第 5 章　进 程 管 理

在操作系统中,进程是一个极其重要的概念。本章首先会通过比较进程和程序的区别来说明进程的基本概念,然后详细讲解在 EOS 中创建进程的过程。在现代操作系统中,线程已经获得了广泛的应用。所以,在 EOS 中也同样引入了线程的概念。本章会结合线程来重点讲解状态转换、同步以及调度等问题。

5.1　进程的描述与控制

操作系统中最核心的概念就是进程,其他所有的内容都围绕着进程。所以读者应该尽早地理解进程的概念。对于 EOS 操作系统来说,进程就是资源分配的单位。

5.1.1　进程和程序

程序可以被理解为是一组有序指令和数据的集合,通常以文件的形式被存放在某种介质(例如磁盘)上。进程是具有独立功能的程序关于某个数据集合上的一次运行活动,是系统进行资源分配的单位。程序和进程是两个完全不同的概念,但这两个概念又是紧密联系不可分割的。可以从以下几个方面来认识这两个概念:

- 进程的状态会随着程序指令的执行而不断地发生变化,可以认为进程是动态的。而程序则没有运动的概念,可以认为是静态的。
- 进程是暂时的,有一定的生命周期。而程序则可以长久保存。
- 进程不但包含了程序中的指令和数据,同时还会包含操作系统内核的部分指令和数据,如图 5-1 所示。
- 一个程序通过多次执行,可以产生多个进程。一个进程通过调用不同的程序,可以包括多个程序,如图 5-2 所示。

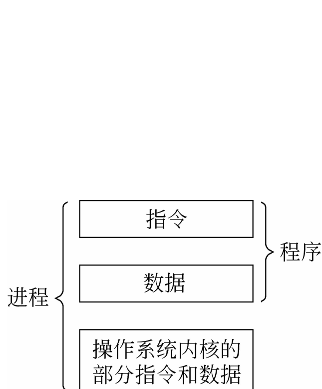

图 5-1　进程和程序的组成

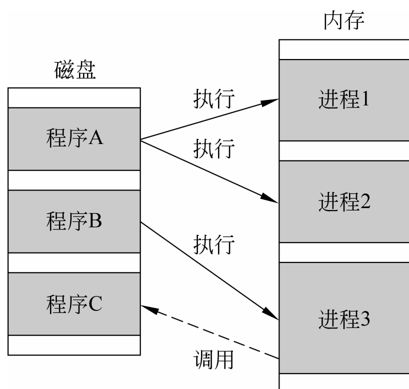

图 5-2　进程和程序的对应关系

5.1.2 进程控制块(PCB)

操作系统准备了一个专门的数据结构用来管理进程,这个数据结构通常被称为进程控制块(PCB:Process Control Block)。在 PCB 中记录了进程的各种信息,操作系统正是使用这些信息对进程进行管理的。所以,每当操作系统创建了一个新进程时,就会为其建立一个 PCB。在 ps/psp.h 中定义了 EOS 操作系统所使用的 PCB 结构体:

```
typedef struct _PROCESS {
    BOOLEAN System;                      //是否系统进程
    UCHAR Priority;                      //进程的优先级
    PMMPAS Pas;                          //进程地址空间
    PHANDLE_TABLE ObjectTable;           //进程的内核对象句柄表
    LIST_ENTRY ThreadListHead;           //线程链表头
    PTHREAD PrimaryThread;               //主线程指针
    LIST_ENTRY WaitListHead;             //等待队列,等待进程结束的线程在此队列等待。

    PSTR ImageName;                      //二进制映像文件名称
    PSTR CmdLine;                        //命令行参数
    PVOID ImageBase;                     //可执行映像的加载基址
    PPROCESS_START_ROUTINE ImageEntry;   //可执行映像的入口地址

    HANDLE StdInput;                     //标准输入句柄
    HANDLE StdOutput;                    //标准输出句柄
    HANDLE StdError;                     //标准错误输出句柄

    ULONG ExitCode;                      //进程退出码
} PROCESS;
```

可以看到,在 PCB 结构体中主要包括了进程控制信息(例如优先级 Priority、退出码 ExitCode 等),以及进程所拥有的资源(例如地址空间 Pas、内核对象句柄表 ObjectTable 等)。

一般来说,PCB 中还应该包括用来保存 CPU 现场的结构,这样才能使多个进程并发执行。但是由于 EOS 中使用了线程概念,将线程作为 CPU 执行和调度的单位,而进程仅仅作为资源的容器。所以,用来保存 CPU 现场的结构就被定义在线程控制块(Thread Control Block,TCB)中了。在 ps/psp.h 中可以找到 TCB 结构体的定义:

```
typedef struct _THREAD {
    ⋮
    CONTEXT KernelContext;
    ⋮
} THREAD;
```

CONTEXT 结构体中的域用来保存线程的 CPU 现场。在 inc/ke.h 文件中可以找到 CONTEXT 结构体的定义:

```
typedef struct _CONTEXT
{
    ULONG Eax;
    ULONG Ecx;
    ULONG Edx;
    ULONG Ebx;
    ULONG Esp;
    ULONG Ebp;
    ULONG Esi;
    ULONG Edi;
    ULONG Eip;
    ULONG EFlag;
    ULONG SegCs;
    ULONG SegSs;
    ULONG SegDs;
    ULONG SegEs;
    ULONG SegFs;
    ULONG SegGs;
}CONTEXT, * PCONTEXT;
```

可以看到 CPU 现场主要包含了 CPU 中各个寄存器的值。当一个线程被中断执行时，CPU 现场被保存在该线程的 TCB 中，当该线程可以继续执行时，TCB 中保存的现场被恢复到 CPU 中，该线程就可以从中断处继续执行了。

5.1.3 进程的创建

当 EOS 创建一个进程时，会首先创建一个进程对象，并且进程对象的对象体使用的就是 PCB 结构体（关于对象体的概念可参见 4.2 节）。接下来，操作系统会为进程分配一个进程地址空间和一个句柄表（进程地址空间的概念将会在第 6 章中介绍，而句柄表的概念可以参见 4.6 节）。一般情况下，每个进程都是由一个可执行文件（后缀名为 EXE 的文件）来创建的，所以，操作系统会将可执行文件装入进程地址空间的用户地址空间中，并和内核地址空间中的内核进行动态链接（Dynamic Link）。最后，操作系统会为进程创建一个主线程，并使主线程从进程可执行文件的入口点开始执行。关于进程创建的详细过程，请参看 ps/create.c 源文件中的 PsCreateProcess 函数。

1. CreateProcess 函数介绍

EOS 提供了一个用于创建进程的 API 函数 CreateProcess，EOS 应用程序可以调用此函数为一个可执行文件（应用程序）创建进程。CreateProcess 函数定义如下。

```
BOOL CreateProcess(
    IN PCSTR ImageName,
    IN PCSTR CmdLine,
    IN ULONG CreateFlags,
    IN PSTARTUPINFO StartupInfo,
```

```
    OUT PPROCESS_INFORMATION ProcInfo
    )
```

参数 ImageName 用来指定应用程序的可执行文件的路径和名称。EOS 会使用指定的可执行文件创建一个进程。此参数不能为空指针(NULL)。例如,如果要使用软盘根目录下的可执行文件 Hello. exe 创建进程,可以将此参数设置为“A:\\Hello. exe”(注意在字符串常量中的反斜线要使用转义字符来表示)。参数 CmdLine 是应用程序的命令行参数,如果应用程序不需要任何参数,可以设置此参数为空指针(NULL)。参数 CreateFlags 目前没有用到,设置为 0 即可。

参数 StartupInfo 是一个 STARTUPINFO 结构体变量的指针。STARTUPINFO 结构体在 inc/eosdef. h 中定义如下:

```
typedef struct _STARTUPINFO {
    HANDLE StdInput;
    HANDLE StdOutput;
    HANDLE StdError;
} STARTUPINFO, * PSTARTUPINFO;
```

在此结构体中定义了子进程需要用到的标准句柄(标准输入、标准输出、标准错误)。在调用 CreateProcess 函数之前,应该首先定义一个 STARTUPINFO 结构体的变量,在正确初始化此变量的各个成员后,才能将此变量的指针作为参数传入 CreateProcess 函数。例如在调用 CreateProcess 函数之前,先调用三次 GetStdHandle 函数(分别使用参数 STD_INPUT_HANDLE、STD_OUTPUT_HANDLE 和 STD_ERROR_HANDLE)来得到父进程拥有的标准句柄,然后将这些句柄分别赋值给 STARTUPINFO 结构体变量对应的成员,这样,子进程和父进程就可以使用相同的标准句柄了(也就是说如果父进程的标准输出是显示器,子进程的标准输出也是显示器;如果父进程的标准输入是控制台,子进程的标准输入也是控制台)。

参数 ProcInfo 是一个 PPROCESS_INFORMATION 结构体变量的指针,用来返回子进程的信息。PPROCESS_INFORMATION 结构体在 eosdef. h 文件中定义如下:

```
typedef struct _PROCESS_INFORMATION {
    HANDLE ProcessHandle;
    HANDLE ThreadHandle;
    ULONG ProcessId;
    ULONG ThreadId;
} PROCESS_INFORMATION, * PPROCESS_INFORMATION;
```

在调用 CreateProcess 函数之前,应该首先定义一个 PPROCESS_INFORMATION 结构体的变量,然后将此变量的指针作为参数传入 CreateProcess 函数,用来返回子进程的信息。子进程创建成功后,父进程可以调用 WaitForSingleObject 函数,并将 PPROCESS_INFORMATION 结构体变量的 ProcessHandle 成员(子进程句柄)作为第一个参数,将 INFINITE 作为第二个参数。这样 WaitForSingleObject 函数将一直等待,直到子进程执行

完毕后才会返回。待 WaitForSingleObject 函数返回后，可以调用 GetExitCodeProcess 函数来得到子进程的退出码，从而判断子进程执行的结果。最后，如果不再使用子进程的句柄，应该调用函数 CloseHandle 关闭由此参数返回的子进程句柄和子进程的主线程句柄。

如果创建子进程成功，CreateProcess 函数会返回 TRUE。如果创建失败，返回 FALSE，此时可以调用 GetLastError 函数来得到错误码，然后根据错误码判断导致错误的原因。在 inc/error.h 文件中列出了 EOS 能够返回的所有错误码。下列源代码片断演示了一个典型的创建子进程的过程：

```
STARTUPINFO StartupInfo;
PROCESS_INFORMATION ProcInfo;
ULONG ulExitCode;

StartupInfo.StdInput=GetStdHandle(STD_INPUT_HANDLE);
StartupInfo.StdOutput=GetStdHandle(STD_OUTPUT_HANDLE);
StartupInfo.StdError=GetStdHandle(STD_ERROR_HANDLE);

if(CreateProcess("A:\\Hello.exe", NULL, 0, &StartupInfo, &ProcInfo)) {
    WaitForSingleObject(ProcInfo.ProcessHandle, INFINITE);
    GetExitCodeProcess(ProcInfo.ProcessHandle, &ulExitCode);
    printf("\nThe process exit with %d.\n", ulExitCode);

    CloseHandle(ProcInfo.ProcessHandle);
    CloseHandle(ProcInfo.ThreadHandle);
} else {
    printf("CreateProcess Failed, Error code: 0x%X.\n", GetLastError());
}
```

严格来说，EOS 并不保存进程的父子关系。虽然当父进程创建子进程时，父进程会得到子进程及其主线程的对象句柄。父进程也可以使用这些句柄对子进程或其主线程进行控制，例如调用 WaitForSingleObject 函数等待子进程结束、调用 TerminateProcess 函数强制结束子进程或者调用 GetExitCodeProcess 得到子进程的结束码。但是，父进程也可以什么都不做，只是关闭这些句柄。关闭这些句柄不会对子进程的运行产生任何影响，仅仅是使父进程失去了对子进程的控制权。

2. 其他函数的介绍

在之前提到的其他 API 函数如下所示：

```
HANDLE GetStdHandle(
    ULONG StdHandle
    )
```

功能：得到调用此函数的进程所使用的标准句柄。

参数：StdHandle 是进程标准句柄的索引。如果此参数设置为 STD_INPUT_HANDLE，返回进程的标准输入句柄，设置为 STD_OUTPUT_HANDLE 返回进程的标准输出句柄，设置为 STD_ERROR_HANDLE 返回进程的标准错误句柄。

返回值：返回调用此函数的进程的标准句柄。如果失败返回 NULL，此时可以调用 GetLastError 函数得到错误码。

```
ULONG WaitForSingleObject(
    IN HANDLE Handle,
    IN ULONG Milliseconds
    )
```

功能：等待直到指定的句柄进入有信号(Signaled)状态或者超时后才返回。

参数：Handle 指定要等待的句柄。例如此参数可以是一个进程句柄，当进程正在执行时，进程句柄是无信号的(Nonsignaled)，此函数不会返回，当进程结束后，进程句柄变为有信号(Signaled)，此函数就会返回。Milliseconds 指定超时时间，单位为毫秒，INFINITE 表示永远不会超时。

返回值：如果句柄变为有信号，返回 0。如果句柄无信号并且超时，返回 WAIT_TIMEOUT。如果返回-1 说明执行失败，此时可以调用 GetLastError 函数得到错误码。

```
BOOL GetExitCodeProcess(
    IN HANDLE ProcessHandle,
    OUT PULONG ExitCode
    )
```

功能：在进程执行完毕后，得到进程的退出码。

参数：ProcessHandle，指定要得到其退出码的进程句柄。ExitCode，ULONG 变量指针，输出进程的退出码。

返回值：如果成功得到进程的退出码，返回 TRUE。否则返回 FALSE，此时可以调用 GetLastError 函数得到错误码。

```
BOOL CloseHandle(
    HANDLE Handle
    )
```

功能：关闭指定的句柄。

参数：Handle 指定要关闭的句柄。例如进程句柄、线程句柄。

返回值：如果成功关闭指定的句柄，返回 TRUE。否则返回 FALSE，此时可以调用 GetLastError 函数得到错误码。

```
ULONG GetLastError(
    VOID
    )
```

功能：得到调用此函数的线程保存的最后的错误码。

参数：无

返回值：返回错误码。

5.1.4 进程的终止

EOS 中的进程终止运行可能由如下原因引起：

- 主线程终止运行。当进程的主线程终止运行时，进程中其他所有正在运行的线程都将被系统强制终止，从而使整个进程终止运行。操作系统会将主线程的退出码设置为进程的退出码。关于线程终止运行的原因，稍后会有介绍。
- 进程中的任意线程调用了 API 函数 ExitProcess。
- 其他进程调用了 API 函数 TerminateProcess 来结束本进程。

进程终止运行，最终是通过执行 ps/delete.c 源文件中的 PspTerminateProcess 函数来完成的，大致过程如下：

（1）设置进程的结束标志。PROCESS 结构体中的 PrimaryThread 指针赋值为 NULL 标志进程已经结束。

（2）唤醒所有正在阻塞等待进程结束的线程。

（3）遍历进程的线程链表，结束进程的所有线程。

（4）释放进程的句柄表。在释放句柄表时，句柄表内所有尚未关闭的句柄都将被关闭。

（5）释放进程地址空间。在释放进程地址空间时，会对地址空间进行清理，释放所有虚拟内存。

（6）释放所有对进程对象的引用。

注意：由于 EOS 仅仅是一个教学操作系统，为了保持结构简单，并没有过多考虑其健壮性。所以，不推荐使用 TerminateProcess 函数来强制结束一个进程，这可能会引起资源泄漏或者死锁。

5.2 线程的描述与控制

在 EOS 中，线程是处理器调度的基本单位。当一个进程被创建时，系统首先会为该进程分配一些资源（包括内存，内核对象，以及指令和数据等），然后系统会为该进程创建一个默认线程，作为该进程的主线程。进程的主线程开始执行后，就可以认为是进程开始执行了。多数情况下，进程只需要在主线程运行的过程中就可以完成工作。但是，随着单个处理器中内核数量的增加，越来越多的软件要求使用多线程进行并行处理，从而提高硬件资源的利用率，以及软件执行的效率。EOS 支持多线程并发执行，除了在前一节提到的多个进程（每个进程都有一个主线程）并发执行的情况外，还可以在一个进程中创建多个线程。例如，在一个进程的主线程中，可以调用 API 函数 CreateThread 来创建一个新线程，这个新线程与主线程共享该进程的所有资源，例如访问进程的地址空间、执行进程的代码、读写进程打开的文件等。而且，EOS 中的线程是属于内核级的，所有线程会一起竞争处理机的使用权，不会区分线程属于哪个进程。本节的主要内容就是向读者详细讲解以上的各种概念，帮助读者理解线程的本质。

5.2.1 线程控制块（TCB）

操作系统使用进程控制块（PCB）来管理进程，同样的，操作系统还使用线程控制块（TCB：Thread Control Block）来管理线程。EOS 中的 TCB 由 THREAD 结构体来描述，其在 ps/psp.h 中定义如下所示：

```
typedef struct _THREAD {
    PPROCESS Process;                   //线程所属进程指针
    LIST_ENTRY ThreadListEntry;         //进程的线程链表项
    UCHAR Priority;                     //线程优先级
    UCHAR State;                        //线程当前状态
    ULONG RemainderTicks;               //剩余时间片,用于时间片轮转调度
    STATUS WaitStatus;                  //阻塞等待的结果状态
    KTIMER WaitTimer;                   //用于有限等待唤醒的计时器
    LIST_ENTRY StateListEntry;          //所在状态队列的链表项
    LIST_ENTRY WaitListHead;            //等待队列,所有等待线程结束的线程都在此队列等待

    PVOID KernelStack;                  //线程位于内核空间的栈
    CONTEXT KernelContext;              //线程执行在内核状态的上下文环境状态

    PMMPAS AttachedPas;                 //线程在执行内核代码时绑定进程地址空间

    PTHREAD_START_ROUTINE StartAddr;            //线程的入口函数地址
    PVOID Parameter;                            //传递给入口函数的参数

    ULONG LastError;                            //线程最近一次的错误码
    ULONG ExitCode;                             //线程的退出码
} THREAD;
```

THREAD 结构体中的域比较多,这里先结合已有的知识简单介绍几个域,其他域会在后面的内容中一一介绍。首先,所有的线程在某一时刻都必须依附于某一个进程,TCB 的 Process 域就指向了线程所依附的进程。其次,每个线程都有一个自己的栈(用来保存函数返回地址、函数参数和局部变量),TCB 的 KernelStack 域指向栈在内存中的位置。在 KernelContext 域中会保留线程被中断执行时的处理器的上下文(就是处理器各个寄存器的值)。通过 KernelStack 和 KernelContext 这两个域的合作,就可以完全记录下线程被中断执行时的状态,当线程恢复执行时,就可以从之前的状态继续执行了。

5.2.2 线程的创建和终止

在 EOS 内核中,线程的创建最终都是通过调用 PspCreateThread 函数(在文件 ps/create.c 中定义)完成的。该函数的流程比较简单,首先是创建一个空白的线程控制块,然后为线程分配栈,并初始化线程的上下文环境,最后使线程进入就绪状态。更加详细的过程请读者阅读该函数的源代码。

在 EOS 应用程序中可以调用 API 函数 CreateThread 来创建一个新线程。该 API 函数的定义如下所示:

```
HANDLE CreateThread(
    IN SIZE_T StackSize,
    IN PTHREAD_START_ROUTINE StartAddr,
    IN PVOID ThreadParam,
```

```
    IN ULONG CreateFlags,
    OUT PULONG ThreadId
);
```

如果该函数成功的创建了一个新线程，就返回线程对象的句柄，否则就会返回 NULL。该函数各个参数的意义如下所示：

参　　数	描　　述
StackSize	线程栈的大小。目前栈的大小总是使用默认值，输入 0 即可
StartAddr	线程入口函数的指针
ThreadParam	传递给线程函数的参数
CreateFlags	创建参数，目前尚无参数可选，输入 0 即可
ThreadId	指向用于保存线程 ID 变量的指针，如果为 NULL 就不获取 ID

线程入口函数的类型在文件 inc/eosdef.h 中定义如下：

```
typedef ULONG(* PTHREAD_START_ROUTINE)(PVOID ThreadParameter);
```

在调用 CreateThread 函数创建线程之前，要首先按照线程入口函数类型的定义，编写一个线程入口函数。需要强调的一点是，当线程创建成功后，CreateThread 函数会立即返回，也就是说创建者线程和被创建线程是异步执行的。

当线程入口函数返回时，新建的线程就结束执行了。此时，线程对象会由 nonsignaled 状态变为 signaled 状态。注意，EOS 应用程序启动执行时会创建一个主线程，在主线程执行的过程中，使用 CreateThread 函数创建的线程可以认为是子线程。当主线程结束后，进程就会结束，此时，所有的子线程无论是否执行完毕，都会被强制结束执行。所以，一般情况下，主线程应该等待所有的子线程执行完毕后再结束执行。

5.2.3　线程的状态和转换

1. 状态

线程在其整个生命周期中(从创建到终止)会在多个不同的状态间进行转换。EOS 线程的状态由线程控制块 TCB 中的 State 域保存，在 ps/psp.h 中定义的线程状态如下所示：

```
typedef enum _THREAD_STATE {
    Zero,                           //线程状态转换过程中的中间状态
    Ready,                          //就绪
    Running,                        //运行
    Waiting,                        //等待(阻塞)
    Terminated                      //结束
} THREAD_STATE;
```

EOS 线程的状态及其转换过程参见图 5-3。在椭圆圈内的是线程的状态，箭头表示状

态的转换过程。EOS 被设计为运行在单处理器上的多任务操作系统，所以，在任意时刻，最多只能有一个处于运行状态的线程占用处理器，而处于其他状态的线程数量可以为 0 个或多个。注意，由于 Zero 状态是线程状态转换过程中的中间状态，所以在图 5-3 中并没有出现。

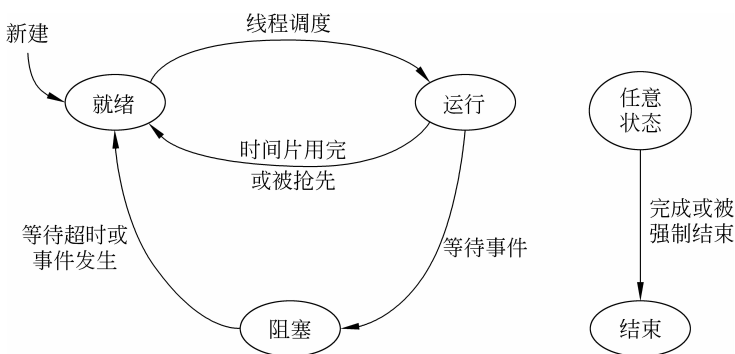

图 5-3　线程的状态和转换过程

2. 转换

下面对 EOS 线程状态的转换过程做一些简单的介绍。

1）新建→就绪

当创建一个进程或线程时，新进程的主线程或者新线程都会被初始化为就绪状态，并被放入就绪队列中。

2）就绪→运行

当调度程序认为某个处于就绪状态的线程应当执行时，便使其成为当前运行的线程，该线程就会从就绪状态进入运行状态。

3）运行→就绪

当前运行线程因时间片用完或被更高优先级线程抢先时，当前运行线程就会由运行状态转入就绪状态。

4）运行→阻塞

当前运行线程可能因调用 API 函数 WaitForSingleObject 等待事件或者执行 I/O 请求而被阻塞，从而由运行状态转入阻塞状态。

5）阻塞→就绪

处于阻塞状态的线程所等待的事件变为有效后，或等待超时后，该线程将被唤醒，从而由阻塞状态进入就绪状态。

6）任意状态→结束状态

线程可以由任意一个状态转入结束状态。例如，线程执行完毕会由运行状态转入结束状态，就绪线程或者阻塞线程如果被强制结束，也会转入结束状态。

7）转换时调用的函数

在 EOS 中，线程在不同的状态间相互转换时，最终都是通过调用 ps/sched. c 文件中的函数来完成的。

（1）PspReadyThread：将指定线程插入其优先级对应的就绪队列的队尾，并修改其状