Extensão dos mecanismos de gerência de tarefas do sistema operacional TinyOS

Bolsista: Pedro Rosanes Orientador: Silvana Rossetto

Departamento de Ciência da Computação

11 de abril de 2011

Sumário

Resumo

Resumo 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. O sistema operacional mais usado na programação desses dispositivos é o TinyOS, um sistema leve, projetado especialmente para consumir pouca energia, um dos requisitos mais importante para RSSFs. O modelo de programação adotado pelo TinyOS prioriza o atendimento de interrupções. Em função disso, as operações são normalmente 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, 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, e um modelo de programação cooperativo para o TinyOS, visando a construção de abstrações de programação de nível mais alto que facilitem o desenvolvimento de aplicações nessa área.

1 Introdução

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 redes de sensores devem lidar apropriadamente com as restrições e características particulares desses ambientes. A arquitetura adotada pelo TinyOS [1] — 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 o nesC [2], uma extensão de C que provê um modelo de programação baseado em componentes e orietado 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). Este comando retorna imediatamente dando continuidade à execução. Quando o serviço é terminado, o hardware envia uma interrupção, sinalizada como um evento pelo TinyOS. Então, o tratador do evento recebe as informações (ex.: valor sensoreado) e trata/processa essas informações conforme programado[3]. 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 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 elas), por isso as condições de corrida, neste contexto, são evitadas. As tarefas são todas escalonadas por um componente do TinyOS que usa uma política padrão de escalonamento do tipo *First-in First-out* [3].

Com o objetivo de oferecer maior flexibilidade aos desenvolvedores de aplicações, a versão mais atual do TinyOS (versão 2.1.x) trouxe novas facilidades. Uma delas é a possibilidade de substituir o componente de escalonamento de tarefas para implementar diferentes políticas de escalonamento [4]. A outra é 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.

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. Avaliamos o modelo de multithreading oferecido, comparando diferentes formas de implementação de uma aplicação básica e o custo da gerência de threads. 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. Visamos uma solução alternativa entre o modelo de escalonamento de tarefas que executam até terminar, e o modelo de execução alternada entre as tarefas 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 fios 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. 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.

Acreditamos que os escalonadores desenvolvidos oferecerão uma maior flexibilidade à programação, facilitando o desenvolvimento de algoritmos complexos. Por meio de experimentos, constatamos que o custo de escolamento aumentou de trinta a cem por cento, para uma quantidade razoável de vinte tarefas, dependendo do escalonador utilizado. A gerência cooperativa de tarefas desenvolvida facilitou a programação ao transformar os comandos de duas fases em comandos bloqueantes de uma fase, e ao eliminar a necessidade de gerência do uso concorrente de recursos. Nos experimentos realizados, o tempo de processamento necessário para gerênciar rotinas cooperativas atingiu metade do tempo necessário para gerências threads do modelo de programação multithreading.

Na seção 2 abordamos conceitos básicos relacionados a redes de sensores sem fio, ao sistema

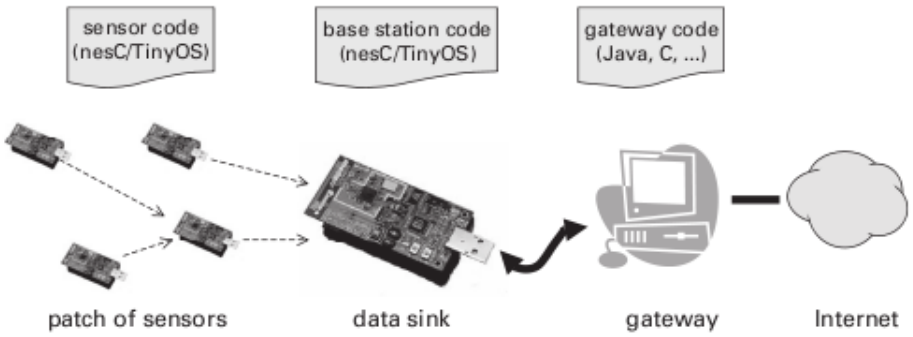


Figura 1: Rede de sensores sem fio

operacional TinyOS, e à sua linguagem de programação *nesC*. Além disto, detalhamos as etapas da sequência de inicialização, e descrevemos o modelo de concorrência. Na seção 3 fazemos uma introdução teórica sobre escalonamento de tarefas, e apresentamos os escalonadores propostos. Por último, descrevemos os experimentos realizados e os resultados obtidos. Na seção 4 abordamos os conceitos de *multithreading* e co-rotinas. Descrevemos o uso e a implementação da biblioteca *TOSThreads* que oferece um modelo de programação *multithreding* ao TinyOS. Depois, detalhamos a nossa implementação de gerência cooperativa de tarefas, e mostramos um exemplo de uso de co-rotinas. Por último, descrevemos os experimentos realizados para comparar estes dois modelos e apresentamos os resultados obtidos. NA seção 5 apresentamos algumas conclusões.

2 Conceitos Básicos

2.1 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 todas essas informações para detectar um evento importante no local, como um incêndio. Essa conclusão é então encaminhada para um nó com maior capacidade computacional, conhecido como estação base. Este nó pode decidir uma ação a ser tomada, ou enviar a informação pela Internet.

Estes sensores são desenvolvidos para monitoriar ambientes de difícil acesso, portanto, devem ser pequenos e utilizar comunicação sem fio para facilitar a instalação no ambiente e minimizar o custo financeiro. Para evitar manutenções frequêntes, eles também devem consumir pouca energia. Devido a estas características, o hardware destes dispositivos tende a ter recursos computacionais limitados. Ao invés de utilizar CPUs, são usados microcontroladores de 8 ou 16 bits, normalmente, com baixas frequências de relógio. Para armazenar o código da aplicação é utilizada uma pequena memória flash, da ordem de 100kB, e para as variáveis existe uma memória RAM, da ordem de 10kB. Os circuitos de rádio também têm uma capaci-

dade reduzida de transferência, da ordem de kilobytes por segundos [3]. Aliado ao hardware, o software também deve ser voltado para o baixo consumo de energia e de memória. Detalhes serão vistos na próxima seção.

Uma RSSF é usada para monitorar ambientes de difícil acesso, onde uma rede cabeada seria inviável ou custosa. Alguns exemplos reais do uso de RSSF são: monitoramento da ponte Golden Gate em São Francisco, e dos vulcões Reventador e Tungurahua no Equador [3].

2.2 TinyOS e nesC

O TinyOS é o sistema operacional mais usado para auxiliar os programadores a desenvolverem aplicações para rede de sensores sem fio de baixo consumo. O modelo de programação provido é baseado em componentes e orientado a eventos. Os componentes são pedaços de código reutilizáveis, onde são definidas claramente suas dependências e os serviços oferecidos, por meio de interfaces. A linguagem *nesc* implementa esse modelo estendendo a linguagem C. É através da conexão (*wiring*) de diversos componentes que o sistema é montado.

Já o modelo de programação orientado a eventos permite que o TinyOS rode 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 a execução. A conclusão da operação é sinalizada através de um evento, que será tratado pela segunda fase da operação de E/S.

O modelo de programação baseada em compontenes está intimamente ligada à programação orientada a eventos: Um componente oferece a interface, implementando os comandos e sinalizando eventos relacionandos, enquanto outro componente utiliza esta interface, através do uso dos comandos e da implementação dos tratadores de evento.

A divisão em componentes também facilita a implementação da camada de abstração de hardware. De forma que cada plataforma tem 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.

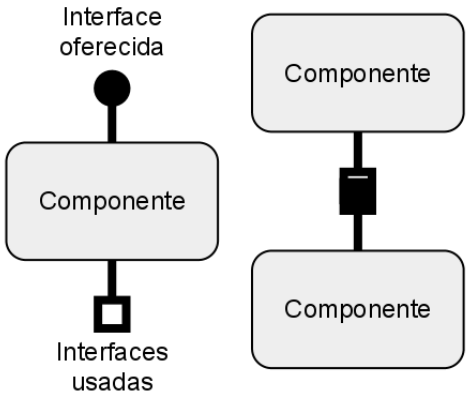


Figura 2: Ilustração de componentes e suas interfaces

Exemplo A aplicação *Blink*, no anexo B.1, é utilizada para ilustrar estes conceitos básicos. Esta aplicação faz os LEDs da plataforma piscarem continuamente. Toda aplicação utiliza uma configuração para descrever os componentes que serão usados, e quais são as conexões entre as interfaces.

Listing 1: Configuração (BlinkAppC.nc)

```
1  configuration BlinkAppC
   {}
3  implementation
   {
5    components MainC, BlinkC, LedsC;
     components new TimerMilliC() as Timer0;
7    components new TimerMilliC() as Timer1;
     components new TimerMilliC() as Timer2;
9
     BlinkC.Boot -> MainC.Boot;
11   BlinkC.Timer0 -> Timer0;
     BlinkC.Timer1 -> Timer1;
13   BlinkC.Timer2 -> Timer2;
     BlinkC.Leds -> LedsC.Leds;
15 }
```

Na listagem 1, pode-se ver que alguns dos componentes utilizados são *MainC*, *BlinkC*, *LedsC*. *MainC* é o responsável pela inicialização do sistema. E indica o termino deste processo através do evento *booted*, da interface *Boot* (mais detalhes na seção 2.3). *BlinkC* implementa a lógica da aplicação. E *LedsC* implementa as operações necessárias para acender ou apagar os Leds da plataforma.

O outro componente utilizado é um temporizador. O comando *new* é usado para criar instâncias de componentes genéricos. Isso permite a criação de cópias distintas de uma mesma funcionalidade. Neste caso são criados três temporizadores diferentes. Alguns componentes não podem ter cópias distintas, como o *LedsC* que representa uma estrutra física do sensor, logo não pode ser multiplicada.

Na listagem 1 também são definidas as conexões das interfaces de cada componente. A construção da linha 14, por exemplo, indica que o módulo *BlinkC* utiliza a interface oferecida por *LedsC* para apagar ou acender os LEDs.

A lógica da aplicação está implementada no componente *BlinkC*, através da contrução de um módulo..

Listing 2: Interfaces usadas pelo módulo (BlinkC.nc)

```
   #include "Timer.h"
2
   module BlinkC @safe()
4  {
     uses interface Timer<TMilli> as Timer0;
6    uses interface Timer<TMilli> as Timer1;
```

```
  uses interface Timer<TMilli> as Timer2;
8 uses interface Leds;
  uses interface Boot;
10 }
```

Na listagem 2, estão especificadas as interfaces utilizadas, que formaram as conexões vistas na configuração.

Listing 3: Eventos, Comandos, Postagem de tarefas (BlinkC.nc)

```
implementation
12 {
  event void Boot.booted()
14 {
    call Timer0.startPeriodic( 250 );
16  call Timer1.startPeriodic( 500 );
    call Timer2.startPeriodic( 1000 );
18
    post tarefa();
20 }

22 event void Timer0.fired()
  {
24  call Leds.led0Toggle();
  }
```

Finalmente, na listagem 3, é definida a lógica do componente principal da aplicação. O tratador do evento *Boot.booted* é o primeiro a ser ativado após a inicialização do sistema. Dentro deste, é invocado o comando para inicializar os temporizadores. A periodicidade de cada um é definida pelo parâmetro passado. E por último, é postada uma tarefa, cujos detalhes serão vistos a seguir. O tratador do evento *Timer0.fired*, toda vez que é ativado, invoca o comando responsável por acender/apagar o LED 0. O mesmo acontece para os outros temporizadores.

Tasks O TinyOS também utiliza o conceito de procedimento postergados, chamados de tarefas (*tasks*). As próprias tarefas, comando e tratadores de eventos podem postar uma nova tarefas, a qual é enfileirada para execução posterior. Esta será atendida, de forma síncrona, pelo escalonador. Ser executada de forma síncrona, significa que tarefas não são preemptiva entre si. Portanto, se diversas tarefas compartilha uma mesma variável, não haverá condições de corrida. Mais detalhes serão vistos na seção 2.4.

Listing 4: Implementação de tarefas (BlinkC.nc)

```
  task void tarefa()
38 {
      dbg("BlinkC", "tarefa\n");
40 }
}
```

Na listagem 3, existe um exemplo de uma postagem de tarefa. E na listagem 4, um exemplo da implementação de uma tarefa, que neste caso somente emite uma mensagem de depuração.

Interfaces Ao desenvolver aplicações mais complexas, é preciso desenvolver componentes intermediários. Estes devem além de usar, também devem oferecer interfaces. Para oferecer novas interfaces, o programador deve declará-las, implementar seus comandos, e sinalizar seus eventos. O componente que usar estas interfaces, será responsável por utilizar os comandos e implementar os tratadores de eventos.

Listing 5: interface

```
interface Send {
    command error_t send(message_t* msg, uint8_t len);
    command error_t cancel(message_t* msg);
    event void sendDone(message_t* msg, error_t error);
    command uint8_t maxPayloadLength();
    command void* getPayload(message_t* msg, uint8_t len);
}

module SendExample {
    provides interface Send;
}
implementation {
    command error_t Send.send(message_t* msg, uint8_t len) {
        //Implementacao do comando send.
        ...

        signal Send.sendDone(msg_var, SUCCESS);
    }
    ...
}
```

Interfaces não são uma relação de um para um, e sim de n para m. Ou seja, diversos componentes podem utilizar ou oferecer uma mesma interface. Isto permite que diversos códigos diferentes sejam chamados por somente um comando. Esta propriedade é chamada de *fan-out*. Por exemplo, suponha que dois componentes utilizam a interface SplitControl (6). Logo, os dois componentes deveram implementar os tratadores dos eventos *startDone* e *stopDone*. O *fan-out* ocorre quando um evento é sinalizado, o que leva a execução dos dois tratadores. Esta propriedade também ocorre quando uma interface que é oferecida por mais de um componente, tem um comando executado.

Listing 6: SplitControl

```
interface SplitControl {
    command error_t start();
    event void startDone(error_t error);

    command error_t stop();
```

```
6     event void stopDone(error_t error);
}
```

2.3 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* se liga aos componentes *RealmainP*, *PlataformC*, *TinySchedulerC*, e utiliza a interface *SoftwareInit*.

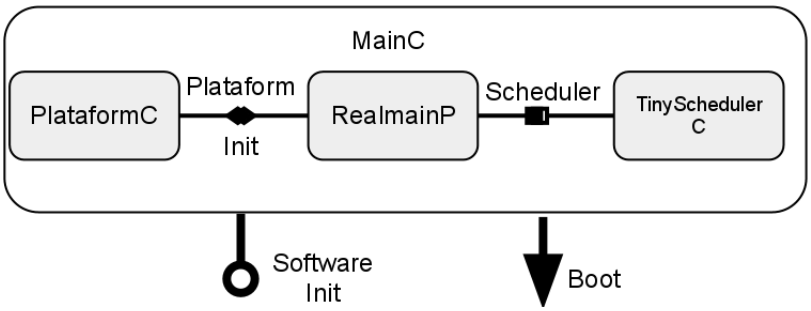


Figura 3: MainC

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.

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 [5].

Listing 7: Código de inicialização

```
1 module RealMainP {
    provides interface Booted;
3   uses {
      interface Scheduler;
```

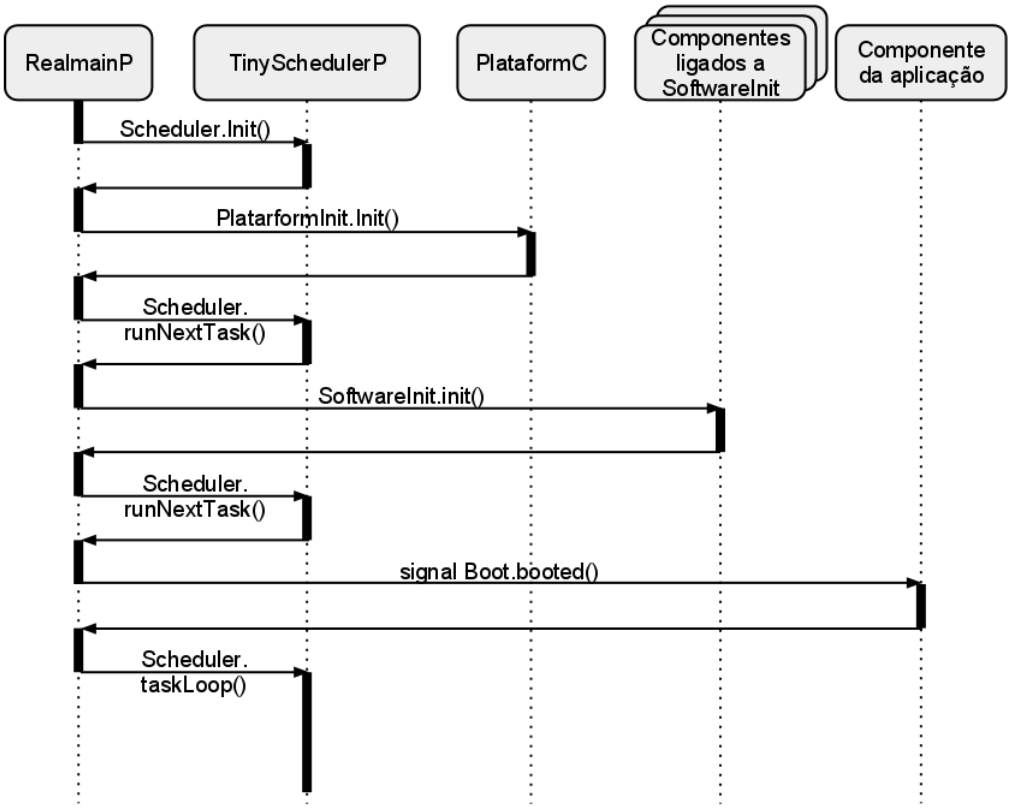


Figura 4: Sequência de Inicialização

```
5         interface Init as PlatformInit;
          interface Init as SoftwareInit;
7     }
  }
9 implementation {
      int main() __attribute__ ((C, spontaneous)) {
11        atomic {
              platform_bootstrap();
13            call Scheduler.init();
              call PlatformInit.init();
15            while (call Scheduler.runNextTask());
              call SoftwareInit.init();
17            while (call Scheduler.runNextTask());
          }
19        __nesc_enable_interrupt();
          signal Boot.booted();
21        call Scheduler.taskLoop();
          return -1;
23    }
```

2.4 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 ao postar uma tarefa, o fluxo de execução continua, sem desvio, e ela só será processada mais tarde. Na definição básica do TinyOS, as tarefas não recebem parâmetros e não retornam resultados.

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 2.3, 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. E 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 termino 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 devem ser marcados como *async* no código fonte. Para contornar isto, deve-se usar 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. Existe a possibilidade de requisição imediata. 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. [3, Cap.11].

3 Escalonamento de tarefas

Nesta seção é feita uma abordagem teórica sobre escalonamento de tarefas. Depois apresentamos a implementação do escalonador padrão de tarefas do TinyOS. Também apresentamos 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. Por último mostramos os experimentos e resultados obtidos, usados para comprar os diferentes escalonadores.

3.1 Abordagem teórica sobre escalonamento de tarefas

As tarefas, por serem procedimentos adiados, necessitam de algoritmos de escalonamento. Estes algoritmos também não podem ser preemptivos, devido a natureza das tarefas do TinyOS. O algoritmo mais simples, 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 [6]. Neste algoritmo a próxima tarefa a ser executada é aquela com menor prazo (*deadline*). As diversas variações deste algoritmo utilizam parâmetros como: tempo de entrada na fila de prontos, prazo para começar a tarefa, prazo para terminar a tarefa, tempo de processamento, recursos utilizados, prioridade, existência de preempção. Porém, o principal parâmetro utilizado pelos algoritmos é a existência ou não de preempção. 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 [6]. Um *overhead* maior passa a existir, devido à 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 outra 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 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.

3.2 Escalonador padrão 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 contruções:

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

Essas duas contruções são transformadas pelo compilador, fazendo com que a aplicação implemente uma interface chamada *TaskBasic*. A primeira 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 sintaxe 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 [3].

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

Todo escalonador, além de prover a interface *TaskBasic*, também deve prover a interface *Scheduler*.

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

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.

Para criar novos tipos de tarefas, é preciso definir uma interface nova, com o comando *postTask* e o evento *runTask* que será provida pelo escalonador como visto acima. Por exemplo:

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

Na 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 aceitar a interface de tarefa padrão (*TaskBasic*), e 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. 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. Por exemplo:

```
configuration TinySchedulerC {
    provides interface Scheduler;
    provides interface TaskBasic[uint8_t id];
    provides interface TaskDeadline<TMilli>[uint8_t id];
}
implementation {
    components SchedulerDeadlineP as Sched;
    ...
    Scheduler = Sched;
    TaskBasic = Sched;
    TaskDeadline = Sched;
}
```

Um exemplo de aplicação que utiliza um escalonador novo pode ser visto no anexo B.3. Para que o escalonador funcione corretamente no simulador TOSSIM é preciso adicionar funções que lidam com eventos no simulador. Um exemplo desta extensão poder ser achada no arquivo *tinyOS-root-dir/tos/lib/tossim/SimSchedulerBasicP.nc*, ou no apêndice A

3.3 Escalonador EDF (*Earliest Deadline First*)

Este escalonador (anexo B.7.1)[1] aceita tarefas com deadline e elege aquelas com menor *deadline* para executar. A interface usada para criar esse tipo de tarefas é *TaskDeadline*. O *deadline* é passado por parâmetro pela função *postTask*. As tarefas básicas (*TaskBasic*) também são aceitas, como recomendado pelo TEP 106[4].

Em contraste, 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

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

deadline esperando para executar. A fila é implementada da mesma forma que a do escalonador padrão3.2, 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 $\bigcirc(1)$.

3.4 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 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 B.7.2)
2. Fila comum com envelhecimento (anexo B.7.3)
3. Heap sem envelhecimento (anexo B.7.4)
4. Heap com envelhecimento (anexo B.7.5)

A seguir uma tabela com a complexidade de inserção e remoção para cada escalonador:

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	$\bigcirc(\log(n))$	$\bigcirc(n)$

3.5 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. Uma aplicação de exemplo pode ser vista no anexo B.4.

3.6 Experimentos e resultados obtidos

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 B.2). 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. Os valores medidos não variaram mais de uma unidade entre diferentes execuções, e cada cenário foi executado dez vezes.

Escalonador	20 Tarefas	50 Tarefas	100 Tarefas
Escalonador Padrão	1366	1849	2652

Experimentos com o escalonador com prioridades Para avaliar o desempenho com o escalonador com prioridades foi desenvolvida a mesma aplicação de teste (anexo B.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 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. Os valores medidos não variaram mais de uma unidade entre diferentes execuções, e cada cenário foi executado dez vezes.

Escalonador	20 Tarefas	50 Tarefas	100 Tarefas
Escalonador Padrão	1366	1849	2652
Fila, sem envelhecimento	1733	4660	13721
Heap, sem envelhecimento	2603	4308	7486
Fila, com envelhecimento	2278	7887	26066
Heap, com envelhecimento	2665	4510	7887

Podemos perceber que, para um número pequeno de tarefas, a fila é mais eficiente que a heap. não é compensado o *overhead* do algoritmo da heap.

4 Modelos de programação

Nesta seção, primeiro é feita uma abordagem teórica sobre Multithreading e co-rotinas. Também apresentamos a biblioteca *TinyOS Threads*, que oferece um modelo de programação em threads, como alternativa ao modelo orientado a eventos. Depois apresentamos o projeto e as etapas de implementação de co-rotinas para o TinyOS. Por último mostramos os experimentos e resultados obtidos, usados para comprar o modelo de threads com o de co-rotinas.

4.1 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, chamadas de *threads*. Cada *thread* contém um contexto que inclui instruções, variáveis, uma pilha de execução, e um bloco de controle. O suporte de diversar 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 multiplos 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 excução pode ser interrompida, após qualquer instrução, para ceder a CPU a outra *thread*. Esta técnica permite que a CPU seja usada por todos, 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 a preempção modifica a sequência de instruções de uma operação. Para permitir que a operação execute sem interrupções, são utilizadas primitivas que desabilitam a preempção temporariamente, garantindo a exclusão mútua de tais regiões. Quando diversas *threads* estão trabalhando em conjunto, as vezes é preciso garantir uma ordem de execução. Isso é garantido com o uso de primitivas de sincronização. Porém essas primitivas que gerenciam o uso concorrente de recurso são custosas. [6]

Rotinas coopertativas, ou co-rotinas, têm as mesmas características das *threads*, quando classificadas como completas [7, 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 possuêm um comportamente semelhante ao comportamento de funções. [7]

4.2 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 executa 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.

Três tipos de fluxo de execução passam a existir: tarefas, interrupções e threads. Como foi visto na seção **??**, 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 força a troca de contexto da thread, e caso seu tratador poste uma tarefa, esta será executada antes da thread retomar o controle.

Trocas de Contextos acontecem por três motivos diferentes: ocorrência de uma interrupção, termino do tempo de execução da thread, ou chamadas bloqueantes ao sistema. 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*. Caso contrário, a thread continua a executar logo após o termino 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.

Chamadas de sistemas foram introduzidas para transformar chamadas de duas fases em chamadas de uma fase. 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* e acorda a thread do kernel. Eventualmente, a tarefa executará a primeira fase do serviço. Na segunda fase, o resultado é enviado à thread, e esta é desbloqueada.

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

Mutex Garante a exclusão mútua, como visto na seção 4.1.

Semáforo Garante uma ordem de execução, como visto na seção 4.1.

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[8].

4.2.1 Exemplo de aplicação produtor/consumidor

Nesta seção ilustraremos o uso de threads no TinyOS, por meio de uma aplicação que utiliza o modelo produtor/consumidor.

Ao codificar o módulo principal, é preciso definir quantas threads serão utilizadas.

```
4  module BenchmarkC {
     uses {
6      interface Boot;
       interface Thread as Produtor;
8      interface Thread as Consumidor;
       interface Thread as SerialSender;
10     interface Leds;
```

E quais primitivas de gerência de concorrência.

```
18     interface Mutex;
       interface Semaphore;
20   }
  }
22
  implementation {
24     uint32_t t1;
       uint32_t * tempo;
26     uint8_t buffer;
       mutex_t mutex_buffer;
28     semaphore_t sem_produto, sem_termino, sem_buffer_cheio;
```

O próximo passo é inicializar estas primitivas, e as threads.

```
     event void Boot.booted() {
34     buffer = 0;
       t1 = call Timer.get();
36
       call Mutex.init(&mutex_buffer);
38     call Semaphore.reset(&sem_produto, 0);
       call Semaphore.reset(&sem_buffer_cheio, 1);
40     call Semaphore.reset(&sem_termino, 0);

42     call Produtor.start(NULL);
       call Consumidor.start(NULL);
```

```
44        call SerialSender.start(NULL);
    }
```

A seguir a thread responsável por criar os produtos. Aqui podemos ver como é feito o uso das primitivas de gerência de concorrência.

```
72    event void Produtor.run(void* arg) {
        uint16_t counter = 1;
74      uint16_t num_prods, j;

76      for (num_prods = 0; num_prods < 1000; num_prods++)
        {
78          call Semaphore.acquire(&sem_buffer_cheio);

80          //Tempo simulado para criar um produto
            for (j = 0; j < 100; j++)
82              counter *= 3;

84          call Mutex.lock(&mutex_buffer);
                buffer = counter;
86          call Mutex.unlock(&mutex_buffer);

88          call Semaphore.release(&sem_produto);
        }
90    }
```

A thread consumidora também é a responsável por acordar a thread que terminará de calcular o tempo de execução de todo o programa.

```
92    event void Consumidor.run(void* arg) {
        uint16_t num_prods, j;
94      uint16_t counter = 0;

96      for (num_prods = 0; num_prods < 1000; num_prods++)
        {
98          call Semaphore.acquire(&sem_produto);

100         call Mutex.lock(&mutex_buffer);
                counter = buffer;
102         call Mutex.unlock(&mutex_buffer);

104         //Tempo simulado para consumir produto
            for (j = 0; j < 100; j++)
106             counter *= 3;

108         call Semaphore.release(&sem_buffer_cheio);
        }
110     call Semaphore.release(&sem_termino);
    }
```

Ao final do consumo de todos os produtos a thread *SerialSender* é desbloqueada. Ela é responsável por calcular o tempo final de execução, e enviar este valor pela porta serial para um computador. Na linha 60, podemos ver a utilização da chamada de sistema responsável por enviar este valor.

```
    event void SerialSender.run(void* arg)
48  {
        message_t msg;
50
        call Semaphore.acquire(&sem_termino);
52
        t1 = call Timer.get() - t1;
54
        while( call AMControl.start() != SUCCESS );
56
        tempo = call Packet.getPayload(&msg, sizeof(uint32_t));
58      (*tempo) = t1;

60      while( call BlockingAMSend.send(AM_BROADCAST_ADDR,
                    &msg, sizeof(uint32_t)) != SUCCESS );
62
        //Para conferir a corretude da execucao
64      (*tempo) = buffer;
        while( call BlockingAMSend.send(AM_BROADCAST_ADDR,
66                  &msg, sizeof(uint32_t)) != SUCCESS );

68      call Leds.led1Toggle();
    }
```

Na configuração, é preciso declarar os componentes responsáveis pelas threads, além de definir o tamanho de suas pilhas.

```
1 configuration BenchmarkAppC{
  }
3 implementation {
    components MainC, BenchmarkC,  LedsC;
5   components new ThreadC(800) as Produtor;
    components new ThreadC(800) as Consumidor;
7   components new ThreadC(800) as SerialSender;
```

```
    BenchmarkC.Produtor -> Produtor;
22  BenchmarkC.Consumidor -> Consumidor;
    BenchmarkC.SerialSender -> SerialSender;
```

O mesmo deve ser feito para as primitivas de gerência de concorrência.

```
    components MutexC;
28  BenchmarkC.Mutex -> MutexC;

30  components SemaphoreC;
```

```
    BenchmarkC.Semaphore -> SemaphoreC;
32 }
```

4.2.2 Implementação

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[2]:

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

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.

```
  event void ThreadSchedulerBoot.booted() {
2   num_runnable_threads = 0;
    //Pega as informacoes da thread principal, seu ID.
4   tos_thread = call ThreadInfo.get[TOSTHREAD_TOS_THREAD_ID]();
    tos_thread->id = TOSTHREAD_TOS_THREAD_ID;
```

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

```
6       //Insere a thread principal na fila de threads prontas.
        call ThreadQueue.init(&ready_queue);
8
        current_thread = tos_thread;
10      current_thread->state = TOSTHREAD_STATE_ACTIVE;
        current_thread->init_block = NULL;
12      signal TinyOSBoot.booted();
    }
```

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() {
       atomic {
3          //Inicializa hardware
           platform_bootstrap();
5          call TaskScheduler.init();
           call PlatformInit.init();
7          //Executa tarefas postas pela funcao a cima
           while (call TaskScheduler.runNextTask());
9          call SoftwareInit.init();
           //Executa tarefas postas pela funcao a cima
11         while (call TaskScheduler.runNextTask());
       }
13     __nesc_enable_interrupt();
       //Sinaliza boot para o usuario
15     signal Boot.booted();
       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() {
       for (;;) {
3          uint8_t nextTask;

5          atomic {
               while((nextTask = popTask()) == NO_TASK) {
7                  call ThreadScheduler.suspendCurrentThread();
               }
9          }
           signal TaskBasic.runTask[nextTask]();
11     }
   }
```

***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.

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

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.

```
struct thread {
volatile struct thread* next_thread;
    //Pointer to next thread for use in queues when blocked
thread_id_t id;
    //id of this thread for use by the thread scheduler
init_block_t* init_block;
    //Pointer to an initialization block from which this thread was spawned
stack_ptr_t stack_ptr;
    //Pointer to this threads stack
volatile uint8_t state;
    //Current state the thread is in
volatile uint8_t mutex_count;
    //A reference count of the number of mutexes held by this thread
uint8_t joinedOnMe[(TOSTHREAD_MAX_NUM_THREADS - 1) / 8 + 1];
    //Bitmask of threads waiting for me to finish
void (*start_ptr)(void*);
    //Pointer to the start function of this thread
void* start_arg_ptr;
    //Pointer to the argument passed as a parameter to the start
    //function of this thread
syscall_t* syscall;
    //Pointer to an instance of a system call
thread_regs_t regs;
    //Contents of the GPRs stored when doing a context switch
};
```

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

```
struct syscall {
struct syscall* next_call;
```

```
3     //Pointer to next system call for use in syscall queues when
      //blocking on them
5 syscall_id_t id;
      //client id of this system call for the particular syscall_queue
7     //within which it is being held
  thread_t* thread;
9     //Pointer back to the thread with which this system call is associated
  void (*syscall_ptr)(struct syscall*);
11    //Pointer to the the function that actually performs the system call
  void* params;
13    //Pointer to a set of parameters passed to the system call once it is
      //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.

```
1 interface Thread {
    command error_t start(void* arg);
3   command error_t stop();
    command error_t pause();
5   command error_t resume();
    command error_t sleep(uint32_t milli);
7   event void run(void* arg);
    command error_t join();
9 }
```

***interfaces/ThreadInfo.nc*:** Contém comandos para receber ou apagar as informações da thread, vistas em 4.2.2.

```
1 interface ThreadInfo {
    async command error_t reset();
3   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 {
2   //Comandos para obter informacoes de uma thread
    async command uint8_t currentThreadId();
4   async command thread_t* currentThreadInfo();
    async command thread_t* threadInfo(thread_id_t id);
6
```

```
    //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();
    async command error_t interruptCurrentThread();
    async command error_t wakeupThread(thread_id_t id);
    async command error_t joinThread(thread_id_t id);
}
```

***system/ThreadInfoP.nc*:** Contém o vetor que representa a pilha, as informações da thread, como visto em 4.2.2 e a função que sinaliza a execução.

```
generic module ThreadInfoP(uint16_t stack_size, uint8_t thread_id) {
provides {
    interface Init; // Para Inicializar as informacoes
    interface ThreadInfo; // Para exportar as Informacoes da thread
    interface ThreadFunction; // Sinaliza o evento responsavel
                              //por executar a thread
}}

implementation {
  uint8_t stack[stack_size];
  thread_t thread_info;

  void run_thread(void* arg) __attribute__((noinline)) {
    signal ThreadFunction.signalThreadRun(arg);
  }

  error_t init() {
    thread_info.next_thread = NULL;
    thread_info.id = thread_id;
    thread_info.init_block = NULL;
    thread_info.stack_ptr = (stack_ptr_t)(STACK_TOP(stack, sizeof(stack)));
    thread_info.state = TOSTHREAD_STATE_INACTIVE;
    thread_info.mutex_count = 0;
    thread_info.start_ptr = run_thread;
    thread_info.start_arg_ptr = NULL;
    thread_info.syscall = NULL;
    return SUCCESS;
  }

  ...
```

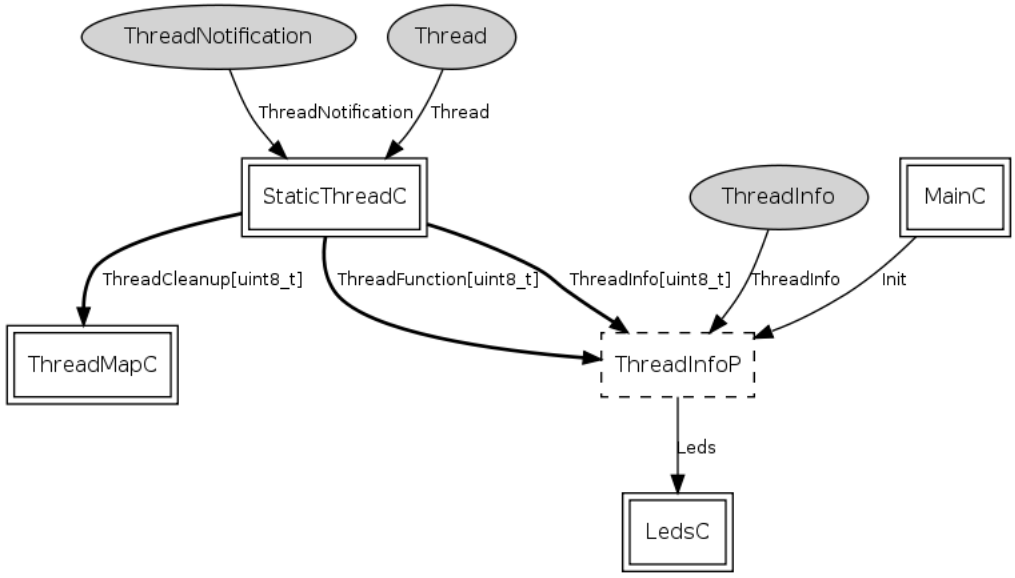
```
}
```

***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*.

```
module StaticThreadP.nc { ... }
implementation {

error_t init(uint8_t id, void* arg) {
    error_t r1, r2;
    thread_t* thread_info = call ThreadInfo.get[id]();
    thread_info->start_arg_ptr = arg;
    thread_info->mutex_count = 0;
    thread_info->next_thread = NULL;
    r1 = call ThreadInfo.reset[id]();
    r2 = call ThreadScheduler.initThread(id);
    return ecombine(r1, r2);
}

event void ThreadFunction.signalThreadRun[uint8_t id](void *arg) {
    signal Thread.run[id](arg);
}

command error_t Thread.start[uint8_t id](void* arg) {
    atomic {
        if( init(id, arg) == SUCCESS ) {
            error_t e = call ThreadScheduler.startThread(id);
            if(e == SUCCESS)
                signal ThreadNotification.justCreated[id]();
            return e;
        }
    }
    return FAIL;

    ... Continuacao da implementacao da interface thread ...
    ... Todos os comandos sao simplesmente passados para o ...
    ... equivalente no ThreadScheduler ...
}
```

***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/Thread-*

InfoP.nc como visto em 4.2.2, e *system/StaticThreadC.nc* como visto em 4.2.2. 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*.

```
typedef struct thread_regs {
    uint8_t status;
    uint8_t r0;
    ...
    uint8_t r31;
} thread_regs_t;
```

Existem também algumas macros para salvar e restaurar estes registradores.

```
#define SAVE_STATUS(t)                                          \
    __asm__("in %0,__SREG__ \n\t" : "=r" ((t)->regs.status) : );

//Save General Purpose Registers
#define SAVE_GPR(t)                                             \
    __asm__("mov %0,r0 \n\t" : "=r" ((t)->regs.r0) : );   \
    ...

//Save stack pointer
#define SAVE_STACK_PTR(t)                \
    __asm__("in %A0, __SP_L__\n\t"       \
    "in %B0, __SP_H__\n\t"               \
    :"=r"((t)->stack_ptr) : );

#define SAVE_TCB(t) \
   SAVE_GPR(t);       \
```

```
    SAVE_STATUS(t);    \
18  SAVE_STACK_PTR(t)

20 //Definicao das macros de restauracao
   ...
22

#define SWITCH_CONTEXTS(from, to) \
24  SAVE_TCB(from);                \
    RESTORE_TCB(to)
```

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

```
1  #define SWAP_STACK_PTR(OLD, NEW) \
     __asm__("in %A0, __SP_L__\n\t in %B0, __SP_H__":"=r"(OLD):);\
3    __asm__("out __SP_H__,%B0\n\t out __SP_L__,%A0"::"r"(NEW))

5 #define PREPARE_THREAD(t, thread_ptr)                    \
  {
7     uint16_t temp;                                       \
      SWAP_STACK_PTR(temp, (t)->stack_ptr);                \
9     __asm__("push %A0\n push %B0"::"r"(&(thread_ptr)));  \
      SWAP_STACK_PTR((t)->stack_ptr, temp);                \
11    SAVE_STATUS(t)                                       \
  }
```

***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.

```
command error_t ThreadScheduler.initThread(uint8_t id) {
2     thread_t* t = (call ThreadInfo.get[id]());
      t->state = TOSTHREAD_STATE_INACTIVE;
4     t->init_block = current_thread->init_block;
      call BitArrayUtils.clrArray(t->joinedOnMe, sizeof(t->joinedOnMe));
6     PREPARE_THREAD(t, threadWrapper);
          //O codigo abaixo e' definicao da macro PREPARE_THREAD,
8         //inserido aqui para facilitar o entendimento do codigo.
          //uint16_t temp;                                       \
10        //SWAP_STACK_PTR(temp, (t)->stack_ptr);                \
          //__asm__("push %A0\n push %B0"::"r"(&(threadWrapper)));   \
12        //SWAP_STACK_PTR((t)->stack_ptr, temp);                \
          //SAVE_STATUS(t)
14    return SUCCESS;
  }
```

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.

```
void threadWrapper() __attribute__((naked, noinline)) {
    thread_t* t;
    atomic t = current_thread;

    __nesc_enable_interrupt();
    (*(t->start_ptr))(t->start_arg_ptr);

    atomic {
        stop(t);
        sleepWhileIdle();
        scheduleNextThread();
        restoreThread();
    }
}
```

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.

```
async command error_t ThreadScheduler.suspendCurrentThread() {
    atomic {
        if(current_thread->state == TOSTHREAD_STATE_ACTIVE) {
            current_thread->state = TOSTHREAD_STATE_SUSPENDED;
            //suspend(current_thread);
            #ifdef TOSTHREADS_TIMER_OPTIMIZATION
                num_runnable_threads--;
                post alarmTask();
            #endif
            sleepWhileIdle();
            //interrupt(current_thread);
            yielding_thread = current_thread;
            //scheduleNextThread();
            if(tos_thread->state == TOSTHREAD_STATE_READY)
                current_thread = tos_thread;
            else
                current_thread = call ThreadQueue.dequeue(&ready_queue);

            current_thread->state = TOSTHREAD_STATE_ACTIVE;
            //fim scheduleNextThread();

            if(current_thread != yielding_thread) {
                //switchThreads();
                void switchThreads() __attribute__((noinline)) {
                    SWITCH_CONTEXTS(yielding_thread, current_thread);
                 }
                //fim switchThreads();
```

```
28              }
                //fim interrupt(...)
30              //fim suspend(current_thread);
                return SUCCESS;
32          }
            return FAIL;
34      }
}
```

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 termino 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[3], 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
   command error_t BlockingReceive.receive[uint8_t am_id](message_t* m,
3                                                          uint32_t timeout) {
       syscall_t s;
5      params_t p;

7      atomic {
           if((blockForAny == TRUE) ||
9             (call SystemCallQueue.find(&am_queue, am_id) != NULL))
               return EBUSY;
11         call SystemCallQueue.enqueue(&am_queue, &s);
       }
13
       p.msg = m;
15     p.timeout = timeout;
       atomic {
```

[3]!!!Descrever mensagens ativas? !!!

```
17          p.error = EBUSY;
            if(timeout != 0)
19              call SystemCall.start(&timerTask, &s, am_id, &p);
            else
21              call SystemCall.start(SYSCALL_WAIT_ON_EVENT, &s, am_id, &p);
        }
23
        atomic {
25          call SystemCallQueue.remove(&am_queue, &s);
            return p.error;
27      }
    }
```

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,
2                                syscall_t* s,
                                 syscall_id_t id,
4                                void* p) {
    atomic {
6
        current_call = s;
8       current_call->id = id;
        current_call->thread = call ThreadScheduler.currentThreadInfo();
10      current_call->thread->syscall = s;
        current_call->params = p;
12
        if(syscall_ptr != SYSCALL_WAIT_ON_EVENT) {
14          current_call->syscall_ptr = syscall_ptr;
            post threadTask();
16          call ThreadScheduler.wakeupThread(TOSTHREAD_TOS_THREAD_ID);
        }
18
        return call ThreadScheduler.suspendCurrentThread();
20  }
}
```

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

```
1 //Temporizador responsavel por calcular o timeout.
  void timerTask(syscall_t* s) {
3     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 termino 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);
```

```
2
command error_t BlockingReceive.receive[uint8_t am_id](message_t* m,
4                                                      uint32_t timeout) {
    syscall_t s;
6   params_t p;
    //...
8   call SystemCallQueue.enqueue(&am_queue, &s);
    //...
10  p.msg = m;
    //...
12  call SystemCall.start(SYSCALL_WAIT_ON_EVENT, &s,am_id ,&p) ;
}
14
command error_t SystemCall.start(void* syscall_ptr,
16                                 syscall_t* s,
                                   syscall_id_t id,
18                                 void* p) {
    current_call = s;
20  //...
    current_call->params = p;
22  //...
}
24
event message_t* Receive.receive[uint8_t am_id](message_t* m,
26                                              void* payload,
                                                uint8_t len) {
28  syscall_t* s;
    params_t* p;
30
    //...
32  s = call SystemCallQueue.find(&am_queue, am_id);
    //...
34  p = s->params;
    //...
36  *(p->msg) = *m;
}
```

4.3 Co-rotinas para o TinyOS

O modelo que decidimos implementar foi um descrito por Ana Moura em sua tese de doutorado[7, s. 6.2]. Neste modelo existe uma co-rotina principal que é responsável por escalonar as outras co-rotinas.

4.3.1 Exemplo de aplicação produtor/consumidor

Nesta seção ilustraremos o uso de co-rotinas no TinyOS, por meio de uma aplicação que utiliza o modelo produtor/consumidor.

Ao codificar o módulo principal, é preciso definir co-rotinas serão utilizadas.

```
module BenchmarkC {
  uses {
    interface Boot;
    interface Thread as Produtor;
    interface Thread as Consumidor;
    interface Thread as SerialSender;
    interface Leds;
```

O próximo passo é ativar as primeiras co-rotinas

```
    event void Boot.booted() {
      buffer = 0;
      t1 = call Timer.get();

      call Produtor.start(NULL);
      call Consumidor.start(NULL);
    }
```

A seguir a co-rotina responsável por criar os produtos. Aqui podemos ver como é feito o uso do comando *yield()*, responsável por ceder o controle.

```
    event void Produtor.run(void* arg) {
        uint16_t counter = 1;
        uint16_t num_prods, j;

        for (num_prods = 0; num_prods < 1000; num_prods++)
        {
            //Tempo simulado para criar um produto
            for (j = 0; j < 100; j++)
                counter *= 3;

            buffer = counter;

            call Produtor.yield();
        }
    }
```

A co-rotina consumidora também é a responsável por ativar a co-rotina que terminará de calcular o tempo de execução de todo o programa. É importante notar que a co-rotina *SerialSender* não será executada imediatamente. Ela entrará em uma fila, e será executada quando for escalonada. Obedecendo o modelo adodato (**??**).

```
    event void Consumidor.run(void* arg) {
        uint16_t num_prods, j;
        uint16_t counter = 0;

        for (num_prods = 0; num_prods < 1000; num_prods++)
        {
            counter = buffer;
```

```
78              //Tempo simulado para consumir produto
                for (j = 0; j < 100; j++)
80                  counter *= 3;

82              call Consumidor.yield();
            }
84          call SerialSender.start(NULL);
        }
```

SerialSender é responsável por calcular o tempo final de execução, e enviar este valor pela porta serial para um computador. Repare que as mesmas chamadas de sistema utilizadas para threads, são utilizadas aqui.

```
        }
32
        event void SerialSender.run(void* arg)
34      {
            message_t msg;
36
            t1 = call Timer.get() - t1;
38
            while( call AMControl.start() != SUCCESS );
40
            tempo = call Packet.getPayload(&msg, sizeof(uint32_t));
42          (*tempo) = t1;

44          while( call BlockingAMSend.send(AM_BROADCAST_ADDR,
                        &msg, sizeof(uint32_t)) != SUCCESS );
46
            //Para conferir a corretude da execucao
48          (*tempo) = buffer;
            while( call BlockingAMSend.send(AM_BROADCAST_ADDR,
50                      &msg, sizeof(uint32_t)) != SUCCESS );
        }
```

Na configuração, é preciso declarar os componentes responsáveis pelas co-rotinas, além de definir o tamanho de suas pilhas.

```
1   configuration BenchmarkAppC{
    }
3   implementation {
      components MainC, BenchmarkC,  LedsC;
5     components new ThreadC(800) as Produtor;
      components new ThreadC(800) as Consumidor;
7     components new ThreadC(800) as SerialSender;
```

```
      BenchmarkC.Produtor -> Produtor;
22    BenchmarkC.Consumidor -> Consumidor;
      BenchmarkC.SerialSender -> SerialSender;
```

4.3.2 Implementação de co-rotinas para o TinyOS

Nossa implementação utilizou como base a extensão *TOSThreads*, vista na seção 4.2. O primeiro passo foi criar uma cópia do diretório desta extensão e um novo *target* referente a este diretório para o *make* do TinyOS.

Na implementação do *TOSThreads* existem dois casos em que ocorre preempção: termino do *timeslice* e acontecimento de uma interrupção de hardware. Portanto foi preciso modificar esses dois casos. A primeira alteração foi retirar o limite de tempo de execução de cada thread. Para isso o temporizador responsável por essa contagem foi desabilitado. A segunda alteração foi criar um novo tipo de interrupção, que chamamos de interrupção curta. Originalmente, no *TOSThreads*, quando o tratador de interrupção postava uma tarefa, o kernel assumia o controle, executava a tarefa e escalonava a próxima thread da fila. Na nossa implementação, após o kernel executar a tarefa, a thread que foi originalmente interrompida volta a executar. Para isso, foi criado um novo comando no escalonador de threads: *brieflyInterruptCurrentThread()*

```
async command error_t ThreadScheduler.brieflyInterruptCurrentThread() {
    atomic {
        if(current_thread->state == TOSTHREAD_STATE_ACTIVE) {
            briefly_interrupted_thread = current_thread;
            briefly_interrupted_thread->state =
                TOSTHREAD_STATE_BRIEFLYINTERRUPTED;
            interrupt(current_thread);
            return SUCCESS;
        }
        return FAIL;
    }
}

/* schedule_next_thread()
 * This routine does the job of deciding which thread should run next.
 * Should be complete as is.  Add functionality to getNextThreadId()
 * if you need to change the actual scheduling policy.
 */
    void scheduleNextThread() {
        if(tos_thread->state == TOSTHREAD_STATE_READY)
            current_thread = tos_thread;
        else if (briefly_interrupted_thread != NULL)
        {
            current_thread = briefly_interrupted_thread;
            briefly_interrupted_thread = NULL;
        }
        else
            current_thread = call ThreadQueue.dequeue(&ready_queue);

```

```
        current_thread->state = TOSTHREAD_STATE_ACTIVE;
31  }
```

Uma vez excluída a preempção, o próximo passo foi modificar a interface da thread para permitir passagem de controle ao escalonador. Para isso, foi criado o comando *yield()*.

Listing 8: Arquivo interfaces/Thread.nc

```
interface Thread {
2     ...
    command error_t yield();
4     ...
}
6
//Arquivo: system/StaticThreadP.nc
8 module StaticThreadP {
    ...
10    command error_t Thread.yield[uint8_t id]() {
        return call ThreadScheduler.interruptCurrentThread();
12    }
    ...
14 }
```

4.4 Experimentos e resultados obtidos

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* (anexo B.5), e outra utilizando co-rotinas (anexo B.6). Foram utilizados uma linha de execução para o produtor, e outra para o consumidor, e um *buffer* de tamanho único. 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 mil produtos, uma nova linha de execução é ativada, para calcular o tempo de execução.

O tempo de execução foi medido em uma plataforma *MicaZ*, utilizando o temporizador *Counter<TMicro,uint32_t>*, utilizando uma precisão de microsegundos. Os valores medidos não variaram mais de uma unidade entre as diferentes execuções e cada cenário foi executado dez vezes.

No primeiro experimento, variamos a quantidade de produto, referente ao laço:

```
for (num_prods = 0 ; num_prods < 1000; num_prods++)
```

No segundo experimentos, variamos a quantidade de operações realizadas para simular a produção e o consumo, referente ao laço:

```
1 for (j = 0 ; j < 100; j++)
    counter *= 3;
```

Na figura 5 podemos ver um gráfico com os resultados.

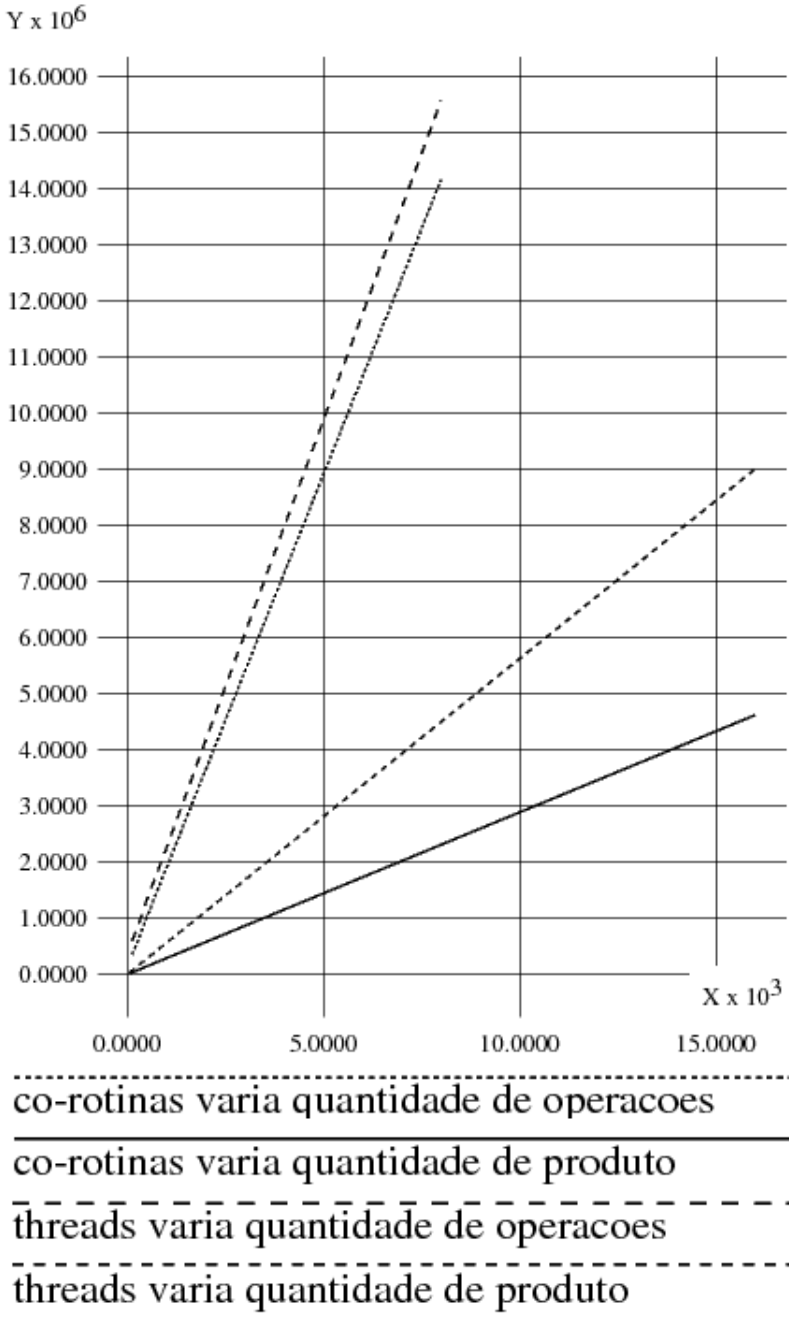


Figura 5: Gráfico dos experimentos de threads e co-rotinas

5 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[8] facilita este 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 Trabalhos Futuros

7 Apêndice

A Extensão para o Simulador TOSSIM

Para que o escalonador funcione corretamente no simulador TOSSIM é preciso adicionar funções que lidam com eventos no simulador. Essas funções foram retiradas do arquivo *tinyOS-root-dir/tos/lib/tossim/SimSchedulerBasicP.nc*. Primeiro é preciso alterar a implementação das interfaces de tarefa e da interface *Scheduler*, criando as funções e variáveis do código abaixo:

```
bool sim_scheduler_event_pending = FALSE;
sim_event_t sim_scheduler_event;

int sim_config_task_latency() {return 100;}

void sim_scheduler_submit_event() {
  if (sim_scheduler_event_pending == FALSE) {
    sim_scheduler_event.time = sim_time() + sim_config_task_latency();
    sim_queue_insert(&sim_scheduler_event);
    sim_scheduler_event_pending = TRUE;
  }
```

```
12  }

14  void sim_scheduler_event_handle(sim_event_t* e) {
      sim_scheduler_event_pending = FALSE;
16    if (call Scheduler.runNextTask())
        sim_scheduler_submit_event();
18  }

20  void sim_scheduler_event_init(sim_event_t* e) {
      e->mote = sim_node();
22    e->force = 0;
      e->data = NULL;
24    e->handle = sim_scheduler_event_handle;
      e->cleanup = sim_queue_cleanup_none;
26  }
```

Depois, no comando *init()* deve-se adicionar:

```
  sim_scheduler_event_pending = FALSE;
2 sim_scheduler_event_init(&sim_scheduler_event);
```

E, por último, nos comandos *postTask()*, deve-se adicionar:

```
  sim_scheduler_submit_event();
```

B Anexos

B.1 Blink

B.1.1 BlinkC.nc:

```
1 #include "Timer.h"

3 module BlinkC @safe()
  {
5   uses interface Timer<TMilli> as Timer0;
    uses interface Timer<TMilli> as Timer1;
7   uses interface Timer<TMilli> as Timer2;
    uses interface Leds;
9   uses interface Boot;
  }
11 implementation
  {
13  event void Boot.booted()
    {
15    call Timer0.startPeriodic( 250 );
      call Timer1.startPeriodic( 500 );
17    call Timer2.startPeriodic( 1000 );
```

```
19      post tarefa();
    }
21
    event void Timer0.fired()
23  {
      call Leds.led0Toggle();
25  }

27  event void Timer1.fired()
    {
29    call Leds.led1Toggle();
    }
31
    event void Timer2.fired()
33  {
      call Leds.led2Toggle();
35  }

37  task void tarefa()
    {
39      dbg("BlinkC", "tarefa\n");
    }
41 }
```

B.1.2 BlinkAppC.nc:

```
1  configuration BlinkAppC
   {}
3  implementation
   {
5    components MainC, BlinkC, LedsC;
     components new TimerMilliC() as Timer0;
7    components new TimerMilliC() as Timer1;
     components new TimerMilliC() as Timer2;
9
     BlinkC.Boot -> MainC.Boot;
11   BlinkC.Timer0 -> Timer0;
     BlinkC.Timer1 -> Timer1;
13   BlinkC.Timer2 -> Timer2;
     BlinkC.Leds -> LedsC.Leds;
15 }
```

B.2 Aplicação de Teste do Escalonador Padrão

B.2.1 aplicacaoTesteC.nc:

```
2  /**
    * Implementa aplicativo de teste do Scheduler de prioridade
4   **/

6  #include "Timer.h"
   #include "printf.h"
8
   module aplicacaoTesteC @safe()
10 {
       uses interface Boot;
12     uses interface Leds;

14     uses interface Counter<TMicro, uint32_t> as Timer1;
   }
16 implementation
   {
18     /* Variaveis */
       unsigned int t1;
20     bool over;

22     async event void Timer1.overflow()
       {
24         over = TRUE;
       }
26


28     /* Tarefas
       */
30     task void Tarefa1()
       {
32         uint16_t i = 0;
           uint16_t k = 1;
34         for (i = 0; i < 65000; i++)
           {
36             k = k * 2;
           }
38         t1 = call Timer1.get();
           printf("tempo final: %u\n", t1);
40         if (over == TRUE)
               printf("OVERFLOW!!\n");
42         printfflush();
       }
44     task void Tarefa2()
       {
46         uint16_t i = 0;
           uint16_t k = 1;
48         for (i = 0; i < 65000; i++)
```

```
        {
50          k = k * 2;
        }
52  }

54  // ...

56  task void Tarefa99()
    {
58      uint16_t i = 0;
        uint16_t k = 1;
60      for (i = 0; i < 65000; i++)
        {
62          k = k * 2;
        }
64  }

66  /* Boot
    */
68  event void Boot.booted()
    {
70      over = FALSE;
        t1 = call Timer1.get();
72      printf("tempo inicial: %u\n", t1);
        printfflush();
74
        post Tarefa2();
76      post Tarefa3();
        // ...
78      post Tarefa99();

80      post Tarefa1();
    }
82  }
```

B.2.2 aplicacaoTesteAppC.nc:

```
/**
2  * Aplicativo de teste do Scheduler de prioridade
 *
4  **/

6 #include "MsgSerial.h"
  #include "Timer.h"
8 #include "printf.h"

10 configuration aplicacaoTesteAppC
```

```
{
}
implementation
{
  components MainC, aplicacaoTesteC, LedsC, TinySchedulerC;
  components CounterMicro32C as Timer1;

  aplicacaoTesteC.Timer1 -> Timer1;

  aplicacaoTesteC-> MainC.Boot;
  aplicacaoTesteC.Leds -> LedsC;

  aplicacaoTesteC.Tarefa1->
  TinySchedulerC.TaskPrioridade[unique("TinySchedulerC.TaskPrioridade")];
  aplicacaoTesteC.Tarefa2->
  TinySchedulerC.TaskPrioridade[unique("TinySchedulerC.TaskPrioridade")];
  aplicacaoTesteC.Tarefa3->
  TinySchedulerC.TaskPrioridade[unique("TinySchedulerC.TaskPrioridade")];
  aplicacaoTesteC.Tarefa4->
  TinySchedulerC.TaskPrioridade[unique("TinySchedulerC.TaskPrioridade")];
  aplicacaoTesteC.Tarefa5->
  TinySchedulerC.TaskPrioridade[unique("TinySchedulerC.TaskPrioridade")];

// ...

  aplicacaoTesteC.Tarefa98->
  TinySchedulerC.TaskPrioridade[unique("TinySchedulerC.TaskPrioridade")];
  aplicacaoTesteC.Tarefa99->
  TinySchedulerC.TaskPrioridade[unique("TinySchedulerC.TaskPrioridade")];
}
```

B.3 Aplicação de Teste do Escalonador com Prioridades

B.3.1 aplicacaoTesteC.nc:

```

/**
 * Implementa aplicativo de teste do Scheduler de prioridade
 **/

#include "MsgSerial.h"
#include "Timer.h"
#include "printf.h"

module aplicacaoTesteC @safe()
{
    uses interface Boot;
```

```
13      uses interface Leds;
        uses interface TaskPrioridade as Tarefa1;
15      uses interface TaskPrioridade as Tarefa2;
        uses interface TaskPrioridade as Tarefa3;
17      uses interface TaskPrioridade as Tarefa4;
        uses interface TaskPrioridade as Tarefa5;
19      // ...
        uses interface TaskPrioridade as Tarefa98;
21      uses interface TaskPrioridade as Tarefa99;

23      uses interface Counter<TMicro, uint32_t> as Timer1;
    }
25  implementation
    {
27      /* Variaveis */
        unsigned int t1;
29      bool over;

31      async event void Timer1.overflow()
        {
33          over = TRUE;
        }
35
        /* Boot
37      */
        event void Boot.booted()
39      {
            over = FALSE;
41          t1 = call Timer1.get();
            printf("tempo inicial: %u\n", t1);
43          printfflush();

45          call Tarefa1.postTask(20);

47          call Tarefa2.postTask(10);
            call Tarefa3.postTask(10);
49          // ...
            call Tarefa98.postTask(10);
51          call Tarefa99.postTask(10);
        }
53
        /* Tarefas
55      */
        event void Tarefa1.runTask()
57      {
            uint16_t i = 0;
59          uint16_t k = 1;
```

```
        for (i = 0; i < 65000; i++)
        {
            k = k * 2;
        }
        // Calculo do tempo de execucao
        t1 = call Timer1.get();
        printf("tempo final: %u\n", t1);
        if (over == TRUE)
            printf("Ocorreu Overflow\n");
        printfflush();
    }
    event void Tarefa2.runTask()
    {
        uint16_t i = 0;
        uint16_t k = 1;
        for (i = 0; i < 65000; i++)
        {
            k = k * 2;
        }
    }

    // ...

    event void Tarefa99.runTask()
    {
        uint16_t i = 0;
        uint16_t k = 1;
        for (i = 0; i < 65000; i++)
        {
            k = k * 2;
        }
    }
}
```

B.3.2 aplicacaoTesteAppC.nc:

```
/**
 * Aplicativo de teste do Scheduler de prioridade
 *
 **/

#include "MsgSerial.h"
#include "Timer.h"
#include "printf.h"

configuration aplicacaoTesteAppC
{
```

```
12 }
   implementation
14 {
     components MainC, aplicacaoTesteC, LedsC, TinySchedulerC;
16   components CounterMicro32C as Timer1;

18   aplicacaoTesteC.Timer1 -> Timer1;

20   aplicacaoTesteC-> MainC.Boot;
     aplicacaoTesteC.Leds -> LedsC;
22
     aplicacaoTesteC.Tarefa1->
24   TinySchedulerC.TaskPrioridade[unique("TinySchedulerC.TaskPrioridade")];
     aplicacaoTesteC.Tarefa2->
26   TinySchedulerC.TaskPrioridade[unique("TinySchedulerC.TaskPrioridade")];
     aplicacaoTesteC.Tarefa3->
28   TinySchedulerC.TaskPrioridade[unique("TinySchedulerC.TaskPrioridade")];
     aplicacaoTesteC.Tarefa4->
30   TinySchedulerC.TaskPrioridade[unique("TinySchedulerC.TaskPrioridade")];
     aplicacaoTesteC.Tarefa5->
32   TinySchedulerC.TaskPrioridade[unique("TinySchedulerC.TaskPrioridade")];

34 // ...

36   aplicacaoTesteC.Tarefa98->
     TinySchedulerC.TaskPrioridade[unique("TinySchedulerC.TaskPrioridade")];
38   aplicacaoTesteC.Tarefa99->
     TinySchedulerC.TaskPrioridade[unique("TinySchedulerC.TaskPrioridade")];
40 }
```

B.4 Aplicação de Teste do Escalonador Multi-nível

B.4.1 aplicacaoTesteC.nc:

```
1  #include "Timer.h"

3  module aplicacaoTesteC @safe()
   {
5      uses interface Boot;
       uses interface Leds;
7      uses interface TaskSerial as TarefaSerial;
       uses interface TaskRadio as TarefaRadio;
9      uses interface TaskSense as TarefaSense;
       uses interface Timer<TMilli>;
11
       uses interface Read<uint16_t>;
13
```

```
    uses interface Packet;
    uses interface AMSend;
    uses interface SplitControl as RadioControl;

    uses interface SerialPacket;
    uses interface SerialAMSend;
    uses interface SerialSplitControl as SerialControl;
}
implementation
{
    /* Variaveis */
    unsigned int t1;
    uint16_t valorLido;
    message_t packet;
    message_t serialPacket;

    /* Tarefas
    */
    task void TarefaBasic()
    {
        printf("tarefa Basic\n");
        printfflush();
    }

    event void TarefaSense.runTask()
    {
        call Read.read();
    }

    event void TarefaRadio.runTask()
    {
        uint16_t *msg;

        msg = call Packet.getPayload(&packet, sizeof(uint16_t));

        (*msg) = valorLido;
        call AMSend.send(AM_BROADCAST_ADDR, &packet, sizeof(uint16_t));
    }

    event void TarefaSerial.runTask()
    {
        uint16_t *msg;

        msg = call Packet.getPayload(&serialPacket, sizeof(uint16_t));

        (*msg) = valorLido;
        call AMSend.send(AM_BROADCAST_ADDR, &packet, sizeof(uint16_t));
```

```
61      }
63      /* events */
        event void Boot.booted()
65      {
            call RadioControl.start();
67          call SerialControl.start();
            call Timer.startPeriodic(250);
69      }
71      event void RadioControl.startDone(error_t err) {
            if (err == SUCCESS)
73              radioOn = 1;
        }
75      event void RadioControl.stopDone(error_t err) {}
77      event void SerialControl.startDone(error_t err) {
            if (err = SUCCESS)
79              serialOn = 1;
        }
81      event void SerialControl.stopDone(error_t err) {}
83      event void Timer.fired()
        {
85          call TarefaSense.postTask();
        }
87
        event void Read.readDone(error_t result, uint16_t data)
89      {
            valorLido = data;
91          if (radioOn) call TarefaRadio.postTask();
            if (serialOn) call TarefaSerial.postTask();
93      }
95  }
```

B.4.2 aplicacaoTesteAppC.nc:

```
1  #include "Timer.h"
3  configuration aplicacaoTesteAppC
   {
5  }
   implementation
7  {
     components MainC, aplicacaoTesteC, LedsC, TinySchedulerC;
9    components new DemoSensorC() as Sensor;
```

```
    components new TimerMilliC() as Timer;
11
    components ActiveMessageC;
13  components new AMSenderC(1);

15  components SerialActiveMessageC;
    components new SerialAMSenderC(2);
17
    aplicacaoTesteC.Timer -> Timer;
19
    aplicacaoTesteC-> MainC.Boot;
21  aplicacaoTesteC.Leds -> LedsC;

23  aplicacaoTesteC.Read -> Sensor;

25  aplicacaoTesteC.AMSend -> AMSenderC;
    aplicacaoTesteC.RadioControl -> ActiveMessageC;
27  aplicacaoTesteC.Packet -> AMSenderC;

29  BlinkToRadioC.SerialAMPacket -> SerialAMSenderC;
    BlinkToRadioC.SerialAMSend -> SerialAMSenderC;
31  BlinkToRadioC.SerialSplitControl -> SerialActiveMessageC;

33  aplicacaoTesteC.TarefaSense->
    TinySchedulerC.TaskSense[unique("TinySchedulerC.TaskSense")];
35  aplicacaoTesteC.TarefaRadio->
    TinySchedulerC.TaskRadio[unique("TinySchedulerC.TaskRadio")];
37  aplicacaoTesteC.TarefaSerial->
    TinySchedulerC.TaskSerial[unique("TinySchedulerC.TaskSerial")];
39 }
```

B.5 Aplicação de Teste de Threads

B.5.1 BenchmarkAppC.nc:

```
configuration BenchmarkAppC{
2 }
implementation {
4   components MainC, BenchmarkC,  LedsC;
    components new ThreadC(800) as Produtor;
6   components new ThreadC(800) as Consumidor;
    components new ThreadC(800) as SerialSender;
8
    components CounterMicro32C as Timer;
10  BenchmarkC.Timer -> Timer;


12
```

```
    components BlockingSerialActiveMessageC;
14  components new BlockingSerialAMSenderC(228);
    BenchmarkC.AMControl -> BlockingSerialActiveMessageC;
16  BenchmarkC.BlockingAMSend -> BlockingSerialAMSenderC;
    BenchmarkC.Packet -> BlockingSerialAMSenderC;
18
    MainC.Boot <- BenchmarkC;
20
    BenchmarkC.Produtor -> Produtor;
22  BenchmarkC.Consumidor -> Consumidor;
    BenchmarkC.SerialSender -> SerialSender;
24
    BenchmarkC.Leds -> LedsC;
26
    components MutexC;
28  BenchmarkC.Mutex -> MutexC;

30  components SemaphoreC;
    BenchmarkC.Semaphore -> SemaphoreC;
32 }
```

B.5.2 BenchmarkC.nc:

```
1  #include "mutex.h"
   #include "semaphore.h"
3
   module BenchmarkC {
5    uses {
       interface Boot;
7      interface Thread as Produtor;
       interface Thread as Consumidor;
9      interface Thread as SerialSender;
       interface Leds;
11
       interface BlockingStdControl as AMControl;
13     interface BlockingAMSend;
       interface Packet;
15
       interface Counter<TMicro, uint32_t> as Timer;
17
       interface Mutex;
19     interface Semaphore;
     }
21 }

23 implementation {
       uint32_t t1;
```

```
25      uint32_t * tempo;
        uint8_t buffer;
27      mutex_t mutex_buffer;
        semaphore_t sem_produto, sem_termino, sem_buffer_cheio;
29
        async event void Timer.overflow()
31      {}

33      event void Boot.booted() {
          buffer = 0;
35        t1 = call Timer.get();

37        call Mutex.init(&mutex_buffer);
          call Semaphore.reset(&sem_produto, 0);
39        call Semaphore.reset(&sem_buffer_cheio, 1);
          call Semaphore.reset(&sem_termino, 0);
41
          call Produtor.start(NULL);
43        call Consumidor.start(NULL);
          call SerialSender.start(NULL);
45      }

47      event void SerialSender.run(void* arg)
        {
49          message_t msg;

51          call Semaphore.acquire(&sem_termino);

53          t1 = call Timer.get() - t1;

55          while( call AMControl.start() != SUCCESS );

57          tempo = call Packet.getPayload(&msg, sizeof(uint32_t));
            (*tempo) = t1;
59
            while( call BlockingAMSend.send(AM_BROADCAST_ADDR,
61                          &msg, sizeof(uint32_t)) != SUCCESS );

63          //Para conferir a corretude da execucao
            (*tempo) = buffer;
65          while( call BlockingAMSend.send(AM_BROADCAST_ADDR,
                            &msg, sizeof(uint32_t)) != SUCCESS );
67
            call Leds.led1Toggle();
69      }


71
```

```
    event void Produtor.run(void* arg) {
73      uint16_t counter = 1;
        uint16_t num_prods, j;
75
        for (num_prods = 0; num_prods < 1000; num_prods++)
77      {
            call Semaphore.acquire(&sem_buffer_cheio);
79
            //Tempo simulado para criar um produto
81          for (j = 0; j < 100; j++)
                counter *= 3;
83
            call Mutex.lock(&mutex_buffer);
85              buffer = counter;
            call Mutex.unlock(&mutex_buffer);
87
            call Semaphore.release(&sem_produto);
89      }
    }
91
    event void Consumidor.run(void* arg) {
93      uint16_t num_prods, j;
        uint16_t counter = 0;
95
        for (num_prods = 0; num_prods < 1000; num_prods++)
97      {
            call Semaphore.acquire(&sem_produto);
99
            call Mutex.lock(&mutex_buffer);
101             counter = buffer;
            call Mutex.unlock(&mutex_buffer);
103
            //Tempo simulado para consumir produto
105         for (j = 0; j < 100; j++)
                counter *= 3;
107
            call Semaphore.release(&sem_buffer_cheio);
109     }
        call Semaphore.release(&sem_termino);
111 }

113 }
```

B.6 Aplicação de Teste de Co-rotinas

B.6.1 BenchmarkAppC.nc:

```
configuration BenchmarkAppC{
}
implementation {
  components MainC, BenchmarkC,  LedsC;
  components new ThreadC(800) as Produtor;
  components new ThreadC(800) as Consumidor;
  components new ThreadC(800) as SerialSender;

  components CounterMicro32C as Timer;
  BenchmarkC.Timer -> Timer;


  components BlockingSerialActiveMessageC;
  components new BlockingSerialAMSenderC(228);
  BenchmarkC.AMControl -> BlockingSerialActiveMessageC;
  BenchmarkC.BlockingAMSend -> BlockingSerialAMSenderC;
  BenchmarkC.Packet -> BlockingSerialAMSenderC;

  MainC.Boot <- BenchmarkC;

  BenchmarkC.Produtor -> Produtor;
  BenchmarkC.Consumidor -> Consumidor;
  BenchmarkC.SerialSender -> SerialSender;

  BenchmarkC.Leds -> LedsC;

}
```

B.6.2 BenchmarkC.nc:

```
module BenchmarkC {
  uses {
    interface Boot;
    interface Thread as Produtor;
    interface Thread as Consumidor;
    interface Thread as SerialSender;
    interface Leds;

    interface BlockingStdControl as AMControl;
    interface BlockingAMSend;
    interface Packet;

    interface Counter<TMicro, uint32_t> as Timer;
  }
}

implementation {
```

```
18      uint32_t t1;
        uint32_t * tempo;
20      uint8_t buffer;

22      async event void Timer.overflow()
        {}
24
        event void Boot.booted() {
26        buffer = 0;
          t1 = call Timer.get();
28
          call Produtor.start(NULL);
30        call Consumidor.start(NULL);
        }
32
        event void SerialSender.run(void* arg)
34      {
            message_t msg;
36
            t1 = call Timer.get() - t1;
38
            while( call AMControl.start() != SUCCESS );
40
            tempo = call Packet.getPayload(&msg, sizeof(uint32_t));
42          (*tempo) = t1;

44          while( call BlockingAMSend.send(AM_BROADCAST_ADDR,
                        &msg, sizeof(uint32_t)) != SUCCESS );
46
            //Para conferir a corretude da execucao
48          (*tempo) = buffer;
            while( call BlockingAMSend.send(AM_BROADCAST_ADDR,
50                      &msg, sizeof(uint32_t)) != SUCCESS );
        }
52

54      event void Produtor.run(void* arg) {
            uint16_t counter = 1;
56          uint16_t num_prods, j;

58          for (num_prods = 0; num_prods < 1000; num_prods++)
            {
60              //Tempo simulado para criar um produto
                for (j = 0; j < 100; j++)
62                  counter *= 3;

64              buffer = counter;
```

```
66              call Produtor.yield();
            }
68      }

70      event void Consumidor.run(void* arg) {
            uint16_t num_prods, j;
72          uint16_t counter = 0;

74          for (num_prods = 0; num_prods < 1000; num_prods++)
            {
76              counter = buffer;

78              //Tempo simulado para consumir produto
                for (j = 0; j < 100; j++)
80                  counter *= 3;

82              call Consumidor.yield();
            }
84          call SerialSender.start(NULL);
        }
86
}
```

B.7 Escalonadores

B.7.1 SchedulerDeadlineP.nc:

```
1  // $Id: SchedulerBasicP.nc,v 1.1.2.5 2006/02/14 17:01:46 idgay Exp $

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

```
 * ON AN "AS IS" BASIS, AND THE UNIVERSITY OF CALIFORNIA HAS NO OBLIGATION TO
 * PROVIDE MAINTENANCE, SUPPORT, UPDATES, ENHANCEMENTS, OR MODIFICATIONS."
 *
 * Copyright (c) 2002-2003 Intel Corporation
 * All rights reserved.
 *
 * This file is distributed under the terms in the attached INTEL-LICENSE
 * file. If you do not find these files, copies can be found by writing to
 * Intel Research Berkeley, 2150 Shattuck Avenue, Suite 1300, Berkeley, CA,
 * 94704. Attention: Intel License Inquiry.
 */
/*
 *
 * Authors:   Philip Levis
 * Date last modified:  $Id: SchedulerBasicP.nc,v 1.1.2.5 2006/02/14 17:01:46 idgay Exp $
 *
 */

/**
 * SchedulerBasic implements the default TinyOS scheduler sequence, as
 * documented in TEP 106.
 *
 * @author Philip Levis
 * @author Cory Sharp
 * @date   January 19 2005
 */

#include "hardware.h"

module SchedulerDeadlineP {
    provides interface Scheduler;
    provides interface TaskBasic[uint8_t id];
    provides interface TaskDeadline<TMicro>[uint8_t id];
    uses interface McuSleep;
    uses interface LocalTime<TMicro>;
}
implementation
{
    enum
    {
        NUM_TASKS = uniqueCount("TinySchedulerC.TaskBasic"),
        NUM_DTASKS = uniqueCount("TinySchedulerC.TaskDeadline"),
        NO_TASK = 255,
    };

    volatile uint8_t m_head;
    volatile uint8_t m_tail;
```

```
67      volatile uint8_t m_next[NUM_TASKS];
        volatile uint8_t d_head;
69      volatile uint8_t d_tail;
        volatile uint8_t d_next[NUM_DTASKS];
71      volatile uint32_t d_time[NUM_DTASKS];

73      // Helper functions (internal functions) intentionally do not have atomic
        // sections.  It is left as the duty of the exported interface functions to
75      // manage atomicity to minimize chances for binary code bloat.

77      // move the head forward
        // if the head is at the end, mark the tail at the end, too
79      // mark the task as not in the queue
        inline uint8_t popMTask()
81      {
            if( m_head != NO_TASK )
83          {
                uint8_t id = m_head;
85              m_head = m_next[m_head];
                if( m_head == NO_TASK )
87              {
                    m_tail = NO_TASK;
89              }
                m_next[id] = NO_TASK;
91              return id;
            }
93          else
            {
95              return NO_TASK;
            }
97      }

99      bool isMWaiting( uint8_t id )
        {
101         return (m_next[id] != NO_TASK) || (m_tail == id);
        }
103
        bool pushMTask( uint8_t id )
105     {
            if( !isMWaiting(id) )
107         {
                if( m_head == NO_TASK )
109             {
                    m_head = id;
111                 m_tail = id;
                }
113             else
```

```
            {
115             m_next[m_tail] = id;
                m_tail = id;
117         }
            return TRUE;
119     }
        else
121     {
            return FALSE;
123     }
    }
125
    inline uint8_t popDTask()
127 {
        if( d_head != NO_TASK )
129     {
            uint8_t id = d_head;
131         d_head = d_next[d_head];
            if( d_head == NO_TASK )
133         {
                d_tail = NO_TASK;
135         }
            d_next[id] = NO_TASK;
137         return id;
        }
139     else
        {
141         return NO_TASK;
        }
143 }

145 bool isDWaiting( uint8_t id )
    {
147     return (d_next[id] != NO_TASK) || (d_tail == id);
    }
149
    bool pushDTask( uint8_t id, uint32_t deadline )
151 {
        if( !isDWaiting(id) )
153     {
            if( d_head == NO_TASK )
155         {
                d_head = id;
157             d_tail = id;
            }
159         else
            {
```

```
161                 uint8_t t_curr = d_head;
                    uint8_t t_prev = d_head;
163                 uint32_t local = call LocalTime.get();
                    while (d_time[t_curr] - local <= deadline &&
165                         t_curr != NO_TASK) {
                        t_prev = t_curr;
167                     t_curr = d_next[t_curr];
                    }
169                 d_next[id] = t_curr;
                    if (t_curr == d_head) {
171                     d_head = id;
                    }
173                 else {
                        d_next[t_prev] = id;
175                     if (t_curr == NO_TASK) {
                            d_tail = id;
177                     }
                    }
179             }
                d_time[id] = call LocalTime.get() + deadline;
181             return TRUE;
            }
183         else
            {
185             return FALSE;
            }
187     }


189
    command void Scheduler.init()
191 {
        atomic
193     {
            memset( (void *)m_next, NO_TASK, sizeof(m_next) );
195         m_head = NO_TASK;
            m_tail = NO_TASK;
197         memset( (void *)d_next, NO_TASK, sizeof(d_next) );
            d_head = NO_TASK;
199         d_tail = NO_TASK;
            //      sim_scheduler_event_init(&sim_scheduler_event);
201     }
    }
203
    command bool Scheduler.runNextTask()
205 {
        uint8_t nextTask;
207     atomic
```

```
        {
209         nextTask = popDTask();
            if( nextTask == NO_TASK )
211         {
                nextTask = popMTask();
213             if (nextTask == NO_TASK) {
                    return FALSE;
215             }
                dbg("Deadline", "Running basic task %i\n", (int)nextTask);
217             signal TaskBasic.runTask[nextTask]();
                return TRUE;
219         }
        }
221     dbg("Deadline", "Running deadline task %i\n", (int)nextTask);
        signal TaskDeadline.runTask[nextTask]();
223     return TRUE;
    }
225
    command void Scheduler.taskLoop()
227 {
        for (;;)
229     {
            uint8_t nextDTask = NO_TASK;
231         uint8_t nextMTask = NO_TASK;

233         atomic
            {
235             while ((nextDTask = popDTask()) == NO_TASK &&
                        (nextMTask = popMTask()) == NO_TASK)
237             {
                    call McuSleep.sleep();
239             }
            }
241         if (nextDTask != NO_TASK) {
                dbg("Deadline", "Running deadline task %i\n", (int)nextDTask);
243             signal TaskDeadline.runTask[nextDTask]();
            }
245         else if (nextMTask != NO_TASK) {
                dbg("Deadline", "Running basic task %i\n", (int)nextMTask);
247             signal TaskBasic.runTask[nextMTask]();
            }
249     }
    }
251
    /**
253  * Return SUCCESS if the post succeeded, EBUSY if it was already posted.
     */
```

```
255
    async command error_t TaskBasic.postTask[uint8_t id]()
257 {
        atomic return pushMTask(id) ? SUCCESS : EBUSY;
259 }

261 default event void TaskBasic.runTask[uint8_t id]()
    {
263 }

265


267 async command error_t TaskDeadline.postTask[uint8_t id](uint32_t deadline)
    {
269     atomic return pushDTask(id, deadline) ? SUCCESS : EBUSY;

271 }

273 default event void TaskDeadline.runTask[uint8_t id]()
    {
275 }

277


279 }
```

B.7.2 SchedulerPrioridadeFilaP.nc

```
#define NO_STARVATION_NUM 10
2
//Define this to use on the TOSSIM
4 //#define SIM__

6 #include "hardware.h"

8 #ifdef SIM__
#include <sim_event_queue.h>
10 #endif


12
module SchedulerPrioridadeFilaP {
14     provides interface Scheduler;
    provides interface TaskBasic[uint8_t id];
16     provides interface TaskPrioridade[uint8_t id];
    uses interface McuSleep;
18 }
implementation
```

```
20 {
       enum
22     {
           NUM_TASKS = uniqueCount("TinySchedulerC.TaskBasic"),
24         NUM_PTASKS = uniqueCount("TinySchedulerC.TaskPrioridade"),
           NO_TASK = 255,
26     };

28     volatile uint8_t m_head;
       volatile uint8_t m_tail;
30     volatile uint8_t m_next[NUM_TASKS];
       volatile uint8_t p_head;
32     volatile uint8_t p_tail;
       volatile uint8_t p_next[NUM_PTASKS];
34     volatile uint8_t p_prioridade[NUM_PTASKS];

36     #ifdef SIM__
       // Aqui entram as funcoes responsaveis pelos eventos do simulador
38     // As tasks sao simuladas por eventos no TOSSIM

40     bool sim_scheduler_event_pending = FALSE;
       sim_event_t sim_scheduler_event;
42
       int sim_config_task_latency() {return 100;}
44


46     /* Only enqueue the event for execution if it is
          not already enqueued. If there are more tasks in the
48        queue, the event will re-enqueue itself (see the handle
          function). */
50
       void sim_scheduler_submit_event() {
52         if (sim_scheduler_event_pending == FALSE) {
               sim_scheduler_event.time = sim_time() + sim_config_task_latency();
54             sim_queue_insert(&sim_scheduler_event);
               sim_scheduler_event_pending = TRUE;
56         }
       }
58
       void sim_scheduler_event_handle(sim_event_t* e) {
60         sim_scheduler_event_pending = FALSE;

62         // If we successfully executed a task, re-enqueue the event. This
           // will always succeed, as sim_scheduler_event_pending was just
64         // set to be false.  Note that this means there will be an extra
           // execution (on an empty task queue). We could optimize this
66         // away, but this code is cleaner, and more accurately reflects
```

```
        // the real TinyOS main loop.

        if (call Scheduler.runNextTask()) {
            sim_scheduler_submit_event();
        }
    }


    /* Initialize a scheduler event. This should only be done
     * once, when the scheduler is initialized. */
    void sim_scheduler_event_init(sim_event_t* e) {
        e->mote = sim_node();
        e->force = 0;
        e->data = NULL;
        e->handle = sim_scheduler_event_handle;
        e->cleanup = sim_queue_cleanup_none;
    }
    #endif

    // Helper functions (internal functions) intentionally do not have atomic
    // sections.  It is left as the duty of the exported interface functions to
    // manage atomicity to minimize chances for binary code bloat.

    // move the head forward
    // if the head is at the end, mark the tail at the end, too
    // mark the task as not in the queue
    inline uint8_t popMTask()
    {
        dbg("Prioridade", "Poped a Mtask (ou nao)\n");
            if( m_head != NO_TASK )
            {
                uint8_t id = m_head;
                m_head = m_next[m_head];
                if( m_head == NO_TASK )
                {
                    m_tail = NO_TASK;
                }
                m_next[id] = NO_TASK;
                return id;
            }
            else
            {
                return NO_TASK;
            }
    }

    bool isMWaiting( uint8_t id )
```

```
114     {
            dbg("Prioridade", "isMWaiting: %d\n", (m_next[id] != NO_TASK) || (m_tail == id));
116         return (m_next[id] != NO_TASK) || (m_tail == id);
        }
118
        bool pushMTask( uint8_t id )
120     {
            dbg("Prioridade", "pushMTask %i\n", (int)id);
122         if( !isMWaiting(id) )
            {
124             if( m_head == NO_TASK )
                {
126                 m_head = id;
                    m_tail = id;
128             }
                else
130             {
                    m_next[m_tail] = id;
132                 m_tail = id;
                }
134             return TRUE;
            }
136         else
            {
138             return FALSE;
            }
140     }

142     inline uint8_t popPTask()
        {
144         dbg("Prioridade", "Poped a Dtask (ou nao)\n");
            if( p_head != NO_TASK )
146         {
                uint8_t id = p_head;
148             p_head = p_next[p_head];
                if( p_head == NO_TASK )
150             {
                    p_tail = NO_TASK;
152             }
                p_next[id] = NO_TASK;
154             dbg("Prioridade_run", "Rodou PTask %i\n", (int) id);
                return id;
156         }
            else
158         {
                return NO_TASK;
160         }
```

```
    }
162
    bool isPWaiting( uint8_t id )
164 {
        dbg("Prioridade", "isPWaiting: %d\n", (p_next[id] != NO_TASK) || (p_tail == id));
166     return (p_next[id] != NO_TASK) || (p_tail == id);
    }
168
    bool pushPTask( uint8_t id, uint8_t prioridade )
170 {
        dbg("Prioridade", "pushPTask %i\n", (int)id);
172     if( !isPWaiting(id) )
        {
174         if( p_head == NO_TASK )
            {
176             p_head = id;
                p_tail = id;
178         }
            else
180         {
                uint8_t t_curr = p_head;
182             uint8_t t_prev = p_head;
                while (p_prioridade[t_curr] <= prioridade &&
184                     t_curr != NO_TASK) {
                    t_prev = t_curr;
186                 t_curr = p_next[t_curr];
                }
188             p_next[id] = t_curr;
                if (t_curr == p_head) {
190                 p_head = id;
                }
192             else {
                    p_next[t_prev] = id;
194                 if (t_curr == NO_TASK) {
                        p_tail = id;
196                 }
                }
198         }
            p_prioridade[id] = prioridade;
200         return TRUE;
        }
202     else
        {
204         return FALSE;
        }
206 }
```

```
208
    command void Scheduler.init()
210 {
        dbg("Prioridade", "init\n");
212     atomic
        {
214         memset( (void *)m_next, NO_TASK, sizeof(m_next) );
            m_head = NO_TASK;
216         m_tail = NO_TASK;
            memset( (void *)p_next, NO_TASK, sizeof(p_next) );
218         p_head = NO_TASK;
            p_tail = NO_TASK;
220
            #ifdef SIM__
222         sim_scheduler_event_pending = FALSE;
            sim_scheduler_event_init(&sim_scheduler_event);
224         #endif
        }
226 }

228 command bool Scheduler.runNextTask()
    {
230     uint8_t nextTask;
        dbg("Prioridade", "runNextTask\n");
232     atomic
        {
234         nextTask = popPTask();
            dbg("Prioridade", "popPTask: %i\n", (int)nextTask);
236         if( nextTask == NO_TASK )
            {
238             nextTask = popMTask();
                dbg("Prioridade", "popMTask: %i\n", (int)nextTask);
240             if (nextTask == NO_TASK) {
                    return FALSE;
242             }
                dbg("Prioridade", "Running basic task %i\n", (int)nextTask);
244             signal TaskBasic.runTask[nextTask]();
                return TRUE;
246         }
        }
248     dbg("Prioridade", "Running prioridade task %i\n", (int)nextTask);
        signal TaskPrioridade.runTask[nextTask]();
250     return TRUE;
    }
252
    command void Scheduler.taskLoop()
254 {
```

```
        uint8_t max_ptask = 0;
256     dbg("Prioridade", "Taskloop\n");
        for (;;)
258     {
            uint8_t nextPTask = NO_TASK;
260         uint8_t nextMTask = NO_TASK;

262         if (max_ptask > NO_STARVATION_NUM)
            {
264             max_ptask = 0;
                atomic {
266             nextMTask = popMTask();
                }
268             if (nextMTask != NO_TASK)
                    signal TaskBasic.runTask[nextMTask]();
270         }
            if (nextMTask == NO_TASK)
272         {
                atomic
274             {
                    while ((nextPTask = popPTask()) == NO_TASK &&
276                        (nextMTask = popMTask()) == NO_TASK)
                    {
278                     call McuSleep.sleep();
                    }
280             }
                if (nextPTask != NO_TASK) {
282                 dbg("Prioridade", "Running prioridade task %i\n", (int)nextPTask);
                    max_ptask++;
284                 signal TaskPrioridade.runTask[nextPTask]();
                }
286             else if (nextMTask != NO_TASK) {
                    dbg("Prioridade", "Running basic task %i\n", (int)nextMTask);
288                 max_ptask = 0;
                    signal TaskBasic.runTask[nextMTask]();
290             }
            }
292     }
    }
294

    /**
296  * Return SUCCESS if the post succeeded, EBUSY if it was already posted.
     */
298

    async command error_t TaskBasic.postTask[uint8_t id]()
300 {
        error_t result;
```

```
302
        dbg("Prioridade", "postTaskBasic\n");
304     atomic {
            result = pushMTask(id) ? SUCCESS : EBUSY;
306     }
        #ifdef SIM__
308     if (result == SUCCESS)
            sim_scheduler_submit_event();
310     #endif

312     return result;

314 }

316 default event void TaskBasic.runTask[uint8_t id]()
    {
318 }


320


322 async command error_t TaskPrioridade.postTask[uint8_t id](uint8_t prioridade)
    {
324     error_t result;

326     dbg("Prioridade", "postTaskBasic\n");
        atomic {
328         result = pushPTask(id, prioridade) ? SUCCESS : EBUSY;
        }
330     #ifdef SIM__
        if (result == SUCCESS)
332         sim_scheduler_submit_event();
        #endif
334
        return result;
336 }

338 default event void TaskPrioridade.runTask[uint8_t id]()
    {
340 }

342 }
```

B.7.3 SchedulerPrioridadeFilaAgingP.nc

```
1 #define NO_STARVATION_NUM 10

3 //Define this to use on the TOSSIM
```

```
//#define SIM__

#include "hardware.h"

#ifdef SIM__
#include <sim_event_queue.h>
#endif


module SchedulerPrioridadeFilaAgingP {
    provides interface Scheduler;
    provides interface TaskBasic[uint8_t id];
    provides interface TaskPrioridade[uint8_t id];
    uses interface McuSleep;
}
implementation
{
    enum
    {
        NUM_TASKS = uniqueCount("TinySchedulerC.TaskBasic"),
        NUM_PTASKS = uniqueCount("TinySchedulerC.TaskPrioridade"),
        NO_TASK = 255,
    };

    volatile uint8_t m_head;
    volatile uint8_t m_tail;
    volatile uint8_t m_next[NUM_TASKS];
    volatile uint8_t p_head;
    volatile uint8_t p_tail;
    volatile uint8_t p_next[NUM_PTASKS];
    volatile uint8_t p_prioridade[NUM_PTASKS];

    #ifdef SIM__
    // Aqui entram as funcoes responsaveis pelos eventos do simulador
    // As tasks sao simuladas por eventos no TOSSIM

    bool sim_scheduler_event_pending = FALSE;
    sim_event_t sim_scheduler_event;

    int sim_config_task_latency() {return 100;}


    /* Only enqueue the event for execution if it is
       not already enqueued. If there are more tasks in the
       queue, the event will re-enqueue itself (see the handle
       function). */
```

```
51      void sim_scheduler_submit_event() {
            if (sim_scheduler_event_pending == FALSE) {
53              sim_scheduler_event.time = sim_time() + sim_config_task_latency();
                sim_queue_insert(&sim_scheduler_event);
55              sim_scheduler_event_pending = TRUE;
            }
57      }

59      void sim_scheduler_event_handle(sim_event_t* e) {
            sim_scheduler_event_pending = FALSE;
61
            // If we successfully executed a task, re-enqueue the event. This
63          // will always succeed, as sim_scheduler_event_pending was just
            // set to be false.  Note that this means there will be an extra
65          // execution (on an empty task queue). We could optimize this
            // away, but this code is cleaner, and more accurately reflects
67          // the real TinyOS main loop.

69          if (call Scheduler.runNextTask()) {
                sim_scheduler_submit_event();
71          }
        }
73

75      /* Initialize a scheduler event. This should only be done
         * once, when the scheduler is initialized. */
77      void sim_scheduler_event_init(sim_event_t* e) {
            e->mote = sim_node();
79          e->force = 0;
            e->data = NULL;
81          e->handle = sim_scheduler_event_handle;
            e->cleanup = sim_queue_cleanup_none;
83      }
        #endif
85
        // Helper functions (internal functions) intentionally do not have atomic
87      // sections.  It is left as the duty of the exported interface functions to
        // manage atomicity to minimize chances for binary code bloat.
89
        // move the head forward
91      // if the head is at the end, mark the tail at the end, too
        // mark the task as not in the queue
93      inline uint8_t popMTask()
        {
95          dbg("Prioridade", "Poped a Mtask (ou nao)\n");
                if( m_head != NO_TASK )
97              {
```

```
                uint8_t id = m_head;
99              m_head = m_next[m_head];
                if( m_head == NO_TASK )
101             {
                    m_tail = NO_TASK;
103             }
                m_next[id] = NO_TASK;
105             return id;
            }
107         else
            {
109             return NO_TASK;
            }
111     }

113     bool isMWaiting( uint8_t id )
        {
115         dbg("Prioridade", "isMWaiting: %d\n", (m_next[id] != NO_TASK) || (m_tail == id));
            return (m_next[id] != NO_TASK) || (m_tail == id);
117     }

119     bool pushMTask( uint8_t id )
        {
121         dbg("Prioridade", "pushMTask %i\n", (int)id);
            if( !isMWaiting(id) )
123         {
                if( m_head == NO_TASK )
125             {
                    m_head = id;
127                 m_tail = id;
                }
129             else
                {
131                 m_next[m_tail] = id;
                    m_tail = id;
133             }
                return TRUE;
135         }
            else
137         {
                return FALSE;
139         }
        }
141
        inline uint8_t popPTask()
143     {
            uint8_t atual;
```

```
145         dbg("Prioridade", "Poped a Dtask (ou nao)\n");
            if( p_head != NO_TASK )
147         {
                uint8_t id = p_head;
149             p_head = p_next[p_head];
                if( p_head == NO_TASK )
151             {
                    p_tail = NO_TASK;
153             }
                p_next[id] = NO_TASK;
155
                //antes de retornar, aumentar a prioridade de todas tarefas
157             atual = p_head;
                while (atual != NO_TASK)
159             {
                    if (p_prioridade[atual] > 0)
161                     p_prioridade[atual]--;
                    atual = p_next[atual];
163             }

165             dbg("Prioridade_run", "Rodou PTask %i\n\tcom prioridade %i\n", (int) id, (int) p_pri
                return id;
167         }
            else
169         {
                return NO_TASK;
171         }
        }
173
        bool isPWaiting( uint8_t id )
175     {
            dbg("Prioridade", "isPWaiting: %d\n", (p_next[id] != NO_TASK) || (p_tail == id));
177         return (p_next[id] != NO_TASK) || (p_tail == id);
        }
179
        bool pushPTask( uint8_t id, uint8_t prioridade )
181     {
            dbg("Prioridade", "pushPTask %i\n", (int)id);
183         if( !isPWaiting(id) )
            {
185             if( p_head == NO_TASK )
                {
187                 p_head = id;
                    p_tail = id;
189             }
                else
191             {
```

```
                uint8_t t_curr = p_head;
193             uint8_t t_prev = p_head;
                while (p_prioridade[t_curr] <= prioridade &&
195                     t_curr != NO_TASK) {
                    t_prev = t_curr;
197                 t_curr = p_next[t_curr];
                }
199             p_next[id] = t_curr;
                if (t_curr == p_head) {
201                 p_head = id;
                }
203             else {
                    p_next[t_prev] = id;
205                 if (t_curr == NO_TASK) {
                        p_tail = id;
207                 }
                }
209         }
            p_prioridade[id] = prioridade;
211         return TRUE;
        }
213     else
        {
215         return FALSE;
        }
217 }


219
    command void Scheduler.init()
221 {
        dbg("Prioridade", "init\n");
223     atomic
        {
225         memset( (void *)m_next, NO_TASK, sizeof(m_next) );
            m_head = NO_TASK;
227         m_tail = NO_TASK;
            memset( (void *)p_next, NO_TASK, sizeof(p_next) );
229         p_head = NO_TASK;
            p_tail = NO_TASK;
231
            #ifdef SIM__
233         sim_scheduler_event_pending = FALSE;
            sim_scheduler_event_init(&sim_scheduler_event);
235         #endif
        }
237 }
```

```
239     command bool Scheduler.runNextTask()
        {
241         uint8_t nextTask;
            dbg("Prioridade", "runNextTask\n");
243         atomic
            {
245             nextTask = popPTask();
                dbg("Prioridade", "popPTask: %i\n", (int)nextTask);
247             if( nextTask == NO_TASK )
                {
249                 nextTask = popMTask();
                    dbg("Prioridade", "popMTask: %i\n", (int)nextTask);
251                 if (nextTask == NO_TASK) {
                        return FALSE;
253                 }
                    dbg("Prioridade", "Running basic task %i\n", (int)nextTask);
255                 signal TaskBasic.runTask[nextTask]();
                    return TRUE;
257             }
            }
259         dbg("Prioridade", "Running prioridade task %i\n", (int)nextTask);
            signal TaskPrioridade.runTask[nextTask]();
261         return TRUE;
        }
263
        command void Scheduler.taskLoop()
265     {
            uint8_t max_ptask = 0;
267         dbg("Prioridade", "Taskloop\n");
            for (;;)
269         {
                uint8_t nextPTask = NO_TASK;
271             uint8_t nextMTask = NO_TASK;

273             if (max_ptask > NO_STARVATION_NUM)
                {
275                 max_ptask = 0;
                    atomic {
277                 nextMTask = popMTask();
                    }
279                 if (nextMTask != NO_TASK)
                        signal TaskBasic.runTask[nextMTask]();
281             }
                if (nextMTask == NO_TASK)
283             {
                    atomic
285                 {
```

```
                    while ((nextPTask = popPTask()) == NO_TASK &&
287                         (nextMTask = popMTask()) == NO_TASK)
                    {
289                     call McuSleep.sleep();
                    }
291             }
                if (nextPTask != NO_TASK) {
293                 dbg("Prioridade", "Running prioridade task %i\n", (int)nextPTask);
                    max_ptask++;
295                 signal TaskPrioridade.runTask[nextPTask]();
                }
297             else if (nextMTask != NO_TASK) {
                    dbg("Prioridade", "Running basic task %i\n", (int)nextMTask);
299                 max_ptask = 0;
                    signal TaskBasic.runTask[nextMTask]();
301             }
            }
303     }
    }
305
    /**
307  * Return SUCCESS if the post succeeded, EBUSY if it was already posted.
     */
309
    async command error_t TaskBasic.postTask[uint8_t id]()
311 {
        error_t result;
313
        dbg("Prioridade", "postTaskBasic\n");
315     atomic {
            result = pushMTask(id) ? SUCCESS : EBUSY;
317     }
        #ifdef SIM__
319     if (result == SUCCESS)
            sim_scheduler_submit_event();
321     #endif

323     return result;

325 }

327 default event void TaskBasic.runTask[uint8_t id]()
    {
329 }

331
```

```
333     async command error_t TaskPrioridade.postTask[uint8_t id](uint8_t prioridade)
        {
335         error_t result;

337         dbg("Prioridade", "postTaskBasic\n");
            atomic {
339             result = pushPTask(id, prioridade) ? SUCCESS : EBUSY;
            }
341         #ifdef SIM__
            if (result == SUCCESS)
343             sim_scheduler_submit_event();
            #endif
345
            return result;
347     }

349     default event void TaskPrioridade.runTask[uint8_t id]()
        {
351     }

353 }
```

B.7.4 SchedulerPrioridadeHeapP.nc

```
/* Escalonador de prioridade
2       Utiliza uma heap como fila de prioridades

4       Autor: Pedro Rosanes
*/
6
//Descomentar para rodar no simulador
8 //#define SIM__

10 #include "hardware.h"

12 #ifdef SIM__
#include <sim_event_queue.h>
14 #endif

16 #define NO_STARVATION_NUM 10

18 module SchedulerPrioridadeHeapP {
    provides interface Scheduler;
20  provides interface TaskBasic[uint8_t id];
    provides interface TaskPrioridade[uint8_t id];
22  uses interface McuSleep;
}
```

```
24 implementation
   {
26     enum
       {
28         NUM_TASKS = uniqueCount("TinySchedulerC.TaskBasic"),
           NUM_MTASKS = uniqueCount("TinySchedulerC.TaskPrioridade"),
30         NO_TASK = 255,
       };
32
       volatile uint8_t m_head;
34     volatile uint8_t m_tail;
       volatile uint8_t m_next[NUM_TASKS];
36     volatile uint8_t tamanho;
       volatile uint8_t p_fila[NUM_MTASKS];
38     volatile uint8_t p_prioridade[NUM_MTASKS];
       volatile uint8_t p_isDWaiting[NUM_MTASKS];
40
       // Aqui entram as funcoes responsaveis pelos eventos do simulador
42     // As tasks sao simuladas por eventos no TOSSIM

44     #ifdef SIM__
       bool sim_scheduler_event_pending = FALSE;
46     sim_event_t sim_scheduler_event;

48     int sim_config_task_latency() {return 100;}


50
       /* Only enqueue the event for execution if it is
52        not already enqueued. If there are more tasks in the
          queue, the event will re-enqueue itself (see the handle
54        function). */

56     void sim_scheduler_submit_event() {
           if (sim_scheduler_event_pending == FALSE) {
58             sim_scheduler_event.time = sim_time() + sim_config_task_latency();
               sim_queue_insert(&sim_scheduler_event);
60             sim_scheduler_event_pending = TRUE;
           }
62     }

64     void sim_scheduler_event_handle(sim_event_t* e) {
           sim_scheduler_event_pending = FALSE;
66
           // If we successfully executed a task, re-enqueue the event. This
68         // will always succeed, as sim_scheduler_event_pending was just
           // set to be false.  Note that this means there will be an extra
70         // execution (on an empty task queue). We could optimize this
```

```
        // away, but this code is cleaner, and more accurately reflects
72      // the real TinyOS main loop.

74      if (call Scheduler.runNextTask()) {
            sim_scheduler_submit_event();
76      }
    }
78

80  /* Initialize a scheduler event. This should only be done
     * once, when the scheduler is initialized. */
82  void sim_scheduler_event_init(sim_event_t* e) {
        e->mote = sim_node();
84      e->force = 0;
        e->data = NULL;
86      e->handle = sim_scheduler_event_handle;
        e->cleanup = sim_queue_cleanup_none;
88  }
    #endif
90

92  // Helper functions (internal functions) intentionally do not have atomic
    // sections.  It is left as the duty of the exported interface functions to
94  // manage atomicity to minimize chances for binary code bloat.

96  // move the head forward
    // if the head is at the end, mark the tail at the end, too
98  // mark the task as not in the queue
    inline uint8_t popMTask()
100 {
        dbg("Prioridade", "Poped a Mtask (ou nao)\n");
102     if( m_head != NO_TASK )
        {
104         uint8_t id = m_head;
            m_head = m_next[m_head];
106         if( m_head == NO_TASK )
            {
108             m_tail = NO_TASK;
            }
110         m_next[id] = NO_TASK;
            return id;
112     }
        else
114     {
            return NO_TASK;
116     }
    }
```

```
118
    bool isMWaiting( uint8_t id )
120 {
        dbg("Prioridade", "isMWaiting: %d\n", (m_next[id] != NO_TASK) || (m_tail == id));
122     return (m_next[id] != NO_TASK) || (m_tail == id);
    }
124
    bool pushMTask( uint8_t id )
126 {
        dbg("Prioridade", "pushMTask %i\n", (int)id);
128     if( !isMWaiting(id) )
        {
130         if( m_head == NO_TASK )
            {
132             m_head = id;
                m_tail = id;
134         }
            else
136         {
                m_next[m_tail] = id;
138             m_tail = id;
            }
140         return TRUE;
        }
142     else
        {
144         return FALSE;
        }
146 }

148 inline uint8_t popPTask()
    {
150     uint8_t id, i, menor, temp;

152     dbg("Prioridade", "Poped a Dtask (ou nao)\n");
        //Se nao tem ninguem na fila
154     if (tamanho == 0)
            return NO_TASK;
156
        //Se tem alguem na fila
158     id = p_fila[0];

160     p_fila[0] = p_fila[tamanho-1];
        p_prioridade[0] = p_prioridade[tamanho-1];
162     tamanho--;

164     i = 0;
```

```
        while (i < tamanho)
166     {
            menor = i;
168         if (2*i+1 < tamanho &&
                    p_prioridade[2*i+1] < p_prioridade[menor])
170             menor = 2*i+1;
            if (2*i+2 < tamanho &&
172                 p_prioridade[2*i+2] < p_prioridade[menor])
                menor = 2*i+2;
174
            if (menor != i)
176         {
                temp = p_fila[i];
178             p_fila[i] = p_fila[menor];
                p_fila[menor] = temp;
180
                temp = p_prioridade[i];
182             p_prioridade[i] = p_prioridade[menor];
                p_prioridade[menor] = temp;
184         }
            else
186             break;
            i = menor;
188     }

190     p_isDWaiting[id] = 0;
        dbg("Prioridade_run", "Rodou PTask %i\n", (int) id);
192     return id;

194 }

196 bool pushPTask( uint8_t id, uint8_t prioridade )
    {
198     int16_t temp, pai, filho;

200     dbg("Prioridade", "pushPTask %i\n", (int)id);
        if( !p_isDWaiting[id] )
202     {
            p_isDWaiting[id] = 1;
204
            p_fila[tamanho] = id;
206         p_prioridade[tamanho] = prioridade;
            pai = (tamanho - 1)/2;
208         filho = tamanho;
            while (pai >= 0)
210         {
                if (p_prioridade[pai] > p_prioridade[filho])
```

```
212                 {
                        temp = p_fila[pai];
214                     p_fila[pai] = p_fila[filho];
                        p_fila[filho] = temp;
216
                        temp = p_prioridade[pai];
218                     p_prioridade[pai] = p_prioridade[filho];
                        p_prioridade[filho] = temp;
220                 }
                    else
222                    break;

224                 filho = pai;
                    pai = (filho-1)/2;
226             }
                tamanho++;
228             return TRUE;
            }
230         else
            {
232             return FALSE;
            }
234     }

236
    command void Scheduler.init()
238 {
        dbg("Prioridade", "init\n");
240     atomic
        {
242         memset( (void *)m_next, NO_TASK, sizeof(m_next) );
            m_head = NO_TASK;
244         m_tail = NO_TASK;
            memset( (void *)p_fila, NO_TASK, sizeof(p_fila) );
246         memset( (void *)p_prioridade, NO_TASK, sizeof(p_prioridade) );
            memset( (void *)p_isDWaiting, 0, sizeof(p_isDWaiting) );
248         tamanho = 0;

250         #ifdef SIM__
            sim_scheduler_event_pending = FALSE;
252         sim_scheduler_event_init(&sim_scheduler_event);
            #endif
254     }
    }
256
    command bool Scheduler.runNextTask()
258 {
```

```
        uint8_t nextTask;
260     dbg("Prioridade", "runNextTask\n");
        atomic
262     {
            nextTask = popPTask();
264         dbg("Prioridade", "popPTask: %i\n", (int)nextTask);
            if( nextTask == NO_TASK )
266         {
                nextTask = popMTask();
268             dbg("Prioridade", "popMTask: %i\n", (int)nextTask);
                if (nextTask == NO_TASK) {
270                 return FALSE;
                }
272             dbg("Prioridade", "Running basic task %i\n", (int)nextTask);
                signal TaskBasic.runTask[nextTask]();
274             return TRUE;
            }
276     }
        dbg("Prioridade", "Running prioridade task %i\n", (int)nextTask);
278     signal TaskPrioridade.runTask[nextTask]();
        return TRUE;
280 }

282 command void Scheduler.taskLoop()
    {
284     uint8_t max_ptask = 0;
        dbg("Prioridade", "Taskloop\n");
286     for (;;)
        {
288         uint8_t nextPTask = NO_TASK;
            uint8_t nextMTask = NO_TASK;
290
            if (max_ptask > NO_STARVATION_NUM)
292         {
                max_ptask = 0;
294             atomic {
                    nextMTask = popMTask();
296             }
                if (nextMTask != NO_TASK)
298                 signal TaskBasic.runTask[nextMTask]();
            }
300         if (nextMTask == NO_TASK)
            {
302             atomic
                {
304                 while ((nextPTask = popPTask()) == NO_TASK &&
                           (nextMTask = popMTask()) == NO_TASK)
```

```
306                     {
                            call McuSleep.sleep();
308                     }
                    }
310                 if (nextPTask != NO_TASK) {
                        dbg("Prioridade", "Running prioridade task %i\n", (int)nextPTask);
312                     max_ptask++;
                        signal TaskPrioridade.runTask[nextPTask]();
314                 }
                    else if (nextMTask != NO_TASK) {
316                     dbg("Prioridade", "Running basic task %i\n", (int)nextMTask);
                        max_ptask = 0;
318                     signal TaskBasic.runTask[nextMTask]();
                    }
320             }
        }
322     }

324     /**
         * Return SUCCESS if the post succeeded, EBUSY if it was already posted.
326      */

328     async command error_t TaskBasic.postTask[uint8_t id]()
        {
330         error_t result;

332         dbg("Prioridade", "postTaskBasic\n");
            atomic {
334             result = pushMTask(id) ? SUCCESS : EBUSY;
            }
336         #ifdef SIM__
            if (result == SUCCESS)
338             sim_scheduler_submit_event();
            #endif
340
            return result;
342
        }
344
        default event void TaskBasic.runTask[uint8_t id]()
346     {
        }
348



350
        async command error_t TaskPrioridade.postTask[uint8_t id](uint8_t prioridade)
352     {
```

```
        error_t result;
354
        dbg("Prioridade", "postTaskBasic\n");
356     atomic {
            result = pushPTask(id, prioridade) ? SUCCESS : EBUSY;
358     }
        #ifdef SIM__
360     if (result == SUCCESS)
            sim_scheduler_submit_event();
362     #endif

364     return result;
    }
366
    default event void TaskPrioridade.runTask[uint8_t id]()
368 {
    }
370


372
}
```

B.7.5 SchedulerPrioridadeHeapAgingP.nc

```
/* Escalonador de prioridade
2     Utiliza uma heap como fila de prioridades

4     Autor: Pedro Rosanes
*/
6
//Descomentar para rodar no simulador
8 //#define SIM__

10 #include "hardware.h"

12 #ifdef SIM__
#include <sim_event_queue.h>
14 #endif

16 #define NO_STARVATION_NUM 10

18 module SchedulerPrioridadeHeapAgingP {
    provides interface Scheduler;
20  provides interface TaskBasic[uint8_t id];
    provides interface TaskPrioridade[uint8_t id];
22  uses interface McuSleep;
}
```

```
24 implementation
   {
26     enum
       {
28         NUM_TASKS = uniqueCount("TinySchedulerC.TaskBasic"),
           NUM_MTASKS = uniqueCount("TinySchedulerC.TaskPrioridade"),
30         NO_TASK = 255,
       };
32
       volatile uint8_t m_head;
34     volatile uint8_t m_tail;
       volatile uint8_t m_next[NUM_TASKS];
36     volatile uint8_t tamanho;
       volatile uint8_t p_fila[NUM_MTASKS];
38     volatile uint8_t p_prioridade[NUM_MTASKS];
       volatile uint8_t p_isDWaiting[NUM_MTASKS];
40
       // Aqui entram as funcoes responsaveis pelos eventos do simulador
42     // As tasks sao simuladas por eventos no TOSSIM

44     #ifdef SIM__
       bool sim_scheduler_event_pending = FALSE;
46     sim_event_t sim_scheduler_event;

48     int sim_config_task_latency() {return 100;}


50
       /* Only enqueue the event for execution if it is
52        not already enqueued. If there are more tasks in the
          queue, the event will re-enqueue itself (see the handle
54        function). */

56     void sim_scheduler_submit_event() {
           if (sim_scheduler_event_pending == FALSE) {
58             sim_scheduler_event.time = sim_time() + sim_config_task_latency();
               sim_queue_insert(&sim_scheduler_event);
60             sim_scheduler_event_pending = TRUE;
           }
62     }

64     void sim_scheduler_event_handle(sim_event_t* e) {
           sim_scheduler_event_pending = FALSE;
66
           // If we successfully executed a task, re-enqueue the event. This
68         // will always succeed, as sim_scheduler_event_pending was just
           // set to be false.  Note that this means there will be an extra
70         // execution (on an empty task queue). We could optimize this
```

```
        // away, but this code is cleaner, and more accurately reflects
72      // the real TinyOS main loop.

74      if (call Scheduler.runNextTask()) {
            sim_scheduler_submit_event();
76      }
    }
78

80  /* Initialize a scheduler event. This should only be done
     * once, when the scheduler is initialized. */
82  void sim_scheduler_event_init(sim_event_t* e) {
        e->mote = sim_node();
84      e->force = 0;
        e->data = NULL;
86      e->handle = sim_scheduler_event_handle;
        e->cleanup = sim_queue_cleanup_none;
88  }
    #endif
90

92  // Helper functions (internal functions) intentionally do not have atomic
    // sections.  It is left as the duty of the exported interface functions to
94  // manage atomicity to minimize chances for binary code bloat.

96  // move the head forward
    // if the head is at the end, mark the tail at the end, too
98  // mark the task as not in the queue
    inline uint8_t popMTask()
100 {
        dbg("Prioridade", "Poped a Mtask (ou nao)\n");
102     if( m_head != NO_TASK )
        {
104         uint8_t id = m_head;
            m_head = m_next[m_head];
106         if( m_head == NO_TASK )
            {
108             m_tail = NO_TASK;
            }
110         m_next[id] = NO_TASK;
            return id;
112     }
        else
114     {
            return NO_TASK;
116     }
    }
```

```
118
    bool isMWaiting( uint8_t id )
120 {
        dbg("Prioridade", "isMWaiting: %d\n", (m_next[id] != NO_TASK) || (m_tail == id));
122     return (m_next[id] != NO_TASK) || (m_tail == id);
    }
124
    bool pushMTask( uint8_t id )
126 {
        dbg("Prioridade", "pushMTask %i\n", (int)id);
128     if( !isMWaiting(id) )
        {
130         if( m_head == NO_TASK )
            {
132             m_head = id;
                m_tail = id;
134         }
            else
136         {
                m_next[m_tail] = id;
138             m_tail = id;
            }
140         return TRUE;
        }
142     else
        {
144         return FALSE;
        }
146 }

148 inline uint8_t popPTask()
    {
150     uint8_t id, i, menor, temp;

152     dbg("Prioridade", "Poped a Dtask (ou nao)\n");
        //Se nao tem ninguem na fila
154     if (tamanho == 0)
            return NO_TASK;
156
        //Se tem alguem na fila
158     id = p_fila[0];
        dbg("Prioridade_run", "Rodou PTask %i\n\tcom prioridade %i", (int) p_fila[0], (int) p_p
160
        p_fila[0] = p_fila[tamanho-1];
162     p_prioridade[0] = p_prioridade[tamanho-1];
        tamanho--;
164
```

```
        i = 0;
166     while (i < tamanho)
        {
168         menor = i;
            if (2*i+1 < tamanho &&
170                 p_prioridade[2*i+1] < p_prioridade[menor])
                menor = 2*i+1;
172         if (2*i+2 < tamanho &&
                    p_prioridade[2*i+2] < p_prioridade[menor])
174             menor = 2*i+2;

176         if (menor != i)
            {
178             temp = p_fila[i];
                p_fila[i] = p_fila[menor];
180             p_fila[menor] = temp;

182             temp = p_prioridade[i];
                p_prioridade[i] = p_prioridade[menor];
184             p_prioridade[menor] = temp;
            }
186         else
                break;
188         i = menor;
        }
190
        p_isDWaiting[id] = 0;
192
        //Antes de retornar, aumenta a prioridade de todos que estao na fila.
194     for (i = 0; i < tamanho; i--)
            if (p_prioridade[i] > 0)
196             p_prioridade[i]--;

198     return id;

200 }

202 bool pushPTask( uint8_t id, uint8_t prioridade )
    {
204     int16_t temp, pai, filho;

206     dbg("Prioridade", "pushPTask %i\n", (int)id);
        if( !p_isDWaiting[id] )
208     {
            p_isDWaiting[id] = 1;
210
            p_fila[tamanho] = id;
```

```
212             p_prioridade[tamanho] = prioridade;
                pai = (tamanho - 1)/2;
214             filho = tamanho;
                while (pai >= 0)
216             {
                    if (p_prioridade[pai] > p_prioridade[filho])
218                 {
                        temp = p_fila[pai];
220                     p_fila[pai] = p_fila[filho];
                        p_fila[filho] = temp;
222
                        temp = p_prioridade[pai];
224                     p_prioridade[pai] = p_prioridade[filho];
                        p_prioridade[filho] = temp;
226                 }
                    else
228                    break;

230                 filho = pai;
                    pai = (filho -1)/2;
232             }
                tamanho++;
234             return TRUE;
            }
236         else
            {
238             return FALSE;
            }
240     }


242
    command void Scheduler.init()
244 {
        dbg("Prioridade", "init\n");
246     atomic
        {
248         memset( (void *)m_next, NO_TASK, sizeof(m_next) );
            m_head = NO_TASK;
250         m_tail = NO_TASK;
            memset( (void *)p_fila, NO_TASK, sizeof(p_fila) );
252         memset( (void *)p_prioridade, NO_TASK, sizeof(p_prioridade) );
            memset( (void *)p_isDWaiting, 0, sizeof(p_isDWaiting) );
254         tamanho = 0;

256         #ifdef SIM__
            sim_scheduler_event_pending = FALSE;
258         sim_scheduler_event_init(&sim_scheduler_event);
```

```
            #endif
        }
    }

    command bool Scheduler.runNextTask()
    {
        uint8_t nextTask;
        dbg("Prioridade", "runNextTask\n");
        atomic
        {
            nextTask = popPTask();
            dbg("Prioridade", "popPTask: %i\n", (int)nextTask);
            if( nextTask == NO_TASK )
            {
                nextTask = popMTask();
                dbg("Prioridade", "popMTask: %i\n", (int)nextTask);
                if (nextTask == NO_TASK) {
                    return FALSE;
                }
                dbg("Prioridade", "Running basic task %i\n", (int)nextTask);
                signal TaskBasic.runTask[nextTask]();
                return TRUE;
            }
        }
        dbg("Prioridade", "Running prioridade task %i\n", (int)nextTask);
        signal TaskPrioridade.runTask[nextTask]();
        return TRUE;
    }

    command void Scheduler.taskLoop()
    {
        uint8_t max_ptask = 0;
        dbg("Prioridade", "Taskloop\n");
        for (;;)
        {
            uint8_t nextPTask = NO_TASK;
            uint8_t nextMTask = NO_TASK;

            if (max_ptask > NO_STARVATION_NUM)
            {
                max_ptask = 0;
                atomic {
                    nextMTask = popMTask();
                }
                if (nextMTask != NO_TASK)
                    signal TaskBasic.runTask[nextMTask]();
            }
```

```
306             if (nextMTask == NO_TASK)
                {
308                 atomic
                    {
310                     while ((nextPTask = popPTask()) == NO_TASK &&
                                (nextMTask = popMTask()) == NO_TASK)
312                     {
                            call McuSleep.sleep();
314                     }
                    }
316                 if (nextPTask != NO_TASK) {
                        dbg("Prioridade", "Running prioridade task %i\n", (int)nextPTask);
318                     max_ptask++;
                        signal TaskPrioridade.runTask[nextPTask]();
320                 }
                    else if (nextMTask != NO_TASK) {
322                     dbg("Prioridade", "Running basic task %i\n", (int)nextMTask);
                        max_ptask = 0;
324                     signal TaskBasic.runTask[nextMTask]();
                    }
326             }
        }
328 }

330 /**
     * Return SUCCESS if the post succeeded, EBUSY if it was already posted.
332  */

334 async command error_t TaskBasic.postTask[uint8_t id]()
    {
336     error_t result;

338     dbg("Prioridade", "postTaskBasic\n");
        atomic {
340         result = pushMTask(id) ? SUCCESS : EBUSY;
        }
342     #ifdef SIM__
        if (result == SUCCESS)
344         sim_scheduler_submit_event();
        #endif
346
        return result;
348

    }
350
    default event void TaskBasic.runTask[uint8_t id]()
352 {
```

```
    }
354
356
    async command error_t TaskPrioridade.postTask[uint8_t id](uint8_t prioridade)
358 {
        error_t result;
360
        dbg("Prioridade", "postTaskBasic\n");
362     atomic {
            result = pushPTask(id, prioridade) ? SUCCESS : EBUSY;
364     }
        #ifdef SIM__
366     if (result == SUCCESS)
            sim_scheduler_submit_event();
368     #endif

370     return result;
    }
372
    default event void TaskPrioridade.runTask[uint8_t id]()
374 {
    }
376
378
}
```

B.7.6 SchedulerMultinivelP.nc

```
#define NO_STARVATION_NUM 10
2
//Define this to use on the TOSSIM
4 //#define SIM__

6 #include "hardware.h"

8 #ifdef SIM__
#include <sim_event_queue.h>
10 #endif


12
module SchedulerMultinivelP {
14     provides interface Scheduler;
    provides interface TaskBasic[uint8_t id];
16  provides interface TaskRadio[uint8_t id];
    provides interface TaskSerial[uint8_t id];
```

```
18      provides interface TaskSense[uint8_t id];
        uses interface McuSleep;
20  }
    implementation
22  {
        enum
24      {
            NUM_TASKSBASIC  = uniqueCount("TinySchedulerC.TaskBasic"),
26          NUM_TASKSRADIO  = uniqueCount("TinySchedulerC.TaskRadio"),
            NUM_TASKSSERIAL = uniqueCount("TinySchedulerC.TaskSerial"),
28          NUM_TASKSSENSE  = uniqueCount("TinySchedulerC.TaskSense"),
            NO_TASK = 255,
30      };

32      volatile uint8_t basic_head;
        volatile uint8_t basic_tail;
34      volatile uint8_t basic_next[NUM_TASKSBASIC];
        volatile uint8_t radio_head;
36      volatile uint8_t radio_tail;
        volatile uint8_t radio_next[NUM_TASKSRADIO];
38      volatile uint8_t serial_head;
        volatile uint8_t serial_tail;
40      volatile uint8_t serial_next[NUM_TASKSSERIAL];
        volatile uint8_t sense_head;
42      volatile uint8_t sense_tail;
        volatile uint8_t sense_next[NUM_TASKSSENSE];
44
        #ifdef SIM__
46      // Aqui entram as funcoes responsaveis pelos eventos do simulador
        // As tasks sao simuladas por eventos no TOSSIM
48
        bool sim_scheduler_event_pending = FALSE;
50      sim_event_t sim_scheduler_event;

52      int sim_config_task_latency() {return 100;}


54
        /* Only enqueue the event for execution if it is
56          not already enqueued. If there are more tasks in the
            queue, the event will re-enqueue itself (see the handle
58          function). */

60      void sim_scheduler_submit_event() {
            if (sim_scheduler_event_pending == FALSE) {
62              sim_scheduler_event.time = sim_time() + sim_config_task_latency();
                sim_queue_insert(&sim_scheduler_event);
64              sim_scheduler_event_pending = TRUE;
```

```
        }
66  }

68  void sim_scheduler_event_handle(sim_event_t* e) {
        sim_scheduler_event_pending = FALSE;
70
        // If we successfully executed a task, re-enqueue the event. This
72      // will always succeed, as sim_scheduler_event_pending was just
        // set to be false.  Note that this means there will be an extra
74      // execution (on an empty task queue). We could optimize this
        // away, but this code is cleaner, and more accurately reflects
76      // the real TinyOS main loop.

78      if (call Scheduler.runNextTask()) {
            sim_scheduler_submit_event();
80      }
    }
82

84  /* Initialize a scheduler event. This should only be done
     * once, when the scheduler is initialized. */
86  void sim_scheduler_event_init(sim_event_t* e) {
        e->mote = sim_node();
88      e->force = 0;
        e->data = NULL;
90      e->handle = sim_scheduler_event_handle;
        e->cleanup = sim_queue_cleanup_none;
92  }
    #endif
94
    // Helper functions (internal functions) intentionally do not have atomic
96  // sections.  It is left as the duty of the exported interface functions to
    // manage atomicity to minimize chances for binary code bloat.
98
    /*****************************
100 *** Funcoes Task Basic ******
    *****************************/
102 inline uint8_t popTaskBasic()
    {
104     dbg("Prioridade", "Poped a Mtask (ou nao)\n");
            if( basic_head != NO_TASK )
106         {
                uint8_t id = basic_head;
108             basic_head = basic_next[basic_head];
                if( basic_head == NO_TASK )
110             {
                    basic_tail = NO_TASK;
```

```
112                 }
                    basic_next[id] = NO_TASK;
114                 return id;
                }
116             else
                {
118                 return NO_TASK;
                }
120     }

122     bool isWaitingBasic( uint8_t id )
        {
124         dbg("Prioridade", "isWaitingBasic: %d\n", (basic_next[id] != NO_TASK) || (basic_tail ==
            return (basic_next[id] != NO_TASK) || (basic_tail == id);
126     }

128     bool pushTaskBasic( uint8_t id )
        {
130         dbg("Prioridade", "pushTaskBasic %i\n", (int)id);
            if( !isWaitingBasic(id) )
132         {
                if( basic_head == NO_TASK )
134             {
                    basic_head = id;
136                 basic_tail = id;
                }
138             else
                {
140                 basic_next[basic_tail] = id;
                    basic_tail = id;
142             }
                return TRUE;
144         }
            else
146         {
                return FALSE;
148         }
        }
150
        /***************************
152     *** Funcoes Task Radio ******
        ***************************/
154     inline uint8_t popTaskRadio()
        {
156         dbg("Prioridade", "Poped a task Radio (ou nao)\n");
                if( radio_head != NO_TASK )
158             {
```

```
                uint8_t id = radio_head;
160             radio_head = radio_next[radio_head];
                if( radio_head == NO_TASK )
162             {
                    radio_tail = NO_TASK;
164             }
                radio_next[id] = NO_TASK;
166             return id;
            }
168         else
            {
170             return NO_TASK;
            }
172     }

174     bool isWaitingRadio( uint8_t id )
        {
176         dbg("Prioridade", "isWaitingRadio: %d\n", (radio_next[id] != NO_TASK) || (radio_tail ==
            return (radio_next[id] != NO_TASK) || (radio_tail == id);
178     }

180     bool pushTaskRadio( uint8_t id )
        {
182         dbg("Prioridade", "pushTaskRadio %i\n", (int)id);
            if( !isWaitingRadio(id) )
184         {
                if( radio_head == NO_TASK )
186             {
                    radio_head = id;
188                 radio_tail = id;
                }
190             else
                {
192                 radio_next[radio_tail] = id;
                    radio_tail = id;
194             }
                return TRUE;
196         }
            else
198         {
                return FALSE;
200         }
        }
202

        /*************************
204     *** Funcoes Task Serial ******
        *************************/
```

```
206     inline uint8_t popTaskSerial()
        {
208         dbg("Prioridade", "Poped a task Serial (ou nao)\n");
            if( serial_head != NO_TASK )
210         {
                uint8_t id = serial_head;
212             serial_head = serial_next[serial_head];
                if( serial_head == NO_TASK )
214             {
                    serial_tail = NO_TASK;
216             }
                serial_next[id] = NO_TASK;
218             return id;
            }
220         else
            {
222             return NO_TASK;
            }
224     }

226     bool isWaitingSerial( uint8_t id )
        {
228         dbg("Prioridade", "isWaitingSerial: %d\n", (serial_next[id] != NO_TASK) || (serial_tail
            return (serial_next[id] != NO_TASK) || (serial_tail == id);
230     }

232     bool pushTaskSerial( uint8_t id )
        {
234         dbg("Prioridade", "pushTaskSerial %i\n", (int)id);
            if( !isWaitingSerial(id) )
236         {
                if( serial_head == NO_TASK )
238             {
                    serial_head = id;
240                 serial_tail = id;
                }
242             else
                {
244                 serial_next[serial_tail] = id;
                    serial_tail = id;
246             }
                return TRUE;
248         }
            else
250         {
                return FALSE;
252         }
```

```
    }
254
    /**************************
256 *** Funcoes Task Sense ******
    **************************/
258 inline uint8_t popTaskSense()
    {
260     dbg("Prioridade", "Poped a task Sense (ou nao)\n");
        if( sense_head != NO_TASK )
262     {
            uint8_t id = sense_head;
264         sense_head = sense_next[sense_head];
            if( sense_head == NO_TASK )
266         {
                sense_tail = NO_TASK;
268         }
            sense_next[id] = NO_TASK;
270         return id;
        }
272     else
        {
274         return NO_TASK;
        }
276 }

278 bool isWaitingSense( uint8_t id )
    {
280     dbg("Prioridade", "isWaitingSense: %d\n", (sense_next[id] != NO_TASK) || (sense_tail ==
        return (sense_next[id] != NO_TASK) || (sense_tail == id);
282 }

284 bool pushTaskSense( uint8_t id )
    {
286     dbg("Prioridade", "pushTaskSense %i\n", (int)id);
        if( !isWaitingSense(id) )
288     {
            if( sense_head == NO_TASK )
290         {
                sense_head = id;
292             sense_tail = id;
            }
294         else
            {
296             sense_next[serial_tail] = id;
                sense_tail = id;
298         }
            return TRUE;
```

```
300         }
            else
302         {
                return FAILSE;
304         }
        }
306

308     command void Scheduler.init()
        {
310         dbg("Prioridade", "init\n");
            atomic
312         {
                memset( (void *)basic_next, NO_TASK, sizeof(basic_next) );
314             basic_head = NO_TASK;
                basic_tail = NO_TASK;
316
                memset( (void *)radio_next, NO_TASK, sizeof(radio_next) );
318             radio_head = NO_TASK;
                radio_tail = NO_TASK;
320
                memset( (void *)serial_next, NO_TASK, sizeof(serial_next) );
322             serial_head = NO_TASK;
                serial_tail = NO_TASK;
324
                memset( (void *)sense_next, NO_TASK, sizeof(sense_next) );
326             sense_head = NO_TASK;
                sense_tail = NO_TASK;
328
                #ifdef SIM__
330             sim_scheduler_event_pending = FALSE;
                sim_scheduler_event_init(&sim_scheduler_event);
332             #endif
            }
334     }

336     command bool Scheduler.runNextTask()
        {
338         uint8_t nextTask;
            dbg("Prioridade", "runNextTask\n");
340         atomic
            {
342             nextTask = popTaskSerial();
                dbg("Prioridade", "popTaskSerial: %i\n", (int)nextTask);
344
                if( nextTask == NO_TASK )
346             {
```

```
                nextTask = popTaskRadio();
348             dbg("Prioridade", "popTaskRadio: %i\n", (int)nextTask);

350             if (nextTask == NO_TASK)
                {
352                 nextTask = popTaskSense();
                    dbg("Prioridade", "popTaskSense: %i\n", (int)nextTask);
354                 if (nextTask == NO_TASK)
                    {
356                     nextTask = popTaskBasic();
                        dbg("Prioridade", "popTaskBaic: %i\n", (int)nextTask);
358
                        if (nextTask == NO_TASK) {
360                         return FALSE;
                        }
362                 dbg("Prioridade", "Running task basic %i\n", (int)nextTask);
                    signal TaskBasic.runTask[nextTask]();
364                 return TRUE;
                    }
366
                    dbg("Prioridade", "Running task sense %i\n", (int)nextTask);
368                 signal TaskSense.runTask[nextTask]();
                    return TRUE;
370             }
                dbg("Prioridade", "Running task radio %i\n", (int)nextTask);
372             signal TaskRadio.runTask[nextTask]();
                return TRUE;
374         }
        }
376     dbg("Prioridade", "Running task serial %i\n", (int)nextTask);
        signal TaskSerial.runTask[nextTask]();
378     return TRUE;
    }
380
    command void Scheduler.taskLoop()
382 {
        dbg("Prioridade", "Taskloop\n");
384     for (;;)
        {
386         uint8_t nextTaskBasic = NO_TASK;
            uint8_t nextTaskRadio = NO_TASK;
388         uint8_t nextTaskSerial = NO_TASK;
            uint8_t nextTaskSense = NO_TASK;
390
            atomic
392         {
                while (
```

```
394                     (nextTaskSerial = popTaskSerial()) == NO_TASK &&
                        (nextTaskRadio  = popTaskRadio() ) == NO_TASK &&
396                     (nextTaskSense  = popTaskSense() ) == NO_TASK &&
                        (nextTaskBasic  = popTaskBasic() ) == NO_TASK )
398                 {
                        call McuSleep.sleep();
400                 }
                }
402             if (nextTaskSerial != NO_TASK) {
                    dbg("Prioridade", "Running task serial %i\n", (int)nextTaskSerial);
404                 signal TaskSerial.runTask[nextTaskSerial]();
                }
406             else if (nextTaskRadio != NO_TASK) {
                    dbg("Prioridade", "Running task radio %i\n", (int)nextTaskRadio);
408                 signal TaskRadio.runTask[nextTaskRadio]();
                }
410             else if (nextTaskSense != NO_TASK) {
                    dbg("Prioridade", "Running task sense %i\n", (int)nextTaskSense);
412                 signal TaskSense.runTask[nextTaskSense]();
                }
414             else if (nextTaskBasic != NO_TASK) {
                    dbg("Prioridade", "Running task basic %i\n", (int)nextTaskBasic);
416                 signal TaskBasic.runTask[nextTaskBasic]();
                }
418         }
        }
420

422     /**
         * Return SUCCESS if the post succeeded, EBUSY if it was already posted.
424      */

426     async command error_t TaskBasic.postTask[uint8_t id]()
        {
428         error_t result;

430         dbg("Prioridade", "postTaskBasic\n");
            atomic {
432             result = pushTaskBasic(id) ? SUCCESS : EBUSY;
            }
434         #ifdef SIM__
            if (result == SUCCESS)
436             sim_scheduler_submit_event();
            #endif
438

            return result;
440     }
```

```
442     default event void TaskBasic.runTask[uint8_t id]()
        {
444     }

446     async command error_t TaskSerial.postTask[uint8_t id]()
        {
448         error_t result;

450         dbg("Prioridade", "postTaskSerial\n");
            atomic {
452             result = pushTaskSerial(id) ? SUCCESS : EBUSY;
            }
454         #ifdef SIM__
            if (result == SUCCESS)
456             sim_scheduler_submit_event();
            #endif
458
            return result;
460     }

462     default event void TaskSerial.runTask[uint8_t id]()
        {
464     }

466     async command error_t TaskRadio.postTask[uint8_t id]()
        {
468         error_t result;

470         dbg("Prioridade", "postTaskRadio\n");
            atomic {
472             result = pushTaskRadio(id) ? SUCCESS : EBUSY;
            }
474         #ifdef SIM__
            if (result == SUCCESS)
476             sim_scheduler_submit_event();
            #endif
478
            return result;
480     }

482     default event void TaskRadio.runTask[uint8_t id]()
        {
484     }

486     async command error_t TaskSense.postTask[uint8_t id]()
        {
```

```
488         error_t result;

490         dbg("Prioridade", "postTaskSense\n");
            atomic {
492             result = pushTaskSense(id) ? SUCCESS : EBUSY;
            }
494         #ifdef SIM__
            if (result == SUCCESS)
496             sim_scheduler_submit_event();
            #endif
498
            return result;
500     }

502     default event void TaskSense.runTask[uint8_t id]()
        {
504     }
}
```

Referências

[1] J. Hill, R. Szewczyk, A. Woo, S. Hollar, D. Culler, and K. Pister. System architecture directions for networked sensors. In *9th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 93–104, New York, NY, USA, 2000. ACM Press.

[2] David Gay, Phil Levis, Rob von Behren, Matt Welsh, Eric Brewer, and David Culler. The nesC language: A holistic approach to networked embedded systems. *Proceedings of Programming Language Design and Implementation (PLDI)*, June 2003.

[3] Philip Levis and David Gay. *TinyOS Programming*. Cambridge University Press, 2009.

[4] Philip Levis and Cory Sharp. Tep106: Schedulers and tasks. http://www.tinyos.net/tinyos-2.1.0/doc/html/tep106.html.

[5] Philip Levis. Tep107: Tinyos 2.x boot sequence. http://www.tinyos.net/tinyos-2.1.0/doc/html/tep107.html.

[6] William Stallings. *Operating Systems: Internals and Design Principles*. Prentice Hall, 5a edition, 2004.

[7] Ana L. de Moura. *Revisitando co-rotinas*. PhD thesis, PUC-Rio, Rio de Janeiro, Brasil, 2004.

[8] Kevin Klues, Chieh-Jan Liang, Jeongyeup Paek, Razvan Musaloiu-E, Ramesh Govindan, Andreas Terzis, and Philip Levis. Tep134: The tosthreads thread library. http://www.tinyos.net/tinyos-2.1.0/doc/html/tep134.html.