

**FELIPE GIUNTE YOSHIDA
MARIANA RAMOS FRANCO
VINICIUS TOSTA RIBEIRO**

MICROKERNEL PARA PROCESSADOR ARM7

São Paulo
2009

**FELIPE GIUNTE YOSHIDA
MARIANA RAMOS FRANCO
VINICIUS TOSTA RIBEIRO**

MICROKERNEL PARA PROCESSADOR ARM7

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 PROCESSADOR ARM7

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 PROCESSADOR ARM7 / F.G. Yoshida, M.R. Franco, V.T. Ribeiro. – São Paulo, 2009. 39 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

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

2.1	Pipeline de 3 estágios	18
2.2	Pipeline do ARM7TDMI	19
2.3	Organização dos registradores no modo ARM	22
2.4	Formato dos registradores de estado CPSR e SPSR	24
3.1	Estrutura de dados do PCB. Fonte: (??)	30
3.2	Estrutura da memória. Fonte: (??)	32
3.3	Encadeamento de interrupções. Fonte: (??)	35

LISTA DE TABELAS

2.1	Modos de operação	20
2.2	Valores para o bit de modo	26
2.3	Valores para o bit de modo	27
2.4	Mapa da memória flash	28
3.1	Relação de arquivos com suas funções	29

LISTA DE ABREVIATURAS

B2B Business to Business

ED Especificação Deontica

EE Especificação Estrutural

EF Especificação Funcional

EnO Entidade Organizacional

EO Especificação Organizacional

ES Esquema Social

IA Inteligência Artificial

IAD Inteligência Artificial Distribuída

KQML Knowledge Query and Manipulation Language

MOISE Model of Organization for multi-agent SystEms

OO Orientação a Objetos

RDP Resolução Distribuída de Problemas

SMA Sistemas Multiagentes

TAEMS Task Analysis, Environment Modeling, and Simulation

LISTA DE SÍMBOLOS

\sim - indiferente a

\succ - melhor que

\succeq - melhor ou indiferente a

$\langle \mathbf{x}, \mathbf{y} \rangle$ - produto escalar entre os vetores \mathbf{x} e \mathbf{y}

\mathcal{A} - um conjunto de ações disponíveis

a - uma ação

SUMÁRIO

1	Introdução	13
1.1	Objetivo	13
1.2	Motivação	13
1.3	Justificativa	14
1.4	Metodologia de Trabalho	14
1.5	Organização do Documento	15
2	Conceitos e Tecnologias Envolvidas	17
2.1	O Processador ARM7TDMI	17
2.1.1	Arquitetura RISC	17
2.1.2	Pipeline	18
2.1.3	Estados de Operação	20
2.1.4	Modos de Operação	20
2.1.5	Registradores	21
2.1.6	Registradores de Estado	23
2.1.7	Interrupções	26
2.2	A Placa Experimental Evaluator-7T	27
2.2.1	Os programas Bootstrap Loader e Angel	28
3	O Sistema Operacional KinOS	29
3.1	Organização	29
3.1.1	Process Control Block	30
3.1.2	Lista de processos	31

3.1.3	Memória	31
3.1.4	Modos	31
3.1.5	Modos de operação	31
3.2	Angel	32
3.3	Inicialização	33
3.3.1	Modo	33
3.3.2	Pilhas	33
3.3.3	Tabela de processos	34
3.3.4	Periféricos	34
3.3.5	Instalação do tratamento de interrupção	34
3.3.6	Interrupção de timer	35
3.3.7	Habilitando interrupções	35
3.4	Processos	35
3.5	Chaveamento de processos	36
3.5.1	Identificação da interrupção	36
3.5.2	Recomeçar o timer	36
3.5.3	Identificação do PCBs da troca de processos	36
3.5.4	A troca de processos	37
3.5.5	Um caso especial	37
3.5.6	Retorno à execução da nova rotina	38
3.6	System calls	38
3.6.1	Propriedades gerais das system calls	38
3.6.2	fork	38
3.6.3	exec	39
3.6.4	exit	39
3.7	Shell	39

1 INTRODUÇÃO

1.1 Objetivo

O objetivo deste projeto de formatura é desenvolver um microkernel para a placa experimental Evalutator-7T, contituída 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 alguns drivers 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 do usuário 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 então em como aproximar essas duas disciplinas, surgiu a idéia de desenvolver uma ferramenta didática que unisse um hardware de estudo simples a um sistema operacional igualmente 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 cuja as 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, rodamos alguns exemplos 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, montamos 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 levantamos 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 visto 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.

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.

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 à 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.

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.

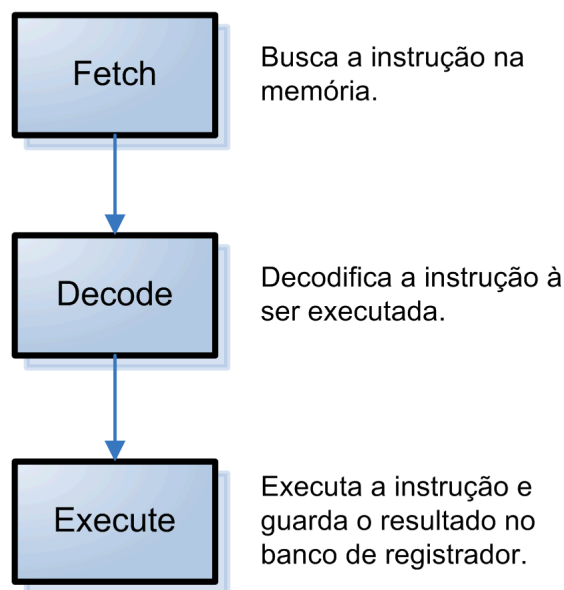


Figura 2.1: Pipeline de 3 estágios

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.

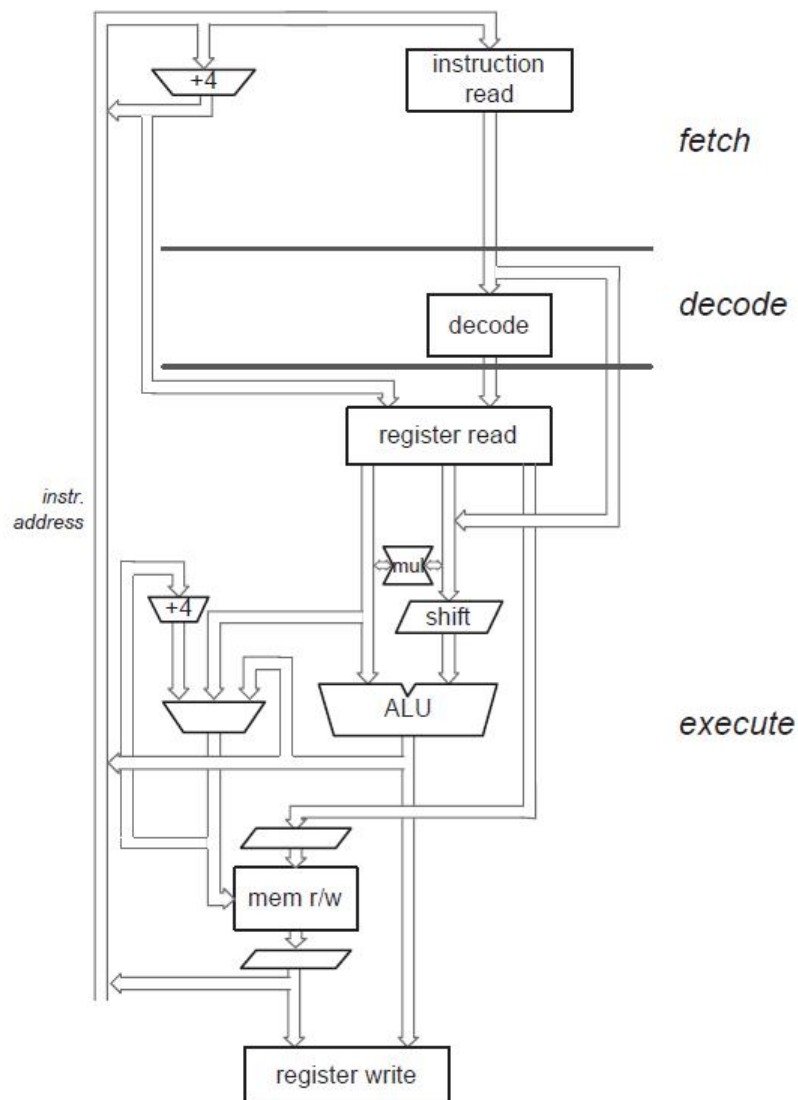


Figura 2.2: Pipeline do ARM7TDMI

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 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 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 encher 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: 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, decidimos pela utilização do processador 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:

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

Mudanças no modo de operação podem ser feitas 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 está no modo Usuário, o programa que está sendo executado não pode acessar alguns recursos protegidos do sistema ou mudar de modo sem ser através de uma interrupção.

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.

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 às operações que necessitam de acesso aos recursos do sistema, mas que 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.

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 adicional registrador, o CPSR (Current Program Status Register), contém os bits de flag e de modo.
















Os registradores R13 à R15 possuem as seguintes funções especiais:

- 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.

System & User	FIQ	Supervisor	Abort	IRQ	Undefined
R0	R0	R0	R0	R0	R0
R1	R1	R1	R1	R1	R1
R2	R2	R2	R2	R2	R2
R3	R3	R3	R3	R3	R3
R4	R4	R4	R4	R4	R4
R5	R5	R5	R5	R5	R5
R6	R6	R6	R6	R6	R6
R7	R7	R7	R7	R7	R7
R8	 R8_fiq	R8	R8	R8	R8
R9	 R9_fiq	R9	R9	R9	R9
R10	 R10_fiq	R10	R10	R10	R10
R11	 R11_fiq	R11	R11	R11	R11
R12	 R12_fiq	R12	R12	R12	R12
R13	 R13_fiq	 R13_svc	 R13_abt	 R13_irq	 R13_und
R14	 R14_fiq	 R14_svc	 R14_abt	 R14_irq	 R14_und
R15 (PC)	R15 (PC)	R15 (PC)	R15 (PC)	R15 (PC)	R15 (PC)

ARM State Program Status Registers

CPSR	CPSR	CPSR	CPSR	CPSR	CPSR
	 SPSR_fiq	 SPSR_svc	 SPSR_abt	 SPSR_irq	 SPSR_und


 = banked register

Figura 2.3: Organização dos registradores no modo ARM

Como se pode verificar na figura, 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).

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 R8-R12.

Esses registradores especiais mapeiam de um pra um os registradores do modo usuário. Se você mudar o modo do processador, um registrador particular do novo modo irá substituir o registrador existente.

Por exemplo, quando o processador está 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 R0 à R12.

2.1.6 Registradores de Estado

O Current Program Status Register (CPSR) 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 exceção possui também um Saved Program Register (SPSR), que é usado para preservar o valor do CPSR quando a exceção associada acontece.

Os registradores de estado do programa:

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

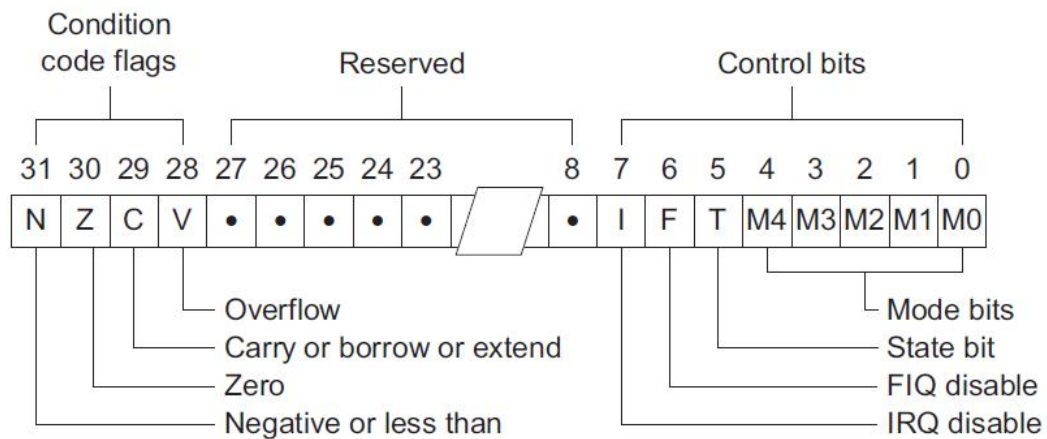


Figura 2.4: Formato dos registradores de estado CPSR e SPSR

2.1.6.1 Flags de Condição

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

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. Eles são:

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

Os bits de controle mudam quando uma interrupção acontece. Quando o processador está 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 limpo, 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 tome cuidado para usar somente as combinações mostradas a seguir:

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

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.

A arquitetura ARM suporta 7 tipos de interrupções. A figura a baixo 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.

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

Tipo de interrupção	Modo de operação	Endereço
Reset	Supervisor	0x00000000
Instrução indefinida	Indefinido	0x00000000
Interrupção de Software (swi)	Supervisor	0x00000000
10011	Abort	0x00000000
10111	Abort	0x00000000
Interrupção normal (IRQ)	IRQ	0x00000000
Interrupção rápida (FIQ)	FIQ	0x00000000

Tabela 2.3: Valores para o bit de modo

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 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

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 abaixo mostra a faixa de endereços de cada região da memória.

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

Tabela 2.4: 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

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 Os programas Bootstrap Loader e Angel

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

3 O SISTEMA OPERACIONAL KINOS

Lembretes:

- Escrever algo aqui
- Explicar em algum lugar as divisões que existem no assembly.
- Ver como colocar referencias e colocar a referencia das figuras
- Colocar a referencia das seções
- Variáveis em assembly

3.1 Organização

Nesta etapa descreveremos algumas das estruturas de dados em assembly que utilizamos em nosso projeto, assim como organizamos a memória na placa e utilizamos os modos. O código está estruturado da maneira indicada na tabela 3.1.

Arquivo	Função
startup.s	Inicialização em assembly ARM
cinit.c	Inicialização em C
button, dips, segment e timer	Rotinas de inicialização e uso dos periféricos
fork.s	Rotina de duplicação de processo em assembly
handler.s	Rotina de tratamento de interrupções em assembly
mutex.mcp	Arquivo de projeto do CodeWarrior
rpsarmul.h	Arquivo com as constantes da placa e do emulador
tasks	Processos a serem executados
irq e swi	Inicialização e controle das interrupções de HW e SW

Tabela 3.1: Relação de arquivos com suas funções

3.1.1 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 usando o processador. Temos um PCB para cada uma das nove threads, onde cada uma ocupa 68 bytes. Estes 68 bytes estão estruturados como explicitado na figura 3.1. Cada posição da tabela ocupa uma palavra (4 bytes). A primeira posição é a base do PCB menos quatro bytes, a segunda a base menos oito bytes e assim por diante. Como podemos observar pela figura, as posições um a quinze (base - 4 a base - 60) armazenam o conteúdo dos registradores r0 a r14 do modo user em ordem inversa. A posição dezesseis (base - 64) armazena o registrador quatorze do modo IRQ, que nada mais é o endereço de retorno (link register) da interrupção que realizou a troca de processos. Finalmente, a posição dezessete armazena o registrador de estado do modo user. Estes registradores armazenados permitem estabelecer um “snapshot” preciso do estado do processo quando houve o chaveamento, e permite que este estado seja restabelecido quando for o turno deste processo voltar a ser executado. Esta estrutura tem seu espaço reservado no arquivo handler.s, e é nomeado com a variável handler_task_bottom, que indica a base da estrutura. Cada um dos PCBs está logo a seguir do anterior. Por exemplo, para acessarmos o segundo PCB, devemos subtrair 68 do endereço de base, 136 bytes para o terceiro PCB e assim por diante.

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 3.1: Estrutura de dados do PCB. Fonte: (??)

3.1.2 Lista de processos

A lista de processos é um vetor que armazena quais dos processos estão ativos e quais não estão. Este vetor tem 40 bytes, onde 4 são para cada processo. Ele é declarado no arquivo `handler.s` com o nome de `Process_Table`, que indica sua base. Cada endereço reservado deve ter o valor 0, caso o processo esteja inativo ou 1, caso esteja ativo.

3.1.3 Memória

Estruturamos a memória utilizada como indicado na figura 3.2. Para todo espaço das pilhas, programas, kernel, vetor de interrupções e área de dados, vamos limitar o espaço disponível para 128KB (de 0x0 a 0x20000). Como pode ser visto na seção ??, 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.

3.1.4 Modos

Dentre os sete modos disponíveis na placa, utilizamos apenas três deles: o modo user, o modo SVC e o modo IRQ. O primeiro é o modo não privilegiado no qual rodamos os processos. O segundo, é o modo de inicialização do kernel e de execução das system calls, que é privilegiado. Já o terceiro, 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.

3.1.5 Modos de operação

Debugar 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 podemos debugar o programa na placa. Para contornar tal problema, utilizamos o emulador disponível na IDE CodeWarrior, o ARMu-

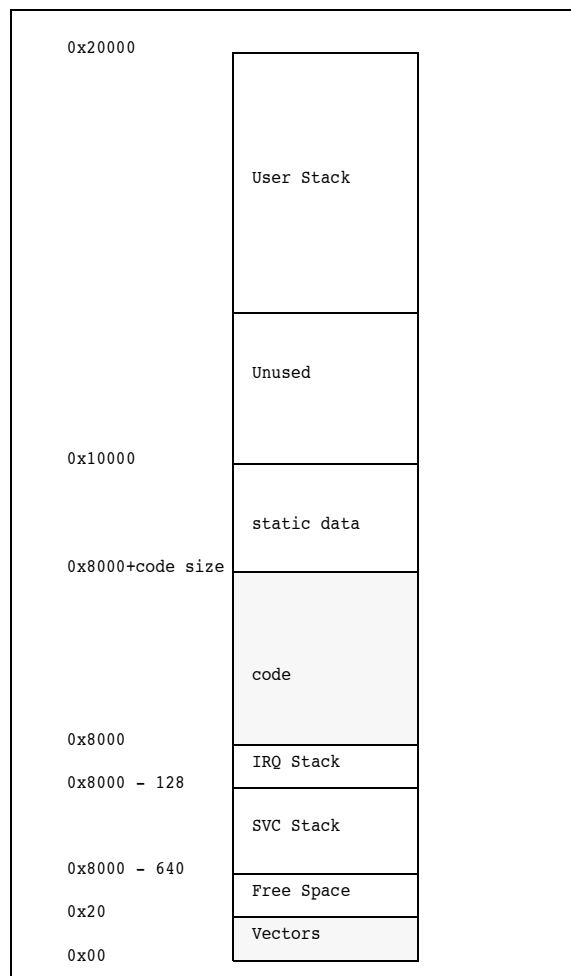


Figura 3.2: Estrutura da memória. Fonte: (??)

lador. 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, colocamos ambos códigos. A seleção de qual dos dois será executado depende de uma variável global emulador, que é declarada no arquivo rpsarmul.h. Caso seja 1, o código executado é o do emulador, caso seja 0, é o código da placa.

3.2 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 constantes. 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 queiramos adicionar alguma rotina de tratamento de interrupções, não podemos de nos esquecer de encadear a rotina do Angel caso identifiquemos que a fonte de interrupção não foi de algo causado por um código do usuário. Descreveremos mais à frente como isso é feito durante a instalação da rotina de tratamento de interrupções.

3.3 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 processos. Após esta etapa, a uma inicialização em C, feita no arquivo cinit.c, que ao fim de sua execução, roda o primeiro processo.

3.3.1 Modo

O primeiro passo do programa é informar qual o tipo de código que será executado no programa. Como descrito anteriormente, podemos ter o assembly ARM ou o assembly thumb. Em nosso programa, utilizaremos apenas código ARM, já que ele fornece mais instruções (favorece a legibilidade) e porque o tamanho do código não é um grande problema.

3.3.2 Pilhas

A fim de podemos utilizar as pilhas, precisamos inicializá-las em cada um dos modos que virão a ser utilizados. Em nosso programa, utilizamos os modos SVC, user e IRQ. Para inicializar as pilhas, devemos primeiro mudar o modo, desabilitar as interrupções temporariamente e só depois mudar o ponteiro de pilha. Podemos fazer os dois primeiros passos de uma vez só, já que estas operações são feitas através do registrador de estado, como descrito anteriormente. Porém, um problema nos surge ao tentarmos inicializar o ponteiro de pilha do modo user: uma vez que entremos neste modo, não podemos ir para nenhum outro estado privilegiado. A solução para este problema é que ao invés de mudarmos para o modo de usuário, vamos ao modo system, que nada mais é do que o modo usuário privilegiado.

Como podemos ver na figura ??, a pilha do modo user começa no endereço 0x20000, a do modo IRQ em 0x8000 e a do modo SVC em 0x7F80. Como convenção para todas as pilhas que utilizaremos, não podemos nos esquecer que elas crescem sempre para baixo.

3.3.3 Tabela de processos

Uma vez inicializadas as pilhas, podemos inicializar também a tabela de processos. Quando estamos inicializando o kernel, apenas um programa está a funcionar. Portanto, colocamos 0 em todas as entradas da tabela de processos, indicando que os processos estão inativos, menos para o processo 1, que nada mais é que o primeiro programa a ser rodado após a inicialização. Além disso, devemos setar como o processo atual (variável `handler_currenttaskid_str`, declarada em `handler.s`), o processo 1.

Após este ponto, o código pode ser escrito em C, já que inicializamos todas as estruturas de baixo nível. Portanto, passamos o controle para o arquivo `C_Entry`, que vai continuar com o processo de inicialização descrito a partir do item abaixo.

3.3.4 Periféricos

Para cada periférico da placa, como o display de sete segmentos, o DIP switch, os botões, o timer e os LEDs, há uma rotina de inicialização. Elas consistem apenas em habilitar flags e são executadas logo no início da etapa C do processo de inicialização da placa.

3.3.5 Instalação do tratamento de interrupção

Como á foi descrito anteriormente na seção ??, quando uma interrupção ocorre, a instrução no endereço 0x18 é executada caso ela seja uma interrupção de hardware ou no endereço 0x08 caso seja uma interrupção de software. Portanto, devemos colocar instruções adequadas nestas posições, o que é feito pela rotina de instalação do tratamento de interrupção.

A rotina de tratamento de interrupção funciona de modo similar para interrupções de hardware e se software. Em ambos os casos, recebemos o endereço da posição do vetor de interrupções onde ela deve ser instalada e o endereço da rotina que será executada quando uma interrupção ocorre. Com isso, adicionamos através de uma máscara a instrução `branch`.

Porém, há um caso especial quando executamos o código com a placa. Como já descrito anteriormente na seção ??, 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 anteriormente no vetor de interrupção, para o caso da interrupção ser do Angel, a rotina correta ser executada

(vide figura 3.3).

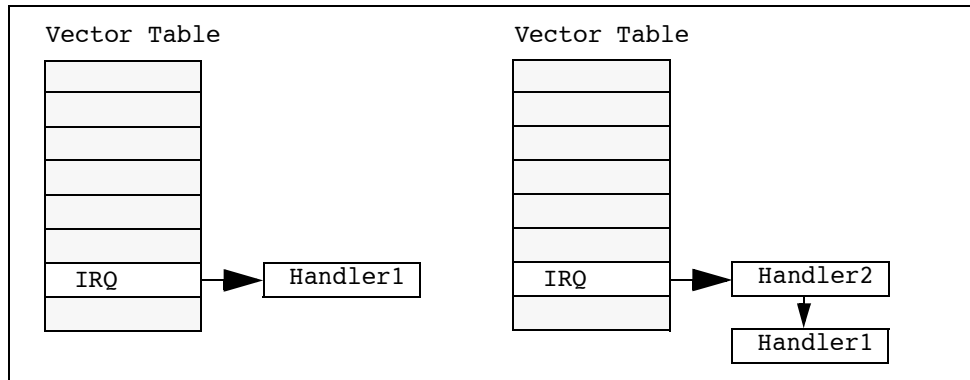


Figura 3.3: Encadeamento de interrupções. Fonte: (??)

3.3.6 Interrupção de timer

A fim de se utilizar as interrupções de timer para o chaveamento de processos, devemos além de inicializá-lo, definir algumas configurações como o tempo para a interrupção.

3.3.7 Habilitando interrupções

O último passo antes de se começar a executar o código do primeiro programa é habilitar simultaneamente o modo user e as interrupções de hardware. Como isso só pode ser feito por código assembly, temos de usar a instrução especial de C `__asm`. Finalmente, após este passo passamos o controle para o primeiro processo.

3.4 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 poderíamos 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-los de threads.

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

3.5 Chaveamento de processos

O chaveamento de processos é realizado inteiramente com o assembly escrito no arquivo handler.s. Nele, o estado do processo que está sendo executado é armazenado e o próximo processo, obtido através de round-robin, é colocado para execução.

3.5.1 Identificação da interrupção

Uma vez que há a interrupção de timer, a chamada de interrupção de hardware é realizada. Durante a instalação da rotina de tratamento de interrupção de hardware, apontamos para a rotina handler caso estejamos usando a placa ou a rotina handler_emulator caso estajamos usando o emulador. A diferença básica é que enquanto a primeira tenta identificar qual a fonte de interrupção que pode ser tanto o timer quando o Angel, a segunda já assume que a fonte é o timer, já que não há o Angel no emulador. Devemos salientar que deve-se armazenar toda informação contida nos registradores que é alterada durante o processo de tratamento de interrupção. Para tal, empilhamos os valores dos registradores usados a fim de se poder recuperar estes valores durante a etapa de salvamento do estado atual.

No caso do uso da placa, a fonte da interrupção se encontra no endereço 0x03ff4004, identificado como 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.

3.5.2 Recomeçar o timer

Quando é identificada a interrupção de timer, devemos reinicializar o contador do timer a fim de que possa interromper novamente no futuro. Para tal, executamos a rotina timer_irq, encontrada no arquivo irq.c.

3.5.3 Identificação do PCBs da troca de processos

A rotina de troca de processos tem como entrada duas variáveis: o fundo do PCB do processo atual e o fundo do PCB do próximo processo. Para obtermos tais dados, inicialmente precisamos identificar o número do processo atual, armazenado na variável handler_currenttaskid_str. A partir disso, com o auxílio da tabela de processos, verificamos um a um em round-robin qual o próximo processo ativo.

Tendo o número dos processos atual e próximo, podemos calcular o endereço do PCB de ambos. Para isso, realizamos a simples equação $PCB_{id} = (id - 1) * 68 + base$ para os dois processos e armazenamos os resultados nas variáveis `handler_currenttaskid_str` e `handler_nexttask_str`, que serão usadas na etapa da troca de processos.

Caso o próximo processo a ser executado venha a ser o mesmo que está atualmente sendo executado, a rotina de tratamento de interrupção ignora a etapa de troca de processos e retorna à execução da rotina que estava sendo executada anteriormente sem qualquer alteração, já que o valor dos registradores antes da interrupção estavam sendo armazenados na pilha.

3.5.4 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 IRQ e restabelece-se os registradores como estavam antes do início da troca de processos. Depois disso, muda-se o endereço do ponteiro de pilha para o PCB do processo atual. O grande truque vem no próximo passo: empilha-se todos os registradores. 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 3.1. 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 valores dos registradores, que são exatamente como estava empilhado este processo quando foi armazenado. Com isso, falta apenas restaurar o process counter para o ponto que o processo estava executando, o que vai ser descrito em seguida.

3.5.5 Um caso especial

Como descrito acima, caso se confirme que há apenas uma instrução ativa na tabela de processos, não é feita a troca de processos. Porém, antes de se retornar ao processo anterior, não podemos nos esquecer que devemos zerar a pilha do modo IRQ já não há mais dados relevantes nela e assim prevenimos que ela estoure seu tamanho. Outra ação importante é a restauração dos registradores que foram utilizados durante a rotina de tratamento.

3.5.6 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, devemos agora apenas fazer com que a instrução imediatamente posterior à aquela executada antes da interrupção seja executada. Porém, não podemos nos esquecer que o pipeline do processador fez com que o endereço da instrução duas vezes à frente tivesse sido armazenada. Para compensar isso, devemos subtrair o tamanho de uma instrução (quatro bytes) do endereço que vai ser colocado no process counter.

3.6 System calls

Uma system call nada mais é do que uma interrupção de software causada pelo kernel. 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 também está localizada no arquivo handler.s e é executada em modo SVC, que é privilegiado. As únicas instruções que chamam tais system calls são as rotinas fork, exec e exit.

3.6.1 Propriedades gerais das system calls

Escrever quando terminar de juntar as system calls. . .

3.6.2 fork

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.

3.6.3 exec

3.6.4 exit

3.7 Shell