FELIPE GIUNTE YOSHIDA
MARIANA RAMOS FRANCO
VINICIUS TOSTA RIBEIRO

MICROKERNEL PARA A PLACA ARM EVALUATOR-7T

São Paulo
2009

FELIPE GIUNTE YOSHIDA
MARIANA RAMOS FRANCO
VINICIUS TOSTA RIBEIRO

MICROKERNEL PARA A PLACA ARM EVALUATOR-7T

Monografia apresentada à Escola Politécnica da Universidade de São Paulo para a Conclusão do Curso de Engenharia da Computação.

São Paulo
2009

FELIPE GIUNTE YOSHIDA
MARIANA RAMOS FRANCO
VINICIUS TOSTA RIBEIRO

MICROKERNEL PARA A PLACA ARM EVALUATOR-7T

Monografia apresentada à Escola Politécnica da Universidade de São Paulo para a Conclusão do Curso de Engenharia da Computação.

Orientador:
Prof. Dr. Jorge Kinoshita

São Paulo
2009

FICHA CATALOGRÁFICA

Yoshida, Felipe Giunte
MICROKERNEL PARA A PLACA ARM EVALUATOR-7T / F.G. Yoshida, M.R. Franco, V.T. Ribeiro. – São Paulo, 2009. 60 p.

Trabalho de Formatura — Escola Politécnica da Universidade de São Paulo. Departamento de Engenharia de Computação e Sistemas Digitais.

1. Microkernel. 2. ARM. I. Yoshida, Felipe Giunte II. Franco, Mariana Ramos III. Ribeiro, Vinicius Tosta IV. Universidade de São Paulo. Escola Politécnica. Departamento de Engenharia de Computação e Sistemas Digitais. II. t.

DEDICATÓRIA

a ...

AGRADECIMENTOS

Agradeço a

RESUMO

Exemplo de modelo de teses e dissertações da poli utilizando LaTeX. O estilo foi baseado no modelo da ABNT e "adaptado" para particularidades da Poli.

ABSTRACT

This document is an example of the Poli's thesis format using LaTeX. The document class is based on the ABNT class with little changes to fit some Poli singularities.

LISTA DE FIGURAS

LISTA DE TABELAS

LISTA DE ABREVIATURAS

ALU Arithmetic Logic Unit

ARM Advanced RISC Machine

CISC Complex Instruction Set Computer

CPSR Current Program Status Register

FIQ Fast Interrupt

IDE Integrated Development Environment

IRQ Interrupt Request

LR Link Register

PC Program Counter

PSR Program Status Register

RISC Reduced Instruction Set Computer

SP Stack Pointer

SPRS Saved Program Register

USP Universidade de São Paulo

LISTA DE SÍMBOLOS

RX - registrador número X

RX_Y - registrador número X do modo de operação Y

SUMÁRIO

1 INTRODUÇÃO

1.1 Objetivo

O objetivo deste projeto de formatura é desenvolver um *microkernel* para a placa experimental ARM Evalutator-7T, constituída de um processador ARM7 e de alguns periféricos simples.

O *microkernel* implementa os mecanismos básicos de um sistema operacional, como o chaveamento de processos, as chamadas de sistema e utiliza algumas rotinas para a comunicação com os periféricos da placa.

Além disso, foram criados alguns programas para testar e exemplificar o funcionamento do *microkernel*. Entre esses programas, um simples terminal foi desenvolvido para a interação dos usuários com o sistema.

1.2 Motivação

As disciplinas de Laboratório de Processadores e de Sistemas Operacionais do curso de Engenharia da Computação na Escola Politécnica da USP, atualmente, estão muito distantes entre si, no entanto o conteúdo das mesmas é muito próximo.

Pensando em como aproximar essas duas disciplinas, surgiu a idéia de desenvolver uma ferramenta didática que unisse um *hardware* e sistema operacional de estudo simples, e que pudesse ser utilizada nas experiências do Laboratório de Processadores.

Para criação desta ferramenta, foi escolhida a placa experimental ARM Evaluator-7T, que possui uma arquitetura ARM e um poder de processamento bastante superior aos sistemas didáticos utilizados atualmente (baseados nos processadores Intel 8051 e o Motorola 68000). Assim sendo, pretende-se atualizar o material didático da disciplina de processadores, trazendo um sistema mais moderno e mais próximo da realidade atual, além de poder se relacionar com o conteúdo da disciplina de Sistemas Operacionais.

Outra motivação do projeto foi aprofundar nossos conhecimentos sobre sistemas operacionais e sobre a arquitetura dos processadores ARM, visto que este processador é, hoje em dia, largamente utilizado em sistemas embarcados e aparelhos celulares.

1.3 Justificativa

O objetivo inicial do projeto era portar um sistema operacional Unix já existente para a placa didática Evaluator-7T.

Inicialmente pensamos em utilizar os sistemas Android e Minix 3, mas ao estudar o *kernel* dos dois sistemas, vimos que os recursos de memória necessários para executá-los era muito maior que os 512kB disponíveis na placa. Além disso, no caso do Minix 3, teríamos que reescrever o *assembly* do *kernel* que atualmente só tem versão para i386, para *assembly* ARM, o que seria impossível com o tempo disponível para o projeto.

Assim surgiu a idéia de desenvolver um *microkernel* próprio, com as funcionalidades básicas de um sistema operacional, e que fosse de fácil entendimento; pois como mencionado anteriormente, espera-se que o material desenvolvido seja destinado a melhorar e aproximar o ensino de Sistemas Operacionais com as experiências do Laboratório de Processadores.

1.4 Metodologia de Trabalho

Para a realização desse projeto de formatura procurou-se seguir uma metodologia de trabalho cujas etapas são descritas a seguir:

- Estudo da Arquitetura ARM e da Placa Didática Evaluator-7T:

  Antes de especificar as funcionalidades que seriam desenvolvidas, um estudo aprofundado da arquitetura ARM foi realizado para compreender o funcionamento do processador para o qual o *microkernel* foi desenvolvido, o ARM7TDMI.

  Além disso, foram executados alguns programas exemplo na placa didática Evaluator-7T para adquirir conhecimentos sobre o seu funcionamento e limitações.

- Montagem do Ambiente de Trabalho:

  Paralelamente ao estudo descrito no item anterior, foi montado um ambiente de trabalho utilizando a IDE CodeWarrior para o desenvolvimento do código-fonte e o AXD Debugger para depurar o funcionamento do *microkernel* com ou sem a utilização da placa didática.

Um repositório de controle de versão também foi montado para estocar o material produzido durante do projeto (documentação e código-fonte) e para sincronizar o trabalho dos integrantes do grupo.

- Especificação Funcional do Microkernel:

  O *microkernel* desenvolvido foi especificado nessa etapa, onde foram levantadas as funcionalidades básicas de um sistema operacional que deveriam ser implementadas, como o chaveamento de processos e as chamadas de sistema.

- Desenvolvimento do Microkernel:

  Nessa fase, foi desenvolvido o *microkernel* utilizando como base a especificação definida no item anterior.

- Análise do Microkernel e Conclusões:

  Ao final do desenvolvimento, com base nas dificuldades e soluções encontradas, foi feita uma análise e conclusão sobre o *microkernel* desenvolvido e sua possível utilização no Laboratório de Processadores para exemplificar os conceitos vistos na disciplina de Sistemas Operacionais.

1.5 Organização do Documento

Este documento foi estruturado da seguinte maneira:

- Capítulo 1 (Introdução):

  Apresenta objetivo, motivações, justificativas e a metodologia do trabalho.

- Capítulo 2 (Conceitos e Tecnologias Envolvidas):

  Contextualiza o leitor em aspectos técnicos específicos utilizados no desenvolvimento do trabalho.

- Capítulo 3 (O Sistema Operacional KinOS):

  Descreve como o *microkernel* foi desenvolvido, quais as suas funcionalidades e como funciona a sua integração com os periféricos da placa didática, com o terminal e com os outros programas implementados.

- Capítulo 4 (Considerações Finais):

Analisa os resultados obtidos em relação ao objetivo do projeto, as conclusões, as contribuições deste trabalho e indica possíveis trabalhos futuros com base neste.

2 CONCEITOS E TECNOLOGIAS ENVOLVIDAS

2.1 O Processador ARM7TDMI

O ARM7TDMI faz parte da família de processadores ARM7 32 bits conhecida por oferecer bom desempenho aliado a um baixo consumo de energia. Essas características fazem com que o ARM7TDMI seja bastante utilizado em media players, vídeo games e, principalmente, em sistemas embarcados e num grande número de aparelhos celulares (SLOSS; SYMES; WRIGHT, 2004).

2.1.1 Arquitetura RISC

Os processadores ARM, incluindo o ARM7TDMI, foram projetados com a arquitetura RISC.

RISC (*Reduced Instruction Set Computer*) é uma arquitetura de computadores baseada em um conjunto simples e pequeno de instruções capazes de serem executadas em um único ou poucos ciclos de relógio.

A idéia por trás da arquitetura RISC é de reduzir a complexidade das instruções executadas pelo *hardware* e deixar as tarefas mais complexas para o *software*. Como resultado, o RISC demanda mais do compilador do que os tradicionais computadores CISC (*Complex Instruction Set Computer*) que, por sua vez, dependem mais do processador já que suas instruções são mais complicadas (SLOSS; SYMES; WRIGHT, 2004).

As principais características da arquitetura RISC são:

1. Conjunto reduzido e simples de instruções capazes de serem executadas em único ciclo de máquina.

2. Uso de *pipeline*, ou seja, o processamento das instruções é quebrado em pequenas unidades que podem ser executadas em paralelo.

3. Presença de um conjunto de registradores.

4. Arquitetura *Load-Store*: o processador opera somente sobre os dados contidos nos registradores e instruções de *load*/*store* transferem dados entre a memória e os registradores.

5. Modos simples de endereçamento de memória.

2.1.2 Pipeline

A arquitetura de *pipeline* aumenta a velocidade do fluxo de instruções para o processador, pois permite que várias operações ocorram simultaneamente, fazendo o processador e a memória operarem continuamente (ARM LIMITED, 2001).

O ARM7 possui uma arquitetura de *pipeline* de três estágios. Durante operação normal, o processador estará sempre ocupado em executar três instruções em diferentes estágios. Enquanto executa a primeira, decodifica a segunda e busca a terceira.

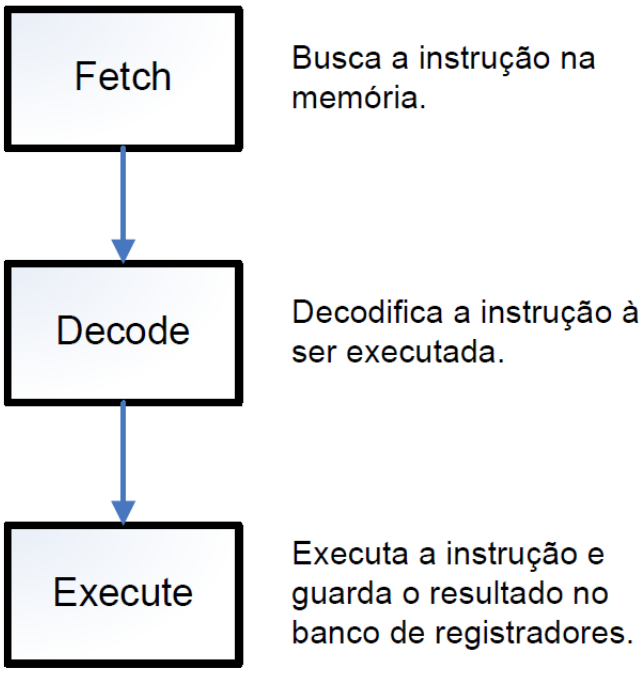

Figura 2.1: *Pipeline* de 3 estágios (ARM LIMITED, 2001)

O primeiro estágio de *pipeline* lê a instrução da memória e incrementa o valor do registrador de endereços, que guarda o valor da próxima instrução a ser buscada. O próximo estágio decodifica a instrução e prepara os sinais de controle necessários para executá-la. O terceiro lê os operandos do banco de registradores, executa as operações através da ALU (*Arithmetic*

Logic Unit), lê ou escreve na memória, se necessário, e guarda o resultado das instruções no banco de registradores.

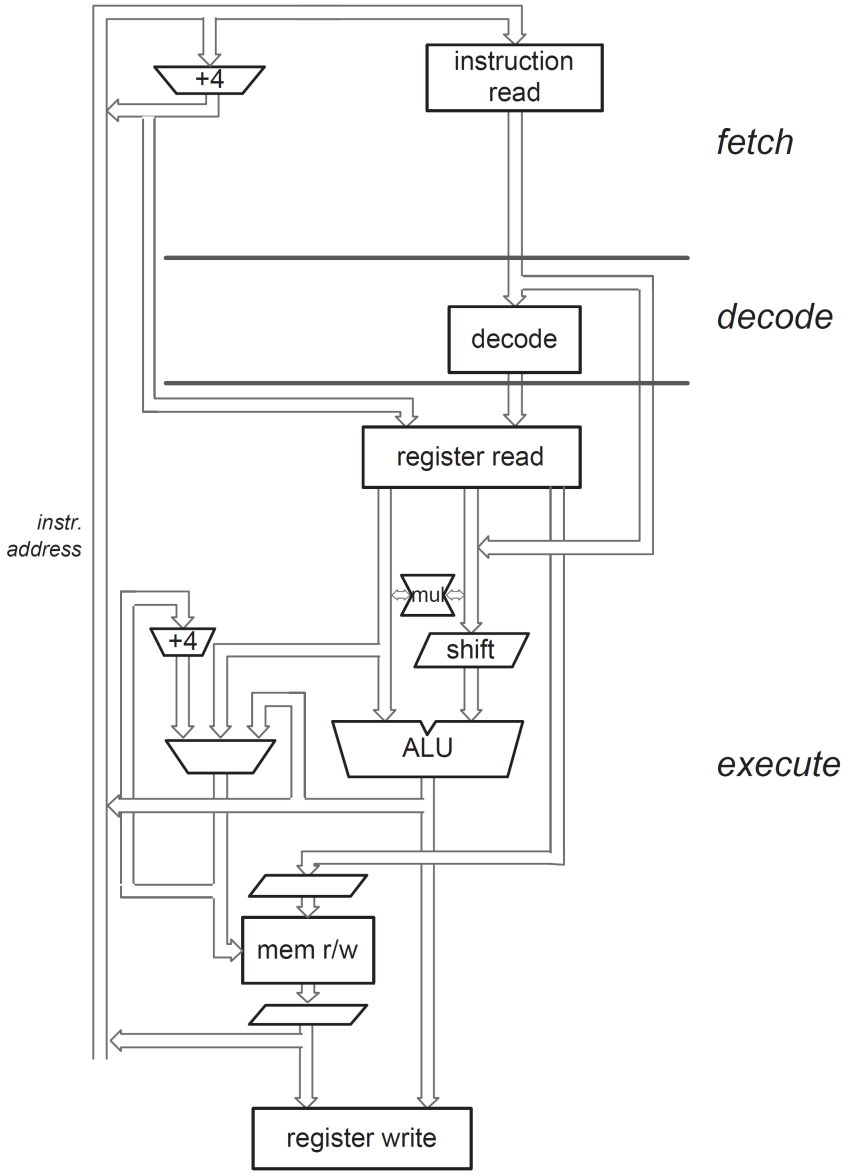

Figura 2.2: *Pipeline* do ARM7TDMI (RYZHYK, 2006)

Algumas características importantes do *pipeline* do ARM7:

- O *Program Counter* (PC) ao invés de apontar para a instrução que esta sendo executada, aponta para a instrução que esta sendo buscada na memória.
- O processador só processa a instrução quando essa passa completamente pelo estágio de execução (*execute*). Ou seja, somente quando a quarta instrução é buscada (*fetched*).

- A execução de uma instrução de *branch* através da modificação do PC provoca a descarga, eliminação, de todas as outras instruções do *pipeline*.

- Uma instrução no estágio *execute* será completada mesmo se acontecer uma interrupção. As outras instruções no *pipeline* serão abandonadas e o processador começará a preencher o *pipeline* a partir da entrada apropriada no vetor de interrupção.

2.1.3 Estados de Operação

O processador ARM7TDMI possui dois estados de operação (ARM LIMITED, 2001):

- ARM: modo normal, onde o processador executa instruções de 32 bits (cada instrução corresponde a uma palavra);

- Thumb: modo especial, onde o processador executa instruções de 16 bits que correspondem à meia palavra.

Instruções Thumbs são um conjunto de instruções de 16 bits equivalentes as instruções 32 bits ARM. A vantagem em tal esquema, é que a densidade de código aumenta, já que o espaço necessário para um mesmo número de instruções é menor. Em compensação, nem todas as instruções ARM tem um equivalente Thumb.

Neste projeto, o processador é usado no modo ARM que facilita o desenvolvimento por possuir um número maior de instruções.

2.1.4 Modos de Operação

Os processadores ARM possuem 7 modos de operação, como apresentado na tabela 2.1.

Mudanças no modo de operação podem ser realizadas através de programas, ou podem ser causadas por interrupções externas ou exceções (interrupções de software).

A maioria dos programas roda no modo Usuário. Quando o processador esta no modo Usuário, o programa que esta sendo executado não pode acessar alguns recursos protegidos do sistema ou mudar de modo sem ser através de uma interrupção (ARM LIMITED, 2005).

Os outros modos são conhecidos como modos privilegiados. Eles têm total acesso aos recursos do sistema e podem mudar livremente de modo de operação. Cinco desses modos são conhecidos como modos de interrupção: FIQ, IRQ, Supervisor, *Abort* e Indefinido.

Modo	Identificador	Descrição
Usuário	usr	Execução normal de programas.
FIQ(*Fast Interrupt*)	fiq	Tratamento de interrupções rápidas.
IRQ (*Interrupt*)	irq	Tratamento de interrupções comuns.
Supervisor	svc	Modo protegido para o sistema operacional.
Abort	abt	Usado para implementar memória virtual ou manipular violações na memória.
Sistema	sys	Executa rotinas privilegiadas do sistema operacional.
Indefinido	und	Modo usado quando uma instrução desconhecida é executada.

Tabela 2.1: Modos de operação (ARM LIMITED, 2005)

Entra-se nesses modos quando uma interrupção ocorre. Cada um deles possui registradores adicionais que permitem salvar o modo Usuário quando uma interrupção ocorre.

O modo remanescente é o modo Sistema, que não é acessível por interrupção e usa os mesmos registradores disponíveis para o modo Usuário. No entanto, este é um modo privilegiado e, assim, não possui as restrições do modo Usuário. Este modo destina-se as operações que necessitam de acesso aos recursos do sistema, mas querem evitar o uso adicional dos registradores associados aos modos de interrupção.

2.1.5 Registradores

O processador ARM7TDMI tem um total de 37 registradores:

- 31 registradores de 32 bits de uso geral
- 6 registradores de estado

Esses registradores não são todos acessíveis ao mesmo tempo. O modo de operação do processador determina quais registradores são disponíveis ao programador (ARM LIMITED, 2001).

2.1.5.1 Modo Usuário e Sistema

O conjunto de registradores para o modo Usuário (o mesmo usado no modo Sistema) contém 16 registradores diretamente acessíveis, R0 à R15. Um registrador adicional, o CPSR (*Current Program Status Register*), contém os bits de *flag* e de modo.

Os registradores R13 à R15 possuem as seguintes funções especiais (SLOSS; SYMES; WRIGHT, 2004):

- R13: usado como ponteiro de pilha, *Stack Pointer* (SP)
- R14: é chamado de *Link Register* (LR) e é onde se coloca o endereço de retorno sempre que uma sub-rotina é chamada.
- R15: corresponde ao *Program Counter* (PC) e contém o endereço da próxima instrução à ser executada pelo processador.

2.1.5.2 Modos privilegiados

Além dos registradores acessíveis ao programador, o ARM coloca à disposição mais alguns registradores nos modos privilegiados. Esses registradores são mapeados aos registradores acessíveis ao programador no modo Usuário e permitem que estes sejam salvos a cada interrupção.

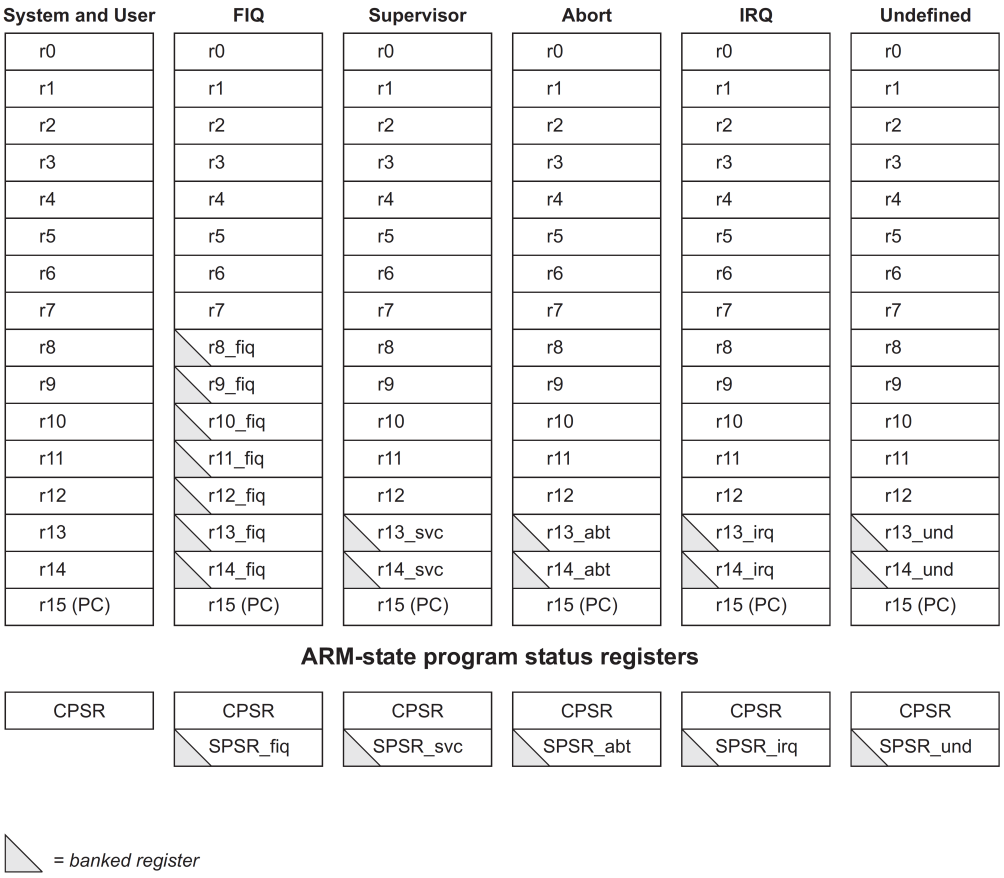

Figura 2.3: Organização dos registradores no modo ARM (ARM LIMITED, 2001)

Como se pode verificar na figura 2.3, cada modo tem o seu próprio R13 e R14. Isso permite que cada modo mantenha seu próprio ponteiro de pilha (SP) e endereço de retorno (LR) (ZAITSEFF, 2003).

Além desses dois registradores, o modo FIQ possui mais cinco registradores especiais: R8_fiq-R12_fiq. Isso significa que quando o processador muda para o modo FIQ, o programa não precisa salvar os registradores de R8 à R12.

Esses registradores especiais mapeiam de um pra um os registradores do modo Usuário. Se ocorrer uma mudança de modo do processador, um registrador particular do novo modo irá substituir o registrador existente.

Por exemplo, quando o processador esta no modo IRQ, as instruções executadas continuarão a acessar os registradores R13 e R14. No entanto, esses serão os registradores especiais R13_irq e R14_irq. Os registradores do modo usuário (R13_usr e R14_usr) não serão afetados pelas instruções referenciando esses registradores. O programa continua tendo acesso normal aos outros registradores de R0 à R12 (SLOSS; SYMES; WRIGHT, 2004).

2.1.6 Registradores de Estado

O *Current Program Status Register* (CPRS) é acessível em todos os modos do processador. Ele contém as *flags* de condição, os bits para desabilitar as interrupções, o modo atual do processador, e outras informações de estado e controle. Cada modo de interrupção possui também um *Saved Program Register* (SPRS), que é usado para preservar o valor do CPSR quando a interrupção associada acontece (ARM LIMITED, 2005).

Assim, os registradores de estado (ARM LIMITED, 2001):

- Guardam informação sobre a operação mais recente executada pela ALU.
- Controlam o ativar e desativar de interrupções.
- Determinam o modo de operação do processador.

Como mostrado na figura 2.4 o CPSR é dividido em 3 campos: *flag*, reservado (não utilizado) e controle.

O campo de controle guarda os bits de modo, estado e de interrupção, enquanto o campo *flag* armazena os bits de condição.

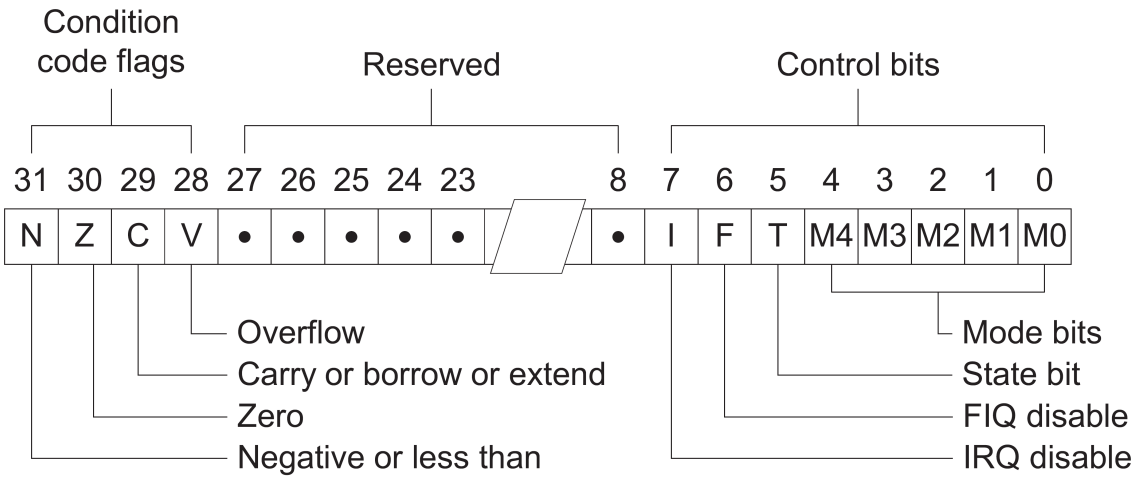

Figura 2.4: Formato dos registradores de estado CPSR e SPSR (ARM LIMITED, 2001)

2.1.6.1 Flags de Condição

Os bits N, Z, C e V são *flags* de condição, e é possível alterá-los através do resultado de operações lógicas ou aritméticas (ARM LIMITED, 2005).

Os *flags* de condição são normalmente modificados por:

- Uma instrução de comparação (CMN, CMP, TEQ, TST).
- Alguma outra instrução aritmética, lógica ou *move*, onde o registrador de destino não é o R15 (PC).

Nesses dois casos, as novas *flags* de condição (depois de a instrução ter sido executada) normalmente significam:

- N: Indica se o resultado da instrução é um número positivo (N=0) ou negativo (N=1).
- Z: Contém 1 se o resultado da instrução é zero (isso normalmente indica um resultado de igualdade para uma comparação), e 0 se o contrário.
- C: Pode possuir significados diferentes:
  - Para uma adição, C contém 1 se a adição produz "vai-um"(*carry*), e 0 caso contrário.
  - Para uma subtração, C contém 0 se a subtração produz "vem-um"(*borrow*), e 1 caso contrário.

  - Para as instruções que incorporam deslocamento, C contém o último bit deslocado para fora pelo deslocador.
  - Para outras instruções, C normalmente não é usado.

- V: Possui dois significados:
  - Para adição ou subtração, V contém 1 caso tenha ocorrido um *overflow* considerando os operandos e o resultado em complemento de dois.
  - Para outras instruções, V normalmente não é usado.

2.1.6.2 Bits de Controle

Os oito primeiros bits de um PSR (*Program Status Register*) são conhecidos como bits de controle (ARM LIMITED, 2005). Eles são:

- Bits de desativação de interrupção
- Bit T
- Bits de modo

Os bits de controle mudam quando uma interrupção acontece. Quando o processador esta operando em um modo privilegiado, programas podem manipular esses bits.

Bits de desativação de interrupção

Os bits I e F são bits de desativação de interrupção:

- Quando o bit I é ativado, as interrupções IRQ são desativadas.
- Quando o bit F é ativado, as interrupções FIQ são desativadas.

Bit T

O bit T reflete o modo de operação:

- Quando o bit T é ativado, o processador é executado em estado Thumb.
- Quando o bit T é desativado, o processador é executado em estado ARM.

Bits de modo

Os bits M[4:0] determinam o modo de operação. Nem todas as combinações dos bits de modo definem um modo válido, portando deve-se tomar cuidado para usar somente as combinações mostradas na tabela 2.2.

Bit de modo	Modo de operação	Registradores acessíveis
10000	Usuário(usr)	PC,R14-R0,CPSR
10001	FIQ(fiq)	PC,R14_fiq-R8_fiq,R7-R0,CPSR,SPSR_fiq
10010	IRQ(irq)	PC,R14_irq, R13_irq,R12-R0,CPSR,SPSR_irq
10011	Supervisor(svc)	PC,R14_svc, R13_irq,R12-R0,CPSR,SPSR_svc
10111	*Abort*(abt)	PC,R14_abt, R13_irq,R12-R0,CPSR,SPSR_abt
11011	Indefinido(und)	PC,R14_und, R13_irq,R12-R0,CPSR,SPSR_und
11111	Sistema(sys)	PC,R14-R0,CPRS

Tabela 2.2: Valores para o bit de modo (ARM LIMITED, 2005)

2.1.7 Interrupções

Interrupções surgem sempre que o fluxo normal de um programa deve ser interrompido temporariamente, por exemplo, para servir uma interrupção vinda de um periférico ou a tentativa de executar uma instrução desconhecida. Antes de tentar lidar com uma interrupção, o ARM7TDMI preserva o estado atual de forma que o programa original possa ser retomado quando a rotina de interrupção tiver acabado (ARM LIMITED, 2001).

A arquitetura ARM suporta 7 tipos de interrupções. A tabela 2.3 lista os tipos de interrupção e o modo do processador usado para lidar com cada tipo. Quando uma interrupção acontece, a execução é forçada para um endereço fixo de memória correspondente ao tipo de interrupção. Esses endereços fixos são chamados de vetores de interrupção (ARM LIMITED, 2005).

Deve-se notar olhando para a tabela 2.3, que existe espaço suficiente para apenas uma instrução entre cada vetor de interrupção (4 bytes). Estes são inicializados com instruções de desvio (*branch*).

2.1.7.1 Prioridade das Interrupções

Quando várias interrupções acontecem ao mesmo tempo, uma prioridade fixa do sistema determina a ordem na qual elas serão manipuladas. Essa prioridade é listada na tabela 2.4:

Tipo de interrupção	**Modo de operação**	**Endereço**
Reset	Supervisor	0x00000000
Instrução indefinida	Indefinido	0x00000004
Interrupção de Software (swi)	Supervisor	0x00000008
Prefetch abort	*Abort*	0x0000000C
Data abort	*Abort*	0x00000010
Interrupção normal(IRQ)	IRQ	0x00000018
Interrupção rápida(FIQ)	FIQ	0x0000001C

Tabela 2.3: Vetor de interrupção (ARM LIMITED, 2005)

Prioridade	Interrupção
alta	Reset
	Data abort
	FIQ
	IRQ
	Prefetch abort
baixa	Instrução indefinida e interrupção de software (swi)

Tabela 2.4: Ordem de prioridade das interrupções (ARM LIMITED, 2001)

2.1.7.2 Entrada de interrupção

Executar uma interrupção necessita que o processador preserve o estado atual: em geral, o conteúdo de todos os registradores (especialmente PC e CPSR) devem ser o mesmo depois de uma interrupção.

O processador ARM usa os registradores adicionais de cada modo para ajudar a salvar o estado do processador. Quando uma interrupção acontece, o R14 e o SPSR são usados para guardar o estado atual da seguinte maneira (ARM LIMITED, 2001):

1. Preserva o endereço da próxima instrução (PC+4 ou PC+8, depende da interrupção) no apropriado LR (R14). Isso permite ao programa continuar do lugar de onde parou no retorno da interrupção.
2. Copia o CPSR para o apropriado SPSR.
3. Força os bits de modo do CPSR para um valor que corresponde ao tipo de interrupção.
4. Força o PC buscar a próxima instrução no vetor de interrupção.

O processador ARM7TDMI também pode ativar a *flag* de interrupção para desabilitar próximas interrupções.

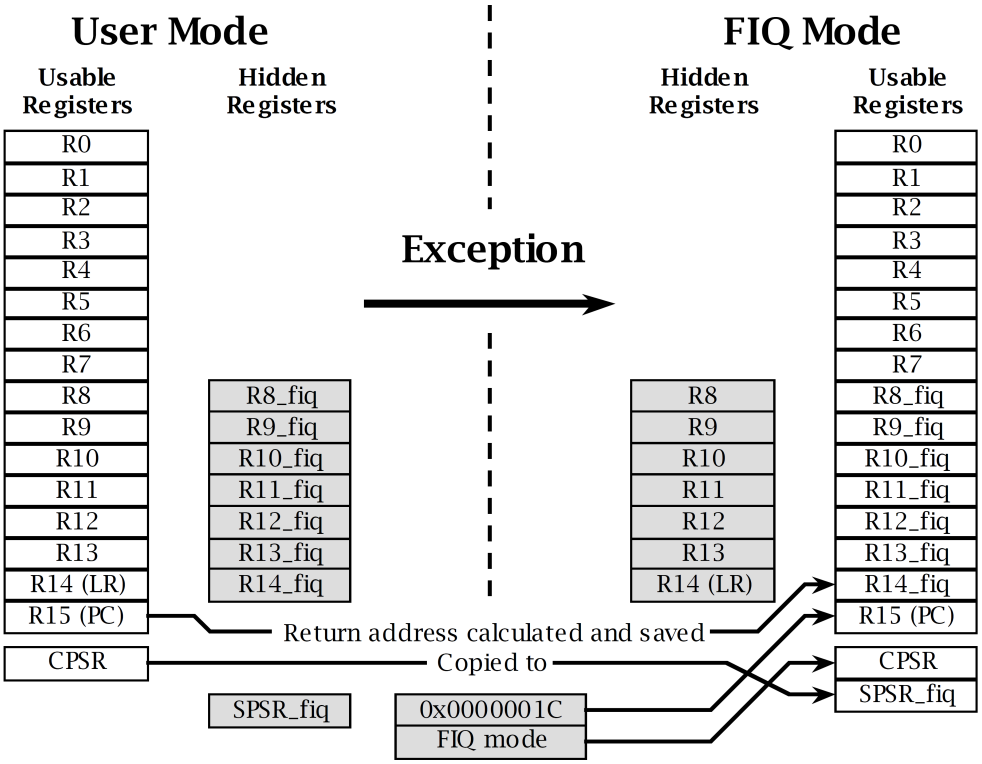

Figura 2.5: Esquema de uma interrupção no ARM7TDMI (ZAITSEFF, 2003)

2.1.7.3 Saída de interrupção

Quando uma interrupção é completada deve-se (ARM LIMITED, 2001):

1. Mover o LR (R14), menos um *offset*, para o PC. O offset varia de acordo com o tipo de interrupção mostrada na figura anterior.
2. Copiar o SPSR de volta para o CPSR.
3. Desativa as *flags* de interrupção que foram ativadas na entrada.

2.1.7.4 Interrupções de software

Uma interrupção de software é uma interrupção inicializada inteiramente por um programa para entrar no modo Supervisor e assim poder utilizar alguma rotina particular, como operações de entrada e saída do sistema (ZAITSEFF, 2003).

Quando uma interrupção de software é executada, as seguintes ações são realizadas (ARM LIMITED, 2005):

1. Copia o endereço da próxima instrução no registrador LR_svc (R14_svc).

```
R14_svc = endereço da próxima instrução
```

2. Copia o CPSR no SPSR_svc.

```
SPSR_svc = CPSR
```

3. Ativa os bits de modo do CPSR com o valor correspondente ao modo Supervisor.

```
CPSR[4:0] = 0b10011 /* modo Supervisor */
```

4. Reforça o estado ARM colocando o bit T do CPSR à zero.

```
CPSR[5] = 0 /* estado ARM */
```

5. Desabilita as interrupções normais ativando o bit I do CPSR. Interrupções FIQ não são desabilitadas e podem continuar ocorrendo.

```
CPSR[7] = 1 /* desabilita interrupções normais */
```

6. Carrega o endereço do vetor de interrupções, 0x00000008, no PC.

```
PC = 0x00000008
```

Para retornar da operação de interrupção, é usada a seguinte instrução para restaurar o PC (a partir do R14_svc) e o CPSR (a partir do SPSR_svc):

```
MOVS PC, LR
```

2.1.7.5 Interrupções de hardware

Interrupções de hardware são mecanismos que permitem que um sinal externo (pedido de interrupção) interrompa a execução normal do programa corrente e desvie a execução para um bloco de código chamado de rotina de interrupção (KINOSHITA, 2007).

Interrupções são úteis, pois permitem que o processador manuseie periféricos de uma maneira mais eficiente. Sem interrupções o processador teria que verificar periodicamente a entrada/saída de um dispositivo para ver se esse necessita de atenção. Com interrupções, por outro lado, a entrada/saída do dispositivo pode indicar diretamente a ocorrência de um dado

evento externo, que será tratado com maior facilidade e rapidez, de modo que o microprocessador não necessite consumir tempo de processamento para pesquisar a ocorrência de eventos externos.

O processador ARM fornece dois sinais que são usados pelos periféricos para pedir uma interrupção: o sinal de interrupção nIRQ e o sinal de interrupção rápida nFIQ. Ambos são ativados em nível baixo, ou seja, colocando o sinal em nível baixo geramos a interrupção correspondente, se a interrupção não tiver sido desabilitada no CPSR (ZAITSEFF, 2003).

Quando uma interrupção de *hardware* IRQ (ou FIQ) é detectada, as seguintes ações são realizadas (ARM LIMITED, 2005):

1. Copia o endereço da próxima instrução à ser executada + 4 no registrador LR_irq (R14_irq). Isso significa que o LR_irq irá apontar para a segunda instrução à partir do ponto de pedido da interrupção.

```
R14_irq = endereço da próxima instrução + 4
```

2. Copia o CPSR no SPSR_irq.

```
SPSR_irq = CPSR
```

3. Coloca os bits de modo do CPSR para o valor correspondente ao modo IRQ.

```
CPSR[4:0] = 0b10010 /* modo IRQ */
```

4. Reforça o estado ARM colocando o bit T do CPSR à zero.

```
CPSR[5] = 0 /* estado ARM */
```

5. Desabilita as interrupções normais ativando o bit I do CPSR. Interrupções FIQ não são desabilitadas e podem continuar ocorrendo.

```
CPSR[7] = 1 /* desabilita interrupções normais */
```

6. Carrega o endereço do vetor de interrupções, 0x00000008, no PC.

```
PC = 0x00000018
```

Assim que a rotina de interrupção é terminada, o processador retorna ao que estava fazendo antes através das seguintes ações:

1. Move o conteúdo do registrador LR_irq menos 4 para o PC.
2. Copia SPSR_irq de volta para CPSR.

A seguinte instrução executa os passos mostrados a cima:

```
SUBS PC, R14,#4
```

Note que a instrução é SUBS, e não SUB: a instrução SUBS copia automaticamente SPSR no CPSR, mas apenas quando o registrador de destino é o PC (R15) e a instrução é executada em um modo privilegiado.

O processamento das *Fast Interrupt* (FIQ) é praticamente igual ao de uma interrupção normal (IRQ). As diferenças são que um conjunto diferente de registradores é usado (i.e. R14_fiq no lugar de R14_irq), que tanto as interrupções IRQ quanto as FIQ são desativadas (ou seja, os bits I e F do CPSR são ativados), e que o endereço do vetor de interrupção é 0x0000001C (ZAITSEFF, 2003).

2.2 A Placa Experimental Evaluator-7T

O principal elemento de hardware deste projeto é a placa experimental ARM Evaluator-7T, baseada no processador ARM7TDMI, um processador RISC de 32 bits capaz de executar o conjunto de instruções denominado Thumb.

Os principais elementos presentes na arquitetura da placa Evaluator-7T são os seguintes:

- Microcontrolador Samsung KS32C50100
- 512kB EPROM flash
- 512kB RAM estática (SRAM)
- Dois conectores RS232 de 9 pinos tipo D
- Botões de reset e de interrupção
- Quatro LEDs programáveis pelo usuário e um display de 7 segmentos
- Entrada de usuário por um interruptor DIP com 4 elementos

- Conector Multi-ICE
- Clock de 10MHz (o processador usa-o para gerar um clock de 50MHz)
- Regulador de tensão de 3.3V

A figura 2.6 mostra a organização desses elementos na placa experimental.

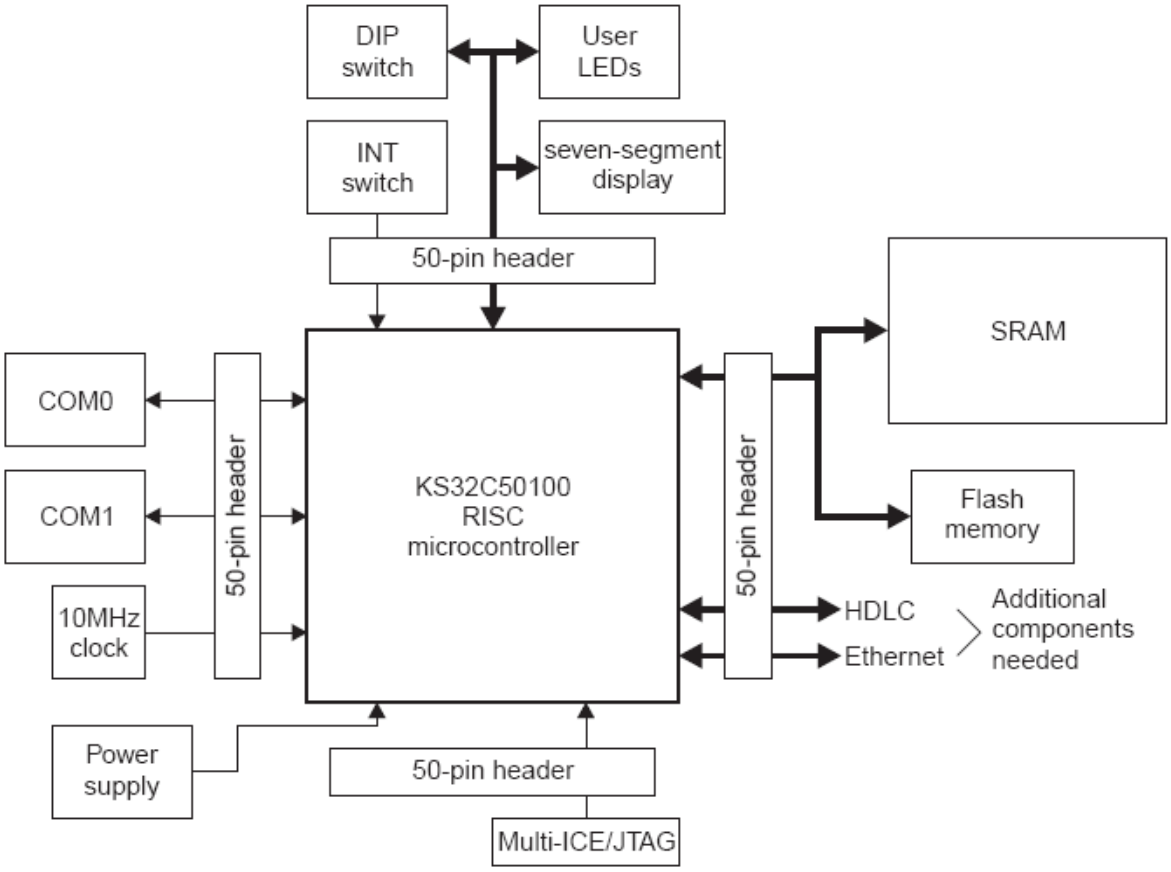

Figura 2.6: Arquitetura da placa Evaluator-7T. Fonte:

Com relação à memória flash da placa, ela vem de fábrica com o bootstrap loader da placa e programa monitor de debug. O restante dela pode ser usado para os programas de usuário. A tabela 2.5 mostra a faixa de endereços de cada região da memória.

Tabela 2.5: Mapa da memória flash

Faixa de endereço	Descrição
0x01800000 a 0x01806FFF	Bootstrap loader
0x01807000 a 0x01807FFF	Teste de produção
0x01808000 a 0x0180FFFF	Reservado
0x01810000 a 0x0181FFFF	Angel
0x01820000 a 0x0187FFFF	Disponível para outros programas e dados

Já em relação às duas portas seriais presentes na placa, cada uma tem usos específicos. A primeira, chamada DEBUG, é usada pelo monitor de debug ou pelo programa bootstrap presente na placa. Ela está conectada ao UART1 do microcontrolador. A segunda, chamada USER, é de uso genérico e está disponível para uso em programas. Ela está conectada ao UART0 do microcontrolador.

2.2.1 Bootstrap Loader

Como mencionado anteriormente, a memória flash da placa contém uma região reservada para os programas Bootstrap Loader (BSL) e o programa monitor de debug chamado Angel.

O BSL é o primeiro programa a ser executado pelo microcontrolador quando esta é ligada ou reiniciada. Suas principais funções são:

- Fazer a conexão com o computador através da porta serial e uma aplicação de terminal, como o HyperTerminal do Windows
- Prover a infraestrutura necessária à configuração da placa
- Prover ajuda ao usuário
- Gerenciar imagens de memória como um conjunto de módulos executáveis
- Carregar aplicações na SRAM e executá-las

2.2.2 Angel Debug Monitor

O monitor de debug Angel é fornecido conjuntamente com diversas placas da ARM e suas parceiras. Suas principais funcionalidades são:

- Função de depuração de código, incluindo inspeção de memória, download e execução de imagens de memória, uso de breakpoints e execução passo-a-passo
- Inicialização da CPU e da placa e tratamento básico de exceções
- Uma biblioteca ANSI C completa, com uso de semihosting para prover serviços do computador host que não estão disponíveis na placa

Há duas maneiras pelas quais o Angel se comunica com o ambiente de desenvolvimento de software.

A primeira é através da biblioteca de interfaces chamada "Remote_A". Por ela, os depuradores se comunicam com um alvo do Angel quando fazem depuração ou execução de código.

A segunda é por meio de interrupções de software (SWI). O código do programa faz uma SWI para solicitar serviços dos Angel diretamente ou através da biblioteca C dod toolkit.

3 O SISTEMA OPERACIONAL KINOS

O principal objetivo do projeto é auxiliar o ensino de sistemas operacionais e da arquitetura ARM nas disciplinas de Sistemas Operacionais e Laboratório de Microprocessadores. Para tal, foi desenvolvido um microkernel, apelidado de KinOS, cujas funções básicas são o chaveamento de threads através de interrupção de timer, as chamadas de sistema, as rotinas de manipulação de hardware, funções de semáforo e um shell.

3.1 Organização do código

A estrutura de arquivos do projeto pode ser vista na figura 3.1. Pode-se dividi-lo em cinco partes:

- **Raiz** Contém os arquivos de inicialização da placa
- **Pasta "apps"** Contém os programas que serão executados pelo microkernel
- **Pasta "interrupt"** Contém as rotinas de tratamento de interrupção
- **Pasta "peripherals"** Contém rotinas de manipulação de hardware
- **Pasta "syscalls"** Contém as chamadadas de sistema
- **Pasta "mutex"** Contém rotinas de semáforo

A pasta *KinOS_Data* não é considerada parte do projeto pois é utilizada pelo CodeWarrior para o armazenamento do código compilado.

3.1.1 Raiz

Os arquivos encontrados na raiz do projeto são responsáveis pela inicialização da placa e pela declaração de constantes globais. O arquivo startup.s contém a chamada inicial do

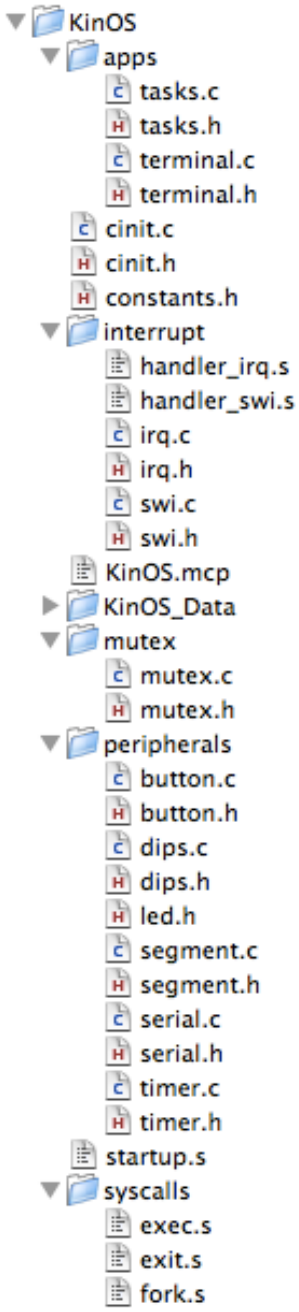

Figura 3.1: Estrutura de arquivos.

microkernel, onde toda parte de inicialização em assembly é feita. Já o arquivo cinit.c também contém a parte de inicialização, porém, a parte que deve ser executada em C. Finalmente, o arquivo constants.h é responsável por armazenar as constantes que são utilizadas em todo o projeto.

3.1.2 Pasta “apps”

Há apenas um arquivo e seu cabeçalho nesta pasta. Nele, várias funções são declaradas, onde cada declaração é considerada uma thread pelo microkernel.

3.1.3 Pasta “interrupt”

Todas as rotinas que tratam e instalam interrupções – tanto de hardware quanto de software – estão localizadas nesta pasta. O arquivo handler_irq.s contém a rotina em assembly que trata das interrupções de hardware, as encaminha para a rotina específica de acordo com a sua fonte e faz o chaveamento de threads. O arquivo irq.c contém uma única rotina, que realiza a instalação da rotina de tratamento de interrupção tanto de hardware quanto de software. A rotina de tratamento de interrupção de software é feita no arquivo handler_swi.s, que identifica o tipo de interrupção e encaminha para alguma das chamadas de sistema, encontradas em swi.c.

3.1.4 Pasta “peripherals”

As rotinas de inicialização e controle dos periféricos se encontram todas nesta pasta. As do botão estão no arquivo button, da chave DIP no arquivo dips, do display de sete segmentos em segment, dos LEDs em led e do timer em timer.

3.1.5 Pasta “syscalls”

As chamadas de sistema estão escritas em assembly e se encontram em três arquivos, uma para cada chamada. São elas as chamadas fork, exec e exit.

3.1.6 Pasta “mutex”

No arquivo mutex há apenas as funções que permitem a exclusão mútua de código.

3.2 Estruturas de dados

A fim de se facilitar a programação e o entendimento do projeto, foram criadas duas estruturas de dados que são acessadas em assembly. A primeira, o Process Control Block é

responsável pelo armazenamento do estado de um processo. Já a Lista de Threads realiza o controle de quais threads estão ativas.

3.2.1 Process Control Block

O Process Control Block (ou simplesmente PCB) é um estrutura de dados que guarda todas as informações de uma thread que aguarda para ser executada enquanto outras estão ativas. Há um PCB para cada uma das nove threads e cada uma ocupa 68 bytes. Ou seja, o espaço total ocupado pelos PCBs é de $9 \cdot 68 = 612$ bytes. Estes 68 bytes estão estruturados como explicitado na figura 3.2. Cada posição da tabela ocupa uma palavra (4 bytes). A primeira posição é em (base do PCB - 4), a segunda em (base do PCB - 8) e assim por diante. Como podemos observar pela figura, as posições 1 a 15 ((base do PCB - 4) a (base do PCB - 60)) armazenam o conteúdo dos registradores r0 a r14 do modo user em ordem inversa. A posição 16 (base do PCB - 64) armazena o link register do modo IRQ, ou seja, o endereço de retorno da interrupção. Finalmente, a posição 17 armazena o registrador de estado do modo user. Estes registradores armazenados permitem estabelecer um retrato preciso do estado do processo quando houve o chaveamento e permite também que este estado seja restabelecido quando for o turno deste processo voltar a ser executado. A estrutura tem seu espaço reservado no arquivo handler_irq.s, e é nomeado com a variável process_control_block, que indica a base da estrutura. Cada um dos PCBs está logo a seguir do anterior. Por exemplo, a base do primeiro PCB está em (process_control_block - 68), do segundo em (process_control_block - $2 \cdot 68$) e assim por diante.

3.2.2 Vetor de threads

O vetor de threads é uma lista que armazena quais das threads estão ativas e quais não estão, a fim de se identificar quais devem ser colocadas em execução. Cada identificador ocupa 4 bytes, e pode ter os valores 0 (inativo) ou 1 (ativo). Como há 9 processos, o tamanho deste vetor é de $4 \cdot 9 = 36$ bytes. Seu espaço é reservado no arquivo handler_irq.s, com o nome de thread_array. No exemplo na figura 3.3 podemos ver que as threads 1, 2 e 4 estão ativas, enquanto que as outras não estão.

Offset	Task Register
-4	r14_usr
-8	r13_usr
-12	r12_usr
-16	r11_usr
-20	r10_usr
-24	r9_usr
-28	r8_usr
-32	r7_usr
-36	r6_usr
-40	r5_usr
-44	r4_usr
-48	r3_usr
-52	r2_usr
-56	r1_usr
-60	r0_usr
-64	r14_irq
-68	SPSR

Figura 3.2: Estrutura de dados do PCB. Fonte: (SLOSS, 2001)

T1	T2	T3	T4	T5	T6	T7	T8	T9
1	1	0	1	0	0	0	0	0

Figura 3.3: Vetor de threads.

3.3 Configuração de hardware e software

Nesta seção são apresentados os modos como o hardware e o software descritos anteriormente são utilizados. Será indicado como foi feito o particionamento da memória, a utilização dos modos do processador e os modos de teste do código.

3.3.1 Memória

A memória volátil da placa foi estruturada como indicado na figura 3.4. Para todo espaço das pilhas, programas, código, vetor de interrupções e área de dados, o espaço disponível é de 128KB (de 0x0 a 0x20000). Como pôde ser visto anteriormente, a memória entre 0x0 e 0x20 contém o vetor de interrupções e deve ser reservado. A pilha do modo SVC começa no endereço 0x7F80, cresce para baixo e não deve invadir a área reservada para o vetor de interrupção. Já a pilha do modo IRQ, começa no endereço 0x8000, também cresce para baixo e não deve invadir o espaço reservado para a pilha do modo SVC. O código do kernel e dos

programas começa no endereço 0x8000, mas ao contrário da pilha do modo SVC, cresce para cima. Logo após o código, temos uma área reservada para os dados globais. Finalmente, as pilhas de usuário começam no endereço 0x20000 e crescem para baixo. Cada uma tem um offset relativo à anterior de 4048 bytes.

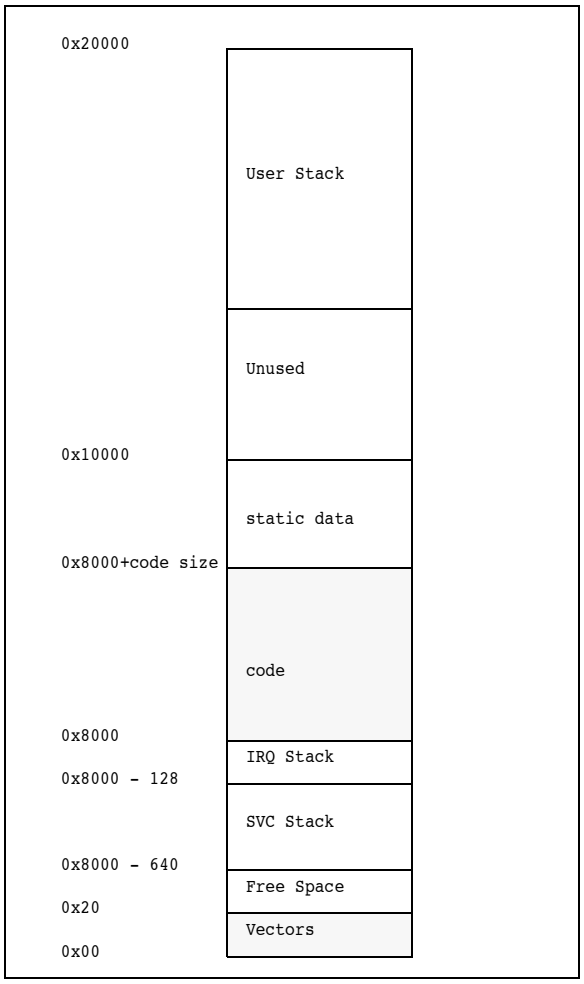

Figura 3.4: Estrutura da memória. Fonte: (SLOSS, 2001)

3.3.2 Modos do processador

Dentre os sete modos disponíveis na placa, apenas quatro deles são utilizados: o modo d usuário (user), o modo de serviço (SVC), o modo de sistema (SYS) e o modo de interrupção (IRQ). O primeiro é o modo não privilegiado no qual os processos são executados. O segundo, é o modo de inicialização do kernel e de execução das system calls, que é privilegiado. Já o terceiro, é idêntico ao modo de usuário, mas com privilégios. Ele é utilizado na inicialização do sistema para definir a pilha do modo de usuário. Finalmente, o quarto é um modo que também é privilegiado, mas que é usado quando há interrupções de hardware e portanto, é usado quando há o chaveamento de processos (interrupção de timer) ou qualquer outra interrupção

que não a de software. É importante ressaltar que os modos privilegiados quando chamados por interrupção desabilitam outras interrupções. Isso não permite que ocorra interrupções aninhadas, essencial para o funcionamento do código.

3.3.3 Modos de teste

Depurar o código com a placa não é possível em todas as situações. Quando o código que está sendo executado está dentro de uma região onde as interrupções estão desabilitadas, como no código de tratamento de interrupção, não se pode fazê-lo. Para contornar tal problema, foi utilizado o emulador disponível na IDE CodeWarrior, o ARMulator. Como ele foi desenvolvido para vários modelos de placa, utiliza endereços de periféricos diferentes da placa Evaluator 7-T, e além disso, não têm o módulo Angel de debug. Para manter a compatibilidade entre o emulador e a placa, nas partes onde o código se diferencia, como na inicialização do timer, foram colocados ambos códigos. A seleção de qual dos dois será executado depende de uma variável global emulator, que é declarada no arquivo constants.h. Caso seja 1, o código executado é o do emulador, caso seja 0, é o código da placa com Angel e caso seja 2 é o código para a placa com o Angel desabilitado. Uma outra vantagem do código no emulador é que ele permite com que ele possa ser testado sem a presença da placa.

3.3.4 Angel

O Angel é um programa contido na ROM da placa que realiza a comunicação entre a mesma e o computador que efetuou o upload do código. Além de permitir com que o código seja carregado na placa, o Angel realiza o processo de debug do código durante a execução. Para isso, deve haver uma comunicação constante entre a placa e o computador, que é feita através de interrupções. Uma vez que a placa é iniciada, o endereço do vetor de interrupções responsável pelas interrupções de hardware e se software apontam para um endereço pré-estabelecido do Angel. Caso se queira adicionar alguma outra rotina de tratamento de interrupções, deve-se encadear a rotina do Angel para que a comunicação com a placa não seja perdida.

3.4 Inicialização

O início do programa se dá no arquivo assembly statup.s. Nele, são feitas todas as operações que não podem ser feitas no código em C, como a inicialização das pilhas ou a

criação da tabela de threads. Após esta etapa, há a inicialização em C, feita no arquivo cinit.c, que inicializa periféricos, instala rotinas de tratamento e inicia a primeira thread em modo usuário. A rotina completa de inicialização pode ser vista no esquema da figura 3.5.

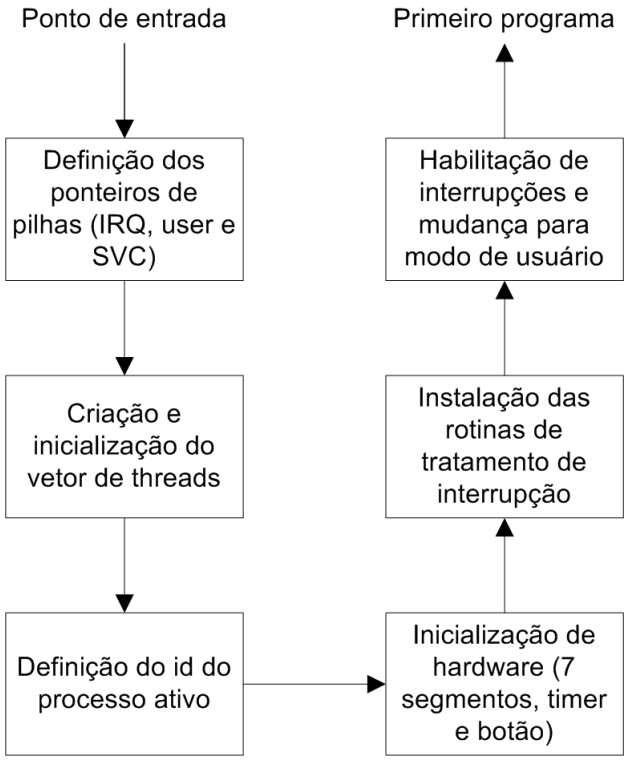

Figura 3.5: Fluxograma de inicialização.

3.4.1 Ponto de entrada e tipo de código

O ponto de entrada do código é indicado pela instrução `ENTRY`. Por padrão, o compilador assume que o código de entrada é ARM. Como descrito anteriormente, há dois tipos de assembly, o ARM e o THUMB. No microkernel, é utilizado apenas código ARM, já que ele fornece mais instruções e favorece a legibilidade. Um ponto negativo deste tipo de código é seu maior espaço ocupado na memória, mas isso não vem a ser um grande problema, pois temos espaço suficiente.

3.4.2 Pilhas

Antes de poder utilizar as pilhas é preciso que elas sejam inicializadas em cada um dos modos que virão a ser utilizados. Neste microkernel, são utilizados os modos de serviço, usuário/sistema e de interrupção. O modo como isto é feito é descrito abaixo:

```
  MOV    r0, #0xC0|0x12     ; r0 = 0xC0 or 0x12 (0xC0 = IRQ disabled, 0x12 =
     IRQ mode)
  MSR    CPSR_c, r0         ; status_register = r0
  MOV    sp, #0x8000        ; stack pointer = 0x8000
```

A primeira instrução copia para r0 o que será substituído no registrador de estado. Neste exemplo, está se desabilitando as interrupções e mudando o modo do processador para o modo de interrupção. Em seguida, os dados do registrador r0 são colocados no registrador de estado. Uma vez que o estado foi alterado, pode-se mudar o ponteiro de pilha, que neste caso aponta para o endereço 0x8000. Uma operação semelhante pode ser feita tanto no modo de serviço quanto no modo de usuário, usando os endereços de pilha indicados anteriormente. Porém, se o estado for alterado para o modo de usuário fica impossível de se alterar o estado novamente. Para se resolver este problema, ao invés de se mudar para o estado de usuário, muda-se para o estado de sistema. Este é o mesmo modo que o de usuário (usa a mesma pilha e registradores), mas permite que o modo seja alterado novamente.

3.4.3 Vetor de threads e número da thread

O outro ponto importante da inicialização do código em assembly é a criação do vetor de threads. Para tal, temos de definir que todos os processos exceto o primeiro são inicializados desabilitados. Isto é feito com o código apresentado a seguir:

```
  ; Initializes the thread array with zeros (0 = thread disabled,
  ; 1 = thread enabled)
  LDR    r0, =thread_array    ; r0 = thread_array start address
  MOV    r1, #1               ; r1 = 1
  STR    r1, [r0]             ; address(r0) = r1
  MOV    r1, #0               ; r1 = 0 (disabled)
  MOV    r2, #0               ; r2 = 0
init_thread_array_loop
  ADD    r2, r2, #4           ; r2 = r2 + 4
  CMP    r2, #36              ; r2 = 36?
  BEQ    set_active_thread    ; if yes, go to set_active_thread
  ADD    r3, r0, r2           ; r3 = r0 + r2
  STR    r1, [r3]             ; address(r3) = r1
  B    init_thread_array_loop  ; return to init_thread_array_2
```

Nele, r0 armazena a base do vetor, que coincide com o espaço relativo à primeira thread. r1 contém o dado que será colocado na posição de memória. Na posição este valor é 1, e nos demais 0. r2 contém o offset que será somado à base para o cálculo do endereço absoluto,

armazenado em r3. O algoritmo funciona inicialmente colocando 1 na base. Após isso, entra em um loop que aumenta o offset de 4 em 4 e coloca 0 em todos os outros espaços.

Ainda na inicialização em assembly, deve-se definir o número da thread que está sendo executada. Este dado é armazenado na variável current_thread_id. Pode-se ver abaixo como é definido o id do primeiro processo para 1:

```
LDR   r0, =current_thread_id   ; r0 = current thread id address
MOV   r1, #1                   ; r1 = 1
STR   r1, [r0]                 ; current thread id = 1
```

Finalmente, a inicialização em C pode ser iniciada. A chamada é feita definindo como endereço de retorno a função C_entry e colocando este mesmo endereço no process counter.

```
LDR   lr, =C_Entry             ; link register = C entry
MOV   pc, lr                   ; process counter = C entry
```

3.4.4 Periféricos

Para alguns periférico da placa, como o display de sete segmentos, o timer e os botões, há uma rotina de inicialização que os habilita e define suas configurações. Suas chamadas são `segment_init()`, `timer_init()` e `button_init()` respectivamente. Estas funções se encontram nos arquivos de cada um dos periféricos e são executadas logo no início da etapa C do processo de inicialização da placa.

3.4.5 Instalação do tratamento de interrupção

Como descrito anteriormente, caso uma interrupção de hardware ocorra, a instrução no endereço 0x18 é executada e caso seja uma interrupção de software, a instrução no endereço 0x08. Toda vez que se reinicia a placa, são colocados nestes endereços uma instrução que realiza um desvio para a rotina Angel, descrita anteriormente.

Porém, se algum dos periféricos vai ser utilizado, a interrupção gerada por esse periférico não deve desviada para o Angel, e sim para uma rotina adequada. Para poder identificar qual a origem da interrupção e desviar para a rotina correta, devemos instalar uma nova rotina no vetor de interrupções, substituindo o desvio para o Angel. A instalação da rotina se dá através do desvio para a tal rotina. Todavia, não se pode apenas descartar o endereço do Angel, já que caso não se identifique a origem da interrupção, ainda deve-se desviar para ele. Este processo

pode ser observado na figura 3.6. Nele, *Handler2* é a rotina de tratamento de interrupções, e *Handler1* é o Angel.

A instalação da rotina de tratamento de interrupção é a mesma para interrupções de hardware e de software se dá abaixo:

```
/* Angel branch instruction */
unsigned Angel_branch_instruction;
/* Angel instruction */
unsigned *Angel_address;
/* Getting Angel branch instruction */
Angel_branch_instruction  = *vector_address;
/* Separate the instruction from the address */
Angel_branch_instruction ^= 0xe59ff000;
/* Calculating absolute address */
Angel_address = (unsigned *) ((unsigned)vector_address +
    Angel_branch_instruction + 0x8);
/* Store address in the propoer position */
if ((unsigned)vector_address == 0x18) {
  Angel_IRQ_Address = *Angel_address;
}
else {
  Angel_SWI_Address = *Angel_address;
}
/* Inserting handler instruction in the vector table */
*Angel_address = handler_routine_address;
```

Os parâmetros de entrada desta função são `handler_routine_address`, o endereço da rotina de tratamento de interrupção e `vector_address`, um ponteiro para a posição no vetor de interrupções onde será instalada a rotina. Sucintamente, o que esta rotina realiza é obter a instrução que está em `vector_address`, aplica uma máscara à rotina para obter apenas o endereço e o salva em uma das variáveis: `Angel_IRQ_Address` caso se esteja instalando a rotina de interrupção de hardware ou `Angel_SWI_Address` caso seja a de software, além de colocar a nova instrução no vetor de interrupções.

Um fator importante que deve ser ressaltado a importância do Angel quando se está usando a placa. Como já descrito anteriormente, o Angel se utiliza das interrupções de hardware e software para se comunicar com a placa. Portanto, se apenas modificarmos o código e substituirmos a instrução que está contida no vetor de interrupção, essa comunicação não se realiza e tanto a placa quanto o programa de debugger travam. Para solucionarmos este problema, devemos passar para a rotina de tratamento de interrupção os endereços que estavam ante-

riormente no vetor de interrupção, para o caso da interrupção ser do Angel, a rotina correta ser executada. Já no caso em que o código é apenas simulado no emulador, não é preciso armazenar o endereço do Angel.

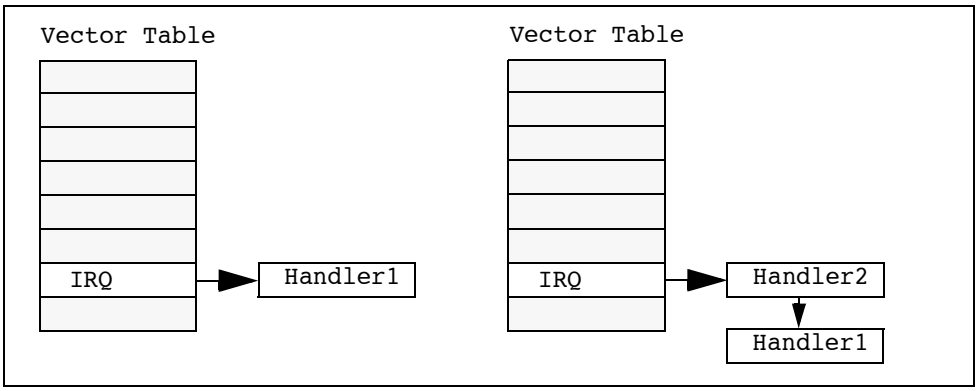

Figura 3.6: Encadeamento de interrupções. Fonte: (SLOSS, 2001)

3.4.6 Interrupção de timer

A interrupção de timer é utilizada neste projeto para realizar o chaveamento entre as threads. Uma vez que haja a interrupção, o estado da thread atual é salva e a próxima thread é colocada em processamento. Para utilizá-la, devemos tanto habilitar quanto iniciar o timer. Essas tarefas são executadas com duas rotinas, sendo que a primeira já foi descrita anteriormente. Já o início do timer é dado pela função `timer_start()`.

3.4.7 Habilitando interrupções

O último passo antes de se começar a executar o código do primeiro programa é habilitar simultaneamente o modo de usuário e as interrupções. Como isso só pode ser feito por código assembly, temos de usar a instrução especial de C __asm, conforme o exemplo abaixo

```
__asm {
  MOV    r1, #0x40|0x10
  MSR    CPSR_c, r1
}
```

O registrador r1 recebe 0x40, que indica a habilitação das interrupções e 0x10 que altera para o modo de usuário. Logo em seguida, o conteúdo deste registrador é passado para o registrador de estado. Finalmente, o primeiro programa é chamado com a função `shell()`.

3.5 Processos

O kernel pode lidar com no máximo nove processos, nomeados de task1 a task9 no arquivo tasks.c. Como eles não têm área de dados própria, não pode-se chamá-los de processos. A implicação de se ter uma área de dados em comum é que todos os processos que rodam um mesmo programa compartilham os valores das variáveis. O mais correto, portanto, seria o chamá-os de threads.

Criamos algums programas exemplo que se utilizam dos periféricos da placa.

TODO... (Fazer código antes)

3.6 Chaveamento de processos

O chaveamento de processos é realizado inteiramente com o assembly escrito no arquivo handler_irq.s. Ele consiste em sete passos, indicados na figura 3.7.

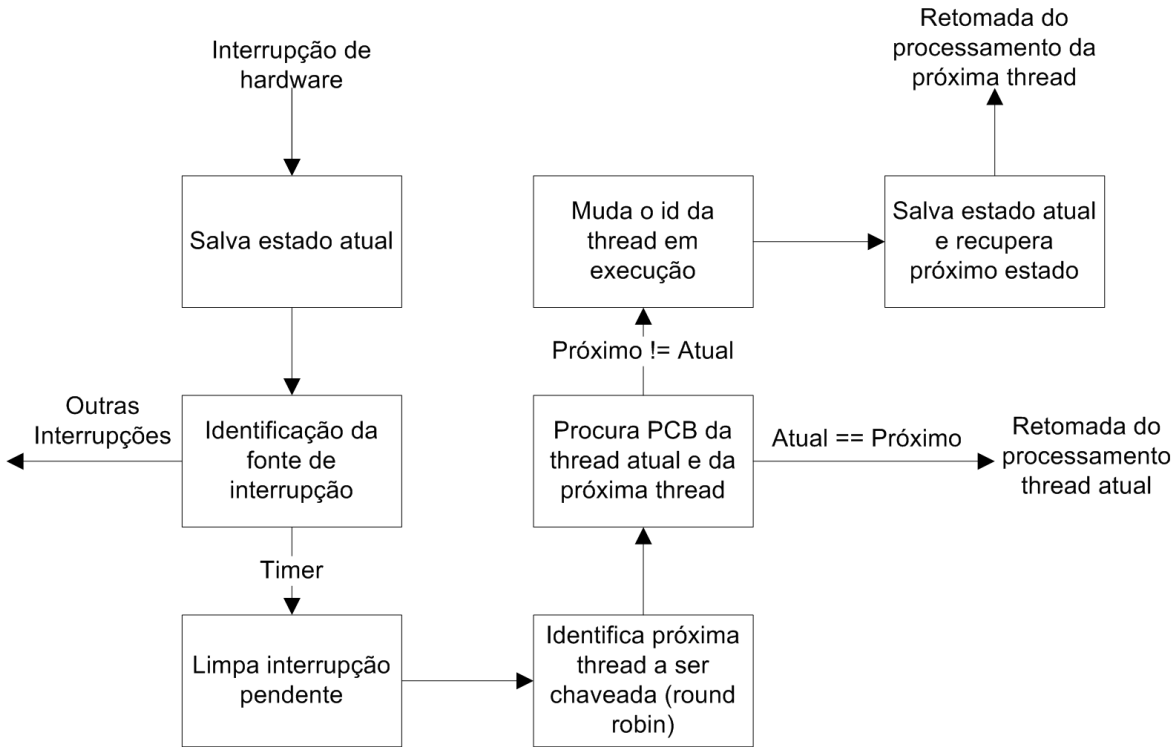

Figura 3.7: Chaveamento de processos.

3.6.1 Identificação da interrupção

```
STMFD sp!, {r0 - r3, lr}     ; Stacking r0 to r3 and the link register
```

```
LDR   r0, IRQStatus      ; r0 = irq type address
LDR   r0, [r0]          ; r0 = irq type
TST   r0, #0x0400        ; irq type == 0x0400?
BNE   handler_timer        ; If yes, go to handler_timer
TST   r0, #0x0001        ; irq type = 0x0001?
BNE   handler_button       ; If yes, go to handler_button
LDMFD sp!, {r0 - r3, lr}     ; If it is not any of them, restore r0-r3 and
    lr
LDR   pc, Angel_IRQ_Address ; and branch to the Angel routine
```

Uma vez que há a interrupção de timer, a chamada de interrupção de hardware que se encontra no vetor de interrupção é executada. Durante a instalação da rotina de tratamento de interrupção de hardware, colocou-se nesta posição a rotina handler_board_angel caso se estivesse usando a placa com o Angel, a rotina handler_board_no_angel caso se estivesse usando a placa sem o Angel ou a rotina handler_emulator caso estivesse usando o emulador. A direrença é que enquanto a primeira e a segunda tentam identificar qual a fonte de interrupção, a terceira já assume que a fonte é o timer, já que não há outros periféricos no emulador. Deve-se armazenar toda informação contida nos registradores que sao alterados durante o processo de tratamento de interrupção. Para tal, empilhamos os valores dos registradores r0 a r3, usados durante a rotina de chaveamento, a fim de que nenhum dado se perca durante o processo.

No caso do uso da placa, a fonte da interrupção se encontra no endereço 0x03ff4004, identificado com a variável `INTPND`. Se o valor contido neste endereço é 0x0400, a fonte foi uma interrupção de timer, caso seja 0x0001, a fonte foi o botão da placa e caso contrário, a fonte foi o Angel. No primeiro caso, há um desvio para a rotina `handler_timer`, no segundo para a rotina `handler_button` e na terceira, para o endereço salvo durante a instalação de rotina de tratamento.

3.6.2 Limpeza da interrupção de timer

Quando é identificada a interrupção de timer, deve-se limpar a interrupção de timer, a fim de que ele possa interromper novamente no futuro. Para tal, executa-se a rotina timer_irq, encontrada no arquivo timer.c. Como não podemos garantir que a rotina em C manterá intactos os registradores, temos de salvar todos e recupera-los após a chamada. Abaixo podemos observar o código que realiza o salvamento e a recuperação destes registradores.

```
STMFD sp!, {r4 - r12}      ; Stack the rest of the registers (r4-r12)
BL    timer_irq         ; Clear timer interruption
LDMFD sp!, {r4 - r12}      ; Load r4-12 registers again
```

Os registradores r0 a r3 não precisam ser salvos ou recuperados, pois no início da rotina de tratamento eles já foram empilhados para recuperação futura.

3.6.3 Identificação da próxima thread

O método de escolha da próxima thread que será posta em execução é escolhida pelo método *round-robin*, ou seja, a próxima thread é escolhida por ordem numérica. O código para tal tarefa é apresentado abaixo:

```
  CMP   r0, #9          ; r0 == 9? (it is the last thread?)
  BEQ   last_thread       ; If yes, branch last_thread
  ADD   r1, r0, #1        ; If not, r1 = r0 + 1
  B   next_thread       ; and branch to next_thread
last_thread
  MOV   r1, #1          ; r1 = 1
next_thread
  SUB   r2, r1, #1        ; r2 = r1 - 1
  MOV   r3, #4          ; r3 = 4
  MUL   r2, r3, r2        ; r2 = r2 * r3
  LDR   r3, =thread_array   ; r3 = thread_array bottom address
  ADD   r2, r2, r3        ; r2 = r3 + r2
  LDR   r2, [r2]        ; r2 = thread array content
  CMP   r2, #1          ; thread array content = 1?
  BEQ   set_addresses     ; If yes, branch to set_addresses
                ; Send to the next step the next active
                ; thread in r1
  MOV   r0, r1          ; If not, r0 = r1
  B   get_next_taskid_loop  ; and loop to get_next_taskid_loop
```

Nele, r0 inicia com o número da thread atual. Caso ele seja igual a 9, a última thread da lista, deve-se iniciar novamente a procura desde a thread 1. Caso contrário, inicia-se com o próximo número. O resultado é armazenado em r1, onde se encontra o número da próxima thread. O valor em r1 é incrementado sucessivamente até encontrar um ponto no vetor de threads que tenha o valor 0, indicando que a thread não está ativa. O cálculo da posição de memória é dado a partir da seguinte função: $(r1-1)*4+bottom =$ posição relativa à thread r1, onde bottom é o endereço do início do vetor e 4 é o tamanho de cada espaço dentro do vetor.

3.6.4 Localização dos PCBs

A rotina de troca de processos tem como entrada duas variáveis: o PCB da thread atual e o PCB do próxima thread. Para obter tais dados, é necessário o número da thread atual e da thread que será colocada em execução. Como visto nos itens anteriores, estes dados já foram obtidos. Pode-se então aplicar o seguinte algoritmo:

```
  LDR   r2, =current_thread_id    ; r2 = current thread id address
  LDR   r2, [r2]          ; r2 = current thread id
  CMP   r2, r1            ; Is r2 = current thread id ==
                  ; next thread id
  BEQ   no_thread_switch      ; If yes, branch to no_thread_switch
; Setting current_task_addr
  MOV   r0, #68           ; Else start thread switch. r0 = 68
  MUL   r0, r2, r0          ; r0 = current thread id * 68
  LDR   r2, =process_control_block  ; r2 = PCB bottom
  ADD   r0, r0, r2          ; r0 = PCB bottom + id * 68
  LDR   r2, =current_task_addr    ; r2 = current task addr addr
  STR   r0, [r2]          ; current_task_addr = r0
; Setting next_task_addr
  MOV   r0, #68           ; r0 = 68
  MUL   r0, r1, r0          ; r0 = next thread id * 68
  LDR   r2, =process_control_block  ; r2 = PCB_bottom
  ADD   r0, r2, r0          ; r0 = PCB bottom + next id * 68
  LDR   r2, =next_task_addr     ; r2 = next_task_addr addr
  STR   r0, [r2]          ; next_task_addr = r0
```

O primeiro ponto checado é se a thread atual é igual à thread que vai ser substituída. Caso isso se confirme, o chaveamento se encerra e nada ocorre. Caso contrário, o cálculo dos endereços dos PCBs é iniciado. A fórmula utilizada é: $PCB_{id} = (id - 1) * 68 + base$, onde id é o número da thread e base é o endereço do início dos PCBs. Ao fim do cálculo, estes dados são armazenados nas variáveis `current_task_addr` e `next_task_addr`, que serão utilizadas na próxima etapa do processo.

3.6.5 A troca de processos

A troca de processos se dá em poucos passos usando-se instruções especiais que permitem que haja um grande número de dados empilhados/desempilhados com apenas uma instrução. Inicialmente zera-se a pilha do modo de interrupção e restabelece-se os registradores r0 a r3, que estavam empilhados desde o do início da rotina de tratamento. Nota-se que o ponteiro

não é totalmente zerado, ele é colocado em uma posição 20 bytes acima do esperado. Isto se dá porque há empilhadas 5 palavras (r0 a r3 e o link register) que logo em seguida virão a ser desempilhadas.

Depois disso, muda-se o endereço do ponteiro de pilha para o PCB do processo atual. Um truque vem no próximo passo: empilha-se todos os registradores com o ponteiro de pilha apontando para a posição (base - 60) do PCB. Deste modo, em uma única instrução todos os registradores são colocados em suas respectivas posições. Como a estrutura do PCB foi feita tendo este processo em mente, a posição dos dados dos registradores cai exatamente como foi descrito na figura 3.2. Após o armazenamento do estado atual, muda-se novamente o endereço do ponteiro de pilha para o PCB da próxima instrução. Do mesmo modo que o armazenamento, desempilha-se os o valor dos registradores, que são exatamente como estava empilhado este processo quando foi armazenado.

```
; Reset and save IRQ stack
  LDR   r0, =irq_stack_pointer    ; r0 = irq_stack_pointer addr
  MOV   r1, sp            ; r1 = irq stack pointer
  ADD   r1, r1, #5*4        ; r1 = irq stack pointer + 5 (# of data in
                  ; the stack, r0-r3, lr) * 4 (size of a word)
  STR   r1, [r0]          ; irq_stack_pointer = irq stack pointer
                  ; without the data that will be removed next
  LDMFD   sp!,{r0-r3,lr}      ; Restore the remaining registers
; Load and position r13 to point into current PCB
  LDR   r13, =current_task_addr   ; r13 = current task PCB bottom address
     address
  LDR   r13, [r13]          ; r13 = current task PCB bottom address
  SUB   r13, r13,#60        ; r13 = current task PCB bottom address - 60
                  ; to point to the right place for the stacking
                  ; (next step)
; Store the current user registers in current PCB
  STMIA   r13, {r0-r14}^        ; Stacks the r0-r14 registers in the PCB
  MRS   r0, SPSR          ; r0 = status register
  STMDB r13, {r0,r14}       ; Stacks r0 and r14
;Load and position r13 to point into next PCB
  LDR   r13, =next_task_addr    ; r13 = next task PCB bottom address
     address
  LDR   r13, [r13]          ; r13 = next task PCB bottom address
  SUB   r13, r13,#60        ; r13 = next task PCB bottom address - 60
                  ; to point to the right place for the stacking
                  ; (next step)
; Load the next task and setup PSR
  LDMNEDB r13, {r0,r14}       ; Restore r0 and r14 (IRQ mode)
```

```
  MSRNE   spsr_cxsf, r0        ; Restore status register
  LDMNEIA r13, {r0-r14}^       ; Restore r0-r14 for the user mode
  NOP                  ; NOP! (required for the above instruction)
; Load the IRQ stack into r13_irq
  LDR   r13, =irq_stack_pointer   ; r13 = stack pointer address address
  LDR   r13,[r13]          ; Restore previous stack pointer
  B   return             ; Go to the end
```

3.6.6 Retorno à execução da nova rotina

Como os registradores, o ponteiro de pilha, o endereço de retorno e o registrador de estados já estão com os dados do próximo processo, deve-se apenas fazer com que o a instrução imediatamente posterior à aquela executada antes da interrupção seja executada. Porém, o pipeline do processador fez com que o endereço da instrução duas vezes à frente tivesse sido armazenado. Para compensar isso, deve-se subtrair o tamanho de uma instrução (4 bytes) do endereço que vai ser colocado no process counter. Todo este processo é feito com apenas uma instrução: `SUBS  pc, r14, #4`, que simultaneamente decrementa do endereço de retorno 4 e coloca o resultado no process counter.

3.7 System calls

Uma system call é uma interrupção de software causada pelo kernel para a execução de código que necessita de privilégios para ser executado. Como uma interrupção de hardware, uma vez que é causada, ela executa a instrução apontada no vetor de interrupções, que foi instalada anteriormente na inicialização do sistema. A rotina de tratamento está localizada no arquivo handler_swi.s e é executada em modo SVC. As únicas instruções que chamam tais system calls são as rotinas fork, exec e exit.

3.7.1 Propriedades gerais

Uma vez que uma system call é chamada, umas das funções encontradas em `swi.c` é invocada. O motivo para este passo intermediário é que todas as chamadas de sistema do kernel devem ter a mesma identificação junto à rotina de tratamento. Neste caso, todas são passadas com o primeiro parâmetro como 0. Além disso, todas devem passar o mesmo número de parâmetros, pois todas estão invocando a mesma função, chamada de `syscall` que também é realizado nesta etapa.

Uma vez que a chamada `syscall` é feita, ocorre uma interrupção de software. O procedimento que se passa neste caso é muito parecido com o de uma interrupção de hardware.

```
STMFD   sp!,{r0-r12,lr}      ; Stack registers r0-12 and link register
LDR   r0,[lr,#-4]        ; Calculate address of SWI instruction (r0 = lr-4)
BIC   r0,r0,#0xff000000    ; Mask off top 8 bits of instruction to give SWI
                ; number
LDR   r1, Angel_SWI_Number  ; r1 = Angel SWI Number
CMP   r0, r1            ; Compare SWI number to angel interrupt number
BEQ   goto_angel          ; If it is angel interrupt, branch to goto_angel
MOV   r1, #0            ; r1 = 0
CMP   r0, r1            ; Compare SWI number to r1
BEQ   os_swi            ; If it is OS SWI, branch to os_swi
```

Novamente há uma rotina de identificação da fonte de interrupção, que pode vir a ser uma do sistema operacional, ou do Angel. O primeiro passo desta rotina é o empilhamento de todos os registradores, para poder futuramente restaurar o estado atual. Em seguida, ocorre a identificação em si, onde uma máscara de bits é aplicada para se obter o identificador da interrupção. Caso ela seja Angel_SWI_Number, o estado do processador é restaurado e há um desvio para a instrução previamente armazenada durante a instalação. Caso ela seja 0, o valor estabelecido para o sistema, há um desvio para outro código que identifica quais das chamadas de sistema foi ativada.

Esta nova identificação pode ser observada abaixo. O primeiro passo é restaurar e armazenar novamente os valores dos registradores, já que na arquitetura ARM os valores passados pelos parâmetros de uma função são passados pelos primeiros registradores. Neste caso, r1 contém o tipo da chamada. Dependendo de qual for o valor, há desvios para `pre_routine_fork`, `pre_routine_exec` e `pre_routine_exit`

```
LDMFD sp!,{r0-r12,lr}    ; Restore r0-r12 registers and link registers
STMFD   sp!,{r0-r12,lr}    ; and stores them again (in order to clean the
    registers)
MOV   r1, #0          ; r1 = 0
CMP   r0, r1          ; Compare the first parameter to 0
BEQ   pre_routine_fork   ; If it is equal, branch to the fork
MOV   r1, #1          ; r1 = 1
CMP   r0, r1          ; Compare the first parameter to 1
BEQ   pre_routine_exec   ; If it is equal, branch to the exec
MOV   r1, #2          ; r1 = 2
CMP   r0, r1          ; Compare the first parameter to 2
BEQ   pre_routine_exit   ; If it is equal, branch to the exit
LDMFD sp!,{r0-r12,pc}^   ; If it is an unidentified syscall, go back to the
```

```
program,
            ; restoring the registers and putting the return address in
            ; the process counter
```

3.7.2 fork

TODO ...

Em um sistema operacional, a system call fork é responsável pela criação de novos processos. Para tal, ela duplica o processo que a criou, onde o único meio de se identificar qual o processo pai é pelo número de retorno. Caso o número de retorno seja 0, significa que este é o processo filho, e caso seja qualquer outro número, é o processo pai que retornou o identificador do processo filho.

Nosso fork teve de ser ligeiramente alterado por causa de uma simplificação que fizemos em nosso kernel. Como dito anteriormente, temos uma área de dados única para todos os processos. Com isso, fica impossível de se duplicar a área de dados de um processo, o que não fazemos.

O processo de duplicação de um processo se inicia com o empilhamento dos registradores de dados (r0 a r12) e do endereço de retorno (link register) por duas vezes. O motivo é que o primeiro empilhamento serve para a restauração do estado ao fim do processo de duplicação e a segunda para o processo que vai a ser duplicado. Então, tentamos encontrar qual o primeiro espaço disponível dentro da tabela de processos. Uma vez encontrado o espaço, temos de encontrar o espaço do PCB reservado para este processo, onde iremos popular com os dados do estado em execução. Porém, além disso, temos também de duplicar a pilha, que é um processo um pouco mais complexo. Para tal, primeiro temos de descobrir o tamanho da pilha atual. Então, começamos a copiar os dados de uma pilha para a outra. Finalmente, colocamos no ponteiro de pilha do PCB do novo processo o topo da pilha recém copiada. Uma vez resolvido o problema da cópia de pilha, apenas duplicamos os dados do registrador de estados, do link register, do process counter e de todos os registradores de dados. Finalmente, quando o processo está totalmente copiado, devemos habilitar o processo na tabela de processos e restaurar todos os registradores empilhados de volta ao seus lugares, onde o link register entrará no lugar do process counter.

3.7.3 exec

A chamada de sistema *exec* é responsável por substituir a imagem núcleo de um processo pela imagem do programa passado como argumento (TANENBAUM; WOODHULL, 2006).

Nos sistemas operacionais tradicionais, como o Linux ou o Minix, o *exec* é utilizado para iniciar um novo programa no mesmo ambiente do programa que executa a chamada de sistema. Normalmente o *exec* é utilizado na criação de um novo processo da seguinte maneira: um processo já existente se duplica através da chamada de sistema *fork*. O processo filho tem, então, seu código substituído pelo código que deve ser executado através da chamada de sistema *exec*, que permite ao processo filho assumir seu próprio conteúdo, apagando de si o conteúdo do processo pai.

No KinOS, para que um *thread* passe a executar outro programa, é necessário reinicializar o seu PCB, isso é feito pela chamada de sistema *exec*.

Existem 4 principais entradas do PCB que necessitam ser reinicializadas:

- o *program counter* (PC - R13);
- o *link register* (LR - R14);
- o *stack pointer* (SP - R15);
- e o *saved processor status register* (SPSR).

Para reinicializar essas entradas, de forma que a *thread* passe à executar um novo programa, primeiro é necessário calcular o início do PCB da *thread* correspondente.

A rotina *exec*, recebe como parâmetros o id da *thread* que será alterada e o ponteiro para a função/programa que pretende-se executar, como mostrado a seguir:

```
void exec(int process_id, pt2Task process_addr);
```

Assim para calcular o endereço inicial do PCB, obtêm-se o endereço inicial da área reservada para armazenar todos os PCBs, a **process_control_block**, e adiciona-se à esta o valor de 68 multiplicado por **process_id**, visto que cada PCB ocupa um espaço de 68 endereços de memória como mencionado na sessão 3.2.1. O código responsável por calcular o PCB é apresentado a baixo:

```
LDR r3, =process_control_block  ; r3 = the start address of the PCB area
MOV r4,#68                      ; r4 = 68 (space for each process in the PCB)
```

```
MUL r5,r1,r4        ; r5 = (task id) * 68
ADD r3,r3,r5        ; r3 = PCB start address + r5
```

Em seguida, calculado o endereço inicial do PCB, altera-se suas entradas da seguinte maneira:

- LR (PCB[-4]) e PC (PCB[-64]) recebem o endereço da primeira instrução do novo programa (**process_addr**).

```
PCB[-4] = process_addr;
PCB[-64] = process_addr;
```

- SP (PCB[-8]) recebe o endereço de início da pilha da *thread*, fazendo com que esta seja zerada. Para cada pilha de *thread*, 4048 bytes são reservados.

```
PCB[-8] =  início da pilha do modo usuário - (4048 * thread id);
```

- SPSR (PCB[-68]) recebe 0x10, pois os programas devem rodar no modo usuário.

```
PCB[-68] = 0x10;
```

Finalmente, após alterar as entradas mostradas a cima, a *thread* começa a executar o novo programa.

3.7.4 exit

A chamada de sistema *exit* é responsável por finalizar um processo, liberando espaço de memória para a execução de um novo processo (TANENBAUM; WOODHULL, 2006).

No KinOS isso é realizado apenas colocando como desativado (igual à 0) o byte na lista de processos que corresponde a *thread* que se deseja finalizar.

Para isso a rotina *exit* recebe como parâmetro o id da *thread* a ser terminada.

```
void exit(int process_id);
```

3.8 Shell

TODO...

3.9 Semáforos

TODO...

3.10 Inspiração

Grande parte do código foi retirada do código presente nos exemplos incluídos no CD de demonstração da placa. O principal deles, é o código *mutex*, desenvolvido por Andrew N. Sloss, de onde foi baseado o chaveamento de processos, a função de semáforo e as rotinas de manipulação de hardware.

TODO ...

REFERÊNCIAS BIBLIOGRÁFICAS

ABDELRAZEK, A. F. M. *Exception and Interrupt Handling in ARM*. [S.l.], Setembro 2006. Disponível em: <http://www.iti.uni-stuttgart.de/ radetzki/Seminar06/08_report.pdf>.

ARM LIMITED. *ARM7TDMI Data Sheet (ARM DDI 0029E)*. [S.l.], Agosto 1995. Disponível em: <http://www.eecs.umich.edu/ panalyzer/pdfs/ARM_doc.pdf>.

ARM LIMITED. *Application Note 25 - Exception Handling on the ARM (ARM DAI 0025E)*. [S.l.], Setembro 1996. Disponível em: <http://www.imit.kth.se/courses/2B1445/0304/material/Apps25vE.pdf>.

ARM LIMITED. *ARM7TDMI Technical Reference Manual (ARM DDI 0029G)*. Rev 3. [S.l.], Abril 2001. Disponível em: <http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.ddi0029g/index.html>.

ARM LIMITED. *ARM Architecture Reference Manual (ARM DDI 0100I)*. [S.l.], Julho 2005. Disponível em: <http://www.arm.com/miscPDFs/14128.pdf>.

FURBER, S. *ARM System-On-Chip Architecture*. 2. ed. [S.l.]: Addison-Wesley, 2000. ISBN 0-20167-519-6.

KINOSHITA, C. C. e. A. H. J. *Experiência 5: Interrupções*. [S.l.], 2007. Disponível em: <http://www.pcs.usp.br/ jkinoshi/2007/tomas5.doc>.

MORROW, M. G. *ARM7TDMI Instruction Set Reference*. [S.l.], Setembro 2008. Disponível em: <http://eceserv0.ece.wisc.edu/ morrow/ECE353/arm7tdmi_instruction_set_reference.pdf>.

RYZHYK, L. The arm architecture. Junho 2006. Disponível em: <http://www.cse.unsw.edu.au/ cs9244/06/seminars/08-leonidr.pdf>.

SLOSS, A. *Interrupt Handling*. [S.l.], Abril 2001.

SLOSS, A.; SYMES, D.; WRIGHT, C. *ARM System Developer's Guide: designing and optimizing system software*. 1. ed. [S.l.]: Morgan Kauffman, 2004. ISBN 1-55860-874-5.

TANENBAUM, A. S.; WOODHULL, A. S. *Operating Systems: Design and Implementation*. 3rd. ed. [S.l.]: Pearson Prentice Hall, 2006. ISBN 0-13-142938-8.

ZAITSEFF, J. *ELEC2041 Microprocessors - Laboratory Manual*. [S.l.], Junho 2003. Disponível em: <http://www.zap.org.au/elec2041-cdrom/unsw/elec2041/README.html>.