COL331 Assignment 2

Anubhav Pandey
2022CS51136

April 10, 2025

Table of Contents

1 Introduction

This report presents the implementation of a custom gang scheduling policy (`SCHED_GANG`) in the Linux kernel. Gang scheduling is designed for tightly-coupled parallel applications in high-performance computing (HPC), where multiple threads must execute simultaneously on separate CPUs. The scheduler ensures coordinated execution and collective preemption of gang threads using a new scheduling class and associated system calls. All these implementation is done for ARM64 architecture and developed on MacOS (UTM).

2 Design Overview

The design of the gang scheduler introduces a new scheduling class `gang_sched_class`, integrated between the RT and FAIR classes. The gang policy facilitates:

- Simultaneous execution of all threads in a gang.
- Coordinated preemption using inter-processor interrupts (IPIs).
- Per-CPU gang runqueues for isolated management.
- Lifecycle management via three custom system calls.

3 Kernel Code Modifications

3.1 Syscall table

In `arch/arm64/tools/syscall_64.tbl`:

```
472 common  register_gang        sys_register_gang
473 common  exit_gang            sys_exit_gang
474 common  list_gang            sys_list_gang
```

3.2 Core Integration

The following kernel files were modified:

- `kernel/sched/core.c` – Registered the `gang_sched_class`.

```
#define SCHED_GANG 8
    ...
BUG_ON(!sched_class_above(&dl_sched_class, &rt_sched_class));
BUG_ON(!sched_class_above(&rt_sched_class, &fair_sched_class));
BUG_ON(!sched_class_above(&rt_sched_class, &gang_sched_class)); //<--
BUG_ON(!sched_class_above(&gang_sched_class, &fair_sched_class));  //<--
    BUG_ON(!sched_class_above(&fair_sched_class, &idle_sched_class));



    ...
```

- `include/uapi/linux/sched.h` – Added `SCHED_GANG` as identifier.
- `kernel/sched/Makefile` – Linked new gang scheduler files.

3.3 New Files Added

- `gang.c`, `gang.h` – Core scheduling logic and data structures.
- `gang_syscalls.c` – Handlers for gang management system calls.

4 Gang Scheduler Logic

4.1 Key Structures

- gang_task – Per-task metadata in a gang.
- gang_struct – Tracks all tasks in a gang; stores gang ID, leader, exec time.
- gang_rq – Per-CPU runqueue with leader_rq, member_rq, and gang ID context.

4.2 Scheduling Operations

- enqueue_task_gang – Places leader/member into runqueue.
- pick_next_task_gang – Prioritizes leaders, schedules synchronized members.
- put_prev_task_gang – Handles task rotation and preemption logic.
- wake_up_gang() – Broadcasts IPI to all gang member CPUs.

4.3 Context Switching

Upon leader task selection, an IPI is issued to wake member threads. CPUs check their gang_rq.next field to switch context to the appropriate gang member. Preemption of the leader triggers rescheduling of all gang members for strict lock-step execution.

4.4 Modification to vmlinux.lds.h

To ensure that the gang scheduling class is properly recognized and linked during kernel build time, a modification was made to the linker script header file located at include/asm-generic/vmlinux.lds.h. This file controls how kernel objects are organized in memory and how different scheduling classes are registered during kernel initialization.

Context and Role The Linux kernel uses a predefined order of scheduling classes, each identified by section markers such as __rt_sched_class, __fair_sched_class, and others. These markers are resolved during the linking phase by the kernel's linker script, and they determine the hierarchy and initialization order of schedulers in sched_init().

Why Modify vmlinux.lds.h? By default, the kernel linker does not know about the new SCHED_GANG class. Without proper declaration in vmlinux.lds.h, the gang scheduler would either not be initialized or would be incorrectly positioned in the scheduling class hierarchy.

To fix this, a new entry was inserted into the sched_class section of vmlinux.lds.h:

- *__gang_sched_class was added directly after the real-time scheduler (*__rt_sched_class) and before the fair scheduler (*__fair_sched_class).

This ensures that:

1. The gang scheduler is compiled into the final kernel image.
2. The gang scheduler appears in the correct order relative to other scheduling classes.
3. The scheduler hierarchy check in sched_init() does not fail due to missing or misordered entries.

```
    *(__stop_sched_class)        \
    *(__dl_sched_class)          \
    *(__rt_sched_class)          \
    *(__gang_sched_class)        \ //<-- here
    *(__fair_sched_class)        \
    *(__ext_sched_class)         \
    *(__idle_sched_class)        \
```

Final Result With this change, the gang scheduler is treated as a first-class scheduling class by the kernel. It is linked, initialized, and placed correctly between the RT and CFS classes, which matches its intended priority level.

This subtle yet critical change ensures that the kernel's scheduling framework can accommodate and delegate to the newly defined `gang_sched_class` at runtime.

5 System Calls

Three new syscalls were introduced:

`sys_register_gang(int pid, int gangid, int exec_time)`

```
   ...
    g_task->task = task;
    g_task->exec_time = exec_time;
    g_task->gang_id = gangid;
    g_task->gang = gang;
    task_lock(task);
    task->gang_id = gangid;
    g_task->cpu = task_cpu(task);
    task->gang_task = g_task;
    task_unlock(task);
    if(g_task && !g_task->gang) g_task->gang = gang;
    INIT_LIST_HEAD(&g_task->list);
    INIT_LIST_HEAD(&g_task->rq_node);
    list_add_tail(&g_task->list, &gang->list);
    gang->num_tasks;

    if (gang->leader_cpu == -1) {
        gang->leader_cpu = task_cpu(task);
        gang->leader_task = task;
    }
...
}
```

This basically registers the process, and stores its gang related information in all_gang_list.

`sys_exit_gang(int pid)`

```
 ...
rcu_read_lock();
task = find_task_by_vpid(pid);
if(task){
    if (task && task->gang_id==-1){
        rcu_read_unlock();
        return -22;
    }

}
rcu_read_unlock();

spin_lock(&all_gang_lock);
gang = find_gang(task->gang_id);
```

```
spin_unlock(&all_gang_lock);
if(!gang){
    return -22;
}

if(task)
sched_setscheduler(task, SCHED_NORMAL, &param);

...
        if (g_task->task->pid == pid) {
            task = g_task->task;
            g_task->gang = NULL;
            list_del(&g_task->list);

            gang->num_tasks--;
            if(task == gang->leader_task){
                gang->leader_cpu = -1;
                gang->leader_running = false;

            }
            ...

            if (gang->num_tasks == 0) {
                list_del(&gang->gang_list);
                kfree(gang);
            }

            ...
```

- Exits a task from the gang and triggers gang cleanup if last member.

sys_list_gang(int gangid, int* pids)

```
    ...
    list_for_each_entry(g_task, &gang->list, list) {
        if (count < 100)
            k_pids[count++] = g_task->task->pid;
    }
    ...
```

- Lists all active tasks in a given gang ID.

6 Gang Runqueue Architecture

Each CPU's gang_rq maintains:

- leader_rq – Queue of leaders.
- member_rq – Queue of members.
- curr, next, prev – Context for synchronized transitions.

This design modularly isolates gang scheduling from existing CFS/RT runqueues.

```
struct gang_rq {
    struct list_head leader_rq;
    struct list_head member_rq;
    int prev;
    int curr;
    int next;
    int counter;
};
```

This was done in kernel/sched/sched.h.

6.1 Gang Runqueue Internals and Motivation

The gang scheduler introduces a custom per-CPU structure, `gang_rq`, which manages tasks belonging to registered gangs. Unlike traditional schedulers that maintain a single runqueue per CPU, `gang_rq` is designed with two separate lists:

- **leader_rq:** A runqueue exclusively for leader tasks of gangs. Each gang has exactly one leader, determined when the first task registers with that gang ID.
- **member_rq:** A runqueue for all remaining tasks (members) of the gang, which are scheduled only after the leader initiates gang-wide wake-up.

This separation addresses the fundamental synchronization challenge inherent in gang scheduling: all gang tasks must start and pause execution together to avoid inter-thread latency or stalls.

Why Two Runqueues? Using separate runqueues for leaders and members serves two main purposes:

1. **Leader-Driven Activation:** In gang scheduling, the leader is responsible for triggering the rest of the gang. By isolating leaders in a dedicated runqueue, we can easily prioritize their selection via `pick_next_task()`, ensuring that a leader is always scheduled before its members.
2. **Simplified Synchronization:** Gang members must not run before their leader. If both leader and members were queued together, we would need additional logic to prevent premature scheduling of member tasks. Instead, by maintaining a distinct `member_rq`, we delay their selection until an IPI sets the `gang_rq.next` field, authorizing the CPU to dequeue a member for execution.

Gang Context Transition When a leader is scheduled, it triggers an IPI broadcast using `wake_up_gang()`, which sets the `gang_rq.next` field on all involved CPUs. This field instructs the corresponding CPU to look into its `member_rq` for the relevant gang ID. The member task is then dequeued and scheduled, ensuring that all gang members execute in unison.

Additional State Tracking The `gang_rq` structure also maintains three integer fields—`prev`, `curr`, and `next`—which help track transitions between gangs and detect rescheduling needs. These markers are reset after each synchronized execution cycle to prepare for the next gang's turn.

Scalability and Safety This two-queue model avoids contention, enables predictable gang wakeups, and reduces complexity in scheduling decisions. It ensures that multiple CPUs, each with its own `gang_rq`, can independently maintain local consistency while respecting global gang semantics coordinated by the leader's core.

7 Core Scheduling Functions of `SCHED_GANG`

The gang scheduler class defines the behavior of gang tasks using a set of core scheduling hooks implemented in the kernel. These functions are invoked by the core kernel scheduler during the scheduling lifecycle of a task. The key functions are: `enqueue_task`, `dequeue_task`, `pick_next_task`, `put_prev_task`, and `task_tick`. Each of these plays a distinct role in ensuring coordinated execution of gang tasks across CPUs.

7.1 Enqueue Task

The `enqueue_task()` function is responsible for placing a gang task into the appropriate runqueue. The logic differentiates between leader and member threads based on the CPU to which the task is affinitized.

- The function begins by identifying the gang ID and verifying that the task belongs to a registered gang.
- It then locates the relevant `gang_task` object and determines whether the task is the gang leader (based on the first thread registered).
- The task is inserted into either the `leader_rq` or the `member_rq` list within the per-CPU `gang_rq`.
- Finally, the system-wide runnable task count is updated.

```
    ...
    if (gang && gang->leader_cpu != cpu) {
        INIT_LIST_HEAD(&g_task->rq_node);
       list_add_tail(&g_task->rq_node, &g_rq->member_rq);
    } else {
        INIT_LIST_HEAD(&g_task->rq_node);
        list_add_tail(&g_task->rq_node, &g_rq->leader_rq);
   }

    add_nr_running(rq, 1);
    ...
```

This dual-queue strategy enables the gang scheduler to manage gang state explicitly and separate scheduling logic for leader and member tasks.

7.2 Dequeue Task

The `dequeue_task()` function removes a task from its runqueue. This typically occurs when the task goes to sleep, blocks on I/O, or exits.

- The function searches both the `leader_rq` and `member_rq` for a matching `gang_task` structure.
- Upon finding the task, it is removed from the list and the runnable count is decremented.
- The operation is logged for debugging and tracing purposes.

```
   list_for_each_entry(g_task, &g_rq->leader_rq, rq_node) {
       if (g_task->task == p) {
           list_del_init(&g_task->rq_node);
           done = true;
           break;
       }
   }
   if(done){
       ...
       if (gang && gang->leader_task == p) {
           gang->leader_cpu = -1;
           gang->leader_running = false;
       }
       spin_unlock(&all_gang_lock);
   }
   if (!done) {
       list_for_each_entry(g_task, &g_rq->member_rq, rq_node) {
           if (g_task->task == p) {
               list_del_init(&g_task->rq_node);
               done = true;
               break;
   }
   ...
```

This ensures that only valid, running tasks remain in the queue, maintaining consistency of the runqueue state.

7.3 Pick Next Task

The pick_next_task() function is central to the operation of the gang scheduler. It decides which task should be executed next on the current CPU.

- If the CPU has received a gang wake-up IPI, the gang_rq.next field is set, and the scheduler attempts to dequeue a gang member from the member_rq.
- If no gang member is found, or if the CPU was not signaled for a gang wake-up, the scheduler attempts to pick a gang leader from the leader_rq.
- Once selected, the task is returned to the core scheduler, and the gang ID is cleared from the next field.

```
if (g_rq->next != -1 ) { //some leader ran and set this to non -1 value
       g_rq->curr = g_rq->next;
       g_rq->next = -1;

       ...

       if(g_task){
           spin_lock(&all_gang_lock);
           gang = find_gang(g_task->gang_id);
           spin_unlock(&all_gang_lock);

           if(gang){

               if(gang && gang->leader_running){
                   if (list_empty(&g_rq->member_rq)) return NULL;
                   if(next_task) return next_task;
               }else{
                   spin_lock(&all_gang_lock);
                   if(gang && gang->leader_cpu == -1 && next_task){
                       ... //if leader exist, change this to leader
                       if(g_task){
                           list_del_init(&g_task->rq_node);
                           INIT_LIST_HEAD(&g_task->rq_node);
                           list_add_tail(&g_task->rq_node, &g_rq->leader_rq);
                       }
                       ...
                   }
...

   if (list_empty(&g_rq->leader_rq)) return NULL;

   list_for_each_entry(g_task, &g_rq->leader_rq, rq_node) {
       if (g_task && g_task->task) {
           next_task = g_task->task;
           temp = g_task;
           break;
       }
   }

   if (temp) {
       g_task = temp;
       list_del_init(&g_task->rq_node);
       INIT_LIST_HEAD(&g_task->rq_node);
       list_add_tail(&g_task->rq_node, &g_rq->leader_rq);
       // for FIFO logic
   }

   int gang_id;
   if(next_task && g_task) {
       gang_id = g_task->gang_id;
       gang = g_task->gang;

       if(next_task && gang) gang->leader_running = true;
       if(next_task && g_task) wake_up_gang(g_task);

       return next_task;
   }
...
```

This approach guarantees that gang members are scheduled only after the leader initiates coordinated execution.

7.4 Put Previous Task

The put_prev_task() function is invoked at the end of a task's time slice or during a context switch. It allows the scheduler to perform cleanup or trigger necessary events.

- If the outgoing task is a gang leader, the function triggers a preemption for all gang members using the preempt_entire_gang() helper.
- The function resets the internal state of the gang_rq, indicating that the leader is no longer running.
- For member tasks, it sets the next gang ID field so that another task from the same gang may be scheduled.

```
    spin_lock(&all_gang_lock);
    rcu_read_lock();
    if(p->gang_task) gang = p->gang_task->gang;
    rcu_read_unlock();


    if (gang && gang->leader_cpu == rq->cpu) {
        gang->leader_running = false;
        spin_unlock(&all_gang_lock);

    }else{
        spin_unlock(&all_gang_lock);
        struct gang_rq *g_rq = &rq->gang_runqueue;
        if(g_rq->next==-1){
            g_rq->next = p->gang_id;
        }

    }
```

This ensures synchronized preemption: either the entire gang runs together, or none do.

7.5 Task Tick

The task_tick() function is invoked on every timer interrupt for the currently executing task. It typically handles quantum expiration and fairness policies.

- In SCHED_GANG, this function logs the occurrence of the tick but does not enforce time-sharing within a gang.
- Since gang tasks are assumed to be preempted together and share execution time, no additional quantum logic is required here.
- However, future implementations could enhance this to support time-based resource enforcement or gang-level quotas.

Currently, task_tick() serves a passive role and may be leveraged for monitoring or custom signaling in future work.

Together, these functions define the lifecycle of a gang-scheduled task and enable coordinated execution, preemption, and termination. They ensure that the gang semantics—particularly simultaneous start and stop—are enforced at all levels of the kernel's scheduling pipeline.

8 Preemption and Synchronization

- **Preemption:** When a leader is preempted, IPIs reschedule all member threads, from task tick after regular task ticks.
- **Locking:** all_gang_lock protects global gang list and ensures consistent gang lifecycle.
- **IPI handler:** Uses gang_rq.next for coordinated gang wakeups.

9 Task Lifecycle

- First thread registering to a gang is leader.
- Threads are affinitized to distinct CPUs using sched_setaffinity.
- When all threads exit via exit_gang, the gang is removed.

9.1 Kprobe on do_exit for Cleanup

To ensure that processes are cleanly removed from global tracking structures upon termination, a kprobe was registered on the do_exit() kernel function. This mechanism is critical to maintain consistency and avoid stale entries in both the gang scheduler and resource tracking subsystems.

Motivation In the Linux kernel, a process may exit unexpectedly due to signals, exceptions, or normal termination. Without explicit cleanup, entries in the global all_gang_list and monitored_list would persist indefinitely, leading to memory leaks, incorrect scheduling behavior, and potential kernel panics. A synchronous cleanup mechanism was thus essential to handle such lifecycle events.

Kprobe Mechanism A kprobe is a dynamic instrumentation facility that allows handlers to be inserted at arbitrary kernel instruction addresses or symbol names. In this implementation, a pre-handler was installed on the do_exit function:

- When any process exits, the pre-handler is triggered before the actual body of do_exit() executes.
- The handler retrieves the exiting process's PID using current->pid.
- It invokes __exit_gang(pid) to remove the task from the global gang list.

Implementation Details The kprobe is defined and registered in a dedicated kernel module named resource_tracker_cleaner.c. Upon module initialization:

- A kprobe structure is configured with .symbol_name = "do_exit".
- The handler_pre() function is assigned as the probe's pre-handler.
- The probe is registered using register_kprobe().

On module exit, the probe is unregistered to avoid dangling instrumentation using unregister_kprobe().

Safety and Isolation To avoid race conditions, the cleanup logic uses spinlocks around global list modifications and employs RCU-safe operations to ensure no pointer dereferencing occurs after deletion. In addition, reference counting via `get_task_struct()` ensures task structures are not freed mid-cleanup.

10 Challenges and Design Decisions

- Designing scalable per-CPU gang runqueues.
- Affinitizing threads to prevent scheduling imbalance.
- Ensuring clean preemption of all gang tasks upon leader preemption.
- Handling edge cases (early exits, oversubscription, duplicate IDs).
- Ensuring synchronization using proper locking and atomic updates.