

FACULDADE GRAN TIETÊ
ENGENHARIA DA COMPUTAÇÃO
GUILHERME ZAMBONI DEL MENICO
LEONARDO ZOLA
RAFAEL RODRIGUES DE BARROS
VINICIUS FERNANDES ESCOBEDO

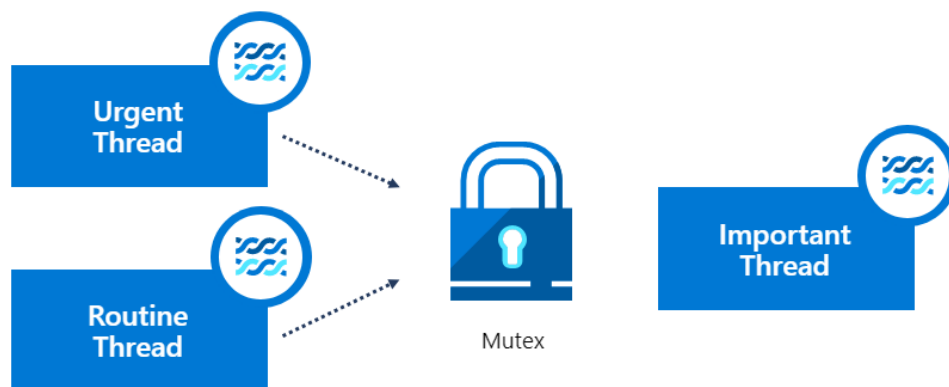
PROBLEMA DE INVERSÃO DE PRIORIDADES E SUAS SOLUÇÕES

BARRA BONITA

2023

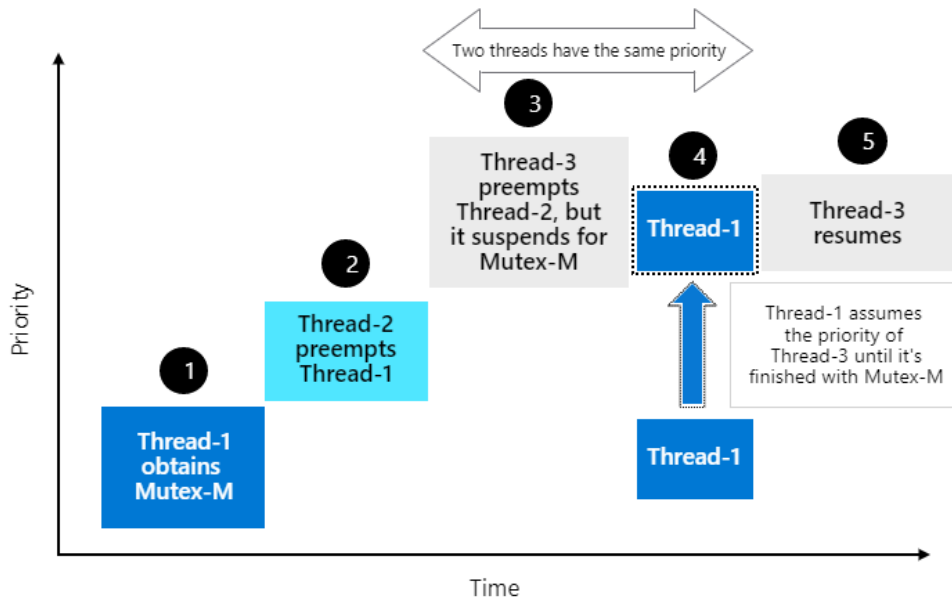
O PROBLEMA DE INVERSÃO DE PRIORIDADE

A imagem a seguir contém uma ilustração do cenário do projeto. Temos o thread urgente, o thread importante e o thread de rotina. O thread Urgente tem prioridade 1, o thread Importante tem prioridade 10 e o thread Rotina tem prioridade 15. O thread urgente e o thread de rotina precisam do mutex em vários momentos, o que é uma situação plausível. O thread importante não precisa do mutex, mas tem uma prioridade maior do que o thread de rotina, portanto, essa situação pode causar um problema de inversão de prioridade.

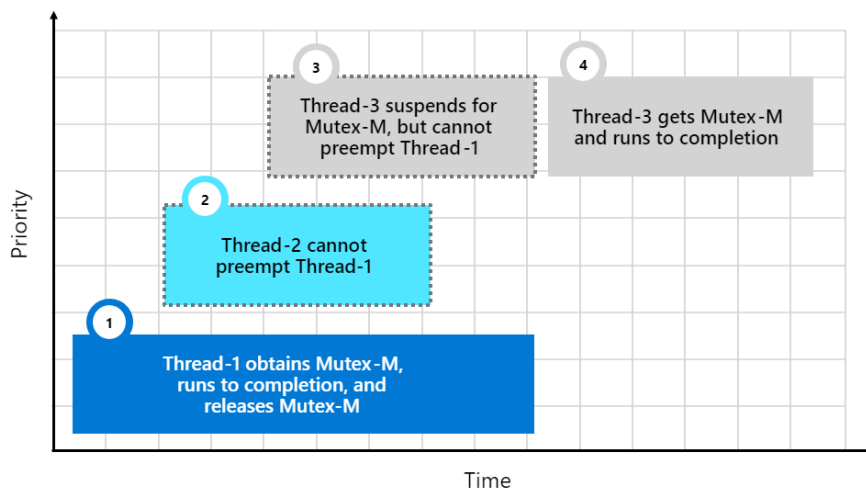


SOLUÇÕES

A maioria dos RTOSes modernos tem um recurso de herança de prioridade, assim como o ThreadX. Como o nome sugere, herdar estrategicamente uma prioridade pode eliminar o problema da inversão de prioridades. A imagem a seguir contém uma ilustração de como a herança de prioridades funciona:



O limite de preempção é um recurso avançado exclusivo do ThreadX. Este conceito foi estudado extensivamente nos trabalhos de pesquisa publicados e às vezes é chamado de protocolo de teto de prioridade. A próxima imagem contém uma ilustração de como funciona o limite de preempção:



Conforme ilustrado na imagem anterior, o uso do limite de preempção impediu que a inversão de prioridades ocorresse. Também diminuiu a quantidade de sobrecarga reduzindo o número de alternâncias de contexto, como consequência da redução do número de preempções de thread.

EXPLICAÇÕES SOBRE A IMPLEMENTAÇÃO DAS SOLUÇÕES DE INVERSÃO DE PRIORIDADE DO THREADX

Bloco 1 – Rafael Rodrigues de Barros

```
1  /*****
2  /*  Declarations, Definitions, and Prototypes  */
3  *****/
4
5  #include  "tx_api.h"
6  #include  <stdio.h>
7
8  #define    STACK_SIZE        1024
9  #define    BYTE_POOL_SIZE    9120
10 #define    DISPLAY_INTERVAL   5001
11 #define    UPDATE_INTERVAL    200
12
13 /* Define the ThreadX object control blocks */
14 TX_THREAD    Urgent_thread;
15 TX_THREAD    Important_thread;
16 TX_THREAD    Routine_thread;
17
18 TX_MUTEX      my_mutex;
19 TX_BYTE_POOL  my_byte_pool;
20 TX_TIMER      stats_timer, update_timer;
21
22 /* Define variables for Routine thread performance info */
23 ULONG         resumptions_Routine;
24 ULONG         suspensions_Routine;
25 ULONG         solicited_preemptions_Routine;
26
27 /* Define variables for Urgent thread performance info */
28 ULONG         resumptions_Urgent;
29 ULONG         suspensions_Urgent;
30
31 /* Define variables for use with mutex performance information */
32 ULONG         mutex_puts;
33 ULONG         mutex_gets;
34
35 /* Define variable for time analysis */
36 ULONG         current_time;
37
38 /* Define prototypes */
39 void          Urgent_thread_entry(ULONG thread_input);
40 void          Important_thread_entry(ULONG thread_input);
41 void          Routine_thread_entry(ULONG thread_input);
42 void          print_stats(ULONG), print_update(ULONG);
```

Declarações, definições e protótipos

#include "tx_api.h": Inclui um arquivo de cabeçalho chamado "tx_api.h" que contém todos os equivalentes do sistema, estruturas de dados e protótipos de serviço.

#include <stdio.h>: Inclui um arquivo de cabeçalho padrão da linguagem "C/C++" chamado "<stdio.h>" que fornece funções relacionadas à entrada e saída padrão, permitindo assim imprimir mensagens na tela, ler dados, abrir e manipular arquivos.

#define STACK_SIZE 1024: Define uma constante como tamanho da pilha na memória segura com o valor de 1024 bytes. A área de pilha do thread deve ser grande o suficiente para tratar seu aninhamento de chamada de função de pior caso possível e o uso de variável local.

#define BYTE_POOL_SIZE 9120: Define uma constante com tamanho de pool de bytes para 9120 bytes, não há limite para o número de pools de bytes de memória em um aplicativo. Os threads podem suspender em um pool até que a memória solicitada esteja disponível.

#define DISPLAY_INTERVAL 5001: Define uma constante com intervalo de exibição para o tempo de 5001. É a variação do intervalo de tempo, até o próximo intervalo de tempo de uma determinada tarefa, para a exibição do experimento. Não há limite para o número de temporizadores em um aplicativo. No caso de vários temporizadores, respeita-se a ordem que foram ativados.

#define UPDATE_INTERVAL 200: Define uma constante com intervalo de atualização para o tempo com valor de 200. Utilizada para controlar a frequência de atualizações de exibição.

Essa parte do cabeçalho definiu todas as bibliotecas e as constantes que serão utilizadas na aplicação. As bibliotecas padrão, já são pré-definidas na linguagem, ou pelo sistema operacional, bastando serem incluídas. As constantes foram definidas com um valor específico, onde cada vez que a constante for chamada, a mesma, será substituída por seu valor definido no cabeçalho.

Defina os blocos de controle de objeto ThreadX

TX_THREAD Urgent_thread; Define o bloco de controle de objeto da thread como "Urgent_thread" (THREAD DE URGÊNCIA)

TX_THREAD Important_thread; Define o bloco de controle de objeto da thread como "Important_thread" (THREAD DE IMPORTANCIA)

TX_THREAD Routine_thread; Define o bloco de controle de objeto da thread como "Routine_thread" (THREAD DE ROTINA)

TX_MUTEX my_mutex; Define o bloco de controle de objeto da thread MUTEX com o nome de "my_mutex". Um mutex, garante que sempre haja apenas um thread na seção crítica por vez.

TX_BYTE_POOL my_byte_pool; Define o bloco de controle de objeto da thread POOL DE BYTES DE MEMÓRIA com o nome de "my_byte_pool". Para fornecer memória disponível das pilhas aos threads.

TX_TIMER stats_timer, update_timer; Define o bloco de controle da thread TIMER (Temporizador da aplicação) com os nomes de "stats_timer, update_timer".

Definido todos os blocos de controle que serão utilizados nas solicitações de prioridades da aplicação.

Definir variáveis para informações de desempenho do THREAD DE ROTINA.

ULONG resumptions_Routine; ULONG (“Unsigned Long” – dado longo sem sinal) é um tipo de dado de chamada de serviço, utilizado para definir “resumptions_Routine”, um ponteiro para o destino do número de retomadas desse thread na aplicação de rotina.

ULONG suspensions_Routine; ULONG (“Unsigned Long” – dado longo sem sinal) é um tipo de dado de chamada de serviço, utilizado para definir “suspensions_Routine”, um ponteiro para o destino do número de suspensões desse thread nas ações de suspensão de rotina.

ULONG solicited_preemptions_Routine; ULONG (“Unsigned Long” – dado longo sem sinal) é um tipo de dado de chamada de serviço, utilizado para definir “solicited_preemptions_Routine”, uma solicitação de um ponteiro, para o destino do número de preempções, como resultado de uma chamada de serviço de API do ThreadX feita por esse thread de rotina.

*PREEMPÇÃO: Refere-se à capacidade do sistema operacional de interromper a execução de uma tarefa em execução e permitir que outra tarefa com uma prioridade mais alta seja executada. Isso é importante em sistemas de tempo real, onde tarefas críticas podem precisar ser executadas imediatamente, mesmo que outra tarefa esteja em andamento.

Definido todas as variáveis que serão utilizadas para buscar informações de desempenho das prioridades de rotina, com a utilização de ponteiros.

Definir variáveis para informações de desempenho das thread de THREAD DE URGÊNCIA.

ULONG resumptions_Urgent; ULONG (“Unsigned Long” – dado longo sem sinal) é um tipo de dado de chamada de serviço, utilizado para definir “resumptions_Urgent”, um ponteiro para o destino do número de retomadas desse thread na aplicação de urgência.

ULONG suspensions_Urgent; ULONG (“Unsigned Long” – dado longo sem sinal) é um tipo de dado de chamada de serviço, utilizado para definir “suspensions_Urgent”, um ponteiro para o destino do número de suspensões desse thread nas ações de suspensão de urgência.

Definido todas as variáveis que serão utilizadas para buscar informações de desempenho das prioridades de urgência, com a utilização de ponteiros.

Definir variáveis para uso com informações de desempenho MUTEX.

ULONG mutex_puts; ULONG (“Unsigned Long” – dado longo sem sinal) é um tipo de dado de chamada de serviço, utilizado para definir “mutex_puts”, um ponteiro para o destino do número de solicitações put executadas nesse mutex.

***TX_MUTEX_PUTS:** A operação put libera um mutex previamente obtido. Para um thread liberar um mutex, o número de operações put deve ser igual ao número de operações get anteriores.

ULONG mutex_gets; ULONG (“Unsigned Long” – dado longo sem sinal) é um tipo de dado de chamada de serviço, utilizado para definir “mutex_gets”, um ponteiro para o destino do número de solicitações de obtenção executadas nesse mutex.

***TX_MUTEX_GET:** A operação get obtém um mutex que não pertence a outro thread.

***MUTEX:** É basicamente um semáforo binário, o que significa que apenas um thread pode ter um mutex por vez. Além disso, em um mutex de propriedade, o mesmo thread pode executar uma operação get com mutex bem-sucedida várias vezes, 4.294.967.295 vezes para ser exato.

Definido todas as variáveis que serão utilizadas para buscar informações de desempenho dos mutexes, com a utilização de ponteiros.

Definir variável para análise de tempo

ULONG current_time; ULONG (“Unsigned Long” – dado longo sem sinal) é um tipo de dado de chamada de serviço, utilizado para definir “current_time”, uma variável utilizada para sincronização de tarefas, controlando os eventos da aplicação com base no tempo fornecido pelo sistema.

Definição dos protótipos:

void Urgent_thread_entry(ULONG thread_input); Cria uma função que não retorna um valor de entrada, recebendo o nome de “Urgent_thread_entry” (entrada da thread de urgência) e recebe como parâmetros da função a variável “thread_input” (entrada da thread) que é do tipo ULONG.

void Important_thread_entry(ULONG thread_input); Cria uma função que não retorna um valor de entrada, recebendo o nome de “Important_thread_entry” (entrada da thread de importância) e recebe como parâmetros da função a variável “thread_input” (entrada da thread) que é do tipo ULONG.

void Routine_thread_entry(ULONG thread_input); Cria uma função que não retorna um valor de entrada, recebendo o nome de “Routine_thread_entry” (entrada da thread de rotina) e recebe como parâmetros da função variável “thread_input” (entrada da thread) que é do tipo ULONG.

void print_stats(ULONG), print_update(ULONG); Função criada para declarar 2 argumentos, sendo uma para impressão de informações e a outra função para impressão de atualizações.

Cria uma função que não retorna um valor de entrada, recebendo o nome de “print_stats”, sendo utilizada para impressão de informações, com base no valor passado no argumento “ULONG”.

Cria uma função que não retorna um valor de entrada, recebendo o nome de “print_update”, sendo utilizada para impressão de atualizações, com base no valor passado no argumento “ULONG”.

Definido todas as funções utilizadas no decorrer da execução da aplicação, definidas para a criação de informações e atualizações dos threads de prioridades, variável para análise de tempo e impressão dos dados.

Bloco 2 – Leonardo Zola

```
1  /*****  
2  /*          Main Entry Point          */  
3  *****/  
4  /* Define main entry point.  */  
5  int main()  
6  {  
7      /* Enter the ThreadX kernel.  */  
8      tx_kernel_enter();  
9  }
```

int main(): Esta é a função principal do programa. Em C e C++, todo programa C começa a execução a partir da função main(). Ela é uma função que não recebe argumentos e retorna um valor inteiro (int).

{: Início do bloco de código da função main(). Todas as instruções dentro deste bloco serão executadas quando o programa for iniciado.

/* Enter the ThreadX kernel. */: Este é um comentário que descreve a próxima ação no código, que é entrar no kernel ThreadX. Um kernel de sistema operacional é responsável por gerenciar recursos de hardware, escalonar tarefas e fornecer um ambiente de execução para programas.

tx_kernel_enter(); : Esta linha de código chama uma função chamada **tx_kernel_enter()**. Essa função provavelmente faz parte da biblioteca ThreadX e é responsável por iniciar o kernel ThreadX. Quando essa função é chamada, o programa provavelmente passa o controle para o kernel ThreadX, que então gerenciará a execução de várias tarefas e operações em tempo real.

}: Fim do bloco de código da função main().


Em resumo, este código define a função **main()** como o ponto de entrada principal do programa e, em seguida, chama a função **tx_kernel_enter()**, que provavelmente inicia o kernel ThreadX. Esse kernel será responsável por executar tarefas e operações em tempo real no sistema.

Bloco 3 – Vinicius Fernandes Escobedo

```
1  /*****
2  /*      Application Definitions      */
3  /*****/
4  /* Define what the initial system looks like. */
5  void tx_application_define(void* first_unused_memory)
6  {
7      CHAR* Urgent_stack_ptr, * Important_stack_ptr,
8          * Routine_stack_ptr;
9
10     /* Create a memory byte pool from which to allocate
11        the thread stacks */
12     tx_byte_pool_create(&my_byte_pool, "my_byte_pool",
13         first_unused_memory, BYTE_POOL_SIZE);
14
15     /* Put system definition stuff in here, e.g., thread
16        creates and other assorted create information */
17
18     /* Allocate the stack for the Urgent_thread */
19     tx_byte_allocate(&my_byte_pool, (VOID*)&Urgent_stack_ptr,
20         STACK_SIZE, TX_NO_WAIT);
21
22     /* Create the Urgent_thread. */
23     tx_thread_create(&Urgent_thread, "Urgent_thread",
24         Urgent_thread_entry, 0, Urgent_stack_ptr,
25         STACK_SIZE, 10, 10, TX_NO_TIME_SLICE,
26         TX_AUTO_START);
27
28     /* Allocate the stack for the Important_thread. */
29     tx_byte_allocate(&my_byte_pool, (VOID*)&Important_stack_ptr,
30         STACK_SIZE, TX_NO_WAIT);
31
32     /* Create the Important_thread. */
33     tx_thread_create(&Important_thread, "Important_thread",
34         Important_thread_entry, 0, Important_stack_ptr,
35         STACK_SIZE, 15, 15, TX_NO_TIME_SLICE,
36         TX_AUTO_START);
37
38     /* Allocate the stack for the Routine_thread. */
39     tx_byte_allocate(&my_byte_pool, (VOID*)&Routine_stack_ptr,
40         STACK_SIZE, TX_NO_WAIT);
41
42     /* Create the Routine_thread.
43        **** for PREEMPTION-THRESHOLD, change 20,20 to 20,10 */
44     tx_thread_create(&Routine_thread, "Routine_thread",
45         Routine_thread_entry, 1, Routine_stack_ptr,
46         STACK_SIZE, 20, 20,
47         TX_NO_TIME_SLICE, TX_AUTO_START);
48
49     /* Create the mutex used by both threads
50        **** for PRIORITY INHERITANCE change to TX_INHERIT */
51     tx_mutex_create(&my_mutex, "my_mutex", TX_NO_INHERIT);
52
53     /* Create and activate the display timer */
54     tx_timer_create(&stats_timer, "stats_timer", print_stats,
55         0x1234, DISPLAY_INTERVAL, 0, TX_AUTO_ACTIVATE);
56
57     /* Create and activate the update timer */
58     tx_timer_create(&update_timer, "update_timer", print_update,
59         0x1234, UPDATE_INTERVAL, UPDATE_INTERVAL, TX_AUTO_ACTIVATE);
60 }
```

Nesse bloco temos as criações e alocações iniciais do sistema. Começamos criando os ponteiros que serão utilizados pelas pilhas e o pool de bytes de memória para alocar a pilha de threads. Após são feitas as alocações e criações de cada um dos threads (URGENTE, IMPORTANTE e ROTINA) e a criação do mutex que será usado pelos threads. Por fim são criados e ativados os temporizadores de status e de atualização.

Para a correção do problema de inversão de prioridades será necessário habilitar a herança de prioridades no mutex fazendo a seguinte alteração:



```
1  /* Create the mutex used by both threads
2      **** for PRIORITY INHERITANCE change to TX_INHERIT */
3      tx_mutex_create(&my_mutex, "my_mutex", TX_INHERIT);
```

Bloco 4 – Guilherme Zamboni Del Menico

```
1  /*****
2  /*      Function Definitions      */
3  *****/
4  /* Entry function definition of the Urgent thread
5  it has the highest priority */
6  void Urgent_thread_entry(ULONG thread_input)
7  {
8      while (1)
9      {
10         /* Processing, then Urgent Thread needs mutex */
11         tx_thread_sleep(4);
12
13         /* Get the mutex with suspension */
14         tx_mutex_get(&my_mutex, TX_WAIT_FOREVER);
15         tx_thread_sleep(3);
16
17         /* Release the mutex */
18         tx_mutex_put(&my_mutex);
19     }
20 }
21
22 /*****
23 /* Entry function definition of the Important_thread
24 it has the medium-level priority */
25 void Important_thread_entry(ULONG thread_input)
26 {
27     ULONG kount;
28     char letter;
29     while (1)
30     {
31         /* Important thread potentially causes priority inversion */
32
33         /* Processing time */
34         tx_thread_sleep(3);
35
36         /* simulate work by Important thread */
37         for (kount = 1; kount < 100000000; kount++) letter = 'A';
38     }
39 }
40
41 /*****
42 /* Entry function definition of the Routine_thread
43 it has the lowest priority */
44 void Routine_thread_entry(ULONG thread_input)
45 {
46     CHAR letter;
47     ULONG kount;
48     while (1)
49     {
50         /* Routine thread and Urgent thread compete for the mutex. */
51
52         /* Get the mutex with suspension */
53         tx_mutex_get(&my_mutex, TX_WAIT_FOREVER);
54
55         /* Simulate work by Routine thread */
56         for (kount = 1; kount < 100000000; kount++) letter = 'A';
57
58         /* Release the mutex */
59         tx_mutex_put(&my_mutex);
60
61         /* Processing time */
62         tx_thread_sleep(1);
63     }
64 }
65
66 /*****
67 /* Display update info at periodic times */
68 void print_update(ULONG inval)
69 {
70     current_time = tx_time_get();
71     printf("Priority Inversion Experiment--Time Remaining: %lu timer ticks ...\n", DISPLAY_INTERVAL - 1 - current_time);
72 }
73
74 /*****
75 /* print statistics at specified times */
76 void print_stats(ULONG inval)
77 {
78     /* Retrieve performance information on Routine thread */
79     tx_thread_performance_info_get(&Routine_thread, &resumptions_Routine, &suspensions_Routine,
80                                     &solicited_preemptions_Routine, TX_NULL, TX_NULL,
81                                     TX_NULL, TX_NULL, TX_NULL, TX_NULL, TX_NULL);
82
83     /* Retrieve performance information on Urgent thread */
84     tx_thread_performance_info_get(&Urgent_thread, &resumptions_Urgent, &suspensions_Urgent,
85                                     TX_NULL, TX_NULL, TX_NULL, TX_NULL, TX_NULL, TX_NULL, TX_NULL);
86
87     /* Retrieve performance information on my_mutex */
88     tx_mutex_performance_info_get(&my_mutex, &mutex_puts, &mutex_gets, TX_NULL,
89                                     TX_NULL, TX_NULL, TX_NULL);
90
91     printf("\nProjectPriorityInversion: 3 threads, 1 byte pool, 1 mutex, and 2 timers\n\n");
92
93     printf("  Routine thread resumptions: %lu\n", resumptions_Routine);
94     printf("  Routine thread suspensions: %lu\n", suspensions_Routine);
95     printf("  Routine solicited_preemptions: %lu\n", solicited_preemptions_Routine);
96
97     printf("  Urgent thread resumptions: %lu\n", resumptions_Urgent);
98     printf("  Urgent thread suspensions: %lu\n", suspensions_Urgent);
99
100    printf("      mutex puts: %lu\n", mutex_puts);
101    printf("      mutex gets: %lu\n", mutex_gets);
102
103    tx_timer_deactivate(&update_timer);
104 }
```

FUNÇÃO DE ENTRADA DO THREAD URGENTE

void Urgent_thread_entry(ULONG thread_input) Cria uma função chamada "Ugernt_thread_entry" que recebe a variável "thread_input" do tipo "ULONG" (unsigned integer) como parâmetro. Essa função será responsável pela entrada de um thread urgente.

while (1) Cria um loop while dentro da função criada anteriormente. Esse loop recebe como parâmetro "(1)" que representa uma condição verdadeira, ou seja, o 1 é utilizado para criar um loop infinito que só parará caso haja o uso de um "break".

tx_thread_sleep(4); Nesse trecho a thread é colocada em espera por quatro unidades de tempo, simulando o processamento.

tx_mutex_get(&my_mutex, TX_WAIT_FOREVER); Nessa linha de código a thread tenta receber a mutex "&my_mutex" e caso ela não consiga ela entra em espera por limite de tempo indeterminado até que ela consiga adquirir essa mutex.

tx_thread_sleep(3); Após obter o mutex, a thread entrará em espera por mais 3 unidades de tempo, simulando o processamento.

tx_mutex_put(&my_mutex); Por fim, a thread libera a mutex permitindo que outras threads acessem ela para usar o recurso.

O código define uma função de entrada para um thread de alta prioridade que executa um loop infinito. Dentro do loop, a thread introduz atrasos, adquire um mutex, executa algum processamento dentro da seção crítica protegida pelo mutex e, em seguida, libera o mutex. Esse padrão se repete indefinidamente.

FUNÇÃO DE ENTRADA DE THREAD IMPORTANTE

void Important_thread_entry(ULONG thread_input): Função que não retorna nenhum valor que recebe a entrada de uma thread de nível importante (médio) de prioridade.

ULONG kount; É criada uma variável do tipo unsigned integer que servirá como contador.

char letter; Cria uma variável do tipo caractere que receberá uma letra como valor.

while (1) Inicia um loop infinito.

tx_thread_sleep(3); Simula um processamento com o tempo de espera sendo de 3 unidades.

for (kount = 1; kount < 100000000; kount++) letter = 'A'; Essa linha de código representa um loop "for" que se repetirá até que o contador atinja o valor 100000000, e a cada ciclo o valor "A" será atribuído para a variável "letter". Essa linha simula uma tarefa sendo realizada pelo thread importante (média prioridade).

Basicamente, o código define uma função de entrada para um thread que executa um loop infinito. Dentro do loop, a thread faz uma pausa de 3 unidade de tempo do sistema e, em seguida, executa um loop intensivo de CPU, simulando um trabalho intenso. Isso é feito para simular uma tarefa sendo realizada por um thread de prioridade média. A thread importante acessa o mesmo recurso que a de rotina e a de urgência, causando assim uma inversão de prioridade pois ela acessara o recurso enquanto a de urgência terá que aguardar. Isso ocorre pois ela não está em seção crítica, diferente dos threads de prioridade baixa e alta, que estão.

FUNÇÃO DE ENTRADA DE THREAD DE ROTINA

void Routine_thread_entry(ULONG thread_input) Cria uma função que não retorna nenhum valor e recebe a entrada de uma thread de rotina (baixa prioridade).

CHAR letter; Cria uma variável do tipo caractere que receberá uma letra como valor.

ULONG kount; É criada uma variável do tipo unsigned integer que servirá como contador

while (1) Cria um loop while dentro da função criada anteriormente. Ele recebe como parâmetro “ (1) ” criando um loop infinito.

tx_mutex_get(&my_mutex, TX_WAIT_FOREVER); Nessa linha de código a thread tenta receber a mutex “&my_mutex” e caso ela não consiga ela entra em espera por limite de tempo indeterminado até que ela consiga adquirir essa mutex.

for (kount = 1; kount < 100000000; kount++) letter = 'A'; Essa parte simula uma tarefa sendo realizada pela thread de rotina, no caso a mesma tarefa que já foi explicada anteriormente na seção da thread importante.

tx_mutex_put(&my_mutex); Essa parte representa a liberação da mutex para que outra thread possa adquiri-la.

tx_thread_sleep(1); Simula um processamento com uma unidade de tempo.

Resumindo, o código define uma função de entrada para um thread com a menor prioridade. Dentro do loop infinito, a thread adquire um mutex, executa um trabalho intensivo de CPU dentro da seção crítica protegida pelo mutex, libera o mutex e faz uma pequena pausa antes de repetir o ciclo. Isso cria uma situação em que a thread de rotina compete com outras threads (como a thread "Urgent") pelo acesso ao mutex.

FUNÇÃO DE TEMPORIZADOR DE APLICATIVO PRINT_UPDATE

void print_update(ULONG inval) Cria uma função que não retorna nada e recebe como entrada a variável “inval” que é do tipo unsigned integer. Essa função será usada para imprimir informações sobre o experimento.

current_time = tx_time_get(); Cria uma variável que recebe o tempo do sistema.

printf("Priority Inversion Experiment--Time Remaining: %lu timer ticks ...\n", DISPLAY_INTERVAL - 1 - current_time); Essa linha imprime o intervalo de tempo até o próximo intervalo de tempo para a exibição do experimento. Esse intervalo é calculado pela subtração do **DISPLAY_INTERVAL** (constante com o valor de 5001) – **1 - current_time** (tempo do sistema).

Portanto, a função print_update é usada para imprimir uma mensagem que informa sobre o intervalo de tempo do experimento.

FUNÇÃO DE TEMPORIZADOR DE APLICATIVO PRINT_STATS

void print_stats(ULONG inval) cria a função que será responsável por exibir status relacionado ao experimento de inversão de prioridade.

tx_thread_performance_info_get(&Routine_thread, &resumptions_Routine, &suspensions_Routine, &solicited_preemptions_Routine, TX_NULL, TX_NULL, X_NULL, TX_NULL, TX_NULL, TX_NULL, TX_NULL); Neste trecho, são coletadas informações de desempenho sobre a thread chamada Routine_thread e essas informações são armazenadas nas variáveis resumptions_Routine, suspensions_Routine e solicited_preemptions_Routine. Isso inclui o número de vezes que a thread foi retomada (resumptions), o número de vezes que foi suspensa (suspensions) e o número de preempções solicitadas pelo thread (solicited preemptions). Essas informações de desempenho são obtidas por meio da função tx_thread_performance_info_get.

tx_thread_performance_info_get(&Urgent_thread, &resumptions_Urgent, &suspensions_Urgent, TX_NULL, TX_NULL, TX_NULL, TX_NULL, TX_NULL, TX_NULL, TX_NULL, TX_NULL); Nessa linha de código, são coletadas informações de desempenho sobre a thread chamada Urgent_thread e essas informações são armazenadas nas variáveis resumptions_Urgent e suspensions_Urgent que serão usadas para exibir essas informações posteriormente.

tx_mutex_performance_info_get(&my_mutex, &mutex_puts, &mutex_gets, TX_NULL, TX_NULL, TX_NULL, TX_NULL); Aqui, são coletadas informações de desempenho sobre o mutex chamado my_mutex, e essas informações são armazenadas nas variáveis mutex_puts e mutex_gets. Isso inclui o número de vezes que o mutex foi colocado (mutex puts) e o número de vezes que foi obtido (mutex gets).

printf("\nProjectPriorityInversion: 3 threads, 1 byte pool, 1 mutex, and 2 timers\n\n"); Imprime informações sobre o experimento como o número de threads, número de byte pool, mutex e temporizadores.

printf(" Routine thread resumptions: %lu\n", resumptions_Routine); Exibe o número de vezes que a thread de rotina foi retomada.

printf(" Routine thread suspensions: %lu\n", suspensions_Routine); Exibe o número de vezes que a thread de rotina foi suspensa.

printf("Routine solicited_preemptions: %lu\n\n", solicited_preemptions_Routine); imprime o número de preempções solicitadas pelas threads.

printf(" Urgent thread resumptions: %lu\n", resumptions_Urgent); Exibe o número de vezes que a thread de urgência foi retomada.

printf(" Urgent thread suspensions: %lu\n\n", suspensions_Urgent); Exibe o número de vezes que a thread de urgência foi suspensa.

printf(" mutex puts: %lu\n", mutex_puts); Imprime o número de vezes a mutex foi colocada.

printf(" mutex gets: %lu\n\n", mutex_gets); Exibi o número de vezes que a mutex foi obtida.

tx_timer_deactivate(&update_timer); Por fim, o timer de update criado no bloco de construção três é desativado.

A função print_stats é usada para imprimir estatísticas de desempenho relacionadas a threads e mutexes, bem como informações sobre o contexto do sistema.