UNIVERSIDADE FEDERAL DE SANTA CATARINA
CENTRO TECNOLÓGICO, DE CIÊNCIAS EXATAS E EDUCAÇÃO
DEPARTAMENTO DE ENG. DE CONTROLE, AUTOMAÇÃO E COMPUTAÇÃO
CURSO DE GRADUAÇÃO EM ENGENHARIA DE CONTROLE E AUTOMAÇÃO

Emanuel Lucas Rodrigues Moura

RTOS para processadores RISC-V com separação espacial entre tarefas

Blumenau
2023

Emanuel Lucas Rodrigues Moura

RTOS para processadores RISC-V com separação espacial entre tarefas

Trabalho de Conclusão de Curso de Graduação em Engenharia de Controle e Automação do Centro Tecnológico, de Ciências Exatas e Educação da Universidade Federal de Santa Catarina como requisito para a obtenção do título de Engenheiro de Controle e Automação.
Orientador: Prof. Carlos Roberto Moratelli, Dr.

Blumenau
2023

Ficha de identificação da obra elaborada pelo autor,
através do Programa de Geração Automática da Biblioteca Universitária da UFSC.

Moura, Emanuel Lucas Rodrigues
RTOS para Processadores RISC-V com separação espacial entre tarefas / Emanuel Lucas Rodrigues Moura ; orientador, Carlos Roberto Moratelli, 2023.
61 p.

Trabalho de Conclusão de Curso (graduação) - Universidade Federal de Santa Catarina, Campus Blumenau, Graduação em Engenharia de Controle e Automação, Blumenau, 2023.

Inclui referências.

1. Engenharia de Controle e Automação. 2. RISC-V PMP. 3. RTOS. 4. Segurança. I. Moratelli, Carlos Roberto. II. Universidade Federal de Santa Catarina. Graduação em Engenharia de Controle e Automação. III. Título.

Emanuel Lucas Rodrigues Moura

RTOS para processadores RISC-V com separação espacial entre tarefas

Este Trabalho de Conclusão de Curso foi julgado adequado para obtenção do Título de "Engenheiro de Controle e Automação" e aprovado em sua forma final pelo Curso de Graduação em Engenharia de Controle e Automação.

Blumenau, 19 de Junho de 2023.

Banca Examinadora:

Documento assinado digitalmente
Carlos Roberto Moratelli
Data: 04/07/2023 10:50:46-0300
CPF: ***.999.369-**
Verifique as assinaturas em https://v.ufsc.br

Prof. Primeiro, Dr. Carlos Roberto Moratelli
Universidade Federal de Santa Catarina

Documento assinado digitalmente
Fabio Rafael Segundo
Data: 07/07/2023 00:56:44-0300
CPF: ***.593.909-**
Verifique as assinaturas em https://v.ufsc.br

Prof. Segundo, Dr. Fábio Rafael Segundo
Universidade Federal de Santa Catarina

Documento assinado digitalmente
Guilherme Brasil Pintarelli
Data: 04/07/2023 12:42:54-0300
CPF: ***.523.129-**
Verifique as assinaturas em https://v.ufsc.br

Prof. Terceiro, Dr. Guilherme Brasil Pintarelli
Universidade Federal de Santa Catarina

AGRADECIMENTOS

Primeiramente, gostaria de agradecer a Deus, cuja graça se renova todos os dias sobre minha vida, permitindo-me superar cada desafio ao longo dessa jornada. Sei que sou totalmente dependente Dele e Ele é a razão pela qual dei o meu melhor em cada momento do curso.

Agradeço aos meus pais, Fábio e Vanda, por tudo que fizeram e fazem por mim. Só Deus sabe o quanto eles abriram mão para me proporcionar o melhor, mesmo diante das dificuldades da vida. Agradeço também por todos os ensinamentos que carregarei para sempre, os quais valem mais do que qualquer coisa que o dinheiro possa comprar. Também sou grato ao meu irmão João, meus tios e avós. Cada um deles me ajudou em minha jornada até hoje de maneira única e sempre estiveram dispostos a contribuir para o meu crescimento. Podem ter certeza de que cada um teve uma grande influência em minhas conquistas. À minha esposa Jéssica, obrigado pelo amor, paciência, compreensão e por sempre ter me apoiado e estar do meu lado, mesmo nos momentos mais difíceis.

Não poderia deixar de agradecer aos meus colegas de curso por todas as horas de estudo e lazer que compartilhamos. Em especial, agradeço ao João Paulo, Jefferson e Luiz Alfredo por terem dividido a moradia comigo, e ao Luiz Augusto e Lucas Heilbuth, com quem fizemos diversos trabalhos juntos sempre que possível. Aos professores, expresso minha eterna gratidão por cada aula e conhecimento compartilhado. Em particular, gostaria de agradecer ao Carlos Moratelli, que me orientou em diversos projetos, incluindo este trabalho, e nunca mediu esforços para me auxiliar.

"Há um tempo determinado para todas as coisas debaixo do céu."
(Eclesiastes 3:1)

RESUMO

O presente trabalho tem como objetivo implementar um RTOS para processadores RISC-V, com ênfase na garantia de separação entre as tarefas como medida de segurança. Nos capítulos iniciais, são apresentados os conceitos básicos relacionados a sistemas operacionais, incluindo os tipos de sistemas (GPOS e RTOS), destacando-se o Linux e o FreeRTOS, sistemas embarcados e Internet das Coisas. Também, são abordadas questões de segurança de software, arquitetura de processadores, com foco no RISC-V, e os recursos de proteção de memória do PMP (*Physical Memory Protection*) e Qemu. Na sequência, a arquitetura do sistema é discutida, assim como a implementação do escalonador Round-Robin com prioridades e envelhecimento. São apresentadas as funcionalidades desenvolvidas, incluindo a disponibilidade de *timer*, a utilização de mutex para sincronização entre tarefas e o mecanismo de comunicação entre tarefas. Além disso, são realizados dois testes: um teste de produtor/consumidor para avaliar as funcionalidades implementadas e um teste de segurança para verificar se o PMP proporcionou a separação adequada entre as tarefas.

Palavras-chave: RISC-V PMP; RTOS; Segurança.

ABSTRACT

This work aims to implement an RTOS for RISC-V processors, with emphasis on ensuring task separation as a security measure. In the initial chapters, the basic concepts related to operating systems are presented, including the types of systems (GPOS and RTOS), highlighting Linux and FreeRTOS, embedded systems and the Internet of Things. Also, software security issues, processor architecture, with a focus on RISC-V, and the memory protection features of PMP (Physical Memory Protection) and Qemu are covered. In the sequel, the system architecture is discussed, as well as the implementation of the Round-Robin scheduler with priorities and aging. The developed functionalities are presented, including the availability of timer, the use of mutex for synchronization between tasks, and the inter-task communication mechanism. In addition, two tests are performed: a producer/consumer test to evaluate the implemented functionalities and a security test to verify that the PMP provided adequate separation between tasks.

Keywords: RISC-V PMP; RTOS; Security.

LISTA DE FIGURAS

LISTA DE QUADROS

LISTA DE TABELAS

LISTA DE ABREVIATURAS E SIGLAS

CPU	*Central Processing Unit*
FPGA	*field-programmable gate array*
GPOS	*General-Purpose Operating System*
I/O	*Input and Output*
IPC	*Inter-Process Communication*
ISA	*Instruction Set Architecture*
PMP	*Physical memory protection*
RTOS	*Real-Time Operating System*
SO	Sistema Operacional
TCB	*Task Control Block*

SUMÁRIO

1 INTRODUÇÃO

Sistemas Embarcados são sistemas eletrônicos projetados para executar tarefas específicas e possuem limitações em termos de recursos computacionais, como processamento, memória e energia (VALAVANIS; VACHTSEVANOS, 2014). Eles são utilizados em diversos dispositivos, desde eletrodomésticos, automóveis, aeronaves e equipamentos médicos, além de serem fundamentais em aplicações de Internet das Coisas (IoT, do inglês, *Internet of Things*) onde, dispositivos conectados à rede precisam ser gerenciados e monitorados. Essa tecnologia está possibilitando a criação de novos modelos de negócios e melhorias significativas em diversos setores, como saúde, agricultura, indústria, transporte e cidades inteligentes (AKYILDIZ *et al.*, 2002). IoT consiste em conectar objetos físicos à internet, permitindo a comunicação entre eles e com sistemas de gerenciamento de dados. Entretanto, é importante destacar que a implementação de soluções de IoT também traz desafios importantes, como a segurança da informação e a privacidade dos usuários. Por isso, é necessário haver uma preocupação constante com a proteção de dados e informações sensíveis, bem como com a implementação de sistemas robustos e seguros. A segurança desses sistemas é uma questão crítica, já que muitos desses dispositivos estão conectados à Internet e podem ser alvos de ataques cibernéticos. Vulnerabilidades em dispositivos embarcados podem permitir que um invasor acesse informações confidenciais, controle o dispositivo remotamente ou até mesmo cause danos físicos (SADEGHI; WACHSMANN; WAIDNER, 2019).

É comum que se use sistemas operacionais em sistemas embarcados (LEE; SESHIA, 2015), devido as suas vantagens, como a redução do tempo e dos custos de desenvolvimento, além da capacidade de gerenciamento de recursos e execução de múltiplas tarefas em paralelo. Existem dois tipos de sistemas operacionais: os sistemas operacionais de propósito geral (GPOS, do inglês, *General Purpose Operating System*) e os sistemas operacionais de tempo real (RTOS, do inglês, *Real-Time Operating System*). Os GPOS, como o próprio nome sugere, são sistemas operacionais projetados para atender uma ampla variedade de aplicações em um computador, como a execução de tarefas do usuário, gerenciamento de recursos de entrada/saída, gerenciamento de memória e processos, e outras funcionalidades. Já os RTOS são sistemas operacionais projetados especificamente para aplicações de tempo real, onde a precisão temporal é crucial. Eles são amplamente utilizados em sistemas embarcados, pois oferecem gerenciamento eficiente de recursos e suporte a tarefas críticas em tempo real (DAVIS, 2017).

A família de processadores RISC-V é uma arquitetura aberta. Eles se diferenciam de outras arquiteturas de processadores por sua simplicidade e flexibilidade, permitindo que os desenvolvedores criem processadores personalizados que se adequem a suas necessidades (RISC-V FOUNDATION, 2022). Um dos recursos importantes do RISC-V é o PMP (*Physical Memory Protection*), que é um mecanismo de proteção de memória física que

permite definir uma região de memória e atribuir permissões específicas a essa região. Isso serve para garantir que um processo não acesse partes da memória que não lhe são permitidas, evitando possíveis vulnerabilidades de segurança (RISC-V FOUNDATION, 2022).

Nesse contexto, o objetivo deste trabalho é apresentar o projeto e implementação de um RTOS para processadores RISC-V, que forneça recursos que atendam à necessidades comuns em sistemas embarcados e que consiga gerar separação entre as tarefas como forma de segurança.

1.1 OBJETIVOS

O objetivo desse trabalho é implementar um RTOS que é executado em Processadores RISC-V e que garanta separação entre as tarefas como fator de segurança.

1.1.1 Objetivos Específicos

Para cumprir o objetivo geral, é preciso que os seguintes objetivos específicos sejam alcançados:

1. Implementar o escalonador de tarefas Round-Robin com prioridades e envelhecimento;
2. Implementar as chamadas do sistema que fornecem recursos de *timer*, sincronização e comunicação entre tarefas;
3. Realizar testes do RTOS com o emulador QEMU;
4. Realizar testes de segurança para verificar a separação entre as tarefas.

2 INTRODUÇÃO AOS SISTEMAS OPERACIONAIS

Um sistema operacional (SO) tem por objetivo facilitar a utilização do hardware. Os computadores modernos podem ter diversos processadores, dispositivos de armazenamento de dados, dentre outros componentes, o que os torna tão complexos que seria muito difícil desenvolver aplicações tendo que se preocupar com cada detalhe de seu funcionamento(TANENBAUM; BOS, 2015). Nesse capítulo, é comentado sobre os SOs e como podem ser divididos em duas categorias, sistemas operacionais de propósito geral (do inglês, *General Purpose Operating System* (GPOS)) e sistemas operacionais de tempo real (do inglês, *Real Time Operating System* (RTOS)), destacando as principais características de cada tipo. Para demonstrar a diferença entre as duas classes, também é apresentado dois exemplos de sistemas operacionais, Linux e FreeRTOS, e suas principais propriedades.

2.1 SISTEMA OPERACIONAL

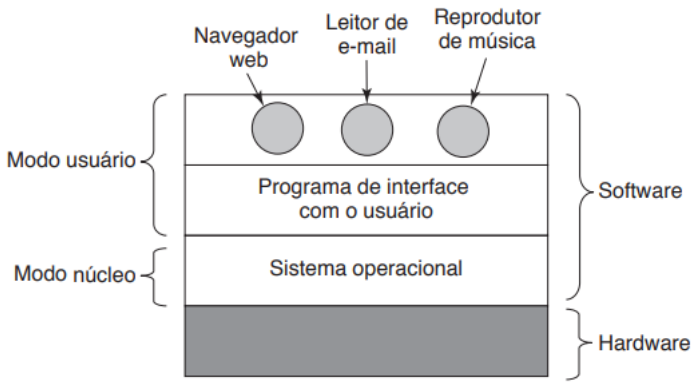
Um SO cria uma camada de abstração entre o hardware e os processos de usuário(OLIVEIRA, R., 2018). Assim, a utilização de um recurso, como acessar um periférico, é abstraída e isso torna a tarefa de criar softwares mais simples, já que o programador só precisa se preocupar em desenvolver sua aplicação. Essa camada de software é chamada de núcleo ou *kernel* do SO e também administra os recursos para os diversos processos, aumentando a eficiência do hardware e tornando-o capaz de executar diversas tarefas distintas simultaneamente.

Para ser possível utilizar um SO, é necessário que o processador trabalhe em pelo menos dois modos, sendo o primeiro chamado de Modo Núcleo que detêm total controle do hardware e o segundo chamado de Modo Usuário, com menos permissões. Como ilustrado na Figura Figura 1, os modos mencionados apresentam uma hierarquia entre si. Sempre que há a necessidade de utilizar um recurso indisponível para o Modo Usuário, o Sistema Operacional no Modo Núcleo é invocado através de uma chamada de sistema ou *syscall*.

As *syscalls* são solicitações que os programas fazem ao SO. Elas são efetuadas pelos programas sempre que algum recurso de hardware ou do Sistema Operacional precisa ser acessado, por exemplo, ao imprimir um texto ou se comunicar com outras aplicações. Cada arquitetura possui uma forma específica de lidar com isso, mas no geral, ao invocar uma chamada de sistema, o processador, que está executando a aplicação no Modo Usuário, é alternado para o Modo Núcleo, que executa a *syscall* e retorna à execução do processo de usuário, conforme exemplificado na Figura 2. Assim, os programas ficam separados e sem acesso ao hardware e não precisam se preocupar em como imprimir um documento ou ler o movimento do mouse, além de outros recursos que podem ser oferecidos pelo SO como execuções concorrentes e comunicação entre processos.

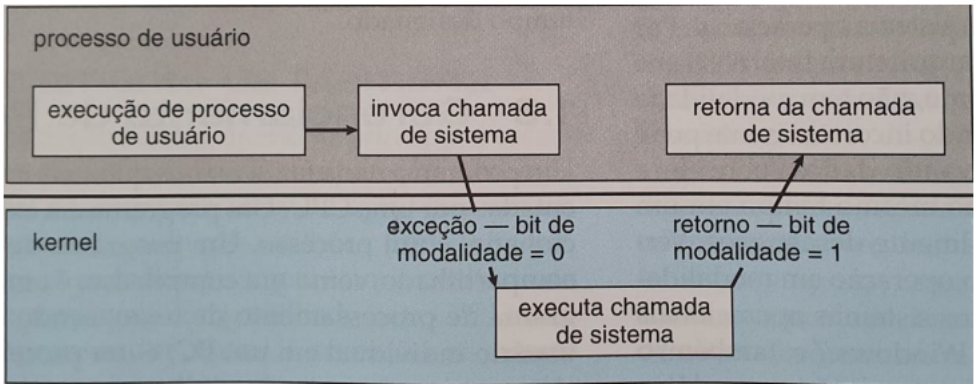
Cada atividade que está sendo gerenciada pelo SO precisa de alguns recursos, como CPU, memória, arquivos e dispositivos de I/O. Essa unidade de trabalho é conhecida

Figura 1 – Hierarquia dos Modos do Processador.



Fonte: (TANENBAUM; BOS, 2015).

Figura 2 – Exemplo de execução de uma *syscall*



Fonte: (SILBERSCHATZ; GALVIN; GAGNE, 2010).

como processo, e pode ser considerada como um programa em execução(SILBERSCHATZ; GALVIN; GAGNE, 2010). Assim, um programa é um arquivo com uma série de instruções armazenadas em disco. Quando esses dados são puxados para a memória com um contador que especifica a próxima instrução a ser executada e outros recursos, se torna um processo(SILBERSCHATZ; GALVIN; GAGNE, 2010).

Um processo pode assumir diversos estados, sendo eles:

- Novo: O processo é criado.
- Executando: As instruções estão sendo executadas.
- Em espera: O processo está esperando a ocorrência de algum evento.
- Pronto: O processo está esperando para ser executado.
- Finalizado: O processo terminou sua execução.

Na Figura 3, é possível visualizar esses estados e os eventos que podem alterá-los.

Assim, após o processo ser criado, ele é admitido pelo SO e passa para o estado pronto. Ao ser escalonado, passa a ser executado pelo processador, caso precise esperar por um evento muda para o estado em espera, e assim que o evento é completado, volta ao estado de pronto. Quando está executando, pode sofrer preempção, operação que será explicada posteriormente, e retorna ao estado de pronto. Quando sua execução é terminada, passa para o estado de finalizado.

Figura 3 – Estados de um processo.

Fonte: (SILBERSCHATZ; GALVIN; GAGNE, 2010).

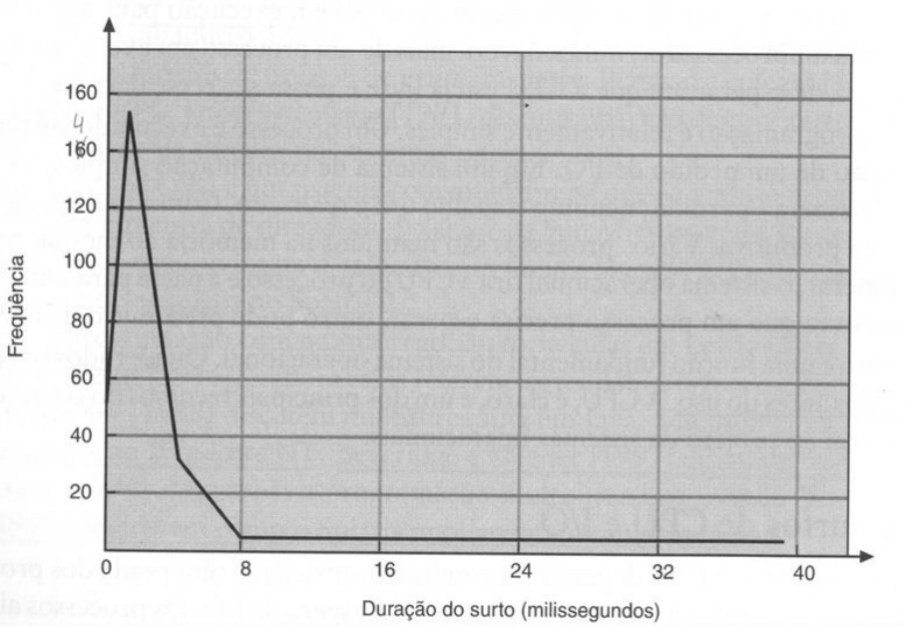
O escalonamento é o mecanismo de alternar os diversos processos que estão sendo executados por um sistema operacional. Sempre que a CPU fica ociosa, cabe ao sistema operacional através do escalonamento escolher qual processo deve ser executado. Isso ocorre diversas vezes, pois é comum que se entre em estado de espera de I/O, quer seja um periférico, escrita em disco ou comunicação com outro processo ou computador. Esse procedimento só é viável, pois os softwares costumam ter um surto de uso da CPU muito rápido e logo após uma grande espera por uma entrada ou saída. Como pode ser observado na Figura 4, a maioria dos surtos de uso de CPU são de pequena duração.

Sempre que um processo entra em estado de espera ou sua execução é terminada, é necessário escalonar um novo processo para ser executado pelo processador. Porém, alguns escalonadores possuem capacidade de interromper a execução de um processo para escalonar outro processo. Isso é chamado de preempção e o escalonador que possui essa capacidade é chamado de escalonador preemptivo e os que não conseguem realizar a preempção de um processo são chamados de escalonador não-preemptivo(TANENBAUM; BOS, 2015).

2.2 *GENERAL PURPOSE OPERATING SYSTEM*

Os computadores pessoais são objetos muito comuns ao dia-a-dia das pessoas. Para tornar seu uso acessível, sem que haja a necessidade de um conhecimento especializado, é comum que venha algum GPOS instalado. Como o seu nome diz, pode ser utilizado

Figura 4 – Histograma das durações de surto de CPU.



Fonte: (SILBERSCHATZ; GALVIN; GAGNE, 2010).

nas mais diversas aplicações e têm por objetivo facilitar o uso do hardware sem que seja necessário muitas modificações para utilizar ferramentas distintas, já que cada aplicação pode demandar recursos diferentes. Por exemplo, usuários podem executar navegadores, jogos, editores de texto, compiladores, players de vídeos, dentre outros softwares, que podem ser executados simultaneamente, e sem necessidade de recompilar ou reiniciar o sistema. Isso se deve ao trabalho de um GPOS, como o Linux, MacOS e Windows, que administra toda a execução dos mais diversos processos. Os usuários não costumam perceber o uso do sistema operacional, e esse é o seu objetivo, conforme disse Linus Torvalds(STROSS, 2001): "a principal característica de um sistema operacional é que você nunca deveria notar a sua presença."GPOSs costumam vir com diversos outros softwares além do *kernel*, como uma interface gráfica, compiladores e editores de texto. Esses recursos não são essenciais e não trabalham na separação e abstração do hardware, mas tornam possível que um usuário comum utilize o sistema computacional, sendo o conjunto formado pelo hardware e o software.

Devido ao seu uso geral, o GPOS apresenta grande complexidade para conseguir atender as suas demandas. Isso obriga que o software seja desenvolvido com o foco em desempenho e que o hardware, seja robusto o suficiente para conseguir executá-lo. Assim, existem aplicações que não podem utilizar esse tipo de SO, pois não necessitam de hardware complexo e/ou necessitam de resposta em Tempo Real. Para obter o desempenho necessário, um GPOS não costuma se preocupar com prazos para desempenhar a suas tarefas e não consegue ter previsibilidade com o tempo de execução para uma determinada

tarefa.

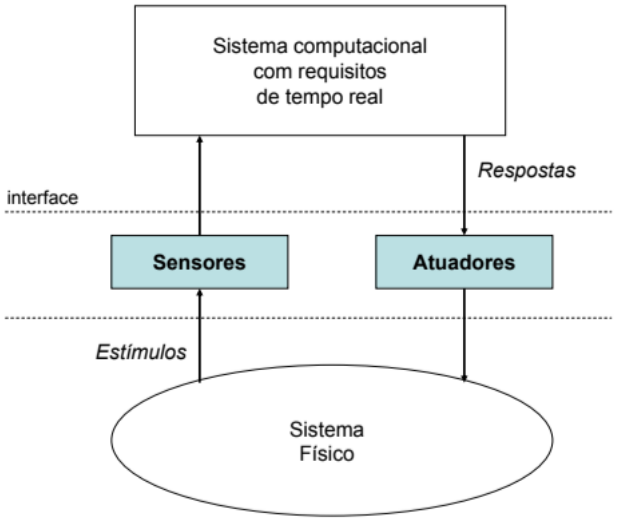
2.3 *REAL TIME OPERATING SYSTEM*

Outra classe de sistemas operacionais são os RTOSs (do inglês, *Real Time Operating System*). A sua do inglês,principal característica é garantir que requisitos de natureza temporal sejam cumpridos. Esses requisitos podem aparecer de diversas formas, conforme é definido por Rômulo Silva de Oliveira(OLIVEIRA, R. S., 2017):

> "Requisitos de natureza temporal podem aparecer de várias formas: um prazo máximo para a execução de uma dada tarefa, um período no qual dada tarefa deve ser sempre executada, um intervalo máximo de tempo entre duas ações, um intervalo máximo de tempo entre duas ações, um intervalo mínimo de tempo entre duas ações, um prazo máximo para a validade dos dados, etc."

Cabe ressaltar que a demanda por tempo real sempre é gerada pelo problema que se deseja resolver e não pela solução proposta. Ou seja, as características da aplicação e do problema que ela pretende resolver irão especificar certos requisitos de tempo real, cabendo ao sistema atender as demandas(OLIVEIRA, R. S., 2017). Os RTOSs são fortemente acoplados ao ambiente. Ocorrem estímulos ao sistema que deve processar as informações e gerar uma resposta física em um tempo determinado. A Figura 5 ilustra o acoplamento entre o sistema e o mundo físico.

Figura 5 – Sistema de tempo real acoplado ao ambiente.



Fonte: (OLIVEIRA, R. S., 2017).

O não cumprimento de prazos de tempo real podem trazer grandes danos. Por exemplo, quando o pedal de freio de um automóvel é acionado, a resposta deve ocorrer em um limite de tempo para que o carro pare sem danos, caso ocorra um atraso, pode

acontecer um acidente, pois o carro pode não parar a tempo de colidir com algum obstáculo. Sistemas onde o não cumprimento dos prazos trazem grandes danos são conhecidas como sistemas críticos ou de *hard real-time*(TANENBAUM; BOS, 2015).

Existem alguns sistemas que demandam tempo real, mas o não cumprimento de alguns prazos pode ser aceito sem muito efeitos colaterais. Por exemplo, um sistema de vídeo apresenta uma quantidade definida de imagens por segundo para dar impressão de movimento ao olho humano. Caso ocorra um atraso na geração das imagens, o vídeo irá ficar travado, o que não é ideal, mas não irá trazer dano aos usuários ou ao ambiente. Os sistemas cujas requisições podem aceitar alguns atrasos são definidos como sistemas não-críticos ou de *soft real-time*(TANENBAUM; BOS, 2015).

Os programas em execução em um SO são chamados de processos, conforme foi comentado na Seção 2.1. Porém, para RTOSs, as aplicações são chamadas de tarefas, e isso se deve a sua estrutura distinta, conforme definiu Rômulo Silva de Oliveira(OLIVEIRA, R. S., 2017): "[...] em sistemas de tempo real, o termo tarefa é usado de forma mais específica, significando a execução de um segmento de código que possui algum atributo ou restrição temporal própria, tal como um período ou um deadline."

Uma das aplicações mais comuns para RTOSs são os Sistemas Embarcados(OLIVEIRA, R. S., 2017). Para esses casos, o projetista tem que se preocupar com o custo do processador e é geralmente escolhido o com menor custo capaz de executar o processamento no tempo exigido (OLIVEIRA, R. S., 2017). Ainda é bastante comum ter recursos limitados para a execução de um RTOS, e por isso, eles costumam ser um sistema operacional pequeno, em termos de requisitos computacionais, que oferece poucos recursos, criando apenas uma pequena camada de separação entre o hardware e as tarefas. Devido à necessidade de ser um software pequeno, não é comum que um RTOS tenha muitos recursos de segurança e separação entre as tarefas, conforme será abordado na Seção 3.3.

2.4 LINUX E FREERTOS

O Linux é um kernel de sistema operacional escrito inicialmente por Linus Torvalds e mantido e atualizado por uma comunidade de desenvolvedores que se comunicam pela internet. Seu código é aberto, o que é uma das principais razões de seu sucesso, já que possui uma enorme comunidade que contribui com o projeto e permite ser utilizado de forma gratuito.

Sua história está ligada ao GNU(acrônimo recursivo que significa "GNU is not UNIX"), idealizado por Richard Stallman em 1983, que foi um projeto para criar um SO semelhante ao UNIX, porém, de código aberto. O UNIX é um SO criado por Dennis Ritchie e Ken Thompson em 1969 que trabalhavam para Bell Labs. Ficou bastante popular, mas é um software proprietário, o que levou a criação do GNU, já que o seu criador é um ativista que defende o software livre. Esse projeto recriou diversos componentes presentes em no UNIX, como um compilador C, biblioteca C e comandos essenciais do terminal.

Porém, houve dificuldade para se criar o kernel do sistema, o que impedia a finalização do sistema. Ao final da década de 80, essas ferramentas estavam finalizadas e, em 1991, Torvalds tornou seu kernel público. A comunidade percebeu a compatibilidade entre os dois projetos e, através da união dos dois, foi criado o GNU/Linux, sendo a base de diversas distribuições utilizadas em computadores pessoais, servidores, dispositivos móveis, dentre outros(PROJECT, 2023).

Como pode ser visto por suas diversas aplicações, as distribuições Linux são exemplos de GPOSs. Permitem a execução dos mais diversos programas e podem ser utilizados desde hardwares simples até supercomputadores. Porém, pode ser difícil e pouco funcional executá-lo em um hardware com poucos megabytes de memória(PRADO, 2019). Também não apresenta nenhuma garantia de resposta em tempo real no seu formato comum, apesar de haver modificações que visam melhorar seu desempenho para esses casos.

O FreeRTOS é um RTOS de código aberto e muito conhecido no mercado. Ele pode ser executado em diversos processadores embarcados, já que é suportado de maneira oficial por algumas dezenas de famílias(SUPPORTED DEVICES..., 2018). Também é possível aproveitar *ports*, alterações no software para permitir a sua execução em outro hardware, feitos pelos próprios usuários, que divulgam esses projetos, ou, se necessário, desenvolver seu próprio *port*.

Com esse sistema, é possível utilizar diversos recursos, dentre eles, ferramentas para sincronização, como *mutexes* e semáforos, *timers* e mensagens entre tarefas por região de memória compartilhada. Tudo isso, utilizando pouca memória, já que costuma ocupar apenas alguns kilobytes. Seu kernel possui a capacidade de gerenciar as tarefas a serem executadas de acordo com prazos e prioridades, garantindo uma boa resposta em tempo real(FREERTOS..., 2018).

2.5 ESTADO DA ARTE

Nesta seção, é discutido sobre três artigos que abordam assuntos importantes para o trabalho. Os artigos selecionados fornecem informações sobre algoritmos de escalonamento de prioridade baseados em Round-Robin, mecanismos de agendamento baseados em separação temporal e a verificação da proteção física de memória em arquiteturas RISC-V.

O primeiro artigo foi apresentado na Terceira Conferência Internacional sobre Manufatura Digital e Automação, propõe um algoritmo de escalonamento de prioridade baseado em Round-Robin para RTOS. O objetivo do algoritmo é lidar com a questão do envelhecimento de tarefas em sistemas com alta carga. O algoritmo proposto é capaz de garantir que as tarefas de alta prioridade recebam ciclos de CPU regulares, evitando assim o envelhecimento. Os autores realizaram experimentos e compararam o algoritmo proposto com outros algoritmos de escalonamento populares, demonstrando sua eficácia em termos de justiça e desempenho(ZHONG; ZENG, 2012).

Já o segundo artigo, publicado na revista IEEE *Transactions on Parallel and Distributed Systems*, propõe um mecanismo de agendamento baseado em separação temporal para multiprocessadores. O objetivo é evitar a interferência entre tarefas concorrentes e melhorar a previsibilidade dos sistemas em tempo real. O mecanismo proposto é baseado na atribuição de janelas temporais exclusivas para tarefas em diferentes núcleos, garantindo assim que as tarefas não interfiram umas nas outras. Os autores realizaram extensos experimentos utilizando simulações e comparações de desempenho para validar a eficácia do mecanismo proposto(HAN *et al.*, 2017).

O último artigo, apresentado na Conferência Internacional de Projeto Assistido por Computador (ICCAD), aborda a verificação do PMP em processadores RISC-V. A proteção física de memória é uma funcionalidade importante para garantir a segurança e isolamento de tarefas em sistemas operacionais de tempo real. Os autores propõem uma metodologia para a verificação formal dessa proteção usando técnicas de modelagem e verificação de hardware. O artigo descreve a aplicação dessa metodologia em um processador RISC-V de referência e apresenta resultados promissores que demonstram a eficácia da abordagem proposta(CHEANG *et al.*, 2021).

3 CONCEITOS BÁSICOS

Alguns temas são muito importantes para o desenvolvimento deste trabalho e o seu entendimento. Neste capítulo, são apresentados os principais conceitos sobre Sistemas Embarcados, Internet das Coisas, Segurança de Software, Arquitetura de Processadores, RISC-V e Qemu, que são assuntos relacionados a toda discussão dos capítulos posteriores.

3.1 SISTEMAS EMBARCADOS

Sistemas Embarcados (SEs) são sistemas computacionais que estão embutidos em um sistema físico e são dedicados a realização de tarefas relacionadas a esse sistema. Essa dedicação está relacionada com a capacidade do sistema de executar tarefas específicas para o propósito de um sistema maior que contém um sistema embarcado(SOUZA *et al.*, 2015).

Pode-se dizer que um SE é especializado em realizar a tarefa para a qual ele foi projetado. Devido a essa especialização, a economia de recursos é bastante importante, tanto em termos de hardware e software, e o projetista visa cumprir os requisitos do projeto com hardware mais barato e software mais otimizado possível.

Alguns sistemas costumam conter forte acoplamento com o meio físico(OLIVEIRA, R. S., 2017), o que os define como Sistemas Ciber-físicos, como automóveis e aviões. Devido a essa relação, costuma-se ter necessidades temporais a serem cumpridas, já que é necessário monitorar e responder o processo físico, geralmente em uma malha fechada(LEE; SESHIA, 2015). Isso não ocorre, com outros sistemas, como Televisões e Celulares, que não atuam diretamente no ambiente.

3.2 INTERNET DAS COISAS

A Internet das Coisas, ou IoT(*Internet of Things*) é a conexão de dispositivos eletrônicos à rede mundial de computadores. Sejam eles, objetos das residências, como geladeiras e ar-condicionados ou aplicações industriais, IoT visa integrá-los à Internet para diversas utilidades, mas principalmente para automação(VIEIRA *et al.*, 2014). Por exemplo, uma câmera de segurança conectada à rede pode transmitir imagens para um aplicativo em um celular de qualquer lugar do mundo, desde que esteja conectado à Internet. Esse conceito está fortemente ligado aos sistemas embarcados, pois para haver conexão, é necessário um sistema computacional que está geralmente dedicado a tarefa exercida pelo dispositivo. Essa mudança de paradigma traz diversos desafios, dentre eles, questões de segurança, já que uma conexão à rede sempre está suscetível a ataques(VIEIRA *et al.*, 2014).

3.3 SEGURANÇA DE SOFTWARE

Qualquer sistema computacional lida com informações. Essas informações podem ser sigilosas e de extrema importância, como dados bancários e informações de localização. Quando se trata de sistemas operacionais, por ser um sistema que permite a utilização por diversos usuários e multi-processos, o acesso aos dados precisa ser controlado(SILBERSCHATZ; GALVIN; GAGNE, 2010). Assim, um software malicioso instalado em um SO pode atacar processos para tentar obter dados sigiloso. Por isso, é comum que GPOSs tenham ferramentas para evitar que esses softwares sejam bem sucedidos em seu ataque.Esse tema tem ainda mais relevância quando se trata de um sistema com acesso à Internet. Como essa rede interliga milhões de computadores, o sistema encontra-se mais vulnerável e suscetível a ataques.

Em sistemas com forte acoplamento físico, um ataque bem-sucedido pode causar danos no mundo físico. Por exemplo, se um terrorista invade o sistema de um carro, ele pode gerar um acidente que vai trazer danos materiais e humanos. Esse tema é muito importante para à Internet das Coisas, pois há o risco de vazamento de informações e também o risco de danos à integridade do sistema. Por exemplo, um sistema de segurança pode ter uma câmera que está coletando dados importantes cujo vazamento pode violar à privacidade das pessoas e pode controlar um portão que pode ser utilizado como arma ao fechar em cima de uma pessoa. Os ataques podem ser divididos em duas categorias: aqueles que visam ao sigilo das informações e aqueles relativos à integridade e controle do sistema(SOUZA *et al.*, 2015).

Como sistemas embarcados por muito tempo não tinham conexão à Internet, a segurança de software não era algo tão relevante, pois todo o código que está sendo executado costuma ser conhecido e também não havia contato com sistemas externos, sendo um sistema isolado. Com o advento da IoT, os sistemas embarcados passaram a ter conexão com a rede mundial de computadores e, portanto, estão suscetíveis a ataques.

3.3.1 Separação Espacial

Uma das formas de um SO garantir que softwares mal-intencionados não ataquem outros é através da separação espacial. Cada software deve ocupar um espaço da memória administrado pelo Sistema Operacional, e não deve ter acesso ao espaço ocupado pelas outras tarefas e pelo *kernel*. GPOS possuem sistemas complexos de gerenciamento de memória para garantir a separação espacial. A quantidade de processos varia com o tempo e podem ser criados em tempo de execução, além de alocarem e desalocarem memória, e isso demanda um custo computacional para ser administrado.

Já os RTOS costumam ter uma quantidade fixa de tarefas definidas antes da compilação do sistema, não havendo iniciação de tarefas em tempo de execução. Também é comum que uma quantidade fixa de memória seja disponibilizada para cada tarefa. Essas

escolhas de projeto são necessárias para diminuir o custo computacional, que costuma ser um fator bem limitante para RTOS. Dessa forma, o gerenciamento de memória em sistemas de tempo real costuma ser bastante simples, o que é bom em termos de custo de hardware, mas não costuma considerar questões de segurança.

3.4 ARQUITETURA DE PROCESSADORES

Um processador moderno trabalha através da execução de instruções, que são operações suportadas pelo hardware. Cada processador possui um tipo de arquitetura, o qual é a estrutura do hardware, e também um conjunto de instruções chamado de ISA (*Instruction Set Architecture*)(TANENBAUM; AUSTIN, 2013), e é a linguagem de máquina específica de cada família de processadores. Essas instruções definem comandos básicos com somar dois números e carregar ou salvar um valor na memória.

Cada instrução possui uma representação legível para um ser humano, definido pela Linguagem Assembly, sendo uma linguagem de programação onde há correspondência direta entre as suas instruções com as instruções da ISA. Cada comando escrito em Assembly pelo programador é transformado em um código binário que o processador entende como uma instrução. Algumas linguagens, como C, costumam utilizar a linguagem de montagem como passo intermediário para chegar no código binário, sendo possível, para essa linguagem, adicionar código Assembly no meio de um código-fonte em C.

Antigamente, era bastante comum a escrita de códigos Assembly para se obter um bom resultado em termos de desempenho, se desenvolvido por alguém experiente. Porém, atualmente, os compiladores costumam ser muito eficientes e só costumam ter um resultado pior em casos muito específicos. Mesmo assim, em algumas situações é necessário escrever trechos de código em Assembly para acessar certos recursos que só podem ser acessados dessa forma.

Como cada arquitetura de processador possui seu próprio conjunto de instruções, é necessário que cada família tenha o seu próprio ecossistema de software, o que costuma tornar certas famílias bem populares. Por exemplo, os processadores x86 são largamente utilizados em computadores pessoais, e uma das grandes razões é a grande quantidade de softwares desenvolvidos para funcionar nesses hardwares, como compiladores, sistemas operacionais, drivers, dentre outros.

3.4.1 CISC versus RISC

Quando se trata de arquitetura de processadores, existem dois paradigmas que podem ser seguidos, sendo o primeiro dele o CISC, que significa *Complex Instruction Set Computer* (computador com conjunto de instruções complexo)(TANENBAUM; AUSTIN, 2013). Conforme o seu nome diz, ISAs que seguem essa filosofia tendem a ter instruções mais complexas e uma grande quantidade de instruções distintas, chegando a casa de

muitas centenas de operações que podem ser executadas pelo processador. Isso facilita a programação em liguagem Assembly, já que será preciso utilizar menos instruções e elas são mais poderosas, pois executam operações mais complexas. Porém, isso torna o hardware complexo e o tempo de processar instruções mais complexas aumenta. Além disso, novas gerações precisam solucionar os novos problemas com novas instruções e o tamanho de uma ISA de filosofia CISC só aumenta.

O outro paradigma que nortea um projeto de um processador chama-se RISC, ou *Reduced Instruction Set Computer* (computador com conjunto de instruções reduzido). É caracterizado por possuir uma ISA com poucas instruções que são mais simples em comparação as instruções CISC. O seu projeto de hardware simplificado resulta em um menor tempo propagação dos bits pelo meio físico. Isso, por sua vez, leva a um tempo de processamento de instrução reduzido, permitindo um tempo de clock maior, conforme disse Andrew S. Tanebaum(TANENBAUM; AUSTIN, 2013):

> "[...]mesmo que uma máquina RISC precisasse de quatro ou cinco instruções para fazer o que uma CISC fazia com uma só, se as instruções RISC fossem dez vezes mais rápidas (porque não eram interpretadas), o RISC vencia."

Durante os anos, parecia ser questão de tempo que os processadores RISC iriam ter um desempenho melhor e dominar o mercado(TANENBAUM; AUSTIN, 2013). Porém, isso não ocorreu por, basicamente, dois motivos:

- Os processadores CISC da Intel dominavam o mercado, e era preciso que mantivesse toda a compatibilidade, o que tornava uma transição para processadores RISC complicada.
- As CPUs da Intel passaram a ter uma arquitetura híbrida, com um núcleo RISC para execução de instruções mais simples e outro CISC que executava as instruções mais complexas(TANENBAUM; AUSTIN, 2013).

Apesar do domínio da Intel e seus processadores híbridos no mercado de computadores pessoais, em outros mercados, como o de consoles e de celulares, há um domínio de processadores RISC, como os da família ARM. Além da vantagem de desempenho, os processadores RISC apresentam um consumo de energia menor, já que o hardware é mais simples e mais compacto, ocupando também menos espaço(TANENBAUM; AUSTIN, 2013).

3.5 RISC-V

O mercado dos processadores é dominado por poucas arquiteturas, sejam, RISC, CISC, ou híbridas. Apesar de suas diferenças e de dominarem nichos diferentes, processadores ARM, MIPS, x86, dentre outros, possuem uma característica em comum, é necessário pagar por licenças para poder produzi-los, já que empresas, como IBM, ARM e Intel, patentearam as arquiteturas.

Com base na necessidade de uma ISA livre, uma nova arquitetura chamada RISC-V foi criada em 2010 na Universidade de Berkeley, Califórnia, por pesquisadores para ser utilizado na educação e pesquisas e essa segue a filosofia de projeto RISC. Porém, como pode ser utilizada livremente, surgiu o interesse de sua utilização comercial. As principais metas que nortearam a concepção do RISC-V foram:

- Um ISA completamente livre (gratuito) para a academia e a indústria;
- Permitir implementação em hardware e não apenas simulação;
- Ser genérica o bastante para permitir implementações eficientes em hardware ou FPGA ou outras variações;
- Permitir fácil extensão e variações especializadas;
- Suportar implementações *multicore* ou *manycore*.

A arquitetura RISC-V oferece até três níveis de proteção ou modos de operação. Pode-se optar por ter apenas um nível, que limita o uso a aplicações *bare metal* sem a presença de um sistema operacional. Também é possível ter dois níveis de proteção, permitindo a utilização de um sistema operacional. Por fim, o RISC-V oferece um terceiro nível de proteção, que permite a virtualização e a execução de vários sistemas operacionais simultaneamente em um único hardware.

3.5.1 PMP

O PMP (*Physical memory protection*), ou Memória de Proteção Física é uma funcionalidade que pode estar contida em um processador RISC-V que permite proteger e controlar regiões de memória(CHEANG *et al.*, 2021). Isso pode ser utilizado para gerar separação espacial de regiões críticas, como o *kernel* e as tarefas que não estão sendo executadas. Essa proteção é realizada através da configuração de registradores que só estão disponíveis quando o processador está no modo *kernel*, e o hardware verifica a cada execução se há ou não violação das áreas protegidas.

Para configurar o PMP em um processador RISC-V, é necessário configurar os registradores PMP conforme as políticas de proteção desejadas. Existem 2 tipos de registradores que precisam ser configurados, o *PMPADDR* e o *PMPcfg*.

O registrador *PMPADDR* define o endereço inicial e final da região protegida(MAZIDI, M., 2015). Um processador pode possui até 16 registradores desse tipo, indo do *PMPADDR0* ao *PMPADDR15*.

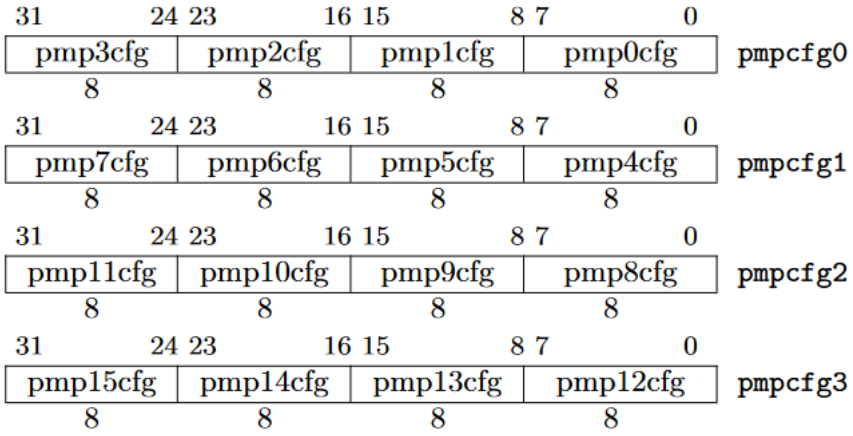
O registrador *PMPcfg*, cujo formato é apresentado na Figura 6, define a política de proteção para a região de memória protegida pelo registrador *PMPADDR* correspondente. Ele é dividido em três tipos de campos:

- Permissão(R, W e X): define as permissões de acesso para a região protegida, como permissão de leitura, permissão de gravação e permissão de execução.

Possui 3 bits, sendo que o R indica permissão de leitura, o W permite permissão de escrita e o X de execução de instrução.

- Correspondência de Endereço(A): define como os endereços serão lidos, sendo 1 para correspondência normal, e 2 para correspondência em potência de 2.
- Controle(L): define a política de proteção específica para a região protegida. Ele pode especificar se a região protegida é global (válida em todos os modos) e a configuração só pode ser modificada ao reiniciar o processador, ou local, que válida apenas no modo de privilégio menores e pode ser reconfigurado.

Figura 6 – Registradores PMPcfg.



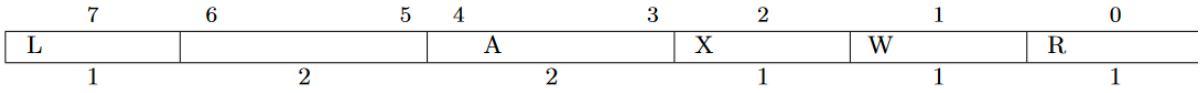
Fonte: (RISC-V FOUNDATION, 2021).

Cada registrador *PMPcfg* possui 4 regiões de 8 bits, sendo necessário 4 registradores de 32 bits para controlar 16 registradores *PMPADDR*, conforme a Figura 7. Cada parte do *PMPcfg* controla uma região formada pelos registradores *PMPADDR*, assim, o *PMP0cfg* controla a região formada do início da memória até o endereço salvo no *PMPADDR0*, o *PMP1cfg* controla a região entre o *PMPADDR0* e *PMPADDR1*, e assim por diante.

3.6 QEMU

O QEMU é software capaz de emular arquiteturas de processadores. Ele consegue ler instruções de um software compilado para uma arquitetura e traduzi-las para outra ISA(OLIVEIRA, L. S.; NOGUEIRA; FREITAS, 2022). Por exemplo, um SO Linux sendo executado em um processador x86 com o QEMU instalado pode executar um código gerado para processadores RISC-V. Essa ferramenta é muito útil para desenvolver e testar aplicações sem a necessidade de ter um hardware real. Isso torna o processo de desenvolvimento mais barato, e muitas vezes mais prático de testar, principalmente em fases iniciais de projetos.

Figura 7 – Formato do PMPcfg.



Fonte: Modificado de (RISC-V FOUNDATION, 2021).

4 PROJETO E ESPECIFICAÇÃO DE UM RTOS

O projeto e especificação de um RTOS é uma etapa crucial no seu desenvolvimento. Nesse capítulo, vamos explorar as principais funcionalidades de um RTOS, incluindo o escalonador Round-Robin com prioridades e envelhecimento, a comunicação entre tarefas, o acesso ao *timer* e mutexes, propostos para o RTOS. Além disso, vamos discutir a importância da separação espacial e temporal entre tarefas para garantir a segurança do sistema.

4.1 ESCALONADOR ROUND-ROBIN

O Escalonador Round-Robin é um dos algoritmos de escalonamento mais populares em RTOSs. Seu funcionamento é o baseado em tempo, onde é atribuído um tempo fixo para cada tarefa ser executada (também conhecido como *quantum*), executando-as em ordem circular. Quando o tempo de uma tarefa expira, esta é preemptada, passando-se para próxima da fila e iniciando um novo *quantum*. Esse processo é repetido até que todas as tarefas sejam concluídas(ZHONG; ZENG, 2012).

RTOSs tem como principal característica a previsibilidade, o que é essencial para sistemas de tempo real. Além disso, é desejável definir prioridades para as tarefas, para garantir o cumprimento dos limites temporais, o que pode ser feito com uma variação do algoritmo Round-Robin que contém prioridades(SILBERSCHATZ; GALVIN; GAGNE, 2010). Cada tarefa recebe uma prioridade, usada pelo escalonador para determinar qual tarefa deve ser executada primeiro. Quando há várias tarefas com a mesma prioridade, o escalonador usa o algoritmo Round-Robin para decidir qual tarefa deve ser executada. Porém, isso pode causar problema de *starvation*, que ocorre quando um processo com menor prioridade fica bloqueado ou adiado indefinidamente porque processos com prioridades mais altas estão constantemente ocupando o processador. Em outras palavras, o processo com menor prioridade nunca é executado, porque sempre há outros processos com prioridade mais alta na frente.

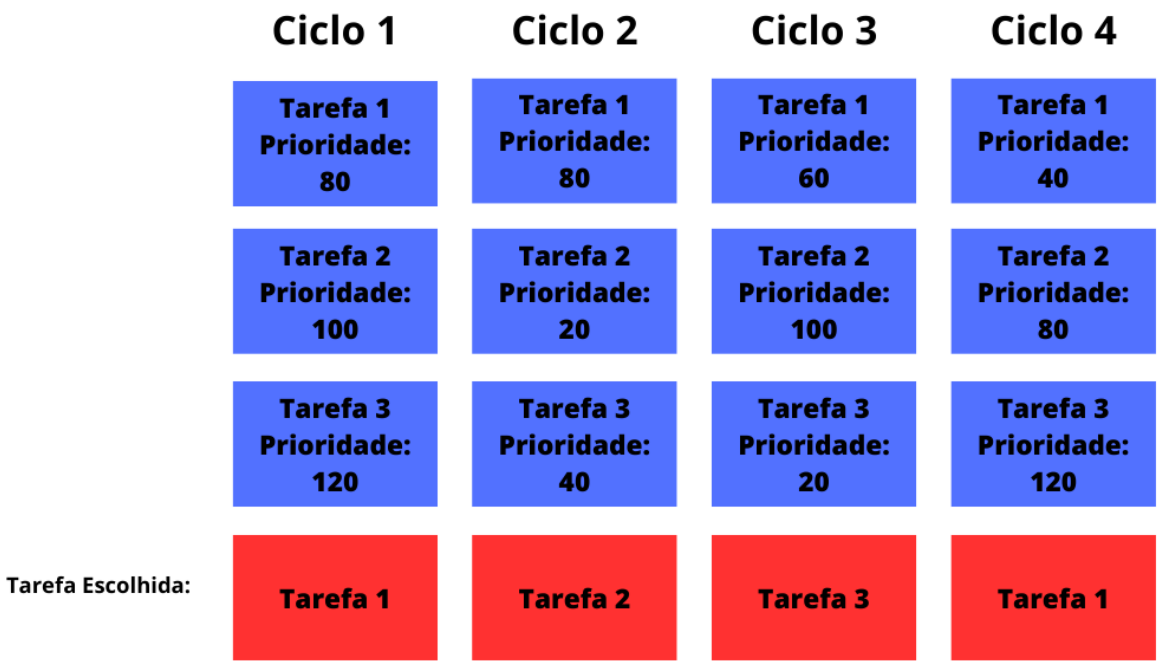
Uma das técnicas para solucionar o problema de *starvation* é o envelhecimento (TANENBAUM; BOS, 2015). A ideia por trás dessa técnica é aumentar gradualmente a prioridade das tarefas à medida que eles aguardam na fila de espera, de forma que nenhum processo fique bloqueado indefinidamente. Dessa forma, mesmo que inicialmente o processo tenha uma prioridade mais baixa, ele pode acabar tendo sua prioridade aumentada até ser executado, evitando assim o problema de *starvation*.

Devido a essas considerações, o escalonador do RTOS implementará o escalonador Round-Robin com prioridades e envelhecimento. Isso traz certo custo computacional, se comparado a algoritmos mais simples, porém garante o cumprimento dos limites temporais sem a ocorrência de *starvation*.

O algoritmo consiste em atribuir um tempo de processador a cada tarefa de forma

circular e repetitiva, evitando que uma tarefa monopolize o processador. A cada interrupção do timer, o escalonador seleciona a próxima tarefa a ser executada de acordo com sua prioridade. Caso haja mais de uma tarefa com a mesma prioridade, é utilizado o envelhecimento para evitar que tarefas fiquem muito tempo esperando na fila. Essa técnica garante que tarefas de menor prioridade sejam eventualmente executadas, mesmo que não haja mudanças nas tarefas de maior prioridade. Assim, o escalonador seleciona a tarefa que deve ser executada escolhendo a com maior prioridade, onde um valor numericamente menor corresponde a uma prioridade maior, ele atualiza o valor das prioridades das outras tarefas para o próximo ciclo, subtraindo da prioridade restante de cada tarefa a prioridade restante da tarefa selecionada, como pode ser visualizado na Figura 8. Isso garante que todas as tarefas tenham a mesma quantidade de tempo de CPU e que o escalonador siga a política Round-Robin de envelhecimento.

Figura 8 – Diagrama de ciclos do Algoritmo.



Fonte: Do Autor.

4.2 FUNCIONALIDADES DO SISTEMA

Algumas funcionalidades do sistema são fundamentais em um RTOS para permitir que as tarefas realizem suas atividades. Neste trabalho, será implementado três funcionalidades essenciais: acesso ao *timer*, exclusão mútua e comunicação entre tarefas.

O acesso ao *timer* é uma funcionalidade essencial de um RTOS, pois permite que o programador consiga medir o tempo em uma tarefa específica, por exemplo, quando é necessário esperar um tempo mínimo, ou verificar um prazo máximo. Nos processadores RISC-V, existe um registrador específico que só pode ser acessado em Modo Núcleo que armazena quantidade de ciclos de *clock* desde que o processador foi iniciado e com base em sua frequência, é possível calcular o tempo.

A exclusão mútua é um conceito fundamental em sistemas concorrentes, onde vários processos ou tarefa competem pelo acesso a recursos compartilhados(TAUBENFELD, 2006). Em resumo, a exclusão mútua é uma técnica usada para garantir que apenas uma tarefa possa acessar um recurso compartilhado por vez.

A funcionalidade de mutexes é importante para garantir que recursos compartilhados sejam acessados exclusivamente por uma tarefa em um determinado momento. Quando várias tarefas tentam acessar o mesmo recurso simultaneamente, pode ocorrer uma condição de corrida, o que pode levar a inconsistências de dados. Os mutexes ajudam a evitar esses problemas, permitindo que apenas uma tarefa acesse o recurso compartilhado por vez.

Um *mutex* é uma estrutura de dados usada para indicar que um recurso está sendo acessado por um processo específico. Enquanto um recurso estiver sendo usado, o *mutex* ficará bloqueado e outros processos não poderão acessá-lo. Quando o recurso não estiver mais sendo utilizado, deve-se liberar o *mutex* para permitir acesso(BUTENHOF, 1997).

Por fim, a funcionalidade de comunicação entre tarefas é essencial para permitir que as tarefas se comuniquem e sincronizem suas ações. As tarefas podem enviar e receber mensagens ou sinais para se comunicar, permitindo que ações específicas sejam tomadas em resposta a eventos. Isso será implementado através de IPC(do inglês,*Inter-Process Communication*) que é uma comunicação entre processos através do envio de mensagens.

4.3 SEPARAÇÃO ENTRE TAREFAS

Uma das formas de garantir segurança e estabilidade em SOs, é através da separação espacial e temporal. Essas técnicas consistem em isolar diferentes processos uns dos outros para garantir a segurança e a estabilidade do sistema(HAN *et al.*, 2017).

A separação espacial envolve a alocação de recursos de hardware e software, como memória, CPU, dispositivos de entrada/saída e arquivos, para garantir que cada processo ou tarefa tenha seu próprio espaço exclusivo. Isso evita que um processo interfira ou acesse acidentalmente os recursos de outro processo, o que pode causar falhas ou vulnerabilidades

de segurança. Para que a separação espacial seja eficiente e confiável, é necessário atender a requisitos específicos de segurança, como(SRIDHARAN; GARG, 2016):

- Isolamento de recursos: cada tarefa deve ter acesso apenas aos recursos que foram explicitamente designados para ela, evitando que uma tarefa possa interferir ou corromper a memória de outras tarefas.
- Controle de acesso: é importante que apenas tarefas autorizadas possam acessar determinados recursos. Para isso, é necessário que o sistema de segurança tenha um mecanismo de controle de acesso adequado.
- Proteção contra ataques: o sistema deve ser projetado para resistir a tentativas de violação por parte de usuários mal-intencionados. Para isso, é necessário que o sistema tenha um conjunto de mecanismos de proteção que possam identificar e bloquear tentativas de invasão.
- Integridade: é importante que a integridade das informações de uma tarefa seja preservada, evitando que elas sejam corrompidas ou modificadas por outras tarefas.
- Disponibilidade: é importante que o sistema esteja disponível para atender às necessidades dos usuários, mesmo em condições adversas. Para isso, é necessário que o sistema tenha mecanismos de manter o funcionamento das tarefas, mesmo que uma esteja comprometida.

Esses requisitos podem ser garantidos pelo uso do PMP dos processadores RISC-V, que é uma abordagem eficiente para garantir a segurança do sistema em tempo real. O PMP permite separar o espaço de endereçamento de diferentes tarefas, o que significa que cada tarefa só pode acessar as áreas de memória designadas para ela. Dessa forma, caso uma tarefa apresente problemas ou seja comprometida, as demais tarefas e áreas de memória permanecerão seguras. Através do PMP, somente as regiões da tarefa terão acesso permitido no Modo Usuário, enquanto o Kernel irá definir as regiões e mudar as permissões de acordo com as trocas de contexto.

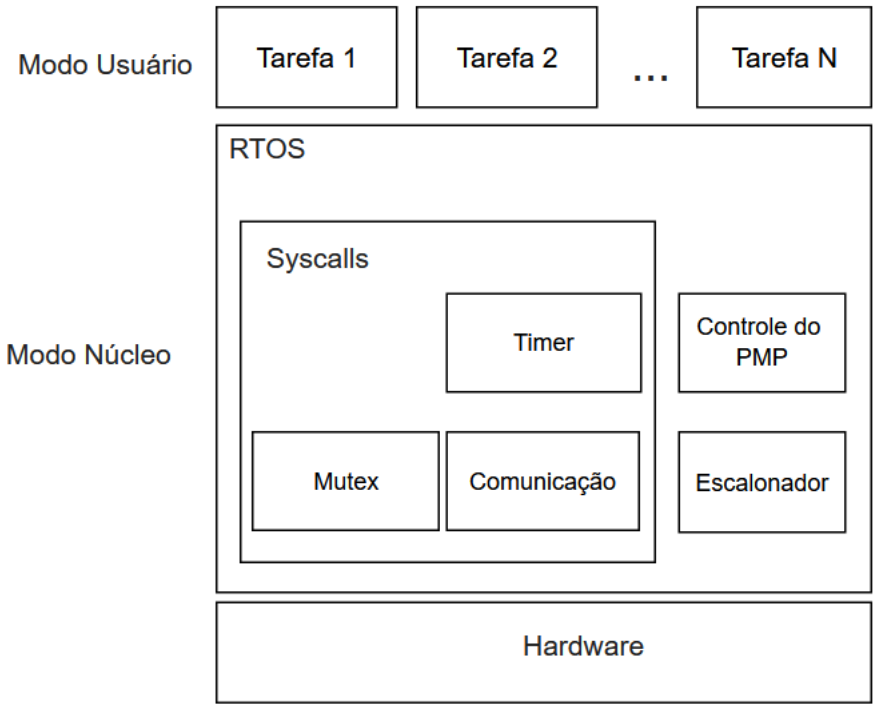
A separação temporal refere-se ao gerenciamento do tempo de execução de diferentes processos. O SO utiliza técnicas de escalonamento para garantir que cada processo receba uma quantidade justa de tempo de CPU e que nenhum processo monopolize o tempo de CPU. O escalonador Round-Robin, combinado com as técnicas de prioridade e envelhecimento, garante a separação temporal entre as tarefas em um RTOS, ao dar prioridade às tarefas mais importantes, mas garantindo que as tarefas menos importantes não sofram *starvation*.

4.4 ARQUITETURA PROPOSTA

A arquitetura proposta para o RTOS desenvolvido é composta por diversos componentes que trabalham em conjunto para garantir a execução confiável e segura de tarefas

em tempo real, sendo possível visualizar um esquema na Figura 9. O RTOS está localizado acima do hardware e é dividido em dois modos: Modo Núcleo, onde se encontram as *syscalls* e outros componentes críticos do sistema, e Modo Usuário, onde as tarefas do usuário são executadas.

Figura 9 – Arquitetura do RTOS proposto.



Fonte: Do Autor.

O escalonador é um dos principais componentes do RTOS e é responsável por gerenciar o tempo de CPU de cada tarefa e decidir qual tarefa deve ser executada em cada momento. A arquitetura proposta utiliza o escalonador Round-Robin com prioridades e envelhecimento, que garante a separação temporal entre as tarefas e evita que uma tarefa monopolize a CPU. Além disso, o mecanismo do PMP é utilizado para garantir a separação espacial entre as tarefas. Ele define as regiões de memória a que cada tarefa tem acesso, evitando que uma tarefa possa interferir ou corromper a memória de outras tarefas. As *syscalls* implementadas no Modo Núcleo do RTOS incluem o *timer*, os mutexes, utilizados para garantir a sincronização e a exclusão mútua, e a comunicação entre as tarefas, que permite que elas compartilhem informações e cooperem entre si. Por fim, as tarefas do usuário são executadas no Modo Usuário do RTOS, onde elas têm acesso aos recursos do sistema através das *syscalls*. A arquitetura proposta garante a separação entre as tarefas e a execução confiável e segura de tarefas em tempo real.

5 IMPLEMENTAÇÃO E RESULTADOS

Neste capítulo, será apresentado a implementação do RTOS. Serão detalhados os principais aspectos da estrutura das tarefas, compilação do sistema, escalonador, timer, mutex, comunicação e PMP.

5.1 TCB

O Bloco de Controle de Tarefas(do inglês, *Task Control Block* (TCB) é uma estrutura de dados fundamental em um RTOS. É responsável por armazenar todas as informações necessárias para a execução de uma tarefa, incluindo seu estado, prioridade, identificador, tempo restante de execução e outros atributos (MAZIDI, M. A., 2015). A TCB permite que o sistema operacional gerencie adequadamente o escalonamento de tarefas e o controle de recursos, garantindo a execução correta e oportuna das tarefas. Portanto, é essencial que a TCB seja projetada de maneira eficiente e robusta para garantir o desempenho adequado do sistema (LIU; LAYLAND, 1973). Nesta seção, vamos explorar a estrutura do TCB do RTOS proposto, cuja estrutura encontra-se no Código 5.1, e suas variáveis internas em detalhes.

```
1  struct task_entry{
2      uint16_t id;
3      int8_t name[20];
4      uint8_t state;
5      uint8_t priority;
6      uint8_t priority_rem;
7      uint8_t critical;
8      uint8_t init;
9      uint32_t delay;
10     uint32_t rtjobs;
11     uint32_t bgjobs;
12     uint32_t deadline_misses;
13     uint16_t period;
14     uint16_t capacity;
15     uint16_t deadline;
16     uint16_t period_rem;
17     uint16_t capacity_rem;
18     uint16_t deadline_rem;
19     uint32_t cp0_registers[3];
20     uint32_t pc;
21     uint32_t task_sp;
22     uint32_t task_tp;
23     void (*ptask)(void);
24     size_t *pstack;
25     uint32_t stack_size;
26     uint32_t *gpr;
```

```
27      uint32_t *ipc;
28      uint32_t mutex[MAX_MUTEX];
29      void *other_data;
30  };
```

Código 5.1 – Estrutura da TCB

Na TCB (Task Control Block), existem algumas variáveis que são utilizadas pelo escalonador para gerenciar as tarefas do sistema, como:

- *id*: identificador da tarefa, é um valor único que permite ao escalonador identificar a tarefa no sistema.
- *state*: estado atual da tarefa que permite ao escalonador saber se a tarefa está pronta para ser executada ou se está aguardando algum evento.
- *priority*: prioridade da tarefa, que pode variar de 1 a 255, e é utilizada pelo escalonador para determina a ordem de execução das tarefas.

Essas variáveis são essenciais para o escalonador, pois permitem que ele tome decisões sobre a execução das tarefas e garanta que o sistema opere de forma eficiente e confiável.

Na TCB, existem variáveis que são utilizadas para o controle de memória, permitindo o gerenciamento dos recursos disponíveis no sistema. São elas:

- **pstack**: ponteiro para a área da pilha da tarefa.
- **stack_size**: tamanho da pilha da tarefa.
- **gpr**: ponteiro para o endereço onde os registradores gerais serão salvos.

Essas variáveis são essenciais para o gerenciamento correto da memória, permitindo que as tarefas acessem os recursos necessários sem gerar conflitos ou corrupção de dados. O uso correto dessas variáveis garante a estabilidade e a segurança do sistema.

As variáveis *task_sp* e *task_tp* também têm papel importante no gerenciamento de memória da TCB. *task_sp* é o valor atual do ponteiro de pilha da tarefa, enquanto *task_tp* é o valor atual do registrador de ponteiro de topo da tarefa. Essas informações são cruciais para garantir que a tarefa tenha acesso somente a sua própria área de pilha, evitando corromper dados de outras tarefas.

Além dessas variáveis, a TCB também possui um ponteiro para uma área de memória utilizada para o envio e leitura de mensagens, e um vetor de inteiros que representam mutexes. A área de memória pode ser utilizada para comunicação e compartilhamento de dados entre diferentes tarefas e os mutexes são utilizados para garantir sincronização.

5.2 IMPLEMENTAÇÃO DO ESCALONADOR

Conforme especificado na Seção 4.1, o escalonador implementado no sistema é o Round-Robin, com prioridades e envelhecimento. A primeira função a ser executada é a

run_scheduler(), que pode ser visualizada no Código 5.2, e é invocada quando há uma interrupção do temporizador do sistema operacional, executado em intervalos regulares. A cada interrupção, a função verifica se é necessário fazer a troca de contexto para outra tarefa. Isso ocorre se:

- Já passou o tempo determinado desde a última preempção;
- A tarefa em execução está bloqueada.

```
1  void run_scheduler(){
2    struct task_entry *aux;
3    if ( tick_count % TICKS_BEFORE_SCHEDULING == 0 ||
4      pending ||
5      (task_in_execution && task_in_execution->state == TASK_BLOCKED))
6    {
7      aux = round_robin_scheduler();
8      if (aux != scheduler_info.task_executing || aux == NULL) {
9        contextSave();
10       scheduler_info.task_executing = aux;
11       contextRestore();
12     }
13
14   }
15   tick_count++;
16 }
```

Código 5.2 – Função *run_scheduler()*.

Se uma das condições for verdadeira, a função *round_robin_scheduler()*, que está no Código 5.3, é invocada para escolher a próxima tarefa a ser executada. Essa função implementa o algoritmo de escalonamento.

Se a tarefa escolhida for diferente da tarefa em execução atual, ou se não houver tarefa escolhida, a função *contextSave()* é chamada para salvar o contexto da tarefa em execução. Em seguida, a função *contextRestore()* é chamada para restaurar o contexto da nova tarefa. Dessa forma, a nova tarefa pode continuar a partir do ponto em que foi interrompida anteriormente, enquanto a tarefa anterior fica suspensa até que seja retomada novamente.

```
1  static struct task_entry* round_robin_scheduler(){
2    uint32_t i;
3    int32_t k;
4    uint32_t highestp = 255;
5    struct task_entry * task = NULL;
6    struct task_entry * task_aux = NULL;
7
8    k = queue_count(scheduler_info.krnl_queue);
9
10   for (i = 0; i < k; i++){
```

```
11      task = queue_remhead(scheduler_info.krnl_queue);
12      if (!task){
13        goto error1;
14      }
15      if (queue_addtail(scheduler_info.krnl_queue, task)){
16        goto error2;
17      }
18      if (highestp > task->priority_rem && task->state == TASK_RUNNING){
19        highestp = task->priority_rem;
20        task_aux = task;
21      }
22    }
23
24    if (task_aux){
25      for (i = 0; i < k; i++){
26        task = queue_remhead(scheduler_info.krnl_queue);
27        if (!task){
28          goto error1;
29        }
30        if (queue_addtail(scheduler_info.krnl_queue, task)){
31          goto error2;
32        }
33          /*Atualizacao das prioridades, onde ocorre o envelhecimento.*/
34        if (task != task_aux  && task->state == TASK_RUNNING){
35          task->priority_rem -= task_aux->priority_rem;
36        }
37      }
38
39      task = task_aux;
40      task->priority_rem = task->priority;
41      task->bgjobs++;
42      return task;
43    }
44    return NULL;
45
46  error1:
47    printf("Error removing Task to head.");
48
49  error2:
50    printf("Error adding Task to tail.");
51
52    return NULL;
53  }
```

Código 5.3 – Algoritmo Round-Robin implementado.

A função *round_robin_scheduler()* começa obtendo o número de tarefas em execução no momento e inicializando a variável *highestp* com o valor máximo de prioridade

possível (255), e duas variáveis para tarefa (*task* e *task_aux*), ambas inicialmente nulas.

O escalonador seleciona a tarefa com a maior prioridade entre as tarefas restantes em execução. Isso é feito comparando a prioridade restante (*priority_rem*) de cada tarefa com a variável *highestp*. Se a prioridade restante de uma tarefa for menor e a tarefa estiver em execução, essa tarefa é selecionada para execução e os valores das prioridades são atualizados.

Por fim, a função retorna a tarefa selecionada ou nulo se não houver tarefas em execução. A função também lida com possíveis erros de adicionar ou remover tarefas da fila de tarefas do sistema.

A troca de contexto é uma operação crucial em um sistema operacional multitarefa. Quando uma tarefa está em execução, ela usa os recursos do processador e do sistema, como registradores, memória, pilha e outros. Quando outra tarefa precisa ser executada, o contexto da tarefa em execução é salvo e o contexto da nova tarefa é carregado para o processador.

A função *contextSave()*, Código 5.4, é responsável por salvar o contexto da tarefa que está em execução, para que posteriormente, seja possível restaurar seu estado e continuar sua execução a partir do ponto em que ela foi interrompida.

```
1  void contextSave(){
2    struct task_entry *tasktosave;
3
4    tasktosave = task_in_execution;
5
6    if (!is_task_executing){
7      return;
8    }
9
10   gpr_context_save(tasktosave->gpr);
11
12   tasktosave->cp0_registers[0] = read_csr(mstatus);
13   tasktosave->cp0_registers[1] = read_csr(mie);
14   tasktosave->cp0_registers[2] = read_csr(mip);
15   tasktosave->pc = read_csr(mepc);
16   tasktosave->task_sp = readSP();
17 }
```

Código 5.4 – Algoritmo de salvamento de contexto.

Inicialmente, a função obtém o ponteiro para a tarefa que está sendo executada. A partir daí, verifica-se se há realmente uma tarefa em execução. Caso contrário, a função simplesmente retorna.

Se houver uma tarefa em execução, a função *gpr_context_save()* é chamada, responsável por salvar os registradores da tarefa em execução. Essa função recebe como parâmetro o ponteiro para o vetor que armazena o valor dos registradores da tarefa em

execução.

Em seguida, alguns registradores do processador são salvos. O registrador *mstatus*, que armazena o estado do processador, o registrador *mie*, que controla as interrupções externas, e o registrador *mip*, que armazena as interrupções pendentes, são salvos no vetor *cp0_registers* da tarefa em execução.

O valor do registrador de endereço de exceção (*mepc*), que armazena o endereço da próxima instrução a ser executada depois da interrupção, é salvo para que a execução da tarefa retorne no ponto de parada, quando o contexto é restaurado. Por fim, o valor do ponteiro de pilha atual é salvo no campo *task_sp* da tarefa em execução.

Dessa forma, a função *contextSave()* é responsável por salvar o estado completo da tarefa que está em execução, para que posteriormente seja possível restaurar seu estado e continuar a execução a partir do ponto em que foi interrompida.

A função *contextRestore()*, Código 5.5, é responsável por restaurar o contexto da tarefa que será executada após uma troca de contexto. Em outras palavras, ela é responsável por carregar os valores salvos da tarefa que foi pausada e configurar o processador para executar a próxima tarefa.

```
1  void contextRestore(){
2
3    struct task_entry *task = task_in_execution;
4
5    if(!task){
6      config_idle_cpu();
7      return;
8    }
9
10   if(task->init){
11     task->init = 0;
12
13     setSP((uint32_t)task->task_sp);
14   }else{
15     write_csr(mstatus, task->cp0_registers[0]);
16     write_csr(mie, task->cp0_registers[1]);
17     write_csr(mip, task->cp0_registers[2]);
18
19     setSP((uint32_t)task->task_sp);
20     gpr_context_restore(task->gpr);
21   }
22
23   pmp_controller();
24   setEPC(task->pc);
25
26 }
```

Código 5.5 – Algoritmo de restauração de contexto.

O primeiro passo é verificar se há tarefas prontas para serem executadas. Se não houver, o processador é colocado em modo ocioso. Em seguida, é verificado se a tarefa a ser executada já foi inicializada. Se sim, significa que ela já tem um ponteiro de pilha associado e não é necessário realizar a inicialização novamente. Se não, o ponteiro de pilha da tarefa é definido usando o valor salvo anteriormente. Depois, os registradores *mstatus*, *mie* e *mip* do processador são configurados com os valores salvos da tarefa pausada. Esses registradores controlam aspectos como interrupções e privilégios do processador. Em seguida, o ponteiro de pilha é definido novamente com o valor salvo da tarefa pausada. E finalmente, a função *gpr_context_restore()* é chamada para restaurar o estado dos registradores gerais da CPU a partir dos valores salvos da tarefa pausada.

Por fim, a função *pmp_controller()* é chamada para configurar as permissões de acesso à memória para a tarefa que será executada, isolando-a, e a função *setEPC()* é chamada para definir o valor do registrador *mepc* com o endereço da próxima instrução a ser executada.

5.3 *SYSCALLS*

Syscall é uma forma das tarefas interagirem com o kernel. Elas são rotinas que permitem que as tarefas possam solicitar serviços ao sistema operacional. No RTOS implementado, as *Syscalls* são registradas durante a inicialização do software, em funções específicas para cada serviço. Para registrar uma *Syscall*, é necessário um código que a identifique, uma função que implemente o serviço desejado e uma tabela de *Syscalls*, representada na Tabela 1, que mapeia o código para a função correspondente.

Núm.	**Nome**	**Funcionamento**
0	SCALL_GET_MTIMER_VALUE	Obtém o valor do registro do temporizador.
1	SCALL_MUTEX_INIT	Inicializa um mutex.
2	SCALL_MUTEX_LOCK	Bloqueia um mutex.
3	SCALL_MUTEX_TRYLOCK	Tenta bloquear um mutex sem esperar.
4	SCALL_MUTEX_UNLOCK	Desbloqueia um mutex.
5	SCALL_IPC_INIT	Inicializa uma região de comunicação IPC.
6	SCALL_IPC_WRITE	Escreve em uma região de comunicação IPC.
7	SCALL_IPC_READ	Lê de uma região de comunicação IPC.

Tabela 1 – Syscalls do RTOS

Cada serviço realiza a chamada *register_syscall()* (Código 5.6) durante a inicialização do sistema. Ela recebe como parâmetros um ponteiro para a função que será executada

e o código que identifica essa *Syscall*. A função verifica se o código passado está dentro dos limites permitidos pela tabela e se já não foi registrada anteriormente. Se o código for válido e a *Syscall* não tiver sido registrada, a função é registrada na tabela.

As *Syscalls* são chamadas pelas tarefas através de uma macro, como a exemplificada no Código 5.7 que coloca o número da *Syscall* no registrador a0 e chama a instrução *ecall*, que gera uma interrupção. Essa interrupção é tratada pelo kernel, que identifica a *Syscall* a ser executada através do código passado pelo registrador a0 e chama a função correspondente, passando os argumentos necessários e retornando o resultado para o registrador correto.

```
int32_t register_syscall(syscall_t* sys, uint32_t code){
    if (code > SCALL_TABLE_SIZE-1 || code < 0){
        return SCALL_CODE_INVALID;
    }

    /* Syscall code already registered. */
    if(syscall_table[code]){
        return SCALL_CODE_USED;
    }

    /* Register the syscall */
    syscall_table[code] = sys;

    return code;
}
```

Código 5.6 – Algoritmo de registro das *Syscalls*.

```
#define get_mtimer_value() ({ int32_t __ret; \
asm volatile (                               \
"li a7, %1 \n\
 ecall \n\
 move %0, a0 " \
 : "=r" (__ret) : "I" (SCALL_GET_MTIMER_VALUE) : "a0", "a7"); \
 __ret; })
```

Código 5.7 – Exemplo de macro da *Syscall* do timer.

5.3.1 *Timer*

O *Timer* é um componente importante em um sistema operacional e é utilizado para realizar a contagem do tempo de execução de um processo ou para programar a execução de uma tarefa em um tempo específico. A *Syscall get_mtimer()* (Código 5.8) é uma chamada de sistema responsável por retornar o valor atual do registrador *mtime*, que contém o valor atual do tempo do sistema. O valor do registrador é retornado para o processo que fez a chamada de sistema através do registrador a0.

Em uma tarefa, o cálculo da passagem de tempo é feita por duas funções, presentes no Código 5.9. A função *wait_time()* usa o valor retornado pela *Syscall* para calcular o tempo de espera até que a função retorne 1, o que indica que o tempo de espera foi atingido. Por fim, a função *mdelay()* chama a função *wait_time()* para aguardar a quantidade de tempo especificada em milissegundos.

```
static void get_mtimer(){
  uint32_t mtime = MTIME;
  MoveToPreviousGuestGPR(REG_A0, mtime);
}
```

Código 5.8 – *Syscall* de leitura do timer.

```
uint32_t wait_time(uint32_t old_time, uint32_t ms_delay){
  uint32_t diff_time;
  uint32_t now = get_mtimer_value();

  if (now >= old_time)
    diff_time = now - old_time;
  else
    diff_time = 0xffffffffffffffff - (old_time - now);

  if(diff_time > (ms_delay * MILISECOND)){
    return 1;
  }

  return 0;
}

void mdelay(uint32_t msec){
    uint32_t now = get_mtimer_value();

    while(!wait_time(now, msec));

}
```

Código 5.9 – Funções de cálculo da passagem de tempo.

5.3.2 Mutex

Outra funcionalidade implementada é um mecanismo de exclusão mútua que pode ser usado para sincronizar o acesso a recursos compartilhados entre processos em um sistema operacional. O mutex é uma técnica de sincronização que permite que apenas uma tarefa tenha acesso exclusivo a um recurso compartilhado em um determinado momento, evitando que múltiplas tarefas acessem o recurso simultaneamente e causem condições de corrida ou outras situações indesejadas.

A biblioteca do mutex implementa quatro chamadas de sistema: *Mutex_Init*, *Mutex_Lock*, *Mutex_TryLock* e *Mutex_Unlock*. A chamada de sistema *Mutex_Init* é usada para inicializar uma variável de mutex. A chamada de sistema *Mutex_Lock* é usada para bloquear um mutex. Se o mutex já estiver bloqueado, a tarefa que chamou *Mutex_Lock* será bloqueada e adicionada a uma lista de tarefas em espera. A chamada de sistema *Mutex_TryLock* tenta bloquear um mutex, mas não bloqueia a tarefa se o mutex já estiver bloqueado, em vez disso, retorna um valor indicando se o bloqueio foi bem-sucedido. A chamada de sistema *Mutex_Unlock* é usada para desbloquear um mutex. Se houver tarefas em espera na lista de espera do mutex, a primeira tarefa na lista será acordada e colocada de volta na fila de tarefas prontas para ser executada.

A função *Mutex_Init* é responsável por inicializar um objeto de mutex, que é uma estrutura que controla o acesso concorrente. Na implementação apresentada no Código 5.10, a função *Mutex_Init* é chamada quando a tarefa precisa criar um mutex. Ela obtém um ponteiro para o mutex que foi associado à tarefa, e em seguida inicializa os campos da estrutura *Mutex_t*.

```
1  static void Mutex_Init() {
2
3      Mutex_t* mutex = (Mutex_t*)task_in_execution->mutex;
4
5      mutex->locked = 0;
6      mutex->num_waiting = 0;
7
8      for (uint8_t i = 0; i < MAX_TASKS; i++) {
9
10             mutex->waiting_tasks[i] = 0;
11     }
12
13 }
```

Código 5.10 – Inicialização de um Mutex.

A função *Mutex_Lock()*(Código 5.11) é responsável por tentar bloquear um mutex. Essa função recebe o ID da tarefa cujo mutex foi iniciado através do registrador A0.

O primeiro passo da função é obter o ponteiro para a estrutura *Mutex_t* correspondente ao mutex a ser bloqueado, a partir do vetor de tarefas do RTOS. Em seguida, verifica se o mutex já está bloqueado ou não.

Se o mutex já estiver bloqueado, isso significa que outra tarefa já está usando o recurso protegido pelo mutex. Nesse caso, a tarefa atual deve ser adicionada à lista de espera do mutex e o estado da tarefa passa para bloqueado. Por outro lado, se o mutex ainda não estiver bloqueado , a tarefa atual pode adquiri-lo imediatamente, definindo a variável de bloqueio como 1, bloqueando o mutex a partir desse momento. Note que a implementação atual não lida com prioridades ou inversão de prioridades, ou seja, a lista de espera não é ordenada por prioridade.

```
static void Mutex_Init() {

    Mutex_t* mutex = (Mutex_t*)task_in_execution->mutex;

    mutex->locked = 0;
    mutex->num_waiting = 0;

    for (uint8_t i = 0; i < MAX_TASKS; i++) {

            mutex->waiting_tasks[i] = 0;
    }

}
```

Código 5.11 – Função *Mutex_Lock*.

A função Mutex_TryLock()(Código 5.12) é responsável por tentar adquirir um mutex sem bloquear a tarefa que a está solicitando. Se o mutex estiver disponível, a função irá bloquear o mutex e retornar um valor 1, indicando sucesso na operação. Caso contrário, a função irá simplesmente retornar um valor 0, indicando que o mutex não pode ser bloqueado no momento e que a tarefa deve tentar novamente mais tarde.

A função recebe como argumento o ID da tarefa que o mutex está localizado. Em seguida, a função obtém o ponteiro para o mutex correspondente. A partir do ponteiro para o mutex, a função verifica se o mutex está disponível, verificando se a variável *locked* do mutex é igual a 0. Se o mutex estiver disponível, a função bloqueia o mutex, alterando o valor de *locked* para 1 e retorna um valor 1 indicando sucesso. Caso contrário, a função retorna um valor 0 indicando que o mutex não pode ser bloqueado no momento. Por fim, a função utiliza a função *MoveToPreviousGuestGPR()* para retornar o valor de sucesso ou falha para a tarefa que a chamou pelo registrador a0.

```
static void Mutex_Lock() {

    uint32_t id = MoveFromPreviousGuestGPR(REG_A0);

    Mutex_t* mutex = (Mutex_t*)vector_tasks[id].mutex;

    if (mutex->locked == 1) {
        mutex->waiting_tasks[mutex->num_waiting] = task_in_execution->id;
        mutex->num_waiting++;

       task_in_execution->state = TASK_BLOCKED;
    } else {
        mutex->locked = 1;
    }
```

Código 5.12 – Função *Mutex_Trylock*.

A função *Mutex_Unlock()*(Código 5.13) é responsável por desbloquear o mutex previamente bloqueado com a função *Mutex_Lock*. Ela recebe como parâmetro o identificador do mutex a ser desbloqueado. Primeiramente, a função verifica se existem tarefas esperando pelo mutex. Se existirem, a primeira tarefa da lista de espera é acordada e marcada como "em execução". Para isso, é feita a remoção da tarefa da lista de espera e a atualização do seu estado. Caso não existam tarefas esperando pelo mutex, ele é simplesmente desbloqueado.

```
static void Mutex_Unlock() {

    uint32_t id = MoveFromPreviousGuestGPR(REG_A0);

    Mutex_t* mutex = (Mutex_t*)vector_tasks[id].mutex;

    if (mutex->num_waiting > 0) {

        // There are tasks waiting for the mutex, so wake up the first one
        uint32_t task_id = mutex->waiting_tasks[0];
        // Remove the task from the waiting list
        mutex->num_waiting--;
        for (uint32_t i = 0; i < mutex->num_waiting; i++){
            mutex->waiting_tasks[i] = mutex->waiting_tasks[i+1];
        }

        vector_tasks[task_id].state = TASK_RUNNING;

    } else {
        // No tasks are waiting for the mutex, so unlock it
        mutex->locked = 0;
    }

}
```

Código 5.13 – Função *Mutex_Unlock*.

5.3.3 Comunicação

A comunicação entre tarefas é uma funcionalidade essencial para um sistema operacional multitarefa. Ela permite que as tarefas troquem informações entre si, compartilhando recursos e trabalhando em conjunto para realizar determinadas tarefas.

Para possibilitar essa comunicação, é necessário implementar mecanismos que permitam o acesso controlado aos recursos compartilhados pelas tarefas, como o mutex. Esses mecanismos devem garantir a integridade dos dados e evitar conflitos de acesso que possam levar a erros ou a condições de corrida.

Um dos mecanismos mais comuns para permitir a comunicação entre tarefas é o IPC. Nesse modelo, as tarefas enviam e leem mensagens através do Kernel, permitindo que compartilhem informação entre si.

A função *IPC_Write*(Código 5.14) é responsável por escrever um dado em uma região da TCB identificada pelo ID da tarefa, sendo que cada tarefa possui um inteiro de 32 bits reservado para comunicação.

```
1 static void IPC_Write() {
2
3     uint32_t task_id = (uint32_t)MoveFromPreviousGuestGPR(REG_A0);
4     uint32_t data = (uint32_t)MoveFromPreviousGuestGPR(REG_A1);
5
6     vector_tasks[task_id].ipc = data;
7
8 }
```

Código 5.14 – Função *IPC_Write*.

A função recebe dois argumentos: *task_id* e *data*. O *task_id* é um inteiro que representa o ID da tarefa cuja região de comunicação IPC deve receber a mensagem, enquanto o *data* é um inteiro que representa o dado que será escrito.

A primeira operação realizada pela função é obter os valores dos registradores separados para os parâmetros das *Syscalls*. O valor do dado a ser escrito é armazenado na região da tarefa correspondente através do vetor de tarefas e seu respectivo índice.

A função *IPC_Read()*(Código 5.15) é responsável por ler o valor armazenado na região da tarefa cujo ID é passado como argumento. A função começa recuperando o ID da tarefa alvo, que foi passado pelo registrador a0 para a variável *task_id*. Em seguida, a função acessa a variável *ipc* da estrutura da tarefa correspondente, obtida a partir do vetor global de tarefas, e armazena o valor lido na variável *mem*.

```
1 static void IPC_Read() {
2
3     uint32_t task_id = (uint32_t)MoveFromPreviousGuestGPR(REG_A0);
4
5     uint32_t mem = vector_tasks[task_id].ipc;
6
7     MoveToPreviousGuestGPR(REG_A0,mem);
8
9 }
```

Código 5.15 – Função *IPC_Read*.

Por fim, o valor lido é movido para o registrador a0 usando a função *MoveToPreviousGuestGPR()*, para que possa ser acessado pela tarefa que fez a chamada ao sistema.

5.4 PMP

O uso do PMP desenvolvido visa proteger as tarefas em execução, evitando que elas acessem áreas de memória indevidas e, assim, garantir a integridade do sistema como um todo. A configuração do Controlador PMP é realizada pelo kernel do RTOS a cada troca de contexto, permitindo um gerenciamento dinâmico das permissões de acesso à memória. Para isso, é necessário definir os valores apropriados para os registradores *PMPADDR* e *PMPcfg* correspondentes a cada região de memória protegida e para isso foi criado a função *pmp_controller()*, apresentada no Código 5.16, executado na troca de contexto.

```
1  void pmp_controller(){
2
3    write_csr(pmpaddr0,0x80000000>>2);
4    write_csr(pmpaddr1,((uint32_t)(task->pstack))>>2);
5    write_csr(pmpaddr2,((uint32_t)((task->pstack)+(task->stack_size)))>>2);
6    write_csr(pmpaddr3,((uint32_t)(VMCONF[task->id].ram_base))<<10);
7    write_csr(pmpaddr4,((uint32_t)((VMCONF[task->id].ram_base)+(VMCONF[task->
         id].size)))<<10);
8    write_csr(pmpaddr5,0x3fffffff);
9
10   write_csr(pmpcfg0,0x80f080f);
11   write_csr(pmpcfg1,0x80f);
12
13   write_csr(mstatus, (read_csr(mstatus) & MPP_UMODE_MASK));
14
15 }
```

Código 5.16 – Algoritmo do controlador PMP.

A função *pmp_controller()* apresenta dois principais blocos de configuração do PMP: a definição dos endereços protegidos (*PMPADDR*) e a configuração da política de proteção para cada região protegida (*PMPcfg*).

No primeiro bloco, cada registrador *PMPADDR* é definido com um endereço específico da memória física. O registrador *PMPADDR0* define a primeira região que são endereços abaixo de 0x80000000, que é a base de compilação do sistema. Os registradores *PMPADDR1* e *PMPADDR2* define a região da pilha da tarefa, que está alocada junta com a TCB, e dentro da compilação do RTOS. Os registradores *PMPADDR3* e *PMPADDR4* definem a região onde encontra-se a tarefa, que é compilada separadamente, conforme será explicado posteriormente na Seção 5.5. Já o registrador *PMPADDR5* define o último endereço possível, garantindo que todas as outras tarefas estão protegidas.

No segundo bloco, cada registrador *pmpcfg* é definido com a política de proteção para a região de memória protegida pelo registrador pmpaddr correspondente. A política de proteção é definida por bits que contém informações sobre o tipo de região protegida, as permissões de acesso e a política de controle. Por fim, o registrador *mstatus* é configurado,

já que é necessário configurar um bit correspondente a ativação do PMP.

5.5 COMPILAÇÃO DO SISTEMA

Para garantir a separação espacial necessária pelo PMP, é imprescindível conhecer os endereços iniciais e finais de cada região do software e certificar-se de que o kernel e as tarefas estão em regiões distintas. Para atender a esses requisitos, foi preciso compilar as tarefas de maneira separada do kernel e fixar os tamanhos do kernel e das tarefas para possibilitar a união de todos os binários.

O primeiro passo na compilação do sistema consiste em gerar o arquivo *config.h*, que contém as configurações das tarefas. Para facilitar a modificação dessas configurações sem a necessidade de alterar diretamente a biblioteca, o arquivo é gerado a partir de um arquivo *cfg*, cujo exemplo encontra-se no Código 5.17.

```
/* Configuracao Geral do Sistema */
system = {
    size = 16; /* Kb */
};

/* Configuracao das Tarefas */
tasks = (
    {
        app_name = "test-tasks";
        priority = 100;
        size = 4; /* Kb */
    },
    {
        app_name = "test-tasks2";
        priority = 100;
        size = 4; /* Kb */
    }
);
```

Código 5.17 – Exemplo de arquivo *cfg*.

Para transformar o arquivo *cfg* em *config.h*, foi utilizado um código em Python com a biblioteca *genconf*, responsável pela leitura do arquivo *cfg*. Para exemplificar, apresentamos no Código 5.18 o arquivo *config.h* gerado a partir do arquivo *cfg* de exemplo.

```
/* THIS FILE IS AUTOMATICALLY GENERATED. DO NOT MODIFY IT. */
/* See the input xml tasks-cfg/test.cfg */

#ifndef __CONFIG_H
#define __CONFIG_H

#include <task.h>

```

```
#define MILISECOND ((CPU_FREQ/2)/ 1000)
#define UART_SPEED 115200
#define QUANTUM_SCHEDULER_MS 10
#define GUEST_QUANTUM_MS 1

static const struct tasksconf_t const VMCONF[] = {
  {
    app_name: "test-tasks",
    priority: 100,
    ram_base: 0x80004,
    size: 0x1000,
  },
  {
    app_name: "test-tasks2",
    priority: 100,
    ram_base: 0x80005,
    size: 0x1000,
  },
};

/* Virtual Machine names: test-tasks test-tasks2*/
#define NVMACHINES 2

#endif
```

Código 5.18 – Resultado exemplo do arquivo *config.h*.

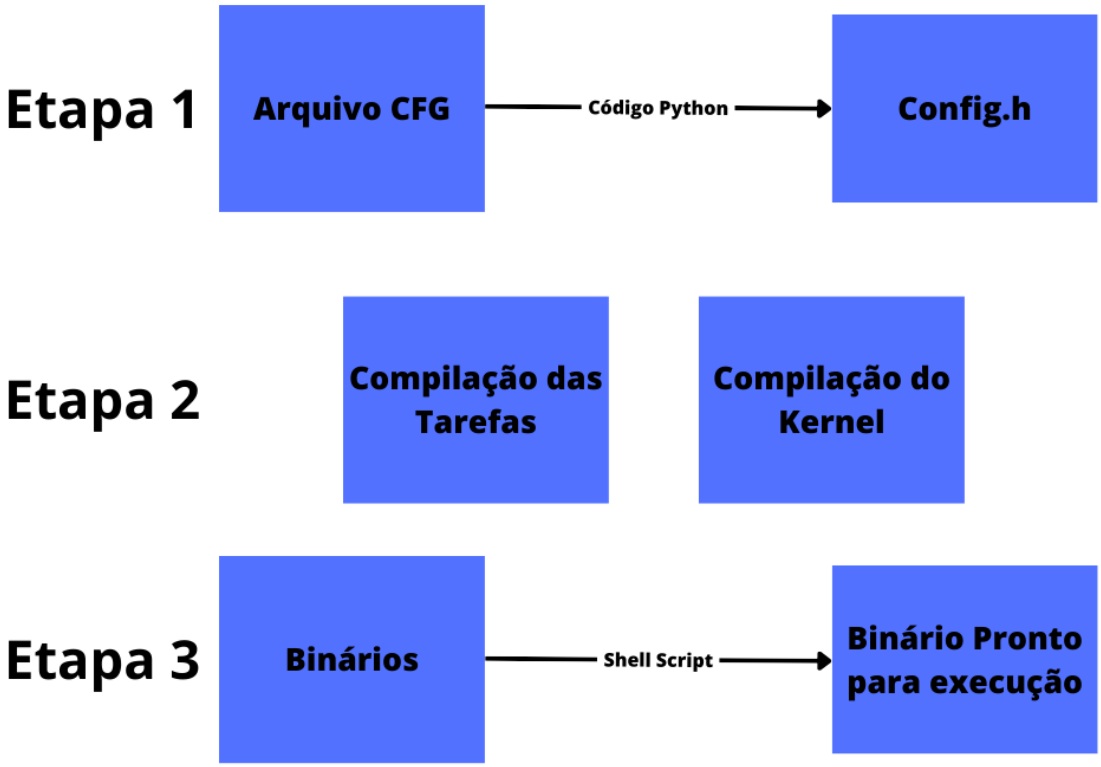
Com base nos tamanhos do kernel e das tarefas definidos, é possível compilá-los separadamente. Para isso, é necessário utilizar um script em shell que realiza a união dos binários em um único arquivo. O processo é automatizado por meio de um *Makefile* que executa o código em Python para gerar o arquivo *config.h*, chamando o compilador para compilar o sistema, o script para juntar os binários e, por último, inicia o QEMU para executar o software. Um resumo das etapas de compilação do sistema pode ser visualizado na Figura 10.

5.6 TESTE DAS FUNCIONALIDADES

O problema do produtor/consumidor é um exemplo clássico de concorrência que envolve duas entidades trabalhando juntas em um processo de compartilhamento de recursos. Na abordagem produtor/consumidor, uma entidade, o produtor, cria dados e os insere em um *buffer* compartilhado, enquanto a outra entidade, o consumidor, retira esses dados do *buffer* e os utiliza.

Para implementar essa abordagem, são necessárias ferramentas que permitam a sincronização e a comunicação entre as entidades envolvidas. O uso de mecanismos de

Figura 10 – Resumo da Compilação do Sistema.



Fonte: Do Autor.

mutex é comum para garantir a exclusão mútua no acesso ao *buffer* compartilhado, evitando condições de corrida.

Além disso, é preciso considerar o tempo de espera entre a produção e o consumo dos dados, o que pode ser tratado utilizando um mecanismo de timer, que permita a implementação de um intervalo de espera definido entre as operações de produção e consumo.

Para testar esses recursos, é possível implementar um sistema produtor/consumidor em que as tarefas de produção e consumo comunicam entre si e utilizam mutex e timer para garantir a exclusão mútua e um intervalo de tempo definido entre as operações. Para isso foi utilizado as duas tarefas seguintes, sendo a do Código 5.19 a produtora e a do Código 5.20.

```
1 int main(){
2
3   uint32_t i = 0;
4
5   IPC_init();
```

```
6    mutex_init();
7    mutex_lock(TASK_ID_PRODUTORA);
8
9    while(1){
10
11         mutex_lock(TASK_ID_CONSUMIDORA);
12
13         mdelay(1000);
14         IPC_write(TASK_ID_CONSUMIDORA,(uint32_t)i++);
15
16         mutex_unlock(TASK_ID_PRODUTORA);
17
18           }
19
20 }
```

Código 5.19 – Tarefa Produtora.

```
1  int main(){
2
3    uint32_t value = 0;
4
5    mutex_init();
6
7    while(1){
8
9                mutex_lock(TASK_ID_PRODUTORA);
10
11               value = (uint32_t)IPC_read(TASK_ID_CONSUMIDORA);
12         printf("Valor(%d)\n",value);
13
14         mutex_unlock(TASK_ID_CONSUMIDORA);
15   }
16
17 }
```

Código 5.20 – Tarefa Consumidora.

A tarefa produtora implementa a lógica do produtor no problema clássico do produtor/consumidor. Ela é responsável por produzir valores enviá-los a tarefa consumidora.

Para garantir a sincronização entre as duas tarefas, a tarefa produtora utiliza um mutex para controlar a sincronização entre o envio das mensagens. Antes de enviar um valor, ela adquire o mutex correspondente e o libera após a escrita.

Além disso, a tarefa produtora utiliza o timer para controlar o intervalo de tempo entre a produção de um valor e outro. Ela usa a função *mdelay* para aguardar 1 segundo antes de produzir um novo valor.

A tarefa consumidora é responsável por ler o valor produzido pela tarefa produtora. Ela também utiliza um mutex para garantir a exclusão mútua ao receber a mensagem.

O código da tarefa começa com a inicialização do mutex e, em seguida, a tarefa entra em um laço infinito. Dentro do laço, a tarefa adquire o mutex com a função *mutex_lock()*. Em seguida, a tarefa lê o valor enviado usando a função *IPC_read()*. Por fim, a tarefa libera o mutex com a função *mutex_unlock()*.

Assim, a tarefa consumidora executa continuamente, lendo e imprimindo os valores produzidos pela tarefa produtora a cada iteração do laço de repetição. Caso a tarefa produtora ainda não tenha produzido um novo valor, a tarefa consumidora espera pelo próximo valor produzido.

Como saída, obteve-se o Código 5.21, o que indica que a tarefa consumidora conseguiu ler os valores produzidos pela tarefa produtora corretamente. O valor inicial lido pela tarefa consumidora foi 0, e a partir daí foram lidos os valores 1, 2, 3, 4, 5, 6, 7 e 8, respectivamente.

```
1 Valor(0)
2 Valor(1)
3 Valor(2)
4 Valor(3)
5 Valor(4)
6 Valor(5)
7 Valor(6)
8 Valor(7)
9 Valor(8)
```

Código 5.21 – Resultado da execução das tarefas.

O resultado indica que as funcionalidades testadas (timer, mutex e IPC) estão funcionando corretamente, permitindo a comunicação entre as tarefas produtora e consumidora.

5.7 TESTE DE SEGURANÇA

Para testar o funcionamento do PMP, foi criado um tarefa simples, encontrada no Código 5.22, que tenta ler o conteúdo de uma área de memória fora dos limites permitidos. A tarefa em questão foi programada para tentar ler o valor de um endereço de memória que está fora da sua permissão de acesso, definida pelo PMP. Ao tentar executar essa operação, o RISC-V identificou que houve uma tentativa de acesso proibido e gerou uma exceção do tipo 5, que indica justamente esse tipo de erro.

Ao analisar a saída do programa (Código 5.23), é possível verificar que a tarefa executou uma vez com sucesso, imprimindo corretamente o valor da variável "i". Então, houve uma tentativa de acesso à área proibida e a exceção foi gerada. Quando verificado na lista de instruções qual é a instrução salva no endereço 0x800058be é uma leitura dentro

do *printf*, sendo exatamente a instrução esperada para gerar a exceção. Isso comprova que o PMP está funcionando corretamente, impedindo o acesso a áreas de memória não autorizadas.

```
int main(){

  uint32_t i = 0;
  uint32_t* p = 0x80000000;

  while(1){

    mdelay(1000);

    printf("Task 2(%d)\n",i);

    printf("Invasao(%x)\n",*p);
  }

}
```

Código 5.22 – Tarefa de teste do PMP.

```
Task 1(0)
Task 2(0)
Error 5 mepc:0x800058be mstatus:0x80
Task 1(0)
Task 1(0)
Task 1(0)
```

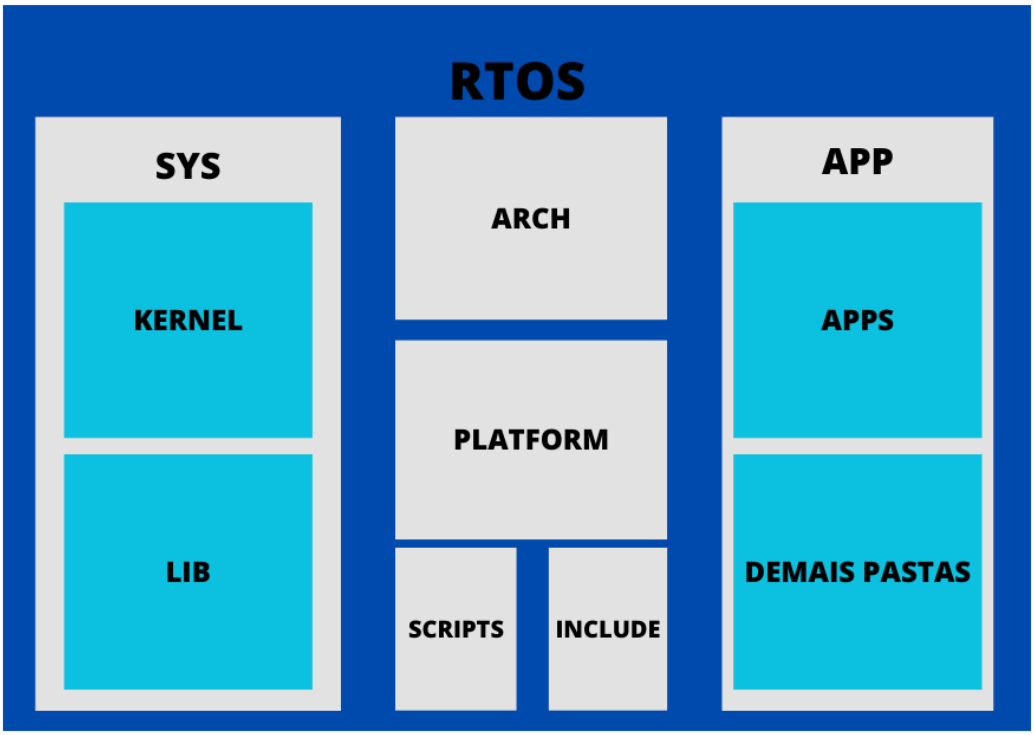
Código 5.23 – Saída teste do PMP.

5.8 ACESSO AO CÓDIGO-FONTE

Este capítulo aborda a organização do código-fonte do sistema. O RTOS aqui apresentado é um projeto de código aberto, disponível no GitHub(MOURA, 2023). O código-fonte do RTOS é organizado em uma estrutura de pastas bem definida, a fim de facilitar o desenvolvimento, manutenção e expansão do sistema. A estrutura é composta pelas pastas SYS, ARCH, PLATFORM, SCRIPT, INCLUDE e APP. Cada uma dessas pastas possui uma finalidade específica. Um diagrama do sistema pode ser visualizado na Figura 11.

O diretório SYS é o diretório principal do sistema e contém dois subdiretórios: KERNEL e LIB. O subdiretório KERNEL é responsável por armazenar os arquivos relacionados ao núcleo do RTOS. Já o subdiretório LIB é destinado às bibliotecas C que são utilizadas pelo sistema. O diretório ARCH abriga os arquivos relacionados às *Syscalls* e configurações específicas do hardware. Nesse diretório, são encontrados os componentes

Figura 11 – Diagrama da organização do código-fonte.



Fonte: Do Autor.

necessários para a interface entre o sistema operacional e o hardware subjacente. A organização dos arquivos nesse diretório permite a portabilidade do RTOS para diferentes arquiteturas de hardware em projetos futuros. O diretório PLATFORM contém os arquivos relacionados à compilação do sistema, onde também são definidas as configurações. O diretório SCRIPT contém um único arquivo que é utilizado durante o processo de compilação do RTOS. Esse arquivo de script auxilia na geração dos binários finais do sistema. O diretório APP é dedicado aos arquivos relacionados às tarefas executadas pelo RTOS. Dentro desse diretório, encontramos a subpasta APPS, que contém os códigos-fonte das tarefas específicas do sistema. Assim como os diretórios anteriores, a pasta APP também possui subdiretórios semelhantes aos do kernel (PLATFORM, ARCH, LIB). O diretório SCRIPT contém um único arquivo que é utilizado durante o processo de compilação do RTOS. O diretório INCLUDE é reservado para a inclusão de bibliotecas comuns ao kernel e às tarefas do RTOS, sendo que só contém a definição das *Syscalls*.

6 CONSIDERAÇÕES FINAIS

Este trabalho teve como objetivo implementar um RTOS que execute em processadores RISC-V e garanta a separação entre as tarefas como fator de segurança. O objetivo geral foi atingido através da implementação do escalonador de tarefas Round-Robin com prioridades e envelhecimento, bem como através do recurso PMP fornecido pelos processadores RISC-V.

Ao longo do desenvolvimento, foram utilizados recursos teóricos sobre SOs, RTOSs, Sistemas Embarcados, Segurança de Software e arquitetura RISC-V, para garantir que o RTOS atendesse às necessidades propostas. Além disso, foram realizados testes do RTOS com o emulador QEMU para verificar o correto funcionamento das funcionalidades desenvolvidas.

O desenvolvimento do escalonador foi uma etapa fundamental do projeto, já que é responsável por gerenciar o tempo de execução das tarefas e definir a ordem em que as tarefas serão executadas e, por isso, é essencial para garantir resposta em tempo real. Durante o desenvolvimento deste trabalho, foram implementados três recursos essenciais para o sistema operacional proposto: *timer*, sistema de mutex e a comunicação por IPC.

O *timer* é responsável por medir o tempo de execução das tarefas e fornecer informações precisas sobre o tempo de espera. A implementação do *timer* foi feita utilizando um contador de tempo baseado no hardware.

O sistema de mutex é responsável por garantir a sincronização de acesso aos recursos compartilhados entre as tarefas. A implementação do sistema de mutex foi feita utilizando uma estrutura de dados que armazena informações sobre as tarefas que estão aguardando para acessar um recurso compartilhado.

A comunicação por IPC é um mecanismo que permite que as tarefas compartilhem informações entre si. A implementação da comunicação foi feita utilizando uma estrutura de dados que armazena informações compartilhadas entre as tarefas. As tarefas podem enviar e receber mensagens para trocar informações.

Outro ponto importante deste trabalho foi a realização de testes de segurança para verificar a separação entre as tarefas, garantindo que elas não possam interferir ou acessar indevidamente outras tarefas executando no mesmo sistema. Essa medida é fundamental para garantir a integridade do sistema como um todo e prevenir ataques maliciosos.

Desta forma, pode-se afirmar que o objetivo proposto neste trabalho foi alcançado com sucesso, uma vez que o RTOS desenvolvido atendeu às necessidades propostas e demonstrou-se eficiente em garantir a separação entre as tarefas. Essa implementação pode ser utilizada em diferentes sistemas embarcados, como em veículos autônomos, sistemas de automação industrial e equipamentos médicos, onde a segurança e confiabilidade são requisitos essenciais.

É importante destacar que este trabalho contribui para o avanço do conhecimento

na área de sistemas operacionais em tempo real e em processadores RISC-V, fornecendo uma implementação de RTOS com separação espacial entre tarefas que pode ser utilizada em diferentes aplicações e contextos.

O presente trabalho possui potencial para o desenvolvimento de mais funcionalidades como novas chamadas de sistema para comunicação com periféricos, conexão com a Internet, dentre outros. Também, é possível pesquisar outros recursos de hardware em outras arquiteturas que possam ser utilizadas para gerar separação espacial, tornando possível realizar o porte para outros processadores.Ainda, há a possibilidade de criar novos recursos de segurança para proteger os dados do usuário e garantir a privacidade. Algumas possíveis melhorias incluem criptografia de dados e autenticação de usuários.

REFERÊNCIAS

AKYILDIZ, Ian; SU, Weilian; SANKARASUBRAMANIAM, Yogesh; CAYIRCI, Erdal. A survey on sensor networks. **IEEE Communications Magazine**, IEEE, v. 40, n. 8, p. 102–114, 2002.

BUTENHOF, David R. **Programming with POSIX Threads**. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1997.

CHEANG, Kevin; RASMUSSEN, Cameron; LEE, Dayeol; KOHLBRENNER, David W; ASANOVI'C, Krste; SESHIA, Sanjit A. Verifying RISC-V Physical Memory Protection. *In*: IEEE. 2021 IEEE/ACM International Conference on Computer-Aided Design (ICCAD). [*S.l.*: *s.n.*], 2021. P. 1–9.

DAVIS, Bryon. **Embedded systems: ARM programming and optimization**. [*S.l.*]: Academic Press, 2017.

FREERTOS - Task Creation Hook. [*S.l.*: *s.n.*], 2018. https://www.freertos.org/implementation/a00005.html.

HAN, Seongjin; LEE, Sangho; LEE, Kyungtae; KIM, Taeho. A Temporal Separation-Based Scheduling Mechanism for Multicore Processors. **IEEE Transactions on Parallel and Distributed Systems**, v. 28, n. 2, p. 373–385, 2017.

LEE, Edward Ashford; SESHIA, Sanjit Arunkumar. **Introduction to Embedded Systems: A Cyber-Physical Systems Approach**. Berkeley, CA: LeeSeshia.org, 2015.

LIU, Chung Laung; LAYLAND, James W. Scheduling algorithms for multiprogramming in a hard-real-time environment. **Journal of the ACM (JACM)**, ACM, v. 20, n. 1, p. 46–61, 1973.

MAZIDI, Muhammad. **The Art of Assembly Language**. [*S.l.*]: MicroDigitalEd, 2015.

MAZIDI, Muhammad Ali. **The Art of Assembly Language**. [*S.l.*]: MicroDigitalEd, 2015.

MOURA, Emanuel Lucas Rodrigues. **RISC-V RTOS**. [*S.l.*: *s.n.*], 2023. https://github.com/emanuellucas2/riscv-rtos. Repositório do GitHub.

OLIVEIRA, Lucas S.; NOGUEIRA, Thiago H.; FREITAS, Henrique C. Port do Sistema Operacional Nanvix para Arquitetura RISC-V Plataforma PULP. *In*: ANAIS do Workshop de Iniciação Científica-Simpósio em Sistemas Computacionais de Alto Desempenho (WSCAD). Florianópolis: Sociedade Brasileira de Computação, 2022. P. 41–48.

OLIVEIRA, Rômulo. **Sistemas Operacionais**. 2. ed. São Paulo: Novatec, 2018. P. 464.

OLIVEIRA, Rômulo Silva. **Fundamentos dos Sistemas de Tempo Real**. 2. ed. [S.l.]: Edição do Kindle, 2017. P. 15. e-book.

PRADO, Sergio. **Diminuindo o tamanho do Kernel Linux**. [*S.l.*: *s.n.*], 2019. https://sergioprado.org/diminuindo-o-tamanho-do-kernel-linux/. Acesso em: 11 maio 2023.

PROJECT, GNU. **GNU Operating System - GNU Project - Free Software Foundation (FSF)**. 2023. Disponível em: https://www.gnu.org/gnu/gnu-history.html.

RISC-V FOUNDATION. **RISC-V Overview**. [*S.l.*: *s.n.*], 2022. https://riscv.org/what-is-risc-v/overview/. Acessado em: 11 de maio de 2023.

RISC-V FOUNDATION. **RISC-V Privileged Architecture Specification**. [*S.l.*], 2021. https://riscv.org/specifications/privileged-isa/.

SADEGHI, Ahmad-Reza; WACHSMANN, Christian; WAIDNER, Michael. Security and privacy challenges in industrial internet of things. **Proceedings of the IEEE**, IEEE, v. 107, n. 8, p. 1596–1611, 2019.

SILBERSCHATZ, Abraham; GALVIN, Peter B.; GAGNE, Greg. **Sistemas Operacionais: Conceitos e Aplicações**. 8. ed. Porto Alegre: Bookman, 2010. P. 848.

SOUZA, Evaldo; CARVALHO, Tereza Cristina Melo de; BARROS, Mateus de Souza; VASCONCELOS, Renata; BEZERRA, Eduardo; FERNANDES, Daniel. Segurança de Software em Sistemas Embarcados: Ataques e Defesas. Versão portuguesa. **Revista de Informática Teórica e Aplicada**, v. 22, n. 3, p. 437–460, 2015. ISSN 2175-2745.

SRIDHARAN, S.; GARG, S. A review of security issues and solutions in cloud computing. **Journal of Network and Computer Applications**, v. 68, p. 1–25, 2016.

STROSS, R. **Revolution OS: Como o software livre e o Linux reinventaram a tecnologia e desafiaram a Microsoft**. 1. ed. [*S.l.*]: [S.l.], 2001. P. 319.

SUPPORTED DEVICES. [*S.l.*]: FreeRTOS org, 2018. https://freertos.org/RTOS_ports.html.

TANENBAUM, Andrew S.; AUSTIN, Todd. **Organização Estruturada de Computadores**. 6. ed. São Paulo: Pearson Prentice Hall, 2013. P. 739.

TANENBAUM, Andrew S.; BOS, Herbert. **Sistemas Operacionais Modernos**. 4. ed. São Paulo: Prentice Hall, 2015. P. 1112.

TAUBENFELD, Gadi. **Synchronization Algorithms and Concurrent Programming**. Upper Saddle River, NJ: Pearson Education, 2006.

VALAVANIS, Kimon P; VACHTSEVANOS, George J. **Handbook of Unmanned Aerial Vehicles**. New York, NY: Springer Science & Business Media, 2014.

VIEIRA, Marcelo de Oliveira; SOUZA, José Neuman de; ARAÚJO, Rubens Takashi de; GOMES, Rafael Barreira. Internet das Coisas: Visão Geral, Desafios e Contribuições. Versão portuguesa. **Revista de Engenharia e Pesquisa Aplicada**, v. 1, n. 1, p. 73–85, 2014. ISSN 2317-4606.

ZHONG, Yuan; ZENG, Hong. A task priority scheduling algorithm based on round-robin. *In*: IEEE. 2012 Third International Conference on Digital Manufacturing & Automation. [*S.l.*: *s.n.*], 2012. P. 376–379.