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

Gabriel Dinse

Avaliação de Desempenho de Abordagens do Linux Embarcado para Aplicações de Tempo Real

Blumenau
2021

Gabriel Dinse

Avaliação de Desempenho de Abordagens do Linux Embarcado para Aplicações de Tempo Real

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

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

Blumenau
2021

Gabriel Dinse

Avaliação de Desempenho de Abordagens do Linux Embarcado para Aplicações de Tempo Real

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

Comissão Examinadora

Documento assinado digitalmente
Carlos Roberto Moratelli
Data: 19/05/2021 09:00:14-0300
CPF: 035.999.369-94
Verifique as assinaturas em https://v.ufsc.br

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

Documento assinado digitalmente
Mauri Ferrandin
Data: 18/05/2021 22:47:11-0300
CPF: 018.580.869-73
Verifique as assinaturas em https://v.ufsc.br

Prof. Dr. Mauri Ferrandin
Universidade Federal de Santa Catarina

Documento assinado digitalmente
Ciro Andre Pitz
Data: 19/05/2021 06:27:19-0300
CPF: 055.728.089-38
Verifique as assinaturas em https://v.ufsc.br

Prof. Dr. Ciro André Pitz
Universidade Federal de Santa Catarina

Blumenau, 19 de maio de 2021

Dedico este trabalho a Deus, minha família, professores, amigos e a todos que me auxiliaram de alguma forma na conclusão deste trabalho.

Agradecimentos

Agradeço primeiramente a Deus, companhia sem a qual eu não teria superado nem uma pequena parcela dos desafios externos e internos pelos quais passei. Sei também que Dele vêm quaisquer capacidades ou habilidades que eu tenho, por mais simples que sejam. Obrigado por estar comigo nos momentos mais difíceis.

Agradeço também ao meu pai Gilberto e minha mãe Vilma, os quais são um exemplo de casal que tenho o prazer de mencionar sempre que tenho chance. Nunca faltaram com amor, atenção e disposição de me ajudar, pelo contrário, sempre sobrou até mesmo para distribuir e "adotar" a outros como se fossem filhos como eu. Que eu nunca deixe morrer tudo o que vocês plantaram em mim. Agradeço a todos os outros membros da minha família, os que estão perto ou distantes geograficamente.

A todos os meus irmãos que não são de sangue, já que sou único de meus pais, também vai meu agradecimento: João, Pedro, Lucas, Gustavo, Juliano, José, Barreto, Patrick e Victor. Obrigado por confiarem em mim, assim como eu igualmente confio em vocês. Que possamos continuar crescendo como um só, assim como Jesus desejou.

Não poderia também esquecer de todos que conheci na universidade, com os quais desenvolvi uma amizade muito bonita e que tenho certeza que não se restringirá apenas às paredes do edifício da UFSC. Alguns que consigo nomear: Eduard, Martin, Christian Borba, André, Gustavo Link, Ricardo e também Henrique Lange, que conheci em um momento posterior. A companhia de vocês foi maravilhosa. Nunca vou me esquecer do dia que alguns de vocês vieram aqui em casa com uma cuca depois que eu havia trancado o curso por motivos de saúde. Agradeço também a todos os professores. Em especial, agradeço ao Carlos Moratelli, meu orientador nesse trabalho, que dedicou muito do seu tempo para me auxiliar, e o Leonardo Mejia Rincon, pelo qual fui orientado como bolsista e também voluntário por bastante tempo.

Agradeço a todos que posso ter esquecido de mencionar, mas que contribuíram na minha vida e consequentemente neste trabalho. Obrigado.

"Se um homem ainda não descobriu algo pelo qual ele morreria, ele não está apto a viver."

(Martin Luther King Jr.)

Resumo

Os sistemas embarcados têm evoluído ao ponto de serem capazes de fazer uso de sistemas operacionais. Porém, para aplicações de tempo real, que são mais comuns nesse segmento, os sistemas operacionais de propósito geral necessitam de adequações para o atendimento desses requisitos. Este trabalho discorre sobre a execução e análise de testes de desempenho em sistemas Linux com aplicações que operam com rigor de tempo real por meio do tratamento de interrupções. São utilizados o Linux Vanilla, o Real-Time Linux e o Xenomai na configuração de kernel duplo, os quais são executados na placa Raspberry Pi 3. A avaliação é feita em duas etapas, a primeira com os sistemas unicamente tratando as interrupções recebidas, enquanto na segunda etapa é também aplicado estresse computacional no sistema operacional alvo. Os resultados são discutidos e verificados através das latências mínima e máxima, da média das latências e do desvio padrão do conjunto de amostras. É também examinada a taxa de cumprimento dos prazos temporais estabelecidos de antemão.

Palavras-Chave: 1. Linux. 2. Tempo Real. 3. Testes de desempenho. 4. Raspberry Pi 3

Abstract

Embedded systems have evolved to the point of being able to use operating systems. However, for real-time applications, which are more common in this segment, general purpose operating systems need adjustments to meet these requirements. This work discusses about the execution and analysis of performance tests on Linux systems with applications that operate with real-time rigor through the interrupt handling. Linux Vanilla, Real-Time Linux and Xenomai are used in the dual kernel configuration, which run on the Raspberry Pi 3 board. The evaluation is done in two steps, the first with the systems only handling incoming interruptions, while in the second stage, computational stress is also applied to the target operating system. The results are discussed and verified through the minimum and maximum latencies, the average of the latencies and the standard deviation of the sample set. The compliance rate of the time limits established in advance is also examined.

Keywords: 1. Linux. 2. Real time. 3. Performance tests. 4. Raspberry Pi 3

Lista de figuras

Lista de tabelas

Lista de Siglas e Abreviaturas

ABS	*Anti-lock Braking System*
API	*Application Programming Interface*
ARM	*Advanced RISC Machines*
ADEOS	*Adaptive Domain Environment for Operating Systems*
BCM	*Broadcom*
CD	*Compact Disk*
CFS	*Completely Fair Scheduler*
CPU	*Central Processing Unit*
CSI	*Camera Serial Interface*
DHCP	*Dynamic Host Configuration Protocol*
DSI	*Display Serial Interface*
DTS	*Device Tree Source*
FCFS	*First Come, First Served*
FIFO	*First In, First Out*
FREEBSD	*Free Berkeley Software Distribution*
GPIO	*General Purpose Input/Output*
GPL	*GNU GeneralPurpose License*
GPOS	*General Purpose Operating System*
GUI	*Graphical User Interface*
HAL	*Hardware Abstraction Layer*
HDMI	*High-Definition Multimedia Interface*
HD	*Hard Disk*
I-PIPE	*Interrupt-Pipeline*
I2C	*Inter-Integrated Circuit*
IDE	*Integrated Development Environment*
IEEE	*Institute of Electrical and Electronics Engineers*
IPC	*Inter-Process Communication*
ISR	*Interrupt Service Routine*
JVM	*Java Virtual Machine*
LGPL	*GNU Lesser General Purpose License*
MIT	*Massachusetts Institute of Technology*
MUTEX	*Mutual Exclusion*
NPTL	*Native POSIX Threads Library*
OS	*Operating System*
PCB	*Process Control Block*

POSIX	*Portable Operating System Interface*
RAM	*Random-Access Memory*
RR	*Round-Robin*
RTDM	*Real-Time Driver Model*
RTOS	*Real-Time Operating System*
RT	*Real-Time*
SD	*Secure Digital*
SJF	*Shortest Job First*
SPI	*Serial Peripheral Interface*
SRAM	*Static Random-Access Memory*
SSD	*Solid State Drive*
SSH	*Secure Shell Protocol*
SoC	*System on a Chip*
Stdout	*Standard Output*
Syscall	*System Call*
UART	*Universal Asynchronous Receiver-Transmitter*
USB	*Universal Serial Bus*
VFS	*Virtual File System*
VGA	*Video Graphics Array*
Voip	*Voice Over Internet Protocol*
gcc	*GNU Compiler Collection*
glibc	*GNU C Library*
ioctl	*input/output control*

Sumário

1 Introdução

Sistemas de tempo real, na computação, são aqueles que não dependem somente da exatidão do resultado das suas operações lógicas, mas também do tempo em que os resultados são entregues [1]. Tempo real, então, não se refere à velocidade de um sistema, mas sim à previsibilidade e determinismo temporal dele. A importância do tempo real está intimamente associada às consequências que o não cumprimento de prazos podem causar ao ambiente ou ao próprio dispositivo. Esse tipo de sistema é principalmente aplicado quando há um tempo máximo esperado para a execução das suas tarefas ou resposta a eventos, de modo a evitar falhas catastróficas.

Sistema operacional (SO) é um programa responsável por servir de interface padronizada entre uma peça de hardware e o usuário. A partir de um sistema operacional, um usuário pode inicializar outros programas que executam as mais diversas tarefas, sem que seja necessário conhecer detalhes do equipamento físico [2]. Quando pessoas referem-se a um SO, muitas vezes estão falando de sistemas operacionais de propósito geral (GPOS). Esses sistemas são abrangentes para realizar diversos tipos de atividades comuns do usuário, como acessar mídias, escrever documentos, jogar, acessar a internet, etc. O foco desses sistemas é, principalmente, fornecer a melhor experiência ao usuário. A prioridade das tarefas e a parcela que ocupam do processador é ajustada ao longo da execução dos programas pelo próprio sistema [3]. GPOSs também providenciam ferramentas para a programação de novas aplicações e ferramentas, não somente para a plataforma de origem, mas também para outros hardwares e arquiteturas dos mais variados tipos.

A extensão do conceito de tempo real para sistemas operacionais dá origem aos sistemas operacionais de tempo real (RTOS), que apresentam ferramentas para criar e gerenciar tarefas de tempo real. Em RTOSs, o tempo é tratado de forma muito mais explícita do que é visto em sistemas operacionais de propósito geral. A prioridade das tarefas é também bastante clara e sempre estabelecida de antemão [3]. RTOSs têm como alvo principal e popular os sistemas embarcados, onde geralmente necessita-se do emprego de tempo real [4]. Por serem muitas vezes utilizados em hardwares mais limitados que os computadores pessoais, o conjunto de bibliotecas e ferramentas disponíveis nesses sistemas é bastante reduzido quando comparados com GPOSs, que visam abranger mais casos de uso.

O GNU/Linux é um GPOS baseado no Unix e desenvolvido de maneira colaborativa por empresas e programadores, como um projeto de código aberto [5]. Desenvolvido inicialmente como um kernel inspirado no Unix, por Linus Torvalds, e agregado ao projeto GNU [6], de Richard Stallman, o Linux é o sistema operacional de software livre mais bem sucedido de todos dos tempos, amplamente utilizado em diversos âmbitos da computa-

ção. É um sistema simples, multiusuário, bastante consolidado e grandemente escalável, tanto para pequenas aplicações como até mesmo em grandes servidores, e que agrega as ferramentas mais essenciais e populares do Unix [7].

Os sistemas embarcados têm, ao longo do tempo, ser tornado cada vez mais potentes e adquirido componentes eletrônicos que possibilitam que portem sistemas operacionais de propósito geral, como o Linux [8]. Os microcontroladores e plataformas de prototipagem micro-controladas hoje são a forma mais usual de se pensar em sistemas embarcados, mas isso tem dado lugar a hardwares que podem fazer tarefas além de executar instruções gravadas no seu dispositivo de armazenamento e atividades em um laço único de execução. Nesse contexto, o Linux entra como um bom candidato de sistema operacional leve e facilmente modificável, por sua característica de ser de código aberto, para ser empregado em sistemas embarcados, permitindo customizações de acordo com as necessidades do usuário.

Entretanto, a execução de atividades de tempo real que exigem o cumprimento de prazos e constância no atendimento de tarefas ainda é um desafio para o Linux, ainda que possa estar presente em sistemas embarcados. Caso essa barreira possa ser rompida de alguma forma, tem-se em mãos um sistema operacional rico em ferramentas e utilidades, sólido e capaz de gerenciar todos os tipos de atividades, de alta e baixa prioridade, centralizados no mesmo equipamento.

Existem algumas abordagens que se propõem a trazer o tempo real ao Linux. O Real-Time Linux refere-se ao Linux tradicional, chamado *mainline* ou *Vanilla*, com a modificação no escalonador de tarefas, que torna-o totalmente preemptivo, isto é, capaz de tirar um processo de execução para dar lugar a outro a qualquer momento (diferentemente do que acontece no Linux padrão) [9]. Há também o projeto Xenomai, que em uma das suas configurações agrega um segundo kernel, o Cobalt, ao kernel do Linux, que se torna responsável por todas as atividades de tempo real do sistema [10]. Nessa configuração, o Linux é então responsável por atividades que não envolvem tempo real. O próprio Linux tradicional, o Vanilla, também apresenta ferramentas para explicitamente iniciar tarefas de tempo real de alta prioridade sem a necessidade de nenhuma modificação no sistema, porém sem total preempção do kernel.

Neste trabalho, busca-se avaliar as capacidades do Linux como um sistema operacional capaz de operar em tempo real, onde cada uma das opções de SOs escolhidas é executada na placa Raspberry Pi 3. Este projeto procura responder a seguinte pergunta: é possível que um sistema operacional de propósito geral, como o Linux, adquira características de tempo real ao ter sua estrutura de funcionamento ou aplicações modificadas, sendo capaz de executar tarefas com determinismo temporal?

1.1 Objetivos do trabalho

Essa seção destina-se a esclarecer quais são os objetivos geral e específicos sob os quais esse trabalho se constrói.

1.1.1 Objetivo geral

O objetivo deste projeto é realizar testes de desempenho, através da medição de latências no tratamento de interrupções, em sistemas operacionais Linux com aplicações e/ou kernel modificados para atender requisitos de tempo real.

1.1.2 Objetivos específicos

Para que o objetivo geral seja alcançado, é preciso que os seguintes objetivos específicos sejam postos em prática:

1. Desenvolvimento da aplicação para o dispositivo gerador de interrupções.
2. Compilação do Linux com *kernel* Vanilla e escrita do algoritmo de tratamento de interrupções.
3. Compilação do Xenomai com o kernel Cobalt e escrita do algoritmo de tratamento de interrupções.
4. Compilação do Linux com o *patch PREEMPT_RT* (Real-Time Linux).
5. Execução das aplicações e coleta das latências.
6. Execução das aplicações em conjunto com a ferramenta de estresse computacional e coleta das latências.
7. Análise e comparação dos dados obtidos para validação da garantia de atendimento temporal dos sistemas de tempo real.

1.2 Estrutura do documento

Em conjunto com o capítulo corrente, que descreveu o trabalho de forma introdutória, estão presentes neste documento mais cinco capítulos, brevemente expostos a seguir. A fundamentação teórica considerada importante para o entendimento dos capítulos subsequentes está disposta no Capítulo 2. O Capítulo 3 é reservado a introduzir o sistema operacional Linux, bem como a apresentar as abordagens de Linux de tempo real testadas neste projeto. A metodologia adotada para o cumprimento dos objetivos propostos é detalhada no Capítulo 4. Os resultados obtidos, bem como sua avaliação, podem ser vistos no Capítulo 5. No Capítulo 6, por fim, é feita a conclusão do trabalho.

2 Fundamentação Teórica

2.1 Estrutura de um Sistema Operacional de Propósito Geral

Os sistemas operacionais de propósito geral, comumente referidos apenas por sistemas operacionais, representam o software fundamental de computadores, servidores web, *datacenters* e celulares modernos. É a partir do sistema operacional que todas as aplicações e ferramentas do usuário são executadas, fazendo pedidos de recursos computacionais, como memória, processamento e armazenamento a todo o momento. Os sistemas operacionais são responsáveis por gerir com justiça todas as tarefas que concorrem por execução e muitas vezes estão estruturados, mesmo que divergindo em alguns aspectos, sob bases comuns que sistematizam e definem essa complexa comunicação com o usuário final. A seção atual busca, então, tratar de todas essas partes fundamentais dos GPOSs.

2.1.1 Kernel

O kernel, *core*, ou núcleo é responsável por gerenciar todos os recursos de um sistema operacional, sejam software ou hardware. Ao se falar de hardware, é o kernel que disponibiliza o acesso a todos os dispositivos que devem interagir com o sistema, como disco, memória RAM, adaptadores de rede, monitor, mouse, teclado, pen-drives, dispositivos de áudio, etc. O usuário do sistema nunca acessa o hardware de forma direta, mas sempre por meio do kernel, que oferece uma forma padronizada e simplificada para se comunicar com os equipamentos e dispositivos através de *drivers* e chamadas de sistema. No que diz respeito ao software, é o kernel que controla cada processo que deve ser executado num determinado instante, cada página da memória que é concedida a um programa e também todas as permissões de acesso que o usuário tem sobre cada parte do sistema de arquivos [11] [2]. Nesse sentido, o kernel é a camada de software situada entre o hardware e o conjunto de ferramentas que dão origem ao sistema operacional, no qual são executadas as aplicações, estrutura essa exemplificada na Figura 1.

2.1.1.1 Kernel monolítico e microkernel

Existem dois modelos de kernel mais conhecidos: o *kernel monolítico* e o *microkernel*, cuja configuração é representada na Figura 2. O kernel monolítico não possui uma separação estrutural muito bem definida, historicamente presente em sistemas operacionais cujo desenvolvimento se iniciou sem a pretensão de atingir alguma popularidade com seu lançamento [2]. O MS-DOS foi um exemplo de sistema de kernel monolítico, cujas

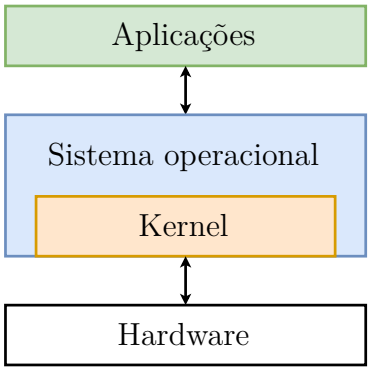

Figura 1 – Camadas de um sistema operacional.

(Fonte: do autor.)

funcionalidades foram sendo agregadas ao kernel com o tempo sem se preocupar com a modularidade dos seus subsistemas. Os níveis de funcionalidade não eram bem separados, ao ponto aplicações conseguirem acessar diretamente os dispositivos de hardware sem nenhum tipo de proteção. Não raramente, falhas em aplicações do usuário podiam causar a falha do sistema operacional inteiro, que necessitava ser reiniciado. No kernel monolítico, muitas funcionalidades são agregadas ao kernel, tornando o sistema operacional mais rápido e leve, porém muito mais complexo de receber manutenção e adições de novas funcionalidades.

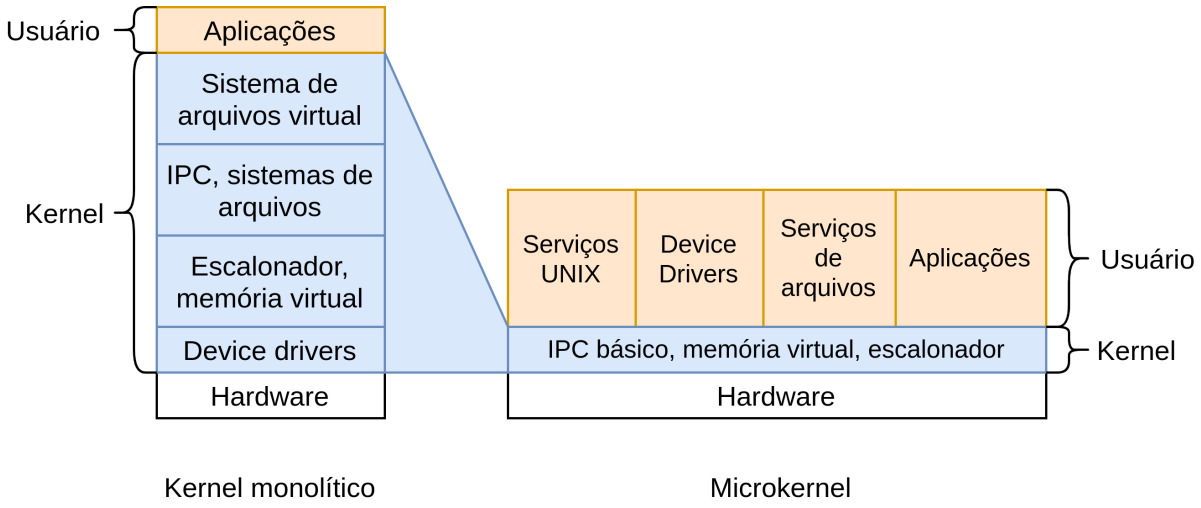

Figura 2 – Comparação entre os kernels do tipo monolítico e microkernel.

(Fonte: adaptado de [12])

O microkernel segue a abordagem de delegar uma quantidade pequena, mas essencial, de responsabilidades ao kernel. As funcionalidades que não forem agregadas ao kernel são implementadas como tarefas do sistema ou aplicações do usuário. Como resultado,

tem-se um kernel menor, que funciona como um intermediário entre os serviços e as aplicações. Toma-se por exemplo uma aplicação que deseja utilizar a rede. Para tal ela deverá se comunicar por meio da troca de mensagens com o kernel, enquanto que o kernel se comunicará da mesma forma com o servidor responsável pelo gerenciamento da rede. Em sistemas baseados em microkernel, a comunicação entre processos e a interação dos processos com o hardware não é feita de forma direta, mas sempre por meio da comunicação indireta com o kernel, que implementa as formas mais básicas desse mecanismo de IPC (comunicação entre processos) [12]. Sistemas com microkernel tendem a ser mais facilmente portados para novas arquiteturas e plataformas, já que menores alterações são necessárias, ao custo que tendem a ser mais lentos por conta do incremento das atribuições recebidas pela camada do sistema.

2.1.1.2 Kernel modular

O kernel em módulos pode ser visto como uma adaptação do microkernel no sentido que ele é focado em agregar apenas funções essenciais. O kernel modular, porém, pode ser expandido por módulos de kernel, mesmo em tempo de execução, e não depende de troca de mensagens para controlar o fluxo das requisições das aplicações, o que torna o sistema mais rápido. Caso o sistema operacional necessite de novos *drivers* para o acesso de dispositivos, eles podem ser adicionados dinamicamente. Esse kernel é comumente encontrado em implementações modernas do Unix, como Linux, Mac OS e Solaris (mais detalhes a respeito do Unix e Linux serão vistos no capítulo 3).

2.1.2 Processos

Um processo é um programa – código ou conjunto de instruções guardado em uma mídia – em execução. O processo é a unidade de trabalho mais básica em sistemas de tempo compartilhado modernos. Em sistemas operacionais mais antigos só era possível que um processo fosse executado por vez, onde o próximo só poderia entrar em execução uma vez que o anterior terminasse. Os sistemas operacionais atuais são compostos por múltiplos programas executando concorrentemente no computador, requisitando recursos de memória, processamento, acesso a arquivos e dispositivos, etc [2].

Os processos devem ser carregados na memória para serem executados, sendo compostos por algumas partes que descrevem a sua atividade. Eles contêm a **seção de texto**, que armazena o código do programa que o originou. Processos também guardam a informações do **contador de programa**, que indica qual a posição atual da sua execução, e dos registradores da CPU para caso sua execução seja interrompida, permitindo retornar de onde pararam – sem nenhuma perda de informação – quando forem selecionados novamente. A memória alocada por um processo é comumente classificada de três formas diferentes: pilha (*stack*) de dados, seção de dados e memória alocada de forma dinâmica.

A **pilha** representa os dados alocados temporariamente, como funções e variáveis temporárias alocadas durante a execução de funções, onde uma vez que a função retorna, a memória utilizada por ela é liberada – o nome pilha surge dessa ilustração de empilhar e desempilhar a memória conforme ela é utilizada. A **seção de dados** representa todas as variáveis globais que são criadas logo no início do processo, cuja memória só é desalocada quando ele for finalizado. Por fim, a **memória dinâmica** representa o que for alocado pelo processo cuja quantidade não é conhecida em tempo de compilação e que oferece liberdade para ser desalocada a qualquer momento – podendo também não ser desalocada. A estrutura da memória de um processo por ser vista na Figura 3.

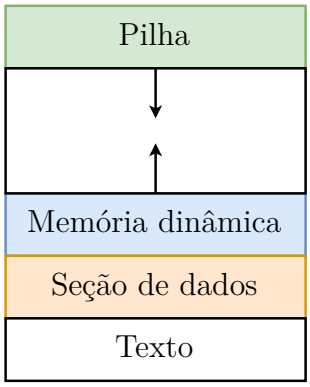

Figura 3 – Representação do processo na memória do sistema.

(Fonte: adaptado de [2])

2.1.3 Estado dos processos e escalonamento

Como é mostrado na Figura 4, um processo pode estar em alguns estados específicos. Esses estados estão diretamente relacionados ao escalonador (ou agendador) de processos, um subsistema presente em qualquer sistema operacional de propósito geral, responsável por decidir qual processo será executado na CPU, e por quanto tempo [7]. Um processo encontra-se no estado **novo** quando acabou de ser carregado para a memória e cujo PCB (bloco de controle do processo) ainda está sendo preenchido com as necessárias informações para classificá-lo para o sistema enquanto não tiver sido finalizado. Um processo pode estar **em espera** por dados do disco, pacotes da internet, eventos ou sinais, não tomando o uso do processador enquanto isso. Ele pode estar **pronto para executar**, posicionado na fila junto aos demais processos também nessa situação e esperando para ser escolhido pelo escalonador – escolha essa baseada no algoritmo de escalonamento e definição de prioridades utilizados – para mais uma bateria de ciclos de execução. Quando **em execução**, o processo terá reservado um tempo da CPU para si até que execute tarefas que o deixem em espera novamente, uma interrupção ocorra ou sofra preempção. Quando

um processo terminou sua execução proposta ele é **finalizado** e liberado da memória do computador [2].

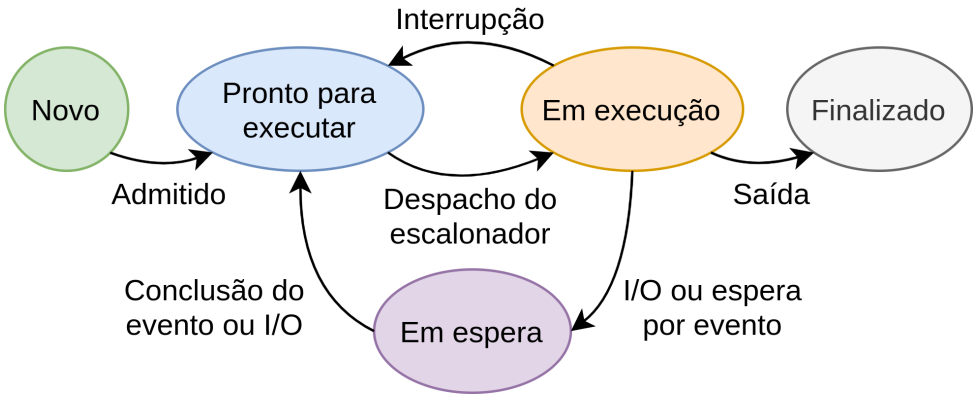

Figura 4 – Diagrama dos possíveis estados de um processo.

(Fonte: adaptado de [2])

2.1.3.1 Interrupções

O ato de interromper um processo o colocará novamente na posição de *pronto para executar*, dando início à execução de alguma tarefa no espaço do kernel. Quando uma interrupção ocorre, todo o estado do processo é salvo no seu respectivo bloco de controle antes da troca do nível de usuário para o nível do kernel. A atividade que será executada no kernel quando intende-se conduzir um novo processo a entrar em execução será a troca de contexto. A interrupção pode ocorrer de forma *cooperativa* (ou *não preemptiva*), onde o próprio processo indica ao escalonador que deseja parar temporariamente sua execução e conceder o espaço da CPU para outro processo na fila. Há também a interrupção *não cooperativa* (ou *preemptiva*) se o processo for forçadamente retirado de execução [2].

2.1.3.2 Algoritmos de escalonamento

O escalonamento de processos em sistemas operacionais de propósito geral visa garantir a melhor experiência na interação do usuário com as aplicações, bem como deve distribuir o máximo da utilização da CPU para os processos e minimizar o tempo de espera. O escalonador pode funcionar de maneiras diferentes dependendo de qual algoritmo é aplicado à ele. Existem três algoritmos mais conhecidos e capazes de serem utilizados em GPOSs: *Primeiro a Chegar, Primeiro a ser Servido* (FCFS); *Menor Trabalho Primeiro* (SJF); *Circular* ou *Round-Robin* (RR) [2]. A Figura 5 aplica os diferentes modos de escalonamento ao tomar como exemplo dois processos P_1 e P_2 com tempos de rajada (execução) de 20ms e 10ms, respectivamente.

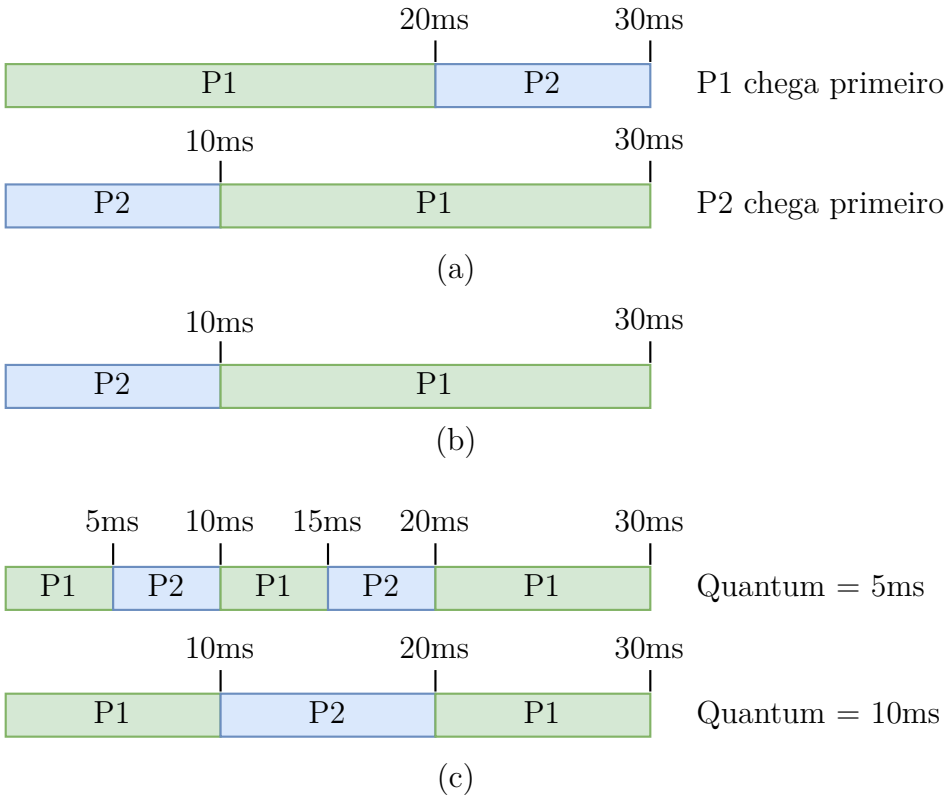

Figura 5 – (a) Primeiro a chegar, primeiro a ser servido (FCFS). (b) Menor trabalho primeiro (SJF). (c) Round-Robin (RR).

O algoritmo FCFS ou FIFO (*First In, First Out*) é o mais simples de ser entendido. Ao utilizá-lo, cada tarefa que precisa ser executada é colocada no último lugar da fila do escalonador, independente do seu tempo de execução; a prioridade dos processos é relacionada ao momento da sua chegada na fila. O FCFS potencialmente oferece o maior tempo de espera médio entre os três por não ajustar as posições da fila, onde se uma tarefa com grande tempo de execução tomar o lugar no processador, uma pequena tarefa subsequente com tempo de execução muito menor obrigatoriamente deverá esperar a finalização da primeira.

O SJF é similar ao FCFS, porém ele ajusta as posições da fila de forma que as tarefas com menor tempo de execução sejam colocadas para serem executadas primeiro. O SJF oferece menor tempo de espera médio em comparação com o FCFS, entretanto ele potencialmente pode causar *starvation*, onde um processo com menor prioridade (maior tempo de execução) pode ficar indefinidamente esperando enquanto houverem processos com menor tempo de execução chegando na fila. Uma possível solução para o problema de *starvation* é o *aging* (envelhecimento), no qual processos em espera na fila têm sua prioridade aumentada com o tempo.

O modo de escalonamento *Round-Robin* foi implementado para resolver o problema de *starvation* e ainda manter um bom tempo médio de espera, tipicamente um pouco

maior que do SJF. Nesse algoritmo cada processo na fila recebe uma pequena quantidade de tempo da CPU, geralmente 10-100 milissegundos, e após transcorrido esse tempo na CPU a tarefa é colocada novamente no final da fila [2]. Para a experiência com o usuário o *Round-Robin* é o mais aconselhável, pois com esse pequeno tempo reservado para cada processo, também chamado *quantum*, é proporcionado paralelismo na execução dos processos. O valor do *quantum*, entretanto, não pode ser reduzido infinitamente pela existência da *latência de despacho*, que é o tempo que o despachador – módulo do escalonador – leva para operar a troca de contexto.

2.1.4 *Threads*

Threads, ou fluxos de execução, são a forma que um processo de computador possui para abstrair as diferentes tarefas que devem ser realizadas simultaneamente por ele, similar a como múltiplos programas em execução pertencem ao sistema operacional. Um mesmo programa pode ter uma interface gráfica, se comunicar com a internet, ter uma área de texto para digitar e estar fazendo determinados cálculos matemáticos exaustivos, todos geridos ao mesmo tempo [2]. Uma possível abordagem de implementação nesse caso é separar esse processo em diferentes *threads*, cada uma responsável por partes específicas da aplicação. Programas de computador que possuem um único fluxo de execução seguem do início ao fim sem se preocupar com o compartilhamento de variáveis e CPU (a não ser com outros processos), já as *threads* concorrem pelo processador, que distribui intervalos de tempo de execução entre todas elas, de onde surge o termo programação concorrente (*multithreaded programming*). Quando várias *threads* necessitam manipular uma mesma variável ou região crítica do sistema é necessário que esse acesso seja sincronizado, permitindo uma *thread* por vez utilizá-lo. Não gerenciar o uso de recursos compartilhados pode levar ao que é chamado de condição de corrida, onde duas ou mais *threads* tentam modificar um recurso ao mesmo tempo, levando a resultados inesperados.

2.1.4.1 Sincronização de *threads*

Para lidar com o problema da sincronização são normalmente utilizados semáforos, que permitem que um número limitado de *threads* acessem uma específica região do programa dependendo do valor máximo estabelecido. Pode-se também empregar um caso específico de semáforo que permite apenas uma *thread* em determinada região por vez: o *lock* (cadeado) – ou *mutex* (*mutual exclusion* ou exclusão mútua). Quando uma *thread* adquire um *mutex*, outras *threads* não poderão acessar a região guardada por ele até que ele seja liberado, ficando no estado *em espera* enquanto isso. Um tipo especial de *lock* é o *spinlock*, cuja *thread* ao invés de ficar *em espera* tomará o uso do processador no que pode ser classificado como *busy wait* (*espera ocupada*). *Spinlocks* são comumente usados no espaço do kernel, onde trocas de contexto podem ser mais demoradas que esperas

ocupadas, porém devem ser utilizados em regiões de espera pequenas a fim de serem de fato mais eficientes que *mutexes*.

2.1.4.2 *Threads* no Linux

O Linux tem uma implementação única de *threads*, já que todas elas são processos comuns e não há nenhum tipo de semântica especial que permita o escalonador diferenciar processos e *threads*. Supondo, por exemplo, que uma determinada aplicação seja implementada fazendo uso de quatro *threads*, todas elas aparecerão na lista de processos do Linux como processos com o mesmo nome. Processos que não implementam *multithreading* – ou seja, possuem um único fluxo de execução – aparecerão como um processo único, como esperado. Quando o escalonador está distribuindo o uso da CPU para os programas do sistema, uma aplicação com duas *threads* é literalmente vislumbrada como dois processos diferentes sem qualquer relação entre si. Quando uma aplicação cria uma nova *thread* no Linux, ela na verdade está dando início a um novo processo através da chamada de sistema `clone()`, similar à conhecida `fork()` – mas com a possibilidade de dividir recursos entre o processo pai e filho. Esse novo processo configurará nas *flags* da `clone()` (como argumentos da função) que haverá o compartilhamento da área de memória, recursos do sistema de arquivos, arquivos abertos e manipuladores de sinais do processo pai [7].

2.1.5 Chamadas de sistema (*syscalls*)

As chamadas de sistema (comumente chamadas de *syscalls*) são o meio pelo qual o sistema e aplicações requisitam os recursos gerenciados pelo kernel. Elas são geralmente implementadas nas linguagens C, C++ ou Assembly. São as chamadas de sistema que definem como que se deve comunicar com o kernel, sendo a única forma aceita para trocar informações com ele. As chamadas de sistema, entretanto, tipicamente não são acessadas de forma direta, mas utilizando-se de uma função de mesmo nome ou nome similar, padronizadas em uma API [7][2]. APIs podem ser exemplificadas como funções, estruturas de dados e outros componentes em uma linguagem de programação que abstraem o acesso ao sistema. As APIs mais conhecidas que definem conjuntos de chamadas de sistema são: a Win32, para sistemas Windows; a POSIX, para sistemas baseados no Unix (seção 3.1.2); e a API do Java, compatível com máquinas virtuais Java (JVM) [2]. A chamada de sistema propriamente dita vem envelopada em uma função que muitas vezes tem um papel bastante simples, sendo responsável por copiar os argumentos recebidos para os respectivos registradores utilizados pela chamada de sistema, para em seguida executá-la [13].

Para clarificar o funcionamento de chamadas de sistemas, toma-se como exemplo a função `printf()` da linguagem C em um ambiente Linux. Essa função tem como o

objetivo imprimir caracteres na saída padrão do sistema, *stdout*, com a possibilidade de empregar uma formatação personaliza através dos seus argumentos opcionais. Por exemplo, ao chamar `printf("Hello world!")` é dado início a uma sucessão de outras chamadas em cascata (resumidas no diagrama da Figura 6), partindo da aplicação e chegando até o kernel [7]:

1. A função `printf()`, escrita na linguagem C, é chamada no espaço do usuário.
2. A função `printf()` chama a função `write()` da biblioteca *glibc*, utilizada para escrever caracteres em um arquivo especificado como parâmetro – lembrando que no Linux, assim como no Unix, tudo é um arquivo, incluindo a saída padrão. Nesse caso o arquivo especificado é o *stdout*.
3. A função `write()` executa a chamada de sistema `write()` disponibilizada pelo kernel, cujos detalhes de implementação variam entre diferentes sistemas operacionais.

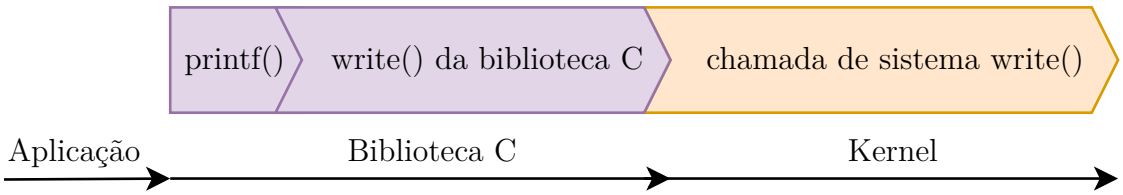

Figura 6 – Relação entre as APIs disponibilizadas numa linguagem de programação e as chamadas de sistema do kernel.

(Fonte: adaptado de [7].)

A função `write()` advinda da execução da função `printf()`, nesse cenário, é quem encapsula a chamada de sistema `write()` que fará o trabalho final de escrever "*Hello World!*" na saída padrão do sistema operacional. Esse tipo de abordagem pode similarmente ser verificado na API Win32, na qual, por exemplo, a função `CreateProcess()` na verdade executa a chamada de sistema `NTCreateProcess()` do kernel do Windows [2].

2.1.6 *Device drivers*

Um *device driver* (traduzido como controlador de dispositivo), também referido apenas como *driver*, é o meio pelo qual – por meio de uma camada de software – o sistema operacional abstrai os dispositivos conectados ao kernel com uma interface de programação bem definida para o usuário, isto é, aquele que desenvolve aplicações em torno do dispositivo [14]. Geralmente os *drivers* podem ser construídos de forma totalmente separada do kernel e adicionados mais tarde, como que plugados no sistema, de forma modular. É mediante os drivers que, por exemplo, um pen drive, um DVD, um disco rígido (HD), uma unidade de estado sólido (SSD) ou um cartão SD podem todos ser vistos pelo usuário como um

bloco contíguo de bytes ou caracteres, ainda que a implementação física de cada um deles seja completamente diferente entre si. *Drivers* são como uma caixa-preta que padroniza o acesso aos dispositivos, com a intenção de deixar o hardware disponível, mas sem forçar políticas de uso ao programador. Para que políticas não sejam forçadas, *drivers* devem ser o mais simples possível, deixando os detalhes de como utilizá-lo para as aplicações. Ainda que não devem forçar políticas, *drivers* podem e *devem* oferecer novas capacidades e possibilidades ao usuário ao conceder outras formas de configurar e comunicar-se com o dispositivo. No Linux existem três tipos de interfaces de dispositivos implementadas através de *drivers* [7], que são os

- Dispositivos de bloco: discos rígidos, pen drivers, CDs, etc, fornecendo acesso aleatório de dados, isto é, os dados podem ser acessados em posições arbitrárias e não necessariamente em sequência.
- Dispositivos de caractere: dispositivos não endereçáveis, cujos dados devem ser acessados de forma sequencial (serial), como teclados, mouses, impressoras.
- Dispositivos de rede: acessados pela API de *sockets*, diferentemente dos dispositivos anteriores. Como exemplo estão os diversos tipos de adaptadores físicos de rede, como *ethernet* e *Wi-Fi*.

2.2 Sistemas Embarcados

Sistema embarcado pode ser classificado como um sistema computadorizado que pretende executar apenas a aplicação para a qual foi feito . Um ar-condicionado, um sistema de injeção eletrônica de um automóvel, um relógio digital, um celular, um forno de micro-ondas ou uma máquina de lavar roupas, todos contém/são sistemas embarcados construídos especificamente para a função que executam, uns mais simples, outros mais complexos. Esses hardwares tendem a ser feitos sob medida, considerando as limitações de espaço, custo dos componentes, autonomia da bateria, custo do software empregado, etc.

Sistemas embarcados podem executar as suas aplicações de forma *bare-metal*. Nessas condições, os recursos de hardware são acessados diretamente, sem a intervenção de um kernel e fazendo uso direto dos seus *drivers*, que muitas vezes são baseados na API de sistemas Unix (`open()`, `close()`, `write()`, etc.), mesmo que não possuam sistema operacional [15]. Sistemas embarcados podem também possuir um sistema operacional, que será responsável por distribuir os recursos de hardware requisitados, consequentemente dando menor liberdade à aplicação – porém maior segurança operacional ao sistema. O termo *embarcado* pode servir tanto para um hardware feito sob medida para uma aplicação como para um software desenvolvido especialmente para algum modelo de hardware.

As linguagens de programação normalmente usadas em sistemas embarcados mais restritos são C, C++ e Assembly, cuja escolha depende totalmente das ferramentas de compilação cruzada disponíveis. Compilação cruzada é quando o software de uma arquitetura alvo é produzido em um sistema portador de uma arquitetura diferente. Esse tipo de compilação é grandemente utilizada em sistemas embarcados, já que na maioria das vezes esses sistemas de destino não são capazes de compilar suas próprias aplicações, diferente do que acontece computadores pessoais comuns. Um computador pode gerar código de máquina para outro microprocessador através de um conjunto de ferramentas para realizar a compilação cruzada, as *cross-compilation toolchains*. Os compiladores comuns geram código para o próprio ambiente operacional do qual foram chamados, já as *cross-compilation toolchains* são capazes de gerar códigos binários de outras arquiteturas, como por exemplo um computador convencional contando com um processador x86 de 64 bits que compila uma aplicação para arquitetura ARM de 32 bits por meio de uma dada *toolchain* que suporte essa operação.

2.3 hardware para GPOS

Todos os sistemas operacionais necessitam de alguns componentes básicos para funcionarem. A parte mais fundamental de um computador é o **processador** ou **CPU** (unidade de processamento central), onde a grande maioria das atividades de computação são realizadas. O processador é um *chip* responsável por executar as instruções dos programas e seu poder é mensurado pela quantidade de ciclos de instruções que podem executar por segundo, o *clock*, e também o número de núcleos que possui, que representam o seu potencial de executar processos em paralelo. O processador e todos os outros componentes são conectados a uma **placa-mãe**, que é a placa de circuito impresso principal do sistema, responsável por disponibilizar conexões elétricas e lógicas entre os componentes essenciais e também conexões para os dispositivos periféricos. Todo computador possui a **RAM** (memória de acesso aleatório), também popularmente referida apenas por *memória*, que armazena os dados e programas acessados pela CPU. Os dados permanecem ativos na RAM enquanto ela estiver conectada a uma fonte de alimentação e são perdidos uma vez que o computador for desligado. O processador possui também a **cache**, uma memória de acesso mais rápido que a RAM que guarda dados acessados com frequência, evitando a necessidade do acesso à RAM. A cache, porém, geralmente tem capacidade ordens de grandeza inferior a RAM por conta de seu preço ser muito mais elevado. É no **dispositivo de armazenamento** que o que sistema operacional e todos os arquivos dos usuários são guardados de forma permanente, podendo ele ser um disco rígido ou uma unidade de armazenamento flash. Por fim, para ser possível a interação com o sistema são necessárias as **portas de entrada e saída**, presentes na placa mãe, para a conexão dos dispositivos periféricos. Os tipos de portas mais comuns são as portas USB, portas

de rede RJ-45, conectores de áudio e portas de vídeo, como VGA ou HDMI [16].

2.3.1 Sistemas Linux embarcados

Com a criação de *sistemas em um chip* (SoC) cada vez mais poderosos, com processadores *multicore*, unidade de gerenciamento de memória (MMU) e capacidade de conectar-se com dispositivos periféricos, o Linux tornou-se viável na área de sistemas embarcados, substituindo gradativamente os microcontroladores [8]. O Sistema Linux é dito *embarcado* quando é construído sob medida para o hardware que irá portá-lo, baseado no microprocessador, memória, placa de vídeo e funções analógicas e digitais disponíveis.

2.3.1.1 Raspberry PI 3 B

Raspberry Pi refere-se a uma linha de dispositivos de placa única baseados na família de SoC da Broadcom que se inicia na versão BCM2708 (cujo modelo específico utilizado é o BCM2835), desenvolvida pela Raspberry Pi Foundation, uma entidade filantrópica do Reino Unido [17]. A finalidade inicial da Raspberry Pi Foundation era de oferecer computadores de baixo custo e capazes de executar o sistema operacional Linux de forma embarcada, incentivando a prática da computação e tornando o acesso a tecnologia um bem comum. Porém, a placa se popularizou rapidamente, atingindo o uso em diversas áreas da ciência [18], robótica e demais aplicações que se utilizam de sistemas embarcados.

A construção física de uma Raspberry Pi pouco se difere de uma placa mãe de um computador pessoal com relação a sua aplicabilidade, porém com tamanho bastante reduzido, e seu hardware é capaz de portar um sistema operacional de propósito geral, o mais comum sendo o Linux, que pode ser gravado no cartão SD usado como armazenamento interno. A placa Raspberry Pi 3 Modelo B (Figura 7) é a primeira versão da terceira geração da Raspberry Pi, sucessora da Raspberry Pi 2 B. O sistema possui as seguintes especificações técnicas:

- CPU: Broadcom BCM2837 (família BCM2710) Quad Core de 64-bit.
- *Clock*: 1.2GHz.
- Memória RAM: 1GB.
- Armazenamento: Porta Micro SD.
- Rede: Gigabit Ethernet 100Mb/s, wireless LAN e Bluetooth.
- GPIO: 40 pinos.
- USB: 4 portas.
- Saída de vídeo: HDMI.

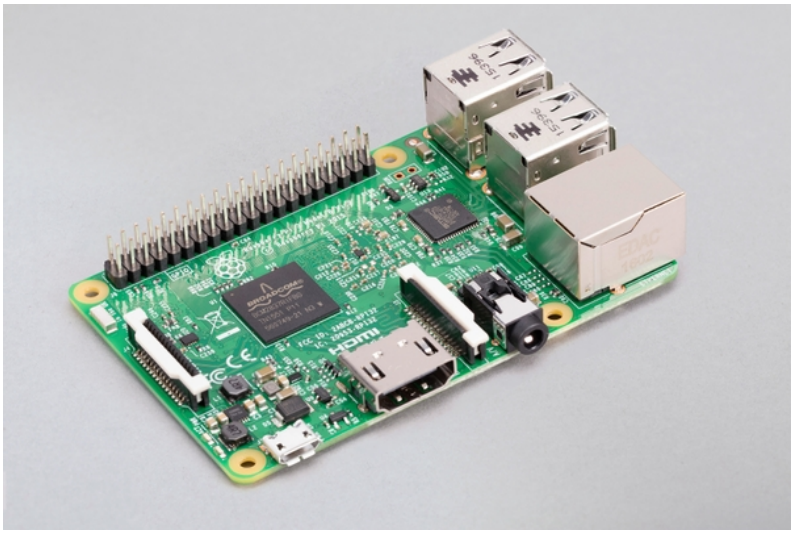

Figura 7 – Placa Raspberry Pi 3 B.

(Fonte: [19].)

- Entrada: 5V/2.5A, conector micro USB.
- Saída: 3.3V/5V (GPIO 3.3v)
- Porta CSI para conectar a câmera Raspberry Pi.
- Saída de vídeo DSI para conectar uma tela *touchscreen* do Raspberry Pi

Quando operada como computador convencional, a Raspberry Pi 3 aceita mouse, teclado e monitor, com a disponibilidade de ambientes gráficos e acesso a todo o tipo de funcionalidade que se espera esteja disponível em um microcomputador. Essas placas se distinguem de microcontroladores, que muitas vezes são empregados em tarefas mais simples, têm poder de processamento bastante inferior (quando comparados com uma Raspberry Pi) e não contam com uma camada de sistema operacional.

2.4 O conceito de *Patch*

Popularmente, um *patch* guarda quaisquer mudanças feitas em um programa de computador, tomando como base uma versão desse sistema. O ato de *aplicar um patch* significa tomar um programa e realizar essas mudanças – sejam adições, remoções ou substituições – através de um mecanismo que automatize esse processo. No Linux existem duas ferramentas relacionadas ao conceito de *patch*: `patch` e `diff`. O comando `diff` é capaz de fazer a diferença entre dois arquivos ou dois diretórios, com a habilidade de realizar essa função recursivamente (acessando subdiretórios). O resultado do `diff` popularmente é referido como um arquivo *patch*, que pode ser utilizado como entrada da ferramenta `patch`. Esse arquivo guarda as posições exatas, linha a linha, do conteúdo

dos arquivos de entrada – geralmente códigos ou *scripts* – onde onde houveram mudanças (adições ou subtrações). O comando `patch` recebe como entrada um arquivo ou diretório e também um arquivo *patch*, e tem como objetivo aplicar as mudanças presentes no *patch* sobre esse arquivo ou diretório. A ferramenta `patch` falha no caso de o arquivo de entrada e o *patch* não forem compatíveis, como no caso de arquivos inexistentes ou trechos pretendidos de mudança inexistentes nas posições indicadas no *patch* [20]. As mudanças recebidas sobrepõem os arquivos originais e essa nova versão é referenciada como *patched version*. A utilização de *patches* é muito comum em ferramentas de versionamento de código, como o *Git* e *Subversion*, que se utilizam de *patches* para representar as transições entre cada versão de um sistema, permitindo a navegação pelo histórico de um projeto e identificação precisa das mudanças.

2.5 Código aberto e licenças de software livre

Códigos aberto e *software livre* são termos bastante relacionados em sua definição. Quando de código aberto, um software deve vir acompanhado de, além dos seus possíveis arquivos binários, seu código fonte. Isso não significa que o software não possar ser monetizado e produto final seja vendido ou até mesmo sejam oferecidas licenças de suporte ao software. Entretanto, o código fonte desse tipo de distribuição deve ser disponível ao público para que ele possa ser modificado pelos usuários de acordo com a sua necessidade [21]. Todo *software livre* também é de código aberto e de acordo com a Free Software Foundation, fundada por Richard Stallman (que também deu início ao projeto GNU), garante [22]:

1. A liberdade executar o programa como desejar.
2. A liberdade de estudar o funcionamento do programa e também de modificá-lo.
3. A liberdade de redistribuir cópias do programa.
4. A liberdade de distribuir cópias do programa modificado de um terceiro.

Existem variados tipos de licenças que dão aporte ao projeto da Free Software Foundation, algumas mais restritivas quando às quatro liberdades, outras mais permissivas. As licenças Apache e FreeBSD, por exemplo, são mais flexíveis e concedem a possibilidade de serem criados softwares proprietários a partir de algum software que esteja licenciado por elas. Já para o caso das licenças advindas do GNU Project, como a GPL (*GNU General Purpose License*) e a LGPL (*GNU Lesser General Purpose License*), diz-se que elas são licenças mais restritivas. Ao utilizar-se de bibliotecas ou programas que se utilizam da GPL, não é possível, a partir delas, criar algum software proprietário. Com a LGPL é concebível derivar projetos proprietários, todavia, apenas em casos do uso de ligação dinâmica (*dynamic linking*) de bibliotecas, e com algumas limitações. Ainda que um software

se utilize de licenças de software livre, isso não impede que ele seja vendido, entretanto, seu código fonte deve ser acessível ao público para modificação.

2.6 Distribuições Linux

Distribuições Linux são ambientes completos que se utilizam do GNU/Linux (conjunto do kernel Linux e ferramentas GNU) como base para sua implementação. Geralmente, quando refere-se a um sistema operacional Linux é subentendido que está se falando de uma distribuição Linux. Todas as distribuições Linux são de código aberto e software-livre assim como o Linux o é, por conta da sua licença GPL versão 2.0, e podem ser baixadas da internet gratuitamente [23]. De fato, o que torna possível a existência de dezenas distribuições do Linux é justamente pelo ser de ele ser de software-livre, diferentemente, a título de exemplo, do Mac OS e Windows, que são proprietários. Algumas distribuições Linux populares são: Debian, Ubuntu, Manjaro, Fedora, Elementary OS e Linux Mint, cada uma delas para diferentes propósitos e tipos de usuários alvo. Distribuições Linux geralmente são disponibilizados na versão *desktop* (área de trabalho), que possuem ambientes gráficos, ou na versão para servidores, os quais são acessados unicamente via terminais de linha de comando e possuem um conjunto de ferramentas mais enxuto que as versões *desktop* – consequentemente sendo mais leves. É possível também criar distribuições Linux por conta própria ao fazer uso de uma das muitas ferramentas oferecidas na internet, como *Linux From Scratch* [24], *Ubuntu Imager* [25], etc.

2.6.1 Gerando distribuições Linux embarcadas

Distribuições Linux embarcadas muitas vezes necessitam de configurações específicas e ferramentas personalizadas de forma que sistema operacional suporte o hardware. Existem alguns projetos com a finalidade de integrar todo o desenvolvimento de distribuições Linux embarcadas em um único lugar, possibilitando a configuração de uma *cross-compilation toolchain*, do kernel, do sistema de arquivos e do *bootloader* (software responsável por carregar o kernel na memória). Dois dos mais conhecidos sistemas dessa categoria são o Buildroot e o Yocto Project, com o primeiro sendo mais simples e direto para construir imagens Linux, enquanto o segundo é mais adequado para famílias de produtos e sistemas com constantes mudanças incrementais.

2.6.1.1 Yocto Project

O Yocto Project é um projeto de código aberto para geração de distribuições Linux para ambientes embarcados. O *Layer Model* (modelo de camadas) do Yocto Project é um diferencial desse sistema com relação aos demais existentes. As camadas são conjuntos de repositórios que dizem ao sistema de construção OpenEmbedded, inerente ao Yocto, o que

deve ser feito na hora da compilação do sistema. As camadas permitem fácil colaboração, compartilhamento, reutilização e também clara separação lógica das configurações de uma *build*. Um mesmo sistema pode ser separado em camadas da GUI (interface gráfica do usuário), das aplicações, das configurações da distribuição, das especificações de hardware (*drivers*), etc. O BitBake é o núcleo do Yocto Project, presente no OpenEmbedded e mantido à parte do Yocto Project, e é usado para a construção das imagens. O BitBake possibilita a fácil distribuição de tarefas do *shell* e também do Python em paralelo que realizam uma complexa intercomunicação (comunicação entre processos) e se utilizam das receitas de construção escritas em um formato específico. O sistema OpenEmbedded também faz uso, além do BitBake, do OpenEmbedded-Core, que é camada de metadados comum a todos os sistemas que se derivam do OpenEmbedded. Para facilitar a construção de sistemas operacionais, o Yocto Project fornece o Poky, uma distribuição de referência que contém o sistema OpenEmbedded (OpenEmbedded e o BitBake) juntamente com um conjunto de camadas de metadados para possibilitar o usuário iniciar a construção de distribuições Linux embarcadas [26].

2.6.1.2 Buildroot

O Buildroot é uma ferramenta que simplifica e automatiza o processo de construir distribuições Linux embarcadas através da compilação cruzada, providenciando suporte a um grande conjunto de processadores e suas variantes. O Buildroot funciona no topo de um grande conjunto de *Makefiles*, que são arquivos de configuração de compilação utilizados pela ferramenta Make, capaz de compilar códigos de quaisquer linguagens de programação cujos compiladores sejam acessíveis pela linha de comando do sistema (*shell*). De modo similar ao que é encontrado ao compilar o kernel do Linux separadamente, o Buildroot fornece uma interface bastante simples para configurar o sistema com o comando `make menuconfig`. Nessa interface é possível: escolher a versão do kernel e seus cabeçalhos, bem como definir um kernel personalizado; configurar o sistema de arquivos e escolher pacotes e ferramentas desejados; definir a *toolchain* que será chamada pelos *Makefiles* (disponibilizada pelo buildroot ou externa); configurar *scripts* personalizados para serem executados na hora da construção da distribuição em questão [27]. Para compilar o sistema, basta utilizar o comando `make`, que irá:

1. Baixar todos os códigos fonte (conforme necessário).
2. Configurar, compilar e instalar a *cross-compilation toolchain*.
3. Configurar, compilar e instalar os pacotes escolhidos.
4. Compilar o kernel.
5. Compilar o *bootloader*.

6. Criar um *root filesystem* (sistema de arquivos a partir da pasta raiz).

7. Executar os *scripts* personalizados definidos previamente.

2.7 Sistemas de Tempo-Real

Tempo real, diferente do que o senso comum costuma atrelar à velocidade de resposta de algum mecanismo, é um conceito que dentro de sistemas computacionais e sistemas embarcados está intimamente relacionado com **determinismo** no atendimento de requisitos temporais [3]. Quando um sistema é dito de tempo real ou que possui alguma capacidade para tal, isso significa que ele possui algum nível de **previsibilidade** temporal nas tarefas que realiza, capaz de definir com certa precisão quais são os piores casos que leva-se para executar suas ações. Sistemas de tempo real não têm apenas a integridade dos resultados das suas operações como cruciais, mas é igualmente importante o tempo em que cada resultado é fornecido [1]. Sistemas de tempo real são cruciais em algumas áreas, como aviação, veículos automotivos e robótica, onde diversas tarefas embarcadas nesses sistemas necessitam de previsibilidade.

Tarefas de tempo real podem ser periódicas, que executam de tempos em tempos ao serem ativadas pelo temporizador do sistema, ou aperiódicas, podendo ser ativadas por algum evento como, por exemplo, o ato de apertar um botão, o atingimento do valor limiar de um sensor ou recebimento de dados por uma porta serial. Sendo periódicas ou não, as tarefas de tempo real definem o que é chamado de *deadline* (prazo final), que descreve o tempo máximo que uma tarefa tem para terminar sua execução após o evento que requisita sua execução tenha sido gerado. A extrapolação do tempo do *deadline* ao executar uma tarefa é chamado *deadline miss* (perda de prazo) e cujas consequências variam entre o propósito de cada sistema.

2.7.1 Tempo-Real *Hard* versus *Soft*

Os sistemas de tempo real são classificados com relação a gravidade das consequências que a extrapolações dos prazos requisitados podem causar. Usualmente esses sistemas são separados em: *hard real-time* (tempo real rígido) e *soft real-time* (tempo real brando). Quando a perda dos prazos das tarefas de um dispositivo podem ser catastróficas tanto para o sistema como para o ambiente, suas atividades são classificadas como de *hard real-time*. Ao tomar como exemplo o sistema dedicado de freios ABS de um automóvel convencional, se ele apresentar algum atraso, mesmo que de alguns poucos milissegundos, isso pode resultar em um acidente fatal. Sistemas *soft real-time*, diferentemente de sistemas *hard real-time*, podem aceitar algumas perdas de *deadline* sem que isso traga resultados críticos, como por exemplo sistemas bancários, um sistema *Voip* (*Voice over*

Internet Protocol) ou a execução de um arquivo multimídia, que podem apresentar atrasos inesperados, mas toleráveis.

2.7.2 Sistemas Operacionais de Tempo-Real (RTOS)

Quando um sistema operacional é capaz de oferecer ferramentas, bibliotecas e um kernel capazes de implementar aplicações de tempo real ele pode ser classificado, não exclusivamente, como um sistema operacional de tempo real (RTOS). Tempo real em sistemas operacionais está primeiramente relacionado com a capacidade de preempção que o escalonador tem sobre os processos. Se qualquer atividade em execução no sistema pode sofrer preempção – ou seja, ser retirada de execução para dar lugar a outra – em qualquer momento, então esse sistema é classificado como totalmente preemptivo. Comumente em GPOSs, nem todas as atividades do sistema podem sofrer preempção, principalmente algumas que são executadas no kernel (tudo o que é executado no espaço do usuário é preemptivo). Se o agendador de tarefas no kernel tem total domínio sobre quando a troca de contexto pode ser realizada, então o sistema tem potencial para ser implementado como um sistema operacional de tempo real. Esses sistemas são capazes de inicializar tarefas de tempo real, as quais tem as suas prioridades definidas logo na sua criação, diferente do que é visto nos GPOSs, que comumente não tornam explícita ou necessária a atribuição de prioridades [3]. Os RTOS, como são focados muitas vezes no mercado de sistemas embarcados, dadas as aplicações que necessitam de tempo real, acabam possuindo uma gama de ferramentas mais reduzida em comparação com GPOSs.

3 Introdução ao Linux

O Linux é um kernel de código aberto escrito por Linus Torvalds. Esse kernel tem fortes influências do Unix, seu antecessor, e também é baseado nos padrões POSIX, porém com características mais modernas [5]. Seu sucesso se deve a sua simplicidade unida ao seu poder, juntamente com a enorme comunidade que acompanha e agrega recursos ao projeto ao longo dos anos. A união dos projetos do Linux e GNU deu origem ao sistema operacional GNU/Linux, base de muitas distribuições Linux populares. O Linux é desenvolvido e mantido pela contribuição de diversos programadores através da internet e é o grandemente utilizado em *webservers*, sistemas embarcados e também em supercomputadores.

3.1 Unix

O Unix até hoje é visto como um dos sistemas operacionais mais elegantes e poderosos que já existiu, mesmo tendo sido originado em 1969. A criação do Unix se deu pelas mãos de Dennis Ritchie e Ken Thompson, na Bell Labs, quando a organização se viu sem um sistema computacional para suas pesquisas. O desenvolvimento do Unix teve forte relação com o surgimento da linguagem de programação C, também produzida na Bell Labs, que serviu como a base – sólida – da sua implementação. A popularidade e o crescimento do sistema se deram pelo Unix ser simples em seu conceito e elaboração. Ainda que o sistema fosse proprietário da Bell Labs, seu código fonte era distribuído juntamente com o sistema, o que fez que com o projeto recebesse contribuição de outras companhias, sendo também amplamente distribuído em universidades dos Estados Unidos [7].

3.1.1 Características do Unix

O Unix é um sistema operacional robusto, poderoso, estável, devido aos muitos anos de inovação e evolução dos quais participou. Nesse sistema operacional tudo é representado por arquivos, até mesmo aquilo que não é guardado no disco de armazenamento do computador, como informações sobre a CPU, memória, processos e dispositivos conectados. Esse mecanismo é chamado de sistema de arquivos virtual (VFS), no qual alguns arquivos que representam informações e interação específicas com o sistema estão apenas alocados na memória, mas disponíveis ao usuário como um arquivo comum. Nesse contexto, o Unix fornece um conjunto reduzido, com finalidades bem definidas, de chamadas de sistemas (*syscalls*) para interagir com esses arquivos, como: abrir (`open()`), fechar (`close()`), ler (`read()`), escrever (`write()`) e mover-se dentro do arquivo (`fseek()`), diferentemente do que pode ser observado em outros sistemas que possuem milhares de funções básicas (en-

quanto o Unix possui centenas) sem objetivos tão claros. O fato de ter sido escrito em C agrega muito para a portabilidade do Unix, já que a linguagem não é atrelada a nenhum tipo de arquitetura ou dispositivo, mas consegue gerar programas que rodam em qualquer arquitetura que possua um compilador C sem necessitar de alterações no código-fonte [28].

O Unix caracteriza-se por ser multiusuário e multitarefa, podendo ser acessado por múltiplos usuários ao mesmo tempo através de múltiplos terminais, fazendo com que em efeito prático cada um tenha seu espaço de trabalho próprio [29]. Essa sistema operacional serve tanto para uso individual como para ser adotado em *mainframes*, com a real diferença sendo apenas a velocidade e capacidade de armazenamento de cada um deles. A criação de processos em sistemas Unix é bastante rápida e a comunicação entre eles (IPC) é bastante robusta, facilitando a criação de múltiplos processos de propósito único que interagem entre si para originarem tarefas mais complexas. O Unix não é apenas um sistema operacional, mas um completo ambiente de desenvolvimento, com poderosas ferramentas amplamente conhecidas e ainda utilizadas em sistemas que derivam dele, como: `make` para executar instruções de compilação; `grep`, `awk`, e `sed` para transformação, tratamento e busca de texto com expressões regulares; pipe para concatenar a saída de comandos na entrada de outros; dentre outros comandos.

3.1.2 POSIX

As APIs mais utilizadas nos sistemas inspirados no Unix baseiam-se no POSIX (*Portable Operating System Interface*). O nome POSIX é utilizado como marca comercial de uma série de padrões da IEEE que fornecem um modelo para sistemas operacionais baseados no Unix, de forma a manter compatibilidade entre projetos que utilizam-se desse protótipo. Em sistemas que têm como referência o POSIX, as APIs que o desenvolvedor utiliza têm forte ligação que as próprias chamadas de sistema que interagem com os recursos de mais baixo nível [7]. Os exemplos de sistemas conhecidos que se baseiam no POSIX são o Mac OS X e o Linux.

3.2 O sistema operacional GNU/Linux

O Linux como hoje é conhecido deriva-se da união do projeto pessoal de Linus Torvalds e do sistema operacional do projeto GNU, de Richard Stallman.

3.2.1 O Projeto GNU

O GNU é um projeto com o intuito de criar um sistema operacional similar ao Unix, com distribuição livre. Iniciado em 1983 por Richard Stallman, o GNU, que é um acrônimo recursivo para "*GNU Is Not Unix*", tinha a finalidade de ir contra a grande quantidade

de softwares proprietários da década de 1980. Um desses softwares que o levou a iniciar o projeto foi o sistema operacional Unix, da Bell Labs, que a partir de certo ponto do projeto teve seu código fonte totalmente fechado. Stallman iniciou sua carreira no MIT como pesquisador, em contexto de total colaboração entre os membros do grupo no qual trabalhava, lançando mão de recursos exclusivamente de software livre. Na visão de Stallman, o software não-livre proibia a cooperação entre usuários na comunidade de informática e como um sistema operacional é a base para qualquer pessoa que decida usar e programar um computador, o primeiro item da agenda do Projeto GNU foi um sistema operacional gratuito/livre.

Um sistema como o Unix deveria incluir várias funcionalidades, como compiladores, editores e formatadores de texto, interfaces gráficas, bibliotecas de desenvolvimento, entre outras. Vale ressaltar que a ideia conceitual de um sistema operacional da época era muito mais simples que o que se espera em um computador hoje. Os componentes presentes no pacote GNU inicialmente eram: compilador C (`gcc`), `make` (a versão GNU do Unix `make`, aprimorada), Emacs, biblioteca C (glibc) e coreutils (comandos essenciais do terminal). Porém, a parte mais básica ainda não tinha sido criada, o kernel, fazendo com que o conjunto de programas até então desenvolvidos só pudesse ser executado em sistemas similares ao Unix (*Unix-like*) já existentes [30].

3.2.2 Surgimento do Kernel Linux

A primeira vez que o kernel Linux foi disponibilizado publicamente na internet foi no ano de 1991, por Linus Torvalds. Jovem finlandês e estudante da *University of Helsinki*, Linux Torvalds estava perturbado pela falta de sistemas baseados no Unix, na época que o Microsoft DOS dominava o mercado de computadores pessoais. Apesar de ter disponível para uso um Minix, sistema operacional Unix com propósito de auxiliar no ensino, Linus via problemas em tentar disponibilizar o seu código fonte por razão da licença que era atrelada ao software. [7]

Linus começou desenvolvendo um emulador de terminal simples com o qual conseguia se conectar nos servidores Unix da sua universidade. Esse emulador foi evoluindo e recebendo novas funções, baseado no contato que tinha com o Unix, até de tornar a forma mais primordial do que é chamado de Linux, uma união das palavras Linus e Unix. A diferença é que esse sistema era totalmente próprio, pois apesar de se basear no Unix e implementar a sua API, o Linux não era um descendente direto do código fonte do sistema operacional que o inspirou. Apesar disso, os conceitos e interfaces fundamentais do Unix não foram deixados de lado ou abandonados por Linus, ainda que alguns caminhos tomados por ele de alguma forma se desviaram da implementação original conforme ele decidiu ser necessário. As características de ser um sistema simples, multiusuário, onde tudo é representado por arquivos, que escala muito bem em hardwares mais potentes e

que possui um conjunto de chamadas de sistema compacto e bem definido permanecem no Linux.

3.2.3 A união do Linux e das ferramentas GNU

Na mesma época que o Projeto GNU tinha finalizado a produção das suas principais ferramentas, porém sem ainda ter dado início a criação de um kernel, foi que Linus tornou público seu kernel. Torvalds disponibilizou o Linux em um fórum público na internet no ano de 1991. A união do conjunto de utilidades GNU e o kernel Linux deu origem ao que melhor se conhece hoje como sistema operacional Linux, também referenciado como GNU/Linux, e que serve como base de muitas distribuições espalhadas pelo mundo [30]. O Linux é oferecido sob a licença GPL (*GNU General Public License*) versão 2.0, fazendo com que obrigatoriamente quaisquer distribuições baseadas no Linux devam ser disponibilizadas gratuitamente, juntamente com seu código fonte. Ser um sistema de código aberto e também livre possibilitou esse projeto crescer grandemente através de toda a contribuição que recebe pela internet, com Linus Torvalds ainda sendo o principal mantenedor desse sistema operacional até os dias de hoje [7]. Mesmo que de código aberto, o Linux também se mantém através de investimentos feitos por empresas das mais variadas áreas de atuação interessadas no projeto.

3.2.4 O Linux no mercado

O Linux ocupa uma ótima parcela do mercado em diversas áreas da computação, mesmo sendo *open-source*. A pesquisa feita pelo Top500 de novembro de 2020 aponta que todos os 500 supercomputadores mais potentes do mundo rodam o Linux [31]. De acordo com uma pesquisa feita pela EE Times em 2019, registra-se que 38.42% dos sistemas embarcados que possuem sistemas operacionais usam Linux, 2.82% são Unix (não Linux), 10.73% são Windows e 37.30% usam outros sistemas [4]. Com relação aos *webservers*, a pesquisa da W3Techs de 3 de abril de 2021 (atualizada diariamente) apresenta que 74.4% de todos os *webservers* são Unix, onde 51.4% dessa fração é ocupada por sistemas Linux (aproximadamente 38.24% do total). A W3Techs também expõe que o restante dos *webservers*, 25.7% do total, é implementado pelo sistema operacional Windows [32]. O kernel do popular sistema operacional de *smartphones* Android utiliza-se de uma versão modificada do Linux. A pesquisa de março de 2021 da Statcounter GlobalStats afirma que o uso dos sistemas operacionais em dispositivos móveis está dividido em: Android - 71.81%; iOs - 27.43%; Samsung - 0.38%; KaiOS - 0.14%; Desconhecido: 0.14%, Linux - 0.02% [33].

3.3 Tempo-Real no Linux

Esta seção destina-se a expor algumas abordagens para tornar o Linux capaz executar processos com características de tempo real. A Seção 3.3.2 apresenta como o Linux Vanilla pode implementar tarefas com teor de tempo real. A Seção 3.3.3 introduz o Real-Time Linux, obtido através da aplicação de um *patch* no Linux Vanilla. A última abordagem de tempo real discutida, na Seção 3.3.4, é o Xenomai. O Xenomai tem a configuração de kernel duplo, com agregando o kernel do Linux e kernel Cobalt, e também pode assumir a forma de kernel único, usando o kernel Mercury.

3.3.1 Por que tempo real no Linux?

Usar o Linux como um sistema operacional de tempo real traz consigo muitas consequências positivas. Caso ele seja capaz de desempenhar tarefas de tempo real, tem-se em mãos, além dessa competência, um sistema operacional muito bem consolidado e estável, com inúmeras ferramentas, bibliotecas e serviços que somente ele (e talvez outros sistemas *Unix like*) oferece. Considerando a gradativa melhoria nos hardwares embarcados, pode-se pensar no Linux no futuro como a ferramenta central de desenvolvimento de aplicações de tempo real, fato esse que agregaria ainda mais pessoas à grande comunidade de desenvolvimento do kernel.

O Linux de tempo real pode servir como base para centralizar projetos que se utilizam de diversos sistemas embarcados distribuídos, uns que portam o Linux e realizam atividades comuns, e outros que necessitam atender a requisitos tempo real. O conceito de tempo real pode, inclusive, ser expandido e aplicado nas atividades comuns que são realizadas em computadores pessoais, garantindo qualidade de serviço e entrega interna através de processos que respeitam as *deadlines*. O tempo real no Linux, apesar disso, não deve ser visto como uma forma de substituir os outros RTOSs, mas pode ser visualizado como uma nova vertente, que fornece todas as possibilidades de GPOSs somado às funcionalidades de tempo real. Nas seções seguintes será visto como algumas abordagens de Linux de tempo real estão até mesmo disponíveis de forma nativa no kernel e configuradas através de chamadas de sistema.

3.3.2 Linux Vanilla

Linux Vanilla, também chamado de *mainline* (linha principal), refere-se a linha de desenvolvimento principal do kernel Linux. Mesmo o Linux padrão é capaz de fornecer ferramentas para aplicações do sistema usufruírem de algum nível de tempo real, além das formas padrão que são oferecidas para configurar os processos.

3.3.2.1 Algoritmos de escalonamento do Linux

O Linux concede aos processos a capacidade de serem escalonados utilizando-se de cinco algoritmos diferentes, três são normais e dois são de tempo real. As formas de escalonamento normais são [34]:

- `SCHED_OTHER`: política de escalonamento padrão do Linux, o CFS (*Completely Fair Scheduler*), baseado no Round-Robin com divisão dinâmica de uma parcela do processador – não de tempo – entre os processos. [7].
- `SCHED_BATCH`: escalonamento direcionado a processos *CPU bound*, isto é, dependentes principalmente de atividades executadas no processador, como a execução cálculos matemáticos, por exemplo.
- `SCHED_IDLE`: para executar processos de baixa prioridade em *background* no sistema.

Enquanto que as formas de escalonamento de tempo real são:

- `SCHED_FIFO`: política baseada em fila, onde o primeiro processo a chegar é o primeiro a ser atendido.
- `SCHED_RR`: política Round-Robin clássica.

A prioridade de um processo pode se definida em tempo de execução através da chamada de sistema `sched_setscheduler()` da biblioteca C do Linux, que de forma simplificada recebe como parâmetros o ID do processo, um dos algoritmos citados anteriormente e também um valor referente ao nível do prioridade do processo. Para especificar o nível de prioridade antes de execução faz-se uso do comando `chrt` do terminal em conjunto de opções que indicam o tipo de escalonamento requisitado, o valor de prioridade e também o processo que será iniciado.

3.3.2.2 Prioridades dos processos

Existem dois tipos de prioridades no Linux: a prioridade para processos normais, também descrita por *nice value*, e a prioridade de tempo real. Os valores *nice*, que variam de -20 a +19, indicam o quão "legal" é um programa ao sistema, onde valores altos significam que ele requer menor prioridade e valores baixos expressam que o programa revindica maior prioridade [35]. A configuração do valor *nice* está disponível para os modos de escalonamento normais: `SCHED_OTHER`, `SCHED_BATCH` e `SCHED_IDLE` e pode ser definido pelo comando `nice` do terminal ao executar um processo. Ao utilizar-se da função `sched_setscheduler()` passando como argumento um dos três algoritmos de escalonamento normal o valor de prioridade deve ser definido como 0 (zero), pois o valor de prioridade efetivamente é a prioridade de tempo real – indisponível nesse contexto. A prioridade de tempo real está disponível para os algoritmos de tempo real: `SCHED_FIFO` e

`SCHED_RR`. Os valores de prioridade de tempo real variam de 0 a 99, onde quantias maiores indicam prioridade superior. Tarefas de tempo real no Linux sempre terão prioridade sobre os processos normais.

3.3.3 Real-Time Linux

O Real-Time Linux, ou RTLinux, é o kernel do Linux modificado para suportar mais capacidades de tempo real que o Vanilla. Para se conseguir o Real-Time Linux é necessário aplicar o *patch* que torna disponível nas configurações de pré-compilação do kernel a opção `CONFIG_PREEMPT_RT`. Esse *patch* é popularmente chamado de `PREEMPT_RT` e deve ser compatível com a versão do kernel escolhido. Essa opção faz com que quase todo o kernel do Linux torne-se preemptivo, lembrando que nem todas as atividades do kernel podem normalmente sofrer preempção [9]. Para atingir esse alvo, a maioria dos *spinlocks* do kernel são substituídos por *mutexes*. Isso possibilita um maior número de trocas de contexto, potencialmente aumentando a latência do sistema, mas aumentando a responsividade do escalonador diante das prioridades dos processos, já que a qualquer momento uma atividade do kernel pode sofrer preempção.

A grande vantagem do RTLinux é que apenas a implementação do espaço do kernel é modificada, ou seja, todas as interfaces POSIX conhecidas e bem estabelecidas no Linux tradicional permanecem disponíveis e exatamente iguais. Para migrar uma aplicação feita no Linux Vanilla para o RTLinux não é necessária nenhuma modificação no código fonte do programa. Outra vantagem é o projeto ser gerenciado e disponibilizado pela própria árvore do projeto Linux e boa parte do desenvolvimento de tempo real já estar integrado no Linux *mainline*. Eventualmente, é possível que o próprio Linux Vanilla possua nativamente a opção de preempção completa do kernel, presente hoje apenas no *patch* `PREEMPT_RT`.

3.3.4 Xenomai

O Xenomai é um projeto de Software Livre que fornece abordagens para tornar o Linux capaz de executar tarefas que necessitem atender a requisitos de tempo real [10]. O Xenomai oferece duas opções: kernel duplo e kernel único.

3.3.4.1 Kernel duplo

Na configuração de kernel duplo, o Xenomai entrega capacidades de tempo real ao usuário ao acrescentar um novo kernel que roda lado-a-lado com o kernel do Linux, como uma extensão dele. Esse kernel (igualmente referenciado como *core*) chama-se Cobalt e é destinado a lidar com todas as atividades que exijam um atendimento de tempo crítico, como o tratamento de interrupções e o escalonamento de *threads* de tempo real. O que é

executado no Cobalt tem maior prioridade sobre as tarefas destinadas a serem executadas no kernel nativo (Linux). Com essa configuração de kernel duplo, também chamada de *co-kernel*, apenas as APIs que interagem com o kernel Cobalt, fornecidas pelo Xenomai, são ditas como qualificadas para atuar em tempo real, com prioridade sobre o Linux. A Figura 8 mostra como se dá a interação do usuário com o Cobalt. O *core* Cobalt é acessado através das diversas interfaces disponíveis, incluindo um subconjunto dos serviços do padrão POSIX, todos implementados na biblioteca *libcobalt* específica do Xenomai. Isso torna a migração de uma aplicação comum do Linux para o Xenomai uma tarefa bastante simples.

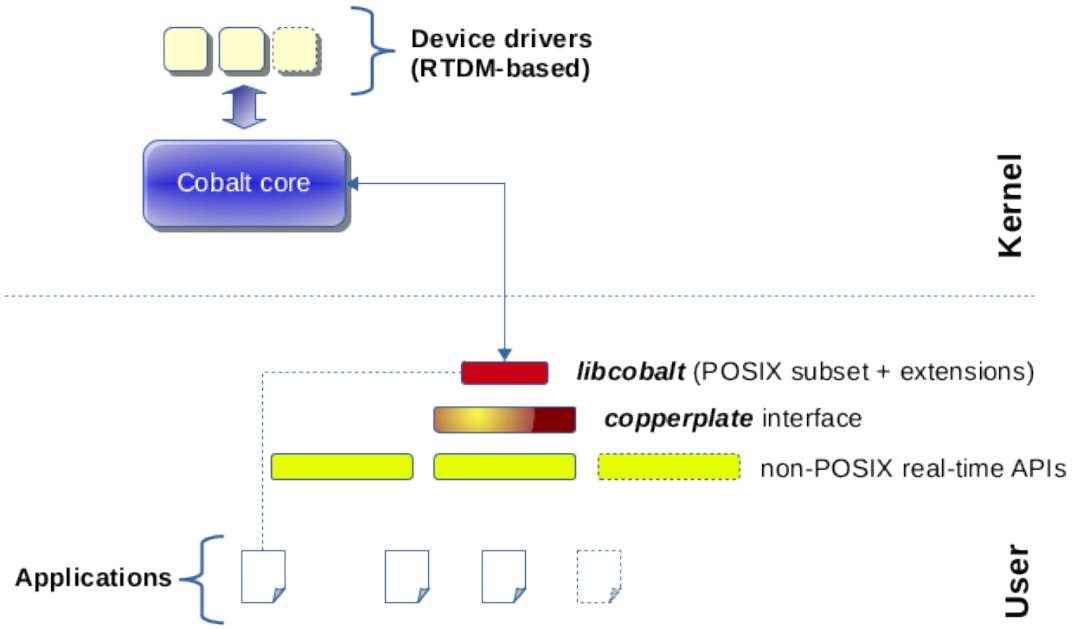

Figura 8 – Configuração de kernel duplo do Xenomai.

(Fonte: [36].)

Quando o Xenomai é usado em modo de kernel duplo, é importante entender que uma *thread* criada pela API do Xenomai pode pertencer a dois modos de operação: o modo primário, escalonado pelo kernel do Xenomai e desfrutando dos benefícios de tempo real *hard*, e o modo secundário, que é uma *thread* comum do Linux [37]. Uma *thread* do Xenomai pode ter seu modo alterado dinamicamente do modo primário para o secundário e vice-versa, dependendo dos serviços de kernel que forem chamados por ela. Se ela estiver no modo primário e chamar um serviço/*syscall* do Linux, seu modo será alterado para o secundário, perdendo as vantagens de tempo real, em que tal condição pode ser nomeada de "relaxamento" de *thread*. Se a tarefa estiver no modo secundário e chamar um serviço de tempo real do Xenomai, será convertida ao modo primário.

3.3.4.2 Kernel único

A partir do Xenomai 3, a versão mais recente até o momento, é oferecida uma nova opção que entrega tempo real dependendo apenas do Kernel do Linux e suas capacidades inerentes de tempo real, frequentemente aliada ao *patch* PREEMPT_RT. Essa abordagem de kernel único é chamada de Mercury. Nessa versão do Xenomai, todas as APIs que não sejam POSIX são emuladas com precisão sobre a biblioteca NPTL (*Native POSIX Threads Library*) do Linux presentes na biblioteca *glibc*.

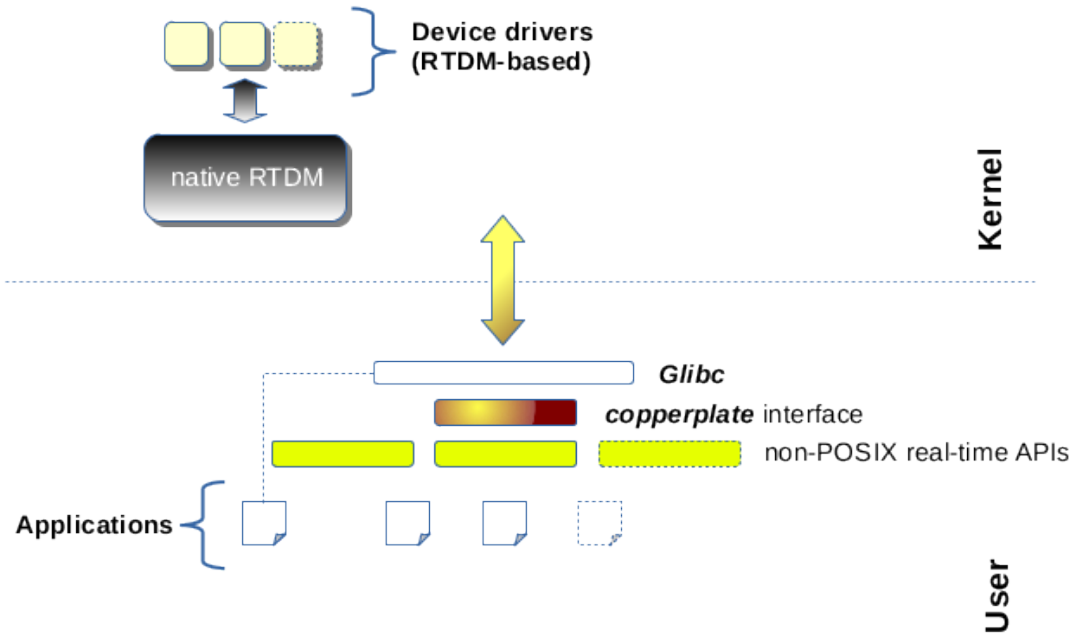

Figura 9 – Configuração de kernel único do Xenomai.

(Fonte: [38].)

3.3.4.3 *Interrupt pipeline* (I-pipe)

O fundamento por trás do funcionamento do Xenomai utilizando-se do mecanismo de kernel duplo está apoiado no que é chamado de *Interrupt pipeline* ou I-pipe. O I-pipe é uma camada de abstração de hardware (HAL), ou *hypervisor*, baseada no Adeos [39], posicionando-se abaixo dos kernels Linux e Xenomai. O I-pipe está disponível através de um *patch* que é aplicado ao kernel do Linux condedido pelo projeto Xenomai, compatível com diversas arquiteturas e versões do Linux. Esse *patch* além de adicionar o I-pipe, prepara o kernel do Xenomai, o Cobalt, para ser instalado juntamente com o Linux.

O conceito funcional do Adeos que se estende ao I-pipe é de adicionar uma camada de software que fornece primitivas e mecanismos que permitem que múltiplos SO, similarmente intitulados de domínios, compartilhem do mesmo hardware [40], funcionando de forma muito próxima ao que é chamado de virtualização completa [41]. Virtualização

completa requer que cada parte do hardware possa ser replicada e utilizada pelas máquinas virtuais, o que inclui todos os conjuntos de instruções da CPU, operações de entrada e saída, interrupções de hardware, acesso à memória e quaisquer outros elementos que rodam na *bare machine*. O I-pipe é capaz de gerenciar todas as interrupções externas ou internas, *syscalls* adivindas do Linux e outros eventos ocasionados pelo kernel, assegurando que a entrega desses eventos respeite as prioridades e a origem de cada um deles, seja do domínio de tempo real (Xenomai) ou não (Linux nativo).

3.3.4.4 APIs do Xenomai

O Xenomai tem um conjunto de APIs bastante flexível tanto para o desenvolvedor que deseja migrar de aplicações de RTOSs tradicionais, como para quem deseja migrar do Linux *mainline*. A interface padrão do Xenomai é a Alchemy [42], que é uma API em conformidade com os RTOSs tradicionais. A API Alchemy deixa explícito o uso de recursos de tempo real, cujas funções e tipos possuem o prefixo "`rt`". É possível também utilizar um subconjunto da API POSIX, onde ao passar uma *flag* específica ao compilador (`--posix`) fará com que as chamadas POSIX sejam convertidas em chamadas de tempo real do Xenomai no instante da compilação. O Xenomai também disponibiliza uma interface para a criação de *device drivers* de tempo real, chamada *Real-Time Driver Model* (RTDM) [43]. *Drivers* que necessitam ter capacidades de tempo real necessitam ser reescritos utilizando-se desse modelo. O RTDM tem uma API em conformidade com a conhecida POSIX, tornando o trabalho de migração mais simples. *Device drivers* que não necessitam de recursos de tempo real podem continuar sendo executados no espaço do Linux (modo secundário).

3.3.5 Vantagens e desvantagens das três implementações de tempo real

Essa seção comenta brevemente as vantagens e desvantagens de se usar cada um dos sistemas Linux com ferramentas de tempo real.

3.3.5.1 Linux Vanilla

Fazer uso do Linux sem modificações específicas (Vanilla) é a maneira mais prática e fácil de desfrutar das alternativas de tempo real presentes no sistema. Das três abordagens, entretanto, o Vanilla é o único que não tem o kernel totalmente preemptivo para executar as tarefas de tempo real. Com isso, o kernel Vanilla possivelmente apresenta o maior variação na resposta das tarefas de alta prioridade de tempo real.

3.3.5.2 RTLinux

A única diferença da construção do RTLinux para o Vanilla é a necessidade da aplicação de um *patch* específico. Apesar de as versões dos *patches* RT não acompanharem todos os lançamentos do Linux lado-a-lado, o seu desenvolvimento é bem próximo ao Vanilla. O Real-Time Linux tem a vantagem de poder se utilizar das exatas mesmas APIs disponíveis no Vanilla, podendo reutilizar os códigos fontes das aplicações ao migrar de um sistema para outro. Essa implementação também tem a vantagem de ter o kernel totalmente preemptivo, onde as atividades do kernel podem sofrer preempção (interrupção não cooperativa) a qualquer momento.

3.3.5.3 Xenomai

O desenvolvimento do Xenomai, diferentemente do RTLinux, não é integrado a árvore de projetos do kernel do Linux. Por ser um projeto a parte, o Xenomai acompanha as novas versões do Linux com menor velocidade. Ainda que as atividades de tempo real sejam executadas no Cobalt, um kernel totalmente reservado para processos de tempo real e diferente do Linux, o Xenomai disponibiliza boa parte do conjunto de interfaces POSIX para implementar as suas aplicações de tempo real. A possibilidade de fazer uso da API POSIX torna a migração de aplicações do Linux ao Xenomai uma tarefa simples. Com o uso do I-pipe como *hypervisor* para decidir como as interrupções são enviadas aos kernels (Linux e Cobalt), espera-se um melhor controle das latências das atividades que são executadas no Cobalt quando comparando-o com as outras duas abordagens.

4 Métodos de Ferramentas Utilizados nos Testes

O propósito principal deste trabalho é realizar testes de desempenho sobre as três abordagens de Linux de Tempo-Real apresentadas no capítulo Capítulo 3: o Linux Vanilla, o Real-Time Linux (ou RTLinux) e o Xenomai de kernel duplo. Os testes baseiam-se em avaliar a latência no atendimento de interrupções de hardware de uma placa Raspberry Pi 3 portando cada um dos sistemas propostos de maneira embarcada. Este capítulo busca detalhar os métodos e ferramentas utilizadas para adquirir e analisar os resultados obtidos. Os procedimentos da instalação de cada um dos sistemas operacionais podem ser vistos na Seção 4.1. A Seção 4.2 discute sobre o uso de interrupções de hardware para avaliação do desempenho de tempo real em sistemas operacionais. O sistema proposto é apresentado de forma geral na Seção 4.3, ao passo que a Seção 4.4 entra em detalhes de implementação das aplicações utilizadas.

4.1 Configuração e instalação dos sistemas operacionais

Para uma comparação mais justa é utilizada a mesma versão do kernel *mainline* para todos os sistemas Linux. A versão do Linux escolhida é a 4.19.144, compatível com o *patch* do I-pipe do Xenomai para a arquitetura ARM (utilizada pela Raspberry Pi 3) e também com o *patch* RT do RTLinux. O *patch* RT utilizado é destinado à versão 4.19.148, porém é compatível com a versão 4.19.144. A versão do Xenomai escolhida é a 3.1, a mais recente até o momento da realização do trabalho.

A configuração e compilação dos sistemas a partir dos kernels escolhidos é feita pelo Buildroot, responsável por gerar a imagem final que será executada na Raspberry Pi 3. O uso do buildroot parte do ponto onde todos os kernels receberam os patchs necessários, fato que não se aplica ao Vanilla/*mainline*. Os detalhes da configuração dos kernels e do buildroot, e da compilação e instalação dos sistemas podem ser vistos no Apêndice A.

4.2 Avaliação de desempenho dos sistemas através do uso de interrupções

Latência é o tempo que determinada tarefa leva para ser executada ou para iniciar sua execução a partir do seu acionamento. A medição da latência no atendimento de interrupções é um método bastante conveniente para verificar o potencial de tempo real

de um sistema. O gerenciamento de interrupções é a parte mais crítica e também de maior prioridade em sistemas operacionais, pois tarefas que dependem delas costumam ser aquelas que exigem a maior responsividade.

O valor absoluto da latência ou da latência média de uma série de execuções sucessivas não é capaz de apontar se um sistema é ou não capaz de tempo real. O *jitter* representa a varição da latência, podendo ser avaliado através de medidas do desvio padrão, por exemplo. É o *jitter* que pode melhor definir se um sistema é mais adequado que outro para executar tarefas de tempo real em uma determinada aplicação. Além disso, caso deseja-se avaliar se um dado sistema se enquadra na categoria de tempo real *hard*, o valor de latência máxima é bastante direto para aferir se todas as *deadlines* estão sendo cumpridas.

O que deseja-se medir, nesse caso, é a latência total, cuja separação pode ser vista na Figura 10. A latência total pode ser dividida em:

- Latência da interrupção: tempo decorrido desde a ocorrência da interrupção até o início do tratamento dela.
- Latência da rotina de tratamento de interrupção: tempo que o sistema leva identificar a interrupção e colocá-la em fila, tornando-a disponível às aplicações.
- Latência do escalonador: tempo para o escalonador entrar em ação após o término da rotina de tratamento de interrupção.
- Duração do escalonador: tempo da troca de contexto, que retorna a execução do tratamento de interrupção (kernel) para o nível de aplicação.
- Latência da execução da tarefa: compreende o tempo necessário para a aplicação ler a interrupção e enviar o sinal de resposta.

Não incluso na Figura 10, há também o tempo que o dispositivo externo necessita para processar a resposta advinda do sistema e assim computar o tempo de chegada dela, o que será discutido principalmente na Seção 5.3.

4.2.1 NodeMcu

Como gerador de interrupções para os testes propostos é utilizado o NodeMcu, um ambiente de prototipagem microcontrolado e baseado no chip Wi-Fi ESP8266 [45]. O NodeMcu possui algumas especificações técnicas bastante oportunas para trabalhar como gerador de interrupções [46]:

- CPU: Xtensa LX106 de 32-bit (ESP8266)
- Taxa de *clock*: 80 MHz

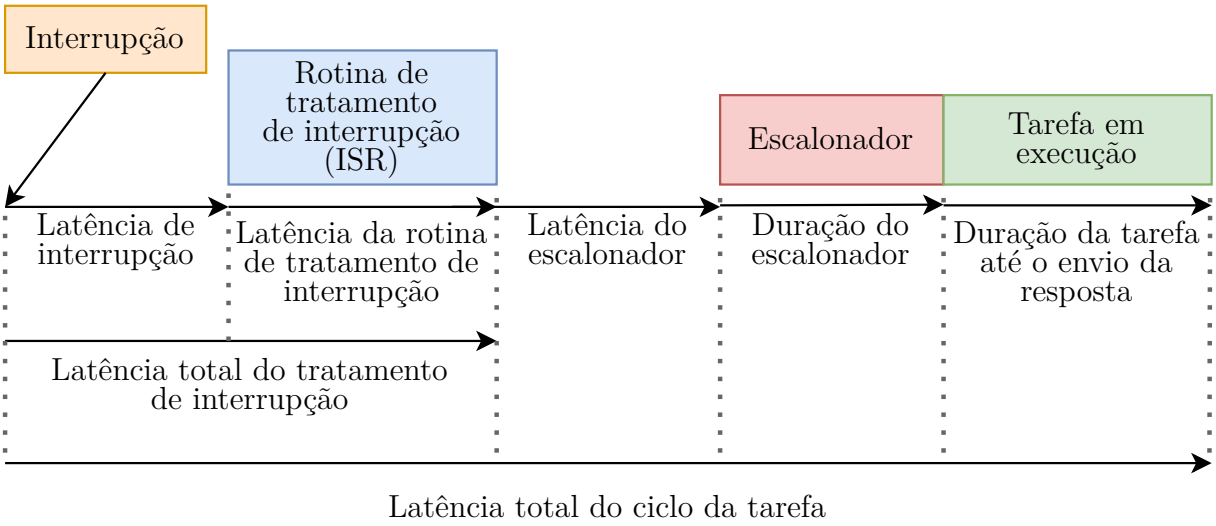

Figura 10 – Representação das diferentes latências encontradas no tratamento de interrupções nesse trabalho.

(Fonte: adaptado de [44])

- Memória: Flash de 4 MB
- Tensão de operação: 3.3V
- Tensão de entrada: 7-12V
- Pinos digitais: 16, sendo 11 deles GPIO
- Pinos analógicos de entrada: 1
- UARTs: 1
- SPIs: 1
- I2Cs: 1
- SRAM: 64 KB

Os pinos GPIO presentes na placa podem ser usados como entrada, saída e também como interrupção. Uma vantagem de se utilizar do NodeMcu é que seus pinos trabalham na mesma tensão que os pinos GPIO da Raspberry Pi, isto é, 3.3v. Esse microcontrolador é adotado como gerador de interrupções ao alterar o valor da saída digital conectada à Raspberry Pi em intervalos pré-definidos, entre *high* e *low*. O NodeMcu, ao mesmo tempo, atua como receptor, esperando a resposta advinda do dispositivo sobre testes. Nesse sentido, o NodeMcu é empregado como uma forma simplificada de gerador de funções e também de osciloscópio, já que é responsável por enviar e receber os sinais e computar os respectivos tempos de ambos.

O NodeMcu apresenta uma taxa de *clock* cinco vezes maior que o conhecido microcontrolador Arduino Uno, que possui frequência 16MHz, o que permite lançar mão de períodos de interrupções menores sem que o poder de processamento seja um gargalo da aplicação. Para criar aplicações destinadas ao NodeMcu pode-se fazer uso do *Arduino core for ESP8266 WiFi chip* [47]. Esse projeto dá suporte ao ESP8266 e os diversos sistemas que nele se baseiam, incluindo o NodeMcu [47], na IDE do Arduino. Com isso, as bibliotecas e ferramentas já conhecidas no desenvolvimento com o Arduino ficam disponíveis para implementar programas do NodeMcu.

4.2.2 Gerenciamento de interrupções no Linux com a Raspberry Pi 3

Por ser o sistema sob testes, o Linux deve tratar as interrupções de hardware recebidas nas portas suas GPIO. A Raspberry Pi 3 se enquadra muito bem para verificar o desempenho dos sistemas propostos (Linux Vanilla, Real-Time Linux e Xenomai), pois além receber suporte de todos eles, também se dispõe de portas GPIO que serão utilizadas no tratamento de interrupções. A Figura 11 mostra detalhes da disposição e função dos pinos e vale ressaltar que a numeração das portas GPIO não segue a sequência numérica do cabeçalho. A Raspberry Pi também é uma opção relativamente barata e prática para testar sistemas operacionais Linux, não sendo necessário uso de hardwares convencionais de computadores para validar os resultados. Como armazenamento é utilizado um cartão SD de 8GB, que mostra-se mais que o suficiente para os sistemas operacionais gerados pelo Buildroot.

4.2.2.1 *libgpiod*

Para comunicar-se com os pinos GPIO no Linux é utilizada a *libgpiod*, uma biblioteca e conjunto de ferramentas escritos na linguagem C para interagir com pinos GPIO através de *character device drivers* do Linux, onde "*gpiod*" significa "*GPIO device*" [49]. Hospedada no site do kernel do Linux (*kernel.org*), ela oferece uma forma mais moderna de interagir com a GPIO através de *device drivers*, após a até então utilizada interface sysfs ser depreciada na versão 4.8 do Linux. A *libgpiod* encapsula chamadas de sistema (*system calls*), como a `ioctl()`, numa API própria e bastante intuitiva de ser usada, possibilitando utilizar os pinos como entrada, saída e na função de interrupção de hardware. As interrupções em um pino GPIO podem ser configuradas para serem ativadas na borda de subida, na borda de descida ou em ambas. A *libgpiod* também agrega ferramentas de linha de comando para trabalhar com os pinos GPIO disponíveis no sistema diretamente pelo terminal.

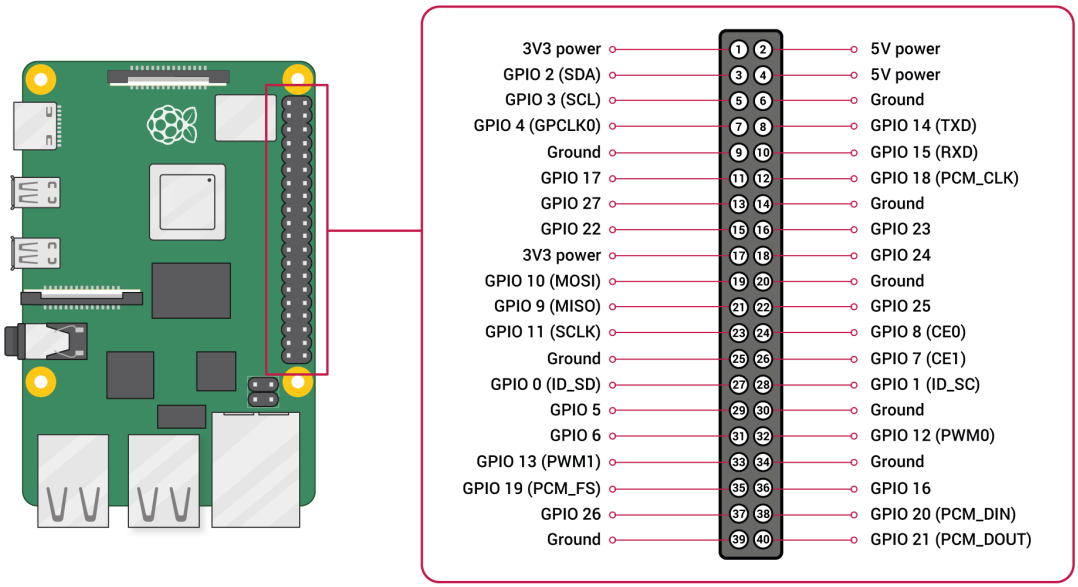

Figura 11 – Diagrama dos pinos GPIO da Raspberry Pi.

(Fonte: [48].)

4.2.3 Estresse do sistema operacional

A avaliação de desempenho centralizada em uma única aplicação em execução é útil para verificar qual é o melhor caso de atuação de um sistema operacional. Essa abordagem, porém, muitas vezes resultará de poucos recursos do sistema sendo utilizados, com requisições aos dispositivos quase que exclusivamente reservadas à aplicação de testes. Depender única e exclusivamente da análise do melhor caso pode resultar em falsos positivos. Aplicar estresse de uso dos recursos de um sistema operacional é muito importante, principalmente quando se tratando de tempo real. Ao executar tarefas de tempo real, com prioridades de tempo real, deseja-se que elas tenham preferência sobre as tarefas normais do sistema.

O estresse pode ser implementado como uma ou mais tarefas que requerem algum recurso específico do sistema operacional ou realizando funções que englobam várias requisições diferentes. Os testes de carga ou estresse podem incluir o uso da CPU, a alocação de memória, escritas na cache, escritas e leituras do disco, uso da rede, tratamento de interrupções internas e externas, etc. A carga aplicada ao sistema poder ser total, de modo a encher um determinado recurso em 100%, ou parcial.

4.2.3.1 stress-ng

Neste trabalho os casos de teste com carga utilizam-se da stress-ng, uma ferramenta usada exclusivamente para estressar sistemas operacionais. *A stress-ng* tinha o propósito

original de verificar *bugs*, travamentos e possíveis superaquecimentos no sistema sob determinadas condições extremas de uso, mas mostrou-se muito relevante para ser agregado em testes de desempenho [50]. Essa ferramenta possui um conjunto muito grande de opções para aplicar carga nos mais diversos âmbitos do sistema. A stress-ng utiliza-se de *workers* ("trabalhadores"), isto é, processos separados e responsáveis por estressar uma parcela do recurso o qual foi especificado. Toma-se como exemplo a execução do comando no terminal do sistema: `stress-ng --cpu 2 -hdd 1 --hdd-bytes 1G`. Nesse caso, serão criados dois processos do stress-ng focados no estresse da CPU, cada um ocupando aproxidamente 25% do uso do processador (50% no total), e um processo que fará escritas, leituras e remoções de arquivos temporários de até 1GB.

4.3 Sistema proposto

Para a computação e coleta dos dados de interrupções nos sistemas Linux usa-se a configuração proposta na Figura 12. Duas portas GPIO da Raspberry Pi, GPIO22 e GPIO24, são conectadas a duas portas digitais (GPIO) D5 e D6 do NodeMcu, respectivamente. As interrupções são geradas pelo NodeMcu na porta D6 e recebidas na Raspberry Pi através do pino GPIO24, enquanto que as respostas, também recebidas como interrupção no NodeMcu, são enviadas pela porta GPIO24 ao pino D5. Como terceiro componente é utilizado um computador para receber os dados do NodeMcu pela porta serial USB, já que a capacidade de memória e armazenamento da placa são limitados para a quantidade de dados levantada.

A Figura 13 exibe um diagrama com o funcionamento geral do sistema proposto. O NodeMcu é encarregado por alternar o valor da saída digital D6 entre os valores *high* (3.3v) e *low* (0v) a cada $1ms$, enquanto que a aplicação no Linux é configurada para ler interrupções de borda de subida e descida no pino GPIO24. Quando a aplicação do Raspberry Pi detecta a mudança do sinal, da mesma maneira o pino GPIO22 tem seu valor alternado (entre 0v e 3.3v), o que representa a confirmação do tratamento da interrupção. Para tal, o NodeMcu também configura o pino D5 como interrupção para receber as confirmações do sistema Linux. Caso a Raspberry Pi demore mais que $1ms$ para enviar a resposta, o NodeMcu atrasa a mudança do valor do pino D6 até que receba a confirmação. O tempo de $1ms$ é escolhido de forma arbitrária, que é considerado suficientemente grande para o NodeMcu realizar todas as operações necessárias até o próximo intervalo de interrupções. O valor de $1ms$ é também classificado como *deadline* da aplicação; caso a resposta da Raspberry Pi 3 demore mais que $1ms$, tal condição será classificada como perda da *deadline* para fins de avaliação. Caso um dos sistemas consiga atender à todos os prazos de $1ms$, então ele é categorizado como capaz de tempo real *hard* nesse contexto em que é aplicado.

Cada vez que o NodeMcu alterna a tensão no pino D6, ele também registra o tempo,

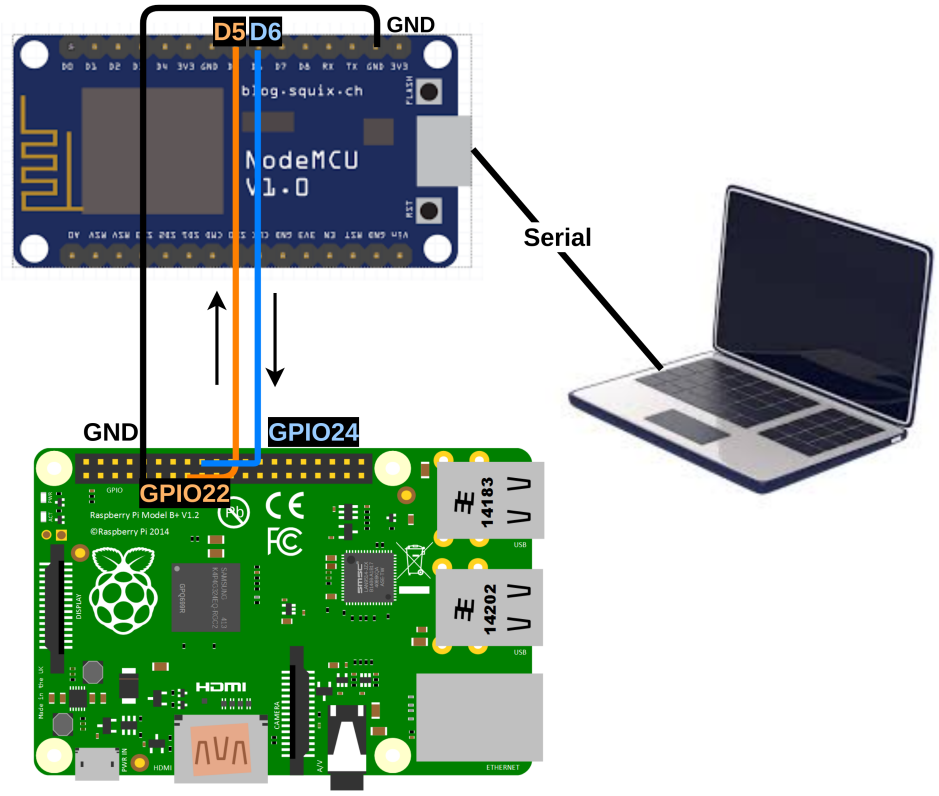

Figura 12 – Esquemático simplificado do sistema.

(Fonte: do autor)

em microssegundos. Quando há o recebimento da resposta no pino D5 pelo tratamento de interrupções, o tempo igualmente é guardado. Possuindo ambos os valores temporais de envio e de recebimento, o NodeMcu calcula a diferença deles e escreve o resultado na interface serial, que está conectada ao computador. Cada latência é enviada como uma nova linha ao incluir o caractere `'\n'` no final da *string*, e será escrito em um arquivo de texto no computador. O computador escuta no terminal serial por tempo indeterminado, da mesma forma que a aplicação do Linux na Raspberry Pi espera indefinidamente por interrupções até que seja encerrada manualmente. O NodeMcu, entretanto, encerra o envio de interrupções depois de transcorrida uma hora de execução. O valor de uma hora é escolhido pois a função `micros()` da API do Arduino sofre *overflow* (reinicia sua contagem do zero) em aproximadamente 70 minutos após o inicialização do NodeMcu.

É importante ressaltar que o tempo contabilizado como latência de interrupção no Linux soma consigo todos os atrasos apresentados pelo próprio NodeMcu. Esse é um problema intrínseco de se fazer uso de uma plataforma não destinada ao processamento de sinais. Espera-se que esse atraso apresente pouca variação, dado que a aplicação do NodeMcu é executada sem sistemas operacionais ou algoritmos de escalonamento, e sim em *bare-metal*, bem como a as atividades que computam os tempos é bastante simples. Idealmente, no lugar do NodeMcu seria utilizado um gerador funções, para o envio de

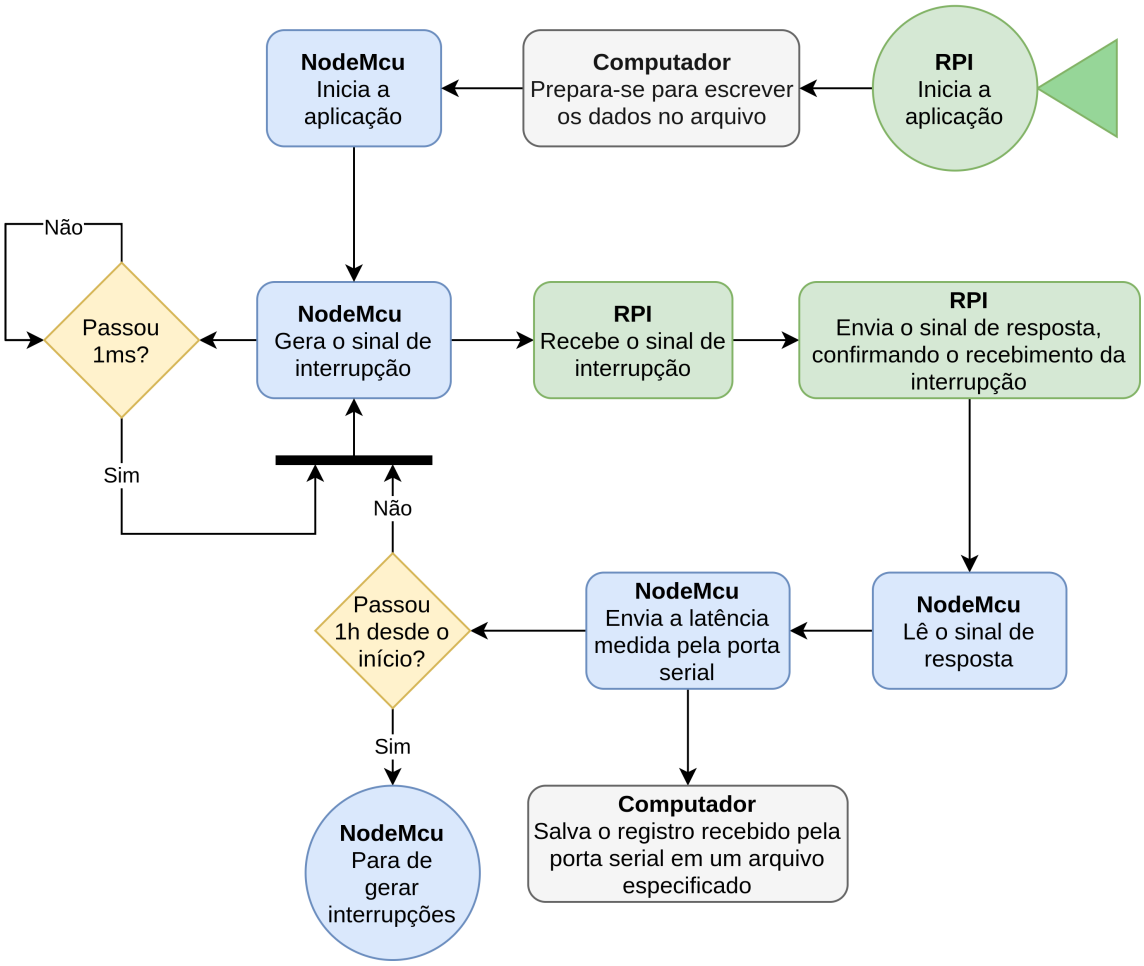

Figura 13 – Diagrama da aplicação de testes.

(Fonte: do autor)

interrupções, e um osciloscópio, designado à contabilização das latências.

4.3.1 Casos de teste

Para a avaliação de desempenho das abordagens do Linux de tempo real são comparados os valores de latência mínimos, máximos, médios e também de desvio padrão de todo o conjunto de amostras. As operações inicialmente são feitas nos sistemas sem nenhuma carga, que executa apenas a aplicação incumbida de lidar com as interrupções. A análise do sistema operacional sem carga, entretanto, não representa com precisão os casos reais, onde por vezes tem-se mais de uma tarefa em execução disputando o tempo do processador.

Para simular casos extremos é feita uma segunda bateria de testes em todos os sistemas fazendo uso da ferramenta stress-ng. A stress-ng é executada de forma a estressar principalmente a CPU, memória e a cache da Raspberry Pi 3, mas sem estar atrelada a prioridades de tempo real. Isso possibilita que o valor de prioridade atribuído à tarefa de

tratamento de interrupções em execução no Linux possa ser provado com maior profundidade. A adição de testes com estresse resulta em seis casos de avaliação, sendo dois para cada sistema operacional: um teste sem estresse e um outro sob carga (ver o Capítulo 5, refente aos resultados obtidos, para maiores informações).

4.4 Aplicações para a medição das latências no tratamento de interrupções

Essa seção trata de entrar em detalhes de implementação das aplicações utilizadas nos testes de desempenho. Aqui são abordados os três sistemas já comentados na Seção 4.3: o NodeMcU, que gera as interrupções e calcula as latências; a Raspberrry Pi 3, que executa os sistemas Linux e trata as interrupções recebidas; o computador, responsável por guardar as informações calculadas. Todos os códigos-fonte das aplicações, bem como dos Makefiles utilizados para a compilação, estão disponíveis na sua forma completa no Apêndice B.

4.4.1 Aplicação do NodeMcu

A aplicação do NodeMcu é feita com o ambiente e bibliotecas do Arduino. A função responsável por enviar as interrupções à Raspberry pode ser vista no trecho de código que segue. Ela escreve no pino de saída (D6) o valor de tensão presente na variável `value`, que alterna seu conteúdo entre 0 e 1 cada vez que a função é chamada. Cada mudança de valor tem o tempo marcado em `time_sent`, em microssegundos, e é interpretada como uma interrupção ao ser captada no Linux. Através de `can_send_interrupt` controla-se quando a próxima interrupção pode ser enviada.

```
void send_interrupt()
{
    digitalWrite(led_pin, value);
    time_sent = time_now;
    value = !value;
    can_send_interrupt = false;
}
```

Para recebimento da resposta do Linux, configura-se o pino D5 como interrupção de borda de subida e de descida, referenciada por `CHANGE`, através de `attachInterrupt()`. A função recebe como argumentos o numero pino digital desejado, uma função especial usada durante o tratamento de interrupção e o modo de ativação (`CHANGE`, nesse caso).

```
  attachInterrupt(digitalPinToInterrupt(interrupt_pin),
   interrupt_handler,
        CHANGE);
```

A rotina de tratamento de interrupção (ISR) é responsável por guardar o tempo de recebimento em microssegundos e setar uma variável de estado indicando que houve o tratamento. O tratamento de interrupção é o mais simples possível, fazendo apenas o essencial, para que a execução seja feita principalmente no `loop()`. A função `micros()` retorna o tempo em microssegundos decorrido desde que o NodeMcu foi ligado, com precisão de $1\mu s$. A ISR deve ser implementada com o macro `ICACHE_RAM_ATTR`, indicando que a interrupção é guardada na RAM. Essa opção é importante no gerenciamento de interrupções, pois por padrão funções são guardadas no armazenamento *flash*. Sabe-se a RAM é acessada muito mais rapidamente que a *flash*, portanto essa opção torna o gerenciamento de interrupções mais responsivo.

```
ICACHE_RAM_ATTR void interrupt_handler()
{
  time_received = micros();
  triggered_isr = true;
}
```

O tempo mínimo do envio de interrupções é de 1 milissegundo.

```
unsigned long interval = 1000; // 1ms
```

O trecho a seguir refere-se ao laço de execução do programa. O primeiro bloco `if` é executado caso uma interrupção tenha ocorrido, situação essa que é marcada por `triggered_isr`, como visto na implementação da ISR. Nessa parte é computada a diferença dos tempos do acionamento da interrupção e recebimento da resposta coletados, bem como o valor é escrito na porta serial – seguido de uma nova linha. A variável `triggered_isr` é então redefinida, bem como a variável de controle `can_send_interrupt` passa a indicar que uma nova interrupção pode ser enviada. Essa condição é verificada no próximo bloco juntamente com a checagem da passagem de tempo. Caso a expressão retorne verdadeiro, uma nova interrupção é gerada.

```
void loop()
{
  if (triggered_isr)
  {
    int time_diff = time_received - time_sent;
    triggered_isr = false;
    can_send_interrupt = true;
    Serial.print(time_diff);
    Serial.print('\n');

  }

  time_now = micros();
  if (can_send_interrupt && (time_now - time_sent >= interval))
  {
```

```
      send_interrupt();
  }

  if (time_now >= time_start + time_to_stop)
  {
    stop();
  }
}
```

A interface serial é configurada com *baudrate* de 115200, indicando o envio de 115200 bits por segundo ao computador:

```
Serial.begin (115200);
```

Por fim, caso tenha-se passado uma hora ou mais de execução desde que o NodeMcu iniciou seu funcionamento o sistema é parado através da função `stop()`. A tempo de parada é definido em microssegundos:

```
unsigned long time_to_stop = 60UL * 60UL * 1000000UL;
```

A função `stop()` limpa configuração de interrupção no pino digital D5 e entra em um estado de espera ocupada através de chamadas em *loop* da função `delay()`.

```
void stop()
{
  detachInterrupt(digitalPinToInterrupt(interrupt_pin));
  digitalWrite(led_pin, LOW);
  while(1) { delay(1000); }
}
```

Esse programa não faz uso de nenhuma biblioteca adicional além do conjunto padrão de funções disponíveis através da IDE do Arduino. A aplicação pode ser reiniciada através do botão *reset* da placa, uma vez que o programa esteja carregado no NodeMcu. De fato, o NodeMcu é a última parte a entrar em execução pelo fato de ser o dispositivo que conduz os testes. Os outros sistemas devem estar preparados antes de ser dado o início na aplicação do NodeMcu.

4.4.2 Tratamento de interrupções nos sistemas Linux

Nessa seção são exibidas as aplicações executadas no Linux que é executado na Raspberry Pi 3. Uma aplicação é específica para as versões Vanilla e RTLinux, que compartilham bibliotecas, enquanto que a outra é escrita para o ambiente do Xenomai na configuração de kernel duplo (fazendo uso do Cobalt). Essa aplicações são inicialmente executadas com o sistema livre e após isso rodam em conjunto com a ferramenta stress-ng.

4.4.2.1 Linux Vanilla e Real-Time Linux

Como apresentado na Seção 3.3.3, não é necessária modificação alguma no código-fonte de um programa para trazê-lo do Linux Vanilla para o RTLinux, e vice-versa. Isso permite que a aplicação de testes seja exatamente a mesma para esses dois sistemas. A aplicação baseia-se no uso da biblioteca *libgpiod* para manipulação dos pinos GPIO do sistema Linux, inclusa através do arquivo de cabeçalho `gpiod.h`. Para disponibilizar a *libgpiod* na compilação, é necessário passar a *flag* `-lgpiod` (Apêndice B.3.1) ao compilador.

A tarefa principal é implementada na função `led_interrupt()`, criada dentro dos padrões das *POSIX Threads* disponíveis no Linux [51]. Inicialmente, configura-se a prioridade de tempo real da tarefa como 99, isto é, o valor máximo possível, com modo de escalonamento Round-Robin. O início do laço de repetição é caracterizado pela chamada da função `gpiod_line_event_wait()`, que permanece em espera por interrupções por tempo indeterminado.

```
void *led_interrupt(void *arg)
{
    int ret, value = 1;
    struct sched_param param = { .sched_priority = 99 };
    ret = sched_setscheduler(getpid(), SCHED_RR, &param);

    while (1)
    {
        ret = gpiod_line_event_wait(interrupt, NULL);
        if (ret > 0)
        {
            ret = gpiod_line_event_read(interrupt, &event);
            // ... tratamento de erros pelo valor de ret ...
            ret = gpiod_line_set_value(led, value);
            // ... tratamento de erros pelo valor de ret ...
            value = !value;
        }
    // ... se ret <= 0 (erro) ...
    }
}
```

Uma vez que a interrupção tenha sido gerada, as respectivas informações são guardadas em `interrupt`. Caso nenhum erro ocorra, escreve-se na variável `event` os dados da interrupção lida, cujo valor não é usado nessa aplicação. A resposta é enviada pela escrita no pino GPIO22, configurado como saída, que alterna seu valor no final de cada laço. O tratamento dos erros, que não é o foco desse trabalho, é propriamente representado nos códigos-fonte no Apêndice B.

O *chip* responsável pela interface GPIO pode ser verificado na pasta `/dev` sob o nome `/dev/gpiochip*`. Na Raspberry Pi 3 ele é representado por `gpiochip0`:

```
const char *chipname = "gpiochip0";
```

A configuração dos pinos pode ser vista a seguir. Inicialmente, acessa-se o *chip* GPIO, que dá acesso à manipulação do funcionamento dos pinos pelas funções `_get_line()`. O pino GPIO22, `led_pin`, é então configurado como saída, enquanto que o pino GPIO24, `interrupt_pin`, como interrupção. O acionamento das interrupções é requisitado para ocorrer em ambas as transições de sinal na porta GPIO24, isto é, nas trocas de 0v para 3.3v ou de 3.3v para 0v (valores aproximados).

```
chip = gpiod_chip_open_by_name(chipname);
// ... tratamento de erros pelo valor de chip ...
led = gpiod_chip_get_line(chip, led_pin);
// ... tratamento de erros pelo valor de led ...
interrupt = gpiod_chip_get_line(chip, interrupt_pin);
// ... tratamento de erros pelo valor de interrupt ...
ret = gpiod_line_request_output(led, CONSUMER, 0);
// ... tratamento de erros pelo valor de ret ...
ret = gpiod_line_request_both_edges_events(interrupt, CONSUMER);
// ... tratamento de erros pelo valor de ret ...
```

A tarefa `task_interrupt()` é inicializada na função `main()` como uma nova *thread* ao ser especificada a função `led_interrupt()`. Como *threads* no Linux são também processos, o que está sendo feito é isolar a execução do tratamento de interrupções em um processo separado com alta prioridade de tempo real. O segundo parâmetro, passado como `NULL`, determina atributos adicionais para a *thread*, enquanto que o último parâmetro indica um ponteiro para os dados usados na função – não utilizado nessa aplicação. Para utilizar as *POSIX Threads*, usa-se a *flag* `-lpthread` (Apêndice B.3.1) na compilação.

```
pthread_create(&task_interrupt, NULL, led_interrupt, NULL);
```

Chama-se também em `main()` a função `mlockall()`, que através do parâmetros recebidos previne que as páginas de memória alocadas, atuais e futuras, sofram *swap* [52]. *Swap* é o ato de armazenar parte do conteúdo da RAM na unidade de armazenamento permanente do sistema para liberar espaço de memória. O swap, todavia, é indesejável no contexto de tempo real, pois escritas no disco tornam a aplicação mais lenta e imprevisível temporalmente. O *swap* é mais mais crítico ainda no contexto de sistemas que dispõem de armazenamento *flash*, como a Raspberry Pi 3, onde a muitas escritas no disco significam redução da sua vida útil.

```
mlockall(MCL_CURRENT | MCL_FUTURE);
```

4.4.2.2 Xenomai

A aplicação de tempo real no xenomai utiliza-se das três APIs disponíveis: Alchemy, POSIX e RTDM. A função `led_interrupt()`, mostrada a seguir, tem o mesmo funciona-

mento da função de mesmo nome presente na aplicação feita com a *libgpiod* para o Linux Vanilla e RTLinux. O laço de execução consiste em inicialmente esperar pela interrupção no pino GPIO24 (`pin_isr`) através da função `pin_read()`, que nessa situação é bloqueante pelo pino ser configurado como tal. Uma vez recebida a interrupção, escreve-se no pino GPIO22 o valor 0 ou 1, que é invertido a cada iteração.

```
void led_interrupt(void *arg)
{
    // ... setup ...
    while (1)
    {
        pin_read(pin_isr);
        pin_write(pin_out, value);
        value = !value;
    }
}
```

A as funções `pin_write()` e `pin_read()` são envelopes ao redor *syscalls* da API POSIX `write()` e `read()`, que fazem o real trabalho de tempo real. Ao realizar a compilação, deve-se utilizar a *flag* `--posix` (Apêndice B.3.2), que transforma as chamadas de sistema POSIX em chamadas do Xenomai, de tempo real. As *syscalls* `read()` e `write()`, como pode ser subentendido, servem para ler e escrever valores nos pinos digitais. Sem a indicação dessa *flag*, essas chamadas serão tratadas como funções padrão do modo secundário, ou seja, serão executadas no kernel do Linux e sem especificações de tempo real.

```
void pin_write(int pin, int value)
{
    int ret, error_code;
    ret = write(pin, &value, sizeof(value));
    // ... tratamento de erros pelo valor de ret ...
}

int pin_read(int pin)
{
    int ret, error_code, value;
    ret = read(pin, &value, sizeof(value));
    // ... tratamento de erros pelo valor de ret ...
    return value;
}
```

A definição dos pinos também é feita totalmente pela interface de sistema de arquivos virtuais, bastante conhecida no Linux. O caminho especificado nas chamadas de sistema de `open()` refere-se ao acesso dos pinos GPIO através da interface *Real-Time Driver Model* (RTDM), do Xenomai. A interface do RTDM torna-se disponível ao incluir `rtdm/gpio.h` (Apêndice B.2.2) na aplicação e a flag `--rtdm` (Apêndice B.3.2) no seu respectivo Makefile. Os arquivos `gpio2016` e `gpio2018` referem-se aos pinos GPIO22 e GPIO24, respectiva-

mente. O pino de saída `pin_out` (GPIO22) é aberto com o modo `O_WRONLY`, indicando que a chamada de sistema `open()` – a qual é convertida para a chamada de sistema do Cobalt – retorna um descritor de arquivo que permite apenas a escrita de valores no pino. A chamada de sistema `ioctl()`, abreviação de *input/output control*, é empregada para quando são necessárias requisições de configuração mais específicas, além do que as chamadas de sistema básica podem oferecer. Esse é o caso dos pinos utilizados pela interface do Xenomai, cujas chamadas `read()` e `write()` não unicamente leem e escrevem em arquivos do sistema, mas configuram o hardware físico (pinos GPIO).

```
pin_out = open("/dev/rtdm/pinctrl-bcm2835/gpio2016", O_WRONLY);
 // ... tratamento de erros pelo valor de pin_out ...
ret = ioctl(pin_out, GPIO_RTIOC_DIR_OUT, &value);
// ... tratamento de erros pelo valor de ret ...
ret = write(pin_out, &value, sizeof(value));
// ... tratamento de erros pelo valor de ret ...
pin_isr = open("/dev/rtdm/pinctrl-bcm2835/gpio2018", O_RDONLY);
// ... tratamento de erros pelo valor de pin_isr ...
int xeno_trigger = GPIO_TRIGGER_EDGE_RISING | GPIO_TRIGGER_EDGE_FALLING;
ret = ioctl(pin_isr, GPIO_RTIOC_IRQEN, &xeno_trigger);
// ... tratamento de erros pelo valor de ret ...
```

Além de ser configurado como apenas leitura (`O_RDONLY`), o pino `pin_isr` (GPIO24) também recebe parâmetros adicionais através da chamada `ioctl()`. Em adição de ser especificado como interrupção, através do parâmetro adicional `xeno_trigger` especifica-se que o tratamento de interrupção ocorre em ambas as bordas de subida e descida.

A criação e inicialização da tarefa responsável pelo tratamento das interrupções é feita em dois estágios por meio da interface Alchemy, incluindo o cabeçalho `alchemy/task.h` (Apêndice B.2.2) e a *flag* `--alchemy` (Apêndice B.3.2) durante a compilação. Primeiramente, a tarefa é criada ao passar o descritor do tipo `RT_TASK` que identifica a tarefa, o nome da tarefa, o tamanho da pilha de dados (que quando atribuído como 0 deixa o sistema responsável por configurar um valor adequado), o valor da prioridade de tempo real e também o modo de operação. A prioridade de tempo real assume o valor máximo, nesse caso, que pode variar entre 0 e 99. O valor 0 para o modo de operação significa que a função `rt_task_join()` pode ser chamada para esperar o término da execução da tarefa.

```
RT_TASK task_interrupt;
ret = rt_task_create(&task_interrupt, "Interrupt task", 0, 99, 0);
// ... tratamento de erros pelo valor de ret ...
```

O início da execução da tarefa se dá por uma nova chamada da interface do Alchemy, bastante parecida com o que é encontrado no POSIX. Aqui, especifica-se o descritor da tarefa, o endereço da função que deve ser executada e um ponteiro para dados que serão passados para a função (nesse caso o valor 0 indica um ponteiro nulo).

```
ret = rt_task_start(&task_interrupt, &led_interrupt, 0);
// ... tratamento de erros pelo valor de ret ...
```

4.4.3 Estresse dos sistemas

No modo de testes sob estresse, a ferramenta stress-ng é executada no terminal do sistema da seguinte forma antes de ser dado início à aplicação de interrupções:

```
$ nice -19 stress-ng --cpu 4 --vm 1 --vm-bytes 128M&
```

O uso de `nice -19` indica que o processo inicia-se com valor *nice* de -19, significando que ele possui a maior **prioridade normal** possível. Usa-se prioridade normal (valor *nice*) e não de tempo real pois não deseja-se que o processo stress-ng concorra em igualdade com a tarefa de gerenciamento de interrupções, mas que seja interpretado como um processo normal e custoso do sistema. O caractere `'&'` no final do comando especifica que o processo é executado em segundo plano a partir da sessão do *shell* atual.

São inicializados quatro *workers* para estresse da CPU fazendo com que o uso do processador seja de 100% (aproximadamente 25% de uso em cada processo). Esses *stressors* (processos de estresse) iteram sobre diversas operações diferentes de forma cíclica, algumas mais atreladas à CPU, outras mais abrangentes quanto ao alvo de estresse [50]. Como a aplicação de tratamento de interrupções ocupa muito pouco do processador, quase todo o uso é designa à ferramenta stress-ng. Em acréscimo, cria-se um *worker* para manipulação da memória virtual, que faz alocações e liberações de memória de até 128MB. Considerando que a Raspberry Pi 3 tem 1GB de RAM, 128MB de memória alocada tornam-se bastante relevantes.

As três principais formas de estresse desejadas são: CPU, memória e cache. O uso do disco é descartado nos testes realizados, pois dificilmente aplicações de tempo real requerem que o acesso ao disco tenha determinismo temporal. Há também a questão de o armazenamento *flash* da Raspberry Pi 3 ser limitado (8GB), onde muitas escritas sucessivas podem diminuir a vida útil do cartão SD.

Na configuração de CPU e memória virtual a cache também é bastante estressada, apesar de não ser explicitamente especificada. O estresse da cache está relacionado principalmente ao aumento da ocorrência de *cache misses*, situação em que um dado não é encontrado na cache e o sistema necessita buscá-lo na memória. O acontecimento de muitos *cache misses* aumenta a lentidão do sistema, que precisará acessar muito mais vezes a memória (cujo acesso é mais lento que a cache).

Para finalizar o todos os processos stress-ng na árvores de processos do sistema, executa-se no terminal:

```
$ killall stress-ng
```

4.4.4 Computador

O computador adotado para o recebimento dos dados pelo porta serial possui um sistema operacional Linux. Os comandos apresentados, portanto, devem ser executados em um computador com Linux nativo ou em uma máquina virtual Linux.

4.4.4.1 Recebimento dos dados

Se o NodeMcu for o único dispositivo cujo console serial estiver conectado no computador, ele estará localizado em `/dev/ttyUSB0`. Primeiro é necessário configurar a mesma *baudrate* usada no NodeMcu para que haja sincronia entre os dois lados da comunicação:

```
$ stty -F /dev/ttyUSB0 115200
```

Configura-se também o modo como os dados serão interpretados para serem escritos no arquivo. A opção `raw` indica que os bits recebidos não devem ser processados ao serem lidos, nem mesmo caracteres de controle, como `^C` ou `^Z`:

```
$ stty raw < /dev/ttyUSB0
```

Por fim, redireciona-se o conteúdo da porta serial para o arquivo de texto especificado, nesse caso `~/output/output_file.txt`. O comando é executado de forma contínua, bloqueando o terminal, até que o usuário entre o sinal de controle `^C` (*Ctrl+C*) para encerrar o processo:

```
$ cat /dev/ttyUSB0 > ~/output/output_file.txt
```

4.4.4.2 Acesso à Raspberry Pi 3

O acesso à Raspberry Pi 3 é feito sem o uso de teclado ou monitor, mas somente através de conexão SSH (*Secure Shell*). Esse modo de acesso também é conhecido como *headless*, onde o dispositivo é acessado através da rede. A Raspberry Pi é acessada via SSH, conectada à rede local através do adaptador *ethernet* presente na placa, para dar início à stress-ng (quando necessário) e a aplicação de tratamento de interrupções, bem como para finalizá-las. Para conectar à Raspberry Pi via SSH considerando que o usuário é *root*, faz-se:

```
$ ssh root@<ip da RPI>
```

Para encontrar o endereço ipv4 da Raspberry Pi na rede local, caso ele não tenha sido estaticamente configurado, é possível acessar o roteador e checar a lista de distribuição dos endereços pelo servidor DHCP. Pode-se também fazer uso do `nmap` (instala-se no Ubuntu através de: `sudo apt install nmap`) para listar todos os endereços da rede especificada [53]:

```
$ nmap -sn <endereço da rede ou roteador padrão>/24
```

5 Resultados

Neste capítulo são apresentados e analisados os resultados dos testes aplicados a cada um dos sistemas operacionais Linux propostos. Os três sistemas sob análise – o Linux Vanilla, o Real-Time Linuix e o Xenomai – foram construídos e testados a partir das mesmas fontes, que é o kernel do Linux na versão 4.19.144 compilado para a Raspberry Pi 3. Na Seção 5.1 são avaliadas as latências e o *jitter* dos sistemas, enquanto que a Seção 5.2 aprofunda-se no conceito de *deadline* e como ela foi respeitada por cada um deles. A Seção 5.3 adiciona a análise da latência do próprio NodeMcu, que não é avaliada nas duas primeiras seções.

5.1 Avaliação das latências e do *jitter*

Na presente seção são avaliadas as latências obtidas nas baterias de testes. Os dados usados para fins de análise são: as latências mínima e máxima, a latência média do conjunto e o desvio padrão. A média, ainda que relevante, não é o fator crítico para classificar um sistema capaz de servir tarefas de tempo real. Um valor médio de latências acima da *deadline* pode traduzir-se, juntamente a avaliação da latência máxima, na necessidade de aumentar a *deadline*. Nesse caso, agregar a análise do desvio padrão é necessário para apontar o quão dispersas estão as latências calculadas.

Como os dados colhidos utilizaram-se da função `micros()` do NodeMcu para registrar o tempo, a precisão das amostras únicas é de $1\mu s$. A duração de cada ciclo de testes é de uma hora, onde cada sistema operacional foi testado com e sem o uso da ferramenta stress-ng. Assim, são verificados seis casos de teste no total. Como o tempo é fixado em uma hora o número de amostras colhidas varia entre cada conjunto. A máxima quantia possível em cada sessão, julgando o tratamento de latências instantâneo no NodeMcu e no Linux, é de 3.6×10^6 interrupções ocorridas, de acordo com a Equação (5.1). Lembrando que o período mínimo de envio de interrupções no NodeMcu é de $1ms$.

$$\frac{\textit{tempo total dos testes}}{\textit{período de envio de interrupções mín.}} = \frac{1\ \textit{hora}}{1\ \textit{milissegundo}} = 3.6 \times 10^6\ \textit{amostras} \quad (5.1)$$

5.1.1 Testes sem estresse

A Tabela 1 exibe os dados colhidos para o cenário sem a aplicação de carga nos sistemas. Os resultados aqui avaliados representam o melhor desempenho possível desses sistemas, uma vez que a aplicação é executada sozinha no sistema e possui alta prioridade. De fato, ainda há processos em segundo plano ou em modo de baixo consumo da CPU.

Um exemplo disso é o *daemon* do servidor SSH em execução na Rasbperry Pi, que durante a execução dos testes não recebe interação do usuário.

Tabela 1 – Latência dos sistemas

Medidas	Sistema		
	Linux Vanilla	Real-Time Linux	Xenomai
Latência mínima (μs)	33.0	34.0	17.0
Latência máxima (μs)	114.0	346.0	38.0
Latência média (μs)	40.1	46.8	18.4
Desvio padrão (μs)	1.6	4.4	0.59
Número de amostras	3570843	3571109	3568593

O Xenomai foi o sistema operacional que desempenhou melhor, dispondo do menor desvio padrão e média de latência. A Figura 14 mostra o histograma das latências obtidas nesse sistema. Pode-se observar que o intervalo que contêm todas as latências do Xenomai é bastante pequeno; o atendimento das interrupções sempre se situou entre $17\mu s$ e $38\mu s$. O baixo valor mínimo das latências em comparação com os outros dois sistemas pode estar relacionado às bibliotecas do Cobalt serem menos custosas que as do Linux para o atendimento de interrupções ao lançar mão do I-pipe como gerenciador da fila de interrupções.

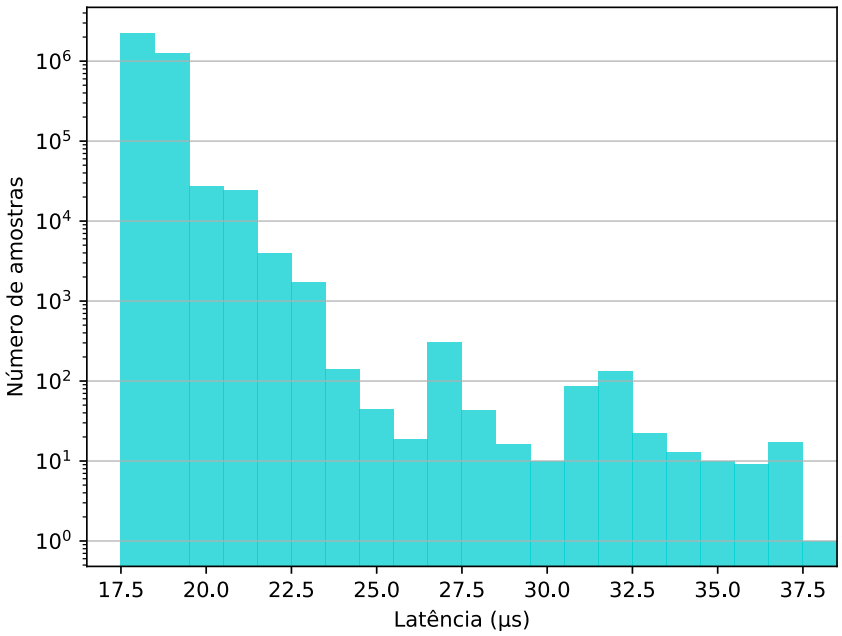

Figura 14 – Atendimento de interrupções no Xenomai.

(Fonte: do autor.)

Surpreendentemente, o segundo melhor sistema foi o Linux Vanilla, e não o RTLinux, o qual é uma modificação do Vanilla com foco em tempo real. O porquê de o RTLinux, concebido lidar com o problemas de atividades não-preemptivas no kernel, apresentar desempenho pior que o Vanilla não pôde ser identificado. O tempo máximo de atendimento de interrupções no Vanilla é aproximadamente um terço do valor máximo do RTLinux. O mesmo acontece para o desvio padrão (interpretado como o *jitter*), que também é quase um terço do desvio padrão do RTLinux. A distribuição das latências do Vanilla pode ser vista na Figura 15.

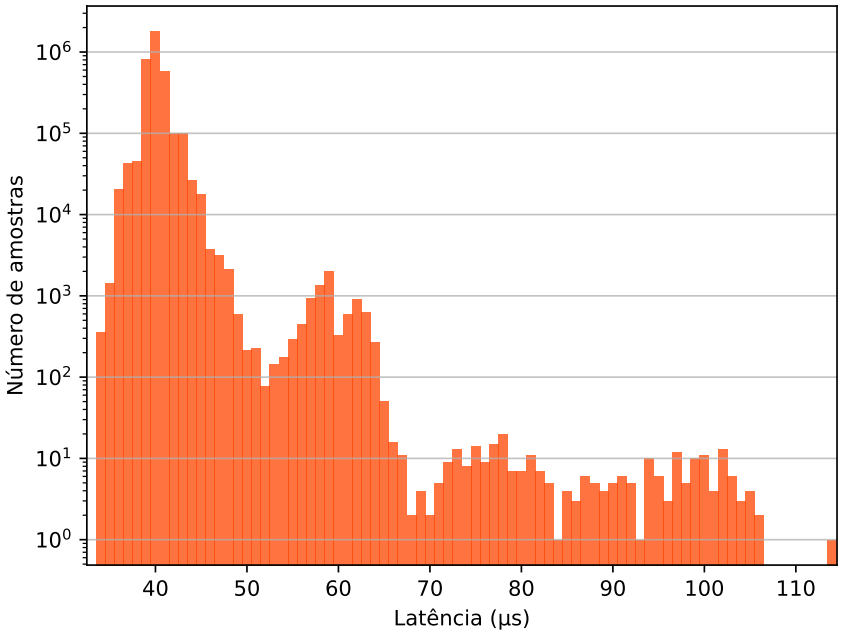

Figura 15 – Atendimento de interrupções no Linux Vanilla.

(Fonte: do autor.)

O Real-Time Linux apresentou o pior desempenho dos três (Figura 16). A latência máxima encontrada no sistema pode estar relacionada com os processos secundários em execução, como discutido anteriormente. Porém, em sistemas de tempo real, processos em segundo plano não devem tomar a execução no processador em detrimento do atendimento de interrupções. Como pode ser visto na Seção 5.1.2, o Real-Time Linux apresenta melhor desempenho que o Linux Vanilla sob estresse.

5.1.2 Testes com estresse

Os ciclos de testes que fazem uso da stress-ng compõem o critério mais importante para aferir a capacidade de tempo real dos sistemas avaliados. A Tabela 2 possui os resultados

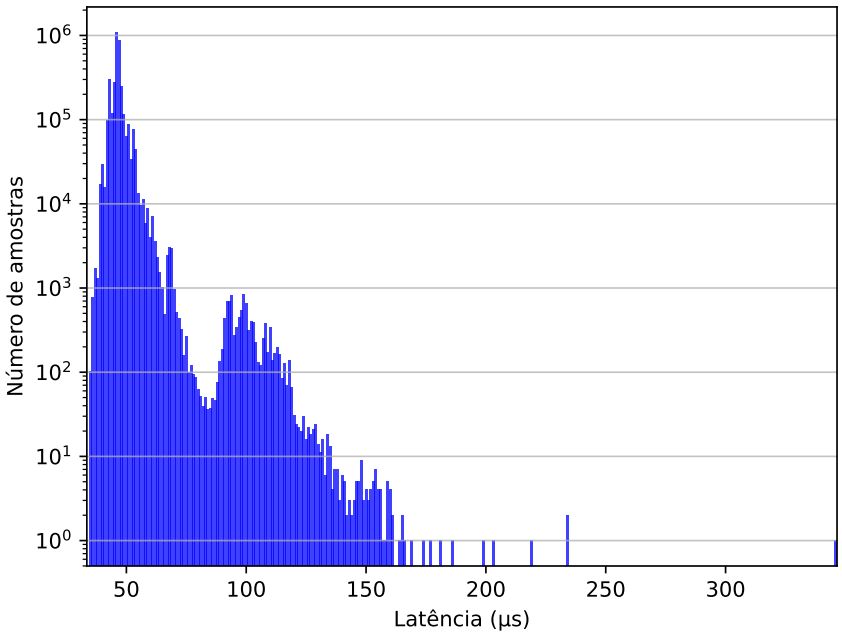

Figura 16 – Atendimento de interrupções no RTLinux.

(Fonte: do autor.)

dos testes efetuados quando há a presença de carga aplicada aos sistemas. A ferramenta stress-ng é inicializada e finalizado conforme os passos abordados na Seção 4.4.3.

Tabela 2 – Latência dos sistemas sob uso da ferramenta stress-ng

Medidas	Sistema		
	Linux Vanilla	Real-Time Linux	Xenomai
Latência mínima (μs)	31.0	35.0	18.0
Latência máxima (μs)	2974.0	1717.0	883.0
Latência média (μs)	63.5	67.6	37.9
Desvio padrão (μs)	70.2	39.4	36.5
Número de amostras	3572130	3572579	3570094

O Xenomai, como na Seção 5.1.1, teve os melhores resultados gerais no tratamento de interrupções, cujo histograma das latências calculadas no NodeMcu pode ser visto na Figura 17. Vale ressaltar o valor da latência máxima, de 883.0μs, a qual revela que o atendimento das interrupções no Xenomai nunca invadiu o próximo período de envio de interrupções no NodeMcu. O desvio padrão ficou próximo do RTLinux, indicando que a dispersão dos dados em ambos na condição de estresse foi similar.

O segundo melhor sistema foi o RTLinux (Figura 19). Embora ele exiba a maior média de latência (não muito superior ao Vanilla), esse valor não deve ser avaliado isoladamente. É preciso que a latência máxima e o desvio padrão sejam levados em consideração, uma

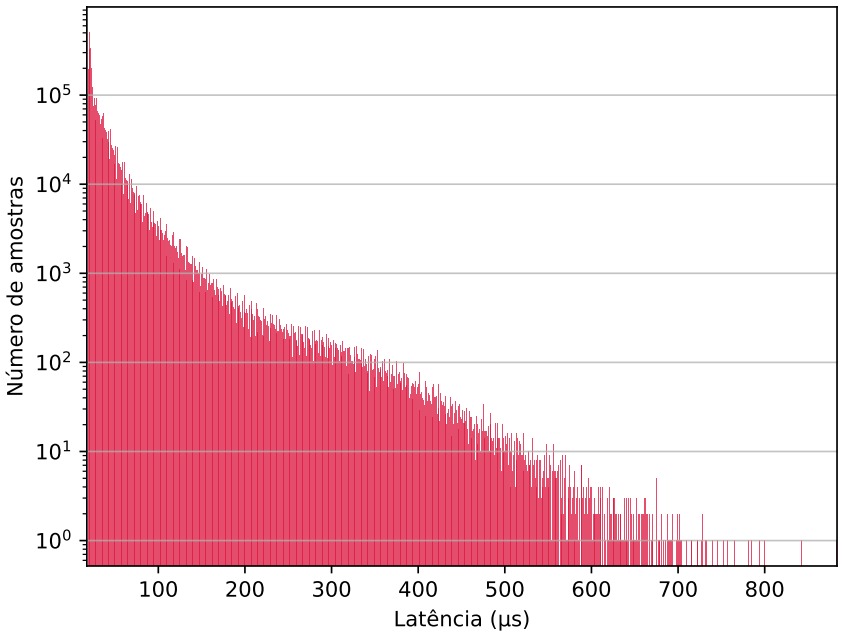

Figura 17 – Atendimento de interrupções no Xenomai com estresse aplicado ao sistema.

(Fonte: do autor.)

vez que o atendimento mais demorado do RTLinux é $1257\mu s$ inferior ao pico de atraso do Vanilla e o desvio padrão bastante próximo ao Xenomai, que mostra a menor variação dos três sistemas.

O sistema operacional que demonstrou ter o pior desempenho quando é feito uso da stress-ng foi o Linux Vanilla. A latência máxima, que pode ser identificada na Figura 18, é bem maior que nos outros sistemas, ainda que os altos valores de latência não sejam muito frequentes e a latência média seja menor que no RTLinux. O impacto dos altos valores de latência, as vezes dispersos do conjunto total de amostras, são melhor avaliados na Seção 5.2. O desvio padrão é aproximadamente $30\mu s$ maior que no RTLinux, significando menor estabilidade e precisão temporal (maior *jitter*) ao tratar as interrupções de hardware.

Pode-se perceber que quando há a presença de carga no sistema o valor mínimo pouco se alterou ao comparar os resultados com os testes sem aplicação de estresse. Isso provavelmente está relacionado com o fato de que apenas uma parte das atividades dos *workers* da stress-ng, presentes na opção `--cpu`, estressam a cache. Quando tarefas mais atreladas ao processador (com pouco acesso à RAM) estão em execução pela stress-ng, há espaço para o atendimento mais rápido das interrupções por parte da aplicação responsável por isso. Quando a cache está sendo pouco atualizada, ela provavelmente conterá as os dados frequentes e essenciais da aplicação de testes.

A abordagem de kernel duplo do Xenomai se mostrou ser a mais adequada a lidar

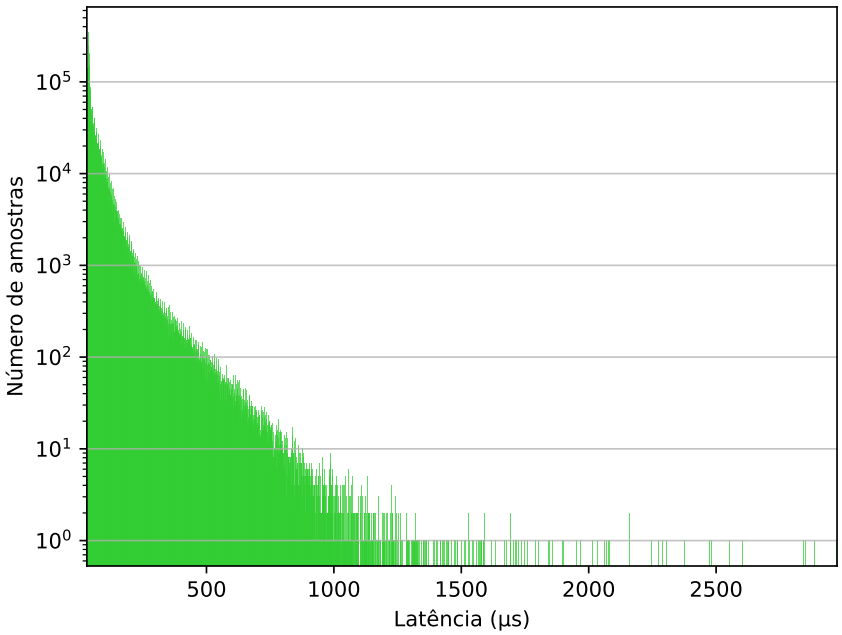

Figura 18 – Atendimento de interrupções no Linux Vanilla com estresse aplicado ao sistema.

(Fonte: do autor.)

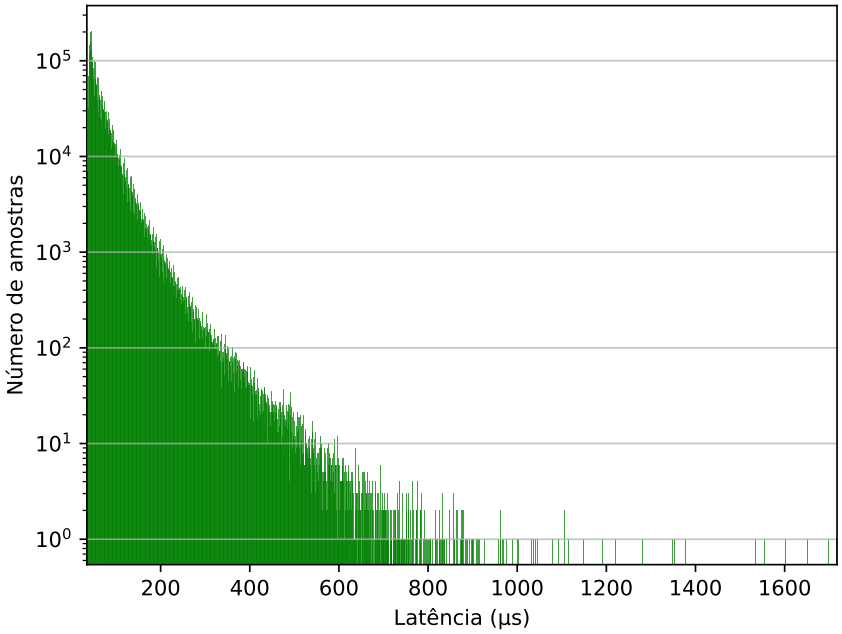

Figura 19 – Atendimento de interrupções no RTLinux com estresse aplicado ao sistema.

(Fonte: do autor.)

com grande concorrência de recursos. Isso de deve, muito provavelmente, por conta da presença do I-pipe, que concede a maior prioridade à tarefa que trata as interrupções (executada no Cobalt). Porém, o valor máximo de latência levanta suspeitas de que a cache foi suficientemente estressada ao ponto de gerar um valor máximo ($883.0\mu s$) bem acima do que foi encontrado quando não havia estresse no sistema ($38.0\mu s$). No entanto, para comprovar a influência da cache nos testes de latência somados ao estresse do Xenomai, seriam necessários maiores estudos a respeito desse efeito. Nenhum dos sistemas se mostrou capaz de se manter com latências próximas à primeira bateria de testes, quando não foi feito uso da ferramenta stress-ng. Apesar de ser conhecido que os sistemas estavam sob grande carga, espera-se que um sistema de tempo real não tenha muita variação nas tarefas de alta prioridade mesmo quando sob estresse. As tarefas de baixa prioridade, que nesse caso são representadas pelos processos stress-ng, devem ter sua execução postergada para dar lugar ao tratamento de interrupções quando a rotina for acionada.

5.2 Perdas de *deadline*

Essa seção busca avaliar o quanto as *deadlines* foram respeitadas pelas aplicações. A importância de as latências se limitarem ao tempo estabelecido como *deadline* está intimamente ligado com os conceitos de tempo real *hard* e *soft* (Seção 2.7.1). O período arbitrário de $1ms$ é estipulado como o prazo máximo de resposta da aplicação de interrupções na Raspberry Pi 3, além de ser o período mínimo de envio de novas interrupções pelo NodeMcu. Ainda que o NodeMcu possa lidar com atrasos por parte dos sistemas Linux, qualquer extrapolação do valor de $1000\mu s$ é tratado como perda de *deadline*.

Ao realizar os testes sem estresse não houve nenhuma perda de prazos no atendimento de interrupções para todos os sistemas operacionais utilizados, como pode ser visualizado na Tabela 3. Apesar disso, esses dados são de certa forma previsíveis, visto que o período de $1ms$ é consideravelmente alto para a capacidade de processamento da Raspberry Pi 3 e condição da aplicação ser executada quase que de forma exclusiva no sistema operacional.

Tabela 3 – Perdas de *deadlines* no tratamento de interrupções

Sistema	Perdas de *deadline*	% do total
Linux Vanilla	0	0
Real-Time Linux	0	0
Xenomai	0	0

A Tabela 4 mostra a relação de perdas de *deadline* quando aplicada a ferramenta stress-ng no sistema operacional. O Linux Vanilla perdeu 404 prazos, o que representa uma extrapolação dos requisitos a cada quase 8842 interrupções recebidas. O RTLinux

apresentou 24 perdas de prazos, que significa uma perda de prazo a cada 148857 interrupções ocorridas, aproximadamente. Apesar da parcela ser bastante pequena, tanto no Linux Vanilla, como no RTLinux, pode-se dizer que para exigências de tempo real *hard* esses dois sistemas operacionais não são capazes de cumprir *deadlines* de $1ms$. Os dois sistemas, porém, podem ser vistos como de tempo real *soft*, quando aplicados nesse contexto.

Tabela 4 – Perdas de *deadlines* no tratamento de interrupções com o sistema sob carga

Sistema	Perdas de *deadline*	% do total
Linux Vanilla	404	1.1×10^{-4}
Real-Time Linux	24	6.7×10^{-6}
Xenomai	0	0

O Xenomai foi o único dos três sistemas que não ultrapassou o tempo estipulado nenhuma vez. Observa-se, então, que o Xenomai é capaz de atender requisitos de tempo real *hard* quando a *deadline*, ou período máximo de atendimento, for de $1ms$. Como a carga computacional das aplicações de gerenciamento de interrupções é baixa, tentar reduzir o uso do processador nesse programa para atingir a *deadline* nos sistemas Vanilla e RTLinux não parece uma ideia factível. A aplicação que lida com as interrupções no Linux é feita para ser o mais simples possível. Caso seja necessário utilizar um dos dois sistemas, seja pela maior frequência de atualizações que os projetos recebem ou pela API mais bem estabelecida, existem três opções: reduzir a carga computacional das outras aplicações (stress-ng), aumentar a *deadline* ou, em últimos casos, substituir o hardware empregado.

O Linux Vanilla demonstrou precisar de uma *deadline* de mais de $3ms$ para ser qualificado como um sistema *hard real-time*, já o RTLinux, um valor próximo de $2ms$. Aumentar o prazo máximo de atendimento das interrupções pode significar no reprojeto completo da aplicação para a qual o dispositivo está sendo usado. Elevar a *deadline* de $1ms$ para $1.5ms$, por exemplo, significaria em potencialmente tornar o sistema 50% mais lento. Além disso, o valor da *deadline* muitas vezes está relacionado ao ambiente, e seus parâmetros físicos intrínsecos, no qual o sistema de tempo real se integra. A mudança da *deadline*, portanto, nem sempre é uma tarefa simples. No ambiente de testes utilizado, coloca-se o processador em uso máximo. Utilizar-se da opção `--cpu 3` da stress-ng, que ocuparia em média 75% da CPU, levantaria um novo caso de testes onde as aplicações secundárias disputam menos o processador com o processo principal. Este trabalho, todavia, preocupa-se em avaliar apenas o pior caso de estresse do sistema operacional, portanto casos de teste com menor uso da CPU não foram realizados. Se alterar a *deadline* ou reduzir o custo computacional das outras aplicações no sistema não for praticável, pode ser necessário reprojetar ou substituir o hardware utilizado, e possivelmente o sistema operacional (caso não haja suporte ao Linux), para o cumprimentos dos prazos desejados. Trocar exclusivamente o

sistema operacional, mantendo o hardware, para um outro além do Linux também torna-se uma opção ao considerar a utilização de RTOSs mais simplificados, abrindo mão das características do kernel Linux.

5.3 Latência do NodeMcu

Essa seção investiga os atrasos advindos do uso do NodeMcu como instrumento responsável por calcular as latências dos sistemas Linux. Através da sessão de testes proposta, deseja-se ter uma estimativa da influência do NodeMcu nos resultados obtidos na Seção 5.1 e na Seção 5.2. O esquemático pode ser visto na Figura 20, que é uma simplificação da configuração vista na Figura 12. O pino D6, antes conectado na porta GPIO24 da Raspberry Pi 3, agora é conectado diretamente no pino D5. Isso faz com que o NodeMcu gere interrupções (anteriormente geradas pela Raspberry Pi como resposta ao NodeMcu) para a própria placa, como forma de estimar qual a latência referente à esse dispositivo de forma isolada. A aplicação que é executada no NodeMcu permanece a mesma, onde os testes são realizados durante uma hora, ao passo que as latências são enviadas ao computador da mesma maneira.

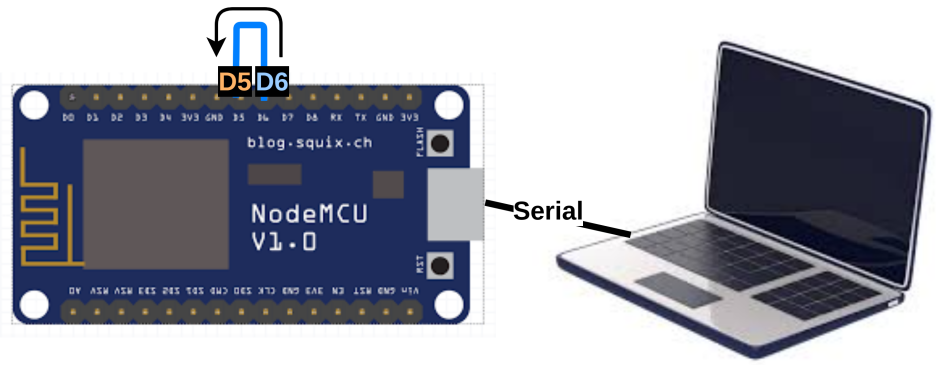

Figura 20 – Esquemático para os testes de latência do NodeMcu.

(Fonte: do autor.)

As latências obtidas podem ser visualizadas na Tabela 5, enquanto que o histograma das amostras é apresentado na Figura 21. Como pode ser visto, o NodeMcu está adicionando um atraso considerável aos dados extraídos dos sistemas Linux sob testes. O NodeMcu, ainda que faça uso de uma aplicação *bare-metal*, mostra que o tratamento de interrupções nele possui variações temporais. Essas medições, porém, não podem ser facilmente adicionadas à análise final, uma vez que esse é um teste indireto da latência do NodeMcu.

Os dados obtidos e aqui apresentados destinam-se principalmente a apontar que o atraso existe e não deve ser ignorado, mesmo que a interferência real introduzida pela

Tabela 5 – Latência do NodeMcu

Latência mínima (μs)	6.0
Latência máxima (μs)	25.0
Latência média (μs)	6.6
Desvio padrão (μs)	0.50
Número de amostras	3574138

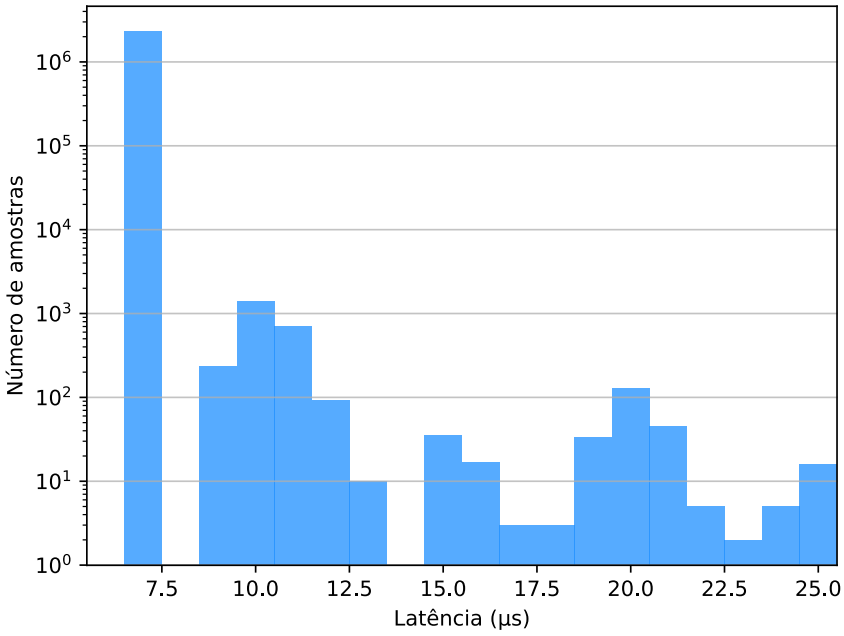

Figura 21 – Latências do NodeMcu.

(Fonte: do autor.)

plataforma não seja diretamente medida. Com isso, sem a contabilização do erro adicionado pelo NodeMcu os resultados finais poderiam ser um pouco melhores do que os que foram obtidos, para todos os sistemas e casos de teste. Como todos os testes utilizam-se do NodeMcu, é esperado que os atrasos nesse dispositivo não deem vantagem suficiente a um dos sistemas de forma que ele tenha sido erroneamente avaliado. Ainda assim, a realização de futuras modificações no esquema de testes para tratar desse tópico é sem dúvidas uma opção a ser estudada.

6 Considerações finais

O propósito deste trabalho foi de avaliar o desempenho de abordagens de sistemas operacionais de tempo real baseados no kernel Linux, sendo eles: o próprio Linux Vanilla, o Real-Time Linux e o Xenomai com o kernel Cobalt. O desempenho foi mensurado ao verificar as latências no tratamento de interrupções nos pinos GPIO da Raspberry Pi 3, que executou cada um dos sistemas propostos. Adicionalmente, foi introduzido o uso da ferramenta *stress-ng*, destinada ao estresse de sistemas operacionais, com a intenção de colocar o sistema sob condições extremas de uso computacional e proporcionar um novo ciclo de testes. As atividades de envio das interrupções, recebimento das respostas da Raspberry Pi e cálculo das latências foram encarregadas à plataforma NodeMcu. As latências foram simultaneamente enviadas via interface serial ao computador para posterior análise.

O método proposto foi suficiente para obter as características de tempo real dos sistemas executados na Raspberry Pi 3. Os testes foram bem sucedidos dentro das ferramentas que estavam disponíveis, pois é sabido que o NodeMcu adiciona incertezas quanto à real latência do tratamento de interrupções dos sistemas. Se o erro introduzido pelo NodeMcu puder se separado dos resultados, as latências finais medidas podem ser melhores do que as que foram apresentadas. Espera-se, entretanto, que o emprego do NodeMcu não tenha dado vantagem a um sistema específico em detrimento de outro, visto que foi utilizado em conjunto com todos os sistemas.

As interrupções de hardware demonstraram ser uma maneira bastante simples de aferir o grau dos diferentes tipos de kernel do Linux. A bateria de testes sem a aplicação de carga, todavia, não foi conclusiva quanto a qual sistema teve o melhor desempenho. O uso da ferramenta *stress-ng* foi essencial, nesse sentido, para o conjunto total dos dados levantados. A avaliação final pôde concluir que o Xenomai apresentou o melhor desempenho entre os três sistemas operacionais testados. O segundo melhor sistema ao considerar apenas o caso sob estresse foi o RTLinux, uma vez que apresentou o pior resultado na sessão de testes inicial. O Real-Time Linux mostrou lidar melhor com a carga nos recursos do sistema do que o Linux Vanilla, o que pode evidenciar a vantagem de usar do kernel Linux com esse *patch*. O Linux somado a características de tempo real pode ser muito bem utilizado, principalmente ao considerar que os hardwares disponíveis para aplicações embarcadas estão cada vez mais potentes. Muitos hardwares embarcados capazes de executar sistemas operacionais podem experimentar essas opções de RTOS somado às facilidades do Linux.

6.1 Trabalhos futuros

Aqui são listados alguns itens que poderiam ser agregados à análise de desempenho dos sistemas operacionais de tempo real em trabalhos futuros:

1. Nos testes efetuados, o NodeMcu assumiu o papel de gerar as interrupções e também de ler o sinal advindo da Raspberry Pi 3. Apesar de ter se mostrado suficiente para o gerar interrupções e ler respostas no tempo de $1ms$ adotado, o uso de um gerador de funções e também de um osciloscópio em substituição à placa poderiam ser uma alteração bastante importante no projeto. A mudança poderia minimizar muito as incertezas na medida da latência.

2. Verificar o porquê de o Real-Time Linux não conseguir um melhor desempenho que o Vanilla em condições sem estresse. O aumento do tempo de execução, citado no item anterior, poderia conceder melhores informações desse sistema e se, de fato, ele tem o desempenho pior que o Vanilla quando não há carga computacional.

3. Em adição ao gerenciamento de interrupções, é possível usar novas formas de testar as capacidades de tempo real dos sistemas. A criação de tarefas periódicas (que executam de forma cíclica) nos sistemas Linux, e que não respondem à eventos externos, poderia integrar outros pontos de vista à análise.

4. Em etapas futuras, poderiam ser adicionados outros sistemas operacionais de tempo real não relacionados com o Linux ao conjunto de testes. Isso traria uma melhor ideia do comportamento do Linux frente a outros RTOSs do mercado.

5. A *stress-ng* possui um enorme conjunto de opções que não foram exploradas nesse trabalho e que poderiam gerar novos casos de teste úteis.

6. A avaliação mais aprofundada dos efeitos e impactos da cache na latência do sistemas também pode trazer novas conclusões. A inclusão de condições de estresse totalmente atreladas à CPU e com o menor uso possível da memória, através da configuração correta do stress-ng, pode revelar se é a cache o fator decisivo no aumento *jitter* e da latência máxima no tratamento de interrupções.

7. É possível que a cache de nível 2 (L2) seja desabilitada da placa Raspberry Pi 3, o que pode agregar à análise do item anterior.

8. Outros hardwares além da linha de dispositivos Raspberry Pi podem ser explorados.

Referências Bibliográficas

1 STANKOVIC, J. A. Real-time systems. In: _____. *Encyclopedia of Computer Science*. [S.l.]: John Wiley and Sons Ltd., 2003. p. 1504–1506. ISBN 978-0470864125. 16, 35

2 SILBERSCHATZ PETER BAER GALVIN, G. G. A. *Operating Systems Concepts*. 8. ed. [S.l.]: Wiley, 2009. ISBN 978-0470128725. 16, 19, 21, 22, 23, 25, 26, 27

3 FARINES JONI DA SILVA FRAGA, R. S. d. O. J.-M. *Sistemas de Tempo Real*. [S.l.: s.n.], 2000. 16, 35, 36

4 2019 Embedded Markets Study. <https://www.embedded.com/wp-content/uploads/2019/11/EETimes_Embedded_2019_Embedded_Markets_Study.pdf>. 16, 40

5 ABOUT Linux Kernel. <https://www.kernel.org/linux.html>. 16, 37

6 GNU Software. <https://www.gnu.org/software/software.en.html>. 16

7 LOVE, R. *Linux Kernel Development: A thorough guide to the design and implementation of the Linux kernel*. 3. ed. [S.l.]: Addison-Wesley, 2010. ISBN 978-0672329463. 17, 22, 26, 27, 28, 37, 38, 39, 40, 42

8 STREIF, R. J. *Embedded Linux Systems with the Yocto Project*. [S.l.]: 978-0133443240, 2016. (Pearson Open Source Software Development Series). 17, 30

9 CONFIG PREEMPT RT Patch. <https://rt.wiki.kernel.org/index.php/CONFIG_PREEMPT_RT_Patch>. 17, 43

10 XENOMAI wiki. <https://source.denx.de/Xenomai/xenomai/-/wikis/Start_Here>. 17, 43

11 BLUM, C. B. R. *Linux Command Line and Shell Scripting Bible*. 3. ed. [S.l.]: Wiley, 2015. ISBN 978-1118983843. 19

12 MUTIA, R. I. Inter-process communication mechanism in monolithic kernel and microkernel. 20, 21

13 SYSCALLS(2) — Linux manual page. <https://man7.org/linux/man-pages/man2/syscalls.2.html>. 26

14 CORBET ALESSANDRO RUBINI, G. K.-H. J. *Linux Device Drivers*. 3. ed. [S.l.]: O'Reilly Media, 2005. ISBN 978-0596005900. 27

15 WHITE, E. *Making Embedded Systems: Design Patterns for Great Software*. [S.l.]: O'Reilly Media, 2011. ISBN 978-1449302146. 28

16 ANFINSON, K. Q. D. *IT Essentials: PC Hardware and Software Companion Guide*. 3. ed. [S.l.]: Cisco Press, 2008. ISBN 978-1587131998. 30

17 RASPBERRY Pi Foundation - About Us. <https://www.raspberrypi.org/about/>. 30

18 THE Raspberry Pi in scientific research. <https://www.raspberrypi.org/blog/the-raspberry-pi-in-scientific-research/>. 30

19 RASPBERRY Pi 3 Model B. <https://www.raspberrypi.org/homepage-9df4b/static/532b4c25752c4235d76cc41051baf9ab/ae23f/877fb653-7b43-4931-9cee-977a22571f65_3b%2BAngle%2B2%2Brefresh.jpg>. 31

20 PATCH(1) — Linux manual page. <https://man7.org/linux/man-pages/man1/patch.1.html>. 32

21 THE Open Source Definition. <https://opensource.org/osd>. 32

22 WHAT is free software? <https://www.gnu.org/philosophy/free-sw.en.html>. 32

23 WHAT Is Linux? <https://www.linux.com/what-is-linux/>. 33

24 LINUX From Scratch. <http://www.linuxfromscratch.org/>. 33

25 DISTROSHARE Ubuntu Imager. <https://github.com/Distroshare/distroshare-ubuntu-imager>. 33

26 INTRODUCING the Yocto Project. <https://docs.yoctoproject.org/overview-manual/yp-intro.html>. 34

27 THE Buildroot user manual. <https://buildroot.org/downloads/manual/manual.html>. 34

28 KERNIGHAN, D. M. R. B. W. *The C Programming Language*. 2. ed. [S.l.]: Prentice Hall PTR, 1988. 38

29 NARTEN, M. B. T. Unix operating system. *Encyclopedia of Computer Science*, John Wiley and Sons Ltd., v. 4, p. 1816–1819, 2003. 38

30 GNU Software. <https://www.gnu.org/gnu/gnu-history.html>. 39, 40

31 ABOUT Linux Kernel. <https://itsfoss.com/linux-runs-top-supercomputers/>. 40

32 USAGE statistics of operating systems for websites. <https://w3techs.com/technologies/overview/operating_system>. 40

33 MOBILE Operating System Market Share Worldwide. <https://gs.statcounter.com/os-market-share/mobile/worldwide#.YHxF5opr1vw.link>. 40

34 SCHED_SETSCHEDULER(2) — Linux manual page. <https://man7.org/linux/man-pages/man2/sched_setscheduler.2.html>. 42

35 NICE(2) — Linux manual page. <https://man7.org/linux/man-pages/man2/nice.2.html>. 42

36 COBALT. <https://source.denx.de/Xenomai/xenomai/-/wikis/figures/x3-cobalt-interfaces.png>. 44

37 PORTING a Linux application to Xenomai dual kernel. <https://source.denx.de/Xenomai/xenomai/-/wikis/Porting_To_Xenomai_POSIX>. 44

38 MERCURY. <https://source.denx.de/Xenomai/xenomai/-/wikis/figures/x3-mercury-interfaces.png>. 45

39 LIFE with Adeos. <https://source.denx.de/Xenomai/xenomai/-/wikis/Life_With_Adeos>. 45

40 YAGHMOUR, K. *Adaptive Domain Environment for Operating Systems.* <https://www.opersys.com/ftp/pub/Adeos/adeos.pdf>. 45

41 FULL virtualization. <https://en.wikipedia.org/wiki/Full_virtualization>. 45

42 ALCHEMY API. <https://xenomai.org/documentation/xenomai-3/html/xeno3prm/group___alchemy.html>. 46

43 RTDM. <https://xenomai.org/documentation/xenomai-3/html/xeno3prm/group__rtdm.html>. 46

44 TASK latency. <http://www.cs.ru.nl/J.Hooman/DES/XenomaiExercises/tasklatency.jpg>. 50

45 ESP8266. <https://www.espressif.com/en/products/socs/esp8266>. 49

46 NODEMCU ESP8266. <https://components101.com/development-boards/nodemcu-esp8266-pinout-features-and-datasheet>. 49

47 ARDUINO core for ESP8266 WiFi chip. <https://github.com/esp8266/Arduino>. 51

48 RASPBERRY Pi GPIO. <https://www.raspberrypi.org/documentation/usage/gpio/images/GPIO-Pinout-Diagram-2.png>. 52

49 ABOUT libgpiod. <https://git.kernel.org/pub/scm/libs/libgpiod/libgpiod.git/about/>. 51

50 STRESS-NG - a tool to load and stress a computer system. <https://manpages.ubuntu.com/manpages/artful/man1/stress-ng.1.html>. 53, 63

51 PTHREADS(7) — Linux manual page. <https://man7.org/linux/man-pages/man7/pthreads.7.html>. 59

52 MLOCK(2) — Linux manual page. <https://man7.org/linux/man-pages/man2/mlockall.2.html>. 60

53 NMAP. <https://nmap.org>. 64

54 BUILDROOT - Making Embedded Linux Easy. <https://buildroot.org/>. 81

55 GITHUB. <https://github.com/>. 82

56 INSTALLING Xenomai 3.x. <https://xenomai.org/documentation/xenomai-3/html/README.INSTALL/>. 83

57 JOHANSSON, G. *Real-Time Linux Testbench on Raspberry Pi 3 using Xenomai.* Http://kth.diva-portal.org/smash/get/diva2:1251188/FULLTEXT01.pdf. 84

58 TROUBLESHOOTING a dual kernel configuration. <https://xenomai.org/documentation/xenomai-3/html/TROUBLESHOOTING.COBALT/>. 84

A Configuração e instalação

Esse apêndice discorre sobre a configuração e instalação de cada um dos sistemas operacionais utilizados no trabalho. Para a replicação dos passos apresentados é necessário usar um sistema operacional Linux.

A.1 Obtenção das fontes

Inicialmente, cria-se a pasta `Testes/` dentro da pasta *home* do usuário, a partir da qual será feita toda a instalação:

```
~/ $ mkdir ~/Testes
```

Cria-se também uma pasta para armazenar os *patches* do I-Pipe e `PREEMPT_RT`:

```
~/ $ mkdir ~/Testes/patches
```

A.1.1 Xenomai

Os códigos-fonte do Xenomai 3 são adquiridos ao clonar o repositório git do projeto:

```
~/ $ cd Testes
~/Testes/ $ git clone http://git.xenomai.org/xenomai-3.git/
   ↪ ~/Testes/xenomai3
```

A.1.2 Kernel Linux

O Linux 4.19.144 (Vanilla ou *mainline*) é adquirido pelo repositório git oficial do *linux-stable*:

```
~/Testes/ $ clone --single-branch --branch v4.19.144
   ↪ https://git.kernel.org/pub/scm/linux/kernel/git/stable/linux.git
   ↪ linux-vanilla
```

Para diminuir o espaço ocupado pelas fontes do linux, remove-se a pasta escondida `.git/`, de forma que apenas as fontes da versão 4.19.144 permanecem no diretório. Primeiramente verifica-se que o kernel está de fato na versão 4.19.144 pelos comandos:

```
~/Testes/ $ cd linux-vanilla
~/Testes/ $ make kernelversion
~/Testes/linux-vanilla/ $ rm -rf .git
```

Por fim, copia-se a pasta do linux para que ela posteriormente seja utilizada para configurar o Xenomai e também o RTLinux:

```
~/Testes/linux-vanilla/ $ cd ..
~/Testes/ $ cp linux-vanilla/ linux-xenomai/
~/Testes/ $ cp linux-vanilla/ linux-rt/
```

A.1.3 Buildroot

O Buildroot é obtido pelo seu site oficial [54], nesse caso é utilizada a versão 2020-02-11:

```
~/Testes/ $ wget https://buildroot.org/downloads/buildroot-2020.02.11.tar.gz
~/Testes/ $ tar -xvzf buildroot-2020.02.11.tar.gz
~/Testes/ $ rm buildroot-2020.02.11.tar.gz
```

É usado um repositório do buildroot para cada sistema operacional Linux, portanto, faz-se:

```
~/Testes/ $ mv buildroot-2020.02.11 buildroot-vanilla
~/Testes/ $ cp buildroot-vanilla buildroot-xenomai
~/Testes/ $ cp buildroot-vanilla buildroot-rt
```

A.1.4 *Patches*

O *tarball* do *patch* `PREEMPT_RT` na versão 4.19.148 (compatível com o kernel 4.19.144) é obtido e descompactado através dos comandos:

```
~/Testes/ $ cd patches
~/Testes/patches/ $ wget https://mirrors.edge.kernel.org/pub/linux/-
   ↪kernel/projects/rt/4.19/older/patch-4.19.148-rt64.patch.gz
~/Testes/patches/ $ tar -xvzf patch-4.19.148-rt64.patch.gz
~/Testes/patches/ $ rm patch-4.19.148-rt64.patch.gz
```

O *patch* do I-Pipe para a arquitetura ARM de 32 bits, a mesma que em que o sistema será compilado, e versão 4.19.144 do kernel Linux é obtido pelo comando:

```
~/Testes/patches/ $ wget https://xenomai.org/downloads/ipipe/v4.x/-
   ↪arm/ipipe-core-4.19.144-arm-10.patch
```

A.2 Configuração dos kernels

A.2.1 Kernel Vanilla

A configuração do kernel vanilla para a Raspberry Pi 3 é baseada na opção que por padrão está disponível nos códigos-fonte, chamada `multi_v7_defconfig`. Cria-se, então,

uma nova configuração para quaisquer futuras modificações no kernel:

```
~/ $ cd ~/Testes/linux-vanilla
~/Testes/linux-vanilla/ $ cp arch/arm/configs/multi_v7_defconfig
   ↪ arch/arm/configs/rpi3_vanilla_defconfig
```

Qualquer configuração adicional pode ser feita com a interface `menuconfig` (nesse caso não é necessário fazer nenhum modificação). A arquitetura desejada deve ser especificada através das variáveis do sistema, que no caso da Raspberry Pi é ARM (32 bits):

```
~/Testes/linux-vanilla/ $ export ARCH=arm
~/Testes/linux-vanilla/ $ make rpi3_vanilla_defconfig
~/Testes/linux-vanilla/ $ make menuconfig
```

Ao navegar pelo menu, é possível se certificar que a família de SoC BCM2835, utilizada pela Raspberry Pi, está selecionada como sistema suportado na instalação:

```
System Type  --->
    [*] Broadcom SoC Support  --->
        [*]   Broadcom BCM2835 family
```

Para sair do `menuconfig`, escolhe-se a opção `<Exit>`. Foi escolhido utilizar um repositório remoto no Github [55] para que o Buildroot mais tarde possa obter as fontes do Vanilla sem necessitar que o kernel esteja acessível localmente. O repositório remoto criado é adicionado ao novo repositório local:

```
~/Testes/linux-vanilla/ $ git init
~/Testes/linux-vanilla/ $ git commit -m "Adicionado o kernel vanilla e
   ↪ configuracao da RPI3"
~/Testes/linux-vanilla/ $ git remote add origin https://github.com/<caminho
para o repositório>.git
~/Testes/linux-vanilla/ $ git push origin master
```

A.2.2 Kernel com o *patch* `PREEMPT_RT`

O kernel do RTLinux necessita da aplicação do patch `PREEMPT_RT`, que possibilita que ele seja compilado na forma totalmente preemptiva (opção `PREEMPT_RT_FULL`, não disponível por padrão). O comando `patch` pode ser utilizado com a opção `--dry-run`, que não fará nenhuma alteração no código fonte, mas mostrará se o *patch* pode ser aplicado com sucesso (sem erros):

```
~/ $ cd ~/Testes/linux-rt/
 /Testes/linux-rt/patch $ -p1 -dry-run < ../patches/-
```

```
  ↪patch-4.19.148-rt64.patch
```

Caso caso nenhum erro tenha ocorrido, o que deve ser verdade para as versões utilizadas, aplica-se o *patch* propriamente dito:

```
/Testes/linux-rt/patch $ patch -p1 < ../patches/-
  ↪patch-4.19.148-rt64.patch
```

Agora a opção `PREEMPT_RT_FULL` está disponível para ser selecionada no `menuconfig`:

```
~/Testes/linux-rt/ $ export ARCH=arm
~/Testes/linux-vanilla/ $ make multi_v7_defconfig
~/Testes/linux-rt/ $ make menuconfig
```

Escolhe-se o modo de preempção *Fully Preemptible Kernel* (`PREEMPT_RT_FULL`):

```
General setup  --->
    Preemption Model (...)  --->
        (X) Fully Preemptible Kernel (RT)
```

Agora, seleciona-se a opção `<Save>`, onde o arquivo destino é mantido como `.config`. Após isso, a configuração é salva ao fazer:

```
~/Testes/linux-rt/ $ make savedefconfig
~/Testes/linux-rt/ $ mv defconfig arch/arm/configs/rpi3_rt_defconfig
```

Agora, como é executado no Apêndice A.2.1, inicializa-se um novo repositório remoto para o kernel do Real-Time Linux, que será utilizado pelo Buildroot, salvando as modificações feitas no kernel.

A.2.3 Kernel com o *patch* do I-Pipe e adição do Cobalt

O Xenomai precisa que o *hypervisor* I-Pipe seja integrado ao kernel do Linux para a adição do Cobalt, o kernel de tempo real. A configuração aqui efetuada se baseia nos passos da documentação oficial do Xenomai [56]. Como foi feito com o *patch* do RTLinux, verifica-se se não ocorrem erros na aplicação do I-Pipe:

```
~/ $ cd ~/Testes/linux-xenomai/
~/Testes/linux-xenomai/ $ patch -p1 --dry-run < ../patches/-
  ↪ipipe-core-4.19.144-arm-10.patch
```

A aplicação de *patch* do I-Pipe e o preparo do Cobalt, podem ser realizados pelo script `prepare-kernel.sh` presente no Xenomai:

```
~/Testes/linux-xenomai/ $ cd ../xenomai3/
```

```
~/Testes/xenomai3/ $ ./scripts/prepare-kernel.sh --linux=~/Testes/-
   ↪linux-xenomai --arch=arm --ipipe=~/Testes/patches/-
   ↪ipipe-core-4.19.144-arm-10.patch
```

O kernel está pronto, mas ainda é necessário desativar algumas configurações que são recomendadas para a redução da latência do sistema na configuração de kernel duplo [57][58]. Algumas dessas mudanças são explicitamente apresentadas no `menuconfig`, como pode ser visto na Figura 22.

```
[*] Xenomai/cobalt  --->
    *** WARNING! Page migration (CONFIG_MIGRATION) may increase ***
    *** latency. ***
    *** WARNING! At least one of APM, CPU frequency scaling, ACPI 'processor' ***
    *** or CPU idle features is enabled. Any of these options may ***
    *** cause troubles with Xenomai. You should disable them. ***
```

Figura 22 – Alertas durante a configuração de kernel duplo do Xenomai.

(Fonte: do autor.)

As primeiras opções a serem desmarcadas são relacionada à memória. A configuração *Page migration*, a qual o Xenomai recomenda que não esteja selecionada, está presente dentro de *Allow for memory compaction*:

```
Memory Management options  --->
    [ ] Allow for memory compaction
    [ ] Contiguous Memory Allocator
```

A opção *CPU Frequency scaling* faz com que a frequência da CPU seja dinâmica de acordo com a carga de trabalho imposta a ela. Essa opção é totalmente indesejada em um sistema de tempo real e deve ser desabilitada:

```
CPU Power Management  --->
    CPU Frequency scaling  --->
        [ ] CPU Frequency scaling
```

Por fim, dentro das configurações que não espera-se que estejam presentes no Xenomai, desabilita-se a opções KGDB, não deve ser habilitada em hardwares de arquitetura ARM:

```
Kernel hacking  --->
    [ ] KGDB: kernel debugger  ----
```

Com o kernel configurado, adiciona-se uma nova configuração em `arch/arm/configs/`, chamada `rpi3_xenomai_defconfig`, da mesma forma que é feito no final do Apêndice

A.2.2. Adiciona-se também esse kernel a um novo repositório remoto, como é feito com o Vanilla e o RTLinux.

A.3 Configuração das distribuições no Buildroot

O Buildroot é similarmente configurado pela ferramenta `menuconfig`. Com o buildroot é possível escolher a *toolchain* para compilação da arquitetura ARM da Raspberry Pi 3, escolher o kernel (dos repositórios criados no Apêndice A.2), escolher as bibliotecas e utilidades desejadas, bem como fazer algumas configurações pontuais adicionais.

A.3.1 Configurações comuns a todos os kernels

As configurações aqui presentes são feitas em todos os repositórios do Buildroot criados ao executar as devidas modificações para cada sistema, quando for indicado (entre '<' e '>'). Inicialmente, são criadas algumas pastas que conterão o script customizado e também a aplicação:

```
~/ $ cd ~/Testes/buildroot-<xenomai/vanilla/rt>
~/Testes/buildroot-<xenomai/vanilla/rt> $ mkdir custom_scripts/
~/Testes/buildroot-<xenomai/vanilla/rt> $ mkdir custom_scripts/-
   ↪latency_app/
```

Dentro da pasta `custom_scripts/` deve estar presente o arquivo `pre-build.sh`. Esse script é específico de cada distribuição e pode ser visto no Apêndice B.4. O propósito desse *shell script* é de compilar a aplicação e copiar o binário resultante, chamado `latency`, na pasta destino do sistema compilado. No Buildroot do Xenomai, esse script também tem a responsabilidade de compilar o espaço do usuário do Xenomai. Na pasta `latency_app/` deve ser colocado o *Makefile* (Apêndice B.3) e a aplicação (Apêndice B.2).

O Buildroot possui uma configuração padrão para Raspberry Pi 3, que serve como base para todas as distribuições desse trabalho. Para iniciar a configuração, basta que seja feito:

```
~/Testes/buildroot-<xenomai/vanilla/rt> $ make raspberrypi3_defconfig
~/Testes/buildroot-<xenomai/vanilla/rt> $ make menuconfig
```

A *toolchain* deve ser a mesma para todos os kernels. A *toolchain* escolhida é disponibilizada pelo próprio Buildroot, chamada *Linaro*:

```
Toolchain  --->
    Toolchain type (...)  --->
        (X) External toolchain
```

```
    Toolchain (...)  --->
        (X) Linaro ARM 2018.05
```

É necessário estabelecer uma senha para o usuário *root* para que seja possível estabelecer uma conexão SSH com a Raspberry Pi:

```
System configuration  --->
    [*] Enable root login with password
    (123456) Root password
```

Configura-se, juntamente com o script padrão do Buildroot para Raspberry Pi 3 `post-build.sh`, o script customizado `post-build.sh`:

```
System configuration  --->
    (board/raspberrypi3/post-build.sh custom-scripts/post-build.sh)
        Custom scripts to run before creating filesystem images
```

O kernel deve ser o mesmo que foi enviado ao repositório remoto:

```
[*] Linux Kernel
    Kernel version (...)  --->
        (X) Custom Git repository
(https://github.com/<caminho para o repositório>.git)
    URL of custom repository
(<último commit hash do repositório remoto do kernel>)
    Custom repository version
```

O arquivo de configuração que foi gerado anteriormente agora é escolhido. Define-se também o arquivo *DTS* (*Device Tree Source*) que representa o árvore de dispositivos (hardware) da Raspberry Pi 3, o qual nesse caso é o `bcm2837-rpi-3-b.dts` presente no código fonte do kernel:

```
[*] Linux Kernel
    (rpi3_<vanilla/rt/xenomai>) Defconfig name
    (bcm2837-rpi-3-b) In-tree Device Tree Source file names
```

Para o acesso SSH ao dispositivo, seleciona-se os seguintes pacotes:

```
Target packages  --->
    Networking applications  --->
        [*] dropbear
        [*] openssh
```

A.3.2 Linux Vanilla e Real-Time Linux

O Linux Vanilla e Real-Time Linux usam a biblioteca *libgpiod* para o controle dos pinos GPIO, que é adicionada da seguinte forma:

```
Target packages  --->
    Libraries  --->
        Hardware handling  --->
            [*] libgpiod
```

Da mesma maneira de quando o kernel foi configurado, salva-se as configurações do Buildroot no arquivo `.config`. A partir do `.config`, cria-se uma nova configuração:

```
~/Testes/buildroot-<vanilla/rt>/ $ cp .config configs/-
   ↪rpi3_<vanilla/rt>_defconfig
```

A.3.3 Xenomai

Como deseja-se controle sobre os pinos GPIO da placa Raspberry Pi 3, o driver de tempo real deve ser habilitado dentro das configurações do Xenomai:

```
[*] Xenomai/cobalt  --->
    Drivers  --->
        Real-time GPIO drivers  --->
            [*] GPIO controller
            <M>   Support for BCM2835 GPIOs
```

A partir do `.config` que é salvo, cria-se uma nova configuração:

```
~/Testes/buildroot-xenomai/ $ cp .config configs/rpi3_xenomai_defconfig
```

Deve-se também fazer uma modificação no script padrão da Raspberry Pi utilizado pelo buildroot de forma que a placa possa corretamente utilizar a rede. Apenas a distribuição do Xenomai mostrou necessitar dessa alteração, que modifica o arquivo `cmdline.txt` na pasta de *boot* do sistema. A partir da pasta raiz do Buildroot, o arquivo está localizado

em `board/raspberrypi` e com o nome `post-image.sh`. O script deve receber a seguinte modificação:

```
@@ -47,7 +47,7 @@ done

 trap 'rm -rf "${ROOTPATH_TMP}"' EXIT
 ROOTPATH_TMP="$(mktemp -d)"
+echo "dwc_otg.fiq_enable=0 dwc_otg.fiq_fsm_enable=0 dwc_otg.nak_holdoff
    =0" >> ${BINARIES_DIR}/rpi-firmware/cmdline.txt

 rm -rf "${GENIMAGE_TMP}"
```

A.4 Compilação e instalação

Uma vez que a configuração dos kernels e do Buildroot esteja pronta no diretório de cada distribuição, basta utilizar o comando `make`, que fará todo o procedimento de baixar a *toolchain*, o kernel e as bibliotecas e fará a compilação do sistema operacional:

```
~/Testes/buildroot-<vanilla/rt/xenomai>/ $ make
```

Todos os arquivos de compilação aparecem na pasta `output/`. Os arquivos utilizados pelo sistema responsável pela compilação (isto é, o computador *host*) estão em `output/host/`, enquanto que o sistema de arquivos (*rootfs*) da Raspberry Pi está na pasta `output/target/`. Caso seja necessário adicionar uma biblioteca ou fazer algumas alteração na aplicação é possível apenas utilizar o comando `make` novamente, que compilará apenas as dependências. Se for realizada uma mudança maior, como uma alteração no kernel, por exemplo, é necessário limpar a pasta `output` através seguinte comando antes de utilizar o `make` novamente:

```
~/Testes/buildroot-<vanilla/rt/xenomai>/ $ make clean
```

A imagem final da distribuição está localizada em `output/images/`, chamada `sdcard.img`. Para descobrir onde o cartão SD conectado no computador está localizado na pasta `/dev/` é possível listar todos os dispositivos de armazenamento e suas partições ao fazer:

```
~/ $ sudo fdisk -l
(SAÍDA DO COMANDO)

[...]

Disk /dev/sdb: 7.41 GiB, 7948206080 bytes, 15523840 sectors
Disk model: Storage Device
Units: sectors of 1 * 512 = 512 bytes
Sector size (logical/physical): 512 bytes / 512 bytes
```

```
I/O size (minimum/optimal): 512 bytes / 512 bytes
Disklabel type: dos
Disk identifier: 0x00000000

Device     Boot Start    End Sectors  Size Id Type
/dev/sdb1  *        1  65536   65536   32M  c W95 FAT32 (LBA)
/dev/sdb2       65537 311296  245760  120M 83 Linux
```

Nesse exemplo, o cartão SD (de aproximadamente 8GB) está localizado em `/dev/sdb`. Existem duas partições presentes nele, `sdb1` e `sdb2`, que referem-se a um sistema operacional Linux já instalado. O cartão deve estar livre dessas partições antes da cópia da imagem do novo sistema. Elas podem ser facilmente removidas ao executar:

```
~/ $ sudo fdisk /dev/sdb
Command (m for help):  d <número da partição>
```

Quando todas as partições do dispositivo tiverem sido apagadas, deve ser feito:

```
Command (m for help):  w
```

Por fim, a imagem pode ser copiada para o cartão SD pelo comando que segue:

```
~/ $ sudo dd bs=1M if=~/Testes/buildroot-<vanilla/rt/xenomai>/output/-
   ↪images/sdcard.img of=/dev/sdb
```

Com isso, o cartão SD está pronto para ser conectado à Raspberry Pi 3 para a inicialização do sistema operacional.

B Códigos-fonte

B.1 Aplicação do NodeMcu

```
// Arquivo: isr_gen.ino

const int interrupt_pin = 14; // D5
const int led_pin = 12; // D4
unsigned long time_now;
unsigned long time_sent = 0;
unsigned long time_received;
unsigned long interval = 1000; // 1ms
unsigned long time_start = micros();
unsigned long time_to_stop = 60UL * 60UL * 1000000UL; // 1h

bool triggered_isr = false;
bool can_send_interrupt = true;
int value = LOW;


ICACHE_RAM_ATTR void interrupt_handler()
{
  time_received = micros();
  triggered_isr = true;
}

void send_interrupt()
{
    digitalWrite(led_pin, value);
    time_sent = time_now;
    value = !value;
    can_send_interrupt = false;
}

void stop()
{
  detachInterrupt(digitalPinToInterrupt(interrupt_pin));
  digitalWrite(led_pin, LOW);
  while(1) { delay(1000); }
}

void setup()
{
  Serial.begin(115200);
```

```
  pinMode(led_pin, OUTPUT);
  digitalWrite(led_pin, LOW);
  value = !value;

  delay(10000);

  attachInterrupt(digitalPinToInterrupt(interrupt_pin),
   interrupt_handler,
        CHANGE);
}

void loop()
{
  if (triggered_isr)
  {
    int time_diff = time_received - time_sent;
    triggered_isr = false;
    can_send_interrupt = true;
    Serial.print(time_diff);
    Serial.print('\n');

  }

  time_now = micros();
  if (can_send_interrupt && (time_now - time_sent >= interval))
  {
      send_interrupt();
  }

  if (time_now >= time_start + time_to_stop)
  {
    stop();
  }
}
```

B.2 Aplicações da Raspberry Pi 3

B.2.1 Aplicação de interrupções do Linux Vanilla e do Real-Time Linux

```
// Arquivo: latency.c

#include <stdio.h>
#include <errno.h>
```

```c
#include <stdlib.h>
#include <unistd.h>
#include <stdint.h>
#include <sched.h>
#include <pthread.h>

#include <sys/mman.h>

#include <gpiod.h>

#ifndef CONSUMER
#define CONSUMER "Consumer"
#endif

const char *chipname = "gpiochip0";
struct gpiod_chip *chip;
struct gpiod_line *led;
struct gpiod_line *interrupt;
struct gpiod_line_event event;

int led_pin = 22;
int interrupt_pin = 24;


void *led_interrupt(void *arg)
{
    int ret, value = 1;
    struct sched_param param = { .sched_priority = 99 };

    ret = sched_setscheduler(getpid(), SCHED_RR, &param);

    while (1)
    {
        ret = gpiod_line_event_wait(interrupt, NULL);
        if (ret > 0)
        {
            ret = gpiod_line_event_read(interrupt, &event);
            if (ret < 0)
            {
                perror("Error: reading last GPIO interrupt failed\n");
                gpiod_line_release(interrupt);
                gpiod_line_release(led);
                gpiod_chip_close(chip);
            }

            ret = gpiod_line_set_value(led, value);
            if (ret < 0)
```

```
            {
                perror("Error: writing to led pin failed\n");
                gpiod_line_release(interrupt);
                gpiod_line_release(led);
                gpiod_chip_close(chip);
            }

            value = !value;
        }
        else if (ret < 0)
        {
            perror("Error: waiting for event notification failed\n");
            gpiod_line_release(interrupt);
            gpiod_line_release(led);
            gpiod_chip_close(chip);
            exit(1);
        }
    }
}

void setup_gpio()
{
    int ret;

    chip = gpiod_chip_open_by_name(chipname);
    if (!chip)
    {
        perror("Error: Opening chip GPIO failed\n");
        exit(1);
    }

    led = gpiod_chip_get_line(chip, led_pin);
    if (!led)
    {
        perror("error: getting gpio led line failed\n");
        gpiod_chip_close(chip);
        exit(1);
    }

    interrupt = gpiod_chip_get_line(chip, interrupt_pin);
    if (!interrupt)
    {
        perror("error: getting gpio interrupt line failed\n");
        gpiod_chip_close(chip);
        exit(1);
    }
```

```
    ret = gpiod_line_request_output(led, CONSUMER, 0);
    if (ret < 0)
    {
        perror("Error: requesting led line as output failed\n");
        gpiod_line_release(led);
        gpiod_chip_close(chip);
        exit(1);
    }

    ret = gpiod_line_request_both_edges_events(interrupt, CONSUMER);
    if (ret < 0)
    {
        perror("Error: requesting interrupt failed\n");
        gpiod_line_release(interrupt);
        gpiod_line_release(led);
        gpiod_chip_close(chip);
        exit(1);
    }
}

int main(int argc, char *argv[])
{
    pthread_t task_interrupt;

    mlockall(MCL_CURRENT | MCL_FUTURE);

    setup_gpio();

    pthread_create(&task_interrupt, NULL, led_interrupt, NULL);

    printf("Press any key to stop . . .\n");
    getchar();
    gpiod_line_release(interrupt);
    gpiod_line_release(led);
    gpiod_chip_close(chip);
    return 0;
}
```

B.2.2 Aplicação de interrupções do Xenomai

```
// Arquivo: latency.c

#include <stdio.h>
#include <stdlib.h>
#include <errno.h>
#include <string.h>
```

```
#include <unistd.h>
#include <sys/mman.h>
#include <alchemy/task.h>
#include <rtdm/gpio.h>

int pin_out;
int pin_isr;


void pin_write(int pin, int value)
{
    int ret, error_code;

    ret = write(pin, &value, sizeof(value));
    if (ret == -1)
    {
        error_code = errno;
        rt_printf("Error: writing to led pin failed. Code %d, %s\n",
                error_code, strerror(error_code));
        exit(1);
    }
}

int pin_read(int pin)
{
    int ret, error_code, value;

    ret = read(pin, &value, sizeof(value));
    if (ret == -1)
    {
        error_code = errno;
        rt_printf("Error: reading interrupt pin failed. Code %d, %s\n",
               error_code, strerror(error_code));
        exit(1);
    }

    return value;
}

void led_interrupt(void *arg)
{
    int ret, error_code, value = 1;
    int count = 0;
    RT_TASK_INFO taskinfo;
    rt_task_inquire(NULL, &taskinfo);
    rt_printf("Executing task: %s\n", taskinfo.name);
```

```
    while (1)
    {
        pin_read(pin_isr);
        pin_write(pin_out, value);
        value = !value;
    }
}

void setup_gpio()
{
    int ret, error_code, value = 0;

    pin_out = open("/dev/rtdm/pinctrl-bcm2835/gpio2016", O_WRONLY);
    if (pin_out == -1)
    {
        error_code = errno;
        rt_printf("Error: opening led pin failed. Code %d, %s\n",
                error_code, strerror(error_code));
        exit(1);
    }

    ret = ioctl(pin_out, GPIO_RTIOC_DIR_OUT, &value);
    if (ret == -1)
    {
        error_code = errno;
        rt_printf("Error: led pin setup failed. Code %d, %s\n",
                error_code, strerror(error_code));
        exit(1);
    }

    ret = write(pin_out, &value, sizeof(value));
    if (ret == -1)
    {
        error_code = errno;
        rt_printf("Error: writing to led pin failed. Code %d, %s\n",
                error_code, strerror(error_code));
        exit(1);
    }

    pin_isr = open("/dev/rtdm/pinctrl-bcm2835/gpio2018", O_RDONLY);
    if (pin_isr == -1)
    {
        error_code = errno;
        rt_printf("Error: opening interrupt pin failed. Code %d, %s\n",
                error_code, strerror(error_code));
        exit(1);
    }
```

```
    int xeno_trigger = GPIO_TRIGGER_EDGE_RISING |
   GPIO_TRIGGER_EDGE_FALLING;
    ret = ioctl(pin_isr, GPIO_RTIOC_IRQEN, &xeno_trigger);
    if (ret == -1)
    {
        error_code = errno;
        rt_printf("Error: interrupt pin setup failed. Code %d, %s\n",
                error_code, strerror(error_code));
        exit(1);
    }
}

int main(int argc, char *argv[])
{
    int ret, error_code;
    RT_TASK task_interrupt;

    mlockall(MCL_CURRENT | MCL_FUTURE);

    setup_gpio();

    ret = rt_task_create(&task_interrupt, "Interrupt task", 0, 99, 0);
    if (ret != 0)
    {
        error_code = -ret;
        rt_printf("Error: creating interrupt task failed. Code %d, %s\n"
   ,
                error_code, strerror(error_code));
        exit(1);
    }

    ret = rt_task_start(&task_interrupt, &led_interrupt, 0);
    if (ret != 0)
    {
        error_code = -ret;
        rt_printf("Error: starting interrupt task failed. Code %d, %s\n"
   ,
                error_code, strerror(error_code));
        exit(1);
    }

    rt_printf("Press Ctrl + C to stop . . .");
    pause();
}
```

B.3 Makefiles

B.3.1 Makefile da aplicação do Vanilla e do Real-Time Linux

```
TOOLCHAIN=$(BASE_DIR)/host/bin
CC := $(TOOLCHAIN)/arm-linux-gnueabihf-gcc
CFLAGS := -lpthread -lgpiod

OUTPUT := latency

all: $(OUTPUT)

%.o: %.c
    $(CC) -o $@ $< $(CFLAGS)
```

B.3.2 Makefile da aplicação do Xenomai

```
STAGE=../../output/target/
XENO_CONFIG := ../../output/target/usr/xenomai/bin/xeno-config
CFLAGS := $(shell DESTDIR=$(STAGE) $(XENO_CONFIG) --rtdm --posix --
   alchemy --cflags)
LDFLAGS := $(shell DESTDIR=$(STAGE) $(XENO_CONFIG) --rtdm --posix --
   alchemy --ldflags)
CC := $(shell DESTDIR=$(STAGE) $(XENO_CONFIG) --cc)

EXECUTABLE := latency


all: $(EXECUTABLE)

%: %.c
    $(CC) -o $@ $< $(CFLAGS) $(LDFLAGS)
```

B.4 Script utilitário de pós-compilação

B.4.1 Script `post-build.sh` do Linux Vanilla e Real-Time Linux

```
#!/bin/bash

# Compila  o e c pia do bin rio para o rootfs do sistema da
# Raspberry Pi
make -C $BASE_DIR/../custom-scripts/latency_app
cp $BASE_DIR/../custom-scripts/latency_app/latency $BASE_DIR/target/usr/
   bin
```

```
# Para permitir o acesso SSH como root
sed -i "s/#PermitRootLogin prohibit-password/PermitRootLogin yes/g"
    $BASE_DIR/target/etc/ssh/sshd_config
```

B.4.2 Script post-build.sh do Xenomai

```
#!/bin/bash

# Compila  o do espa o do usu rio do Xenomai
tar -zxvf $BASE_DIR/../custom-scripts/packages/xenomai-3.1.tar.gz -C
    $BASE_DIR/build
pushd $BASE_DIR/build/xenomai-3.1
scripts/bootstrap
export CFLAGS="-march=armv7-a -mfloat-abi=hard -mfpu=neon -ffast-math"
export LDFLAGS="-march=armv7-a -mfloat-abi=hard -mfpu=neon -ffast-math"
export DESTDIR=$BASE_DIR/target
mkdir -p build
pushd build
../configure --enable-smp --host=arm-linux-gnueabihf --with-core=cobalt
make && make install
popd
popd

# Compila  o e c pia do bin rio para o rootfs do sistema
# da Raspberry Pi
make -C $BASE_DIR/../custom-scripts/latency_app
cp  $BASE_DIR/../custom-scripts/latency_app/latency $BASE_DIR/target/usr
    /xenomai/bin
```