Escalonamento no Próximo Nível: Implementando uma Nova Política de Escalonamento no Kernel Linux

Eduardo H. A. Izidorio[1], Gabriel P. M. Costa[2]

[1]Ciência da Computação - Universidade Federal de Roraima (UFRR)

CEP - 69310-000 - Boa Vista - RR - Brazil

[2]Departamento de Ciência da Computação

eduardo57izidorio@gmail.com, gabrielpeixoto371@gmail.com

Resumo.

Este relatório descreve o projeto de implementação de uma nova política de escalonamento no kernel do Linux, chamada de SCHED_BACKGROUND. O escalonador de processos é responsável por selecionar qual processo será executado quando a CPU estiver disponível. O objetivo dessa nova política é suportar processos que só precisam ser executados quando o sistema não tem mais nada a fazer, assim apresentar tutoriais e exemplos de código, comparando com outras políticas existentes e avaliar seu desempenho por meio de testes e análise de tempo de execução.

Abstract.

This report describes the project of implementing a new scheduling policy in the Linux kernel, called SCHED_BACKGROUND. The process scheduler is responsible for selecting which process will be executed when the CPU is available. The aim of this new policy is to support processes that only need to be executed when the system has nothing else to do, thus presenting tutorials and code examples, comparing them with other existing policies, and evaluating their performance through testing and runtime analysis.

1. O que é o Escalonador do Linux?

Antes de explicar o funcionamento do escalonador, é necessário conhecer sua definição. Em sistemas computacionais atuais, vários threads aguardam execução. Assim, o escalonador é um componente que ajuda a decidir qual processo será executado pela CPU em um determinado tempo. A maioria dos algoritmos de escalonamento é baseada em prioridades, sendo assim o escalonador do Linux desempenha um papel crucial na distribuição de recursos de CPU e E/S entre os processos em execução, visando oferecer um ambiente de execução equitativo e eficiente.

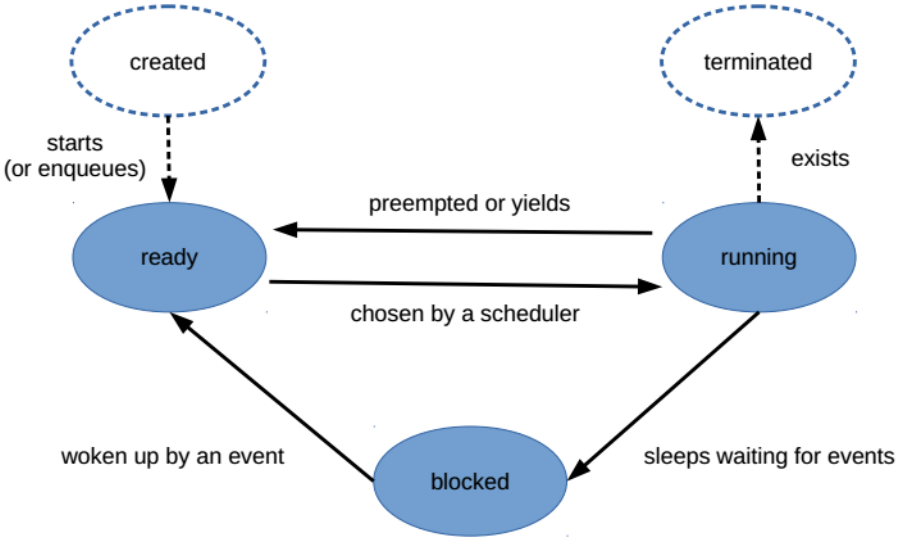
2. Como funciona o escalonador no Linux

O kernel do Linux apresenta um escalonamento preemptivo, isso significa que em determinada situação um thread pode ser parada para execução de outra antes de ser

concluída. O kernel do Linux apresenta um escalonamento preemptivo, isso significa que em determinada situação um thread pode ser parada para execução de outra antes de ser concluída. Como dito anteriormente, o escalonador do Linux trabalha com base em prioridades, desta maneira os threads ou processos que possuírem uma maior prioridade são executados primeiro.

Com isso o encadeamento antecipado retoma o thread suspensa após a outra finalizar ou simplesmente ser bloqueada. Quando ocorre essa intercalação de threads, o contexto do thread antigo é salvo em algum lugar e o contexto do novo thread é carregado. Para ocorrer a escolha o escalonador precisa saber quais são as threads que estão disponíveis para serem executadas naquele momento, por isso cada thread tem um estado atual.

3. Estados das Threads



Os Threads possuem 3 estados, sendo eles Pronto, Executando e Bloqueado.

Pronto: O encadeamento está pronto para ser executado, mas não tem permissão, ele só terá a permissão após os processadores desocuparem os que estão executando e se o escalonador o escolher;

Executando: O thread está sendo executada em um processador;

Bloqueado: O thread está bloqueado esperando apenas algum evento externo como um I/O Bound ou um sinal. A thread bloqueada não pode ser executada.

4. Classificações de uma Thread

Assim como os threads possuem seus estados, elas são classificadas em dois tipos principais, sendo I/O-Bound e CPU-Bound.

I/O-Bound: São encadeamentos que fazem o uso intensivo de chegadas de entradas (por exemplo, cliques no mouse ou até copiar um arquivo para um pendrive) ou a conclusão de saídas (por exemplo, gravação em discos). Esses processos são conhecidos assim por fazer pouco uso da CPU e por isso eles são processados mais rapidamente e o escalonador considera que ele tenha uma maior prioridade em relação ao CPU-Bound;

CPU-Bound: São aqueles onde o tempo de processamento depende mais do processador do que as entradas e saídas, fazendo assim com que atrapalhe o tempo total de processamento (como por exemplo, a compilação de programa). Os threads que têm vinculação com o CPU são pouco escolhidas, mas em compensação quando são escolhidas elas mantêm a CPU por mais tempo.

4.1. Tempo Real e Não Real

Outro critério para categorizar os encadeamentos é saber se eles são real-time ou não.

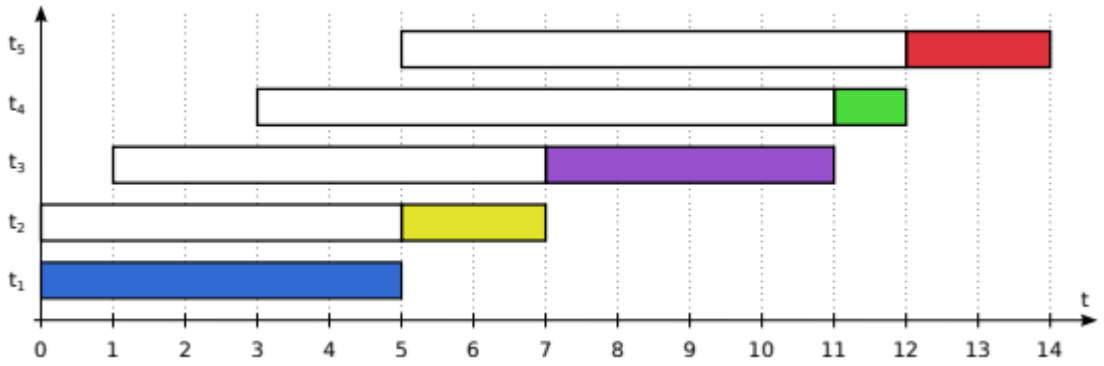
Tempo Real: São aquelas que geralmente trabalham com uma restrição de tempo, um prazo, também conhecido como deadline. A exatidão operacional de um encadeamento não depende apenas do resultado do cálculo, mas também dos resultados serem entregues antes do prazo. Por esse motivo o escalonador deixa com a mais alta prioridade os threads real-time.

Tempo Não-Real: Não está associado a nenhum prazo. Elas podem ser threads com interação humana ou threads do tipo batch, os threads batch são aquelas que processam uma grande quantidade de dados sem intervenção manual. Para os batch threads, um tempo de resposta rápido não é crítico e, portanto, eles podem ser programados para serem executados conforme os recursos permitirem.

5. Políticas de Escalonamento do Linux

5.1. First-Come, First-Served (FCFS)

Conhecido também como First-In, First-Out (FIFO), é um algoritmo de escalonamento de processos, onde os processos são executados de acordo com a ordem em que chegaram à fila de prontos. Uma de suas vantagens é sua simplicidade e uma desvantagem é que o FCFS não leva em consideração o tempo de execução, assim como também a prioridade de cada processo, permanecendo em execução até ser concluído ou interrompido.

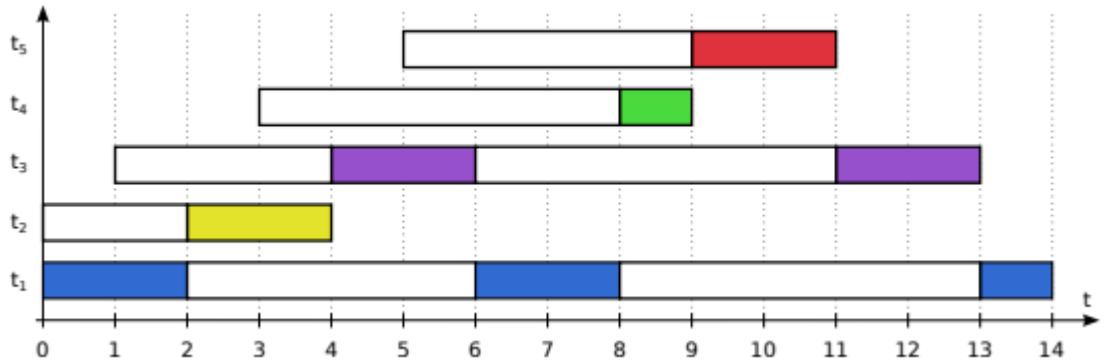


No gráfico acima apresenta o escalonamento baseado no tempo, onde cada bloco colorido é um processo e o bloco branco seria o tempo de espera na fila.

5.2. Round-Robin (RR)

É um algoritmo que é projetado para lidar com a divisão de tempo de CPU entre os processos e garantir uma resposta rápida para as requisições interativas. Os processos nesse escalonamento são colocados em uma fila circular ou buffer, onde cada processo recebe uma

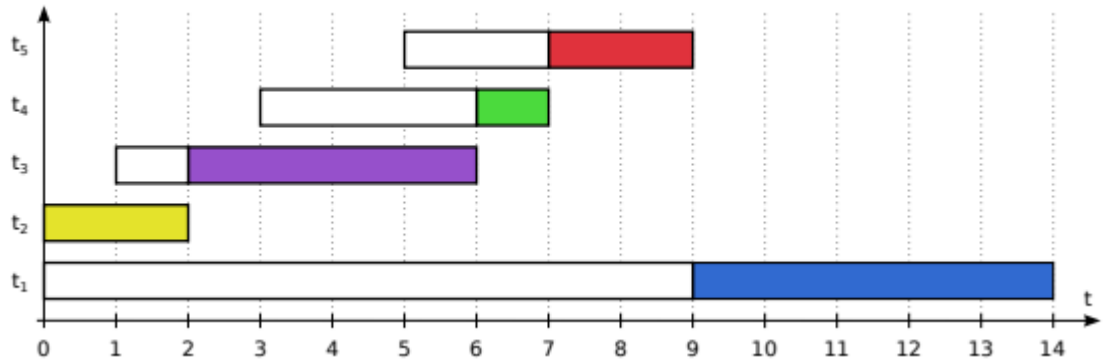
fatia de tempo fixa, chamada de *quantum* ou *Time-Slice*. O RR garante uma justa distribuição do tempo de CPU, assim garantindo também que nenhum processo monopolize a CPU por muito tempo, evitando a inanição(Starvation) de outros processos.



No gráfico acima, podemos observar a execução de processos, onde a sequência não vai em uma ordem óbvia como t1 → t2 → t3 → t4 → t5 → t1 → t2 →… , mas uma sequência bem mais complexa: t1 → t2 → t3 → t1 → t4 → t5 →… . Isso ocorre por causa da ordem na fila de prontas e num *quantum* de tempo t = 2s. Assim, os processos a cada 2 segundos voltam para a fila de prontas e outro processo é executado.

5.3. Shortest Job First (SJF)

É um algoritmo de escalonamento de processos, que consiste em atribuir o processador à menor (mais curta) tarefa da fila de tarefas prontas. Esse algoritmo (e sua versão preemptiva, SRTF) proporciona os menores tempos médios de espera das tarefas. A maior dificuldade no uso do algoritmo SJF consiste em estimar a prioridade da duração de cada tarefa, antes de sua execução. Com exceção de algumas tarefas em lote ou de tempo real, essa estimativa é inviável.



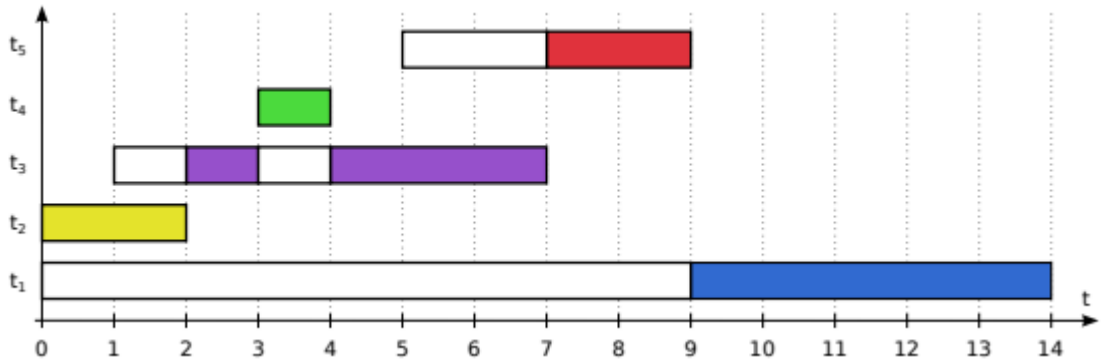
No gráfico acima, podemos ver os processos sendo executados nas partes coloridas e nas partes em branco é onde os processos estão em espera até serem chamados para serem executados.

5.4. Shortest Remaining Time First (SRTF)

É um algoritmo que seleciona o processo com o menor tempo restante de execução para ser executado em primeiro lugar. O SRTF é a variação preemptiva do escalonamento

SJF, que é o cooperativo, mas a diferença entre o SRTF para o SJF, é que o SRTF leva em consideração tempo restante de execução dos processos, o que significa que um processo com um tempo de execução restante menor pode interromper um processo em execução que tenha um tempo de execução maior.

Uma vantagem do SRTF é o fato de minimizar o tempo médio de espera e melhorar o tempo de resposta, já que os processos mais curtos são executados em primeiro lugar. Mas uma desvantagem é que pode sofrer de um problema conhecido como inanição (Starvation) para processos longos, já que os processos mais curtos têm prioridade.



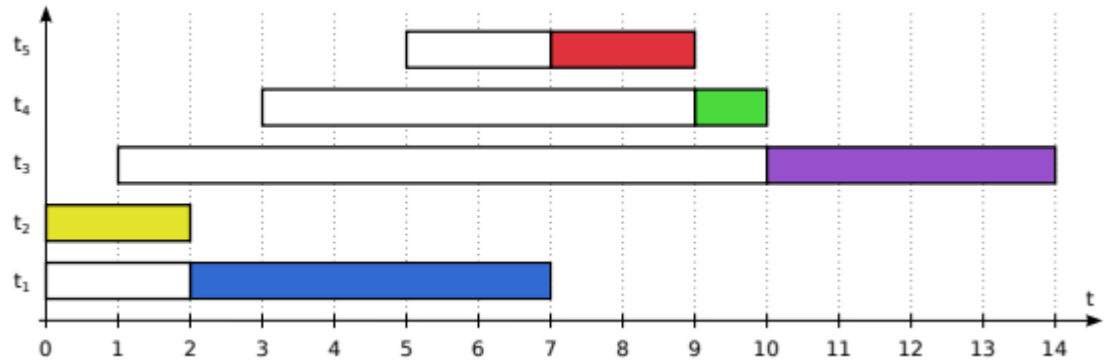
No gráfico acima, a prioridade não é tão importante para um processo ser executado e sim o menor tempo para ser executado. Nota-se o comportamento de t3, que com a chegada de t4 ele para seu processamento e o t4 pega o seu lugar para ser processado, isso ocorre por o t4 ter o menor tempo de processamento em relação ao t3.

5.5. Escalonamento por Prioridades Fixas (PRIOc, PRIOp)

São algoritmos de escalonamento de processos, no qual cada processo é atribuído a uma prioridade fixa, cada processo é designado com uma prioridade estática, que pode ser definida pelo Sistema Operacional ou pelo Usuário.

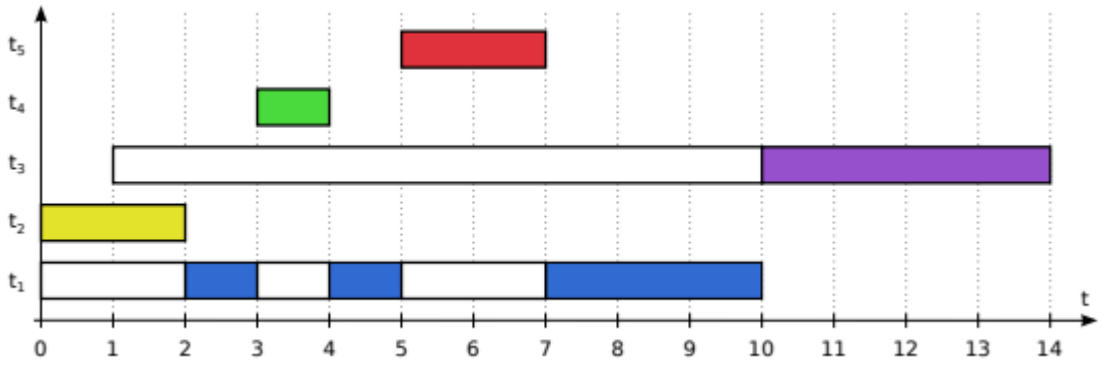
Esse tipo de escalonamento é eficaz para sistemas com requisitos de prioridade claros, onde determinados processos devem receber atenção prioritária.

PRIOc: No PRIOc, além da prioridade fixa, é aplicado uma técnica chamada envelhecimento (Aging) dos processos, que aumenta gradualmente a prioridade de um processo que está esperando na fila de prontos há um determinado tempo, isso é feito para evitar a inanição dos processos de prioridade baixa.



No gráfico acima os blocos coloridos significam os processos em processamento e os brancos é o tempo de espera, por estar nesse tipo de escalonamento cooperativo a prioridade é de total importância, nota-se que o t1 e t2 iniciaram ao mesmo tempo, mas por t2 ter maior prioridade ele acabou sendo processado primeiro e o t1 ficou em tempo de espera. Esse tipo de comportamento acontece até o final de todos os processos.

PRIOp: Por outro lado, o PRIOp não possui o envelhecimento. Os processos mantêm suas prioridades fixas durante todo o tempo de execução. Dessa forma, a prioridade atribuída a um processo no início da execução é mantida até a sua conclusão ou até ser bloqueada.



Já nesse tipo de escalonamento preemptivo, nota-se que a prioridade também é de suma importância, tão importante que pode ocorrer de um processo com maior prioridade interromper o processo atual e tomar o seu lugar para ser processado. Esse tipo de comportamento ocorre com o t1, t4 e t5 de forma que no meio do processamento do t1, o t4 toma seu lugar para ser processado e logo após o t5 faz a mesma coisa e apenas depois, que o t1 termina de ser processado.

6. Tutorial

Antes mesmo de começar o tutorial o Linux 2.6.24 é a versão padrão do kernel no Ubuntu 8.04, mas em versões mais recentes, a encapsulação é aumentada, o que aumenta a complexidade do código do agendador. Nas versões mais recentes, não há mais um arquivo "sched.c", mas sim um diretório "sched" que consiste em várias partes diferentes do antigo agendador. Precisamos dizer que é complexo demais fazer o escalonador nas versões mais atuais do kernel do linux então agora podemos seguir com o tutorial na versão linux 2.6.24:

1. Para começar, precisamos descobrir qual versão do kernel estamos executando no momento:

$ uname -r

2. Obtenha o código do kernel Linux no site Kernel.org e usando o comando wget:

$ cd /tmp

$wget
https://git.kernel.org/pub/scm/linux/kernel/git/stable/linux.git/snapshot/linux-2.6.24.y.tar.gz

3. Extraia o arquivo .tar:

$ tar -xzvf 2.6.24.y.tar.gz -C /usr/src
$ cd /usr/src/2.6.24.y

4. instalar a biblioteca curses e algumas outras ferramentas para nos ajudar a compilar

$ sudo apt-get install kernel-package libncurses5-dev fakeroot

5. Aplicar as mudanças de implementação nas pastas sched.c,sched.h e chrtB.c:
/include/linux/sched.h
/kernel/sched.c
/usr/bin/chrt

6. Fazemos uma cópia da configuração do kernel para usar durante a compilação:

$ cp boot/config-2.6.24-generic /usr/src/linux/.config

7. Primeiro vamos fazer um make clean, só para ter certeza que está tudo pronto para a compilação:

$ make-kpkg clean

8. Em seguida, vamos realmente compilar o kernel. Isso levará um "TEMPO LONGO" talvez 40-50 minutos.

$ fakeroot make-kpkg --initrd --append-to-version=-custom kernel_image kernel_headers
Este processo criará dois arquivos .deb em /usr/src que contém o kernel

9.Observe que, ao executar os próximos comandos, isso definirá o novo kernel como o novo kernel padrão. Isso pode dar problema! Se sua máquina não inicializar, você pode pressionar Esc no menu de carregamento do GRUB e selecionar seu kernel antigo. Você pode então desabilitar o kernel em /boot/grub/menu.lst ou tentar compilar novamente.

$ dpkg -i linux-image-2.6.24-custom_2.6.24-custom-10.00.Custom_i386.deb
$ dpkg -i linux-headers-2.6.24-custom_2.6.24-custom-10.00.Custom_i386.deb

10. Agora reinicie sua máquina. Se tudo funcionar, você deve estar executando seu novo kernel personalizado. Você pode verificar isso usando uname. Observe que o número exato será diferente em sua máquina.:

$ uname -r 2.6.24-custom

11. Para alterar a política de um processo para SCHED_BACKGROUND política em seu tempo de execução, comando usado: :

$ chrt -g -p 0 < pid>

7. Exemplo de Implementação

Essa primeira parte estaria sendo editado no arquivo “sched.h” que se encontra na pasta “/include/linux/sched.h”.

```
#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
/* SCHED_BACKGROUND head */
#define SCHED_BACKGROUND        6
```

Define **SCHED_NORMAL** como 0, que representa a política de escalonamento normal ou padrão.

Define **SCHED_FIFO** como 1, que representa a política de escalonamento FIFO (First-In, First-Out), onde as tarefas são executadas na ordem em que foram inseridas na fila.

Define **SCHED_RR** como 2, que representa a política de escalonamento Round Robin, onde as tarefas são executadas por um determinado período de tempo (quanta) antes de serem trocadas com outras tarefas de mesma prioridade.

Define **SCHED_BATCH** como 3, que representa a política de escalonamento em lote, onde as tarefas são executadas em segundo plano, priorizando a eficiência energética em vez de uma resposta imediata.

Define **SCHED_IDLE** como 5, que representa a política de escalonamento em modo ocioso, onde as tarefas de baixa prioridade são executadas apenas quando não há outras tarefas a serem executadas.

Definindo **SCHED_BACKGROUND** como 6. Esse valor pode ser usado para representar uma política de escalonamento específica que eu estou desejando definir, como uma política personalizada para tarefas em segundo plano, por exemplo.

Nessa parte do código em diante estaria sendo editado no arquivo “sched.c” e que se encontra no diretório “/kernel/sched.c”.

```c
asmlinkage long sys_sched_get_priority_max(int policy)
{
        int ret = -EINVAL;

        /* SCHED_BACKGROUND head */
        switch (policy) {
        case SCHED_FIFO:
        case SCHED_RR:
                ret = MAX_USER_RT_PRIO-1;
                break;
        case SCHED_NORMAL:
        case SCHED_BATCH:
        case SCHED_IDLE:
                ret = 0;
                break;
        case SCHED_BACKGROUND:
          ret = 0;
          break;
        }
        return ret;
}
```

Na linha **asmlinkage long sys_sched_get_priority_max(int policy)** define a assinatura da função sys_sched_get_priority_max, que recebe um argumento policy do tipo inteiro e retorna um valor longo.

Está declarando uma variável **ret** e a inicializa com o valor de erro **-EINVAL** (um valor de erro comum no Linux que indica que um argumento é inválido). (**int ret = -EINVAL**)

No bloco de código onde indica um **switch statement** que verifica o valor do argumento policy. Dependendo do valor, ele define o valor correto para a variável **ret**.

- Se policy for igual a **SCHED_FIFO** ou **SCHED_RR**, ret é definido como o valor **MAX_USER_RT_PRIO-1**.
- Se policy for igual a **SCHED_NORMAL**, **SCHED_BATCH** ou **SCHED_IDLE**, ret é definido como 0.
- Se policy for igual a **SCHED_BACKGROUND**, ret é definido como 0 (conforme sua definição personalizada).

No final do código acima (**return ret;**) Retorna o valor de ret, que foi definido no switch statement.

Ou seja, essa função recebe um argumento que representa uma política de agendamento e retorna o valor máximo de prioridade para essa política específica. A política de escalabilidade **SCHED_BACKGROUND** é incluída e o valor de retorno dessa política é definido como 0 pela implementação.

Ainda no arquivo “sched.c”:

```
__setscheduler(struct rq *rq, struct task_struct *p, int policy, int prio)
{
        BUG_ON(p->se.on_rq);

        p->policy = policy;

        /* SCHED_BACKGROUND head */
        switch (p->policy) {
        case SCHED_NORMAL:
        case SCHED_BATCH:
        case SCHED_IDLE:
                p->sched_class = &fair_sched_class;
                break;
        case SCHED_FIFO:
        case SCHED_RR:
                p->sched_class = &rt_sched_class;
                break;
        case SCHED_BACKGROUND:
          p->sched_class = &fair_sched_class;
          break;
        }


        p->rt_priority = prio;
        p->normal_prio = normal_prio(p);
        /* we are holding p->pi_lock already */
        p->prio = rt_mutex_getprio(p);
        set_load_weight(p);
}
```

Nessa função (__setscheduler) é responsável por configurar a política de escalonamento e a prioridade de uma tarefa específica.

Na linha “**BUG_ON(p->se.on_rq);**” contém uma macro **BUG_ON** que verifica uma condição de erro. Nesse caso, ela verifica se a tarefa p já está na fila de execução (on_rq). Se essa condição for verdadeira, um erro grave (BUG) será acionado.

Em “**p->policy = policy;**” define a política de escalonamento da tarefa **p** como o valor do argumento policy recebido pela função.

No bloco onde se encontra o **switch statement** é onde se verifica o valor da política de escalonamento **p->policy**. Dependendo do valor, ele define a classe de escalonamento correspondente para a tarefa **p**. Se a política for **SCHED_NORMAL**, **SCHED_BATCH** ou **SCHED_IDLE**, a classe de escalonamento é definida como fair_sched_class. Se a política for **SCHED_FIFO** ou **SCHED_RR**, a classe de escalonamento é definida como **rt_sched_class**. E, se a política for **SCHED_BACKGROUND**, a classe de escalonamento também é definida como **fair_sched_class** (conforme foi personalizada).

Nas últimas linhas(“**p->rt_priority = prio;**”, “**p->normal_prio = normal_prio(p);**”, “**p->prio = rt_mutex_getprio(p);**”, “**set_load_weight(p);”**) definem a prioridade da tarefa p com base no valor do argumento **prio**. A prioridade em tempo real (**rt_priority**) é definida como **prio**. A prioridade normal (**normal_prio**) é calculada com base nas características da tarefa p. A prioridade (prio) é obtida por meio de um mecanismo de bloqueio (**rt_mutex_getprio**). E, por fim, o peso de carga (**load_weight**) da tarefa é configurado.

Resumindo, este trecho de código define a política de agendamento, prioridade e outras propriedades relacionadas para uma determinada tarefa com base nos valores fornecidos como argumentos. A política de agendamento **SCHED_BACKGROUND** é adicionada e a classe de agendamento dessa política é definida como **fair_sched_class** como outras políticas relacionadas a tarefas em segundo plano.

8. RTAI e PREEMPT_RT

O **RTAI** (Real-Time Application Interface) é um framework de código aberto desenvolvido para sistemas operacionais Linux. Ele fornece um ambiente de tempo real para a execução de aplicativos em tempo real em sistemas baseados em x86.Como o próprio Linux, este software é um esforço da comunidade. Tendo suporte até a versão do kernel 4.19.

O **PREEMPT_RT**, também conhecido como Preempt-RT ou Kernel Preempt-RT, é um patch (ou conjunto de patches) aplicado ao kernel Linux para fornecer suporte a escalonamento preemptivo em tempo real. O patch **PREEMPT_RT** tem como objetivo transformar o kernel Linux em um sistema operacional de tempo real, oferecendo maior determinismo e capacidade de resposta para aplicações críticas em tempo real. Ele permite que tarefas em tempo real tenham prioridade sobre tarefas não críticas, garantindo que interrupções e eventos urgentes sejam tratados com baixa latência e alta precisão. No site tem várias versões porém alguns patchs de arquivos estão com problema de downloads sendo que quando é feito o download tem muita perda de pacote, sendo assim demorando muito para ter o arquivo.

References

Linux kernel scheduler | Jinkyu Koo. Disponível em: <https://helix979.github.io/jkoo/post/os-scheduler/>. Acesso em: 26 Jun de 2023

How To Build Linux Kernel {Step-By-Step} | phoenixNAP KB. Disponível em: <https://phoenixnap.com/kb/build-linux-kernel>. Acesso em: 26 Jun de 2023

How to: Compile Linux kernel 2.6. Disponível em: <https://embeddedlinuz.wordpress.com/2011/06/10/how-to-compile-linux-kernel-2-6/>. Acesso em: 26 jun. 2023.

Index of /pub/linux/kernel/. Disponível em: <https://mirrors.edge.kernel.org/pub/linux/kernel/>. Acesso em: 26 jun. 2023.

RAKAVI 1408. SCHED BACKGROUND. Disponível em: <https://github.com/rkavi1408/SCHED_BACKGROUND.git>. Acesso em: 26 jun. 2023.

CARLOS, A.MAZIERO. Sistemas Operacionais: Conceitos e Mecanismos. [s.l: s.n.]. Disponível em: <https://wiki.inf.ufpr.br/maziero/lib/exe/fetch.php?media=socm:socm-livro.pdf>.

RTAI. Disponível em: <https://www.rtai.org/>. Acesso em: 28 jun. 2023.

realtime:start [Wiki]. Disponível em: <https://wiki.linuxfoundation.org/realtime/start>. Acesso em: 28 jun. 2023.