

library computing CPU mainframe task web apps desktop program applications OS user interface allocation file system resources storage real-time security command processor memory web server process code update

operating system

河南大学

操作系统

计算机学院

instruction multi-tasking software hardware computer graphical interface CUI management interrupt virtual memory input output device driver multi-user game console supercomputer services microcomputer RAM user job version control networking

library computing CPU mainframe task web apps desktop program applications OS user interface allocation file system resources storage real-time security command processor memory web server process code kernel update



operating system

instruction multi-tasking software hardware computer graphical interface CUI management interrupt virtual memory input output device driver multi-user game console supercomputer services microcomputer RAM user job version control networking

张 帆

教授



13839965397



zhangfan@henu.edu.cn



计算机学院 412 房间



第 2 章

进程的描述与控制



大纲

1 2.1 前趋图和程序执行

2 2.2 进程的描述

3 2.3 进程控制



从进程的观点研究操作系统

- 进程是资源分配和独立运行的基本单位，每一进程都完成一个特定任务。
- 把 OS 看作是由若干个进程和一个可对这些进程进行协调控制的核心（内核）组成。
- OS 的内核则必须要控制和协调进程的运行，解决进程之间的通信，并解决共享资源的竞争问题。



1 2.1 前趋图和程序执行

2 2.2 进程的描述

3 2.3 进程控制



前趋图的定义

- 为了更好地描述程序的顺序和并发执行情况，我们首先介绍用于描述程序执行先后顺序的前趋图 (Precedence Graph)。
- **前趋图**：有向无循环图 (DAG)。描述一个程序的各部分（程序段或语句）间的依赖关系，或者是一个大的计算的各个子任务间的因果（前后）关系。

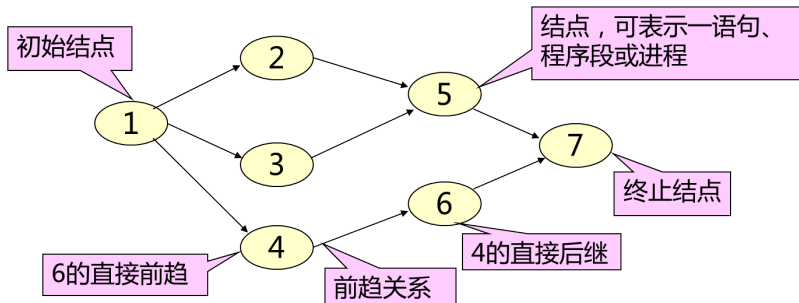


前趋图的定义

- 前趋图中的每个 **结点** 可以表示一条语句、一个程序段或一个进程，结点间的 **有向边** 表示两个结点之间存在的偏序关系或前趋关系 “ \rightarrow ”。
- $\rightarrow = (P_i, P_j) |$ 在 P_j 开始前 P_i 必须完成。如果 $(P_i, P_j) \in \rightarrow$ ，可写成 $P_i \rightarrow P_j$ ， P_i 是 P_j 的直接前趋， P_j 是 P_i 的直接后继。
- 没有前趋的结点称为初始结点，没有后继的结点称为终止结点。此外，每个结点还具有一个 **权** 值，用于表示该结点所含有的程序量或结点的执行时间。



前趋图的定义



前趋关系： $P_1 \rightarrow P_2$, $P_2 \rightarrow P_5$, $P_5 \rightarrow P_7$

$P_1 \rightarrow P_3$, $P_3 \rightarrow P_5$

$P_1 \rightarrow P_4$, $P_4 \rightarrow P_6$, $P_6 \rightarrow P_7$

有向无循环图



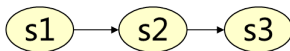
前趋图的定义

以下三条语句的前趋图为：

S1: $a := x + y$

S2: $b := a - 5$

S3: $c := b + 1$

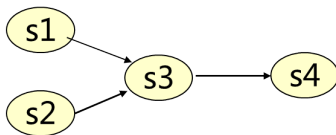


S1 : $a := x + 2$

S2: $b := y + 4$

S3: $c := a + b$

S4: $d := c + 6$



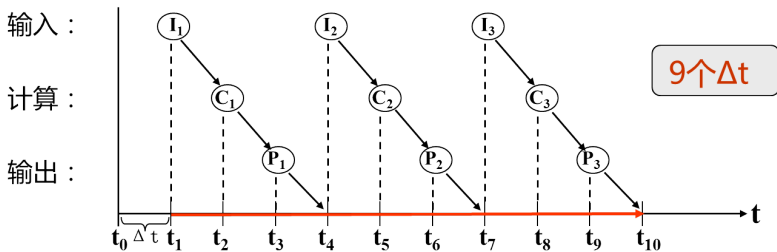
程序顺序执行

- **程序顺序执行**：必须按照某种先后次序逐个执行。
如：数据输入 I \rightarrow 计算 C \rightarrow 数据输出 P
- 程序顺序执行时有如下特征
 - **顺序性**：一个程序各个部分的执行，严格地按照某种先后次序执行。
 - **封闭性**：程序在封闭的环境下运行，即程序运行时独占全部系统资源。
 - **可再现性**：只要程序执行时的环境和初始条件相同，当程序重复执行时，都将获得相同的结果。
- 程序顺序执行的特性为程序员检测和校正程序的错误带来很大方便。



程序顺序执行

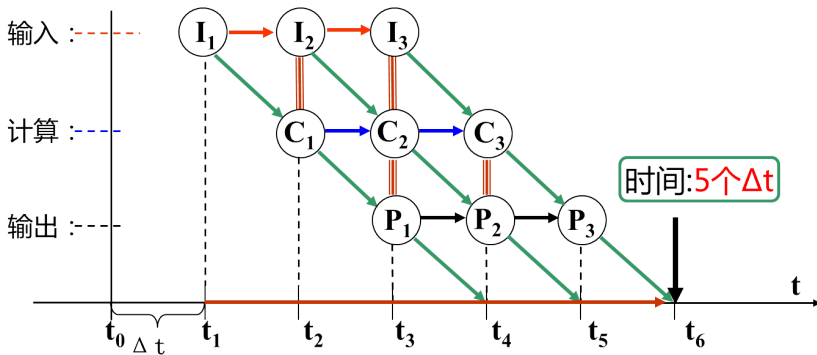
程序1: $I_1 \rightarrow C_1 \rightarrow P_1$ 程序2: $I_2 \rightarrow C_2 \rightarrow P_2$ 程序3: $I_3 \rightarrow C_3 \rightarrow P_3$



三个程序间的顺序执行



程序并发执行



前趋关系： $I_i \rightarrow C_i$, $I_i \rightarrow I_{i+1}$, $C_i \rightarrow P_i$, $C_i \rightarrow C_{i+1}$, $P_i \rightarrow P_{i+1}$



程序并发执行

- **程序的并发执行**：指一组在逻辑上互相独立的程序或程序段在执行时间上客观上互相重叠，即一个程序或程序段的执行尚未结束，另一个程序（段）的执行已经开始的方式。
- 程序并发执行时的特征
 - **间断性（异步性）**：互斥、同步
 - **失去封闭性**：共享资源 → 失去封闭性
 - **不可再现性**：失去封闭性 → 失去可再现性



程序并发执行

- 程序在并发执行时，由于失去了封闭性，也将导致其再失去可再现性。
- 例如，有两个循环程序 A 和 B，它们共享一个变量 N。程序 A 每执行一次时，都要做 $N:=N+1$ 操作；程序 B 每执行一次时，都要执行 $\text{Print}(N)$ 操作，然后再将 N 置成“0”。程序 A 和 B 以不同的速度运行。这样，可能出现下述三种情况（假定某时刻变量 N 的值为 n）。



程序并发执行

1	repeat	repeat
2	$N = N + 1$;	Print N; $N = 0$;
3	until	until



程序并发执行

1	repeat	repeat
2	$N = N + 1$;	Print N; $N = 0$;
3	until	until

- 1 $N:=N+1$ 在 Print(N) 和 $N:=0$ 之前, 此时得到的 N 值分别为 $n+1$, $n+1$, 0 。



程序并发执行

1	repeat	repeat
2	$N = N + 1$;	Print N; $N = 0$;
3	until	until

- 1 $N:=N+1$ 在 Print(N) 和 $N:=0$ 之前, 此时得到的 N 值分别为 $n+1$, $n+1$, 0 。
- 2 $N:=N+1$ 在 Print(N) 和 $N:=0$ 之后, 此时得到的 N 值分别为 n , 0 , 1 。



程序并发执行

1	repeat	repeat
2	$N = N + 1$;	Print N; $N = 0$;
3	until	until

- 1 $N:=N+1$ 在 Print(N) 和 $N:=0$ 之前, 此时得到的 N 值分别为 $n+1$, $n+1$, 0 。
- 2 $N:=N+1$ 在 Print(N) 和 $N:=0$ 之后, 此时得到的 N 值分别为 n , 0 , 1 。
- 3 $N:=N+1$ 在 Print(N) 和 $N:=0$ 之间, 此时得到的 N 值分别为 n , $n+1$, 0 。



程序并发执行

航班座位订票程序

```
var
row, col : integer;
ticket[n][m] : integer;
```

共享数据

row=3 ; col=4

```
...
procedure booking
1 : begin
2 : if row <= n
3 :   begin
4 :     ticket[row][col]:= 1; 赋值1表示已售
5 :     write ( "座位:" + row+ "排" + col + "号" );
6 :     col = col mod m + 1;
7 :     if col = =1
8 :       row= row + 1;
9 :     end
10 : else
11 :   write ( "座位已售完!" );
12 : end
```

	1	2	3	4	...	m
1						
2						
3						
4						
...						
i						
...						
n						

当前位置

程序订下当前座位票，
并指向下一个空座位。



程序并发执行

两个订票机B1、B2
有相同的预定程序
并共享公共数据。

B1订票机

```
1 : begin
2 : if row <= n
3 :   begin
4 :     ticket[row][col]:= 1;
5 :     write ( "座位:" +row+ "排" +col
+ "号" );
...
```

row=3 ;
col=4

B2订票机

```
1 : begin
2 : if row <= n
3 :   begin
4 :     ticket[row][col]:= 1;
5 :     write ( "座位:" +row+ "排" +col
+ "号" );
...
```

row=3 ;
col=4

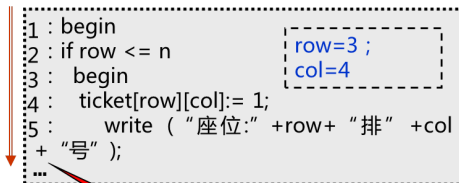


程序并发执行

两个订票机B1、B2
有相同的预定程序
并共享公共数据。

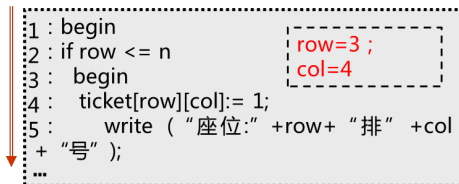
B1订票机

```
1 : begin
2 : if row <= n
3 :   begin
4 :     ticket[row][col]:= 1;
5 :     write ( "座位:" +row+ "排" +col
+ "号" );
...
```



B2订票机

```
1 : begin
2 : if row <= n
3 :   begin
4 :     ticket[row][col]:= 1;
5 :     write ( "座位:" +row+ "排" +col
+ "号" );
...
```



程序并发执行

- 上述情况说明，程序在并发执行时，由于失去了封闭性，其计算结果已与并发程序的执行速度有关，从而使程序的执行失去了可再现性，亦即，程序经过多次执行后，虽然它们执行时的环境和初始条件相同，但得到的结果却各不相同。
- 程序并发执行时的不可再现性是绝对不允许的；因此应采取措施使并发程序保持其 **可再现性**。



程序并发执行

- 上述情况说明，程序在并发执行时，由于失去了封闭性，其计算结果已与并发程序的执行速度有关，从而使程序的执行失去了可再现性，亦即，程序经过多次执行后，虽然它们执行时的环境和初始条件相同，但得到的结果却各不相同。
- 程序并发执行时的不可再现性是绝对不允许的；因此应采取措施使并发程序保持其 **可再现性**。

引入进程，对并发执行的程序加以描述和控制。



程序并发执行条件 (Bernstein 条件)

- 将程序中任一语句 S_i 划分为两个变量的集合 $R(S_i)$ 和 $W(S_i)$, 其中 $R(S_i) = (a_1, a_2, \dots, a_m)$, $W(S_i) = (b_1, \dots, b_n)$ 。
- $R(S_i)$ 是语句 S_i 在执行期间必须对其进行 **参考** 的变量
- $W(S_i)$ 是语句 S_i 在执行期间必须对其进行 **修改** 的变量



程序并发执行条件 (Bernstein 条件)

- 将程序中任一语句 S_i 划分为两个变量的集合 $R(S_i)$ 和 $W(S_i)$, 其中 $R(S_i) = (a_1, a_2, \dots, a_m)$, $W(S_i) = (b_1, \dots, b_n)$ 。
- $R(S_i)$ 是语句 S_i 在执行期间必须对其进行 **参考** 的变量
- $W(S_i)$ 是语句 S_i 在执行期间必须对其进行 **修改** 的变量

$R(S_i)$ 理解为**读集**

$W(S_i)$ 理解为**写集**



程序并发执行条件 (Bernstein 条件)

如果对于语句 S1 和 S2 , 有:

1 $R(S1) \cap W(S2) = \phi$

2 $W(S1) \cap R(S2) = \phi$

3 $W(S1) \cap W(S2) = \phi$

同时成立, 则语句 S1 和 S2 是可以并发执行的。



程序并发执行条件 (Bernstein 条件)

[例] S1:a:=x+2, S2:b:=z+4, S3:c:=a-b, S4:w:=c+1 试利用 Bernstein 条件证明: (1) S1 与 S2 可并发执行; (2) S1 与 S3, S2 与 S3, S3 与 S4 不能并发执行。



程序并发执行条件 (Bernstein 条件)

[例] S1:a:=x+2, S2:b:=z+4, S3:c:=a-b, S4:w:=c+1 试利用 Bernstein 条件证明: (1) S1 与 S2 可并发执行; (2) S1 与 S3, S2 与 S3, S3 与 S4 不能并发执行。

解: 各语句的读、写集分别为:

$R(S1)=x, W(S1)=a, R(S2)=z, W(S2)=b, R(S3)=a,b,$
 $W(S3)=c, R(S4)=c, W(S4)=w$



程序并发执行条件 (Bernstein 条件)

[例] S1:a:=x+2, S2:b:=z+4, S3:c:=a-b, S4:w:=c+1 试利用 Bernstein 条件证明: (1) S1 与 S2 可并发执行; (2) S1 与 S3, S2 与 S3, S3 与 S4 不能并发执行。

解: 各语句的读、写集分别为:

$R(S1)=x, W(S1)=a, R(S2)=z, W(S2)=b, R(S3)=a,b,$
 $W(S3)=c, R(S4)=c, W(S4)=w$

- 1 由 Bernstein 条件, $R(S1) \cap W(S2) = \phi, R(S2) \cap W(S1) = \phi,$
且 $W(S1) \cap W(S2) = \phi$, 所以 S1 与 S2 可并发执行。



程序并发执行条件 (Bernstein 条件)

[例] S1:a:=x+2, S2:b:=z+4, S3:c:=a-b, S4:w:=c+1 试利用 Bernstein 条件证明: (1) S1 与 S2 可并发执行; (2) S1 与 S3, S2 与 S3, S3 与 S4 不能并发执行。

解: 各语句的读、写集分别为:

$R(S1)=x, W(S1)=a, R(S2)=z, W(S2)=b, R(S3)=a,b,$
 $W(S3)=c, R(S4)=c, W(S4)=w$

- 1 由 Bernstein 条件, $R(S1) \cap W(S2) = \phi, R(S2) \cap W(S1) = \phi,$
且 $W(S1) \cap W(S2) = \phi$, 所以 S1 与 S2 可并发执行。
- 2 同理可证 S1 与 S3, S2 与 S3, S3 与 S4 不能并发执行。



1 2.1 前趋图和程序执行

2 2.2 进程的描述

3 2.3 进程控制



进程的定义

- 1 进程是程序的一次执行。
- 2 进程是一个程序及其数据在处理机上顺序执行时所发生的活动。
- 3 进程是程序在一个数据集合上的运行过程，它是系统进行资源分配和调度的一个独立单位。
- 4 引入进程实体的概念后（程序段 + 数据段 + 进程控制块），进程可以定义为：进程是进程实体的运行过程，是系统进行资源分配和调度的一个独立单位。



进程的定义

为了描述和记录进程的运动变化过程，并使之能正确运行，每个进程都应配置一个进程控制块（PCB）。从结构上看，每个进程（进程实体）都是由 **程序段、相关数据段及进程控制块**组成。

- 1 进程实际上是指进程实体。
- 2 创建、撤销进程实际上指创建、撤销进程实体的 PCB。
- 3 PCB 是进程存在的唯一标志。



进程的特征

1 动态性

进程的实质是程序在处理器上的一次执行过程，因此是动态。**动态性是进程的最基本的特征**。同时动态性还表现在进程则是有生命期的，它**因创建而产生，因调度而执行，因得不到资源而暂停，因撤消而消亡**。

2 并发性

指多个进程实体同时存在于内存中，能在一段时间内同时运行。**引入进程的目就是为了使进程能并发执行**，以提高资源利用率，所以并发性是进程的重要特征，也是 OS 的重要特征。



进程的特征

3 独立性

指进程是一个能独立运行的基本单位，也是系统进行资源分配和调度的独立单位。

4 异步性

指进程以各自独立的、不可预知的速度向前推进。



进程与程序的主要区别

- 程序是指令的有序集合，其本身没有任何运行的含义，它是一个**静态**的概念。而进程是程序在处理机上的一次执行过程，它是一个**动态**概念。
- 程序的存在是**永久**的，而进程则是**有生命期**的，它因创建而产生，因调度而执行，因得不到资源而暂停，因撤消而消亡。
- 程序仅是指令的有序集合，而进程则由程序段、相关数据段进程控制块（PCB）组成。
- 进程与程序之间不是一一对应的。



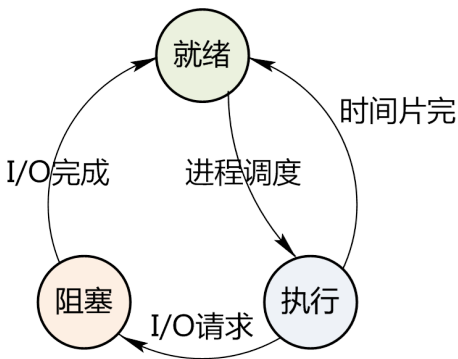
进程的三种基本状态

- **New 新建/ 创建**：进程正在创建中的状态。
- **Ready 就绪**：进程已获得了除处理机以外的所有资源，等待分配处理机执行的等待状态。
- **Running 运行/ 执行**：当一个进程获得必要的资源并正在处理机上执行的状态。
- **Waiting 等待/ 阻塞**：正在执行的进程由于发生某事件而暂时无法执行下去，此时进程所处的状态。
- **Terminated 终止/ 撤消/ 退出**：进程执行完毕，释放所占资源的状态。



进程状态的转换

进程在运行期间并非固定处于某个状态，而是不断从一个状态转换到另一个状态。



进程的三种基本状态

■ 进程的挂起状态

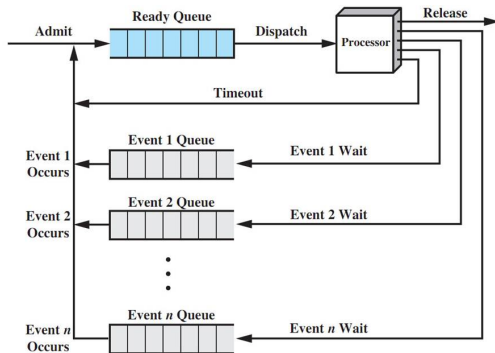
在某些系统中，为了更好地管理和调度进程，引入了挂起状态。

■ 挂起状态/ 静止状态

程序在运行期间，由于某种需要，往往要将进程暂停执行，使其静止下来，以满足其需要。这种静止状态就称为进程的挂起状态。



为什么需要挂起状态



- 处理器的高速使得所有进程都处于阻塞的情况很常见。
- 解决方法：扩大内存、内外存之间的交换。



挂起状态

- 挂起状态使进程置于**静止状态**
- 一般情况下，挂起状态的进程将**从内存移到外存**
- 正在执行的进程暂停执行
- 就绪的进程暂不接受调度
- 阻塞的进程即使阻塞事件释放也不能继续执行

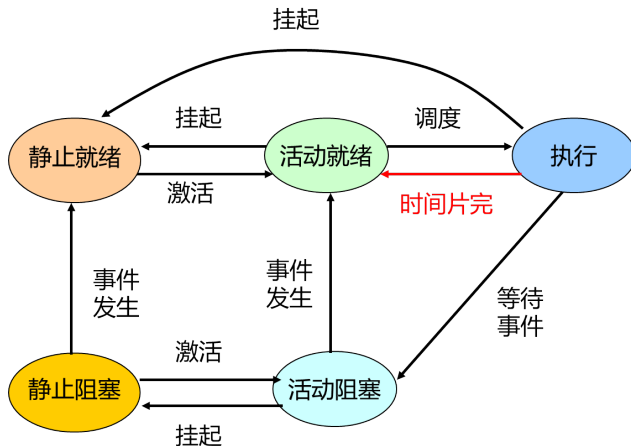


引起挂起状态的原因

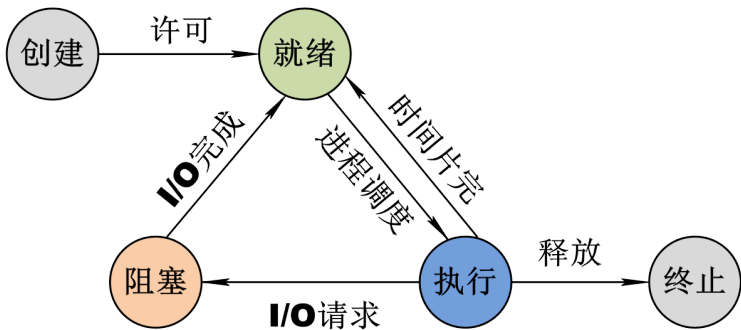
- **终端用户的需要**：终端用户在程序运行中发现问题，要求正在执行的进程暂停执行而使进程处于挂起状态。
- **父进程的需要**：父进程为了考查和修改某个子进程，或者协调各子进程间的活动，需要将该子进程挂起。
- **操作系统的需要**：操作系统为了检查运行中的资源使用情况或进行记帐，而将某些进程挂起。
- **对换的需要**：为了提高内存的利用率，而将内存中某些进程挂起，以调进其它程序运行。
- **负荷调节的需要**：工作负荷较重时，把一些不重要的进程挂起，以保证系统能正常运行（实时操作系统）。



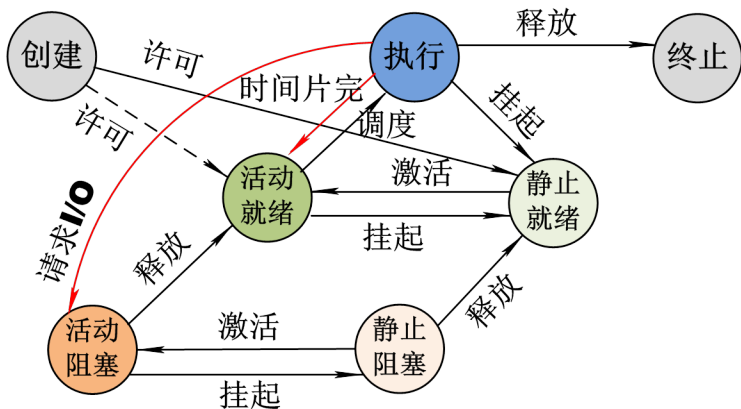
具有挂起状态的进程状态转换



具有创建、终止的进程状态转换



具有创建、终止、挂起状态的进程状态转换



进程控制块 PCB

- PCB 是操作系统为了管理和控制进程的运行，而为每一个进程定义的一个数据结构，它记录了系统管理进程所需的全部信息。
- PCB 常驻内存，存放在操作系统中专门开辟的 PCB 区内。



进程控制块 PCB 的作用

1 作为独立运行基本单位的标志

- 系统根据 PCB 而感知进程的存在，**PCB 是进程存在的唯一标志。**
- 系统创建一个新进程时，就为它建立了一个 PCB；进程结束时又回收其 PCB，进程于是也随之消亡。

2 能实现间断性运行方式

- 利用 PCB 保存处理机状态信息，保护和恢复 CPU 现场。



进程控制块 PCB 的作用

3 提供进程管理所需要的信息

- 根据 PCB 中的程序和数据内存地址，找到程序和数据。
- 系统根据 PCB 了解进程所需的全部资源。

4 提供进程控制所需要的信息

- OS 要调度某进程执行时，从 PCB 中查现行状态及优先级。

5 实现与其它进程的同步和通信



进程控制块中的信息

操作系统不同，PCB 所包含信息有些不同，但通常包含以下四类信息：

1 进程标志符

- 内部标识符。由系统创建进程时分配给进程的唯一标识号，通常为一整数，称为进程号，用于区分不同的进程。
- 外部标识符。用户在访问该进程时使用，称为用户号。设置父进程标识及子进程标识。还可设置用户标识，以指示拥有该进程的用户。



进程控制块中的信息

2 处理机状态（断点信息）

- 通用寄存器：用户程序可以访问，用于暂存信息。
- 指令计数器：程序计数器 PC，存放要访问的下一条指令的地址。
- PSW：程序状态字，其中含有状态信息，如条件码、执行方式、中断屏蔽标志等。
 - 条件码：指令执行完毕后设置状态标志，包括进位标志、溢出标志、符号标志、零标志、奇偶标志等，用于后续指令的控制条件。
 - 执行方式：用户态、核心态
- 用户栈指针，用于存放系统调用参数及调用地址。



进程控制块中的信息

3 进程调度信息

- 进程状态：进程的当前状态（执行、就绪、阻塞）。
- 优先级
- 进程调度所需的信息：如等待时间、已执行时间等。
- 事件：状态转换的原因。

4 进程控制信息

- 程序和数据地址
- 同步和通信机制：消息队列指针、信号量等。
- 资源清单：除 CPU 以，所需资源及已分配资源的清单
- 链接指针：下一个进程的 PCB 的首地址



进程控制块 PCB 的组织方式

在一个系统中通常存在着许多进程，为了方便进程的调度和管理，需要将不同状态的各进程用适当方法组织起来。

1 链接方式

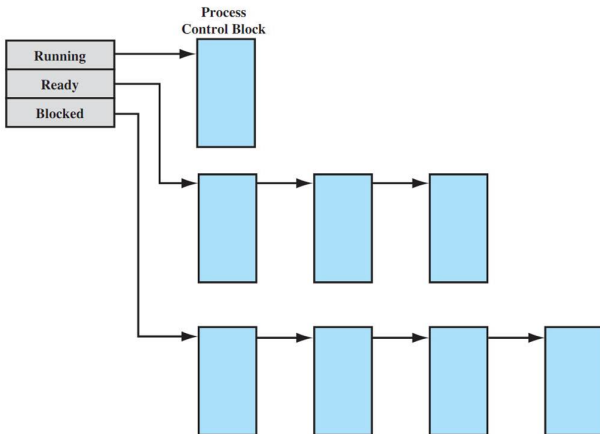
把同一状态的 PCB 链接成一个队列，这样就形成了就绪队列、阻塞队列等。

2 索引方式

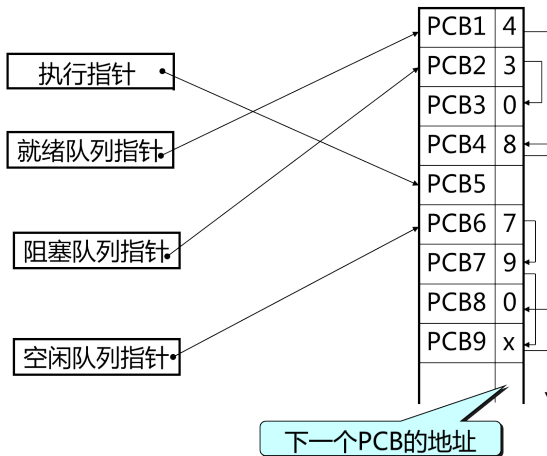
将同一状态的进程组织在一个索引表中，索引表的表项指向相应的 PCB，不同状态对应不同的索引表。



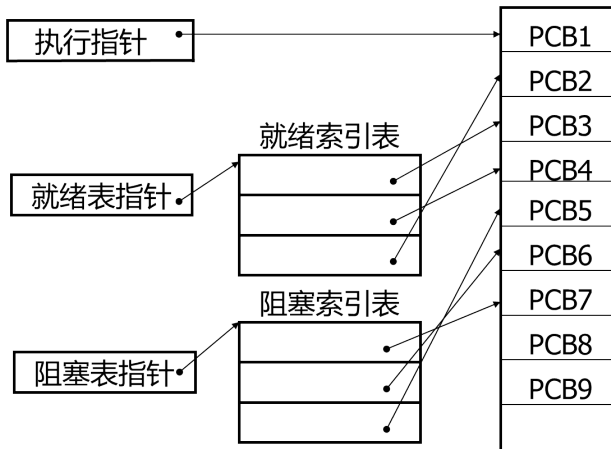
按链接方式组织 PCB



按链接方式组织 PCB



按索引方式组织 PCB



1 2.1 前趋图和程序执行

2 2.2 进程的描述

3 2.3 进程控制



操作系统内核

进程控制是进程管理中最基本的功能，即对系统中所有的进程实施有效的管理，其功能包括 **进程的创建、撤消、阻塞与唤醒** 等，这些功能一般是由 **操作系统的内核** 通过 **原语** 来完成。

操作系统内核：在现代 OS 中，常把一些功能模块（与硬件紧密相关的及运行频率较高的）放在紧靠硬件的软件层次中，加以特殊保护，使它们常驻内存，以提高 OS 的运行效率，这部分功能模块就称 OS 的内核。内核是基于硬件的第一层软件扩充，它为系统控制和管理进程提供了良好的环境。



操作系统内核的功能

1 支撑功能

- 1 中断处理
- 2 时钟管理 (时间片)
- 3 原语操作

2 资源管理功能

- 1 进程管理
- 2 存储器管理
- 3 设备管理



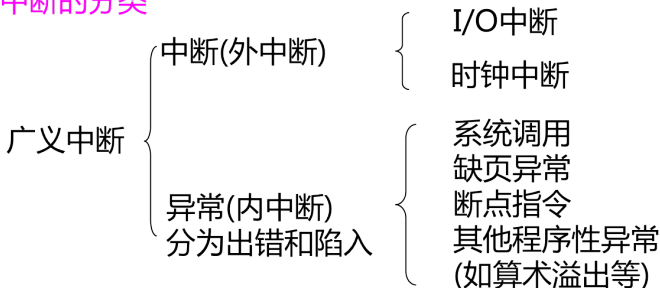
中断处理

- 中断：CPU 对系统中或系统外发生的某个事件作出的一种反应。如外部设备完成数据传输、实时设备出现异常等。
- 引入中断的目的：中断机制是操作系统得以正常工作的最重要的手段，**操作系统是由中断驱动的**。
- 中断可以解决
 - 主机与外设的并行工作问题
 - 提高可靠性
 - 实现实时控制
 - 中断是实现多道程序的必要条件



中断处理

中断的分类



中断处理过程

保存现场 → 转到中断处理程序 → 恢复现场

把程序计数器置为中断处理程序的开始地址

把处理机模式从用户态切换到核心态（中断处理可能要执行特权指令）



原语操作

- **原语 (Primitive)**：由若干条指令组成的，用于完成一定功能的一个过程。它与一般过程的区别在于：它们是“原子操作 (Atomic Operation)”。
- **原子操作**：原子操作，是指一个操作中的所有动作要么全做，要么全不做。换言之，它是一个不可分割的基本单位，因此，在执行过程中不允许被中断。
- 原语在 **核心态**下执行，常驻内存。
- 例如：Suspend 挂起原语；Active 激活原语
- 原语的作用是实现进程的控制和通信，在进程控制中如果不使用原语，会造成进程状态的不确定性。



处理机的执行状态

为防止 OS 及其关键数据（如 PCB 等）不被用户有意或无意破坏，通常将处理机的执行状态分为两种：

处理机状态	执行指令及访问权限
核心态	一切指令, 所有寄存器及存储区
用户态	规定指令, 指定寄存器及存储区

这两种状态由一位触发器标识，通常属于 **程序状态字** 的一部分，即由 PSW 中的一位标识。



处理机的执行状态

- 由于利用特权指令可以修改程序状态字，而机器状态是 PSW 的一部分，因而在核心态下可以改变机器状态，由核心态转换为用户态。
- 由于“置程序状态字”为特权指令，用户态程序不能将其运行状态改为核心态，这样就可防止用户有意或无意地侵入系统。
- 处理机状态由用户态转为核心态的唯一途径是**中断**。
- **访管指令**：在用户态下执行的指令，当处理器执行到访管指令时就产生一个**中断**，暂停用户程序的执行，而让操作系统来为用户服务。它本身不是特权指令。



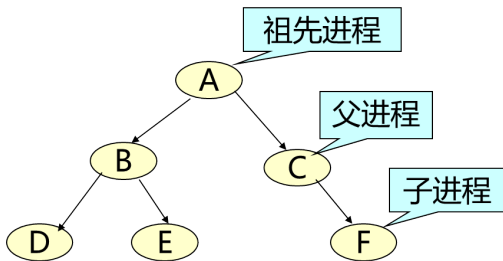
进程的层次结构

- 进程的家族关系
 - 在每个进程的 PCB 中都设置家族关系的表项，以表明自己的父进程和所有的子进程。
- 了解进程家族关系的必要性
 - 子进程可继承父进程的所有资源。
 - 子进程撤销时要把资源归还给父进程。
 - 父进程撤销时也必须撤销所有子进程。



进程图

- 一个进程可以创建若干个新进程，新创建的进程又可以创建子进程。
- **进程图**：又称为进程树或进程家族树，是描述进程家族关系的一棵有向树。



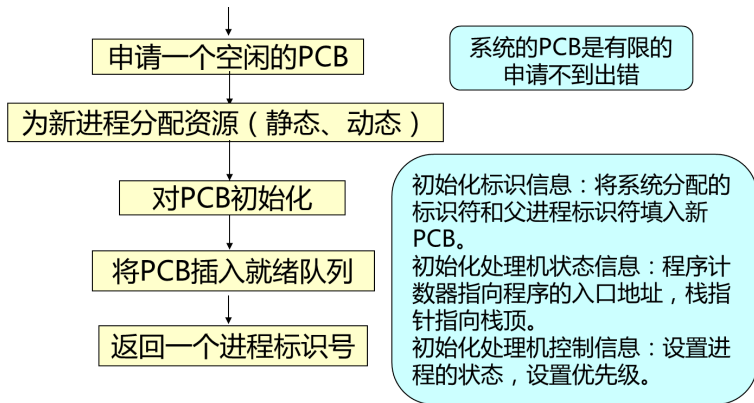
引起进程创建的事件

- 在多道程序环境中，只有进程才可以在系统中运行。为了使一个程序能运行，必须为它创建进程。
- 导致进程创建的事件
 - 用户登录：为该终端用户创建一个进程
 - 作业调度：为被调度的作业创建进程
 - 提供服务：如要打印时建立打印进程
 - 应用请求：由应用程序建立多个进程（并发）



进程的创建

- 操作系统一旦发现了要求创建进程的事件后，便调用进程 **创建原语 Create** 按以下过程创建新进程。

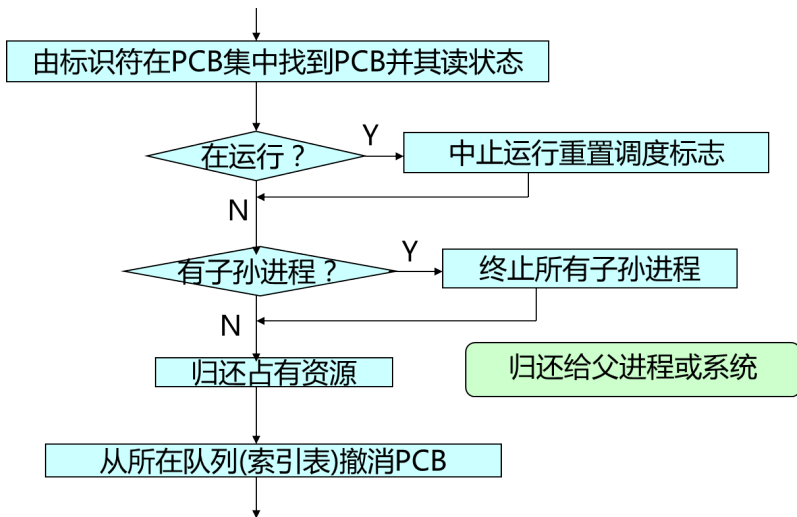


进程的终止

- 一个进程在完成其任务后，应加以撤消，以便及时释放其占有的各类资源。
- 导致进程终止的事件
 - 进程正常结束（Holt、Logoff 等）
 - 进程异常结束（越界、超时、非法指令、I/O 故障等）
 - 外界干预（操作员、OS、父进程）
- 如果系统中发生了要求撤消进程的事件，OS 便调用 **撤消原语 Kill** 去撤消进程。（Destroy、Termination）



进程的终止

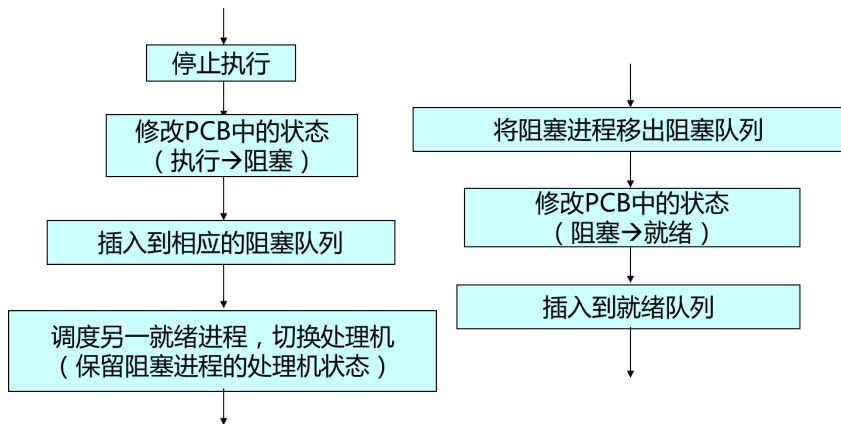


进程的阻塞与唤醒

- 当进程期待的事件尚未出现时，该进程调用**阻塞原语 Block**（由执行状态转为阻塞状态）把自己阻塞起来。
- 处于阻塞状态的进程，当期待的事件出现时，由其它相关进程调用**唤醒原语 Wakeup**（由阻塞状态变为就绪状态）把阻塞的进程唤醒，使其进入就绪状态。
- 导致进程阻塞和唤醒的事件
 - 向系统请求共享资源失败
 - 等待某种操作的完成
 - 新数据尚未到达
 - 无新工作可做



进程的阻塞与唤醒



进程的阻塞与唤醒

- 进程从执行态到阻塞态是 **主动** 的。进程发现需要等待某一事件，主动向系统申请进入阻塞态。
- 进程从阻塞态到就绪态是 **被动** 的。当系统（或其它进程）发现阻塞进程阻塞的条件已释放，向系统申请将该进程置为就绪态。
- Block 原语和 Wakeup 原语是一对作用刚好相反的原语。如果在某进程中调用了阻塞原语，必须在与之相合作的另一进程中或其他相关的进程中安排唤醒原语，以能唤醒阻塞进程。



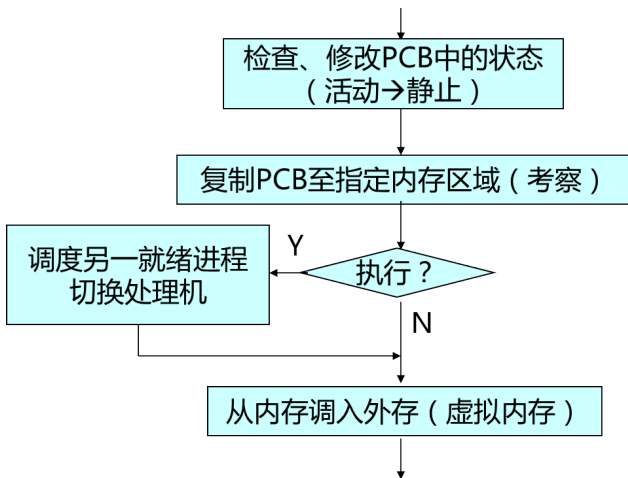
进程的挂起与激活

- 当引起进程挂起的事件发生时，系统就将利用挂起原语 **Suspend** 将指定进程挂起。（三种基本状态均可能挂起）
- 当发生激活进程的事件时，系统就将利用激活原语 **Active** 将指定进程激活。

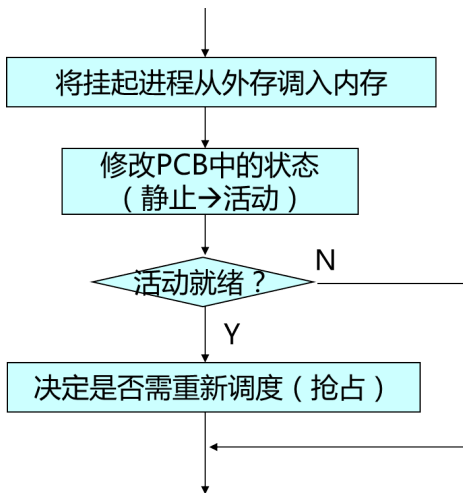
挂起是由进程自己或其父进程调 Suspend 原语完成。激活是由父进程或用户进程请求激活指定进程，系统利用 Active 原语将指定进程激活。



进程的挂起



进程的激活



课堂练习

下列哪些情况是错误的？

- 进程自己创建自己



课堂练习

下列哪些情况是错误的？

- 进程自己创建自己 × 由父进程、OS、操作员创建
- 进程自己阻塞自己



课堂练习

下列哪些情况是错误的？

- 进程自己创建自己 × 由父进程、OS、操作员创建
- 进程自己阻塞自己 ✓
- 进程自己挂起自己



课堂练习

下列哪些情况是错误的？

- 进程自己创建自己 × 由父进程、OS、操作员创建
- 进程自己阻塞自己 ✓
- 进程自己挂起自己 ✓ 也可由父进程、OS 挂起
- 进程自己激活自己



课堂练习

下列哪些情况是错误的？

- 进程自己创建自己 × 由父进程、OS、操作员创建
- 进程自己阻塞自己 ✓
- 进程自己挂起自己 ✓ 也可由父进程、OS 挂起
- 进程自己激活自己 × 由其它进程激活
- 进程自己唤醒自己



课堂练习

下列哪些情况是错误的？

- 进程自己创建自己 × 由父进程、OS、操作员创建
- 进程自己阻塞自己 ✓
- 进程自己挂起自己 ✓ 也可由父进程、OS 挂起
- 进程自己激活自己 × 由其它进程激活
- 进程自己唤醒自己 × 由其它进程唤醒

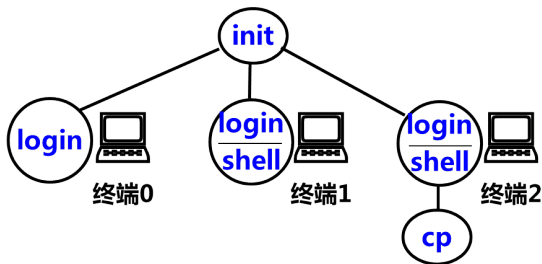


UNIX 的启动过程

- 0 号进程：核心程序在做完自身的初始化工作后建立系统的第一个进程。它的功能是完成 UNIX 系统的进程调度，始终运行在系统的核心态。
- 1 号进程：系统初启后，由 0 号进程创建 init 进程。init 进程为根进程，所有进程都是它的子进程。运行在用户态。
- 当有终端请求注册时，init 进程就为该用户创建一个 login 进程。
- 若用户注册成功，则 login 进程就为该用户再创建一个 Shell 进程。



UNIX 的启动过程



- 终端 1 上的子进程已成功登录了用户，正运行 shell 等待命令输入。
- 在终端 2 上的子进程也成功登录了用户，该用户正运行 cp 程序，cp 是 shell 的子进程，shell 则等待该进程结束。cp 结束后，shell 再接收输入，创建新进程执行新输入命令。



Shell

Shell 俗称壳（用来区别于核），是指提供使用界面的软件（**命令解析器**+**命令语言**+**程序设计语言**）。

- **命令解析器**：Shell 可对输入的命令解释执行。
- **命令语言**：拥有自己内建的 Shell 命令集，为用户提供使用操作系统的接口。
- **程序设计语言**：Shell 程序设计语言支持绝大多数在高级语言中能见到的程序元素，如函数、变量、数组和程序控制结构。

用户成功登录后系统执行一个 Shell 程序。普通用户用“\$”作提示符，超级用户（root）用“#”作提示符。



系统调用

- **系统调用**是内核提供的功能十分强大的一系列的函数，是应用程序和操作系统内核之间的功能接口。
- 用户进程一般不能访问内核，既不能访问内核所占内存空间也不能调用内核函数。 **解决方案：系统调用。**
- 系统调用的原理是进程先用适当的值填充寄存器，然后调用一个特殊的指令，这个指令会跳到一个事先定义的内核中的一个位置（由中断实现）。
- 系统调用和普通的函数调用的区别
 - 系统调用由操作系统核心提供，运行于核心态
 - 普通函数调用由函数库或用户自己提供，运行于用户态



进程控制

UNIX 系统向用户提供了一组系统调用，用户可利用这些系统调用来实现对进程的控制。

系统调用	功能
------	----

Fork()	创建一新进程
--------	--------

Exec()	改变进程的原有代码
--------	-----------

Exit()	实现进程的自我终止
--------	-----------

Wait()	挂起进程，等待子进程终止
--------	--------------

Getpid()	获取进程标识符
----------	---------

Nice()	改变进程的优先级
--------	----------



Fork 系统调用

进程由 fork 函数创建，在 unistd.h 库中定义如下

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

返回值

- 1 >0: 表示系统将执行父进程的程序段。
- 2 =0: 表示系统将执行子进程的有关程序段。
- 3 -1: 表示子进程未创建成功。

pid_t 类型：进程 ID(标识符) 的类型，与 int 类型完全兼容。



Fork 系统调用

- 被 fork 创建的新进程叫做子进程。
- fork 调用的奇妙之处在于它仅被调用一次，却能够返回两次，可能有三种不同的返回值。
- 在子进程中返回 0，在父进程中返回子进程的 pid。
- 在父进程中要返回子进程的 pid 的原因是父进程通过这个返回的子进程 ID 来跟踪子进程。
- 对子进程而言，只有一个父进程，故返回 0。
- 父进程的 ID 可以通过 getpid 取得。



Fork 系统调用

例：(假定创建子进程标识符为 1000)

```
1  #include <stdio.h>
2  main()
3  {
4      pid_t val;
5      while ((val=fork())!=-1);          /* 若失败则反复创建 */
6      printf("val=", val);              /* 打印进程标识符 */
7      if(val!=0)                          /* val>0,在父进程中 */
8          printf("It is a parent process!");
9      else                                /* val=0,在子进程中 */
10         printf("It is a child process!");
11 }
```



Fork 系统调用

运行结果

```
val=0  
It is a child process!  
val=1000  
It is a parent process!
```



Fork 系统调用

- 在语句 `val=fork()` 之前，只有一个进程在执行这段代码，但在这条语句之后，就变成**两个进程**在执行了，这两个进程的代码部分完全相同，将要执行的下一条语句都是

```
printf( "val=" , val )
```

- 两个进程中，原先就存在的那个被称作“父进程”，新出现的那个被称作“子进程”。



Fork 系统调用

- 在调用 `fork()` 之后，父进程和子进程均在下一条语句上继续运行。
- 父、子进程的 `fork()`返回值不同
 - 在子进程中返回时，`pid` 为 0；
 - 在父进程中返回时，`pid` 为所创建的子进程的标识。
- `fork` 出错可能有两种原因
 - 当前的进程数已经达到了系统规定的上限；
 - 系统内存不足。



父进程

```
main()
{
    pid_t val;
    printf( "PID..." );
    val=fork();
    if(val!=0)
        printf( "parent..." );
    else
        printf( "child..." );
}
```

假定不考虑创建失败的情况



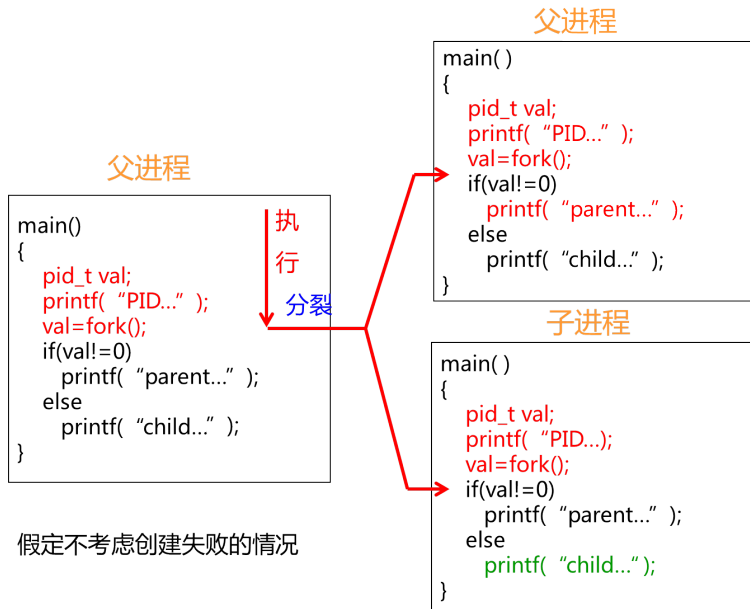
父进程

```
main()
{
    pid_t val;
    printf( "PID..." );
    val=fork();
    if(val!=0)
        printf( "parent..." );
    else
        printf( "child..." );
}
```

执行
↓ 分裂

假定不考虑创建失败的情况





父进程

```
main()
{
    pid_t val;
    printf( "PID..." );
    val=fork();
    if(val!=0)
        printf( "parent..." );
    else
        printf( "child..." );
}
```

继续执行

子进程

```
main()
{
    pid_t val;
    printf( "PID...");
    val=fork();
    if(val!=0)
        printf( "parent..." );
    else
        printf( "child..." );
}
```

继续执行

父进程

```
main()
{
    pid_t val;
    printf( "PID..." );
    val=fork();
    if(val!=0)
        printf( "parent..." );
    else
        printf( "child..." );
}
```

执行

分裂

假定不考虑创建失败的情况



Fork 系统调用

- fork 创建一个新进程（子进程），除了子进程标识符和某些特性参数不同之外，子进程是父进程的**精确复制**。
- 子进程和父进程都执行在 fork 函数调用之后的代码，子进程是父进程的一个拷贝。例如，父进程的数据空间、堆栈空间都会给子进程一个**拷贝，而不是共享**这些内存。
- 父、子进程的运行是无关的，所以运行顺序也不固定。



Fork 系统调用

```
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>
/* fork 系统调用的一个简单例子（不含出错检查） */
```

```
void main(void)
{ printf( "Hello" );
  fork();
  printf( "Bye" );
}
```



Fork 系统调用

```
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>
/* fork系统调用的一个简单例子（不含出错检查） */
```

```
void main(void)
{ printf( "Hello" );
  fork();
  printf( "Bye" );
}
```

运行结果

Hello
Bye
Bye



Fork 系统调用

```
1 void main( void ) {  
2     if ( fork () == 0 )  
3         printf ( "In the CHILD process" );  
4     else  
5         printf ( "In the PARENT process" );  
6 }
```

- 程序的运行无法保证输出顺序。
- 有时语句 “In the CHILD process” 会在 “In the PARENT process” 之前，有时却相反。
- 输出顺序依赖于内核所用的调度算法。



```
1  pid_t pid;
2  int count=0;
3  pid = fork();          /*假定创建的子进程标识符为 3512*/
4  printf("This is the first time, pid = ", pid );
5  printf("This is the second time, pid = ", pid );
6  count++;
7  printf("count = ", count );
8  if ( pid>0 )
9      { printf("This is the parent process, the child has the pid: "
10             , pid ); }
11 else if (pid==0 )
12     { printf("This is the child process."); }
13 else
14     { printf("fork failed." ); }
15 printf("This is the third time, pid = ", pid );
16 printf("This is the fourth time, pid = ", pid );
17 return 0;
```



运行结果

```
This is the first time , pid = 0
This is the second time , pid = 0
count = 1
This is the child process .
This is the third time , pid = 0
This is the fourth time , pid = 0
This is the first time , pid = 3512
This is the second time , pid = 3512
count = 1
This is the parent process , the child has the pid: 3512
This is the third time , pid = 3512
This is the fourth time , pid = 3512
```

为什么 “count++;” 语句只执行了一次？



Fork 系统调用

- 子进程的数据和堆栈空间和父进程是独立的，而不是共享数据。
- 在子进程中对 count 进行自加 1 的操作，但是并没有影响到父进程中的 count 值，父进程中的 count 值仍然为 0。




```
1  #include <stdio.h>
2  main()
3  {
4  int p1,p2;
5  while ((p1=fork())!=-1);           // 创建子进程p1
6  if (p1==0)                         // 子进程创建成功p1
7      putchar( 'b' );
8  else
9      {
10     while ((p2=fork())!=-1);       // 创建子进程p2
11     if (p2==0)                     // 子进程创建成功p2
12         putchar( 'c' );
13     else
14         putchar( 'a' );           // 父进程执行
15     }
16 }
```



Fork 系统调用

运行结果：bca（有时会出现 bac）

分析：进程并发执行，输出 cab，acb 等情况都有可能。



Fork 系统调用

运行结果：bca（有时会出现 bac）

分析：进程并发执行，输出 cab，acb 等情况都有可能。

原因：fork() 创建进程所需的时间要多于输出一个字符的时间，因此在主进程创建进程 2 的同时，进程 1 就输出了“b”，而进程 2 和主程序的输出次序是有随机性的，所以会出现上述结果。



Exec 系统调用

- 利用 fork 系统调用创建一个新进程时，只是将父进程的用户级上下文拷贝到新建的子进程中，子进程是父进程的**精确复制**。子进程如何执行一个新的程序文本？
- 通过一个**系统调用 exec**，子进程可以拥有自己的可执行代码。
- exec 根据指定的文件名找到可执行文件，并用它来取代调用进程的内容，用一个新进程覆盖调用进程。换句话说，将一个可执行的二进制文件覆盖在新进程的用户级上下文的存储空间上，以更新新进程的用户级上下文。



Exec 系统调用

- 它的参数包括新进程对应的文件和命令行参数。成功调用时，不再返回；否则，返回出错原因。
- 在大多数程序中，**系统调用 fork 和 exec 是结合在一起使用的**。父进程生成一个子进程，然后通过调用 exec 覆盖该子进程。
- Linux 中实际上并不存在一个 exec() 的函数形式，exec 指的是一组函数，共 6 个，其中只有 execve () 是真正意义上的系统调用。
- 调用 exec 时可采用"写时拷贝 (copy-on-write)"技术，使得 fork 结束后并不立刻复制父进程的内容，而是到了真正使用的时候才复制。



Exit 系统调用

原型

```
#include <stdlib.h>
void exit(int status);
```

- `exit()` 系统调用是用来终止一个进程的。无论在程序中的什么位置，只要执行到 `exit` 系统调用，进程就会停止剩下的所有操作，清除包括 PCB 在内的各种数据结构，并终止本进程的运行。
- `exit` 系统调用带有一个整数类型的参数 `status`，可以利用这个参数传递进程结束时的状态。0 表示正常结束。其他的数值表示出现了错误，进程非正常结束。



Exit 系统调用

```
1 #include <stdlib.h>
2 main()
3 {
4     printf("this process will exit!");
5     exit(0);
6     printf("never be displayed!");
7 }
```

- 运行结果: `this process will exit!`
- 程序并没有打印后面的"never be displayed! ", 在执行到 `exit(0)` 时, 进程就已经终止了。



Exit 系统调用

- 系统调用 `exit` 的作用是使进程退出，但也仅仅限于将一个正常的进程变成一个**僵尸进程**(zombie)，并不能将其完全销毁。
- 僵尸进程中保存着对程序员和系统管理员非常重要的信息，如进程死亡原因、占用的 CPU 时间等。
- 僵尸进程对其他进程几乎没有什么影响，不占用 CPU 时间，消耗的内存也几乎可以忽略不计，如果存在太多的僵尸进程，也会影响到新进程的产生。



Wait 系统调用

原型

```
#include  
pid_t wait(int *status);
```

- 进程一旦调用了 wait，就立即阻塞自己，由 wait 自动分析是否当前进程的某个子进程已经退出。
- 如果让它找到了一个已经变成僵尸的子进程，wait 就会收集这个子进程的信息，并把它彻底销毁后返回。
- 如果没有找到这样一个子进程，wait 就会一直阻塞在这里，直到有一个出现为止。



Wait 系统调用

- 参数 status 用来保存被收集进程退出时的一些状态，它是一个指向 int 类型的指针。
- 如果对这个子进程是如何死掉的毫不在意，只想把这个僵尸进程消灭掉，可以设定这个参数为 NULL：

```
pid = wait(NULL);
```

- 如果成功，wait 调用会返回被收集的僵尸子进程的进程 ID。
- 如果当前进程没有僵尸子进程，就一直 wait。



```
1  main()
2  pid_t pc, pr;
3  pc=fork();
4  if (pc<0)                /* 如果出错 */
5      printf ("error occurred!");
6  else if (pc==0)          /* 如果是子进程 */
7      {
8          printf ("This is child process with pid of"
9                  ,getpid());
9          sleep(10);        /* 睡眠 */
10         exit(0);
11     }
12     else {                /* 如果是父进程 */
13         pr=wait (NULL);   /* 在这里等待 */
14         printf ("I catch a child process with pid
15                 of", pr);
16     }
```



Wait 系统调用

- 运行结果：

```
This is child process with pid of 1507  
I catch a child process with pid of 1507
```

- 只有子进程从睡眠中苏醒过来，它才能正常退出，也就能被父进程捕捉到。所以在第 2 行结果打印出来前，应该会有 10 秒钟的等待时间，这就是子进程睡眠的时间。
- 不管设定子进程睡眠的时间有多长，父进程都会一直等待下去。



Wait 系统调用

- 利用 wait 系统调用可以解决进程**同步问题**。
- **同步问题**就是要协调好两个以上的进程，使之以安排好的次序依次执行。
- fork 调用成功后，父子进程各自执行，当父进程的工作需要用到子进程的结果时，可以停下来调用 wait，一直等到子进程运行结束，然后利用子进程的结果继续执行。



```
1  pid_t pc, pr;
2  int status;
3  pc=fork();
4  if (pc<0)
5      printf ("Error occurred on forking.");
6  else if (pc==0)
7      {
8          ...                               /* 子进程的工作 */
9          exit(0);
10     }
11     else{
12         ...                               /* 父进程的工作 */
13         pr=wait (&status);             /* &: 取地址运算符 */
14         ...                               /* 利用子进程的结果 */
15     }
```



Waitpid 系统调用

Waitpid () : 等待由参数 pid 指定的子进程退出。

```
#include  
pid_t waitpid(pid_t pid, int *status, int options);
```

- options 提供了一些额外的选项来控制 waitpid。
- 如果不想使用选项，也可以把 options 设为 0。
- 如果使用了选项 WNOHANG 参数调用 waitpid，即使没有子进程退出，也会立即返回，不会像 wait 那样永远等下去。



Waitpid 系统调用

- waitpid 的返回值共有 3 种情况
 - 当正常返回的时候，waitpid 返回收集到的子进程的进程 ID；
 - 如果设置了选项 WNOHANG，而调用中 waitpid 发现没有已退出的子进程可收集，则返回 0；
 - 如果调用中出错，则返回 -1。




```

1  pid_t pc, pr;
2  pc=fork();
3  if(pc<0)                                /* 如果出错.. */
4      printf("Error occured on forking. ");
5  else if (pc==0){                         /* 如果是子进程 */
6      sleep(10);                          /* 睡眠 */
7      exit(0);
8  }
9  else{                                    /* 如果是父进程 */
10     do{
11         pr=waitpid(pc, NULL, WNOHANG);    /* 不会在这里等待 */
12         if(pr==0){                       /* 如果没有收集到子进程 */
13             printf("No child exited");
14             sleep(1);
15         }
16     }while(pr==0);                       /* 没收集到子进程就继续尝试 */
17     if(pr==pc)
18         printf("successfully get child", pr);
19     else
20         printf("some error occurred");
21 }

```



Waitpid 系统调用

运行结果

```
No child exited  
No child exited  
No child exited  
No child exited  
No child exited  
No child exited  
No child exited  
No child exited  
No child exited  
No child exited  
successfully get child 1526
```

- 只有子进程从睡眠中苏醒过来，它才能正常退出，也就才能被父进程捕捉到。
- 父进程经过 10 次失败的尝试之后，终于收集到了退出的子进程。
- 父进程和子进程分别睡眠了 10 秒钟和 1 秒钟，也可以让它们分别做 10 秒钟和 1 秒钟的工作。



THE END

School of Computer & Information Engineering

Henan University

Kaifeng, Henan Province

475001

China

