CIS600/CSE691 Threaded Programming HW2 (70 points)

Due: 11:59pm, March 1, Sunday

Consider a plant, where there are m part workers (each will be implemented with a thread) whose jobs are to produce four types of parts (A, B, C, D). Each part worker will produce 4 pieces of part of any type combination, such as (3,0,1,0), (2,0,1,1), (1,1,1,1), etc., given that it takes 50, 70, 90, 110 microseconds to make each part of type A, B, C, D, respectively. For example, it takes a part worker 50*2 +110*2 microseconds (*us*) to make a (2,0,0,2) part combination. Each part worker will attempt to load the produced parts to a buffer area, which has a capacity for 6, 5, 4, 3 pieces of type A, B, C, D parts, respectively. That is, the buffer capacity is (6,5,4,3). It will take a part worker 20, 30, 40, 50 *us* to move a part of type A, B, C, D, respectively, to the buffer. Each part combination, such as (1,1,1,1) is referred to as a **load order**.

The current number of parts of each type in the buffer, such as (6,2,1,3) 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 (1,1,1,1) and the buffer state is (6,2,1,3), then the part worker can place a type B part, and a type C part to the buffer; thus, the updated load order will be (1,0,0,1) and the updated buffer state will be (6,3,2,3). The part worker will **wait** near the buffer area for the buffer space to become available to complete the load order. A wait time of 600 microseconds will make a part worker stop waiting. When this timeout event occurs, the part worker will check if the buffer allows the load order to complete. If yes, the part work will complete the load order and finish this current iteration. If the buffer cannot allow the load order to complete, the part worker will discard the remaining, unloaded parts. Assume that it takes the same move time for each type of parts to discard. Recall that it takes a part worker 20, 30, 40, 50 *us* to move a part of type A, B, C, D, respectively. The part worker then ends the current iteration. 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 five pieces of parts each time; however, the five pieces will be from exactly three types of parts, such as (1,2,2,0), (1,3,1,0), (1,1,0,3), (0,2,2,1), etc. with equal occurrence probability. For example, a product worker will not generate a order of (1,1,2,1), (3,2,0,0), 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 from the buffer is 20, 30 40, 50 *us*, respectively. Like that for part workers, partial fulfillment policy is adopted. If the current buffer state is (4,0,2,1) and a pickup order is (1,1, 0,3), then the updated buffer state will be (3,0,2,0) and the updated pickup order will be (0,1,0,2). 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, will be 80, 100, 120, 140, respectively. If the wait time reaches 1000 *us*, the product worker will stop waiting. The product worker will check if the buffer allows the pickup order to complete. If yes, the product worker will move all parts back and proceed to the product assembly work. After the required assembly time, the product worker then ends the current iteration. If the buffer cannot allow the pickup order to complete, then the product worker will give up the pickup order and **discard the parts that have been picked up**. The

time to discard is the same as the time to move as described above. The current iteration then ends. The product worker will then re-produce a brand-new pickup order.

Use a distributed locking mechanism with notifications to develop a simulation of the activity of the above-described plant. Your notifications should be 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. Your program should allow each worker to finish 5 iterations. Clearly you need to protect the shared resource, buffer, with a 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 as shown below to a file call log.txt . Note that each product worker thread will also print the total number of completed product.

Current Time:XXXXXXXus //The program starting time is 0 us
Part Worker ID: 8
Iteration: 2
Status: New Load Order
Accumulated Wait Time: 0 us
Buffer State: (5,2,3,2)
Load Order: (2,0,1,1)
Updated Buffer State: (6,2,4,3)
Updated Load Order: (1,0,0,0)

Current Time:XXXXXXXus
Product Worker ID: 5
Iteration: 3
Status: New Pickup Order
Accumulated Wait Time: 0 us
Buffer State: (2,3,4,0)
Pickup Order: (3,1,0,1)
Updated Buffer State: (0,2,4,0)
Updated Pickup Order: (1,0,0,1)
Total Completed Products: 25
…

Current Time: XXXXXXus
Part Worker ID: 3
Iteration: 2
Status: Wakeup-Notified
Accumulated Wait Time: XXXXXXus
Buffer State: (6,3,4,0)
Load Order: (2,0,0,0)
Updated Buffer State: (6,3,4,0)
Updated Pickup Order: (2,0,0,0)

Current Time:XXXXXXXus

Product Worker ID: 3
Iteration: 4
Status: Wakeup-Timeout
Accumulated Wait Time: XXXXXXus
Buffer State: (2,2,3,0)
Pickup Order: (3,1,0,1)
Updated Buffer State: (0,1,3,0)
Updated Pickup Order: (1,0,0,1)
Total Completed Products: 35

The following is a sample main function.

```
int main(){
	const int m = 20, n = 16; //m: number of Part Workers
				  //n: number of Product Workers
				  //m>n

	thread partW[m];
	thread prodW[n];
	for (int i = 0; i < n; i++){
		partW[i] = thread(PartWorker, i);
		prodW[i] = thread(ProductWorker, i);
	}
	for (int i = n; i<m; i++) {
		partW[i] = thread(PartWorker, i);
	}


	/* Join the threads to the main threads */
	for (int i = 0; i < n; i++) {
		partW[i].join();
		prodW[i].join();
	}
	for (int i = n; i<m; i++) {
		partW[i].join();
	}
	cout << "Finish!" << endl;

	return 0;
}
```