SISTEMA OPERACIONAL COM PERMISSÃO DE EXECUÇÃO CONTÍNUA BASEADO EM WATCHDOG

Eduardo Augusto Ribeiro Dueñas
Orientador: Rodrigo Maximiano Antunes de Almeida
Instituto de Engenharia de Sistemas e Tecnologia da Informação
Universidade Federal de Itajubá
Itajubá, Brasil, 2023
eduardo.aug.rib.duenas@gmail.com

***Abstract*—This work is the development, analysis and results related to implementation of permanence function inside of an Operational System . The premise of the project is a system that allows a task to stay in the processor in a determined period without being exchanged for a higher priority task on a context switch. The objective is to allow a lower priority task to execute without being switched out for a higher priority without lowering the systems robustness and ensuring safety against race conditions on the critical sections.**

***Index Terms*—embedded operating system, real-time operating system, FreeRTOS, preemption, watchdog, race safety**

***Resumo*—Esse trabalho consiste no desenvolvimento, análise e resultado da implantação de uma função de permanência num Sistema Operacional. A ideia do projeto é um sistema que permite uma tarefa executar durante um período determinado no processador sem ser substituída por outra tarefa em uma troca de contexto. O objetivo é fazer possível que uma tarefa de baixa prioridade seja executada sem ser substituída por uma de maior prioridade e sem diminuir a robustez do sistema, garantindo segurança contra race condition nas seções críticas.**

***Index Terms*—sistema operacional embarcado, sistema operacional de tempo real, FreeRTOS, preempção, watchdog, segurança contra race condition**

I. Introdução

Os SOs (Sistemas Operacionais) podem ser divididos em duas categorias, os de núcleo preemptivo e os de núcleo cooperativo, ou não preemptivo. Os SOs de núcleo cooperativo funcionam de forma amistosa, tendo que uma tarefa em execução ativamente conceder o processador para que outra tarefa possa executar. Eles têm a vantagem de serem menos complexos uma vez que uma tarefa sabe quando será interrompida por outra, apresentando menos conflitos de recursos compartilhados. Porém, devido à necessidade de ceder o processador, não se pode garantir a latência de tarefas mesmo que de alto nível, por depender que o código em execução ceda o processador [1].

Por outro lado, os SOs de núcleo preemptivo apresentam uma interrupção de troca de contexto, trocando as tarefas durante sua execução pela próxima da fila que pode ser determinada por múltiplos fatores e atualizada a cada interrupção. Dessa forma, como o escalonador define a próxima tarefa a ser executada pela prioridade, entre outros fatores, diminui-se o tempo que uma tarefa, especialmente de alta prioridade, necessita para receber o processador, caso o escalonador acredite que ela deva ser executada, diminuindo a latência para altas prioridades. [1]

Uma das desvantagens de um sistema de núcleo cooperativo é sua menor robustez, pois a tarefa tem que ativamente ceder o processador. Portanto, se a tarefa não devolver o controle do processador, por estar travada ou por algum erro, as outras tarefas também não podem rodar. Dessa forma o sistema fica travado em sua totalidade por causa de uma tarefa (*timeline break*) [2].

Seções críticas de código, como definido por Dijkstra, são recursos compartilhados por diferentes entidades o qual só pode ter uma o utilizando por vez [3]. Nessas seções pode-se observar um problema similar ao de sistemas cooperativos caso a utilização de ferramentas como o *mutex* não seja possível. Nesses casos as formas comumente utilizadas para garantir uma execução segura dessa seção crítica de código é a pausa do escalonador e em casos mais graves a desabilitação das interrupções, o que impede a preempção das tarefas.

Uma *mutex lock* é uma solução aos problemas de exclusão mútua, ou seja, ela impede que mais de uma tarefa esteja dentro da mesma seção crítica de código simultaneamente [4]. Essas *mutex locks* funcionam com uma aquisição e uma devolução, ou seja, uma tarefa pega a permissão para executar e ao final a devolve. Isso, se utilizado corretamente, garante que apenas uma tarefa pode executar a seção crítica associada por vez [5]. Contudo, esse funcionamento acaba por ter problemas similares a um sistema cooperativo. Como a *mutex locks* depende que a tarefa devolva a permissão para que o próximo possa usar o recurso, o recurso fica indisponível se for respeitado o *mutex* a fim de garantir segurança contra *race conditions*. Porém, como o recurso indisponível se limita a uma seção específica do código e como ele não trava o sistema este problema pode ser tratado.

Mas o que é uma *race condition*? De forma geral, uma *race condition* ocorre quando duas ou mais *threads*, ou no caso tarefas, executam concorrentemente ou paralelamente

utilizando um recurso compartilhado sem sincronização. Elas podem resultar em diversos erros como corrompimento de informações ou até falhas no sistema [7].

Como o *mutex* tenta proteger um recurso específico, ele não impede que outras tarefas executem entre o início e o fim da seção crítica, desde que elas não necessitem utilizar o recurso protegido. Isso pode ser um problema caso o tempo de início e de finalização seja de suma importância, ou caso a execução de outras tarefas possam apresentar *race conditions*.

O objetivo do projeto é criar uma forma alternativa de evitar que um segmento de código apresente uma *race condition*, fazendo que apenas uma tarefa execute durante a seção crítica. A implementação utiliza mecanismos presentes no SO para que o programa seja modular e não altere o funcionamento geral do sistema.

II. Proposta e Metodologia

A proposta deste projeto é criar uma prova de conceito no código do *FreeRTOS* de um módulo que possibilite que uma tarefa se mantenha por um período em execução no processador, sem que outras tarefas sejam escalonadas, sem desabilitar as interrupções ou pausar o escalonador, o que deixaria o sistema vulnerável.

Esse problema não poderia ser resolvido com uma simples reconfiguração das prioridades inicial das tarefas, uma vez que isso interferiria em toda a ordem de prioridades, o que pode não ser possível ou desejável.

A ideia central é a implementação de um programa que, utilizando de recursos disponíveis no próprio sistema operacional, permite que a tarefa seja executada sem que o kernel a retire de execução. Como o processador é utilizado por um tempo limitado, torna-se possível um tratamento de erro durante a execução do sistema caso a tarefa não termine no tempo esperado, impedindo que uma falha no trecho crítico comprometa todo o sistema. Essa ideia se assemelha ao conceito de um *watchdog timer* que é um circuito de timer que, caso não receba um *reset* dentro de seu tempo periodicamente, inicia um processo de recuperação do sistema [6].

Uma vez que o sistema não é comprometido é possível um tratamento de erro da tarefa afetada. Com isso em mente, foi utilizado um aumento temporário de prioridade da tarefa solicitada, a fim de evitar interrupções e retornando a tarefa a sua prioridade original ao final do período solicitado.

A não utilização da desabilitação de interrupções ou pausa do escalonador permite que o sistema não tenha vulnerabilidades inerentes a um sistema cooperativo como *timeline break* ou perda total do sistema decorrente do erro em uma tarefa. O retorno da prioridade da tarefa ao seu valor anterior é feito automaticamente pela implementação do programa. Isto permite que se tenha o comportamento de uma pausa no escalonador, sem os problemas inerentes de segurança caso a tarefa não retorne o funcionamento do escalonador.

A implementação será feita na linguagem de programação *C* e *C++* no ambiente de desenvolvimento *Visual Studio 2022* e testado com a utilização do simulador nativo do *FreeRTOS* para a IDE (*Integrated Development Environment*), *WIN32-MSVC*, pela facilidade de teste e apuração fornecida pelo simulador.

III. Desenvolvimento

O projeto consiste em duas partes, um decrementador e uma função de requisição de tempo. A implementação foi feita no formato de uma biblioteca nomeada “TimeStretch” e será dividida em suas duas partes que serão descritas em suas respectivas seções.

Para a implementação no *FreeRTOS* foi necessário algumas adições no arquivo “FreeRTOSConfig.h”, que são mostradas na Figura 1.

```
#define configUSE_PREEMPTION                    1
#define configUSE_MUTEXES                       1
#define configSUPPORT_STATIC_ALLOCATION         1
#define configSUPPORT_DYNAMIC_ALLOCATION        1
#define configUSE_TICK_HOOK                     1
#define configUSE_TIMERS                        1
#define INCLUDE_vTaskPrioritySet                1
#define INCLUDE_uxTaskPriorityGet               1
#define INCLUDE_vTaskSuspend                    1
#define INCLUDE_xEventGroupSetBitFromISR        1
#define INCLUDE_xTimerPendFunctionCall          1
```

Figura 1. Configurações adicionadas ao arquivo “FreeRTOSConfig.h”.

Para a implementação da biblioteca “TimeStretch.h” foi necessário incluir as bibliotecas apresentadas na Figura 2 com exceção a “conio.h” que é usada apenas para facilitar a visualização do caso teste.

```
#include "FreeRTOS.h"
#include "task.h"
#include "semphr.h"
#include <stdio.h>
#include <conio.h>
```

Figura 2. Bibliotecas para o arquivo “TimeStretch.h”.

A. Requisição de Tempo

O tempo pode ser requisitado pela função “uxAskForTicks”, incluída na biblioteca “TimeStretch.h”, em múltiplos do *tick* do RTOS. Essa função pode receber o *handle* de um semáforo para uma tarefa de *failsafe* para tratar a tarefa original caso ocorra algum erro e ela não retorne dentro do tempo pedido. Essa chamada para a tarefa de *failsafe* pode ser implementada de diversas formas, mas como o objetivo deste trabalho é a prova de conceito foi utilizado um semáforo com uma tarefa de alta prioridade por sua simplicidade.

A função “uxAskForTicks” recebe três parâmetros, como mostrado na Figura 3. O primeiro parâmetro é “uxTickNumb” que representa o número de *ticks* pedido pela tarefa. O segundo

parâmetro é “pxTask” que é o *handle* da tarefa que precisa de mais tempo. O último parâmetro é “xFailSafeSemaphore” que é o *handle* do semáforo da tarefa de *failsafe*.

```
UBaseType_t uxAskForTicks(
    UBaseType_t uxTickNumb,                 //numero de ticks pedidos
    TaskHandle_t pxTask,                    //guia da tarefa a receber ticks
    SemaphoreHandle_t xFailSafeSemaphore);  //guia do semaforo de failsafe
```

Figura 3. Função “uxAskForTicks”.

O primeiro passo da função é a criação, caso não exista, e a aquisição do *mutex*, salvo na variável “pxSemaphore”, para que a função não apresente *race condition*. Em seguida, caso não seja passado o *handle* da tarefa, é pega o *handle* da tarefa atualmente em execução e em ambos os casos ela é salva em uma variável global nomeada “pxCurrentTask”. Caso ainda assim a variável seja *NULL* a função é finalizada e retorna “pdFAIL”. A seguir, os outros parâmetros são salvos em variáveis globais, o ponteiro do semáforo da tarefa de *failsafe* é salvo na variável “xCurrentFailSafeSemaphore”, a prioridade da tarefa em questão é salva na variável “uxPriorPriority”, a prioridade da tarefa é alterada para a segunda maior, é ligada uma variável global “uxIsRunning” que indica que a função está sendo usada e é retornado “pdPASS”. Essa seção de código é mostrada na Figura 4.

```
if (pxSemaphore == NULL)vInitTask();
if (xSemaphoreTake(pxSemaphore, portMAX_DELAY) == pdFAIL) return pdFAIL;

if (pxTask == NULL) pxCurrentTask = xTaskGetCurrentTaskHandle();
else pxCurrentTask = pxTask;
if (pxCurrentTask == NULL) {
    vFinishTicksTask();
    return pdFAIL;
}

xCurrentFailSafeSemaphore = xFailSafeSemaphore;
uxPriorPriority = uxTaskPriorityGet(pxCurrentTask);
vTaskPrioritySet(pxCurrentTask, (configMAX_PRIORITIES - 1));
uxTickCoun = uxTickNumb;
uxIsRunning = pdTRUE;
return pdPASS;
```

Figura 4. Implementação da função “uxAskForTicks”.

A função pela qual a requisição de tempo é finalizada é chamada “vFinishTicksTask”, ela não recebe nenhum parâmetro e não retorna nenhum valor. Essa função volta os valores das variáveis globais ao seu valor inicial e devolve o *mutex* para preparar a requisição de tempo para uma nova execução. O código dessa função é apresentado na Figura 5.

B. Decrementador

O decrementador é uma função nomeada “vDecTickCount” que é executada em toda interrupção de *tick* por meio do *tick* hook, como pode ser visto na Figura 6. Essa função não recebe parâmetros e não retorna nenhum valor. Sua definição pode ser vista na Figura 7.

A função consiste em três condicionais, a primeira verifica se a função “uxAskForTicks” está sendo usada pela variável global “uxIsRunning”. Caso esteja em uso, a função faz uma impressão para visualização no caso teste e decrementa o contador de *tick* pedidos “uxTickCoun”. A próxima condicional

```
void vFinishTicksTask(void) {
    uxIsRunning = pdFALSE;
    uxTickCoun = 0;
    vTaskPrioritySet(pxCurrentTask, uxPriorPriority);
    pxCurrentTask = NULL;
    uxPriorPriority = 0;
    xCurrentFailSafeSemaphore = NULL;
    xSemaphoreGive(pxSemaphore);
    return;
}
```

Figura 5. Implementação da função “vFinishTicksTask”.

```
void vApplicationTickHook( void )
{
    /* This function will be called by each tick interrupt if
    * configUSE_TICK_HOOK is set to 1 in FreeRTOSConfig.h.  User code can be
    * added here, but the tick hook is called from an interrupt context, so
    * code must not attempt to block, and only the interrupt safe FreeRTOS API
    * functions can be used (those that end in FromISR()). */

    vDecTickCount();

    #if ( mainCREATE_SIMPLE_BLINKY_DEMO_ONLY != 1 )
        {
            vFullDemoTickHookFunction();
        }
    #endif /* mainCREATE_SIMPLE_BLINKY_DEMO_ONLY */
}
/*-----------------------------------------------------------*/
```

Figura 6. Função “vDecTickCount” dentro do *tick* hook.

dentro da anterior verifica se o tempo pedido terminou, ou seja, se “uxTickCoun” é igual a 0. Caso o tempo pedido tenha se esgotado e caso o ponteiro para o semáforo do *failsafe* não seja *NULL*, ou seja, a variavel “xCurrentFailSafeSemaphore” seja diferente de *NULL*, é dado um “give” no semáforo, ou seja, é liberada a função de *failsafe* para rodar. Independentemente do valor de “xCurrentFailSafeSemaphore” é chamada a função “vFinishTicksTask” para finalizar a requisição de *tick*. A implementação da função “vDecTickCount” pode ser vista na Figura 8.

```
void vDecTickCount(void);
```

Figura 7. Função “vDecTickCount”.

```
void vDecTickCount(void) {
    if (uxIsRunning == pdTRUE) {
        printf("vDecTickCount\n");
        uxTickCoun--;
        if (uxTickCoun == 0) {
            if (xCurrentFailSafeSemaphore != NULL) {
                xSemaphoreGiveFromISR(xCurrentFailSafeSemaphore, NULL);
            }
            vFinishTicksTask();

        }
    }
}
```

Figura 8. Implementação da função “vDecTickCount”.

C. Caso Teste

O caso de teste consiste em três funções sem um propósito específico, servindo apenas para mostrar a funcionalidade do código. As três são iniciadas como tarefa na função *main* do arquivo “main.c” normalmente, como pode ser visto na Figura 9.

```
/* My_Main */
xTaskCreate(prvSayOneTask, "SO", 10*configMINIMAL_STACK_SIZE, NULL, 3, NULL);
xTaskCreate(prvSayTenTask, "ST", 10*configMINIMAL_STACK_SIZE, NULL, 2, &xHandle_prvSayTenTask);
xTaskCreate(prvFailSafeTask, "FS", 10*configMINIMAL_STACK_SIZE, NULL, 5, NULL);


vTaskStartScheduler();

for (;;);
```

Figura 9. Iniciação das três tarefas do caso teste.

A primeira função, nomeada "prvSayOneTask", é uma função iniciada com prioridade 3, tendo em vista que a prioridade do *FreeRTOS* é crescente iniciando em 0. A função consiste em um loop infinito que imprime "_01_"a cada 2 *ticks*. Sua implementação pode ser vista na Figura 10.

```
static void prvSayOneTask(void* pvParameters) {
    for (;;) {
        printf("_01_\n");
        vTaskDelay(2);
    }
}
```

Figura 10. Implementação da função “prvSayOneTask”.

A segunda função é nomeada "prvFailSafeTask” e foi iniciada com prioridade 5. Essa função cria um semáforo e salva o *handle* em uma variável global. Caso o ponteiro retorne *NULL* seria necessário um tratamento que por depender da aplicação não foi implementado. Em seguida, a função entra em um loop infinito em que ela tenta obter o semáforo por tempo indefinido e, caso obtenha, suspende a tarefa “prvSayTenTask” e imprime "_Fail_”. Sua implementação pode ser vista na Figura 11.

```
static void prvFailSafeTask(void* pvParameters) {
    xSemaphoreSayTenTask = xSemaphoreCreateBinary();
    if (xSemaphoreSayTenTask == NULL) {
        //implementação de fail safe
    }
    for (;;) {
        xSemaphoreTake(xSemaphoreSayTenTask, portMAX_DELAY);
        vTaskSuspend(xHandle_prvSayTenTask);
        printf("_Fail_\n");
    }
}
```

Figura 11. Implementação da função “prvFailSafeTask”.

A terceira função, que tem o nome "prvSayTenTask", é iniciada com prioridade 2. A função declara algumas variáveis auxiliares e entra em um loop infinito. Dentro desse laço a função pede 10 *tick* com a função “uxAskForTicks”, passando o semáforo da função “prvFailSafeTask” como *failsafe*. Após receber os *ticks*, a função imprime "_(%i)_10_Start_", faz contas para gastar tempo de processamento um número de vezes igual ao quadrado do número de iterações do laço, imprime "(%i)_10_Finish_", finaliza o tempo pedido e espera 19 *ticks*, sendo %i o número de iterações. Sua implementação pode ser vista na Figura 12.

```
static void prvSayTenTask(void* pvParameters) {
    unsigned int n = 0;
    unsigned int i = 0;
    unsigned int j = 0;
    double aux = 0;
    for (;;) {
        n++;
        uxAskForTicks(10, NULL, xSemaphoreSayTenTask);
        printf("_(%i)_10_Start_\n", n);
        for (i = 0; i <= n; i++) {
            for (j = 0; j <= n; j++) {
                aux = sin(sin(sin((aux * aux * i + 1) / (aux + j + 1)) + 20) - 10) / ((double) ((i + 1) / (j+1)));
            }
        }
        printf("(%i)_10_Finish_\n", n);
        vFinishTicksTask();
        vTaskDelay(19);
    }
}
```

Figura 12. Implementação da função “prvSayTenTask”.

A interação dessas funções foi pensada de forma que a função “prvSayOneTask” tem maior prioridade e maior frequência de modo que a princípio ela sempre executaria inclusive no meio da iteração do loop de “prvSayTenTask” se não fosse por seu aumento de prioridade. Como a função “prvSayTenTask” faz a requisição de *tick*, ela é executada sem ser interrompida apesar da prioridade de “prvSayOneTask” ser inicialmente maior. Como o tempo de execução de “prvSayTenTask” aumenta a cada iteração, em algum momento ela ultrapassa o tempo de 10 *ticks* pedido perdendo as vantagens adquiridas com “uxAskForTicks” e a função “prvFailSafeTask” será habilitada. Com isso veremos a função “prvFailSafeTask” ser executada.

A Figura 13 podemos ver o fluxograma do comportamento da funcionalidade de permanência uma vez acionada, sendo apenas um diagrama lógico, representando o conjunto de múltiplas funções.

Na Figura 14 podemos ver uma comparação entre o funcionamento ideal do sistema, imaginando um tempo pedido de 5 ticks e com a “prvSayTenTask” ultrapassando esse tempo, com a utilização do programa, sem a utilização do programa e com a pausa do escalonador.

Podemos ver na Figura 15 o sistema no início de sua execução, na qual a tarefa “prvSayOneTask” executa entre 9 e 10 vezes entre as execuções da tarefa “prvSayTenTask” que executa em menos de um *tick*.

Na Figura 16, vemos um aumento do uso do tempo de execução da tarefa “prvSayTenTask” que já demora múltiplos *ticks*, como podemos ver pela impressão “vDecTickCount”.

Na Figura 17 já podemos ver o momento em que a tarefa “prvSayTenTask” ultrapassa a quantidade de *ticks* pedidos.

Por último, vemos na Figura 18 que a tarefa “prvSayTenTask” foi suspensa e não é mais executada.

IV. Resultado

O caso teste demonstrou um bom funcionamento do código impedindo sua interrupção por tarefas de maior prioridade durante o período solicitado e, uma vez este terminado, seja por a tarefa retornar ou o tempo se esgotar, o sistema volta a seu funcionamento comum. O sistema, desta forma, alcançou

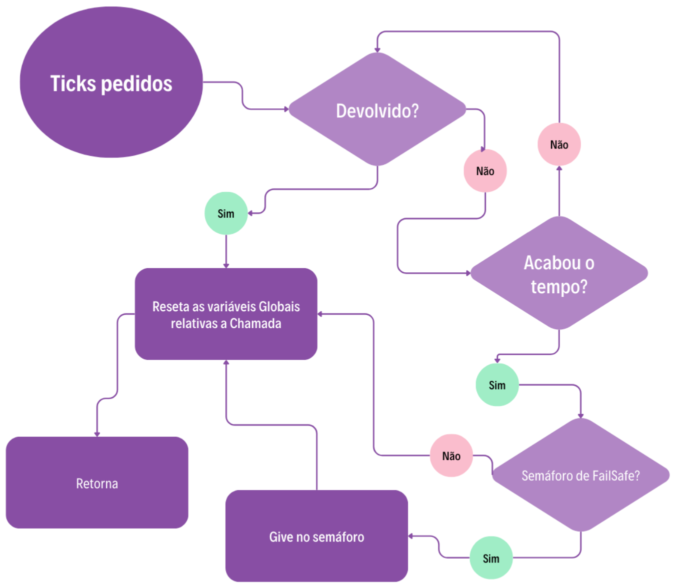
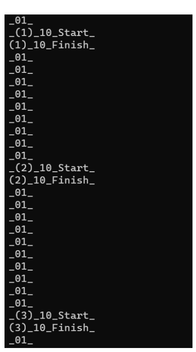


Figura 13. Fluxograma lógico do comportamento da funcionalidade de permanência.

time line	17 Tick	18 Tick	19 Tick	20 Tick	21 Tick	22 Tick	23 Tick	24 Tick	25 Tick	26 Tick	27 Tick	28 Tick	29 Tick	32 Tick
com projeto	1 c	- c	10 c	10 c	10 c	10 c	10 c	x c	1 c	- c	1 c	- c	1 c	- c
sem projeto	1	-	10	1	10	1	10	1	10	1	10	1	10	1
suspendendo o scheduler	1	-	10	10	10	10	10	10	10	10	10	10	10	10

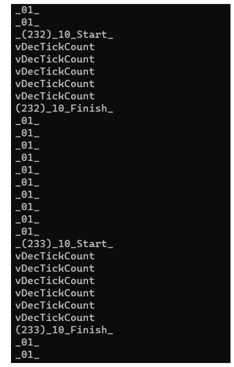
SayOneTask	1
SayTenTask	10
FailSafeTask	x
DecTickCount	c
Idle	-

Figura 14. Comparação entre funcionamentos ideais do sistema.



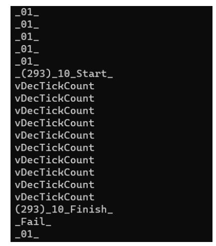
```
_01_
_(1)_10_Start_
(1)_10_Finish_
_01_
_01_
_01_
_01_
_01_
_01_
_01_
_01_
_01_
_(2)_10_Start_
(2)_10_Finish_
_01_
_01_
_01_
_01_
_01_
_01_
_01_
_01_
_01_
_01_
_(3)_10_Start_
(3)_10_Finish_
_01_
```

Figura 15. Impressões iniciais do programa no console.



```
_01_
_01_
_(232)_10_Start_
vDecTickCount
vDecTickCount
vDecTickCount
vDecTickCount
vDecTickCount
(232)_10_Finish_
_01_
_01_
_01_
_01_
_01_
_01_
_01_
_01_
_01_
_01_
_(233)_10_Start_
vDecTickCount
vDecTickCount
vDecTickCount
vDecTickCount
vDecTickCount
vDecTickCount
(233)_10_Finish_
_01_
_01_
```

Figura 16. Impressões do programa no console após alguns segundos.



```
_01_
_01_
_01_
_01_
_01_
_(293)_10_Start_
vDecTickCount
vDecTickCount
vDecTickCount
vDecTickCount
vDecTickCount
vDecTickCount
vDecTickCount
vDecTickCount
vDecTickCount
vDecTickCount
(293)_10_Finish_
_Fail_
_01_
```

Figura 17. Impressões do programa no console ao "prvSayTenTask" ultrapassar o tempo pedido.

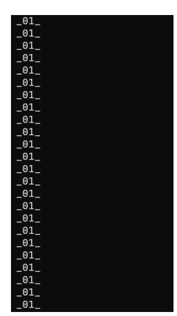
seu objetivo sem criar vulnerabilidades ao funcionamento do sistema operacional.

O caso teste também demonstrou o funcionamento da tarefa de *failsafe*, que foi executada no momento correto e suspendendo a tarefa como desejado.

Dada a natureza experimental do caso teste, há diversas possíveis adaptações ao código que melhor adequariam a diferentes objetivos, tal como, uma tarefa de *failsafe* que reiniciasse a tarefa referente. Porém como essas mudanças deixarem o sistema melhor apenas para alguns casos, foi deixado para que essas alterações sejam feitas ao se implementar a função para que as necessidades do projeto sejam melhor atendidas.

V. CONCLUSÃO

Vimos no decorrer do artigo as vantagens de um sistema que não desabilita as interrupções ou pausa o escalonador.



```
_01_
_01_
_01_
_01_
_01_
_01_
_01_
_01_
_01_
_01_
_01_
_01_
_01_
_01_
_01_
_01_
_01_
_01_
_01_
_01_
_01_
_01_
_01_
_01_
```

Figura 18. Impressões do programa no console ao "prvSayTenTask" ter sido suspensa.

O sistema sugerido resolve de forma relativamente eficiente a aplicação, demonstrando a funcionalidade da solução proposta, de forma genérica o suficiente para ser aplicável ao maior número de casos, com suas devidas considerações e alterações tendo em vista as necessidades de cada aplicação.

Além disso, o programa pode ser implementado para distintos *hardwares* com as devidas adequações de configuração do *FreeRTOS*. Da mesma forma, a ideia pode ser adaptada para outros SOs, tendo em vista suas características e funcionalidades específicas. Foi observada também a possibilidade da aplicação do código proposto, com algumas alterações, na aplicação de um mascaramento de tarefas de até determinada prioridade, podendo essa prioridade ser ajustada ou variável. Apesar de não se ter estudado o caso, é uma possibilidade promissora de aplicação.

REFERÊNCIAS

[1] G. W. Denardin e C. H. Barriquello. Sistemas operacionais de tempo real e sua aplicação em sistemas embarcados. São Paulo: Editora Blucher, 2019.

[2] M. Short e I. Sheikh. Timely recovery from task failures in non-preemptive, deadline-driven schedulers. In: 2010 10th IEEE International Conference on Computer and Information Technology. IEEE, 2010.

[3] E. W. Dijkstra. Solution of a problem in concurrent programming control. Communications of the ACM, 8(9):569, Setembro de 1965.

[4] W. H. Hesselink e M. IJbema. Starvation-free mutual exclusion with semaphores. Formal Aspects of Computing. 2013.

[5] R. Chawla. The problem of mutual exclusion: a new distributed solution. 1991.

[6] A. M. El-Attar e G. Fahmy. An improved watchdog timer to enhance imaging system reliability in the presence of soft errors. IEEE International Symposium. 2007.

[7] D. Cunningham, S. Drossopoulou e S. Eisenbach. Universes for Race Safety. VAMP. 2007