Memória

Wedson Almeida Filho, 29/05/2025

Recapitulando nosso modelo de CPU

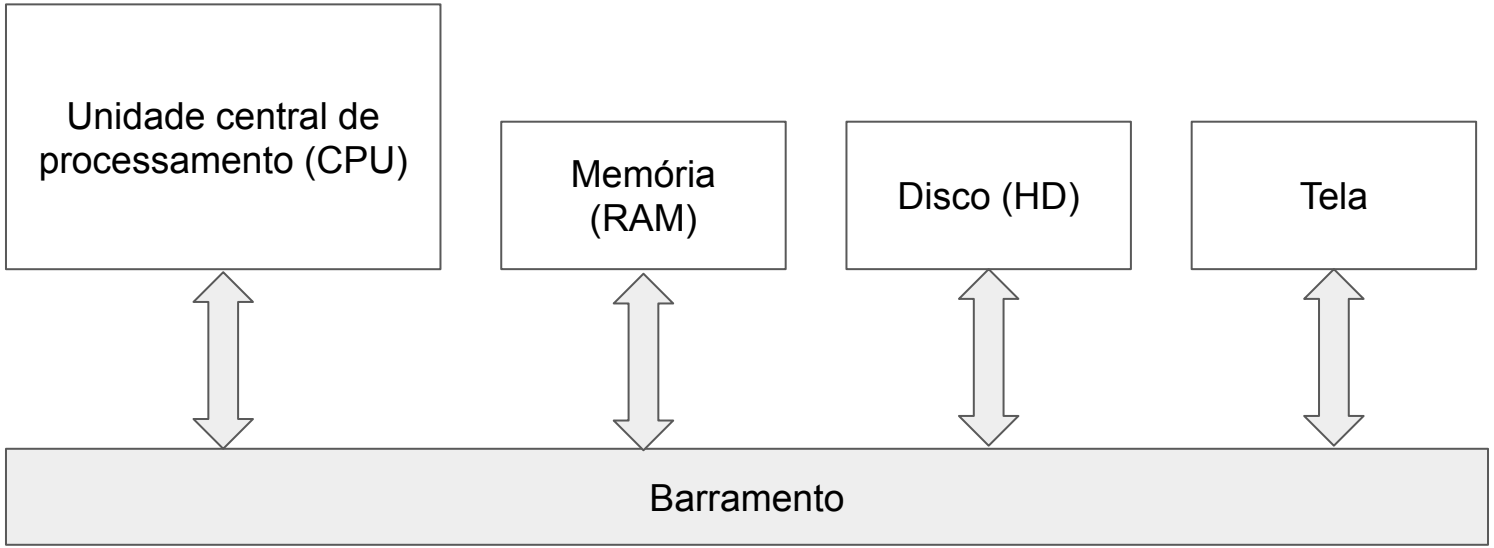
Um laço como o seguinte:

```
loop {
    if irq_enabled then check_irq()

    instr, instr_len = fetch_instr(pc)
    pc = pc + instr_len

    execute_instr(instr)
}
```

Dois tipos de memória

- Registradores
  - Memória mais rápida
  - Quantidade limitada
    - x86-64: 16 registradores gerais (rax, rbx, rcx, rdx, rsi, rdi, rsp, rbp, r8, …, 15)
    - aarch64: 31 registradores gerais (x0, …, x30), x31 pode de sp ou zero
  - CPU opera diretamente
- Memória de uso geral
  - Por exemplo, pentes DDR
  - 100 a 1000 vezes mais lenta
  - Requerem transações em um barramento

Revisão: modelo de computador



Isolamento de processos

- Como o sistema operacional consegue isolar um processo de outro?
- Como ele consegue isolar o núcleo (kernel) dos processos?

Isolamento de processos

- Como o sistema operacional consegue isolar um processo de outro?
- Como ele consegue isolar o núcleo (kernel) dos processos?

- Com a ajuda do hardware!

Isolamento de processos e do sistema operacional

- Nenhum isolamento
  - Como no DOS
  - Qualquer processo pode acessar toda a memória
- Unidade de proteção de memória (*memory protection unit*, MPU)
  - Comum em microcontroladores
  - Uma faixa de memória é "ativa" por vez
  - O kernel modifica a faixa em cada troca de contexto
  - Cada processo tem uma faixa
- Unidade de gerenciamento de memória (*memory management unit*, MMU)
  - Comum processadores modernos
  - Introdução do conceito de memória virtual
  - O kernel gerencia a tradução de memória virtual para física

Unidade de proteção de memória

- A CPU tem dois registradores adicionais:
  - Endereço base
  - Tamanho

- Exemplo:
  - Endereço base: 0x1000000
  - Tamanho: 0x1000
  - Endereços aceitos pela CPU: de 0x1000000 a 0x1000fff

Paginação

- A CPU tem um registrador a mais (dois no caso do aarch64)
  - Endereço da tabela de paginação
    - x86-64: cr3
    - aarch64: ttbr0_el1 e ttbr1_el1
- O espaço de endereçamento é particionado em páginas (4KB normalmente)
- Acessos à memória são convertidos
  - De memória virtual para memória física
  - Pelo hardware, antes de chegar ao barramento
- Erros de conversão resultam em exceções da CPU

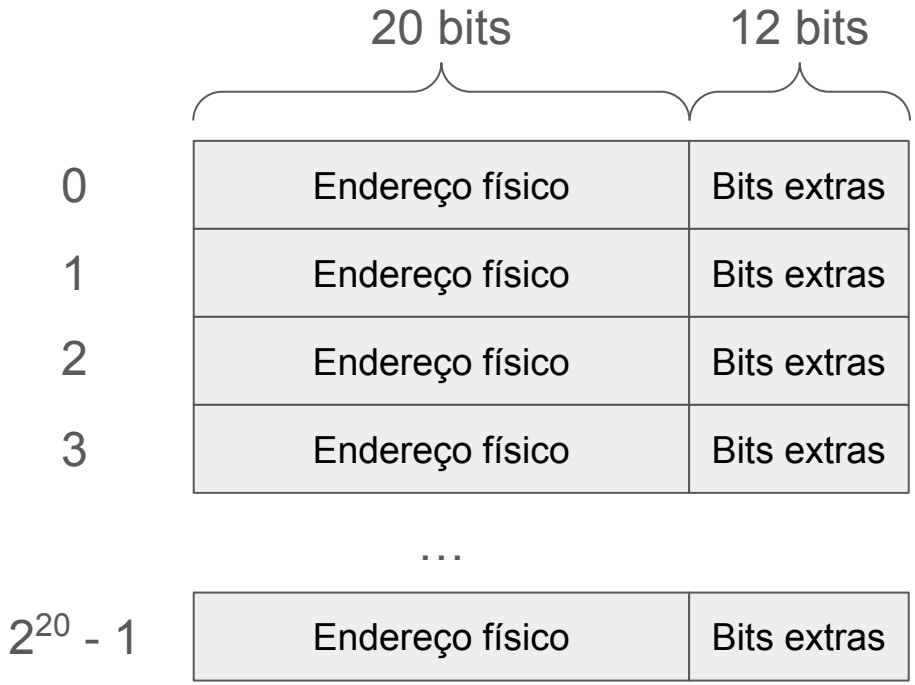
Exemplo espaço de endereçamento de 32 bits

- Tamanho da página: 4KB

Tabela de paginação para exemplo anterior

- 2^{20} elementos
- Se cada elemento tiver 20 bits, tabela gasta 2,5MB
- Se cada elemento tiver 32 bits, tabela gasta 4MB

Bits extras para informações adicionais



Bits extras do x86

1. P (*present*) – a página está presente ou não
2. R/W (*read/write*) – a página é *writable* ou não
3. U/S (*user/supervisor*) – a página é acessível de modo usuário
4. PWT (*page write-through*) – escritas para a página são escritas direto para memória; caso contrário ficam no cache
5. PCD (*page cache disable*) – cache é desabilitado
6. A (*accessed*) – a página foi acessada
7. D (*dirty*) – a página está "suja" (foi modificada)
8. PAT (*page attribute table*) – combinado com PCD and PWT, escolhe cache
9. G (*global*) – mapeamento não é invalidado quando cr3 muda
10. AVL (*available*, 3 bits) – disponível para uso do sistema operacional

E no caso de 64 bits?

- Espaço de endereçamento de 64 bits
- Página de 4KB
- Tabela teria 2^{52} elementos, mais de 10^{15} elementos
- Considerando 52 bits por elemento, a tabela precisaria de 26 PB (petabytes)

Representando uma tabela de paginação

- Precisamos de uma versão mais compacta

- Tabela contém 0 na maioria dos elementos
- Sugestões de como representar?

Representando uma tabela de paginação

- Precisamos de uma versão mais compacta

- Tabela contém 0 na maioria dos elementos
- Sugestões de como representar?

- E se usarmos uma **árvore**?

Representação em árvore

- Quantos filhos por nó?

- O que influencia essa decisão?

Em 32 bits

- Cada nó da árvore ocupa uma página (4KB)
- Cada filho ocupa 32 bits (4B)
- Então cada nó tem 1024 filhos
- O índice de cada filho requer 10 bits ($2^{10} = 1024$)
- Em dois níveis, todos os endereços são representáveis:
  - 10bits + 10bits + 12bits = 32bits

Diagrama extraído do manual da Intel (p. 4-10)

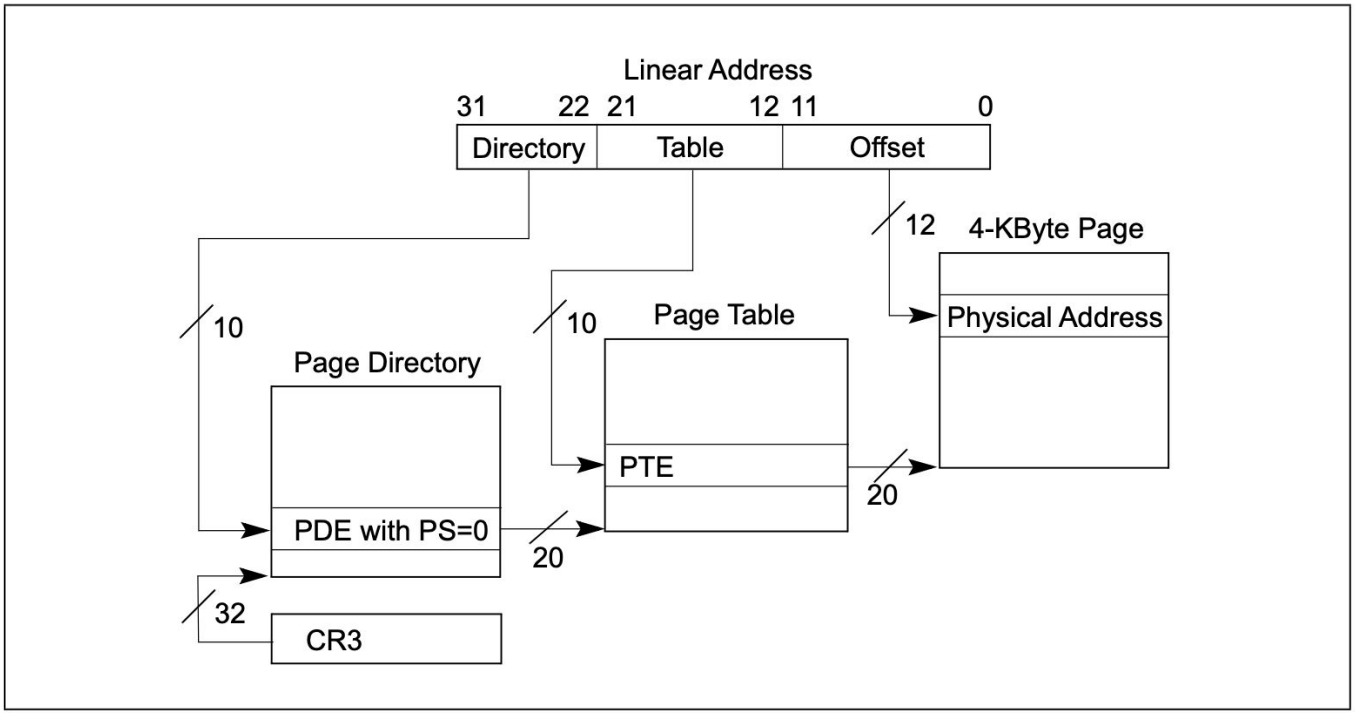


Figure 4-2. Linear-Address Translation to a 4-KByte Page using 32-Bit Paging

Como converter um endereço virtual em físico?

- A raiz (cr3) é o nó atual
- Ler os 10 bits mais significativos do endereço para escolher o filho
- Se o filho não estiver presente, falha de paginação
- Caso contrário, o nó atual é o filho
- Ler os 10 bits seguintes do endereço para escolher o filho
- Se o filho não estiver presente, falha de paginação
- Caso contrário, usar filho como página física
- E os últimos 12 bits como deslocamento na página física

Como converter um endereço virtual em físico?

- A raiz (cr3) é o nó atual
- Ler os 10 bits mais significativos do endereço para escolher o filho
- Se o filho não estiver presente, falha de paginação
- Caso contrário, o nó atual é o filho
- Ler os 10 bits seguintes do endereço para escolher o filho
- Se o filho não estiver presente, falha de paginação
- Caso contrário, usar filho como página física
- E os últimos 12 bits como deslocamento na página física

- Exceção: é possível que a raiz aponte diretamente para a página final (4MB)
  - Quando o bit 7 está ligado

Diagrama no caso de páginas de 4MB

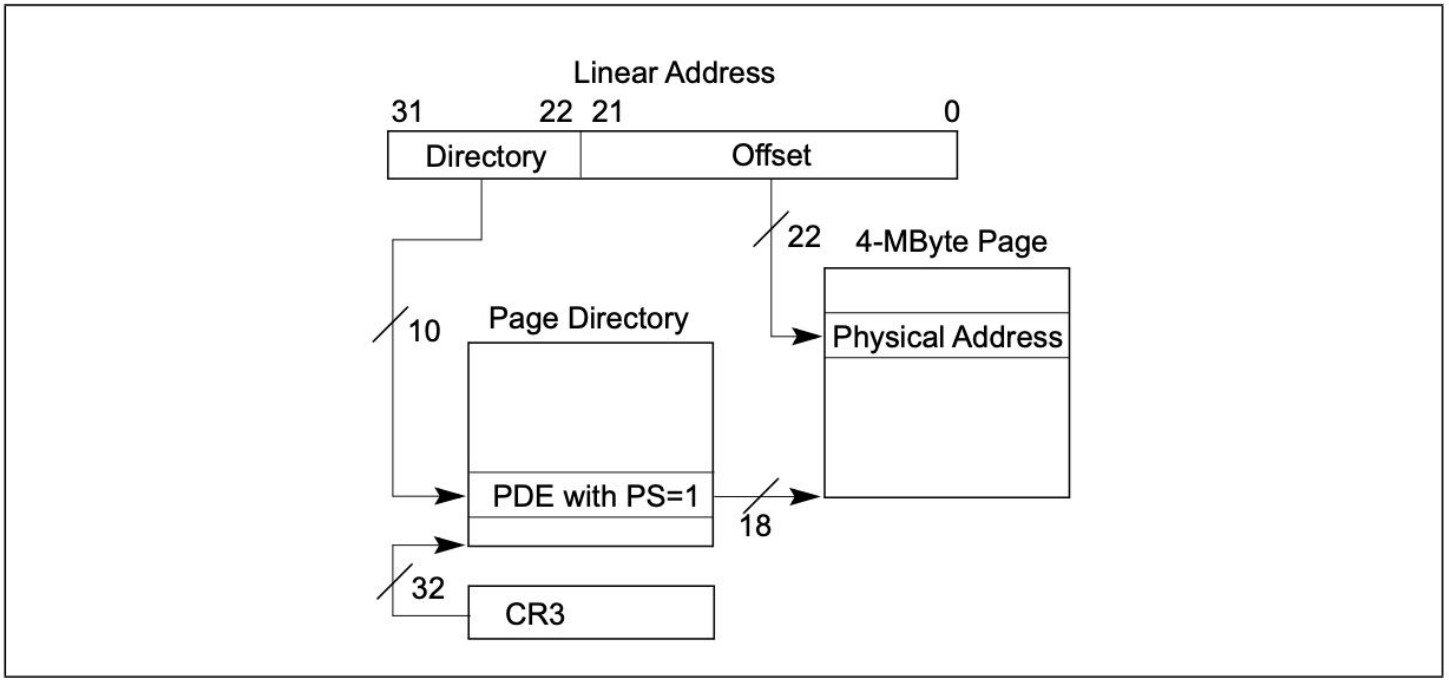
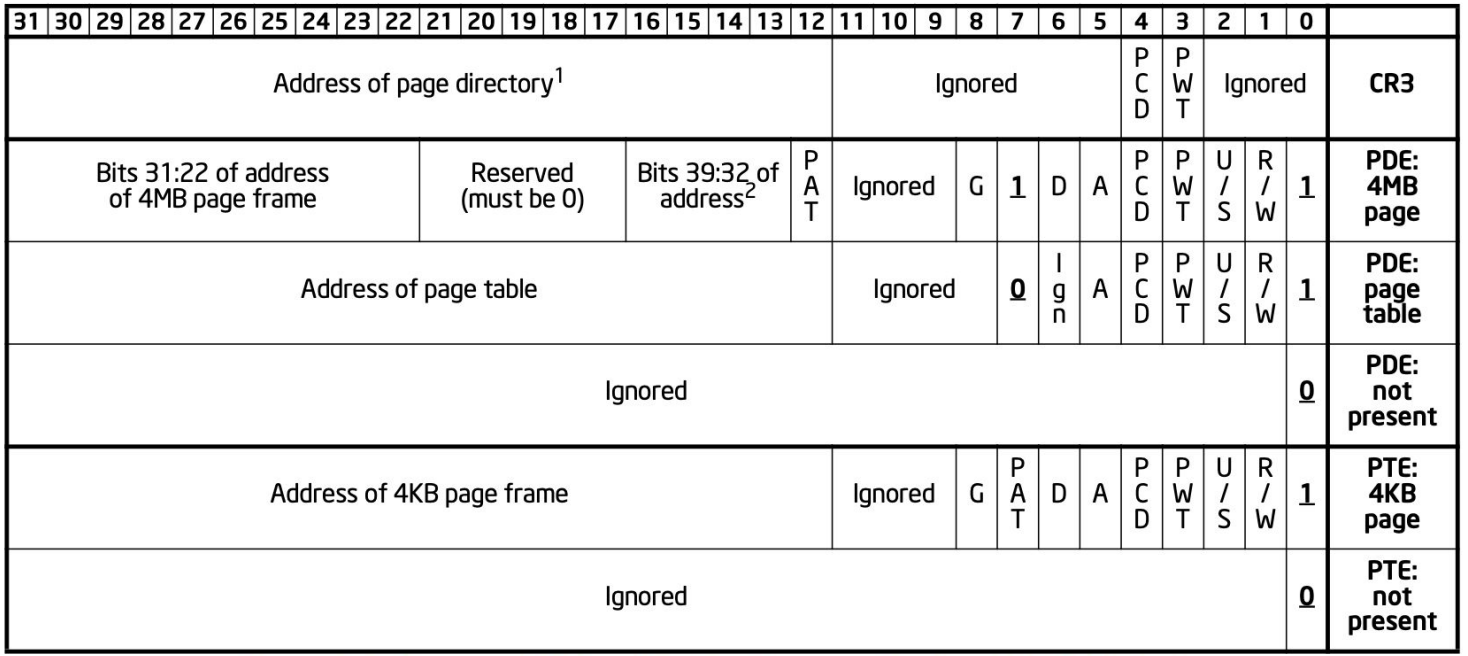


Figure 4-3. Linear-Address Translation to a 4-MByte Page using 32-Bit Paging

Formato da entrada no x86 (p. 4-11)



31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
Address of page directory[1]																				Ignored							PCD	PWT	Ignored			CR3
Bits 31:22 of address of 4MB page frame										Reserved (must be 0)						Bits 39:32 of address[2]			PAT	Ignored			G	1	D	A	PCD	PWT	U/S	R/W	1	PDE: 4MB page
Address of page table																				Ignored				0	Ign	A	PCD	PWT	U/S	R/W	1	PDE: page table
Ignored																															0	PDE: not present
Address of 4KB page frame																				Ignored			G	PAT	D	A	PCD	PWT	U/S	R/W	1	PTE: 4KB page
Ignored																															0	PTE: not present

Figure 4-4. Formats of CR3 and Paging-Structure Entries with 32-Bit Paging

Prática: rodar diretamente em uma máquina virtual

- Clonar repositório `https://github.com/wedsonaf/metal_c.git`
  - Por exemplo, `git clone https://github.com/wedsonaf/metal_c.git`
- Rodar
  - Por exemplo, `make run`
  - Expectativa: "Olá mundo"
  - Para sair: ctrl+a x

Prática: habilitar paginação

- Declarar a raiz, alinhada com 4096
  - `__attribute__((aligned(4096)))`
- Inicializar os primeiros 4MB com identidade, valor:
  - (1 << 7) | (1 << 1) | (1 << 0)
  - Página de 4MB, leitura/escrita,presente
- Inicializar `cr3` para apontar para a raiz
  - `set_cr3()`
- Habilitar páginas de 4MB
  - Ligar bit 4 de `cr4`
  - `set_cr4()`, `get_cr4()`
- Habilitar paginação
  - Ligar bit 31 de `cr0`
  - `set_cr0()`, `get_cr0()`
- Escrever "Paginação habilitada"

Solução

```
uint32_t raiz[1024] __attribute__((aligned(4096))) = {
	// 0: Present, 1: Read/write, 7: Page size
	(1 << 7) | (1 << 1) | (1 << 0)
};

int main()
{
	puts("\nOlá mundo!\n");

	set_cr3((uint32_t)&raiz[0]);
	set_cr4(get_cr4() | (1 << 4));
	set_cr0(get_cr0() | (1 << 31));

	puts("Paginação habilitada!\n");

	for(;;);
}
```

Inspecionar se bits estão configurados com gdb

- Rodar qemu
- Rodar gdb em outro terminal
  - `gdb ./kernel`
  - `target remote :1234`
  - `p/x $cr3`
  - `p/x $cr4`
  - `p/x $cr0`
  - `q`
- Comparar resultados com paginação habilitada e desabilitada

Prática: testar que paginação está ativa

- Acessar uma página que não está mapeada

- Declarar uma variável global
- Criar um ponteiro apontando para a variável mais 4MB
  - Certamente não está mapeado, já que só mapeamos os primeiros 4MB
- Tentar ler desse ponteiro

- Qual é o resultado?
- Qual é o resultado se paginação estiver desabilitada?

Habilitar detalhes de exceção no qemu

- Modificar Makefile
- Adicionar "-d int" à linha do qemu
- Detalhes sobre exceções aparecerão no terminal

Exceções do x86 (p. 6-2 do manual)

Table 6-1. Protected-Mode Exceptions and Interrupts

Vector	Mnemonic	Description	Type	Error Code	Source
0	#DE	Divide Error	Fault	No	DIV and IDIV instructions.
1	#DB	Debug Exception	Fault/ Trap	No	Instruction, data, and I/O breakpoints; single-step; and others.
2	—	NMI Interrupt	Interrupt	No	Nonmaskable external interrupt.
3	#BP	Breakpoint	Trap	No	INT3 instruction.
4	#OF	Overflow	Trap	No	INTO instruction.
5	#BR	BOUND Range Exceeded	Fault	No	BOUND instruction.
6	#UD	Invalid Opcode (Undefined Opcode)	Fault	No	UD instruction or reserved opcode.
7	#NM	Device Not Available (No Math Coprocessor)	Fault	No	Floating-point or WAIT/FWAIT instruction.
8	#DF	Double Fault	Abort	Yes (zero)	Any instruction that can generate an exception, an NMI, or an INTR.
9		Coprocessor Segment Overrun (reserved)	Fault	No	Floating-point instruction.[1]
10	#TS	Invalid TSS	Fault	Yes	Task switch or TSS access.
11	#NP	Segment Not Present	Fault	Yes	Loading segment registers or accessing system segments.
12	#SS	Stack-Segment Fault	Fault	Yes	Stack operations and SS register loads.
13	#GP	General Protection	Fault	Yes	Any memory reference and other protection checks.
14	#PF	Page Fault	Fault	Yes	Any memory reference.

Exceções do x86 (continuação)

Table 6-1. Protected-Mode Exceptions and Interrupts (Contd.)

Vector	Mnemonic	Description	Type	Error Code	Source
15	—	(Intel reserved. Do not use.)		No	
16	#MF	x87 FPU Floating-Point Error (Math Fault)	Fault	No	x87 FPU floating-point or WAIT/FWAIT instruction.
17	#AC	Alignment Check	Fault	Yes (Zero)	Any data reference in memory.[2]
18	#MC	Machine Check	Abort	No	Error codes (if any) and source are model dependent.[3]
19	#XM	SIMD Floating-Point Exception	Fault	No	SSE/SSE2/SSE3 floating-point instructions[4]
20	#VE	Virtualization Exception	Fault	No	EPT violations[5]
21	#CP	Control Protection Exception	Fault	Yes	RET, IRET, RSTORSSP, and SETSSBSY instructions can generate this exception. When CET indirect branch tracking is enabled, this exception can be generated due to a missing ENDBRANCH instruction at target of an indirect call or jump.
22-31	—	Intel reserved. Do not use.			
32-255	—	User Defined (Non-reserved) Interrupts	Interrupt		External interrupt or INT *n* instruction.

Exemplo de saída

```
Booting from ROM..
Olá mundo!
Paginação habilitada!
Endereços de ptr1 e ptr2: 0x101000 0x501000
Valores em ptr1 e ptr2: 0x10 check_exception old: 0xffffffff new 0xe
     0: v=0e e=0000 i=0 cpl=0 IP=0008:001000c8 pc=001000c8 SP=0010:00113fc4 CR2=00501000
EAX=00000020 EBX=00103000 ECX=000003fd EDX=00100363
ESI=001003a3 EDI=00100363 EBP=00113ff8 ESP=00113fc4
EIP=001000c8 EFL=00000046 [---Z-P-] CPL=0 II=0 A20=1 SMM=0 HLT=0
ES =0010 00000000 ffffffff 00cf9300 DPL=0 DS   [-WA]
CS =0008 00000000 ffffffff 00cf9a00 DPL=0 CS32 [-R-]
SS =0010 00000000 ffffffff 00cf9300 DPL=0 DS   [-WA]
DS =0010 00000000 ffffffff 00cf9300 DPL=0 DS   [-WA]
FS =0010 00000000 ffffffff 00cf9300 DPL=0 DS   [-WA]
GS =0010 00000000 ffffffff 00cf9300 DPL=0 DS   [-WA]
LDT=0000 00000000 0000ffff 00008200 DPL=0 LDT
TR =0000 00000000 0000ffff 00008b00 DPL=0 TSS32-busy
GDT=     000cb2b4 00000027
IDT=     00000000 000003ff
CR0=80000011 CR2=00501000 CR3=00102000 CR4=00000010
DR0=00000000 DR1=00000000 DR2=00000000 DR3=00000000
DR6=ffff0ff0 DR7=00000400
CCS=00000014 CCD=00000000 CCO=LOGICB
EFER=0000000000000000
```

Solução

```
int global = 0x10;

void teste(void)
{
        int *ptr1 = &global;
        int *ptr2 = (int *)((char *)ptr1 + 4 * 1024 * 1024);

        puts("Endereços de ptr1 e ptr2: ");
        put_hex((uint32_t)ptr1);
        puts(" ");
        put_hex((uint32_t)ptr2);
        puts("\n");

        puts("Valores em ptr1 e ptr2: ");
        put_hex(*ptr1);
        puts(" ");
        put_hex(*ptr2);
        puts("\n");
}
```

Prática: mapear os 4MB seguintes aos primeiros

- Atualizar exemplo anterior
- Modificar raiz tal que
  - O segundo elemento (índice 1), aponte para o endereço físico 0
  - Ou seja, os endereços virtuais 0 - 4MB e 4MB - 8MB apontam para o mesmo endereço físico
- Mostrar o conteúdo dos dois ponteiros
- Modificar o valor
- Mostrar o conteúdo dos dois ponteiros

Solução

```
void teste(void)
{
        int *ptr1 = &global;
        int *ptr2 = (int *)((char *)ptr1 + 4 * 1024 * 1024);

        raiz[1] = (1 << 7) | (1 << 1) | (1 << 0);

        puts("Valores em ptr1 e ptr2: ");
        put_hex(*ptr1);
        puts(" ");
        put_hex(*ptr2);
        puts("\n");

        *ptr1 = 0x20;
        puts("Valores em ptr1 e ptr2: ");
        put_hex(*ptr1);
        puts(" ");
        put_hex(*ptr2);
        puts("\n");
}
```

Prática: efeito na tabela de paginação ao ler

- Modificar o exemplo anterior
- Somente ler o conteúdo de ptr2
- Mostrar o valor de raiz[1] antes e depois da leitura

- Qual é o resultado?

Solução

```
void teste(void)
{
        int *ptr1 = &global;
        int *ptr2 = (int *)((char *)ptr1 + 4 * 1024 * 1024);

        raiz[1] = (1 << 7) | (1 << 1) | (1 << 0);

        puts("raiz[1] antes: ");
        put_hex(raiz[1]);
        puts("\n");

        put_hex(*ptr2);
        puts("\n");

        puts("raiz[1] depois: ");
        put_hex(raiz[1]);
        puts("\n");
}
```

Prática: efeito na tabela de paginação ao escrever

- Modificar o exemplo anterior
- Somente alterar o conteúdo de ptr2
- Mostrar o valor de raiz[1] antes e depois da escrita

- Qual é o resultado?

Solução

```
void teste(void)
{
        int *ptr1 = &global;
        int *ptr2 = (int *)((char *)ptr1 + 4 * 1024 * 1024);

        raiz[1] = (1 << 7) | (1 << 1) | (1 << 0);

        puts("raiz[1] antes: ");
        put_hex(raiz[1]);
        puts("\n");

        *ptr2 = 0x20;

        puts("raiz[1] depois: ");
        put_hex(raiz[1]);
        puts("\n");
}
```

Prática: remover mapeamento

- Modificar prática anterior
- Ler o conteúdo de ptr2
- Remover o mapeamento
- Ler o conteúdo de ptr2 de novo

- Qual é o resultado esperado?
- Qual é o resultado real?

Solução

```
void teste(void)
{
        int *ptr1 = &global;
        int *ptr2 = (int *)((char *)ptr1 + 4 * 1024 * 1024);

        raiz[1] = (1 << 7) | (1 << 1) | (1 << 0);

        puts("Valor de ptr2: ");
        put_hex(*ptr2);
        puts("\n");

        raiz[1] = 0;

        puts("Valor de ptr2: ");
        put_hex(*ptr2);
        puts("\n");
}
```

Prática: invalidar o buffer de mapeamento

- Modificar exemplo anterior
- Após remover o mapeamento, invalidar o TLB
  - Função `invlpg()`

- Qual é o resultado?

Descrição da instrução (Manual Intel, Vol. 2A p. 3-536)

Description

Invalidates any translation lookaside buffer (TLB) entries specified with the source operand. The source operand is a memory address. The processor determines the page that contains that address and flushes all TLB entries for that page.[25]

The INVLPG instruction is a privileged instruction. When the processor is running in protected mode, the CPL must be 0 to execute this instruction.

The INVLPG instruction normally flushes TLB entries only for the specified page; however, in some cases, it may flush more entries, even the entire TLB. The instruction invalidates TLB entries associated with the current PCID and may or may not do so for TLB entries associated with other PCIDs. (If PCIDs are disabled — CR4.PCIDE = 0 — the current PCID is 000H.) The instruction also invalidates any global TLB entries for the specified page, regardless of PCID.

For more details on operations that flush the TLB, see "MOV—Move to/from Control Registers" in the Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2B, and Section 4.10.4.1, "Operations that Invalidate TLBs and Paging-Structure Caches," in the Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A.

This instruction's operation is the same in all non-64-bit modes. It also operates the same in 64-bit mode, except if the memory address is in non-canonical form. In this case, INVLPG is the same as a NOP.

Solução

```
void teste(void)
{
        int *ptr1 = &global;
        int *ptr2 = (int *)((char *)ptr1 + 4 * 1024 * 1024);

        raiz[1] = (1 << 7) | (1 << 1) | (1 << 0);

        puts("Valor de ptr2: ");
        put_hex(*ptr2);
        puts("\n");

        raiz[1] = 0;
        invlpg(ptr2);

        puts("Valor de ptr2: ");
        put_hex(*ptr2);
        puts("\n");
}
```

Prática: fazer a página ser somente leitura

- Modificar o exemplo anterior
- Ligar bit 16 em `cr0`
  - Para habilitar proteção em modo kernel também
- Mudar mapeamento para somente leitura
- Tentar modificar o conteúdo de `ptr2`

Solução: função `main`

```
int main()
{
        puts("\nOlá mundo!\n");

        set_cr3((uint32_t)&raiz[0]);
        set_cr4(get_cr4() | (1 << 4));
        set_cr0(get_cr0() | (1 << 31) | (1 << 16));

        puts("Paginação habilitada!\n");

        teste();

        for(;;);
}
```

Solução: função `teste`

```
void teste(void)
{
        int *ptr1 = &global;
        int *ptr2 = (int *)((char *)ptr1 + 4 * 1024 * 1024);

        raiz[1] = (1 << 7) | (1 << 0);

        puts("Prestes a modificar\n");
        *ptr2 = 0x30;
        puts("ptr2 modificado\n");
}
```

Prática: instalar tratador de exceção

- Baixar a última versão do repositório
  - Incluindo a função `init_idt()`
- Chamar a função init_idt()
  - Instala um tratador de exceção para falhas de paginação

- Qual é o resultado?

Solução

```
void teste(void)
{
        int *ptr1 = &global;
        int *ptr2 = (int *)((char *)ptr1 + 4 * 1024 * 1024);

        init_idt();

        raiz[1] = (1 << 7) | (1 << 0);

        puts("Prestes a modificar\n");
        *ptr2 = 0x30;
        puts("ptr2 modificado\n");
}
```

Prática: personalizar o tratador de exceção

- Implementar uma função com o seguinte nome e protótipo:
  `void page_fault_handler(struct state *s)`
- Colocar um corpo que imprime "Falha de paginação"

- Qual é o resultado?

Solução

```
void page_fault_handler(struct state *s)
{
        puts("Falha de paginação\n");
}
```

Prática: mostrar detalhes sobre as falhas

- Registrador `cr2` (função get_cr2()) contém o endereço que falhou
- Campo `eip` da estrutura `state` é o endereço da instrução que falhou
- Campo `error` da estrutura `state` contém detalhes sobre a falha

Descrição do erro (do manual p. 6-44)

Exception Error Code

Yes (special format). The processor provides the page-fault handler with two items of information to aid in diagnosing the exception and recovering from it:

- An error code on the stack. The error code for a page fault has a format different from that for other exceptions (see Figure 6-11). The processor establishes the bits in the error code as follows:
  - P flag (bit 0).
    This flag is 0 if there is no translation for the linear address because the P flag was 0 in one of the paging-structure entries used to translate that address.
  - W/R (bit 1).
    If the access causing the page-fault exception was a write, this flag is 1; otherwise, it is 0. This flag describes the access causing the page-fault exception, not the access rights specified by paging.
  - U/S (bit 2).
    If a user-mode access caused the page-fault exception, this flag is 1; it is 0 if a supervisor-mode access did so. This flag describes the access causing the page-fault exception, not the access rights specified by paging.

Continuação

— RSVD flag (bit 3).
This flag is 1 if there is no translation for the linear address because a reserved bit was set in one of the paging-structure entries used to translate that address.

— I/D flag (bit 4).
This flag is 1 if the access causing the page-fault exception was an instruction fetch. This flag describes the access causing the page-fault exception, not the access rights specified by paging.

— PK flag (bit 5).
This flag is 1 if the access causing the page-fault exception was a data access to a linear address with a protection key for which the protection-key rights registers disallow access.

— SS (bit 6).
If the access causing the page-fault exception was a shadow-stack access (including shadow-stack accesses in enclave mode), this flag is 1; otherwise, it is 0. This flag describes the access causing the page-fault exception, not the access rights specified by paging.

— HLAT (bit 7).
This flag is 1 if there is no translation for the linear address using HLAT paging because, in one of the paging-structure entries used to translate that address, either the P flag was 0 or a reserved bit was set. An error code will set this flag only if it clears bit 0 or sets bit 3. This flag will not be set by a page fault resulting from a violation of access rights, nor for one encountered during ordinary paging, including the case in which there has been a restart of HLAT paging.

— SGX flag (bit 15).
This flag is 1 if the exception is unrelated to paging and resulted from violation of SGX-specific access-control requirements. Because such a violation can occur only if there is no ordinary page fault, this flag is set only if the P flag (bit 0) is 1 and the RSVD flag (bit 3) and the PK flag (bit 5) are both 0.

See Section 4.6, “Access Rights,” and Section 4.7, “Page-Fault Exceptions,” for more information about page-fault exceptions and the error codes that they produce.

Solução

```
void page_fault_handler(struct state *s)
{
        puts("Falha de paginação: error=");
        put_hex(s->error);
        puts(", eip=");
        put_hex(s->eip);
        puts(", cr2=");
        put_hex(get_cr2());
        puts("\n");
}
```

Prática: modificando a tabela no tratador

- Habilitar escrita no segundo bloco de 4MB
  - Ligar bit 1 de `raiz[1]`
- Qual é o resultado?

Solução

```
void page_fault_handler(struct state *s)
{
        puts("Falha de paginação: error=");
        put_hex(s->error);
        puts(", eip=");
        put_hex(s->eip);
        puts(", cr2=");
        put_hex(get_cr2());
        puts("\n");

        raiz[1] |= 1 << 1;
}
```

Desafio

- Recuperar a execução sem modificar `raiz`
- Solução específica para este programa

Solução

```
void page_fault_handler(struct state *s)
{
        puts("Falha de paginação: error=");
        put_hex(s->error);
        puts(", eip=");
        put_hex(s->eip);
        puts(", cr2=");
        put_hex(get_cr2());
        puts("\n");

        s->ebx = 0xffd03008;
}
```

Como chegar na solução

- Verificar instrução falhando:
  - Falha de paginação: error=0x3, eip=0x100103, cr2=0x501000

Como chegar na solução

- Verificar instrução falhando:
  - Falha de paginação: error=0x3, eip=0x100103, cr2=0x501000
  - `> gdb ./kernel`
  - `> x/1i 0x100103`
  - `0x100103 <teste+51>: movl   $0x30,0x3fdff8(%ebx)`

Como chegar na solução

- Verificar instrução falhando:
  - Falha de paginação: error=0x3, eip=0x100103, cr2=0x501000
  - `> gdb ./kernel`
  - `> x/1i 0x100103`
  - `0x100103 <teste+51>: movl   $0x30,0x3fdff8(%ebx)`
- Que valor `ebx` deve ter para o endereço ser 0x501000 - 4MB?

Como chegar na solução

- Verificar instrução falhando:
  - Falha de paginação: error=0x3, eip=0x100103, cr2=0x501000
  - `> gdb ./kernel`
  - `> x/1i 0x100103`
  - `0x100103 <teste+51>: movl   $0x30,0x3fdff8(%ebx)`
- Que valor ebx deve ter para o endereço ser 0x501000 - 4MB?
  - `0x101000 - 0x3fdff8 = 0xffd03008`

Como chegar na solução

- Verificar instrução falhando:
  - Falha de paginação: error=0x3, eip=0x100103, cr2=0x501000
  - `> gdb ./kernel`
  - `> x/1i 0x100103`
  - `0x100103 <teste+51>: movl   $0x30,0x3fdff8(%ebx)`
- Que valor ebx deve ter para o endereço ser 0x501000 - 4MB?
  - `0x101000 - 0x3fdff8 = 0xffd03008`
- Modificar o valor de ebx no tratador de exceções
  - `s->ebx = 0xffd03008;`

Como chegar na solução

- Verificar instrução falhando:
  - Falha de paginação: error=0x3, eip=0x100103, cr2=0x501000
  - `> gdb ./kernel`
  - `> x/1i 0x100103`
  - `0x100103 <teste+51>: movl   $0x30,0x3fdff8(%ebx)`
- Que valor ebx deve ter para o endereço ser 0x501000 - 4MB?
  - `0x101000 - 0x3fdff8 = 0xffd03008`
- Modificar o valor de ebx no tratador de exceções
  - `s->ebx = 0xffd03008;`
- Risco: ebx ser usado para outros fins depois

Prática: paginação em 2 níveis

- Inicializar `raiz[2]` para apontar para um nó filho
  - Ou seja, usando páginas de 4KB
- Nó filho é todo zerado para começar

- Criar um ponteiro `ptr3`, 4MB à frente de `ptr2`
- Tentar ler o conteúdo de `ptr3`

- Qual é o resultado?

Solução

```
uint32_t filho[1024] __attribute__((aligned(4096)));
void teste(void)
{
        int *ptr1 = &global;
        int *ptr2 = (int *)((char *)ptr1 + 4 * 1024 * 1024);
        int *ptr3 = (int *)((char *)ptr2 + 4 * 1024 * 1024);

        init_idt();

        raiz[1] = (1 << 7) | (1  << 1) | (1 << 0);
        raiz[2] = (uint32_t)&filho[0] | (1 << 1) | (1 << 0);

        puts("Valor de ptr3: ");
        put_hex(*ptr3);
        puts("\n");
}
```

Prática: atualizar nó filho para dar acesso a `ptr3`

- Modificar o tratador de exceções
- Para mapear o endereço de `ptr3`
  - Apontar para a mesma página física que `ptr1` e `ptr2`
- Outras entradas precisam continuar ausentes

- Mostrar o conteúdo de `ptr1`, `ptr2` e `ptr3`

Solução

```
void teste(void)
{
	int *ptr1 = &global;
	int *ptr2 = (int *)((char *)ptr1 + 4 * 1024 * 1024);
	int *ptr3 = (int *)((char *)ptr2 + 4 * 1024 * 1024);
	uint32_t indice = ((uint32_t)ptr3 >> 12) & 0x3ff;

	init_idt();

	raiz[1] = (1 << 7) | (1  << 1) | (1 << 0);
	raiz[2] = (uint32_t)&filho[0] | (1 << 1) | (1 << 0);

	filho[indice] = (indice << 12) | (1 << 1) | (1 << 0);

	puts("Valor de ptr3: ");
	put_hex(*ptr3);
	puts("\n");
}
```

Prática: duas tabelas de paginação

- Criar uma segunda tabela de paginação
- Fazer ptr3 ser diferente em cada uma:
  - Um aponta para o primeiro bloco de 4MB, outro aponta para o segundo bloco
- Mostrar o conteúdo de ptr3 com cada tabela de paginação

Solução

```
void teste(void)
{
        int *ptr1 = &global;
        int *ptr2 = (int *)((char *)ptr1 + 4 * 1024 * 1024);
        int *ptr3 = (int *)((char *)ptr2 + 4 * 1024 * 1024);
        uint32_t indice = ((uint32_t)ptr3 >> 12) & 0x3ff;

        init_idt();

        raiz[1] = (1 << 7) | (1  << 1) | (1 << 0);
        raiz[2] = (uint32_t)&filho[0] | (1 << 1) | (1 << 0);
        filho[indice] = (indice << 12) | (1 << 1) | (1 << 0);

        raiz1[2] = (uint32_t)&filho1[0] | (1 << 1) | (1 << 0);
        filho1[indice] = (indice << 12) | (1 << 1) | (1 << 0) + 4 * 1024 * 1024;

        puts("Valor de ptr3: ");
        put_hex(*ptr3);
        puts("\n");

        set_cr3((uint32_t)&raiz1[0]);
        puts("Valor de ptr3: ");
        put_hex(*ptr3);
        puts("\n");
}
```

Usos da tabela de paginação

Troca de memória (*memory swapping*)

- Objetivo: disponibilizar mais memória do que disponível
- Método: usar memória auxiliar mais lenta para armazenamento
  - Existe um arquivo ou dispositivo de "troca"
  - Usado para armazenar páginas de memória

Mecanismo de troca de memória

- Quando um processo precisa de memória e não há mais páginas
- Memória de um outro processo é escrita no arquivo de troca
  - Sua entrada na tabela de paginação é removida: bit 0 fica zero
  - Demais bits são usados para determinar onde no arquivo está o conteúdo da página
- Página física "recuperada" é disponibilizada para outro processo

- Quando o processo que teve sua memória escrita pro disco acessá-la
- O tratado de exceções será chamado
  - Percebendo que se trata de memória escrita no arquivo de troca
  - Recupera memória física de outro processo (possivelmente escrevendo pro arquivo de troca)
  - Lê do arquivo de troca o conteúdo da memória
  - Atualiza a tabela de paginação e retoma a execução do processo

Áreas de memória virtual

- Endereço de início e comprimento, por exemplo
  - Endereço virtual 0x1a000, 0xf000 bytes
- Áreas com atributos iguais, por exemplo
  - Área de somente leitura, vinda do arquivo X a partir do deslocamento Y
  - Área alocada dinamicamente (e zerada)
- Menos granular que a tabela de paginação
  - Tenta armazenar dados por regiões e não por página
- Armazenada normalmente em uma estrutura de árvore balanceada
- Nem sempre em sincronia com a tabela de paginação

Paginação sob demanda (*demand paging*)

- Marcar endereço como "ausente" na tabela de paginação
  - Normalmente é o estado padrão de uma tabela de paginação (tudo zerado)
- Se o processo nunca acessar (ler/escrever) a página
  - Ela não será alocada
  - Recursos são economizados
- Se o processo acessar a página:
  - Tratador de exceções é chamado
  - Consulta a árvore de memórias virtuais
    - Se a região for válida, atualiza a tabela de paginação faz processo tentar novamente
    - Caso contrário, gera um sinal (resulta em segmentation fault normalmente)

Operações que usam paginação sob demanda

- Carga de executáveis em memória
  - Partes do executável são adicionadas na árvore de áreas de memória virtual
  - Paginação sob demanda carrega apenas as áreas usadas
- Alocação de memória
  - Quando modo usuário precisa de memória, o kernel aloca somente regiões
  - Páginas físicas são alocadas sob demanda
- Mapeamento de arquivos
  - Por exemplo, com a chamada de sistema `mmap`
  - Também somente a árvore é atualizada
  - Memória física só é alocada e o arquivo só é lido quando o acesso acontece
- No primeiro e terceiro casos
  - Se os arquivos não forem alterados, podem ser descartados
  - Se o sistema estiver precisando de páginas físicas

Cópia na escrita (CoW - copy on write)

- Processos compartilham páginas que deviam ser privadas
- Enquanto nenhum processo modifica uma página
  - Página permanece compartilhada
  - Economizando recursos
- Quando um processo modifica uma página
  - O sistema operacional faz uma cópia da página

Mecanismo de cópia na escrita

- Quando uma página privada é compartilhada
  - Sua entrada na tabela de paginação é válida mas somente para leitura
- Processador lida com tradução de endereço virtual para físico
  - Enquanto somente leituras forem feitas
- Quando um processo *escreve* na página
  - Como a página não permite escritas, uma exceção é gerada
  - Tratador de exceções percebe que página é CoW
    - Consulta na árvores de áreas de memória virtual
  - Aloca nova memória física (possivelmente tirando de outro processo)
  - Faz cópia da página
  - Retoma processo

Operações que copiam em escrita

- Depois da chamada de sistema `fork`
  - O processo pai e filho compartilham páginas
  - Quando um dos dois modifica uma página, uma cópia é feita
- Mapeamento de arquivos com cópia privada
  - Um arquivo é mapeado e pode ser compartilhado por vários processos
  - Mas modificações da memória não deve ser escritas de volta e nem compartilhadas
    - Então cópias são feitas

Alocação tardia

- Memória física só é alocada quando necessário
- Permite melhor utilização de recursos
- Permite o "uso" de mais memória do que disponível
  - A soma das páginas utilizadas é maior do que a memória real e área de troca

- Entretanto: não há interface para indicar para processo que alocação falhou

- No Linux: oom (out-of-memory) killer
  - Heurísticas são usadas para escolher um processo para ser terminado
- No Windows: não aceita que a soma de memória exceda máximo
  - Sempre tem memória RAM ou espaço no arquivo de troca