OPERATING SYSTEMS

# Monitors & Deadlocks

## Operating Systems

郑贵锋 博士
中山大学计算机学院
zhenggf@mail.sysu.edu.cn
https://gitee.com/code_sysu

中山大學 SUN YAT-SEN UNIVERSITY



- **目录**
  - 同步硬件
  - 互斥锁
  - 信号量
  - 经典问题
  - 管程
    - 管程的概念
    - 管程条件
    - 使用管程的示例
    - 管程的实现
    - 在管程中恢复进程
  - 死锁
    - 系统模型
    - 死锁特征
    - 银行家算法
  - 同步示例

## Monitors



- **管程的概念**
  - 当程序员错误地使用信号量来解决临界区问题时，很容易产生各种类型的错误。
    - 所有进程共享一个信号量变量mutex，该变量被初始化为1。每个进程必须在进入临界区之前执行wait(mutex)，然后执行signal(mutex)。
    - 如果不遵守此顺序，则两个进程可能同时处于其临界区。
    - 很难检测到这些错误，因为它们只在特定的执行顺序发生时发生，并且这些顺序并不总是发生。
    - 这种情况可能是由无意的编程错误或不合作的程序员造成的。
  - 后面的示例1和示例2向我们展示了可能导致的各种困境。请注意，即使单个进程表现不好，也会出现这些困境。

## Monitors



- **管程的概念**
  - 示例1
    - 假设某个进程交换对信号量互斥体执行wait()和signal()操作的顺序，从而导致以下执行：

```
signal(mutex);
...
临界区
...
wait(mutex);
```

    - 在这种情况下，多个进程可能同时在其临界区执行，从而违反互斥要求。
    - 只有当几个进程在其临界区同时处于活动状态时，才能发现此错误。请注意，这种情况可能并不总是可以重现的。

## Monitors



- **管程的概念**
  - 示例2
    - 假设一个进程将signal(mutex)替换为wait(mutex)。也就是说，它执行：

```
wait(mutex);
...
临界区
...
wait(mutex);
```

    - 在这种情况下，将发生死锁。
  - 为了处理这些错误，研究人员开发了高级语言结构。
    - 管程类型是一种基本的高级同步构造，它为进程同步提供了方便有效的机制。
      - 在许多并发编程语言中都可以找到

## Monitors



- **管程的概念**
  - 管程的用法
    - 管程类型是一种抽象数据类型(ADT)，它包括一组程序员定义的操作，这些操作在管程中是互斥的。
    - 管程类型还声明了一些变量，其值定义ADT实例的状态，以及对这些变量进行操作的函数体。
    - 管程的布局

```
monitor monitor-name
{
    shared variable declarations
        ...
    function P1 ( … ) { … }
    function P2 ( … ) { … }
        …
    function Pn ( … ) { … }

    Initialization code (…) { … }
}
```

## Monitors

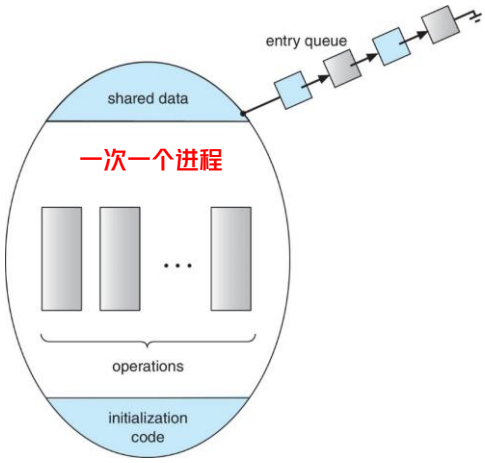


- **管程的概念**
  - 管程类型
    - *共享：*管程由并发进程共享。
    - *封装和安全性：*管程的表示不能直接由各种进程使用。
      - 管程中定义的任何函数只能访问管程中本地声明的变量及其形式参数。
      - 管程的局部变量只能由局部函数访问。
    - *互斥：*管程构造确保管程中一次只有一个进程处于活动状态。
      - 程序员不需要显式地编写这个同步约束。
        - 因此，通过将共享数据放在管程中来保护它们。
        - 管程在进程条目上锁定共享数据。
    - 管程可以通过信号量实现。

## Monitors



- **管程的概念**
  - 管程类型
    - 管程的示意图



## Monitors



- **管程条件**
  - 条件类型
    - 管程的功能不足以对某些同步方案进行建模。
    - 条件构造为程序员实现进程同步提供了额外的同步机制。
      - 条件类型的变量：

```
condition x, y;
```

      - 条件变量是管程的本地变量(仅可在管程内访问)。
    - 唯一可以在条件变量上调用的操作是在关联队列上执行的条件等待cwait和条件信号csignal。
    - 例如，条件等待操作

```
x.wait();
```

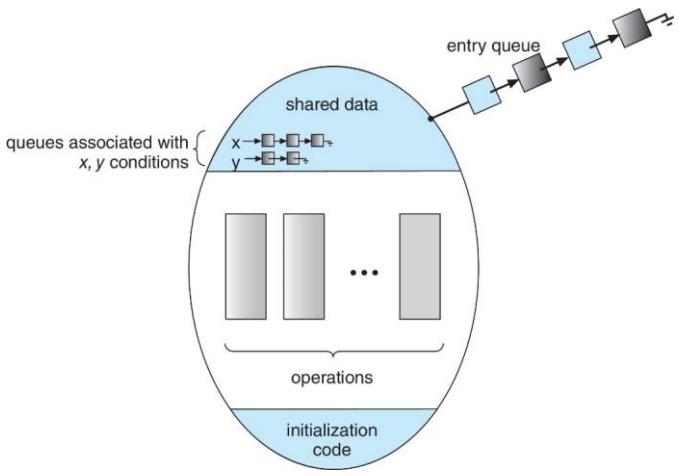
      表示调用此操作的进程被挂起，直到另一个进程调用

```
x.signal();
```

## Monitors



- **管程条件**
  - 带条件变量的管程
    - 等待进程要么在入口队列中，要么在条件队列中。



## Monitors

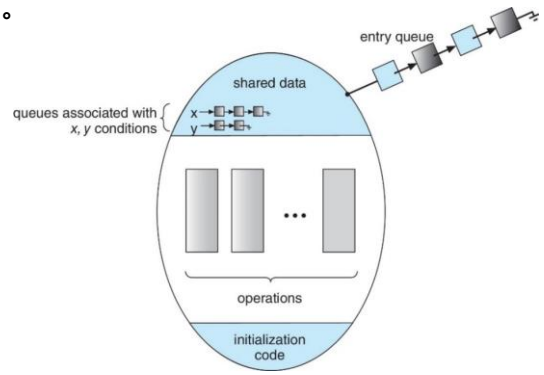


- **管程条件**
  - cwait和csignal操作
    - 进程同步由程序员使用条件变量完成
      - 在管程中执行之前，进程可能需要等待条件。
    - cwait操作x.wait()；或cwait(x)；
      - x.wait()表示调用此操作的进程被挂起/阻塞，直到另一个进程调用x.signal()
    - csignal操作x.signal()；或csignal(x)；
      - x.signal()只恢复调用x.wait()的一个挂起进程。如果未挂起任何进程，则csignal操作无效
      - 将此操作与相关的正常signal(mutex)信号量操作进行对比，后者总是通过mutex.value++来影响信号量的状态
    - 可能需要一个额外的整数计数器x_count来描述条件x的等待队列(条件队列)的长度。

## Monitors

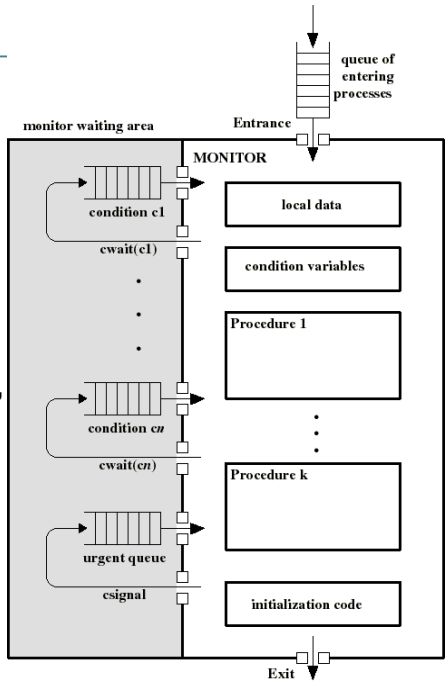


- **管程条件**
  - cwait和csignal操作
    - 等待进程要么在入口队列中，要么在条件队列中。
    - 进程通过发出cwait(cn)将自己放入条件cn队列。
    - csignal(cn)将条件cn队列中的一个进程带入管程。
      - 请记住，管程是互斥的，因此csignal(cn)会阻塞调用进程并将其放入紧急队列(等待进入管程)，除非csignal是管程过程的最后一个操作。



## Monitors

- **管程条件**
  - cwait和csignal操作—一种可能的动态
    - 等待进程要么在入口队列中，要么在条件队列中。
    - 进程通过发出cwait(cn)将自己放入条件cn队列。
    - csignal(cn)将条件cn队列中的一个进程带入管程。
      - 请记住，管程是互斥的，因此csignal(cn)会阻塞调用进程并将其放入紧急队列(等待进入管程)，除非csignal是管程过程的最后一个操作。




## Monitors



- **管程条件**
  - cwait和csignal操作
    - 假设对于条件x，进程P调用x.signal()，进程Q挂起在x.wait()中。接下来会发生什么？
      - 管程是互斥的，Q和P不能并行执行。如果Q恢复，那么P必须等待。
    - 选项包括：(*Tony Hoare&Per B.Hansen,* 1974)
      - *唤醒并等待 Signal-and-wait*：P要么等待Q离开管程，要么等待其他条件。(Hoare的方法)
        - P发出信号并在紧急队列中等待或等待另一种条件。Q恢复执行。
      - *唤醒并继续 Signal-and-continue*：Q要么等待P离开管程，要么等待另一种条件。
        - P发出信号并继续。Q在紧急队列中等待。
        - 正方：P已在管程中执行，让P继续下去似乎是合理的
        - 反方：当Q从紧急队列恢复时，Q等待的逻辑条件x可能不再有效。这可能会导致系统开销。

## Monitors



- **管程条件**
  - cwait和csignal操作
    - 假设对于条件x，进程P调用x.signal()，进程Q挂起在x.wait()中。接下来会发生什么？
      - 管程是互斥的，Q和P不能并行执行。如果Q恢复，那么P必须等待。
    - 选项包括：(*Tony Hoare&Per B.Hansen,* 1974)
      - 改进的 *Signal-and-wait* (*Hansen*)
        - signal()是离开管程前的最后语句。
        - 当P发出信号时，它立即离开管程。
        - Q立即恢复
      - 这些选择各有利弊 – 可由编程语言实现者决定。
    - 并发Pascal
      - P执行signal()立即离开管程；恢复Q
    - 已实现这些操作的其他语言包括Mesa、C#、Java.

## Monitors



- **使用管程的示例**
  - 用管程解决生产者-消费者问题
    - 同步现在仅限于管程内。
    - append()和take()是管程中的过程/函数
      - 是生产者或消费者访问缓冲区的唯一方式。
    - 生产者

```
repeat
    produce v;
    append(v);
forever
```

    - 消费者

```
repeat
    take(v);
    consume v;
forever
```

## Monitors



- **使用管程的示例**
  - 用管程解决有界缓冲生产者-消费者问题
    - 需要保存缓冲区：
      - buffer: array[n] of items;
    - 需要两个条件变量：
      - notfull:csignal(notfull)表示缓冲区未满。
      - notempty:csignal(notempty)表示缓冲区不是空的。
    - 需要缓冲区指针和计数：
      - nextin: 指向要追加的下一项。
      - nextout: 指向下一项。
      - count: 保存缓冲区中的项目数。

## Monitors



- **使用管程的示例**
  - 用管程解决有界缓冲生产者-消费者问题
    - 共享数据和条件：

```
struct items array[n];
int nextin = 0, nextout = 0, count = 0;
condition notfull, notempty;
```

    - Procedure append(v):

```
if (count == n) cwait(notfull);
buffer[nextin] = v;
nextin = (nextin + 1) mod n;
count++;
csignal(notempty);
```

    - Procedure take(v):

```
if (count == 0) cwait(notempty);
v = buffer[nextout];
nextout = (nextout + 1) mod n;
count--;
csignal(notfull);
```

## Monitors



- **使用管程的示例**
  - 用管程解决有界缓冲生产者-消费者问题
    - PASCAL语言代码实现管程

```
Type pc = monitor
Var buffer: array[0,..,n-1] of items;
    notfull, notempty: condition;
    nextin, nextout, count: integer; /* initially 0 */
Procedure entry append(v);
  Begin
    if (count >= n) then notfull.wait;
    buffer[nextin]:= v;
    nextin:= (nextin + 1) mod n;
    count:= count + 1;
    notempty.signal;
  End;
Procedure entry take(v);
  Begin
    if (count = 0) notempty.wait;
    v:= buffer[nextout];
    nextout:= (nextout + 1) mod n;
    count:= count – 1;
    notfull.signal;
  End;
```

## Monitors



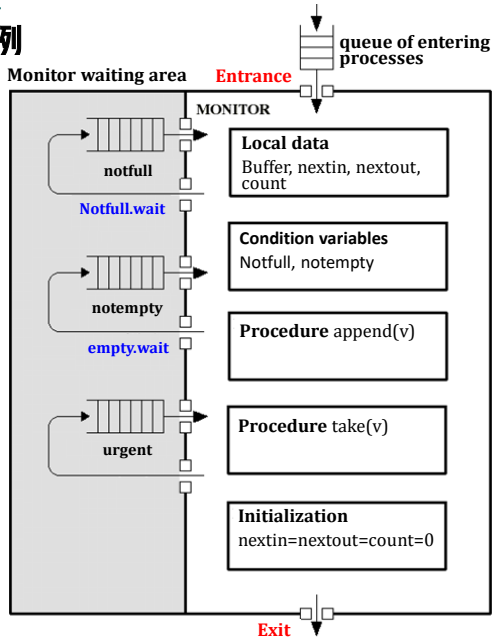
- **使用管程的示例**
  - 用管程解决有界缓冲生产者-消费者问题
    - PASCAL语言代码实现生产者-消费者

```
Begin
  nextin:= nextout:= count:= 0;
End.

Producer:
  Begin
    repeat
      produce an item v;
      pc.append(v);
    until false
  End;

Consumer:
  Begin
    repeat
      pc.take(v);
      consume the item v;
    until false
  End;
```

## Monitors



- 使用管程的示例




## Monitors



- 使用管程的示例
  - 用管程解决哲学家就餐问题
    - 这是该问题的无死锁解。
      - 限制哲学家只在两种筷子都可用的情况下才能拿起筷子
      - 使用enum枚举类型定义哲学家的三种状态：

```
enum {THINKING, HUNGRY, EATING} state[5];
```

    - HUNGRY的哲学家i(i=0..4)在她的两个邻居不在吃饭

```
(state[(i + 4) % 5] != EATING) &&
(state[(i + 1) % 5] != EATING)
```

    时，可以将自己的状态设定为EATING

```
state[i] = EATING;
```

    - 我们还需要声明用于等待的条件

```
condition self[5];
```

      - 这使得哲学家i在她HUNGRY但无法获得所需筷子时可以推迟自己的时间。

## Monitors



- **使用管程的示例**
  - 用管程解决哲学家就餐问题

```
1.  monitor DiningPhilosophers
2.  {
3.      enum { THINKING; HUNGRY, EATING } state [5];
4.      condition self [5];

5.      void pickup(int i) {
6.          state[i] = HUNGRY;
7.          test(i);        /* 尝试拿筷子吃饭 */
8.          if (state[i] != EATING)
9.              self[i].wait();
10.     }

11.     void putdown(int i) {
12.         state[i] = THINKING;
13.         test((i + 4) % 5); /* 左邻居尝试 */
14.         test((i + 1) % 5); /* 右邻居尝试 */
15.     }
```

## Monitors



- **使用管程的示例**
  - 用管程解决哲学家就餐问题

```
16.     void test(int i) {/* 哲学家i尝试拿起筷子吃饭 */
17.         if ((state[(i + 4) % 5] != EATING) &&
18.                         (state[i] == HUNGRY) &&
19.               (state[(i + 1) % 5] != EATING) ) {
20.             state[i] = EATING;
21.             self[i].signal();
22.         }
23.     }

24.     initialization_code() {
25.         for (int i = 0; i < 5; i++)
26.             state[i] = THINKING;
27.     }
28. }
```

## Monitors



- **使用管程的示例**
  - 用管程解决哲学家就餐问题
    - 示例
      - 让 state[2] = EATING, state[3] = THINKING, state[4] = EATING
      - 并发进程

```
pickup(3); putdown(2); putdown(4);
```

      - 一种可能的时间交错：进程3等待左右邻居吃完饭才能吃

| Pro(2) #Line | state[2] | Pro(3) #Line | state[3] | Pro(4) #Line | state[4] |
|---|---|---|---|---|---|
| | EATING | | THINKING | | EATING |
| | | 6 | HUNGRY | | |
| | | 7 | waiting | | |
| 12 | THINKING | | | | |
| 13 | | | | | |
| 14 | | | | | |
| | continuing | | | 12 | THINKING |
| | | | EATING | 13 | |
| | | | resuming | 14 | continuing |

## Monitors



- **使用管程的示例**
  - 用管程解决哲学家就餐问题
    - 每个哲学家i调用操作pickup()和putdown()，如下所示：

```
DiningPhilosophers.pickup(i);
  /* eating */
DiningPhilosophers.putdown(i);
```

    - 该解决方案确保没有两个邻居同时吃饭。
    - 这是一个无死锁的解决方案，但是饥饿是可能的。

## Monitors



- **管程的实现**
  - 使用信号量实现管程
    - 共享数据：
      - mutex：二值信号量，初始化为1。
        - 为每个管程提供，以确保管程互斥
      - next：二值信号量，初始化为0。
        - 默认情况下使用Hoare方案（唤醒并等待），进程可以使用signal(next)唤醒next进程时挂起自己。
      - next_count：整数变量，初始化为0。
        - 提供用于统计next暂停的进程数。

## Monitors



- **管程的实现**
  - 使用信号量实现管程
    - 共享数据：

```
semaphore mutex;  /* initially to 1 */
semaphore next;   /* initially to 0 */
int next_count = 0;
```

    - 每个外部程序F将替换为

```
wait(mutex);
body of F
if (next_count > 0)
    signal(next);
  /* 因有挂起在next信号量的进程，唤醒其中一个进程 */
else
  /* 无进程挂起，则F发mutex信号量 */
    signal(mutex);
```

## Monitors



- **管程的实现**
  - 使用信号量实现管程
    - 每个条件变量x的更多共享数据：

```
semaphore x_sem;  /* 初始为 0 */
int x_count = 0;  /* 等待 x 的进程数 */
```

    - x.wait()的实现：

```
x_count++;
if (next_count > 0)
    signal(next);  /* 唤醒紧急队列中的一个进程 */
else
    signal(mutex); /* 打开管程的入口 */
wait(x_sem);       /* 加入条件x队列  */
x_count--;         /* 退出条件x队列  */
```

    - x.signal()的实现：

```
if (x_count > 0) {
    next_count++;
    signal(x_sem);  /* 在条件x队列中唤醒一个进程 */
    wait(next);     /* 加入紧急队列(Hoare)*/
    next_count--;   /* 退出紧急队列 */
}                   /* 否则x.signal什么都不做 */
```

## Monitors



- **在管程中恢复进程**
  - 在管程中恢复进程
    - 如果在条件x下挂起了多个进程，且某个进程执行了x.signal()，那么接下来应该恢复哪个进程？
    - 如果没有简单的FCFS（先到先得）解决方案，可以为进程增加优先级，使用以下形式的条件等待(*conditional-wait*)构造

```
x.wait(c);
```

    其中c称为优先级编号。
    - 当一个进程调用x.wait(c)并挂起时，优先级号c与挂起的进程一起存储。
    - 当执行x.signal()时，具有最小优先级的进程将继续执行。

## Monitors



- **在管程中恢复进程**
  - 示例：用于分配单个资源的管程
    - 使用优先级编号在竞争进程之间分配单个资源，优先级编号指定进程计划使用该资源的最长时间。

```
R.acquire(t);
        ...
access the resource;
        ...
R.release;
```

其中R是ResourceAllocator管程类型的实例。

## Monitors



- **在管程中恢复进程**
  - 示例：用于分配单个资源的管程

```
monitor ResourceAllocator
{
    boolean busy;
    condition x;

    void acquire(int time) {
        if (busy)
            x.wait(time);
        busy = TRUE;
    }

    void release() {
        busy = FALSE;
        x.signal();
    }

    initialization_code() {
        busy = FALSE;
    }
}
```

## Monitors



- **在管程中恢复进程**
  - 示例：用于分配单个资源的管程
    - 不幸的是，管程概念不能保证遵守前面的访问顺序。特别是，可能会出现以下问题：
      - 进程可能在未获得资源访问权限的情况下访问该资源
      - 进程可能在获得资源的访问权后，永远不会释放该资源
      - 进程可能试图释放从未请求过的资源。
      - 一个进程可能会请求同一资源两次(不首先释放资源)。
    - 可能的解决方案：将资源访问操作内置在资源分配管程中
      - 资源调度也是管程内置，而非我们自己编写的调度算法
    - 检查所有使用资源分配管程及其管理资源的程序，确保进程遵守适当顺序：用户程序必须按正确顺序调用管程；不合作的进程不能简单忽略管程提供的互斥，不能试图直接访问资源
      - 对大的或动态的系统不现实（需附加机制来解决）

## Deadlocks 课本第7章



- **系统模型**
  - 资源和实例
    - 一个系统由有限数量的资源组成，这些资源在多个相互竞争的进程之间分配
    - 资源被划分为若干类型，每个类型由若干相同的实例组成。
      - 存储空间
      - CPU周期
      - 文件
      - I/O设备：打印机、DVD驱动器...
    - 如果进程请求某种资源类型的一个实例，则该类型的任意实例的分配都将满足该请求。
    - 进程必须在使用资源之前请求资源，并且必须在使用资源之后释放资源。
    - 进程为执行其指定任务，可以请求所需的尽可能多的资源。
    - 请求的资源数量不得超过系统中可用的资源总数。

## 系统模型

- 使用资源的顺序
  - *申请：*进程请求资源。如果无法立即满足请求，则请求进程必须等待，直到它能够获取资源。
  - *使用：*进程可以对资源操作。
  - *释放：*进程释放资源。
- 资源管理
  - 系统表：记录每个资源是空闲的还是已分配的；对于已分配的每个资源，该表还记录正使用该资源的进程。
  - 等待队列：如果某个进程申请当前已分配给另一个进程的资源，则可以将申请进程添加到等待此资源的进程队列中。
- 死锁
  - 当一组进程中的每个进程都在等待一个事件，而这一事件只能由该组中的另一个进程引起，那么这组进程就处于死锁状态。



## 死锁特征

- 必要条件
  - 如果系统中同时存在以下四种情况，则可能出现死锁情况：
  - 互斥：必须至少一个资源以非共享模式持有；也就是说，一次只有一个进程可以使用该资源。如果另一进程申请该资源，则申请进程必须延迟，直到资源被释放。
  - 保持等待：一个进程必须至少持有一个资源，并等待获取其他进程当前持有的其他资源。
  - 非抢占：资源不能被抢占；也就是说，资源只能由持有它的进程在该进程完成其任务后自愿释放。
  - 循环等待：等待进程的集合 $\{P_0, P_1, \ldots, P_n\}$ 必须存在，使得$P_0$正在等待$P_1$持有的资源，$P_1$正在等待$P_2$持有的资源，…，$P_{n-1}$正在等待$P_n$持有的资源，$P_n$正在等待$P_0$持有的资源。(这蕴含保持等待条件)

## Deadlocks



# 死锁特征

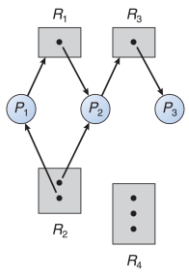
- 资源分配图
  - 死锁可以用称为**系统资源分配图**的有向二部图更精确地描述。
  - 该图由一个顶点集合V和一个边集合E组成。
  - 集合V由两种不同类型的节点组成
    - $P = \{P_1, P_2, ..., P_n\}$, 系统中所有**活动进程**组成的集合
    - $R = \{R_1, R_2, ..., R_m\}$, 系统中所有**资源类型**组成的集合
  - 集合E由两种不同类型的边组成
    - **申请边**：从进程到资源类型的有向边 $P_i \rightarrow R_j$，表示进程$P_i$已申请资源类型$R_j$的实例，并且当前正在等待该资源
    - **分配边**：从资源类型到进程的有向边 $R_j \rightarrow P_i$，表示资源类型$R_j$的实例已分配给进程$P_i$

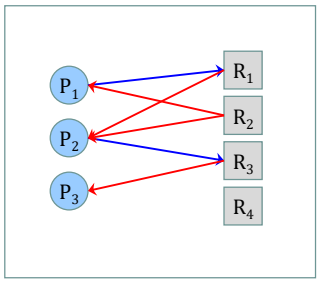
## Deadlocks



# 死锁特征

- 资源分配图
  - 资源分配图的示例
    - $R_1$和R3都只有一个实例。
    - $R_2$有两个实例。
    - $R_4$有三个实例。
    - $P_1$持有$R_2$的一个实例，正等待$R_1$的实例。
    - $P_2$持有$R_1$的一个实例，持有$R_2$的一个实例，正等待$R_3$的实例。
    - $P_3$持有$R_3$的一个实例。



## Deadlocks



- **死锁特征**
  - 资源分配图
    - 图中有环是死锁存在的必要条件，但不是充分条件。
    - 给定资源分配图的定义，可以证明：
      - 如果图中没有环，则系统中就没有进程处于死锁状态。
      - 如果图中包含一个只涉及一组资源类型的环，且每个资源类型都只有一个实例，则发生了死锁。环中涉及的每个进程都处于死锁状态。

## Deadlocks

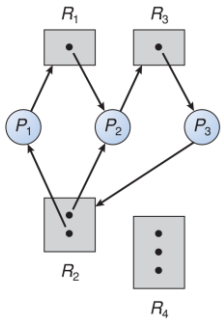


- **死锁特征**
  - 资源分配图
    - 实例
      - 具有死锁的资源分配图
        - 系统中存在两个最小环：

$$P_1 \rightarrow R_1 \rightarrow P_2 \rightarrow R_3 \rightarrow P_3 \rightarrow R_2 \rightarrow P_1$$

$$P_2 \rightarrow R_3 \rightarrow P_3 \rightarrow R_2 \rightarrow P_2$$

        - 进程$P_1$、$P_2$和$P_3$处于死锁状态。
        - 进程$P_2$正在等待由进程$P_3$持有的资源$R_3$。进程$P_3$正在等待进程$P_1$或进程$P_2$释放资源$R_2$。此外，进程$P_1$正在等待进程$P_2$释放资源$R_1$
        - 注意，这里$R_2$有2个实例，但图中有2个环

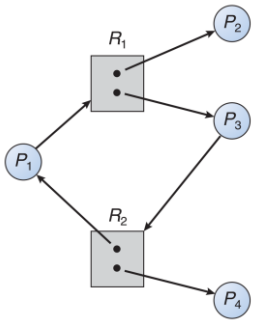


## Deadlocks



- **死锁特征**
  - 资源分配图
    - 实例
      - 具有环但没有死锁的资源分配图
        - 系统中存在一个环：

          $P_1 \rightarrow R_1 \rightarrow P_3 \rightarrow R_2 \rightarrow P_1$
        - 没有死锁。
        - 进程$P_4$可能会释放其资源类型$R_2$的实例。然后可以将该资源分配给$P_3$，从而打破环。
        - 当进程$P_2$释放其资源类型$R_1$的实例时，也会发生同样的情况。




## Deadlocks



- **死锁特征**
  - 处理死锁的方法
    - 死锁问题可通过以下三种方式之一处理：
      - M1
        - 使用协议预防或避免死锁，确保系统永远不会进入死锁状态。
      - M2
        - 允许系统进入死锁状态，检测并恢复。
      - M3
        - 完全忽略这个问题，认为系统中不可能发生死锁。

## Deadlocks



- **死锁特征**
  - 处理死锁的方法
    - M1.预防和避免
      - 为了确保死锁不会发生，系统可以使用死锁预防或死锁避免方案。
      - 死锁预防提供了一组方法，用于确保四个必要条件中至少有一个无法保持。
        - 这些方法通过限制资源请求的方式来防止死锁。
      - 死锁避免要求提前向操作系统提供有关进程在其生命周期内将请求和使用哪些资源的附加信息。

## Deadlocks



- **死锁特征**
  - 处理死锁的方法
    - M2. 检测与恢复
      - 如果系统不采用死锁预防或死锁避免算法，则可能出现死锁情况。
      - 在这种环境中，系统可以提供一种算法来检查系统的状态，以确定是否发生了死锁，并提供一种从发生的死锁中恢复的算法。

## 死锁特征

- 处理死锁的方法
  - M3. 忽略
    - 尽管忽略死锁似乎不是解决死锁问题的切实可行的方法，但它仍然在大多数操作系统中使用。
    - 在许多系统中，死锁很少发生(例如，每年一次)；因此，这种方法比必须经常使用的预防、避免或检测和恢复方法更便宜。
    - 系统可能处于冻结状态，但不处于死锁状态。
      - 例如，以最高优先级运行的实时进程，或在非抢占式调度程序上运行的任意进程。
      - 处于冻结状态的系统中，进程不会将控制权返回到操作系统。
      - 必须具有针对此类情况的手动恢复方法，并且可以简单地将这些技术用于死锁恢复。



## 银行家算法

- 资源分配图算法不适用于每个资源类型有多个实例的资源分配系统
- 银行家算法是一种适用于此类系统的死锁避免算法，但其效率低于资源分配图方案。
- 当一个新线程进入系统时，它必须声明它可能需要的每种资源类型的最大实例数。此数量不得超过系统中的资源总数。
- 当用户请求一组资源时，系统必须确定这些资源的分配是否会使系统处于安全状态。
  - 若会，资源被分配；
  - 否则，该线程必须等待其他线程释放足够的资源。

## Deadlocks



- **银行家算法**
  - 全局数据结构
    - 假设系统中有m种类型的资源和n个需要资源的进程。

```
int available[m];
int work[m]; /* 假定的可用资源 */
```

    - available向量表示每种类型的可用资源数。 available[j] == k表示资源类型$R_j$的实例有k个可用。

```
int max[n][m];
```

    - 矩阵max定义了每个线程的最大需求。max[i][j] == k表示线程$T_i$最多可以请求资源类型$R_j$的k个实例。

```
int allocation[n][m] = {0};
```

    - 矩阵allocation定义了当前分配给每个线程的每种类型的资源数量。allocation[i][j] == k表示线程$T_i$当前分配了资源$R_j$的k个实例。

## Deadlocks



- **银行家算法**
  - 全局数据结构
    - 假设系统中有m种类型的资源和n个需要资源的进程。

```
int need[n][m] = {0};
```

    - 矩阵need指示每个线程的剩余资源需求。need[i][j]==k意味着线程$T_i$可能还需要k个资源类型为$R_j$的实例来完成其任务。
    - 注意

```
need[i][j] == max[i][j] - allocation[i][j].
```

  - 安全状态与不安全状态
    - 系统状态反映了当前分配给进程的资源。因此，状态由available向量和两个矩阵(need和allocation)组成。
    - 安全状态是指，系统按一定顺序给进程分配资源，不会导致进程死锁的状态，即所有进程都可以运行到完成。

## 银行家算法

- 定义
  - 设X和Y是长度为m的两个向量。
    - X ≤ Y 当且仅当 X[i] ≤ Y[i] 对于所有i=1, 2, …, m成立
    - 此外，X < Y 如果 Y ≤ X 且 Y ≠ X
    - 我们需要$O$(m)来判断条件是否成立 X ≤ Y
- 记法
  - 对于矩阵M[n][m]，我们使用M[i]表示矩阵M的第i行向量。



## 银行家算法

- 安全算法 – safetycheck()
  - 数据结构

```
#define TRUE 0
#define FALSE -1
int finish[n] = {FALSE};
int safe_seq[n];
```

  - 程序

```
for (i = 1; i <= n; i++) {    /* 需要进行n次检查，记下顺序 */
    for (j = 0; j < n; j++) { /* 选择一个进程j    */
        if (finish[j] == FALSE    /* Tj尚未测试   */
            && need[j] <= work) { /* 且资源足够Tj完成它的任务     */
                work = work + allocation[j]; /* 假定Tj释放其资源*/
                finish[j] = TRUE; /* Tj 已测试     */
                safe_seq[i] = j;  /* 记下安全顺序   */
                break; /* 标记下一进程i */
        } /* 否则选择下一进程j测试 */
    }
    if(j == n)  /* 第i次，找不到通过安全测试的进程j */
        break;
}
if (i > n) /* n个进程均可完成 */
    return EXIT_SECCESS;
else /* 存在进程不够资源完成 */
    return EXIT_FAILURE;
```

**判断条件 need[j] <= work 的复杂度是$O$(m)**

safetycheck()的复杂度为$O(n^2m)$

## Deadlocks



- **银行家算法**
  - 资源请求算法。
    - 数据结构

```
int request[m]; /* 线程Ti对资源的申请 */
```

    - 程序

```
input request from process Ti
if (request > need)
    exit(1); /* Ti的申请已超过其声明的最大需求 */
if (request > available)
    exit(1); /* 申请的资源超过可用资源 */

  /* 假定分配 */
work = available - request;
allocation[i] = allocation[i] + request;
need[i] = need[i] - request;

ret = safetycheck(); /* 安全检查 */
return ret;          /* 如果返回EXIT_FAILURE需要回滚分配 */
```

  - 阅读课本7.5.3的例子

## Deadlocks



- **银行家算法**
  - 死锁避免的局限性
    - 银行家算法需要知道一个进程可能需要多少资源。在大多数系统中，这些信息不可用，因此无法实现银行家算法。
    - 假设进程的数量是静态的是不现实的，因为在大多数系统中，进程的数量是动态变化的。
    - 进程最终释放其所有资源(当进程终止时)的要求对于算法的正确性来说是足够的，但是对于实际系统来说是不够的。等待数小时(甚至数天)来释放资源通常是不可接受的。

## 银行家算法

- 死锁避免的局限性
  - 死锁避免策略不能确定地预测死锁；它只是预测了死锁的可能性，并确保永远不存在这种可能性。
  - 死锁避免的优点是不需要像死锁检测那样抢占和回滚进程，并且局限性比死锁预防小。但是，它的使用确实有一些限制：
    - 必须提前声明每个进程的最大资源需求。
    - 所考虑的进程必须是独立的；也就是说，它们的执行顺序必须不受任何同步要求的限制。
    - 必须分配固定数量的资源。
    - 持有资源时，任何进程都不能退出。



## 银行家算法

- 算法. 17-1-bankers-5.c

```
lab@ByteDanceServer:~/os/lec17$ ./a.out
==== initial state ====
 available       need            allocation
10, 5, 7,        7, 5, 3,        0, 0, 0,
                 3, 2, 2,        0, 0, 0,
                 9, 0, 2,        0, 0, 0,
                 2, 2, 2,        0, 0, 0,
                 4, 3, 3,        0, 0, 0,
```

## Deadlocks



- **银行家算法**
  - **算法**. 17-1-bankers-5.c

```
==== safecheck starting ====
safe_seq[0] = 1
new work vector: 3, 2, 2,
safe_seq[1] = 3
new work vector: 5, 2, 4,
safe_seq[2] = 4
new work vector: 9, 5, 7,
safe_seq[3] = 0
new work vector: 9, 5, 7,
safe_seq[4] = 2
new work vector: 10, 5, 7,
A safty sequence is: P1, P3, P4, P0, P2,
==== new state ====
 available       need            allocation
 0, 0, 0,        7, 5, 3,        0, 0, 0,
                 0, 0, 0,        3, 2, 2,
                 8, 0, 2,        1, 0, 0,
                 0, 2, 0,        2, 0, 2,
                 0, 0, 0,        4, 3, 3,
```

通过安全检查，
全部进程可完成

## Deadlocks



- **银行家算法**
  - **算法**. 17-1-bankers-5.c

```
==== new state ====
 available       need            allocation
 5, 5, 2,        7, 5, 0,        0, 0, 3,
                 3, 2, 2,        0, 0, 0,
                 9, 0, 0,        0, 0, 2,
                 0, 2, 2,        2, 0, 0,
                 1, 3, 3,        3, 0, 0,
input process number (0 - 4, -1 quit): 4
input request vectore of length 3, separated by a space: 0  0 2
pro_i = 4
request = 0, 0, 2,
==== safecheck starting ====
safecheck failed, process 4 suspended
```

此时，
如果给进程4分配资源0 0 2，
将不能通过安全检查。
为什么？

因为分配后，
将没有进程可以完成任务。