Pedro Philippe Costa Rosanes

Extensão dos mecanismos de gerência de tarefas do sistema operacional TinyOS

Rio de Janeiro, Brasil

18 de Novembro de 2012

Pedro Philippe Costa Rosanes

Extensão dos mecanismos de gerência de tarefas do sistema operacional TinyOS

Monografia apresentada para obtenção do Grau de Bacharel em Ciência da Computação pela Universidade Federal do Rio de Janeiro.

Orientador:
Silvana Rossetto

Curso de Bacharelado em Ciência da Computação
Universidade Federal do Rio de Janeiro

Rio de Janeiro, Brasil

18 de Novembro de 2012

Extensão dos mecanismos de gerência de tarefas do sistema operacional TinyOS

Pedro Philippe Costa Rosanes

Trabalho de Conclusão de Curso submetido ao Departamento de Ciência da Computação do Instituto de Matemática da Universidade Federal do Rio de Janeiro como parte dos requisitos necessários para a obtenção do grau de Bacharel em Ciência da Computação.

Autor do Trabalho:

Pedro Philippe Costa Rosanes

Aprovado por:

Prof. Silvana Rossetto
Universidade Federal do Rio de Janeiro
Orientador

Prof. Flavia Delicato
Universidade Federal do Rio de Janeiro

Prof. Valeria Bastos
Universidade Federal do Rio de Janeiro

Resumo

Extensão dos mecanismos de gerência de tarefas do sistema operacional TinyOS

Redes de Sensores Sem Fio (RSSFs) são formadas por pequenos dispositivos de sensoreamento, com espaço de memória e capacidade de processamento limitados, fonte de energia esgotável e comunicação sem fio. Um dos sistemas operacionais mais usados na programação desses dispositivos é o TinyOS, um sistema leve, projetado especialmente para consumir pouca energia, um dos requisitos mais importantes para RSSFs. No modelo de programação adotado pelo TinyOS as operações básicas (ex.: leitura do sensor, envio de mensagem) são divididas em duas fases: uma para envio do comando, e outra para o tratamento da resposta (evento sinalizado via interrupção). Esse modelo de programação, baseado em eventos é adequado para o contexto de RSSF, mas quebra o fluxo de execução normal, dificultando a tarefa dos desenvolvedores de aplicações. Para que os tratadores de eventos (interrupções) sejam curtos, tarefas maiores são postergadas para execução futura e, para evitar concorrência entre elas, as tarefas são executadas em sequência, uma após a outra (i.e., uma tarefa só é iniciada após a tarefa anterior ser concluída). O objetivo deste trabalho é propor e implementar políticas alternativas de escalonamento de tarefas para o TinyOS, visando a construção de abstrações de programação de nível mais alto que facilitem o desenvolvimento de aplicações nessa área.

Abstract

Extensions of TinyOS's Scheduling Mechanisms

A wireless sensor network consists of small monitoring devices with limited amout of memory space and processing capability. It works with limited amout of energy source and comunicates wirelessly. TinyOS is the most used operating system for programming those devices, especially made for low energy consuming. The programming model offered by TinyOS gives priority to the handling of interruptions. For that reason TinyOS's services are split-phase operations. One phase for the service invocation and the other phase for the response via an event. This programming model makes the task of programming applications harder, since there is no main flow of execution. TinyOS also offers a system of postponed tasks, that are executed in sequence, therefore avoiding concurrency. This work proposes and implements different task scheduling policy and a cooperative programming model that will help the application development.

Lista de Figuras

Lista de Tabelas

Lista de Códigos

Conteúdo

1 Introdução

1.1 Motivação

Redes de Sensores Sem Fio (RSSFs) caracterizam-se pela formação de aglomerados de pequenos dispositivos que atuando em conjunto permitem monitorar ambientes físicos ou processos de produção com elevado grau de precisão. O desenvolvimento de aplicações que permitam explorar o uso dessas redes requer o estudo e a experimentação de protocolos, algoritmos e modelos de programação que se adequem às suas características e exigências particulares, entre elas, uso de recursos limitados, adaptação dinâmica das aplicações, e a necessidade de integração com outras redes, como a Internet.

Sistemas projetados para os dispositivos que formam as RSSFs devem lidar apropriadamente com as restrições e características particulares desses ambientes. A arquitetura adotada pelo TinyOS[2] — um dos sistemas operacionais mais usados na pesquisa nessa área — prioriza fortemente o tratamento dessas restrições em detrimento da simplicidade oferecida para o desenvolvimento de aplicações. A linguagem de programação usada é nesC [3], uma extensão de C que provê um modelo de programação baseado em componentes e orientado a eventos. Para lidar com as diversas operações de entrada e saída, o TinyOS utiliza um modelo de execução em duas fases, evitando bloqueios e, consequentemente, armazenamento de estados. A primeira fase da operação é um comando que pede ao hardware a execução de um serviço (ex.: sensoreamento). Esse comando retorna imediatamente dando continuidade à execução. Quando o serviço é concluido, o hardware envia uma interrupção, sinalizada como um evento pelo TinyOS. O tratador do evento recebe as informações (ex.: valor sensoreado) e trata/processa essas informações conforme programado[1]. O problema gerado por essa abordagem é a falta da visão de um fluxo contínuo de execução na perspectiva do programador.

O modelo de concorrência do TinyOS divide o código em dois tipos: assíncrono e síncrono. Um código assíncrono pode ser alcançável a partir de pelo menos um tratador de interrupção. Em função disso, a execução desses trechos do programa pode ser

interrompida a qualquer momento e é necessário tratar possíveis condições de corrida. Um código síncrono é alcançável somente a partir de tarefas (*tasks*) que são procedimentos adiados (postergados). Uma tarefa executa até terminar (não existe concorrência entre tarefas), por isso as condições de corrida, nesse caso, são evitadas. As tarefas são escalonadas para execução por um componente do TinyOS que usa uma política padrão de escalonamento do tipo *First-in First-out* [1].

Com a finalidade de oferecer maior flexibilidade aos desenvolvedores de aplicações, a versão 2.X do TinyOS, lançada em novembro de 2006, trouxe novas facilidades. Uma delas foi a possibilidade de substituir o componente de escalonamento de tarefas para implementar diferentes políticas de escalonamento [4]. A outra foi a possibilidade de usar o modelo de programação multithreading, mais conhecido pelos desenvolvedores de aplicações e que pode ser usado como alternativa para lidar com as dificuldades da programação orientada a eventos.

1.2 Objetivos

Neste trabalho avaliamos essas novas facilidades do TinyOS e propomos extensões que visam oferecer facilidade adicionais para os desenvolvedores de aplicações. Inicialmente, propusemos novos escalonadores de tarefas, implementando diferentes políticas de escalonamento por prioridade. Essas políticas ofereceram maior flexibilidade à programação, permitindo, por exemplo, que tarefas de encaminhamento de mensagens tenham menor prioridade sobre tarefas de sensoreamento.

Em seguida, tomando como base o modelo multithreading oferecido, projetamos um mecanismo de gerência cooperativa de tarefas para o TinyOS baseado no conceito de co-rotinas. [5] Visamos uma solução alternativa entre o modelo de escalonamento de tarefas que executam até terminar (escalonamento *First-in First-out*), e o modelo de execução alternada entre as tarefas (*multithreading*) que permite maior flexibilidade durante a execução, mas com custo de gerência alto.

O modelo de gerência cooperativa de tarefas é uma solução apropriada para as redes de sensores sem fio devido à simplicidade do hardware. Como os microcontroladores têm somente um núcleo, e não possuem tecnologia hyperthreading, não é possível existir duas unidades de execução executando em paralelo. A gerência cooperativa de tarefas permite manter contextos distintos de execução e alternar entre eles de acordo com as necessidades da aplicação, minimizando as trocas de contexto e eliminando a necessidade

de mecanismos de sincronização.

1.3 Organização do texto

O restante deste texto está organizado da seguinte forma. No capítulo 2 introduziremos conceitos básicos necessários para o entendimento deste trabalho. Apresentaremos a área de rede de sensores sem fio, e o sistema operacional TinyOS.

No capítulo 3 exploramos o funcionamento do sistema operacional TinyOS, foco deste trabalho. Apresentaremos o modelo de concorrência, o escalonador de tarefas e sequência de inicialização. Por último, apresentamos uma extensão ao modelo de concorrência nativo do TinyOS, a biblioteca *TOSThreads*.

No capítulo 4 abordaremos as teorias e implementações dos escalonadores de tarefas e do modelo de concorrência proposto neste trabalho.

No capítulo 5 apresentaremos os experimentos e os resultados utilizados para avaliar nossas propostas.

Finalmente, no capítulo 6, apresentaremos as conclusões deste trabalho e as propostas para trabalhos futuros.

2 Conceitos básicos

2.1 Introdução

Neste capítulo apresentaremos a área de rede de sensores sem fio e faremos uma breve descrição do *hardware* usado nesta área. Também serão expostos os conceitos básicos do sistema operacional TinyOS e da linguagem de programação *nesC*, destancando seus principais comandos.

2.2 Rede de Sensores Sem Fio

Uma rede de sensores sem fio (RSSF) é um conjunto de dispositivos formando uma rede de comunicação *ad-hoc*. Cada sensor tem a capacidade de monitorar diversas propriedades físicas, como intensidade luminosa, temperatura, aceleração, entre outras. Através de troca de mensagens, esses dispositivos podem agregar as informações coletadas para detectar um evento importante no local monitorado, como um incêndio. Essa conclusão pode ser encaminhada para um nó com maior capacidade computacional, conhecido como estação base. Esse nó pode decidir uma ação a ser tomada, ou enviar a informação pela Internet, como ilustrado na Figura 2.1.

Os sensores são normalmente utilizados para monitoriar ambientes de difícil acesso,

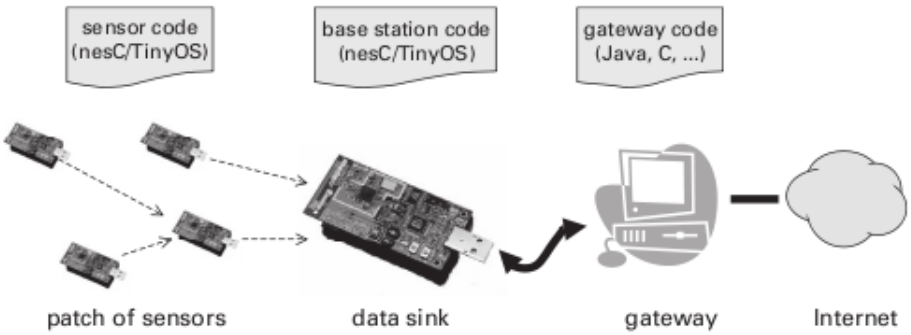


Figura 2.1: Exemplo de rede de sensores sem fio[1]

portanto, devem ser pequenos e utilizar comunicação sem fio para facilitar a instalação no ambiente e minimizar o custo financeiro. Para evitar manutenções frequentes, eles também devem consumir pouca energia. Devido a essas características, o hardware desses dispositivos possui recursos computacionais limitados. Ao invés de utilizar CPUs, são usados normalmente microcontroladores de 8 ou 16 bits com baixas frequências de relógio. Para armazenar o código da aplicação é utilizada uma pequena memória flash, da ordem de 100kB, e para a execução de código existe uma memória RAM da ordem de 10kB. Os circuitos de rádio também têm uma capacidade reduzida de transferência de dados, da ordem de kilobytes por segundos [1] [6]. Aliado ao hardware, o software também deve ser voltado para o baixo consumo de energia e de memória.

Alguns exemplos reais do uso de RSSF são: monitoramento da ponte Golden Gate em São Francisco, e dos vulcões Reventador e Tungurahua no Equador [1].

2.3 TinyOS e nesC

O TinyOS [7] é um dos sistema operacionais mais usados para auxiliar os programadores a desenvolverem aplicações de baixo consumo para rede de sensores sem fio. O modelo de programação provido é baseado em componentes e orientado a eventos. Os componentes são pedaços de código reutilizáveis, cujas dependências e serviços oferecidos são claramente definidos por meio de interfaces. Os componentes são conectados (*wiring*) entre si para formar uma aplicação TinyOS. A linguagem *nesc* [3], uma extensão de *C* é a responsável por implementar este modelo de programação.

O modelo de programação orientado a eventos permite que o TinyOS execute uma aplicação com somente uma linha de execução, respondendo a diferentes interrupções de sistema sem a necessidade de ações bloqueantes. Para isso todas as operações de entrada e saída são realizadas em duas fases. Na primeira fase, o comando de E/S sinaliza para o hardware o que deve ser feito, e retorna imediatamente, dando continuidade à execução. A conclusão da operação é sinalizada através de um evento, que será tratado pela segunda fase da operação de E/S.

Os modelos de programação baseada em compontenes e orientado a eventos estão diretamente relacionados. Um componente oferece uma interface, implementando os comandos e sinalizando os eventos relacionados, enquanto outro componente utiliza essa interface, através do uso destes comandos e da implementação dos tratadores de evento. A figura 2.2 ilustra essa conexão de interfaces entre componentes. O TinyOS permite

que diversos componentes utilizem a mesma instância de interface, permitindo, portanto, que tratadores distindos sejam executados através de somente uma sinalização de evento (*fan-out*).

O modelo de componentes também facilita a implementação da camada de abstração de hardware permitindo que cada plataforma tenha um conjunto diferente de componentes para lidar com as instruções de cada hardware. As abstrações providas são de serviços como sensoreamento, comunicação por rádio e armazenamento na memória flash.

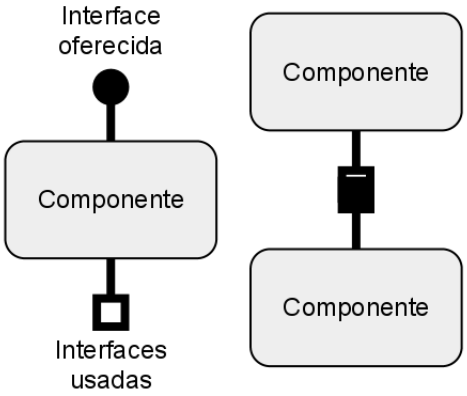


Figura 2.2: Ilustração de componentes e suas interfaces

2.3.1 Exemplo de aplicação TinyOS/nesC

Para apresentar os conceitos básicos da linguagem nesC e a estrutura geral de uma aplicação TinyOS/nesC usaremos a aplicação *Blink* (anexo 8.1). Esta aplicação faz os LEDs do dispositivo piscarem continuamente, criando um contador binário.

Como visto anteriormente, o modelo de programação TinyOS é baseado em componentes, portanto toda aplicação TinyOS/nesC deve definir uma configuração que indica quais componentes serão usados na sua composição. A configuração também é responsábel por definir como os componentes se conectam, especificando a ligação entre suas interfaces. Já a lógica de cada componente é implementada em um módulo separado cuja estrutura principal será apresentada a seguir.

Primeiramente define-se quais interfaces serão usadas pelo módulo, como pode ser visto no código 2.1. A aplicação utiliza as interfaces: *Boot*, responsável por sinalizar o fim da inicialização do sistema; *Leds* para controlar os LEDs da plataforma; e *Timer* para periodização.

Código 2.1: Interfaces usadas pelo módulo (BlinkC.nc)

```
1 module BlinkC @safe()
2 {
3   uses interface Timer<TMilli> as Timer0;
4   uses interface Timer<TMilli> as Timer1;
5   uses interface Timer<TMilli> as Timer2;
6   uses interface Leds;
7   uses interface Boot;
8 }
```

Uma vez definidas as interfaces, é preciso implementar seus respectivos tratadores de eventos. No código 2.2, pode-se ver a implementação do tratador do evento *Boot.booted()*, onde são feitas chamadas aos comandos das três interfaces de *Timer*.

Código 2.2: Eventos e comandos (BlinkC.nc)

```
11 implementation
12 {
13   event void Boot.booted()
14   {
15     call Timer0.startPeriodic( 250 );
16     call Timer1.startPeriodic( 500 );
17     call Timer2.startPeriodic( 1000 );
18   }
```

Ao final do tempo determinado pelos comandos *startPeriodic()*, os respectivos eventos *fired()* serão sinalizados. Cada tratador de evento fará então a chamada ao comando responsável por ligar ou desligar um dos LEDs.

Código 2.3: Eventos e comandos (BlinkC.nc)

```
20   event void Timer0.fired()
21   {
22     call Leds.led0Toggle();
23   }
24
25   event void Timer1.fired()
26   {
27     call Leds.led1Toggle();
28   }
29
30   event void Timer2.fired()
31   {
32     call Leds.led2Toggle();
33   }
34 }
```

Para utilizar as interfaces vistas no código 2.3, é preciso especificar quais componentes oferencem essas interfaces. Para isso define-se a configuração da aplicação, listada no código 2.4. *MainC* é compontente responsável pela inicialização de todo sistema operacional e pela sinalização do evento *Boot.booted()*, pelo qual é passado o controle para o componente principal da aplicação (*BlinkC*). Os componetes *LedsC* e *TimerMilliC* são os responsáveis pelo controle dos LEDs e pela temporização.

Código 2.4: Configuração (BlinkAppC.nc)

```
1  configuration BlinkAppC {}
2  implementation {
3    components MainC, BlinkC, LedsC;
4    components new TimerMilliC() as Timer0;
5    components new TimerMilliC() as Timer1;
6    components new TimerMilliC() as Timer2;
7
8    BlinkC.Boot -> MainC.Boot;
9    BlinkC.Timer0 -> Timer0;
10   BlinkC.Timer1 -> Timer1;
11   BlinkC.Timer2 -> Timer2;
12   BlinkC.Leds -> LedsC.Leds;
13 }
```

3 Concorrência e gerência de tarefas no TinyOS

3.1 Introdução

Neste capítulo apresentaremos o modelo de concorrência nativo do TinyOS, onde serão descritos os diferentes contextos de execução do sistema, e como os problemas de concorrência são tratados pelo TinyOS. Ilustraremos o uso de tarefas, mecanismo central do sistema para lidar com a concorrência, e explicaremos o funcionamento de seu escalonador. Apresentaremos a sequência de inicialização do TinyOS, permitindo ao leitor entender como os componentes do hardware, do sistema operacional e da aplicação se conectam. Por último apresentaremos a biblioteca *TOSThreads* que implementa uma extensão ao modelo de concorrência nativo do TinyOS.

3.2 Modelo de concorrência do TinyOS

O TinyOS define o conceito de *tasks* (tarefas) como mecanismo central para lidar com as questões de concorrência nas aplicações. Tarefas têm duas propriedades importantes. Elas não são preemptivas entre si, e são executadas de forma adiada. Isso significa que as tarefas executam até terminar e que ao submeter uma tarefa para execução (postar), o fluxo de execução continua, sem desvio, e ela só será processada mais tarde. As próprias tarefas, além dos comandos e tratadores de eventos podem postar novas tarefas, as quais são enfileiradas para execução posterior. Na definição básica do TinyOS, as tarefas não recebem parâmetros e não retornam resultados. No código 3.1, pode-se ver um exemplo de utilização de tarefas.

Código 3.1: Exemplo de utilização de tarefas

```
1 event void Timer.fired() {
2     post toogleLed();
3 }
```

```
4
5 task void toogleLed() {
6     call Leds.led0Toogle();
7 }
```

3.3 Sequência de inicialização do TinyOS

O principal componente do TinyOS, responsável por inicializar o sistema, é chamado *MainC*. Ele inicializa os componentes de hardware e software e o escalonador de tarefas. Para isso, *MainC* utiliza os componentes *RealmainP*, *PlataformC*, *TinySchedulerC*, utiliza a interface *SoftwareInit* e oferece a interface *Boot*. Os componentes internos de *MainC* se conectam entre si através das interfaces *PlataformInit* e *Scheduler*, como pode ser visto na figura 3.1.

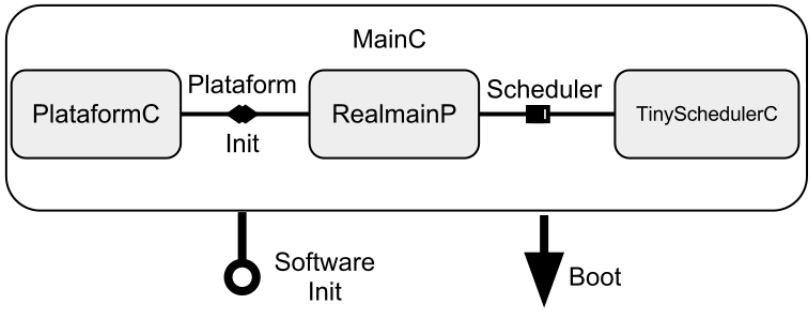


Figura 3.1: MainC

Primeiro é configurado o sistema de memória e escolhido o modo de processamento. Com esses pré-requisitos básicos estabelecidos, o escalonador de tarefas é inicializado para permitir que as próximas etapas possam postar tarefas. O segundo passo é inicializar os demais componentes de hardware, permitindo a operabilidade da plataforma. Alguns exemplos são configuração de pinos de entrada e saída, calibração do clock e dos LEDs. Como esta etapa exige códigos específicos para cada tipo de plataforma, o MainC se liga ao componente *PlataformC* que implementa o tratamento requerido por cada tipo de plataforma.

O terceiro passo trata da inicialização dos componentes de software. Além de configurar os aplicativos básicos do sistema, como os *timer*s, nesta etapa são executados também os procedimentos de inicialização dos componentes da aplicação. Para isso, os componentes da aplicação que precisam ser inicializados devem oferecer a interface *SoftwareInit*. Assim, durante a etapa de inicialização do sistema, os códigos de inicialização dos componentes da aplicação são automaticamente chamados.

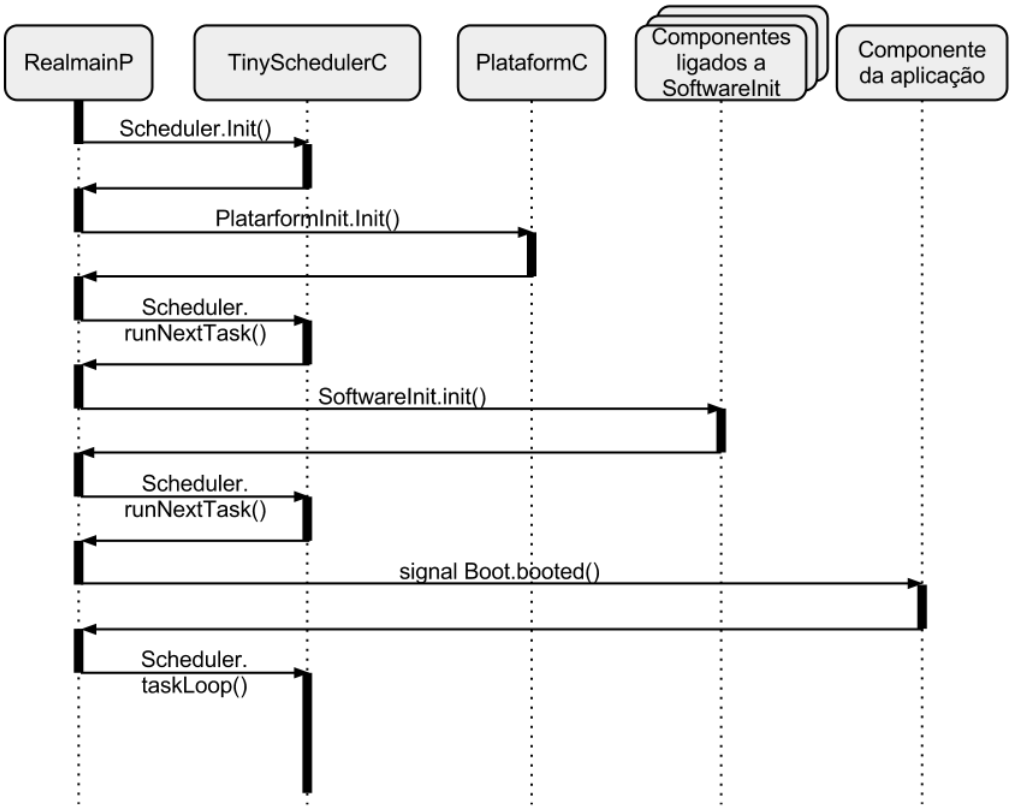


Figura 3.2: Sequência de Inicialização

Por último, quando todas as etapas foram concluídas, o MainC avisa a aplicação que a inicialização terminou, através do evento *Boot.booted()*. O TinyOS entra no seu laço principal, no qual o escalonador espera por tarefas e as executa. É importante notar que durante todo o processo de inicialização as interrupções do sistema ficam desabilitadas [8].

A figura 3.2 ilustra o diagrama de sequência desse processo e o anexo 8.2 contém a implementação da inicialização do sistema.

3.4 Código síncrono e assíncrono

O TinyOS minimiza os problemas clássicos de concorrência garantindo que qualquer possível condição de corrida gerada a partir de tratadores de interrupção, seja detectada em tempo de compilação. Para que isso seja possível, o código em nesC é dividido em dois tipos:

Código Assíncrono Código alcançável a partir de pelo menos um tratador de interrupção.

Código Síncrono Código alcançável somente a partir de tarefas.

Como visto na seção 3.3, ao final do processo de inicialização, as interrupções são ativadas, e o evento *Boot.booted* é sinalizado. O tratador deste evento, implementado no módulo principal da aplicação, é executado. Por último, o TinyOS entra em um laço infinito, onde as tarefas passam a ser atendidas. Este fluxo, quando não interrompido, é chamado de fluxo principal do TinyOS, e corresponde ao código síncrono. Como não há preempção entre as tarefas, variáveis compartilhadas entre elas são imunes a condições de corrida. Porém quando há uma interrupção de hardware, o tratador da interrupção assume o controle, e o fluxo principal é congelado até o término daquele. Qualquer variável compartilhada, quando acessada por estes códigos assíncronos, está sujeita a condições de corrida.

Como tarefas são postergadas, e atendidas pelo fluxo de execução principal, elas são usadas para fazer uma transição de contexto assíncrono para síncrono. Para fazer isto, um tratador de interrupção deve fazer somente o processamento mínimo, como transferência de dados entre o *buffer* e a memória. Após isto, deve postar uma tarefa para sinalizar o evento de conclusão da operação de E/S. Quando a tarefa for atentida, ela sinalizará o evento de forma síncrona. E portanto, seu tratador também será um código síncrono.

Para auxiliar no controle de condições de corrida, qualquer código assíncrono deve ser marcado como *async* no código fonte. Para evitar que um trecho de código seja interrompido, utiliza-se o comando *atomic* ou *power locks*.

O comando *atomic* garante exclusão mútua desabilitando interrupções. Dois fatos importantes surgem com o seu uso, primeiro a ativação e desativação de interrupções consome ciclos de CPU. Segundo, longos trechos atômicos podem atrasar outras interrupções, portanto é preciso tomar cuidado ao chamar outros componentes a partir desses blocos.

Algumas vezes é preciso usar um determinado hardware por um longo tempo, sem compartilhá-lo. Como a necessidade de atomicidade não está no processador e sim no hardware, pode-se conceder sua exclusividade a somente um usuário (componente) através de *Power locks*. Para isso, primeiro é feito um pedido através de um comando, depois quando o recurso desejado estiver disponível, um evento é sinalizado. Assim não há espera ocupada. Também existe a possibilidade de atendimento imediato. Nesse caso nenhum evento será sinalizado: se o recurso não estiver locado por outro usuário (componente), ele será imediatamente cedido, caso contrário, o comando retornará falso. [1].

3.5 Escalonador de tarefas do TinyOS

O componente responsável por gerenciar e escalonar tarefas no TinyOS é o componente *TinySchedulerC*. O escalonador padrão adota uma política *First-in First-out* para agendar as tarefas. Ele também cuida de parte do gerenciamento de energia, colocando a CPU em um estado de baixo consumo quando não há nada para ser executado.

O programador, ao codificar uma tarefa, utiliza duas construções:

```
1    post nome_da_tarefa();
2    task void nome_da_tarefa() { //Definicao da tarefa }
```

Essas duas construções são transformadas pelo compilador, fazendo com que a aplicação use uma interface chamada *TaskBasic*, exibida no código 3.2. A primeira construção é transformada em um comando, usado para indicar ao escalonador que esta tarefa deve entrar na fila. O escalonador por sua vez, quando decidir que esta tarefa será a próxima a executar, sinalizará o evento relacionado a este comando. A segunda construção é transformada no tratador deste evento, que implementa o que a tarefa deverá executar quando escalonada. É esta interface que permite a conexão das tarefas ao escalonador [1].

Código 3.2: Interface TaskBasic

```
1 interface TaskBasic {
2     async command error_t postTask();
3     event void runTask();
4 }
```

O escalonador, além de prover a interface *TaskBasic*, também deve prover a interface *Scheduler*, exibida no código 3.3.

Código 3.3: Interface Scheduler

```
1 interface Scheduler {
2     command void init();
3     command bool runNextTask();
4     command void taskLoop();
5 }
```

A implementação dessas interfaces se dá da seguinte forma:

command postTask() Decide onde a tarefa será inserida na fila.

event runTask() Indica que a tarefa deve executar.

command runNextTask() Retira a primeira tarefa da fila e sinaliza sua execução com o evento *runTask()*

command taskLoop() Laço infinito que executa o comando *runNextTask()*. Caso não haja tarefa para executar, coloca a CPU em modo de baixo consumo.

command init() Comando responsável pela inicialização do escalonador.

O componente *TinySchedulerC*, citado no inicio da seção, é o responsável por implementar as interfaces *Scheduler* e *TaskBasic*. Esse componente padrão do TinyOS implementa os comandos acima utilizando um algoritmo *First-in First-out*, e aceita somente tarefas do tipo *TaskBasic*.

3.5.1 Escalonadores personalizados

A partir da versão 2.1.x do TinyOS é possível mudar a política de gerenciamento de tarefas substituindo o componente escalonador padrão. Qualquer novo escalonador tem de prover a interface de tarefa padrão (*TaskBasic*) e a interface de escalonamento (*Scheduler*), além de garantir a execução de todas as tarefas (ausência de *starvation*) [4].

O componente *TinySchedulerC* é uma configuração que conecta as interfaces de tarefa à implementação do escalonador (chamado de *SchedulerP* no código 3.4). Para alterar o escalonador basta definir um novo componente *TinySchedulerC* e adicioná-lo ao diretório da aplicação. Neste novo componente, a interface *Scheduler* deve ser associada ao componente que implementa o escalonador proposto, como ilutrado abaixo.

Por último, deve-se amarrar a interface da tarefa com a interface do escalonador, como no código 3.4.

Código 3.4: Configuração TinySchedulerC

```
1  configuration TinySchedulerC {
2      provides interface Scheduler;
3      provides interface TaskBasic[uint8_t id];
4  }
5  implementation  {
6      components SchedulerP as Sched;
7      ...
8      Scheduler = Sched;
9      TaskBasic = Sched;
10 }
```

Um exemplo de aplicação que utiliza um escalonador novo pode ser visto no anexo 8.3.

3.5.2 Novos tipos de tarefas

Criando um novo escalonador também é possível criar novos tipos de tarefa. Para isso, basta definir uma nova interface com o comando *postTask* e o evento *runTask*.

O novo escalonador deve então prover esta interface e portanto implementar o comando *postTask*, e disparar o evento *runTask*.

Nos códigos 3.5 e 3.6 há um exemplo de um novo tipo de tarefa, e de um escalonador que a aceita. O escalador implementado ao receber uma tarefa *TaskDeadline*, também recebe por parametro o prazo de execução da mesma. E o componente *SchedulerDeadlineP* é o responsável por implementar o comando *postTask(deadline)*, e inserir a tarefa na fila de acordo com seu prazo.

Código 3.5: Interface da tarefa TaskDeadline

```
1 interface TaskDeadline<precision_tag> {
2     async command error_t postTask(uint32_t deadline);
3     event void runTask(); }
```

Código 3.6: Configuração TinySchedulerC provendo um novo tipo de tarefa

```
1  configuration TinySchedulerC {
2      provides interface Scheduler;
3      provides interface TaskBasic[uint8_t id];
4      provides interface TaskDeadline<TMilli>[uint8_t id];
5  }
6  implementation  {
7      components SchedulerDeadlineP as Sched;
8      ...
9      Scheduler = Sched;
10     TaskBasic = Sched;
11     TaskDeadline = Sched;
12 }
```

3.6 TinyOS Threads

TOSThreads é uma biblioteca que permite programação com threads no TinyOS sem violar ou limitar o modelo de concorrência do sistema. O TinyOS executa em uma

única thread — a thread do kernel — enquanto a aplicação pode executar em uma ou mais threads — nível de usuário. Em termos de escalonamento, o kernel tem prioridade máxima, ou seja, a aplicação só executa quando o núcleo do sistema está ocioso. Ele é responsável pelo escalonamento de todas as tarefas e execução das chamadas de sistemas. O escalonador de threads utiliza uma política *Round-Robin* com um tempo padrão de 5 milisegundos. Ele oferece toda a interface para manipulação de threads, como pausar, criar e destruir threads.

Três tipos de fluxo de execução passam a existir: tarefas, interrupções e threads. Como foi visto na seção 3.2, tarefas correspondem a um fluxo de execução, e tratadores de interrupção a outro. Além disso, foi observado que os tratadores de interrupção podem interromper a execução de uma tarefa, porém o contrário não é possível. Com esta observação, pode-se dizer que tratadores de interrupção têm prioridade maior do que tarefas. Para não violar o modelo de concorrência do TinyOS, as threads foram introduzidas com a menor prioridade de execução. Isto significa que uma interrupção de hardware suspende a execução da thread corrente, para permitir que seu tratador execute. Caso o tratador dessa interrupção poste uma tarefa, a thread corrente sairá de contexto, e a thread do *kernel* assumirá a CPU, permitindo que todas as tarefas sejam executadas. No final da execução de todas tarefas, a próxima thread é escalonada.

As trocas de contextos entre threads acontecem por dois motivos básicos: ocorrência de uma interrupção, término do tempo de execução da thread. Para implementar o primeiro caso, é inserida a função *postAmble* ao final de todas as rotinas de processamento de interrupção. Esta função verifica se foi postada uma nova tarefa, e caso positivo, o controle é passado para o *kernel*, como visto anteriormente. Caso contrário, a thread continua a executar logo após o término do tratador de interrupção. Para implementar o segundo caso, é utilizado um temporizador que provoca uma interrupção ao final de cada *timeslice*. O tratador da interrupção posta uma tarefa, forçando o *kernel* a assumir o controle e escalonar a próxima thread.

Para explorar o modelo *multithreading* dentro do ambiente TinyOS, chamadas de sistemas foram introduzidas para transformar operações de duas fases em operações síncronas (de uma fase), fornecendo assim uma visão de fluxo contínuo de execução para o programador. Como os serviços oferecidos pelo TinyOS são naturalmente *split-phase*, estas chamadas devem ser bloqueantes. Para fazer isto, a chamada de sistema bloqueia e adquire as informações da thread que a invocou, posta uma tarefa que executará o serviço *split-phase* original do TinyOS e acorda a thread do kernel. Posteriomente, esta tarefa

será executada pelo kernel, e a primeira fase do serviço será invocada. Na segunda fase, o resultado é enviado à thread, e esta é desbloqueada.

No código 3.7 expomos o uso de chamadas de sistema para transformar operações de duas fases em código síncrono. A aplicação da qual este código foi extraído utiliza dois nós para executar. O nó com identificador 1 é o responsável por enviar mensagens ao nó com identificador 0. Ao receber ou enviar uma mensagem, os nós alternam os estados dos LEDs. Na linha 5, pode-se ver o uso de uma chamada de sistema. Ao invocar este comando, a thread ficará suspensa até o recebimento da mensagem. Ao recebê-la, ela será repassada para a variável *m0*, e a thread será desbloqueada. O segundo parâmetro desta chamada define um tempo máximo de espera. Com isso é eliminada a necessidade de implementar o tratador do evento de recebimento de mensagem.

Código 3.7: Exemplo do uso de chamadas de sistema.

```
1  event void RadioStressThread0.run(void* arg) {
2      call BlockingAMControl.start();
3      for(;;) {
4          if(TOS_NODE_ID == 0) {
5              call BlockingReceive0.receive(&m0, 5000);
6              call Leds.led0Toggle();
7          }
8          else {
9              call BlockingAMSend0.send(!TOS_NODE_ID, &m0, 0);
10             call Leds.led0Toggle();
11             call RadioStressThread0.sleep(500);
12         }
13     }
14 }
```

Para gerenciar o uso concorrente de recursos entre threads a seguintes primitivas de sincronização são oferecidas:

Mutex Garante a sincronização por exclusão mútua, como será visto na seção 4.4.

Semáforo Garante sincronização por condição, como será visto na seção 4.4.

Barreira Garante que n threads tenham chegado em um mesmo ponto. Todas threads que chamarem *Barrier.block()* são bloqueadas até que n chamadas tenham acontecido.

Variável de condição Garante a suspensão de uma thread até que certa condição seja verdadeira.

Contador bloqueante Garante a suspensão de uma thread até que o contador atinja o

valor determinado.

O programador pode utilizar threads estáticas ou dinâmicas. A diferença está no momento de criação da pilha e do bloco de controle da thread. Nas threads estáticas a criação é feita em tempo de compilação, enquanto nas threads dinâmicas a criação é feita em tempo de execução. O bloco de controle, também chamado de *Thread Control Block* (TCB), contém informações sobre a thread, como seu identificador, seu estado de execução, o valor dos registradores (para troca de contexto), entre outras[9].

O apêndice 7.1 detalha a implementação de threads no TinyOS.

4 Abordagens teóricas e implementações das propostas

4.1 Introdução

Neste trabalho, foram propostas e implementadas extensões aos mecânismos de gerência de tarefas e ao modelo de programação do sistema TinyOS. Neste capítulo serão apresentadas as propostas e implementações feitas, precedidos de abordagens teóricas sobre os assuntos relacionados.

4.2 Abordagem teórica sobre escalonamento de tarefas

Neste capítulo iremos tratar do escalonamento de tarefas no contexto do sistema operacional, levando em consideração as características do TinyOS. Os algoritmos que abordaremos são não-preemptivos, devido a natureza das tarefas do TinyOS. O algoritmo mais simples de escalonamento, e também o padrão do TinyOS, é o *First-Come, First-Served*, onde as tarefas são atendidas segundo a ordem de chegada. O *overhead* gerado é mínimo, e não há possibilidade de *starvation*. Porém o tempo de resposta pode ser alto, se houver uma grande quantidade de tarefas na fila.

Escalonamento utilizando *deadline* é muito usado em sistemas operacionais de tempo real [10]. Neste algoritmo a próxima tarefa a ser executada é aquela com menor prazo (*deadline*). Esse algoritmo utiliza como parâmetro principal o prazo, determinado pelo desenvolvedor, para começar ou terminar a tarefa. Ou seja, o tempo limite recomendado para que a tarefa comece ou termine de executar. Para auxiliar o escalonamento, outros parâmetros, calculados pelo sistema operacional, podem ser utilizados. Alguns desses são: tempo de entrada na fila de prontos, tempo de processamento, recursos utilizados e prioridade. Caso não exista preempção, faz mais sentido utilizarmos, no escalonamento,

o prazo para começar a tarefa. Caso exista preempção, o prazo para terminar a tarefa é utilizado [10]. Um *overhead* maior passa a existir, devido à necessidade de ordenação das tarefas na fila e à preempção, caso exista. Porém o tempo de resposta pode ser aproximadamente estipulado pela própria tarefa.

Em um escalonamento de prioridade fixa, cada tarefa indica, no momento de entrada para fila de prontos (tempo de execução), sua importância em relação às outras tarefas. Nestes algoritmos podemos ter preempção por parcela de tempo, na entrada de outras tarefas, ou não ter preempção. No primeiro tipo, pode existir um *overhead* desnecessário quando o *time-slice* da tarefa atual terminou, porém não existe nenhuma outra com prioridade maior. O segundo tipo resolve este problema: se existe uma ordem de tarefas na fila, esta ordem só pode ser alterada caso uma nova tarefa entre. Quando não há preempção, a troca de tarefa só ocorre no término da execução de uma tarefa. Neste escalonamento também há um *overhead* maior, devido a ordenação das tarefas na fila. A possibilidade de *starvation* passa a existir, e o tempo de resposta varia de acordo com a prioridade das tarefas.

O escalonamento de multi-nível é um caso especial do escalonamento de prioridade fixa. Cada tarefa determina seu nível de prioridade em tempo de compilação. Onde cada nível de prioridade tem uma fila, com política *First-in First-out*, e as filas mais importantes devem ser atendidas por completo para que outras sejam atendidas.

O escalonamento de prioridade dinâmica visa eliminar a possibilidade de *starvation*. Neste caso, a tarefa ainda indica sua importância no momento de entrada para a fila de prontos. Porém, as tarefas que estão esperando para executar aumentam de prioridade toda vez que não são atendidas. Apesar disto aumentar significativamente o *overhead*, o *starvation* é eliminado.

4.3 Escalonadores propostos

Nesta seção apresentamos o escalonador padrão de tarefas do TinyOS. Também descrevemos o projeto e as etapas de implementação de novos escalonadores de tarefas para o TinyOS. Implementamos três propostas: escalonador EDF (*Earliest Deadline First*), escalonador com prioridades, e escalonador multi-nível.

4.3.1 Escalonador EDF (*Earliest Deadline First*)

Este escalonador (anexo 8.4)[1] aceita tarefas com *deadline* e elege aquela com menor *deadline* para executar, e utiliza a interfaze *TaskDeadline* para criar esse tipo de tarefa. O *deadline* utilizado é o prazo para começar a execução da tarefa. que é passado por parâmetro pela função postTask. As tarefas básicas, sem indicação de *deadline*, também são aceitas, como recomendado pelo TEP 106 [4], sendo executadas quando não há nenhuma tarefa com *deadline* para ser executada.

Porém, com isso, o escalonador não segue outra recomendação: não elimina a possibilidade de *starvation* pois as tarefas básicas só são atendidas quando não há nenhuma tarefa com *deadline* esperando para executar. A fila é implementada da mesma forma que a do escalonador padrão, a única mudança está na inserção. Para inserir, a fila é percorrida do começo até o fim, procurando-se o local exato de inserção. Portanto, o custo de inserir é $\bigcirc(n)$, e o custo de retirar da fila é $\bigcirc(1)$.

4.3.2 Escalonador por prioridades

Desenvolvemos um escalonador onde é possível estabelecer prioridades para as tarefas. A prioridade é passada como parâmetro através do comando *postTask*. Quanto menor o número passado, maior a preferência da tarefa, sendo 0 a mais prioritária e 254 a menos prioritária. As *Tasks* básicas também são aceitas, e são consideradas as tarefas de menor prioridade.

Foram encontrados dois problemas de *starvation*. O primeiro relacionado com as tarefas básicas, onde elas só seriam atendidas caso não houvesse nenhuma tarefa de prioridade na fila. Para resolver isso, foi definido um limite máximo de tarefas prioritárias que podem ser atendidas em sequência. Caso esse limite seja excedido, uma tarefa básica é atendida. O segundo é relacionado às próprias tarefas de prioridade. Se entrar constantemente *tasks* de alta prioridade, é possível que as tarefas de baixa prioridade não sejam atendidas. A solução se deu através do envelhecimento de tarefas. Ou seja, *tasks* que ficam muito tempo na fila, têm sua importância aumentada.

Dois tipos de estrutura de dados foram usadas para a organização das tarefas, uma fila comum e uma *heap*. Com isso, totalizou-se quatro diferentes versões do escalonador:

1. Fila comum sem envelhecimento (anexo 8.5)

[1] O TEP 106 [4] disponibiliza um protótipo

2. Fila comum com envelhecimento (anexo 8.6)

3. Heap sem envelhecimento (anexo 8.7)

4. Heap com envelhecimento (anexo 8.8)

Para implementar o envelhecimento, toda vez que uma tarefa é atendida, o escalonador percorre toda a fila incrementando a priodade de cada tarefa.

A complexidade das operações de inserção e remoção para cada escalonador é mostrada na tabela a seguir:

Escalonador	Inserção	Remoção
Fila, sem envelhecimento	$\bigcirc(n)$	$\bigcirc(1)$
Heap, sem envelhecimento	$\bigcirc(\log(n))$	$\bigcirc(\log(n))$
Fila, com envelhecimento	$\bigcirc(n)$	$\bigcirc(n)$
Heap, com envelhecimento	$\bigcirc(\log(n))$	$\bigcirc(n)$

4.3.3 Escalonador multi-nível

No TinyOS, percebe-se uma divisão clara dos tipos de serviços:

Rádio Comunicação sem fio entre diferentes nós da rede através de ondas de rádio.

Sensor Sensoriamento de diferentes características do ambiente.

Serial Comunicação por fio entre um nó e uma estação base (PC).

Básica Outros serviços, como por exemplo temporizador.

Por isso, desenvolvemos um escalonador que divide as tarefas de acordo com os tipos definidos acima. Onde cada tipo de tarefa utiliza uma interface distinta. Cada tipo de tarefa tem sua própria fila com política *First-in First-out*, e as filas mais importantes devem ser atendidas por completo para que as outras sejam antendidas. A prioridade padrão segue a disposição da lista acima, ou seja, tarefas de rádio tem maior prioridade sobre tarefas de sensor, que por sua vez tem maior prioridade sobre tarefas de comunicação serial. As tarefas básicas são atendidas por último. Porém, esta ordem pode ser facilmente modificada no código do escalonador. Uma aplicação de exemplo pode ser vista no anexo 8.9.

4.4 Abordagem teórica sobre multithreading e co-rotinas

Multithreading refere-se a capacidade do sistema operacional e/ou do hardware de suportar diversas linhas de execução simultâneas em uma mesma aplicação, chamadas de *threads*. Cada *thread* contém um contexto que inclui instruções, variáveis, uma pilha de execução, e um bloco de controle. O suporte de diversas unidades de execução se dá por meio de paralelismo real ou aparente. O primeiro tipo ocorre quando diferentes *threads* executam em diferentes processadores, núcleos, ou em processadores superescalares com vários bancos de registradores. O segundo tipo ocorre quando as *threads* intercalam o uso da CPU, por meio da gerência de um escalonador.

Em *multithreading*, o escalonador faz uso de um artifício chamado preempção. Isso significa que uma thread em execução pode ser interrompida, após qualquer instrução, para ceder a CPU a outra *thread*. Esta técnica permite que a CPU seja usada por todas as linhas de execução, sem intervenção do programador. Ou seja, a alternância de uso da CPU entre as *threads* ocorre de forma independente ao código implementado por elas.

Quando diferentes linhas de execução compartilham dados, o uso de preempção pode causar problemas de integridade destes dados. Este problema, conhecido como condição de corrida, ocorre quando uma thread é interrompida no meio da execução de uma sentença de linguagem de alto nível, podem gerar dados/respostas inconsistentes no programa. Uma forma de resolver condições de corrida, usada normalmente no nível do sistema operacional, é desabilitar a preempção temporariamente, garantindo a exclusão mútua de tais regões. Quando diversas threads estão trabalhando em conjunto, é preciso garantir um estado de sincronia entre eles, além de impedir acessos simultâneos a váriaveis compartilhadas. Para isso, utiliza-se primitivas de sincronização e de exclusão mútua. Porém essas primitavas que gerenciam o uso concorrente de recursos são custosas. [10]

Rotinas coopertativas, ou co-rotinas, têm as mesmas características das *threads*, quando classificadas como completas [5, s. 2.4]. Porém elas cooperam no uso da CPU através de transferência explícita de controle. Com isso elimina-se a necessidade de preempção, e consequentemente de gerêcia do uso concorrente de recursos.

Co-rotinas podem ser classificadas de acordo com o tipo de transferência de controle: simétricas e assimétricas. Co-rotinas do primeiro tipo têm a capacidade de ceder o controle para outra co-rotina explicitamente nomeada. As assimétricas só podem ceder o controle para a co-rotina que lhes ativou e possuem um comportamente semelhante ao

comportamento de funções. [5]

4.5 Implementação de co-rotinas para o TinyOS

O modelo de gerência cooperativa de tarefas é uma solução apropriada para as redes de sensores sem fio devido à simplicidade do hardware. Como os microcontroladores têm somente um núcleo, e não possuem tecnologia hyperthreading, não é possível existir duas unidades de execução executando em paralelo. A gerência cooperativa de tarefas permite manter contextos distintos de execução e alternar entre eles de acordo com as necessidades da aplicação, minimizando as trocas de contexto e eliminando a necessidade de mecanismos de sincronização.

Nossa implementação utilizou como base da implementação a extensão TOSThreads, vista na seção 3.6, por oferecer códigos responsáveis pela troca de contexto, pelo escalonamento de unidades de execução e de tarefas, e por chamadas de sistema. O modelo tomado como base foi o modelo síncrono, proposto por Ana Moura [5] em sua Tese. Nesse modelo a co-rotina cede o controle ao escalonador, esse então decide qual será a próxima co-rotina a executar.

Como visto na seção 4.4, a principal diferença entre threads e co-rotinas está na forma com que a preempção ocorre. Para modificar este comportamento foi necessário efetuar alterações no sistema, impedindo que uma unidade de execução perca o controle para outra involuntariamente.

Após estudar todo o funcionamento do escalonador de threads, verificamos que na implementação da extensão *TOSThreads* as threads sofrem preempção por duas causas distintas: término do *timeslice* ou interrupção de hardware. Portanto foi preciso modificar essa abordagem.

A primeira alteração efetuada foi retirar o limite de tempo de execução de cada thread, desabilitando o temporizador responsável por essa contagem. A segunda alteração foi criar um novo tipo de interrupção de hardware, que chamamos de interrupção curta. Originalmente, no *TOSThreads*, quando o tratador de interrupção solicita uma tarefa, o kernel assume o controle, executa a tarefa e escalona a próxima thread da fila. Na nossa implementação, após o kernel executar a tarefa, a unidade de execução que foi originalmente interrompida volta a executar. Para isso, foi criado um novo comando no escalonador: *brieflyInterruptCurrentThread()*

```
1 async command error_t ThreadScheduler.brieflyInterruptCurrentThread() {
```

```
2       atomic {
3           if(current_thread->state == TOSTHREAD_STATE_ACTIVE) {
4               briefly_interrupted_thread = current_thread;
5               briefly_interrupted_thread->state =
6                   TOSTHREAD_STATE_BRIEFLYINTERRUPTED;
7               interrupt(current_thread);
8               return SUCCESS;
9           }
10          return FAIL;
11      }
12  }
13
14  void scheduleNextThread() {
15      if(tos_thread->state == TOSTHREAD_STATE_READY)
16          current_thread = tos_thread;
17      else if (briefly_interrupted_thread != NULL)
18      {
19          current_thread = briefly_interrupted_thread;
20          briefly_interrupted_thread = NULL;
21      }
22      else
23          current_thread = call ThreadQueue.dequeue(&ready_queue);
24
25      current_thread->state = TOSTHREAD_STATE_ACTIVE;
26  }
```

Uma vez modificada a forma com que a preempção ocorre, o próximo passo foi modificar a interface da thread para permitir passagem de controle ao escalonador, e conseguentemente permitir que uma nova unidade de execução assuma o controle da CPU. Para isso, foi criado o comando *yield()*, que interrompe a co-rotina cedendo controle ao escalonador.

Código 4.1: Arquivo interfaces/Coroutine.nc

```
1  interface Coroutine {
2      ...
3      command error_t yield();
4      ...
5  }
6
7  //Arquivo: system/CoroutineP.nc
8  module StaticThreadP {
9      ...
```

```
10      command error_t Coroutine.yield[uint8_t id]() {
11          return call ThreadScheduler.interruptCurrentThread();
12      }
13      ...
14  }
```

5 Avaliação

5.1 Introdução

Neste capítulo apresentamos os experimentos feitos e os resultados obtidos, usados para avaliar as propostas deste trabalho.

5.2 Avaliação de desempenho dos escalonadores propostos

Experimentos com o escalonador de tarefas padrão Antes de começar a desenvolver outros escalonadores de tarefas, foi feito um experimento com o escalonador padrão que utiliza a política *First in, First Out*. Para medir a complexidade na prática, foi desenvolvida uma aplicação de teste (anexo 8.10). Nela cada tarefa executa um loop de 65000 iterações, fazendo uma simples multiplicação em cada iteração. O número de tarefas variou entre 20, 50 e 100. O tempo de execução foi medido em uma plataforma *MicaZ*, utilizando o temporizador *Counter<TMicro,uint32_t>*, utilizando uma precisão de microsegundos. A tabela 5.1 exibe os resultados obtidos com intervalos de confiança de 95%.

Experimentos com o escalonador de tarefas com prioridades Para avaliar o desempenho com o escalonador de tarefas com prioridades foi desenvolvida a mesma aplicação de teste (anexo 8.3), onde cada tarefa executa um loop de 65000 iterações, fazendo uma simples multiplicação em cada iteração. A prioridade de todas as tarefas, exceto uma, era igual, de forma que toda inserção deveria percorrer toda a fila. A tarefa

Escalonador	20 Tarefas	50 Tarefas	100 Tarefas
Escalonador Padrão	(1365,1367)	(1848,1850)	(2651,2653)

Tabela 5.1: Resultado dos experimentos com o escalonador padrão

Escalonador	20 Tarefas	50 Tarefas	100 Tarefas
Escalonador Padrão	(1365,1367)	(1848,1850)	(2651,2653)
Fila, sem envelhecimento	(1732,1734)	(4659,4661)	(13720,13722)
Heap, sem envelhecimento	(2602,2603)	(4306,4308)	(7485,7487)
Fila, com envelhecimento	(2277,2279)	(7886,7888)	(26065,26066)
Heap, com envelhecimento	(2664,2666)	(4509,4511)	(7886,7888)

Tabela 5.2: Resultado dos experimentos dos escalonadores

responsável por calcular o tempo de execução do experimento tinha a menor prioridade, para que esta fosse a última a executar. O número de tarefas variou entre 20, 50 e 100. O tempo de execução foi medido em uma plataforma *MicaZ*, utilizando o temporizador *Counter<TMicro,uint32_t>*, utilizando uma precisão de microsegundos. vezes. A tabela 5.2 exibe os resultados obtidos com intervalos de confiança de 95%.

Podemos perceber que, para um número pequeno de tarefas, a fila é mais eficiente que a heap, pois o *overhead* do algoritmo de inserção e remoção da heap não é compensado. Também pode-se notar que o uso de envelhecimento na estrutura de fila comum é menos eficiente quando comparada ao uso na estrutura de heap.

5.3 Comparação entre co-rotinas e threads

Com o objetivo de comparar o desempenho da implementação de co-rotinas com a biblioteca *TOSThread*, foram desenvolvidas duas aplicações para implementar o problema do produtor-consumidor. Uma utilizando *threads*, e outra utilizando co-rotinas. Foram utilizados duas linha de execução para o produtor, e uma para o consumidor. Para simular o tempo de processamento da produção e do consumo de uma unidade, foi implementado um laço de cem iterações, onde cada passo executa uma operação aritmética. Após consumir o número determinado de produtos, o tempo de execução é calculado.

5.3.1 Aplicação produtor consumidor (Threads)

Nesta seção apresentaremos a aplicação produtor/consumidor com o uso de threads da biblioteca *TOSThreads*.

Após a inicilização do sistema, as threads produtoras começam a gerar os produtos. A thread consumidora após consumir todos os produtos determinados, é responsável por calcular o tempo de execução de todo o programa e enviar este valor pela porta serial para um computador.

Abaixo temos o módulo e a configuração desta aplicação.

```
#include "mutex.h"
#include "semaphore.h"

#define BUF_SIZE 10
#define NUM_PROD 16000
#define NUM_SIM 100

module BenchmarkC {
  uses {
    interface Boot;
    interface Thread as Produtor1;
    interface Thread as Produtor2;
    interface Thread as Consumidor1;
    interface Leds;

    interface BlockingStdControl as AMControl;
    interface BlockingAMSend;
    interface Packet;

    interface Counter<TMicro, uint32_t> as Timer;

    interface Semaphore;
    interface Mutex;
  }
}

implementation {
    uint32_t t1;
    uint32_t * tempo;
    uint8_t buffer[BUF_SIZE+10];
    uint8_t p = 0;
    uint32_t numProdutosConsumidos = 0;
    uint32_t numProdutosProduzidos = 0;
    semaphore_t sem_produto, sem_buffer_cheio;
    mutex_t mutex_buffer;

    async event void Timer.overflow()
    {}

    event void Boot.booted() {
      uint8_t i;
      for (i = 0; i < BUF_SIZE; i++)
```

```
43            buffer[i] = 0;
44        t1 = call Timer.get();
45
46        call Semaphore.reset(&sem_produto, 0);
47        call Semaphore.reset(&sem_buffer_cheio, (uint8_t) BUF_SIZE);
48        call Mutex.init(&mutex_buffer);
49
50        call Produtor1.start(NULL);
51        call Produtor2.start(NULL);
52        call Consumidor1.start(NULL);
53    }
54
55
56    event void Produtor1.run(void* arg) {
57        uint16_t counter = 1;
58        uint16_t num_prods, j;
59
60
61        for (num_prods = 0; num_prods < NUM_PROD/2; num_prods++)
62        {
63            //Tempo simulado para criar um produto
64            for (j = 0; j < NUM_SIM; j++)
65                counter *= 3;
66
67            call Semaphore.acquire(&sem_buffer_cheio);
68
69            call Mutex.lock(&mutex_buffer);
70                buffer[p] = counter;
71                p++;
72                numProdutosProduzidos++;
73            call Mutex.unlock(&mutex_buffer);
74
75            call Semaphore.release(&sem_produto);
76        }
77        call Leds.led0On();
78    }
79
80    event void Produtor2.run(void* arg) {
81        uint16_t counter = 1;
82        uint16_t num_prods, j;
83
84        for (num_prods = 0; num_prods < NUM_PROD/2; num_prods++)
85        {
```

```
86              //Tempo simulado para criar um produto
87              for (j = 0; j < NUM_SIM; j++)
88                  counter *= 3;
89
90              call Semaphore.acquire(&sem_buffer_cheio);
91
92              call Mutex.lock(&mutex_buffer);
93                  buffer[p] = counter;
94                  p++;
95                  numProdutosProduzidos++;
96              call Mutex.unlock(&mutex_buffer);
97
98
99              call Semaphore.release(&sem_produto);
100         }
101
102         call Leds.led1On();
103     }
104
105     event void Consumidor1.run(void* arg) {
106         uint16_t num_prods, j;
107         uint16_t counter = 0;
108         message_t msg;
109
110         while(1)
111         {
112             if (sem_produto.v >= 9)
113                 call Leds.led0On();
114             call Semaphore.acquire(&sem_produto);
115
116             call Mutex.lock(&mutex_buffer);
117                 p--;
118                 counter = buffer[p];
119                 numProdutosConsumidos++;
120             call Mutex.unlock(&mutex_buffer);
121
122             call Semaphore.release(&sem_buffer_cheio);
123
124             //Tempo simulado para consumir produto
125             for (j = 0; j < NUM_SIM; j++)
126                 counter *= 3;
127
128             if (numProdutosConsumidos >= NUM_PROD - 1)
```

```
129             {
130                 t1 = call Timer.get() - t1;
131
132                 while( call AMControl.start() != SUCCESS );
133
134                 tempo = call Packet.getPayload(&msg, sizeof(uint32_t));
135                 (*tempo) = t1;
136
137                 while( call BlockingAMSend.send(AM_BROADCAST_ADDR,
138                                 &msg, sizeof(uint32_t)) != SUCCESS );
139             }
140         }
141     }
142
143 }
```

5.3.2 Aplicação produtor consumidor (Co-rotinas)

Nesta seção apresentaremos a aplicação produtor/consumidor com o uso de corotinas.

O funcionamento da aplicação é o mesmo visto na seção 5.3.1, mudamos somente o modelo de concorrência. Aqui podemos ver como é feito o uso do comando *yield()*, responsável por ceder o controle para o escalonador de co-rotinas. A co-rotina consumidora continua sendo a responsável por calcular o tempo de execução de todo o programa. É importante notar que no modelo de threads, todas as unidades de execução devem ser gerenciadas por meio de semáforos. No modelo de co-rotinas, não há essa necessidade, pois sabemos qual será a ordem de execução.

```
1  #define BUF_SIZE 10
2  #define NUM_PROD 16000
3  #define NUM_SIM 100
4
5  module BenchmarkC {
6    uses {
7      interface Boot;
8      interface Coroutine as Produtor1;
9      interface Coroutine as Produtor2;
10     interface Coroutine as Consumidor1;
11     interface Leds;
12
13     interface BlockingStdControl as AMControl;
14     interface BlockingAMSend;
```

```
15      interface Packet;
16
17      interface Counter<TMicro, uint32_t> as Timer;
18    }
19  }
20
21  implementation {
22      uint32_t t1;
23      uint32_t * tempo;
24      uint8_t buffer[BUF_SIZE+10];
25      uint8_t p = 0;
26      uint32_t numProdutosConsumidos = 0;
27      uint32_t numProdutosProduzidos = 0;
28
29      async event void Timer.overflow()
30      {}
31
32      event void Boot.booted() {
33        uint8_t i;
34        for (i = 0; i < BUF_SIZE; i++)
35            buffer[i] = 0;
36        t1 = call Timer.get();
37
38        call Produtor1.start(NULL);
39        call Produtor2.start(NULL);
40        call Consumidor1.start(NULL);
41      }
42
43      event void Produtor1.run(void* arg) {
44          uint16_t counter = 1;
45          uint16_t num_prods, j;
46
47          for (num_prods = 0; num_prods < NUM_PROD/2; num_prods++)
48          {
49              //Tempo simulado para criar um produto
50              for (j = 0; j < 100; j++)
51                  counter *= 3;
52
53              buffer[p] = counter;
54              p++;
55              numProdutosProduzidos++;
56
57              call Produtor1.yield();
```

```
58          }
59          call Leds.led0On();
60      }
61
62      event void Produtor2.run(void* arg) {
63          uint16_t counter = 1;
64          uint16_t num_prods, j;
65
66          for (num_prods = 0; num_prods < NUM_PROD/2; num_prods++)
67          {
68              //Tempo simulado para criar um produto
69              for (j = 0; j < 100; j++)
70                  counter *= 3;
71
72              buffer[p] = counter;
73              p++;
74              numProdutosProduzidos++;
75
76              call Produtor2.yield();
77          }
78          call Leds.led1On();
79      }
80
81      event void Consumidor1.run(void* arg) {
82          uint16_t num_prods, j;
83          uint8_t k = 0;
84          uint16_t counter = 0;
85          uint8_t ok = 0;
86          message_t msg;
87
88          while(1)
89          {
90              for (k = 0; k < 2; k++)
91              {
92                  if(p > 0)
93                  {
94                      p--;
95                      counter = buffer[p];
96                      numProdutosConsumidos++;
97                  }
98                  //Tempo simulado para consumir produto
99                  for (j = 0; j < 100; j++)
100                     counter *= 3;
```

```
101             }
102
103             if (numProdutosConsumidos >= NUM_PROD - 1 && !ok)
104             {
105                 call Leds.led2On();
106                 ok = 1;
107
108                 t1 = call Timer.get() - t1;
109
110                 while( call AMControl.start() != SUCCESS );
111
112                 tempo = call Packet.getPayload(&msg, sizeof(uint32_t));
113                 (*tempo) = t1;
114
115                 while( call BlockingAMSend.send(AM_BROADCAST_ADDR,
116                                 &msg, sizeof(uint32_t)) != SUCCESS );
117
118         call Produtor1.pause();
119         call Produtor2.pause();
120         call Consumidor1.pause();
121             }
122
123             call Consumidor1.yield();
124         }
125     }
126
127 }
```

5.3.3 Resultados

O tempo de execução foi medido em uma plataforma física (*MicaZ*), utilizando o temporizador *Counter<TMicro,uint32_t>*, utilizando uma precisão de microsegundos. O simulador não foi utilizado, pois a biblioteca *TOSTHREADS* não executa em simuladores.

Por meio do gráfigo da figura 5.1 podemos perceber que o modelo de gerência cooperativa foi mais eficiente que o modelo de threads.

Também foi constatado que a implementação do modelo de co-rotinas utilizou menos memória nesta aplicação, comparada a biblioteca *TOSThreads*, como pode ser visto na tabela 5.3. Isto ocorre pois a implementação de co-rotinas exclui todos os elementos de sincronização utilizados na biblioteca *TOSThreads*.

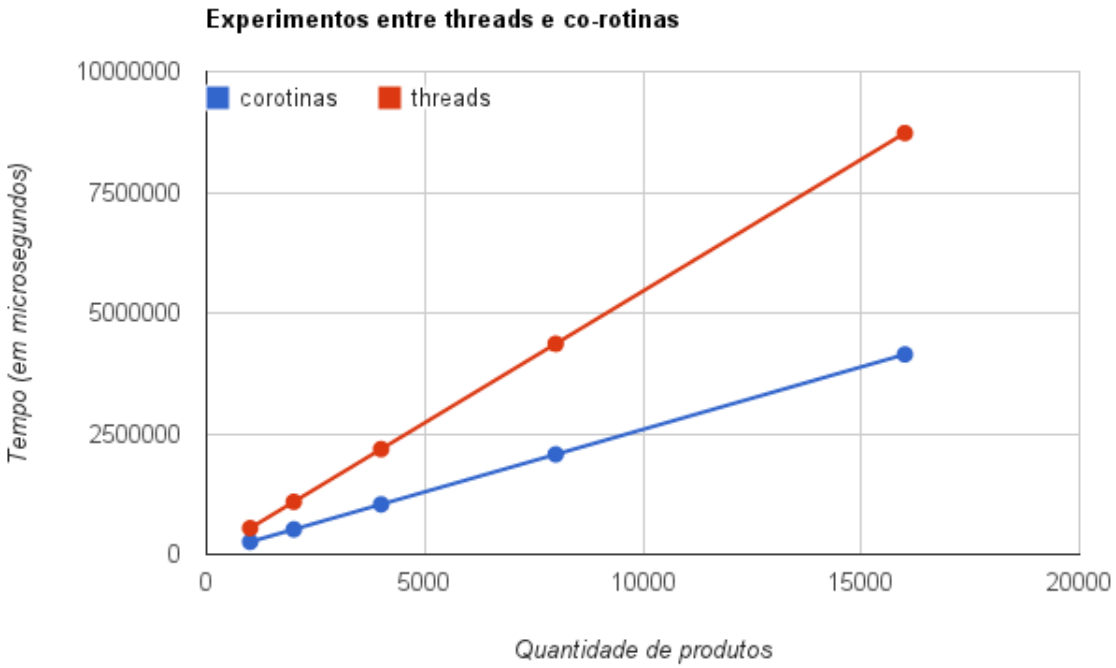


Figura 5.1: Gráfico dos experimentos de threads e co-rotinas

Modelo	ROM	RAM
Co-rotinas	10572 Bytes	3194 Bytes
Threads	11330 Bytes	3224 Bytes

Tabela 5.3: Uso de memória nas aplicações produtor/consumidor de acordo com o modelo de programação utilizado

6 Conclusões e trabalhos futuros

6.1 Conclusões

As redes de sensores sem fio podem ser aplicadas em diversas áreas, por exemplo, monitoramento de oscilações e movimentos de pontes, observação de vulcões ativos, previsão de incêndio em florestas, entre outras. Muitas dessas aplicações podem atingir alta complexidade, exigindo a construção de algoritmos robustos, como roteamento de pacotes diferenciado. Os escalonadores desenvolvidos neste trabalho poderão ajudar os desenvolvedores dessas aplicações complexas, oferecendo maior flexibilidade no projeto das soluções, como a possibilidade de priorizar certas atividades da aplicação (comunicação via rádio ou serial, sensoriamento, etc.). Através da análise dos experimentos realizados, cabe ao desenvolvedor decidir se a flexibilidade oferecida compensará o *overhead* gerado. Alguns dos pontos que devem ser levados em conta são: a quatidade de tarefas utilizadas, a necessidade ou não de eliminar *starvation*, e a complexidade da programação do algoritmo sem o uso dos escalonadores propostos.

Sem um fluxo contínuo de execução, sobre a perspectiva do programador, as aplicações complexas ficam difíceis de implementar e entender. O modelo de *threads* oferecido no TinyOS 2.1.X[9] facilita a solução deste problema. Entretanto, por ser um modelo preemptivo, o custo de gerência das threads pode implicar em queda de desempenho das aplicações. Com a implementação de um mecanismo de cooperação baseado em co-rotinas oferecemos uma alternativa ao programador, com um custo menor.

6.2 Trabalhos futuros

A principal proposta para trabalhos futuros consiste em experimentar o uso dos escalonadores propostos e do modelo de co-rotinas em diferentes tipos de aplicações, e avaliar tanto a facilidade de uso quanto a eficiência trazida pelas implementações.

Uma segunda proposta seria alterar a implementação do simulador para permitir a simulação de aplicações que utilizam o modelo de co-rotinas.

7 Apêndice

7.1 Implementação da biblioteca TOSThread

A seguir, descrevemos detalhes da implementação da biblioteca *TOSThread*. Mostraremos a organização dos diretórios e os códigos fonte mais importantes.

Organização dos diretórios: O diretório raiz do *TOSThread* é *tinyos-root-dir/tos/lib/tosthreads*. Abaixo descrevemos sua estrutura básica de subdiretórios e as respectivas descrições[1]:

chips: Código específico de hardware.
interfaces: Interfaces do sistema.
lib: Extensões e subsistemas.

- **net:** Protocolos de rede (protocolos *multihop*).
- **printf:** Componente que facilita a impressão de mensagens através da porta serial (para depuração).
- **serial:** Comunicação serial.

platforms: Código específico de plataformas.
sensorboards: Drivers para placas de sensoreamento.
system: Componentes do sistema.
types: Tipos de dado do sistema (arquivos header).

Sequência de Boot: Na inicialização do *TinyOS* com threads, primeiro há um encapsulamento da thread principal. Depois o curso original é tomado. A função *main()* está implementada em *system/RealMainImplP.nc*. A partir dela, o escalonador de threads é chamado através de um signal.

```
1 module RealMainImplP {
```

[1] Todos os arquivos serão referenciados a partir do diretório raiz *tinyos-root-dir/tos/lib/tosthreads/*. i.e. *types/thread.h*

```
2     provides interface Boot as ThreadSchedulerBoot;}
3 implementation {
4     int main() @C() @spontaneous() {
5         atomic signal ThreadSchedulerBoot.booted();}
6 }
```

O escalonador de threads, implementado em *TinyThreadSchedulerP.nc* encapsula a atual linha de execução como a thread do kernel. A partir de então, o curso normal de inicialização é executado.

```
1 event void ThreadSchedulerBoot.booted() {
2     num_runnable_threads = 0;
3     //Pega as informacoes da thread principal, seu ID.
4     tos_thread = call ThreadInfo.get[TOSTHREAD_TOS_THREAD_ID]();
5     tos_thread->id = TOSTHREAD_TOS_THREAD_ID;
6     //Insere a thread principal na fila de threads prontas.
7     call ThreadQueue.init(&ready_queue);
8
9     current_thread = tos_thread;
10    current_thread->state = TOSTHREAD_STATE_ACTIVE;
11    current_thread->init_block = NULL;
12    signal TinyOSBoot.booted();
13 }
```

Na fase final do *boot*, é feita a inicialização do hardware, do escalonador de tarefas, dos componentes específicos da plataforma, e de todos os componentes que se ligaram a *SoftwareInit*. É então sinalizado que o *boot* terminou, permitindo que o compontente do usuário execute. Por ultimo, o kernel passa o controle para o escalonador de tarefas.

```
1 void TinyOSBoot.booted() {
2     atomic {
3         //Inicializa hardware
4         platform_bootstrap();
5         call TaskScheduler.init();
6         call PlatformInit.init();
7         //Executa tarefas postas pela funcao a cima
8         while (call TaskScheduler.runNextTask());
9         call SoftwareInit.init();
10        //Executa tarefas postas pela funcao a cima
11        while (call TaskScheduler.runNextTask());
12    }
13    __nesc_enable_interrupt();
14    //Sinaliza boot para o usuario
```

```
15      signal Boot.booted();
16      call TaskScheduler.taskLoop();
17  }
```

No escalonador de tarefas, quando não houver mais *tasks* para executar, o controle é passado para o escalonador de threads.

```
1  command void TaskScheduler.taskLoop() {
2      for (;;) {
3          uint8_t nextTask;
4
5          atomic {
6              while((nextTask = popTask()) == NO_TASK) {
7                  call ThreadScheduler.suspendCurrentThread();
8              }
9          }
10         signal TaskBasic.runTask[nextTask]();
11     }
12 }
```

***types/thread.h*:** Este arquivo contém os tipos de dados e constantes excenciais para threads. A seguir estão listados esses dados, e seus respectivos códigos. Estados que uma thread pode assumir, como ativo, inativo, pronto e suspenso.

```
1  enum {
2      TOSTHREAD_STATE_INACTIVE = 0,  //This thread is inactive and
3                                     //cannot be run until started
4      TOSTHREAD_STATE_ACTIVE = 1,    //This thread is currently running
5                                     //on the cpu
6      TOSTHREAD_STATE_READY = 2,     //This thread is not currently running,
7                                     //but is not blocked and has work to do
8      TOSTHREAD_STATE_SUSPENDED = 3, //This thread has been suspended by a
9                                     //system call (i.e. blocked)
10 };
```

Constantes que controlam a quantidade máxima de threads, e o periodo de preempção. Estrutura da thread que contém dados como identificador, ponteiro para pilha, estado, ponteiro para função, registradores.

```
1  struct thread {
2  volatile struct thread* next_thread;
3      //Pointer to next thread for use in queues when blocked
4  thread_id_t id;
```

```
5     //id of this thread for use by the thread scheduler
6  init_block_t* init_block;
7     //Pointer to an initialization block from which this thread was spawned
8  stack_ptr_t stack_ptr;
9     //Pointer to this threads stack
10 volatile uint8_t state;
11    //Current state the thread is in
12 volatile uint8_t mutex_count;
13    //A reference count of the number of mutexes held by this thread
14 uint8_t joinedOnMe[(TOSTHREAD_MAX_NUM_THREADS - 1) / 8 + 1];
15    //Bitmask of threads waiting for me to finish
16 void (*start_ptr)(void*);
17    //Pointer to the start function of this thread
18 void* start_arg_ptr;
19    //Pointer to the argument passed as a parameter to the start
20    //function of this thread
21 syscall_t* syscall;
22    //Pointer to an instance of a system call
23 thread_regs_t regs;
24    //Contents of the GPRs stored when doing a context switch
25 };
```

Estrutura para controle de chamadas de sistema. Contém seu identificador, qual thread está executando, ponteiro para função que a implementa.

```
1  struct syscall {
2  struct syscall* next_call;
3     //Pointer to next system call for use in syscall queues when
4     //blocking on them
5  syscall_id_t id;
6     //client id of this system call for the particular syscall_queue
7     //within which it is being held
8  thread_t* thread;
9     //Pointer back to the thread with which this system call is associated
10 void (*syscall_ptr)(struct syscall*);
11    //Pointer to the the function that actually performs the system call
12 void* params;
13    //Pointer to a set of parameters passed to the system call once it is
14    //running in task context
15 };
```

***interfaces/Thread.nc*:** Contém os comandos de gerenciamento da thread e um evento para executá-la. Estes comandos permitem começar, terminar, pausar ou resumir a execução da thread.

```
interface Thread {
    command error_t start(void* arg);
    command error_t stop();
    command error_t pause();
    command error_t resume();
    command error_t sleep(uint32_t milli);
    event void run(void* arg);
    command error_t join();
}
```

***interfaces/ThreadInfo.nc*:** Contém comandos para receber ou apagar as informações da thread, vistas em 7.1.

```
interface ThreadInfo {
    async command error_t reset();
    async command thread_t* get();
}
```

***interfaces/ThreadScheduler.nc*:** Contém os comandos para gerenciar todas as threads. Essas funções servem para obter informações das threads, inicializá-las e trocar de contexto. Alguns comandos de *interfaces/Thread.nc* são simplesmente mapeados para os comandos abaixo.

```
interface ThreadScheduler {
    //Comandos para obter informacoes de uma thread
    async command uint8_t currentThreadId();
    async command thread_t* currentThreadInfo();
    async command thread_t* threadInfo(thread_id_t id);

    //Comandos para gerenciar a execucao de uma thread
    //Estes sao usados pelas proprias threads
    command error_t initThread(thread_id_t id);
    command error_t startThread(thread_id_t id);
    command error_t stopThread(thread_id_t id);

    //Comandos para gerenciar a execucao de uma thread
    //Estes sao usados por tratadores de interrupcao ou syscalls
    async command error_t suspendCurrentThread();
```

```
16      async command error_t interruptCurrentThread();
17      async command error_t wakeupThread(thread_id_t id);
18      async command error_t joinThread(thread_id_t id);
19  }
```

***system/ThreadInfoP.nc*:** Contém o vetor que representa a pilha, as informações da thread, como visto em 7.1 e a função que sinaliza a execução.

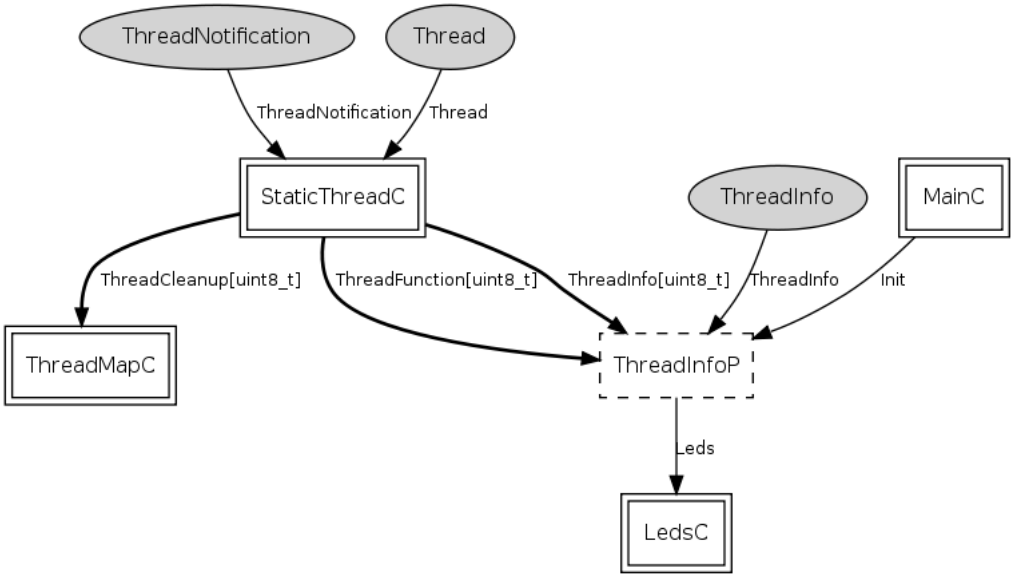
```
1  generic module ThreadInfoP(uint16_t stack_size, uint8_t thread_id) {
2  provides {
3      interface Init; // Para Inicializar as informacoes
4      interface ThreadInfo; // Para exportar as Informacoes da thread
5      interface ThreadFunction; // Sinaliza o evento responsavel
6                                //por executar a thread
7  }}
8
9  implementation {
10   uint8_t stack[stack_size];
11   thread_t thread_info;
12
13   void run_thread(void* arg) __attribute__((noinline)) {
14     signal ThreadFunction.signalThreadRun(arg);
15   }
16
17   error_t init() {
18     thread_info.next_thread = NULL;
19     thread_info.id = thread_id;
20     thread_info.init_block = NULL;
21     thread_info.stack_ptr = (stack_ptr_t)(STACK_TOP(stack, sizeof(stack)));
22     thread_info.state = TOSTHREAD_STATE_INACTIVE;
23     thread_info.mutex_count = 0;
24     thread_info.start_ptr = run_thread;
25     thread_info.start_arg_ptr = NULL;
26     thread_info.syscall = NULL;
27     return SUCCESS;
28   }
29
30   ...
31 }
```

***system/StaticThreadP.nc*:** Tem como principal objetivo servir de interface entre uma thread específica e o escalonador. Por exemplo, se StaticThreadC recebe um comando de pausa, este é repassado para o escalonador executar. Também termina de inicializar a thread e sinaliza o evento *Thread.run*.

```
1  module StaticThreadP.nc { ... }
2  implementation {
3
4  error_t init(uint8_t id, void* arg) {
5      error_t r1, r2;
6      thread_t* thread_info = call ThreadInfo.get[id]();
7      thread_info->start_arg_ptr = arg;
8      thread_info->mutex_count = 0;
9      thread_info->next_thread = NULL;
10     r1 = call ThreadInfo.reset[id]();
11     r2 = call ThreadScheduler.initThread(id);
12     return ecombine(r1, r2);
13 }
14
15 event void ThreadFunction.signalThreadRun[uint8_t id](void *arg) {
16     signal Thread.run[id](arg);
17 }
18
19 command error_t Thread.start[uint8_t id](void* arg) {
20     atomic {
21         if( init(id, arg) == SUCCESS ) {
22             error_t e = call ThreadScheduler.startThread(id);
23             if(e == SUCCESS)
24                 signal ThreadNotification.justCreated[id]();
25             return e;
26         }
27     }
28     return FAIL;
29
30     ... Continuacao da implementacao da interface thread ...
31     ... Todos os comandos sao simplesmente passados para o ...
32     ... equivalente no ThreadScheduler ...
33 }
```

***system/ThreadC.nc*:** Esta configuração é a "interface" da thread com o usuário e com o escalonador. Primeiramente, é ela que prove a interface *interfaces/Thread.nc*, portanto o programador deve codificar o tratador do evento *Thread.run* e amarrá-lo a

este componente. Em segundo lugar, conecta entre si todos os componentes importantes para o gerenciamento. Os principais são *system/MainC* para inicialização da thread no *boot* do sistema, *system/ThreadInfoP.nc* como visto em 7.1, e *system/StaticThreadC.nc* como visto em 7.1. A figura abaixo permite uma melhor visualização. As elipses são interfaces, os retângulos são componentes e as setas indicam qual interface liga os dois componentes.



***chips/atm128/chip_thread.h*:** Antes de expor as funções do escalonador de threads, é importante expor algumas macros de baixo nível que realizam a troca de contexto. Para guardar o contexto de hardware da thread, criaram a estrutura *thread_regs_t*. Existem também algumas macros para salvar e restaurar estes registradores.

Por último, são definidas duas macros para preparação da thread.

***system/TinyThreadSchedulerP.nc*:** Durante a inicialização do sistema muitas inicializações são feitas através da interface *Init* amarrada ao compontente *MainC*. Isso ocorre com a *system/StaticThreadP.nc*. Como visto acima, durante a execução desta função, o escalonador é chamado através do comando a seguir.

```
1 command error_t ThreadScheduler.initThread(uint8_t id) {
2     thread_t* t = (call ThreadInfo.get[id]());
3     t->state = TOSTHREAD_STATE_INACTIVE;
4     t->init_block = current_thread->init_block;
5     call BitArrayUtils.clrArray(t->joinedOnMe, sizeof(t->joinedOnMe));
6     PREPARE_THREAD(t, threadWrapper);
7         //O codigo abaixo e' definicao da macro PREPARE_THREAD,
```

```
8           //inserido aqui para facilitar o entendimento do codigo.
9           //uint16_t temp;                                          \
10          //SWAP_STACK_PTR(temp, (t)->stack_ptr);                   \
11          //__asm__("push %A0\n push %B0"::"r"(&(threadWrapper)));    \
12          //SWAP_STACK_PTR((t)->stack_ptr, temp);                   \
13          //SAVE_STATUS(t)
14      return SUCCESS;
15  }
```

É importante notar que na macro *PREPARE_THREAD()*, o endereço da função *threadWrapper* está sendo empilhado na pilha da thread. Esta função encapsula a chamada para a execução da thread.

```
1  void threadWrapper() __attribute__((naked, noinline)) {
2      thread_t* t;
3      atomic t = current_thread;
4
5      __nesc_enable_interrupt();
6      (*(t->start_ptr))(t->start_arg_ptr);
7
8      atomic {
9          stop(t);
10         sleepWhileIdle();
11         scheduleNextThread();
12         restoreThread();
13     }
14 }
```

No laço principal do escalonador de tarefas, quando não há mais nada para executar, a thread atual é suspensa. Com isso o controle é passado para o escalonador de threads através do comando *suspendCurrentThread()*. Na demostração de código abaixo, algumas chamadas a funções são substituídas pelo seus corpos, para facilitar o entendimento.

```
1  async command error_t ThreadScheduler.suspendCurrentThread() {
2      atomic {
3          if(current_thread->state == TOSTHREAD_STATE_ACTIVE) {
4              current_thread->state = TOSTHREAD_STATE_SUSPENDED;
5              //suspend(current_thread);
6              #ifdef TOSTHREADS_TIMER_OPTIMIZATION
7                  num_runnable_threads--;
8                  post alarmTask();
9              #endif
10             sleepWhileIdle();
```

```
11              //interrupt(current_thread);
12              yielding_thread = current_thread;
13              //scheduleNextThread();
14              if(tos_thread->state == TOSTHREAD_STATE_READY)
15                  current_thread = tos_thread;
16              else
17                  current_thread = call ThreadQueue.dequeue(&ready_queue);
18
19              current_thread->state = TOSTHREAD_STATE_ACTIVE;
20              //fim scheduleNextThread();
21
22              if(current_thread != yielding_thread) {
23                  //switchThreads();
24                  void switchThreads() __attribute__((noinline)) {
25                      SWITCH_CONTEXTS(yielding_thread, current_thread);
26                   }
27                  //fim switchThreads();
28              }
29              //fim interrupt(...)
30              //fim suspend(current_thread);
31              return SUCCESS;
32          }
33          return FAIL;
34      }
35  }
```

É muito importante notar que a função *switchThreads()* não é *inline*. Isso significa que os valores dos registradores serão empilhados. Haverá então uma troca de contexto e o registrador SP apontará para a pilha da nova thread. Por último, a função *switchThreads()* retornará para o endereço que está no topo da nova pilha. Este novo endereço, como visto acima, aponta para a função *threadWrapper()*. Esta por sua vez, através de uma função e duas sinalizações executa a thread.

Chamadas de sistema A seguir mostraremos detalhes da implementação de uma chamada de sistema. Para isso utilizaremos como exemplo a chamada *BlockingAMReceiver*, que bloqueia uma thread até o recebimento de uma mensagem, ou até o término de tempo de espera.

A chamada é feita utilizando o comando *call BlockingReceive.receive(&mensagemASerRecebida, timeout)*. A mensagem recebida será inserida no endereço de memória passado como

primeiro parâmetro, e o retorno indicará se houve sucesso ou não no recebimento da mesma.

Este comando primeiramente aloca espaço na pilha para os dados da chamada de sistema e para os parâmetros. Como esta chamada pode ser feita com diferentes identificadores de mensagens ativas, é preciso utilizar uma fila com as chamadas de sistema ativas. Depois, é verificado se existe tempo máximo de espera, ou não, para chamar o comando *SystemCall.start()*. Por último, quando a chamada é completada, ela é retirada da fila, e o comando retorna.

```
1  //Chamada bloqueante
2  command error_t BlockingReceive.receive[uint8_t am_id](message_t* m,
3                                                         uint32_t timeout) {
4      syscall_t s;
5      params_t p;
6
7      atomic {
8          if((blockForAny == TRUE) ||
9             (call SystemCallQueue.find(&am_queue, am_id) != NULL))
10             return EBUSY;
11         call SystemCallQueue.enqueue(&am_queue, &s);
12     }
13
14     p.msg = m;
15     p.timeout = timeout;
16     atomic {
17         p.error = EBUSY;
18         if(timeout != 0)
19             call SystemCall.start(&timerTask, &s, am_id, &p);
20         else
21             call SystemCall.start(SYSCALL_WAIT_ON_EVENT, &s, am_id, &p);
22     }
23
24     atomic {
25         call SystemCallQueue.remove(&am_queue, &s);
26         return p.error;
27     }
28 }
```

O comando *SystemCall.start* é o responsável por bloquear e armazenar as informações da thread que invocou a chamada. Dependendo do tipo de chamada de sistema, a thread de kernel pode acordar ou não. No caso de uma chamada que simplesmente espera por

um evento, como o recebimento de uma mensagem por rádio, o kernel não é acordado. Porém, se a chamada precisa executar um comando, como o envio de uma mensagem, este é executado, pelo kernel, através de uma tarefa. Portanto é preciso postar esta tarefa e acordar o kernel.

```
command error_t SystemCall.start(void* syscall_ptr,
                                 syscall_t* s,
                                 syscall_id_t id,
                                 void* p) {
    atomic {

        current_call = s;
        current_call->id = id;
        current_call->thread = call ThreadScheduler.currentThreadInfo();
        current_call->thread->syscall = s;
        current_call->params = p;

        if(syscall_ptr != SYSCALL_WAIT_ON_EVENT) {
            current_call->syscall_ptr = syscall_ptr;
            post threadTask();
            call ThreadScheduler.wakeupThread(TOSTHREAD_TOS_THREAD_ID);
        }

        return call ThreadScheduler.suspendCurrentThread();
    }
}
```

No exemplo da chamada *BlockingAMReceive.receive*, caso tenha sido determinado um *timeout*, o temporizador deste *timout* será inicializado através desta tarefa.

```
//Temporizador responsavel por calcular o timeout.
void timerTask(syscall_t* s) {
    params_t* p = s->params;
    call Timer.startOneShot[s->thread->id](*(p->timeout));
}

//Tarefa que chama a funcao da chamada de sistema.
task void threadTask() {
    (*(current_call->syscall_ptr))(current_call);
}
```

Após o kernel ter executado a primeira fase do serviço, é preciso esperar pela segunda fase. No exemplo sendo utilizado aqui, a segunda fase pode ser o evento correspondente

ao recebimento de uma mensagem (*event message_t* Receive.receive*), ou correspondente ao término do tempo de espera (*event void Timer.fired*). Nos dois casos, os tratadores dos eventos são responsáveis por copiar o resultado (mensagem recebida e/ou resposta de erro) para a variável passada como parâmetro para a chamada de sistema.

```
event message_t* Receive.receive[uint8_t am_id](message_t* m,
                                                 void* payload,
                                                 uint8_t len) {
    syscall_t* s;
    params_t* p;

    if(blockForAny == TRUE)
        s = call SystemCallQueue.find(&am_queue, INVALID_ID);
    else
        s = call SystemCallQueue.find(&am_queue, am_id);
    if(s == NULL) return m;

    p = s->params;
    if( (p->error == EBUSY) ) {
        call Timer.stop[s->thread->id]();
        *(p->msg) = *m;
        p->error = SUCCESS;
        call SystemCall.finish(s);
    }
    return m;
}

event void Timer.fired[uint8_t id]() {
    thread_t* t = call ThreadScheduler.threadInfo(id);
    params_t* p = t->syscall->params;
    if( (p->error == EBUSY) ) {
        p->error = FAIL;
        call SystemCall.finish(t->syscall);
    }
}
```

Como são invocadas muitas funções, reunimos na listagem abaixo todas as passagem do ponteiro *params_t* p*, para facilitar o entendimento.

```
call BlockingReceive.receive(&mensagemASerRecebida, timeout);

command error_t BlockingReceive.receive[uint8_t am_id](message_t* m,
                                                       uint32_t timeout) {
    syscall_t s;
```

```
6       params_t p;
7       //...
8       call SystemCallQueue.enqueue(&am_queue, &s);
9       //...
10      p.msg = m;
11      //...
12      call SystemCall.start(SYSCALL_WAIT_ON_EVENT, &s,am_id ,&p) ;
13  }
14
15  command error_t SystemCall.start(void* syscall_ptr,
16                                   syscall_t* s,
17                                   syscall_id_t id,
18                                   void* p) {
19      current_call = s;
20      //...
21      current_call->params = p;
22      //...
23  }
24
25  event message_t* Receive.receive[uint8_t am_id](message_t* m,
26                                                  void* payload,
27                                                  uint8_t len) {
28      syscall_t* s;
29      params_t* p;
30
31      //...
32      s = call SystemCallQueue.find(&am_queue, am_id);
33      //...
34      p = s->params;
35      //...
36      *(p->msg) = *m;
37  }
```

8 Anexos

8.1 Aplicação Blink

Código 8.1: Aplicação Blink (Configuração)

```
1  configuration BlinkAppC {}
2  implementation {
3    components MainC, BlinkC, LedsC;
4    components new TimerMilliC() as Timer0;
5    components new TimerMilliC() as Timer1;
6    components new TimerMilliC() as Timer2;
7
8    BlinkC.Boot -> MainC.Boot;
9    BlinkC.Timer0 -> Timer0;
10   BlinkC.Timer1 -> Timer1;
11   BlinkC.Timer2 -> Timer2;
12   BlinkC.Leds -> LedsC.Leds;
13 }
```

Código 8.2: Aplicação Blink (Módulo)

```
1  #include "Timer.h"
2
3  module BlinkC @safe()
4  {
5    uses interface Timer<TMilli> as Timer0;
6    uses interface Timer<TMilli> as Timer1;
7    uses interface Timer<TMilli> as Timer2;
8    uses interface Leds;
9    uses interface Boot;
10 }
11 implementation
12 {
13   event void Boot.booted()
14   {
15     call Timer0.startPeriodic( 250 );
```

```
16      call Timer1.startPeriodic( 500 );
17      call Timer2.startPeriodic( 1000 );
18    }
19
20    event void Timer0.fired()
21    {
22      call Leds.led0Toggle();
23    }
24
25    event void Timer1.fired()
26    {
27      call Leds.led1Toggle();
28    }
29
30    event void Timer2.fired()
31    {
32      call Leds.led2Toggle();
33    }
34  }
```

8.2 Código da inicialização do sistema

Código 8.3: RealMainP

```
1  module RealMainP {
2      provides interface Booted;
3      uses {
4          interface Scheduler;
5          interface Init as PlatformInit;
6          interface Init as SoftwareInit;
7      }
8  }
9  implementation {
10       int main() __attribute__ ((C, spontaneous)) {
11           atomic {
12               platform_bootstrap();
13               call Scheduler.init();
14               call PlatformInit.init();
15               while (call Scheduler.runNextTask());
16               call SoftwareInit.init();
17               while (call Scheduler.runNextTask());
18           }
```

```
19            __nesc_enable_interrupt();
20            signal Boot.booted();
21            call Scheduler.taskLoop();
22            return -1;
23        }
```

8.3 Aplicação com uso de escalonador de prioridades

Código 8.4: Aplicação com escalonador de prioridades (Configuração)

```
1  /**
2   * Aplicativo de teste do Scheduler de prioridade
3   *
4   **/
5
6  #include "MsgSerial.h"
7  #include "Timer.h"
8  #include "printf.h"
9
10 configuration aplicacaoTesteAppC
11 {
12 }
13 implementation
14 {
15   components MainC, aplicacaoTesteC, LedsC, TinySchedulerC;
16   components CounterMicro32C as Timer1;
17
18   aplicacaoTesteC.Timer1 -> Timer1;
19
20   aplicacaoTesteC-> MainC.Boot;
21   aplicacaoTesteC.Leds -> LedsC;
22
23   aplicacaoTesteC.Tarefa1->
24   TinySchedulerC.TaskPrioridade[unique("TinySchedulerC.TaskPrioridade")];
25   aplicacaoTesteC.Tarefa2->
26   TinySchedulerC.TaskPrioridade[unique("TinySchedulerC.TaskPrioridade")];
27   aplicacaoTesteC.Tarefa3->
28   TinySchedulerC.TaskPrioridade[unique("TinySchedulerC.TaskPrioridade")];
29   aplicacaoTesteC.Tarefa4->
30   TinySchedulerC.TaskPrioridade[unique("TinySchedulerC.TaskPrioridade")];
31   aplicacaoTesteC.Tarefa5->
32   TinySchedulerC.TaskPrioridade[unique("TinySchedulerC.TaskPrioridade")];
```

```
33
34 // ...
35
36   aplicacaoTesteC.Tarefa98->
37   TinySchedulerC.TaskPrioridade[unique("TinySchedulerC.TaskPrioridade")];
38   aplicacaoTesteC.Tarefa99->
39   TinySchedulerC.TaskPrioridade[unique("TinySchedulerC.TaskPrioridade")];
40 }
```

Código 8.5: Aplicação com escalonador de prioridades (Módulo)

```
1
2 /**
3  * Implementa aplicativo de teste do Scheduler de prioridade
4  **/
5
6 #include "MsgSerial.h"
7 #include "Timer.h"
8 #include "printf.h"
9
10 module aplicacaoTesteC @safe()
11 {
12     uses interface Boot;
13     uses interface Leds;
14     uses interface TaskPrioridade as Tarefa1;
15     uses interface TaskPrioridade as Tarefa2;
16     uses interface TaskPrioridade as Tarefa3;
17     uses interface TaskPrioridade as Tarefa4;
18     uses interface TaskPrioridade as Tarefa5;
19     // ...
20     uses interface TaskPrioridade as Tarefa98;
21     uses interface TaskPrioridade as Tarefa99;
22
23     uses interface Counter<TMicro, uint32_t> as Timer1;
24 }
25 implementation
26 {
27     /* Variaveis */
28     unsigned int t1;
29     bool over;
30
31     async event void Timer1.overflow()
32     {
33         over = TRUE;
```

```
34      }
35
36      /* Boot
37      */
38      event void Boot.booted()
39      {
40          over = FALSE;
41          t1 = call Timer1.get();
42          printf("tempo inicial: %u\n", t1);
43          printfflush();
44
45          call Tarefa1.postTask(20);
46
47          call Tarefa2.postTask(10);
48          call Tarefa3.postTask(10);
49          // ...
50          call Tarefa98.postTask(10);
51          call Tarefa99.postTask(10);
52      }
53
54      /* Tarefas
55      */
56      event void Tarefa1.runTask()
57      {
58          uint16_t i = 0;
59          uint16_t k = 1;
60          for (i = 0; i < 65000; i++)
61          {
62              k = k * 2;
63          }
64          // Calculo do tempo de execucao
65          t1 = call Timer1.get();
66          printf("tempo final: %u\n", t1);
67          if (over == TRUE)
68              printf("Ocorreu Overflow\n");
69          printfflush();
70      }
71      event void Tarefa2.runTask()
72      {
73          uint16_t i = 0;
74          uint16_t k = 1;
75          for (i = 0; i < 65000; i++)
76          {
```

```
77              k = k * 2;
78          }
79      }
80
81      // ...
82
83      event void Tarefa99.runTask()
84      {
85          uint16_t i = 0;
86          uint16_t k = 1;
87          for (i = 0; i < 65000; i++)
88          {
89              k = k * 2;
90          }
91      }
92  }
```

8.4 Escalonador Deadline

Código 8.6: Escalonador Deadline

```
1  // $Id: SchedulerBasicP.nc,v 1.1.2.5 2006/02/14 17:01:46 idgay Exp $
2
3  /*                  tab:4
4   * "Copyright (c) 2000-2003 The Regents of the University  of California.
5   * All rights reserved.
6   *
7   * Permission to use, copy, modify, and distribute this software and its
8   * documentation for any purpose, without fee, and without written
9   * agreement is hereby granted, provided that the above
10  * copyright notice, the following
11  * two paragraphs and the author appear in all copies of this software.
12  *
13  * IN NO EVENT SHALL THE UNIVERSITY OF CALIFORNIA BE LIABLE TO ANY PARTY FOR
14  * DIRECT, INDIRECT, SPECIAL, INCIDENTAL, OR CONSEQUENTIAL DAMAGESi
15  * ARISING OUT  * OF THE USE OF THIS SOFTWARE AND ITS DOCUMENTATION, EVEN IF
16  * THE UNIVERSITY OF  * CALIFORNIA HAS BEEN ADVISED OF
17  * THE POSSIBILITY OF SUCH DAMAGE.
18  *
19  * THE UNIVERSITY OF CALIFORNIA SPECIFICALLY DISCLAIMS ANY WARRANTIES,
20  * INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY
21  * AND FITNESS FOR A PARTICULAR PURPOSE.  THE SOFTWARE PROVIDED
```

```
22  * HEREUNDER IS ON AN "AS IS" BASIS, AND THE UNIVERSITY OF
23  * CALIFORNIA HAS NO OBLIGATION TO
24  * PROVIDE MAINTENANCE, SUPPORT, UPDATES, ENHANCEMENTS, OR MODIFICATIONS."
25  *
26  * Copyright (c) 2002-2003 Intel Corporation
27  * All rights reserved.
28  *
29  * This file is distributed under the terms in the attached INTEL-LICENSE
30  * file. If you do not find these files, copies can be found by writing to
31  * Intel Research Berkeley, 2150 Shattuck Avenue, Suite 1300, Berkeley, CA,
32  * 94704.  Attention:  Intel License Inquiry.
33  */
34 /*
35  *
36  * Authors:    Philip Levis
37  * Date last modified:  $Id: SchedulerBasicP.nc,v
38  * 1.1.2.5 2006/02/14 17:01:46 idgay Exp $
39  *
40  */
41
42 /**
43  * SchedulerBasic implements the default TinyOS scheduler sequence, as
44  * documented in TEP 106.
45  *
46  * @author Philip Levis
47  * @author Cory Sharp
48  * @date   January 19 2005
49  */
50
51 #include "hardware.h"
52
53 module SchedulerDeadlineP {
54     provides interface Scheduler;
55     provides interface TaskBasic[uint8_t id];
56     provides interface TaskDeadline<TMicro>[uint8_t id];
57     uses interface McuSleep;
58     uses interface LocalTime<TMicro>;
59 }
60 implementation
61 {
62     enum
63     {
64         NUM_TASKS = uniqueCount("TinySchedulerC.TaskBasic"),
```

```
65          NUM_DTASKS = uniqueCount("TinySchedulerC.TaskDeadline"),
66          NO_TASK = 255,
67      };
68
69      volatile uint8_t m_head;
70      volatile uint8_t m_tail;
71      volatile uint8_t m_next[NUM_TASKS];
72      volatile uint8_t d_head;
73      volatile uint8_t d_tail;
74      volatile uint8_t d_next[NUM_DTASKS];
75      volatile uint32_t d_time[NUM_DTASKS];
76
77      // move the head forward
78      // if the head is at the end, mark the tail at the end, too
79      // mark the task as not in the queue
80      inline uint8_t popMTask()
81      {
82          if( m_head != NO_TASK )
83          {
84              uint8_t id = m_head;
85              m_head = m_next[m_head];
86              if( m_head == NO_TASK )
87              {
88                  m_tail = NO_TASK;
89              }
90              m_next[id] = NO_TASK;
91              return id;
92          }
93          else
94          {
95              return NO_TASK;
96          }
97      }
98
99      bool isMWaiting( uint8_t id )
100     {
101         return (m_next[id] != NO_TASK) || (m_tail == id);
102     }
103
104     bool pushMTask( uint8_t id )
105     {
106         if( !isMWaiting(id) )
107         {
```

```
108             if( m_head == NO_TASK )
109             {
110                 m_head = id;
111                 m_tail = id;
112             }
113             else
114             {
115                 m_next[m_tail] = id;
116                 m_tail = id;
117             }
118             return TRUE;
119         }
120         else
121         {
122             return FALSE;
123         }
124     }
125
126     inline uint8_t popDTask()
127     {
128         if( d_head != NO_TASK )
129         {
130             uint8_t id = d_head;
131             d_head = d_next[d_head];
132             if( d_head == NO_TASK )
133             {
134                 d_tail = NO_TASK;
135             }
136             d_next[id] = NO_TASK;
137             return id;
138         }
139         else
140         {
141             return NO_TASK;
142         }
143     }
144
145     bool isDWaiting( uint8_t id )
146     {
147         return (d_next[id] != NO_TASK) || (d_tail == id);
148     }
149
150     bool pushDTask( uint8_t id, uint32_t deadline )
```

```
151     {
152         if( !isDWaiting(id) )
153         {
154             if( d_head == NO_TASK )
155             {
156                 d_head = id;
157                 d_tail = id;
158             }
159             else
160             {
161                 uint8_t t_curr = d_head;
162                 uint8_t t_prev = d_head;
163                 uint32_t local = call LocalTime.get();
164                 while (d_time[t_curr] - local <= deadline &&
165                         t_curr != NO_TASK) {
166                     t_prev = t_curr;
167                     t_curr = d_next[t_curr];
168                 }
169                 d_next[id] = t_curr;
170                 if (t_curr == d_head) {
171                     d_head = id;
172                 }
173                 else {
174                     d_next[t_prev] = id;
175                     if (t_curr == NO_TASK) {
176                         d_tail = id;
177                     }
178                 }
179             }
180             d_time[id] = call LocalTime.get() + deadline;
181             return TRUE;
182         }
183         else
184         {
185             return FALSE;
186         }
187     }
188
189
190     command void Scheduler.init()
191     {
192         atomic
193         {
```

```
194             memset( (void *)m_next, NO_TASK, sizeof(m_next) );
195             m_head = NO_TASK;
196             m_tail = NO_TASK;
197             memset( (void *)d_next, NO_TASK, sizeof(d_next) );
198             d_head = NO_TASK;
199             d_tail = NO_TASK;
200             //      sim_scheduler_event_init(&sim_scheduler_event);
201         }
202     }
203
204     command bool Scheduler.runNextTask()
205     {
206         uint8_t nextTask;
207         atomic
208         {
209             nextTask = popDTask();
210             if( nextTask == NO_TASK )
211             {
212                 nextTask = popMTask();
213                 if (nextTask == NO_TASK) {
214                     return FALSE;
215                 }
216                 dbg("Deadline", "Running basic task %i\n", (int)nextTask);
217                 signal TaskBasic.runTask[nextTask]();
218                 return TRUE;
219             }
220         }
221         dbg("Deadline", "Running deadline task %i\n", (int)nextTask);
222         signal TaskDeadline.runTask[nextTask]();
223         return TRUE;
224     }
225
226     command void Scheduler.taskLoop()
227     {
228         for (;;)
229         {
230             uint8_t nextDTask = NO_TASK;
231             uint8_t nextMTask = NO_TASK;
232
233             atomic
234             {
235                 while ((nextDTask = popDTask()) == NO_TASK &&
236                         (nextMTask = popMTask()) == NO_TASK)
```

```
237                 {
238                     call McuSleep.sleep();
239                 }
240             }
241             if (nextDTask != NO_TASK) {
242                 dbg("Deadline", "Running deadline task %i\n",
243                     (int)nextDTask);
244                 signal TaskDeadline.runTask[nextDTask]();
245             }
246             else if (nextMTask != NO_TASK) {
247                 dbg("Deadline", "Running basic task %i\n", (int)nextMTask);
248                 signal TaskBasic.runTask[nextMTask]();
249             }
250         }
251     }
252
253     /**
254      * Return SUCCESS if the post succeeded, EBUSY if it was already posted.
255      */
256
257     async command error_t TaskBasic.postTask[uint8_t id]()
258     {
259         atomic return pushMTask(id) ? SUCCESS : EBUSY;
260     }
261
262     default event void TaskBasic.runTask[uint8_t id]()
263     {
264     }
265
266
267
268     async command error_t TaskDeadline.postTask[uint8_t id]
269                                             (uint32_t deadline)
270     {
271         atomic return pushDTask(id, deadline) ? SUCCESS : EBUSY;
272
273     }
274
275     default event void TaskDeadline.runTask[uint8_t id]()
276     {
277     }
278
279
```

```
280
281 }
```

8.5 Escalonador de Prioridades (Fila, sem envelhecimento)

Código 8.7: Escalonador de Prioridades (Fila, sem envelhecimento)

```
1  #define NO_STARVATION_NUM 10
2
3  //Define this to use on the TOSSIM
4  //#define SIM__
5
6  #include "hardware.h"
7
8  #ifdef SIM__
9  #include <sim_event_queue.h>
10 #endif
11
12
13 module SchedulerPrioridadeFilaP {
14     provides interface Scheduler;
15     provides interface TaskBasic[uint8_t id];
16     provides interface TaskPrioridade[uint8_t id];
17     uses interface McuSleep;
18 }
19 implementation
20 {
21     enum
22     {
23         NUM_TASKS = uniqueCount("TinySchedulerC.TaskBasic"),
24         NUM_PTASKS = uniqueCount("TinySchedulerC.TaskPrioridade"),
25         NO_TASK = 255,
26     };
27
28     volatile uint8_t m_head;
29     volatile uint8_t m_tail;
30     volatile uint8_t m_next[NUM_TASKS];
31     volatile uint8_t p_head;
32     volatile uint8_t p_tail;
33     volatile uint8_t p_next[NUM_PTASKS];
34     volatile uint8_t p_prioridade[NUM_PTASKS];
```

```
35
36      #ifdef SIM__
37      // Aqui entram as funcoes responsaveis pelos eventos do simulador
38      // As tasks sao simuladas por eventos no TOSSIM
39
40      bool sim_scheduler_event_pending = FALSE;
41      sim_event_t sim_scheduler_event;
42
43      int sim_config_task_latency() {return 100;}
44
45
46      /* Only enqueue the event for execution if it is
47         not already enqueued. If there are more tasks in the
48         queue, the event will re-enqueue itself (see the handle
49         function). */
50
51      void sim_scheduler_submit_event() {
52          if (sim_scheduler_event_pending == FALSE) {
53              sim_scheduler_event.time = sim_time() +
54                  sim_config_task_latency();
55              sim_queue_insert(&sim_scheduler_event);
56              sim_scheduler_event_pending = TRUE;
57          }
58      }
59
60      void sim_scheduler_event_handle(sim_event_t* e) {
61          sim_scheduler_event_pending = FALSE;
62
63          // If we successfully executed a task, re-enqueue the event. This
64          // will always succeed, as sim_scheduler_event_pending was just
65          // set to be false.  Note that this means there will be an extra
66          // execution (on an empty task queue). We could optimize this
67          // away, but this code is cleaner, and more accurately reflects
68          // the real TinyOS main loop.
69
70          if (call Scheduler.runNextTask()) {
71              sim_scheduler_submit_event();
72          }
73      }
74
75
76      /* Initialize a scheduler event. This should only be done
77       * once, when the scheduler is initialized. */
```

```
78      void sim_scheduler_event_init(sim_event_t* e) {
79          e->mote = sim_node();
80          e->force = 0;
81          e->data = NULL;
82          e->handle = sim_scheduler_event_handle;
83          e->cleanup = sim_queue_cleanup_none;
84      }
85      #endif
86
87      // move the head forward
88      // if the head is at the end, mark the tail at the end, too
89      // mark the task as not in the queue
90      inline uint8_t popMTask()
91      {
92          dbg("Prioridade", "Poped a Mtask (ou nao)\n");
93              if( m_head != NO_TASK )
94              {
95                  uint8_t id = m_head;
96                  m_head = m_next[m_head];
97                  if( m_head == NO_TASK )
98                  {
99                      m_tail = NO_TASK;
100                 }
101                 m_next[id] = NO_TASK;
102                 return id;
103             }
104             else
105             {
106                 return NO_TASK;
107             }
108     }
109
110     bool isMWaiting( uint8_t id )
111     {
112         dbg("Prioridade", "isMWaiting: %d\n",
113             (m_next[id] != NO_TASK) || (m_tail == id));
114         return (m_next[id] != NO_TASK) || (m_tail == id);
115     }
116
117     bool pushMTask( uint8_t id )
118     {
119         dbg("Prioridade", "pushMTask %i\n", (int)id);
120         if( !isMWaiting(id) )
```

```
121         {
122             if( m_head == NO_TASK )
123             {
124                 m_head = id;
125                 m_tail = id;
126             }
127             else
128             {
129                 m_next[m_tail] = id;
130                 m_tail = id;
131             }
132             return TRUE;
133         }
134         else
135         {
136             return FALSE;
137         }
138     }
139
140     inline uint8_t popPTask()
141     {
142         dbg("Prioridade", "Poped a Dtask (ou nao)\n");
143         if( p_head != NO_TASK )
144         {
145             uint8_t id = p_head;
146             p_head = p_next[p_head];
147             if( p_head == NO_TASK )
148             {
149                 p_tail = NO_TASK;
150             }
151             p_next[id] = NO_TASK;
152             dbg("Prioridade_run", "Rodou PTask %i\n", (int) id);
153             return id;
154         }
155         else
156         {
157             return NO_TASK;
158         }
159     }
160
161     bool isPWaiting( uint8_t id )
162     {
163         dbg("Prioridade", "isPWaiting: %d\n",
```

```
164             (p_next[id] != NO_TASK) || (p_tail == id));
165         return (p_next[id] != NO_TASK) || (p_tail == id);
166     }
167
168     bool pushPTask( uint8_t id, uint8_t prioridade )
169     {
170         dbg("Prioridade", "pushPTask %i\n", (int)id);
171         if( !isPWaiting(id) )
172         {
173             if( p_head == NO_TASK )
174             {
175                 p_head = id;
176                 p_tail = id;
177             }
178             else
179             {
180                 uint8_t t_curr = p_head;
181                 uint8_t t_prev = p_head;
182                 while (p_prioridade[t_curr] <= prioridade &&
183                         t_curr != NO_TASK) {
184                     t_prev = t_curr;
185                     t_curr = p_next[t_curr];
186                 }
187                 p_next[id] = t_curr;
188                 if (t_curr == p_head) {
189                     p_head = id;
190                 }
191                 else {
192                     p_next[t_prev] = id;
193                     if (t_curr == NO_TASK) {
194                         p_tail = id;
195                     }
196                 }
197             }
198             p_prioridade[id] = prioridade;
199             return TRUE;
200         }
201         else
202         {
203             return FALSE;
204         }
205     }
206
```

```
207
208     command void Scheduler.init()
209     {
210         dbg("Prioridade", "init\n");
211         atomic
212         {
213             memset( (void *)m_next, NO_TASK, sizeof(m_next) );
214             m_head = NO_TASK;
215             m_tail = NO_TASK;
216             memset( (void *)p_next, NO_TASK, sizeof(p_next) );
217             p_head = NO_TASK;
218             p_tail = NO_TASK;
219
220             #ifdef SIM__
221             sim_scheduler_event_pending = FALSE;
222             sim_scheduler_event_init(&sim_scheduler_event);
223             #endif
224         }
225     }
226
227     command bool Scheduler.runNextTask()
228     {
229         uint8_t nextTask;
230         dbg("Prioridade", "runNextTask\n");
231         atomic
232         {
233             nextTask = popPTask();
234             dbg("Prioridade", "popPTask: %i\n", (int)nextTask);
235             if( nextTask == NO_TASK )
236             {
237                 nextTask = popMTask();
238                 dbg("Prioridade", "popMTask: %i\n", (int)nextTask);
239                 if (nextTask == NO_TASK) {
240                     return FALSE;
241                 }
242                 dbg("Prioridade", "Running basic task %i\n", (int)nextTask);
243                 signal TaskBasic.runTask[nextTask]();
244                 return TRUE;
245             }
246         }
247         dbg("Prioridade", "Running prioridade task %i\n", (int)nextTask);
248         signal TaskPrioridade.runTask[nextTask]();
249         return TRUE;
```

```
250     }
251
252     command void Scheduler.taskLoop()
253     {
254         uint8_t max_ptask = 0;
255         dbg("Prioridade", "Taskloop\n");
256         for (;;)
257         {
258             uint8_t nextPTask = NO_TASK;
259             uint8_t nextMTask = NO_TASK;
260
261             if (max_ptask > NO_STARVATION_NUM)
262             {
263                 max_ptask = 0;
264                 atomic {
265                 nextMTask = popMTask();
266                 }
267                 if (nextMTask != NO_TASK)
268                     signal TaskBasic.runTask[nextMTask]();
269             }
270             if (nextMTask == NO_TASK)
271             {
272                 atomic
273                 {
274                     while ((nextPTask = popPTask()) == NO_TASK &&
275                             (nextMTask = popMTask()) == NO_TASK)
276                     {
277                         call McuSleep.sleep();
278                     }
279                 }
280                 if (nextPTask != NO_TASK) {
281                     dbg("Prioridade", "Running prioridade task %i\n",
282                         (int)nextPTask);
283                     max_ptask++;
284                     signal TaskPrioridade.runTask[nextPTask]();
285                 }
286                 else if (nextMTask != NO_TASK) {
287                     dbg("Prioridade", "Running basic task %i\n",
288                         (int)nextMTask);
289                     max_ptask = 0;
290                     signal TaskBasic.runTask[nextMTask]();
291                 }
292             }
```

```
293         }
294     }
295
296     /**
297      * Return SUCCESS if the post succeeded, EBUSY if it was already posted.
298      */
299
300     async command error_t TaskBasic.postTask[uint8_t id]()
301     {
302         error_t result;
303
304         dbg("Prioridade", "postTaskBasic\n");
305         atomic {
306             result = pushMTask(id) ? SUCCESS : EBUSY;
307         }
308         #ifdef SIM__
309         if (result == SUCCESS)
310             sim_scheduler_submit_event();
311         #endif
312
313         return result;
314
315     }
316
317     default event void TaskBasic.runTask[uint8_t id]()
318     {
319     }
320
321
322
323     async command error_t TaskPrioridade.postTask[uint8_t id]
324                                             (uint8_t prioridade)
325     {
326         error_t result;
327
328         dbg("Prioridade", "postTaskBasic\n");
329         atomic {
330             result = pushPTask(id, prioridade) ? SUCCESS : EBUSY;
331         }
332         #ifdef SIM__
333         if (result == SUCCESS)
334             sim_scheduler_submit_event();
335         #endif
```

```
336
337         return result;
338     }
339
340     default event void TaskPrioridade.runTask[uint8_t id]()
341     {
342     }
343
344 }
```

8.6 Escalonador de Prioridades (Fila, com envelhecimento)

Código 8.8: Escalonador de Prioridades (Fila, com envelhemcimento

```
1  #define NO_STARVATION_NUM 10
2
3  //Define this to use on the TOSSIM
4  //#define SIM__
5
6  #include "hardware.h"
7
8  #ifdef SIM__
9  #include <sim_event_queue.h>
10 #endif
11
12
13 module SchedulerPrioridadeFilaAgingP {
14     provides interface Scheduler;
15     provides interface TaskBasic[uint8_t id];
16     provides interface TaskPrioridade[uint8_t id];
17     uses interface McuSleep;
18 }
19 implementation
20 {
21     enum
22     {
23         NUM_TASKS = uniqueCount("TinySchedulerC.TaskBasic"),
24         NUM_PTASKS = uniqueCount("TinySchedulerC.TaskPrioridade"),
25         NO_TASK = 255,
26     };
27
```

```
28      volatile uint8_t m_head;
29      volatile uint8_t m_tail;
30      volatile uint8_t m_next[NUM_TASKS];
31      volatile uint8_t p_head;
32      volatile uint8_t p_tail;
33      volatile uint8_t p_next[NUM_PTASKS];
34      volatile uint8_t p_prioridade[NUM_PTASKS];
35
36      #ifdef SIM__
37      // Aqui entram as funcoes responsaveis pelos eventos do simulador
38      // As tasks sao simuladas por eventos no TOSSIM
39
40      bool sim_scheduler_event_pending = FALSE;
41      sim_event_t sim_scheduler_event;
42
43      int sim_config_task_latency() {return 100;}
44
45
46      /* Only enqueue the event for execution if it is
47         not already enqueued. If there are more tasks in the
48         queue, the event will re-enqueue itself (see the handle
49         function). */
50
51      void sim_scheduler_submit_event() {
52          if (sim_scheduler_event_pending == FALSE) {
53              sim_scheduler_event.time = sim_time() +
54                  sim_config_task_latency();
55              sim_queue_insert(&sim_scheduler_event);
56              sim_scheduler_event_pending = TRUE;
57          }
58      }
59
60      void sim_scheduler_event_handle(sim_event_t* e) {
61          sim_scheduler_event_pending = FALSE;
62
63          // If we successfully executed a task, re-enqueue the event. This
64          // will always succeed, as sim_scheduler_event_pending was just
65          // set to be false.  Note that this means there will be an extra
66          // execution (on an empty task queue). We could optimize this
67          // away, but this code is cleaner, and more accurately reflects
68          // the real TinyOS main loop.
69
70          if (call Scheduler.runNextTask()) {
```

```
71              sim_scheduler_submit_event();
72          }
73      }
74
75
76      /* Initialize a scheduler event. This should only be done
77       * once, when the scheduler is initialized. */
78      void sim_scheduler_event_init(sim_event_t* e) {
79          e->mote = sim_node();
80          e->force = 0;
81          e->data = NULL;
82          e->handle = sim_scheduler_event_handle;
83          e->cleanup = sim_queue_cleanup_none;
84      }
85      #endif
86
87      // move the head forward
88      // if the head is at the end, mark the tail at the end, too
89      // mark the task as not in the queue
90      inline uint8_t popMTask()
91      {
92          dbg("Prioridade", "Poped a Mtask (ou nao)\n");
93              if( m_head != NO_TASK )
94              {
95                  uint8_t id = m_head;
96                  m_head = m_next[m_head];
97                  if( m_head == NO_TASK )
98                  {
99                      m_tail = NO_TASK;
100                 }
101                 m_next[id] = NO_TASK;
102                 return id;
103             }
104             else
105             {
106                 return NO_TASK;
107             }
108     }
109
110     bool isMWaiting( uint8_t id )
111     {
112         dbg("Prioridade", "isMWaiting: %d\n",
113             (m_next[id] != NO_TASK) || (m_tail == id));
```

```
114         return (m_next[id] != NO_TASK) || (m_tail == id);
115     }
116
117     bool pushMTask( uint8_t id )
118     {
119         dbg("Prioridade", "pushMTask %i\n", (int)id);
120         if( !isMWaiting(id) )
121         {
122             if( m_head == NO_TASK )
123             {
124                 m_head = id;
125                 m_tail = id;
126             }
127             else
128             {
129                 m_next[m_tail] = id;
130                 m_tail = id;
131             }
132             return TRUE;
133         }
134         else
135         {
136             return FALSE;
137         }
138     }
139
140     inline uint8_t popPTask()
141     {
142         uint8_t atual;
143         dbg("Prioridade", "Poped a Dtask (ou nao)\n");
144         if( p_head != NO_TASK )
145         {
146             uint8_t id = p_head;
147             p_head = p_next[p_head];
148             if( p_head == NO_TASK )
149             {
150                 p_tail = NO_TASK;
151             }
152             p_next[id] = NO_TASK;
153
154             //antes de retornar, aumentar a prioridade de todas tarefas
155             atual = p_head;
156             while (atual != NO_TASK)
```

```
157             {
158                 if (p_prioridade[atual] > 0)
159                     p_prioridade[atual]--;
160                 atual = p_next[atual];
161             }
162 
163             dbg("Prioridade_run", "Rodou PTask %i\n\tcom prioridade %i\n",
164                 (int) id, (int) p_prioridade[id]);
165             return id;
166         }
167         else
168         {
169             return NO_TASK;
170         }
171     }
172 
173     bool isPWaiting( uint8_t id )
174     {
175         dbg("Prioridade", "isPWaiting: %d\n",
176             (p_next[id] != NO_TASK) || (p_tail == id));
177         return (p_next[id] != NO_TASK) || (p_tail == id);
178     }
179 
180     bool pushPTask( uint8_t id, uint8_t prioridade )
181     {
182         dbg("Prioridade", "pushPTask %i\n", (int)id);
183         if( !isPWaiting(id) )
184         {
185             if( p_head == NO_TASK )
186             {
187                 p_head = id;
188                 p_tail = id;
189             }
190             else
191             {
192                 uint8_t t_curr = p_head;
193                 uint8_t t_prev = p_head;
194                 while (p_prioridade[t_curr] <= prioridade &&
195                         t_curr != NO_TASK) {
196                     t_prev = t_curr;
197                     t_curr = p_next[t_curr];
198                 }
199                 p_next[id] = t_curr;
```

```
200                 if (t_curr == p_head) {
201                     p_head = id;
202                 }
203                 else {
204                     p_next[t_prev] = id;
205                     if (t_curr == NO_TASK) {
206                         p_tail = id;
207                     }
208                 }
209             }
210             p_prioridade[id] = prioridade;
211             return TRUE;
212         }
213         else
214         {
215             return FALSE;
216         }
217     }
218
219
220     command void Scheduler.init()
221     {
222         dbg("Prioridade", "init\n");
223         atomic
224         {
225             memset( (void *)m_next, NO_TASK, sizeof(m_next) );
226             m_head = NO_TASK;
227             m_tail = NO_TASK;
228             memset( (void *)p_next, NO_TASK, sizeof(p_next) );
229             p_head = NO_TASK;
230             p_tail = NO_TASK;
231
232             #ifdef SIM__
233             sim_scheduler_event_pending = FALSE;
234             sim_scheduler_event_init(&sim_scheduler_event);
235             #endif
236         }
237     }
238
239     command bool Scheduler.runNextTask()
240     {
241         uint8_t nextTask;
242         dbg("Prioridade", "runNextTask\n");
```

```
243         atomic
244         {
245             nextTask = popPTask();
246             dbg("Prioridade", "popPTask: %i\n", (int)nextTask);
247             if( nextTask == NO_TASK )
248             {
249                 nextTask = popMTask();
250                 dbg("Prioridade", "popMTask: %i\n", (int)nextTask);
251                 if (nextTask == NO_TASK) {
252                     return FALSE;
253                 }
254                 dbg("Prioridade", "Running basic task %i\n", (int)nextTask);
255                 signal TaskBasic.runTask[nextTask]();
256                 return TRUE;
257             }
258         }
259         dbg("Prioridade", "Running prioridade task %i\n", (int)nextTask);
260         signal TaskPrioridade.runTask[nextTask]();
261         return TRUE;
262     }
263
264     command void Scheduler.taskLoop()
265     {
266         uint8_t max_ptask = 0;
267         dbg("Prioridade", "Taskloop\n");
268         for (;;)
269         {
270             uint8_t nextPTask = NO_TASK;
271             uint8_t nextMTask = NO_TASK;
272
273             if (max_ptask > NO_STARVATION_NUM)
274             {
275                 max_ptask = 0;
276                 atomic {
277                 nextMTask = popMTask();
278                 }
279                 if (nextMTask != NO_TASK)
280                     signal TaskBasic.runTask[nextMTask]();
281             }
282             if (nextMTask == NO_TASK)
283             {
284                 atomic
285                 {
```

```
286                     while ((nextPTask = popPTask()) == NO_TASK &&
287                             (nextMTask = popMTask()) == NO_TASK)
288                     {
289                         call McuSleep.sleep();
290                     }
291                 }
292                 if (nextPTask != NO_TASK) {
293                     dbg("Prioridade", "Running prioridade task %i\n",
294                         (int)nextPTask);
295                     max_ptask++;
296                     signal TaskPrioridade.runTask[nextPTask]();
297                 }
298                 else if (nextMTask != NO_TASK) {
299                     dbg("Prioridade", "Running basic task %i\n",
300                         (int)nextMTask);
301                     max_ptask = 0;
302                     signal TaskBasic.runTask[nextMTask]();
303                 }
304             }
305         }
306     }
307
308     /**
309      * Return SUCCESS if the post succeeded, EBUSY if it was already posted.
310      */
311
312     async command error_t TaskBasic.postTask[uint8_t id]()
313     {
314         error_t result;
315
316         dbg("Prioridade", "postTaskBasic\n");
317         atomic {
318             result = pushMTask(id) ? SUCCESS : EBUSY;
319         }
320         #ifdef SIM__
321         if (result == SUCCESS)
322             sim_scheduler_submit_event();
323         #endif
324
325         return result;
326
327     }
328
```

```
329     default event void TaskBasic.runTask[uint8_t id]()
330     {
331     }
332
333
334
335     async command error_t TaskPrioridade.postTask[uint8_t id]
336                                         (uint8_t prioridade)
337     {
338         error_t result;
339
340         dbg("Prioridade", "postTaskBasic\n");
341         atomic {
342             result = pushPTask(id, prioridade) ? SUCCESS : EBUSY;
343         }
344         #ifdef SIM__
345         if (result == SUCCESS)
346             sim_scheduler_submit_event();
347         #endif
348
349         return result;
350     }
351
352     default event void TaskPrioridade.runTask[uint8_t id]()
353     {
354     }
355
356 }
```

8.7 Escalonador de Prioridades (Heap, sem envelhecimento)

Código 8.9: Escalonador de Prioridades (Heap, sem envelhecimento)

```
1 /* Escalonador de prioridade
2     Utiliza uma heap como fila de prioridades
3
4     Autor: Pedro Rosanes
5 */
6
7 //Descomentar para rodar no simulador
8 //#define SIM__
```

```
9
10 #include "hardware.h"
11
12 #ifdef SIM__
13 #include <sim_event_queue.h>
14 #endif
15
16 #define NO_STARVATION_NUM 10
17
18 module SchedulerPrioridadeHeapP {
19     provides interface Scheduler;
20     provides interface TaskBasic[uint8_t id];
21     provides interface TaskPrioridade[uint8_t id];
22     uses interface McuSleep;
23 }
24 implementation
25 {
26     enum
27     {
28         NUM_TASKS = uniqueCount("TinySchedulerC.TaskBasic"),
29         NUM_MTASKS = uniqueCount("TinySchedulerC.TaskPrioridade"),
30         NO_TASK = 255,
31     };
32
33     volatile uint8_t m_head;
34     volatile uint8_t m_tail;
35     volatile uint8_t m_next[NUM_TASKS];
36     volatile uint8_t tamanho;
37     volatile uint8_t p_fila[NUM_MTASKS];
38     volatile uint8_t p_prioridade[NUM_MTASKS];
39     volatile uint8_t p_isDWaiting[NUM_MTASKS];
40
41     // Aqui entram as funcoes responsaveis pelos eventos do simulador
42     // As tasks sao simuladas por eventos no TOSSIM
43
44     #ifdef SIM__
45     bool sim_scheduler_event_pending = FALSE;
46     sim_event_t sim_scheduler_event;
47
48     int sim_config_task_latency() {return 100;}
49
50
51     /* Only enqueue the event for execution if it is
```

```
52          not already enqueued. If there are more tasks in the
53          queue, the event will re-enqueue itself (see the handle
54          function). */
55
56      void sim_scheduler_submit_event() {
57          if (sim_scheduler_event_pending == FALSE) {
58              sim_scheduler_event.time = sim_time() +
59                  sim_config_task_latency();
60              sim_queue_insert(&sim_scheduler_event);
61              sim_scheduler_event_pending = TRUE;
62          }
63      }
64
65      void sim_scheduler_event_handle(sim_event_t* e) {
66          sim_scheduler_event_pending = FALSE;
67
68          // If we successfully executed a task, re-enqueue the event. This
69          // will always succeed, as sim_scheduler_event_pending was just
70          // set to be false.  Note that this means there will be an extra
71          // execution (on an empty task queue). We could optimize this
72          // away, but this code is cleaner, and more accurately reflects
73          // the real TinyOS main loop.
74
75          if (call Scheduler.runNextTask()) {
76              sim_scheduler_submit_event();
77          }
78      }
79
80
81      /* Initialize a scheduler event. This should only be done
82       * once, when the scheduler is initialized. */
83      void sim_scheduler_event_init(sim_event_t* e) {
84          e->mote = sim_node();
85          e->force = 0;
86          e->data = NULL;
87          e->handle = sim_scheduler_event_handle;
88          e->cleanup = sim_queue_cleanup_none;
89      }
90      #endif
91
92      // move the head forward
93      // if the head is at the end, mark the tail at the end, too
94      // mark the task as not in the queue
```

```
95      inline uint8_t popMTask()
96      {
97          dbg("Prioridade", "Poped a Mtask (ou nao)\n");
98          if( m_head != NO_TASK )
99          {
100             uint8_t id = m_head;
101             m_head = m_next[m_head];
102             if( m_head == NO_TASK )
103             {
104                 m_tail = NO_TASK;
105             }
106             m_next[id] = NO_TASK;
107             return id;
108         }
109         else
110         {
111             return NO_TASK;
112         }
113     }
114
115     bool isMWaiting( uint8_t id )
116     {
117         dbg("Prioridade", "isMWaiting: %d\n",
118             (m_next[id] != NO_TASK) || (m_tail == id));
119         return (m_next[id] != NO_TASK) || (m_tail == id);
120     }
121
122     bool pushMTask( uint8_t id )
123     {
124         dbg("Prioridade", "pushMTask %i\n", (int)id);
125         if( !isMWaiting(id) )
126         {
127             if( m_head == NO_TASK )
128             {
129                 m_head = id;
130                 m_tail = id;
131             }
132             else
133             {
134                 m_next[m_tail] = id;
135                 m_tail = id;
136             }
137             return TRUE;
```

```
138         }
139         else
140         {
141             return FALSE;
142         }
143     }
144
145     inline uint8_t popPTask()
146     {
147         uint8_t id, i, menor, temp;
148
149         dbg("Prioridade", "Poped a Dtask (ou nao)\n");
150         //Se nao tem ninguem na fila
151         if (tamanho == 0)
152             return NO_TASK;
153
154         //Se tem alguem na fila
155         id = p_fila[0];
156
157         p_fila[0] = p_fila[tamanho-1];
158         p_prioridade[0] = p_prioridade[tamanho-1];
159         tamanho--;
160
161         i = 0;
162         while (i < tamanho)
163         {
164             menor = i;
165             if (2*i+1 < tamanho &&
166                     p_prioridade[2*i+1] < p_prioridade[menor])
167                 menor = 2*i+1;
168             if (2*i+2 < tamanho &&
169                     p_prioridade[2*i+2] < p_prioridade[menor])
170                 menor = 2*i+2;
171
172             if (menor != i)
173             {
174                 temp = p_fila[i];
175                 p_fila[i] = p_fila[menor];
176                 p_fila[menor] = temp;
177
178                 temp = p_prioridade[i];
179                 p_prioridade[i] = p_prioridade[menor];
180                 p_prioridade[menor] = temp;
```

```
181             }
182             else
183                 break;
184             i = menor;
185         }
186 
187         p_isDWaiting[id] = 0;
188         dbg("Prioridade_run", "Rodou PTask %i\n", (int) id);
189         return id;
190 
191     }
192 
193     bool pushPTask( uint8_t id, uint8_t prioridade )
194     {
195         int16_t temp, pai, filho;
196 
197         dbg("Prioridade", "pushPTask %i\n", (int)id);
198         if( !p_isDWaiting[id] )
199         {
200             p_isDWaiting[id] = 1;
201 
202             p_fila[tamanho] = id;
203             p_prioridade[tamanho] = prioridade;
204             pai = (tamanho - 1)/2;
205             filho = tamanho;
206             while (pai >= 0)
207             {
208                 if (p_prioridade[pai] > p_prioridade[filho])
209                 {
210                     temp = p_fila[pai];
211                     p_fila[pai] = p_fila[filho];
212                     p_fila[filho] = temp;
213 
214                     temp = p_prioridade[pai];
215                     p_prioridade[pai] = p_prioridade[filho];
216                     p_prioridade[filho] = temp;
217                 }
218                 else
219                    break;
220 
221                 filho = pai;
222                 pai = (filho -1)/2;
223             }
```

```
224             tamanho++;
225             return TRUE;
226         }
227         else
228         {
229             return FALSE;
230         }
231     }
232
233
234     command void Scheduler.init()
235     {
236         dbg("Prioridade", "init\n");
237         atomic
238         {
239             memset( (void *)m_next, NO_TASK, sizeof(m_next) );
240             m_head = NO_TASK;
241             m_tail = NO_TASK;
242             memset( (void *)p_fila, NO_TASK, sizeof(p_fila) );
243             memset( (void *)p_prioridade, NO_TASK, sizeof(p_prioridade) );
244             memset( (void *)p_isDWaiting, 0, sizeof(p_isDWaiting) );
245             tamanho = 0;
246
247             #ifdef SIM__
248             sim_scheduler_event_pending = FALSE;
249             sim_scheduler_event_init(&sim_scheduler_event);
250             #endif
251         }
252     }
253
254     command bool Scheduler.runNextTask()
255     {
256         uint8_t nextTask;
257         dbg("Prioridade", "runNextTask\n");
258         atomic
259         {
260             nextTask = popPTask();
261             dbg("Prioridade", "popPTask: %i\n", (int)nextTask);
262             if( nextTask == NO_TASK )
263             {
264                 nextTask = popMTask();
265                 dbg("Prioridade", "popMTask: %i\n", (int)nextTask);
266                 if (nextTask == NO_TASK) {
```

```
267                     return FALSE;
268                 }
269                 dbg("Prioridade", "Running basic task %i\n", (int)nextTask);
270                 signal TaskBasic.runTask[nextTask]();
271                 return TRUE;
272             }
273         }
274         dbg("Prioridade", "Running prioridade task %i\n", (int)nextTask);
275         signal TaskPrioridade.runTask[nextTask]();
276         return TRUE;
277     }
278 
279     command void Scheduler.taskLoop()
280     {
281         uint8_t max_ptask = 0;
282         dbg("Prioridade", "Taskloop\n");
283         for (;;)
284         {
285             uint8_t nextPTask = NO_TASK;
286             uint8_t nextMTask = NO_TASK;
287 
288             if (max_ptask > NO_STARVATION_NUM)
289             {
290                 max_ptask = 0;
291                 atomic {
292                     nextMTask = popMTask();
293                 }
294                 if (nextMTask != NO_TASK)
295                     signal TaskBasic.runTask[nextMTask]();
296             }
297             if (nextMTask == NO_TASK)
298             {
299                 atomic
300                 {
301                     while ((nextPTask = popPTask()) == NO_TASK &&
302                             (nextMTask = popMTask()) == NO_TASK)
303                     {
304                         call McuSleep.sleep();
305                     }
306                 }
307                 if (nextPTask != NO_TASK) {
308                     dbg("Prioridade", "Running prioridade task %i\n",
309                         (int)nextPTask);
```

```
310                     max_ptask++;
311                     signal TaskPrioridade.runTask[nextPTask]();
312                 }
313                 else if (nextMTask != NO_TASK) {
314                     dbg("Prioridade", "Running basic task %i\n",
315                         (int)nextMTask);
316                     max_ptask = 0;
317                     signal TaskBasic.runTask[nextMTask]();
318                 }
319             }
320         }
321     }
322
323     /**
324      * Return SUCCESS if the post succeeded, EBUSY if it was already posted.
325      */
326
327     async command error_t TaskBasic.postTask[uint8_t id]()
328     {
329         error_t result;
330
331         dbg("Prioridade", "postTaskBasic\n");
332         atomic {
333             result = pushMTask(id) ? SUCCESS : EBUSY;
334         }
335         #ifdef SIM__
336         if (result == SUCCESS)
337             sim_scheduler_submit_event();
338         #endif
339
340         return result;
341
342     }
343
344     default event void TaskBasic.runTask[uint8_t id]()
345     {
346     }
347
348
349
350     async command error_t TaskPrioridade.postTask[uint8_t id]
351                                             (uint8_t prioridade)
352     {
```

```
353         error_t result;
354
355         dbg("Prioridade", "postTaskBasic\n");
356         atomic {
357             result = pushPTask(id, prioridade) ? SUCCESS : EBUSY;
358         }
359         #ifdef SIM__
360         if (result == SUCCESS)
361             sim_scheduler_submit_event();
362         #endif
363
364         return result;
365     }
366
367     default event void TaskPrioridade.runTask[uint8_t id]()
368     {
369     }
370
371
372
373 }
```

8.8 Escalonador de Prioridades (Heap, com envelhecimento)

Código 8.10: Escalonador de Prioridades (Heap, com envelhecimento)

```
1  /* Escalonador de prioridade
2      Utiliza uma heap como fila de prioridades
3
4      Autor: Pedro Rosanes
5  */
6
7  //Descomentar para rodar no simulador
8  //#define SIM__
9
10 #include "hardware.h"
11
12 #ifdef SIM__
13 #include <sim_event_queue.h>
14 #endif
15
```

```
16 #define NO_STARVATION_NUM 10
17
18 module SchedulerPrioridadeHeapAgingP {
19     provides interface Scheduler;
20     provides interface TaskBasic[uint8_t id];
21     provides interface TaskPrioridade[uint8_t id];
22     uses interface McuSleep;
23 }
24 implementation
25 {
26     enum
27     {
28         NUM_TASKS = uniqueCount("TinySchedulerC.TaskBasic"),
29         NUM_MTASKS = uniqueCount("TinySchedulerC.TaskPrioridade"),
30         NO_TASK = 255,
31     };
32
33     volatile uint8_t m_head;
34     volatile uint8_t m_tail;
35     volatile uint8_t m_next[NUM_TASKS];
36     volatile uint8_t tamanho;
37     volatile uint8_t p_fila[NUM_MTASKS];
38     volatile uint8_t p_prioridade[NUM_MTASKS];
39     volatile uint8_t p_isDWaiting[NUM_MTASKS];
40
41     // Aqui entram as funcoes responsaveis pelos eventos do simulador
42     // As tasks sao simuladas por eventos no TOSSIM
43
44     #ifdef SIM__
45     bool sim_scheduler_event_pending = FALSE;
46     sim_event_t sim_scheduler_event;
47
48     int sim_config_task_latency() {return 100;}
49
50
51     /* Only enqueue the event for execution if it is
52        not already enqueued. If there are more tasks in the
53        queue, the event will re-enqueue itself (see the handle
54        function). */
55
56     void sim_scheduler_submit_event() {
57         if (sim_scheduler_event_pending == FALSE) {
58             sim_scheduler_event.time = sim_time() +
```

```
59                  sim_config_task_latency();
60              sim_queue_insert(&sim_scheduler_event);
61              sim_scheduler_event_pending = TRUE;
62          }
63      }
64
65      void sim_scheduler_event_handle(sim_event_t* e) {
66          sim_scheduler_event_pending = FALSE;
67
68          // If we successfully executed a task, re-enqueue the event. This
69          // will always succeed, as sim_scheduler_event_pending was just
70          // set to be false.  Note that this means there will be an extra
71          // execution (on an empty task queue). We could optimize this
72          // away, but this code is cleaner, and more accurately reflects
73          // the real TinyOS main loop.
74
75          if (call Scheduler.runNextTask()) {
76              sim_scheduler_submit_event();
77          }
78      }
79
80
81      /* Initialize a scheduler event. This should only be done
82       * once, when the scheduler is initialized. */
83      void sim_scheduler_event_init(sim_event_t* e) {
84          e->mote = sim_node();
85          e->force = 0;
86          e->data = NULL;
87          e->handle = sim_scheduler_event_handle;
88          e->cleanup = sim_queue_cleanup_none;
89      }
90      #endif
91
92      // move the head forward
93      // if the head is at the end, mark the tail at the end, too
94      // mark the task as not in the queue
95      inline uint8_t popMTask()
96      {
97          dbg("Prioridade", "Poped a Mtask (ou nao)\n");
98          if( m_head != NO_TASK )
99          {
100             uint8_t id = m_head;
101             m_head = m_next[m_head];
```

```
102             if( m_head == NO_TASK )
103             {
104                 m_tail = NO_TASK;
105             }
106             m_next[id] = NO_TASK;
107             return id;
108         }
109         else
110         {
111             return NO_TASK;
112         }
113     }
114
115     bool isMWaiting( uint8_t id )
116     {
117         dbg("Prioridade", "isMWaiting: %d\n",
118             (m_next[id] != NO_TASK) || (m_tail == id));
119         return (m_next[id] != NO_TASK) || (m_tail == id);
120     }
121
122     bool pushMTask( uint8_t id )
123     {
124         dbg("Prioridade", "pushMTask %i\n", (int)id);
125         if( !isMWaiting(id) )
126         {
127             if( m_head == NO_TASK )
128             {
129                 m_head = id;
130                 m_tail = id;
131             }
132             else
133             {
134                 m_next[m_tail] = id;
135                 m_tail = id;
136             }
137             return TRUE;
138         }
139         else
140         {
141             return FALSE;
142         }
143     }
144
```

```
145     inline uint8_t popPTask()
146     {
147         uint8_t id, i, menor, temp;
148 
149         dbg("Prioridade", "Poped a Dtask (ou nao)\n");
150         //Se nao tem ninguem na fila
151         if (tamanho == 0)
152             return NO_TASK;
153 
154         //Se tem alguem na fila
155         id = p_fila[0];
156         dbg("Prioridade_run", "Rodou PTask %i\n\tcom prioridade %i",
157             (int) p_fila[0], (int) p_prioridade[0]);
158 
159         p_fila[0] = p_fila[tamanho-1];
160         p_prioridade[0] = p_prioridade[tamanho-1];
161         tamanho--;
162 
163         i = 0;
164         while (i < tamanho)
165         {
166             menor = i;
167             if (2*i+1 < tamanho &&
168                     p_prioridade[2*i+1] < p_prioridade[menor])
169                 menor = 2*i+1;
170             if (2*i+2 < tamanho &&
171                     p_prioridade[2*i+2] < p_prioridade[menor])
172                 menor = 2*i+2;
173 
174             if (menor != i)
175             {
176                 temp = p_fila[i];
177                 p_fila[i] = p_fila[menor];
178                 p_fila[menor] = temp;
179 
180                 temp = p_prioridade[i];
181                 p_prioridade[i] = p_prioridade[menor];
182                 p_prioridade[menor] = temp;
183             }
184             else
185                 break;
186             i = menor;
187         }
```

```
188
189         p_isDWaiting[id] = 0;
190
191         //Antes de retornar, aumenta a prioridade de todos que estao na fila.
192         for (i = 0; i < tamanho; i--)
193             if (p_prioridade[i] > 0)
194                 p_prioridade[i]--;
195
196         return id;
197
198     }
199
200     bool pushPTask( uint8_t id, uint8_t prioridade )
201     {
202         int16_t temp, pai, filho;
203
204         dbg("Prioridade", "pushPTask %i\n", (int)id);
205         if( !p_isDWaiting[id] )
206         {
207             p_isDWaiting[id] = 1;
208
209             p_fila[tamanho] = id;
210             p_prioridade[tamanho] = prioridade;
211             pai = (tamanho - 1)/2;
212             filho = tamanho;
213             while (pai >= 0)
214             {
215                 if (p_prioridade[pai] > p_prioridade[filho])
216                 {
217                     temp = p_fila[pai];
218                     p_fila[pai] = p_fila[filho];
219                     p_fila[filho] = temp;
220
221                     temp = p_prioridade[pai];
222                     p_prioridade[pai] = p_prioridade[filho];
223                     p_prioridade[filho] = temp;
224                 }
225                 else
226                    break;
227
228                 filho = pai;
229                 pai = (filho-1)/2;
230             }
```

```
231             tamanho++;
232             return TRUE;
233         }
234         else
235         {
236             return FALSE;
237         }
238     }
239
240
241     command void Scheduler.init()
242     {
243         dbg("Prioridade", "init\n");
244         atomic
245         {
246             memset( (void *)m_next, NO_TASK, sizeof(m_next) );
247             m_head = NO_TASK;
248             m_tail = NO_TASK;
249             memset( (void *)p_fila, NO_TASK, sizeof(p_fila) );
250             memset( (void *)p_prioridade, NO_TASK, sizeof(p_prioridade) );
251             memset( (void *)p_isDWaiting, 0, sizeof(p_isDWaiting) );
252             tamanho = 0;
253
254             #ifdef SIM__
255             sim_scheduler_event_pending = FALSE;
256             sim_scheduler_event_init(&sim_scheduler_event);
257             #endif
258         }
259     }
260
261     command bool Scheduler.runNextTask()
262     {
263         uint8_t nextTask;
264         dbg("Prioridade", "runNextTask\n");
265         atomic
266         {
267             nextTask = popPTask();
268             dbg("Prioridade", "popPTask: %i\n", (int)nextTask);
269             if( nextTask == NO_TASK )
270             {
271                 nextTask = popMTask();
272                 dbg("Prioridade", "popMTask: %i\n", (int)nextTask);
273                 if (nextTask == NO_TASK) {
```

```
274                     return FALSE;
275                 }
276                 dbg("Prioridade", "Running basic task %i\n", (int)nextTask);
277                 signal TaskBasic.runTask[nextTask]();
278                 return TRUE;
279             }
280         }
281         dbg("Prioridade", "Running prioridade task %i\n", (int)nextTask);
282         signal TaskPrioridade.runTask[nextTask]();
283         return TRUE;
284     }
285 
286     command void Scheduler.taskLoop()
287     {
288         uint8_t max_ptask = 0;
289         dbg("Prioridade", "Taskloop\n");
290         for (;;)
291         {
292             uint8_t nextPTask = NO_TASK;
293             uint8_t nextMTask = NO_TASK;
294 
295             if (max_ptask > NO_STARVATION_NUM)
296             {
297                 max_ptask = 0;
298                 atomic {
299                     nextMTask = popMTask();
300                 }
301                 if (nextMTask != NO_TASK)
302                     signal TaskBasic.runTask[nextMTask]();
303             }
304             if (nextMTask == NO_TASK)
305             {
306                 atomic
307                 {
308                     while ((nextPTask = popPTask()) == NO_TASK &&
309                             (nextMTask = popMTask()) == NO_TASK)
310                     {
311                         call McuSleep.sleep();
312                     }
313                 }
314                 if (nextPTask != NO_TASK) {
315                     dbg("Prioridade", "Running prioridade task %i\n",
316                         (int)nextPTask);
```

```
317                     max_ptask++;
318                     signal TaskPrioridade.runTask[nextPTask]();
319                 }
320                 else if (nextMTask != NO_TASK) {
321                     dbg("Prioridade", "Running basic task %i\n",
322                         (int)nextMTask);
323                     max_ptask = 0;
324                     signal TaskBasic.runTask[nextMTask]();
325                 }
326             }
327         }
328     }
329
330     /**
331      * Return SUCCESS if the post succeeded, EBUSY if it was already posted.
332      */
333
334     async command error_t TaskBasic.postTask[uint8_t id]()
335     {
336         error_t result;
337
338         dbg("Prioridade", "postTaskBasic\n");
339         atomic {
340             result = pushMTask(id) ? SUCCESS : EBUSY;
341         }
342         #ifdef SIM__
343         if (result == SUCCESS)
344             sim_scheduler_submit_event();
345         #endif
346
347         return result;
348
349     }
350
351     default event void TaskBasic.runTask[uint8_t id]()
352     {
353     }
354
355
356
357     async command error_t TaskPrioridade.postTask[uint8_t id]
358                                             (uint8_t prioridade)
359     {
```

```
360         error_t result;
361
362         dbg("Prioridade", "postTaskBasic\n");
363         atomic {
364             result = pushPTask(id, prioridade) ? SUCCESS : EBUSY;
365         }
366         #ifdef SIM__
367         if (result == SUCCESS)
368             sim_scheduler_submit_event();
369         #endif
370
371         return result;
372     }
373
374     default event void TaskPrioridade.runTask[uint8_t id]()
375     {
376     }
377
378
379
380 }
```

8.9 Aplicação com uso de escalonador Multi-nível

Código 8.11: Aplicação com uso de escalonador Multi-nivel

```
1  #include "Timer.h"
2
3  module aplicacaoTesteC @safe()
4  {
5      uses interface Boot;
6      uses interface Leds;
7      uses interface TaskSerial as TarefaSerial;
8      uses interface TaskRadio as TarefaRadio;
9      uses interface TaskSense as TarefaSense;
10     uses interface Timer<TMilli>;
11
12     uses interface Read<uint16_t>;
13
14     uses interface Packet;
15     uses interface AMSend;
16     uses interface SplitControl as RadioControl;
```

```
17
18      uses interface SerialPacket;
19      uses interface SerialAMSend;
20      uses interface SerialSplitControl as SerialControl;
21  }
22  implementation
23  {
24      /* Variaveis */
25      unsigned int t1;
26      uint16_t valorLido;
27      message_t packet;
28      message_t serialPacket;
29
30      /* Tarefas
31      */
32      task void TarefaBasic()
33      {
34          printf("tarefa Basic\n");
35          printfflush();
36      }
37
38      event void TarefaSense.runTask()
39      {
40          call Read.read();
41      }
42
43      event void TarefaRadio.runTask()
44      {
45          uint16_t *msg;
46
47          msg = call Packet.getPayload(&packet, sizeof(uint16_t));
48
49          (*msg) = valorLido;
50          call AMSend.send(AM_BROADCAST_ADDR, &packet, sizeof(uint16_t));
51      }
52
53      event void TarefaSerial.runTask()
54      {
55          uint16_t *msg;
56
57          msg = call Packet.getPayload(&serialPacket, sizeof(uint16_t));
58
59          (*msg) = valorLido;
```

```
60          call AMSend.send(AM_BROADCAST_ADDR, &packet, sizeof(uint16_t));
61      }
62
63      /* events */
64      event void Boot.booted()
65      {
66          call RadioControl.start();
67          call SerialControl.start();
68          call Timer.startPeriodic(250);
69      }
70
71      event void RadioControl.startDone(error_t err) {
72          if (err == SUCCESS)
73              radioOn = 1;
74      }
75      event void RadioControl.stopDone(error_t err) {}
76
77      event void SerialControl.startDone(error_t err) {
78          if (err = SUCCESS)
79              serialOn = 1;
80      }
81      event void SerialControl.stopDone(error_t err) {}
82
83      event void Timer.fired()
84      {
85          call TarefaSense.postTask();
86      }
87
88      event void Read.readDone(error_t result, uint16_t data)
89      {
90          valorLido = data;
91          if (radioOn) call TarefaRadio.postTask();
92          if (serialOn) call TarefaSerial.postTask();
93      }
94
95  }
```

8.10 Aplicação de teste do escalonador padrão

Código 8.12: Aplicação de teste do escalonador padrão

```
1
```

```
2  /**
3   * Implementa aplicativo de teste do Scheduler de prioridade
4   **/
5
6  #include "Timer.h"
7  #include "printf.h"
8
9  module aplicacaoTesteC @safe()
10 {
11     uses interface Boot;
12     uses interface Leds;
13
14     uses interface Counter<TMicro, uint32_t> as Timer1;
15 }
16 implementation
17 {
18     /* Variaveis */
19     unsigned int t1;
20     bool over;
21
22     async event void Timer1.overflow()
23     {
24         over = TRUE;
25     }
26
27
28     /* Tarefas
29     */
30     task void Tarefa1()
31     {
32         uint16_t i = 0;
33         uint16_t k = 1;
34         for (i = 0; i < 65000; i++)
35         {
36             k = k * 2;
37         }
38         t1 = call Timer1.get();
39         printf("tempo final: %u\n", t1);
40         if (over == TRUE)
41             printf("OVERFLOW!!\n");
42         printfflush();
43     }
44     task void Tarefa2()
```

```
45      {
46          uint16_t i = 0;
47          uint16_t k = 1;
48          for (i = 0; i < 65000; i++)
49          {
50              k = k * 2;
51          }
52      }
53
54      // ...
55
56      task void Tarefa99()
57      {
58          uint16_t i = 0;
59          uint16_t k = 1;
60          for (i = 0; i < 65000; i++)
61          {
62              k = k * 2;
63          }
64      }
65
66      /* Boot
67      */
68      event void Boot.booted()
69      {
70          over = FALSE;
71          t1 = call Timer1.get();
72          printf("tempo inicial: %u\n", t1);
73          printfflush();
74
75          post Tarefa2();
76          post Tarefa3();
77          // ...
78          post Tarefa99();
79
80          post Tarefa1();
81      }
82  }
```

Bibliografia

[1] LEVIS, P.; GAY, D. *Tinyos programming*. Cambridge University Press, 2009.

[2] HILL, J.; SZEWCZYK, R.; WOO, A.; HOLLAR, S.; CULLER, D.; PISTER, K. System architecture directions for networked sensors. In: . New York, NY, USA: ACM Press, c2000. p. 93–104.

[3] GAY, D.; LEVIS, P.; VON BEHREN, R.; WELSH, M.; BREWER, E.; CULLER, D. The nesC language: A holistic approach to networked embedded systems. *Proceedings of Programming Language Design and Implementation (PLDI)*, June 2003.

[4] LEVIS, P.; SHARP, C. Tep106: Schedulers and tasks. http://www.tinyos.net/tinyos-2.1.0/doc/html/tep106.html.

[5] DE MOURA, A. L. *Revisitando co-rotinas*. 2004. Tese (Doutorado em Física) - PUC-Rio, Rio de Janeiro, Brasil, 2004.

[6] WEBSITE. Micaz datasheet. http://www.openautomation.net/ uploadsproductos/micaz_datasheet.pdf.

[7] WEBSITE, T. Universidade de Berkeley. http://www.tinyos.net.

[8] LEVIS, P. Tep107: Tinyos 2.x boot sequence. http://www.tinyos.net/tinyos-2.1.0/doc/html/tep107.html.

[9] KLUES, K.; LIANG, C.-J.; PAEK, J.; MUSALOIU-E, R.; GOVINDAN, R.; TERZIS, A.; LEVIS, P. Tep134: The tosthreads thread library. http://www.tinyos.net/tinyos-2.1.0/doc/html/tep134.html.

[10] STALLINGS, W. *Operating systems: Internals and design principles*. 5a. ed. Prentice Hall, 2004.