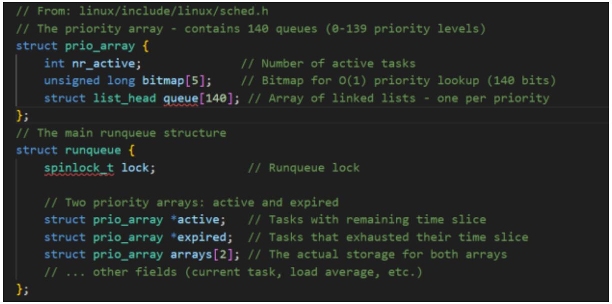
01 Gain a deep understanding of the working principles of the Linux process scheduler

02 Optimize the system's response to different workloads by implementing a custom scheduling policy

03 Specifically, the goal is to modify the kernel scheduling code to implement at least two different scheduling algorithms and demonstrate their behavioral characteristics using visualization tools.

Original Design

Before Linux kernel version-2.6.23, we use Priority Array to organize our task in system:

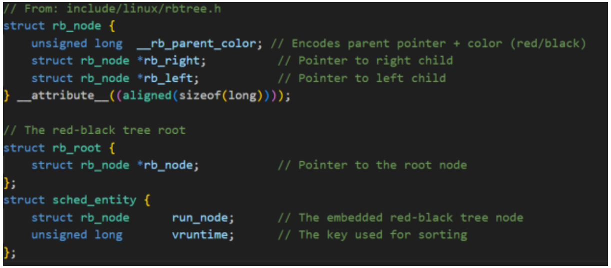


```c
// From: linux/include/linux/sched.h
// The priority array - contains 140 queues (0-139 priority levels)
struct prio_array {
    int nr_active;              // Number of active tasks
    unsigned long bitmap[5];    // Bitmap for O(1) priority lookup (140 bits)
    struct list_head queue[140]; // Array of linked lists - one per priority
};
// The main runqueue structure
struct runqueue {
    spinlock_t lock;             // Runqueue lock

    // Two priority arrays: active and expired
    struct prio_array *active;   // Tasks with remaining time slice
    struct prio_array *expired;  // Tasks that exhausted their time slice
    struct prio_array arrays[2]; // The actual storage for both arrays
    // ... other fields (current task, load average, etc.)
};
```

Design History II

Original Design

After that, they choose to use Red-Black Tree:



```
// From: include/linux/rbtree.h
struct rb_node {
    unsigned long  __rb_parent_color; // Encodes parent pointer + color (red/black)
    struct rb_node *rb_right;          // Pointer to right child
    struct rb_node *rb_left;           // Pointer to left child
} __attribute__((aligned(sizeof(long))));

// The red-black tree root
struct rb_root {
    struct rb_node *rb_node;           // Pointer to the root node
};
struct sched_entity {
    struct rb_node      run_node;      // The embedded red-black tree node
    unsigned long       vruntime;      // The key used for sorting
};
```

But Why?

Design Comparison(Time Complexity Analysis)

Operation	O(1) Scheduler - Priority Array	CFS Scheduler - Red-Black Tree	Winner
Pick Next Task	O(1) - Bitmap search	O(1) - Cached leftmost node	Tie
Enqueue Task	O(1) - Insert at tail of linked list	O(log N) - Tree insertion + rebalaance	O(1) Array
Dequeue Task	O(1) - Remove from linked list	O(log N) - Tree deletion + rebalance	O(1) Array
Priority Change	O(1) - Migrate between queues	O(log N) - Remove and re-insert	O(1) Array
Impact of N Processes	Constant - Independent of N	Grows with O(log N)	O(1) Array

Priority Array win among almost all operation complexities!

Design Comparison (System Stability Analysis)

But the huge system need： FAIR, FAIR, and FAIR !

Consideration	Explanation
Algorithmic Shift	From complex heuristics → simple mathematical model
Fairness First	Strict ordering by vruntime for true fairness
Scalability	O(log N) acceptable for large-scale servers
Maintainability	Unified logic removes special-case code
Future-Proof	Tree structure supports cgroups and group scheduling

So now we get a classic scheduler algorithm: CFS

The Principle Of CFS Scheduler

The Completely Fair Scheduler (CFS) implements an "ideal multi-tasking CPU" by tracking each process's virtual runtime (vruntime), which increases as the process runs. All runnable processes are stored in a red-black tree sorted by vruntime, with the smallest value on the left. The scheduler always picks the leftmost node—the process that has received the least CPU time—to run next.

When a process runs, its vruntime increases until it exceeds others, causing it to move rightward in the tree as another process takes its place at the left. Higher priority processes have their vruntime grow slower, naturally receiving more CPU time. This simple model achieves perfect fairness without complex heuristics.

Our Motivation

Why and What we do

As we have seen in past slides, CFS scheduler have various disadvantages in operation complexity.
When faced situations which have strict requirements for low-latency responding, we need faster scheduling. So we smoothly come up a idea:

Make Priority Array Great Again

We produced two scheduler algorithm(WRR and Lottery) in Linux kernel version-5.15.162

Outline

Overall architecture design

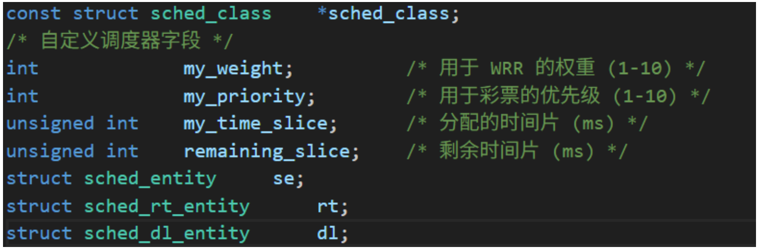
Linux Scheduling Framework

Based on the polymorphic design of Linux scheduling classes, we retained the original CFS scheduler while adding two new policies:

Priority	Scheduling Class	Policies	Description
Highest	stop_class	-	Migration/Shutdown kernel threads
	dl_class	SCHED_DEADLINE	Hard real-time, Earliest Deadline First
	rt_class	SCHED_FIFO/RR	POSIX real-time, static priority 0-99
	fair_class	SCHED_NORMAL	CFS, normal user processes (99%)
Lowest	idle_class	SCHED_IDLE	Idle thread when nothing to run

Core Data Structure Expansion

To support custom scheduling, we added the necessary fields to the `task_struct`:



```
const struct sched_class    *sched_class;
/* 自定义调度器字段 */
int                my_weight;          /* 用于 WRR 的权重 (1-10) */
int                my_priority;        /* 用于彩票的优先级 (1-10) */
unsigned int       my_time_slice;      /* 分配的时间片 (ms) */
unsigned int       remaining_slice;    /* 剩余时间片 (ms) */
struct sched_entity       se;
struct sched_rt_entity        rt;
struct sched_dl_entity        dl;
```

WRR Scheduling Implementation I

Design Principles：

WRR scheduling allocates time zones of varying lengths based on task priority and executes all tasks in a round-robin fashion. Higher-priority tasks receive longer time zones, ensuring they get more CPU time

Algorithm characteristics：

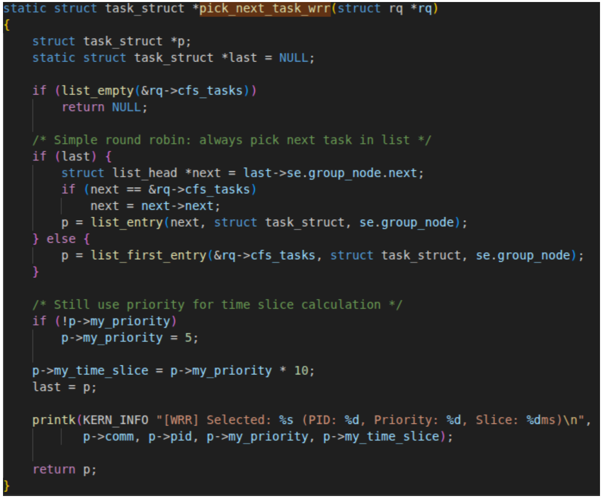
Time complexity: O(1) - Maintain the current task using static variables

Fairness: All tasks receive CPU time, so no one will starve.

Predictability: Deterministic scheduling, which facilitates analysis.

WRR Scheduling Implementation II

Core code implementation：



```c
static struct task_struct *pick_next_task_wrr(struct rq *rq)
{
    struct task_struct *p;
    static struct task_struct *last = NULL;

    if (list_empty(&rq->cfs_tasks))
        return NULL;

    /* Simple round robin: always pick next task in list */
    if (last) {
        struct list_head *next = last->se.group_node.next;
        if (next == &rq->cfs_tasks)
            next = next->next;
        p = list_entry(next, struct task_struct, se.group_node);
    } else {
        p = list_first_entry(&rq->cfs_tasks, struct task_struct, se.group_node);
    }

    /* Still use priority for time slice calculation */
    if (!p->my_priority)
        p->my_priority = 5;

    p->my_time_slice = p->my_priority * 10;
    last = p;

    printk(KERN_INFO "[WRR] Selected: %s (PID: %d, Priority: %d, Slice: %dms)\n",
           p->comm, p->pid, p->my_priority, p->my_time_slice);

    return p;
}
```

Lottery Scheduling Implementation I

Design Principles：

The lottery scheduling mechanism allocates a number of "lotteries" to each task in direct proportion to its priority. During each scheduling cycle, a winning lottery ticket is randomly drawn, and the task holding that ticket receives CPU time. Statistically, the CPU time a task receives is directly proportional to the number of lottery tickets it possesses.

Algorithm characteristics：

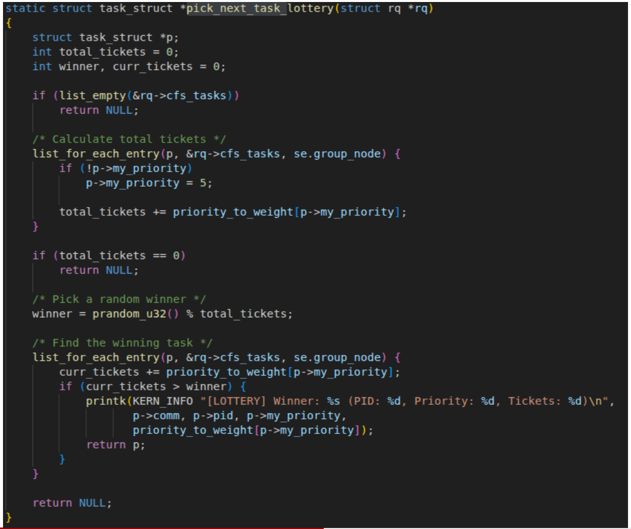
Time Complexity: O(n) - Requires traversing the task list to calculate the total number of votes and finding the winner.

Statistical Fairness: Under long-term operation, CPU time allocation is directly proportional to priority.

Randomness: Avoids performance issues caused by certain specific patterns.

Lottery Scheduling Implementation II

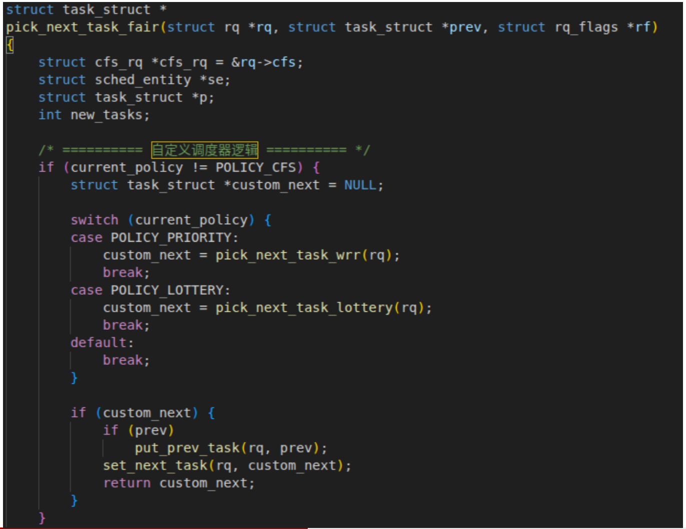
Core code implementation:



```c
static struct task_struct *pick_next_task_lottery(struct rq *rq)
{
    struct task_struct *p;
    int total_tickets = 0;
    int winner, curr_tickets = 0;

    if (list_empty(&rq->cfs_tasks))
        return NULL;

    /* Calculate total tickets */
    list_for_each_entry(p, &rq->cfs_tasks, se.group_node) {
        if (!p->my_priority)
            p->my_priority = 5;

        total_tickets += priority_to_weight[p->my_priority];
    }

    if (total_tickets == 0)
        return NULL;

    /* Pick a random winner */
    winner = prandom_u32() % total_tickets;

    /* Find the winning task */
    list_for_each_entry(p, &rq->cfs_tasks, se.group_node) {
        curr_tickets += priority_to_weight[p->my_priority];
        if (curr_tickets > winner) {
            printk(KERN_INFO "[LOTTERY] Winner: %s (PID: %d, Priority: %d, Tickets: %d)\n",
                   p->comm, p->pid, p->my_priority,
                   priority_to_weight[p->my_priority]);
            return p;
        }
    }

    return NULL;
}
```

Scheduling strategy switching mechanism

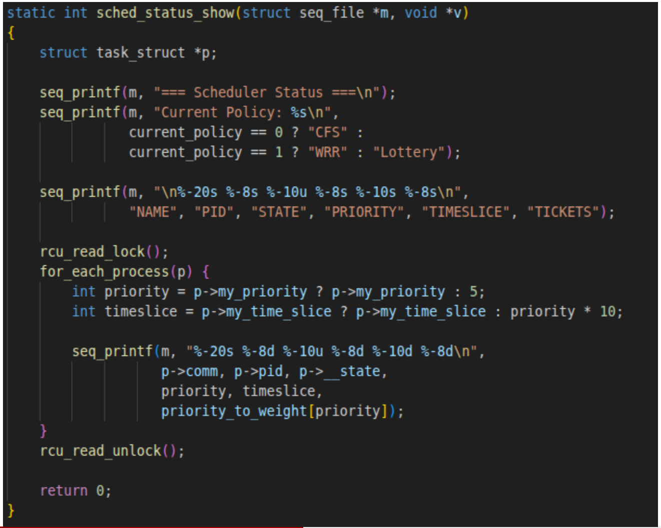
We added strategy selection logic to the pick_next_task_fair function:



```c
struct task_struct *
pick_next_task_fair(struct rq *rq, struct task_struct *prev, struct rq_flags *rf)
{
    struct cfs_rq *cfs_rq = &rq->cfs;
    struct sched_entity *se;
    struct task_struct *p;
    int new_tasks;

    /* ========== 自定义调度器逻辑 ========== */
    if (current_policy != POLICY_CFS) {
        struct task_struct *custom_next = NULL;

        switch (current_policy) {
        case POLICY_PRIORITY:
            custom_next = pick_next_task_wrr(rq);
            break;
        case POLICY_LOTTERY:
            custom_next = pick_next_task_lottery(rq);
            break;
        default:
            break;
        }

        if (custom_next) {
            if (prev)
                put_prev_task(rq, prev);
            set_next_task(rq, custom_next);
            return custom_next;
        }
    }
```

/proc interface implementation I

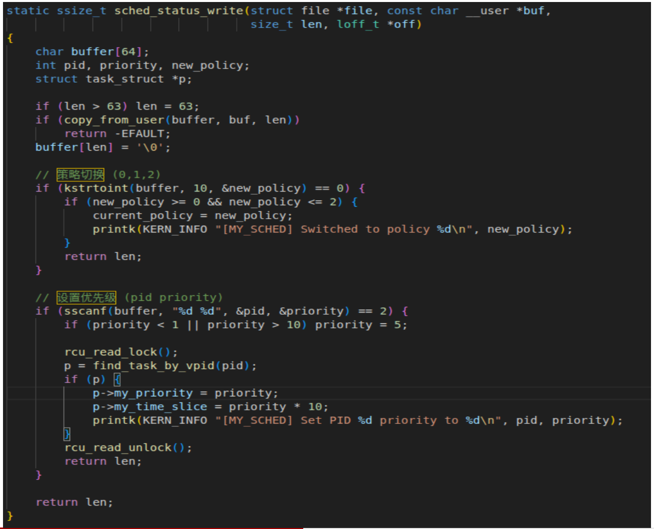
To interact with user space, we created the /proc/sched_status interface:



```c
static int sched_status_show(struct seq_file *m, void *v)
{
    struct task_struct *p;

    seq_printf(m, "=== Scheduler Status ===\n");
    seq_printf(m, "Current Policy: %s\n",
            current_policy == 0 ? "CFS" :
            current_policy == 1 ? "WRR" : "Lottery");

    seq_printf(m, "\n%-20s %-8s %-10u %-8s %-10s %-8s\n",
            "NAME", "PID", "STATE", "PRIORITY", "TIMESLICE", "TICKETS");

    rcu_read_lock();
    for_each_process(p) {
        int priority = p->my_priority ? p->my_priority : 5;
        int timeslice = p->my_time_slice ? p->my_time_slice : priority * 10;

        seq_printf(m, "%-20s %-8d %-10u %-8d %-10d %-8d\n",
                p->comm, p->pid, p->__state,
                priority, timeslice,
                priority_to_weight[priority]);
    }
    rcu_read_unlock();

    return 0;
}
```

/proc interface implementation II

To interact with user space, we created the /proc/sched_status interface:



```
static ssize_t sched_status_write(struct file *file, const char __user *buf,
                                  size_t len, loff_t *off)
{
    char buffer[64];
    int pid, priority, new_policy;
    struct task_struct *p;

    if (len > 63) len = 63;
    if (copy_from_user(buffer, buf, len))
        return -EFAULT;
    buffer[len] = '\0';

    // 策略切换 (0,1,2)
    if (kstrtoint(buffer, 10, &new_policy) == 0) {
        if (new_policy >= 0 && new_policy <= 2) {
            current_policy = new_policy;
            printk(KERN_INFO "[MY_SCHED] Switched to policy %d\n", new_policy);
        }
        return len;
    }

    // 设置优先级 (pid priority)
    if (sscanf(buffer, "%d %d", &pid, &priority) == 2) {
        if (priority < 1 || priority > 10) priority = 5;

        rcu_read_lock();
        p = find_task_by_vpid(pid);
        if (p) {
            p->my_priority = priority;
            p->my_time_slice = priority * 10;
            printk(KERN_INFO "[MY_SCHED] Set PID %d priority to %d\n", pid, priority);
        }
        rcu_read_unlock();
        return len;
    }

    return len;
}
```

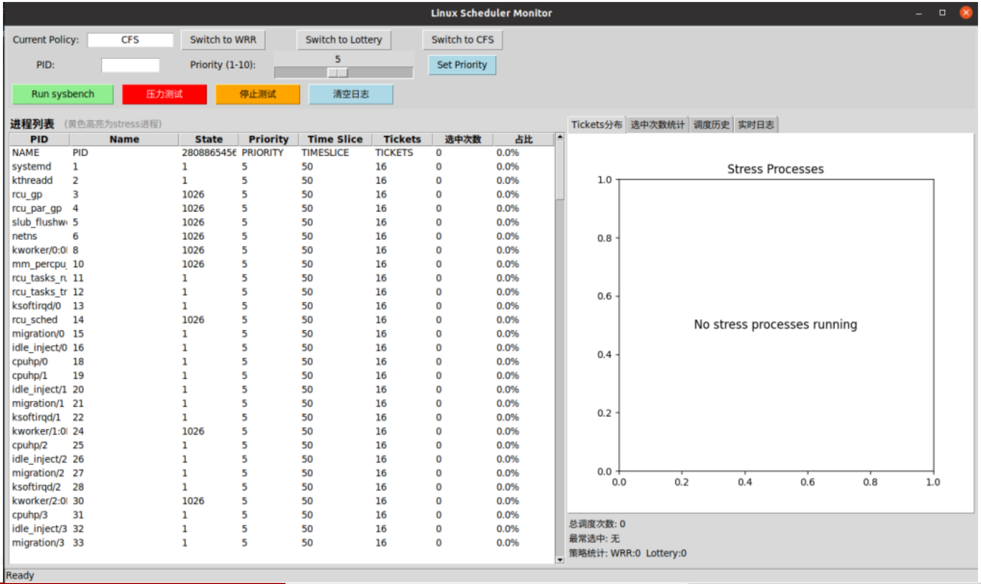
GUI Monitoring Tool Implementation

To visually demonstrate the scheduler's behavior, we developed a Python GUI tool.

Main functional modules：

Module	Function Description
Strategy Control	Real-time switching strategy for CFS/WRR/Lottery
Priority settings	Set the priority (1-10) for a specified PID.
Process list	Display the PID, name, status, priority, time zone, and tickets of all processes.
Tickets distribution	The pie chart shows the Ticket allocation status of the stress process.
Selection count statistics	Count the number of times each process is scheduled.
Dispatch history	Show the most recent scheduling decision record
Real-time logs	Kernel log output

GUI Monitoring Tool Desgin



进程列表 (黄色高亮为stress进程)

PID	Name	State	Priority	Time Slice	Tickets	选中次数	占比
NAME	PID	280886545€	PRIORITY	TIMESLICE	TICKETS	0	0.0%
systemd	1	1	5	50	16	0	0.0%
kthreadd	2	1	5	50	16	0	0.0%
rcu_gp	3	1026	5	50	16	0	0.0%
rcu_par_gp	4	1026	5	50	16	0	0.0%
slub_flushw	5	1026	5	50	16	0	0.0%
netns	6	1026	5	50	16	0	0.0%
kworker/0:0l	8	1026	5	50	16	0	0.0%
mm_percpu_	10	1026	5	50	16	0	0.0%
rcu_tasks_rı	11	1	5	50	16	0	0.0%
rcu_tasks_tr	12	1	5	50	16	0	0.0%
ksoftirqd/0	13	1	5	50	16	0	0.0%
rcu_sched	14	1026	5	50	16	0	0.0%
migration/0	15	1	5	50	16	0	0.0%
idle_inject/0	16	1	5	50	16	0	0.0%
cpuhp/0	18	1	5	50	16	0	0.0%
cpuhp/1	19	1	5	50	16	0	0.0%
idle_inject/1	20	1	5	50	16	0	0.0%
migration/1	21	1	5	50	16	0	0.0%
ksoftirqd/1	22	1	5	50	16	0	0.0%
kworker/1:0l	24	1026	5	50	16	0	0.0%
cpuhp/2	25	1	5	50	16	0	0.0%
idle_inject/2	26	1	5	50	16	0	0.0%
migration/2	27	1	5	50	16	0	0.0%
ksoftirqd/2	28	1	5	50	16	0	0.0%
kworker/2:0l	30	1026	5	50	16	0	0.0%
cpuhp/3	31	1	5	50	16	0	0.0%
idle_inject/3	32	1	5	50	16	0	0.0%
migration/3	33	1	5	50	16	0	0.0%

No stress processes running

总调度次数: 0
最常选中: 无
策略统计: WRR:0 Lottery:0

Outline

DEMO

Performance Comparison Test : Sysbench In Detail

Scheduler	Test	events/sec	Avg Latency (ms)	Max Latency (ms)
CFS	1	1907.26	0.52	3.00
	2	1901.92	0.52	1.51
	3	1906.58	0.52	1.51
Lottery	1	1893.16	0.53	1.54
	2	1889.12	0.53	1.99
	3	1895.84	0.52	1.48
WRR	1	1910.16	0.52	1.63
	2	1901.92	0.52	1.41
	3	1906.58	0.52	1.51

Performance Comparison Analysis : Sysbench In General

Metric	CFS	Lottery	WRR
Throughput	1905.25	1892.71	1906.22
Avg Latency (ms)	0.52	0.53	0.52
Max Latency (ms)	2.01	1.67	1.52

Key Observations:

WRR scheduler outperforms both CFS and Lottery by delivering higher throughput while significantly reducing maximum latency. The deterministic weighted round-robin approach proves more suitable for this workload than CFS's fair-share model or Lottery's randomized selection.

Outline

Problems Encountered And Solutions I

Scheduler lag issue

Under high load stress testing, WRR scheduling caused the system to freeze, but it returned to normal after the test ended.

Cause Analysis:

- The WRR implementation traverses the entire task list for each scheduling iteration, resulting in a time complexity of O(n).
- It does not properly handle the case where time runs out.
- It lacks a pre-calculation mechanism.

Solution:

- Increase time zone management to reduce scheduling frequency
- Pre-calculate total weight to avoid repeated traversal
- Use static pointers to maintain the current task, achieving O(1) selection

Problems Encountered And Solutions

A sad story : the initrd is too big

When you have done all code modification and compilation, you need to make a new initrd image like initrd.img-5.15.162.

However, if you encounter:

Error 24: Write error : cannot write compressed block E: mkinitramfs failure cpio 141 lz4 -9 -l 24

You need to check:

```
root@mxx:~# ls -lh /boot/initrd.img-5.15.162
-rw-r--r-- 1 root root 821M Feb 25 13:31 /boot/initrd.img-5.15.162
```

If this file is bigger than 821M, your new kernel can not start up, because this image is too big.

But why?

A bug in the initramfs microcode handling script caused the AMD microcode file to be repeatedly included or corrupted during generation, bloating the initramfs while leaving out essential system modules and files.

Problems Encountered And Solutions III

Strong Suggestion

If your cpu infrastructure is based on AMD, just do it :

# Forbid microcode definitely:

```
echo "EARLYMICROCODE=n" >> /etc/initramfs-tools/initramfs.conf
echo "MODULES=dep" >> /etc/initramfs-tools/initramfs.conf
```

Believe me, this setting could help you to save at least six hours!

Outline

Summary and Outlook

Project Achievements

- Successfully implemented two scheduling algorithms: WRR and Lottery scheduling.
- Developed a complete monitoring tool: a Python GUI for real-time display of scheduling status.
- Performance comparison analysis: differences in throughput and latency among the three strategies.
- Stable operation: thorough testing showed no kernel panics.

Innovations

- Dynamic Strategy Switching: Seamless switching of scheduling strategies at runtime
- Real-time Priority Adjustment: Dynamically set via the /proc interface
- Visualized Tickets Distribution: Intuitively demonstrates the fairness of lottery scheduling
- Complete Monitoring System: Includes selection count statistics and scheduling history

Reference

1.Linux Kernel Documentation and Archives. https://www.kernel.org/doc/html/v5.12/
2. C. S. Wong, I. K. T. Tan, R. D. Kumari, J. W. Lam and W. Fun, "Fairness and interactive performance of O(1) and CFS Linux kernel schedulers," 2008 International Symposium on Information Technology, Kuala Lumpur, Malaysia, 2008, pp. 1-8, doi: 10.1109/ITSIM.2008.4631872.
3. Waldspurger, Carl A. and William E. Weihl. “Lottery scheduling: flexible proportional-share resource management.” USENIX Symposium on Operating Systems Design and Implementation (1994).
4. Weighted Round Robin Scheduling. (2006). Linux Virtual Server Project. http://zh.linuxvirtualserver.org/comment/101604
5. Schneider, V. (2020). Re: [RFC PATCH 2/3] docs: scheduler: Add scheduler overview documentation. Linux Kernel Mailing List. https://lkml.iu.edu/hypermail/linux/kernel/2004.0/00643.html
6. Linux Kernel Mailing List Glossary. KernelNewbies. https://kernelnewbies.org/KernelGlossary