进程同步 (Process Synchronization) 之临界区问题 (Critical Section Problem)

让我们先达成共识

- 对共享数据 (Shared Data) 的并发访问 (Concurrent Access)，可能导致数据不一致问题
- 确保数据一致性 (Data Consistency)，是个合理的要求。它需要一个机制，以保证合作进程们有序地执行
- 以生产者 - 消费者问题为例。设计一个整型变量 count，总是记录缓冲区中被占用的单元总数。count 的初始值为 0；当生产者进程注入一个单元数据时，count 增 1；当消费者进程消费掉一个单元数据时，count 减 1。

生产者 - 消费者问题的一种解法

- 共享变量 (Shared data) 描述有界缓冲区 (Bounded Buffer)

```
#define BUFFER_SIZE 10
typedef struct {
   . . .
} item;
item buffer[BUFFER_SIZE];
int count = 0;
```

  - 进程独占变量

```
int in = 0;
int out = 0;
```

生产者 - 消费者问题的一种解法（续）

- 生产者进程

```
item nextProduced;

while (1) {
        while (count == BUFFER_SIZE)
                ; /* do nothing */
        buffer[in] = nextProduced;
        in = (in + 1) % BUFFER_SIZE;
        count++;
}
```

生产者 - 消费者问题的一种解法（续）

- 消费者进程

```
item nextConsumed;

while (1) {
        while (count == 0)
                ; /* do nothing */
        nextConsumed = buffer[out];
        out = (out + 1) % BUFFER_SIZE;
        count--;
}
```

生产者 - 消费者问题的一种解法（续）

- 其中的 2 条语句

  count++;
  count--;

  必须独立不受干扰地执行

- 原子操作 (Atomic operation) 要求该操作完整地一次性完成，不允许中间被打断
- （e. g. 中断响应， CPU 重新调度等）

生产者 - 消费者问题的一种解法（续）

- 高级语言的语句“ **count++**”可能被编译翻译成如下机器语言：

  register1 = count

  register1 = register1 + 1
  count = register1

- 高级语言的语句“ **count --**”可能被编译翻译成如下机器语言：

  register2 = count
  register2 = register2 – 1
  count = register2

生产者 - 消费者问题的一种解法（续）

- 假如并发执行的生产者进程和消费者进程恰巧经历一个时机点：它们都意欲修改共享变量 count ！上述汇编语句可能交叉执行。
- 是否交叉执行，取决于 CPU 调度器的调度结果。

生产者 - 消费者问题的一种解法（续）

- 假设 **count** 的当前值为 5，一种交叉执行的场景:
  producer: **register1 = count** (*register1 = 5*)
  producer: **register1 = register1 + 1** (*register1 = 6*)
  consumer: **register2 = count** (*register2 = 5*)
  consumer: **register2 = register2 – 1** (*register2 = 4*)
  producer: **count = register1** (*count = 6*)
  consumer: **count = register2** (*count = 4*)
- 生产者进程和消费者进程在一次并发执行后，共享变量 **count** 变为 4 or 6
- 正确的结果应该是 5 ！

Race Condition

- **Race condition(竞争)**: The situation where several processes access and manipulate shared data concurrently. The final value of the shared data depends upon which process finishes last.
- 若干进程并发访问并且操纵共享数据的特殊情形。共享数据的最终稳定值取决于最后完成操纵的那个进程

Race Condition

- 为了避免 race conditions， 并发进程必须同步 (**synchronized)**

临界区问题 (The Critical-Section Problem)

- 假设 *n* 个进程竞相访问共享数据的情形
- 每个进程存在一段代码，称作为*临界区 (critical section)*，进程就是通过这段代码访问了共享数据 (shared data)
- 其它代码段没有访问共享数据
- 这 n 个进程中，至少存在 1 个以上的进程甚至修改了共享数据

临界区问题（续）

- 有很多例子，关于临界区
- 临界区问题 – 怎样确保，当有一个进程 i 正在其自己的临界区执行时，没有任何其它进程 j 也在它（进程 j）的临界区中执行

临界区问题的解决方案必须满足 3 条件

- “互斥” (Mutual Exclusion) – 如果进程 P_i 正在其临界区执行，那么，其它任何进程均不允许在他们的临界区中

临界区问题的解决方案必须满足 3 条件

- “空闲让进” (Progress) – 如果
- 没有进程处于它的临界区， and
- 某些进程申请进入其临界区

那么

- 只有那些不在 remainder sections 的进程，才能参与能否进入临界区的选举， and
- 这个选举不允许无限期 (indefinitely) 推迟

临界区问题的解决方案必须满足 3 条件

- “有限等待” (Bounded Waiting) - 某一进程从其提出请求，至它获准进入临界区的这段时间里，其它进程进入他们的临界区的次数存在上界
  - 假设进程各自都在持续执行
  - 不考虑 N 个进程之间的相对执行速度

临界区问题解决方案的简化框架

- 只限于 2 processes, P_0 and P_1
- 进程 P_i 代码段的一般化结构 (进程 P_j 也一样)

do {

进入临界区前 (*entry section*)

临界区 (critical section)

离开临界区后 (*exit section*)

其它代码段 (remainder section)

} **while (1)**;

- 当然，两个进程可能有多处共享变量，因此有多处这样的一般化结构。


End