

# Lab1 生产者与消费者问题

2022011120

方睿泉

## 环境配置

在Ubuntu20.04, Vscode中运行

需修改 `c_cpp_properties.json`

```
{
  "configurations": [
    {
      "name": "Linux",
      "includePath": [
        "${workspaceFolder}/**"
      ],
      "defines": [],
      "cStandard": "c17",
      "cppStandard": "c++20",
      "intelliSenseMode": "linux-gcc-x64"
    }
  ],
  "version": 4
}
```

类似的, `tasks.json` 中也需要添加两个参数, 参见 `agrs` 列表

其中, `-pthread` 是外部库, 编译时需指明, `-std=c++20` 则用于指明版本, 因为 `semaphore` 库直到 `c++20` 作为标准库的一部分被引入。

另外吐槽, Ubuntu20.04中使用 `sudo apt install build-essential` 默认的gcc/g++版本是9,经检查, 该版本 `/usr/include` 不包含`semaphore`库, 只有 `semaphore.h`。本人在高铁上以 `kb/s` 的网速花了一个多小时才下好gcc/g++ 10和11并测试, 发现 `semaphore` 是在 `gcc-11` 中。具体位置是 `/usr/include/c++/11`

对应命令为

```
g++-11 -std=c++20 -pthread -o producer_consumer producer_consumer.cpp
```

```
{
  "tasks": [
    {
      "type": "cppbuild",
      "label": "C/C++: g++-11 build active file",
      "command": "/usr/bin/g++-11",
      "args": [
        "-fdiagnostics-color=always",
        "-std=c++20",
        "-pthread",
        "-g",
        "${file}",
        "-o",
        "${fileDirname}/${fileBasenameNoExtension}"
      ],
      //其余略
    }
  ],
  "version": "2.0.0"
}
```

## 代码补全

基本的调用库函数即可实现。  
完整的代码放在包报告尾部。

```

//producer
for (int i = 0; i < 20; ++i)
{
    produce_item(i); // 生产一个新的项目

    // TODO: 等待缓冲区有空位
    empty.acquire();

    // TODO: 进入临界区
    mutex.acquire();

    buffer[in] = i;    // 将项目放入缓冲区
    in = (in + 1) % N; // 更新下一个生产者将放置项目的位置

    // TODO: 离开临界区
    mutex.release();

    // TODO: 增加缓冲区已占用位置数量
    full.release();
}

//consumer
for (int i = 0; i < 20; ++i)
{
    // TODO: 等待缓冲区有数据
    full.acquire();

    // TODO: 进入临界区
    mutex.acquire();

    int item = buffer[out]; // 从缓冲区取出项目
    out = (out + 1) % N;    // 更新下一个消费者将取出项目的位置

    // TODO: 离开临界区
    mutex.release();

    // TODO: 增加缓冲区空余位置数量
    empty.release();

    consume_item(item); // 使用项目
}

```

# 代码分析

## 基本介绍

[api reference document](#)

### 基本类

1. `counting_semaphore` 是一个轻量同步元件，能控制对共享资源的访问。不同于 `std::mutex`，`counting_semaphore` 允许同一资源有多于一个同时访问，至少允许 `LeastMaxValue` 个同时的访问者若 `LeastMaxValue` 为负则程序为谬构。
- 2.
3. `binary_semaphore` 是 `std::counting_semaphore` 的特化的别名，其 `LeastMaxValue` 为 1。实现可能将 `binary_semaphore` 实现得比 `std::counting_semaphore` 的默认实现更高效。

### 成员函数

1. `release`  
增加内部计数器并除阻获取者
2. `acquire`  
减少内部计数器或阻塞到直至能如此
3. `try_acquire`  
尝试减少内部计数器而不阻塞
4. `try_acquire_for`  
尝试减少内部计数器，至多阻塞一段时长
5. `try_acquire_until`  
尝试减少内部计数器，阻塞直至一个时间点
6. `max`  
返回内部计数器的最大可能值

### 基本逻辑

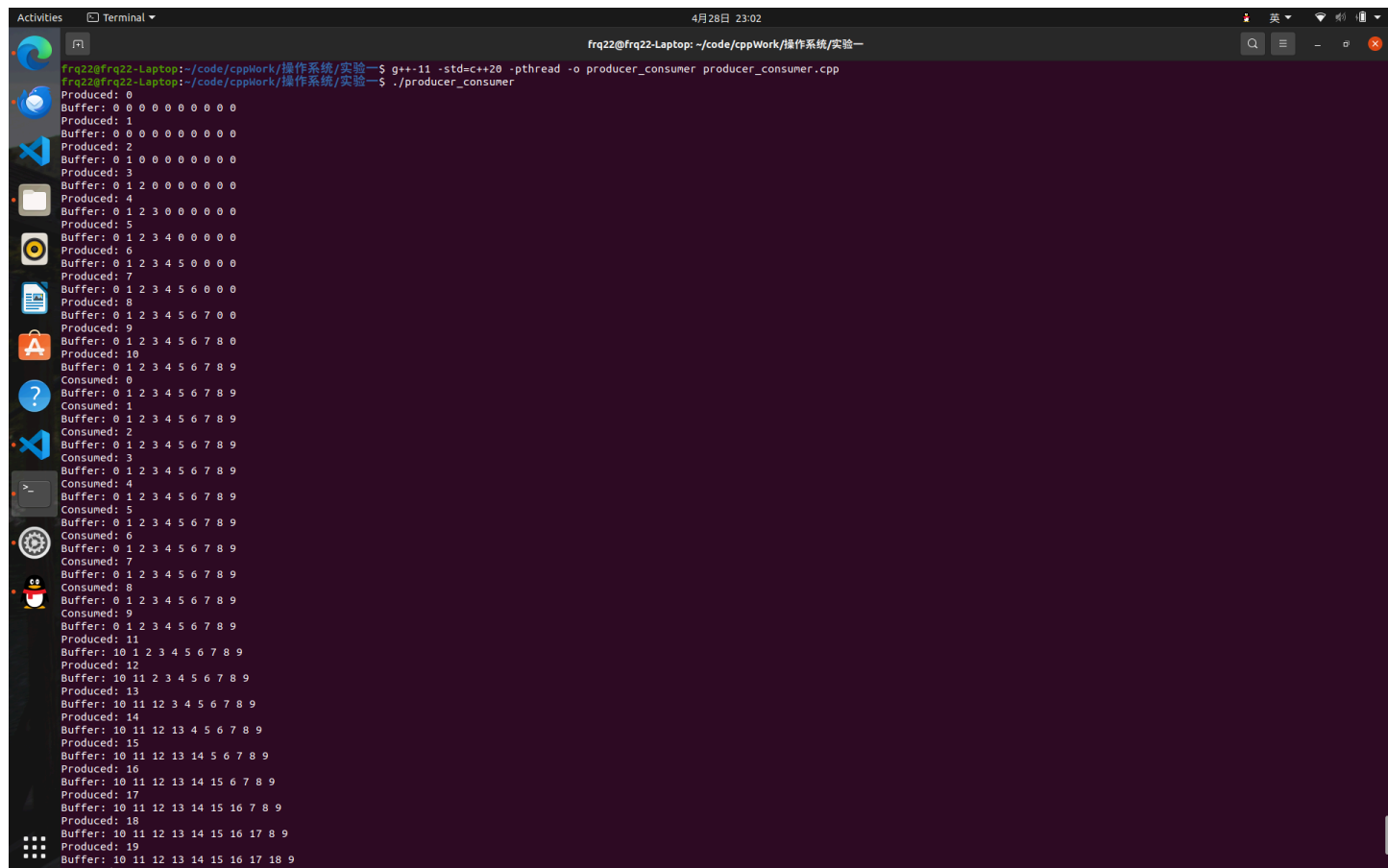
`full` 和 `empty` 分别用来记录缓冲区空余的、占用的位置数目。

`producer` 和 `consumer` 程序内部第一个语句都调用了 `acquire` 方法。`producer`负责减少空位数，并在线程结束时增加占用数。`consumer`负责减少占用数，并在线程结束时增加空位数。

对临界区的处理则是调用了 `mutex` 这个初始值为1的信号量。进入临界区需要保证`mutex`大于0，并且在单个for循环后会将`mutex`重新释放为1，这样保证了两个线程的每个for循环都是原子的，实现互斥访问。

经过实验源代码时常出现输出混乱的问题，猜测是`cout`流同步（或者类似）的问题。更换成`printf`后输出正常。

输出示例如下图。



```
frq22@frq22-Laptop: ~/code/cppWork/操作系统/实验一$ g++-11 -std=c++20 -pthread -o producer_consumer producer_consumer.cpp
frq22@frq22-Laptop: ~/code/cppWork/操作系统/实验一$ ./producer_consumer
Produced: 0
Buffer: 0 0 0 0 0 0 0 0 0 0
Produced: 1
Buffer: 0 0 0 0 0 0 0 0 0 0
Produced: 2
Buffer: 0 1 0 0 0 0 0 0 0 0
Produced: 3
Buffer: 0 1 2 0 0 0 0 0 0 0
Produced: 4
Buffer: 0 1 2 3 0 0 0 0 0 0
Produced: 5
Buffer: 0 1 2 3 4 0 0 0 0 0
Produced: 6
Buffer: 0 1 2 3 4 5 0 0 0 0
Produced: 7
Buffer: 0 1 2 3 4 5 6 0 0 0
Produced: 8
Buffer: 0 1 2 3 4 5 6 7 0 0
Produced: 9
Buffer: 0 1 2 3 4 5 6 7 8 0
Produced: 10
Buffer: 0 1 2 3 4 5 6 7 8 9
Consumed: 0
Buffer: 0 1 2 3 4 5 6 7 8 9
Consumed: 1
Buffer: 0 1 2 3 4 5 6 7 8 9
Consumed: 2
Buffer: 0 1 2 3 4 5 6 7 8 9
Consumed: 3
Buffer: 0 1 2 3 4 5 6 7 8 9
Consumed: 4
Buffer: 0 1 2 3 4 5 6 7 8 9
Consumed: 5
Buffer: 0 1 2 3 4 5 6 7 8 9
Consumed: 6
Buffer: 0 1 2 3 4 5 6 7 8 9
Consumed: 7
Buffer: 0 1 2 3 4 5 6 7 8 9
Consumed: 8
Buffer: 0 1 2 3 4 5 6 7 8 9
Consumed: 9
Buffer: 0 1 2 3 4 5 6 7 8 9
Produced: 11
Buffer: 10 1 2 3 4 5 6 7 8 9
Produced: 12
Buffer: 10 11 2 3 4 5 6 7 8 9
Produced: 13
Buffer: 10 11 12 3 4 5 6 7 8 9
Produced: 14
Buffer: 10 11 12 13 4 5 6 7 8 9
Produced: 15
Buffer: 10 11 12 13 14 5 6 7 8 9
Produced: 16
Buffer: 10 11 12 13 14 15 6 7 8 9
Produced: 17
Buffer: 10 11 12 13 14 15 16 7 8 9
Produced: 18
Buffer: 10 11 12 13 14 15 16 17 8 9
Produced: 19
Buffer: 10 11 12 13 14 15 16 17 18 9
```

经过多次验证和仔细查看，并未发现有“生产超过空余数”和“消费超过现有数”的现象。

## PV原语的思考

PV原语在本问题的模拟中，很好的模拟了被容量限制的生产和消费问题。acquire和release分别代表着P，V操作。

在生产者和消费者问题中，锁的作用是保证操作的原子性，不会在生产的过程中消费，或者相反。保证了对共享资源的互斥访问。

P原语和锁不能调换，否则这样可能会出现上锁之后发现资源为0无法操作，这样就导致了死锁。

# 完整代码

```
#include <stdio>
#include <thread>
#include <semaphore>

const int N = 10; // 缓冲区大小
int buffer[N];    // 缓冲区
int in = 0;       // 下一个生产者将放置项目的位置
int out = 0;      // 下一个消费者将取出项目的位置

std::counting_semaphore<> empty(N); // 缓冲区空余位置数量
std::counting_semaphore<> full(0);  // 缓冲区已占用位置数量
std::binary_semaphore mutex(1);    // 用于缓冲区的互斥访问

void produce_item(int item)
{
    // 生产项目的代码2
    printf("Produced: %d\n", item);
    printf("Buffer: ");
    for (int i = 0; i < 10; i++)
    {
        printf("%d ", buffer[i]);
    }
    printf("\n");
}

void consume_item(int item)
{
    // 消费项目的代码
    printf("Consumed: %d\n", item);
    printf("Buffer: ");
    for (int i = 0; i < 10; i++)
    {
        printf("%d ", buffer[i]);
    }
    printf("\n");
}

void producer()
{
    for (int i = 0; i < 20; ++i)
    {
        produce_item(i); // 生产一个新的项目
    }
}
```

```

// TODO: 等待缓冲区有空位
empty.acquire();

// TODO: 进入临界区
mutex.acquire();

buffer[in] = i;    // 将项目放入缓冲区
in = (in + 1) % N; // 更新下一个生产者将放置项目的位置

// TODO: 离开临界区
mutex.release();

// TODO: 增加缓冲区已占用位置数量
full.release();
}
}

void consumer()
{
    for (int i = 0; i < 20; ++i)
    {
        // TODO: 等待缓冲区有数据
        full.acquire();

        // TODO: 进入临界区
        mutex.acquire();

        int item = buffer[out]; // 从缓冲区取出项目
        out = (out + 1) % N;    // 更新下一个消费者将取出项目的位置

        // TODO: 离开临界区
        mutex.release();

        // TODO: 增加缓冲区空余位置数量

        empty.release();

        consume_item(item); // 使用项目
    }
}

int main()
{
    std::thread producerThread(producer);
    std::thread consumerThread(consumer);

```

```
producerThread.join();  
consumerThread.join();
```

```
return 0;
```

```
}
```