**操作系统概念读书笔记**

**段卫东**

**2017-04-06**

目录

[死锁 2](#_Toc479351800)

[死锁的四个条件 2](#_Toc479351801)

[解决死锁的方法 2](#_Toc479351802)

[死锁预防（DeadLock Prevention） 2](#_Toc479351803)

[死锁避免（DeadLock Avoidance） 4](#_Toc479351804)

[死锁检测（Deadlock Detection） 6](#_Toc479351805)

[死锁恢复（recovery from deadlock） 9](#_Toc479351806)

[总结 9](#_Toc479351807)

[内存管理（Memory Management） 10](#_Toc479351808)

[背景（background） 10](#_Toc479351809)

[基本的硬件（basic hardware） 10](#_Toc479351810)

[地址绑定（Address binding） 10](#_Toc479351811)

[逻辑空间和物理空间（Logical versus physical address space） 10](#_Toc479351812)

[动态加载（dynamic loading） 10](#_Toc479351813)

[动态链接共享程序包（Dynamic linking and shared libraries） 11](#_Toc479351814)

[交换（swapping） 11](#_Toc479351815)

[连续内存分配（contiguous memory allocation） 11](#_Toc479351816)

[内存映射和保护（memory mapping and protection） 11](#_Toc479351817)

[内存的分配（memory allocation） 11](#_Toc479351818)

[内存碎片(fragmentation) 12](#_Toc479351819)

# 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。

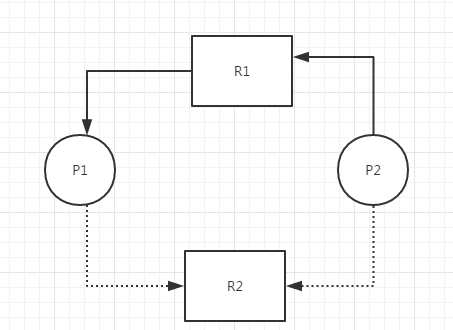


图1

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）

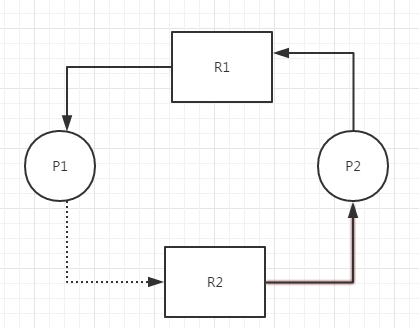


图2

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。

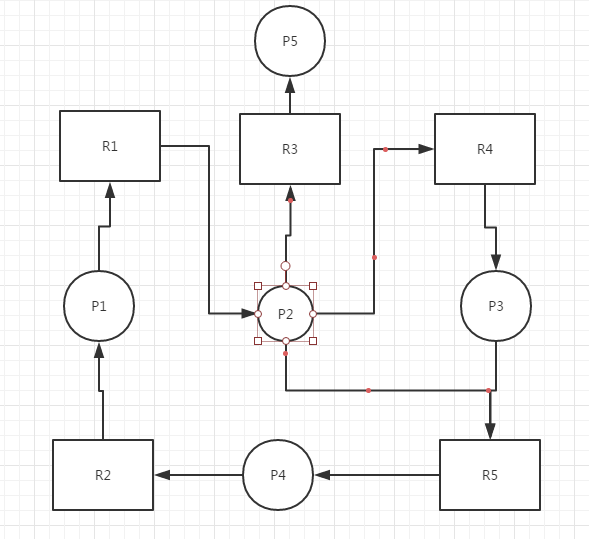


图1

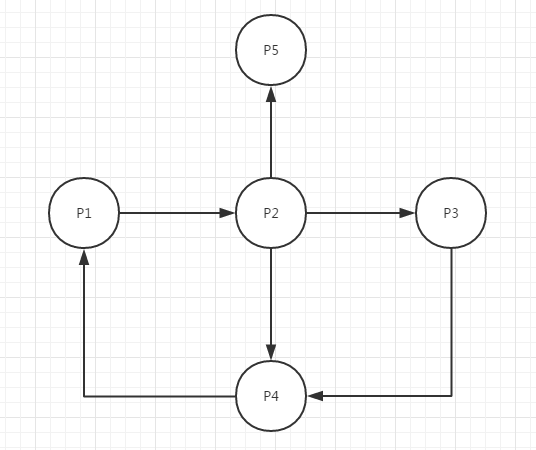


图2 wait-for graph

#### 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。

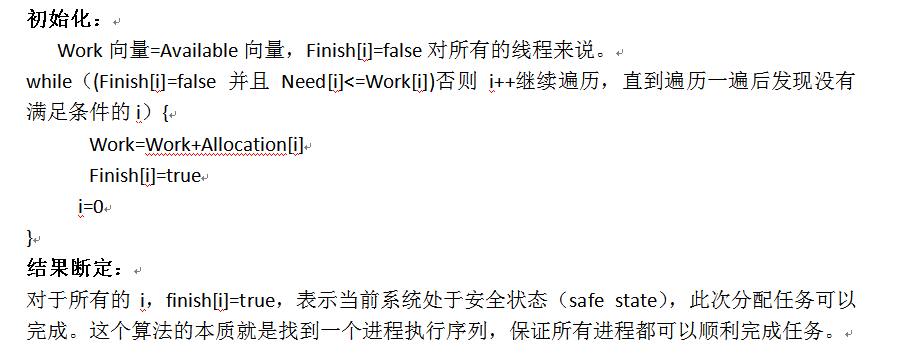
**数据结构：**

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。

# 内存管理（Memory Management）

章节目标：

1.介绍多重内存硬件的组织方式。

2.介绍多重内存管理技术，包括分页（paging）和分片（segmentation）

3.详细介绍因特尔奔腾技术，支持纯粹的分片(segmentation)和分片于分页整合（segmentation with paging）。

## 背景（background）

内存管理涉及到多方面的知识背景，此章节根据一些背景进行介绍。

### 基本的硬件（basic hardware）

地址分为逻辑地址和物理地址，而逻辑地址到物理地址的映射，就需要专门的寄存器来配合。基础寄存器（base register），限制寄存器(limit register),这两个寄存器内数据不能被用户随意改动，所以只能没操作系统的代码进行操作。另外，CPU与内存速度不匹配的问题，中间需要增加缓存（cache）来解决.种种这些问题，都需要提供相应的硬件来作为支撑。

### 地址绑定（Address binding）

程序平时是存储在磁盘上，当需要运行的时候，才加载到内存中运行。指令和数据绑定到内存上，可以在一下三个步骤中进行：

1.编译时间（compile time）

编译的时候可以进行数据绑定到内存上。

2.加载时间（load time）

3.执行时间（execution time）

### 逻辑空间和物理空间（Logical versus physical address space）

区分逻辑空间和物理空间，用户在进行代码编写的时候，不用关注真正的物理地址是什么，只当程序正在使用一块连续的逻辑空间即可。一块逻辑空间可以映射成分散的几个物理空间。

### 动态加载（dynamic loading）

动态加载可以提高内存的利用率，当程序执行的时候，从来不适用的模块，是不需要从磁盘加载到内存的。动态加载也不需要操作系统的特殊支持，只需要操作系统提供加载程序到内存的方式即可。

### 动态链接共享程序包（Dynamic linking and shared libraries）

动态链接共享程序包，是为了节省磁盘和内存使用的。比如程序a和程序B都要使用读文件这个服务，需要操作系统读文件的程序包，如果不使用动态链接，程序A和B都要拷贝一份程序包到响应的程序中，在运行的时候加载到内存，各自执行。而使用动态链接，则可以A和B只把程序包加载到内存一次就可以了，就可以一直使用了。而此时也要注意控制程序包的版本，各个用户程序需要使用的程序包的版本。

## 交换（swapping）

交换涉及到很多交换的策略，此处只是大体的介绍一下交换的含义。即CPU可能同时执行很多process，而这些进程又不能一次全部加载到内存当中，这就需要不停的进行内存和磁盘的换入（swap in）和换出(swap out)操作.但是换入和换出也需要考虑一些问题。

**问题：**

**1.**如果每次换入和换出的都是整个进程的资源，假设一个进程的大小是10MB，这种换入和换出的开销还是很大的，是否可以考虑仅仅换出在程序运行过程中有改动的内容。

2.如果换出的进程正在等待I/O，则不可以换出，没有等待I/O的可以换出。而如果该程序使用的是异步I/O，那该程序换出的话，可能存有数据的BUFFER已经是换入程序的内存范畴了。解决此问题，可以考虑异步IO分配buffer的时候占用操作系统的内存，而数据从buffer到用户内存的拷贝，只能发生在该程序在内存中的时候。

## 连续内存分配（contiguous memory allocation）

内存分配分为连续内存分配和不连续内存分配。

### 内存映射和保护（memory mapping and protection）

内存映射是指用户代码里使用的内存地址都是相对于0开始的逻辑地址，在程序真正被执行的时候，才会被映射成物理地址。另外，映射成物理地址的寄存器，也只能被操作系统更改，不能被用户随意修改，这就保护了用户数据不被随意操作。

### 内存的分配（memory allocation）

内存的分配有很多机制。

第一种：将内存划分为固定的几个模块。每个块被一个进程所使用，当有进程需要运行的时候，就给该进程分配一个模块，当进程结束的时候，就将其移出内存，然后把下一个需要运行的进程移入。这种方式在IBM的OS/360上使用，后来就再也没被使用过。

第二种：由操作系统维护一张大表，当有进程需要运行的时候，就为其分配一块连续的内存，并记录。依次进行分配，当该进程结束的时候，就释放该内存。查找适合进程的内存块有三种方式：

A.First fit.找到第一块时候的内存，就结束查找。

B.Best Fit。遍历，找到一个适合该进程运行的最小内存。但是会造成一些内存碎片不能被使用。

C.Worst Fit。遍历，找到一个适合该进程的最大内存。这种会尽量减少不能被使用的内存碎片。

不过通过模拟，First fit 和best fit比较使用，其中，first fit 是时间性能上表现更好。

### 内存碎片(fragmentation)

内存碎片可以通过两种方式解决，不同的解决方式产生不同的内存碎片。

一种是internal 另一种是external。

**internal:**

内存可以被划分为几个比较小的块，当程序运行需要内存的时候，则可以给程序分配多个内存块，这样就会造成总会比程序需要的内存多分配一点内存。

**external:**

在external中产生的不可利用的内存块，操作系统可以压缩（compaction），即整理零散的内存块，但是整理内存块也不是总是使用的，对于编译时间(compile time)进行地址映射或者加载时间（load time）进行的内存映射，这种方式不适合，只适合运行时（execution time）进行的内存映射。操作系统移动程序的位置的同事修改响应的寄存器里面的数值（basic register, limit register），保证正常映射到正确的位置。

## 分页（paging）

分页（paging）是一种内存管理机制，可以实现程序分配到物理内存是不连续的。分页引起的内存碎片化是internal。即当某一页物理内存没有被完全利用的时候。

实现分页的方式：硬件、硬件和操作系统结合。

#### 基本的方式（basic method）

最基本的分页方式就是把内存分割成一块一块的，叫做frame,逻辑地址也分成大小相同的块，叫做page。这样逻辑地址就可以根据page number 和page offset来进行定位自己在物理地址的哪个位置上。

操作系统为每个进程维护一个page table，来清楚的知道每个逻辑地址映射到物理地址的哪个位置上。而这个page table是存储在进程块上的。

#### 硬件支持(hardware support)

逻辑地址和物理地址之间存在一个映射关系，所以每次CPU访问数据的时候，都需要先访问pagetable映射到绑定的物理地址，然后再真正的访问物理地址。这样取数需要访存两次，大大降低了访存的效率。为了降低访存的性能损失，出现了硬件支持。

TLB(translation look-aside buffer):是由硬件实现的一个键值对。但是呢，空间有限，只能存储pagetable的一部分。如果访问的KEY在TLB中不存在，则需要进一步访问内存，找到相应的pagetable，然后取出映射。这会造成TLB的hit rate问题，想办法提高hit rate也是一个很好的方式。里面涉及到一些替换算法，以及TLB中kernel mode状态执行的代码是不可被替换的。

#### 权限保护（Protection）

内存中的数据，并不是所有的process都可以访问的，所以有权限限制，比如read-only，write，执行权限。这就需要在page table中增加一个bit,来表示该页数据的权限情况。对每个程序分配了一个内存块，但是这个用户只能访问其中一部分内存。另外一部分系统才可以访问，需要增加该页文件的一个有效位。有效或者无效来表示该数据是否可以访问。但是这种情况又会造成internal fragmentation . 而process实际运行过程中，并不会用到自己所有的内存，所以page table映射所有的逻辑内存，又会存在内存浪费的情况。这时候会增加一个字段，来记录该程序的page table的大小，以此来节约空间。