[COL331] Assignment 2: Gang Scheduling

Hemanshu Garg — 2022CS11090

Implementing the Gang Scheduling Class

In general about adding a scheduler/how schedule works in the kernel

- There is runqueue `rq` of each cpu/core which has runqueues each scheduling class which further have schedulable entities of that sched class.
- Essentially the `__schedule` function in linux/kernel/sched/core.c is called for scheduling which after some checks calls the `__pick_next_task` of each sched class in the `rq` in the order as decided by the `for_each_class` macro, whose definition leads to linux/include/asm-generic/vmlinux.lds.h file, here our new sched class can added at the desired priority order.
- Also there is the `sched_init` function here which verifies the order of scheduling classes and initilizes elements of the `struct rq` of each CPU, for example `init_sched_cfs`
- `pick_next_task` function of each class returns the task struct of which process to run. The kernel also calls functions `enqeueue, dequeue, task_tick, put_prev_task` etc at various times.
- In order to add our own scheduling class, essentially we need to have it first be part of the system by adding it in the vmlinux.lds.h, the `rq` and `sched_init` and other kinds of places where our sched class policy becomes a new valid policy etc and then Implementing all the required function pointers of a `SCHED_CLASS` by adding and leverging our data structures of scheduable entitites kind in `task_struct`;

Design for our Gang Scheduling Class: Case single Gang

- Several threads/gangs will first set their affinity to distinct cores and then use the `sys_register_gang` system call to register themselves. The first thread to register itself will be chosen as the gang governer and its core as the gang governer core.
- Every `QUANTA` amount of milliseconds the gang governer will send IPI's for calling scheduler to the cores on which its gang is running to ensure synchronizations overtime.
- Only when gang goverener goes to sleep the other threads will stop running their execution completely.
- When either time limit set exceeds or a thread exits itself from gang, its scheduling class is changed down to cfs.
- If governer exits from the gang before other threads have, a new governer will be chosen from the still existing gang.

Design for multiple Gangs

- There are two scenarios here, one is that if any thread of a gang is overlapping a core with any other gang's thread. The other scenario being two gangs are completely disjoint.
- In the former scenario our we would alternate between which gang will be scheduled every quanta.

- In the latter we would ideally want both gangs to run concurrently.
- These functionality can be achieved using the idea of a global governer which would decide every quanta which gang's governer (local) be allowed to send the IPI's. Using this we can handle both cases, as with global governer can maintain masks of each cpu so that we can run either one gang or concurrently.

Kernel Implementation of the Gang Scheduling Class

linux/include/vmlinux.lds.h

Adding our sched class to the Hierarchy.

```
   #define SCHED_DATA        \
  STRUCT_ALIGN();         \
  __sched_class_highest = .;  \
  *(__stop_sched_class)      \
  *(__dl_sched_class)     \
  *(__rt_sched_class)     \
  *(__gang_sched_class)      \
  *(__fair_sched_class)      \
  *(__idle_sched_class)      \
  __sched_class_lowest = .;
```

linux/include/uapi/linux/sched.h

Define our new policy

```
   /*
 * Scheduling policies
 */
#define SCHED_NORMAL  0
#define SCHED_FIFO    1
#define SCHED_RR   2
#define SCHED_BATCH   3
/* SCHED_ISO: reserved but not implemented yet */
#define SCHED_IDLE    5
#define SCHED_DEADLINE  6
#define SCHED_GANG 7
```

linux/kernel/sched/sched.h

Make our policy a valid policy.

```
    static inline int gang_policy(int policy)
{
   return policy == SCHED_GANG;
}
static inline bool valid_policy(int policy)
{
   return idle_policy(policy) || fair_policy(policy) ||
      rt_policy(policy) || dl_policy(policy) || gang_policy(policy);
}
```

Define our struct gang runqeuue

```
    struct gang_rq
{

   unsigned long rqll_size;
   int cur_entity_index;
   int cur_running_gang;
   struct list_head sentinal_grq_head;
};
```

Add to the struct rq with the others

```
     struct cfs_rq   cfs;
     struct gang_rq  gang;
     struct rt_rq    rt;
     struct dl_rq    dl;
```

extern the gang_sched_class and init of gang rq to implement it later in one place in gang.c

```
    extern const struct sched_class stop_sched_class;
    extern const struct sched_class dl_sched_class;
    extern const struct sched_class rt_sched_class;
    extern const struct sched_class gang_sched_class;
    extern const struct sched_class fair_sched_class;
    extern const struct sched_class idle_sched_class;
    /*later in the file*/
    extern void init_cfs_rq(struct cfs_rq *cfs_rq);
    extern void init_rt_rq(struct rt_rq *rt_rq);
    extern void init_dl_rq(struct dl_rq *dl_rq);
    extern void init_gang_rq(struct gang_rq* gang_rq);
```

linux/include/linux/sched.h

Here define the sched gang entity that will be used the rq , and in order to maintain it initialize it in the task_struct like the other sched class entities.

```
struct sched_gang_entity
    {
        int gangid;
        int pid;
        struct task_struct* p;
        int* gang_gov_pid;
        bool* gov_exists;
        cpumask_t* cpumask;
      u64 start_time;
      u64 exec_time;
      u64 max_exec_time;
        struct list_head gang_run_list;
    };
struct sched_entity { ...

/*later in task struct*/
   struct sched_entity   se;
   struct sched_rt_entity  rt;
   struct sched_dl_entity  dl;
   struct sched_gang_entity  sge;
```

```
   const struct sched_class *sched_class;
```

linux/kernel/sched/core.c

The following changes are made in core.c file.

```
/* Added case for SCHED_GANG in priority max and min functions. */
    SYSCALL_DEFINE1(sched_get_priority_max, int, policy) // Also for sched_get_priority_min
{
   int ret = -EINVAL;

   switch (policy) {
   case SCHED_FIFO:
   case SCHED_RR:
      ret = MAX_RT_PRIO-1;
      break;
   case SCHED_DEADLINE:
   case SCHED_GANG:
   case SCHED_NORMAL:
   case SCHED_BATCH:
   case SCHED_IDLE:
      ret = 0;
      break;
   }
   return ret;
}

/* In void __init sched_init(void) */
   BUG_ON(&idle_sched_class != &fair_sched_class + 1 ||
          &fair_sched_class != &gang_sched_class + 1 ||
         &gang_sched_class != &rt_sched_class + 1 ||
          &rt_sched_class != &dl_sched_class + 1);
    // and later :
    init_cfs_rq(&rq->cfs);
    init_rt_rq(&rq->rt);
    init_dl_rq(&rq->dl);
    init_gang_rq(&rq->gang);
```

A very important change in the following function which is called by `__sched_setscheduler` which allows use to change scheduling class of the process.

```
    static void __setscheduler_prio(struct task_struct *p, int prio)
{
   if (dl_prio(prio))
      p->sched_class = &dl_sched_class;
   else if (rt_prio(prio))
      p->sched_class = &rt_sched_class;
   else if (p->policy == SCHED_GANG)
   {
      p->sched_class = &gang_sched_class; // Most important change in core.c
   }
   else
      p->sched_class = &fair_sched_class;

   p->prio = prio;
```

```
}
```

We can now discuss the meat of our gang scheduler implemented in :-

linux/kernel/sched/gang.h

```
/* after all the includes */
#define QUANTA_MSEC 10
struct gang_node
{
    int gangid;
    int num_th;
    u64 exec_time;
    int gang_gov_pid;
    bool
    bool gov_exists;
    cpumask_t gg_cpumask;
    cpumask_t cpumask;
    struct list_head sentinal_pid_node;
    struct list_head gang_node_list;
};

struct pid_node
{
    int pid;
    u64 exec_time;
    int prio;
    cpumask_t cpumask;
    struct list_head pid_node_list;
};


int __check_ncore(struct gang_node* gang_node,int online_cpus);
int __update_pid_list(struct gang_node* gang_node,int pid,u64 exec_time);
int __find_pid_node(struct gang_node** gang_node,struct pid_node** pid_node,pid_t pid);
int __exit_gang(pid_t pid,bool called_from_do_exit);
int __dereg_gang_node(struct gang_node* gang_node);
int __get_pid_ll(struct gang_node* gang_node,int* head);
```

linux/kernel/sched/gang.c

gang.c has more helper functions directly in the file itself as they are not expected to be used elsewhere in the kernel.

```
 #include "gang.h"

/* Global variables */
static LIST_HEAD(sentinal_gangs_head);
static unsigned long gll_size = 0;
static DEFINE_SPINLOCK(sentinal_lock);

/* Optional locking macros */
#ifdef DISABLE_LOCKS
#define spin_lock(lock)      do { } while(0)
```

```
#define spin_unlock(lock)     do { } while(0)
#define spin_lock_irqsave(sentinal_lock, flags) do { } while(0)
#define spin_unlock_irqrestore(sentinal_lock, flags) do { } while(0)
#endif

int __check_ncore(struct gang_node* gang_node, int online_cpus) { ... }

int __update_pid_list(struct gang_node* gang_node, int pid, u64 exec_time) { ... }

SYSCALL_DEFINE3(register_gang, pid_t, pid, int, gangid, int, exec_time) { ... }

int __find_pid_node(struct gang_node** gang_node, struct pid_node** pid_node, pid_t pid) { ... }
/*Following functions for custom_sched_setscheduler to change back to cfs task if lock is held*/
static inline int __rt_effective_prio(struct task_struct *pi_task, int prio) { ... }

static inline int __normal_prio(int policy, int rt_prio, int nice) { ... }

static inline int normal_prio(struct task_struct *p) { ... }

static inline int rt_effective_prio(struct task_struct *p, int prio) { ... }

static void __setscheduler_params(struct task_struct *p,
                                  const struct sched_attr *attr) { ... }

static void __setscheduler_prio(struct task_struct *p, int prio) { ... }

static inline void set_next_task_wrap(struct rq *rq, struct task_struct *next) { ... }

static inline void check_class_changed(struct rq *rq, struct task_struct *p,
                                       const struct sched_class *prev_class,
                                       int oldprio) { ... }

static int cust_sched_setscheduler(struct task_struct *p, int policy,
                                   const struct sched_param *param,
                                   const struct sched_attr *attr,
                                   bool user, bool pi, struct rq* rq) { ... }
/* actual exit gang logic */
int __exit_gang(pid_t pid, bool called_from_do_exit) { ... }

int __dereg_gang_node(struct gang_node* gang_node) { ... }

SYSCALL_DEFINE1(exit_gang, int, pid) { ... }

int __get_pid_ll(struct gang_node* gang_node, int* head) { ... }

SYSCALL_DEFINE2(list, int, gangid, int*, pids) { ... }

void init_gang_rq(struct gang_rq* gang_rq) { ... }

static void enqueue_task_gang(struct rq* rq, struct task_struct* p, int flags) { ... }

static void dequeue_task_gang(struct rq* rq, struct task_struct* p, int flags) { ... }

static inline bool _check_gov_alive(pid_t gang_gov_pid) { ... }

inline void __assign_new_gov(int gangid) { ... }
```

```
/* function for sending IPI's */
void __governer(struct rq* rq) { ... }

static struct task_struct * pick_next_task_gang(struct rq* rq) { ... }

static void _check_time_exceed(struct sched_gang_entity* sge, struct rq* rq) { ... }

static void __update_exec_time(struct rq* rq, struct task_struct* p) { ... }

static void put_prev_task_gang(struct rq* rq, struct task_struct* p) { ... }

static void task_tick_gang(struct rq* rq, struct task_struct* p, int queued) { ... }

// Rest of the functinos not mentioned of the sched class are implemented as Stubs.
DEFINE_SCHED_CLASS(gang) = {
...
};
```

Details about gang.c

- We maintain a global datastructe about the gangs and its pids in linked list of gang nodes which each point to a linked list of pid nodes both containing information like cpu masks, pid, exec_time etc.
- We rely on `__gang_rq` data structure which has a linked list of `sched_gang_entity` which are actually used on the runqueue to schedule.
- After we register we call the `sched_setscheduler_nocheck(p, SCHED_GANG,&param)` function to change our sched class from cfs to gang, leverging all our earlier changes in the kernel.
- When we exit from the gang we update our global gang_node list and call function to change the sched class to cfs.
- In `task_tick` we send the IPI of the corresponding gang using `__governer()` function.
- Below is our timing logic and how we change class to CFS if time has exceeded.

Timing logic

- We use `rq_clock_task` function to get time stamps about the runqueue and in each sched gang entity we have variables start_time, exec_time (time executing so far) and max_exec_time.
- When `pick_next_task` is called before return task_struct of the entity we set its start_time to now = `rq_clock_task(rq)`
- In `task_tick` and `put_prev_task` we do the following:-

```
static void _check_time_exceed(struct task_struct* p,struct rq* rq)
{
    struct sched_param param;
    int policy ;
    struct sched_attr attr = {
        .sched_policy  = SCHED_NORMAL,
        .sched_priority = 0,
      .sched_nice = PRIO_TO_NICE((p)->static_prio),
   };
    policy = SCHED_NORMAL;
```

```
    param.sched_priority = 0 ;
    if((p->sge).exec_time > (p->sge).max_exec_time)
    {
        /* Fixup the legacy SCHED_RESET_ON_FORK hack. */
        if ((policy != -1) && (policy & SCHED_RESET_ON_FORK)) {
            attr.sched_flags |= SCHED_FLAG_RESET_ON_FORK;
            policy &= ~SCHED_RESET_ON_FORK;
            attr.sched_policy = policy;
        }
        __exit_gang(p->pid,true);
        cust_sched_setscheduler(p,SCHED_NORMAL,&param,&attr,0,1,rq);
    }
}

static void __update_exec_time(struct rq* rq,struct task_struct* p)
{
    u64 delta;
    u64 now = rq_clock_task(rq);
    if((p->sge).start_time == 0)
    {
        (p->sge).start_time = now;
    }
    delta = now - (p->sge).start_time ;
    (p->sge).exec_time += delta;
    (p->sge).start_time = now;
}
static void
put_prev_task_gang(struct rq* rq, struct task_struct* p)
{
    if(!p || !pid_alive(p)) return ;
    if(task_is_running(p))
    __update_exec_time(rq,p) ;
}
static void
task_tick_gang(struct rq* rq,struct task_struct* p,int queued)
{
    static DEFINE_PER_CPU(unsigned long, gang_tick_counter);
    unsigned long *counter = this_cpu_ptr(&gang_tick_counter);
    unsigned long quanta_ticks = msecs_to_jiffies(QUANTA_MSEC);
    if(task_is_running(p))
    __update_exec_time(rq,p);
    _check_time_exceed(p,rq);
    (*counter)++;
    if (*counter >= quanta_ticks) {
        *counter = 0; // Reset the counter
        __governer(rq);
    }
}
```

- That is, our _update_exec function will update time slice, we need both task_tick and the put_prev function because due to premption we might miss task_tick but in that case put_prev function will get called.
- once timer exceeds our cust function imitates behaviour of the real sched setscheduler under assumption it has the locks, this was done by carefully understanding the real function.

Insights

- Learnt a lot about scheduling in linux kernel and various features and macros used, for example the use lds files.
- Learnt a lot about debugging such a sensitive part of the kernel and understanding how everything works from boot till the end of process from scheduling point of view.
- Appreciated reading so much kernel code from elixir bootlin to understand where I can insert my code and what various functions/macros lead to.

Novelties

- Actually implemented and tested the multiple gang handling logic, that is every quanta will alternate between gangs in round robin fashion if there is overlap between cores.
- Gang governer is decided automatically and even if deregisters first it automatically chooses a new governer and system remains stable.
- Implemented custom sched_setscheduler to imitate real one's behaviour as when called scheduling class function rq lock is held.

Challenges

- Biggest challenge was realizing the fact the scheduler is very senstive to changes, due to many reason like deadlock or null pointer deference, the vm kept crashing.
- Realized how sched_setscheduler works which in its params arguement wants rt_priotiy .

Further extendability

- In real HPC scenario because governer going to sleep can have deadlock like issues (due to lock being held by some other thread or condition variables). Ideally we might want dynamic governer.
- We wouldnt want to waste time changing the governer too much, but maybe what can be done is some heurestic managing both dynamic governer and if multiple gangs exists.

Testing interesting cases

- Tested that if one gang is sleep the other gang starts running.
- Tested if governer sleeps then no other thread should also be working.
- Tested if one gang exec time exceeds it changes to CFS and therefore other gang is able to completely monopolize the cores from that point and once the latter gang finished the gang which is now in CFS actually does finish.

References

References

[1] 0xax, *Linux CPU Architecture (Part 2)*. Retrieved from `https://0xax.gitbooks.io/linux-insides/content/Concepts/linux-cpu-2.html`

[2] Bootlin, *Linux Kernel Source – sched/core.c*. Retrieved from `https://elixir.bootlin.com/linux/v6.1.130/source/kernel/sched/core.c#L2064`

[3] The Linux Kernel Documentation, *Kernel API*. Retrieved from `https://www.kernel.org/doc/html/v4.14/core-api/kernel-api.html`

[4] O'Reilly, *Understanding the Linux Kernel*. Retrieved from `https://www.oreilly.com/library/view/understanding-the-linux/0596005652/ch04s06.html#understandlk-CHP-4-TABLE-2`

[5] Sarangi, S. M., *Operating Systems Book*. Retrieved from `https://www.cse.iitd.ac.in/~srsarangi/osbook/osbook-v0.82.pdf`

[6] Assignment 1 Materials, *Reference Materials for Assignment 1*.

[7]
- Stack Overflow, *Linux Scheduler Modification*. Retrieved from `https://stackoverflow.com/questions/22045393/linux-scheduler-modification`
- M. Gouzenko, *Understanding sched_class in Linux*. Retrieved from `https://mgouzenko.github.io/jekyll/update/2016/11/24/understanding-sched-class.html`
- SCSLab Interns, *Linux Kernel Hacking Guide - Chapter 4*. Retrieved from `https://scslab-intern.gitbooks.io/linux-kernel-hacking/content/chapter04.html`
- T. Mäki, *Modifying the Linux Process Scheduler*. Retrieved from `https://trepo.tuni.fi/bitstream/handle/10024/96864/GRADU-1428493916.pdf`