Arcabouço para o Escalonamento de Processos em Tempo Real para Linux Embarcado

José Luís do Nascimento

Dissertação de Mestrado submetida à Coordenadoria do Programa de Pós-Graduação em Engenharia Elétrica da Universidade Federal de Campina Grande - Campus de Campina Grande como parte dos requisitos necessários para a obtenção do grau de Mestre em Ciências no Domínio da Engenharia Elétrica.

Área de Concentração: Processamento da Informação

Angelo Perkusich, Dr.
Orientador

Campina Grande, Paraíba, Brasil

FICHA CATALOGRÁFICA ELABORADA PELA BIBLIOTECA CENTRAL DA UFCG

N244a
2008 Nascimento, José Luis do.
Arcabouço para o escalonamento de processos em tempo real / José Luis do Nascimento — Campina Grande, 2008.
61 f. : il.

Dissertação (Mestrado em Engenharia Elétrica) – Universidade Federa de Campina Grande, Centro de Engenharia Elétrica e Informática.
Referências.
Orientador: Prof. Dr. Angelo Perkusich.

1. Linux Embarcado. 2. Sistemas em Tempo real. 3. Integração de Soluções. 4. Controle de Processos. 5. Realimentação. I. Título.

CDU – 621.3:004.451.9(043)

ARCABOUÇO PARA O ESCALONAMENTO DE PROCESSOS EM TEMPO REAL PARA LINUX EMBARCADO

JOSÉ LUIS DO NASCIMENTO

Dissertação Aprovada em 29.09.2008

ANGELO PERKUSICH, D.Sc., UFCG
Orientador

ANTONIO MARCUS NOGUEIRA LIMA, Dr., UFCG
Componente da Banca

HYGGO OLIVEIRA DE ALMEIDA, D.Sc., UFCG
Componente da Banca

CAMPINA GRANDE - PB
SETEMBRO - 2008

Dedicatória

Dedico esta dissertação a minha família, em especial a minha mãe, e a todos que me apoiaram e incentivaram nesta jornada.

Agradecimentos

Agradeço a Deus primeiramente.

Agradeço aos meus pais José Nivaldo e Maria do Socorro pelo apoio incondicional durante todas as fases de minha vida.

Agradeço aos meus irmãos Edmar, Sandra, Maria de Lourdes e Sérgio pelo companheirismo, empenho e dedicação na passagem pelas dificuldades durante a trajetória até aqui.

Agradeço ao meu amigo Marcos Morais e sua família por toda a atenção e amizade.

Agradeço aos meus amigos, colegas de sala Ádrian Guedes, André Felipe, Danilo Santos, Diego Bezerra, Marcos Fábio, Mário Hozano, Olympio Silva e Yuri Gomes pelo apoio e pelo companheirismo de sempre.

Agradeço ao meu orientador Angelo Perkusich pelo apoio e orientação dispensados no desenvolvimento do presente trabalho.

Por fim, agradeço a todos aqueles que de forma direta ou indireta tiveram participação no presente trabalho e em minha vida.

Resumo

A utilização do sistema operacional Linux em sistemas embarcados propicia a reutilização de uma grande quantidade de aplicativos. Os sistemas embarcados utilizados em sistemas industriais podem requerer o processamento de tarefas em tempo real. No Linux existem várias alternativas para prover a execução de tarefas em tempo real, mas nenhuma delas atende os requisitos necessários para ser incorporada oficialmente ao seu núcleo. A integração das soluções RT-Preempt e LITMUS^{RT} trata-se de uma alternativa interessante para a provisão de serviços em tempo real no Linux. A primeira trata de problemas como as longas seções críticas com interrupções desabilitadas e inversões de prioridades, enquanto que a segunda introduz algoritmos como o EDF e permite o uso de controladores proporcional integral para o escalonamento de processos em malha fechada. Desta forma, o objetivo deste trabalho é a integração das soluções RT-Preempt e do LITMUS^{RT} em um arcabouço que permita o desenvolvimento e execução de aplicações de controle e automação em uma plataforma embarcada típica.

Abstract

The utilization of Linux operating system in embedded platforms provides a large reuse of software applications. Embedded systems used in industrial systems may require the processing of tasks in real time. Actually, on Linux there are several alternatives to provide the implementation of tasks in real time, but none of them meets the requirements to be officially incorporated in its core. The integration of RT-Preempt and LITMUS^{RT} solutions is an interesting alternative for the provision of services in real time in the Linux system. The first solution deals with problems like long critical sections with disabled interruptions and priority inversions, while the second provides algorithms like EDF and allows the use of proportional integral controllers for process scheduling in closed loop. Thus, the objective of this work is the integration of the solutions RT-Preempt and LITMUS^{RT} in a framework that allows the development and implementation of control and automation applications in a typical embedded platform.

Índice

Lista de Tabelas

Lista de Figuras

Capítulo 1

Introdução

Nos dias atuais, o sistema operacional Linux tem se tornado um dos principais sistemas operacionais para sistemas embarcados [11, 23]. É notória a utilização do Linux em roteadores, pontos de acesso sem fio, sistemas de entretenimento, *PDAs*, computadores ultramóveis, Internet *Tabblets*, telefones celulares, terminais de acesso remoto, sistemas de monitoração de trânsito e robôs [17]. Fatores como a disponibilidade aberta e gratuita de aplicativos, uma grande comunidade de suporte, suporte as principais arquiteturas de hardware, grande disponibilidade de *drivers* e suporte aos diversos protocolos de rede contribuem para esta aceitação.

A despeito deste uso crescente, alguns fatores podem ser considerados como empecilhos a uma utilização mais abrangente do Linux no mercado de sistemas embarcados. Dentre estes fatores destacam-se: a falta de suporte por parte de alguns fabricantes, a falta de informação a cerca de *drivers*, a heterogeneidade da comunidade, uma grande quantidade de aplicativos de *software* com mesmo objetivo, mas com funcionalidades incompletas, suporte a execução nativa de tarefas em tempo real. Este último fator tem um impacto mais significativo em aplicações no contexto de sistemas embarcados para aplicações de tempo real, como os usados em plantas industriais com finalidades de monitoração e controle, e são o foco principal deste trabalho.

Neste contexto, várias soluções que oferecem suporte a tempo real usando o Linux podem ser utilizadas, entretanto nenhuma destas foi incorporada ao núcleo do sistema operacional. A solução mais próxima de ser incorporada, denominada de RT-Preempt [35], é mantida por Ingo Molnar que trabalha na Red Hat mantendo o núcleo do Linux. Dentre os seus trabalhos, os de maior impacto no desenvolvimento do Linux são os Escalonadores O(1) [26] e CFS [34].

Em um sistema em tempo real o problema principal refere-se ao escalonamento de tarefas; as tarefas de maior prioridade, ou seja, as tarefas mais críticas devem ser executadas antes das de menor prioridade dentro do prazo requerido. Alguns sistemas realizam

o escalonamento a priori, de maneira estática. Dentre os algoritmos estáticos o RM[1] é ótimo, embora possibilite utilizar no máximo 69,3% do tempo de processamento [25]. Nos sistemas em que o escalonamento é dinâmico pode-se conseguir uma utilização de CPU de 100% usando o algoritmo EDF[2]. Em ambos os casos o escalonamento é realizado em malha aberta, e uma sobrecarga de processos no sistema degrada de maneira acentuada o comportamento do escalonador e portanto resultando em tarefas que perdem o prazo. O uso de escalonadores com realimentação com base na teoria de controle tem se mostrado uma alternativa viável quando a distribuição e o custo computacional para execução das tarefas não pode ser prevista a priori, o que ocorre na maioria dos sistemas dinâmicos.

Na literatura, existem vários trabalhos que utilizam a teoria de controle para o escalonamento de recursos em sistemas computacionais.

John Stankovic junto com seu grupo de pesquisa desenvolveu uma série de formalizações para projeto e avaliação de algoritmos de controle com realimentação em sistemas em tempo real [29, 30, 43]. Em seus trabalhos foram propostos e analisados algoritmos como o FC-EDF[3] e o FC-RM[4] para escalonamento de processos em tempo real usando controladores Proporcionais Integrais Derivativos (PID) [29]. O uso da teoria de controle nos sistemas computacionais requer a definição dos controladores, de uma política de admissão de tarefas e um controlador de níveis de serviço. O objetivo do controle aplicado ao escalonamento de processos é minimizar a quantidade de tarefas em tempo real que perdem o prazo, dado que elas foram aceitas no sistema, maximizando o uso de CPU.

Recentemente, Amirijoo e outros desenvolveram um trabalho de avaliação experimental dos algoritmos de controle usando modelos de primeira e segunda ordem na modelagem de sistemas de bancos de dados [4]. Neste contexto, Årzén [18] desenvolveu um trabalho que utiliza controle linear quadrático para a otimização do uso de malhas de controle com freqüências de amostragem variáveis.

Outras técnicas podem ser usadas em conjunto com o controle com realimentação para que as tarefas em tempo real possam ser executadas dentro de seus prazos. Uma dessas técnicas que pode ser aplicada a arquiteturas com vários núcleos é a blindagem de CPU [9]. Ela permite que um processador possa ser reservado para executar as tarefas em tempo real, enquanto que outro execute tarefas de baixa prioridade.

---

[1] RM - *Rate Monotonic*

[2] EDF - *Earliest Deadline First*

[3] FC-EDF - *Feedback Control Earliest Deadline First*

[4] FC-RM - *Feedback Control Rate Monotonic*

1.1 Objetivos

O objetivo deste trabalho é desenvolver um ambiente operacional que possibilite a execução de tarefas em tempo real em uma plataforma de *hardware* embarcada típica e que possibilite o escalonamento de processos através de técnicas de controle com realimentação. Para que isto seja possível são considerados vários aspectos dos componentes do sistema operacional, de modo que a solução obtida seja aplicável a um sistema embarcado real. Como componentes da solução optou-se pelo uso de tecnologias de código aberto disponibilizadas de forma livre, em particular das tecnologias baseadas no sistema operacional Linux. Como parte integrante deste trabalho são apresentados os principais conceitos envolvidos na concepção de um ambiente operacional para desenvolvimento e utilização de aplicações embarcadas.

1.2 Organização da dissertação

Esta dissertação está organizada da seguinte forma:

- No Capítulo 2 são apresentadas as formas que um sistema embarcado pode assumir, bem como os conceitos de Linux embarcado;
- No Capítulo 3 são apresentadas as principais configurações de hardware usadas em projetos de sistemas embarcados com Linux;
- No Capítulo 4 são apresentados os conceitos envolvidos na a elaboração e configuração do ambiente de desenvolvimento típico;
- No Capítulo 5 são apresentados os passos para a escolha e configuração do núcleo do Linux;
- No Capítulo 6 são apresentados os principais *bootloaders* usados na inicialização de sistemas embarcados com Linux;
- No Capítulo 7 são apresentados os principais conceitos e variações de Linux em tempo real;
- No Capítulo 8 mostra-se o uso da teoria de controle aplicada aos sistemas em tempo real;
- No Capítulo 9 mostra-se a implementação da solução integrada para o escalonamento de processos em tempo real baseada na teoria de controle com realimentação;
- Finalmente, no Capítulo 10 são elencadas as considerações finais do presente trabalho.

Capítulo 2

Anatomia dos sistemas embarcados

A maioria dos computadores existentes atualmente está embarcada em dispositivos das mais variadas formas e são voltados para aplicações específicas. Esta ampla faixa de dispositivos com poderes de processamento diversos e voltados para aplicações específicas são classificados como sistemas embarcados. De acordo com este conceito, são exemplos de sistemas embarcados: tocadores de MP3, telefones portáteis, robôs, computadores de bordo, controladores industriais, roteadores, videogames, impressoras e receptores GPS, entre outros.

Em linhas gerais, os sistemas embarcados apresentam as seguintes características:

- São projetados para executar tarefas específicas. Alguns destes sistemas necessitam que estas tarefas sejam executadas em tempo real, já outros apresentam baixos requisitos de desempenho;
- Nem sempre são sistemas completos. Na maioria das vezes, os sistemas embarcados consistem de pequenas partes que servem a um sistema com propósitos maiores;
- Na maioria das vezes o software do sistema recebe o nome de *firmware*, e é armazenado em memórias ROM ou FLASH sem o uso de discos rígidos.

2.1 Tipos de sistemas embarcados

A partir da explanação inicial, nota-se a existência de vários tipos de sistemas embarcados. Uma classificação inicial destes sistemas pode ser realizada a partir de uma divisão de acordo com tamanho, restrições de tempo, necessidade de rede e interação com o usuário.

2.1.1 Tamanho

O tamanho dos sistemas embarcados pode variar segundo três aspectos principais: o tamanho físico, a capacidade de armazenamento de dados e a capacidade de processa-

mento. O fato de o sistema ser embarcado não está relacionado às suas dimensões físicas, e sim a sua habilidade de lidar com uma tarefa específica, deste modo podem existir sistemas embarcados do tamanho de uma caixa de fósforos ou de uma geladeira. Em relação à capacidade de armazenamento de dados, podem-se encontrar sistemas embarcados mínimos com 4 MB de memória RAM e 2 MB de ROM para armazenamento do *firmware*, sistemas médios com 32 MB de ROM e 64 MB de RAM, a partir dos 32 MB de RAM os sistemas podem ser classificados como de grande porte. Um dos fatores limitantes ao aumento da velocidade de processamento é o consumo de bateria, que limita a capacidade de processamento entre 300 e 400 MHz, nos dispositivos com maiores capacidades.

2.1.2 Restrições de tempo

De acordo com o tipo de aplicação do dispositivo, este pode necessitar operar em tempo real ou não. As aplicações que geralmente necessitam de operação em tempo real estão ligadas ao processamento multimídia e a execução de ações de monitoração e controle em plantas industriais. Neste contexto, as restrições de tempo real podem ser suaves ou estritas.

Nos sistemas embarcados com restrições de tempo suave a perda do prazo das tarefas não resulta em uma perda completa do valor da computação realizada. Já em um sistema com restrições de tempo estritas, a perda de prazos pode resultar em perdas catastróficas ao sistema. O processamento de fluxo de vídeo contínuo é um exemplo de aplicação com restrições de tempo suave, em que a perda de prazos pode resultar em um vídeo de qualidade inferior. Por outro lado em uma aplicação de controle de uma planta nuclear, a perda de prazos pode resultar na fusão de um núcleo radioativo, por exemplo.

Os sistemas embarcados que necessitam de intervenção humana devem ser interativos no sentido que, para uma dada ação a reação do sistema não seja lenta, não significando, porém, que a resposta tenha que ser em tempo real.

2.1.3 Necessidade de rede

O suporte a rede nos sistemas embarcados está se tornando um item obrigatório. Até mesmo em dispositivos em que as funções de navegação não são essenciais para o seu funcionamento. Neste contexto, pode-se citar os dispositivos embarcados em uma residência interconectados através de rede de automação doméstica.

2.1.4 Interação com o usuário

O grau de interação com o usuário varia bastante de um sistema para outro. Alguns sistemas têm suas funcionalidades direcionadas ao modo de interação com o usuário, é

o caso dos telefones portáteis, enquanto outros podem disponibilizar alguns meios para indicar se estão ligados e chaves que permitam desligar ou reiniciar o dispositivo. O controle do dispositivo também pode ser realizado via rede, através de uma interface web ou de um protocolo de comunicação como o Telnet.

2.2 Linux Embarcado

Devido a peculiaridades existentes nos processos de desenvolvimento, configuração de *hardware*, redução de funcionalidades de alguns aplicativos e da personalização da interface com o usuário, o uso do Linux nos sistemas embarcados é tratado de forma distinta. Linux embarcado é, portanto a arte de personalização e modificação de características existentes no Linux de modo a adaptá-lo às restrições dos sistemas embarcados.

Entre as principais razões para a grande aceitação do Linux no mercado de sistemas embarcados destacam-se: a grande quantidade de aplicativos de software existentes, o suporte as mais diversas arquiteturas e *hardware* existentes, as boas implementações dos protocolos de comunicação, o suporte da comunidade, a independência de fabricantes e a ausência de *royalties* nos produtos comercializados. Com o uso do Linux, os fabricantes podem se concentrar no diferencial do produto, ganhando tempo muito importante para o lançamento do produto no mercado.

2.2.1 Disponibilidade de software

As aplicações desenvolvidas pela comunidade que suporta o Linux são abertas, neste sentido, qualquer fabricante pode acessar o código livremente na internet, personalizá-lo de acordo com sua aplicação e embarcá-lo em um novo produto. A única obrigação do fabricante é a de manter a compatibilidade com a licença do código utilizado, que pode exigir que este disponibilize as modificações realizadas. Outro ponto importante é que nem todos os aplicativos do sistema necessitam ser reinventados.

2.2.2 Suporte de hardware

Hoje em dia, o Linux suporta uma grande quantidade de arquiteturas com as mais diversas configurações de hardware existentes. A adição de novos componentes de hardware também é facilitada, pois o núcleo do Linux é modular o que permite a utilização dos subsistemas existentes. As soluções de hardware utilizadas em projetos envolvendo Linux embarcado são apresentadas no Capítulo 3.

2.2.3 Protocolos de comunicação

São disponíveis no Linux implementações para os principais protocolos de comunicação. Vale ressaltar que uma das principais razões da proliferação do Linux no mercado de sistemas embarcados é o excelente suporte aos protocolos de rede e as bibliotecas de comunicação. Alguns dos módulos de comunicação do Linux são herdados do UNIX e do BSD, e são conhecidos por serem estáveis e bem testados. Outro fator importante é a integração dos serviços, de forma que, por exemplo, a implementação de um ponto de acesso que suporte Wi-Fi, Bluetooth e Ethernet, pode ser feita através de um simples *script*. Um script consiste de uma lista de comandos e funções que podem ser executados em terminais baseados no UNIX. O interpretador de scripts de maior sucesso no mundo UNIX é o aplicativo GNU Bash [19].

2.2.4 Suporte da comunidade

O suporte da comunidade é um dos principais trunfos de quem utiliza soluções baseadas em Linux. Este suporte é dado através de tutoriais, documentação, listas de *email* e fóruns *online*. Ao contrário dos ambientes corporativos, em um ambiente de código livre, os desenvolvedores sentem-se prestigiados ao ajudar outros desenvolvedores a resolver seus problemas.

2.2.5 Independência de fabricantes

A maioria das soluções proprietárias está intimamente relacionada a um determinado fabricante, o que muitas vezes limita a solução. No modelo do Linux sempre é possível procurar as melhores soluções técnicas, sendo possível até a aglutinação de soluções em busca de um objetivo comum.

2.2.6 Ausência de *royalties*

A ausência do pagamento de *royalties* permite que as soluções baseadas em Linux cheguem ao mercado a um preço mais acessível e em menor tempo. Hoje em dia, percebe-se como tendência global a migração de soluções proprietárias para soluções baseadas em código aberto. A gigante finlandesa Nokia é um exemplo clássico desta mudança de comportamento, primeiramente ao investir em soluções abertas nas plataformas maemo [31] e QT [45], e mais recentemente por anunciar a abertura do sistema operacional Symbian [44].

2.3 Componentes da solução

A grande quantidade de opções abertas disponível pode se tornar um empecilho na hora da concepção da solução embarcada. Para mitigar estes problemas, a escolha dos componentes do sistema pode ser baseada em aspectos como:

- tamanho da comunidade;
- atividade do projeto;
- funcionalidades implementadas;
- compatibilidade com padrões existentes;
- facilidade de reconfiguração.

2.4 Criação do sistema Linux

O sistema Linux é criado a partir da agregação e configuração dos componentes de softwares apropriados a solução desejada. Os passos para se criar um sistema Linux são:

- determinação dos componentes do sistema;
- configuração e construção do núcleo;
- construção do sistema de arquivos raiz;
- configuração do software de inicialização (*bootloader*).

A determinação dos componentes do sistema é realizada a partir da avaliação de quais funcionalidades devem ser providas no sistema. A partir desta análise, é possível obter uma lista de aplicativos e componentes de *software* que as provêem. A lista resultante pode apresentar redundâncias, que devem ser resolvidas de acordo com base em critérios visando: a minimização do sistema de arquivos raiz, melhor desempenho, a utilização de aplicativos mais conhecidos/suportados. A configuração do núcleo depende da plataforma utilizada e dos periféricos a ela conectados. Depois de construídos o sistema de arquivos raiz e o núcleo do sistema, a tarefa final consiste na escolha do bootloader usado na inicialização do sistema.

Os requisitos de projeto de sistemas embarcados utilizados neste texto levam em consideração os aspectos de simplicidade, configuração automática e a utilização de aplicativos bem estabelecidos.

2.5 Sumário

Neste capítulo foram apresentados os principais aspectos e escolhas referentes as entidades que compõe uma solução para sistemas embarcados.

Capítulo 3

Hardware para sistemas embarcados

Neste capítulo são apresentadas as principais configurações de *hardware* utilizadas em projetos com Linux embarcado. Estas configurações são apresentadas sob os aspectos de processadores, barramentos e interfaces, entrada e saída, sistemas de armazenamento e interfaces de comunicação.

3.1 Processadores

Embora o Linux possa ser utilizado com diversos tipos de arquiteturas, nem todas elas são apropriadas para sistemas embarcados. Dentre as mais apropriadas estão as que utilizam os processadores ARM [28], MIPS [2], PPC [3] e o tradicional X86 [14]. Segundo uma pesquisa anual [24] as arquiteturas citadas são as mais utilizadas em projetos de sistemas embarcados. O resultado é apresentado na Figura 3.1[1].

3.1.1 ARM

ARM[2] é uma família de processadores desenvolvida pela ARM Holdings LTDA. Ao contrário de seus concorrentes, a ARM não fabrica seus processadores, ela apenas define os núcleos de CPU para os seus clientes, baseados no núcleo ARM, e deixa que eles se encarreguem da fabricação dos chips. A ARM Holdings também desenvolve tecnologias que auxiliam o desenvolvimento como: ferramentas de *software*, placas de desenvolvimento, ferramentas de depuração *hardware*/*software*, arquiteturas em barramento e periféricos.

Os processadores ARM compartilham o mesmo conjunto de instruções, de modo que o *software* permaneça compatível entre as diversas variações apresentadas.

No mercado de aparelhos eletrônicos portáteis a arquitetura ARM é dominante, principalmente, por apresentar soluções que utilizam baixo consumo de energia. Dentre os

[1] Fonte: www.linuxdevices.com

[2] ARM - *Advanced RISC Machine*

principais clientes da ARM estão lideres mundiais no mercado de semicondutores como a Texas Instruments, a Intel e a Freescale.

- Texas Instruments ARM:
  - Usa núcleos ARM na família de processadores integrados OMAP[3];
  - Os processadores OMAP apresentam muitos periféricos integrados em um único *chip*;
  - São apropriados para uso em dispositivos multimídia.
- Intel ARM XScale:
  - Uso de processadores integrados baseados na arquitetura ARM;
  - Linhas de produtos voltadas para dispositivos portáteis e equipamentos de rede para processamento rápido de pacotes.
- Freescale ARM:
  - A Freescale produz chips dedicados com vários periféricos integrados, requeridos para usao em aplicações multimídia;
  - Os produtos são amplamente utilizados em aplicações envolvendo multimídia como plataformas de jogos portáteis, PDAs e telefones celulares.

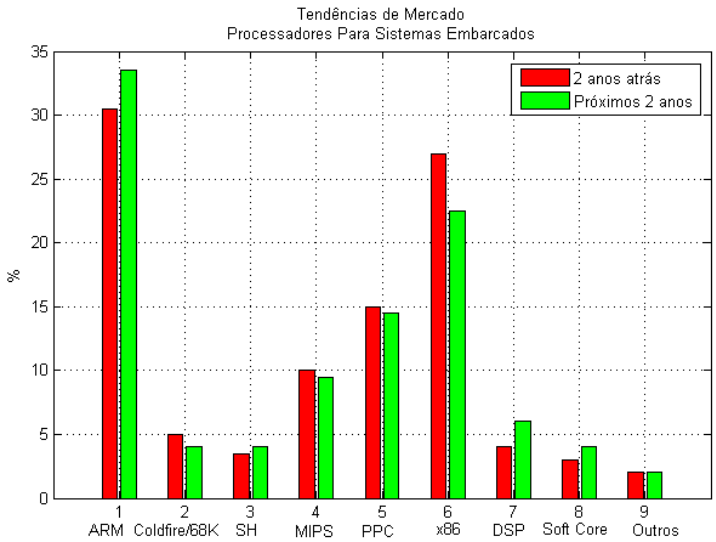

Figura 3.1: Tendências na escolha de processadores para projetos de sistemas embarcados

[3]OMAP-Open Multimedia Applications Platform

3.1.2 MIPS

MIPS é uma arquitetura micro-processada desenvolvida pela MIPS Technologies. O modelo de negócios é o mesmo usado pela ARM, ou seja, a MIPS licencia núcleos de sua arquitetura para que terceiros manufaturarem. Mas ao contrário da ARM, o conjunto de instruções MIPS não é o mesmo. A variação do MIPS de maior sucesso é a desenvolvida pela Broadcom.

No mercado de sistemas embarcados a MIPS está presente em videogames, impressoras, roteadores, pontos de acesso, modems, *set-top boxes*, PDAs e telefones celulares.

3.1.3 PPC

PowerPC é uma arquitetura RISC criada em 1991 a partir de uma aliança entre Apple, IBM e Motorola. Entre os anos de 1994 e 2006 foi utilizada na linha de computadores Macintosh da Apple.

No mercado de sistemas embarcados, esta arquitetura está presente em subsistemas usados em automóveis, em equipamentos de rede e em produtos voltados ao usuário final.

3.1.4 X86

X86 é o nome genérico dado a família de processadores baseados no Intel 8086. O surgimento desta família data de 1985 com o desenvolvimento do 386 pela Intel, a partir de então vieram os processadores 486, a família Pentium, até os processadores atuais de 4 núcleos. Paralelamente, a AMD e a National desenvolveram processadores compatíveis.

Apesar de ser a arquitetura mais popular no mercado de computação pessoal, o X86 não conseguiu repetir o mesmo sucesso no mundo embarcado. Hoje em dia, os processadores Intel ATOM e os padrões nano-ITX da VIA tentam colocar esta arquitetura na cena do mercado de sistemas embarcados.

Apesar de não ser muito utilizada em produtos finais, é nesta arquitetura que são desenvolvidas as aplicações embarcadas.

3.2 Barramentos

A conexão entre os periféricos no sistema e a CPU é de realizada por barramentos e interfaces. O suporte no Linux aos diversos barramentos existentes é implementado no núcleo do sistema. O Linux suporta vários tipos de barramentos e dentre os mais utilizados em sistemas embarcados, os seguintes estão disponíveis para o Linux: ISA, PCI, PCMCIA, PC/104, SCSI, USB e I2C.

3.3 Entrada e saída

As operações de entrada e saída desempenham um papel fundamental em qualquer computador. O Linux possibilita o uso de uma grande variedade de dispositivos de entrada e saída. A interligação desses sistemas no núcleo do Linux é realizada de duas formas principais: através da implementação nativa do *driver* que gerencia a conexão com o sistema e através de camadas genéricas que permitem a interconexão de vários dispositivos de uma mesma classe, como é o caso do USB.

O Linux proporciona a interconexão de dispositivos de entrada e saída através de interfaces serial, paralela, USB e *firewire*. Além de prover suporte à dispositivos tradicionais como teclados, mouses, *displays*, monitores, impressoras e placas de rede, o Linux também possibilita o uso sistemas de aquisição de dados, controle e sistemas de automação doméstica.

3.4 Armazenamento

A maioria dos sistemas embarcados não utiliza discos rígidos, pois estes têm partes móveis, são volumosos, sensíveis a choques físicos e necessitam de fontes de alimentação externa. Desta forma, os dispositivos descritos como MTD [4] no núcleo do Linux que incluem memórias do tipo ROM, RAM e FLASH são os mais utilizados em projetos de sistemas embarcados.

Os dois tipos de memória FLASH encontrados são o NOR e o NAND. As memórias FLASH do tipo NOR têm sido tradicionalmente usadas para armazenar pequenas quantidades de código executável em dispositivos embarcados. Memórias NOR são mais confiáveis, apresentam operações de leitura mais rápidas e permitem acesso aleatório. São ideais para armazenar *firmwares*, código de inicialização, sistemas operacionais e outros dados que mudam com pouca freqüência.

As memórias FLASH do tipo NAND se tornaram o formato preferido para armazenamento de grandes quantidades de dados em dispositivos como USB FLASH *drivers*, câmeras digitais e tocadores de MP3. Por apresentar características como alta densidade, baixo custo, tempos de escrita e apagamento baixos além de permitir um ciclo maior de leituras e apagamentos, as memórias do tipo NAND são amplamente utilizadas em aplicações que necessitam da leitura seqüencial de grandes arquivos de maneira rápida, com a possibilidade de serem trocados freqüentemente.

Todos os sistemas embarcados requerem, no mínimo, uma forma de armazenamento persistente para inicializar, mesmo que seja apenas para o processo de *boot*. A maioria dos

[4] MTD - *Memory Tecnology Devices*

sistemas, incluindo os que utilizam Linux embarcado, continuam a utilizar este dispositivo de armazenamento para acessar código ou dados. Comparado aos sistemas embarcados tradicionais, o Linux requer mais do *hardware* de armazenamento, seja em termos de tamanho ou em termos de organização.

3.5 Comunicação

Hoje em dia, a maioria dos sistemas embarcados necessita de uma interface de comunicação, seja para enviar, seja para receber dados remotos. Um dos principais ganhos ao se utilizar Linux é que o mesmo suporta os principais padrões e protocolos de comunicação existentes. Dentre as principais interfaces de comunicação suportadas pelo Linux pode-se citar: Ethernet, Infravermelho (IrDA), Bluetooth, IEEE 802.11 (Wi-Fi), IEEE 802.16 (Wimax) e UWB[5].

Ethernet

A tecnologia Ethernet (IEEE 802.3) é uma das mais utilizadas em sistemas embarcados, seja no ambiente de desenvolvimento, seja em produtos finais que não imponham restrições ao uso de cabos. O uso desta tecnologia encontra-se maduro no Linux, desta forma é possível observar que uma grande variedade de placas comunicação Ethernet, com capacidades de transmissão entre 10 Mbps e 10 Gbps, podem ser utilizadas com o Linux.

IrDA

O protocolo IrDA já foi mais utilizado em sistemas embarcados, hoje em dia ele vem perdendo espaço para os protocolos Bluetooth e UWB. Os concorrentes do IrDA apresentam maior alcance, e não sofrem do problema da falta de diretividade.

Apesar de ter os dias contados, este protocolo também é suportado no Linux e pode ser utilizado em projetos de sistemas embarcados.

Bluetooth

A tecnologia de comunicação Bluetooth é designada para o estabelecimento de enlaces de comunicação sem fio de curto alcance, 10 metros para os dispositivos classe 2 e 100 metros para os dispositivos classe 1. As conexões realizadas usando Bluetooth podem ser ponto a ponto e ponto a multiponto.

O suporte oficial no Linux é obtido através da utilização da pilha Bluez.

[5]UWB - *Ultra Wide Band*

IEEE 802.11

O padrão IEEE 802.11, conhecido popularmente como WiFi, é utilizado para o estabelecimento de redes locais sem fio que abrangem um alcance de até 100 metros. As taxas de comunicação neste protocolo variam de 11 a 54 Mbps.

O Linux suporta a maioria dos cartões WiFi existentes no mercado, algumas exceções ocorrem devido aos fabricantes de *hardware* não liberarem as especificações do *hardware*. Retirando-se estas exceções, o Linux suporta os protocolos de criptografia WEP, WPA, WPA2 usados nesta tecnologia.

IEEE 802.16

O padrão IEEE 802.16, conhecido como Wimax, promete ser um dos concorrentes as redes de telefonia 3G existentes com taxas de transmissão de até 75 Mbps. O suporte no Linux foi incorporado desde o surgimento desta tecnologia. Inclusive, já existem produtos embarcados que utilizam Linux com suporte a Wimax.

UWB

UWB (IEEE 802.15.3) é uma tecnologia de curto alcance com taxas de transmissão de até 60 Mbps que pretende substituir as tecnologias USB e Bluetooth.

Apesar de ainda não ter se popularizado, esta tecnologia também já é suportada no núcleo do Linux.

3.6 Sumário

Neste capítulo foram apresentados os principais elementos de *hardware* utilizados em projetos com Linux embarcado. Um aspecto importante a ser considerado é que os elementos usados no ambiente de desenvolvimento, guardadas algumas ressalvas, também podem ser utilizados no ambiente de produção, ou seja, no sistema embarcado. Isto ocorre pelo fato do Linux para dispositivos embarcados ser o mesmo que o Linux para ambientes de computação pessoal e de servidores.

Capítulo 4

Ambiente de desenvolvimento

A configuração do ambiente de desenvolvimento para sistemas embarcados envolve duas entidades fundamentais, a plataforma hospedeira e a plataforma alvo. A comunicação entre estas entidades geralmente é realizada através de conexões de rede ou seriais. O ambiente de desenvolvimento é ilustrado na Figura 4.1.

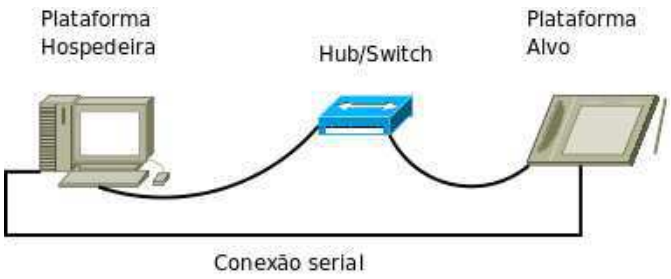

Figura 4.1: Ambiente de desenvolvimento típico

A necessidade de uma plataforma hospedeira é justificada pela baixa capacidade de processamento e de armazenamento da plataforma alvo, bem como da ausência de interfaces de entrada e saídas apropriadas. Por exemplo, no ambiente de desenvolvimento considerado neste trabalho a estação hospedeira é um computador com processador com dois núcleos de 2,13 Ghz e 1Gb de memória RAM, enquanto que a estação alvo é um OMAP 1610 H2 com processador de 192 Hz e 32 Mb de memória RAM. Na maioria das vezes, a plataforma alvo apresenta uma arquitetura diferente da plataforma hospedeira. Neste caso, devem-se utilizar compiladores cruzados para que seja produzido, na plataforma hospedeira, código executável para a plataforma alvo.

O compilador cruzado é o principal componente de um conjunto de ferramentas denominado *toolchain*, o qual é detalhado na seção seguinte.

Além dos *toolchains*, o ambiente de desenvolvimento é composto de servidores de rede para o fornecimento de arquivos (TFTP, NFS) e atribuição de endereços aos clientes na rede (BOOTP, DHCP).

TFTP

O TFTP é um protocolo de transferência de arquivos semelhante ao FTP. Na plataforma hospedeira ele é usado para prover imagens de núcleos recém compilados para teste, antes que estes sejam gravados em meios persistentes. Em máquinas que utilizam a distribuição Ubuntu, este serviço é gerenciado pelo programa *openbsd-inetd*. Para configurar o *openbsd-inetd* basta acrescentar uma linha ao arquivo */etc/inetd.conf*, conforme apresentado na listagem a seguir:

```
tftp dgram udp wait nobody /usr/sbin/tcpd /usr/sbin/in.tftpd /realtime/kernel_images
```

NFS

O NFS é um sistema de arquivos via rede muito útil para o desenvolvimento de sistemas embarcados, devido às seguintes características:

- O tamanho do sistema de arquivos raiz não se restringe a capacidade de armazenamento da plataforma alvo;
- As mudanças feitas no sistema de arquivos da plataforma alvo, realizadas na plataforma hospedeira, são refletidas imediatamente na plataforma alvo;
- É possível testar primeiro as mudanças no núcleo do sistema, antes do desenvolvimento de um sistema de arquivos raiz para a plataforma alvo.

4.1 *Toolchains*

O *toolchain* é composto por um conjunto de ferramentas que permitem a compilação de códigos fonte para um dado sistema alvo. Seus componentes são o compilador C, os utilitários para a manipulação de arquivos binários, uma biblioteca C e os cabeçalhos do núcleo do Linux.

A configuração e construção de um *toolchain* a partir do código fonte das aplicações componentes é uma tarefa complexa que envolve vários conhecimentos. Devido à complexidade da tarefa e ao modo como ela está relacionada às versões dos softwares componentes, geralmente são utilizados *toolchains* compilados por terceiros. A obtenção destes *toolchains* depende da arquitetura alvo, dos componentes do *toolchain* e da versão dos aplicativos e bibliotecas disponibilizados.

As fontes primárias de *toolchains* para desenvolvimento de aplicações embarcadas com Linux são mostradas na Tabela 4.1.

Fonte	Arquitetura	Endereço
CodeSourcery	ARM, MIPS, PPC	http://www.codesourcery.com
Linux Omap	ARM (OMAP)	http://linux.omap.com
Embedded Debian	ARM, PPC	http://www.emdebian.org/

Tabela 4.1: Fontes de *Toolchains*

Os *toolchains* também podem ser construídos utilizando ferramentas baseadas em *scripts* e receitas. Com estas ferramentas é possível a construção de um *toolchain* personalizado. Algumas destas ferramentas possibilitam a criação de um sistema de arquivos raiz personalizado. Dentre as ferramentas que compõe esta opção se destacam o Buildroot [5], o Crosstool [21] e o OpenEmbedded [36].

4.1.1 Componentes do *toolchain*

Compilador C

O principal compilador utilizado em projetos de Linux e Linux embarcado é o GCC[1]. Além da linguagem C, o GCC é *frontend* para as linguagens C++, Fortran, Objc, Obj-C++ e Java. Estas linguagens podem ser habilitadas na configuração do *toolchain*.

Utilitários para a manipulação de arquivos binários

Os utilitários para a manipulação de arquivos binários são usados na manipulação dos arquivos objeto gerados pelo compilador. Cada um desses utilitários é descrito na Tabela 4.2. Os utilitários mais importantes são o *assembler*, as, o *linker*, ld, e o strip.

Biblioteca C

As bibliotecas C provêem as principais funcionalidades do software do sistema embarcado. No Linux existem duas opções principais: a Glibc e a uClibc que são descritas em detalhes no capítulo Sistemas de arquivos.

4.1.2 Ferramentas para a geração automática de *toolchains*

Buildroot

O Buildroot é composto por um conjunto de Makefiles e *patches* que permitem a geração *toolchains* e de sistemas de arquivos raiz para sistemas embarcados usando a biblioteca

[1]GCC- GNU Compiler Collection

Utilitário	Função
as	GNU assembler
ld	GNU Linker
gasp	Pré-processador do assembler GNU
ar	Cria e manipula conteúdo de arquivos
nm	Lista os símbolos em um arquivo objeto
objcopy	Copia e traduz arquivos objeto
objdump	Mostra as informações do arquivo objeto
ranlib	Gera um índice para o conteúdo de um arquivo
readelf	Mostra informações sobre um arquivo objeto no formato ELF
size	Mostra o tamanho das seções dentro de um arquivo objeto
strings	Imprime o texto dos caracteres imprimíveis no arquivo objeto
strip	Retira os símbolos dos arquivos objeto
c++filt	Converte rótulos resultantes da sobrecarga de funções em C++ em seus nomes no espaço do usuário
addr2line	Converte endereços em números de linha dentro dos arquivos fonte originais

Tabela 4.2: Utilitários para a manipulação de arquivos binários

uClibc. O Buildroot suporta 18 arquiteturas diferentes, entre as principais se destacam PPC, MIPS e ARM.

Os passos para a construção de um *toolchain* usando o Buildroot são mostrados a seguir:

1. Obtenha a última versão do buidroot:

```
wget http://buildroot.uclibc.org/downloads/snapshots/buildroot-snapshot.tar.bz2
```

2. Descompacte o arquivo baixado:

```
tar xjvf buildroot-snapshot.tar.bz2
cd buildroot
```

3. Escolha as opções para o sistema de arquivos raiz e o *toolchain*, a partir da interface gráfica, conforme Figura 4.2:

```
make menuconfig
```

4. Compile os arquivos:

```
make
```

5. Após 18 minutos de compilação[2], desconsiderando-se o tempo em que os pacotes são carregados pela rede, o Buildroot constrói o *toolchain* e o sistema de arquivos raiz. No processo de construção são utilizados 1.4 GB de espaço em disco.

[2]Dados de tempo obtidos usando um computador Intel com dois núcleos, processador de 2,13 Ghz e 1 Gb de memória RAM

- O *toolchain* é salvo no diretório build_arm/staging_dir/
- O sistema de arquivos é salvo no diretório project_build_arm/uclibc/root/

6. Para testar o *toolchain* pode-se criar um programa hello.c e compilá-lo executando o comando:

```
/build_arm/staging_dir/usr/bin/arm-linux-gcc hello.c -o hello
```

7. O comando *file* permite verificar o tipo do binário gerado:

```
jluisn@fenix:/realtime/buildroot$ file hello
hello_uclib: ELF 32-bit LSB executable, ARM, version 1,
dynamically linked (uses shared libs), not stripped
```

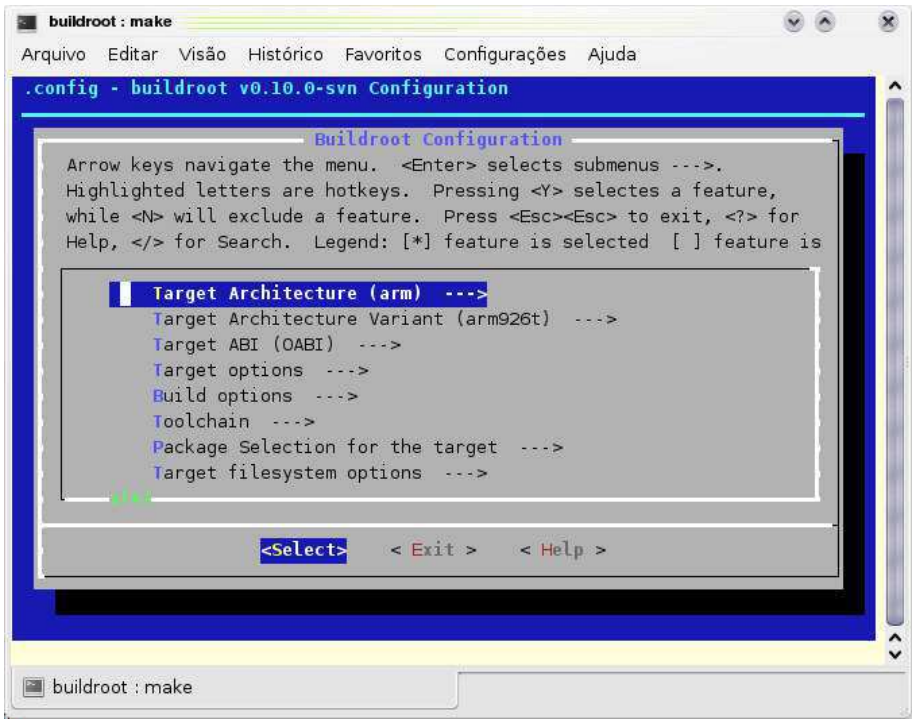

Figura 4.2: Tela de configuração do Buildroot

4.1.3 Crosstool

O Crosstool é um conjunto de *scripts* para construção e teste de *toolchains* Linux usando a biblioteca Glibc. Embora permita a combinação de várias versões de GCC e Glibc, apenas algumas delas resultam em *toolchains* funcionais. Uma tabela completa das combinações de configuração que produzem *toolchains* válidos é encontrada no endereço http://www.kegel.com/crosstool/crosstool-0.43/buildlogs/. Entre as principais arquiteturas suportadas pelo Crosstool estão: Alfa, ARM, MIPS, PPC e Sparc.

Os passos para a construção de um *toolchain* usando o Crosstool são mostrados a seguir:

1. Carregue e descompacte os arquivos:

```
wget http://kegel.com/crosstool/crosstool-0.43.tar.gz
tar -xzvf crosstool-0.43.tar.gz
```

2. Crie a pasta /opt/crosstool com permissão de escrita para o usuário $USER:

```
sudo mkdir /opt/crosstool
sudo chown $USER /opt/crosstool
```

3. A construção de um *toolchain* para a plataforma ARM sem suporte a ponto flutuante, é obtida a partir do script *demo-arm-softfloat.sh*. Por padrão, este *script* gera um *toolchain* com o gcc-3.4.5 e a glibc-2.3.6. Para mudar as versões das bibliotecas e as configurações das variáveis de ambiente basta editar o *script*. Com tudo pronto basta executar o *script*:

```
./demo-arm-softfloat.sh
```

4. Após cerca de 25 minutos de compilação e com 1.1 GB de espaço em disco usados na construção, o *toolchain* é instalado no diretório: /opt/crosstool/gcc-3.4.5-glibc-2.3.6/arm-softfloat-linux-gnu/.

   - O *toolchain* é salvo no diretório /opt/crosstool/gcc-3.4.5-glibc-2.3.6/arm-softfloat-linux-gnu/

5. Para testar o *toolchain* pode-se criar um programa hello.c e compilá-lo com o comando:

```
export PATH=$PATH:/opt/crosstool/gcc-3.4.5-glibc-2.3.6/arm-softfloat-linux-gnu/bin
arm-softfloat-linux-gnu-gcc hello.c -o hello
```

6. Com o comando *file* verifica-se o tipo do binário gerado:

```
jluisn@fenix:/realtime/crosstool-0.43$ file hello
hello: ELF 32-bit LSB executable, ARM, version 1, for GNU/Linux 2.4.3,
dynamically linked (uses shared libs), not stripped
```

4.1.4 OpenEmbedded

O OpenEmbedded é uma meta distribuição que facilita a criação de distribuições de Linux embarcado a partir do código fonte. Em relação às ferramentas até então apresentadas, o OpenEmbedded é a ferramenta que apresenta a curva de aprendizagem mais acentuada.

Os dois principais componentes do OpenEmbedded são:

- O executor de tarefas: o utilitário bitbake;
- Os arquivos de metadados: Consistem de milhares de regras na forma de arquivos especiais. Estes arquivos é que são interpretados pelo executor de tarefas.

Os metadados do Open Embedded são uma estrutura de dados complexa que definem três unidades fundamentais na distribuição embarcada.

- Definições do tipo de máquina;
- Definições do tipo de distribuição;
- Definições dos pacotes individuais.

Um tutorial completo de utilização do OpenEmbedded é disponibilizado no endereço http://wiki.openembedded.net/index.php/Getting_Started. O OpenEmbedded é utilizado na criação das distribuições embarcadas Poky [27], Mamona [20], Openmoko [37] e OpenWrt [38]. O tempo gasto na compilação de uma distribuição embarcada usando o OpenEmbedded pode passar de 1 dia.

4.2 Sumário

Neste capítulo foram apresentados os componentes utilizados para a geração de sistemas de arquivos raiz para distribuições embarcadas. Foram apresentadas três ferramentas para a geração automática de *toolchains*, os componentes gerados por elas, a complexidade de uso delas e o tempo que elas consomem na geração do sistema.

O tempo é um fator relevante para a obtenção de um ambiente de desenvolvimento funcional, não definitivo, no qual alguns aspectos gráficos podem ser deixados em segundo plano. Ou seja, quando é necessário testar as características básicas do sistema sem a necessidade de um visual final. Neste caso pode-se utilizar um ambiente mais simples, o qual poderá evoluir para um ambiente mais complexo quando as características funcionais do produto já estiverem implementadas e testadas.

Capítulo 5

O núcleo do Linux

O núcleo do Linux é o principal componente de uma distribuição Linux. Em sistemas embarcados o núcleo do Linux ganha mais importância, pois é possível com a adição de poucos aplicativos, obter um ambiente inteiramente funcional. Uma configuração funcional usada em sistemas embarcados envolve um *bootloader*, o núcleo do Linux e o aplicativo Busybox para disponibilizar as funcionalidades básicas do sistema.

Neste capítulo são tratados os principais conceitos envolvidos na seleção, escolha, configuração e instalação do componente principal do *software* de qualquer sistema embarcado, o núcleo do sistema operacional.

5.1 Seleção

Apesar do código do Linux suportar várias arquiteturas e dispositivos, as versões mais novas de componentes específicos dependentes de arquitetura têm que ser obtidas a partir de projetos paralelos ao kernel.org[1], principal repositório do núcleo do Linux. Excetuando-se o caso da arquitetura do sistema embarcado ser x86, nas demais arquiteturas é necessário obter códigos ainda não sincronizados com a árvore principal do núcleo do Linux. As principais fontes para obter códigos específicos de arquitetura para o núcleo do Linux são listadas na Tabela 5.1

Em alguns destes sítios o código fonte completo não está disponibilizado com as modificações realizadas dependentes da arquitetura, ao invés disso são disponibilizados arquivos com as diferenças (*patches*). Os *patches* devem ser aplicados a versão do Linux obtida do kernel.org. Portanto, para ter acesso às últimas funcionalidades implementadas para a plataforma OMAP, por exemplo, deve-se obter o código fonte do núcleo do Linux, linux-2.6.24.tar.bz2 a partir do kernel.org e o patch patch-2.6.24-omap1.bz2 a partir do endereço

[1] Além de hospedar as últimas versões do código fonte do núcleo do Linux, o kernel.org hospeda outros projetos de código aberto

Arquitetura	Localização dos arquivos fonte
X86	http://www.kernel.org
ARM	http://www.arm.linux.org.uk/developer/
ARM-OMAP	http://www.muru.com/linux/omap
PowerPC	http://penguinppc.org
MIPS	http://www.linux-mips.org

Tabela 5.1: Fontes de obtenção do Linux

www.muru.com.

5.2 Configuração

A configuração das características do núcleo do Linux a serem selecionadas, depende obviamente das funcionalidades desejadas para o sistema embarcado, dos periféricos conectados ao sistema e dos protocolos que devam ser suportados. A maioria das placas usadas em sistemas embarcados tem uma configuração padrão localizada nos diretório arch/ARCH/configs, onde ARCH é a arquitetura da placa em questão. Para configurar o núcleo do Linux para ser usado em uma plataforma OMAP 1610, por exemplo, basta digitar o comando make omap_h2_1610_defconfig que um arquivo de configuração .config é gerado.

A partir da geração do arquivo de configuração base é possível navegar nas opções de configuração, inclusive usando a interface gráfica, para a escolha de algumas características adicionais. A lista de opções completa engloba os seguintes tópicos:

- Configuração geral;
- Suporte ao carregamento de módulos;
- Dispositivos de bloco;
- Tipo de processador e características;
- Opções de gerenciamento de energia;
- Opções de barramentos;
- Tipos de arquivos executáveis;
- Rede;
- Drivers dos dispositivos;

- Firmware;
- Sistemas de arquivos;
- Suporte a instrumentação;
- Hacking do núcleo;
- Opções de segurança;
- API de criptografia;
- Biblioteca de rotinas.

Os detalhes de cada um desses subsistemas podem ser obtidos na ajuda disponibilizada pela própria ferramenta de configuração do núcleo.

As características do núcleo podem ser configuradas de 3 formas básicas. Na primeira forma de configuração com o comando make config, o usuário necessita responder Y/N/M para selecionar/de selecionar/usar como módulo cada uma das várias opções presentes no núcleo. Na segunda forma de configuração, com o comando make menuconfig, o usuário tem uma interface de configuração baseada em um terminal que permite a escolha através da navegação entre os diversos subsistemas do núcleo do Linux com o teclado. Na terceira forma de configuração, com o comando make xconfig/gconfig a seleção dos componentes do sistema é realizada a partir de uma interface gráfica utilizando o mouse. As interfaces de configuração menuconfig e xconfig/gconfig são mostradas nas Figuras 5.1 e 5.2, respectivamente.

5.3 Compilação

Depois de configurado, o núcleo do Linux já está pronto para ser compilado. Nesta etapa o único cuidado que deve ser tomado é na especificação do compilador cruzado. O compilador cruzado deve estar na variável de ambiente PATH, de modo que seja possível executar o comando *arm-linux-gcc*, no caso de compilação para a plataforma ARM, por exemplo.

Desta forma a compilação do núcleo é obtida com o comando *make ARCH=arm CROSS_ COMPILE=arm-linux-*. Em arquiteturas hóspedes com múltiplos núcleos é possível obter uma redução razoável do tempo de compilação paralelizando o comando make através da opção -jX, onde X pode ser o número de núcleos do sistema.

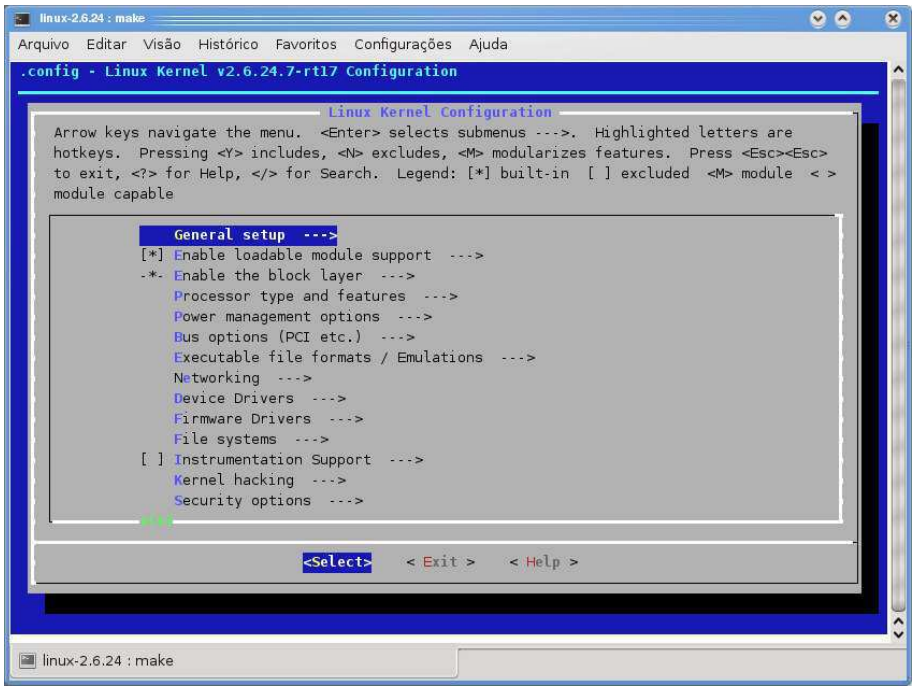

Figura 5.1: Tela de interface gráfica gerada pelo comando make menuconfig

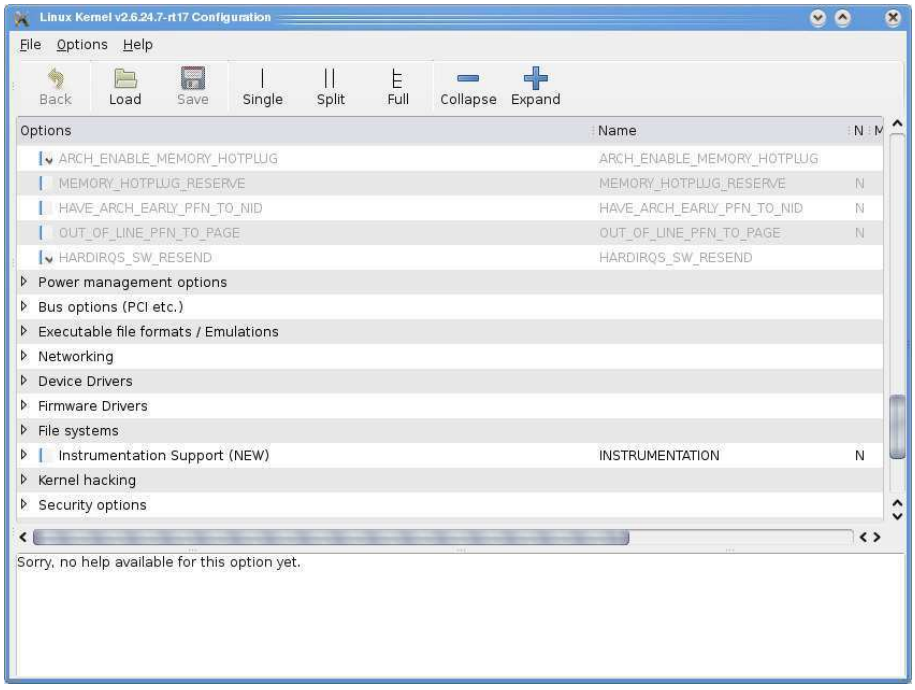

Figura 5.2: Tela da interface gráfica gerada pelo comando make xconfig/gconfig

5.4 Instalação

A imagem compilada pode ser instalada em qualquer diretório, bastando para isto a inclusão da variável INSTALL_PATH na linha de comando do *make*, ou seja, *make install INSTALL_PATH=/mytarget/boot*. Após a instalação do núcleo deve-se instalar os módulos do núcleo com o comando *make modules_install INSTALL_MOD_PATH=/mytarget*.

A partir de então, o sistema de arquivos básico da plataforma alvo já possui o núcleo do sistema, que será complementado com mais funcionalidades a serem descritas nos capítulos posteriores.

5.5 Sumário

Neste capítulo foi mostrado o processo de obtenção e configuração do núcleo do Linux. Um dos pontos importantes a se ressaltar é que as configurações são dependentes de projeto, ou seja, dependem entre outros detalhes do *hardware* utilizado, das interfaces de comunicação, dos protocolos, dos dispositivos de armazenamento e da arquitetura do sistema.

Capítulo 6

Bootloaders

O *bootloader* é um dos principais componentes do *software* embarcado no sistema. Seu objetivo é garantir o carregamento do núcleo do sistema operacional e então passar o controle para ele. Nos sistemas tradicionais, a configuração inicial do sistema é realizada pela BIOS[1]. Nos sistemas embarcados a configuração inicial é realizada pelo *bootloader*. E por isso, os *bootloaders* de sistemas embarcados devem apresentar as seguintes características mínimas:

- Inicialização do *hardware*, especialmente o controlador de memória;
- Prover os parâmetros de inicialização ao núcleo do Linux;
- Iniciar o núcleo.

Para facilitar o processo de desenvolvimento, a maioria dos *bootloaders* provê características adicionais:

- Leitura e escrita em locações de memória arbitrárias;
- Carregamento de imagens binárias para a memória RAM do sistema alvo através de interfaces RS-232 e Ethernet;
- Cópia de imagens binárias da memória RAM para a memória FLASH.

Existem vários *bootloaders* para sistemas embarcados. Os mais usados e que suportam mais arquiteturas são o U-Boot e o RedBoot. Ambos incluem implementações para os protocolos BOOTP[2] e DHCP[3] para a atribuição dinâmica de endereços IP e o protocolo TFTP para o download de imagens de arquivos que também podem ser obtidos através de comunicação RS-232.

[1]BIOS - Basic Input/Output System
[2]BOOTP - *Bootstrap Protocol*
[3]DHCP - *Dynamic Host Configuration Protocol*

6.1 U-Boot

Conhecido como *bootloader* universal, o U-Boot é um dos *bootloaders* mais utilizados por sua flexibilidade. O seu código foi desenvolvido de modo a permitir que novos dispositivos sejam suportadas de maneira fácil. O U-Boot suporta as arquiteturas X86, ARM, PowerPC e MIPS.

O projeto U-Boot está hospedado no endereço http://www.denx.de/wiki/U-Boot. Este sítio disponibiliza o código fonte para download, uma documentação bem escrita e explicativa, explicações sobre o processo de desenvolvimento e as formas de contato. Devido ao U-Boot ser a principal escolha em relação ao *software* de *boot* dos sistemas embarcados, seu modo de funcionamento e os principais comandos são detalhados a seguir.

6.1.1 Inicialização com o U-Boot

A inicialização típica de um sistema embarcado usando o U-Boot é realizada segundo o esquema apresentado na Figura 6.1. Inicialmente, o U-Boot executa o comando *bootp*, o qual obtém através dos protocolos DHCP e BOOTP um endereço IP válido e a localização do servidor de imagens. Após a obtenção de um IP válido, o U-Boot carrega a imagem do núcleo em memória RAM através do comando *tftpboot 0x100000000 uImage* usando o protocolo TFTP. O endereço de memória para o qual a imagem é carregada pode ser diferente. Com a imagem carregada e gravada em memória RAM é executado o comando *bootm*, que ler a última posição do início de uma operação de gravação. Encontrada a imagem do núcleo, o U-Boot a descompacta, preenche alguns registradores com informações do sistema e passa o controle para o núcleo do Linux.

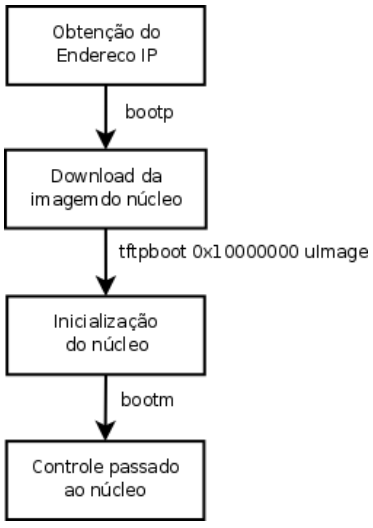

Figura 6.1: Esquema de inicialização usando o U-Boot

O arquivo uImage é uma imagem do núcleo gerada para o U-Boot com a ferramenta

mkimage provida nos arquivos fonte do mesmo. Este arquivo trata-se da imagem do núcleo do Linux com um cabeçalho que tem as seguintes informações:

- Sistema operacional alvo;
- Arquitetura de CPU alvo;
- Tipo de compactação;
- Endereço de carregamento;
- Ponto de entrada;
- Nome da imagem;
- Data da imagem.

O cabeçalho é marcado com um número mágico especial e, junto com os dados, é protegido contra erros com um código de redundância cíclica de 32 bits.

Para que o processo de inicialização funcione conforme mostrado no esquema da Figura 6.1 as variáveis de ambiente do U-Boot têm que estar configuradas corretamente. A manipulação destas variáveis é realizada a partir dos comandos *printenv* e *setenv*. Os valores das variáveis de ambiente podem ser observados na listagem seguinte:

```
OMAP1610 H2 # printenv
bootdelay=3
baudrate=115200
autoload=n
ethaddr=00:50:C2:27:1A:41
filesize=1597c8
fileaddr=10000000
bootargs=mem=32M console=ttyS0,115200n8 noinitrd root=/dev/nfs init=linuxrc
rw nfsroot=192.168.1.232:/realtime/buildroot,nolock ip=dhcp
bootfile=pxelinux.0
gatewayip=192.168.1.1
netmask=255.255.255.0
ipaddr=192.168.1.157
serverip=192.168.1.232
bootcmd=bootp;set serverip 192.168.1.232;tftp 0x10000000 uImage;bootm
stdin=serial
stdout=serial
stderr=serial

Environment size: 454/131068 bytes
```

6.1.2 Gravando imagens binárias na memória FLASH

Anteriormente foi mostrado o caso típico de inicialização em um ambiente de desenvolvimento, em que todas as imagens executáveis são obtidas através da rede. Embora útil no processo de desenvolvimento, esta abordagem é impraticável no ambiente de produção.

No ambiente de produção tanto a imagem do núcleo quanto a imagem do sistema de arquivos são armazenadas na memória FLASH do dispositivo ou outra forma de armazenamento não volátil. Para gravar uma dada imagem na memória FLASH do dispositivo, o caminho mais rápido consiste em obtê-la através do comando tftpboot e depois transferir o conteúdo da memória RAM para a memória FLASH, conforme a listagem a seguir:

```
=> erase 0x40100000 0x401FFFFF (Apaga 1M de espaço)
=> tftpboot 0x10000000 uImage  (baixa o núcleo para a RAM)
=> printenv filesize (verifica o tamanho do arquivo)
=> cp.b 0x10000000 0x40100000 $(filesize)  (copia da RAM para a flash)
=> imi 0x40100000 (verifica os dados copiados)
```

6.2 RedBoot

O RedBoot é um *bootloader* para sistemas embarcados baseado na camada de abstração de *hardware* do eCos. Além de ser baseado no HAL[4] do eCos[5], o RedBoot herdou deste as características de confiabilidade, portabilidade, compactação e configuração. O eCos é um sistema operacional em tempo real voltado para aplicações embarcadas.

Além das características descritas anteriormente, o RedBoot ainda prove um canal de comunicação com o GDB[6] para a depuração de aplicações através de uma linha serial ou via Ethernet.

6.3 Sumário

Neste capítulo foram detalhados aspectos referentes a escolha e configuração do *bootloader* para um sistema embarcado. Observou-se que o papel do *bootloader* em um sistema embarcado é um pouco mais complexo que em um sistema para computadores domésticos. Pois os últimos dispõem de uma BIOS que inicializa alguns registradores antes de passar o controle para o *bootloader*.

[4]HAL - Hardware Abstract Layer
[5]embedded Configurable operating system
[6]GDB - GNU debugger

Capítulo 7

Sistemas de arquivos

O sistema de arquivos raiz é um dos principais componentes do *software* do sistema embarcado, uma vez que, nele é que estão presentes diferentes soluções de *software* que implementam as funcionalidades do dispositivo embarcado. Nas próximas seções são observados alguns dos principais detalhes dos sistemas de arquivos, como a estrutura básica, as opções de biblioteca C, o núcleo do sistema, os sistemas de arquivos especiais e o *software* usado em projetos de sistemas embarcados.

7.1 Estrutura básica

A estrutura básica de um sistema de arquivos raiz do Linux segue a padronização hierárquica de sistemas de arquivos FHS[1] [40]. Na Tabela 7.1 são descritos os diretórios que devem compor a raiz do sistema de arquivos, definidos na especificação FSH 2.3.

A forma como os arquivos são organizados nas pastas segundo a especificação FHS, embora não mandatória, é importante para garantir a interoperabilidade entre o *software* existente para dispositivos embarcados em diferentes plataformas.

7.1.1 `sys`

O diretório `sys` não está descrito na especificação, mas por ser um componente importante, específico do Linux, é apresentado na tabela também. O sistema de arquivos virtual `sysfs`, montado no diretório `/sys`, foi incorporado ao Linux a partir do da versão 2.6 do núcleo do sistema. Ele permite que informações presentes no núcleo sejam exportadas ao espaço do usuário na forma de arquivos texto. Algumas informações presentes no `/proc` são duplicadas no `/sys`. A partir dos arquivos `dev` presentes no `/sys` é possível popular o diretório `/dev`. A raiz do `/sys` contém os diretórios apresentados na listagem a seguir:

[1] FHS - Filesystem Hierarchy Standard

Nome	Descrição
bin	Comandos binários essenciais
boot	Arquivos estáticos do *bootloader*
dev	Arquivos de dispositivos
etc	Arquivos de configuração do sistema
lib	Bibliotecas compartilhadas essenciais e módulos do núcleo
media	Pontos de montagem para mídias removíveis
mnt	Pontos de montagem para sistemas de arquivos temporários
opt	Aplicativos adicionais
sbin	Programas essenciais ao sistema
srv	Dados para serviços providos pelo sistema
tmp	Arquivos temporários
usr	Hierarquia secundária
var	Dados variáveis
home	Diretório raiz dos usuários
root	Diretório do usuário root (administrador do sistema)
proc	Sistema de arquivos virtual com informações acerca do estado do núcleo e dos processos no sistema
sys	Sistema de arquivos virtual que exporta informações a cerca dos dispositivos e *drivers* ao espaço do usuário

Tabela 7.1: Principais diretórios do sistema de arquivos raiz

```
jluisn@fenix:~$ tree -L 1 /sys/
/sys/
|-- block
|-- bus
|-- class
|-- devices
|-- firmware
|-- fs
|-- kernel
|-- module
`-- power
```

block

O diretório `block` contém subdiretórios para cada dispositivo de bloco que tenha sido descoberto no sistema. No diretório `/sys/block/sda`, por exemplo, que representa um disco rígido SATA[2] existe um arquivo `dev`, que contém a informação para a geração do nó `/dev/sda`, nesta pasta estão disponíveis informações de escalonamento, os diretórios das partições do disco e os eventos associados a este dispositivo de bloco.

bus

O diretório bus contém subdiretórios para cada tipo de barramento físico suportado no núcleo do Linux, seja de maneira estática ou através de módulos carregados em tempo de execução. São exemplos de barramentos suportados: ACPI, Bluetooth, I2C, IDE, IEEE1394, PCI, PCI Express, PNP, SCSI, Serial e USB.

Em cada diretório representante de um barramento existem dois subdiretórios: *devices* e *drivers*. O diretório `devices` apresenta uma lista com os apontadores para todos os dispositivos descobertos no sistema. O diretório `drivers` contém um diretório para cada dispositivo registrado e contém arquivos para a manipulação dos parâmetros do `driver`, além de apresentar apontadores para o dispositivo físico real.

class

O diretório `class` contém representações de cada classe de dispositivos que é registrada no núcleo.

devices

O diretório `devices` contém a hierarquia global de dispositivos. Ou seja, ele contém todos os dispositivos físicos que foram descobertos em cada barramento conectado ao sistema. Esta representação segue a forma das conexões físicas (elétricas).

[2]SATA - Serial ATA

`firmware`

O diretório *firmware* contém interfaces para visualização e manipulação de objetos e atributos específicos de *firmwares*. O *firmware*, neste caso, é caracterizado como sendo um código dependente de arquitetura, associado a algum driver no núcleo. É um artifício muito utilizado por empresas que não querem liberar informações a cerca dos seus produtos.

`fs`

Este diretório representa o componente do núcleo do Linux que permite a criação de sistemas de arquivos no espaço do usuário (FUSE[3]).

`kernel`

O diretório *kernel* contém informações de depuração e a cerca do compartilhamento de CPU definido na nova política de escalonamento CFS introduzida a partir da versão 2.6.23 do núcleo do Linux.

`module`

O diretório *module* contém um subdiretório para cada módulo do núcleo carregado no sistema. Os nomes dos subdiretórios são os nomes dos respectivos módulos. No diretório de cada módulo é possível verificar o estado inicial, a versão, o número de instâncias que o utilizam, e os endereços das seções do arquivo objeto carregado no núcleo do sistema.

power

Diretório contendo informações de gerenciamento de energia dos discos no sistema.

7.2 Bibliotecas C

Todos os sistemas baseados em Linux necessitam de uma biblioteca C. A biblioteca C provê funções que implementam as operações com arquivos (leitura, escrita), operações de gerenciamento de memória (alocação, liberação) e muitas outras operações que fazem do Linux um sistema completo. A maioria dos sistemas usa a biblioteca Glibc. A Glibc é uma biblioteca madura, bem testada e com um processo de desenvolvimento contínuo. Para lidar com as restrições impostas pelos sistemas embarcados, foram desenvolvidas novas bibliotecas C. Dentre as principais bibliotecas C usadas em sistemas embarcados, destacam-se a uClibc, a newlib, a diet libc, além da Glibc.

[3]FUSE - *Filesystem in Userspace*

7.2.1 Glibc

A biblioteca Glibc[4] é usada como a biblioteca C nos sistemas GNU e na maioria dos sistemas com o núcleo do Linux. Nela são disponibilizadas funções para acesso às chamadas de sistemas implementadas no núcleo do sistema operacional. A biblioteca GNU C segue os padrões ISO C e POSIX. Por ser desenvolvida para suportar internacionalização, ser portável e ter alto desempenho, os executáveis gerados por esta biblioteca acabam ocupando mais espaço físico do que os gerados pelas bibliotecas a seguir.

A geração de *toolchains* para as mais diversas arquiteturas usando a Glibc pode ser realizada usando as ferramentas Crosstool e OpenEmbedded. Algumas empresas especializadas em Linux embarcado provêem Glibc *toolchains* pré-compilados, este é o caso da Timesys (www.timesys.com), da MontaVista (www.mvista.com), e da CodeSourcery (www.codesourcery.com).

7.2.2 uClibc

A biblioteca uClibc originou-se do projeto uClinux, que é uma versão do Linux adaptada e otimizada para sistemas sem unidade de gerenciamento de memória (MMU[5]), tipicamente usada em microcontroladores. O desenvolvimento desta biblioteca tornou-se independente do uClinux, ganhando suporte em sistemas com MMU e com arquiteturas em ponto flutuante. A uClibc suporta todas as principais arquiteturas usadas em sistemas embarcados. A uClibc pode ser usada com ligação dinâmica ou estática em todas as arquiteturas.

Apesar de não tentar implementar de forma estrita as funcionalidades presentes na biblioteca Glibc (API), a uClibc provê a maioria de suas funções. A uClibc tenta seguir os padrões C89, o C99 da linguagem C e a especificação SUSv3 [1]. *Toolchains* da biblioteca uClibc compilados para as arquiteturas ARM, X86, MIPS, PowerPC e SH4, assim como o código fontes desta biblioteca, podem ser encontrados no sítio http://uclibc.org/. Estes *toolchains* são gerados usando a ferramenta Buildroot descrita anteriormente.

Na Tabela 7.2 apresenta-se uma comparação entre a Glibc e a uClibc com relação ao tamanho de dois aplicativos compilados com ligação dinâmica e estática. Os dois aplicativos compilados com *toolchains* das duas bibliotecas foram o *Hello world* e o Busybox.

7.2.3 Newlib

A newlib é uma biblioteca C voltada para o uso em sistemas embarcados. Ela é disponibilizada unicamente na forma de código fonte. Esta biblioteca pode ser compilada para uma

[4]Glibc - GNU C Library

[5]MMU - Memory Management Unit

	Glibc		uClibc	
	Estático	Dinâmico	Estático	Dinâmico
Hello world	502 KB	8,0 KB	19 KB	4,4 KB
Busybox	1,6 MB	744 KB	839 KB	732 KB

Tabela 7.2: Comparação entre a Glibc e a uClibc

faixa ampla de processadores. Para que ela funcione em um nova arquitetura é necessário que algumas rotinas de baixo nível sejam implementadas.

7.2.4 Diet libc

A biblioteca diet libc é uma biblioteca otimizada para tamanho reduzido. Ela pode ser usada para criar arquivos binários estaticamente ligados para Linux em diversas arquiteturas. Dentre as arquiteturas suportadas estão: alpha, arm, hppa, ia64, i386,mips, s390, sparc, sparc64, ppc e x86_64.

7.3 Núcleo do Linux

As imagens do núcleo do Linux podem ser armazenadas ou não no sistema de arquivos raiz. Esta dependência deve-se ao fato do *bootloader* saber ou não ler a partição em que o sistema se encontra. Na maioria dos sistemas embarcados, o núcleo do Linux é gravado em uma partição de memória FLASH separada, conhecida apenas pelo *bootloader*. Este é o caso dos *bootloaders* u-boot, redboot e do *bootloader* dos dispositivos Nokia Internet Tabblets. Nos sistemas que usam o Grub e o Lilo as imagens do núcleo do Linux são armazenas no diretório */boot*.

Os módulos do núcleo do sistema são encontrados no diretório */lib/modules/‘uname -r‘/*, onde o comando entre aspas representa a versão atual do núcleo do sistema.

7.4 Sistemas de arquivos especiais

Seguindo a tradição do UNIX, todos os objetos no Linux são representados através de arquivos. Esta regra é seguida até pelos dispositivos físicos conectados ao sistema, que podem ser acessados através de arquivos presentes no diretório `/dev` a partir de operações de leitura e escrita tradicionais, usando a biblioteca C presente no sistema.

Os dispositivos presentes no `/dev` são criados pela leitura dos arquivos **dev** criados pelo núcleo no `/sys`. Por exemplo, o dispositivo de bloco `sda`, cujo diretório no `/sys`

corresponde à entrada `/sys/block/sda`, tem um arquivo `dev` com o conteúdo `8:0`, que são os números principal e secundário do dispositivo. A entrada */dev/sda* é criada com o comando *mknod /dev/sda c 8 0*. Na Tabela 7.3 são descritos os principais dispositivos:

Nome	Descrição	Número principal	Número secundário
null	Dispositivo nulo	1	3
console	Console do sistema	5	1
zero	Fonte de zeros	1	5
mem	Acesso a memória física	1	1
random	Gerador de números aleatórios	1	8

Tabela 7.3: Principais dispositivos preserntes no /dev

Outro sistema de arquivos especial presente no Linux é o */proc*, que contém informações a cerca dos processos em execução no sistema. Estas informações são organizadas em diretórios rotulados com o identificador do processo. Entre as informações providas pelo */proc* estão as variáveis de ambiente, os endereços de memória virtual utilizados, a política de escalonamento utilizada e o status do processo. Além de descrever os processos do sistema, o proc também contém informações sobre os barramentos e drivers registrados no sistema.

7.5 Aplicativos de *software*

Além das bibliotecas C e do núcleo do Linux, para completar o sistema são necessários alguns aplicativos. A maioria dos aplicativos essenciais (linha de comando) pode ser obtida com a utilização do Busybox. O ambiente gráfico pode ser obtido usando o Matchbox, que provê um ambiente gráfico agradável e leve, desenvolvido para uso em sistemas embarcados.

7.5.1 Busybox

O Busybox combina os principais utilitários usados em sistemas UNIX em um único executável de tamanho reduzido. Ele provê alternativas para a maioria dos utilitários encontrados nas ferramentas GNU, como os utilitários para a manipulação de arquivos, versões leves de interpretadores de linha de comando, servidores web, etc. Os utilitários no Busybox geralmente têm menos opções do que o aplicativo o qual ele substitui, entretanto provêem a mesma funcionalidade. Na versão 1.11 o Busybox disponibiliza mais de 280 comandos conforme pode ser observado na Figura 7.1. Mesmo provendo uma quantidade

significativa de aplicativos, o tamanho do executável do busybox permanece em torno de 648K.

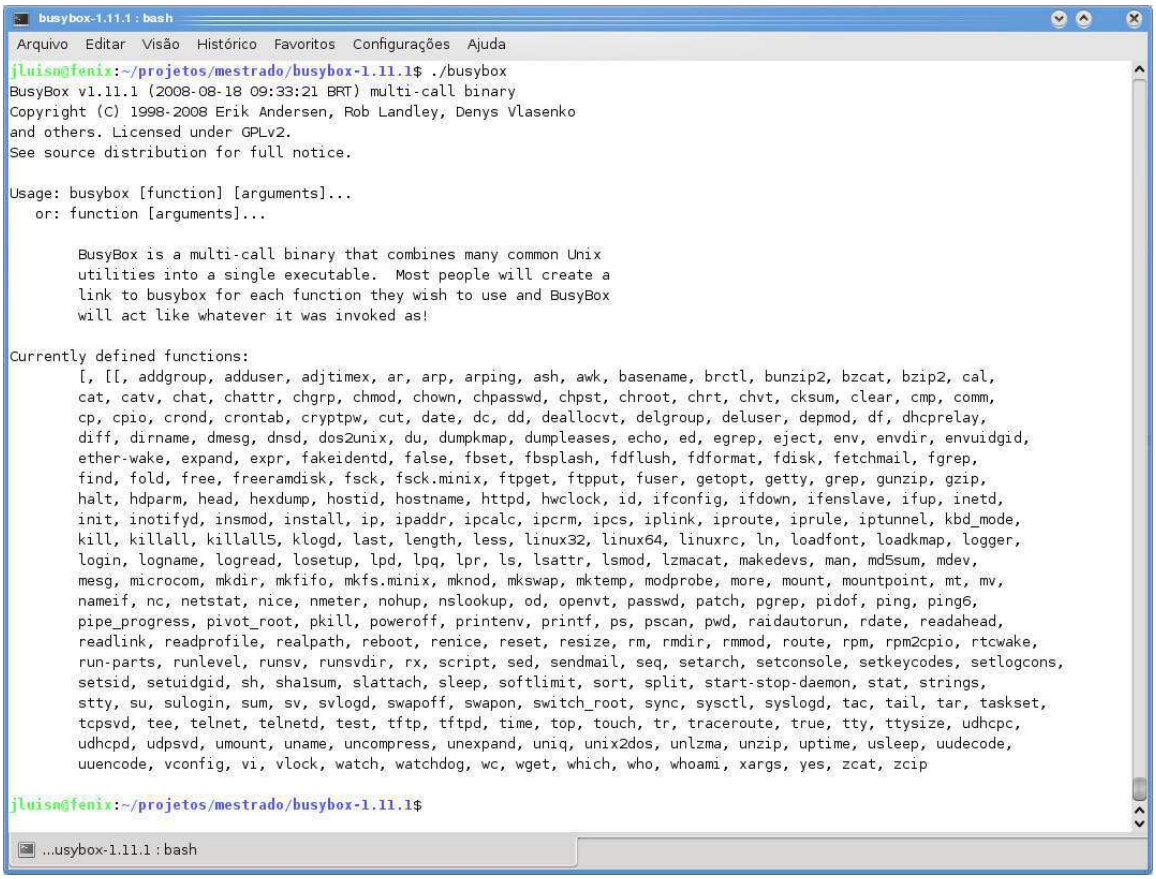

Figura 7.1: Funcionalidades providas pelo Busybox

O Busybox foi desenvolvido para ser utilizado em ambientes com restrições de armazenamento físico e para ser executado em ambientes com baixa capacidade de processamento. Ele é extremamente modular, de forma que é possível incluir ou excluir comandos em tempo de compilação. Isto permite um bom grau de personalização específico para as funcionalidades do sistema embarcado. Para que um sistema funcional fique pronto, usando o Busybox, basta adicionar alguns nós representativos dos dispositivos no */dev*, alguns arquivos de configuração no */etc* e o núcleo do Linux.

O Busybox é suportado nas principais arquiteturas usadas em sistemas embarcados. Seu código fonte pode ser obtido a partir do sítio www.busybox.net, um sistema completo pode ser criado usando a ferramenta Buildroot conforme descrito anteriormente.

7.5.2 Matchbox

O Matchbox, Figura 7.2, é um gerenciador de janelas que tem por objetivo maximizar a usabilidade em plataformas limitadas. Ele é projetado para ser pequeno em termos de

bytes utilizados, ter poucas dependências, usar uma quantidade mínima de recursos do sistema e ser extremamente flexível.

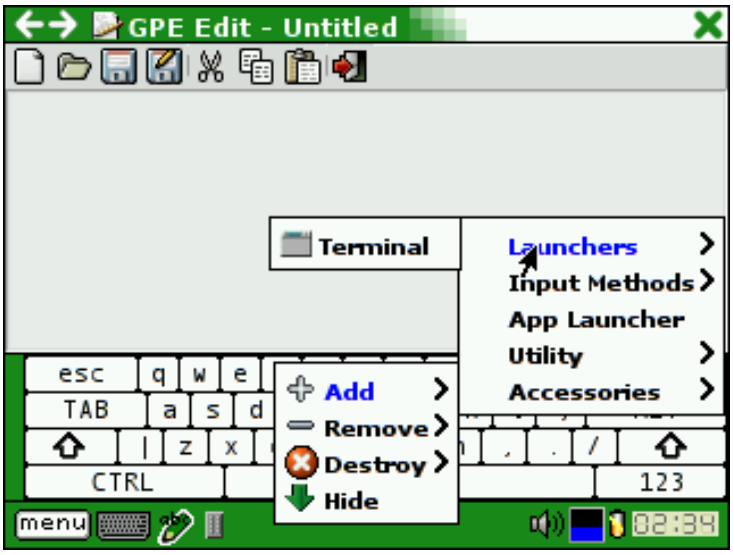

Figura 7.2: Tela gráfica do Matchbox

Apesar de suas limitações, o Matchbox apresenta algumas características únicas. Dentre as características únicas estão o suporte a temas baseados em arquivos XML, a possibilidade de utilização com o Tiny-X, que é um servidor de janelas para sistemas embarcados, o suporte a rotação da tela e a fontes *anti-aliased*.

O Matchbox pode ser encontrado em PDAs, Internet Tabblets, PCs ultra móveis e telefones celulares. O Matchbox é usado na plataforma maemo, no Ubuntu Mobile e no Openmoko. Um sistema de arquivos com o ambiente gráfico Macthbox pode ser construído de maneira fácil usando o OpenEmbedded ou o Buildroot.

7.6 Sumário

Neste capítulo foram apresentados os conceitos envolvidos na concepção de um sistema de arquivos raiz para um sistema embarcado. O uso de aplicativos como o Busybox e o Matchbox permitem a criação de um sistema de arquivos raiz enxuto em termos de espaço físico utilizado e em termos de funcionalidades providas. E como esses aplicativos são desenvolvidos com foco em sistemas embarcados, o consumo de recursos do sistema é minimizado.

Capítulo 8

Linux em tempo real

O objetivo de um sistema operacional em tempo real (STR) é criar um ambiente previsível e determinístico para a execução de tarefas. A proposta principal, portanto, não é aumentar a velocidade do sistema ou diminuir a latência entre uma ação e uma resposta, embora ambas aumentem a qualidade do STR. A proposta principal consiste na eliminação de atrasos aleatórios ilimitados, adicionando determinismo à execução das tarefas [42]. Um bom STR prioriza o desempenho na execução de tarefas, podendo diminuir a vazão de serviço para garantir que os prazos sejam cumpridos.

Os sistemas em tempo real podem ser divididos em duas classes principais: com restrições estritas de tempo (*hard real time*) e sistemas com restrições suaves de tempo (*soft real time*) para a execução de tarefas. Nos sistemas com restrições estritas de tempo, o fato de uma tarefa não ser executada dentro do prazo pode causar conseqüências catastróficas, como a queda de um avião, a explosão de uma caldeira, etc. Nos sistemas com restrições suaves, a execução da tarefa após o prazo pode acarretar em uma degradação da qualidade do serviço.

8.1 Linux em Tempo Real

As soluções de tempo real que usam Linux podem ser enquadradas em dois grupos principais: soluções dual núcleo, para as quais o núcleo do Linux é executado como uma *thread* de baixa prioridade sendo controlado por um executivo em tempo real, e soluções monolíticas para as quais modificações de tempo real são realizadas no núcleo do Linux permitindo que todas as aplicações do sistema sejam beneficiadas com as mudanças.

8.1.1 Soluções dual núcleo

Entre os projetos que utilizam esta solução dual, os principais são o RTLinux [39], o RTAI [32], o Xenomai [13], e o ADEOS [46]. Estes projetos são oferecidos a partir das

licenças de software livre com código aberto. O RTLinux, além de manter a versão livre, comercializa sua versão proprietária na qual são oferecidas mais funcionalidades, ferramentas de gerência de processos e APIs[1] de programação. O projeto RTAI surgiu do cerne livre do RTLinux quando este passou a comercializar sua versão fechada. O RTAI é utilizado no projeto EMC (*Enhanced Machine Control*) [12] e em inúmeros outros projetos de controle. A partir do RTAI aconteceram outras divisões resultando nos projetos ADEOS e Xenomai.

Nestes projetos implementa-se uma camada de abstração de *hardware*[2] para interceptar e gerenciar as interrupções, deste modo o núcleo pode sofrer retomada em todos os pontos de seu código. As regiões de código com interrupções desabilitadas são mais curtas, isto é, com menos instruções, devido a simplicidade e dimensão do HAL que foi projetado desde os primórdios para tratar deste problema. As latências de escalonamento nestes sistemas podem ser da ordem de 15 μs [16], conforme testes realizados, apresentados em [6], com plataformas VIA C3 de 600 MHz.

Os programas em tempo real são desenvolvidos como módulos do sistema, sendo executados no espaço do núcleo. Ao se executar uma aplicação no espaço do núcleo algumas precauções extras devem ser tomadas, uma vez que, não há proteção na alocação de memória. Erros de programação cometidos nas aplicações podem travar o sistema, não apenas a aplicação em tempo real. Por serem executados no espaço do núcleo, estes programas são bem mais difíceis de serem depurados.

A principal desvantagem destes sistemas é que as APIs de programação são mais restritas em comparação ao grande número de bibliotecas GNU/Linux existentes que poderiam ser utilizadas.

8.1.2 Soluções monolíticas

Dentre os projetos nos quais o núcleo do Linux é modificado, o principal é denominado *RT-Preempt* cujo principal objetivo é fazer com que o Linux tenha comportamento determinístico apresentando baixas latências de escalonamento. Este projeto é suportado por empresas como Monta Vista[3], Red Hat[4] e TimeSys[5], que oferecem além de binários compilados, suporte e ferramentas para manutenção dos sistemas. Estas empresas pagam desenvolvedores para que estes contribuam com código aberto para o núcleo do Linux.

A utilização deste tipo de alternativa permite que as aplicações em tempo real sejam desenvolvidas no espaço do usuário, ou seja, não é necessária a inserção e remoção

[1] API - Interface de Programação de Aplicação
[2] HAL - *Hardware Abstraction Layer*
[3] http://www.mvista.com/
[4] http://www.timesys.com/
[5] http://www.redhat.com/

de módulos no núcleo do STR. A API de programação segue a especificação POSIX 1003.1b [41], a qual define as políticas de escalonamento SCHED_FIFO, SCHED_RR e SCHED_OTHER. Usando a política SCHED_FIFO os processos são escalonados de acordo com a ordem de chegada, para cada nível de prioridade existe uma fila respectiva. Usando SCHED_RR os processos são escalonados de maneira *Round Robin*. A política SCHED_OTHER não é definida na especificação, de modo que em geral ela implementa a política de escalonamento padrão Unix/Linux, a qual procura distribuir de forma eqüitativa ou justa o acesso aos recursos do sistema.

Outra solução que se enquadra nesta categoria foi desenvolvida por pesquisadores da universidade da Carolina do Norte – EUA, o LITMUS^{RT} [10]. O LITMUS^{RT} modifica o núcleo do Linux, fornecendo uma arquitetura baseada em *plugins* que permite a adição de novas políticas de escalonamento. A utilização das políticas de escalonamento definidas a partir do LITMUS^{RT} é realizada a partir de uma API no espaço do usuário que mapeia funções em chamadas de sistema no núcleo. Ao contrário do *RT-Preempt* que suporta as versões mais novas do núcleo e tem desenvolvimento em paralelo ao mesmo, o LITMUS^{RT} suporta apenas algumas versões específicas do núcleo.

8.1.3 Projeto RT-Preempt

As implementações anteriores que introduziam comportamento em tempo real ao Linux não foram aceitas na linha principal do núcleo por dois motivos principais: ou tinham um desenvolvimento completamente independente; ou eram direcionadas apenas ao nicho de tempo real. Várias mudanças intrusivas eram realizadas no núcleo de modo que não eram aceitas pela maioria dos desenvolvedores do núcleo. Para que as mudanças pudessem ser incorporadas ao núcleo do Linux, Ingo Molnar[6] iniciou um pequeno projeto, o RT-Preempt[7], para incorporar os métodos de tempo real a linha principal do núcleo de forma gradual.

No contexto do projeto RT-Preempt o objetivo é garantir um comportamento determinístico para a execução de processos em tempo real no núcleo do Linux e torná-lo completamente preemptível. Para que isso seja possível uma série de modificações são definidas para o núcleo do Linux. As principais modificações são as seguintes:

- Os *spinlocks* que implementam espera ocupada são convertidos em mutexes: as seções críticas protegidas por *spinlocks* podem sofrer preempção;
- As rotinas de tratamento de interrupções são convertidas em processos do núcleo:

[6] http://people.redhat.com/mingo/
[7] http://rt.wiki.kernel.org

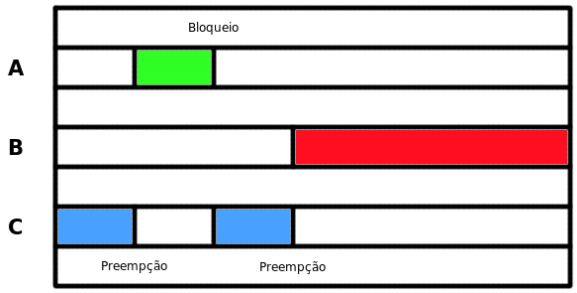

Figura 8.1: Inversão de prioridades

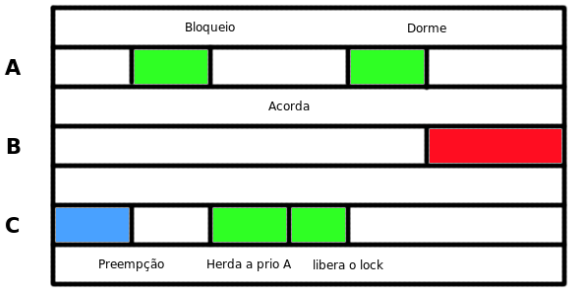

Figura 8.2: Com herança de prioridades

cada interrupção por *software* tem uma prioridade associada e processos com prioridades mais elevadas podem executar sem sofrerem preempção;

- Implementação do algoritmo herança de prioridades: impede que processos de menor prioridade que tenham acesso a um recurso compartilhado bloqueiem indefinidamente processos de maior prioridade. Por exemplo, conforme ilustrado na Figura 8.1, considere o uso de três *threads* A (maior prioridade), B (prioridade média) e C (menor prioridade), C inicia e adquire alguns *locks*, A acorda e preempta C, quando A tenta acessar um recurso mantido por C, A é bloqueado, C continua, mas sofre preempção por B antes de liberar o recurso que A necessita. A solução é mostrada na Figura 8.2, onde a *thread* C tem sua prioridade elevada a prioridade de A, enquanto mantiver o acesso ao recursos compartilhados, deste modo não sofre preempção por B;

- Conversão da antiga API de temporizadores em uma infraestrutura separada que inclui os temporizadores de alta resolução e *timeouts*, permitindo que os usuários acessem temporizadores de alta resolução a partir da API POSIX.

Com o uso do RT-Preempt é possível a obtenção de *jitters* da ordem de μs como os apresentados nos projetos RTAI e RTLinux [6]. Apesar disso, algumas seções críticas longas com interrupções desabilitadas permanecem e o Linux ainda não pode ser usado no contexto de sistemas em tempo real com comportamento estrito [6] usando o RT-Preempt. O seu uso para sistemas em tempo real com restrições de tempo suave é viável.

8.1.4 LITMUS^{RT}

O LITMUS^{RT} é uma plataforma de testes para a avaliação de algoritmos de escalonamento em tempo real com suporte a multiprocessadores [8]. O LITMUS^{RT} apresenta uma arquitetura baseada em *plugins* que permite a adição de novas políticas de escalonamento

ao núcleo do Linux. Na sua versão atual, o LITMUS^{RT} utiliza a versão 2.6.24 do núcleo do Linux. O LITMUS^{RT} provê uma API no espaço do usuário que possibilita o acesso aos aplicativos no espaço do usuário o uso das políticas de escalonamento acopladas ao núcleo do sistema operacional.

Por ser um ambiente de testes para algoritmos de escalonamento em plataformas com múltiplos processadores, o LITMUS^{RT} é suportado apenas nas arquiteturas X86 e Sparc64. A adição do suporte a uma nova arquitetura pode ser obtido com a implementação das chamadas de sistema específicas da arquitetura.

Dentre os algoritmos de escalonamento implementados pelo LITMUS^{RT} estão as variações do EDF: P-EDF (EDF particionado), G-EDF (EDF global) e NG-EDF (EDF global sem retomada). Conforme explicado anteriormente, o algoritmo EDF escalona os processos de acordo com os seus respectivos prazos, quanto menor o prazo maior a possibilidade de o processo ser escalonando. No algoritmo P-EDF as tarefas são atribuídas de forma estática a um determinado processador, que as escalona na forma EDF tradicional. No algoritmo NG-EDF as tarefas podem migrar entre os processadores, mas uma vez que ela começar a executar em um processador ela executa até seu término. O G-EDF permite que as tarefas migrem entre os processadores e que sofram retomada.

Na versão anterior do LITMUS^{RT}, desenvolvida usando a versão 2.6.20 do núcleo do Linux, foi desenvolvido o algoritmo A-GEDF (G-EDF adaptativo) no qual o escalonamento de tarefas

Ao contrário do EDF em um único processador, nenhuma das variações com múltiplos processadores do EDF é ótima. Ou seja, algumas tarefas podem perder prazo mesmo em ambientes em que a utilização total não supere a capacidade de processamento.

8.2 Benchmarks

Para determinar o efeito das modificações no núcleo alguns *benchmarks* conhecidos podem ser utilizados. Entre os principais estão o *interbench* [22], *lmbench* [33], *hackbench* [15], e as ferramentas de depuração presentes no núcleo com o *patch* de tempo real. A seguir são apresentados alguns detalhes dos *benchmarks* citados.

Interbench

Trata-se de uma aplicação para testar a interatividade dos processos no Linux. Emula o escalonamento de tarefas interativas e afere a latência de escalonamento e o *jitter*. Com este *benchmark* é possível emular tarefas como a carga gerada pelo servidor de janelas X, por processamento de áudio, de vídeo, leituras e escritas no disco, compilação e alocação de memória RAM. Alguns destes testes são descritos a seguir:

- Gerenciador de Janelas X: o gerenciador de janelas X é simulado como uma tarefa que utiliza a CPU de 0 a 100 %. Deste modo, pode-se simular quando uma janela gráfica inicialmente ociosa é arrastada pela tela e solta.
- Áudio: o áudio é simulado como uma tarefa que tenta executar a cada intervalo de 50 *ms* e requerendo 5 % de CPU. Esta tarefa é executada em tempo real seguindo a política SCHED_FIFO.
- Vídeo: o vídeo é simulado como uma tarefa que usa 40 % de CPU 60 vezes por segundo (simulando a execução de um vídeo a 60 quadros por segundo). Esta tarefa também usa a política de escalonamento SCHED_FIFO.

Lmbench

O *lmbench* consiste de um conjunto de *benchmarks* compatível com as especificações POSIX para a análise de latências e vazão. Em relação a vazão, é possível aferir a velocidade de leitura de arquivos em *cache*, a velocidade de leitura/escrita/cópia de áreas de memória e a vazão de fluxos de dados TCP. As latências do sistema que podem ser aferidas são relativas ao chaveamento de contexto, ao estabelecimento de conexões de rede, a criação e o apagamento de arquivos, a criação de processos, a manipulação de sinais, o uso de chamadas de sistema e a leitura da memória.

Hackbench

O *Hackbench* é uma ferramenta desenvolvida para a aferição de desempenho, sobrecarga e escala do escalonador do Linux. É baseado em uma arquitetura cliente/servidor em que os pares se comunicam através de conexões estabelecidas via *sockets*.

Ferramentas de depuração do núcleo

As ferramentas de depuração do núcleo permitem que sejam traçados histogramas a partir da aferição das latências no escalonamento de processos. Estas opções contribuem para o aumento das latências do sistema. As medições importantes para a caracterização do Linux, em relação a sua utilização como sistema em tempo real, são obtidas através dos dados apresentados na seção *Kernel hacking* do núcleo, e são as seguintes:

- *Non-preemptible critical section latency timing*;
- *Interrupts-off critical section latency timing*;
- *Wakeup latency histogram*;

- *Non-preemptible critical section latency histogram.*

Estas informações de escalonamento são disponibilizadas através do sistema de arquivos virtual */proc* na forma de arquivos.

8.3 O Escalonador do Linux

A execução de processos no Linux segue um diagrama de transição com seis estados para o escalonador de curto prazo, conforme mostrado na Figura 8.3. O escalonador de curto prazo é o que escalona os processos residentes na memória principal do sistema. Inicialmente, o processo no estado **Novo** é admitido no sistema indo para o estado de **Pronto**, em seguida o processo pode ser escalonado indo para o estado **Executando**, neste estado o processo pode ser interrompido e voltar para o estado de Pronto ou ser suspenso por outro processo (**Parado**) ou ainda aguardar por operações de entrada e saída (**Suspenso**) finalmente o processo termina e entra no estado *Zumbi*, aguardando pelo término dos processos filhos.

Figura 8.3: Diagrama de estados de um processo

A operação de escalonamento, ou seja, quando um processo sai do estado de Pronto para Executando segue uma estrutura de filas multinível divididas por prioridade conforme apresentado na Figura 8.4. Existem 141 níveis de prioridades, sendo que as maiores prioridades (menor número) são atribuídas aos processos em tempo real e as demais aos processos de propósito geral. Ao contrário da maioria dos sistemas operacionais, o Linux atribui uma fatia de tempo maior para as tarefas em tempo real, ver Figura 8.5. Para cada nível de prioridade são definidos dois *arrays* de tarefas: o primeiro com as tarefas ativas e o segundo com as tarefas expiradas.

As tarefas são ordenadas em cada fila de acordo com a política de escalonamento adotada por cada processo. O chaveamento das filas ocorre quando todos os processos de cada fila expirarem seus tempos de execução. Esta operação é realizada pela troca dos

Figura 8.4: Divisão das prioridades no Linux

apontadores das duas filas. As tarefas fora da faixa reservada para execução em tempo real apresentam prioridades dinâmicas, que dependem do seu grau de iteratividade. As tarefas mais iterativas têm suas prioridades aumentadas, enquanto que as tarefas em *background* podem sofrer uma penalidade de prioridade, podendo ser rebaixadas de fila. As tarefas iterativas são limitadas por operações de entrada e saída.

Figura 8.5: Filas de tarefas multiníveis

8.4 Sumário

Neste capítulo apresentou-se as soluções para escalonamento de processos em tempo real no núcleo do Linux. Embora atualmente, as soluções dual núcleo ofereçam melhores resultados em relação a previsibilidade na execução das tarefas em tempo real, as soluções monolíticas são mais promissoras no sentido de manutenção e comunidade de suporte. Neste sentido foi observado que o projeto *RT-Preempt* modifica o núcleo do Linux com o objetivo de eliminar longas seções críticas com interrupções desabilitadas, permitindo que processos sofram retomada em quase todos os pontos do núcleo. Ainda neste contexto observou-se que a arquitetura baseada em *plugins* do LITMUS^{RT} facilita o desenvolvimento de uma nova política de escalonamento de tarefas em tempo real no núcleo do Linux.

Capítulo 9

Teoria de controle aplicada aos sistemas em tempo real

O uso da teoria de controle com realimentação permite que técnicas amplamente conhecidas para a sintonia de plantas industriais possam ser aplicadas aos sistemas computacionais de modo que, por exemplo, as técnicas de escalonamento possam ser validadas através de esquemas de alocação de pólos. Com o uso da teoria de controle é possível, por exemplo, desenvolver um escalonador capaz de prover estabilidade ao sistema, que tenha um determinado tempo de resposta e que apresente uma resposta transitória característica.

Em se tratando do escalonamento de processos através de um controlador com realimentação, deve-se levar em consideração que as operações realizadas na tomada das decisões de controle devem ser simples e rápidas. Desta forma, algoritmos estocásticos não deverão ser considerados a priori. A tarefa de escalonamento pode resultar em modelos não lineares, os quais deverão ser simplificados para que a decisão de escalonamento não sobrecarregue o sistema.

Um dos algoritmos mais utilizados em sistemas em tempo real para escalonamento dinâmico de tarefas é o EDF. No EDF as tarefas de menor prazo são escalonadas primeiro. O fato deste algoritmo ser em malha aberta faz com que, em ambientes onde o pior caso de execução não possa ser determinado, haja uma grande degradação no desempenho do sistema. Para que o sistema torne-se robusto a tal tipo de degradação, foram propostas variações deste algoritmo que utilizam realimentação baseada na teoria de controle [7,30].

A abordagem adotada em [30] define um arcabouço para escalonamento de processos em tempo real usando a teoria de controle. São definidos controladores PI[1] para ajustar os níveis de serviço das tarefas no sistema com o objetivo de maximizar o uso de CPU e minimizar a quantidade de tarefas que perdem prazo. O laço de controle definido nesta

[1] PI – Proporcional Integral

abordagem utiliza um Monitor, que mede as variáveis controladas, um Controlador, que compara as variáveis controladas com as referências e calcula o novo valor das variáveis e um Atuador para ajuste dos níveis de serviço das tarefas.

A abordagem adotada em [7] estende o trabalho anterior [30] permitindo a monitoração dos tempos de execuções individuais de cada tarefa. Com isso é possível modificar os níveis de serviço de uma única tarefa, ao invés de modificar os níveis de serviço de todas as tarefas no sistema. Esta abordagem utiliza o LITMUS^{RT} e é implementada no núcleo do Linux através de um *patch* obtido no sítio eletrônico do projeto[2].

9.1 Diagrama de blocos

O diagrama de blocos do sistema de controle definido em [7] é ilustrado na Figura 9.1.

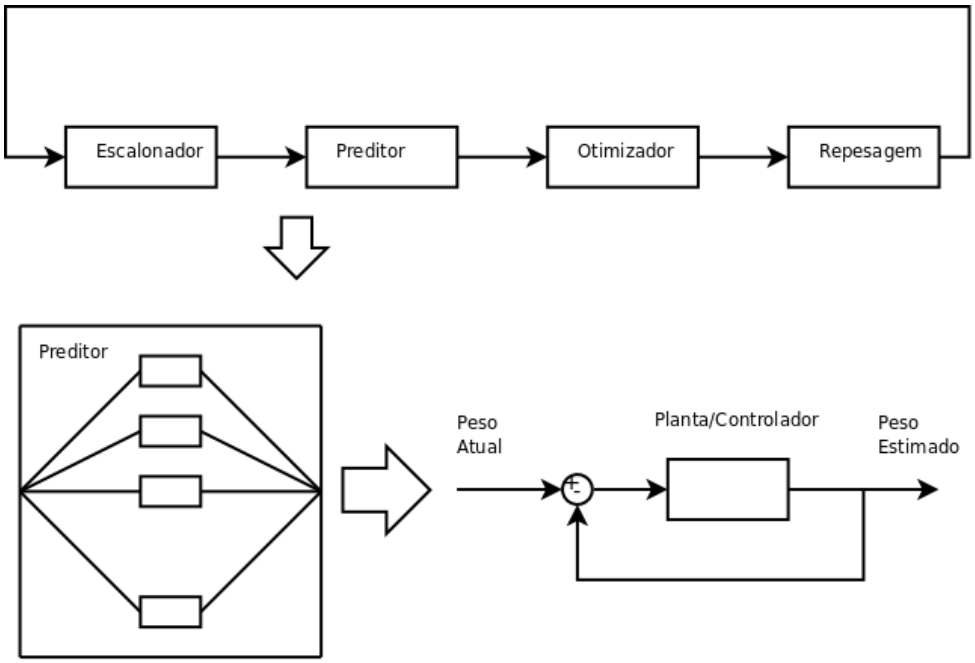

Figura 9.1: Malha de Controle

Os componentes do sistema são o Escalonador, o Preditor, o Otimizador e uma função de Repesagem das tarefas. O Escalonador escalona as tarefas de acordo com a política de escalonamento EDF a partir dos pesos das mesmas. O Preditor usa realimentação para fazer uma estimativa dos pesos de cada tarefa a partir de um controle proporcional integral. O Otimizador usa os pesos estimados das tarefas para estabelecer seus níveis de serviço. A função de repesagem é usada quando uma tarefa muda sua importância no sistema (peso), e é responsável por modificar o segmento de código que a tarefa executará.

[2]http://www.cs.unc.edu/ anderson/litmus-rt/

9.2 Sumário

Neste capítulo foram mostradas as duas principais soluções que usam teoria de controle para escalonamento de processos em tempo real. A solução implementada usando o LITMUS^{RT} é mais interessante do ponto de vista que ela estende a solução de Lu e outros [30], tem código fonte disponível, usa o Linux e é baseada em *plugins*.

Capítulo 10

Política de escalonamento

Nos capítulos anteriores, discutiu-se a cerca das principais características dos sistemas embarcados e em particular dos que utilizam o Linux como sistema operacional. Foram observados os principais conceitos envolvidos no desenvolvimento de aplicações para Linux embarcado, desde a configuração do ambiente de desenvolvimento, a geração das ferramentas de desenvolvimento a partir do código fonte, a configuração do *bootloader*, a criação de um sistema de arquivos raiz, a obtenção e configuração do núcleo e dos principais softwares para a plataforma alvo.

Durante o texto foram mostrados os principais motivos pelos quais o Linux é uma das principais opções no projeto de sistemas embarcados de médio porte. Não obstante a realidade, também foram apresentados alguns desafios que impedem uma aceitação maior do Linux no mercado de sistemas embarcados. Observou-se que um dos principais pontos fracos do Linux é a ausência de uma política de escalonamento de tarefas em tempo real, integrada ao seu núcleo, que garanta que processos em tempo real com restrições de tempo suave sejam escalonados com sucesso.

Este motivo é um dos que impedem que o Linux seja mais utilizado em aplicações de controle e automação. Para solucionar este problema, alternativas como o LITMUS^{RT} e o RT-Preempt podem ser utilizadas em conjunto, pois algumas das causas de latências e atrasos ilimitados que afetam o comportamento do LITMUS^{RT} são tratadas com o *patch* de tempo real disponibilizado pelo RT-Preempt.

Verificou-se também que a principal arquitetura usada em sistemas embarcados é a ARM, que atualmente não é suportada pelo LITMUS^{RT}.

O objetivo deste trabalho, portanto é integrar as soluções providas pelo LITMUS^{RT} e pelo RT-Preempt na plataforma ARM em um ambiente ou arcabouço que permita o desenvolvimento de aplicações de automação e controle. A plataforma ARM OMAP 1610 foi escolhida por prover uma série de dispositivos de entrada e saída e ser voltada para uso em ambientes de desenvolvimento.

10.1 Desafios

Para viabilizar este trabalho alguns desafios e problemas tiveram que ser solucionados, os principais são destacados a seguir:

- A ausência de uma distribuição mínima e personalizável de Linux embarcado para a plataforma ARM;
- O algoritmo EDF adaptativo do LITMUS^{RT} é baseado na versão 2.6.20 do núcleo do Linux, que utilizava *jiffies*[1] na temporização, enquanto que na versão 2.6.24 são usados temporizadores de alta resolução;
- A implementação das chamadas de sistema para a plataforma ARM;
- A falta de suporte a tecnologia SMP na plataforma ARM-OMAP, pois o LITMUS^{RT} tem foco principal em arquiteturas com múltiplos processadores;
- A integração com o projeto RT-Preempt;
- A elaboração dos testes para a verificação da solução obtida.

10.2 Ambiente operacional

O ambiente operacional para a realização dos testes foi desenvolvido com base na plataforma OMAP 1610 H2 da *Texas Instruments* que utiliza um processador ARM 926T de 192 MHz e tem 32 MB de memória RAM. O sistema de arquivos raiz foi criado usando a ferramenta Buildroot, conforme descrito no Capítulo 4. A escolha dos aplicativos de *software* para o sistema considerou os aspectos de desenvolvimento para sistemas embarcados, ou seja, aplicações com funcionalidades reduzidas e que utilizam poucos recursos do sistema, sem contudo deixarem de prover as características desejadas ao sistema.

10.3 Implementação

Conforme descrito anteriormente, o LITMUS^{RT} apresenta uma arquitetura modular baseada em *plugins* que permite a adição de novas políticas de escalonamento para as tarefas em tempo real.

As políticas de escalonamento implementadas no Linux usando o LITMUS^{RT} seguem uma estrutura padrão definida pela estrutura de programação observada a seguir:

[1] *jiffies* - Unidade de tempo definida a partir do número de ciclos de relógio a partir da inicialização do sistema

```
static struct sched_plugin fc_edf_plugin = {
    .plugin_name     = "FC-EDF",
    .finish_switch   = fcedf_finish_switch,
    .tick            = fcedf_tick,
    .task_new        = fcedf_task_new,
    .complete_job    = complete_job,
    .task_exit       = fcedf_task_exit,
    .schedule        = fcedf_schedule,
    .task_wake_up    = fcedf_task_wake_up,
    .task_block      = fcedf_task_block,
    .admit_task      = fcedf_admit_task
};
```

A função desta estrutura é mapear as ações de escalonamento à suas respectivas funções. Além destas funções definidas na estrutura `sched_plugin` são definidas outras funções que são invocadas pelas funções definidas na estrutura citada. A lista completa das funções usadas no *plugin fc_ edf_ plugin* que define a política de escalonamento FC-EDF é detalhada a seguir:

- fcedf_finish_switch: Função que move uma tarefa de uma posição para outra;
- fcedf_tick: Função que é invocada a cada interrupção de relógio;
- fcedf_task_new: Função que prepara uma tarefa para ser executada em tempo real;
- complete_job: Função invocada quando a tarefa completa;
- fcedf_task_exit: Função invocada quando uma tarefa é completada;
- fcedf_schedule: Função principal, chamada a ação de escalonamento;
- fcedf_task_wake_up: Função invocada quando uma tarefa acorda;
- fcedf_task_block: Função responsável por bloquear tarefas;
- fcedf_admit_task: Função responsável por admitir uma nova tarefa no sistema;
- fcedf_optimize: Função responsável pelas ações de otimização no sistema;
- change_weight: Função responsável por alterar os pesos das tarefas;
- set_service_level: Função que altera os níveis de serviço das tarefas.

A política de escalonamento FC-EDF foi implementada a partir da política ADAPTIVE-EDF definida no LITMUS^{RT} usando a versão 2.6.20 do núcleo do Linux. Tal política foi adaptada para um sistema com um único processador e para a versão 2.6.24 do núcleo do Linux. A melhoria do desempenho em tempo real do sistema foi obtida com a aplicação

do *patch* de tempo real disponibilizado pelo RT-Preempt ao núcleo. Desta forma foram tratados os problemas de escalonamento de processos em tempo real usando teoria de controle com realimentação, controle Proporcional-Integral, além das longas seções críticas e das inversões de prioridades presentes no núcleo do Linux.

10.4 Sumário

Neste capítulo foram apresentadas as escolhas tomadas em relação a solução proposta e a implementação da mesma. Foram consideradas as restrições do ambiente alvo, a plataforma embarcada, na configuração das soluções de *software* embarcado para a plataforma. Para cumprir os objetivos do trabalho foram integradas várias soluções abertas, entre elas o RT-Preempt, o LITMUS^{RT}, o núcleo do Linux e a ferramenta Buildroot entre as mais importantes.

Capítulo 11

Considerações Finais

A partir deste trabalho tornou-se possível a concepção de um ambiente integrado para sistemas embarcados, usando a teoria de controle para o escalonamento de processos em tempo real. A seguir são elencadas as principais contribuições advindas deste trabalho, bem como os desdobramentos futuros que o mesmo possa vir a ter.

11.1 Conclusão

Dentre as principais contribuições deixadas por este trabalho, destacam-se as seguintes:

- A apresentação dos conceitos envolvidos na escolha e configuração de sistemas embarcados, e em particular de Linux embarcado;
- A apresentação dos passos para a configuração de um ambiente de desenvolvimento funcional para Linux em uma plataforma embarcada;
- A apresentação das ferramentas de desenvolvimento;
- A criação de sistema de arquivos raiz mínimos para uso em aplicações embarcadas;
- A adição de novas chamadas de sistema ao núcleo do Linux;
- A avaliação empírica de um algoritmo de escalonamento usando a teoria de controle com realimentação em uma plataforma embarcada;
- A avaliação do impacto em relação à eliminação das longas seções críticas com interrupções desabilitadas no núcleo do Linux, através do uso do projeto RT-Preempt.
- A integração de soluções abertas propiciou a verificação por parte do autor, a grande importância de se prover boas soluções abertas.

11.2 Trabalhos futuros

Em relação as extensões deste trabalho pode-se vislumbrar as seguintes como principais:

- O uso da plataforma OMAP para controle de uma planta industrial com restrições de tempo real;
- A aplicação de outras técnicas de controle diferentes de controle no escalonamento de processos;
- O desenvolvimento de uma política de escalonamento em tempo real voltada para a conservação de energia do sistema;
- A comparação com as outras alternativas de tempo real introduzidas no núcleo do Linux.

Referências Bibliográficas

[1] Standard for information technology - portable operating system interface (posix). shell and utilities. Technical report, 2004. http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=1309816.

[2] Mips technologies, Acessado em Fevereiro de 2008. http://www.mips.com/.

[3] Power architecture, Acessado em Janeiro de 2008. http://www.power.org.

[4] M. Amirijoo, J. Hansson, S. H. Son, and S. Gunnarsson. Experimental evaluation of linear time-invariant models for feedback performance control in real-time systems. *Real-Time Syst.*, 35(3):209–238, 2007.

[5] Erik Andersen. Buildroot, Acessado em Janeiro de 2008. http://buildroot.uclibc.org/.

[6] Siro Arthur, Carsten Emde, and Nicholas Mc Guire. Assessment of the realtime preemption patches (rt-preempt) and their impact on the general purpose performance of the system. *RTL Workshop*, (9), Novembro 2007.

[7] Aaron Block, Björn Brandenburg, James H. Anderson, and Stephen Quint. An adaptive framework for multiprocessor real-time system. In *ECRTS '08: Proceedings of the 2008 Euromicro Conference on Real-Time Systems*, pages 23–33, Washington, DC, USA, 2008. IEEE Computer Society.

[8] Björn B. Brandenburg, Aaron D. Block, John M. Calandrino, UmaMaheswari Devi, Hennadiy Leontyev, and James H. Anderson. Litmusît: A status report. *Proceedings of the 9th Real-Time Linux Workshop*, Novembro 2007.

[9] S. Brosky and S. Rotolo. Shielded processors: guaranteeing sub-millisecond response in standard linux. *Parallel and Distributed Processing Symposium, 2003. Proceedings. International*, pages 9 pp.–, 22-26 April 2003.

[10] John M. Calandrino, Hennadiy Leontyev, Aaron Block, UmaMaheswari C. Devi, and James H. Anderson. Litmusît: A testbed for empirically comparing real-time

multiprocessor schedulers. In *RTSS '06: Proceedings of the 27th IEEE International Real-Time Systems Symposium*, pages 111–126, Washington, DC, USA, 2006. IEEE Computer Society.

[11] Arc Chart. Mobile operating systems: The new generation. Technical report, 2006.

[12] Linux CNC Community. Enhanced machine control project, Acessado em Dezembro de 2007. http://www.linuxcnc.org/.

[13] Xenomai Community. Xenomai: Real-time framework for linux, Acessado em Dezembro de 2007. http://www.xenomai.org.

[14] Intel Corporation. Intel architecture software developer's manual, Acessado em Janeiro de 1999. http://download.intel.com/design/PentiumII/manuals/24319002.PDF.

[15] Craiger. Hackbench, Acessado em Dezembro de 2007. http://devresources.linux-foundation.org/craiger/hackbench/.

[16] Kevin Dankwardt. Comparing real-time linux alternatives, Acessado em Dezembro de 2000. http://www.linuxdevices.com/articles/AT4503827066.html.

[17] Linux Devices. The linux devices showcase, Acessado em Janeiro de 2008. http://linuxdevices.com/articles/AT4936596231.html.

[18] Johan Eker, Per Hagander, and Karl-Erik Årzén. A feedback scheduler for real-time controller tasks. *Control Engineering Practice*, 8(12):1369–1378, Janeiro 2000.

[19] Free Software Foundation. Gnu bash, Acessado em Agosto de 2008. http://www.gnu.org/software/bash/.

[20] INdT. Mamona, Acessado em Janeiro de 2008. http://dev.openbossa.org/trac/mamona/.

[21] Dan Kegel. Crosstool, Acessado em Janeiro de 2008. http://kegel.com/crosstool/.

[22] Con Kolivas. The linux interactivity benchmark, Acessado em Dezembro de 2007. http://members.optusnet.com.au/ckolivas/interbench/.

[23] Chris Lanfear. Open source in the embedded market: Linux and much more, 2006. http://electronicdesign.com/Articles/Index.cfm?AD=1&AD=1&ArticleID=11733.

[24] LinuxDevices.com. Snapshot of the embedded linux market, Acessado em Abril de 2007. http://linuxdevices.com/articles/AT7065740528.html.

[25] C. L. Liu and James W. Layland. Scheduling algorithms for multiprogramming in a hard-real-time environment. *Journal of the ACM*, 20(1):46–61, 1973. citeseer.ist.psu.edu/liu73scheduling.html.

[26] Robert Love. *Linux Kernel Development (2nd Edition) (Novell Press)*. Novell Press, 2005.

[27] OpenedHand Ltd. Poky platform builder, Acessado em Janeiro de 2008. http://www.pokylinux.org.

[28] ARM Holdings LTDA. Arm - product backgrounder, Acessado em Janeiro de 2005. http://www.arm.com/miscPDFs/3823.pdf.

[29] Chenyang Lu, John A. Stankovic, Gang Tao, and Sang H. Son. Design and evaluation of a feedback control edf scheduling algorithm. *Proceedings of Real-Time Systems Symposium*, 1999.

[30] Chenyang Lu, John A. Stankovic, Gang Tao, and Sang H. Son. Feedback control real-time scheduling: Framework, modeling, and algoritms. *Journal of Real Time Systems*, 2001.

[31] maemo Community. maemo white paper, Acessado em Agosto de 2008. http://maemo.org/intro/white_paper.html.

[32] Paolo Mantegazza. Realtime application interface, Acessado em Dezembro de 2007. https://www.rtai.org/.

[33] Larry McVoy and Carl Staelin. Lmbench, Acessado em Dezembro de 2007. http://www.bitmover.com/lmbench/.

[34] Ingo Molnar. The completely fair scheduler, Acessado em Dezembro de 2007. http://kerneltrap.org/node/8059.

[35] Ingo Molnar. Realtime preemption patch, Acessado em Dezembro de 2007. http://rt.wiki.kernel.org.

[36] OpenEmbedded. Openembedded wiki, Acessado em Janeiro de 2008. http://www.openembedded.org/.

[37] Openmoko. Openmoko wiki, Acessado em Janeiro de 2008. http://www.openmoko.org/.

[38] OpenWrt. Openwrt community, Acessado em Janeiro de 2008. http://openwrt.org/.

[39] Wind River. Rtlinux, Acessado em Dezembro de 2007. http://www.rtlinuxfree.com/.

[40] Rusty Russell, Daniel Quinlan, and Christopher Yeoh. Filesystem hierarchy standard, Acessado em Agosto de 2008. http://www.pathname.com/fhs/.

[41] IEEE Computer Society. Ieee standard for information technology - portable operating sytem interface - part 1: System application program interface amendment 1: Realtime extension, 1993.

[42] John A. Stankovic. Misconceptions about real-time computing: A serious problem for next-generation systems. *Computer*, 21(10):10–19, 1988.

[43] John A. Stankovic, Chenyang Lu, and Sang H. Son. The case for feedback control real-time scheduling. Technical report, Charlottesville, VA, USA, 1998.

[44] Symbian. Symbian foundation website, Acessado em Agosto de 2008. http://www.symbianfoundation.org/.

[45] Trolltech. Qt cross-platform application framework, Acessado em Agosto de 2008. http://trolltech.com/products/qt.

[46] Karim Yaghmour. Adaptive domain environment for operating systems, Acessado em Dezembro de 2007. www.opensys.com.