西安科技大学
XI' AN UNIVERSITY OF SCIENCE TECHNOLOGY

第三章 进程管理

刘晓建

2017年9月11日

本章概要

进程管理是操作系统的核心，也是这门课学习的关键内容。本章内容的重点是理解和掌握进程的概念。可以从多个角度看待进程：

- 可以把进程看作操作系统分配资源的单位，在这个意义上，进程把计算机系统的各种资源紧密的关联起来，使得进程管理成为知识体系的核心和纽带。
- 可以把进程看作执行一个程序的环境和上下文。在这个意义上，进程的概念又与先修课程，如程序设计、编译原理等内容连贯起来，成为程序设计和软件开发活动中不可回避，必须掌握的最基本概念之一。

3.1 进程的概念和结构

进程的概念

进程是对一个计算任务的抽象和封装，使每个计算任务更好的实现隔离性、资源共享性和同步的需求。

存在很多关于进程的说法，归纳起来，可以这样理解进程的概念：

- 进程是计算机程序在处理器上执行时所发生的活动，即进程是程序的一次执行活动；
- 进程是对一个计算任务的抽象和封装，它是由一个执行流、一个数据集和相关系统资源所组成的一个活动单元；
- 进程是程序执行的一个实例，是动态的概念，而程序是行为的一个规则，通常以文件的形式存在，是静态的概念。一个计算机可以同时运行一个程序的多个进程。

思考：程序和进程的联系和区别?

- 程序是由按照规定的语法规则，所编写的实现特定计算任务的语句序列。分为高级语言程序和机器指令程序。
- 程序本质上是一个二进制串，是静态概念。如果没有处理器、内存等资源要素，程序本身是不可能执行的；而进程是程序的一个执行过程，是动态的概念，进程为程序的执行提供了一个环境，进程不仅包括程序本身，还包括供它运行的环境要素。
- 一个程序可以对应多个进程实例，但一般而言，一个进程只能执行一个程序。
- 程序是以文件的形式存储的，因此其生命周期可以跨越开关机；而进程的生命周期不能超越一次开关机周期。

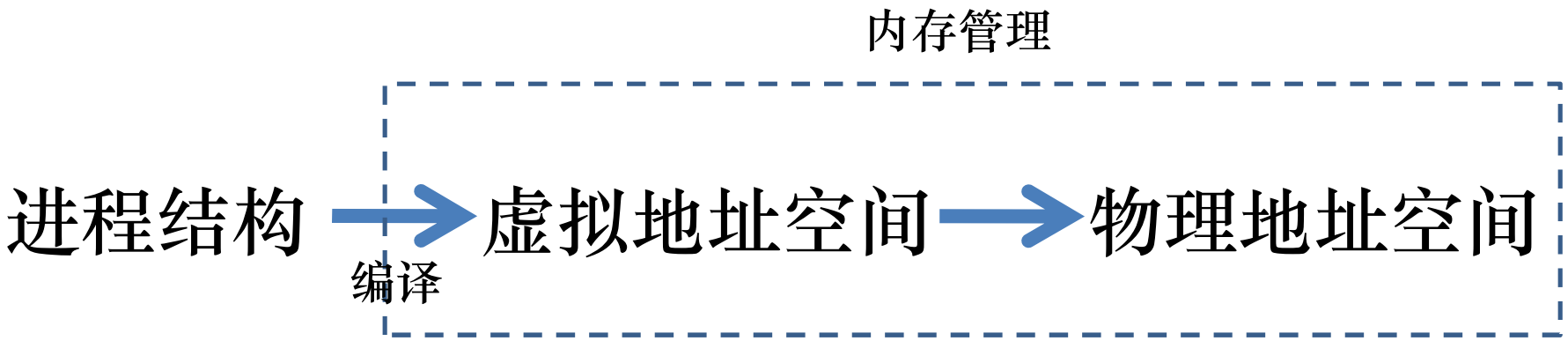
进程的结构——进程包括哪些部分，这些部分的关系是什么？

一个程序的执行不仅依赖于该程序本身，而且还需要一个运行环境，即上下文。一个进程包括五个部分：

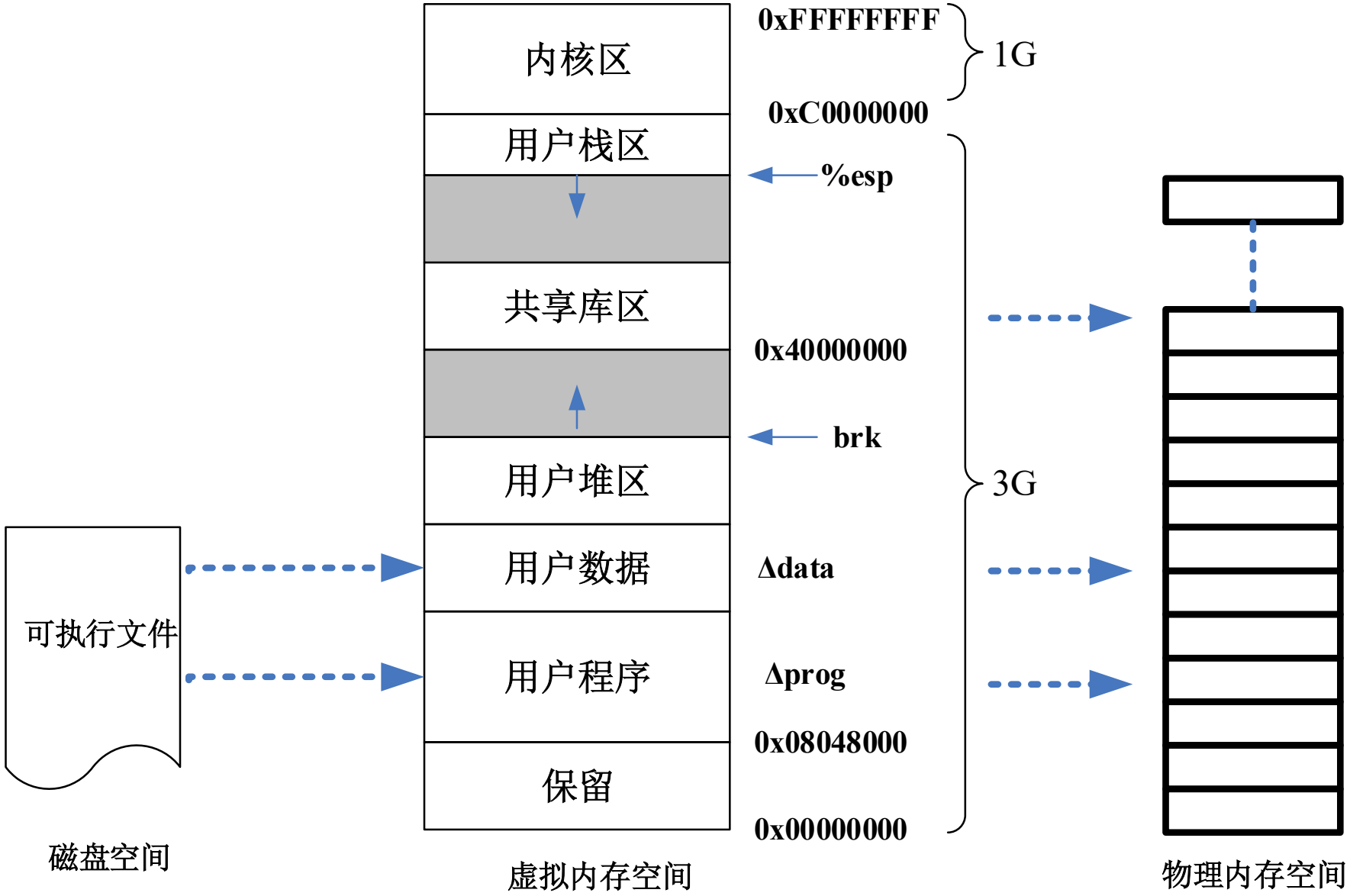
组成部分	私有/共享	说明
用户程序	私有	将被执行的程序，这部分通常是只读的，不可更改
用户数据	私有	包括程序数据、用户堆栈区和可修改的程序。这部分内容是可以更改的
系统栈	私有或共享	每个进程有一个或多个系统栈。栈用于保存参数、过程调用地址和系统调用地址
进程控制块	私有	操作系统控制进程所需要的各种数据信息
共享区域	共享	用于多个进程共享操作系统代码、共享库、共享内存等

处理器（CPU）：唯一具有处理能力的单元

进程的这些组成部分必须被映射到物理内存中，才能让程序运行起来。采用虚拟内存管理的操作系统不是把这些组成部分直接映射到物理内存，而是首先把它们映射到进程**虚拟地址空间**，然后再通过内存管理机制，把进程虚拟地址空间映射到物理内存地址空间。



我们把进程组成部分在虚拟地址空间中的分布称为**进程虚拟地址空间布局**，或**进程映像（Image）**。对于一个32位系统，使用32位编码一个字节地址，因此进程虚拟空间为{0,1,2,…, 2^{32}-1}，虚拟地址空间的大小为2^{32}，即4GB。



不同操作系统的进程虚拟地址空间布局通常是不同的。
比如，图3-1是Linux 32位系统的进程地址空间布局。

- 练习：

  假定一个程序编译后，生成的可执行文件：

- 代码段：16K
- 数据段：1K
- 用户栈区为4K，
- 预留的堆区为4K。

请给出该程序在Linux32位系统中的虚拟地址空间布局。

由于Linux中，用户程序的虚拟地址从0x08048000开始分布。

思考：每个进程的用户程序都是从虚拟地址0x08048000开始分布的，那么多个进程占据的内存会不会发生冲突呢？

不会。这里的地址空间是指进程的虚拟地址空间，并不是物理内存地址空间。在后面的内存管理章节，我们可以看到不同进程的虚拟地址空间实际上被内存管理机制映射到不同的物理内存地址空间，因此不会造成冲突。

进程控制块（PCB，Process Control Block）

操作系统需要有关每个进程的大量信息，这些信息保存在进程控制块中。进程控制块实际上是一个描述进程元信息的数据结构，由操作系统来分配和管理，并保存在内核中。

进程标识信息	
标识符	● PID（Process ID）：一个进程的标识符 ● PPID（Parent Process ID）：创建这个进程的进程（父进程）标识符 ● UID（User ID）：用户标识符
处理器状态信息	
用户可见寄存器	用户可见寄存器是用户模式下，处理器能够访问的寄存器。通常有8到32个此类寄存器，而在一些RISC处理器中有超过100个此类寄存器
控制和状态寄存器	用于控制处理器操作的各种处理器寄存器，包括： ● 程序计数器PC：包含将要读取的下一条指令的地址 ● 条件码：最近的算术或逻辑运算的结果（例如符号、零、进位、等于、溢出） ● 状态信息：包括中断允许/禁止标志、处理器异常模式
栈指针	每个进程有一个或多个与之关联的栈。栈用于保存参数和过程调用或系统调用的地址，栈指针指向栈顶或栈底

进程控制信息	
调度和状态信息	这是操作系统执行其调度功能所需要的信息，典型的信息项包括： ● 进程状态：如就绪态、运行态、阻塞态等 ● 优先级：描述进程调度优先级的一个或多个域。在某些系统中，需要多个值（例如默认、当前、最高许可） ● 调度相关信息：这取决于所使用的调度算法。例如进程等待的时间总量和进程在上一次运行时执行时间总量 ● 事件：进程在继续执行前需要等待的事件标识
数据结构	进程可以以队列、环或者别的结构形式与其它进程进行链接。例如，所有具有某一特定优先级且处于等待状态的进程可链接在一个队列中；进程还可以表示与另一个进程的父子（创建-被创建）关系。进程控制块为支持这些结构，需要包含指向其它进程的指针。
进程间通信	与两个独立进程间的通信相关联的各种标记、信号和信息。
进程特权	进程根据其可以访问的内存空间以及可以执行的指令类型被赋予各种特权。此外，特权还用于系统实用程序和服务的程序使用
存储管理	包括指向该进程虚拟内存空间的段表和页表的指针
资源的所有权和使用情况	进程控制的资源可以表示成诸如一个打开的文件，还可能包括处理器或其它资源的使用历史；调度器需要这些信息

3.2 进程的状态

- 状态机

通常采用状态机模型（State machine）来描述进程的状态以及状态之间的迁移关系。一个状态机实际上是一个由节点集合Node和边集合Edge构成的有向图<Node, Edge>，其中

- Node被解释为状态集，
- Edge被解释为状态迁移的集合。

状态迁移是一个五元组：

<*state, event* [*guard*]/*action, state'*>

五状态模型

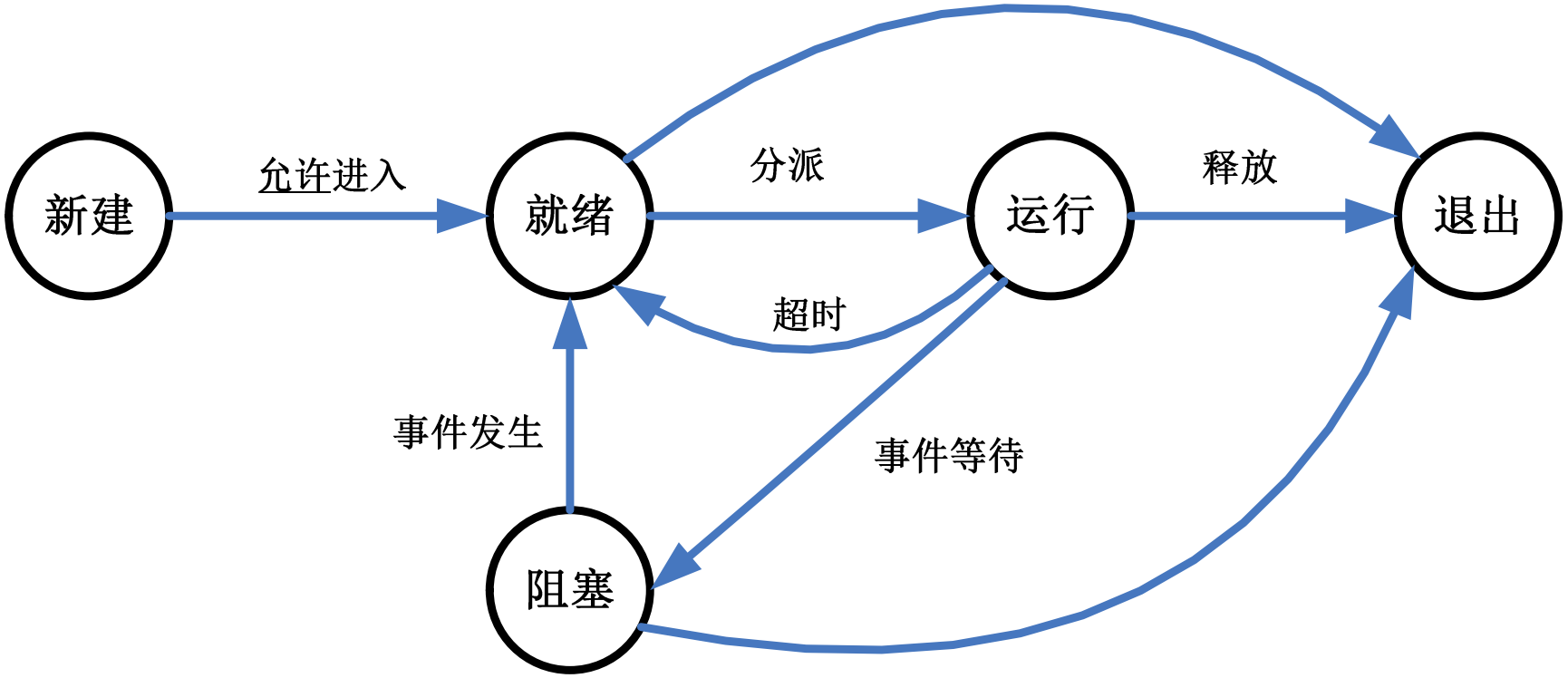


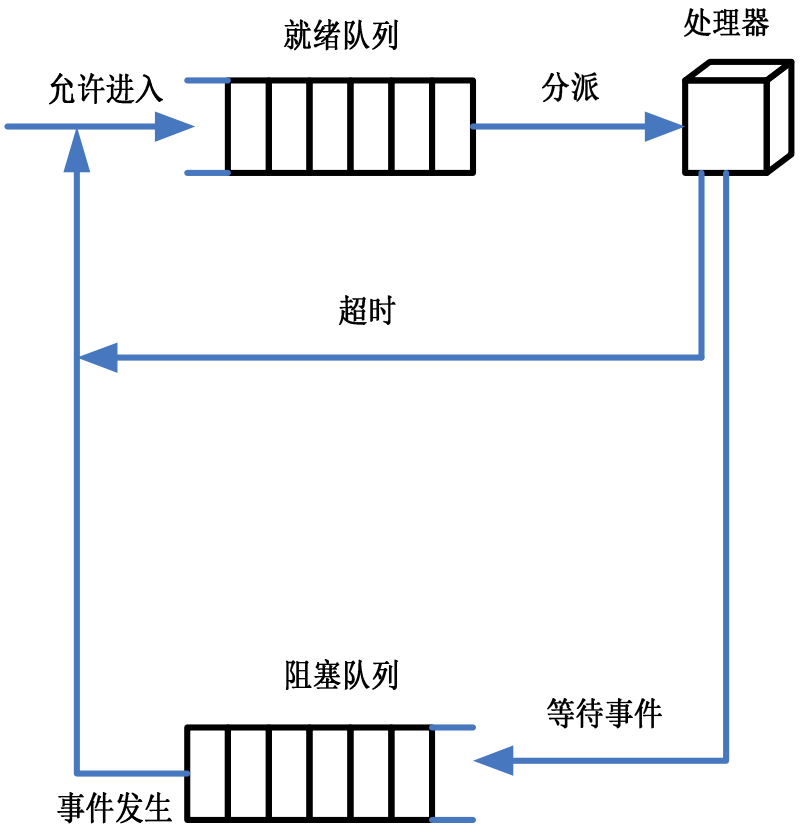
表3-3 状态迁移说明

状态迁移	说明
空→新建	创建一个新进程。创建新进程的事件可能有： ● 当终端用户登录到系统时，操作系统为该用户创建一个进程 ● 操作系统因为提供一项服务而创建一个进程 ● 由现有的进程派生。比如，基于模块化的考虑或为了开发并行性，用户程序可以指示创建多个进程
新建→就绪	操作系统准备好再接纳一个进程时，把一个进程从新建态转换到就绪态。大多数操作系统为了确保系统的性能，通常会限制系统中活跃进程的数量。因此当新建进程的加入仍然满足这种限制时，才允许进程被加载到内存中，从而进入就绪态。
就绪→运行	操作系统按照特定的调度策略选择一个就绪态的进程时，该进程就进入运行态

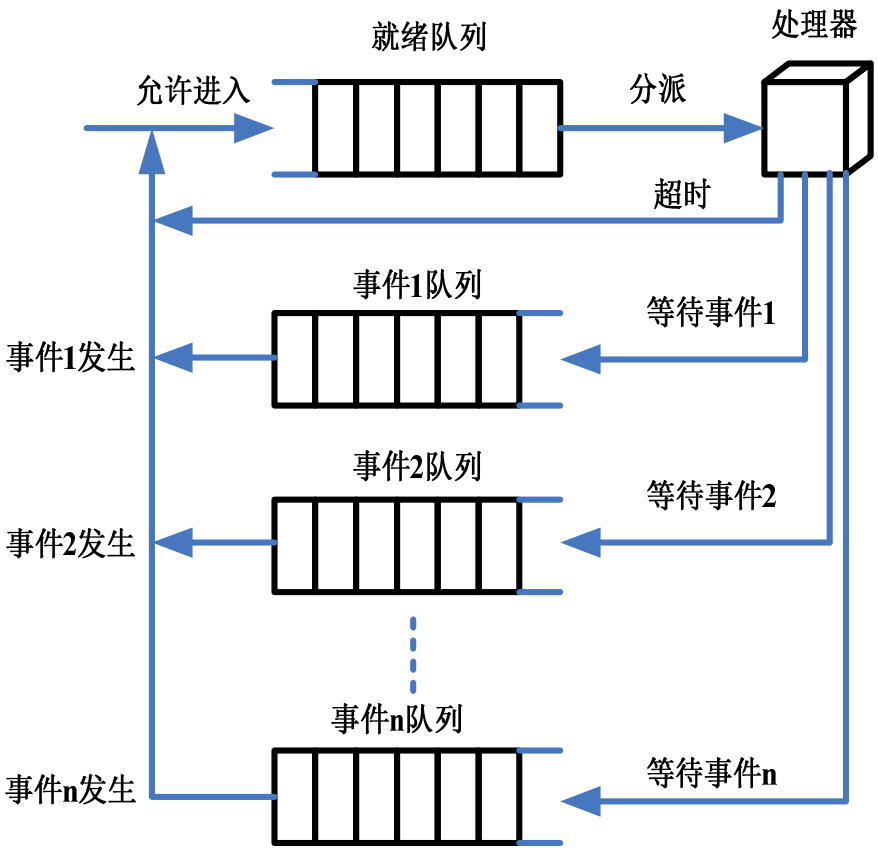
运行→退出	导致进程退出的事件有： ● 正常完成。进程自行执行一个指令（如Halt）或系统调用（如exit），表示它已经结束运行 ● 超过时限。进程运行时间超过规定的时限。时限的计量方法有多种类型，比如可以采用进程总的运行时间、花费在执行上的时间，或者对于交互进程，从上一次用户输入到当前时刻的总时间等。 ● 无可用内存。系统无法满足进程需要的内存空间 ● 发生越界、保护错误、算术错误等，或执行无效指令、特权指令、数据误用等时 ● 进程等待某一事件发生的时间超出了规定的最大值 ● I/O失败时。在输入或输出期间发生错误，如找不到文件、在超过规定的最多努力次数后，仍然读写失败（如遇到磁带上的一个坏区时）或者操作无效（如从行式打印机中读） ● 操作员或操作系统终止进程（例如，如果存在死锁） ● 父进程终止。当一个父进程终止时，操作系统可能会自动终止该进程的所有后代进程 ● 父进程请求。父进程通常具有终止其任何后代进程的权利
运行→就绪	● 正在执行的进程达到了“允许不中断执行”的最大时限 ● 高优先级进程抢占了该进程 ● 进程自愿释放对处理器的控制，如执行sleep( )
运行→阻塞	通常是进程请求了操作系统的一个服务或调用之后进入阻塞状态。 ● 进程可能请求操作系统的一个服务，但操作系统无法立即予以服务 ● 请求了一个无法立即得到的资源，如文件或虚拟内存中的共享区域 ● 需要进行某种初始化工作，如I/O操作，而且只有当该初始化工作完成后才能继续执行 ● 当进程相互通信，一个进程等待另一个进程提供输入时，或等待来自另一个进程的信息时
阻塞→就绪	当所等待的事件发生时，由阻塞态迁移到就绪态
就绪→退出	在某些系统中

【例3-1】设系统中有三个进程P_1、P_2和P_3。某一时刻，每个进程可能处于上述五个状态之一，我们用<s_1,s_2,s_3>表示由这三个进程所构成的系统的状态，其中s_i表示进程P_i（i=1,2,3）的状态。如果这些进程相互独立，对于单处理器系统，那么可能存在多少个系统状态？对于双处理器系统，可能的系统状态又有多少个?

五状态模型的实现



a) 单一的阻塞队列

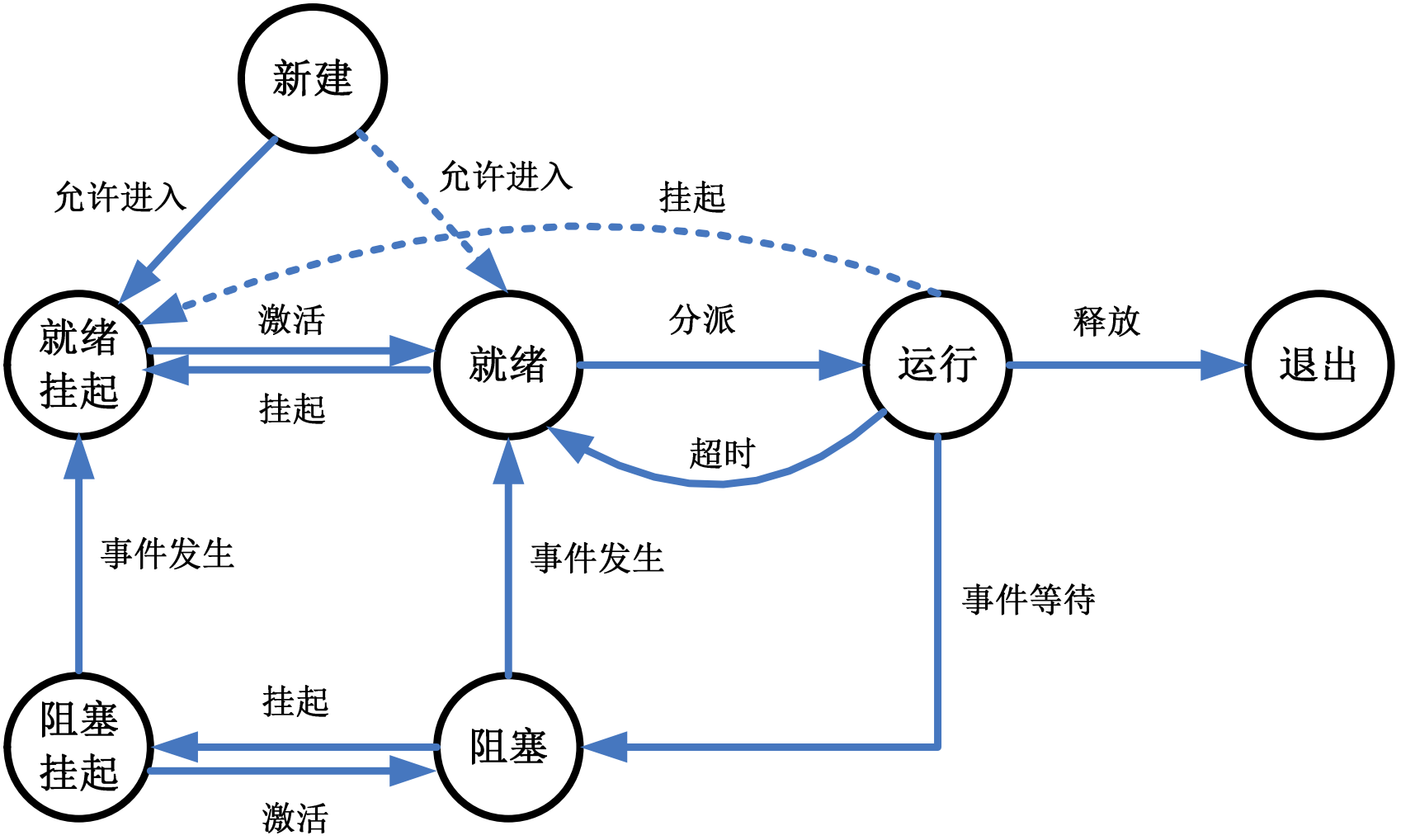


b) 多个阻塞队列

七状态模型

当一个进程被交换到磁盘中时，我们把它的状态称为“挂起”状态。使用挂起状态可以区分进程是在内存中还是在磁盘中。挂起状态与原来的就绪和阻塞状态一经组合，就出现了4种可能的状态：

- 就绪态：进程在内存中并可以执行
- 阻塞态：进程在内存中并等待一个事件
- 阻塞挂起：进程在外存中并等待一个事件
- 就绪挂起：进程在外存中，但是只要被载入内存就可以被调度执行


新建
允许进入
允许进入
挂起
就绪
挂起
激活
就绪
分派
运行
释放
退出
挂起
超时
事件发生
事件发生
事件等待
阻塞
挂起
阻塞
挂起
激活

状态迁移	说明
新建→就绪挂起	● 当内存中没有足够的空间分配给新进程 ● 操作系统为了维护大量未阻塞的进程，通常推迟创建进程以减少操作系统的开销，并在系统被阻塞态进程阻塞时，才执行进程创建任务。 在这两种情况下，都把一个新建进程放入就绪挂起队列。
新建→就绪	当新建进程的加入满足操作系统对内存、性能的各种限制时，允许进程被加载到内存中，从而进入就绪态
就绪挂起→就绪	● 如果内存中没有就绪态进程，操作系统需要调入一个就绪挂起态进程到内存中 ● 当处于就绪挂起态的进程比处于就绪态的任何进程的优先级高时
就绪→就绪挂起	通常，操作系统更倾向于挂起阻塞态进程而不是就绪态进程，因为就绪态进程可以立即执行，而阻塞态进程占用了内存但不能执行。 ● 如果释放内存以得到足够空间的唯一方法是挂起一个就绪态进程时 ● 如果操作系统确信高优先级的阻塞态进程很快将会就绪，那么它可能挂起一个低优先级的就绪态进程，而不是高优先级的阻塞态进程
阻塞→阻塞挂起	● 如果没有就绪进程，则至少一个阻塞进程被换出，为新建进程或就绪挂起进程让出空间 ● 操作系统为了维护基本的性能要求而需要更多的内存空间时，即使有可用的就绪态进程，也需要把一个或多个阻塞进程换出内存
阻塞挂起→阻塞	该迁移在设计中比较少见，因为如果一个进程没有准备好执行，并且不在内存中，那么把它调入内存没有什么意义。但是考虑到下面情况： ● 一个进程终止，释放了一些内存空间，阻塞挂起队列中有一个进程比就绪挂起队列中的任何一个进程的优先级都高，并且操作系统确信阻塞进程的事件很快就会发生，这时，把阻塞进程而不是就绪进程调入内存是合理的
阻塞挂起→就绪挂起	等待的事件发生时
运行→就绪挂起	通常，当分配给一个运行进程的时间片到期时，它将转换到就绪态。但是在某些情况下，操作系统为了释放一些内存，也可以直接把这个进程从运行态转换为就绪挂起态。

导致进程挂起的原因

事件	说明
交换	操作系统需要释放足够的内存空间，以调入并执行处于就绪状态的进程
其它OS原因	操作系统可能挂起后台进程或工具程序进程，或者被怀疑导致问题的进程
交互式用户请求	为了调试或者与一个资源的使用进行连接，用户可能需要挂起一个程序的执行
定时	一个进程可能会周期性的执行（如记账或系统监视进程），大多数情况下是空闲的，则在它两次使用之间应该被换出，即在等待下一个时间间隔时被挂起
父进程请求	有时，父进程会挂起后代进程的执行，以检查或修改挂起的进程，或者协调不同后代进程之间的行为

- 进程控制——操作系统如何管理进程生命周期中的一系列活动，如创建、切换和退出

操作系统创建一个进程的步骤：

- 为新进程分配一个唯一的进程标识符pid，在主进程表中增加一个新表项，每个表项对应一个进程。
- 给该进程各组成部分分配地址空间。
- 给进程控制块PCB分配空间。
- 初始化进程控制块。进程标识符部分包括进程ID和其它相关的ID，如父进程的ID等；除了程序计数器（被置为程序入口点）和系统栈指针（用来定义进程栈边界）之外，处理器状态信息部分的大多数条目通常初始化为0。
- 设置正确的连接。例如，如果操作系统把每个调度队列都保存成链表，则新进程必须放置在就绪或就绪挂起链表中。
- 创建或扩充其它数据结构。例如，操作系统可能为每个进程保存着一个记账文件，可用于编制账单和/或进程性能评估。

操作系统退出一个进程的步骤：

- 根据退出进程ID号，从相应队列找到它的PCB；
- 将该进程拥有的资源归还给父进程或操作系统；
- 若该进程拥有子进程，则先退出它的所有子孙进程，以防它们脱离控制；
- 将进程出队，释放它的PCB。

进程切换——操作系统打断一个正在运行的进程，把处理器指派给另一个进程，让其拥有处理器资源并开始或继续执行的过程。

进程切换 != 状态转换

进程切换与进程状态转换的关系：

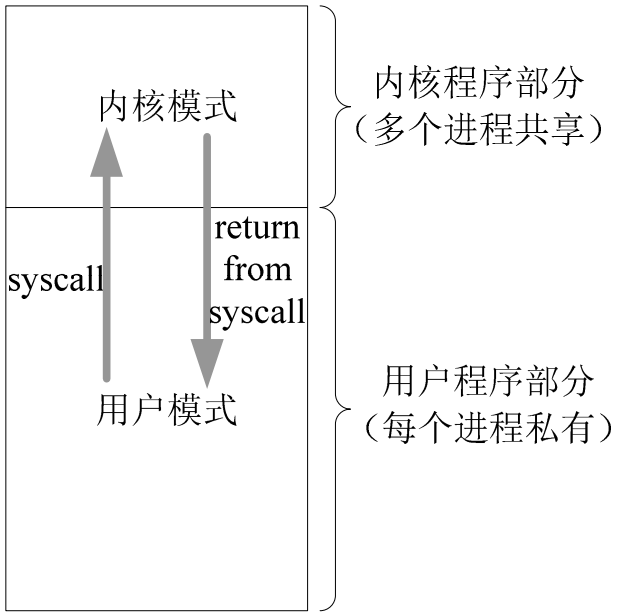
◆ 进程切换过程中一定伴随着多个进程的状态转换，

◆状态转换仅仅是进程切换中的一个活动，除此之外，还有一些其它必要活动。

一般的，一个进程上下文切换应包括如下几个活动：

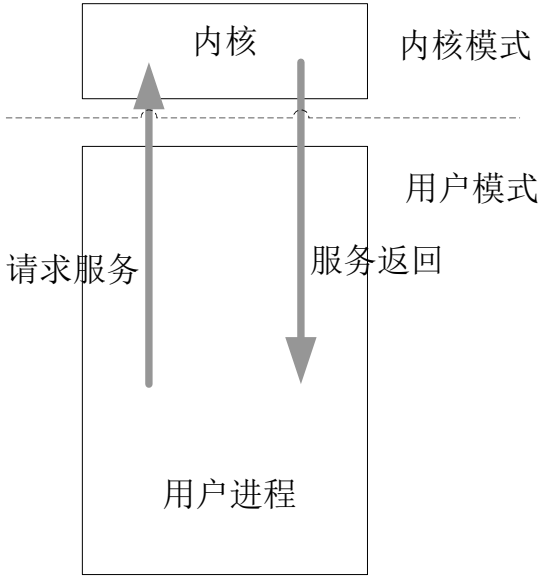
- 把处理器状态信息（包括程序计数器和其它寄存器、栈指针等）保存在进程控制块中。
- 更新当前该进程的进程控制块，包括将进程的状态改变到另一状态（就绪态、阻塞态或退出态）。还必须更新其它相关域，包括离开运行态的原因和记账信息等。
- 将该进程的进程控制块移到相应的队列。
- 选择另一个进程执行。究竟选择哪个就绪的进程执行，取决于操作系统所采用的调度算法。
- 更新所选择进程的进程控制块，包括将进程的状态变为运行态。
- 更新内存管理的数据结构，这取决于如何管理地址转换，这方面内容将在内存管理章节介绍。
- 使用被选择的进程最近一次切换出运行态时的上下文环境，恢复处理器状态。这可以通过载入程序计数器和其它寄存器以前的值来实现。

进程切换与模式切换的关系——依赖于操作系统的结构



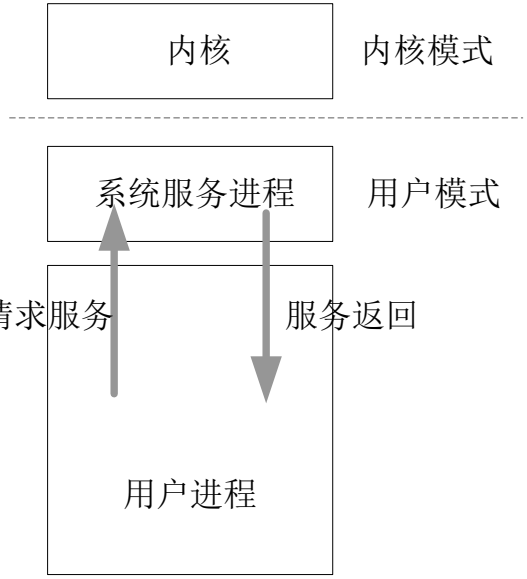
进程虚拟地址空间

在基于虚拟内存的操作系统中



进程上下文切换伴随
模式切换

老的操作系统中



只有进程上下文切换

微内核操作系统中

进程切换的时机——异常事件的发生是操作系统获取处理器的控制权，实施进程切换的唯一时机。

- 中断异常发生时，如时钟中断，I/O中断等
- 陷阱异常发生时
- 故障和致命故障发生时。

过程调用和系统调用的区别：

◆ 过程调用主要是为了模块化程序设计的需要，将常用的、公共的程序部分封装为一个例程，方便用户的使用以及程序的维护；而系统调用是用户程序使用计算机服务的一种方式。在设计系统调用时，除了考虑模块化的需求之外，更重要的是考虑到对计算机资源的保护。

◆ 过程调用的发生是通过指令的跳转来实现的；而系统调用是通过发起陷阱异常来实现的，系统调用的代码是由异常处理程序来执行的。

◆ 过程调用不会触发进程切换和进程状态转换，而系统调用**可能会**引起进程切换和状态转换。比如，有关I/O的系统调用通常会使进程从运行态变换为阻塞态，同时CPU切换到其它就绪进程。

◆ 进程在用户模式下调用一个过程时，处理器始终处于用户模式，不会发生模式切换；而调用一个系统调用时，处理器将从用户模式切换到内核模式；当系统调用返回时，处理器又从内核模式切换回用户模式。

3.4 UNIX中的进程控制

获取进程ID

```
pid_t getpid(void);   //返回当前进程的PID
pid_t getppid(void);  //返回当前进程的父进程（即创建调用进程的进程）的PID
                      Returns: PID of either the caller or the parent
```

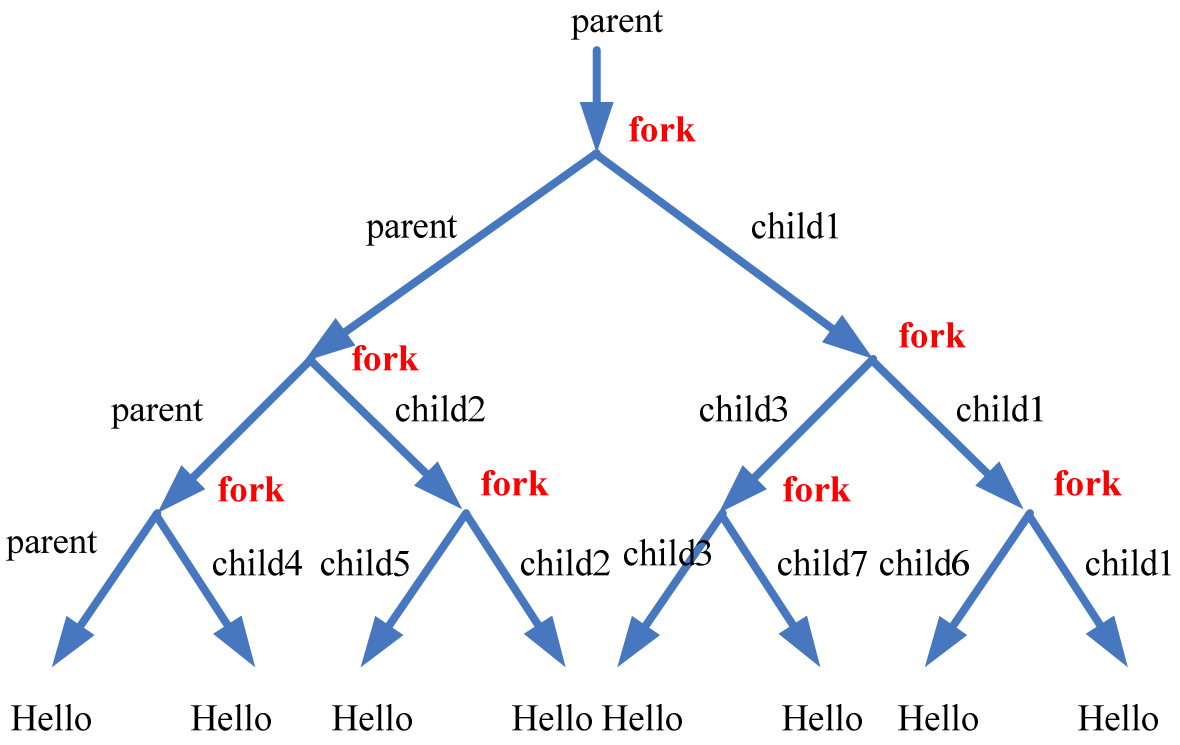
创建和退出一个进程

```
pid_t  fork(void);        Returns: 0 to child, PID of child to parent, -1 on error
                          调用一次，返回两次

void exit(int status);        This function does not return，调用但不返回值
```

```
1    int main()
2    {
3        pid_t pid;
4        int x=1;
5        pid=fork( );
6        if(pid==0){          /*Child*/
7            printf( “child : x=%d\n” , ++x);
8            exit(0);
9        }
10
11       /*Parent*/
12       printf( “parent : x =%d\n” , --x);
13       exit(0);
14   }
```

```
1  int main()
2  {
3      fork( );
4      fork( );
5      fork();
6      printf( “hello\n” );
7      exit(0);
8  }
```



```
#include <unistd.h>
int execve (const char *filename, const char *argv[], const char
                                                        *envp[]);
```

如果执行成功，不返回；如果出错，则返回-1

execve只有当出错时，才返回调用进程，否则将一直执行下去，不会返回。

因此，与fork不同，execve是“调用一次但从不返回”。

- 练习

1、进程是程序的一次运行，或者说，进程是程序的一个_____，是_____(静态/动态)的概念。

2、进程的五个部分中，____和____来自于可执行目标文件，____保存进程的控制信息，程序运行时动态分配的内存存放在_____。

3、32位Linux中，一个进程的虚拟地址空间大小为___，其中分配给用户进程的大小为___，程序入口点的虚拟地址为___。程序中一个字符指针变量的大小为___位，指针变量的值是地址，该地址是_____(虚拟地址/物理地址)。

4、一个正在运行的进程执行了一个I/O系统调用后，其状态将由运行态迁移到____状态；若在运行时发生了一个保护错误，那么状态将由运行态迁移到____状态。通常，操作系统更倾向于把一个处于____状态的进程挂起；操作系统进行调度时，通常选择处于____状态的进程来调度执行。

5、进程切换中通常伴随_____。在采用虚拟内存的操作系统中，当一个用户程序调用一个系统调用时，程序的执行流从用户空间转移到____空间，同时处理器的模式从用户模式切换到_____模式，但是程序仍然在进程的_____中运行。对于微内核结构的操作系统，当一个用户进程向一个系统服务进程发送一个请求服务的消息时，通常会引起该用户进程与_____进程的切换。

§3.5 进程调度策略

进程调度也称处理器调度，是指为了满足特定系统目标（如响应时间、吞吐率、处理器效率），操作系统把进程分派到一个或多个处理器中执行的过程。

- 如果执行进程的处理器只有一个，则称该调度为单处理器调度；
- 如果执行进程的处理器有多个，则称该调度为多处理器调度。进程调度是操作系统的核心功能之一，通过调度算法来实现。

调度问题实质上是一个最优化问题，即在给定的约束下，使系统的一个或多个目标函数值达到最大。系统的约束不同，采用的目标函数不同，那么采取的优化调度策略也就有所不同。

一般来说，进程调度目标可以分为两类：**面向用户**的目标和**面向系统**的目标。

- 面向用户的目标与单个用户或进程所能感知到的系统行为相关，如以响应时间作为调度目标。
- 面向系统的目标重点关注系统层面的优化目标，如吞吐量、处理器利用率、公平性和负载均衡等。

面向用户	
周转时间	指一个进程从提交到完成之间的时间间隔，包括实际执行时间加上等待资源（包括处理器资源）的时间。对批处理作业，这是用户很关心的一个调度目标。
响应时间	对一个交互进程，响应时间是指从提交一个请求到开始接收响应之间的时间间隔。通常进程在处理该请求的同时，就开始给用户产生一些输出，以缩短响应时间。以响应时间作为目标函数的调度策略应该在满足最低响应时间的情况下，使得可以交互的用户数目达到最大。
最后期限	指进程完成的最后时间点。
可预测性	是指：无论系统的负载如何，用户总是希望响应时间或周转时间的变化不太大，总是被控制在一定范围内。为了达到该目标，需要在系统工作负载大范围抖动时发出信号或者需要系统处理不稳定性。

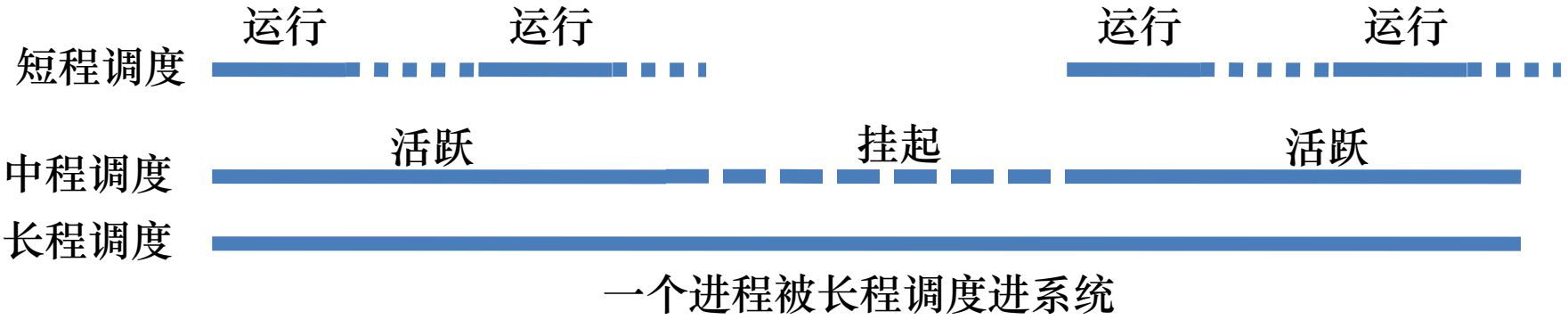
面向系统	
并发度	是指等待处理器执行的进程的个数
吞吐量	是指单位时间内完成的进程数目，它取决于进程的平均执行长度，也受调度策略的影响。
处理器利用率	指处理器处于忙状态的时间比例。对昂贵的计算机系统来说，这是一个主要目标，但对于单用户系统和实时系统，该目标相对来说并不太重要。
公平性	是指在没有来自用户的指示或其它系统提供的指示时，进程应该被平等对待，没有一个进程会处于饥饿状态。
资源平衡	是指调度策略将保持系统中所有资源处于繁忙状态，较少使用紧缺资源的进程应该受到照顾。

进程调度分为三个层次：

•**长程调度**。决定是否将一个新建进程添加到当前活跃或挂起的进程集合中。创建一个新进程时，执行长程调度；

•**中程调度**。中程调度是内存管理功能的一部分，它决定是否把一个进程的部分或全部虚拟地址空间换入内存（即把进程从挂起态变换到活跃态），或者把一个进程从内存中换出到磁盘上（即把进程从活跃态变换到挂起态）。

•**短程调度**。决定把处理器分派给哪个进程，即在活跃就绪进程队列中，如何选择一个进程去执行。



一个进程被长程调度进系统

我们主要介绍短程调度，即调度程序如何从进程就绪队列中选择一个进程去执行。必须考虑：

- 何时执行短程调度程序——执行短程调度的时机只有一个，就是异常发生时。
- 如何从就绪队列中选择一个进程去执行——调度策略。

进程调度与进程切换的关系？

- 进程切换是进程调度的结果，或者说，
- 进程切换依赖于进程调度。
- 进程调度和进程切换不可分割，因此二者发生的时机是一致的。

短程调度策略

选择函数（selection function）确定在就绪进程中选择哪一个进程来执行。这个函数可以基于优先级、资源需求或者进程的执行特性。进程的执行特性主要由这几个量来度量：

- *w*：到目前为止，进程在系统中花费的等待时间
- *e*：到目前为止，进程已执行的时间
- *s*：进程所需要的总服务时间。通常*s*由估计得到或由用户提供。
- *T*：周转时间。进程在系统中花费的总时间，即等待时间和服务时间的总和：

$$T = s + w。$$

- *T*/*s*：归一化周转时间（turnaround time），即进程在系统中驻留的相对时间。使用这个量可以比较服务时间长短不同的进程的相对周转时间。

决策模式（Decision mode），通常可分为以下两类：

- **非抢占式**。在这种情况下，一旦进程处于运行状态，它就一直执行直到终止，或者因为等待I/O或请求某些操作系统服务而阻塞。也就是说，对于非抢占式调度策略，一个正在运行的进程不能被抢占，除非它终止，或者进入阻塞状态。
- **可抢占式**。当前正在运行的进程可能被其它进程抢占，并转移到就绪状态。抢占发生的时机主要包括：一个新进程到达时；一个中断发生后把一个阻塞态的进程置为就绪状态时；或者基于周期性的时钟中断，如时间片到期时。

类别	选择函数	决策模式	吞吐量	响应时间	开销	对进程的影响	饥饿
FCFS	max[*w*]	非抢占	不强调	可能很高，特别是当进程的执行时间差别很大	最小	对短时间进程（简称短进程）不利；对I/O密集型的进程不利	无
轮转	常数	抢占（在时间片用完时）	如果时间片小，吞吐量会很低	为短进程提供好的响应时间	最小	公平对待	无
SPN(最短进程优先)	min[*s*]	非抢占	高	为短进程提供好的响应时间	可能比较高	对长时间进程（简称长进程）不利	可能
SRT(最短剩余时间优先)	min[*s*-*e*]	抢占（在到达时）	高	提供好的响应时间	可能比较高	对长进程不利	可能
HRRN(最高响应比优先)	max[(*w*+*s*)/*s*]	非抢占的	高	提供好的响应时间	可能比较高	很好的平衡	无

【例3-3】设有5个进程P1~P5，其到达时间和服务时间分别如下表所示。

试分析FCFS和轮转调度策略下，每个进程的周转时间、归一化周转时间以及平均周转时间。其中，轮转策略分为时隙长度q为1和4两种情况考虑。

进程	P1	P2	P3	P4	P5
到达时间	0	2	4	6	8
服务时间s	3	6	4	5	2

进程	P1	P2	P3	P4	P5	平均值
到达时间	0	2	4	6	8	
服务时间s	3	6	4	5	2	
			FCFS			
完成时间	3	9	13	18	20	
周转时间	3	7	9	12	12	8.60
归一化周转时间	1.00	1.17	2.25	2.40	6.00	2.56

进程	P1	P2	P3	P4	P5	平均值
到达时间	0	2	4	6	8	
服务时间s	3	6	4	5	2	
			q=1			
完成时间	4	18	17	20	15	
周转时间	4	16	13	14	7	10.80
归一化周转时间	1.33	2.67	3.25	2.80	3.50	2.71
			q=4			
完成时间	3	17	11	20	19	
周转时间	3	15	7	14	11	10.00
归一化周转时间	1.00	2.5	1.75	2.80	5.50	2.71

§3.6 线程

实际上，进程的概念包含两方面抽象：

•环境和资源的抽象。进程为程序的执行提供了一个环境或上下文。程序执行需要的所有部件都被布局在进程虚拟地址空间中。另外，程序执行所需要的各种资源，如内存、打开文件和I/O设备等，通过进程控制块中的特定属性与进程关联起来。

• 程序执行流的抽象。进程中还包括一个处理器执行程序的流程。进程控制块中保存了处理器的状态信息、进程的调度状态信息等，操作系统使用这些信息来调度进程以及让处理器正确的执行进程。

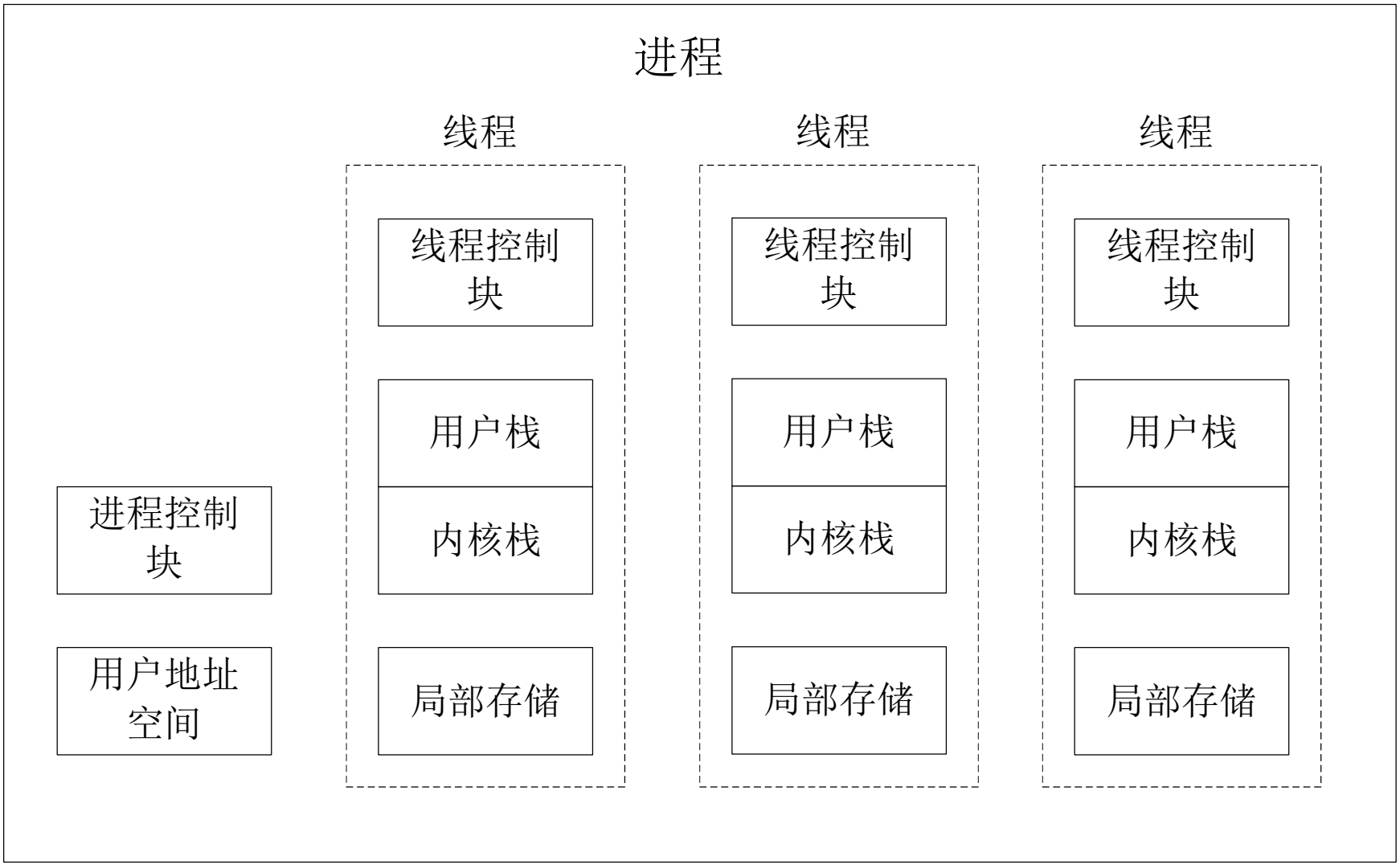
• 上述两方面抽象是独立的，有必要将它们分离开来单独进行考虑。把有关程序执行流方面的抽象从进程概念中分离出去，就形成了线程的概念，而进程的概念主要集中在环境和资源的抽象方面。

进程与运行在其上的线程之间的关系就不局限于1：1。一个进程中可能有一个或多个线程，每个线程也具有特定的结构，线程的结构中包括如下信息：

•**线程控制块**：与进程控制块类似，包含了描述线程属性的信息，如线程ID、线程的栈指针、程序计数器、条件码和通用寄存器的值等，以及线程的执行状态（运行、就绪等）。

•**线程执行栈**：保存一个线程执行过程中的活动记录，包括用户栈和内核栈。其中用户栈用于保存过程调用的活动记录，内核栈用于保存系统调用的活动记录。执行栈对于每个线程来说都是私有的，因此不同线程的执行流不会发生相互干扰。

• **线程局部存储**（TLS，Thread Local Storage）：是某些操作系统为线程单独提供的私有空间，用于存储每个线程私有的全局变量，即一个线程内部的各个过程调用都能访问、但其他线程不能访问的变量。


进程
线程
线程
线程
线程控制块
线程控制块
线程控制块
用户栈
用户栈
用户栈
内核栈
内核栈
内核栈
局部存储
局部存储
局部存储
进程控制块
用户地址空间

在一个进程中创建多个线程具有以下优点：

•在一个已有进程中创建一个线程比创建一个新进程所需的时间开销要少许多。一些研究表明，创建一个线程要比创建一个进程快10倍。

•终止一个线程比终止一个进程花费的时间少。

•同一进程内线程间的切换比进程间的切换花费的时间少。

• 线程提高了程序间通信的效率。在大多数操作系统中，独立进程间的通信需要内核的介入，以提供保护和通信所需要的机制。但是，由于在同一个进程中的线程共享内存和文件，它们无需调用内核就可以相互通信。

线程与进程的关系

线程:进程	描述	示例系统
1:1	一个进程中只有一个线程，或者说一个线程就是一个进程	传统的UNIX
M:1	可以在一个进程中创建和执行多个线程，所有这些线程共享进程用户地址空间	Windows NT, Solaris, Linux, OS/2, MACH
1:M	一个线程可以从一个进程环境迁移到另一个进程环境	RS(Clouds), Emerald
M:N	结合了M:1和1:M情况	TRIX

```
1  #include “csapp.h”
2  #define N 2
3  void *thread(void *vargp);
4
5  char **ptr;     /*Global variable*/
6
7  int main()
8  {
9          int   i;
10         pthread_t tid;
11         char *msgs[N]={“Hello from foo”, “Hello from bar”};
12
13         ptr = msgs;
14         for (i=0; i<N; i++)
15             pthread_create(&tid, NULL, thread, (void*)i);
16         pthread_exit(tid);
17 }//main
18
19 void *thread(void* vargp)
20 {
21         int myid = (int)vargp;
22         static int cnt=0;
23         printf(“[%d]: %s (cnt=%d)\n”, myid, ptr[myid], ++cnt);
24         return NULL;
25 }//thread
```

- 作业　P85

4、5、13