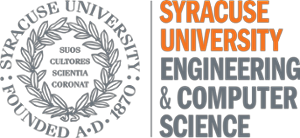
**CSE 691 Parallel Prog and Multi-threading**



**Spring 2023**

**HW4 Project Report**

**Plant simulation program**

Name: Dongze Xie

NetID: dxie03

SUID: 375402964

Date: Dec 11, 2022

**Abstract**

This is a simulation program for a plant with a specific assembly line pattern. The project is compatible with C++ 14, 17, and 20 standards, and Visual Studio 2022 is used as the compiler. The program utilizes various techniques, such as mutex, condition variables, threads, atomic, file operations, and data structures, to implement multiple workers as threads working simultaneously in their own processes. Two types of workers work together to produce products that meet the given order.

**Table of contents**

目录

[HW4 Project Report 1](#_Toc19733)

[Abstract 2](#_Toc26680)

[Table of contents 3](#_Toc7691)

[1. Working Process and Standard 4](#_Toc30926)

[2. Original Implementation 5](#_Toc32256)

[2.1Classes 5](#_Toc20333)

[2.1.1 Buffer 5](#_Toc5069)

[2.1.2 Part Worker 6](#_Toc18856)

[2.1.3 Product Worker 6](#_Toc22610)

[2.2 Functions 6](#_Toc12308)

[2.2.1 Overview 6](#_Toc8251)

[2.2.2 Random Orders 7](#_Toc8559)

[2.2.3 Working process function 8](#_Toc16651)

[2.2.4 Print functions for Logging 13](#_Toc28674)

[2.3 main function 14](#_Toc4972)

[2.4 Log information sample 15](#_Toc19247)

[3. Upgraded Implementation 17](#_Toc25877)

[3.1 Strategy overview 17](#_Toc7748)

[3.1.1 Set cost 17](#_Toc16068)

[3.1.2 Multiple buffer working mechanism 18](#_Toc5898)

[3.2 Multiple buffer implementation 19](#_Toc18203)

[3.2.1 The ‘preFill’ process 19](#_Toc6569)

[3.2.2 Buffer switch 20](#_Toc4458)

[3.3 Log update 21](#_Toc21722)

[3.4 Optimization analysis 23](#_Toc24479)

1. **Working Process and Standard**

Consider a plant, where there are ***m* part workers**(each will be implemented with a thread) whose jobs are to produce five types of parts (A, B, C,D, E). Each part worker will produce 6pieces of all possible combinations, such as (2,0,0,2,2), (0,2,1,0,3), (6,0,0,0,0), etc., given that it takes 500, 500,600, 600, 700 microseconds(*us*)to make each part of type A, B, C, D, E, respectively. For example, it takes a part worker 500\*2 +600\*3+700*us* to make a (2,0,0,3,1) part combination.Each part worker will attempt to load the produced parts to a buffer area, which has a capacity for 7, 6, 5, 5, 4pieces of type A, B, C, D, E parts, respectively. That is**, the buffer capacity is (7,6,5,5,4).** It will take a part worker 200, 200, 300, 300,400 *us* to move a part of type A, B, C, D,E,respectively, to/from the buffer. Each part combination, such as (1,0,2,2,1) is referred to as a **load order**.

The current number of parts of each type in the buffer, such as (5,2,1,3,2) is referred to as **buffer state**.A part worker will load the number of parts of each type to the buffer, restricted by the buffer’s capacity of each type. For example, if a load order is (2,1,0,2,1) and the buffer state is (7,2,1,4,2), then the part worker can place a type B part, a type D and a type E part to the buffer; thus, the updated load order will be (1,0,0,1,0) and the updated buffer state will be (7,3,1,5,3). The part worker will **wait** near the buffer area for the buffer space to become available to complete the load order. If the wait time reaches **MaxTimePart** us(maximum wait time for a part

worker), the part worker will stop waiting, move the un-loaded parts back, and randomly re-generate a new load order, which must re-use the previously un-loaded parts(that got moved back). Recall that it takes a part worker 200, 200, 300, 300, 400usto move a part of type A, B, C, D, E, respectively. A part worker will then repeat the process to produce a brand-new load order.

In addition, there are ***n* product workers**(each implemented as a thread)whose jobs are to take the parts from the buffer area and assemble them into products. Each product assembly needs 5pieces of parts each time; however, the 5pieces will be from exactly two or three types of parts, such as (1,2,2,0,0), (1,0,3,1,0), (1,0,0,0,4),

(0,2,3,0,0), etc. with equal occurrence probability. For example, a product worker will not generate an order of (1,1,2,1,0), (5,0,0,0), etc.Each such legal combination from a product worker is referred to as a **pickup order**.The time it takes a product worker to move a part of type A, B, C, D, E from the buffer is 200, 200,300,300,400 us, respectively. Like that for part workers, partial fulfillment policy is adopted. If the current buffer state is (4,0,2,1,3) and a pickup order is (1,1,0,0,3), then the updated buffer state will be (3,0,2,1,0) and the updated pickup order will be (0,1,0,0,0). Note that when a product worker goes to the buffer area to pick up parts, the product worker will pick up parts and load them on the cart. At this moment, the numbers of parts on the cart will be (1,0,0,0,3), which is referred to as **cart state**.The product

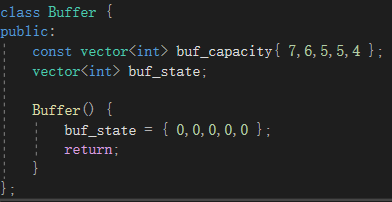
worker will wait next to the buffer area, looking to complete the pickup order. Once all needed parts are obtained, they will be moved back to assembly area and then assembled into products. The move time for parts of each type has been described. The assembly time needed for parts of type A, B, C, D, E,will be 600, 600, 700, 700,800 us,respectively. If the wait time reaches **MaxTimeProduct** us(maximum wait time for a product worker), the product worker will move back the parts already picked up and randomly re-generate a load order which has tore-use all the parts that were moved back.The product worker will then re-produce a brand-new pickup order.If the parts moved back after timeout event are (1,1,0,0,0). During the next iteration, if a new pickup order (2,1,2,0,0) is generated, then the real pickup order

that the product worker will bring to the buffer area is (1,0,2,0,0), while the parts (1,1,0,0,0), which were brought back during the previous iteration due to timeout event, will stay at local area; (1,1,0,0,0) will be referred to as **local state**.

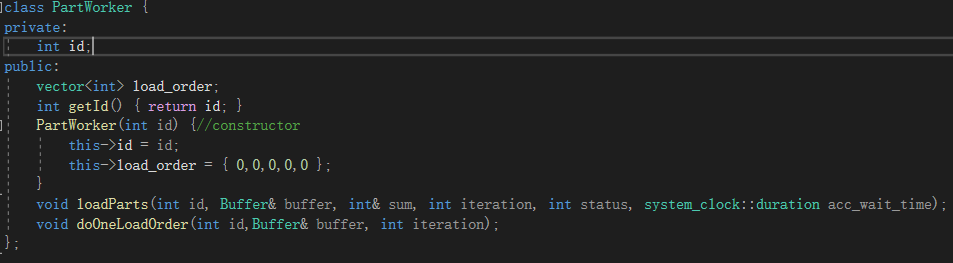
1. **Original Implementation**

The goal is to develop a simulation program of the activity of the above-described plant, which designed to improve the performance of the plant, while ensuring a fair treatment to all workers. Each part work or product worker is said to have completed one iteration when a load order or pickup order is completed, or if timeout event occurs such that an order is aborted.Clearly we need to protect the shared resource, buffer, with proper lock/mutex. Every time when a part worker thread or a product worker thread gain the access to the shared resource, we need to print information to a file called log.txt . Note that each product worker thread will also print the total number of completed products.

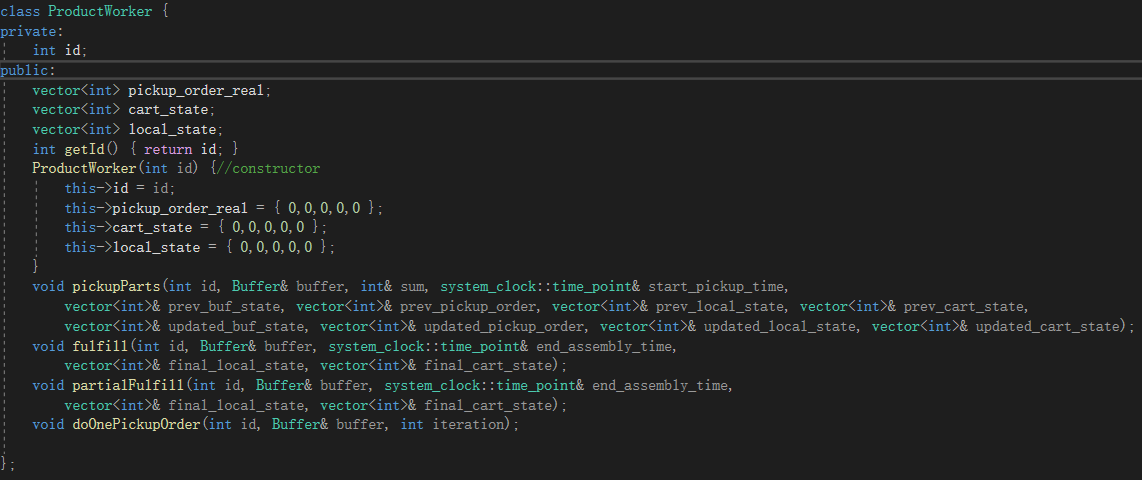
**2.1Classes**

**2.1.1 Buffer**

**2.1.2 Part Worker**

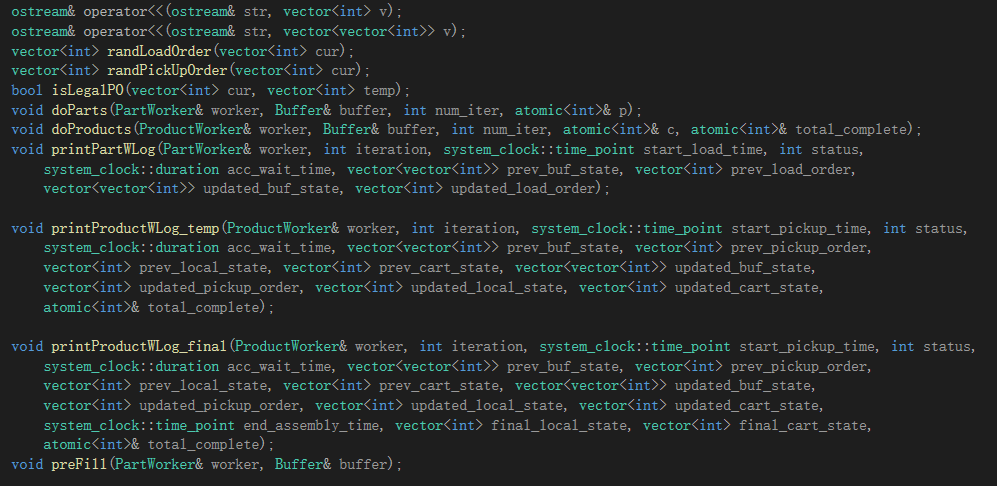


**2.1.3 Product Worker**

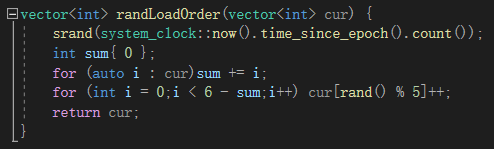


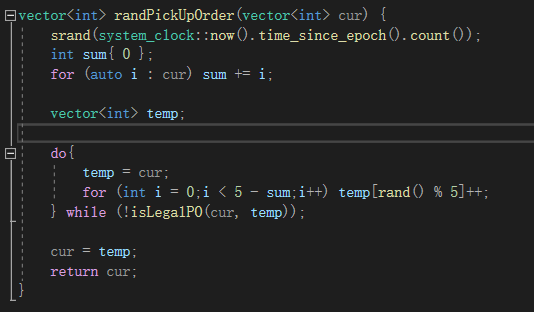
**2.2 Functions**

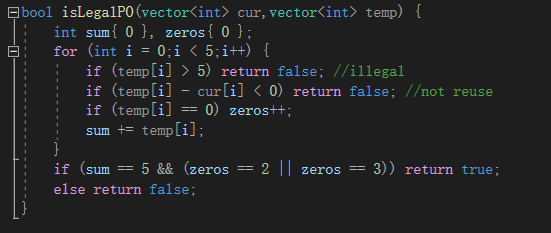
**2.2.1 Overview**



**2.2.2 Random Orders**

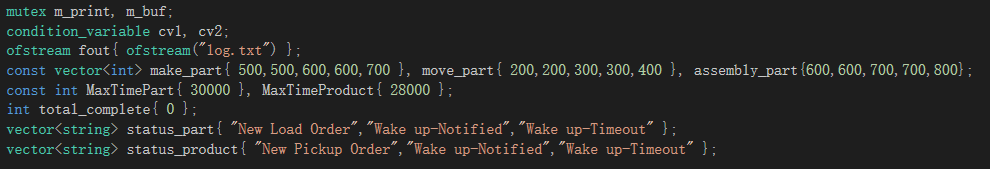
Set microsecond level time point as seed to perform rand() function. Each time add one to a random slot among all five slots. Note that the orders requires to follow a specific pattern, so every time it violates the pattern, we should revoke and random again. And sometimes there are remaining parts need to reuse from the former iteration, so we should take the current state as parameter and perform random base on it.  






**2.2.3 Working process function**

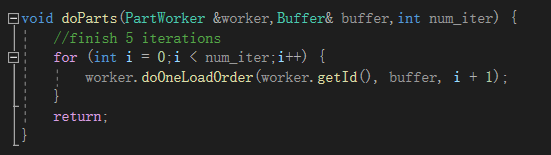
Note that m\_print is the mutex and cv1, cv2 are two conditional variables for each type of worker. Other consts are also listed below:

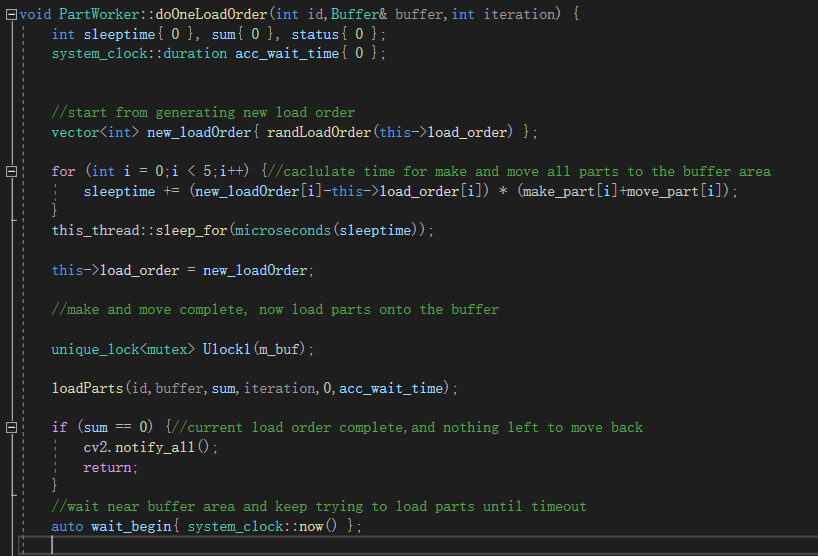


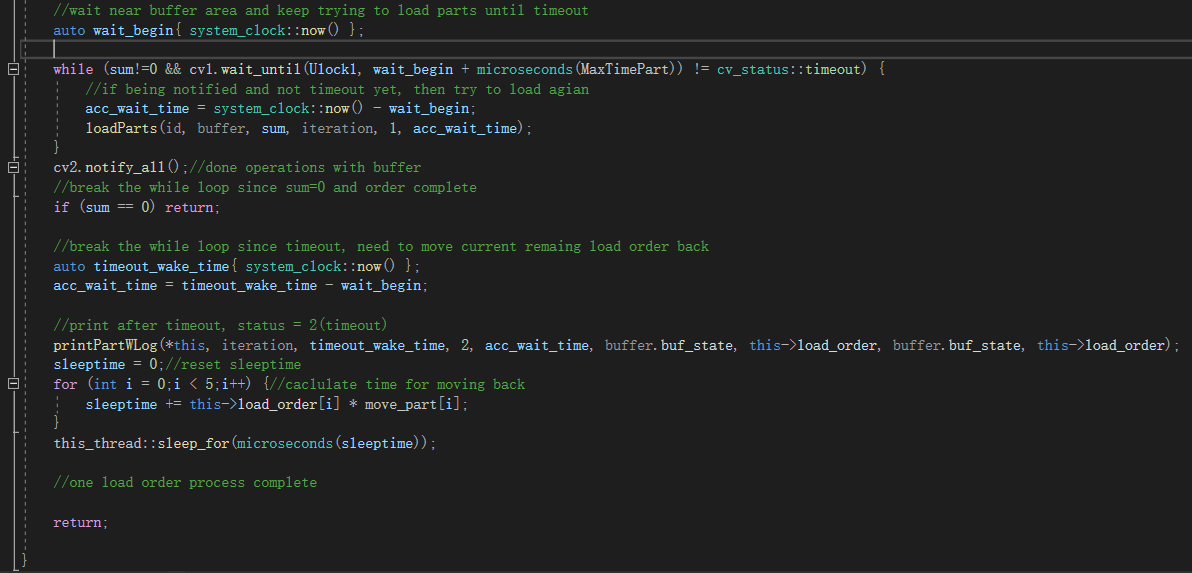
**2.2.3 (1) Part worker process**

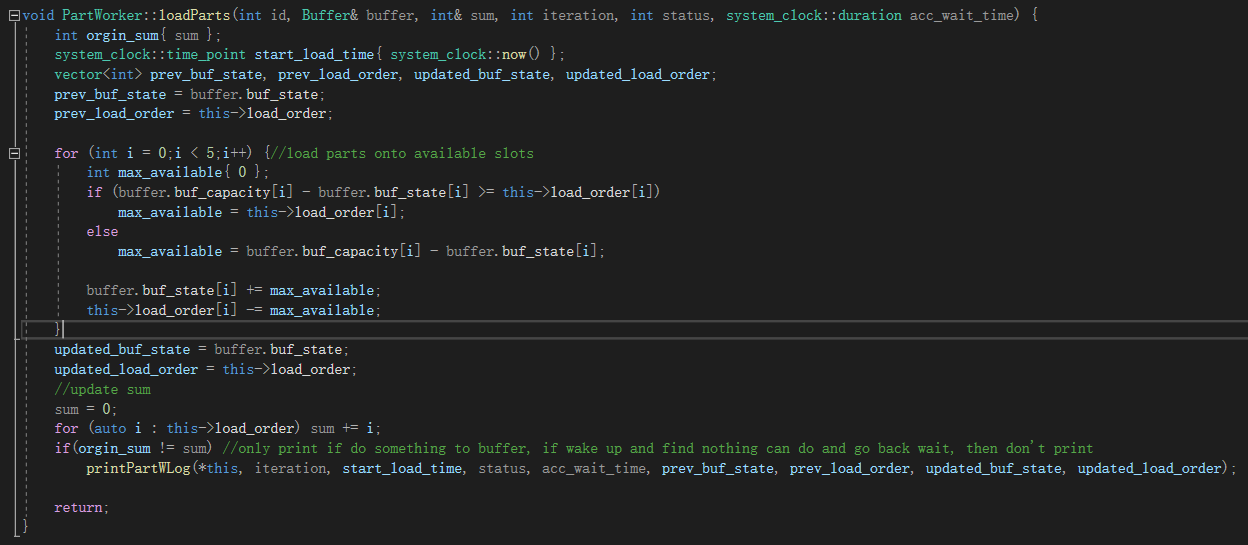
**doParts()** function executes x iterations of part worker working process.

**doOneLoadOrder()** function sequentially perform each task of a part worker.  
**loadParts()** function is for the convenience of retrying the loading operation.It is an abstraction when a part worker needs to load the parts they currently have onto the buffer.







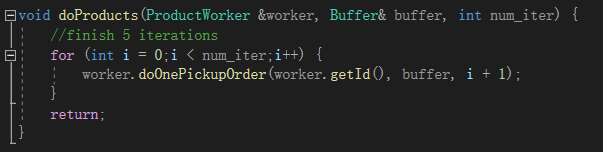


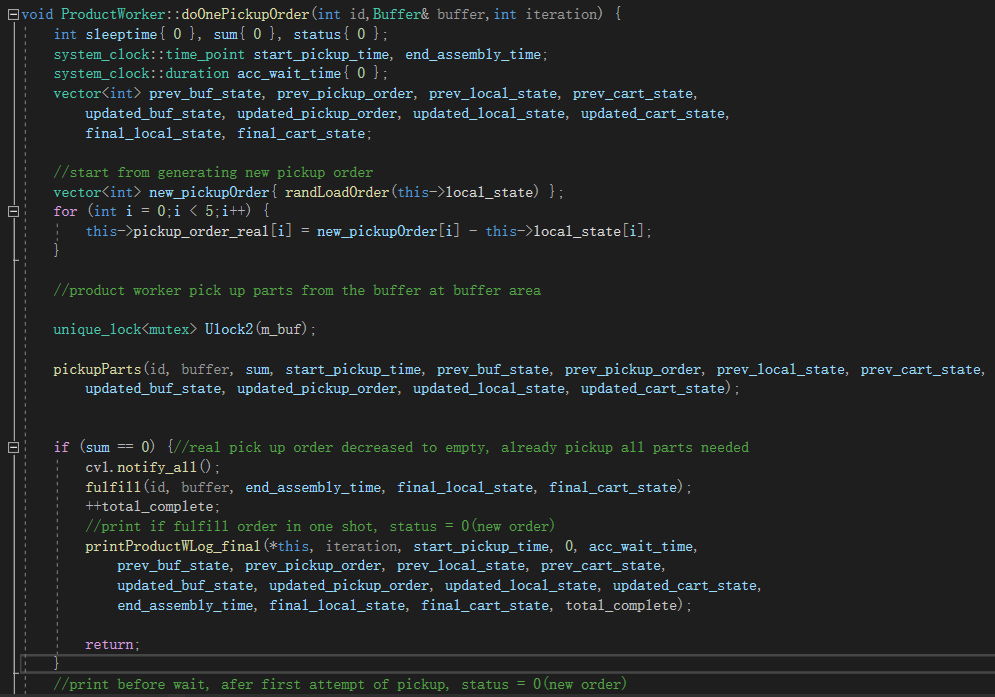
**2.2.3 (2) Product worker process**

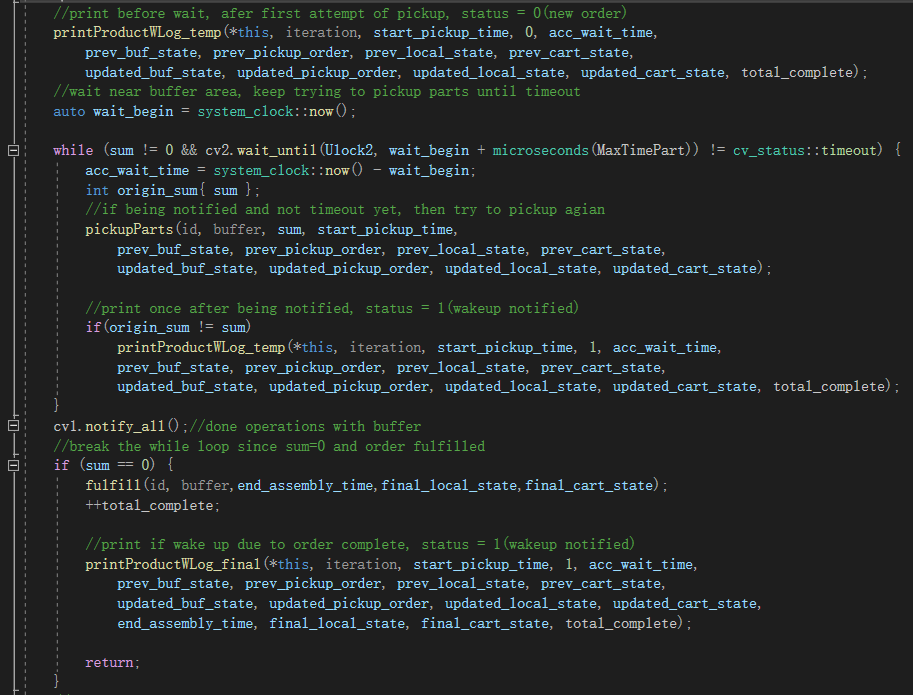
**doProducts()** function executes x iterations of part worker working process.

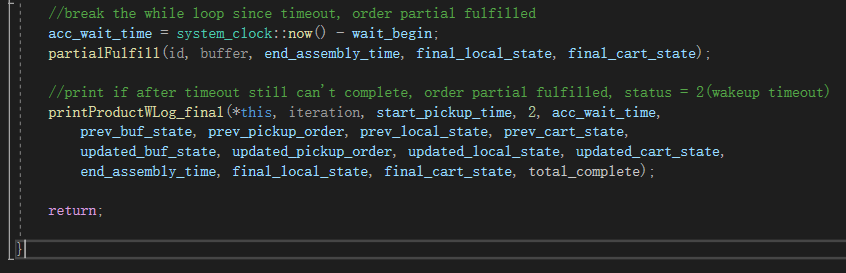
**doOnePickupOrder()** function sequentially perform each task of a part worker.  
**pickupParts()** function is for the convenience of retrying the pickup operation.It is an abstraction when a part worker needs to pickup the parts from the buffer to the cart.

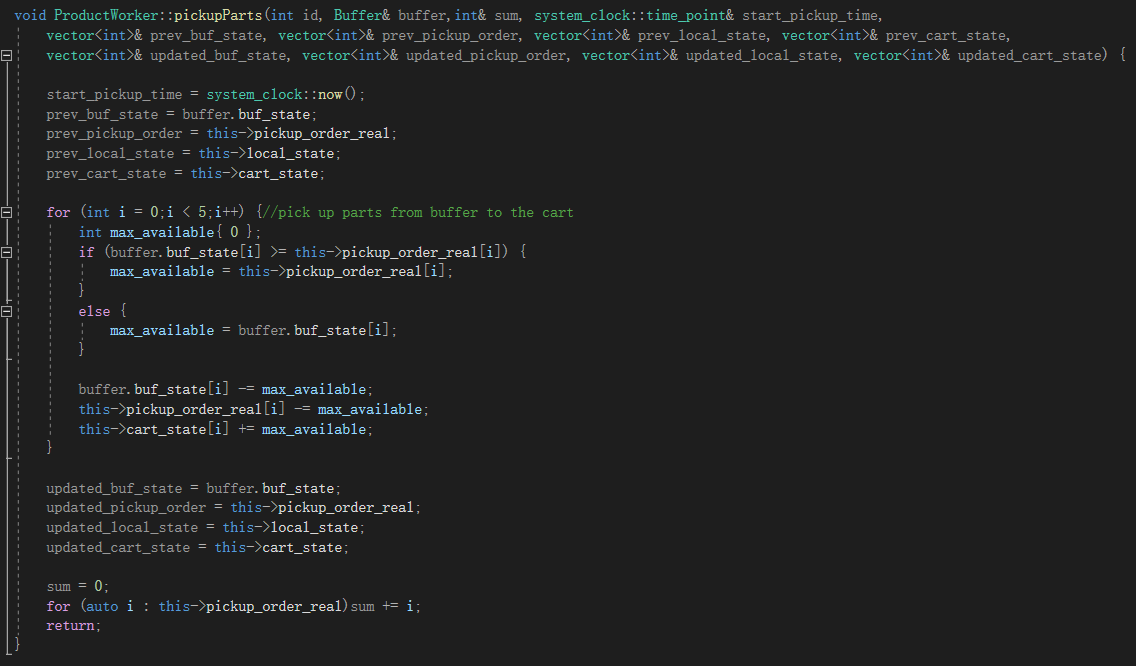
**fulfill()** and **partialFulfill()** are the functions dealing with different iteration completion ways of a product worker.

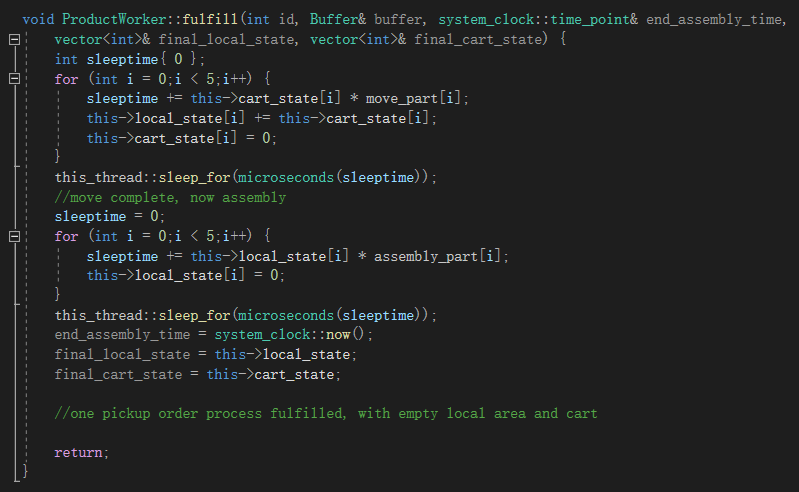


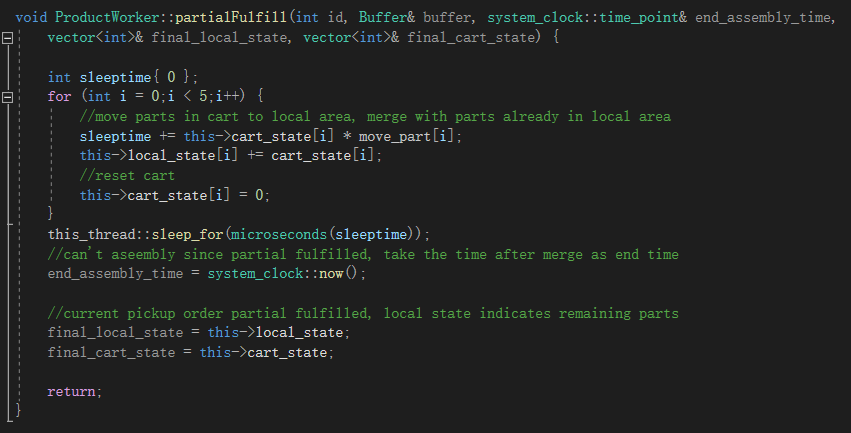






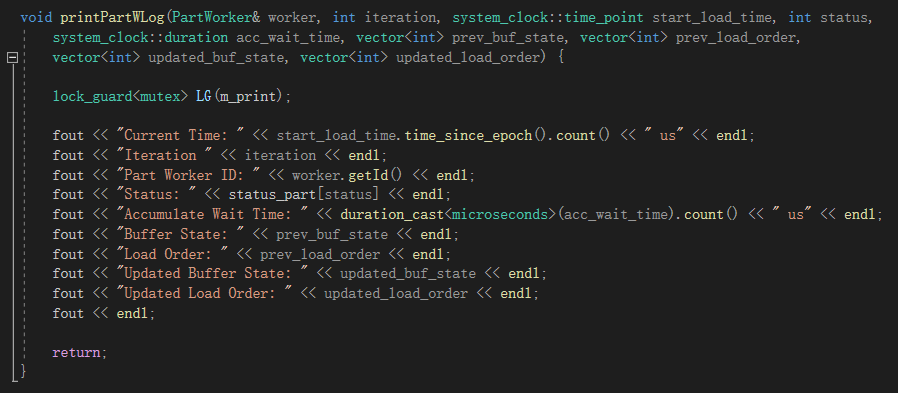


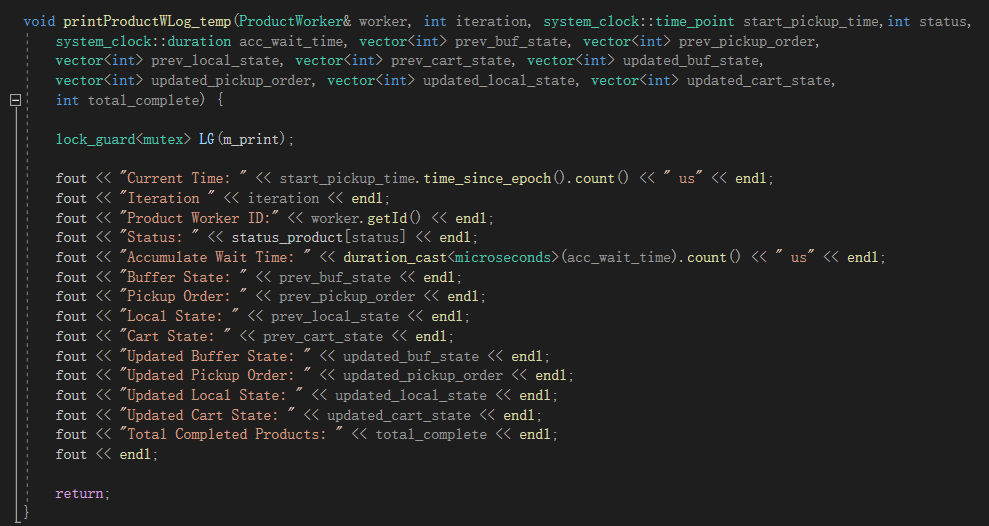


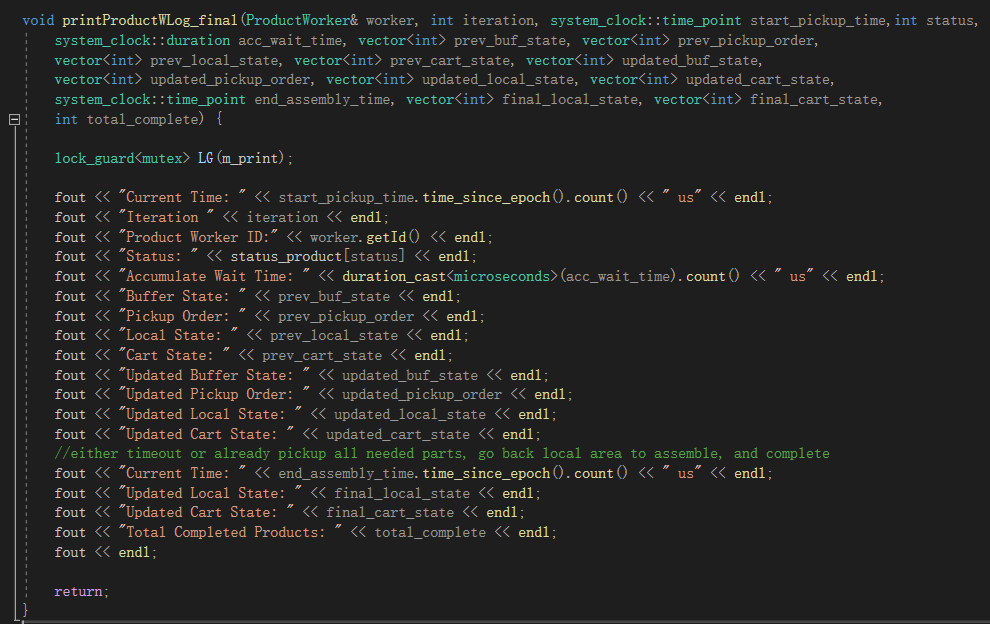


**2.2.4 Print functions for Logging**

ofstream fout{ ofstream("log.txt") };

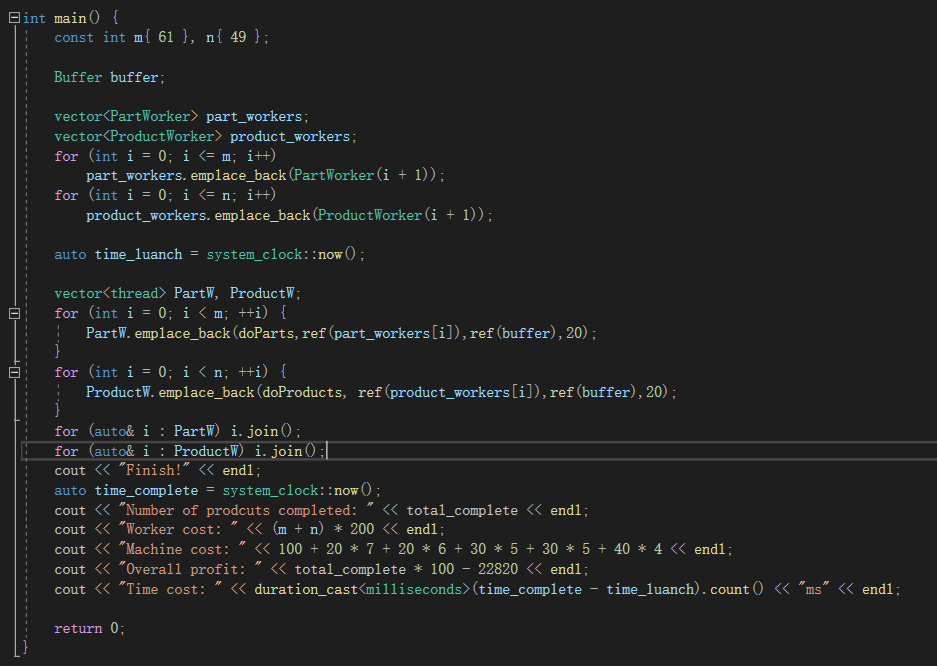


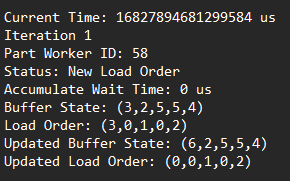


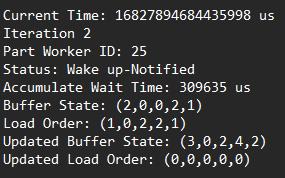
  
**2.3 main function**

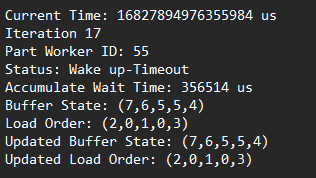
Set the number of part workers(m) and product workers(n).  
Created instance of buffer and two types of workers, and assign them to execute doParts() or doProducts() function in their own thread.

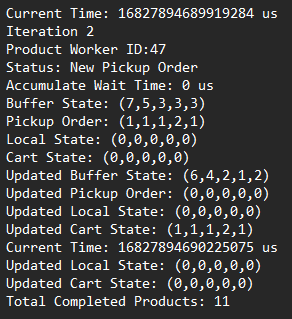
The log message will be output to the file log.txt, while some information are print to the console for efficiency analysis.

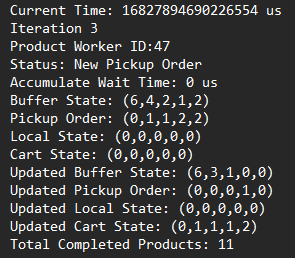


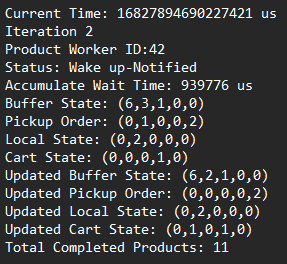
**2.4 Log information sample**

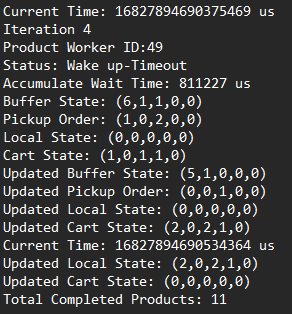












1. **Upgraded Implementation**

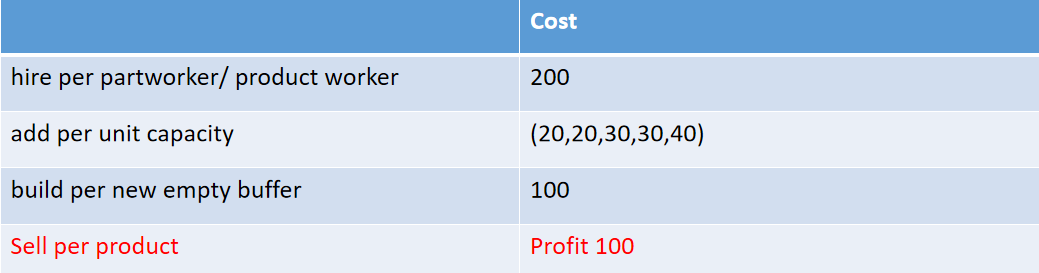
**3.1 Strategy overview**

**3.1.1 Set cost**

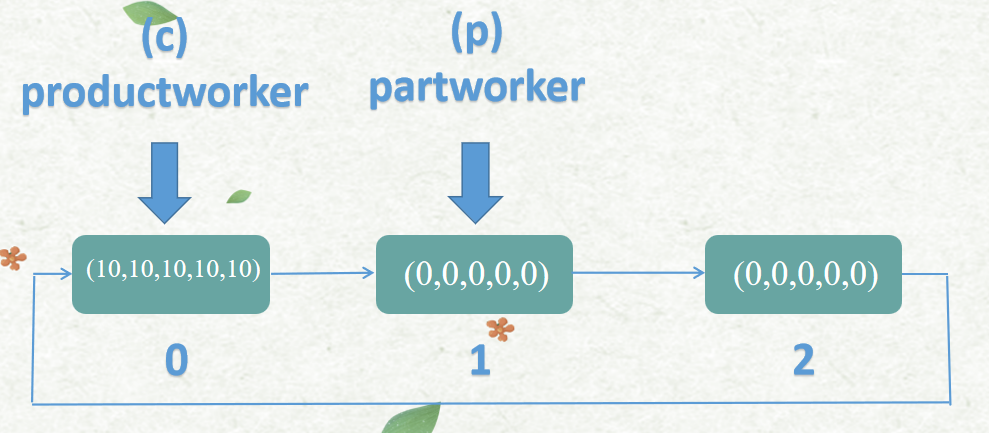
To reduce the compete of sharing resource, I adopt the combination of adding more capacity to the buffer and adding more buffers. Note that this operation is not unlimited, so I set cost for each modification which can improve the productivity.

We need to balance all these costs to make more profit, and take it as a benchmark.  
And also, the per unit capacity of buffer should be set differently based on different types of parts. I simply associated it with the time we need to move or make the parts,

that is the larger a part the more space it needed to be stored in.The specific number is shown as the table below:



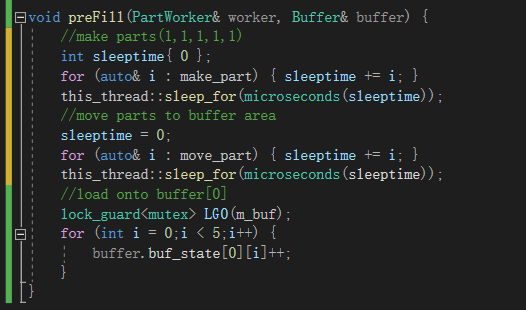
**3.1.2 Multiple buffer working mechanism**

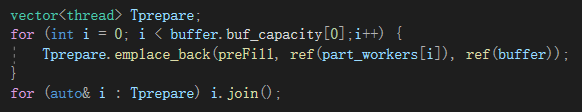
To make full use of multiple buffer, we should not let workers access a buffer pool randomly. Instead, we need to make part workers more likely to access a nearly empty buffer, so that they can load all parts they have and have more possibility to completed the order with one attempt without any waiting process.On the other hand,  
product workers should also be managed to access a nearly full buffer more easily, for the same reason of higher chance to complete the order without waiting.My strategy is to use a 2D vector to store the buffers, note that each buffer is also a vector of 5 integers. The new working process will be: workers will try to complete their own task on the current buffer as first. If it fails, then give the second try on the next buffer which has high possibility to success. And if the second attempt also fails, they will stay at this buffer and wait for notification, for the rest are just like the original version, which is keep trying until complete or timeout.

So, the key is to determine which buffer is the next a worker should try for. Here I introduce a new process (preFill()), which is letting some part workers to start ahead of time. These part workers will fill the index 0 buffer to its fullness, and the preparation is done. When the program runs, part workers are initially start at index 1, which is empty, while product workers are initially start at index 0, which is full. Suppose their are x buffers in total,which is, the length of buffer vector, then once a worker need to switch buffer, he will move to the index (cur+1)%x. In this way, it works like product workers ‘chasing’ part workers to consume the parts they left, and it performs circularly. Though the preparation process will cost some time, it is worthy than spending extra money for hiring more part workers than product workers, to generate a supply overflow which cause part workers take the lead and make full use of multiple buffers. Furthermore, the preparation process is also done in parallel, as a result, the time cost would be even less.Here I let the same as the capacity of the buffer numbers of part workers start simultaneously to make and move one part of each type, and load them onto the index 0 buffer.  
  
 Note that we access the buffer via index, so we need two variables for each type of worker(p and c, p stands for producer which is for part worker, c stands for consumer which is for product worker). And when we need to switch to the next buffer, this operation should be done atomically, which means, at one time, only one worker has the authority to modify the p or c value. Therefore, p and c should be atomic<int> and use the compare and exchange function to update value.

**3.2 Multiple buffer implementation**

**3.2.1 The ‘preFill’ process**



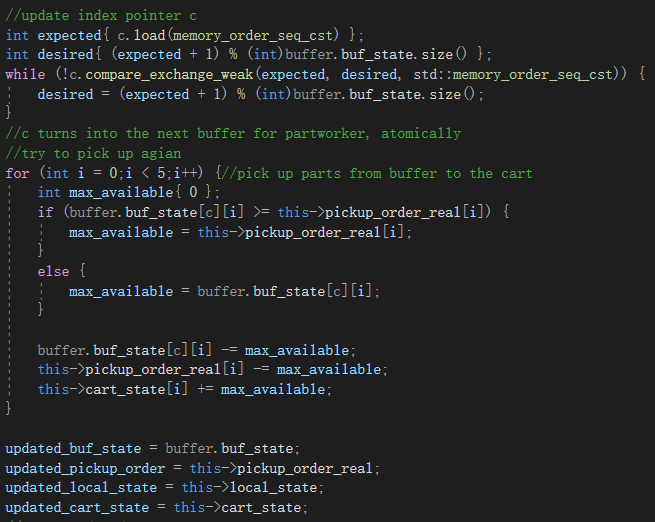


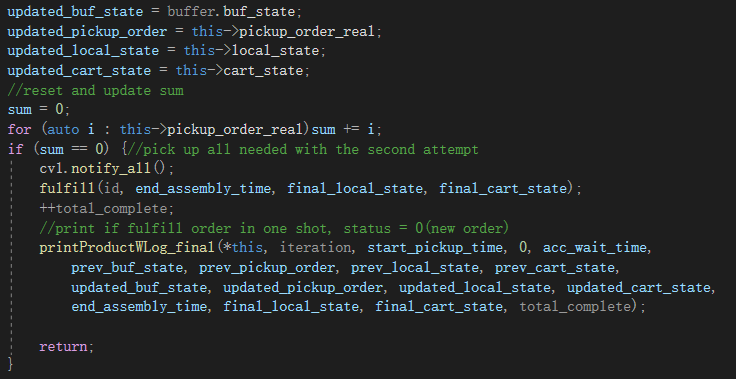
**3.2.2 Buffer switch**

Note that wherever it takes the buffer object as parameter it should take atomic<int> p or c to access a specific slot. And in the upgraded version I also make the total complete variable atomic<int>.

Modification in loadParts():

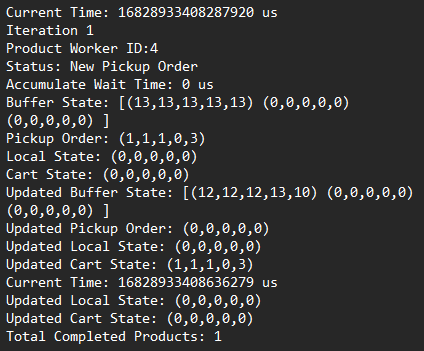
Modification in doOnePickupOrder():

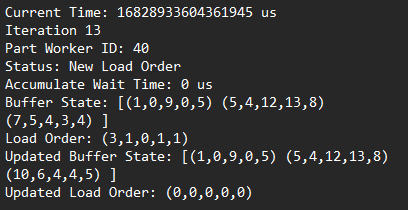


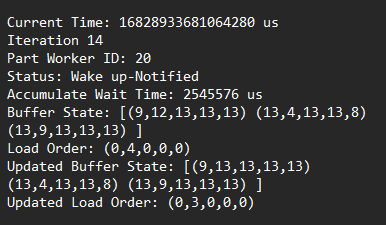


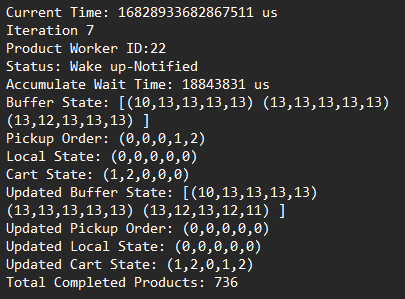
**3.3 Log update**

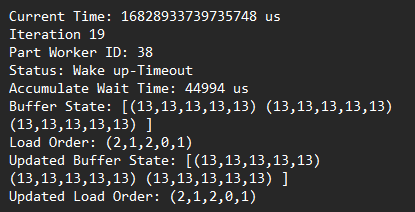
Now the buffer state information should show all the buffers.

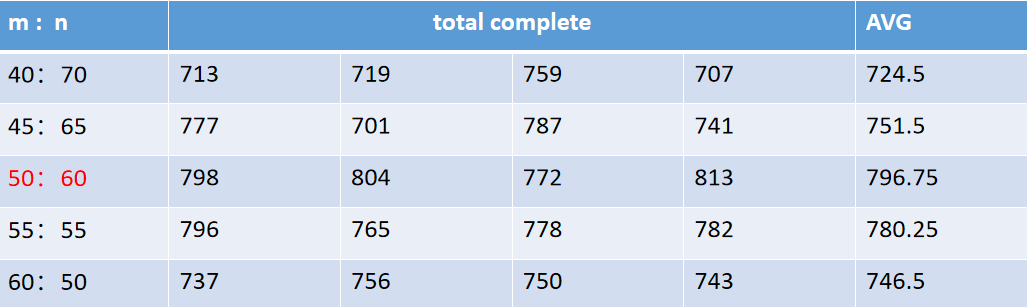
Here are some samples:  




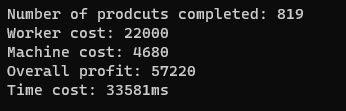


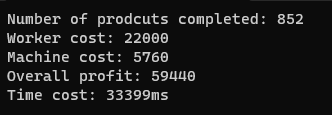


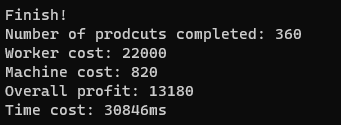
 **3.4 Optimization analysis**

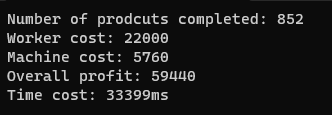
Besides the former modification we made, we still need to set proper values to specific variables to balance the trade off in order to gain more productivity for this program. First part would be the ratio of numbers of two type of workers. Since a load order contains 6 parts while a pickup order contains 5 parts, we may assume m:n = 5:6 is a proper solution. Then through the test under total workers hired=110, three buffers used, each with capacity(10,10,10,10,10), we can find that 50: 60 is the optimal indeed.(Note that since the order generate and OS scheduling is random, every time the test results will fluctuate slightly, 49: 61and 51:59 is also considered works well).  


Second part would be the combination of how many buffers we should use and the capacity of them.It turns out that keep adding capacity unit to existing buffers or keep adding more buffers won’t help. For example 6 of capacity 7 buffers is worse than 4 capacity 10 buffers, for about 150 total complete products in 20 iterations. Not to mention the extra cost it takes. Through the test we can find two proper settings, which are three (13,13,13,13,13) buffers and two(16,16,16,16,16) buffers.





Finally, let’s test under m+n=110, original ratio of m/n = 20/16, original buffer is single buffer with (7,6,5,5,4) capacity. And let the old version and the upgraded version both work for 20 iterations under same max wait time.We can observe the following improvement.  




In conclusion, we can see that with a little sacrifice at 8% more time consumption for the preparation process and the buffer switching, the plant can gain a 351% more profit due to higher efficiency.