第 3 章

进程的抽象

操作系统中最核心的概念是进程。进程是对正在运行的程序的一种抽象，是资源分配和独立运行的基本单位。进程是操作系统世界的一等公民。

3.1 多道程序设计与进程的引入

在现在使用的计算机上，我们经常会一边听着音乐，一边从网上下载文件，同时还编辑文档等。

假设计算机为单处理器的，在计算机的内存中有音乐播放(M)、文件下载(F)和文档编辑(W)三道运行中的程序，这三道运行程序在处理器上交替运行，如图 3.1 所示。当然，每次切换的时间对处理器而言是不可忽略的，而对我们而言实在太短暂，短到完全可忽略。这就是我们前面提到过的“并发”。

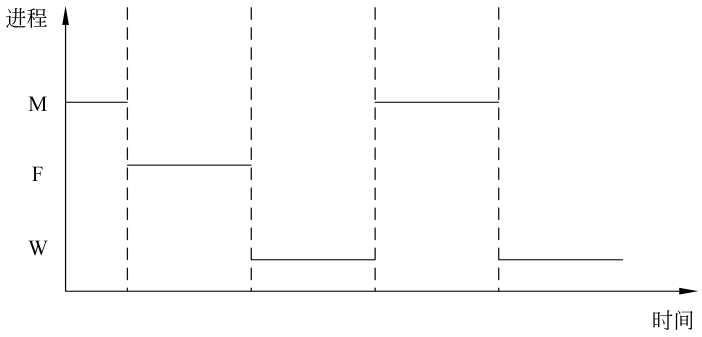


图 3.1 多道程序并发运行

在早期的计算机系统中，一次只有一个程序在计算机中运行。该程序能够完全地控制系统，访问计算机的所有资源。在多道程序设计的环境下，程序的并发执行代替了程序的顺序执行，使得程序活动不再处于一个封闭系统中，而出现了许多新的特征，即：间断性、失去封闭性和不可再现性。在此情形下，严格且准确地区分静态的程序和动态的(运行中的)程序就非常必要。

我们引入“进程”(Process)来刻画程序的动态执行，计算机上所有可运行的软件(通常包括操作系统)被组织成若干顺序进程。一个进程就是一个正在执行的程序，包括程序计数器、寄存器和变量的当前值。进程是可并发执行的程序在一个数据集合上的运行过程，是系统进行资源分配和调度的一个独立单位。单个处理器被若干进程共享，它使用某

种调度算法决定何时停止一个进程的工作，并转而为另一个进程提供服务。

程序与进程概念的区分

程序是静态的概念，而进程是动态的，是程序的动态执行。做一个类比：程序好比是一本书，而进程好比是对这本书的阅读过程。

直观而言，在硬盘上会存储很多的程序，但是仅当运行它时，它才成为进程。此外，同一个程序可以对应多个进程。

3.2 进程结构与进程控制

3.2.1 进程控制块

如果将操作系统比喻为计算机世界的国王，就可以将进程看作是这个计算机世界的一等公民。如此说来，管理计算机世界首先就要归结到对进程的管理，要对进程进行管理，首先需要建立一个专用数据结构来描述进程的相关信息，我们称这个数据结构为进程控制块(Process Control Block，PCB)。PCB也称为任务控制块、进程表项、任务结构体或者切换框(Switch Frame)，它是操作系统内核中的一个数据结构，包含管理进程所需的信息。可以说，PCB是进程在操作系统中的显现(the manifestation of a process in an operating system)[①]。

PCB在进程管理中扮演核心的角色，它被大多数操作系统部件所访问和修改，涉及调度、内存与I/O资源访问和性能监控等。可以说PCB定义操作系统的当前状态。PCB存储许多不同的数据项，尽管这些结构的细节与特定系统相关，我们还是能够标识出一些通用的部分，并将其分为进程标识信息、进程状态信息与进程控制信息三大类，具体内容如表3.1所示。

表3.1 进程控制块的相关信息

大 类	项	可能的用途
进程标识信息	进程ID	用于操作系统唯一所指称
	父进程	用于CPU时间和I/O时间的记账
	用户标识符	用于解析进程的所有权
	用户组标识符	用在组内成员之间的进程共享
进程状态信息	进程可见的寄存器组	在进程运行过程中存储临时值
	位置计数器	当进程开始运行时，用作连续点
	标识的条件码	反映以前执行的指令的状态
	栈指针	指向过程和系统调用的栈，以跟踪调用序列的轨迹

① Deitel, Harvey M. (1984) [1982]. *An introduction to operating systems* (revisited first ed.). Addison-Wesley. p. 673. ISBN 0-201-14502-2. pages 57-58.

续表

大　类	项	可能的用途
进程控制信息	状态	进程状态(例如,就绪、运行等)
	优先级	某个进程相对于其他进程的先导关系,用于调度
	进程起始时间	用于记账和调度
	所用的 CPU 时间	用于响应时间的计算和调度
	子进程所用的 CPU 时间	用于响应时间的计算和调度
	调度参数	用于调度进程中所需的资源和 CPU
	链接	进程的列表、队列或栈,以考虑未来的使用和它接收的顺序
	进程特权	能够指派特定的许可给特定的进程以使用系统资源
	内存指针	跟踪指派给进程的内存位置
	进程间通信信息	跟踪来自或指向其他进程所挂起的信号量、消息等,以便之后使用
	分配给进程的设备	在进程终止之前,释放进程所占有的所有资源
	打开的文件	在进程终止之前,将所有未写入的数据刷新到文件中,并关闭文件
	标志	用于不同用途,例如中断标志将通知系统未来必须要处理的一个挂起的中断信号

1. 进程标识信息

进程标识信息通常包含进程的唯一标识符,在多用户、多任务系统中,还会包括父进程标识符、用户标识符、用户组标识符等。

2. 进程状态信息

进程状态信息定义进程状态相关的信息,当进程挂起的时候,允许操作系统在之后重启进程,而进程还能正确执行。进程状态信息通常包含 CPU 中通用寄存器的内容、CPU 进程状态字、栈和帧指针等。当发生进程调度时,操作系统必须能够将硬件寄存器的值复制到 PCB 中。

3. 进程控制信息

进程控制信息是操作系统管理进程所用的信息,通常包括:

(1) 进程调度状态:进程调度状态与其他调度信息,例如优先级、进程获得 CPU 的运行时间。同时,对于挂起的进程,尚需记录进程所等待事件的标识。

(2) 进程结构化信息:包含子进程标识符或者与当前进程有关联的其他进程的标识符。这些信息可以队列、环或者其他数据结构来表征。

(3) 进程间通信信息:与进程间通信有关的各种标志、信号量或者消息。

(4) 其他信息:包含授权许可系统资源访问的特权级、进程执行所用的 CPU 数量、时间限制和执行标识符等记账信息,以及分配给进程的 I/O 设备列表等的 I/O 状态信息等。

由于 PCB 包含进程的关键信息，它必须保存在被保护的内存区域，防止普通用户非法访问。在某些操作系统中，PCB 置于进程内核栈的起始位置。

值得强调的是，进程的现场信息都保留在 PCB 中。PCB 保存了进程调度过程中进程恢复所需用到的许多信息。由于多个进程并发执行，轮流使用处理器，当某进程不在处理器上运行时，必须保留其被中断的进程现场。进程现场信息包括断点地址、程序状态字、通用寄存器的内容、堆栈内容、程序当前状态、程序的大小和运行时间等信息。保存进程现场信息，可以使得进程再次获得处理器时能够正确执行。

现场保护与还原

进程并发执行中非常重要的一个技术环节是进程运行环境的无缝切换。好比说，我们的教室上一节课上的是外语听力课，下一节课是化学实验课，那么当外语听力课结束后，教室管理人员就必须将课堂内的听力设备换成化学实验的设备，而且最好是能够保持上一次化学实验课结束后的状态。

现场保护与还原是操作系统实现过程中非常重要且富有技巧性的一步。

进程控制块是进程存在的唯一标志，它跟踪进程执行的情况，表明了进程在当前时刻的状态以及与其他进程和资源的关系。当创建一个进程时，实际上就是为其创建一个进程控制块。

【例 3.1】 xv6 操作系统的 PCB。

xv6 是美国麻省理工学院(MIT)为操作系统工程的课程(课程编号 6.828)开发的一个教学操作系统。图 3.2 是 xv6 中 PCB 的数据结构描述。

```
//xv6 保存和恢复的寄存器组,用于停止并恢复进程的运行
struct context {
    int eip;
    int esp;
    int ebx;
    int ecx;
    int edx;
    int esi;
    int edi;
    int ebp;
};
//进程的状态(枚举值)
enum proc_state { UNUSED, EMBRYO, SLEEPING,RUNNABLE, RUNNING, ZOMBIE };
//xv6 跟踪每个进程的相关信息,包括寄存器上下文与状态
struct proc {
    char * mem;                    //Start of process memory 进程内存的初始状态
    uint sz;                       //Size of process memory 进程内存的大小
    char * kstack;                 //Bottom of kernel stack 内核栈的栈底
//for this process
    enum proc_state state;         //Process state 进程状态
    int pid;                       //Process ID 进程 ID
    struct proc * parent;          //Parent process 父进程
```

图 3.2　xv6 进程结构

```
    void * chan;        //If non-zero, sleeping on chan 如果非零,则在该 chan 上睡眠
    int killed;         //If non-zero, have been killed 如果非零,则已被杀死
    struct file * ofile[NOFILE];        //Open files 所打开的文件
    struct inode * cwd;                 //Current directory 当前目录
    struct context context;             //Switch here to run process 上下文(现场)
    struct trapframe * tf;              //Trap frame for the 陷阱帧
//current interrupt
};
```

图 3.2 （续）

可以看出，xv6 的 PCB 是一个比较简化版的 PCB，其中核心包含的内容有进程编号 pid、内存地址 mem 以及内存栈栈底 kstack。此外还会记录进程的状态 state、所打开的文件列表 ofile[NOFILE]以及与进程的当前目录相关的 i 节点信息 cwd。最后包括涉及进程切换的上下文，包含保存寄存器组的 context 和陷阱帧 tf 等信息。

3.2.2 进程的创建

操作系统需要一种创建进程的方法。一些简单的系统，例如微波炉控制器，或许在系统启动的时候，所需的进程都已经就绪。然而，在通用系统中，就需要在运行过程中动态创建或者撤销进程。

通常引发进程创建的事件包括：

(1) 系统初始化。

(2) 运行的进程执行了创建进程的系统调用。

(3) 用户请求创建一个新进程。

(4) 初始化一个批处理作业。

从技术上来看，上述 4 种情形都是由一个已经存在的进程执行一个创建进程的系统调用来创建一个新进程。创建进程的系统调用通知操作系统创建一个新进程，并直接或间接指定新进程所运行的程序。

创建进程的系统调用一般过程如下：

(1) 申请 PCB 空间，根据建立的进程名字查找 PCB 表，若找到了则非正常终止(即已有同名进程)，否则申请分配一块 PCB 空间。

(2) 为新进程分配资源，若进程的程序或数据不在内存中，则应将它们从外存调入分配的内存中，然后把有关信息(如进程名字、信号量和状态位等)分别填入 PCB 的相应栏目中。

(3) 最后把 PCB 插入到就绪队列中。

创建进程的系统调用在不同的操作系统中实现有所不同，例如在 Linux 系统中，通常使用 fork()，而在 Windows 中，则会使用一个 CreateProcess 系统调用。在调用 fork()时，操作系统会创建一个与调用进程完全相同的副本。这样，两个进程(父进程和子进程)拥有相同的内存映像以及同样的环境字符串。

【例 3.2】 Linux 中调用 fork()创建进程。

如图 3.3 所示，在 Linux 中调用 fork()创建进程。父进程在执行 fork()之后，将克隆出一个子进程(如果没有发生错误的话)。

```
#include<sys/types.h>
#include<stdio.h>
#include<unistd.h>
int main()
{
    pid_t pid;
/* 创建一个子进程 */
    pid=fork();
    if (pid<0) { /* 发生错误 */
        fprintf(stderr, "Fork Failed");
        return 1;
    }
    else if (pid==0) { /* 表明是子进程 */
        execlp("/bin/ls","ls",NULL);
    }
    else { /* 表明是父进程 */
        /* 父进程等待子进程运行结束 */
        wait(NULL);
        printf("Child Complete");
    }
    return 0;
}
```

图 3.3　Linux 中调用 fork ()创建进程

其中 fork()函数生成当前运行进程的一个副本。副本中还包含父进程的进程栈、数据区域和堆。在 fork 语句之后开始执行。

fork()函数包含在头文件<unistd.h>中，其函数原型如下：

```
#include<unistd.h>
pid_t fork(void);
```

其中函数的返回值说明如表 3.2 所示。

表 3.2　fork()函数的返回值

返回值	含　义
>0	子进程的 ID，父进程可以通过该 ID 来跟踪子进程
0	返回的是子进程
<0	发生一个错误。没有创建子进程，错误号标识具体的问题

3.2.3　进程的终止

进程完成了其“历史使命”之后，应当退出系统而消亡，系统及时收回它占有的全部资

源以供其他进程使用。引起进程终止的事件通常包括：

(1) 进程的正常退出(自愿)。

(2) 进程的出错退出(自愿)。

(3) 进程出现严重错误(非自愿)。

(4) 被其他进程杀死(非自愿)。

进程的终止对于操作系统而言需要回收进程的各种资源，一般的终止过程如下：

(1) 根据提供的欲终止进程的名字(ID)，在 PCB 链中查找对应的 PCB，若找不到要终止的进程或该进程尚未停止，则转入异常终止处理程序，否则从 PCB 链中撤销该进程及其所有子孙进程(因为仅撤销该进程可能导致其子进程与进程家族隔离开来，而成为难以控制的进程)。

(2) 检查此进程是否有等待读取的消息，如果有，则释放所有缓冲区。

(3) 最后释放该进程的工作空间、PCB 空间以及其他资源。

3.3 进程状态转换与上下文切换

3.3.1 进程的状态及其转换

既然进程是程序的动态执行，那就可以用动态的观点分析进程的状态变化及相互制约关系。假设进程具有三种基本状态：运行状态、阻塞状态与就绪状态。这三种状态构成了最简单的进程生命周期模型，进程在其生命周期内处于这三种状态之一，其状态将随着自身的推进和外界环境的变化而变化，由一种状态变迁到另一种状态。

(1) 运行状态：进程正处于 CPU 上运行的状态，这表明该进程已获得必要的资源。在单 CPU 系统中，至多只有一个进程处于运行状态，在多 CPU 系统中，可以有多个进程处于运行状态。

(2) 阻塞状态：进程等待某个事件完成(例如，等待输入/输出操作的完成)而暂时不能运行的状态。处于该状态的进程不能参与竞争 CPU，因为此时即使分配给它 CPU，它也不能运行。

(3) 就绪状态：进程处于等待 CPU 的状态，这表明该进程已经获得了除 CPU 之外的一切运行所需的资源。由于 CPU 个数少于进程个数，所以该进程不能运行，而必须等待分配 CPU 资源，一旦获得 CPU 就立即投入运行。在一个系统中，处于就绪状态的进程可能有多个，排成一个队列，称为就绪队列。

进程的状态转换如图 3.4 所示。

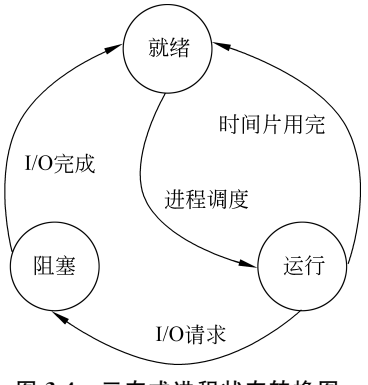


图 3.4 三态式进程状态转换图

(1) 从就绪状态转换到运行状态(就绪态→运行态)：处于就绪状态的进程已具备了运行条件，但由于未能获得 CPU，故仍然不能运行。对于单 CPU 系统而言，因为处于就绪状态的进程往往不止一个，同一

时刻只能有一个就绪进程获得CPU。进程调度程序根据调度算法把CPU分配给某个就绪进程，把它由就绪状态变为运行状态，并把控制转到该进程，这样进程就投入运行。即就绪状态的进程一旦被调度进程选中，获得CPU，便发生此状态转换。触发此状态转换的事件通常包括进程调度。

(2) 从运行状态转换到阻塞状态(运行态→阻塞态)：处于运行状态的进程申请新资源而又不能立即被满足时，进程状态便由运行状态转换为阻塞状态。例如，运行中的进程需要等待文件的输入，系统便自动转入系统控制程序进行文件输入，在文件输入过程中该进程进入阻塞状态。而系统将控制转给进程调度，进程调度根据调度算法把CPU分配给处于就绪状态的其他进程。触发此状态转换的事件通常包括I/O请求等。

(3) 从阻塞状态转换到就绪状态(阻塞态→就绪态)：被阻塞的进程在其被阻塞的原因获得解除后，并不能立即投入运行，于是将其状态由阻塞状态转换为就绪状态继续等待CPU。仅当进程调度程序把CPU再次分配给它时，才可恢复曾被中断的现场继续运行。触发此状态转换的事件通常包括阻塞进程的I/O请求完成等。

(4) 从运行状态转换到就绪状态(运行态→就绪态)：对于一个正在运行的进程，由于规定运行时间片用完而将该进程的状态修改为就绪状态，并根据其自身的特征而插入就绪队列的适当位置，同时保存进程现场信息，CPU开始运行进程调度。触发此状态转换的事件通常包括时间片用完等。

思　考　题

请列举什么事件会触发如下的状态转换：

(1) 进程从运行态转换到阻塞态。

(2) 进程从运行态转换到就绪态。

(3) 进程从阻塞态转换到就绪态。

(4) 进程从就绪态转换到运行态。

在许多系统中，进程除了具有上述三种基本状态以外，又增加了一些新状态，其中最常见的是因为挂起(Suspend)而使得阻塞态和就绪态这两个状态分裂为4个状态，即阻塞态、阻塞挂起态、就绪态与就绪挂起态。挂起的主要原因是内存资源不足，其中涉及虚拟内存管理(参考第10章)的一些问题。此外，与挂起对偶的操作是恢复(Resume)。在引入挂起后，进程的状态转换如图3.5所示。

(1) 从就绪状态转换到就绪挂起状态(就绪态→就绪挂起态)：当处于就绪状态的进程被挂起后，该进程便转换为就绪挂起状态，处于该状态的进程不再被调度执行。

(2) 从阻塞状态转换到阻塞挂起状态(阻塞态→阻塞挂起态)：当处于阻塞状态的进程被挂起后，该进程便转换为阻塞挂起状态，处于该状态的进程在其所期待的事件出现后，将从阻塞挂起状态转换为就绪挂起状态。

(3) 从就绪挂起状态转换到就绪状态(就绪挂起态→就绪态)：对于处于就绪挂起状态的进程，若恢复后，该进程将转换为就绪状态。

(4) 从阻塞挂起状态转换到阻塞状态(阻塞挂起态→阻塞态)：对于处于阻塞挂起状

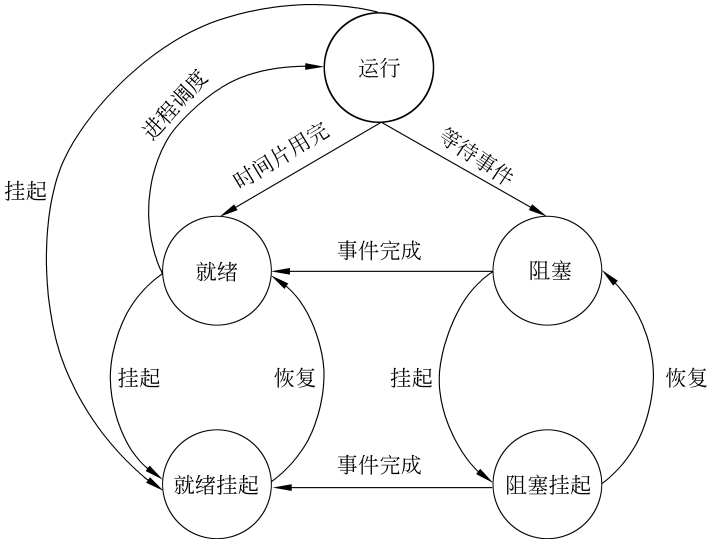


图 3.5　引入挂起的进程状态转换图

态的进程，若恢复后，该进程将转换为阻塞状态。

阻塞和挂起的区别是什么？

阻塞是因为要等待某个资源而无法运行，而挂起是进程暂时被调离出内存，当条件允许的时候会被操作系统再次调回内存。挂起依赖于操作系统存储管理中的虚拟存储，即将部分硬盘虚拟化为内存。挂起的进程实际上是被挪置于虚拟存储上。

3.3.2　上下文切换

所谓的并发，其核心就是多个进程在同一时间段内“同时”运行。这意味着任何一个进程从微观角度讲都是“走走停停”的状态，即经常性地出现进程切换。这种进程上下文切换(Context Switch)可以是在操作系统从当前运行进程中获得控制权的任何时刻发生的。

图 3.6 展示了两个进程 P_1 和 P_2 交替执行的场景。起初 CPU 控制权是在进程 P_1 手中，当进程 P_1 由于中断被打断运行后，控制权转移给操作系统。此时，操作系统按照算法决策切换进程 P_2 执行，首先将进程 P_1 的 PCB_1 保存起来，然后从 PCB_2 获取进程 P_2 的运行上下文，并恢复现场。操作系统在完成这一系列“魔法”之后，把控制权交给进程 P_2，然后 P_2 运行。在一段时间后，进程 P_2 也由于中断被打断运行，如法炮制，操作系统将上下文信息保存到 P_2 的 PCB_2 中，然后从 P_1 的 PCB_1 中获取运行上下文并恢复现场，P_1 便继续运行。

进程切换有时候也称为上下文切换，此处的上下文是指进程运行的环境，一般包括 CPU 寄存器的内容和程序计数器。上下文切换指的是将 CPU 切换到另一个进程需要执行当前进程的状态保存和不同进程的状态恢复。典型的上下文切换步骤如下：

(1) 保存处理器的上下文，包括程序计数器和其他寄存器。

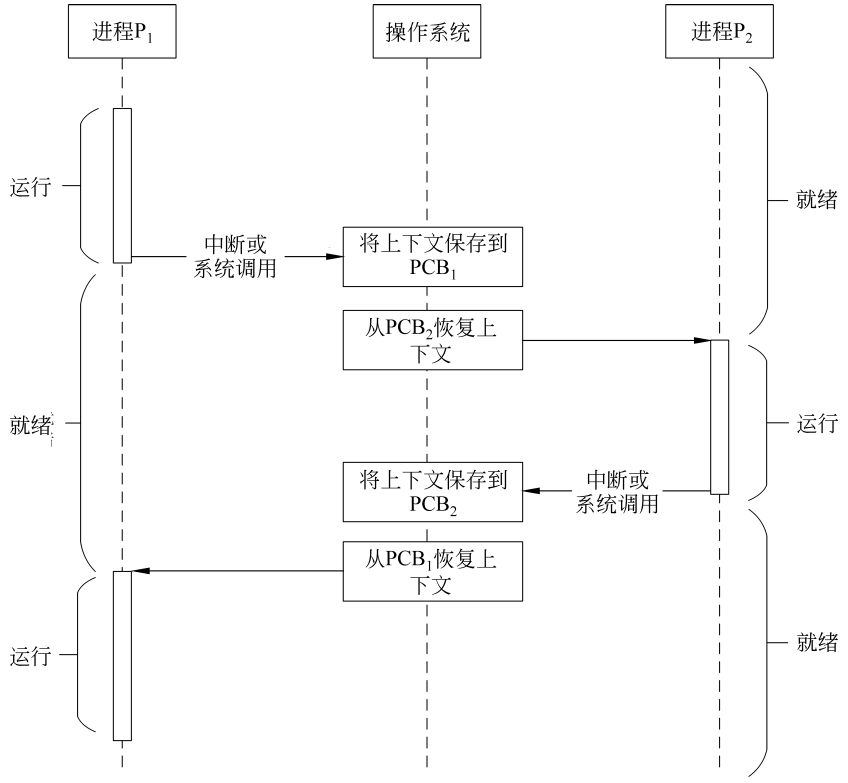


图 3.6　典型的进程切换图

(2) 更新当前处于运行态进程的进程控制块。这包括将进程的状态置于其他状态(例如,就绪态、阻塞态等)。同时更新其他相关的数据,例如离开运行态的原因和记账信息等。

(3) 将该进程的进程控制块移到合适的队列中(例如就绪队列或者阻塞队列等)。

(4) 通过某种调度算法选择另外一个进程运行。

(5) 更新被选中进程的进程控制块,同时将该进程的状态置于运行态。

(6) 更新内存管理的数据结构,这依赖于地址映射等管理方式。

(7) 通过加载程序寄存器或其他寄存器的值,将处理器的上下文恢复到被选中进程上次被切换出运行态时的状态。

之所以称之为上下文切换,其中一个含义便是进程运行的上下文(环境)发生改变与切换,由进程 P_1 的运行上下文切换到进程 P_2 的运行上下文。

此外,需要将上下文切换与模式切换区分开,模式切换指的是内核态和用户态之间的切换。上下文切换只出现在内核模式中。内核模式是 CPU 的特权模式,只有内核才能在此模式下运行,它有权对所有内存位置和所有其他系统资源进行访问。其他程序(包括应用程序)通常都是在用户模式下运行,但是可以通过系统调用来运行部分内核代码。上下文切换必然会伴随模式切换,而模式切换则不一定涉及上下文切换。模式切换并不需要改变当前处于运行态的进程状态。

当发生上下文切换时，内核将旧进程的上下文保存在其 PCB 中，并加载计划运行的新进程的上下文。上下文切换的时间是纯粹的开销，因为系统在切换时没有做任何有用的工作。它的速度因机器而异，具体取决于内存速度、必须复制的寄存器数量以及是否存在特殊指令（例如加载或存储所有寄存器的单条指令）。此外，操作系统越复杂，在上下文切换期间必须完成的工作就越多。例如，当操作系统引入高级内存管理技术之后，它可能需要在每个上下文中切换额外的数据，涉及当前进程地址空间的保留问题，需要考虑如何保留地址空间以及保存多少地址空间等问题。

上下文切换通常是计算密集型的。也就是说，它需要一定的处理器时间，大致是在纳秒级。这样，上下文切换对于 CPU 的时间而言是一个可观的数量，实际上是成本很高的操作。通常，操作系统设计的重点之一便是尽可能地避免不必要的上下文切换。然而，这并不容易实现。事实上，尽管就所消耗的 CPU 时间的绝对数量而言，上下文切换的开销一直在减少，然而这仅仅是因为 CPU 时钟速度的提升而非上下文切换效率的改进。

3.4 线　　程

3.4.1 线程的引入

在操作系统中引入进程的目的，是为了使多个进程并发执行，以改善资源利用率及提高系统的吞吐量。在此基础上，操作系统中再引入线程一方面是为了进一步地促进系统的并发，另一方面则是为了减少进程并发执行时所付出的时空开销。

从进一步促进系统并发的角度来看，对于多道程序设计而言，进程的并发使得多个进程能够“同时”运行，然而对于单个进程内部而言，只能串行。从程序员的视角来看，他们所开发的程序以进程的方式运行时，也希望能够“同时”进行多项任务与工作。例如，一个文字编辑与排版的 Word 进程，该进程的一个任务是能够进行文档的编辑与排版，另一个任务是能够在后台定期地对所编辑的文档进行自动保存。这两个任务必须“同时”完成，这要求在一个进程内部能有多个执行序列。通过引入线程，将进程的执行转换为进程内的多个线程的执行，从而实现进程内的并发执行。

此外，线程的引入也有助于操作系统应对进程并发过程中上下文切换所带来的开销。为了说明这一点，我们首先回顾进程的两个基本属性：

（1）进程是一个拥有资源的独立单位。

（2）进程同时又是一个独立调度和分配的基本单位。

正是由于进程具有这两个基本属性，才使之成为一个能独立运行的基本单位，从而构成了进程并发执行的基础。

由于进程是一个资源拥有者，因而在进程的创建、撤销和切换中，系统必须为之付出较大的时空开销。也正因为如此，在系统中所设置的进程数目不宜过多，进程切换的频率也不宜过高，但这也就限制了并发度。

如何能使多个进程更好地并发执行，同时又尽量减少系统的开销，已成为设计操作系统时所追求的重要目标。于是，操作系统的研究学者们想到，可否将进程的上述属性分

开，由操作系统“分而治之”。即将 CPU 调度和资源分配针对不同的活动实体进行，以使之轻装运行。而对拥有资源的基本单位，又不频繁地对其进行切换。正是在这种思想的指导下，操作系统引入了线程的概念。

在引入线程的操作系统中，线程是进程中的一个实体，是被系统独立调度的基本单位。线程自己基本上不拥有系统资源，只拥有一些在运行中必不可少的资源（如程序计数器、一组寄存器和栈），但它可与同属一个进程的其他线程共享进程所拥有的全部资源。一个线程可以创建和撤销另一个线程，同一进程中的多个线程之间可以并发执行。由于线程之间的相互制约，致使线程在运行中也呈现出间断性。相应地，线程也同样有就绪、阻塞和运行三种基本状态，有的系统中线程还有终止状态。

与进程类似，线程控制块（Thread Control Block，TCB）是操作系统内核管理线程相关信息的数据结构。在 TCB 中通常包含：

（1）线程标识符。它是指派给每个线程的唯一标识符。

（2）栈指针。它指向进程中的线程栈的指针。

（3）程序计数器。它是指向线程的程序指令。

（4）线程的状态。它通常包括就绪态、阻塞态和运行态等。

（5）线程的寄存器值。

（6）指向线程所在进程的进程控制块 PCB 的指针。

3.4.2　进程与线程的关系

线程具有许多传统进程所具有的特征，故又称为轻量型进程（Light-Weight Process）或进程元，而把传统的进程称为重型进程（Heavy-Weight Process），它相当于只有一个线程的任务。在引入了线程的操作系统中，通常一个进程都有若干个线程，至少需要有一个线程。为了更加清晰地辨析进程与线程的关系，我们从调度、并发性、拥有资源和系统开销等方面将两者加以比较，对比情况如表 3.3 所示。

表 3.3　进程与线程的对比

	进　　程	线　　程
调度	跨进程时，会涉及调度	调度的基本单位
并发性	多进程并发	并发粒度更高，进程内仍可并发
资源拥有	资源拥有的基本单位	拥有寄存器组等运行所需的最少资源
系统开销	进程切换的开销较大	线程切换的开销较小

1. 调度

在传统的操作系统中，拥有资源的基本单位和独立调度的基本单位都是进程。而在引入线程的操作系统中，则把线程作为调度的基本单位，而进程只是作为拥有资源的基本单位，这使传统进程的两个属性分开，线程便能轻装运行，从而可显著地提高系统的并发程度。在同一进程中，线程切换不会引发进程切换，只有当从一个进程中的线程切换到另一个进程中的线程时，才会引发进程切换。

2. 并发性

在引入线程的操作系统中，不仅进程之间可以并发执行，而且在一个进程中的多个线程之间亦可并发执行，因而使操作系统具有更好的并发性，从而能更有效地使用系统资源和提高系统吞吐量。例如，在一个未引入线程的单 CPU 操作系统中，若仅设置一个文件服务进程，当它由于某种原因被阻塞时，便没有其他的文件服务进程来提供服务。在引入了线程的操作系统中，可以在一个文件服务进程中设置多个服务线程。当第一个线程等待时，文件服务进程中的第二个线程可以继续运行；当第二个线程阻塞时，第三个线程可以继续执行，以此类推，从而显著地提高了文件服务的质量以及系统吞吐量。

3. 资源拥有

无论是传统的操作系统，还是设有线程的操作系统，进程都是拥有资源的一个独立单位，它可以拥有自己的资源。一般地说，线程自己不拥有系统资源（只有一些必不可少的运行资源），但它可以访问其所在进程的资源，包括进程的代码段、数据段以及系统资源，如已打开的文件、I/O 设备等，这些资源可供同一进程的所有线程共享。

4. 系统开销

由于在创建或撤销进程时，系统都要为之分配或回收资源，如内存空间、I/O 设备等，因此操作系统所付出的开销将显著地大于创建或撤销线程时的开销。类似地，在进行进程切换时，涉及当前进程上下文的保存以及被调度运行进程的上下文设置。而线程切换只需保存和设置少量寄存器的内容，并不涉及存储器管理方面的操作。在有的系统中，线程的切换、同步和通信都无需操作系统内核的干预。

更进一步，图 3.7 展示了没有线程的进程模型和多线程进程模型的直观差异。在一个多线程环境中，进程是资源的分配和保护的单元，与进程相关的部分包括：

（1）容纳进程镜像的虚拟地址空间。

（2）对处理器、其他进程和进程内的文件及 I/O 资源的访问保护。

在进程内，可以有一个或者多个线程，每个线程包括：

（1）线程执行状态（运行态、就绪态等）。

（2）当不处于运行态的时候，具有被保存的线程上下文。

（3）执行栈。

（4）每个线程都有自己独立的局部变量存储空间。

在没有线程的进程模型中，进程模型包括进程控制块、用户地址空间以及管理进程执行中的调用或返回行为的用户栈与内核栈。在多线程的模型中，仍然有单独的进程控制块，以及与进程相关的用户地址空间，但是对于每个线程具有各自的栈以及各自的控制块，线程控制块包含寄存器的值、优先级以及其他与线程相关的信息。

一个进程中的所有线程共享进程的状态和资源。它们栖居在同一个地址空间，访问同样的数据。如果一个线程修改内存中的某个数据项，那么其他线程访问该数据项的时候，将看到修改后的结果。如果一个线程用读权限打开一个文件，那么同一个进程中的其他线程仍可以读取该文件。

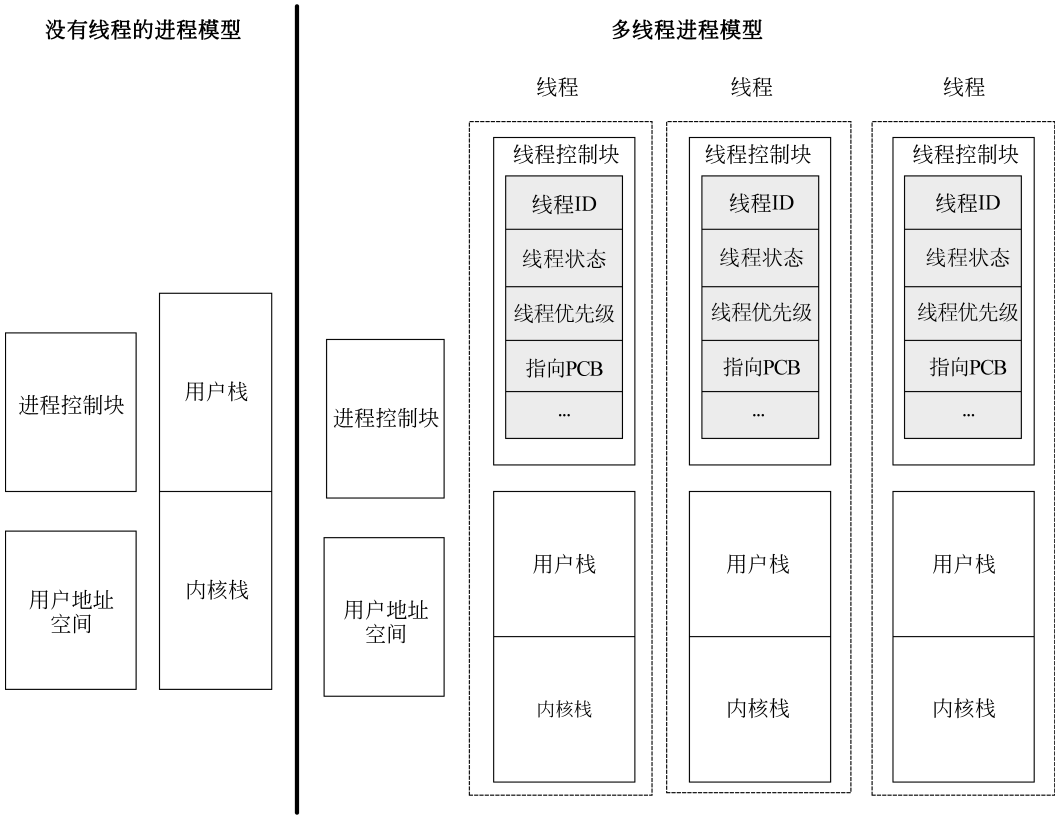


图 3.7　没有线程的进程模型和多线程进程模型

3.4.3　POSIX 线程

POSIX 线程(POSIX Threads,简写为 Pthreads)是 POSIX 的线程标准,定义了创建和操纵线程的一套应用程序接口(API)。实现 POSIX 线程标准的库常被称为 Pthreads,一般用于 UNIX 类型的 POSIX 系统,如 Linux、Solaris 等。但是 Microsoft Windows 上也有相应的实现。Pthreads 定义了一套 C 语言的类型、函数与常量,它以 pthread.h 头文件和一个线程库实现。Pthreads API 中大致共有 100 个函数调用,全都以“pthread_”开头。

在 Pthreads 中包含的主要数据类型包括:

(1) pthread_t: pthread_t 指代线程句柄,使用函数 pthread_self()获取自身所在线程 id。

(2) pthread_attr_t: pthread_attr_t 指代线程属性。主要包括 scope 属性、detach 属性、堆栈地址、堆栈大小和优先级。

在 Pthreads 中包含的主要函数如下。

(1) 创建一个线程,函数原型如下:

```
#include<pthread.h>
```

```
int  pthread_create(pthread_t *thread,
                    pthread_attr_t *attr,
                    void *(*start_routine)(void *),
                    void *arg);
```

形参说明如表 3.4 所示。

表 3.4　pthread_create 的形参说明

形　参	说　明
thread	指向新创建线程句柄的指针。如果函数出现错误,该值则为未定义
attr	指向新创建线程所继承的属性指针。如果该参数为 NULL,那么继承默认的属性
start_routine	指向新创建线程所执行的函数指针
arg	传递给 start_routine 的参数。只能传递一个参数,如果希望传递多个参数,则需要将参数打包到一个结构体,并将结构体的地址作为参数

创建线程将为当前进程增加一个新的线程。一个新创建的线程与进程中的其他线程共享进程的全局数据,同时它具有自身的属性集合以及私有的执行栈。新的线程继承调用线程的信号量掩码,也有可能继承调度优先级。新创建的线程的挂起信号量不会继承其调用线程,其值为空。

(2) 获取线程标识符,函数原型如下:

```
#include<pthread.h>
pthread_t pthread_self( void );
```

pthread_self()获取调用线程的线程句柄。

(3) 让出线程的执行,函数原型如下:

```
#include<pthread.h>
void pthread_yield( void );
```

pthread_yield()使得当前线程让出 CPU 给具有相同或者更高优先级的其他线程。

(4) 终止一个线程,函数原型如下:

```
#include<pthread.h>
void pthread_exit( void * status );
```

pthread_exit()函数终止调用线程,释放所有与线程相关的绑定数据。

(5) 等待线程的结束,函数原型如下:

```
#include<pthread.h>
int pthread_join( pthread_t wait_for, void **status );
```

pthread_join()函数阻塞调用线程,直到所等待的线程终止。所等待的线程必须处于当前进程。当参数 status 不为 NULL 时,那么当 pthread_join()成功返回时,它会指向终止线程的退出状态。

【例 3.3】 使用 POSIX 线程库创建线程。

使用 POSIX 线程库创建线程的一个典型方式如图 3.8 所示。在创建线程之前，需要首先定义两个变量：一个是 pthread_t 类型的变量，用来存储线程标识符；另一个是 pthread_atttr_t 类型的变量，它存储线程属性集合，同时该变量需要通过函数 pthread_attr_init 进行初始化。然后，调用 pthread_create()函数创建线程，函数的实参除了上述两个参数之外，还需要一个指向函数的指针作为实参，这个函数指针所指向的函数是线程的执行体。在图 3.8 的示例中，线程体名为 runner，其主要操作是对所输入的参数值做累加赋值给 sum，即如果输入参数为 N，那么：

$$sum = 1 + 2 + 3 + \cdots + N$$

```
#include<pthread.h>
#include<stdio.h>
int sum;                              /*该数据由所有线程共享*/
void *runner(void *param);            /*线程体*/
int main(int argc, char *argv[])
{
    pthread_t tid;                    /*线程标识符*/
    pthread_attr_t attr;              /*线程属性集合*/
    if (argc !=2) {
        fprintf(stderr,"usage: a.out<integer value>\n");
        return -1;
    }
    if (atoi(argv[1])<0) {
        fprintf(stderr,"%d must be>=0\n",atoi(argv[1]));
        return -1;
    }
/*获得默认参数*/
    pthread_attr_init(&attr);
/*创建线程*/
    pthread_create(&tid,&attr,runner,argv[1]);
/*等待线程退出*/
    pthread_join(tid,NULL);
    printf("sum=%d\n",sum);
}
/*线程体*/
void *runner(void *param)
{
    int i, upper=atoi(param);
    sum=0;
    for (i=1; i<=upper; i++)
        sum +=i;
    pthread_exit(0);
}
```

图 3.8　使用 POSIX 线程库创建线程

本章小结

操作系统具有4个主要的管理功能：处理器管理、存储管理、设备管理和文件管理，这些管理的核心在于对计算机资源的管理，那么问题在于究竟是谁在使用计算机的资源呢？可能有人会说是用户，是我们人类在使用计算机资源，那我们怎么使用计算机资源呢？应该是应用程序，进程便是程序在计算机运行时的显现。在操作系统设计中抽象出进程的概念是开创性的。进程（或者此后的线程）是整个操作系统王国的一等公民，是多道程序设计的核心。进程也是计算机资源调度和分配的一个基本单位，计算和存储等资源都是以进程为单位进行调度和分配的。

操作系统对进程的管理以进程控制块（PCB）为主要数据结构，PCB中包含了与进程切换、进程调度和管理相关的各类信息。要理解与掌握进程的概念，必须将其与程序的概念进行对比和分析。

所有进程犹如钟表中的齿轮一般，在操作系统的调度下，繁忙而有序地工作着。进程在整个生命周期中，会处于不同的状态，各种状态之间会相互转换。转换过程中会涉及上下文切换，这种切换是操作系统所施展的一个戏法，整个过程非常精确且富有技巧性，此外这个过程要求效率极高，它的时延是操作系统性能的一个重要指标。

此外，随着并发度需求的进一步提升，为了进一步促进并发，同时减少进程切换所引发的系统开销，操作系统进一步引入了线程。在设计线程的过程中，操作系统实质上解耦了进程的资源分配和调度两个基本单位，即保留进程作为资源分配和拥有的基本单位，而将调度的基本单位赋予了线程，从而提高了并发度且降低系统因过度进行进程切换所带来的开销。

在研究领域，由于进程应该算是一个经典概念，因此针对进程本身的研究已经不太多了。线程相对而言是比进程更新的概念，不过它也被反复咀嚼过了。现在还略有一些研究，例如塔姆（Tam）等人在2007年开展了在多处理器上进行线程集群的研究[①]，博伊德·维克泽（Boyd-Wickizer）等人在2010年对多线程和多核下的Linux操作系统的扩展性进行了研究[②]。

习　　题

1. 现代操作系统中为什么要引入进程的概念？它与程序有什么区别？

2. 进程的含义是什么？试简述进程控制块的组成。

3. 一个应用程序包括几个进程：一个主进程和一些子进程。如果（　　），这种安排便能够加速计算。

---

① Tam D, Azimi R, Stumm M. *Thread clustering: sharing-aware scheduling on SMP-CMP-SMT multiprocessors*.[J]. Acm Sigops Operating Systems Review, 2007, 41(3): 47-58.

② Boyd-Wickizer S, Clements A T, Mao Y, et al. *An Analysis of Linux Scalabitity to Many Cores*[C]// Usenix Symposium on Operating Systems Design & Implementation. DBLP, 2010.

A. 计算机系统具有多个 CPU　　B. 一些进程受 I/O 限制

C. 某些进程受 CPU 限制　　D. 以上都不是

4. 判定以下陈述哪些为真，哪些为假。

A. 如果两个用户执行相同的程序，则操作系统会创建一个进程

B. 由于请求某个资源被阻塞的进程，当获得资源的时候，进程将改变为运行状态

C. 终止的进程和挂起的进程之间没有区别

D. 处理完事件后，如果没有任何进程状态发生更改，则内核无需在分发之前执行调度

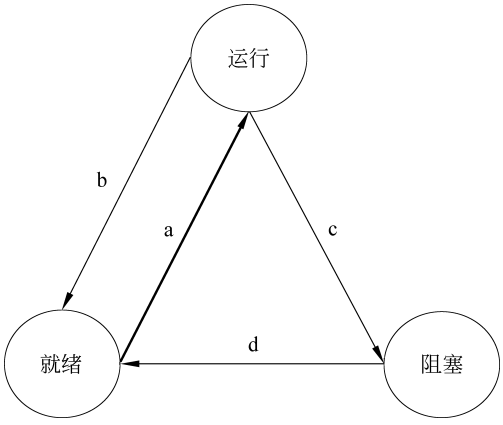
E. 当进程的用户级线程执行导致阻塞的系统调用时，进程的所有线程都将被阻塞

F. 无论是在单处理器还是多处理器系统中，内核级线程都能比用户级线程提供更好的并发性

G. 当进程终止时，应记住其终止代码，直到其父进程终止

5. 描述内核对进程之间的上下文切换所采取的操作。

6. 某系统的进程状态转换如下图所示，请回答如下问题。



(1) 引起各种状态转换的典型事件有哪些?

(2) 当我们观察系统中某些进程时，能够看到某一进程产生的一次状态转换能引起另一进程进行一次状态转换。在什么情况下，当一个进程发生转换 c 时能立即引起另一个进程发生转换 a?

7. 下述特征能够充分地表征某个操作系统为多道程序操作系统的是(　　)。

Ⅰ. 不止一个程序能够装载到内存中同时执行。

Ⅱ. 如果一个程序等待某些事件，例如 I/O，另外一个程序能够立即被调度执行。

Ⅲ. 如果程序的执行结束，另一个程序立即被调度执行。

A. Ⅰ　　B. Ⅰ和Ⅱ　　C. Ⅰ和Ⅲ　　D. Ⅰ、Ⅱ和Ⅲ

8. 进程的以下状态转换可能导致一个或者多个其他进程从阻塞态转换成就绪态的是(　　)。

A. 进程启动 I/O 操作并阻塞　　B. 一个进程终止

C. 进程发出资源请求并阻塞　　D. 进程发送消息

E. 进程从阻塞态转换为阻塞挂起状态

9. 当进程为下述目的进行系统调用时，请描述内核行为。

A. 请求接收消息　　B. 请求执行 I/O 操作

C. 请求有关进程的状态信息　　D. 请求创建进程

E. 请求终止子进程

10. 进程当前处于阻塞挂起状态。

A. 给出一个状态转换序列，通过它可以达到这种状态

B. 给出一系列状态转换，通过它可以达到就绪状态

在这些情况下，是否可能有多个状态转换序列？

11. 对比以下应用中内在的并发性。

A. 一个在线银行应用，允许用户通过基于 Web 的浏览器进行银行交易

B. 基于 Web 的航空预订系统

12. 为何引入线程？线程与进程的关系是什么？

13. 一个航空预订系统并发使用集中式数据库为用户请求提供服务。在此系统中是否最好使用线程而不是进程？解释你的回答。

14. 如果在用户级别实现了线程，给出线程应该避免使用的两个系统调用，并解释你的原因。

15. 评论计算机系统中以下应用程序的计算速度，这些计算机系统具有(i)单个 CPU 和(ii)多个 CPU。

A. 在服务器中创建许多线程，以高速方式处理用户请求，其中用户请求的服务涉及 CPU 和 I/O 行为

B. 表达式 z：＝a＊b＋c＊d 的计算是通过产生两个子进程来计算 a＊b 和 c＊d 来执行的

C. 服务器创建一个新线程来处理收到的每个用户请求，每个用户请求的服务涉及对数据库的访问

D. 两个矩阵有 m 行和 n 列，其中 m 和 n 都非常大。进程通过创建 m 个线程来获得两个矩阵相加的结果，每个线程执行一行矩阵的相加

16. 假设父进程和子进程的进程标识 pid 分别为 2600 和 2603，请给出 A、B、C 和 D 这 4 行的输出结果。

```
#include<sys/types.h>
#include<stdio.h>
#include<unistd.h>
int main()
{
    pid t pid, pid1;
    /* fork a child process */
    pid=fork();
    if (pid<0) {                                    /* error occurred */
        fprintf(stderr, "Fork Failed");
```

```
        return 1;
    }
    else if (pid==0) {                          /* child process */
        pid1=getpid();
        printf("child: pid=%d",pid);            /* A */
        printf("child: pid1=%d",pid1);          /* B */
    }
        else {                                  /* parent process */
            pid1=getpid();
            printf("parent: pid=%d",pid);       /* C */
            printf("parent: pid1=%d",pid1);     /* D */
            wait(NULL);
    }
    return 0;
}
```

17. 注释中的第 X 行和第 Y 行的输出是多少?

```
#include<sys/types.h>
#include<stdio.h>
#include<unistd.h>
#define SIZE 5
int nums[SIZE]={0,1,2,3,4};
int main()
{
    int i;
    pid t pid;
    pid=fork();
    if (pid==0) {
        for (i=0; i<SIZE; i++) {
            nums[i] *=-i;
            printf("CHILD: %d ",nums[i]);       /* X行 */
        }
    }
    else if (pid>0) {
        wait(NULL);
        for (i=0; i<SIZE; i++)
            printf("PARENT: %d ",nums[i]);      /* Y行 */
    }
    return 0;
}
```

18. 多线程进程中的线程之间共享的程序状态组件有(　　)。

A. 寄存器值　　B. 堆内存　　C. 全局变量　　D. 堆栈内存

19. 系统调用通常使用(　　)调用。