 **导致死锁发生的四个基本条件有哪些？**

答：造成死锁的四个必要条件是：

* **互斥条件**：至少有一个资源处于被排他性使用的状态。
* **不可抢占条件**：已经分配给一个进程的资源，不能被强制抢占。
* **占有且等待条件**：一个进程至少占有一个资源，并且正在等待获取其他被别的进程占有的资源。
* **循环等待条件**：存在一个进程等待资源的循环链，每个进程等待链上的下一个进程所占有的资源。

 **请列出并解释现有的几种高级通信方法。**

答：当前常见的高级通信机制包括：

* **共享存储器系统**：多个进程通过共享一块内存区域进行数据交换和通信。这种方法速度快，但需要同步机制以防止数据冲突。
* **消息传递系统**：进程通过发送和接收消息来交换信息，消息可以是同步的或异步的，适用于分布式系统。
* **管道通信系统**：利用管道作为数据传输的通道，常用于具有父子关系的进程之间的数据交换，适用于简单的单向或双向通信。

 **多级反馈队列调度算法为何能较好地满足不同用户的需求？**

答：多级反馈队列调度算法通过设立多个队列并根据进程的行为动态调整其所在的队列，从而兼顾不同类型的作业需求：

* **短交互型作业**：优先在高优先级的队列中快速执行，确保响应时间短，用户满意。
* **中等长度的批处理作业**：分配适当的时间片，使其能够在有限时间内完成，保持较短的周转时间。
* **长时间运行的批处理作业**：通过逐级降低优先级，避免其长期占用资源，同时确保最终能得到执行，防止饥饿现象。

因此，多级反馈队列调度算法能够适应多种作业类型，满足不同用户的需求。

 **操作系统为何需要引入进程这一概念？**

答：引入进程概念使操作系统能够在多道程序环境下实现程序的并发执行，提供对并发程序的有效管理和调度，从而提高系统的整体效率和资源利用率。

1. **以下哪项不是操作系统的主要功能？（ ）**

A. 组织和管理计算机软硬件资源

B. 合理地对各类进程进行调度

C. 高级程序设计语言的编译处理

D. 为用户和计算机提供友好的交互界面

1. **操作系统采用多道程序设计的目的是什么？（ ）**

A. 有利于代码共享，减少主、辅存信息交换量

B. 充分利用存储器

C. 充分利用CPU，减少CPU等待时间

D. 提高实时响应速度

1. **指定当前工作目录的主要原因是什么？（ ）**

A. 节省内存空间

B. 节省外存空间

C. 加快文件的读写速度

D. 加快文件的检索速度

1. **假设有四个进程共享同一程序段，且每次最多允许三个进程进入该段，那么信号量的可能变化范围是什么？（ ）**

A. 3，2，1，0

B. 3，2，1，0，－1

C. 4，3，2，1，0

D. 2，1，0，－1，－2

1. **分区管理与分页管理的主要差异是什么？（ ）**

A. 分区中的块比分页中的页要小

B. 分页有地址映射而分区没有

C. 分页有存储保护而分区没有

D. 分区要求一道程序存放在连续的空间内而分页没有这种要求

1. **在以下进程状态转换中，哪一种是不可能发生的？（ ）**

A. 就绪→执行

B. 执行→就绪

C. 就绪→阻塞

D. 执行→阻塞

1. **以下哪项不是创建进程时必需的？（ ）**

A. 建立一个进程的进程表项

B. 为进程分配内存

C. 为进程分配CPU

D. 将进程表项放入就绪队列

1. **假设三个进程共享一个互斥区，每次最多允许两个进程进入该区域，信号量初值应设为多少？（ ）**

A. 0

B. 1

C. 2

D. 3

1. **如果两个进程共享一个临界资源，且互斥信号量mutex等于1时，表示什么？（）**

A. 一个进程进入了临界区，另一个进程等待

B. 没有一个进程进入临界区

C. 两个进程都进入临界区

D. 两个进程都在等待

1. **假如系统S1使用死锁避免策略，S2使用死锁检测策略，以下哪项描述是正确的？（ ）**

Ⅰ. S1会限制用户申请资源的顺序

Ⅱ. S1需要进行所需资源总量信息，而S2不需要

Ⅲ. S1不会给可能导致死锁的进程分配资源，S2会

A. 仅Ⅰ Ⅱ

B. 仅Ⅱ Ⅲ

C. 仅Ⅰ Ⅲ

D. Ⅰ Ⅱ Ⅲ

综合题：

某大学设有一间多媒体教室，配备了4m台投影仪。有4n名学生需要进行小组演示。规定：

1. 每四人一组，各占一台投影仪，协同完成演示任务；
2. 只有同组的四个人都到齐，并且有空闲的四台投影仪时，该组学生才能进入教室；
3. 演示结束后，教师检查，之后一组学生同时离开教室。

请使用P、V操作模拟演示过程。

**答案解析**

为了模拟上述演示过程，我们需要使用信号量来管理投影仪的资源和同步组内成员的到达。以下是详细的步骤和解释：

**1. 定义信号量和变量**

* **proj\_sem**：用于控制投影仪的数量，初始值为4m（即教室中可用的投影仪数量）。
* **mutex**：用于保护共享变量，防止竞态条件，初始值为1。
* **group\_count**：记录当前组内已到齐的学生数，初始值为0。
* **group\_sem**：用于同步同组内学生的到达，初始值为0。
* **teacher\_sem**：用于同步教师检查完成后，通知学生离开教室，初始值为0。

**2. 学生进程**

每个学生在到达教室时执行以下操作：

void student() {

// 学生到达教室，准备参与演示

mutex.P(); // 进入互斥区，保护group\_count

group\_count = group\_count + 1;

if (group\_count == 4) { // 当四名学生都到齐时

proj\_sem.P(); // 请求四台投影仪

group\_sem.V(4); // 唤醒其他三名学生

mutex.V(); // 释放互斥锁

} else {

mutex.V(); // 释放互斥锁

group\_sem.P(); // 等待其他组员到齐

}

// 四名学生一起进入教室，开始演示

perform\_presentation(); // 执行演示任务

// 等待教师检查完成

teacher\_sem.P();

// 演示结束，离开教室

proj\_sem.V(); // 释放投影仪

}

**3. 教师进程**

教师在演示完成后进行检查，并通知学生离开教室：

void teacher() {

while (1) {

wait\_for\_presentation\_completion(); // 等待演示完成的信号

inspect\_presentation(); // 检查演示

teacher\_sem.V(4); // 通知四名学生可以离开

}

}

**4. 详细解释**

* **信号量 proj\_sem**：
  + 初始值为4m，表示教室中有4m台投影仪可用。
  + 每次一组学生进入教室，proj\_sem.P() 操作会减少4m的值，表示四台投影仪被占用。
  + 学生离开教室后，通过 proj\_sem.V() 操作释放四台投影仪。
* **信号量 mutex 和变量 group\_count**：
  + mutex 保证对 group\_count 的互斥访问，防止多个学生同时修改 group\_count 导致不一致。
  + group\_count 用于记录当前组内已到齐的学生数。
  + 当 group\_count 达到4时，表示一组学生都到齐，可以进入教室。此时，执行 proj\_sem.P() 请求四台投影仪，并通过 group\_sem.V(4) 唤醒其他三名学生。
* **信号量 group\_sem**：
  + 初始值为0，表示学生需要等待其他组员到齐。
  + 当第四名学生到达时，通过 group\_sem.V(4) 同时唤醒其他三名等待的学生。
* **信号量 teacher\_sem**：
  + 初始值为0，表示学生需要等待教师检查完成。
  + 教师在检查完成后，通过 teacher\_sem.V(4) 通知四名学生可以离开教室。

**5. 流程总结**

1. **组内同步**：
   * 学生通过 mutex 和 group\_count 确保同一组的四名学生同时进入教室。
   * 第四名到达的学生负责请求四台投影仪，并唤醒其他组员。
2. **资源管理**：
   * proj\_sem 确保同一时间内最多有4m台投影仪被占用，避免资源冲突。
3. **教师同步**：
   * 教师在每组演示完成后进行检查，并通过 teacher\_sem 通知学生离开，释放资源。

通过上述P、V操作和信号量的合理使用，实现了对多组学生协同进入教室、使用投影仪以及有序离开的同步控制，确保了演示过程的有序进行。

系统中有三个进程，分别为 **Q1**、**Q2**、**Q3**，三个进程共享两个缓冲区 **C1** 和 **C2**。**C1** 中最多可以放入 **8** 个任务，现已放入了 **3** 个任务；**C2** 中最多可以放入 **5** 个任务，目前为空。**Q1** 进程负责不断地将任务添加到 **C1** 中，**Q2** 进程负责从 **C1** 中取出任务进行处理，并将处理后的任务送到 **C2** 中，**Q3** 进程负责从 **C2** 中取出任务执行。请写出实现 **Q1**、**Q2**、**Q3** 的同步与互斥的算法（要求：用类 C 语言描述；信号量原语统一使用 wait 和 signal）。

**答案解析**

为了实现 **Q1**、**Q2**、**Q3** 三个进程对缓冲区 **C1** 和 **C2** 的同步与互斥访问，我们需要使用信号量来管理缓冲区的资源，确保各个进程按照规定的顺序进行操作，避免竞态条件和资源冲突。

**1. 定义信号量和变量**

* **缓冲区 C1 相关信号量**：
  + C1\_empty：表示 **C1** 中剩余的空位数，初始值为 **8 - 3 = 5**。
  + C1\_full：表示 **C1** 中已放入的任务数，初始值为 **3**。
  + mutex\_C1：用于保护 **C1** 的互斥访问，初始值为 **1**。
* **缓冲区 C2 相关信号量**：
  + C2\_empty：表示 **C2** 中剩余的空位数，初始值为 **5**。
  + C2\_full：表示 **C2** 中已放入的任务数，初始值为 **0**。
  + mutex\_C2：用于保护 **C2** 的互斥访问，初始值为 **1**。

c

复制代码

// 信号量初始化

semaphore C1\_empty = 5; // C1中剩余空位数

semaphore C1\_full = 3; // C1中已放入的任务数

semaphore C2\_empty = 5; // C2中剩余空位数

semaphore C2\_full = 0; // C2中已放入的任务数

semaphore mutex\_C1 = 1; // C1的互斥锁

semaphore mutex\_C2 = 1; // C2的互斥锁

**2. 进程 Q1 的实现**

**Q1** 进程负责不断地将任务添加到 **C1** 中。它需要确保 **C1** 有空位，并且在添加任务时对 **C1** 进行互斥访问。

void Q1() {

while (1) {

// 生产任务并尝试放入 C1

wait(C1\_empty); // 等待 C1 中有空位

wait(mutex\_C1); // 进入 C1 的互斥区

// 向 C1 中添加一个任务

add\_task\_to\_C1();

printf("Q1: 添加一个任务到 C1，C1 中任务数: %d\n", C1\_full + 1);

signal(mutex\_C1); // 离开 C1 的互斥区

signal(C1\_full); // 增加 C1 中已放入的任务数

}

}

**3. 进程 Q2 的实现**

**Q2** 进程负责从 **C1** 中取出任务进行处理，并将处理后的任务送到 **C2** 中。它需要确保 **C1** 中有任务可取，同时 **C2** 有空位，并且在操作 **C1** 和 **C2** 时进行互斥访问。

void Q2() {

while (1) {

// 从 C1 中取出任务进行处理

wait(C1\_full); // 等待 C1 中有任务可取

wait(mutex\_C1); // 进入 C1 的互斥区

// 从 C1 中取出一个任务

remove\_task\_from\_C1();

printf("Q2: 从 C1 取出一个任务，C1 中任务数: %d\n", C1\_full - 1);

signal(mutex\_C1); // 离开 C1 的互斥区

signal(C1\_empty); // 增加 C1 中的空位数

// 处理任务

process\_task();

// 将处理后的任务放入 C2

wait(C2\_empty); // 等待 C2 中有空位

wait(mutex\_C2); // 进入 C2 的互斥区

add\_task\_to\_C2();

printf("Q2: 将处理后的任务放入 C2，C2 中任务数: %d\n", C2\_full + 1);

signal(mutex\_C2); // 离开 C2 的互斥区

signal(C2\_full); // 增加 C2 中已放入的任务数

}

}

**4. 进程 Q3 的实现**

**Q3** 进程负责从 **C2** 中取出任务执行。它需要确保 **C2** 中有任务可取，并且在取出任务时对 **C2** 进行互斥访问。

void Q3() {

while (1) {

// 从 C2 中取出任务执行

wait(C2\_full); // 等待 C2 中有任务可取

wait(mutex\_C2); // 进入 C2 的互斥区

// 从 C2 中取出一个任务

remove\_task\_from\_C2();

printf("Q3: 从 C2 取出一个任务，C2 中任务数: %d\n", C2\_full - 1);

signal(mutex\_C2); // 离开 C2 的互斥区

signal(C2\_empty); // 增加 C2 中的空位数

// 执行任务

execute\_task();

}

}

**5. 详细解释**

* **信号量 C1\_empty 和 C1\_full**：
  + C1\_empty 初始值为 **5**，表示 **C1** 中还有 **5** 个空位。
  + C1\_full 初始值为 **3**，表示 **C1** 中已放入 **3** 个任务。
  + **Q1** 在添加任务时，先进行 wait(C1\_empty) 确保 **C1** 有空位，然后通过 wait(mutex\_C1) 进入互斥区，安全地修改 **C1** 的状态，最后通过 signal(C1\_full) 增加 **C1** 中的任务数。
  + **Q2** 在从 **C1** 中取出任务时，先进行 wait(C1\_full) 确保 **C1** 中有任务可取，然后通过 wait(mutex\_C1) 进入互斥区，安全地修改 **C1** 的状态，最后通过 signal(C1\_empty) 增加 **C1** 中的空位数。
* **信号量 C2\_empty 和 C2\_full**：
  + C2\_empty 初始值为 **5**，表示 **C2** 中还有 **5** 个空位。
  + C2\_full 初始值为 **0**，表示 **C2** 中当前没有任务。
  + **Q2** 在将处理后的任务放入 **C2** 时，先进行 wait(C2\_empty) 确保 **C2** 有空位，然后通过 wait(mutex\_C2) 进入互斥区，安全地修改 **C2** 的状态，最后通过 signal(C2\_full) 增加 **C2** 中的任务数。
  + **Q3** 在从 **C2** 中取出任务时，先进行 wait(C2\_full) 确保 **C2** 中有任务可取，然后通过 wait(mutex\_C2) 进入互斥区，安全地修改 **C2** 的状态，最后通过 signal(C2\_empty) 增加 **C2** 中的空位数。
* **互斥信号量 mutex\_C1 和 mutex\_C2**：
  + 这两个信号量用于保护对 **C1** 和 **C2** 的互斥访问，防止多个进程同时修改缓冲区的状态，导致数据不一致。
* **进程之间的同步**：
  + **Q1** 和 **Q2** 通过 **C1** 的信号量进行同步，确保 **Q1** 不会在 **C1** 已满时继续添加任务，**Q2** 不会在 **C1** 为空时继续取出任务。
  + **Q2** 和 **Q3** 通过 **C2** 的信号量进行同步，确保 **Q2** 不会在 **C2** 已满时继续放置任务，**Q3** 不会在 **C2** 为空时继续取出任务。

**6. 总结流程**

1. **Q1** 进程不断地生产任务并尝试放入 **C1**：
   * 等待 **C1** 有空位 (wait(C1\_empty))。
   * 进入 **C1** 的互斥区 (wait(mutex\_C1))。
   * 添加任务到 **C1**。
   * 离开互斥区 (signal(mutex\_C1))。
   * 增加 **C1** 的任务数 (signal(C1\_full))。
2. **Q2** 进程不断地从 **C1** 中取出任务进行处理，并将处理后的任务放入 **C2**：
   * 等待 **C1** 中有任务可取 (wait(C1\_full))。
   * 进入 **C1** 的互斥区 (wait(mutex\_C1))。
   * 从 **C1** 中取出任务。
   * 离开互斥区 (signal(mutex\_C1))。
   * 增加 **C1** 的空位数 (signal(C1\_empty))。
   * 处理任务。
   * 等待 **C2** 有空位 (wait(C2\_empty))。
   * 进入 **C2** 的互斥区 (wait(mutex\_C2))。
   * 将处理后的任务放入 **C2**。
   * 离开互斥区 (signal(mutex\_C2))。
   * 增加 **C2** 的任务数 (signal(C2\_full))。
3. **Q3** 进程不断地从 **C2** 中取出任务执行：
   * 等待 **C2** 中有任务可取 (wait(C2\_full))。
   * 进入 **C2** 的互斥区 (wait(mutex\_C2))。
   * 从 **C2** 中取出任务。
   * 离开互斥区 (signal(mutex\_C2))。
   * 增加 **C2** 的空位数 (signal(C2\_empty))。
   * 执行任务。

通过上述信号量的合理使用和进程之间的同步与互斥控制，确保了 **Q1**、**Q2**、**Q3** 三个进程能够有序地对缓冲区 **C1** 和 **C2** 进行操作，避免了资源冲突和竞态条件，提高了系统的整体效率和可靠性。

**代码实现**

以下是使用类 C 语言描述的 **Q1**、**Q2**、**Q3** 进程的同步与互斥算法：

#include <stdio.h>

#include <stdlib.h>

#include <pthread.h>

#include <semaphore.h>

// 定义信号量

sem\_t C1\_empty;

sem\_t C1\_full;

sem\_t C2\_empty;

sem\_t C2\_full;

sem\_t mutex\_C1;

sem\_t mutex\_C2;

// 初始化缓冲区任务数

int C1\_tasks = 3;

int C2\_tasks = 0;

// 模拟添加任务到 C1

void add\_task\_to\_C1() {

C1\_tasks++;

}

// 模拟从 C1 中移除任务

void remove\_task\_from\_C1() {

C1\_tasks--;

}

// 模拟添加任务到 C2

void add\_task\_to\_C2() {

C2\_tasks++;

}

// 模拟从 C2 中移除任务

void remove\_task\_from\_C2() {

C2\_tasks--;

}

// 模拟任务处理

void process\_task() {

printf("Q2: 正在处理任务...\n");

// 模拟处理时间

sleep(1);

}

// 模拟任务执行

void execute\_task() {

printf("Q3: 正在执行任务...\n");

// 模拟执行时间

sleep(1);

}

// Q1 进程函数

void\* Q1\_process(void\* arg) {

while (1) {

// 生产任务并尝试放入 C1

sem\_wait(&C1\_empty); // 等待 C1 中有空位

sem\_wait(&mutex\_C1); // 进入 C1 的互斥区

// 向 C1 中添加一个任务

add\_task\_to\_C1();

printf("Q1: 添加一个任务到 C1，C1 中任务数: %d\n", C1\_tasks);

sem\_post(&mutex\_C1); // 离开 C1 的互斥区

sem\_post(&C1\_full); // 增加 C1 中已放入的任务数

// 模拟生产时间

sleep(1);

}

return NULL;

}

// Q2 进程函数

void\* Q2\_process(void\* arg) {

while (1) {

// 从 C1 中取出任务进行处理

sem\_wait(&C1\_full); // 等待 C1 中有任务可取

sem\_wait(&mutex\_C1); // 进入 C1 的互斥区

// 从 C1 中取出一个任务

remove\_task\_from\_C1();

printf("Q2: 从 C1 取出一个任务，C1 中任务数: %d\n", C1\_tasks);

sem\_post(&mutex\_C1); // 离开 C1 的互斥区

sem\_post(&C1\_empty); // 增加 C1 中的空位数

// 处理任务

process\_task();

// 将处理后的任务放入 C2

sem\_wait(&C2\_empty); // 等待 C2 中有空位

sem\_wait(&mutex\_C2); // 进入 C2 的互斥区

add\_task\_to\_C2();

printf("Q2: 将处理后的任务放入 C2，C2 中任务数: %d\n", C2\_tasks);

sem\_post(&mutex\_C2); // 离开 C2 的互斥区

sem\_post(&C2\_full); // 增加 C2 中已放入的任务数

}

return NULL;

}

// Q3 进程函数

void\* Q3\_process(void\* arg) {

while (1) {

// 从 C2 中取出任务执行

sem\_wait(&C2\_full); // 等待 C2 中有任务可取

sem\_wait(&mutex\_C2); // 进入 C2 的互斥区

// 从 C2 中取出一个任务

remove\_task\_from\_C2();

printf("Q3: 从 C2 取出一个任务，C2 中任务数: %d\n", C2\_tasks);

sem\_post(&mutex\_C2); // 离开 C2 的互斥区

sem\_post(&C2\_empty); // 增加 C2 中的空位数

// 执行任务

execute\_task();

}

return NULL;

}

int main() {

// 初始化信号量

sem\_init(&C1\_empty, 0, 5); // C1 中剩余空位数

sem\_init(&C1\_full, 0, 3); // C1 中已放入的任务数

sem\_init(&C2\_empty, 0, 5); // C2 中剩余空位数

sem\_init(&C2\_full, 0, 0); // C2 中已放入的任务数

sem\_init(&mutex\_C1, 0, 1); // C1 的互斥锁

sem\_init(&mutex\_C2, 0, 1); // C2 的互斥锁

// 创建进程线程

pthread\_t Q1, Q2, Q3;

pthread\_create(&Q1, NULL, Q1\_process, NULL);

pthread\_create(&Q2, NULL, Q2\_process, NULL);

pthread\_create(&Q3, NULL, Q3\_process, NULL);

// 等待线程结束（实际上不会结束）

pthread\_join(Q1, NULL);

pthread\_join(Q2, NULL);

pthread\_join(Q3, NULL);

// 销毁信号量（代码不会执行到这里）

sem\_destroy(&C1\_empty);

sem\_destroy(&C1\_full);

sem\_destroy(&C2\_empty);

sem\_destroy(&C2\_full);

sem\_destroy(&mutex\_C1);

sem\_destroy(&mutex\_C2);

return 0;

}

**7. 代码说明**

* **信号量初始化**：
  + C1\_empty 初始化为 **5**，表示 **C1** 中有 **5** 个空位。
  + C1\_full 初始化为 **3**，表示 **C1** 中有 **3** 个任务。
  + C2\_empty 初始化为 **5**，表示 **C2** 中有 **5** 个空位。
  + C2\_full 初始化为 **0**，表示 **C2** 中没有任务。
  + mutex\_C1 和 mutex\_C2 初始化为 **1**，用于互斥访问 **C1** 和 **C2**。
* **进程 Q1**：
  + 不断地生产任务并尝试放入 **C1**。
  + 使用 sem\_wait(&C1\_empty) 确保 **C1** 有空位。
  + 使用 sem\_wait(&mutex\_C1) 进入 **C1** 的互斥区，安全地添加任务。
  + 使用 sem\_post(&C1\_full) 增加 **C1** 中的任务数。
* **进程 Q2**：
  + 不断地从 **C1** 中取出任务进行处理，并将处理后的任务放入 **C2**。
  + 使用 sem\_wait(&C1\_full) 确保 **C1** 中有任务可取。
  + 使用 sem\_wait(&mutex\_C1) 进入 **C1** 的互斥区，安全地取出任务。
  + 使用 sem\_post(&C1\_empty) 增加 **C1** 中的空位数。
  + 处理任务后，使用 sem\_wait(&C2\_empty) 确保 **C2** 有空位。
  + 使用 sem\_wait(&mutex\_C2) 进入 **C2** 的互斥区，安全地放入处理后的任务。
  + 使用 sem\_post(&C2\_full) 增加 **C2** 中的任务数。
* **进程 Q3**：
  + 不断地从 **C2** 中取出任务执行。
  + 使用 sem\_wait(&C2\_full) 确保 **C2** 中有任务可取。
  + 使用 sem\_wait(&mutex\_C2) 进入 **C2** 的互斥区，安全地取出任务。
  + 使用 sem\_post(&C2\_empty) 增加 **C2** 中的空位数。
  + 执行任务。

通过上述算法，**Q1**、**Q2**、**Q3** 三个进程能够协同工作，安全地访问共享缓冲区 **C1** 和 **C2**，实现任务的生产、处理和执行，避免了资源冲突和竞态条件，确保了系统的稳定和高效运行。