

Escalonamento Round Robin e EDF

Mateus Barbos e Matheus de Oliveira Rocha

Universidade do Vale do Itajaí - UNIVALI

Escola Politécnica

Ciência da Computação

{mateus.barbosa, matheus.rocha}@edu.univali.br

1. Introdução.....	1
2. Explicação e Contexto da Aplicação.....	1
3. Resultados Obtidos com as Simulações	2
4. Códigos Importantes da Implementação	4
4.1 Round Robin com Prioridade.....	4
4.2 EDF com Prioridade	6
5. Resultados obtidos com a implementação	10
6. Análise e discussão sobre os resultados.....	11
6.1 Round Robin com Prioridade.....	11
6.2 EDF com Prioridade	12
7. Conclusão.....	13
Referências.....	14

1. Introdução

Este relatório detalha a implementação de dois algoritmos de escalonamento de tarefas com prioridade: Round Robin com Prioridade (RR_p) e Escalonamento por Deadline Mais Próximo com Prioridade (EDF_p). O objetivo é desenvolver e testar esses algoritmos em um ambiente de simulação, utilizando uma estrutura de código base que inclui gerenciamento de tarefas, monitoramento de execução e manipulação de listas.

Dos 6 arquivos base disponibilizados (CPU.c, CPU.h, driver.c, list.c, list.h e task.h), apenas alteramos os códigos do list.c e list.h, criando uma lógica que permite inserir elementos no final da lista.

2. Explicação e Contexto da Aplicação

A escalonamento de tarefas é uma parte crucial dos sistemas operacionais em tempo real, onde múltiplas tarefas competem por recursos da CPU. A prioridade das tarefas influencia a ordem de execução, garantindo que tarefas críticas sejam atendidas em tempo hábil. Este projeto foca em dois algoritmos de escalonamento:

- **Round Robin com Prioridade (RR_p):** Este algoritmo usa múltiplas filas para gerenciar diferentes níveis de prioridade. A prioridade é considerada na escolha das tarefas a serem executadas, e a contagem de tempo (slice) é implementada para garantir que cada tarefa receba uma fatia justa de tempo de CPU.
- **Escalonamento por Deadline Mais Próximo com Prioridade (EDF_p):** Este algoritmo avalia os deadlines das tarefas e escolhe a tarefa com o menor deadline para execução. Um thread extra simula o timer de hardware, gerando uma flag de estouro de tempo para controle de slices e avaliação de deadlines.

3. Resultados Obtidos com as Simulações

Os resultados das simulações são obtidos através da execução dos algoritmos com conjuntos de tarefas definidos em arquivos de entrada (rr-schedule_pri.txt e edf-schedule_pri.txt). As simulações foram realizadas para verificar a eficiência dos algoritmos em termos de utilização da CPU e cumprimento dos deadlines.

Um ponto de atenção é que todas as tarefas foram inseridas no mesmo momento e quando a CPU iniciou, ou seja, o tempo de chegada delas a fila de escalonamento é de 0 unidades de tempo.

Figura 1: Execução do Algoritmo de Round Robin com Prioridade

```

make: *** No rule to make target 'for' . Stop.
[mateusbarbosa@ArchLinux scheduler_rr_p]$ ./rr_p rr-schedule_pri.txt
Running task = [T1] [1] [50] for 10 units.
Running task = [T7] [1] [50] for 10 units.
Running task = [T11] [1] [50] for 10 units.
Running task = [T1] [1] [40] for 10 units.
Running task = [T7] [1] [40] for 10 units.
Running task = [T11] [1] [40] for 10 units.
Running task = [T1] [1] [30] for 10 units.
Running task = [T7] [1] [30] for 10 units.
Running task = [T11] [1] [30] for 10 units.
Running task = [T1] [1] [20] for 10 units.
Running task = [T7] [1] [20] for 10 units.
Running task = [T11] [1] [20] for 10 units.
Running task = [T1] [1] [10] for 10 units.
Task T1 completed.
Running task = [T7] [1] [10] for 10 units.
Task T7 completed.
Running task = [T11] [1] [10] for 10 units.
Task T11 completed.
Running task = [T2] [2] [50] for 10 units.
Running task = [T4] [2] [50] for 10 units.
Running task = [T2] [2] [40] for 10 units.
Running task = [T4] [2] [40] for 10 units.
Running task = [T2] [2] [30] for 10 units.
Running task = [T4] [2] [30] for 10 units.
Running task = [T2] [2] [20] for 10 units.
Running task = [T4] [2] [20] for 10 units.
Running task = [T2] [2] [10] for 10 units.
Task T2 completed.
Running task = [T4] [2] [10] for 10 units.
Task T4 completed.
Running task = [T3] [3] [50] for 10 units.
Running task = [T8] [3] [50] for 10 units.
Running task = [T3] [3] [40] for 10 units.
Running task = [T8] [3] [40] for 10 units.
Running task = [T3] [3] [30] for 10 units.
Running task = [T8] [3] [30] for 10 units.
Running task = [T3] [3] [20] for 10 units.
Running task = [T8] [3] [20] for 10 units.
Running task = [T3] [3] [10] for 10 units.
Task T3 completed.
Running task = [T8] [3] [10] for 10 units.
Task T8 completed.
Running task = [T5] [4] [50] for 10 units.
Running task = [T10] [4] [50] for 10 units.
Running task = [T5] [4] [40] for 10 units.
Running task = [T10] [4] [40] for 10 units.
Running task = [T5] [4] [30] for 10 units.
Running task = [T10] [4] [30] for 10 units.
Running task = [T5] [4] [20] for 10 units.
Running task = [T10] [4] [20] for 10 units.
Running task = [T5] [4] [10] for 10 units.
Task T5 completed.
Running task = [T10] [4] [10] for 10 units.
Task T10 completed.
Running task = [T9] [5] [50] for 10 units.
Running task = [T9] [5] [40] for 10 units.
Running task = [T9] [5] [30] for 10 units.
Running task = [T9] [5] [20] for 10 units.
Running task = [T9] [5] [10] for 10 units.
Task T9 completed.
No tasks available. Waiting...

```

Fonte: Elaborado pelos Autores

Figura 2: Execução do Algoritmo EDF com Prioridade

```

[mateusbarbosa@ArchLinux scheduler_edf]$ ./edf ./edf-schedule_pri.txt
EDF: Reordering ready queues...
Running task = [T2] [1] [20] [30] for 10 units.
EDF: Reordering ready queues...
Running task = [T1] [1] [10] [50] for 10 units.
Task T1 completed.
EDF: Reordering ready queues...
Running task = [T2] [1] [10] [30] for 10 units.
Task T2 completed.
EDF: Reordering ready queues...
Running task = [T7] [1] [20] [50] for 10 units.
EDF: Reordering ready queues...
Running task = [T11] [1] [20] [50] for 10 units.
EDF: Reordering ready queues...
Running task = [T8] [1] [20] [80] for 10 units.
EDF: Reordering ready queues...
Running task = [T11] [1] [10] [50] for 10 units.
EDF WARNING: Deadline missed for task T11.
Task T11 completed.
EDF: Reordering ready queues...
Running task = [T3] [1] [20] [100] for 10 units.
EDF: Reordering ready queues...
Running task = [T10] [1] [20] [110] for 10 units.
EDF: Reordering ready queues...
Running task = [T5] [1] [10] [110] for 10 units.
Task T5 completed.
EDF: Reordering ready queues...
Running task = [T10] [1] [10] [110] for 10 units.
Task T10 completed.
EDF: Reordering ready queues...
Running task = [T9] [1] [10] [130] for 10 units.
Task T9 completed.
EDF: Reordering ready queues...
Running task = [T7] [1] [10] [50] for 10 units.
EDF WARNING: Deadline missed for task T7.
Task T7 completed.
EDF: Reordering ready queues...
Running task = [T3] [1] [10] [100] for 10 units.
EDF WARNING: Deadline missed for task T3.
Task T3 completed.
EDF: Reordering ready queues...
Running task = [T4] [1] [10] [70] for 10 units.
EDF WARNING: Deadline missed for task T4.
Task T4 completed.
EDF: Reordering ready queues...
Running task = [T8] [1] [10] [80] for 10 units.
EDF WARNING: Deadline missed for task T8.
Task T8 completed.
EDF: Reordering ready queues...
No tasks available. Waiting...

```

Fonte: Elaborado pelos Autores

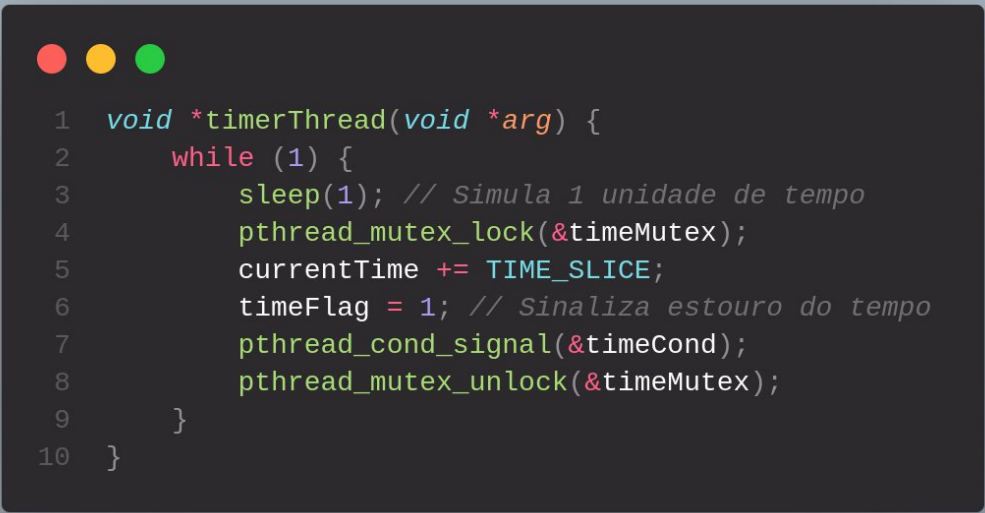
4. Códigos Importantes da Implementação

Neste tópico, detalharemos os principais trechos de código usados na implementação dos algoritmos Round Robin com Prioridade (RR_p) e Earliest Deadline First com Prioridade (EDF_p). A compreensão desses trechos é crucial para entender como cada algoritmo funciona e como eles gerenciam as tarefas e prioridades.

4.1 Round Robin com Prioridade

- **timerThread**: Esta função simula o temporizador de hardware. Ela incrementa o tempo atual (currentTime) e sinaliza o estouro de tempo (timeFlag) a cada intervalo de tempo definido.

Figura 3: Código do timer de hardware



```

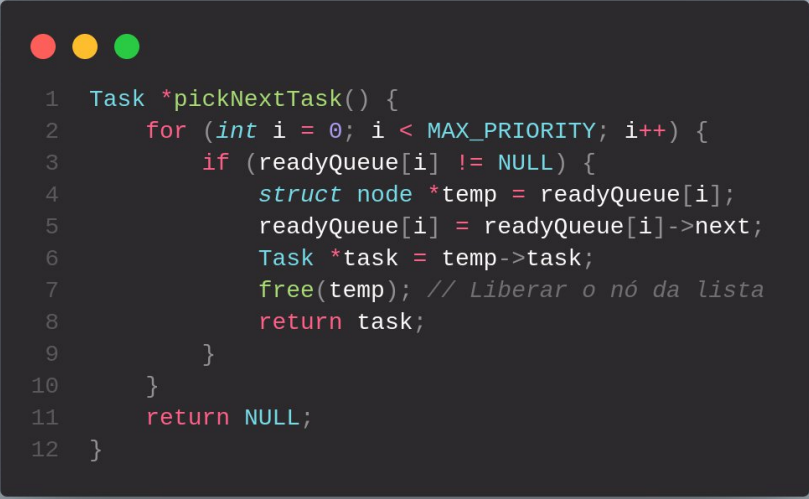
1 void *timerThread(void *arg) {
2     while (1) {
3         sleep(1); // Simula 1 unidade de tempo
4         pthread_mutex_lock(&timeMutex);
5         currentTime += TIME_SLICE;
6         timeFlag = 1; // Sinaliza estouro do tempo
7         pthread_cond_signal(&timeCond);
8         pthread_mutex_unlock(&timeMutex);
9     }
10 }

```

Fonte: Elaborado pelos Autores

- **pickNextTask**: Esta função percorre as filas de prioridade (representadas por readyQueue). Ela começa pela fila de maior prioridade (índice 0) e verifica se há tarefas disponíveis. Se encontrar uma tarefa, ela é removida da fila (usando dequeue) e retornada para execução. Caso contrário, a função continua verificando as filas de prioridades inferiores até encontrar uma tarefa disponível ou retornar NULL se todas as filas estiverem vazias.

Figura 4: Código de seleção da próxima tarefa



```

1 Task *pickNextTask() {
2     for (int i = 0; i < MAX_PRIORITY; i++) {
3         if (readyQueue[i] != NULL) {
4             struct node *temp = readyQueue[i];
5             readyQueue[i] = readyQueue[i]->next;
6             Task *task = temp->task;
7             free(temp); // Liberar o nó da lista
8             return task;
9         }
10    }
11    return NULL;
12 }

```

Fonte: Elaborado pelos Autores

- **schedule**: A função principal do escalonador. Ela itera enquanto houver tarefas na fila de aptos, selecionando a próxima tarefa a ser executada,

executando-a por um intervalo de tempo (definido por `TIME_SLICE`), e ajustando o burst da tarefa. Se a tarefa não for concluída, ela é reinserida na fila de aptos; caso contrário, a memória é liberada.

Figura 5: Código de escalonamento Round Robin com Prioridade

```
1 void schedule() {
2     pthread_t timer;
3     pthread_create(&timer, NULL, timerThread, NULL);
4
5     while (1) {
6         Task *task = pickNextTask();
7         if (task != NULL) {
8             int timeToRun = (task->burst < TIME_SLICE) ? task->burst : TIME_SLICE;
9
10            pthread_mutex_lock(&timeMutex);
11            while (timeFlag == 0) {
12                pthread_cond_wait(&timeCond, &timeMutex);
13            }
14            timeFlag = 0;
15            pthread_mutex_unlock(&timeMutex);
16
17            run(task, timeToRun); // Executa a task por 1 unidade de tempo
18            task->burst -= timeToRun;
19
20            if (task->burst == 0) {
21                printf("Task %s completed.\n", task->name);
22                free(task->name); // Liberar a string duplicada
23                free(task); // Liberar a estrutura da tarefa
24                continue;
25            }
26
27            if (task->burst > 0) {
28                // Re-adiciona a tarefa de volta à fila de aptos no fim da list (evitando starvation)
29                int queueIndex = task->priority - 1;
30                insert_end(&readyQueue[queueIndex], task);
31            }
32        } else {
33            printf("No tasks available. Waiting...\n");
34            sleep(1); // Espera um tempo antes de tentar novamente
35            break;
36        }
37    }
38 }
39
40
```

Fonte: Elaborado pelos Autores

4.2 EDF com Prioridade

- **timerThread**: Esta função simula o temporizador de hardware. Ela incrementa o tempo atual (`currentTime`) e sinaliza o estouro de tempo (`timeFlag`) a cada intervalo de tempo definido.

Figura 5: Código do timer de hardware



```

1 void *timerThread(void *arg) {
2     while (1) {
3         sleep(1); // Simula 1 unidade de tempo
4         pthread_mutex_lock(&timeMutex);
5         currentTime += TIME_SLICE;
6         timeFlag = 1; // Sinaliza estouro do tempo
7         pthread_cond_signal(&timeCond);
8         pthread_mutex_unlock(&timeMutex);
9     }
10 }

```

Fonte: Elaborado pelos Autores

- **calculateSlackTime**: Esta função calcula o tempo de folga (slack time) de uma tarefa. O tempo de folga é a diferença entre o tempo limite (tempo de chegada + deadline) e o tempo estimado de conclusão (tempo atual + burst). Este cálculo ajuda a determinar a prioridade de uma tarefa com base em sua urgência.

Figura 6: Código para calcular a folga de uma tarefa



```

1 int calculateSlackTime(Task *task) {
2     int tempo_limite = task->arrival_time + task->deadline;
3     int Tedf = currentTime + task->burst;
4     return tempo_limite - Tedf;
5 }

```

Fonte: Elaborado pelos Autores

- **reorderReadyQueue**: Esta função reordena a fila de aptos periodicamente com base nos deadlines das tarefas. Ela percorre cada fila de prioridade, recalcula o tempo de folga das tarefas e insere-as na fila correspondente, garantindo que as tarefas com deadlines mais próximos sejam priorizadas.

Figura 7: Código de reordenar a fila de aptos de acordo com o deadline


```

1 void reorderReadyQueue() {
2     struct node *newReadyQueue[MAX_PRIORITY] = {NULL};
3
4     for (int i = 0; i < MAX_PRIORITY; i++) {
5         while (readyQueue[i] != NULL) {
6             struct node *temp = readyQueue[i];
7             readyQueue[i] = readyQueue[i]->next;
8
9             int slack = calculateSlackTime(temp->task);
10            int priority_level = (slack > 0) ? temp->task->priority - 1 : 0;
11            temp->task->priority = priority_level + 1; // Define a nova prioridade no caso de reordenação
12            insert(&newReadyQueue[priority_level], temp->task);
13            free(temp);
14        }
15    }
16
17    for (int i = 0; i < MAX_PRIORITY; i++) {
18        readyQueue[i] = newReadyQueue[i];
19    }
20 }

```

Fonte: Elaborado pelos Autores

- **reorderThread**: Esta função é responsável por iniciar a reordenação das filas de aptos. Ela é executada periodicamente para garantir que a fila de aptos esteja sempre atualizada com base nos deadlines.

Figura 8: Código de thread para reordenar a fila de aptos

```

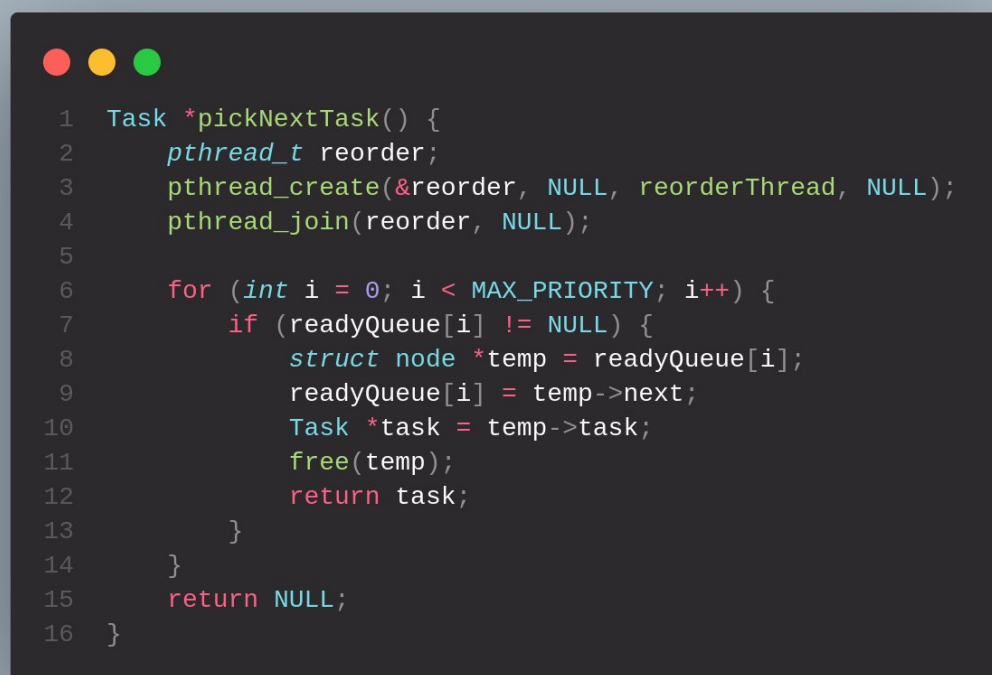
1 void *reorderThread(void *arg) {
2     printf("EDF: Reordering ready queues...\n");
3     pthread_mutex_lock(&timeMutex);
4     reorderReadyQueue();
5     pthread_mutex_unlock(&timeMutex);
6     return NULL;
7 }
8

```

Fonte: Elaborado pelos Autores

- **pickNextTask**: Esta função primeiro chama a função de reordenação (reorderReadyQueue) para garantir que a fila de aptos esteja ordenada. Em seguida, percorre as filas de prioridade para selecionar a próxima tarefa a ser executada.

Figura 9: Código de seleção da próxima tarefa



```

1 Task *pickNextTask() {
2     pthread_t reorder;
3     pthread_create(&reorder, NULL, reorderThread, NULL);
4     pthread_join(reorder, NULL);
5
6     for (int i = 0; i < MAX_PRIORITY; i++) {
7         if (readyQueue[i] != NULL) {
8             struct node *temp = readyQueue[i];
9             readyQueue[i] = temp->next;
10            Task *task = temp->task;
11            free(temp);
12            return task;
13        }
14    }
15    return NULL;
16 }

```

Fonte: Elaborado pelos Autores

- **schedule**: A função principal do escalonador EDF. Ela sincroniza com o temporizador, seleciona a próxima tarefa a ser executada, executa-a por um intervalo de tempo (TIME_SLICE), e ajusta o burst da tarefa. Se a tarefa não for concluída, ela é reinserida na fila de aptos. Se a tarefa for concluída, verifica-se se o deadline foi cumprido e, caso não tenha sido, uma mensagem de aviso é impressa.

Figura 10: Código de escalonamento EDF

```

1 void schedule() {
2     pthread_t timer;
3     pthread_create(&timer, NULL, timerThread, NULL);
4
5     while (1) {
6         // Sincroniza com o temporizador
7         pthread_mutex_lock(&timeMutex);
8         while (timeFlag == 0) {
9             pthread_cond_wait(&timeCond, &timeMutex);
10        }
11        timeFlag = 0;
12        pthread_mutex_unlock(&timeMutex);
13
14        Task *task = pickNextTask();
15        if (task != NULL) {
16            int timeToRun = (task->burst < TIME_SLICE) ? task->burst : TIME_SLICE;
17            run(task, timeToRun);
18            task->burst -= timeToRun;
19
20            int tempo_limite = task->arrival_time + task->deadline;
21            // Verifica se a tarefa perdeu o prazo antes de incrementar currentTime
22            if (currentTime > tempo_limite) {
23                printf("EDF WARNING: Deadline missed for task %s.\n", task->name);
24            }
25
26            if (task->burst > 0) {
27                int slack = calculateSlackTime(task);
28                int priority_level = (slack > 0) ? task->priority - 1 : 0;
29                insert(&readyQueue[priority_level], task);
30            } else {
31                printf("Task %s completed.\n", task->name);
32                free(task->name); // Liberar a string duplicada
33                free(task); // Liberar a estrutura da tarefa
34            }
35        } else {
36            printf("No tasks available. Waiting...\n");
37            sleep(1); // Espera um tempo antes de tentar novamente
38            break;
39        }
40    }
41 }
42

```

Fonte: Elaborado pelos Autores

5. Resultados obtidos com a implementação

Os resultados obtidos com a implementação foram organizados em tabelas para melhor visualização e análise. E a ordem das tarefas está de acordo com sua conclusão durante a execução dos códigos implementados.

Tabela 1: Round Robin com Prioridade (RR_p)

Tarefa	Prioridade	Tempo Total de Execução	Tempo de Resposta	Turnaround Time
T1	1	50	10	50
T7	1	50	10	50
T11	1	50	10	50
T2	2	50	70	50
T4	2	50	70	50
T3	3	50	150	50
T8	3	50	150	50

T5	4	50	230	50
T10	4	50	230	50
T9	5	50	310	50

Fonte: Elaborado pelos Autores

A tabela de resultados mostra que o algoritmo Round Robin com Prioridade é eficaz em priorizar tarefas de alta prioridade enquanto mantém uma alocação justa do tempo de CPU. No entanto, a penalidade para tarefas de baixa prioridade pode ser um tempo de resposta muito longo, o que pode ser um ponto de consideração importante ao aplicar este algoritmo em sistemas onde todas as tarefas precisam ser atendidas de maneira mais equitativa.

Tabela 2: EDF com Prioridade (EDF_p)

Tarefa	Prioridade	Deadline	Tempo Total de Execução	Cumpriu Deadline?
T1	1	50	10	Sim
T2	2	30	20	Sim
T7	3	50	10	Não
T11	1	50	30	Não
T3	3	100	20	Não
T8	3	80	10	Não
T5	4	110	20	Sim
T10	4	110	20	Sim
T9	5	130	10	Sim
T4	2	70	10	Não

Fonte: Elaborado pelos Autores

A análise da tabela de resultados mostra que o EDF com Prioridade é eficaz em garantir que tarefas com deadlines próximos sejam executadas, mas pode falhar em situações de alta concorrência e quando há múltiplas tarefas com prazos semelhantes. Estes achados são cruciais para a avaliação da adequação do EDF_p em sistemas de tempo real, sugerindo que, em alguns casos, pode ser necessário combinar este algoritmo com outras estratégias de gerenciamento de carga para melhorar seu desempenho global.

6. Análise e discussão sobre os resultados.

Abaixo realizamos uma análise mais aprofundada para cada algoritmo de escalonamento, usando todos os dados coletados até então.

6.1 Round Robin com Prioridade

Como todas as tarefas tinham o mesmo burst de CPU de 50 unidades de tempo, a principal diferenciação entre elas se deu em função de suas prioridades. As tarefas com maior prioridade (1) foram executadas primeiro, seguidas pelas de prioridade inferior, conforme esperado no algoritmo Round Robin com múltiplas filas de prioridade.

O tempo de resposta, definido como o tempo entre o início da simulação e o início da execução da tarefa, variou significativamente conforme a prioridade. Tarefas com prioridade 1 (T1, T7, T11) tiveram um tempo de resposta baixo, em torno de 10 unidades de tempo, indicando que foram rapidamente atendidas pelo escalonador. Em contraste, tarefas com prioridade 5 (T9) tiveram tempos de resposta muito mais altos, chegando a 310 unidades de tempo, demonstrando a deferência do escalonador a tarefas de maior prioridade.

O tempo de turnaround, que é a diferença entre o tempo total de execução e o tempo de chegada da tarefa, foi consistente para todas as tarefas devido ao burst uniforme de 50 unidades de tempo e ao tempo de chegada zero para todas. Isso significa que o tempo de turnaround para todas as tarefas foi igual ao burst, refletindo a execução completa da tarefa sem interrupções adicionais além da alocação regular do tempo de CPU.

Este comportamento é característico do algoritmo Round Robin, onde o foco está em distribuir o tempo de CPU de maneira justa entre todas as tarefas, com ajustes baseados em prioridades. No entanto, para tarefas de baixa prioridade, isso pode resultar em tempos de resposta significativamente mais longos, como evidenciado na tarefa T9.

6.2 EDF com Prioridade

Inicialmente, observa-se que o algoritmo foi capaz de cumprir os deadlines de tarefas com deadlines mais próximos. Por exemplo, a tarefa T1 com deadline de 50 unidades de tempo foi concluída a tempo. Da mesma forma, as tarefas T2 e T5, com deadlines de 30 e 110 unidades de tempo, respectivamente, também cumpriram seus deadlines, indicando que o algoritmo prioriza corretamente as tarefas urgentes.

Entretanto, há um número significativo de tarefas que não cumpriram seus deadlines. Especificamente, as tarefas T7, T11, T3, T8 e T4 falharam em cumprir seus deadlines, o que corresponde à metade das tarefas testadas. Este resultado sugere que, embora o EDF_p seja eficiente na maioria dos casos, ele pode enfrentar dificuldades quando múltiplas tarefas têm deadlines próximos. A competição entre essas tarefas pode resultar em atrasos, especialmente se elas também tiverem diferentes prioridades que influenciem a ordem de execução.

A tarefa T11, por exemplo, falhou em cumprir seu deadline de 50 unidades de tempo, mesmo tendo sido reordenada várias vezes pelo escalonador. A tarefa T7, com um deadline de 50 unidades de tempo, também não conseguiu ser concluída a tempo, o que sugere que o escalonador pode ter dificuldades em gerenciar múltiplas tarefas de alta prioridade com deadlines semelhantes.

Por outro lado, tarefas com deadlines mais longos, como T9 e T10, cumpriram seus prazos sem dificuldades, demonstrando que o algoritmo funciona bem para tarefas menos urgentes ou em cenários com menor concorrência por recursos.

7. Conclusão

A análise dos resultados obtidos com os dois algoritmos de escalonamento revela que:

- **Round Robin com Prioridade** é eficiente na distribuição de tempo do CPU entre tarefas de diferentes prioridades, garantindo que tarefas de alta prioridade recebam mais atenção. No entanto, pode não ser ideal para sistemas com requisitos rígidos de tempo real devido à sua natureza cíclica.
- **Earliest Deadline First com Prioridade** é mais adequado para sistemas de tempo real, onde o cumprimento dos deadlines é crucial. A combinação de deadlines e prioridades garante que tarefas críticas sejam executadas a tempo. No entanto, observou-se que em alguns casos, devido à concorrência entre tarefas com deadlines próximos, alguns deadlines não foram cumpridos.

Em conclusão, ambos os algoritmos têm suas vantagens e desvantagens dependendo do contexto do sistema. O RR_p é mais simples e justo em termos de tempo de CPU, enquanto o EDF_p é mais complexo, mas crucial para sistemas de tempo real com requisitos de deadlines rígidos.

Referências

- CPPREFERENCE. **C Programming Language**. Disponível em: <https://devdocs.io/c/>. Acesso em: 24 mai. 2024.
- TUTORIALSPPOINT. **C Library** -. Disponível em: https://www.tutorialspoint.com/c_standard_library/stdio_h.htm. Acesso em: 25 mai. 2024.
- CPPREFERENCE. **C Stdlib**. Disponível em: <https://devdocs.io/c/program>. Acesso em: 25 mai. 2024.
- OPENGROUP. **C Unistd**. Disponível em: <https://pubs.opengroup.org/onlinepubs/7908799/xsh/unistd.h.html>. Acesso em: 25 mai. 2024.
- OPENGROUP. **C Pthreads**. Disponível em: <https://pubs.opengroup.org/onlinepubs/7908799/xsh/pthread.h.html>. Acesso em: 25 mai. 2024.
- STUDYTONIGHT. **Round Robin Scheduling**. Disponível em: <https://www.studytonight.com/operating-system/round-robin-scheduling>. Acesso em: 25 maio 2024.
- JULIANO; GABRIELA. **ROUND ROBIN, uma técnica preemptiva de escalonamento**. Disponível em: <https://deinfo.uepg.br/~alunoso/2016/ROUNDROBIN/>. Acesso em: 25 maio 2024.
- CORDEIRO, Lucas. **Escalonamento (Algoritmos Clássicos)**. Disponível em: <https://home.ufam.edu.br/lucascordeiro/ptr/slides/03-escalonamento.pdf>. Acesso em: 25 maio 2024.