EPOS for Raspberry Pi

Software/Hardware Integration Lab at UFSC

EPOS for Raspberry Pi

Table of contents

- EPOS for Raspberry Pi
- 1. Running EPOS on Raspberry Pi
  - 1.1. Compiling
  - 1.2. Running and Debugging
  - 1.3. Running Raspberry Pi3b in a real Hardware
    - 1.3.1. Setting up the SD Card
      - 1.3.1.1. Firmware Files
      - 1.3.1.2. Application Image
    - 1.3.2. Connecting the UART to your PC
- 2. ARMv8 and AArch64
- 3. System Calls in ARMv8 with AArch64
  - 3.1. Introdução
  - 3.2. Motivação
  - 3.3. Exception Levels no ARMv8
    - 3.3.1. Registradores Especiais do AArch64
  - 3.4. A Instrução SVC
  - 3.5. Exemplo de uma System Call no Linux (em ARMv8)
  - 3.6. Implementação de uma System Call no ARMv8: Exception Handling
  - 3.7. Demonstração Bare Metal
  - 3.8. System Call Reentrante
  - 3.9. Referências
- 4. ARMv7 and AArch32
- 5. Cortex-A53
- 6. Task Memory Model
  - 6.1. O modelo de memória de um processo
    - 6.1.1. Espaços de endereçamento
  - 6.2. System calls
    - 6.2.1. Execução
    - 6.2.2. Exemplo
  - 6.3. Switch context
    - 6.3.1. Principais causas para a troca de contexto entre processos
  - 6.4. Kernel space
  - 6.5. Referências
- 7. MMU for Paging
  - 7.1. Conceitos importantes
  - 7.2. MMU
  - 7.3. Paging
  - 7.4. MMU - ARMv7
  - 7.5. Paging - ARMv7
    - 7.5.1. Super Section
    - 7.5.2. Section
    - 7.5.3. Large Page
    - 7.5.4. Small Page
  - 7.6. Exemplo de tradução de endereço no ARMv7
  - 7.7. Ativação da MMU no ARMv7

    - 7.8. Referências
- 8. Task Context Switching
    - 8.1. What is Context Switching
    - 8.2. How does a Context Switching happen
    - 8.3. What is a PCB
    - 8.4. Perfomance considerations
        - 8.4.1. Translation Lookaside Buffer
        - 8.4.2. Address Space Identifier
            - 8.4.2.1. ASID on Context Switch
            - 8.4.2.2. Limitation of ASID
    - 8.5. Switching Context in ARMv7/Raspberry Pi3
        - 8.5.1. ARM Processor Mode
        - 8.5.2. IRQ
        - 8.5.3. Managing Address Spaces
            - 8.5.3.1. Acessing translation table support registers
    - 8.6. Validation code
    - 8.7. References
- 9. System Calls
    - 9.1. Motivação
    - 9.2. Definição de System Call
    - 9.3. Métodos de Implementação
    - 9.4. Funcionamento no ARMv7 e Cortex-A53
        - 9.4.1. System Calls na arquitetura ARMv7
        - 9.4.2. System Calls na família Cortex-A
    - 9.5. Como iniciar uma system call
        - 9.5.1. Registradores
        - 9.5.2. Instruções
            - 9.5.2.1. SVC - Supervisor Call
            - 9.5.2.2. HVC - Hypervisor Call
            - 9.5.2.3. SMC - Secure Monitor Call
            - 9.5.2.4. SRS - Store Return State
        - 9.5.3. Parâmetros
    - 9.6. Identificação do tipo de exceção
        - 9.6.1. System calls aninhadas
    - 9.7. Identificação da system call
    - 9.8. Como retornar de uma system call
        - 9.8.1. Instruções
            - 9.8.1.1. RFE - Return From Exception
            - 9.8.1.2. ERET - Exception Return
        - 9.8.2. Valor de retorno
    - 9.9. Referências
- 10. System Object Proxies and Agents
    - 10.1. Gerenciamento de Syscalls
        - 10.1.1. Stubs
        - 10.1.2. Agents
    - 10.2. Sugestão de Implementação
        - 10.2.1. Implementação da Mensagem
        - 10.2.2. Implementação de Stubs
        - 10.2.3. Implementação dos Agentes
    - 10.3. Referências
    - 10.4. Autores

- 11. Resource Management
  - 11.1. Resource Management in Multitasking
    - 11.1.1. Memory
    - 11.1.2. Processing Time
  - 11.2. Desalocação de recursos
  - 11.3. EPOS Multitasking
  - 11.4. Exemplo de Implementação
    - 11.4.1. Classe Task
    - 11.4.2. Ponteiro para Task nos Elementos
    - 11.4.3. Adição dos Elementos nas listas da Task
    - 11.4.4. Criação da Task
    - 11.4.5. Destructor da Task
  - 11.5. Resource Management Autores
  - 11.6. Resource Management Referências
- 12. Inter-Process Communication
  - 12.1. Motivação
  - 12.2. Modelos de IPC
    - 12.2.1. Memória Compartilhada
    - 12.2.2. Troca de Mensagens
      - 12.2.2.1. Comunicação Direta
      - 12.2.2.2. Comunicação Indireta
      - 12.2.2.3. Comunicação Síncrona ou Assíncrona
      - 12.2.2.4. Buffers
      - 12.2.2.5. Pipes
        - 12.2.2.5.1. Unnamed pipes
        - 12.2.2.5.2. Named pipes
      - 12.2.2.6. Sockets
  - 12.3. Exemplo de implementação
    - 12.3.1. Memória compartilhada
    - 12.3.2. Troca de mensagens
      - 12.3.2.1. Troca de mensagem assíncrono
      - 12.3.2.2. Troca de mensagem síncrona
  - 12.4. Referências
- 13. I/O
  - 13.1. Memory Mapped Peripherals
  - 13.2. Registradores de dispositivo
  - 13.3. Direct Memory Access(DMA)
  - 13.4. Memory Mapped Regions
  - 13.5. Níveis de Privilégio
  - 13.6. Controle de nível de privilégio
  - 13.7. Bits para controle de privilégio de uma região
    - 13.7.1. Modelo de permissões de acesso AP 2: 1
    - 13.7.2. Modelo de permissões de acesso AP 2: 0
  - 13.8. Single-channel DMA transfer
  - 13.9. Dual-channel DMA transfer
  - 13.10. Referências

1. Running EPOS on Raspberry Pi

1.1. Compiling

To compile an APP for Raspberry Pi3b, first configure the application *Traits<Build>* as follows:

```
template<> struct Traits<Build>: public Traits<void> { static const unsigned int MODE = LIBRARY; static const unsigned int ARCHITECTURE = ARMv7; // You can use ARMv8 or ARMv7 on QEMU. static const unsigned int MACHINE = Cortex; static const unsigned int MODEL = Raspberry_Pi3; static const unsigned int CPUS = 1; // or 4 static const unsigned int NODES = 1; // (> 1 => NETWORKING) static const unsigned int EXPECTED_SIMULATION_TIME = 60; // s (0 => not simulated, using real hardware) };
```

At the directory where you installed EPOS' source code, just type:

```
$ make APPLICATION=<appname>
```

1.2. Running and Debugging

To run and debug applications, follow the steps described in EPOS documentation.

1.3. Running Raspberry Pi3b in a real Hardware

First, to run an application in real Raspberry Pi3 hardware, use ARMv8 as the ARCHITECTURE in Traits<Build>. In EPOS, ARMv8 is very similar to ARMv7, it just replaces the cores() function in cpu.h, as Raspberry Pi3b hardware does not support the ARMv7 implementation of cores() function.

1.3.1. Setting up the SD Card

To boot a Raspberry Pi3b in a real hardware, you first need to configure an SD Card with the EPOS application image and some additional firmware files required by the Raspberry Pi3b hardware.

1.3.1.1. Firmware Files

The additional Firmware files required are available at the Raspberry Pi3b Official Github. From the firmware folder, you only need bootcode.bin and start.elf files. Setup the SD card with a single partition, fat32, and copy the bootcode.bin and start.elf files to the SD card.

1.3.1.2. Application Image

Raspberry Pi3b CPU boot is started by the GPU. The GPU reads the SD card and copies the kernel image to the specific initial address, where the first piece of code in this image is expected to be the Vector Table. The default image names are related to the compatibility to Pi models. You can specifically select your own name in config.txt. Considering no config.txt override, the search order for a Pi3 is:

```
if kernel8.img is found: boot in 64 bits mode else if any of kernel8-32.img, kernel7.img, or kernel.img are found: boot in 32 bits mode
```

The address of the Vector Table changes from 64 and 32 bits modes. For 32 bits, the vector table is initially located at the address 0x00008000, and 0x00080000 for 64 bits.

Currently, EPOS supports only 32 bits Raspberry Pi3b. Thus, after compiling, copy the final application binary file to the SD card, renaming it to kernel8-32.img or kernel7.img or kernel.img.

1.3.2. Connecting the UART to your PC

Warning: Attain to the Raspberry Pi3b energy supply requirements, and to the correct pin connection when connecting the RaspberryPi3b UART / FTDI / PC.

EPOS uses Raspberry MiniUART as the default Serial Display. To connect the Raspberrypi 3b UART to your PC, an FTDI is needed to intermediate the UART pins and connect the EPOS app output to the PC USB over serial protocol. The following configuration is then needed:

FTDI	Pi3b
TX	RX (pin 10)
RX	TX (pin 8)
GND	GNC (pin 6)

After connecting the FTDI to PC using a USB cable, the Raspberry Pi3b output can be seen by reading the USB content (e.g., using minicom or cutecom). The UART configuration is the following:

Baudrate	115200
Data bits	8
Stop bits	1
Parity	None
Flow Control	None

2. ARMv8 and AArch64

3. System Calls in ARMv8 with AArch64

3.1. Introdução

O conjunto de chamadas de sistema (syscalls) é a interface entre o sistema operacional e seus programas aplicativos, sendo esta utilizada para que a aplicação a nível de usuário requisite serviços privilegiados a nível do kernel do sistema operacional. Estes serviços podem ser relacionados à hardware, tal como acessar um disco rígido, ou relacionado ao sistema operacional, como a criação de novos processos.

Esta interface, separa o ambiente em que o usuário tem controle, daquele que ele não tem, que seria o kernel. Portanto uma system call irá executar de forma bastante restritiva e sem controle do programa de usuário, visto que irá operar sobre estruturas críticas do sistema operacional.

Se o programa de usuário tivesse acesso a funções do kernel sem esta interface, há a possibilidade deste programa corromper outras aplicações, acessar memória que não tem autorização e até mesmo corromper o sistema operacional como um todo.

De forma geral, alguns tipos de chamadas de sistema são:

- Gerenciamento de processo
  - fork

    - waitpid
- Gerenciamento de arquivos
    - open
    - close
    - read
    - write
- Gerenciamento do sistema de diretório e arquivo
    - mkdir
    - rmdir
    - mount
- Diversas
    - chmod
    - kill
    - time

Uma forma de implementar uma syscall é forçar uma exceção de hardware que por sua vez irá chamar o tratador de interrupções/exceções. Algumas arquiteturas possuem instruções específicas para system calls, o que é o caso do ARMv8 que possui uma instrução chamada Supervisor Call (SVC) que irá executar o serviço do sistema operacional que foi requisitado.

3.2. Motivação

Um uso comum de chamadas de sistema para SOs como o Linux é criar uma separação entre espaço de usuário e de sistema. Assim, os serviços do sistema são acessados apenas via system calls o que implica em:

- maior segurança do sistema operacional sob as aplicações, visto que essas não terão acesso direto às estruturas e funções internas do sistema; e
- nível maior de compatibilidade binária entre arquivos compilados por diferentes versões do Linux, pois uma chamada de sistema permanece a mesma: uma única execução da instrução de system call em vez de uma chamada de função direto para o endereço na biblioteca do kernel.

3.3. Exception Levels no ARMv8

No ARMv8, a execução acontece em um de 4 Exception Levels. No AArch64 cada exception level está associado com um nível de privilégio, de forma similar aos PL (Privilege Levels) do ARMv7.

- **EL0** possui o menor nível de privilégio e será onde as aplicação de usuário executarão;
- **EL1** agrega, tipicamente, o Kernel do sistema operacional, e portanto neste nível que serão executadas as system calls do SO. Visto que o espaço de usuário é em EL0 e as system calls estão em EL1, a aplicação precisa executar a instrução SVC (Supervisor Call) para provocar uma exceção síncrona e elevar o seu Exception Level para EL1.
- **EL2** é o nível de exceção onde o Hypervisor executará. Para elevarmos para este nível, a partir do EL1, devemos utilizar a instrução HVC
- **EL3** o nível de exceção de maior privilégio, e será onde teremos o firmware de baixo nível executando, que tipicamente seria o Secure Monitor. Para subir à este nível utiliza-se a instrução SMC.

Em cada Exception Level, o processador tem um modo de execução diferente, como pode ser visto na tabela abaixo. No caso quando executamos a instrução SVC, além de subirmos para o EL1, estamos mudando o modo do processador para SVC.

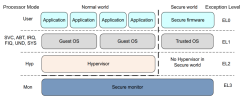


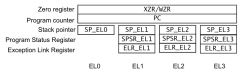
Figura: Exception Levels e Processor Modes do ARMv8

3.3.1. Registradores Especiais do AArch64

Além do mais, em cada EL, teremos um conjunto de registradores especiais. Dois deles servirão para o salvamento de contexto ao tratar uma exceção:

- **SPSR_ELn** (Saved Program Status Register): salva o estado do processo PSTATE quando uma exceção é tomada. Teremos um SPSR para cada Exception Level (com exceção do 0), portanto o n será substituido por 1, 2 ou 3
- **ELR_ELn** (Exception Link Register): armazena o endereço de retorno do nível em que ocorreu a exceção. Novamente teremos um desse registrador para EL com exceção do 0. Portanto o ELR_EL1 irá armazenar o endereço de retorno de uma exceção que ocorreu no EL0 e assim por diante

Teremos também um SP (Stack Pointer) para cada Exception Level. O PC (Program Counter) e o XZR/WZR (registrador zero) é o mesmo em todos os Exception Levels.



3.4. A Instrução SVC

Para uma aplicação de usuário chamar uma syscall ela deve definir os 32 bits inferiores do registrador x8 com o número da syscall que deseja-se chamar, deve também, se necessário, passar os argumentos nos registradores x0 à x4, e por fim emitir a instrução svc 0. O valor de retorno da system call estará disponível no registrador x0 após a instrução SVC.

Ao executar esta instrução, a execução da aplicação será interrompida pela execução da system call por parte do Kernel

A sintaxe da instrução svc é a seguinte:

```
svc {cond} #imm
```

- **imm:** expressão que é avaliada para um número inteiro. Ela é ignorada pelo processador, mas pode ser usada pelo tratador de interrupções para saber qual é o serviço requisitado.
- **cond:** condição opcional que as instruções da arquitetura ARMv8 possuem que verifica flags setadas por instruções anteriores para decidir se o svc será ou não executado.

3.5. Exemplo de uma System Call no Linux (em ARMv8)

O programa abaixo faz uma simples operação de escrever “Hello World!” no stdout e faz a system

call exit:

```
.global _start .section .text _start: // write system call mov x8, #64 // Passing the syscall number in decimal mov x0, #1 // File descriptor. 1 is stdout ldr x1, =message // Loading to x1 mov x2, #12 // Lenght of we are writing svc 0 // Invoke syscall // exit system call mov x8 , #93 mov x0 , #41 // Passing a dummy error code svc 0 .section .data message: .ascii "Hello World\n"
```

A função _start é o ponto de início do programa. A chamada de sistema relacionada a escrita em arquivo é a write e o número dela de acordo com a tabela de system calls do ARMv8.

x8	syscall name	references	x0	x1	x2	x3	x4	x5
64	write	man 2 write	fd	const char *buf	size_t count			

A tabela com a especificação dos códigos de syscall do Linux para ARM64 (que implementa ARMv8) pode ser encontrada no seguinte link:

https://chromium.googlesource.com/chromiumos/docs/+/master/constants/syscalls.md#arm64-64_bit

A primeira instrução é responsável por escrever o número da system call no registrador X8.

Revisando a tabela de syscalls, o primeiro argumento é o número correspondente ao descritor de arquivo no qual a escrita será feita, que no caso do valor 1 trata-se do descritor de arquivo referente ao stdout, então o registrador X0, que guarda o primeiro argumento, é carregado com o valor 1.

O segundo argumento é o início do buffer que guarda o conteúdo a ser escrito, então no registrador X1 é guardado o endereço do dado estático que representa a string “Hello World!\n”. Esse endereço é referenciado pela label message.

O terceiro e último argumento é quantos bytes desse buffer serão copiados para a saída definida no primeiro argumento. A string “Hello World!” codificada em ASCII ocupa exatos 12 bytes, com a quebra de linha representada pelo \n o tamanho total fica 13 bytes.

Por último, a instrução svc é chamada com o valor 0. Esse valor é escolhido pois o imediato precisa de algum valor que não será necessariamente usado, e o valor da system call já está em X8.
Após a system call de write, é feita a system call do exit, que possui número 93 e recebe um argumento no registrador x0, que é o código de erro que será retornado. Após a compilação e execução deste código, em SOs Linux pode-se verificar o valor de retorno de uma aplicação com ‘echo $?’, que no caso desta aplicação irá retornar 41.

A tabela com a especificação dos códigos de syscall para ARM64 (que implementa ARMv8) pode ser encontrada no seguinte link:

https://chromium.googlesource.com/chromiumos/docs/+/master/constants/syscalls.md#arm64-64_bit

3.6. Implementação de uma System Call no ARMv8: Exception Handling

Quando uma exceção ocorre, o processador necessita trocar para o Exception Level que dê suporte para o tratamento da exceção, ou seja, nenhuma exceção será tratada em EL0, que é o nível da aplicação.

Portanto uma exceção causará uma alteração no fluxo do programa, seguido de uma alteração de

Exception Level e Processor Mode, tornando necessário o salvamento do contexto para que possamos voltar depois a executar a aplicação.

Visto que a instrução SVC provoca uma elevação de nível de privilégio para o EL1, será feito o uso dos registradores SPSR_EL1 e ELR_EL1 para salvamento do contexto do programa aplicativo.

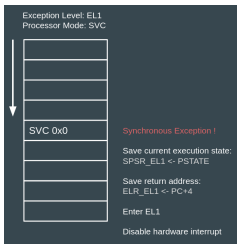


Figura: Execução de um programa aplicativo interrompida por uma exceção síncrona

Feito isto, o PC será setado para o vetor da Vector Table que possui o Tratador de Exceção correspondente a exceção que estamos tratando. Para descobrir o endereço deste vetor, tomaremos como base o endereço base da Vector Table, disponível no registrador VBAR_EL1, e somaremos a este endereço o offset do tratador da exceção corrente.

Portanto a Vector Table, de modo geral, é uma tabela de Tratadores de Exceção, sendo que esta tabela contém instruções a serem executadas, ao invés de um conjunto de endereços. Cada vetor da tabela possui tamanho de 32 instruções, e como cada instrução no ARMv8 possui 4 bytes, teremos que os vetores serão espaçados entre si por 128 bytes (ou 0x80 bytes)

Note também que teremos uma Vector Table para cada Exception Level a partir do 1, portanto teremos também VBAR_EL2 e VBAR_EL3.

Address	Exception type	Description
VBAR_ELn + 0x000	Synchronous	Current EL with SP0
+ 0x080	IRQ/vIRQ	
+ 0x100	FIQ/vFIQ	
+ 0x180	SError/vSError	
+ 0x200	Synchronous	Current EL with SPx
+ 0x280	IRQ/vIRQ	
+ 0x300	FIQ/vFIQ	
+ 0x380	SError/vSError	
+ 0x400	Synchronous	Lower EL using AArch64
+ 0x480	IRQ/vIRQ	
+ 0x500	FIQ/vFIQ	
+ 0x580	SError/vSError	

Address	Exception type	Description
+ 0x600	Synchronous	Lower EL using AArch32
+ 0x680	IRQ/vIRQ	
+ 0x700	FIQ/vFIQ	
+ 0x780	SError/vSError	

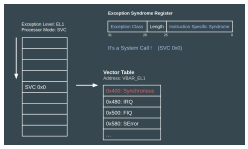
Figura: Offsets da Vector Table relativo ao endereço da Vector Table

A exceção provocada por SVC é uma exceção síncrona, uma vez que esta foi provocada pela execução de uma instrução, e portanto será tratada pelo Exception Handler em 0x400, pois viemos de um EL abaixo (do EL0 ao EL1) e estamos usando o modo AArch64.

Uma exceção síncrona pode ter diversos motivos, e para que seja possível distinguir cada motivo, no tratador de exceção será consultado o Exception Syndrome Register, que irá conter informações necessárias para determinar a razão da exceção:

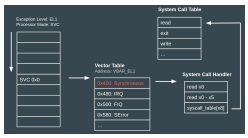
- Bits 31:26: classe da exceção, que pode ser unallocated instruction, data abort, SVC, HVC, entre outros.
- Bit 25: indicará o tamanho da instrução que provocou a exceção: 0 se for de 16 bits e 1 se for de 32 bits
- Bits 24:0: contém informação específica para cada classe de exceção. No caso de uma exceção

SVC, este campo contém o valor imediato da instrução SVC, que de acordo com os nossos exemplos será 0.



Após identificar que trata-se de uma exceção do tipo SVC 0 (system call), é chamado o system call handler, que irá indexar a tabela de system calls com o número da system call que foi requisitada e ler e encaminhar os argumentos que foram passados pela aplicação para a system call correspondente.

A system call por sua vez irá executar um conjunto de instruções que exigem um nível de privilégio superior ao de uma aplicação e que podem vir a envolver o kernel do sistema operacional, de forma a satisfazer o serviço que foi requisitado pela aplicação. Por fim, a system call retorna para o system call handler, este por sua vez retorna para o exception handler, e este irá retornar para o fluxo da aplicação por meio da instrução ERET (Exception Return).



A instrução ERET finaliza o tratamento da exceção e retorna ao Exception Level anterior à exceção, que no caso de uma system call será o EL0. Para que seja possível retornar ao contexto da aplicação, esta instrução irá fazer com que o SPSR_EL1 seja copiado para o PSTATE, e que o ELR_EL1 seja copiado para o PC.

Voltando ao contexto da aplicação, volta-se a seguir o fluxo da aplicação.

3.7. Demonstração Bare Metal

Arquivos de destaque:

- **exc.c:** Exception Handler, Syscall Handler e Syscall Table
- **main.c:** Inicializa o UART, MMU e provoca uma exceção síncrona
- **start.S:** Código assembly para setup e Vector Table

O código fonte encontra-se disponível em: https://github.com/JPADN/syscall-bare-metal

Leia o README.md para instruções de compilação e execução.

Tudo começa em main.c, onde temos um bloco asm que irá demonstrar uma aplicação chamando uma system call. Primeiramente, o programa de usuário irá salvar na pilha os registradores que vão ser utilizados pela system call por meio da instrução str. Logo após são passados os argumentos da system call nos registradores x0 à x5, e o número da system call que deseja-se executar no registrador x8. Por fim executa-se a instrução SVC.

```
void main()
{
    unsigned long ret;

    // set up serial console
    uart_init();

    // set up paging
    mmu_init();

    // Perform a system call here
    __asm__ volatile (
    "ldr x0, [sp, #-8]\n\t"
    "ldr x1, [sp, #-8]\n\t"
    "ldr x2, [sp, #-8]\n\t"
    "ldr x3, [sp, #-8]\n\t"
    "ldr x4, [sp, #-8]\n\t"
    "ldr x5, [sp, #-8]\n\t"
    "ldr x6, [sp, #-8]\n\t"
    "ldr x7, [sp, #-8]\n\t"

    "mov x0, #0x2\n\t"
    "mov x1, #0x2\n\t"
    "mov x2, #0x3\n\t"
    "mov x3, #0x4\n\t"
    "mov x4, #0x5\n\t"
    "mov x5, #0x6\n\t"
    "mov x6, #0x7\n\t"

    "svc #0x67"

    "mov %0, x0\n\t"

    "ldr x0, [sp], #8\n\t"
    "ldr x1, [sp], #8\n\t"
    "ldr x2, [sp], #8\n\t"
    "ldr x3, [sp], #8\n\t"
    "ldr x4, [sp], #8\n\t"
    "ldr x5, [sp], #8\n\t"
    "ldr x6, [sp], #8\n\t"
    "ldr x7, [sp], #8\n\t"
    : "=r" (ret)
    );
```

Como foi visto anteriormente, a instrução SVC irá provocar uma exceção síncrona, e portanto o fluxo de execução mudará para o vetor da vector table que irá tratar essa exceção. A vector table está definida em start.S.

start.S trata-se de um arquivo em linguagem assembly que tem a função de fazer o setup inicial do Raspberry Pi que estamos emulando, que envolve instruções para forçar o uso de apenas uma CPU, verificar em qual EL a máquina foi bootada, e principalmente (para o contexto deste seminário), configurar o registrador VBAR_EL1 para apontar para a nossa Vector Table.

```
// set up exception handlers
ldr     x2, =_vectors
msr     vbar_el1, x2
```

```
_vectors:
    // synchronous
    .align  7
    mov     x6, #0  // Passing exception type
    bl      exc_handler
    eret

    // IRQ
    .align  7
    mov     x6, #1
    bl      exc_handler
    eret

    // FIQ
    .align  7
    mov     x6, #2
    bl      exc_handler
    eret

    // SError
    .align  7
    mov     x6, #3
    bl      exc_handler
    eret
```

Cada vetor da Vector Table irá setar o registrador x6 para um número que identifique o tipo de exceção, e fazer um branch and link para o Exception Handler (função em C), que será o mesmo para todas as exceções. Ao término do tratamento da exceção, executamos eret para voltar ao contexto anterior a exceção.

Visto que vamos tratar uma exceção síncrona, iremos setar x6 para 0 e fazer um branch para o exception handler.

O exception handler está definido em exc.c. Ao entrarmos nele, iremos ler os registradores da system call (de x0 à x5 estarão os argumentos e em x8 o número da system call), o registrador x6 para poder identificar o tipo de exceção, e também iremos ler os registradores de sistema do ARM, que são: ESR_EL1, ELR_EL1, SPSR_EL1, FAR_EL1 e CurrentEl.

```
void exc_handler()
{
    // Since we are working with 64 bit registers, we use long & variables (sizeof(long) is 8 bytes)
    unsigned long x0, x1, x2, x3, x4, x5, x6, x7, x8, ret;
    unsigned long esr, elr, spsr, far, current_el;  // System Registers for EL1

    int e = 0;

    // Reading syscall registers
    __asm__ volatile (
    "mov    %0, x0\n\t"  // Reading System Call arguments (x0 to x5)
    "mov    %1, x1\n\t"
    "mov    %2, x2\n\t"
    "mov    %3, x3\n\t"
    "mov    %4, x4\n\t"
    "mov    %5, x5\n\t"
    "mov    %6, x8\n\t"  // Reading System Call number
    : "=r" (x0), "=r" (x1), "=r" (x2), "=r" (x3), "=r" (x4), "=r" (x5), "=r" (x8)
    );

    // Reading ELX System Registers and CurrentEl
    __asm__ volatile (
    "mov    %0, x6\n\t"
    "mrs    %1, esr_el1\n\t"  // Reading Exception Syndrome Register
    "mrs    %2, elr_el1\n\t"  // Reading Exception Link Register
    "mrs    %3, spsr_el1\n\t"  // Reading Saved Program Status Registers
    "mrs    %4, far_el1\n\t"  // Reading Fault Address Register
    "mrs    %5, CurrentEL\n\t"  // Reading Current Exception Level
    "and    %5, %5, #0x4\n\t"  // Applying mask
    "lsr    %5, %5, #2\n\t"
    : "=r" (type), "=r" (esr), "=r" (elr), "=r" (spsr), "=r" (far), "=r" (current_el)
    );
```

Feito isto, iremos printar no terminal qual o tipo de exceção que estamos tratando, a partir da leitura do registrador x6.

Logo após, iremos utilizar o ESR_EL1 (Exception Syndrome Register) com um bit shift para a direita de 26 casas, a fim de lermos o campo Exception Class (Bits 31:26) do registrador ESR_EL1 e distinguir qual a causa da exceção, que no nosso caso será uma system call.

```
// print out information type
switch(type) {
    case 0: uart_puts("\nSynchronous Exception"); break;
    case 1: uart_puts("\nIRQ"); break;
    case 2: uart_puts("\nFIQ"); break;
    case 3: uart_puts("\nSError"); break;
}
uart_puts(": ");
// decode exception type (some, not all)
switch(esr>>26U) {
    case 0b000000: uart_puts("Unknown"); break;
    case 0b000001: uart_puts("Trapped WFI/WFE"); break;
    case 0b001110: uart_puts("Illegal execution"); break;
    case 0b010101: uart_puts("SVC instruction execution"); break;
    case 0b100000: uart_puts("Instruction abort, lower EL"); break;
    case 0b100001: uart_puts("Instruction abort, same EL"); break;
    case 0b100010: uart_puts("Instruction alignment fault"); break;
    case 0b100100: uart_puts("Data abort, lower EL"); break;
    case 0b100101: uart_puts("Data abort, same EL"); break;
    case 0b100110: uart_puts("Stack alignment fault"); break;
    case 0b101100: uart_puts("Floating point"); break;
    default: uart_puts("Unknown"); break;
}
```

Será chamado então o System Call Handler (syscall_handler()), que irá receber os argumentos e o número da system call, irá indexar a System Call Table (syscall_table[]) com o número da system call requisitada. Serão repassados os argumentos para a system call requisitada e esta por sua vez irá iniciar sua execução.

```
unsigned long syscall_handler(unsigned long syscall_num, unsigned long x0, unsigned long x1, unsigned long x2,
                              unsigned long x3, unsigned long x4) {
uart_puts("\n\nSyscall Handler\n");
if (syscall_num < SYSCALL_TABLE_LEN) {
  uart_puts("Number is less syscall table\n");
  uint64_t (*syscall_func)();
  uint64_t ret;
  syscall_func = syscall_table[syscall_num];
  ret = syscall_func();
  return ret;
}
return syscall_notsupported(syscall_num, x0, x1, x2, x3, x4);
}
```

```
#define SYSCALL_TABLE_LEN 4

const unsigned long (*syscall_table[SYSCALL_TABLE_LEN])() = {
    syscall_a,
    syscall_b,
    syscall_c,
    syscall_d,
};
```

Nesta demonstração, temos quatro system calls ilustrativas, isto é, elas não desempenham uma operação com o kernel do sistema operacional, pois nem temos um sistema operacional neste contexto. Portanto são funções simplórias que apenas fazem um print no terminal. A syscall_b é diferente das outras, e irá printar no terminal também os argumentos que foram passados a ela e retornar a soma destes argumentos.

```
unsigned long syscall_b(unsigned long x0, unsigned long x1, unsigned long x2, unsigned long x3,
                        unsigned long x4, unsigned long x5) {
uart_puts("\n\n I'm Syscall B\n\nThose were the arguments:\n");
uart_hex(x0);
uart_puts(" ");
uart_hex(x1);
uart_puts(" ");
uart_hex(x2);
uart_puts(" ");
uart_hex(x3);
uart_puts(" ");
uart_hex(x4);
uart_puts(" ");
uart_hex(x5);
uart_puts(" ");
uart_hex(x0);
uart_puts(" ");
return x0 + x1 + x2 + x3 + x4 + x5;
}
```

Este valor será retornado para o exception handler que irá mover este valor para registrador x0, de modo a seguir a convenção de system calls do ARMv8.

```
        case 0b010101: ret = syscall_handler(x0, x1, x2, x3, x4, x5); break;
        case 0b100000: uart_puts("Instruction abort, lower EL"); break;
        case 0b100001: uart_puts("Instruction abort, same EL"); break;
        case 0b100010: uart_puts("Instruction alignment fault"); break;
        case 0b100100: uart_puts("Data abort, lower EL"); break;
        case 0b100101: uart_puts("Data abort, same EL"); break;
        case 0b100110: uart_puts("Stack alignment fault"); break;
        case 0b101100: uart_puts("Floating point"); break;
        default: uart_puts("Unknown"); break;
    }

    // dump registers
    // ... omitted code ...

    asm volatile (
    "msr     x0, esr_el1\n\t"
    :: "r"(esr)
    );
    return;
}
```

Ao retornar do exception handler, será executada a instrução eret como visto anteriormente, e portanto voltaremos ao contexto da aplicação. Retornando para a aplicação, será restaurado o contexto anterior a system call que estava armazenado na pilha e printado no terminal o valor de retorno da system call.

3.8. System Call Reentrante

A diferença entre um tratador reentrante e um não reentrante é que durante a execução de um reentrante as interrupções que não são do mesmo tipo que está sendo tratado no momento podem ocorrer normalmente sem afetar a corretude da execução.

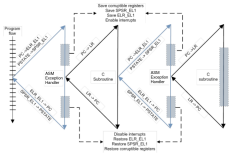
Um tratador de interrupções começa com as interrupções desabilitadas, e portanto se o seu tempo de execução for longo demais a ponto de que deveria ocorrer outra exceção durante o seu tratamento, ela será perdida. Este é um cenário que poderia vir a acontecer no caso de system calls, visto que pode haver uma latência na comunicação com o kernel.

Um tratador reentrante, nesse caso, salva na pilha informações essenciais para o tratamento da interrupção de forma que ele possa habilitar logo em seguida as interrupções novamente.
Abaixo temos um exemplo de um tratador de exceções que não é reentrante, esse exemplo simplesmente carrega os registradores do coprocessador com informações sobre a exceção e quatro

argumentos da exceção (x0, x1, x2, x3).

```
Exception_Handler: // Storing in the stack STP X0, X1, [SP, #-16]! STP X2, X3, [SP, #-16]! // Reading System Registers MRS X0, ESR_EL1 MRS X1, ELR_EL1 MRS X2, SPSR_EL1 MRS X3, FAR_EL1 … LDP X2, X3, [SP], #16 LDP X0, X1, [SP], #16 ERET
```

Para modificar esse tratador de forma que ele se torne reentrante seria necessário permitir que interrupções aconteçam novamente, e então chamar a subrotina do tratador, que no nosso exemplo é um tratador de exceções síncronas, ou seja, exceções provocadas pela instrução SVC. Após o término do tratamento da exceção, as interrupções são desabilitadas, e os valores de SPSR_EL1, ELR_EL1, e de registradores “sujáveis” são restaurados da pilha.




```
Exception_Handler: // Storing in the stack STP X2, X3, [SP, #-16]! STP X0, X1, [SP, #-16]! // Reading System Registers MRS X0, ESR_EL1 MRS X1, ELR_EL1 MRS X2, SPSR_EL1 MRS X3, FAR_EL1 STP X0, X1, [SP, #-16]! STP X2, X3, [SP, #-16]! BL identify_and_clear_source // Read interrupt source, clearing interrupt in controller MSR DAIFClr, #0b0010 BL C_Sync_Handler MSR DAIFSet, #0b0010 LDP X2, X3, [SP], #16 LDP X0, X1, [SP], #16 MSR ESR_EL1, X0 MSR ELR_EL1, X1 MSR SPSR_EL1, X2 MSR FAR_EL1, X3 LDP X2, X3, [SP], #16 LDP X0, X1, [SP], #16 … ERET
```

A diferença para o não reentrante é a seguinte:

1. Após mover os valores de ESR, SPSR, ELR e FAR dos registradores do coprocessador para os registradores ARM, eles são postos na pilha.
2. Em seguida ocorre um BL que irá para uma rotina que deve sinalizar que a interrupção em questão está sendo tratada.
3. E então a instrução MSR DAIFClr, #0b0010 é executada para permitir que interrupções ocorram novamente, limpando o bit da máscara de interrupções.
4. É feito um BL para uma rotina em C que tratará da interrupção.
5. Após o tratamento as interrupções são desabilitadas setando o bit de máscara.
6. Então, o valor prévio dos coprocessadores também é restaurado.
7. E por último, o valor inicial dos registradores é restaurado da pilha.

3.9. Referências

Sistemas Operacionais projeto e implementação, Andrew S. Tanembaum e Albert S. Woodhull. 4ª edição

https://eastrivervillage.com/Anatomy-of-Linux-system-call-in-ARM64/

ARM Cortex-A Series Programmer’s Guide for ARMv8-A

Arm Architecture Reference Manual Armv8, for Armv8-A architecture profile

https://github.com/bztsrc/raspi3-tutorial

https://developer.arm.com/documentation/100933/0100/Interrupt-handling

4. ARMv7 and AArch32

5. Cortex-A53

6. Task Memory Model

6.1. O modelo de memória de um processo

Um processo é uma instância de um programa em execução. A memória alocada para cada processo é composta por diversas partes, geralmente denominadas **segmentos**. Os principais segmentos de um processo são:

- **Segmento de texto**: o segmento de texto contém instruções de máquina do programa o qual o processo está executando. O segmento de texto é marcado com a permissão de somente leitura para que o processo não modifique acidentalmente suas próprias instruções com um valor de ponteiro incorreto. Uma vez que múltiplos processos podem estar executando o mesmo programa, o segmento de texto é marcado como compartilhado, de modo que uma única cópia do código do programa pode ser mapeada para o espaço de endereçamento de todos os processos.

- **Segmento de dados inicializados**: o segmento de dados inicializados contém variáveis globais e estáticas que foram explicitamente inicializadas. Os valores destas variáveis são lidas do arquivo executável quando o programa é carregado para a memória.

- **Segmento de dados não-inicializados**: o segmento de dados não inicializados contém variáveis globais e estáticas que não são explicitamente inicializadas. Antes de iniciar o programa, o sistema inicializa toda a memória neste segmento com o valor 0. Por razões históricas, este segmento é frequentemente chamado por segmento *bss*, um nome derivado de um antigo mnemônico *assembler* para "*block started by symbol*".

- **Pilha**: a pilha é um segmento que cresce e encolhe dinamicamente ao se adicionar a remover *stack frames*. Um *stack frame* é alocado para cada chamada de função. Um *frame* armazena as variáveis locais da função (também chamado de variáveis automáticas), parâmetros e valor de retorno.

- **Heap**: A *heap* é uma área em que a memória (para variáveis) pode ser dinamicamente alocada em tempo de execução.

6.1.1. Espaços de endereçamento

Espaços de endereçamento são representações de memória dada para cada processo situado no espaço de usuário no sistema. Cada processo é capaz de enxergar somente o seu espaço de endereçamento, como se toda a memória do sistema pertencesse a ele. Além disso, os espaços de endereçamento podem ser muito maiores do que a memória física disponível. O processo, por meio do *kernel*, pode dinamicamente adicionar e remover áreas de memória de seu espaço de endereçamento.

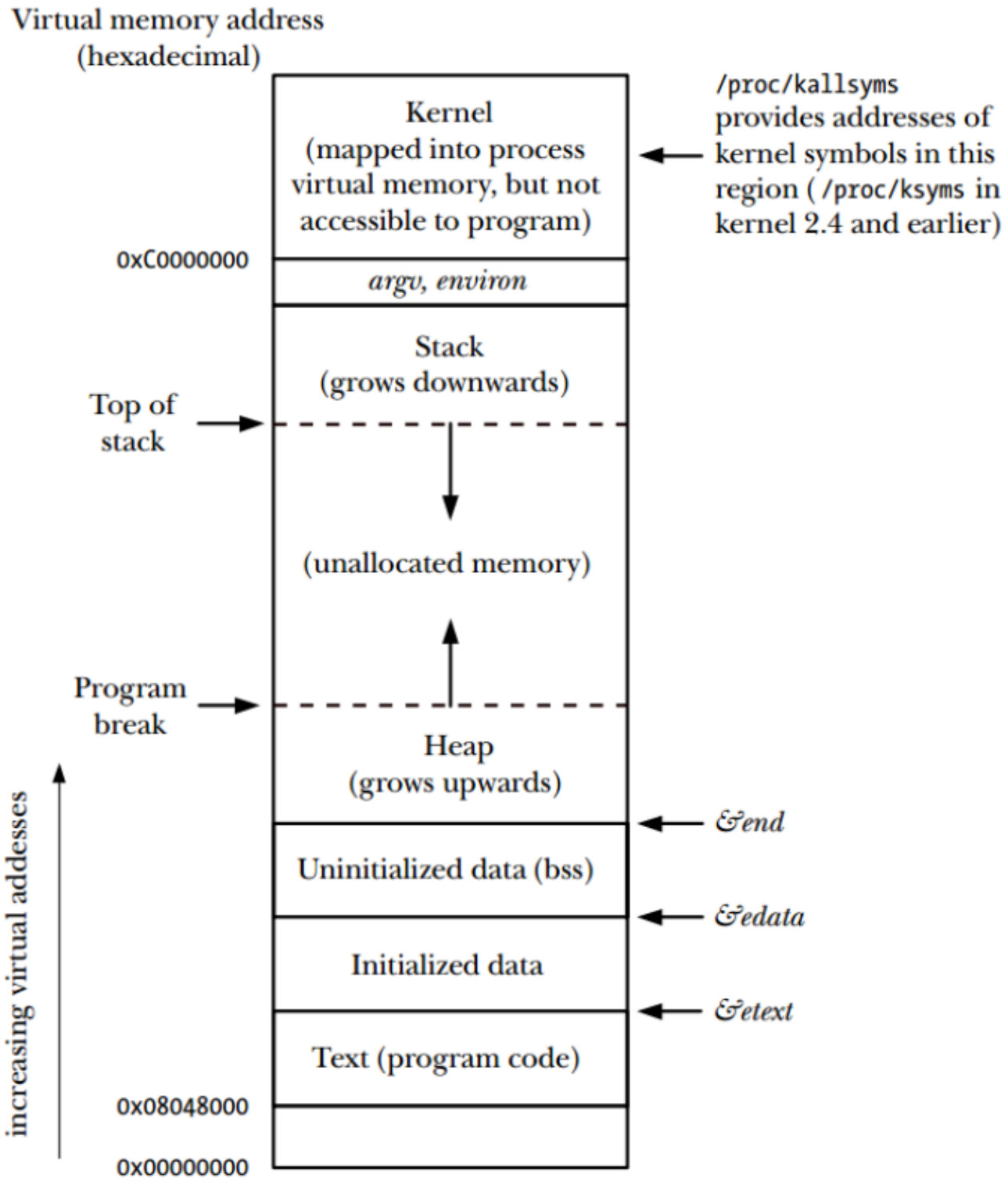
Cada processo possui um espaço de endereçamento associado, que nada mais é do que uma lista com endereços de memória, indo de 0 até algum valor máximo, no qual o processo pode ler e escrever, respeitando-se as permissões de cada área.

Endereços de memória localizados no espaço de endereçamento de um processo são totalmente

isolados dos mesmos endereços de memória do espaço de endereçamento de um outro processo, salvo casos onde um processo compartilha partes de seu espaço de endereçamento com outro processo. Espaços de endereçamento permitem que a memória dada a um processo esteja isolada dos demais. Diversos processos podem ter dados diferentes no mesmo endereço em seus respectivos espaços de endereçamento, pois estes dados estão disponíveis apenas em seu contexto, e não são visíveis externamente.

As áreas de memória dentro de um espaço de endereçamento possuem algumas permissões associadas: leitura, escrita e execução. Para que o processo possa acessar determinada área, ele deve possuir as permissões necessárias. Um acesso a um endereço de memória em uma área inválida faz com que o processo seja morto pelo kernel, seguido de uma mensagem de "*segmentation fault*".

A figura abaixo ilustra o modelo de memória de um processo, sendo representado por seu espaço de endereçamento. Como pode ser observado, o espaço de endereçamento realiza o mapeamento do espaço de memória do *kernel* no topo de seu endereço. Em geral, os sistemas operacionais possuem o *kernel* mapeado no espaço de endereçamento de cada um dos processos. No entanto, este mapeamento não resulta em desperdício de memória, visto que já apenas uma instância do *kernel* alocada na memória física.



6.2. *System calls*

Quando um programa aplicação deseja utilizar algum recurso do sistema, este deve ser feito por meio das abstrações oferecidas pelos sistemas operacionais. Este processo é realizado utilizando uma *system call*, que nada mais é do que uma função de um tipo especial, no qual o processo entra no modo *kernel*, ou seja, executa funções ligadas ao *kernel* com permissões privilegiadas. De modo geral, fazer uma *system call* é como fazer um tipo especial de chamada de função, só que *system calls* entram no *kernel*, e chamadas comuns, não.

System calls proveem uma camada entre o hardware e os processos a nível de usuário. Esta camada

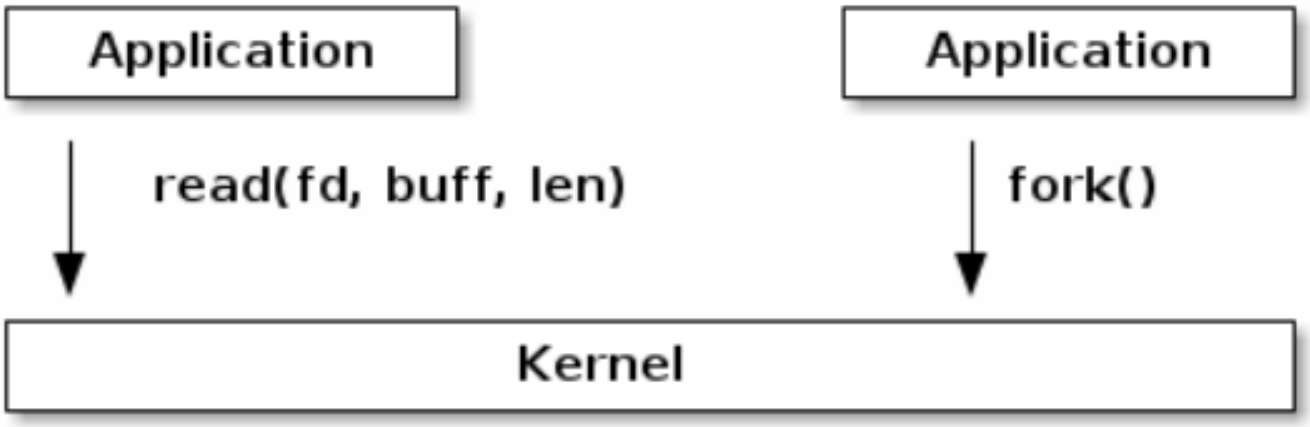
serve para três propósitos principais:

- Primeiro, ela provê uma interface abstrata de hardware para o espaço de usuário. Quando há leitura ou escrita de um arquivo, por exemplo, aplicações não se preocupam com o tipo de disco, mídia ou até mesmo o tipo do sistema de arquivos sob o qual o arquivo reside.
- Segundo, *system calls* garantem a segurança do sistema e sua estabilidade. Com o *kernel* atuando como um intermediário entre os recursos do sistema e o espaço de usuário, o *kernel* pode arbitrar o acesso baseado nas permissões, usuários e outros critérios.
- Por último, uma camada única entre o espaço de usuário e o resto do sistema permite a utilização de uma abstração do sistema real por parte dos processos. Se aplicações fossem livres para acessar os recursos do sistema sem o conhecimento do *kernel*, seria quase impossível implementar mecanismos de *multitasking* e memória virtual, e certamente impossível de fazê-lo com estabilidade e segurança.

Alguns pontos sobre *system calls*

- Uma *system call* altera o estado do processo de modo usuário para o modo kernel, para que a CPU, executando o processo de usuário, possa acessar a memória protegida do *kernel*.
- O conjunto de *system calls* é fixo. Cada *system call* é identificada por um número único.
- Cada *system call* possui um conjunto de parâmetros que especificam as informações que serão transferidas do modo usuário, ou seja, do espaço de endereçamento do processo, para o espaço do *kernel*, e vice-versa.

A figura abaixo exibe uma representação envolvendo aplicações e *system calls*. Ao executar um *read()*, ou um *fork()*, a aplicação está, na verdade, realizando *system calls* ao *kernel*.



6.2.1. Execução

Para realizar uma *system call* em uma arquitetura x86-32, por exemplo, são executados os seguintes passos:

1. O programa aplicação faz uma *system call* invocando uma função C que serve como um invólucro (uma função da *libc*, por exemplo), que por sua vez chama uma função do *kernel*.
2. Esta função C deve tornar todos os parâmetros da *system call* disponíveis para a rotina que realiza o tratamento da exceção (*trap*) da *system call*. Estes parâmetros são passados para a

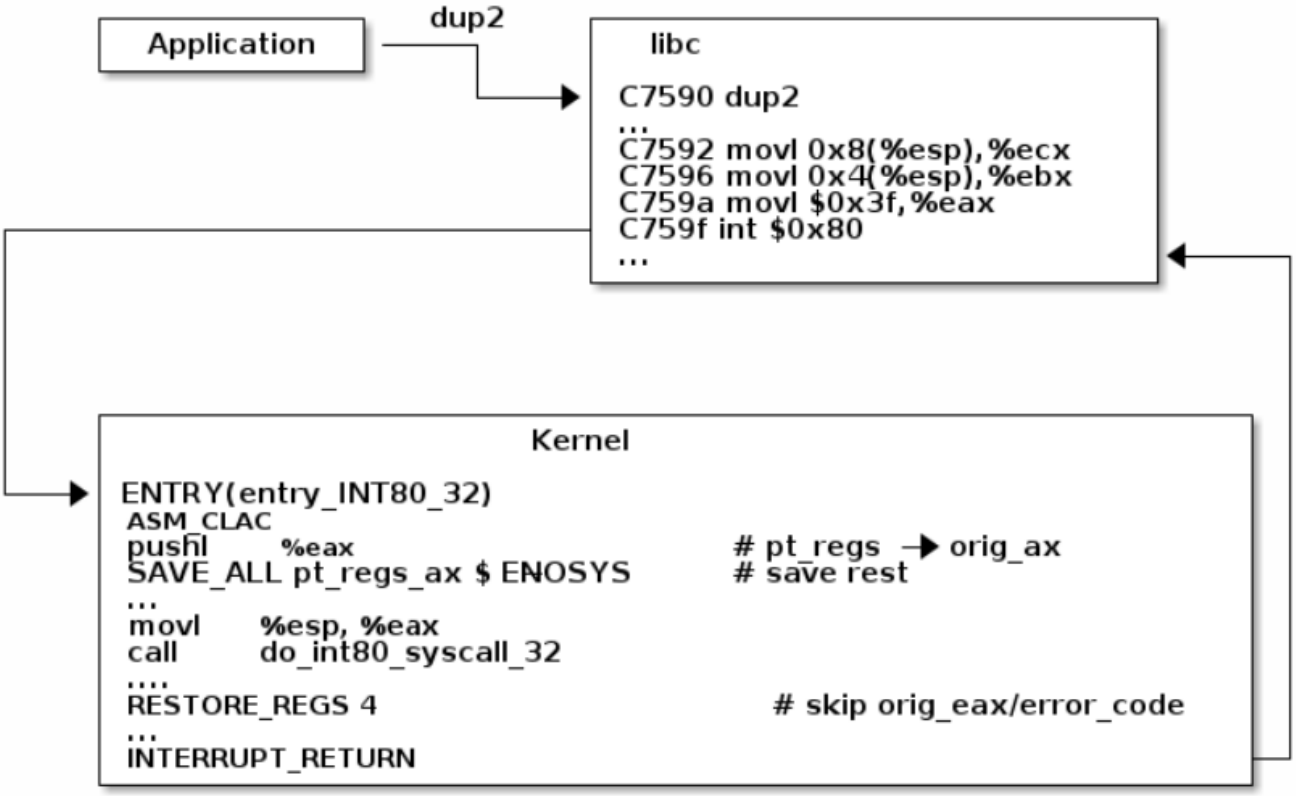
função C via uma estrutura de dados de pilha, mas o *kernel* os espera em registradores específicos. A função C, então, copia os argumentos para estes registradores.

- Para o kernel Linux/x86-32, os registradores para a passagem de parâmetros entre o espaço de usuário e o *kernel* estão definidos no arquivo *arch/x86/ia32entry.S*, e são os seguintes:

Registrador	**Descrição**
%eax	Número da *system call*
%ebx	Parâmetro 1
%ecx	Parâmetro 2
%edx	Parâmetro 3
%esi	Parâmetro 4
%edi	Parâmetro 5
%ebp	Parâmetro 6

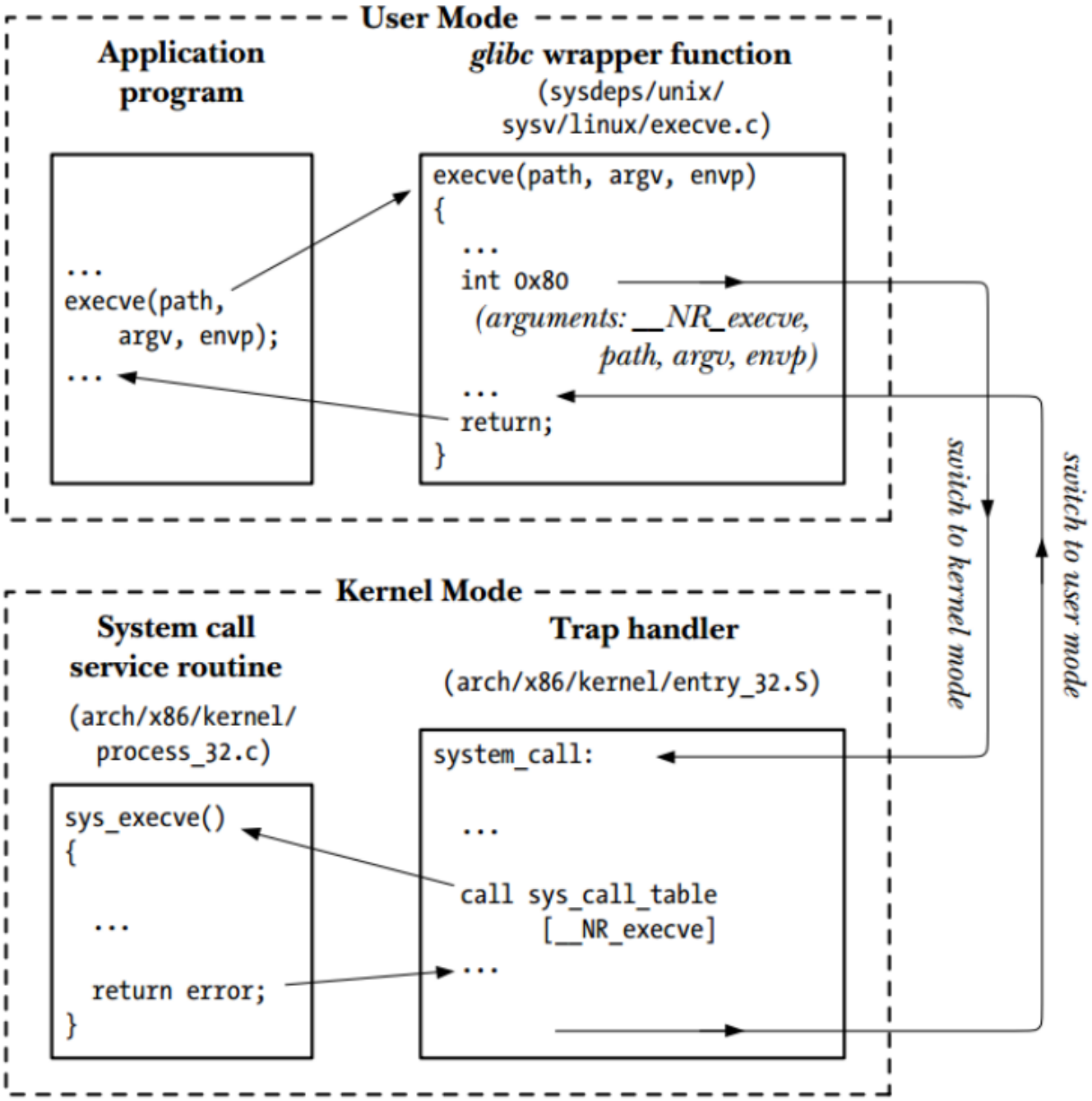
1. Uma vez que a *system call* entra no modo *kernel*, o *kernel* necessita de algum método para identificar qual é a *system call* que está sendo invocada. Para isso, a função C copia o número da *system call* em um registrador específico da CPU (*eax*).
2. A função C executa uma instrução de máquina (*int 0x80*), o que força o processador a trocar do modo de usuário para o modo *kernel* e executar o código apontado pela localização *0x80* (128, em decimal) do vetor de exceções do sistema.
3. Em resposta à exceção da posição *0x80*, o kernel invoca sua rotina *system_call()* (localizada no arquivo *assembly arch/i386/entry.S*, no *kernel* Linux) para lidar com a exceção. Este *handler*:
   1. Salva os valores dos registradores na pilha do *kernel*.
   2. Verifica a validade do número da *system call*.
   3. Invoca a rotina apropriada para a *system call*, a qual é encontrada utilizando o número da *system call* como índice em uma tabela que contém todas as rotinas de *system calls* (no *kernel* Linux, é a variável *sys_call_table*). Se a rotina de serviço da *system call* possuir algum parâmetro, primeiro ele verifica a sua validade (por exemplo, é verificado se o endereço aponta para posições válidas na memória de usuário). Então a rotina de serviço executa a tarefa requisitada, o que pode envolver a modificação de valores nos endereços especificados nos parâmetros recebidos e também a transferência de dados entre memória de usuário e memória do *kernel* (operações de *I/O*). Por fim, a rotina de serviço retorna o estado do resultado por meio da chamada *system_call()*.
   4. Restaura os valores dos registradores da pilha do *kernel* e coloca o valor de retorno da *system call* na pilha.
   5. Retorna para a função C, ao mesmo tempo que retorna o processador para o modo usuário, utilizando a instrução *iret*.
4. Se o valor de retorno da rotina de serviço da *system call* sinalizar algum erro, a função C atribui o valor recebido à variável global *errno*. A função C então retorna para quem a chamou, fornecendo um número inteiro como retorno, indicando o sucesso/falha da *system call*.

A figura abaixo apresenta uma visão geral dos passos mencionados acima. Uma aplicação, ao chamar uma função da *libc*, está, na verdade, invocando uma *system call* ao *kernel*. A função da *libc* prepara os parâmetros que serão enviados ao *kernel* e, ao fim do processo, prepara o retorno que será dado ao processo do usuário.



6.2.2. Exemplo

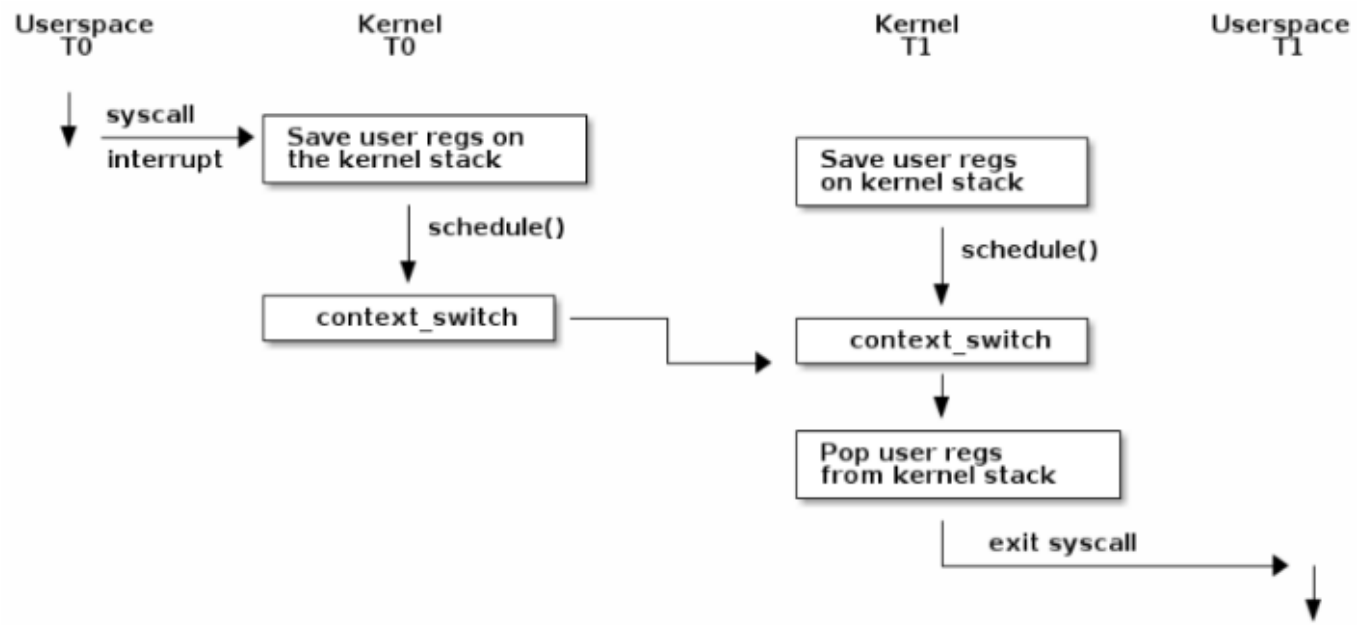
A figura abaixo apresenta a sequência explicitada acima, usando o exemplo da *system call execve()*. No Linux/x86-32, *execve()* é a *system call* número 11 (*__NR_execve*). Portanto, no vetor *sys_call_table*, a entrada 11 contém o endereço de *sys_execve()*, a rotina de serviço para esta *system call*. No Linux, rotinas de serviço de *system calls* geralmente têm nomes na forma *sys_xyz()*, onde *xyz* é o nome da ''system call' 'em questão.



Versões mais recentes da arquitetura x86-32 implementam as instruções *sysenter* e *sysexit*, as quais proveem um método para entrar e sair do modo *kernel* mais rápido do que a instrução *int 0x80* convencional. O uso destas novas instruções são suportadas pela versão 2.6 do *kernel* do Linux, e para as versões 2.3.2 ou maiores do *glibc*.

6.3. *Switch context*

A figura abaixo exibe uma visão geral do procedimento da troca de contexto em um kernel Linux. A *thread* **T0** realiza uma operação de *I/O* no disco. Antes que uma troca de contexto possa ocorrer, é necessário realizar um chaveamento para o modo *kernel*. Neste ponto, os registradores do espaço de usuário são salvos na pilha do *kernel*, assunto que será visto mais adiante. Em algum momento, a função *schedule()* é chamada para decidir que uma troca de contexto deve ocorrer de **T0** para **T1**, pois **T0** está bloqueada esperando a operação de *I/O* completar.



6.3.1. Principais causas para a troca de contexto entre processos

- ***Multitasking***

É muito comum a utilização de algoritmos de escalonamento para dividir o tempo de CPU entre diversos processos, escolhendo quais processos devem executar e se o processo atual deve interromper o seu processamento para que outro processo tome o seu lugar. Retirar um processo da CPU envolve a troca de contexto entre processos, e ela pode ser disparada porque o processo tornou-se inapto a executar, talvez porque esteja esperando por uma operação de *I/O*, ou então a finalização de uma operação de sincronização.

Em um sistema *multitasking* preemptivo, o escalonador também deve trocar processos que estão executando e poderiam continuar executando por mais algum tempo. Para garantir que o tempo de CPU seja dividido justamente entre todos os processos, escalonadores preemptivos frequentemente configuram uma interrupção de *timer*, que dispara sempre que um processo excede a sua quantidade de tempo. Esta interrupção garante que o escalonador ganhe o controle da CPU e realize uma troca de contexto.

- **Interrupções**

Arquiteturas modernas são guiadas por interrupções. Isso significa que, se a CPU realiza uma requisição para recuperar dados de um disco, por exemplo, ela não precisa realizar uma operação de *busy wait* até que a leitura complete; ela pode simplesmente iniciar a requisição (para o dispositivo de *I/O*) e realizar alguma outra tarefa até que a resposta retorne. Quando a leitura é finalizada, a CPU é interrompida e recebe resultado da leitura. Para isso, um programa chamado **tratador de interrupções** (*interrupt handler*) é utilizado.

Quando uma interrupção ocorre, o hardware automaticamente troca uma parte do contexto, deixando ao menos o necessário para permitir ao tratador de interrupções retornar ao código interrompido. O tratador pode salvar contextos adicionais, dependendo dos detalhes do hardware em particular e do projeto de software. Frequentemente, somente uma parte mínima do contexto é trocada a fim de minimizar a quantidade de tempo gasto tratando a interrupção. O *kernel* não invoca ou escalona um processo especial para lidar com as interrupções mas, em vez disso, o tratador de interrupções é executado, geralmente de maneira parcial, no contexto estabelecido no início do tratamento da interrupção. Uma vez que o serviço de interrupção está completo, o contexto anterior à interrupção é restaurado para que o processo interrompido possa retomar a execução.

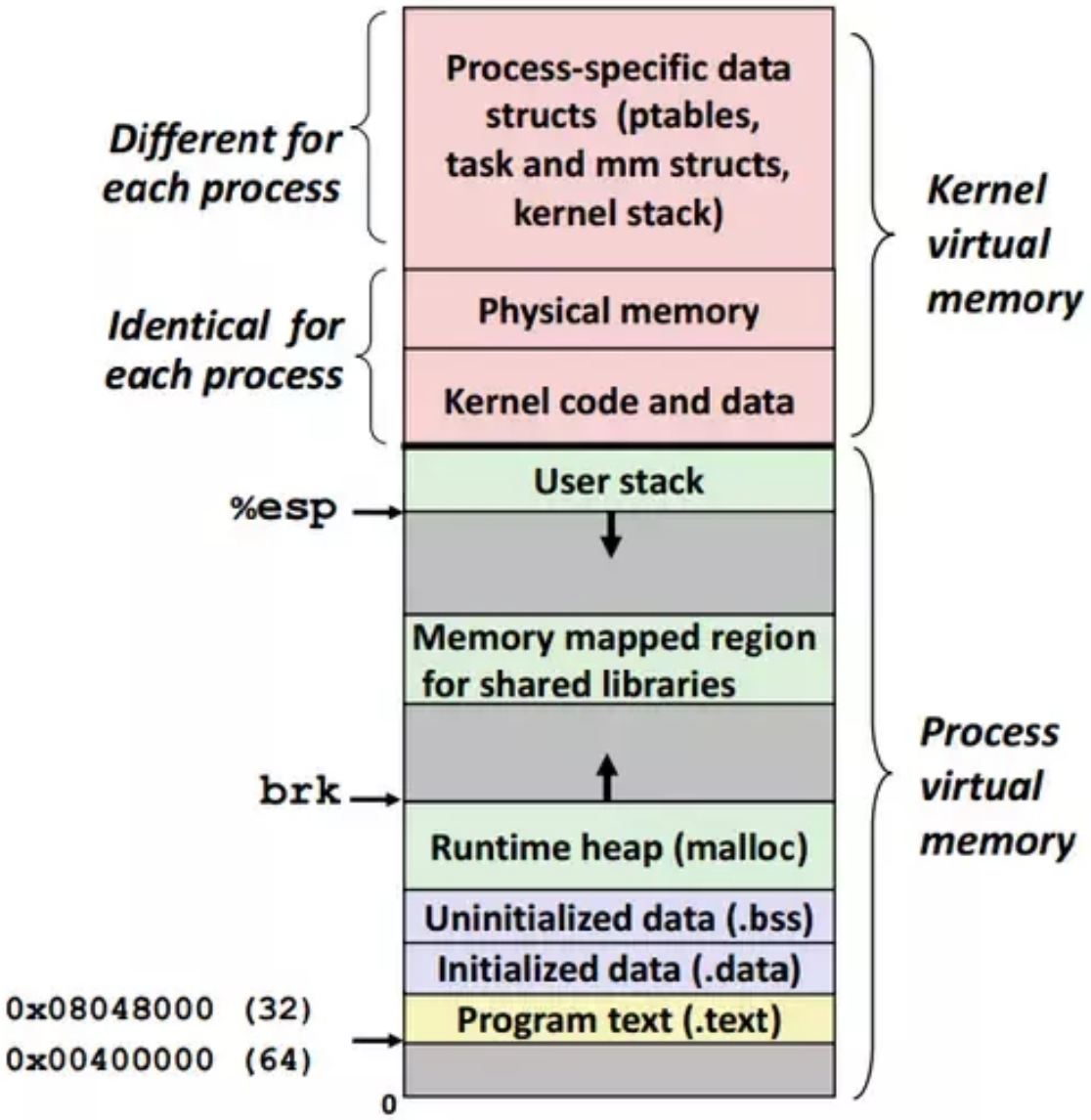
6.4. *Kernel space*

A troca de contexto entre threads de um mesmo processo requer apenas o salvamento dos registradores utilizados pela *thread* anterior. Devido ao fato de que as threads compartilham dos mesmos recursos, esta tarefa é relativamente simples. Porém, quando a troca de contexto envolve processos diferentes, são necessários alguns passos a mais para que o contexto do processo anterior seja salvo antes que a CPU possa carregar o contexto do próximo processo.

A troca de contexto entre dois processos exige que o espaço de endereçamento do primeiro seja devidamente salvo, e que o espaço de endereçamento do segundo processo seja carregado.

No entanto, considerando as plataformas de computação atuais, em que processadores *multicores* são capazes de executar diversas aplicações simultaneamente, é comum o cenário onde vários processos realizam *system calls* ao *kernel* e logo em seguida sofram preempção, antes mesmo de receber a resposta da *system call*. Para lidar com estes casos, os serviços do *kernel* são projetados para serem reentrantes, permitindo que múltiplos processos entrem no espaço do *kernel* e utilizem estes serviços. Para realizar o controle de quais processos estão executando quais *system calls*, e quais são os seus contextos dentro do *kernel*, cada processo recebe sua própria pilha de *kernel* privada, com o objetivo de manter os dados de chamadas de funções, armazenar dados locais das funções de *kernel*, entre outros usos.

Na imagem abaixo podemos ver a estrutura geral da memória de um processo. A parte mais alta exibe a memória virtual do kernel, que possui uma parte que é idêntica para cada processo, envolvendo dados e código do *kernel*, e outra parte que é diferente para cada processo, e é nesta área onde são armazenados os dados específicos de cada processo relacionados ao *kernel*, como *ptables* e as pilhas de *kernel*, por exemplo. Embora a memória do *kernel* esteja no espaço de endereçamento do processo, ela é uma área protegida e o processo não pode acessá-la diretamente.



Como foi visto na figura anterior, o *kernel* é mapeado no espaço de endereçamento de cada processo. Dentro do espaço utilizado pelo *kernel* há uma pilha especial para cada processo. A representação desta pilha é apresentada na figura abaixo. No kernel do Linux, cada pilha contém as informações do processo associado, representado pela *struct thread_info*, a qual armazena informações sobre o descritor do processo.

A pilha do kernel está diretamente mapeada à memória física, obrigando que os endereços estejam dispostos fisicamente em uma região contígua de memória. Para sistemas x86-32, a pilha do kernel possui, por padrão, 8 KB de tamanho, sendo possível configurá-la para tamanhos de 4 KB (durante a build do *kernel*) e 16 KB (para sistemas x86-64).

6.5. Referências

TANENBAUM, Andrew S., BOS, Herbert. **Modern Operating Systems**. Fourth edition, 2015. Pearson.

LOVE, Robert. **Linux Kernel Development - A thorough guide to the design and**

implementation of the Linux kernel. Third edition. Developer's Library.

The Linux Kernel, 5.10.14. **System Calls - Linux system calls implementation**. Disponível em: <https://linux-kernel-labs.github.io/refs/heads/master/lectures/syscalls.html>. Acesso em: 26 de julho de 2021.

KERRISK, Michael. **The Linux Programming Interface - A Linux and UNIX System Programming Handbook**. no starch press, San Francisco.

BHARADWAJ, Raghu. **Mastering Linux Kernel Development: A Kernel Developer's Reference Manual**. Packt Publishing.

7. MMU for Paging

7.1. Conceitos importantes

A seguir são definidos alguns conceitos importantes para o entendimento dos conceitos de MMU e Paging:

- **Virtual Address** - O endereço usado pelo processador (pela aplicação em execução). O Stack Pointer, Instruction Counter e registradores de retorno usam endereços virtuais. Esses endereços não necessariamente são únicos, e do ponto de vista do programador, os endereços vão de 0 até o valor definido como tamanho máximo do espaço de endereçamento da aplicação. Portanto, dois programas rodando em um mesmo sistema podem por exemplo apontar para o mesmo endereço virtual, mas que na memória física são completamente diferentes.

- **Physical Address** - Endereço na memória principal (RAM), tido a partir do processo de tradução do endereço virtual para determinada aplicação.

- **Page/Section** - Uma página(Page) é um espaço de endereçamento na memória da aplicação com tamanho definido pela arquitetura. Páginas de memória de um programa não necessariamente estão carregadas na memória RAM, e podem estar armazenadas no disco, por exemplo. Páginas são carregadas pelo sistema operacional de acordo com a necessidade e o espaço disponível. Quando uma página da memória virtual é carregada para a memória RAM, ela é disposta em um frame da memória física. A memória física é dividida em frames. Seções(Sections) são semelhantes a páginas, porém maiores.

- **Page Frame** - Espaço na memória física do tamanho de uma página. A memória física contém um determinado número de frames de um tamanho pré definido pela arquitetura.

- **Page Table/Page Directory** - Um vetor de registros usados para tradução de endereços virtuais para físicos. Para cada programa há uma tabela de páginas(Page tables). As tabelas de páginas primárias(aquelas pelas quais o processo de tradução de endereço se inicia) podem ser chamadas de tabelas de diretório(Page directory).

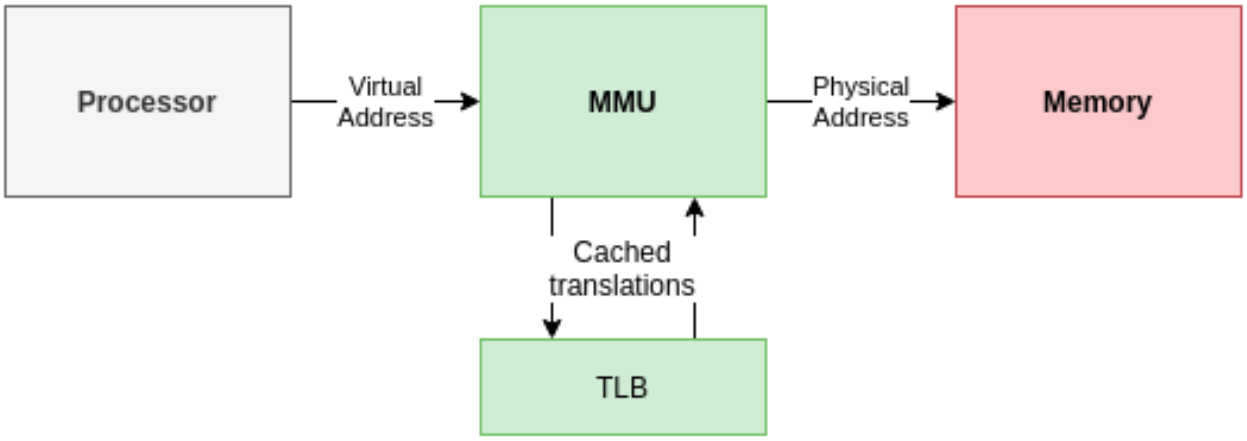
- **ASID(Address Space Identifier)** - Identificador do espaço de armazenamento.

- **TLB(Table Lookahead Buffer)**-Buffer com os últimos endereços virtuais traduzidos. É mantido pela MMU.

7.2. MMU

A MMU é um componente de hardware responsável por realizar a tradução de endereços virtuais para endereços físicos quando a paginação está habilitada. Todos os endereços físicos absolutos são

calculados com base nas entradas definidas na tabela de diretório; esse comportamento restringe os endereços alcançáveis para aqueles mapeados na tabela de diretório de um processo. Podem existir diferentes tabelas de diretório, o que torna possível limitar o acesso de diferentes processos a diferentes segmentos de memória. Essa restrição também pode levar em consideração aspectos como o modo em que o processador está operando ou o tratamento de exceções.



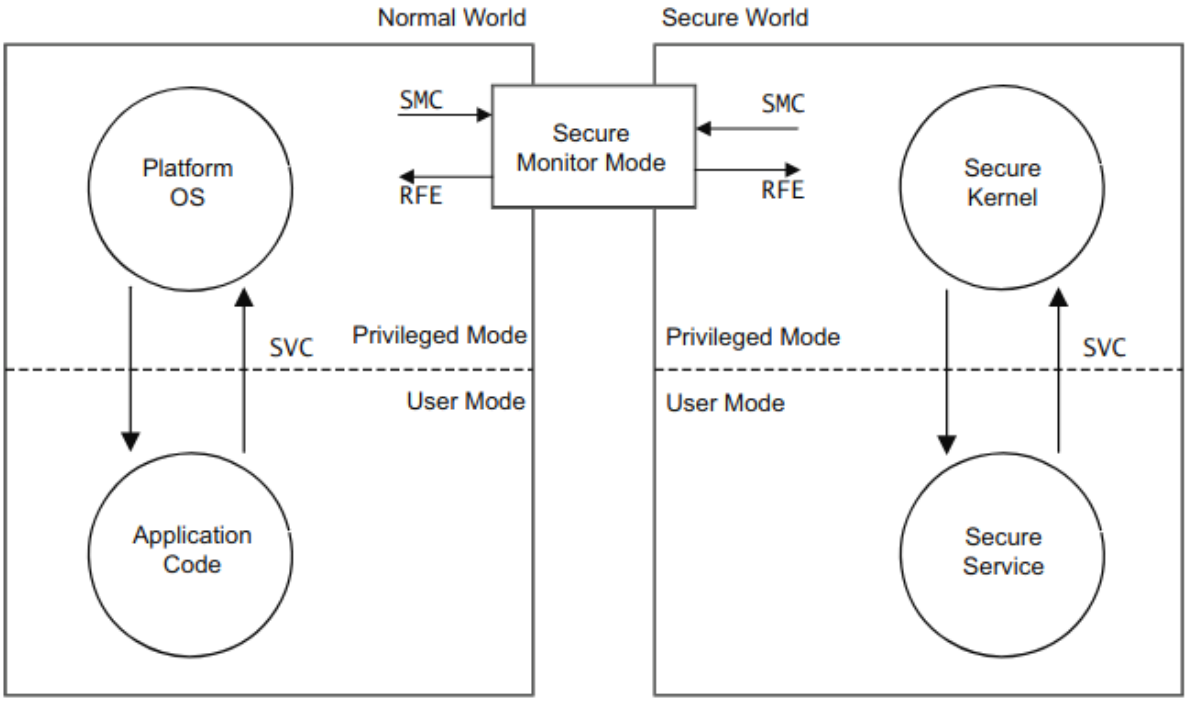
7.3. Paging

O conceito de paginação dentro de sistemas operacionais é fortemente atrelado à ideia de memória virtual. Em um sistema sem memória virtual, os endereços de memória referenciados pelo programa em execução no processador são exatamente os endereços da memória principal que se pretende acessar. Isso implica que a memória endereçável pelo processador se limita ao tamanho da memória. Em um sistema com endereçamento virtual, cada programa tem um espaço de endereçamento próprio, que vai de 0 até um tamanho definido pelo sistema operacional. Cada programa também detém uma Page Table, que é usada para mapear os endereços virtuais utilizados pelo programa para endereços físicos. Cada entrada da tabela de páginas contém ou um endereço físico para onde o endereço virtual está mapeado, ou um indicador de que aquela página não está na memória principal (de rápido acesso), e precisa ser carregada da memória secundária (significativamente mais lenta).
Dessa forma, para um programa executar, não é necessário que todos os dados deste estejam em memória de forma contínua (tanto em espaço quanto em tempo). O programa tem acesso ao espaço de endereçamento máximo possibilitado pela arquitetura, e não é limitado pela memória primária (RAM) instalada. Por exemplo, um programa executando em uma arquitetura de 32 bits, terá acesso a um espaço de endereçamento de 2^{32} bytes (4GB), mesmo que a máquina tenha apenas 1GB de memória RAM instalada.
Dentro da arquitetura do EPOS, a abstração da MMU é feita pela classe MMU_Common, que é especializada para cada arquitetura-alvo. Essa classe leva é parametrizada (ela é um template) com os valores para DIRECTORY_BITS, PAGE_BITS, OFFSET_BITS que definem quantos bits dos endereços são usados para cada parte da tradução. O DIRECTORY_BITS indica o número de bits utilizados para acessar a tabela de diretório, PAGE_BITS para a tabela de paginação e OFFSET_BITS é concatenado diretamente ao final da tradução para completar o endereço físico, e basicamente move o acesso a memória dentro de uma página específica. O tamanho das páginas dentro do EPOS é dado por 2 elevado ao valor de PAGE_BITS.

7.4. MMU - ARMv7

Além das funcionalidades básicas conceitualmente atribuídas a uma MMU, a VMSA (Virtual Memory

System Architecture) do ARMv7 possui extensões que alteram o funcionamento do endereçamento virtual. Com base no escopo dos próximos entregáveis serão brevemente apresentadas apenas 3: Security Extensions, Virtualization e Large Physical Address.
Security extensions, também referenciada como TrustZone, provê a possibilidade de separar código e dados considerados sensíveis ao isolá-los em uma região da memória chamada de secure world. Utilizando essa extensão o hardware garante que nenhum recurso presente no secure world seja acessível do normal world (local onde as aplicações consideradas não seguras serão salvas e executadas); ao habilitar essa extensão um novo bit (o bit NS) é adicionado à todas as transações que envolvem acesso a memória, tornando possível a divisão da memória entre as aplicações do normal world e secure world.
Por fim, é possível que qualquer core execute código referente a qualquer um dos modos; para tal é utilizado um novo modo: o monitor mode; é através dele que quaisquer modificações necessárias no sistema para execução de códigos de níveis diferentes é feita.



Virtualization é uma extensão da VMSA que possibilita aos processadores o acesso a um novo modo, o hypervisor mode; esse modo possui um nível de privilégio maior do que os níveis de privilégio padrões do ARMv7. Os softwares de virtualização, chamados hypervisors, utilizarão esse método para gerenciar a execução dos múltiplos sistemas operacionais em execução.
Essa extensão se relaciona a VMSA devido ao fato de seu suporte implicar na necessidade de um espaço de endereçamento maior(uso da extensão large physical address) e alteração no funcionamento de tradução de endereços virtuais. Tais tópicos não serão desenvolvidos devido a complexidade e falta de relação com a idéia principal de paginação.
A última extensão, Large Physical Address, aumenta a faixa de endereços físicos endereçáveis de 4GB para 1TB. Em termos de MMU essa extensão adiciona um nível a mais no processo de tradução de endereços virtuais, mantendo os endereços virtuais com 32 bits. Como citado anteriormente essa extensão é necessária para que possa ocorrer virtualização.

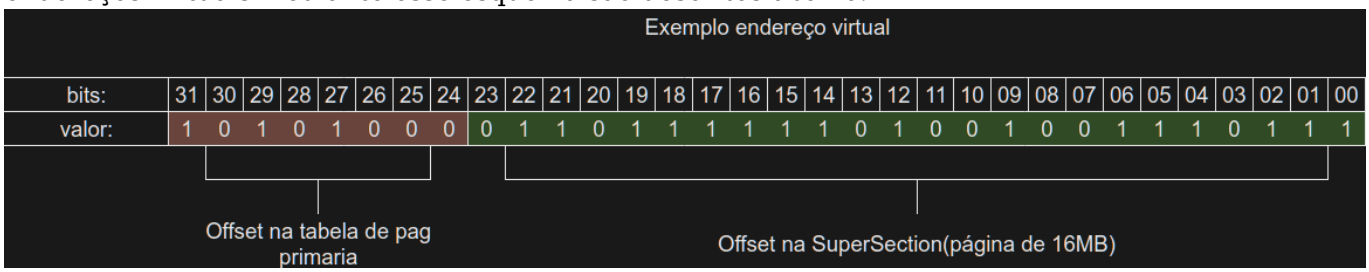
7.5. Paging - ARMv7

A VMSA do ARMv7 possui quatro esquemas de paginação padrão. A diversidade de esquemas torna possível explorar as necessidades específicas das aplicações que serão executadas e levá-las em consideração na hora de configurar a MMU. A principal diferença entre os quatro esquemas é a unidade de fragmentação da memória. Os esquemas que utilizam Sections(seções) dividem a

memória em fragmentos maiores, enquanto os esquemas que utilizam páginas usam valores menores. A seguir serão apresentados os quatro esquemas e o significado de cada campo do endereço virtual para cada um deles.

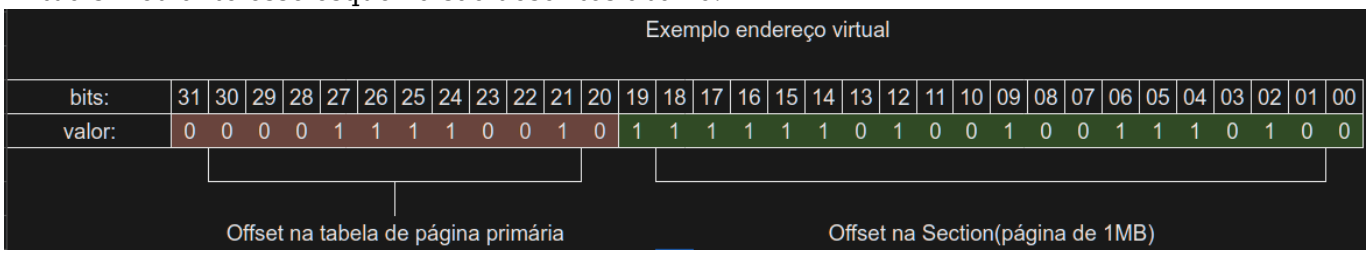
7.5.1. Super Section

Esquema de endereçamento virtual de 1 nível. Ou seja, ao adotar esse modelo os processos necessitam de apenas uma tabela de páginas para traduzir seus endereços virtuais. Nesse esquema a unidade de fragmentação de memória são as Super Sections(também é possível imaginar as Super Sections como páginas), fragmentos de memória física de 16MB. Os significados de cada campo dos endereços virtuais mediante esse esquema são descritos abaixo.



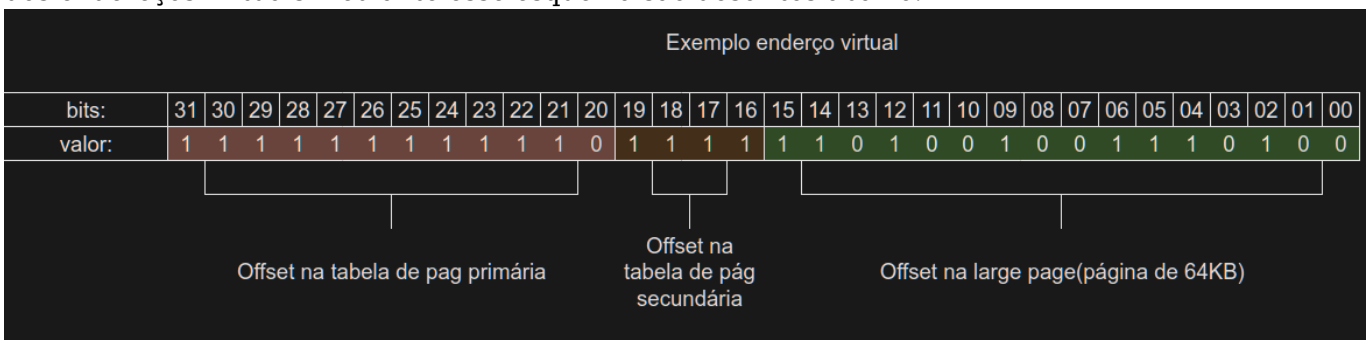
7.5.2. Section

Esquema de endereçamento virtual de 1 nível. Ou seja, ao adotar esse modelo os processos necessitam de apenas uma tabela de páginas para traduzir seus endereços virtuais. Nesse esquema a unidade de fragmentação de memória são as Sections(também é possível imaginar as Sections como páginas), fragmentos de memória física de 1MB. Os significados de cada campo dos endereços virtuais mediante esse esquema são descritos abaixo.



7.5.3. Large Page

Esquema de endereçamento virtual de 2 níveis. Ou seja, ao adotar esse modelo os processos necessitam de percorrer duas tabelas de páginas para traduzir seus endereços virtuais. Nesse esquema a unidade de fragmentação de memória são as Large Pages(também é possível imaginar as Large Pages como páginas), fragmentos de memória física de 64KB. Os significados de cada campo dos endereços virtuais mediante esse esquema são descritos abaixo.



7.5.4. Small Page

Esquema de endereçamento virtual de 2 níveis. Ou seja, ao adotar esse modelo os processos necessitam de percorrer duas tabelas de páginas para traduzir seus endereços virtuais. Nesse esquema a unidade de fragmentação de memória são as Small Pages(também é possível imaginar as

Small Pages como páginas), fragmentos de memória física de 4KB. Os significados de cada campo dos endereços virtuais mediante esse esquema são descritos abaixo.

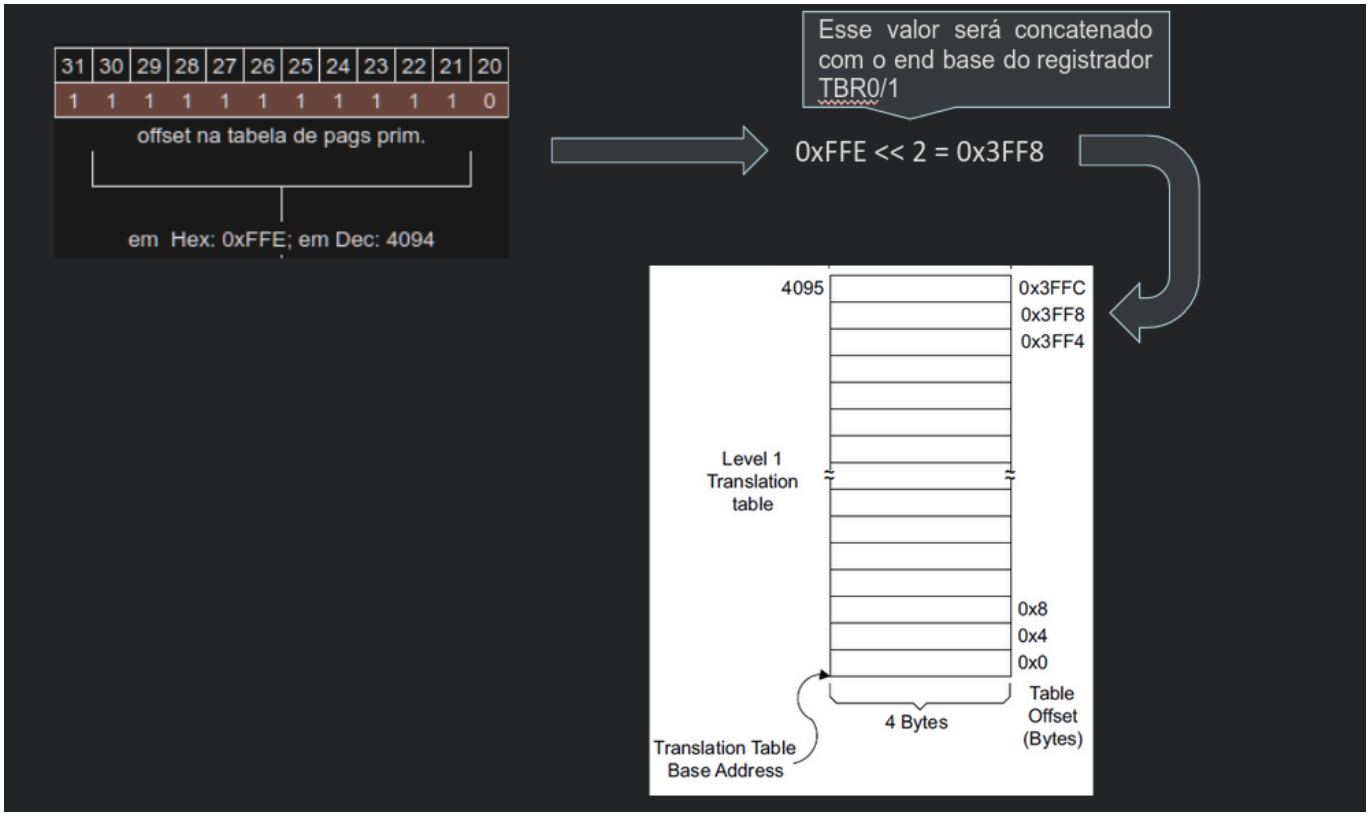
Exemplo end virtual

bits:	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	09	08	07	06	05	04	03	02	01	00
valor:	1	1	1	1	1	1	1	1	1	1	1	0	1	1	1	1	1	1	0	1	0	0	1	0	0	1	1	1	0	1	1	1
	offset na tabela de pags prim.												offset na tabela de pags sec.								offset da página final											

7.6. Exemplo de tradução de endereço no ARMv7

Supondo o uso de páginas de 4KB(small pages) e N = 1 e que a consulta a TLB não obteve sucesso, a tradução de endereço se dá por:

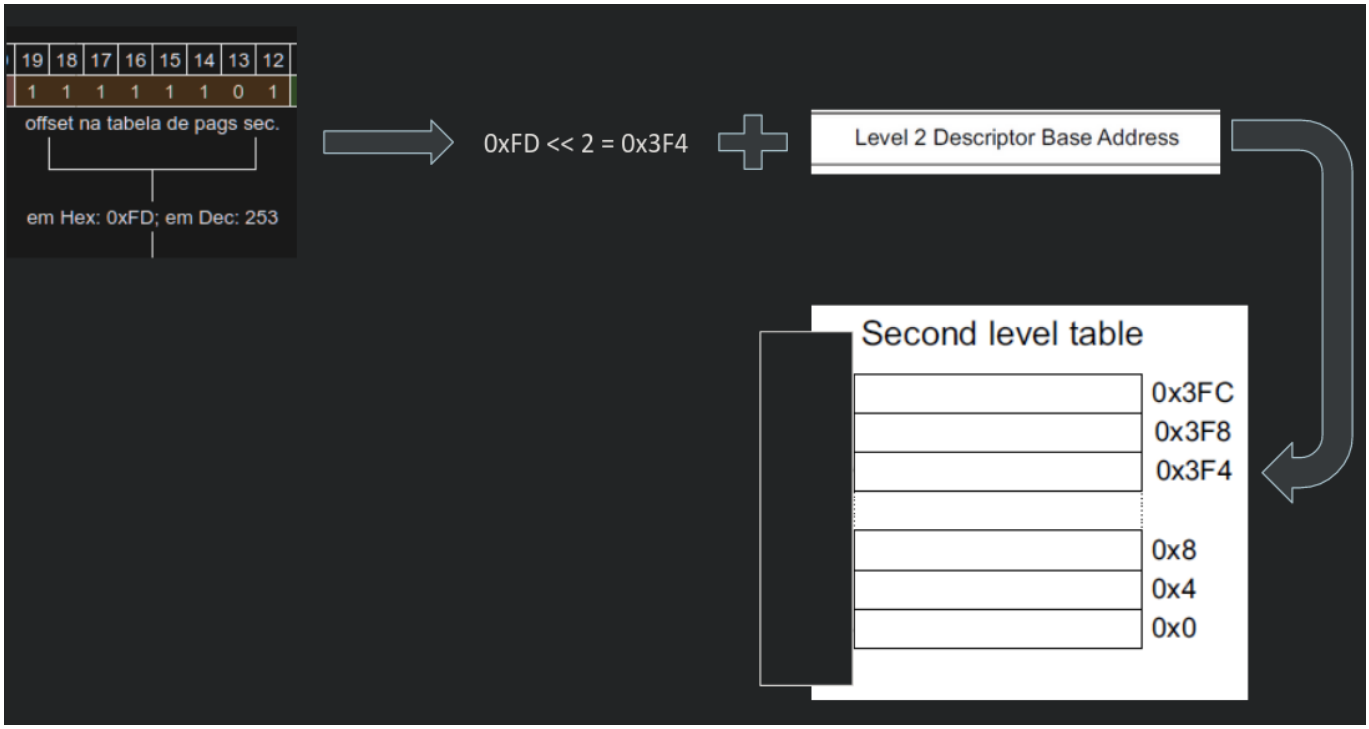
1. É selecionado o endereço base da tabela de diretório a ser consultado usando os bits 31-14 do TTBR0, ou TTBR1 se o endereço for igual ou maior do que 0x8000000.
2. Os bits 31-20 são concatenados ao endereço base da tabela de diretório. O resultado é deslocado 2 bits para a esquerda, formando um endereço de 32 bits que aponta para uma entrada na tabela de diretório.



3. A entrada da tabela de páginas primária tem a seguinte estrutura:

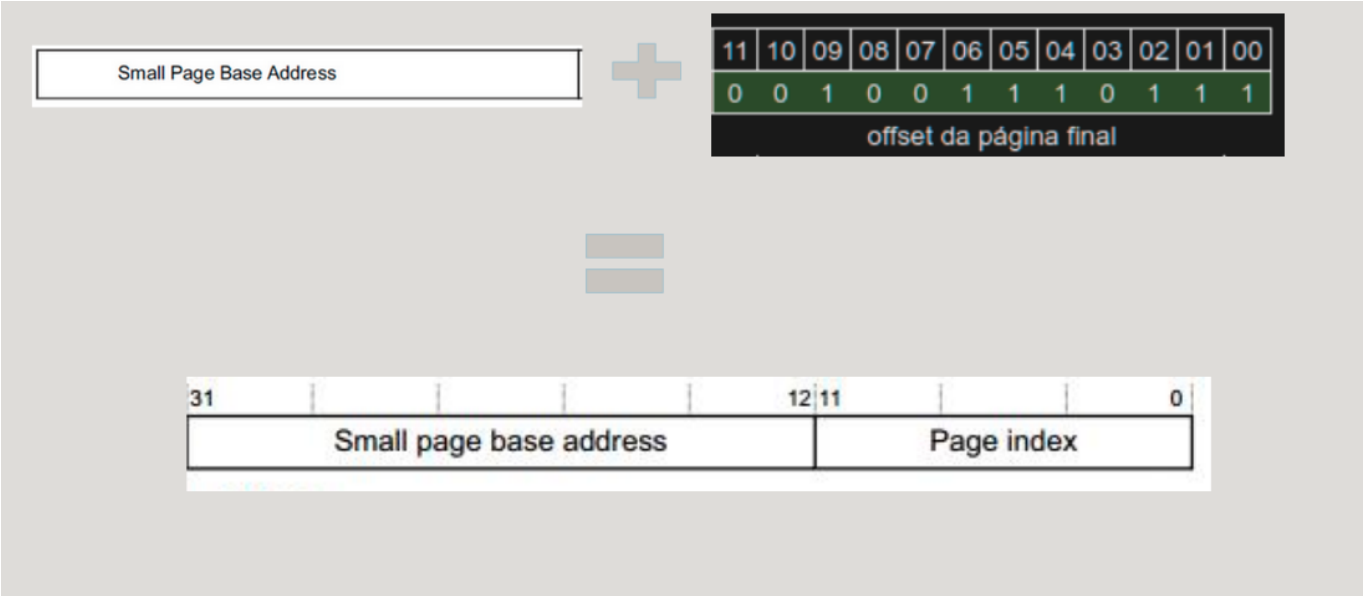
Pointer to 2nd level page table	Level 2 Descriptor Base Address	P	Domain	SBZ	0	1

4. Os bits 31-10 do valor no endereço formado são concatenados aos bits 19-12 do endereço de entrada; o resultado é descolado 2 bits para a esquerda;



5. O endereço formado é usado para acessar uma entrada específica na tabela de páginas secundária apontada pela entrada do item 3.

Se a entrada for válida (bit na posição 1 igual a 1), ou seja, estiver na memória principal, os bits 31-12 do valor no endereço são concatenados aos bits 11-0 do endereço de entrada (bits de offset), e o valor resultante é um endereço físico na memória.



7.7. Ativação da MMU no ARMv7

Abaixo segue um exemplo de ativação da MMU. A implementação não segue as definições da arquitetura do EPOS para simplificar, porém utiliza alguns termos dela para facilitar a conexão com ela. As constantes PD_ENTRIES, PT_ENTRIES, DIRECTORY_BITS e PAGE_BITS são definidas na classe MMU_Common do EPOS, por exemplo. As estruturas Page_Directory e Page_Table também existem na classe MMU_ARMv7, porém são classes mais complexas. Os mecanismos usados para garantir o alinhamento em memória dos endereços das estruturas também são outros, porém no exemplo é utilizado um alocador específico.

struct Page_Directory { PD_Entry entries[PD_ENTRIES]; }; struct Page_Table { PT_Entry entries[PT_ENTRIES]; }; void* pd_address = aligned_alloc(0x1 << 14, sizeof(Page_Directory)); Page_Directory* page_dir = reinterpret_cast<Page_Directory *> pd_address; // Para cada tabela de páginas de segundo nível, é necessário alocar o espaço // e apontar elas em cada linha da Page_Directory. O alinhamento de cada // tabela de páginas precisa ser alinhada em PAGE_OFFSET, ou seja, 12 bits // ou 0x1 << PAGE_OFFSET // Faz setup do TTBCR, com N = 0 uint32_t ttbcr; // Carrega o valor atual do registrador __asm__ __volatile__ ("mrc p15, 0, %r, c2, c0, 2" : "=r"(ttbcr) :); // Zera os 3 últimos bits (N), indicando uso apenas do TTBR0 ttbcr &= 0xFFFFFFF8; // Escreve de volta __asm__ __volatile__ ("mcr p15, 0, %r, c2, c0, 2" : : "r"(ttbcr)); // Carrega o endereço base do diretório (page table de primeiro nível) uint32_t ttbr0; // Carrega o ttbr0 existente __asm__ __volatile__ ("mrc p15, 0, %r, c2, c0, 0" : "=r"(ttbr0) :); // Aplica máscara 0b 0000 0000 0000 0000 0000 0011 1111 1111 // Zera os bits [31-14], já que N=0 ttbr0 &= 0x000003FF; // Preenche os bits [31-14] com endereço alinhado alocado anteriormente ttbr0 += pd_address // Escreve de volta __asm__ __volatile__ ("mcr p15, 0, %r, c2, c0, 0" : : "r"(ttbr0)); // Por fim, ativa a MMU setando o bit M do registrador de controle // SCTLR (System Control Register) CRn = c1, Op1 = 0, CRm = c0, Op2 = 0 __asm__ __volatile__ ("mrc p15, 0, r1, c1, c0, 0 ;Read control register \n" "orr R1, #0x1 ;Set M bit \n" "mcr p15, 0, r1, c1, c0, 0 ;Write control register and enable MMU \n");

7.8. Referências

Acesso a registradores:

- https://developer.arm.com/documentaation/100511/0401/system-control/register-summary/cp15-system-control-registers-grouped-by-crn-order

Assembler guide:

- https://www.keil.com/support/man/docs/armasm/armasm_dom1361289850039.htm

Funcionamento da paginação e memória virtual:

- https://www.youtube.com/watch?v=qcBIvnQt0Bw&list=PLiwt1iVUib9s2Uo5BeYmwkDFUh70fJPxX&index=1&ab_channel=DavidBlack-Schaffer
- https://www.youtube.com/watch?v=zP4tBRpK3iM&ab_channel=MallikarjunK
- https://sudonull.com/post/11570-Virtual-memory-in-ARMv7

Noções gerais do processador e uso dos registradores:

- https://moodle.ufsc.br/pluginfile.php/4417210/mod_resource/content/1/DEN0013D_cortex_a_series_PG.pdf

8. Task Context Switching

8.1. What is Context Switching

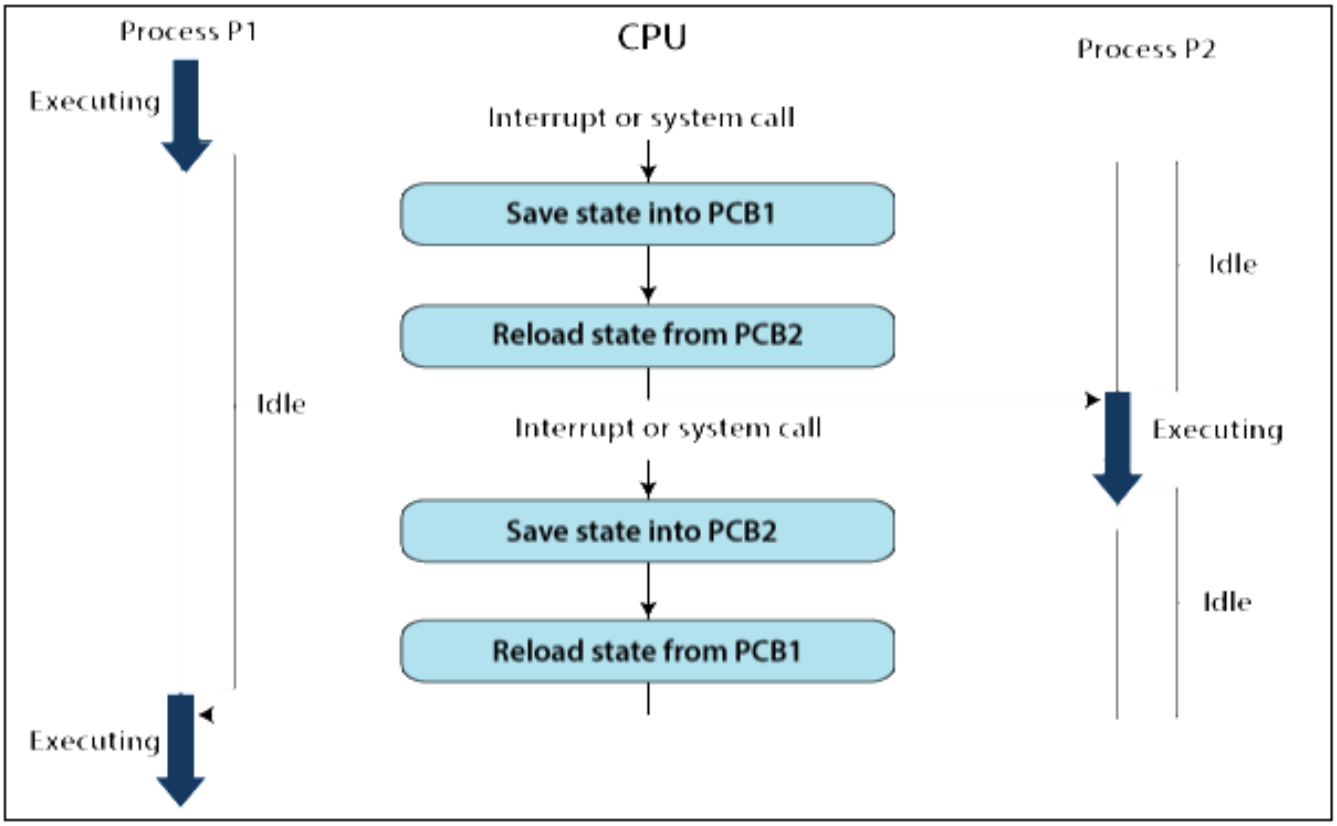
Context Switching is the saving and restoring of computational state when switching between different threads or processes, known as tasks. This is an essential feature of a multitasking

operating system so that a task can be restored and resume execution at a later point. This allows multiple processes to share a single central processing unit (CPU).

There are multiple cases where context switching may occur:

- In a multitasking context, it refers to the action of storing the system state for one task, so that one task can be paused and another task resumed.
- A context switch can also occur as the result of an interrupt, such as when a task needs to access disk storage, freeing up CPU time for other tasks.
- Some operating systems also require a context switch to move between user mode and kernel mode tasks.

8.2. How does a Context Switching happen



In the figure above, you can see that initially process P1 is in running state and process P2 is in ready state. Now, when an interruption occurs and calls for a context switch, you should switch process P1 from running state to ready state after saving the context, and then switch process P2 from ready state to running state. The following steps will be performed:

1. First, the context of process P1, that is, the process present in the execution state, will be saved in the Process Control Block of process P1, that is, PCB1.
2. Now, you must move PCB1 to the relevant queue, i.e., ready queue, I/O queue, waiting queue, etc.
3. In the ready state, select the new process that should be executed, i.e. process, P2.
4. Now update the Process Control Block of process P2, ie PCB2, setting the process state to run. If process P2 was previously executed by the CPU, you can get the position of the last instruction executed so that it can resume execution of P2.
5. Likewise, if you want to run process P1 again, you must follow the same steps mentioned above (from step 1 to 4).

Context switching is used to achieve multitasking, that is, time-sharing multiprogramming. Multitasking gives users the illusion that more than one process is running at the same time. But, in reality, only one task is being performed at any given time by a processor. Here, context switching is so fast that the user feels that the CPU is performing more than one task at the same time.

8.3. What is a PCB

A process control block (PCB) is a data structure used by computer operating systems to store all information about a process. It is also known as a process descriptor. When a process is created (started or installed), the operating system creates a corresponding process control block.

Process-Id
Process state
Process Priority
Accounting Information
Program Counter
CPU Register
PCB Pointers
.....

Process Control Block

The figure above shows the main information that a PCB may include in its structure. While the details of these structures are system dependent, the common elements fall into three main categories:

- Process identification
- Process state
- Process control

Process identification data includes a unique identifier for the process (almost invariably an integer) and, in a multi-role-multitasking system, data such as parent process identifier, user identifier, user group identifier, etc. The process ID is particularly relevant as it is often used to cross-reference the definitions defined above, for example to show which process is using which I/O devices or memory areas.

The defined process state data or status of a process when it is suspended, allowing the operating system to restart later. This always includes the contents of general purpose CPU registers, a CPU process status word, stack and frame pointers, and so on. During a context switch, the running process is stopped and another process is completed. The kernel must stop an execution of the running process, copy the values from the hardware registers to its PCB, and update the hardware registers with the values from the PCB of the new process.

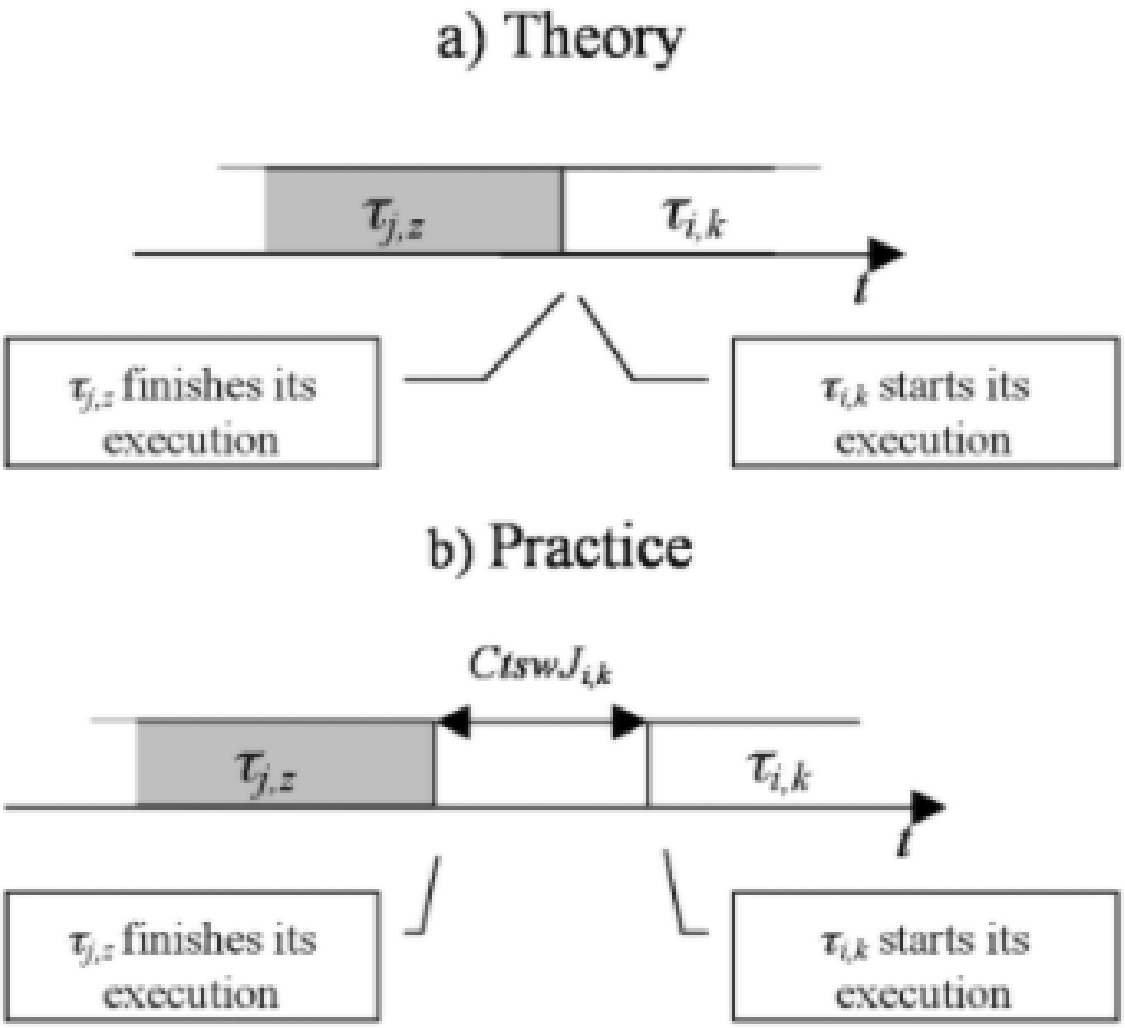
Process control information is used by the operating system to manage the process itself. That includes:

- Process scheduling state - The state of the process in terms of "ready", "suspended", etc., and also other scheduling information such as priority value, the amount of time elapsed since the process gained control of the CPU or since it was suspended. Also, in case of a suspended process, event identification data must be recorded for the event that the process is waiting for.
- Process structuring information - the child ids of the process, or the ids of other processes related to the current in some functional way, which can be represented as a queue, ring, or other data structures
- Inter-process communication information - flags, signals and messages associated with communication between independent processes
- Process privileges - allowed / not allowed access to system resources
- Process number (PID) - unique identification number for each process (also known as process ID)
- Program counter (PC) - A pointer to the address of the next instruction to be executed for this process
- CPU registers - set of registers where the process needs to be stored for execution to the execution state
- CPU Schedule Information - CPU Time Schedule Information
- Memory Management Information - Page Table, Memory Limits, Segment Table
- Accounting information - amount of CPU used for the execution process, time limits, execution ID, etc.
- I/O status information - list of I/O devices allocated to the process.

8.4. Perfomance considerations

Context switching involves costs that may affect the system's overall performance. These direct costs arise mainly from the fact that it takes time to save the context of a process that is running and then restore the context of another process that is about to run. During this time, there is no useful

work done by the CPU from the user's perspective. Therefore, context switching is sheer overhead in this condition.



8.4.1. Translation Lookaside Buffer

Translating a virtual address to a physical address is expensive. The processor must access the pages table structures, which usually have 3-4 levels. Thus, a single memory access actually requires 4-5 memory accesses.

To mitigate this issue, most modern processors cache virtual-to-physical translations in a translation lookaside buffer (TLB). The TLB is part of the MMU and can be understood as a cache for the MMU.

When virtual memory is updated - for instance, when one process's address space is replaced with another's during a software context switch - the TLB suddenly contains "stale" translations that are no longer valid. These translations must be flushed for correct behavior. This is less than ideal, as the next few operations must wait for the slow virtual-to-physical translations.

Recent Intel and AMD processors sport a tagged TLB, which allows you to tag a given translation with a certain address space configuration. In this scheme TLB entries never get "stale", and thus there is no need to flush the TLB.

8.4.2. Address Space Identifier

On ARM systems, this TLB tagging mechanism is implemented as follows: a value called address space identifier (ASID) is assigned by the OS to each task, so the MMU can distinguish between memory pages which share the same virtual address. For ARMv7 systems in particular, the ASID is an eight-bit value. For ARMv8, it can be 8 or 16 bits in length. The presence of the ASID in the TLB allows it to identify for each entry which Address Space it belongs to. When it comes to context switching, one of the necessary steps in the switching of task context is making sure the translation process using TLB, won’t translate to a physical address of another address space. One of the solutions is to use and update the current ASID value, identifying if an entry in cache should or not be used. In some system implementations, ASID values might be ignored altogether, in this case, the solution to this problem is to just invalidate the whole TLB cache, always resulting in page-faults and fetching the translation data from the correct process page table.

When using the short-descriptor translation table, the ASID value is stored in the CONTEXTIDR register. In case of the long-descriptor, TTBR0 register is a 64 bits register and it also stores the current ASID value.

8.4.2.1. ASID on Context Switch

Below we present two implementations for updating the process page table address and the ASID value.

Change Translation Table Base Register to the global-only mappings ISB Change ASID to new value ISB Change Translation Table Base Register to new value

In the first example, the address of the translation table (page table) is changed to a translation table that only global-pages could be accessed or translated, ensuring that no non-global pages can be fetched, because and it is uncertain if the old or new address space would be used for translating virtual addresses.

Change ASID to 0 ISB Change Translation Table Base Register ISB Change ASID to new value

In the second example, the ASID value is set to zero, which is a value normally not used for any operations and there should not exist any entries on TLB with such ASID. In this situation, we also ensure that translation would occur correctly, since translation will have to access the translation table.

8.4.2.2. Limitation of ASID

In some cases, the ASID value is represented by 8 bit, therefore, there can only be 256 different address spaces, since we only have 256 different identifiers. Because, most likely, processes do not share address spaces with each other, as a result, we are also bound to have up to 256 different tasks running at once in the system. The long-descriptor is a solution to this problem, because it offers not 8 but 16 bits for address space identifiers.

Linux uses a rollover mechanism for ASID, where once the ASID options run out, ASID values are invalidated from the branch predictor, caches and TLBs, and should be allocated again for each process, offering a chance for processes without an ASID (unable to run) to get one.

8.5. Switching Context in ARMv7/Raspberry Pi3

8.5.1. ARM Processor Mode

The ARM processor has many execution modes, this is important for task context switching because some of the indispensable register read and write requires it to be running on a privileged mode. Also, privileged modes offer banked registers that allow easier stack manipulation. A process running on user mode will have to enter a privileged mode by an interrupt before switching context. IRQ timer interrupt will bring the processor to IRQ mode, this is an example of an interrupt that can be used to achieve a privileged reschedule. Also, system mode has no banked register, this mode allows to update stack pointer registers, among others, for the next user process while in a privileged mode.

8.5.2. IRQ

IRQ handlers or interrupt request handlers is a hardware signal sent to the processor that temporarily stops a running program and allows a special program, an interrupt handler, to run instead hardware interrupts are used to handle events such as receiving data from a modem or network card, key presses, or mouse movements.

In the general case to enter a exception handler, we first must:

1. Save the address of the next instruction in the appropriate Link Register LR.
2. Copy CPSR to the SPSR of new mode.
3. Change the mode by modifying bits in CPSR.
4. Fetch next instruction from the vector table.

And to exit it:

1. Move the Link Register LR (minus an offset) to the PC.
2. Copy SPSR back to CPSR, this will automatically changes the mode back to the previous one.
3. Clear the interrupt disable flags (if they were set).

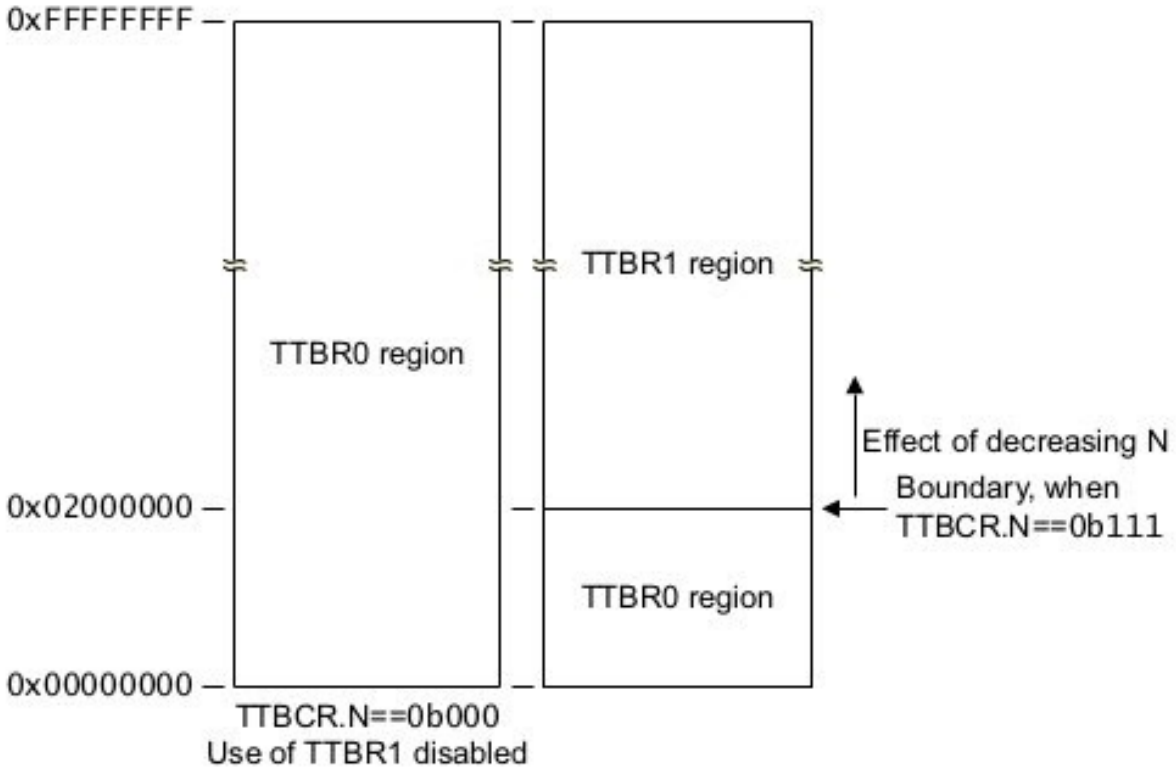
8.5.3. Managing Address Spaces

One of the necessary operations when switching to the current task context is managing the address spaces and references to the process page table. In the Armv7 architecture, we have a set of translation table support registers composed of TTBR0, TTBR1 and TTBCR.

The Translation Table Base Register 0 (TTBR0) holds information regarding the process page table base address and the memory it occupies. The Translation Table Base Register 1 (TTBR1) holds information regarding the system page table base address and the memory it occupies. Such division allows the entries translating virtual addresses allocated by the kernel (on the system page table) into physical addresses without duplicating these entries on multiple tasks page tables. Therefore, when it comes to context switching, it is only required to update the information contained on TTBR0.

The Translation Table Base Control Register (TTBCR) determines which of the Translation Table Base Registers, TTBR0 or TTBR1, should be used to translate a virtual address when it is not found on the TLB cache. The least significant two bits of the TTBCR represents an unsigned integer N, where, if the most significant N bits of the virtual address is zero, then the translation should occur on TTBR0, otherwise, translation should occur on TTBR1. Although, there is a special case, in which the value of N is zero, in this case, TTBR1 should be completely ignored and TTBR0 will be the only translation table used on the machine.

The TTBR0 register, bits 31:14 is used to store the base address of the translation table, and should be accessed by the MMU to translate virtual addresses.



In the presented image, the TTBCR.N (4 least significant bits of TTBCR) decides which of the translation tables is going to be used. Therefore, in the example on the left, where TTBCR.N is equal to 0x0, that means that every virtual address will useTTBR0. In the example on the right, TTBCR.N could be something similar to 0x1, so addresses in the format 0xdXXXXXXX will use TTBR0 as the translation table.

8.5.3.1. Acessing translation table support registers

It should first be noted that to access any of the translation table support register the ARM processor should be running on privileged mode at the moment of the access. The access of each register requires the use of the “MCR” and “MRC” assembly instructions. We show below the access of each register.

Assembly	Register
MRC p15, 0, <Rt>, c2, c0, 0	Read Translation Table Base Register 0
MCR p15, 0, <Rt>, c2, c0, 0	Write on Translation Table Base Register 0
MRC p15, 0, <Rt>, c2, c0, 1	Read Translation Table Base Register 1
MCR p15, 0, <Rt>, c2, c0, 1	Write on Translation Table Base Register 1
MRC p15, 0, <Rt>, c2, c0, 2	Read Translation Table Base Constro Register
MCR p15, 0, <Rt>, c2, c0, 2	Write on Translation Table Base Constro Register

8.6. Validation code

The validation code presented during the seminar can be accessed at:
https://github.com/gustavobiage/seminario_INE5424

8.7. References

https://en.wikipedia.org/wiki/Process_control_block
https://www.tutorialandexample.com/what-is-context-switching/
https://afteracademy.com/blog/what-is-context-switching-in-operating-system
https://wiki.osdev.org/Context_Switching
https://developer.arm.com/documentation/den0024/a/The-Memory-Management-Uni- /Context-switching
https://github.com/sokoide/rpi-baremetal
https://github.com/bztsrc/raspi3-tutorial
https://developer.arm.com/documentation/ddi0406/c/System-Level-Architecture/The-System-Level-Programmers--Model/ARM-processor-modes-and-ARM-core-registers/ARM-processor-modes?lang=en#CIHGHDGI

9. System Calls

9.1. Motivação

Atualmente, a maioria das arquiteturas de processadores utiliza diferentes níveis de privilégio para limitar o acesso a certos recursos. Esse controle é feito para, por exemplo, proteger as regiões críticas de memória ou evitar que usuários cometam erros que podem comprometer o funcionamento do dispositivo. Assim, os programas que rodam em níveis menores de privilégio são impedidos de executar certos conjuntos de instruções, como por exemplo, instruções de acesso a memória ou a dispositivos externos. Portanto, é comum serem criados diversos modos de execução para que tal controle ocorra. Normalmente, os processos de usuário se encaixam no Modo de Usuário, tendo esse o menor nível de prioridade. Porém, muitas vezes tais processos ainda necessitam utilizar recursos como acesso ao HD, criação de novos processos e comunicação com serviços do kernel. E, esses são impedidos de executar tais instruções por possuírem um PL baixo. Para resolver o problema em questão são utilizadas as chamadas de sistema (syscalls).

9.2. Definição de System Call

As chamadas de sistema (system calls) são mecanismos programáticos seguros e unificados para todos os processos, que oferecem acesso aos serviços de modos com PL mais alto. Em geral, as syscalls fornecem uma interface que gerencia a comunicação entre os PLs. Tal interface permite que os processos de baixo nível de prioridade passem as informações necessárias para a execução das instruções, mas que essas sejam efetivamente executadas no nível de operação do kernel, por exemplo.

9.3. Métodos de Implementação

A forma mais comum de se implementar uma syscall é através de software interrupt ou trap. Assim, o processo apenas carrega alguns registradores com o número da chamada de sistema e o handler da interrupção se encarrega de transferir o controle para o kernel. Porém, algumas outras arquiteturas da Intel utilizam as instruções SYSENTER/SYSEXIT que são um par de instruções específicas para realizar a troca entre os modos. Outro método específico usado da Intel é o call gate, ele utiliza um ponteiro que pode ser usado como uma chamada de função comum.

9.4. Funcionamento no ARMv7 e Cortex-A53

Como explicado anteriormente, os modos de execução do processo dita, entre outras coisas, o PL desse. No caso específico do ARMv7, os modos de execução são os apresentados abaixo:

Processor mode		Encoding	Privilege level	Implemented	Security state
User	usr	10000	PL0	Always	Both
FIQ	fiq	10001	PL1	Always	Both
IRQ	irq	10010	PL1	Always	Both
Supervisor	svc	10011	PL1	Always	Both
Monitor	mon	10110	PL1	With Security Extensions	Secure only
Abort	abt	10111	PL1	Always	Both
Hyp	hyp	11010	PL2	With Virtualization Extensions	Non-secure only
Undefined	und	11011	PL1	Always	Both
System	sys	11111	PL1	Always	Both

Fonte: ARM Architecture reference manual: ARMv7-A and ARMv7-R edition, pg B1-1139

User mode: Executa em PL0, o nível mais baixo de prioridade, também é chamado de execução não-privilegiada. Normalmente, as aplicações executam neste modo e possuem acesso restrito aos recursos do sistema. Execuções em User mode só podem alterar o modo através de uma exceção.
System mode: Executa em PL1 e não pode ser acessado por nenhuma exceção.
Supervisor mode: A instrução SVC (Supervisor Call) gera uma exceção Supervisor Call que é levado ao Supervisor mode. Este modo é o modo padrão para a recepção dessas exceções.
Hypervisor mode: Executa em PL2 e é acessado através das exceções Hypervisor Call e Hyp Trap.
Monitor mode: Executa em PL1 e é acessado através das exceções Secure Monitor Call.

Os modos hypervisor e monitor estão apenas disponíveis quando implementados com Extensões de Virtualização. Para o caso específico do projeto, os modos mais importantes tratam-se do modo User (onde rodam as aplicações) e o modo Supervisor (onde o SO é executado e possui acesso às instruções privilegiadas).

9.4.1. System Calls na arquitetura ARMv7

Nesta arquitetura, o modo do processador muda automaticamente quando recebe uma exceção. Quando é lançada uma exceção, são salvos o estado de execução atual e o endereço de retorno e, então, entra-se no modo solicitado. Caso necessário, é possível que ocorra a desabilitação de interrupções de hardware.

9.4.2. System Calls na família Cortex-A

Algumas instruções ou funções do sistema podem ser utilizadas somente em certos modos de execução. Se um código está rodando em um nível de menor privilégio e precisa de uma operação de um nível de maior privilégio, ele pode realizar uma requisição por meio de uma system call. Um jeito de fazer isso é por meio da instrução SVC. Isso permite que a aplicação gere uma exceção. Podem ser passados parâmetros por meio de registradores ou codificados dentro da system call.

Dessa forma, a instrução SVC pode ser usada para realizar requisições de aplicações de usuário em PL0 para o kernel no nível PL1. Também existem as instruções HVC e SMC para realizar mudanças no processador de forma similar para níveis de privilégio mais altos. Quando o processador está executando no nível PL0 (aplicação), ela não pode fazer uma requisição direta para o hypervisor (PL2). Portanto, as aplicações usam a instrução SVC para realizar requisições para o kernel e este se encarrega de requisitar as operações dos níveis superiores.

9.5. Como iniciar uma system call

9.5.1. Registradores

- **LR_<mode>** - Link Register

Armazena o endereço de retorno. Existe um registrador desse tipo para cada modo de operação.

- **CPSR** - Current Program Status Register

Indentifica o estado atual do processador.

- **SPSR_<mode>** - Saved Program Status Register

Usado para armazenar o CPSR do modo de execução atual ao trocar de modo. Existe um registrador desse tipo para cada modo de operação.

ARM processor modes and ARM core registers

Application level view	System level view								
	User	System	Hyp [1]	Supervisor	Abort	Undefined	Monitor [2]	IRQ	FIQ
R0	R0_usr								
R1	R1_usr								
R2	R2_usr								
R3	R3_usr								
R4	R4_usr								
R5	R5_usr								
R6	R6_usr								
R7	R7_usr								
R8	R8_usr								R8_fiq
R9	R9_usr								R9_fiq
R10	R10_usr								R10_fiq
R11	R11_usr								R11_fiq
R12	R12_usr								R12_fiq
SP	SP_usr		SP_hyp	SP_svc	SP_abt	SP_und	SP_mon	SP_irq	SP_fiq
LR	LR_usr			LR_svc	LR_abt	LR_und	LR_mon	LR_irq	LR_fiq
PC	PC								
APSR	CPSR								
			SPSR_hyp	SPSR_svc	SPSR_abt	SPSR_und	SPSR_mon	SPSR_irq	SPSR_fiq
			ELR_hyp						

‡ Part of the Security Extensions. Exists only in Secure state.
† Part of the Virtualization Extensions. Exists only in Non-secure state.
Cells with no entry indicate that the User mode register is used.

Fonte: ARM Architecture reference manual: ARMv7-A and ARMv7-R edition, pg 1144

9.5.2. Instruções

A syscall é realizada por meio de instruções que geram exceção.

Estas instruções irão copiar o CPSR para o SPSR do modo de operação atual e o endereço de retorno para o LR do modo atual.

O endereço preferencial de retorno dessas 3 instruções é o endereço da instrução seguinte.

9.5.2.1. SVC - Supervisor Call

Nas versões anteriores do ARM era chamada de SWI (Software Interrupt). É uma requisição de uma função do supervisor, faz com que o processador entre no modo Supervisor.

Com o HCR.TGE definido com 1, se o processador executar uma instrução SVC no modo usuário não-seguro, a exceção gerada leva ao modo Hyp.

Sintaxe assembler:

```
SVC{<cond>}{<q>} {#}<imm>
```

Exemplo:

```
MOV R0, #65 ; load R0 with the value 65 SVC 0x0 ; Call SVC 0x0 with parameter value in R0
```

No C/C++ pode ser feita a declaração de SVC como uma função __SVC:

```
__svc(0) void my_svc(int); . . . my_svc(65);
```

9.5.2.2. **HVC** - Hypervisor Call

Esta instrução serve para um Guest OS requisitar serviços do Hypervisor e está disponível se as extensões de virtualização estiverem implementadas.

9.5.2.3. **SMC** - Secure Monitor Call

Esta instrução permite que o Normal World requisite serviços do Secure World. Estando disponível se as extensões de segurança estiverem implementadas.

9.5.2.4. **SRS** - Store Return State

Armazena o LR e o SPSR do modo atual na pilha de um modo especificado.

9.5.3. Parâmetros

Por convenção, podem ser passados parâmetros para a system call por meio dos registradores R0-R3. Caso sejam necessários mais parâmetros, estes podem ser colocados na stack.

9.6. Identificação do tipo de exceção

A exceção gerada por essas instruções levará a um tratador cujo endereço é identificado na vector table. Nos manuais a entrada da vector table usada para system calls pode aparecer identificada como software_interrupt, pois a instrução SVC, antes do ARMv7, era SWI (Software Interrupt).

9.6.1. System calls aninhadas

No caso de system call aninhada, os valores de CPSR e o endereço de retorno são armazenados na pilha em vez do SPSR e do LR.

9.7. Identificação da system call

O identificador da system call é passado por meio de um valor imediato junto à instrução de entrada (SVC, HVC ou SMC). Este identificador é usado no tratador de system call para levar à system call requisitada.

9.8. Como retornar de uma system call

De acordo com o tipo da exceção é necessário ajustar o valor LR. A tabela a seguir mostra as instruções MOV e SUB sendo utilizadas como instruções de retorno. Ambas com o PC como o registrador de destino. O sufixo S nas instruções indica que o SPSR é copiado para o CPSR ao mesmo tempo.

Se o código de entrada do tratador de exceção usa a pilha para armazenar os registradores a serem preservados, o retorno pode ser feito usando uma instrução de load multiple com ^.

Exemplos:

```
LDM sp! {pc}^ LDMFD sp!, {R0-R12, pc}^
```

Ajustes para o Link Register

Exception	Adjustment	Return instruction	Instruction returned to
SVC	0	MOVS PC, R14	Next instruction
Undef	0	MOVS PC, R14	Next instruction
Prefetch Abort	-4	SUBS PC, R14, #4	Aborting instruction
Data abort	-8	SUBS PC, R14, #8	Aborting instruction if precise
FIQ	-4	SUBS PC, R14, #4	Next instruction
IRQ	-4	SUBS PC, R14, #4	Next instruction

Fonte: ARM Cortex-A Series v4 - Programmer’s Guide, pg 168

9.8.1. Instruções

9.8.1.1. **RFE** - Return From Exception

Carrega o PC e o CPSR retornando de uma exceção na qual o estado foi salvo com SRS. Se for utilizado ! o endereço final é escrito no registrador Rn.

```
RFE{addr_mode}{cond} Rn{!}
```

Exemplo:

```
RFE sp!
```

Valores de addr_mode:
IA - Increment address After (padrão, pode ser omitido)
IB - Increment address Before (apenas ARM)
DA - Decrement address After (apenas ARM)
DB - Decrement address Before

9.8.1.2. **ERET** - Exception Return

Retorna de uma exceção tratada no modo Hyp. Ela carrega o PC a partir do LR_hyp e o CPSR do SPSR_hyp. Esta instrução não deve ser usada nos modos User ou System.

9.8.2. Valor de retorno

Por convenção, os registradores R0 podem usados para retornar valores da system call.

9.9. Referências

ARM Architecture reference manual: ARMv7-A and ARMv7-R edition
ARM Cortex-A Series v4 - Programmer's Guide
RealView Compilation Tools Developer Guide
https://talk.dallasmakerspace.org/t/assembly-tutorial-syscalls-via-arm/24969
https://balau82.wordpress.com/2010/02/28/hello-world-for-bare-metal-arm-using-qemu/
https://wiki.osdev.org/Calling_Conventions

10. System Object Proxies and Agents

Em um contexto de chamadas de sistema, a partir de um certo nível do desenvolvimento de um microkernel torna-se necessário proteger o sistema das ações do usuário, implementando mecanismos de abstração, proteção e gerência de acesso aos recursos do sistema operacional, como as threads, mutexes, semáforos, criação de processos, dentre outros.

A forma mais fácil de acessar os recursos do sistema operacional é a partir de uma invocação direta das classes de sistema, manipulando-as como se fossem parte da aplicação do usuário. Mas para esse caso, apesar de oferecer total liberdade ao usuário para utilizar o sistema, há um problema claro que é a questão da segurança, não garantindo nenhuma forma de controle sobre o que está sendo feito no SO.

A solução para esse problema é proteger o kernel do usuário com a implementação de chamadas de sistema, comunicando-se com o kernel a partir de mensagens no modo supervisor. Mas para isso funcionar, é necessária a criação de um mecanismo de gerenciamento de chamadas de sistema, com a criação de uma interface pela qual o usuário poderá se comunicar com o kernel. É dentro desse contexto que o padrão Proxies/Agents/Stubs está inserido.

10.1. Gerenciamento de Syscalls

A imagem a seguir mostra a ideia geral da comunicação entre a aplicação e o kernel. Está subseção descreverá cada elemento dela.

Como já discutido na introdução, o usuário não pode chamar os métodos diretos do Kernel, logo o usuário chama um Stub que envia uma mensagem ao Kernel, trocando o sistema para o modo supervisor. Esta mensagem é enviada por meio de uma chamada de sistema, que gera uma interrupção (no armv8, pela instrução SVC), que então cai no método IC::software_interrupt() definido no ./machine/cortex/raspberry_pi3/raspberry_pi_3_ic.cc, o qual chama o Agente no modo Supervisor. Este agente recebe a mensagem, manda executar as funções necessárias no Kernel, e devolve o resultado ao usuário.

10.1.1. Stubs

São as interfaces pelas quais a aplicação utiliza para abstrair as chamadas de sistema. As stubs representam objetos de sistema e se comunicam com o kernel por meio de mensagens padronizadas que são enviadas através de syscalls. Cada stub possui como identificação um id definido pelo kernel. Este id é utilizado nas mensagens para identificar o objeto de sistema referido pelo stub. Existe uma classe stub para cada entidade. Em cada classe há um método para cada “method” de “Message” referente à respectiva entidade.

Observação: uma sugestão de implementação do mapeamento do id criado por requisições em uma Stub é apenas fazer uma re-interpretação do ponteiro do objeto para inteiro. Existem problemas de segurança ao utilizar-se desta medida, porém soluções melhores são bastante complexas, logo, utilizaremos esta solução sugerida na nossa sugestão de implementação.

10.1.2. Agents

Os agentes recebem a mensagem mandada pelo stub e trabalham no modo Supervisor. O objetivo do agente é traduzir a mensagem a uma ação que precisa ser executada pelo kernel, e então lidar com as chamadas de funções do sistema, garantindo que os parâmetros requisitados cheguem ao kernel, e o resultado volte ao usuário com a identificação correta dos objetos requisitados.

10.2. Sugestão de Implementação

Como o usuário deve chamar os stubs dentro da aplicação, é sugerido criar uma pasta “stubs” dentro do diretório ./include, que contenha todas as stubs respectivas a cada elemento do kernel desejável. A mensagem é implementada pela classe “Message”, que padroniza a comunicação entre as stubs e os agentes. E para os agentes, é criada uma classe “Agent” que consigue acessar os atributos de “Message” (a fim de manter o padrão), e então implemente funções que lidem com a chamada adequada de cada função equivalente do Kernel.

10.2.1. Implementação da Mensagem

O código abaixo mostra uma possível implementação de mensagem.

```
__BEGIN_SYS class Message { public: enum { DO_FORK, PRINT, DELETE, THREAD_CREATE, THREAD_JOIN, THREAD_EXIT, THREAD_WAIT_NEXT, MUTEX_CREATE, MUTEX_LOCK, MUTEX_UNLOCK, SEMAPHORE_CREATE, SEMAPHORE_P, SEMAPHORE_V, ... }; enum ENTITY { FORK, DISPLAY, THREAD, MUTEX, SEMAPHORE, ... }; public: template<typename ... Tn> Message(int id, int entity, int method, Tn ... an): _id(id), _entity(entity), _method(method) { set_params(an ...); } ... void act() { _syscall(this); } private: int _id; int _entity; int _method; int _result; char _params[256]; }; __END_SYS
```

A mensagem é uma classe de sistema que possui todas as mensagens possíveis entre o usuário e o kernel. Essa classe possui os seguintes atributos:

- id: É o identificador da mensagem.
- entity: É a identidade do kernel que a mensagem se refere.
- method: É a ação requisitada sobre a entidade.
- result: É o retorno da chamada.
- params: São os parâmetros do método.

Para identificar as mensagens, a classe utiliza duas enumerações que correspondem aos padrões que serão reconhecidos pelo kernel. A primeira enumeração é correspondente à entidade, que são as possíveis classes de syscall e que serão referenciadas pelo atributo entity, como Thread, Task, Mutex, dentre outras. Já a segunda enumeração é referente às próprias mensagens possíveis, como THREAD_CREATE, THREAD_JOIN, MUTEX_CREATE e demais, que serão referenciadas pelo atributo method.

A classe message também deverá ter algum método que chame uma chamada de sistema, a fim do sistema operacional conseguir entrar no modo supervisor e então executar o código do Kernel (já do lado do agente) de forma segura. Na nossa sugestão de implementação, este é o método `act`.

10.2.2. Implementação de Stubs

As stubs são implementadas individualmente em classes respectivas a cada entidade do kernel. Cada stub possui uma mensagem e um id, que será definido pelo agente após o envio da mensagem durante o construtor. A implementação a seguir mostra a stub referente a entidade Thread.

```
__BEGIN_API // Exemplo de Stub para as Threads class Stub_Thread { private: int id; // id único que representa o objeto no Kernel typedef _SYS::Message Message; // mensagem a ser enviada para o Agente public: Stub_Thread(){} // Exemplo: construtor template<typename ... Tn> Stub_Thread(int (* entry)(Tn ...), Tn ... an){ // Ao criar um novo objeto, passe o ID zero a mensagem // Passe para a mensagem a entidade, o que vai se fazer e os argumentos necessários Message * msg = new Message(0, Message::ENTITY::THREAD, Message::THREAD_CREATE, entry); // Envie a mensagem ao agente msg->act(); // Pegue o resultado id = msg->result(); } // Exemplo de stub para o método join int join() { // Como ja existe o stub, logo temos ID a se passar
```

// Novamente, passe para a mensagem a entidade e o que vai se fazer // Argumentos não são necessários para o join Message * msg = new Message(id, Message::ENTITY::THREAD, Message::THREAD_JOIN); msg->act(); // O resultado da função sempre esta em result, ele não é necessariamente o ID remetente a stub return msg->result(); } // Exemplo para o método pass void pass() { Message * msg = new Message(id, Message::ENTITY::THREAD, Message::THREAD_PASS); msg->act(); // Função void, resultado ignorado } } __END_API

O construtor de Stub_Thread é um bom exemplo para mostrar como é feita a criação de uma mensagem e a execução de uma stub. Primeiramente, é definida a mensagem, passando os seguintes argumentos:

- Id da mensagem: consiste no identificador único do objeto a ser acessado pelo Kernel. Quando este objeto não existe (ou seja, durante sua construção), o id 0 é enviado. Uma vez que a mensagem chegar ao Agente, este chamará as funções devidas no Kernel, o objeto será criado pelo Kernel, e então o Agente devolverá um identificador único, que poderá ser usado para eventuais modificações pelo usuário.
- Tipo da entidade: Para este caso, a própria Thread.
- A funcionalidade da entidade que deseja ser utilizada, por exemplo: criação da thread, join, yield, etc.
- Os argumentos do método (quando necessário). Para este exemplo da construção de uma thread, o argumento a ser passado é o entry point da thread.

A chamada de sistema é realizada pelo método act da mensagem e por fim, o id do objeto criado via stub é definido como o retorno da mensagem. Dessa forma, toda chamada de sistema feita para aquela thread é identificada pelo kernel a partir deste identificador.

10.2.3. Implementação dos Agentes

Como já comentado anteriormente, a classe Agent sugerida é uma Message (via herança). Ela também é implementada no kernel e gerenciará as mensagens recebidas por meio de syscalls. O código a seguir mostra uma possível implementação de agentes.

□□□□□□□

__BEGIN_SYS class Agent: public Message { public: // entry point (syscall) static void _exec(){ Agent * agt; ASM("mov %0, x0 " : "=r"(agt) :); agt->exec(); } // identificando a entidade correta void exec() { switch(entity()) { case Message::ENTITY::FORK: handle_fork(); break; case Message::ENTITY::DISPLAY: handle_display(); break; case Message::ENTITY::THREAD: handle_thread(); break; case Message::ENTITY::TASK: handle_task(); break; ... } // Identificando o método correto void handle_thread() { switch(method()) { case Message::THREAD_CREATE: { int (* entry)(); get_params(entry); Thread * t = new (SYSTEM) Thread(Thread::Configuration(Thread::READY, Thread::NORMAL), entry); result(reinterpret_cast<int>(t)); } break; case Message::THREAD_JOIN: { db<Agent>(TRC) << "THREAD JOIN" << endl; Thread * t = reinterpret_cast<Thread *>(id()); int r = t->join(); ASM("_banana:"); result(r); } break; ... } __END_SYS

Durante o processo de troca para o modo supervisor, a mensagem é preservada em um registrador conhecido (recomendamos o registrador x0). O entry point do agent é uma função que reinterpreta o conteúdo deste registrador para uma mensagem, e então chama sua função principal, que lida com a execução do conteúdo requisitado pela mensagem.

O método principal é chamado de exec, que possui um switch case que distribui as mensagens para os agentes respectivos de cada entidade, que também são métodos da classe Agent, entretanto, privados. A título de exemplo, o método handle_thread é responsável pela entidade thread, logo o primeiro switch case chama a função que lida com todos os métodos relacionados com threads. Dentri deste método, existe também uma estrutura de switch case que identifica qual é a ação que deve ser realizada para a respectiva entidade. Quando identificada, a ação é executada, e o valor de retorno é atualizado. Note que para os métodos que constroem objetos, o retorno é um identificador único deste objeto dentro do Kernel, o qual estamos atribuindo como ponteiro do objeto reinterpretado para inteiro (como discutido anteriormente).

10.3. Referências

- https://en.wikipedia.org/wiki/Stub_(distributed_computing)
- https://developer.arm.com/documentation/ddi0595/latest
- https://amp.freejournal.info/11253992/1/stub-distributed-computing.html

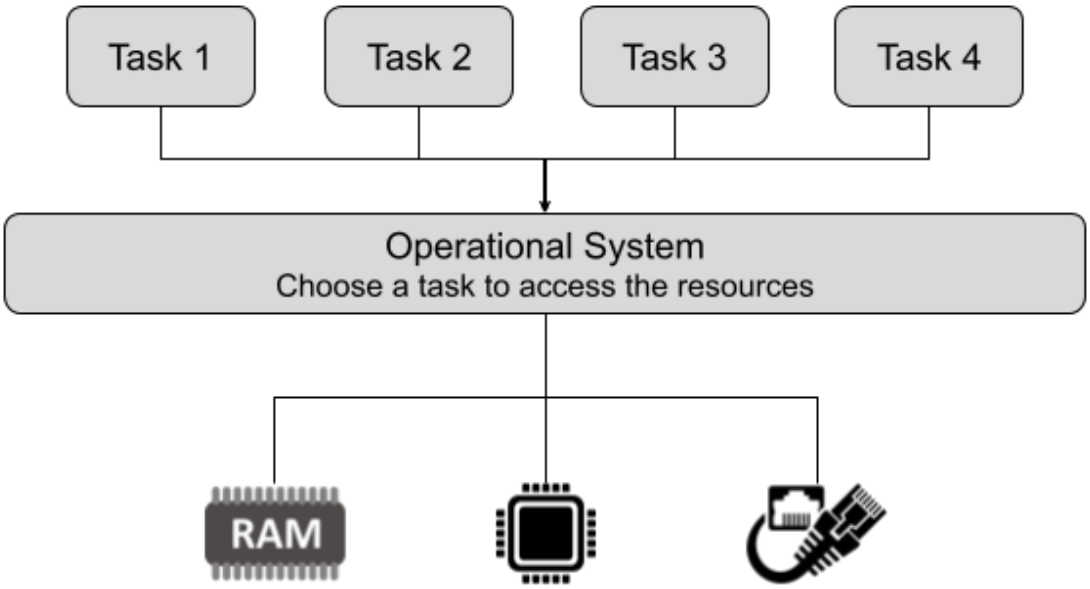
10.4. Autores

- Eduardo Willwock Lussi
- Mateus Favarin Costa
- Paulo Arthur Sens Coelho

11. Resource Management

11.1. Resource Management in Multitasking

A execução de um processo em uma máquina fica limitada pela quantidade de recursos físicos disponíveis. Há um número fixo de registradores na CPU, um certo número de computações por segundo, uma quantidade fixa de memória RAM, a qual pode ser consumida por um único programa em execução, ou múltiplos programas em execução sequencial.
Para solucionar esse problema, surgiram métodos que gerenciam tais recursos fixos e limitados com o intuito de executar múltiplos processos passando a impressão de estarem sendo executados ‘ao mesmo tempo’.
Em um sistema multitasking, cada um desses recursos é potencialmente disputado por vários processos e é de responsabilidade do SO definir o compartilhamento e contabilizar o uso de cada um deles.



11.1.1. Memory

Com múltiplas tarefas, o SO tem a responsabilidade de alocar e contabilizar memória para cada uma delas. Isso é suportado pelo uso da memória virtual e da paginação, que nos permitem fazer a troca de programas na memória física com mais facilidade, visto que podemos remover páginas da memória real enquanto que o programa continua vendo as páginas porém no disco, e só quando ele voltar a executar que elas serão carregadas no disco. Além de oferecer de acordo com a demanda da tarefa, o SO pode limitar quantos recursos uma única tarefa pode tomar. A proteção também é um aspecto relevante: o SO precisa garantir que as tarefas não possam interferir na memória uma das outras (a não ser no caso de memórias compartilhadas).

11.1.2. Processing Time

No multitasking preemptivo, o SO não espera que a tarefa rodando ceda a CPU. Em vez disso, uma troca de contexto ocorre a cada intervalo de tempo definido. Desse modo, as tarefas passam tempos parecidos sendo processados na CPU. Naturalmente, isso envolve um pouco mais de overhead para salvar as informações relevantes. Assim garantimos mais justiça de tempo de execução na CPU entre tarefas com diferentes números de Threads, pois com o tempo definido por Thread, tarefas com mais Threads passariam mais tempo na CPU do que com menos Threads.
Outro aspecto relevante é a possibilidade de prioridades diferentes, de modo que as tarefas mais importantes (como os que impactam a experiência do usuário ou tarefas de tempo real com um prazo, por exemplo) possam tomar a CPU de outras.

11.2. Desalocação de recursos

Ao finalizar uma tarefa em um sistema multitasking tem que ser feita a desalocação dos recursos utilizados, i.e. a memória, semáforos, mutexes, alarmes, e outros elemento que precisem de uma declaração implícita de destruição. Porém não podemos ter sempre certeza que a tarefa conseguiu finalizar como o esperado, é possível que algum erro fatal ocorra durante a sua execução, ou que o programador simplesmente não deletou corretamente os recursos criados pelo programa. Sendo assim, é necessário que o sistema operacional tenha a informação dos recursos que a tarefa havia criado para poder fazer a desalocação segura desses elementos.
Por essa razão as Tasks possuem listas ou tabelas, de referências à mutexes, semáforos e alarmes, e

cada vez que um elemento desse tipo é criado por uma tarefa, a Task responsável adiciona uma referência ao elemento a sua respectiva lista. Assim como toda vez que o elemento é propriamente deletado, a referência dele deve ser removida da lista na Task. Ao final da execução da tarefa, a Task deve propriamente deletar todos os mutexes, semáforos e alarmes presentes nas listas, visto que não foram devidamente finalizados.

11.3. EPOS Multitasking

No caso do Epos, devemos focar nas implementações que garantam uma desalocação segura dos recursos criados pelas tarefas que não foram devidamente deletados, como foi explicado no 10.2, utilizando listas de referências a mutexes, semáforos e alarmes na Task, e durante a criação desse elemento fazemos a adição da referência dele na Task, e na desalocação apropriada dele, removemos a referência dele na lista. No final da execução da tarefa fazemos a desalocação correta dos elementos restantes nas listas da Task.
Cada Task possui uma função 'main' a qual executa o código principal da aplicação, dessa forma, pode-se executar múltiplas Threads concorrentemente dentro do contexto da Task, utilizando recursos, i.e. memória e tempo de processamento, disponibilizados para a Task em questão.

No caso da divisão do tempo de processamento, onde cada Task recebe uma quantidade de tempo para ficar no processador, o escalonamento deve levar em conta a Task que esteja apta a executar, isso implica em 2 possíveis situações, uma em que é feito o escalonamento por Task, onde o uso de recursos de cada Task seria contabilizado por um algoritmo, como por exemplo, Round-robin, enquanto que as Threads pertencentes a Task são escalonadas por outro algoritmo.
No caso onde as Threads são escalonadas independentemente da Task, deve ser feita a alteração do algoritmo de escalonamento das Threads, levando em conta também a Task à qual a Thread pertence, de maneira a escolher Threads de Tasks disponíveis no momento, i.e. Tasks que não usaram seu tempo de processamento ainda.

11.4. Exemplo de Implementação

Como a implementação da Task inteira não era o foco, mas sim a implementação do gerenciamento dos recursos, inevitavelmente tivemos que criar a classe Task para ter as listas de mutexes, semáforos e alarmes, e no final da sua execução, liberar corretamente os recursos que não foram deletados antes.

11.4.1. Classe Task

A classe Task deve ser 'friend' das demais classes a fim de garantir a inserção e remoção dos recursos de suas respectivas listas.

```
// A Sample Task implementation class Task { friend class Thread; // To insert thread in queue of threads friend class Mutex; // To insert mutex in task friend class Semaphore; // To insert semaphore in task friend class Alarm; // To insert alarm in task protected: static const unsigned int STACK_SIZE = Traits<Application>::STACK_SIZE;
```

O construtor da Task irá receber o entry point da função principal e então criará uma Thread Main para a mesma. O atributo _current corresponde a Task que está executando atualmente. Ela pode ser utilizada no futuro para o escalonamento, mas nesse caso ela é utilizada pelas demais classes para saber a Task em execução.

```
public: template<typename ... Tn> Task(const Thread::Configuration & conf, int (* entry)(Tn ...), Tn ... an) { _main = new Thread(conf, entry, an ...); } ~Task(); Thread * main() { return _main; } static Task * volatile self() { return current(); }
```

Em seguida, criamos as funções de inserção e remoção de acordo com o tipo do elemento. Além disso, há a criação de funções públicas que acessam as listas apenas para serem utilizadas no teste.

```
private: // Creating what is the current task static Task * volatile current() { return _current; } static void current(Task * t) { _current = t; } // Add or remove task's thread void insert(Thread * t) { _threads.insert(new (SYSTEM) Thread::Queue::Element(t)); } void remove(Thread * t) { Thread::Queue::Element * el = _threads.remove(t); if(el) delete el; } void insert(Mutex * m) { _mutexs.insert(new (SYSTEM) Queue<Mutex>::Element(m)); } void remove(Mutex * m) { Queue<Mutex>::Element * el = _mutexs.remove(m); if(el) delete el; } void insert(Semaphore * s) { _semaphores.insert(new (SYSTEM) Queue<Semaphore>::Element(s)); } void remove(Semaphore * s) { Queue<Semaphore>::Element * el = _semaphores.remove(s); if(el) delete el; } void insert(Alarm * a) { _alarms.insert(new (SYSTEM) Queue<Alarm>::Element(a)); } void remove(Alarm * a) { Queue<Alarm>::Element * el = _alarms.remove(a); if(el) delete el; } private: // Managing Threads Thread * _main; static Task * volatile _current; Thread::Queue _threads; Queue<Mutex> _mutexs; Queue<Semaphore> _semaphores; Queue<Alarm> _alarms; public: Thread::Queue threads() { return _threads; } Queue<Mutex> mutexs() { return _mutexs; } Queue<Semaphore> semaphores() { return _semaphores; } Queue<Alarm> alarms() { return _alarms; } };
```

11.4.2. Ponteiro para Task nos Elementos

Para que os recursos saibam a qual Task elas pertencem, foi criado um ponteiro para a mesma.

```
class Synchronizer_Common { .... Task * _task; };
```

```
class Thread { .... Task * task; };
```

```
class Alarm { .... Task * _task; };
```

11.4.3. Adição dos Elementos nas listas da Task

Quando um elemento é criado, ele se adiciona na lista de Task corrente. Vale ressaltar que para essa implementação, estamos desconsiderando as Threads Main e Idle como pertencentes a lista de threads da Task. Isso foi feito para conseguir compatibilidade com a versão atual do EPOS.

```
template<typename ... Tn> inline Thread::Thread(const Configuration & conf, int (* entry)(Tn ...), Tn ... an) :_task(conf.task ? conf.task : Task::self()), _state(conf.state), _waiting(0), _joining(0), _link(this, conf.criterion) { constructor_prologue(conf.stack_size); _context = CPU::init_stack(0, _stack + conf.stack_size, &__exit, entry, an ...); constructor_epilogue(entry, conf.stack_size); // Not add Idle in task's threads list if (conf.criterion != Thread::IDLE) { _task->insert(this); } }
```

```
Mutex::Mutex(): _locked(false) { db<Synchronizer>(TRC) << "Mutex() => " << this << endl;
_task = Task::self(); _task->insert(this); }
```

```
Mutex::~Mutex() { db<Synchronizer>(TRC) << "~Mutex(this=" << this << ")" << endl;
_task->remove(this); }
```

A adição da task no Semaphore e no Alarm é análoga à mostrada para o Mutex.

11.4.4. Criação da Task

Para criar a Task, deve ser passado o valor “true” do traits “multitask_test”. Um auxiliar é criado para fazer com que a Idle seja criada após a Main, caso seja interessante que a mesma se encontre na lista de threads da Task.

```
void Thread::init() { .... static volatile bool task_ready = false; if(Traits<System>::multitask_test)
{ db<Init, Thread>(TRC) << "----------TASK----------" << endl; new (SYSTEM) Task(main);
task_ready = true; } else { new (SYSTEM) Thread(Thread::Configuration(Thread::READY,
Thread::MAIN), main); } if(Traits<System>::multitask_test) while (!task_ready); // Idle thread
creation does not cause rescheduling (see Thread::constructor_epilogue) new (SYSTEM)
Thread(Thread::Configuration(Thread::READY, Thread::IDLE), &Thread::idle); .... }
```

11.4.5. Destructor da Task

Quando uma Task é destruída, ela antes deleta todos os recursos que possuem e que não foram deslocados ainda .

```
// Class attributes Task * volatile Task::_current; Task::~Task() { db<Task>(TRC) <<
"========Caling Task's destructor========" << endl; // Remove Mutex in Task
while(!_mutexs.empty()) { db<Task>(TRC) << "----->Removing Mutex from Task" << endl; delete
_mutexs.remove()->object(); } // Remove Semaphore in Task while(!_semaphores.empty()) {
db<Task>(TRC) << "----->Removing Semaphore from Task" << endl; delete
_semaphores.remove()->object(); } img // Remove Alarms in Task while(!_alarms.empty()){
db<Task>(TRC) << "Removing Alarm from Task" << endl; delete _alarms.remove()->object(); }
// Remove threads in Task while(!_threads.empty()) { db<Task>(TRC) << "----->Removing Thread
from Task" << endl; delete _threads.remove()->object(); } }
```

11.5. Resource Management Autores

- Paulo Barbato Fogaça de Almeida
- Robson Zagre Júnior
- Wesly Carmesini Ataide

11.6. Resource Management Referências

- https://isaaccomputerscience.org/concepts/sys_os_resource_management
- https://ibcomputerscience.xyz/resource-management/
- https://computersciencewiki.org/index.php/Operating_Systems_management_techniques
- https://science.jrank.org/computer-science/Multitasking_Operating_Systems.html
- https://people.cs.ksu.edu//~schmidt/300s05/Lectures/OSNotes/os.html

12. Inter-Process Communication

12.1. Motivação

IPC (inter-process communication) é o mecanismo que permite a troca de mensagens entre processos cooperativos, e por definição são os processos que compartilham dados com outros processos. O IPC é importante pois possibilita: compartilhamento de informações, aumento da velocidade de computação e modularidade. Referente a modularidade, no projeto de sistemas operacionais a comunicação entre processos é o que torna possível o microkernel.
Existem dois modelos fundamentais de comunicação entre processos: memória compartilhada (shared memory) e troca de mensagens (message passing).

12.2. Modelos de IPC

12.2.1. Memória Compartilhada

No modelo de memória compartilhada, uma região de memória compartilhada é criada em um processo, e os demais anexam essa região compartilhada ao seu próprio espaço de endereçamento. Com isso, podem ler e escrever nesta região. Para garantir que os processos acessem a memória compartilhada sem condição de corrida, é necessário usar semáforo ou mutex, já que este meio de troca de mensagens é full-duplex, ou seja, ambas as partes podem escrever quanto ler.

A região de memória compartilhada é composta de um buffer, sendo que ele pode ou não ter um tamanho limite.

- Buffer Ilimitado: Um buffer ilimitado é exatamente o que o nome sugere, não possui um limite de tamanho para o espaço de memória compartilhada, podendo crescer indefinidamente.
- Buffer Limitado: Um buffer limitado possui um tamanho máximo, e com isso é necessário verificar o estado do buffer antes de inserir um novo dado, caso ele esteja cheio, é necessário aguardar alguem consumir um dado, para então adicionar a nova informação.

Como curiosidade, é possível verificar as regiões de memória compartilhada de sistemas Linux utilizando o comando **ipcs**.

12.2.2. Troca de Mensagens

No modelo de troca de mensagens a comunicação é intermediada pelo kernel, que faz a ponte entre os processos que estão querendo compartilhar informações. A API de troca de mensagens implementada pelo kernel por ser feita de dois jeitos principais, sendo considerado mensagens de tamanho fixo, ou tamanho variável.
Implementar a API com mensagens de tamanho fixo, irá facilitar a sua implementação, mas irá dificultar a sua utilização, como por exemplo nos casos em que as mensagens precisam ser maiores do que o tamanho máximo. Por outro lado, uma API com mensagens de tamanho variável se tornam mais complexas, mas sua utilização se torna mais simples.
Há dois principais modos de troca de mensagens, sendo elas: comunicação direta e comunicação indireta.

12.2.2.1. Comunicação Direta

A comunicação direta se dá pelo nome dos processos. Sendo assim, se tivermos dois processos A e B, e o processo A deseja enviar uma mensagem para o processo B, teremos algo como:

```
// process A write(B, message) // process B read(A, message)
```

Esse tipo de comunicação pode se tornar problemática, visto que é necessário saber o nome do processo que irá receber a mensagem, assim como o nome do processo que está enviando-a. Caso ocorra uma mudança no nome de um dos processos, seria necessário atualizar todos os processos que estão se comunicando com ele. Resolvendo este problema, temos a comunicação indireta.

12.2.2.2. Comunicação Indireta

A comunicação indireta se dá por mailboxes. A mailbox é um buffer com um identificador, em que os processos podem escrever ou ler mensagens. Assim como no exemplo anterior, se tivermos dois processos A e B, sendo A querendo enviar uma mensagem para B, teremos:

```
// process A write(M, message) // mailbox M // process B read(M, message)
```

A mailbox pode ser criada pelo sistema operacional ou por um processo. Caso seja do processo, se o processo for finalizado, é importante ressaltar que a mailbox irá desaparecer junto com ele.
Diferente da comunicação direta, podemos ter um problema de concorrência para ler as mensagens da mailbox, já que mais de um processo pode estar acessando o seu conteúdo.
Este problema pode ser resolvido de algumas maneiras:

- Bloquear a utilização das mailboxes por mais de dois processos.
- Permitir que apenas o primeiro processo que pediu a mensagem consuma-a
- Utilizar um escalonador para fazer os processos alternarem quem está consumindo as mensagens.

12.2.2.3. Comunicação Síncrona ou Assíncrona

Independente do tipo de comunicação, sendo ela direta ou indireta, temos dois meios de lidar em como os processos vão agir na hora de enviar a mensagem e quando irão recebê-la. Temos como enviar mensagens de forma síncrona (*blocking sender*) ou assíncrona (*nonblocking sender*), do mesmo jeito que temos a opção de receber mensagens de forma síncrona (*blocking receiver*) quanto assíncrona (*nonblocking receiver*).

- blocking sender: o processo é bloqueado quando uma mensagem é enviada até que o processo destinatário receba a mensagem;
- nonblocking sender: o processo envia a mensagem e não é aguarda nenhum tipo de confirmação se a mensagem foi recebida;
- blocking receiver: o processo destinatário bloqueia até que a mensagem esteja disponível;
- nonblocking receiver: o processo destinatário tenta ler uma mensagem, caso ela não exista, recebe um NULL.

12.2.2.4. Buffers

Toda a comunicação é feita utilizando buffers, sendo que eles podem ser de três tipos:

- Capacidade Zero: em que o processo sender aguarda o receiver receber a mensagem;
- Capacidade Limitada: em que o processo sender envia mensagens de forma assíncrona enquanto tiver espaço no buffer. Caso o buffer fique cheio, o processo aguarda ter um espaço no buffer antes de enviar a próxima mensagem,
- Capacidade Ilimitada: o processo sender sempre envia mensagens de forma assíncrona.

12.2.2.5. Pipes

Um exemplo de message passing, é o pipe. O pipe é uma forma de comunicação entre dois processos de forma unidirecional, e ele vem de duas formas distintas: unnamed e named.

12.2.2.5.1. Unnamed pipes

O unnamed pipe pode ser usado no terminal de forma simples, quando executamos um comando e querendo passar o output deste comando para o próximo. Um exemplo da sua utilização num terminal do sistema Linux:

```
$ ps aux | grep <user>
```

12.2.2.5.2. Named pipes

Named pipes também são conhecidos como filas FIFO. Named pipes são criados e podem durar por um longo período. Nos sistemas tipo Unix ele utiliza o filesystem, assim para utilizá-lo é necessário criar um arquivo com o comando **mkfifo**.

```
mkfifo pipe // cria uma FIFO chamada pipe cat file > pipe // coloca o output do cat dentro do pipe cat pipe // le o conteudo do pipe, pode ser feito em qualquer terminal rm pipe // utiliza o filesystem para deletar o pipe
```

12.2.2.6. Sockets

O socket é outro exemplo de implementação de troca de mensagens, sendo mais voltado para comunicação cliente/servidor. A troca de mensagens pode ser no mesmo sistema operacional, ou pela rede. Um socket é um IP + porta, então o sistema operacional associa um socket a um processo cliente e outro socket ao processo servidor, e os conecta, possibilitando a troca de informações.

12.3. Exemplo de implementação

12.3.1. Memória compartilhada

Um exemplo de memória compartilhada é a implementaçao do UP_RTOS 1, um SO de tempo real. Quando o SO é iniciado, é criado um array de structs para estar disponível para uso de memória compartilhada.

```
#define NPID NPROC/sizeof(int) struct shmem { int procID[NPID]; Mutex mutex; char * address; } shmem[32];
```

As structs são inicializadas sem processos utilizando-as, o mutex aberto e apontando para uma região vazia de 64kB de memória.

```
procId = { 0 }; mutex = UNLOCKED; address = emptyRegion
```

Para utilizar uma região de memória compartilhada, o processo atacha ele mesmo à struct de memória compartilhada utilizando o método **shmem_attach**, que por sua vez irá registrar o id do processo. Com isso, o processo pode escrever ou ler a região compartilhada com os métodos

shmem_read e **shmem_write**.

```
int shmem_attach(struct shmem *mp); shmem_read( struct shmem *mp, char buf[], int nbytes); shmem_write(struct shmem *mp, char buf[], int nbytes);
```

12.3.2. Troca de mensagens

Assume-se que o kernel tem um conjunto finito de buffers de mensagem, que são definidos como:

```
typedef struct mbuf{ struct mbuf *next; // pointer to next mbuf int pid; // sender pid int priority; // message priority char contents[128]; // message contents }MBUF; MBUF mbuf[NMBUF]; // NMBUF = number of mbufs
```

Inicialmente, todos os buffers de mensagens estão em uma mbufList livre. Para enviar uma mensagem, um processo deve primeiro obter um mbuf livre. Depois
recebendo uma mensagem, ele libera o mbuf para reutilização. Uma vez que o mbufList é acessado por muitos processos, é uma região crítica que deve ser protegida. Portanto, define-se um semáforo mlock = 1 para os processos acessarem exclusivamente a mbufList. O algoritmo de get_mbuf () e put_mbuf () são:

```
MBUF *get_mbuf() { P(mlock); MBUF *mp = dequeue(mbuflList); // return first mbuf pointer V(mlock); return mp; } int put_mbuf(MBUF *mp) { P(mlock); enqueue(mbufList) V(mlock); }
```

12.3.2.1. Troca de mensagem assíncrono

No esquema de troca de mensagem assíncrona, as operações de envio e recebimento são não bloqueantes. Se um processo não puder enviar ou receber uma mensagem, ele retornará um status de falha e, nesse caso, o processo pode repetir a operação novamente mais tarde. A comunicação assíncrona destina-se principalmente a sistemas fracamente acoplados, nos quais a comunicação entre processos é pouco frequente, ou seja, os processos não trocam mensagens de forma planejada ou regular. Para tais sistemas, a passagem assíncrona de mensagens é mais adequada devido à sua maior flexibilidade.

```
int a_send(char *msg, int pid) // send msg to target pid { MBUF *mp; // validate target pid, e.g. proc[pid] must be a valid processs if (!(mp = get_mbuf())) // try to get a free mbuf return -1; // return -1 if no mbuf mp->pid = running->pid; // running proc is the sender mp->priority = 1; // assume SAME priority for all messages copy(mp->contents, msg); // copy msg to mbuf // deliver mbuf to target proc's message queue P(proc[pid].mlock); // enter CR // enter mp into PROC[pid].mqueue by priority V(proc[pid].lock); // exit CR V(proc[pid].message); // V the target proc's messeage semaphore return 1; // return 1 for SUCCESS } int a_recv(char *msg) // receive a msg from proc's own mqueue { MBUF *mp; P(running->mlock); // enter CR if (running->mqueue==0){ // check proc's mqueue V(running->mlock); // release CR lock return -1; } mp = dequeue(running->mqueue); // remove first mbuf from mqueue V(running->mlock); // release mlock copy(msg, mp->contents); // copy contents to msg int sender=mp->pid; // sender ID put_mbuf(mp); // release mbuf as free return sender; }
```

Esse algoritmo funciona em condições normais. No entanto, se todos os processos apenas enviarem,

mas nunca receberem, ou se um processo malicioso enviar mensagens repetidamente, o sistema pode ficar sem buffers de mensagens livres. Quando isso acontecesse, o recurso de mensagem seria interrompido, pois nenhum processo pode enviar mais. Por outro lado, não pode haver nenhum deadlock porque ele é não-bloqueante.

12.3.2.2. Troca de mensagem síncrona

Para suportar a passagem síncrona de mensagens, foram definidos semáforos adicionais para sincronização de processos.

```
SEMAPHORE nmbuf = NMBUF; // number of free mbufs SEMAPHORE PROC.nmsg = 0; // for proc to wait for messages MBUF *get_mbuf() // return a free mbuf pointer { P(nmbuf); // wait for free mbuf P(mlock); MBUF *mp = dequeue(mbufList) V(mlock); return mp; } int put_mbuf(MBUF *mp) // free a used mbuf to freembuflist { P(mlock); enqueue(mbufList, mp); V(mlock); V(nmbuf); } int s_send(char *msg, int pid)// synchronous send msg to target pid { // validate target pid, e.g. proc[pid] must be a valid processs MBUF *mp = get_mbuf(); // BLOCKing: return mp must be valid mp->pid = running->pid; // running proc is the sender copy(mp->contents, msg); // copy msg from sender space to mbuf // deliver msg to target proc's mqueue P(proc[pid].mlock); // enter CR enqueue(proc[pid].mqueue, mp); V(proc[pid].lock); // exit CR V(proc[pid].nmsg); // V the target proc's nmsg semaphore } int s_recv(char *msg) // synchronous receive from proc's own mqueue { P(running->nmsg); // wait for message P(running->mlock); // lock PROC.mqueue MBUF *mp = dequeue(running->mqueue); // get a message V(running->mlock); // release mlock copy(mp->contents, msg); // copy contents to Umode put_mbuf(mp); // free mbuf }
```

Sempre que um protocolo de bloqueio é usado, há chances de deadlock. Esse algoritmo pode levar às seguintes situações de impasse.

1. Se os processos apenas enviam, mas não recebem, todos os processos eventualmente serão bloqueados em P (nmbuf) quando não houver mais mbufs livres.

2. Se nenhum processo enviar, mas todos tentarem receber, todos os processos serão bloqueados em seu próprio semáforo nmsg.

3. Um processo Pi envia uma mensagem para outro processo Pj e aguarda uma resposta de Pj, que faz exatamente o contrário. Então Pi e Pj esperariam mutuamente um pelo outro, que é o conhecido deadlock travado.

12.4. Referências

K.C. Wang. Embedded and Real-Time Operating Systems. 1 Ed. 2017.

Silberschatz, Peter Galvin and Greg Gagne. Operating System Concepts, 10th Ed. 2019.

Interprocess Communication Mechanisms. Disponível em: <https://tldp.org/LDP/tlk/ipc/ipc.html> Acessado em: Julho de 2021.

13. I/O

13.1. Memory Mapped Peripherals

Para controlar as operações de Entrada e Saída a ARM utiliza o conceito de "Memory Mapped

Peripherals". Este conceito significa que qualquer periférico terá seus registradores mapeados para memória da máquina, possuindo seus conteúdos lidos e escritos utilizando as mesmas condições de outras regiões da memória, inclusive as mesmas instruções "load" e "store". Para informar ao processador erros ou alteração no conteúdo da memória, o periférico utiliza interrupções normais (IRQ) e rápidas (FIQ) dependendo de sua programação, que serão tratadas e depois gerenciadas em software.

13.2. Registradores de dispositivo

Um dispositivo pode ter os seguintes tipos de registradores mapeados para memória:

- Transmit Data Register (Somente Escrita): Envia dados para memória
- Receive Data Register (Somente Leitura): Recebe os dados para o dispositivo
- Control Register (Escrita e Leitura): Ajusta os sinais recebidos pelo periférico
- Interrupt Enable Register(Escrita e Leitura): Controla quais eventos de hardware geram interrupções
- Status Register (Somente Leitura): Controla a disponibilidade dos dados necessários para leitura e escrita

13.3. Direct Memory Access(DMA)

Algumas implementações utilizam o hardware denominado "Direct Memory Acess (DMA)" para tratar transferências sem utilizar o processador, utilizando buffers para transmitir os dados. DMAs são utilizadas quando os dados a serem transferidos são muito grandes, pois para não sobrecarregarem o processador elas geram interrupções menos frequentemente, somente em exceções ou na inicialização/término da transferência.

13.4. Memory Mapped Regions

Para cada região da memória alguns atributos são definidos, entre eles:

- Shareability: Indica para o sistema a chance de a região ser acessada por processadores diferentes.
- Cacheability: Indica para o sistema se alocar um valor para a cache pode melhorar a performance do sistema.
- Transient: Indica que o benefício de usar a cache é por um período de tempo relativamente curto, portanto podendo ser melhor restringir a alocação da cache.
- Execute-never: Indica se o processador pode executar instruções lidas desta região de memória, ou qual o nível de privilégio necessário para que elas possam ser executadas.

Para uma região de memória um tipo também é atribuído, sendo que ele pode ser do tipo Normal, Dispositivo ou Fortemente-Ordenado.
Regiões do tipo Normal podem ser regiões com leitura/escrita ou somente escrita, sendo que os atributos são sensíveis à região em que eles estão. Esta região também apresenta as seguintes propriedades:

- Leituras podem ser repetidas sem efeitos colaterais;
- Leituras repetidas retornam o último valor escrito;
- Leituras podem pegar regiões de memória adicionais sem efeitos colaterais;
- Acesso desalinhado a memória pode ser suportado;
- Acessos podem ser mesclados antes terminar o acesso ao sistema de memória final;
- Escritas consecutivas podem ser repetidas sem efeitos colaterais se o conteúdo da região não foi alterado entre as escritas ou são resultados de uma exceção.

Regiões do tipo Dispositivo e Fortemente-Ordenado são regiões onde um acesso pode causar efeitos colaterais, ou leituras consecutivas podem retornar valores diferentes, a depender do número de leituras realizadas. Elas nunca são mantidas na cache, apresentam o atributo Shared ativado e as seguintes propriedades:

- O número, ordem e tamanho dos acessos não pode ser mudado, sendo especificado pelo programa;
- Leituras e escritas não podem ser repetidas;
- Todos acessos ocorrem no tamanho do programa.

A principal diferença entre estes dois tipos de região está no fato de que uma escrita para regiões Fortemente-Ordenado só pode ser completada quando atinge o componente ou dispositivo periférico acessado pela escrita, enquanto que em regiões do tipo Dispositivo ela pode ser concluída antes.

13.5. Níveis de Privilégio

A arquitetura ARMv7 define diferentes níveis de privilégio de execução:

- Secure state: níveis PL1 e PL0.
- Non-secure state: níveis PL2, PL1 e PL0.

O modo de processador atual determina o nível de privilégio de execução e, portanto, o nível de privilégio de execução pode ser descrito como o nível de privilégio do processador. Os níveis tem as seguintes características:

- PL0 : O nível de privilégio do software aplicativo, que é executado no modo de usuário. Portanto, software executado no modo de usuário é descrito como software sem privilégios. Este software não pode acessar algumas características da arquitetura. Em particular, ele não pode alterar muitas das configurações.O software em execução no PL0 faz apenas acessos à memória sem privilégios.

- PL1: A execução do software em todos os modos, exceto o modo Usuário e o modo Hyp, está em PL1. Normalmente, o software do sistema é executado em PL1. O software em execução no PL1 pode acessar todos os recursos da arquitetura, e pode alterar as configurações para esses recursos, exceto para alguns recursos adicionados pelas extensões de virtualização que só são acessíveis no PL2. O software em execução no PL1 faz acessos privilegiados à memória por padrão, mas também pode fazer acessos não privilegiados.

- PL2: O software em execução no modo Hyp é executado no PL2. O software em execução no PL2 pode realizar todas as operações acessíveis no PL1 e pode acessar algumas funcionalidades adicionais. O modo Hyp é normalmente usado por um hipervisor, que controla e pode alternar entre sistemas operacionais convidados, que executam em PL1.

13.6. Controle de nível de privilégio

As permissões de acesso à memória atribuídas em PL1 podem definir que uma região de memória é:

- Não acessível a quaisquer acessos;
- Acessível apenas para acessos em PL1;
- Acessível para acessos em qualquer nível de privilégio.

No estado não seguro, permissões de acesso à memória separadas podem ser atribuídas no PL2 para:

- Acessos feitos no PL1 e PL0.;
- Acessos feitos no PL2.

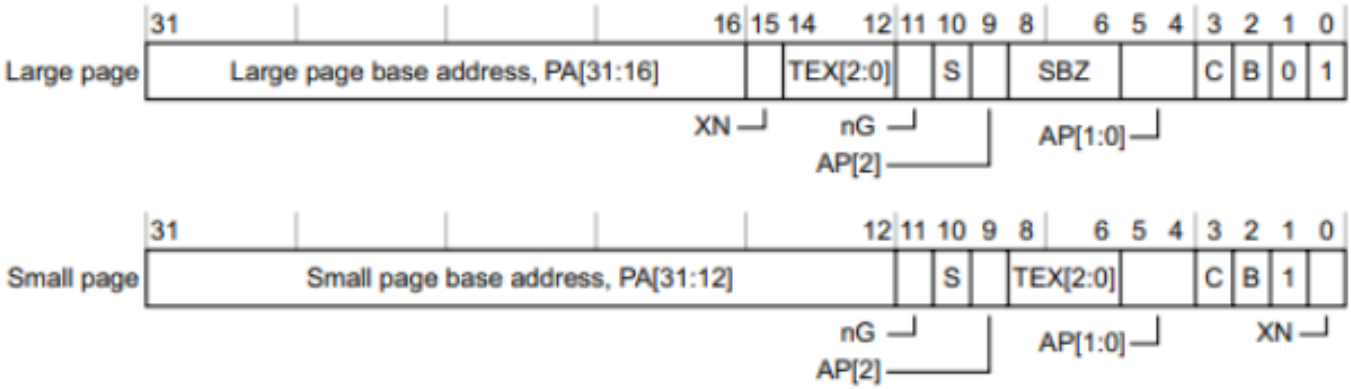
Um acesso privilegiado à memória é um acesso feito durante a execução em PL1 ou superior, como resultado de uma instrução load/store diferente de LDRT, STRT, LDRBT, STRBT, LDRHT, STRHT, LDRSHT e LDRSBT.
Um acesso à memória sem privilégios é um acesso feito como resultado de uma operação de leitura ou escrita (load/store) realizada em um destes casos:

- Quando o processador está em PL0.
- Quando o processador está em PL1, e o acesso é feito como resultado de uma instrução LDRT, STRT, LDRBT, STRBT, LDRHT, STRHT, LDRSHT ou LDRSBT.

Um Data Abort é gerado se o processador tenta um acesso a dados que os direitos de acesso não permitem. Por exemplo, um Data Abort é gerado se o processador estiver em PL0 e tentar acessar uma região de memória que está marcada como acessível apenas para acessos de memória privilegiados.

13.7. Bits para controle de privilégio de uma região

Além de um endereço de saída, uma entrada da tabela de tradução que se refere à página ou região da memória inclui campos que definem propriedades da região de memória de destino. Os campos de controle de acesso, que serão descritos a seguir, determinam se o processador, em seu estado atual, tem permissão para realizar o acesso necessário ao endereço de saída fornecido no descritor da tabela de tradução.
Os bits que controlam as permissões de acesso a uma memória correspondente estão em um descritor de tabela de tradução e são três, denominados de AP2:0. Para descritores curtos de segundo nível, sua localização está de acordo com a figura abaixo.



Estes bits podem ser utilizados para definir as permissões de acesso de duas maneiras:

- Utilizando três bits, AP 2: 0.
- Utilizando dois bits, AP 2: 1 e utilizando AP0 como flag de acesso.

O bit 29 de system control register, SCTLR.AFE, define a maneira que será utilizada para controlar o acesso. Para descritores curtos, setar este bit para 1 seleciona o uso de AP 2: 1 , enquanto este bit em 0 seleciona o uso de AP2:0. Para descritores longos, sempre será utilizado AP2:1, não importando o valor do bit. Para setar o valor de SCTLR, é necessário estar em modo de privilégio de execução mínimo de PL1, realizando leitura/escrita no registrador p15 com opcode igual a 0.

13.7.1. Modelo de permissões de acesso AP 2: 1

Neste modelo:

- Um bit, AP 2, seleciona entre acesso somente leitura e acesso de leitura / escrita.
- Um segundo bit, AP 1, seleciona entre o controle de nível de aplicativo (PL0) e nível de sistema (PL1).

Na arquitetura ARM, este modelo permite quatro combinações de acesso, descritas pela tabela:

AP[2], disable write access	AP[1], enable unprivileged access	Access
0	0[a]	Read/write, only at PL1
0	1	Read/write, at any privilege level
1	0[a]	Read-only, only at PL1
1	1	Read-only, at any privilege level

a. Not valid for Non-secure PL2 stage 1 translation tables. AP[1] is SBO in these tables.

13.7.2. Modelo de permissões de acesso AP 2: 0

Quando este modelo de permissões é utilizado, se o bit AP0 for igual a 1, o modelo se torna igual a AP2:1. As permissões válidas para o modelo são descritas pela seguinte tabela:

AP[2]	AP[1:0]	PL1 access	Unprivileged access	Description
0	00	No access	No access	All accesses generate Permission faults
	01	Read/write	No access	Access only at PL1
	10	Read/write	Read-only	Writes at PL0 generate Permission faults
	11	Read/write	Read/write	Full access
1	00	-	-	Reserved
	01	Read-only	No access	Read-only, only at PL1
	10	Read-only	Read-only	Read-only at any privilege level, deprecated[a]
	11	Read-only	Read-only	Read-only at any privilege level[b]

a. From VMSAv7, ARM strongly recommends use of the 0b11 encoding for Read-only at any privilege level.
b. This mapping is introduced in VMSAv7, and is reserved in VMSAv6.

13.8. Single-channel DMA transfer

Exemplo de um Interrupt handler em uma I/O orientado por interrupção para transferências de memória. O código é FIQ Handler. Ele usa banked FIQ registers para manter o estado entre as interrupções.Este código está localizado em 0x1C.
Toda a sequência para lidar com uma transferência normal é de quatro instruções. O código situado após o retorno condicional é usado para sinalizar que a transferência foi concluída.

Assembly	Descrição
□□□□□□□ LDR R11, [R8, #IOData]	Load port data from the IO device.

□□□□□□□□ STR R11, [R9], #4	Store it to memory: update the pointer.
□□□□□□□□ CMP R9, R10	Reached the end ?
□□□□□□□□ SUBLSS pc, lr, #4	No, so return.
□□□□□□□□	Insert transfer complete code here.

R8: Aponta para o endereço base do dispositivo de E / S do qual os dados são lidos.
IOData: É o deslocamento do endereço base para o registro de dados de 32 bits que é lido. Ler este registro limpa a interrupção.
R9: Aponta para o local da memória para onde os dados estão sendo transferidos.
R10: Aponta para o último endereço para o qual transferir.
As transferências de bytes podem ser feitas substituindo as instruções de load por load byte. As transferências da memória para um dispositivo de E / S são feitas trocando os modos de endereçamento entre a instrução de load e a instrução de store.

13.9. Dual-channel DMA transfer

Exemplo de um Interrupt handler em uma I/O orientado por interrupção para transferências de memória só que com dual channel. O código é FIQ Handler. Ele usa banked FIQ registers para manter o estado entre as interrupções.Este código está localizado em 0x1C.
Toda a sequência para lidar com uma transferência normal é de nove instruções. O código situado após o retorno condicional é usado para sinalizar que a transferência foi concluída.

Assembly	Descrição
□□□□□□□□ LDR sp, [R8, #IOStat]	Load status register to find which port caused the interrupt.
□□□□□□□□ TST sp, #IOPort1Active	Store it to memory: update the pointer.
□□□□□□□□ LDREQ sp, [R8, #IOPort1]	Load port 1 data.
□□□□□□□□ LDRNE sp, [R8, #IOPort2]	Load port 2 data.
□□□□□□□□ STREQ sp, [R9], #4	Store to buffer 1.
□□□□□□□□ STRNE sp, [R10], #4	Store to buffer 2.
□□□□□□□□ CMP R9, R11	Reached the end?
□□□□□□□□ CMPLE R10, R12	On either channel?
□□□□□□□□ SUBSNE pc, lr, #4	Return
□□□□□□□□	Insert transfer complete code here.

R8: Aponta para o endereço base do dispositivo de E / S a partir do qual os dados são lidos.
IOStat: É o deslocamento do endereço de base para um register que indica qual das duas portas

causou a interrupção.
IOPort1Active: É uma máscara de bits que indica se a primeira porta causou a interrupção. Caso contrário, presume-se que a segunda porta causou a interrupção.
IOPort1, IOPort2: São deslocamentos para os dois registers de dados a serem lidos. Ler um registro de dados limpa a interrupção para a porta correspondente.
R9: Aponta para o local da memória para o qual os dados da primeira porta estão sendo transferidos.
R10: Aponta para o local da memória para o qual os dados da segunda porta estão sendo transferidos.
R11, R12: Aponta para o último endereço para o qual transferir. R11 para a primeira porta, R12 para a segunda.

13.10. Referências

Steve Furber. ARM System-on-Chip Architecture. 2 Ed. Reino Unido, 2000.
ARM. ARM Architecture Reference Manual: ARMv7-A and ARMv7-R edition. Disponível em <https://developer.arm.com/documentation/ddi0406/latest/>. Acessado em Julho, 2021.
ARM. Arm Cortex-A53 MPCore Processor Technical Reference Manual. Disponível em <https://developer.arm.com/documentation/ddi0500/j/>. Acessado em Julho, 2021.
ARM Compiler Software Development Guide. Disponível em <https://developer.arm.com/documentation/dui0471/m> . Acessado em Julho, 2021.