PGCOMP - Programa de Pós-Graduação em Ciência da Computação
Universidade Federal da Bahia (UFBA)
Av. Milton Santos, s/n - Ondina
Salvador, BA, Brasil, 40170-110

https://pgcomp.ufba.br
pgcomp@ufba.br

Os sistemas de tempo real estão presentes em diversas aplicações, desde aplicações críticas, como a automotiva e aviação, até as não críticas como multimídia e jogos *on-line*. Para que os sistemas de tempo real críticos funcionem corretamente, é necessário garantir que todas as suas tarefas executem dentro de prazos (*deadlines*) pré-definidos. Para tanto, o escalonador de tarefas do sistema tem papel fundamental, pois determina a cada instante qual tarefa deve executar para que todas possam cumprir seus prazos. Em sistemas com um processador, o escalonador *Earliest Deadline First* (EDF) é uma das melhores opções, pois garante que todas as tarefas cumprem seus *deadlines* sempre que isso é possível. Em sistemas com mais de um processador, o mesmo não é verdade. Este trabalho propõe o escalonador *Partitioning and Server Shadowing Algorithm* (PSSA) para sistemas multiprocessados compostos de tarefas periódicas e independentes. A ideia do algoritmo é particionar as tarefas entre os processadores para então utilizar o escalonamento EDF em cada processador. Quando tal particionamento não é encontrado, alguma tarefa deve executar em mais de um processador, enquanto alguns processadores são deixados com capacidades ociosas. A fim de gerenciar execução de tarefas que executam em mais de um processador, o PSSA utiliza uma nova abordagem denominada *server shadowing*, de acordo com a qual as partes ociosas dos processadores reais são utilizadas para criar processadores lógicos. Quando uma tarefa executa em um processador lógico, na verdade ela executa em algum dos processadores reais que forneceram sua capacidade ociosa ao processador lógico onde a tarefa está alocada. Os conceitos de *server shadowing* e processadores lógicos deram ao PSSA a capacidade de obter resultados similares aos melhores resultados publicados até hoje em termos de quantidade de migrações e preempções geradas para tarefas periódicas, como demonstrado através de extensivas simulações.

Palavras-chave: Tempo real, Escalonamento, Multiprocessadores, Servidor, *Server Shadowing*

MSC | 184 | 2024

PSSA: Um algoritmo semi-particionado com *server shadowing* para o escalonamento de tarefas de tempo real periódicas em múltiplos processadores

João Victor Alves Barreto

UFBA

PSSA: Um algoritmo semi-particionado com *server shadowing* para o escalonamento de tarefas de tempo real periódicas em múltiplos processadores

João Victor Alves Barreto

Dissertação de Mestrado

Universidade Federal da Bahia

Programa de Pós-Graduação em
Ciência da Computação

Setembro | 2024

Universidade Federal da Bahia
Instituto de Computação

Programa de Pós-Graduação em Ciência da Computação

PSSA: UM ALGORITMO SEMI-PARTICIONADO COM *SERVER SHADOWING* PARA O ESCALONAMENTO DE TAREFAS DE TEMPO REAL PERIÓDICAS EM MÚLTIPLOS PROCESSADORES

João Victor Alves Barreto

DISSERTAÇÃO DE MESTRADO

Salvador
10 de setembro de 2024

JOÃO VICTOR ALVES BARRETO

PSSA: UM ALGORITMO SEMI-PARTICIONADO COM *SERVER SHADOWING* PARA O ESCALONAMENTO DE TAREFAS DE TEMPO REAL PERIÓDICAS EM MÚLTIPLOS PROCESSADORES

Esta Dissertação de Mestrado foi apresentada ao Programa de Pós-Graduação em Ciência da Computação da Universidade Federal da Bahia, como requisito parcial para obtenção do grau de Mestre em Ciência da Computação.

Orientador: George Marconi de Araújo Lima
Co-orientador: Ernesto de Souza Massa Neto

Salvador
10 de setembro de 2024

Ficha catalográfica elaborada pela Biblioteca Universitária de Ciências
e Tecnologias Prof. Omar Catunda, SIBI – UFBA.

B273 Barreto, João Victor Alves

PSSA: Um algoritmo semi-particionado com server shadowing para o escalonamento de tarefas de tempo real periódicas em múltiplos processadores – Salvador, 2024.

52 p.: il.

Orientador: Prof. Dr. George Marconi de Araújo Lima.
Co-orientador: Prof. Dr. Ernesto de Souza Massa Neto.

Dissertação (Mestrado) – Universidade Federal da Bahia. Instituto de Computação, 2024.

1. Sistemas de tempo real. 2. Escalonamento. Multiprocessadores. 4. Server Shadowing. I. Lima, George Marconi de Araújo. II. Massa Neto, Ernesto de Souza. III. Universidade Federal da Bahia. Instituto de Computação. IV. Título

CDU:004.031.43

MINISTÉRIO DA EDUCAÇÃO
UNIVERSIDADE FEDERAL DA BAHIA
INSTITUTO DE COMPUTAÇÃO
PGCOMP - Programa de Pós-Graduação em Ciência da Computação
http://pgcomp.ufba.br

"*PSSA: Um algoritmo semi-particionado com server shadowing para o escalonamento de tarefas de tempo real periódicas em múltiplos processadores*"

João Victor Alves Barreto

Dissertação apresentada ao Colegiado do Programa de Pós-Graduação em Ciência da Computação na Universidade Federal da Bahia, como requisito parcial para obtenção do Título de Mestre em Ciência da Computação.

Banca Examinadora

Profº. Drº. George Marconi de Araújo Lima (Orientador - PGCOMP)

Profº. Drº. Konstantinos Bletsas (CISTER/ISEP)

Profº. Drº. Maycon Leone Maciel Peixoto(UFBA)

AGRADECIMENTOS

Agradeço a Deus pelo discernimento, força e oportunidade de realizar este trabalho, colocando no meu caminho pessoas maravilhosas.

Agradeço ao meu orientador Prof. Dr. George Marconi de Araújo Lima e ao meu co-orientador Prof. Dr. Ernesto de Souza Massa Neto por todo o suporte e aprendizado.

Sou grato à minha família, em especial à minha mãe (Cristiane), meu pai (Edilson) e meu irmão (Eduardo) por me darem apoio e motivação.

Por fim, quero agradecer à CAPES pelo financiamento da pesquisa. O presente trabalho foi realizado com apoio da Coordenação de Aperfeiçoamento de Pessoal de Nível Superior - Brasil (CAPES) - Código de Financiamento 001.

RESUMO

Os sistemas de tempo real estão presentes em diversas aplicações, desde aplicações críticas, como a automotiva e aviação, até as não críticas como multimídia e jogos *on-line*. Para que os sistemas de tempo real críticos funcionem corretamente, é necessário garantir que todas as suas tarefas executem dentro de prazos (*deadlines*) pré-definidos. Para tanto, o escalonador de tarefas do sistema tem papel fundamental, pois determina a cada instante qual tarefa deve executar para que todas possam cumprir seus prazos. Em sistemas com um processador, o escalonador *Earliest Deadline First* (EDF) é uma das melhores opções, pois garante que todas as tarefas cumprem seus *deadlines* sempre que isso é possível. Em sistemas com mais de um processador, o mesmo não é verdade. Este trabalho propõe o escalonador *Partitioning and Server Shadowing Algorithm* (PSSA) para sistemas multiprocessados compostos de tarefas periódicas e independentes. A ideia do algoritmo é particionar as tarefas entre os processadores para então utilizar o escalonamento EDF em cada processador. Quando tal particionamento não é encontrado, alguma tarefa deve executar em mais de um processador, enquanto alguns processadores são deixados com capacidades ociosas. A fim de gerenciar execução de tarefas que executam em mais de um processador, o PSSA utiliza uma nova abordagem denominada *server shadowing*, de acordo com a qual as partes ociosas dos processadores reais são utilizadas para criar processadores lógicos. Quando uma tarefa executa em um processador lógico, na verdade ela executa em algum dos processadores reais que forneceram sua capacidade ociosa ao processador lógico onde a tarefa está alocada. Os conceitos de *server shadowing* e processadores lógicos deram ao PSSA a capacidade de obter resultados similares aos melhores resultados publicados até hoje em termos de quantidade de migrações e preempções geradas para tarefas periódicas, como demonstrado através de extensivas simulações.

Palavras-chave: Tempo real, Escalonamento, Multiprocessadores, Servidor, *Server Shadowing*

ABSTRACT

Real-time systems are present in several applications, from critical applications, like automotive and aviation, to non-critical such as multimedia and online games. For critical real-time systems to function correctly, it's necessary to ensure that all tasks execute within pre-defined deadlines. To this end, the system's task scheduler plays a fundamental role, as it determines at each moment which task must be executed so that they can all meet their deadlines. In systems with one processor, the Earliest Deadline First (EDF) scheduler is one of the best options, as it guarantees that all tasks meet their deadlines whenever possible. On systems with more than one processor, the same is not true. This work proposes the Partitioning and Server Shadowing Algorithm (PSSA) scheduler for multiprocessor systems composed of periodic and independent tasks. The idea of the algorithm is to partition tasks between processors and then use EDF scheduling on each processor. When such partitioning is not found, some tasks must execute on more than one processor, while some processors are left with idle capacities. In order to manage the execution of tasks that run on more than one processor, PSSA uses a new approach called server shadowing, according to which the idle parts of real processors are used to create logical processors. When a task runs on a logical processor, it actually runs on one of the real processors that have provided their idle capacity to the logical processor where the task is allocated. The concepts of server shadowing and logical processors gave PSSA the ability to obtain results similar to the best results published to date in terms of the number of migrations and preemptions generated for periodic tasks, as demonstrated through extensive simulations.

Keywords: Real-time, Scheduling, Multiprocessors, Server, Server Shadowing

SUMÁRIO

LISTA DE FIGURAS

LISTA DE TABELAS

LISTA DE SIGLAS

Capítulo 1

INTRODUÇÃO

Sistemas de tempo real diferem dos sistemas computacionais convencionais porque seu funcionamento não depende apenas da correção lógica das saídas, mas também do instante físico em que esses resultados são produzidos (KOPETZ, 1997). Esse tipo de sistema torna-se cada vez mais relevante à medida em que está presente em diversas aplicações, tais como *streaming* de vídeos, jogos eletrônicos, sistemas de aviação, o *antilock braking system* (ABS) em automóveis, controle industrial (KORKI; JIN; TIAN, 2022). Os Sistemas de tempo real (STR) são geralmente estruturados por um conjunto de tarefas, que são programas que devem ser executados recorrentemente cujos resultados estão associados a atributos temporais. No contexto de um STR automotivo, o conjunto de instruções necessárias para frear o automóvel é um exemplo de tarefa. Após ser ativada no sistema, há um prazo máximo para que sua execução seja finalizada. Nesse caso especificamente, não atender ao requisito temporal pode causar um grave acidente. Este limite de tempo é o prazo que a tarefa tem para satisfazer toda demanda de execução exigida por ela, o qual é chamado na literatura de *deadline*. Cada tarefa tem como características essenciais o quanto ela precisa executar e o prazo em que essa execução deve ser feita. Para que o sistema possa fornecer o recurso computacional para o conjunto de tarefas que compõem o sistema, é necessário que cada instância de cada tarefa, denominada *job*, receba os recursos computacionais necessários ao logo do tempo. Uma tarefa pode liberar infinitos *jobs*. Quando o freio do automóvel é acionado, é liberado um *job* da tarefa de frenagem para disputar tempo de execução com os demais. Outro exemplo seria uma tarefa periódica que monitora a temperatura em fábricas. Ao contrário da frenagem que cria um *job* no acionamento do freio, no monitoramento essas instâncias são criadas periodicamente dentro de um intervalo de tempo. Por exemplo, a tarefa pode ter um período de 5 segundos, liberando *jobs* nos instantes 0, 5, 10, ...

A falha em atender os requisitos temporais de um sistema gera uma consequência cuja severidade dependerá da aplicação da qual eles fazem parte. Se uma transmissão ao vivo ou um jogo *on-line* parar, é possível que o usuário fique insatisfeito e perca o interesse em continuar naquela atividade, não sendo nada grave. Contudo, se um sistema de controle aéreo, em resposta às entradas, lida com muitas perdas de prazos (*deadlines*), isso pode resultar até na queda da aeronave, sendo uma consequência catastrófica (KORKI; JIN; TIAN, 2022). É por isso que os STR possuem categorias distintas quanto à sua criticidade. Nos exemplos citados anteriormente, mídias ao vivo e jogos *on-line* são categorizados como não críticos (*soft real-time systems*), os

quais toleram algumas perdas de *deadlines* já que as consequências são brandas. Sistemas como os aviônicos e automotivos são considerados críticos (*hard real-time systems*), porque perdas de *deadlines* podem gerar consequências catastróficas, causando danos no mundo real e até perdas de vidas (KOPETZ, 1997). Dessa forma, um sistema de tempo real crítico deve sustentar um comportamento temporal garantido sob todas as condições de carga e falha especificadas (KOPETZ, 1997).

Sistemas críticos devem executar um conjunto de tarefas de tempo real concorrentes de modo que todas cumpram seus *deadlines* específicos. Toda tarefa precisa de recursos computacionais e de dados. O problema do escalonamento visa alocar esses recursos de modo que todos os requisitos temporais do sistema sejam atendidos (KOPETZ, 1997). Garantir o desempenho em tempo real ao fazer o uso mais eficaz da capacidade de processamento disponível requer o uso de políticas ou algoritmos de escalonamento eficientes (DAVIS; BURNS, 2011). Inicialmente esses algoritmos escalonadores eram projetados para sistemas compostos por somente um único processador. Porém, atualmente, existem muitos resultados-chave para o escalonamento em um único processador, documentados em livros e transferidos com sucesso para a prática industrial (AKESSON et al., 2022). Com a área de sistemas uniprocessados tida como madura, aliada ao fato do surgimento e popularização da tecnologia de múltiplas unidades computacionais através do *multicore*, os esforços voltaram-se à resolução do problema do escalonamento de tarefas de tempo real em arquiteturas com mais de um processador. Entretanto, arquiteturas multiprocessadas trazem mais desafios ao escalonamento e muitas soluções eficientes para somente um processador não conseguem manter essa característica em sistemas multiprocessados, causando as indesejadas perdas de *deadlines*. Outra preocupação é quanto ao desempenho do escalonador em relação aos *overheads* gerados. Por exemplo, os escalonadores geralmente permitem preempção. Isto é, há a possibilidade interromper a execução de uma tarefa para executar outra mais prioritária. Uma preempção gera *overhead* relacionado à troca de contexto. Em múltiplos processadores, nos escalonadores que permitem a migração de tarefas de um processador para outro em tempo de execução, além das trocas de contexto, ainda existem *overheads* relacionados à memória cache. Perder conteúdo de memória cache significa aumentar o tempo de processamento da tarefa devido à busca das informações em memória necessárias para sua execução.

Para escalonar tarefas de tempo real em múltiplos processadores, existem duas abordagens tradicionais (ANDERSSON, 2022): global e particionada. Algoritmos particionados se beneficiam das soluções consolidadas voltadas para sistemas com um único processador (DAVIS; BURNS, 2011), como *Earliest Deadline First* (EDF), porque essa estratégia considera o problema do escalonamento em múltiplos processadores como um conjunto de escalonamentos independentes para um sistema uniprocessado. Por exemplo, para aplicar EDF na sua versão particionada (pEDF), basta aplicar uma heurística de empacotamento (*bin-packing*) e, uma vez que as tarefas foram alocadas individualmente a processadores, cada unidade de processamento escalona suas tarefas de acordo com EDF em tempo de execução. Após a execução dessa etapa *off-line* de alocação, tarefas ficam associadas a processadores dedicados onde ficarão durante a geração *on-line* do escalonamento, ou seja, sem migrar para outro processador. Evitar migração de tarefas entre processadores pode ser visto como vantagem no aspecto de recuperação de dados na memória, fazendo os algoritmos particionados possuírem a característica de baixo *overhead* no que diz respeito aos tempos adicionais que a obtenção dos dados exige. Infelizmente, os algoritmos particionados também estão associados a uma baixa utilização da capacidade de processamento do sistema. Por exemplo, sendo m a quantidade de processadores, um sistema com $m + 1$ tarefas, cada uma exigindo pouco mais de 0.5 de um processador, não é escalonado

por nenhuma abordagem particionada (ANDERSSON, 2022). Isso significa que o limite máximo de utilização dos escalonadores particionados não é maior que 50% do sistema no pior caso.

Escalonadores globais utilizam uma única fila (global), na qual os *jobs* liberados pelas tarefas são selecionados em tempo de execução. No EDF implementado de forma global, os *jobs* na fila estão dispostos na ordem EDF. Portanto, ao contrário dos particionados, os algoritmos globais permitem que uma tarefa migre entre os processadores do sistema, já que uma unidade de processamento pode atender a qualquer tarefa do conjunto. A possibilidade da migração gera um ponto de atenção que é a restrição na execução simultânea de *jobs* de uma mesma tarefa em processadores distintos. Com isso, os escalonadores que permitem migração devem implementar mecanismos de sincronização para evitar que essa execução paralela aconteça. Outro aspecto a ser considerado é que essa classe de algoritmos pode sofrer com algumas anomalias no escalonamento, como o Efeito Dhall (DHALL; LIU, 1978), reduzindo a utilização efetiva dos processadores do sistema. Em resumo, o efeito Dhall demonstra que tarefas que liberam *jobs* periodicamente em intervalos infinitesimais geram um bloqueio de execução em *jobs* de tarefas com alta demanda de execução e longos períodos. Escalonadores globais mais elaborados conseguem atingir 100% de limite de utilização. A maioria deles, no entanto, gera um alto *overhead* em termos de migração e preempção devido à necessidade de sincronizar as decisões de escalonamento entre os diferentes processadores, e de atribuir corretamente a porção que é dividida entre as tarefas do sistema.

Além das estratégias ditas como tradicionais, existe uma terceira que é a semi-particionada. A abordagem semi-particionada faz uma mescla das estratégias particionada e global, de modo que há uma fase de particionamento, mas algumas tarefas podem migrar em tempo de execução. O particionamento estabelece quais tarefas ficarão fixas, assim como nos algoritmos particionados, mas as tarefas que são divididas entre mais de um processador migram como na estratégia global. Aplicar o particionamento previamente permite diminuir o número de migrações e seus *overheads* correspondentes, enquanto a divisão de algumas tarefas entre mais de um processador oferece um maior limite de utilização do sistema, evitando a limitação que os escalonadores particionados possuem nesse quesito. O termo escalonamento semi-particionado foi cunhado no artigo de Kato e Yamasaki (2009). Logo, algoritmos anteriores a 2009 que utilizaram esta estratégia não receberam originalmente esta denominação. Posteriormente, o termo escalonamento semi-particionado passou a ter uma interpretação mais ampla; qualquer algoritmo em que uma tarefa pode ser dividida antes do tempo de execução passou a ser referido como escalonamento semi-particionado (ANDERSSON, 2022).

Sincronizar as tarefas para que não executem ao mesmo tempo, obter um alto limite de utilização dos processadores e um baixo número de *overheads*. Esses são alguns dos desafios que motivam os pesquisadores da área a pensarem em escalonadores cada vez mais eficientes voltados para essa arquitetura em multiprocessadores. Diante desses desafios, o objetivo deste trabalho é propor um escalonador eficiente voltado para sistemas críticos de tempo real em um ambiente com múltiplos processadores. O algoritmo escalonador proposto neste trabalho é o *Partitioning and Server Shadowing Algorithm* (PSSA) (BARRETO; MASSA; LIMA, 2023), o qual utiliza o semi-particionamento devido ao propósito dessa estratégia de ter um alto limite de utilização sem gerar um grande número de trocas de contexto e migrações. Para validar seu funcionamento, o algoritmo foi implementado em um simulador de eventos discretos e testado através de simulações com conjuntos de tarefas sintéticas. O número de preempção e migração por *job* foram as métricas estabelecidas e o algoritmo teve seu desempenho comparado com outros escalonadores através dos resultados obtidos com a simulação. É importante salientar

que grande parte do material deste trabalho foi amplamente baseado no artigo publicado no Simpósio Brasileiro de Engenharia de Sistemas Computacionais, SBESC 2023.

Nesta introdução foram contextualizados os sistemas de tempo real, bem como o problema do escalonamento em multiprocessadores. O Capítulo 3 apresenta conceitos intrínsecos ao escalonamento de forma mais aprofundada. No Capítulo 2, são abordados os trabalhos relacionados com esta pesquisa. O escalonador proposto tem seu funcionamento detalhado no Capítulo 4, onde também é apresentada sua prova de correção. O percurso metodológico e avaliação dos resultados obtidos são expostos no Capítulo 5. As modificações feitas para a tentativa de um suporte ao modelo de tarefas esporádicas são discutidas no Capítulo 6. Por fim, no Capítulo 7, são apresentadas considerações finais com base nos resultados obtidos junto com os trabalhos futuros.

Capítulo

2

FUNDAMENTAÇÃO TEÓRICA

As tarefas de tempo real devem atender a eventos, que podem ser externos ou internos para o sistema (MALL, 2009). Por exemplo, a execução de uma tarefa pode ser requisitada por um evento interno, como uma interrupção de relógio a cada milissegundo para periodicamente checar a temperatura de uma indústria química. Outra possibilidade é atender a um evento externo que seja associado a um comando do usuário ou a alguma variação de valores de variáveis monitoradas. Um conjunto de tarefas compõe um sistema de tempo real. A fundamentação teórica aqui apresentada tem um papel importante para a compreensão dos componentes de um sistema de tempo real. Serão detalhadas as propriedades de uma tarefa de tempo real, assim como estruturas mais complexas tal qual servidores. Além disso, uma notação será formalizada com o intuito de facilitar o entendimento ao longo de todo o trabalho.

2.1 TIPOS DE TAREFAS

Considerando que as tarefas de tempo real requisitam execução recorrentemente, é possível classificá-las em três principais categorias: periódicas, esporádicas e aperiódicas (MALL, 2009). A maioria das pesquisas em escalonamento de tarefas de tempo real em múltiplos processadores foca nos modelos de tarefas periódicas e esporádicas. Em ambos os modelos, as tarefas podem dar origem a uma sequência potencialmente infinita de *jobs* (DAVIS; BURNS, 2011), os quais são instâncias das tarefas.

No modelo de tarefas periódicas, os *jobs* de uma tarefa chegam estritamente periodicamente, separados por um intervalo de tempo fixo. Exemplificando, se um tarefa possui período $T = 10$, a cada 10 unidades de tempo sempre chegará um *job* dela, o que faz deste um modelo razoavelmente previsível. No modelo de tarefas esporádicas, cada *job* de uma tarefa pode chegar a qualquer momento, desde que seja sua primeira liberação ou já tenha passado o intervalo de tempo mínimo desde a chegada do *job* anterior da mesma tarefa (DAVIS; BURNS, 2011). Isso quer dizer que se um *job* chega no tempo $t = 5$ e possui um intervalo mínimo de $T = 10$, não será disparado outro *job* dessa tarefa antes do tempo $t = 15$. Em um sistema com múltiplos processadores, o paralelismo entre as tarefas não é permitido por nenhum dos modelos. Consequentemente, a qualquer momento, cada *job* deve executar em, no máximo, um processador (DAVIS; BURNS, 2011).

Além de seu período T_i, cada tarefa τ_i é ainda caracterizada pelo seu *deadline* relativo D_i e seu tempo de execução no pior caso ou *Worst-case execution time* (WCET) C_i (DAVIS;

BURNS, 2011). O *deadline* é o prazo que cada *job* da tarefa tem para ser concluído. Quando o *deadline* é relativo, considera-se o prazo a partir de sua chegada. Já o *deadline* absoluto leva em conta o tempo que já passou desde o início do escalonamento. Por exemplo, se a tarefa tem 10 unidades de tempo de *deadline* relativo (D_i) e seu *job* foi liberado no tempo $t = 5$, seu *deadline* absoluto é 15 ($D_i + t$). Quanto aos *deadlines* também existem três classificações sendo elas: implícitos, restritos ou arbitrários. É classificado como *deadline* implícito quando $D = T$, ou seja, o *deadline* da tarefa é igual ao tempo mínimo entre chegadas. Ele é restrito quanto $D \leq T$ e arbitrário quando pode ser menor, maior ou igual ao tempo mínimo entre chegadas.

O *worst-case execution time* ou tempo de execução no pior caso C_i de uma tarefa τ_i é o tempo que cada *job* de uma tarefa precisa executar antes que chegue o seu *deadline*. Como o próprio nome sugere, esse tempo é estimado considerando o pior caso para que não corra o risco de uma tarefa não ter seu *deadline* cumprido.

2.2 MODELO DE TAREFAS

O modelo de tarefas adotado leva em conta um conjunto de tarefas $\Gamma = \{\tau_1, \ldots, \tau_n\}$ a serem escalonadas em m processadores idênticos. Uma tarefa τ_i libera *jobs* periodicamente, a cada instante $k\,T_i$ ($k = 0, 1, \ldots$), cada um deles necessitando de até C_i unidades de tempo para completar sua execução até o tempo $(k + 1)\,T_i$, $k = 0, 1, \ldots$. O *deadline* (absoluto) atual de uma tarefa τ_i no instante t é indicado por $D(\tau_i, t)$. Além de caracterizar uma tarefa pelos seus parâmetros individualmente, a tarefa também pode ser representada por $\tau_i = (C_i, D_i)$. Como os *deadlines* são implícitos, o valor de T_i pode ser obtido através de D_i e vice-versa.

As tarefas são tidas como independentes, o que significa que não há relação de precedência entre as tarefas e elas não compartilham nenhum recurso além dos processadores. A utilização de uma tarefa τ_i é definida como $U(\tau_i) = C_i/T_i$ e representa a parcela de processamento exigida por τ_i. A utilização de todo o conjunto de tarefas Γ é dada por $U(\Gamma) = \sum_{\tau_i \in \Gamma} U(\tau_i)$. É assumido que $U(\Gamma) \leq m$, do contrário não há escalonamento factível para Γ. Custos associados com preempção e migração de tarefas entre processadores durante o tempo de execução são considerados como incorporados ao tempo de execução das tarefas.

A Tabela 2.1 resume toda a notação utilizada ao longo do documento. Essa notação foi escolhida para refletir o que é frequentemente utilizado na literatura. Sendo assim, elementos dela serão vistos em outros trabalhos na área de sistemas de tempo real.

2.3 SERVIDORES

Um servidor é uma entidade escalonada pelo sistema e capaz de escalonar outras entidades. Um servidor pode ser visto como uma tarefa, com a mesma característica de gerar uma sequência de *jobs*. Mas, na verdade, servidores não são tarefas para o sistema; cada servidor aglomera uma coleção de tarefas ou outros servidores que são denominadas clientes desse servidor (REGNIER et al., 2011). Quando um servidor está executando, o tempo de processamento é usado por um de seus clientes. A escolha de qual cliente deve ser executado em um instante específico é feita através de um mecanismo de escalonamento. A utilização total $U(\sigma)$ representa a demanda de execução de um servidor σ, que não deve ultrapassar 1 ($U(\sigma) \leq 1$), já que isso significaria exigir mais de 100% da capacidade de um processador. A quebra dessa condição tornaria impossível a alocação de um servidor em um processador. Diante disso, o mecanismo de escalonamento interno pode aproveitar-se de algoritmos consolidados para sistemas com um processador. Vale ressaltar que o tipo do servidor utilizado neste trabalho e nos demais trabalhos relacionados

τ; σ, J	Tarefa; Servidor; *Job*
Γ	Conjunto de tarefas ou servidores
C_i	Tempo de execução no pior caso (WCET) de uma tarefa τ_i
T_i	Intervalo mínimo entre chegadas de uma tarefa τ_i
$D_i; D(\tau_i)$	*Deadline* relativo de uma tarefa τ_i
$D(\tau, t)$	*Deadline* absoluto de uma tarefa τ no instante t
$H(\sigma, t)$	Horizonte de um servidor σ no instante t
$U(\tau_i)$	Utilização de uma tarefa ou servidor τ_i
r	Instante de liberação (*release*) de um *job* de uma tarefa τ
c	Tempo de execução no pior caso (WCET) de um *job* de uma tarefa τ
d	*deadline* de um *job* de uma tarefa τ
m	Quantidade de processadores no sistema
t	Tempo/instante
n	Número de tarefas do sistema
(σ_i^1, σ_i^2)	Par de servidores *shadow* oriundos de um processador com índice i

Tabela 2.1 Notação utilizada neste trabalho.

é o servidor EDF de taxa fixa. Ele recebe o nome de Servidor EDF pelo seu mecanismo de escalonamento seguir a ordem EDF (LIU; LAYLAND, 1973), detalhado na Seção 3.1. A propriedade de taxa fixa é porque, apesar dos clientes possuírem *deadlines* distintos, a utilização necessária para atender às demandas de execução do servidor permanece a mesma em cada intervalo definido por dois *deadlines* consecutivos. O termo "taxa"é utilizado nos algoritmos RUN e QPS como a fração de capacidade computacional reservada para o servidor. Dessa forma, a referida taxa de um servidor é equivalente à sua utilização. Visto que esse tipo de servidor já foi implementado nos algoritmos RUN (REGNIER et al., 2011) e QPS (MASSA et al., 2014), a definição desses servidores a seguir é embasada nas definições presentes nesses trabalhos.

Definição 2.3.1 (Servidores EDF de taxa fixa)**.** *Um servidor EDF de taxa fixa é um mecanismo de escalonamento o qual possui a capacidade de regular a execução de um conjunto de tarefas ativas ou outros servidores* Γ*, conhecido como seus clientes. Um servidor* σ *possui utilização* $U(\sigma) \leq 1$ *que representa a fração da capacidade de processamento reservada para execução de seus clientes. Os atributos e comportamentos de um servidor* σ *podem ser resumidos da seguinte maneira:*

- ***Deadline****: O deadline de um servidor* σ *em um instante* t *é expresso por* $D(\sigma, t)$*. Essa notação representa o tempo no qual um job de* σ*, liberado em* t *ou antes, deve finalizar. Para qualquer cliente* σ' *de um servidor* σ *e qualquer instante* t*,* $D(\sigma', t) \geq D(\sigma, t)$*. Essa restrição é necessária porque* σ' *é escalonado somente quando* σ *é selecionado para executar. De fato, a prioridade de um servidor (via EDF) não pode ser menor que seu cliente com maior prioridade. Do contrário, a execução de um cliente mais prioritário pode ser atrasada se seu servidor tiver menor prioridade e não for escolhido para rodar.*

- ***Liberação de job e carga de trabalho****: Um job de um servidor* σ*, liberado num instante* r*, representa a quantidade de tempo (i.e., carga de trabalho) que ele requer*

para executar seus clientes de r até seu deadline $D(\sigma, r)$. Um servidor libera um job em qualquer instante que algum dos seus clientes libera um job. A carga de trabalho imposta por um job do servidor σ em um instante de liberação r é obtido por $(D(\sigma, r) - r) \times U(\sigma)$. Essa carga de trabalho define o budget de σ no momento da liberação de um job, que é consumido sempre que σ executa. A taxa de consumo do budget é uma unidade de tempo por unidade de execução. Nenhum cliente pode ser executado se o budget do servidor for nulo.

- ***Ordem de execução**: Quando um job de um servidor executa, ele escalona seus jobs clientes de acordo com a ordem EDF.*

Daqui em diante, os servidores EDF de taxa fixa serão mencionados apenas como servidores. Esse tipo de servidor é utilizado por PSSA da mesma forma empregada no algoritmo QPS (MASSA et al., 2014), sendo mais abrangente que no RUN (REGNIER et al., 2011). No RUN os servidores possuem utilização igual a soma das utilizações de seus clientes, e compartilham dos mesmos *deadlines* e tempos de liberações com seus clientes. No QPS, é dito que seus servidores são mais flexíveis que os presentes em RUN porque dois servidores podem compartilhar um mesmo conjunto de clientes. Entretanto, a soma da utilização do conjunto compartilhado não pode ultrapassar a soma das utilizações dos servidores que o compartilham. Os instantes de liberação de *jobs* e *deadlines* do conjunto atendido também são compartilhados entre os servidores. Essa flexibilidade para os servidores introduzida no QPS é importante para o funcionamento do PSSA, já que os processadores lógicos são compostos por pelo menos dois servidores, e um conjunto Γ atribuído a um processador lógico é compartilhado entre os servidores que o compõem.

Quando a utilização do servidor σ é igual a soma das utilizações dos seus clientes, o Teorema 2.3.1 oriundo do RUN (REGNIER et al., 2011) prova que nenhum cliente de σ perde *deadline*. Contudo, assim como no QPS, a soma das utilizações dos clientes de um servidor em PSSA não é restrita à utilização do servidor. Não há a restrição devido ao compartilhamento dos clientes entre mais de um servidor. Portanto, uma tarefa τ pode ser compartilhada entre três servidores σ^1, σ^2 e σ^3 desde que $U(\sigma^1) + U(\sigma^2) + U(\sigma^3) = U(\tau)$. Embora a demonstração do Teorema 2.3.1 presente no artigo do RUN (REGNIER et al., 2011) não atenda aos servidores do PSSA de forma direta, ele será importante para a prova de correção do PSSA, a qual será detalhada no Capítulo 4.

Teorema 2.3.1. *Um servidor EDF de taxa fixa σ produz um escalonamento válido para um conjunto de clientes $\mathcal{C}$ quando $U(\mathcal{C}) \leq 1$ e todos os jobs de σ cumprem seus deadlines.*

O comportamento de um servidor será ilustrado através de um exemplo, retratado na Figura 2.1. Considere um conjunto Γ de duas tarefas periódicas, com $C_1 = 1$, $C_2 = 0.9$, $T_1 = 2$ e $T_2 = 3$. Essas tarefas são clientes de um servidor σ cuja utilização é exatamente o que elas necessitam, *i.e.,* $U(\sigma) = 0.8$. Além disso, assuma que os dois clientes começam liberando seus *jobs* (periódicos) no tempo 0. A tarefa τ na figura representa uma possível execução concorrente e seus parâmetros não são relevantes para a explicação. No instante 0, σ libera seu *job* com carga de trabalho $(D(\sigma, 0) - 0) \times 0.8 = 2 \times 0.8 = 1.6$. Assumindo que σ tem o *deadline* mais próximo nesse momento, ele é selecionado para executar e escolhe um dos seus clientes via EDF. No instante 1.6 seu *budget* é consumido totalmente, prevenindo τ_2 de executar dentro do intervalo $[1.6, 2)$, quando τ executa. Note que no instante 2, o tempo de execução restante do primeiro *job* de τ_2 é $0.9 - 0.6 = 0.3$. O segundo *job* de σ é liberado no instante 2 com carga de

trabalho $(D(\sigma, 2) - 2) \times 0.8 = (3 - 2) \times 0.8 = 0.8$. Então σ seleciona seu cliente mais prioritário (τ_2 dessa vez), que completa sua execução e o segundo *job* de τ_1 consome o *budget* restante de σ até o instante 2.8. Como pode ser visto, σ gerencia a execução de seus clientes através do *budget*. Ademais, note que o *budget* de σ é mantido (*i.e.,* não consumido) durante intervalos que ele não é selecionado para executar.

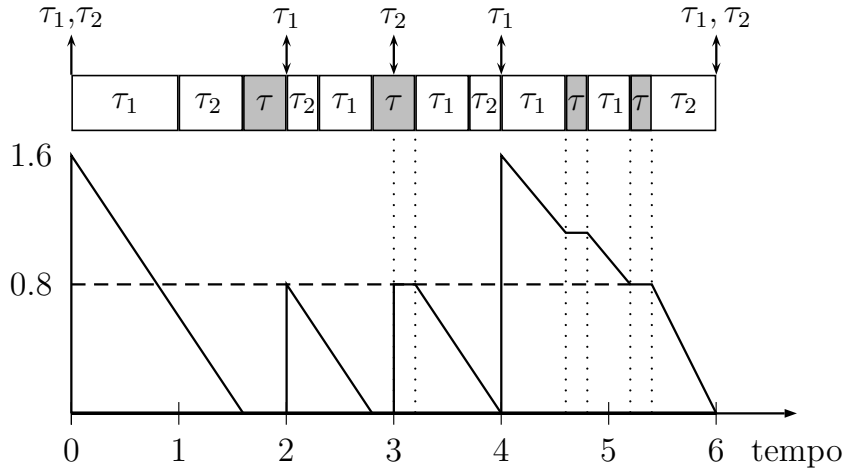

Figura 2.1 Um servidor σ executando de forma concorrente com uma tarefa τ em um processador. O servidor possui dois clientes, τ_1 e τ_2. As setas para cima e para baixo representam respectivamente os instantes de liberação e *deadlines*. O *budget* do servidor é recarregado nos momentos de liberação de seus *jobs* e é mantido constante sempre que o servidor não executa. Exemplo retirado de Lima (2023)

A utilização de servidores é parte fundamental do algoritmo proposto neste trabalho, assim como nos escalonadores RUN (REGNIER et al., 2011) e QPS (MASSA et al., 2014), os quais serão discutidos mais adiante. Cada escalonador emprega esse conceito de forma diferente. Porém, apesar de aplicados de forma distinta, o arcabouço teórico é o mesmo que o apresentado nesta seção. Outro ponto em comum é o papel desempenhado pelos servidores nesses algoritmos que é a sincronização das tarefas sem gerar muitos *overheads*, visto que obtiveram melhores resultados em comparação com outros escalonadores. Foi com o intuito de obter uma sincronização eficiente que foram adotados os servidores no *Partitioning and Server Shadowing Algorithm* (PSSA).

Capítulo 3

TRABALHOS RELACIONADOS

Escalonar as tarefas de maneira apropriada é o mecanismo básico adotado pelo sistema operacional de tempo real para que seus requisitos temporais sejam cumpridos. Portanto, a escolha de um algoritmo de escalonamento é crucial para um funcionamento correto de um sistema de tempo real (MALL, 2009). Um algoritmo de escalonamento é classificado como ótimo para uma classe de sistemas se ele gera um escalonamento para esta classe de acordo com o qual nenhum *deadline* das tarefas é perdido sempre que possível (LIMA; MASSA; REGNIER, 2022). Os algoritmos ótimos projetados para sistemas uniprocessados são muito importantes também para a evolução dos escalonadores multiprocessados. EDF é um exemplo disso, servindo de base para diversos algoritmos voltados aos sistemas multiprocessadores. Infelizmente, os algoritmos ótimos para um único processador não mantêm essa característica quando aplicados sem modificações em uma arquitetura com mais de uma unidade de processamento. Mesmo assim, existem diversos escalonadores para multiprocessadores na literatura que são ótimos. Neste Capítulo, serão abordados alguns desses algoritmos ótimos para múltiplos processadores, os quais se relacionam com o modelo de tarefas do PSSA, que é o escalonador proposto neste trabalho. O *Notional Processors* será o único algoritmo abordado que não é ótimo. Contudo, a ideia de usar processadores lógicos no escalonamento torna-o um trabalho relacionado com a proposta deste trabalho. É importante salientar que todos os algoritmos discutidos são preemptivos. Isso quer dizer que tarefas com maior prioridade podem interromper a execução de outra que seja menos prioritária. A quantidade de preempções geradas por um algoritmo é um ponto relevante que será utilizado para medir o desempenho de algoritmos de escalonamento, como será visto posteriormente.

3.1 ESCALONAMENTO EM SISTEMAS UNIPROCESSADOS

Apesar de este trabalho ser voltado ao escalonamento em múltiplos processadores, é importante entender como funciona o algoritmo *Earliest Deadline First* (EDF) (LIU; LAYLAND, 1973). Mesmo não sendo ótimo em multiprocessadores de forma direta, ele serve como base para muitos escalonadores do estado da arte. Além disso, a política de escalonamento do EDF está presente no PSSA.

EDF é um critério de escalonamento ótimo para sistemas com um processador, com tarefas independentes e sem restrições com relação a preempção (DERTOUZOS, 1974). Isso significa que se há um escalonamento viável para um conjunto arbitrário de *jobs*, então EDF gera um

escalonamento viável para tal conjunto. A otimalidade também se aplica para tarefas periódicas com deadlines implícitos, o que significa que qualquer conjunto de tarefas Γ tal que $\Gamma \leq 1$ é escalonável por EDF. Sua característica principal é a atribuição dinâmica de prioridades em tempo de execução, sendo atribuída a maior prioridade à tarefa que possui *job* com *deadline* absoluto mais próximo.

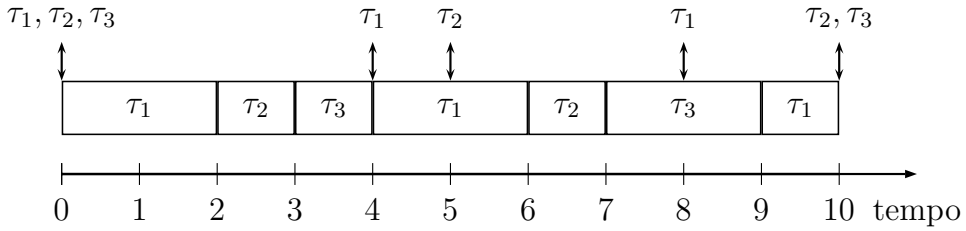

Figura 3.1 Escalonamento utilizando EDF. As setas para cima e para baixo representam, respectivamente, instantes de liberação e *deadlines*.

A Figura 3.1 mostra o escalonamento por EDF de um conjunto de tarefas $\Gamma = \{\tau_1 : (2, 4), \tau_2 : (1, 5), \tau_3 : (3, 10)\}$, com *deadlines* implícitos. Todas as tarefas liberam *jobs* no instante $t = 0$. Observe que o EDF consegue escalonar esse sistema totalmente utilizado ($U(\Gamma) = 1$). Por outro lado o escalonamento gerado por um algoritmo de prioridade fixa como *Rate Monotonic* (RM) (LIU; LAYLAND, 1973), para esse mesmo conjunto, perderia *deadline*. Isso ocorre porque no instante $t = 8$ chega um *job* da tarefa τ_1, e a prioridade fixa faria ela executar ao invés de τ_3 devido à prioridade maior de τ_1. Porém, no EDF, τ_3 continua executando por ter *deadline* absoluto de 10, que é menor que o *deadline* absoluto do *job* de τ_1 (12).

3.2 ESCALONAMENTO EM MÚLTIPLOS PROCESSADORES

Para a diversidade de escalonadores voltados aos sistemas com mais de uma unidade de processamento, também há diferentes modelos de tarefas os quais eles atendem. Alguns escalonadores são voltados para sistemas com múltiplos processadores heterogêneos. Isso é, os processadores podem ter características diferentes (como o poder computacional), o que permite variações na demanda de execução das tarefas. Já que esse tipo de sistema não é o foco deste trabalho, todas as soluções expostas nesta seção são para múltiplos processadores homogêneos, ou seja, processadores idênticos onde as demandas das tarefas não variam. É importante ressaltar que os escalonadores ótimos expostos nesta seção utilizam diferentes estratégias, não sendo restritos ao semi-particionamento empregado no PSSA. Há menções a algoritmos com modelos de tarefas diferentes do PSSA, mas que contribuíram com conceitos importantes. São eles o QPS e o *Notional Processors*, que foram criados para lidar com tarefas esporádicas.

3.2.1 DP-WRAP

DP-WRAP (LEVIN et al., 2010) é um algoritmo global ótimo para tarefas periódicas com *deadlines* implícitos. Para ser capaz de utilizar toda a capacidade de processamento do sistema, esse escalonador recorre a uma estratégia de compartilhamento de *deadline* (*deadline sharing*). Isto é, considerando um instante t, é estabelecida uma janela de tempo $[t, d)$ onde d é o menor *deadline* absoluto dentre todas as tarefas. A janela $[t, d)$ é compartilhada por todo o sistema, e a fração do processador que cada uma recebe para execução dentro dessa janela segue o

algoritmo de McNaughton (MCNAUGHTON, 1959), que estabelece uma execução proporcional à sua utilização. Considere um conjunto de tarefas $\Gamma = \{\tau_1 : (4,5), \tau_2 : (8,10), \tau_3 : (4,10)\}$. O escalonamento da Figura 3.2 mostra que é criada uma janela de 5 unidades de tempo, o qual é o menor *deadline* entre as tarefas. Dentro dessa janela, é possível perceber que τ_2 recebe sua execução proporcional requisitada, sendo $0.8 \times 5 = 4$ unidades de tempo. A grande desvantagem do algoritmo é a possibilidade do intervalo $[t, d)$ ser muito pequeno dependendo dos parâmetros da tarefa. Consequentemente, o uso desse tipo de janela de escalonamento em todos os processadores pode gerar altos *overheads* em tempo de execução.

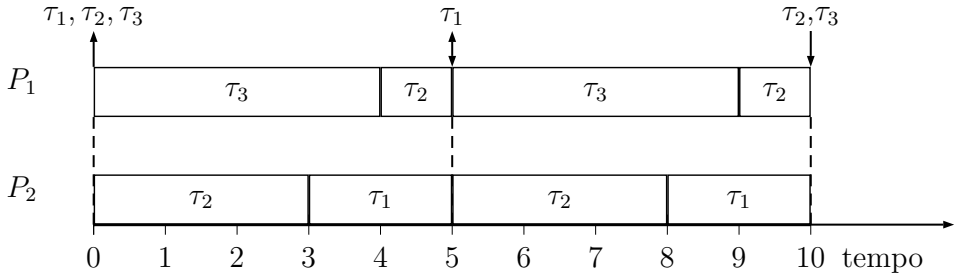

Figura 3.2 Escalonamento de um conjunto Γ com $C_1 = C_3 = 4, C_2 = 8, T_1 = 5, T_2 = T_3 = 10$ utilizando DP-Wrap.

3.2.2 *Reduction to Uniprocessor* (RUN)

Reduction to uniprocessor (RUN) (REGNIER et al., 2011) é um exemplo de algoritmo global ótimo, para um conjunto de tarefas periódicas com *deadlines* implícitos, que não se baseia na distribuição de tempo de execução para as tarefas proporcional às suas utilizações. Isto significa que há potencial diminuição de *overheads* de escalonamento que são inerentes a essa estratégia e também a quantidade de migrações e preempções ocorridas. Sendo assim, esse escalonador leva uma grande vantagem em comparações com outros que utilizam distribuição proporcional, isto é, *fairness*. Lima, Massa e Regnier (2022) dizem que a agregação de tarefas pode melhorar a escalonabilidade, justificando que: se lidar com muitas tarefas de baixa utilização é um problema, juntá-las em servidores resultará num conjunto menor de entidades para serem escalonadas e cada uma com maior utilização. Se os servidores são escalonados corretamente e eles escalonam corretamente seus clientes, um escalonamento válido em nível de sistema é obtido (REGNIER et al., 2011). Esse mecanismo interno de escalonamento dos servidores empregados no RUN é EDF. A agregação das tarefas em servidores é um dos conceitos principais para que o RUN possa fazer a redução da quantidade de processadores necessários até que somente seja preciso um único processador.

O RUN utiliza o princípio da equivalência entre escalonar as tarefas propriamente ditas com o escalonamento de seus períodos ociosos. O período ocioso de cada entidade, servidor ou tarefa, é chamado de *dual*. Se uma tarefa τ_i possui utilização $U(\tau_i) = 0.6$, seu *dual* é obtido por $U(\tau_1^*) = 1 - U(\tau_1) = 0.4$. Isto é, a utilização do *dual* é o complemento da utilização de uma tarefa, representando a fração que a tarefa não deve ser executada. A ideia é encontrar a partir do escalonamento para as entidades duais o escalonamento para as entidades originais, que é o escalonamento nos processadores reais. Considere o exemplo anterior das três tarefas de tempo real τ_1, τ_2 e τ_3, para serem executadas em dois processadores. A Figura 3.3 mostra como é possível gerar um escalonamento válido a partir do escalonamento de seus servidores

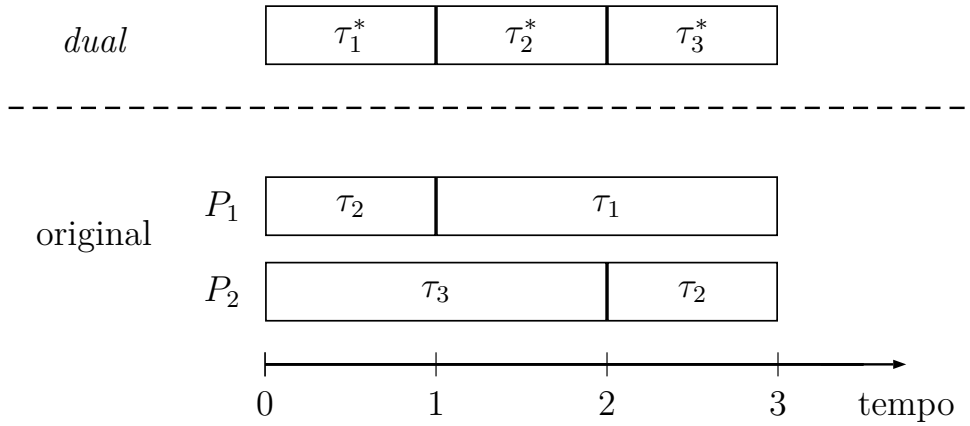

Figura 3.3 Equivalência de escalonamento de τ_1, τ_2, τ_3 em 2 processadores e $\tau_1^*, \tau_2^*, \tau_3^*$ em um processador. Exemplo retirado do RUN (REGNIER et al., 2011)

duais em um processador. No escalonamento *dual*, do instante 0 até 1, é escolhido o *dual* da tarefa τ_1 (τ_1^*). Isso significa que, neste instante, a tarefa τ_1 não deve ser executada, indicando a execução das demais tarefas. Portanto, são escolhidas as tarefas τ_2 e τ_3. É importante frisar que o processador onde o escalonamento *dual* da Figura 3.3 está sendo feito é fictício, sendo uma abstração utilizada para gerar o escalonamento original que, no caso do exemplo, usa 2 processadores reais (físicos).

Para fazer as reduções, primeiro as tarefas são aglomeradas em servidores EDF através de uma heurística de empacotamento (*bin-packing*), na operação chamada de `PACK`. Com as tarefas alocadas nos servidores, são associados a eles servidores duais na operação homônima `DUAL`. Esses servidores duais são utilizados para indicar os períodos ociosos dos outros servidores obtidos pela operação `PACK` anterior. Cada nível de redução é composto por uma operação `PACK` seguida de `DUAL`, que são repetidas sucessivamente até que só seja necessário um processador para escalonar o conjunto. Toda essa parte de criação dos servidores e a realização das reduções é o que compõe a parte *off-line* do algoritmo, que tem como produto uma árvore de redução, onde a raiz é fruto de um empacotamento com utilização igual a 1. É possível afirmar isso porque no RUN o sistema é sempre 100% utilizado, sendo preenchido com tarefas fictícias se o conjunto não utilizar toda a capacidade de processamento. Como o próprio nome do escalonador sugere, a redução converge para um conjunto que requer somente uma unidade de processamento. A Figura 3.4 demonstra essa estrutura de árvore com um conjunto de 7 tarefas. As setas e linhas horizontais representam uma operação `DUAL` e `PACK` respectivamente. Nessa árvore, `DUAL` e `PACK` são executadas em conjunto duas vezes consecutivamente, representando dois níveis de redução. Para realizar o escalonamento de todas as tarefas que estão nas folhas da árvore, é utilizado o escalonamento dos servidores duais $\{\sigma_8^{*(1/7)}, \sigma_9^{*(1/7)}, \sigma_{10}^{*(1/7)}\}$, que só precisam de um processador.

Na fase *on-line*, RUN utiliza os escalonamentos *dual* para achar o escalonamento nos processadores reais (*primal*) e assim utilizar os servidores EDF para escalonar seus clientes. Considerando a árvore de servidores obtida através das reduções, são mencionadas duas regras importantes. A primeira diz respeito ao servidor EDF e estabelece que um servidor empacotado em execução escolhe o nó filho com o *deadline* mais próximo que ainda possua trabalho pendente. Caso contrário, ele não executa nenhum dos seus filhos. A segunda regra mostra que

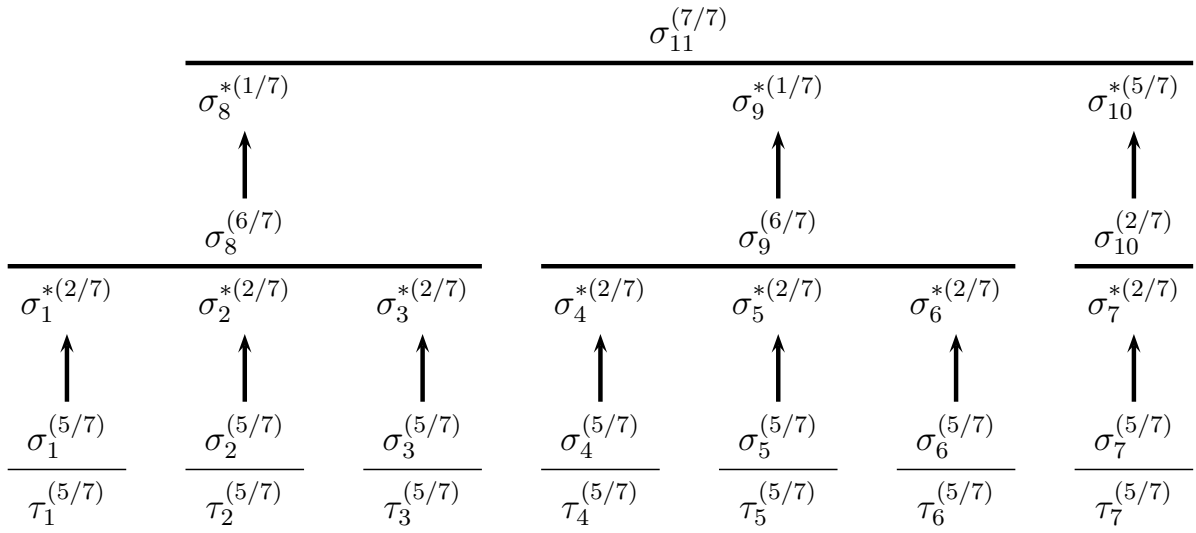

Figura 3.4 Ilustração de uma árvore redução gerada a partir das operações *dual* e *pack*. Fonte: (LIMA, 2023)

o servidor *dual* somente executa seu filho se o próprio servidor *dual* não está executando. Dessa forma, sempre que um *dual* for escolhido para executar, ele impede a execução do servidor EDF associado a ele. Atendendo a essas regras, é possível obter o escalonamento para o sistema físico original.

3.2.3 EKG

O algoritmo *EDF with task splitting and k processors in a group* (EKG), apresentado por Andersson e Tovar (2006), pertence à classe de algoritmos semi-particionados (ANDERSSON, 2022). A grande motivação para a criação desse algoritmo foi reduzir o número de preempções quando comparado a uma abordagem global que usa distribuição proporcional de recursos. EKG também fornece um limite maior de utilização do sistema em relação às estratégias particionadas. O modelo de tarefas da solução é baseado em tarefas periódicas com *deadlines* implícitos.

A parte de divisão de tarefas é um dos pilares da solução, alocando-as aos processadores de forma que a utilização não exceda 100%. O parâmetro k é um valor escolhido por quem implementa o algoritmo tal que $1 \leq k \leq m$, sendo m o número de processadores. A partir do valor de k, é calculado o valor de SEP, obtido pela Equação 3.1, que serve para separar as tarefas em leves e pesadas. Uma tarefa τ_i é pesada se $U(\tau) > SEP$, caso contrário, é leve. As tarefas pesadas são atribuídas aos seus próprios processadores dedicados, os restantes ficam responsáveis pelas tarefas leves.

$$SEP = \begin{cases} \dfrac{k}{k+1} & k < m \\ 1 & k = m \end{cases} \tag{3.1}$$

De acordo com o escalonamento EKG, as tarefas pesadas são executadas nos seus respectivos processadores assim que seus jobs são liberados. As leves são segmentadas e, após o cálculo dos dois instantes os quais as tarefas devem sofrer preempção, uma das tarefas dividida é executada antes do primeiro instante e a outra tarefa dividida é executada depois do segundo instante.

Durante esse intervalo, as tarefas não divididas são escalonadas utilizando EDF. Perceba que existe a utilização de uma janela de tempo que separa a execução de tarefas migratórias e não migratórias. Quanto menor a duração da janela, mais alto será número de segmentações no escalonamento. Por outro lado, janelas menores proporcionam maiores limites de utilização do sistema.

O limite de utilização do algoritmo é depende do valor da variável k que é usada no cálculo de SEP. Isso é mostrado por Andersson e Tovar (2006) através do teorema que diz: se as tarefas possuírem uma utilização total do sistema de $U(S) \leq SEP$, e são escalonados pelo EKG, então todos os *deadlines* serão cumpridos. Ao selecionar $k = 2$, o limite de utilização do algoritmo é de 66%. Selecionando $k = m$, o limite de utilização é 100%. É importante destacar que o aumento de k para a obtenção de um limite de utilização maior impacta diretamente o limite de preempções do algoritmo, que é de $2k$ por *job*. Dessa forma, quanto maior o k, maior limite de utilização e maior quantidade de preempções. Isso porque o k influencia no tamanho das janelas de tempo (*timeslots*) utilizadas para escalonar as tarefas. Quanto maior o k, menor será a duração dos *timeslots*. Além de proporcionar um alto limite de utilização do sistema, o limite de preempções provado no artigo de até $2k$ por *job* é apontado como uma grande vantagem dele em relação a outras abordagens.

3.2.4 *Notional Processors*

O algoritmo *Notional processors* (BLETSAS; ANDERSSON, 2009) tem a característica também de ser semi-particionado, mas foi criado para atender tarefas esporádicas com *deadlines* implícitos em sistemas com múltiplos processadores. O conceito mais importante para realizar o escalonamento é o uso dos processadores lógicos.

A criação dos processadores lógicos tem como visa a estabelecer reservas de execução onde os processadores reais executam as tarefas que foram integralmente atribuídas a eles. Com a criação das reservas, sobram as áreas ociosas desses processadores reais que darão origem ao processadores virtuais. As reservas são calculadas com base no intervalo de tempo S, que possui duração igual ao menor tempo entre chegadas das tarefas do sistema $\min_{\tau_i}(T_i)$. Uma reserva é a fração do intervalo S em um processador suficiente para que as tarefas não migratórias cumpram os seus *deadlines* quando escalonadas pelo EDF. As reservas são intercaladas com *gaps*, que são períodos ociosos restantes nos processadores após a definição do tempo reservado para as tarefas fixas. Sempre que o *gap* do processador P_p termina, começa o do processador P_{p+1}. A junção das capacidades de processamento dos períodos ociosos é o que dá a origem aos processadores lógicos. A Figura 3.5 mostra 5 processadores físicos e o processador lógico P_6 resultante do mapeamento das áreas ociosas dos processadores físicos, as quais estão representadas por partes com cores diferentes.

Bletsas e Andersson (2009) definem a estrutura do algoritmo em três etapas:

- Primeiro, as tarefas são atribuídas aos processadores físicos, até que uma tarefa não possa ser atribuída a nenhum processador;
- Então, a carga de trabalho é restrita em cada processador para execução dentro de reservas periódicas (de tamanho apropriado) e são organizados os intervalos de tempo entre as reservas nos processadores físicos em processadores lógicos;
- As tarefas restantes são atribuídas aos processadores lógicos.

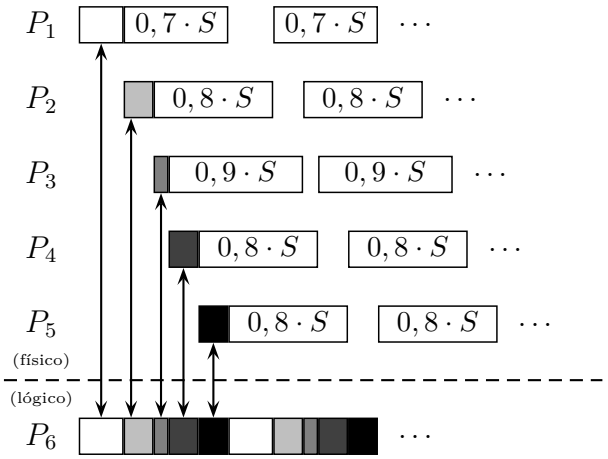

Figura 3.5 Ilustração de um *notional processor* P_6 representado pela junção das áreas coloridas (ociosas) dos processadores físicos P_1 até P_5. Exemplo do trabalho de Bletsas e Andersson (2009)

Para fazer essa atribuição aos processadores, é utilizado o empacotamento (*bin-packing*) *First-Fit Heavy-First* (FFHF). A técnica *First-Fit* (FF) atribui uma tarefa ainda não alocada ao primeiro processador que ela cabe, considerando as atribuições anteriores. Se a primeira tentativa de alocação falhar, o algoritmo tenta alocar a tarefa a outro processador até que seja possível atribuí-la. O FFHF utilizado pelo *Notional Processors* é uma variante que ordena as tarefas previamente, garantindo que elas sejam atribuídas sem a necessidade de criar novos processadores, a não ser que todos os processadores sejam utilizados mais que 50%. Um conjunto de tarefas está ordenado seguindo a ordem *heavy-first* se e somente se toda tarefa com utilização maior que 50% preceder toda tarefa com utilização igual ou menor que 50% (BLETSAS; ANDERSSON, 2009). Essa estratégia de empacotamento é utilizada para as atribuições tanto no processador físico quanto nos processadores lógicos criados posteriormente.

O algoritmo de empacotamento pode executar mais de uma vez caso as tarefas não caibam na primeira iteração, executando primeiro para alocar as tarefas aos processadores físicos e sendo interrompido caso uma tarefa não consiga ser atribuída a nenhum deles. Essa falha na alocação é onde começa a criação dos processadores lógicos. Depois de criados, a rotina de empacotamento será requisitada novamente para alocar as tarefas aos processadores lógicos. Se na segunda vez uma tarefa não conseguir ser atribuída a nenhum dos *notional processors*, o algoritmo não progride, não sendo possível escalonar esse conjunto.

Os processadores lógicos são estruturas que fazem o mapeamento para cada instante de escalonamento, indicando algum processador físico (provavelmente ocioso) para que a tarefa possa executar. Isso é, quando a tarefa está executando no processador lógico, na verdade ela está executando no processador físico que o *notional processor* indicou estar livre naquele momento. O algoritmo utiliza EDF para a escalonar as tarefas tanto nos processadores físicos quanto nos processadores lógicos. Quando o instante t pertence ao intervalo de uma reserva de tempo do processador físico, as tarefas são escalonadas normalmente de acordo com EDF. Fora das reservas, é preciso fazer uma verificação se no instante específico algum processador lógico aponta para um processador real através do mapeamento feito previamente de maneira estática. Em resumo, o processador físico irá executar uma tarefa se o processador lógico requisitá-lo naquele instante ou estiver dentro de sua reserva. Não ocorrendo uma dessas duas

condições, o processador físico fica ocioso.

A grande vantagem do algoritmo é superar o limite de utilização da versão esporádica do EKG (EKG-S), que é de 65,7% contra os 66% de limite de utilização do *Notional Processors*. Além disso, o *Notional Processor* ainda leva vantagem sobre o EKG-S no que diz respeito ao número de preempções geradas, sendo mais baixo do que o melhor caso possível do EKG-S (BLETSAS; ANDERSSON, 2009).

3.2.5 QPS

Quasi-partitioning scheduling (QPS) é um algoritmo que tem uma classificação distinta das tradicionais como global ou particionada, utilizando do "quasi-particionamento". Massa et al. (2014) define-o como o primeiro algoritmo de escalonamento capaz de adaptar a estratégia de escalonamento em função da carga do sistema. Isso porque o algoritmo utiliza o modelo de tarefas esporádicas com *deadlines* implícitos. Sendo as tarefas esporádicas, elas podem atrasar e exigir menos do sistema do que se todas tarefas estivessem ativas. QPS monitora a carga do sistema, em tempo de execução, e alterna entre o modo QPS e EDF, conforme a necessidade do conjunto ativo. A adaptação dinâmica é apontada como responsável por reduzir drasticamente o número de migrações que o escalonador necessita. É importante ressaltar que esse escalonador é ótimo tanto para tarefas periódicas quanto para tarefas esporádicas.

De forma resumida, o funcionamento baseia-se no particionamento das tarefas em subconjuntos de dois tipos: conjuntos de execução menores e maiores. Os subconjuntos menores requerem somente um processador, enquanto os maiores necessitam de dois processadores para serem escalonados. Caso todos os subconjuntos sejam menores, QPS se comporta como o EDF particionado. Já os conjuntos de execução maiores são escalonados ou por um conjunto de 4 servidores QPS em dois processadores (modo QPS), ou por EDF em um único processador (modo EDF), dependendo dos requisitos de execução. Através do monitoramento dos conjuntos de execução maiores em tempo de execução, QPS é capaz de permutar entre os dois modos, permitindo a adaptação dinâmica à carga do sistema (MASSA et al., 2014).

Para realizar o escalonamento, QPS usa servidores de taxa-fixa (ver Seção 2.3) cujo mecanismo interno para escolha dos clientes é o EDF. Considerando P um conjunto de servidores com utilização $U(P) \leq 1$, P é um conjunto de execução menor e o escalonamento consegue ser realizado com sucesso utilizando o EDF. O conjunto de execução maior possui utilização $U(P) > 1$ (MASSA et al., 2014). Os servidores QPS lidam com esses conjuntos maiores, cuja criação e funcionamento serão explicados a seguir.

Dado um conjunto de tarefas periódicas Γ e considerando $P = \{\tau_1 : (6, 15), \tau_2 : (12, 30), \tau_3 : (5, 10)\}$ um subconjunto de Γ. A utilização desse subconjunto é igual a 1.3 ($U(\tau_1) = 0.4$; $U(\tau_2) = 0.4$; $U(\tau_3) = 0.5$), podendo ser representada por $U(P) = 1 + 0.3$, indicando um excedente de 0.3. Como a utilização foi superior a 1, é criada uma bi-partição sendo elas $P^A = \{\tau_1, \tau_2\}$ e $P^B = \{\tau_3\}$. Como trata-se de um conjunto de execução maior, são criados os quatro servidores QPS σ^A, σ^B, σ^M e σ^S. Para definir a utilização e clientes dos servidores, será utilizada a notação $\sigma : (U(\sigma), \Gamma)$. Sendo o excesso do conjunto de execução maior $x = 0.3$, os servidores possuem as seguintes características: $\sigma^A : (U(P^A) - x, P^A)$, $\sigma^B : (U(P^B) - x, P^B)$, $\sigma^S : (x, P)$ e $\sigma^M : (x, P)$. Nesse exemplo, os servidores são $\sigma^A : (0.5, P^A)$, $\sigma^B : (0.2, P^B)$, $\sigma^S : (0.3, P)$ e $\sigma^M : (0.3, P)$. A qualquer momento t, todos os servidores QPS associados a P compartilham do mesmo *deadline* $D(P, t)$. σ^A e σ^B são servidores dedicados associados a P^A e P^B, respectivamente. σ^M e σ^S são os servidores mestre (*master*) e escravo (*slave*), respectivamente. A Figura 3.6 mostra como esses servidores seriam escalonados no intervalo de

tempo $[0, 10)$ (MASSA et al., 2014).

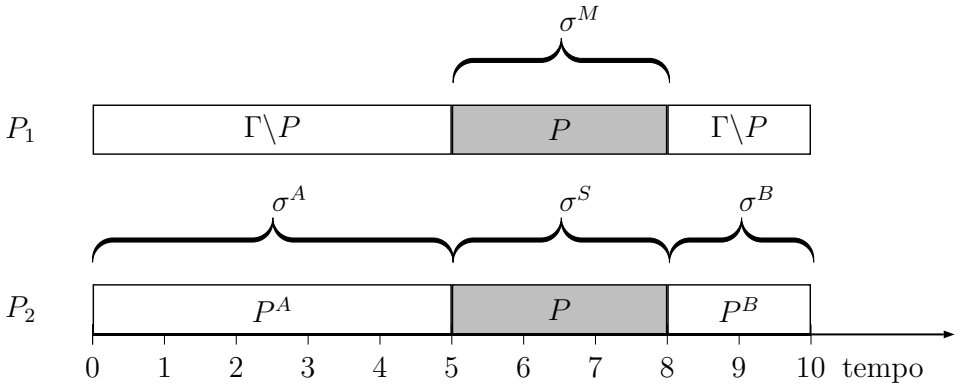

Figura 3.6 Ilustração de como funcionam os servidores QPS com σ^A, σ^B e σ^S executando no mesmo processador enquanto σ^M executa em outro. σ^M e σ^S devem executar sempre ao mesmo tempo. Na figura, eles executam durante o intervalo $[5, 8)$. Fonte: Massa et al. (2014).

Enquanto os servidores QPS σ^A e σ^B lidam com a execução não paralela de P^A e P^B, respectivamente, os servidores σ^M e σ^S são responsáveis pela execução paralela de toda a partição P. Como a soma das utilizações $\sigma^A + \sigma^B + \sigma^S = 1$, os servidores σ^A, σ^B e σ^S podem executar em um único processador. O servidor σ^M precisa ser executado em outro processador. Sempre que σ^M (mestre) é escalonado para executar, σ^S (escravo) também é, mas em outro processador. Esse comportamento justifica o nome dado a eles. O servidor escravo também é restringido de executar caso o mestre correspondente não execute. O comportamento hierárquico deles resulta numa execução paralela de uma tarefa de P^A e outra de P^B (MASSA et al., 2014). A Figura 3.6 exibe esse modo de operação, com os servidores mestre e escravo atuando no intervalo $[5, 8)$ realizando essa execução paralela. O mestre e escravo podem potencialmente servir qualquer cliente do conjunto de execução. Então, como forma de prevenir que o mesmo cliente seja escalonado para executar simultaneamente em dois processadores, sempre são escolhidos clientes diferentes para mestre e escravo (MASSA et al., 2014). Esse mecanismo é a forma encontrada para realizar a sincronização das tarefas migratórias.

3.2.6 Considerações

Neste capítulo foram apresentados alguns escalonadores relacionados com o PSSA, bem como suas características. É perceptível a importância do algoritmo EDF. Apesar de não manter sua condição de otimalidade em sistemas com múltiplos processadores, ele faz parte de muitos escalonadores que são projetados para atender a mais de um processador. Todos os algoritmos apresentados que usam servidores, inclusive o deste trabalho, aplicam EDF como mecanismo de escalonamento interno para a escolha de seus clientes. Isso deve-se ao fato que o servidor possui utilização $U(\sigma) \leq 1$, podendo ser interpretado como o escalonamento em um único processador. E, como abordado neste capítulo, EDF é ótimo em um único processador para os modelos de tarefas aqui considerados.

Os algoritmos globais mencionados possuem diferenças de desempenho. A estratégia global proporciona limite de utilização alto, como é o caso de DP-Wrap, ótimo para tarefas periódicas e *deadlines* implícitos. Contudo, essa condição de otimalidade vem ao custo de um alto número de preempções e migrações. RUN é um algoritmo ótimo para tarefas periódicas que não se

Algoritmo	Ano	Limite de utilização	Modelo de tarefas	Estratégia
DP-WRAP	2010	100%	esporádico	global
RUN	2011	100%	periódico	global
EKG	2006	configurável de 66% até 100%	periódico	semi-particionada
Notional Processors	2009	66%	esporádico	semi-particionada
QPS	2014	100%	esporádico	quasi-particionada

Tabela 3.1 Comparativo dos algoritmos para múltiplos processadores que fazem parte dos trabalhos relacionados com PSSA.

baseia em distribuição proporcional de recursos às tarefas, diminuindo bastante o número de preempções e migrações. Outro ponto notável é que ele também gera menos pontos de escalonamento necessários quando comparado a algoritmos que baseiam-se na distribuição proporcional de recursos.

EKG e *Notional Processors* são escalonadores semi-particionados que buscam obter um escalonamento com limites de utilizações mais altos que os particionados (que só garantem o escalonamento de conjuntos com até 50% de utilização), com os benefícios dessa estratégia de ter menos preempções e migrações que os escalonadores globais. EKG chega até ser ótimo para tarefas periódicas se $k = m$, sendo k seu parâmetro de configuração e m o número de processadores. Entretanto, se o objetivo for obter o máximo de utilização do sistema no EKG, isso gera alto *overhead* em consequência da segmentação no escalonamento com *timeslots*. *Notional Processors* é voltado para o modelo de tarefas esporádicas, tendo um limite de utilização de 66%. Infelizmente, ele não consegue garantir otimalidade mas, comparativamente ao EKG-S, versão do EKG para tarefas esporádicas, seu limite de utilização um pouco maior e menos preempções são geradas.

QPS também é um algoritmo voltado para o modelo de tarefas esporádicas com uma nova abordagem dita como quasi-particionada. Através de sua capacidade adaptativa e utilização de mecanismos de sincronização como servidores mestre-escravos, ele consegue atingir otimalidade para o modelo de tarefas periódicas e esporádicas.

A Tabela 3.1 mostra uma comparação dos algoritmos discutidos neste capítulo através das suas características destacadas. Com base nos escalonadores e conceitos apresentados, percebe-se que existe a procura por escalonadores que sejam capazes de escalonar qualquer conjunto de tarefa factível (ótimos), com baixo número de preempções e migrações, e que não sejam muito complexos. Esse seria o mundo ideal, uma vez que os autores optam por privilegiar uma ou mais características positivas em detrimento de outras. Mas as pesquisas focam na concepção de um escalonador que tenha o máximo possível dessas qualidades, sendo essa a maior motivação deste trabalho. O Capítulo 4 descreve PSSA, cujo objetivo é ser um escalonador ótimo que gera um baixo número de *overheads*.

Capítulo 4

PARTITIONING AND SERVER SHADOWING ALGORITHM (PSSA)

Neste capítulo será detalhado o funcionamento do PSSA, um algoritmo para escalonar tarefas de tempo real em multiprocessadores. Esse escalonador é voltado para sistemas de tempo real crítico (*hard real-time system*), que estão presentes em diversas áreas como: aviação, medicina, indústrias e *etc*. Levando em conta que as tarefas desses sistemas críticos não podem perder nenhum *deadline* sequer, porque as consequências são graves, é muito importante que o escalonador garanta que nenhuma tarefa do sistema perca *deadline*. O PSSA é um algoritmo que consegue escalonar qualquer conjunto de tarefa viável, ou seja, que não ultrapasse a capacidade do sistema ($U(\Gamma) \leq m$), considerando tarefas periódicas com deadlines implícitos. Além disso, PSSA não gera muitas preempções e migrações comparado com alguns escalonadores voltados para o mesmo modelo de tarefas. Essas características são importantes para torná-lo atrativo a fazer parte dos sistemas de tempo real críticos mencionados anteriormente.

Para ilustrar o raciocínio por trás do PSSA, será utilizado um conjunto composto de três tarefas de tempo real τ_1, τ_2 e τ_3, a serem executadas em dois processadores. Estas tarefas requerem 2/3 de cada processador. Ou seja, esse é um sistema totalmente utilizado. O PSSA primeiro tenta atribuir cada tarefa a um processador (físico). Nesse caso, τ_1 foi atribuído ao processador P_1 e τ_2 ao processador P_2. Como pode ser visto, um escalonamento válido para esse sistema leva em consideração o fato que τ_3 deve migrar entre os processadores durante sua execução de modo que ela utilize 1/3 de cada processador. No PSSA, essas áreas compartilhadas em ambos os processadores físicos são unidas para formar um novo processador (lógico) P_3 com capacidade de 2/3. Dessa forma, em uma segunda tentativa de particionamento, a tarefa restante (τ_3) pode ser atribuída de forma inteira a P_3. A Figura 4.1 mostra esse esquema de particionamento tarefa-processador. É claro que durante o tempo de execução o gerenciamento de migração das tarefas deve ser tratado adequadamente. Por exemplo, τ_3 não pode ser selecionada para executar em cada processador físico no mesmo instante. Com a finalidade de realizar essa sincronização, o PSSA emprega o conceito de servidores utilizados em esquemas de escalonamento anteriores como o *Reduction to uniprocessor* (RUN) (REGNIER et al., 2011) e *Quasi-partitioning scheduling* (QPS) (MASSA et al., 2014).

Os servidores são responsáveis pela atribuição das tarefas migratórias aos processadores e seu gerenciamento de migração em tempo de execução. Quando um servidor é selecionado para executar, um de seus clientes é escolhido. Nesse exemplo das três tarefas, a migração de τ_3 é

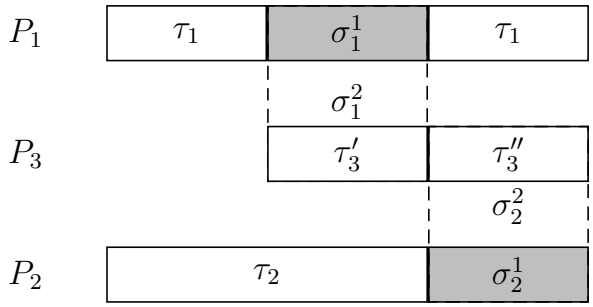

Figura 4.1 Raciocínio pro trás do PSSA. Tarefas não migratórias são atribuídas aos processadores P_1 e P_2 e um processador lógico P_3 gerencia a tarefa migratória τ_3 cuja execução estão na verdade ocorrendo em processadores físicos.

gerenciada da seguinte forma. A tarefa τ_3 é cliente de dois pares de servidores (σ_1^1, σ_1^2) e (σ_2^1, σ_2^2), que são chamados de *shadow* (sombra). Eles são chamados assim porque a sombra é uma projeção que repete as ações de um objeto. O comportamento desses servidores é análogo ao da sombra porque eles executam ao mesmo tempo, mas a tarefa só pode estar executando em um processador (físico) que existe de verdade. O outro servidor só imita o que está acontecendo no processador físico, a fim reservar tempo de processamento e não permitir execução paralela de *jobs* de uma mesma tarefa. Sendo assim, os servidores σ_1^2 e σ_2^2 são alocados a P_3 enquanto seus servidores (*shadow*) correspondentes são atribuídos a P_1 e P_2. As decisões de escalonamento tomadas no processador lógico (P_3) sempre antecedem as decisões tomadas nos processadores os quais ele está relacionado (P_1 e P_2). Essa restrição garante que os *deadlines* nos processadores lógicos não sejam violados, sem impacto no escalonamento dos outros processadores, uma vez que há o compartilhamento de *deadlines* entre os servidores. Sempre que um dos servidores de P_3 executa, servindo τ_3, na verdade ela estará executando num processador físico por σ_1^1 ou σ_2^1. Esses pares de servidores *shadow* são definidos de forma que não mais que 1/3 do processador é reservado. Esse esquema proposto provê uma abstração equivalente a dividir τ_3 em duas sub-tarefas, τ_3' e τ_3''.

4.1 *SERVER SHADOWING*

Como mencionado anteriormente (retomar Figura 4.1), os servidores são utilizados em pares no PSSA, os quais são denominados servidores *shadow*. *Shadow* é a palavra na língua inglesa para sombra, justificando o motivo de serem um par, pois um é a sombra do outro. O par de servidores *shadow* é representado por σ_i^1 e σ_i^2 sendo i o índice do processador P_i que dá origem a eles, e é onde σ_i^1 está alocado. Como será visto mais adiante, σ_i^2 é atribuído a um processador P_j, $j > i$, com P_j sendo necessariamente lógico.

Mais precisamente, os servidores *shadow* são definidos da seguinte forma.

Definição 4.1.1 (Servidores shadow)**.** *Servidores shadow são servidores de taxa fixa, com comportamento similar àquele descrito na Definição 2.3.1 no que se refere à definição de seus jobs, à maneira como os servidores servem seus clientes, escalonados por EDF, e ao fato de que um servidor herda todos os deadlines de seus clientes. No entanto, servidores shadow possuem algumas regras adicionais:*

R1 Servidores shadow são concebidos em pares, formado por um servidor denominado do

tipo 1 e outro do tipo 2;

R2 Um servidor shadow, do tipo 1 ou do tipo 2, quando alocado a um processador lógico, consome seu budget quando executa, mas não executa nenhum cliente;

R3 Servidores do tipo 2 somente podem ser alocados a processadores lógicos enquanto que os do tipo 1 podem ser alocados a processadores lógicos ou físicos;

R4 Se σ^1 e σ^2 são um par de servidores shadow, ambos atendem ao mesmo conjunto de clientes e possuem a mesma utilização, que não necessariamente é igual à soma das utilizações de seus clientes;

R5 Servidores shadow do tipo 2 alocados ao mesmo processador lógico P compartilham todos os clientes alocados a P.

O comportamento dos servidores *shadow* ficará mais evidente durante a descrição de PSSA. Em resumo, as novas regras são necessárias para permitir que dois servidores sejam conectados (Regra R1) e possam imitar o comportamento um do outro em processadores distintos. Para tanto eles compartilham os mesmos clientes e, por conseguinte, compartilham os mesmos deadlines, que herdam dos seus clientes, tal como um servidor de taxa fixa (Regra R4). Como processadores lógicos são apenas para fornecer uma abstração, não pode haver execução real em tais processadores (Regra R2). Um conjunto de servidor do tipo 2 é o que forma a abstração de processadores lógicos (Regra R3). Por fim, como tais processadores contém entidades (tarefas ou servidores) que migram entre processadores físicos, há a necessidade de fazer com que as decisões de escalonamento em cada processador (por EDF) sejam compatíveis de forma que as tarefas migratórias possam ser tratadas adequadamente. Neste contexto, os *deadlines* são compartilhados entre os servidores *shadow* do tipo 2 (Regra R4).

4.2 PARTICIONAMENTO E ALOCAÇÃO DE SERVIDORES

Seguindo a estratégia semi-particionada, as tarefas de um conjunto devem ser previamente alocadas em processadores que terão a responsabilidade de escalonar as tarefas a eles designadas. A diferença do semi-particionamento para um esquema particionado é a possibilidade da tarefa ter suas partes alocadas em mais de um processador, caso uma tarefa não caiba de forma inteira em nenhuma das unidades de processamento. No semi-particionamento, existe a vantagem de ter tarefas que possuem seus processadores dedicados e não migram, evitando os *overheads* inerentes à migração. Ademais, as tarefas divididas ampliam a capacidade do algoritmo de lidar com conjuntos cuja utilização total do sistema é mais elevada, superando o baixo limite de 50% da estratégia particionada convencional.

Atribuir um conjunto Γ de tarefas em m processadores se assemelha ao problema do empacotamento ou *bin-packing*. Cada tarefa τ possui utilização $U(\tau) \leq 1$ a ser acomodada por um processador P de capacidade 1. Visto que o *bin-packing* é conhecido por ser um problema de otimização pertencente à classe de não polinomiais, são utilizadas heurísticas como *first-fit*, *best-fit*, *worst-fit*, *etc* a fim de atribuir os itens aos *bins*. No caso do PSSA, tarefas aos processadores. Contudo, a necessidade da divisão de tarefas surge da premissa que nem sempre ao final da execução do empacotamento todas as tarefas estarão atribuídas aos processadores. Algoritmos particionados se limitam aos 50% de utilização por estarem restritos a executar o *bin-packing* uma só vez. PSSA executa o empacotamento quantas vezes forem necessárias enquanto existir tarefas não alocadas. No contexto do PSSA, isso só é possível porque, quando o empacotamento

não atribui todas as tarefas, ele é capaz de gerar novos processadores lógicos criados a partir dos servidores *shadow* do tipo 2. *Shadows* esses que são oriundos de mais de um processador. Os novos processadores lógicos são considerados em uma execução posterior do *bin-packing* para que nenhuma tarefa fique sem processador. Note que a capacidade de um processador lógico é advinda de *shadows* de unidades de processamento distintas. Isso significa que atribuir tarefa a um processador lógico equivale a dividi-la em mais de um processador. Portanto, PSSA atribui partes de uma tarefa entre várias unidades computacionais através da sua alocação como cliente de servidores *shadow*. Em outras palavras, alocando em um processador lógico que contém esses servidores.

Retomando o exemplo base de 3 tarefas com 2/3 de utilização cada e 2 processadores, é fácil perceber que ao alocar uma tarefa a um dos processadores não é possível alocar uma segunda tarefa ao mesmo processador. Isso porque o processador estaria ocupado em 2/3 restando somente 1/3; capacidade insuficiente para abrigar nenhuma das tarefas restantes. Restando 2 tarefas pendentes de alocação, ainda há uma unidade de processamento que não possui nenhuma tarefa, dando a oportunidade para uma tarefa de 2/3 ser atribuída. A partir desse momento, a situação é idêntica em ambos os processadores: uma tarefa com utilização 2/3 alocada e 1/3 disponível. Porém, o conjunto é composto por 3 tarefas. Ainda existe uma tarefa de 2/3 que não consegue ser alocada em nenhum dos processadores do sistema, já que $1/3 < 2/3$. Se fosse em um algoritmo particionado, essa situação seria o seu término, mostrando a impossibilidade de escalonar esse conjunto. Entretanto, esse conjunto é viável, visto que $U(\Gamma) \leq m$. Isso pode ser percebido pela capacidade sobressalente do sistema, que é justamente o que a terceira tarefa (τ_3) precisa ($1/3 + 1/3 = 2/3$). O detalhamento de como PSSA consegue particionar esse tipo sistema será apresentado a seguir.

O Algoritmo 1 descreve como as tarefas do sistema são atribuídas aos processadores, possivelmente com a criação de processadores extras (lógicos) e alocações de servidores *shadow*. O algoritmo usa uma rotina de `BinPacking` para atribuir tarefas (itens) aos processadores (*bins*). PSSA utiliza como principal heurística de *bin-packing* o *first-fit*, considerando que as tarefas são empacotadas em ordem decrescente de utilização. Inicialmente, há uma tentativa de atribuir todas as tarefas aos processadores (Linha 2). Se for encontrado um particionamento sem a necessidade de tarefas migratórias, $\mathcal{T}$ torna-se vazio e o particionamento está finalizado. Por exemplo, quando um conjunto possui um número de tarefas igual ao de processadores ($m = n$), cada tarefa pode ser alocada a um processador e o sistema consegue ser particionado em uma só tentativa de empacotamento. Nesse caso PSSA é reduzido a *EDF particionado* (pEDF). Isto é, não é preciso criar os servidores *shadow* para dividir e gerenciar tarefas compartilhadas. Caso ainda restem tarefas pendentes de alocação após o primeiro empacotamento, porções das tarefas em $\mathcal{T}$ devem ser atribuídas a mais de um processador. No exemplo base de três tarefas com 2/3 de utilização, após a execução do *bin-packing*, τ_3 não foi alocada. Essa situação mostra a necessidade dessa tarefa não atribuída ser dividida entre os processadores, senão não seria possível alocá-la. Como um conjunto viável tem como requisito $U(\Gamma) \leq m$, sempre haverá folga (*slack*) em mais de um processador quando o *bin-packing* falhar em atribuir todas as tarefas. O exemplo base demonstra isso, com as folgas de 1/3 deixadas pelos dois processadores do sistema. A Função `slack`(P) denota a capacidade computacional que ainda resta no processador P, definida como

$$\texttt{slack}(P) = \begin{cases} 1 - \sum_{\tau \in P} U(\tau) & \text{Se } P \text{ é físico} \\ \sum_{\sigma \in P} U(\sigma) - \sum_{\tau \in P} U(\tau) & \text{Se} P \text{ é lógico} \end{cases}$$

Observe que a capacidade associada ao processador físico é 1 enquanto os processadores

lógicos têm suas capacidades fornecidas pelos servidores alocados neles. Isso é, se um processador lógico é composto por dois servidores *shadow* (do tipo 2) com taxas $U(\sigma_i^2) = 0.3; U(\sigma_{i+1}^2) = 0.5$, a capacidade do processador lógico P_j que abriga esses servidores é de 0.8.

A atribuição de tarefas migratórias é realizada nas linhas 5-18, através de sua atribuição aos processadores lógicos. A entrada no laço que inicia na Linha 5 demonstra a exigência da criação de um novo processador lógico, já que não foi possível alocar todas as tarefas na linha 2. Para cada novo processador lógico (linha 7), processadores com *slack* disponível em $\mathcal{P}$ são escolhidos e servidores *shadow* são alocados (Linhas 10-14). Para um processador P_i selecionado na iteração k, um par de servidores *shadow* (σ_i^1, σ_i^2) é criado. No exemplo base, para a criação dos pares de servidores *shadow* os processadores disponíveis são $\mathcal{P} = \{P_1, P_2\}$ com $\texttt{slack}(P_1) = \texttt{slack}(P_2) = 1/3$. Haja vista os *slacks* remanescentes, todos os servidores pertencentes a ambos os pares (σ_1^1, σ_1^2) e (σ_2^1, σ_2^2) recebem utilização igual a 1/3. Além disso, os servidores σ_i^1 são alocados no processador P_i que deu origem ao par através do seu *slack*. Já o servidor σ_i^2 é associado ao novo processador lógico P_{m+k} da iteração k, indicando assim que o processador lógico P_{m+k} utilizará o poder computacional de P_i pelo *slack* apontado por σ_i^1. Ao final dessa criação dos pares de *shadow* e associação de servidores aos processadores correspondentes (linha 10), são atribuídos ao novo processador P_{m+k} (P_3) σ_1^2 e σ_2^2. Somando as utilizações desses servidores atribuídos a P_3, a qual é igual a soma dos *slacks* de P_1 e P_2, o resultado é uma capacidade de 2/3. Como o P_3 é adicionado ao conjunto de processadores do sistema (linha 15), ele é mais um *bin* a ser considerado no empacotamento executado na linha 16. Após a execução desse novo *bin-packing*, τ_3 pode ser alocada com sucesso em P_3, já que $U(\tau_3) = 2/3$, sendo igual a capacidade de P_3. Observe que ao alocar τ_3 em P_3, τ_3 torna-se cliente de todos os servidores que fazem parte de P_3 de forma direta (σ_2^1, σ_2^2) ou indireta (σ_1^1, σ_1^2) por estarem em outros processadores. Tendo em vista que P_1 e P_2 ficaram responsáveis por uma tarefa cada um e ainda cederam 1/3 para execução de τ_3 em P_3 via σ^1, todas as tarefas foram alocadas com sucesso.

A Figura 4.1 ilustra o resultado do Algoritmo 1 para esse exemplo base de 3 tarefas e 2/3 de utilização cada. Resumindo os passos realizados, um processador lógico P_3 foi criado, ao qual τ_3 foi alocada. Dois pares de servidores *shadow* são responsáveis por reservar 1/3 de cada processador físico para executar τ_3.

Considere um exemplo um pouco mais complexo baseado num conjunto Γ com $n = 7$ tarefas a serem escalonadas em $m = 5$ processadores. A utilização de todas as tarefas $\tau_i \in \Gamma$ é de $U(\tau_i) = 5/7$. Partindo do pressuposto que o resultado da primeira etapa de particionamento é a atribuição de $\{\tau_i\}$ ao processador P_i, $i = 1, \ldots, 5$ (ver a Figura 4.2 como referência). Consequentemente, 2/7 é o *slack* deixado em cada processador físico antes do começo da linha 5. Ainda falta alocar $\mathcal{T} = \{\tau_6, \tau_7\}$, cuja atribuição é feita da seguinte forma. O primeiro processador lógico, denominado P_6, e três pares de servidores *shadow* são criados durante a primeira iteração das linhas 5-18. Os pares desses servidores *shadow*, denominados σ_i^1 e σ_i^2, com utilização de 2/7 cada, são atribuídos ao processador P_i e P_6 sendo $i = 1, 2, 3$ respectivamente. O novo processador P_6 (com capacidade 6/7) é então considerado na segunda etapa de particionamento (linha 16). Contudo, somente uma tarefa das que estão em $\mathcal{T}$ pode ser inteiramente atribuída a P_6. Nesse exemplo, a tarefa atribuída é τ_6. Diante disso, outro processador lógico é necessário para a alocação de $\mathcal{T} = \{\tau_7\}$, cujo conjunto de processadores disponíveis é $\mathcal{P} = \{P_4, P_5, P_6\}$. Durante a última etapa de particionamento, P_7 é criado junto com os servidores *shadow* σ_4^2, σ_5^2 e σ_6^2. Como as utilizações dos servidores $U(\sigma_4^2) + U(\sigma_5^2) + U(\sigma_6^2) = 5/7$ proveem a capacidade de P_7, τ_7 é atribuído a este processador, finalizando a rotina de atribuição.

É importante ressaltar que σ_6^1 e σ_6^2 estão ambos alocados a processadores lógicos. Sempre

Algoritmo 1: Particionamento e alocação de servidores.

Input: Um conjunto de tarefas Γ ($U(\Gamma) \leq m$) a serem atribuídos a um conjunto de processadores $\mathcal{P} = \{P_1, P_2, \ldots, P_m\}$

Output: Atribuições tarefa-processador, tarefa-servidor e servidores-processador

```
1  k ← 0 ;
2  BinPacking(Γ, P);
3  P ← P\{P ∈ P | slack(P) = 0};
4  T ← {τ ∈ Γ | τ não está alocada a um processador};
5  while T ≠ ∅ do
6      k ← k + 1;
7      Crie um processador lógico P_{m+k};
8      Escolha algum P_i ∈ P;
9      u ← slack(P_i);
10     while u ≤ 1 e P ≠ ∅ do
11         P ← P\{P_i};
12         Crie servidores shadow σ_i^1 e σ_i^2 com U(σ_i^1) = U(σ_i^2) = slack(P_i);
13         Atribua σ_i^1 para P_i e σ_i^2 para P_{m+k};
14         Escolha algum P_i ∈ P, e se há algum faça u ← u + slack(P_i);
15     P ← P ∪ {P_{m+k}};
16     BinPacking(T, P);
17     P ← P\{P ∈ P | slack(P) = 0};
18     T ← {τ ∈ Γ | τ não está alocada a um processador};
```

que σ_6^1 executa, entretanto, ele na verdade está sendo servido por σ_1^2 ou por σ_2^2 ou por σ_3^2. Isso quer dizer que a execução real de σ_6^1 (ou seja a porção de τ_7) está acontecendo nos processadores físicos P_1, P_2, ou P_3, respectivamente.

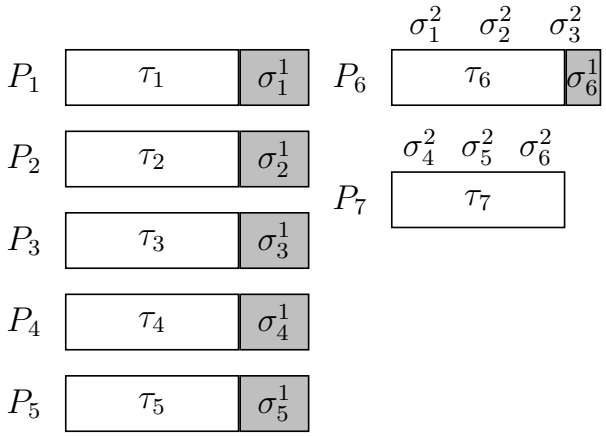

Figura 4.2 Possível resultado do particionamento de um conjunto com sete tarefas cada uma com utilização 5/7 em um sistema com cinco processadores. Dois processadores lógicos são criados, sendo que P_7 depende de P_6.

Esta seção termina com algumas definições, que serão úteis durante a demonstração de correção do esquema de escalonamento PSSA. Como pode ser observado nas ilustrações da Figura 4.2, o Algoritmo 1 produz uma hierarquia de processadores. Por exemplo, os primeiros processadores considerados (processadores físicos) estão na base desta hierarquia. Cada par de servidores shadow define uma relação nesta hierarquia. Por exemplo, na Figura 4.2, P_6 é visto num nível mais alto nesta hierarquia que os processadores físicos.

Definição 4.2.1 (Hierarquia de processadores)**.** *Considere o conjunto de processadores $\mathcal{P}$, físicos e lógicos, resultado da alocação do Algoritmo 1. Defina um grafo $G(V, E)$ com um conjunto V de vértices e um conjunto E de arestas da seguinte forma:*

- *Para cada $P_i \in \mathcal{P}$, crie um vértice $v_i \in V$;*
- *Para cada par de servidores shadow σ^1 e σ^2, definidos pelo Algoritmo 1, alocados respectivamente nos processadores P_i e P_j de $\mathcal{P}$, crie uma aresta $(v_i, v_j) \in E$.*

O grafo G assim definido é chamado de hierarquia de processadores e é caracterizado por um conjunto de árvores, cujas folhas representam os processadores físicos. O nível l de um vértice $v_i \in G$, ou equivalentemente de $P_i \in \mathcal{P}$, é a distância de v_i a um nó folha. Nós folhas estão no nível $l = 0$. Todos os vértices de G que não têm pai são denominados nós raiz de G e representam processadores raiz em $\mathcal{P}$.

A Figura 4.3 representa a única árvore de processadores associada ao exemplo da Figura 4.2.

4.3 GERAÇÃO DO ESCALONAMENTO

As decisões de escalonamento em cada processador, físico ou lógico, seguem EDF. Desta maneira, é escolhido para executar, a qualquer instante do escalonamento, o *job* da tarefa ou do

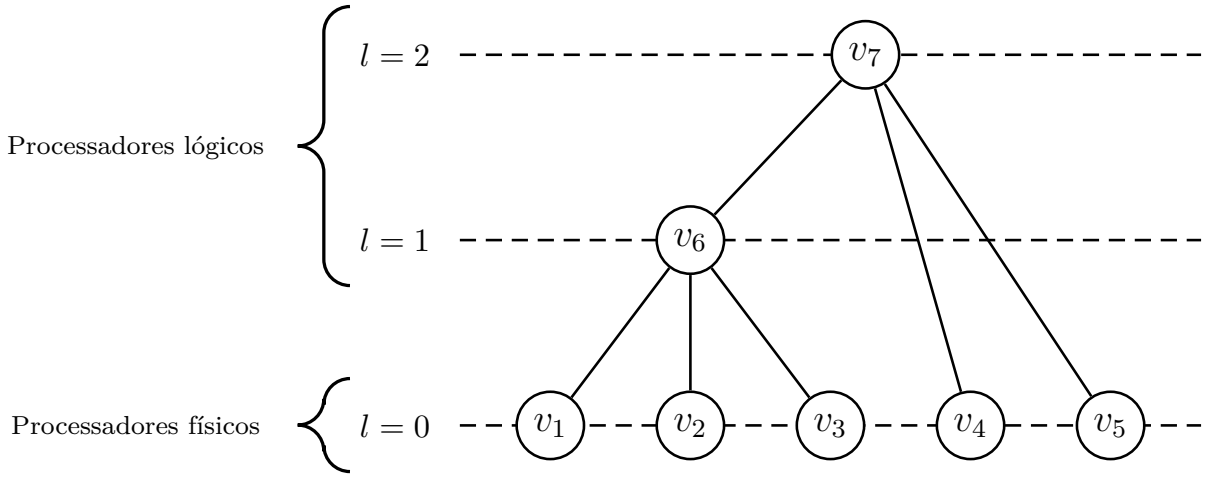

Figura 4.3 Grafo G, construído de acordo com a Definição 4.2.1, representando a hierarquia para o exemplo da Figura 4.2. Uma única árvore foi gerada para este exemplo.

servidor com o *deadline* mais próximo em qualquer processador. Todavia, as decisões devem seguir também uma ordem específica dada a necessidade dos servidores *shadow* de imitar um ao outro. Para o exemplo base da Figura 4.1, a decisão em P_3 deve ser tomada antes das decisões em P_1 e P_2, já que τ_3 não pode ser escalonado ao mesmo tempo em diferentes processadores. De forma similar, para o exemplo da Figura 4.2, as decisões de escalonamento em P_7 devem preceder aquelas tomadas em P_6, que por sua vez também devem anteceder as decisões dos demais processadores. Em outras palavras, as decisões de escalonamento devem seguir a ordem, da raíz para as folhas, na hierarquia de processadores. Para mostrar a necessidade dessa ordenação, observe o escalonamento da Figura 4.4. Se em um dado instante P_1 é escolhido para executar τ_1 e P_3 escolhe σ_1^2, suas decisões de escalonamento estariam em conflito. Entretanto, o problema é resolvido selecionando primeiro σ_1^2 em P_3 para depois tomar a decisão de escalonamento em P_1. Ao escalonar σ_1^2 em P_3, força-se a execução de σ_1^1 para que τ_3 seja executada em P_1. Só é possível realizar o escalonamento de τ_3 em P_1 levando em conta que σ_1^2 ao σ_1^1 compartilham os mesmos clientes. Como σ_1^1 contém os *deadlines* de τ_1, em qualquer instante de escalonamento t, ele segue $D(\sigma_1^1, t) \leq D(\tau_1, t)$, garantindo a compatibilidade com a ordem EDF em P_1. O Algoritmo 2 define como o escalonador deve funcionar em tempo de execução.

Notar ainda as linhas 5 e 8 do Algoritmo 2, forçam a execução paralela do par (σ_1^1,σ_1^2). Perceba que o Algoritmo 2 comporta-se diferente dependendo se o processador é físico ou lógico. Tem que existir essa distinção porque o processador lógico depende da escolha de um servidor *shadow* σ^2 (Linha 3) para executar seus clientes em uma ociosidade mapeada por um σ^1 associado. É importante ressaltar que nem sempre σ^2 tem uma tarefa como cliente. Quando um processador lógico depende de outro processador lógico, σ^2 pode ter um σ^1 como cliente, desde que a condição da linha 5 seja satisfeita. Contudo, no final do processo, sempre quando um processador lógico escalonar um σ^2, algum σ^1 obrigatoriamente executa em um processador físico (linha 8. Lembre-se que os servidores *shadows* existem para fazer um mapeamento até um processador físico ocioso. Por isso, a execução de processadores lógicos é só um artifício que culmina no escalonamento de uma tarefa num processador físico. No processador físico, tanto uma tarefa atribuída diretamente a ele (linha 10) quanto um servidor σ^1 (linha 8) podem ser escalonados. Este último caso ocorre se seu σ^2 correspondente for escolhido por um processador

Algoritmo 2: Escalonador PSSA

Input: Um conjunto de tarefas e servidores atribuídos a m processadores físicos e k processadores lógicos pelo Algoritmo 1 com P_{m+k} o último processador lógico criado, se existir.

Output: A decisão de escalonamento a ser tomada em cada processador a qualquer instante de escalonamento t.

```
1  for i ← m + k, ... 1 do
2      if P_i é um processador lógico then
3          Seleciona por EDF um servidor shadow σ² ∈ P_i com budget positivo, se
           houver;
4          if existe um servidor σ_i^1 ∈ P_i cujo servidor shadow σ_i^2 já foi escolhido em
           P_j, j > i, then
5              Escolhe σ_i^1 (como cliente de σ²);
6      else
7          if existe um servidor σ_i^1 ∈ P_i com seu shadow σ_i^2 já selecionado then
8              Seleciona o servidor σ_i^1;
9          else
10             Seleciona por EDF uma tarefa pronta τ ∈ P_i, se houver;
11     while a entidade escolhida σ em P_i não é uma tarefa do
12         σ ← o mais prioritário (por EDF) cliente de σ
13 Despacha para execução todas as tarefas selecionadas;
```

lógico. Caso σ^1 seja escalonado, é escolhida a tarefa cliente mais prioritária pertencente ao processador lógico que solicitou a sua capacidade computacional. Depois de escolher por EDF todos os possíveis servidores, clientes e tarefas, eles são despachados para execução no sistema.

A Figura 4.4 mostra o escalonamento do exemplo das três tarefas durante o intervalo $[0, 6)$. Todas as três tarefas liberam seus primeiros *jobs* no instante 0. É assumido que $T_1 = 3$ e $T_2 = T_3 = 6$. Como pode ser observado, as escolhas de escalonamento em P_3 dita onde τ_3 executa. σ_1^2 executa antes de σ_2^2 devido ao fato que $D(\sigma_1^2, 0) = 3 < D(\sigma_2^2, 0) = 6$. Isso ocorre porque os *deadlines* de σ_1^2 e σ_2^2 são os mesmos, já que os *deadlines* de seus servidores *shadow*, σ_1^1 e σ_2^1 herdam os *deadlines* das tarefas atribuídas a P_1 e P_2, respectivamente, como necessário. Veja que a tarefa τ_3 migra entre P_1 e P_2 para concluir a sua execução. Durante $[0, 1)$ τ_3 executa em P_1 σ_1^1 pela escolha de σ_1^2 em P_3 nesse mesmo instante. Ao finalizar o *budget* de 1/3 do par $(\sigma_1^1, sigma_1^2)$, τ_3 precisa migrar para P_2. Em $[1, 2)$ ocorre o mesmo cenário de $[0, 1)$, mudando o processador de P_1 para P_2. No instante 3 chega o *deadline* dos *jobs* disparados em 0 para os *shadows*, assim recarregando o *budget* de todos os servidores. Isso reforça o compartilhamento dos *deadlines* entre os *shadows* porque o par (σ_2^1, σ_2^2) possuem tarefas com *deadline* 6 (τ_2 e τ_3). Entretanto, os *jobs* são liberados com *deadline* 3, porque σ_1^2 está no mesmo processador lógico que σ_2^2, e σ_1^2 tem τ_1 como cliente com *deadline* 3. Sendo assim, 3 é o *deadline* mais próximo para limitar a execução dos pares (σ_1^1, σ_1^2) e (σ_2^1, σ_2^2). O comportamento de gerenciar o escalonamento de *shadows* para executar seus clientes em vários processadores físicos distintos é o que garante a função de sincronização dos processadores lógicos. Através dessa coordenação, uma tarefa

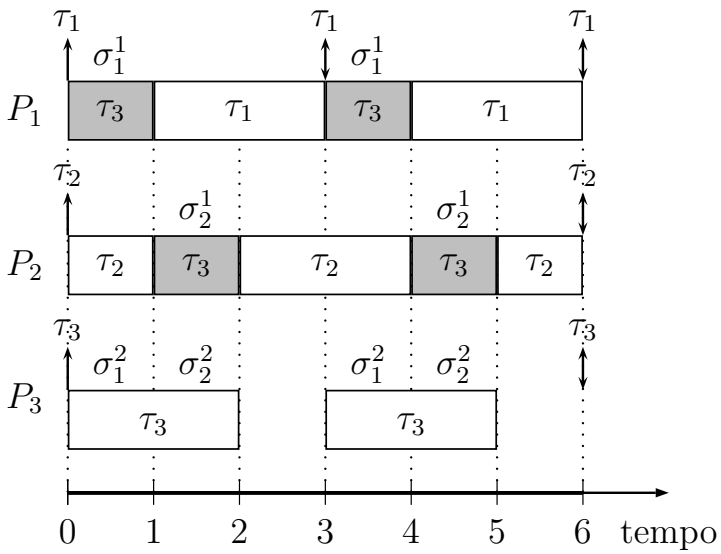

Figura 4.4 Escalonamento produzido pelo PSSA para um conjunto de três tarefas periódicas Γ com $C_1 = 2$, $C_2 = 4$, $C_3 = 4$, $T_1 = 3$, $T_2 = T_3 = 6$.

migratória jamais executa em paralelo em dois processadores físicos distintos. Esse exemplo ilustra isso, demonstrando que a execução da tarefa migratória tem que ser interrompida em um processador físico para ser escalonada por outro.

O escalonamento gerado pelo PSSA para o exemplo mais complexo com 7 tarefas de utilização 5/7 em 5 processadores pode ser visualizado na Figura 4.5. A prioridade dos processadores com maiores índices fica evidente ao ver que τ_7 não é interrompida do início da sua execução até o término da sua demanda computacional. Porém, perceba que o escalonamento de P_7 é fictício, sendo τ_7 escalonado por P_4, P_5 e P_6, respectivamente. Portanto, mesmo que a representação seja uma execução sem interrupções, na verdade a tarefa está migrando entre os processadores que podem executá-la por estarem ociosos. O ponto mais importante que difere esse exemplo do anterior é a relação entre os processadores lógicos P_6 e P_7. Na atribuição, foi mencionado que há uma ociosidade de P_6 depois do empacotamento realizado. A utilização dessa ociosidade representada por σ_6^1 é percebida no intervalo $[4, 5)$, quando P_7 solicita a capacidade computacional através do escalonamento de σ_6^2. Observe que, ao escalonar σ_6^2, P_7 obriga P_6 a escolher σ_6^1 (linha 5) como cliente de um σ^2 escolhido na Linha 3. No caso do escalonamento da Figura 4.5, o *shadow* σ^2 escolhido foi σ_3^2. Logo, quem executa a tarefa τ_7 em $[4, 5)$ é P_3, já que τ_7 também é cliente de σ_3^1.

Como pode ser observado na ilustração das Figuras 4.4 e 4.5, tarefas alocadas a processadores físicos não migram durante execução enquanto que aquelas atribuídas a processadores lógicos executam em mais de um processador físico. Por conveniência, para distingui-las, estas tarefas serão chamadas de não migratórias e migratórias, respectivamente.

4.4 PROVA DE CORREÇÃO

Como visto nas seções anteriores, PSSA é composto por duas partes. A primeira parte acontece antes das tomadas de decisões do escalonamento, a qual é chamada de *off-line*. O processamento *off-line* é composto pela atribuição das tarefas aos seus processadores, bem como a criação de

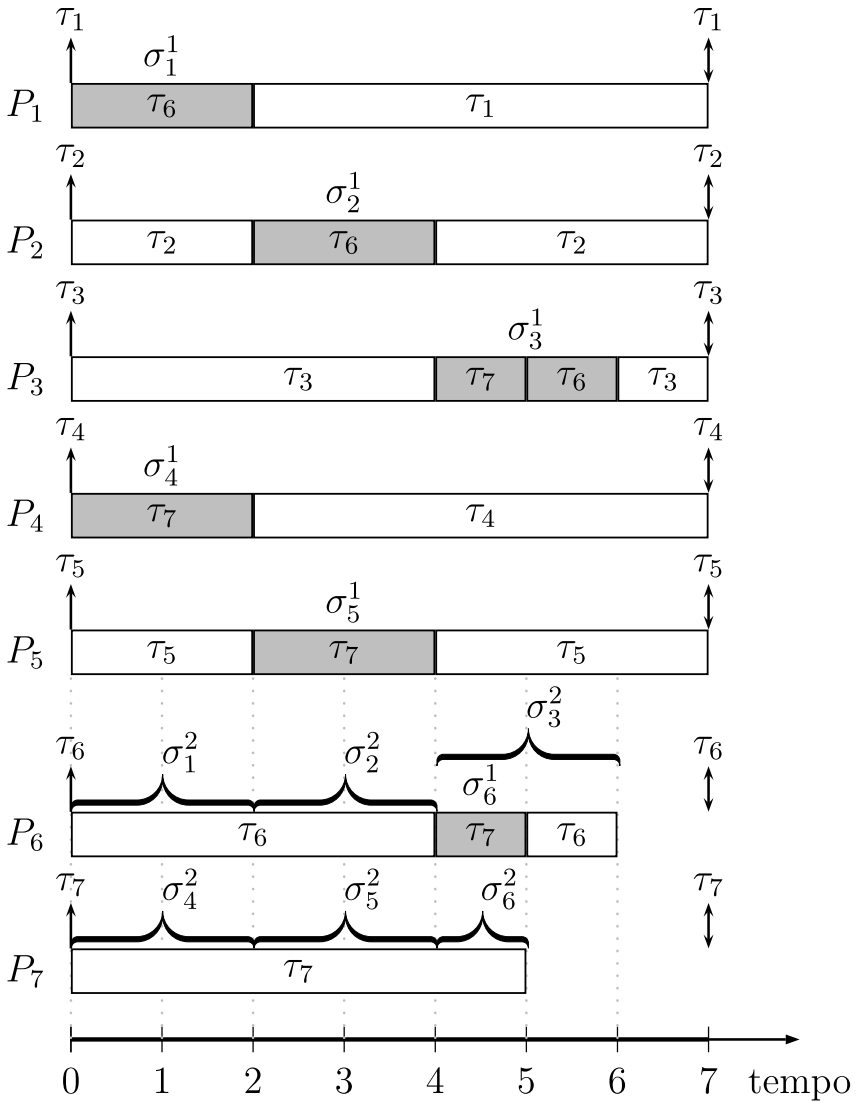

Figura 4.5 Escalonamento produzido pelo PSSA para um conjunto de 7 tarefas periódicas Γ a serem escalonadas por 5 processadores. Todas com $C = 5$, e $T = 7$. Dois processadores lógicos são criados (P_7 e P_6) e P_7 possui o *shadow* σ_6^2 criado por P_6.

novos processadores (lógicos) caso necessário. Dado um conjunto de tarefas viável Γ ($U(\Gamma) \leq m$), cabe ao Algoritmo 1, apresentado na Seção 4.2, atribuir todas as tarefas de modo que não reste nenhuma pendente de alocação. A habilidade de distribuição integral das tarefas entre os processadores de um sistema pelo Algoritmo 1 é assegurada, como demonstrado a seguir.

Lema 4.4.1. *Considere Γ um conjunto de tarefas, com $U(\Gamma) \leq m$, alocadas a processadores pelo Algoritmo 1. Seja P_{m+k} um processador lógico criado pelo algoritmo na linha 7. Considere $\mathcal{T}$ o conjunto de tarefas ainda não alocadas no início do laço da linha 10. Se as tarefas de Γ são alocadas a processadores em ordem decrescente de utilização nas linhas 2 e 16,*

$$\max_{\tau \in \mathcal{T}} U(\tau) \leq \sum_{\sigma_i^2 \in P_{m+k}} U(\sigma_i^2)\,, \tag{4.1}$$

Demonstração. Seja $\tau \in \mathcal{T}$ a tarefa com maior utilização e ainda não alocada no momento em que P_{m+k} é criado na linha 7. Como $\mathcal{T} \neq \emptyset$, sabe-se que para todo $P_i \in \mathcal{P}$, $\texttt{slack}(P_i) < 1$; caso contrário τ poderia ter sido alocada em P_i. Isto implica que $u < 1$ após a execução da linha 9. Além disso, como há tarefas (em $\mathcal{T}$) que não foram alocadas na linha 2 e $U(\Gamma) \leq m$, sabe-se que $\mathcal{P} \neq \emptyset$, o que permite que o laço da linha 10 execute. Adicionalmente, note que antes da primeira iteração deste laço, $U(\tau) > u$, caso contrário τ poderia ter sido alocada anteriormente no processador escolhido na linha 8. Isso significa que há $p \geq 2$ iterações do laço da linha 10. Não é difícil notar que p é finito, pois o valor de u cresce a cada iteração e o número de processadores em $\mathcal{P}$ é finito. Portanto, ao final da p-ésima iteração do laço da linha 10, a capacidade de P_{m+k} será formada pelos p servidores *shadow* alocados a ele na linha 13. Sem perda de generalidade, considere que $P_{i+1}, P_{i+2}, \ldots, P_{i+p}$ foram os processadores em $\mathcal{P}$ escolhidos para a definição dos servidores *shadow*. Desta forma,

$$\sum_{\sigma_i^2 \in P_{m+k}} U(\sigma_i^2) = \sum_{j=1}^{p} \texttt{slack}(P_{i+j})\,. \tag{4.2}$$

Suponha agora por contradição que a relação (4.1) não procede. Há dois cenários a considerar, dependendo da condição que faz o laço da linha 10 terminar, que pode ser ou devido a $\mathcal{P} = \emptyset$ ou se $u > 1$:

1. $\mathcal{P} = \emptyset$. Neste caso, como não há mais processador com *slack* disponível, toda a capacidade dos m processadores já foi alocada. O fato de a tarefa τ ainda estar sem alocação significa que $U(\Gamma) > m$, uma contradição.

2. $u > 1$. Isto significa que há processador P_{i+p+1} que faz

$$\sum_{\sigma_i^2 \in P_{m+k}} U(\sigma_i^2) + \texttt{slack}(P_{i+p+1}) > 1\,. \tag{4.3}$$

   Por hipótese (de contradição), (4.3) leva a

$$U(\tau) + \texttt{slack}(P_{i+p+1}) > 1\,. \tag{4.4}$$

   Usando a igualdade (4.2), chega-se a

$$\sum_{j=1}^{p} \texttt{slack}(P_{i+j}) > 1 - \texttt{slack}(P_{i+p+1}) \tag{4.5}$$

Se P_{i+p+1} é um processador físico,

$$\sum_{\tau_l \in P_{i+p+1}} U(\tau_l) = 1 - \texttt{slack}(P_{i+p+1})\,.$$

Se P_{i+p+1} é um processador lógico,

$$\sum_{\tau_l \in P_{i+p+1}} U(\tau_l) \leq 1 - \texttt{slack}(P_{i+p+1})\,.$$

Em ambos os casos, a relação (4.4) implica que

$$U(\tau) > \sum_{\tau_l \in P_{i+p+1}} U(\tau_l).$$

Isto significa que houve alocação de tarefa τ_l ao processador P_{i+p+1} com utilização menor que $U(\tau)$ antes de τ ser considerada para alocação, contradizendo o fato de que as tarefas são alocadas em ordem decrescente de utilização.

∎

Lema 4.4.2. *Seja Γ um conjunto de tarefas, com $U(\Gamma) \leq m$. Se as tarefas são alocadas pelo Algoritmo 1 a processadores em ordem decrescente de suas utilizações, nas linhas 2 e 16, cada uma das tarefas de Γ é alocada a apenas um processador e todas as tarefas são alocadas.*

Demonstração. O lema se verifica de maneira trivial caso todas as tarefas de Γ sejam alocadas a processadores físicos na linha 2. Considere então que, após a execução da linha 2, $\mathcal{T} \neq \emptyset$, o que satisfaz a condição do laço da linha 5. Suponha por contradição que Algoritmo 1 falha ao alocar tarefas a processadores. Há dois casos a considerar:

- Alguma tarefa τ não pode ser alocada a um processador lógico P_{m+k}. Pela ordem de alocação, τ é a tarefa com maior utilização ainda não alocada a algum processador. Como a capacidade do processador P_{m+k} é dada pela soma das utilizações dos servidores *shadow* do tipo 2 nele alocados, sabe-se que $U(\tau) > \sum_{\sigma^2 \in P_{m+k}} U(\sigma^2)$, contradizendo o Lema 4.4.1.

- O processador P_{m+k} não pode ser criado. Isto significa que o laço da linha 10 termina com $\mathcal{P} = \emptyset$. Portanto, todos os processadores físicos e lógicos, estes criados antes de P_{m+k}, não possuem capacidade suficiente para alocação de τ. Este cenário só é possível se $U(\Gamma) > m$, caso em que o Γ não é escalonável em m processadores e o Algoritmo 1 não se aplica.

∎

Sabendo-se que o Algoritmo 1 consegue atribuir corretamente qualquer conjunto de tarefas viável, ainda há a necessidade saber se, dada esta alocação, o Algoritmo 2 sempre garante o cumprimento de todos os *deadlines*. Este aspecto é tratado a seguir.

Inicialmente, é interessante notar que um conjunto de servidores num mesmo processador podem compartilhar um conjuntes sem causar perda de *deadlines* destes, contanto que a utilização dos servidores seja adequada.

Lema 4.4.3. *Seja $\mathcal{C}$ um conjunto de clientes de um servidor de taxa fixa σ alocado a um processador P, com $U(\sigma) = \sum_{\sigma_i \in \mathcal{C}} U(\sigma_i)$. Se as decisões de escalonamento em P estão de acordo com EDF e σ cumpre todos os seus deadlines, então nenhum cliente em $\mathcal{C}$ perde deadline quando escalonados por um conjunto de servidores de taxa fixa alocados a P contanto que $U(\sigma) = \sum_{\sigma_i \in \mathcal{S}} U(\sigma_i) \leq 1$.*

Demonstração. Pela Definição 2.3.1, se cada servidor em $\mathcal{S}$ serve o mesmo conjunto $\mathcal{C}$ de clientes, então tais servidores compartilham os mesmos *deadlines*, que incluem todos os *deadlines* dos *jobs* de seus clientes. As decisões de escalonamento em P, portanto, são arbitrárias quanto à seleção de qual servidor em $\mathcal{S}$ (com *budget* positivo) executar. Além disso, os *jobs* dos servidores de $\mathcal{S}$ são liberados no mesmo instante que os *jobs* de σ são liberados e possuem os mesmos *deadlines* que σ. Como $U(\sigma) = \sum_{\sigma_i \in \mathcal{S}} U(\sigma_i)$, isso significa que os clientes em $\mathcal{C}$ executam exatamente nos mesmos instantes, independentemente se estão sendo escalonados por σ ou conjuntamente pelos servidores em $\mathcal{S}$. Isto significa que se algum deadline for perdido no escalonamento por σ, ele também seria perdido no escalonamento gerado pelos servidores de $\mathcal{S}$. Sabe-se por hipótese que σ cumpre todos seus *deadlines* em P e, portanto, o Teorema 2.3.1 garante que todos os clientes em $\mathcal{C}$ cumprem seus *deadlines* quando escalonados por σ em P. Portanto, não haverá nenhum deadline de cliente perdido no escalonamento de $\mathcal{C}$ gerado pelos servidores em $\mathcal{S}$ ∎

O Lema 4.4.3 é geral no sentido em que não especifica o tipo de processador, se lógico ou físico. O resultado seguinte passa a considerar especificidades associadas a processadores lógicos.

Lema 4.4.4. *Sejam P_i e P_{m+k} dois processadores, P_{m+k} pai de P_i, criado durante o laço da linha 5 do Algoritmo 1 para algum $k > 0$. No escalonamento gerado para P_i pelo Algoritmo 2, se nenhum servidor alocado a P_{m+k} perde qualquer dos seus deadlines, então tanto servidores quanto tarefas não migratórias, que estejam alocados a P_i, cumprem todos seus deadlines.*

Demonstração. Como P_{m+k} é pai de P_i, pelo Algoritmo 1, há apenas um servidor σ^1 alocado em P_i, criado pelo Algoritmo 1 na linha 12, juntamente com um servidor σ^2, alocado a P_{m+k} na linha 13. Há dois casos a se analisar:

- P_i é um processador físico. Neste caso, além de σ^1, apenas tarefas não migratórias foram alocadas a P_i, na linha 2 do Algoritmo 1. Pela Definição 4.1.1: (a) os *deadlines* de σ^1 são os mesmos de σ^2; (b) σ^1 e σ^2 têm a mesma utilização (linha 12 do Algoritmo 1); (c) os *deadlines* de σ^1 e σ^2 contemplam todos os *deadlines* das tarefas não migratórias alocadas a P_i. Além disso, (d) σ^1 e σ^2 são escalonados concomitantemente (pelas linhas 3-5 do Algoritmo 2). Devido aos fatos (a), (b) e (d), como σ^2 cumpre todos seus *deadlines*, σ^1 também cumpre os seus *deadlines*. Note ainda que (d) é compatível com as decisões de escalonamento tomadas por EDF em P_i devido a (a) e (c). Portanto, a otimalidade de EDF implica que as tarefas não migratórias alocadas a P_i também cumprem seus *deadlines*.

- P_i é um processador lógico. Neste caso, σ^1 é cliente de servidores do tipo 2 em P_i. De fato, pela Definição 4.1.1, qualquer servidor do tipo 2 alocado a P_i contém o mesmo conjunto de clientes e todos estes servidores compartilham seus *deadlines*. Desta forma, quando σ^1 é forçado a executar em P_i pela seleção de σ^2 em P_{m+k}, qualquer um dos servidores do tipo 2 em P_{m+k} pode ser selecionado (por EDF) pelo Algoritmo 2, juntamente com σ^1 quando o processador P_i é visitado. Desta forma, similarmente ao caso anterior, σ^1

cumprirá seus *deadlines* pois σ^2 cumpre os seus, eles executam em paralelo e nenhum outro servidor do tipo 2 alocado a P_i perde deadline (por otimalidade de EDF).

∎

Teorema 4.4.5. *Qualquer conjunto viável de tarefas periódicas, independentes e com deadlines implícitos, quando escalonados por PSSA cumprem todos os seus deadlines.*

Demonstração. Considere um sistema com m processadores físicos. Como Γ é viável por hipótese, $U(\Gamma) \leq m$ e, portanto, o Lema 4.4.2 assegura que todas as tarefas de Γ são alocadas em até m processadores físicos e, caso necessário, em até $\kappa \geq 0$ processadores lógicos. Seja $\mathcal{P} = \{P_1, P_2, \ldots, P_{m+\kappa}\}$ o conjunto de processadores aos quais tarefas/servidores foram alocados pelo Algoritmo 1. Se $\kappa = 0$, PSSA se comporta essencialmente como EDF particionado. Neste caso particular, o teorema se verifica trivialmente devido à otimalidade de EDF.

Para o caso de $\kappa > 0$, a demonstração se dará em duas partes. A primeira, por indução usando o nível do processador na hierarquia de processadores, terá como foco as tarefas não migratórias e os servidores. A segunda parte tratará das tarefas migratórias.

Parte 1. Nenhum servidor ou tarefa não migratória perde quaisquer de seus deadlines. Por indução no nível l dos processadores. Para o caso base, considere todos os processadores em $\mathcal{P}$ que estão no nível raiz na hierarquia de processadores. Sem perda de generalidade, seja P um destes processadores. Assuma por contradição que algum servidor nele alocado perde seu deadline num instante d num escalonamento Σ. Como P é um processador raiz, P é um processador lógico e não contém nenhum servidor do tipo 1. A capacidade de P é $U(P) \leq 1$, o que corresponde à soma das utilizações de todos os servidores do tipo 2 alocados a P, que, pela linha 10 do Algoritmo 1, não ultrapassa 1. Considere $\mathcal{S}$ o conjunto destes servidores alocados a P e seja σ um servidor de taxa fixa tal que $U(\sigma) = U(P)$. Faça $\mathcal{S}$ clientes de σ. O escalonamento Σ' gerado para $\mathcal{S}$ por σ é exatamente igual a Σ, pois P é escalonado por EDF (mesma política de σ) e σ executa sem concorrência em P. Portanto, o deadline d também é perdido em Σ'. Além disso, a ausência de concorrência e o fato de $U(\sigma) \leq 1$ implica que σ cumpre todos os seus *deadlines* em P, o que contradiz o Teorema 2.3.1

Para o passo de indução, assuma que servidores alocados a processadores nos níveis $l > 0$ cumprem seus *deadlines*. Como resultado do Lema 4.4.4, sabe-se que nem servidores alocados a processadores no nível $l - 1 > 0$ nem tarefas ou servidores alocados em processadores no nível $l - 1 = 0$ perdem seus *deadlines*.

Parte 2. Nenhuma tarefa migratória perde deadline. Tarefas migratórias são aquelas atribuídas a algum processador lógico P. Sem perda de generalidade, considere tal processador P, formado por um conjunto de servidores do tipo 2 criado pelo Algoritmo 1. Sabe-se pelos argumentos apresentados na parte 1 acima que todos os servidores cumprem seus *deadlines*. Como servidores em P forçam a execução dos servidores correspondes do tipo 1, pode-se conceber que o escalonamento em P é uma projeção dos escalonamentos gerados nos processadores físicos para as tarefas migratórias (ver Figura 4.5 para ilustração). Desta forma, pode-se analisar o escalonamento em P como se este fosse um processador físico. Usando esta abstração, sabe-se que os servidores em $\mathcal{S}$ são escalonados pelo Algoritmo 2 por EDF em P. Sabe-se ainda que os servidores em $\mathcal{S}$ compartilham clientes e contêm todos os seus *deadlines*. Desta forma, o mesmo escalonamento que os servidores de $\mathcal{S}$ geram para seus clientes pode ser obtido por um único servidor σ com utilização $U(\sigma) = \sum_{\sigma^2 \in \mathcal{S}} U(\sigma^2)$, que seria executado em P sem concorrência para escalonar seus clientes. Portanto, pelo Teorema 2.3.1, sabe-se que todos os clientes de σ (que são os mesmos dos servidores em $\mathcal{S}$) cumpririam todos os seus *deadlines*. ∎

Capítulo 5

AVALIAÇÃO

Para validar a correção do algoritmo e obter resultados a fim de comparar com outras propostas da literatura, será utilizada uma metodologia experimental que requer a implementação do PSSA. Uma opção é implementar o algoritmo em um sistema operacional de tempo real, propiciando a obtenção do comportamento do escalonador diante de um ambiente real. Contudo, o custo para a realização de experimentos como esse é muito alto, pois necessita-se equipamentos e instrumentos reais. Por exemplo, se for necessário testar como o algoritmo se comporta em um sistema com 30 processadores, essa quantidade elevada e incomum de unidades de processamento teria que estar à disposição. Além disso, essa modalidade de implementação gera muitas variações no experimento (JAIN, 1990), devido à distinção de comportamento tanto do *hardware* quanto do *software*. No caso do escalonamento de tempo real, o comportamento do processador, a utilização da memória cache e outras variáveis são exemplos do que pode provocar interferência nos resultados obtidos no experimento. Isso pode ser visto como um ponto positivo, considerando que essas instabilidades testam o algoritmo diante de condições apropriadas e adversas. Entretanto, existe também vantagem em lidar com experimentos que ao serem replicados geram o mesmo resultado, principalmente na construção de um novo algoritmo.

A outra opção para implementar o algoritmo é em um ambiente de simulado através de uma simulação de eventos discretos a qual considera os instantes que os eventos ocorrem. A simulação proporciona uma flexibilidade e agilidade maior na realização de testes onde são alteradas diversas variáveis como a quantidade de processadores e conjuntos de tarefas específicos. A maior desvantagem por optar pela simulação é fazer algumas suposições para eventos que acontecem durante um experimento real, como é o caso dos custos atrelados à troca de contexto. Contudo, ainda que esses custos não sejam aferidos diretamente, eles são incorporados ao WCET das tarefas, como abordado na Subseção 4.2. Ademais, a escolha da quantidade de preempções e migrações como métricas de avaliação ajuda a ter uma noção do desempenho do algoritmo em um ambiente real. Diante dessas considerações, este trabalho opta por utilizar a simulação para obter resultados próximos do real, mas sem a interferência de muitas variáveis ambientais e sem os custos associados à implementação num sistema real, principalmente com equipamentos.

A simulação de eventos discretos é adequada para o escalonamento de tempo real, pois lida com uma sequência de eventos que ocorrem em pontos distintos do tempo. O eventos são escolhidos para serem simulados em ordem cronológica com o relógio sempre avançando para

o tempo do próximo evento a ser simulado (MISRA, 1986). Isso evita que o simulador tenha que fazer as operações a cada mudança do relógio (tique), assim, otimizando a quantidade de iterações necessárias no processo. Os simuladores utilizados neste trabalho foram desenvolvidos na linguagem *Python*, sendo um próprio e o outro denominado SimSo (CHÉRAMY; HLADIK; DÉPLANCHE, 2014). Tanto as entradas quanto as saídas desses programas foram padronizadas.

5.1 GERADOR DE TAREFAS

Para avaliar PSSA, é preciso fornecer como entrada um conjunto de tarefas. A implementação do PSSA no simulador lê essa entrada como um arquivo de texto cujo conteúdo traz todas as características conjunto. Esse arquivo inicia com as informações gerais que são: quantidade m de processadores e o número de tarefas n. Após a leitura que há n tarefas, são lidas as n linhas contendo as informações de WCET e *deadline* de cada tarefa do sistema. Logo, para conseguir criar um conjunto de tarefas nesse modelo, é necessário gerar todas essas informações.

A criação dos conjuntos para este trabalho teve como base três critérios com a finalidade de variar os sistemas gerados, sendo eles a utilização $U(\Gamma)$ total do sistema, a quantidade m de processadores e o número n de tarefas. O primeiro critério base de $U(\Gamma)$ indica que deve-se criar conjuntos com diversas utilizações do sistema, partindo de $0.5\,m$ até $1.0\,m$, com incrementos de $0.1\,m$. O total de conjuntos gerados é o mesmo para todas as utilizações contempladas. Para cada utilização, há variações na quantidade m de processadores e n tarefas. O número m de processadores foi definido como $2, 3, \ldots, 14$. O número de tarefas n em cada conjunto de tarefas Γ em função de m, variando de acordo com o intervalo $[m + 1, 3m)$. A escolha por partir de $m + 1$ tarefas é uma decisão para não favorecer cenários onde todas as tarefas sejam alocadas inteiramente aos processadores, o que faria o PSSA comportar-se como EDF particionado (pEDF). Comportando-se como pEDF, a migração de tarefas não seria necessária e os resultados não ajudariam a avaliar o algoritmo quando os servidores *shadow* são exigidos. No total, 13.260 conjuntos sintéticos foram gerados, 10 para cada configuração de m, n e $U(\Gamma)$. Cada utilização avaliada possui 2210 conjuntos.

Diante dos valores estabelecidos de n e $U(\Gamma)$, o algoritmo gerador traduz essas informações nos WCET's e *deadlines* que cada tarefa precisa ter para que a utilização total seja a esperada. Inicialmente foram gerados os períodos T das tarefas aleatoriamente de acordo com uma distribuição uniforme, sendo um valor inteiro no intervalo $[1, 100]$. Concomitantemente, são definidas as utilizações que cada tarefa precisa ter para que a soma atinja a utlização $U(\Gamma)$ especificada. Essas utilizações são geradas pelo algoritmo "RandFixedSum" de Stafford (EMBERSON; STAFFORD; DAVIS, 2010) cujo propósito é retornar um vetor aleatório que tenha uma soma pré-estabelecida. De posse da utilização e período de cada tarefa, os WCET's são obtidos de maneira simples multiplicando $U(\tau_i) \times T(\tau_i)$, utilizando uma precisão de 10^{-6} para lidar com os pontos flutuantes. Ao final do processo, todas as informações necessárias para alimentar a simulação do algoritmo foram concebidas.

5.2 RESULTADOS

Para cada sistema de tarefas, a simulação executou por 1000 unidades de tempo. Preempção e migração por *job* foram as métricas de interesse por serem relacionadas aos *overheads* de tempo de execução causados por trocas de contexto ou tempo adicional para carregar os dados/instruções durante a migração da tarefa. A necessidade de realizar a preparação dos requisitos

necessários para a execução de uma tarefa que migra torna uma migração mais onerosa que uma preempção. Sempre que um *job* inicia sua execução há avaliação que verifica se ele já havia sido executado anteriormente. Caso a execução já tenha acontecido, sendo interrompida em algum momento, é contabilizada uma preempção ou migração dependendo do processador onde a execução foi retomada. Se o *job* continuou no mesmo processador, é adicionada uma preempção, do contrário uma migração é computada.

Conforme discutido no Capítulo 4, PSSA é dividido em duas partes sendo a primeira parte *off-line* composta pelo particionamento do sistema. Essa parte *off-line* permite a utilização de qualquer heurística de *bin-packing* para atribuir as tarefas aos processadores, desde que o conjunto esteja em ordem decrescente de utilização (ver Seção 4.4). A Figura 5.1 mostra o resultado de migrações por *job* e preempções por *job* ao escalonar os 13.260 conjuntos em diferentes heurísticas de *bin-packing*. As heurísticas *Next-Fit* (NF), *First-Fit* (FF), *Best-Fit* (BF), *Worst-Fit* (WF) são conhecidas da literatura. A heurística *Period-fit* (PF) foi criada para este trabalho e consiste em atribuir cada tarefa a um processador disponível onde o maior período de uma tarefa já atribuída possui mais similaridade com o período da tarefa a ser alocada. Os gráficos da Figura 5.1 denotam uma grande semelhança no desempenho de diferentes heurísticas, com exceção do NF. Em preempções por *job*, NF é melhor que todas as outras heurísticas quando a utilização total do sistema é igual ou inferior a 80%. Contudo, em utilizações superiores a 80%, o desempenho é o contrário, sendo o pior dentre todas as heurísticas. Em migrações por *job*, NF já não tem mais a vantagem de ser o melhor em momento algum, sendo igual ou pior que as outras heurísticas.

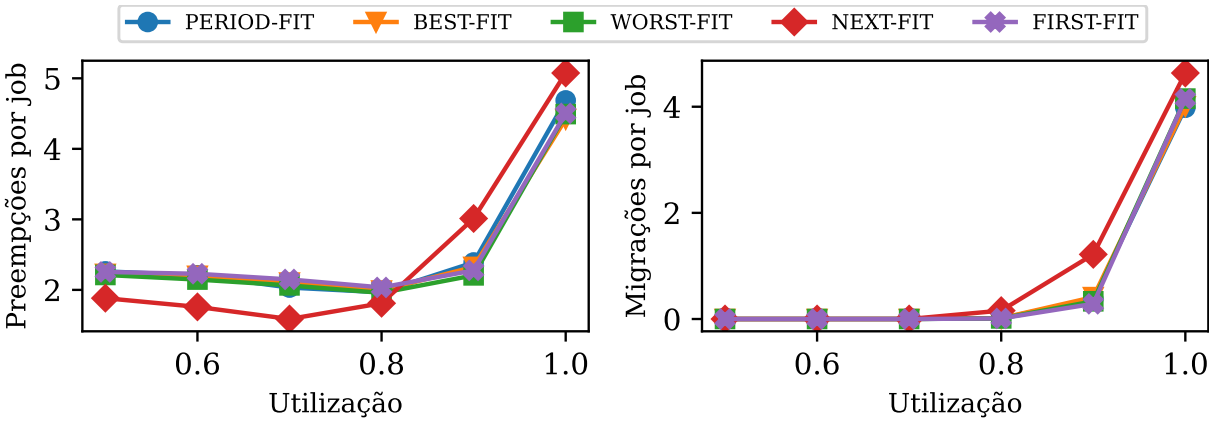

Figura 5.1 Comparação entre diferentes heurísticas considerando a média de preempções e migrações por *job*

O comportamento visto na Figura 5.1 pode ser explicado pela Figura 5.2. Sendo 2210 conjuntos gerados por utilização, observe que até 70% todos os conjuntos puderam ser particionados sem a necessidade de tarefas migratórias. Diante desse cenário, não há migração para os conjuntos com 50% até 70% de utilização, explicando o motivo das migrações estarem zeradas para todas as heurísticas, somente começando a tê-las a partir de 80% de utilização. Isso quer dizer que, em conjuntos puramente particionados, NF se saiu melhor pelo seu particionamento gerado. Quando os conjuntos passaram a exigir a migração de tarefas para serem escalonados, essa vantagem desaparece, dando ao NF a característica de pior desempenho. Note também que a Figura 5.2 mostra que o NF foi o único algoritmo que destoou na quantidade de conjuntos totalmente particionados, com uma grande diferença sendo construída a partir dos 70% de

utilização. Essa diferença na quantidade de conjuntos capazes de serem particionados é outro motivo da queda de performance do NF, mostrando o que pode ser facilmente constatado que é o fato de conjuntos totalmente particionados terem melhor desempenho que conjuntos com tarefas migratórias. No entanto, há a igualdade em 100% de utilização onde nenhum conjunto pode ser puramente particionado. Mesmo assim, NF ainda tem performance pior que os outros algoritmos. Isso leva ao gráfico presente na Figura 5.3 que exibe a média de níveis de processadores lógicos gerados. A indicação de 0 níveis representa que não são necessários processadores lógicos. A partir de 2 níveis há uma dependência entre processadores lógicos, resultando em uma hierarquia cada vez maior conforme o acréscimo de níveis. Como NF foi a heurística que gerou mais acoplamento de unidades lógicas, isso impactou em seu desempenho, mostrando uma relação direta entre níveis e migrações/preempções geradas.

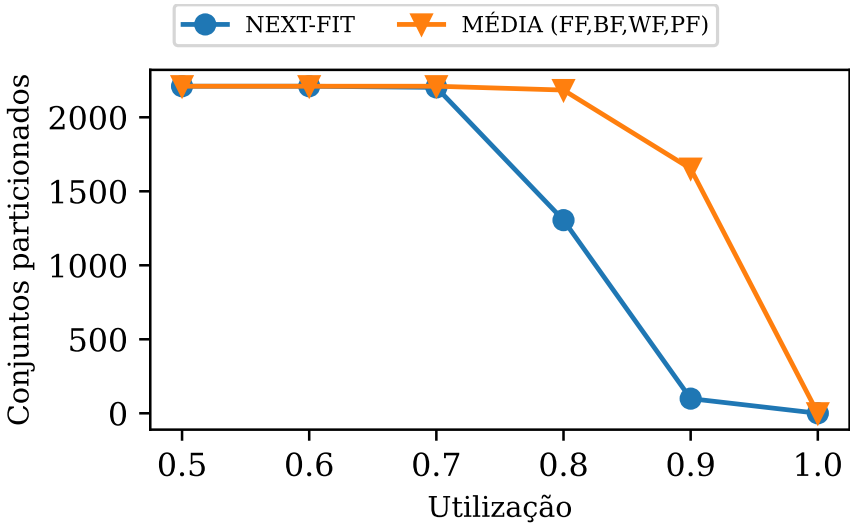

Figura 5.2 Número de conjuntos particionados, *i.e* sem tarefas migratórias, sendo o número máximo 2210.

De forma resumida, as heurísticas tiveram desempenho próximos enquanto os conjuntos não exigiram processadores lógicos. Isso pode ser visto como uma vantagem do PSSA, visto que poder comporta-se como pEDF ajuda muito no desempenho em termos de migração e preempção. Quando são exigidos os processadores lógicos, há um salto no número de preempções e migrações geradas, mas eles são necessários para que o algoritmo seja ótimo. Mais níveis de processadores lógicos também geram um número maior das métricas avaliadas, sendo um ponto de atenção. A quantidade de níveis gerados também pode ser relevante pensando em implementação em um sistema real, porque pode gerar mais *overheads* de escalonamento. A partir dessas obervações, é importante avaliar diferentes heurísticas, almejando a utilização do pEDF e, se não for possível, gerar o menor número de níveis de processadores lógicos possível.

PSSA foi comparado com DP-Wrap (LEVIN et al., 2010), EKG (ANDERSSON; TOVAR, 2006) e RUN (REGNIER et al., 2011) que, como visto no Capítulo 3, são algoritmos ótimos e utilizam o mesmo modelo de tarefas deste trabalho. Sendo o RUN conhecido por ser o algoritmo com melhor desempenho para esse modelo de tarefas até o momento, ele foi usado como a base da comparação. Não é possível afirmar que PSSA é melhor que RUN, embora eles possuam desempenho similar. Os resultados da simulação do *Notional Processors* (BLETSAS; ANDERSSON, 2009) não foram apresentados porque a quantidade de conjuntos que ele conse-

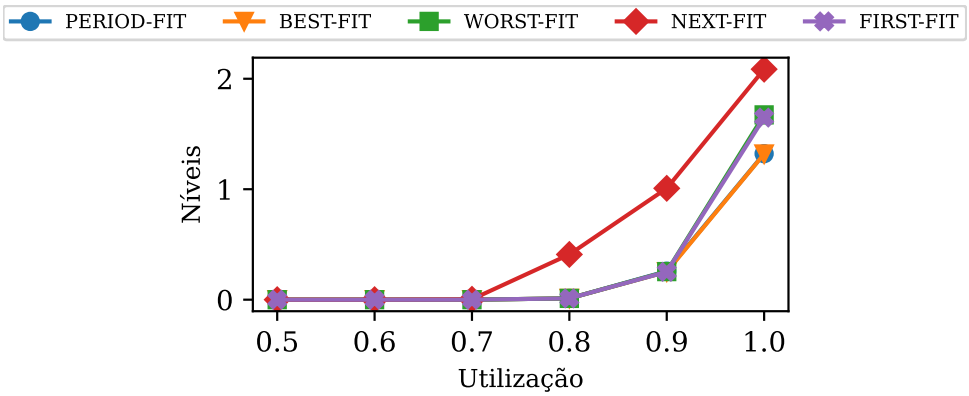

Figura 5.3 Média de níveis de processadores lógicos necessários para escalonar cada utilização.

guiu escalonar foi bem abaixo dos outros algoritmos. Já que os demais algoritmos são ótimos para o modelo de tarefas considerado, eles conseguiram escalonar 100% dos conjuntos testados. EKG foi configurado de modo que ele possa lidar com sistemas totalmente utilizados ($k = m$).

A Figura 5.4 mostra os resultados obtidos ao escalonar os mesmos 13.260 conjuntos gerados nos escalonadores escolhidos para comparação. A heurística utilizada na avaliação obtida nos gráficos foi a FF. Visto que a maioria das heurísticas avaliadas desempenham de forma semelhante, não há grande variação no resultado. Como esperado, o DP-Wrap e EKG desempenharam bem pior que RUN e PSSA. Conforme exposto anteriormente, eles sofrem com possíveis pequenas janelas de escalonamento utilizadas para sincronizar as decisões de escalonamento entre os diferentes processadores. PSSA e RUN possuem desempenho similar. Em termos de média de preempção por *job*, PSSA mostrou-se um pouco melhor que o RUN para sistemas com utilizações maiores que 90%, apesar de RUN ainda superar todos os algoritmos na média de migração por *job*. Ter menos migrações dá a vantagem ao RUN, uma vez que a migração possui um custo maior associado do que a preempção.

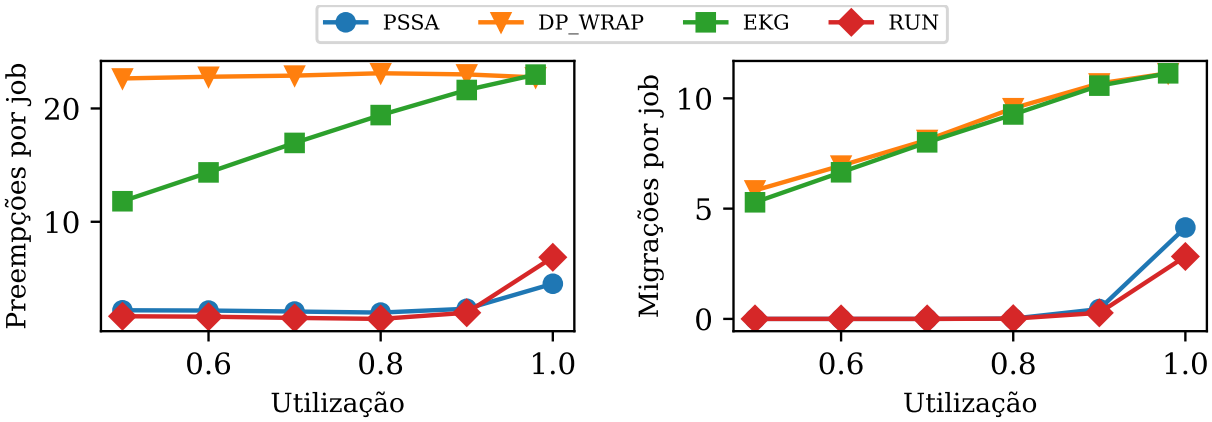

Figura 5.4 Comparação da média de preempções por *job* e migrações por *job* entre algoritmos ótimos para sistemas periódicos

A Figura 5.4 expôs uma maior diferença entre PSSA e RUN quando os conjuntos escalonados possuem uma alta utilização do sistema. A fim de obter uma avaliação mais detalhada sob essas condições, foram gerados mais conjuntos sintéticos. Os critérios foram os mesmos do conjunto

principal de 13.260 tarefas, sendo gerados 2210 conjuntos para cada utilização considerada. As utilizações avaliadas pertencem ao intervalo $[0.95\,m, 1.0\,m]$ com incrementos de $0.01\,m$. Os resultados retornados, presentes na Figura 5.5, evidenciam melhor como são próximos os valores de migrações por *job*, com PSSA distanciando um pouco quando todo o sistema é exigido. Entretanto, não é uma diferença tão grande como do PSSA e RUN para o EKG e DP-WRAP. Em preempções por *job* PSSA consegue ser melhor, sendo o inverso da outra métrica comparada. Além disso, a vantagem que PSSA tem sobre o RUN em preempções por *job* é superior a desvantagem em migrações por *job* no que diz respeito à diferença entre os dois algoritmos.

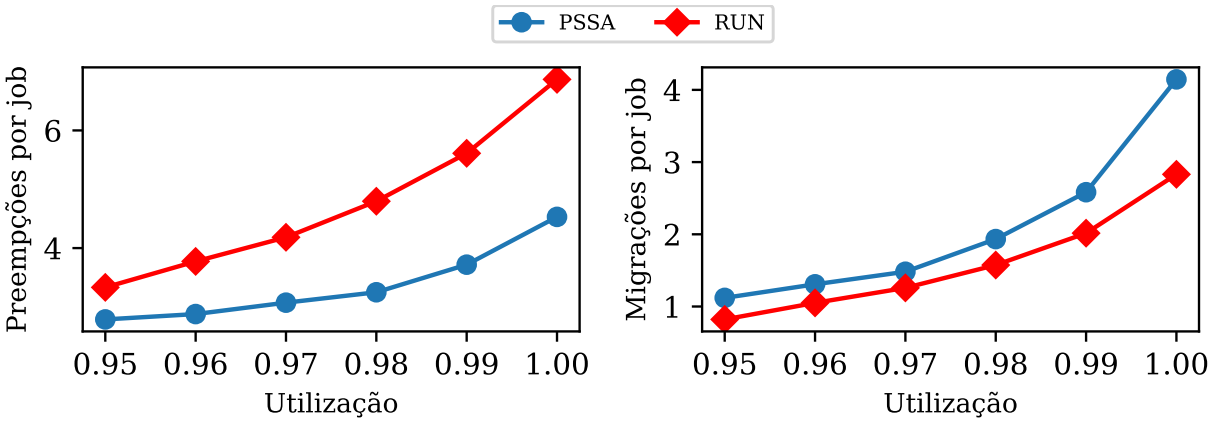

Figura 5.5 Comparação da média de preempções e migrações por *job* entre algoritmos RUN e PSSA considerando altas utilizações.

Neste capítulo, foi possível observar que PSSA superou DP-WRAP e EKG, ficando próximo de RUN que é o algoritmo conhecido por ter o melhor desempenho para esse modelo de tarefas. Como PSSA tem seu desempenho muito próximo do de RUN e RUN não consegue lidar com tarefas que são liberadas esporadicamente (LIMA, 2019), adaptar PSSA para esse e outros modelos de tarefa é um problema natural a ser abordado, assunto a ser comentado no próximo capítulo.

Capítulo 6

PSSA ESPORÁDICO

Ser flexível para atender diferentes modelos de tarefas é uma característica desejável ao escalonador de tempo real, pois pode-se adaptar às necessidades de varias aplicações. Com o objetivo de atender a mais modelos que o periódico e, consequentemente, obter a adaptabilidade do algoritmo, o modelo de tarefas esporádicas foi testado através de conjuntos de tarefas gerados com 1 até n tarefas esporádicas. A diferença na geração de tarefas esporádicas é que precisa ser dito quando uma tarefa irá chegar, respeitando o intervalo mínimo entre a chegada de seus *jobs*. Os instantes de chegada de cada tarefa esporádica foram determinados com o auxílio da distribuição de Poisson, considerando o tempo que a simulação executa, o qual foi estabelecido como 1000 unidades de tempo. Infelizmente, ao simular o escalonamento dos conjuntos com uma ou mais tarefas esporádicas,*Partitioning and Server Shadowing Algorithm* (PSSA) perdeu sua condição de ótimo ao não garantir o cumprimento de todos os *deadlines* de suas tarefas. Através da experimentação foi percebido que as atuais regras que regem PSSA não se adequam ao modelo de tarefas esporádicas, necessitando de alterações para atendê-lo. Neste capítulo serão discutidas algumas das alterações tentadas bem como os problemas enfrentados ao lidar com esse novo modelo.

6.1 HORIZONTE

O *horizonte* foi um conceito apresentado por Lima (2023) visando uma adaptação do RUN (REGNIER et al., 2011) para suportar o modelo de tarefas esporádicas (LIMA, 2023). Assim como na adaptação do RUN, o horizonte também é necessário para que PSSA possa atender tarefas esporádicas, visto que nem todos os deadlines são previamente conhecidos desde o início do escalonamento. A possibilidade de uma tarefa atrasar torna seu deadline desconhecido. Portanto, o deadline de seus servidores não fica bem definido. No entanto, é necessário que deadlines de servidores estejam definidos, pois através deles defini-se seus o tamanho de seus jobs. O conceito de horizonte veio para contornar este problema, o conceito este que será detalhado ao longo desta seção.

Uma tarefa é definida como ativa desde a liberação do seu *job* até o momento de chegada do seu *deadline* absoluto. Pelo outro lado, se é chegado o intervalo mínimo entre chegada de um *job* da tarefa e ela não dispara nenhum *job*, a tarefa está inativa (LIMA, 2023). Por esse contexto, conclui-se que no sistema periódico todas as tarefas estão ativas durante todo o escalonamento, uma vez que sempre é disparado um *job* no seu intervalo mínimo entre chegadas.

Em um sistema esporádico, algumas tarefas podem estar inativas no instante t, fazendo que nem todos os *deadlines* sejam conhecidos. Por exemplo, um servidor σ composto por duas tarefas τ_1 e τ_2 sendo $D(\tau_1) = 4$ e $D(\tau_2) = 10$. Supondo que no instante 0 somente τ_2 tenha liberado um *job*, ao estabelecer o *deadline* mais próximo de σ o resultado será $D(\sigma, 0) = 10$. Porém, τ_1 chegando no instante r sendo $r = 1$ mudaria o *deadline* de σ para $D(\sigma, 1) = 5$. Essa mudança pode causar perda de *deadlines*, pois entre $[0, 1)$ a carga do servidor é maior que deveria ser. Se neste período σ executar mais que o que se espera de sua utilização para o novo deadline de 5, a escalonabilidade do sistema como um todo pode estar comprometida. Em outras palavras, não conhecer todos os *deadlines* dos seus clientes no início deixa o servidor suscetível a mudanças repentinas de prioridade, levando ao sistema a subestimar a prioridade de um servidor (e superestimar suas cargas), o que pode resultar na perdas do *deadlines*. Tendo isso em vista, é necessário ter um recurso para que essa avaliação errônea de prioridade, ocasionada pelo *deadline* do servidor, não aconteça. Para isso, é utilizado o Horizonte do servidor. Sendo t um instante qualquer e $r \leq t$ o último instante até t o qual o servidor σ libera um *job*; $r = 0$ se não tiver liberado nenhum *job* em um instante t. O Horizonte de σ é obtido pela Equação (6.1).

$$H(\sigma, t) = \begin{cases} r + D & \text{Se } \sigma \text{ é uma tarefa e } r + D > t \\ t + D & \text{Se } \sigma \text{ é uma tarefa e } r + D \leq t \\ min_{\sigma_i \in \sigma} H(\sigma_i, t) & \text{Se } \sigma \text{ é um servidor} \end{cases} \tag{6.1}$$

A prioridade de um servidor está assegurada com o horizonte, visto que o *deadline* mais próximo vigente não depende dos clientes ativos, *i.e* sem a necessidade de ter que conhecer todos os *deadlines* a qualquer instante. Ele pode até ser restritivo não mudando a prioridade de um servidor perante ao sistema, mas é isso que garante que a prioridade não será subestimada. Como exemplo de comportamento restritivo, considere que a tarefa τ_1 não chega em momento algum. Da mesma forma, os *deadlines* absolutos de σ serão $D(\sigma, 0) = 4$, $D(\sigma, 4) = 8$ e $D(\sigma, 4) = 10$, respectivamente. Isso é, o escalonamento de σ fica segmentado nos intervalos $[0, 4)$, $[4, 8)$ e $[8, 10)$ ao invés de ser escalonado em $[0, 10)$ diretamente. De agora em diante, ao tratar de servidores considere $D(\sigma, r) = H(\sigma, r)$.

6.2 LIBERAÇÃO DE *JOBS*

Outro ponto a ser revisado pensando em um sistema esporádico é a definição da carga de trabalho do servidor. O cálculo da sua demanda de execução $C(\sigma, t)$ ou *budget* como foi apresentado leva em consideração a utilização total do servidor $U(\sigma)$. Com base no servidor de exemplo σ apresentado anteriormente, se $U(\tau_1) = 2/4$ e $U(\tau_2) = 5/10$, então $U(\sigma) = 1$. Supondo uma liberação de *job* no instante 0 ($r = 0$) o *budget* $C(\sigma, r)$ obtido é dado por $(D(\sigma, r) - r) \times U(\sigma) = (4 - 0) \times 1 = 4$. Pensando em um sistema onde todas as tarefas cliente de um servidor estão ativas, como no caso de tarefas periódicas, essa carga feita no instante 0 é o tempo necessário para garantir que todos os seus clientes executem o que precisam antes da chegada do *deadline* do servidor. Contudo, assuma que a tarefa τ_2 atrasou e só irá liberar um *job* no instante 5. Nessa situação a liberação só ocorrerá após o *deadline* de σ ($D(\sigma, 0) = 4$), sendo somente necessário executar a demanda exigida pela tarefa ativa τ_1 em $[0, 4)$, ou seja, $(D(\tau_1, r) - r) \times U(\sigma) = (4 - 0) \times 0.5 = 2$. Visto que σ só necessita de 2 unidades de tempo para a execução de seus clientes, não há motivos para basear o *budget* na sua utilização total $U(\sigma)$. Caso haja a manutenção de $C(\sigma, 0) = 4$, serão reservadas 4 unidades de tempo para σ,

implicando numa ociosidade por 2 unidades onde o servidor não tem nada para executar, já que τ_2 não chegou. Para evitar o desperdício de poder computacional, pensando nos casos onde nem todas as tarefas estão ativas, o *budget* de um servidor $C(\sigma, r)$ só irá considerar a utilização das tarefas ativas em r. Portanto, sendo o conjunto de tarefas ativas no instante r representado por $A(\Gamma, r)$, um *job* de σ liberado em r tem sua demanda de execução $C(\sigma, r)$ calculada através da Equação (6.2).

$$C(\sigma, r) = (D(\sigma, r) - r) \times \sum_{\sigma_i \in A(\sigma, r)} U(\sigma_i) \tag{6.2}$$

Considere o caso em que a tarefa τ_2 chega no instante 1 ao invés de 5. τ_2 chega dentro do intervalo $[0, 4)$ já considerado para o cálculo da demanda $C(\sigma, 0)$ do *job* de σ. Observe que não é possível utilizar a Equação (6.2) para obter a demanda do *job* liberado em 1. Isso deve-se ao fato que, ao calcular a demanda do segundo *job* pela Equação (6.2), o *budget* total em $[0, 4)$ seria superdimensionado. A demanda obtida pela Equação (6.2) no instante 1 seria 3. Se o primeiro *job* liberado em 0 não for escalonado para executar em $[0, 1)$, sua demanda se manteria em 2. Somando com a fração exigida calculada para o segundo *job*, a demanda de execução total seria 5 unidades de tempo, sendo que há 3 unidades de tempo disponíveis em $[1, 4)$, que seria o momento atual até o próximo *deadline* de σ. A mesma falha ocorreria se σ fosse executado em $[0, 1)$, pois, apesar de só restar 1 de *budget* das 2 unidades de tempo calculada por $C(\sigma, 0)$, a soma com as 3 unidades obtida por $C(\sigma, 1)$ ainda ultrapassaria o tempo disponível de 1 até o próximo *deadline* de σ. Sendo assim, deverá existir um incremento (recarga) em $C(\sigma, 0)$ para que ele tenha condições de atender τ_2 também durante o intervalo considerado. O incremento ao *budget* deve ser feito sempre que há uma liberação de *job* de um cliente que estava inativo antes da chegada do $D(\sigma, r^-)$, sendo r^- o último instante de liberação de um *job* de σ. O quanto deve ser incrementado ao *budget* depende da soma de utilizações dos clientes que tornaram-se ativos no instante atual r. Então, a demanda de execução resultante após o incremento é dada pela Equação (6.3). Desta forma, a exigência de execução adicionada seria de 1.5, obtendo $C(\sigma, 1) = 3.5$ ou $C(\sigma, 1) = 2.5$, a depender do *budget* que σ possui no instante r ($C(\sigma, r^-)$).

$$C(\sigma, r) = C(\sigma, r^-) + (D(\sigma, r) - r) \times \sum_{\sigma_i \in A(\sigma, r) \setminus A(\sigma, r^-)} U(\sigma_i) \tag{6.3}$$

No contexto do algoritmo PSSA esporádico, as recargas são feitas da mesma forma no par de servidores $(\sigma_i^1; \sigma_i^2)$. Por exemplo, um sistema tem um processador lógico P_j que possui σ_i^2 entre seus σ^2 onde $U(\sigma_i^2) = 1/2$. Caso uma tarefa τ chegue para P_j requisitando $1/4$ de sua fração computacional, só será adicionado ao *budget* de σ_i^2 e σ_i^1 o correspondente a $1/4$ no instante r de liberação de τ. Outra observação a ser ponderada é o fato de quem dita a recarga de σ_i^1 é σ_i^2. Portanto, um servidor *shadow* σ_i^1 só tem seu *budget* recarregado quando uma tarefa chega a um processador lógico exigindo a recarga de σ_i^2. Quando não, não é necessário creditar nenhuma demanda de execução para σ_i^1.

O novo comportamento na liberação de *jobs* discutido nesta seção é pensado para que só seja fornecido exatamente o *budget* é necessário, evitando que as tarefas que não estejam ativas imponham ociosidade indevida aos processadores. A ociosidade indevida é responsável por bloquear a execução de tarefas prontas no sistema que não são escalonadas por ter prioridade menor por causa do seu *deadline*, já que PSSA utiliza *Earliest Deadline First* (EDF). A ocorrência desse fenômeno não ocorre no modelo periódico porque os *budgets* sempre representam exatamente o que é exigido pelos clientes de um servidor, já que todas as tarefas sempre es-

tão ativas. Sendo assim, para o modelo periódico, é plausível liberar o *job* do servidor σ com sua utilização total porque não será exigida mais do que a demanda calculada do instante de liberação r até $D(\sigma, r)$.

6.3 PROBLEMAS ENFRENTADOS

Considere o conjunto de $\Gamma = \{\tau_1 : (9, 10), \tau_2 : (7, 10), \tau_3 : (4, 10)\}$ para serem executadas em dois processadores (P_1 e P_2) e os mecanismos relativos aos conceitos de horizonte e de liberação de *jobs* de servidores com *budgets* parciais, explicados nas seções anteriores. Ao escalonar esse conjunto com PSSA é necessária a criação de um processador lógico P_3 com 40% de capacidade. Como o sistema é esporádico, suponha que no instante 0 somente τ_3 chegou. Sendo assim, são recarregados apenas os *budgets* dos servidores *shadow* $(\sigma_1^1; \sigma_1^2)$ e $(\sigma_2^1; \sigma_2^2)$ com 1 e 3 respectivamente, levando em conta o horizonte de 10 unidades de tempo. Os dois σ^2 em P_3 têm a mesma prioridade e os dois processadores físicos estão ociosos, então a escolha é arbitrária. Em um caso onde as prioridades são as mesmas e existe um processador ocioso, deve ser atribuída maior prioridade ao σ^2 que o seu σ^1 correspondente está em um processador ocioso. Neste exemplo, como as condições são as mesmas de prioridade e ociosidade, foi escolhido o primeiro processador P_1 através de σ_1^2. Então, o par $(\sigma_1^1; \sigma_1^2)$ executa até seus *budgets* se esgotarem no instante 1. No instante 1 também chega τ_2 para P_2. Contudo, como a prioridade do escalonamento é do processador lógico P_3, além de $(\sigma_2^1; \sigma_2^2)$ ter *deadline* 10 menor que τ_2 (11), $(\sigma_2^1; \sigma_2^2)$ são escolhidos para executar. Dessa forma, todo o *budget* (3 unidades de tempo) do par $(\sigma_2^1; \sigma_2^2)$ é utilizado no intervalo $[1, 4)$, restando 6 unidades de tempo para τ_2 executar entre $[4, 10)$. No instante 10 é liberado outro *job* da tarefa τ_3. Tendo em vista que o horizonte do servidor é 11 (*deadline* de τ_2), é feita a recarga de 0.1 em $(\sigma_1^1; \sigma_1^2)$ e 0.3 em $(\sigma_2^1; \sigma_2^2)$. Novamente repete-se o cenário do intervalo $[0, 10)$, agora considerando que P_1 tem prioridade sobre P_2 porque τ_1 não chegou, fazendo P_1 ficar ocioso. Mas, após a execução de τ_3 através de 0.1 unidade de tempo de $(\sigma_1^1; \sigma_1^2)$ e 0.4 de $(\sigma_2^1; \sigma_2^2)$. Observe que restou apenas 0.7 unidade de tempo para a execução de τ_2 no intervalo $[10, 11)$. Como τ_2 executou 6 unidades dentro de $[1, 10)$ e 0.7 em $[10, 11)$, $6 + 0.7 = 6.7$, restando executar 0.3, já que τ_2 deveria executar por 7 unidades de tempo entre 1 e 11. O problema exposto pode ser acompanhado pelo escalonamento presente na Figura 6.1.

A falta de execução de 0.3 unidade de tempo de τ_2 já poderia ser identificada no momento da sua chegada em 1. Observe que, por não ter precisado executar dentro de $[0, 1)$, o servidor σ_2^1 manteve seu *budget* de 3 intacto. Portanto, σ_2^1 ainda teria que executar 3 entre $[1, 10)$. Contudo, se for pensado o horizonte de τ_2 como 10 quando a tarefa chega, temos que ela deveria executar 70% no intervalo $[1, 10)$, totalizando 6.3. A soma das demandas de execução de σ_2^1 e τ_2 dentro de $[1, 10)$ é: $3 + 6.3 = 9.3$. Isso mostra que a demanda ultrapassa 0.3 do tempo de execução disponível.

O problema descrito não ocorre na versão periódica, uma vez que as tarefas estão sempre ativas. Sendo assim, o processador só fica ocioso quando todas os *jobs* já tiveram sua necessidade de execução cumprida antes do mais próximo *deadline* absoluto. Em outras palavras, existiriam *jobs* prontos capazes de executar no intervalo $[0, 1)$ do exemplo anterior. A Figura 6.2 mostra como seria o escalonamento desse mesmo conjunto caso o sistema fosse periódico, com um *job* de τ_2 executando entre 0 e 1. No caso de tarefas esporádicas, a ociosidade não está necessariamente associada ao cumprimento das execuções, podendo ser resultante dos atrasos das tarefas. No exemplo anterior, as tarefas atribuídas a P_2 representam a utilização de 100% do seu poder computacional. Portanto, é nítido que se há *jobs* prontos eles devem ser executados.

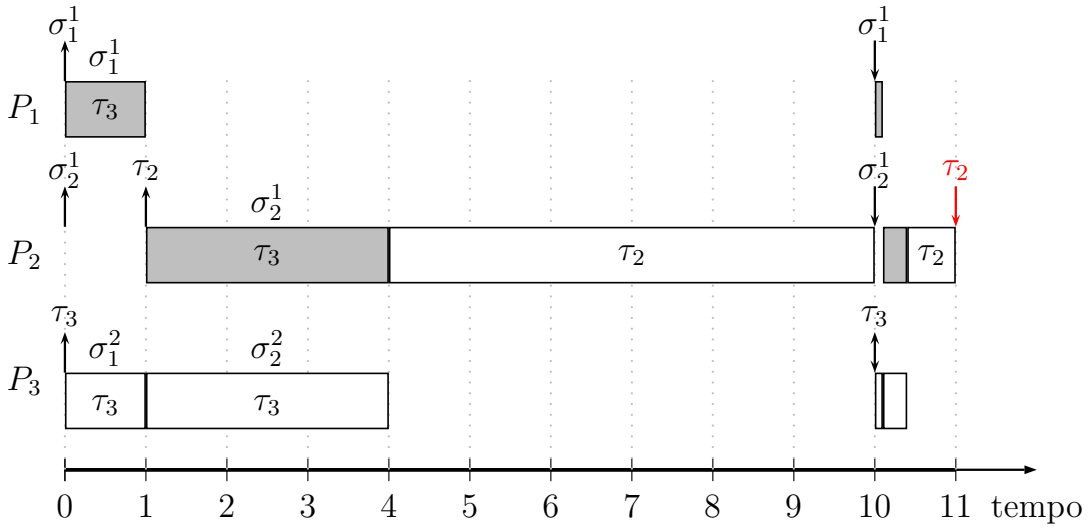

Figura 6.1 Escalonamento via PSSA esporádico do conjunto $\Gamma = \{\tau_1 : (9, 10), \tau_2 : (7, 10), \tau_3 : (4, 10)\}$ em 2 processadores físicos. τ_2 perde *deadline* no instante 11 devido ao seu atraso em 1 unidade de tempo

Contudo, isso não acontece porque um *job* de σ_2^1 foi liberado com *budget* $C(\sigma_2^1, 0) = 3$ e ele não foi executado em $[0, 1)$. Isso ocorre pela característica dos processadores lógicos de controlar a execução entre seus σ^2 para fazer seu papel de sincronização, evitando a execução paralelas de *jobs*. Diante disso, σ_2^1 não deve executar se σ_2^2 não executar também, e quem controla esse escalonamento é o processador lógico. Nesse cenário, há um conflito, sendo quebrada a obrigatoriedade da execução de *jobs* prontos em P_2, o que leva à perda de *deadline*. No caso periódico, essa regra não é quebrada já as tarefas ativas liberariam *jobs* que estariam executando quando o σ_2^1 não fosse escalonado. Isso quer dizer que, no caso periódico, há a compensação do tempo em que σ^1 não executa por parte das tarefas alocadas ao processador onde está σ^1. Através dessa propriedade, é garantido que o *budget* atribuído ao servidor σ^1 é no máximo complementar à demanda que o processador precisa para executar suas demais tarefas. No caso esporádico, quanto mais tempo a tarefa atrasar, combinado com o não escalonamento de σ^1, a janela para execução até o *deadline* fica cada vez menor. Este cenário pode chegar ao limite onde a soma das demandas de σ^1 com as tarefas restantes que compartilham processador com o servidor *shadow* ultrapasse o tempo disponível para cumprir todas as requisições computacionais. Como já discutido, foi esse caso que ocorreu no instante 10 mostrado na Figura 6.1.

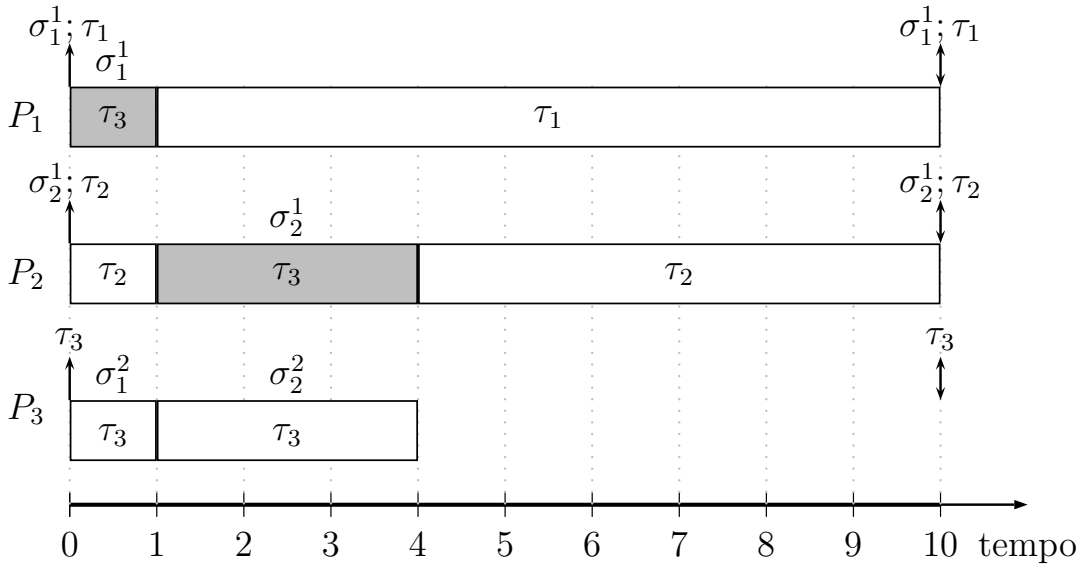

Figura 6.2 Escalonamento via PSSA do conjunto $\Gamma = \{\tau_1 : (9, 10), \tau_2 : (7, 10), \tau_3 : (4, 10)\}$ em 2 processadores físicos, considerando o sistema periódico.

Capítulo 7

CONSIDERAÇÕES FINAIS

PSSA foi concebido pensando nos benefícios que o uso de servidores traria a uma estratégia semi-particionada. Estratégia essa que foi pensada para se obter um equilíbrio entre o limite de utilização do sistema e o *overhead* em termos de números de preempção e de migração de *jobs*. O conceito de servidor também se mostrou benéfico quando utilizado para fazer a sincronização das tarefas de uma forma mais dinâmica, ao invés das janelas estáticas de escalonamento. Consequentemente, PSSA proporciona um bom desempenho em termos de *overheads* de escalonamento e trocas de contexto. Partindo dessa ideia, o algoritmo foi codificado num simulador de eventos discretos, e os resultados foram obtidos considerando as métricas de número de preempção e de migração por *job*. De posse desses resultados, PSSA foi comparado com outros algoritmos ótimos (para o modelo de tarefas periódicas), tendo sido observado grande vantagem do seu desempenho sobre DP-Wrap e EKG. O resultado do EKG, especificamente, é positivo por ser outro algoritmo que também utiliza a estratégia semi-particionada. Isso mostra que a adoção do mecanismo de sincronização baseado em servidores foi superior à utilização de *timeslots* fixos. Os resultados também mostraram que o desempenho do PSSA foi similar ao *Reduction to uniprocessor* (RUN), o que é também relevante visto que RUN é conhecido por ser ter um dos melhores resultados ao escalonar conjuntos de tarefas para o modelo considerado. Para que PSSA não fosse somente um algoritmo com desempenho similar ao do RUN, sua extensão para o modelo esporádico seria desejável, pois até então não se conhece uma extensão de RUN para esse modelo (LIMA, 2019) que não sacrifique otimalidade. Entretanto, infelimente, as tentativas de estender PSSA para tarefas esporádicas tentadas no contexto do presente trabalho foram foram infrutíferas.

Não obstante, demonstrou-se aqui que PSSA é um algoritmo ótimo para sistemas compostos de tarefas periódicas com deadlines implícitos. Isto é, o algoritmo consegue escalonar qualquer conjunto de tarefas periódicas independentes, desde que sua utilização não ultrapasse a capacidade total do sistema. Porém, ao gerar e testar conjuntos de tarefas esporádicos no simulador, alguns sistemas perderam *deadline*, demonstrando a perda da condição de ótimalidade do PSSA ao lidar com o modelo esporádico. Na tentativa de tornar PSSA ótimo também para tarefas esporádicas, foram feitas modificações no *deadline* do servidor, usando o conceito de horizonte, e incorporou-se a liberação parcial de seus *jobs*. Infelizmente, mesmo após as necessárias alterações, estas se mostraram insuficientes.

Além do desafio de se estender PSSA para o modelo de tarefas esporádicas, é necessário verificar o comportamento de sua implementação num sistema real de forma a melhor avaliar

a sobrecarga em tempo de execução, principalmente no que se refere ao gerenciamento de servidores *shadow*. Tais temas podem ser explorados em trabalhos futuros.

REFERÊNCIAS BIBLIOGRÁFICAS

AKESSON, B. et al. A comprehensive survey of industry practice in real-time systems. Real-Time Systems, Springer, v. 58, n. 3, p. 358–398, 2022.

ANDERSSON, B. Semi-partitioned multiprocessor scheduling. In: _____. Handbook of Real-Time Computing. Singapore: Springer Nature Singapore, 2022. p. 175–192. ISBN 978-981-287-251-7. Disponível em: <https://doi.org/10.1007/978-981-287-251-7_2>.

ANDERSSON, B.; TOVAR, E. Multiprocessor scheduling with few preemptions. In: 12th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA'06). [S.l.: s.n.], 2006. p. 322–334. ISSN 2325-1301.

BARRETO, J.; MASSA, E.; LIMA, G. Partitioning and server shadowing for scheduling periodic real-time tasks on multiprocessors. In: SBESC 2023 (). [S.l.: s.n.], 2023.

BLETSAS, K.; ANDERSSON, B. Notional processors: An approach for multiprocessor scheduling. In: 2009 15th IEEE Real-Time and Embedded Technology and Applications Symposium. [S.l.: s.n.], 2009. p. 3–12. ISSN 1545-3421.

CHÉRAMY, M.; HLADIK, P.-E.; DÉPLANCHE, A.-M. Simso: A simulation tool to evaluate real-time multiprocessor scheduling algorithms. In: Proc. of the 5th International Workshop on Analysis Tools and Methodologies for Embedded and Real-time Systems. [S.l.: s.n.], 2014. (WATERS).

DAVIS, R. I.; BURNS, A. A survey of hard real-time scheduling for multiprocessor systems. ACM Comput. Surv., Association for Computing Machinery, New York, NY, USA, v. 43, n. 4, oct 2011. ISSN 0360-0300. Disponível em: <https://doi.org/10.1145/1978802.1978814>.

DERTOUZOS, M. L. Control robotics: The procedural control of physical processes. In: Proceedings IF IP Congress, 1974. [S.l.: s.n.], 1974.

DHALL, S. K.; LIU, C. L. On a real-time scheduling problem. Operations research, INFORMS, v. 26, n. 1, p. 127–140, 1978.

EMBERSON, P.; STAFFORD, R.; DAVIS, R. I. Techniques for the synthesis of multiprocessor tasksets. In: Proc. of WATERS. [S.l.: s.n.], 2010. p. 6–11.

JAIN, R. The art of computer systems performance analysis: techniques for experimental design, measurement, simulation, and modeling. [S.l.]: John Wiley & Sons, 1990.

KATO, S.; YAMASAKI, N. Semi-partitioned fixed-priority scheduling on multiprocessors. In: 2009 15th IEEE Real-Time and Embedded Technology and Applications Symposium. [S.l.: s.n.], 2009. p. 23–32.

KOPETZ, H. Real-time systems: design principles for distributed embedded applications. Kluwer Academic Publisher, 1997.

KORKI, M.; JIN, J.; TIAN, Y.-C. Real-time cyber-physical systems: State-of-the-art and future trends. In: TIAN, Y.-C.; LEVY, D. C. (Ed.). Handbook of Real-Time Computing. [S.l.]: Springer Nature Singapore, 2022. p. 509–540.

LEVIN, G. et al. Dp-fair: A simple model for understanding optimal multiprocessor scheduling. In: Proc. of ECRTS. [S.l.: s.n.], 2010. p. 3–13.

LIMA, G. Can the run scheduling algorithm go beyond periodic task models? In: Proc. of RTSOPS. [S.l.: s.n.], 2019. p. 4–5.

LIMA, G. On the possibilities and limitations of multiprocessor real-time scheduling based on reduction to uniprocessor. Tese (Doutorado) — Federal University of Bahia (UFBA), 2023.

LIMA, G.; MASSA, E.; REGNIER, P. Practical considerations in optimal multiprocessor scheduling. In: _____. Handbook of Real-Time Computing. Singapore: Springer Nature Singapore, 2022. p. 193–231. ISBN 978-981-287-251-7. Disponível em: <https://doi.org/10.1007/978-981-287-251-7_3>.

LIU, C. L.; LAYLAND, J. W. Scheduling algorithms for multiprogramming in a hard-real-time environment. J. ACM, Association for Computing Machinery, New York, NY, USA, v. 20, n. 1, p. 46–61, jan 1973. ISSN 0004-5411. Disponível em: <https://doi.org/10.1145/321738.321743>.

MALL, R. Real-Time Systems: Theory and Practice. 1st. ed. USA: Prentice Hall Press, 2009. ISBN 8131700690.

MASSA, E. et al. Optimal and adaptive multiprocessor real-time scheduling: The quasi-partitioning approach. In: 2014 26th Euromicro Conference on Real-Time Systems. [S.l.: s.n.], 2014. p. 291–300. ISSN 2377-5998.

MCNAUGHTON, R. Scheduling with deadlines and loss functions. Management science, INFORMS, v. 6, n. 1, p. 1–12, 1959.

MISRA, J. Distributed discrete-event simulation. ACM Comput. Surv., Association for Computing Machinery, New York, NY, USA, v. 18, n. 1, p. 39–65, mar. 1986. ISSN 0360-0300. Disponível em: <https://doi.org/10.1145/6462.6485>.

REGNIER, P. et al. Run: Optimal multiprocessor real-time scheduling via reduction to uniprocessor. In: 2011 IEEE 32nd Real-Time Systems Symposium. [S.l.: s.n.], 2011. p. 104–115. ISSN 1052-8725.