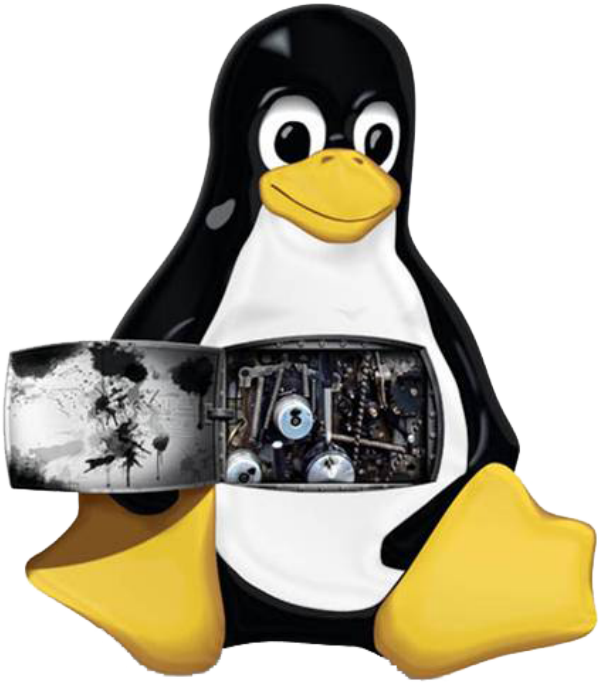
Linux Kernel Development (LKD)

Session 3



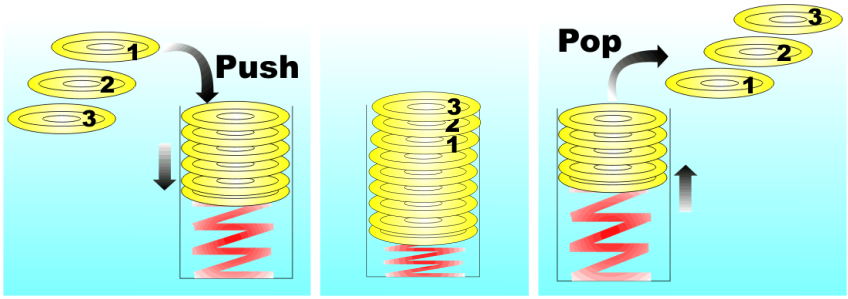
CISTER Framework: Laboratory

Paulo Baltarejo Sousa
pbs@isep.ipp.pt
2017

CISTER - Research Centre in Real-Time & Embedded Computing Systems
isep Instituto Superior de Engenharia do Porto
INESCTEC

1 LIFO Scheduling Algorithm

It will be added a new scheduling policy to the CISTER framework. The scheduling algorithm is the well known Last-In First-Out (LIFO). According to LIFO scheduling algorithm, the last task ready for execution execution will be selected to be executed. That is, whenever a task arrives (become ready for execution) the currently executing is preempted and the recently arrived task is selected for execution.



A set of steps is required to add a new scheduling policy.

1. Add a new configuration entry to the `linux-4.12.4-cister/kernel/cister/Kconfig` file, called `CISTER_SCHED_LIFO_POLICY`.

```
menu "CISTER framework"
config CISTER_FRAMEWORK
  bool "CISTER Framework"
  default y

config CISTER_TRACING
  bool "CISTER tracing"
  default y
  depends on CISTER_FRAMEWORK

config CISTER_SCHED_LIFO_POLICY
  bool "CISTER scheduling policy:LIFO"
  default y
  depends on CISTER_FRAMEWORK

endmenu
```

2. The second step is to define a macro to identify the scheduling policy. For that purpose, it must be checked the already defined scheduling policy identifiers. They are defined in the `linux-4.12.4-cister/include/uapi/linux/sched.h` file.

```
/*
 * 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
```

Create a new macro called `SCHED_LIFO` with value 7 and add it to `linux-4.12.4-cister/include/uapi/linux/sched.h` file.

```
#ifndef _UAPI_LINUX_SCHED_H
#define _UAPI_LINUX_SCHED_H
...
/*
 * 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

#ifdef CONFIG_CISTER_SCHED_LIFO_POLICY
#define SCHED_LIFO  7
#endif

...
#endif /* _UAPI_LINUX_SCHED_H */
```

3. Each processor holds a run-queue of all runnable processes assigned to it. The scheduling policy uses this run-queue to select the "best" process to be executed. Therefore, the run-queue is the starting point for many scheduling actions. Note, however, that processes are not directly managed by the general elements of the run-queue. This is the responsibility of the individual scheduling classes, and a class-specific sub run-queue is therefore embedded in each processor run-queue.

   The information for these processes is stored in a per-processor data structure called `struct rq`, which is defined in the `linux-4.12.4-cister/kernel/sched/sched.h`.

```
/*
 * This is the main, per-CPU runqueue data structure.
 *
 * Locking rule: those places that want to lock multiple runqueues
 * (such as the load balancing or the thread migration code), lock
 * acquire operations must be ordered by ascending &runqueue.
 */
struct rq {
  /* runqueue lock: */
  raw_spinlock_t lock;

  /*
   * nr_running and cpu_load should be in the same cacheline because
   * remote CPUs use both these fields when doing load calculation.
   */
  unsigned int nr_running;
...
  #define CPU_LOAD_IDX_MAX 5
  unsigned long cpu_load[CPU_LOAD_IDX_MAX];
..
  u64 nr_switches;

  struct cfs_rq cfs;
  struct rt_rq rt;
  struct dl_rq dl;
...
};
```

As it can be seen from the snippet code there is on sub-run-queue for CFS, RT and DL `struct cfs_rq cfs`, `struct rt_rq rt` and `struct dl_rq rt` respectively. Using the same approach, it is defined a new data structure, called `struct lf_rq` for the new scheduling policy. For that purpose, create a new file called `lf_rq.h` in the `linux-4.12.4-cister/kernel/cister` directory.

```
#ifndef __LF_RQ_H_
#define __LF_RQ_H_

#include <linux/sched.h>
#include <linux/list.h>
#include <linux/spinlock.h>

struct lf_rq{
  struct list_head tasks;
  struct task_struct *task;
  spinlock_t lock;
};
void init_lf_rq(struct lf_rq *rq);
#endif
```

A stack is the appropriated container for implementing the LIFO policy. In Linux it can be easily implemented using the List API (defined in `linux-4.12.4-cister/include/linux/list.h`). The head of the stack and also the nodes have to be of `struct list_head tasks` data structure type. Being the entry point, in this `struct lf_rq` data structure it is defined the head of the list. In order to to protect list operations, it is also defined a `spinlock_t` variable. For optimization purpose, it is also defined a `struct task_struct` pointer that will be

used to, at any instant, point to the highest priority task (that is, the most arriving recent task). It is also defined the `init_lf_rq` prototype function, which purpose is to initialize the `struct lf_rq` data structure fields.

4. Put this data structure visible to all kernel, by updating the `linux-4.12.4-cister/kernel/sched/sched.h`

```
...
#ifdef CONFIG_SCHED_DEBUG
#define SCHED_WARN_ON(x) WARN_ONCE(x, #x)
#else
#define SCHED_WARN_ON(x) ((void)(x))
#endif

#ifdef CONFIG_CISTER_SCHED_LIFO_POLICY
  #include "../cister/lf_rq.h"
#endif

#ifdef CONFIG_CISTER_TRACING
  #include "../cister/trace.h"
#endif
...
```

5. So, the next step is to add `struct lf_rq` field to the `struct rq` data structure (defined in `linux-4.12.4-cister/kernel/sched/sched.h`).

```
struct rq {
  ...

  struct cfs_rq cfs;
  struct rt_rq rt;
  struct dl_rq dl;

#ifdef CONFIG_CISTER_SCHED_LIFO_POLICY
  struct lf_rq lf;
#endif

  ...
};
```

6. The `spinlock_t lock` as well as `struct list_head tasks` variables need to be initialized. This must be done before the Linux kernel scheduler starts operating. For that purpose, create `linux-4.12.4-cister/kernel/cister/lf_rq.c` file and implement `init_lf_rq` function. As it can be seen from the snippet code, all `struct lf_rq` data structure fields are initialized.

```
#include "lf_rq.h"

void init_lf_rq(struct lf_rq *rq)
{
  INIT_LIST_HEAD(&rq->tasks);
  spin_lock_init(&rq->lock);
  rq->task = NULL;
}
```

The best place for invoking `init_lf_rq` function, is in the `sched_init` function implemented in the `linux-4.12.14-cister/kernel/sched/core.c` file.

```
void __init sched_init(void)
{
  int i, j;
  unsigned long alloc_size = 0, ptr;

  sched_clock_init();

  ...

  for_each_possible_cpu(i) {
    struct rq *rq;

    rq = cpu_rq(i);
    raw_spin_lock_init(&rq->lock);
    rq->nr_running = 0;
    rq->calc_load_active = 0;
    rq->calc_load_update = jiffies + LOAD_FREQ;
    init_cfs_rq(&rq->cfs);
    init_rt_rq(&rq->rt);
    init_dl_rq(&rq->dl);

#ifdef CONFIG_CISTER_SCHED_LIFO_POLICY
    init_lf_rq(&rq->lf);
#endif

    ...
  }
  ...
}
```

`sched_init` function creates the CPU run queues, scheduler related data structures and initializes them.

7. A Linux process is an instance of a program in execution. To manage processes, the kernel maintains information about each process in a process descriptor. The information stored in each process descriptor (`struct task_struct`, defined in `linux-4.12.4-cister/include/linux/sched.h`) concerns with the run-state of a process, its address space, the list of open files, the process priority and its scheduling class, just to mention some.

   Since, it was created a new list in the run-queue for storing ready `LIFO` tasks (`LIFO` tasks are those scheduled according to new scheduling

policy, LIFO). So, it is required to add a new `struct list_head` field to the `struct task_struct`.

For that purpose, create a new data structure called `struct lf_task` in the `linux-4.12.4-cister/kernel/cister/lf_task.h` file. In this case, the `struct lf_task` has only one field that is required to be used in the list, which head, called `tasks`, it is defined in the `linux-4.12.4-cister/kernel/cister/lf_rq.h` file.

```
#ifndef __LF_TASK_H_
#define __LF_TASK_H_

struct lf_task{
  struct list_head node;
};
#endif
```

8. Put this data structure visible to all kernel, by updating the `linux-4.12.4-cister/include/linux/sched.h`

```
...
#include <linux/seccomp.h>
#include <linux/nodemask.h>
#include <linux/rcupdate.h>
#include <linux/resource.h>
#include <linux/latencytop.h>
#include <linux/sched/prio.h>
#include <linux/signal_types.h>
#include <linux/mm_types_task.h>
#include <linux/task_io_accounting.h>

#ifdef CONFIG_CISTER_SCHED_LIFO_POLICY
#include "../../kernel/cister/lf_task.h"
#endif

...
```

9. Next step is to add a new filed, called `lf_task`, of the `struct lf_task` data structure, to the `struct task_struct` data structure (defined in `linux-4.12.4-cister/include/linux/sched.h`.

```
struct task_struct {
  ...
  int    on_rq;

  int    prio;
  int    static_prio;
  int    normal_prio;
  unsigned int   rt_priority;

  const struct sched_class *sched_class;
  struct sched_entity  se;
  struct sched_rt_entity  rt;
#ifdef CONFIG_CGROUP_SCHED
  struct task_group  *sched_task_group;
#endif
  struct sched_dl_entity  dl;

#ifdef CONFIG_CISTER_SCHED_LIFO_POLICY
  struct lf_task    lf_task;
#endif

#ifdef CONFIG_PREEMPT_NOTIFIERS
  /* List of struct preempt_notifier: */
  struct hlist_head  preempt_notifiers;
#endif

#ifdef CONFIG_BLK_DEV_IO_TRACE
  unsigned int   btrace_seq;
#endif
  ...
};
```

10. Scheduling classes are implemented using the `sched_class` data structure, which contains hooks to functions that must be called whenever an scheduling event occurs and it is defined in `linux-4.12.4-cister/kernel/sched/sched.h` file.

```
struct sched_class {
  const struct sched_class *next;

  void (*enqueue_task) (struct rq *rq, struct task_struct *p, int flags);
  void (*dequeue_task) (struct rq *rq, struct task_struct *p, int flags);
  ...
  void (*check_preempt_curr) (struct rq *rq, struct task_struct *p, int flags);
...
  struct task_struct * (*pick_next_task) (struct rq *rq,
           struct task_struct *prev,
           struct rq_flags *rf);
  void (*put_prev_task) (struct rq *rq, struct task_struct *p);

...
};
```

Any scheduling class must be implemented as an instance of `sched_class` data structure.

For that purpose, create a new file called `sched.c` in the `linux-4.12.4-cister/kernel/cister` directory. In that file, define the `lf_sched_class` variable.

```
const struct sched_class lf_sched_class = {

  .next    = &fair_sched_class,
  .enqueue_task  = enqueue_task_lf,
  .dequeue_task  = dequeue_task_lf,
  .yield_task  = yield_task_lf,

  .check_preempt_curr = check_preempt_curr_lf,

  .pick_next_task  = pick_next_task_lf,
  .put_prev_task  = put_prev_task_lf,

#ifdef CONFIG_SMP
  .select_task_rq  = select_task_rq_lf,
#endif

  .set_curr_task = set_curr_task_lf,
  .task_tick  = task_tick_lf,
  .get_rr_interval = get_rr_interval_lf,

  .prio_changed  = prio_changed_lf,
  .switched_to  = switched_to_lf,
  .update_curr  = update_curr_lf,

};
```

Note that, almost all functions have as arguments pointers of `struct task_struct` and `struct rq`. The `lf_sched_class` scheduling policy it is positioned between the `RT` and `CFS` scheduling classes.

Then, using the `next` field, which is a `sched_class` pointer, it points to `fair_sched_class`, which implements the `CFS`. So, in order to positioned `lf_sched_class` between them, it is required to change the `rt_sched_class`, which implements the `RT` scheduling class, that is implemented in the `linux-4.12.4-cister/kernel/sched/rt.c` file.

```
...
const struct sched_class rt_sched_class = {

#ifdef CONFIG_CISTER_SCHED_LIFO_POLICY
  .next   = &lf_sched_class,
#else
  .next   = &fair_sched_class,
#endif

  .enqueue_task  = enqueue_task_rt,
  .dequeue_task  = dequeue_task_rt,
  .yield_task  = yield_task_rt,

  .check_preempt_curr = check_preempt_curr_rt,

  .pick_next_task  = pick_next_task_rt,
  .put_prev_task  = put_prev_task_rt,

#ifdef CONFIG_SMP
  .select_task_rq  = select_task_rq_rt,

  .set_cpus_allowed = set_cpus_allowed_common,
  .rq_online = rq_online_rt,
  .rq_offline = rq_offline_rt,
  .task_woken  = task_woken_rt,
  .switched_from  = switched_from_rt,
#endif

  .set_curr_task = set_curr_task_rt,
  .task_tick  = task_tick_rt,

  .get_rr_interval = get_rr_interval_rt,

  .prio_changed  = prio_changed_rt,
  .switched_to  = switched_to_rt,

  .update_curr  = update_curr_rt,
};
...
```

11. Implement the scheduling algorithm. Basically, it is required to code 4 `lf_sched_class` functions: `enqueue_task_lf`, codedequeue_task_lf, `check_preempt_curr_lf` and `pick_next_task_lf`. Update `linux-4.12.4-cister/kernel/cister/sched.c` file with:

```
#include "../sched/sched.h"

/*
 * LIFO scheduling class.
 * Implements SCHED_LIFO
 */

static void enqueue_task_lf(struct rq *rq, struct task_struct *p, int flags)
{

  spin_lock(&rq->lf_rq.lock);
  list_add(&p->lf_task.node,&rq->lf_rq.tasks);
  rq->lf_rq.task = p;
  spin_unlock(&rq->lf_rq.lock);

}
```

```
static void dequeue_task_lf(struct rq *rq, struct task_struct *p, int flags)
{

  struct lf_task *t = NULL;
  spin_lock(&rq->lf.lock);
  list_del(&p->lf_task.node);
  if(list_empty(&rq->lf_rq.tasks)){
    rq->lf_rq.task = NULL;
  }else{
    t = list_first_entry(&rq->lf_rq.tasks,struct lf_task, node);
    rq->lf.task = container_of(t,struct task_struct, lf_task);
  }
  spin_unlock(&rq->lf.lock);

}


static void yield_task_lf(struct rq *rq)
{

}
static void check_preempt_curr_lf(struct rq *rq, struct task_struct *p, int flags)
{
  switch(rq->curr->policy){
    case SCHED_DEADLINE:
    case SCHED_FIFO:
    case SCHED_RR:
    break;
    case SCHED_NORMAL:
    case SCHED_BATCH:
    case SCHED_IDLE:
    //case SCHED_RESET_ON_FORK:
    case SCHED_LIFO:
      resched_task(rq->curr);
    break;
  }
}

static struct task_struct *pick_next_task_lf(struct rq *rq)
{

  struct task_struct * p = NULL;
  spin_lock(&rq->lf.lock);
  p = rq->lf.task;
  spin_unlock(&rq->lf.lock);
  return p;
}
static void put_prev_task_lf(struct rq *rq, struct task_struct *prev)
{
}
#ifdef CONFIG_SMP
static int select_task_rq_lf(struct task_struct *p, int sd_flag, int flags)
{
  return task_cpu(p);
}
#endif /* CONFIG_SMP */
static void set_curr_task_lf(struct rq *rq)
{
}
static void task_tick_lf(struct rq *rq, struct task_struct *curr, int queued)
{
}
static unsigned int get_rr_interval_lf(struct rq *rq, struct task_struct *task)
{
  return 0;
}
static void switched_to_lf(struct rq *rq, struct task_struct *p)
{
}
static void prio_changed_lf(struct rq *rq, struct task_struct *p, int oldprio)
{
}
static void update_curr_lf(struct rq *rq)
```

```
{
}

const struct sched_class lf_sched_class = {

  .next    = &fair_sched_class,
  .enqueue_task  = enqueue_task_lf,
  .dequeue_task  = dequeue_task_lf,
  .yield_task  = yield_task_lf,

  .check_preempt_curr = check_preempt_curr_lf,

  .pick_next_task  = pick_next_task_lf,
  .put_prev_task  = put_prev_task_lf,

#ifdef CONFIG_SMP
  .select_task_rq  = select_task_rq_lf,
#endif

  .set_curr_task = set_curr_task_lf,
  .task_tick  = task_tick_lf,
  .get_rr_interval = get_rr_interval_lf,

  .prio_changed  = prio_changed_lf,
  .switched_to  = switched_to_lf,
  .update_curr  = update_curr_lf,

};
```

Update the `linux-4.12.4-cister/kernel/sched/sched.h` file to put the `lf_sched_class` in the hierarchy order.

```
#define sched_class_highest (&stop_sched_class)
#define for_each_class(class) \
   for (class = sched_class_highest; class; class = class->next)

extern const struct sched_class stop_sched_class;
extern const struct sched_class dl_sched_class;
extern const struct sched_class rt_sched_class;

#ifdef CONFIG_CISTER_SCHED_LIFO_POLICY
extern const struct sched_class lf_sched_class;
#endif

extern const struct sched_class fair_sched_class;
extern const struct sched_class idle_sched_class;
```

12. Right now, it was defined a `LIFO` task and a scheduling class to schedule `LIFO` tasks. Next step, it is associate `LIFO` tasks to the `lf_sched_class` scheduling class. This is done by invoking the `sched_setscheduler` system call, which is coded in the `linux-4.12.4-cister/kernel/sched/core.c`. From the `sched_setscheduler` function it is invoked the `_sched_setscheduler` function, which in turn calls `__sched_setscheduler` function, a long function. In this function are invoked two functions that needs to be updated to support LIFO tasks. The functions are: `valid_policy` and `__setscheduler`.

```
static int __sched_setscheduler(struct task_struct *p,
       const struct sched_attr *attr,
       bool user, bool pi)
{
  int newprio = dl_policy(attr->sched_policy) ? MAX_DL_PRIO - 1 :
        MAX_RT_PRIO - 1 - attr->sched_priority;
  int retval, oldprio, oldpolicy = -1, queued, running;
  int new_effective_prio, policy = attr->sched_policy;
  const struct sched_class *prev_class;
  struct rq_flags rf;
  int reset_on_fork;
  int queue_flags = DEQUEUE_SAVE | DEQUEUE_MOVE | DEQUEUE_NOCLOCK;
  struct rq *rq;

  /* May grab non-irq protected spin_locks: */
  BUG_ON(in_interrupt());
recheck:
  /* Double check policy once rq lock held: */
  if (policy < 0) {
    reset_on_fork = p->sched_reset_on_fork;
    policy = oldpolicy = p->policy;
  } else {
    reset_on_fork = !!(attr->sched_flags & SCHED_FLAG_RESET_ON_FORK);

    if (!valid_policy(policy))
      return -EINVAL;
  }
  ...
  __setscheduler(rq, p, attr, pi);
  ...
  return 0;
}
```

The `valid_policy` function it is implemented in `linux-4.12.4-cister/kernel/sched/sched.h` file. Update this file in order to accept `SCHED_LIFO` policy.

```
static inline int idle_policy(int policy)
{
  return policy == SCHED_IDLE;
}
static inline int fair_policy(int policy)
{
  return policy == SCHED_NORMAL || policy == SCHED_BATCH;
}

static inline int rt_policy(int policy)
{
  return policy == SCHED_FIFO || policy == SCHED_RR;
}

static inline int dl_policy(int policy)
{
  return policy == SCHED_DEADLINE;
}

#ifdef CONFIG_CISTER_SCHED_LIFO_POLICY
static inline int lf_policy(int policy)
{
  return policy == SCHED_LIFO;
}
#endif

static inline bool valid_policy(int policy)
{
  return idle_policy(policy) || fair_policy(policy) ||
    rt_policy(policy) || dl_policy(policy)
#ifdef CONFIG_CISTER_SCHED_LIFO_POLICY
    || lf_policy(policy)
#endif
    ;
}
```

The `__setscheduler` function it is implemented `linux-4.12.4-cister/kernel/sched/core.c` file. Update this function in order to assign LIFO tasks to the `lf_sched_class` scheduling class.

```
static void __setscheduler(struct rq *rq, struct task_struct *p,
        const struct sched_attr *attr, bool keep_boost)
{
  ...

  if (dl_prio(p->prio))
    p->sched_class = &dl_sched_class;
  else if (rt_prio(p->prio))
    p->sched_class = &rt_sched_class;
  else
    p->sched_class = &fair_sched_class;

#ifdef CONFIG_CISTER_SCHED_LIFO_POLICY
  if(p->policy == SCHED_LIFO)
    p->sched_class = &lf_sched_class;
#endif

}
```

13. Since the LIFO scheduling algoritm depends on the enqueue order, so it is relevant that such information being collected. For that purpose update `linux-4.12.4-cister/kernel/cister/trace.h` and `linux-`

4.12.4-cister/kernel/cister/trace.c file to support such events.

```
#ifndef __TRACE_H_
#define __TRACE_H_
...

enum evt{
  SCHED_TICK = 0,
  SWITCH_AWAY,
  SWITCH_TO,
  ENQUEUE_RQ,
  DEQUEUE_RQ

};

...

#endif
```

```
...
static int dequeue (char *buffer)
{
  int ret = 0, len;
  char evt[20];
  spin_lock(&trace.lock);
  if(!is_empty(trace.read_item,trace.write_item)){ //if it is not empty

    switch((int)trace.events[trace.read_item].event){
      case SCHED_TICK:
        strcpy(evt,"SCH_TK");
      break;
      case SWITCH_AWAY:
        strcpy(evt,"SWT_AY");
      break;
      case SWITCH_TO:
        strcpy(evt,"SWT_TO");
      break;

      case ENQUEUE_RQ:
        strcpy(evt,"ENQ_RQ");
      break;
      case DEQUEUE_RQ:
        strcpy(evt,"DEQ_RQ");
      break;

    }
    len = sprintf(buffer,"%llu,",trace.events[trace.read_item].time);
    len += sprintf(buffer+len,"%s,",evt);
    len += sprintf(buffer+len,"pid,%d,",(int)trace.events[trace.read_item].pid);
    len += sprintf(buffer+len,"prio,%d,",(int)trace.events[trace.read_item].prio);
    len += sprintf(buffer+len,"policy,%d,",(int)trace.events[trace.read_item].policy);
    len += sprintf(buffer+len,"state,%d,",(int)trace.events[trace.read_item].state);
    len += sprintf(buffer+len,"%s\n",trace.events[trace.read_item].comm);


    increment(&trace.read_item);
    ret = 1;
  }
  spin_unlock(&trace.lock);
  return ret;
}
```

14. Update linux-4.12.4-cister/kernel/cister/sched.c file to support collect data.

```
#include "../sched/sched.h"

/*
 * LIFO scheduling class.
 * Implements SCHED_LIFO
 */

static void enqueue_task_lf(struct rq *rq, struct task_struct *p, int flags)
{

  spin_lock(&rq->lf.lock);
  list_add(&p->lf_task.node,&rq->lf.tasks);
  rq->lf.task = p;
  spin_unlock(&rq->lf.lock);

#ifdef CONFIG_CISTER_TRACING
  cister_trace(ENQUEUE_RQ,p);
#endif

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

  struct lf_task *t = NULL;
  spin_lock(&rq->lf.lock);
  list_del(&p->lf_task.node);
  if(list_empty(&rq->lf.tasks)){
    rq->lf.task = NULL;
  }else{
    t = list_first_entry(&rq->lf.tasks,struct lf_task, node);
    rq->lf.task = container_of(t,struct task_struct, lf_task);
  }
  spin_unlock(&rq->lf.lock);

#ifdef CONFIG_CISTER_TRACING
  cister_trace(DEQUEUE_RQ,p);
#endif

}
...

void cister_trace(enum evt event, struct task_struct *p)
{
//this is a filter for collecting LIFO tasks data
  if(p->policy != SCHED_LIFO)
    return;

  if(enabled){
    unsigned long long time = ktime_to_ns(ktime_get());
    enqueue(event, time, p);
  }
}
```

15. Since, the scheduler tick event is not important for LIFO scheduling algorithm let us ignore it by updating the `scheduler_tick function` (`linux-4.12.4-cister/kernel/sched/core.c`)

```
/*
 * This function gets called by the timer code, with HZ frequency.
 * We call it with interrupts disabled.
 */
void scheduler_tick(void)
{
  int cpu = smp_processor_id();
  struct rq *rq = cpu_rq(cpu);
  struct task_struct *curr = rq->curr;
  struct rq_flags rf;

  sched_clock_tick();

  rq_lock(rq, &rf);

  update_rq_clock(rq);
  curr->sched_class->task_tick(rq, curr, 0);

/*
#ifdef CONFIG_CISTER_TRACING
  cister_trace(SCHED_TICK, curr);
#endif
*/

  cpu_load_update_active(rq);
  calc_global_load_tick(rq);

  rq_unlock(rq, &rf);

  perf_event_task_tick();

#ifdef CONFIG_SMP
  rq->idle_balance = idle_cpu(cpu);
  trigger_load_balance(rq);
#endif
  rq_last_tick_reset(rq);
}
```

16. For compiling all this code update the `linux-4.12.4-cister/kernel/cister/Makefile` file

```
# CISTER Framework makefile

obj-$(CONFIG_CISTER_TRACING) += trace.o
obj-$(CONFIG_CISTER_SCHED_LIFO_POLICY) += lf_rq.o sched.o
```

17. Compile the Linux kernel:
    `> sudo ./kcompile.sh`
    `> sudo reboot`

18. Download `tasks.tar.gz` file from `http://www.cister.isep.ipp.pt/summer2017/w1/tasks.tar.gz` by typing:
    `> wget http://www.cister.isep.ipp.pt/summer2017/w1/tasks.tar`

`.gz`

Extract it by typing: `> > tar -xf tasks.tar.gz`

Compile
`> make`
Put the set of tasks running:
`> sudo ./launcher taskset.txt`
Wait and get the scheduling trace:
`> cat /proc/cister_trace > trace.txt`
`> gedit trace.txt`

```
45981436360,SWT_AY,pid,1642,prio,120,policy,7,state,64,task
45981436442,SWT_TO,pid,1640,prio,120,policy,7,state,0,task
46104206034,DEQ_RQ,pid,1640,prio,120,policy,7,state,1,task
46104209448,SWT_AY,pid,1640,prio,120,policy,7,state,1,task
46104488879,ENQ_RQ,pid,1640,prio,120,policy,7,state,256,task
46104803769,SWT_TO,pid,1640,prio,120,policy,7,state,0,task
48052716066,DEQ_RQ,pid,1640,prio,120,policy,7,state,1,task
48052719767,SWT_AY,pid,1640,prio,120,policy,7,state,1,task
48054663247,ENQ_RQ,pid,1640,prio,120,policy,7,state,256,task
48058464320,ENQ_RQ,pid,1639,prio,120,policy,7,state,0,task
48058477341,DEQ_RQ,pid,1639,prio,120,policy,7,state,1,task
48058477857,SWT_AY,pid,1639,prio,120,policy,7,state,1,task
48058477918,SWT_TO,pid,1640,prio,120,policy,7,state,0,task
48058915869,ENQ_RQ,pid,1639,prio,120,policy,7,state,256,task
48058915920,SWT_AY,pid,1640,prio,120,policy,7,state,0,task
48058947143,SWT_TO,pid,1639,prio,120,policy,7,state,0,task
50004466843,DEQ_RQ,pid,1639,prio,120,policy,7,state,1,task
```

Check if the scheduling output is according to the LIFO algorithm.

2 Enabling and disabling tracing mechanism

In order to implement a new system call to enable or disable the TRACING mechanism it is required a set of steps:

1. Add a new entry to the system call table. This is located at `linux-4.12.4-cister/arch/x86/entry/syscalls/syscall_64.tbl`.

   Add a new entry to the `linux-4.12.4-cister/arch/x86/entry/syscalls/syscall_64.tbl` file. The number of the new system call is 333. This system call is used for x86 and x86_64 systems, so the ABI is `common`. The system call name is `cister_tracing` and the entry point is `sys_cister_tracing`.

```
#
# 64-bit system call numbers and entry vectors
#
# The format is:
# <number> <abi> <name> <entry point>
#
# The abi is "common", "64" or "x32" for this file.
#
0 common read   sys_read
1 common write   sys_write
2 common open   sys_open
3 common close   sys_close
...
330 common pkey_alloc  sys_pkey_alloc
331 common pkey_free  sys_pkey_free
332 common statx   sys_statx

#ifdef CONFIG_CISTER_FRAMEWORK
333 common cister_tracing  sys_cister_tracing
#endif

#
# x32-specific system call numbers start at 512 to avoid cache impact
# for native 64-bit operation.
#
512 x32 rt_sigaction  compat_sys_rt_sigaction
...
```

2. Provide a function prototype in the `linux-4.12.4-cister/include/linux/syscalls.h` file.

   Create a new file called `syscalls.h` file in the `linux-4.12.4-cister/kernel/cister` directory. This file will host all CISTER-related system call prototypes. Write into it, the prototype of the new system call called `sys_cister_tracing` that receive a parameter to enable (`enable` equal to 1 or 0, for enabling or disabling the TRACING mechanism, respectively).

   ```
   #ifndef __SYSCALLS_H
   #define __SYSCALLS_H

   asmlinkage long sys_cister_tracing(int enable);

   #endif
   ```

   Include the `linux-4.12.4-cister/kernel/cister/syscalls.h` file in the `linux-4.12.4-cister/include/linux/syscalls.h` file.

```
/*
 * syscalls.h - Linux syscall interfaces (non-arch-specific)
 *
 * Copyright (c) 2004 Randy Dunlap
 * Copyright (c) 2004 Open Source Development Labs
 *
 * This file is released under the GPLv2.
 * See the file COPYING for more details.
 */

#ifndef _LINUX_SYSCALLS_H
#define _LINUX_SYSCALLS_H

...

asmlinkage long sys_pkey_alloc(unsigned long flags, unsigned long init_val);
asmlinkage long sys_pkey_free(int pkey);
asmlinkage long sys_statx(int dfd, const char __user *path, unsigned flags,
       unsigned mask, struct statx __user *buffer);


#ifdef CONFIG_CISTER_FRAMEWORK
#include "../../kernel/cister/syscalls.h"
#endif

#endif
```

3. Implementation of the system call function. Create a new file called `syscalls.c` file in the `linux-4.12.4-cister/kernel/cister` directory. This file will host all CASIO-related system call implementations. Write into it, the implementation of the new system call called `sys_cister_tracing`.

```
#include <linux/syscalls.h>
#include "trace.h"

asmlinkage long sys_cister_tracing(int enable)
{
#ifdef CONFIG_CISTER_TRACING
  enable_tracing(enable);
#endif
  return 0;
}
```

This system call calls a function called `enable_tracing` that must be implemented in the `linux-4.12.4-cister/kernel/cister/trace.h` and `linux-4.12.4-cister/kernel/cister/trace.c` files. Declare the function prototype in `linux-4.12.4-cister/kernel/cister/trace.h`.

```
#ifndef __TRACE_H_
#define __TRACE_H_

#include <linux/sched.h>

...
void cister_trace(enum evt event, struct task_struct *p);

void enable_tracing(int enable);

#endif
```

Implement such function in `linux-4.12.4-cister/kernel/cister/trace.c`.

```
#include <linux/module.h>
#include <linux/kernel.h>
#include <linux/proc_fs.h>
#include <linux/uaccess.h>

#include "trace.h"

//////////////////////////////////////////////////////////////////////
// Ring Buffer implementation
struct trace_evt_buffer trace;

unsigned char enabled = 0;

...

//This function will be used to get the event.

void cister_trace(enum evt event, struct task_struct *p)
{
  if(p->policy != SCHED_LIFO)
    return;
  if(enabled){

    unsigned long long time = ktime_to_ns(ktime_get());
    enqueue(event, time, p);
  }
}



void enable_tracing(int enable)
{
  enabled = (unsigned char) enable;
}
```

4. Include the system call function in the Linux kernel compilation process. For that purpose, add `syscalls.o` to the `linux-4.12.4-cister/kernel/cister/Makefile` file.

```
# CISTER Framework makefile

obj-$(CONFIG_CISTER_TRACING) += trace.o
obj-$(CONFIG_CISTER_SCHED_LIFO_POLICY) += lf_rq.o sched.o

obj-y += syscalls.o
```

The next step is to invoke `kcompile.sh` script file to compile the Linux kernel by typing:

```
> sudo ./kcompile.sh
```

Then, reboot the system.

```
> sudo reboot
```

5. Update the tasks source code: `defs.h` and `launch.c`

```
#ifndef DEFS_H_
#define DEFS_H_

...
#define SCHED_LIFO  7

#define __NR_cister_tracing  333

struct task {
  int id;
  unsigned long long C; //exec
  unsigned long long T; //period
  unsigned long long D; //deadline
  unsigned long long O; //first job offset
};

#endif
```

```
...
int main(int argc, char *argv[])
{
  ...

  time0 += (unsigned long long)(5 * OFFSET); //for safety purposes

  printf("LAUNCH: Trace enabled\n");
  if(syscall(__NR_cister_tracing, 1)){
    printf("Error: __NR_cister_tracing\n");
    exit(0);
  }

  printf("LAUNCH: Forking tasks\n");
  for(i=0;i<ntasks;i++){
    sprintf(arg[0],"%d",tasks[i].id);
    sprintf(arg[1],"%llu",tasks[i].C);
    sprintf(arg[2],"%llu",tasks[i].T);
    sprintf(arg[3],"%llu",tasks[i].D);
    sprintf(arg[4],"%llu",tasks[i].O);
    sprintf(arg[5],"%d",njobs);
    sprintf(arg[6],"%llu",time0);
    pid_tasks[i]=fork();
    if(pid_tasks[i]==0){
      execl("./task","task",arg[0],arg[1],arg[2],arg[3],arg[4],arg[5],argv[6],NULL);
      printf("Error: execv: task\n");
      exit(0);
    }

  }
  printf("LAUNCH: Waiting ...\n");
  for(i=0;i<ntasks;i++){
    waitpid(0,&status,0);
    if(WIFEXITED(status)){
      printf("LAUNCH:task:%d: has finished\n",WEXITSTATUS(status));
    }
  }

  printf("LAUNCH: Trace disabled\n");
  if(syscall(__NR_cister_tracing, 0)){
    printf("Error: __NR_cister_tracing\n");
    exit(0);
  }

  printf("LAUNCH: Finishing ...\n");
  return 0;
}
```

Compile it:
`> make`

6. Put the set of tasks running:
   `> sudo ./launcher taskset.txt`
   Wait and get the scheduling trace:
   `> cat /proc/cister_trace > trace.txt`

3 Parametrizing LIFO tasks

Adding a new system cal for parametrizing LIFO tasks, namely, assigning a task identifier from user space.

1. Open `linux-4.12.4-cister/arch/x86/entry/syscalls/syscall_64.tbl` file and add a new entry:

```
#
# 64-bit system call numbers and entry vectors
#
# The format is:
# <number> <abi> <name> <entry point>
#
# The abi is "common", "64" or "x32" for this file.
#
0 common read   sys_read
1 common write   sys_write
2 common open   sys_open
3 common close   sys_close
...
330 common pkey_alloc  sys_pkey_alloc
331 common pkey_free  sys_pkey_free
332 common statx   sys_statx

#ifdef CONFIG_CISTER_FRAMEWORK
333 common cister_tracing  sys_cister_tracing

334 common cister_set_task_id sys_cister_set_task_id

#endif

#
# x32-specific system call numbers start at 512 to avoid cache impact
# for native 64-bit operation.
#
512 x32 rt_sigaction  compat_sys_rt_sigaction
...
```

2. Open `linux-4.12.4-cister/kernel/cister/syscalls.h` and

```
#ifndef __SYSCALLS_H
#define __SYSCALLS_H

asmlinkage long sys_cister_tracing(int enable);

asmlinkage long sys_cister_set_task_id(int id);


#endif
```

3. Open `linux-4.12.4-cister/kernel/cister/syscalls.c` file and

```
#include <linux/syscalls.h>
#include "trace.h"

asmlinkage long sys_cister_tracing(int enable)
{
#ifdef CONFIG_CISTER_TRACING
  enable_tracing(enable);
#endif
  return 0;
}

asmlinkage longsys_cister_set_task_id(int pid)
{
#ifdef CONFIG_CISTER_SCHED_LIFO_POLICY
  set_task_id(id);
#endif
  return 0;
}
```

4. Open `linux-4.12.4-cister/kernel/cister/lf_task.h` and

```
#ifndef __LF_TASK_H_
#define __LF_TASK_H_

#include <linux/list.h>

struct lf_task{
  struct list_head node;
  int id;
};
void set_task_id(int id);
#endif
```

5. Create `linux-4.12.4-cister/kernel/cister/lf_task.c` and code:

```
#include <linux/sched.h>
#include "lf_task.h"

void set_task_id(int id){
  current->lf_task.id = id;
}
```

6. Include the `linux-4.12.4-cister/kernel/cister/lf_task.c` in the compilation process by updating the `linux-4.12.4-cister/kernel/cister/Makefile`:

```
# CISTER Framework makefile

obj-$(CONFIG_CISTER_TRACING) += trace.o
obj-$(CONFIG_CISTER_SCHED_LIFO_POLICY) += lf_rq.o sched.o lf_task.o
obj-y += syscalls.o
```

7. Update the `linux-4.12.4-cister/kernel/cister/trace.h` and `linux-4.12.4-cister/kernel/cister/trace.c` files to output the task id field:

```
#ifndef __TRACE_H_
#define __TRACE_H_
...
struct trace_evt{
  enum evt event;
  unsigned long long time;
  pid_t pid;
  int state;
  int prio;
  int policy;
  char comm[TRACE_TASK_COMM_LEN];

#ifdef CONFIG_CISTER_SCHED_LIFO_POLICY
  int task_id;
#endif

};
...
#endif
```

```
static int dequeue (char *buffer)
{
  ...

    len = sprintf(buffer,"%llu,",trace.events[trace.read_item].time);
    len += sprintf(buffer+len,"%s,",evt);

#ifdef CONFIG_CISTER_SCHED_LIFO_POLICY
    len += sprintf(buffer+len,"id,%d,",(int)trace.events[trace.read_item].task_id);
#endif

    len += sprintf(buffer+len,"pid,%d,",(int)trace.events[trace.read_item].pid);
    len += sprintf(buffer+len,"prio,%d,",(int)trace.events[trace.read_item].prio);
    len += sprintf(buffer+len,"policy,%d,",(int)trace.events[trace.read_item].policy);
...
  }
...
}

static int enqueue (enum evt event, unsigned long long time, struct task_struct *p)
{
  ...
  trace.events[trace.write_item].state  = p->state;
  trace.events[trace.write_item].prio  = p->prio;
  trace.events[trace.write_item].policy  = p->policy;
  strcpy(trace.events[trace.write_item].comm, p->comm);

#ifdef CONFIG_CISTER_SCHED_LIFO_POLICY
  trace.events[trace.write_item].task_id  = p->lf_task.id;
#endif

...
  return 1;
}
...
```

The next step is to invoke `kcompile.sh` script file to compile the Linux

kernel by typing:
`> sudo ./kcompile.sh`

Then, reboot the system.
`> sudo reboot`

8. Update the tasks source code: `defs.h` and `task.c`

```
#ifndef DEFS_H_
#define DEFS_H_

...
#define SCHED_LIFO  7
#define __NR_cister_tracing  333

#define __NR_cister_set_task_id 334

struct task {
  int id;
  unsigned long long C; //exec
  unsigned long long T; //period
  unsigned long long D; //deadline
  unsigned long long O; //first job offset
};

#endif
```

```
int main(int argc, char** argv)
{
  struct sched_param param;
  unsigned long long C,T, D, O, time0, release, deadline;
  unsigned int task_id,njobs,i=0;

  struct timespec r, now;

  task_id=atoi(argv[1]);
  C=(unsigned long long)atoll(argv[2]);
  T=(unsigned long long)atoll(argv[3]);
  D=(unsigned long long)atoll(argv[4]);
  O=(unsigned long long)atoll(argv[5])+OFFSET;
  njobs=atoi(argv[6]);
  time0 = (unsigned long long)atoll(argv[7]);

  if(syscall(__NR_cister_set_task_id, task_id)){
    printf("Error: __NR_cister_set_task_id\n");
    exit(0);
  }

  printf("Task(%d,%d): before SCHED_LIFO\n",task_id,getpid());
  param.sched_priority = 0;
  if((sched_setscheduler(0,SCHED_LIFO,&param)) == -1){
    perror("ERROR:sched_setscheduler failed");
    exit(-1);
  }
  printf("Task(%d,%d): after SCHED_LIFO\n",task_id,getpid());

  release = time0 + O;
  for(i=0;i<njobs;i++){
    r.tv_sec = release / NSEC_PER_SEC;
    r.tv_nsec = release % NSEC_PER_SEC;

    clock_gettime(CLOCK_MONOTONIC, &now);
    if(now.tv_sec * NSEC_PER_SEC + now.tv_nsec){
      printf("Task(%d,%d,%d): missed deadline: %lld\n",task_id,getpid(),i,deadline);
    }

    printf("Task(%d,%d,%d): sleeping until %lld\n",task_id,getpid(),i,release);
    clock_nanosleep(CLOCK_MONOTONIC,TIMER_ABSTIME, &r,NULL);
    printf("Task(%d,%d,%d): ready for execution\n",task_id,getpid(),i);
    deadline += D;
    do_work(C);
    release += T;

  }

  exit(task_id);
  return 0;
}
```

Compile it:

```
> make
```

9. Put the set of tasks running:

   ```
   > sudo ./launcher taskset.txt
   ```

   Wait and get the scheduling trace:

   ```
   > cat /proc/cister_trace > trace.txt
   > gedit trace.txt
   ```

```
...
196483323902,SWT_AY,id,3,pid,1731,prio,120,policy,7,state,1,task
196483860546,ENQ_RQ,id,2,pid,1730,prio,120,policy,7,state,0,task
196483867561,DEQ_RQ,id,2,pid,1730,prio,120,policy,7,state,1,task
196483867878,SWT_AY,id,2,pid,1730,prio,120,policy,7,state,1,task
196483893208,ENQ_RQ,id,3,pid,1731,prio,120,policy,7,state,256,task
196483893259,ENQ_RQ,id,2,pid,1730,prio,120,policy,7,state,256,task
196484392425,ENQ_RQ,id,1,pid,1729,prio,120,policy,7,state,0,task
196484405200,DEQ_RQ,id,1,pid,1729,prio,120,policy,7,state,1,task
196484405424,SWT_AY,id,1,pid,1729,prio,120,policy,7,state,1,task
196484405456,SWT_TO,id,2,pid,1730,prio,120,policy,7,state,0,task
196495325025,ENQ_RQ,id,1,pid,1729,prio,120,policy,7,state,256,task
196495715294,SWT_AY,id,2,pid,1730,prio,120,policy,7,state,0,task
196495718398,SWT_TO,id,1,pid,1729,prio,120,policy,7,state,0,task
197085369151,DEQ_RQ,id,1,pid,1729,prio,120,policy,7,state,1,task
...
```

4 Assignment

1. Implement Rate Monotonic (RM), Deadline Monotonic (DM) and Earliest Deadline First (EDF) real-time scheduling algorithms.