进程与线程

一、进程与线程

1. 进程的概念和特征

(1) 进程的概念

引入进程 (Process) 的概念，以便更好地描述和控制程序的并发执行，**实现操作系统的并发性和共享性**(最基本的两个特性)。

从不同角度，进程有不同的定义：

- 进程是一个正在执行程序的实例
- 进程是一个程序及其数据从磁盘加载到内存后，在CPU 上的执行过程。
- 进程是一个程序及其数据从磁盘加载到内存后，在CPU 上的执行过程。

进程控制块 (Process Control Block,PCB)：

- 系统利用PCB来描述进程的基本情况和运行状态，进而管理和控制进程。

进程实体（进程映像）

- 由程序段，相关数据段，PCB三部分组成

创建进程，就是创建进程的PCB，撤销进程，就是撤销进程的PCB

传统操作系统中的进程定义为：“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。

> 系统资源：它指 CPU 、存储器和其他设备服务于某个进程的“时 间”,例如将CPU 资源理解为CPU 的时间片。

> 因为进程是这些资源分配和调度的独立 单位，即“时间片”分配的独立单位，这就决定了进程一定是一个动态的、过程性的概念。

（2）进程的特征

程序是静态的， 进程是动态的

- 动态性：（最基本的特征）
  - 进程的一次执行，有创建，活动，暂停，终止等进程，具有生命周期，动态产生，变化，消亡
- 并发性：
  - 指多个进程同存于内存中，能在一段时间内同时运行
  - 并发性是进程的重要特征，也是操作系统的重要特征。
  - **宏观的并行，微观的串行**
- 独立性：
  - 指进程是一个能独立运行、独立获得资源和独立接受调度的基本单位。
- 异步性：
  - 由于进程的相互制约，使得进程按各自独立的、不可预知的速度向前推进。
  - 异步性会导致执行结果的不可再现性，为此在操作系统中必须配置相应的进程同步机制。

2. 进程的组成

（1）进程控制块

PCB 是进程实体的一部分，是进程存在的唯一标志。

在进程的整个生命期中，系统总是通过PCB 对 进程进行控制的，亦即系统唯有通过进程的PCB 才能感知到该进程的存在。

PCB中包括：

表 2.1　PCB 通常包含的内容

进程描述信息	进程控制和管理信息	资源分配清单	处理机相关信息
进程标识符（PID）	进程当前状态	代码段指针	通用寄存器值
用户标识符（UID）	进程优先级	数据段指针	地址寄存器值
	代码运行入口地址	堆栈段指针	控制寄存器值
	程序的外存地址	文件描述符	标志寄存器值
	进入内存时间	键盘	状态字
	CPU 占用时间	鼠标	
	信号量使用		

> 处理机相关信息：也称CPU上下文，主要是CPU中各种寄存器的值

常用的PCB组织方式：

- 链接方式：将同一状态的PCB链接城一个队列，不同状态对应不同队列，也可以将处于阻塞态的PCB根据阻塞原因的不同，排成多个阻塞队列。
- 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，不同状态对应不同索引表，如就绪索引表，阻塞索引表。

（2）程序段

能被进程调度程序调度到CPU执行的程序代码段。

程序可以被多个进程共享

（3）数据段

进程对应的程序加工处理的原始数据

或程序执行时产生的中间结果或最终结果

3. 进程的状态与切换

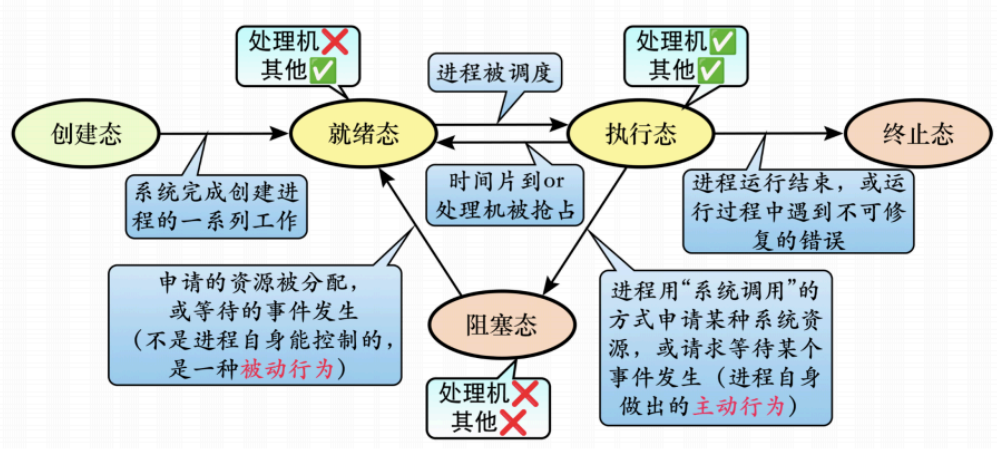
（1）进程的五种状态

通常进程有以下5种状态，前3种是进程的基本状态。

- **运行态**：
  - 进程正在CPU 上运行。在单CPU 中，每个时刻只有一个进程处于运行态。
- **就绪态：**
  - 进程获得了除CPU 外的一切所需资源，一旦得到CPU，便可立即运行。
  - 系统中 处于就绪态的进程可能有多个，通常将它们排成一个队列，称为就绪队列。
- **阻塞态：**
  - 等待态，进程正在等待某一时间而暂停运行
  - 如等待某个资源可用（不包括CPU）或等待I/O完成，即使CPU空闲，该进程也不能运行
  - 根据阻塞原因的不同，设置多个阻塞队列
- **创建态：**
  - 进程正在被创建，尚未转到就绪态。
  - 进程创建需要多步：
    - 申请空白PCB，向PCB中填写用于控制和管理进程的信息
    - 为进程分配运行时所必须的资源
    - 将进程转入就绪态，并插入就绪队列
  - 进程所需的资源尚不能得 到满足，如内存不足，则创建工作尚未完成，进程此时所处的状态称为创建态。
- **终止态：**
  - 进程正从系统中消失，可能是进程正常结束或其他原因退出运行。
  - 系统先将该进程设置为终止态，进一步处理资源释放和回收工作

> 区分就绪态和阻塞态：
>
> - 就绪态是指进程仅缺少CPU，只要获得 CPU 就立即运行；
> - 而阻塞态 是指进程需要其他资源(除了CPU) 或等待某一事件
>
> 进程在运行过程中实际上是频繁地转换到就绪态的；
>
> 而其他资源(如外 设)的使用和分配或某一事件的发生(如 I/O 完成)对应的时间相对来说很长，进程转换到阻塞 态的次数也相对较少。
>
> 这样来看，就绪态和阻塞态是进程生命周期中两个完全不同的状态。

（2）引起进程状态转换的事件



就绪态 → 运行态	● 触发条件：进程被调度程序选中。 ● 说明：进程获得了处理机（CPU）资源，开始执行。
运行态→ 就绪态	● 触发条件：时间片用完，进程分配的时间片耗尽，必须让出处理机。 ● 被抢占：可剥夺操作系统中，有更高优先级的进程进入就绪队列时，调度程序会将当前正在执行的进程转换为就绪态，让位给高优先级进程。
运行态 → 阻塞态（主动行为）	● 触发条件：进程请求等待某个事件或资源。 ● 等待系统分配资源（如外设、缓冲区）。 ● 等待某事件发生（如 I/O 操作完成）。 ● 说明：这是进程自身的主动行为（如发起系统调用）。
阻塞态 → 就绪态（被动行为）	● 触发条件：资源分配到位或等待的事件已发生。 ● 说明：通常由操作系统（OS）发起（如 I/O 中断处理程序），进程被动地被移入就绪队列，等待下一次调度。
不可逆转的路径	● 阻塞态 ↛ 运行态：进程在阻塞解除后必须先进入就绪队列，不能直接获得 CPU。 ● 就绪态 ↛ 阻塞态：因为就绪态进程尚未获得 CPU 执行权，无法发出导致阻塞的请求（如 I/O 请求）。
终止路径	● 任何状态 → 终止态：进程可以从任何状态直接转换为终止态（例如父进程终止子进程，或发生致命错误）。

4. 进程控制

进程控制的主要功能是对系统中的所有进程实施有效的管理，它具有创建新进程、撤销已有 进程、实现进程状态转换等功能

在操作系统中，一般将进程控制用的程序段称为原语，原语的 特点是执行期间不允许中断，它是一个不可分割的基本单位。

（1）进程的创建

a. 父进程与子进程的关系和特点

允许一个进程创建另一个进程，此时创建者称为父进程，被创建的进程称为子进程。

子进程 可以继承父进程所拥有的资源。

子进程终止时，应将其从父进程那里获得的资源还给父进程。

b. 导致创建进程的操作

在操作系统中，终端用户登录系统、作业调度、系统提供服务、用户程序的应用请求等都会 引起进程的创建。

C.创建进程的操作

- 为新进程分配一个唯一的进程标识号，并申请一个空白PCB(PCB 是有限的)
- 为进程分配其运行所需的资源，如内存、文件、 I/O 设备和CPU 时间等(在PCB 中体 现)。
  - 些资源或从操作系统获得，或仅从其父进程获得。
  - 若资源不足(如内存),则并 不是创建失败，而是处于创建态，等待内存资源。
- 初始化PCB，主要包括初始化标志信息、初始化CPU 状态信息和初始化CPU 控制信息， 以及设置进程的优先级等。
- 若进程就绪队列能够接纳新进程，则将新进程插入就绪队列，等待被调度运行。

（2）进程的终止

a. 终止进程的事件

- 正常结束，表示进程的任务已完成，并退出运行
- 异常 结束：
  - 表示进程在运行时，发生了某种异常事件，使程序无法继续运行，如存储区越界、保护错、非法指令、特权指令错、运行超时、算术运算错、I/O 故障等
- 外界干预：
  - 指进程应外界的请 求而终止运行，如操作员或操作系统干预、父进程请求和父进程终止。

b. 终止进程的操作

作系统终止进程的过程如下(终止原语):

1. 根据被终止进程的标识符，**检索出该进程的PCB,** 从中读出该进程的状态。
2. 若被终止进程**处于运行状态，**立即终止该进程的执行，将CPU 资源分配给其他进程。
3. 若该进程还有子孙进程，则通常需将其所有**子孙进程终止**(有些系统无此要求)。
4. 将该进程所拥有的全部**资源**，或归还给其父进程，或归还给操作系统。
5. 将 该PCB 从所在**队列(链表)中删除。**

（3）进程的阻塞和唤醒

a. 进程阻塞的事件与时机

正在执行的进程，由于期待的某些事件未发生，如请求系统资源失败、等待某种操作的完成、 新数据尚未到达或无新任务可做等，进程便通过调用阻塞原语(Block)，使自己由运行态变为阻 塞态。（主动）

阻塞原语的执行过程：

- 找到将要被阻塞进程的标识号 (PID) 对应的PCB。
- 若该进程为运行态，则保护其现场，将其状态转为阻塞态，停止运行。
- 将该PCB 插入相应事件的等待队列，将CPU 资源调度给其他就绪进程。

b. 进程唤醒的事件与时机

当被阻塞进程所期待的事件出现时，如它所期待的I/O 操作已完成或其所期待的数据已到达， 由有关进程(比如，释放该I/O 设备的进程，或提供数据的进程)调用唤醒原语 (Wakeup)， 将 等待该事件的进程唤醒。

唤醒原语的执行过程：

- 在该事件的等待队列中找到相应进程的PCB。
- 将其从等待队列中移出，并置其状态为就绪态。
- 将该PCB 插入就绪队列，等待调度程序调度。

> Block和Wakeup必须成对使用。
>
> 若在某 个进程中调用了 Block 原语，则必须在与之合作的或其他相关的进程中安排一条相应的 Wakeup 原语，以便唤醒阻塞进程；
>
> 否则，阻塞进程将因不能被唤醒而永久地处于阻塞态。

5. 进程的通信

指进程之间的信息交换。

低级通信方式：信号量机制，管程，PV操作

高级通信方式：以较高效率传输大量数据

a. 共享存储

在进程之间存在一块可直接访问的共享空间，通过对这片共享空间进行读/写操作实现 进程之间的信息交换。

在对共享空间进行读/写操作时，需要使用同步互斥工具(如 P 操作、V 操作)对共享空间的读/写进行控制。

共享存储分为两种：

- 低级方式：基于数据结构的共享
- 高级方式：基于存储区的共享

操作系统只负责为通信进程提供可共享 使用的存储空间和同步互斥工具，而数据交换则由用户自己安排读/写指令完成。

进程空间是独立的，进程运行期间不能访问其他进程的空间。

进程内的线程自然共享进程空间

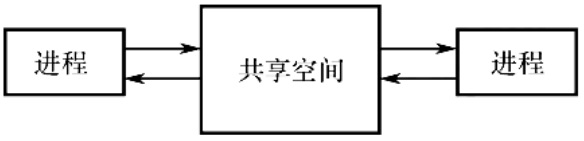


图 2.2　共享存储

b. 消息传递

进程之间不存在可直接访问的共享空间，消息交换以格式化的消息为单位。

进 程通过操作系统提供的发送消息和接收消息两个原语进行数据交换。

- 直接通信方式：
  - 发送进程直接将消息发送给接收进程，并将它挂在接收进程的消息缓冲 队列上，接收进程从消息缓冲队列中取得消息

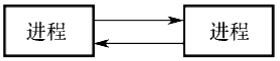


图 2.3　消息传递

- 间接通信方式：
  - 发送进程将消息发送到某个中间实体，接收进程从中间实体取得消息。 这种中间实体一般称为信箱。

啊啊啊啊啊啊啊啊啊啊啊啊啊啊啊啊啊啊啊啊啊啊好难

c. 管道通信

管道是一个特殊的共享文件，也称pipe 文件，数据在管道中是先进先出的。

管道通信允许 两个进程按生产者-消费者方式进行通信

为了协调双方的通信，管道机 制必须提供三方面的协调能力：

- 互斥，指当一个进程对管道进行读/写操作时，其他进程必须 等待。
- 同步
  - 指写进程向管道写入一定数量的数据后，写进程阻塞，直到读进程取走数据后 再将它唤醒
  - 读进程将管道中的数据取空后，读进程阻塞，直到写进程将数据写入管道后才将 其唤醒。
- 确定对方的存在。

d. 信号

信号 (Signal) 是一种用于通知进程发生了某个事件的机制。不同的系统事件对应不同的信 号类型，每类信号对应一个序号。

在进程的PCB 中，用至少n 位向量记录该进程的待处理信号，并且记录信号的处理情况。

信号的发送方式：

- 内核给某个进程发送信号。当内核检测到某个特定的系统事件时，就给进程发送信号。
- 一个进程给另一个进程发送信号。
  - 进程可以调用kill 函数，要求内核发送一个信号给目的 进程(需要指明接收进程的PID 和信号的序号)。
  - 进程也可给自己发送信号。

> 当操作系统把一个进程从内核态切换到用户态时(如系统调用返回时),会检查该进程是否 有未被阻塞的待处理信号，
>
> 若有，则强制进程接收信号，并立即处理信号
>
> 若有多个待处理信号， 则通常先处理序号更小的信号

信号处理方式：

- 执行默认的信号处理程序。操作系统为每类信号预设了默认的信号处理程序。
- 执行进程定义的信号处理程序。程可为某类信号自定义信号处理程序。

> 信号处理程序运行结束后，通常会返回进程的下一条指令继续执行。

6. 线程和多线程模型

（1）线程的基本概念

引入进程的目的：更好地使多道程序并发执行，提高资源利用率和系统吞吐量

引入线程的目的：减小程序在并发执行时所付出的时空开销，提高操作系统的并发性能。

线程是一个基本的CPU执行单元，也是程序执行流的最小单元，由线程ID，程序计数器，寄存器集合和堆栈组成。

线程是进程中的一个实体，被系统独立调度和分派的单位，线程自己不拥有系统资源，只拥有一点儿在运行中必不可少的资源， 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。

> 引入线程后：进程只作为除CPU外的系统资源的分配单位，线程作为CPU的分配单元。

（2）线程与进程的比较

- **调度：**
  - 没有线程时，每次调度都发生上下文切换，而引入线程后，线程切换的代价远小于进程，同一进程中的线程切换，不会引起进程切换
- **并发性：**
  - 在引入线程的操作系统中，线程和一个进程中的线程也可以并发执行，不同进程线程的线程也可以并发执行，提高系统资源的利用率和系统吞吐量。
- **拥有资源：**
  - 线程不拥有系统资源，（仅有一点必不 可少、能保证独立运行的资源,但线程可以访问其隶属进程的系统资源），这主要表现在 属于同一进程的所有线程都具有相同的地址空间。
- **独立性：**
  - 每个进程都有独立的地址空间和资源，除了共享全局变量，不允许其他进程 访问。同一进程中的线程共享地址空间和资源
- **系统开销：**
  - 操作系统创建进程的开销远比创建线程大，切换时开销也大，通信开销比较小。
- **支持多处理器系统：**
  - 对于多线程进程，可将多个线程分配到多个CPU上执行

（3）线程的属性

- 线程是一个轻型实体，它不拥有系统资源，但每个线程都应有一个唯一的标识符和一个 线程控制块，线程控制块记录线程执行的寄存器和栈等现场状态。
- 不同的线程可以执行相同的程序
- 同一进程中的各个线程共享该进程所拥有的资源
- 线程是CPU 的独立调度单位，多个线程是可以并发执行的。
- 一个线程被创建后，便开始了它的生命周期，直至终止。

> 由于有了线程，线程切换时， 有可能发生进程切换，也有可能不发生进程切换，平均而言每次切换所需的开销就变小了，因此 能够让更多的线程参与并发，而不会影响到响应时间等问题。

（4）线程的状态和转换

三种基本状态：

- 执行态：线程已获得 CPU 而正在运行。
- 就绪态：线程已具备各种执行条件，只需再获得 CPU 便可立即执行。
- 阻塞态：线程在执行中因某事件受阻而处于暂停状态。

（5）线程的组织和控制

a. 线程控制块

系统也为每个线程配置一个线程控制块 `TCB`，用于记录控制和管理线程的信息。

包括：

- 线程标识符；
- 一组寄存器，包括程序计数器、状态寄存器和通 用寄存器；
- 线程运行状态，用于描述线程正处于何种状态；
- 优先级；
- 线程专有存储区， 线程切换时用于保存现场等；
- 堆栈指针，用于过程调用时保存局部变量及返回地址等。

每一个线程都拥有自己的堆栈，且互不共享

b. 线程的创建

在操 作系统中就有用于创建线程和终止线程的函数(或系统调用)。

- 用户程序启动时，通常仅有一个称为初始化线程的线程正在执行，其主要功能是用于创建新 线程。
- 在创建新线程时，需要利用一个线程创建函数，并提供相应的参数，如指向线程主程序的 入口指针、堆栈的大小、线程优先级等。

c. 线程的终止

当一个线程完成自己的任务后，或线程在运行中出现异常而要被强制终止时，由终止线程 调用相应的函数执行终止操作。

被终止但尚未释放资源的线程仍可被其他线程调用，以使被终止线程重新恢复运行。

（6）线程的实现方式

线程的实现分为两类：用户级线程 (User-Level　Thread,ULT) 和内核级线程(Kernel-Level Thread,KLT)。内核级线程也称内核支持的线程。

a. 用户级线程

在用户级线程中，有关线程管理(创 建、撤销和切换等)的所有工作都由应用程序在用户空间内(用户态)完成，无须操作系统干预， 内核意识不到线程的存在。

对于设置了用户级线程的系统，其调度仍然以进程为单位进行，各个进程轮流执行一个时间 片。

优点：

- 线程切换不需要转换到内核空间，节省了模式切换的开销。
- 调 度算法可以是进程专用的，不同的进程可根据自身的需要，对自己的线程选择不同的调度算法。

- 用户级线程的实现与操作系统平台无关，对线程管理的代码是属于用户程序的一部分。

缺点：

- 系统调用的阻塞问题，当线程执行一个系统调用时，不仅该线 程被阻塞，进程内的所有线程也都被阻塞。
- 不能发挥多CPU 的优势，内核每次分配给一个进程 的仅有一个 CPU，因此进程中仅有一个线程能执行。

b. 内核级线程

在操作系统中，无论是系统进程还是用户进程，都是在操作系统内核的支持下运行的，与内 核紧密相关。

内核级线程同样也是在内核的支持下运行的，线程管理的所有工作也是在内核空间 内(内核态)实现的。

操作系统也为每个内核级线程设置一个线程控制块TCB，内核根据该控制 块感知某线程的存在，并对其加以控制。

优点：

- 能发挥多 CPU 的优势，内核能同时调度同一进程中的多个线 程并行执行。
- 若进程中的一个线程被阻塞，则内核可以调度该进程中的其他线程占用CPU，也 可运行其他进程中的线程。
- 内核支持线程具有很小的数据结构和堆栈，线程切换比较快、开销 小。
- 内核本身也可采用多线程技术，可以提高系统的执行速度和效率。

缺点：

- 同一进程中的线程切换，需要从用户态转到内核态进行，系统开 销较大。
- 这是因为用户进程的线程在用户态运行，而线程的调度和管理在内核实现

c. 组合方式

在组合实现方式中，内核支持多个内核级线程的建立、 调度和管理，同时允许用户程序建立、调度和管理用户级线程。

线程库：目前使用的三种主要线程库是：POSIX Pthreads、Windows API、Java。

实现线程库有两种方式：

- 在用户空间中提供一个没有内核支持的库。这种库的所有代码和数据结构都位于用户空 间中。
- 实现由操作系统直接支持的内核级的一个库。对于这种情况，库内的代码和数据结构位 于内核空间。调用库中的一个API 函数通常会导致对内核的系统调用。

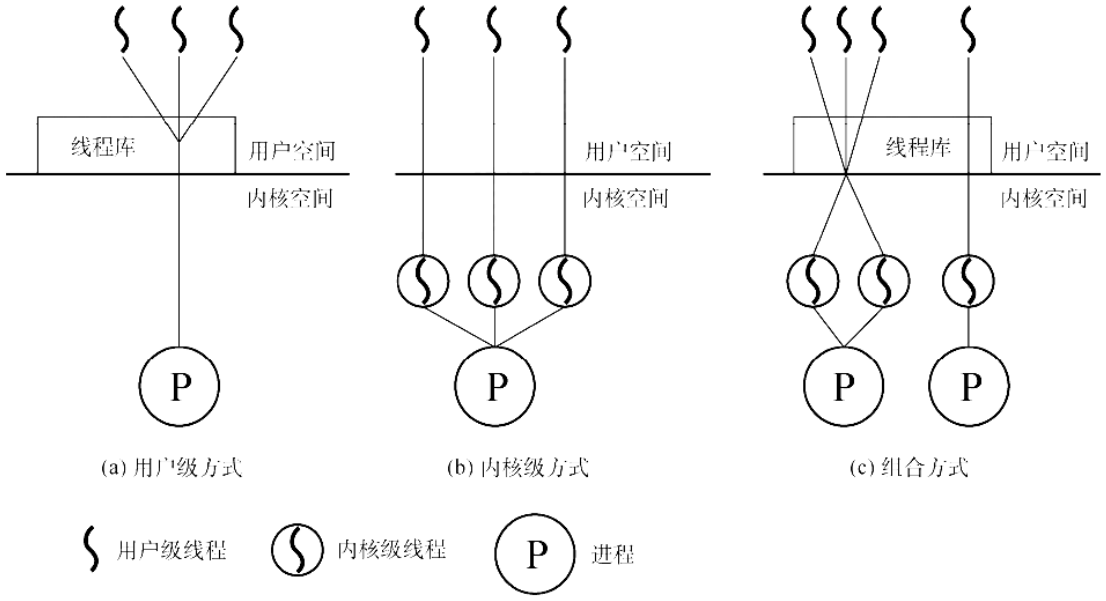


图 2.5　用户级线程和内核级线程

（7）多线程模型

在同时支持用户级线程和内核级线程的系统中，用户级线程和内核级线程连接方式的不同， 形成了下面三种不同的多线程模型。

a. 多对一模型

将多个用户级线程映射到一个内核级线程，仅当用户线程需要访问内核 时，才将其映射到一个内核级线程上，但每次只允许一个线程进行映射。

- 优点：线程管理是在用户空间进行的，无须切换到内核态，因此效率比较高
- 缺点：若一个线程在访问内核时发生阻塞，则整个进程都会被阻塞；在任何时刻，只有 一个线程能够访问内核，多个线程不能同时在多个CPU 上运行。

b. 一对一模型

将每个用户级线程映射到一个内核级线程，，线程切换由内核完成，需要切换到内核态。

- 优点：当一个线程被阻塞后，允许调度另一个线程运行，所以并发能力较强。
- 缺点：每创建一个用户线程，相应地就需要创建一个内核线程，开销较大。

c. 多对多模型

将n 个用户级线程映射到m 个内核级线程上，要求n≥m

- 既克服了多对一模型并发度不高的缺点，又克服了一对一模型的一个用户进程占 用太多内核级线程而开销太大的缺点。
- 此外，还拥有上述两种模型各自的优点。

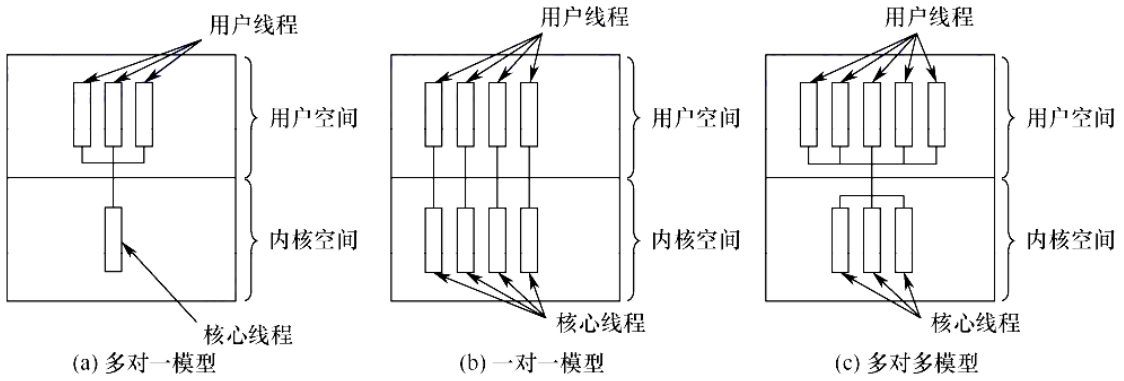


图 2.6　多线程模型

二、CPU调度

1. 调度的概念

（1）调度的基本概念

CPU 调度是对 CPU 进行分配，即从就绪队列中按照一定的算法(公平、高效的原则)选择一个 进程并将 CPU 分配给它运行，以实现进程并发地执行。

（2）调度的层次

一个作业从提交开始知道完成，往往要经历以下三级调度。

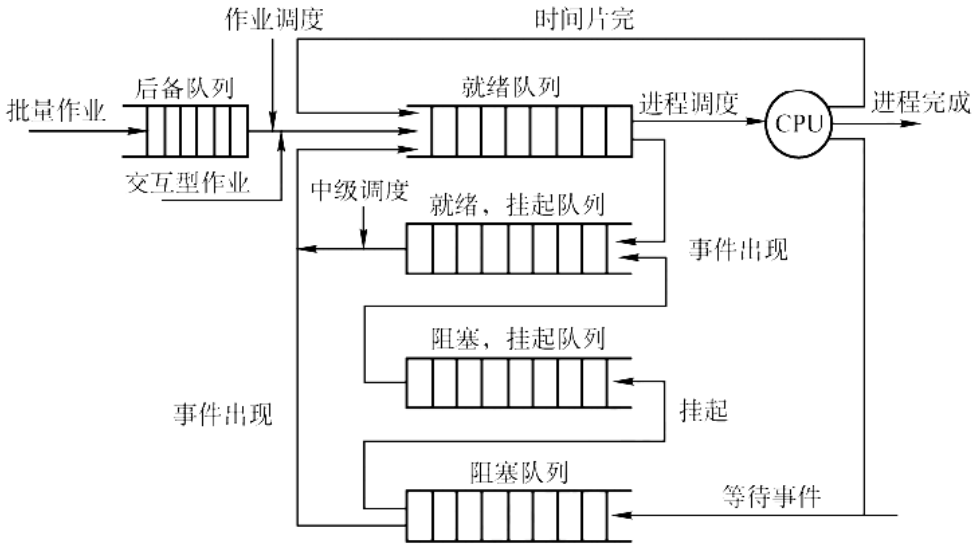


图 2.7　CPU 的三级调度

- **高级调度（作业调度）：**
  - 按照某种规则从外存上处于后备队列的作业中挑选一个(或多个),给它(们)分配内存、I/O 设备等必要的资源，并建立相应的进程，以使它(们)获得竞争CPU 的权利。
  - 简言之，作业 调度就是内存与辅存之间的调度。
  - 每个作业只调入一次，调出一次
  - 多道批处理系统中大多配有作业调度，而其他系统中通常不需要配置作业调度。
- **中级调度（内存调度）：**
  - 引入中级调度的目的是提高内存利用率和系统吞吐量。

- 将那些暂时不能运行的进程调 至外存等待，此时进程的状态称为挂起态。
- 当它们已具备运行条件且内存又稍有空闲时，由中级 调度来决定将外存上的那些已具备运行条件的挂起进程再重新调入内存，并修改其状态为就绪 态，挂在就绪队列上等待。
- 中级调度实际上是存储器管理中的对换功能。

- **低级调度（进程调度）：**
  - 按照某种算法从就绪队列中选取一个进程，将 CPU 分配给它。
  - 进程调度是最基本的一种调 度，在各种操作系统中都必须配置这级调度。
  - 进程调度的频率很高，一般几十毫秒一次。

（3）三级调度的关系

- **作业调度**从外存的后备队列中选择一批作业进入内存，为它们建立进程，这些进程被送入就 绪队列，
- **进程调度**从就绪队列中选出一个进程，并将其状态改为运行态，将CPU 分配给它。
- **中级调度**是为了提高内存的利用率，系统将那些暂时不能运行的进程挂起来。

1. 作业调度为进程活动做准备，进程调度使进程正常活动起来。
2. 中级调度将暂时不能运行的进程挂起，中级调度处于作业调度和进程调度之间。
3. 作业调度次数少，中级调度次数略多，进程调度频率最高。
4. 进程调度是最基本的，不可或缺。

2. 调度的实现

（1）调度程序（调度器）

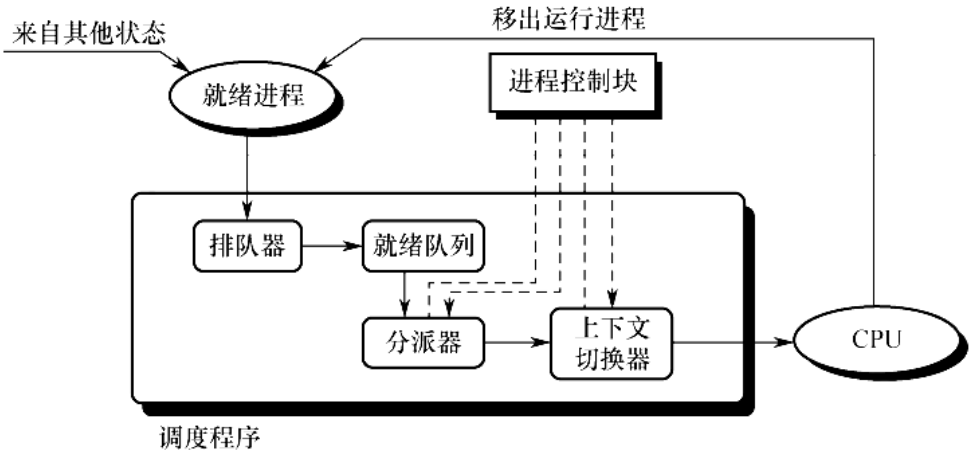


图 2.8　调度程序的结构

- 排队器：
  - 将系统中的所有就绪进程按照一定策略排成一个或多个队列，以便调度程序选择
- 分派器：
  - 依据调度程序所选的进程，从就绪队列中取出，将CPU分配给新进程
- 上下文切换器：
  - 对CPU进行切换时，会发生两队上下文的切换操作
    - 第一对：将当前进程的上下文保存在PCB中，再装入分派程序的上下文。
    - 第二对：移出分派程序的上下文，将新选进程的CPU现场信息装入CPU的各个相应寄存器

> 在上下文切换时，需要执行大量load 和 store 指令，以保存寄存器的内容，因此会花费较多时间。
>
> 现在已有硬件实现的方法来减少上下文切换时间。通常采用两组寄存器，其中一组 供内核使用，一组供用户使用。
>
> 这样，上下文切换时，只需改变指针，让其指向当前寄存器组即可。

（2）调度的时机，切换与过程

调度程序是内核程序，

请求调度　——>　调度程序运行　——>　进程切换

理论上顺序执行，但某时刻发生引起调度因素，不一定马上进行调度与切换

进行进程调度与切换的情况如下：

- 创建新进程后，父子进程都是就绪态，调度程序选择其中一个进程先运行
- 进程正常结束或异常终止后，从就绪队列中选择某个进程运行，没有就绪进程，就选择闲逛进程
- 进程因I/O请求，信号量操作，或其他原因被阻塞时候，必须调度其他进程运行。
- 当I/O 设备准备就绪后，发出I/O 中断，原先等待I/O 的进程从阻塞态变为就绪态，决定是让新的进程运行，还是让被中断的进程继续执行。

> 在有些系统中，当有更紧急的任务(如更高优先级的进程进入就绪队列)需要处理时， 或者当前进程的时间片用完时，也会被强行剥夺CPU。

进程切换在调度完成后立刻发生，要求保存原进程当前断点的现场信息，恢复被调度的进程的现场信息。

现场切换时，操作系统内核将原进程的现场信息**推入当前进程的内核堆**栈来保 存它们，并更新堆栈指针。

内核完成从新进程的内核栈中装入新进程的现场信息，并更新当前运行进程空间指针，重设PC寄存器等，开始运行新进程。

不能进行进程的调度与切换的情况如下：

- 在处理中断的过程中。
- 需要完全屏蔽中断的原子操作过程中。如加锁、解锁、中断现场保护、恢复等原子操作。

当在上述过程中发生了引起调度的条件，不能马上进行调度和切换，应设置系统请求调度标志，上述过程结束后再进行相应的调度和切换。

(3)进程调度的方式

是指当某个进程正在 CPU 上执行时，若有某个更为重要或紧迫的进程 需要处理，即有优先权更高的进程进入就绪队列，此时应如何分配CPU。

进程调度方式，是指当某个进程正在 CPU 上执行时，若有某个更为重要或紧迫的进程 需要处理，即有优先权更高的进程进入就绪队列，此时应如何分配CPU。

a. 非抢占调度方式

（非剥夺方式）

- 是指当一个进程正在CPU 上执行时，即使有某个更 为重要或紧迫的进程进入就绪队列，仍然让正在执行的进程继续执行，直到该进程运行 完成(

- 如正常结束、异常终止)或发生某种事件(如等待I/O 操作、在进程通信或同步中 执行了Block 原语)而进入阻塞态时，才将CPU 分配给其他进程。

优点：实现简单，系统开销小，适用于早期批处理系统，但不能用于分时系统和实时系统

b. 抢占方式

（剥夺方式）

- 是指当一个进程正在CPU 上执行时，若有某个更为重要 或紧迫的进程需要使用CPU，则允许调度程序根据某种原则去暂停正在执行的进程，将 CPU 分配给这个更为重要或紧迫的进程。

优点：提高系统吞吐率，响应效率。

抢占遵循的原则：优先权、短进程优先和时间片原则等。

（4）闲逛进程

- 进程切换时，若系统中没有就绪进程，则会调度闲逛进程 (Idle Process) 运行，它的PID 为0。
- 若没有其他进程就绪，则该进程就一直运行，并在指令周期后测试中断。
- 闲逛进程的优先 级最低，没有就绪进程时才会运行闲逛进程，只要有进程就绪，就会立即让出CPU。
- 闲逛进程不需要CPU 之外的资源，它不会被阻塞。

（5）两种线程的调度

- **用户级线程调度：**
  - 内核并不知道线程的存在，所以内核还是和以前一样，选择一个 进程，并给予时间控制。由进程中的调度程序决定哪个线程运行。
- **内核级线程调度：**
  - 内核选择一个特定线程运行，通常不用考虑该线程属于哪个进程。对 被选择的线程赋予一个时间片，若超过了时间片，则会强制挂起该线程。

用户级线程的切换在同一进程中进行，仅需少量机器指令

内核级线程的线程切换 需要完整的上下文切换、修改内存映像、使高速缓存失效，这就导致了若干数量级的延迟。

3. 调度的目标

不同的调度算法具有不同的特性，在选择调度算法时，必须考虑算法的特性。

为了比较CPU 调度算法的性能，有几种评价标准：

- **CPU利用率：**
  - $$\text{CPU的利用率} = \frac{\text{CPU有效工作时间}}{\text{CPU有效工作时间} + \text{CPU空闲等待时间}}$$
- **系统吞吐量：**
  - 表示单位时间内CPU 完成作业的数量。
- **周转时间：**
  - 指从作业提交到作业完成所经历的时间

- 是作业等待、在就绪队列中排队、 在CPU 上运行及I/O 操作所花费时间的总和。
- $$周转时间 = 作业完成时间 - 作业提交时间$$
- **平均周转时间**
  - $$平均周转时间=(作业\ 1\ 的周转时间 +\cdots+ 作业\ n\ 的周转时间)/n$$
  - 指多个作业周转时间的平均值
- **带权周转时间：**
  - $$带权周转时间=\frac{作业周转时间}{作业实际运行时间}$$
  - 指作业周转时间与作业实际运行时间的比值
- **平均带权周转时间：**
  - 指多个作业带权周转时间的平均值
  - $$平均带权周转时间 =(作业\ 1\ 的带权周转时间 +\cdots+ 作业\ n\ 的带权周转时间)/n$$

- **等待时间：**
  - 进程处于等待CPU的时间之和，等待时间越长，用户满意度越低
  - CPU调度算法不影响作业执行或I/O操作的时间，只影响作业在就绪队列中等待所花的时间
- **响应时间：**
  - 指从用户提交到系统首次产生响应所用的时间
  - 调度策略应尽量降低响应时间
- **设计调度算法，一方面要考虑用户要求，另一方面要考虑系统整体效率，和调度算法开销。**

4. 进程切换

进程的创建，撤销，以及要求系统设备完成的I/O操作，都是利用系统调用而进入内核，由内核中的处理程序完成的。

进程的切换也是在内核的支持下实现的。

（1）上下文切换

- 切换 CPU 到另一个进程需要保存当前进程状态并恢复另一个进程的状态，这个任务称为上 下文切换。
- 进程上下文采用进程PCB 表示，包括CPU 寄存器的值、进程状态和内存管理信息等。
- 进行上下文切换时，将旧进程的转换保存在其PCB中，然后加载经调度而要执行的新进程的上下文。

上下文切换的流程：

1. 挂起一个进程，将CPU 上下文保存到PCB, 包括程序计数器和其他寄存器。
2. 将进程的PCB 移入相应的队列，如就绪、在某事件阻塞等队列。
3. 选择另一个进程执行，并更新其PCB。
4. 恢复新进程的CPU 上下文。
5. 跳转到新进程PCB 中的程序计数器所指向的位置执行。

（2）上下文切换的消耗

下文切换对系统来说意味着消耗大量的CPU 时间。

有些CPU 提供多个寄存器组，这样，上下文切换就只需要简单改变当前寄存器组的指针。

（3）上下文切换与模式切换

- 用户态和内核态之间的切换称为模式切换，模式切换时，CPU 逻辑上可能还在执行同一进程，没有改变当前进程，不是上下文切换
- 用户进 程最开始都运行在用户态，若进程因中断或异常进入内核态运行，执行完后又回到用户态刚被中 断的进程运行。

> 切换和调度的区别：
>
> - 调度是指决定资源分配给哪个进程的行为，是一种决策行为
> - 切换是指实际分 配的行为，是执行行为
> - 一般，先有资源的调度，再有进程的切换

5. CPU调度算法

操作系统中存在多种调度算法，有的调度算法适用于作业调度，有的调度算法适用于进程调 度，有的调度算法两者都适用。

（1）先来先服务算法

既可以用于作业调度，也可以用于进程调度。

a. FCFS的思想

- **作业调度中：**
  - 每次从后备作业队列中选择最先进入该队列的一个或几个作业，将它们调入内存，分配必要的资源，创建进程并放入就绪队列。
- **进程调度中：**
  - 每次从就绪队列中选择最先进入该队列的进程，将CPU分配给他，使之投入运行，直到运行完成或者因为某种原因阻塞了才释放CPU

b. 批处理系统中作业完成时间分析

FCFS 调度算法属于不可剥夺算法。

在使用优先级作为调度策略的系统 中，往往对多个具有相同优先级的进程按FCFS 原则处理。

- 特点：
  - 算法简单，效率低
  - 对长作业有利，对短作业不利
  - 有利于CPU繁忙型作业，不利于I/O繁忙型作业。

（2）短作业优先调度算法

- 短作业优先：
  - 从后备队列中选择一个或几个估计运行时间最短的作业，将它们调入内存运行；
- 短进程优先：

- 从就绪队列中选择一个估计运行时间最短的进程

缺点：

- 对长作业不利，长作业可能长期不被调度，产生饥饿现象。（与死锁区分）
- 不能保证紧迫性作业会被及时处理
- 用户又可能有意或无意地 缩短其作业的估计运行时间，致使该算法不一定能真正做到短作业优先调度

可以为抢占式和非抢占式。

当一个新进程到达就绪 队列时，若其估计执行时间比当前进程的剩余时间小，则立即暂停当前进程，将CPU 分配给新进 程。

因此，抢占式SPF 调度算法也称最短剩余时间优先调度算法。

短作业调度算法的平均等待时间、平均周转时间最优

（3）高响应比优先调度算法

高响应比优先调度算法主要用于作业调度，是对FCFS 调度算法和SJF 调度算法的一种综合 平衡，同时考虑了**每个作业的等待时间**和**估计的运行时间。**

计算后备队列中的作业的响应比，选择响应比最高的作业投入运行。

响应比的变化规律可描述为

$$响应比R_p = \frac{等待时间+要求服务时间}{要求服务时间}$$

- 作业等待时间相同时，要求服务越短，响应比越高，越有利于短作业
- 要求服务时间相同时，等待时间越长，响应比越高，类似于FCFS
- 对于长作业，作业的响应比可以随着等待时间的增加而提高，等待足够长时，可以获得CPU，克服了饥饿现象

（4）优先级调度算法

可以用于进程调度，作业调度。

- 每次从就绪队列（后备队列）中选择优先级最高的一个或几个进程（作业）
  - 将作业调入内存，分配资源，创建进程，放入就绪队列
  - 将进程上CPU，投入运行。

a. 两类优先级调度算法

- 非抢占式优先级调度算法
- 抢占式优先级调度算法

b. 进程优先级的分类

- 静态优先级：
  - 在创建进程时确定，整个运行期间不变
  - 依据是：进程类型，进程对资源的要求，用户要求
  - 优点：简单易行，系统开销小
  - 缺点：不够精确，可能出现优先级低的进程长时间得不到调度的情况

- 动态优先级：
  - 创建进程时先赋予一个优先级，随着进程的推进或等待时间的增加而改变
  - 例如可以规定优先级随等待时间的增加而提高， 于是，对于优先级初值较低的进程，在等待足够长的时间后也可获得CPU。

c. 进程优先级设置原则

- 系统进程 > 用户进程
- 交互型进程 > 非交互型进程（前台进程 > 后台进程）
- I/O型进程 > 计算型进程

（5）时间片轮转调度算法

主要适用于分时操作系统。（可抢占）

- 系统将所有 的就绪进程按FCFS 策略排成一个就绪队列，每隔一定的时间(如30ms) 便产生一次时钟中断， 激活调度程序进行调度，将CPU 分配给就绪队列的队首进程，并令其执行一个时间片。
- 在执行完 一个时间片后，即使进程并未运行完成，它也必须释放出(被剥夺)CPU 给就绪队列的新队首进 程，而被剥夺的进程返回到就绪队列的末尾重新排队，等候再次运行。

当时间片未用完，进程运行完，调度程序被激活；当时间片用完，产生一个时钟中断，由时钟中断激活调度程序。

- **时间片对性能影响很大**
  - 时间片足够大，每个进程都可以在一个时间片内执行完，退化为先来先服务算法
  - 时间片很小，CPU在进程间的切换过于频繁，使CPU开销增大，用于用户进程的时间减小。
- **时间片的选择因素：**
  - 系统的响应 时间、就绪队列中的进程数目、系统的处理能力。

（6）多级队列调度算法

之前的方法中，只设置一个就绪队列，无法满足系统中不同用户对进程调度策略的不同要求。

在多CPU系统中，这一单一的调度策略实现机制的缺点更突出。

- 设置多个就绪队列，将不同类型或性质的进程固定分配到不同的就绪队 列。每个队列可实施不同的调度算法
- 同一队列中的进程可以设置不同的优先级，不同的队列本身也可以设置不同的优先 级。

（7）多级反馈队列调度算法

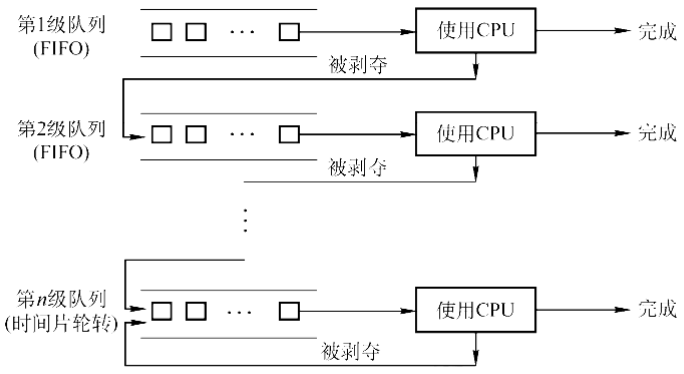


图 2.9　多级反馈队列调度算法

思想：

- 设置多个就绪队列，每个队列赋予不同优先级
- 赋予各个队列的进程运行时间片的大小各不相同
- 每个队列都采用FCFS算法。新进程首先放入一队列末尾，若在一个时间片内未完成，则转入下一级队列的末尾，并按照FCFS原则等待。
- 只有当所有更高优先级的队列为空时，才调度低优先级的队列中的进程
- 有新进程进入高优先级队列，会抢占正在运行进程的CPU，被抢占的进程放入原队列末尾

优点：

- 终端型作业用户：短作业优先。
- 短批处理作业用户：周转时间较短。
- 长批处理作业用户：经过前面几个队列得到部分执行，不会长期得不到处理。

（8）基于公平原则的调度算法

a. 保证调度算法：

保证调度算法向用户做出明确的性能保证，而非优先运行保证。

- 有n个用户，保证每个用户获得1/n的CPU时间
- 单用户系统有n个进程运行，保证每个进程获得1/n的CPU时间

b. 公平分享调度算法

保证所有用户能获得相同的 CPU 时间，或所要求的时间比例。

不管用户启动多少进程，都能保证每个用户分配得到应得的CPU份额。

6. 多处理机调度

多处理机调度比较复杂，与系统结构有关。

- **非对称处理机(Asymmetric MultiProcessing,AMP)**
  - 采用主从式操作系统，内核驻留在主机上，从机只运行用户程序，进程调度由主机负责。
  - 当从机空闲时，向主机发送一个索求进程的信号
  - 主机上有一个就绪队列，只要队列不为空，主机便从队首摘下一个进程分配给索求进程的从机

  缺点：容易导致主机繁忙，成为系统瓶颈。
- **对称多处理机(Symmetric MultiProcessing,SMP)**
  - 所有处理机都是相同的，因此由调度程 序将任何一个进程分配给任何一个 CPU 。本节主要讨论SMP 系统的调度问题。

（1）亲和性和负载平衡

当一个进程从一个CPU移到其他CPU上时，将第一个CPU上的缓存设为无效，重新填充第二个CPU的缓存，这种方式的代价比较高，应尽可能试图让一个进程运行在同一个CPU上，称为**处理器亲和性**。

对于SMP系统，应尽量保证所有CPU的负载平衡，将负载平均分配到SMP 系统的所有CPU 上。

负载平衡通常会抵消处理器亲和性带来的好处，因此在某些系统中，只有当不平衡达到一定程度后才移动进程。

（2）多处理机调度方案

a. 方案一：公共就绪队列

系统中仅设置一个公共就绪队列，所有 CPU 共享同个就绪队列。

很好的实现负载平衡，但处理器亲和性不好

提升亲和性的方法有两种：

- **软亲和：**
  - 指由调度程序尽量保持一个进程到某个CPU上，但也可以迁移到其他CPU上
- **硬亲和：**
  - 用户进程通过系统调用，主动请求系统分配到固定CPU上。

> 例如，Linux 系统实现了软亲和，也支持硬亲和的系统调用。

b. 方案二：私有就绪队列

系统为每一个CPU设置一个私有就绪队列，当CPU空闲时，就从各自的私有就绪队列中选择一个进程运行。

很好的实现了亲和性，但必须进行负载平衡。

平衡负载的方法有两种：

- **推迁移：**一个特定的系统程序周期性检查每个CPU的负载，发现不平衡，就从超载的CPU中“推”一些进程到空闲的CPU就绪队列。
- **拉迁移：**若一个CPU的负载很低，则从超载的CPU就绪队列中“拉”一些进程到自己的就绪队列。

三、同步与互斥

1. 同步与互斥的基本概念

（1）临界资源

将一次仅允许一个进程使用的资源称为临界资源。

对于临界资源的访问，必须互斥的进行。

在每个进程中，访问临界资源的那段代码成为临界区。

临界资源的访问过程可以分为4个部分：

- 进入区：
  - 在进入区要检查可否进入临界区，能进入临界区，应设置正在访问临界区的标志
- 临界区：
  - 进程中访问临界资源的那段代码，也称为临界段
- 退出区：
  - 将正在访问临界区的标志清除
- 剩余区：
  - 代码中的剩余部分

（2）同步

同步亦称直接制约关系。

指为完成某种任务而建立的两个或多个进程，这些进程因为需要**协调它们的运行次序**而**等待、传递信息**所产生的制约关系。

同步关系源于进程之间的相互合作

（3）互斥

互斥亦称为间接制约关系。

当一个进程进入临界区使用临界资源时，另一个进程必须等待，当占用临界资源的进程退出临界区后，另一进程才允许去访问此临界资源。

为禁止两个进程同时进入临界区，同步机制应遵循以下准则：

- **空闲让进。**
  - 临界区空闲时，可以允许一个请求进入临界区的进程立即进入临界区。
- **忙 则 等 待。**
  - 当已有进程进入临界区时，其他试图进入临界区的进程必须等待。
- **有限等待。**
  - 对请求访问的进程，应保证能在有限时间内进入临界区，防止进程无限等待。
- **让权等待**
  - (原则上应该遵循，但非必须)。当进程不能进入临界区时，应立即释放处理器， 防止进程忙等待。

2. 实现临界区互斥的基本方法

（1）单标志法

该算法设置一个公用整型变量 turn，指示允许进入临界区的进程编号，当 turn = 0 时，表示允许 P_0 进入临界区；当 turn = 1 时，表示允许 P_1 进入临界区。进程退出临界区时将临界区的使用权赋予给另一个进程，当 P_i 退出临界区时，将 turn 置为 j（$i=0$、$j=1$ 或 $i=1$、$j=0$）。

```
进程 P0:                                进程 P1:
while(turn!=0);                         while(turn!=1);          //进入区
critical section;                       critical section;        //临界区
turn=1;                                 turn=0;                  //退出区
remainder section;                      remainder section;       //剩余区
```

该算法可以实现每次只允许一个进程进入临界区。但两个进程必须交替进入临界区，若某个进程不再进入临界区，则另一个进程也将无法进入临界区（违背“空闲让进”准则）。这样很容易造成资源利用不充分。若 P_0 顺利进入临界区并从临界区离开，则此时临界区是空闲的，但 P_1 并没有进入临界区的打算，而 turn = 1 一直成立，则 P_0 就无法再次进入临界区。

不满足：空闲让进，只能交替进入

（2）标志先检查法

该算法设置一个布尔型数组 flag[2]，用来标记各个进程想进入临界区的意愿，flag[i]=true 表示 P_i 想要进入临界区（i=0 或 1）。P_i 进入临界区前，先检查对方是否想进入临界区，若想，则等待；否则，将 flag[i]置为 true 后，再进入临界区；当 P_i 退出临界区时，将 flag[i]置为 false。

```
进程 P0:
while(flag[1]);      ①
flag[0]=true;        ③
critical section;
flag[0]=false;
remainder section;

进程 P1:
while(flag[0]);      ②   //进入区
flag[1]=true;        ④   //进入区
critical section;        //临界区
flag[1]=false;           //退出区
remainder section;       //剩余区
```

优点：不用交替进入，可连续使用

缺点：P_0和P_1肯呢个同时进入临界区，按序列1234执行时，即即检查双发标志后和设置自己的标志前可能发生进程切换，结果双方都检查通过，会同时进入临界区。

原因：检查和设置操作不是同步的

不满足：忙则等待

（3）双标志后检查法

- 思想：先设置自己的标志，再检查对方的标志

```
进程 P0:
flag[0]=true;        ①
while(flag[1]);      ③
critical section;
flag[0]=false;
remainder section;

进程 P1:
flag[1]=true;        ②   //进入区
while(flag[0]);      ④   //进入区
critical section;        //临界区
flag[1]=false;           //退出区
remainder section;       //剩余区
```

按序列1234执行时，发现对方也想进入临界区，双方都争着进，结果都进不了。

不满足：空闲让进，有限等待（会导致饥饿）

（4）Peterson算法

结合算法一和算法三，利用flag[]解决互斥访问问题，利用turn解决饥饿问题。

```
进程 P0:
flag[0]=true;
turn=1;
while(flag[1]&&turn==1);
critical section;
flag[0]=false;
remainder section;

进程 P1:
flag[1]=true;                //进入区
turn=0;                      //进入区
while(flag[0]&&turn==0);     //进入区
critical section;            //临界区
flag[1]=false;               //退出区
remainder section;           //剩余区
```

思想：

表达意愿——> 表示谦让——>对方有意愿，并且接受谦让，则自己等待，对方进入

出临界区的时候，把自己的意愿设置为false。

- 优点：满足空闲让进，忙则等待，有限等待
- 缺点：不满足让权等待

（5）中断屏蔽方法

当一个进程正在执行他的临界区代码时，关中断，防止其他进程进入其临界区。

因为CPU只在发生中断时引起进程切换，因此屏蔽中断能够有限保证当前运行的进程让临界区代码顺利地执行完，保证互斥的正确实现，然后开中断。

```
…
关中断；
临界区；
开中断；
…
```

- **缺点：**
  - 限制了CPU交替执行程序的能力，系统效率明显降低
  - 对内核来说，在它执行更新变量的几条指令期间，关中断很方便，但是交给用户去关中断，若一个进程关中断后不再开，系统可能因此终止
  - 不适用于多处理器系统，因为在一个CPU上关中断并不能防止在其他CPU上执行相同的临界区代码

（6）硬件指令方法——TestAndSet指令

（2）硬件指令方法——TestAndSet 指令

借助一条硬件指令——TestAndSet 指令（简称 TS 指令）实现互斥，这条指令是原子操作。其功能是读出指定标志后将该标志设置为真。指令的功能描述如下：

```
boolean TestAndSet(boolean *lock){
    boolean old;
    old=*lock;                  //old 用来存放 lock 的旧值
    *lock=true;                 //无论之前是否已加锁，都将 lock 置为 true
    return old;                 //返回 lock 的旧值
}
```

命题追踪 ▶ TestAndSet 指令实现互斥的分析（2016）

用 TS 指令管理临界区时，为每个临界资源设置一个共享布尔变量 lock，表示该资源的两种状态：true 表示正被占用（已加锁）；false 表示空闲（未加锁），初值为 false，因此可将 lock 视为一把锁。进程在进入临界区之前，先用 TS 指令检查 lock 值：①若为 false，则表示没有进程在临界区，可以进入，并将 lock 置为 true，这意味着关闭了临界资源（加锁），使任何进程都不能进入临界区；②若为 true，则表示有进程在临界区，进入循环等待，直到当前访问临界区的进程退出时解锁（将 lock 置为 false）。利用 TS 指令实现互斥的过程描述如下：

```
while TestAndSet(&lock);        //加锁并检查
进程的临界区代码段;
lock=false;                     //解锁
进程的其他代码;
```

相比于软件实现方法，TS 指令将“加锁”和“检查”操作用硬件的方式变成了一气呵成的原子操作。相比于关中断方法，由于“锁”是共享的，这种方法适用于多处理器系统。缺点是，暂时无法进入临界区的进程会占用 CPU 循环执行 TS 指令，因此不能实现“让权等待”。

(7) 硬件指令方法——Swap指令

（3）硬件指令方法——Swap 指令

Swap 指令的功能是交换两个字（字节）的内容。其功能描述如下：

```
Swap(boolean *a, boolean *b){
    boolean temp=*a;
    *a=*b;
    *b=temp;
}
```

注 意

以上对 TS 和 Swap 指令的描述仅为功能描述，它们由硬件逻辑实现，不会被中断。

命题追踪 ▶ Swap 指令与函数实现的分析（2023）

用 Swap 指令管理临界区时，为每个临界资源设置一个共享布尔变量 lock，初值为 false；在每个进程中再设置一个局部布尔变量 key，初值为 true，用于与 lock 交换信息。从逻辑上看，Swap 指令和 TS 指令实现互斥的方法并无太大区别，都是先记录此时临界区是否已加锁（记录在变量 key 中），再将锁标志 lock 置为 true，最后检查 key，若 key 为 false，则说明之前没有其他进程对临界区加锁，于是跳出循环，进入临界区。其处理过程描述如下：

```
boolean key=true;
while(key!=false)
```



```
    Swap(&lock, &key);
进程的临界区代码段;
lock=false;
进程的其他代码;
```

用硬件指令方法实现互斥的优点：①简单、容易验证其正确性；②适用于任意数量的进程，支持多处理器系统；③支持系统中有多个临界区，只需为每个临界区设立一个布尔变量。缺点：①等待进入临界区的进程会占用 CPU 执行 while 循环，不能实现“让权等待”；②从等待进程中随机选择一个进程进入临界区，有的进程可能一直选不上，从而导致“饥饿”现象。

3. 互斥锁

解决临界区最简单的工具是互斥锁（mutex lock）

- 一个进程在进入临界区之前调用acquire()，获得锁
- 在退出临界区时调用release()函数，释放锁
- 每个互斥锁有一个available，表示锁是否可用。
- acquire()或 release()的执行必须是原子操作，因此互斥锁通常采用硬件机制来实现。

```
acquire(){                          //获得锁的定义
    while(!available)
        ;                           //忙等待
    available=false;                //获得锁
}
release(){                          //释放锁的定义
    available=true;                 //释放锁
}
```

互斥锁也称为自旋锁，主要缺点是忙等待。

当有一个进程在临界区时，任何其他进程在临界区之前必须连续循环调用acquire()。

类似的还有单标志法，TS指令，Swap指令。

当多个进程共享同一CPU时，浪费了CPU周期，因此互斥锁常用于多处理器系统。

自选锁的优点：进程在等待锁期间，没有上下文切换，若上锁时间较短，则等待代价不高。

4. 信号量

信号量只能被两个标准的原语wait()和signal()访问，也可简写为P()，V()，或者简称P操作，V操作。

原语是指完成某种功能且不被分割，不被中断执行的操作序列，通常可以由硬件来实现。

原语之所以不能被中断执行，是因为原语对变量的操作过 程若被打断，可能去运行另一个对同一变量的操作过程。

（1）整型信号量

整型信号量被定义为一个用于表示资源数量的整型量S，相比于普通整型变量，对整型信号量的操作只有三种：初始化，wait操作，signal操作。

```
wait(S){                    //相当于进入区
    while(S<=0);            //若资源数不够，则一直循环等待
    S=S-1;                  //若资源数够，则占用一个资源
}
signal(S){                  //相当于退出区
    S=S+1;                  //使用完后，就释放一个资源
}
```

整型信号量机制的wait操作，只要信号量S小于等于0，就会不断循环测试。

该机制并**未遵循让权等待准则**，而是使进程处于**忙等**状态。

（2）记录型信号量

记录型信号量机制，除了有一个用于表示资源数量的整型变量value外，还有一个进程链表L，用于连接所有等待该资源的进程。

记录型信号量数据结构：

```
typedef struct{
    int value;
    struct process *L;
} semaphore;
```

- P操作
  - ```
    void wait(semaphore S){        //相当于申请资源
        S.value--;
        if(S.value<0){
            add this process to S.L;
            block(S.L);
        }
    }
    ```
  - 利用block原语阻塞进程
    ```

- V操作

```
void signal(semaphore S){      //相当于释放资源
    S.value++;
    if(S.value<=0){
        remove a process P from S.L;
        wakeup(P);
    }
}
```

  - 利用wakeup原语唤醒进程

(3) 利用信号量实现进程互斥

```
semaphore S=1;                  //初始化信号量，初值为 1
P1(){
    ...
    P(S);                       //准备访问临界资源，加锁
    进程 P1 的临界区;
    V(S);                       //访问结束，解锁
    ...
}
P2(){
    ...
    P(S);                       //准备访问临界资源，加锁
    进程 P2 的临界区;
    V(S);                       //访问结束，解锁
    ...
}
```

- S的取值范围为（-1,0,1）
  - S=1，表示两个进程都没有进入临界区
  - S=0，表示有一个进程已经进入临界区
  - S=-1，表示有一个进程正在临界区，另一个进程因等待阻塞

注 意

①对不同的临界资源需要设置不同的互斥信号量。②P(S)和 V(S)必须成对出现，缺少 P(S)就不能保证对临界资源的互斥访问；缺少 V(S)会使临界资源永远不被释放，从而使因等待该资源而阻塞的进程永远不能被唤醒。③考试还可能考查多个资源的问题，有多少资源就将信号量初值设为多少，申请资源时执行 P 操作，释放资源时执行 V 操作。

(4) 利用信号量实现同步

同步源于进程之间的相互合作，要让本来异步的并发进程相互配合，有序推进。例如，进程 P_1 和 P_2 并发执行，存在异步性，因此二者交替推进的次序是不确定的，若 P_2 的语句 y 要使用 P_1 的语句 x 的运行结果，则必须保证语句 y 一定在语句 x 之后执行。为了实现这种同步关系，需要设置一个同步信号量 S，其初值为 0（可以这么理解：刚开始是没有这种资源的，P_2 需要使用这种资源，而又只能由 P_1 产生这种资源）。其实现如下：

```
semaphore S=0;                  //初始化信号量，初值为 0
P1(){
    x;                          //执行语句 x
    V(S);                       //告诉进程 P2，语句 x 已经完成
    ...
}
P2(){
    ...
    P(S);                       //检查语句 x 是否运行完成
    y;                          //获得 x 的运行结果，执行语句 y
    ...
}
```

- 在同步问题中，若某个行为会提供某种资源，则在这个行为之后V这种资源
- 若某个行为要用到这种资源，则在这个行为之前P这种资源，在互斥问题中，P,V操作要加紧使用临界区的那个行为，中间不能有其他冗余代码

(5) 利用信号实现前驱关系

对于每对前驱关系都是一个同步问题，因此要为每对前驱关系设置一个同步信号量，其初值均为0。

命题追踪 ▶ 信号量实现前驱关系的应用题（2020、2022）

信号量也可用来描述程序或语句之间的前驱关系。图2.10给出了一个前驱关系举例，其中S_1, S_2, S_3, …, S_6是简单的程序段（只有一条语句）。

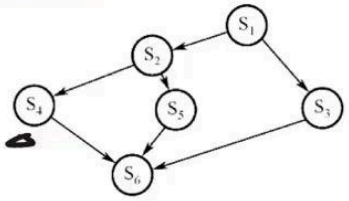


图2.10 前驱关系举例

其实，每对前驱关系都是一个同步问题，因此要为每对前驱关系设置一个同步信号量，其初值均为0。在“前驱操作”之后，对相应的同步信号量执行V操作，在“后继操作”之前，对相应的同步信号量执行P操作。以图2.10为例，S_2是S_1的后继，要用到S_1的资源，前面总结过，在同步问题中，要用到某种资源，就要在行为之前P这种资源；S_2是S_4, S_5的前驱，给S_4, S_5提供资源，因此要在S_2之后V由S_4和S_5所产生的资源。为保证$S_1 \to S_2$, $S_1 \to S_3$, $S_2 \to S_4$, $S_2 \to S_5$, $S_3 \to S_6$, $S_4 \to S_6$, $S_5 \to S_6$的前驱关系，需分别设置同步信号量a12, a13, a24, a25, a36, a46, a56。其实现如下：

```
semaphore a12=a13=a24=a25=a36=a46=a56=0; //初始化信号量
S1(){
    …;
    V(a12);V(a13);              //S1 已经运行完成
}
S2(){
    P(a12);                     //检查 S1 是否运行完成
    …;
    V(a24);V(a25);              //S2 已经运行完成
}
S3(){
    P(a13);                     //检查 S1 是否已经运行完成
    …;
    V(a36);                     //S3 已经运行完成
}
S4(){
    P(a24);                     //检查 S2 是否已经运行完成
    …;
    V(a46);                     //S4 已经运行完成
}
S5(){
    P(a25);                     //检查 S2 是否已经运行完成
    …;
    V(a56);                     //S5 已经运行完成
}
S6(){
    P(a36);                     //检查 S3 是否已经运行完成
    P(a46);                     //检查 S4 是否已经运行完成
    P(a56);                     //检查 S5 是否已经运行完成
    …;
}
```

（6）分析进程同步和互斥

- 关系分析：找到问题中的进程数，并分析他们之间的同步和互斥关系，同步、互斥、前驱关系直接按照例子中的经典范式改写
- 整理思路：找出解决问题的关键点，根据进程的操作流程确定P操作，V操作的大致顺序
- 设置信号量：根据以上两步，设置需要的信号量，确定初值。

5. 经典同步问题

（1）生产者-消费者问题

（2）读者-写者问题

（3）哲学家进餐问题

6. 管程

（1）管程的定义

利用共享数据结构抽象的表示系统中的共享资源，而将该数据结构实施的操作定义为一组过程。

进程对共享资源的申请，释放，都通过过程来实现

这个代表共享资源的数据结构，以及对该数据结构实施操作的一组过程所组成的资源管理程序，叫做管程。

管程定义了一个数据结构 和 能为并发进程所执行的一组操作，这组操作能同步进程，改变管程中的数据。

管程由4部分组成：

- 管程的名称
- 局部用于管程内部的共享数据结构说明
- 对该数据结构进行操作的一组过程（函数）
- 对局部于管程内部的共享数据设置初始值的语句

```
monitor Demo{                    //①定义一个名称为 Demo 的管程
    //②定义共享数据结构，对应系统中的某种共享资源
    共享数据结构 S;
    //④对共享数据结构初始化的语句
    init_code(){
        S=5;                     //初始资源数等于 5
    }
    take_away(){                 //③过程 1：申请一个资源
        对共享数据结构 S 的一系列处理;
        S--;                     //可用资源数-1
        ...
    }
    give_back(){                 //③过程 2：归还一个资源
        对共享数据结构 S 的一系列处理;
        S++;                     //可用资源数+1
        ...
    }
}
```

- 管程将对共享资源的操作封装起来，管程内共享数据结构只能被管程内的过程所访问。
- 每次仅允许一个进程进入管程，从而实现互斥

(2) 条件变量

一个进程进入管程后被阻塞，知道阻塞的原因解除时，再次期间，若该进程不释放管程，则其他进程无法进入管程，为此，将阻塞原因定义为条件变量condition。

- 一个进程被阻塞的原因有多种，在管程内设置了多个条件变量
- 每个条件变量保存了一个等待队列，用于记录因该条件而阻塞的所有进程
- 对条件变量只能进行两种操作，wait和signal

x.wait：当 x 对应的条件不满足时，正在调用管程的进程调用 x.wait 将自己插入 x 条件的等待队列，并释放管程。此时其他进程可以使用该管程。

x.signal：x 对应的条件发生了变化，则调用 x.signal，唤醒一个因 x 条件而阻塞的进程。

下面给出条件变量的定义和使用：

```
monitor Demo{
    共享数据结构 S;
    condition x;                    //定义一个条件变量 x
    init_code(){ ... }
    take_away(){
        if(S<=0) x.wait();          //资源不够，在条件变量 x 上阻塞等待
        资源足够，分配资源，做一系列相应处理;
    }
    give_back(){
        归还资源，做一系列相应处理;
        if(有进程在等待) x.signal();//唤醒一个阻塞进程
    }
}
```

- **信号量和条件变量的比较：**

- **相似点：**
  - 条件变量的wait/signal类似于信号量的P/V操作，可以实现进程的阻塞唤醒
- **不同点：**
  - 条件变量没有值，仅实现了排队等待的功能
  - 信号量有值，反映了剩余资源数
  - 而在管程中，剩余资源数通过共享数据结构表示。

四、死锁

1. 死锁的概念

（1）死锁的定义

多个进程因竞争资源而造成的一种僵局（互相等待对方手里的资源），使得各个进程都被阻塞，没有外力干涉，这些进程都无法向前推进。

（2）死锁与饥饿

一组进程处于死锁状态是指组内的每个进程都在等待一个事件，而该事件只可能由组内的另一个进程产生。

饥饿：进程在信号量内无穷等待的情况

- 产生饥饿的主要原因：
  - 系统中有多个进程同时申请某类资源，由分配策略确定资源分配给进程的次序，有的分配策略不公平，不能保证等待时间上界的存在。

死锁和饥饿的主要差别：

- 饥饿的进程可能只有一个，而死锁必然大于或等于两个
- 发生饥饿的进程可能处于就绪态（长期得不到CPU），也可能处于阻塞态（长期得不到所需要的I/O设备），而发生死锁的进程必然处于阻塞态。

（3）死锁产生的原因

- **系统资源的竞争：**
  - 对不可剥夺资源的竞争会产生死锁，对可剥夺资源（CPU和主存）则不会引起死锁
- **进程推进顺序非法：**
  - 请求和释放资源的顺序不当，也会导致死锁。
  - P1占用资源R1，P2占用资源R2，P1请求R2,P2请求R1，两个进程都会因为

（4）死锁产生的必要条件

产生死锁必须满足以下4个条件：

- **互斥资源**
- **不可剥夺条件：**
  - 进程获得的资源，只能由该进程自己释放
- **请求并保持条件：**

- 进程已经保持了至少一个资源，但又提出来了新的资源请求，而该资源被其他进程占有，此时请求进程被阻塞，但是保持自己获得的资源不放。

- **循环等待条件：**
  - 存在一种进程资源的循环等待链，链中每个进程已获得资源的资源同时被链中的下一个进程请求。

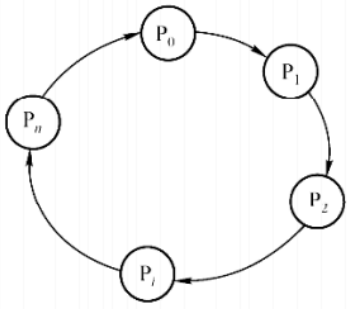


图 2.13　循环等待

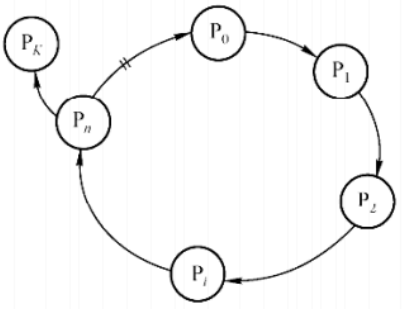


图 2.14　满足条件但无死锁

> 循环等待只是死锁的必要条件。

资源分配图含圈但是系统又不一定有死锁的原因：

- 同类资源数大于1，若系统中每类资源都只有一个资源，则资源分配图含圈，就变成了系统出现死锁的充分必要条件。

> 区分不可剥夺条件，和 请求保持条件：
>
> 手里拿着一个苹果（即使不打算吃），别人不能将手中的苹果拿走，这是不可剥夺条件
>
> 左手拿着一个苹果，允许右手再去拿一个苹果，这是请求保持条件

（5）死锁的处理策略

为使系统不发生死锁，必须设法破坏产生死锁的4个必要条件之一，或允许死锁发生，但当死锁发生时，可以检测出来。

- 死锁预防：
  - 破坏产生死锁的4个必要条件中的1个或多个
- 避免死锁：
  - 在资源动态分配过程中，采用某种方法，防止系统进入不安全状态。
- 死锁的检测及解除
  - 通过系统的检测结构及时的检测出来死锁的发生，采用某种方法解除死锁。

表 2.5　死锁处理策略的比较

	资源分配策略	各种可能模式	主要优点	主要缺点
死锁预防	保守，宁可资源闲置	一次请求所有资源，资源剥夺，资源按序分配	适用于突发式处理的进程，不必进行剥夺	效率低，初始化时间延长；剥夺次数过多；不便灵活申请新资源
死锁避免	是预防和检测的折中（在运行时判断是否可能死锁）	寻找可能的安全允许顺序	不必进行剥夺	必须知道将来的资源需求；进程不能被长时间阻塞
死锁检测	宽松，只要允许就分配资源	定期检查死锁是否已经发生	不延长进程初始化时间，允许对死锁进行现场处理	通过剥夺解除死锁，造成损失

2. 死锁预防

（1）破坏互斥条件

若将只能互斥使用的资源改为允许共享使用，则系统不会进入死锁状态，但有些资源不能同时访问，对于某些情况不可行。**（不能通过算法实现，其余都可以）**

（2）破坏不可剥夺条件

当一个已经保持了某些不可剥夺资源的进程，请求新的资源而得不到满足时，就必须释放已经保持的所有资源。

实现起来比较复杂，释放已获得的资源可能导致前一阶段工作的失效，因此这种方法适用于容易保存和恢复的资源。

反复的申请和释放资源既影响进程推进速度，又增加系统开销，从而降低系统吞吐量。

（3）破坏请求并保持条件

要求进程在请求资源时，不能持有不可剥夺资源，有两种方式：

- 采用预先静态分配方法，进程在运行前一次申请完成它所需要的全部资源，破坏了保持条件
- 允许进程只获得运行前期所需的资源后，便可开始运行，进程在运行过程中再逐步释放已分配给自己并且使用完毕的全部资源后，才能请求新的资源

> 方案一会使资源利用率低，并且导致饥饿现象。方案二改进了这些缺点。

（4）破坏循环等待条件

采用顺序资源分配法，首先给系统的各类资源编号，规定每个进程必须按照编号递增的顺序请求资源，同类资源（编号相同的资源）一次申请完。

也就是，一个进程只有在已经占有小编号的资源时，才能申请更大编号的资源。

按此规则，已持有大资源编号的进程不能再逆向申请小编号的资源，因此不会产生循环

3. 死锁避免

（1）系统安全状态

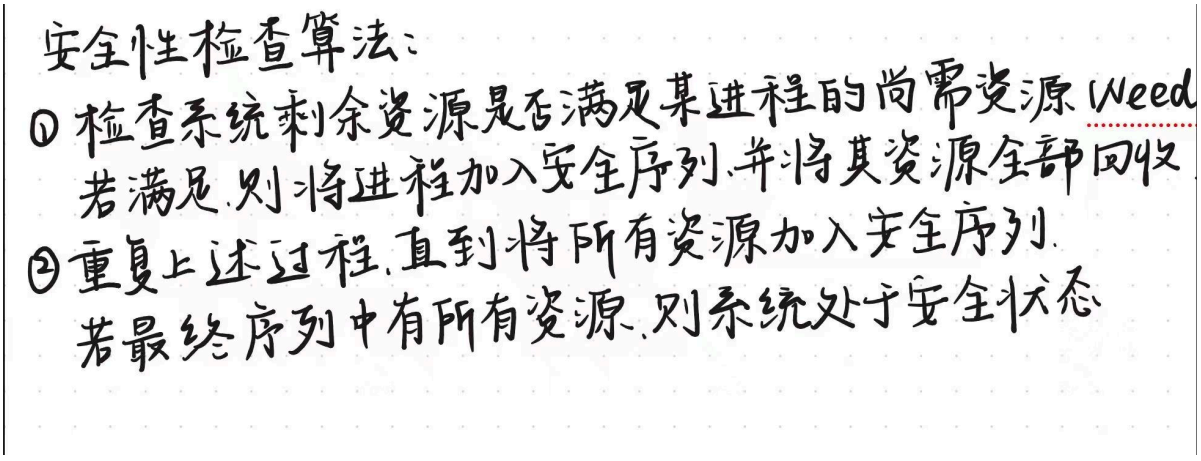
是指系统能按照某种推进顺序，为每个进程P_i分配其所需的资源，知道满足每个进程对资源的最大需求，使得每个进程都可以顺利完成。此时称P_1，P_2，...，P_n为安全序列。

若系统无法找到一个安全序列，则系统处于不安全状态。

若系统处于安全状态，则一定不会发生死锁。

若系统进入不安全状态，则有可能发生死锁。

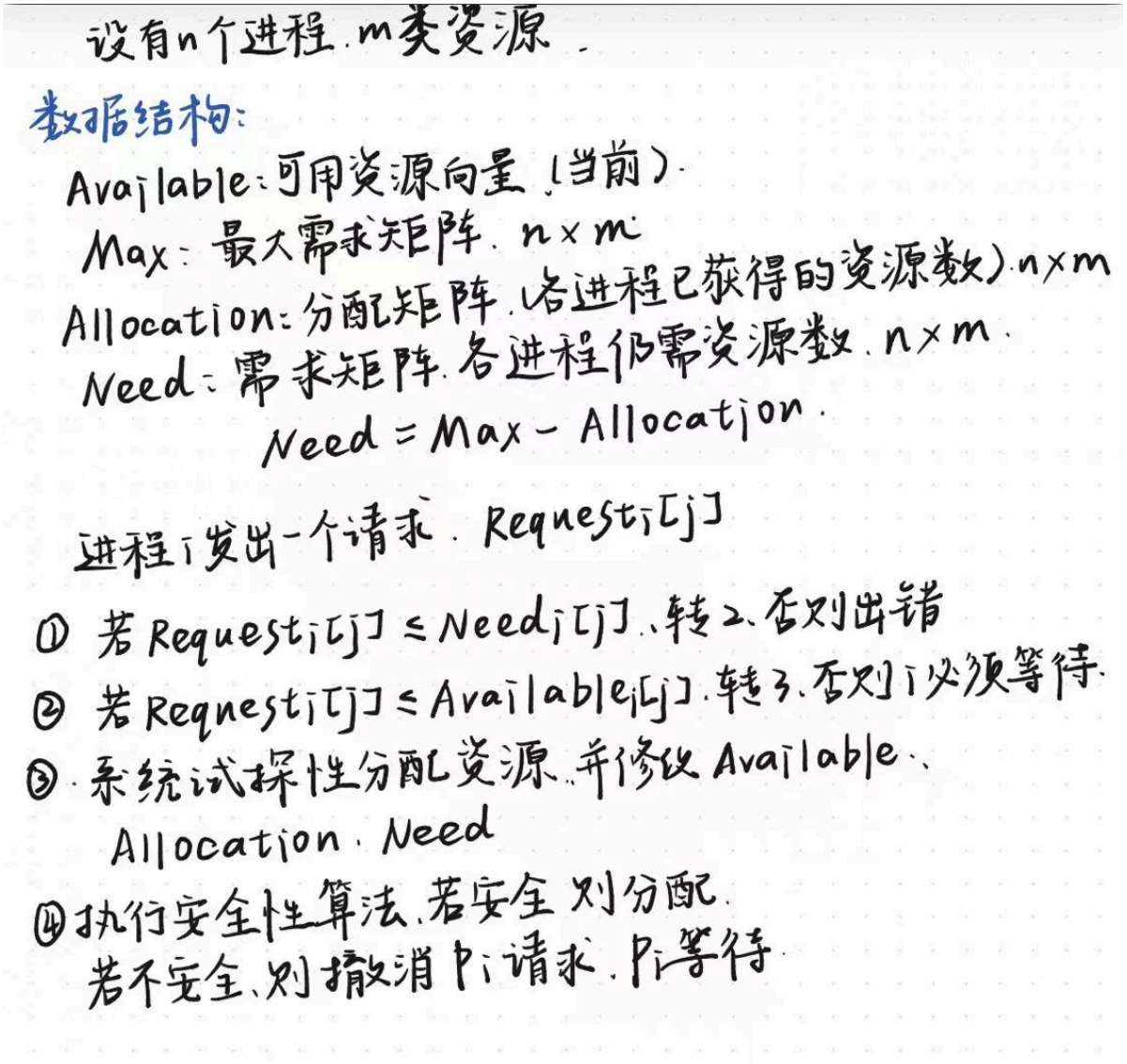
> 处于不安全状态未必就会发生死锁，但发生死锁时一定处于不安全状态



（2）银行家算法

核心思想：

在资源分配之前预先判断这次分配是否会导致系统进入不安全状态，以此决定是否答应资源的分配请求。



> 银行家算法的特点：不会限制用户的申请资源的顺序，而死锁预防方法（如破坏循环等待条件）则会限制。

4. 死锁检测和消除

（1）死锁检测

死锁检测和死锁避免的区别：

- **死锁避免：**
  - 需要在进程运行的过程中一直保证之后不出现死锁，需要知道进程从开始到结束的所有资源请求。
- 死锁检测：
  - 只检测某个时刻是否发生死锁，不需要知道进程在整个生命周期中的资源请求。

可以用资源分配图排检测系统中是否出现了死锁：

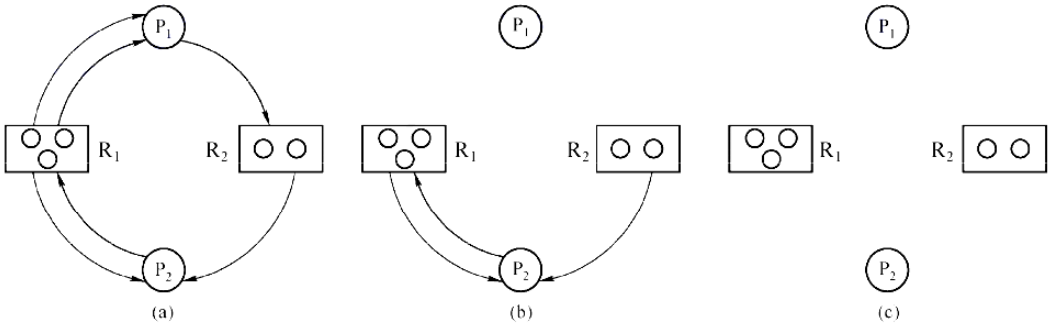


图 2.15　资源分配图及其化简过程

从资源到进程的边：分配边，该类资源已有一个资源分配给了该进程

从进程到资源的边：请求边，该进程请求一个单位的该资源

S为思索的条件是当且仅当，当S状态的资源分配图是不可完全简化，该条件为死锁定理。

（2）死锁解除

- 资源剥夺法：
  - 挂起死锁进程，并抢占他的资源，把资源分配给其他进程，但要避免长时间挂起出现得不到资源而匮乏的状态
- 撤销进程法：
  - 强制撤销部分，甚至全部死锁进程，并剥夺这些进程的资源。
  - 撤销的原则是按照进程的优先级和撤销进程代价的高低进行
- 进程回退法：
  - 让一个或多个死锁进程回退到足以回避死锁的地步，进程回退时，自愿释放资源并非被剥夺
  - 要求系统保持进程的历史信息，设置还原点