第 3 章　系统级软件开发中的死锁问题

3.1　死 锁 概 述

死锁是指多个任务因争夺资源而相互等待，最终导致系统停滞的状态。死锁会导致系统资源无法被正常利用，进而影响系统的稳定性和性能。在多任务系统软件开发中，死锁的产生原因可归纳为以下两点。

（1）竞争系统资源。当多个任务同时竞争有限的系统资源时，如果它们各自持有部分资源并等待对方释放其他资源，就可能形成死锁。例如，在数据库系统中，多个事务可能同时请求锁定不同的数据行，从而相互阻塞。

（2）任务的执行推进顺序不当。如果任务的推进顺序不当，进而形成一个循环等待链，可能导致死锁。如图 3.1 所示，系统中存在两个并发执行的进程，分别是 P1 和 P2。在它们的执行过程中，两者都需要同时访问资源 R1 和 R2，而 R1 和 R2 都是一次只能被一个进程使用的临界资源。其中，进程 P1 先申请资源 R1，再申请资源 R2，等同时使用完资源 R1 和

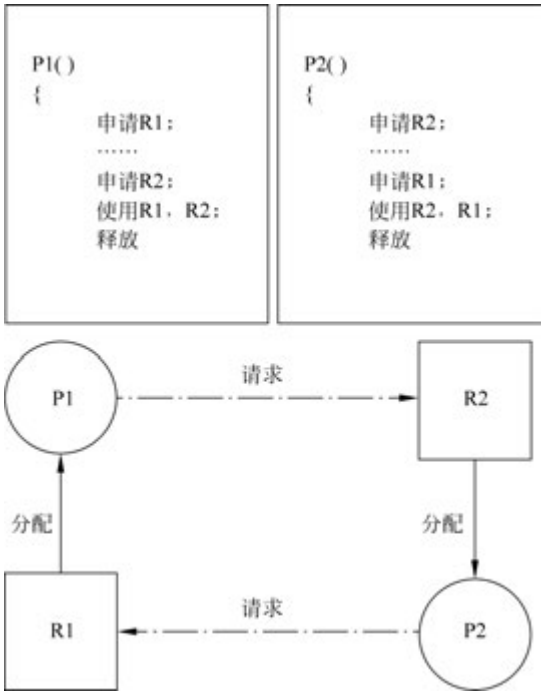

图 3.1　死锁示意图

R2 后，再分别释放；相反，进程 P2 先申请资源 R2，再申请资源 R1，也是等同时使用完资源 R2 和 R1 后，再分别释放；在这种资源分配方式下，存在一种可能，恰好进程 P1 申请成功资源 R1，进程 P2 申请成功资源 R2，此时进程 P1 和 P2 需要分别再次申请资源 R2 和 R1 时，造成 P1 和 P2 互相等待对方释放资源，进而导致两个进程无法继续执行，形成死锁。

死锁发生时满足以下 4 个必要条件(也称为 Coffman 条件)。

1) 互斥条件(Mutual Exclusion)

定义：至少有一个资源必须处于非共享模式，即一次只能由一个任务使用。如果另一个任务请求该资源，请求者将被阻塞，直到资源被释放。

示例：如打印机、文件等，一次只能被一个进程使用。

2) 占有并等待条件(Hold and Wait)

定义：一个已经占有了某种资源的任务可能还会申请新的资源，而这些新资源可能暂时无法分配给它，但是这个任务不会释放已经占有的资源。

示例：任务 A 已经持有了资源 1，但还需要资源 2 才能继续执行。此时，如果资源 2 被任务 B 持有，任务 A 将等待任务 B 释放资源 2，同时不会释放已占有的资源 1。

3) 非抢占条件(No Preemption)

定义：已经分配给一个任务的资源不能被抢占，只有该任务显式地释放资源后，才能把资源分配给其他任务。

示例：任务 A 持有了资源 1，即使系统需要资源 1 来执行其他进程，也不能从任务 A 那里强行剥夺资源 1。

4) 循环等待条件(Circular Wait)

定义：存在一个任务集合$\{T_1, T_2, \cdots, T_n\}$，其中 T_1 正在等待 T_2 占有的资源，任务 T_2 正在等待 T_3 占有的资源，……，T_n 正在等待 T_1 占有的资源，形成一个环路。

示例：任务 A 等待任务 B 持有的资源，任务 B 等待任务 C 持有的资源，而任务 C 又在等待任务 A 持有的资源，形成一个循环等待链。

3.2 死锁的解决策略

解决死锁可以采取三种策略：破坏死锁必要条件，利用银行家算法等避免死锁，死锁检测和解除。

1. 破坏死锁必要条件

破坏互斥条件：这通常不是一个实际可行的方案。因为互斥条件是由资源的本质属性决定的，某些资源由于其物理或逻辑特性，必须互斥使用。例如，打印机、文件等公共资源在任意时刻只能由一个任务访问，否则会导致数据不一致或其他问题。因此，企图通过破坏互斥条件来防止死锁是不太实际的。

破坏占有并等待条件：可以在任务申请资源时，要求其一次性申请整个运行过程所需的全部资源。如果所需资源得不到满足，则任务必须等待，直到所有资源都可用时才能开始执行。这样可以避免任务在持有部分资源的同时又请求其他资源的情况。该方法会导致降低资源利用率，并增加任务饿死的风险。

破坏非抢占条件：允许资源被剥夺，即当一个任务占有的资源需要被其他任务使用时，

可以强制剥夺该任务占有的资源，破坏不可剥夺条件。但该方法可能导致任务异常中断，甚至数据丢失。

循环等待条件：资源有序分配法是破坏循环等待条件的一种有效方法。系统给每类资源赋予一个全局序号，每个任务按编号递增的顺序请求资源。这样，每个任务在请求资源时都会遵循一个固定的顺序，从而避免了形成环路等待。例如，在银行转账场景中，可以按照账户编号来顺序获取资源，避免死锁的发生。

2. 死锁避免策略

银行家算法(Banker's Algorithm)是一种著名的死锁避免算法，主要用于操作系统中的资源分配管理。其核心思想是通过事先预判进程请求资源后是否会导致系统进入不安全状态，从而决定是否满足该请求。如果请求满足后系统仍处于安全状态，则分配资源；否则，拒绝请求或让进程等待。

虽然银行家算法在理论上避免死锁是可行的，但在实际的软件开发中直接应用是非常有限的，主要原因有以下几点。

(1) 银行家算法主要用于操作系统层面的资源分配，如 CPU、内存、I/O 设备等。在软件开发中通常不会直接应用操作系统级别的资源分配算法。

(2) 银行家算法需要维护系统的资源状态，包括可用资源、已分配资源、最大需求等信息。在软件开发中，特别是高层应用中，实现和维护这样的算法可能过于复杂且开销较大。

(3) 银行家算法主要用于需要严格控制资源分配的系统，如实时系统、高可靠性系统等。在大多数软件开发中，资源分配通常不是主要关注点，更多的是关注业务逻辑、用户体验和性能优化。

3. 进行死锁的检测和解除

一旦检测出死锁，就需要立即采取措施来解除死锁，以恢复系统的正常运行。常见的解除死锁的方法如下。

(1) 回滚：将系统回滚到发生死锁之前的状态，并重新执行相关操作。这种方法可以确保系统的一致性，但可能会导致之前的工作丢失。

(2) 重启：重启受影响的进程或整个系统，以清除死锁状态。这种方法简单直接，但可能会导致数据丢失或服务中断。

(3) 抢占资源：强制抢占一个或多个任务持有的资源，并将其分配给其他任务。这种方法需要谨慎使用，以避免引起数据不一致或系统崩溃。

由于系统软件开发倾向于使用更简单、更轻量级的机制来解决死锁问题，综上所述，银行家算法的复杂性、开销以及与应用层并发控制机制的不匹配，死锁检测和解除策略会导致一系列问题，包括性能下降、资源浪费、系统崩溃等，而在破坏死锁必要条件策略中，资源有序分配法是解决死锁问题常用的简便有效的一种策略，但需要在具体应用场景中进行权衡和选择。

3.3 编程技巧与实践

1. 死锁预防及避免编程技巧

(1) 确保互斥锁的顺序，多个任务在访问多个资源时，应确保它们总是以相同的顺序申

请锁。这样可以避免循环等待条件，从而预防死锁。

（2）优化互斥锁的使用，尽量避免嵌套使用互斥锁，尽量缩短持有锁的代码段，减少锁的持有时间。

（3）在请求互斥锁时使用超时机制，如果无法在一定时间内获得锁，则释放已经持有的锁并重试。

（4）当任务请求资源时，设置一个超时时间。如果在超时时间内无法获得所有资源，线程就释放已持有的资源并重试。

【编程实例 3.1】 下面是一个可能会发生死锁的实例。实例中创建了两个线程及两个互斥锁 mutex1 和 mutex2，由于两个线程使用临界区中资源的顺序不同，则线程 1 先使用互斥锁 mutex1，再使用互斥锁 mutex2；线程 2 先使用互斥锁 mutex2，再使用互斥锁 mutex1，并且两个线程都在同时获取两个互斥锁后，再分别释放两个互斥锁，代码如下：

```
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <unistd.h>

//创建并初始化互斥锁 mutex1 和 mutex2
pthread_mutex_t mutex1 = PTHREAD_MUTEX_INITIALIZER;
pthread_mutex_t mutex2 = PTHREAD_MUTEX_INITIALIZER;

//线程 1 执行函数
void* thread1(void* arg) {
     pthread_mutex_lock(&mutex1);
     printf("Thread 1 acquired mutex1\n");
     sleep(1);
     pthread_mutex_lock(&mutex2);
     printf("Thread 1 acquired mutex2\n");
     pthread_mutex_unlock(&mutex2);
     pthread_mutex_unlock(&mutex1);
     pthread_exit(NULL);
}

//线程 2 执行函数
void* thread2(void* arg) {
     pthread_mutex_lock(&mutex2);
     printf("Thread 2 acquired mutex2\n");
     sleep(1);
     pthread_mutex_lock(&mutex1);
     printf("Thread 2 acquired mutex1\n");
     pthread_mutex_unlock(&mutex1);
     pthread_mutex_unlock(&mutex2);
     pthread_exit(NULL);
}

int main(void) {
     pthread_t tid1, tid2;
     //创建线程 1
     if (pthread_create(&tid1, NULL, thread1, NULL) != 0) {
          perror("Thread 1 creation failed");
```

```
            return 1;
        }
        //创建线程 2
        if (pthread_create(&tid2, NULL, thread2, NULL) != 0) {
            perror("Thread 2 creation failed");
            return 1;
        }
        //等待线程 1 和线程 2 执行结束
        pthread_join(tid1, NULL);
        pthread_join(tid2, NULL);
        return 0;
    }
```

这段代码演示了一个潜在的死锁情况，其中两个线程试图以不同的顺序获取两个互斥锁(mutex1 和 mutex2)。由于线程的执行顺序是非确定性的，这段代码的运行结果可能会有所不同。以下是可能的一种发生死锁的运行结果，如图 3.2 所示。

```
rocos2204@rocos2204-virtual-machine:~/桌面$ gcc deadlock_ex1.c -lpthread -o deadlock_ex1
rocos2204@rocos2204-virtual-machine:~/桌面$ ./deadlock_ex1
Thread 1 acquired mutex1
Thread 2 acquired mutex2
```

图 3.2　死锁示例程序运行结果

在这个结果中，线程 1 首先获取了 mutex1，然后睡眠了 1 秒。与此同时，线程 2 获取了 mutex2，然后也睡眠了 1 秒。当两个线程都醒来后，线程 1 试图获取 mutex2，而线程 2 试图获取 mutex1。由于这两个互斥锁都已经被对方持有，因此两个线程都将被阻塞，等待对方释放互斥锁，从而导致了死锁，程序无法继续往下执行，进而无法正常结束。当然，由于线程调度和睡眠时间的非确定性，也有可能出现线程 1 或线程 2 成功获取两个互斥锁并完成执行的情况，但在多次运行这段代码时，很可能会至少遇到一次死锁。

2. 死锁预防编程实例

本示例以哲学家就餐问题为例。哲学家就餐问题(如图 3.3 所示)是指有五位哲学家围坐在圆桌旁，每位哲学家左右各有一支筷子。哲学家们需要交替思考(不需要筷子)和就餐(需要拿起左右两根筷子)。如果哲学家们同时拿起左右筷子，就可能导致死锁(例如，每位哲学家都拿起了左边的筷子并等待右边的筷子，而右边的筷子被相邻的哲学家拿着)。采用资源有序分配法，可以规定每位哲学家必须先拿起编号小的筷子(假设从左到右编号递增)，然后再拿起编号大的筷子。这样，即使所有哲学家同时开始就餐，也不会发生死锁，因为每个哲学家都会按照固定的顺序申请筷子。

图 3.3　哲学家就餐

【编程实例 3.2】 下面是使用资源有序分配法解决哲学家就餐问题的编程实例。实例中使用线程来模拟哲学家，通过互斥锁来表示筷子，并给筷子进行编号，哲学家须按照筷子编号(先左后右)的顺序来获取筷子。

```
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

#define NUM_PHILOSOPHERS 5                    //哲学家的数量
#define NUM_CHOPSTICKS 5                      //筷子的数量

//定义互斥锁数组,每个元素代表一支筷子
pthread_mutex_t chopsticks[NUM_CHOPSTICKS];

//哲学家线程函数
void *philosopher(void *num) {
    int id = *(int *)num;                     //从参数中解析哲学家编号(0～4)
    int left_chopstick = id;                  //左筷子编号与哲学家编号相同
    int right_chopstick = (id + 1) % NUM_CHOPSTICKS;      //右筷子编号+1(环形排列)

    while (1) {
        //先思考(随机1-3秒)
        printf("Philosopher %d 正在思考\n", id);
        sleep(rand() % 3 + 1);

        //按照编号循环,先尝试获取左筷子
        printf("Philosopher %d 尝试拿起左筷子 %d\n", id, left_chopstick);
        pthread_mutex_lock(&chopsticks[left_chopstick]);  //阻塞直到左筷子可用
        printf("Philosopher %d 拿起了左筷子 %d\n", id, left_chopstick);

        //再尝试获取右筷子
        printf("Philosopher %d 尝试拿起右筷子 %d\n", id, right_chopstick);
        pthread_mutex_lock(&chopsticks[right_chopstick]); //阻塞直到右筷子可用
        printf("Philosopher %d 拿起了右筷子 %d\n", id, right_chopstick);

        //左右两支筷子都拿到后,就餐(随机1-3秒)
        printf("Philosopher %d 开始进餐\n", id);
        sleep(rand() % 3 + 1);

        //先释放右筷子
        pthread_mutex_unlock(&chopsticks[right_chopstick]);
        printf("Philosopher %d 放下了右筷子 %d\n", id, right_chopstick);

        //再释放左筷子
        pthread_mutex_unlock(&chopsticks[left_chopstick]);
        printf("Philosopher %d 放下了左筷子 %d\n", id, left_chopstick);
    }

    return NULL;
}

int main(void)
{
    pthread_t threads[NUM_PHILOSOPHERS];          //存储线程ID的数组
    int philosopher_ids[NUM_PHILOSOPHERS];        //哲学家编号数组
```

```
    //初始化所有互斥锁(筷子)
    for (int i = 0; i < NUM_CHOPSTICKS; i++) {
        pthread_mutex_init(&chopsticks[i], NULL);
    }

    //创建哲学家线程
    for (int i = 0; i < NUM_PHILOSOPHERS; i++) {
        philosopher_ids[i] = i;                          //为每个哲学家分配编号
        //创建线程,传入哲学家编号指针
        pthread_create(&threads[i], NULL, philosopher, &philosopher_ids[i]);
    }

    //等待所有线程结束(实际因无限循环不会执行到此)
    for (int i = 0; i < NUM_PHILOSOPHERS; i++) {
        pthread_join(threads[i], NULL);
    }

    //销毁互斥锁
    for (int i = 0; i < NUM_CHOPSTICKS; i++) {
        pthread_mutex_destroy(&chopsticks[i]);
    }

    return 0;
}
```

代码运行结果如图 3.4 所示。

```
rocos2204@rocos2204-virtual-machine:~/桌面$ gcc deadlock_ex2.c -lpthread -o deadlock_ex2
rocos2204@rocos2204-virtual-machine:~/桌面$ ./deadlock_ex2
Philosopher 0 is thinking.
Philosopher 1 is thinking.
Philosopher 2 is thinking.
Philosopher 3 is thinking.
Philosopher 4 is thinking.
Philosopher 2 is trying to pick up chopstick 2.
Philosopher 2 picked up chopstick 2.
Philosopher 2 is trying to pick up chopstick 3.
Philosopher 2 picked up chopstick 3.
Philosopher 2 is eating.
```

图 3.4　哲学家就餐程序运行结果

示例中定义了 NUM_PHILOSOPHERS 和 NUM_CHOPSTICKS 为 5,表示有 5 位哲学家和 5 支筷子。每位哲学家都有一个唯一的编号,并且根据编号来确定左右筷子的编号。哲学家会按照编号的顺序来获取筷子。

在这个示例运行结果中,可以看到哲学家 0 首先开始思考,然后哲学家 2 尝试并成功拿起了他左右两边的筷子(互斥锁),接着开始就餐。就餐结束后,他放下筷子(释放互斥锁),然后再次开始思考。这个过程对于每个哲学家都是相同的,并且他们会不断地重复这个过程。由于每个哲学家在尝试拿起筷子时都会遵循一定的顺序(先左后右),并且由于互斥锁的存在,他们不会同时拿起同一支筷子,因此这个实现避免了死锁的发生。本实例程序并没有设置终止条件,所以它会无限地运行下去,直到被外部强制终止,读者可以添加一个退出条件或者信号来终止这些线程,比如通过设置一个全局变量来请求线程终止。

3.4 本章小结

在多任务系统软件开发中，死锁是一个普遍存在的问题，当发生死锁时，会导致两个或多个任务因为彼此互相依赖而陷入无限等待的状态，无法继续运行下去，导致整个系统无响应的现象，系统的 CPU 资源被浪费，进程的运行被阻塞，对系统的性能和可用性产生负面影响。为了防止死锁现象的发生，务必在设计和开发过程中注重资源管理和锁的使用，采用合适的策略，进而保证系统的稳定和高效运行。

3.5 习 题 3

1. 请阐述产生死锁的原因，并举例说明。

2. 请阐述产生死锁的 4 个必要条件。

3. 请阐述破坏死锁的 4 个必要条件的可行性。

4. 有三个线程 P1、P2 和 P3，它们都需要使用资源 R1、R2 和 R3。如果线程 P1 申请 R1 和 R2，线程 P2 申请 R2 和 R3，线程 P3 申请 R1 和 R3，它们申请资源的顺序不当，可能会导致死锁。编程模拟实现上述过程，并使用资源有序分配法避免死锁，要求写出完整代码。