操作系统原理第四章作业

姓名：马福泉　学号：23336179　截止日期：2025 年 4 月 9 日

完成日期：2025 年 4 月 7 日

Question 1:第一个著名的正确解决两个进程的临界区问题的软件方法是由 Dekker 设计的。两个进程 P1 和 P2 共享以下变量:

```
boolean flag[2];/*initially false*/
int turn;
```

进程 P_i（i==0 或 1）的结构见图 6.25，另一个进程为 P_j（j==0 或 1）。试证明这个算法满足临界区问题的所有三个要求。

```
do {
    flag[i] = TRUE;
    while (flag[j]) {
        if (turn == j) {
            flag[i] = false;
            while (turn == j)
                ; // do nothing
            flag[i] = TRUE;
        }
    }
        // critical section
    turn = j;
    flag[i] = FALSE;
        // remainder section
}while (TRUE);
```

图 6.25　Dekker 算法中的进程 P_i 结构

Answer 1：

```c
boolean flag[2]; /* initially false */
int turn;

do {
    flag[i] = TRUE; // 设置当前进程的标志为真，表示想要进入临界区
    while (flag[j]) { // 检查另一个进程是否也想进入临界区
        if (turn == j) { // 如果轮到另一个进程进入临界区
            flag[i] = false; // 取消当前进程的请求
            while (turn == j) // 等待直到轮到当前进程
                ; // do nothing
            flag[i] = TRUE; // 重新设置当前进程的标志为真
        }
    }
    // critical section临界区
    turn = j; // 设置轮到下一个进程进入临界区
    flag[i] = FALSE; // 离开临界区后，设置当前进程的标志为假
    // remainder section剩余区
} while (TRUE);
```

代码解读如上图：

（1） 互斥：根据上述代码解读，有两种情况：

① 两个进程的 flag 同时为 TRUE：根据 turn 变量，只有一个进程会在 if 条件中失败并把自己的 flag 设为 FALSE，另一个可以继续。

② 一个进程的 flag 为 TRUE 时另一个为 FALSE：只有 flag 为 TRUE 的进程可以进入临界区。

因此，不可能两个进程同时进入临界区，满足互斥。

（2） 进程推进：如果只有一个进程 Pi 想进入（flag[i]=TRUE，flag[j]=FALSE），它可以立即进入。如果两个进程都想进入，turn 变量决定谁可以进入，另一个进程会暂时放弃（设置 flag=FALSE），确保至少一个进程可以进入。第一个进程完成后会改变 turn 变量，从而允许等待的进程进入。算法避免了两个进程互相阻塞的情况，满足进程推进。

（3） 有限等待：当一个进程 Pi 退出临界区时，它会设置 turn = j，这确保 Pj 有下一次进入的机会。因此，每个进程最多等待另一个进程进入一次临界区后就能获得机会。等待时间有限。

Question 2：6.3 术语 忙等 的含义是什么?操作系统中其他类型的等待有哪些?忙等能否完全避免?为什么?

Answer 2：

（1） 忙等是指进程/线程在等待某个条件满足时，持续占用 CPU 资源循环检查条件状态，而非释放 CPU，导致进入阻塞状态。典型例子是自旋锁。

（2） 其他类型的等待：

阻塞等待：进程主动让出 CPU 进入休眠状态（如通过 wait()系统调用）

被唤醒需依赖中断/信号机制（例如 I/O 完成中断）。

定时等待：设置超时时间的阻塞，超时后自动唤醒，避免永久阻塞

事件驱动等待：通过回调/事件通知机制（如 epoll、IOCP），进程注册事件后立即释放 CPU，事件触发时由内核通知。

（3）不能完全避免，原因如下：

① 实时性要求：在某些对实时性要求极高的系统（如嵌入式系统或实时操作系统）中，忙等可以保证极低的延迟。例如，当需要快速响应某个事件时，忙等可以避免进程切换的开销。

② 多处理器系统的优势：在多处理器系统中，忙等可以避免进程切换的开销，因为其他处理器可以继续执行其他任务。

③ 系统设计的复杂性：完全避免忙等需要复杂的同步机制和调度算法，可能会增加系统的复杂性和开销。

Question 3：6.4 试解释为什么自旋锁对单处理器系统不合适而对多处理器系统合适。

Answer 3：

（1） 单处理器系统不适用

① 浪费 CPU 资源：在单处理器系统中，如果线程 A 持有自旋锁，线程 B 尝试获取锁时会持续自旋（占用 CPU 循环检查）。但由于只有一个 CPU，线程 A 无法与线程 B 并发执行，因此线程 A 无法释放锁。这会导致线程 B 无限空转，而线程 A 因无法调度而无法执行，形成死锁或资源浪费。

② 降低系统性能：即使系统支持抢占式调度，自旋的线程仍会持续占用 CPU 时间片，导致其他就绪线程（包括持有锁的线程）无法被调度，降低系统效率。

③ 上下文切换无法优化：在单处理器系统中，上下文切换是不可避免的。即使使用自旋锁，当线程自旋等待锁时，操作系统仍然需要在一定时间后将其挂起，切换到其他线程运行。这种情况下，自旋锁无法像在多处理器系统中那样减少上下文切换的开销。

（2） 多处理器系统适用

① 并发执行的可能性：在多处理器系统中，线程 A 可以在一个 CPU 上持有锁并执行临界区代码，而线程 B 在另一个 CPU 上自旋等待。此时线程 A 和线程 B 是真正并发的，线程 A 可以很快释放锁，线程 B 的自旋时间通常较短。

② 避免上下文切换的开销：如果临界区代码非常短，如几条指令，自旋锁的效率高于睡眠/唤醒机制。因为线程休眠和唤醒会触发上下文切换，而上下文切换的开销可能远大于短时间的自旋。

③ 适合短临界区：多核环境下，自旋锁适用于锁持有时间极短的场景（如修改一个标志位）。长时间的自旋仍会浪费 CPU 资源，但通过优化（如自适应自旋锁）可以缓解这一问题。

Question 5：6.5 试解释为什么在单处理器系统上通过禁止中断实现同步原语方法不适用于用户级程序。

Answer 5：

(1) 用户程序无权禁止中断。特权指令限制：禁止中断是特权操作，只有内核可以执行。用户程序尝试执行此类指令会触发异常。
(2) 安全性问题：如果允许用户程序随意禁止中断，恶意程序可能通过长期关闭中断导致系统无法响应外部事件，从而破坏系统的稳定性和实时性。
(3) 多线程场景下的失效：即使某个线程通过某种方式禁止了中断，其他线程仍可能被调度器切换执行（例如时间片到期）。中断禁止仅对当前 CPU 核心有效，而用户程序无法控制调度行为，导致同步失效。内核可通过禁止中断+自旋保证原子性（因为调度器依赖中断），但用户程序无法阻止调度器切换线程。
(4) 性能长时间禁止中断的代价：内核中禁止中断的时间必须极短（如修改就绪队列）。用户程序若模拟这种行为（如通过系统调用），频繁的内核切换会带来巨大开销。
(5) 不适用于多处理器：即使在单处理器上“可行”，这种方案也无法扩展到多核系统，因为中断禁止仅对当前核心有效，其他核心仍可能并发访问共享数据。

Question 6：6.6 试解释为什么通过禁止中断实现同步原语不适合于多处理器系统。

Answer 6：

（1） 单处理器系统中，通过禁止中断来实现同步原语（如临界区的保护），中断是导致线程/进程切换的唯一途径。禁止中断可以确保当前执行的线程不会被抢占，从而独占 CPU 资源，避免竞态条件。

（2）不适合多处理器系统，原因如下：

① 局部性问题：中断屏蔽只能影响当前处理器，无法阻止其他处理器访问共享资源。

② 无法全局互斥：多个处理器可以同时进入临界区，导致数据竞争和冲突。

③ 性能瓶颈：禁止中断会阻塞其他处理器的正常运行，降低系统效率

Question 7：6.7 描述如何用 swap()指令来实现互斥并满足有限等待要求。

Answer 7：

（1） 初始化：定义一个共享的锁变量（lock），初始值为“未占用”。如果使用条件变量，还需要初始化相关的同步机制，如互斥锁和条件变量。

（2） 线程尝试获取锁：线程通过原子操作尝试将锁变量从“未占用”改为“已占用”。如果成功，线程进入临界区。如果失败，线程进入等待状态。

（3） 线程释放锁：线程完成临界区的操作后，通过原子操作将锁变量从“已占用”改为“未占用”。如果使用了条件变量，唤醒等待的线程。
（4） 线程重新尝试获取锁：被唤醒的线程再次尝试通过原子操作获取锁。如果锁仍然被占用，线程继续等待；否则，线程进入临界区。

Question 8：6.8 服务器可设计成限制打开的连接数。例如,某个服务器可能需要在某个时刻只能打开 N 个 Socket6.8 连接。一旦有 N 个连接，那么服务器就不再接受新的连接请求，直到有现有连接释放为止。解释如何采用信号量来限制并发连接的数量。

Answer 8：
（1） 定义一个信号量 sem，初始化值为 N，表示服务器允许的最大并发连接数。
（2） 当服务器收到一个新连接请求时：执行 P(sem)，即 sem.wait() 。如果信号量值大于 0，则减少信号量值，允许该连接进入；如果信号量值为 0，则阻塞该连接请求，直到有其他连接释放。
（3） 当一个连接完成服务并关闭时：执行 V(sem)，即 sem.signal，增加信号量值，允许新的连接进入。
代码示例：

```
// 初始化信号量
semaphore sem = N;  // N为最大并发连接数

// 服务器处理新连接的逻辑
void handle_new_connection() {
    P(sem);  // 尝试获取信号量
    if (sem > 0) {
        // 建立连接，处理请求
        accept_connection();
        process_request();
    } else {
        // 阻塞等待，直到有连接释放
        wait_for_connection_release();
    }
}

// 服务器处理连接关闭的逻辑
void handle_connection_close() {
    // 释放信号量，允许新的连接进入
    V(sem);
}
```

Question 9：6.9 试说明如果 wait()和 signal()操作不是原子化操作，那么互斥可能是不稳定的。

Answer 9：

（1）wait() 非原子化导致的问题：假设 wait() 操作不是原子的，那么在执行过程中可能会被中断。如果此时其他进程刚好执行了 signal()，那么这个 signal() 可能会丢失。因为当被打断的进程恢复执行时，它会继续执行 wait()，但由于之前的 signal() 已经完成，它会误以为条件仍然不满足，从而继续等待。

结果是，进程可能会永远阻塞。

（2）signal() 非原子化导致的问题：如果 signal() 不是原子操作，那么在执行过程中也可能被中断。如果此时其他进程刚好调用了 wait()，那么这个 wait() 可能会错过 signal()。因为当被打断的进程恢复执行时，它会继续执行 signal()，但由于等待的进程已经完成了 wait()，这个 signal() 就会无效。

结果是，等待的进程可能一直等待下去，因为它没有被正确地唤醒。

（4） 互斥不稳定的具体表现

① 死锁：如果 wait() 和 signal() 不是原子操作，可能会导致进程或线程无法正确地进入或退出临界区，从而导致死锁。

② 竞态条件：由于 wait() 和 signal() 的非原子性，可能会导致多个进程或线程对共享资源的操作顺序混乱，从而引发竞态条件。例如，两个进程都试图通过 wait() 和 signal() 来同步对某个共享变量的访问，但由于操作不是原子的，它们可能会同时进入临界区，导致数据不一致。

③ 资源泄露：如果 signal() 操作被中断，可能会导致信号丢失，使得等待的进程无法被唤醒，从而占用系统资源（如进程控制块、内存等）而无法释放，最终导致资源泄露。