第 3 章

进程与线程

在早期的单道批处理系统中，计算机一次只能执行一个程序。该程序完全控制机器，并访问所有的系统资源。这种控制方式存在资源浪费、系统运行效率低等问题。为了提高资源利用率和系统的吞吐量，现代计算机系统采用多道程序技术，允许多个程序并发执行，共享系统资源。在多道程序环境下，由于 CPU 需要在各程序之间来回切换，程序的执行具有间断性。此外，由于并发执行的程序共享系统中的资源，任一程序对这些资源状态的改变都会影响其他程序的运行环境，即程序之间存在制约关系。然而，程序只是对计算任务和数据的静态描述，无法刻画并发执行过程带来的这些新特征。因此，计算机系统使用进程作为描述程序执行过程且能用来共享资源的基本单位。另外，由于进程的创建和切换开销较大，为了进一步提高执行效率，操作系统引入了“线程”的概念。本章先通过程序的并发执行过程引出进程这一抽象，并介绍系统对进程的描述和控制；随后介绍进程是如何通过系统调用在 CPU 上来回切换，从而实现并发执行的；最后对线程进行了详细阐述。

3.1 进程的概念

为了让程序源代码从人类易于理解的高级语言转换成计算机能够执行的机器语言，所有程序都将经过编译、链接、加载和执行 4 个阶段。一段时间内，机器通常并不只执行一个程序，而是并发地执行多个程序。为了对并发执行的程序加以描述和控制，操作系统引入了“进程”这一抽象。

3.1.1 程序：从源代码到执行

图 3-1 展示了一份 C 语言源代码（符合 C99 标准），它的功能是判断一个年份是否是闰年。下面以该程序为例，介绍一个程序从编写源代码到执行的过程。其中，链接用于将多个可重定位目标文件（由程序编译而成或是来自静态库）合并成一个可执行文件。由于链接过程与本章相关性不强，此处省略，感兴趣的读者可查阅编译原理相关书籍进行了解。

1. 编译阶段

编译的目的是将基于高级语言编写的源代码转换成计算机硬件能够执行的机器语言。假设图 3-1 的 C 程序保存在文件 example.c 中，那么可以使用图 3-2 中的交叉编译命令将 example.c 编译成 ARMv8 架构下可执行的二进制文件。

```
1.    #include <stdio.h>
2.    #include <stdlib.h>
3.    #include <stdbool.h>
4.    int global_var = 1;
5.    char * warning = "Wrong Input!\n";
6.    bool leap_year(int year) {
7.        bool result;
8.        if ((year % 4 == 0 && year % 100 != 0) || (year % 400 == 0))
9.            result = true;              //判断闰年
10.       else
11.           result = false;
12.       return result;
13.   }
14.   int main(void) {
15.       int y;
16.       scanf("%d", &y);
17.       if (y < 0) {                    //非法输入
18.           printf("%s\n", warning);    //打印出错信息
19.           return 0;
20.       }
21.       bool r;
22.       r = leap_year(y);
23.       printf("r = %d\n", r);
24.       int * dynmc = (int *)malloc(sizeof(int) * 2);
25.       dynmc[0] = y;
26.       dynmc[1] = leap_year(dynmc[0]);
27.       printf("dynmc[1] = %d\n", dynmc[1]);
28.       free(dynmc);
29.       return 0;
30.   }
```

图 3-1　C 程序示例

```
1.    aarch64-linux-gnu-gcc -o example example.c
```

图 3-2　编译命令示例

为了使操作系统能够以标准的方法对编译后形成的二进制文件进行处理,类 UNIX 操作系统通常采用 ELF 格式(Executable and Linkable Format,可执行可链接文件格式)作为二进制文件的标准格式。图 3-3 展示了 ELF 文件的执行视图(Execution View)。

图 3-3　ELF 文件的执行视图

ELF 头部包含了描述整个 ELF 文件的基本信息,段头(Program Header)表包含了描述各个段(Segment)的信息,其中,Segment 是存储程序中数据或代码的逻辑结构。在 ELF 文件中,比较重要的 Segment 有:.text 段保存机器指令序列;.data 段保存可读可写的全局变量和静态局部变量;

.rodata 段保存只读数据和常量；.bss 段保存未初始化的全局变量。示例程序编译后的二进制可执行文件反汇编后的部分内容如图 3-4 所示：.text 段包含了函数 leap_year()和 main()的汇编指令，.data 段中保存有全局变量 global_var 的值 01000000(整数 1 在小端格式下的存储形式即是 01000000)，.rodata 段中保存有字符串常量 warning 的值。

```
1.    Disassembly of section .text:
2.    ...
3.    0000000000000934 <leap_year>:
4.     934:    d10083ff     sub  sp, sp, #0x20
5.     938:    b9000fe0     str  w0, [sp, #12]
6.     93c:    b9400fe0     ldr  w0, [sp, #12]
7.    ...
8.    00000000000009d0 <main>:
9.     9d0:    a9bd7bfd     stp  x29, x30, [sp, #-48]!
10.    9d4:    910003fd     mov  x29, sp
11.    9d8:    90000080     adrp    x0, 10000 <__FRAME_END__+0xf470>
12.   ...
13.   Contents of section .rodata:
14.    0b58 01000200 00000000 57726f6e 6720496e  ........Wrong In
15.    0b68 70757421 0a000000 25640000 00000000  put!....%d......
16.    0b78 72203d20 25640a00 64796e6d 635b315d  r = %d..dynmc[1]
17.    0b88 203d2025 640a00                        = %d..
18.   ...
19.   Contents of section .data:
20.    11000 00000000 00000000 08100100 00000000  ................
21.    11010 01000000 00000000 600b0000 00000000  ................
```

图 3-4　二进制文件的部分反汇编内容

2. 加载阶段

由于内存是掉电易失的存储设备，因此，在被编译后，程序一般保存在持久化存储设备(例如磁盘)中。当用户想运行某个存储在磁盘中的程序时，操作系统将程序的 ELF 文件装入内存，这个过程称为程序的加载。程序的加载主要包括两个步骤：①解析 ELF 头部，进行程序加载的前期检查，例如检查 ELF 文件格式与当前 CPU 架构是否匹配；②读取段头表获取每个段的基本信息，为即将加载至内存的段分配内存空间，进而将这些段装入所分配内存空间。

3. 执行阶段

操作系统完成程序的加载后，利用 ELF 头部提供的信息，找到程序的入口地址。在 CPU 的程序计数器 PC 中，保存着下一条指令在内存中的地址。操作系统将.text 段中的程序入口地址赋值给 PC，随后 CPU 执行该程序的指令。此时，该程序获得了 CPU 的控制权。

在进一步介绍程序的执行前，回顾一下图 3-1 中的示例程序，并思考两个问题：①在图 3-1 的示例程序的第 24 行中，函数 malloc()动态申请的内存位于哪里？②在函数 leap_

year()内,局部变量 result 保存在哪里?这些数据所占空间是程序执行时才临时分配的,无须占用 ELF 文件空间,但需要占用内存空间。对于程序中动态申请的内存,操作系统专门开辟一段称为堆的内存空间,让编程人员自主地申请(函数 malloc())及释放(函数 free())。在内存空间中,堆通常由低地址向高地址生长。此外,result、y 和 r 等局部变量,保存在操作系统为每个程序专门开辟的一段称为栈的空间中。栈是先入后出的结构,在内存空间中,通常由高地址向低地址生长。程序被复制至内存中的布局如图 3-5 所示。

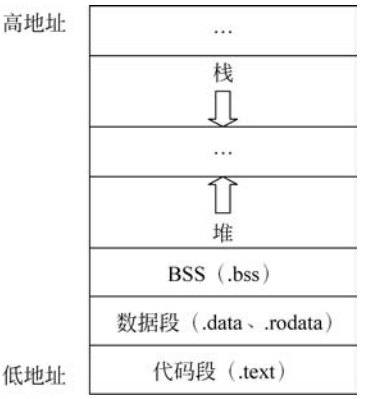

图 3-5　程序在内存中的布局

在函数运行时,程序计数器 PC 保存着即将执行的指令地址,链接寄存器 LR 保存着函数调用返回后下一条指令地址,堆栈指针寄存器 SP 保存着栈顶地址,而帧指针寄存器 FP 保存着栈底地址。栈上保存着为被调函数分配的局部变量以及由调用函数压入的函数参数,若被调函数再调用其他函数(或自身),还将由 CPU 的调用指令向栈内压入寄存器 LR 和 FP 中的数据。以上栈内数据共同构成被调函数的一个栈帧。当被调函数中再次发送函数调用,则继续在栈上为新被调用函数构建栈帧,以此类推。在被调函数执行结束后,按先入后出的顺序将寄存器数据、函数参数以及局部变量出栈,并将返回结果存入某个寄存器(例如 X0)或是存入某段内存中然后将该内存地址存入寄存器 X8 中返回给调用函数。这样为每次函数调用构建了独立的上下文(Context),使得多次调用互不影响。图 3-6 描述了图 3-1 实例中主函数 main()调用函数 leap_year()时的栈空间。函数的栈帧可视为函数切换的上下文,当调用函数跳转到被调函数执行时,在调用函数栈帧中,保存了调用函数的运行状态(局部变量、栈顶地址等)。当被调函数运行结束,栈帧的内容将帮助调用函数恢复之前的状态。

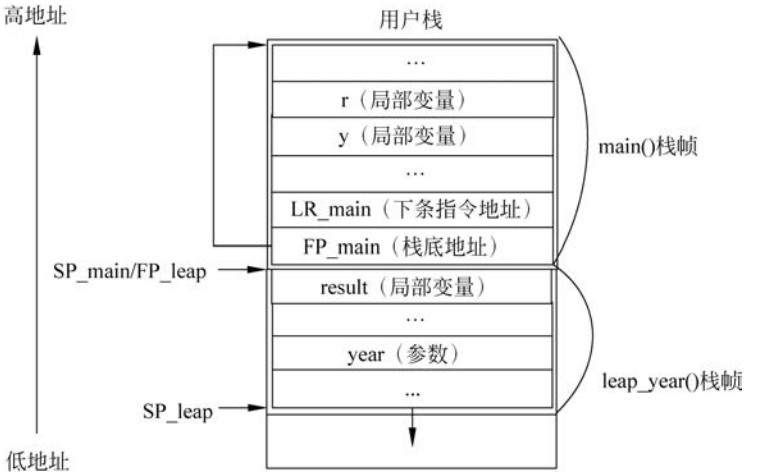

图 3-6　程序栈的内容

在程序执行过程中，可能会因为需要某些软硬件资源而处于等待的状态。例如，在示例程序中，当运行到函数 scanf()时，程序需要等待控制台的输入，在获得控制台的输入后它才会继续运行下去。为了避免 CPU 资源的浪费，在当前程序处于等待状态时，操作系统会剥夺当前程序的 CPU 使用权，调度其他程序来使用 CPU。

归纳起来，程序的执行过程依赖以下三种硬件状态：

(1) 寄存器。程序运行至少会用到寄存器 PC、LR、SP、FP，它们保存着函数计算的状态。

(2) 内存。程序的指令，以及在运行时读取和写入的数据都存储在内存中。

(3) I/O 信息。程序可能需要读写文件，涉及磁盘 I/O。另外，在 Linux 系统中，默认情况下，每个程序都有三个打开的文件描述符，即标准输入、标准输出、标准错误，分别用于接收用户输入、显示输出结果和错误信息。

3.1.2　程序的并发执行与进程抽象

一般情况下，一台机器上会同时运行多个程序。但是，在同一时刻，每个 CPU 只可以运行一个程序，而需要同时运行的程序数量可能远多于 CPU 的数量。解决此问题的关键在于：在 CPU 数量有限的情况下，如何为多个程序提供对 CPU 的复用，为用户制造多个程序同时执行的假象。因此，操作系统对 CPU 进行细粒度的时域共享以实现程序的并发执行，即允许 CPU 由一个程序占用一段时间，再由另一个程序占用一段时间，让多个程序交替地、断断续续地占用 CPU。只要 CPU 的计算速度和程序间的切换速度足够快，对于用户而言，这些交替执行的程序就是在同时执行。

程序的并发执行带来了一些影响：

(1) 程序执行的间断性。由于 CPU 需要在各程序之间来回切换，使得程序的执行具有间断性。虽然一个程序只能占用 CPU 一小段时间，但需要确保计算是逐渐趋向完成状态，而不是每次切换都从头开始执行。因此，让出 CPU 的程序需要保存当前的状态，而获得 CPU 的程序需要恢复上次保存的状态继续往下执行。例如，图 3-1 中的程序如果执行到第 12 行时，被操作系统切出(即让出 CPU 使用权)，操作系统应该为其保存好当前的程序状态，如局部变量(如 result 值)、PC 指针等。只有这样，程序下次才能恢复被剥夺 CPU 之前的状态，从而正确返回 result 值。因此，操作系统应该为程序提供状态的保存和恢复。

(2) 资源共享带来的制约性。由于并发执行的程序需要共享系统中的 CPU、内存等资源，任一程序对这些资源状态的改变都会影响其他程序的运行环境，造成程序之间存在制约关系。因此，为防止不同程序之间相互影响，操作系统应该确保程序之间所使用的资源是相互隔离的。

然而，程序只是对计算任务和数据的静态描述，无法刻画并发执行过程带来的这些新特征。因此，为了对并发执行的程序加以描述与控制，操作系统引入了“进程”概念。进程是操作系统为程序运行所提供的基本抽象概念，它是一个动态的概念，除了包括程序指令，还包括 CPU 寄存器状态，以及保存数据的堆栈段、数据段等内存空间。

3.2 进程的描述

为了反映并发执行的动态特征,操作系统引入 PCB(Process Control Block,进程控制块)这一结构。PCB 是操作系统感知进程存在的唯一实体。在创建一个进程时,操作系统首先为其创建 PCB,然后根据 PCB 中的信息对进程实施有效的管理和控制。在一个进程完成其任务之后,操作系统释放其 PCB,进程也随之终止。在不同的操作系统中,进程的 PCB 所包含的内容也会不同。本节将详细阐述 openEuler 中的 PCB,以及进程在其生命周期中的不同状态。

3.2.1 进程控制块

在 openEuler 中,PCB 的数据结构是 struct task_struct,它主要包含一个进程的描述信息、控制信息、CPU 上下文和资源管理信息四方面的内容。

1. 描述信息

PCB 中的进程描述信息主要包含进程标识符、用户标识号以及家族关系等,其相关成员变量如图 3-7 所示。

(1) 进程标识符。每个进程都有唯一的进程标识符,所以操作系统是依靠进程标识符来区分不同进程的。在 openEuler 中,进程标识符是一个 32 位正整型数,也就是说,操作系统中可以同时有 2^{31} 个进程标识符。openEuler 采用位图(bitmap)来记录进程标识符分配情况。

(2) 用户标识号。每个进程都隶属于某个用户,为了加以区分,操作系统引入用户标识符。在 openEuler 中,结构体 kuid_t 的成员 val 是一个无符号整数(即 U32),代表用户标识号。

(3) 家族关系。进程并不独立存在,通常与其他进程组成家族关系,便于操作系统进行组织和管理。除了 0 号进程外,其他所有进程都会有父进程,也可能有子进程;父进程的父进程以及再往上的父进程都是子进程的祖先进程。同一个父进程的多个子进程之间构成兄弟关系。在 openEuler 启动时,0 号进程进行一些内核初始化工作,还创建出 1 号进程。1 号进程完成用户空间初始化后,成为 init 进程,它是之后创建的所有进程的共同祖先进程。图 3-7 展示了 openEuler 中结构体 task_struct 描述这种家族关系的部分成员。

```
1.    //源文件: include/linux/sched.h
2.    struct task_struct __rcu  * real_parent;   //指向真正父进程
3.    struct task_struct __rcu * parent;         //指向跟踪当前进程的进程
4.    struct list_head   children;               //指向子进程
5.    struct list_head   sibling;                //指向兄弟进程
6.    struct list_head {
7.        struct list_head   * next, * prev;
8.    };
```

图 3-7 进程家族关系相关成员变量

在 openEuler 中，一个进程的父进程指针有两个，包括 real_parent 和 parent。real_parent 指向真正父进程，它是创建出当前进程的进程。而 parent 指向与信号响应相关的父进程，比如，进程的终止信号(SIGCHLD)会被发到父进程而不是真正父进程。当真正父进程正常存在时，real_parent 和 parent 指向同一个进程。如果一个进程的真正父进程先终止了，那么会有其他进程(例如 init 进程)成为当前进程的父进程，但它不是真正父进程。另外，openEuler 通过成员 sibling 来指向上一个和下一个兄弟进程。

2. 控制信息

PCB 中的进程控制信息主要包括进程的状态信息、进程优先级信息以及记账信息等，其相关成员如图 3-8 所示。

(1) 进程的状态信息。进程在活动期间处于就绪、运行、阻塞和终止等状态中的任意一种。有关进程的状态将在 3.2.2 节中进一步讨论。openEuler 中 PCB 结构体 task_struct 的 state 字段用于描述进程的状态，它是一个长整型数。

(2) 进程优先级信息。进程优先级用来确定进程被调度到 CPU 上执行的优先程度。openEuler 中的进程有多种优先级。图 3-8 展示了进程的静态、动态优先级，还有普通优先级与实时优先级的代码定义。静态优先级在进程启动时被给定，其值越小代表该进程优先级越高。在进程运行期间，静态优先级通常保持不变，可以由相关系统调用[如 nice()]修改。动态优先级与普通优先级默认等于静态优先级，但动态优先级会因调度策略的影响而被临时修改。为了满足实时需求，进程分为实时进程与普通进程。实时进程的优先程度仅与实时优先级相关，其值越大代表优先级越高。由于静态优先级对实时进程无效，普通优先级使得进程继承时可以不对普通进程与实时进程做额外区分操作。实时进程在调度时总是优先于普通进程。

(3) 记账信息。进程的记账信息主要给出进程占有和利用资源的有关情况，包括占用 CPU 的时钟周期数、时间总和等。进程的调度和控制以这些信息为依据来执行。例如，用来调度进程占用 CPU 的调度器根据 PCB 中记录的进程运行时间是否达到阈值来决定是否剥夺当前进程的 CPU 控制权。

```
1.    //源文件: include/linux/sched.h
2.    int   prio;                    //保存动态优先级
3.    int   static_prio;             //保存静态优先级
4.    int   normal_prio;             //取决于静态优先级和调度策略
5.    unsigned int    rt_priority;   //保存实时优先级
```

图 3-8　控制信息相关成员

openEuler 的 PCB 中的记账信息相关成员如图 3-9 所示，其中不仅记录了进程占用 CPU 的时长，还对进程切换时间进行了计数。

3. CPU 上下文

CPU 上下文是指进程执行到某时刻时 CPU 各寄存器中的值，这些值代表着当前进程活动的状态信息。在支持进程并发的场合，一个进程的执行是间断性获取 CPU 控制权的过

```
1.    //源文件: include/linux/sched.h
2.    u64   utime                    //进程在用户态下占用的 CPU 时钟周期数
3.    u64   stime;                   //进程在内核态下占用的 CPU 时钟周期数
4.    u64   utimescaled              //记录进程在用户态下的运行时间
5.    u64   stimescaled;             //记录进程在内核态下的运行时间
6.    u64   gtime;                   //虚拟机运行的 CPU 时钟周期数
7.    u64   start_time               //进程创建时间
8.    u64   real_start_time;         //进程创建时间,还包括进程睡眠时间
9.    unsigned long nvcsw, nivcsw;   //上下文切换计数
```

图 3-9　记账信息相关成员

程。当进程被剥夺(或主动放弃)CPU 控制权,为了让该进程之后能恢复被打断前的状态,操作系统需要保存进程在切换时的 CPU 上下文。

openEuler 在进行进程切换时,CPU 上下文保存在 task_struct－> thread_struct－> cpu_context 中。其中,结构体 thread_struct 用于记录与 CPU 相关的所有状态信息,包括 CPU 上下文、错误信息等,它的定义与体系结构是强相关的。结构体 thread_struct 与 cpu_context 的定义如图 3-10 所示,其成员包括通用寄存器 X19～X28、栈帧寄存器 FP、堆栈指针寄存器 SP 和程序计数器 PC。

```
1.    //源文件: arch/arm64/include/asm/processor.h
2.    struct task_struct {
3.        struct cpu_context   cpu_context;   //CPU 上下文
4.        ...
5.        unsigned long   fault_address;      //错误信息
6.        unsigned long   fault_code;         //寄存器 ESR_EL1 的值,表示发生错误原因
7.        ...
8.    }
9.    //源文件: arch/arm64/include/asm/processor.h
10.   struct cpu_context {
11.       unsigned long x19;
12.       unsigned long x20;
13.       unsigned long x21;
14.       unsigned long x22;
15.       unsigned long x23;
16.       unsigned long x24;
17.       unsigned long x25;
18.       unsigned long x26;
19.       unsigned long x27;
20.       unsigned long x28;
21.       unsigned long fp;
22.       unsigned long sp;
23.       unsigned long pc;
24.   };
```

图 3-10　CPU 上下文相关数据结构

4. 资源管理信息

PCB 中包含最多的是资源管理信息，其中包括关于存储器、文件系统和使用输入/输出设备的信息等。

在 openEuler 中，PCB 中的资源管理信息主要是与内存和文件相关的，如图 3-11 所示。成员 stack 指向进程的内核栈。内核栈是内核为每个进程开辟的一块内核空间，是进程陷入内核态后使用的栈空间；内存描述符对应结构体 mm_struct 记录了进程的内存布局；结构体 fs_struct 则记录与进程相关联的文件系统信息，包括当前目录和根目录；结构体 files_struct 是进程正打开的所有文件的列表。这里值得注意的是，输入/输出设备在 openEuler 中也以文件的形式存在。

```
1.    //源文件: include/linux/sched.h
2.    void    * stack;                        //指向进程的内核栈
3.    struct mm_struct * mm, * active_mm;     //进程的用户空间描述符
4.    struct fs_struct * fs;                  //进程相关联的文件系统信息
5.    struct files_struct * files;            //指向打开的文件列表
6.    ...
7.    //源文件: include/linux/mm_types.h
8.    struct mm_struct {                      //内存描述符
9.        spinlock_t arg_lock;                //自旋锁,保护下面这些字段
10.       //内存空间中各段起始/结束地址
11.       //包括栈、映射段、堆、BSS 段、数据段、代码段
12.       unsigned long start_code, end_code, start_data, end_data;
13.       unsigned long start_brk, brk, start_stack;
14.       unsigned long arg_start, arg_end, env_start, env_end;
15.       ...
16.   };
17.   //源文件: include/linux/fs_struct.h
18.   struct fs_struct {                      //文件系统描述符
19.       int users;                          //该结构的引用用户数
20.       spinlock_t lock;                    //自旋锁
21.       struct path root, pwd;              //根目录与当前目录
22.       ...
23.   };
24.   //源文件: include/linux/fdtable.h
25.   struct files_struct {
26.       atomic_t count;                     //引用计数
27.       struct fdtable __rcu * fdt;         //默认指向 fdtab,可用于动态申请内存
28.       struct fdtable fdtab;               //为 fdt 提供初始值
29.       ...
30.   }
```

图 3-11　PCB 中的资源管理相关成员

3.2.2　进程状态

进程在其整个生命周期中可以呈现不同的状态，随着进程的执行和外部条件的变化，进

程可以在不同的状态之间转换。操作系统通过 PCB 中的状态值来描述进程的当前状态。结合图 3-12 中进程状态转换示意图,进程的各个状态及转换关系详述如下。

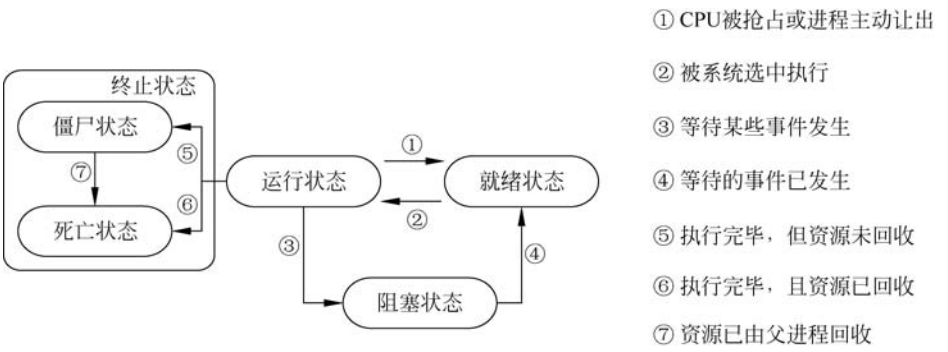

图 3-12　进程状态转换

1. 就绪状态

进程处于就绪状态时,进程位于运行队列中,表明其已经获得除 CPU 之外的其他资源。当进程被操作系统选中去占用 CPU 时,处于就绪状态的进程将转换为运行状态,如图 3-12 中的转换②。

2. 运行状态

处于运行状态时,进程中的指令正在被 CPU 执行。只有处于就绪状态的进程才可以转换为运行状态。进程在遇到以下几种情况时会退出运行状态:

(1) 当 CPU 被其他进程抢占或者进程主动让出 CPU 时发生转换①,进入就绪状态;

(2) 当进程需要等待资源或者等待一些事件发生而不得不退出执行时发生转换③,进入阻塞状态;

(3) 进程执行完毕后,通常发生转换⑤或转换⑥,进入终止状态。

3. 阻塞状态

处于阻塞状态时,进程通常是在等待着某些外部事件发生。在这些事件到来后,进程具备了继续执行的条件,但还无法直接获得 CPU 控制权,因此发生转换④进入就绪状态。处于阻塞状态的进程也可以通过其他方式唤醒。根据唤醒的困难程度,唤醒方式可以分为三种:

(1) 轻度阻塞状态,可以被一些系统调用(System Call)显式地唤醒或者由一些急需处理的信号唤醒;

(2) 中度阻塞状态,可以被显式地唤醒,或是被一些致命信号(可能导致进程终止)唤醒;

(3) 深度阻塞状态,只能被显式地唤醒,不可因信号退出。

4. 终止状态

进程的终止状态又包括僵尸状态和死亡状态。进程处于僵尸状态代表着此时该进程的

父进程并未回收此进程，也未回收此进程所占用的资源(包括 PCB)。如果父进程早于子进程退出，操作系统会让 init 进程成为子进程的父进程，所以在子进程退出后将由 init 进程回收子进程占用的相关资源。当父进程回收了此进程的资源，将发生转换⑦，进程进入死亡状态，生命周期完全结束。

在 openEuler 中，结构体 task_struct 中的状态域相关成员 state 可参见图 3-13 示例程序的第 6 行。进程的大部分状态通过成员 state 描述，但是僵尸状态与死亡状态不仅需要把成员 state 设为 TASK_DEAD，还需把成员 exit_state 分别设为 EXIT_ZOMBIE 与 EXIT_DEAD。

```
1.    //源文件: include/linux/sched.h
2.    struct thread_struct  thread;            //进程切换时 CPU 状态(存放各寄存器值)
3.    pid_t  pid;                              //进程标识符
4.    struct task_struct __rcu  * real_parent; //指向真正的父进程
5.    struct task_struct __rcu  * parent;      //指向父进程
6.    volatile long  state;                   //进程状态. -1: 不可运行; 0: 可运行; > 0: 停止
7.    u64    utimescaled, timescaled;          //进程在用户态/内核态下运行了多长时间
8.    struct mm_struct     * mm;               //内存描述符,属于进程的资源
9.    void   * stack;                          //指向进程的内核栈
10.   struct fs_struct     * fs;               //进程的资源(关于文件系统)
11.   struct files_struct  * files;            //进程的资源(关于已打开文件)
12.   ...
```

图 3-13　描述进程状态的相关成员

除了上面描述的几种状态，openEuler 中还定义了停止状态和跟踪状态，成员 state 分别为 TASK_STOPPED 和 TASK_TRACED。当进程收到停止信号(包括 SIGSTOP、SIGTTIN、SIGTSTP 及 SIGTTOU)时会停止执行；而当进程被 debugger 进程或其他进程监视跟踪时，被监视进程处于跟踪状态。

3.3　进程的控制

为了实现进程的创建、销毁等操作以及完成进程各状态间的转换，操作系统将完成这些特定功能的程序段设计成原语。这些原语被称为进程控制原语。本节先介绍进程控制原语的概念，然后依次介绍在 openEuler 中进程创建、程序装载及进程终止三种原语，最后介绍 openEuler 中进程树的创建过程。

3.3.1　进程控制原语

当用户双击应用程序图标或在 shell 中输入命令时，操作系统需要创建新的进程，以运

行该应用;当进程完成其计算任务时,操作系统需要销毁该进程,以释放它占用的物理资源;在进程向磁盘发起 I/O 请求后,在等待 I/O 完成的过程中,操作系统需要转换进程状态,剥夺此进程的 CPU 控制权,以提高 CPU 的利用率。在上述场景中,操作系统使用一些具有特定功能的程序段来创建、销毁进程以及完成进程各状态间的转换,这个过程称为进程控制。

在操作系统中,通常把用于进程控制的程序段设计成原语。原语指完成某种特定功能的一段机器指令集合。原语的一个重要特征是执行期间不可被打断,这可以通过关中断实现。原语常驻在内存中,通常需要在内核态下执行,但操作系统也为用户程序提供了调用的接口。当这些接口被调用时,操作系统由用户态切换为内核态,并执行相应的原语。

操作系统通常不允许原语并发地执行,这是因为:若是这些程序段可以并发执行,容易引发控制错误。假如允许同时对一个进程使用两次销毁原语,将有一个触发错误。若这些原语不是原子的,在执行过程中就可能被打断,那么,进程的控制信息(如进程状态)在被打断期间可能被其他原语修改,将导致执行结果不可靠,达不到进程控制的目的。进程控制的原语主要包括创建、销毁、阻塞与唤醒。

1. 创建

新创建的进程通常称为子进程,创建出子进程的进程是其真正父进程。为了简化称呼,除了 3.3.4 节中的父进程指结构体 task_struct 成员 parent 指向的进程外,后文其他位置的父进程都指结构体 task_struct 成员 real_parent 指向的进程。进程一般通过以下四种方式创建。

(1) 为批处理作业创建。操作系统从外存中的批处理作业控制流中获取新任务,并为其创建进程。

(2) 用户登录系统。终端用户登录时,登录界面就需要运行在一个新进程中。

(3) 系统进程为提供服务而创建。例如,当用户请求一个功能时,比如视频聊天,操作系统会为其创建控制摄像头、喇叭、麦克风等的进程,不需用户自己去创建。

(4) 由已存在的进程创建。例如,在多核 CPU 中,父进程创建子进程以实现并行工作。

前三种情况都是由操作系统来创建进程,最后一种是由父进程来创建新进程。

无论是由操作系统还是由父进程来创建,新进程都必须通过调用进程创建原语来生成。创建原语首先构建代表进程的数据结构,然后将其代码加载到内存空间。具体来说,创建原语将申请一个空白 PCB(进程控制块),填入调用者提供的有关参数,并完成 PCB 的初始化。这些参数包括进程名、进程优先级、进程正文段起始地址、资源清单等。其实现过程如图 3-14 所示。之后需要为进程分配资源,例如内存、文件等。最后将进程加入就绪队列中,等待被调度执行。

2. 销毁

在操作系统中,进程的销毁操作可能由以下三种情况触发:①该进程已完成所要求的功能而正常终止;②由于某种错误导致非正常终止;③父进程或是更高级别的祖先进程要求销毁某个子进程。

无论哪一种情况导致进程被销毁，进程都必须释放它所占用的各种资源和 PCB 结构本身，以利于资源的有效利用。当然，一个进程所占有的某些资源在使用结束时可能早已释放。另外，在一个父进程或祖先进程销毁某个子孙进程后，如果该子孙进程还有自己的子进程，这些子进程将被其他进程或 init 进程收养。因为，进程占用的一部分内核资源需要由其父进程来回收。正常情况下，子进程应该先于父进程终止，否则子进程可能一直占用资源不放。

销毁一个进程需要调用销毁原语实现。销毁原语首先检查 PCB 进程链表或进程家族，寻找所要销毁的进程是否存在。如果找到了所要销毁进程的 PCB，销毁原语将释放该进程所占有的资源，并把对应的 PCB 结构从进程链表或进程家族中摘下并返回给 PCB 空队列。销毁原语的实现流程图如图 3-15 所示。

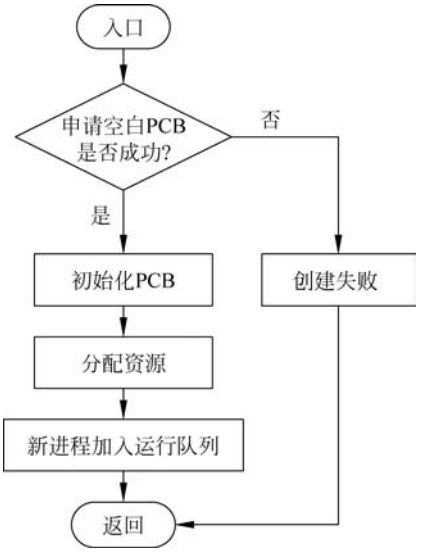

图 3-14　创建原语流程图

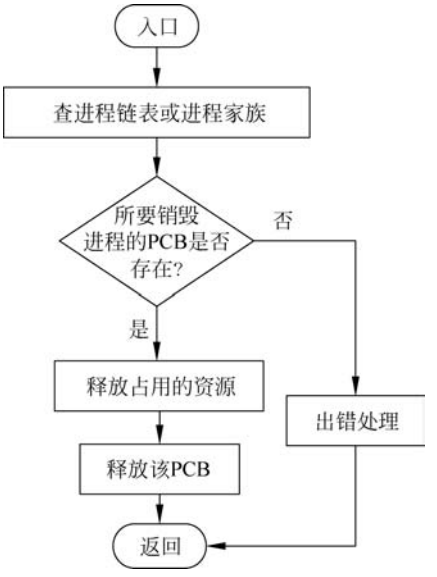

图 3-15　销毁原语流程图

进程的创建原语和销毁原语完成了进程从无到有、从存在到终止的变化。被创建后的进程最初处于就绪状态，然后经调度程序选中后进入运行状态。有关进程调度部分将放在第 4 章中详述，这里主要介绍实现进程由运行状态到阻塞状态，又由阻塞状态到就绪状态转换的两种原语，即阻塞原语与唤醒原语。

3. 阻塞与唤醒

在一个进程等待某一事件（例如键盘输入数据、磁盘 I/O 和其他进程发来的数据等）发生但该事件尚未发生时，该进程调用阻塞原语来阻塞自己。阻塞原语在阻塞一个进程时，由于该进程正处于运行状态，故应先保存该进程的 CPU 现场，然后将被阻塞进程置为阻塞状态后插入等待队列中，再转到进程调度程序选择新的就绪进程投入运行。阻塞原语的实现流程图如图 3-16 所示。在阻塞原语中，转到进程调度程序是非常关键的步骤，否则，CPU

将会出现空转而造成资源浪费。

当等待队列中的进程所等待的事件发生时,等待该事件的所有进程都将被唤醒。但是,进入阻塞状态的进程已经不具备 CPU 控制权,无法执行任何指令,因此也无法自我唤醒。唤醒一个等待进程有两种方法:一种是由操作系统或系统进程唤醒;另一种是由事件相关的其他进程唤醒。若是由操作系统负责唤醒,系统进程统一进行管理,并将“事件发生”这一消息通知等待进程,从而使得等待进程被唤醒后进入运行队列。等待进程也可由事件处理进程唤醒,此时,事件发生进程和被唤醒进程之间是相互合作的关系。因此,唤醒原语既可被系统进程调用,也可被事件发生进程调用。调用唤醒原语的进程称为唤醒进程。唤醒原语首先将被唤醒进程从相应的等待队列中移除,再将被唤醒进程置为就绪状态并送入运行队列。在把被唤醒进程插入运行队列之后,唤醒原语既可以返回原调用程序,也可以转到进程调度程序,以便让调度程序有机会选择一个合适的进程执行。唤醒原语的实现流程图如图 3-17 所示。

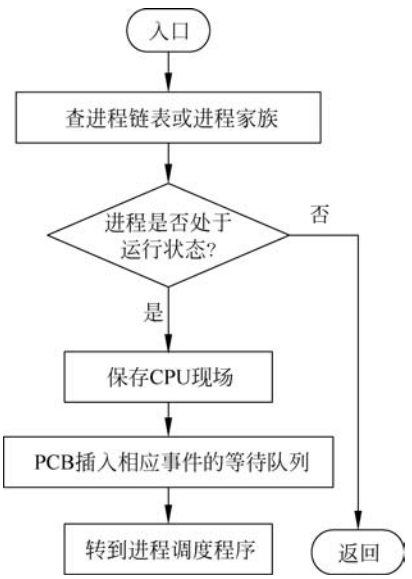

图 3-16　阻塞原语流程图

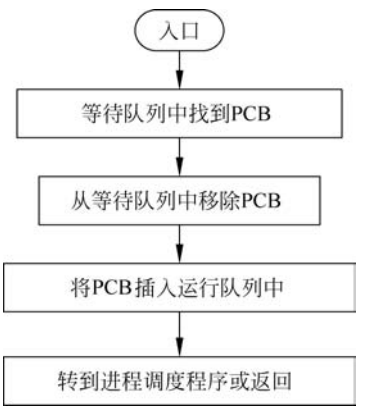

图 3-17　唤醒原语流程图

3.3.2　进程创建

如果每个进程都从零开始创建,那必然有大量初始化工作(如初始化 PCB、构建虚拟内存空间等)是重复的。类 UNIX 操作系统提供创建原语函数——fork()/clone():使用已有的进程复制出新进程,相当于新进程完成了与已有进程同样的初始化工作。就像细胞分裂一样,先将一个进程的核心内容复制一份,然后分裂成两个完整的进程。相较于重新创建,对已有进程进行复制可直接越过大量初始化工作。其中,clone()函数主要用于线程创建,将在 3.6 节介绍。fork()函数创建的进程是已有进程的一个副本,执行的程序也与已有进

程相同。如果要让新进程运行新的程序，需要联合 exec()函数簇来实现。本节仅介绍新进程的创建，而程序的加载将在 3.3.3 节介绍。

那么，fork()函数是如何创建新进程的呢？首先，每一个新创建的进程(除了 0 号、1 号进程外)都有其父进程，因此可以通过其父进程来完成新进程的创建工作。下面主要从进程运行的三个必备要素考虑：①操作系统需要通过 PCB 对进程进行管理，所以 fork()函数先为新进程创建 PCB，并进行初始化；②进程运行时的相关状态及数据保存在 CPU 上下文中。fork()函数通过复制父进程的 CPU 上下文到新进程的 PCB 中，使得新进程拥有与其父进程相同的执行环境；③由于进程实体是存储在内存中的，进程必须拥有物理内存空间才能执行，所以 fork()函数也需要为新进程分配物理内存。

1. PCB 的复制

在 fork()函数中，新进程 PCB 的初始化通过复制其父进程的 PCB 来实现，由此继承父进程的进程状态等信息。PCB 的创建与初始化的关键代码如图 3-18 所示。在 openEuler 中，函数 dup_task_struct()通过调用函数 alloc_task_struct_node()为新进程申请管理 PCB 的页面(第 3～7 行[①])，实现对新进程的创建，并使用函数 arch_dup_task_struct()将父进程的 PCB 赋值给新进程(第 13 行)。

```
1.    //源文件: include/asm/thread_info.h
2.    //分配 PCB
3.    #define alloc_task_struct_node(node) ({
4.        struct page *page = alloc_pages_node(node,
5.                        GFP_KERNEL | __GFP_COMP, KERNEL_STACK_SIZE_ORDER);
6.        struct task_struct *ret = page ? page_address(page) : NULL; ret;
7.    })
8.    //源文件: kernel/process.c
9.    //复制 PCB
10.   int arch_dup_task_struct(struct task_struct *dst, struct task_struct
11.                                                                    *src) {
12.       ...
13.       *dst = *src;     //将父进程的结构体 task_struct 各个成员复制给新进程
14.       dst->thread.sve_state = NULL;
15.       clear_tsk_thread_flag(dst, TIF_SVE);
16.       return 0;
17.   }
```

图 3-18　PCB 的创建与初始化

2. CPU 上下文的复制

经过函数 dup_task_struct()之后，新进程拥有了自己的 PCB，但此时新进程只是一个

① 全书关于代码引用中使用的“第几行”如没有指定引用位置的则默认为正在描述的代码段。例如本处“第 3～7 行”则指图 3-18 代码段第 3～7 行。

未初始化的进程,还不可运行。拥有 PCB 的新进程还需要有执行环境才能执行。由于父进程的一些运行状态被存储在 CPU 的寄存器中,新进程可以通过复制父进程 CPU 上下文的状态,来拥有与父进程相同的执行环境。如图 3-19 所示,在 openEuler 中,函数 fork()先从新进程的内核栈的栈底获取 pt_regs 结构(第 2 行),并且将新进程的结构体 thread_struct(即 p—> thread)清空(第 4 行)。其中,结构体 pt_regs 存储的是用户空间进程进入内核模式时,需要保存的用户进程寄存器状态。结构体 thread_struct 中存储的是内核态执行进程切换时当前进程的 CPU 上下文。然后,函数 fork()通过函数 current_pt_regs()获取当前进程的 pt_regs 结构,并直接将当前寄存器的值赋值给新进程(第 6 行)。这里值得注意的是,新进程复制父进程的寄存器的值后,还需要为成员 reg[0]赋 0(第 14 行,ARM 架构下 reg[0]代表寄存器 X0)。由于 ARM 架构中通常使用 X0 作为返回值寄存器,因此在新进程中返回后,函数 fork()的返回值为 0。函数 fork()将新进程的 PC 指针指向函数 ret_from_fork(),使新进程从函数 ret_from_fork()开始运行(第 20 行),最后将内核栈指针指向 childregs(第 21 行)。至此,新进程的执行环境设置完毕。

```
1.    //源文件: kernel/process.c
2.    struct pt_regs *childregs = task_pt_regs(p);    //获取 pt_regs 结构
3.    //将新进程的内核态需要的寄存器信息清 0
4.    memset(&p->thread.cpu_context, 0, sizeof(struct cpu_context));
5.    ...
6.     *childregs = *current_pt_regs();              //将当前寄存器值复制给新进程
7.    //reg[0]为 X0 寄存器,新进程 X0 置 0,因此 fork 在新进程中返回 0
8.    childregs->regs[0] = 0;
9.    //将新进程的内核态需要的寄存器信息清 0
10.   memset(&p->thread.cpu_context, 0, sizeof(struct cpu_context));
11.   ...
12.   if (stack_start) {
13.      //如果用户设置了栈的起始地址,设置用户栈的地址是 stack_start
14.      if (is_compat_thread(task_thread_info(p)))
15.          childregs->compat_sp = stack_start;
16.      else
17.          childregs->sp = stack_start;
18.   }
19.   ...
20.   p->thread.cpu_context.pc = (unsigned long)ret_from_fork;
21.   p->thread.cpu_context.sp = (unsigned long)childregs;
```

图 3-19　进程执行环境的设置

3. 地址空间的复制

函数 fork()创建的新进程与父进程有着完全一样的地址空间,这可以用两种方案来实现。第一种方案是给新进程分配与父进程等量的物理内存,并把父进程在内存中的所有数据都给新进程复制一份。这个过程需要完成的事情有:

(1) 分配物理页作为新进程的页表；

(2) 复制父进程页表内容；

(3) 对照父进程页表，为新进程页表项分配物理页并建立映射关系；

(4) 将父进程的物理页内容复制到新进程相应页中。

由于第(3)、(4)步中分配物理页并复制旧页内容都是非常耗时的操作，父进程拥有的物理页越多，这个过程的时间开销就越大。然而，新进程创建成功后通常立即装载新的程序到地址空间，之前复制的内容只有很少数被使用，而且其内容也会被覆盖。例如，shell 创建的新进程需要立马执行用户命令指定的程序。那么，之前复制父进程整个地址空间的操作是不必要的。

因此，openEuler 选择了第二种实现方案：在新进程中建立与父进程同样的映射关系，让两个进程以只读的形式共享同一片物理内存，直到某个进程试图修改某一页内容时，再为其分配新的物理页并复制原页面内容，这被称作写时复制(copy-on-write)，写时复制过程如图 3-20 所示。两种方案关键的不同点在于，前者复制了父进程拥有的所有物理页的内容，而后者仅复制了映射关系。

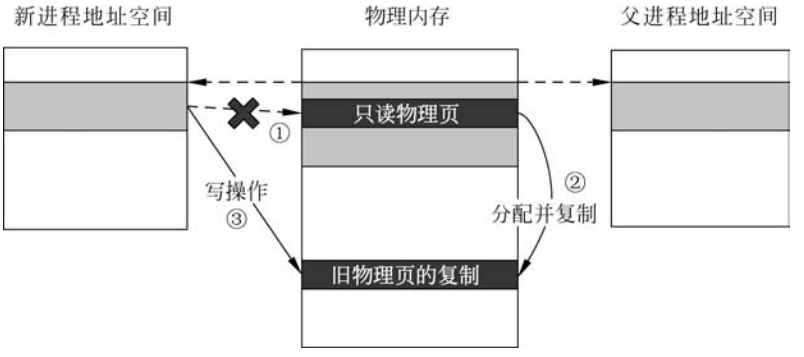

图 3-20　写时复制过程展示

实现写时复制首先需要思考如何完成映射关系的复制。进程地址空间与物理内存的映射关系保存在页表中，复制父进程的页表就意味着让新进程与父进程地址空间指向相同的物理内存。但是，为了避免某一方修改这部分共享内存对另一方造成影响，在页表复制过程中，还需要将用户空间中支持写时复制的物理页都标记为只读(Read-Only)，任意一方想写入内容都会触发缺页异常，请求内核来处理这种情况。

内核收到缺页异常后，如何处理才能做到允许当前进程继续执行写操作而不影响其他进程？首先内核需要确认该缺页异常是由于写时复制引起的，之后内核会将触发缺页异常的只读物理页内容复制到一个新的可读写物理页中，并在当前进程页表中的映射关系中用新页替换旧页。这样就在进程不知情的情况下，将写操作的对象换成了新物理页。由于新物理页并未映射到其他地址空间，所以内容的变化也不会影响其他进程。以下是 openEuler 解决这两个部分的关键技术。

1) 复制映射关系

进程地址空间分为内核空间与用户空间,映射关系的复制也分为两部分完成。openEuler 在全局层面维护了一份主内核页表,由于所有进程共享内核,所有进程的内核空间部分都是对主内核页表的一个复制或引用。因此,所有进程的内核空间都是相同的。不同进程的用户空间可以不同,所以函数 fork()需要为新进程的每一级页表分配物理页,并从父进程对应页表中复制所有页表项。由于 ARMv8 架构的 CPU 最大支持 48 根地址线,即只能寻址 2^{48} 的地址空间,最多支持 4 级页表,在编译 openEuler 时,编程人员可以通过配置宏 CONFIG_ARM64_VA_BITS 来选择编译出支持 4 级或者 3 级页表的系统,如第 5 章内存管理的三级页表示例。此处以 openEuler 的 4 级页表为例进行介绍。4 级页表包括 PGD(页全局目录)、PUD(页上级目录)、PMD(页中间目录)和 PTE(页表项)页表。每一级页表中都包含若干保存下一级页表基地址的页表项,例如,PGD 页表中每个不为空的 PGD 页表项都指向一个 PUD 页表,最终,PTE 页表项指向的是一个实际物理页。函数 fork()需要遍历每一级页表,先分配物理页用作新进程页表,然后将父进程所有不为空的页表项内容赋给新进程,其中,页表项内容包括下级页表基址、标志位以及权限位等。每一级页表项的复制都对应一个特定的复制函数,如图 3-21 所示。下面以 PTE 页表的复制为例展开详细的介绍,对应的是函数 copy_pte_range()。

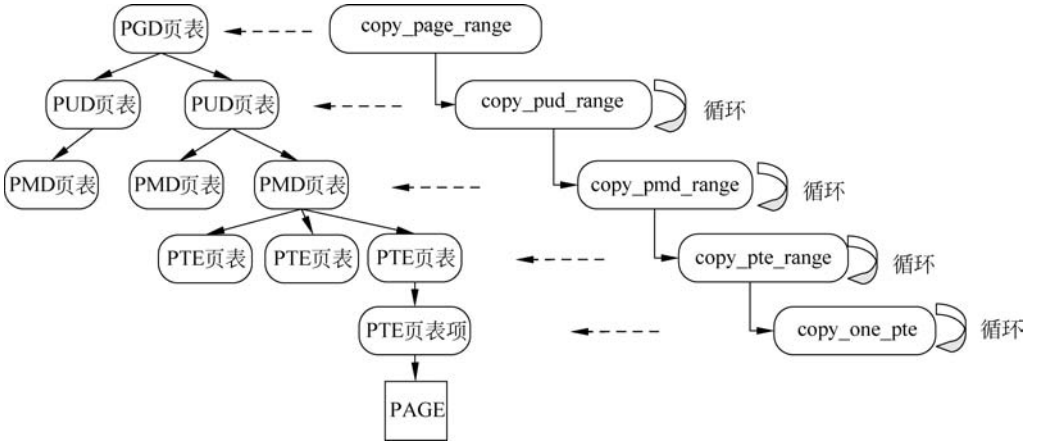

图 3-21 遍历各级页表完成页表项复制

函数 copy_pte_range()先为新进程分配一个物理页作为 PTE 页表,接着遍历父进程的 PTE 页表,逐个地去完成所有 PTE 页表项的复制。如图 3-22 所示,PTE 页表项实际上保存的是一个 64 位整数,在每次循环中,函数 copy_pte_range()会将父进程当前 PTE 内的整数直接赋值给新进程(第 9 行),然后选中下一个 PTE 页表项(第 13 行)。待到当前页的所有 PTE 页表项复制完毕,就完成了一个 PTE 页表的复制。其他级的页表都是循环地调用下一级的复制函数,PMD 页表中所有页表项指向的 PTE 页表都复制完成后,代表该 PMD 页表复制完成,这样一级一级地完成页表复制,进而完成整个页表的复制。

```
1.    //源文件：mm/memory.c
2.    //分配一个物理页作为新进程的 PTE 页表,返回起始地址
3.    dst_pte = alloc_pages(PGALLOC_GFP, 0);
4.    //从父进程 PMD 页表项中获取 PTE 页表起始地址
5.    src_pte = pte_offset_map(src_pmd, addr);
6.    do {
7.        ...
8.        //一个 PTE 页表项占 8 字节(共 64bit)
9.        entry.val = copy_one_pte(dst_mm, src_mm, dst_pte, src_pte,
10.                                                vma, addr, rss);
11.       ...
12.       //指向下一个页表项
13.   } while (dst_pte++, src_pte++, addr += PAGE_SIZE, addr != end);
```

图 3-22　PTE 页表项的逐个复制

在页表复制过程中,还有一个关键步骤是将物理页标记为只读,确保父进程与新进程都不能随意地修改其内容,这可以通过修改 PTE 页表项的权限位实现,如图 3-23 所示。每个进程指向内核空间的页表都是相同的,所以只需修改用户空间部分的 PTE 页表项权限位。针对当前页表项,如果判定其最终指向的物理页不属于共享可写页,即支持写时复制映射(第 3 行),就会对父进程与新进程的当前 PTE 页表项进行写保护：先清除可写权限,再设成只读权限(第 15～17 行)。那么,任何一方在执行过程中尝试去向该表项指向的物理页写入内容都会触发缺页异常,避免直接修改共享页内容影响其他进程。

```
1.    //源文件：mm/memory.c
2.    //对于标有"写时复制映射"的内存区域,且当前指向的物理页是可写的：
3.    if (is_cow_mapping(vm_flags) && pte_write(pte)) {
4.        //父进程 PTE 权限改为写保护
5.        ptep_set_wrprotect(src_mm, addr, src_pte);
6.        pte = pte_wrprotect(pte);        //新进程 PTE 权限位改为写保护
7.    }
8.    //源文件：mm/internal.h
9.    //判断该页是否支持"写时复制"：
10.   static inline bool is_cow_mapping(vm_flags_t flags) {
11.       return (flags & (VM_SHARED | VM_MAYWRITE)) == VM_MAYWRITE;
12.   }
13.   //源文件：arc/arm64/include/asm/pgtable.h
14.   static inline pte_t pte_wrprotect(pte_t pte) {          //写保护设置
15.       pte = clear_pte_bit(pte, __pgprot(PTE_WRITE));      //清除可写权限
16.       pte = set_pte_bit(pte, __pgprot(PTE_RDONLY));       //设为只读权限
17.       return pte;
18.   }
```

图 3-23　PTE 页表项权限位改为只读

2) 写时复制触发的缺页异常处理

首先,异常处理器需要检查错误类型。在确认缺页异常是因为对写时复制页面执行了写操作引起的之后,异常处理器才能调用到对应的处理函数,为进程复制出一份当前页面的副本,或是修改页面权限。触发缺页异常的情况很多,例如进程试图访问本应无权访问的页面、进程访问的位置还没有分配物理页、需要访问的页面被换出到外存中等,异常处理器需要依次排除这些情况,才能定位到“写时复制”这种情况。如图 3-24 所示,异常处理器应该先进行访问权限判断,对于进程权限不足导致的缺页异常返回一个错误信号给进程,而不做其他处理(第 2～4 行)。接着,异常处理器需要判断 PTE 页表项是否为空,如果不为空,说明其确实映射着一个物理页。若此物理页仍在内存中,再做进一步判断,否则会请求从外存中将物理页换入。然后,异常处理器可以通过标志 FAULT_FLAG_WRITE 确定缺页异常是由写访问触发的(第 8 行),若 PTE 页表项的标志位 PTE_WRITE 也未置位,表明此页确实不可写(第 10 行),至此,异常处理器可以断定缺页异常是由对写时复制页面执行写操作导致的,进入相应的处理函数。

```
1.    //源文件: mm/memory.c
2.    if (!arch_vma_access_permitted(vma, flags & FAULT_FLAG_WRITE,
3.        flags & FAULT_FLAG_INSTRUCTION, flags & FAULT_FLAG_REMOTE))
4.        return VM_FAULT_SIGSEGV;              //权限不足,不做处理
5.    ...
6.    //PTE 不为空,且物理页还在内存中
7.    if (vmf->pte && pte_present(vmf->orig_pte))
8.       if (vmf->flags & FAULT_FLAG_WRITE)   //缺页异常是由写访问触发
9.           //PTE 页表项的 PTE_WRITE 标志位未置位,该页不可写
10.          if (!pte_write(entry)) {
11.             进入写时复制处理函数
12.          }
```

图 3-24　判断错误类型,找到写时复制处理函数

确定了缺页异常是由写时复制引起后,又存在两种情况:第一,当前有两个及以上的进程以只读形式共享该页,异常处理器会按图 3-25 所示步骤处理。在原只读共享页上肯定是不允许进程执行写操作的,异常处理器必须为其分配一个新的可写物理页(第 5 行)。新页需要复制触发缺页异常的旧页内容,因为进程希望的是对这部分内容进行修改而不是向一个空白页写入内容。接下来要做的是用新页替换掉触发异常的只读页与新进程页表的映射关系:先将新页地址保存在一个临时的 PTE 页表项,再把临时 PTE 页表项权限位设为可读可写(第 11～13 行),最后用临时 PTE 页表项内容覆盖新进程页表中原页表项内容(第 16 行)。此时,进程的写操作变成了分配新的物理页,不会再影响原来的共享页。第二,当前仅有一个进程在使用这个页面,即其他进程因写操作已取消与该页的映射关系,异常处理器只需要将页面的只读权限改为可读写即可。

```
1.    //源文件：mm/memory.c
2.    static vm_fault_t wp_page_copy(struct vm_fault * vmf) {
3.        ...
4.        //分配一个新物理页
5.        new_page = alloc_page_vma(GFP_HIGHUSER_MOVABLE, vma, vmf->address);
6.        ...
7.        //复制旧页内容到新页
8.        cow_user_page(new_page, old_page, vmf->address, vma);
9.        ...
10.       //使用新页地址与 vma 生成一个临时的 PTE 页表项
11.       entry = mk_pte(new_page, vma->vm_page_prot);
12.       ...
13.       entry = maybe_mkwrite(pte_mkdirty(entry), vma);     //临时 PTE 设为可读可写
14.       ...
15.       //将临时 PTE 内容写入页表中，即建立新页与新进程页面的映射关系
16.       set_pte_at_notify(mm, vmf->address, vmf->pte, entry);
17.       ...
18.   }
```

图 3-25　写时复制触发的缺页异常处理

3.3.3　程序装载

上面讲述了父进程通过调用函数 fork()完成一个新进程创建的过程。函数 fork()创建的新进程会完全复制其父进程的上下文，并映射其父进程的内存空间，从而回到与父进程调用函数 fork()后相同的执行点。此时，新创建的进程完全是其父进程的一个副本。然而，在大多数情况下，新进程创建后需要执行一个与其父进程不同的新程序来完成新的功能。那么，新创建的进程如何加载一个新程序呢？

当前操作系统采用的一种普遍方案是，先将各种程序编译成二进制可执行文件存在外存中，当有需要时，进程从外存中将所需的可执行文件内容装入地址空间，然后从中获取程序入口地址并开始执行新的程序指令。下面将以 ELF 文件为例展开介绍。openEuler 中由 exec 函数簇来实现该方案。此处将调用 exec 函数簇的进程称为调用进程。exec 函数簇用新程序替换调用进程地址空间中的程序实体，包括代码段、数据段，还为调用进程分配新的用户堆栈。并且，exec 函数簇会沿用调用进程的 PCB，除了修改其中部分描述资源的成员外，会保留包括进程标识符在内的大部分信息。所以，exec 函数簇并不会创建一个新进程，只是为调用进程分配了一个新任务。另外，除非执行失败会让 exec 函数簇中的函数返回－1，否则调用进程不会收到返回值，而是直接从新程序的主函数入口开始执行。

exec 函数簇的系列函数的挑战在于：①用户怎么告诉操作系统自己需要哪个程序？②操作系统怎么根据给定的信息去外存中找到所需的可执行文件？③操作系统怎么将找到的文件内容装载到调用进程的地址空间中？④操作系统怎么为进程构建新的执行环境，使其能从新程序的入口地址开始执行？

1. exec 函数簇的系列函数接口

用户必须通过 exec 接口向操作系统传递一些参数,例如文件名、文件路径、环境变量等。这样,操作系统内核才能去外存加载正确的可执行文件。这里的环境变量指 PATH,是多条文件路径的集合。当用户没有给出待加载程序的完整路径时,内核从环境变量 PATH 指定的路径中寻找程序。每个进程都有独立的环境变量。进程的环境变量大部分继承自其父进程,还有一部分由内核默认添加,进程也可以自己添加或修改自己的环境变量。在进程创建时,环境变量被压入用户栈中。

openEuler 提供了 6 种不同的 exec 函数接口,使用户在不同场景下可以调用合适的接口。用户通过这些接口将文件名、文件路径、环境变量等参数传递到内核,帮助其找到正确的可执行文件。这些接口的声明如下:

```
int execl(char const * path, char const * arg0, ...);
int execlp(char const * file, char const * arg0, ...);
int execle(char const * path, char const * arg0, ..., char const * envp[]);
int execv(char const * path, char const * argv[]);
int execvp(char const * file, char const * argv[]);
int execve(char const * path, char const * argv[], char const * envp[]);
```

这 6 个函数的作用相似,仅在使用规则上有细微差别。前三个函数希望传入以逗号分隔、以 NULL 结尾的参数列表;后三个函数希望传入一个指向由参数组成的字符串数组 argv 的指针;函数 execlp()与函数 execvp()只需在参数 file 中传入文件名,然后操作系统将从环境变量 PATH 指出的路径中查找该文件,而其他 4 个函数则需要在参数 path 中传入完整的文件路径;另外,函数 execle()与函数 execve()可以在参数 envp 中显式地指定环境变量。

在这 6 个函数中,前 5 个都是库函数,只有函数 execve()是系统调用函数。所以实际上,exec 函数簇中的函数最终都是调用函数 execve()去完成新程序的加载。

2. 可执行文件的寻找与打开

在 openEuler 中,文件系统的每个目录文件或文件都对应一个 inode 对象。inode 对象记录了一个文件的重要信息,包括文件字节数、拥有者、读写权限、在外存的位置等(将在第 7 章详细介绍)。用户将文件名传给内核,表明自己想要操作的文件。但是,inode 对象并不记录文件名,而是通过特定的 inode 号码与文件相关联,这有利于文件移动、重命名、更新等操作。为了建立文件名与 inode 对象的联系,内核还提供了一个 dentry 对象。通过 exec 函数簇的接口可知,用户可能使用文件路径来帮助内核寻找文件。所以,openEuler 又实现了一个路径查找辅助结构体 nameidata,用于根据路径名寻找所需目录或文件的 dentry 对象。

由此,可以简要描述出内核打开可执行文件的流程,并结合源码(图 3-26)做简要分析。

(1) 文件路径(包括文件名)会由用户调用接口传递到内核。该路径通常不是完整路径,所以内核需要获取进程当前的工作目录作为查找的起始目录(第 5 行)。

```
1.    //源文件: fs/namei.c
2.    //(1)确定查找的起始目录,假定用户未给出完整路径
3.    static const char * path_init(struct nameidata * nd, unsigned flags) {
4.        …
5.        nd->path = fs->pwd;                                  //获取当前进程的工作目录
6.        nd->inode = nd->path.dentry->d_inode;                //该目录对应的虚拟文件系统 inode
7.        …
8.    }
9.    //源文件: fs/namei.c
10.   //(2)解析文件路径中除最后分量外的每个分量,根据路径名得到对应 dentry 对象
11.   static int link_path_walk(const char * name, struct nameidata * nd) {
12.       …
13.       for(;;) {
14.           hash_len = hash_name(nd->path.dentry, name);
15.           struct dentry * parent = nd->path.dentry;  //获取父目录的目录项
16.           nd->last.hash_len = hash_len;              //设置当前分量的长度
17.           nd->last.name = name;                      //设置当前分量的路径名
18.           nd->last_type = type;                      //设置当前分量的类型
19.           name += hashlen_len(hash_len);             //加上当前分量长度,指向下一分量
20.           if (!*name)  goto OK;
21.           …
22.
23.   OK:
24.           //获取当前目录项的 dentry 对象以及 inode 对象,更新到 nameidata 对象 nd
25.           err = walk_component(nd, WALK_FOLLOW | WALK_MORE);
26.           …
27.           if (!name)  return 0;          //已经找到最后一个分量,结束解析,否则继续循环
28.       }
29.       …
30.   }
31.   //源文件: fs/file_table.c
32.   //(3)根据最后分量,打开对应的可执行文件,并填充 file 结构
33.   static struct file * alloc_file(const struct path * path, int flags,
34.                                   const struct file_operations * fop) {
35.       …
36.       file->f_path = *path;
37.       file->f_inode = path->dentry->d_inode;
38.       file->f_mapping = path->dentry->d_inode->i_mapping;
39.       file->f_op = fop;
40.       return file;
41.       …
42.   }
```

图 3-26　寻找和打开可执行文件的关键代码

(2) 内核会从起始目录开始,对路径中的每个分量进行解析,找到各分量对应的 dentry 对象和 inode 对象(第13～28 行)。例如,路径是 a/b/c.txt,那 a、b、c.txt 都是该路径的分量。内核借助结构体 nameidata,最终将找到最后一级分量对应的 dentry 对象。然后,内核

从 dentry 对象中获得 inode 号码,进而找到对应的 inode 对象。

(3) 内核从 inode 对象中获得目标文件在外存中的位置,并打开该文件。打开文件过程中,有一个关键步骤是内核创建 file 对象,并将该打开文件的路径、inode 对象、文件打开模式等信息填充到 file 对象中(第 36～40 行)。之后,内核可以通过该 file 对象直接访问和管理该打开的文件。

3. 可执行文件的装载

openEuler 中的可执行文件大多是 ELF 格式,它的格式在 3.1.1 节中已介绍。装载可执行文件的关键代码如图 3-27 所示。ELF 文件的 header(头部)包含整个文件的基本信息,包括文件大小、Program Header(段头)位置、执行入口地址等。ELF header 通常是 ELF 文件的前 256 字节,该部分会被内核率先读入缓存区 bprm－> buf 中。其中,bprm 是 struct linux_binprm 结构,专用于二进制文件加载过程中参数、数据等的临时保存。接着,内核从

```
1.    //源文件: fs/binfmt_elf.c
2.    static int load_elf_binary(struct linux_binprm * bprm) {
3.        loc -> elf_ex = *((struct elfhdr * )bprm -> buf); //转成 ELF 的 header 格式
4.        ...
5.        //从 ELF 文件中读出 Program Header
6.        elf_phdata = load_elf_phdrs(&loc -> elf_ex, bprm -> file);
7.        //遍历 Program Header,把 ELF 中代码段、数据段等映射到内存中
8.        for(i = 0, elf_ppnt = elf_phdata; i < loc -> elf_ex.e_phnum;
9.                                                  i++, elf_ppnt++) {
10.           ...
11.           vaddr = elf_ppnt -> p_vaddr;
12.           load_bias = ELF_ET_DYN_BASE;                //load_bias 为实际映射的起始地址
13.           load_bias = ELF_PAGESTART(load_bias - vaddr);     //找到正确偏移值
14.           total_size = total_mapping_size(elf_phdata,loc
15.                                          -> elf_ex.e_phnum);
16.           ...
17.           //映射当前段等内容到内存空间
18.           error = elf_map(bprm -> file, load_bias + vaddr, elf_ppnt,
19.                           elf_prot, elf_flags, total_size);
20.           ...
21.       }
22.   }
23.   //源文件: fs/binfmt_elf.c
24.   static struct elf_phdr * load_elf_phdrs(struct elfhdr * elf_ex,
25.                                          struct file * elf_file) {
26.       ...
27.       size = sizeof(struct elf_phdr) * elf_ex -> e_phnum;       //Program Header 大小
28.       elf_phdata = kmalloc(size, GFP_KERNEL);                   //分配 size 大小的内存区域
29.       retval = kernel_read(elf_file, elf_phdata, size, &pos);  //从文件读内容
30.       ...
31.       return elf_phdata;
32.   }
```

图 3-27　装载可执行文件的关键代码

ELF header 中获取 Program Header 地址，并从 ELF 文件中读出其内容（第 6 行、第 27～31 行）。Program Header 中记录了 ELF 文件各个 Segment（段）的地址及大小。所以，内核可以借助这些内容，通过一个 for 循环将 ELF 文件的代码段、数据段等都映射到内存中，并将相应信息记录在调用进程的内存描述符对应成员中（第 8～21 行）。

4. 新程序的执行

内核将代码段、数据段等映射到内存后，还需要更新这些 Segment 在地址空间中的起始地址和结束地址。之前，这些地址只是偏移地址，加上映射在内存中的起始地址 load_bias 后，就真正指向各个 Segment（见图 3-28 中代码第 5～11 行）在地址空间中的位置。然后，内核将从 ELF header 中获取的入口地址设为程序执行的入口地址（第 13～15 行）。

```
1.    //源文件：fs/binfmt_elf.c
2.    static int load_elf_binary(struct linux_binprm * bprm) {
3.        ...
4.        //调整程序各个 segment 的具体位置
5.        loc->elf_ex.e_entry += load_bias;       //主函数入口地址
6.        elf_bss += load_bias;                   //bss 段起始地址
7.        elf_brk += load_bias;
8.        start_code += load_bias;                //代码段起始地址
9.        end_code += load_bias;
10.       start_data += load_bias;                //数据段起始地址
11.       end_data += load_bias;
12.       ...
13.       elf_entry = load_elf_interp(&loc->interp_elf_ex, interpreter,
14.                                   &interp_map_addr,load_bias,interp_elf_phdata);
15.       interp_load_addr = elf_entry;           //设置执行入口地址
16.       ...
17.       retval = create_elf_tables(bprm, &loc->elf_ex,
18.                       load_addr, interp_load_addr);
19.       ...
20.   }
21.   //源文件：fs/binfmt_elf.c
22.   //进一步设置堆栈，例如辅助向量、环境变量、程序参数等入栈
23.   static int create_elf_tables(struct linux_binprm * bprm,
24.              struct elfhdr * exec, unsigned long load_addr,
25.                        unsigned long interp_load_addr) {
26.       ...
27.       elf_info = (elf_addr_t * )current->mm->saved_auxv;
28.       sp = (elf_addr_t __user * )bprm->p;     //sp 指向的是用户栈栈顶
29.       __put_user(argc, sp++);                 //将参数的总数入栈
30.       p = current->mm->arg_end = current->mm->arg_start;
31.       while (argc-- > 0) {                    //让参数逐一入栈
32.            __put_user((elf_addr_t)p, sp++);
33.            len = strnlen_user((void __user * )p, MAX_ARG_STRLEN);
34.            p += len;
```

图 3-28　执行新程序的关键代码

```
35.        }
36.        ...
37.        current->mm->env_end = current->mm->env_start = p;
38.        while (envc-- > 0) {              //环境变量逐一入栈
39.             __put_user((elf_addr_t)p, sp++);
40.             len = strnlen_user((void __user *)p, MAX_ARG_STRLEN);
41.             p += len;
42.        }
43.        ...
44.        //auxiliary vector 入栈
45.         copy_to_user(sp, elf_info, ei_index * sizeof(elf_addr_t));
46.        ...
47.    }
48.    //源文件: arch/arm64/include/asm/processor.h
49.    //配置寄存器环境,开始执行
50.    static inline void start_thread(struct pt_regs *regs,
51.                    unsigned long pc, unsigned long sp) {
52.        start_thread_common(regs, pc);    //pc 保存的是即将执行的指令地址
53.        regs->pstate = PSR_MODE_EL0t;
54.        ...
55.        regs->sp = sp;
56.    }
```

图 3-28 (续)

虽然新程序的数据都被加载到内存中,但是,由 3.1.1 节可知,程序的执行还需要用户堆栈、寄存器的配合。内核需要将辅助向量(auxiliary vector)、环境变量、用户参数等一一入栈(第 27～45 行),构建如图 3-29 所示栈。

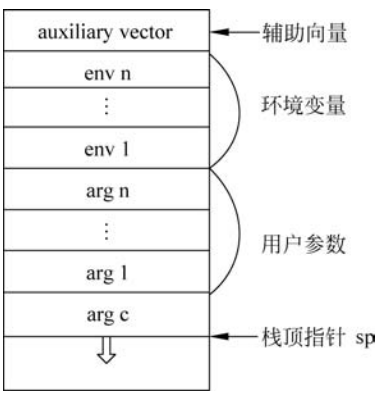

图 3-29 初始用户栈的内容构建

因为在程序运行过程中,这些参数很可能被用到。其中,辅助向量是一种从内核到用户空间的信息交流机制。最后,内核还需为结构体 pt_regs 中各寄存器成员设置正确的值,包括 PC、PSTATE、SP 等(第 52～55 行)。这些成员通常保存着程序运行中断时的 CPU 上下文。当程序装载完毕、进程正常执行时,结构体 pt_regs 中各成员内容会被写到 CPU 的各个寄存器中。由此,程序可以从程序计数器 PC 中保存的主函数入口地址开始执行。

3.3.4 进程终止

每个进程都有自己的生命周期,总会有需要终止的时候。导致进程终止的原因有许多:可能是进程正常结束,例如主函数最后一条指令执行完毕,或者是进程主动执行终止进程的系统调用,如 exit();也可能是进程资源使用超限、在内核态发生不可处理异常、收到不可

处理或忽视的信号等原因，导致进程被操作系统强行终止。终止后的进程会被操作系统销毁，彻底退出。

无论进程因何种原因终止，操作系统都需要回收其占用的所有资源，这部分操作也是在内核中完成。在进程终止发生时，内核先回收该进程拥有的大部分资源，例如关闭进程打开的文件、回收分配给进程的物理内存并解除内存映射等，这些资源都属于用户空间的基本资源。在内核完成前面的操作之后，进程还占有着内核栈、PCB 等内核资源，但内核不会将这部分资源也直接回收掉。这是为了提供关于终止进程的一些状态信息，例如进程是正常终止还是异常终止，创建后运行多长时间等，这些信息通常由其父进程收集。父进程接收到终止进程的状态信息后，将陷入内核去回收其剩余资源。所以，内核只需将终止进程的相关状态信息发送给它的父进程，然后让终止进程进入僵尸状态。那么，剩下的回收工作自然会由父进程完成。

在 openEuler 中，父进程创建子进程后，通常使用系统调用 wait()或 waitpid()进入阻塞状态，直到获取了子进程终止信号才继续执行。其中，系统调用 wait()等待任一子进程的终止，而系统调用 waitpid()只等待进程 PID 与参数 pid 相同的子进程。当参数 pid 为－1 时，这两个系统调用等效。另外，如果 waitpid()的参数中传入标志 WNOHANG，那么父进程是以一种非阻塞的方式回收子进程。下面以 wait()为例展开介绍。得到子进程终止的信号后，父进程会陷入内核去回收子进程的剩余资源。在父进程调用 wait()之前，处于僵尸状态的子进程称为僵尸进程，它们持有着一定的内核资源但又不会被任何信号结束。如果编程人员忘记在父进程中使用 wait()，或是父进程在调用 wait()前异常终止，都可能导致系统中存在大量僵尸进程。这些僵尸进程会占用大量的内核资源，影响系统性能，此外，它们的进程标识符也不会被回收，甚至可能导致系统因没有可用的进程标识符而无法创建新进程。

为了降低僵尸进程的危害，openEuler 采取了许多策略，其中有 4 种策略比较重要：①父进程终止时会自动调用 wait()帮助处于僵尸状态的子进程完成终止；②如果父进程提前因异常终止，那么 init 进程会定期调用 wait()清除僵尸进程；③为了有效避免僵尸进程的产生，如果进程不被跟踪，内核将不会向其父进程发送状态信息而直接回收该进程全部非共享资源(线程的相关概念将在 3.6 节介绍)；④某些父进程并不负责子进程剩余资源回收，在这种情况下，内核在向父进程发送状态信息后，会直接回收子进程的剩余资源，使得子进程不进入僵尸状态而是变为死亡状态。

在 openEuler 中，进程终止主要是通过系统调用 exit()与 wait()配合来完成的。exit()主要完成两部分工作：①回收当前进程的用户空间所占用的内存资源；②向当前进程的父进程发送状态信息，并判断父进程是否愿意回收当前进程的剩余资源。如果父进程的 SIGHAND 信号处理函数设为 SIG_IGN 或是处理标志设置了 SA_NOCLDWAIT，表明父进程不关心当前进程的终止，那么 exit()会继续回收当前进程剩余内核资源；否则，让当前进程进入僵尸状态，并等待父进程调用 wait()去接收这些信息并完成资源回收。然而，当前进程可能还存在多个运行正常的子进程(未成为僵尸进程)，这些失去父进程的子进程称为孤儿进程。内核需要为孤儿进程寻找新的父进程，用于接收和处理孤儿进程的状态信息并在它

们终止时予以帮助。下面结合 openEuler 的代码具体介绍进程终止和资源回收的详细步骤。

1. 用户空间资源的回收

在进程终止时,内核优先回收的是当前进程所占用且不与其他进程共享的用户空间资源,下面主要以回收内存资源为例展开介绍。映射到进程用户空间的物理内存有三个组成部分:分配到的物理内存、页表占用的物理内存、内存描述符占用的物理内存。所以,内核也需要分三个过程回收这些资源。内存资源回收的关键代码如图 3-30 所示。

```
1.    //源文件: mm/memory.c
2.    //清空页表项,释放分配的物理页
3.    static unsigned long zap_pte_range(struct mmu_gather *tlb,
4.        struct vm_area_struct *vma, pmd_t *pmd, unsigned long addr,
5.            unsigned long end, struct zap_details *details) {
6.        ...
7.        start_pte  = pte_offset_map_lock(mm, pmd, addr, &ptl);   //获取 PMD 锁
8.        pte = start_pte;
9.        do {
10.           pte_t ptent =  *pte;
11.            ...
12.           //找到 ptent 页表项指向的物理页
13.           page = _vm_normal_page(vma, addr, ptent, true);
14.           //清除 PTE 值,并返回原页表项内容到 ptent
15.           ptent = ptep_get_and_clear_full(mm, addr, pte, tlb->fullmm);
16.           ...
17.           page_remove_rmap(page, false);                          //解除映射关系
18.           put_page(page);                                         //释放物理页
19.           ...
20.   } while (pte++, addr += PAGE_SIZE, addr != end);
21.   //源文件: mm/memory.c
22.   //回收页表占用的物理页,以回收 PMD 为例,其他级页表类似
23.   static inline void free_pmd_range(struct mmu_gather *tlb, pud_t *pud,
24.                                     unsigned long addr, unsigned long end,unsigned long floor,
25.                                                                   unsigned long ceiling) {
26.       ...
27.       pmd = pmd_offset(pud, addr);
28.       do {
29.           next = pmd_addr_end(addr, end);   //获取下一个页表项
30.           ...
31.           free_pte_range(tlb, pmd, addr);    //释放 PMD 页表项指向物理页
32.       } while (pmd++, addr = next, addr != end);
33.       ...
34.       pmd = pmd_offset(pud, start);          //回到当前 PMD 页表所在物理页起始地址
35.       pud_clear(pud);
36.       pmd_free_tlb(tlb, pmd, start);         //释放 PMD 页表所在物理页
37.   }
38.   //源文件: kernel/fork.c
39.   //回收内存描述符
40.   #define free_mm(mm)  (kmem_cache_free(mm_cachep, (mm)))
```

图 3-30　内存资源回收的关键代码

首先，分配给进程的物理内存都会与进程页表建立映射关系。所以，PTE 页表中的每个不为空的 PTE 页表项都指向一个待回收的物理页。内核会遍历进程的用户空间页表，将所有 PTE 页表项清 0，对于不为空的页表项，找到其指向的物理页（第 13 行）。同一个物理页可以被映射到多个进程的地址空间中，所以内核需要先检查物理页的引用计数，只有引用计数为零的物理页才能被释放。

清除所有 PTE 页表项，意味着解除了当前进程页表与物理内存的映射关系，那么其页表所占用的内存也可以回收了。在获取上一级页表的页表项内容后，内核将该页表项置 0，并根据其内容找到下一级页表所在物理页，在判断物理页引用计数为 0 后释放它（第 34～36 行）。

最后，还剩下内存描述符未回收。内存描述符、进程标识符、PCB 以及内核栈等内核重要数据结构对象通常比较小且需要频繁分配和回收。为了加速对它们的管理，这些对象通常被组织为 slab 块。内核在回收 slab 块时，并不会直接释放它们占用的物理内存，而是将其作为一个空闲对象回收到 slab 块的空闲链表中。在新进程创建时，这些空闲对象可以被重新分配，以此加快进程创建速度。当内核中的空闲物理内存不足时，slab 分配器才会选择释放部分空闲对象实际占用的物理内存。最后，内核将当前内存描述符回收到结构体指针 mm_cachep 指向的 slab 空闲链表中（第 40 行）。

2. 状态信息的发送与僵尸状态的设置

回收了大部分资源后，即使当前进程被调度器再次选中，也无法再运行，所以内核会将当前进程设为僵尸状态（EXIT_ZOMBIE），使其被调度器忽略。之后，内核还需要向当前进程的父进程发送 SIGCHLD 信号与状态信息。发送状态信息的关键代码如图 3-31 所示。SIGCHLD 信号用于提醒父进程去回收子进程剩余资源，状态信息主要包括 pid、exit_code 以及当前进程运行时间等（第 4～12 行）。其中，exit_code 可以帮助父进程判断当前进程是正常终止还是异常终止。此时，父进程可能正因系统调用 wait()而处于阻塞状态。因此，内核还需要唤醒父进程，使其可被调度并接收信号（第 16 行）。

```
1.    //源文件：kernel/signal.c;kernel/exit.c
2.    //向父进程发送的信息保存在 info 结构体中，参数 sig 传入的是 SIGCHLD
3.    bool do_notify_parent(struct task_struct *tsk, int sig) {
4.        struct siginfo info;
5.        ...
6.        clear_siginfo(&info);
7.        info.si_signo = sig;
8.        info.si_errno = 0;
9.        info.si_pid = task_pid_nr_ns(tsk, task_active_pid_ns(tsk->parent));
10.       info.si_utime = nsec_to_clock_t(utime + tsk->signal->utime);
11.       info.si_stime = nsec_to_clock_t(stime + tsk->signal->stime);
12.       info.si_status = tsk->exit_code & 0x7f;
13.       ...
14.       __group_send_sig_info(sig, &info, tsk->parent);          //将状态信息发父进程
15.       ...
16.       __wake_up_parent(tsk, tsk->parent);          //唤醒因 wait()进入阻塞状态的父进程
17.   }
```

图 3-31　发送状态信息的关键代码

3. 内核资源的回收

无论是由内核还是由父进程来为子进程回收内核资源,内核需要完成的步骤都是一样的。内核资源回收的关键代码如图 3-32 所示。第一步就是回收子进程占用的 PID(进程标识号)。为了能根据 PID 快速索引到 PCB,openEuler 引入了 PID 散列表。在子进程终止后,内核会先从 PID 散列表中删除它的 PID,然后将该 PID 回收到 slab 的空闲链表中(第 5～7 行)。

```
1.    //源文件: kernel/pid.c
2.    //从 PID 散列表中删除当前进程标识符
3.    static void __change_pid(struct task_struct * task, enum pid_type type,
4.                                                        struct pid * new) {
5.        hlist_del_rcu(&task->pid_links[type]);
6.        ...
7.        free_pid(pid);//回收该进程标识符到 slab cache 中
8.    }
9.    //源文件: tools/include/linux/list.h
10.   //从进程链表中删除当前 PCB
11.   static inline void __list_del(struct list_head * prev,
12.                                 struct list_head * next) {
13.       next->prev = prev;
14.       WRITE_ONCE(prev->next, next);
15.   }
16.   ...
17.   //源文件: kernel/fork.c
18.   //回收内核栈
19.   static inline void free_thread_stack(struct task_struct * tsk) {
20.       ...
21.       __free_pages(virt_to_page(tsk->stack), THREAD_SIZE_ORDER);
22.   }
23.   //源文件: kernel/fork.c
24.   //回收 PCB
25.   static inline void free_task_struct(struct task_struct * tsk) {
26.       kmem_cache_free(task_struct_cachep, tsk);
27.   }
```

图 3-32　内核资源回收的关键代码

openEuler 内核中存在进程链表(一个双向循环链表),用于链接所有进程的 PCB。所以回收 PCB 之前,内核需要先将其从进程链表中删除(第 11～15 行)。之后,内核再把子进程的内核栈、PCB 回收到 slab cache 的空闲链表中(第 19～27 行),其中,内核栈对应结构体指针 thread_stack_cache 指向的链表,而 PCB 对应结构体指针 task_struct_cachep 指向的链表。

4. 为所有子进程寻找新父进程

由于僵尸进程会一直占用内核资源进而影响新进程的创建,所以在当前终止的进程还有大量子进程未终止的情况下,内核必须为它们寻找新的父进程,否则这些子进程终止时可

能会变成无法销毁的僵尸进程，带来巨大的危害。内核会优先从当前进程所在的进程组中寻找合适的父进程。内核遍历该进程组，选中第一个未执行终止操作的进程（即进程标志位不是 PF_EXITING）。如果进程组内除 init 进程外的其他进程都已开始终止，那么当前进程的所有子进程会被挂载到 init 进程下。之后，某个子进程终止时，会等待 init 进程调用 wait()完成它的终止。图 3-33 展示了 openEuler 中为子进程寻找新父进程的部分代码。

```
1.    //源文件：kernel/exit.c
2.    static struct task_struct * find_new_reaper(struct task_struct * father,
3.                                        struct task_struct * child_reaper) {
4.        struct task_struct * p, * t, * reaper;
5.        …
6.        //找一个合适的 reaper(新父进程)
7.        for (reaper = father -> real_parent;
8.            task_pid(reaper) -> level == ns_level;
9.            reaper = reaper -> real_parent) {      //从当前进程所在进程组中寻找
10.           if (reaper == &init_task)              //找到 init 进程
11.               break;
12.           …
13.           thread = find_alive_thread(reaper);  //寻找不处于终止状态的进程
14.           if (thread) return thread;
15.   }
16.   //源文件：kernel/exit.c
17.   static struct task_struct * find_alive_thread(struct task_struct * p) {
18.       for_each_thread(p, t) {
19.           if (!(t -> flags & PF_EXITING)) {
20.               return t;
21.       }
22.       return NULL;
23.   }
24.   //源文件：kernel/exit.c
25.   //让 reaper 成为当前进程所有子进程的父进程
26.   static void forget_original_parent(struct task_struct * father,
27.                                      struct list_head * dead) {
28.       list_for_each_entry(p, &father -> children, sibling) {
29.           for_each_thread(p, t) {
30.               t -> real_parent = reaper;
31.               …
32.           }
33.       }
34.   }
```

图 3-33　为子进程寻找新父进程的关键代码

3.3.5　openEuler 中的进程树

openEuler 中的各个进程通过创建的先后顺序，组成了一棵进程树。进程树的整个创建流程见图 3-34。在 openEuler 启动后，内核会使用静态数据 init_task 创建第一个进程，其

PID 为 0。0 号进程完成内核初始化(包括初始化页表、中断处理表、系统时间等)后,会调用函数 kernel_thread()创建 1 号进程与 2 号进程。此时,三个进程都运行在内核态且无用户空间。之后,0 号进程演变为 idle 进程,一直运行在内核态中。而 1 号进程会完成剩下的系统初始化工作,接着执行/sbin/init 程序,初始化用户空间,成为 init 进程,运行在用户态下。init 进程就是之后操作系统中所有用户进程的共同祖先,它与所有用户进程共同构成一棵倒立的进程树。init 进程还负责孤儿进程的管理与回收。2 号进程又被称为 kthreadd 内核线程,它会一直运行在内核空间,对之后所有内核线程进行管理和调度。

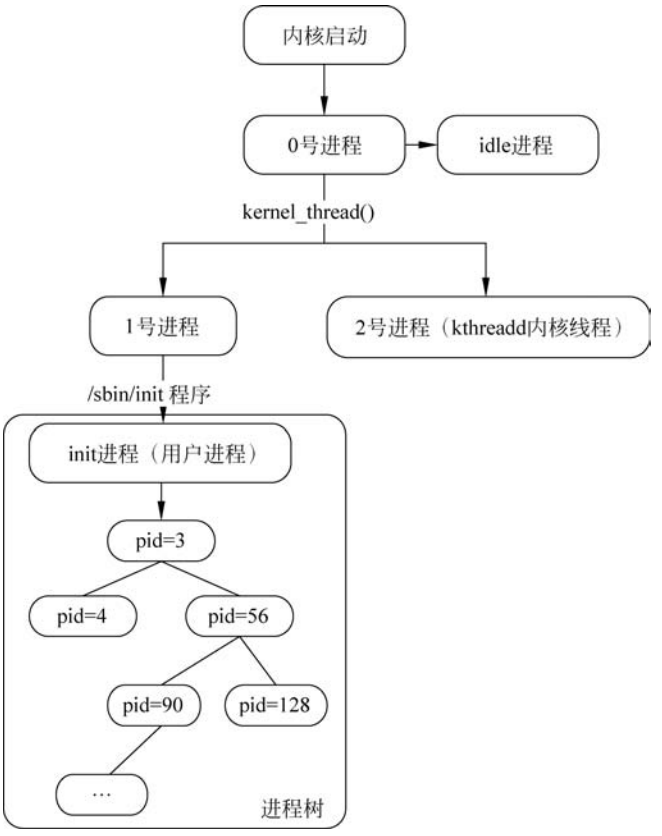

图 3-34　进程树建立过程

3.4　系统调用

为了让不同进程合理共享硬件资源,操作系统需要保持对硬件资源的管理。操作系统通过系统调用向进程提供服务接口,限制进程直接进行硬件资源操作。如果希望执行受限

操作，进程只能调用这些系统调用接口，向操作系统传达服务请求，并将 CPU 控制权移交给操作系统。操作系统接收到请求后，再调用相应的处理程序完成进程所请求的服务。

3.4.1　基本概念

操作系统通过引入进程，允许多个程序共享 CPU 等硬件资源。那么，进程之间应该以何种方式使用机器上的硬件资源呢？假如允许进程直接地使用、控制硬件资源，将导致很多问题。例如，若某个进程在执行期间修改了用于存储异常向量表基址的寄存器 VABR_EL1，则将导致操作系统因找不到异常向量表而无法正常响应异常，依赖异步异常工作的键盘和鼠标会失灵，整个系统也会因无法处理异常而崩溃。另外，如何确保进程能够在合适时机释放 CPU 控制权？为解决不同进程之间共享硬件资源的问题，采取的基本思路是：操作系统保持对 CPU 的控制权，并负责硬件资源的管理；限制进程直接操作硬件资源；在进程希望执行受限的操作时，进程需要将 CPU 控制权移交给操作系统，由操作系统执行进程所请求的服务。通过这种方式，进程之间以受控的方式来共享资源。

然而，在实现上述思想时面临以下关键挑战：①如何限制进程所能进行的操作？②操作系统如何为进程服务？现代计算机基于处理器硬件和操作系统软件的协作来解决这两个挑战。

在硬件方面，CPU 模式分为内核模式与用户模式，并用寄存器的特定字段标识当前模式。例如，基于 ARMv8 架构的 CPU 有异常级别 EL1（内核模式）和 EL0（用户模式），寄存器 CurrentEL 的字段 EL 标识了 CPU 当前所处的异常级别。CPU 在内核模式执行的是内核态代码，在用户模式执行的是用户态代码。操作系统的内核就运行在内核态，而操作系统的其他部分及用户进程运行在用户态。在内核态运行的代码，对资源的操作不受限制，能够执行所有的指令、操作所有的寄存器和内存空间、发出 I/O 请求；而在用户态运行的代码，对资源的操作受到限制，不能执行可能对其他用户进程或内核造成威胁的指令，不能进行磁盘 I/O，而且也只能访问部分内存空间。CPU 模式将内核与用户进程的操作隔离在两个不同的世界。此外，为了使能用户态代码调用内核态代码，以安全地执行需要高权限的操作，CPU 提供一些陷阱指令用于模式的切换。例如，基于 ARMv8 架构的 CPU 提供陷入指令 SVC（Supervisor Call）和陷阱返回指令 ERET，分别用于从 EL0 陷入 EL1，以及从 EL1 返回 EL0（详细信息见第 2 章），如图 3-35 所示。当用户态的代码试图执行需要高权限的操作，就会触发异常，导致 CPU 模式的切换。

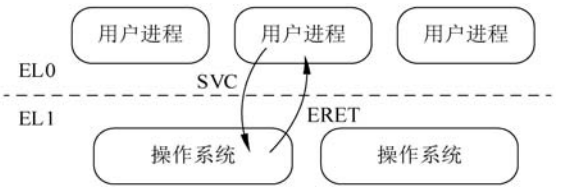

图 3-35　CPU 模式切换示意图

在软件方面,将受限的操作设计成内核功能,并以系统调用的形式暴露给用户进程。用户通过暴露的系统调用接口来使用操作系统的关键功能。当用户进程在执行的过程中调用了系统调用接口,将触发异常,导致 CPU 陷入内核模式,同时硬件将 CPU 的控制权移交给预先设定的异常处理程序。异常处理程序进而将该系统调用派发给内核中该系统调用的处理程序。当操作系统完成用户进程所请求的服务,再通过陷阱返回指令使 CPU 回到用户模式,并将 CPU 控制权移交给用户进程。

3.4.2 系统调用的实现

在不同的硬件架构上、不同的操作系统中,系统调用的实现细节有所不同。下面以一个常用的系统调用 sys_getpid()为例,阐述系统调用在 openEuler 中的实现。

系统调用 sys_getpid()可返回当前进程的 PID。进程通信中大量使用 PID,例如,进程在运行过程中可能会产生一些临时文件,为防止多个进程产生的临时文件名命名冲突,一般会使用各个进程唯一的 PID 来命名临时文件。然而,进程的 PID 存储在 PCB 中,而 PCB 是不允许用户进程任意访问或修改的(例如,若该进程修改其运行时间,在时间片轮转调度下,该进程就可以一直获得 CPU 的使用权);因此,为了满足用户程序获取其 PID 的需求,同时又限制其对 PCB 的访问,操作系统通过服务函数 sys_getpid()获得当前进程的 PID 并返回给应用程序。

系统调用实现的关键步骤如图 3-36 所示,包括库函数调用、异常处理、系统调用服务函数的查找、服务函数的执行和异常返回。

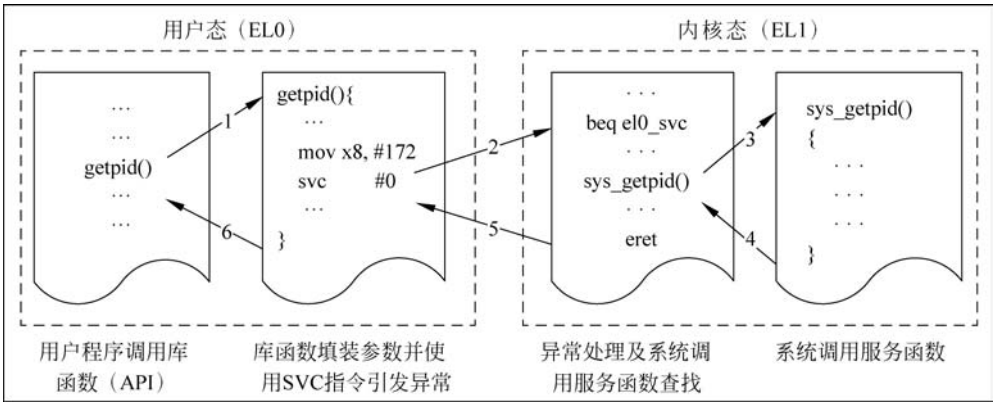

图 3-36 系统调用实现的关键步骤

1. 库函数调用

在用户态下,用户进程需配置参数寄存器并调用指令 SVC 去发起系统调用。然而,这一系列操作需要使用汇编语言编写,对编程人员来说不太友好。因此,glibc 函数库对系统调用进行了封装,屏蔽了指令 SVC 与参数传递等细节,仅向用户进程提供一个库函数。编

程人员只需调用该库函数就可以使用 openEuler 提供的系统调用服务。图 3-37 是使用库函数 getpid()的 C 程序示例。

```
1.    int main() {
2.        int pid = getpid();
3.        printf("pid = %d/n",pid);
4.        return 0;
5.    };
```

图 3-37　使用库函数 getpid()的 C 程序示例

库函数 getpid()与用户程序一样工作在异常级别 EL0,其需要完成两项任务：参数填充和触发系统调用。库函数将高级语言传递的参数通过汇编语言赋值给特定的寄存器,向内核传递系统调用参数与系统调用号。在 openEuler 中,待传递的参数保存在寄存器 X0～X6 中,系统调用号保存在寄存器 X8 中。完成参数填充后,库函数调用指令 SVC(AArch64 状态下)以触发异常。库函数 getpid()的实现如图 3-38 所示。

```
1.    ENTRY(__getpid)
2.        mov     x8, __NR_getpid   //将系统调用号存入 x8
3.        svc     #0                //使用指令 SVC 引发异常
4.        ...
5.        ret                       //返回应用程序
6.    END(__getpid)
```

图 3-38　库函数 getpid()的实现

2. 异常处理

用户进程执行指令 SVC 时就会产生一个异常。CPU 会先对该异常进行初步处理,自动执行以下操作：

(1) 将 PSTATE 相关寄存器的数据作为字段内容一起存入寄存器 SPSR_EL1 中。

(2) 将返回地址保存到寄存器 ELR_EL1 中,使得当该进程从异常处理程序返回时可以从它离开的地方继续执行。对系统调用而言,保存的是系统调用发生时即将执行的下一条指令的地址。

(3) 将异常屏蔽寄存器的 4 个掩码位 DAIF 置为 1,即关中断。

(4) 如果是同步异常,将生成异常的原因保存到寄存器 ESR_EL1 中。

(5) 将寄存器组 PSTATE 中寄存器 CurrentEL 的字段 EL 置为 1,即把异常级别提升到 EL1。

在 CPU 完成初步处理之后,操作系统需执行异常处理程序来进一步处理该异常。为了能找到对应的异常处理程序,内核维护着异常向量表。内核可以根据异常类型号从异常向量表中快速找到异常处理程序的入口。在 MMU 初始化时,操作系统将该表的首地址存到向量基址寄存器 VBAR_EL1 中,之后 CPU 直接通过该寄存器获取异常向量表首

地址。

在完成前面的初步处理之后,CPU 会自动读取寄存器 VBAR_EL1 以获取异常向量表基址,把此基地址加上异常类型对应的偏移量,即可在异常向量表中找到相应的异常处理函数入口地址,随后可跳转进入此处理函数。图 3-39 展示了部分异常向量表。指令 SVC 引发同步异常,故跳转至 sync,即同步异常处理函数(第 4 行)。

```
1.    //源文件: arch/arm64/kernel/entry.S
2.    ENTRY(vectors)
3.        ...
4.        kernel_ventry   0, sync                 //64 位 EL0 同步异常
5.        kernel_ventry   0, irq                  //IRQ 64 位 EL0
6.        kernel_ventry   0, fiq_invalid          //FIQ 64 位 EL0
7.        kernel_ventry   0, error                //Error 64 位 EL0
```

图 3-39　部分异常向量表

同步异常处理函数可执行三个步骤:①CPU 状态保存;②触发异常原因判断;③调用相应异常处理函数及传递参数。

1) CPU 状态保存

操作系统进行同步异常处理时,首先应保存进程在用户状态时的 CPU 状态,以备异常处理结束并返回用户态后,能从中断发生的地方继续执行该进程。同步异常处理函数的第一步是调用 CPU 状态保存函数 kernel_entry(),如图 3-40 所示。

```
1.    //源文件: arch/arm64/kernel/entry.S
2.    el0_sync:
3.        kernel_entry 0                      //保存用户进程的信息
4.        mrs x25, esr_el1                    //将 ESR 寄存器的内容读到 x25 寄存中
5.        lsr x24, x25, #ESR_ELx_EC_SHIFT     //获取异常产生原因
6.        //定义 ESR_ELx_EC_SVC64 = (0x15)
7.        cmp x24, #ESR_ELx_EC_SVC64          //比较是否为指令 SVC 产生的同步异常
8.        b.eq    el0_svc                     //跳转到 el0_svc 处理函数
```

图 3-40　同步异常处理

存储用户态下进程相关 CPU 状态的寄存器有 X0～X29、LR(X30)、SP_EL0、ELR_EL1、SPSR_EL1,这些寄存器依次被存入一片由内核管理的内存空间(内核栈)中,形成 pt_regs 栈帧,其结构如图 3-41 所示。而函数 kernel_entry()的执行过程就是将这些寄存器依次压入内核栈中,其在 openEuler 中的实现代码如图 3-42 所示。

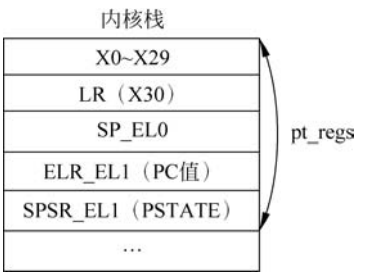

图 3-41　pt_regs 栈帧示意图

```
1.    //源文件: arch/arm64/kernel/entry.S
2.    stp  x0, x1, [sp, #16 * 0]    //将寄存器 X0、X1 内两个双字数据存放到 sp + 16 * 0
3.    stp  x2, x3, [sp, #16 * 1]
4.    stp  x4, x5, [sp, #16 * 2]
5.    stp  x6, x7, [sp, #16 * 3]
6.    stp  x8, x9, [sp, #16 * 4]
7.    stp  x10, x11, [sp, #16 * 5]
8.    stp  x12, x13, [sp, #16 * 6]
9.    stp  x14, x15, [sp, #16 * 7]
10.   stp  x16, x17, [sp, #16 * 8]
11.   stp  x18, x19, [sp, #16 * 9]
12.   stp  x20, x21, [sp, #16 * 10]
13.   stp  x22, x23, [sp, #16 * 11]
14.   stp  x24, x25, [sp, #16 * 12]
15.   stp  x26, x27, [sp, #16 * 13]
16.   stp  x28, x29, [sp, #16 * 14]
17.   add  x21, sp, #S_FRAME_SIZE    //S_FRAME_size 是栈帧的大小
18.   mrs  x22, elr_el1              //将 elr_el1 寄存器的数据存到 x22
19.   mrs  x23, spsr_el1             //将 spsr_el1 寄存器的数据存到 x23
```

图 3-42　压栈-状态保存

2) 触发异常原因判断

在 ARMv8 架构中,有多种原因可引起同步异常,那么操作系统如何确定引发异常的原因是用户进程请求系统调用呢? 为了使操作系统能够区分,每种引发异常的原因都应有一个唯一的编号。在 openEuler 中由 SVC 引发的同步异常被编号为 ESR_ELx_EC_SVC64 = 0x15。异常发生时,硬件自动把异常状态信息(如异常级别、被捕获指令长度等)保存在寄存器 ESR_EL1 中。通过将 ESR_EL1 中所存的状态信息与各编号比较,操作系统可以判断出当前产生的同步异常是由 SVC 引发的,并跳转至函数 el0_svc()执行,处理过程的实现如图 3-40 所示。

3) 调用相应异常处理函数及传递参数

函数 el0_svc()是系统调用异常处理的开始。操作系统并不直接在这个函数中获取系统调用号等参数,进而选择执行相应的服务函数,而是先跳转到一个 C 语言函数去做一些预备工作。例如,检查系统调用号是否合法(有没有在允许的系统调用号范围内)、是否开中断等,这些操作使用高级语言编写更直观、方便。

至此,系统调用下一步需要解决的问题是将系统调用参数和系统调用号传递给 C 语言函数 el0_svc_handler()。有关用户进程信息的结构体 pt_regs 包含了用于参数传递的寄存器 X0～X6 和 X8 中的内容,因此操作系统将其作为参数传递给函数 el0_svc_handler()。内存空间的传递利用内存首地址进行,结构体 pt_regs 所在内存的首地址保存在堆栈指针寄存器 SP 中,通过执行"mov x0, sp"指令可将堆栈指针读到寄存器 X0 中,作为参数传递给函数 el0_svc_handle()。在这种设计下,函数 el0_svc_handler()只需要定义一个形参,用来接收一个 pt_regs 结构的指针。函数 el0_svc_handle()将该指针指向的内存解释为 pe_regs 结构,进而可通过结构体变量 regs 引用栈内保存的相关寄存器内容,从而获得系统调用号

等参数。函数 el0_svc_handler()再通过所给的参数完成查找、执行指定的系统调用的逻辑。实现以上逻辑的代码如图 3-43 所示。

```
1.    //源文件: arch/arm64/kernel/entry.S
2.    el0_svc:
3.        mov x0, sp                        //将系统调用参数传递给 svc_handler
4.        bl  el0_svc_handler               //跳转执行异常处理程序
5.        b   ret_to_user
6.    ENDPROC(el0_svc)
7.    //el0_svc_handler 函数原型,形参 struct pt_regs
8.    asmlinkage void el0_svc_handler(struct pt_regs *regs);
```

图 3-43　系统调用处理

3. 系统调用服务函数的查找

异常处理过程是一致的,无论执行哪个系统调用,都会由软硬件配合来执行上述流程。在进入系统调用处理函数 el0_svc_handler()后,内核需要确定用户执行的是哪个系统调用,进而执行对应的服务函数。openEuler 内核中定义的服务函数与系统调用号一一对应。因此,在实现过程中,操作系统在用户进程执行系统调用时,通过寄存器向内核传递系统调用号和参数,并使用系统调用号来找到对应的服务函数。

内核用一个数据结构来保存系统调用号和与服务函数的对应关系,这个数据结构就是系统调用表(syscall table)。系统调用表的本质是一个全局数组,数组元素是指向处理函数的指针。内核以系统调用号作为索引,可得到相应处理函数的地址。例如,系统调用 getpid()对应的内核中的服务函数为 sys_getpid()。如图 3-44 所示,把该服务函数编号为 172,并将其函数地址保存在系统调用表中的第 172 项。

```
1.    //源文件: include/uapi/asm-generic/unistd.h
2.    #define __NR_getpid 172
3.    SYSCALL(__NR_getpid,sys_getpid)
```

图 3-44　系统调用号定义

服务函数的调用过程如图 3-45 所示:在函数 invoke_syscall()中,首先读取结构体变量 regs 中的 regs[8]获取系统调用号并将其赋值给 scno,然后使用 scno 作为系统调用表下标,得到该系统调用号所对应的服务函数的地址,最后将保存系统调用参数的结构体变量 regs 传递给该服务函数。服务函数将根据需求读取 regs[0]~regs[6]。

```
1.    //源文件: arch/arm64/kernel/syscall.c
2.    syscall_fn_t syscall_fn;
3.    scno = regs->regs[8];
4.    syscall_fn = syscall_table[scno];   //用系统调用号索引系统调用表,返回函数指针
5.    ret = syscall_fn(regs);             //代入系统调用参数,调用相应的函数
```

图 3-45　系统调用的查找

4. 服务函数的执行

在根据系统调用表找到对应的服务函数后，内核将执行该函数来为用户进程服务。对于获得进程号这个功能，就是去执行函数 sys_getpid()。因为 openEuler 内核使用宏 current 来代表指向当前进程结构体 task_struct 的指针，所以在该函数的实现过程中，服务函数 sys_getpid()可根据宏 current 来获得所求 PID，并存入寄存器 X0 中作为返回值。

5. 异常返回

得到了用户进程期望的结果，接下来要做的是异常返回(返回用户态)，并继续执行用户进程。在用户进程返回用户态时，需要做的是恢复 CPU 状态。具体而言，状态寄存器 PSTATE、程序计数器 PC 以及堆栈指针寄存器 SP_EL0 等都将被恢复。图 3-46 给出了恢复 CPU 状态的关键代码：在通过寄存器 X21、X22、X23 完成寄存器 ELR_EL1、SPSR_EL1 和 SP_EL0 数据的恢复后，将所有的通用寄存器恢复，最后将寄存器 LR 的数据恢复。简单来说，上述步骤就是将先前压入的 pt_regs 结构栈帧中的数据依次恢复。

```
1.    //源文件: arch/arm64/kernel/entry.S
2.    .macro  kernel_exit, el
3.        ...
4.        ldp x21, x22, [sp, #S_PC]     //加载保存的 ELR_EL1,SPSR_EL1 数据至寄存器
5.        ...
6.        ldr x23, [sp, #S_SP]          //加载栈指针至寄存器
7.        ...
8.        msr elr_el1, x21
9.        msr spsr_el1, x22
10.       ldp x0, x1, [sp, #16 * 0]
11.       ldp x2, x3, [sp, #16 * 1]
12.       ldp x4, x5, [sp, #16 * 2]
13.       ldp x6, x7, [sp, #16 * 3]
14.       ldp x8, x9, [sp, #16 * 4]
15.       ldp x10, x11, [sp, #16 * 5]
16.       ldp x12, x13, [sp, #16 * 6]
17.       ldp x14, x15, [sp, #16 * 7]
18.       ldp x16, x17, [sp, #16 * 8]
19.       ldp x18, x19, [sp, #16 * 9]
20.       ldp x20, x21, [sp, #16 * 10]
21.       ldp x22, x23, [sp, #16 * 11]
22.       ldp x24, x25, [sp, #16 * 12]
23.       ldp x26, x27, [sp, #16 * 13]
24.       ldp x28, x29, [sp, #16 * 14]
25.       ldr lr, [sp, #S_LR]
26.       ...
27.   eret                              //返回用户空间
28.   .endm
```

图 3-46　出栈及返回用户空间

在完成所有的数据恢复工作后,就执行指令 ERET 以返回到用户态。在返回过程中 CPU 自动使用寄存器 SPSR_EL1 保存的值来恢复状态寄存器 PSTATE,使用寄存器 ELR_EL1 保存的返回地址恢复程序计数器 PC。随后用户进程便可以从进程被挂起的地方继续运行。如果在返回用户态时发现运行队列中有比当前进程优先级更高的进程,那就发生进程切换,使当前进程插入运行队列而不是继续执行。由于 SPSR_EL1、ELR_EL1 等寄存器的内容仍保存在当前进程的内核栈中,所以下次当前进程被选中执行时,依旧可从内核栈中恢复最初被挂起时的运行环境。

3.5 进程切换

为了实现多个进程的并发执行,各进程需以时分复用的方式共享 CPU。这意味着操作系统应该支持进程切换:在一个进程占用 CPU 一段时间后,操作系统应该停止它的运行并选择下一个进程来占用 CPU。为了避免恶意进程一直占用 CPU,操作系统利用时钟中断,每隔一个时钟中断周期就中断当前进程的执行进行进程切换。本节将详细阐述进程切换的基本原理并展示进程切换的完整过程。

3.5.1 基本原理

当一个进程正在 CPU 上运行时,该进程拥有 CPU 的控制权,那么此时没有 CPU 控制权的操作系统应该如何实现进程切换?

3.4 节提到一种解决方式:进程可以通过系统调用,将 CPU 的控制转交给操作系统。在这种方式中,操作系统等待进程主动地交还 CPU 控制权。然而,如果某个进程从不进行系统调用,或是某个进程恶意地执行无限循环代码,那么操作系统将一直无法取得 CPU 控制权。

为了确保能够回收 CPU 控制权,操作系统采取一种更为强硬的方式——时钟中断。时钟模块每隔一小段时间产生一次中断;当中断发生时,当前进程的运行会被中断,并让出 CPU 控制权给操作系统预先设置的中断处理程序(Interrupt Handler);在操作系统重获对 CPU 的控制权后,就有机会执行进程切换。但是,这将面临两个关键问题:①接下来应该调度哪个进程来运行?②如果执行进程切换,操作系统如何保存当前进程的上下文,并恢复下一个运行进程的上下文?

第一个问题反映的是选择进程所用的策略,第二个问题则是针对进程切换的实现机制。操作系统的通用设计模式是,将策略与机制分开,使得策略的选择更为灵活。本章只讨论进程切换涉及的机制,策略部分将在第 4 章阐述。

图 3-47 描述的是这样一个过程:在进程 A 运行过程中,发生时钟中断,随后操作系统决定将 CPU 控制权交给进程 B,最后进程 B 开始运行。

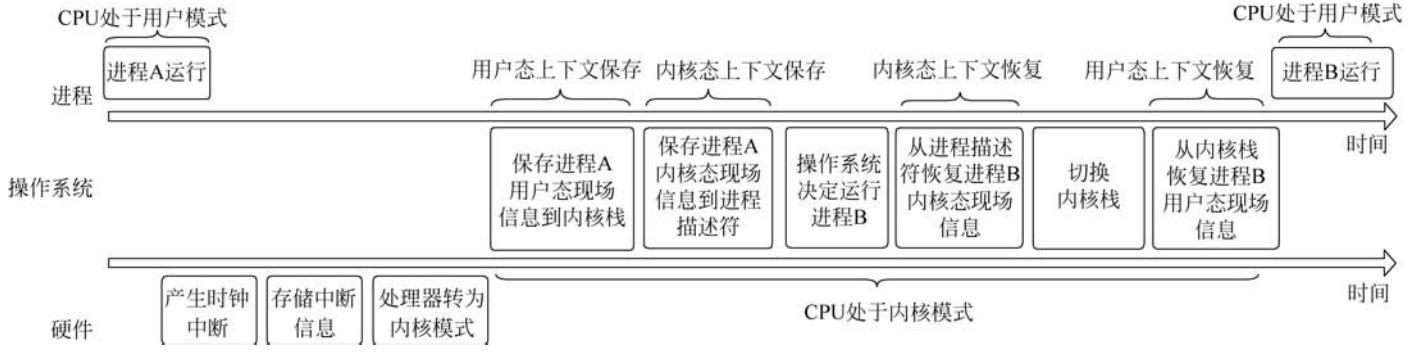
CPU处于用户模式
CPU处于用户模式
进程
进程A运行
用户态上下文保存
内核态上下文保存
内核态上下文恢复
用户态上下文恢复
进程B运行
时间
操作系统
保存进程A
用户态现场
信息到内核栈
保存进程A
内核态现场
信息到进程
描述符
操作系统
决定运行
进程B
从进程描述
符恢复进程B
内核态现场
信息
切换
内核栈
从内核栈
恢复进程B
用户态现场
信息
硬件
产生时钟
中断
存储中断
信息
处理器转为
内核模式
CPU处于内核模式
时间

图 3-47　进程切换实例

具体来说,这个过程主要包括 7 个步骤:

(1) 硬件产生时钟中断信号,触发异常,使得进程 A 由用户态陷入内核态;

(2) 在异常处理程序中,操作系统将进程 A 在用户态下的现场信息保存到内核栈中;

(3) 在异常处理程序中,操作系统选择进程 B 为即将获得 CPU 的进程;

(4) 操作系统将进程 A 在内核态下的现场信息保存到进程 A 的 PCB 中;

(5) 操作系统从进程 B 的 PCB 中恢复进程 B 在内核态下的现场信息,并切换内核栈;

(6) 操作系统从内核栈中恢复进程 B 在用户态下的现场信息;

(7) 进程 B 从内核态返回用户态。

在整个因为时钟中断而导致进程切换的过程中,需要保存和恢复的现场信息主要有两种:一是中断上下文,包括发生中断时由硬件自动保存的一些环境信息(例如当前异常级别等)以及进程在用户态下的 CPU 状态(各寄存器值),主要被保存在进程的内核栈中;二是切换上下文,又称为 CPU 上下文,即进程在内核态下的 CPU 状态,被保存在 PCB 的成员 thread 中。

3.5.2 进程切换过程

本节将根据上文叙述的进程切换过程详述进程切换过程中的关键步骤。

1. 异常处理

在 3.4 节已阐述过异常发生后 CPU 执行的操作,与此处的区别是时钟中断属于 el1_irq 类型的异常,因此内核会跳转到 el1_irq 类型的异常处理程序。该异常处理程序首先会把中断上下文保存到当前被中断进程的内核栈中,保存步骤与系统调用中的中断上下文保存步骤一致。

2. 进程调度

将中断上下文保存到内核栈后,操作系统开始进程调度,选择下一个进程来运行。调度的细节将在第 4 章讨论。

3. 切换上下文的保存与恢复

在选中下一个运行的进程后,操作系统从该选中进程的 PCB 中恢复出切换上下文。本节以进程 A 切换到进程 B 为例,阐述上下文切换的关键步骤。

1) 保存进程 A 的上下文

在 ARMv8 架构中,寄存器 TTBR0_EL1 保存着当前进程地址空间的页表首地址。由于进程 PCB 的成员变量 thread_info 已包含寄存器 TTBR0_EL1 的内容,所以进程地址空间的页表首地址不用重新保存。

进程 A 的上下文包括通用寄存器 X19～X29、堆栈指针寄存器 SP、链接寄存器 LR 和浮点寄存器 FP 中的内容。为了确保进程 A 再次运行时可以从离开的地方开始,这些寄存器内容都将保存到进程 A 的 PCB 中。首先使用指令 stp(store pair)将浮点寄存器的值依次保存到 PCB 的成员变量 thread.fpsimd_state 中,此时寄存器 X0 存放的是 PCB 中 thread.

fpsimd_state 的首地址。在图 3-48 中，指令“stp q0，q1，[x0，＃16 * 0]”表示把寄存器 q0、q1 中的内容存储到 x0＋16＊0 地址处。

```
1.    //源文件：arch/arm64/include/asm/fpsimdmacros.h
2.    stp q0, q1, [x0, #16 * 0]
3.    stp q2, q3, [x0, #16 * 2]
4.    stp q4, q5, [x0, #16 * 4]
5.    stp q6, q7, [x0, #16 * 6]
6.    stp q8, q9, [x0, #16 * 8]
7.    ...
```

图 3-48　浮点寄存器的保存

之后，内核将进程 A 的 PCB 的起始地址存放在 X0 中，PCB 中成员变量 thread.cpu_context 的相对偏移地址存放在 X10 中，再将这两个寄存器中的值相加，得到进程 A 的 PCB 中成员变量 thread.cpu_context 的首地址，存放到 X8 寄存器中。接下去内核使用指令 stp，将当前 CPU 的通用寄存器 X19～X29、堆栈指针寄存器 SP、链接寄存器 LR 的数据依次保存到进程 A 的 PCB 成员变量 thread.cpu_context 中。在图 3-49 中，指令“stp x19，x20，[x8]，＃16”表示将寄存器 X19、X20 中的内容保存到 X8 地址处，然后将 X8 加 16。

```
1.    //源文件：arch/arm64/kernel/entry.S
2.    //将 thread.cpu_context 在 PCB 中的偏移值赋值给 X10
3.    mov x10, #THREAD_CPU_CONTEXT
4.    add x8, x0, x10              //寄存器 X0 中存放着进程 A 的 PCB 首地址
5.    mov x9, sp                   //sp 的值移到 X9 寄存器中
6.    stp x19, x20, [x8], #16      //将寄存器 X19,X20 中的内容保存到 X8 地址处开始的地方,
                                   //然后将 X8 加上 16
7.    stp x21, x22, [x8], #16
8.    stp x23, x24, [x8], #16
9.    stp x25, x26, [x8], #16
10.   stp x27, x28, [x8], #16
11.   stp x29, x9, [x8], #16
12.   str lr, [x8]
```

图 3-49　通用寄存器等保存

2）恢复进程 B 的上下文

在保存了进程 A 的切换上下文后，接下来，由调度程序选择的 B 进程将会在 CPU 上运行。进程 B 必须恢复自己的切换上下文。在进程 A 的切换上下文保存后，内核立即着手将进程 B 的切换上下文恢复到对应的寄存器中。在实现时，内核依旧使用在保存进程 A 的切换上下文时使用的偏移地址，该偏移地址都是成员变量 thread.cpu_context 的相对位置，它被保存在寄存器 X10 中。这是因为无论是进程 A 还是进程 B，成员变量 thread.cpu_context

的相对位置都是一样的；而不同的地方在于,内核不再使用进程 A 的 PCB 地址,而是转而使用进程 B 的 PCB 地址,即从寄存器 X1 获得进程 B 的 PCB 地址。将进程 B 的 PCB 地址加上成员变量 thread.cpu_context 的偏移地址,内核计算出成员变量 thread.cpu_context 的绝对位置,保存到寄存器 X8 中。

接着,从进程 B 的成员变量 thread.cpu_context 中恢复出寄存器 X19～X28、X29、SP、LR、SP_EL0 的值。这样,进程 B 就可以从上次被挂起的位置继续执行。图 3-50 给出了恢复切换上下文的示例代码。其中,指令 ldp(load pair)意为加载一对寄存器。指令“ldp x19, x20, [x8], #16”表示从寄存器 X8 里面的地址加载两个 64 位数据到寄存器 X19 和 X20 中,然后把寄存器 X8 加 16。这一系列加载指令所做的事情就是将进程 B 的切换上下文从成员变量 thread.cpu_context 中恢复。

```
1.    //源文件: arch/arm64/kernel/entry.S
2.    add x8, x1, x10            //从下一个进程中的 PCB 中恢复进程 B 的寄存器数据
3.    ldp x19, x20, [x8], #16    //第 2～8 行,恢复寄存器 X9,X19～X29 的数据
4.    ldp x21, x22, [x8], #16
5.    ldp x23, x24, [x8], #16
6.    ldp x25, x26, [x8], #16
7.    ldp x27, x28, [x8], #16
8.    ldp x29, x9, [x8], #16
9.    ldr lr, [x8]
10.   mov sp, x9                 //将寄存器 X9 中的数据放入 SP
11.   msr sp_el0, x1             //使用 SP_EL0 存储 B 进程的 thread_info 地址
```

图 3-50　进程 B 切换上下文的恢复工作

在进行了切换上下文的切换后,操作系统使用的内核栈为进程 B 的内核栈。此时,内核栈中情况如图 3-51 所示。

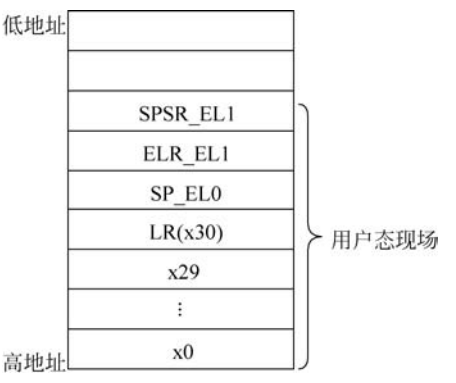

图 3-51　内核栈布局

4. 进程 B 中断上下文的恢复

中断上下文的恢复步骤与 3.4 节系统调用时返回用户态一致。在完成所有的数据恢复工作后,执行指令 ERET 返回,便可以从进程 B 被挂起的地方继续运行。在执行指令 ERET 时,CPU 自动使用寄存器 SPSE_EL1 保存的值来恢复 CPU 状态,使用寄存器 ELR_EL1 保存的返回地址恢复程序计数器。同时,内核将 thread_info.ttbr0 中保存的数据恢复到寄存器 TTBR0_EL1。这样,在回到用户态时,TTBR0_EL1 中保存的就是进程 B 的页全局目录物理地址。

3.6　线程

前面介绍了操作系统在计算机资源层面提供的并发抽象概念——进程。本节将进一步介绍操作系统在进程内部提供的并发抽象概念——线程。

3.6.1　基本概念

将运行中的应用程序抽象成进程后，在操作系统的调度下，多个进程可以并发地执行。并发执行大大提高了 CPU 的利用率。然而，在同一时间，单个进程只能处理一个任务，并不能同时处理多个任务。如果进程在执行的过程中，由于等待输入等原因被阻塞，那么整个进程将被挂起。即使在进程中有部分工作并不依赖于该输入，这些工作也无法继续执行。在很多情况下，我们希望即使进程的某部分被阻塞，但进程的其他部分还能继续执行。例如，一个音乐播放软件通常需要处理 4 种类型的工作：显示用户界面、响应用户的输入、播放音乐、将音乐保存到本地。当音乐播放软件以进程的形式运行时，用户希望该软件在等待用户输入时，播放音乐等其他功能不受影响。为此，操作系统引入线程（Thread）这一抽象概念。

线程是操作系统在进程内部提供的并发抽象概念。线程可视为进程的一个组成部分。在一个进程中，如果有一个线程由于等待输入等原因发生阻塞，那么将只有这个线程发生阻塞，其他不依赖该输入的线程可以继续运行。线程之间共享进程的地址空间等资源。在引入线程前，进程是资源分配和调度的基本单位。在引入线程后，进程的这两个属性被剥离：进程作为资源分配的基本单位，而不再作为调度的基本单位。线程则作为调度的基本单位，但不作为拥有资源的基本单位。作为调度的基本单位，线程除了提高进程的并发度，还能更高效地利用多核处理器。在多核处理器上，将一个进程拆分成多个线程后，不同的线程可以运行在不同的处理器核上，从而加速进程的执行。

在实现上，有些操作系统（如 Windows）内核提供专门的线程实现机制。还有一些操作系统，其内核未提供专门的线程实现机制（如 openEuler）。openEuler 并未为线程提供特有的数据结构，而是复用进程的数据结构 task_struct。在 openEuler 中，共享同一个进程地址空间的一组线程称为一个线程组，而进程实际上由一个线程组，以及这些线程组共享的资源组成。那么，线程与进程的主要区别是什么？

1. 是否有独立的地址空间

进程拥有独立的地址空间。一个进程发生崩溃，不会对操作系统中的其他进程产生影响。线程没有自己独立的地址空间，而是同一个线程组的所有线程共享相同的地址空间，但每个线程在共享地址空间中有自己的栈。因此，如果一个线程改乱了其他线程的栈内数据

或是触发段错误,可能导致整个进程崩溃并被操作系统终止。

以 openEuler 为例,进程和线程在地址空间中的布局如图 3-52 所示。如图 3-52(a)所示,进程 A 与进程 B 拥有独立的地址空间。在没有引入线程之前,进程在地址空间中拥有唯一的用户栈、内核栈以及切换上下文。其中,用户栈用于支持进程在用户空间中的函数调用,内核栈用于保存进程陷入内核态前的 CPU 状态,切换上下文是内核态下进程切换时保存的各寄存器数据。如图 3-52(b)所示,线程 A、B 和 C 属于同一个程序,共享同一个进程的地址空间。虽然它们共享数据段、代码段、打开的文件以及堆等,但它们的运行是彼此独立的。在不同的线程中,执行的是同一个程序的不同部分。也就是说,不同的线程有着不一样的函数调用过程。为了保存线程各自的函数调用过程,在用户空间中,为线程 A、B 和 C 分别开辟了一个用户栈。此外,在发生线程切换时,操作系统也应该保存好线程在陷入内核态前的 CPU 状态,以及在内核态下进行切换时的切换上下文,确保线程在将来能恢复被抢占之前的状态。因此,在内核空间中,为线程 A、B 和 C 各维持了一个内核栈以及一块用于保存切换上下文的区域。

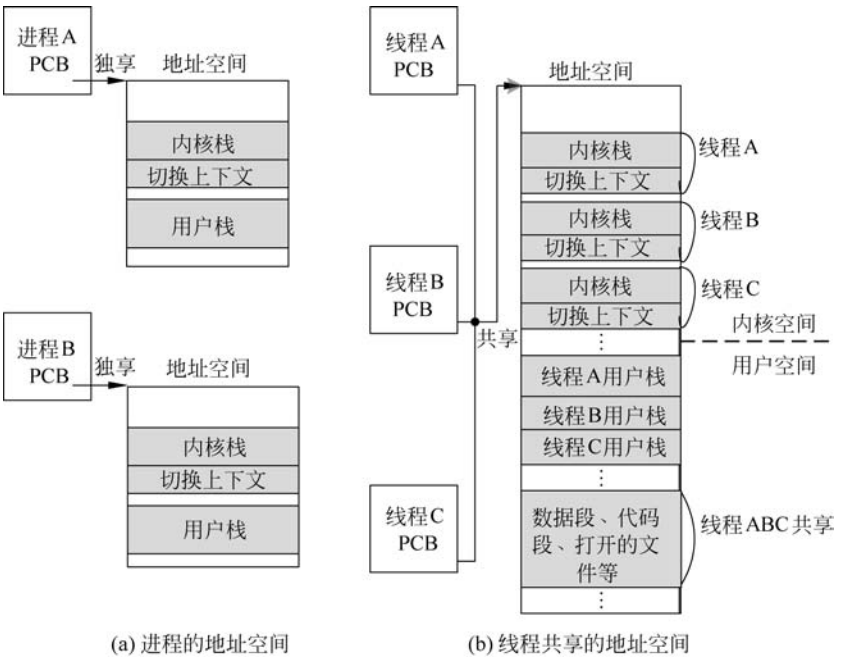

图 3-52　进程和线程在地址空间中的布局

2. 线程更为轻量级

相较于进程,线程的轻量级体现在线程创建、线程切换两个方面。每个进程都有独立的地址空间,在发生进程创建或进程切换时,涉及地址空间、文件、信号量、I/O 等资源的操作,具有较大的直接开销。而进程内的一组线程位于同一个地址空间,并共享代码段、数据段、文件等资源,所以在发生线程创建或线程切换时,最主要的是省去了地址空间的分配或切换

操作。因为地址空间的切换还可能导致 TLB(第 5 章介绍)中部分缓存失效,从而影响内存访问性能,产生间接开销。所以,线程的创建或切换开销较小。

在一些场景中,服务器可能在短时间内面临大量的服务请求。例如,一个 Web 服务器可能需要同时处理来自不同用户的上千个网页访问请求。在用户数量过多时,通过进程实现并发难以保证服务的响应时间和吞吐量。对于具有不确定用户和随机访问特性的 Web 服务器而言,用线程管理用户访问请求的系统所能支持的用户数多于用进程进行管理的系统。进程创建和切换所带来的开销也是服务器操作系统的主要瓶颈之一。由于线程相对进程而言更为轻量级,在创建和切换时的系统开销远小于进程,因此多线程是服务器操作系统处理并发请求的主要机制。

3. 通信方式

由于每个进程拥有相互隔离的地址空间,因此进程间的通信较为复杂。进程间通信通过共享内存、消息队列以及套接字等方式实现。一个进程的多个线程之间共享同一个地址空间,它们通过共享数据(如全局变量)即可实现通信。但是,多个并发线程在访问同一共享数据时,将产生竞争。因此,在多线程并发访问共享数据时,操作系统需要提供互斥与同步等特殊的通信机制。线程/进程间通信的详细内容可参考第 6 章。

3.6.2　线程模型

在多线程操作系统中,线程的实现模型可以分为三种:在用户空间实现的线程称为用户级线程;在内核空间实现的线程称为内核级线程;混合型线程是用户级线程和内核级线程的组合实现。这三种模型的主要区别在于用户空间的线程与内核调度实体的对应关系不同,如图 3-53 所示。

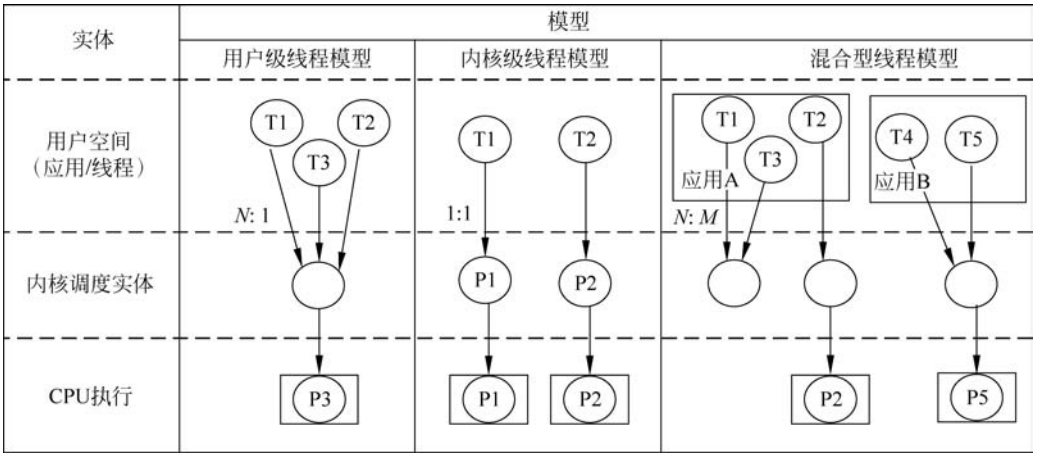

图 3-53　三种线程实现模型

1. 用户级线程

在线程的概念提出之初,操作系统内核还未提供线程支持,当时出于稳定性考虑,并未修改内核而是选择在用户空间使用线程库来实现线程,也就是用户级线程。系统开发人员将线程的创建、通信、同步及销毁等功能都封装在线程库中,无须借助系统调用来实现。用户级线程仅存在于用户空间中,其相关管理工作均由用户进程完成。内核不能感知用户级线程的存在,而是把隶属于同一个用户进程的所有用户级线程当成一个进程来实施管理。所以,用户级线程与内核调度实体是多对一的关系。这种实现模型有以下两个特点。

(1) 用户级线程的调度算法和调度过程可由用户自行决定,与操作系统内核无关。在这样的操作系统中,内核仍以进程为调度单位。用户可以决定的是,在进程被内核选中后,选择哪个线程在 CPU 上执行。

(2) 用户级线程的切换并不会导致进程的切换,而是在内核不参与的情况下完成线程上下文切换。也就是说,线程上下文切换只是在用户栈、用户寄存器等之间进行切换,不涉及 CPU 状态,带来的系统开销小。

用户级线程的弊端也包括两个方面。首先,在多核处理器中,同一个进程中的线程只能对一个 CPU 进行时分复用,即同一时刻只有一个线程可以获得 CPU,不能利用多核带来的并发优势。其次,如果一个线程被阻塞,该进程的其他线程都会被阻塞。

2. 内核级线程

与用户级线程相对应,内核级线程是由操作系统内核进行管理的。内核向用户进程提供相应的系统调用,以供用户进程创建、执行、撤销线程。在这类系统中,用户进程中的线程与内核调度实体是一对一的关系,例如,openEuler 借助 NPTL(Native POSIX Thread Library,POSIX 标准线程库)实现这种对应关系。内核级线程就是系统调度的最小单位,既可以被调度到一个 CPU 上并发执行,也可以被调度到不同 CPU 上并行处理。若是一个线程被阻塞,操作系统可以调度该用户进程的其他线程去执行,而不至于阻塞整个用户进程。

虽然内核级线程似乎解决了用户级线程的缺点,但是内核级线程的管理与调度需要由内核完成。这意味着,每次线程切换都需要陷入内核态,陷入过程会带来不小的开销,所以内核级线程的切换代价要更大。此外,内核需要维护一份线程表去管理内核级线程。由于内核资源有限,能维持的线程数量也有限,因此其扩展性不如用户级线程。

3. 混合型线程

上述两种线程的实现模型都有各自缺点,有些操作系统(如 Solaris 操作系统)采用用户级线程和内核级线程的组合的方式实现线程管理,尽可能利用各自的优点而规避缺点。在这些操作系统中,线程的创建、同步等仍在用户空间完成,并且 N 个用户级线程可以被映射到 M 个内核级线程上($N \geqslant M$),这是多对多的关系。这种实现模型下,调度可以分为两级,先由内核决定获得 CPU 的内核级线程,然后由用户调度器从映射到该内核级线程的多个用户级线程中选择一个执行。

表 3-1 简要总结了这三种线程实现模型的优缺点对比。

表 3-1　三种线程实现模型的优缺点对比

线程实现模型	优　　点	缺　　点
用户级线程模型	(1) 用户自行决定调度算法 (2) 线程切换在用户态,开销小	(1) 用户进程的多个线程不能并行执行 (2) 用户进程因某个线程阻塞而阻塞
内核级线程模型	(1) 线程可在不同的 CPU 上并行处理 (2) 某线程阻塞,其他线程可继续执行	(1) 线程创建、切换需陷入内核态,开销大 (2) 占用内核资源
混合型线程模型	上述优点都具备	高度复杂,实现困难

3.6.3　openEuler 中线程的实现

openEuler 采用的是上述三种线程实现模型中的内核级线程模型,其面向用户提供的线程库是 NPTL。下面先介绍用户调用 NPTL 中的 API 函数,完成调用 openEuler 系统调用接口,进而实现对线程的控制这一过程。随后,以线程创建与线程切换为例,结合 openEuler 中的代码阐述线程与进程在创建和切换时的主要区别,以突出线程轻量级的特点。

1. 线程的生命周期

在 openEuler 中,线程的生命周期主要包括图 3-54 展示的 5 种状态。用户通过调用函数 pthread_create()创建一个线程。在创建完成后,该线程处于就绪状态(转换①)。当该线程被操作系统调度执行,将发生转换②,进入运行状态。之后,若因 CPU 被抢占或主动让出 CPU,线程将发生转换③,回到就绪状态。在运行状态下,线程还可能因为调用函数 pthread_join()(需要等待子线程返回)、sleep()或是 I/O 操作而发生转换④,进入阻塞状态。当导致阻塞的条件得到满足时,将发生转换⑤,回到就绪状态。最后,线程因计算任务结束或是因异常终止,将隐式地退出。另外,用户也可以调用函数 pthread_exit()让线程显式地退出并获得一个返回值。线程由运行到退出将发生转换⑥。

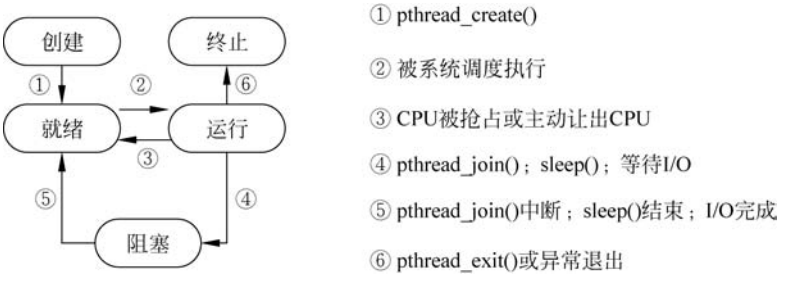

图 3-54　线程的生命周期

NPTL 中的 API 函数最终会调用内核提供给用户空间的系统调用接口,进而借助内核中的原语,完成线程控制。由于 openEuler 并没有为线程定义原语,而是使用进程原语对其控制,所以 NPTL 中的函数最终对应的是进程原语,其对应关系如表 3-2 所示。例如,函数

pthread_create()将通过原语 clone()在内核中创建一个内核级线程,而函数 pthread_exit()将通过原语 exit()终止线程。

表 3-2　线程控制接口与进程原语的对应关系

基 本 控 制	线程库 API 函数	进 程 原 语
创建	pthread_create()	fork()/clone()
终止	pthread_exit()	exit()
等待回收	pthread_join()	wait()/waitpid()
获取 ID	pthread_self()	getpid()

2. 线程创建

下面以 API 函数 pthread_create()为例,介绍其调用原语 clone()完成内核级线程创建的完整过程。pthread_create()原型如图 3-55 所示。其中,参数 thread 用于指定线程号;参数 attr 用于指定线程属性;参数 start_routine 传入的是新线程创建后要指向的函数(回调函数);参数 arg 用于指定回调函数的参数。

```
1.    int pthread_create(pthread_t *thread, const pthread_attr_t *attr,
2.                       void *(*start_routine) (void *), void *arg);
```

图 3-55　API 函数 pthread_create()

函数 pthread_create()的执行流程如图 3-56 所示。函数 pthread_create()首先配置线程的用户空间环境,包括线程属性、用户栈空间及线程描述符等信息。这些信息将用于帮助用户进行线程控制。接着,函数 pthread_create()调用函数 create_thread(),进而调用系统调用接口 do_clone()去请求内核创建一个内核级线程。内核在接收到线程创建请求后,将调用内核函数 sys_clone(),最终调用函数_do_fork()完成内核级线程的创建。如图 3-56 左侧所示,进程创建原语 fork()也需调用函数_do_fork()实现。也就是说,线程与进程创建的步骤大致相同,仅在资源复制时有所差别,此处不再赘述。下文将重点阐述两者在资源复制时的区别,以突出线程轻量级的特点。

进程与线程的创建都是对一个现有进程内容的复制或引用。函数 copy_process()用于实现资源复制。图 3-57 示例程序第 3～9 行展示了需要复制的主要资源,包括打开的文件列表、文件系统相关信息(第 7 章介绍)、信号处理相关资源(第 6 章介绍)、内存描述符以及 I/O 资源等。图 3-57 示例程序第 13～18 行以函数 copy_files()为例(其他复制函数实现大致相似),展示了进程与线程的资源复制过程差异。进程是对父进程资源进行复制(第 19 行),其拥有父进程大部分资源实体的一个副本,所以复制时间成本高;而线程对进程大部分资源只是引用,它共享着进程的多数资源,只需要将进程结构体 task_struct 中的对应资源项引用计数加一即可(第 16 行),并不实际复制资源实体,是轻量级的。因此,线程的创建速度要快于进程的创建速度。

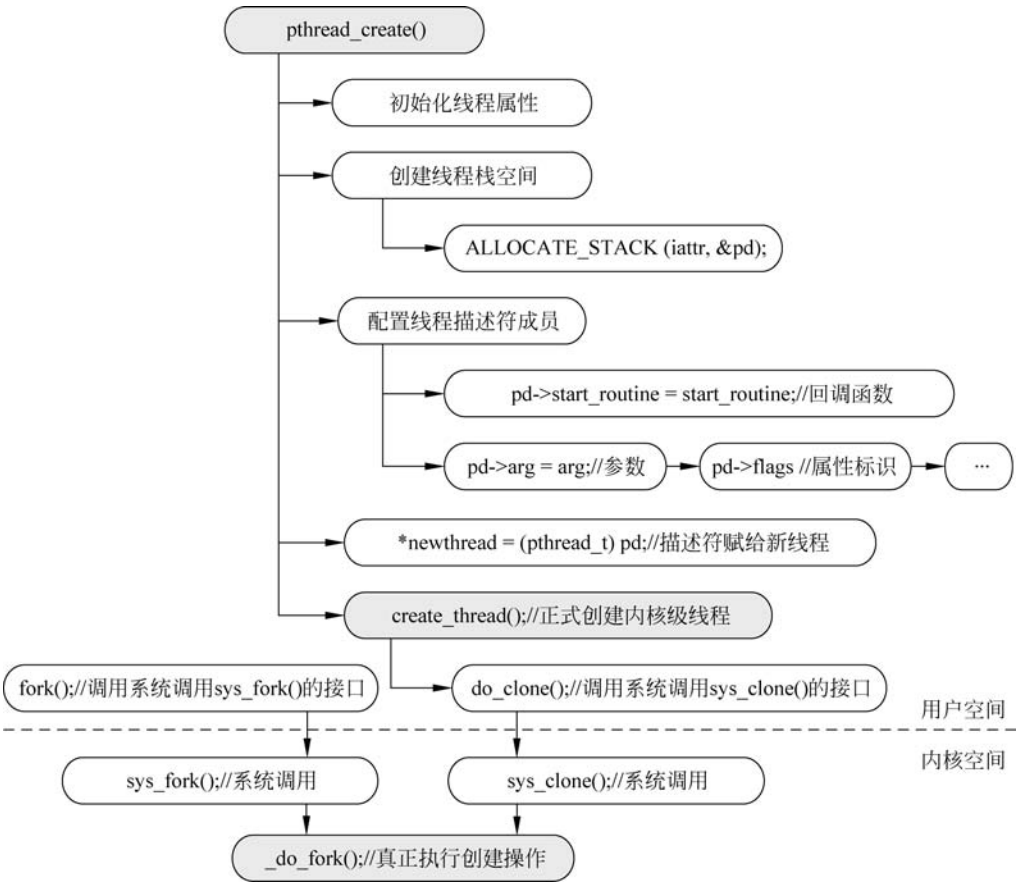

图 3-56　函数 pthread_create()的创建流程

```
1.    //源文件：kernel/fork.c
2.    //函数 copy_process()
3.    retval = copy_files(clone_flags, p);         //复制打开的文件列表
4.    retval = copy_fs(clone_flags, p);            //复制相关联文件系统信息
5.    retval = copy_sighand(clone_flags, p);       //复制信号处理函数
6.    retval = copy_signal(clone_flags, p);        //复制信号
7.    retval = copy_mm(clone_flags, p);            //复制内存描述符
8.    retval = copy_namespaces(clone_flags, p);    //复制命名空间
9.    retval = copy_io(clone_flags, p);            //复制 I/O 资源
10.   …
11.   //源文件：kernel/fork.c
12.   //以函数 copy_files()为例,介绍创建进程与创建线程的资源复制差异
13.   static int copy_files(unsigned long clone_flags,
```

图 3-57　创建进程与创建线程的资源复制差异

```
14.                                 struct task_struct * tsk) {
15.         if (clone_flags & CLONE_FILES) {          //创建线程时参数设置了 CLONE_FILES
16.             atomic_inc(&oldf->count);             //只需要将打开文件的引用计数加一
17.             goto out;
18.         }
19.         newf = dup_fd(oldf, &error);              //创建进程则需要完全复制打开的文件列表
20.         tsk->files = newf;                        //记录在 PCB 中
21.         ...
22.     }
```

图 3-57 (续)

3. 线程切换

进程切换有三个主要步骤:地址空间切换、内核栈切换和上下文切换。不同线程组内的线程切换过程与进程切换过程是相同的,而同一个线程组内的线程因为共享地址空间则省去了地址空间切换操作(见图 3-58 中代码第 5～6 行)。地址空间切换本身具有直接开销,同时还带来了间接开销。当地址空间发生变化时,CPU 的 TLB 等缓存机制也可能随之被刷新(第 17 行),之后的内存访问将耗时更长。相较于进程的地址空间切换开销与复杂的 TLB 刷新管理而言,在更多情况下,线程切换速度更快。

```
1.      //源文件: include/asm/mmu_context.h
2.      static inline void switch_mm(struct mm_struct * prev,
3.          struct mm_struct * next, struct task_struct * tsk) {
4.          //线程切换不需要进入函数__switch_mm()
5.          if (prev != next)          //prev 为要切出的地址空间,next 为要切入的地址空间
6.              __switch_mm(next);     //只有 prev 和 next 不等时才真正进行地址空间切换
7.          ...
8.      }
9.      //源文件: arch/arm64/mm/context.c
10.     //进程切换时地址空间切换通过函数 check_and_switch_context()
11.     void check_and_switch_context(struct mm_struct * mm,unsigned int cpu) {
12.         ...
13.         cpu_switch_mm(mm->pgd, mm);      //真正进行 MMU 页表切换
14.         ...
15.         //如果满足 TLB 刷新条件,就要将所有 local TLB entries 刷新
16.         if (cpumask_test_and_clear_cpu(cpu, &tlb_flush_pending))
17.             local_flush_tlb_all();
18.         ...
19.     }
```

图 3-58 进程与线程在切换时的主要开销差异

本章小结

由于程序只是对计算任务和数据的静态描述，所以，为了刻画程序并发执行带来的动态特征，操作系统引入了进程的概念。进程是操作系统中最重要、最基本的概念之一，它是系统分配资源的基本单位，是一个具有独立功能的程序段对某个数据集的一次执行活动。

进程是一个动态的概念。反映进程动态特性的是进程状态的变化。进程要经历创建、等待资源、就绪准备执行，以及执行和执行后释放资源终止等几个过程和状态。进程的状态转换要由不同的原语执行完成。本章结合 openEuler 源码，对进程创建、程序装载以及进程终止的相关原语进行了详细介绍。

操作系统借助进程对并发执行的程序加以描述和控制。进程的并发特性反映在执行的间断性和资源共享带来的制约性上。多个进程在 CPU 上来回切换，虽然它们是间断地执行，但是却让用户以为这些进程是并发且连续运行的。并发执行的多个进程共享系统中的 CPU、内存及 I/O 等资源，它们对共享资源的使用存在制约。一方面，操作系统需要保证每个进程正常使用资源；另一方面，操作系统也要确保进程不影响其他进程使用资源。进程并发执行时涉及进程切换，进程切换是一个在用户态无法完成的受限操作。进程需要通过系统调用或是被中断才能陷入内核态，借助内核完成进程切换。本章也阐述了 openEuler 的系统调用以及进程切换两个过程所涉及的细节。

尽管进程是一个动态概念，但是从处理机执行的观点来看，进程仍需要静态描述。一个进程的静态描述是处理机的一个执行环境，被称为进程上下文。进程上下文由以下部分组成：PCB、代码段和数据段以及各种寄存器和堆栈中的值。寄存器中主要存放将要执行指令的逻辑地址、执行模式以及执行指令时所要用到的各种调用和返回参数等。而堆栈中则存放 CPU 现场保护信息、各种资源控制管理信息等。

为了满足应用中多任务并行处理的要求，并且尽量减少应用因等待 I/O 而整个陷入阻塞的情况，操作系统引入了线程的概念。线程由寄存器、堆栈以及程序计数器等组成，同一进程的线程共享该进程的地址空间和其他所有资源。可以说，进程是资源管理的基本单位，而线程才是基本调度单位。线程可分为用户级线程、内核级线程以及混合型线程。其中，用户级线程的管理全部由线程库完成，与操作系统内核无关。线程主要用于多处理器系统中。

进程与线程的管理与 CPU 架构是强相关的。随着 CPU 架构的不断发展，CPU 核数逐渐增加，多处理器系统能支持的进程、线程数也会随之增加。降低进程与线程的创建以及切换开销将依旧是进程与线程未来研究的重要方向。进程、线程数量的增多还会导致对内存资源、外设资源竞争的加剧，同样会给缓存带来巨大的压力，从而影响系统性能。另外，随着 CPU 速度的不断增加，与磁盘读写速度间的差距将越来越大，这使得更多进程出现频繁等待磁盘 I/O 的情况，针对这样的场景如何合理地管理进程也是一个重要的研究方向。