UNIVERSIDADE DE LISBOA
FACULDADE DE CIÊNCIAS
DEPARTAMENTO DE INFORMÁTICA

Estudo, implementação e avaliação de um sistema de tempo-real na plataforma RaspberryPi

Mestrado em Engenharia Informática
Especialização em Arquitetura, Sistemas e Redes de Computadores

Hugo Coutinho Fontes Viegas

Dissertação orientada por:
Prof. Doutor António Casimiro Ferreira da Costa

2015

Resumo

A maioria dos sistemas operativos e do hardware comerciais que existem para implementar sistemas de tempo-real têm associados custos muito elevados. Este trabalho pretende demonstrar se é possível construir um sistema de tempo-real com componentes de baixo custo e genéricos. Antes de definir o sistema que se iria implementar é feita uma revisão de alguns conceitos de tempo-real (e.g. classes de tempo-real e escalonamento) e uma pesquisa de sistemas operativos também de tempo-real (e.g. RTEMS, VxWorks) para que existissem termos de comparação com os componentes escolhidos para o trabalho. Deste modo, neste trabalho é usada a plataforma RaspberryPi de baixo custo, o sistema operativo Raspbian com o patch PREEPT_RT e a ferramenta Safety Kernel, definido pelo KARYON, para gestão temporal de alguns componentes do sistema. Como não existia ainda um sistema implementado com estes três componentes uma parte deste trabalho passa primeiro por entender o que seria necessário para a integração dos mesmos. Após a implementação da arquitetura definida pelo trabalho foram feitos testes de desempenho ao sistema, inicialmente para garantir que se tinha capacidade de preempção por parte do escalonador do sistema operativo, numa segunda fase para testar os diferentes modos de definir a preempção para os processos (manual ou automática) e finalmente para perceber que tipo de desempenho se tinha comparativamente a outros trabalhos realizados com o Safety Kernel. Esta última fase de testes era especialmente importante por se ter expectativas de um aumento no desempenho do sistema implementado. Os resultados finais para os tempos obtidos com a arquitetura desenvolvida neste projeto mostram de facto um aumento de desempenho (valores dos tempos obtidos foram mais baixos) face aos resultados de trabalhos anteriores, no entanto não corresponderam exatamente ao que era esperado. Deste modo este trabalho mostrou que é possível ter um sistema, com garantias temporais razoáveis, de baixo custo e com componentes genéricos.

Palavras-chave: Tempo-real; Raspbian; Safety Kernel; Sistemas operativos de tempo-real; RaspberryPi.

Abstract

The majority of commercial operating systems and hardware that exists for implementing real-time systems have a very high cost associated to them. This work serves to demonstrate if it is possible to build a real-time system with cheap and generic components. Before defining the system that was going to be implemented a revision of some real-time concepts (e.g. real-time classes and scheduling) and also a research of real-time operating systems (e.g. RTEMS and VxWorks) are presented, these serve has to have some level of comparison to the components chosen for this project. Thus, in this work the chosen components were the cheap RaspberryPi platform, the Raspbian operating system with the PREEMPT_RT patch and the Safety Kernel tool, defined by KARYON, for managing the timeliness of some of the system components. Since no system implemented with these three components was ever tried, as a first step in this work there was a need to understand what was necessary to integrate said components. After implementing the architecture defined by this project several performance tests were made, initially to guarantee that the operating system's scheduler was indeed able to preempt tasks, and following that to test the different ways to define preemption for processes (manual or automatic) and finally to understand what kind of performance the system had compared to other projects done with the Safety Kernel. This last round of tests was of special importance due to the fact that there were some expectations of an improvement of the implemented system's performance. The final results for the times obtained with the architecture development by this project had in fact an improvement (lower values for the obtained times) when compared to the results of the other projects, although they match exactly the expectations. In the end, this work shows that it is possible to build a cheap system, with reasonable timeliness guarantees, with generic off-the-self components.

Keywords: Real-Time; Raspbian; Safety Kernel; Real-Time operating systems; RaspberryPi.

Conteúdo

Lista de Figuras

Capítulo 1

Introdução

1.1 Motivação

Os sistemas distribuídos de tempo-real são indispensáveis nos dias que correm. As suas características fazem com que apenas eles consigam solucionar certo tipo de problemas. No entanto, devido à sua especificidade, são na maior parte bastantes dispendiosos de implementar. Assim, nasce naturalmente uma necessidade de implementar sistemas deste tipo com equipamento de baixo custo. Um destes equipamentos é o RaspberryPi. Este mostra ter a potencialidade de concretizar o objetivo deste trabalho.

O desenvolvimento de sistemas de tempo-real em ambientes abertos, onde a influência de incertezas pode afetar o funcionamento do sistema, pode ser desafiante. Por exemplo, se se considerar sistemas autónomos cooperativos, que têm de comunicar em redes *wireless*, as incertezas que atrasos e falhas introduzem na comunicação tornam-se obstáculos que põem em causa o funcionamento correto desses sistemas se não forem tratados adequadamente.

As soluções típicas para desenvolver sistemas de tempo-real e de segurança críticos baseiam-se em estabelecer a priori todas as possíveis condições operacionais e dimensionar os recursos do sistema para o pior caso possível. Num sistema em que as incertezas não são fáceis de caracterizar, esta não parece ser uma aproximação adequada. Uma forma diferente é considerar modelos de sistemas distribuídos híbridos que permitem que certas partes do mesmo não funcionem de forma determinística. Mais, a hibridização arquitetural tem dado provas de que é um paradigma adequado para a implementação de sistemas distribuídos fiáveis com base em modelos de sistemas híbridos.

A hibridização de arquiteturas requer um suporte para o sistema que permita o particionamento do mesmo em domínios de execução diferentes, onde cada partição está isolada das outras, tendo propriedades de tempo e segurança próprias. Usufruindo deste particionamento espacial e temporal, pode-se ter diferentes tipos de componentes para concretizar diferentes partes do sistema.

1.2 Objectivos

O objetivo principal deste trabalho é perceber se é possível ter sistemas cooperativos distribuídos com uma componente de tempo-real utilizando equipamentos de baixo custo. O projeto KARYON (Kernel-based ARquitecture for safetY-critical cONtrol)[2] foca-se em ter coordenação e previsibilidade em veículos autónomos em ambientes que são inerentemente imprevisíveis e incertos. Estando este estudo relacionado com projeto KARYON e tendo em foco a parte do Safety Kernel criado no mesmo, o que se pretende é usar a plataforma RaspberryPi para dar suporte ao Safety Kernel. Esta plataforma foi escolhida precisamente por ser de baixo custo. Assim, o que será necessário estudar são as capacidades desta plataforma de respeitar garantias temporais, em particular para a execução do Safety Kernel.

Deste modo, este trabalho irá passar por testar um sistema operativo capaz de suportar as restrições de tempo-real do Safety Kernel e que tem a particularidade de ser compatível com a plataforma em estudo e estruturar de uma forma compreensível as suas garantias e características. De seguida, tendo percebido o funcionamento e as capacidades que o sistema operativo tem relativamente às garantias necessárias para tempo-real passa-se à fase de instalação do Safety Kernel no sistema. Tendo este a funcionar corretamente, serão feitos testes de tempos de execução dos módulos do Safety Kernel e uma análise dos mesmos. Os resultados dos testes ao serem comparados com os resultados obtidos num trabalho anterior [2, página 70] permitirão entender se a plataforma RaspberryPi é capaz de dar suporte, e ser parte integrante de um sistema distribuído cooperativo com componentes de tempo-real.

1.3 Metodologia

Para a realização deste trabalho, e para que se cumpram os objetivos de uma forma produtiva e fiel aos mesmos, o caminho tem que passar em primeiro lugar por perceber o que são sistemas de tempo-real, porque existe uma necessidade de ter este conceito, como se utilizam esses sistemas e que restrições estes sistemas têm. São também revistos alguns conceitos associados a sistemas de tempo-real (e.g. Escalonamento, prioridade de tarefas e interrupções 2.1).

Após uma esquematização dessa informação deve passar-se para o estudo do que nos interessa neste trabalho que é perceber se é possível ter um sistema com garantias de tempo-real em plataformas de baixo custo, mais precisamente utilizando a plataforma em foco que é o RaspberryPi. Utilizando o sistema operativo Raspbian com o *patch* PREEMPT_RT, iremos estudar então as capacidades de tempo-real do sistema, ou seja, serão feitos testes de escalonamento do sistema para perceber se o sistema de facto permite ou não prioritizar tarefas e portanto se permite ter preempção. Este passo é importante para

que se possa garantir que as tarefas de tempo-real que serão executadas no sistema têm prioridade máxima.

Finalmente, irá testar-se o Safety Kernel na plataforma para perceber se se consegue ter resultados que demonstrem que este sistema consegue fornecer garantias de tempo-real razoáveis.

1.4 Contexto

Tendo como objetivo principal aumentar as capacidades funcionais e a variedade de componentes, mais especificamente na utilização de componentes de baixo custo, num sistema distribuído com particionamento espacial e temporal, este trabalho relaciona-se com o projeto KARYON, mais especificamente utilizando o Safety Kernel nele definido.

Tendo o projeto KARYON um foco em coordenação e cooperação em sistemas imprevisíveis e incertos, mais uma vez um dos pontos de interesse é ter neste tipo de sistemas baixos custos na execução destes mecanismos, excluindo assim componentes que sejam especializados e portanto consideravelmente mais caros (e.g., sensores e hardware especializado, redundância de componentes e subsistemas). Claro que não podemos esquecer que apesar de ter sistemas de baixo custo ainda assim não se pode ter uma quebra excessiva na desempenho do sistema. Existe uma relação de compromisso entre baixo custo e desempenho, já que utilizar componentes de baixo custo representa na maioria das situações ter heterogeneidade no sistema. Assim, o KARYON trata deste problema propondo três princípios que definem um novo padrão arquitetural: hibridização arquitetural, nível de serviço e um modelo abstrato de sensor. É, portanto, no aspeto de hibridização arquitetural que este trabalho se relaciona com o projeto, dando especial relevância ao componente Safety Kernel do KARYON.

1.5 Estrutura do documento

Este documento encontra-se organizado da seguinte forma:

- Capítulo 2 - Panorâmica de Tempo-Real: descrição de conceitos relevantes para a área de tempo-real.
- Capítulo 3 - Arquitetura: descrição dos componentes utilizados no trabalho, são eles a plataforma RaspberryPi e o Safety Kernel.
- Capítulo 4 - Implementação: nesta secção fala-se do processo de integração do Safety Kernel com o RaspberryPi, ou seja, questões que existiam antes de verificar a compatibilidade destes componentes e que resultados se obteve relativamente a essas questões.

- Capítulo 5 - Avaliação: aqui descreve-se o processo de testes sobre as capacidades de tempo-real do sistema. Descreve-se que testes foram usados e que resultados se obteve.
- Capítulo 6 - Conclusão: apresentam-se algumas conclusões sobre o trabalho e perspectivas para trabalhos futuros.

Capítulo 2

Panorâmica de Tempo-Real

Este capítulo serve para explicar o âmbito em que este trabalho se integra. É aqui que se descrevem mais pormenorizadamente alguns conceitos, da área de tempo-real, que são usados durante este trabalho.

2.1 Conceitos de Tempo-Real

Em primeiro lugar, uma descrição do que é um sistema de tempo-real. O que define este paradigma é a necessidade de relacionar eventos do ambiente onde o sistema se encontra com o funcionamento do próprio sistema [13]. O ambiente tem o seu próprio ritmo, os eventos podem ser, e são muitas vezes, imprevisíveis em termos temporais, pelo que o sistema tem que se poder adaptar a esta característica. Daqui podemos dizer que um sistema de tempo-real é um sistema cujo progresso é especificado em termos de requisitos temporais definidos pelo ambiente [9]. Como consequência desta noção temos o aparecimento de diferentes classes de sistemas de tempo-real. Tem-se também um conjunto de algoritmos de escalonamento de tarefas para que estes sistemas sejam capazes de escalonar os seus processos e conseguir respeitar as restrições temporais das mesmas.

2.1.1 Classes de Tempo-Real

O nível de criticidade das restrições temporais para que um evento do ambiente seja tratado corretamente é o que vem distinguir as seguintes classes de sistemas de tempo-real:

- Sistemas de tempo-real estrito, onde falhas temporais devem ser evitadas (e.g., sistema *on-board* de controlo de voo, *fly-by-wire*);
- Sistemas de tempo-real lato, onde falhas temporais ocasionais são aceites (e.g., sistema para reserva de voos *on-line*);
- Sistemas de tempo-real de missão-crítica, onde falhas temporais devem ser evitadas e falhas ocasionais são tratadas como um evento excecional (e.g., sistema de controlo de tráfego aéreo).

É ainda possível fazer uma distinção entre sistemas de tempo-real distribuídos e centralizados. No caso dos distribuídos:

- As garantias temporais têm que ser respeitadas através de um sistema de componentes interligados por uma rede;
- Quando respeitadas, as garantias temporais podem ser ou estar associadas a outros atributos, como modularidade, separação geográfica, independência de falhas, distribuição de carga, entre outras.

2.1.2 Escalonamento

Para que as regras temporais desejadas para as diferentes tarefas sejam respeitadas é preciso ter um mecanismo que faça a gestão da janela temporal ao longo da qual se vai ter a execução das tarefas e da gestão de recursos que são partilhados pelas tarefas.

Deste modo aparece um conceito de escalonamento. Escalonamento é o que define como é que um recurso é partilhado pelas diferentes atividades do sistema, de acordo como uma dada política. No caso de sistemas de tempo-real o escalonamento prende-se com a necessidade de providenciar uma utilização dos recursos disponíveis de forma correta para permitir que o sistema respeite requisitos temporais.

A definição da política a seguir para a gestão correta dos recursos no sistema pode variar por diferentes razões. No caso do escalonamento em tempo-real, como o objetivo é chegar a um ponto em que todas as tarefas, de tempo-real, são executadas atempadamente, querendo isto dizer que não falham metas de execução, as políticas têm de ter em conta estas metas. Aqui, a preempção de tarefas é baseada na ocorrência de eventos que têm que ser processados num dado intervalo de tempo, uma forma de fazer esta gestão é atribuir prioridades aos eventos.

Dependendo do tipo de eventos com os quais o sistema tem que lidar, também é possível fazer uma análise a priori do tipo de escalonamento a fazer. Quer isto dizer, que se se souber a periodicidade de um evento e esta for constante é possível calcular de uma forma estática a janela temporal de execução deste evento. Por outro lado, se se tiver eventos que são irregulares, ou seja, não é possível prever quando acontecem, o cálculo da janela temporal tem que ser feito de forma dinâmica. Temos, então, dois conceitos importantes: escalonamento estático e escalonamento dinâmico.

- Escalonamento estático: o escalonamento segue um plano predefinido e é assumido que é possível prever todos os tempos (tempo de execução, conflito de recursos partilhados).
- Escalonamento dinâmico: o escalonamento é calculado em tempo de execução e as decisões de escalonamento são baseadas em tarefas prontas a executar.

O passo que segue é a atribuição de prioridades. Esta é uma tarefa que também não é trivial, ao analisar o tipo de eventos que o sistema tem que processar é necessário também perceber se se pode ter eventos sempre com a mesma prioridade relativamente a outros eventos ou se essa prioridade tem que ser alterada por alguma razão.

A prioridade de uma atividade deve ter em consideração alguns aspetos: deve indicar o seu nível de urgência em ser processada, deve variar de acordo com as propriedades de tempo-real do sistema e deve existir para que o escalonador possa tomar decisões de preempção.

Assim, para um escalonador saber que política seguir na atribuição de prioridades das atividades do sistema, existem algoritmos que ajudam no cálculo da janela temporal global do sistema. Estes algoritmos baseiam-se nos seguintes conceitos: tempo máximo da atividade durante uma execução contínua (WCET), tempo limite em que a atividade pode acabar (*deadline*) e tempo de sobra para a execução da atividade (*laxity*).

- Rate Monotonic (RM): baseia-se em prioridades fixas e as prioridades são definidas a priori.
- Earliest-Deadline-First (EDF): baseia-se em prioridades dinâmicas e a prioridade é inversamente proporcional à *deadline*.
- Least-Laxity (LL): baseia-se em prioridades dinâmicas e a prioridade é inversamente proporcional à *laxity*.
- Deadline Monotonic: baseia-se em prioridades fixas e as prioridades são inversamente proporcionais à *deadline*.
- Prioridade mais alta primeiro: baseia-se em prioridades fixas e as prioridades são definidas a priori.
- First-Come-First-Served (FCFS): baseia-se em prioridades atribuídas de acordo com o estado de preparação das tarefas (a primeira a estar pronta a executar é a que tem prioridade mais alta).

2.1.3 Algoritmos de Escalonamento

Rate Monotonic

No RM o escalonamento é estático, ou seja, as prioridades são fixas e baseadas no período de execução da tarefa. A prioridade das tarefas é inversamente proporcional ao período. Uma característica importante do Rate Monotonic é que se um conjunto de tarefas não é escalonável neste, então, também não o é para algoritmos da mesma classe (i.e. prioridades fixas). Assim, se o escalonamento é possível, então, diz-se ótimo.

As seguintes assunções são necessárias:

- As atividades são independentes e periódicas.
- A *deadline* é igual ao período.
- O WCET é defenido e conhecido.
- O tempo de mudança de contexto é de valor negligenciável.

Earliest-Deadline-First

Neste tipo de escalonamento temos uma abordagem de escalonamento dinâmico através da execução baseada em prioridades. A ordenação das tarefas varia de acordo com o tempo absoluto da *deadline*, sendo a prioridade mais alta atribuída à tarefa com a *deadline* mais baixa. Mais uma vez, temos a situação em que o escalonamento é ótimo, sendo que se o escalonamento não é possível com o EDF, também não será para o resto da classe de algoritmos com prioridades dinâmicas.

Least-Laxity

O LLF pertence à classe dos algoritmos dinâmicos. Neste algoritmo aparece uma situação em que temos uma sutilidade na sua definição comparativamente ao EDF. Apesar de à primeira vista parecerem iguais, visto terem a ver diretamente com a *deadline*, a diferença está no cálculo do tempo para a prioritização do conjunto das tarefas. Ao passo que no EDF a prioridade varia apenas com a *deadline*, no LLF existe o conceito de *laxity*. Este é calculado da seguinte forma: *laxity* = *deadline* - tempo de execução restante. Assim, é este valor que é usado para prioritizar as tarefas, tendo a tarefa com menor valor de *laxity* a maior prioridade.

Mais uma vez as tarefas são independentes e são selecionadas preemptivamente por ordem de prioridade. Como já referido, as prioridades são alteradas dinamicamente, no entanto a prioridade de uma tarefa em execução mantem-se constante e apenas nas restantes é que o valor é recalculado. Este processo leva por vezes a uma situação de *thrashing*, ou seja, constante preempção de tarefas. Como se pode perceber pelo exemplo da Figura 2.1, temos uma alternação constante entre as três tarefas do escalonamento. Este processo deriva da atualização dinâmica da prioridade das tarefas, que obriga a preemptar a tarefa em execução porque outra ganhou uma prioridade mais alta.

Deadline Monotonic

Anteriormente falou-se em tempo absoluto da *deadline* para o EDF. No caso do DM como o escalonamento é baseado em prioridades estáticas, falamos antes do tempo relativo da *deadline*. As prioridades das tarefas variam, no entanto, da mesma forma, a tarefa com tempo mais baixo tem prioridade mais alta.

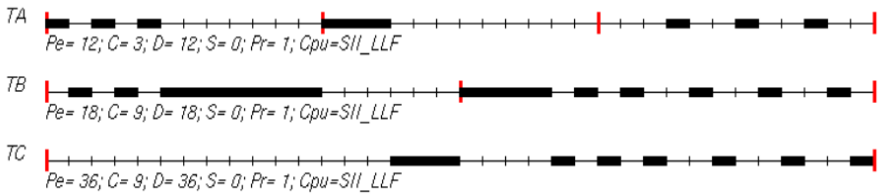

Figura 2.1: Exemplo de *thrashing* no Least Laxity First

Prioridade mais alta primeiro

Este algoritmo é bastante simples, as prioridades das tarefas são definidas a priori e a preempção é feita às tarefas de prioridade mais baixa pelas de prioridade mais alta.

First-Come-First-Served

No FCFS as prioridades são atribuídas de acordo com prontidão das tarefas. Isto significa que a tarefa que está pronta a executar mais depressa fica com a prioridade mais alta.

2.1.4 Acesso a secções críticas e Interrupções

Visto nas secções anteriores se ter falado em preempção e em várias formas de gerir as janelas temporais de execução por parte do escalonador, de seguida fala-se de alguns dos mecanismos que existem para implementar preempção e que são utilizados pelo escalonador.

Spinlock

O *spinlock* é um tipo de bloqueio que regra geral é implementado usando uma técnica chamada *busy waiting*, ou seja, quando uma tarefa tenta obter este bloqueio fica constantemente a ”perguntar”se o mesmo está passível de ser obtido. Assim, enquanto não for possível obter o bloqueio, a tarefa continua a gastar recursos do CPU para avaliar o estado do bloqueio. Mais, este tipo de bloqueio apenas pode ser obtido por uma tarefa de cada vez. Quando a tarefa obtém então o bloqueio, pode já aceder à zona crítica guardada pelo mesmo. Este tipo de bloqueios é geralmente implementado usando operações atómicas. O seu uso deve ter em consideração o gasto de ciclos de CPU, pelo que este tipo de bloqueios deve ser usado para instruções rápidas e em que valha a pena não usar mudança de contexto (e.g., mudança de contexto de hardware para contexto de processos).

Semáforo

Os semáforos são normalmente usados para sincronização entre processos. Este tipo de bloqueios têm, por contraste aos *spinlocks*, muito pouco tempo de ”*spinning*”ou até nenhum. Quando uma tarefa tentar obter o bloqueio e isso não é possível a tarefa é parada,

o que permite outras tarefas serem executadas, até poder ser reescalonada. Esta característica faz com que este tipo de bloqueios sejam bastante eficientes em termos de custo para o CPU. Uma outra particularidade é o facto de várias tarefas poderem obter o bloqueio, esse controlo é feito por intermédio de um contador. Os semáforos devem ser usados quando é necessário manter o acesso a zonas críticas durante bastante tempo, ou caso se tenham muitas tarefas a competir pelo mesmo semáforo.

Interrupção

Uma interrupção é um sinal enviado ao CPU para assinalar um evento. As interrupções podem ser de hardware ou de software. Uma interrupção é usada por um componente de hardware ou software para assinalar ao CPU a necessidade de parar a execução corrente e executar uma outra tarefa. Num sistema de tempo-real é importante ter controlo total sobre as interrupções porque caso contrário não é possível garantir as janelas temporais de execução das tarefas. Isto porque não tendo esse controlo podem-se gerar situações em que tarefas estão constantemente a ser interrompidas, para que outras executem, e por isso não conseguem cumprir a janela temporal requirida para a sua execução.

2.2 Características dos Sistemas Operativos de Tempo-Real

Esta secção servirá não para uma definição de sistema operativo de tempo-real, mas sim do que é esperado de um sistema destes para uma plataforma com poucos recursos como o RaspberryPi. O interesse aqui será perceber que tipo de restrições podem ou não existir para sistemas de segurança crítica.

Entender o que é desejável para um sistema de tempo-real é também importante para este estudo, já que é nesse âmbito que as limitações de sistemas de baixo custo podem ter impacto. Apesar de existir alguma divergência nas definições exatas para os conceitos das características principais destes sistemas, de seguida apresenta-se alguns pontos essenciais a observar:

- Qualidade de serviço;
- Quantidade de código que necessita ser inspecionado para garantir qualidade de serviço;
- API disponível;
- Complexidade relativa do sistema operativo e aplicações;
- Isolamento de faltas;
- Que configurações de hardware e software são suportadas?

Quantidade de código que necessita ser inspecionado para garantir qualidade de serviço

Ao alterar uma caraterística de um sistema operativo a nível do funcionamento do seu kernel requer que se considere as repercuções na compatibilidade do resto das suas funções. Portanto, ao acrescentar uma propriedade nova que acomode tempo-real tem que se pensar que quantidade de código do resto do sistema operativo é preciso inspecionar e validar, o ideal é que essa quantidade seja a menor possível, para que o trabalho de depuração, retro compatibilidade e de evolução seja mais fácil.

No que diz respeito a zonas de código críticas que devem ser inspecionadas, deve ter-se em especial atenção a zonas que contenham: código que use qualquer tipo de interrupção ou bloqueio que possa afetar o funcionamento do novo código, tratamento de interrupções e preempções a baixo nível, manipulação de recursos que possam causar deadlocks ou atrasos (i.e. hardware que possa afetar de alguma forma o fluxo de execução das tarefas), e código do escalonador de tarefas.

API disponível

Ao estudar estes sistemas torna-se clara a necessidade de que existam APIs (Application Programming Interface) disponíveis para se poder evoluir e trabalhar nestes ambientes. Uma API define funcionalidades independentemente das suas implementações, o que permite ter várias implementações sem comprometer a interface e facilitar o desenvolvimento aplicacional. Algumas APIs conhecidas são: API para a norma POSIX, DirectX para Windows e OpenGL que serve multiplataformas.

Complexidade arquitetural

Os sistemas de tempo-real devido às suas características e necessidade de respeitar restrições temporais tornam a noção de onde estas propriedades devem ser tratadas e a forma como devem ser tratadas num assunto sensível. A complexidade do código seja ele ao nível de aplicação ou de sistema operativo é de extrema relevância para que não sejam quebradas as regras impostas pelo ambiente onde se vai utilizar o sistema. Um exemplo, numa situação em que se está a fazer alterações a um sistema de controlo de voo de um avião, onde claramente a existência de contingentes para faltas é de crítica necessidade, inserir complexidade extra e desnecessária pode pôr em risco vidas.

Assim, deve-se ter em consideração onde se quer ter ganhos em termos de carga de trabalho. Deve ser tida em conta a quantidade de código extra que se vai se gerar e onde se vai gerar, se é preferível acrescentar peso sobre o sistema operativo ou se se deve ter uma maior separação de zonas críticas para faltas. Claro que a redundância para ter tolerância a faltas é sempre de ter em consideração também, mas mais uma vez o número de sistemas a que essa redundância diz respeito também faz variar a carga de trabalho no

sistema global.

O uso de ferramentas como nano-kernels e supervisores pode e deve também ser tida em consideração na separação de carga de trabalho. O uso de intermediários ou camadas de abstração pode ajudar no tratamento de faltas.

De facto, as opiniões sobre que design usar para dado sistema são sempre variadas e não parece haver nunca um consenso pelo que a tarefa de otimizar ou modificar um sistema destes é sempre morosa e complexa.

Isolamento de faltas

Quando se está a tratar de sistemas com partições e níveis de abstração deve-se ter em atenção que tipo de faltas podem surgir que coloquem em risco as suas propriedades e garantias temporais. Portanto, deve-se questionar que faltas põem em perigo o código de tempo-real e que tipo de faltas devem ser isoladas. É importante perceber que não são todos os erros aplicacionais que impedem uma dada tarefa de ser executada corretamente. A questão passa por entender se o erro que levou a uma execução incorreta ou mesmo à não execução de uma dada tarefa afeta de algum modo as restrições temporais do resto do sistema. Por exemplo, um sistema em que a parte de tempo-real trata da parte de comunicação com um drone autónomo enviando-lhe informações telemétricas para que este possa navegar corretamente o espaço em que se encontra, e a parte sem restrições de tempo-real está a compilar e a guardar dados estatísticos da dita telemetria. Aqui pode dar-se uma situação de quebra das restrições temporais se ocorrer uma falta durante o acesso à zona de recursos partilhados levando, por exemplo, a uma situação de bloqueio na execução das restantes tarefas. É para evitar situações como esta que se torna crítico ter isolamento de faltas.

Para o isolamento de faltas existem alguns pontos que se destacam a nível de relevância: desativação de interrupções excessiva, desativação excessiva da capacidade de preemptar, corrupção de memória ou acesso direto a memória de forma incorreta, e semáforos, mutexes ou bloqueios que se mantêm durante demasiado tempo impedindo código de tempo-real de aceder a recursos que necessita.

Que configurações de hardware e software são suportadas?

O funcionamento a nível de recursos, tempo de execução de tarefas e comunicação (BUS) de operações de E/S num sistema de segurança crítico pode impactar bastante na capacidade deste manter as suas garantias temporais, sejam essas operações feitas a nível de hardware ou software. No entanto, não basta pensar em termos de funcionamento, é também preciso considerar a quantidade de componentes que o sistema integra. Devemos usar SMP, devemos ter mais do que um CPU, devemos ter multi-threading, devemos ter muito ou pouco espaço de disco rígido e esse espaço deve ser particionado ou não, as considerações na construção de um sistema são de facto bastantes. Devido a estas

questões todas, muitos elementos de hardware e software restringem a quantidade e tipo de recursos com que são compatíveis, principalmente se tiverem de respeitar limitações temporais.

2.3 Sistemas Operativos de Tempo Real

Quando pensamos em sistemas operativos raramente a primeira preocupação tem a ver com exigências temporais estritas. No entanto, quando se trabalha no âmbito de tempo-real é preciso não só ter esse aspeto em consideração, como também é preciso perceber que tipo de exigências existem e que tipo de tempo-real é necessário respeitar (Página 5).

Assim, na secção seguinte faz-se uma breve descrição de alguns sistemas operativos de tempo-real mais conhecidos e usados na industria. Alguns destes sistemas são comerciais, outros são abertos e a custo zero, esta distinção é feita para termos uma relação com o objetivo principal do trabalho (1.2).

2.3.1 Xenomai x86

O Xenomai é uma *framework* Linux aberta que está disponível para qualquer um usar. O objetivo deste projeto é permitir que aplicações industriais possam ser migradas para sistemas Linux e que deixe de ser necessário ter sistemas proprietários.

O Xenomai permite usar várias APIs de sistemas operativos embebidos em sistemas Linux, e quando o kernel do Linux que está a ser usado não permite respeitar as garantias temporais desejadas pode colmatar esta situação usando garantias temporais mais fortes baseadas numa tecnologia de co-kernel.

O Xenomai aproveita as características comuns a muitos sistemas de tempo-real (RTOS) especialmente no que diz respeito ao escalonamento de *threads* e sincronização. Essas semelhanças são aproveitadas e daí nasce uma implementação de um núcleo através da exportação de um conjunto genérico de serviços. Esses serviços agrupados numa interface de alto nível podem então ser usados para emular interfaces de programação para aplicações de tempo-real, mimetizando a API do kernel do sistema embebido correspondente.

Fundamentalmente, o Xenomai segue uma filosofia de co-kernel, querendo isto dizer, que se tem um escalonador de tempo-real para tarefas de tempo-real e o escalonador Linux para tarefas sem regras temporais estritas. O que isto implica na interação destes dois é, de uma forma direta, que ambos coexistem mas o de tempo-real tem uma prioridade superior ao nativo (i.e., Xenomai com prioridade superior a Linux). Para isso, aparece uma interseção dos eventos que estão direcionados ao Linux para que se garanta a prioridade mais alta das tarefas de tempo-real. O tipo de eventos que interessa capturar são eventos como interrupções ou *traps*. Pois são estes que podem introduzir perturbações na correta execução de processos e que portanto podem afetar as garantias temporais que têm que

ser respeitadas. Uma interface genérica para gestão destes eventos é a ADEOS (Adaptive Domain Environment for Operating Systems). Pode-se perceber a hierarquia dos vários componentes na arquitetura Xenomai pela figura 2.2.

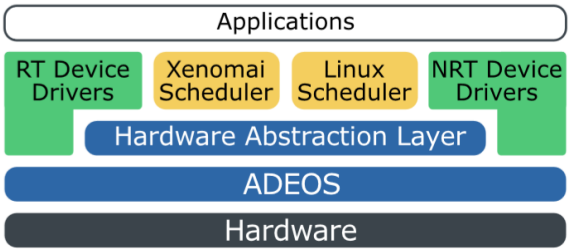

Figura 2.2: Arquitetura do Xenomai

Esta interface oferece um camada de virtualização de recursos (interrupções e *traps*) que podem ser acedidos através de domínios (figura2.3). Estes domínios estão separados por níveis de prioridade e podem ser tanto uma simples aplicação como mesmo um kernel. Os domínios podem também ter conhecimento da existência de outros domínios e portanto ter acesso aos mesmos. Assim, o ADEOS gere o fluxo de eventos distribuindo-os pelos vários domínios.

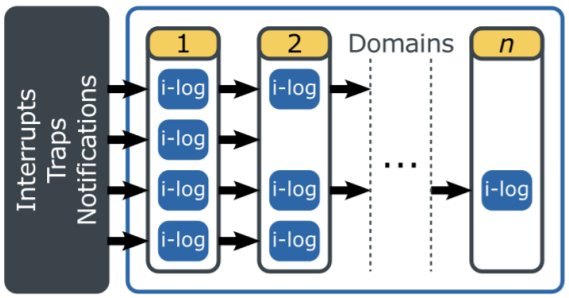

Figura 2.3: Camadas de domínios e fluxo de eventos do ADEOS

Deste modo os eventos fluem por ordem de prioridade de domínios, ou seja, do domínio com maior prioridade para o com mais baixa. Os domínios podem então processar os seus eventos e podem passá-los para os domínios seguintes ou mesmo atrasar outros eventos, isto acontece precisamente para que os eventos se processem de forma correta e respeitando as restrições temporais do sistema. O ADEOS garante sempre que os eventos são tratados pelo Xenomai antes de chegarem ao Linux, já que a ordem de prioridade é, como referido, Xenomai (tempo-real) e depois o Linux (não tem restrições temporais estritas). No entanto, para que esta separação seja bem gerida existe um terceiro elemento que gere as interrupções a ser propagadas para o Linux, este componente chama-se *Interrupt-Shield*. É, portanto, neste componente onde é concretizado o atraso, que o sistema requere, da chegada dos eventos ao Linux. Deste modo, quando não existirem mais tarefas a executar no domínio do Xenomai as interrupções são então propagadas

para os próximos domínios, neste caso o Linux, e o escalonador dos mesmos pode então começar a correr.

Claro está que para usufruir destas novas características é necessário o sistema conseguir responder a todo um novo conjunto de instruções, ou pelo menos que se possa redefinir instruções já existentes. Para isto existem APIs, que podem ser diversas. Neste caso, iremo-nos focar principalmente na API POSIX por ser uma das mais utilizadas e com provas já dadas. Para que a API consiga comunicar instruções corretamente ao ADEOS temos uma camada, designada por Nucleus, que fornece uma abstração da entidade de tempo-real do sistema operativo, esta contém os serviços básicos de tempo-real, entidades e primitivas. Isto quer dizer que é o Nucleus que fornece interação com o estado do escalonador e das tarefas e *threads*, serviços do Temporizador e gestão de interrupções e serviços de alocação dinâmica de memória. É então através desta interface que a API pode invocar os vários serviços que necessita para definir as restrições temporais dos processos a executar.

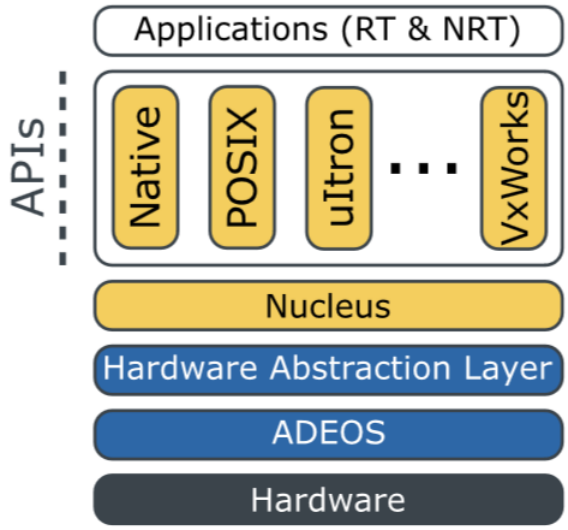

Figura 2.4: APIs e Nucleus no ADEOS

Assim, para executar *threads*, o Xenomai oferece um conceito de modos primário e secundário, que muito simplesmente querem referenciar o escalonador de tempo-real e o escalonador sem garantias temporais, respetivamente. No entanto, ao executar uma *thread* num destes modos não impede que a mesma migre para o outro modo, i.e. a *thread* pode começar a executar em modo primário e migrar para modo secundário caso faça uma invocação sem restrições de tempo-real (e.g., instrução printf() em C). A prioridade da tarefa para o escalonador não é, apesar dessa migração de modo, alterada. A prioridade em Xenomai é definida por um valor inteiro que varia entre 0 e 99 com a prioridade sendo atribuída por ordem crescente desses valor (i.e., valor mais alto tem uma prioridade superior).

A mudança de modo processa-se então da seguinte forma:

- O Xenomai interceta uma chamada ao sistema através do ADEOS;
- É feita uma preempção da *thread*;
- É escalonada para Linux com a mesma prioridade;
- A chamada ao sistema é executada em Linux;
- Finalmente a *thread* é reescalonada para o Xenomai.

2.3.2 Raspbian-Sep-2014-RT-B+

Esta distribuição Debian é uma versão, na perspetiva de tempo-real, melhorada do Raspbian já existente. Foi-lhe acrescentada a capacidade de preempção de código do kernel do sistema operativo com o objetivo de minimizar a quantidade de código que não é preemptável. Esta evolução foi feita através de um *patch*, PREEMPT_RT , que usa as capacidades de multiprocessamento simétrico do kernel de Linux para que as alterações necessárias para permitir integrar operações de preempção extra não necessitassem de uma reestruturação total do kernel.

O *patch* referido anteriormente tem sido modificado de acordo com a necessidade ter garantias temporais mais ou menos restritas. O *patch* PREEMPT, onde se tenta melhorar as capacidades de preempção do próprio kernel do sistema operativo, diminuindo o tempo em que operações com prioridades altas podem ser bloqueadas. O resultado deste tipo de alterações é que temos uma maior capacidade de prioritização de tarefas, podendo mesmo interromper a execução de uma tarefa mesmo que esta esteja a meio de operação no kernel. Uma versão deste *patch*, chamada de CONFIG_PREEMPT, permite já fazer preempção da maior parte do kernel, salvo algumas exceções como rotinas de atendimento de interrupções e regiões guardadas por *spinlocks*, o que resulta em latências (no pior caso) na ordem das unidades de milisegundos. Se se necessitar de valores de latências mais baixos temos então a versão de *patch* CONFIG_PREEPT_RT que vem trazer capacidades de tempo-real mais estritas. Esta versão já permite preemptar também algumas regiões guardadas por *spinlocks*, mover as rotinas de atendimento de IRQs para *threads* bem como outras operações de tempo-real. É esta versão do *patch* que foi usada na distribuição Debian que figura na arquitetura definida neste trabalho.

Ao utilizar este sistema operativo é-nos permitido definir as prioridades das tarefas e a política do escalonamento das mesmas de uma forma dinâmica. As prioridades podem ser atribuídas a priori usando a função sched_setschduler() [3] ou em tempo de execução, às várias tarefas (incluindo as tarefas do kernel). Esta capacidade traz a vantagem necessária para tirar partido do *patch* referido acima, já que se torna possível definir tarefas como tarefas de tempo-real, ou seja, com uma prioridade extremamente elevada. Para fazer estas atribuições em tempo de execução é necessário o seguinte comando: *chrt [-política] -p [1..99] {pid}*

As políticas disponibilizadas pelo sistema são:

- SCHED_FIFO(-f): política de *first in-first out*.
- SCHED_RR(-r): política de *round robin*.
- SCHED_OTHER(-o): a partilha de tempo de escalonamento é equilibrada e justa entre as várias tarefas.
- SCHED_BATCH(-b): para execução de tarefas em bloco.

As únicas políticas que permitem diferenciação para tempo-real são o SCHED_FIFO e o SCHED_RR. Para o caso em que é necessário subir o nível de prioridade de uma tarefa para tempo-real (prioridades acima do valor 90, inclusive) é necessário estar em modo *sudo*. Ao alterar estes valores é possível avaliar os mesmos através de comandos como *top*, onde as prioridades de cada atividade estão visíveis.

Para definir a prioridade do processo ou tarefas a priori pode-se usar a função sched_ _setscheduler(). Os parâmetros necessários são: o id do processo a aplicar a política de escalonamento (pid_t pid), o tipo de política(int policy) e os parâmetros da política a aplicar(const struct sched_param *param).

2.3.3 AIR

O AIR [11] é um trabalho importante para o avanço na utilização de sistemas operativos abertos já conhecidos, como kernels RTOS ou mesmo sistemas operativos genéricos (e.g., Linux embebido e de tempo-real). O AIR é criado para funcionar em conjunto com sistemas baseados na especificação ARINC 653, esta arquitetura é um passo essencial no mundo aeronáutico (e.g., sistema operativo do Airbus A380 e Boeing 787) como bloco fundamental do conceito IMA (Integrated Modular Avionics), e mais recentemente no campo de sistemas espaciais com um relevante interesse por parte da Agencia Espacial Europeia (ESA) que tem financiado vários estudos para analisar a utilização e as capacidades de particionamento espacial e temporal em sistemas de bordo espaciais.

A ideia fundamental por trás do AIR é providenciar as funcionalidades estipuladas pela norma ARINC 653 que estão em falta nos kernels de sistemas operativos de tempo-real genéricos. Para isto tem que ser feito um encapsulamento destas funcionalidades em componentes com uma interface bem definida e adicioná-las à arquitetura do sistema operativo, deste modo o AIR preserva a independência entre o hardware e o sistema operativo definida na especificação do ARINC 653.

Existem três componentes fundamentais na arquitetura do AIR (Figura 2.5): o kernel de gestão de partições (Partition Management kernel - PMK), é um microkernel que trata do *i*) escalonamento e execução das partições e *ii*) da comunicação inter partições; o kernel dum sistema operativo de tempo-real por partição (Partition Operating System); e

a interface APEX, que define um conjunto de serviços em conformidade com a norma ARINC 653.

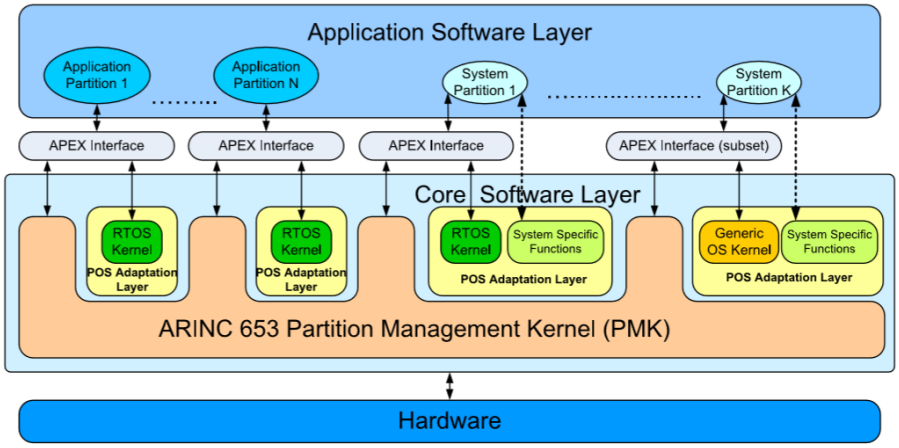

Figura 2.5: Visão global da arquitetura AIR.

A arquitetura AIR permite a integração dos sistemas operativos sem serem necessárias alterações fundamentais a um determinado kernel POS, de facto só o processo de inicialização do sistema operativo e a rotina de atendimento do relógio de sistema é que precisam de ser adaptados.

A segregação temporal é assegurada através de um esquema de escalonamento de partições cíclico fixo sobre um MTF (major time frame), em conformidade com a norma ARINC 653. Esta segurança é fornecida ao nível do PMK (Figura 2.6). Um componente dedicado (AIR PMK Partion Scheduler) confirma a cada *tick* do relógio de sistema se uma preempção duma partição vai ocorrer, se assim for, outro componente (AIR PMK Partition Dispatcher) tem que guardar o contexto da partição em execução e restaurar o contexto da partição herdeira. O escalonador nativo do POS da partição herdeira é notificado dos *ticks* de relógio que passaram desde a sua última vez que foi preemptado, e deste modo o tempo de sistema da partição herdeira ajusta-se a um referencial comum entre as partições.

Para dar suporte a particionamento espacial, no AIR foi seguido um desenho altamente modular. Os requisitos de particionamento espacial são descritos em tempo de execução através de uma camada de alto nível abstrata independente do processador. Um conjunto de descritores é dado por partição, correspondentes principalmente aos vários níveis de execução de uma atividade (e.g. aplicação, kernel do POS e AIR PMK) e às suas diferentes secções de memória como se pode observar na Figura 2.7.

2.3.4 RTEMS

O RTEMS é um sistema operativo para sistemas multiprocessador [12]. É de tempo-real e suporta uma variedade de APIs e padrões de interfaces. Este sistema operativo pro-

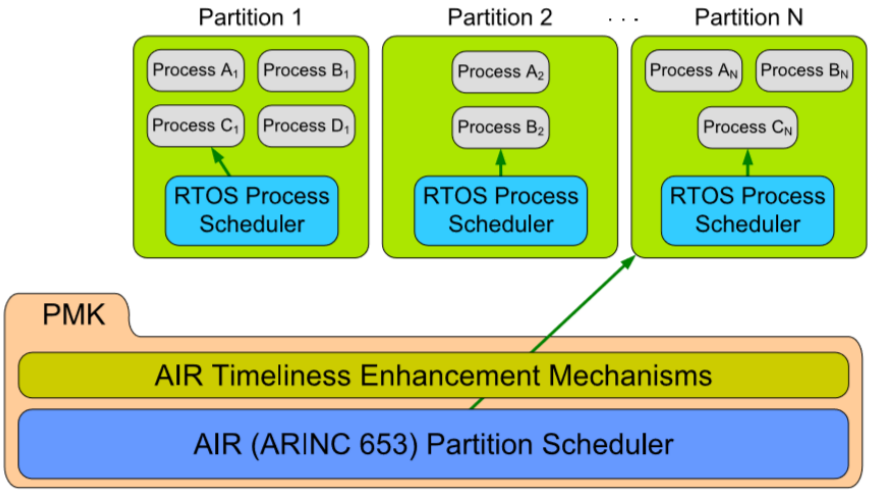

Figura 2.6: Integração do escalonador de partições do AIR PMK e do escalonador de processos do POS.

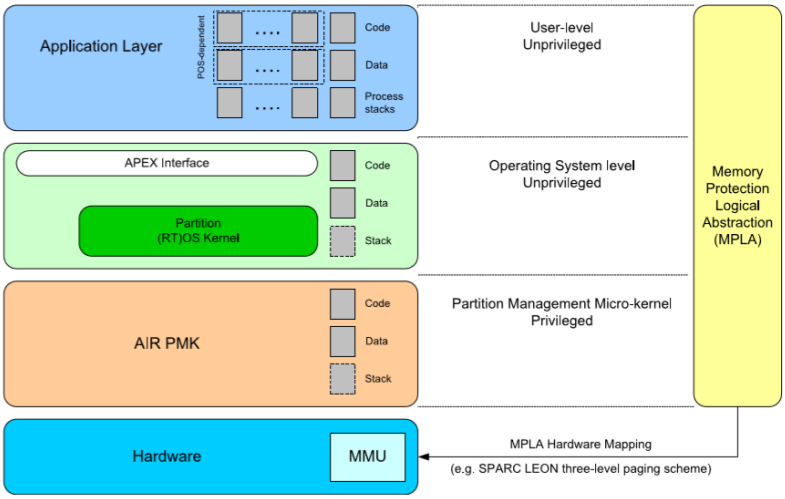

Figura 2.7: Esquema de segregação espacial do AIR.

vidência um ambiente de alto desempenho para aplicações embebidas, incluindo capacidades de multitarefa, sistemas de microprocessador homogéneos e heterogéneos, escalonamento (por eventos, por prioridade ou com preempção), sincronização e comunicação entre tarefas, herança de prioridades, gestão de interrupções responsivo, alocação dinâmica de memória e alto nível de configurabilidade de utilizadores. O RTEMS é um sistema operativo RTOS de custo zero e aberto construído para sistemas altamente embebidos com o objetivo de competir com sistemas operativos proprietários e comerciais. Consegue dar suporte aos requisitos temporais mais restritos, pelo que consegue dar suporte a sistemas de tempo-real estrito. Apesar de muitas funcionalidades e suporte para diferentes plataformas deste sistema operativo terem sido desenvolvidas pela comunidade, o RTEMS é mantido pela On-Line Research Corporation (OAR). A organização do RTEMS está feita

para respeitar os seguintes requisitos:

- Encorajar o desenvolvimento de componentes de uma forma modular.
- Isolar o processador de código específico dos componentes, e ainda assim permitir ter o máximo de código fonte possível partilhado pelos múltiplos processadores e placas.
- Permitir a vários utilizadores do RTEMS fazerem compilações simultâneas do RTEMS.

O RTEMS pode como conceito ser caracterizado por três camadas: suporte de hardware, kernel e APIs. O utilizador desenvolve então a aplicação usando as APIs. Na Figura 2.8 mostra o conceito da arquitetura do RTEMS.

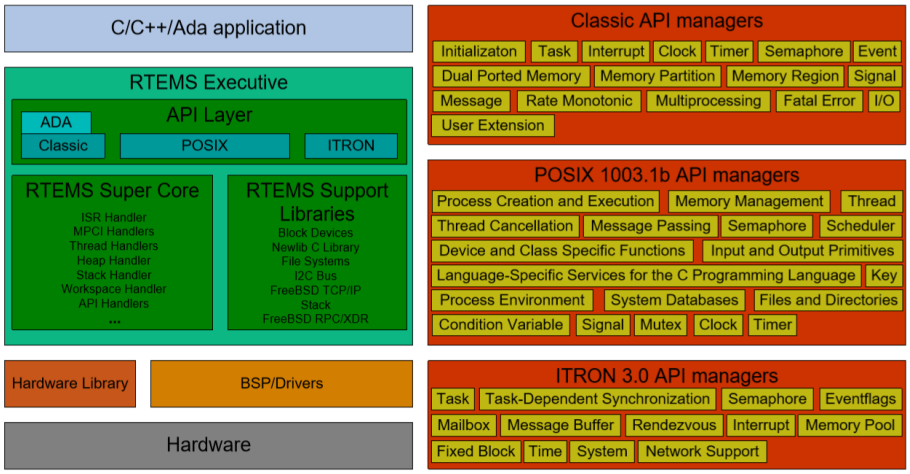

Figura 2.8: Ilustração do conceito da arquitetura do RTEMS.

2.3.5 L4Re

O sistema L4Re é baseado num sistema de microkernel / microhypervisor com uma pequena parte de computação privilegiada e que tem que consolidar múltiplas aplicações com múltiplos requisitos [8]. A Figura 2.9 mostra uma visão global de um sistema L4Re.

O microkernel Fiasco.OC usa uma interface baseada num conceito: *capabilities* de objetos. Este conceito pode ser entendido como chamadas de funções de objetos em programação orientada a objetos. Os serviços do kernel são implementados em objetos do kernel [4], e as tarefas guardam referências para esses objetos, estas referências são guardadas numa tabela com espaço reservado que é gerido e protegido pelo kernel. São, portanto, estas referências a que o conceito se refere como *capabilities*. Através deste sistema, se uma tarefa detém uma *capability* pode conceder a outra tarefa os mesmos direitos sobre esse objeto transferindo essa *capability* da sua tabela para a tabela da outra. Este microkernel é no fundo a componente de kernel do sistema L4Re. Nesta componente apenas são implementados os componentes que necessitam de privilégios e que têm

que funcionar em modo de kernel: espaços de endereçamento, threads e comunicação interprocessos. Todos os outros componentes de sistema operativo são implementados em modo de utilizador como aplicações. O microkernel suporta processadores ARM e x86, processadores multicore e multi processamento SMP.

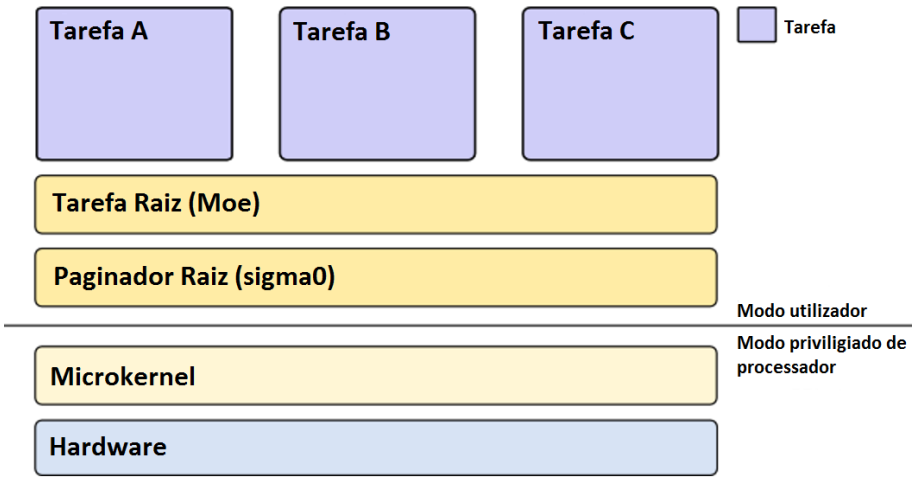

Figura 2.9: Estrutura global de um sistema baseado no L4Re.

O ambiente de execução do L4Re contempla um conjunto de serviços, APIs e bibliotecas para desenvolver aplicações nativas L4Re. Estas aplicações não depedem, por isso, de outros sistemas operativos, ou seja, não é necessário ter máquinas virtuais ou sistemas operativos a executar noutro compartimento. Assim é possível ter uma base de computação privilegiada (TCB – Trusted Computing Base) muito reduzida. Em termos funcionais o sistema está estruturado da seguinte forma: microkernel, ambiente de execução e aplicações. O microkernel disponibiliza primitivas para executar tarefas, para poder existir isolamento entre tarefas e para ter comunicação segura. Como já foi dito o kernel é a parte mais privilegiada do sistema, deve por isto ser o mais pequeno possível para reduzir a superfície de ataque. Apesar do kernel fornecer um conjunto de interfaces, estes podem não ser adequados para o desenvolvimento aplicacional, para isso o ambiente de execução do L4Re contém componentes de baixo nível que interagem diretamente com o micro-kernel. O paginador da raiz (sigma0) e a tarefa raiz (Moe) são os componentes mais básicos do ambiente de execução. Os outros serviços fornecem as suas próprias interfaces (e.g. enumeração de periféricos). As aplicações executam por cima do sistema e usam os serviços fornecidos pelo ambiente de execução, ou por outras aplicações.

2.3.6 VxWorks

O VxWorks é o sistema operativo de tempo-real comercial mais utilizado na industria dos sistemas embebidos [7]. É desenvolvido pela WindRiver com o objetivo de ser um sistema operativo com capacidade de mudança de contexto determinística, rápida e eficiente. O

seu microkernel suporta políticas de escalonamento preemptivas e *round-robin*, e permite ter um número ilimitado de tarefas. O VxWorks disponibiliza também um vasto conjunto de bibliotecas e ferramentas que permitem reduzir significativamente o desenvolvimento aplicacional.

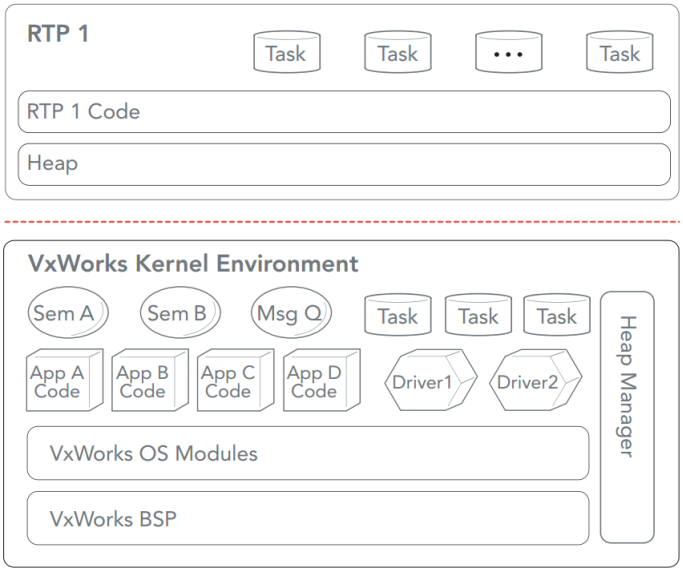

Figura 2.10: Ambiente do kernel do VxWorks.

Os processos de tempo-real do VxWorks [15] são em muitos aspetos semelhantes aos de outros sistemas operativos (e.g. Unix e Linux) incluindo compatibilidade com POSIX. O modelo de processo do VxWorks é desenhado para ser utilizado em sistemas de tempo-real e com processos VxWorks. As características que distinguem os processos VxWorks são:

- Ocupação de blocos contínuos de memória.
- A instanciação do processo está separado do carregamento da aplicação.
- Carregar as aplicações por inteiro (não existe carregamento de páginas *on demand*).
- Manter o modelo de escalonamento de tarefas do VxWorks, onde todas as tarefas são escalonadas globalmente (os processos em si não são escalonados).
- Inclusão da API do VxWorks para além das APIs POSIX.

Estas diferenças permitem ter determinismo e fornecer um modelo comum entre sistemas que usam uma unidade de gestão de memória (MMU - Memory mangement unit) e sistemas que não usam, isto faz com que o Vxworks seja especialmente adequado para sistemas de tempo-real estrito. Estes processos permitem executar aplicações em modo utilizador e cada processo tem o seu espaço de endereçamento. Esse espaço contém: o

programa propriamente dito, os dados do programa, *stacks* para cada tarefa, *heap* e recursos de gestão do processo. Podem existir vários processos em memória ao mesmo tempo e cada um destes processos pode ter mais que uma tarefa (*thread*). As regiões da memória virtual criadas são especificas de cada processo o que significa que não existe sobreposição destes na memória virtual. Este tipo de mapeamento da memória virtual permite ter um velocidade rápida num modelo de programação com ou sem MMU. As aplicações são criadas como ficheiros executáveis únicos independentemente do sistema operativo com o código de utilizador ligado às bibliotecas de aplicação da API do VxWorks.

Como o sistema operativo VxWorks é configurado e construído independentemente de aplicações que tenha de executar (Figura 2.10), este só precisa de ser configurado com os componentes de tempo-real que dão suporte aos processos e outros precisos pelas aplicações (e.g. filas de espera para mensagens).

2.3.7 QNX

O QNX é um sistema operativo que fornece um ambiente de tempo-real distribuído de multiprocessador e em rede completo com alto desempenho [6].

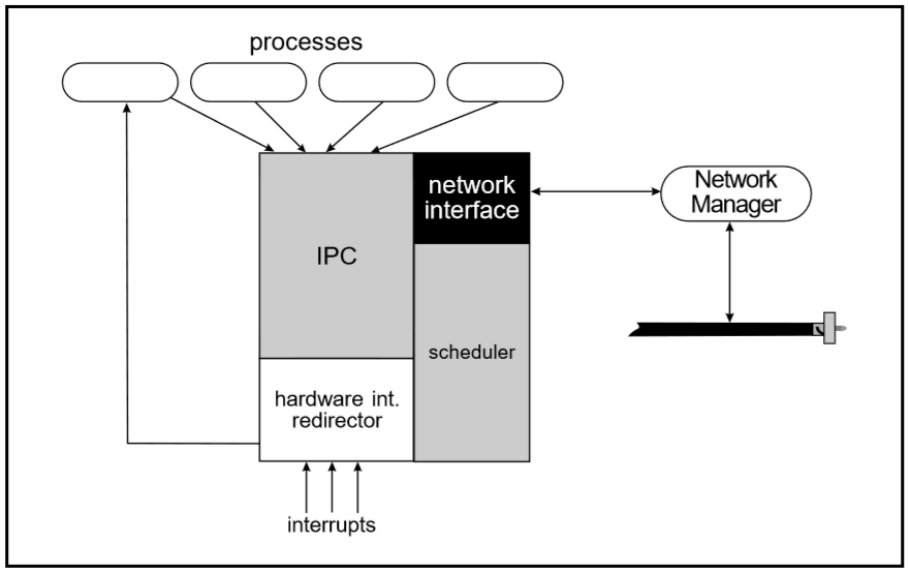

Figura 2.11: Microkernel do QNX.

A arquitetura usada para implementar esse ambiente é uma arquitetura com um microkernel de tempo-real (Figura 2.11), englobado por um conjunto de processos opcionais que fornecem serviços compatíveis com sistemas Unix e POSIX.

Este sistema é por isso adequado a sistemas de tempo-real estrito e de missão-crítica. O microkernel do QNX implementa apenas quatro serviços: comunicação entre processos, comunicação de rede de baixo nível, escalonamento de processos e tratamento de

interrupções. Para estes serviços existem 14 chamadas de kernel associadas. No total, estas funções ocupam 7K de código e fornecem funcionalidades e desempenho de tempo-real. Dado o reduzido tamanho do kernel, qualquer processador que tenha uma quantidade razoável de *cache* interna consegue oferecer um excelente desempenho para aplicações que requeiram muitos serviços de microkernel.

Como conclusão apresenta-se uma tabela onde figuram os sistemas operativos referidos e algumas características relevantes; esta comparação serve para representar de uma forma mais direta as diferenças, entre estes sistemas operativos, que são relevantes para este trabalho.

S.O.	Complexidade	Custo	API	Isolamento
Xenomai x86	alta	baixo	POSIX	Espacial
Raspbian (PREEMPT_RT)	alta	baixo	POSIX	Espacial
AIR	média	baixo	APEX	Temporal e Espacial
RTEMS	alta	baixo	POSIX	Espacial
L4Re	baixa	baixo	Base	Temporal e Espacial
VxWorks	baixa	elevado	VxWorks e POSIX	Temporal e Espacial
QNX	baixa	médio	POSIX	Temporal e Espacial

Capítulo 3

Arquitetura

Neste capítulo, apresenta-se a descrição da arquitetura usada no trabalho e um detalhe dos vários componentes usados na mesma.

3.1 Visão global da arquitetura

A arquitetura do sistema criado neste trabalho (Figura 3.1) divide-se em três camadas: o Safety Kernel, o sistema operativo Raspbian e a plataforma RaspberryPi.

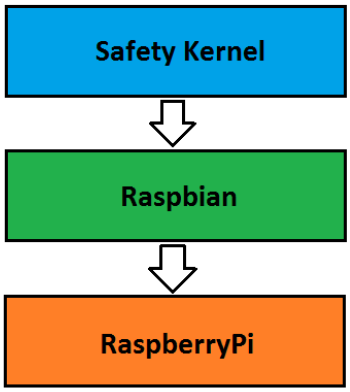

Figura 3.1: Arquitetura do sistema.

Nas próximas secções faz-se uma descrição dos processos que levaram à integração dos componentes referidos e das características destes que são relevantes para o trabalho.

3.2 RaspberryPi

A plataforma que será usada neste trabalho é o RaspeberryPi Model B Revision 2.0. Esta plataforma é de baixo custo, pelo que vai de encontro ao objetivo deste trabalho de testar

as capacidades de um sistema de tempo-real em plataformas que não exijam grande financiamento. Esta plataforma foi escolhida também por algumas das características do seu hardware, nomeadamente em termos da memória RAM (512Mbs) e processador (ARM11 de 700Mhz). Estes componentes permitem ter um desempenho bom para as necessidades do sistema global. Finalmente, a decisão de usar esta plataforma teve em conta o facto de ser compatível com o sistema operativo escolhido (Página 16).

3.3 RaspbianOS com PREEMPT_RT

Como já referido o PREEMPT_RT é um *patch* para o sistema operativo Linux. Este *patch* tem como principais objetivos minimizar tanto a quantidade de código do kernel que não pode ser preemptado, como a quantidade de código que tem que ser alterado para acrescentar essa capacidade extra de preempção [10]. Para isso são acrescentadas as seguintes funcionalidades:

- Secções críticas preemptáveis;
- Rotinas de atendimento de interrupções preemptáveis;
- Sequências de código de desativação de interrupções preemptável;
- Herança de prioridade para *spinlocks* e semáforos internos do kernel;
- Atraso de operações;
- Medidas para reduzir latências.

Secções críticas preemptáveis

Com o PREEMPT_RT, os *spinlocks* normais (spinlock_t e rwlock_t) e secções críticas de leitura "read-copy-update"(rcu_read_lock() e rcu_read_unlock()) são preemptáveis. As secções críticas de semáforos também são preemptáveis, mas isto já acontece nos kernels de PREEMPT e não PREEMPT, o que quer dizer, que se pode bloquear quando se está a obter um *spinlock*. Por outro lado, quer dizer que é ilegal obter um *spinlock* com preempção ou interrupções desativados. Para obter um bloqueio quando se tem uma situação de preempção ou interrupções desativados, chama-se o spin_lock() dentro do raw_spinlock_t em vez de usar spinlock_t. Isto porque o PREEMPT_RT tem um conjunto de macros que faz com que o spin_lock(), quando usado dentro de raw_spinlock_t, se comporte como um *spinlock* tradicional, enquanto que se for invocado dentro do spinlock_t a sua secção crítica pode ser preemptada. No entanto, não se deve abusar do uso destes *raw locks*, pois não deviam ser necessários fora zonas de baixo nível, como o escalonador, código específico de arquitetura e RCU(read-copy-update).

Desta forma, ao ter secções críticas a serem preemptadas, já não é possível ter a certeza de que uma destas secções será executada num dado CPU, ou seja, em multi CPU é possível que a secção ao ser preemptada possa passar para outro CPU. Portanto, ao usar variáveis específicas por CPU a possibilidade de preempção tem que ser tratada individualmente, já que não se tem este tratamento a ser feito por spinlock_t e rwlock_t. Algumas formas de fazer este tratamento são: explicitamente desativar a preempção (e.g.,get_cpu_var(), preempt_disable(), desativar interrupções de hardware) ou utilizar um bloqueio por CPU para salvaguardar as variáveis por CPU.

Rotinas de atendimento de interrupções preemptáveis

Quase todas as rotinas de atendimento de interrupções correm em contexto de processo no ambiente do PREEMPT_RT. Apesar de qualquer interrupção poder ser marcada com SA_NODELAY para que mude para contexto de interrupções, apenas as fpu_irq, irq0, irq2 (IRQ - pedido de interrupção), e lpptest têm SA_NODELAY especificado, e normalmente apenas a irq0 (interrupção do temporizador por CPU) é usada. Casos como o add_timer() e instruções da mesma família correm em contexto de processos, pelo que são perfeitamente preemptáveis.

O SA_NODELAY pode causar uma degradação grave nas latências de interrupções e de escalonamento, pelo que não deve ser utilizada sem muito cuidado.

Outro pormenor é a forma de tratar as interrupções de temporizadores por CPU (e.g., scheduler_tick()), já que estes funcionam em contexto de interrupções de hardware. Nestes casos, todos os bloqueios partilhados com o código de contexto de processos devem ser usados com *raw spinlocks* (exemplos referidos acima), mais, quando adquiridos do contexto de processos, deveram ser usadas as variantes _irq (e.g.,spin_lock_irqsave()). Por último, as interrupções de hardware, tipicamente, deverão estar desativadas quando código de contexto de processos acede a variáveis por CPU, que são partilhadas com a rotina de atendimento de interrupção SA_NODELAY, mais na secção seguinte.

Sequências de código de desativação de interrupções preemptável

O conceito de código para desativar interrupções pode parecer contraditório, mas é importante não esquecer a filosofia do PREEMPT_RT, esta filosofia baseia-se nas capacidades que o SMP(Multiprocessamento simétrico) do kernel do Linux tem para lidar com corridas de rotinas de atendimento de interrupções, tendo em atenção que a maior parte destas corre em contexto de processos. Qualquer código que lide com uma rotina destas, tem que ter em atenção o facto de essa rotina estar a correr concorrentemente noutro CPU. Assim, instruções como o spin_lock_t não precisam de desativar preempção, isto porque mesmo que a rotina de atendimento de interrupção corra e preempte o código que tem o spinlock_t, aquela bloqueia assim que tenta adquirir o spinlock_t, preservando assim a zona crítica.

No entanto, o local_irq_save() ainda desativa preempção, já que não existe um bloqueio correspondente para fazer controlo da zona crítica. Neste caso, o uso de bloqueios em vez do local_irq_save() ajuda bastante a baixar latências de escalonamento, em contrapartida perde-se a eficiência ao nível de SMP. Ainda relativamente ao local_irq_save(), este não desativa interrupções de hardware pelo que todo o código que lide com SA_NODELAY não o pode usar. Em vez disso, deve ser usado o raw_local_irq_save().

Herança de prioridade para *spinlocks* e semáforos internos do kernel

Uma das preocupações quando se programa em tempo-real são as situações de deadlock derivados de inversão de prioridades, isto é:

- Um processo A (baixa prioridade) está em execução e adquire o recurso X, por exemplo um bloqueio.
- Outro processo B (média prioridade) começa a executar e preempta A.
- Um processo C (alta prioridade) tenta adquirir o recurso X, mas como este está a ser usado por A, que por sua vez foi preemptado por B, C não pode continuar a execução.

Para resolver este tipo de situações existem duas formas principais, ou se tem preempção desativada ou se tem herança de prioridade. Na primeira situação é fácil de perceber porque se resolve o problema, como não existe preempção o processo B tem que esperar que A acabe antes de começar a executar, pelo que C terá o recurso X livre quando precisar do mesmo. Este caso é utilizado por kernels PREEMPT em relação a *spinlocks*. No entanto, em casos em que se tenha grande carga de trabalho esta solução pode provocar um aumento nas latências o que não é desejável.

A herança de prioridade pode, então, ser usada em situações em que não se quer desativar a preempção. O princípio passa por uma tarefa de alta prioridade transitar temporariamente a sua prioridade a uma tarefa de mais baixa prioridade, para que esta possa libertar recursos necessários. Esta herança é transitiva, ou seja, se uma quarta tarefa precisasse de obter o recurso podia herdar a prioridade de C e seguidamente A herdava desta nova tarefa. Entenda-se que esta herança é temporalmente limitada, assim que o recurso necessário é libertado todas as tarefas envolvidas voltam a ter as suas prioridades iniciais. Estas transições também só acontecem se as tarefas de mais baixa prioridade já tiverem adquirido o bloqueio do recurso, ou seja, se no entretanto ainda não têm o recurso reservado, não existe necessidade de transferência de prioridade, já que as tarefas mais prioritárias podem simplesmente obtê-lo primeiro.

Apesar destes mecanismos, a herança de prioridades em bloqueios de escrita-para-leitor é especialmente problemática. A solução do PREEMPT_RT para este problema é

permitir que apenas uma tarefa de cada vez bloqueie para leitura um bloqueio de escritor-leitor ou de um semáforo. Uma desvantagem desta solução é que pode limitar escalabilidade.

Atraso de operações

Como agora o *spinlock* pode ficar adormecido não é permitido invocá-lo enquanto se tem preempção ou interrupções ativados. Em alguns casos, pode-se resolver este problema atrasando a operação até aqueles serem ativados outra vez, exemplos:

- put_task_struct_delayed() põe uma put_task_struct() numa fila para ser executada mais tarde quando já for permitido aceder à zona crítica.

- mmdrop_delayed() põe numa fila um mmdrop() para ser executado mais tarde, como no exemplo acima referido.

- TIF_NEED_RESCHED_DELAYED não faz reescalonamento, mas espera até o próximo preempt_check_resched_delayed(). Sendo que o objetivo é evitar preempções desnecessárias em casos em que uma tarefa de prioridade elevada está a ser acordada e não consegue prosseguir até a tarefa corrente largar o bloqueio. A solução é mudar o wake_up() que é imediatamente seguido do spin_unlock() para wake_up_process_sync(). Isto porque que o wake_up() iria preemptar a tarefa corrente causando um *deadlock* por estar à espera do bloqueio. A lógica é, se o processo que está a ser acordado iria preemptar o que está a executar no momento, o acordar é atrasado através da flag TIF_NEED_RESCHED_DELAYED.

Em todos estes casos o objetivo é atrasar uma ação para que esta possa ser executada num altura mais segura.

Medidas para reduzir latências

Visto que um dos principais objetivos do PREEMPT_RT é baixar latências foram introduzidas algumas alterações para cumprir esse fim.

A primeira tem que ver com o hardware x86 MMX/SSE. Este hardware é tratado no kernel com preempção desligada, o que por vezes pode querer dizer que se vai esperar por instruções MMX/SSE. Em alguns casos, isto pode ser problemático já que algumas dessas instruções são bastante morosas. Assim, o PREEMPT_RT não usa de todo as instruções lentas.

A segunda alteração aplica variáveis por CPU ao alocador SLAB, isto como alternativa ao deliberado desativar de interrupções.

3.4 KARYON Safety Kernel

Um problema fundamental no KARYON é o seu funcionamento em sistemas cooperativos, ou seja, execução de funcionalidades de uma forma cooperativa. Assim, as implicações no padrão KARYON, e mais especificamente na definição do Safety Kernel, obrigam a que as regras de segurança sejam aplicáveis num sistema distribuído, que pode ter nós ligados através de uma rede possivelmente pouco fiável. Um sistema cooperativo é, então, composto por vários nós KARYON e cada nó tem um Safety Kernel, implicando que no sistema de veículos cada um tem sempre pelo menos um Safety Kernel.

O Safety Kernel é uma parte do padrão arquitetural que é o KARYON. O primeiro é responsável por tratar da informação que diz respeito à integridade de dados sensoriais e da informação relativa a restrições temporais, de alguns componentes do sistema, com o objetivo de determinar o melhor modo operacional possível para que as diferentes funcionalidades cooperativas possam ser executadas de forma segura.

De seguida, e de modo a fornecer algum contexto para o tipo de ambiente em que o Safety Kernel tem que operar, mostra-se um exemplo de um sistema KARYON.

Uma dada funcionalidade cooperativa é efetuada por um grupo de veículos cooperativos, e a funcionalidade é separada num número de funcionalidades definido pelos componentes funcionais. Cada uma destas funcionalidades pode ter modos de implementação diferentes: *i)* um componente com um modo de operação singular, *ii)* um componente que pode ser parametrizável e pode ter vários modos de operação, ou *iii)* com vários componentes independentes, redundantes e com níveis de desempenho diferentes, que dependem da implementação escolhida.

Os componentes podem ser de variados tipos(i.e., sensores, atuadores, de computação e de comunicação) dependendo do seu propósito. A informação gerada por estes componentes, mais especificamente pelos sensores, pode ter um atributo para níveis de validade. Dependendo dos diferentes modos de operação escolhidos para os componentes, a funcionalidade global será executada com diferentes níveis de serviço. Aqui mostra-se o principal objetivo do Safety Kernel, que é poder avaliar diferentes níveis de serviço para cada funcionalidade e selecionar as combinações que permitem garantir que os requisitos de segurança são respeitados. Alguns pontos que devem ser tidos em conta aquando da implementação do Safety Kernal são:

- Ter em conta componentes que, dada a sua complexidade, não será possível garantir o seu comportamento temporal;
- Ter em conta que a validade dos dados sensoriais pode não ser garantida durante a implementação e portanto pode extrapolar um limite para um dado nível de serviço;
- Ter em conta que vai determinar a melhor configuração e modo de operação que leva a um melhor desempenho, estando esta avaliação restrita pelos níveis de serviço de

cada funcionalidade definidos a priori;

Como cada funcionalidade integra um ou mais dos componentes já referidos, é atribuído um nível de serviço à mesma, que vai variar de acordo com os ditos componentes. Como é possível que alguns componentes possam não garantir que as restrições temporais sejam respeitadas é necessário haver outros, em paralelo, que o façam para poder ter um nível de serviço mínimo garantido.

Componentes do Safety Kernel

Para que a tarefa do safety kernel seja possível é necessário tratar de vários aspetos diferentes do funcionamento dos componentes [1]. Estas tarefas são tratadas no seu núcleo por partes distintas mas que podem comunicar entre si. A composição do safety kernel pode ser vista na figura 3.2.

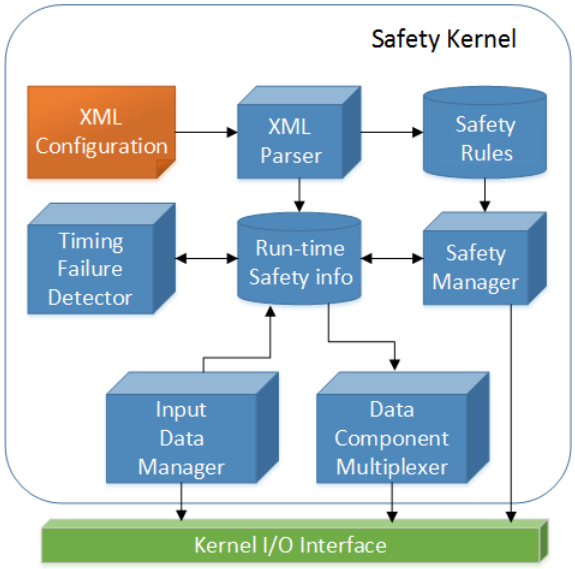

Figura 3.2: Composição do Safety Kernel

O primeiro passo no funcionamento do safety kernel é a definição das regras para os níveis de serviço. Estas são definidas num ficheiro XML a priori, offline e seguindo um padrão bem definido (Figura 3.3). Neste padrão são entendidos como unidades cada entrada ou saída de informação correspondente a um componente e valores que sejam calculados localmente. No arranque do safety kernel o módulo XML Parser usa este ficheiro para construir um repositório de regras de segurança e inicializa estruturas RSI (*Run-time Safety Information*). As regras de segurança são expressões de valor de verdade com combinações de valores estáticos e as definições das unidades, que estão diretamente relacionadas com os níveis de serviço.

A informação dos componentes externos é dirigida então ao IDM (*Input Data Manager*), fazendo este a atualização do RSI. Para a parte da análise dos requisitos temporais,

```
1 <?xml version="1.0"?>
2 <config>
3    <unit id="2">
4       <rule level="2" >
5          <test type="sup">
6             <validity id="0"/>
7             <value>70</value>
8          </test>
9       </rule>
10   </unit>
11   <unit id="3">
12      <from id="1" level="1"/>
13      <from id="2" level="2"/>
14   </unit>
15 </config>
```

Figura 3.3: Exemplo do ficheiro XML de configuração.

temos um TFD (*Timing Failure Detector*) que executa periodicamente dentro da ronda de execução do safety kernel. Quando é detetada uma falta temporal, o TFD guarda a informação na RSI correspondente. O DCM (*Data Component Multiplexer*) serve para escolher entre entradas de dados, ou seja, quando é necessário decidir sobre a informação recebida de dois componentes. Imagine-se uma linha de tempo-real, acima desta temos incerteza temporal, abaixo temos garantias temporais conservadas. O safety kernel está sempre abaixo dessa linha. Temos, então, um componente A abaixo dessa linha e outro B acima, o DCM serve para decidir entre A e B. Se B está dentro dos limites temporais estabelecidos, os seus dados podem ser usados, caso contrário são usados os de A. Por último, o SM (*Safety Manager*) é o módulo central do safety kernel já que é este faz a avaliação em tempo de execução das regras de segurança estabelecidas.

Para que todas estas interações sejam possíveis os dados enviados e recebidos têm que ser pensados a priori, isto é, os componentes têm que enviar dados num formato pré-estabelecido para que os valores analisados pelo safety kernel correspondam aos definidos no ficheiro XML. Entenda-se que os valores a enviar pelo componente, para o safety kernel, não têm que ser os valores analisados pelo componente (e.g., sensores), mas tem que ser definida uma correspondência. Um exemplo é uma distância de 100 metros pode corresponder a um valor unitário 50. Estas definições estão, por isso, dependentes apenas da implementação de cada sistema.

Capítulo 4

Implementação

Nesta secção serão explicados os processos de integração dos três componentes da arquitetura desenvolvida por estes trabalho.

4.1 Integração do Sistema Operativo com o RaspberryPi

O primeiro problema que se põe neste projeto é o sistema operativo a usar, sendo que não existe uma versão ainda compilada de um sistema operativo de tempo-real, com todas as funcionalidades de tempo-real garantidas. Uma boa solução é a versão Debian referida anteriormente. Esta versão tem também a particularidade de ter já incorporado o *patch* PREEMPT_RT, permitindo por isso testar as características de tempo-real dos sistemas em estudo.

A seguinte questão é, já com um sistema funcional na placa, como funciona o sistema de ficheiros deste equipamento e se isso trará algum impedimento para o funcionamento do sistema global. Isto porque não tem um disco rígido, mas em vez disso o sistema de ficheiros no RaspPi assenta sobre um cartão SD. Nesta questão existem já alguns cartões catalogados como funcionais e não funcionais para a plataforma [5].

Tendo já um sistema operativo integrado com a plataforma, o próximo passo é entender como se faz a integração com as capacidades de preempção do sistema, ou seja, como se atribui prioridades no sistema (Página 16), que tipo de prioridades estão à disposição e que tipo de políticas existem para o escalonador. Claro está, que também havia uma questão importante de após definir todos estes parâmetros se os processos estariam de facto a funcionar com prioridades, ou seja, se de facto se aplicava preempção por parte do escalonador.

4.2 Integração do Safety Kernel com o Raspbian

Daqui, segue-se que seria necessário perceber se o Safety Kernel teria alguma incompatibilidade com este sistema. Entenda-se, que sendo um sistema novo, e no qual o SK não

foi testado apenas se podiam fazer previsões (apesar de boas) para a integração global destes componentes.

A versão do Safety Kernel usada no trabalho é a 1.3.3. Para fazer os testes de integração com o sistema operativo foi modificado o XML de configuração do executável de teste pretendido, para que contivesse a definição do número de ciclos de *benchmark* [14], que neste caso foi definido a trezentos ciclos. O teste é então compilado através do *script* compile.sh, e em dois terminais separados executa-se o teste (e.g., ./example) e de seguida o kernel (i.e., ./kernel.sh). Os testes são então feitos enquanto ambos os processos decorrem. Durante a execução destes dois processos é também definida a prioridade dos mesmos, ou seja, ambos passam a ter prioridades de tempo-real. Para esta parte o comando usando foi: chrt -f -p 99 XXXX, o que representa que o processo com PID = XXXX irá executar com prioridade máxima, de tempo-real, e com política FIFO. Foi também criado um programa StressCPU.c para gerar carga no processador, para melhor evidenciar as capacidades de preempção do escalonador. Este processo corre com prioridade normal, ou seja, mais baixa relativamente aos descritos anteriormente. Assim, foram feitas várias rondas de teste, onde se faz variar a prioridade das tarefas em estudo e funcionalidades em execução (i.e., interface gráfica do sistema operativo e o programa StressCPU.c).

No *benchmarking* usado nestes exemplos, o Safety Kernel mede os tempos de execução de três componentes: TFD, SM e finalmente o DCM. As medições são feitas independentemente dos valores das unidades do XML (i.e., validade ou Nível de Serviço), todas as regras de todas as unidades são avaliadas já que o SM não para quando a avaliação de uma regra tem um valor de verdade. Assim tem-se uma boa aproximação a um cenário de pior caso possível. Os resultados obtidos destas medições são mostradas na forma de média e desvio padrão por ciclo de *benchmark*. Cada ciclo corresponde a um período do Safety Kernel, quer isto dizer que uma fase de testes de 100 ciclos durará 10 segundos, se o período for de 100ms. A média é útil, no caso de se ter um sistema grande com muitas regras, para que se possa ajustar o período do Safety Kernel da forma mais precisa possível. Por exemplo, se se tiver uma média de 12ms e o período for de 10ms, corre-se o risco de sobrecarregar o Safety Kernel e até mesmo perder sincronismo. Neste exemplo, seria melhor ter um período superior (20ms ou 30ms). O desvio padrão dá-nos uma ideia do quão estável é o nosso sistema, sendo que os tempos mais altos devem ser obtidos em sistemas que não possuem capacidades de tempo-real e sistemas como RTEMS devem ter valores mais baixos.

O sistema operativo é o Raspbian-Sep-2014-RT-B+ 2.3.2, relativamente ao seu funcionamento não houve necessidade de fazer alterações de maior. Para fazer a implementação do SK neste, a instalação é feita no sistema de ficheiros, e posteriormente foram feitas as alterações já referidas para que as propriedades de preempção estivessem ativadas quando assim fosse necessário, ou seja, foram criadas duas versões uma com preempção

e outra sem preempção para ter dois elementos de comparação nos testes de desempenho (Capítulo 5).

Finalmente, ao perceber que o sistema estava integrado corretamente e a funcionar com preempção ativada, chega o momento de perceber se existiam ganhos temporais face a outros testes feitos com o Safety Kernel (FPGA). Neste último, caso havia uma expectativa de ganhos temporais na ordem de 3 unidades de magnitude [2, Página 70].

Capítulo 5

Avaliação

Neste capítulo passamos à avaliação do sistema criado. Ou seja, é aqui que se mostra as capacidades do sistema de prioritizar tarefas, e se este tem as capacidades necessárias para gerir um sistema distribuído cooperativo de uma forma fiável e estável. Isto para perceber se se pode ter um sistema de tempo-real criado a partir de componentes baratos e genéricos, validando ou invalidando o objetivo principal deste trabalho.

5.1 Descrição dos testes

Numa primeira fase de testes o objetivo principal era provar a capacidade de preempção de tarefas, e portanto evidenciar as capacidades de tempo-real do *patch* PREEMPT_RT. Os dados obtidos servem assim, para que melhor se entenda as diferenças nas latências da execução das tarefas, que requerem restrições temporais com e sem preempção ativada. Nesta fase todas as prioridades foram atribuídas manualmente, pelo que as variações dos valores registados por vezes não representam o esperado. Servem estes exemplos também para demonstrar as diferenças no funcionamento do escalonador, tendo a atribuição de prioridades sido feita manualmente ou automaticamente. Para efeitos de leitura entenda-se, nos gráficos seguintes, as siglas S.P. e C.P. como sem prioridades e com prioridades, respetivamente. Os resultados mostrados são as médias de três corridas com o ficheiro de configuração XML definido com trezentos ciclos de *benchmarking*.

No segundo passo da fase de testes, em vez de usar o comando anteriormente referido (Página 33), passou-se a definir internamente ao processo a sua prioridade através da função sched_setschduler() (Página 16). Isto para mitigar os problemas observados anteriormente, e portanto poder obter os melhores resultados possíveis e o mais próximos de uma situação real. Assim, os processos têm as prioridades de tempo-real pré-definidas e podem preemptar outros processos assim que começam a executar. Deste modo, não existe interferência humana no fluxo da atribuição das prioridades, pelo que não há variação de latências dos processos, proveniente de factores externos.

Para a terceira e última fase de testes chega o momento de comparar as experiências

feitas anteriormente com o SK, nomeadamente no projeto KARYON [2, página 67-70], para perceber que mais valias o sistema implementado, por este trabalho, trás para o uso do SK e se as expectativas face aos tempos obtidos se comprova 35. É também importante olhar para estes testes no sentido de entender se as necessidades e características de tempo-real do SK se mantêm, ou seja, se se tem salvaguardado o papel de gestão de um sistema cooperativo distribuído.

5.2 Resultados da primeira fase de testes

Na figura 5.1 é possível ver a diferença de resultados com a interface gráfica do sistema operativo a funcionar. O objetivo de testar prioridades com as opções gráficas ativadas foi relevante para tentar perceber se haveria algum tipo de interferência das mesmas no funcionamento do escalonador. Claro que fazendo variar as prioridades dos processos em questão do SK se pode observar pequenas variações nas latências dos processos. Neste gráfico pode-se observar para cada componente do SK as variações nas médias e nos desvios padrão. Sendo que seria de esperar que ao ter prioridades ativadas os tempos de execução baixassem face aos tempos sem prioridades, pode-se também observar que nestes resultados é difícil perceber essa expectativa. Na verdade apenas nos tempos totais, e mesmo aí apenas no desvio padrão, é que se torna obvio o decréscimo dos tempos, sendo que a média é praticamente igual. Isto deve-se a algum *jitter* inserido pela interface gráfica e pelo atraso inerente da atribuição manual de prioridades, como referido acima.

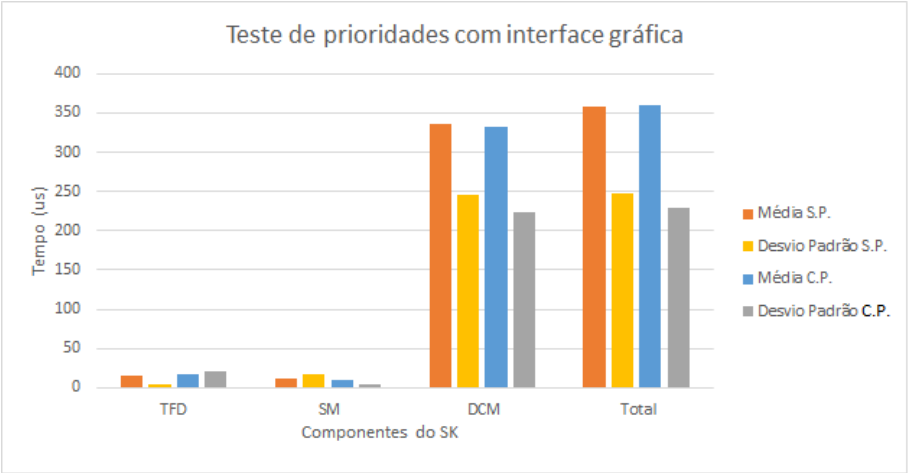

Figura 5.1: Gráfico de variação de tempos com interface gráfica ligada.

Usando agora o mesmo processo de teste mas com a interface gráfica desativada nota-se uma variação bastante contrária ao que seria de esperar. Embora, já não se verifique o *jitter* provocado pela interferência da interface gráfica, seria de esperar que os tempos dos processos sem prioridade fossem inferiores do que quando se ativa prioridades. É no entanto óbvio, pelos resultados (Figura 5.2), que tal não aconteceu. Vemos mesmo

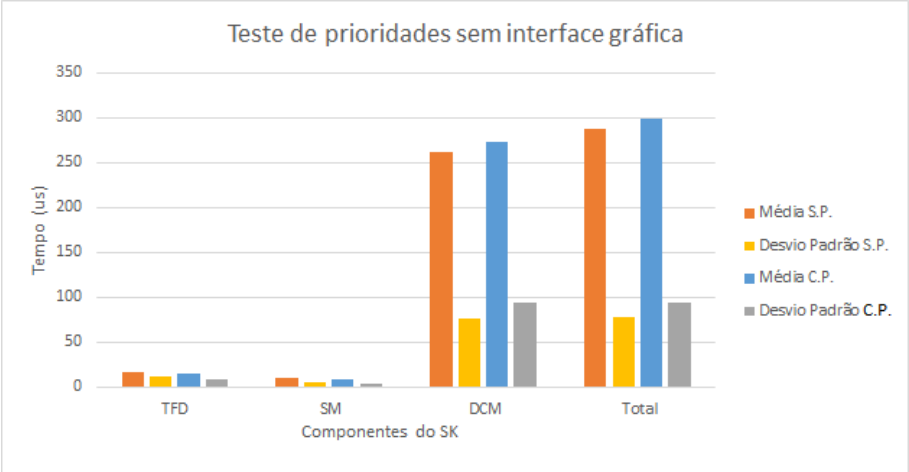

Figura 5.2: Gráfico de variação de tempos sem interface gráfica ligada.

que aconteceu o contrário, temos um aumento tanto nas médias como nos desvios padrão dos tempos analisados ao ativar prioridades. Isto leva a repensar se o sistema estaria de facto a funcionar como o esperado, ou se este sistema não servia para o propósito deste projeto. Como se demonstra mais à frente isto não é representativo da verdade, mas é no entanto, importante para demonstrar que mais uma vez que o facto das prioridades terem sido definidas manualmente, insere muita interferência para a análise dos dados, dando origem a variações altas nos tempos gerados.

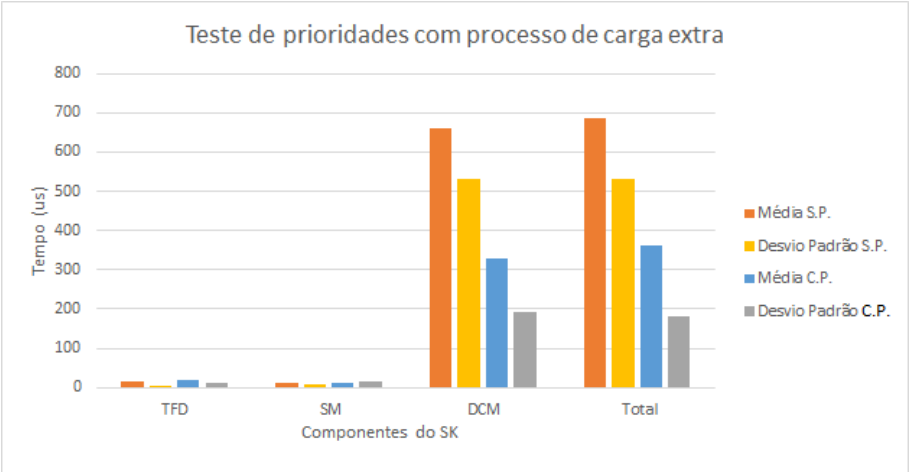

Figura 5.3: Gráfico de variação de tempos com processo StressCPU para gerar carga extra.

No caso dos tempos representados na Figura 5.3, os testes feitos são semelhantes aos anteriores mas desta vez tem-se um processo a correr constantemente para gerar carga no CPU, ou seja, este processo foi criado para forçar o escalonador a fazer constantemente chamadas ao processador. O raciocínio, neste caso, foi no sentido de perceber definitivamente se a atribuição das prioridades funciona da forma esperada ao ter um

sistema operativo com capacidades de tempo-real. Assim, ao observar os tempos analisados durante a execução das tarefas do SK, percebemos um decréscimo acentuado ao ativar prioridades para as mesmas. Note-se que também neste caso as prioridades foram atribuídas manualmente, mas o facto de ter um processo a gerar carga extra no CPU, foi suficiente para que o funcionamento de gestão de prioridades se torne óbvio. Temos por isto um aumento significativo dos tempos das tarefas quando as mesmas não têm prioridades de tempo-real definidas, principalmente quando comparados com os tempos após definir prioridades. Obtemos, assim, um sinal de que o escalonador faz de facto gestão de prioridades e preempção de tarefas.

5.3 Resultados da segunda fase de testes

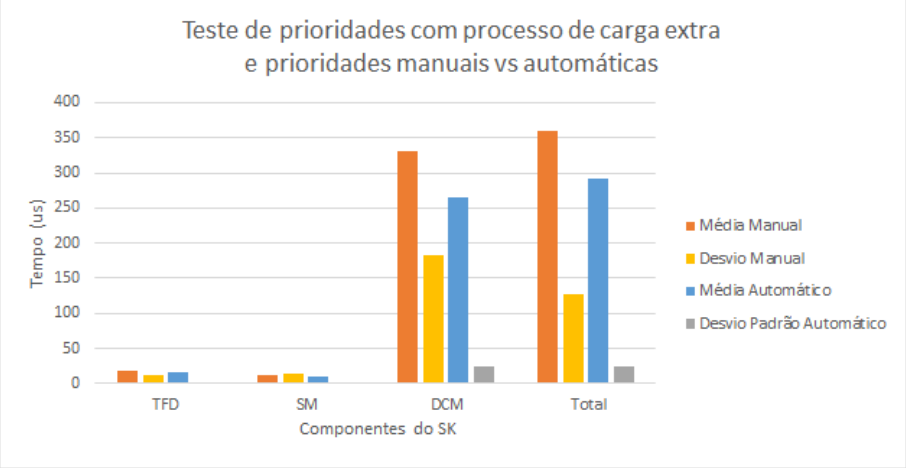

Figura 5.4: Gráfico de variação de tempos com processo StressCPU para gerar carga extra, e prioridades inseridas manualmente e automaticamente.

Como primeiro exemplo das diferenças de como as prioridades podem ser atribuídas no sistema operativo em estudo, temos os resultados que se pode observar em 5.4. Estes resultados demonstram de uma forma clara o impacto que pode ter a gestão manual das prioridades das tarefas. Como se demonstra pelos resultados obtidos, existe um decréscimo bastante acentuado nos tempos obtidos quando se utiliza uma forma automática e a priori de atribuição de prioridades. Estes resultados são importantes pois evidenciam as capacidades de preempção do sistema mas também as diferenças na sua utilização.

O gráfico 5.5 mostra os resultados obtidos ao atribuir prioridades aos processos do SK de uma forma automatizada, com e sem o processo StressCPU a executar. Neste exemplo, temos que as médias e desvios padrão sem StressCPU e com StressCPU representados por S.S.CPU e C.S.CPU, respetivamente. Deste modo é possível perceber que tipo de variação se tem ao exercer carga extra no CPU e que peso isso tem sobre a execução do

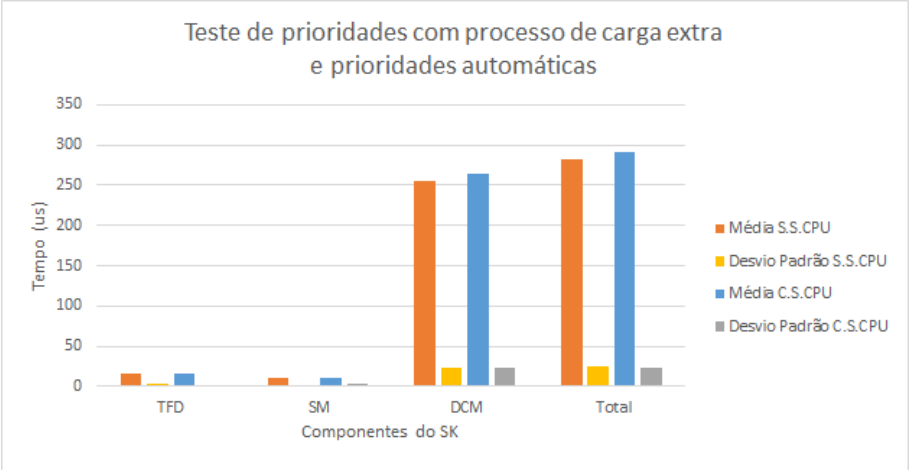

Figura 5.5: Gráfico de variação de tempos com e sem processo StressCPU para gerar carga extra, e prioridades automáticas.

SK. Como se pode ver através dos resultados, não existem diferenças de maior, o que à partida pode parecer estranho, mas na realidade é precisamente isso que se pretende obter. Quer isto dizer, que o facto de não existir uma diferença significativa nos tempos, tendo o processo StressCPU a correr ou não, que o mesmo é preemptado pelos processos do SK, ou seja, não interessa se existe carga extra para o escalonador, se uma tarefa tem prioridade superior ao querer executar preempta qualquer outra com uma prioridade inferior. Estes resultados vêm então comprovar o comportamento esperado de um sistema operativo com capacidades de tempo-real, e por isso do Raspbian com o *patch* PREEMPT_RT.

5.4 Resultados da terceira fase de testes

As configurações usadas para esta fase do *benchmarking* servem para entender que impacto o número de unidades ou número de nós têm no tempo de execução do Safety Kernel. Os tempos avaliados são separados em duas caraterísticas principais: *i)* o tempo de *parse* do ficheiro XML e criação das regras; *ii)* os tempos de execução dos três principais componentes do SK (i.e., TFD, SM e DCM). O sistema de medição é intrusivo pois são usadas funções de *start/stop* embebidas no kernel, no entanto esta intrusão é considerada insignificante.

Tem-se então para cada uma destas medições três configurações onde se faz variar o número de unidades, regras e nós. Entenda-se que para cada configuração se tem uma unidade de *input* e N de saída, sendo que N varia de 1 a 10. Assim, têm-se como mostra a figura 5.6 os valores definidos para cada configuração. Em termos de ficheiros entenda-se que cada N é uma corrida de *benchmark* e, portanto, tem um ficheiro XML por corrida.

De seguida mostra-se os tempos obtidos durante o *parse* de cada XML, estes tempos correspondem a criar todas as regras e guardá-las no repositório de regras de segurança

Configuração 1			
N	Unidades	Regras	Nós
1	2	1	3
2	2	4	12
3	2	9	27
4	2	16	48
5	2	25	75
6	2	36	108
7	2	49	147
8	2	64	192
9	2	81	243
10	2	100	300

Configuração 2			
N	Unidades	Regras	Nós
1	2	1	3
2	3	4	12
3	4	9	27
4	5	16	48
5	6	25	75
6	7	36	108
7	8	49	147
8	9	64	192
9	10	81	243
10	11	100	300

Configuração 3			
N	Unidades	Regras	Nós
1	2	1	3
2	5	4	12
3	10	9	27
4	17	16	48
5	26	25	75
6	37	36	108
7	50	49	147
8	65	64	192
9	82	81	243
10	101	100	300

Figura 5.6: Configurações usadas para os testes de comparação com os resultados obtidos no projeto KARYON.

(tempos de acesso a ficheiros não foram considerados).

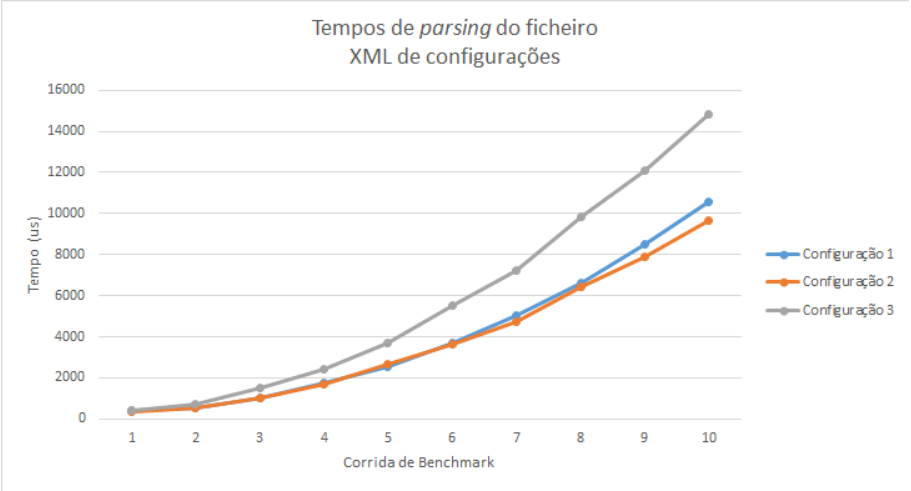

Figura 5.7: Configurações usadas para os testes de comparação com os resultados obtidos no projeto KARYON.

Ao estudar os tempos obtidos em 5.7 é óbvio que comparativamente aos valores temporais registados ao usar uma FPGA (*soft cpu*) se tem valores muito mais baixos, o que responde à questão: se o sistema traria mais valias ao uso do Safety Kernel. No entanto, o que se queria demonstrar não era só se se mantinham as características de tempo-real do sistema, mas também se as expectativas no decréscimo dos valores estariam representadas nos resultados (ganho de 2 ou 3 ordens de magnitude). Isto verifica-se, já que nos resultados obtidos para o projeto KARYON se obtiveram valores na ordem dos segundos (o valor mais alto foi para a configuração três, 1,5 segundos) e no caso do RaspberryPi tem-se valores que não ultrapassam as dezenas de milisegundos, mais especificamente 15 milisegundos.

Como referido acima, os próximos resultados servem para avaliar os tempos de execução dos três principais componentes do SK. Assim, tem-se que os valores registados representam as médias de 100 ciclos de *benchmark* para cada um dos componentes. Mais uma vez é considerado o pior caso possível, ou seja, cada regra de cada unidade é avaliada.

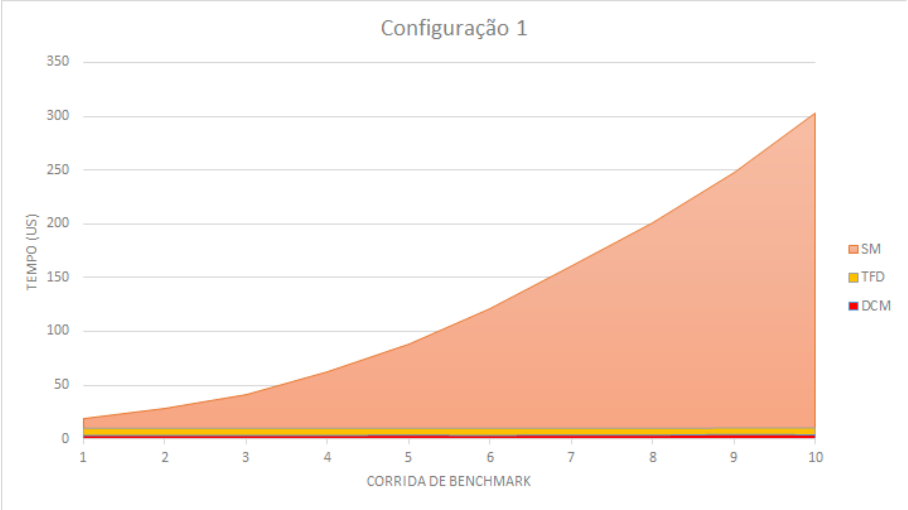

Figura 5.8: Tempos de execução para a configuração 1.

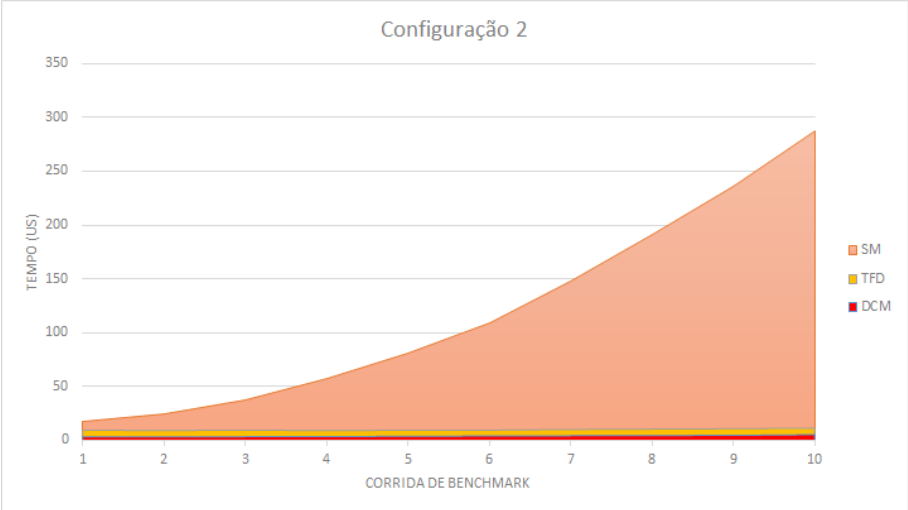

Figura 5.9: Tempos de execução para a configuração 2.

Os valores temporais registados nas três figuras acima mostram mais uma vez um ganho significativo face aos obtidos para o projeto KARYON, sendo que no pior caso (i.e., configuração três) se tem valores máximos na ordem das centenas de microsegundo para o SM, usando a plataforma deste trabalho. No caso da FPGA, os valores obtidos foram na ordem das unidades de milisegundo. É de referir que os valores que mais variam são os do SM, porque este é responsável por avaliar as regras de segurança, e esta é a tarefa mais complexa do SK. Assim os valores do TFD e do DCM são praticamente insignificantes face aos do SM, já que só dependem do número de unidades. Este teste também vem comprovar esse facto, isto é, o número de unidades tem pouco impacto no tempo de execução do SK. Estes valores não correspondem, no entanto, ao ganho esperado, já que só se ganha uma ordem de magnitude (8 milisegundos para 0,367 milisegundos). Ainda assim o ganho é significativo para as operações em estudo.

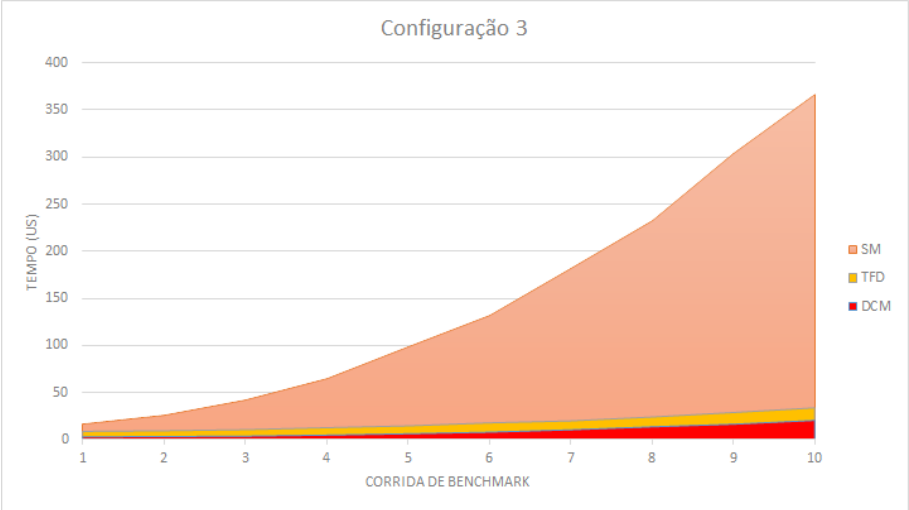

Figura 5.10: Tempos de execução para a configuração 3.

Capítulo 6

Conclusão

Conclui-se muito realisticamente que os resultados obtidos servem para demonstrar que é possível implementar numa plataforma de baixo custo um sistema com garantias de tempo-real aceitáveis, dados os requisitos do Safety Kernel. Existe uma melhoria significativa face a resultados obtidos anteriormente para o desempenho do Safety Kernel, mostrando que a plataforma RaspberryPi juntamente com o sistema operativo Raspbian, com o *patch* PREEMPT_RT, conseguem responder a necessidades mais exigentes e restritas. As restrições impostas pelo SK são asseguradas juntamente com as suas necessidades de prioritização de tarefas. Mais, verificou-se com este trabalho que se pode construir um sistema de gestão deste tipo de forma simples e rápida, apesar de neste caso terem existido alguns contratempos, por ser um sistema desconhecido e ser necessário fazer várias validações a priori. Deste modo, o objetivo principal do trabalho, que um sistema genérico e barato pode responder a restrições temporais exigentes, fica aqui demonstrado.

A gestão do sistema não deve, no entanto, ser feita de forma manual, como se demonstrou através dos resultados obtidos, ou seja, este ponto deve ser tido em conta aquando da implementação do sistema. A gestão de prioridades pelo escalonador é automática, mas os processos que precisam de poder preemptar o resto do sistema devem ter representado no código essas funcionalidades (usando os mecanismos descritos em 2.3.2).

Futuramente seria interessante ver um implementação deste sistema feita com um RTEMS.

Bibliografia

[1] A. Casimiro and E. Vial. Evaluation of safety rules in a safety kernel-based architecture. Technical report, 2014.

[2] A. Casimiro and E. Vial. Safety kernel definition (public version). Technical report, September 2014.

[3] DIE.net. Linux man page - sched setscheduler. `http://lwn.net/Articles/146861/`.

[4] TUD Technische Universitat Dresden. L4re - l4 runtime environment. `https://os.inf.tu-dresden.de/L4Re/doc/`.

[5] Elinux. Rpi sd cards. `http://elinux.org/RPi_SD_cards`.

[6] Dan Hildebrand. An architectural overview of qnx.

[7] Benjamin Ip. Performance analysis of vxworks and rtlinux.

[8] Kernkonzept. L4re technology. `http://www.kernkonzept.com/l4re.html`.

[9] H. Kopetz. *Real-Time Systems*. Kluwer Academic Publishers, 1997.

[10] LWN.net. A realtime preemption overview. `http://lwn.net/Articles/146861/`.

[11] J. Rufino, J. P. Craveiro, T. Schoofs, C. Tatibana, and J. Windsor. Air technology: a step towards arinc 653 in space. In *Proceedings of the Eurospace "Data Systems in Aerospace"Conference (DASIA 2009)*, Istanbul, Turkey, May 2009.

[12] J. Rufino, H. Silva, A. Constantino, D. Freitas, M. Coutinho, S. Faustino, M. Mota, P. Colaço, J. Sousa, L. Dias, B. Damjanovic, and M. Zulianello. Rtems centre – support and maintenance centre to rtems operating system.

[13] P. Veríssimo and L. Rodrigues. *Distributed Systems for System Architects*. Kluwer Academic Publishers, 2001.

[14] E. Vial. Safety kernel implementation (configuration guide). Technical report, 2014.

[15] Inc. Wind River Systems. Vxworks - application programmer's guide. Technical report, October 2005.