Universidade Federal de Santa Catarina
Centro de Blumenau
Departamento de Engenharia de
Controle e Automação e Computação

Henrique Marchi Lange

Desenvolvimento de um RTOS para Processadores da Arquitetura *Intel* 8052

Blumenau
2021

Henrique Marchi Lange

Desenvolvimento de um RTOS para Processadores da Arquitetura *Intel* 8052

Trabalho de Conclusão de Curso apresentado à Universidade Federal de Santa Catarina como parte dos requisitos necessários para a obtenção do Título de Engenheiro de Controle e Automação.
Orientador: Prof. Dr. Carlos R. Moratelli

Universidade Federal de Santa Catarina
Centro de Blumenau
Departamento de Engenharia de
Controle e Automação e Computação

Blumenau
2021

Henrique Marchi Lange

Desenvolvimento de um RTOS para Processadores da Arquitetura *Intel* 8052

Trabalho de Conclusão de Curso apresentado à Universidade Federal de Santa Catarina como requisito parcial para a obtenção do título de Engenheiro de Controle e Automação.

Comissão Examinadora

Documento assinado digitalmente
Carlos Roberto Moratelli
Data: 21/05/2021 15:13:49-0300
CPF: 035.999.369-94
Verifique as assinaturas em https://v.ufsc.br

Prof. Dr. Carlos R. Moratelli
Universidade Federal de Santa Catarina
Orientador

Documento assinado digitalmente
Adao Boava
Data: 20/05/2021 21:09:14-0300
CPF: 645.640.969-15
Verifique as assinaturas em https://v.ufsc.br

Prof. Dr. Adão Boava
Universidade Federal de Santa Catarina

Documento assinado digitalmente
Fabio Rafael Segundo
Data: 21/05/2021 10:37:43-0300
CPF: 901.593.909-82
Verifique as assinaturas em https://v.ufsc.br

Prof. Dr. Fabio Rafael Segundo
Universidade Federal de Santa Catarina

Blumenau, 20 de maio de 2021

A todos que fizeram parte desta jornada, dedico este trabalho.

Agradecimentos

Primeiramente agradeço ao meu pai, o homem que me introduziu ao mundo da engenharia desde meus primeiros anos de educação. Agradeço a minha mãe por toda a paciência no processo de me educar e por todo o suporte durante os anos cruciais de minha vida. Também tenho imensa gratidão por toda a ajuda que recebi de meu irmão no decorrer dos anos e por ter sido um guia, trazendo a luz da experiência para iluminar os caminhos pelos quais passei.

Deixo meus agradecimentos aos amigos Alessandro Girardi, Jorge Buzzi, Lucas Tomio, Alan Grava e Alan Cabral que estiveram presentes desde o início da minha jornada pela vida acadêmica. Igualmente sou grato pelos anos em que trabalhei junto com meus grandes amigos Felipe Pegoretti e Gabriel Dinse na universidade.

Tenho imenso respeito e gratidão à Universidade Federal de Santa Catarina e todos os seus servidores por prover os meios necessários para que eu obtivesse a minha formação. Em especial, gostaria de enfatizar minha gratidão aos professores Adão Boava e Carlos Roberto Moratelli, que orientaram minhas pesquisas durante meus anos na academia.

Por fim, mas não menos importante, deixo meus agradecimentos a todas as pessoas que estiveram presentes de alguma forma nos últimos cinco anos de minha vida. Sua passagem, sendo curta ou longa, ajudou a moldar a pessoa que sou agora e a definir os caminhos que tomei para chegar neste ponto.

"Se vi mais longe, é porque me apoiei nos ombros de gigantes"
(Isaac Newton)

Resumo

Sistemas operacionais de tempo real (RTOS's) estão muito presentes nos sistemas embarcados atuais, pelo fato de serem compactos e conseguirem atender as exigências temporais de projetos onde o não cumprimento dessas pode acarretar em desastres. Contudo, muitas arquiteturas de microcontroladores ainda não possuem suporte dos principais RTOS's do mercado, ou só possuem suporte de ferramentas pagas, muitas vezes com um alto valor agregado. No presente trabalho foi desenvolvido um RTOS para a família de processadores baseados na arquitetura *intel 8051* totalmente *opensource*. Assim como o sistema operacional, as ferramentas de compilação e edição de código utilizadas são *opesource* ou gratuitas. Para validar o sistema obtido foram criadas aplicações testes envolvendo problemas computacionais comuns, sendo estas divididas em um programa baseado no problema do produtor consumidor, um programa com variáveis sendo manipuladas por várias tarefas para verificar a exclusão mútua e, por fim, um *firmware* para controladores de peso, de modo a testar o sistema em aplicações mais complexas. Todos os programas foram executados em uma placa baseada no microcontrolador ADuC847, verificando se o sincronismo entre tarefas e o escalonamento preemptivo estavam ocorrendo conforme o esperado. Ao final dos testes constatou-se que o sistema atendeu às demandas, com pequenos *footprints* de memória e código, possuindo um código fonte compacto.

Palavras-Chave: 1. Sistema Operacional. 3. Sistema Operacional de Tempo Real. 3. Intel 8052. 4. Linguagem C. 5. OpenSource.

Abstract

Real-time operating systems (RTOS's) are very present in today's embedded systems, due to the fact that they are compact and able to meet the temporal requirements of projects where failure to comply with them can lead to disasters. However, many microcontroller architectures still do not have support from the main RTOS's on the market, or only support paid tools, often with a high added value. In the present work, a fully *opensource* RTOS was developed for the processor family based on the textit intel 8051 architecture. Like the operating system, the code compilation and editing tools used are either textit opesource or free. To validate the system obtained, test applications were created involving common computational problems, which were divided into a program based on the consumer producer problem, a program with variables being manipulated by various tasks to verify mutual exclusion and, finally, a textit firmware for checkweighers in order to test the system in more complex applications. All programs were executed on a board based on the ADuC847 microcontroller, checking if the synchronization between tasks and preemptive scheduling were occurring as expected. At the end of the tests it was found that the system met the demands, with small textit footprints of memory and code, having a compact source code.

Keywords: 1. Operating System. 2. Real Time Operating System. 3. Intel 8052. 4. C Language. 5. OpenSource.

Lista de figuras

Lista de tabelas

Lista de Siglas e Abreviaturas

UFSC	*Universidade Federal de Santa Catarina*
RTOS	*Real Time Operating System*
GPOS	*General Porpouse Operating System*
I/O	*Input and Output*
IDE	*Integrated Development Environment*
CPU	*Central Processing Unit*
SO	*Sistema Operacional*
RM	*Rate Monotonic*
FPGA	*Field-programmable gate array*
AD	*Analog to Digital*
SFR	*Special Function Register*
RAM	*Random-Access Memory*
UART	*Universal asynchronous receiver/transmitter*
DPTR	*Data Pointer*
ACC	*Acomulador*
SP	*Stack Pointer*
PSW	*Program Status Word*
API	*Application Program Interface*
ISR	*Interruption Service Routine*
FCFS	*First Come First Served*
IP	*Internet Protocol*
ROM	*Read-Only Memory*
TCB	*Task Control Block*

Sumário

1 Introdução

O avanço dos microcontroladores culminou, nas últimas décadas, em seu uso na grande maioria dos dispositivos utilizados pelo ser humano. Atualmente nossos carros, geladeiras, celulares, televisões, aparelhos de som e vários outros sistemas possuem pequenas placas de circuito impresso que controlam estes dispositivos através de um microcontrolador.

Com essa nova tendência, ficou comum ouvirmos falar em sistemas embarcados [1]. Essa classe de sistemas é caracterizada por sua especialização. Em essência, um sistema embarcado é desenvolvido para resolver um problema específico. Contudo, os desafios no desenvolvimento de um novo sistema embarcado são diversos, entre eles está a escolha dos componentes de *hardware*, tamanho do sistema, número de entradas e saídas, linguagem de programação a ser utilizada, sistema operacional, etc.

Muitos sistemas embarcados atuais utilizam um sistema operacional como base. O papel principal de um sistema operacional está em gerenciar os recursos de *hardware* de maneira eficiente [2], abstraindo tais recursos para o desenvolvedor, que muitas vezes não lida com o *hardware* diretamente. Comumente, desenvolvedores e engenheiros utilizam o *kernel* Linux [3] como base para seu sistema.

Apesar de amplamente utilizado, o Linux é um sistema operacional de propósito geral (GPOS) e não foi projetado para cumprir requisitos de tempo real, além de ser um sistema que exige requisitos de *hardware* muitas vezes ausentes em microcontroladores utilizados para controle de processos, gerência de memória, entre outros. Neste cenário surgem os sistemas operacionais de tempo real como uma alternativa mais leve e capaz de cumprir com os requisitos temporais do sistema.

Segundo os dados encontrados em [4], 18% das empresas entrevistadas utilizam o sistema operacional de tempo real FreeRTOS [5], ficando atrás apenas do Linux embarcado (21%) e de soluções proprietárias (19%). Esse número torna-se ainda mais expressivo se considerarmos que uma grande fatia das soluções proprietárias também são RTOS's e que outros RTOS's como o Keil RTX [6] também estão na lista.

Outros dados importantes revelados pela pesquisa estão nas linguagens de programação utilizadas, os requisitos mais procurados em sistemas operacionais e os fatores decisivos na escolha de um sistema operacional. As linguagens mais utilizadas são C e C++, obtendo juntas 79% do mercado total, entre os requisitos procurados destaca-se a capacidade de tempo real do sistema operacional (42% dos usuários procuram isso em um SO embarcado). Ainda, dentro dos fatores decisivos para escolha do SO, 35% dos usuários procuram sistemas com o código fonte disponível (fator mais procurado dentre os citados na pesquisa).

Com estes dados pode-se extrair a informação de que um sistema operacional de

tempo real totalmente *opensource* e feito em linguagem C ou C++ possui uma grande procura dentro do mercado de sistemas embarcados. Somado a isso, temos o fato de que arquiteturas mais antigas de microprocessadores não suportadas pelos RTOS's mais famosos do mercado ainda são amplamente utilizadas em chips de fabricantes como a Analog Devices e Texas Instruments. Logo, torna-se interessante desenvolver um sistema operacional de tempo real para arquiteturas como a do Intel 8052.

1.1 Objetivos do projeto

Conforme observado anteriormente, no mercado de sistemas embarcados existe um espaço para sistemas operacionais de tempo real focados em arquiteturas mais antigas baseado na linguagem de programação C. Por isso, este trabalho terá como foco desenvolver um RTOS compacto para a arquitetura Intel 8052.

A arquitetura escolhida neste trabalho deveu-se a dois fatores principais, sendo eles o fato de que durante o estágio da graduação na empresa MPA foi utilizado um microcontrolador ADuC847 [7] baseado no 8052 e também o fato de que esta arquitetura possui carência de ferramentas gratuitas ou *opensource* para desenvolvimento.

Ainda, apesar de ser adotada a linguagem C como principal linguagem de programação do projeto, será necessário utilizar pequenas partes de código em Assembly, uma vez que a manipulação da pilha e dos registradores do processador somente é possível em baixo nível.

Sendo assim, os principais objetivos levantados para o trabalho são:

1. Desenvolver um escalonador em linguagem C, com pequenos trechos de código em Assembly;
2. Desenvolver mecanismos de sincronismo, em essência semáforos e *mutexes*;
3. Trabalhar com o compilador SDCC [8] e ferramentas de edição de código gratuitas;
4. Desenvolver uma aplicação de demonstração, utilizando os principais recursos do sistema operacional.

1.2 Estrutura do documento

Este capítulo encarregou-se de explanar o assunto de maneira breve, situando o leitor quanto ao tema e aos fatores que motivaram o desenvolvimento do trabalho. Também foram apresentados dados que comprovam a importância do mesmo para o mercado de sistemas embarcados. Os demais capítulos estão resumidos abaixo.

- No capítulo 2 é dado um *background* sobre o tema central do trabalho. Nele serão explicadas as características gerais de sistemas operacionais modernos, as principais classes de sistemas operacionais e suas diferenças, os modelos de *kernel* mais utilizados e suas vantagens e desvantagens, o funcionamento das principais ferramentas de um sistema operacional e, por fim, será feita uma introdução à arquitetura de processadores Intel 8052;
- No capítulo 3 é apresentado o estado da arte, fazendo uma breve discussão sobre o principal RTOS hoje presente no mercado, seu funcionamento e limitações. O capítulo também aborda soluções específicas para a arquitetura escolhida e o porquê delas não conseguirem preencher a lacuna existente hoje no mercado;
- O capítulo 4 trata dos requisitos do projeto. Nele são detalhadas as características desejadas no sistema obtido e o porquê delas terem sido escolhidas em detrimento de outras características.
- O capítulo 5 trás todos os dados de desenvolvimento do projeto. Nele serão discutidos os algoritmos adotados e a escolha das ferramentas de trabalho utilizada;
- No capítulo 6 serão mostrados os resultados obtidos em testes com o sistema operacional final, bem como o tamanho final do sistema, quais objetivos foram alcançados e um exemplo de aplicação suprida pelo RTOS;
- O capítulo 7 trata-se da conclusão, onde os resultados finais são analisados, assim como postas as perspectivas de trabalhos futuros.

2 Background

Para um melhor entendimento deste trabalho, primeiro precisamos ter em mente os conceitos gerais sobre os temas abordados. Neste capítulo serão apresentados os principais conceitos acerca dos temas a serem discutidos nos capítulos que seguem. As próximas seções percorrerão sobre os temas programação *bare metal*, linguagens C e Assembly, arquiteturas de *kernel* de sistemas operacionais, sistemas operacionais de propósito geral e de tempo real, escalonamento, sincronização de *threads* e processos e, por fim, discutiremos as características da plataforma utilizada no trabalho.

2.1 Programação Bare Metal

Apesar dos sistemas operacionais servirem como base para um grande número de dispositivos embarcados atualmente, até este momento existem sistemas em que os desenvolvedores optam por não utilizar nenhuma espécie de sistema operacional. Nestes sistemas é utilizada a programação *bare metal*.

Na programação *bare metal* o engenheiro responsável pelo projeto tem a responsabilidade de implementar todas as funcionalidades do mesmo manualmente. Caso o projeto precise lidar com entradas e saídas o engenheiro deverá ler a documentação do microprocessador utilizado e programar os bits dos registradores encarregados, o mesmo ocorre para comunicação *serial*, leitura de conversores AD, escrita e leitura de memória e todas as outras funções presentes no sistema.

O desafio torna-se maior em casos onde o sistema embarcado precise realizar comunicação com algum protocolo de rede. Vamos supor que o engenheiro precise implementar o protocolo de comunicação industrial Modbus [9], neste caso fica a cargo dele ler toda a documentação e as especificações do protocolo para implementá-lo manualmente [10].

Entretanto, é comum vermos aplicações simples sendo implementadas em *bare metal*, um bom exemplo seria uma balança para pesar alimentos: como o sistema precisará apenas ler um sensor e mostrar a leitura na tela, não faz sentido utilizar um sistema operacional, liberando o desenvolvedor para utilizar um microprocessador mais simples e barato no projeto.

Mesmo não sendo um trabalho trivial em um primeiro momento, muitas plataformas oferecem bibliotecas prontas com funções para manipulação de recursos do processador, como o caso da plataforma Arduino [11]. Com efeito, tais plataformas não incentivam o desenvolvedor a manipular os recursos através da escrita em registradores, pelo fato que esta prática requer um grande tempo de estudo acerca do processador sendo utilizado no projeto.

Também, mesmo para processadores mais antigos como o 8051, existem plataformas ricas em recursos de desenvolvimento e depuração como o famoso IDE µVision da Keil [12]. Estes ambientes geram códigos Assembly altamente otimizados, além de fazerem uma análise completa no código fonte e disponibilizar outras ferramentas para facilitar e agilizar o desenvolvimento de programas em linguagem C ou C++. Porém, o preço a ser pago para ter acesso a tais funcionalidades muitas vezes ultrapassa o orçamento de pequenos projetos.

Felizmente, compiladores *opensource* e gratuitos como o SDCC [8] estão presentes no mercado, com uma rica biblioteca de funções que vão de comunicação serial a complexas operações matemáticas, seguindo ao máximo o padrão C-ANSI [13]. Este compilador pode ser usado via linha de comando, assim como está integrado em IDE's gratuitos como o MIDE-51, possuindo recursos de simulação e emulação, dentre outros [14].

Existem duas maneiras para programar um microcontrolador diretamente, manipulando seus endereços de memória e registradores, sendo elas o uso da linguagem C [13] ou da linguagem Assembly [15]. As sub-sessões a seguir introduzem o leitor a estas duas linguagens e suas principais características.

2.1.1 Assembly

A linguagem Assembly é o nível mais baixo de programação, estando apenas acima da escrita direta em endereços de memória, também chamada de linguagem de máquina, que pode ser alcançado. Cada microprocessador possui sua própria linguagem Assembly, pois ela se trata de um conjunto de instruções para manipulação de recursos, em especial endereços de memória e registradores [15].

Atualmente, esta linguagem possui seu uso apenas em soluções específicas, ficando no 11º lugar do famoso *ranking* da empresa TIOBE [16], em janeiro do ano de 2021. Isso se deve ao fato de que linguagens como C possuírem compiladores capazes de gerar códigos tão otimizados quanto códigos Assembly e ao mesmo tempo mais portáveis para outras arquiteturas de processadores [17].

Todavia, existem projetos feitos completamente em Assembly por precisarem extrair ao máximo os recursos de *hardware* em um sistema limitado, ou por lidarem com arquiteturas que não possuem um amplo suporte de compiladores e ferramentas para a linguagem C. Não obstante, muitos projetos *bare metal* feitos em C necessitam de trechos de código em Assembly, uma vez que determinados recursos dos microcontroladores, como a pilha, só podem ser acessados através de instruções Assembly.

Apesar das dificuldades envolvidas no desenvolvimento através de instruções Assembly, a linguagem nos oferece acesso direto a todos os recursos dos processadores e, se bem dominada, é uma ferramenta poderosa na criação de sistemas complexos que utilizam processadores mais limitados e baratos [18]. O código 2.1 mostra um exemplo de código

com instruções Assembly para o famoso processador Intel 8086.

```
1
2  move_cima:
3      out  leds,al
4
5  move_cima0:
6      test al,bl              ;verifica se elevador chegou
7      jnz  fim_cima           ;vai pro fim caso o elevador chegou ao andar
8      mov  cx,20
9
10 atraso:                 ;atraso para melhorar a animacao leds
11     cmp  panico,1
12     je   verifica_panico
13     loop atraso
14     push ax             ;salva na pilha valor em ax e bx
15     push bx
16     mov  bx,offset rot_h     ;rotacao sentido horario
17     call gira_motor        ;chama sub-rotina_3
18     pop  bx                  ;retira da pilha bx e ax
19     pop  ax
20     ror  al,1
21     jmp  move_cima           ;volta para o inicio
22
23 fim_cima:                    ;volta para verifacar presenca
24     jmp  verifica_presenca
```

Código 2.1 – Código Assembly para o microprocessador Intel 8086. Fonte: O autor

2.1.2 C Bare Metal

A linguagem C é muito conhecida e amplamente utilizada. Segundo o *ranking* da empresa TIOBE citado na seção 2.1.1, a linguagem C está em primeiro lugar em termos de uso no ano de 2021. Apesar do surgimento de linguagens mais modernas como Java ou Python, ela continua no topo da lista ano após ano. Segundo Maurício Sucheuski (1996) [13]:

> "A linguagem C é uma linguagem que permite ao desenvolvedor criar softwares de alto desempenho sem necessariamente recorrer a linguagem de baixo nível (Assembly).Ao mesmo tempo permite trabalhar em um nível de abstração elevado como nas linguagens de última geração."

Devido a esta característica, a linguagem C é atualmente a linguagem mais utilizada no desenvolvimento de sistemas embarcados, conforme visto no capítulo 1. Também, com o avanço dos compiladores modernos, o seu uso foi potencializado, abrindo um mercado para a linguagem em sistemas com baixo poder de processamento [8], sendo que a geração

de códigos Assembly por parte dos compiladores resulta em códigos mais otimizados do que os obtidos pelos programadores e engenheiros com pouco conhecimento acerca do conjunto de instruções dos processadores.

Muitas vezes citada na literatura como uma linguagem de alto nível, a abstração encontrada na linguagem C é muito menor do que em outras linguagens modernas como Python, por ela possibilitar que o desenvolvedor acesse regiões específicas de memória e registradores diretamente através do uso de ponteiros [19].

Quando falamos de C *bare metal*, fica a cargo dos desenvolvedores programar os bits dos registradores do microprocessador responsáveis pelo acesso às funcionalidades do *hardware* através da escrita em endereços de memória [17]. Uma vez configurados os registradores, o projeto torna-se simples e semelhante à programação em linguagem C com um sistema operacional por baixo. No Código 2.2 temos um exemplo de como registradores de configuração são programados em linguagem C.

```
1  // CONFIGURACAO DA SERIAL:
2    SCON = 0x53;  //MODO 1;
3    // Configurando timer 3 para 9600 de baud rate:
4    PLLCON = 0x48;  //Freq 12.582912.
5    T3CON = 0x85;
6    T3FD = 0x12;
7    TR1 = 1;
8    TI = 1;
9    // Delay para gravar os valores nos registradores:
10   delay(700);
11
12   // INICIALIZACAO DO AD:
13   RDY0 = 0;
14   SF = 3;
15   ADC0CON1 = 0x20;  //Buffer enable, unipolar, +/- 2,56.
16   ADC0CON2 = 0x4A;  //REFIN +/-, Ain1 -> Ain2.
17   ADCMODE = 0x25;   //Gain calibration Ain1 -> AinCOM, sem chop.
18   ICON = 0;
19   delay (700);
20   ADCMODE = 0x33;   //Continuous conversion mode, chop enabled.
21   delay (700);
```

Código 2.2 – Configuração de registradores em linguagem C. Fonte: O autor

É muito comum que desenvolvedores criem sua própria pilha de funções C durante o desenvolvimento *bare metal*, de modo a abstrair as configurações feitas em baixo nível nos registradores e tornando o projeto como um todo mais portátil. A plataforma Arduino [20] é um exemplo muito popular neste cenário, tendo uma IDE proprietária onde desenvolvedores podem programar suas placas baseadas em *chips* Atmega diretamente em linguagem C ou C++ *bare metal* com um conjunto de funções disponibilizadas juntamente com o ambiente de desenvolvimento.

Outra técnica comumente usada em códigos em linguagem C *bare metal* são máquinas de estado [21], onde o desenvolvedor define os estados em que seu sistema pode estar e como dão-se as transições de um estado para o outro. Máquinas de estado são uma maneira eficiente e organizada de desenvolver sistemas que facilitam o entendimento do código e o posterior suporte para o mesmo.

Contudo, grandes projetos dificilmente são implementados em C *bare metal*, uma vez que este paradigma não consegue prover recursos de tempo real como em um RTOS nem escalonamento ao sistema e a implementação de recursos como comunicação através de protocolos modernos não é trivial, podendo gastar muito tempo da equipe de desenvolvimento [17]. Sendo assim, é comum que sistemas embarcados tenham como base um sistema operacional, seja ele um GPOS ou um RTOS.

2.2 Kernel de um sistema operacional

Antes de entrarmos no mérito de como os sistemas operacionais funcionam e são construídos, precisamos compreender o conceito de *kernel*. Em um SO o *kernel* é a parte responsável por todo o funcionamento do sistema e a abstração do *hardware* para o usuário, sendo chamado de núcleo do sistema [2].

Quando falamos de programação não *bare metal*, ou seja, com um sistema operacional gerenciando recursos, é indispensável compreender que existem dois modos de execução: O modo de usuário e o modo *kernel* [22]. No modo de usuário, os programas não possuem qualquer acesso ao *hardware* do sistema, evitando assim que um programa afete o desempenho e segurança do mesmo. Quando um programa deseja acessar os recursos do sistema, ele precisa fazer uma requisição para o SO tomar o comando e acessar o *hardware*, retornando para a aplicação a resposta obtida pela operação.

As requisições que um programa em modo usuário faz para o SO ocorrem via *syscalls*, conhecidas como chamadas de sistema. Funções comumente utilizadas como *printf* [13] não são nada além de chamadas de sistema. Ocorrendo uma chamada, o sistema operacional toma o controle e, neste momento, a execução passa a ocorrer em modo *kernel*. Note que, a aplicação em si não estará acessando os recursos diretamente, sendo assim o núcleo do sistema é sempre responsável por fazer a troca de informações entre processos e *hardware*. A Figura 1 ilustra a troca entre modo de usuário para modo de *kernel*, nela temos um processo sendo executado no modo de usuário. Quando o processo precisa fazer o uso de recursos de *hardware*, o sistema operacional é evocado, esta chamada altera o bit de modalidade, dando acesso ao *hardware* para o *kernel* do sistema. No momento em que o sistema acaba seus processamentos ele retorna os dados pertinentes para o processo, alterando novamente o bit de modalidade, de modo que o processo será executado no modo de usuário, não possuindo acesso direto ao *hardware*.

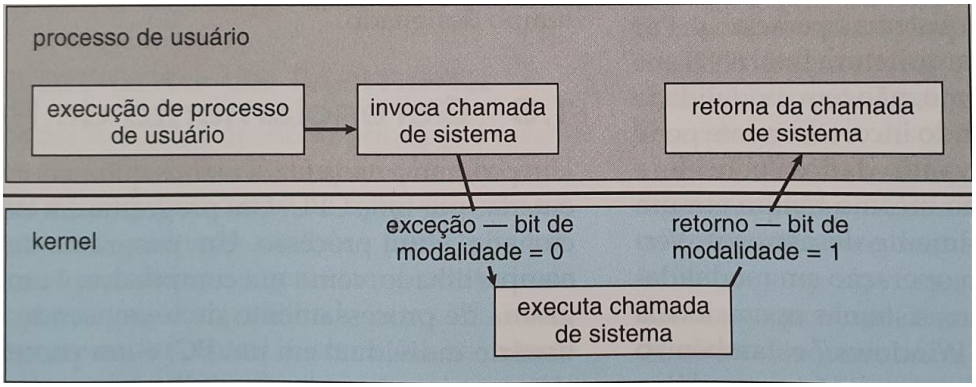

Figura 1 – Diferentes modos de execução em um sistema operacional. [Fonte: Silberschatz [2]]

O núcleo de um sistema operacional pode ser construído como *microkernel* ou *kernel* monolítico. Na primeira, o *kernel* é reduzido o máximo possível, tendo dentro de si apenas operações essenciais como gerência de memória e processos, outras funções como implementação de sistemas de arquivos e gerência de disco ficam no espaço de usuário. Já em um *kernel* monolítico ocorre o oposto, de modo que o tamanho do núcleo é consideravelmente maior. Vamos, agora, analisar a fundo as vantagens e desvantagens das duas implementações.

2.2.1 Kernel Monolítico

Conforme visto anteriormente, um *kernel* monolítico possui dentro de si um grande número de funções, indo de funções essenciais como gerência de processos até funções menos essenciais como comunicação com a rede [23]. Neste modelo o tamanho do núcleo acaba por ser consideravelmente maior do que em *micro kernels* e sua manutenção torna-se mais trabalhosa, com enfase, qualquer mudança no código como inserção de uma nova funcionalidade ou correção de *bug* obriga o desenvolvedor a compilar novamente todo o *kernel* [22]. A Figura 2 ilustra como um núcleo monolítico é construído.

Outros problemas deste tipo de construção são a quantidade de memória necessária para o *kernel*, dificuldades na manutenção, perda de desempenho e maior suscetibilidade a falhas. Em especial, a ultima desvantagem ocorre pelo fato de que, processos como leitura de memória secundária podem gerar uma série de erros e, por estarem na mesma área de memória que o resto do *kernel*, fazem com que todo o sistema pare de funcionar.

Devido a estes problemas, os desenvolvedores de sistemas operacionais de tempo real optam, em sua grande maioria, pelo uso de *microkernels*, sendo eles mais rápidos e confiáveis [24]. Entretanto, *kernels* monolíticos continuam sendo amplamente utilizados em sistemas de propósito geral, pois a implementação de um *microkernel* em sistemas muito grandes e complexos é inviável, conforme discutiremos na próxima seção.

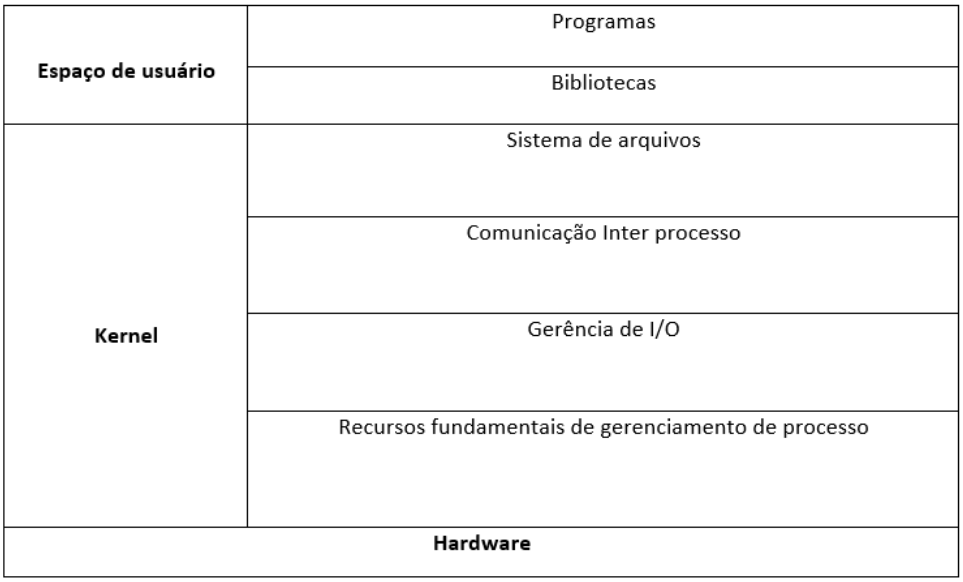

Figura 2 – Componentes de um *kernel* monolítico. [Fonte: Roch [22]]

2.2.2 MicroKernel

Na contramão dos *kernels* monolíticos, temos a arquitetura de *microkernel*, onde o desenvolvedor insere no núcleo apenas os elementos necessários para o funcionamento do sistema operacional. Nesta arquitetura funções como comunicação em rede, acesso a discos de memória secundária, sistemas de arquivos e várias outras que não são essenciais ao sistema ficam na área de memória dos processos de usuário.

Separar tais funcionalidades acaba por diminuir o tamanho de espaço ocupado pelo núcleo, aumentar a velocidade do mesmo e facilitar a manutenção do *kernel* [22]. Também vale observar que a modularidade desse tipo de arquitetura é maior, cabendo ao usuário optar por inserir elementos que ocupam muita memória como leituras de arquivos ou deixar seu sistema sem tais funcionalidades. Na Figura 3 temos a representação de como este tipo de *kernel* é construído.

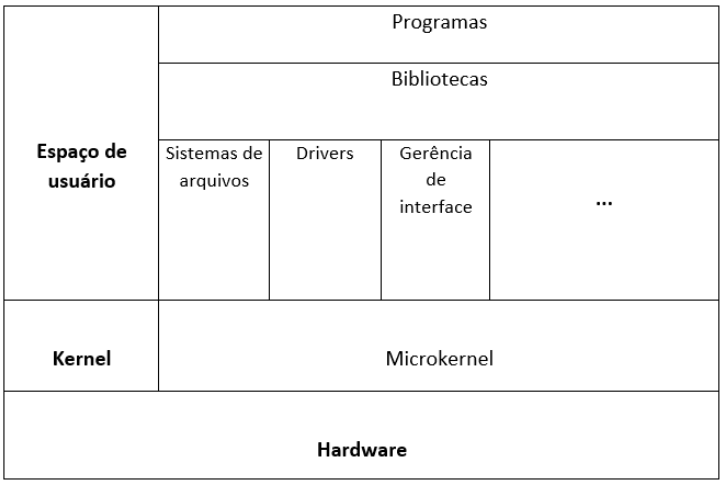

Figura 3 – Componentes de um *microkernel*. [Fonte: Roch [22]]

Para que um programa acesse funcionalidades do *kernel* que estão na área de usuário, adota-se um sistema de comunicação inter-processo [2], onde cada processo comunica-se com os demais, sendo o *kernel* do sistema o intermediário desta comunicação. Esta metodologia funciona da seguinte forma: O programa de usuário envia uma mensagem ao *kernel* requisitando acesso a determinada funcionalidade, este recebe a mensagem e envia ao processo correspondente. Quando este termina todo o processamento necessário uma mensagem é enviada ao *kernel* que, por fim, a encaminhara para o programa de usuário, conforme a Figura 4. Note que, neste exemplo, o processo requisita ao sistema a leitura de um arquivos. O *kernel* recebe a requisição e a encaminha para o processo responsável pela leitura de arquivos, este fará as tarefas necessárias, retornando ao *kernel* uma resposta que será repassada para o processo. Neste esquema, os processos não estão conversando diretamente entre si.

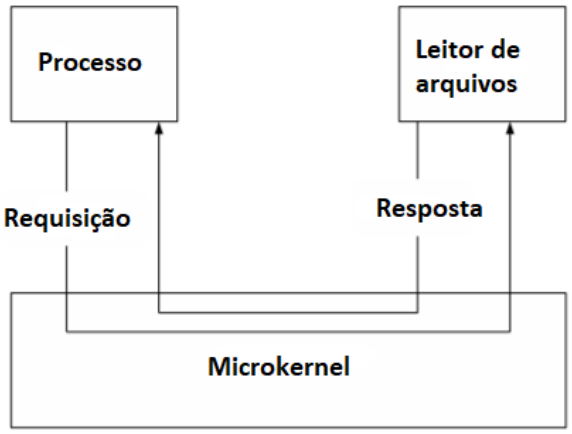

Figura 4 – Etapas da comunicação inter processos. [Fonte: Roch [22]]

Todavia, as vantagens da arquitetura ficam restritas a sistemas menos complexos, como os RTOS's [24]. Quando em sistemas de propósito geral, a implementação de um *microkernel* torna-se problemática, uma vez que o número de componentes se comunicando é muito grande e a chance de ocorrerem falhas nesta comunicação é imensa, gerando problemas de desempenho e, em alguns casos, falhas críticas do sistema. Tais falhas são dificilmente identificadas e sua depuração toma um grande tempo por parte de desenvolvedores [25]. Um exemplo disso está nos casos de sistemas de arquivos remotos comumente usados atualmente, uma vez que eles envolvem processos de leitura e escrita em disco e comunicação via rede. Ainda, tais processos podem ser divididos em uma série de processos menores, todos comunicando entre si. Neste exemplo nota-se que a probabilidade de ocorrer uma falha é considerável, por envolver muitos componentes se comunicando, e a dificuldade de encontrar tal falha é elevada.

2.3 Sistemas Operacionais de Propósito Geral

Segundo Linus Torvalds (2001) [25]:

> "a principal característica de um sistema operacional é que você nunca deveria notar a sua presença."

Esta afirmação não poderia estar mais correta. Efetivamente, usuários comuns nunca estão utilizando o sistema operacional de suas máquinas diretamente.

Quando o usuário abre seu computador, esteja ele rodando Linux, Windows ou MacOS, desde os primeiros instantes ele está em contato com as aplicações. O papel de um sistema operacional é fornecer meios de acesso ao *hardware* para estas aplicações, como acesso a memória principal e secundária, comunicação com periféricos via *drivers*, escrita na tela, leitura das entradas de texto, gerência de memória, etc [2].

Devido ao fato de o usuário comum não utilizar o sistema operacional diretamente, podemos nos questionar o que é um sistema operacional. Esta discussão existe a muito tempo e possui diversas respostas divergentes, contudo, neste trabalho iremos considerar que o sistema operacional é o *kernel* mais um conjunto mínimo de funções para acesso ao *hardware* e uma interface mínima de gerência para o usuário. Sendo assim, cabe a ele a importante tarefa de gerenciar os recursos de um sistema computacional, evitando que aplicações os utilizem de maneira indevida, afetando o desempenho do sistema como um todo. a Figura 5 mostra de maneira simplificada a arquitetura de um sistema computacional moderno.

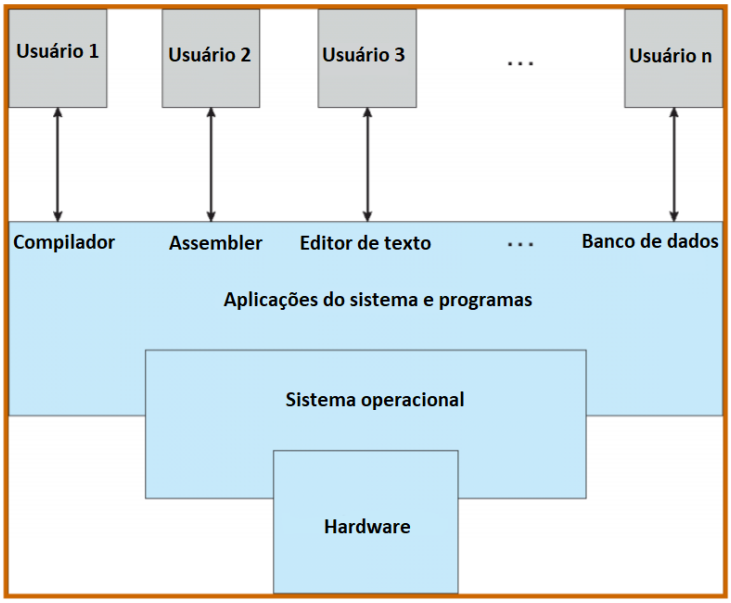

Figura 5 – Arquitetura simplificada de um sistema computacional. [Fonte: Silberschatz [2]]

Um sistema operacional de propósito geral, ou GPOS, como o seu nome revela, é designado a atender diversos tipos de aplicações de maneira eficiente, deste modo é muito

comum nos referirmos a este tipo de sistema como sistemas de alto desempenho. GPOS's estão presentes na grande maioria dos dispositivos de consumo atuais, sendo eles celulares, computadores, relógios inteligentes, centrais multimídia de veículos, roteadores de rede, etc.

A complexidade envolvida no desenvolvimento de um GPOS é alta, devendo ele oferecer uma ampla gama de recursos para os programas, ao mesmo tempo que precisa trabalhar de maneira rápida (nenhum usuário gosta de ver seu computador travando) e presente no maior número de arquiteturas possíveis [26]. Devido a estas exigências, GPOS's dificilmente possuem características de tempo real, sendo seu desenvolvimento focado em um melhor desempenho e não necessariamente em atender tarefas em instantes de tempo precisos.

Devido à alta complexidade de desenvolvimento, sistemas operacionais de propósito geral como o Linux utilizam arquitetura de *kernel* monolítico. O projeto GNU [27] tentou em meados da década de 80 desenvolver um *microkernel Unix-like* para o seu sistema operacional, todavia Richard Stallman, principal envolvido no projeto, e seus colegas notaram que o número de problemas encontrados era maior do que esperado e não conseguiram obter sucesso em sua ambiciosa empreitada [25]. A Figura 6 mostra como o núcleo do sistema Linux está organizado, esta organização é semelhante para os diversos sistemas operacionais que seguem o padrão *POSIX* [28], ou que são baseadas no antigo sistema Unix.

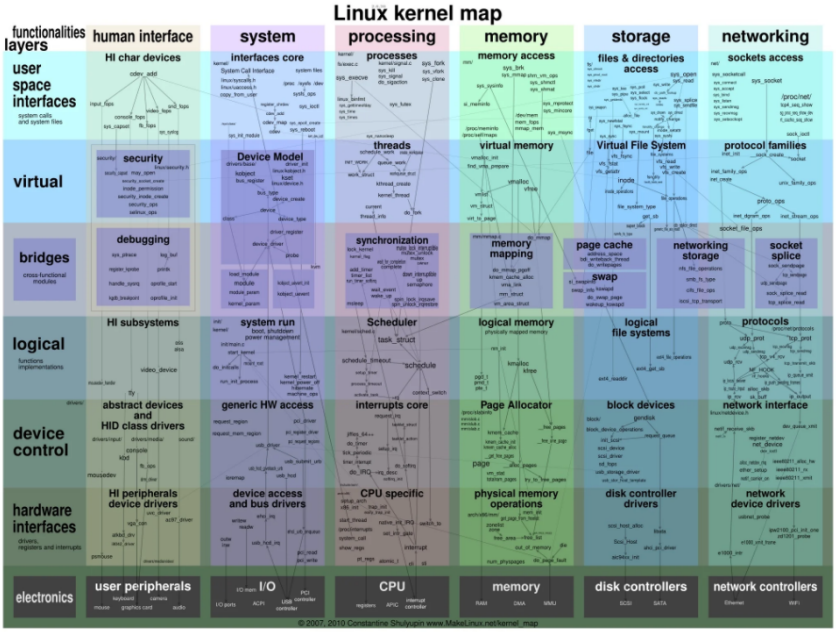

Figura 6 – Arquitetura de *kernel* monolítico. [Fonte: Fossbytes [23]]

Uma das principais tarefas em um sistema operacional de propósito geral é o escalonamento de processos. Mesmo em sistemas com múltiplos processadores, como os *chips* Intel multi-core modernos [29], não é possível ter vários processos rodando ao mesmo tempo.

Por isso, o sistema operacional realiza o escalonamento de processos, o que faz com que o usuário tenha a impressão de que existe um real paralelismo no sistema. Supondo que existam dois processos, sendo eles o processo A e B, o escalonamento ocorre, de maneira simplificada, da seguinte forma:

- Ambos os processos estão prontos para serem executados;
- O processo A ganha direito aos recursos de CPU e memória, passando a executar em primeiro plano;
- Caso o processo fique muito tempo utilizando os recursos de *hardware*, cabe ao sistema operacional tirar o seu acesso, liberando recursos para o processo B. Este processo é chamado de preempção;
- Também, se o processo A estiver rodando e tiver que esperar por alguma I/O ou algum outro processamento demorado ele entra em espera e libera recursos para o processo B;
- Quando ambos os processos são finalizados o sistema operacional fica a espera de novas requisições por parte de outros processos.

A Figura 7 demonstra as etapas presentes no ciclo de vida de um processo.

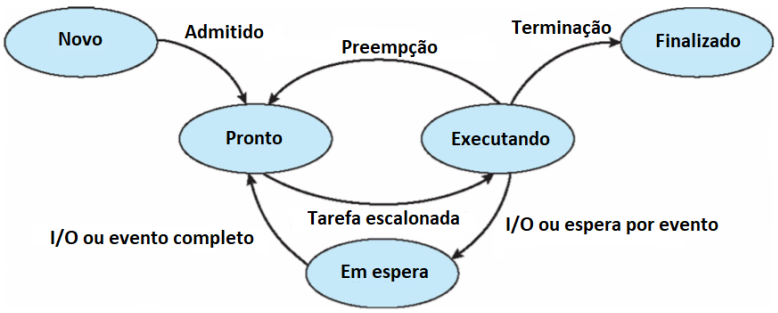

Figura 7 – Ciclo de vida de um processo. [Fonte: Silberschatz [2]]

Vemos na figura que um processo, após admitido, permanece alternando entre os estados pronto, executando e em espera conforme é escalonado pelo sistema operacional ou ocorram manipulações de entradas e saídas ou demais eventos. Quando o usuário fechar o processo, o sistema operacional se encarrega de finaliza-lo, liberando sua área de memória e tratando o retorno dado pelo processo durante esta etapa.

Para que o sistema operacional saiba em que parte de sua execução o processo parou ele guarda uma estrutura de dados denominada de bloco de controle de processo para cada processo na fila de execução [2]. Esta estrutura contém informações como o estado do processo, seu código, os valores dos registradores utilizados, o apontador de programa,

uma lista de arquivos abertos, etc. Deste modo, quando um processo sai de execução e libera recursos para o próximo ocorre a chamada troca de contexto, onde o sistema operacional salva o bloco de controle de processo do processo atual, lê o do próximo e libera uma parte de memória e CPU para que este seja executado, conforme a Figura 8.

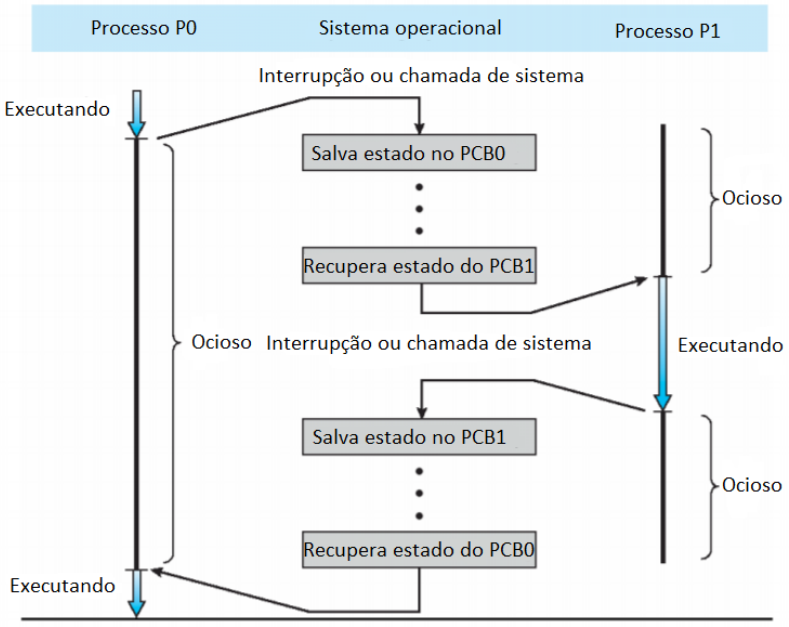

Figura 8 – Etapas envolvidas na troca de contexto. [Fonte: Silberschatz [2]]

2.4 Sistemas Operacionais de Tempo Real

Ao contrário de GPOS's, os RTOS's são sistemas designados para plataformas com poder de processamento muitas vezes reduzido onde o intuito não é rodar o maior número de programas ao mesmo tempo de maneira rápida, mas de atender às características temporais do sistema com uma alta previsibilidade. Sendo assim, um sistema de tempo real deve reagir a estímulos oriundos de seu ambiente em prazo específicos [30].

Grande parte dos sistemas atuais possuem requisições temporais. Por exemplo, um reprodutor de mídia precisa ler os dados provindos de uma unidade de armazenamento ou da rede e mostra-los ao usuário de acordo com os requisitos temporais da aplicação. Também temos dentro das indústrias diversos processos que precisam ser controlados de maneira rígida e cuidadosa, uma vez que a falha destes pode ocasionar graves acidentes [31].

Deste modo, as requisições de tempo real ainda podem ser divididas em *soft real time*, onde o não cumprimento dos requisitos temporais não irá ocasionar falhas catastróficas, apenas um incômodo para o usuário (como no caso dos reprodutores de mídia), e *hard real time*, onde o não cumprimento poderá causar falhas catastróficas, como no caso de sistemas de controle. Não obstante, é comum existirem sistemas mistos, ou seja, o mesmo

sistema deve atender requisições *soft real time* e *hard real time*, como no caso de sistemas de controle que mostram dados do sistema em uma tela e leem entradas do usuário via teclado, conforme mostrado na Figura 9.

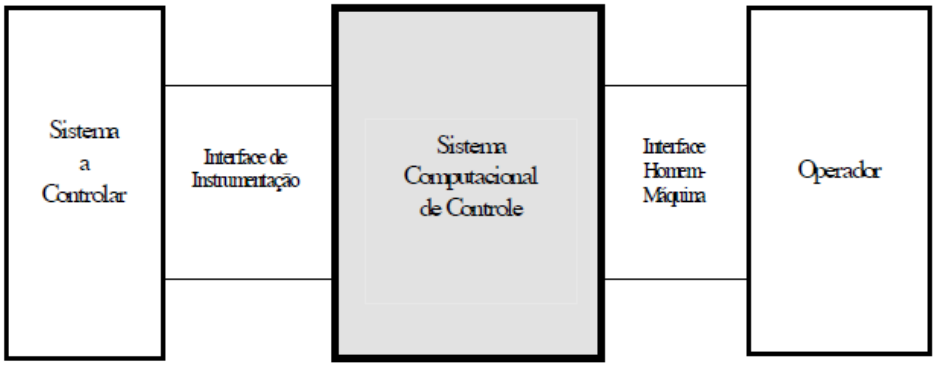

Figura 9 – Sistema de tempo real. [Fonte: Rômulo [30]]

Outro fator importante a ser analisado está no modelo de tarefas, podendo estas ser ativadas de maneira periódica, onde sabe-se de quanto em quanto tempo o sistema precisará atender à requisição como na Figura 10, ou aperiódicas, sendo ativadas por eventos externos e aleatórios como na Figura 11. O conhecimento desta características nos dá a base para a construção de um bom algoritmo de escalonamento, capaz de atender às requisições do sistema [30].

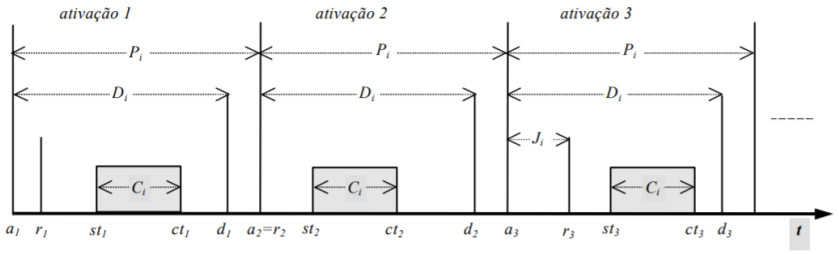

Figura 10 – Tarefas periódicas. [Fonte: Rômulo [30]]

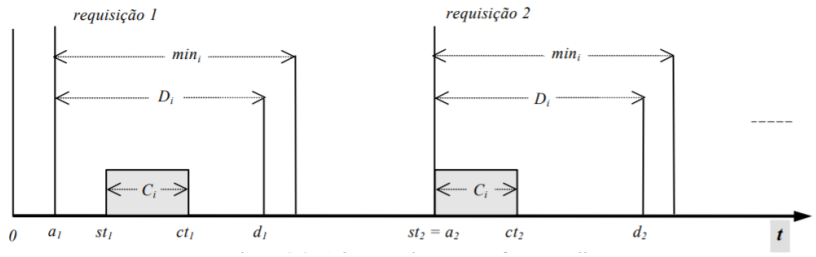

Figura 11 – Tarefas esporádicas. [Fonte: Rômulo [30]]

Por fim, o modelo de tarefas de um sistema de tempo real deve levar em conta outras variáveis como tempo de computação dos dados, tempo de resposta do sistema a eventos,

deadlines (instante máximo em que uma tarefa deve ter sua resposta), duração das tarefas, etc. Estes requisitos são a base para a escolha de linguagens de programação e algoritmos de escalonamento [30].

O escalonador de um RTOS é semelhante ao de um GPOS, porém aqui estamos falando de tarefas e não mais de processos, ou seja, em um RTOS usa-se o termo tarefa de maneira equivalente a processos em um GPOS. Neste sentido, um escalonador deverá ser o responsável por alocar recursos para diferentes tarefas de modo a atender todas as necessidades temporais de um sistema.

Existem vários algoritmos de escalonamento focados em resolver diferentes modelos de tarefa. Um escalonador pode adotar a simples estratégia *round robin*, onde uma lista de tarefas é percorrida do início ao fim repetitivamente, intercalando entre as mesmas, até estratégias mais complexas como o caso de algoritmos *rate monotinic* e *earliest deadline first*, que atribuem as prioridades com base nos períodos e *deadlines* de cada tarefa, respectivamente [2], estes algoritmos serão vistos com mais detalhes na seção de escalonamento. A Figura 12 demonstra como tarefas são escalonadas em um sistema de tempo real com o algoritmo *rate monotonic*.

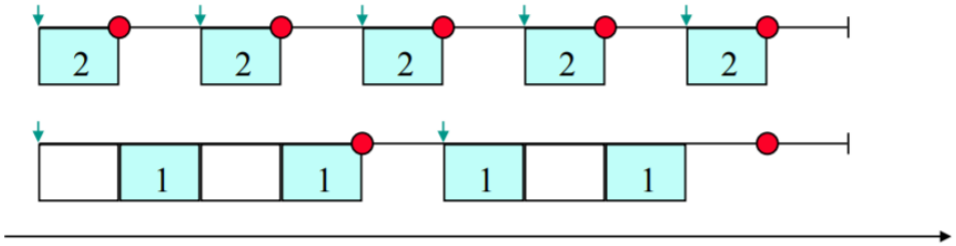

Figura 12 – Escalonamento com algoritmo RM. [Fonte: Rômulo [30]]

Como deseja-se obter uma alta previsibilidade nesta classe de sistemas [30], o desenvolvedor deve pensar em algoritmos que sejam ao máximo imune a falhas. Neste sentido, tempos aleatórios de escrita e leitura de memória secundária, imprevisibilidade de escalonamento, falhas gerais no sistema, dentre outros problemas comuns em GPOS's não são bem vindos em RTOS's. Por causa disso, os *kernels* dos principais sistemas operacionais de tempo real do mercado são baseados na arquitetura de *microkernel* [5].

A arquitetura de *microkernel* aplica-se neste cenário pelo fato de que um sistema operacional de tempo real possui um tamanho significativamente menor do que um sistema como o Windows ou Linux e suas funcionalidades também são reduzidas [5], de modo que o problema de comunicação entre as diferentes tarefas do sistema não se aplica aqui. Pelo contrário, a arquitetura em sistemas do tipo é o principal pilar para a sua previsibilidade [24], pois em um *microkernel* caso uma tarefa do sistema falhe, as demais tarefas não serão afetadas, por estarem em diferentes espaços de memória.

As diferenças entre o escalonamento em um RTOS para o visto em um GPOS recaem no fato de que RTOS's trabalham com tarefas que compartilham a mesma área de memória

e são mais simples do que processos que possuem sua própria área, gerando assim um bloco de controle menor. Na Figura 13 vemos a diferença entre uma *thread* (ou tarefa) e um processo.

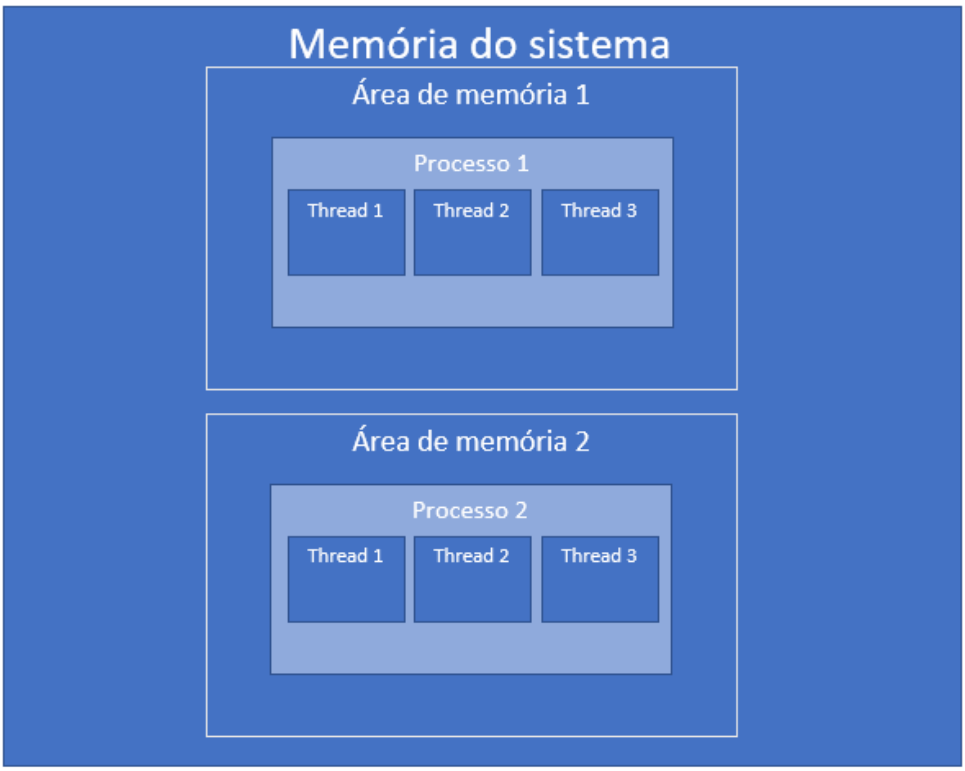

Figura 13 – Threads e processos alocados em memória. [Fonte: O autor]

2.5 Escalonamento

Conforme visto nas seções anteriores, o escalonamento é fundamental em um sistema operacional por prover um ambiente multi-tarefa ao usuário. Dentre as muitas vantagens presentes em um ambiente do tipo está a otimização de uso da CPU, visto que, tipicamente, processos e tarefas se dividem em ciclos de uso intenso de CPU seguidos por ciclos

de uso intenso de I/O [2]. A Figura 14 ilustra como ocorrem os ciclos de execução em um processo.

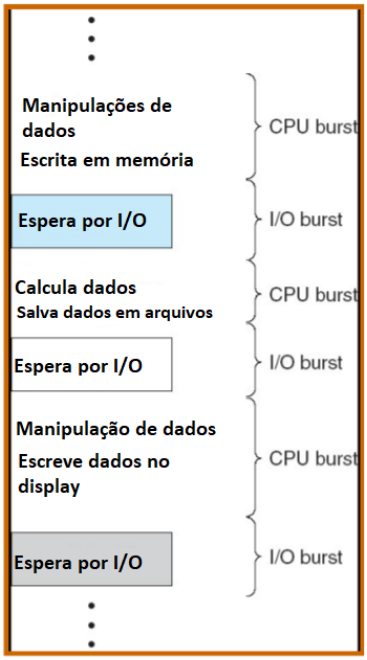

Figura 14 – Ciclos de execução em um processo. [Fonte: Silberschatz [2]]

Sendo assim, uma boa prática para otimizar o uso da CPU é liberar recursos de processamento para outros processos quando o processo atual entra em uma fase com pico de I/O. Em sistemas sem escalonamento, muito processamento é desperdiçado pelo algoritmo ficar parado em espera de operações de I/O.

O componente responsável pelo escalonamento de tarefas é o escalonador, cabendo a ele a tarefa de selecionar o próximo processo a ser executado na CPU e quando determinado processo deverá ser retirado do estado de execução, um escalonador pode ser preemptivo ou cooperativo [17], o escalonador é independente do *hardware*. O responsável pela troca de contexto é o despachante, ou *dispacher*, este sim é exclusivo do processador, dado que ele irá salvar os principais registradores e a pilha do sistema durante a troca de contexto.

No caso preemptivo, o escalonador possui a possibilidade de retirar um processo do estado de execução caso ele esteja ocupando os recursos de CPU por um longo período de tempo, substituindo-o por outro processo da fila de espera. Para o caso cooperativo, um processo apenas será retirado da execução quando terminar seu processamento ou entrar em estado de espera (saída voluntária). Este tipo de escalonamento é pouco utilizado em sistemas operacionais e possui sua aplicação limitada a processos curtos.

O desenvolvimento de um escalonador não segue uma receita fixa, existindo atualmente uma gama de algoritmos de escalonamento para cada tipo de aplicação [32]. A escolha por um algoritmo depende dos critérios de escalonamento do projeto, sendo alguns [2]:

1. **Utilização da CPU:** Deseja-se utilizar o máximo possível da CPU;

2. ***Throughput*:** Esta taxa indica o número de processos concluídos por unidade de tempo e é um indicativo do quanto de trabalho está sendo executado pela CPU. O foco, geralmente, é aumentar o *throughput*;

3. **Tempo de *turnaround*:** Este parâmetro indica o quanto de tempo um processo leva para ser executado;

4. **Tempo de espera:** O tempo de espera é a soma dos períodos gastos em espera na fila de processos prontos;

5. **Tempo de resposta:** Ao contrário do tempo de *turnaround*, o tempo de resposta leva em conta apenas o período de tempo entre uma solicitação e a primeira resposta a ser produzida, ignorando o tempo gasto no dispositivo de saída.

Não obstante, conforme visto na seção de Sistemas Operacionais de Tempo Real, em um RTOS existe uma série de características e requisições de tempo real a serem levadas em consideração [33]. Para estes sistemas, os escalonadores devem ser projetados de maneira a atender os *deadlines* impostos. Por isso, muitos algoritmos utilizados em GPOS's não são capazes de prover um bom escalonamento em sistemas operacionais de tempo real.

2.5.1 Escalonamento em GPOS's

Os principais algoritmos de escalonamento para GPOS's podem ser listados da seguinte forma:

1. ***First come, First Served*:** O escalonador seleciona a primeira tarefa disponível para ser executada assim que a CPU estiver livre. Esta técnica não é ideal, podendo produzir tempos de espera aleatórios, sendo que não existe nenhuma preferência por processos mais curtos serem executados antes de processos longos;

2. **Menor trabalho primeiro:** Ao contrário da técnica anterior, esta leva em conta o tempo de pico de CPU dos processos. Sendo assim, processos que usam a CPU por menos tempo irão ser executados antes e, em casos de picos de CPU iguais, utiliza-se o algoritmo *first come, first served*. Os tempos de espera resultantes são menores quando utilizado este tipo de algoritmo;

3. **Escalonamento por prioridades:** Atribui-se uma prioridade para cada processo, de modo que processos com maior prioridade são executados antes. Este tipo de algoritmo é muito útil em sistemas de tempo real, de modo a garantir que processos *hard real time* atendam às suas *deadlines*. A atribuição das prioridades pode ser fixa ou dinâmica, dependendo da aplicação;

4. ***Round-Robin*:** Outro algoritmo comum em RTOS's, este tipo de escalonador seleciona os processos de maneira similar ao *first come, first served*, contudo é adicionada a preempção ao escalonador, de modo que cada processo possui um *quantum*, ou fatia de tempo na qual deverá ser executado antes de voltar a fila de espera novamente.

Existem uma série de outros algoritmos com funcionalidades similares, porém que possuem prioridades dinâmicas, *quantum's* baseados em filas e até algoritmos que unem diversos tipos de escalonadores. Vale ressaltar aqui que os algoritmos em estudo são aplicados em sistemas *singlecore*, existindo um amplo número de técnicas e algoritmos de escalonamento específicos para sistemas multi processados que levam em conta determinadas características destes sistemas [32].

2.5.2 Escalonamento em RTOS's

No caso de sistemas de tempo real, o escalonamento possui características diferentes dos GPOS's. Devido a isso, existem algoritmos específicos adotados por RTOS's cujo intuito é assegurar o cumprimento de requisitos temporais do sistema. Abaixo vemos alguns dos principais algoritmos de escalonamento usados em RTOS's [30]:

1. ***Rate Monotonic*:** Este algoritmo atribui uma prioridade fixa para cada tarefa, definida pelo período em que esta tarefa irá ficar em execução. Neste cenário, tarefas com um período menor possuem uma prioridade maior, ou seja, se tivermos três tarefas com períodos iguais a 2 segundos, 3 segundos e 10 segundos, a tarefa de 2 segundos terá a maior prioridade, seguida pela de 3 segundos e, por fim, pela de 10 segundos.

2. ***Earliest Deadline First*:** Ao contrário do Rate Monotonic, este algoritmo possui prioridade dinâmica, atribuida conforme o *deadline* das tarefas. Como sugere o seu nome, ele atribui a maior prioridade a tarefa com o menor *deadline*, de modo a garantir que todos os *deadlines* sejam cumpridos.

3. ***Round Robin*:** Assim como nos GPOS's, este algoritmo é popular em RTOS's, sendo construído da mesma maneira que vimos anteriormente. Com ênfase, sistemas como o FreeRTOS [34] possuem seu escalonador baseado neste algoritmo.

Escalonar tarefas em sistemas de tempo real pode ser uma tarefa desafiadora, dada a necessidade de cumprimento dos requisitos temporais impostos, cumprindo com todos os *deadlines*. Diante disso, diferentes algoritmos de escalonamento podem ser adotados baseados em uma série de métricas, como no caso do algoritmo *Earliest Deadline First*. Todavia, atualmente os sistemas operacionais de tempo real, em sua grande maioria, adotam o algoritmo *round robin*.

2.6 Sincronização

A programação concorrente, apesar de todas as suas vantagens, trás consigo uma série de detalhes que devem ser levados em conta pelos desenvolvedores na hora da criação de um sistema, caso contrário o funcionamento do mesmo será prejudicado. Um dos maiores pontos a serem levados em consideração quando em ambientes com concorrência é o problema da região crítica, local do programa aonde há compartilhamento de recursos.

Para ilustrarmos este problema vamos supor uma aplicação com duas *threads*, onde cada uma incrementa o valor de uma variável em uma unidade. O funcionamento esperado para este algoritmo segue a sequência abaixo:

1. A *thread* 1 incrementa a variável em uma unidade;
2. Após a *thread* 1 ter feito seu trabalho, a segunda *thread* incrementa novamente o valor da variável;
3. Repetem-se os passos anteriores até que o sistema seja desligado ou reiniciado.

Todavia, processos de leitura e escrita na memória não ocorrem de maneira atômica (em uma só instrução). Muito pelo contrário, as arquiteturas modernas de microcontroladores dividem este processo em três etapas, sendo elas:

1. Carrega-se o valor em memória para um registrador. Em casos onde será aplicada alguma operação matemática no valor, utilizam-se registradores reservados para tal;
2. Aplica-se o processamento desejado no valor carregado para o registrado;
3. O resultado é carregado do registrador para o endereço de memória correspondente.

Percebe-se que, em um sistema preemptivo, existe uma grande probabilidade de que uma *thread* seja preemptada no meio do processo de alteração de valores em uma variável. Caso mais de uma *thread* esteja acessando os valores da mesma variável isso causa uma condição de corrida ou competição por recursos e, sem um mecanismo de sincronização, ambas estarão acessando valores errados [2].

Para contornar este problema existem mecanismos capazes de fazer com que apenas uma *thread* acesse sua região crítica (exclusão mútua), deixando as outras em espera, até que seja liberado o acesso pela *thread* atual. De maneira resumida, os principais mecanismos existentes são os mutexes e semáforos, sendo que a sua implementação dependerá das características da arquitetura utilizada no projeto.

Em processadores *singlecore* pode-se adotar a estratégia de desabilitar todas as interrupções do sistema, evitando que o sistema desvie o fluxo do programa para uma tarefa ou processo que utilize a variável compartilhada, conforme a Figura 15. De fato, esta é

uma estratégia muito utilizada, porém a sua simplicidade de implementação trás a inconveniente desvantagem de que o processo de escalonamento não será mais tão previsível e, se não implementado com muito cuidado, o sistema de exclusão mútua irá impedir que os *deadlines* em sistemas de tempo real sejam atendidos. Além do mais, esta estratégia não funciona em sistemas com múltiplos processadores por um série de motivos que afetam o desempenho dos mesmos e, em muitos casos, não funcionam devidamente, porém não daremos foco a esta classe de sistemas no trabalho.

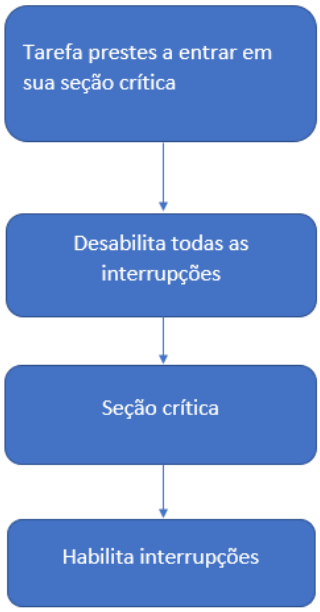

Figura 15 – Sincronização desabilitando interrupções. [Fonte: O autor]

Vários microprocessadores modernos possuem suporte de *hardware* para sincronização [35]. Este suporte é dado através de instruções do tipo *test_and_set()* e *compare_and_swap()* [36]. Em essência, os microprocessadores garantem a sincronização pelo fato de que tais instruções são atômicas, de modo que não será possível uma *thread* ler determinada variável enquanto outra estiver alterando o seu valor. Contudo, estas instruções não ficam disponíveis para o usuário e devem ser utilizadas de maneira eficiente durante o desenvolvimento do *kernel* de um sistema operacional.

Graças às limitações das técnicas acima descritas, a sincronização de tarefas dá-se comumente através de mutexes e semáforos. Mutexes podem ser vistos como chaves, sempre que uma tarefa quiser entrar em sua seção crítica ela deverá requisitar a chave, caso outra tarefa tenha requisitado e obtido esta chave anteriormente, a tarefa que requisitou depois não poderá entrar em sua região crítica até que a chave seja devolvida pela tarefa que a possui. Este comportamento está explicitado na Figura 16.

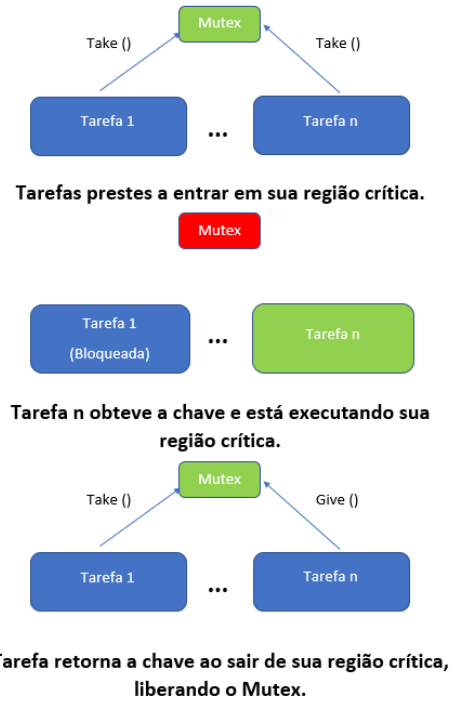

Figura 16 – Sincronização através de mutexes. [Fonte: O autor]

Semáforos, por outro lado, podem ser vistos como contadores. Neste tipo de mecanismo as tarefas podem adicionar recursos no semáforo (aumentar o valor do contator) ou retirar recursos dele (decrementar o valor do contator). Semáforos são especialmente úteis quando queremos, por exemplo, liberar um determinado número de tarefas a entrar em suas regiões críticas ou a usar determinados recursos. A Figura 17 ilustra como semáforos funcionam.

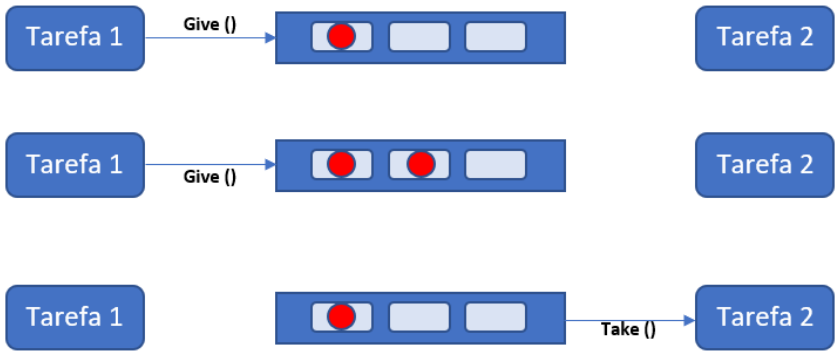

Figura 17 – Uso de semáforos. [Fonte: O autor]

Mesmo sendo ferramentas poderosas, os mecanismos de sincronização entre tarefas devem ser utilizados com muita cautela, pois podem gerar condições como *deadlocks*, onde o sistema fica bloqueado indefinitivamente devido a má manipulação de mutexes, por exemplo. Uma situação em que ocorre *deadlock* pode ser vista abaixo [2].

- Temos duas tarefas no sistema, sendo chamadas de tarefa 1 e tarefa 2. A tarefa 1 precisa ter acesso ao mutex 1 para entrar em sua região crítica e a tarefa 2 precisa de acesso ao mutex 2;
- Ao entrar em sua região crítica a tarefa 1 bloqueia o mutex 2, sem posteriormente desbloqueá-lo;
- Para que o mutex 1 seja desbloqueado, a tarefa 2 precisa entrar em sua região crítica.

No exemplo acima temos que, caso a primeira tarefa entre em sua região crítica, todo o sistema irá ficar bloqueado, pois o seu mutex apenas pode ser desbloqueado pela tarefa 2 e esta não consegue entrar em sua região crítica, sendo que o seu mutex foi bloqueado pela tarefa 1.

2.7 A família de processadores 8051

A história da família de processadores Intel 8051 é, no mínimo, curiosa. Isso porque, atualmente é difícil conseguirmos imaginar que uma arquitetura de microprocessadores de 8 bits criada em meados dos anos 1980 tenha algum espaço no altamente competitivo mundo da tecnologia. Mesmo com a popularização de arquiteturas como a ARM [37], com processadores de até 64 bits, a família Intel 8051 continua presente em uma grande gama de dispositivos no mercado e, não obstante, novos chips baseados nela continuam a surgir no mercado.

No momento em que este trabalho é escrito, dispositivos como teclados, aspiradores de pó, ar condicionados e vários outros são controlados por uma série de microprocessadores baseados na arquitetura Intel 8051 [38]. Juntamente a isso, muitos fabricantes de semicondutores estenderam a arquitetura, aumentando a sua velocidade de processamento, o tamanho de memória interna, adicionando memória EEPROM dentro do chip, dentre outras novas funcionalidades. Devido a isto, a família Intel 8051 recebeu uma nova variante, a Intel 8052, com mais memória disponível, adição de mais *timers* e outros recursos de *hardware* [17].

Para termos noção do tamanho da família Intel 8051/52, em 2008 um levantamento da empresa KEIL [38] listava os seguintes fabricantes de chips contendo processadores da família 8051/52:

1. Analog Devices;
2. Atmel;
3. Cypress Semiconductor;
4. California Eastern Laboratories;

5. SiLABS;

6. Teridian Semiconductor.

Se entrarmos no site de dispositivos suportados pelo compilador C51, um dos mais famosos para a arquitetura, vamos nos deparar com uma lista extensa de dispositivos suportados [39]. Se o número de dispositivos baseados em uma arquitetura de 8 bits impressiona, vale ressaltar que a página conta com uma série de novos chips suportados continuamente. Vemos dispositivos da família sendo aplicados em FPGA's [1] até processadores voltados para leituras de sensores com conversores AD de 24 bits [7].

Alguns dos fatores que são favoráveis a adoção de processadores baseados na arquitetura são: Processadores compactos, componentes com bom custo-benefício, poucos ciclos por instrução (de 12 ciclos até apenas 1 ciclo), microcontroladores com memória secundária e outros recursos como conversores AD inclusos, altas velocidades de processamento (a família ADuC84x possui processadores com 12,58 MIPS, por exemplo), etc.

2.7.1 Especificações técnicas

Em conformidade com os dados vistos acima, fica difícil elencarmos todas as especificações técnicas da família 8051/52, em virtude de cada fabricante adicionar funcionalidades específicas para as aplicações atendidas por seus processadores. Em razão disso, vamos nos ater às características padrões de um processador 8052, que estarão presentes em qualquer chip baseado na plataforma.

Originalmente, o Intel 8051 possui as seguintes especificações [17]:

- Frequência de operação de até 12MHz;
- Memória RAM interna de 128 bytes;
- 32 portas digitais de entrada e saída;
- Dois *timers*/contadores de 16 bits;
- Versões com e sem memória para programa;
- 5 fontes de interrupção com dois níveis de prioridade;
- Uma porta serial *full-duplex*.

Posteriormente, o 8052 foi lançado pela Intel com algumas melhorias. Este microprocessador era totalmente compatível com os chips 8051 em termos de pinos e código. No dias de hoje, quando falamos de processadores baseados na família 8051, existe uma grande probabilidade de estarmos lidando com um dispositivo baseado no Intel 8052. As características técnicas deste processador são as seguintes:

- Memória RAM interna aumentada para 256 bytes;
- 3 *timers*/contadores;
- 6 fontes de interrupção disponíveis, com dois níveis de prioridade.

Além disso, processadores atuais baseados na arquitetura incrementam estas funcionalidades. Por exemplo, processadores como o AT89S53 da Atmel [40] possuem nove fontes de interrupção e timer de *whatchdog*, já o processador Dallas 89C420 [41] consegue operar em até 50MHZ, trazendo uma performance entre 40 e 50 MIPS.

A organização da memória dentro de um microprocessador da família segue a Figura 18, sendo dividida em:

- CODE: Espaço em memória *flash* para armazenar o programa a ser executado pelo processador;
- DATA: Memória RAM interna do processador, este espaço é dividido com a pilha e quatro bancos de registradores;
- IDATA: 128 bytes de memória RAM interna, expandindo o espaço presente em DATA;
- SFR's: Espaço de memória dedicado aos registradores do processador;
- PDATA: Memória RAM externa, acessada inderetamente;
- XDATA: Memória RAM externa de 64 KBytes. Este espaço de memória, quando disponível, sobrepõe o espaço PDATA.

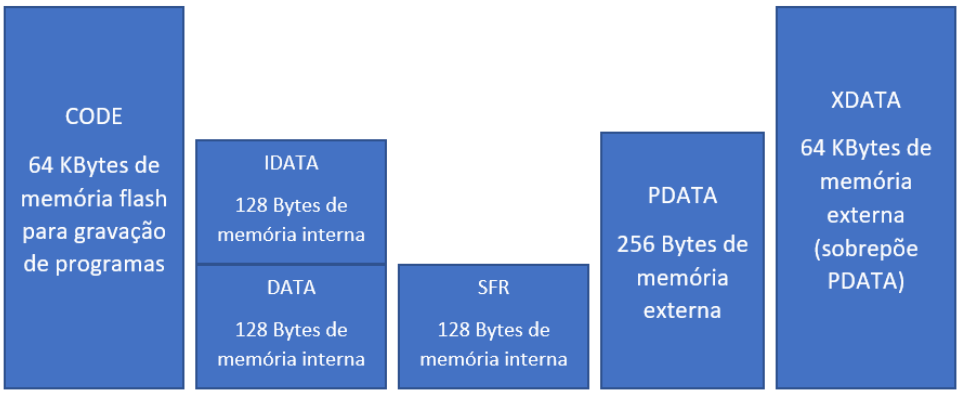

Figura 18 – Arquitetura da memória de um processador 8051/52. [Fonte: Pont [17]]

É comum encontrarmos no mercado microcontroladores modernos que estendem a memória RAM disponível adicionando uma memória externa nativa (espaço denominado XDATA). Compiladores modernos como o C51 [42] e o SDCC [8] possuem a opção de

alocar o maior número possível de variáveis nesta região de memória. Este tipo de organização de memória é muitas vezes denominado de modelo estendido ou, no inglês, *model large*.

A desvantagem de usar a memória externa de um processador 8051 está no fato de que a leitura e escrita nestes endereços é mais lenta, por não ser acessada diretamente. Mesmo em microcontroladores com memória externa nativa, o acesso é feito através do registrador DPTR, que atua como ponteiro para o microprocessador, da seguinte forma:

1. Carregar o endereço de memória no registrador DPTR;
2. Mover o valor presente no endereço de memória apontado pelo DPTR para um registrador;
3. Manipular os dados no registrador;
4. Mover o resultado do registrador para o endereço apontado pro DPTR.

Na arquitetura, os SFR's (*Special Function Registers*) são os responsáveis pela configuração do processador. Existem diversos SFR's dentro de um dispositivo da família 8051, sendo alguns genéricos (presentes em todos os processadores da família) e alguns especiais. Estes registradores são alocados em endereços de memória específicos, normalmente ficando na parte alta da memória interna do processador. Todos os recursos do processador são configurados através de seus SFR's, por exemplo, a leitura e escrita de valores em portas digitais é feita através do SFR correspondente a porta, a configuração da UART é feita através de alguns SFR's destinados a isso, configuração do *clock* da CPU, opções de economia de energia, dentre outras funções somente podem ser acessadas via um SFR.

Na memória interna este acesso é feito diretamente, carregando o valor do endereço sem necessidade de utilizar o registrador DPTR. O microcontrolador ADuC847 da Analog Devices é um exemplo de chip que disponibiliza uma memória externa nativa, a organização da memória externa dele pode ser vista na Figura 19.

De acordo com a Figura 18, vemos que a memória interna de 256 bytes é dividida em duas partes de 128 bytes. Comumente, a parte alta fica reservada para os registradores comuns da arquitetura 8051 e específicos de cada processador [35], os 128 bytes da parte baixa se dividem da seguinte maneira:

- Bytes 00 a 1F: 32 bytes contendo 4 bancos de registradores R0 a R7 de propósito geral;
- Bytes 20 a 2F: 16 bytes disponíveis para endereçamento de 128 variáveis binárias, conforme a Figura 20;
- Bytes 30 a 7F: 80 bytes para propósito geral.

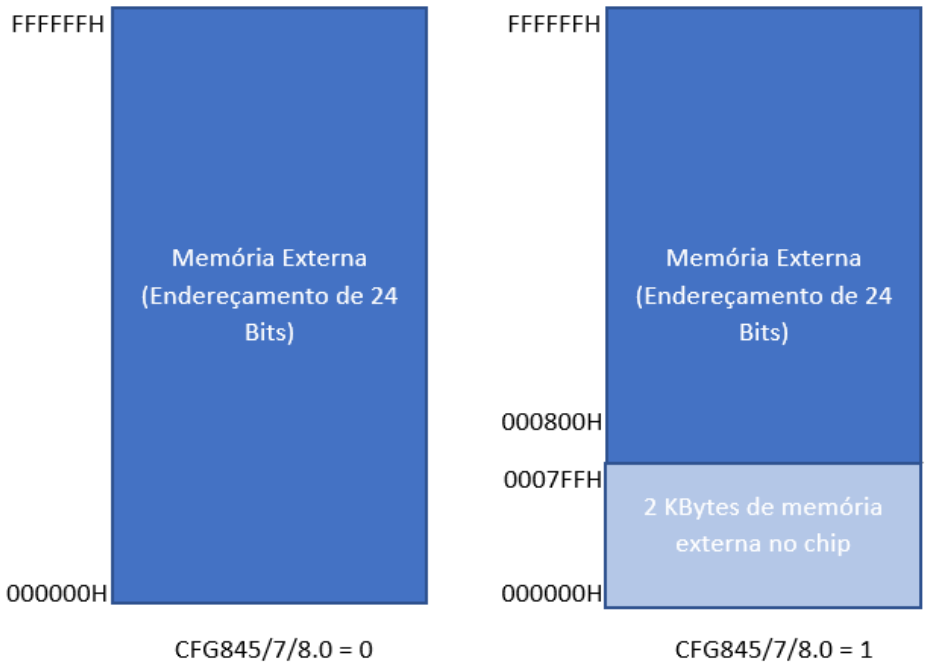

Figura 19 – Memória externa da família ADuC84x. [Fonte: Datasheet ADuC84x [7]]

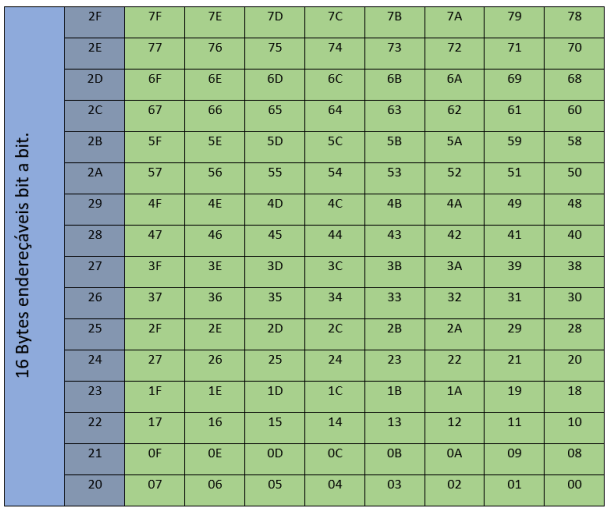

Figura 20 – Variáveis disponíveis para o endereçamento bit a bit. [Fonte: O autor]

Em especial, os 4 bancos de registradores são muito úteis para promover maior velocidade, podendo-se alocar bancos diferentes para cada fonte de interrupção do programa, evitando manipulações da pilha no momento da entrada na rotina de tratamento de interrupção. Os 16 bytes disponíveis para endereçamento de variáveis do tipo binárias podem ser usados deste jeito ou como endereços normais, mais a frente trataremos da vantagem de utilizar estes endereços como variáveis binárias. Por fim, os 80 bytes de propósito geral devem ser utilizados com muita cautela, dado que eles irão dividir espaço de memória com a pilha do processador.

Levando em conta a disponibilidade de 64k bytes de memória externa, uma boa prática

de programação para a arquitetura é guardar apenas variáveis importantes e que devem ser manipuladas de maneira mais rápida na memória interna, utilizando a memória externa para variáveis secundárias.

Por fim, os principais registradores de um processador 8051 e suas funções são as seguintes [7]:

- Acumulador (ACC) e B: Registradores utilizados para operações matemáticas de soma, subtração, divisão e multiplicação. O acumulador pode ser utilizado para manipulações Booleanas em bits;
- DPTR: Registrador utilizado para manipulação de dados em endereços de memória. Este registrador é dividido em DPP, DPL e DPH;
- *Stack Pointer (SP)*: Registrador que contém o endereço em memória interna do topo da pilha do sistema;
- *Program Status Word (PSW)*: Contém dados sobre o *status* da CPU como o seletor do banco de registradores, *flag* de *carry* e *flag* de *overflow*.

3 Estado da Arte

Discutimos nos Capítulos 1 e 2 que o campo de estudos sobre sistemas operacionais não é novo e possui uma extensa bibliografia, contando com algoritmos consolidados e amplamente utilizados tanto em GPOS's quanto em RTOS's. Também foi citado um exemplo de RTOS popular, o FreeRTOS [5].

Neste capítulo faremos uma breve análise acerca de como as funcionalidades de escalonamento e sincronismo, principais focos do trabalho, são implementadas no FreeRTOS, bem como uma discussão sobre outros recursos presentes no sistema e o porquê destes não serem pertinentes para o trabalho em questão. Posteriormente, listaremos uma série de RTOS's com algum nível de suporte para a arquitetura, discutindo até onde eles realmente suportam a mesma, seu principal foco, suas características como o uso de memória, algoritmos de escalonamento e sincronismo.

3.1 FreeRTOS

Existe uma grande chance de que, se o leitor tiver afinidade com a área de sistemas computacionais, esse nome já tenha aparecido em algum momento de sua vida. O FreeRTOS é, provavelmente, o sistema operacional de tempo real mais conhecido do mercado na atualidade, contando com um *kernel* monolítico poderoso *opensource* que está por trás de um imenso número de sistemas embarcados. Dispositivos como a pulseira inteligente Galaxy Fit2 [43] da Samsung são baseadas neste sistema operacional.

Ainda, por ser um sistema com código fonte aberto, o FreeRTOS suporta um grande número de processadores. É comum que desenvolvedores precisando de um RTOS para seu sistema optem por portarem este sistema operacional para a arquitetura ao invés de construírem seu próprio RTOS. Desta forma, ele suporta os principais processadores do mercado de maneira oficial [44], bem como possui *ports* para outras arquiteturas feitas por usuários de maneira independente [45].

Em termos de ferramentas, o sistema possui uma extensa biblioteca, suportando funcionalidades como proteção de memória, modos de execução de usuário e *kernel*, escalonamento preemptivo, funções diversas para uso de *timers*, semáforos e mutexes, filas, área de memória compartilhada, dentre outras [34].

Um parâmetro importante a ser analisado em um sistema operacional são os *footprints* de memória e de código. O *footprint* de memória trata-se da quantidade da memória SRAM utilizada pelo sistema e o *footprint* de código trata da memória *flash* utilizada pelo mesmo. Em suma, *footprint* de memória se refere ao quanto o sistema usa para as suas variáveis e o *footprint* de código se refere ao quanto o código do sistema usa na

memória não volátil dedicada a armazenar os códigos pelo processador. Os *footprints* de memória e código no FreeRTOS são, respectivamente, entre 5 e 10 Kbytes e 250 bytes [46], o que é impressionante se levarmos em conta todas as funcionalidades do sistema.

3.1.1 Gerenciamento de tarefas

De acordo com o Capítulo 2, uma tarefa em um RTOS é o equivalente a um processo em um GPOS. Deste modo, uma tarefa pode ser vista como um pequeno programa que irá ser executado até que todo o sistema pare. Também existe a possibilidade de uma tarefa rodar por um número finito de vezes e, após isso, ser excluída.

No FreeRTOS tarefas são procedimentos sem retorno de valor e que têm como parâmetro um ponteiro do tipo *void* [47]. No caso de sistemas *single core*, uma tarefa irá receber recursos de processamento, ficando por um determinado período de tempo em execução e, após isso, irá entrar em um estado de espera. Esta é apenas uma simplificação para começarmos a entender o ciclo de vida de uma tarefa no FreeRTOS, deste modo, a Figura 21 ilustra este primeiro modelo de ciclo de vida das tarefas no sistema.

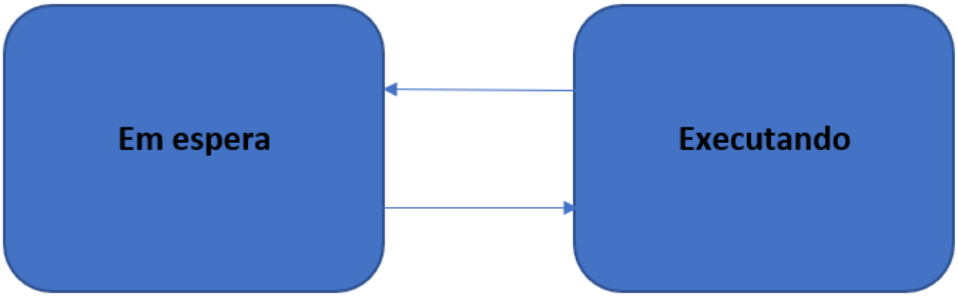

Figura 21 – Ciclo de vida simplificado no FreeRTOS. [Fonte: Melot [47]]

Em termos de escalonamento, o sistema adota um algoritmo *round robin* com prioridade. Neste tipo de algoritmo cada tarefa é associada a uma prioridade, definida pelo usuário, sendo 0 a menor e a maior prioridade definida pelo usuário na constante *MAX_PRIORITIES* encontrada no arquivo FreeRTOS.h. Posteriormente, é possível que o usuário altere a prioridade de uma tarefa. Contudo este tipo de escalonamento muito comumente gera o que chamamos de *starvation*.

Starvation é um problema comum em sistemas multi tarefas e ocorre quando, em linhas gerais, determinadas tarefas nunca conseguem obter recursos de processamento, ou seja, nunca são escalonadas [2]. No caso do FreeRTOS, se uma tarefa tiver prioridade 1 e outra prioridade 0, a tarefa de menor prioridade somente será executada quando a de maior prioridade for excluída ou estiver em um estado em que não pode ser escalonada, conforme veremos mais a frente. Portanto, é uma boa prática adotar a mesma prioridade para todas as tarefas, ao menos em um primeiro momento do projeto.

A preempção é feita através de uma técnica conhecida como *time slicing* [48], nesta técnica o escalonador é chamado de maneira periódica, dando vez para uma das tarefas que estiver na fila de espera. O FreeRTOS possibilita que o usuário defina a frequência com a qual o escalonador será evocado, deixando livre para o usuário escolher frequências que se encaixem melhor em suas aplicações. A Figura 22 nos mostra como ocorre a preempção no FreeRTOS.

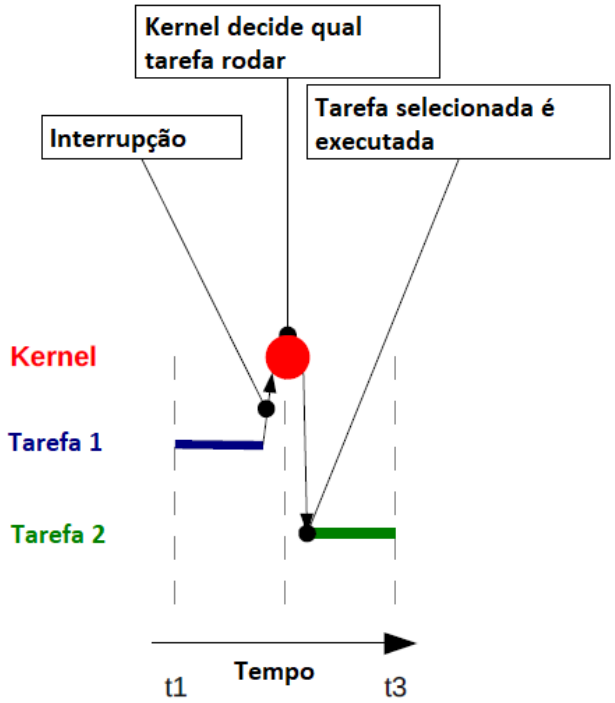

Figura 22 – Processo de escalonamento do FreeRTOS. [Fonte: Melot [47]]

Por fim, podemos expandir o diagrama visto na Figura 21, cobrindo todos os estados em que uma tarefa pode estar quando não estiver sendo executada. Conforme mostra a Figura 23, estes estados são:

- Pronta: Neste estado a tarefa é alocada em uma fila de espera, junto com as demais tarefas a serem executadas, para que eventualmente o escalonador a selecione, liberando a CPU para a mesma;
- Suspensa: Uma tarefa somente será suspensa quando chamar o método vTaskSuspend(). Este estado é interessante para aplicações que possuam tarefas com diferentes prioridades, de modo que uma tarefa de maior prioridade pode se suspender, liberando as demais de menor prioridade para serem executadas. Ao chamar o método vTaskResume() ou xTaskResumeFromISR(), a tarefa volta ao estado de pronta;
- Bloqueada: Este estado é atingido de duas maneiras, sendo elas um bloqueio devido a um evento temporal ou devido a sincronização. No caso de eventos temporais,

ao chamar um método de delay, a tarefa fica bloqueada até que o período tenha passado. No caso de sincronização uma tarefa pode ficar parada até que um mutex, por exemplo, seja liberado para a mesma.

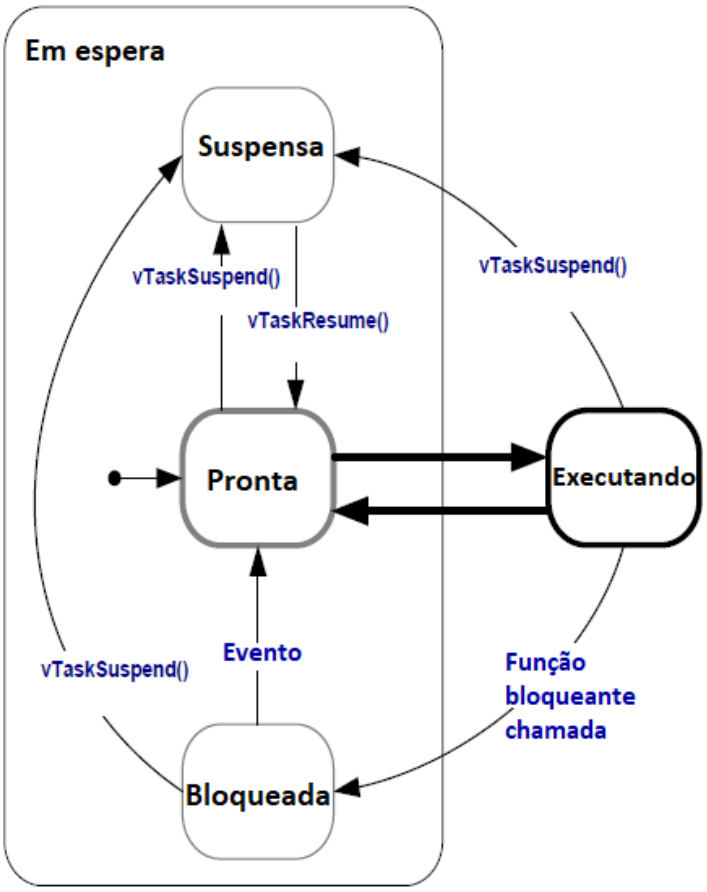

Figura 23 – Ciclo de vida completo de uma tarefa no FreeRTOS. [Fonte: Barry [48]]

3.1.2 Mecanismos de sincronização

O uso de mutexes e semáforos ocorre via chamadas de funções da API de semáforos do sistema. Tanto mutexes quanto semáforos são manipulados por variáveis do tipo *SemaphoreHandle_t*. O sistema oferece a possibilidade de criar semáforos binários, cuja funcionalidade é semelhante a dos mutexes.

A Tabela 1 contém as funções utilizadas para manipulação de mutexes, seguindo de uma breve descrição de sua funcionalidade.

Tabela 1 – Funções para manipulações de mutexes

Nome	**Descrição**
xSemaphoreCreateMutex().	Cria um novo mutex, retornando NULL caso não exista espaço na *heap* para o mesmo.
xSemaphoreCreateMutexStatic().	Cria um mutex estaticamente, no endereço definido pelo usuário. Útil em sistemas com memória limitada.
xSemaphoreCreateRecursiveMutex().	Cria um novo mutex a ser usado em situações aonde possa existir problemas de recursividade.
xSemaphoreCreateRecursiveMutexStatic().	Cria um mutex estaticamente, no endereço definido, para ser usado em situações com recursividade.
xSemaphoreGetMutexHolder().	Retorna a função que está, atualmente, com a posse do mutex.
uxSemaphoreGetCount().	Retorna zero caso o mutex esteja livre ou um caso ele tenha sido obtido.
xSemaphoreGive(), xSemaphoreGiveFromISR(), xSemaphoreGiveRecursive().	Libera o mutex.
xSemaphoreTake(), xSemaphoreTakeFromISR(), xSemaphoreTakeRecursive().	Tenta obter o mutex, caso este esteja bloqueado, a tarefa entra em estado de bloqueio.
vSemaphoreDelete()	Exclui o mutex.

Fonte – O autor

Nas tabelas 2 e 3 podemos encontrar as principais funções para manipulação de semáforos binários e semáforos não binários, respectivamente. A principal diferença entre semáforos binários e mutexes residem no fato de que o sistema irá aumentar a prioridade da tarefa que obteve o recurso durante o uso de um mutex, sendo esta obrigada a desbloquear o mutex para retornar a sua prioridade normal, enquanto semáforos binários não alteram a prioridade da tarefa, permitindo que outras tarefas desbloqueiem o recurso sem necessariamente terem o obtido antes. Em linhas gerais, semáforos binários são recomendados para criação de sincronismo entre as tarefas, ao passo que mutexes são recomendados para questões de exclusão mútua [34].

Tabela 2 – Funções para manipulações de semáforos binários

Nome	**Descrição**
xSemaphoreCreateBinary()	Cria um semáforo binário, retorna NULL caso não existe espaço na *heap* para o mesmo.
xSemaphoreCreateBinaryStatic()	Cria um semáforo binário estaticamente, no endereço definido pelo usuário. Útil em sistemas com memória limitada.
uxSemaphoreGetCount()	Retorna zero caso o semáforo binário esteja bloqueado e um caso esteja livre.
xSemaphoreGive(), xSemaphoreGiveFromISR(), xSemaphoreGiveRecursive()	Faz com que o valor do semáforo volte a ser um.
xSemaphoreTake(), xSemaphoreTakeFromISR(), xSemaphoreTakeRecursive()	Faz com que o valor no semáforo binário seja igual a zero.
vSemaphoreDelete()	Exclui o semáforo binário.

Fonte – O autor

Tabela 3 – Funções para manipulações de semáforos comuns

Nome	**Descrição**
xSemaphoreCreateCounting()	Cria um semáforo, retorna NULL caso não existe espaço na *heap* para o mesmo.
xSemaphoreCreateCountingStatic()	Cria um semáforo estaticamente, no endereço definido pelo usuário. Útil em sistemas com memória limitada.
uxSemaphoreGetCount()	Retorna o valor da variável de contagem no semáforo. Este valor varia entre zero e o valor máximo estipulado na criação do mesmo.
xSemaphoreGive(), xSemaphoreGiveFromISR(), xSemaphoreGiveRecursive()	Incrementa uma unidade na variável de contagem do semáforo
xSemaphoreTake(), xSemaphoreTakeFromISR(), xSemaphoreTakeRecursive()	Decrementa em uma unidade a variável de contagem do semáforo.
vSemaphoreDelete()	Deleta o semáforo.

Fonte – O autor

Quanto a implementação dos mecanismos, esta pode variar de acordo com a arquitetura do processador para o qual o FreeRTOS foi compilado, podendo o sistema utilizar instruções *test_and_set* ou simplesmente desabilitar interrupções enquanto estiver alterando os dados dos mutexes ou semáforos.

3.1.3 Recursos adicionais do sistema

A API do FreeRTOS, conforme dito anteriormente, é extremamente extensa. Para termos noção, apenas o manual oficial da API possui 400 páginas, isso sem levarmos em conta o fato de que ela esta em constante desenvolvimento e vários desenvolvedores colaboram com o projeto. Apenas a API de escalonamento e gerenciamento de tarefas conta com 35 funções no momento em que este trabalho é escrito.

Além dos recursos de escalonamento e sincronismo citados nas seções anteriores, o sistema também possui funcionalidades como listas, funções para uso de *timers*, alocação dinâmica de memória, grupos de eventos, troca de mensagens entre tarefas, dentre outros. Estes recursos são muito utilizados em sistemas de tempo real, todavia sua implementação em sistemas com poucos recursos pode ser problemática ou inviável.

O maior fator limitante em arquiteturas como a da família 8051 na implementação de recursos como filas, alocação dinâmica de memória e área de memória compartilhada para troca de mensagens está no espaço disponível em SRAM. Vimos no Capítulo 2 que a memória interna do dispositivo possui apenas 256 byte, a serem compartilhados com SFR's, pilha, bancos de registradores R0 a R7 e banco de variáveis binárias. Este espaço deve ser utilizado apenas para um número pequeno de variáveis que precisam ser acessadas de maneira rápida e não é o suficiente para implementação dos recursos em estudo.

Mesmo utilizando a área de memória externa, teremos apenas 64 kbytes disponíveis, a serem compartilhados com outras variáveis do sistema operacional. Alguns RTOS's disponíveis no mercado oferecem um grande número de funcionalidades, porém limitam o número de tarefas que o usuário pode criar. O mesmo problema referente ao número de recursos disponíveis está presente na implementação de funções para manipulação de *timers* e outros recursos de *hardware*.

Neste cenário, em um RTOS para arquiteturas com recursos limitados faz mais sentido primar pela implementação de algoritmos de escalonamento e sincronismo, deixando a cargo do usuário a implementação de funções extras, gerando assim um sistema com pequenos *footprints* de código e memória.

3.2 RTOS's para a família 8051

Existe uma série de sistemas operacionais de tempo real construídos para a família 8051, alguns deles contendo funcionalidades semelhantes as encontradas no FreeRTOS. De

maneira geral, a grande maioria destes sistemas é feita em cima de compiladores pagos, pelas mesmas empresas que os vendem.

Existem sistemas que são feitos de maneira independente por uma série de desenvolvedores que encontraram a necessidade de utilizar um RTOS em seus projetos, mas que não se contentaram ou não dispunham de recursos financeiros para trabalhar com os sistemas mais populares pagos do mercado. Junto a esse grupo, existem projetos que portaram sistemas conhecidos, como o FreeRTOS, para a arquitetura.

Nesta seção iremos elencar alguns dos sistemas operacionais de tempo real que possuam suporte para a arquitetura Intel 8051 e quais os seus focos de trabalho e suas características técnicas. Ao final, faremos um compilado de todos os dados obtidos, de modo a obter-se um melhor entendimento acerca do cenário em que este trabalho se encontra.

3.2.1 KR-51

Começaremos nossa lista com o RTOS da Reisonance, o KR-51. Não mais suportado, este sistema era entregue em duas versões, sendo elas a normal e a *tiny*, estas versões eram adquiridas juntamente com o ambiente de desenvolvimento da Reisonance, contendo uma IDE completa com compilador C para a arquitetura, simulador, macro assembler, dentre outras ferramentas.

Apesar de, atualmente, o único resquício deste sistema estar em um site de vendas de produtos relacionados a microcontroladores [49], podemos ter uma visão do que este RTOS oferecia. As funcionalidades que chamam a atenção dentro do sistema são:

- Escalonador com suporte para até 125 tarefas;
- Gerenciamento de memória e tempo;
- Semáforos;
- Eventos de sincronização;
- Comunicação entre tarefas.

Infelizmente, dados acerca de uso de memória pelo sistema não estão presentes de maneira clara, sendo assim nos limitaremos a uma breve analise das funcionalidades acima. O que chama a atenção no sistema é o número de tarefas permitidas, sendo um indicativo da otimização do mesmo, tendo que os PCB's implementados precisarão ter um tamanho de apenas 16 Bytes, levando em conta os processadores com memória de 2 Kbytes vistos no Capítulo anterior.

Por fim, o suporte para gerenciamento de memória e eventos de sincronização são recursos interessantes no sistema, tendo em vista todas as limitações da arquitetura. Adicionando a comunicação entre tarefas, este sistema possuía uma API poderosa, capaz de suprir aplicações complexas.

3.2.2 Abassi RTOS

Com suporte para a arquitetura 8051, AVR32A, MSP430X e algumas arquiteturas *multi core*, o Abassi RTOS é um sistema gratuito, contudo de código fonte fechado. em seu site encontram-se para *download* apenas os binários, dificultando o trabalho de suporte para aplicações baseadas no sistema [50]. Isso torna-se mais crítico em arquiteturas como a do Intel 8051, onde vários fabricantes fazem alterações nos registradores de controle do processador, tornando-se necessário alterar trechos do código fonte do sistema para compatibilização com o processador.

Atualmente o sistema é construído em cima do compilador Keil C51 [42], o que o torna inacessível para usuários que não possuam o compilador ou alguma IDE com ele incluso [12]. Os principais recursos do sistema são [51]:

- Funções para configurações de interrupções e recursos de *timer* e comunicação serial;
- Mutexes e semáforos;
- Escalonamento com prioridade;
- Escalonamentos do tipo FCFS e *round robin*;
- Troca de prioridade em tempo de execução;
- Eventos;
- Troca de mensagens entre tarefas.

Devido ao fato de o sistema ser de código fonte fechado, sua documentação não nos traz muitos detalhes acerca dos algoritmos implementados para troca de contexto, escalonamento, sincronismo e demais funcionalidades. Felizmente, a documentação nos dá uma boa ideia dos *footprints* de código e memória do sistema, sendo estes de 12350 bytes e aproximadamente 215 bytes, respectivamente. Para obter estes valores considerou-se o pior cenário para o sistema, utilizando todas as funcionalidades e o maior número de tarefas e mutexes/semáforos possíveis. Caso o usuário deseje adicionar interrupções, o *footprint* de código será aumentado em 647 bytes, no pior cenário.

3.2.3 Keil RTX51

Mesmo fazendo parte da pilha de recursos presentes no compilador C51 da Keil, este sistema operacional de tempo real foi descontinuado pela empresa. Logo o seu uso torna-se injustificável ou impraticável, pois todo o suporte para o mesmo acabou e a situação não fica melhor se levarmos em conta que ele somente estará disponível para o usuário que comprar um kit de desenvolvimento como o PK51 [52] por um valor elevado.

Ainda existe certo suporte por meio da Keil para a versão *tiny* do RTX51, contudo este *kernel* é projetado para sistemas extremamente limitados, não contendo semáforos, troca de mensagens e nem um escalonador preemptivo [53]. Deste modo, pouca contribuição tem para o presente trabalho estudar o *kernel* desta versão reduzida do sistema.

Segundo a Keil, o RTX51 possuía as seguintes características [54]:

- Escalonador preemptivo ou cooperativo;
- Escalonamento por algoritmo *round robin*;
- Escalonamento com 4 níveis de prioridade e possibilidade de 19 tarefas ativas;
- Eventos;
- Comunicação entre tarefas e semáforos.

Quanto ao *footprint* temos, para o código, 8 Kbytes no pior cenário e para a memória 896 bytes. Entretanto, o *footprint* de memória pode ser maior, uma vez que o sistema precisa de, no mínimo, 650 bytes de memória externa, podendo aumentar esta quantidade conforme o usuário utilize mais recursos.

Infelizmente, assim como no caso do KR-51, não conseguimos ter mais informações acerca deste RTOS e os algoritmos implementados por ele. Apesar de ter sido descontinuado, seria de grande valia se a Keil abrisse seu código fonte, de modo a permitir que desenvolvedores o utilizem e modifiquem.

3.2.4 805x RTOS

Com um *kernel* totalmente *open source* e coberto pela licença BSD, este RTOS é perfeito para aplicações dos mais diversos tipos, tendo em vista que a licença permite que os usuários modifiquem o código fonte sem terem a obrigatoriedade de divulgar suas modificações, como ocorre em licenças do tipo GPL. Apesar de ser construído para o compilador C51, é possível que desenvolvedores com certo grau de experiência portem o sistema para outros compiladores, como o SDCC. O sistema possui uma documentação ampla e detalhada, mostrando as funções do mesmo, os tipos de variáveis e entrando em questões técnicas, facilitando o trabalho de alteração do seu *kernel* [55].

Com enfase, este RTOS serviu como base para o desenvolvimento das *macros* utilizadas no processo de salvar e recuperar a pilha de cada tarefa durante a troca de contexto do presente trabalho. Outro fato que chama a atenção é que a ultima atualização que consta no site em que o sistema é disponibilizado data do ano de 2012, colocando-o como um dos sistemas mais atualizados para a plataforma no momento em que este trabalho é escrito.

As principais funcionalidades do sistema são [56]:

- Escalonador com prioridades, utilizando o algoritmo *round robin*;

- Escalonador preemptivo ou cooperativo;
- Mutexes e semáforos;
- Numero máximo de tarefas configurável de acordo com necessidades do usuário;
- Possibilidade de suspender tarefas;
- *Traps* definidas pelo usuário no caso de funcionamento inadequado.

Com um *footprint* de código de apenas 2.4 Kbytes e de memória de 105 bytes, este é um dos sistemas mais leves em comparação com os outros sistemas vistos neste capítulo. Também, ele possui uma alta possibilidade de personalização, podendo o usuário escolher por escalonamento cooperativo caso seja de seu interesse, ou aumentar o número máximo de tarefas que o sistema suporta, dentre outras possibilidades. Os *footprint* de código e memória, contudo, aumentam conforme mais tarefas são adicionadas ao sistema.

3.2.5 Outros sistemas disponíveis

O site OSRTOS [57] nos traz uma lista de sistemas operacionais de tempo real. Dentre eles, alguns possuem certo grau de suporte para a arquitetura Intel 8051, contudo seus focos não são os mesmos que os focos deste trabalho. Tendo este fato em mente, nesta seção vamos nos ater a listar alguns destes RTOS e seus principais focos de trabalho.

- NuttX: Este é um RTOS completo, com uma ampla gama de recursos muito semelhante ao que é encontrado no FreeRTOS. Arquiteturas como ARM, AVR, PIC e MIPS são suportadas pelo sistema, além da arquitetura 8051. Entretanto, o suporte para a última é, no mínimo, reduzido, sendo que a documentação do sistema não cita a mesma, deixando para o usuário todo o trabalho de portar e compatibilizar a aplicação [58];
- QuarkTS: Muito semelhante ao RTX51 Tiny, este RTOS é designado para aplicações com grandes limitações em termos de recursos de *hardware* e memória. Contendo apenas um escalonador cooperativo, o QuarkTS suporta arquiteturas como ARM, AVR, PIC, ColdFire e MIPS, além da 8051 [59];
- Contiki OS: Apesar de estar na lista de RTOS's, este sistema tem por intuito trazer funcionalidades de rede para arquiteturas mais limitadas. Ele implementa toda a pilha do protocolo IP, dando pouca prioridade para algoritmos de escalonamento. Todavia, este sistema oferece suporte para as arquiteturas MSP430, ARM, AVR e 8051 [60];

- Protothreads: Por último, este RTOS é extremamente limitado e fornece apenas suporte para as arquitetura AVR e 8051. Cada tarefa ocupa apenas 2 bytes de memória [61], o que nos leva a crer que ele se trata de um escalonador cooperativo, hipótese reforçada pelo fato do sistema ser escrito totalmente em linguagem C, o que impossibilita manipulações na pilha do sistema. Contudo, é um sistema interessante para aplicações com tarefas breves e com grandes limitações de recursos [17].

3.3 Comparativo entre os RTOS's estudados

Agora que o cenário em que este trabalho se encontra foi elucidado, é válido fazermos um breve comparativo entre os sistemas de tempo real mencionados acima. A Tabela 4 trás os dados acerca dos principais sistemas operacionais estudados neste capítulo. Sistemas como o NuttX e o Protothreads foram deixados de fora da tabela por não apresentarem um nível de suporte satisfatório ou não cumprirem com as características desejadas em um RTOS's, conforme explicado no Capítulo 2.

Com estes dados podemos passar para o próximo capítulo, onde será feita uma discussão sobre as características desejadas para o projeto, bem como os algoritmos que foram escolhidos para construção do *kernel*. Posteriormente, no Capítulo 5, estes algoritmos serão discutidos mais a fundo, trazendo trechos de código onde eles foram implementados.

Tabela 4 – Comparativo entre RTOS's

Nome	***Footprint* de código (KB)**	***Footprint* de memória (bytes)**	**Escalonador**
FreeRTOS	5 - 10	250	*Round robin* com prioridade
KR-51	Não informado	Não informado	Não informado
Abassi	1.23	215	*Round robin* ou FCFS com prioridade
RTX51	6 - 8	896	Preemptivo *round robin* com prioridade ou cooperativo
805x	2.4	105	Preemptivo *round robin* com prioridade ou cooperativo

Fonte – O autor

4 Implementação

Os capítulos anteriores serviram de base para que o leitor pudesse compreender melhor como sistemas operacionais funcionam e, mais especificamente, o que é esperado de um RTOS. No Capítulo 3 trouxemos uma série de exemplos de sistemas operacionais desta classe, tornando possível o entendimento do cenário atual em termos de desenvolvimento de RTOS's para a família 8051 e quais características são de interesse para o presente trabalho.

Neste capítulo entraremos mais a fundo na implementação do RTOS desenvolvido, de modo que as seções que seguem dissertarão acerca das características desejadas para o projeto, da implementação do escalonador, do funcionamento dos algoritmos de sincronização utilizados e de outras funcionalidades e características do sistema.

4.1 Características desejadas

4.1.1 Tamanho desejado

Por padrão, a família 8051 possui 64 kbytes de memória ROM, tipicamente *flash*, para o armazenamento de código [17], o que pode parecer um espaço significativo para aplicações em um processador de 8 bits. Contudo, mesmo uma aplicação simples pode possuir um grande *footprint*, sendo o simples uso de semáforos e diferentes tarefas capaz de gerar códigos que ocupem 10% desta capacidade de armazenamento [46]. Sendo assim, o sistema a ser projetado deve ser leve, contendo um *footprint* de código que não ultrapasse 3 Kbytes no pior cenário.

Ainda mais crítico é o *footprint* de memória. Na arquitetura convencional do 8051 temos apenas 256 bytes de memória RAM, a ser compartilhada com bancos de registradores, variáveis do tipo bit, a pilha do sistema e os SFR's de cada processador [35]. Neste cenário, existem poucas possibilidades para um RTOS, possibilitando apenas o uso de escalonadores cooperativos. Para contornarmos esta limitação iremos trabalhar apenas com processadores baseados na arquitetura 8051 expandida, também chamada de 8052 [17], onde existem 2 Kbytes de memória externa disponível para o usuário via acesso indireto.

Na prática, usar a memória externa disponível nestes processadores não irá alterar o método como a pilha irá ser manipulada ou a localização da mesma no sistema. A única mudança está no método de acesso à memória, conforme vimos no Capítulo 2. Alguns processadores baseados no 8051 possibilitam que a pilha seja expandida e utilize parte da memória externa no sistema, contudo não trataremos esta funcionalidade no trabalho, tendo em vista que o tamanho da pilha nas aplicações tende a não ultrapassar o limite de 25 bytes, o acesso à pilha irá ser mais lento, uma vez que ocorrerá por intermédio

do registrador DPTR e alguns processadores da família não possuem suporte para tal expansão.

Mesmo utilizando este modelo expandido de memória, ainda precisamos ter um grande cuidado quanto ao uso de memória RAM do sistema, sabendo alocar nos espaços corretos cada tipo de variável. Assim, as estruturas utilizadas para os mecanismos de sincronização e as tarefas do sistema devem ser projetadas de modo a armazenar apenas informações críticas para o funcionamento do mesmo. Com isso em mente, vamos adotar que o sistema final deva ter um *footprint* de memória menor que 200 bytes.

4.1.2 Algoritmo de escalonamento adotado

Comumente em sistemas operacionais de tempo real, utiliza-se o algoritmo *round robin* para o escalonamento de tarefas. Existem várias vantagens em utilizar este algoritmo, dentre elas estão a sua simplicidade e portabilidade. Enquanto outros algoritmos requerem um grande conhecimento acerca dos requisitos temporais do sistema e, muitas vezes, são construídos para atender uma aplicação específica, o *round robin* atende a grande maioria das aplicações sem a necessidade de ter um conhecimento prévio do sistema.

No capítulo anterior também foi constatado que os RTOS's oferecem a possibilidade de dar prioridades para as tarefas no escalonamento. De maneira geral, escalonadores *round robin* com prioridade trazem a grande vantagem de assegurar que tarefas mais críticas sejam atendidas antes que ocorra o seu *deadline*, entretanto, é necessário tomar certo cuidado ao utilizar prioridades em algoritmos de escalonamento, pois estas podem gerar *starvation*, conforme visto no Capítulo 3.

Devido aos problemas de *starvation* que podem ser causados pela adição de prioridades no escalonador, faz-se necessário implementar mecanismos de envelhecimento, onde as prioridades das tarefas vão diminuindo com o tempo, de modo a assegurar que todas serão atendidas. Estes mecanismos consomem mais recursos do sistema e aumentam a complexidade da implementação do escalonador. Devido a isso, o presente trabalho se limitará a implementar apenas um algoritmo *round robin* sem prioridade, prezando pela simplicidade do código fonte e pela obtenção dos menores *footprints* de memória e código possíveis.

4.1.3 Algoritmos de sincronização

A implementação de mecanismos de sincronização pode ocorrer de diversas maneiras, conforme visto nos capítulos anteriores. Neste trabalho iremos implementar mecanismos de semáforos e mutexes, deixando ainda disponível para o usuário a possibilidade de desabilitar interrupções e variáveis binárias para sincronização via instruções atômicas.

Elencados os dados técnicos do sistema, podemos inseri-los na Tabela 5, visualizando de maneira resumida o que é esperado do sistema final. Nas próximas seções iremos

discutir a fundo os algoritmos presentes nos RTOS, tendo uma visão mais técnica de como o escalonador e os mecanismos de sincronização foram implementados, além das demais funcionalidades criadas.

Tabela 5 – Características técnicas do sistema

***Footprint* de memória**	***Footprint* de código**	**Escalonador**	**Sincronização**
200 bytes	3 Kbytes	*Round robin* sem prioridade	Semáforos, Mutexes, variáveis binárias, desabilitação de interrupções

Fonte – O autor

4.2 Implementação do escalonador

Em um sistema multitarefas é essencial que exista um escalonador, uma vez que este componente é o responsável pela manipulação das tarefas, liberando os recursos de *hardware* de maneira eficaz, conforme discutido no Capítulo 2. Para isso, é crucial que o sistema possua uma estrutura de dados capaz de capturar uma imagem de cada tarefa para, no momento da troca de contexto, poder recuperá-la.

Vimos anteriormente que em GPOS's esta estrutura é denominada PCB, ou bloco de controle de processo, em tradução livre. Um PCB contém um grande número de informações acerca de cada processo e, em muitos casos, pode ocupar todo o espaço de memória disponível em um sistema embarcado. Já em RTOS's trabalha-se com TCB's, ou blocos de controle de tarefa, sendo estes estruturas mais simples que guardam informações como apontador de programa, uma cópia da pilha do processador, estado da tarefa e código de identificação da mesma. Um exemplo de TCB pode ser visto na Figura 24.

A implementação do TCB é um dos passos mais importantes no desenvolvimento de um escalonador, isso pois é necessário que o desenvolvedor conheça bem a arquitetura com a qual está trabalhando e quais os dados importantes a serem salvos para que o sistema consiga funcionar corretamente. Alguns dos dados que precisam ser considerados durante esta etapa de desenvolvimento são os principais registradores do processador, a localização de sua pilha e do apontador de programa. No Capítulo 2 vimos os principais registradores de um processador 8051, logo iremos incluí-los na cópia da pilha, de modo a salvar uma imagem do estado em que eles estavam na última troca de contexto.

Além destes registradores, o compilador SDCC ainda cria um vetor denominado bits. Neste vetor são armazenados os dados acerca das variáveis binárias presentes na arquitetura (endereços 20 a 2F da memória interna). Deste modo, é importante também salvar este vetor, para o caso em que o usuário utilize alguma variável deste tipo.

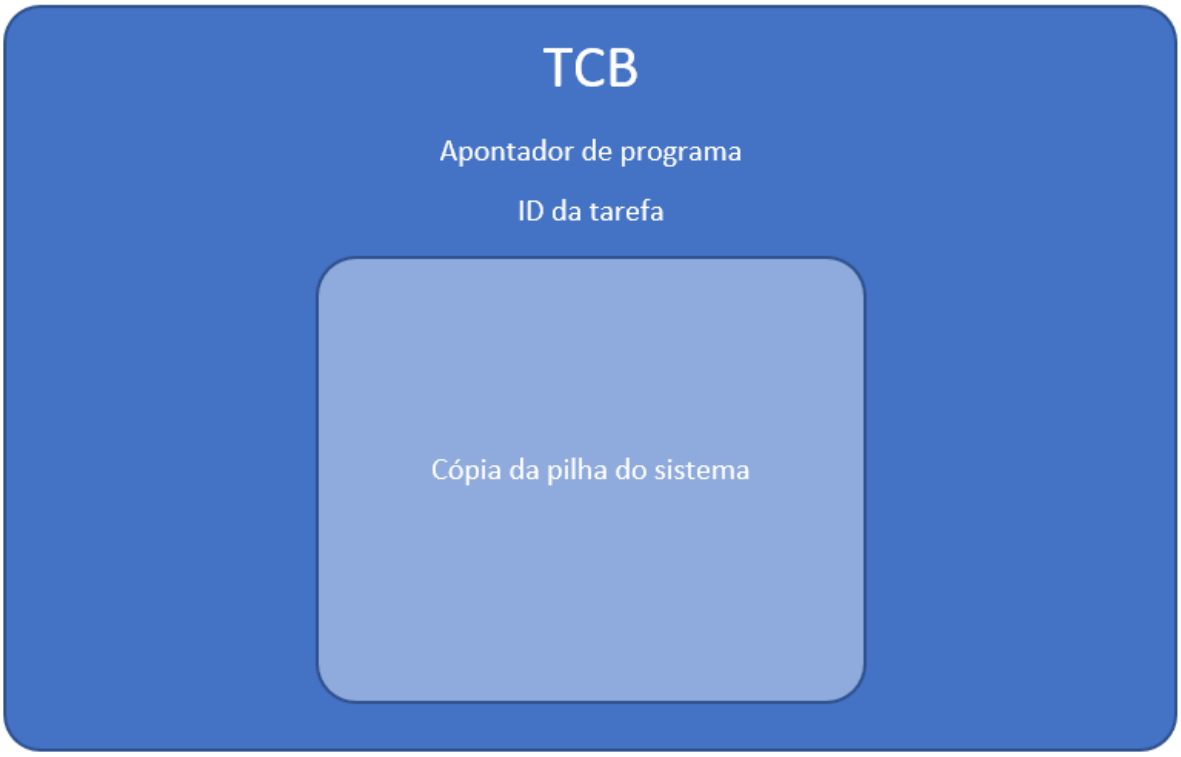

Figura 24 – Exemplo de TCB. [Fonte: O autor]

Na família 8051 o apontador de programa se encontra nos dois últimos endereços da pilha. Assim, sempre que o programa encontrar uma instrução de retorno (ret ou reti) a seguinte sequência de eventos ocorrerá [62]:

- Os dados presentes no endereço apontado por SP são carregados para o byte mais significativo do apontador de programa;
- Decrementa-se SP em uma unidade;
- Carrega-se os dados no novo endereço apontado por SP para o byte menos significativo do apontador de programa;
- Decrementa-se SP em uma unidade.

Entretanto, a arquitetura 8051 não deixa o apontador de programa acessível para o usuário. Se olharmos os códigos assembler gerados pelos compiladores, o retorno de funções e ISR's possui apenas as instruções ret ou reti, sem nenhum parâmetro. Consequentemente, para salvarmos o apontador de programa precisamos salvar a pilha do processador.

Como a pilha aumenta e diminui conforme dados são colocados e retirados da mesma, precisamos saber qual o seu tamanho no momento da troca de contexto, de modo que os processos de salvamento e recuperação dela ocorram corretamente. Para isso, cada TCB

possui uma variável para armazenar o tamanho da pilha no momento da troca de contexto. Por fim, inclui-se no TCB uma variável de estado da tarefa, para que o escalonador saiba quais tarefas estão disponíveis para serem escalonadas. A Figura 25 mostra o bloco de controle de tarefa criado para o RTOS em questão.

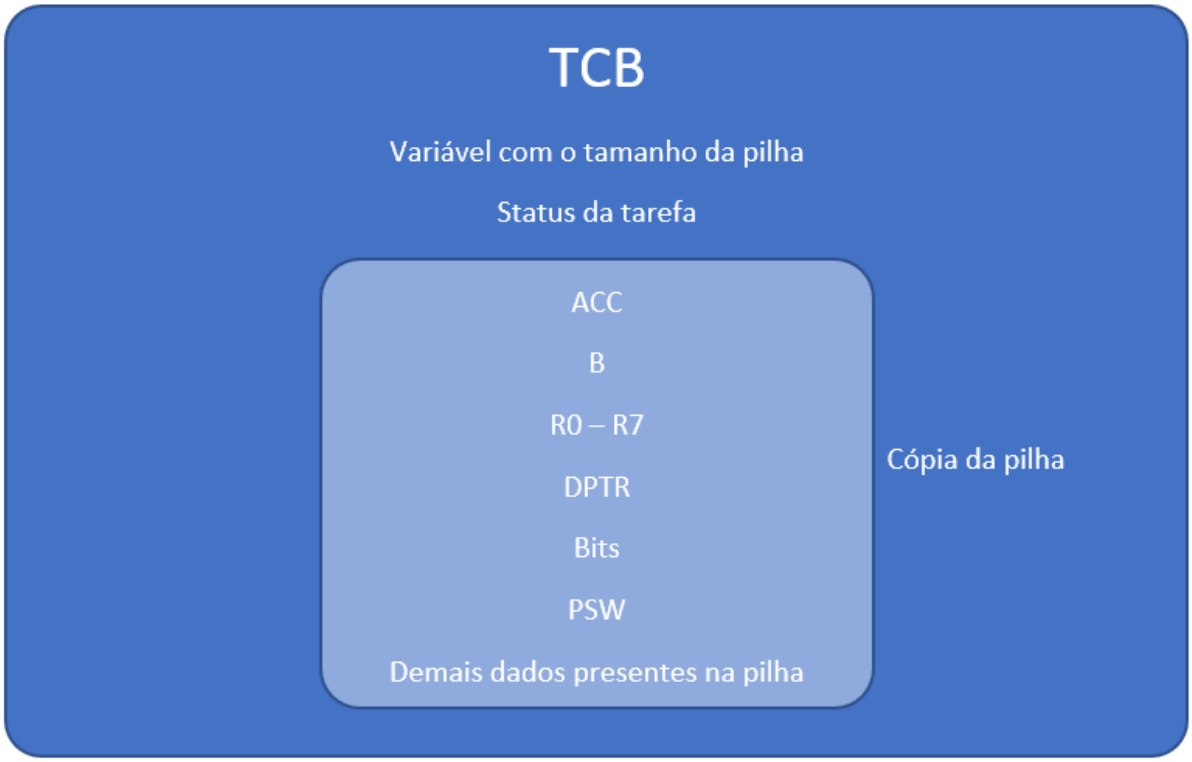

Figura 25 – TCB utilizado no RTOS. [Fonte: O autor]

O tamanho do TCB varia conforme a configuração do sistema. Se adotarmos uma pilha com tamanho de 20 bytes, por exemplo, cada TCB ocupará 22 bytes na memória. Assim sendo, cada TCB possui um tamanho equivalente ao tamanho máximo da pilha, definido na macro *SCH_STACK_SIZE*, mais dois bytes para as variáveis de *status* da tarefa e tamanho atual da pilha.

Para que o escalonador saiba quais tarefas estão disponíveis para serem escalonadas é necessário termos um vetor contendo as informações de cada tarefa. Este vetor é chamado de lista de tarefas e possui um tamanho correspondente ao número máximo de tarefas suportadas pelo sistema, configurado na macro *SCH_MAX_TASKS*. Cada posição corresponde a uma tarefa e seu valor indica o estado em que aquela tarefa está no momento do escalonamento. A Figura 26 ilustra esta estrutura de dados, onde n é o número máximo de tarefas que o sistema comporta.

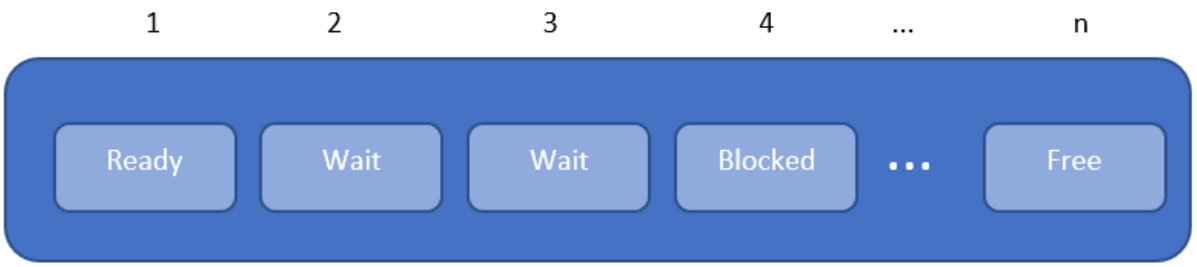

Figura 26 – Fila de tarefas do sistema. [Fonte: O autor]

Quanto aos estados em que uma tarefa pode estar, temos os seguintes:

- *Free*: Nenhuma tarefa foi alocada no endereço, estando este disponível para uma nova tarefa;
- *Blocked*: A tarefa está bloqueada por tentar obter algum mutex ou semáforo já obtido por outra tarefa;
- *Wait*: A tarefa esta disponível para ser escalonada;
- *Ready*: A tarefa esta em execução.

Com isso podemos criar um primeiro ciclo de vida simplificado para as tarefas do RTOS, conforme a Figura 27. Na seção de funções extras do RTOS iremos expandir este ciclo de vida, adicionando as chamadas de funções do sistema que fazem as tarefas irem de um estado para o outro.

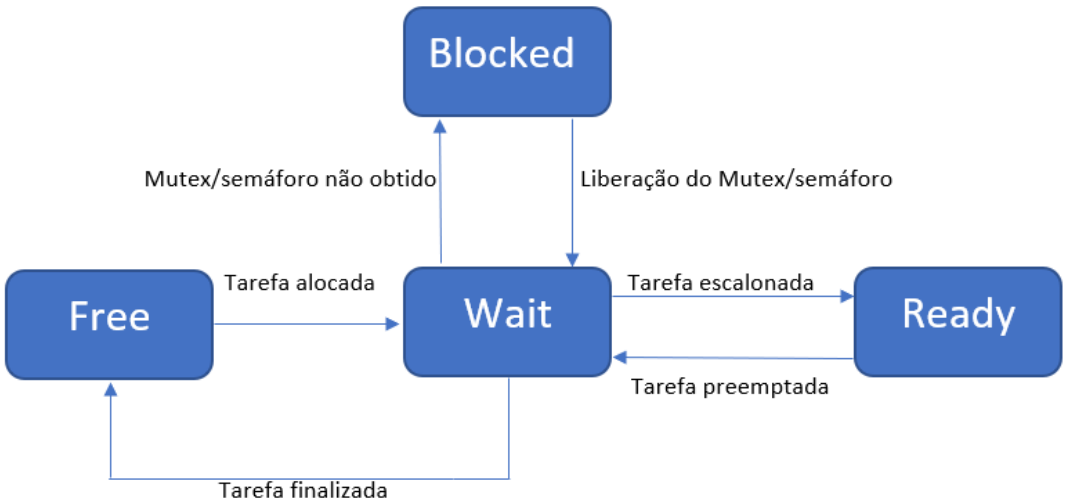

Figura 27 – Ciclo de vida resumido de uma tarefa. [Fonte: O autor]

Antes de partirmos para o estudo do escalonador, precisamos analisar como o RTOS é inicializado e como as tarefas são adicionadas na lista de tarefas do sistema. A inicialização

do sistema da-se por meio da função *sch_init*, nesta inicialização todas as tarefas são postas no estado *free*, deixando seu endereço livre para ser alocado por novas tarefas. Além disso, duas variáveis do sistema são inicializadas sendo elas *sch_index* e *sch_num_tasks*. A finalidade destas variáveis é, respectivamente, saber qual a atual tarefa em execução pelo sistema e o número de tarefas disponíveis para execução. O Código 4.1 mostra o algoritmo de inicialização do sistema.

```
1  void sch_init(){
2    Byte i = 0;
3    sch_index = 0;
4    sch_num_tasks = 0;
5    sch_time = SCH_TIMEOUT;
6    for(i = 0; i < SCH_MAX_TASKS; i++){
7      sch_tasks[i].sp = 0;
8      sch_tasks[i].state = FREE;
9    }
10 }
```

Código 4.1 – Inicialização das variáveis do sistema. Fonte: O autor

Com as variáveis do sistema inicializadas, o usuário pode adicionar suas tarefas à fila do sistema. Como, neste momento, a tarefa estará sendo chamada pela primeira vez, precisa-se adicionar na sua cópia da pilha um ponteiro apontando para o endereço da mesma. Todas as tarefas do sistema são do tipo *void* e recebem como parâmetro um ponteiro *void*. Também, é interessante capturar os valores atuais dos registradores principais do sistema. No caso do vetor de bits e dos registradores R0 a R7 pode-se adotar apenas o valor de zero no momento inicial. A adição de uma tarefa pode ser conferida no Código 4.2.

```
1  void sch_add_task(fptr *f){
2    Byte i = 0;
3    for(i = 0; i < SCH_MAX_TASKS; i++){
4      if(sch_tasks[i].state == FREE){
5        break;
6      }
7    }
8
9    if(sch_tasks[i].state != FREE){
10     return;
11   }
12
13   sch_tasks[i].stack_save[0] = ((unsigned short)f) & 0xff;
14   sch_tasks[i].stack_save[1] = ((unsigned short)f >> 8) & 0xff;
15   sch_tasks[i].stack_save[2] = 0;    //BITS
16   sch_tasks[i].stack_save[3] = ACC;
17   sch_tasks[i].stack_save[4] = B;
18   sch_tasks[i].stack_save[5] = DPL;
19   sch_tasks[i].stack_save[6] = DPH;
```

```
20
21    Byte j = 0;
22
23    // r0 - r7
24    for(j = 7; j < 15; j++){
25      sch_tasks[i].stack_save[j] = 0;
26    }
27
28    sch_tasks[i].stack_save[15] = PSW;
29
30    sch_tasks[i].state = WAIT;
31    sch_tasks[i].sp = 15;
32    sch_num_tasks++;
33  }
```

Código 4.2 – Procedimento para adicionar uma tarefa no sistema. Fonte: O autor

Adicionadas as tarefas no sistema pode-se partir para a inicialização do mesmo. A rotina de inicialização pauta-se na configuração do *timer* 2 e no escalonamento da primeira tarefa. Neste primeiro momento iremos abstrair o conteúdo das macros *TO_STACK* e *POP_BANK*, assumindo apenas que elas carregam a cópia da pilha presente no TCB da tarefa a ser escalonada para a pilha do processador e restauram os valores de cada um dos principais registradores presentes nela, respectivamente. O Código 4.3 possui o algoritmo para inicialização do sistema e escalonamento da primeira tarefa.

```
1  void sch_start(){
2    EA = 0;
3
4    // timer 2 initialization:
5    T2CON = 0;
6    RCAP2H = 0xCE;
7    RCAP2L = 0xD9;
8    TH2 = 0xCE;
9    TL2 = 0xD9;
10   ET2 = 1;
11   TR2 = 1;
12
13   // scheduling the first task:
14   sch_index = 0;
15   sch_tasks[sch_index].state = READY;
16   TO_STACK
17   POP_BANK
18   EA = 1;
19 }
```

Código 4.3 – Algoritmo de inicialização do sistema. Fonte: O autor

Note que, nesta função o registrador EA é desabilitado e posteriormente habilitado novamente. Este registrador é responsável pelo funcionamento das interrupções do sistema e, caso esteja desabilitado, nenhuma interrupção irá ocorrer. Adota-se esta técnica para evitar que interrupções ocorram durante a troca de contexto ou configuração do *timer*, afetando a inicialização do sistema.

Tendo em mente como ocorre a configuração do sistema, podemos partir para a analise do escalonamento de tarefas. A estratégia adotada para este processo foi a de *time slicing*, onde o escalonador é chamado em períodos fixos de tempo para selecionar uma nova tarefa a ser rodada. O período em que o escalonador é evocado pode ser configurado pelo usuário através da macro *SCH_TIMEOUT*, tendo como valor padrão 15. Uma vez que a interrupção do *timer* foi configurada para ocorrer a cada 1ms, tem-se que o escalonamento ocorre de 15 em 15ms, podendo ter leves variações ao passo que o usuário adicione mais interrupções ao sistema.

No momento em que o escalonador é evocado ocorrem três processos, sendo eles o salvamento da pilha do processador no TCB da tarefa atual, a seleção da nova tarefa a ser rodada pelo sistema e a recuperação da cópia da pilha presente no TCB da nova tarefa. Esta sequência de eventos nada mais é do que a troca de contexto no sistema e pode ser vista de maneira simples na Figura 28.

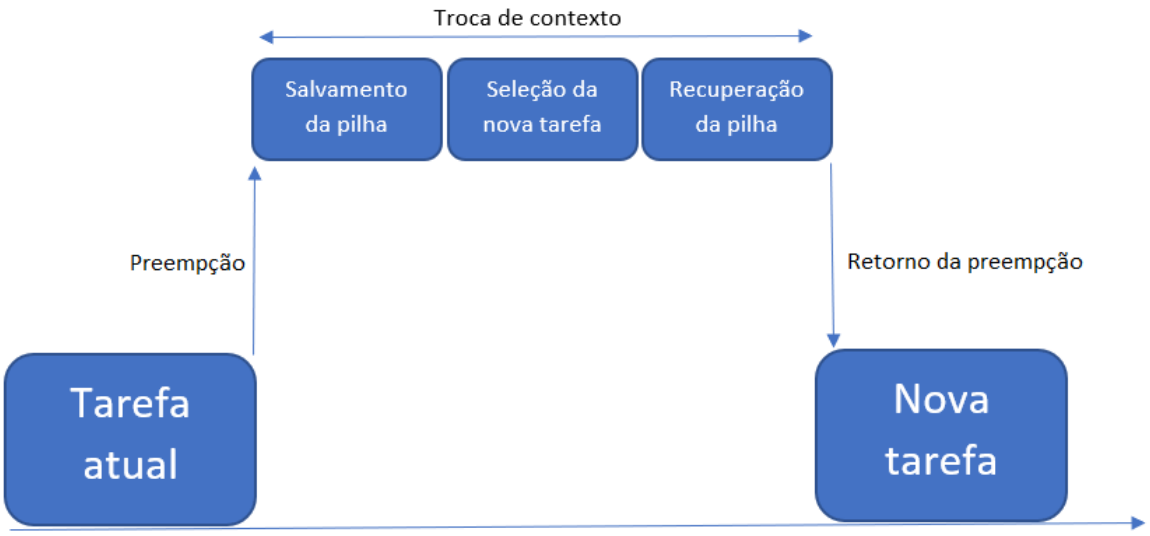

Figura 28 – Fluxo de execução durante a troca de contexto no RTOS. [Fonte: O autor]

Os processos de salvamento e recuperação da pilha no TCB são feitos através de duas macros do sistema. A macro responsável por salvar a pilha no TCB funciona da seguinte forma:

- Inicializa-se um ponteiro apontando para o primeiro endereço da pilha, configurado na macro *SCH_STACK_INIT*;

- Outro ponteiro é inicializado apontando para o primeiro endereço de memória no TCB onde a pilha será salva;
- Atualiza-se a variável do TCB que contêm o tamanho da pilha, capturando o tamanho atual da pilha do processador;
- Copia-se os valores presentes na pilha para o TCB da tarefa atual.

O processo de recuperar a pilha salva no TCB é muito semelhante, porém os dados são passados do vetor presente no TCB contendo a cópia da pilha para os endereços de memória onde a pilha está. Os algoritmos de salvamento e recuperação da pilha são vistos nos Códigos 4.4 e 4.5, respectivamente.

```
1 #define TO_XRAM { \
2   __data Byte * __data ram = (Byte __data *)SCH_STACK_INIT; \
3   __xdata Byte *__data xram = &(sch_tasks[sch_index].stack_save[0]);  \
4   sch_tasks[sch_index].sp = SP - SCH_STACK_INIT; \
5   while((Byte)ram <= SP) *(xram++) = *(ram++); }
```

Código 4.4 – Macro para salvamento da pilha do sistema. Fonte: O autor

```
1 #define TO_STACK { \
2   __data Byte * __data ram = (Byte __data *)SCH_STACK_INIT; \
3   __xdata Byte *__data xram = &(sch_tasks[sch_index].stack_save[0]);  \
4   SP = SCH_STACK_INIT + sch_tasks[sch_index].sp; \
5   while((Byte)ram <= SP) *(ram++) = *(xram++); }
```

Código 4.5 – Macro para recuperação da pilha do sistema. Fonte: O autor

A ultima etapa para compreendermos como o escalonamento funciona no RTOS é estudarmos o algoritmo responsável pela seleção da tarefa. Conforme discutido, o trabalho baseou-se em um escalonador do tipo *round robin*, ou seja, quando na troca de uma tarefa para a próxima o escalonador irá percorrer a lista de tarefas e selecionar a próxima disponível para ser escalonada. Caso não exista nenhuma outra tarefa disponível, continua-se a execução da atual.

Caso o sistema não possua nenhuma tarefa na fila ele irá colocar o processador no modo de economia de energia. Na família 8051, isso é feito através do registrador PCON, atribuindo a ele o valor 2 caso o usuário queira que o controlador fique no estado de economia de energia até ser reiniciado ou o valor 50, caso ele queira que o interrupções do *timer* 2 façam o processador sair do modo de economia de energia. Neste modo, nenhum processamento é feito, ficando o processador no estado *sleep* até ser acordado. A Figura 29 ilustra o processo de seleção de tarefas.

Como o compilador SDCC gera instruções para carregar os valores dos principais registradores na pilha quando o algoritmo entra em uma ISR [35] não é necessário, neste momento, utilizar macros para tal finalidade. Contudo, caso uma tarefa queira, voluntariamente, liberar recursos de processamento para outra, é necessário utilizar estas macros.

Neste momento vamos nos ater a implementação do escalonador, discutindo estes detalhes na seção de funções extras do RTOS.

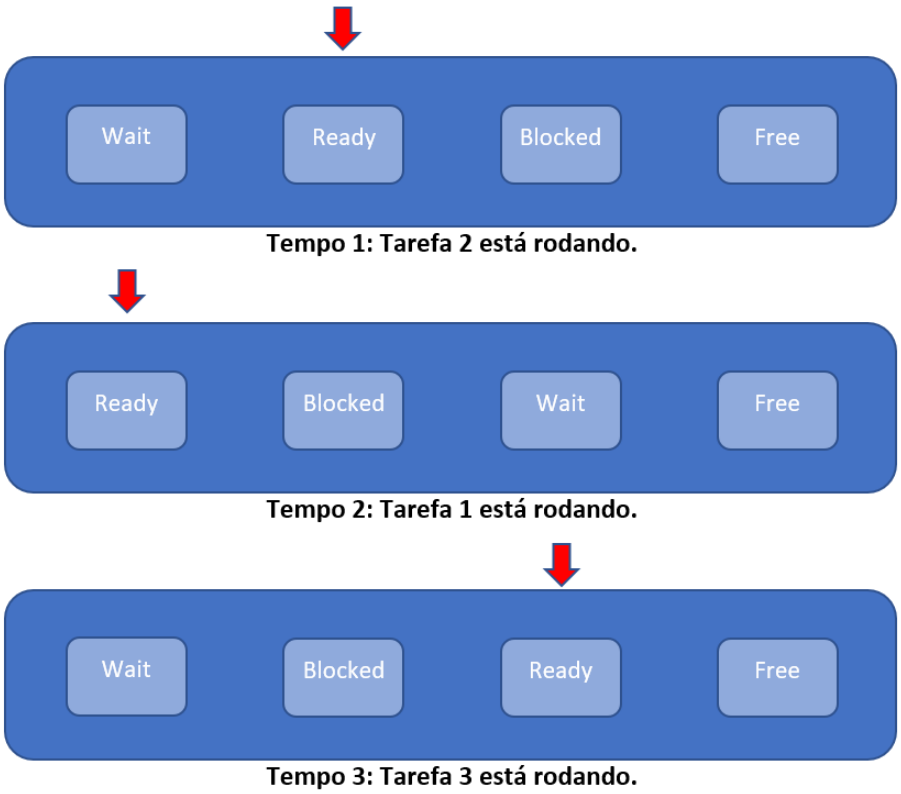

Figura 29 – Seleção de tarefas pelo algoritmo *round robin*. [Fonte: O autor]

Uma vez vistas as principais estruturas de dados do escalonador e os algoritmos utilizados para escalonamento de tarefas, podemos descrever o funcionamento do escalonador nos seguintes passos:

1. O escalonador libera os recursos de *hardware* para uma tarefa;
2. A cada intervalo de tempo definido pelo usuário o escalonador entra em ação, selecionando a próxima tarefa a receber os recursos;
3. Quando na troca de contexto, a tarefa atualmente em execução salva uma cópia da pilha em seu TCB, bem como o tamanho da mesma no instante da troca;
4. A tarefa selecionada, por sua vez, carrega a cópia da pilha presente em seu TCB para a pilha do processador, fazendo com que a tarefa retorne para o ponto em que parou no momento em que foi preemptada ou liberou os recursos de *hardware* voluntariamente;
5. Este processo repete até que nenhuma tarefa esteja mais disponível no sistema ou o mesmo for desligado.

4.3 Sincronização de tarefas

Para solucionar os problemas de sincronismo ocasionados pela concorrência, foram implementados três mecanismos de sincronização no RTOS, sendo eles mutexes, semáforos e uso de instruções binárias *test_and_set()*, além da possibilidade de o usuário desabilitar as interrupções do sistema. Nas subseções que seguem vamos analisar a fundo o funcionamento de cada mecanismo e das funções disponíveis na biblioteca do sistema para a manipulação das estruturas de dados correspondentes aos mesmos.

4.3.1 Mutexes

A grande maioria dos problemas de sincronização gerados pela concorrência entre tarefas pode ser resolvida com o uso de mutexes, fazendo deste tipo de mecanismo um dos mais populares em sistemas operacionais. No RTOS criado, mutexes são acessados via estruturas de dados denominadas *sch_mutex_sync* que contêm uma variável com a chave do mutex, indicando quando ele está livre ou bloqueado, e uma lista de tarefas bloqueadas esperando para obter a trava do mutex.

Cada estrutura de mutex possui um tamanho igual ao número máximo de tarefas configurado na macro *SCH_MAX_TASKS* mais um byte para a variável de trava. Por exemplo, por padrão o sistema possui um número máximo de dez tarefas, logo cada mutex ocupará 11 bytes de memória. Apesar de ser possível que o usuário manipule a estrutura de dados do mutex diretamente, é aconselhável que o único meio de contato dele com a mesma seja através das funções disponíveis na biblioteca do sistema.

Ao todo estão disponíveis quatro funções para o uso de mutexes, sendo elas *sch_mutex_start*, *sch_mutex_lock*, *sch_mutex_trylock* e *sch_mutex_release*. O funcionamento destas funções pode ser brevemente descrito da seguinte maneira:

- *sch_mutex_start*: Função para inicializar a estrutura de dados do mutex. Esta função recebe como parâmetros a estrutura de dados e o estado em que o mutex deve começar (bloqueado ou livre). Além disso, ela inicializa o vetor de tarefas bloqueadas pelo mutex, deixando todas as tarefas livres;

- *sch_mutex_lock*: Função para uma tarefa tentar obter a trava do mutex e entrar em sua região crítica. Esta função é bloqueante, ou seja, caso a tarefa não obtenha a trava ela irá ficar bloqueada até que o mutex seja obtido pela tarefa.

- *sch_mutex_trylock*: Função para obtenção de um mutex. Diferentemente da função *sch_mutex_lock*, esta é não bloqueante, retornando um valor igual a 1 quando o mutex for obtido ou igual a 0 caso contrário.

- *sch_mutex_release*: Função para que uma tarefa libere o mutex. Esta função deve ser chamada quando a tarefa sair de sua seção crítica, caso contrário o mutex ficará bloqueado, podendo gerar um *deadlock*.

O Código 4.6 possui o algoritmo presente na função *sch_mutex_start*. Note que, caso o usuário passe um valor de estado diferente dos valores definidos nas macros *MUTEX_RELEASED* e *MUTEX_LOCKED*, o mutex será iniciado como bloqueado. Nenhum valor é retornado pela função, uma vez que ela irá apenas configurar os valores presentes na estrutura de dados passada como parâmetro.

```
1  void sch_mutex_start(struct sch_mutex_sync *mut, Byte state){
2    Byte i;
3    // Mutex starting released.
4    mut->lock = MUTEX_RELEASED;
5    if(state == MUTEX_LOCKED || state == MUTEX_RELEASED){
6      mut->lock = state;
7    }
8    for(i = 0; i < SCH_MAX_TASKS; i++){
9      mut->waiting_list[i] = 0;
10   }
11 }
```

Código 4.6 – Inicialização de um mutex. Fonte: O autor

Devido ao tamanho limitado da pilha do processador 8051, os compiladores adotam a estratégia de alocar os parâmetros e variáveis locais de funções em uma área da memória interna do processador, a ser compartilhada entre todas as funções. No caso do SDCC, isso é chamado de *overlay* [35] e tem o potencial de gerar problemas em sistemas multi-tarefa. Vamos supor o seguinte exemplo:

- Duas tarefas chamam funções diferentes durante a sua execução, vamos nomeá-las de função A e função B;
- As duas funções compartilham os mesmos endereços de memória para suas variáveis locais devido ao *overlay*;
- Vamos supor que, ao ser preemptada, a primeira tarefa estava no meio da execução da função A;
- Ao entrar na função B, a segunda tarefa estará alterando os dados de memória que estão sendo utilizados pelas função A;
- Quando no retorno para a primeira tarefa, a função A estará manipulando dados que foram corrompidos devido a função B ter sido evocada pela segunda tarefa.

Apesar da limitação poder ser resolvida pelo uso de mutexes, ela permanece no caso das funções para manipulação da estrutura de dados dos mesmos, em especial na função *sch_mutex_lock*. Para solucionar isso, utiliza-se a *pragma nooverlay* que, em essência, serve como um comando para que o compilador saiba que os parâmetros e variáveis locais de determinada função precisam de uma área reservada de memória, sem que ocorra *overlay*. O Código 4.7 ilustra o uso desta funcionalidade do compilador.

```
#pragma nooverlay
void sch_mutex_lock(struct sch_mutex_sync *mut){
  EA = 0;
  while(1){
    if(mut->lock == MUTEX_RELEASED){
      mut->lock = MUTEX_LOCKED;
      EA = 1;
      break;
    }else{
      sch_tasks[sch_index].state = BLOCKED;
      mut->waiting_list[sch_index] = 1;
      sch_next();
      EA = 0;
    }
  }
}
```

Código 4.7 – Função bloqueante para a obtenção da trava de um mutex. Fonte: O autor

Ademais, os Códigos 4.8 e 4.9 ilustram os algoritmos utilizados para obter a trava do mutex de maneira não bloqueante e para liberar o mutex para outras tarefas, respectivamente.

```
#pragma nooverlay
Byte sch_mutex_trylock(struct sch_mutex_sync *mut){
  EA = 0;
  if(mut->lock == MUTEX_RELEASED){
    mut->lock = MUTEX_LOCKED;
    EA = 1;
    return 1;
  }else{
    EA = 1;
    return 0;
  }
}
```

Código 4.8 – Função não bloqueante para a obtenção da trava de um mutex. Fonte: O autor

```
#pragma nooverlay
Byte sch_mutex_release(struct sch_mutex_sync *mut){
```

```
3   EA = 0;
4   Byte i;
5   if(mut->lock == MUTEX_RELEASED){
6     EA = 1;
7     return 0;
8   }else{
9     mut->lock = MUTEX_RELEASED;
10    for(i = 0; i < SCH_MAX_TASKS; i++){
11      if(mut->waiting_list[i] == 1 && sch_tasks[i].state == BLOCKED){
12        sch_tasks[i].state = WAIT;
13        mut->waiting_list[i] = 0;
14      }else if(mut->waiting_list[i] == 1 && sch_tasks[i].state == FREE){
15        mut->waiting_list[i] = 0;
16      }
17    }
18    EA = 1;
19    return 1;
20  }
21 }
```

Código 4.9 – Função para liberação do mutex. Fonte: O autor

Por fim, a implementação de mutexes pelo sistema adota a técnica de desabilitar interrupções durante a manipulação da estrutura de dados referente aos mesmos. Isso é necessário pois o escalonador pode preemptar uma tarefa enquanto ela estiver entre os processos de conferir o valor da variável de trava no mutex e alterá-lo, bloqueando o mesmo, fazendo com que o sistema apresente comportamento errôneo e os dados não estejam devidamente sincronizados.

4.3.2 Semáforos

Conforme visto no Capítulo 2, semáforos trabalham de maneira semelhante aos mutexes, porém ao invés de possuírem apenas uma chave, possibilitando que apenas uma tarefa entre em sua região crítica, eles possuem mais do que uma chave, conforme configurado pelo usuário. No caso do RTOS criado, os semáforos podem ter um número máximo de 256 tarefas entrando na região crítica, este número ainda pode ser configurado pelo usuário.

A estrutura de dados do semáforo possui um byte a mais do que a estrutura de um mutex, por possuir uma variável correspondente ao número máximo de chaves que o semáforo poderá ter. Isso pode ser verificado no Código 4.10, correspondente a função de inicialização de um semáforo. Note que, assim como nos mutexes, os semáforos possuem em sua estrutura uma lista com as tarefas que estão bloqueadas por tentarem obter seus recursos sem sucesso.

```
1 void sch_semaphore_start(struct sch_semaphore_sync *sem, Byte size){
```

```
2   sem->lock = size;
3   sem->share = size;
4   Byte i;
5   for(i = 0; i < SCH_MAX_TASKS; i++){
6     sem->waiting_list[i] = 0;
7   }
8 }
```

Código 4.10 – Inicialização de um semáforo. Fonte: O autor

Além da função *sch_semaphore_start*, o RTOS possui as seguintes funções para o uso de semáforos:

- *sch_semaphore_get*: Função bloqueante para obtenção da trava de um semáforo. A cada trava adquirida a variável *lock* presente na estrutura do semáforo é decrementada em uma unidade, caso esta variável seja zerada e uma tarefa tente adquirir a trava, esta ficará bloqueada até que o valor de *lock* seja incrementado por outra tarefa através de uma chamada da função *sch_semaphore_put*;

- *sch_semaphore_tryget*: Função não bloqueante para a obtenção da trava de um semáforo. Ao contrário da função *sch_semaphore_get*, caso uma tarefa tente obter a trava de um semáforo cuja variável *lock* esteja zerada, esta tarefa não será bloqueada e poderá continuar rodando;

- *sch_semaphore_put*: Função para adicionar uma trava ao semáforo. Sempre que for chamada a variável*lock* será incrementada em uma unidade, até que seu valor atinja o valor máximo configurado na inicialização do semáforo.

estas funções podem ser conferidas, respectivamente, nos Códigos 4.11, 4.12 e 4.13. Note que, assim como nas funções para manipulação de mutexes, as funções de manipulações de semáforos utilizam o *pragma nooverlay*, o que deve ser levado em consideração durante o projeto, uma vez que esta técnica aumenta o uso de memória do sistema. Por fim, a implementação de semáforos segue a mesma estratégia de desabilitar interrupções vista na implementação de mutexes.

```
1 #pragma nooverlay
2 void sch_semaphore_get(struct sch_semaphore_sync *sem){
3   EA = 0;
4   while(1){
5     if(sem->lock > 0){
6       sem->lock--;
7       EA = 1;
8       break;
9     }else{
10      sch_tasks[sch_index].state = BLOCKED;
11      sem->waiting_list[sch_index] = 1;
```

```
12        sch_next();
13        EA = 0;
14      }
15    }
16 }
```

Código 4.11 – Função bloqueante para a obtenção de um recurso do semáforo. Fonte: O autor

```
1  #pragma nooverlay
2  Byte sch_semaphore_tryget(struct sch_semaphore_sync *sem){
3      EA = 0;
4    if(sem->lock > 0){
5      sem->lock--;
6      EA = 1;
7      return 1;
8    }else{
9        EA = 1;
10     return 0;
11   }
12 }
```

Código 4.12 – Função não bloqueante para a obtenção de um recurso do semáforo. Fonte: O autor

```
1  #pragma nooverlay
2  Byte sch_semaphore_put(struct sch_semaphore_sync *sem){
3      EA = 0;
4    Byte i;
5    if(sem->lock < sem->share){
6      sem->lock++;
7      for(i = 0; i < SCH_MAX_TASKS; i++){
8        if(sem->waiting_list[i] == 1 && sch_tasks[i].state == BLOCKED){
9          sem->waiting_list[i] = 0;
10         sch_tasks[i].state = WAIT;
11       }else if(sem->waiting_list[i] == 1 && sch_tasks[i].state == FREE){
12         sem->waiting_list[i] = 0;
13       }
14     }
15     EA = 1;
16     return 1;
17   }else{
18       EA = 1;
19     return 0;
20   }
21 }
```

Código 4.13 – Função para a adição de recursos em um semáforo. Fonte: O autor

4.3.3 Instruções *test_and_set()*

Dentro do conjunto de instruções de processadores da família Intel 8051 existe a instrução atômica de *test and set* denominada JBC. Esta instrução recebe como parâmetros o endereço para onde o apontador de programa deverá apontar e uma variável do tipo binário (endereços 20 a 2F da memória interna). Caso a variável possua valor igual a 1, o algoritmo irá saltar para o endereço presente no primeiro parâmetro e limpar o valor presente na variável, deixando-o em zero, caso contrário a execução pula para a próxima instrução existente [7].

O compilador SDCC gera esta instrução sempre que um padrão como o visto no Código 4.14. Na prática, esta é a maneira mais eficiente de implementação de um algoritmo de sincronização do tipo mutex, uma vez que o uso de memória será de apenas um bit e o código assembler gerado consiste em apenas uma instrução. Outra vantagem do uso desta técnica reside no fato de que não é necessário desabilitar interrupções para a implementação do sincronismo.

```
1 volatile bit resource_is_free= 1;
2
3 if(resource_is_free){
4     resource_is_free = 0;
5     ...
6     resource_is_free = 1;
7 }
```

Código 4.14 – Implementação de mecanismos de sincronização com variáveis binárias. Fonte: O autor

Mesmo sendo de grande utilidade, o uso de variáveis binárias pelo compilador SDCC possui limitações, por exemplo, não é possível incluí-las em estrutura de dados, muito menos criar funções que recebam como parâmetro variáveis do tipo bit. Devido a isso, não é possível criar funções que manipulem este tipo de variável, nem manipular mutexes e semáforos sem desabilitar as interrupções do sistema.

Todavia, no arquivo de cabeçalho do RTOS foi incluída a denifição de variáveis do tipo *sch_mutex* que, em essência, são variáveis do tipo bit voláteis. Deste modo, o usuário pode implementar seus mecanismos de mutex e semáforos diretamente na aplicação, utilizando menos memória do que as implementações típicas vistas anteriormente.

4.4 Funções extras do RTOS

Além das funções explicadas acima, o sistema criado disponibiliza duas funções para que o usuário possa remover tarefas do sistema e fazer com que uma tarefa force a sua preempção. Nas subseções abaixo são expostas e explicadas as implementações destas

funções do sistema, bem como situações em que elas podem ser utilizadas. No final desta seção, teremos o conhecimento necessário acerca do RTOS para expandirmos a Figura 27.

4.4.1 sch_remove_task

Dado que o retorno de funções na arquitetura 8051 ocorre apenas via instrução ret, sem nenhum parâmetro, não é possível fazer com que as tarefas retornem para a função *main()*, por exemplo, ao chegarem no fim de sua execução de maneira abstraída ao usuário. Tal limitação se deve ao fato de que a arquitetura não foi projetada pensando em sistemas multi-tarefas.

Com o intuito de contornar tal limitação da arquitetura, criou-se a função especial *sch_remove_task*. Esta função deve ser chamada pelo usuário antes que a tarefa atinja o seu final e irá fazer com que o sistema marque a sua posição na fila de tarefas como estando livre para ser alocada por outras tarefas (estado *FREE*), além de escalonar a próxima tarefa disponível na fila que esteja no estado *WAIT*. Além disso, a variável de sistema *sch_num_tasks* é decrementada em uma unidade, mantendo o sistema a par do número de tarefas existentes.

O código 4.15 contém o algoritmo para a remoção de uma tarefa. Ao final da função vemos o uso da macro *POP_BANK*, uma vez que o compilador não gera instruções para recuperar os valores presentes na pilha dos principais registradores durante o retorno de funções que não sejam ISR's. Por isso, é essencial utilizar tal macro, garantindo o correto funcionamento do sistema.

```
1  void sch_remove_task(){
2    EA = 0;
3    sch_num_tasks--;
4    sch_tasks[sch_index].state = FREE;
5
6    // SCHEDULE THE NEXT TASK:
7    Byte i;
8
9    if(sch_num_tasks == 0){
10     PCON = 0x02;
11     PCON = 0x32;
12   }else{
13     i = (sch_index + 1)%SCH_MAX_TASKS;
14
15     while(i != sch_index){
16       if(sch_tasks[i].state == WAIT){
17         break;
18       }
19       i = (i + 1)%SCH_MAX_TASKS;
20     }
21
```

```
22      sch_tasks[sch_index].state = WAIT;
23      sch_tasks[i].state = READY;
24      sch_index = i;
25    }
26
27    TO_STACK
28    POP_BANK
29    EA = 1;
30  }
```

Código 4.15 – Função de remoção de uma task do sistema. Fonte: O autor

Visto que a única forma de recuperar dados da pilha diretamente para os registradores é através de instruções assembler *push* e *pop*, é necessário criar um "incherto"de código assembler dentro do código C do sistema, o que é permitido pelo compilador SDCC, bastando adicionar o código assembler entre *___asm* e *___endasm;*. A macro *POP_BANK* pode ser vista no Código 4.16.

```
1  #define POP_BANK     \
2      __asm        \
3        pop psw    \
4        pop 0    \
5        pop 1    \
6        pop 2    \
7        pop 3    \
8        pop 4    \
9        pop 5    \
10       pop 6    \
11       pop 7    \
12       pop dph    \
13       pop dpl    \
14       pop b    \
15       pop acc    \
16       pop bits   \
17     __endasm;
```

Código 4.16 – Recuperação de valores salvos na pilha. Fonte: O autor

4.4.2 sch_next

Tipicamente, uma tarefa possui períodos com grande volume de processamento, seguidos por períodos de espera por eventos como leitura de dados de sensores ou interação do usuário com a interface. Em decorrência deste comportamento, uma alternativa para tornar o sistema mais eficiente é fazer com que uma tarefa, ao entrar em um período de espera, libere a quantidade de tempo restante de seu *time slice* para que outra tarefa execute os seus processamentos.

Para tanto existe a função *sch_next* que, ao ser evocada, força a ocorrência de uma troca de contexto. Deste modo o uso do processador por parte da aplicação será maximizado, deixando-o ocioso apenas em ocasiões onde todas as tarefas estiverem em seu período de espera ao mesmo tempo. O Código 4.17 possui o algoritmo utilizado nesta função.

```
1 void sch_next(){
2   EA = 0;
3   PUSH_BANK
4   TO_XRAM
5   sch_schedule();
6   TO_STACK
7   POP_BANK
8   EA = 1;
9 }
```

Código 4.17 – Troca de contexto forçada pelo usuário. Fonte: O autor

Em seções anteriores foram expostas as macros para manipulação da pilha, sendo elas, *POP_BANK*, *TO_XRAM* e *TO_STACK*. A novidade aqui se encontra na macro *PUSH_BANK*, utilizada para salvar na pilha os valores atuais dos registradores. Sem esta macro, ao retornar para a tarefa em que se encontrava no momento em que a função foi chamada, o sistema pode produzir resultados errôneos ou, em casos mais graves, ter todo o seu funcionamento afetado. O Código 4.18 ilustra o algoritmo presente na macro.

```
1  #define PUSH_BANK   \
2      __asm     \
3        push bits \
4        push acc  \
5        push b    \
6        push dpl  \
7        push dph  \
8        push 7    \
9        push 6    \
10       push 5    \
11       push 4    \
12       push 3    \
13       push 2    \
14       push 1    \
15       push 0    \
16       push psw  \
17     __endasm;
```

Código 4.18 – Carregamento de registradores na pilha. Fonte: O autor

Finalmente, a função *sch_schedule* é a responsável pela seleção da próxima tarefa a ser rodada através do algoritmo *round robin*. Tal função é utilizada pelo sistema durante a preempção de tarefas e por ser conferida no Código 4.19.

```
#pragma nooverlay
void sch_schedule(){
  Byte i;

  if(sch_num_tasks == 0){
    PCON = 0x02;
    PCON = 0x32;
  }else{
    i = (sch_index + 1)%SCH_MAX_TASKS;

    while(i != sch_index){
      if(sch_tasks[i].state == WAIT){
        break;
      }
      i = (i + 1)%SCH_MAX_TASKS;
    }

    sch_tasks[sch_index].state = WAIT;
    sch_tasks[i].state = READY;
    sch_index = i;
  }
}
```

Código 4.19 – Seleção de tarefas pelo algoritmo *round robin*. Fonte: O autor

Cabe ao usuário utilizar esta função da maneira adequada conforme a sua aplicação. Uma boa prática é adicionar a chamada da função sempre após o término de algum processamento de dados por parte da tarefa, conforme mostrado no Código 4.20. Note que o cabeçalho da função possui um *__reentrant*, assim como vimos anteriormente no caso do uso de *pragma nooverlay*, este é um comando para que o compilador SDCC salve os parâmetros e variáveis locais da tarefa na pilha do processador, de forma a garantir o correto funcionamento do sistema [35]. É recomendado que todas as tarefas sejam declaradas deste jeito ou utilizando o *pragma nooverlay*.

```
// TAREFA EXEMPLO
void task(void *data) __reentrant{
    data;
    while(1){
        ...             // PROCESAMENTO DE DADOS PELA TAREFA.
        sch_next();     // TROCA DE CONTEXTO FORCADA.
    }
}
```

Código 4.20 – Uso da função *sch_next()*. Fonte: O autor

4.4.3 Ciclo de vida completo de uma tarefa

Uma vez vistas as funções presentes no RTOS, torna-se possível expandir o ciclo de vida de uma tarefa visto na Figura 27, adicionando as chamadas de funções responsáveis por fazerem as tarefas irem de um estado para o outro. A Figura 30 ilustra o ciclo de vida completo no RTOS criado.

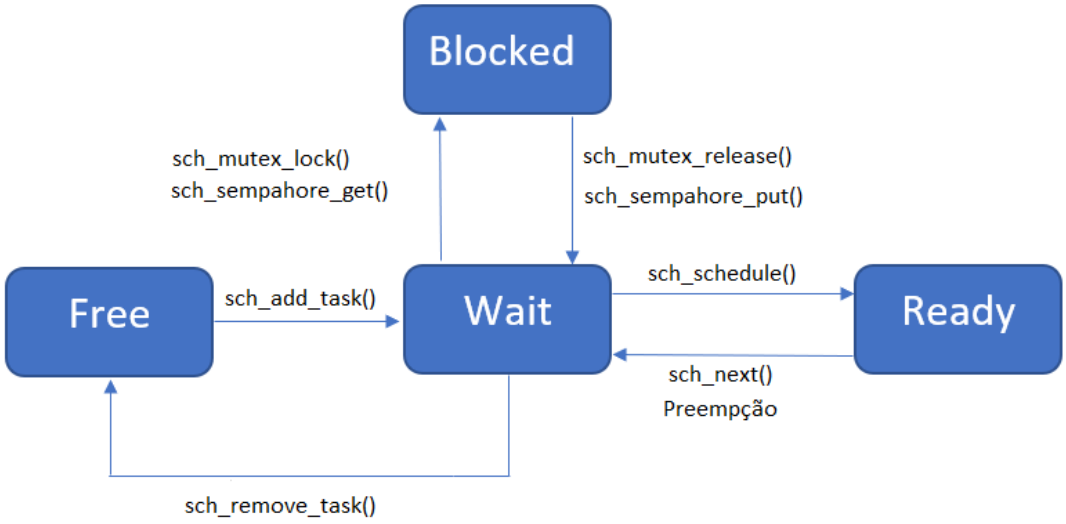

Figura 30 – ciclo de vida completo de uma tarefa no sistema. [Fonte: O autor]

5 Resultados

Este capítulo analisa os resultados que foram alcançados pelo sistema operacional obtido. Deste modo, as seções que seguem tratarão de uma análise acerca dos *footprints* de código e memória obtidos, das ferramentas que efetivamente foram implementadas no sistema e, por fim, serão demonstradas duas aplicações construídas em cima do RTOS criado.

5.1 Objetivos alcançados

Na Tabela 5 elencamos as principais características desejadas para o RTOS final. Dentre as principais funcionalidades do sistema estavam os mecanismos de mutex, semáforos e variáveis binárias vistos no Capítulo 4, além da simples desabilitação de interrupções que fica disponível ao usuário através da manipulação do registrador EA. Para o escalonador do sistema definiu-se que seria usado um escalonador *round robin* sem prioridade, o que foi exposto no algoritmo da função *sch_schedule()*, presente no Código 4.19.

Falamos também, no início deste trabalho, que era de grande interesse que o sistema criado fosse *open source*. Para tanto, todos os códigos fontes relacionados ao sistema foram disponibilizados na plataforma GitHub [63]. Deste modo, qualquer usuário que tenha interesse em utilizar o sistema ou até mesmo contribuir para o seu desenvolvimento tem total liberdade para tal. Para além destas características, também foram definidos valores desejados para os *footprints* de código e memória na Seção 4.1, sendo estes de 3 Kbytes e 200 bytes, respectivamente. Estes dados são obtidos nos arquivos .lst e .sym gerados pelo compilador SDCC [35].

O Código 5.1 mostra o conteúdo presente no arquivo .lst gerado. Este arquivo contém todo o mapeamento de memória feito pelo compilador, mostrando os endereços específicos em que cada variável foi alocada e em quais partes da memória (memória interna, externa, etc). Ainda, dela podemos extrair o *footprint* de memória do RTOS, dado que as variáveis do sistema foram todas alocadas na memória externa do processador.

```
1          449 ;--------------------------------------------------------
2          450 ; external ram data
3          451 ;--------------------------------------------------------
4          452  .area XSEG    (XDATA)
5 000000   453 _sch_tasks = 0x0000
6 000208   454 _sch_time  = 0x0208
7 00020A   455 _sch_index = 0x020a
```

```
00020B   456 _sch_num_tasks = 0x020b
```

Código 5.1 – Arquivo de mapeamento de memória gerado pelo compilador. Fonte: O autor

Vemos que os endereços ocupados pela variável começam em 0x000 e vão até 0x20B. Sendo assim, temos que o *footprint* de memória é de 524 bytes, ficando acima dos 200 bytes desejados no início do projeto. Isso ocorre devido ao número de tarefas máximo estipulado para o sistema e ao tamanho de 50 bytes adotado para o vetor utilizado durante o salvamento da pilha do processador na troca de contexto. Estes valores podem ser reduzidos, uma vez que aplicações baseadas em processadores 8051 dificilmente terão mais do que 5 tarefas e a profundidade da pilha geralmente ficará em torno de 25 bytes.

Devido às restrições quanto ao tamanho da memória e ao mapeamento de memória feito pelo compilador SDCC não foi construído um algoritmo para alocação dinâmica de memória. Sem isso, torna-se complexo criar um sistema onde o usuário possa determinar tamanhos diferentes para o vetor de salvamento da pilha em cada tarefa, portanto, o sistema limita-se a usar um tamanho fixo para este vetor.

Se adotarmos estas restrições, a estrutura de dados de cada tarefa deixará de ocupar 52 bytes na memória, passando a ocupar apenas 27 bytes. Além disso, a lista de tarefas passará a ocupar apenas 135 bytes (adotando um número máximo de 5 tarefas), no lugar dos atuais 520 bytes. Neste novo cenário o *footprint* de memória do sistema passa ser de apenas 139 bytes, o que atinge os requerimentos iniciais para o sistema. O Código 5.2 mostra como fica o novo mapeamento de memória ao levar-se em consideração este novo cenário.

Levando em conta o tamanho do TCB adotado nesta nova configuração, ainda existe margem para a adição de uma nova tarefa sem comprometer os requisitos iniciais do sistema. Contudo, optou-se por deixar um espaço de memória maior livre para implementação de recursos futuros no sistema. Além disso, 5 tarefas foi considerado um número suficiente para um processador 8051, visto que as aplicações baseadas neste tipo de *chip* são compactas.

```
         449 ;--------------------------------------------------------
         450 ; external ram data
         451 ;--------------------------------------------------------
         452  .area XSEG    (XDATA)
000000   453 _sch_tasks = 0x0000
000087   454 _sch_time  = 0x0087
000089   455 _sch_index = 0x0089
00008A   456 _sch_num_tasks = 0x008a
```

Código 5.2 – Arquivo de mapeamento de memória atualizado. Fonte: O autor

Finalmente, o arquivo .sym gerado possui como último campo as informações acerca da quantidade de memória ocupada em cada segmento de memória do processador. Para

obtermos o *footprint* de código do RTOS basta olharmos o tamanho da área CSEG presente no arquivo. Para o sistema criado, obteve-se um tamanho de 2412 bytes, ficando abaixo do valor estipulado inicialmente. Portanto, a Tabela 6 mostra as especificações finais alcançadas pelo sistema.

Tabela 6 – Características finais do sistema

	***Footprint* de memória**	***Footprint* de código**	**Escalonador**	**Sincronização**
Atingido	139 bytes (5 tarefas)	2412 bytes	*Round robin* sem prioridade	Semáforos, Mutexes, variáveis binárias, desabilitação de interrupções
Desejado	200 bytes	3 Kbytes	*Round robin* sem prioridade	Semáforos, Mutexes, variáveis binárias, desabilitação de interrupções

Fonte – O autor

5.2 Aplicações baseadas no sistema

Os testes feitos em cima do RTOS ocorreram em uma eletrônica dedicada baseada no processador ADuC847 da *Analog Devices*. Para exibição de dados utilizou-se um *display* LCD e a interação do usuário com o dispositivo dá-se por meio de 8 botões que podem ser programados para diversas funcionalidades. Dado que o projeto da placa foi pensado para a leitura de células de carga, a escolha do processador foi determinada pelo seu conversor AD de 24 bits.

Ao todo, três aplicações foram criadas para realização de testes no sistema. A primeira aplicação foi criada para testar se o funcionamento dos mutexes está ocorrendo da maneira desejada, a segunda é um exemplo de programa produtor/consumidor e a terceira aplicação trata-se de um *firmware* para controladores de peso. As duas primeiras aplicações serviram de base para verificar se os mecanismos de sincronização foram implementados corretamente, solucionando problemas comuns gerados pela concorrência. Abaixo vamos descrever com mais detalhes como estas aplicações são construídas e analisar seus resultados.

5.2.1 Primeira aplicação

Quando queremos testar o funcionamento de mutexes é comum criarmos um pequeno programa de testes que consiste em sincronizar um número de tarefas de modo a fazer com que a execução destas sempre ocorra conforme uma ordem desejada. No programa

criado foram utilizadas três tarefas, a ideia foi fazer com que a execução delas seguisse a ordem tarefa 1, tarefa 2 e tarefa 3. Estas tarefas não realizam nenhum processamento pesado como visto em operações matemáticas, elas apenas escrevem no *display* que estão sendo executadas.

Para que ocorra o sincronismo precisamos de três mutexes, cada mutex corresponde a uma das tarefas. Começaremos com dois mutexes bloqueados e um mutex desbloqueado, de modo que uma tarefa comece a sua execução. A forma como os mutexes são manipulados é a seguinte:

- Mutex 1 começa desbloqueado e os mutexes 2 e 3 começam bloqueados;
- Ao entrar na tarefa 1 ela irá verificar se o mutex 1 está desbloqueado. Em caso afirmativo ela irá bloqueá-lo e executar sua região critica, escrevendo "Task 1"no *display* e, após isso, irá destravar o mutex 2;
- Como o mutex 2 está desbloqueado, a tarefa 2 consegue agora entrar em sua região crítica, bloqueando o mutex 2, escrevendo "Task 2"no *display* e então desbloqueando o mutex 3;
- A tarefa 3, por sua vez, irá poder agora entrar em sua região critica, bloqueando o mutex 3, escrevendo "Task 3"no *display* e então desbloqueando o mutex 1;
- Este ciclo se repete indeterminadamente.

Como cada tarefa apenas consegue executar sua região crítica se o seu mutex estiver desbloqueado, cria-se um sincronismo no sistema, de modo que a ordem de execução das tarefas será sempre tarefa 1, tarefa 2 e tarefa 3. A Figura 31 demonstra como o algoritmo funciona de maneira mais intuitiva.

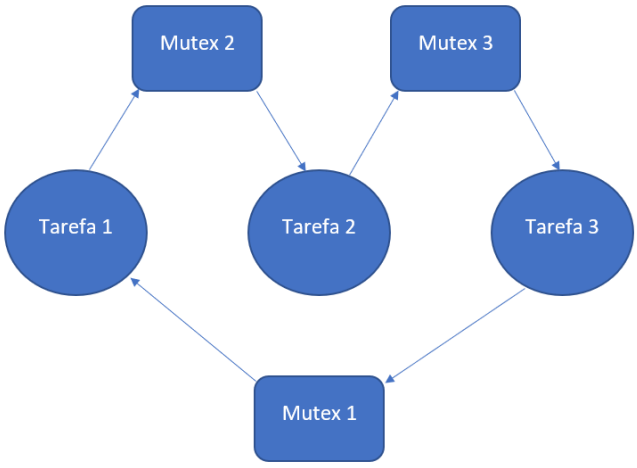

Figura 31 – Sincronização entre tarefas com o uso de mutexes. [Fonte: O autor]

Na figura vemos que a tarefa 1 estará desbloqueando o mutex 2, porém esta não funcionará enquanto o mutex 1 não for liberado para ela, o mesmo ocorre com as outras duas tarefas. Esta construção é comumente chamada de grafo de dependências [2].

Ao executar este programa no *hardware* utilizado, o resultado obtido esteve conforme o esperado. De fato ocorreu sincronismo na execução das tarefas, o que demonstra que o mecanismo de mutex foi corretamente implementado no RTOS. Os Códigos 5.3, 5.4 e 5.5 mostram como cada tarefa foi programada.

```
1 void Task1(void *data) __reentrant{
2     data;
3     while(1){
4         sch_mutex_lock(&mutex1);
5             escreve_display(LINHA1, "Task 1");
6         sch_mutex_release(&mutex2);
7     }
8 }
```

Código 5.3 – Tarefa 1. Fonte: O autor

```
1 void Task2(void *data) __reentrant{
2     data;
3     while(1){
4         sch_mutex_lock(&mutex2);
5             escreve_display(LINHA1, "Task 2");
6         sch_mutex_release(&mutex3);
7     }
8 }
```

Código 5.4 – Tarefa 2. Fonte: O autor

```
1 void Task3(void *data) __reentrant{
2     data;
3     while(1){
4         sch_mutex_lock(&mutex3);
5             escreve_display(LINHA1, "Task 3");
6         sch_mutex_release(&mutex1);
7     }
8 }
```

Código 5.5 – Tarefa 3. Fonte: O autor

5.2.2 Produtor/consumidor

Durante o estudo de sistemas multi-tarefas, é comum nos depararmos com exemplos de problemas do tipo produtor/consumidor. Nestes problemas uma ou mais tarefas ficam encarregadas de gerar recursos ao passo que uma ou mais tarefas irão consumir estes recursos conforme eles são disponibilizados. Na prática, este tipo de situação é comum

em sistemas concorrentes, por exemplo, no caso onde um grupo de tarefas obtém dados via rede e disponibiliza estes dados para outras.

A aplicação criada consiste em uma tarefa produtora e uma consumidora. Quando em execução, a tarefa produtora irá adicionar recursos em um semáforo de maneira periódica, conforme visto no Código 5.6. Para a visualização, a cada vez que a tarefa produtora adicionar um recurso no semáforo ela escreve 1 em uma posição do *display*, essa posição é incrementada para que na próxima execução o valor 1 seja escrito em uma nova posição.

```
1  void producer(void *data) __reentrant{
2      data;
3      long i = 0;
4      Byte pos = 0;
5      while(1){
6          sch_mutex_lock(&mutex);
7              escreve_display(LINHA1^(5 + pos), "1");
8              posiciona_cursor(LINHA1^-1);
9              pos = (pos + 1)%BUFFER_SIZE;
10             sch_semaphore_put(&sem);
11         sch_mutex_release(&mutex);
12         for(i = 0; i < 5000; i++);
13         sch_mutex_release(&consumer_lock);
14     }
15 }
```

Código 5.6 – Tarefa produtora. Fonte: O autor

O consumidor irá ficar a espera de recursos no semáforo. Quando existirem recursos disponíveis, ele irá consumi-los e escrever o valor 0 na tela, incrementando a posição assim como feito pelo produtor. O Código 5.7 possui o algoritmo presente na tarefa consumidora.

```
1  void consumer(void *data) __reentrant{
2      data;
3      long i = 0;
4      Byte pos = 0;
5      while(1){
6          sch_mutex_lock(&consumer_lock);
7          sch_mutex_lock(&mutex);
8             if(sch_semaphore_tryget(&sem)){
9                  escreve_display(LINHA1^(5 + pos), "0");
10                 posiciona_cursor(LINHA1^-1);
11                 pos = (pos + 1)%BUFFER_SIZE;
12              }
13         sch_mutex_release(&mutex);
14     }
15 }
```

Código 5.7 – Tarefa consumidora. Fonte: O autor

Ambas as tarefas (produtor e consumidor) ficam em repouso por um tempo não exato, de modo a facilitar a visualização do funcionamento do algoritmo para o usuário. O tempo de espera não é exato pelo motivo de o *delay* ser feito por *software*, no laço *for* encontrado no final das tarefas. O uso de *delays* via *software* não é recomendável para casos onde a precisão no tempo é necessária, porém em situações como a da aplicação em questão é aceitável que o tempo de espera não seja preciso. Finalmente, as Figuras 32 e 33 mostram o código em execução em dois momentos de tempo diferentes.

Figura 32 – Programa após o consumidor ter consumido recursos. [Fonte: O autor]

Figura 33 – Programa após o produtor ter produzido novos recursos. [Fonte: O autor]

5.2.3 Firmware para controladores de peso

Controladores de peso são equipamentos de automação industrial amplamente utilizados em linhas de produção onde existe necessidade de pesagem ou contagem de produtos. Neste tipo de equipamentos, o usuário define os principais parâmetros da produção como o peso desejado, o peso máximo que o produto pode ter e o peso mínimo. Com base nisso o equipamento classifica os produtos como estando dentro do peso adequado ou não, fazendo também a retroalimentação da planta, de modo a corrigir o processo de dosagem para que todos os produtos fiquem dentro do peso esperado.

A leitura do peso dá-se por meio de células de carga e é a parte mais delicada no que tange o desenvolvimento desse tipo de equipamento, dado que é necessária uma grande resolução para que a leitura seja precisa, além de uso de filtros analógicos e digitais. Graças a essa natureza, o uso de eletrônicas dedicadas em controladores de peso torna-se indispensável.

O maior desafio em construir um controlador de peso com eletrônica dedicada reside no fato de que, ao mesmo tempo em que o sistema precisa gerir a interação do usuário com a interface, também é necessário garantir que as leituras da célula de carga estão sendo tratadas da maneira correta. No caso em que nenhum RTOS é utilizado, existe um grande risco de que leituras vindas do conversor AD não sejam atendidas num prazo válido, fazendo com que o equipamento não funcione corretamente.

Para tanto, o *firmware* construído utilizou o RTOS criado com duas tarefas, uma para leitura e tratamento de dados vindos do conversor AD e outra para gerência da interface do sistema. Antes de entrarmos em mais detalhes acerca da implementação do projeto, é importante ressaltar que, em razão do desenvolvimento deste algoritmo ter sido feito durante o estágio obrigatório da faculdade, não foi autorizado por parte da empresa dar grandes detalhes acerca dos algoritmos utilizados de filtros digitais na leitura do conversor AD, nem demais detalhes acerca de como é realizada a rejeição de pacotes. Assim sendo, podemos resumir a tarefa de leitura de peso nos seguintes passos:

- Verifica se a leitura atual é maior do que 25% do peso esperado;
- Caso a leitura for maior, a tarefa começa a adicionar as próximas leituras em uma fila durante um intervalo de tempo necessário para a estabilização do peso em cima da célula de carga. Passado este tempo, a tarefa faz uma média com as últimas 40 leituras presentes na fila;
- Caso a leitura for menor, o equipamento espera um intervalo de tempo configurado pelo usuário para então iniciar o procedimento de tara do sistema. Neste procedimento o equipamento considera que o peso lido é, na verdade, o valor sem peso algum na célula de carga;

- Ainda, caso a leitura for maior que 25% do peso esperado, o equipamento irá classificar o pacote como bom ou ruim e atualizar variáveis do sistema como produção total, número de pacotes rejeitados, média de peso dos últimos pacotes, etc.

Na Figura 34 temos um fluxograma referente a tarefa de leitura de peso.

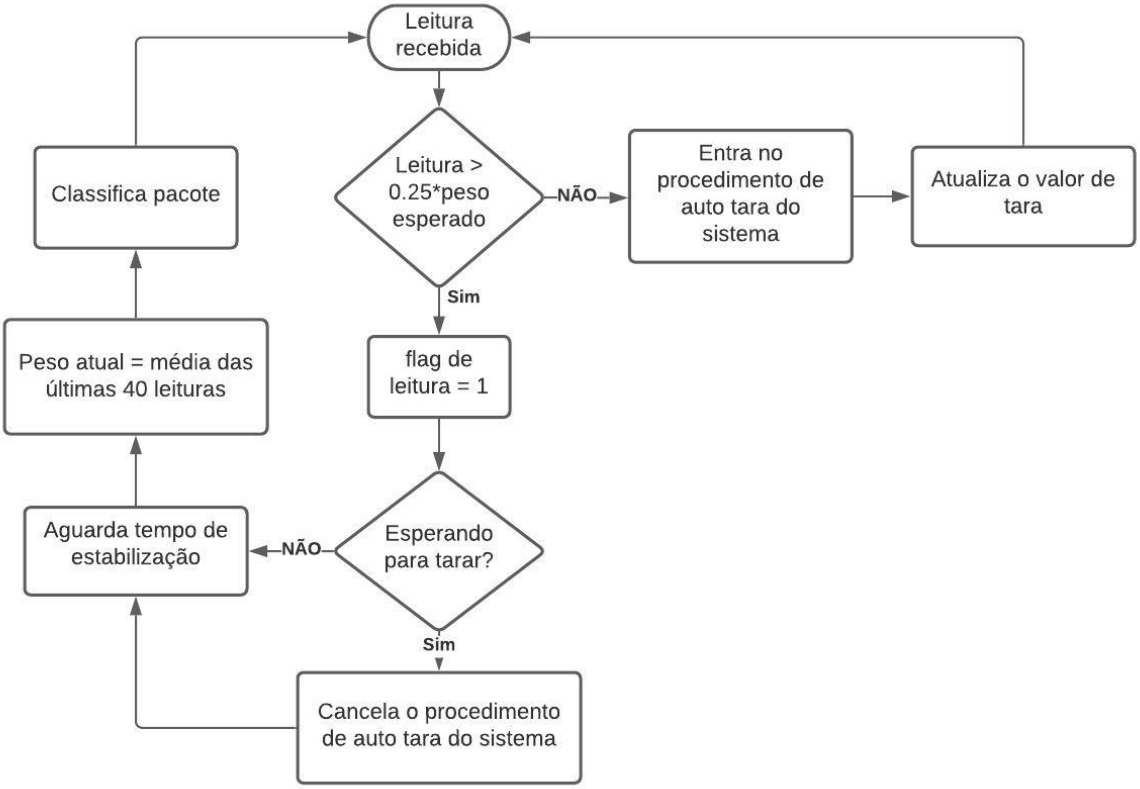

Figura 34 – Fluxograma da tarefa responsável pela leitura de peso. [Fonte: O autor]

Quanto a tarefa responsável pela gerência da interface, temos que esta irá verificar se o usuário está interagindo com algum dos botões do equipamento e atualizar os dados que são mostrados na tela conforme esta interação. Para além desta função, a tarefa ainda deve atualizar os dados a serem mostrados conforme novos pacotes são pesados.

Dado que as duas tarefas do sistema compartilham variáveis como o peso do último pacote, quantidade de pacotes rejeitados, número total de pacotes pesados, dentre outras, precisa-se adotar o uso de um mutex para sincronização desses dados. Antes de uma das duas tarefas manipular estas variáveis, seja para alterar seus valores ou para exibi-las na tela, o mutex é bloqueado, sendo desbloqueado após o uso das mesmas.

Também, uma vez que existe um espaço de tempo entre uma leitura do conversor AD e outra, utiliza-se a técnica que liberar o espaço de tempo de uma tarefa para a outra via função *sch_next()*. Deste modo o processador é mantido em uso na maior parte do tempo em que o programa está rodando.

Por fim, os recursos utilizados do sistema são duas tarefas e um mutex. Com este pequeno número de recursos torna-se possível criar uma aplicação complexa em um processador com apenas 2 Kbytes de memória externa disponível. O *footprint* de memória do sistema nesta aplicação é de 139 bytes (conforme visto anteriormente) mais 11 bytes para a estrutura de dados do mutex, totalizando em um *footprint* compacto de apenas 150 bytes na memória externa. Na Figura 35 temos o arquivo .mem gerado pelo compilador, este arquivo nos mostra o mapeamento de memória feito pelo mesmo na memória interna do processador, note que a pilha do sistema começa no endereço 0x63, possuindo disponíveis 157 bytes, o que é mais do que o suficiente para a aplicação.

```
Internal RAM layout:
     0 1 2 3 4 5 6 7 8 9 A B C D E F
0x00:|0|0|0|0|0|0|0|0|b|b|b|b|c|c|c|c|
0x10:|d|d|d|d|d|Q|Q|Q|Q|Q|Q|Q|Q|Q|Q|Q|
0x20:|B|T|a|a|a|a|a|a|a|a|a|a|a|a|a|a|
0x30:|a|a|a|a|a|a|a|a|a|a|a|a|a|a|a|a|
0x40:|a|a|a|a|a|a|a|a|a|a|a|a|a|a|a|a|
0x50:|a|a|a|a|a|a|a|a|a|a|a|a|a|a|a|a|
0x60:|a|a|a|S|S|S|S|S|S|S|S|S|S|S|S|S|
0x70:|S|S|S|S|S|S|S|S|S|S|S|S|S|S|S|S|
0x80:|S|S|S|S|S|S|S|S|S|S|S|S|S|S|S|S|
0x90:|S|S|S|S|S|S|S|S|S|S|S|S|S|S|S|S|
0xa0:|S|S|S|S|S|S|S|S|S|S|S|S|S|S|S|S|
0xb0:|S|S|S|S|S|S|S|S|S|S|S|S|S|S|S|S|
0xc0:|S|S|S|S|S|S|S|S|S|S|S|S|S|S|S|S|
0xd0:|S|S|S|S|S|S|S|S|S|S|S|S|S|S|S|S|
0xe0:|S|S|S|S|S|S|S|S|S|S|S|S|S|S|S|S|
0xf0:|S|S|S|S|S|S|S|S|S|S|S|S|S|S|S|S|
0-3:Reg Banks, T:Bit regs, a-z:Data, B:Bits, Q:Overlay, I:iData, S:Stack, A:Absolute
```

Figura 35 – Mapeamento da memória interna do processador. [Fonte: O autor]

6 Considerações finais

Este trabalho propôs-se a explicar a situação atual da família de processadores Intel 8051 e a sua participação dentro do campo de sistemas embarcados. Para tal, foram levantados dados acerca do mercado de sistemas embarcados que serviram como base para compreensão de como são feitos projetos dentro dele e quais as características mais comuns nestes projetos. Também, vimos qual é a situação atual dos processadores baseados na família 8051, de modo a entendermos o porquê de ainda ser pertinente realizar estudos em cima desta plataforma.

Segundo os dados levantados, ficou evidente a demanda do mercado de sistemas embarcados por RTOS's. Estes, além de facilitarem o desenvolvimento do sistema, fornecem uma ampla gama de benefícios para o mesmo como capacidade de atender aos requisitos temporais da aplicação e aumentar a eficiência no uso do processador. Diante disso, explicamos a fundo como um RTOS funciona e quais os seus principais componentes, trazendo exemplos de RTOS's atualmente populares no mercado e como tais componentes são implementados por eles.

A maior motivação para o desenvolvimento deste trabalho residiu no fato de a família 8051 não possuir suporte de grandes sistemas operacionais de tempo real e, apenas possuir suporte daqueles baseados em ferramentas pagas e de alto valor monetário. Por isso, a proposta do mesmo foi a de criar um RTOS totalmente gratuito e *open source* baseado em ferramentas também gratuitas e, de preferência, que também fossem *open source*.

O compilador escolhido para converter os códigos fonte em linguagem de máquina foi o SDCC, por ser um dos compiladores mais populares para a família, oferecendo um grande número de recursos e gerando códigos assembler satisfatoriamente otimizados. Junto a isso, este foi o principal compilador *open source* encontrado para a plataforma, disponibilizando ainda os códigos fontes de todas as suas bibliotecas. Este compilador possui grande compatibilidade com o padrão C ANSI, o que facilita o seu uso por programadores com pouca experiência em programação C para a família 8051.

Além do compilador, o editor de código escolhido foi o Microsoft Visual Studio Code. Este editor é totalmente gratuito e recebe grande suporte da comunidade de desenvolvedores no mundo inteiro, incluido da comunidade *open source*.

Quanto aos objetivos alcançados, o RTOS obtido foi capaz de fornecer todas as características desejadas ao mesmo tempo em que produziu pequenos *footprints* de código e memória, o que é essencial em sistemas com recursos limitados como nos processadores baseados na arquitetura 8051. Os mecanismos de sincronização entre tarefas implementados se comportaram da maneira desejada, de modo que produziram os resultados esperados em aplicações como a produtor/consumidor estudada no Capítulo 5.

Outros objetivos mais gerais acerca da construção do RTOS também foram cumpridos, dentre eles estão o uso da linguagem C como principal linguagem nos códigos fontes do projeto e disponibilização dos fontes para a comunidade *open source*. Vale ressaltar que, apesar de ter sido projetado de modo a utilizar o menor número de instruções assembler possível, o sistema possui macros em assembler para manipulação da pilha do processador, uma vez que não fora encontrada outra alternativa para tal. Quanto a disponibilidade do código fonte, o mesmo foi publicado na plataforma GitHub, onde é atualizado constantemente conforme novos avanços são feitos em seu desenvolvimento [64].

6.1 Trabalhos futuros

Ainda que um RTOS contendo as funcionalidades implementadas neste trabalho consiga atender a grande maioria das aplicações, especialmente em sistemas com recursos limitados como na família 8051, existe uma série de outros recursos populares em outros RTOS, como o FreeRTOS, que podem ser implementadas futuramente no sistema criado. Tais funcionalidades são:

- **Área de memória compartilhada:** Este recurso é comum em sistemas operacionais e é um forma simples de prover comunicação entre tarefas. Em essência, a sua implementação se baseia em reservar uma área de memória aonde tarefas irão colocar e ler dados. A manipulação desta área é feita pelo sistema operacional, de modo que as tarefas insiram e leiam dados delas via chamadas de funções do sistema. É essencial também que o sistema sincronize a escrita e leitra desta área por diferentes tarefas;

- **Filas, pilhas e listas:** Mesmo em processadores com pouca memória disponível, estas estruturas de dados são comumente utilizadas. Pelo fato de sua implementação ser relativamente complexa, os principais sistemas operacionais de tempo real fornecem API's com funções para sua criação e manipulação. Tipicamente, as manipulações de listas, filas e pilhas em RTOS's é *thread safe*, ou seja, o sistema implementa mecanismos de sincronização para que várias tarefas utilizem as estruturas sem que ocorram problemas devido à concorrência;

- **Opção de escalonamento cooperativo:** Apesar de não possuir todas as vantagens de um sistema multi-tarefas não cooperativo, um escalonador cooperativo é útil para aplicações que necessitem retirar o máximo de sistemas limitados. Uma vez que neste tipo de escalonador não ocorre a preempção, o uso de mecanismos de sincronização não é necessário, assim como listas para monitoramento das tarefas do sistema. Deste modo, escalonadores cooperativos geram *footprints* de memória e código menores do que os não cooperativos;

- **Implementação de funções de *delays*:** Atualmente, caso o usuário queira gerar *delays* ele precisará fazê-lo manualmente através do uso de uma interrupção de *timer* ou via software (*delay* não preciso). Existem dois problemas nessa técnica, o primeiro é que este tipo de *delay* gera um *busy wait*, ou seja, a tarefa irá ficar parada esperando a passagem de tempo, além disto, mais recursos do processador serão utilizados, muitas vezes sem necessidade. A melhor forma de implementar estes *delays* é utilizando a interrupção de *timer* já utilizada pelo sistema e colocando a tarefa em espera, sem escaloná-la enquanto não tiver passado o tempo de espera.

Referências Bibliográficas

1 PIMENTA, T. T. *Circuitos Digitais - Análise e Síntese Lógica: Análise e Síntese Lógica - Aplicações em FPGA*. [S.l.]: GEN LTC, 2016. ISBN 8535265775. 13, 38

2 SILBERSCHATZ PETER BAER GALVIN, G. G. A. *Fundamentos de sistemas operacionais*. [S.l.]: Editora LTC, 2015. ISBN 8521629397. 13, 20, 21, 23, 24, 26, 27, 29, 31, 34, 36, 44, 82

3 WHAT IS LINUX. <https://www.linux.com/what-is-linux/>. Acessado em 12 Out. 2019. 13

4 EETIMES, E. *2019 Embedded Markets Study*. Cambridge, MA, USA, 2019. 13

5 FREERTOS. [S.l.]: FreeRTOS org. <https://www.freertos.org>. Acessado em 01 jan. 2021. 13, 29, 43

6 KEIL RTX. <https://www.keil.com/arm/rl-arm/kernel.asp>. Acessado em 01 jan. 2021. 13

7 DATASHEET ADuC847. [S.l.]. Acessado em 04 Jan. 2021. 14, 38, 41, 42, 72

8 COMPILADOR SDCC. <http://sdcc.sourceforge.net>. Acessado em 04 Jan. 2021. 14, 17, 18, 39

9 PROTOCOLO modbus. [S.l.]: Modbus organization. <https://modbus.org>. Acessado em 10 jan. 2021. 16

10 MODBUS-IDA. *MODBUS APPLICATION PROTOCOL SPECIFICATION*. 2006. Documento com as especificações tecnicas de implementação do protocolo MODBUS. 16

11 MONK, S. *Programação com arduino*. [S.l.]: Bookman, 2013. ISBN 978-85-8260-026-9. 16

12 KEIL uvision. [S.l.]: Arm Limited. <https://www2.keil.com/mdk5/uvision/>. Acessado em 10 fev. 2021. 17, 51

13 SUCHEUSKI, M. *Linguagem C*. [S.l.]: Editora Lísias, 1996. 17, 18, 20

14 MIDE-51. <https://pt.freedownloadmanager.org/Windows-PC/MIDE-51-GRATUITO.html>. Acessado em 10 fev. 2021. 17

15 IRVINE, K. R. *Assembly Language for Intel-Based Computers*. [S.l.]: Pearson, 2002. ISBN 0-13-091013-9. 17

16 TIOBE INDEX. [S.l.]: TIOBE software BV. <https://www.tiobe.com/tiobe-index/>. Acessado em 10 jan. 2021. 17

17 PONT, M. J. *Patterns for Time-Triggered Embbeded Systems*. [S.l.]: ACM Press Books, 2001. ISBN 0-201-33138-1. 17, 19, 20, 31, 37, 38, 39, 54, 55

18 ZILLER, R. M. *Microprocessadores*. [S.l.]: UFSC, 2000. ISBN 85-901037-2-2. 17

19 LOUDON, K. *Mastering Algorithms with C: Useful Techniques from Sorting to Encryption*. [S.l.]: O'Reilly Media, 1999. ISBN 1565924533. 19

20 ARDUINO. [S.l.]: Arduino. <https://www.arduino.cc>. Acessado em 10 jan. 2021. 19

21 MáQUINA de Estados em C. [S.l.]: Embedded Labworks. <https://sergioprado.org/maquina-de-estados-em-c/>. Acessado em 10 jan. 2021. 20

22 ROCH, T. W. B. Monolithic kernel vs. microkernel. 2004. 20, 21, 22, 23

23 MAPA do kernel linux. [S.l.]: Fossbytes Media PVT. <https://fossbytes.com/learn-it-faster-the-complete-linux-kernel-in-a-single-map/>. Acessado em 23 jan. 2021. 21, 25

24 STANKOV, G. S. I. Discussion of microkernel and monolithic kernel approaches. *International Scientific Conference Computer Science 2006*, 2006. 21, 23, 29

25 REVOLUTION OS. [S.l.]: Seventh Arth Releasing, 2001. 23, 24, 25

26 TORVALDS, L. *Linux: a Portable Operating System*. Dissertação (Mestrado) — university of helsinki, helsinki, finland, 1997. 25

27 GNU project. [S.l.]: Free Software Foundation. <https://www.gnu.org/home.pt-br.html>. Acessado em 09 jan. 2021. 25

28 GETTING started with POSIX. [S.l.]: Tutorial Pedia. <https://riptutorial.com/posix>. Acessado em 24 jan. 2021. 25

29 SITE intel. [S.l.]: Intel. <https://www.intel.com.br/content/www/br/pt/products/processors/core.html>. Acessado em 23 jan. 2021. 25

30 FARINES JONI DA SILVA FRAGA, R. S. d. O. J.-M. *Sistemas de tempo real*. [S.l.]: Universidade Federal de Santa Catarina, 2000. 27, 28, 29, 33

31 FRANKLIN J. DAVID POWELL, A. E.-N. G. F. *Sistemas de controle para engenharia*. [S.l.]: Bookman, 2013. ISBN 8582600674. 27

32 RAMAMRITHAM J.A. STANKOVIC, P. S. K. Efficiente scheduling algorithms for real-time multiprocesso systems. *COINS Technical Report*, 1989. 31, 33

33 MOHAMMADI, S. G. A. A. Scheduling algorithms for real-time systems. 2005. 32

34 AMAZON.COM. The freertos™ reference manual. 33, 43, 47

35 DUTTA, S. Manual sdcc. 2016. 35, 40, 55, 64, 67, 76, 78

36 HOHMUTH, H. H. M. Pragmatic nonblocking synchronization for real-time systems. *Proceedings of the 2001 USENIX Annual Technical Conference (USENIX '01)*, 2001. 35

37 DEAN, A. G. *Embedded Systems Fundamentals with Arm Cortex-M based Microcontrollers: A Practical Approach*. [S.l.]: ARM education media, 2017. ISBN 978-1-911531-03-6. 37

38 BOYS, R. What is the 8051 doing in the year 2008? 2008. 37

39 SUPPORTED Devices. [S.l.]: ARM KEIL. <https://www.keil.com/c51/chips.asp>. Acessado em 08 fev. 2021. 38

40 DATASHEET AT89S52. [S.l.]. Acessado em 09 fev. 2021. 39

41 DATASHEET DS89C420. [S.l.]. Acessado em 09 fev. 2021. 39

42 COMPILADOR C51, guia de usuário. <https://www.keil.com/c51/c51.asp>. Acessado em 09 fev. 2021. 39, 51

43 GALAXY Fit2. [S.l.]: Samsung. <https://www.samsung.com/br/watches/galaxy-fit/galaxy-fit2-black-sm-r220nzkazto/>. Acessado em 14 fev. 2021. 43

44 SUPPORTED Devices. [S.l.]: FreeRTOS org. <https://freertos.org/RTOS_ports.html>. Acessado em 14 fev. 2021. 43

45 CYGNAL (Silicon Labs) 8051 Port. [S.l.]: FreeRTOS org. <https://www.freertos.org/portcygn.html>. Acessado em 14 fev. 2021. 43

46 QUAL o footprint de memória do FreeRTOS? [S.l.]: Sérgio Prado. <https://sergioprado.org/qual-o-footprint-de-memoria-do-freertos/>. Acessado em 21 fev. 2021. 44, 55

47 MELOT, N. Study of an operating system: Freertos. 44, 45

48 BARRY, R. *Mastering the FreeRTOS™ Real Time Kernel*. [S.l.]: Real Time Engineers Ltd., 2016. 45, 46

49 KR-51. [S.l.]: MicroController Pros LLC. <https://microcontrollershop.com/raisonance_rtos.php>. Acessado em 18 fev. 2021. 50

50 ABASSI Free RTOS. [S.l.]: Code Time Technologies Inc. <https://code-time.com/rtos_free.html>. Acessado em 18 fev. 2021. 51

51 INC, C. T. T. Abassi rtos. 2011. 51

52 PK51 Professional Developer's Kit. <https://www.keil.com/c51/pk51kit.asp>. Acessado em 19 fev. 2021. 51

53 KEIL RTX Tiny. <https://www.keil.com/rtx51tiny/>. Acessado em 18 fev. 2021. 52

54 SPECIFICATIONS of the Keil RTX Real-Time Kernel. <https://www.keil.com/rtx51/specs.asp>. Acessado em 19 fev. 2021. 52

55 KURSANCEW, V. 805xrtos. 2011. 52

56 805X RTOS - a simple free RTOS (scheduler) for the 8051 microcontroller. <https://sites.google.com/site/viniciusalexandre/arquivos/805x-rtos-8051>. Acessado em 19 fev. 2021. 52

57 LIST of open source real-time operating systems. <https://www.osrtos.com>. Acessado em 19 fev. 2021. 53

58 NUTTX. <https://www.osrtos.com/rtos/nuttx/>. Acessado em 19 fev. 2021. 53

59 QUARKTS. <https://www.osrtos.com/rtos/quark-ts/>. Acessado em 19 fev. 2021. 53

60 CONTIKIOS. <https://www.osrtos.com/rtos/contiki-os/>. Acessado em 19 fev. 2021. 53

61 PROTOTHREADS. <https://www.osrtos.com/rtos/protothreads/>. Acessado em 19 fev. 2021. 54

62 8051 Instruction Set Manual. [S.l.]: Arm keil. <https://www.keil.com/support/man/docs/is51/is51_ret.htm#:~:text=The%20RET%20instruction%20pops%20the,are%20affected%20by%20this%20instruction.> Acessado em 14 mar. 2021. 58

63 GITHUB. [S.l.]: GitHub, inc. <https://github.com>. Acessado em 24 mar. 2021. 78

64 REPOSITóRIO oficial do sistema. [S.l.]: Henrique Marchi Lange. <https://github.com/henriquemlange/SCH8051>. Acessado em 11 abr. 2021. 89