deer.uc
DEPARTAMENTO DE ENGENHARIA ELETROTÉCNICA
E DE COMPUTADORES

Sistemas de tempo Real
Trabalho Prático Nº1
Escalonamento, Atribuição de Prioridades e Cálculo dos tempos de Computação

Leonardo Gonçalves - 2020228071 PL1
Gonçalo Bastos -2020238997 PL1

Novembro 2023

Resumo

No âmbito do curso de Sistemas de Tempo Real foi proposto a implementação de uma aplicação que utiliza as rotinas POSIX para explorar a atribuição de prioridades, o escalonamento e o cálculo de tempos de computação de processos.

Tarefa 1: Medição dos Tempos de Computação

Para a tarefa inicial, o objetivo era implementar um programa que mede o tempo de computação de três funções distintas, f1, f2 e f3, cada uma representando uma tarefa periódica com requisitos específicos de tempo de computação. O programa foi projetado para executar num único processador para simular um ambiente de tempo real.

Para garantir que a aplicação corre em apenas um nucleo, defenimos uma estrutura 'cpuset' do tipo 'cpu_set _t' que vamos usar para representar o conjunto das CPU's, no caso os núcleos lógicos do sistema multiprocessado, inicializamos esse conjunto de CPU's para estar vazio, usando a macro 'CPU_ZERO' de seguida usamos 'CPU_SET()' para especificar que o processo (aplicação) seja executada somente no núcleo 0, finalmente usamos a função sched_setaffinity para definir a afinidade de CPU do processo para o conjunto de CPU's criado.

Para calcular os tempos de computação utilizamos clock_gettime com CLOCK_MONOTONIC para obter um cronometragem de alta resolução e que nunca volta atrás pois não se deixa afetar pelo comportamento do programa, com isto conseguimos obter o tempo de inicio e o tempo de fim de execução, fazendo a diferença obtemos o tempo de computação

Os tempos medidos foram os seguintes:

- Tempo de computação de f1: 30.00 milissegundos
- Tempo de computação de f2: 50.00 milissegundos
- Tempo de computação de f3: 80.00 milissegundos

Em conclusão, o código desenvolvido mede com sucesso e exibe o tempo de computação para as tarefas dadas.

Tarefa 2: Teste de escalonabilidade (RMPO and inverse)

Neste exercício vamos verificar se o sistema é escalonável usando os resultados do exercício anterior. Utilizámos dois métodos de ordenação de prioridades: o método de atribuição de prioridades por razão monótona e o seu inverso. No método de atribuição de prioridades por razão atribuímos a prioridade mais elevada à tarefa com menor período, isto é, maior frequência de ativação. Assim analisámos os períodos das 3 funções e atribuímos a prioridade mais elevada a f1, prioridade média a f2 e prioridade baixa a f3.

Utilizámos a fórmula deduzida por Audsley para calcular o tempo de resposta de cada função:

$$W_i^{(n+1)} = C_i + \sum_{j \in hp(i)} \left\lceil \frac{W_i^{(n)}}{T_j} \right\rceil \cdot C_j \tag{1}$$

Começando na tarefa de prioridade mais elevada, neste caso *f1()*:

$$w_1^0 = 30ms = R_1$$

Agora vamos calcular o tempo de cálculo da tarefa de prioridade intermédia *f2()*:

$$w_2^0 = 50 + \lceil \tfrac{50}{100} \rceil \cdot 30 = 80ms$$

$$w_2^1 = 50 + \lceil \tfrac{80}{100} \rceil \cdot 30 = 80ms$$

Como:

$$w_2^0 = w_2^1 \hookrightarrow R_2 = 80ms$$

Por último vamos calcular o tempo de resposta para a tarefa de mais baixa prioridade *f3()*:

$$w_3^0 = 80 + \lceil \tfrac{80}{100} \rceil \cdot 30 + \lceil \tfrac{80}{200} \rceil \cdot 50 = 110ms$$

$$w_3^1 = 80 + \lceil \tfrac{110}{100} \rceil \cdot 30 + \lceil \tfrac{110}{200} \rceil \cdot 50 = 190ms$$

$$w_3^2 = 80 + \lceil \tfrac{160}{100} \rceil \cdot 30 + \lceil \tfrac{160}{200} \rceil \cdot 50 = 190ms$$

Como:

$$w_3^2 = w_3^1 \hookrightarrow R_3 = 190ms$$

Com o método de escalonamento RMPO podemos verificar que as tarefas são escalonáveis, ou seja comparando os tempos de resposta para as funções $f1, f2, f3$ que são respetivamente: $R_1 = 30ms, R_2 = 80ms, R_3 = 190ms$ com os períodos: $T_1 = 100ms, T_2 = 200ms, T_3 = 300ms$, podemos verificar que todas as metas são cumpridas, logo as tarefas são escalonáveis.

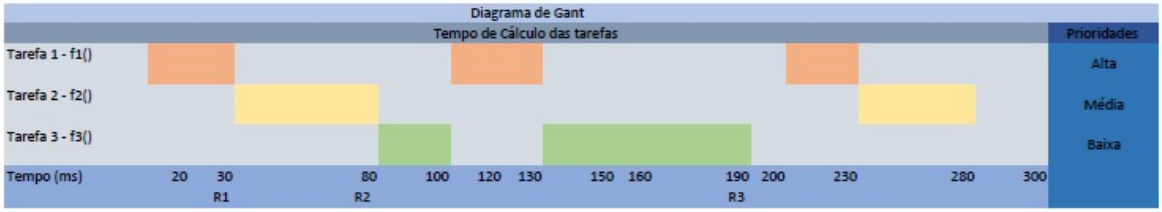

Figura 1: Gant Chart - RDMO

Agora vamos proceder ao cálculo dos tempos de resposta atribuindo as prioridades pelo método RMPO inverso, ou seja as tarefas de maior prioridade vão ser aquelas com maior período. Logo temos a tarefa *f3()* com a prioridade mais elevada, a tarefa *f2()* com prioridade intermédia e a tarefa *f3()* com a prioridade mais baixa.

Começando na tarefa de prioridade mais elevada, neste caso *f3()*:

$$w_3^0 = 80ms = R_3$$

Agora vamos calcular o tempo de cálculo da tarefa de prioridade intermédia *f2()*:

$$w_2^0 = 50 + \lceil \frac{50}{300} \rceil \cdot 80 = 130ms$$

$$w_2^1 = 50 + \lceil \frac{130}{300} \rceil \cdot 80 = 130ms$$

Como:

$$w_2^0 = w_2^1 \hookrightarrow R_2 = 130ms$$

Por último vamos calcular o tempo de resposta para a tarefa de mais baixa prioridade *f1()*:

$$w_1^0 = 30 + \lceil \frac{30}{200} \rceil \cdot 50 + \lceil \frac{30}{300} \rceil \cdot 80 = 160ms$$

$$w_1^1 = 30 + \lceil \frac{160}{200} \rceil \cdot 50 + \lceil \frac{160}{300} \rceil \cdot 80 = 160ms$$

Como:

$$w_1^1 = w_1^0 \hookrightarrow R_1 = 160ms$$

Com o método de escalonamento RMPO inverso podemos verificar que as tarefas não são escalonáveis, ou seja comparando os tempos de resposta para as funções $f1, f2, f3$ que são respetivamente: $R_1 = 80ms, R_2 = 130ms, R_3 = 160ms$ com os períodos: $T_1 = 100ms, T_2 = 200ms, T_3 = 300ms$, podemos verificar que a meta da função $f1()$ não é cumprida, logo basta uma das metas não ser cumprida para as tarefas não serem escalonáveis.

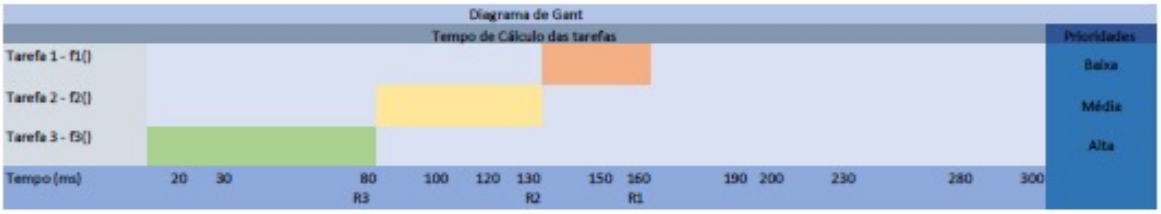

Figura 2: Gant Chart - RDMO Inverso

Tarefa 3:

Configuração da Afinidade do processador e bloqueio da memoria

Assim como no tarefa 1 aqui também configuramos o programa para correr em um único processador.

Utilizamos a função mlockall() com as flags MCL_CURRENT MCL_FUTURE para bloquear o espaço de endereçamento virtual do processo na RAM, impedindo que essa memória seja paginada para a área de swap:

- MCL_CURRENT: Esta flag indica que todas as páginas que estão atualmente mapeadas no espaço de endereçamento do processo no momento da chamada serão bloqueadas na RAM.
- MCL_FUTURE: Esta flag indica que quaisquer páginas que sejam mapeadas no espaço de endereçamento do processo após a chamada também serão bloqueadas na RAM.

Configuração das Threads

Para assegurar que o instante de inicio e de fim será igual para todas as threads, definimos a macro WAIT_TIME (1 segundo) e somamos essa mesma ao tempo de inicio do progrmama,de forma assegurar que todas as threads estão prontas a executar quando for o momento de executar.

O programa procede para configurar os atributos de cada thread. Estes atributos incluem a afinidade de CPU (para garantir que cada thread corra no CPU especificado), a política de escalonamento (FIFO neste caso, para facilitar o escalonamento em tempo real) e os parâmetros de escalonamento (que inclui a prioridade das threads).

São declarados arrays para os argumentos das threads (args), parâmetros de escalonamento (param), atributos das threads (attr) e valores de retorno das threads (retval). Os tempos de início e fim são configurados para definir a janela de execução das threads.

Para cada thread, os seus atributos são inicializados com pthread_attr_init().

A afinidade de CPU para cada thread é definida com pthread_attr_etaffinity_np().

A política de escalonamento é definida para FIFO com pthread_attr_setschedpolicy(), e a herança do atributo de escalonamento é definida explicitamente com pthread_attr_setinheritsched().

A prioridade de cada thread é definida com pthread_attr_setschedparam(), garantindo diferentes prioridades entre elas.

Criação e execução das Threads

A função pthread_create() é chamada para cada thread, passando os atributos e argumentos correspondentes. Esta função cria as threads e deixa-as prontas para correr

Os argumentos que serão passados a cada thread são preparados. Isso inclui a função que cada thread irá executar, o período de cada tarefa e os tempos de início e fim da janela de execução.

Já na função thread esta começa por converter o parâmetro void* arg de volta para o tipo struct threadArgs*, que contém os parâmetros necessários para a execução da thread (como qual função executar, o seu período e a janela de execução global).

Em seguida, aloca dinamicamente memória para um objeto struct thread_Val usando malloc(). Esta estrutura destina-se a armazenar estatísticas de execução, como contagem de execuções, prazos não cumpridos e tempos de resposta.

São feitas verificações para garantir que a alocação de memória para retval e para o seu membro miss_deadline foi bem-sucedida. Se qualquer chamada malloc() falhar, o programa imprime uma mensagem de erro e termina com falha.

A função calcula o número esperado de execuções (retval.execution_expected) com base no tempo de execução definido e no período da thread.

O loop começa com nextTime definido para o tempo de início fornecido pelo main(). Isso garante que todas as tarefas comecem a execução simultaneamente. Em cada iteração, a thread dorme até o nextTime usando clock_nanosleep(), garantindo que acorde precisamente quando for suposto executar a sua tarefa.

Ao acordar, a função executa a tarefa designada (f1, f2 ou f3) baseada no valor de args.f. Após a execução da tarefa, o tempo atual é registado para determinar se a tarefa cumpriu o seu prazo.

Calcula-se a diferença de tempo (aux) entre o momento em que a tarefa era suposto acordar (nextTime) e o tempo atual após a execução da tarefa. Esta diferença é usada para atualizar os tempos de resposta máximos (`max_time`) e mínimos (`min_time`) observados até agora. Se o tempo atual for anterior ao próximo tempo programado (`nextTime`), considera-se que a tarefa cumpriu o seu prazo, e isso é registado no array `miss_deadline`. Se não, incrementa-se a contagem de prazos não cumpridos (`retval.n_miss_deadline`).

O `nextTime` é incrementado pelo período para o próximo ciclo de execução. O loop continua até o tempo atual ser menor que o tempo de término especificado em `args.finish`. Ao sair do loop o `max_response_time`, `min_response_time` são convertidos para milissegundos e é calculado o `response_time_jitter` como sendo a diferença absoluta entre o `max_response_time` e o `min_response_time`.

A função então termina a thread chamando `pthread_exit()`, passando de volta o ponteiro para a estrutura `retval`. Este ponteiro será utilizado pela thread que faz a junção (no `main()`) para recuperar as estatísticas de execução.

Junção das Threads, Resultados e Limpeza

Após todas as threads terem sido criadas, o programa chama `pthread_join()` para cada thread, o que faz com que a thread principal aguarde até que as threads especificadas tenham completado a sua execução.

Uma vez que todas as threads tenham terminado, o programa imprime os resultados. Isso inclui o número de execuções, os tempos de resposta, o jitter e o número de prazos perdidos para cada tarefa. A memória alocada para os valores de retorno das threads é libertada, o bloqueio de memória é desfeito com `munlockall()`, e o programa termina com sucesso com `exit(EXIT_SUCCESS)`.

Tarefa 4: Comutação de Prioridades dentro das tarefas

Agora na tarefa 4 alterámos o código desenvolvido na tarefa 3, de modo a possibilitar a alternância das prioridades, entre RMPO e RMPO inverso, durante a execução das tarefas, especificamente nos instantes $t = 1,95(s) et = 3,95(s)$

A fim de implementar o pretendido editámos o código da função $void * thread(void * arg)$ de modo a detetar os instantes pretendidos e efetuar as alterações necessárias para a comutação das prioridades das tarefas, usando a função $pthread_setschedparam()$.

Análise dos Resultados

Como era espectável pela análise dos resultados do exercício 1, a tarefa não cumpre as metas temporais com escalonamento por RMPO inverso, ou seja f1 é a tarefa menos prioritária. Durante o intervalo de tempo em que as prioridades foram comutadas a função 1 não cumpriu a meta sete vezes. Além disso os tempos de cálculo das tarefas também se alteraram durante a comutação das suas prioridades, pelo que podemos comprovar os resultados obtidos na tarefa 1.
Durante o escalonamento por RMPO inverso, a função 1 (tarefa de prioridade mais baixa) teve um tempo de cálculo de 160.067ms, pelo que podemos verificar que não cumpriu a sua meta de 100ms, já no escalonamento por RMPO, agora com a prioridade mais elevada o seu tempo de cálculo foi 30.00ms. Nas outras tarefas também foi possível verificar os resultados obtidos no exercício 1. Para a função 2 o mínimo tempo de resposta foi 50ms e o máximo foi 130ms. Por último, o tempo máximo de resposta da tarefa 3 foi 190.21ms e o minimo foi 80.02ms.

Tarefa 5: Criação de um ficheiro Objeto

Neste ponto do trabalho prático foi criado um ficheiro objeto com as três funções com tempos de resposta semelhantes aos tempos de resposta calculados utilizando o ficheiro objeto fornecido. Assim utilizámos os tempos de resposta calculados no exercício 1 e desenvolvemos o ficheiro $func2.c$ que posteriormente foi compilado para gerar o ficheiro objeto $func2.o$. Utilizámos um algoritmo semelhante a *busywaiting*, ou seja, o programa não faz nada até ser alcançado o tempo definido para a tarefa. Para calcular este tempo definimos uma função que calcula o tempo decorrido desde que a tarefa entrou em execução e assim determinar o instante em que a tarefa atinge o tempo de execução desejado. Esta implementação foi feita com o desenvolvimento da função $waitTime()$. É de salientar que a execução da tarefa deve realizar um tempo de cálculo e não uma suspensão dela própria.

Para testar a nossa implementação utilizámos o código desenvolvido para o exercício 3 e conseguimos verificar que o desenvolvimento das tarefas foi realizado com sucesso, pois obtivemos tempos de cálculo similares aos calculados no exercício 3 utilizando o makefile com o ficheiro objeto original e com o novo ficheiro objeto $func2.0$.

Tarefa 6: Escalonamento Round-Robin

Nesta tarefa apenas alteramos a política (`#define POLICY SCHED_RR`) de escalonamento na etapa de configuração das threads, e mantivemos as prioridades de todas as tarefas iguais comentando a linha `// param[i].sched_priority -= i; // Agora todas as tarefas têm a mesma prioridade.`

Correndo as tarefas sob a prioridade Round Robin, verificamos que os máximos tempos de resposta seriam todos iguais, como seria de esperar para esta política de escalonamento onde todas as tarefas recebem o mesmo tempo de CPU, e ainda que os jitters aumentaram significativamente.

A transição para o escalonamento Round Robin e a igualdade de prioridades demonstrou que, embora possa ser mais justa em termos de distribuição do tempo de CPU entre as tarefas, compromete as características de tempo real de previsibilidade e confiabilidade. Os aumentos nos tempos de resposta e no jitter poderiam potencialmente levar a falhas no cumprimento de prazos em um ambiente de tempo real estrito onde as tarefas têm restrições de tempo rígidas.

Em conclusão, embora a política Round Robin ofereça uma abordagem igualitária para o escalonamento de tarefas, ela não atende bem às exigências estritas dos sistemas de tempo real, especialmente aqueles que requerem precisão de tempo e variabilidade mínima no tempo de resposta. Os resultados sugerem que uma política de escalonamento de prioridade fixa, como o FIFO, é mais adequada para aplicações onde os prazos das tarefas não podem ser comprometidos.

Fim