UNIVERSIDADE FEDERAL DE SANTA CATARINA
CENTRO TECNOLÓGICO DE JOINVILLE
CURSO DE ENGENHARIA MECATRÔNICA

LUCAS MATHEUS DOS SANTOS

MODELAGEM E IMPLEMENTAÇÃO DE PROTOCOLOS DE ACESSO A RECURSOS COMPARTILHADOS EM SISTEMAS OPERACIONAIS DE TEMPO REAL

Joinville
2022

LUCAS MATHEUS DOS SANTOS

MODELAGEM E IMPLEMENTAÇÃO DE PROTOCOLOS DE ACESSO A RECURSOS COMPARTILHADOS EM SISTEMAS OPERACIONAIS DE TEMPO REAL

Trabalho de Conclusão de Curso apresentado como requisito parcial para obtenção do título de bacharel em Engenharia Mecatrônica no curso de Engenharia Mecatrônica, da Universidade Federal de Santa Catarina, Centro Tecnológico de Joinville.

Orientador: Dr. Giovani Gracioli

Joinville
2022

AGRADECIMENTOS

Acima de tudo, gostaria de agradecer à minha família, em especial a meus pais, Robson e Adriana, que acreditaram em mim e possibilitaram que tudo isto fosse possível, me apoiando incondicionalmente durante toda minha jornada na universidade, desde minha inscrição no vestibular até o presente momento, independentemente da distância. Gostaria de agradecer também a meus professores e membros do laboratório LISHA, que contribuíram de certa forma em meu crescimento profissional durante minha estadia na universidade, em especial, gostaria de agradecer ao meu professor orientador Giovani Gracioli, não apenas pela imensa participação e disponibilidade durante a elaboração deste trabalho, mas também pelas oportunidades a quais me foram oferecidas durante minha graduação, tenha certeza que suas atitudes foram muito importantes em minha trajetória e se refletem hoje em minha carreira. Também gostaria de agradecer imensamente à toda a instituição Los Santos e agregados, da qual me orgulho muito de fazer parte. Amigos que certamente tornaram a faculdade alegre, e principalmente, suportável nos momentos difíceis. E que espero levar comigo para o resto de minha vida. Deixo aqui também meus agradecimentos a meus poucos, porém muito verdadeiros amigos "joinvilenses", que certamente nunca lerão estas palavras, mas tiveram, e continuam tendo, participação ativa em minha vida.

RESUMO

Uma das principais características de sistemas de tempo real é a previsibilidade e confiabilidade, sendo que os sistemas frequentemente utilizam tarefas que devem acessar recursos compartilhados, considerados fonte de imprevisibilidade no tempo de execução de uma tarefa. Como o uso de recursos compartilhados, muitas vezes, não pode ser evitado, os sistemas operacionais de tempo real devem prover formas de evitar a imprevisibilidade temporal, aumentando a confiabilidade do sistema. Uma das formas de restringir o tempo de execução de uma tarefa que utiliza esses recursos, é utilizar protocolos de acesso a recursos compartilhados, porém, existem muitos tipos de protocolos, completamente diferentes entre si e que normalmente, são implementados especificamente para serem utilizados naquele sistema. Tais protocolos de acesso a recursos compartilhados são o objeto de estudo deste trabalho, que tem como intuito projetar uma solução que seja facilmente modificável, estendível e replicável em qualquer sistema que utilize uma linguagem orientada a objeto. Além de modelar a solução, os protocolos foram implementados em um sistema operacional de tempo real e, posteriormente, tiveram suas implementações comparadas e avaliadas para garantir o funcionamento adequado e leveza do sistema modelado. O modelo foi analisado por meio de parâmetros como tempo de execução, rastro de memória e escalonabilidade de conjuntos de tarefas, que também foram utilizados como comparação entre diferentes protocolos e suas categorias.

Palavras-chave: Semáforos. Ceiling Protocols. Análise de escalonabilidade. Multicore.

LISTA DE FIGURAS

LISTA DE QUADROS

LISTA DE TABELAS

LISTA DE SÍMBOLOS

τ	Tarefa
p	Prioridade de uma tarefa
d, D	Deadline de uma tarefa
t	Instante de tempo
C	Tempo de computação / execução de tarefa
T	Período de uma tarefa
U	Utilização de uma tarefa
U_s	Utilização do sistema
N	Número de tarefas no sistema
R	Recurso compartilhado ou seção crítica
$\Pi(R)$	Prioridade teto de um recurso
Π_k	Prioridade teto do sistema
m	Número de processadores em um sistema
p^{acc}	Probabilidade de uma tarefa acessar um recurso compartilhado em sua execução
n_{res}	Número de recursos compartilhados

SUMÁRIO

1 INTRODUÇÃO

Sistemas operacionais de tempo real são particularmente diferentes de sistemas operacionais tradicionais, pois, toda tarefa deve atender restrições temporais, independente do estado atual do sistema como uso do processador e memória (HAMBARDE; VARMA; JHA, 2014). Por esse motivo, existe um campo de pesquisa que visa desenvolver e avaliar métodos que aumentem a confiabilidade de sistemas de tempo real, tentando minimizar parâmetros e métodos que insiram imprevisibilidade temporal no sistema.

A exclusão mútua de recursos é um método utilizado em sistemas operacionais para garantir que recursos críticos do sistema não sejam acessados por duas tarefas diferentes simultaneamente, garantindo, assim, a integridade dos dados desse recurso, que é comumente chamado de recurso compartilhado ou seção crítica (YANG; WIEDER; BRANDENBURG, 2015). Para garantir a exclusão mútua, normalmente são implementados métodos de suspensão que bloqueiam tarefas que tentem acessar recursos compartilhados já acessados naquele instante de tempo por outra tarefa (YANG et al., 2015).

Métodos de suspensão são responsáveis por garantir a integridade das seções críticas, porém, são uma fonte de imprevisibilidade temporal no sistema, pois, uma tarefa que tente acessar um recurso compartilhado utilizado por outra tarefa naquele instante, ficará bloqueada por uma quantidade indeterminada de tempo. Sendo assim, sistemas operacionais de tempo real devem implementar protocolos de acesso a recursos compartilhados (BUTTAZZO, 2011) que trabalhem de forma a limitar o tempo máximo que uma tarefa pode ser bloqueada, permitindo assim, que o máximo tempo de bloqueio de uma tarefa seja mensurado, e com isso, seu *worst case execution time* (WCET), o tempo de execução de uma tarefa no pior caso (BUTTAZZO, 2011).

Existem muitos protocolos de acesso a recursos compartilhados, baseados em diferentes premissas, como suspensão (RAJKUMAR, 1990) ou *spin-locks* (WIEDER; BRANDENBURG, 2013), sendo um tópico amplamente estudado na computação. Um programador que venha a utilizar um *real-time operating system* (RTOS) (HAMBARDE et al., 2014), normalmente escolhe um protocolo dentre os disponíveis no sistema, ou revisa a literatura em busca dos mais populares, escolhendo um para ser implementado na linguagem escolhida no sistema em questão. Dessa forma, as implementações dos protocolos são normalmente feitas para um único sistema, em que um RTOS implementa dois ou três protocolos, como, por exemplo, a extensão de tempo real para Linux, o Litmus-RT (CALANDRINO et al., 2006).

Neste trabalho, propõe-se um modelo genérico que comporte diferentes tipos

de protocolos de acesso a recursos compartilhados, de forma a resolver problemas como implementações específicas que só funcionam no RTOS para a qual foram feitas. Pretendeu-se que qualquer RTOS que seja escrito utilizando uma linguagem orientada a objeto, possa implementar o modelo proposto, escolhendo o protocolo que atenda os requisitos a serem utilizados no sistema. Porém, reforça-se que sistemas escritos em linguagens que não suportam programação orientada à objeto (OOP), como a linguagem C, também podem aproveitar o modelo de forma a estender e estruturar a arquitetura do sistema com base no modelo proposto. No entanto, a tradução do diagrama de classes proposto no modelo será menos direta para estas linguagens, destacando a vantagem de utilizar linguagens que suportam OOP. O modelo também deve ser facilmente extensível, para que eventualmente sejam adicionados novos protocolos sem maiores trabalhos ou mudanças na hierarquia de classes, solucionando assim os problemas de implementações específicas realizadas unicamente em sistemas e soluções pontuais.

O modelo proposto foi implementado no *Embedded Parallel Operating Programing* (EPOS, 2022), um RTOS escrito em C++, perfeito para a implementação, pois suporta programação orientada a objeto, além de ser uma linguagem com baixo sobrecusto (*overhead*). Alguns protocolos foram escolhidos para serem adicionados ao modelo, de forma que o trabalho abranja protocolos para sistemas *single* e *multicore* (BUTTAZZO, 2011) usando diferentes algoritmos de escalonamento (LIU, 2000). A implementação do modelo proposto foi realizada considerando características de boas práticas de programação como reusabilidade de código e proteção de acesso a dados entre diferentes classes, de forma a atingir alguns dos objetivos subjetivos em relação ao modelo. As classes implementadas tiveram seus rastros de memória (*memory footprint*) e sobrecustos (HAMBARDE et al., 2014) mensurados para validar a eficiência e leveza da implementação do modelo. Com o mesmo propósito, os dados de sobrecusto foram utilizados para avaliar o impacto da implementação na escalonabilidade dos algoritmos de escalonamento. Esses valores foram utilizados como parâmetros de comparação entre os diferentes protocolos e tipos de protocolos.

1.1 OBJETIVO

Para resolver a problemática da falta de padronização na estrutura de modelo e implementação de protocolos de acesso a recursos para diferentes sistemas operacionais de tempo real, propõe-se neste trabalho os seguintes objetivos.

1.1.1 Objetivo geral

Um modelo para protocolos de acesso a recursos compartilhados que seja genérico e extensível, permitindo fácil implementação em qualquer RTOS escrito em

uma linguagem orientada a objeto.

1.1.2 Objetivos específicos

- Verificar a generalidade do modelo implementando diferentes protocolos no RTOS escolhido;
- Observar e comparar resultados de diferentes protocolos implementados a fim de comprovar a correta execução dos protocolos no modelo;
- Comprovar a eficiência do design proposto mensurando parâmetros como rastro de memória e sobrecusto dos protocolos;
- Avaliar o impacto do sobrecusto dos protocolos realizando análises de escalonabilidade em conjuntos de tarefas;

2 FUNDAMENTAÇÃO TEÓRICA

Neste capítulo serão descritos com detalhes todos os protocolos a serem analisados no trabalho. Seus algoritmos, vantagens e desvantagens serão discutidos de forma a esclarecer teoricamente os objetivos que levaram à sua criação, e consequentemente, seu uso em aplicações de sistemas operacionais de tempo real.

Também serão discutidos temas relacionados ao uso desses protocolos, introduzindo e familiarizando o tema por meio de definições importantes como sistemas embarcados, sistemas operacionais, condições de corrida, entre outras.

2.1 SISTEMAS OPERACIONAIS

Pode-se dizer que não há uma definição exata para sistemas operacionais. Seu código, usualmente, encontra-se em *kernel mode*, mas isso pode mudar dependendo do tipo do sistema operacional (SO) utilizado. O sistema operacional tem duas funções, simplificar o uso do hardware por parte dos programadores que trabalham em *user mode*, e gerenciar os recursos da máquina (TANENBAUM; BOS, 2014, p. 3).

A primeira função é de extrema importância para que não seja necessário o entendimento de detalhes de um hardware por parte dos programadores, desta forma, os programas executados em *user mode* interagem com o hardware sem que os programadores entendam os detalhes. Um programa que necessita salvar ou ler dados de um disco rígido, por exemplo, não precisa interagir com o hardware, mas sim, com a interface disponibilizada pelo sistema operacional. Essa interface esconde os detalhes dos dispositivos, e permite a reusabilidade do mesmo código para diferentes máquinas.

Já a segunda função, gerenciamento de recursos, inclui gerenciamento de memória, processadores, dispositivos de entrada e saída, entre outros. Um sistema operacional deve possibilitar a execução de diferentes aplicações simultâneas, alocar memória para todas essas diferentes aplicações, além de detectar o uso de um mesmo recurso por diferentes aplicações, para multiplexá-lo se necessário (TANENBAUM; BOS, 2014, p. 3).

2.1.1 Sistemas operacionais de tempo real

No cotidiano da população, os sistemas operacionais estão amplamente presentes, de dispositivos móveis a computadores pessoais. Usualmente, tem-se contato com sistemas operacionais do tipo interativo, em que o sistema responde a comandos do usuário (TANENBAUM; BOS, 2014, p. 62). Por exemplo, uma interação de *click* com um ícone de aplicativo instalado, gera sua execução no sistema.

Este trabalho baseia-se porém, em um tipo diferente de sistema operacional, o de tempo-real. Sistemas operacionais de tempo real (RTOS) diferem-se dos sistemas operacionais usuais em restrições temporais que devem ser respeitadas por todas as tarefas (BUTTAZZO, 2011, p. 8). Executar os processos e tarefas necessárias não é o suficiente nestes sistemas, a execução das tarefas tem um prazo, um limite temporal que deve ser respeitado. Este prazo limite de execução atribuído a cada tarefa chama-se ***deadline***.

Isto tem um propósito, diretamente interligado com as aplicações e usos de um RTOS. Por exemplo, um software de um veículo, sistema que tipicamente utiliza RTOS, detecta uma batida durante sua execução e invoca um processo para abrir os *air-bags*. Assim, a execução do processo iniciado deve terminar obrigatoriamente antes de sua *deadline*, que neste caso deve ser calculada de forma a garantir a segurança dos passageiros. As condições atuais do veículo como estado e leitura de outros sensores, estado atual da memória ou número de processadores ocupados, não deve influenciar na execução da tarefa de forma que a faça perder sua *deadlines*.

Sistemas operacionais de tempo real podem ser classificados quanto a necessidade de cumprir suas *deadlines*. Sistemas nas quais *deadlines* devem ser cumpridas independentemente das condições atuais do sistema a todo custo, como veículos, são chamados de *hard real-time* (BUTTAZZO, 2011, p. 9). Normalmente esses sistemas utilizam de *deadlines* para garantir a segurança e consistência do sistema. Enquanto sistemas que tentam atender a maior parte das *deadlines*, não se importando caso uma ou outra tarefa não for atendida exatamente dentro do tempo estipulado, são considerados sistemas *soft real-time*, de menor criticalidade.

Em um sistema operacional usual, tais restrições temporais não existem, simplesmente por não serem necessárias na maioria das vezes. Por exemplo, um usuário ao iniciar seu computador, clica no ícone de seu *browser*, navegador, preferido e aguarda que seu processo seja executado para que a aplicação possa ser utilizada. Dependendo do estado atual do computador (uso de memória e processadores disponíveis), o tempo que levará para a aplicação ser iniciada pode variar muito. Nestes casos não são necessárias restrições temporais para a execução das tarefas, o sistema deve apenas tentar executá-las o mais rápido possível, sem sofrer punições, pois a integridade do sistema e do usuário não foi afetada.

2.2 PROCESSOS E *THREADS*

Segundo Tanenbaum (2014), um processo se resume a um programa, trecho de código, em execução tendo um **espaço de endereçamento** associado a ele. O espaço de endereçamento contém uma lista de endereços de memória que podem ser acessados pelo seu processo associado. Nesta lista estão contidos o executável do

programa, dados do programa e sua pilha.

Para que um único processo possa ser executado de forma paralela com diferentes funcionalidades relacionadas ao mesmo programa, as ***threads*** são utilizadas. *Threads* podem ser consideradas fluxos de execução em um espaço de endereçamento, sendo utilizadas por exemplo para que o áudio de um videogame seja processado simultaneamente em relação ao processamento de entradas do usuário. Ou para garantir que um processo editor de texto seja capaz de salvar automaticamente o progresso e verificar a sintaxe digitada de forma paralela, cada funcionalidade com seu próprio fluxo de execução.

Threads são basicamente processos executando no mesmo contexto de memória, podendo compartilhar a mesma memória durante a execução (TANENBAUM; BOS, 2014, p. 67–68). Múltiplas *threads* tem acesso ao contexto do processo ao qual estão inseridas, mas não têm acesso ao contexto de outros processos.

2.2.1 Tarefas

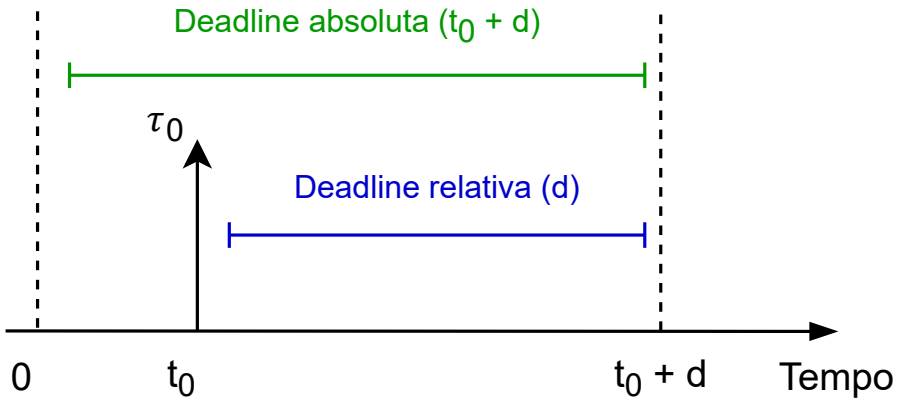
Um processo ou *thread* também é conhecido como uma **tarefa**, que por definição representa um conjunto de instruções a serem executadas carregadas na memória (TANENBAUM; BOS, 2014, p. 6–8). Em sistemas operacionais, tarefas podem ser periódicas, aperiódicas ou esporádicas (BUTTAZZO, 2011, p. 28). Tarefas aperiódicas normalmente são iniciadas por uma condição bem definida atingida em um intervalo aleatório de tempo, como por exemplo o acionamento de um botão ou a detecção de uma batida, que neste caso gera uma tarefa aperiódica para abrir o *air-bag* do veículo.

Tarefas periódicas usualmente fazem parte do fluxo principal de uma aplicação, representando tarefas que devem sempre ser executadas no mesmo intervalo durante toda a execução do processo. Uma nova instância de uma tarefa periódica é invocada a cada X unidades de tempo. Cada execução de uma tarefa iniciada em um tempo diferente é chamada instância, também conhecida como *job*. Tarefas periódicas podem ter uma *deadline* diferente do tempo de invocação da sua próxima tarefa.

As *deadlines* podem ser vistas de duas formas, *deadline* relativa e absoluta (BUTTAZZO, 2011, p. 27). Na absoluta o instante de tempo que representa a *deadline* é representado em relação ao início da execução da aplicação. Já na relativa, a *deadline* é representada em relação a unidade tempo na qual o *job* atual dessa tarefa foi iniciado. A Figura 1 demonstra a diferença entre ambas as *deadlines* para uma tarefa τ_0 invocada no instante t_0. O eixo x da figura representa unidades de tempo, e d representa o valor atribuído a *deadline* da tarefa.

Tarefas esporádicas são parecidas com as periódicas, a única diferença é que entre as diferentes instâncias das tarefas não há um tempo fixo, mas sim dinâmico. Entre cada instância de uma tarefa esporádica, existe um tempo mínimo para que um

Figura 1 – Representação das deadlines absolutas e relativas para uma tarefa



Fonte: Autor.

novo *job* da tarefa possa ser criado (BUTTAZZO, 2011, p. 28). A partir deste tempo estipulado, uma nova instância da tarefa pode ser iniciada a qualquer instante.

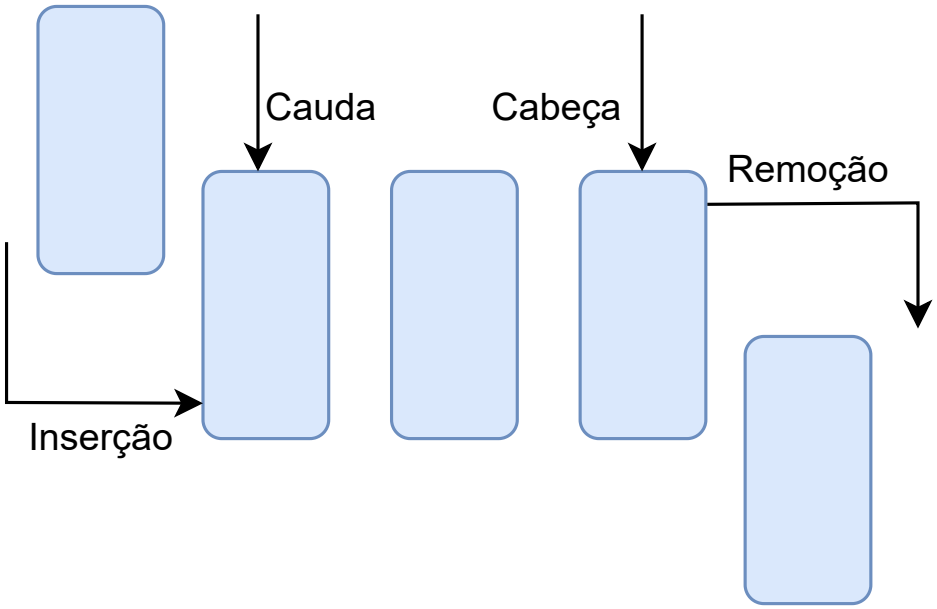
2.3 ALGORITMOS DE ARMAZENAMENTO DE DADOS

Na computação frequentemente faz-se uso de contêineres para armazenar dados temporariamente na memória, desta forma os dados podem ser ordenados, acrescidos e acessados mais tarde durante a execução do programa. Existem diversos tipos de algoritmos frequentemente utilizados para armazenar dados, cada qual com suas respectivas vantagens e desvantagens na implementação.

2.3.1 *First In, First Out*

O algoritmo *First In, First Out* (FIFO), também conhecido como *First Come, First Served* (FCFS) é um algoritmo para guardar dados ordenando-os em um sistema de fila simples. Basicamente o algoritmo guarda duas referências para dados, a cauda e a cabeça. Na Figura 2 o funcionamento do algoritmo FIFO é representado. Nela pode-se observar a cauda e cabeça da fila, que apontam para o último e primeiro dado armazenado, respectivamente. Sempre que um dado for inserido, sua inserção deve ser feita na cauda, considerada a última posição da fila (KRUSE; RYBA, 1998, p. 79). Caso o programador necessite de um dado armazenado utilizando a FIFO, o dado retornado será sempre a referência da cabeça, desta forma o algoritmo simula o funcionamento de uma fila.

Figura 2 – Representação das operações e funcionamento do algoritmo FIFO



Fonte: Autor.

2.3.2 Lista ordenada

Listas são utilizadas para armazenar dados de forma flexível, pois permite inserções e remoções de dados em quaisquer posição (KRUSE; RYBA, 1998, p. 213). Para isso, a lista também mantém referências para seus dados de cauda e cabeça, que devem ser atualizados conforme itens são adicionados ou removidos. A flexibilidade da lista permite que sejam utilizados diferentes tipos de algoritmos de inserção, dentre eles, a lista ordenada se destaca.

Manter os dados ordenados em uma lista é particularmente útil para simplificar o algoritmo de seleção e diminuir o tempo de busca por itens na lista. Por exemplo, caso uma lista ordenada seja utilizada para armazenar números inteiros, ao garantir que os dados estão ordenados a cada inserção e remoção, quando o usuário da lista necessitar do menor ou maior número armazenado, sabe-se que ele deve procurar na cabeça ou cauda da lista. Enxergando um número inteiro como uma prioridade *p* atribuída a uma determinada tarefa, observa-se que tarefas podem ser ordenadas em uma lista seguindo sua prioridade, garantindo assim, que a tarefa de maior prioridade sempre estará na cabeça da lista.

2.4 ALGORITMOS DE ESCALONAMENTO

Em uma aplicação executando em um sistema operacional de tempo real, muitas tarefas são invocadas e ficam ativas simultaneamente durante a execução.

Considerando que apenas uma tarefa pode ser executada simultaneamente em cada processador (TANENBAUM; BOS, 2014, p. 103–104), como até mesmo computadores com processadores *singlecore* (contendo apenas um núcleo) são capazes de executar múltiplas tarefas?

Na realidade, os sistemas operacionais convencionais e RTOS executam um rodízio constante nas tarefas que estão sendo executadas em seus processadores. Desta forma, todas as tarefas ativas são executadas de forma que o usuário não perceba que apenas uma tarefa está em execução no processador por vez. A forma como esse rodízio de tarefas nos processadores ocorre é chamada de algoritmo de escalonamento (TANENBAUM; BOS, 2014, p. 103–104). Esses algoritmos são responsáveis por ditar as regras que o sistema operacional utilizá para escolher qual tarefa deve ser executada em um determinado instante de tempo.

Algoritmos de escalonamento podem ser classificados como ótimos, *optimal*, caso consigam encontrar um escalonamento possível para qualquer conjunto de tarefas que seja escalonável de acordo com o algoritmo. Caso contrário, o algoritmo é considerado heurístico (BUTTAZZO, 2011, p. 36).

2.4.1 Testes de escalonabilidade

É muito importante que sejam executados testes em RTOS para determinar se um conjunto de tarefas é ou não **escalonável**, ou seja, se o sistema é capaz de executar com sucesso todas as suas tarefas periódicas durante a execução da aplicação. Estes testes são conhecidos como **testes de escalonabilidade**. Como pode-se imaginar, caso uma tarefa tenha um período menor do que o tempo necessário para sua execução, ela começará a perder *deadlines* fazendo com que existam mais de uma instância da mesma tarefa simultaneamente em execução.

Este problema torna-se mais complexo pois diferentes tarefas podem afetar o tempo de execução de uma tarefa, não apenas ela mesma. Não basta que uma tarefa sozinha no sistema seja capaz de executar dentro de sua *deadline*, ela deve ser capaz de atender a *deadline* considerando o conjunto de tarefas do sistema (BUTTAZZO, 2011, p. 46–48). Isto é importante caso por exemplo a tarefa seja bloqueada ao tentar acessar um recurso compartilhado, desta forma seu tempo de execução estará condicionado ao tempo que a outra tarefa levou para executar na seção crítica.

Os testes de escalonabilidade existem para garantir que um determinado conjunto de tarefas será escalonável para um algoritmo de escalonamento em específico. O cálculo muda para cada algoritmo, pois a maneira como as tarefas são selecionadas para serem executadas é diferente. Em geral, os testes usam as utilizações das tarefas do sistema para calcular uma utilização geral para o sistema (U_s).

A **utilização** de uma tarefa representa quantas unidades de tempo uma tarefa

executa em relação ao tempo disponível para que outra instância dela seja invocada. Representada pela letra U_i, a utilização ser calculada conforme a Equação 1, em que C_i é o tempo de computação da tarefa e T_i o período da mesma. O tempo de computação da tarefa é o tempo que uma instância da tarefa leva para terminar sua execução, considerando sempre o pior caso, conhecido como WCET (*worst case execution time*) (BUTTAZZO, 2011, p. 256).

$$U_i = \frac{C_i}{T_i} \tag{1}$$

As utilizações das tarefas do conjunto são então utilizadas para calcular a utilização do sistema, como apresentado na Equação 2 (BUTTAZZO, 2011, p. 82), em que N é o número de tarefas do conjunto, e as utilizações U_i são referentes as tarefas que compõem o conjunto. Para sistemas *multicore*, cada processador possui sua própria utilização a ser calculada. Após isso, a utilização do sistema é comparada com um limite estipulado. Limite este, calculado de forma diferente para cada teste de escalonabilidade.

$$U_s = \sum_{i=1}^{N} U_i \tag{2}$$

2.4.2 Classificações de testes de escalonabilidade

Os testes de escalonabilidade podem ser classificados quanto a sua capacidade de distinguir se determinado conjunto de tarefas é ou não escalonável, sendo classificados como suficientes, exatos e necessários (DAVIS; ZABOS; BURNS, 2008). Testes da categoria suficiente são restritivos e utilizados para concluir que determinados conjuntos de tarefas são escalonáveis. Caso o conjunto seja aprovado no teste, pode-se afirmar que ele é escalonável, porém existirão conjuntos escalonáveis que reprovarão no teste, logo, a reprovação não indica com certeza o conjunto não é escalonável.

Testes exatos são importantes, pois conseguem discernir exatamente se o conjunto de tarefas é escalonável ou não baseando-se em seu resultado. Usualmente são testes com comprovação mais complexa, não existindo para todos os tipos de algoritmos. Nos testes necessários, um conjunto de tarefas deve ter resultado positivo para que tenha uma chance de ser escalonável. Caso o conjunto reprove no teste, pode-se afirmar que o conjunto não é escalonável, porém, a aprovação no teste não significa que o conjunto em questão é escalonável.

2.4.3 Prioridades e preempção

Um parâmetro extremamente relevante aos algoritmos de escalonamento é a prioridade das tarefas. Em um sistema operacional de tempo-real toda tarefa deve ser atrelada a uma prioridade que indica o nível de relevância da tarefa em relação às outras (BUTTAZZO, 2011, p. 27). Os algoritmos alteram o nível de prioridade das tarefas, cada um à sua própria maneira, para que esse nível seja então utilizado pelo sistema para definir qual a tarefa que deve ser executada a seguir.

Os algoritmos podem ser preemptivos ou não preemptivos. Se uma tarefa τ_1 com prioridade p_1 é executada, e durante sua execução, uma nova tarefa τ_2 é invocada com prioridade p_2, considerando $p_2 > p_1$, o sistema pode ou não substituir a tarefa τ_1 em execução se for ou não preemptivo (BUTTAZZO, 2011, p. 24). Usualmente sistemas operacionais consideram que quanto menor o valor associado a prioridade de uma tarefa, maior seu nível de prioridade. Portanto, uma tarefa com nível de prioridade $p = 1$ tem maior prioridade que outra tarefa que tenha $p = 5$.

Se o algoritmo utilizado pelo RTOS for preemptivo, a tarefa executada é preemptada, e a tarefa de maior prioridade inicia sua execução. Caso o algoritmo seja não preemptivo, a tarefa τ_1 será executada até encerrar, para que então o algoritmo escolha a próxima tarefa considerando os níveis de prioridade.

Os algoritmos também são classificados pela forma com a qual definem a prioridade das tarefas. Se o algoritmo calcula as prioridades das tarefas antes da execução das mesmas e essa prioridade nunca é modificada de acordo com o instante de tempo, o algoritmo é considerado estático. Caso as prioridades das tarefas sejam modificadas ao longo de sua execução, diz-se que o algoritmo é dinâmico (BUTTAZZO, 2011, p. 36).

2.4.4 *Rate Monotonic*

O algoritmo *Rate Monotomic* (RM) (BUTTAZZO, 2011, p. 86) consiste em atribuir prioridades de acordo com a taxa de invocação das tarefas, desta forma, tarefas com menores períodos (invocadas com maior frequência) tem maior prioridade quando comparadas a outras tarefas que são invocadas com menor frequência. O RM é preemptivo e utilizado em RTOS para conjuntos de tarefas periódicas. Considerando que os períodos das tarefas são constantes, a prioridade atribuída e elas também o são, classificando o algoritmo como estático, trabalhando com prioridades fixas para as tarefas.

O teste de escalonabilidade para o RM é representado na Equação 3, com U_s calculado conforme a Equação 2, e N sendo o número de tarefas no sistema. Note que para $N = 1$, a utilização permitida ao sistema é de 100%, e conforme valores mais altos são atribuídos a N, o sistema tende a ser escalonável para porcentagens de 69%

de utilização (LIU; LAYLAND, 1973).

$$U_s \leq N(2^{\frac{1}{N}} - 1) \tag{3}$$

Reforça-se que o teste em questão é classificado como suficiente, ou seja, conjuntos de tarefas aprovadas no teste são garantidamente escalonáveis. Porém, podem existir conjuntos de tarefas escalonáveis reprovados no teste.

2.4.5 *Earliest Deadline First*

O algoritmo *Earliest Deadline First* (EDF) (BUTTAZZO, 2011, p. 100) calcula a prioridade das tarefas de acordo com o quão próximo estão de suas *deadlines* absolutas. Conforme a aplicação executa e as tarefas ficam próximas de suas *deadlines*, o EDF aumenta suas prioridades para que a tarefa mais próxima de perder sua *deadline* esteja sempre sendo executada.

O EDF é tipicamente utilizado de forma preemptiva, podendo ser utilizado em conjuntos de tarefas periódicas ou aperiódicas. Como a prioridade das tarefas é modificada durante a execução, o algoritmo é considerado dinâmico. O teste de escalonabilidade para o EDF consiste em garantir que a utilização do sistema seja menor ou igual a 100%, ou seja, a soma das utilizações de todas as tarefas do sistema não deve ser maior que 1, conforme apresentado na Equação 4 (XU; PARNAS, 1990).

$$U_s = \sum_{i=1}^{N} \frac{C_i}{T_i} \leq 1 \tag{4}$$

O teste é considerado exato quando o período das tarefas é igual a sua *deadline* ($T_i = D_i$). Teste considerado ótimo para o critério de tarefas com prioridades dinâmicas. Apesar de garantir taxas de utilização do sistema maiores quando comparado ao algoritmo RM, caso o sistema perca *deadlines*, seu funcionamento torna-se muito imprevisível, pois o sistema tentará atender todas as tarefas, ao contrário do algoritmo RM, que tenderia a perder *deadlines* das tarefas de menor prioridade, mantendo a consistência nas tarefas de maior prioridade.

2.4.6 Escalonamento de tempo real em sistemas *multicore*

Em sistemas *multicore*, no entanto, existem mais fatores a serem considerados. Resumidamente, existem duas formas principais de se trabalhar com algoritmos de escalonamento em sistemas *multicore*, de forma global ou particionada. Em algoritmos particionados, sempre que uma tarefa é invocada, é atribuída a um processador. Cada processador tem uma fila contendo as tarefas que devem ser executadas nele, e o algoritmo de escalonamento escolhido é aplicado a cada processador (GRACIOLI, 2014, p. 93). Já nos algoritmos globais, todas as tarefas invocadas são inseridas em

uma fila global, comum a todos os processadores. O algoritmo é aplicado a esta fila geral e os processadores executam as tarefas de acordo com a ordem da fila global (GRACIOLI, 2014, p. 97).

Tanto o algoritmo RM quanto o EDF podem ser utilizados em sistemas multicore sendo levemente modificados para trabalharem de forma particionada ou global, conforme detalhado anteriormente. A nomenclatura é intuitiva, caso o algoritmo RM seja aplicado a uma fila global de tarefas em sistemas *multicore*, o algoritmo é chamado de G-RM. Caso o algoritmo RM for aplicado em um fila para cada processador existente, ele é chamado de P-RM (GRACIOLI, 2014, p. 93). O mesmo vale para o algoritmo EDF.

2.5 CONDIÇÕES DE CORRIDA

Em um sistema operacional de tempo-real que utiliza várias tarefas trabalhando concorrentemente e/ou paralelamente, em casos de sistemas *multicore*, é importante garantir que essas tarefas não afetem negativamente o funcionamento uma das outras no efeito conhecido como **condição de corrida**.

A condição de corrida é um evento que ocorre quando mais de uma tarefa simultaneamente acessa dados compartilhados por várias tarefas (TANENBAUM; BOS, 2014, p. 82). Dependendo da ordem com que a variável compartilhada é acessada pelas tarefas um comportamento inesperado pode surgir, mesmo que não aconteça 100% das vezes. Para que esse efeito indesejado não ocorra, a aplicação deve garantir o conceito de **exclusão mútua**.

A exclusão mútua define que sejam identificadas **seções críticas** de memória no código, para que sejam devidamente isoladas, e múltiplas tarefas não consigam acessá-la simultaneamente. Todos os recursos compartilhados por mais de uma tarefa devem ser considerados seções críticas de memória, como por exemplo variáveis globais ou parâmetros estáticos de classes compartilhadas.

2.6 SEMÁFOROS

Existem vários métodos que podem ser utilizados para garantir a exclusão mútua em seções críticas do código, uma das mais conhecidas e empregadas é o uso de **semáforos** (TANENBAUM; BOS, 2014, p. 89). Semáforos foram introduzidos como termo da computação nos anos 60 por Edsger Dijkstra, como um método para garantir que variáveis não sejam acessadas simultaneamente por diferentes tarefas.

Um semáforo consiste em guardar uma seção crítica utilizando um método de bloqueio e espera. Ao ser iniciado, recebe um valor que representa a quantidade de recursos que podem acessar o recurso compartilhado ao mesmo tempo. Além disso, o semáforo provê duas operações, conhecidas como *p* e *v*. As siglas vem

do holandês *proberen* (testar) também chamada de *wait*, e *verhogen* (incrementar), também conhecida como *signal*.

Sempre que uma tarefa utilizar um recurso compartilhado, antes de acessá-lo, ela deve realizar a operação *p* do semáforo, que representa uma requisição de acesso ao recurso. Caso a variável interna do semáforo seja diferente de zero, significa que o recurso pode receber acessos, caso contrário, a tarefa fica bloqueada em uma fila de espera (DIJKSTRA, 1968). O pseudocódigo da operação *p* de um semafóro pode ser visualizado no Algoritmo 1.

Algoritmo 1: Operação *p* de um Semáforo

```
   Data: -
   Result: -
1  _res ← Recurso compartilhado
2  _task ← Tarefa tentando acessar o recurso
3  _ctr ← Quantidade de tarefas que podem acessar o recurso
4  _queue ← Tarefas bloqueadas no recurso
5  Início de operação atômica
6  if _ctr == 0 then
7  |  Insere _task na _queue
8  end
9  else
10 |  _ctr- -;
11 |  _task acessa _res
12 end
13 Término de operação atômica
```

Após ter acesso ao recurso, a tarefa efetua a computação desejada, e, ao sair da seção crítica, deve realizar a operação *v*, descrita no Algoritmo 2, sinalizando que a tarefa deixou de utilizar o recurso. A operação *v* implementada pelo semáforo deve incrementar o contador interno, representando que existe um novo espaço disponível para acessar o recurso compartilhado. Caso exista alguma tarefa bloqueada na fila de espera, ela deve ser imediatamente desbloqueada (seguindo a ordem da fila) e ganhar acesso ao recurso (DIJKSTRA, 1968).

É usual que esta fila de espera seja implementada como uma lista ordenada pela prioridade das tarefas, desta forma sempre que uma tarefa terminar sua execução em uma seção crítica, caso exista uma ou mais tarefas bloqueadas aguardando para acessar o recurso, a tarefa que ganhará acesso ao recurso sempre será a tarefa de maior prioridade dentre as bloqueadas.

Algoritmo 2: Operação *v* de um Semáforo

```
   Data: -
   Result: -
1  _res ← Recurso compartilhado
2  _task ← Próxima tarefa bloqueada no recurso a ser executada
3  _ctr ← Quantidade de tarefas que podem acessar o recurso
4  _queue ← Tarefas bloqueadas no recurso
5  Início de operação atômica
6  if tamanho _queue == 0 then
7  |  ctr++;
8  end
9  else
10 |  Remove _task da _queue
11 |  _task acessa _res
12 end
13 Término de operação atômica
```

2.7 *DEADLOCKS*

Deadlock é um fenômeno atribuído a um conjunto de tarefas. Diz-se que um conjunto de tarefas está em *deadlock* quando todas as tarefas deste determinado conjunto estão bloqueadas esperando por um recurso já acessado por outra tarefa do conjunto (TANENBAUM; BOS, 2014, p. 301–320).

Para garantir a exclusão mútua dos recursos compartilhados, diferentes tarefas não podem acessar um mesmo recurso simultaneamente. Desta forma, uma tarefa precisa que todos seus recursos sejam liberados em algum instante para que sua execução possa acontecer normalmente. Caso algum desses recursos não seja liberado, a tarefa fica aguardando na fila do recurso, possivelmente bloqueando outras seções críticas. Este evento pode ocasionar *deadlocks*, caso todas as tarefas de um conjunto necessitem de um recurso já alocado, esperando para sempre bloqueadas na fila dos recursos.

2.8 PROBLEMA DE INVERSÃO DE PRIORIDADE

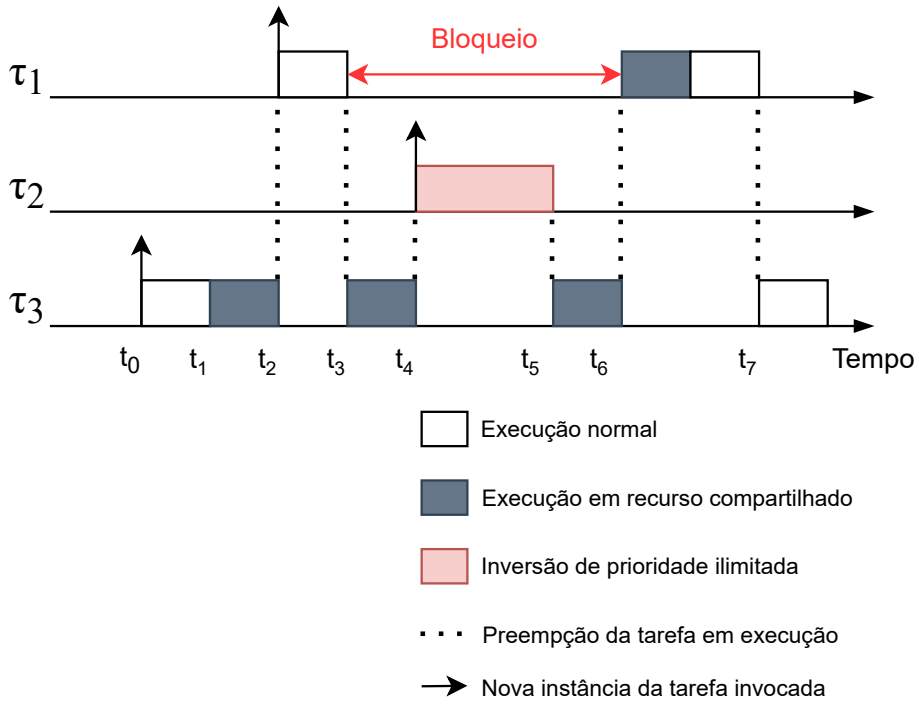
A previsibilidade é uma característica vital em sistemas de tempo real, pois o fator influencia diretamente na garantia da escalonabilidade do conjunto de tarefas. Se uma tarefa não é previsível, não é possível calcular exatamente seu tempo computacional e garantir que suas *deadlines* serão sempre cumpridas. Com isto a modelagem do sistema fica extremamente difícil, comprometendo a segurança de sistemas *hard real-time*.

Uma fonte de imprevisibilidade muito estudada são os recursos compartilhados,

pois os critérios de exclusão mútua fazem com que a execução das tarefas sejam atrasadas em decorrência do estado atual do sistema. Para visualizar isto será proposto um conjunto de tarefas, em que uma tarefa τ_1 tem prioridade $p_1 = A$, a tarefa τ_2 tem prioridade $p_2 = B$, e a última tarefa τ_3 tem prioridade $p_3 = C$. Considerando que $A > B > C$, entende-se que a tarefa de maior prioridade é τ_1, e a tarefa de menor prioridade é τ_3. Uma representação do exemplo citado pode ser observada na Figura 3.

A tarefa τ_3 inicia sua execução no instante de tempo t_0, e durante sua execução, no instante t_1 tenta acessar um recurso que também é compartilhado com uma tarefa de prioridade mais alta que a sua, τ_1. Caso a tarefa de prioridade mais alta seja executada logo em seguida, como ocorre no instante t_2, a tarefa τ_3 será preemptada e τ_1 será bloqueada na fila de espera do semáforo ao tentar acessar o recurso compartilhado durante sua execução no instante t_3. Isso causa uma inversão de prioridades, pois uma tarefa de menor prioridade está bloqueando a execução uma tarefa de maior prioridade, porém, isso é necessário para garantir o critério de exclusão mútua dos recursos.

Figura 3 – Demonstração do fenômeno da inversão de prioridade não limitada



Fonte: Autor.

O verdadeiro problema nesta situação advém da possibilidade da tarefa τ_3 ser preemptada durante sua execução dentro da seção crítica, como pode ser observado no instante t_4. Pois desta forma, a invocação de outras tarefas de prioridades maiores que p_3, como por exemplo τ_2, fazem com que τ_3 seja bloqueada enquanto bloqueia τ_1 na fila para acessar o recurso compartilhado. Assim, mesmo que τ_2 não acesse o recurso compartilhado em nenhum instante de tempo, sua execução bloqueia tarefas

de maiores prioridades. Este cenário pode ser observado entre os instantes t_4 e t_5. O cenário pode ser ainda mais drástico, caso existam mais tarefas sendo invocadas, atrasando assim a execução de τ_3, da qual τ_1 é dependente.

A ocorrência deste evento, chamado de inversão de prioridade ilimitada, *unbounded priority inversion* (BUTTAZZO, 2011, p. 206), pode ser observada na Figura 3 e se inicia no instante *t4*, quando a tarefa τ_2 é executada preemptando a tarefa τ_3. Consequentemente a tarefa τ_1 é bloqueada por um tempo indefinido, dependente da quantidade de tarefas que tenham prioridade contida no intervalo $A > p_i > C$. A solução para este fenômeno é a aplicação de protocolos de acesso a recurso compartilhados aos semáforos que guardam as seções críticas (BUTTAZZO, 2011, p. 208).

2.9 PROTOCOLOS DE ACESSO A RECURSOS COMPARTILHADOS EM SISTEMAS *SINGLECORE*

Estes protocolos tratam o fenômeno da inversão de prioridade ilimitada por meio de uma série de regras e condições impostas às tarefas que desejam acessar os recursos compartilhados. Cada protocolo faz isso de formas diferentes para diferentes algoritmos de escalonabilidade, obtendo por sua vez diferentes resultados no quesito eficiência. A seguir serão descritos alguns dos protocolos existentes voltados para RTOS *singlecore*, ou seja, apenas uma tarefa pode ser executada por vez.

2.9.1 *Priority Inheritance Protocol*

O *priority inheritance protocol* (PIP) (SHA; RAJKUMAR; LEHOCZKY, 1990) recebe este nome devido ao método utilizado para garantir com que a inversão de prioridade ilimitada não exista no conjunto de tarefas acessando um recurso compartilhado que seja guardado utilizando o protocolo. Pode ser utilizado em sistemas single-core preemptivos para conjuntos de tarefas de prioridade fixa.

Considerando um conjunto composto pelas tarefas τ_1 e τ_2, de prioridades A e B, respectivamente. Ambas as tarefas acessam um mesmo recurso compartilhado R_1, que é guardado por um semáforo que utiliza PIP. Dado instante de tempo, τ_2 inicia sua execução e acessa o recurso R_1, que não estava ocupado. Logo após, τ_1 é invocado, preemptando τ_2 e iniciando sua execução. Porém, ao tentar acessar R_1, a tarefa é bloqueada e a τ_2 volta a executar.

Nesse instante, a utilização do PIP faz a diferença. A prioridade da tarefa τ_1 é herdada pela tarefa τ_2, ou seja, no PIP, sempre que uma tarefa é bloqueada ao tentar acessar um recurso compartilhado por uma tarefa de menor prioridade, sua prioridade é repassada para a tarefa atualmente executando na região crítica. Isto faz com que tarefas de prioridade $B < p < A$ deixem de gerar o fenômeno da inversão de prioridade ilimitada ao serem invocadas durante a execução da tarefa de menor prioridade.

O principal ponto negativo do uso de PIP, quando comparado a outros protocolos, é o fato de que ele não garante a extinção de *deadlocks* no conjunto de tarefas. Para exemplificar, as duas tarefas do exemplo anterior serão utilizadas, porém agora, ambas acessam dois recursos compartilhados, R_1 e R_2. Caso τ_2 execute primeiro, acessando R_2, e logo após a tarefa τ_1 seja invocada, preemptando τ_2 e executando. Em sua execução, ela acessa primeiro o recurso R_1, e durante sua execução em R_1 tenta acessar R_2. Isso faz com que ela seja bloqueada e a τ_2 volte a executar. Porém, quando τ_2 tentar acessar R_1 em sua execução, não conseguirá jamais o acesso, pois o recurso está sendo utilizado por τ_1, atualmente bloqueado na fila do recurso R_2. Observa-se portanto, que o PIP não garante que o conjunto de tarefas não possa entrar em *deadlock*.

2.9.2 *Priority Ceiling Protocol*

O *priority ceiling protocol* (PCP) (SHA; RAJKUMAR; LEHOCZKY, 1990) pode ser utilizado em sistemas *singlecore* preemptivos com algoritmos de escalonamento que trabalhem com conjuntos de tarefas de prioridade fixa. Este protocolo utiliza um método em que cada recurso compartilhado tem uma prioridade teto associada a ele, denotada por $\Pi(R_i)$. A prioridade Π_i de cada recurso R_i é calculada como sendo a maior prioridade dentre as tarefas que acessam o recurso R_i. Logo, para que essa prioridade seja calculada, todas as prioridades devem ser conhecidas a priori.

O protocolo é uma extensão do PIP, porém, ao invés de herdar a prioridade da tarefa que é bloqueada no recurso R_i, o conceito de prioridade teto é utilizado. Sempre que uma tarefa tentar acessar uma região crítica de memória já ocupada, a tarefa utilizando o recurso tem sua prioridade aumentada para a prioridade teto $\Pi(R_i)$ de seu semáforo. Isto garante que uma tarefa só pode ser bloqueada sofrendo inversão de prioridade uma vez em cada recurso durante sua execução.

Ao deixar de utilizar o recurso compartilhado, caso a tarefa tenha tido sua prioridade aumentada durante sua execução, sua prioridade retorna ao valor base fixo. Caso exista uma tarefa bloqueada na fila de espera, ela ganha acesso ao recurso com sua prioridade base, só sendo aumentada para o teto do semáforo caso ocorra uma tentativa de acesso no mesmo recurso. Importante ressaltar que as tarefas bloqueadas no recurso são enfileiradas de acordo com sua prioridade, logo, a próxima tarefa a acessar o recurso é a tarefa bloqueada de maior prioridade no recurso.

2.9.3 *Immediate Priority Ceiling Protocol*

O *immediate priority ceiling protocol* (IPCP) (SHA; RAJKUMAR; LEHOCZKY, 1990) (BURNS; WELLINGS, 2001) é uma extensão direta do PCP com uma modificação na regra que dita o momento no qual a prioridade da tarefa executando no recurso deve

ser aumentada. No PCP, a tarefa acessando o recurso tem sua prioridade aumentada para o teto do semáforo quando uma nova tarefa tenta acessar o recurso em uso. Já no IPCP, assim que uma tarefa ganha acesso ao recurso, sua prioridade é imediatamente aumentada. Essa técnica se provou uma melhoria em relação ao PCP, pois diminui a quantidade média de troca de contextos entre tarefas, fazendo assim com que seu *overhead* seja menor em relação ao PCP.

Quando uma tarefa termina sua execução em um recurso sendo guardado por um semáforo que utiliza o IPCP, caso exista uma tarefa bloqueada no recurso, a próxima tarefa iniciará sua execução com sua prioridade imediatamente igual a prioridade teto do semáforo, garantindo assim, o mesmo funcionamento utilizado na aquisição do recurso.

2.9.4 *Stack Resource Policy*

O protocolo *stack resource policy* (SRP) (BAKER, 1991) foi criado para ser utilizado em sistemas com algoritmos de escalonamento de prioridade dinâmica, como o EDF. Por isso o protocolo não pode utilizar das prioridades das tarefas para evitar o fenômeno da inversão de prioridade ilimitada, logo, o protocolo faz uso de níveis de preempção.

Parecido com o PCP, o SRP se difere no instante em que as tarefas são bloqueadas. No SRP as tarefas são bloqueadas ao tentarem preemptar a atual tarefa em execução, ao contrário de outros protocolos em que o bloqueio ocorre quando a tarefa tenta acessar o recurso compartilhado. Desta forma nunca há bloqueio tentando acessar recursos, diminuindo a troca de contexto. Também pode-se afirmar que uma tarefa só pode ser bloqueada uma única vez durante toda a sua execução, sendo este bloqueio no momento em que tenta preemptar a atual tarefa em execução no processador.

Cada tarefa tem um nível de preempção associado a si, indicando o quão importante é a tarefa em comparação a outras, e relacionado a prioridade base da tarefa, aquela que não é modificada durante a execução do algoritmo de escalonamento. Para cada recurso compartilhado R_k existe uma prioridade teto, π_k, composta pelo maior nível de preempção dentre as tarefas bloqueadas aguardando para entrar em execução.

O protocolo também utiliza de um parâmetro global compartilhado entre todos os recursos do processador, a prioridade teto do sistema, Π. A prioridade teto do sistema é o maior valor dentre as prioridades teto de todos os recursos compartilhados. A lógica dos bloqueios é simples, a primeira tarefa ao tentar acessar um recurso compartilhado no sistema conseguirá acesso. Após isso, toda tarefa invocada que tente preemptar uma tarefa em execução deve ser bloqueada e inserida em uma fila do processador. Sempre que um recurso ficar disponível, a próxima tarefa ser executada

deve ter seu nível de preempção maior que a prioridade teto do sistema Π. Desta forma, sempre os recursos acessados por tarefas de maior níveis de preempção são liberados para execução primeiro.

2.10 PROTOCOLOS DE ACESSO A RECURSOS COMPARTILHADOS EM SISTEMAS *MULTICORE*

Trabalhar com sistemas operacionais de tempo real utilizando mais de um processador, *multicore*, é um constante desafio. Por muito tempo, sistemas desabilitavam seus outros núcleos, deixando assim apenas um processador, para fazer com que o sistema funcionasse de forma mais previsível. Previsibilidade é um dos fatores mais importantes para RTOS, pois é impossível garantir que todas as restrições temporais serão respeitadas em um sistema imprevisível.

A imprevisibilidade advinda de recursos compartilhados em sistemas multi-core deve ser tratada de forma diferente, os protocolos citados até então não são capazes de limitar a inversão de prioridade para conjuntos de tarefas executando em múltiplos processadores.

2.10.1 *Multicore Priority Ceiling Protocol*

Um dos protocolos utilizados para compartilhamento de recursos em sistemas *multicore* é o *multicore priority ceiling protocol* (MPCP) (RAJKUMAR, 1990). Sendo inspirado no IPCP e tendo funcionalidades parecidas, este protocolo pode ser utilizado em sistemas preemptivos que utilizem algoritmos de escalonamento com prioridades fixas atribuídas às tarefas. Deste modo, ele pode utilizar versões particionadas ou globais do algoritmo *rate monotonic* (RM), por exemplo.

O MPCP utiliza um conceito de recursos compartilhados locais e globais. Recursos são considerados locais quando é acessado apenas por tarefas de um mesmo processador. Por outro lado, recursos globais são acessados por tarefas de diferentes processadores. O MPCP define regras para recursos globais de forma simular ao IPCP, toda tarefa executando em um recurso global deve ter sua prioridade aumentada no instante de acesso para um teto global.

Diferentemente do PCP e IPCP, em que existe um teto para cada região crítica, cada tarefa tem sua prioridade aumentada para um valor diferente relacionado ao teto global. Para calcular o teto global de cada tarefa, é necessário conhecer a prioridade p_g do sistema antes de sua execução ser iniciada. Esta prioridade é a maior prioridade dentre todas as tarefas que acessam recursos compartilhados no sistema, usualmente calculada como a maior prioridade + 1, para evitar preempções desnecessárias em sistemas que podem preemptar tarefas com o mesmo valor de prioridade.

Calculado o valor de p_g, pode-se definir as prioridades de todas as tarefas do sistema ao acessar regiões críticas globais conforme exemplificado a seguir. Dado o recurso compartilhado global R_g, acessado pelas tarefas $\tau_1, \tau_2\tau_3$ e τ_4, de prioridades $p_{1,2,3,4} = 1, 3, 5, 10$, cada qual em seu respectivo processador. Portanto, pode-se afirmar que $p_g = 11$.

Ao acessarem o recurso R_g, as tarefas terão suas prioridades modificadas da seguinte forma:

- τ_1 terá sua prioridade aumentada de $p_1 = 1$ para $p_1 = 12$
- τ_2 terá sua prioridade aumentada de $p_2 = 3$ para $p_2 = 13$
- τ_3 terá sua prioridade aumentada de $p_3 = 5$ para $p_3 = 14$
- τ_4 terá sua prioridade aumentada de $p_4 = 10$ para $p_4 = 15$

Note que todas as tarefas tiveram suas prioridades aumentadas para valores acima de p_g, e a ordem original de suas prioridades não foi alterada, ou seja, $p_1 < p_2 < p_3 < p_4$. Estas são as condições do MPCP, mantendo a hierarquia original de prioridades, porém modificando a prioridade original para que todos os recursos compartilhados globais executem acima de um teto global, garantindo que recursos locais não interfiram em sua execução.

Caso o sistema tenha apenas um processador em execução, o MPCP se comporta de forma igual ao IPCP, pois o teto global se dá pelo teto das tarefas no processador, garantindo assim o mesmo comportamento. Além disso, o MPCP permite que a tarefa executando em uma região crítica seja preemptada por outra tarefa de maior prioridade acessando outra região crítica global. Como verificado no exemplo do PIP, isto pode causar *deadlocks* quando diferentes tarefas necessitam de recursos já bloqueados para finalizar sua execução, e os recursos já bloqueados necessitarem de outros recursos também em uso. Por este motivo, para que o MPCP seja aplicado, é proibido o acesso a recursos compartilhados aninhados.

2.10.2 *Multiprocessor Stack Resource Policy*

O protocolo *multiprocessor stack resource policy* (MSRP) (GAI; LIPARI; NATALE, 2001) é uma variação do protocolo SRP para sistemas *multicore*, que assim como seu antecessor, utilizam de algoritmos de escalonamento de prioridades dinâmicas, como o P-EDF. No MSRP, recursos compartilhados locais devem utilizar o protocolo SRP, pois o MSRP é uma extensão com funcionamento muito similar. Existem níveis de preempção para cada tarefa, assim como os tetos para cada recurso. Também foi mantido o instante de bloqueio, que deve ocorrer sempre que a tarefa for invocada e tentar preemptar uma tarefa em execução.

No MSRP, existe uma prioridade teto do sistema associada a cada processador, Π_k, esta prioridade é calculada com base na maior prioridade teto dos recursos globais acessados para cada processador k. Sempre que o algoritmo de escalonamento tiver

de escolher uma nova tarefa para executar, a tarefa escolhida deve ter seu nível de preempção maior que Π_k, considerando k o processador atual no qual a tarefa está alocada para execução, desta forma a preempção da tarefa é evitada durante sua execução.

O protocolo também considera que caso uma tarefa tente acessar um recurso compartilhado global, deve primeiramente verificar se o recurso não está sendo ocupado por outra tarefa, de forma similar aos protocolos citados anteriormente. A diferença encontra-se na fila que deve ser utilizada, o MSRP utiliza uma FIFO para armazenar as tarefas bloqueadas no recurso ao invés de uma lista ordenada por prioridade das tarefas.

3 PROJETO E IMPLEMENTAÇÃO DOS ALGORITMOS

Neste capítulo o projeto e implementação dos algoritmos serão explicados em detalhes. O modelo das tarefas, recursos compartilhados utilizados e algoritmos de escalonamento serão abordados na Seção 3.1. Na Seção 3.2, será apresentado o design proposto para a solução do problema, focando em reusabilidade de código. Após isso, a Seção 3.3 apresentará a implementação baseada na modelagem. Finalizando assim, com as considerações parciais na Seção 3.4, apresentando algumas das peculiaridades no modelo.

3.1 MODELO DO SISTEMA E TAREFAS

Ao longo da implementação e discussão de resultados, serão utilizadas tarefas, sempre referidas utilizando a letra τ_i, sendo i um indíce utilizado para diferenciar tarefas. As tarefas serão consideradas sempre periódicas, com suas *deadlines* (representadas pela letra d_i), sempre iguais a seu período T_i. Esta relação entre período e *deadline* é a mais comumente utilizada para tarefas periódicas em RTOS. O tempo de execução de uma tarefa é representado como C_i, sempre considerando o pior caso de execução, o WCET (*worst case execution time*), para que o modelo se baseie em sistemas de alta criticalidade, nos quais sempre o pior caso histórico deve ser utilizado de forma a abrangir todos os casos possíveis. Tarefas podem ser vistas como *threads*, cada qual associada a uma prioridade p_i. As prioridades podem ser fixas ou dinâmicas, caso sejam dinâmicas, cada tarefa terá uma prioridade base, não modificada. Durante sua execução uma tarefa pode variar apenas sua prioridade dinâmica.

Serão considerados dois tipos de algoritmos de escalonamento, o RM para prioridades fixas e o EDF para prioridades dinâmicas em sistemas *singlecore*, ambos preemptivos. Em sistemas *multicore*, serão utilizados as versões particionadas de ambos os algoritmos, P-RM e P-EDF. Desta forma, uma vez que determinada tarefa seja associada a um processador, mesmo que ela seja preemptada por outra tarefa, existe a garantia de que essa tarefa voltará a executar sempre no mesmo processador. A cada instante de tempo, o algoritmo de escalonamento escolherá a tarefa de maior prioridade para ser executada.

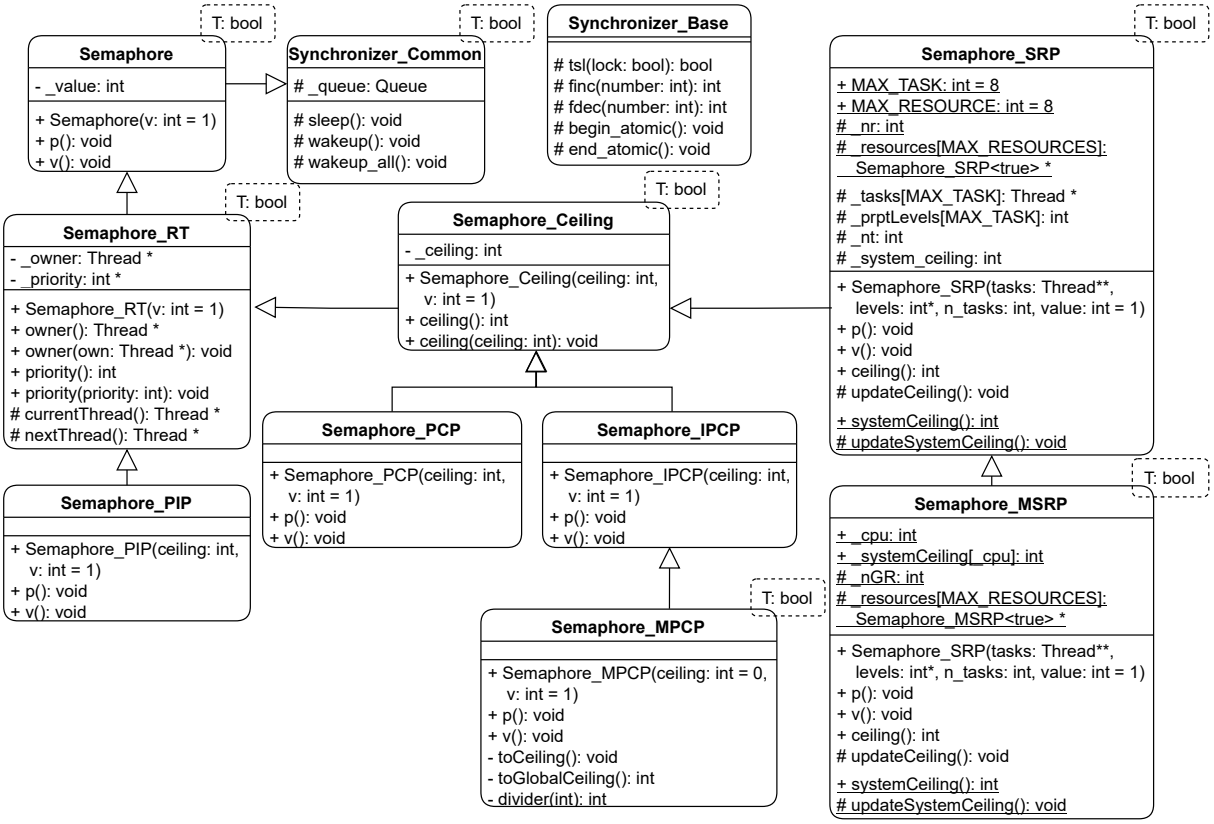
Durante a execução das tarefas, as mesmas terão de acessar e executar em seções críticas de código, ou seja, acessar recursos compartilhados representados pela letra R_i. Uma tarefa pode acessar diferentes recursos compartilhados de forma serial, mas não podem acessar diferentes seções críticas enquanto já executam em um recurso compartilhado. Serão utilizados dois diferentes tipos de seções críticas, locais e globais, sendo as locais acessadas apenas por tarefas de um mesmo processador, e

as globais sendo acessadas por tarefas de diferentes processadores.

3.2 PROJETO DE *SOFTWARE* DOS ALGORITMOS

A modelagem do software tem como principal objetivo permitir a implementação de diferentes protocolos de recursos compartilhados mantendo uma estrutura lógica e garantindo reusabilidade de software. O modelo deve compreender as principais categorias de protocolos, aplicáveis em sistemas *single* e *multicore* para prioridades fixas e dinâmicas, de forma a separar estruturalmente diferentes protocolos permitindo assim fácil extensão do modelo para adição de novos protocolos. Um diagrama de classes completo do sistema proposto utilizando a linguagem *Unified Modeling Language* (UML), pode ser observado na Figura 4, nele, estão contidas todas as classes utilizadas para implementação dos protocolos propostos, além das classes de utilidade essenciais para atingir o objetivo de reusabilidade de software entre diferentes protocolos.

Figura 4 – Modelagem do sistema



Fonte: Autor.

Dentre os mais diversos protocolos de acesso a recurso compartilhado disponíveis na literatura, foram escolhidos alguns dos mais populares e presentes em muitas implementações de sistemas operacionais de tempo real. O protocolo PIP

foi escolhido por ser uma proposta simples de solução para o problema da inversão de prioridade ilimitada, além de ser a base para criação de outros protocolos muito conhecidos e implementados. Por popularidade, foram escolhidos os protocolos PCP, IPCP e MPCP que utilizam a prioridade teto para eliminar a inversão prioridade ilimitada em sistemas que utilizam algoritmos de escalonamento de prioridade fixa *single* e *multicore*. Para expandir o modelo de forma a abranger algoritmos de escalonamento de prioridades dinâmicas, os protocolos SRP e MSRP foram escolhidos.

A classe *Synchronizer_Base* é responsável por implementar métodos que garantam a atomicidade das operações a serem implementadas nos demais protocolos, ou seja, seus métodos garantem que não haverá preempções ou interrupções durante a execução dos protocolos. A classe *Synchronizer_Common* herda da *Synchronizer_Base*, sendo responsável por implementar operações para bloquear ou acordar tarefas ao tentarem acessar os recursos compartilhados. Nela também encontra-se uma fila, em que serão armazenadas as tarefas bloqueadas no recurso. Esta fila pode ser implementada como uma FIFO ou uma lista ordenada por prioridade, dependendo do parâmetro *T* passado na construção da classe, considerando *TRUE* para lista ordenada por prioridade e *FALSE* para FIFO.

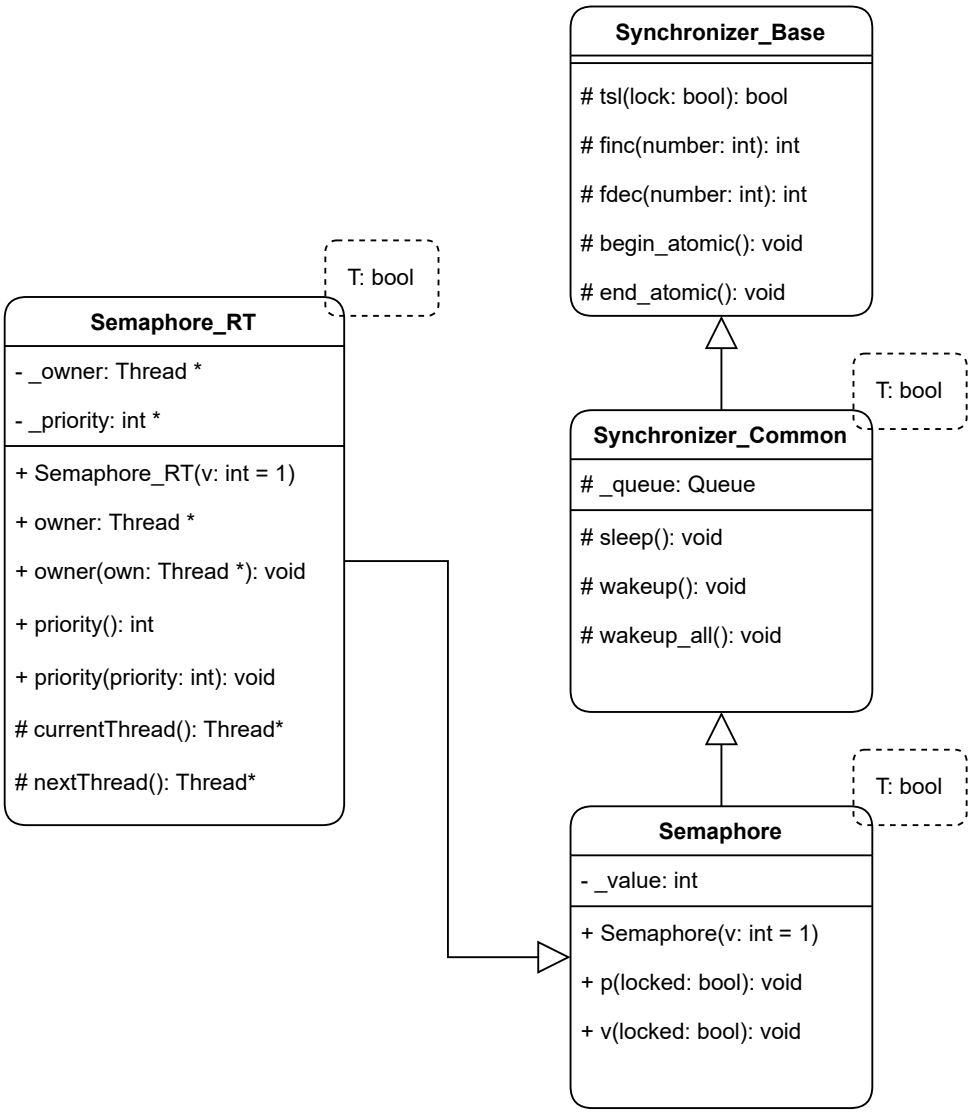
A implementação do semáforo clássico encontra-se na classe *Semaphore* que herda da classe *Synchronizer_Common* para ser capaz de armazenar as tarefas bloqueadas no recurso. A classe implementa os métodos *p* e *v*, além de repassar o parâmetro *T* de template para sua classe pai. Na Figura 5 o semáforo e suas interações com as demais classes pode ser observado com mais detalhes.

A classe *Semaphore_RT* é a responsável por fornecer métodos e operações comuns a muitos dos protocolos, garantindo assim a generalidade do código. A classe é capaz de armazenar a prioridade p_i de uma tarefa, particularmente útil para protocolos que modificam a prioridade da tarefa ao acessar o recurso compartilhado, pois desta forma a prioridade original da tarefa pode ser restaurada com sucesso uma vez que ela termine sua execução na seção crítica.

Como muitos dos protocolos de acesso a recursos compartilhados utilizam o conceito de prioridades teto de um recurso, uma classe foi criada especialmente para tratar de operações comuns para estes protocolos. A *Semaphore_Ceiling* fornece métodos para que os protocolos possam salvar e alterar o teto de recursos. Ambas as classes *Semaphore_Ceiling* e *Semaphore_RT* também necessitam de um parâmetro de template *T*, unicamente para repassá-lo até a classe *Synchronizer_Common*, para que ela seja capaz de escolher corretamente qual tipo de fila deve ser utilizada para o protocolo em questão.

Com a estrutura base do modelo definida, os protocolos de acesso a recursos compartilhados podem ser adicionados ao sistema facilmente. Protocolos que não utilizem prioridades teto como princípio para evitar a inversão de prioridade ilimitada,

Figura 5 – Modelo parcial do sistema, representação das classes utilitárias para os protocolos



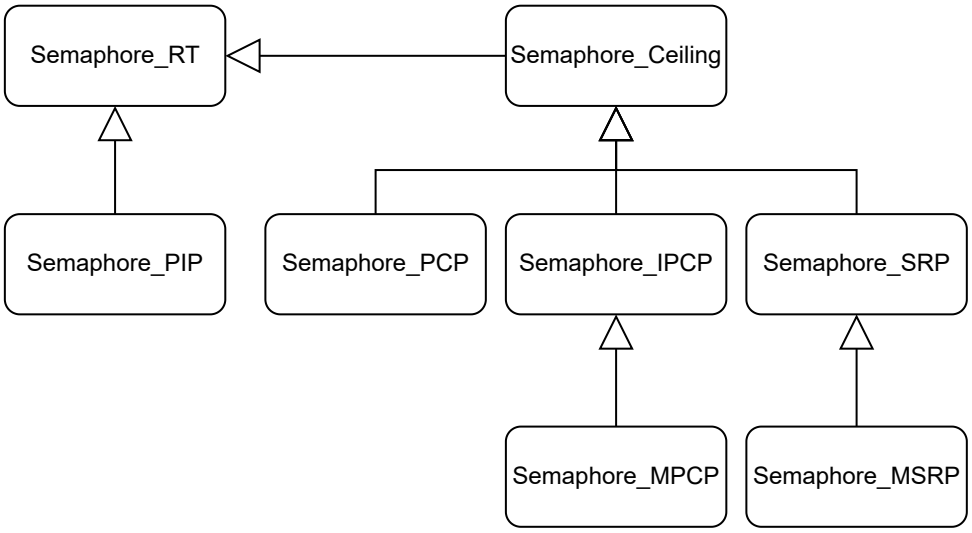
Fonte: Autor.

como o PIP, podem ser adicionados ao modelo herdando diretamente da classe *Semaphore_RT*, desta forma, suas classes podem se beneficiar da utilidade geral do modelo. Reforça-se o fato de que os protocolos implementados devem prover os valores de parâmetros *T* corretos durante a herança da classe para que o modelo funcione. Por exemplo, o protocolo PIP utiliza do sistema baseado em suspensão e fila de bloqueio ordenada por prioridade da tarefa, logo, deve herdar da classe *Semaphore_RT* com *T* igual a *TRUE*. Mais detalhes em relação aos parâmetros de herança na Seção 3.3.

Na Figura 6 podem ser observados os protocolos adicionados ao modelo, nota-se que protocolos *single* e *multicore* podem ser adicionados com facilidade à

modelagem do sistema, pois as classes bases permitem tal adição. Protocolos também podem herdar diretamente de outros protocolos implementados caso os protocolos sejam relacionados, como o MPCP, considerado uma extensão *multi-core* do protocolo IPCP.

Figura 6 – Modelo parcial simplificado do sistema, representação das adições de protocolos no modelo



Fonte: Autor.

3.3 IMPLEMENTAÇÃO

O sistema operacional de tempo real escolhido para implementar o modelo foi o *Embedded Parallel Operating System* EPOS (2022), devido ao seu modelo de fácil extensão para que os protocolos de recursos compartilhados sejam adicionados facilmente aos já funcionais semáforos do sistema. Além disso, o sistema operacional utiliza a linguagem C++ (GRACIOLI et al., 2013) garantindo baixos *overheads* na linguagem, contribuindo assim para um dos objetivos do modelo, que é a baixa interferência na escalonabilidade dos conjuntos de tarefas por meio de baixos *overheads*. A linguagem também é orientada a objeto, proporcionando uma relação bem estabelecida entre o modelo do sistema no diagrama UML proposto e sua implementação no sistema operacional.

O código referente a modificações no sistema EPOS e implementação dos protocolos de acesso a recursos compartilhados propostos encontram-se no Apêndice A. As classes *Synchronizer_Base* e *Synchronizer_Common* já haviam sido implementadas e estavam disponíveis no EPOS, sendo levemente modificadas e utilizadas para atingir os objetivos na extensão do sistema. A seguir, as implementações das classes referentes ao diagrama UML serão detalhadas em relação a sua implementação disponível no Apêndice A.

3.3.1 Implementação do semáforo tradicional

A classe *Semaphore*, responsável por implementar as clássicas operações de *p* e *v* foi levemente modificada em relação a sua versão já existente no sistema. Na prática, a classe foi nomeada *Semaphore_Template*, desta forma não foi necessário modificar todas as instâncias da classe *Semaphore*, já utilizada internamente em grande parte do sistema. Além das clássicas operações, a classe conta com o atributo privado *_value*, que deve ser passado como argumento no construtor da classe. Esta variável indica a quantidade de tarefas que podem acessar a seção crítica guardada simultaneamente, portanto, deve ser acrescido ou decrementado de acordo com a quantidade de tarefas liberando ou acessando o recurso. Por padrão, cada semáforo aceita apenas uma tarefa acessando o recurso simultaneamente.

Nas operações de requisição de acesso e liberação dos recursos, o código sempre garantia que a tarefa não seria preemptada durante o processo ao chamar as funções *begin_atomic* e *end_atomic*. Porém um *bug* foi encontrado durante os testes, em que em alguns casos a tarefa já havia desligado as interrupções em seu processador no momento em que executava a operação *p*, fazendo com que o sistema travasse ao tentar desativar as trocas de contexto novamente para a tarefa. O código foi então modificado para primeiramente verificar se a atomicidade das operações já havia sido garantida, e caso as operações já estejam sendo atômicas, o código não faz as chamadas dos métodos da classe *Synchronizer_Base*, evitando o travamento do sistema.

O problema citado é reproduzido no Quadro 1, nele observa-se um trecho do código contendo a chamada da função *p* implementada para o protocolo PCP, porém, este mesmo problema ocorria para qualquer um dos protocolos implementados. Na linha 2, o protocolo faz a chamada da função *begin_atomic()* da classe *Synchronizer_Base*, evitando assim que a tarefa em execução sofra preempção e garantindo a atomicidade das operações durante o processo de tentativa de aquisição do recurso. Após isso o algoritmo executa as instruções específicas do protocolo PCP, para então chamar a operação *p* clássica implementada no semáforo.

Então a função *p* da classe *Semaphore_Template* era executada e seu primeiro comando executado chamava a função *begin_atomic* novamente. Este comportamento ocasionava travamento no RTOS, pois as interrupções não podem ser desligadas enquanto já estão desligadas, em analogia, não pode-se liberar uma seção de memória alocada que já tenha sido liberada anteriormente. Porém, a função *p* deve manter a chamada da função *begin_atomic* de alguma forma, pois o semáforo pode ser utilizado independentemente pelo usuário, não necessariamente junto de um protocolo, mais detalhes sobre instruções de uso do modelo na Subseção 3.3.8.

Para contornar o problema, o *if* da linha 9 foi implementado e um parâmetro

Quadro 1 – Conserto do bug nas operações clássicas do semáforo

```
1   void Semaphore_PCP::p() {
2       begin_atomic();
3       /* ... */
4       Semaphore_Template::p(true);
5       end_atomic();
6   }
7
8   void Semaphore_Template::p(bool locked = false) {
9       if(!locked)
10        Base::begin_atomic();
11      /* ... */
12  }
```

Fonte: Autor.

booleano adicionado à chamada da função na linha 8. Desta forma, o usuário que utilizar um *Semaphore_Template* para guardar um recurso compartilhado, ao chamar as funções *p* e *v* não passará argumento algum, como é o usual das operações. Por outro lado, caso o usuário escolha guardar uma seção crítica utilizando um protocolo, como a classe *Semaphore_PCP* por exemplo, as funções *p* e *v* do protocolo ativam a atomicidade e chamam a função da classe pai passando o argumento com o valor verdadeiro, desta forma a classe pai entende que o sistema já foi trancado, e não executa a chamada da função *begin_atomic* novamente.

3.3.2 Implementação das classes utilitárias do modelo

A implementação da classe *Semaphore_RT* herda da classe *Semaphore_Template*, desta forma ela tem acesso ao parâmetro *_queue* protegido advindo da classe *Semaphore_Common*. Isto é importante para que a classe possa implementar o método *nextThread()*, que verifica se existe uma *Thread* bloqueada no recurso, e caso exista, a retorna. As principais funções implementadas na classe estão relacionadas a seus dois atributos privados, *_owner* e *_priority*. Por meio deles, a classe armazena a *Thread* atual executando no recurso compartilhado e sua prioridade base no momento de acesso ao recurso. A classe também fornece um método para evitar reescrever o mesmo trecho de código excessivas vezes, o *currentThread*, retornando a atual *Thread* em execução no processador.

A classe *Semaphore_Ceiling* herda diretamente da classe *Semaphore_RT* e fornece operações para armazenar e modificar seu atributo interno *_ceiling*. É por meio destes métodos que os protocolos deverão armazenar e modificar os valores teto dos recursos compartilhados. Importante citar que assim como sua classe pai, a *Semaphore_Ceiling* também recebe o parâmetro de template *T*, repassando-o na hierarquia de classes. Além disso, todas as classes citadas até agora recebem

o argumento *value* em seus construtores, que deverá ser repassado à classe *Semaphore_Template*, indicando a quantidade de tarefas que podem acessar o recurso simultaneamente.

3.3.3 Implementação *Priority Inheritance Protocol*

Com o *backbone* do modelo implementado, o sistema está completamente preparado para as adições dos procolos de acesso a recursos compartilhados em tempo real, tanto para aplicações *singlecore* como *multicore*. O primeiro protocolo a ser implementado foi o PIP, sempre seguindo um mesmo conceito em que, o programador usuário do modelo deve tentar acessar um recurso compartilhado utilizando o protocolo da mesma forma que utilizaria um semáforo tradicional. Para atingir isto, a classe *Semaphore_PIP* herda de *Semaphore_RT* escolhendo o parâmetro *T* de acordo com o tipo de fila necessário para o protocolo, obtendo desta forma acesso as operações utilitárias necessárias do modelo.

É importante que a implementação do protocolo contenha as operações *p* e *v*, desta forma o usuário utilizará um semáforo do tipo PIP, ao invés de um semáforo convencional. Na implementação do protocolo PIP, pode-se observar que a prioridade da tarefa já em execução no recurso apenas é aumentada caso ela tente ser preemptada por uma tarefa de maior prioridade. Vale também ressaltar o uso dos métodos utilitários da classe *Semaphore_RT* recebendo verdadeiro como parâmetro de template *T*, para armazenar a prioridade e referência da tarefa em execução no recurso.

3.3.4 Implementação *Immediate and Priority Ceiling Protocol*

Para os protocolos PCP e IPCP, foram criadas as classes *Semaphore_PCP* e *Semaphore_IPCP*. Ambas as classes herdam da classe *Semaphore_Ceiling*, classe utilitária para protocolos que utilizam o conceito de teto, que é o caso de ambos. Um detalhe que permanece é o valor do parâmetro *T* passado na construção da classe *Semaphore_Ceiling* como verdadeiro, indicando que ambos os protocolos utilizam filas ordenadas por prioridade das tarefas armazenadas.

As classes são extremamente parecidas, se diferenciando apenas no fato de que o IPCP altera a prioridade da tarefa que consegue acesso ao recurso imediatamente, enquanto o PCP espera que outra tarefa tente acessar o recurso para aumentar a prioridade da tarefa em execução para o teto. A simplicidade do modelo é posta a prova aqui, pois dois protocolos diferentes, de categorias similares são implementados sem muito esforço ao reutilizar ao máximo métodos das classes como *Semaphore_Ceiling* e *Semaphore_RT*.

3.3.5 Implementação *Multicore Priority Ceiling Protocol*

A implementação do protocolo MPCP ocorre na classe *Semaphore_MPCP* e contém um parâmetro de template *T* próprio da classe, caso o MPCP seja utilizado para guardar uma seção crítica global, a instância da classe deve ser criada passando *T* como verdadeiro. Desta forma, o algoritmo utilizará de seus próprios métodos para incrementar a prioridade da tarefa em execução no recurso para o teto global. Caso o MPCP seja utilizado para guardar uma seção crítica local, o parâmetro *T* deve receber falso, desta forma a especialização do template fará com que os métodos utilizados pela classe tenham o exato mesmo comportamento do protocolo IPCP.

A classe *Semaphore_MPCP* herda diretamente da classe *Semaphore_IPCP* por este motivo, caso o semáforo seja utilizado para guardar recursos locais, suas operações *p* e *v* chamarão as operações de sua classe pai. Caso o semáforo esteja guardando um recurso global, sua diferença estará na prioridade para o qual a tarefa que consegue acesso ao recurso receberá. No MPCP, diferentes tarefas acessando recursos globais devem ter suas prioridades ordenadas utilizando sua prioridade original como base, para isso a classe conta com métodos como o *divider* e o atributo privado *_pg*.

A função *divider* é responsável por ordenar as prioridades das tarefas utilizando operações matemáticas após serem elevadas para o teto global. Já o atributo *_pg* também é utilizado no cálculo de aumento de prioridade, mas representa o maior valor dentre as prioridades de tarefas que acessam recursos compartilhados no sistema. Seu valor é atribuído na criação da classe e é recebido por meio do sistema de configuração do RTOS. O programador deve preencher o arquivo de configuração com a maior prioridade do sistema, desta forma o protocolo utilizará este valor por padrão.

3.3.6 Implementação *Stack Resource Police*

A implementação do protocolo SRP é representada na classe *Semaphore_SRP*, que em seu construtor recebe um vetor contendo referências para as tarefas que acessarão o recurso, um vetor contendo os níveis de preempção das tarefas, a quantidade de tarefas que acessam o recurso, e por último, a quantidade de tarefas que podem acessar o recurso simultaneamente. Em seu construtor, a classe *Semaphore_SRP* também mantém internamente referências para todas as instâncias de semáforos do tipo SRP criados. O protocolo deve manter referências internas a todos estes parâmetros para ser capaz de calcular os tetos do recurso e do sistema. Os níveis de preempção são utilizados para que o protocolo possa ter um meio fixo de comparação de prioridades das tarefas, pois o protocolo SRP é utilizado em sistemas com algoritmos de escalonamento de prioridade dinâmica, como o EDF.

Um dos princípios que devem ser garantidos no SRP, é o fato de que uma tarefa

só pode ser bloqueada em um instante de tempo, logo após sua criação. Uma tarefa que tem uma instância criada e é escolhida pelo escalonador do sistema para executar, não deve ser bloqueada em nenhum momento, o protocolo deve garantir portanto que todos os recursos compartilhados da tarefa em execução estarão disponíveis.

O protocolo garante isto da seguinte forma, a primeira tarefa a executar no sistema que tente acessar uma seção crítica, irá atualizar os tetos do recurso e do sistema durante suas chamadas das funções *p* e *v*. Tais funções do semáforo SRP chamam a função *updateCeiling*, que atualiza o teto do recurso atual para o maior nível de preempção dentre as tarefas bloqueadas que acessam o recurso. Note que o teto apenas é atualizado caso o recurso compartilhado não aceite mais tarefas simultaneamente no recurso. Após isso, a função *updateSystemCeiling* é chamada para calcular o teto do sistema, que é nada mais que o maior teto dentre os recursos que utilizam o protocolo SRP.

A medida que novas instâncias de tarefas são invocadas no sistema, o escalonador do sistema é responsável por decidir qual tarefa deve ser executada. O código apresentado no Quadro 2 demonstra como este processo está relacionado diretamente com o protocolo SRP. A classe *Scheduler* no EPOS contém alguns métodos para escolher qual a tarefa que deve ser executada, como o *choose_next* na linha 7. O método utiliza um atributo privado da classe, *_queue*, que armazena tarefas ordenadas por prioridade para escolher qual deve ser executada.

Quadro 2 – Seleção de tarefas no escalonador para o protocolo SRP

```
1  Scheduler {
2  public:
3      /* ... */
4
5      // Function that chooses next task to run
6      Thread * choose_next() {
7          return _queue.choose();
8      }
9  private:
10     Queue<Thread *> _queue;
11 };
12
13 template<typename T>
14 Queue<T> {
15 public:
16     /* ... */
17
18     T choose() {
19         if( _head && eligible(_head->object()->criterion()) )
20             return _head->object();
21     }
22 };
```

Fonte: Autor.

Os métodos fornecidos pela classe *Queue* fazem uso da função *eligible* para

confirmar que a tarefa com maior prioridade no sistema pode ser escolhida para execução. A função *eligible* é declarada como parte do conjunto de escalonadores do sistema, porém, é definida juntamente do protocolo de acesso a recursos compartilhados SRP. Basicamente, a função compara o nível de preempção da tarefa que está tentando iniciar sua execução com a prioridade teto do sistema, retornando verdadeiro caso a tarefa tenha nível de preempção maior e possibilitando assim sua escolha por parte do escalonador.

3.3.7 Implementação *Multicore stack resource police*

O protocolo MSRP foi implementado na classe *Semaphore_MSRP* herdando da classe *Semaphore_SRP* devido as similaridades entre os protocolos, já que o MSRP é uma extensão para sistemas *multicore* do protocolo SRP. A implementação da classe de forma similar a implementação do protocolo MPCP, recebe um parâmetro de template *T* indicando se a classe será ou não utilizada para guardar um recurso global. Caso a classe seja criada passando falso para o parâmetro *T*, a classe poderá apenas guardar recursos compartilhados locais, e garantirá o mesmo comportamento do protocolo SRP ao fazer com que suas funções *p* e *v* chamem as funções definidas na classe base.

Caso uma instância da classe seja criada recebendo verdadeiro como valor no parâmetro *T*, a classe deve ser utilizada para guardar seções críticas globais. De forma similar a sua classe base, o protocolo MSRP também utiliza de atributos estáticos para armazenar referências a todas as tarefas e recursos globais no sistema, possibilitando assim que os métodos *updateCeiling* e *updateSystemCeiling* sejam capazes de calcular o teto atual do recurso e do sistema, respectivamente, para cada processador.

Nota-se também que a classe *Semaphore_MSRP* em sua definição herda da classe *Semaphore_SRP* passando o inverso do parâmetro *T* na construção de sua classe, como pode ser observado no código apresentado no Quadro 3. Isto ocorre pois ao longo de todo o modelo proposto, o parâmetro de template *T* tem o propósito de escolher que tipo de fila será utilizada para armazenar as tarefas bloqueadas que não conseguiram acesso ao recurso compartilhado. Considerando verdadeiro para uma lista ordenada por prioridade das tarefas e falso para uma FIFO.

Quadro 3 – Herança da classe MSRP

```
1 template<bool T>
2 class Semaphore_MSRP: protected Semaphore_SRP<!T> {
3 private:
4     typedef Semaphore_SRP <! T > Base;
5     /* ... */
6 };
```

Fonte: Autor.

Na definição do protocolo MSRP (GAI; LIPARI; NATALE, 2001), o autor estabelece em uma das regras que o protocolo deve armazenar as tarefas bloqueadas em um recurso utilizando filas FIFO. Esta definição gerou a sequência de templates adicionados às classes para que a classe *Synchronizer_Common* seja capaz de utilizar filas de diferentes tipos. Assim, quando uma instância da classe *Semaphore_MSRP* é criada para guardar recursos globais (recebendo verdadeiro como parâmetro *T*), automaticamente a classe herda de *Semaphore_SRP* passando falso, indicando que uma fila FIFO deve ser utilizada.

3.3.8 Construção de aplicações utilizando o modelo implementado

Um programador usuário de RTOS usualmente está familiarizado com o processo de aquisição de recursos compartilhados utilizando semáforos. Pensando nisso, o modelo do sistema foi preparado para que exista uma interface amigável para o usuário, em que o programador possa escolher dentre um dos protocolos implementados e utilizar das operações clássicas dos semáforos como se estivesse utilizando um semáforo tradicional.

Considere que para uma determinada aplicação, o sistema operacional EPOS está configurado para trabalhar com o algoritmo de escalonamento RM, ou seja, apenas um processador será utilizado, e as prioridades associadas as tarefas serão fixas. O usuário necessita que em sua aplicação duas tarefas sejam criadas, τ_a e τ_b, e em determinado momento de suas execuções, ambas devem acessar o mesmo recurso compartilhado R_i. O usuário escolhe portanto utilizar um dos protocolos de acesso a recursos compartilhados disponíveis no sistema. Dentre as opções disponíveis compatíveis com o escalonador RM, o protocolo PCP é escolhido. O Quadro 4 apresenta o código da aplicação exemplo criada pelo programador, já considerando a adição do modelo proposto.

Da linha 1 a linha 6, o usuário declara variáveis que serão utilizadas no código. Dentre elas estão, as duas tarefas que utilizarão o recurso compartilhado, um semáforo utilizando o protocolo PCP, os períodos p_a e p_b de ambas as tarefas, a quantidade de instâncias que as tarefas devem executar antes serem finalizadas, e o recurso compartilhado que será acessado por ambas as tarefas, neste caso, representado como uma variável do tipo inteiro, apenas para critérios de demonstração.

Na função *main*, linhas 28 a 45, o usuário cria uma instância da classe *Semaphore_PCP* que recebe dois argumentos em seu construtor, a prioridade teto do recurso que irá guardar e a quantidade de tarefas que podem acessar o recurso simultaneamente, para o último, caso nenhum valor seja repassado como argumento é considerado que apenas uma tarefa pode acessar o recurso simultaneamente. Considerando que no algoritmo de escalonamento RM o período das tarefas é utilizado como prioridade, e as prioridades atribuídas as tarefas são $p_a = 5000$ e $p_b = 4500$,

Quadro 4 – Exemplo de como utilizar o modelo

```
1  Semaphore_PCP * sem;  // Semaphore used to guard 'shared_resource'
2  int shared_resource = 5;
3  Periodic_Thread * t_a, t_b; // Threads that will use the shared resource
4  int period_a = 5000;  // thread a period in ms
5  int period_b = 4500;  // thread b period in ms
6  int iterations = 5;   // number of iterations (jobs) thread will run
7
8  void increment_routine() {
9      for(int i=0; i < iterations; i++)
10     {
11         sem->p(); // tries to access shared resource
12         shared_resource++; // thread computation at the shared resource
13         sem->v(); // release the resource
14         t_a->wait_next(period_a); // sleeps until next period
15     }
16 }
17
18 void decrement_routine() {
19     for(int i=0; i < iterations; i++)
20     {
21         sem->p(); // tries to access shared resource
22         shared_resource--; // thread computation at the shared resource
23         sem->v(); // release the resource
24         t_b->wait_next(period_b); // sleeps until next period
25     }
26 }
27
28 int main() {
29     // PCP receives ceiling priority as argument in constructor
30     sem = new Semaphore_PCP(4500);
31
32     // Creates threads to execute in a given function
33     t_a = new Periodic_Thread(&increment_routine);
34     t_b = new Periodic_Thread(&decrement_routine);
35
36     // Makes main thread await for t_a and t_b to finish
37     t_a->join();
38     t_b->join();
39
40     // Deallocate dynamic memory
41     delete t_a;
42     delete t_b;
43     delete sem;
44     return 0;
45 }
```

Fonte: Autor.

a maior prioridade dentre as tarefas acessando o recurso deve ser 4500, valor este repassado no construtor do semáforo na linha 30.

Após criar o semáforo, o usuário cria as duas instâncias de tarefas, atribuindo a cada uma delas uma função distinta. A tarefa τ_a irá executar a função *increment_routine*,

já a tarefa τ_b irá executar a função *decrement_routine*. Em suas execuções, as tarefas irão tentar acessar o recurso compartilhado exatamente como fariam em um semáforo tradicional, por meio da operação *p* do semáforo antes do trecho de computação na seção crítica. Caso consigam acesso ao recurso, utilizarão o recurso compartilhado, e ao terminarem liberarão o recurso utilizando a operação *v*, aguardando seu próximo período para executar novamente.

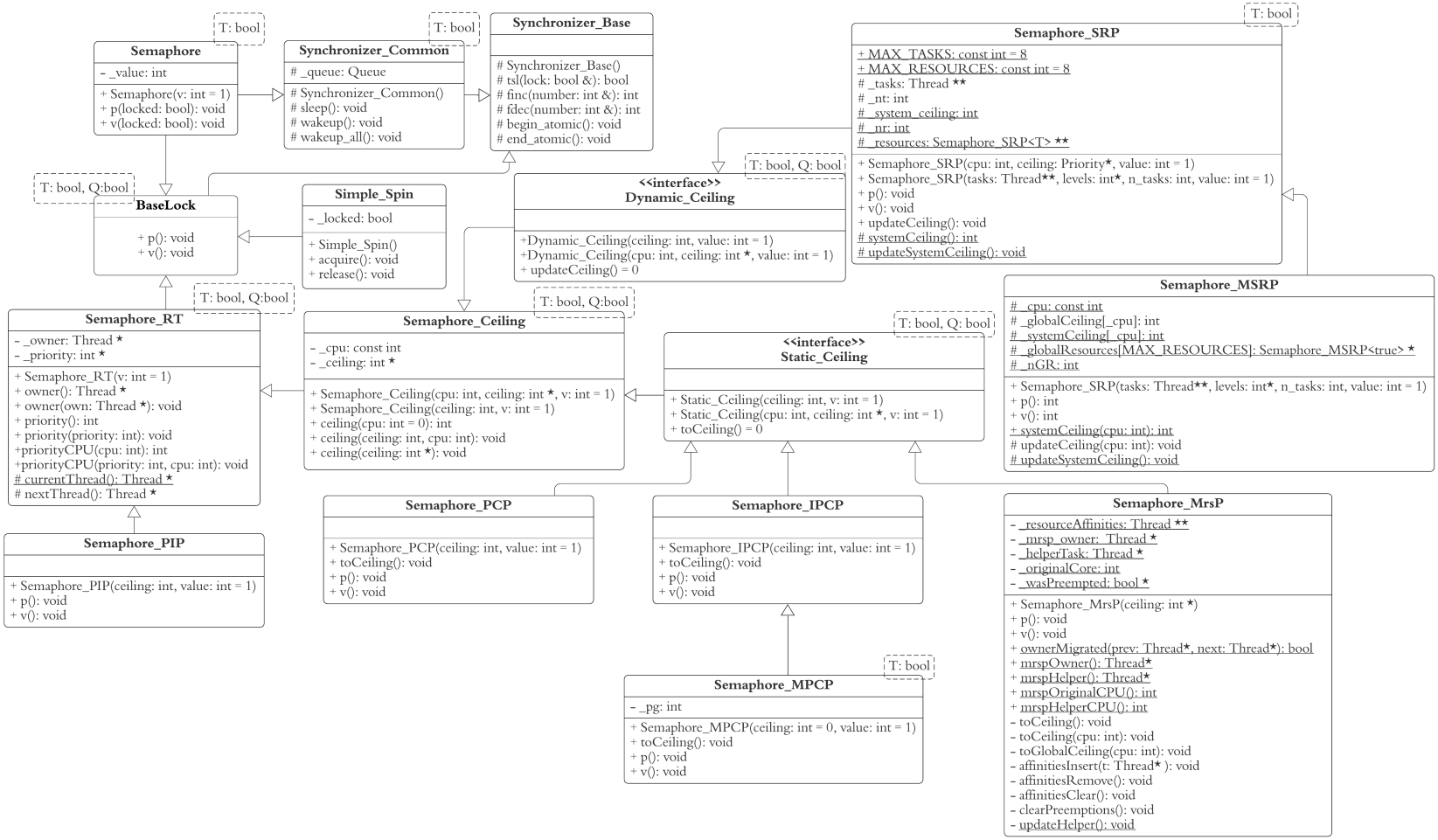
Desta forma, todo o algoritmo do protocolo PCP está escondido do usuário, o programador em momento algum precisa saber que, caso uma tarefa tente acessar o recurso compartilhado e não consiga acesso, pois já existe uma tarefa utilizando o recurso, a tarefa executando no recurso terá sua prioridade aumentada para a prioridade teto configurada na criação do recurso. Caso a tarefa tenha tido sua prioridade aumentada durante sua execução no recurso, no momento em que sinalizar sua saída utilizando a operação *v*, sua prioridade será restaurada e uma das tarefas armazenadas na fila de tarefas bloqueadas será escolhida para executar.

De forma similar, o programador pode escolher utilizar quaisquer um dos outros protocolos implementados, basta escolher um protocolo compatível com o número de processadores e algoritmo de escalonamento utilizado na execução do sistema. Com o protocolo escolhido, o usuário deve então guardar as seções críticas de código utilizando o mesmo procedimento de um semáforo tradicional, chamando a função *p* para requisitar acesso, e a função *v* para indicar que a tarefa deixou de utilizar o recurso compartilhado.

3.4 CONSIDERAÇÕES PARCIAIS

Neste trabalho houve uma tentativa de abranger ainda mais o modelo proposto, facilitando a extensão do modelo não apenas para protocolos que utilizem sistemas de bloqueio de tarefas, semáforos, mas também que contenha operações básicas para protocolos que utilizem métodos como *spin-locks* (WIEDER; BRANDENBURG, 2013). O modelo proposto originalmente pode ser encontrado na Figura 7.

Figura 7 – Modelagem do sistema

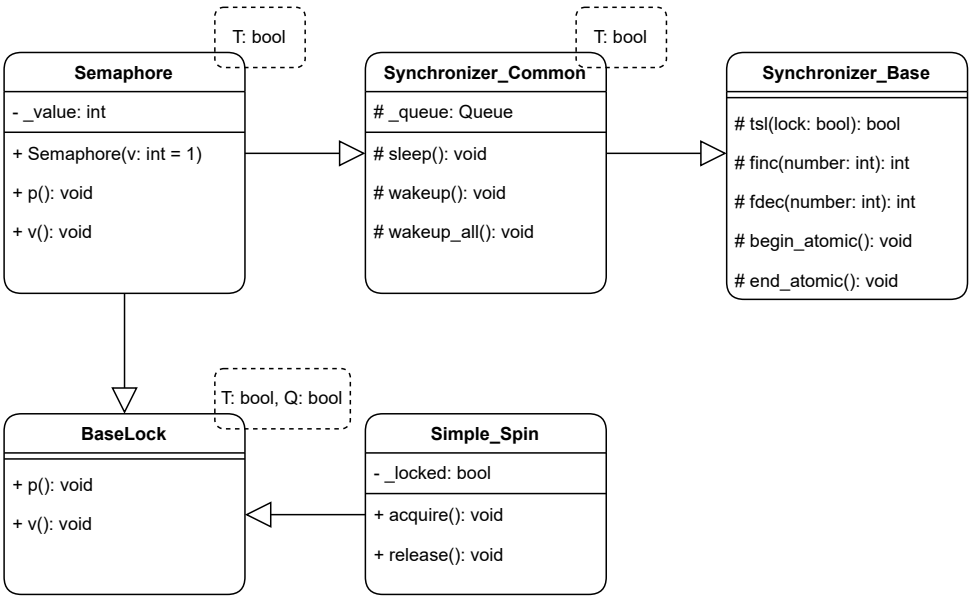


Fonte: Autor.

Uma das diferenças em relação ao modelo apresentado anteriormente é a implementação de *spin-locks*, encontrada na classe *Simple_Spin*. Por meio dela os clássicos métodos *acquire* e *release* são fornecidos. Para garantir a abrangência do modelo a protocolos que utilizem diferentes métodos de suspensão a tarefas, a classe *BaseLock* é utilizada. Desta forma, a classe é responsável por distinguir se o protocolo em questão utilizará o modelo baseado em suspensão (semáforos) ou baseado em *spin-locks*.

A classe tem dois parâmetros template, que se estenderão ao longo das próximas classes com o mesmo significado, o parâmetro *T* repassado na construção da classe *Semaphore* representando o tipo de fila a ser utilizado, e o parâmetro Q utilizado para determinar de qual classe a *BaseLock* deve herdar seu comportamento de contenção no acesso a seções críticas. Caso Q seja verdadeiro a *BaseLock* herdará da classe *Semaphore*, caso contrário, a classe herdará de *Simple_Spin*. Importante notar que o parâmetro *T* só é utilizado caso Q seja verdadeiro, pois a classe *Simple_Spin* não utiliza de filas para armazenar tarefas bloqueadas em um recurso. O diagrama UML demonstrando a hierarquia das classes envolvidas no processo de sincronização e fornecimento de operações base pode ser observado na Figura 8.

Figura 8 – Modelo parcial do sistema, representação das classes bases de sincronia



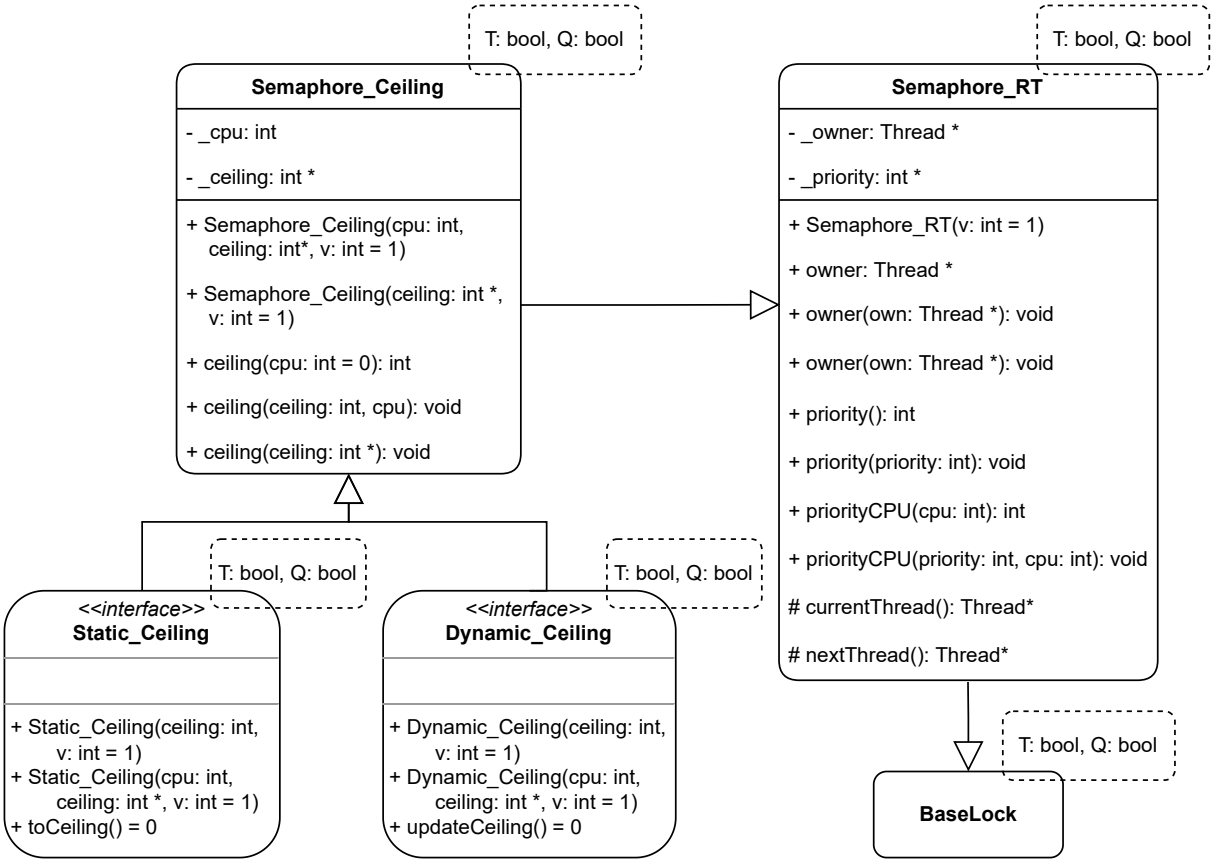
Fonte: Autor.

Duas classes herdam da classe *Semaphore_Ceiling* proporcionando grande reusabilidade de código, as interfaces *Static_Ceiling* e *Dynamic_Ceiling*, que podem ser observadas com mais detalhes na Figura 9. Todos os protocolos que fazem uso de ceilings fixos, devem ser classes filhas de *Static_Ceiling* e prover um método para aumentar a prioridade base da tarefa para o teto de um determinado processador.

Os protocolos que utilizam lógica de teto com algoritmos de escalonamento de

prioridade dinâmica, irão frequentemente atualizar seu teto de acordo com o algoritmo utilizado. Comparando assim a atual prioridade dinâmica com o teto de seu processador. A interface *Dynamic_Ceiling* também obriga que classes filhas da interface forneçam um método para atualizar o teto atual do recurso sendo acessado. Ao utilizar ambas interfaces, o programador pode facilmente estender o modelo ao adicionar novos protocolos implementados utilizando o conceito de herança. Caso o novo protocolo adicionado não trabalhe com a ideia de prioridade teto, sua lógica pode ser posicionada no modelo em outro lugar da cadeia de classes, idealmente, herdando da classe *Semaphore_RT*, criada justamente para este propósito.

Figura 9 – Modelo parcial do sistema, representação das classes utilitárias e interfaces de prioridade teto



Fonte: Autor.

Além dessas mudanças pode-se notar a adição de um protocolo, *multicore resource shared protocol* (MrsP) (BURNS; WELLINGS, 2013). O protocolo tem seu algoritmo baseado em *spin-locks* e uma implementação relativamente complexa. Basicamente o protocolo não bloqueia tarefas, as deixa "girando" enquanto aguardam sua vez de executar no recurso compartilhado. Caso a tarefa em execução no recurso seja preemptada, uma das tarefas "girando" em outro processador, executa na seção crítica o contexto da tarefa preemptada. Desta forma as tarefas não ficam apenas aguardando sua vez, mas também ajudam a tarefa na seção crítica caso seja

necessário.

Embora o protocolo seja interessante, houve diversos problemas durante sua implementação, e a incapacidade de resolver alguns *bugs* em tempo hábil fizeram com que o protocolo fosse abandonado, e com ele, toda a modelagem relativa a *spin-locks*. Desta forma, este modelo não foi utilizado, dando lugar ao modelo apresentado previamente na Seção 3.2.

4 RESULTADOS

Com a implementação do modelo proposto em mãos, o próximo passo consiste em garantir que o objetivo do trabalho tenha sido alcançado por meio da implementação, ou seja, o modelo precisa ser posto a prova numericamente de forma a calcular a leveza da implementação e seus impactos na escalonabilidade dos conjuntos de tarefas. Na Seção 3.2 o modelo proposto apresentado provou suas características subjetivas de desenvolvimento de *software*, como, sua fácil extensão e manutenção devido a reusabilidade de código. O modelo proposto também é facilmente replicável em RTOS que utilizem linguagens orientadas à objeto.

4.1 ANÁLISE DO CÓDIGO EM TEMPO DE COMPILAÇÃO

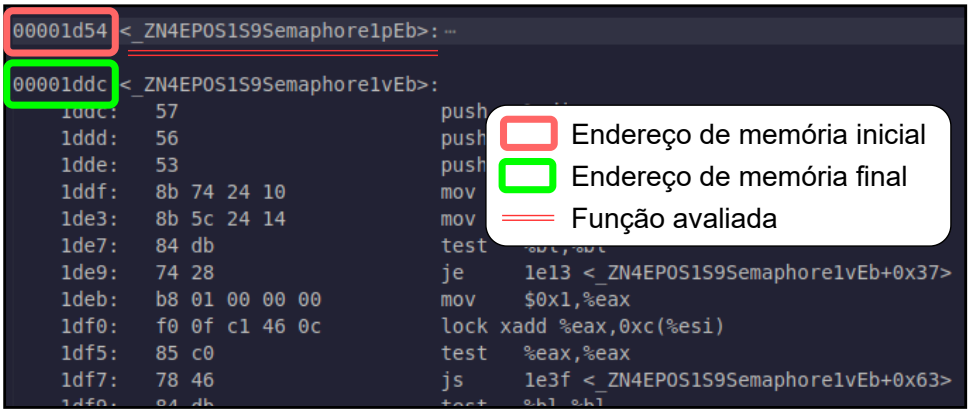
Um dos critérios utilizados para avaliar a leveza da implementação foi a medição do rastro de memória do código implementado, *memory footprint*, que consiste em calcular a quantidade de memória utilizada pelo modelo proposto. Permitindo também efetuar comparações com diferentes implementações do modelo, e principalmente, avaliar se a quantidade de memória utilizada pela adição do modelo de protocolos é viável para sistemas operacionais de tempo real no geral.

Para medir o rastro de memória da implementação, primeiramente o código do EPOS modificado foi compilado utilizando o compilador *GNU gcc* na versão 7.5.0 para gerar o código para a arquitetura Intel 32 bits, conhecida como IA32, mais especificamente para o processador Intel i7-2600 com 3.4 GHz, 8 núcleos e uma memória cache L3 de 8 MB. Com o arquivo executável gerado, a ferramenta *GNU objdump* na versão 2.30 foi utilizada para medir o rastro de memória do código. Ao utilizar o comando `objdump -d <nome_executavel>`, o executável é desmontado e exibido em formato *assembly*, permitindo assim que os tamanhos de seções do código sejam calculados.

Para exemplificar o processo, a Figura 10 mostra um trecho do arquivo gerado após o uso da ferramenta *objdump* para desmontar o executável. Primeiramente a função que irá ter seu tamanho calculada precisa ser localizada no arquivo, usualmente seu nome será relacionado com o nome definido na implementação, podendo variar conforme o compilador. No exemplo da figura, a função escolhida para ser avaliada é a `Semaphore<true>::p()`, devidamente sublinhada. O valor hexadecimal à esquerda do nome da seção é o endereço de memória em que a declaração da função inicia. Para calcular o tamanho total da função também faz-se necessário descobrir o endereço final de sua declaração, para isso, a próxima função declarada no arquivo é localizada, e seu endereço inicial é considerado o endereço final alocado para a função anterior. Assim,

a quantidade de memória necessária para chamar a função `Semaphore<true>::p()` é `0x1DDC - 0x1D54 = 136 bytes`.

Figura 10 – Trecho do executável gerado convertido em assembly



Fonte: Autor.

Os resultados do processo podem ser observados na Tabela 1, que relaciona o rastro de memória em bytes para cada classe do modelo. O consumo de memória das classes foi divido em três tipos permitindo assim melhor comparação entre as diferentes classes e a exibição dos reais motivos caso alguma das implementações utilize quantidade exagerada de memória. A quantidade de memória que uma instância da classe utiliza para armazenar atributos internos é apresentada na coluna **Dados**, logo, a quantidade de memória relacionada na coluna será multiplicada pela quantidade de instâncias da classe criadas na aplicação do usuário. A quantidada de memória armazenada estaticamente em uma classe é apresentada na coluna **Dados estáticos**, representando os atributos das classes que utilizam o modificador *static* da linguagem C++. Estas variáveis são criadas uma única vez independentemente da quantidade de instâncias de classes criadas na aplicação, ou seja, o consumo de memória desta coluna é fixo e independente da quantidade de recursos compartilhados a serem guardados na aplicação do usuário.

Tabela 1 – Rastro de memória das classes implementadas (em bytes)

Classe	Código	Dados	Dados estáticos	Memória total
Semaphore	272	16	0	288
Semaphore RT	0	8	0	296
Semaphore Ceiling	0	4	0	300
PIP	580	0	0	876
PCP	634	0	0	934
IPCP	624	0	0	924
SRP	644	168	53	1165
MPCP Global	666	0	4	1594
MSRP Global	656	32	73	1926

Fonte: Autor.

A coluna **Código** representa o tamanho das seções de código calculadas conforme citado anteriormente utilizando a ferramenta *GNU objdump*, de forma a exprimir em bytes a quantidade de memória utilizada para invocar as funções de requisição e liberação de acesso ao recurso compartilhado, ou seja, incluem apenas as funções chamadas durante o processo de interação com o recurso, *p* e *v*, além das funções de bloqueio no escalonador para os protocolos SRP e MSRP. A última coluna, **Memória total**, representa a soma dos três tipos de consumo de memória relativo a classe, acrescentados da memória total consumida pela implementação de sua classe base. Por exemplo, a classe *Semaphore_PIP* tem um consumo total de memória de 580 bytes, já a classe *Semaphore_RT* tem um consumo de 296 bytes, logo, como a classe herda diretamente da classe *Semaphore_RT*, seu consumo total de memória será 580 + 296, portanto, 876 bytes.

4.2 MEDIÇÃO DOS SOBRECUSTOS DA IMPLEMENTAÇÃO

Para avaliar a influência da implementação no tempo de execução das tarefas, um temporizador foi adicionado ao EPOS para permitir a medição dos sobrecustos nas funções utilizadas pelos protocolos de acesso a recurso compartilhado do modelo. Sobrecusto neste caso pode ser definido como o tempo adicional inserido às operações para que um semáforo funcione utilizando um protocolo. O código do temporizador encontra-se no Apêndice A, de forma resumida, a classe *ITimer* conta com diversos atributos estáticos para armazenar valores de *ticks* contados pelo *Timestamp Counter* (TSC) do sistema operacional. Desta forma o usuário pode criar instâncias da classe em qualquer lugar do código, permitindo calcular a quantidade de tempo que uma tarefa executa em uma função qualquer. O tempo é armazenado sempre no mesmo vetor estático para todas as instâncias da classe, permitindo a fácil instrumentação e recuperação dos valores armazenados posteriormente.

O Quadro 5 exemplifica o uso da classe instrumentando a operação *p* para o semáforo MSRP. Uma instância do temporizador é criada logo após as interrupções serem desligadas na linha 3, e o temporizador começa a contar imediatamente em seu construtor. Após as operações do protocolo, o temporizador é paralisado logo antes das interrupções serem ativadas novamente. A classe também conta com um método *printData* para exibir todos os tempos salvos ao longo da execução da aplicação, assim, a função principal da aplicação de teste deve criar uma instância da classe *ITimer*, e ao final da rotina de testes invocar o método para imprimir o vetor estático da classe. As funções de requisição e liberação de recursos compartilhados (*p* e *v*) de todos os protocolos foram instrumentadas para avaliar a quantidade de tempo que os protocolos levam para executar suas funções essenciais e evitar a inversão de prioridade ilimitada.

Com os protocolos instrumentados, foram criadas aplicações de teste similares

Quadro 5 – Exemplo de função de acesso a recurso instrumentada

```
1  void Semaphore_MSRP::p()
2  {
3    begin_atomic();
4    ITimer t;
5
6    Semaphore_RT<false>::p(true);
7    updateCeiling();
8
9    t.stop("msrp_p", this);
10   end_atomic();
11 }
```

Fonte: Autor.

ao código apresentado no Quadro 4. Para testar os protocolos *singlecore* a aplicação cria N tarefas que tentam acessar um único recurso compartilhado no sistema em cascata. A primeira tarefa a conseguir acesso ao recurso executa na seção crítica com o tempo de computação de 1ms (um temporizador foi utilizado para atingir este comportamento), e logo após, libera o recurso e espera até seu próximo período (50ms). Em cada iteração do teste a quantidade de tarefas foi aumentada até o máximo de 20. Nas aplicações teste para os protocolos *multicore*, cada tarefa é criada e associada a um processador, todas compartilhando um mesmo recurso global.

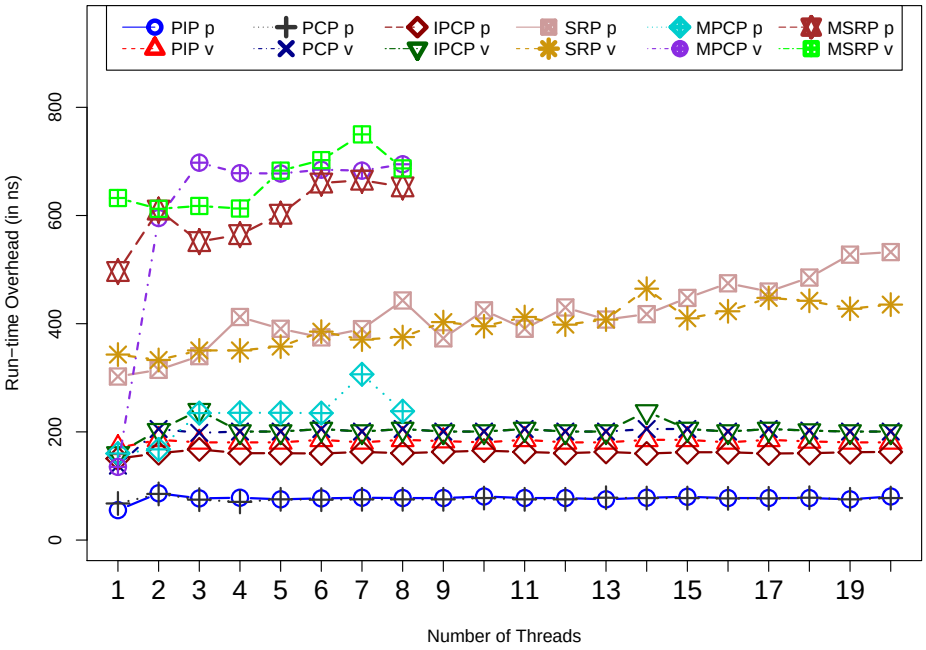
No teste, o número de processadores, e consequentemente de tarefas, foi variado entre 1 a 8 entre diferentes execuções. O número máximo de 8 tarefas foi utilizado devido a uma limitação de *hardware*, o processador utilizado nos testes tinha 8 núcleos. Optou-se por seguir os testes com no máximo uma tarefa por núcleo de processador, para que exista a garantia de que uma tarefa seja bloqueada no recurso sempre por uma tarefa de um núcleo diferente. Isto é importante para protocolos como o MSRP e MPCP, que têm seu funcionamento alterado de acordo com a seção crítica que estão guardando (local ou global). Além disso, em sistemas *multicore*, é muito difícil controlar o sistema com muitas tarefas executando no mesmo núcleo.

A rotina de execução das tarefas no sistema *multicore* foi igual a rotina utilizada nas tarefas das aplicações de teste *singlecore*. Em cada iteração dos testes *single* e *multicore*, a aplicação foi executada 1000 vezes, e o pior caso detectado no tempo de execução (WCET) das tarefas foi considerado o sobrecusto da implementação. A Figura 11 demonstra os resultados encontrados, o *eixo-y* representa os sobrecustos medidos para cada protocolo, ou seja, o tempo que as operações *p* e *v* implementadas levaram para executar. No *eixo-x*, a quantidade de tarefas é variada.

Pela inspeção visual da figura pode-se concluir que o protocolo MSRP em geral tem o maior sobrecusto de execução, isto ocorre devido aos loops executados em suas chamadas para calcular os tetos dos recursos e sistema. Em relação aos protocolos

singlecore, o SRP também teve maior o sobrecusto, a isto aplica-se o mesmo motivo do MSRP, pois o próprio MSRP é uma extensão do SRP. Observa-se também que para os protocolos PIP, PCP, IPCP e MPCP não existem grandes variações nos tempos medidos conforme o número de tarefas é aumentado, isto ocorre pois seu fluxo de execução nas funções *p* e *v* são basicamente os mesmos independente da quantidade de tarefas esperando. Já para protocolos que tem suas operações dependentes da quantidade de tarefas existentes como o SRP e MSRP, ocorre um aumento do tempo medido em relação a quantidade de tarefas e processadores no sistema.

Figura 11 – *Overheads* dos protocolos implementados



Fonte: Autor.

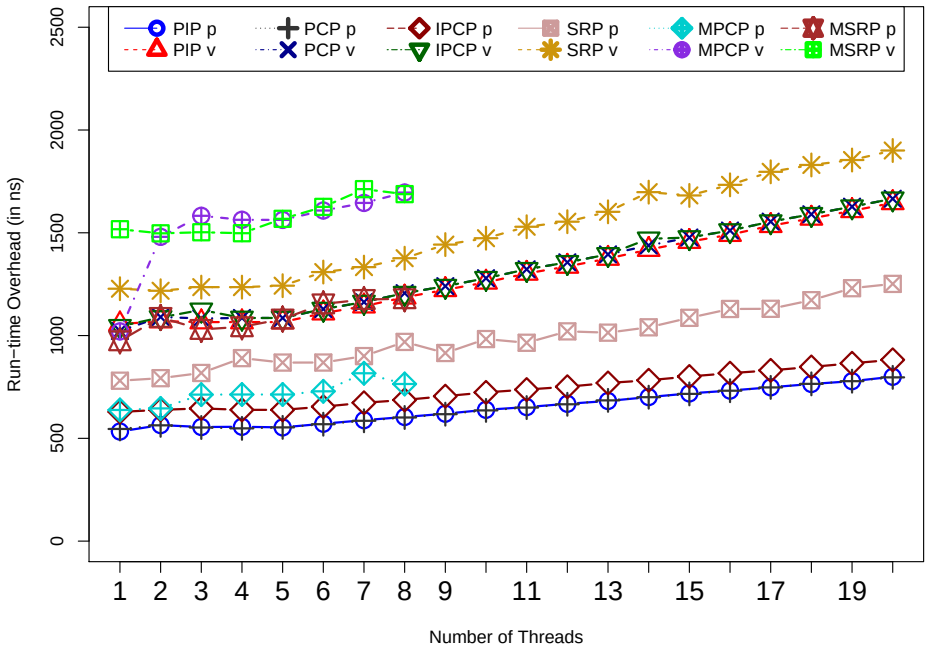
A operação *p* do protocolo MPCP curiosamente teve desempenho superior ao protocolo SRP, apesar de ser um protocolo utilizado em sistemas *multicore*, que exigem maiores restrições e parâmetros para evitar a inversão de prioridade em recursos compartilhados por mais de um processador. Isto reforça o quão intensas são as operações de atualização de teto de recursos nos protocolos SRP e MSRP. As operações *p* e *v* do MPCP com apenas um processador, ou seja, guardando uma seção crítica local, tiveram desempenho muito próximo do protocolo IPCP, conforme esperado segundo a teoria. O mesmo não pode ser dito a respeito do MSRP em relação ao SRP, esperava-se que as operações *p* e *v* do MSRP fossem mais próximas dos valores encontrados para o protocolo SRP quando $m = 1$ (um único processador). Este comportamento não ocorreu pois na criação da aplicação de teste, o semáforo do protocolo MSRP foi criado para guardar sempre seções globais, mesmo com o sistema executando em um único processador. Desta forma o MSRP utilizou seus próprios

métodos e parâmetros na execução do recurso compartilhado, levando a maiores sobrecustos.

Além da instrumentação das operações dos protocolos, outras seções de código do sistema operacional EPOS foram instrumentadas para avaliar com maior precisão o sobrecusto de uma tarefa requisitando acesso a um recurso compartilhado. Dentre estas seções estão o escalonador do sistema, responsável por escolher a próxima tarefa a ser executada, e as filas de bloqueio nos recursos, que tiveram suas operações para inserção e remoção da fila instrumentadas. Todas estas operações são utilizadas pelas classes de protocolos implementadas.

A Figura 12 exibe os sobrecustos totais medidos, somando as operações dos protocolos e operações importantes utilizadas no sistema. Nela pode-se observar que os sobrecustos das operações do sistema são superiores aos sobrecustos adicionados por meio da implementação do modelo, um passo positivo importante para a validação da implementação proposta. Neste cenário, o sobrecusto cresce conforme o número de tarefas aumenta, isto ocorre pois as tarefas bloqueadas são armazenadas nos recursos em uma lista ordenada por prioridade, fazendo com que as operações levem mais tempo conforme a quantidade de tarefas aumenta.

Figura 12 – *Overhead* total dos protocolos implementados



Fonte: Autor.

Apesar da operação *v* dos protocolos escolher como próxima tarefa para executar no recurso sempre a cabeça da lista, pois a lista é ordenada baseada na prioridade das tarefas, observa-se que o sobrecusto da operação também cresce de acordo com a quantidade de tarefas acessando o recurso. A causa deste problema foi

atribuída a construção do RTOS, em que eventos concorrentes resultam em problemas na classe *Alarm*, utilizada para invocar novas instâncias das tarefas e em operações de escalonamento, desta forma, parâmetros internos da estrutura do alarme são compartilhados em diferentes partes do sistema, levando a bloqueios condicionados.

4.3 ANÁLISE DO IMPACTO DA IMPLEMENTAÇÃO NA ESCALONABILIDADE DO SISTEMA

Para medir o impacto dos *overheads* medidos na escalonabilidade do sistema, foram realizados testes com conjuntos de tarefas gerados aleatoriamente com o auxílio da biblioteca *SchedCAT* (SCHEDCAT..., 2020). A biblioteca Python/C++ contém diversos testes de escalonabilidade para sistemas operacionais de tempo real, além de suporte a geração de conjuntos de tarefas e métodos para medição de *overheads*. Os parâmetros utilizados nos testes foram gerados conforme demonstrado a seguir.

Os períodos T_i das tarefas foram escolhidos aleatoriamente utilizando uma distribuição logarítmica uniforme no intervalo [10 ms, 100 ms] para conjuntos de tarefas de período homogêneo, ou no intervalo [1 ms, 1000 ms] para conjuntos com período heterogêneo. As utilizações U_i das tarefas foram escolhidas usando uma distribuição exponencial com diferentes médias, 0,1 (leve), 0,25 (média) ou 0,5 (alta).

As tarefas do conjunto gerado podem acessar $n_{res} \in \{1, 2, 4, 8, 16\}$ recursos compartilhados no sistema, recursos esses que permitem que apenas uma tarefa obtenha acesso simultaneamente, além de proibir acessos aninhados (quando uma tarefa tenta acessar uma seção crítica enquanto já executa em uma seção crítica). As seções críticas tem tamanho variado selecionado de maneira uniforme nos intervalos [1 μs, 25 μs] (seção *curta*), [25 μs, 100 μs] (seção *média*), ou [100 μs, 500 μs] (seção *longa*), variando assim o tempo que uma tarefa bloqueia o recurso. Para diversificar o acesso a recursos na execução das instâncias das tarefas, cada tarefa tem uma probabilidade p^{acc} de acessar cada recurso que compartilha, com $p^{acc} \in$ {0,25, 0,5, 0,75, 1}, assim, apesar de cada instância da tarefa acessar apenas um recurso compartilhado por execução, os recursos acessados variam e garantem a aleatoriedade nos testes.

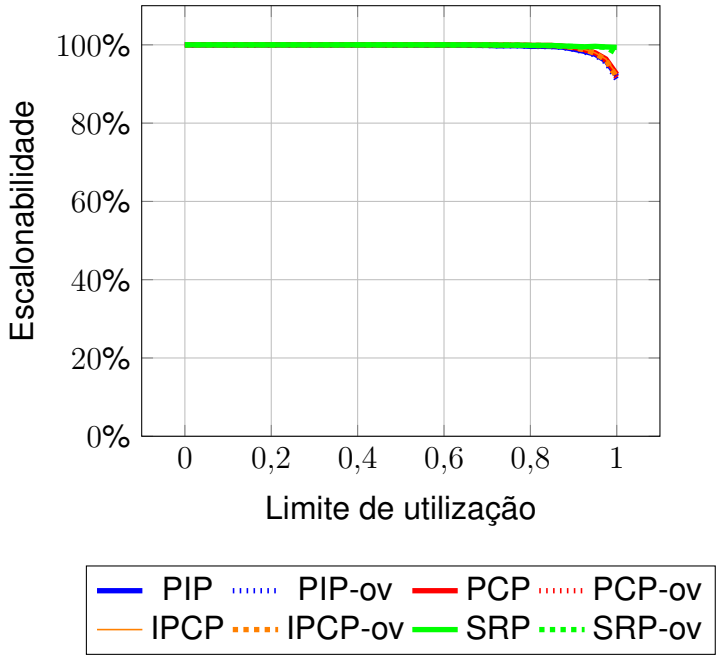
A análise de escalonabilidade foi executada para sistemas single e multiprocessadores, considerando m a quantidade de processadores, com $m \in$ {1, 2, 4, 8}. Para analisar todos os protocolos implementados, foram selecionados dois algoritmos de escalonamento, RM e EDF, de forma a abranger algoritmos de prioridade fixa e dinâmica. Foram analisados os protocolos PIP, PCP e IPCP (todos utilizando o algoritmo RM) e SRP (utilizando EDF) para sistemas *singlecore*.

Para cada combinação de parâmetros informados previamente, 1000 conjuntos de tarefas foram gerados aumentando o limite de utilização em 5% a cada nova iteração do teste. Devido a grande quantidade de diferentes combinações de parâmetros, a

seguir serão apresentados apenas os resultados mais relevantes encontrados. As Figuras 13–17 representam os casos escolhidos para demonstrar comportamentos encontrados na simulação. Nelas, o *eixo-x* do gráfico representa a variação do limite de utilização no conjunto de tarefas, e o *eixo-y* representa a taxa de escalonabilidade do conjunto, ou seja, dentre os 1000 conjuntos gerados a porcentagem que foi escalonável nas condições apresentadas.

A Figura 13 foi gerada utilizando seções críticas longas, períodos heterogêneos e utilização alta no conjunto. Termos esses que remetem aos conjuntos de valores definidos previamente no início deste capítulo. Tarefas com utilização alta implicam em menor flexibilidade, pois seu tempo de computação C_i é muito próximo de sua deadline D_i, fazendo com que a tarefa não possa ficar muito tempo bloqueada em um recurso sem que perca sua *deadline*. O gráfico da esquerda representa um cenário em que grande parte dos conjuntos gerados são escalonáveis, mesmo para maiores utilizações, isto pode ser realizado por meio das baixas probabilidades de acesso ao recurso ($p^{acc} = 0,25$ foi utilizado) e ao baixo número de recursos compartilhados utilizados ($n_{res} = 4$).

Figura 13 – Análise escalonabilidade protocolos *singlecore* para conjuntos de utilização alta, $p^{acc} = 0,25$ e $n_{res} = 4$



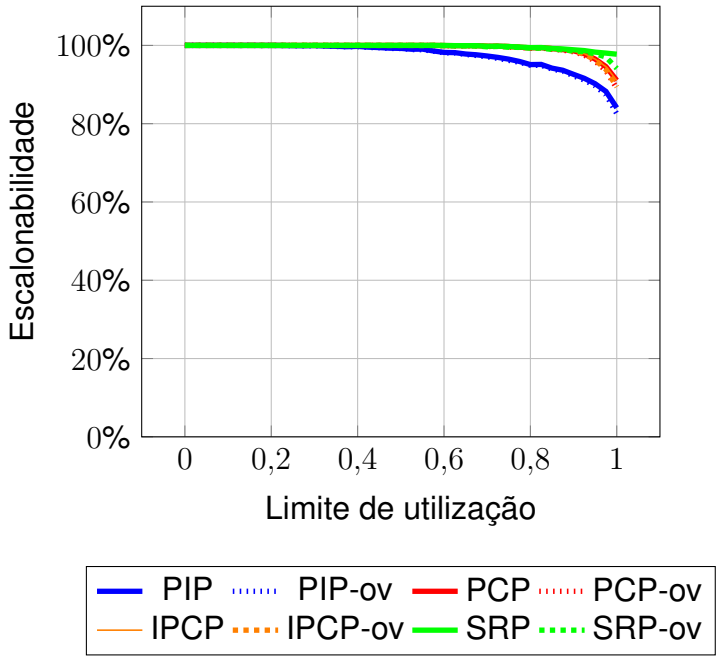
Fonte: Autor.

Neste cenário, o protocolo que teve maior taxa de escalonabilidade para maiores limites de utilização (próximo de 100%) foi o SRP. Nota-se que o sobrecusto da implementação do SRP quase não influenciou no teste, de forma que as curvas SRP e SRP-ov praticamente se sobrepõem. Os protocolos IPCP, PCP e PIP tiveram taxas de escalonabilidade muito semelhantes e inferiores ao SRP para maiores limites

de utilização.

Porém, ao aumentar o número de n_{res} para 6 e aumentar a probabilidade de acesso a recursos compartilhados para 25%, como pode ser observado na Figura 14, nota-se claramente uma diminuição da taxa de escalonabilidade para maiores limites de utilização. O protocolo que mais se degradou neste cenário foi o PIP, e o protocolo menos afetado foi o SRP, de acordo com o esperado em relação a definição e objetivos dos protocolos.

Figura 14 – Análise escalonabilidade protocolos *singlecore* para conjuntos de utilização alta, $p^{acc} = 0,5$ e $n_{res} = 6$



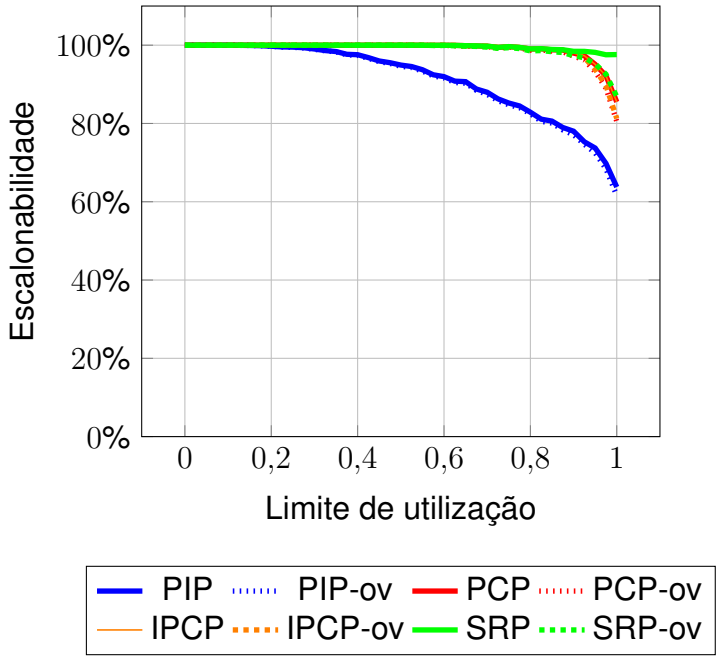
Fonte: Autor.

Importante notar que embora a utilização das tarefas sejam escolhidas aleatoriamente com diferentes valores de médias, 0,1 (baixa utilização), 0,25 (utilização média) e 0,5 (utilização alta), a utilização do conjunto de tarefas é fixa e aumenta em cada iteração do teste. Esta relação influencia diretamente os resultados, pois para conjuntos de baixa utilização são necessárias mais tarefas para atingir maiores limites de utilização, enquanto para conjuntos de alta utilização menos tarefas são necessárias. Desta forma, ao diminuir a utilização média no conjunto gerado, as taxas de escalonabilidade devem diminuir devido ao maior número de tarefas tentando acessar os recursos.

A Figura 15 representa conjuntos de tarefas gerados com seções críticas longas, períodos heterogêneos, utilização média, $p^{acc} = 0,5$ e $n_{res} = 8$. Neste cenário pode-se observar o resultado do aumento da quantidade de tarefas e número de recursos compartilhados no sistema. Em relação ao gráfico apresentado na Figura 14, a diminuição da utilização média do conjunto de tarefas faz com que o maior número de

tarefas no sistema resulte em piores taxas de escalonabilidade. O protocolo PIP teve desempenho muito inferior aos demais protocolos, que tiveram desempenho similar. O protocolo SRP novamente teve o melhor desempenho, seguido dos protocolos IPCP e PCP com resultados muito similares.

Figura 15 – Análise escalonabilidade protocolos *singlecore* para conjuntos de utilização média e $n_{res} = 8$

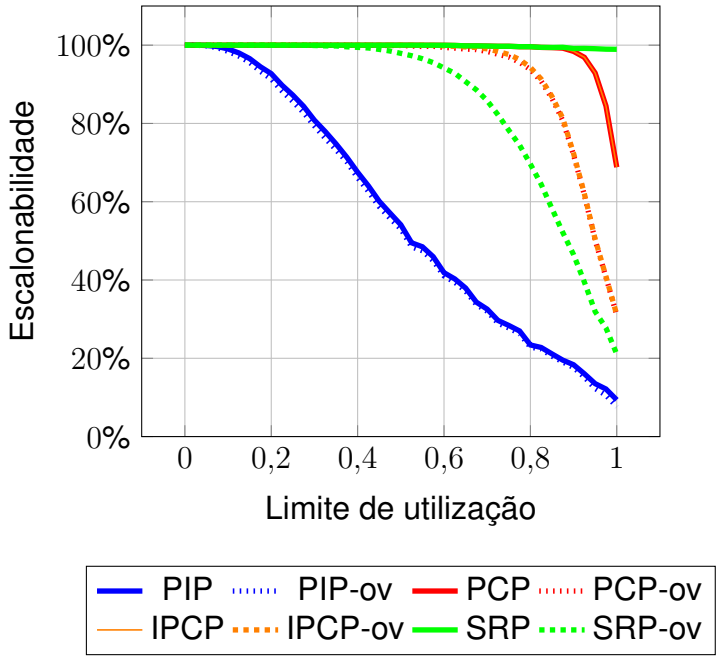


Fonte: Autor.

Na Figura 16 os efeitos colaterais na escalonabilidade do sistema ao se utilizar grande quantidade de recursos compartilhados são apresentados. Neste cenário são utilizadas tarefas com períodos heterogêneos, seções críticas longas, utilização leve, $p^{acc} = 0,75$ e $n_{res} = 16$. Seguindo os cenários anteriores, o PIP continuou sendo o protocolo com pior desempenho em sistemas *singlecore*. Além disso, curiosamente o protocolo SRP, que nos últimos casos tinha tido o melhor desempenho médio, teve pior desempenho quando comparado aos protocolos PCP e IPCP. Este comportamento está relacionado às definições das operações *p* e *v* do protocolo SRP, que calculam e atualizam os tetos de cada recurso compartilhado no sistema. Esta atualização é implementada por meio de um loop nos recursos compartilhados, desta forma o sobrecusto do protocolo é altamente relacionado a quantidade de recursos compartilhados utilizando o protocolo.

As versões particionadas dos algoritmos, P-RM e P-EDF, foram utilizadas para testes em plataformas *multicore* ($m > 1$), nas quais foram analisados os protocolos MPCP, utilizando o algoritmo P-RM, e MSRP usando o algoritmo P-EDF. A Figura 17 mostra os resultados para os conjuntos de tarefas gerados com utilização média, períodos heterogêneos, $p^{acc} = 0,75$, seções críticas curtas e $m = 8$ processadores.

Figura 16 – Análise escalonabilidade protocolos *singlecore* para conjuntos de utilização média e $n_{res} = 16$



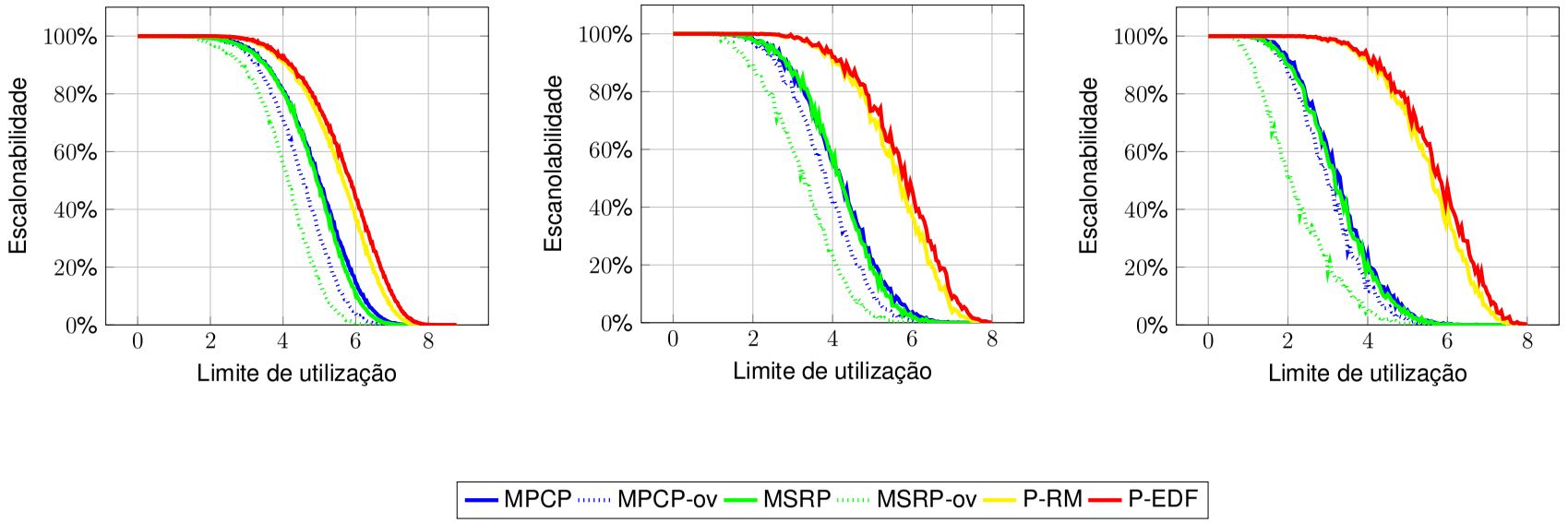
Fonte: Autor.

Para efeito de comparação, além dos protocolos MPCP e MSRP, foram adicionados aos gráficos as taxas de escalonabilidade para os algoritmos de escalonamento P-RM e P-EDF sem utilizar protocolos de recursos compartilhados.

A diferença entre os gráficos está na quantidade de recursos compartilhados por cada tarefa, $n_{res} \in \{2, 4, 8\}$. As linhas sólidas representam a taxa de escalonabilidade teórica para os protocolos e algoritmos de escalonamento, enquanto as linhas tracejadas representam a taxa de escalonabilidade adicionada dos sobrecustos da implementação medidos e apresentados anteriormente na Figura 12.

Como esperado, conforme a quantidade de recursos compartilhados pelas tarefas do conjunto é aumentada, a escalonabilidade do conjunto tende a diminuir. Isto acontece pois a quantidade de bloqueios aumenta quando mais tarefas precisam de um recurso para executar, fazendo com que a fila de espera no recurso tenha mais tarefas e levando a maiores tempos de bloqueio. Observa-se também que ao adicionar protocolos de acesso a recursos compartilhados ao sistema, ocorre uma diminuição na taxa de escalonabilidade em relação à utilização das tarefas, isto não é uma desvantagem ao uso dos protocolos, mas sim, uma consequência das garantias de propriedades garantidas pelos mesmos, como a eliminação da inversão de prioridade ilimitada e garantia contra *deadlocks*.

Figura 17 – Resultados selecionados dos testes de escalonabilidade para protocolos em sistemas *multicore*



a. $n_{res} = 2$ b. $n_{res} = 4$ c. $n_{res} = 8$

Fonte: Autor.

O protocolo MSRP teve desempenho inferior em relação ao MPCP nas simulações, isto pode ser explicado observando as definições das operações *p* e *v* na implementação do protocolo. Nas operações de requisição e liberação do recurso no protocolo MSRP são realizadas operações de iterações em vetores para calcular os tetos dos recursos e tetos do sistema. Em relação ao SRP, o sobrecusto das operações torna-se muito maior pois estas iterações ocorrem para cada processador e recurso compartilhado, como demonstrado na Figura 17 conforme n_{res} é aumentado.

4.4 CONSIDERAÇÕES PARCIAIS

Nesta seção os resultados obtidos nos testes serão discutidos para além das observações já mencionadas durante as prévias seções, abrangendo alguns pontos não abordados durante a apresentação dos dados a fim de esclarecer possíveis dúvidas e estender comentários importantes relacionados aos protocolos, como comportamentos interessantes e diferentes do esperado quando comparados a teoria, além de análises gerais relativas ao impacto dos sobrecustos medidos na escalonabilidade do sistema.

4.4.1 Desempenho protocolo IPCP vs PCP

Um sobrecusto relevante na análise de escalonabilidade dos conjuntos de tarefas é a troca de contexto, que representa o tempo que uma tarefa em execução leva para ser retirada do contexto do processador atual e dar lugar a próxima tarefa, que foi escolhida pelo escalonador e assume o contexto para iniciar sua execução. Este tempo varia entre os algoritmos de escalonamento, e a análise efetuada leva em consideração o pior caso medido (0,3 μs) (GRACIOLI et al., 2013).

O protocolo IPCP foi proposto como uma melhoria em relação ao protocolo PCP, pois ao invés de esperar com que a tarefa em execução sofra uma tentativa de preempção no recurso para aumentar sua prioridade, a prioridade da tarefa em execução é aumentada para o teto assim que consegue acesso ao recurso compartilhado. Desta forma, o protocolo IPCP diminui a quantidade de trocas de contexto das tarefas, melhorando a escalonabilidade do sistema.

Esta melhora pode ser visualizada nas análises de impacto de escalonabilidade, em que o algoritmo IPCP teve resultado ligeiramente superior ao PCP (mais conjuntos de tarefas escalonáveis), mesmo com maiores sobrecustos medidos na implementação. Ou seja, a diminuição das trocas de contexto representaram maiores ganhos em relação as instruções adicionais que o protocolo IPCP tem que realizar em todas suas chamadas de *p* e *v*.

4.4.2 Sobrecusto da operação *v* no protocolo MPCP

O protocolo que mais teve diferença entre os sobrecustos medidos para as operações *p* e *v* foi o MPCP. Esta diferença pode ser observada para valores de $m > 1$, em que o protocolo guarda recursos globais e não tem o exato mesmo funcionamento do protocolo IPCP. Com mais processadores o sobrecusto da operação *p* manteve-se próximo aos sobrecustos medidos no protocolo IPCP, devido ao fato de que mesmo guardando recursos globais, o funcionamento da operação é muito semelhante, tendo apenas que elevar a prioridade da tarefa para o teto global ou fazer uma inserção na fila de bloqueio caso o recurso esteja ocupado. Já para a operação *v*, no cenário em que a tarefa em execução libera o recurso para uma tarefa bloqueada na fila, a próxima tarefa será obrigatoriamente de outro processador, fazendo com que o protocolo tenha que adquirir a referência de uma tarefa em um diferente processador e salvá-la como dona do recurso, o que aumenta significativamente o *overhead* da operação nesses casos.

4.4.3 Bloqueio de tarefas nos protocolos SRP e MSRP

Nos protocolos SRP e MSRP as tarefas teoricamente nunca deveriam ser bloqueadas em uma fila de recurso, apenas no instante inicial de sua invocação, para que a partir do momento em que sejam escolhidas pelo escalonador do sistema, executem sem bloqueios em sistemas *hard real-time*. No entanto, a implementação dos testes associou um mesmo nível de preempção para todas as tarefas do sistema, para que as tarefas sejam aprovadas no teste do escalonador mais facilmente e sejam bloqueadas no recurso, a fim de comparar os *overheads* das operações com os outros protocolos.

Isto não configura um problema, pois os protocolos permitem que tarefas tenham níveis de preempção iguais, e até mesmo definem as filas que devem armazenar tarefas bloqueadas nos recursos, com a definição do MSRP explicitamente indicando que uma FIFO deve ser utilizada para este propósito. Assim, apenas reitera-se que o cenário neste caso é mais próximo de conjuntos de tarefas em sistemas *soft real-time* que utilizam tipos diferentes de tarefas, em que tarefas que tenham o mesmo nível de preempção ainda possam ser bloqueadas no recurso, embora não seja recomendado para tarefas de maiores prioridades.

4.4.4 Proteção de recursos locais utilizando MSRP

Nas análises de *overhead* nas Figuras 11–12 o protocolo MSRP não obteve o comportamento esperado em recursos locais, pois esperava-se que seu *overhead* fosse semelhante ao do protocolo SRP em cenários de sistemas *singlecore*, conforme a definição do protocolo. Isto ocorreu pois a aplicação de teste utilizada criou uma

instância da classe `Semaphore_MSRP<true>` para qualquer número de processadores, ou seja, foi criada uma instância para guardar recursos globais mesmo em $m = 1$, caso que resultaria em um recurso local.

Para solucionar este problema, uma solução simples e funcional seria utilizar a classe `Semaphore_MSRP<false>` para sistemas com apenas um processador (utilizada para guardar recursos locais), ou ainda criar uma instância da classe `Semaphore_SRP` diretamente, já que não faz sentido utilizar protocolos *multicore* em aplicações *singlecore*. O aumento do *overhead* no protocolo MSRP com um processador ocorre devido aos métodos *updateCeiling* da classe terem comandos adicionais ligados ao funcionamento *multicore*.

4.4.5 Observações das análises de impacto na escalonabilidade

Nas medições de sobrecusto um dos protocolos *single-core* que se destacou em sua implementação foi o PIP, apesar disso, o protocolo demonstrou ser altamente sensível a sobrecustos, e seu bom desempenho não se comprovou nas análises de escalonabilidade, em que o protocolo teve o pior desempenho, significativamente inferior aos demais protocolos *single-core*.

Por meio das análises de escalonabilidade também comprovou-se que conjuntos de tarefas com utilizações medianas tem maiores taxas de escalonabilidade. Isto ocorre pois é necessária uma quantidade menor de tarefas para atingir a utilização do sistema quando comparados a conjuntos com baixa utilização, em que existem muitas tarefas tentando acessar recursos compartilhados.

Verificou-se que mesmo com melhor escalonabilidade, o protocolo MSRP pode ter desempenho inferior ao MPCP devido ao grande sobrecusto decorrente da implementação. Além disso, em conjuntos de tarefas de baixa utilização o tamanho da seção crítica se mostrou fator relevante, impactando negativamente a escalonabilidade.

5 CONCLUSÕES

Considerando os objetivos iniciais de unificar e padronizar implementações de protocolos de acesso a recurso compartilhado em sistemas operacionais de tempo real, um modelo foi proposto capaz de atingir os objetivos previamente estabelecidos, a construção de um software facilmente estendível e reutilizável em diferentes sistemas operacionais de tempo real. O modelo foi então posto a prova numericamente, de forma a avaliar sua leveza e viabilidade por meio de medições de rastro de memória e sobrecusto na implementação. Por fim, utilizando os dados de sobrecusto medidos, uma análise de escalonabilidade foi executada de forma a medir o impacto da implementação na escalonabilidade de conjuntos de tarefas gerados.

O uso de boas práticas de programação e de uma linguagem orientada à objetos como o C++ se provaram essenciais durante o processo de modelagem da solução proposta, de forma a garantir a fácil extensão do modelo para protocolos adicionais. A portabilidade do modelo para diferentes sistemas operacionais de tempo real que utilizem linguagens orientadas à objeto podem ocorrer sem muitos problemas, caso o RTOS alvo tenha uma estrutura de classes bem definida em torno de módulos como escalonadores (provendo escalonadores RM e EDF, por exemplo) e semáforos. No entanto, reforça-se que a grande maioria dos sistemas operacionais de tempo real são implementados utilizando a linguagem C, motivo este inclusive, que leva a falta de padronização e modelos na implementação de semáforos nesses sistemas, e por consequência, gera a motivação deste trabalho. Porém, implementar tal modelo em um RTOS escrito em C não é impossível, muito pelo contrário, utilizando estruturas e ponteiros para funções muitos sistemas operacionais simulam orientação à objeto, como por exemplo o kernel Linux.

Os resultados das análises de escalonabilidade demonstraram que os sobrecustos na implementação de protocolos de acesso a recursos compartilhados não são negligenciáveis, especialmente quando a utilização do sistema está próxima de seu limite teórico e o número de recursos compartilhados é alto. No entanto, para a grande maioria dos casos, o design e implementação do trabalho não impacta negativamente a escalonabilidade do sistema. De forma geral, os resultados das análises aplicadas ao modelo proposto foram considerados positivos, validando o modelo e incentivando seu uso em diferentes sistemas operacionais de tempo real, tanto em aplicações *single*, como *multicore*.

Em relação a possível continuidade do trabalho, fica como sugestão a extensão do modelo proposta na Seção 3.4, por meio da finalização da implementação do protocolo MrsP, e junto dele, a adição de suporte a *spinlocks* no sistema, enriquecendo o

modelo e fornecendo outras opções ao usuário programador além do uso de semáforos para garantir a exclusão mútua de recursos compartilhados. Também sugere-se a reavaliação do sobrecusto medido para o protocolo MSRP para sistemas *singlecore*, de forma a garantir que seu comportamento realmente se assemelhe ao SRP em sistemas com apenas um processador. Além disso, o código das implementações como um todo pode ser revisado, buscando a máxima otimização em *loops*, de forma a garantir ainda menores sobrecustos na execução das operações dos protocolos.

REFERÊNCIAS

BAKER, T. P. Stack-based scheduling of real-time processes. **Real-Time Systems**, [S. l.], v. 3, n. 1, p. 67–99, mar. 1991.

BURNS, A.; WELLINGS, A. **Real-Time Systems and Programming Languages**: Ada 95, real-time java and real-time posix. 3. ed. United States: Addison Wesley, 2001.

BURNS, A.; WELLINGS, A. J. A schedulability compatible multiprocessor resource sharing protocol – mrsp. In: **Proceedings** of the 25th Euromicro Conference on Real-Time Systems. France: IEEE, 2013. p. 282–291. Disponível em: https://ieeexplore.ieee.org/document/6602108. Acesso em: 6 jun. 2022.

BUTTAZZO, G. C. **Hard real-time computing systems**: predictable scheduling algorithms and applications. 3. ed. United States: Springer, 2011.

CALANDRINO, J. M. et al. Litmus^rt: a testbed for empirically comparing real-time multiprocessor schedulers. In: **Proceedings** of the 27th IEEE INTERNATIONAL REAL-TIME SYSTEMS SYMPOSIUM (RTSS'06). Rio de Janeiro, Brazil: IEEE, 2006. p. 111–126. Disponível em: https://ieeexplore.ieee.org/document/4032341. Acesso em: 6 set. 2021.

DAVIS, R. I.; ZABOS, A.; BURNS, A. Efficient exact schedulability tests for fixed priority real-time systems. **IEEE Transactions on Computers**, [S. l.], v. 57, n. 9, p. 1261–1276, set. 2008.

DIJKSTRA, E. Co-operating sequential processes. In: **Proceedings** of the Programming languages : NATO Advanced Study Institute : lectures given at a three weeks Summer School held in Villard-le-Lans, 1966 / ed. by F. Genuys. United States: Academic Press Inc., 1968. p. 43–112. Disponível em: https://doi.org/10.1007/978-1-4757-3472-0_2. Acesso em: 2 jul. 2022.

EPOS. **Embedded parallel operating system**. 2022. Disponível em: https://epos.lisha.ufsc.br. Acesso em: 26 abr. 2022.

GAI, P.; LIPARI, G.; NATALE, M. D. Minimizing memory utilization of real-time task sets in single and multi-processor systems-on-a-chip. In: **Proceedings** of the 22nd IEEE REAL-TIME SYSTEMS SYMPOSIUM. England: IEEE, 2001. p. 73–83. Disponível em: https://ieeexplore.ieee.org/document/990598. Acesso em: 11 mar. 2022.

GRACIOLI, G. **Real-time operating system support for multicore applications**. Tese (Doutorado) — Universidade Federal de Santa Catarina, Florianópolis, 2014.

GRACIOLI, G. et al. Implementation and evaluation of global and partitioned scheduling in a real-time os. **Real-Time Systems**, [S. l.], v. 49, n. 6, p. 669–714, nov. 2013.

HAMBARDE, P.; VARMA, R.; JHA, S. The survey of real time operating system: RTOS. In: **Proceedings** of the INTERNATIONAL CONFERENCE ON ELECTRONIC SYSTEMS, SIGNAL PROCESSING AND COMPUTING TECHNOLOGIES. India: IEEE, 2014. p. 34–39. Disponível em: https://ieeexplore.ieee.org/document/6745342. Acesso em: 6 set. 2021.

KRUSE, R. L.; RYBA, A. J. **Data Structures and Program Design in C++**. United States: Prentice-Hall, Inc., 1998.

LIU, C. L.; LAYLAND, J. W. Scheduling algorithms for multiprogramming in a hard-real-time environment. **Journal of the Association for Computing Machinery**, NY, USA, v. 20, n. 1, p. 46–61, jan. 1973.

LIU, J. **Real-time systems**. United States: Pearson, 2000.

RAJKUMAR, R. Real-time synchronization protocols for shared memory multiprocessors. In: **Proceedings** of the 10th INTERNATIONAL CONFERENCE ON DISTRIBUTED COMPUTING SYSTEMS. Paris, France: IEEE, 1990. p. 116–123. Disponível em: https://ieeexplore.ieee.org/document/89257. Acesso em: 6 set. 2021.

SCHEDCAT: Schedulability test collection and toolkit. 2020. Disponível em: https://github.com/brandenburg/schedcat. Acesso em: 12 jun. 2022.

SHA, L.; RAJKUMAR, R.; LEHOCZKY, J. P. Priority inheritance protocols: an approach to real-time synchronization. **IEEE Transactions on Computers**, [S. l.], v. 39, n. 9, p. 1175–1185, set. 1990.

TANENBAUM, A. S.; BOS, H. **Modern Operating Systems**. 4. ed. United States: Pearson, 2014.

WIEDER, A.; BRANDENBURG, B. B. On spin locks in autosar: blocking analysis of fifo, unordered, and priority-ordered spin locks. In: **Proceedings** of the 34th IEEE REAL-TIME SYSTEMS SYMPOSIUM. Vancouver, Canada: IEEE, 2013. p. 45–56. Disponível em: https://ieeexplore.ieee.org/document/6728860. Acesso em: 6 set. 2021.

XU, J.; PARNAS, D. L. Scheduling processes with release times, deadlines, precedence and exclusion relations. **IEEE Transactions on Computers**, [S. l.], v. 16, n. 3, p. 360–369, mar. 1990.

YANG, M.; WIEDER, A.; BRANDENBURG, B. Global real-time semaphore protocols: a survey, unified analysis, and comparison. In: **Proceedings** of the 2015 IEEE REAL-TIME SYSTEMS SYMPOSIUM. Texas, United States: IEEE, 2015. p. 1–12. Disponível em: https://ieeexplore.ieee.org/document/7383559. Acesso em: 6 set. 2021.

APÊNDICE A - MODIFICAÇÕES NO SISTEMA OPERACIONAL DE TEMPO REAL EPOS

```
1  class Synchronizer_Base
2  {
3  protected:
4      Synchronizer_Base() {}
5      ~Synchronizer_Base() {}
6
7      // Atomic operations
8      bool tsl(volatile bool & lock) { return CPU::tsl(lock); }
9      int finc(volatile int & number) { return CPU::finc(number); }
10     int fdec(volatile int & number) { return CPU::fdec(number); }
11
12     // Thread operations
13     void begin_atomic() { Thread::lock(); }
14     void end_atomic() { Thread::unlock(); }
15 };
16
17 template<bool>
18 class Synchronizer_Common : public Synchronizer_Base
19 {
20 protected:
21     // List ordered by priority
22     typedef Thread::Thread_Queue Queue;
23
24     Synchronizer_Common() {}
25
26     ~Synchronizer_Common() {
27         begin_atomic();
28         wakeup_all();
29     }
30
31     void sleep() { Thread::sleep(&_queue); }
32     void wakeup() { Thread::wakeup(&_queue); }
33     void wakeup_all() { Thread::wakeup_all(&_queue); }
34
35 protected:
36     Queue _queue;
37 };
38
39 template<>
40 class Synchronizer_Common<false> : public Synchronizer_Base
41 {
```

```
42 protected:
43     typedef Thread::FIFO_Queue Queue;
44 
45     Synchronizer_Common() {}
46 
47     ~Synchronizer_Common() {
48         begin_atomic();
49         wakeup_all();
50     }
51 
52     void sleep() { Thread::sleep(&_queue); }
53     void wakeup() { Thread::wakeup(&_queue); }
54     void wakeup_all() { Thread::wakeup_all(&_queue); }
55 
56 protected:
57     Queue _queue;
58 };
59 
60 template<bool T>
61 class Semaphore_Template: public Synchronizer_Common<T> {
62 private:
63     typedef Synchronizer_Common<T> Base;
64 public:
65     Semaphore_Template(int v = 1) : _value(v) {}
66 
67     void p(bool locked = false) {
68         if(!locked)
69           Base::begin_atomic();
70 
71         if(Base::fdec(_value) < 1) {
72             Base::sleep(); // implicit end_atomic()
73         }
74         else {
75             if(!locked)
76                 Base::end_atomic();
77         }
78     }
79     void v(bool locked = false) {
80         if(!locked)
81           Base::begin_atomic();
82 
83         if(Base::finc(_value) < 0) {
84             Base::wakeup();  // implicit end_atomic()
85         }
86         else {
87             if(!locked)
88                 Base::end_atomic();
```

```
89          }
90      }
91
92  protected:
93      volatile int _value;
94  };
95
96  template<bool T>
97  class Semaphore_RT: public Semaphore_Template<T> {
98  private:
99      typedef Semaphore_Template<T> Base;
100
101 public:
102     Semaphore_RT(int v = 1):
103         Semaphore_Template<T>(v), _owner(0), _priority(0){}
104
105     typedef Thread Thread_t;
106     typedef int Priority_t;
107
108     Thread_t* owner() { return _owner; }
109     void owner(Thread_t* own) { _owner = own; }
110
111     Priority_t priority() { return _priority; }
112     void priority(Priority_t priority) { _priority = priority; }
113
114 protected:
115     static Thread_t* currentThread() {
116         return( reinterpret_cast<Thread_t *>(Thread::self()) );
117     }
118
119     Thread_t* nextThread() {
120         if(Base::_queue.empty())
121             return 0;
122         else {
123             return reinterpret_cast< Thread_t *>
124                 ( Base::_queue.head()->object() );
125         }
126     }
127
128 private:
129     Thread_t *_owner;
130     Priority_t _priority;
131 };
132
133 template<bool T>
134 class Semaphore_Ceiling: public Semaphore_RT<T> {
135 private:
```

```
136     typedef Semaphore_RT<T> Base;
137
138 public:
139     Semaphore_Ceiling(Priority_t ceiling, int value = 1):
140         Semaphore_RT<T>(value), _ceiling(ceiling) {}
141
142     Priority_t ceiling() { return _ceiling; }
143     void ceiling(Priority_t ceiling) { _ceiling = ceiling; }
144
145 private:
146     Priority_t _ceiling;
147 };
148
149 class Semaphore_PIP: protected Semaphore_RT<true> {
150
151 public:
152     Semaphore_PIP(int value = 1): Semaphore_RT<true>(value) {}
153
154     void p() {
155         begin_atomic();
156         if( !owner() ) {
157             owner( currentThread() );
158             priority( owner()->priority() );
159         }
160         else {
161         Thread_t * aux_current = currentThread();
162             if( aux_current->priority() < owner()->priority() )
163                 owner()->setPriority( aux_current->priority() );
164         }
165         Semaphore_RT::p(true);
166         end_atomic();
167     }
168
169     void v() {
170         begin_atomic();
171         if(owner() == currentThread()) {
172             currentThread()->setPriority( priority() );
173             Thread_t * next = nextThread();
174
175             if(next) {
176                 owner( next );
177                 priority( owner()->priority() );
178             }
179             else {
180                 owner(0);
181                 priority(0);
182             }
```

```
183         }
184         Semaphore_RT::v(true);
185         end_atomic();
186     }
187 };
188
189 class Semaphore_PCP: protected Semaphore_Ceiling<true> {
190 public:
191     Semaphore_PCP(Priority_t ceiling, int value = 1):
192         Semaphore_Ceiling(ceiling, value) {}
193
194     void p() {
195         begin_atomic();
196         if( !owner() ) {
197             owner( currentThread() );
198             priority( owner()->priority() );
199         }
200         else {
201             if( ceiling() < owner()->priority() )
202                 owner()->setPriority( ceiling() );
203         }
204         Semaphore_RT::p(true);
205         end_atomic();
206     }
207
208     void v() {
209         begin_atomic();
210         if(owner() == currentThread()) {
211             owner()->setPriority( priority() );
212             Thread_t * next = nextThread();
213
214             if(next) {
215                 owner(next);
216                 priority( owner()->priority() );
217             }
218             else {
219                 owner(0);
220                 priority(0);
221             }
222         }
223         Semaphore_RT::v(true);
224         end_atomic();
225     }
226 };
227
228 class Semaphore_IPCP: protected Semaphore_Ceiling<true> {
229 public:
```

```
230     Semaphore_IPCP(Priority_t ceiling, int value = 1):
231         Semaphore_Ceiling(ceiling, value) {}
232 
233     void p() {
234         begin_atomic();
235 
236         if(!owner()) {
237             owner( currentThread() );
238             priority( owner()->priority() );
239             owner()->setPriority( ceiling() );
240         }
241 
242         Semaphore_RT::p(true);
243         end_atomic();
244     }
245 
246     void v() {
247         begin_atomic();
248 
249         if(owner() == currentThread()) {
250             owner()->setPriority( priority() );
251             Thread_t * next = nextThread();
252 
253             if(next) {
254                 owner(next);
255                 priority( owner()->priority() );
256                 owner()->setPriority( ceiling() );
257             }
258             else {
259                 owner(0);
260                 priority(0);
261             }
262         }
263         Semaphore_RT::v(true);
264         end_atomic();
265     }
266 };
267 
268 template<bool T>
269 class Semaphore_MPCP: protected Semaphore_IPCP {
270 public:
271     Semaphore_MPCP(Priority_t ceiling = 0, int value = 1):
272         Semaphore_IPCP( ceiling, value ) {}
273 
274     void toCeiling() {
275         priority( owner()->priority() );
276         owner()->setPriority( toGlobalCeiling() );
```

```
277     }
278
279     /* Ordenates global ceilings using task base priority */
280     int divider(int number) {
281         int count = 0, ans = 1, k = 0;
282         for(int num=number; num!=0; num=num/10) {
283             if(num%10 != 0)
284                 k+=1;
285             count++;
286         }
287         count = count - 3 + k;
288         for(int n=1; n <= count; n++)
289             ans = ans*10;
290         return ans;
291     }
292
293     Priority_t toGlobalCeiling() {
294         return _pg - ( priority()/divider(priority()) );
295     }
296
297     void p() {
298         begin_atomic();
299         if( !owner() ) {
300             owner( currentThread() );
301             toCeiling();
302         }
303         Semaphore_RT::p(true);
304         end_atomic();
305     }
306
307     void v() {
308         begin_atomic();
309         if(owner() == currentThread()) {
310             owner()->setPriority( priority() );
311             Thread * next = nextThread();
312
313             if(next) {
314                 owner(next);
315                 toCeiling();
316             }
317             else {
318                 owner(0);
319                 priority(0);
320             }
321         }
322         Semaphore_RT::v(true);
323         end_atomic();
```

```
324     }
325
326 private:
327     /*
328         @brief _pg: highestPriority in the system that uses global
    shared resources + 1
329     */
330     static const unsigned int _pg {
331         Traits<Semaphore_MPCP>::highest_priority - 1
332     };
333 };
334
335 template<> // Template specialization from local critical sections
336 class Semaphore_MPCP<false>: protected Semaphore_IPCP {
337 public:
338     Semaphore_MPCP(Priority_t ceiling = 0, int value = 1):
339         Semaphore_IPCP(ceiling, value) {}
340
341     void toCeiling() { Semaphore_IPCP::toCeiling(); }
342
343     void p() { Semaphore_IPCP::p(); }
344     void v() { Semaphore_IPCP::v(); }
345 };
346
347 template<bool T = true>
348 class Semaphore_SRP: protected Semaphore_Ceiling<T> {
349 private:
350     typedef Semaphore_Ceiling<T> Base;
351
352 public:
353     static const int MAX_TASKS = 20;
354     static const int MAX_RESOURCES = 8;
355     static const int DEFAULT_CEILING = -2147483647; // minimum 32-bit
    int
356
357     Semaphore_SRP(Thread ** tasks, int * levels,
358         int n_tasks, int value = 1): Semaphore_Ceiling<T>(value)
359     {
360         for(int i = 0; i < n_tasks; i++) {
361             _tasks[i] = tasks[i];
362             _tasks[i]->criterion().preempt_level = levels[i];
363             _prptLevels[i] = levels[i];
364         }
365
366         _nt = n_tasks;
367         begin_atomic();
368
```

```
369         if(_resources) {
370             _resources[_nr] = this;
371             _nr++;
372         }
373         end_atomic();
374     }
375 
376     virtual ~Semaphore_SRP() { _resources[--_nr] = 0; }
377 
378     void p() {
379         begin_atomic();
380         Base::p(true);
381         updateCeiling();
382         end_atomic();
383     }
384 
385     void v() {
386         begin_atomic();
387         Base::v(true);
388         updateCeiling();
389         end_atomic();
390     }
391 
392     int ceiling() const { return _ceiling; }
393     static int systemCeiling() { return _system_ceiling; }
394 
395 protected:
396     typename Base::Thread_t * _tasks[MAX_TASKS];
397     int _prptLevels[MAX_TASKS];
398     int _nt;
399     int _ceiling;
400     static Semaphore_SRP* _resources[MAX_RESOURCES];
401     static int _nr;
402     static int _system_ceiling;
403 
404     void updateCeiling() {
405         int maxCeil = DEFAULT_CEILING;
406 
407         for(int i = 0; i < _nt; i++) {
408             // If not enough resource and task!=RUNNING, thread blocks
409             if(Base::_value < 1 && _tasks[i]->state() !=
410                 Thread::State::RUNNING && _prptLevels[i] > maxCeil)
411                     maxCeil = _prptLevels[i];
412         }
413         _ceiling = maxCeil;
414         updateSystemCeiling();
415     }
```

```
416
417     static void updateSystemCeiling() {
418         if(!_nr) {
419             _system_ceiling = DEFAULT_CEILING;
420             return;
421         }
422
423         _system_ceiling = _resources[0]->_ceiling;
424
425         for(int i = 1; i < _nr; i++) {
426             if(_resources[i]->_ceiling > _system_ceiling)
427                 _system_ceiling = _resources[i]->_ceiling;
428         }
429     }
430 };
431
432 bool Scheduling_Criteria::RT_Common::Elector_SRP::eligible(
433     const Scheduling_Criteria::RT_Common * criterion) {
434       /* Scheduler tries to choose idle or main threads */
435       if( criterion->_priority == IDLE || criterion->_priority == MAIN )
436         return true;
437
438       /* Main or idle threads running */
439       if( Thread::self()->preemptLevel() == 0 )
440         return true;
441
442       return ( criterion->preempt_level >
443           Semaphore_SRP<true>::systemCeiling() );
444 }
445
446 template<bool T>
447 class Semaphore_MSRP : protected Semaphore_SRP<!T> {
448 private:
449     typedef Semaphore_SRP<!T> Base;
450
451 public:
452     Semaphore_MSRP(Thread ** tasks, int * levels,
453         int n_tasks, int value = 1):
454         Base(tasks, levels, n_tasks, value) {}
455
456     void p() { Base::p(); }
457     void v() { Base::v(); }
458 };
459
460 /* For global critical sections MSRP will use Semaphore_SRP<false> no
      matter what */
461 template<>
```

```
462 class Semaphore_MSRP<true>: protected Semaphore_SRP<false> {
463 private:
464     typedef Semaphore_SRP<false> Base;
465 public:
466     Semaphore_MSRP(Thread ** tasks, int * levels,
467         int n_tasks, int value = 1):
468         Semaphore_SRP<false>() {
469
470         for(int i = 0; i < n_tasks; i++) {
471             _tasks[i] = tasks[i];
472             _tasks[i]->criterion().preempt_level = levels[i];
473             _prptLevels[i] = levels[i];
474         }
475
476         _nt = n_tasks;
477         begin_atomic();
478
479         if(_globalResources) {
480             _globalResources[_nGR] = this;
481             _nGR++;
482         }
483         end_atomic();
484     }
485
486     ~Semaphore_MSRP() { _globalResources[--_nGR] = 0; }
487
488     void p() {
489       begin_atomic();
490       Semaphore_RT<false>::p(true);
491       updateCeiling();
492       end_atomic();
493     }
494
495     void v() {
496       begin_atomic();
497       Semaphore_RT<false>::v(true);
498       updateCeiling();
499       end_atomic();
500     }
501
502     static int systemCeiling(int cpu){ return _systemCeiling[cpu]; }
503
504 protected:
505     static const int _cpu = Traits<Build>::CPUS;
506     static int _systemCeiling[_cpu];
507     static Semaphore_MSRP<true> * _globalResources[MAX_RESOURCES];
508     static int _nGR; /* Number of current global resources actives */
```

```
509
510     int _globalCeiling[_cpu];
511
512     void updateCeiling(){
513         int maxCeil;
514
515         for(int k = 0; k < _cpu; k++) {
516             maxCeil = DEFAULT_CEILING;
517
518             for(int i = 0; i < _nt; i++) {
519                 /* Compare only tasks from the same core */
520                 if( (int)_tasks[i]->criterion().queue() == k) {
521                     if( (_value < 1 && _tasks[i]->state() !=
522                         Thread::State::RUNNING)
523                         && _prptLevels[i] > maxCeil)
524                             /* Block condition */
525                             maxCeil = _prptLevels[i];
526                 }
527             }
528             _globalCeiling[k] = maxCeil;
529         }
530         updateSystemCeiling();
531     }
532
533     /* This was simplified because updateCeiling already take cares of
    each core ceiling iteration, this wasnt true for SRP */
534     static void updateSystemCeiling(){
535         /* SystemCeiling for each core */
536         for(int k = 0; k < _cpu; k++) {
537
538             if(!_nGR) {
539                 _systemCeiling[k] = DEFAULT_CEILING;
540                 return;
541             }
542
543             _systemCeiling[k] = _globalResources[0]->_globalCeiling[k];
544
545             /* Get the highest preemptLevel between all resources for
    everycore */
546             for(int i = 1; i < _nGR; i++) {
547                 if(_globalResources[i]->_globalCeiling[k] >
    _systemCeiling[k])
548                     _systemCeiling[k] = _globalResources[i]->
    _globalCeiling[k];
549             }
550         }
551     }
```

```
552 };
553
554 bool Scheduling_Criteria::RT_Common::Elector_MSRP::eligible(
555     const Scheduling_Criteria::RT_Common * criterion)
556 {
557   /* Scheduler tries to choose idle or main threads */
558   if( criterion->_priority == IDLE || criterion->_priority == MAIN )
559     return true;
560
561   /* Main or idle threads running */
562   if( Thread::self()->preemptLevel() == 0 )
563     return true;
564
565   return (criterion->preempt_level >
566       Semaphore_MSRP::systemCeiling(criterion->queue()) );
567 }
568
569 // Instrumented timer
570 class ITimer {
571 public:
572     ITimer() {
573         _startTime = timer.time_stamp();
574     }
575
576     void stop(const char * what, void * where) {
577         ITimer::_name[_dataIterator] = what;
578         ITimer::_ticks[_dataIterator] = timer.time_stamp() - _startTime;
579         ITimer::_dataIterator++;
580     }
581
582     void printData() {
583         for(int k = 0; k < _dataIterator; k++)
584         {
585             kout << "|| "<< k << " || " << ITimer::_name[k] << " || "
586                  << ITimer::_ticks[k] << " ||\n";
587         }
588     }
589
590     void cleanData() {
591         for(int k = 0; k < _dataIterator; k++)
592         {
593             ITimer::_name[k] = 0;
594             ITimer::_ticks[k] = 0;
595         }
596         _dataIterator = 0;
597     }
598
```

```
599 private:
600     typedef unsigned long long Time_Stamp;
601 
602     TSC timer;
603     Time_Stamp _startTime;
604 
605     static int _dataIterator; /* data_t container */
606     static const int MAX = 160000;
607 
608     static const char* _name[MAX];
609     static Time_Stamp _ticks[MAX];
610 };
611 
612 int ITimer::_dataIterator = 0;
613 ITimer::Time_Stamp ITimer::_ticks[MAX];
614 const char * ITimer::_name[MAX];
```