第 5 章

内存管理

在现代操作系统中，多进程并发已成为操作系统的基本特征之一。然而，所有的并发进程都需要被加载进入内存后才能被 CPU 调度执行，这也使得内存管理成了影响操作系统性能的关键。从宏观上来说，内存管理的目标主要包括两个：①确保多个并发进程实现安全高效的内存共享；②提高内存利用率和内存寻址效率。

为了实现这两个目标，现代操作系统中采用了众多的内存管理技术，主要包括：①引入虚拟内存使进程对内存地址的访问从直接变为间接，有效地实现了进程地址空间的隔离，增强了进程管理的安全性和灵活性；②引入分页机制，实现了细粒度的动态内存分配和管理，有效减少了内存碎片，提高了内存利用率；③通过 TLB(地址转换旁路缓存)和多级页表等机制，实现了内存的快速寻址，提升了内存寻址效率；④突破物理内存容量的限制，利用外存对物理内存进行扩充，使得实际内存需求量大于剩余物理内存容量的进程依然能在操作系统中顺利运行。本章将针对这些内存管理的关键技术展开详细介绍，并以 openEuler 中的具体实现为例，加深读者对相关技术的理解。

5.1 内存访问：从直接到间接

内存管理的首要任务便是解决并发进程的内存共享问题。本节将详细介绍内存管理中引入的虚拟内存概念，以及如何通过虚拟内存的间接访问机制解决进程地址空间的隔离问题。

5.1.1 程序中的内存访问

首先以一个简单的例子说明 CPU 如何从内存中取指令和数据来执行一个程序。一个简单的 C 语言程序如图 5-1 所示：程序中定义了一个变量 x，然后对该变量 x 进行加 1 操作。该 C 语言程序编译生成的汇编代码(ARMv8 架构)如图 5-2 所示。操作系统将该程序加载到内存后，进程开始执行。操作系统将设置 C 语言的运行环境(例如设置栈指针)，再通过设置程序计数器 PC(Program Counter)跳转到 main()函数的起始地址 0x06e4(实际运行时，此地址会加上一个偏移)。CPU 中的控制逻辑将使用程序计数器 PC 记录的地址去内存中取回指令交由 CPU 执行部件执行。

```
1.    int main(){
2.        int x = 0;
3.        x = x + 1;
4.        return 0;
5.    }
```

图 5-1　C 语言程序内存访问示例

```
1.    06e4 <main>:
2.    d10043ff    sub sp, sp, #0x10
3.    b9000fff    str wzr, [sp, #12]      //x = 0
4.    b9400fe0    ldr w0, [sp, #12]       //将变量 x 的值读入寄存器 w0
5.    11000400    add w0, w0, #0x1        //w0 = w0 + 1
6.    b9000fe0    str w0, [sp, #12]       //将 w0 寄存器中的值写入变量 x
7.    52800000    mov w0, #0x0
8.    910043ff    add sp, sp, #0x10
9.    d65f03c0    ret
```

图 5-2　C 语言程序对应的汇编代码

3.1.1 节提到过,程序的局部变量保存在栈中。CPU 在执行图 5-2 中第 3 行指令时,将向栈中起始地址为 (SP)+12 处的 4 字节写入整型数据 0(对应于 C 程序中的“$x=0$”操作)。其中,(SP)指取 SP 寄存器中的值,(SP)+12 为变量 x 的地址。在 CPU 执行完第 3 行指令后,程序计数器 PC 将自动递增,指向下一条指令(第 4 行)。随后,第 4 行指令将变量 x 的内容读入寄存器 w0 中;第 5 行指令将 w0 寄存器加 1 再放入 w0 中;第 6 行指令将寄存器 w0 中的值写到栈中地址为(SP)+12 的位置。

从程序的角度来看,CPU 执行该程序发生了以下几次访存:

(1) 从地址 0x06e4 开始,CPU 将依次从内存中读入 2～9 行的指令执行;

(2) 在执行第 3、4 和 6 条指令时,分别向内存(变量 x 所在地址)写入 0、从内存读入变量 x 的值以及向内存写入执行“$x+1$”操作后的结果。

5.1.2　虚拟内存

内存(Memory)是 CPU 能直接寻址的存储器,其以字节为单位存储信息。为了使 CPU 能够正确地把信息存放到内存或从内存取得信息,内存的每一个字节单元被分配了一个唯一的存储器地址,称为物理地址(Physical Address)。程序只有被加载到内存后,CPU 才能从内存中读取程序指令和数据。在早期的计算机中,程序中访问的内存地址都是实际的物理地址。如果计算机只需要运行一些事先确定好的程序,这种直接访问物理地址的方式能够满足用户对内存管理的需求。现在,在嵌入式系统中,这种方式依然很常见,这是因为:在洗衣机、烤箱这样的嵌入式设备中,所有运行的程序都是事先确定的,用户不需要在这些设备上自由地运行其他的程序。

然而，在智能手机、服务器等设备中，用户希望在任意时刻都能自由地运行多个程序。在这些场景下，将物理地址直接暴露给程序存在以下问题：由于程序都是直接访问物理地址，所以给恶意程序提供了随意地读取和修改其他程序甚至操作系统内存数据的可能。即使对于非恶意的程序，如果程序存在漏洞，也可能不小心修改了其他程序甚至操作系统的内存数据，将导致程序运行出现异常或严重的系统安全问题。此外，如果某一个程序在运行时发生崩溃，可能导致整个系统崩溃。例如，在图 5-3 所示的例子中，在向地址 0xF000FF80 开始的 4 字节（假定在 32 位系统中）写入数据 65 535 时，如果该地址是物理地址，并指向了另一个进程代码段所在的内存，那么将可能导致另一个进程无法正常执行。

```
1.     *(unsigned int *)0xF000FF80 = 65535;
```

图 5-3　直接内存访问示例

为了解决上述问题，操作系统设计者想到了一种基于中间层的方法：在内存访问的过程中增加一个中间层，程序通过间接的方式访问物理地址。在这种方式下，程序访问的内存地址不再是实际的物理地址，而是一个虚拟出来的地址；在发生内存访问时，系统负责将这个虚拟出来的地址转换成对应的物理地址。这个增加的中间层叫作虚拟内存，而这个被虚拟出来的内存地址称为虚拟地址（Virtual Address）。此外，人们为内存提供一个易于使用的抽象接口：地址空间。地址空间是一个进程可寻址的一套地址的集合，包括虚拟地址空间和物理地址空间。

openEuler 的虚拟地址空间布局如图 5-4 所示，虚拟地址空间被分成了用户空间、内核空间以及不可访问区域。其中，用户空间、内核空间分别分布在地址空间的低位和高位，各拥有 512GB（2^{39}B）的地址空间，寻址范围分别为 0x0000_0000_0000_0000～ 0x0000_007F_FFFF_FFFF 和 0xFFFF_FF80_0000_0000～0xFFFF_FFFF_FFFF_FFFF。

（1）用户空间包含了进程的所有内存状态。由于在编译时，代码段与数据段的大小就已经固定，在运行时不会动态扩缩，它们被放置在低地址处。在程序运行时，由于栈和堆的大小会动态变化，因此，堆和栈被放置在高地址处，并约定栈向低地址方向增长，堆向高地址方向增长。

（2）不可访问区域。openEuler 并没有使用 64 位虚拟地址空间的全部（通常使用 48 位或 39 位虚拟地址空间），因此除用户空间与内核空间外，还存在一块未用区域，称为不可访问区域。当进程访问此区域时，硬件将触发一个异常。

（3）内核空间为操作系统内核运行的地址空间。为了保证内核的安全，现代操作系统通常将内核空间与用户空间分开，并对用户空间和内核空间的访问进行权限控制。

操作系统引入虚拟内存带来了以下两个好处。

（1）在单一的物理内存上，为每个进程提供了拥有全部内存的假象。每个进程都可以自由地访问用户空间所在的地址范围：0x0000_0000_0000_0000～0x0000_007F_FFFF_FFFF。

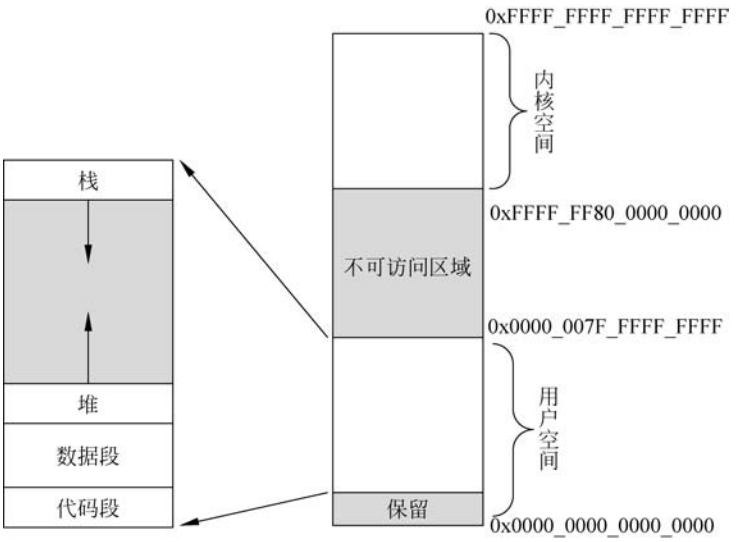

图 5-4　openEuler 虚拟地址空间布局示意

(2) 为进程之间的隔离提供了基础。引入虚拟地址后,进程的内存访问需要经历虚拟地址到物理地址的转换过程。在这个过程中,系统就可以做“手脚”:在每次进行地址转换时,检查进程访问的物理地址是否合法(例如,检查该物理地址是否已被分配给该进程),从而确保各个进程之间不会互相读写,达到进程隔离的目的。

地址空间的引入带来了诸多好处。然而,这个增加的中间层给系统带来了额外的工作:系统需要负责为进程实现从虚拟地址到物理地址的转换。那么,新增的地址转换工作会不会使得程序的运行显著变慢?在设计地址转换机制时,为了实现高效的地址转换,在硬件、操作系统层面应该如何考虑?在进一步介绍内存管理机制提升地址转换效率的关键技术之前,下面先介绍内存管理中的分页机制,即操作系统如何为进程动态分配并管理地址空间。

5.2　分页

如何为用户进程分配合适的内存空间,以充分利用计算机系统的物理内存资源,是内存管理需要解决的关键问题之一。本节将讨论内存管理中的分页机制原理,以及基于分页机制的地址转换与访问控制问题。

5.2.1　基本思想

在操作系统中,对于地址空间的管理,一种方法是将地址空间划分成不同长度的分区,每个进程被加载到其中一个分区中运行。这种方法实现较为简单,但存在严重的内存碎片

问题，使得内存的利用率不高。下面通过一个具体的例子说明产生内存碎片的原因。假定在某个时刻，物理地址空间的布局如图 5-5 的左半部分所示。此时，若需要加载另一个进程 D(假设需内存 50KB)，操作系统将从 64KB 的空闲区为其分配内存。由于所分配的空闲区长度比进程 D 所需求的长度大，操作系统将该空闲区分割成两个部分，一部分成为已分配区，而另一部分成为一个新的小空闲区，如图 5-5 的右半部分所示。随后，如果需要再加载一个所需内存容量为 20KB 的进程 E，由于当前已不存在一个 20KB 的连续空闲区，进程 E 将无法正常加载执行。也就是说，在这样的内存管理方式下，即使某个新进程所需要的空间长度(20KB)小于空闲空间的总长度(16KB＋14KB)，该进程也无法运行。这些尚未分配出去、但又无法分配给新进程的空闲内存块称为外部碎片。在这种地址空间管理(内存管理)方式下，外部碎片产生的原因在于：将空间分割成不同长度的分区后，操作系统为进程分配内存时必须分配一片连续的内存。为了减少外部碎片，提高内存利用率，必须寻求一种不依赖连续内存分配的地址空间管理方式。

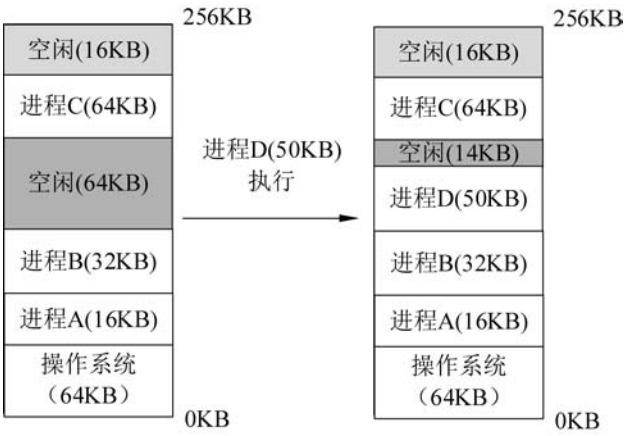

图 5-5　分区与外部碎片

当前大多数操作系统采用分页的地址空间管理方法，它的基本思想是：将进程的虚拟地址空间分割成固定长度的单元(通常取 4KB)，称为页(Page)；将物理地址空间也分割成固定长度的单元，称为页框(Page Frame)；页与页框的长度相等。在这种内存管理方式下，进程在被装入内存时，不再以整个进程为单位，而是以页为单位，所以进程不再需要存放在一块连续的物理地址空间，而可存放在物理地址不一定连续的页框中。

图 5-6 展示了基于分页的进程地址空间布局。在这个例子中，物理内存拥有 8 个页框，页的大小设为 4KB。为了便于读者理解，此处以一个需要 16KB 地址空间的进程为例。该进程的虚拟地址空间需要 4 个页。操作系统在分配内存时，将页框 1 分配给页 0、页框 5 分配给页 1、页框 3 分配给页 2、页框 7 分配给页 3。其他页框未分配，处于空闲状态。

为了记录每个页所分配的页框，操作系统使用一个表来为每个进程记录其页号(Page Numbers)与页框号(Page Frame Numbers)的映射关系，这个表称为页表。每个进程都拥有一个自己的页表。例如，上述虚拟空间大小为 16KB 的进程的页表可表示为如表 5-1 的结构。

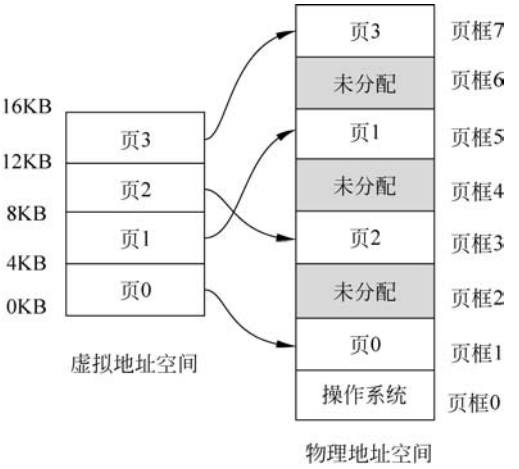

图 5-6　基于分页的进程地址空间布局

表 5-1　页号到页框号的映射关系

页号	页框号	页号	页框号
0	1	2	3
1	5	3	7

分页机制带来了以下两个好处。

(1) 减少了内存碎片。在分页机制和虚拟地址的联合作用下,内存分配过程中的外部碎片将大大减小,甚至不存在外部碎片。而对于分页机制中引入的内部碎片,因为分页在内存分配时最小单位为一个页框,所以任一内部碎片都会小于一个页框的大小。下面通过一个例子来说明内部碎片的定义:假设一个进程需要 102KB 内存。在 4KB 分页的情况下,操作系统需要为其分配 25+1 个页框。第 26 个页框 4KB 的空间只存储了 2KB 的进程内容。这个页框内未使用但又无法分配给其他进程使用的空间被称为内存碎片。尤其在进程普遍需求大内存(几十兆字节以上)时,页式内存分配带来的内部碎片相对连续内存分配带来的外部碎片来说是要小很多的。

(2) 页共享。不同的进程可以将虚拟地址映射到同一个页框,实现页的共享,以减少重复内容的存储。例如,多个进程可以选择映射同一段可重定位的代码。

5.2.2　空闲页框管理

当进程被加载到内存中执行时,操作系统需要为其运行分配物理内存。因此,操作系统需要记录并管理目前计算机系统中空闲的页框。最简单的页框管理方式是:对于未分配的页框,操作系统通过维护一个空闲页框链表进行管理;当进程请求分配页框时,操作系统从链表头取下空闲页框分配给进程。图 5-6 中的空闲页框可组织为如图 5-7 的左半部分所示的结构。此时,若进程向操作系统申请一个页框,操作系统将链表头的页框 2 分配给进程。此后,空闲

页框链表的结构如图 5-7 的右半部分所示。

图 5-7　空闲页框链表

在这种页框管理方式下，在一段时间后，经过频繁的页框申请与释放，空闲页框的分布可能变得分散、零碎。这些空闲页框的总容量可能足够大，但地址不连续。虽然大多数用户进程不需要分配物理上连续的页框，但在一些场景中，内核需要分配物理上连续的页框。例如，在管理与物理内存直接交互的 DMA 硬件设备时，操作系统通常需要为其分配一大块连续的物理内存。DMA 设备在进行数据传输时，可以绕过 CPU，将数据直接发送到指定的物理内存。在对 DMA 设备进行初始化时，操作系统需要为 DMA 设备分配一块连续的物理内存，并指定这块内存的起始地址和长度，以便在数据传输时，DMA 控制器能够自动定位目标物理内存。

解决上述问题的有效方式之一就是 buddy(伙伴)系统。buddy 系统的基本思想是：在分配页框之初就将页框以连续内存的形式组织起来；在页框分配时尽可能按所需连续页框的数目分配对应数量的页框；在页框回收时尽量将页框合并为连续的页框块。物理内存被分为包含多个(2 的幂次方个)连续页框的块。当申请内存的时候，buddy 系统首先去寻找与所申请的内存大小相匹配的空闲页框块。若找到，则直接返回页框块的首地址；若没有相匹配大小的空闲页框，则取更大的页框块(2^i 大小)分割成两个大小为 2^{i-1} 的子块，使用前一个子块进行分配。buddy 系统将两个大小为 2^{i-1} 的子块称为伙伴。当发生页框块的回收时，buddy 系统将检查其伙伴块是否为空闲状态：若空闲则将两者合并为一个更大的页框块，否则直接回收该页框块。

下面将结合 openEuler 中的代码，阐述 buddy 系统的实现。

1. 重要的数据结构

buddy 系统将连续的空闲页框组织成一个空闲页框块，并根据连续页框的数量进行分组和记录。如图 5-8 所示，结构体 zone 定义了 11 个空闲页框链表来记录具有相同连续页框数的页框块；结构体 free_area 定义了链表头和记录空闲页框块数量的成员 nr_free。如图 5-9 所示，第 i 个链表记录所有由 2^i 个连续页框组成的块：11 个链表分别记录大小为 1、2、4、8、16、32、64、128、256、512 和 1024 个连续页框组成的页框块。

```
1.    //源文件: include/linux/mmzone.h
2.    struct zone {
3.        ...
4.        //#define MAX_ORDER 11
5.        struct free_area  free_area[MAX_ORDER];       //空闲页框块链表
6.        ...
7.    };
8.
9.    struct free_area {
10.       struct list_head  free_list[MIGRATE_TYPES];  //链表头
11.       unsigned long     nr_free;                   //空闲页框块数
12.   };
```

图 5-8　空闲页框块组织的结构体定义

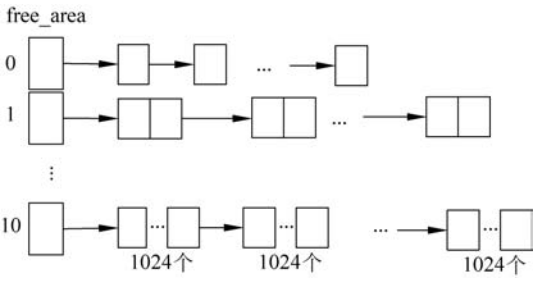

图 5-9　空闲页框块的链表组织图

2. 伙伴关系的定义

在 buddy 系统中,2^i 个连续页框块被称为 i 阶(order)页框块。buddy 系统将满足以下条件的两个 i 阶页框块定义为伙伴关系:①两个页框块的大小相同并且物理地址连续;②将两个 i 阶的页框块合并为一个 $i+1$ 阶的页框块后,页框块中第一个页框的编号必须为 2^{i+1} 的整数倍。以 0 阶页框块为例,页框 0 和页框 1 是伙伴关系,但是页框 1 和页框 2 不是伙伴关系。这是因为页框 1 和页框 2 合并为一个 1 阶页框块后,页框号 1 不是 2^1 的整数倍。

在 openEuler 中,函数__find_buddy_pfn()用于快速地获得页框块的伙伴块号。函数__find_buddy_pfn()的实现如图 5-10 所示。第 4 行代码使用页框块的第一个页框的编号与 1 左移 order 位(order 指该页框块所对应的阶)的结果相“异或”(“^”代表异或操作),可得到该页框块的伙伴块。例如,假设某个 1 阶页框块由页框 8 和页框 9 构成,因为 8^(1<<1)=8^2=10,该页框块的伙伴为由页框 10 和页框 11 构成的页框块。

```
1.    //源文件: mm/internal.h
2.    static inline unsigned long __find_buddy_pfn(unsigned long page_pfn,
3.                                                 unsigned int order) {
4.        return page_pfn ^(1 << order);
5.    }
```

图 5-10　函数__find_buddy_pfn()

3. buddy 系统初始化

操作系统在初始化时,会把连续的页框组织起来,并加入 buddy 系统中。将页框加入 buddy 系统的实现如图 5-11 所示。其中代码第 5～6 行获取物理内存对应的起始和结束页框号。第 10 行获取相邻 64 个页框的空闲状态。如果起始页框号是 64 的倍数并且连续的 64 个页框都可用(第 11 行),则调用函数__free_pages_bootmem()将该 64 个连续页框加入到 buddy 系统中(第 15 行),并继续尝试将下一个连续的 64 个页框加入到 buddy 系统(第 18 行);若不满足要求,则将页框一个一个地加入到 buddy 系统中(第 22 行)。特别地,在

函数__free_pages_bootmem()的实现中，每新加入一个页框块(包括单个页框)，buddy 系统都会递归地检查该页框块的伙伴是否存在。如果存在，则合并伙伴块构成一个更大的页框块。例如，若连续两次加入 64 个页框构成的 6 阶页框块到 buddy 系统中，在第二个页框块加入时，buddy 系统判断其伙伴存在，则将两个 6 阶页框块合并为一个 7 阶页框块 A。此时，如果再连续加入两个 6 阶页框块，这两个 6 阶页框块首先将合并为一个 7 阶页框块 B。随后，7 阶页框块 B 将与 7 阶页框块 A 合并为一个 8 阶页框块。

```
1.    //源文件：mm/bootmem.c
2.    static unsigned long __init free_all_bootmem_core(
3.                                                    bootmem_data_t * bdata) {
4.        ...
5.        start = bdata->node_min_pfn;   //起始页框号
6.        end = bdata->node_low_pfn;     //结束页框号
7.        while (start < end) {
8.            ...
9.            //变量 vec 标识连续的 64 个页框块中可用的页框块数
10.           vec = ~map[idx / BITS_PER_LONG];
11.           if (IS_ALIGNED(start, BITS_PER_LONG) && vec == ~0UL) {
12.               //连续 64 个页框都可用  BITS_PER_LONG = 64
13.               int i = ilog2(BITS_PER_LONG);
14.               //将 64 个页框加入到 buddy 系统中
15.               __free_pages_bootmem(pfn_to_page(start), start, order);
16.               count += BITS_PER_LONG;
17.               //继续尝试下一个连续的 64 个页框
18.               start += BITS_PER_LONG;
19.           } else {
20.               ...
21.               //64 个页框块中有部分是不可用的，尝试一次添加一个页框到 buddy 系统中
22.               __free_pages_bootmem(page, cur, 0);
23.               ...
```

图 5-11　将页框加入 buddy 系统的实现

4. 空闲页框块的分配

当收到 $n(n=2^i, i=0,1,2,\cdots)$个页框请求时，buddy 系统中的处理如图 5-12 所示。

(1) 尝试在 free_area 的第 i 个链表中分配空闲页框块。若有，则将其从空闲链表中移除并直接分配出去。

(2) 若没有空闲页框块，则再查找第 $i+1$ 个链表是否有空闲页框块(第 11～12 行)。若有，则分配 2^i 个页框(第 13～15 行)，余下的 2^i 个页框将插入到第 i 个链表中(第 17 行)。

(3) 若第 $i+1$ 个链表中依然没有空闲页块，则重复步骤(2)，直到找到有空闲页块的链表。假设在第 $i+x(x=2,3,\cdots)$个链表中找到空闲页框块，则把此链表中的第一个空闲页框块取出，分成两个相同大小的子空闲页框块，称为 L 和 R 伙伴页框块。buddy 系统将 R 子页框块插入第 $i+x-1$ 个链表中。此时，L 子页框块的大小大于所请求的页框大小。因

此,buddy 系统将把此 L 子页框块减半,其中一个加入到更小一级的链表中,再递归地考虑对另一半进行分配。例如,假设 $i=2$、$x=2$,此时申请的页框块大小为 4 个页框,L 子页框块大小为 $8(2^4/2)$个页框,此时需要对 L 子页框块再次减半后得到两个 4 个页框,其中一个页框块用于分配,另一个则插入到第 2 个链表中。

(4) 如果找到最后一个链表都没有空闲内存(for 循环结束),则返回空值,表示内存分配失败(第 21 行)。

```
1.    //源文件: mm/page_alloc.c
2.    struct page * __rmqueue_smallest(struct zone * zone,
3.                  unsigned int order, int migratetype) {
4.        ...
5.        //在最佳的链表中寻找一个合适大小的空闲页框块
6.        for(current_order = order;current_order < MAX_ORDER;++current_order){
7.            area = &(zone -> free_area[current_order]);
8.            //从当前的 free_area 中获取一个页框块
9.            page = list_first_entry_or_null(&area -> free_list[migratetype],
10.                                struct page, lru);
11.           if (!page)
12.               continue;            //没有合适的页框,到更高一级的 free_area 中查找
13.           list_del(&page -> lru);  //取消链表的链接
14.           rmv_page_order(page);    //将该空闲页框块从 order 中移除
15.           area -> nr_free -- ;     //空闲页框块数量减 1
16.           //将剩余的页框扩展到低级的 free_area 中
17.           expand(zone, page, order, current_order, area, migratetype);
18.           ...
19.           return page;
20.       }
21.       return NULL;
22.   }
```

图 5-12　空闲页框块的分配

例如,假设内核要分配 $8(2^3)$个页框,buddy 系统会先从第 3 个链表中查询是否有空闲页框块。如果有空闲块,则从中分配,如果没有空闲块,就从它的上一级(第 4 个链表)中取出一个包含 16 个空闲页框的块,从中分配出 8 个页框,并将多余的 8 个页框加入到第 2 个链表中。如果第 3 个链表中也没有空闲页框块,则依次向上查询,直到找到存在的空闲页框块,并将剩余的空间页框块递归地插入下一级链表中。如果第 10 个链表中都没有空闲页框块,则返回内存分配失败信息。

5. 页框块的回收

页框块的回收是分配的逆过程。当一个页框块被内核或进程释放时,buddy 系统将检查其伙伴块是否在该页框块所属的空闲链表中。如果其伙伴块不在空闲链表中,将直接把要释放的页框块插入到对应等级的空闲链表中。如果其伙伴块在空闲链表中,则取出其伙伴块,将这两个页框块合并为一个更大的页框块。随后,进一步检查上一级链表中该合并块

的伙伴块是否处于空闲状态，直到不能再进行合并或者已经合并至最大的块。这个过程与 buddy 系统初始化过程中页框块的合并是一致的。

6. buddy 系统的优缺点

buddy 系统的优点在于原理较简单，外部碎片比较少并且可分配连续的物理内存。但 buddy 系统有一个很明显的缺点：内部碎片问题比较严重。buddy 系统按 2 的幂次方大小进行内存分配，就算请求的内存大小为 $n(2^i < n < 2^{i+1})$ 个页框也要按 2^{i+1} 个页框进行分配。同时，在页框块回收时，buddy 系统仅考虑与伙伴块合并，这使得有些相邻的、大小相同的空闲块没有合并，降低了内存的利用率。

5.2.3　地址转换

地址转换是间接地址访问方式下的关键问题。那么，在分页机制下，系统如何实现虚拟地址到物理地址的转换？

1. 虚拟地址

在介绍地址转换前，先介绍虚拟地址的结构。虚拟地址的结构与页的大小及虚拟地址空间的大小有关。假设每个页的大小为 4KB＝2^{12}B，虚拟地址空间大小为 4MB＝2^{22}B，则虚拟地址空间可划分为 2^{10} 个页。在这种情况下，虚拟地址的结构示意如图 5-13 所示，其中，低 12 位(bits[11:0])表示页内偏移地址，高 10 位(bits[21:12])表示页号。

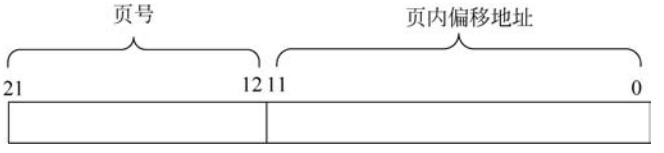

图 5-13　虚拟地址结构示意

2. 页表

在页表较小的情况下，一个页框可存储整个页表。因此页表一般存储在一个地址连续的内存中，且能随机访问，以快速查找页表中相应的记录。系统可通过页号索引线性页表获得页框号(类似于 C 语言中通过数组下标获得数组项的内容)，因此在页表中可不存储页号。一个简单的单级页表结构如图 5-14 所示，每个映射关系的记录被称为页表项(Page Table Entry，PTE)。PTE 的大小与处理器理论上所支持的最大物理内存的大小和页大小有关。例如，当处理器最大可支持的物理内存为 512GB、页大小为 4KB 时，内存被分为 2^{27}(512GB/4KB)个页框，因此 PTE 需要 27 位(bit)来记录页框号。通常，64 位操作系统系统用 8B 空间保存 PTE，因为除了保存页框号之外，还增加一些标志位用来辅助内存访问的权限控制等功能的实现。在 AArch64 中，PTE 被称为页描

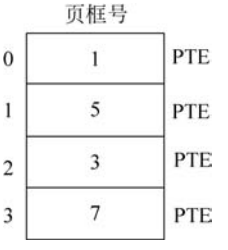

图 5-14　简单的页表结构

述符(Page Descriptor)。如图 2-29 所示,页描述符指定了 PTE 的大小和 PTE 中每个位代表的含义:bits[38:12]输出地址字段用来记录该 PTE 所映射到的页框号;bits[63:51]和 bits[11:2]分别定义了 PTE 的上部(Upper)和下部(Lower)两个属性字段,用来实现内存的访问权限控制(见 5.2.4 节)。

3. 地址转换过程

操作系统借助页表实现虚拟地址到物理地址的转换。假设进程的页表如图 5-14 所示,下面以一个访存指令"ldr x0, 4100"为例,分成 4 步来说明虚拟地址到物理地址的转换过程。

(1) 从虚拟地址中得到页号及偏移:页号=虚拟地址/页大小。页号为 4100/4096,即 1;偏移为 4100%4096("%"代表取余操作),即 4。

(2) 利用页号及页表基址寄存器获得 PTE 的物理地址。在地址转换的过程中,为了获得页与页框的映射关系,系统首先需要获取 PTE 的物理地址。硬件提供页表基址寄存器来保存页表的基址。基于页表基址寄存器,系统通过以下计算获得 PTE 的物理地址:PTE 的物理地址=基址寄存器中的地址+页号×sizeof(PTE)。

(3) 从该物理地址中读取 PTE 的值,获得页框号为 5。

(4) 利用页框号,计算物理地址。物理地址为 5×4096+4,即 20484。最后,硬件读取物理内存 20484 处的内容到寄存器 x0 中。

进程的每一次内存访问都需要进行地址转换。由于需要获得映射关系,每次访存还增加了一次额外的内存引用。因此,上述地址转换的过程如果全部由操作系统负责,将可能显著地降低系统性能。在系统设计中,当使用纯软件的方式不能较理想地解决问题时,软硬件协同是常用的解决方法。下文将分别从硬件和操作系统的角度来阐述其各自在地址转换过程中所承担的职责。

4. 硬件的职责

页表的查询通常由专用的硬件内存管理单元(Memory Management Unit,MMU)完成。以图 5-13 所示的虚拟地址为例,基于 MMU 辅助的地址转换工作流如图 2-27 所示:①CPU 核向 MMU 发送虚拟地址;②MMU 获取该虚拟地址的高 10 位表示的页号,并根据页号以及页表基址寄存器保存的页表基址,获得 PTE 的物理地址;③读取 PTE,获得该页号对应的页框号;④MMU 根据页框号、页大小和虚拟地址中表示偏移地址的低 12 位计算得出物理地址;⑤MMU 将该物理地址发送到地址总线上,进行访存操作。

5. 操作系统的职责

当然,仅仅依靠硬件不足以完成地址转换的全部相关工作,它只是加速地址转换过程。操作系统尚需承担以下三项职责。

(1) 建立页表,保存页表基址。在加载进程时,操作系统为进程分配页框,用于存储页表和进程内容。在此过程中,操作系统应设置 PTE,建立起虚拟地址到物理地址的映射(或称页号到页框号的映射),以便在进程访问某个虚拟地址时可转换为对应的物理地址;再保

存页表基址到某个数据结构中(例如 openEuler 的结构体 mm_struct 的成员 pgd)。

(2) 设置页表基址寄存器。在进程开始运行时,操作系统将页表基址存放到页表基址寄存器中,以保证 MMU 在查询页表时能找到此进程的页表。

(3) 处理访存异常。MMU 在地址转换过程中可能会触发异常,操作系统需要对这种异常进行处理。

5.2.4　内存访问控制

虚拟内存设计的主要目标之一是保护与隔离,即操作系统要控制进程只能以正确的方式访问它自己的内存空间,同时使得进程之间不会互相影响,各进程也不会影响操作系统。为实现进程在内存访问时的保护与隔离,分页机制通过硬件和操作系统协同控制内存访问。

1. 硬件基础

利用硬件来实现内存访问的权限控制是一种常用访问控制方法。因为每次地址转换都需要通过页表查询到 PTE[暂不考虑 TLB(见 5.3.1 节)],所以可以在 PTE 添加一些额外的属性位来实现权限的检查。在 ARMv8 架构中,PTE 中的上部和下部两个属性字段的具体定义如图 2-31 所示。此处介绍与权限控制相关的两个关键属性。

(1) 访问权限(Access Permission,AP),即 bits[7:6]字段,定义了该 PTE 所记录的页框的访问权限。当 AP 字段为 0b00 时,表示在 EL0 下不可读写,在其他异常级别下可读写; AP 字段为 0b01 时,表示所有异常级都可读写; 当 AP 字段为 0b10 时,在 EL0 下不可读写,在其他异常级别下是只读的; 当 AP 字段为 0b11 时,表示在所有异常级别下都是只读的。

(2) 不可执行(Execute Never,XN),即 bit[54]字段,定义了该 PTE 所记录的页框的执行权限。当 XN 值为 1 时,表示任何 EL 的程序都不可从该页框中取指令执行; 为 0 时则表示都可从中取指执行。

MMU 在每次地址转换时,将先查看 PTE 中的各个属性,确定当前地址的访问、执行权限,再判断当前 CPU 所处的异常等级是否有权限访问。一旦有指令不具备访问权限,MMU 就会触发一个权限错误(Permission Fault)的异常,将控制权传递给内核的异常处理程序。

2. 页的访问权限设置

基于以上硬件的支持,操作系统可以实现进程对页的读/写/执行权限的限制。例如,代码段所在的内存通常是可读、可执行但不可写,而数据段可读、可写但不可执行。那么,在将进程加载到内存中时,操作系统可对存储代码段映射关系的 PTE 进行这样的设置: AP 写入 0b11,XN 写入 0b0; 对存储数据段映射关系的 PTE 进行对应的设置: AP 写入 0b01,XN 写入 0b1。

3. 用户空间与内核空间的隔离

进程的地址空间包含用户空间和内核空间。用户进程在运行过程中,可以通过写一个

位于内核地址空间的地址(例如,openEuler 的内核空间范围为 0xFFFF_FF80_0000_0000～0xFFFF_FFFF_FFFF_FFFF)而访问到内核空间。如果操作系统允许这种访问,MMU 将转换该虚拟地址到物理地址来访问内核管理的资源,这对系统来说是十分危险的。

从操作系统设计的目标来说,用户进程不应该读写内核中的代码或数据结构。那么系统可以采用什么方法来实现这个限制呢? 如果 MMU 能预先被"告知"内核空间映射的页框只允许高于 EL1 异常级访问,那么 MMU 就可以拒绝运行在 EL0 上的用户进程对该内存的访问。

为实现这种机制,操作系统需要在硬件支持的基础上,在创建 PTE 时初始化相应的属性位。例如,若当前 PTE 记录了内核地址空间的地址映射关系,操作系统需要将此内核空间中所有 PTE 的 AP 字段初始化为 0b00,从而阻止用户程序对内核地址空间的访问,又保证内核具有读写权限。

在对 PTE 设置之后,操作系统还需实现一个异常处理函数来处理访存异常。若进程尝试对内核空间进行访问或是向自身代码段所在的页框写入数据,硬件将触发异常,并转去执行对应的异常处理函数。通常,访存异常处理函数的实现逻辑是:终止发生异常访问的进程,同时释放该进程使用的相关页框。

4. 进程之间的隔离

进程的页表规定了进程的每个页到页框的映射关系,也就决定了每个虚拟地址所对应的物理地址。在正常情况下,进程通过访问某个虚拟地址是无法访问到其他进程所映射到的物理地址的,因此也就实现了进程之间的隔离。

5.3 更快的地址转换

在地址转换的过程中,为了获取页与页框之间的映射信息,MMU 需要先从存储页表的内存中读取 PTE。由于引入了一次额外的内存访问,基于分页的间接内存访问使得系统性能下降 50%,甚至更多。因此,系统需要引入一种新的机制来加快地址转换过程。当操作系统想要使某个过程更快、更高效时,往往求助于硬件的辅助。

5.3.1 TLB 与局部性原理

为了尽可能降低地址转换带来的性能开销,系统设计者们在 MMU 芯片中增加了一小片称为 TLB(Translation Lookaside Buffer,转址旁路缓存)的硬件模块,用来存放最近访问过的虚拟地址对应的转换映射关系。由于 CPU 对 TLB 的访问速度远大于访问内存的速度,因此,利用 TLB 可有效降低内存访问的时间。当发生地址转换时,硬件先检查 TLB 中是否有期望的转换映射关系:如果有,硬件利用 TLB 中缓存的转换映射关系完成快速的地

址转换；如果没有，硬件再去查找页表，以获得转换映射关系，并更新 TLB。通过这种方式，尽可能避免因读取转换映射关系而导致的额外内存访问。

在 TLB 中，一个条目通常包括如图 5-15 所示的信息：页号、页框号以及标志位（用于标识 TLB 项的各种属性，将在 5.3.2 节详细介绍）。基于 TLB 的地址转换过程如下：①MMU 检查 CPU 发送的内存访问请求，取出该虚拟地址对应的页号；②查找 TLB 表，检查是否存在与虚拟页号匹配的 TLB 项；③若存在，则直接得到该虚拟页对应的页框号，不再需要经过页表查询的过程，这个过程称为 TLB 命中（hit）；④ 若不存在，表示在 TLB 中未缓存该虚拟地址对应的虚拟页号，这个过程称为 TLB 未命中或缺失（miss），在这种情况下，MMU 需要访问页表，以获得该页号所映射的页框号，这会增加一次额外的内存访问；⑤在获得对应的页框号后，MMU 将构建一个新的 TLB 项，将该映射关系添加到 TLB 中，或当 TLB 无存储空间时先淘汰某个 TLB 项再添加新的 TLB 项。当程序接下来对该虚拟地址再次进行访问时，由于在 TLB 中已缓存映射关系，地址转换过程将能够快速地完成。

标志位	页号	页框号

图 5-15　简单 TLB 表项格式

可以注意到，在上述地址转换过程中，如果 TLB 命中经常发生，由于 CPU 对 TLB 的访问速度快，地址转换的性能开销将非常小。如果 TLB 未命中，间接内存访问将显著地影响系统性能。实践表明基于 TLB 的地址转换能够显著地改善地址转换性能。其原因在于：在一般的情况下，由于程序执行的局部性，查询映射关系会以很高的概率在 TLB 中命中。程序执行的局部性体现在时间和空间两个维度。

（1）时间局部性：如果程序中的某条指令被执行，则不久之后该指令可能再次被执行；如果某数据被访问，则不久之后该数据可能再次被访问。

（2）空间局部性：如果在某个时刻程序访问了某个存储单元，则不久之后，该存储单元附近的存储单元也可能被访问。

局部性原理表明：在一个内存地址被访问后，很大可能接下来会被再次访问。因此，通过在 TLB 中缓存此次访问的地址转换关系，避免之后再次访问此地址时发生页表查询，从而显著地改善地址转换性能。

5.3.2　TLB 结构

TLB 缓存了当前运行进程的地址转换关系，这些内容仅对当前正在运行的进程有意义，对其他的进程无效，且不应该被其他进程读取。因此，在发生进程的上下文切换时，需要实现 TLB 中属于不同进程的条目之间的隔离。

一种隔离方式是，每个进程在运行时，单独地占有 TLB。在这种方式下，在发生上下文切换时，需要清空整个 TLB（ARMv8 架构提供 TLBI 指令，可刷新整个 TLB、指定虚拟页或者一个指定地址范围内的所有 TLB 记录）。因此，每个新进程在运行前，面对的都是一个空的 TLB，当它进行内存访问时，就会导致频繁的 TLB 未命中。同时，由于进程切换是一个很频繁的操作，这种方式会带来较大的地址转换开销。

另一种隔离方式是,多个进程共享 TLB,但在 TLB 条目中增加一个字段,用来标识映射关系属于哪个进程。在 ARMv8 架构中,这个新增的字段称为地址空间标识(Address Space ID,ASID)。在 AArch64 下,操作系统可通过配置 TCR 寄存器的 AS(Address Space)字段来控制 ASID 的范围大小(8 位或 16 位)。此外,操作系统会为每个进程分配一个唯一的 ASID,并在进程运行时保存在页表基址寄存器 TTBR0_EL1 的字段 ASID 中,如图 5-16 所示(字段 BADDR 保存页表基址)。在这种方式下,进行地址转换时,MMU 会忽略 TLB 中 ASID 字段与寄存器 TTBR0_EL1 的 ASID 字段不匹配的记录。在进程切换时,操作系统只需要更新寄存器 TTBR0_EL1,不用再刷新整个 TLB 记录。

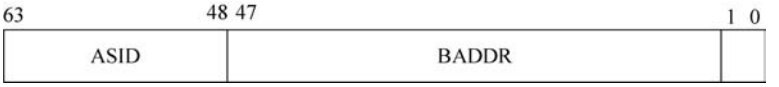

图 5-16　TTBR0_EL1 寄存器的 ASID 字段

同时,为了使各个进程共享内核空间的 TLB 记录,AArch64 再次扩展 TLB 的标记字段:引入非全局位(not Global,nG)区分进程和内核的 TLB 表项。nG 位为 0 表示该记录属于内核,可供任意进程访问;nG 位为 1 表示该记录是进程私有的,只能供对应 ASID 的进程访问。在这种情况下,当判断是否命中 TLB 时,首先比较页号是否相等,再判断该记录是不是全局映射关系。若是全局映射关系,则直接得到 TLB 命中,无须再比较 ASID;若不是全局映射关系,则通过 ASID 字段检查是否为当前进程的 TLB 记录。

在 AArch64 中,具体的 TLB 表项如图 5-17 所示,除了包括页号以及页框号,还包括一些标志位(或称属性):nG、ASID 及虚拟机标识符(Virtual Machine Identifier,VMID)等。VMID 用于区分不同虚拟机的 PTE。

图 5-17　TLB 表项基本构成

5.3.3　TLB 替换

当 TLB 的存储空间(通常可存储 16～512 个表项)耗尽之后,若需要再存储一个新的 TLB 记录,则需要考虑替换一个旧表项。这个表项的选择由 TLB 替换策略来决定。评判替换策略好坏的一个关键点是:是否能最小化未命中率(或提高命中率),从而提高地址转换的性能。

在 ARMv8 架构中,虽然可通过指令刷新 TLB 表项,但架构并没有提供写 TLB 的指令,TLB 的维护完全由硬件来完成,相应的 TLB 替换策略也固定在硬件中。最近最久未使用(Least Recently Used,LRU)是一种常用的 TLB 替换策略,这种策略基于程序的局部性原理,认为最近最久未使用的 TLB 记录是最佳的替换选择。另一种常用的方法是随机法,该策略会随机选择一项 TLB 记录作为替换对象。这种策略实现简单,同时能够避免极端情

况的发生。例如，假设 TLB 可以存储 n 个记录，当程序在一个循环中轮流访问 $n+1$ 个不同的内存块时，LRU 替换策略就会导致 TLB 查询全部失败的情况，但是随机替换策略就可以避免这种情况。

5.4　更小的页表

在较大地址空间下，操作系统为了存储虚拟地址到物理地址的映射关系，给内存带来相对较大的存储开销。本节将重点讨论如何有效地降低页表的内存存储开销。

5.4.1　多级页表

在空间维度，分页机制面临的问题是：存储映射关系的页表会消耗大量的内存。例如，当虚拟地址空间达到 4GB(2^{32}B)，在页大小为 4KB 的情况下，虚拟空间中包括 1M(4GB/4KB)个页。假设在页表中，每条映射关系需要用 4B 空间存储，存储 1M 个映射关系则需要占用 4MB 的空间。为了实现线性表的随机访问，页表又需要连续的内存空间进行存储。在操作系统内存紧张或者内存碎片较多时，这会为系统带来极大的内存负担。考虑到一个进程一般不会用到全部的 4GB 空间，某些 PTE 可能不会被访问到。在这种情况下，这些 PTE 所占用的空间(4B)就白白浪费了。多级页表的基本思想是，将页表分成页大小的单元，只将有效的单元保留在内存中；引入页目录(Page Directory)，用来标记当前目录是否包含有效的单元，以及有效单元所在的位置。页目录中的每个条目称为页目录项(Page Directory Entry，PDE)。如果页目录仍很大，可以在它前面再增加一级索引，成为“页目录的页目录”，这样就构成了多级页表。以上述场景为例(虚拟空间大小为 4GB，页大小为 4KB，且每个 PTE 使用 4B 存储)，图 5-18 展示了它的二级页表结构。

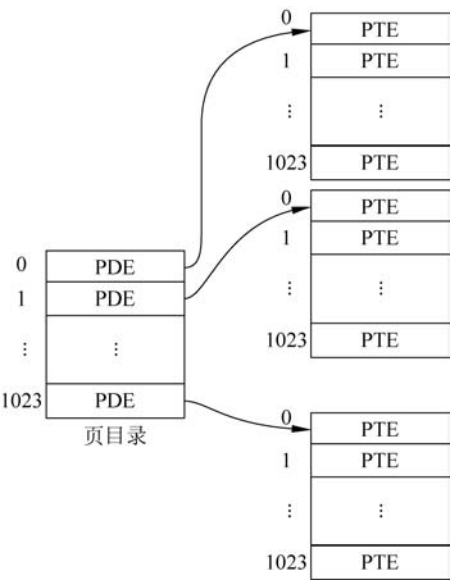

图 5-18　二级页表的结构

在 ARMv8 架构中，PDE 被称为表描述符(Table Descriptor)。如图 2-28 下半部分所示，表描述符指定了 PDE 的大小和 PDE 中每个 bit(位)表示的含义：bits[38:12]保存下一级页表的页框号；bits[63:59]定义了下一级页表的访问权限；bits[58:52]和 bits[11:2]为未使用字段。

下面以一个访存指令“ldr X0，0x080004002”(增大虚拟地址到 32 位)为例，阐述二级页表下虚拟

地址到物理地址的转换过程。0x08004002 转化为二进制的虚拟地址如图 5-19 所示。该虚拟地址的低 12 位(bits[11:0])表示页内偏移地址,高 20 位(bits[31: 12])表示页号(第 32772 页)。区别于一级页表的地址转换,二级页表中的地址转换机制不再是一步到位直接在页表中找到映射关系,而是先由页目录找到相应页表单元,再从页表单元中获得映射关系。由于每个页框可存 1024(2^{10})个 PTE,10 位可以覆盖一个页表单元所保存的记录,因此,索引一个页表单元需要 10 位虚拟地址。再者,页目录需要记录 1024 个页表单元的页框号,索引整个页目录同样需要 10 位虚拟地址。通过这个分析可以发现,实际上高 10 位(bits[31: 22])就是一级索引(页目录索引),为 0x20; 接下来的 10 位(bits[21:12])表示了二级索引(页表单元索引),为 0x04。例如,对上述虚拟地址所对应的第 32772 页来说,存储该页映射关系的页表单元的地址保存在第 32(32772/1024)个页目录项中,并且该页映射关系位于第 32 个页表单元的第 4(32772%1024)项。

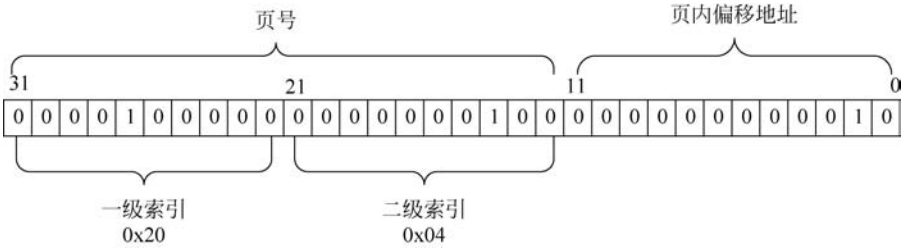

图 5-19　二级页表下虚拟地址结构

操作系统只需要为这个进程保存页目录所在物理页框的基址(例如,openEuler 将全局页目录保存在结构体 mm_struct 的成员 pgd 中),并且在该进程运行时将这个地址放置到页表基址寄存器中,即可在硬件的支持下完成地址转换的过程。在 MMU 的支持下,虚拟地址 0x08004002 到物理地址的转换过程如图 5-20 所示。具体步骤如下:

(1) MMU 取页表基址寄存器中存储的页目录地址作为基址、虚拟地址的高 10 位作为一级索引,读页目录表的第 32 项(以 0 为起始,0x20 代表 32),该 PDE 保存了第 32 个页表单元的物理页号。

(2) MMU 根据该物理页号计算出二级页表的基址,再加上二级索引 4(0x04 代表 4),读取该页框中的第 4 项 PTE,该 PTE 保存了第 32772(1024×32+4)个页对应的页框号。

(3) 由物理页框号乘以页大小(4KB)再加上虚拟地址低 12 位偏移地址得出最终的物理地址。

多级页表的实现方式可只为进程实际使用的那些虚拟地址内存区创建页表信息,从而减少内存使用量。在大虚拟空间情况下,一个用户程序可能并不会使用所有的虚拟地址空间。例如,5.1.2 节提到一个进程的地址空间包括代码段、数据段、堆和栈段,其中堆和栈是动态变化的,在堆和栈之间会存在一个大的、未被使用的地址空间。那么系统可以不存储这一段虚拟内存到物理内存的映射关系,不为这些记录分配保存页表的物理内存。假设堆和栈之间的内存区域的虚拟地址范围对应页目录项 1 和页目录项 2 所映射的范围,那么此时

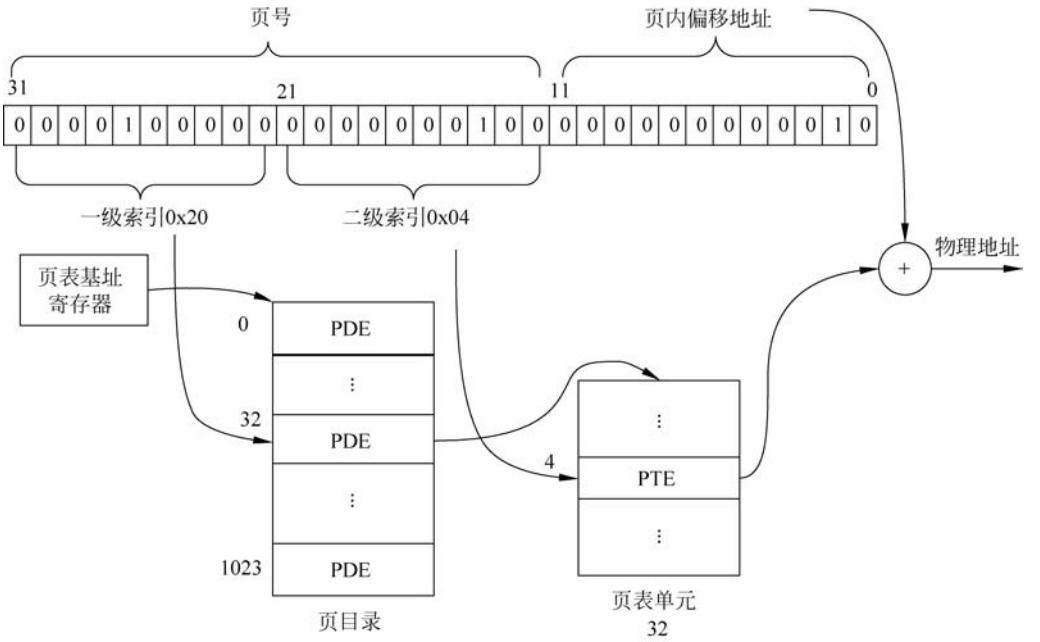

图 5-20　地址转换过程

的二级页表结构如图 5-21 所示。但对于单级页表而言，为了能够随机访问，需要连续的内存空间来存放所有的 PTE，所以就算不记录未使用地址空间的映射，还是要预留所有 PTE 所需的内存。当然，二级页表内存占用最坏的情况是，一个进程用满全部的 4GB 空间。此时，由于分层，会多出一个目录表(4KB)的开销，不过这种概率是极低的。

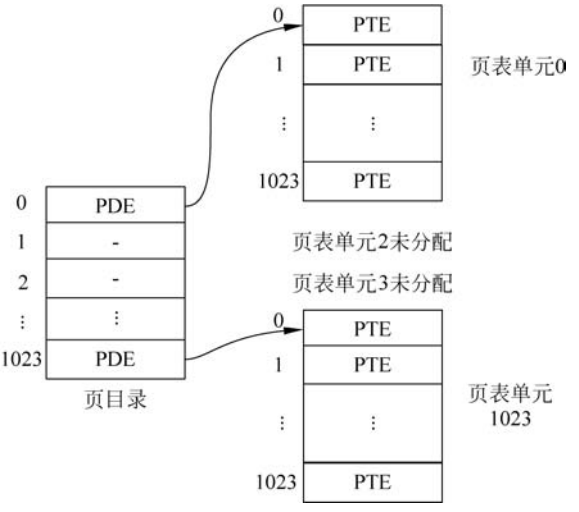

图 5-21　未使用的地址范围不分配页框

5.4.2 openEuler 中的多级页表

ARMv8 架构最大支持 48 位虚拟地址,最大可寻址 256TB 的地址空间。操作系统通过配置寄存器 TCR_EL1 的字段 T0SZ 和字段 T1SZ 可指定实际使用的用户空间和内核空间的大小。在 39 位虚拟地址下,openEuler 可使用 3 级页表(4KB 页),或者 2 级页表(64KB 页)管理内存的映射关系。以下以 39 位虚拟地址、4KB 页大小和 3 级页表为例说明 openEuler 的虚拟地址结构与地址转换过程。

在 openEuler 中,各级页表的表项大小为 8B,在 4KB 分页粒度下,每个页框可保存 512(4KB/8B=2^9,即 512)项记录。9 位可以覆盖一个页框保存的记录,因此,每个页表单元索引或页目录索引占虚拟地址的 9 位。根据之前对多级页表结构的分析,可得到如图 5-22 所示的虚拟地址结构。openEuler 采用了 3 级页表结构,其虚拟内存地址被分成 4 个部分:L1 索引、L2 索引、L3 索引,以及页内偏移地址。

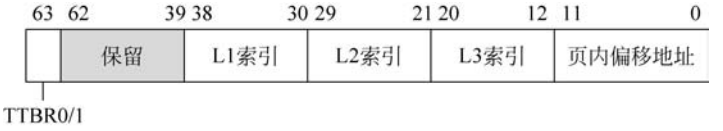

图 5-22 页长度为 4KB,39 位虚拟地址结构

在 ARMv8 架构中,页表基址寄存器 TTBR0_EL1 和 TTBR1_EL1 分别保存当前运行进程用户空间和内核空间的页表基址。在配置用户空间与内核空间位宽为 39 位时,openEuler 的用户空间对应虚拟地址 bits[63:39]为 0,而内核空间的相应位为 1。因此,在 ARMv8 架构中,MMU 使用虚拟地址的 bits[63]决定是对用户空间还是内核空间的访问,从而在地址访问时,选择相应的基址寄存器。

在地址转换的过程中,MMU 将 Lx 索引的值作为 x 级页表内的偏移,据此查询对应的 Lx 表项,得到下一级页表的基址。一级页表,在 openEuler 中称为页全局目录(Page Global Directory,PGD),其基址存储在寄存器 TTBR0/1_EL1 中。PGD 中保存的是表描述符形式的页全局目录项(Page Global Directory Entry,PGDE),指向第二级的页表。第二级页表在 openEuler 中称为页中间目录(Page Middle Directory,PMD)。PMD 中保存表描述符形式的页中间目录项(Page Middle Directory Entry,PMDE),指向第三级页表。第三级页表在 openEuler 中称为直接页表(Page Table,PT),PT 中的 PTE 记录了页框号。通过三级页表查找并计算后,MMU 就可得到虚拟地址对应的物理地址。

5.4.3 标准大页

在用户程序有大内存的需求下,系统仍然采用 4KB 或 16KB 的小分页粒度,将会增加管理的复杂性以及降低程序运行的效率。例如,当用户进程在运行时申请 4MB 的内存,若操作系统以 4KB 作为分页粒度,则需要为用户进程分配 1024 个页框(暂不考虑访问缺页后

的再分配),并在页表中添加 1024 个 PTE(不考虑目录项)记录虚拟地址与物理地址之间的映射关系。当用户进程依次访问这些内存时,系统至少需要经历 1024 次 TLB 缺失和 1024 次的页表查询过程。大量页表表项(任一级页表中的表项统称为页表表项)的管理和存储给系统带来了负担,并且 TLB 缺失后的地址转换是一个耗时的过程,极大降低了程序的执行效率。

解决这个问题的一个方式是,增大分页粒度达到 MB 甚至 GB 级别,这也是标准大页的基本思想。例如,当操作系统采用 2MB 作为分页的基本单位时,在同样用户程序的 4MB 空间申请下,操作系统只需要为其分配 2 个页框,并添加 2 个页表表项来记录映射关系即可。在程序的运行过程中,只会发生 2 次 TLB 缺失以及 2 次页表查询。

要实现大内存分配,最容易想到的方式是直接增大页的大小。但是这样也会导致内存分配的内部碎片增加,并且大页的需求也不是很频繁。因此,在原有分页机制的基础上实现大页是一个有效的方式。大页的实现面临着以下问题:①记录大页映射关系的页表表项应以什么样的结构组织,底层硬件如何处理这种页表表项;②操作系统如何管理大页,并向用户程序提供怎样的接口以使得用户程序方便地使用大页。

1. 块(block)描述符

记录大页的页表表项可直接在原来的三级页表的基础上进行扩展:将 L1 和 L2 级页表中的表描述符形式的表项(指向下一级转换表的基址)作为块描述符形式的表项(指向一块由多个页框组成的连续物理内存)解释。

要理解大页的实现,首先总结一下基本的三级页表下的一些规律。L3 级页表中的 PTE 记录的内容可覆盖 4KB 的内容,这是因为 PTE 记录了物理页框号,而虚拟地址的低 12 位 bits[11:0]可以覆盖 4KB(2^{12}B)的空间,类似起始地址(根据页框号得到起始地址)加偏移地址的概念。那么以此类推,L2 级页表中的 PMDE 记录某个物理页框号(起始地址),并且虚拟地址的低 9+12 位可以覆盖 2^{12+9}B=2MB 的空间;L1 级页表中的 PGDE 可覆盖 2^{12+9+9}B=1GB 空间。在地址转换过程中,可以将 L1～L2 级页表中的一些表项(即表描述符)解释为块描述符(如图 5-23 中的 PGD_block 以及 PMD_block),直接得到物理起始地址(而不去寻址下一级页表),并且使用余下的虚拟地址作为偏移地址来寻址这片空间。例如,在 L1 级页表中存储一个块描述符的表项记录的页框号为 2,该记录转换后的起始地址就为 2GB(2×2^{30}B),代表长度为 1GB 的大页;虚拟地址的 bits[29:0]构成页内偏移地址,在基址上寻址 1GB(2^{30}B)的空间。

ARMv8 架构中的 MMU 硬件支持大页机制下的块描述符。L1～L2 级页表中存在块描述符与表描述符两种形式的表项。MMU 查询页表时需要对这两类描述符进行区分,以决定下一步转换操作:若查找到块描述符形式的表项,则直接进行地址转换。在 ARMv8 架构中,块描述符与表描述符形式的 PDE 格式如图 2-28 所示,bits[1:0]用来指示 PDE 的类型:“01”表示块描述符,“11”表示表描述符;块描述符的 bits[38:n](L1 级页表中的块描述 n 为 30,L2 级页表中的块描述符 n 为 21)保存一个块的基址。在这种支持下,MMU 在地址转换过程中,如果找到一个有效的块描述符形式的 PDE,则可以提前结束页表查询过

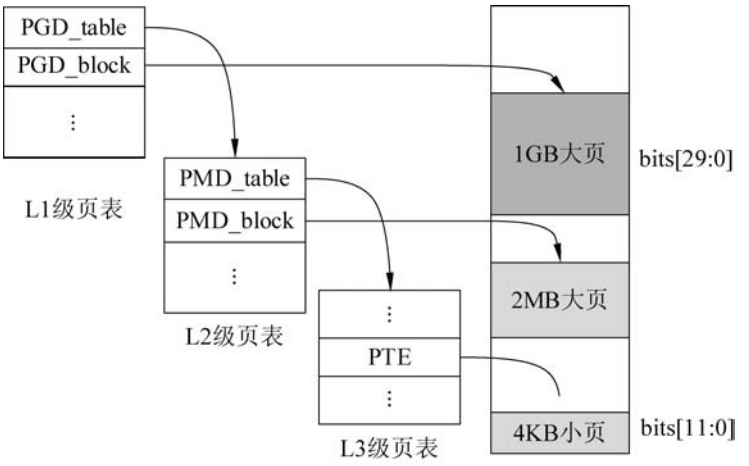

图 5-23　支持大页的块描述符原理

程。这一措施提高了地址转换的效率。同时,一个连续大块的地址映射关系只需要一个表项来存储,降低了页表存储的开销。

2. 标准大页池

在底层硬件的支持下,操作系统已经可以实现大页的虚拟地址到物理地址的映射,下一步是操作系统应该如何记录并管理大页。

openEuler 采用标准大页池(Huge Page Pool)的形式管理大页内存。用户程序在向操作系统申请内存空间时,操作系统就在大页池中寻找空闲可用的大页分配给用户程序。openEuler 在内核启动时,根据用户的配置信息(大页长度、数量等)选择支持的大页长度,并预先在内存空间中预留出相应的大页数量。如图 5-24 所示,openEuler 使用结构体 hstate 记录大页的相关信息(页大小、空闲大页数量等),并将记录相同大页长度的结构体 hstate 组织成一个全局数组 hstates 进行管理。

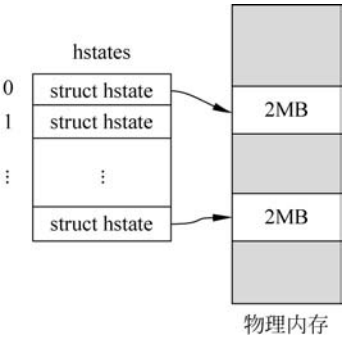

图 5-24　大页的管理结构

那么,操作系统应该提供怎样的接口,使得用户程序可方便地申请大页呢?

3. 伪文件系统:hugetlbfs

openEuler 将标准大页封装为一个伪文件系统(hugetlbfs),提供给用户程序申请并访问。在使用大页前,openEuler 调用函数 hugetlbfs_mount()挂载 hugetlbfs 伪文件系统,创建超级块和根目录,从而将该文件系统和大页内存关联起来。与传统的文件系统指向外存存储空间(例如硬盘)不同,伪文件系统 hugetlbfs 指向大页所分配的内存。

在这种文件系统的抽象下,用户程序可利用简单文件编程接口使用标准大页。hugetlbfs 文件系统的使用例程如图 5-25 所示。用户程序使用 open()函数在 hugetlbfs 文件系统下创

建一个文件(第 3 行),这将触发内核去调用函数 hugetlbfs_create()为该文件创建索引节点(inode)结构;然后,用户程序调用函数 mmap()将该文件映射到虚拟地址空间下(第 5 行),对应地,内核将调用函数 hugetlbfs_file_mmap()为用户进程建立映射;最后用户程序可通过返回的虚拟地址读写标准大页所在的内存(第 6 行)。

```
1.    int main(void) {
2.        //在虚拟文件系统下创建一个文件
3.        fd = open(FILE_NAME, O_CREAT | O_RDWR, 0755);
4.        //将大页内存映射到进程的堆段所处的虚拟地址
5.        void *addr = mmap(0, LENGTH, PROTECTION, FLAGS, fd, 0);
6.        *(int *)addr = 0x42;          //访问大页内存,写入 0x42
7.        ...
8.    }
```

图 5-25　hugetlbfs 文件系统的使用例程

然而,标准大页也存在一些缺点:大页的数量与长度必须在内核启动时就指定,并需要在内存中预先留出大页池所需的内存。同时,用户进程需要主动调用伪文件系统形式提供的接口来申请大页,手动地管理大页的申请与释放,这对编程人员来说是十分麻烦的。

为解决标准大页带来的不便,openEuler 引入透明大页(Transparent Huge Pages)。透明大页的基本思想是:为用户进程分配内存时,操作系统优先选择分配大页,如果不成功(无足够的空闲内存或该虚拟地址区域不允许映射大页内存)再回退到分配普通页。透明大页的实现基于 5.5.1 节讲述的页错误机制。当进程访问某个未映射到物理地址的虚拟地址时,硬件将触发缺页异常,页错误处理程序处理该异常并尝试为其分配物理内存。在遍历页表的过程中,操作系统将依次检查该进程的 L1 和 L2 级页表中的表项是否为空表项。若表项为空表项,虚拟地址范围足够大,并且虚拟内存区域允许映射到透明大页(用户进程可指定某个虚拟内存区域是否允许映射到透明大页),操作系统将尝试为该进程分配一个大页框并建立映射关系;若表项不为空表项或大页内存不足,则再尝试分配普通页。

5.5　物理内存扩充

虚拟内存机制为每个用户程序都提供了接近于系统架构所能支持的最大物理空间容量的虚拟空间,这样编程人员不必担心是否有足够的内存空间来容纳程序,只需按其逻辑来写程序,而操作系统负责根据需要分配内存。但是,多数情况下,系统配置的物理内存容量小于程序的可用虚拟地址空间容量。这就要求操作系统提供一些物理内存的扩充机制,以利用较小的物理内存空间来支持较大的进程虚拟空间。本节将讨论操作系统如何采用一些机制,突破物理内存大小的限制,运行所需内存超过物理内存大小的程序。

5.5.1 请求调页

在之前章节讨论的内存管理与分配方式时,假设操作系统将进程的全部信息都加载到内存中。实际上,在进程运行时,有些数据只用到一次,有些甚至从不使用(条件分支)。操作系统将进程在运行时暂时不用、某种条件下才使用的程序和数据都加载到内存中,是对内存资源的一种浪费,这降低了内存的利用率。那么,系统需要有一种机制,使得操作系统在加载进程时,不必装入进程的全部页,而是仅将当前必须使用的部分页装入内存,在进程运行过程中再根据需要动态地加载其他页到物理内存。

设计这样的机制面临三个问题:①如何判断一个页是否在内存中;②如果一个页所对应的数据不在内存中,而在硬盘上,此时进程访问了不在内存中的地址,硬件或操作系统应该如何协同处理,以将所需的页装入内存;③去哪找到所需的页。解决这些问题的基本思想是:在 PTE 中增加一个新的字段,用来标识该页是否在内存中;如果进程访问到不在内存中的页,硬件将产生一个异常,而操作系统在对应的页错误处理(Page Fault Handler)程序中,将外存(例如硬盘)中相应的页调入内存。这种方式像是用户程序请求操作系统将其需要的页调入内存,因此称为请求调页。

1. PTE 的有效标志位

由于在每次内存访问时,MMU 进行地址转换都需要查询 PTE,因此,PTE 是标识某个页是否在内存中最合适的地方。在 AArch64 架构下,PTE 的 bit[0]定义一个有效(valid)标志位,用来标识该 PTE 是否有效,即该记录所对应的页是否已装入内存。一个无效的 PTE 如图 5-26 所示。MMU 在查询页表时,如果遇到无效的 PTE,就会产生一个转换错误(Translation Fault),交由操作系统继续处理。

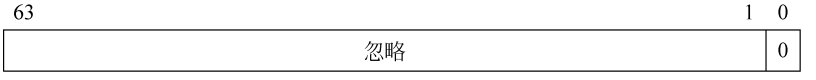

图 5-26 无效的 PTE

2. 页表的初始化

在底层硬件的支持下,操作系统加载进程时要为进程创建页表,此时,仅需为装入内存的部分建立映射关系。例如,假设 openEuler 在加载进程时,只加载从虚拟地址 0 开始的 4KB 内容。如图 5-27 所示,openEuler 需要依次为 PGD、PMD 以及 PT 分配一个页框,并依次初始化相关表项:①将 PGD 的地址(页表基址)保存在结构体 mm_struct 的成员 pgd 中,用来在进程切换时为页表基址寄存器 TTBR0_EL0 赋值;②将 PGD 的第 0 项 PGDE 初始化为表描述符形式(bits[1:0]=0b11),并设置其“输出地址”字段指向 PMD 所在的页框;③将 PMD 的第 0 项 PMDE 初始化为表描述符形式,并将其“输出地址”字段指向 PT 所在的页框;④最后,分配一个页框用来装载进程信息,并将 PT 的第 0 项 PTE 设置为页描述符形式(bits[1:0]=0b11),将其中的“输出地址”字段指向新分配的页框。

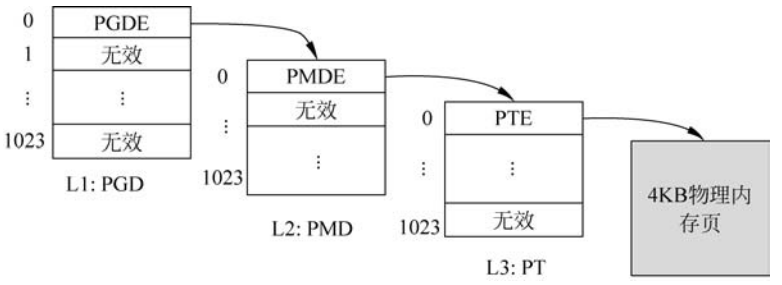

图 5-27　页表的初始化

同时，为了在发生缺页异常时能找到外存中的程序，操作系统还需要采用某种方式记录程序对应的可执行文件的存储位置。openEuler 使用结构体 vm_area_struct 记录一个虚拟地址段的信息。结构体 vm_area_struct 的关键成员定义如图 5-28 所示，包括虚拟地址段的起始地址(vm_start)、结束地址(vm_end)以及映射到的文件(vm_file)等信息。一个进程的所有虚拟地址段的描述组织成一个双向链表(第 8 行)。

```
1.    //源文件: include/linux/mm_types.h
2.    struct vm_area_struct {
3.        ...
4.        unsigned long vm_start;       //虚拟地址段的起始地址
5.        unsigned long vm_end;         //虚拟地址段的结束地址
6.        struct file * vm_file;        //虚拟地址段映射到的文件
7.        //进程的各个虚拟地址段通过链表连接
8.        struct vm_area_struct * vm_next, * vm_prev;
9.        ...
10.   };
```

图 5-28　结构体 vm_area_struct 的定义

在加载进程时，openEuler 根据可执行文件的 ELF 头部信息(代码段以及数据段的大小等)，为进程建立若干虚拟地址段结构体 vm_area_struct，填写其中的起始地址和结束地址，并将程序的可执行文件信息保存在成员变量 vm_file 中。

3. 页错误：硬件的职责

当进程访问一个未建立映射关系的虚拟地址时，MMU 会查到无效的页表表项，并触发转换错误(属于同步异常)。随后，CPU 自动将异常类型以及产生异常的原因分别保存在寄存器 ESR_EL1 的 EC 字段和 ISS 字段(见图 2-32)；触发异常的虚拟地址保存在寄存器 FAR_EL1 中。内存访问的异常分为两种：访问数据时的数据异常和读取指令时的指令异常。在 ARMv8 架构中，每种异常类型都有对应的编号：数据异常和指令异常的编号分别为 0x24 和 0x20。另外，产生内存访问异常的原因除了转换异常之外，还有在 5.2.4 节中涉及的访问权限错误，所以 CPU 还需要将指令异常的具体原因(权限错误、转换错误等)保存在寄存器 ESR_EL1 的 ISS 字段中，以使 openEuler 能对不同的原因做出不同的处理。

在异常信息保存完之后,CPU 将读取寄存器 VBAR_EL1 获得异常向量表地址,再自动跳转到异常向量表偏移地址为 0x400(el0 同步异常)的位置开始执行。

4. 页错误:操作系统的职责

基于上述硬件机制,openEuler 的页错误处理流程如图 5-29 所示。异常向量表的构造以及跳转至 el0_sync 处理函数的过程,在 3.4.2 节介绍系统调用的实现时已经详细描述,这里不再赘述。在 el0_sync 的处理逻辑中,读取寄存器 ESR 的 EC 字段,获得异常类型,以进一步选择指令/数据异常分支。openEuler 的代码实现如图 5-30 所示:第 4～5 行代码取出异常类型;第 7 行代码比较异常类型是否为数据异常,若是,则跳转到数据异常处理函数 el0_da 继续执行;指令异常的处理过程类似。数据异常或指令异常的异常处理过程将进行一些必要的参数(包括异常地址、寄存器 ESR 的值和用户进程各寄存器状态)初始化,以传递给 C 语言函数 do_mem_abort()。

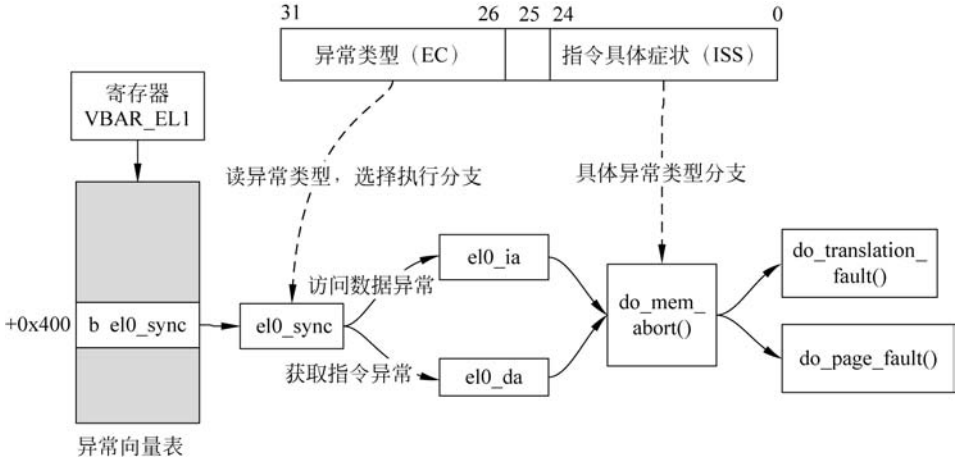

图 5-29 页错误处理流程

```
1.    //源文件: arch/arm64/kernel/entry.S
2.    el0_sync:
3.        kernel_entry 0
4.        mrs x25, esr_el1                //读取 ESR_EL1 寄存器的内容
5.        lsr x24, x25, #ESR_ELx_EC_SHIFT //获取异常类型
6.        ...
7.        cmp x24, #ESR_ELx_EC_DABT_LOW   //数据异常
8.        b.eq  el0_da
9.        cmp x24, #ESR_ELx_EC_IABT_LOW   //指令异常
10.       b.eq    el0_ia
```

图 5-30 异常类型的分支选择

操作系统需要对数据/指令异常的不同原因采取不同的处理逻辑,所以 do_mem_abort()函数将继续区分发生数据/指令异常的具体原因,以选择不同的处理分支。根据 ARMv8 架

构定义的具体异常原因编号，openEuler 定义了一个结构体 fault_info，以方便地根据异常编号找到对应的处理函数。例如，在 L3 级页表中访问到无效的 PTE 时，CPU 将在寄存器 ESR_EL1 的 ISS 字段保存 0x7。基于图 5-31 所示的结构体 fault_info，操作系统将可以直接根据异常信息 0x7 获得处理函数 do_translation_fault()的地址(第 7 行)。操作系统继续调用函数 do_translation_fault()处理此异常，将缺少的页加载进内存中。

```
1.    //源文件：arch/arm64/mm/fault.c
2.    static const struct fault_info fault_info[] = {
3.        ...
4.        {do_translation_fault,SIGSEGV,...,"Level0 Translation Fault"},//4
5.        {do_translation_fault,SIGSEGV,...,"Level1 Translation Fault"},//5
6.        {do_translation_fault,SIGSEGV,...,"Level2 Translation Fault"},//6
7.        {do_translation_fault,SIGSEGV,...,"Level3 Translation Fault"},//7
8.        ...
9.        {do_page_fault, SIGSEGV, ..., "level 1 permission fault"},//13
10.       {do_page_fault, SIGSEGV, ..., "level 2 permission fault"},//14
11.       {do_page_fault, SIGSEGV, ..., "level 3 permission fault"},//15
12.       ...
13.   };
```

图 5-31　异常信息函数调用查找表

在把所缺的数据或代码从外存加载到内存之前，页错误处理程序首先需要进行一些必要的页表设置操作。为了清晰地说明缺页时操作系统的页错误处理流程，假设当前进程访问的虚拟地址为[2MB：2MB＋4KB)。一个 PTE 大小为 8B，一个页框可存储 512(4KB/8B)个 PTE，则一个页框所存储的 PTE 可覆盖 2MB(512×4KB)的内存。因此，地址[2MB：2MB＋4KB)的映射关系对应 L2 级页表(PMD)的第 1 项和 L3 级页表(PT)的第 1 项。页错误处理程序会根据产生页错误的虚拟地址遍历页表，在遍历到 L2 级页表的第 1 项时，发现页表单元 1 未存在于内存中。因此，页错误处理程序需要完成如图 5-32 所示的第(1)步：分配一个页框用作 L3 级页表单元(序号为 1)，并基于其地址初始化 L2 级页表的第 1 项 PMDE。值得一提的是，以上这个发生页错误的地址比较低，页错误处理过程未涉及对 L1 级页表的设置，但实际上当虚拟地址大于 1GB 时，页错误处理程序就需要对 L1 级页表进行设置，并依次为 L2 和 L3 级页表分配一个页框等操作。这个过程与进程加载时的页表内存分配和页表表项设置的过程类似。

之后，openEuler 将根据该可执行程序的文件名，从文件系统中把所缺的内容读入物理内存。不同文件系统的文件存储格式可能不同，因此发生页错误后的文件读取方式也不同。openEuler 采用 ext4 文件系统管理文件，当发生页错误后，系统将调用函数 ext4_filemap_fault()来读取文件内容。函数 ext4_filemap_fault()的实现如图 5-33 所示。该函数首先根据可执行文件名获得记录该文件信息(文件数据块地址、文件的大小和文件拥有者等)的 inode 节点，并对该 inode 节点进行加锁，限制其他程序的读写。然后调用函数

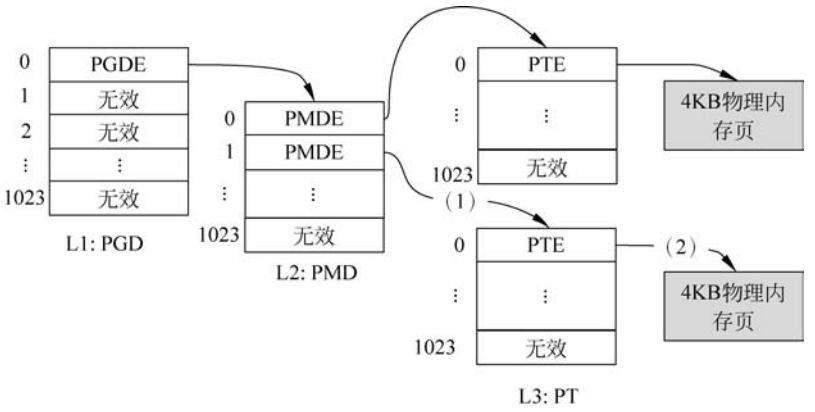

图 5-32　缺页时页表的建立

filemap_fault()。函数 filemap_fault()根据虚拟地址相对虚拟地址段起始地址(vm_start)的偏移获得所缺内容在可执行文件中的偏移,为待加载的代码或数据分配一个页框,将所缺内容装入此页框。最后,页错误处理程序将完成图 5-32 中的第(2)步:设置该虚拟地址对应的 PTE,使其指向存储缺页内容的页框。

```
1.    //源文件: fs/ext4/inode.c
2.    int ext4_filemap_fault(struct vm_fault * vmf) {
3.        //根据可执行文件名获取 inode 节点信息
4.        struct inode * inode = file_inode(vmf->vma->vm_file);
5.        int err;
6.        down_read(&EXT4_I(inode)->i_mmap_sem);        //加锁
7.        err = filemap_fault(vmf)                      //读文件内容
8.        up_read(&EXT4_I(inode)->i_mmap_sem);          //解锁
9.        return err;
10.   }
```

图 5-33　ext4 文件系统的页错误处理

5. 堆空间的请求调页

基于页错误处理程序的实现,可以进一步实现堆空间的请求调入:在进程调用 malloc()等函数向操作系统申请内存时,操作系统可以不立即为进程分配物理内存,而是当进程访问该页而产生页错误时,再分配内存并设置 PTE 建立映射关系。与程序的代码和数据的请求调入不同,堆空间的请求调入并不涉及从外存中调入数据,操作系统只需要在页错误发生时为进程分配一个空闲页框。这个不涉及外存文件的映射关系称为匿名映射(被映射的虚拟地址对应的页称为匿名页),而需从外存文件加载内容的映射关系称为文件映射(对应的页称为文件页)。

显然,页错误处理程序对匿名页与文件页的缺页处理过程是不相同的。openEuler 通过以下方法来区分匿名页和文件页,并调用不同的处理函数。基于结构体 vm_area_struct,在映射虚拟地址段到匿名页时,与映射到文件页不同的是,openEuler 将会把结构体 vm_

area_struct 的成员 vm_file 设置为空。在进行页错误处理时，页错误处理程序只需要判断成员 vm_file 是否为空，即可区分不同的页类型，进而采取不同的处理逻辑。

至此，在硬件的辅助下，操作系统实现了程序的部分加载和按需加载，但仍然面临以下两个问题。

(1) 如果某个程序运行所需的内存大于物理内存大小，尽管操作系统可以先将该程序的部分加载到内存中执行，但随着进程的运行，外存中的程序不断被调入内存，并且如果进程的虚拟地址空间很大，进程可不断地向操作系统申请动态内存。这将导致物理内存不足，系统无法再继续运行。

(2) 在多个进程并发的需求下，各个进程部分加载到内存中执行，也可能占满物理内存。

5.5.2　交换空间

对于上述问题，一个可行的解决方案是：当物理内存不足而又需要加载新的进程代码或数据时，操作系统先将已装入内存的程序部分暂时换出到物理内存之外的某个存储空间，以空出页框用来保存待载入的程序部分；当进程再次访问到换出内存的程序部分时，操作系统再从这个外部存储空间将其调入内存。因为操作系统将页从内存中交换到外部存储空间，又将页从其中交换到内存，所以可以用作内存扩充的外部存储空间通常被称为交换空间(Swap Space)。操作系统通常在外存中划分一块区域作为一个交换空间。外存的特征是它的容量比内存大，但访问速度较慢。用作交换空间的典型外存设备就是磁盘，包括机械硬盘和固态硬盘。

操作系统要实现页的换入与换出，面临以下两个问题。

(1) 在内存满后，操作系统应该选择哪个或哪些页换出内存：如果选择经常被用到的页，在换出内存后马上又要被用到，这样不仅不能降低内存紧张的情形，反而会增加系统的负担。

(2) 在页换出到交换空间后，操作系统再次访问该页时，应如何在交换空间中定位和加载该页。

操作系统应该选择将哪个页换出内存(或称淘汰)的问题将在 5.5.4 节详细讲述。本节将重点描述操作系统如何组织并管理交换空间，进而实现页的换入换出对用户程序透明。为了方便管理交换空间和支持数据的换出换入，操作系统将交换空间划分为与内存页相同大小的单元——在交换空间中的单元称为块(block)。与每个页都有页号类似，每个块都有具体的块号。

1. 页换出

在引入交换空间并对其进行分块后，以一个简单的例子说明操作系统如何处理页框的换出过程。假设系统内两个进程 A 和 B 所需的内存都为 4 个页框，物理内存大小为 6 个页框。运行一定时间后，系统中的状态如图 5-34 左半部分所示：进程 A 的 4 个页已全部装入内存，进程 B 装入 1 个页。若此时进程 B 访问到未载入内存的页 1，系统将进入页错误处理：进行页表的设置，然后分配一个页框装载外存中的程序内容。但此时系统中不存在空闲内存，操作系统将选择一个页(假设进程 A 的页 3)换出到交换空间中，如图 5-34 中的第

(1)步。当页被换出内存时,操作系统需要记录给定页在交换空间中的地址(块号 0)。同时,操作系统还需要将存储虚拟地址到物理地址映射关系的 PTE 设置为无效,因为之后该 PTE 指向的页框将存储新的内容并映射到另一虚拟地址(可能属于另一个进程)。无效的 PTE 项能保证进程再次访问该映射关系时触发页错误异常,使得操作系统能接管控制权,将外存中的页再次调入内存中。最后,空出的内存页用来加载进程 B 的页 1,如图 5-34 中的第(2)步。

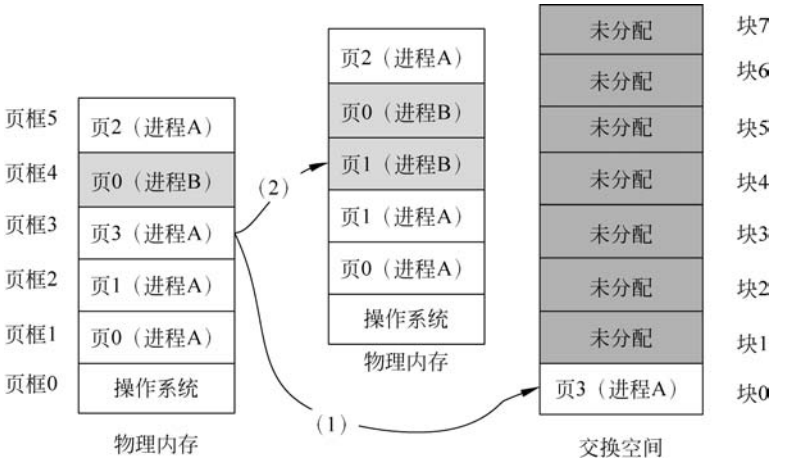

图 5-34　物理内存到交换空间的页换出过程

2. 页换入

在以上例子的基础上,下面进一步讨论页换入过程。若在进程 B 的页装入内存后,系统切换到进程 A 继续执行。进程 A 访问其页 3(此处为说明页换入过程而假设进程访问刚换出的页,这种情况可能存在,但应是操作系统采用的页置换策略需要尽量避免的情况),在地址转换过程中将发生页错误。系统再次进入页错误处理:选择一个页(假设进程 A 的页 0)换出,再将进程 A 的页 3 换入。系统进行换出与换入后,各进程的存储情况如图 5-35 所示。

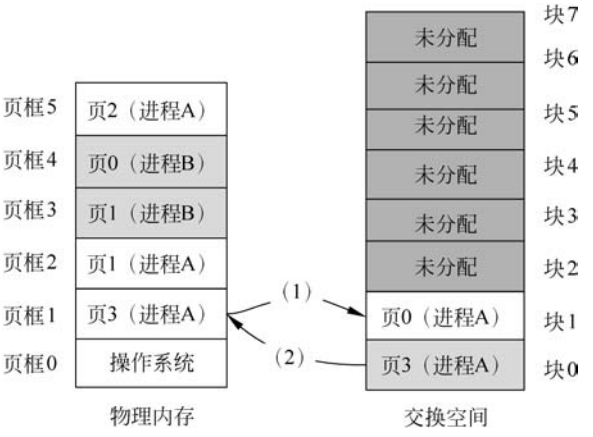

图 5-35　页的换出与换入

在对交换空间进行有效的组织并管理后，借助页错误机制，操作系统利用硬盘这类存储设备实现了逻辑上的物理内存扩充。

5.5.3　openEuler 中页交换的实现

1. 页换出

当物理内存不足时，openEuler 处理页换出的流程如图 5-36 所示。具体步骤如下：

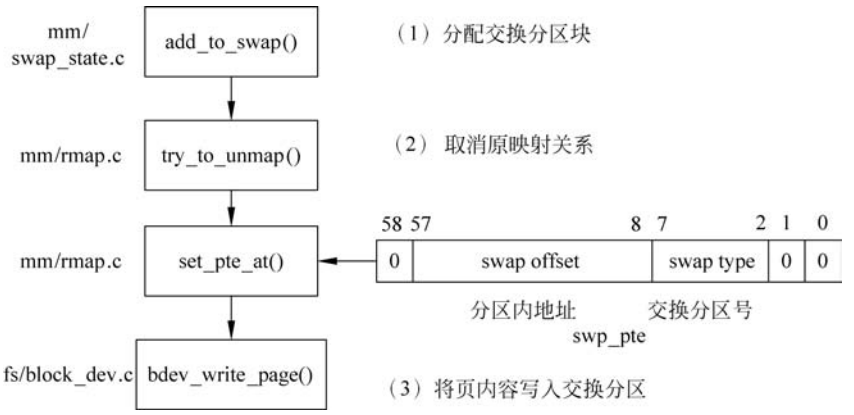

图 5-36　openEuler 的页换出处理流程

（1）分配交换空间块：页换出处理逻辑将会调用函数 add_to_swap()申请一个交换空间块。

（2）更新 PTE：调用函数 try_to_unmap()找出页表中所有引用了待换出页的 PTE 地址，然后构造一个 swp_pte 格式的 PTE：swp_pte 项如图 5-36 所示。因为 openEuler 支持多个交换空间，所以使用 bits[7:2]保存交换空间的编号，bits[57:8]保存被分配的交换空间块的偏移地址。最后调用函数 set_pte_at()向记录该映射关系的 PTE 写入构造好的 swp_pte 项。

（3）将数据写入交换区：调用函数 bdev_write_page()将页内容写入交换空间。

（4）释放内存页：将换出内容后的页框重新加入到空闲链表中。

以上描述的页换出(或称页回收)过程是：当系统中没有可供分配的空闲页框时，操作系统在内存分配函数中同步调用页回收过程。这个过程称为同步内存回收。除了同步内存回收过程之外，openEuler 还实现了异步内存回收过程：系统在运行时对内存进行周期性检查，当空闲页框的数量下降到 page_low(操作系统定义的一个标准)以下时，系统将唤醒 kswapd 进程来主动回收页。kswapd 进程根据特定的页置换策略选择相应的页换出，页换出的实现与同步回收过程是一致的。

2. 页换入

当进程试图引用一个已被换出的页时，系统将会进行页的换入。但是，请求调页与从交

换空间装入页都依赖于无效 PTE 导致的异常,操作系统如何区分这两种情况呢? 实际上,两种情况发生时存储映射关系的 PTE 项是有区别的:对于请求调页,当发生页错误时,存储映射关系的 PTE 项要么还并未分配内存(参考图 5-27),要么内容全为 0(未初始化);而对于交换空间中页的调入,PTE 记录并不全为 0:如上面所述页换出过程,openEuler 在换出页时会以一定的格式设置 PTE 以保存该页被换出的外存地址。因此,openEuler 在初始页错误时,将根据 PTE 记录的内容选择相应的操作。openEuler 对两种情况的处理如图 5-37 所示:第 5～14 行代码对 PTE 可能存在的两种情况(未分配内存或全为 0)进行判断,若符合以上两种情况则将 PTE 指针置为空;第 17～22 行代码根据 PTE 指针是否为空,分别选择不同的处理函数(匿名页错误、文件页错误以及从交换空间换入页)。

```
1.    //源文件: mm/memory.c
2.    static vm_fault_t handle_pte_fault(struct vm_fault * vmf) {
3.
4.        ...
5.        if (unlikely(pmd_none( * vmf -> pmd))) {
6.             vmf -> pte = NULL;                //若不存在 PMD,PTE 就不存在
7.        } else {
8.            //由 PMD 获得虚拟地址对应的 PTE 的地址
9.            vmf -> pte = pte_offset_map(vmf -> pmd, vmf -> address);
10.           vmf -> orig_pte = * vmf -> pte;    //读出 PTE
11.
12.           if (pte_none(vmf -> orig_pte)) {
13.               pte_unmap(vmf -> pte);
14.               vmf -> pte = NULL;             //PTE 全为 0,将 PTE 指向为 NULL
15.           }
16.       }
17.       if (!vmf -> pte) {
18.           if (vma_is_anonymous(vmf -> vma))
19.               return do_anonymous_page(vmf); //处理匿名页
20.           else
21.               return do_fault(vmf);          //处理文件页
22.       }
23.       if (!pte_present(vmf -> orig_pte))
24.           return do_swap_page(vmf);          //从交换分区换入页
25.       ...
26.   }
```

图 5-37　函数 handle_pte_fault()分支处理

openEuler 对页换入的处理过程如图 5-38 所示。交换处理函数 do_swap_page()将调用函数 swapin_readahead()为待加载的块分配一个页,再调用函数 bdev_read_page()将该块读入内存。之后,函数 do_swap_page()调用函数 set_pte_at()设置 PTE 以恢复映射关系,最后调用函数 swap_free()释放交换空间中的块。

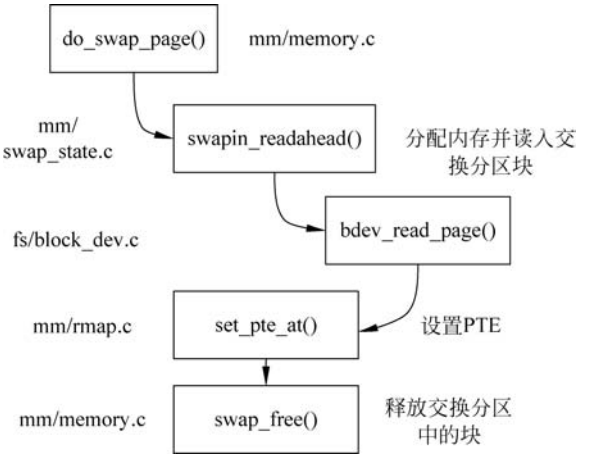

图 5-38　openEuler 页换入处理过程

5.5.4　页置换策略

操作系统需要依据一定的页置换策略决定将哪些页进行换出。在发生页置换时，操作系统将对页加以选择：如果淘汰经常使用到的页，根据局部性原理，该页可能很快又要被换入，这将严重地影响系统的性能。理想的结果是，页置换策略应该将那些在未来被访问概率最小的页移出内存。然而，操作系统无法准确地掌握一个页在未来被访问的概率。在操作系统中，常用的页置换策略主要有以下三种。

(1) 随机淘汰。该策略在所有页中，随机地选出一个页从内存换出，即进行淘汰。

(2) 先进先出(First In First Out，FIFO)。该策略根据页的分配时间，将页放入一个 FIFO 队列。采用该策略，一般认为相较于后换入内存的页，先换入的页再次被访问的可能性较小。因此，在发生页置换时，选择将最先换入内存的页进行淘汰。在实现时，设置一个指向 FIFO 队列头部的置换指针。在发生页置换时，将置换指针所指向的页进行换出，而将换入的页加入到 FIFO 队列的尾部。

由于没有区分页的重要性，FIFO 策略存在 Belady(陷阱)现象。一般地，如果给一个进程所分配的页框数越逼近它所需求的页框数，则发生缺页的次数将越少。在极限情况下，如果给一个进程分配了它所需求的全部页框，则不会发生缺页现象。然而，在使用 FIFO 策略时，当给一个进程未分配其所需求的全部页框时，有时可能出现分配的页框数增多但缺页次数反而增加的现象。这种现象称为 Belady 现象。Belady 现象产生的原因在于：FIFO 策略未考虑程序执行的局部性原理，使得换出的页可能是频繁访问的页。

(3) 最近最久未使用(Least Recently Used，LRU)策略。该策略的基本思想是：在发生页置换时，将最近一段时间内最久未使用过的页进行淘汰。基于程序的局部性原理，采用该策略，一般认为：如果某个页在最近被访问了，则它可能马上还将被访问；反过来，如果某个页在最近很长时间内未被访问，则它在近期也不会被访问。

为了找出最近最久未使用的页,操作系统需要对每个页设置一个访问标志。在每一次访问时,操作系统都必须更新这些访问标志。例如,系统可以通过一个特殊的链式结构来记录页的访问情况。每当某个页被访问时,该页的页号记录将被取出添加到链头。这样链头指针始终指向新访问的页记录,而链尾指针始终指向最近最久未被访问的页记录。假设某一进程大小为 5 个页,页的访问序列为 0、2、0、1、3、4、3、2、4,物理内存大小为 4 个页框,LRU 页置换过程如图 5-39 所示。在步骤(5)~(6)的过程,系统需要淘汰一个页以装入页 4。此时,根据 LRU 策略,系统将选择淘汰链尾的记录所指向的页(页 2),最后将页 4 的访问记录添加到链表中形成图 5-39 中(7)所示的结构。

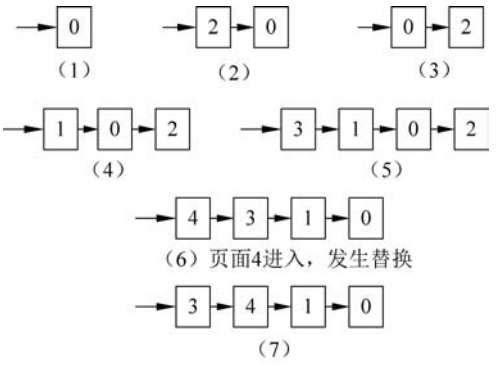

图 5-39　LRU 策略下页置换过程

openEuler 采用 LRU 策略实现页选择。如图 5-40 所示,openEuler 定义了 5 个链表,分别记录五种不同的页类型,以辅助页淘汰的选择:

(1) 非活跃(inactive)匿名页 LRU 链表保存所有最近没被访问的并且可以存放到交换空间的匿名页的记录;

(2) 活跃匿名页 LRU 链表保存所有最近被访问过的匿名页的记录;

(3) 非活跃文件页 LRU 链表保存最近没被访问过的文件页的记录;

(4) 活跃文件页 LRU 链表保存所有最近被访问过的文件页的记录;

(5) 不可回收 LRU 链表保存所有禁止换出的页的记录。

```
1.    //源文件: include/linux/mmzone.h
2.    enum lru_list {
3.        LRU_INACTIVE_ANON = 0,        //非活跃匿名页链表
4.        LRU_ACTIVE_ANON = 1,          //活跃匿名页
5.        LRU_INACTIVE_FILE = 2,        //非活跃文件页
6.        LRU_ACTIVE_FILE = 3,          //活跃文件页
7.        LRU_UNEVICTABLE,              //不可换出的页
8.        NR_LRU_LISTS                  //LRU 链表数为 5
9.    };
```

图 5-40　LRU 不同页类型的记录

处于不可回收 LRU 链表下的页不会被系统换出，在很多情况下，页有不被换出的需求。例如，一个对安全性要求较高的用户程序希望敏感数据不被换出到交换空间中，因为在程序结束后，攻击者可能访问交换空间以恢复这些数据。

openEuler 的内存回收是从非活跃链表末尾开始选择多个记录回收，在非活跃 LRU 链表中的页，更有可能优先被操作系统回收。在进程请求新页时，根据不同的页类型（匿名页或文件页），openEuler 将会把这个页的记录加入到不同的 LRU 链表中。活跃 LRU 链表用来保存最近被访问的页记录，那么进程申请的一个新页理应被加入到活跃 LRU 链表中。但是 openEuler 对不同类型的新页进行不同的处理，可能将某些页加入到非活跃链表中。openEuler 将堆和栈使用的新匿名页加入到活跃匿名页链表中；共享内存对应的页加入到非活跃匿名链表中；程序的代码段对应的文件加入到活跃文件页链表中；进程打开的文件对应的文件页加入到非活跃文件页链表中。以进程打开的文件所对应的文件页为例，采用这种方式的原因在于：若该页对应的内容未被修改，可直接尝试淘汰（释放），不必换出到交换空间。当进程再次访问该页使用时，操作系统可从外存的文件中再次加载。

1. 匿名页的回收

当发生内存回收时，openEuler 将访问 LRU 链表，根据各个页的访问情况在活跃链表与非活跃链表之间移动页记录，再选择非活跃链表末尾的多个记录所对应的页进行回收。以下将以匿名页 LRU 链表的调整和页回收的决策为例。当非活跃 LRU 链表中的页记录数量过少（操作系统定义了一个标准，例如若系统拥有 1GB 内存，非活跃链表应至少记录 250MB 内存页）时，openEuler 将从活跃 LRU 链表的尾部向前扫描，移动一些页记录（通常为 32 个）到非活跃 LRU 链表的头部。之后，操作系统更新非活跃匿名页 LRU 链表。操作系统从非活跃匿名页 LRU 链表尾部开始向前扫描 32 个页记录，对不同状态的页进行不同的处理：①对于最近访问过的页[进程每访问一个页，硬件会自动将该页的 PTE 访问标志（PTE 字段 AF）置为 1，以标记该页被访问]，操作系统将页记录移动到活跃匿名页 LRU 链表的尾部，并清零该页 PTE 的访问标志；②对于最近未访问的页，操作系统将尝试对它们进行回收。

2. 文件页的回收

文件页 LRU 链表的更新流程与匿名页 LRU 链表的更新流程是一致的。在处理过程上有一点不同的是：对文件页的访问记录，openEuler 引入了一个新的页标识 PG_referenced，当该页被访问，openEuler 就将其置为 1。这是因为，匿名页使用地址访问，CPU 会自动标记该页被访问。然而大部分文件页的访问是通过 write()和 read()函数调用访问，无法自动标记文件页的访问。因此操作系统需要在文件操作的代码路径上，显式地置位文件页的 PG_referenced，以标记该文件页被访问。例如，文件系统在管理缓冲区时，函数 touch_buffer()会调用函数 mark_page_accessed()设置 PG_referenced 标识位。

本章小结

直接内存访问的方式使得进程可读写其他进程的代码或数据,进而可破坏其他进程(甚至操作系统)的执行环境。为解决这个问题,现代操作系统引入了虚拟内存这一个间接层,使进程通过虚拟地址间接地访问物理内存。在虚拟地址到物理地址的转换过程中,系统就可以做“手脚”:检查进程访问的物理地址是否合法(例如,检查该物理地址是否在该进程被分配的物理地址空间中),从而确保各个进程之间不会互相读写,达到进程内存访问隔离的目的。

为了减少内存的外部碎片,以提高内存利用率,现代操作系统引入分页的内存管理方式。基于分页,操作系统在硬件 MMU 的辅助下实现虚拟地址到物理地址的转换以及实现内存访问的控制。然而,相对于直接内存访问方式,分页机制的间接内存访问为系统带来了额外的“负担”:①页表的内存存储开销过大;②地址转换访问内存查询页表过于耗时。为了尽可能减小这些“负担”,操作系统采用多级页表仅存储已建立映射关系的页表项,同时引入 TLB 缓存“常用”的映射关系来加速地址转换的过程。

内存管理的另一个重要任务是突破物理内存的限制。虚拟内存机制为每个用户程序都提供了接近于系统架构所能支持的最大物理空间容量的虚拟空间。这可能导致单个程序运行所需要空间大于实际的物理内存容量。操作系统基于请求调页和交换技术实现了物理内存的扩充,突破了物理内存大小的限制。基于请求调页机制,操作系统无须在进程加载时就装入全部内容,而在进程执行过程中根据进程所“需”调入相关代码或数据。交换技术把内存和外存结合起来管理,透明地在内存和外存之间移动进程的代码与数据,利用外存容量为用户进程提供一个比物理内存大得多的内存空间。