Implementação de Escalonamento Rate Monotonic com Protocolos de Prioridade no Kernel Linux

Reinaldo Reis 1201560, Nuno Castro 1240160

Instituto Superior de Engenharia do Porto
https://www.isep.ipp.pt

Abstract. Este trabalho apresenta a implementação e análise de três mecanismos fundamentais em sistemas de tempo real:*Rate Monotonic Scheduling* (RM), *Priority Inheritance Protocol* (PIP) e *Priority Ceiling Protocol* (PCP). O *Rate Monotonic* é utilizado como política de escalonamento de prioridade fixa, enquanto o PIP e o PCP são aplicados para mitigar a inversão de prioridade em ambientes com recursos partilhados. As soluções foram integradas no kernel do Linux, permitindo a avaliação prática do seu desempenho. Os testes realizados demonstram a eficácia de cada abordagem na gestão de tarefas e sincronização de recursos.

Key words: Tempo Real, Rate Monotonic, Priority Inheritance Protocol, Priority Ceiling Protocol, Recursos Partilhados, Kernel.

1 Introdução

Os sistemas de tempo real desempenham um papel crucial em diversas aplicações críticas, onde o cumprimento de restrições temporais é tão importante quanto a funcionalidade. Nesse contexto, algoritmos de escalonamento como o *Rate Monotonic Scheduling* e protocolos de sincronização de recursos, como o *Priority Inheritance Protocol* (PIP) e o *Priority Inheritance Protocol* (PCP), são fundamentais para garantir a previsibilidade e a eficiência na gestão de tarefas e no acesso a recursos compartilhados. Este trabalho foca na implementação e análise desses três mecanismos no kernel do Linux, um ambiente amplamente utilizado em sistemas embarcados e de tempo real. A integração desses algoritmos e protocolos no kernel permite explorar como eles se comportam em cenários práticos, avaliando sua eficácia e limitações.

2 Fundamentação Teórica

2.1 Rate Monotonic Scheduling

O *Rate Monotonic Scheduling* é um algoritmo clássico e determinístico de escalonamento com prioridade estática, amplamente utilizado em sistemas de tempo real com tarefas periódicas. No *Rate Monotonic Scheduling*, utiliza-se a política de

escalonamento *Rate Monotonic*, onde a prioridade de cada tarefa é atribuída com base no seu período. Quanto menor o período de uma tarefa, maior será a sua prioridade.

Os principais fundamentos da política *Rate Monotonic* são:

- As tarefas são independentes.
- O sistema é preemptivo, permitindo interrupções por tarefas de maior prioridade.

Limiar de utilização em *Rate Monotonic*

Uma característica fundamental da política *Rate Monotonic* é a existência de um limiar de utilização que determina se um conjunto de tarefas periódicas é garantidamente escalonável. Para n tarefas, o limite teórico de utilização é dado por:

$$U_{RM}^{*} = n \cdot (2^{1/n} - 1)$$

Com o aumento do número de tarefas, esse limiar converge para:

$$\lim_{n \to \infty} U_{RM}^{*} = \ln(2) \approx 0{,}693$$

Portanto, se a soma das utilizações das tarefas,

$$U = \sum_{i=1}^{n} \frac{C_i}{T_i},$$

for menor ou igual a U_{RM}^{*}, então o conjunto é garantidamente escalonável segundo a política *Rate Monotonic*. Caso a utilização total ultrapasse este limiar, não se pode garantir a escalonabilidade sem uma análise mais aprofundada.

2.2 Priority Inheritance Protocol (PIP)

O *Priority Inheritance Protocol* (PIP) é uma técnica amplamente utilizada em sistemas de tempo real para gerir o acesso a recursos partilhados em ambientes de escalonamento com prioridades. Este protocolo foi elaborado com o objetivo de mitigar o problema de inversão de prioridades.

No PIP, quando uma tarefa de baixa prioridade está a utilizar um recurso de que uma tarefa de prioridade mais alta necessita, a sua prioridade é temporariamente elevada para igualar a prioridade da tarefa bloqueada. Esta elevação permanece até que o recurso seja libertado, momento em que a tarefa de baixa prioridade retoma a sua prioridade original.

Este mecanismo assegura que a tarefa de baixa prioridade não seja preemptada por outras tarefas de prioridade intermédia, reduzindo assim o tempo em que a tarefa de alta prioridade permanece bloqueada.

A aplicação do PIP é particularmente relevante em sistemas de tempo real onde múltiplas tarefas competem por recursos comuns, como semáforos ou mutexes. A sua adoção é fundamental para garantir que tarefas críticas respeitem os seus prazos, mesmo em cenários de conflito no acesso a recursos compartilhados.

2.3 Priority Ceiling Protocol (PCP)

O *Priority Ceiling Protocol* (PCP) é outro protocolo de compartilhamento de recursos projetado para evitar a inversão de prioridade e reduzir os tempos de bloqueio em sistemas de tempo real. Diferentemente do PIP, que ajusta prioridades dinamicamente, o PCP atribui uma prioridade estática de "teto" a cada recurso, igual à maior prioridade de qualquer tarefa que possa acessá-lo. Uma tarefa só pode bloquear um recurso se sua prioridade for maior que o teto de todos os recursos atualmente bloqueados, evitando *deadlocks* e limitando o tempo de bloqueio a, no máximo, uma seção crítica por recurso.

3 Implementação

3.1 Rate Monotonic Scheduling

3.1.1 Estruturas de Dados

3.1.1.1 Fila de Execução baseada em Rate Monotonic

A estrutura `rm_rq` representa a fila de execução específica para o algoritmo *Rate Monotonic* no kernel Linux:

```
1  struct rm_rq{
2      struct rb_root tasks;
3      raw_spinlock_t lock;
4      unsigned nr_running;
5  };
```

- `struct rb_root tasks`: Utiliza uma red-black tree para armazenar as tarefas ordenadas por período. Esta estrutura de dados é ideal para o *Rate Monotonic* porque permite inserção, remoção e pesquisa em tempo O(log n), mantendo automaticamente a ordenação necessária para esta política de escalonamento.
- `raw_spinlock_t lock`: Spinlock para proteger as operações na fila de execução, garantindo atomicidade em ambientes multiprocessador.
- `unsigned nr_running`: Contador do número de tarefas prontas para execução na fila *Rate Monotonic*.

3.1.1.2 Entidade de Escalonamento Rate Monotonic

A estrutura `sched_rm_entity` representa as informações específicas de escalonamento para cada tarefa *Rate Monotonic*:

```
1  struct sched_rm_entity{
2      struct rb_node node;
3      u64 period;
4  };
```

– `struct rb_node node`: Nó da red–black tree que permite à tarefa ser inserida na estrutura `rm_rq`. Este campo estabelece a ligação entre a tarefa e a fila de execução.
– `u64 period`: Período da tarefa em nanossegundos. Este é o parâmetro fundamental do *Rate Monotonic*, pois determina diretamente a prioridade da tarefa (quanto menor o período, maior a prioridade).

3.1.2 Implementação do Rate Monotonic Scheduling

3.1.2.1 Cálculo de Prioridade Rate Monotonic

```
1  #define RM_MIN_PRIO 10
2  #define RM_MAX_PRIO (MAX_RT_PRIO - 1)
3
4  static int calculate_rm_priority(u64 period)
5  {
6      int prio;
7
8      // Periodo zero e invalido, retorna prioridade minima
9      if (period == 0)
10         return RM_MIN_PRIO;
11
12     // Mapeia logaritmicamente o periodo para a gama de prioridades
13     // Periodos menores resultam em prioridades numericas menores (maior
           prioridade)
14     prio = RM_MIN_PRIO + (int)((RM_MAX_PRIO - RM_MIN_PRIO) *
15                                ilog2(period) / ilog2(U64_MAX));
16
17     return prio;
18 }
```

A função `calculate_rm_priority` implementa o princípio fundamental do *Rate Monotonic*: tarefas com períodos menores recebem prioridades mais altas. Para garantir uma distribuição eficiente e não interferir com tarefas críticas do sistema, definimos que a prioridade mínima das nossas tarefas seria 10 e a máxima seria 99, correspondendo ao valor máximo das tarefas de tempo real críticas do Linux. O cálculo da prioridade utiliza um mapeamento logarítmico baseado na seguinte fórmula:

$$prio = RM_{min} + \left\lfloor \frac{(RM_{max} - RM_{min}) \cdot \log_2(period)}{\log_2(U64_{MAX})} \right\rfloor \tag{1}$$

O logaritmo base 2 do período ($\log_2(period)$) produz valores que crescem lentamente à medida que o período aumenta, permitindo uma distribuição mais equilibrada das prioridades mesmo quando os períodos variam exponencialmente. A divisão por $\log_2(U64_{MAX})$ ajusta este valor para o intervalo , garantindo que o resultado final se mantenha sempre dentro dos limites definidos pelas constantes RM_{min} e RM_{max}.

3.1.2.2 Adicionar Tarefas na Red-Black Tree

```c
static void add_task_rm(struct rq *rq, struct task_struct *p){
    struct rb_node **new = &rq->rm.tasks.rb_node;
    struct rb_node *parent = NULL;
    struct sched_rm_entity *entry;

    // Percorre a arvore para encontrar o ponto de insercao correto
    while (*new) {
        parent = *new;
        entry = rb_entry(parent, struct sched_rm_entity, node);

        // Periodos menores vao para a esquerda (maior prioridade RM)
        if (p->rm.period < entry->period) {
            new = &parent->rb_left;
        } else {
            new = &parent->rb_right;
        }
    }

    // Liga o novo no a arvore e rebalanceia
    rb_link_node(&p->rm.node, parent, new);
    rb_insert_color(&p->rm.node, &rq->rm.tasks);
}
```

A função `add_task_rm` implementa a inserção de tarefas na estrutura de dados fundamental do escalonador *Rate Monotonic*: uma red-black tree ordenada por período. Esta função é crucial para manter a lógica de ordenação que permite ao algoritmo *Rate Monotonic* selecionar eficientemente a tarefa de maior prioridade.

```c
static void enqueue_task_rm(struct rq *rq, struct task_struct *p, int flags)
{
    // Calcula e define a prioridade baseada no periodo
    p->prio = calculate_rm_priority(p->rm.period);
    p->static_prio = p->prio;

    raw_spin_lock(&rq->rm.lock);

    // Insere na arvore ordenada por periodo
    add_task_rm(rq, p);

    // Atualiza o apontador para a tarefa de maior prioridade
    rq->rm.task = p;
    rq->rm.nr_running++;
    add_nr_running(rq, 1);

    raw_spin_unlock(&rq->rm.lock);

    #ifdef CONFIG_MOKER_TRACING
    moker_trace(ENQUEUE_RQ, p);
    #endif
}
```

A função `enqueue_task_rm` representa uma das operações mais fundamentais do escalonador *Rate Monotonic*, sendo responsável por todo o processo de integração de uma nova tarefa no sistema de escalonamento. Esta função coordena múltiplas operações críticas que garantem que cada tarefa seja correctamente posicionada na estrutura de dados do escalonador, respeitando rigorosamente os princípios teóricos do algoritmo *Rate Monotonic*.
O processo inicia com o cálculo da prioridade numérica através da função `calculate_rm_priority`, que implementa o mapeamento logarítmico baseado no período da tarefa.
A secção crítica da operação é protegida por um spinlock que garante atomicidade em sistemas multiprocessador, evitando condições de corrida que poderiam

comprometer a integridade da estrutura de dados. Dentro desta secção protegida, a tarefa é inserida na red-black tree através da função `add_task_rm`, que mantém automaticamente a invariante de ordenação por período. Esta estrutura de dados auto-balanceada é essencial para garantir o funcionamento eficiente do algoritmo de escalonamento.
Os metadados da fila são cuidadosamente actualizados durante o processo de enfileiramento. Simultaneamente, os contadores `nr_running` tanto local quanto global são incrementados, fornecendo informações cruciais que permitem ao escalonador principal tomar decisões sobre a carga actual do sistema.
O processo é finalizado com o registo opcional do evento de enfileiramento no sistema de rastreamento, permitindo análise posterior do comportamento do escalonador. Todo este processo mantém uma complexidade temporal de O(log n) devido à inserção na red-black tree, garantindo que o overhead de escalonamento permaneça controlado mesmo com um grande número de tarefas no sistema.

3.1.2.3 Remoção Tarefas da Red-Black Tree

```
1  static bool remove_task_rm(struct rq *rq, struct task_struct *p)
2  {
3      if (RB_EMPTY_NODE(&p->rm.node)) {
4          // The task is not in the tree
5          return false;
6      }
7
8      // Remove the task from the red-black tree
9      rb_erase(&p->rm.node, &rq->rm.tasks);
10     RB_CLEAR_NODE(&p->rm.node);
11
12     return true;
13 }
```

A função `remove_task_rm` implementa a remoção segura de uma tarefa da red-black tree ordenada por período. Esta função inicia com uma verificação através de `RB_EMPTY_NODE`, que determina se o nó está presente na árvore e assim evita uma tentativa de remoção de um nó inexistente. Caso a tarefa não esteja na árvore, retorna `false` imediatamente. Caso contrário, procede-se com a remoção através de `rb_erase`, que mantém automaticamente as propriedades da red-black tree. Após a remoção, `RB_CLEAR_NODE` limpa o nó da tarefa, prevenindo futuras operações inválidas.

```
1  static void dequeue_task_rm(struct rq *rq, struct task_struct *p, int flags)
2  {
3      raw_spin_lock(&rq->rm.lock);
4
5      if (remove_task_rm(rq, p)) {
6          rq->rm.nr_running--;
7          sub_nr_running(rq, 1);
8      }
9
10     raw_spin_unlock(&rq->rm.lock);
11
12     #ifdef CONFIG_MOKER_TRACING
13     moker_trace(DEQUEUE_RQ, p);
14     #endif
15 }
```

A função `dequeue_task_rm` coordena o processo completo de remoção de uma tarefa do escalonador *Rate Monotonic*. O processo inicia com a aquisição do spinlock, garantindo exclusão mútua durante operações críticas. A chamada à função `remove_task_rm` efectua a remoção da red-black tree, com verificação do valor de retorno. Apenas quando confirmada, os contadores `rq->rm.nr_running` e `sub_nr_running` são decrementados. A libertação do spinlock marca o fim da secção crítica. O sistema de rastreamento opcional regista o evento através de `moker_trace`.

3.1.2.4 Seleção da próxima tarefa a executar

```
1  static struct task_struct *pick_next_task_rm(struct rq *rq)
2  {
3      struct rb_node *node;
4      struct sched_rm_entity *entry;
5      struct task_struct *p = NULL;
6
7      if (!rq) {
8          printk(KERN_ERR "RM: NULL run queue in pick_next_task_rm\n");
9          return NULL;
10     }
11
12     raw_spin_lock(&rq->rm.lock);
13
14     node = rb_first(&rq->rm.tasks);
15     if (node) {
16         entry = rb_entry(node, struct sched_rm_entity, node);
17         if (entry) {
18             p = container_of(entry, struct task_struct, rm);
19         }
20     }
21
22     raw_spin_unlock(&rq->rm.lock);
23
24     return p;
25 }
```

A função `pick_next_task_rm` implementa a política de escalonamento *Rate Monotonic* ao selecionar a tarefa com menor período (maior prioridade). Esta função inicia com uma verificação de segurança da fila de execução, seguida da aquisição do spinlock para proteger o acesso à red-black tree. Utiliza `rb_first` para obter o nó mais à esquerda da árvore em tempo O(log n), correspondendo à tarefa com menor período. A conversão da entidade de escalonamento para `task_struct` é realizada através de `container_of`. A função garante que sempre seja retornada a tarefa de maior prioridade *Rate Monotonic* disponível para execução.

3.1.2.5 Implementação da System Call para Definir o Período no Rate Monotonic

```
1  SYSCALL_DEFINE2(set_rm_period, pid_t, pid, u64, period) {
2      return sys_set_rm_period(pid, period);
3  }
4  long sys_set_rm_period(pid_t pid, u64 period) {
5      #ifdef CONFIG_MOKER_SCHED_RM_POLICY
6      struct task_struct *task;
```

```
7
8       if (period <= 0)
9           return -EINVAL;
10
11      rcu_read_lock();
12      task = find_task_by_vpid(pid);
13      if (!task) {
14          rcu_read_unlock();
15          return -ESRCH;
16      }
17
18      get_task_struct(task);
19      rcu_read_unlock();
20
21      task_lock(task);
22      task->rm.period = period;
23      task_unlock(task);
24
25      put_task_struct(task);
26      #endif
27      return 0;
28  }
```

Para permitir a configuração do período de uma thread no contexto do escalonamento *Rate Monotonic* a partir da user space, foi criada a system call **set_rm_period**. Esta chamada de sistema recebe como parâmetros o identificador do processo (pid) e o valor do período (period), possibilitando a comunicação direta com o kernel para ajustar os parâmetros de escalonamento de uma tarefa específica.

3.2 Priority Inheritance Protocol (PIP)

3.2.1 Estruturas de Dados

3.2.1.1 Fila de Espera baseada em Priority Inheritance Protocol

A estrutura `pip_mutex_wq` representa a fila de espera específica para o mutex com *Priority Inheritance Protocol* no kernel Linux:

```
1  struct pip_mutex_wq{
2      struct rb_root tasks;
3      atomic_t flag;
4      raw_spinlock_t lock;
5      struct task_struct *holder;
6      int holder_original_prio;
7  };
```

- `struct rb_root tasks`: Utiliza uma red-black tree para armazenar as tarefas bloqueadas ordenadas por prioridade. Esta estrutura de dados é ideal para o PIP porque permite inserção, remoção e pesquisa em tempo O(log n), mantendo automaticamente a ordenação necessária para a gestão de prioridades.
- `atomic_t flag`: Flag atómica que indica o estado do mutex (0 = desbloqueado, 1 = bloqueado). A atomicidade garante operações thread-safe sem necessidade de sincronização adicional.
- `raw_spinlock_t lock`: Spinlock para proteger as operações na fila de espera, garantindo atomicidade em ambientes multiprocessador durante modificações da estrutura.

- `struct task_struct *holder`: Apontador para a tarefa que detém atualmente o mutex. Este campo é fundamental para implementar a herança de prioridade, permitindo identificar rapidamente qual tarefa deve herdar a prioridade.
- `int holder_original_prio`: Prioridade original do detentor do mutex antes da herança de prioridade. Este campo é essencial para restaurar a prioridade original quando o mutex é libertado.

3.2.1.2 Nó de Tarefa em Espera no Priority Inheritance Protocol

A estrutura `pip_mutex_node` representa as informações específicas de cada tarefa bloqueada na fila de espera do mutex PIP:

```
1  struct pip_mutex_node{
2      struct rb_node node;
3      struct task_struct *task;
4      int priority;
5  };
```

- `struct rb_node node`: Nó da red-black tree que permite à tarefa ser inserida na estrutura `pip_mutex_wq`. Este campo estabelece a ligação entre a tarefa bloqueada e a fila de espera ordenada.
- `struct task_struct *task`: Apontador para a estrutura da tarefa bloqueada. Este campo permite acesso directo às informações completas da tarefa para operações de desbloqueio e herança de prioridade.
- `int priority`: Prioridade da tarefa no momento do bloqueio. Este campo é fundamental para manter a ordenação na red-black tree e para determinar qual prioridade deve ser herdada pelo detentor do mutex.

3.2.2 Implementação das Funções do Priority Inheritance Protocol

3.2.2.1 Aquisição do Mutex com Herança de Prioridade

```
1  int enqueue_pip_mutex_task(struct task_struct *p)
2  {
3      struct rb_node **link = &pip_wq.tasks.rb_node, *parent = NULL;
4      struct pip_mutex_node *entry;
5      struct pip_mutex_node *new_node;
6
7      new_node = kmalloc(sizeof(*new_node), GFP_KERNEL);
8      if (!new_node)
9          return -ENOMEM;
10
11     new_node->task = p;
12     new_node->priority = p->prio;
13
14     raw_spin_lock(&pip_wq.lock);
15
16     while (*link) {
17         parent = *link;
18         entry = rb_entry(parent, struct pip_mutex_node, node);
19
20         if (new_node->priority < entry->priority)
21             link = &(*link)->rb_left;
22         else
23             link = &(*link)->rb_right;
```

```
24      }
25
26      rb_link_node(&new_node->node, parent, link);
27      rb_insert_color(&new_node->node, &pip_wq.tasks);
28
29      return 0;
30  }
```

A função `enqueue_pip_mutex_task` implementa a inserção de tarefas na red-black tree ordenada por prioridade. Esta função cria um novo nó para a tarefa bloqueada, preservando a sua prioridade no momento do bloqueio. O algoritmo de inserção percorre a árvore comparando prioridades para encontrar a posição correcta, mantendo a invariante de que tarefas com prioridades menores (numericamente) ficam à esquerda. A inserção é protegida por spinlock e utiliza as funções padrão da red-black tree do kernel para manter o balanceamento.

```
1   void change_task_prio_pip(struct task_struct *p, int new_prio)
2   {
3       struct rq_flags rf;
4       struct rq *rq;
5       int old_prio = p->prio;
6
7       p->prio = new_prio;
8       p->static_prio = new_prio;
9
10      if (p->policy == SCHED_RM) {
11          rq = task_rq_lock(p, &rf);
12
13          p->sched_class->prio_changed(rq, p, old_prio);
14
15          task_rq_unlock(rq, p, &rf);
16      }
17
18      #ifdef CONFIG_MOKER_TRACING
19      moker_trace(MUTEX_UPDATE, p);
20      #endif
21  }
22
23  void lock_pip_mutex(void) {
24      struct task_struct *p = current;
25
26      while (!atomic_add_unless(&pip_wq.flag, 1, 1)) {
27          raw_spin_lock(&pip_wq.lock);
28          set_current_state(TASK_INTERRUPTIBLE);
29
30          enqueue_pip_mutex_task(p);
31
32          if (pip_wq.holder && p->prio < pip_wq.holder->prio) {
33              // Aumentar prioridade temporariamente
34              change_task_prio_pip(pip_wq.holder, p->prio);
35          }
36          raw_spin_unlock(&pip_wq.lock);
37
38          schedule();
39      }
40
41      // Adquiriu o lock: registar como holder
42      raw_spin_lock(&pip_wq.lock);
43      pip_wq.holder = p;
44      pip_wq.holder_original_prio = p->prio;
45      raw_spin_unlock(&pip_wq.lock);
46
47      #ifdef CONFIG_MOKER_TRACING
48      moker_trace(MUTEX_LOCK, p);
49      #endif
```

```
50 }
```

A função `lock_pip_mutex` implementa o mecanismo de aquisição de um mutex utilizando o protocolo de herança de prioridade. O processo inicia-se com uma tentativa atómica de definir a flag do mutex, indicando a sua aquisição. Caso esta tentativa falhe (ou seja, o mutex já se encontra ocupado), a tarefa é colocada num estado interruptível e adicionada a uma fila de espera. Durante este bloqueio, é aplicado o protocolo de herança de prioridade: se a tarefa bloqueada tiver prioridade superior à da tarefa atualmente detentora do mutex, a prioridade do detentor é temporariamente elevada por meio da função **change_task_prio_pip**. Esta atualização visa evitar o fenómeno de inversão de prioridade, garantindo que tarefas de elevada prioridade não ficam indefinidamente bloqueadas por tarefas de menor prioridade. Uma vez adquirida a posse do mutex, a tarefa é registada como o novo detentor (`holder`), e a sua prioridade original é armazenada para futura recuperação.

3.2.2.2 Libertação do Mutex e Restauração de Prioridade

```
1  struct task_struct * dequeue_pip_mutex_task(void)
2  {
3      struct rb_node *node;
4      struct pip_mutex_node *entry;
5      struct task_struct *task = NULL;
6
7      node = rb_first(&pip_wq.tasks);
8      if (!node) {
9          raw_spin_unlock(&pip_wq.lock);
10         return NULL;
11     }
12
13     entry = rb_entry(node, struct pip_mutex_node, node);
14     task = entry->task;
15
16     rb_erase(&entry->node, &pip_wq.tasks);
17     kfree(entry);
18
19     #ifdef CONFIG_MOKER_TRACING
20     moker_trace(DEQUEUE_WQ, task);
21     #endif
22
23     return task;
24 }
```

A função `dequeue_pip_mutex_task` selecciona e remove a tarefa de maior prioridade da fila de espera. Utiliza `rb_first` para obter o nó mais à esquerda da árvore, correspondendo à tarefa com menor valor numérico de prioridade (maior prioridade real). Após extrair a tarefa, o nó é removido da árvore e a memória é libertada. Esta implementação garante que o *Priority Inheritance Protocol* sempre acorde a tarefa de maior prioridade quando o mutex é libertado.

```
1  void unlock_pip_mutex(void) {
2      struct task_struct *p = current;
3      struct task_struct *t = NULL;
4
5      raw_spin_lock(&pip_wq.lock);
6
7      if (p->prio != pip_wq.holder_original_prio) {
8          // Restaurar prioridade original
9          change_task_prio_pip(p, pip_wq.holder_original_prio);
```

```
10      }
11
12      pip_wq.holder_original_prio = -1;
13      pip_wq.holder = NULL;
14
15      t = dequeue_pip_mutex_task();
16      raw_spin_unlock(&pip_wq.lock);
17      if (t){
18          if (!wake_up_process(t)) {
19              printk(KERN_ERR "BUG: wake up process failed: %d\n", t->pid);
20          }
21      }
22
23      atomic_set(&pip_wq.flag, 0);
24
25      #ifdef CONFIG_MOKER_TRACING
26      moker_trace(MUTEX_UNLOCK, p);
27      #endif
28  }
```

A função `unlock_pip_mutex` coordena a libertação do mutex e a restauração das prioridades originais. Esta função verifica se a prioridade actual do detentor difere da sua prioridade original, indicando que ocorreu herança de prioridade, e procede à restauração. Após limpar os campos do detentor, a próxima tarefa de maior prioridade é removida da fila de espera e acordada. A atomicidade das operações é garantida através de spinlocks, e a flag do mutex é finalmente limpa para permitir novas aquisições.

3.2.2.3 Implementação de System Calls para Controlo de Mutex com PIP

```
1   SYSCALL_DEFINE0(moker_pip_mutex_lock)
2   {
3       return sys_moker_pip_mutex_lock();
4   }
5   int sys_moker_pip_mutex_lock (){
6       #ifdef CONFIG_MOKER_MUTEX_PIP
7       lock_pip_mutex();
8       #endif
9       return 0;
10  }
11
12  SYSCALL_DEFINE0(moker_pip_mutex_unlock)
13  {
14      return sys_moker_pip_mutex_unlock();
15  }
16  int sys_moker_pip_mutex_unlock (){
17      #ifdef CONFIG_MOKER_MUTEX_PIP
18      unlock_pip_mutex();
19      #endif
20      return 0;
21  }
```

Para permitir que a user space interaja com mutexes no kernel e que faça a gestão da sincronização de threads com suporte ao PIP, foram implementadas duas system calls: `moker_pip_mutex_lock` e `moker_pip_mutex_unlock`. Estas chamadas de sistema possibilitam o bloqueio e a libertação de um mutex, garantindo que a inversão de prioridade seja tratada adequadamente para evitar problemas como a inversão de prioridade descontrolada em sistemas de tempo real.

3.3 Priority Ceiling Protocol (PCP)

3.3.1 Estruturas de Dados

3.3.1.1 Fila de Espera baseada em Priority Ceiling Protocol

A estrutura `pcp_mutex_wq` representa a fila de espera específica para o mutex com *Priority Inheritance Protocol* no kernel Linux:

```
1  struct pcp_mutex_wq{
2      struct rb_root tasks;
3      atomic_t flag;
4      raw_spinlock_t lock;
5      struct task_struct *holder;
6      int holder_original_prio;
7      int ceilinged_priority;
8  };
```

- `struct rb_root tasks`: Utiliza uma Red–black tree para armazenar as tarefas bloqueadas ordenadas por prioridade. Esta estrutura de dados permite inserção, remoção e pesquisa em tempo O(log n), mantendo a ordenação necessária para a gestão de prioridades.
- `atomic_t flag`: Flag atómica que indica o estado do mutex (0 = desbloqueado, 1 = bloqueado). A atomicidade garante operações thread-safe sem necessidade de sincronização adicional.
- `raw_spinlock_t lock`: Spinlock para proteger as operações na fila de espera, garantindo atomicidade em ambientes multiprocessador durante modificações da estrutura.
- `struct task_struct *holder`: Apontador para a tarefa que detém atualmente o mutex. Este campo é fundamental para aplicar o teto de prioridade, permitindo identificar rapidamente qual tarefa está sob efeito do teto.
- `int holder_original_prio`: Prioridade original do detentor do mutex antes da elevação para o teto. Essencial para restaurar a prioridade original quando o mutex é libertado.
- `int ceilinged_priority`: Prioridade de teto associada ao mutex, definida estaticamente. Quando uma tarefa adquire o mutex, a sua prioridade é imediatamente elevada para este valor.

3.3.1.2 Nó de Tarefa em Espera no Priority Ceiling Protocol

A estrutura `pcp_mutex_node` representa as informações específicas de cada tarefa bloqueada na fila de espera do mutex PCP:

```
1  struct pcp_mutex_node{
2      struct rb_node node;
3      struct task_struct *task;
4      int priority;
5  };
```

- `struct rb_node node`: Nó da Red–black tree que permite à tarefa ser inserida na estrutura `pcp_mutex_wq`. Este campo estabelece a ligação entre a tarefa bloqueada e a fila de espera ordenada.

– `struct task_struct *task`: Apontador para a estrutura da tarefa bloqueada. Permite acesso direto às informações completas da tarefa para operações de desbloqueio e gestão de prioridade.
– `int priority`: Prioridade da tarefa no momento do bloqueio. Este campo é fundamental para manter a ordenação na Red–black tree e para determinar a ordem de desbloqueio das tarefas.

3.3.2 Implementação das Funções do Priority Inheritance Protocol

3.3.2.1 Aquisição do Mutex com Herança de Prioridade

```
1  void lock_pcp_mutex(void) {
2      struct task_struct *p = current;
3
4      raw_spin_lock(&pcp_wq.lock);
5
6      if(p->prio < pcp_wq.ceilinged_priority){
7          pcp_wq.ceilinged_priority = p->prio;
8      }
9
10     raw_spin_unlock(&pcp_wq.lock);
11
12     while (!atomic_add_unless(&pcp_wq.flag, 1, 1)) {
13         raw_spin_lock(&pcp_wq.lock);
14
15         set_current_state(TASK_INTERRUPTIBLE);
16
17         enqueue_pcp_mutex_task(p);
18
19         raw_spin_unlock(&pcp_wq.lock);
20
21         schedule();
22     }
23
24     // Adquiriu o lock: registar como holder
25     raw_spin_lock(&pcp_wq.lock);
26     pcp_wq.holder = p;
27     pcp_wq.holder_original_prio = p->prio;
28     if(pcp_wq.ceilinged_priority < pcp_wq.holder->prio){
29         change_task_prio_pcp(p, pcp_wq.ceilinged_priority);
30
31         raw_spin_unlock(&pcp_wq.lock);
32         schedule();
33     }else{
34         raw_spin_unlock(&pcp_wq.lock);
35     }
36
37     #ifdef CONFIG_MOKER_TRACING
38     moker_trace(MUTEX_LOCK, p);
39     #endif
40 }
```

A função `lock_pcp_mutex` implementa o mecanismo de aquisição de um mutex com herança de prioridade. Inicialmente, a função verifica se a prioridade da tarefa corrente é superior ao teto de prioridade atualmente registado no mutex. Caso seja, o teto é actualizado com a nova prioridade mais elevada. Em seguida, a função tenta, de forma atómica, adquirir o mutex através da flag `pcp_wq.flag`. Se a tentativa falhar (isto é, o mutex já estiver adquirido), a tarefa é colocada num estado interruptível e adicionada a uma fila de espera

ordenada por prioridade. Uma vez adquirido o mutex, a tarefa é registada como detentora (`holder`) do mesmo, e a sua prioridade original é armazenada. Um aspecto fundamental desta implementação é a verificação da prioridade efectiva da tarefa face ao teto registado. Se a prioridade actual da tarefa for inferior ao teto, a tarefa herda temporariamente a prioridade mais elevada por meio da função **change_task_prio_pcp**, prevenindo assim a inversão de prioridade — um problema clássico em sistemas com escalonamento por prioridades.

3.3.2.2 Libertação do Mutex e Restauração de Prioridade

```
1  void unlock_pcp_mutex(void) {
2      struct task_struct *p = current;
3      struct task_struct *t = NULL;
4
5      raw_spin_lock(&pcp_wq.lock);
6      if (pcp_wq.holder_original_prio != -1 && p->prio != pcp_wq.
           holder_original_prio) {
7          change_task_prio_pcp(p, pcp_wq.holder_original_prio);
8      }
9      pcp_wq.holder_original_prio = -1;
10     pcp_wq.holder = NULL;
11     raw_spin_unlock(&pcp_wq.lock);
12
13     t = dequeue_pcp_mutex_task();
14     if (t){
15         if (!wake_up_process(t)) {
16             printk(KERN_ERR "BUG: wake up process failed: %d\n", t->pid);
17         }
18     }
19
20     atomic_set(&pcp_wq.flag, 0);
21
22     #ifdef CONFIG_MOKER_TRACING
23     moker_trace(MUTEX_UNLOCK, p);
24     #endif
25 }
```

A função `unlock_pcp_mutex` é responsável por libertar o mutex previamente adquirido e restaurar a prioridade original da tarefa que o detinha, caso esta tenha sido elevada devido ao mecanismo de herança de prioridade. O processo inicia-se com a aquisição do spinlock associado ao mutex, garantindo acesso exclusivo à estrutura de dados partilhada. Se a prioridade actual da tarefa for diferente da prioridade originalmente registada no momento da aquisição do mutex, esta é restaurada. De seguida, a função limpa os campos que identificam o detentor do mutex e a sua prioridade original. Após a libertação do spinlock, é verificado se existem tarefas em espera na fila associada ao mutex. Caso exista alguma, a próxima tarefa é retirada da fila e ativada explicitamente com `wake_up_process`. Por fim, o mutex é marcado como livre, através da actualização da flag atómica `pcp_wq.flag`. Este procedimento assegura que a herança de prioridade é revertida de forma correcta, permitindo que o sistema retome o escalonamento normal baseado nas prioridades reais das tarefas.

3.3.2.3 Implementação de System Calls para Controlo de Mutex com PCP

```
1  SYSCALL_DEFINE0(moker_pcp_mutex_lock)
2  {
3      return sys_moker_pcp_mutex_lock();
```

```
4  }
5  int sys_moker_pcp_mutex_lock (){
6      #ifdef CONFIG_MOKER_MUTEX_PCP
7      lock_pcp_mutex();
8      #endif
9      return 0;
10 }
11
12 SYSCALL_DEFINE0(moker_pcp_mutex_unlock)
13 {
14     return sys_moker_pcp_mutex_unlock();
15 }
16 int sys_moker_pcp_mutex_unlock (){
17     #ifdef CONFIG_MOKER_MUTEX_PCP
18     unlock_pcp_mutex();
19     #endif
20     return 0;
21 }
```

Para permitir que a user space interaja com mutexes no kernel e que faça a gestão da sincronização de threads com suporte ao PCP, foram implementadas duas system calls: `moker_pcp_mutex_lock` e `moker_pcp_mutex_unlock`. Estas chamadas de sistema possibilitam o bloqueio e a libertação de um mutex, garantindo que a inversão de prioridade seja tratada adequadamente para evitar problemas como a inversão de prioridade descontrolada em sistemas de tempo real.

4 Testes e Validação

4.1 Rate Monotonic Scheduling

Os testes realizados encontram-se na pasta `tasks-rm`, onde o ficheiro `taskset.txt`, contendo o seguinte conjunto de tarefas:

Table 1. Conjunto de tarefas utilizado para Rate Monotonic Scheduling

Task ID	C (Computação)	T (Período)
1	5	10
2	7	15
3	8	30

Este conjunto representa três tarefas periódicas, caracterizadas pelos seus tempos de computação (C) e períodos (T). As tarefas são escalonadas segundo a política *Rate Monotonic*, onde tarefas com períodos mais curtos recebem prioridades mais elevadas.

A seguir apresenta-se o diagrama de Gantt resultante da execução com *Rate Monotonic Scheduling*:

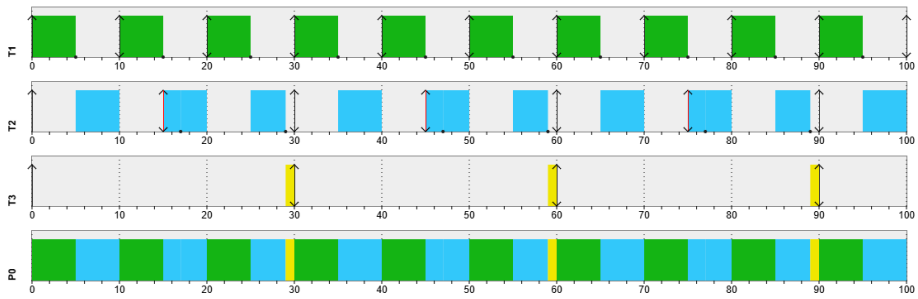

Fig. 1. Gantt - Rate Monotonic

A execução dos testes é feita com o seguinte comando:

```
cd tasks-rm
sudo sh run.sh
```

Após a execução, é gerado o ficheiro `trace_f.csv`, que contem os eventos relevantes para a análise do comportamento do escalonador. Um excerto representativo do conteúdo é apresentado abaixo:

```
time,event,policy,priority,pid,state,period,command
1076685450839,ENQ_RQ,8,58,7465,1,30000000000,task
1076685452115,ENQ_RQ,8,56,7463,1,10000000000,task
1076685452627,ENQ_RQ,8,56,7464,1,15000000000,task
1076690122604,SWT_TO,8,56,7463,0,10000000000,task
1082992128171,DEQ_RQ,8,56,7463,1,10000000000,task
1082992128884,SWT_AY,8,56,7463,1,10000000000,task
1082992128949,SWT_TO,8,56,7464,0,15000000000,task
1086320232456,ENQ_RQ,8,56,7463,1,10000000000,task
1086320348698,SWT_AY,8,56,7464,0,15000000000,task
1092712541168,SWT_TO,8,56,7463,0,10000000000,task
1094935717509,DEQ_RQ,8,56,7463,1,10000000000,task
1094935718487,SWT_AY,8,56,7463,1,10000000000,task
1094935718557,SWT_TO,8,56,7464,0,15000000000,task
1096316577709,ENQ_RQ,8,56,7463,1,10000000000,task
1096316669021,SWT_AY,8,56,7464,0,15000000000,task
1102714782663,SWT_TO,8,56,7463,0,10000000000,task
1104622006818,DEQ_RQ,8,56,7463,0,10000000000,task
1104622007443,ENQ_RQ,8,56,7463,0,10000000000,task
1104622017265,DEQ_RQ,8,56,7463,128,10000000000,task
1104622018011,SWT_AY,8,56,7463,128,10000000000,task
1104622018066,SWT_TO,8,56,7464,0,15000000000,task
1126218672274,DEQ_RQ,8,56,7464,0,15000000000,task
1126218672762,ENQ_RQ,8,56,7464,0,15000000000,task
1126218679367,DEQ_RQ,8,56,7464,128,15000000000,task
1126218680253,SWT_AY,8,56,7464,128,15000000000,task
1126218680308,SWT_TO,8,58,7465,0,30000000000,task
1135254784859,DEQ_RQ,8,58,7465,0,30000000000,task
1135254785471,ENQ_RQ,8,58,7465,0,30000000000,task
1135254790917,DEQ_RQ,8,58,7465,128,30000000000,task
1135254794458,SWT_AY,8,58,7465,128,30000000000,task
```

Observando as primeiras linhas do registo, verifica-se que as tarefas são enfileiradas com as seguintes prioridades:

– Tarefa 1 (período 10) e Tarefa 2 (período 15) recebem ambas a prioridade 56.

– Tarefa 3 (período 30) recebe a prioridade 58, ou seja, menor prioridade (valor numérico mais elevado).

Este comportamento pode dever-se à forma como o sistema atribui prioridades baseando-se em intervalos discretos e arredondamentos, o que pode resultar em colisões de prioridade entre tarefas com períodos distintos. No entanto, conforme se observa no conteúdo dos eventos, a ordem de execução respeita a política RM:

– A Tarefa 1 (período mais curto) tem prioridade mais elevada e é executada primeiro.
– A Tarefa 2 é executada de seguida.
– A Tarefa 3, com o maior período, é executada por último.

4.2 Priority Inheritance Protocol (PIP)

Os testes realizados encontram-se na pasta `tasks-pip`, onde o ficheiro `taskset.txt`, contendo o seguinte conjunto de tarefas:

Table 2. Conjunto de tarefas utilizado para o PIP

Task ID	**C Pré-Zona Crítica**	**C Zona critica**	**T (Período)**	**Offset**
1	1	1	6	0
2	2	0	10	0
3	0	4	12	0

Este conjunto representa três tarefas periódicas, caracterizadas pelos seus tempos de computação antes de aceder à zona crítica (C Pré-Zona Crítica), tempo de computação durante a zona crítica (C Zona Crítica) e períodos (T). As tarefas são escalonadas de acordo com a política *Rate Monotonic*, na qual tarefas com períodos mais curtos recebem prioridades mais elevadas.

A seguir apresenta-se o diagrama de Gantt resultante da execução com *Rate Monotonic Scheduling* com PIP:

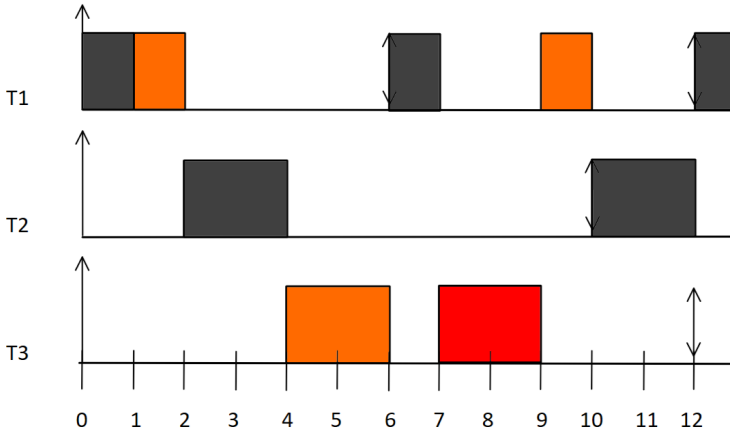

Fig. 2. Gantt - PIP

	Fora do mutex
	Zona crítica (mutex)
	Zona crítica com herança de prioridades

Tal como no caso anterior, a execução dos testes é realizada com os seguintes comandos:

```
cd tasks-pip
sudo sh run.sh
```

A execução gera o ficheiro `trace_f.csv`, que contém os eventos relevantes para a análise do comportamento do escalonador. Um excerto ilustrativo é apresentado abaixo:

```
...
19563812457243,MUT_LK,8,56,39871,0,12000000000,task
19564938342667,ENQ_RQ,8,55,39869,1,6000000000,task
19564938415596,SWT_AY,8,56,39871,0,12000000000,task
19564938415752,SWT_TO,8,55,39869,0,6000000000,task
19565468580503,SWT_AY,8,55,39869,0,6000000000,task
19565468814539,SWT_TO,8,55,39869,0,6000000000,task
19566113297350,MUT_UP,8,55,39871,0,12000000000,task
19566113380424,SWT_AY,8,55,39869,1,6000000000,task
19566113380476,SWT_TO,8,55,39871,0,12000000000,task
...
19572943999683,MUT_UP,8,56,39871,0,12000000000,task
19572944003950,MUT_UL,8,56,39871,0,12000000000,task
...
```

Neste protocolo, quando uma tarefa entra numa zona crítica protegida por um mutex, caso exista uma tarefa de menor prioridade em execução, esta vê a sua prioridade imediatamente elevada para o valor da tarefa de maior prioridade que entrou na zona crítica. Este comportamento é evidenciado por eventos do tipo `MUT_UP`. A elevação de prioridade evita bloqueios encadeados e assegura que nenhuma outra tarefa com prioridade inferior interfira na execução da zona crítica.

Assim que o mutex é libertado, a prioridade da tarefa afectada é restaurada ao seu valor original.

4.3 Priority Ceiling Protocol (PCP)

Os testes realizados encontram-se na pasta tasks-pcp, onde o ficheiro taskset.txt, contendo o seguinte conjunto de tarefas:

Table 3. Conjunto de tarefas utilizado para o PCP

Task ID	C Pre zona critica	C Zona critica	T (Período)	Offset
1	1	1	6	0
2	2	0	10	0
3	0	2	12	0

A seguir apresenta-se o diagrama de Gantt resultante da execucão com *Rate Monotonic Scheduling* com PCP:

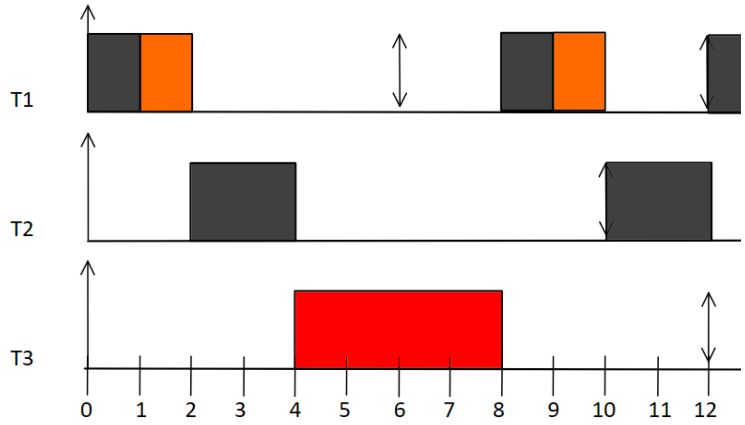

Fig. 3. Gantt - PCP

	Fora do mutex
	Zona crítica (mutex)
	Zona crítica com herança de prioridades

Tal como anteriormente, os testes são executados com:

```
1  cd tasks-pcp
2  sudo sh run.sh
```

O ficheiro `trace_f.csv` resultante permite observar o comportamento do sistema sob o *Priority Ceiling Protocol*. Um excerto representativo é apresentado de seguida:

```
1833558237590,ENQ_RQ,8,56,8974,1,10000000000,task
1833558238669,ENQ_RQ,8,55,8973,1,6000000000,task
1833558240078,ENQ_RQ,8,56,8975,1,12000000000,task
...
1843142838801,MUT_UP,8,55,8975,0,12000000000,task
1843142838857,MUT_LK,8,55,8975,0,12000000000,task
...
1859068451806,MUT_UL,8,56,8975,0,12000000000,task
```

Neste protocolo, quando uma tarefa entra na zona crítica protegida por um mutex, esta eleva imediatamente a sua prioridade ao valor do teto do recurso, independentemente de haver bloqueios. Este comportamento é visível nos eventos do tipo `MUT_UP`. Esta elevação de prioridade evita bloqueios encadeados e garante que nenhuma outra tarefa com prioridade inferior interfira na execução da zona crítica. Assim que o mutex é libertado, a prioridade da tarefa retorna ao valor original.

5 Conclusão

A implementação e análise do *Rate Monotonic Scheduling*, do *Priority Inheritance Protocol* e do *Priority Ceiling Protocol* no kernel do Linux revelaram insights valiosos sobre o comportamento desses mecanismos em sistemas de tempo real. O *Rate Monotonic Scheduling* demonstrou ser uma abordagem eficaz para o escalonamento de tarefas periódicas, respeitando as prioridades baseadas nos períodos e garantindo a escalonabilidade dentro dos limites teóricos de utilização, conforme observado nos testes realizados. Já o PIP e o PCP provaram ser soluções robustas para mitigar a inversão de prioridade, com o PIP ajustando dinamicamente as prioridades de tarefas bloqueadas e o PCP utilizando tetos de prioridade estática para prevenir bloqueios encadeados e reduzir tempos de espera.

Os resultados obtidos nos cenários de teste, documentados por meio de arquivos de rastreamento, confirmaram que essas técnicas conseguem atender às restrições temporais em condições controladas, embora limitações como colisões de prioridade no *Rate Monotonic Scheduling* e a complexidade de gestão no PIP e PCP tenham sido identificadas. Este trabalho não apenas reforça a relevância desses mecanismos para sistemas críticos, mas também destaca a importância de sua adaptação e otimização em ambientes como o kernel do Linux, que oferece um campo fértil para experimentação e desenvolvimento.