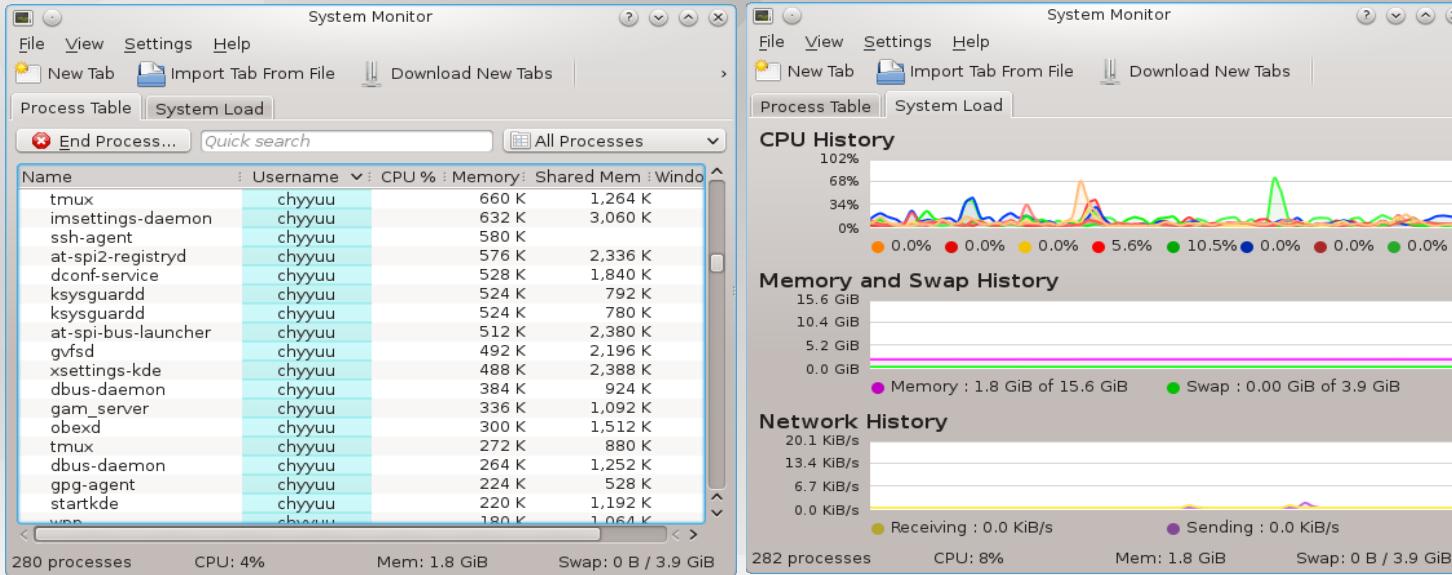


# 实际操作系统中的进程



进程和程序是什么样的关系?

# 进程的定义

进程是指一个具有一定独立功能的程序在一个数据集合上的一次动态执行过程

```
void X (int b) {  
    if(b == 1) {  
        ...  
    }  
  
    int main() {  
        int a = 2;  
        X(a);  
    }  
}
```

编译链接



加载



源代码文件

可执行文件

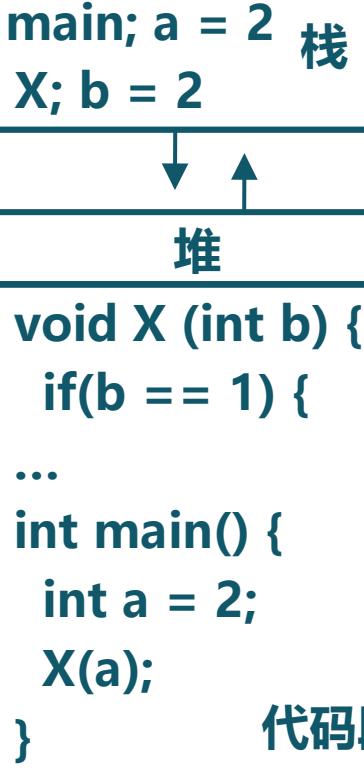
进程地址空间

# 内存中的进程

## 程序源代码

```
void X (int b) {  
    if(b == 1) {  
        ...  
    }  
  
    int main() {  
        int a = 2;  
        X(a);  
    }  
}
```

## 内存中进程



# 进程的组成

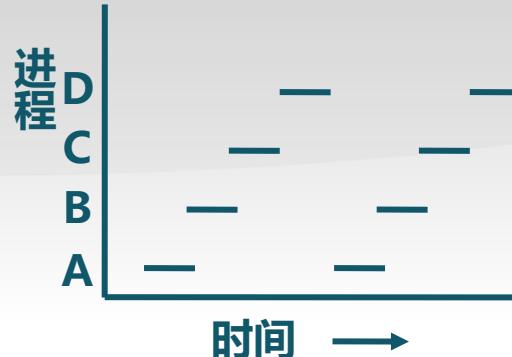
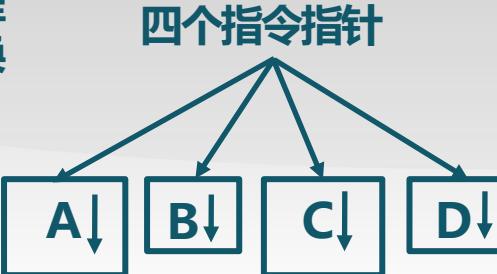
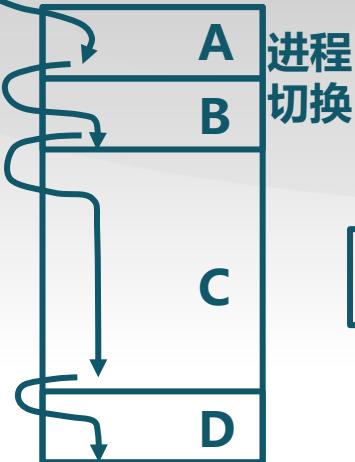
进程包含了正在运行的一个程序的**所有状态信息**

- 代码
- 数据
- 状态寄存器
  - ▶ CPU状态CR0、指令指针IP
- 通用寄存器
  - ▶ AX、 BX、 CX...
- 进程占用系统资源
  - ▶ 打开文件、已分配内存...

# 进程的特点

- 动态性
- 并发地创建、结束进程
- 独立可以被独立调度并占用处理机运行
- 制约进程的工作不相互影响

进程执行过程  
因访问共享数据/资源或进程间同步而产生制约



# 进程与程序的联系

- 进程是操作系统处于执行状态程序的抽象
  - ▶ 程序 = 文件 (静态的可执行文件)
  - ▶ 进程 = 执行中的程序 = 程序 + 执行状态
- 同一个程序的多次执行过程对应为不同进程
  - ▶ 如命令 “ls” 的多次执行对应多个进程
- 进程执行需要的资源
  - ▶ 内存：保存代码和数据
  - ▶ CPU：执行指令

# 进程与程序的区别

- 进程是动态的，程序是静态的
  - ▶ 程序是有序代码的集合
  - ▶ 进程是程序的执行，进程有核心态/用户态
- 进程是暂时的，程序的永久的
  - ▶ 进程是一个状态变化的过程
  - ▶ 程序可长久保存
- 进程与程序的组成不同
  - ▶ 进程的组成包括程序、数据和进程控制块



# 操作系统

Operating System

# 进程控制块 (PCB, Process Control Block)

操作系统管理控制进程运行所用的信息集合

- 操作系统用PCB来描述进程的基本情况以及运行变化的过程
- PCB是进程存在的唯一标志
  - ▶ 每个进程都在操作系统中有一个对应的PCB

# 进程控制块的使用

- 进程创建
  - ▶ 生成该进程的PCB
- 进程终止
  - ▶ 回收它的PCB
- 进程的组织管理
  - ▶ 通过对PCB的组织管理来实现

PCB具体包含什么信息？如何组织的？

进程的状态转换.....?

# 进程控制块



- 进程标识信息
- 处理机现场保存
- 进程控制信息

# 进程控制信息

- 调度和状态信息
  - ▶ 调度进程和处理机使用情况
- 进程间通信信息
  - ▶ 进程间通信相关的各种标识
- 存储管理信息
  - ▶ 指向进程映像存储空间数据结构
- 进程所用资源
  - ▶ 进程使用的系统资源，如打开文件等
- 有关数据结构连接信息
  - ▶ 与PCB相关的进程队列

# 进程控制块的组织

## ■ 链表

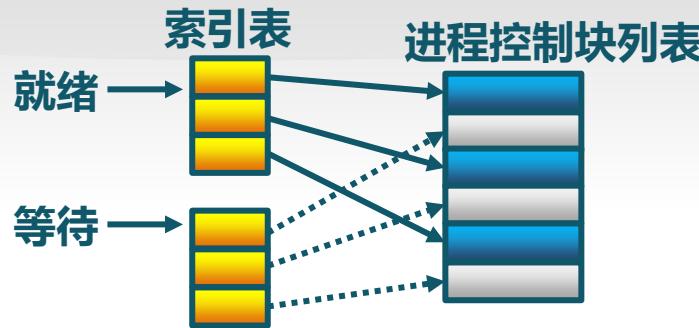
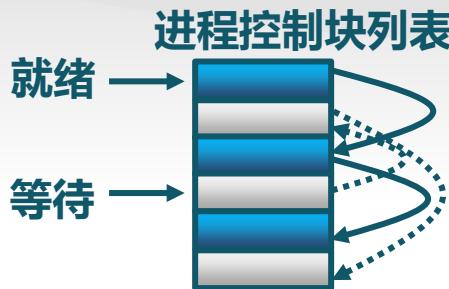
同一状态的进程其PCB成一链表，多个状态对应多个不同的链表

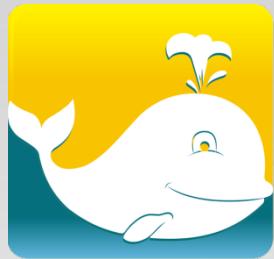
► 各状态的进程形成不同的链表：就绪链表、阻塞链表

## ■ 索引表

同一状态的进程归入一个索引表（由索引指向PCB），  
多个状态对应多个不同的索引表

► 各状态的进程形成不同的索引表：就绪索引表、阻塞索引表





# 操作系统

Operating System

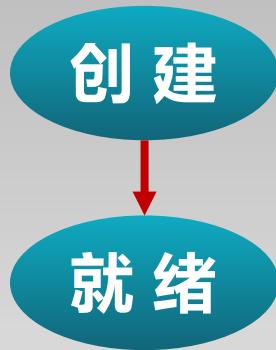
# 进程的生命周期划分

- 进程创建
- 进程执行
- 进程等待
- 进程抢占
- 进程唤醒
- 进程结束

# 进程创建

## 引起进程创建的情况

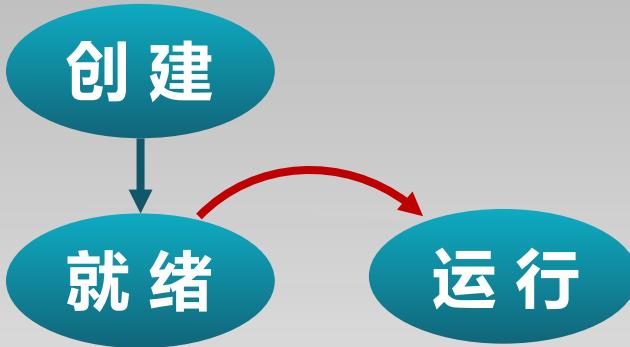
- 系统初始化时
- 用户请求创建一个新进程
- 正在运行的进程执行了创建进程的系统调用



# 进程执行

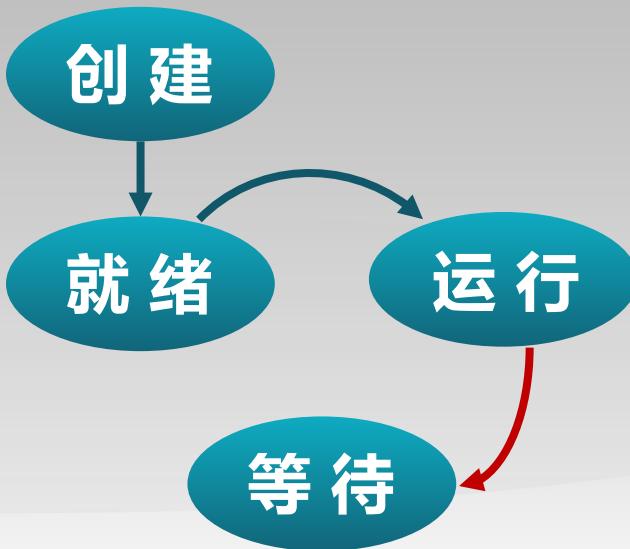
内核选择一个就绪的进程，让它  
占用处理器并执行

- 如何选择？



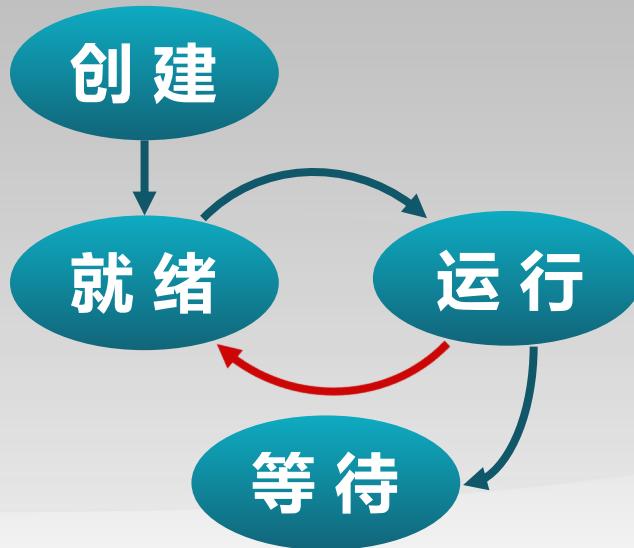
# 进程等待

- 进程进入等待(阻塞)的情况：
  - ▶ 请求并等待系统服务，无法马上完成
  - ▶ 启动某种操作，无法马上完成
  - ▶ 需要的数据没有到达
- 只有进程自身才能知道何时需要等待某种事件的发生



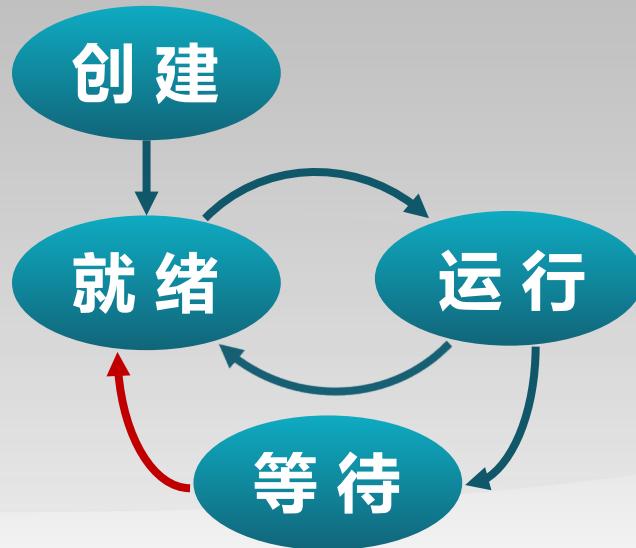
# 进程抢占

- 进程会被抢占的情况
  - ▶ 高优先级进程就绪
  - ▶ 进程执行当前时间用完



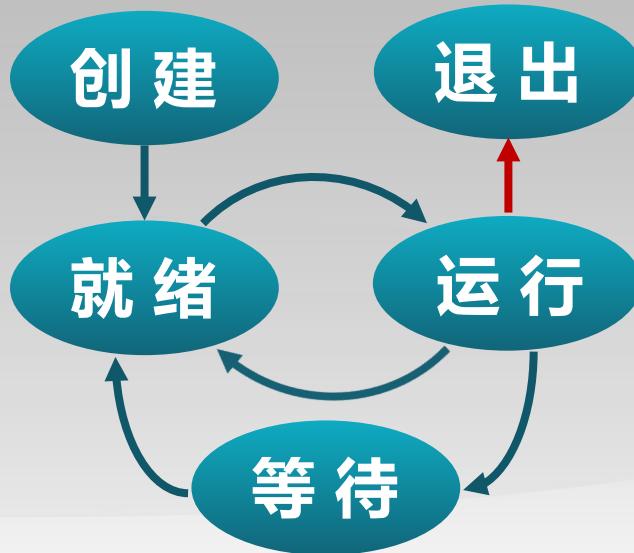
# 进程唤醒

- 唤醒进程的情况：
  - ▶ 被阻塞进程需要的资源可被满足
  - ▶ 被阻塞进程等待的事件到达
- 进程只能被别的进程或操作系统唤醒

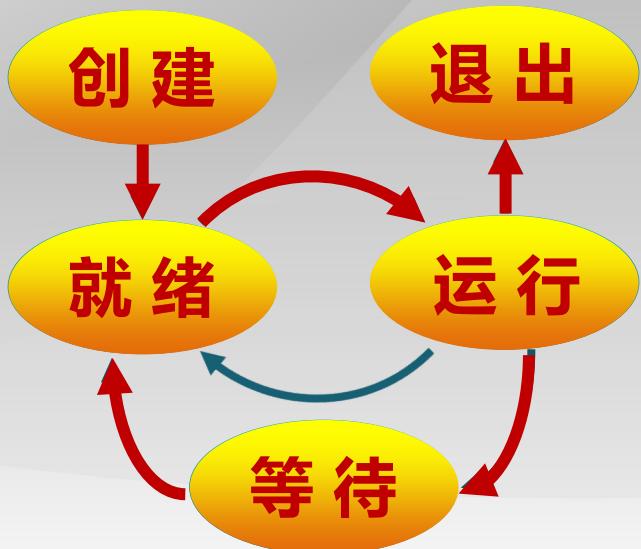


# 进程结束

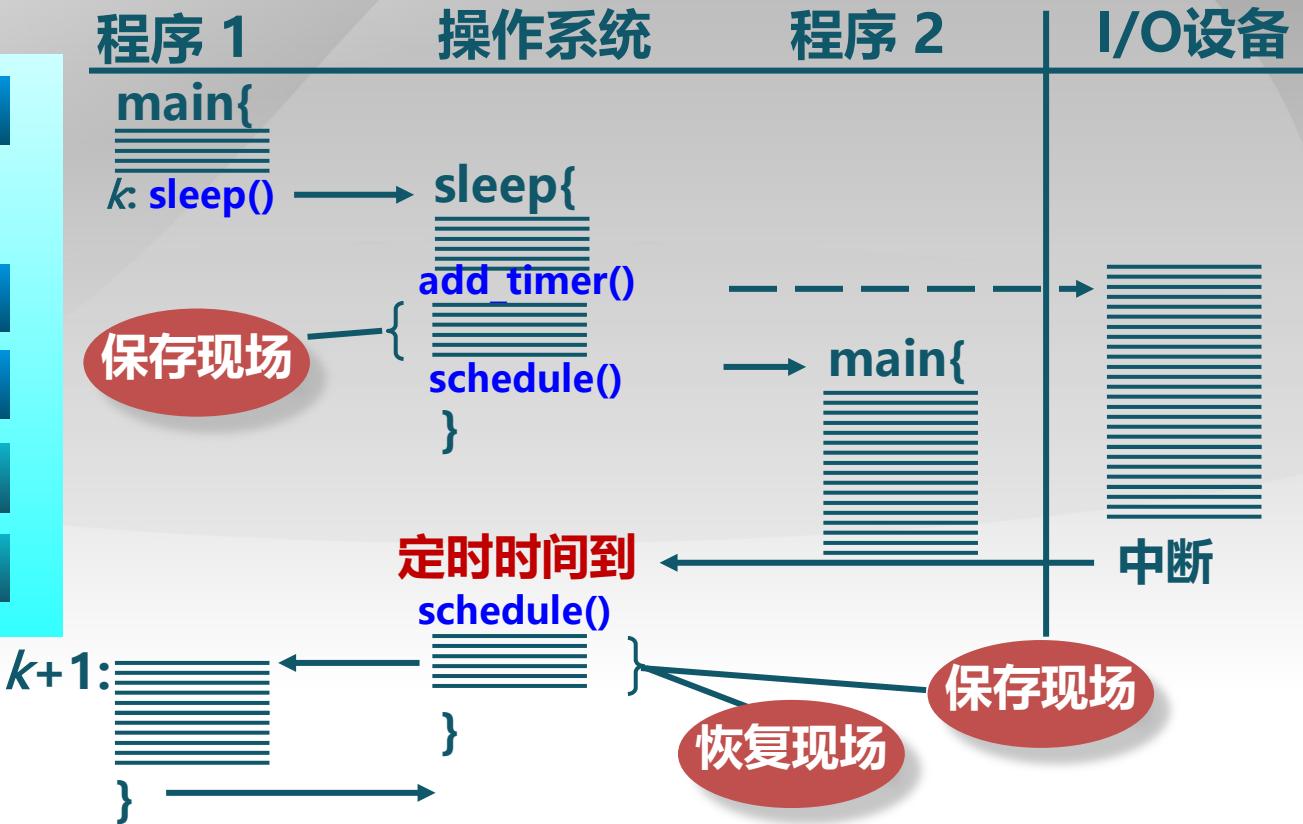
- 进程结束的情况：
  - ▶ 正常退出(自愿的)
  - ▶ 错误退出(自愿的)
  - ▶ 致命错误(强制性的)
  - ▶ 被其他进程所杀(强制性的)

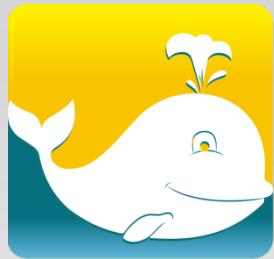


# sleep()系统调用对应的进程状态变化



# 进程切换

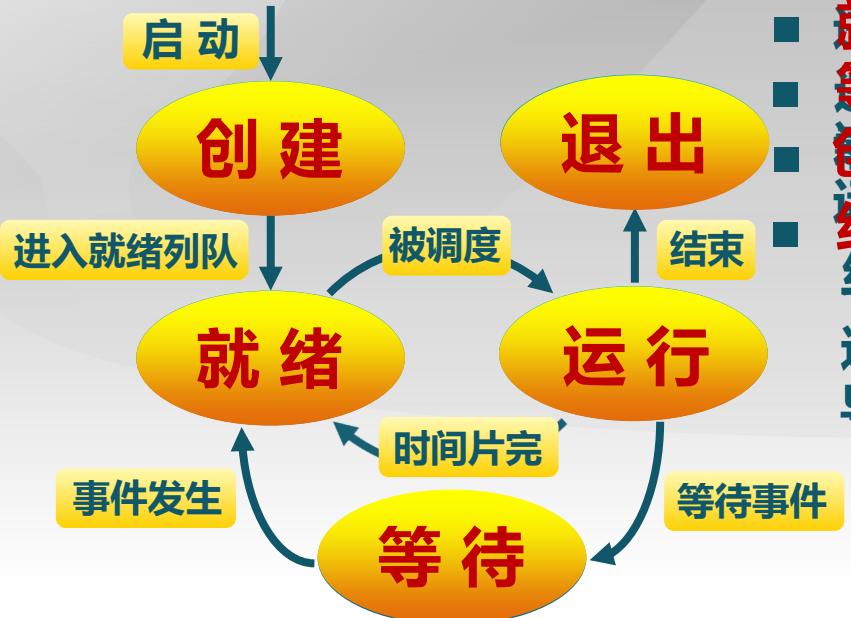




# 操作系统

Operating System

## 三状态进程模型



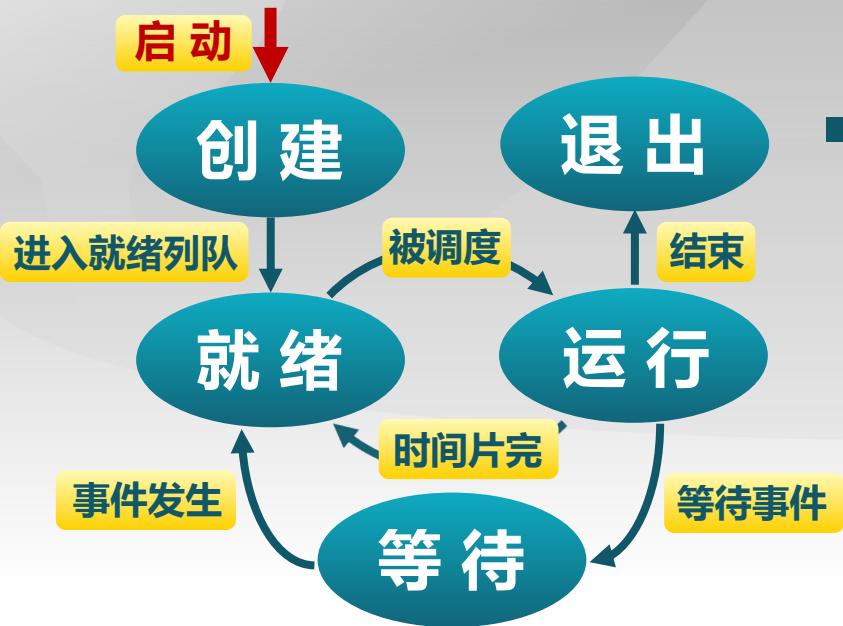
- 运行状态(Running)
  - 就绪状态(Ready)
  - 等待状态(又称阻塞状态Blocked)
  - 创建状态(New)
  - 结束状态(Exit)

# 进程在整个生命周期分为三种基本状态

# 三状态进程模型

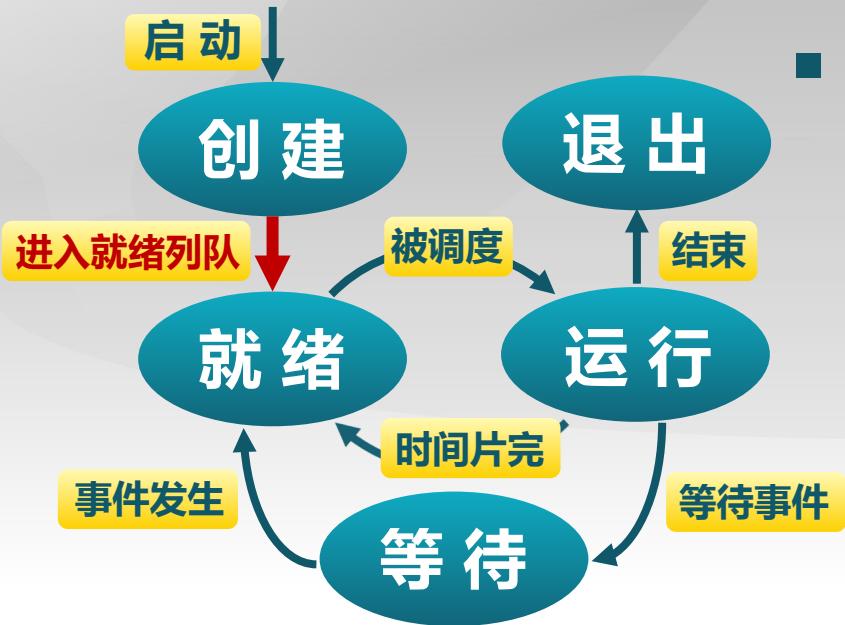


# 三状态进程模型



- **NULL→创建**  
一个新进程被产生出来执行一个程序

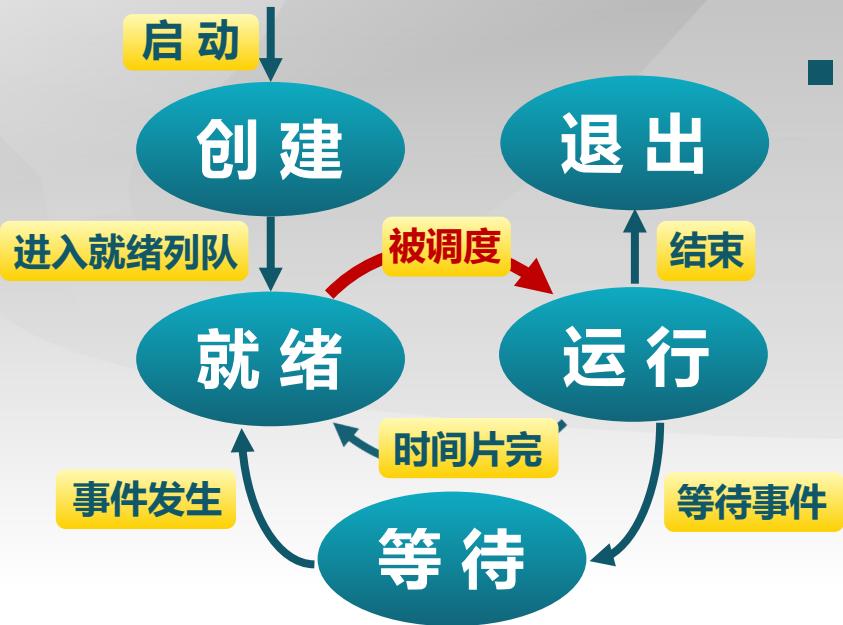
# 三状态进程模型



## ■ 创建→就绪

当进程被创建完成并初始化后，一切就绪准备运行时，变为就绪状态

# 三状态进程模型



## ■ 就绪→运行

处于就绪状态的进程被进程调度程序选中后，就分配到处理机上来运行

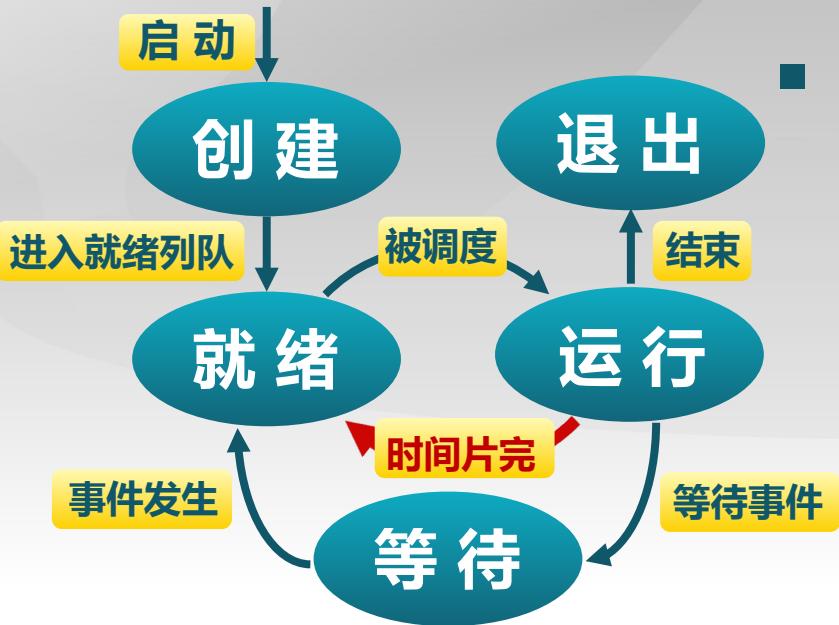
# 三状态进程模型



## ■ 运行→结束

当进程表示它已经完成或者因出错，当前运行进程会由操作系统作结束处理

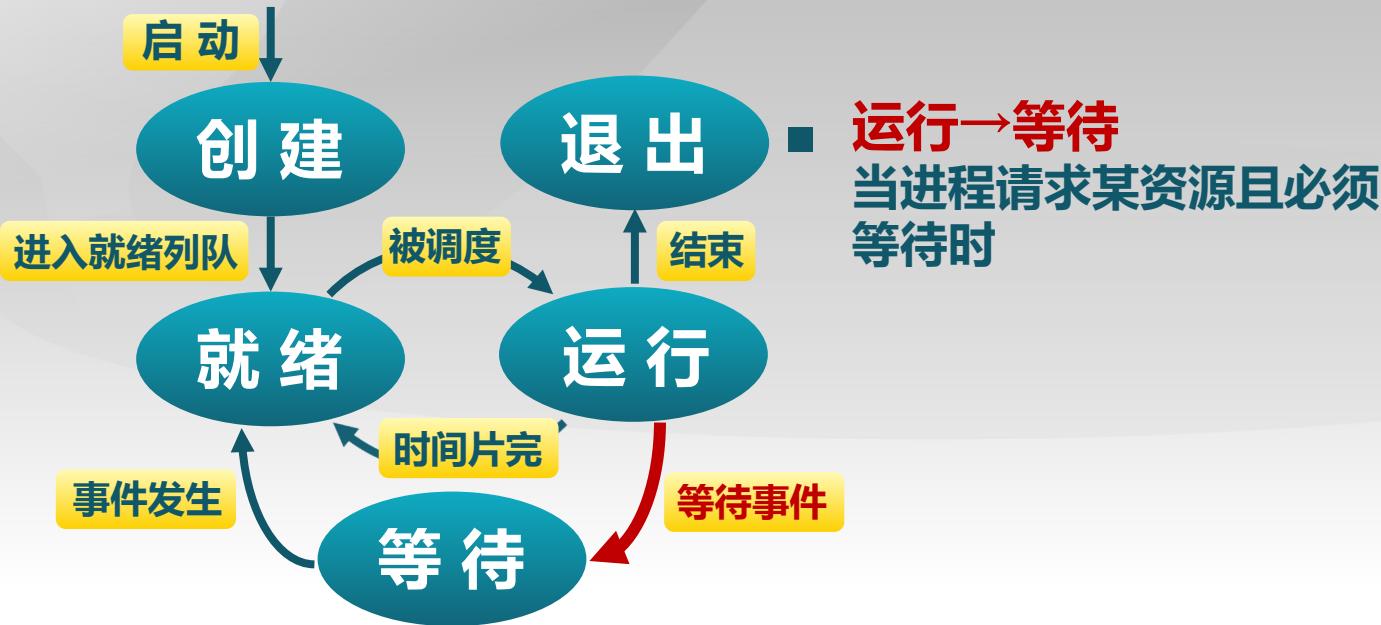
# 三状态进程模型



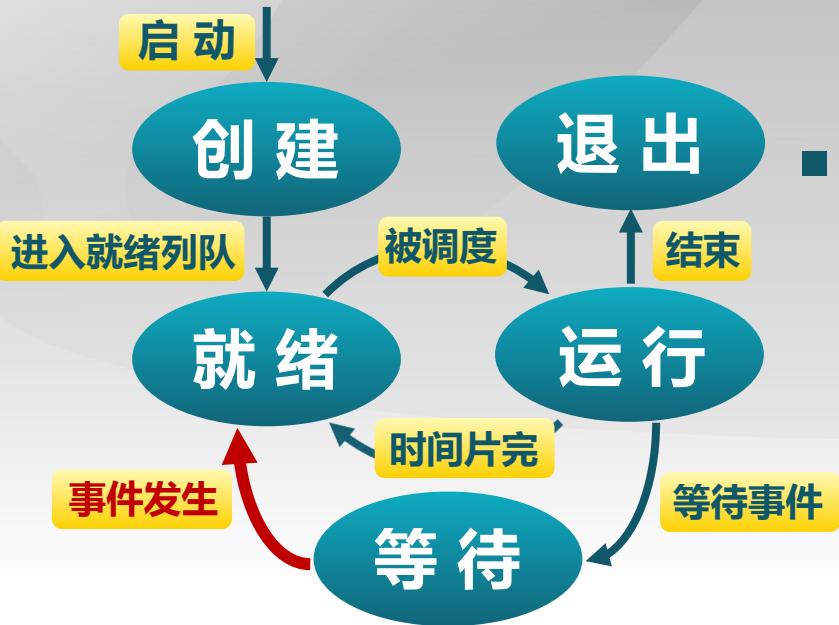
## ■ 运行→就绪

处于运行状态的进程在其运行过程中，由于分配给它的处理器时间片用完而让出处理器

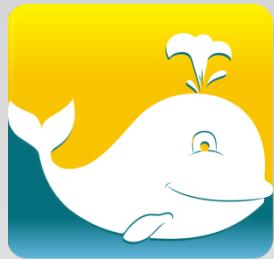
# 三状态进程模型



# 三状态进程模型



- 等待→就绪  
当进程要等待某事件到来时，它从阻塞状态变到就绪状态

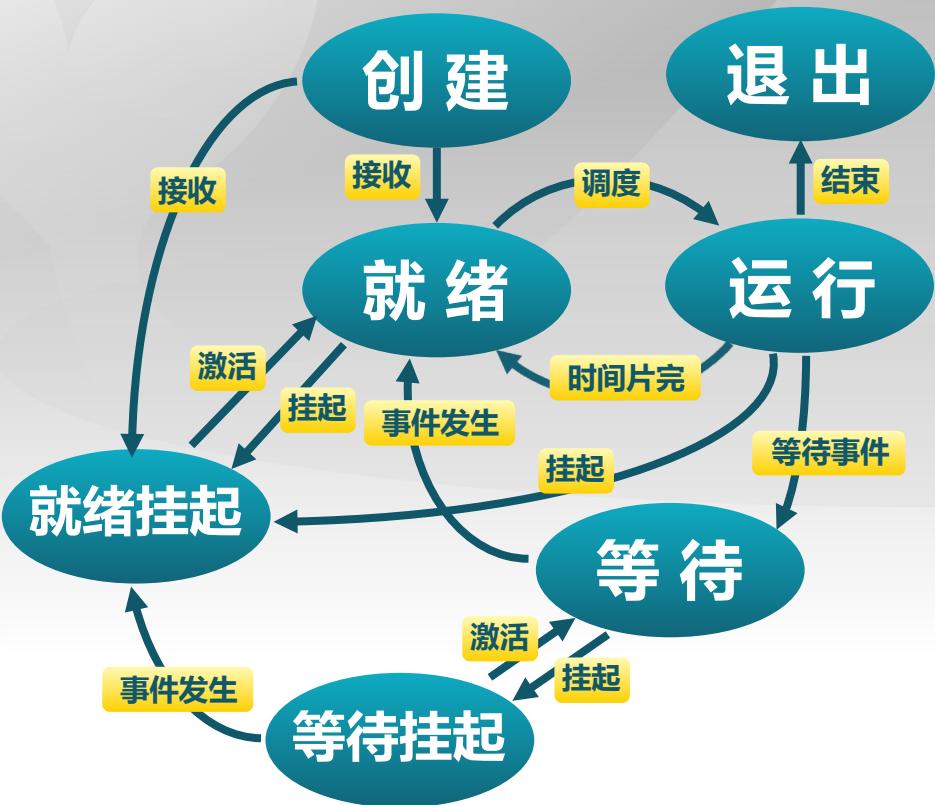


# 操作系统

Operating System

# 进程挂起

处在挂起状态的进程映像在磁盘上，目的是减少进程占用内存

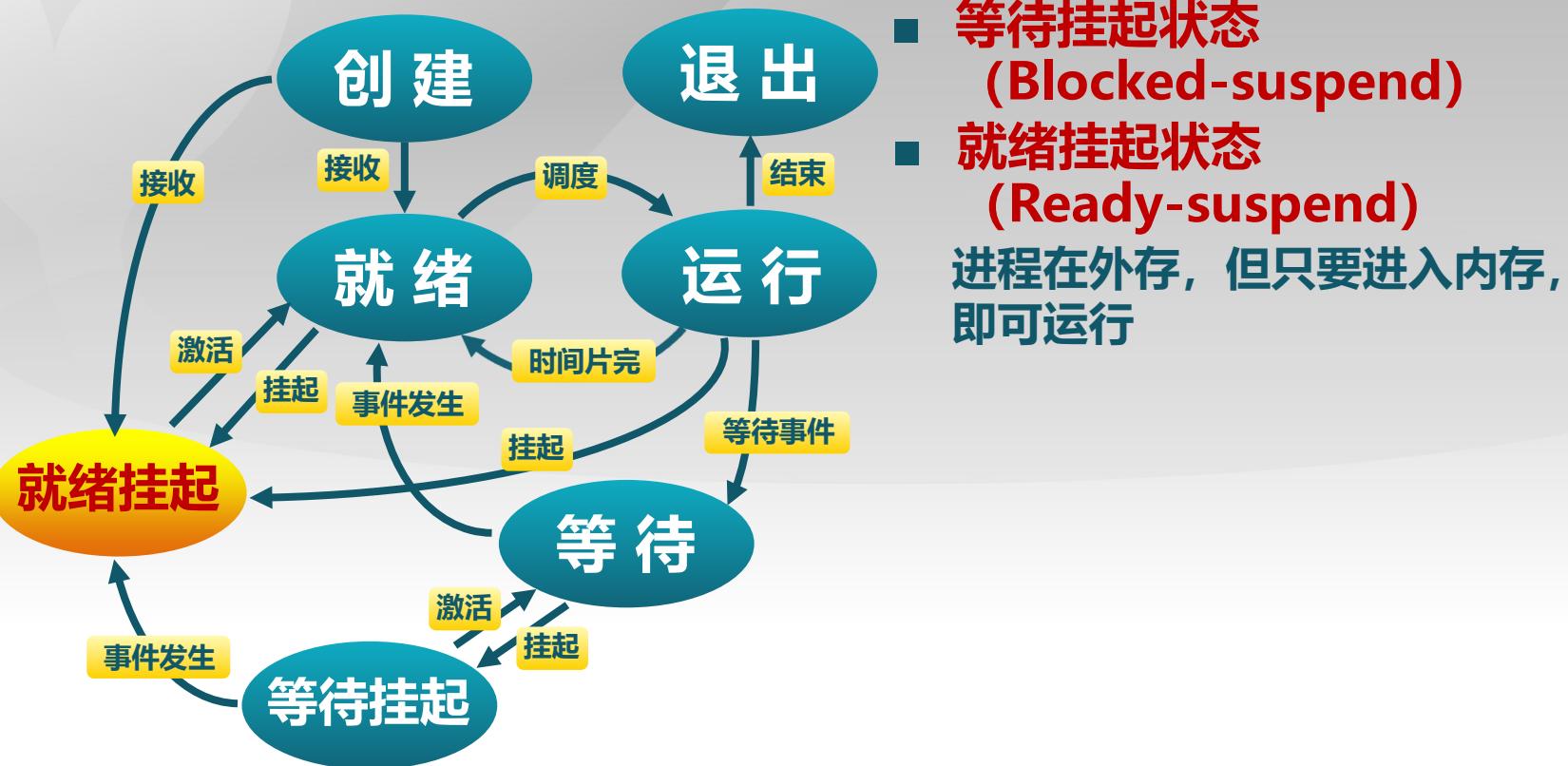


# 挂起状态



- **等待挂起状态  
(Blocked-suspend)**  
进程在外存并等待某事件的出现

# 挂起状态



# 与挂起相关的状态转换



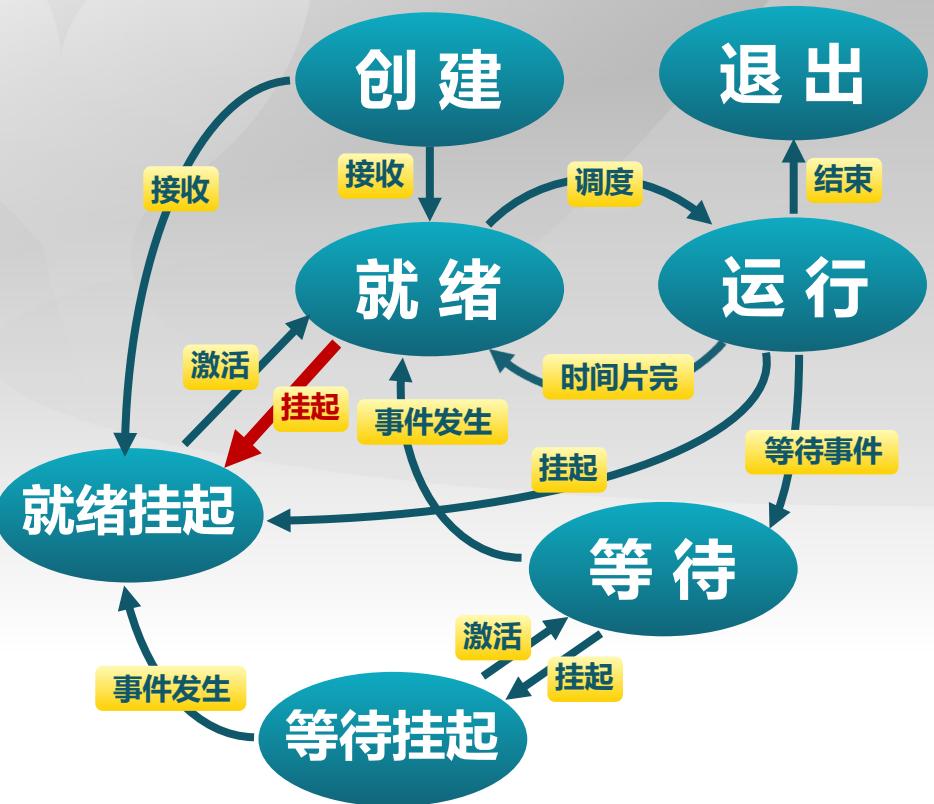
- 挂起(Suspend): 把一个进程从内存转到外存

# 与挂起相关的状态转换



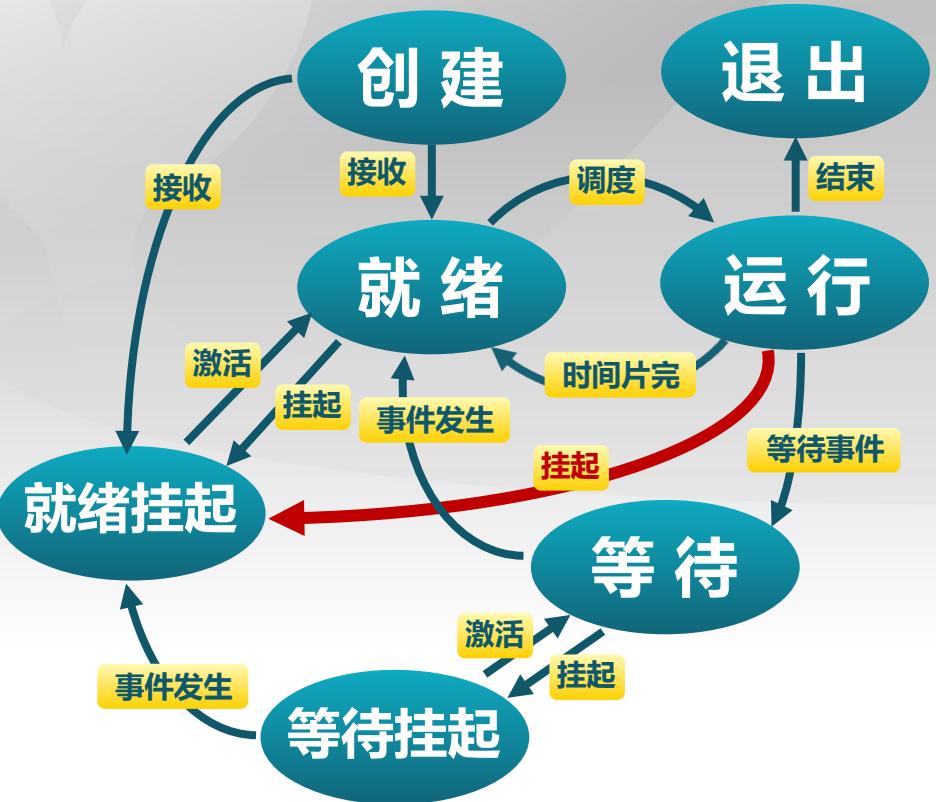
- 挂起(Suspend): 把一个进程从内存转到外存
  - ▶ 等待到等待挂起  
没有进程处于就绪状态或就绪进程要求更多内存资源

# 与挂起相关的状态转换



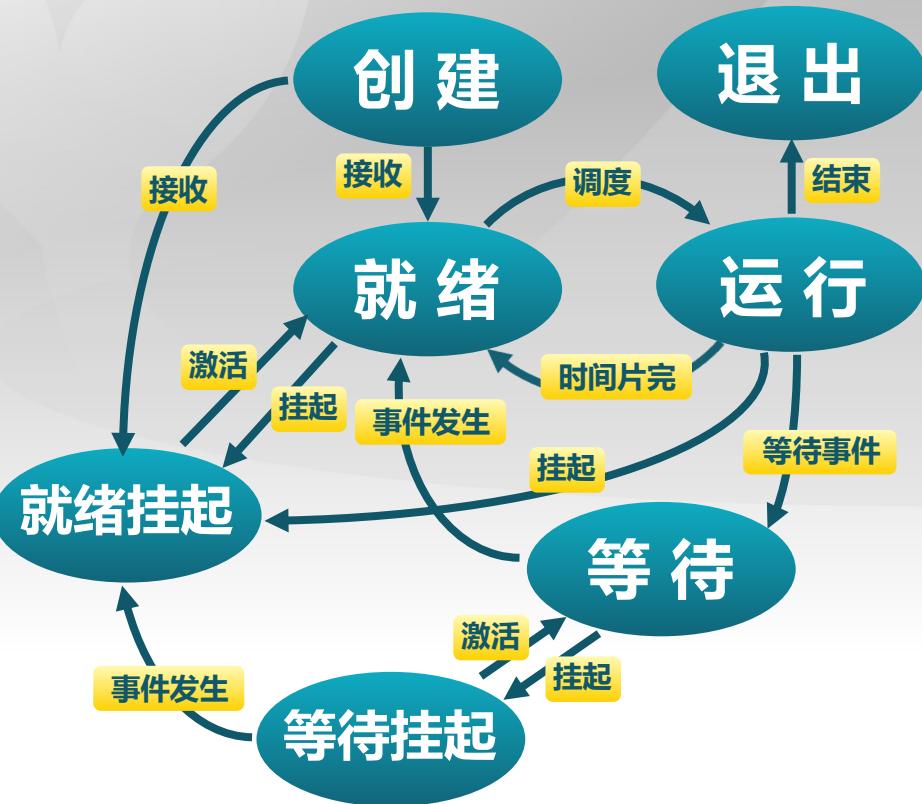
- 挂起(Suspend): 把一个进程从内存转到外存
  - ▶ 等待到等待挂起
  - ▶ 就绪到就绪挂起当有高优先级等待 (系统认为会很快就绪的) 进程和低优先级就绪进程

# 与挂起相关的状态转换



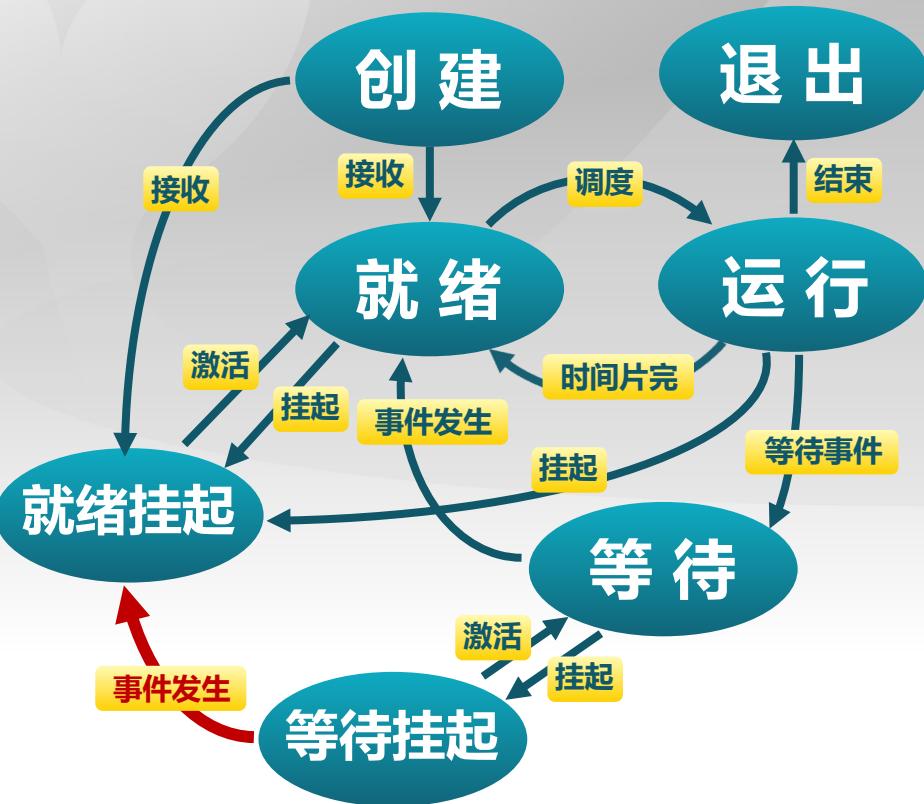
- **挂起(Suspend):** 把一个进程从内存转到外存
  - ▶ 等待到等待挂起
  - ▶ 就绪到就绪挂起
  - ▶ 运行到就绪挂起对抢先式分时系统，当有高优先级等待挂起进程因事件出现而进入就绪挂起

# 与挂起相关的状态转换



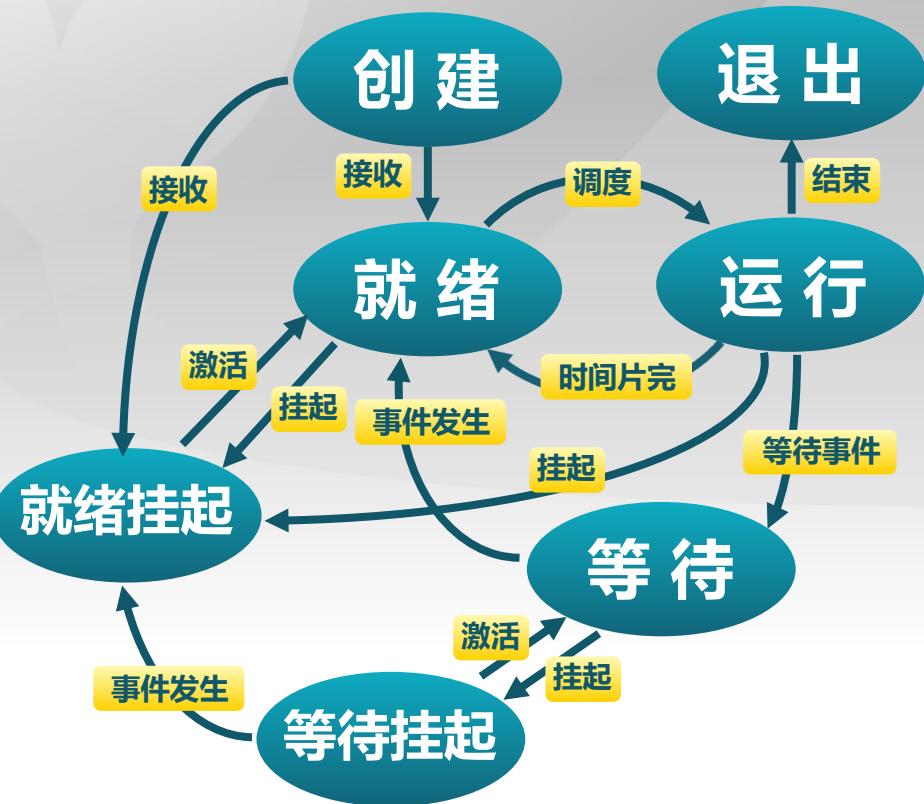
■ 在外存时的状态转换

# 与挂起相关的状态转换



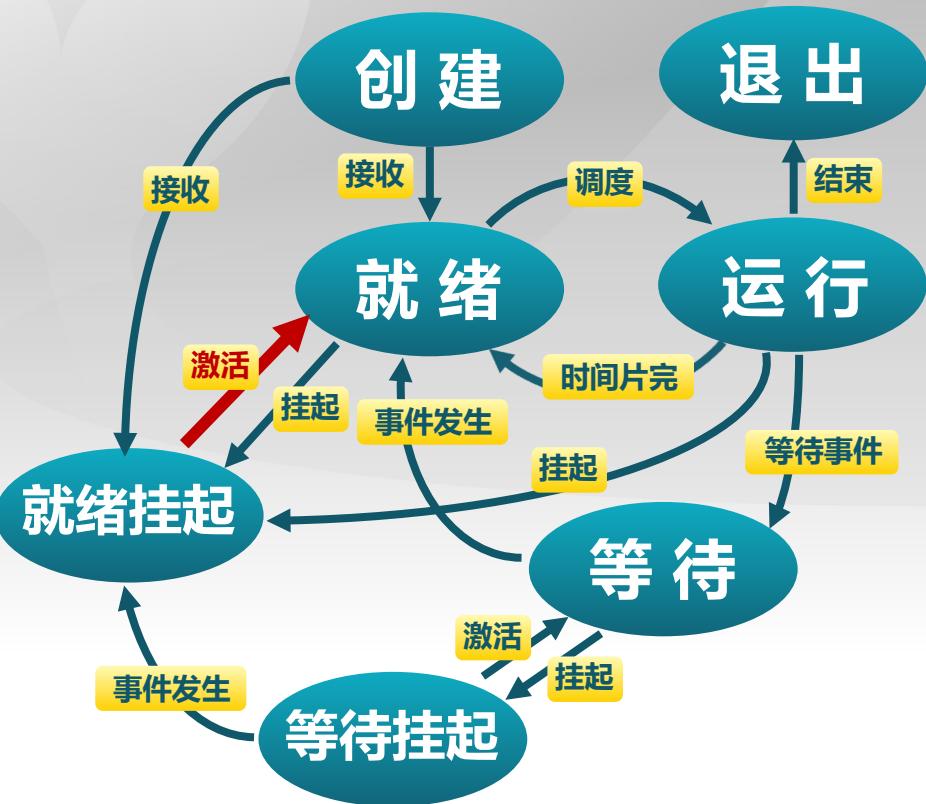
- 在外存时的状态转换
  - 等待挂起到就绪挂起  
当有等待挂起进程因相关事件出现

# 与挂起相关的状态转换



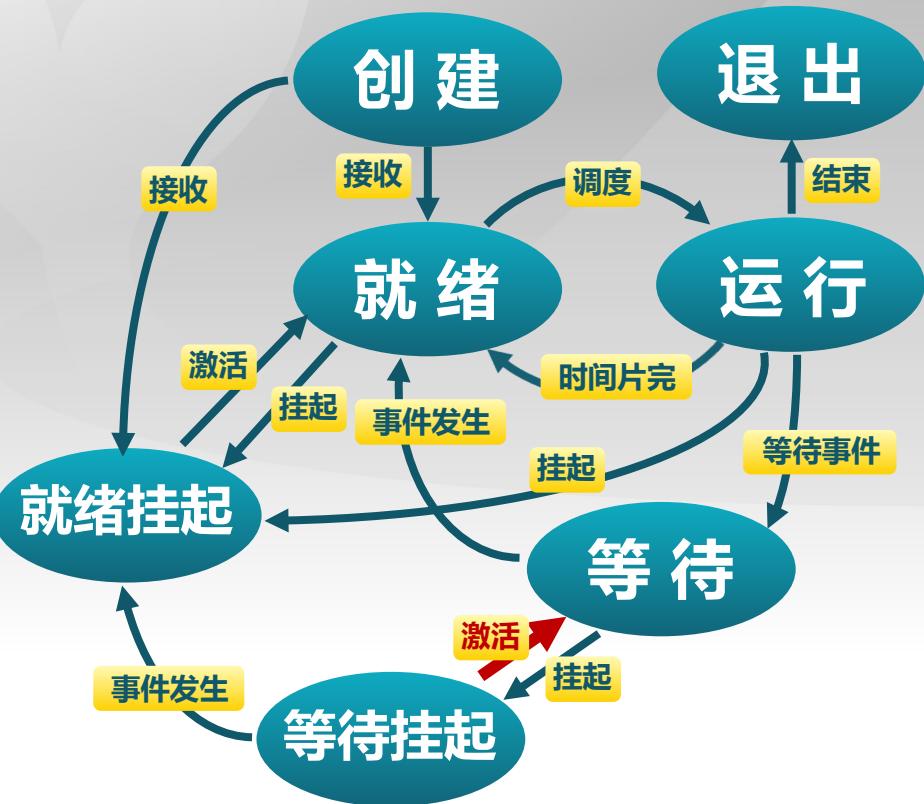
- 激活 (Activate) : 把一个进程从外存转到内存

# 与挂起相关的状态转换



- 激活 (Activate) : 把一个进程从外存转到内存
  - ▶ 就绪挂起到就绪  
没有就绪进程或挂起就绪进程优先级高于就绪进程

# 与挂起相关的状态转换



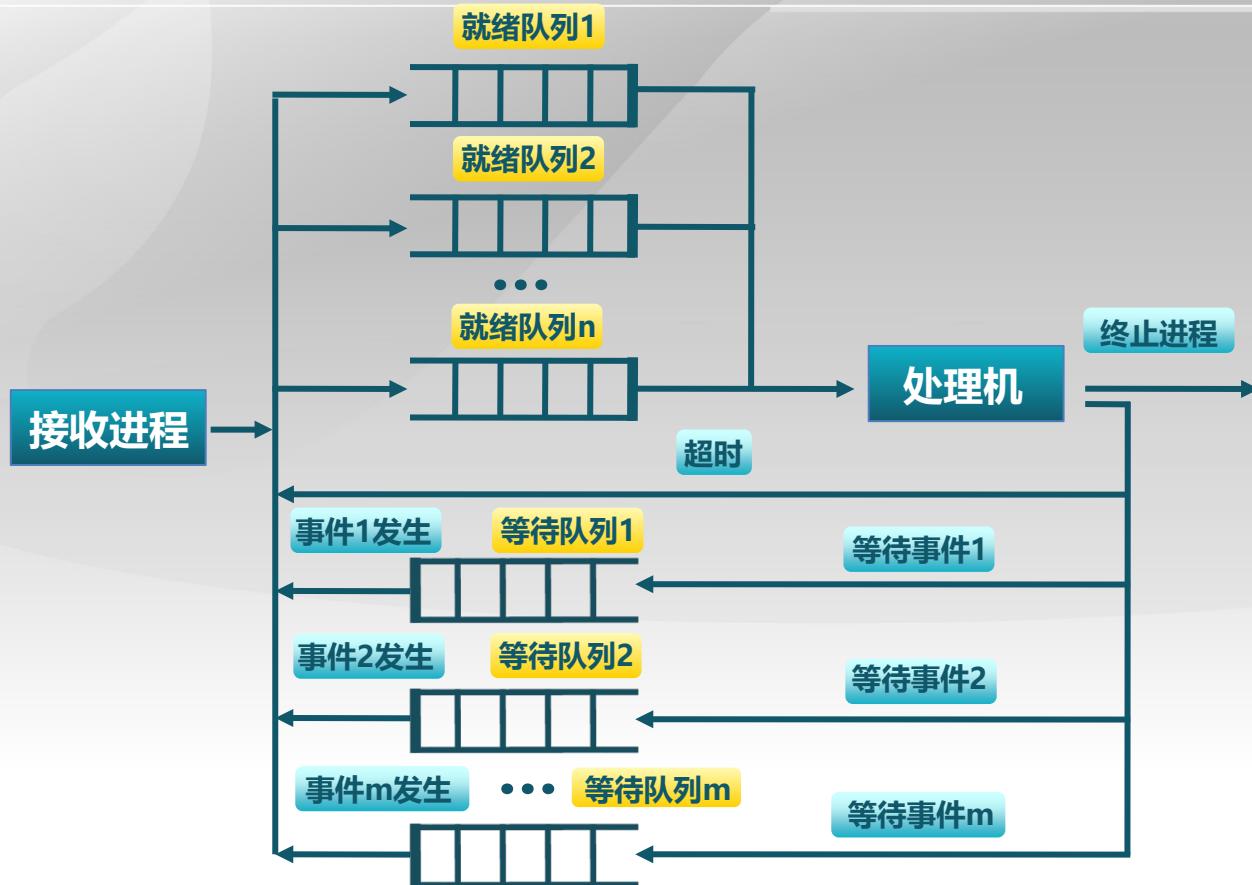
- 激活 (Activate) : 把一个进程从外存转到内存
  - ▶ 就绪挂起到就绪
  - ▶ 等待挂起到等待

当一个进程释放足够内存，并有高优先级等待挂起进程

# 状态队列

- 由操作系统来维护一组队列，表示系统中所有进程的当前状态
- 不同队列表示不同状态
  - ▶ 就绪队列、各种等待队列
- 根据进程状态不同，进程PCB加入相应队列
  - ▶ 进程状态变化时，它所在的PCB会从一个队列换到另一个

# 进程状态的队列表现





# 操作系统

Operating System

# 为什么引入线程

【案例】编写一个MP3播放软件。核心功能模块有三个：

- (1) 从MP3音频文件当中读取数据
- (2) 对数据进行解压缩
- (3) 把解压缩后的音频数据播放出来

# 单进程的实现方法

```
main( )
{
    while(TRUE)
    {
        I/O ----- Read( );
        CPU ----- Decompress( );
        Play( );
    }
}
Read( ) { ... }
Decompress( ) { ... }
Play( ) { ... }
```

问题：

1. 播放出来的声音能否连贯
2. 各个函数之间不是并发执行，影响资源的使用效率

# 多进程的实现方法

程序1

```
main( )
{
    while(TRUE)
    {
        Read( );
    }
}
Read( ) { ... }
```

程序2

```
main( )
{
    while(TRUE)
    {
        Decompress( );
    }
}
Decompress( ) { ... }
```

程序3

```
main( )
{
    while(TRUE)
    {
        Play( );
    }
}
Play( ) { ... }
```

存在的问题：

1. 进程之间如何通信，共享数据？
2. 系统开销较大：创建进程、进程结束、进程切换

# 多线程的解决思路

在进程内部增加一类实体，满足以下特性：

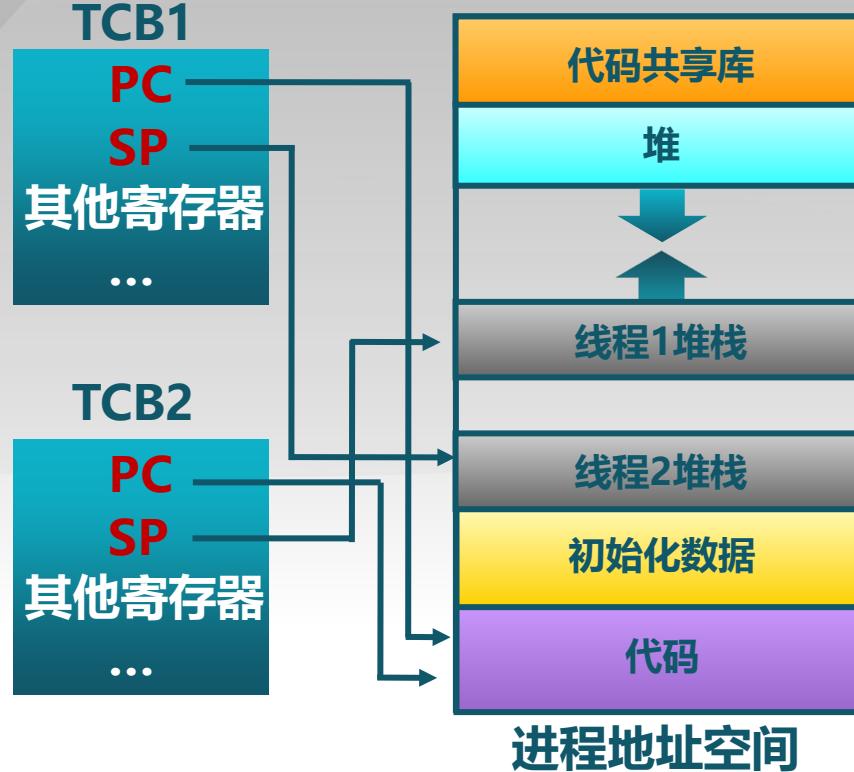
- (1) 实体之间可以并发执行
- (2) 实体之间共享相同的地址空间

这种实体就是线程（Thread）

# 线程的概念

线程是进程的一部分，描述指令流执行状态。它是进程中的**指令执行流的最小单元**，是**CPU调度**的基本单位。

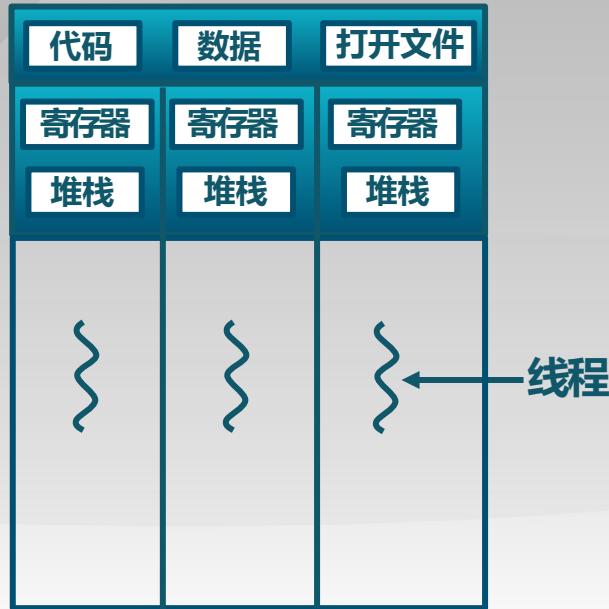
- ▶ 进程的资源分配角色：  
进程由一组相关资源构成，包括地址空间（代码段、数据段）、打开的文件等各种资源
- ▶ 线程的处理机调度角色：  
线程描述在进程资源环境中的指令流执行状态



# 进程和线程的关系



单线程进程



多线程进程

# 线程 = 进程 - 共享资源

- 线程的优点：

- ▶ 一个进程中可以同时存在多个线程
- ▶ 各个线程之间可以并发地执行
- ▶ 各个线程之间可以共享地址空间和文件等资源

- 线程的缺点：

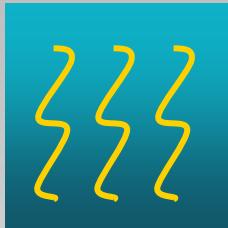
- ▶ 一个线程崩溃，会导致其所属进程的所有线程崩溃

# 不同操作系统对线程的支持



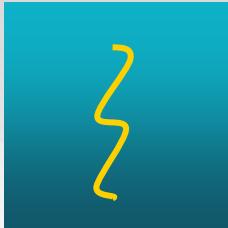
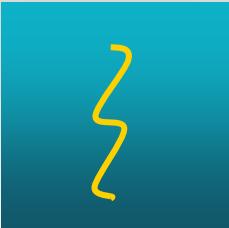
单进程系统

实例：MS-DOS



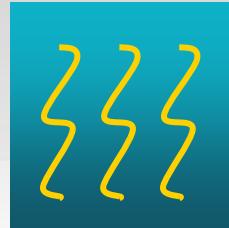
单进程多线程系统

实例：pSOS



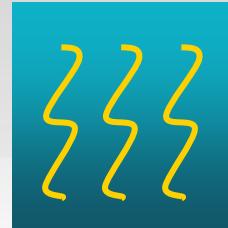
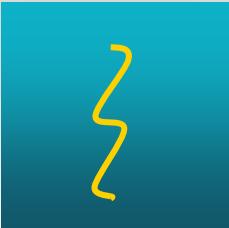
多进程系统

实例：传统UNIX



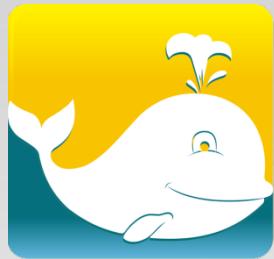
多线程系统

实例：现代UNIX



# 线程与进程的比较

- 进程是资源分配单位，线程是CPU调度单位
- 进程拥有一个完整的资源平台，而线程只独享指令流执行的必要资源，如寄存器和栈
- 线程具有就绪、等待和运行三种基本状态和状态间的转换关系
- 线程能减少并发执行的时间和空间开销
  - ▶ 线程的创建时间比进程短
  - ▶ 线程的终止时间比进程短
  - ▶ 同一进程内的线程切换时间比进程短
  - ▶ 由于同一进程的各线程间共享内存和文件资源，可不通过内核进行直接通信



# 操作系统

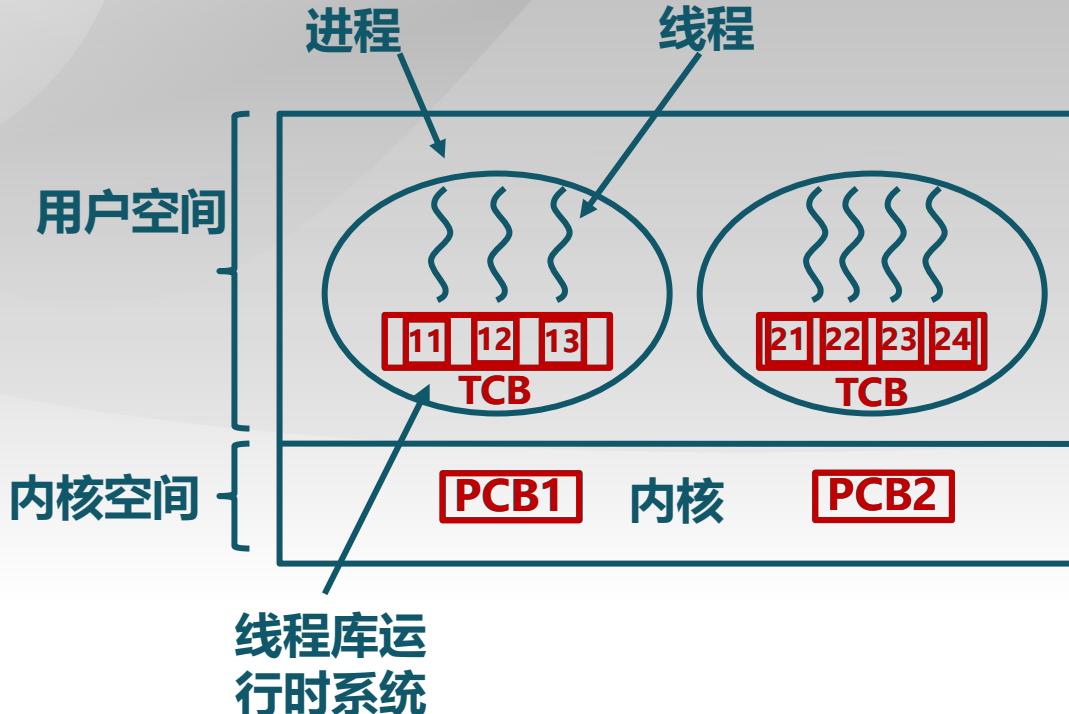
Operating System

# 线程的三种实现方式

- 用户线程：在用户空间实现  
**POSIX Pthreads, Mach C-threads,  
Solaris threads**
- 内核线程：在内核中实现  
**Windows, Solaris, Linux**
- 轻量级进程：在内核中实现，支持用户线程  
**Solaris (LightWeight Process)**

# 用户线程

由一组用户级的线程库函数来完成线程的管理，  
包括线程的创建、终止、同步和调度等

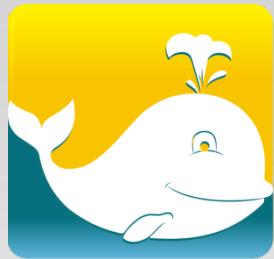


# 用户线程的特征

- 不依赖于操作系统的内核
  - ▶ 内核不了解用户线程的存在
  - ▶ 可用于不支持线程的多进程操作系统
- 在用户空间实现的线程机制
  - ▶ 每个进程有私有的线程控制块 (TCB) 列表
  - ▶ TCB由线程库函数维护
- 同一进程内的用户线程切换速度快
  - ▶ 无需用户态/核心态切换
- 允许每个进程拥有自己的线程调度算法

# 用户线程的不足

- 线程发起系统调用而阻塞时，则整个进程进入等待
- 不支持基于线程的处理机抢占
  - ▶ 除非当前运行线程主动放弃，它所在进程的其他线程无法抢占CPU
- 只能按进程分配CPU时间
  - ▶ 多个线程进程中，每个线程的时间片较少



# 操作系统

Operating System

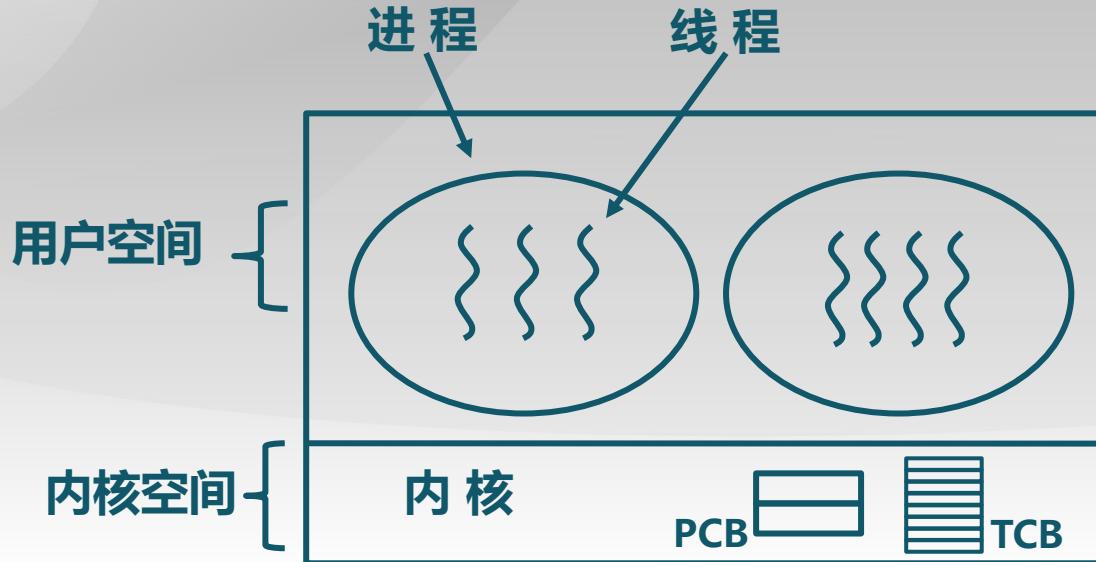


# 操作系统

Operating System

# 内核线程

由内核通过系统调用实现的线程机制，由内核完成  
线程的创建、终止和管理

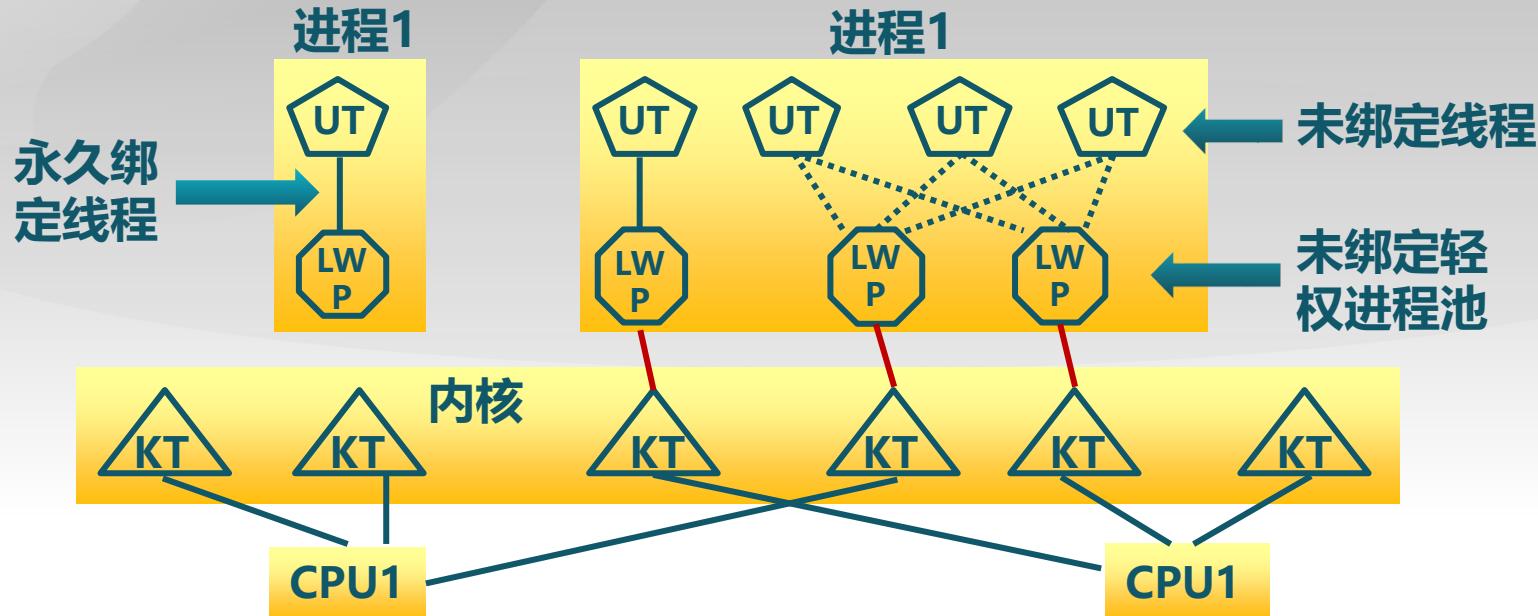


# 内核线程的特征

- 由内核维护PCB和TCB
- 线程执行系统调用而被阻塞不影响其他线程
- 线程的创建、终止和切换相对较大
  - ▶ 通过系统调用/内核函数，在内核实现
- 以线程为单位进行CPU时间分配
  - ▶ 多线程的进程可获得更多CPU时间

# 轻权进程(LightWeight Process)

内核支持的用户线程。一个进程可有一个或多个轻量级进程，每个轻权进程由一个单独的内核线程来支持。 (Solaris/Linux)



# 用户线程与内核线程的对应关系

