Sistema Operativo de Tempo Real para ATmega baseado em microkernel preemptivo

Rui Graça (201004124), Eduardo Almeida (201000641), Tiago Costa (200601289)

Relatório

***Resumo*—O produto do trabalho realizado é um sistema operativo de tempo real (RTOS) baseado num microkernel preemtivo, tendo como tecnologia alvo o microcontrolador ATmega328P.**

O sistema desenvolvido implementa stacks separadas para cada tarefa e escalonamento baseado em prioridades fixas, sendo que a API oferece funções como temporizadores, sinalização para sincronização entre tarefas e semáforos, baseados em SRP, assim como a possibilidade de lançar novas tarefas durante a execução do sistema.

I. Introdução

OS requisitos temporais a que estão sujeitos sistemas de tempo real levam à necessidade da utilização de sistemas operativos adequados, que permitam um comportamento totalmente determinístico, em que pode ser feita, de forma simples, uma análise do cumprimento dos requisitos temporais.

O trabalho aqui apresentado tem como produto final um sistema operativo que se enquadra neste contexto, possibilitando ao programador de sistemas de tempo real garantias de um comportamento determinístico e uma interface (API) com funcionalidades básicas para a implementação de um sistema de tempo real.

O sistema desenvolvido é um sistema com preempção, o que significa que existe uma interrupção periódica (tick) que verifica qual é a tarefa de maior prioridade pronta a executar e atribui-lhe o CPU. Cada tarefa representa uma thread de funcionamento do sistema, à qual é atribuída uma stack independente, que mantém o seu contexto entre as suas diversas ativações. Para isto, é necessário que, na troca de contexto, o stack pointer seja devidamente alterado.

O lançamento de novas tarefas pode ser feito tanto na inicialização do sistema como durante a execução, permitindo a uma tarefa lançar uma nova tarefa. Não existe, no entanto, nenhuma relação hierárquica entre tarefas criadoras e criadas, nem nenhuma relação especial entre elas.

Na API do sistema operativo são fornecidas funções que possibilitam a criação de temporizadores periódicos, que podem ser locais às tarefas que os criam ou globais, sendo que, neste caso, várias tarefas podem esperar por um mesmo temporizador. Após a criação de um temporizador, uma tarefa pode invocar uma função que espera que seja ativada por esse temporizador, colocando a tarefa num estado inativo e adicionando a tarefa à lista de espera do temporizador, que é esvaziada no tick do sistema, ativando todas as tarefas nela existentes.

A API possibilita, também, que uma tarefa fique inativa e seja acordada após um tempo determinado. Esta funcionalidade não é mais que a criação temporária de um temporizador.

Outra funcionalidade de extrema importância em sistemas com preempção são os semáforos, que controlam o acesso a regiões críticas, impedindo que a sequência de execução de tarefas tenha consequências imprevistas e indesejadas, as chamadas race conditions. Em sistemas de tempo real, dado que o escalonamento é baseado em prioridades, a utilização de semáforos como implementados noutros sistemas é problemática, dado que causa inversão de prioridade não limitada, que acontece quando uma tarefa de alta prioridade pode ficar

ilimitadamente bloqueada porque necessita que uma tarefa de mais baixa prioridade liberte um semáforo, sendo que esta tarefa de mais baixa prioridade não poderá executar enquanto houver tarefas de prioridade intermédia a ocupar o CPU. Surge, desta forma, a necessidade de mecanismos de controlo de race conditions que limitem ao mínimo a inversão de prioridade. Estes mecanismos passam, em geral, por uma herança temporária de prioridade, em que a tarefa que tem um semáforo (no qual está bloqueada uma tarefa de prioridade superior) fica temporáriamente com uma prioridade superior à original.

O mecanismo utilizado no sistema apresentado baseia-se em Stack Resource Policy (SRP), implementando semáforos binários (mutexes). De acordo com este protocolo, existe um teto do sistema (System Ceiling), que corresponde à prioridade mais alta entre todas as tarefas que utilizam os semáforos utilizados num dado momento. Por exemplo, se um semáforo é o único trancado num dado momento, e é utilizado por duas tarefas, T_1 com prioridade 1, e T_2 com prioridade 5, sempre que este semáforo está trancado o teto do sistema é 5, mesmo que seja T_1 a trancar o semáforo. Em cada tick, de acordo com SRP, só há uma mudança de contexto no caso de haver uma tarefa pronta a executar com prioridade superior à tarefa que tem o CPU nesse instante e com prioridade superior ao teto do sistema. Este mecanismo tem propriedades bastante boas, dado que, ao mesmo tempo que limita a inversão de prioridade, garante que uma tarefa, após começar a executar, nunca é bloqueada, o que reduz as mudanças de contexto.

Por último, a API oferece um mecanismo de sincronização, designado por sinal, que permite a uma tarefa ficar inativa à espera de ser sinalizada por outra tarefa. Da mesma forma que os temporizadores, um sinal implementa uma lista de espera, onde são colocadas todas as tarefas à espera do sinal. Esta lista é esvaziada quando alguma tarefa as sinaliza, ativando todas as tarefas na lista. Este mecanismo não evita inversão de prioridade e está sujeito a race conditions, como Lost Wakeup, que ocorre quando a sinalização é feita antes de uma tarefa declarar que está à espera do sinal, ficando a dormir infinitamente. Este problema pode levar a deadlocks, devendo, por isso, ser usado com cuidado, tendo sido implementado apenas para sincronização de tarefas event-triggered.

O trabalho tem como objetivo a aplicação de conceitos abordados em diversas unidades curriculares do MIEEC, onde foram introduzidos sem uma componente prática que possibilite compreender de que forma é que eles são aplicados num contexto de implementação real.

O código do trabalho, assim como o seu fluxo de desenvolvimento, está disponível em https://github.com/rpgraca/projsemb.

II. Estrutura do kernel

A. Lista de Tarefas

O kernel opera sobre uma lista de tarefas, onde são guardadas as tarefas existentes. Esta lista é uma estrutura (ListaTarefas_t) que tem como elementos um inteiro positivo de 8 bits (uint8_t) que indica o número de prioridades existentes, parâmetro que pode ser deixado ao critério do utilizador do sistema, e o apontador para um vetor em que cada elemento é um apontador para uma lista de tarefas de um dado nível de prioridade (estrutura TarefasPrioridade_t). Esta estrutura tem como campos o número de tarefas nesse nível de prioridade (uint8_t), um vetor de apontadores para tarefas (estrutura Tarefa_t) e o índice nesse vetor (uint8_t) da última tarefa que executou (este índice é utilizado pelo dispatcher, como explicado mais adiante).

A estrutura Tarefa_t tem como elementos a prioridade da tarefa (uint8_t), uma flag (uint8_t) que indica se a tarefa está pronta a executar ou não e o apontador para a stack da tarefa (char *).

A lista de tarefas é criada na inicialização do sistema, invocando a função Sched_inicia(), que cria uma lista de tarefas com um número de prioridades definido. Para adicionar uma tarefa à lista pode, em qualquer ponto de execução após a inicialização do sistema, ser invocada a função ListaTarefas_adicionaTarefa() (notar que inicialização é a alocação das estruturas de dados, e é uma fase distinta e anterior ao início do sistema, que ocorre com a primeira chamada do dispatcher).

Existe também uma função de terminação de tarefa, que é invocada sempre que uma tarefa faz return. Esta função reorganiza as estruturas de dados, eliminando a tarefa que terminou.

Tanto o escalonamento de tarefas como o dispatcher trabalham diretamente sobre uma única lista de tarefas global, alocada dinamicamente na inicialização do sistema.

B. Temporizadores e Sinais

O funcionamento dos temporizadores e o dos sinais são muito semelhantes entre si. Um temporizador é uma estrutura (Timer_t) que tem como campos o seu período em ticks (uint16_t), o seu tempo atual (uint16_t), o número de tarefas que tem em espera (uint8_t), o máximo de tarefas que podem esperar por ele (uint8_t) (parâmetro definido na criação de cada temporizador) e um vetor de apontadores para Tarefa_t, que contem uma lista das tarefas em espera. A estrutura do sinal (Sinal_t) tem os mesmos campos, à exceção do período e do tempo atual.

Ambas as estruturas são criadas por funções semelhantes (Timers_criaTimer() e Sinais_criaSinal()), que alocam dinamicamente um Timer_t ou um Sinal_t.

Para permitir o controlo dos temporizadores por parte do kernel, existe um vetor de temporizadores, ao qual é adicionado um temporizador sempre que é criado. Este vetor é uma estrutura VectorTimers_t, que tem como elementos um apontador para apontador de Timer_t, onde é alocado o vetor de Timer_t, e o número de elementos do vetor. Este vetor é alocado dinamicamente, alterando o seu tamanho consoante são adicionados ou removidos temporizadores.

Para ficar inativa até uma ativação por um temporizador *timer* ou por um sinal *sinal*, uma tarefa invoca Timers_esperaActivacao(&*timer*) ou Sinais_esperaSinal(&*sinal*), respetivamente. Para ficar inativa durante um determinado número de ticks *n_ticks*, uma tarefa pode invocar Timers_sleep(*n_ticks*). Para sinalizar o sinal *sinal*, uma tarefa pode invocar Sinal_sinaliza(&*sinal*).

Além destas, estão disponibilizadas ao programador funções para remover sinais e temporizadores, libertando a memória utilizada, e, no caso dos temporizadores, eliminando o apontador do temporizador do vetor de temporizadores.

C. Semáforos

Os semáforos implementados baseiam-se, como foi dito anteriormente, em SRP. No entanto, são uma versão simples deste protocolo, dado não possibilitam o controlo do acesso a recursos com mais que um elemento, funcionando, desta forma, como semáforos binários. Como consequência de se utilizar SRP, é necessário que o programador que utiliza o sistema operativo indique como parâmetro o teto do semáforo, que, em geral, é a prioridade da tarefa de mais alta prioridade que utiliza o semáforo.

Um semáforo é uma estrutura Semaforo_t que tem como elementos o teto do semáforo (uint8_t) e o estado do semáforo (uint8_t), que, neste caso, é uma flag binária. Um semáforo é criado com a função Semaforo_init(), que aloca dinamicamente um semáforo. Existe também uma função que permite apagar um semáforo (Semaforo_apaga()), libertando a memória alocada.

Para controlo da execução das tarefas, existe uma stack com a tarefa que trancou pela ultima vez um semáforo e com o teto do sistema. Sempre que é criado um semáforo, o espaço atribuído a esta stack é realocado para que ela suporte o caso limite, em que todos os semáforos estão trancados, sempre que é apagado um semáforo, o espaço em memória é reduzido.

Para adquirir um semáforo *sem*, uma tarefa invoca Semaforo_lock(&*sem*), para o libertar invoca Semaforo_unlock(&*sem*). Dado que, em SRP, sempre que um tarefa faz lock ao semáforo obtém garantidamente o lock, a função de lock limita-se a mudar o estado do semáforo e acrescentar a tarefa e o novo teto do sistema à stack. O novo teto do sistema é o máximo entre o teto do sistema anterior ao lock e o teto do semáforo que vai ser trancado. Quando se invoca unlock, a posição do topo da stack é removida. Como consequência deste modelo de funcionamento, se uma tarefa tem dois semáforos em lock, deve fazer unlock pela ordem inversa a que fez lock, caso contrário poderá comprometer o acesso exclusivo às regiões críticas.