实验二 生产者消费者问题

班级: 07112303 学号: 1120231863 姓名: 左逸龙

一、 实验目的

本次实验的核心目的在于深入理解并掌握操作系统中进程同步、互斥及进程间通信（IPC）的核心原理。通过在 Windows 平台上编程实现一个经典的多进程同步模型——生产者-消费者问题，旨在达成以下具体目标：

1. 深刻理解进程同步与互斥的必要性。分析在不加控制的情况下，多个生产者进程和消费者进程并发访问有限缓冲池可能引发的竞态条件与数据不一致性问题。
2. 掌握利用信号量（`Semaphore`）作为同步工具解决并发问题的方法。具体而言，学习如何运用三个核心信号量：一个用于计数的信号量 `empty` 来追踪空闲缓冲区数量，一个用于计数的信号量 `full` 来追踪已用缓冲区数量，以及一个二值信号量 `mutex` 来确保任何时刻仅有一个进程能访问缓冲池，从而实现进程间的互斥访问。
3. 学习一种重要的进程间通信机制——共享内存（`Shared Memory`）。通过创建和映射共享内存区域，使多个独立的进程能够共同访问和操作同一数据结构（即缓冲池），实现数据交换。
4. 熟悉在 Windows 操作系统环境下进行多进程并发编程的基本流程。掌握 `CreateProcess`、`CreateSemaphore`、`WaitForSingleObject`、`ReleaseSemaphore`、`CreateFileMapping` 及 `MapViewOfFile` 等关键 Windows API 函数的使用，将操作系统理论知识与具体的编程实践相结合。

二、 实验内容

本实验要求在 Windows 平台上编程实现一个多进程的生产者-消费者模型，以解决进程间的同步与互斥问题。实验需严格遵循以下具体设定与要求：

- **共享缓冲池 (Shared Buffer Pool)**
  - 创建一个包含 **6** 个缓冲区的共享缓冲池。
  - 每个缓冲区能够存放一个最大长度为 **10** 个字符的字符串。
- **生产者进程 (Producer Processes)**
  - 创建 **2** 个独立的生产者进程。
  - 每个生产者进程循环执行 **12** 次生产任务。在每次任务中，进程将随机等待一段时间，然后生成一个数据项并尝试将其放入缓冲池。
  - 当缓冲池已满时，生产者进程必须进入阻塞状态，直到有消费者进程取走数据释放出空闲缓冲区后方可继续执行。

- **消费者进程 (Consumer Processes)**
  - 创建 **3** 个独立的消费者进程。
  - 每个消费者进程循环执行 **8** 次消费任务。在每次任务中，进程将随机等待一段时间，然后尝试从缓冲池中取出一个数据项。
  - 当缓冲池为空时，消费者进程必须进入阻塞状态，直到有生产者进程放入新的数据后方可继续执行。
- **输出要求**
  - 每当一个生产者成功放入数据或一个消费者成功取出数据后，程序必须立即在控制台输出该操作发生的**精确时间**，并同时显示此刻整个**缓冲池的映像**，以直观地展示并发操作下共享资源的变化情况。

三、 实验步骤

本实验的核心技术在于借助 Windows API 提供的信号量（`Semaphore`）实现进程同步与互斥，并利用共享内存（`Shared Memory`）完成进程间通信。完整代码请参考附件 `producer_consumer.cpp`。

3.1 全局定义与共享数据结构

为保证程序的可读性与可维护性，首先定义了模型中的各项常量，包括缓冲池大小、生产者与消费者数量等。同时，定义了用于在共享内存中存储的核心数据结构 `SharedBufferPool`，该结构包含了缓冲池实体以及用于读写位置的循环队列指针。

```
// --- 全局常量定义 ---
const int BUFFER_ITEM_LENGTH = 10;
const int BUFFER_COUNT = 6;
const int PRODUCER_COUNT = 2;
const int CONSUMER_COUNT = 3;
const int PRODUCE_LOOPS = 12;
const int CONSUME_LOOPS = 8;
const int MAX_RANDOM_WAIT_MS = 1000;

// --- 共享内存数据结构 ---
struct SharedBufferPool {
    char buffer[BUFFER_COUNT][BUFFER_ITEM_LENGTH + 1]; // +1 是因为需要存储 `\0`
    int write_pos; // 生产者写入位置
    int read_pos; // 消费者读取位置
};

// --- 内核对象命名 ---
const char* SHARED_MEM_NAME = "MySharedMemoryPool";
const char* EMPTY_SEM_NAME = "Semaphore_EmptySlots";
const char* FULL_SEM_NAME = "Semaphore_FullSlots";
const char* MUTEX_SEM_NAME = "Semaphore_MutexLock";
```

3.2 主控进程的初始化

主控进程（`main` 函数在 `argc <= 1` 时的逻辑分支）负责整个运行环境的搭建。它首先使用 `CreateSemaphore` API 创建三个核心的命名信号量：

- **empty**：计数信号量，初始值为缓冲池大小 `BUFFER_COUNT`，用于记录空闲缓冲区的数量。
- **full**：计数信号量，初始值为 0，用于记录已被生产者填充的缓冲区的数量。
- **mutex**：二值信号量，初始值为 1，作为互斥锁，确保同一时间只有一个进程能修改缓冲池。

随后，通过 `CreateFileMapping` 和 `MapViewOfFile` 创建并映射一块共享内存，用于存放 `SharedBufferPool` 结构体实例，并对其进行初始化。

```
// 同步对象1：记录空闲缓冲区数量的信号量
HANDLE hEmpty = CreateSemaphore(NULL, BUFFER_COUNT, BUFFER_COUNT, EMPTY_SEM_NAME);
// 同步对象2：记录已用缓冲区数量的信号量
HANDLE hFull = CreateSemaphore(NULL, 0, BUFFER_COUNT, FULL_SEM_NAME);
// 同步对象3：用于保证缓冲区被互斥访问的二值信号量
HANDLE hMutex = CreateSemaphore(NULL, 1, 1, MUTEX_SEM_NAME);

// 创建用于进程间通信的共享内存区域
HANDLE hMapFile = CreateFileMapping(
    INVALID_HANDLE_VALUE, NULL, PAGE_READWRITE, 0,
    sizeof(SharedBufferPool), SHARED_MEM_NAME);

// ... 错误检查 ...

SharedBufferPool* sharedData = (SharedBufferPool*)MapViewOfFile(hMapFile,
FILE_MAP_ALL_ACCESS, 0, 0, sizeof(SharedBufferPool));

// 初始化共享内存数据
sharedData->write_pos = 0;
sharedData->read_pos = 0;
for (int i = 0; i < BUFFER_COUNT; ++i) {
    sharedData->buffer[i] = '\0';
}
```

3.3 生产者进程实现

生产者进程的逻辑在 `ProducerProcess` 函数中实现。每个生产者进程首先通过 `OpenSemaphore` 和 `OpenFileMapping` 打开主进程已创建的命名内核对象。其核心操作在一个循环中执行，该循环的每一次迭代都严格遵循 “P-V” 操作的同步范式：

1. 执行 `WaitForSingleObject(hEmpty, INFINITE)`，即对 `empty` 信号量执行 **P** 操作。若 `empty` 计数大于 0，则计数减 1 并继续；否则，进程阻塞，等待消费者释放缓冲区。
2. 执行 `WaitForSingleObject(hMutex, INFINITE)`，即对 `mutex` 信号量执行 **P** 操作，以获取对缓冲池的独占访问权。
3. 进入**临界区**：在当前写入位置 `write_pos` 放入新数据，并更新 `write_pos` 指针。

4. 执行 `ReleaseSemaphore(hMutex, 1, NULL)`，即对 `mutex` 信号量执行 **V** 操作，释放互斥锁。
5. 执行 `ReleaseSemaphore(hFull, 1, NULL)`，即对 `full` 信号量执行 **V** 操作，通知消费者已有新数据可用。

```
// --- 生产者进程主函数 ---
void ProducerProcess(int id) {
    // ... 打开内核对象句柄并映射共享内存 ...

    for (int i = 0; i < PRODUCE_LOOPS; ++i) {
        Sleep(rand() % MAX_RANDOM_WAIT_MS);

        // P操作：等待空闲缓冲区
        WaitForSingleObject(hEmpty, INFINITE);
        // P操作：请求互斥访问
        WaitForSingleObject(hMutex, INFINITE);

        // --- 临界区 ---
        int current_pos = sharedData->write_pos;
        std::string item = "P" + std::to_string(id) + "_Item" + std::to_string(i + 1);
        strcpy_s(sharedData->buffer[current_pos], BUFFER_ITEM_LENGTH + 1,
item.c_str());
        sharedData->write_pos = (sharedData->write_pos + 1) % BUFFER_COUNT;

        std::string msg = "生产者 P" + std::to_string(id) + " 在缓冲区 " +
std::to_string(current_pos) + " 中放入数据 \"" + item + "\"";
        PrintBufferSnapshot(msg, sharedData);
        // --- 临界区结束 ---

        // V操作：释放互斥锁
        ReleaseSemaphore(hMutex, 1, NULL);
        // V操作：通知有产品可用
        ReleaseSemaphore(hFull, 1, NULL);
    }
    // ... 清理资源 ...
}
```

3.4 消费者进程实现

消费者进程的逻辑在 `ConsumerProcess` 函数中实现，其结构与生产者对称。消费者进程同样首先获取所有必需的内核对象句柄。其核心循环遵循的同步逻辑如下：

1. 执行 `WaitForSingleObject(hFull, INFINITE)`，即对 `full` 信号量执行 **P** 操作。若 `full` 计数大于 0，则计数减 1 并继续；否则，进程阻塞，等待生产者放入数据。
2. 执行 `WaitForSingleObject(hMutex, INFINITE)`，获取对缓冲池的互斥访问权。
3. 进入**临界区**：从当前读取位置 `read_pos` 取出数据，并更新 `read_pos` 指针。
4. 执行 `ReleaseSemaphore(hMutex, 1, NULL)`，释放互斥锁。
5. 执行 `ReleaseSemaphore(hEmpty, 1, NULL)`，即对 `empty` 信号量执行 **V** 操作，通知生产者已有空闲缓冲区可用。

```
// --- 消费者进程主函数 ---
void ConsumerProcess(int id) {
    // ... 打开内核对象句柄并映射共享内存 ...

    for (int i = 0; i < CONSUME_LOOPS; ++i) {
        Sleep(rand() % MAX_RANDOM_WAIT_MS);

        // P操作：等待产品
        WaitForSingleObject(hFull, INFINITE);
        // P操作：请求互斥访问
        WaitForSingleObject(hMutex, INFINITE);

        // --- 临界区 ---
        int current_pos = sharedData->read_pos;
        std::string item(sharedData->buffer[current_pos]);
        sharedData->buffer[current_pos] = '\0'; // 模拟取出数据
        sharedData->read_pos = (sharedData->read_pos + 1) % BUFFER_COUNT;

        std::string msg = "消费者 C" + std::to_string(id) + " 从缓冲区 " +
std::to_string(current_pos) + " 中取出数据 \"" + item + "\"";
        PrintBufferSnapshot(msg, sharedData);
        // --- 临界区结束 ---

        // V操作：释放互斥锁
        ReleaseSemaphore(hMutex, 1, NULL);
        // V操作：通知有空缓冲区可用
        ReleaseSemaphore(hEmpty, 1, NULL);
    }
    // ... 清理资源 ...
}
```

3.5 进程创建与管理

完成环境初始化后，主控进程使用 `CreateProcess` API，根据设定的 `PRODUCER_COUNT` 和 `CONSUMER_COUNT` 循环创建所需数量的子进程。创建时，通过命令行向子进程传递其角色（“producer” 或 “consumer”）和唯一 ID。所有子进程的句柄被收集起来，最后通过 `WaitForMultipleObjects` 函数等待所有子进程执行完毕，确保主进程在所有任务完成后才退出并清理资源。

```
// ...
char selfPath[MAX_PATH];
GetModuleFileName(NULL, selfPath, MAX_PATH);

// 创建生产者进程
for (int i = 0; i < PRODUCER_COUNT; ++i) {
    std::string cmd = std::string(selfPath) + " producer " + std::to_string(i + 1);
    // ... 调用 CreateProcess ...
}

// 创建消费者进程
for (int i = 0; i < CONSUMER_COUNT; ++i) {
    std::string cmd = std::string(selfPath) + " consumer " + std::to_string(i + 1);
    // ... 调用 CreateProcess ...
}

std::cout << "已创建 " << PRODUCER_COUNT << " 个生产者与 " << CONSUMER_COUNT << " 个消费
者进程。" << std::endl;
```

```
18
19 // 等待所有子进程执行完毕
20 WaitForMultipleObjects(static_cast<DWORD>(processHandles.size()),
processHandles.data(), TRUE, INFINITE);
21
22 std::cout << "所有任务已完成。" << std::endl;
23
24 // 清理句柄
25 // ...
```

四、 实验结果及分析

本章将展示程序的运行结果，并对输出日志进行详细分析，以验证实验设计的正确性，并揭示多进程并发执行的内在规律。

4.1 运行结果

将程序编译并执行后，主控进程成功创建了 2 个生产者和 3 个消费者子进程。各进程并发执行，交替地对共享缓冲池进行读写操作。由于进程调度的随机性，每次运行的输出序列不尽相同，但其行为模式遵循相同的同步逻辑。一次典型的运行输出日志截选如下，完整日志请参考附件 `output.txt`。

```
1  [22:10:33.847] 缓冲池初始化完毕
2  当前缓冲区映像: [  ] [  ] [  ] [  ] [  ] [  ]
3  ------------------------------------------------------------------
4  已创建 2 个生产者与 3 个消费者进程。
5  [22:10:34.140] 生产者 P2 在缓冲区 0 中放入数据 "P2_Item1"
6  当前缓冲区映像: [ P2_Item1  ] [  ] [  ] [  ] [  ] [  ]
7  ------------------------------------------------------------------
8  [22:10:34.141] 消费者 C1 从缓冲区 0 中取出数据 "P2_Item1"
9  当前缓冲区映像: [  ] [  ] [  ] [  ] [  ] [  ]
10 ------------------------------------------------------------------
11 [22:10:34.480] 生产者 P2 在缓冲区 1 中放入数据 "P2_Item2"
12 当前缓冲区映像: [  ] [ P2_Item2  ] [  ] [  ] [  ] [  ]
13 ------------------------------------------------------------------
14 [22:10:34.480] 消费者 C2 从缓冲区 1 中取出数据 "P2_Item2"
15 当前缓冲区映像: [  ] [  ] [  ] [  ] [  ] [  ]
16 ------------------------------------------------------------------
17 [22:10:34.694] 生产者 P1 在缓冲区 2 中放入数据 "P1_Item1"
18 当前缓冲区映像: [  ] [  ] [ P1_Item1  ] [  ] [  ] [  ]
19 ------------------------------------------------------------------
20 [22:10:34.694] 消费者 C2 从缓冲区 2 中取出数据 "P1_Item1"
21 当前缓冲区映像: [  ] [  ] [  ] [  ] [  ] [  ]
22 ------------------------------------------------------------------
23 // ... (后续输出)
24 [22:10:40.670] 生产者 P1 在缓冲区 5 中放入数据 "P1_Item12"
25 当前缓冲区映像: [  ] [  ] [  ] [  ] [  ] [ P1_Item12 ]
26 ------------------------------------------------------------------
27 [22:10:40.670] 消费者 C3 从缓冲区 5 中取出数据 "P1_Item12"
28 当前缓冲区映像: [  ] [  ] [  ] [  ] [  ] [  ]
29 ------------------------------------------------------------------
30 所有任务已完成。
```

4.2 结果分析

对输出日志进行分析，可以得出以下结论：

- **进程同步与互斥的有效性**
  - 实验结果清晰地展示了进程间的同步关系。例如，在 `22:10:34.140` 时生产者 P2 刚执行完生产操作，释放了 `full` 信号量，紧接着在 `22:10:34.141` 时消费者 C1 就立即执行了消费操作。这表明当缓冲池由空变为非空时，原本因等待 `full` 信号量而阻塞的消费者进程被成功唤醒。
  - 整个运行过程中，缓冲区的映像始终保持一致性和正确性，从未出现两个生产者同时写入同一位置或消费者读取已被清空的数据等竞态条件。这证明了 `mutex` 信号量成功地实现了对临界区（缓冲池）的互斥访问。
  - 输出日志呈现出一种典型的 “生产一个、消费一个” 的模式，缓冲池在绝大多数时间内仅存放一个或零个数据项。这说明在该次运行的特定调度环境下，消费者的总体速度与生产者的生产速度相匹配，系统吞吐率较高。
- **进程调度的非确定性**
  - 日志显示了操作系统进程调度的非确定性。生产者 P1、P2 和消费者 C1、C2、C3 的执行顺序是完全交错的，没有固定的先后次序。例如，先是 P2 生产，然后是 C1 和 C2 消费，接着是 P1 生产。这符合多道程序环境下，由操作系统调度器根据特定算法（如时间片轮转）动态决定哪个进程获得 CPU 时间片的原理。
- **循环队列的正确实现**
  - 从日志中可以看到，生产者和消费者的操作位置（缓冲区索引）从 0 依次递增至 5，之后又回到 0 继续。例如，在 `22:10:35.483` 时 P1 在缓冲区 5 操作后，下一次 P1 的操作在 `22:10:35.977` 时回到了缓冲区 0。这验证了 `write_pos` 和 `read_pos` 指针通过取模运算（`% BUFFER_COUNT`）

综上所述，实验结果成功地展示了一个正确同步的多进程生产者-消费者模型。信号量机制确保了数据操作的原子性和进程间的协作，共享内存则为之提供了高效的通信渠道，完整地达成了预设的实验目标。

五、 实验收获与体会

通过本次生产者-消费者问题的编程实现，我将操作系统课程中所学的抽象理论知识与具体的工程实践紧密结合，获得了深刻的理解与体会。

本次实验提供了将操作系统理论知识付诸实践的宝贵机会。在设计和编码过程中，我对进程同步与互斥的必要性和复杂性有了更为深刻的认识。理论学习中，竞态条件与数据不一致性等概念较为抽象，而通过亲手实现并观察一个无同步保护下可能出现的混乱，再到利用同步机制使其正确运行，这一过程直观地展示了同步在并发编程中的核心地位。

在实践层面，我对信号量机制的应用有了实质性的掌握。实验中对三个核心信号量的运用，让我清晰地认识到它们各自不同的职责：计数信号量 `empty` 与 `full` 作为条件变量，精确地控制了生产者与消费者的执行流，确保了进程在资源不可用时能够高效地阻塞与唤醒；而二值信号量 `mutex` 则作为互斥锁，有效地保护了临界区，保证了共享缓冲池数据在任何时刻的完整性。将抽象的 **P**、**V** 操作对应到具体的 `WaitForSingleObject` 和 `ReleaseSemaphore` Windows API 调用，是本次实验的核心技术收获。

此外，本实验加深了我对进程间通信（IPC）机制的理解。操作系统原理指出，进程在内存空间上是相互独立的。共享内存机制通过 `CreateFileMapping` 和 `MapViewOfFile` 等 API 在这些独立的地址空间之间架设了一座高效的数据桥梁。这种允许多个进程直接读写同一内存区域的方式，为大规模数据交换提供了高性能的解决方案。

最后，通过分析实验结果中非确定性的输出序列，我直观地体会到了操作系统进程调度的内在复杂性。这也让我深刻认识到，在编写任何并发程序时，都绝不能对进程的执行顺序做出任何时序上的假设。程序的正确性必须且只能依赖于健全、可靠的同步机制来保障。这对于未来学习和从事分布式系统、多线程服务器开发等领域的工作，是一次至关重要的基础训练。