UFRR

PODER EXECUTIVO
MINISTÉRIO DA EDUCAÇÃO
UNIVERSIDADE FEDERAL DE RORAIMA
DEPARTAMENTO DE CIÊNCIA DA COMPUTAÇÃO

TUTORIAL DO PROJETO: IMPLEMENTAÇÃO DA POLÍTICA SCHEDULE BACKGROUND

ALUNOS:
Victor Hugo Souza Costa
Giovana Oliveira Moraes de Lima
Ryan Kayky Marques Rolins Bastos

Setembro de 2024
Boa Vista/Roraima

PODER EXECUTIVO
MINISTÉRIO DA EDUCAÇÃO
UNIVERSIDADE FEDERAL DE RORAIMA
DEPARTAMENTO DE CIÊNCIA DA COMPUTAÇÃO

SISTEMAS OPERACIONAIS I

TUTORIAL DO PROJETO: IMPLEMENTAÇÃO DA POLÍTICA SCHEDULE BACKGROUND

Setembro de 2024
Boa Vista/Roraima

Resumo

Este projeto tem como objetivo modificar o escalonador de processos do Linux para implementar uma nova classe de escalonador chamada SCHED_BACKGROUND. Esta nova política é projetada para suportar processos que só precisam ser executados quando o sistema não possui outras tarefas ativas. O escalonador em background deve garantir que processos com essa política sejam executados apenas na ausência de processos nas classes SCHED_OTHER, SCHED_RR, ou SCHED_FIFO. Quando múltiplos processos SCHED_BACKGROUND estão prontos para execução, eles competem pela CPU de forma semelhante aos processos da classe SCHED_OTHER.

PRIMEIROS PASSOS:

1. Verificar a versão do kernel

```
uname -r
```

2. Baixar o código fonte (estou usando a 5.15.0)

```
wget https://git.kernel.org/pub/scm/linux/kernel/git/stable/linux.git/snapshot/linux-5.15.tar.gz
```

3. Extraia o código fonte

```
tar -xzvf linux-5.15.tar.gz
```

4. Instale as dependências necessárias

```
sudo apt-get install build-essential libncurses-dev bison flex libssl-dev libelf-dev
```

5. Modificar arquivos relevantes:

• **/INCLUDE/UAPI/LINUX/SCHED.H**: Defina a nova política

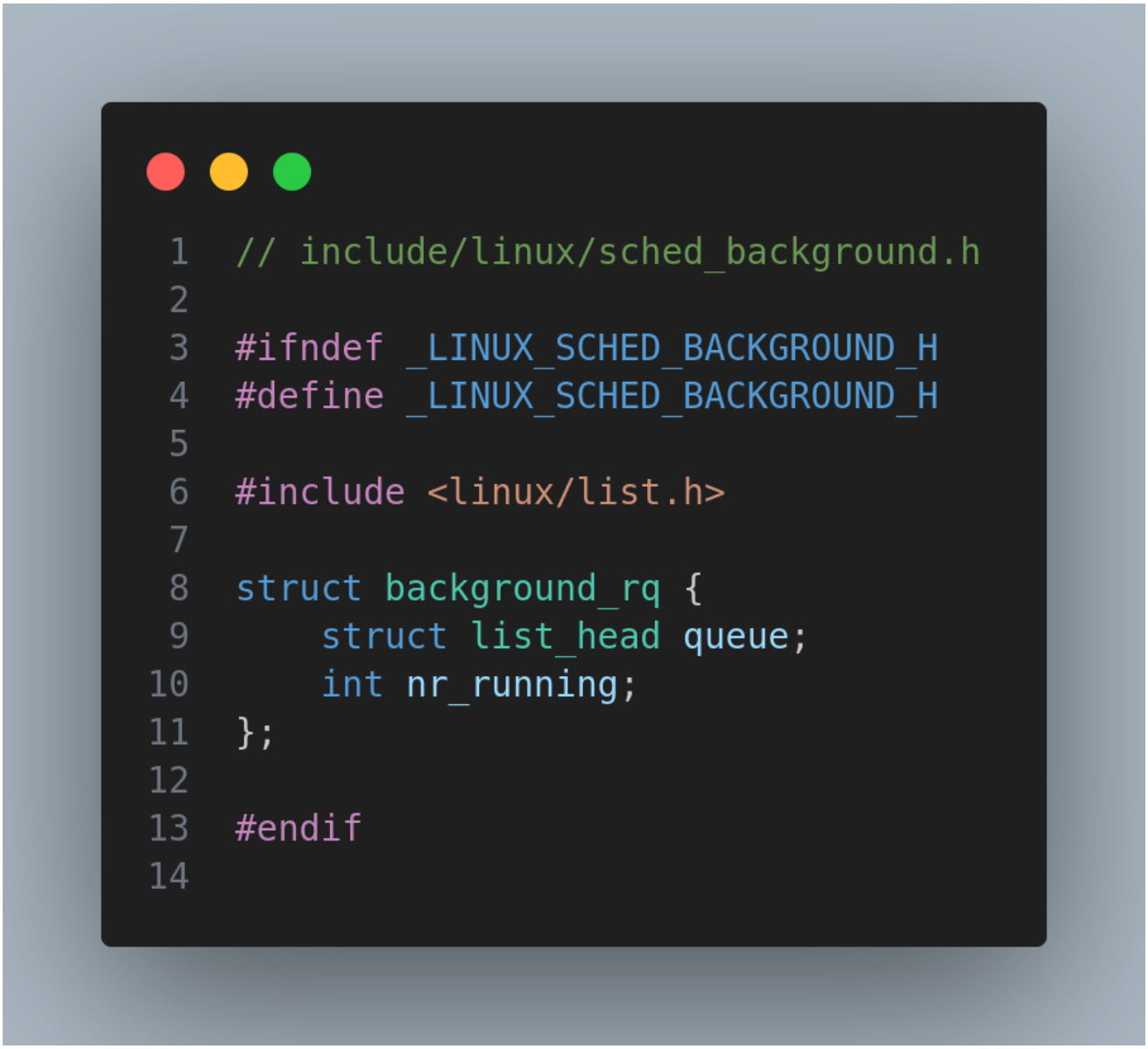
```
#define SCHED_BACKGROUND 7
```

• **/KERNEL/SCHED:** criar dois novos arquivos, *sched_background.c e sched_background.h* (especificados nas imagens abaixo)

/KERNEL/SCHED/SCHED_BACKGROUND.C:

```c
// kernel/sched/sched_background.c

#include "sched.h"

// Função para inicializar a estrutura background_rq
void init_rq_background(struct rq *rq)
{
    INIT_LIST_HEAD(&rq->background.queue);
    rq->background.nr_running = 0;
}

// Função para adicionar tarefa à fila de background
void enqueue_task_background(struct rq *rq, struct task_struct *p, int flags)
{
    add_nr_running(rq, 1);
    list_add_tail(&p->tasks, &rq->background.queue);
}

// Função para remover tarefa da fila de background
void dequeue_task_background(struct rq *rq, struct task_struct *p, int flags)
{
    sub_nr_running(rq, 1);
    list_del(&p->tasks);
}

// Função para escolher a próxima tarefa de background
struct task_struct *pick_next_task_background(struct rq *rq)
{
    struct task_struct *next = NULL;

    if (rq->background.nr_running == 0)
        return NULL;

    list_for_each_entry(next, &rq->background.queue, tasks) {
        return next;
    }

    return NULL;
}
// Estrutura para criação da classe background
const struct sched_class background_sched_class = {
    .enqueue_task       = enqueue_task_background,
    .dequeue_task       = dequeue_task_background,
    .pick_next_task     = pick_next_task_background,
};

EXPORT_SYMBOL(pick_next_task_background);

```

/KERNEL/SCHED/SCHED_BACKGROUND.H:



```c
// include/linux/sched_background.h

#ifndef _LINUX_SCHED_BACKGROUND_H
#define _LINUX_SCHED_BACKGROUND_H

#include <linux/list.h>

struct background_rq {
    struct list_head queue;
    int nr_running;
};

#endif

```

/KERNEL/SCHED/SCHED.H: adicionar o include #include "sched_background.h" e adicionar novas funções.

1. **Criar uma função *background_policy* (preferencialmente próximo da função *valid policy*):**

```c
static inline int background_policy(int policy){
   return policy == SCHED_BACKGROUND;}
```

2. **Adicionar na função *valid_policy* a nova política:**

```c
static inline bool valid_policy(int policy){
   return idle_policy(policy) || fair_policy(policy) ||
          rt_policy(policy) || dl_policy(policy) || background_policy(policy);}
```

3. **Criar uma função *task_has_background_policy*:**

```c
static inline int task_has_background_policy(struct task_struct *p){
    return (p->policy == SCHED_BACKGROUND);}
```

4. Na estrutura *struct rq* adicionar junto das outras estruturas a background:

```
struct cfs_rq                   cfs;
struct rt_rq          rt;
struct dl_rq          dl;
struct background_rq background;
```

5. Junto das outras classes *sched_class* adicionar a background:

```
…
extern const struct sched_class idle_sched_class;
extern const struct sched_class background_sched_class;
```

6. Criar *task_struct* da background:

```
…
extern struct task_struct *pick_next_task_idle(struct rq *rq);
extern struct task_struct *pick_next_task_background(struct rq *rq);
```

/KERNEL/SCHED/CORE.C: Modificar algumas funções.

1. Modificar a função *task_struct * __pick_next_task*

```
static inline struct task_struct *
__pick_next_task(struct rq *rq, struct task_struct *prev, struct rq_flags *rf){
    const struct sched_class *class;
    struct task_struct *p;

    if (likely(prev->sched_class <= &fair_sched_class &&
               rq->nr_running == rq->cfs.h_nr_running)) {

        p = pick_next_task_fair(rq, prev, rf);
        if (unlikely(p == RETRY_TASK))
            goto restart;
        if (!p) {
            put_prev_task(rq, prev);
            p = pick_next_task_idle(rq);}

        return p;}

restart:
    put_prev_task_balance(rq, prev, rf);

    for_each_class(class) {
        p = class->pick_next_task(rq);
        if (p)
            return p;}

    /* Novo caso para SCHED_BACKGROUND */
    if (likely(rq->background.nr_running > 0)) {
        p = pick_next_task_background(rq);
        if (p)
            return p;}
    BUG();}
```

2. Modificar a função *__sched_setscheduler.*

```
static int __sched_setscheduler(struct task_struct *p,
                                const struct sched_attr *attr,
                                bool user, bool pi){
    int oldpolicy = -1, policy = attr->sched_policy;
    int retval, oldprio, newprio, queued, running;
```

```
    const struct sched_class *prev_class;
    struct callback_head *head;
    struct rq_flags rf;
    int reset_on_fork;
    int queue_flags = DEQUEUE_SAVE | DEQUEUE_MOVE | DEQUEUE_NOCLOCK;
    struct rq *rq;

    /* The pi code expects interrupts enabled */
    BUG_ON(pi && 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;}

    if (attr->sched_flags & ~(SCHED_FLAG_ALL | SCHED_FLAG_SUGOV))
        return -EINVAL;

    /*
     * Valid priorities for SCHED_FIFO and SCHED_RR are
     * 1..MAX_RT_PRIO-1, valid priority for SCHED_NORMAL,
     * SCHED_BATCH, SCHED_IDLE, and SCHED_BACKGROUND is 0.
     */
    if (attr->sched_priority > MAX_RT_PRIO-1)
        return -EINVAL;
    if ((dl_policy(policy) && !__checkparam_dl(attr)) ||
        (rt_policy(policy) != (attr->sched_priority != 0)))
        return -EINVAL;

    /*
     * Allow unprivileged RT tasks to decrease priority:
     */
    if (user && !capable(CAP_SYS_NICE)) {
        if (fair_policy(policy)) {
            if (attr->sched_nice < task_nice(p) &&
                !can_nice(p, attr->sched_nice))
                return -EPERM;}

        if (rt_policy(policy)) {
            unsigned long rlim_rtprio =
                    task_rlimit(p, RLIMIT_RTPRIO);

            /* Can't set/change the rt policy: */
            if (policy != p->policy && !rlim_rtprio)
                return -EPERM;

            /* Can't increase priority: */
            if (attr->sched_priority > p->rt_priority && attr->sched_priority > rlim_rtprio)
                return -EPERM;}

        /*
         * Can't set/change SCHED_DEADLINE policy at all for now
         * (safest behavior); in the future we would like to allow
         * unprivileged DL tasks to increase their relative deadline
         * or reduce their runtime (both ways reducing utilization)
         */
        if (dl_policy(policy))
            return -EPERM;
```

```
        /*
         * Treat SCHED_IDLE as nice 20. Only allow a switch to
         * SCHED_NORMAL if the RLIMIT_NICE would normally permit it.
         */
        if (task_has_idle_policy(p) && !idle_policy(policy)) {
            if (!can_nice(p, task_nice(p)))
                return -EPERM;}

        /*
         Trate SCHED_BACKGROUND de forma semelhante a SCHED_IDLE:
         */
        if (task_has_background_policy(p) && !background_policy(policy)) {
            if (!can_nice(p, task_nice(p)))
                return -EPERM;}

        /* Can't change other user's priorities: */
        if (!check_same_owner(p))
            return -EPERM;

        /* Normal users shall not reset the sched_reset_on_fork flag: */
        if (p->sched_reset_on_fork && !reset_on_fork)
            return -EPERM;}

    if (user){
        if (attr->sched_flags & SCHED_FLAG_SUGOV)
            return -EINVAL;

        retval = security_task_setscheduler(p);
        if (retval)
            return retval;}

    /* Update task specific "requested" clamps */
    if (attr->sched_flags & SCHED_FLAG_UTIL_CLAMP) {
        retval = uclamp_validate(p, attr);
        if (retval)
            return retval;}

    if (pi)
        cpuset_read_lock();

    /*
     * Make sure no PI-waiters arrive (or leave) while we are
     * changing the priority of the task:
     *
     * To be able to change p->policy safely, the appropriate
     * runqueue lock must be held.
     */
    rq = task_rq_lock(p, &rf);
    update_rq_clock(rq);

    /*
     * Changing the policy of the stop threads is a very bad idea:
     */
    if (p == rq->stop) {
        retval = -EINVAL;
        goto unlock;}

    /*
     * If not changing anything there's no need to proceed further,
     * but store a possible modification of reset_on_fork.
     */
    if (unlikely(policy == p->policy)) {
        if (fair_policy(policy) && attr->sched_nice != task_nice(p))
            goto change;
        if (rt_policy(policy) && attr->sched_priority != p->rt_priority)
```

```
            goto change;
        if (dl_policy(policy) && dl_param_changed(p, attr))
            goto change;
        if (attr->sched_flags & SCHED_FLAG_UTIL_CLAMP)
            goto change;

        p->sched_reset_on_fork = reset_on_fork;
        retval = 0;
        goto unlock;}

change:

    if (user) {
#ifdef CONFIG_RT_GROUP_SCHED
        /*
         * Do not allow realtime tasks into groups that have no runtime
         * assigned.
         */
        if (rt_bandwidth_enabled() && rt_policy(policy) &&
                task_group(p)->rt_bandwidth.rt_runtime == 0 &&
                !task_group_is_autogroup(task_group(p))) {
            retval = -EPERM;
            goto unlock;}
#endif
#ifdef CONFIG_SMP
        if (dl_bandwidth_enabled() && dl_policy(policy) &&
                !(attr->sched_flags & SCHED_FLAG_SUGOV)) {
            cpumask_t *span = rq->rd->span;

            /*
             * Don't allow tasks with an affinity mask smaller than
             * the entire root_domain to become SCHED_DEADLINE. We
             * will also fail if there's no bandwidth available.
             */
            if (!cpumask_subset(span, p->cpus_ptr) ||
                rq->rd->dl_bw.bw == 0) {
                retval = -EPERM;
                goto unlock;
            }
        }

#endif
    }

    /* Re-check policy now with rq lock held: */
    if (unlikely(oldpolicy != -1 && oldpolicy != p->policy)) {
        policy = oldpolicy = -1;
        task_rq_unlock(rq, p, &rf);
        if (pi)
            cpuset_read_unlock();
        goto recheck;}

    /*
     * If setscheduling to SCHED_DEADLINE (or changing the parameters
     * of a SCHED_DEADLINE task) we need to check if enough bandwidth
     * is available.
     */
    if ((dl_policy(policy) || dl_task(p)) && sched_dl_overflow(p, policy, attr)) {
        retval = -EBUSY;
        goto unlock;}

    p->sched_reset_on_fork = reset_on_fork;
    oldprio = p->prio;

    newprio = __normal_prio(policy, attr->sched_priority, attr->sched_nice);
```

```
    if (pi) {
        /*
         * Take priority boosted tasks into account. If the new
         * effective priority is unchanged, we just store the new
         * normal parameters and do not touch the scheduler class and
         * the runqueue. This will be done when the task deboost
         * itself.
         */
        newprio = rt_effective_prio(p, newprio);
        if (newprio == oldprio)
            queue_flags &= ~DEQUEUE_MOVE;}

    queued = task_on_rq_queued(p);
    running = task_current(rq, p);
    if (queued)
        dequeue_task(rq, p, queue_flags);
    if (running)
        put_prev_task(rq, p);

    prev_class = p->sched_class;

    if (!(attr->sched_flags & SCHED_FLAG_KEEP_PARAMS)) {
        __setscheduler_params(p, attr);
        __setscheduler_prio(p, newprio);}
    __setscheduler_uclamp(p, attr);

    if (queued) {
        /*
         * We enqueue to tail when the priority of a task is
         * increased (user space view).
         */
        if (oldprio < p->prio)
            queue_flags |= ENQUEUE_HEAD;

        enqueue_task(rq, p, queue_flags);}
    if (running)
        set_next_task(rq, p);

    check_class_changed(rq, p, prev_class, oldprio);

    /* Avoid rq from going away on us: */
    preempt_disable();
    head = splice_balance_callbacks(rq);
    task_rq_unlock(rq, p, &rf);

    if (pi) {
        cpuset_read_unlock();
        rt_mutex_adjust_pi(p);}

    /* Run balance callbacks after we've adjusted the PI chain: */
    balance_callbacks(rq, head);
    preempt_enable();

    return 0;

unlock:
    task_rq_unlock(rq, p, &rf);
    if (pi)
        cpuset_read_unlock();
    return retval;}
```

3. **Modificar também SYSCALL_DEFINE1(sched_get_priority_max, int, policy) e SYSCALL_DEFINE1(sched_get_priority_min, int, policy)**

```
SYSCALL_DEFINE1(sched_get_priority_max, int, policy){
    int ret = -EINVAL;

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

/**
 * sys_sched_get_priority_min - return minimum RT priority.
 * @policy: scheduling class.
 *
 * Return: On success, this syscall returns the minimum
 * rt_priority that can be used by a given scheduling class.
 * On failure, a negative error code is returned.
 */
SYSCALL_DEFINE1(sched_get_priority_min, int, policy)
{
    int ret = -EINVAL;

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

7. Configure o kernel:

Existem duas opções de configuração do kernel:

1. Utilizando o comando `cp /boot/config-$(uname -r) .config`, que copia as configurações do kernel atual. No entanto, se o kernel não for o mesmo ou se houver a necessidade de um kernel específico, será necessário realizar modificações.
2. Usando o comando `make defconfig`, que carrega configurações padrão, onde diversos drivers e outros componentes estão desativados.

Para personalizar as configurações, pode-se utilizar o comando `make menuconfig`. Caso essa ferramenta não esteja disponível, pode-se prosseguir sem ela.

8. Compile o kernel

```
make -j$(nproc)
```

9. Instale o kernel

```
sudo make modules_install
sudo make install
```

10. Atualize o grub e reinicie

```
sudo update-grub
sudo reboot
```

11. Verifique a nova versão do kernel

```
uname -r
```

12. Teste a nova política

Para fins de teste, foram criados dois arquivos .c: *teste_background.c* e *teste_performance.c*.

teste_background.c: Destinado a verificar se a nova política foi criada com sucesso.

```c
#include <stdio.h>
#include <sched.h>
#include <unistd.h>

#define SCHED_BACKGROUND 7

int main() {
    struct sched_param param;
    param.sched_priority = 0;

    if (sched_setscheduler(0, SCHED_BACKGROUND, &param) == -1) {
        perror("sched_setscheduler");
        return 1;}

    printf("SCHED_BACKGROUND aplicada com sucesso!\n");
    return 0;}
```

TESTE_PERFORMANCE.C: Para saber a performance da nova política nova

```c
#include <sys/time.h>
#include <sys/resource.h>
#include <stdio.h>
int main() {
struct timeval start, end;
struct rusage usage;
gettimeofday(&start, NULL);
for (long i = 0; i < 1000000000; i++);
gettimeofday(&end, NULL);
getrusage(RUSAGE_SELF, &usage);
long seconds = end.tv_sec - start.tv_sec;
long micros = ((seconds * 1000000) + end.tv_usec) - start.tv_usec;
printf("Tempo de wallclock: %ld microssegundos\n", micros);
printf("Tempo de usuário: %ld microssegundos\n", usage.ru_utime.tv_sec * 1000000 +
usage.ru_utime.tv_usec);
printf("Tempo de sistema: %ld microssegundos\n", usage.ru_stime.tv_sec * 1000000 +
usage.ru_stime.tv_usec);
return 0;
}
```

Após a criação basta usar os comandos:

```
gcc -o teste_background teste_background.c
sudo ./teste_background
```

```
gcc -o teste_performance teste_performance.c
sudo ./teste_performance
```