Sistemas Embarcados - Trabalho Prático 1

Társio Onofrio Cardoso da Silva

30 de Maio de 2020

Escalonador de tarefas de tempo real utilizando interrupções

Professor: Sergio Johann Filho
Disciplina: 98G00-04 - Sistemas Embarcados
Turma:590

1 Organização

O trabalho está organizado nos arquivos em torna de algumas estruturas de dados e em três processos: criação, partida e troca de contexto, respectivamente as funções `rt_create`, `rt_start` e `rt_context_switch`.

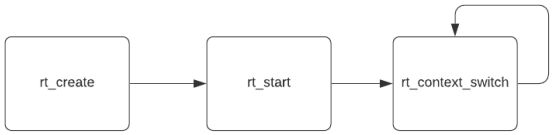

Figura 1: Processos do sistema

1.1 Criação

No processo de criação iniciado pela função `rt_create` é alocado espaço para as lista, a lista das tarefas é inicializada. Após a execução dessa função o usuário pode adicionar tarefas e executar o processo de partida.

1.2 Partida

O processo de partida é executado pela função `rt_start`. Nesse processo a tarefa ociosa é criada e as funções de cada tarefa tem seu contexto salvo nos registradores. Ao final deste processo a função `rt_clock` é chamada e a interrupção por cronômetro é definida e ativada, e por fim é chamado um laço infinito.

1.3 Troca de contexto

Após a execução do laço infinito iniciam as interrupções por cronometro, função `timer1ctc_handler`, que chama as trocas de contexto, função `rt_context_switch`, fazendo uso do algoritmo *rate monotonic*. Após definir definir o estado futuro da tarefa em execução, seleciona uma nova tarefa se necessário, ativa novamente as interrupções e pula para a tarefa escolhida. O laço entre `timer1ctc_handler`, `rt_context_switch` e `rt_task` é repetido infinitamente.

1.4 Variáveis globais e estruturas de dados

O escalonador usa algumas variáveis e estruturas de dados para realizar o escalonador de tarefas em tempo real.

```
1 enum enum_state{READY,
2                 RUNNING,
3                 BLOCKED,
4                 DONE,
5                 SYS}
6                  rt_state;
```

Listing 1: Estado da tarefa

A seguir é descrito cada estado:

- `READY`: tarefas prontas para serem escalonadas;
- `RUNNING`: tarefa em execução;
- `BLOCKED`: tarefas bloqueadas pelo sistema, nesse caso nenhuma;
- `SYS`: tarefas do sistema, nesse caso somente a tarefa de espera(`rt_idle_function`).

Os estados de cada tarefa estão são uma das variáveis dentro da estrutura de dados da tarefa, descrita abaixo:

```
typedef struct{
    void (*_function)();
    int _id;
    char *_name;
    int _period;
    int _capacity;
    int _deadline;
    int state;
    int executed;
} rt_task;

rt_task *rt_running_task;
rt_task *rt_idle_task;
```

Listing 2: Estrutura de dados das tarefa

Abaixo é descrita cada variável da estrutura de dados acima:

- `void (*_function)()`: função da tarefa, definida pelo usuário;
- `int _id`: número único de identificação da função definido sequencialmente pelo sistema;
- `char *_name`: nome da da tarefa;
- `int _period`: período da tarefa;
- `int _capacity`: capacidade ou ou tempo de computação da tarefa;
- `int state`: estado da tarefa, definido acima;
- `int executed`: unidades de tempo executadas pela tarefa.

As tarefas criadas pelo usuário são adicionadas em uma estrutura de dados encadeada descrita a seguir:

```
struct list {
        void *elem;
        struct list *next;
};

struct list *rt_list_task;
```

Listing 3: Lista encadeada

A variável `int rt_time` é o contador global da unidade de tempo, ele é incrementado pela função `rt_context_switch`.

2 Instruções e casos de uso

O sistema é simples de usar, requerendo o *GCC* com suporte para arquitetura *RISCV 32 bits*. Primeiramente baixe o repositório (privado no momento) no endereço *https://github.com/tarsioonofrio/hf-risc*, e abra o arquivo *hf-risc/tp1/app/scheduler.c*. Abaixo é apresentada um pequeno exemplo:

```
void f(void){
    volatile char cushion[1000];        /* reserve some stack space */
    cushion[0] = @ ;                    /* don t optimize my cushion away */

    if (!setjmp(rt_jmp[rt_running_id]))
        longjmp(rt_jmp[0], 1);
```

```
7
8
9       while (1) {
10          /* thread body */
11          printf("task 0...%d\n", rt_running);
12      }
13  }
```

Listing 4: Função de exemplo

O usuário deve escrever uma função sem parâmetros de entrada e com retorno `void`. As linhas 1 e 2 reservam espaço na pilha para essa função e impede que o compilador ao otimizar o código remova o espaço reservado. As linhas 5 e 6 reservam salvam o contexto nos registradores e pulam de volta para a função `rt_start`. Por fim o usuário deve adicionar o laço infinito com alguma execução dentro.

```
1  int main(void){
2      rt_create();
3
4      rt_add_task(f, 20, 3, 0, "1", READY);
5      rt_add_task(f, 05, 2, 0, "2", READY);
6      rt_add_task(f, 10, 2, 0, "3", READY);
7
8      rt_start();
9      return 0;
10 }
```

Listing 5: Exemplo de uso

Na função `main` adicione `rt_create` antes de `rt_add_task` que fará a adição das tarefas no escalonador.

```
1  int rt_add_task(void (*function),
2                  int period,
3                  int capacity,
4                  int deadline,
5                  char *name,
6                  int state);
```

Listing 6: Função de adição das tarefas

Os parâmetros da função `rt_add_task` são praticamente iguais as variáveis descritas em 2, menos `_id` e `executed`. Após isso basta executar `rt_start`.

O arquivo *sys.h* tem a função de auxiliar no desenvolvimento facilitando a troca entre as plataformas *X64* e *riscv 32 bits*. Nele há algumas contantes que são importantes nos testes do sistema:

- `DELAY`: se haverá atraso ou não no sistema, `0` sem atraso e `1` com atraso;
- `DELAY_TIME`: define o tempo de atraso caso a variável haja atraso;
- `LOG`: o tipo de texto que será impresso na tela,`0` sem impressão, `1` número identificador da tarefa em execução, `2` nome da tarefa em execução, `4` quantidade de ciclos usados no escalonamento;
- `TIMER`: se interrupções por relógio estarão ativadas, `0` sem interrupções e `1` com interrupções.

Há ainda outras funções que podem ser usadas no sistema:

- `const int  rt_get_states()`: retorna os estados de todas as tarefas;
- `const int  rt_get_ids()`: retorna os identificadores numéricos de todas as tarefas;

- `int rt_del_task(int id)`: remove uma tarefa pelo seu identificador numérico;
- `int rt_task_count()`: retorna total de tarefas.

3 Avaliação do custo da implementação

O sistema ficou relativamente leve mesmo com poucas otimizações. O sistema tem apenas um módulo, os arquivos *rt*. O arquivo *rt.h* tem apenas 66 linhas e o arquivo *rt.c* tem 254. O arquivo *sys.h* tem 45 linhas e apenas 5 linhas são obrigatórias em produção. No total são apenas *8.4 kB*, além do arquivo *scheduler.c* onde esta a função *main.c*.

Para a avaliação dos custos por ciclo da implementação realizada no presente trabalho usamos a seguinte rotina:

```
1 int main(void){
2     rt_create();
3     for (int i = 1; i < T + 1; i++){
4         rt_add_task(f, i*5, 2, 0, "-", READY);
5     }
6     rt_start();
7         return 0;
8 }
```

Listing 7: Função *main* da rotina de avaliaçào

Onde `T` é o total de tarefas adicionadas, período é 5 multiplicado pelo número de amostras, a capacidade é sempre 2. Foram coletados 1024 amostras de cada execução.

No. de Tarefas	2	4	8	16	32
Mínimo	119	137	179	251	395
Máximo	317	500	762	834	978
Média	242	343	444	515	659
Desvio padrão	58	88	105	105	105

Tabela 1: Medidas estatísticas por ciclo do algoritmo de escalonamento

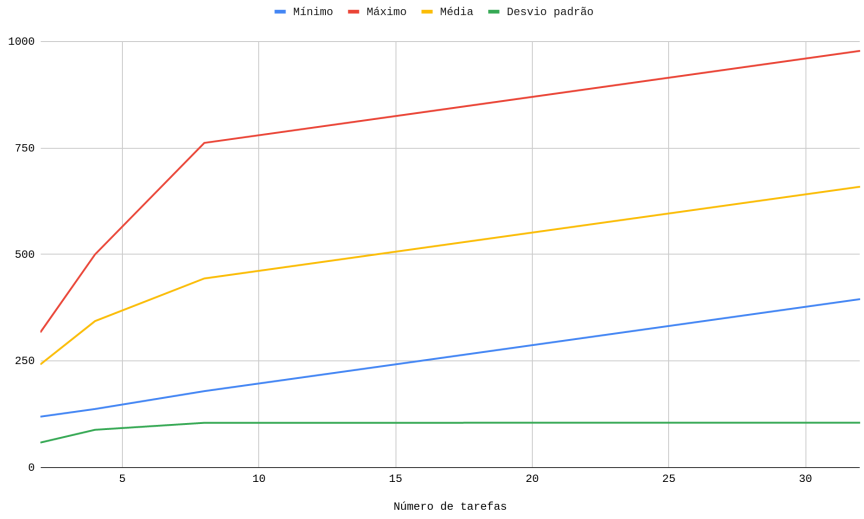

Figura 2: Medidas estatísticas por ciclo do algoritmo de escalonamento

A função `rt_context_switch` que faz o escalonamento tem custo baixíssimo com poucas tarefas porém cresce de forma constante a partir de 8 tarefas, conforme descrito na tabela e gráfico acima. Claramente a quantidade de tarefas aumenta o custo do escalonador.

Infelizmente não foi possível testar com 64 tarefas, o sistema apresenta erro e não executa, o erro está salvo no arquivo *data/log64.txt*.

4 Conclusão

O sistema apresenta bons resultados entretanto é possível fazer várias melhorias como definir a sequência de escalonamento antes da execução e não em tempo de execução, *rate monotonic* permitiria isso. tentar melhorar os aspectos de segurança, a estrutura de dado `rt_jmp` e a variável `rt_running_id` são expostas ao usuário permitindo respectivamente que este altere os contextos salvos e mude o número de identificação da tarefa em execução.