EPOS for Raspberry Pi 3

Software/Hardware Integration Lab at UFSC

EPOS for Raspberry Pi

Table of contents

1. EPOS on Raspberry Pi - ARMv7 32 bits

1.1. Running EPOS on Raspberry Pi

1.1.1. 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 = ARMv8; // 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>
```

1.1.2. Running and Debugging

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

1.1.3. 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.

1.1.3.1. 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.

1.1.3.1.1. 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.

1.1.3.1.2. 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.

1.1.3.2. 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

1.2. MMU for Paging

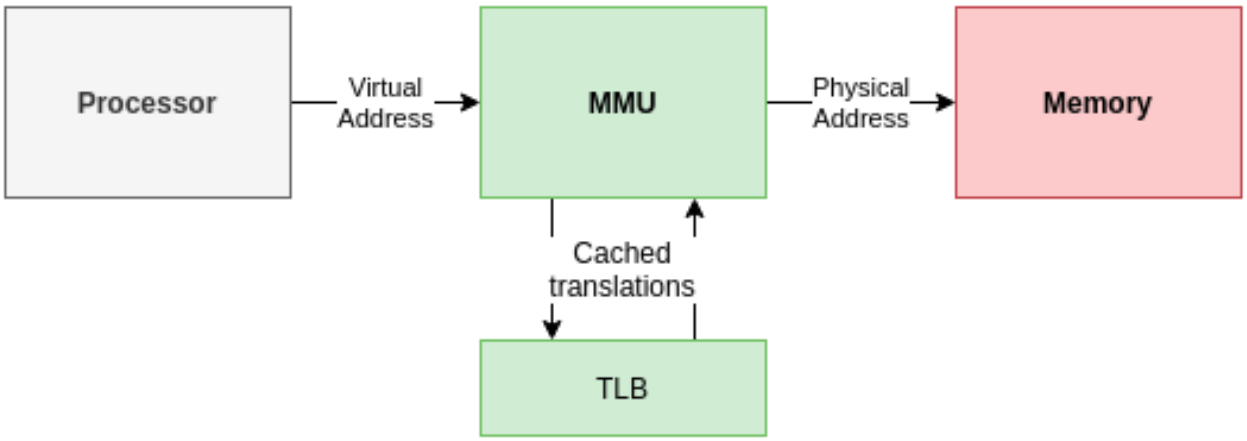
1.2.1. Conceitos importantes

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.

1.2.2. 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.



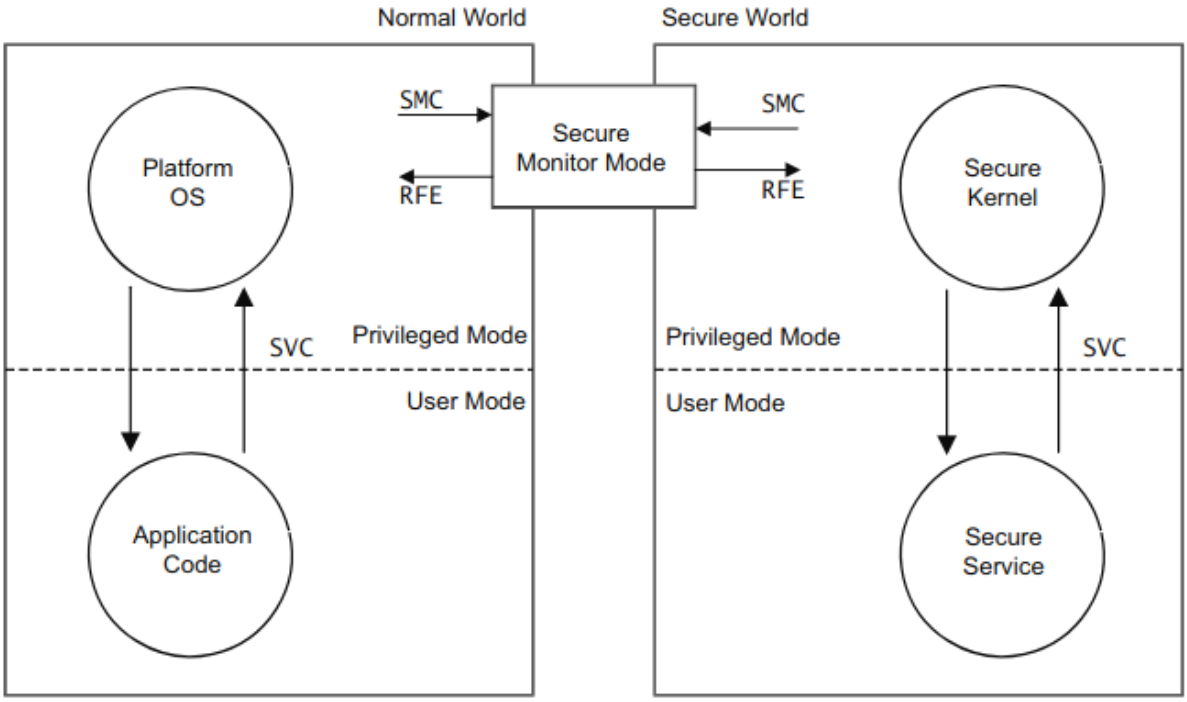
1.2.3. Paging

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.

1.2.4. 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.

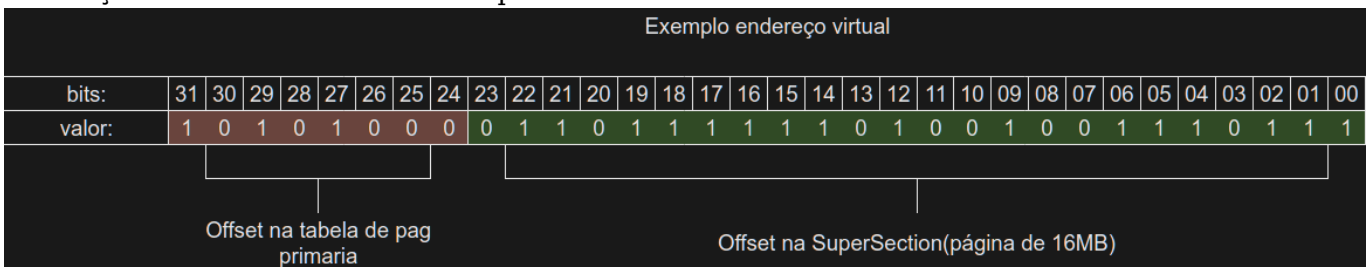
1.2.5. Paging - ARMv7

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.

1.2.5.1. Super Section

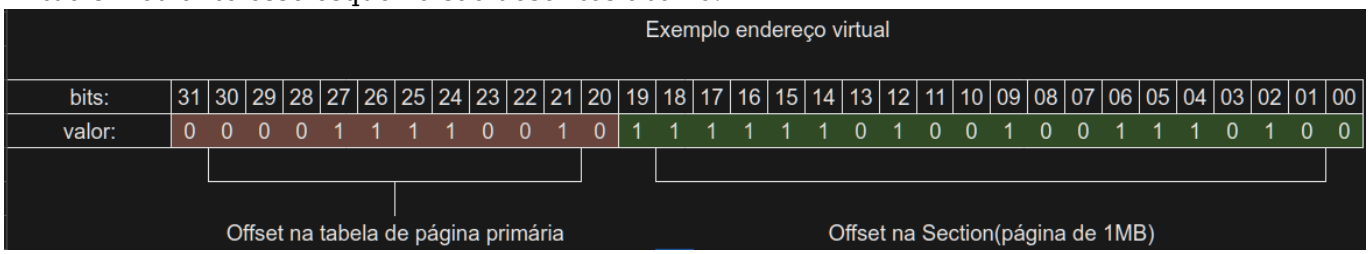
Esquema de endereçamento virtual de 1 nível. Ou seja, ao adotar esse modelo os processos

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.



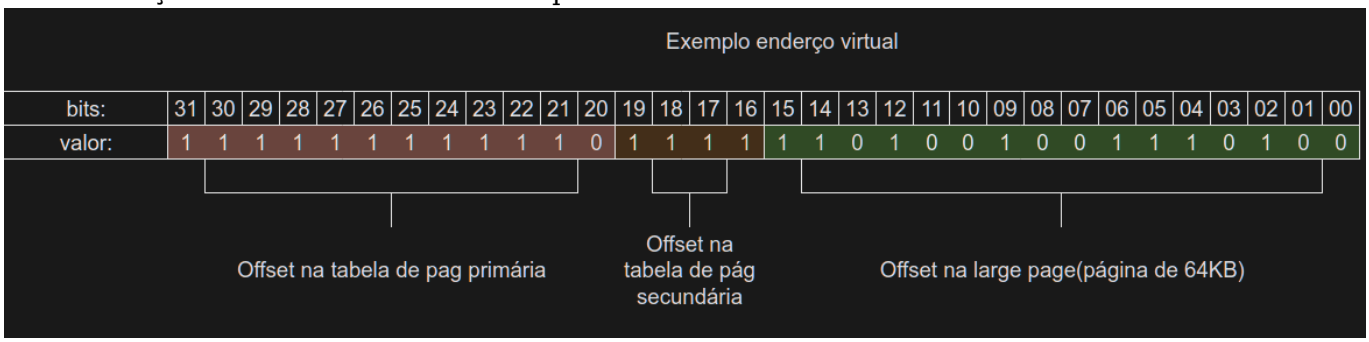
1.2.5.2. Section

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.



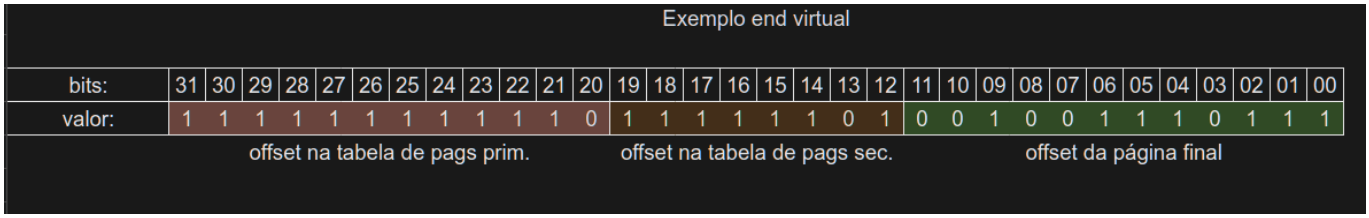
1.2.5.3. Large Page

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.



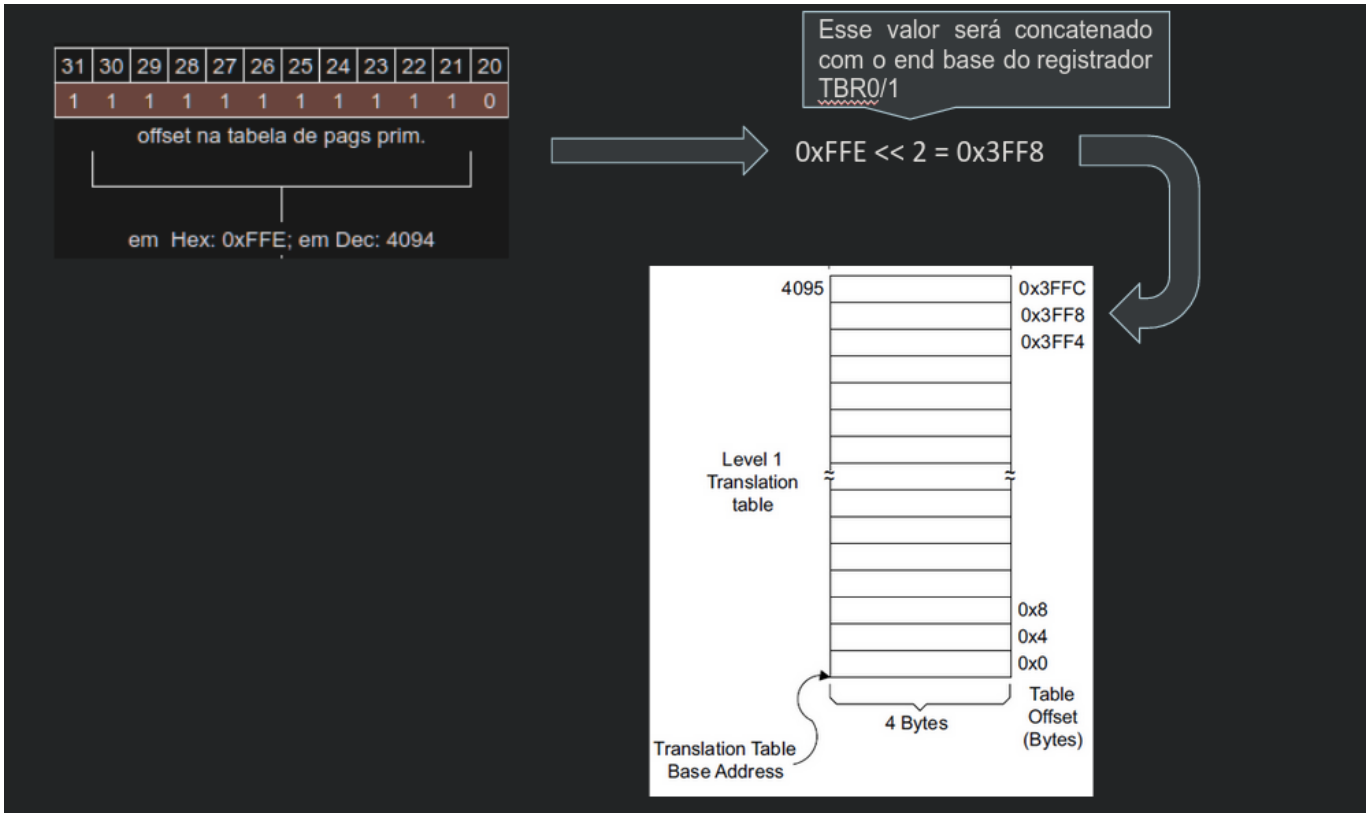
1.2.5.4. Small Page

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.

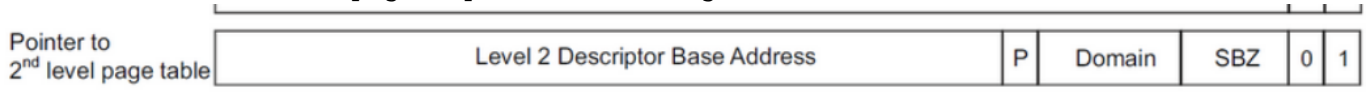


1.2.6. Exemplo de tradução de endereço no ARMv7

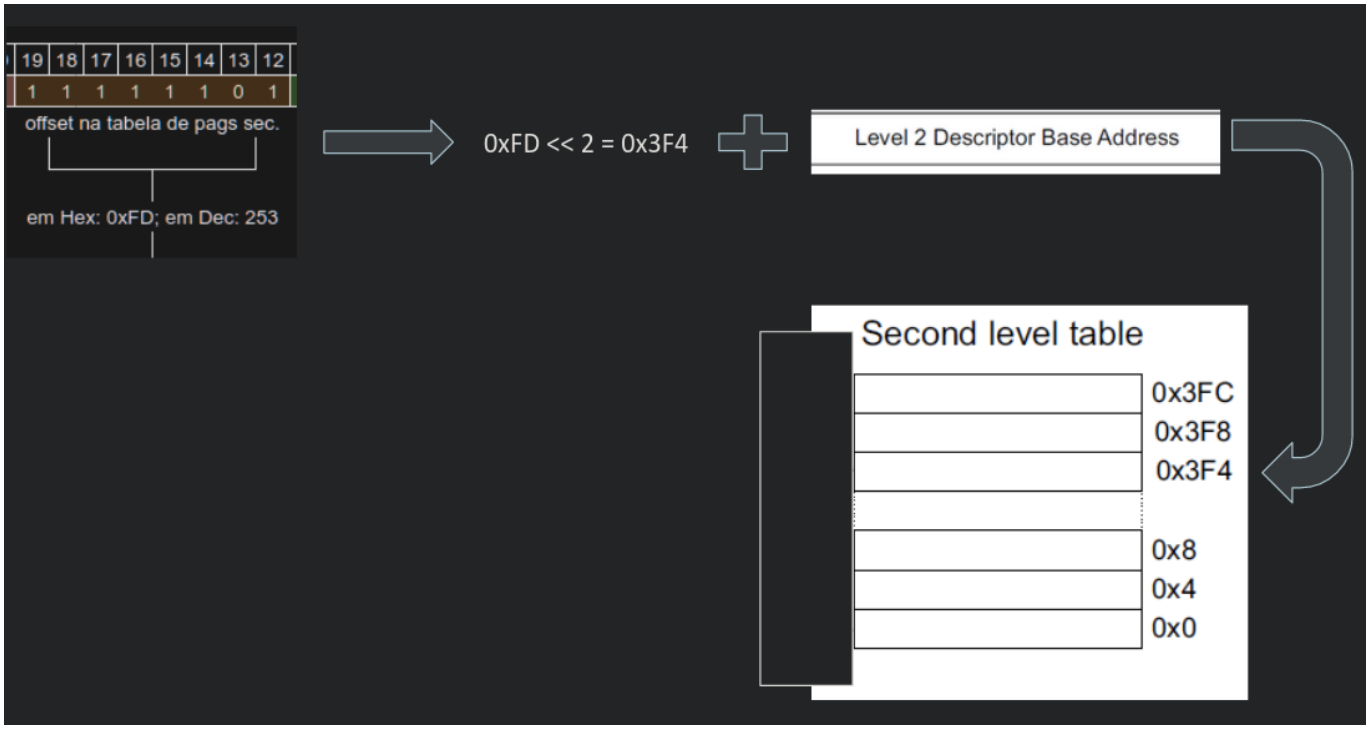
Supondo o uso de páginas de 4KB(small pages) e N = 1 e que a consulta a TLB não obteve sucesso, a tradução de endereço se dá por:
1. É selecionado o endereço base da tabela de diretório a ser consultado usando os bits 31-14 do TTBR0, ou TTBR1 se o endereço for igual ou maior do que 0x8000000.
2. Os bits 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:

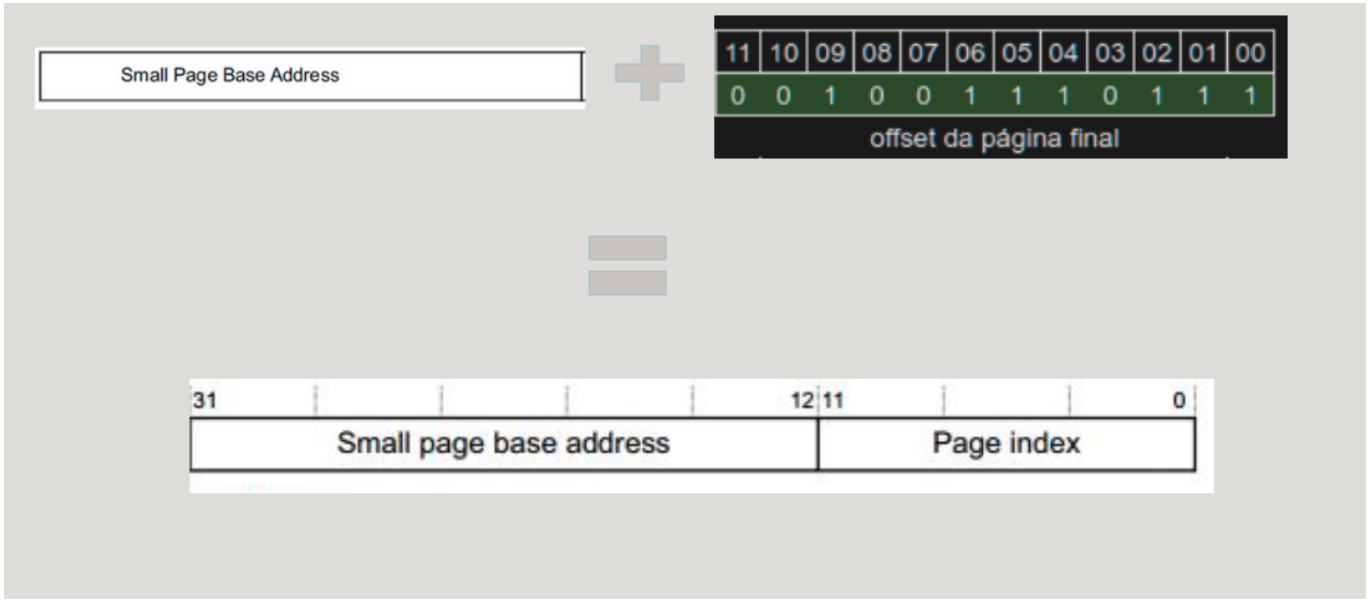


4. Os bits 31-10 do valor no endereço formado são concatenados aos bits 19-12 do endereço de entrada; o resultado é descolado 2 bits para a esquerda;



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.



1.2.7. 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");

1.2.8. 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

1.3. Switching Context in ARMv7/Raspberry Pi3

1.3.1. ARM Processor Mode

The ARM processor has many execution modes, this is important for task context switching because some of the indispensable registers 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, the 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.

1.3.2. IRQ

IRQ or interrupt request 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).

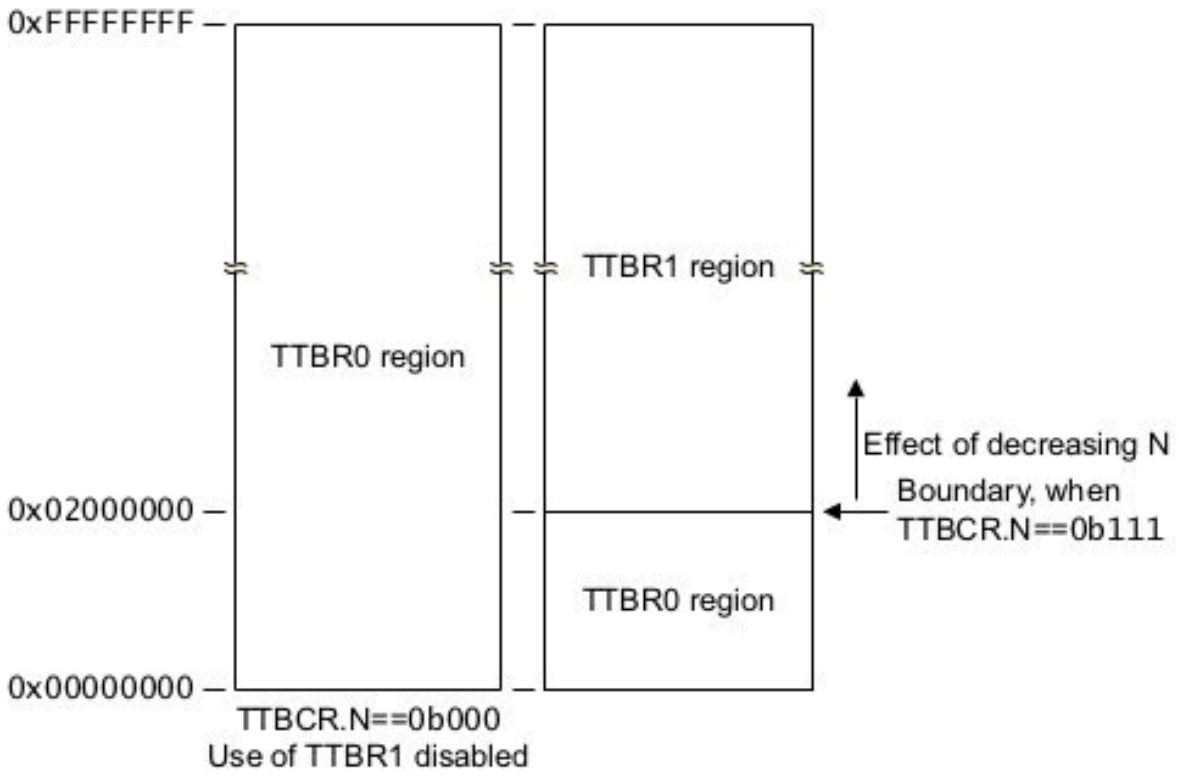
1.3.3. Managing Address Spaces

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.

1.3.3.1.1. 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

1.3.4. Validation code

The validation code presented during the seminar can be accessed at:

https://github.com/gustavobiage/seminario_INE5424

1.3.5. 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

1.4. System Calls - 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).

1.4.1. 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.

1.4.2. 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.

1.4.3. Como iniciar uma system call

1.4.3.1. Registradores

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

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

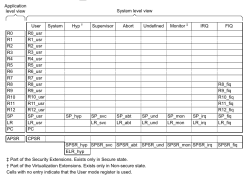
- **CPSR** - Current Program Status Register

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							

1 Part of the Security Extensions. Exists only in Secure state.
2 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

1.4.4. 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.

1.4.4.1. 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);
```

1.4.4.2. **HVC** - Hypervisor Call

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

1.4.4.3. **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.

1.4.4.4. **SRS** - Store Return State

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

1.4.5. 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.

1.4.6. 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).

1.4.7. 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.

1.4.8. 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.

1.4.9. 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

1.4.10. Instruções

1.4.10.1. **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

1.4.10.2. **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.

1.4.10.3. Valor de retorno

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

1.4.11. Referências

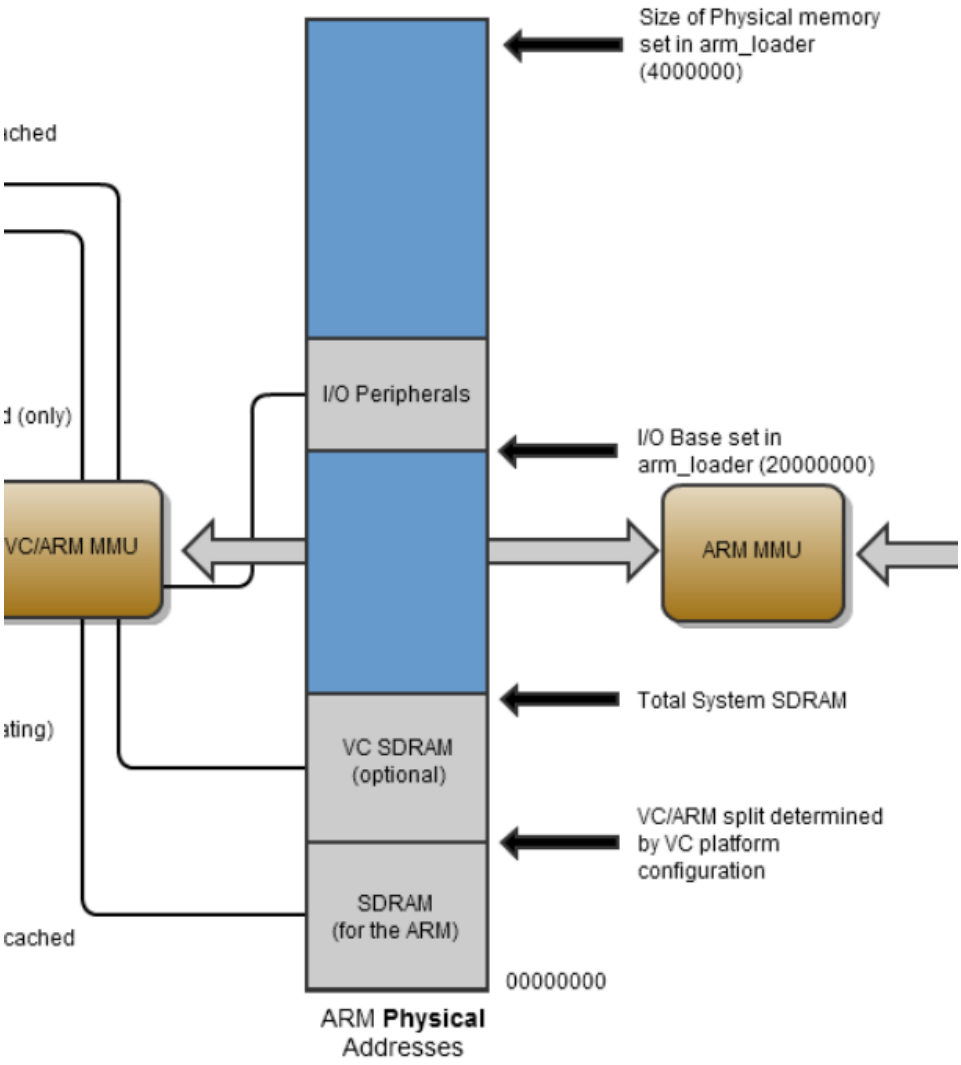
ARM Architecture reference manual: ARMv7-A and ARMv7-R edition
ARM Cortex-A Series v4 - Programmer's Guide
RealView Compilation Tools Developer Guide
https://talk.dallasmakerspace.org/t/assembly-tutorial-syscalls-via-arm/24969
https://balau82.wordpress.com/2010/02/28/hello-world-for-bare-metal-arm-using-qemu/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

2. EPOS on Raspberry Pi - ARMv8 64 bits

2.1. Task Memory Model

2.1.1. O Modelo de Memória Físico

O modelo físico de memória é o que temos mais próximo do hardware. Nesse mapeamento de endereços, temos acesso a todos os dispositivos conectados ao SoC, como visto na figura abaixo.

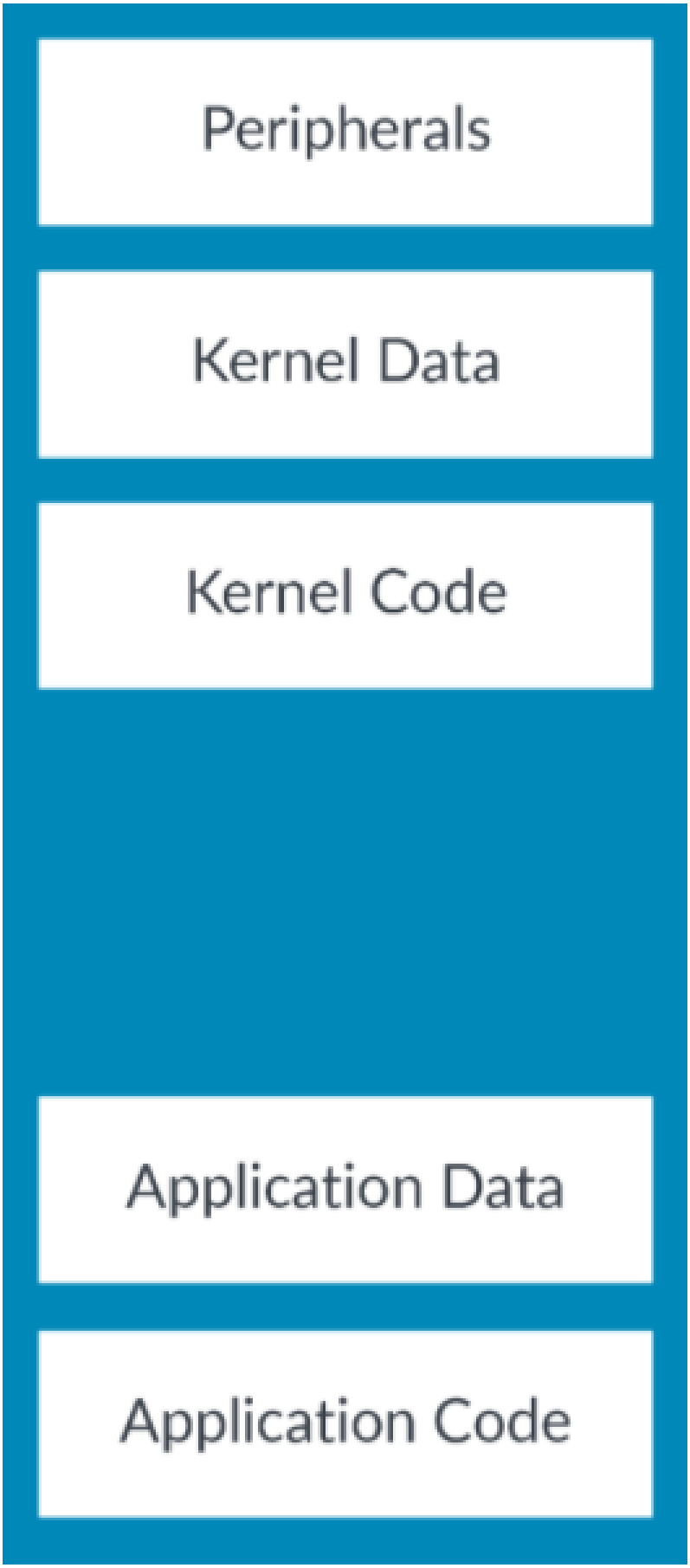


Tais endereços são definidos pelo fabricante e variam de modelo para modelo. Vale ressaltar que nesse modo também não temos o controle sobre a estrutura e acesso da memória, trazendo diversos problemas de segurança e compatibilidade, visto que programas esperam uma memória "ideal". Para isso precisamos abtrair o hardware em um modelo de memória.

2.1.2. O que é um modelo de memória?

Um modelo de mémória nada mais é que um planejamento de como a memória deve ser utilizada. Ele contém não só a descrição da estrutura mas também o comportamento a ser adotado para acesso e uso de endereços e regiões de memória.

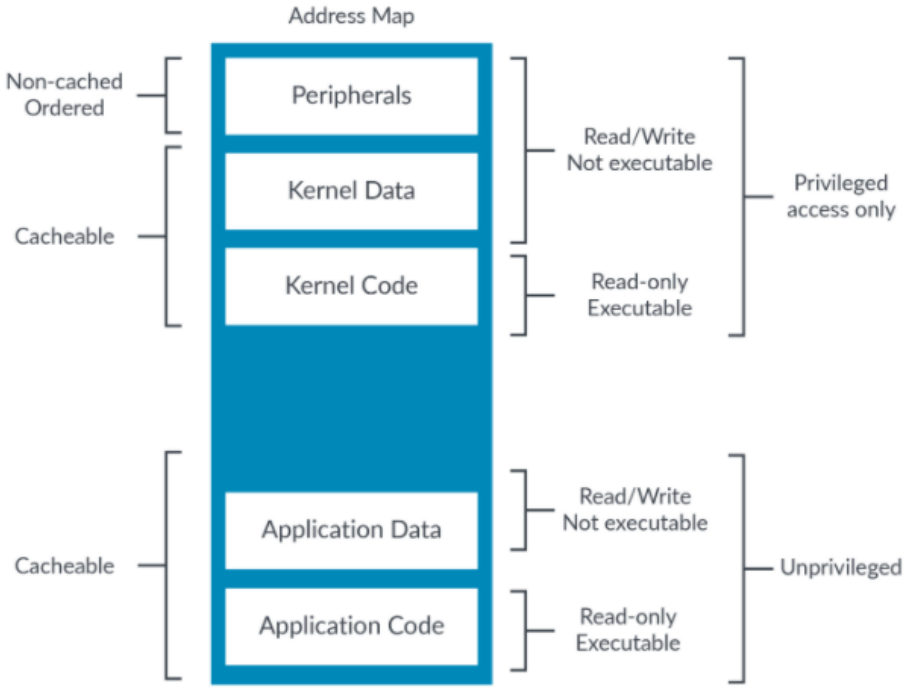
Ao organizar diferentes regiões de memória dentro de um espaço de endereçamento, temos como resultado o que é chamado de Mapa de Endereços. Podemos ver um exemplo abaixo.



Podemos ver as regiões de memórias mapeando periféricos em memória bem como regiões que carregam tanto dados quanto que carregam código do *kernel* e outras que carregam da aplicação.

Quando explicitamos essa informação para o processador, podemos definir comportamentos diferentes para cada região de memória.

Vejamos abaixo esses metadados no mapa de endereços.



Podemos, por exemplo, habilitar o cache de instruções da aplicação e do kernel, enquanto desativamos esta funcionalidade na região de memória responsável pelos periféricos, fazendo com que o sistema se comporte como esperado.

Não só isso, mas com essas informações, podemos fazer com que um processo não possa acessar a memória pertencente a outro (ou até mesmo a regiões do kernel), mantendo a integridade e segurança do sistema.

2.1.3. MMU - Memory Management Unit

Sabemos que a MMU nos permite mapear endereços lógicos para físicos, além de permitir a paginação, abstraindo o tamanho físico da memória. Ela também nos permite criar espaços de endereçamento virtuais, onde podemos reorganizar seções de memória de modo fácil, além nos permitir gerenciar o controle de acesso a estas regiões. Tais elementos são fundamentais para preparar a memória e separá-la entre processos com memória virtualmente ideal.

2.1.4. *Memory Layout* de Aplicações

As aplicações, de modo geral, esperam ter uma memória ideal, ondem assumem ter todo (ou boa parte do) espaço endereçável para elas, além de assumir que a memória é contígua.

Ao compilarmos o código, a memória de um processo é organizada em segmentos, sendo estas:

- Text
- Carregado a partir da seção *.text*
- Inclui as instruções do programa
- Data
- Contém as variáveis estáticas e globais

- Separado entre variáveis inicializadas e não inicializadas
- Initilized Data
- Carregado a partir da seção .data
- Contém os dados das variáveis inicializadas
- Non Initilized Data (BSS)
- Carregado a partir da seção .bss
- Contém o espaço para a alocação das variáveis
- É inicializado com o valor 0 pelo *crt0*
- Heap:
- Contém as variáveis alocadas dinamicamente de modo “global”
- Cresce dinamicamente a partir do fim do segmento BSS em direção ao final de endereçamento, ocupando a memória livre disponível
- Stack
- Contém as variáveis alocadas em um escopo, formando uma pilha
- Cresce dinamicamente a partir do final do espaço de endereçamento em direção ao começo, ocupando a memória livre disponível

2.1.5. Mapeamento Lógico de Memória

Para atingir o leiaute de memória esperado pelas aplicações, utilizaremos 2 grandes ferramentas da MMU no EPOS. A classe *Address_Space* e class *Segment*. Elas são na verdade abrastrações de *MMU

Directory* e *MMU

Chunk*. Então ao carregar um programa, iremos inicializar um novo espaço de endereçamento contendo os seguintes segmentos:

- Code

- Conterá a seção *.code*
— Terá propriedades **APPC**
---- Executável
---- Apenas Leitura
---- Não Global (Exclusivo de Processo)
- Data
— Conterá as seções *.data* e *.bss*
— Terá propriedades **APPD**
---- Não Executável
---- Leitura e Escrita
---- Não Global (Exclusivo de Processo)
- Contém também a Heap e as Stacks
- Tamanho são conhecidos em tempo de compilação através das variáveis ***HEAP_SIZE*** e ***STACK_SIZE***, definidas nas traits da aplicação.
- A alocação do espaço para a heap e as stacks dentro do segmento é feito pelo SETUP
- A heap é apenas uma por processo, sendo alocada após o BSS, localizado pelo símbolo ***_end***
- Cada thread possuiu 2 stacks (uma em nível de usuário e outra em nivel de sistema)
- Para multiplas aplicações, há também o Extra
- Funciona como um parâmetro para o processo inicial.
- Informação inserida durante a compilação

□□□□□□□

```
if(si->lm.has_ext) { // Check for EXTRA data in the boot image si->lm.app_extra =
si->lm.app_data + si->lm.app_data_size; si->lm.app_extra_size = si->bm.img_size -
```

si->bm.extras_offset; if(Traits<System>::multiheap) si->lm.app_extra_size = MMU::align_page(si->lm.app_extra_size); si->lm.app_data_size += si->lm.app_extra_size; }

- Interpretado como parâmetro durante a criação do Processo

```
int argc = static_cast<int>(si->lm.app_extra_size); char ** argv = reinterpret_cast<char **>(si->lm.app_extra); new (SYSTEM) Task(as, cs, ds, main, code, data, argc, argv);
```

Vale ressaltar que o System (kernel), também é separado em segmentos Code e Segment de forma semelhante, tendo atributos de proteção para acesso privilegiado.

2.1.6. Suporte a Múltiplas Aplicações

2.1.6.1. EPOS MKBI

O MKBI é responsável pela montagem das imagens de boot do EPOS, onde recebe uma ou mais aplicações como parâmetro. Essas várias aplicações são colocadas logo após os dados da aplicação principal. Esta região é conhecida como ***extra*** e para cada aplicação extra, é inserido o tamanho da aplicação e depois seu binário, tendo como indicação de fim da lista um número 0.

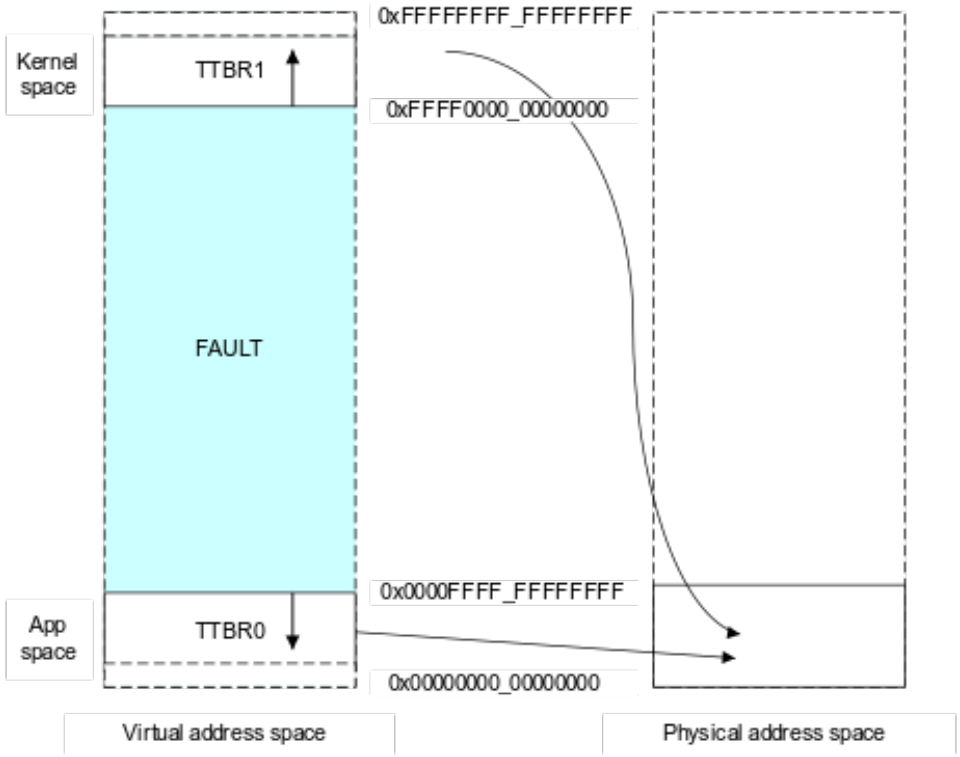
2.1.6.2. Troca de contexto

Entre threads a troca de contexto, segundo o manual, geralmente se salva ou restaura:

- general-purpose registers X0-X30.
- Advanced SIMD and Floating-point registers V0 - V31.
- Some status registers.
- TTBR0_EL1 and TTBR0.
- Thread Process ID (TPIDxxx) Registers.
- Address Space ID (ASID).

Entre processos, a troca de contexto é dita custosa, pois deve-se fazer a invalidação de páginas locais da TLB. Isso é necessário pois muda-se o espaço de endereçamento, e portanto os endereços lógicos não necessariamente apontam para o mesmo endereço físico. No armv8, a troca de contexto entre espaços de endereçamento diferentes, não necessariamente implica em flush da TLB, podendo-se utilizar a Address Space ID ou ASID. O ASID é um valor que é dado pelo SO para identificar exclusivamente um espaço de endereçamento. Utilizar este ID permite com que não seja necessário o flush completo da TLB na mudança de contexto, mantendo páginas não locais do processo antigo ocupando slots da TLB sem causar conflito.

A seguir apresentamos um modelo de memória sugerio pelo manual do arm para melhor explorar as diferentes translation tables.
A sugestão deles, é posicionar o kernel a partir do endereço 0xFFFF0000_00000000 pois acessos a endereços acima deste automaticamente estarão usando o TTBR1. Isso faz com que as páginas de kernel fiquem separadas em sua própria tabela eliminando competição com páginas de usuário.



Para realizar a troca de contexto sem implicar em grande overhead, utiliza-se o seguinte esquema de mapeamento do kernel:
Mapear o kernel space no mesmo endereço em todos os espaços de endereçamento. Este endereço necessariamente deve ser o mesmo entre todos os processos, para evitar overheads na troca de contexto. Para impedir que usuário faça alterações neste kernel space, utiliza-se mecanismos de proteção de acesso da CPU.

Este espaço de kernel conterá o código do kernel para tratar syscall, informações relevantes sobre o processo e uma pilha de kernel. Esta pilha de kernel é necessaria pois o kernel precisa armazenar os dados de chamadas de funções e dados locais. Se durante a execução de uma interrupção o kernel precisar usar a pilha, seria uma falha de segurança vazar dados para a pilha de usuario. Além disso é possível que a pilha do usuário estoure, e se o kernel estiver utilizando ela isso derruba o sistema inteiro.

Ou seja o processo, além de possuir uma pilha e heap de usuário, possui também uma pilha mas que ela mesmo não controla, que fica dentro do kernel space. Esta pilha pode ser única para todos os processos, porém isto poderia implicar em alguns problemas. Suponha a seguinte situação:

- Processo A faz uma syscall e durante a syscall é preemptado
- Processo B começa a executar e faz também uma syscall
- Processo B também é preemptado em favor de A
- A termina de executar e vai desempilhar

O Processo A imagina que tem a pilha que tinha quando perdeu a CPU, e na hora de desempilhar acaba desempilhando a pilha de B e estragando o contexto de B. Para evitar este tipo de problema, cada thread do sistema necessita de seu próprio kernel stack. Deste modo, mesmo que as threads executem system calls, e sofram preempção múltiplas vezes, não ocorrem conflitos.

A seguir temos o código que o construtor da task invoca no EPOS. Fica no arquivo thread.cc. Neste código percebemos que o programa deve ser multitask e não queremos stack no user space para a idle. O constructor_prologue vai criar a stack de kernel da nova thread. Percebe-se que cada thread tem o seu próprio kernel stack. O user stack é criado logo em seguida, como um segmento dentro da

heap do SO. O attach serve para obter um endereço lógico para poder fazer a inicialização da stack de usuário com os parâmetros relevantes. Para poder inicializar a stack é necessário utilizar um endereço virtual para poder endereçar a stack.

Depois de inicializar o stack de usuário utilizando o contexto atual, é feito um reattach no espaço de endereçamento da thread que está sendo criado. Ao final se inicializa a stack de sistema armazenando o endereço absoluto da stack de usuário nela, bem como outros atributos. E por último chama-se o constructor_epilogue que insere essa thread na fila de threads da task e acaba chamando um reschedule.

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) { if(multitask && !conf.stack_size) { // auto-expand, user-level stack // Create kernel stack constructor_prologue(conf.color, STACK_SIZE); _user_stack = new (SYSTEM) Segment(USER_STACK_SIZE); // Attach the thread's user-level stack to the current address space so we can initialize it Log_Addr ustack = Task::self()->address_space()->attach(_user_stack); // Initialize the thread's user-level stack and determine a relative stack pointer (usp) from the top of the stack Log_Addr usp = ustack + USER_STACK_SIZE; if(conf.criterion == MAIN) usp -= CPU::init_user_stack(usp, 0, an ...); // the main thread of each task must return to crt0 to call _fini (global destructors) before calling __exit else usp -= CPU::init_user_stack(usp, &__exit, an ...); // __exit will cause a Page Fault that must be properly handled // Detach the thread's user-level stack from the current address space Task::self()->address_space()->detach(_user_stack, ustack); // Attach the thread's user-level stack to its task's address space so it will be able to access it when it runs ustack = _task->address_space()->attach(_user_stack); // Determine an absolute stack pointer (usp) from the top of the thread's user-level stack considering the address it will see it when it runs usp = ustack + USER_STACK_SIZE - usp; // Initialize the thread's system-level stack _context = CPU::init_stack(usp, _stack + STACK_SIZE, &__exit, entry, an ...); } else { // single-task scenarios and idle thread, which is a kernel thread, don't have a user-level stack constructor_prologue(conf.color, conf.stack_size); _user_stack = 0; _context = CPU::init_stack(0, _stack + conf.stack_size, &__exit, entry, an ...); } constructor_epilogue(entry, STACK_SIZE); }

1. Bibligrafia utilizada

- https://developer.arm.com/documentation/102376/latest
- https://developer.arm.com/documentation/den0024/a
- https://epos.lisha.ufsc.br/EPOS+2+User+Guide
- TANENBAUM, Andrew S., BOS, Herbert. Modern Operating Systems. Fourth edition, 2015. Pearson.
- https://docente.ifrn.edu.br/rodrigotertulino/livros/notas-sobre-sistemas-operacionais

2.2. Task Memory Model - Grupo P

A Organização de Memória do EPOS utiliza de segmentos manipuláveis através de espaços de endereçamento que controlam a alocação e o mapeamento da memória física.

2.2.1. Segmento de Memória

Um Segmento de Memória é uma estrutura que representa uma parcela disponível de memória pronta para ser alocada, independente da política de gerenciamento de memória vigente.
Um Chunk é uma abstração para uma porção da memória.
A construção de um Chunk pode levar um endereço físico, o número de bytes e uma flag

Chunk(Phy_Addr phy_addr, unsigned int bytes, Flags flags)

A classe que representa um segmento de memória é uma especialização de Chunk

class Segment: public MMU::Chunk

2.2.2. MMU

A unidade de gerenciamento de memória (MMU) é a entidade incumbida da conversão de endereços lógicos para endereços físicos reorganizando os segmentos de memória conforme conveniente.

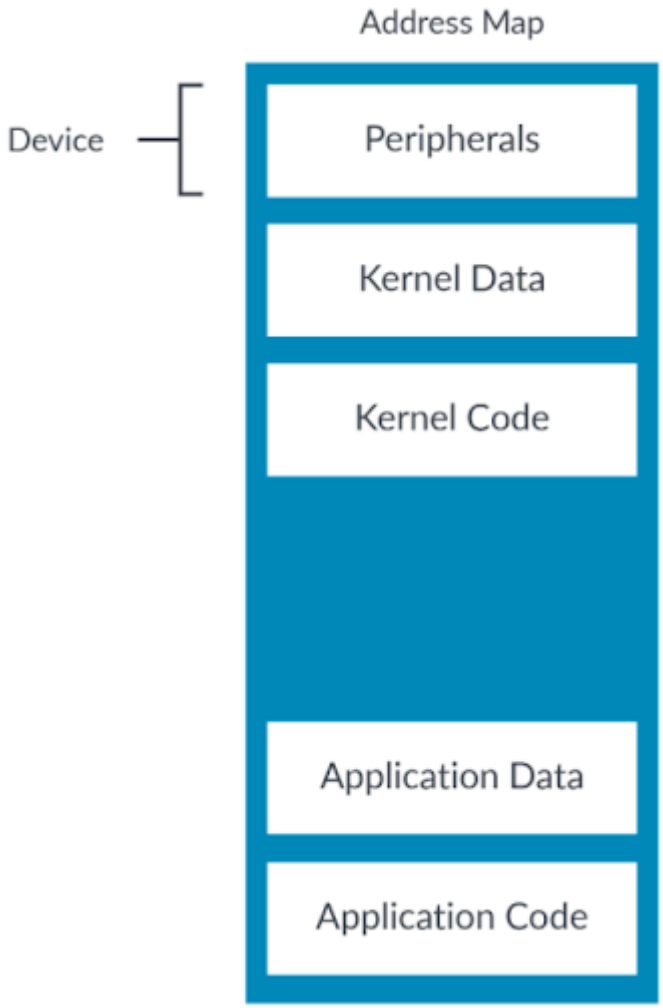
A Gerência de Processos ocorre por meio das mesmas ferramentas empregues na organização de memória.

O Processo é um ente abstrato composto por uma Thread e uma Task, sendo esta uma tarefa especificada e realizada pelo programa e aquela a entidade que se encarrega da execução de cada atividade, permitindo multithreading ainda que para uma mesma função.

2.2.3. Espaço de Endereçamento

O Espaço de Endereçamento determina o escopo do acesso de um processo aos endereços físicos de memória e pode ser configurado para processo único alocando toda a memória disponível de forma contígua ou de modo a permitir o mapeamento lógico dos endereços por paginação.

Com estes recursos gera-se um mapa de memória conforme abaixo estruturando os segmentos de memória pelos através dos espaços de endereçamento.



2.2.4. Task

Uma Task se equivale à região paralela de uma aplicação e pode restringir ou compartilhar seus dados com outras Threads. Sua implementação utiliza da abstração de Segmentos para comportar ambos código e área de dados de uso da Task.

2.2.5. Thread

Threads performam uma Task e para isso compartilham dos recursos que dela advém, a exceção da pilha de segmento de dados - alocada da heap na instanciação - que cada Thread reserva a si própria bem como a preservação do seu contexto de execução.

2.2.6. Pilhas de Usuário e de Sistema

O sistema operacional contém código para syscall, e uma pilha do sistema. Ele é mapeado no mesmo endereço em todos os Address Space. Manter o mesmo endereço evita problemas por trocas de contexto no reescalonamento de processos.
A pilha do kernel é usada para armazenar dados locais e dados de chamadas das funções. Uma pilha de kernel para cada processo permite evitar problemas na stack de usuário afete a pilha de kernel e consequentemente o sistema inteiro.
A construção de uma Thread é definida em process.h

No construtor

```
Thread::Thread
```

temos, para a criação de uma pilha de usuário, primeiro a criação da pilha de sistema. Com a chamada de Thread::constructor_prologue, a pilha de kernel da nova thread é criada. Percebe-se que cada thread tem o seu próprio kernel stack. Na linha seguinte a stack de usuário é criada como um novo Segment. Em seguida é obtido um endereço lógico para a stack de usuário para que possa ser feita sua inicialização
Ao fim da função a chamada de Thread::constructor_epilogue insere faz com que a nova thread seja enfileirada.

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) { if(multitask && !conf.stack_size) { // auto-expand, user-level stack constructor_prologue(conf.color, STACK_SIZE); _user_stack = new (SYSTEM) Segment(USER_STACK_SIZE); // Attach the thread's user-level stack to the current address space so we can initialize it Log_Addr ustack = Task::self()->address_space()->attach(_user_stack); // Initialize the thread's user-level stack and determine a relative stack pointer (usp) from the top of the stack Log_Addr usp = ustack + USER_STACK_SIZE; if(conf.criterion == MAIN) usp -= CPU::init_user_stack(usp, 0, an ...); // the main thread of each task must return to crt0 to call _fini (global destructors) before calling __exit else usp -= CPU::init_user_stack(usp, &__exit, an ...); // __exit will cause a Page Fault that must be properly handled // Detach the thread's user-level stack from the current address space Task::self()->address_space()->detach(_user_stack, ustack); // Attach the thread's user-level stack to its task's address space so it will be able to access it when it runs ustack = _task->address_space()->attach(_user_stack); // Determine an absolute stack pointer (usp) from the top of the thread's user-level stack considering the address it will see it when it runs usp = ustack + USER_STACK_SIZE - usp; // Initialize the thread's system-level stack _context = CPU::init_stack(usp, _stack + STACK_SIZE, &__exit, entry, an ...); } else { // single-task scenarios and idle thread, which is a kernel thread, don't have a user-level stack constructor_prologue(conf.color, conf.stack_size); _user_stack = 0; _context = CPU::init_stack(0, _stack + conf.stack_size, &__exit, entry, an ...); } constructor_epilogue(entry, STACK_SIZE); }

2.2.7. Bibliografia:

- https://epos.lisha.ufsc.br/EPOS+Documentation
- https://developer.arm.com/documentation/102376/0100

2.3. Interprocess Comunication

Como sabemos, processos são abstrações do sistema operacional que permitem que diversos programas sejam executados concorrentemente em uma máquina em um contexto próprio sem precisar ter conhecimento das demais execuções. Porém, pode ser útil que processos interajam entre si a fim de realizar algum objetivo em comum e neste contexto foram criados os mecanismos de comunicação entre processos (IPC - Interprocess Comunication).

2.3.1. Importância

A comunicação entre processos pode permitir aumento na velocidade de computação, modularização do software e flexibilização na realização de algumas tarefas. Microkernels são muito beneficiados por IPC pois trechos normalmente implementados diretamente em SOs monolíticos podem ser modularizados e desta forma o núcleo do SO permanece menor e mais conciso.

2.3.2. Modelos de IPC

Existem dois modelos principais que são usados em IPC, cada um com suas próprias vantagens e desvantagens: Memória compartilhada e troca de mensagens.

2.3.2.1. Memória Compartilhada

No modelo de memória compartilhada, como o nome sugere, uma mesma região da memória faz parte do espaço de endereçamento de ambos os processos interessados. Normalmente uma região de memória compartilhada é criada em um processo e os demais podem anexa-la ao seu próprio espaço de endereçamento.

As vantagens das implementações que utilizam este modelo é certamente a velocidade, já que não é necessário realizar muitas cópias em memória ou chamadas de sistema, a flexibilidade, pois os processos podem utilizar aquele trecho da maneira que preferirem com acesso randômico, e a facilidade de implementação.

Porém existem problemas inerentes a este modelo, pois devido a necessidade de que vários processos possam ler e escrever em uma mesma região de memória é necessário que o usuário implemente mecanismos de controle de concorrência como mutexes e semáforos, aumentando a complexidade da utilização e pode abrir espaço para falhas de segurança.

2.3.2.2. Troca de Mensagem

Neste modelo a comunicação entre os processos é feita enviando mensagens para o processo com o qual desejamos realizar a comunicação. Essa comunicação pode ser direta quando um processo A faz uma referência direta ao processo B, ou indireta, quando o processo A envia a mensagem para uma mailbox, que é um buffer capaz de armazenar em ordem as mesagens que ainda não foram recebidas pelo processo B. Normalmente o SO precisa agir como intermediador para transmitir estas mensagens, e pode ser o responsável por criar e fazer o controle de concorrência das mailboxes.

As vantagens deste modelo de IPC é a faciliade de utilização por parte do usuário e a capacidade de ser utilizado em sistemas distribuidos, onde os processos podem não compartilhar fisicamente a mesma memória.

Os principais problemas deste modelo são a lentidão em comparação com a memória compartilhada devido as várias cópias de mensagens que são realizadas, a necessidade de realizar syscalls em cada interação e a dificuldade de implementação.

Troca de mensagens costuma ser muito útil para realizar RPC (Remote Procedure Call), que é quando um processo pode chamar funções de outro processo.

2.3.2.3. Exemplos de IPC

2.3.2.3.1. Pipes

É uma forma de implementar IPC atravez de troca de mensagens capaz de conectar dois processos apenas. A comunicação é feita de forma unidirecional. Normalmente é possível ler nos pipes sem considerar qual foi o processo que escreveu, e isso pode ser uma vulnerabilidade, tornando esta uma forma de comunicação pouco segura.

Pipes são utilizados por exemplo no shell, quando se deseja que a saída de um programa seja utilizada como entrada para outro. Por exemplo:
$ ps aux | grep <user>

2.3.2.3.2. Sockets

Sockets são uma outra forma de implementar IPC através da troca de mensagens, mas que é capaz de realizar a comunicação nas duas direções e até conectar multiplos processos. Sockets são mais utilizados em um contexto de cliente/servidor, e utilizando a rede, mas devido sua versatilidade podem realizar uma comunicação entre processos da mesma máquina sem dificuldades.

2.3.2.3.3. Área de Transferência

A área de transferência é um mecanismo muito comum em diversos SOs, que pode ser implementado através de memória compartilhada. Este método é o que permite operações de recortar, copiar e colar entre aplicativos, que apenas necessitam concordar com o tipo de formato que está sendo transferido. Este método não é tão versátil para diferentes aplicações, mas especialmente em sistemas com interface gráfica podem ser bem úteis.

2.3.2.4. Microkernel Epos

Para demonstrar a implementação de IPC em um microkernel real, vamos utilizar uma implementação feita no semestre passado para Armv7. Para exemplificar o funcionamento vamos precisar das seguintes classes:

2.3.2.4.1. Message

A classe message abstrai o que deve ser uma mensagem no IPC, isso é feito através de 2 enums. Um representa o que a mensagem deve fazer, como SEMAPHORE_CREATE, SEMAPHORE_P e SEMAPHORE_V, enquanto o outro representa que tipo de entidade deve ser alterada, como MUTEX, SEMAPHORE, CHRONOMETER, etc.

```
template<> struct Traits<Build>: public Traits<void> class Message { public: enum { // ... Economizando espaço SEMAPHORE_CREATE, SEMAPHORE_P, SEMAPHORE_V, // ... Economizando espaço }; enum ENTITY { // ... Economizando espaço MUTEX, SEMAPHORE, CHRONOMETER, // ... Economizando espaço };
```

O construtor da classe recebe um ID, a entidade que desejamos (como citado acima e definido no ENUM), e um método (que também tá no enum) e alguns parâmetros genéricos. Tudo isso é aplicado nas devidas variáveis na inicialização. Perceba que o método act() faz uma chamada syscall passando o próprio ponteiro como argumento. Voltaremos nessa função mais tarde.

```
public: template<typename ... Tn> Message(int id, int entity, int method, Tn ... an): _id(id), _entity(entity), _method(method) {set_params(an ...);} template<typename ... Tn> void get_params(Tn && ... an); {DESERIALIZE(_params, index, an ...);} template<typename ... Tn> void set_params(const Tn & ... an) {SERIALIZE(_params, index, an ...);} void act() { _syscall(this); } int id(){return _id;} int entity(){return _entity;} int method(){return _method;} int result(){return _result;} void result(int r){_result = r;} char* params(){return _params;} private: int _id; int _entity; int _method; int _result; char _params[256]; };
```

2.3.2.4.2. Syscall

O método act(), em Message, chama a função _syscall que no ARMV7 faz uma chamada para CPU::syscall(m) que simplesmente salva o contexto do processo, move o valor de msg para o registrador r0, entra no modo supervisor e depois recupera o contexto.

```
void CPU::syscall(void * msg) { ASM( "push {r0} \n" "mov r0, %0 \n" "SVC 0x0 \n" "pop {r0} \n" "" :: "r"(msg) ); }
```

Essa chamada de sistema eventualmente vai executar a função CPU::syscalled, que simplesmente

chama a função _sysexec e recupera o contexto.

```
void CPU::syscalled() { ASM("push {lr} \n" "push {r0} \n" "bl _sysexec \n" "pop {r0} \n" "pop {lr} \n" ); }
```

A função _sysexec chama a função _exec() da classe Agent.
void _sysexec() { Agent::_exec();}

2.3.2.4.3. Agent

A classe Agent é filha da classe Message. A função _exec que foi chamada na syscall vai recuperar o valor do ponteiro passado no registrador e transforma-lo em um Agent. Logo em seguida chama o método exec() do agente que é um enorme switch case que executa, no nosso destino a função desejada de acordo com o parâmetro entity que passamos lá no início em Message.

```
class Agent: public Message { public: static void _exec(){ Agent * agt; ASM("mov %0, r0 " : "=r"(agt) :); agt->exec(); } void exec() { switch(entity()) { // ... case Message::ENTITY::MUTEX: handle_mutex(); break; case Message::ENTITY::SEMAPHORE: handle_semaphore(); break; case Message::ENTITY::CHRONOMETER: handle_chronometer(); break; // ... default: break; } } // ...
```

Se a entity for um semáforo, por exemplo, a função handle_semaphore será executada e teremos outro switch case que executa a função desejada de acordo com o parâmetro method, que passamos também na criação de Message.

2.3.2.4.4. Stub_Semaphore

Para realizar a comunicação entre processos neste caso, através de chamadas de métodos em outros processos, podemos usar uma classe como Stub_Semaphore. Esta classe, e outras semelhantes, funcionam como uma forma conveniente de realizar as chamadas por Message.

Como é mostrado no código abaixo, podemos instanciar e utilizar Stub_Semaphore como se fosse um semáforo comum. Mas todos os métodos dele são chamadas de sistema através de mensagens. O semáforo real, será criado em um espaço de endereçamento do sistema, como foi mostrado na classe Agent. Assim, vários processos poderão interagir com o mesmo semáforo através apenas de mensagens e sem compartilhar o seu espaço de endereçamento, caracterizando uma forma de comunicação entre processos.

```
class Stub_Semaphore // ... { public: template<typename ... Tn> Stub_Semaphore(int v, Tn ... an){ Message * msg = new Message(0, Message::ENTITY::SEMAPHORE, Message::SEMAPHORE_CREATE, v); msg->act(); _id = msg->result(); } void p(){ Message * msg = new Message(_id, Message::ENTITY::SEMAPHORE, Message::SEMAPHORE_P); msg->act(); } void v(){ Message * msg = new Message(_id, Message::ENTITY::SEMAPHORE, Message::SEMAPHORE_V); msg->act(); } // ... };
```

2.3.3. Referências

Modern Operating Systems - TANEMBAUM, Andrew & BOS, Herbert
https://epos.lisha.ufsc.br/EPOS+for+Raspberry+Pi#Inter-Process_Communication
https://linux.die.net/man/5/ipc

2.4. System Calls in ARMv8 with AArch64

Autores: André William Régis, João Pedro Adami do Nascimento, Nicole Schmidt

2.4.1. 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.

2.4.2. 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.

2.4.3. 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.

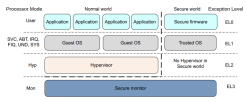


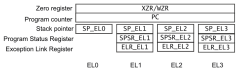
Figura: Exception Levels e Processor Modes do ARMv8 (Fonte: ARM Cortex-A Series Programmer's Guide for ARMv8-A, Pg 3-2)

2.4.3.1. 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.



Fonte: ARM Cortex-A Series Programmer's Guide for ARMv8-A, Pg 4-3

2.4.4. 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.

2.4.5. 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.

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

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.

2.4.6. 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.

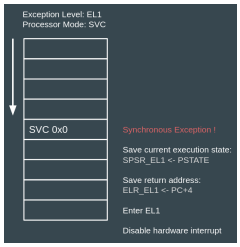


Figura: Execução de um programa aplicativo interrompida por uma exceção síncrona (**Fonte:** Própria)

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	

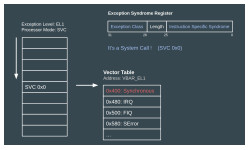
Address	Exception type	Description
+ 0x600	Synchronous	Lower EL using AArch32
+ 0x680	IRQ/vIRQ	
+ 0x700	FIQ/vFIQ	
+ 0x780	SError/vSError	

Figura: Offsets da Vector Table relativo ao endereço da Vector Table (Fonte: ARM Cortex-A Series Programmer's Guide for ARMv8-A, Pg 10-12 e 10-13)

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:

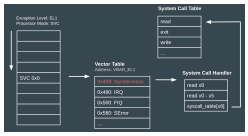
- Bits 31:26: classe da exceção, que pode ser unallocated instruction, data abort, SVC, HVC, entre outros.
- Bit 25: indicará o tamanho da instrução que provocou a exceção: 0 se for de 16 bits e 1 se for de 32 bits
- Bits 24:0: contém informação específica para cada classe de exceção. No caso de uma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



Fonte: Própria

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).



Fonte: Própria

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.

2.4.7. Demonstração Bare Metal

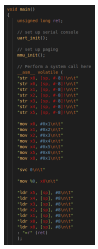
Arquivos de destaque:

- **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.



Fonte: main.c

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
```

Fonte: start.S

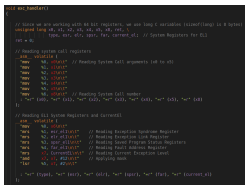
```
_vectors:
    // synchronous
    .align  7
    mov     x0, #0  // Passing exception type
    bl      exc_handler
    eret

    // IRQ
    .align  7
    mov     x0, #1
    bl      exc_handler
    eret

    // FIQ
    .align  7
    mov     x0, #2
    bl      exc_handler
    eret

    // SError
    .align  7
    mov     x0, #3
    bl      exc_handler
    eret
```

Fonte: start.S

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.

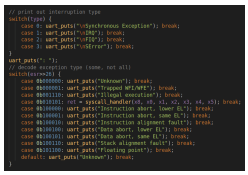
O exception handler está definido em exc.c. Ao entrarmos nele, iremos ler os registradores da system call (de x0 à x5 estarão os argumentos e em x8 o número da system call), o registrador x6 para poder identificar o tipo de exceção, e também iremos ler os registradores de sistema do ARM, que são: ESR_EL1,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



Fonte: exc.c

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.



Fonte: exc.c

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.

Fonte: exc.c

Fonte: exc.c

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.

Fonte: exc.c

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.

Fonte: exc.c

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.

2.4.8. 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.

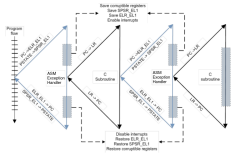
Um tratador de interrupções começa com as interrupções desabilitadas, e portanto se o seu tempo de execução for longo demais a ponto de que deveria ocorrer outra exceção durante o seu tratamento, ela será perdida. Este é um cenário que poderia vir a acontecer no caso de system calls, visto que pode haver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



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

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.

Fonte: Própria

2.4.9. 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

https://github.com/JPADN/syscall-bare-metal

2.5. System Calls ARMv8 - Grupo E

2.5.1. Motivação

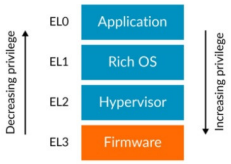
Com avanço desenvolvimento dos processadores modernos e adoção de modos de operação separados, com níveis variados de privilégio, era necessário a criação de um mecanismo para transferir seguramente o controle de modos de menor para código com maior privilégio. O código com menor privilégio não pode simplesmente transferir o controle para código com maior privilégio em qualquer ponto do código e em qualquer estado do processador; permitir essa transferência seria permitir a quebra da segurança do sistema. Por exemplo, o código com menor privilégio poderia levar o código com maior privilégio a ser executado na ordem incorreta, ou disponibilizar a ele uma pilha errada.

2.5.2. O que são Syscall?

A chamada de sistema (System Call) é o mecanismo programático pelo qual um programa de computador solicita um serviço do núcleo do sistema operacional sobre o qual ele está sendo executado. Geralmente, os sistemas fornecem algum tipo de biblioteca ou API que fica entre os

programas normais e o SO. Em sistemas do tipo Unix, essa API geralmente faz parte de uma implementação da biblioteca C (libc), como glibc, que fornece funções de wrapping para as chamadas do sistema.

2.5.3. Níveis de exceção no ARMv8-Cortex A



No ARMv8 existem quatro níveis de exceção - EL0 a EL3. EL0 tem o menor privilégio onde os aplicativos do usuário são executados. O kernel Linux é executado em EL1. O hipervisor é executado em EL2 e, por fim, EL3 tem o maior privilégio . A elevação de um nível de exceção para o próximo nível de exceção é obtida definindo exceções.

Desse modo, existem instruções especiais para fazer essas chamadas de sistema. Essas instruções causam uma exceção, que permite a entrada controlada em um nível de exceção mais privilegiado.

- **SVC** - Chamada de Supervisor (Supervisor Call)

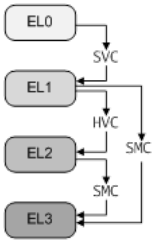
Causa uma exceção visando EL1.
Usado por um aplicativo para chamar o sistema operacional Linux por exemplo.

- **HVC** - chamada de hipervisor (Hypervisor call)

Causa uma exceção visando EL2.
Usado por um sistema operacional para chamar o hypervisor, não disponível em EL0.

- **SMC** - Chamada de monitor segura (Secure monitor call)

Causa uma exceção visando EL3.
Usado por um sistema operacional ou hipervisor para chamar o firmware EL3, não disponível em EL0.



Se uma exceção for executada a partir de um nível de exceção superior ao nível de exceção de destino, a exceção é levada para o nível de exceção atual. Isso significa que um SVC em EL2 causaria uma entrada de exceção em EL2. Da mesma forma, um HVC em EL3 causa entrada de exceção em EL3. Isso é consistente com a regra de que uma exceção nunca pode fazer com que o processador perca privilégios.

2.5.4. Process State (PSTATE) e Registradores de Sistema

Na arquitetura Armv8, PSTATE é uma abstração para representar as informações do estado do processo. Diferentemente da AArch32, AArch64 não dispõe do registrador CPSR (Current Program

State Register), então essas informações são guardadas de forma independente, em registradores de propósito especial que devem ser acessados pelas instruções MRS (leitura) e MSR (escrita).

No contexto de tratamento de exceções, em AArch64, os campos mais relevantes são:

Condition Flags: utilizados para guardar informações de resultados de algumas instruções que utilizam a ALU. Armazenados no registrador 'NZCV'.

- **N**: Negative Condition flag.
- **Z**: Zero Condition flag.
- **C**: Carry Condition flag.
- **V**: Overflow Condition flag

Execution State Controls

- **M**: Current Execution state. (0 = AArch64; 1 = AArch32).
- **EL(M3:2)**: Current Exception Level. Armazenado no registrador 'CurrentEL'
- **SP(M0)**: Stack pointer register selection bit. (0 = SP_EL0; 1 = SP_ELn, sendo n = M3:2). Armazenado no registrador 'SPSel'

Exception Mask Bits: quando ativos, desabilitam o tratamento de suas respectivas exceções. Armazenados no registrador 'DAIF'.

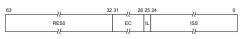
- **D**: Debug exceptions mask.
- **A**: SError interrupt Process state mask.
- **I**: IRQ interrupt Process state mask.
- **F**: FIQ interrupt Process state mask.

2.5.4.1. SPSR_ELn - Saved Program Status Register

Ao tomar uma exceção que leve ao nível ELn, o processador guarda em SPSR_ELn o estado do processo corrente, copiando os campos de PSTATE.

2.5.4.2. ESR_ELn - Exception Syndrome Register

É onde o processador guarda a informação de causa para uma exceção ocorrida, que leve ao nível ELn.



- Exception Class (EC): Indica o motivo da exceção
- IL: Tamanho da instrução (0 = 16 bits; 1 = 32 bits)
- Instruction Specific Syndrome (ISS) : Informação específica sobre a exceção

2.5.4.3. ELR_ELn - Exception Link Register

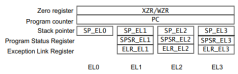
Registrador utilizado para guardar o endereço de retorno ao tomar uma exceção para ELn.

2.5.4.4. VBAR_ELn - Vector Based Address Register

Registrador utilizado para armazenar o endereço base para a vector table de ELn.

Como pode-se ver na tabela a seguir, existem quatro stack pointers distintos, sendo que cada nível

pode ter acesso ao SP do seu próprio nível ou do nível 0.



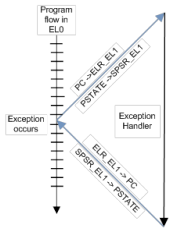
2.5.5. Instrução SVC (Supervisor Call)

A instrução SVC tipicamente é utilizada no código de usuário em nível EL0, causando uma exceção para ser tratada no nível EL1. Ao ser executada o processador guarda o valor 0x15 no campo EC (Exception Class) do registrador ESR_EL1, o qual identifica a exceção como sendo uma Supervisor Call exception, e por convenção, o valor do imediato de argumento da SVC, é guardado no campo ISS do mesmo, apesar de não ter um uso especificado.

```
svc #<imm> // <imm> = valor imediato de 16 bits, sem sinal.
```

2.5.6. Exceções e Vector Table.

Exceções são condições ou eventos de sistema que requerem uma ação por um software especial, o tratador de exceções (exception handler), causando assim uma interrupção no fluxo de execução do programa. A imagem a seguir ilustra bem este conceito.



2.5.6.1. Entrada de uma exceção

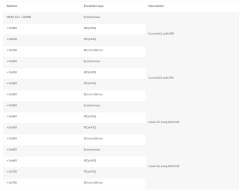
Na ocorrência de uma exceção, que leve ao nível ELn, os seguintes passos são efetuados pelo processador:

- O PSTATE do nível atual é salvo no registrador SPSR_ELn, para que possa ser restaurado ao retornar da exceção.
- O endereço para retornar ao contexto anterior ao da exceção é salvo no registrador ELR_ELn.
- Os bits de PSTATE são atualizados, para refletir o novo estado do processador, sendo que os bits {D, A, I, F} são setados para 1, fazendo com que demais exceções sejam desabilitadas. Nota-se que, por padrão, o stack pointer também é alterado para SP_ELn, mesmo que o processador já estivesse em ELn.
- Se a exceção for síncrona ou do tipo SError, a informação caracterizando o motivo da exceção ter ocorrido é salva no registrador ESR_ELn.
- Por fim, a execução passa para o nível ELn, com PC recebendo o endereço do exception vector (vector table), o qual é obtido a partir do registrador VBAR_ELn.

2.5.6.2. Tratamento de uma exceção

Exceções são tratadas em uma estrutura de dados contínua na memória chamada de vector table sendo que cada nível de exceção tem sua própria. O endereço base pode ser acessado pelo registrador de sistema VBAR_ELn. A vector table é alinhada a 128 bytes (0x80) sendo assim, cada

tratador de exceção pode ter no máximo 32 instruções. A figura a seguir mostra uma implementação típica de uma vector table.



System calls são instruções síncronas, e system calls partindo de aplicações (EL0) são tratadas em VBAR_ELn + 0x400, já que estão vindo de um nível menor em execução AArch64.

Enquanto existe um consenso sobre a estrutura de uma vector table, cada kernel deve implementar seu próprio modo de tratar exceções. A maneira como isso é feito tipicamente consiste em consultar os bits que correspondem a classe de exceção (EC) do registrador ESR_ELn.

2.5.7. Fazendo uma chamada de sistema

Como vimos anteriormente, para realizar uma chamada de sistema precisamos usar a instrução SVC. No entanto é preciso lembrar que uma aplicação sempre irá executar em EL0 e chamando SVC estamos elevando o nível de exceção para EL1. No caso do kernel Linux, o sistema operacional habita em EL1, então para fazer uma chamada de sistema basta identificar a syscall no registrador X8 e passar os argumentos em X0-X5, sendo que o valor retornado será armazenado em X0.

Para saber mais sobre quais as chamadas de sistema do Linux e quais argumentos precisam ser entregues, é possível encontrar a documentação em https://chromium.googlesource.com/chromiumos/docs/+/master/constants/syscalls.md#arm64-64_bit

Segue um exemplo de uma chamda de write (0x40) para stdout (0x1) e logo em seguida exit (0x5d)

```
# AARCH64 Hello World system call

.global _start
.section .text

_start:
    # write system call
    mov x8, #0x40
    mov x0, #0x1
    ldr x1, =message
    mov x2, #13
    svc 0

    # exit system call
    mov x8, #0x5d
    mov x0, #0x41
    svc 0

.section .data:
    message:
    .ascii "Hello, World\0"
```

Para o kernel XNU do macOS, no entanto, a implementação é diferente para como tratar system calls. No XNU passamos o valor que identifica a system call em X16 e os argumentos são lidos de X0-X9.

Assim podemos ver que cada kernel implementa de sua maneira o modo que deve tratar suas chamadas de sistema.

2.5.7.1. Chamadas de sistema para EL2

O nível de exceção EL2, onde em sistemas comuns habita apenas o hypervisor, pode conter o sistema operacional também. Neste caso, sabemos que a chamada de sistema HVC acessa EL2, mas há um pequeno problema, aplicações em EL0 não podem chamar HVC diretamente. Apenas execuções em EL1 podem usar a instrução HVC, sendo assim, quando uma aplicação em EL0 precisa fazer uma chamada de sistema em um kernel que habita EL2 é preciso primeiro chamar SVC, o kernel então, em modo EL1 precisa ter uma implementação no handler de SVC que faça a chamada HVC em nome da aplicação.

2.5.7.2. Retornando de uma exceção

Para retornar de uma exceção, é necessária a chamada instrução ERET. Isso fará com que o nível de exceção volte ao que era quando a exceção foi gerada, a execução seja redirecionada no PC para continuar a partir do endereço guardado em ELR_ELn e o estado PSTATE do processo seja restabelecido de SPSR_ELn. Nota-se que essas operações ocorrem de forma atômica, para assegurar que o processador não entre em um estado indefinido/inconsistente.

2.5.8. Exemplo simples em bare-metal RaspberryPi3

Para o exemplo a seguir foi usado como referência o repositório localizado em: https://github.com/bztsrc/raspi3-tutorial/tree/master/11_exceptions

Neste repositorio encontra-se uma implementação mínima de um kernel para RaspberryPi 3 com incialização da UART e uma vector table simples.

A vector tem apenas os tratamentos para exceções vindas do mesmo nível de exceção, visto que o sistema inicia em EL1 e continua em EL1 durante a execução. Desta maneira, uma chamada de sistema usando SVC deve ser tratada no endereço base da tabela de vetores, onde ocorre o tratamento de exceções síncronas de mesmo nível.

Segue abaixo a parte da vector table que nos interessa.

```
_vectors:
    // synchronous
    .align  7
    mrs         x25, esr_el1
    lsr         x24, x25, #26
    cmp         x24, #0x15
    b.eq        svc_handler
    eret
```

Na primeira instrução extraímos o valor de ESR_EL1 para X25, para na segunda instrução fazermos um shift a direita de 26 bits e guardamos o valor em X24. Esta operação nos dá o valor de EC, a classe da exceção, assim na instrução seguinte comparamos o este valor com 0x15, que é o valor que nos diz que a origem desta exceção foi uma instrução SVC. Após esta confirmação fazemos um branch para a função implementada em exc.c que realiza o tratamento.

A função de tratamento é simples, apenas comparamos os valores dos registradores de entrada e então imprimimos na tela estes valores.

```
/**
* Simple SVC handler
*/
void svc_handler(unsigned long x0, unsigned long x1)
{
    if (x0 == 77) {
        uart_puts("X0 = 0x4d\t");
    }

    if (x1 == 60) {
        uart_puts("X1 = 0x3c\n");
    }
}
```

Podemos confirmar que estes foram os valores inseridos nos registradores olhando o arquivo main.c

```
void main()
{
    // set up serial console
    uart_init();

    // set up paging
    mmu_init();

        __asm__ volatile (
    "mov x0, #0x4D\n\t"   // set register X0 to magic number
    "mov x1, #0x3C\n\t"   // set register X1 to another magic number

    "svc 0\n\t"
    );

    // echo everything back
    while(1) {
        uart_send(uart_getc());
    }
}
```

2.5.9. Referências

https://pt.wikipedia.org/wiki/Chamada_de_sistema
https://developer.arm.com/documentation/102374/0101/System-calls
https://eastrivervillage.com/Anatomy-of-Linux-system-call-in-ARM64/
https://developer.arm.com/documentation/den0024/a/ch10s02s04

2.6. Resource Management - Grupo N

2.6.1. Processing Time

O OS precisa administrar o tempo dos múltiplos processos para tornar a experiência do usuário a mais rápida possível. Essa é a principal tarefa do escalonador, trocar de processos quando necessário, sempre utilizando um algoritmo de escalonamento. São eles que dividem o tempo de processamento para cada processo, e qual a ordem eles irão executar, como também como as interrupções serão tratadas.

Quando o OS troca entre processos, ocorre o que é chamado de Context Switch. Ele é o processo que salva e recupera o contexto dos diferentes processos e threads. Este procedimento é o principal fator no qual se tem a impressão que múltiplas tarefas estão sendo executadas simultaneamente, o que não é o caso é um Sistema Operacional Single Core.

Um exemplo de um context switch seria:

1. Trocar o estado do processo corrente de Running para Ready
2. Salvar o conteúdo de todos os registradores na PCB do processo
3. Inserir o ID do processo no fim da fila de processos
4. Remover da fila o primeiro processo (de acordo com o algoritmo de escalonamento), ajustar o seu estado para Running e copiar os valores dos registradores salvos na área do seu PCB para os registradores da CPU.
5. Por fim, resetar o clock desse novo processo corrente.

Um multitasking OS precisa de uma forma para decidir quando deve ocorrer um context switch. Caso esse procedimento fosse feito de forma randômica, isso reduziria a previsibilidade do sistema assim como a sua eficiência pelos recursos necessários para tal procedimento.

Uma forma de lidar com esse context switch é como se faz em um preemptive multitasking OS. Dessa maneira, é necessário armazenar um número grande de informações de cada processo para manter a troca de contexto em uma velocidade adequada. Mas isso garante um alocamento de recursos justo entre os processos. Além disso, os diferentes níveis de prioridades são essenciais para não impactar a experiência do usuário, quando o mesmo precisar de uma ação que seja mais importante no momento.

2.6.2. Memory Management

A memória é um recurso fundamental para a execução de um processo, pois os dados, e até mesmo o programa a ser executado fica armazenado na memória. Em um sistema multitasking, há a necessidade de gerenciar o uso da memória pelos processos. Como a memória é limitada, e podemos executar diversas tarefas, a memória tende a rapidamente se esgotar. O Sistema Operacional trata, portanto, de fazer a gerência da memória.
Para isso, o SO utiliza paginação e memória virtual, permitindo assim maior eficiência. Através da paginação, a memória é dividida em page frames, e esses espaços são distribuídos entre os programas em execução de acordo com a política adotada. Também é responsabilidade do SO determinar o quanto de memória máxima é permitida à uma tarefa, assim como dos demais recursos.
Outra responsabilidade fundamental, é a proteção da memória, onde o SO deve garantir que uma tarefa não modifique o espaço de endereçamento de outra tarefa.
Por fim, o gerenciamento de memória deve tratar da desalocação de memória após o término de uma tarefa. Caso a desalocação não seja feita pelo próprio programa, o SO, deve garantir que a memória é desalocada após o programa terminar.

Paging: Através da paginação, a memória é dividida em page frames, e o programa em páginas. As páginas em uso são copiadas para a memória de forma não-contígua, permitindo que um programa utilize qualquer área livre da memória. É utilizada uma tabela de páginas para manter o controle.

Segmentation: Os blocos de memória alocados aos processos são divididos em segmentos de diferentes tamanhos, de acordo com a necessidade.

Virtual Memory: A vantagem da memória virtual é permitir estender o tamanho da memória, mesmo que a memória física permaneça menor. Com isso, os locos que não estão em uso no momento, mas podem ser usados futuramente, são salvos no disco, e as informações dos blocos ficam armazenadas, permitindo maior eficiência na troca de blocos.

2.6.3. Managing Input/Output devices

Um papel importante desempenhado pelo Sistema Operacional é o uso e o controle de dispositivos de entrada e saída (I/O).
O controle de vários dispositivos conectados ao computador é uma das principais preocupações dos projetistas de sistemas operacionais, visto que estes dispositivos apresentam diferentes funcionalidades e velocidades, como um mouse, um disco rígido e um CD-ROM. Para controlá-los são necessários métodos variados. Esses métodos formam o subsistema de E/S do kernel do SO que separa o resto do kernel das complicações de gerenciamento de dispositivos de I/O.

Os dispositivos periféricos se comunicam com a máquina por meio de um ponto de conexão também chamado de porta. Uma porta de E/S geralmente consiste em quatro registradores diferentes: (1) status, (2) controle, (3) entrada de dados e (4) saída de dados.

O SO controla os dispositivos I/O por meio de interrupções ou polling. Uma interrupção é um mecanismo de hardware que permite à CPU detectar que um dispositivo precisa de sua atenção. Quando a CPU detecta um sinal de interrupção a CPU para sua tarefa atualmente em execução e responde à interrupção enviada pelo dispositivo de E / S passando o controle para o manipulador de interrupção (interrupt handler). O polling faz referência a uma operação de consulta constante aos dispositivos para criar uma atividade síncrona sem o uso de interrupções para verificar seu estado e caso o dispositivo esteja pronto o SO passa o controle para o dispositivo.

De forma geral, o sistema operacional gerencia os dispositivos de I/O de várias maneiras:

1. O sistema operacional registra qual dispositivo requer tempo de processador para que o processador possa se comunicar com o dispositivo sem conflitos
2. O sistema operacional prioriza os processos com base nos sinais de controle que o dispositivo de I/O envia e recebe
3. Se um dispositivo de I/O faz uma solicitação mais crítica do que o que está sendo executado atualmente, o sistema operacional pode interromper o que está sendo executado e trocar para a tarefa mais importante

2.6.4. Parte prática

2.6.4.1. Classe Task

A classe Task precisa ser 'amiga' da classe Thread para esta poder usa-la na Thread::init()

```
□□□□□□□□
```

```
// Task (only used in multitasking configurations) class Task { friend class Thread; // for insert() private: static const bool multitask = Traits<System>::multitask; typedef Thread::Queue Queue; ...
```

A classe Task possui os seguintes atributos, sendo que o Address_Space é compartilhado por todas as Threads que são criadas por aquela Task.

```
private: Address_Space * _as; Segment * _cs; Segment * _ds; Log_Addr _code; Log_Addr _data; Log_Addr _entry; Thread * _main; Queue _threads; static Task * volatile _current; };
```

2.6.4.2. Construtor

A classe Task possui 4 construtores:
1. O primeiro construtor recebe os segmentos e endereços lógicos e o entry point da função principal, criando uma Thread Main.

```
public: template<typename ... Tn> Task(Segment * cs, Segment * ds, Log_Addr code, Log_Addr data, int (* entry)(Tn ...), Tn ... an) : _as (new (SYSTEM) Address_Space), _cs(cs), _ds(ds), _code(_as->attach(_cs, code)), _data(_as->attach(_ds, data)), _entry(entry) { db<Task>(TRC) << "Task(as=" << _as << ",cs=" << _cs << ",ds=" << _ds << ",entry=" << _entry << ",code=" << _code << ",data=" << _data << ") => " << this << endl; _main = new (SYSTEM) Thread(Thread::Configuration(Thread::READY, Thread::MAIN, this, 0), entry, an ...); }
```

2. O segundo construtor além dos parâmetros mencionados, também recebe uma Thread::Configuration

```
public: ... template<typename ... Tn> Task(const Thread::Configuration & conf, Segment * cs, Segment * ds, Log_Addr code, Log_Addr data, int (* entry)(Tn ...), Tn ... an) : _as (new (SYSTEM) Address_Space), _cs(cs), _ds(ds), _code(_as->attach(_cs, code)), _data(_as->attach(_ds, data)), _entry(entry) { db<Task>(TRC) << "Task(as=" << _as << ",cs=" << _cs << ",ds=" << _ds << ",entry=" << _entry << ",code=" << _code << ",data=" << _data << ") => " << this << endl; _main = new (SYSTEM) Thread(Thread::Configuration(conf.state, conf.criterion, this, 0), entry, an ...); }
```

3. O terceiro construtor recebe apenas a thread corrente, e tem um comportamento semelhante ao fork, copiando os segmentos de dados e de código dentro da própria função.

```
public: ... template<typename ... Tn> Task(Task * task = _current, int (* entry)(Tn ...) = 0, Tn ... an) { // fork-like constructor // Allocate resources _as = new (SYSTEM) Address_Space; _cs = new (SYSTEM) Segment(task->code_segment()->size(), Segment::Flags::APP); _ds = new (SYSTEM) Segment(task->data_segment()->size(), Segment::Flags::APP); _entry = entry ? entry : static_cast<int (*)(Tn ...)>(task->entry()); // Copy segments Log_Addr src_code, src_data; if(task == _current) { src_code = task->code(); src_data = task->data(); } else { src_code = _current->address_space()->attach(task->code_segment()); src_data = _current->address_space()->attach(task->data_segment()); } Log_Addr dst_code = _current->address_space()->attach(_cs); Log_Addr dst_data = _current->address_space()->attach(_ds); memcpy(dst_code, src_code, task->code_segment()->size()); memcpy(dst_data, src_data, task->data_segment()->size());
```

```
_current->address_space()->detach(_cs); _current->address_space()->detach(_ds); if(task != _current) { _current->address_space()->detach(task->code_segment()); _current->address_space()->detach(task->data_segment()); } // Map segments _code = _as->attach(_cs, task->code()); _data = _as->attach(_ds, task->data()); db<Task>(TRC) << "Task(as=" << _as << ",cs=" << _cs << ",ds=" << _ds << ",entry=" << _entry << ",code=" << _code << ",data=" << _data << ") => " << this << endl; // Create the task's main thread _main = new (SYSTEM) Thread(Thread::Configuration(Thread::READY, Thread::MAIN, this, 0), static_cast<int (*)(Tn ...)>(_entry), an ...); }
```

4. O último é usado por Thread::init(), sendo o único que recebe Address_Space como param.

```
protected: // This constructor is only used by Thread::init() template<typename ... Tn> Task(Address_Space * as, Segment * cs, Segment * ds, Log_Addr code, Log_Addr data, int (* entry)(Tn ...), Tn ... an) : _as(as), _cs(cs), _ds(ds), _code(code), _data(data), _entry(entry) { db<Task, Init>(TRC) << "Task(as=" << _as << ",cs=" << _cs << ",ds=" << _ds << ",code=" << _code << ",data=" << _data << ",entry=" << _entry << ") => " << this << endl; _current = this; activate(); _main = new (SYSTEM) Thread(Thread::Configuration(Thread::RUNNING, Thread::MAIN, this, 0), entry, an ...); }
```

2.6.4.3. Manipulação das threads

```
private: ... void insert(Thread * t) { _threads.insert(new (SYSTEM) Queue::Element(t)); } void remove(Thread * t) { Queue::Element * el = _threads.remove(t); if(el) delete el; } ...
```

2.6.4.4. 2.4 Destrutor da classe

Por fim, para a correta desalocação dos recursos, removemos todas as threads da fila e as deletamos. Então, é deletado o address space da Task liberando-o.

```
Task::~Task() { db<Task>(TRC) << "~Task(this=" << this << ")" << endl; while(!_threads.empty()) delete _threads.remove()->object(); delete _as; }
```

2.6.5. Referências

https://science.jrank.org/computer-science/Multitasking_Operating_Systems.html
https://en.wikipedia.org/wiki/Cooperative_multitasking
https://people.cs.ksu.edu//~schmidt/300s05/Lectures/OSNotes/os.html
https://isaaccomputerscience.org/concepts/sys_os_resource_management?examBoard=all&stage=all
https://www.tutorialspoint.com/operating_system/os_io_hardware.htm
https://gitlab.lisha.ufsc.br/epos/ine5424/-/tree/2021_2

2.7. Syscall Security in ARMv8 with AArch64

2.7.1. Introdução

System calls, ou syscalls, são a maneira pela qual um programa qualquer pode requisitar uma funcionalidade do kernel de um sistema operacional, como controle de hardware (HD, camera), criação de um novo processo ou realizar comunicação com alguma funcionalidade do sistema operacional (escalonamento, memory management) 11.

Syscalls são necessárias pois, na maioria das arquiteturas, não é todo programa que tem acesso às funções citadas anteriormente por questões de segurança, sendo necessário requisitar ao kernel que as execute. O sistema operacional possui o nível mais alto de privilégio, e fornece uma interface para que programas com níveis mais baixos acessem suas funcionalidades através de syscalls.

2.7.2. Exceções

Uma exceção é um sinal gerado para alterar o fluxo de processamento de um software. Alguns tipos de exceção são: eventos de debug; uma instrução indefinida detectada; interrupções. Exceções são divididas entre as exceções síncronas e exceções assíncronas.

2.7.2.1. Exceções Síncronas

As principais causas de exceções síncronas são:

- Tentativa de executar uma instrução undefined;

*Uso stack pointer desalinhado;
*Tentativa de executar uma instrução com o program counter desalinhado;
* Instruções SVC, HVC ou SMC

A princípio, uma única instrução qualquer pode gerar uma série de exceções síncronas diferentes, entre os da instrução, sua decodificação e eventual execução. Por conta disso, toda exceção síncrona possui uma prioridade diferente, onde 1 é a prioridade mais alta.

2.7.2.2. Exceções Assíncronas

Na arquitetura ARMv8, exceções assíncronas são chamadas de interrupções. As interrupções por sua vez são separadas em duas categorias: físicas e virtuais. Interrupções físicas são sinais enviados ao Processing Element (PE, termo utilizado para se referir a um Core) de fora do PE, como erros de sistema. Interrupções virtuais são interrupções que o software em execução no EL2 pode ativar e tornar pendentes.

2.7.2.3. Níveis de Exceção

A arquitetura ARMv8 define 4 níveis diferentes de exceção, de 0 a 3, com níveis ascendentes de privilégios. O nível EL0 é chamado de nível sem privilégios; é o modo padrão de aplicativos em modo de usuário. O nível EL1 é o modo supervisor, o nível de privilégio do kernel e de funções associadas. Os sistemas com virtualização em hardware introduzem o nível EL2, o modo de \textit{hypervisor}, dedicado a hipervisores e Virtual Machine Monitors (VMMs). Por fim, o EL3 é o nível secure monitor, que permite a troca de estado de segurança entre os estados Seguro e Não-seguro.

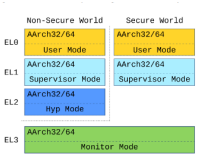
Os únicos níveis que precisam necessariamente ser implementados são os níveis EL0 e EL1, e os níveis implementados não precisam ser contíguos (i. e., é possível implementar apenas os níveis EL0, EL1 e EL3). O nível EL3 é o único que consegue trocar o estado de segurança, logo, não implementá-lo significa não ter acesso a qualquer estado de segurança. Similarmente, não implementar o nível EL2 resulta em não ter acesso a muitas das funcionalidades para virtualização.

2.7.2.4. Estados de Segurança

O nível EL3 permite a troca de estado entre os estados Seguro e Não-seguro. Esses estados definem

o grau de permissão de acesso a endereços físicos de memória que o PE tem acesso:

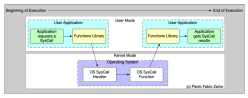
- Estado Seguro: Nesse estado o PE consegue acessar tanto os endereços de memória Seguros quanto os Não-seguros.
- Estado Não-seguro: Nesse estado o PE consegue acessar apenas os endereços de memória Não-seguros.



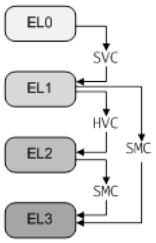
2.7.3. System Calls

Algumas instruções ou funções do sistema só podem ser efetuadas em um nível específico de exceção. Se o código executando em um nível de exceção mais baixo precisa efetuar uma operação privilegiada, por exemplo, quando uma aplicação pede uma funcionalidade do kernel, é necessário que a aplicação gere uma system call, uma exceção síncrona realizada com o intuito de ocorrer a troca de contexto para o nível de exceção desejado e conseguinte execução de função privilegiada pelo nível superior de exceção.

Tradicionalmente, as syscalls são implementadas por meio do processo mostrado na Figura 2: o processo do usuário coloca os dados em um local pré-determinado — geralmente em registradores ou na pilha — e então dispara um mecanismo específico para fazer com que o kernel assuma o controle da execução do processador (por meio de uma interrupção ou uma instrução específica).



Na arquitetura ARMv8, as syscalls são executadas através de instruções específicas de acordo com o nível de privilégio que se pretende usar. Os parâmetros da syscall podem ser passados por registradores ou codificados na própria syscall.2 Uma system call pode ser gerada pela execução de uma instrução SVC, HVC ou SMC. Por padrão, uma instrução SVC gera uma exceção síncrona no nível EL1, permitindo que um aplicativo rodando no nível EL0 tenha acesso ao Kernel. Caso a implementação inclua o nível EL2, a instrução HVC pode ser usada para fazer uma chamada ao Hypervisor.



Abaixo é demonstrado como realizar uma chamada da syscall write() para escrever o tradicional “Hello world'' na tela 8.

/* Our application's entry point. */ .data /* Data segment: define our message string and calculate its length. */ msg: .ascii "Hello, ARM64!\n" len = . - msg .text /* Our application's entry point. */ .globl _start _start: /* syscall write(int fd, const void *buf, size_t count) */ mov x0, #1 /* fd := STDOUT_FILENO */ ldr x1, =msg /* buf := msg */ ldr x2, =len /* count := len */ mov w8, #64 /* write is syscall #64 */ svc #0 /* invoke syscall */ /* syscall exit(int status) */ mov x0, #0 /* status := 0 */ mov w8, #93 /* exit is syscall #1 */ svc #0 /* invoke syscall */

Para a execução do código acima, é feita a compilação utilizando um cross-compiler:

aarch64-linux-gnu-as -o hello.o hello.s aarch64-linux-gnu-ld -s -o hello hello.o qemu-aarch64 ./hello

Para realizar uma syscall com ARMv8, os argumentos são guardados nos registradores $x0 - $x7, e o número da função no registrador $x8. A função write() é representada pelo código 4 — cada função possui um número fixo e pré-definido pela arquitetura. Por fim, a instrução svc $0 invoca a syscall.

De maneira similar podemos observar uma chamada da syscall exit(), que armazena o status no registrador $x0 e possui código 1.

2.7.4. Problemas de segurança do kernel e syscalls

O kernel é responsável por organizar recursos e escalonar os outros processos do sistema. Ele tem acesso direto ao hardware, o que é permitido pois ele roda no modo privilegiado EL1. O kernel também monitora as aplicações em modo de usuário (EL0) em execução e suas alocações de memória e comunicações com o hardware.

Devido ao seu acesso irrestrito à memória e modo privilegiado, o kernel é crítico ao funcionamento correto do sistema e também pode ser alvo de ataques, logo, é fundamental que o kernel seja o mais seguro e livre de erros possível. Contudo, para garantir a segurança do kernel, também é necessário garantir a segurança da interface de system calls, que conecta o nível menos privilegiado EL0 ao nível supervisor EL1. Isso acontece pois, mesmo que a compilação individual do kernel e do espaço de usuário garantam segurança, o fato deles não serem compilados em conjunto significa que essas garantias podem ser quebradas na interface de comunicação entre eles 10.

2.7.4.1. Corrupção de memória

A corrupção de memória acontece quando um programa ganha acesso à memória que ele não deveria poder ter acesso, dando a ele a habilidade de causar comportamento inesperado no sistema. É possível violar a memória de forma espacial ou temporal.

Em C++, a linguagem usada para escrita do EPOS, pode ser difícil alcançar segurança de memória completa. Ponteiros para objetos individuais podem ser confundidos facilmente com arrays, e eles não precisam ter limites de tamanho ou deslocamento definidos na instanciação, então eles podem apontar para qualquer lugar sem ser claro se tal referência é uma violação. Além disso, a segurança temporal se torna difícil em lugares onde a gerência de memória é manual 10.

2.7.4.1.1. Violação espacial

Uma violação espacial de memória ocorre quando um ponteiro é usado para acessar uma região de

memória que ele não foi definido para apontar 7. Por exemplo, se existe um ponteiro para um buffer em memória, aconteceria uma violação se esse ponteiro pudesse acessar ou sobrescrever dados de um buffer adjacente, como acontece em um buffer overflow, em que são escritos mais dados dentro de um buffer do que ele consegue comportar, fazendo com que os dados sobrando *transbordem* para endereços de memória adjacentes. Isso pode ocorrer, por exemplo, em uma system call de escrita que recebe um ponteiro e o seu tamanho, e o tamanho do ponteiro passado como argumento não corresponde ao tamanho real do ponteiro, e também não há checagem sobre esse tipo de discrepância na implementação da syscall.

Se o buffer onde ocorre o overflow for uma variável temporária de uma função, ele possivelmente será armazenado na stack, que também armazena retornos de funções. Dessa forma, com um overflow na stack causado por, por exemplo, uma syscall write() sem checagens de parâmetro corretas, é possível mudar o endereço de retorno de uma função ao sobrescrever o valor original devido à violação de memória causada pelo overflow, o que faz com que seja possível redirecionar o fluxo de um programa de forma maliciosa ou no mínimo causar uma falha de segmentação.

2.7.4.1.2. Violação temporal

Uma violação temporal ocorre quando um ponteiro é usado fora do tempo de vida que foi originalmente definido, como no caso de um ponteiro ser desreferenciado depois de ter sido liberado (use-after-free). Um ponteiro desses é chamado de ponteiro pendente/\textit{dangling pointer} e seu uso causa comportamento não definido, já que os dados para os quais ele apontava podem não estar mais naquele endereço de memória, ou pior, o endereço pode ter sido usado para alocar outra estrutura de dados, permitindo que o ponteiro antigo seja usado para corromper dados que agora têm outro propósito 7.

2.7.4.2. Desreferenciação de ponteiros corrompidos/não validados

Essa categoria cobre qualquer situação em que um ponteiro é usado enquanto o seu conteúdo foi corrompido ou não foi validado o suficiente. Um ponteiro corrompido normalmente é consequência de algum outro erro, como um buffer overflow, que pode corromper um ou mais bytes do conteúdo do ponteiro, como descrito anteriormente. Esse tipo de situação dá a um possível atacante mais controle sobre o conteúdo da variável, o que leva diretamente a um ataque mais confiável 7.

Problemas causados por um ponteiro não-validado fazem mais sentido em um espaço de endereçamento combinado para o kernel e espaço de usuário, como é o caso da arquitetura ARMv8-A. Nela, o espaço de endereçamento do nível EL1 fica em endereços acima do espaço de endereçamento do nível EL0, e os registradores TCR_EL0 e TCR_EL1 são usados para definir o tamanho e portanto endereço limite desses espaços de endereçamento 3. Dessa forma, funções internas do kernel podem usar os endereços limite dos espaços de endereçamento para decidir se um ponteiro específico aponta para o kernel ou para o espaço de usuário. No primeiro caso, normalmente são feitas menos checagens devido ao nível de privilégio maior, enquanto no segundo caso é preciso de mais cuidado ao acessar o endereço. Se essa checagem não estiver presente ou for aplicada incorretamente, um endereço de espaço de usuário pode ser desreferenciado sem o controle necessário 7.

2.7.4.3. Condições de corrida

Uma condição de corrida acontece quando dois ou mais atores (por exemplo, processos ou threads) querem realizar uma ação sob o mesmo objeto ``ao mesmo tempo'' e o resultado será diferente dependendo da ordem que cada ação ocorrer. Para que a condição de corrida ocorra, os atores

precisam executar suas ações ou paralelamente, o que ocorre em um processadores de múltiplos cores, ou, pelo menos, concorrente,de forma intercalada uma com a outra, o que ocorre dentro de um único core devido à alternância de tarefas 7. Ela pode ocorrer, por exemplo, devido a duas threads escrevendo em uma posição de memória compartilhada ao mesmo tempo.

Considerando o sistema operacional, esse tipo de situação não é desejado pois condições de corrida podem causar comportamento inesperado em caminhos fundamentais para o funcionamento correto do sistema. Para preveni-las, é preciso garantir algum tipo de sincronização entre os vários atores, mas falhas nessa sincronização ainda permitem condições de corrida e podem causar problemas acidentais ou serem exploradas por programas maliciosos, como, por exemplo, sobrescrever o código de um programa enquanto ele está sendo executado por uma syscall execve(). Ataques de condição de corrida típicos envolvem abrir um arquivo, validar um arquivo, executar uma subrotina, checar uma senha ou verificar um nome de usuário 5.

2.7.5. Referências

[[1]] ARM Holdings. Arm architecture reference manual armv8, for armv8 a architecture profile. https://developer.arm.com/documentation/ddi0487/ea, 2019. Online; accessed 29-November-2021.

[[2]] ARM Holdings. Arm cortex-a series programmer’s guide for armv8-a system calls. https://developer.arm.com/documentation/den0024/a/AArch64-Exception-Handling/Synchronous-and-asynchronous-exceptions/System-calls, 2019. Online; accessed 29-November-2021.

[[3]] ARM Holdings. Learn the architecture: Aarch64 memory management address spaces in aarch64. https://developer.arm.com/documentation/101811/0101/Address-spaces-in-AArch64, 2019. Online; accessed 29-November-2021.

[[4]] ARM Holdings. Aarch64 exception and interrupt handling. https://developer.arm.com/documentation/100933/0100/ Synchronous-and-asynchronous-exceptions, 2021. Online; accessed 05-December-2021.

[[5]] Tanjila Farah et. al. Study of race condition: A privilege escalation vulnerability. In Proceedings of the 21st World Multi-Conference on Systemics, Cybernetics and Informatics (WMSCI 2017), 2017.

[[6]] Paolo Zaino. Operating systems: System calls (part i). https://paolozaino.wordpress.com/2013/05/22/system-calls-part-i/, 2013.Online; accessed 29-November-2021

[[7]] Enrico Perla and Massimiliano Oldani. A Guide to Kernel Exploitation: Attacking the Core. Elsevier, 2011.

[[8]] Peter Nelson . ’hello world!’ in arm64 assembly. https://peterdn.com/post/2019/02/03/hello-world-in-arm-assembly/, 2019. Online; accessed 04-December-2021.

[[9]] Sergej Proskurin, Tamas Lengyel, Marius Momeu, Claudia Eckert, and Apostolis Zarras. Hiding in the shadows: Empowering arm for stealthy virtual machine introspection. In Proceedings of the 34th Annual Computer Security Applications Conference, pages 407–417, 2018.

[[10]] Jakob H. Weisblat. Improving security at the system-call boundary in a type-safe operating system. https://shorturl.at/oDENV, 2018. Online; accessed 29-November-2021.

[[11]] Wikipedia contributors. System call — Wikipedia, the free encyclopedia.

https://en.wikipedia.org/w/index.php?title=System_call&oldid=1056437221, 2021. Online; accessed 29-November-2021.

2.8. MMU for Paging

2.8.1. 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.

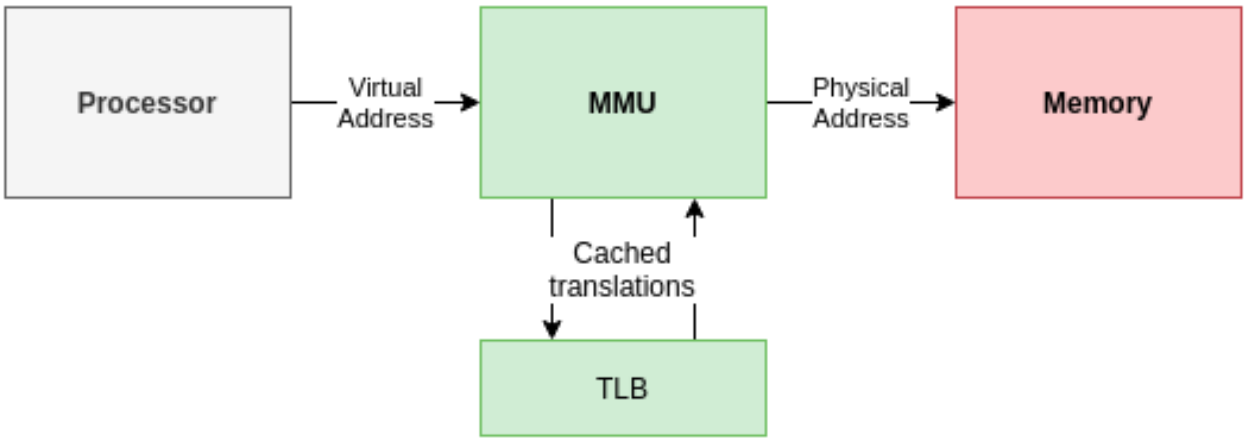
- **Page Table/Page Directory** - Um vetor de registros usados para tradução de endereços virtuais para físicos. As tabelas de páginas primárias(aquelas pelas quais o processo de tradução de endereço se inicia) podem ser chamadas de tabelas de diretório(Page directory).

- **ASID(Address Space Identifier)** - Usado para identificação de páginas de um processo específico

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

2.8.2. 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.



2.8.3. Virtual Adress - ARMv8

- **ARMv8-A** padrão se dispõe de um virtual address de 48 bits
- **ARMv8.2-LVA** se dispõe de um virtual address de 52 bits

Além disso ambos sistemas possuem suporte para 1 ou 2 ranges de virtual address.

Único: 48 bits ou 52 bits (ARMv8.2-LVA)

0x0000000000000000 to 0x0000FFFFFFFFFFFF
ou
0x0000000000000000 to 0x000FFFFFFFFFFFFF

Dois Sub Ranges, um no topo e outro no fundo:

0x0000000000000000 to 0x0000FFFFFFFFFFFF (48 bits)
0xFFFF000000000000 to 0xFFFFFFFFFFFFFFFF
ou
0x0000000000000000 to 0x000FFFFFFFFFFFFF. (52 bits)
0xFFF0000000000000 to 0xFFFFFFFFFFFFFFFF.

2.8.4. MMU - ARMv8

No ARMv8 possuímos dois regimes bem definidos de tradução de endereços que serão utilizados com base nos níveis de exceção habilitados e o estado de segurança.
1) Virtual Address -> Physical Address
Virtual Address -> Intermediate Physical Adress -> Physical Address

1) VA -> PA

- O regime de tradução Secure EL1&0, quando EL2 está desativado.
- O regime de tradução Non-Secure EL2&0.
- O regime de tradução Non-Secure EL2.
- O regime de tradução Secure EL2.
- O regime de tradução Secure EL3.

2) VA -> IPA -> PA

- O regime de tradução Secure EL1&0, quando EL2 está ativado.

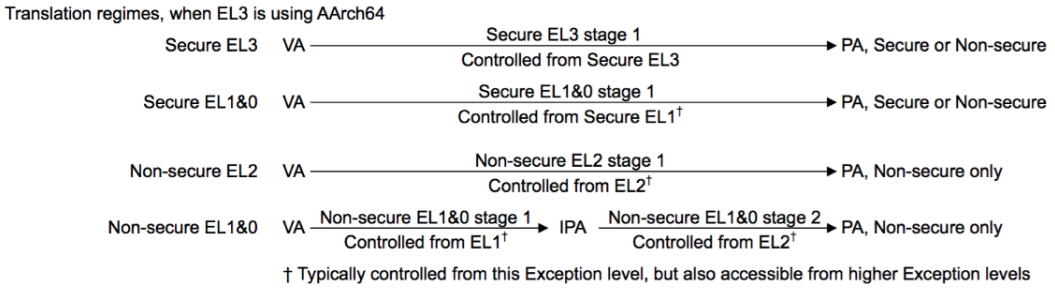


Figure D4-2 VMSAv8 AArch64 translation regimes, translation stages, and associated controls

2.8.4.1. Níveis de Exceção

A arquitetura Armv8-A define níveis de exceção EL0 até EL3, onde quanto mais alto o número maior é o privilégio de execução.

- EL0 é chamado de execução sem privilégios
- EL3 provê suporte para troca entre os dois estados de sistema: Estado Seguro e Estado Não Seguro.
- Em nível de usuário, o usuário envia um VA, que é transformado para um IPA(Intermediate Physical Adress) e o Hypervisor assume o IPA e encontra o respectivo PA.

Obs:

- Na implementação não é obrigatório a utilização de todos os níveis;
- Os níveis de execução mais altos podem usar os recursos dos mais baixos;
- EL2 e EL3 são opcionais: desnecessários caso não precise de segurança ou virtualização.

2.8.4.2. Estado de Segurança

- ARMv8-A define dois tipos de estado de segurança: Seguro e Não Seguro (Normal).
- E isso também define dois endereços de memória físicos: Seguro e Não Seguro.
- Em teoria os dois são completamente separados.

- No entanto, a maioria dos sistemas tratam esse estado como um atributo para o controle de acessos.
- O mundo normal só consegue acessar o endereçamento físico não seguro.
- O mundo seguro pode acessar ambos espaços de armazenamento controlado por translation tables.

2.8.4.3. Registradores

TTBR_ELx (Translation Table Base Register)
Indica o começo da da primeira translation table necessária para o mapeamento do VA para o PA.

TTBR0_ELx TTBR1_ELx
-> Endereços baixos da memória (usuário) -> Endereços altos da memória (kernel)

SCTLR_ELn (System Control Register)
Controla funcionalidades da arquitetura, como a MMU, caches e verificação de alinhamento de memória.
Tem alto nível de controle do sistema, incluindo sua memória.

HCR_EL2 (Hypervisor Configuration Register)
Controla as configurações de virtualização e as exceções para EL2

SCR_EL3 (Secure Configuration Register)
Controla o estado seguro e as exceções do EL3

ID_AA64MMFR0_EL1, AArch64 Memory Model Feature Register 0
Provê informações sobre o modelo de memória implementado e suporte ao gerenciamento de memória do AArch64

SPSR_ELx, Saved Program Status Register
Guarda o estado do processo salvo quando uma exceção é levada ao ELx.

Exception Link Registers (ELR_ELx)
Guarda o endereço de retorno da exceção.

2.8.4.3.1. Memory Attribute Indirection Register (MAIR_ELx)

Provê as codificações de atributos de memória correspondentes aos possíveis valores de AttrIndex em uma entrada de tabela da pagle table.
Onde Attr<n> é dos bits 8n+7:8n, com n 7 até 0

Attr	Meaning
0b0000dd00	Device memory. See encoding of 'dd' for the type of Device memory.
0b0000dd01	If FEAT_XS is implemented: Device memory with the XS attribute set to 0. See encoding of 'dd' for the type of Device memory. Otherwise,UNPREDICTABLE.
0b0000dd1x	UNPREDICTABLE.
0boooooiiii, (oooo != 0000 and iiii != 0000)	Normal memory. See encoding of 'oooo' and 'iiii' for the type of Normal Memory.
0b01000000	If FEAT_XS is implemented: Normal Inner Non-cacheable, Outer Non-cacheable memory with the XS attribute set to 0. Otherwise,UNPREDICTABLE.
0b10100000	If FEAT_XS is implemented: Normal Inner Write-through Cacheable, Outer Write-through Cacheable, Read-Allocate, No-Write Allocate, Non-transient memory with the XS attribute set to 0. Otherwise,UNPREDICTABLE.
0b11110000	If FEAT_MTE2 is implemented: Tagged Normal Inner Write-Back, Outer Write-Back, Read-Allocate, Write-Allocate Non-transient memory. Otherwise,UNPREDICTABLE.
0bxxxx0000, (xxxx != 0000, xxxx != 0100, xxxx != 1010, xxxx != 1111)	UNPREDICTABLE.

R or W	Meaning
0b0	No Allocate
0b1	Allocate

'oooo'	Meaning
0b0000	See encoding of Attr
0b00RW, RW not 0b00	Normal memory, Outer Write-Through Transient
0b0100	Normal memory, Outer Non-cacheable
0b01RW, RW not 0b00	Normal memory, Outer Write-Back Transient
0b10RW	Normal memory, Outer Write-Through Non-transient
0b11RW	Normal memory, Outer Write-Back Non-transient

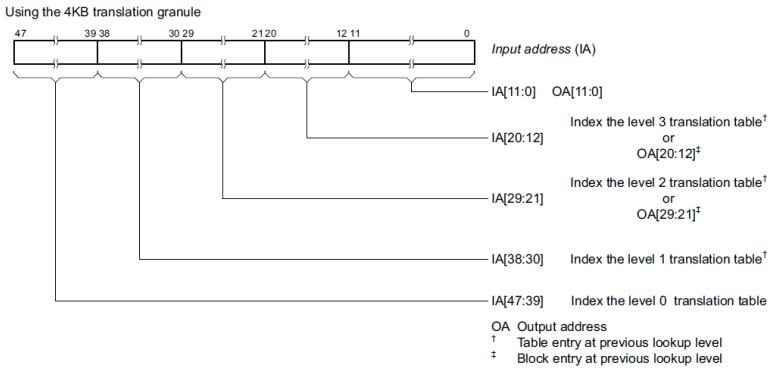
dd	Meaning
0b00	Device-nGnRnE memory
0b01	Device-nGnRE memory
0b10	Device-nGRE memory
0b11	Device-GRE memory

Glossário:

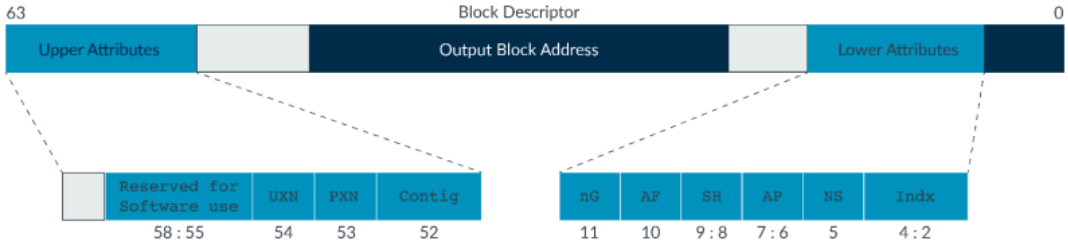
- Gathering: Possível juntar vários acessos de memória mesmo tipo e região de memória em uma unica transação.
- Early write acknowledgement: Um acesso é dado como completo antes de chegar ao destinatário, sendo considerado completo quando fica visível para os outros observadores
- Device-nGnRnE: Dispositivo sem gathering, e sem reordenamento, e No Early write

acknowledgement.
- Device-nGnRE: Dispositivo sem gathering, e sem reordenamento, com Early Write Acknowledgement.
- Device-nGRE: Dispositivo sem gathering, com reordenamento, com Early Write Acknowledgement.
- Device-GRE: Dispositivo com gathering, com reordenamento, com Early Write Acknowledgement.
- Non-shareable: Uma região de memória normal que não possui o atributo Shareable.
- Inner Shareable: Uma região de memória normal possui o atributo Shareable, mas não possui Outer Shareable.
- Outer Shareable: Uma região de memória normal possui os atributos Shareable, Outer Shareable.

2.8.4.4. Estrutura de Endereço Virtual



2.8.4.5. Estrutura de uma Entrada da Tabela de Páginas

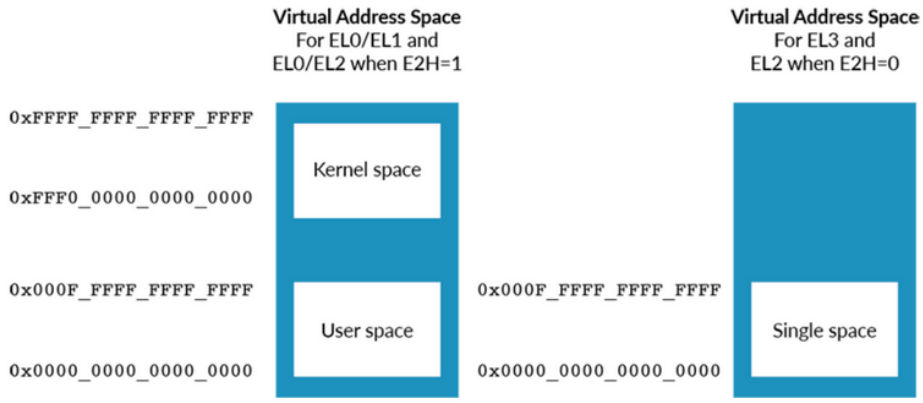


2.8.5. Exemplo de tradução de endereço no ARMv8

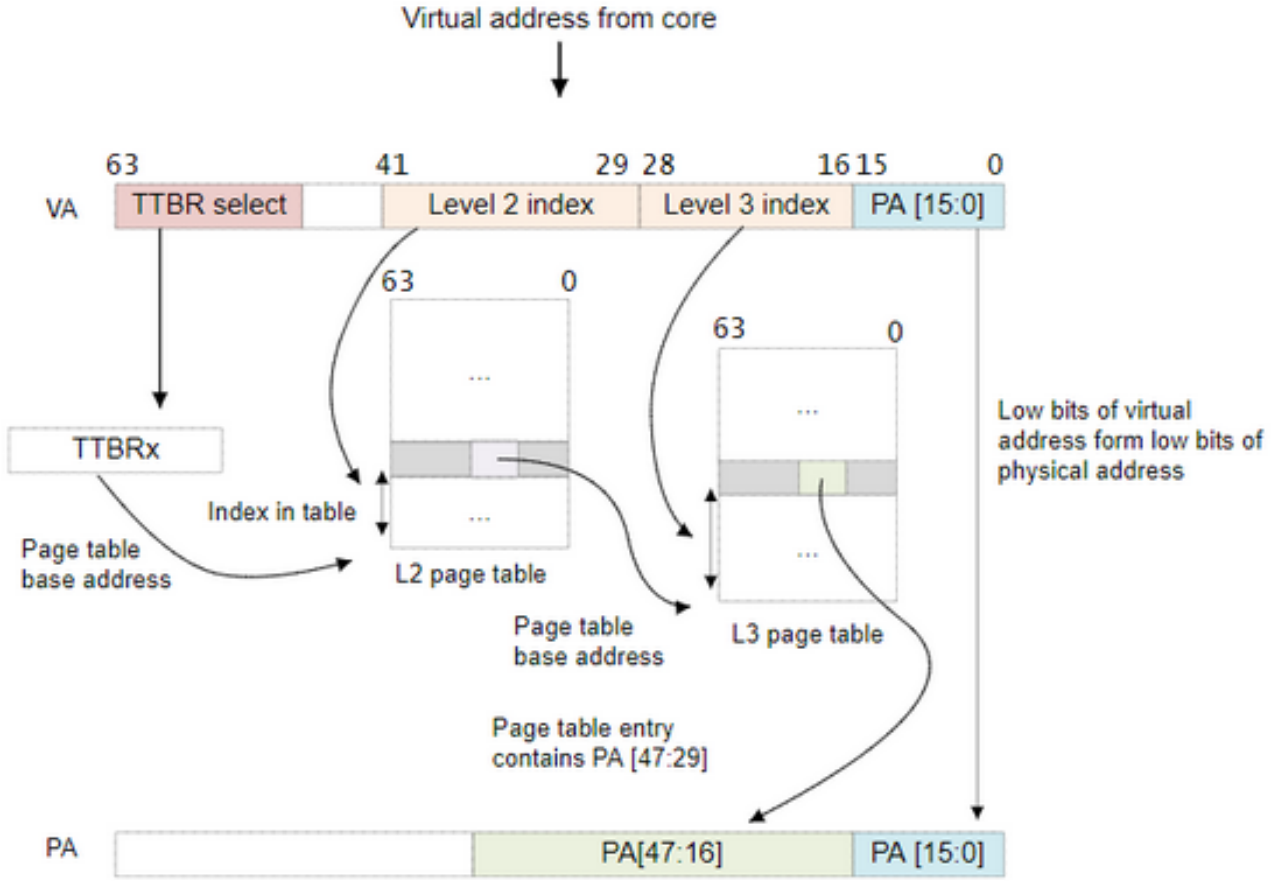
Em uma página de 64K, considerando uma tradução com regime Virtual Address para Physical Address e 2 níveis de Page Tables, temos:
1. Se VA63:42 = 1 (endereços mais altos na memória, que são endereçados pelo kernel) então TTBR1_ELn é utilizado como endereço base para a page table nível 1. Caso VA63:42 seja 0 (endereços mais baixos da memória, que são endereçados pelo usuário), é utilizado o TTBR0_ELn.
2. A page table é acessada na entrada especificada pelo campo "level 1 index". Que indica um endereço base de uma nova tabela de nivel maior.
3. MMU verifica se o endereço é válido e se seu acesso é permitido.
4. Caso verdadeiro: a nova tabela é consultada na entrada identificada pelo campo "level 2 index" do VA.
5. MMU verifica se é valido e se seu acesso é permitido.
6. Caso verdadeiro, a entrada é utilizada para se referir a página de 64KB solicitada.

7. O endereço da página identificada é somado ao campo offset do VA e é retornado, junto com informações adicionais das entradas da page table.

Os bits mais significativos (No caso 63:42) devem ser todos 0s ou 1s, caso contrário é acionada uma falta de página. Além disso, esses mesmos bits definem se é de usuário ou de kernel:
0-> Usuário
1-> Kernel



2.8.5.1. Tradução de Endereço Virtual com 3 níveis de tabela de página



2.8.6. Swap Out

2.8.6.1. Exemplo Assembly: Swap Out

STR X11, [X1] ; updates the translation table for the region being paged out DSB ISH ; ensures visibility of the update to translation table walks TLBI VAE1IS, X10 ; invalidates the old entry DSB ISH ; ensures completion of the invalidation on all PEs ISB ; ensures visibility of the invalidation BL SaveMemoryPageToBackingStore BL LoadMemoryFromBackingStore DSB ISH ; ensures completion of the memory transfer (this could be part of; LoadMemoryFromBackingStore) IC IALLUIS ; also invalidates the branch predictor DSB ISH ; ensures completion of the instruction cache; and branch predictor invalidation STR X9, [X1] ; creates a new translation table entry with a new mapping DSB ISH ; ensures visibility of the new translation table mapping ISB ; ensures synchronisation of this instruction stream

2.8.7. Tipos de Falhas na MMU - ARMv8

A MMU pode apresentar as seguintes falhas:

- Falha e alinhamento em um acesso a dados;
- Falha de permissão;
- Falha de tradução;
- Falha de tamanho de endereço;
- Síncrono: Abortamento externo em uma caminhada na mesa de tradução;
- Falha de sinalizador de acesso;
- Abortar conflito de TLB.

Caso ocorra alguma falta/exceção existem os Fault Status Registers (XFSR) que estão disponíveis para ajudar os fault handlers a identificar a falha.

2.8.8. Ativação da MMU no ARMv8

□□□□□□□□

.equ Mode_USR, 0x10 .equ AArch32_Mode_USR, 0x10 .equ AArch32_Mode_FIQ, 0x11 .equ AArch32_Mode_IRQ, 0x12 .equ AArch32_Mode_SVC, 0x13 .equ AArch32_Mode_ABT, 0x17 .equ AArch32_Mode_UNDEF, 0x1B .equ AArch32_Mode_SYS, 0x1F .equ AArch32_Mode_HYP, 0x1A .equ AArch32_Mode_MON, 0x16 .equ AArch64_EL2_SP2, 0x09 // EL2h .equ AArch64_EL2_SP0, 0x08 // EL2t .equ AArch64_EL1_SP1, 0x05 // EL1h .equ AArch64_EL1_SP0, 0x04 // EL1t .equ AArch64_EL0_SP0, 0x00 .equ AArch32_State_Thumb, 0x20 .equ AArch32_State_ARM, 0x00 .equ TT_S1_TABLE, 0x00000000000000003 // NSTable=0, PXNTable=0, UXNTable=0, APTable=0 // TT block entries templates (L1 and L2, NOT L3) // Assuming table contents: // 0 = b01000100 = Normal, Inner/Outer Non-Cacheable // 1 = b11111111 = Normal, Inner/Outer WB/WA/RA // 2 = b00000000 = Device-nGnRnE .equ TT_S1_FAULT, 0x0 .equ TT_S1_NORMAL_NO_CACHE, 0x00000000000000401 // Index = 0, AF=1 .equ TT_S1_NORMAL_WBWA, 0x00000000000000405 // Index = 1, AF=1 .equ TT_S1_DEVICE_nGnRnE, 0x00600000000000409 // Index = 2, AF=1, PXN=1, UXN=1 .equ TT_S1_UXN, (1 << 54) .equ TT_S1_PXN, (1 << 53) .equ TT_S1_nG, (1 << 11) .equ TT_S1_NS, (1 << 5) .equ TT_S1_NON_SHARED, (0 << 8) // Non-shareable .equ TT_S1_INNER_SHARED, (3 << 8) // Inner-shareable .equ TT_S1_OUTER_SHARED, (2 << 8) // Outer-shareable .equ TT_S1_PRIV_RW, (0x0) .equ TT_S1_PRIV_RO, (0x2 << 6) .equ TT_S1_USER_RW, (0x1 << 6) .equ TT_S1_USER_RO, (0x3 << 6)

□□□□□□□□

// . LDR x0, =tt_l1_base // Get address of level 1 for TTBR0_EL3 MSR TTBR0_EL3, x0 // Set TTBR0_EL3 (NOTE: There is no TTBR1 at EL3) // Set up memory attributes // 0 = b01000100 = Normal, Inner/Outer Non-Cacheable // 1 = b11111111 = Normal, Inner/Outer WB/WA/RA // 2 = b00000000 = Device-nGnRnE MOV x0, #0x000000000000FF44 MSR MAIR_EL3, x0 // Set up TCR_EL3 MOV x0, #32 // T0SZ=0b011001 Limits VA space to 32 bits, translation starts @ l1 ORR

x0, x0, #(0x1 << 8) // IGRN0=0b01 Walks to TTBR0 are Inner WB/WA ORR x0, x0, #(0x1 << 10) // OGRN0=0b01 Walks to TTBR0 are Outer WB/WA ORR x0, x0, #(0x3 << 12) // SH0=0b11 Inner Shareable // TBI0=0b0 Top byte not ignored // TG0=0b00 4KB granule // IPS=0 32-bit PA space MSR TCR_EL3, x0 // Ensure changes to system register are visible before MMU enabled ISB // Invalidate TLBs TLBI ALLE3 DSB SY ISB LDR x1, =tt_l1_base // Address of L1 table // [0]: 0x0000,0000 - 0x3FFF,FFFF LDR x2, =tt_l2_base // Get address of L2 table LDR x0, =TT_S1_TABLE // Entry template for pointer to next level table ORR x0, x0, x2 // Combine template with L2 table Base address STR x0, [x1] // [1]: 0x4000,0000 - 0x7FFF,FFFF LDR x0, =TT_S1_DEVICE_nGnRnE // Entry template // AP=0, RW ORR x0, x0, #0x40000000 // 'OR' template with base physical address STR x0, [x1, #8] // [2]: 0x8000,0000 - 0xBFFF,FFFF (DRAM on the VE and Base Platform) LDR x0, =TT_S1_NORMAL_WBWA // Entry template ORR x0, x0, #TT_S1_INNER_SHARED // 'OR' with inner-shareable attribute ORR x0, x0, #TT_S1_NS // 'OR' with NS==1 ORR x0, x0, #TT_S1_PXN // 'OR' with XN==1 // AP=0, RW ORR x0, x0, #0x80000000 // 'OR' template with base physical address STR x0, [x1, #16] // [3]: 0xC000,0000 - 0xFFFF,FFFF (DRAM on the VE and Base Platform) LDR x0, =TT_S1_NORMAL_WBWA // Entry template ORR x0, x0, #TT_S1_INNER_SHARED // 'OR' with inner-shareable attribute ORR x0, x0, #TT_S1_NS // 'OR' with NS==1 ORR x0, x0, #TT_S1_PXN // 'OR' with XN==1 // AP=0, RW ORR x0, x0, #0xC0000000 // 'OR' template with base physical address STR x0, [x1, #24] // Generate required entries LDR x1, =tt_l2_base // Address of L1 table // [0..31]: 0x0000,0000 - 0x03FF,FFFF (Trusted Boot ROM) LDR x0, =TT_S1_NORMAL_WBWA // Entry template ORR x0, x0, #TT_S1_INNER_SHARED // 'OR' with inner-shareable attribute ORR x0, x0, #TT_S1_PRIV_RO // 'OR' in Read-only ORR x0, x0, xzr // 'OR' template with base physical address MOV x2, #32 1: STR x0, [x1], #8 ADD x0, x0, #0x200000 // Increment the physical address field SUB x2, x2, #1 CBNZ x2, 1b // [32..47]: 0x0400,0000 - 0x05FF,FFFF (Fault) LDR x0, =TT_S1_FAULT // Entry template ORR x0, x0, #0x04000000 // 'OR' template with base physical address MOV x2, #16 1: STR x0, [x1], #8 ADD x0, x0, #0x200000 // Increment the physical address field SUB x2, x2, #1 CBNZ x2, 1b // [48..63]: 0x0600,0000 - 0x07FF,FFFF (Trusted DRAM) LDR x0, =TT_S1_NORMAL_WBWA // Entry template ORR x0, x0, #TT_S1_INNER_SHARED // 'OR' with inner-shareable attribute // RW ORR x0, x0, #0x06000000 // 'OR' template with base physical address MOV x2, #16 1: STR x0, [x1], #8 ADD x0, x0, #0x200000 // Increment the physical address field SUB x2, x2, #1 CBNZ x2, 1b // [64..127]: 0x0800,0000 - 0x0FFF,FFFF (Flash) LDR x0, =TT_S1_NORMAL_WBWA // Entry template ORR x0, x0, #TT_S1_INNER_SHARED // 'OR' with inner-shareable attribute ORR x0, x0, #TT_S1_PRIV_RO // 'OR' in Read-only ORR x0, x0, #TT_S1_NS // 'OR' with NS==1 ORR x0, x0, #TT_S1_PXN // 'OR' with XN==1 ORR x0, x0, #0x08000000 // 'OR' template with base physical address MOV x2, #64 1: STR x0, [x1], #8 ADD x0, x0, #0x200000 // Increment the physical address field SUB x2, x2, #1 CBNZ x2, 1b // [128..511]:0x1000,0000-0x3FFF,FFFF (Fault):initialized with 0 by ".fill" directive at the end of the file DSB SY // Enable MMU MOV x0, #(1 << 0) // M=1 bit Enable the stage 1 MMU ORR x0, x0, #(1 << 2) // C=1 bit Enable data and unified caches ORR x0, x0, #(1 << 12) // I=1 Enable instruction fetches to allocate into unified caches // A=0 Strict alignment checking disabled // SA=0 Stack alignment checking disabled // WXN=0 Write permission does not imply XN // EE=0 EL3 data accesses are little endian MSR SCTLR_EL3, x0 ISB // we are positioning the page tables in a specific section .section ".TT" .align 12 .global tt_l1_base tt_l1_base: .fill 32 , 1 , 0 .align 12 .global tt_l2_base tt_l2_base: .fill 4096 , 1 , 0

2.8.9. Referências

ARM Architecture Reference Manual - Armv8, for Armv8-A architecture profile
ARMv8 Architecture OverviewArquivo
AArch64 - Memory Management Examples
https://developpaper.com/original-armv8-mmu-and-linux-page-table-mapping/

https://developer.arm.com/documentation/ddi0487/ea
https://developer.arm.com/documentation/den0024/a/The-Memory-Management-Unit/Translating-a-Virtual-Address-to-a-Physical-Address
https://stackoverflow.com/questions/64843803/how-this-simple-paging-in-armv8a-works
https://developpaper.com/armv8-a-memory-management/
https://developer.arm.com/documentation/101811/0101/Address-spaces-in-AArch64
https://armv8-ref.codingbelief.com/en/chapter_d4/d42_1_about_the_vmsav8-64_address_translation_syste.html
https://www.youtube.com/watch?v=rh3-62HHkYY&t=1341s

2.9. Task Context Switching (Grupo O)

2.9.1. O que é troca de contexto

Num sistema operacional multitask, múltiplas tarefas (processos e/ou threads) acabam compartilhando uma mesma CPU. Num sistema como esse, os usuários têm a ilusão de que mais de uma tarefa está sendo executada ao mesmo tempo. Na realidade, entretanto, apenas uma delas está sendo executada em um determinado momento por um processador.

Para que essa dinâmica funcione, é preciso que ocorra a troca de contexto (ou context switching) da CPU, de uma tarefa para a outra. Um contexto pode ser definido como um estado computacional e ele precisa ser salvo para que o processo possa continuar a sua execução de onde parou, quando retornar para a CPU.

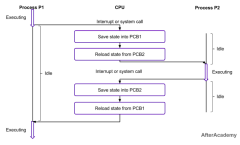
O que exatamente deve ser salvo e restaurado varia entre diferentes sistemas operacionais, mas normalmente uma troca de contexto inclui salvar ou restaurar pelo menos alguns dos seguintes elementos:

- Registradores de propósito geral (X0 a X30);
- Registradores de Advanced SIMD e ponto flutuante (V0 a V31);
- Alguns registradores de status;
- TTBR0_EL1 e TTBR0;
- Registradores de “Thread Process ID” (TPIDxxx);
- “Address Space ID” (ASID).

Vale notar também que uma troca de contexto pode ser disparada em algumas situações, tais como: pausa de uma tarefa para retomada de outra, interrupção devido a uma operação como I/O, ou ainda, na troca entre tarefas de modo usuário e de modo kernel (em alguns sistemas operacionais).

2.9.2. Como acontece uma troca de contexto

O processo de troca de contexto acontece em alguns passos. A imagem abaixo mostra a troca entre os processos P1 e P2.



Pode-se perceber que inicialmente o processo P1 está em estado de execução e o processo P2 em estado de espera. Quando uma das interrupções ocorre, é necessário trocar P1 para o estado de espera e P2 para o estado de execução, isso ocorre na seguinte ordem:

1. O contexto de P1 é salvo no PCB (Process Control Block) do próprio processo.
2. O PCB1 então é movido para a respectiva pilha (pilha de execução, pilha de I/O, pilha de espera, etc)
3. A partir do estado “pronto”, é selecionado um novo processo a ser executado, o processo P2.
4. O PCB2 é atualizado setando o estado do processo para “em execução”. Se esse processo anteriormente já tinha sido executado pela CPU, então pode-se pegar a posição da última instrução executada para continuá-la.
5. De forma similar, se for necessário executar P1 novamente, são repetidos os passos de 1 a 4.

Na maioria dos casos, para a troca de contexto acontecer é necessário pelo menos 2 processos, mas no caso de utilizar o algoritmo Round-Robin é necessário apenas 1 processo.

O Process Control Block (PCB) é uma estrutura de dados utilizada pelo sistema operacional para guardar toda a informação sobre o processo. Ele é criado pelo SO quando no momento de criação de um processo. As informações guardadas no PCB sobre o processo caem em três categorias: identificação do processo, estado do processo, e controle do processo.

Para a identificação, está incluso um ID único para o processo. Em um sistema multi-role-multitasking, também estão incluídos outros dados como identificação do processo pai, entre outras. O mais importante costuma ser o ID do processo, já que ele possibilita encontrar as outras definições que interessam ao processo, como dispositivos I/O e área de memória.

Os dados de estado do processo definem o status de um processo quando ele é suspenso, permitindo que o sistema operacional o reinicie mais tarde. Isso sempre inclui o conteúdo de registradores de CPU de uso geral, a palavra de status do processo da CPU, ponteiros de pilha e quadro, etc. Durante a troca de contexto, o processo em execução é interrompido e outro processo é executado.

Algumas outras informações relevantes para serem utilizadas pelo sistema operacional para administrar o processo incluem:

- Estado de agendamento do processo: o estado do processo no que diz respeito às informações de agendamento. Isso inclui prioridade, tempo de controle da CPU, estado de “ready”, “suspenso”, etc.
- Informações de estruturação de processo: ID’s de processos relevantes ao processo atual de uma maneira funcional. Isso inclui filas, entre outras estruturas de dados.
- Informações de comunicação entre processos: flags, sinais e outras formas de comunicação entre processos independentes;
- Privilégios do processo;
- O número do processo (PID);
- O Program Counter (PC): Aponta para a próxima instrução a ser executada pelo processo;
- Registradores da CPU: Para o processo ser guardado para o estado de execução;
- Informação de gerenciamento de memória;
- Informações contábeis: Quantidade de CPU usada para a execução do processo, tempo limite;
- Informações de I/O.

2.9.3. Considerações sobre performance

A troca de contexto pode causar uma grande sobrecarga da CPU, o que pode afetar o desempenho geral do sistema. Do ponto de vista do usuário, nenhum trabalho útil é realizado pela CPU enquanto o contexto de um processo em execução é salvo e o contexto do próximo processo a executar é recuperado, por isso é de seu interesse que isso ocorra o mais rápido possível.

2.9.3.1. Translation Lookaside Buffer (TLB)

A TLB (Translation Lookaside Buffer) pode ser entendida como um cache para a MMU, ela é utilizada para armazenar as traduções virtuais para físicas. O processo de tradução é muito caro, uma vez que o processador deve ficar acessando a tabela de páginas a todo momento, por isso a maioria dos processadores começaram a armazenar em cache essas traduções.

Além disso, as entradas da translation table contém um bit não global (nG), se este for marcado para uma página específica, ele será associado a uma tarefa ou aplicativo específico; se este estiver marcado como 0, a entrada é global e se aplica a todas as tarefas.

Para as entradas não globais, quando o TLB é atualizado e a entrada é marcada como não global, um valor é armazenado na entrada TLB além das informações de tradução normais.

2.9.3.2. Address Space Identifier (ASID)

Este valor atribuído pelo SO a cada tarefa individual é denominado Address Space ID (ASID). As pesquisas subsequentes da TLB só combinam com tal entrada se o ASID atual corresponder ao ASID que está armazenado nessa entrada.

Isso permite que várias entradas TLB válidas estejam presentes para uma página específica marcada como não global, mas com valores ASID diferentes. Em outras palavras, não precisamos necessariamente liberar os TLBs quando mudamos de contexto. Em AArch64, este valor pode ser especificado como um valor de 8 ou 16 bits, controlado pelo bit TCR_EL1.AS.

Esse valor do ASID atual pode ser especificado em TTBR0_EL1 ou TTBR1_EL1 (o TCR_EL1 que faz essa escolha), mas normalmente fica no primeiro, pois corresponde ao espaço do aplicativo.

Ter o valor atual do ASID armazenado no registro da translation table significa que você pode modificar atomicamente tanto as translation tables quanto o ASID em uma única instrução. Isso simplifica o processo de alteração da tabela e do ASID quando comparado com a arquitetura ARMv7-A.

Uma das etapas necessárias na troca de contexto é garantir que o processo de tradução usando o TLB não seja traduzido para um endereço físico de outro espaço de endereçamento. Para isso, uma das soluções possíveis seria utilizar e depois atualizar o valor do ASID atual, identificando se uma entrada no cache deve ou não ser utilizada.

2.9.4. Trocando de contexto em ARMv8/Raspberry Pi3

2.9.4.1. Modos de processamento do ARM

Alguns dos registradores de leitura e gravação envolvidos na troca de contexto exigem que as operações sejam executadas em um modo privilegiado. Em um modo privilegiado há registradores em banco que permitem uma manipulação mais fácil da stack. Para entrar em um modo privilegiado, um processo em execução no modo de usuário passa por uma interrupção antes de alternar o contexto.

Um exemplo de uma interrupção que pode ser usada para obter um reescalonamento privilegiado é a interrupção do timer de Interrupt Request (IRQ), que traz o processador para o modo IRQ. Além disso, o modo de sistema não possui registradores em banco, e permite atualizar os registradores de stack pointer, entre outros, para o próximo processo do usuário.

2.9.4.2. IRQ

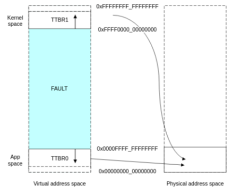
Um IRQ handler é um sinal de hardware enviado ao processador que interrompe temporariamente um programa em execução e permite que um programa especial – um interrupt handler –, seja executado. Interrupções de hardware são usadas para lidar com eventos como o recebimento de dados de um modem ou placa de rede, pressionamentos de tecla ou movimentos do mouse.

2.9.4.3. Gerenciando espaços de endereçamento

Cada tarefa no SO possui o seu próprio conjunto de translation tables e o kernel faz a troca de uma para outra quando realiza a troca de contexto. Mas boa parte do sistema de memória é usada apenas pelo kernel e tem fixos seus mapeamentos de endereços virtuais para físicos, onde as entradas da translation table raramente mudam. Considerando isso, na ARMv8 há vários recursos para lidar com esse aspecto de maneira eficiente.

Os endereços-base para as translation tables são especificados nos Translation Table Base Registers (ou TTBR's): TTBR0_EL1 e TTBR1_EL1. O endereço virtual do processador de uma busca de instrução ou acesso a dados é de 64 bits. No entanto, é preciso mapear ambas as regiões definidas acima em um único mapa de memória de endereço físico de 48 bits.

A figura abaixo mostra como o espaço do kernel (em TTBR1) é mapeado para a área mais significativa da memória e o espaço de endereço virtual associado a cada aplicativo (em TTBR0) é mapeado para a área menos significativa da memória. Porém, ambos são mapeados para um espaço de endereço físico muito menor, à direita.



Algo a se observar é que EL2 e EL3 têm TTBR0, mas não têm TTBR1. Portanto, se EL2 estiver usando AArch64, ele só pode usar endereços virtuais no intervalo de 0x0 a 0x0000FFFF_FFFFFFFF. O mesmo se aplica para EL3 usando AArch64.

Qual das duas translation tables será usada depende dos bits mais significativos do endereço virtual. Se eles estiverem todos setados em 0, será selecionada a translation table apontada por TTBR0. Se eles estiverem todos setados em 1, será selecionada a translation table apontada por TTBR1.

A quantidade exata de bits mais significativos do endereço virtual a ser considerada é definida no Translation Control Register (ou TCR_EL1), pelos campos de tamanho T0SZ5:0 e T1SZ5:0, presentes na figura abaixo.

2.9.4.3.1. Acessando registradores de suporte à translation table

Enquanto no ARMv7 os registradores eram normalmente acessados por meio de operações do coprocessador 15 (CP15), no AArch64 a configuração do sistema é controlada por meio de registradores do sistema e acessada usando instruções MSR e MRS. Note que o sufixo dos registradores permite saber a partir de qual nível é possível acessá-los (por exemplo, TTBR0_EL2 é acessível a partir de EL2 e EL3). Além disso, para acessar qualquer um dos registradores de suporte

a translation tables, o processador deve estar rodando em modo privilegiado no momento do acesso.

2.9.5. Exemplo de Implementação

Abaixo, temos uma função que realiza a troca de contexto, ela recebe o contexto "o" e "n" (assumimos que estes estão em r0/a0 e r1/a1, respectivamente).

```
```

> void CPU::switch_context(Context ** o, Context * n) { // Adiciona o contexto na stack e ajusta "o" para o valor correspondente ASM(" sub sp, #4 \n" // reserva espaço para o Program Counter (PC) " push {r12} \n" // salva r12 para usá-lo como um registrador temporário " adr r12, .ret \n"); // calcula endereço de retorno ASM(" str r12, [sp,#4] \n" // salva PC calculado " pop {r12} \n" // restaura o r12 usado como temporário " push {r0-r12, lr} \n"); // adiciona todos os registradores à stack (LR primeiro, r0 por último) mrs12(); // move flags para registrador temporário ASM(" push {r12} \n"); // salva flags ASM(" str sp, [r0] \n"); // atualiza contexto volátil "o" // Seta sp para "n" e faz o pop do contexto ASM(" mov sp, r1 \n" // lê contexto volátil "n" colocando-o no stack pointer " isb \n"); // serializa o pipeline para que o SP seja atualizado antes do pop ASM(" pop {r12} \n"); // pega (pop) flags colocando-as no registrador temporário r12 msr12(); // restaura flags ASM(" pop {r0-r12, lr} \n"); // pega (pop) todos os registradores (r0 primeiro, LR por último) ASM(" pop {pc} \n" // restaura PC ".ret: bx lr \n"); // retorna }

O estado atual do processador quando ocorre uma exceção é guardado em espaços diferentes do registrador PSTATE. Se uma exceção é levantada, a informação de PSTATE é salva no SPSR_ELn (Saved Program Status Register), sendo n o nível em que a exceção foi chamada.

O registrador especial NZCV irá guardar as flags de condição:

- N: Negative condition flag
- Z: Zero condition flag
- C: Carry condition flag
- V: oVerflow condition flag

Por isso, é necessário definir as funções mrs12() e msr12(), a primeira move essas flags para o registrador temporário "r12" e a segunda restaura as mesmas.

```
```

> public: static Flags flags() { Reg r; ASM("mrs %0, nzcv" : "=r"(r) :); return r; } static void flags(Flags r) { ASM("msr nzcv, %0" : : "r"(r) : "cc"); } static void mrs12() { ASM("mrs r12, nzcv" : : : "r12"); } static void msr12() { ASM("msr nzcv, r12" : : : "cc"); }

2.9.6. Referências

- https://afteracademy.com/blog/what-is-context-switching-in-operating-system
- https://www.tutorialandexample.com/what-is-context-switching/
- https://wiki.osdev.org/Context_Switching
- https://developer.arm.com/documentation/den0024/a/The-Memory-Management-Unit/Context-switching?lang=en
- https://developer.arm.com/documentation/den0024/a/The-Memory-Management-Unit/Separation

-of-kernel-and-application-Virtual-Address-spaces?lang=en
- https://developer.arm.com/documentation/den0024/a/ARMv8-Registers/System-registers?lang=en
- Arm® Architecture Reference Manual (Armv8, for A-profile architecture)
- https://developer.arm.com/documentation/100933/0100/Processor-state-in-exception-handling