**操作系统概念读书笔记**

**段卫东**

**2017-04-06**

目录

[死锁 2](#_Toc479338747)

[死锁的四个条件 2](#_Toc479338748)

[解决死锁的方法 2](#_Toc479338749)

[死锁预防（DeadLock Prevention） 2](#_Toc479338750)

[死锁避免（DeadLock Avoidance） 3](#_Toc479338751)

[死锁检测（Deadlock Detection） 6](#_Toc479338752)

[死锁恢复（recovery from deadlock） 9](#_Toc479338753)

[总结 9](#_Toc479338754)

# 死锁

## 死锁的四个条件

A.临界区（matual exclusion）

B.持有并等待（hold and wait）

C.无竞争（No Preemption）

D.循环等待（Circular Wait）

## 解决死锁的方法

### 死锁预防（DeadLock Prevention）

此策略就是阻断死锁的四个必要条件中的一个，此方法会降低设备的利用率和降低系统的吞吐量。

#### 临界区（mutual exclusion）

如果不存在临界区，就说明资源是可以共享的，那就不可能存在死锁的问题。

#### 持有并等待（Hold and Wait）

这里有两个策略。

**策略1**：

进程在进行调度之前，事先分配好所有的资源。

**策略2:**

进程在最开始的时候，分配满足其运行的必要资源，当额外需要申请临界资源的时候，需要释放持有的所有资源，然后再申请新的资源。

**二者的弊端：**

资源的利用率并不高，临界资源被进程一直持有，可能很长时间都不被利用。很可能造成进程饥饿（starvation），比如该进程需要的所有的临界资源，而某些常用的临界资源一直被其他进程占有，导致该进程长期得不到资源，而无法运行。

#### 无竞争（no preemption）

造成死锁的第三个必要条件是没有竞争。所以，解决方案就是，如果一个进程在运行过程中，需要申请其他的临界资源，而此时，临界资源已经被占有，该进程所持有的资源，必须可以被其他需要该资源的进程进行竞争（preempt）.

**过程：**

当某个进程在运行过程中，需要申请额外的临界资源的时候，如果该临界资源可以获得，那么该进程得到该临界资源，然后继续执行，如果该临界资源不可以获得，那么查看占有该临界资源的进程A是否处于活跃状态。如果进程A处于等待状态，那么直接得到该资源，保持运行。如果进程A处于正常运行状态，那么进程直接进入等待状态，并且所持有的资源，可以被竞争（Preempt）。此时，其他进程需要该进程所持有的临界资源的时候，可以直接抢占并且使用，知道该进程获取了想要得到的临界资源，并且重新获得被抢占去的资源以后，该进程才可以重新恢复运行。

这个策略比较适应临界资源的状态容易改变，例如CPU register(CPU寄存器) and memory space（内存空间），对于打印机或者磁带驱动器，则不可以这么使用。

#### 循环等待（circular wait）

**策略：**

保证临界资源在申请的时候，按照一定的顺序进行申请。每个临界资源都维护一个整数，来代表优先级，按照由小到大的顺序进行申请资源，如果一个进程要申请资源Ri,那么自身持有的临界资源一定比Ri小，否则需要释放所有比Ri大的资源。对于临界资源的排序，按照临界资源使用的顺序进行排序比较好。比如，磁带驱动器一般在打印机之前使用，所以F(TapDriver)<F(printer)

### 死锁避免（DeadLock Avoidance）

死锁避免方法宗旨就是保证整个系统在运行过程中，始终处于安全状态（safe state），处在安全状态的系统就可以保证一定不存在死锁情况。

主要是在避免死锁的第四个条件，循环等待上。通过一些策略，预先估算该种状态是否有可能造成死锁，如果可能，则让系统不进入这种状态，如果没有可能，则可以进入该状态。

#### 安全状态（Safe State）

安全状态是指，系统在资源已分配的某个状态下，存在一个序列（a safe sequence），让各个进程可以顺利完成任务，那么这种状态，就认为是安全的状态。

#### 资源分配图算法（Resource-Allocation-Graph Algorithm）

此算法适合各类资源只有一个实例，是一个有向图。此图有三类边，第一类：Pj--->Rj表示进程P等待资源R，第二类：Rj-->Pj表示资源R已经分配给了进程P。第三类：虚线Pj-->Rj,表示进程P再将来会需要资源R。

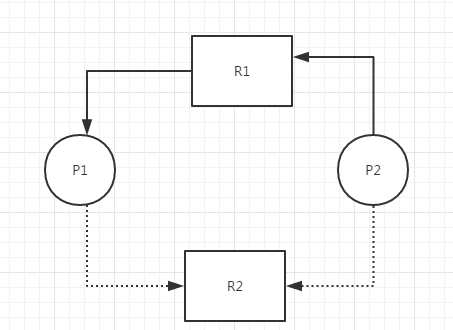


图1

如图1所示，进程P1和P2在运行过程中都需要使用R1和R2资源，而此时，R1资源已经分配给了进程P1,而进程P2正在等待资源R1的释放。此图中不存在环路，所以是安全状态（safe state）

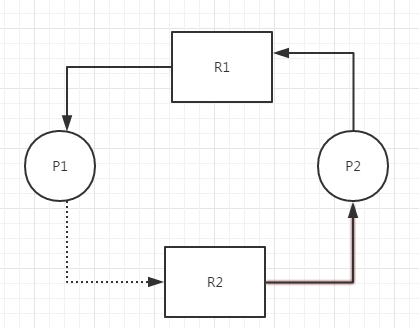


图2

假设在此时，P2进程请求资源R2，而如果R2分配给进程P2,此时如果P1在运行的过程中，如果需要使用R2的资源，就会等待，此时P1和P2两个进程就会进入循环等待的状态，成为死锁。再看图2，图中会有回路，说明此时有可能造成死锁，此时就是不安全状态（unsafe State）

#### 银行家算法（Banker's Algorithm）

银行家算法是银行保证已有的现金至少能满足一个顾客的要求。

资源分配图算法适合每类资源是一个实例，而应对与每类资源存在多种实例，采用银行家算法能有效的解决这种问题，但是效率会不如资源分配图算法。

**算法描述：**

当一个进程进入系统，请求分配资源的时候，必须声明自己需要的资源最大数，这个最大数不能超过该资源能提供的最大数量。一个进程请求分配资源，而系统必须计算，该资源分配给该进程后，是否处于安全状态，如果处于安全状态，该资源可以分配给进程，如果不处于安全状态，该进程只能处于等待状态。

实现该算法，需要几种数据结构：

n代表该操作系统的进程数，m代表资源的种类。

Available:代表m中资源当前可以获得的数量。available[j]代表当前j资源可以获得的实例数量。

Max：n×m阶矩阵，代表n个线程在m种资源上可能获取的最大资源数。

Allocation:n×m阶矩阵，代表n个线程在m中资源上，已经分配的资源数量。

Need:n×m阶矩阵，代表n个线程在m种资源上，还需要的资源数量。Need[i][j]=Max[i][j]-Allocation[i][j]

定义两个向量X和Y的大小，X<=Y,表示对于X和Y中，所有的X[i]<=Y[i]，X<Y表示X<=Y并且X!=Y

##### 安全算法（Safety Algorithm）

此算法可以计算出，当前系统所处的状态，是否是安全的。

**初始化：**

Work向量=Available向量，Finish[i]=false对所有的线程来说。

while（(Finish[i]=false 并且 Need[i]<=Work[i])否则 i++继续遍历，直到遍历一遍后发现没有满足条件的i）{

Work=Work+Allocation[i]

Finish[i]=true

i=0

}

**结果断定：**

对于所有的i，finish[i]=true，表示当前系统处于安全状态（safe state），此次分配任务可以完成。这个算法的本质就是找到一个进程执行序列，保证所有进程都可以顺利完成任务。

##### 资源请求算法（Resource-Request Algorithm）

此算法过程就是在某个进程请求资源后，先假设把资源分配了给该进程，然后检查分配后的资源是否处于安全状态（safe state），如果处于，就真正的分配资源，如果不处于安全状态，就不分配，该进程必须等待。

**算法描述：**

**1.**某个进程Pi的请求为Request[i]，Request[i][j]代表进程i请求资源j的数量。如果Request[i]<=Need[i]，进入第二步，否则抛出异常，该进程所申请的资源总数大于该进程最初声明的max数量。

**2**.如果Request[i]<=Available[i]，进入第三步，否则表示当前资源不足以满足进程i的需求，需要进行等待。

**3.**假设，进行状态变更。Available=Available-Request[i],Allocation=Allocation+Request[i],Need[i]=Need[i]-Request[i]

如果状态变更后，系统还是安全状态（safe state），此时可以将资源分配给进程i,如果系统进入了不安全状态，那么进程i必须进行等待。

### 死锁检测（Deadlock Detection）

如果一个系统不提供任何策略来避免死锁的发生，那么该系统必须提供两种方式：

1.检测死锁的发生。

2.死锁发生后，让程序从死锁中进行恢复。

#### 每种资源一个实例（Single instance of each resource type）

每类资源只有一个实例，在这种算法中，需要构建一个Resource-allocation graph 然后根据该图形构建一个 wait-for graph。wait-for graph中pj-->pi表示进程j正在等待i对资源进行释放，对应于resource-allocation graph图形中pj-->Rq and Rq -->pi这两条边存在。

在wait-for graph中，当且仅当图中存在回路的时候，才确定这个系统出现了死锁的情况。死锁的检测，可以定期执行一次死锁检测算法。

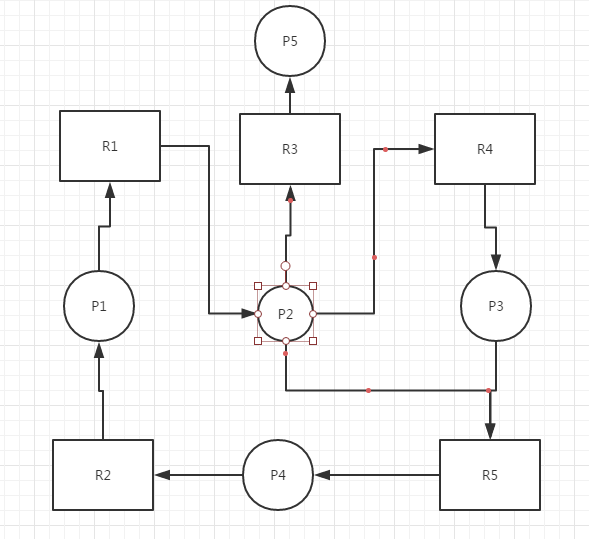


图1

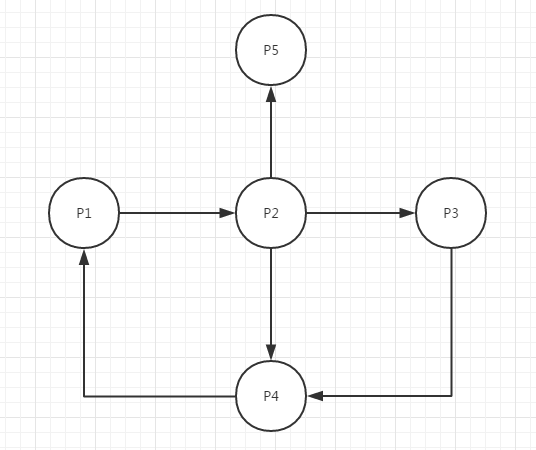


图2 wait-for graph

#### 每种资源有多个实例（several instance of a resource type）

等待图标（wait-for graph）不能满足一种资源存在多个实例的情况，适合多种资源的算法类似于银行家算法。

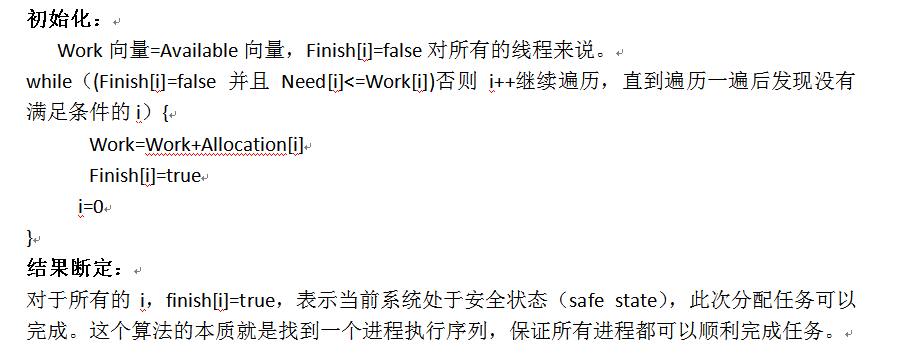
**数据结构：**

Available:表示每一种资源拥有的个数。长度为m。

Allocation：矩阵m乘n，表示当前进程已经分配的资源的数量。

Request：矩阵m乘n，表示当前进程需要请求的资源的数量。

算法描述：



此算法的作用依然是找到一个安全序列，可以保证所有的进程可以正常的结束任务，如果不存在这样的序列，则说明已经进入死锁状态。

#### 监测算法的使用（Detection-Algorithm Usage）

死锁监测应该如何使用呢？

两个因素：

1.How often is a deaclock likely to occur?

2.How many processes will be affected by deadlock when it happens?

根据系统出现死锁的频率来断定使用死锁监测算法。例如：进程不能得到临界资源的时候或者CPU的利用率下降到40%的时候。

### 死锁恢复（recovery from deadlock）

恢复死锁主要有两个策略：

#### 结束进程（process termination）

结束进程有两种方式：

1.Abort all deadlocked process.

2.Abort one process at a time until the deadlock cycle is eliminate.

#### 资源竞争（Resource Preemption）

让处于死锁的线程中，某些临界资源可以被竞争。

A.选择一个牺牲品。（elect a victim）

选择哪个资源以及那个进程来被竞争。需要考虑进程执行的时间等因素来保证成本最低。

B.回滚。（RollBack）

把进程回滚到一个安全的状态。或者回滚到最开始的状态。

C.饥饿（starvation）

需要解决某些进程饥饿的问题。

## 总结

死锁的产生就是多个进程对某个临界资源进行争用导致的。一共有三种策略来解决：

1.采用一些协议，来预防或避免死锁，以此来保证系统从来不进入死锁状态。

2.采用死锁监测算法，当系统进入死锁状态时，可以及时监测出来，并进行合理的处理。

3.假设，系统从来不进入死锁状态。不做任何处理。