HW7 (60 points)

Due: 11:59pm, April 28 (Sunday)

Consider a plant, where there are 20 part workers whose jobs are to produce four types of parts (A, B, C, D). Each of them produces four pieces of parts of **at most** two different types each time, such as (1,3,0,0), (3,1,0,0), (2,2,0,0), (4,0,0,0) etc., while (1,1,1,1) or (2,0,1,1) is not allowed. All possible combinations will carry the same probability. In your program you have to randomly generate one combination. Each part worker will attempt to place the parts generated to a buffer area. Therefore, each combination (a, b, c, d) from a part worker is referred to as a **place request**.

In addition, there are 20 product workers whose jobs are to take the parts from the buffer area and assemble them into products. Each of them needs four pieces of parts each time; however, the four pieces will be from exactly three types of parts, such as (1,1,2,0), (2,1,0,1), (0,2,1,1), etc. with equal occurrence probability. For example, a product worker will not generate a request of (1,1,1,1), (4,0,0,0), etc. Each such combination from a product worker is referred to as a **pickup request**. In your program, you need to randomly generate one combination. Moreover, a buffer area to hold parts for part workers has a capacity of 5 type A parts, 5 type B parts, 4 type C parts, and 3 type D parts. At any moment, the numbers of parts of each type (a, b, c, d) is referred to as **buffer state**. A part worker can place parts to the buffer area up to the capacity, and wait for available space for remaining part(s) in the place request if needed. For example, if the current buffer state is (5, 4, 4, 3) and a part worker generates a (1,3,0,0) place request, then the buffer state will be updated to (5, 5,4, 3) and the part worker's place request will be updated to (1,2,0,0) and the part worker has to wait until the space becomes available. Similarly, a product worker can pick up those available parts from the buffer area and wait for the remaining part(s). For example, if the current buffer state is (2,1,0,1) and a product worker generates a pickup request of (2,0,1,1), then the buffer state becomes (0,1,0,0) and the product worker has to wait with a updated pickup request of (0,0,1,0). Assume that a part worker or product worker will take 2-6 (randomly) microsections to generate a new place request or a pickup request. Also assume that each blocked request, when becoming lock owner, will take 1 microsecond to become active.

In your simulation code, you should intend to allow each part worker and each product worker to complete 6 requests. (Thus, there will be 6 iterations of generating place requests and 6 iterations of generating pickup requests.) Your goal is to ensure that your code can complete the execution unless the generated requests make it impossible.

When a part worker generates a place request or is awakened to continue a place-request, you need to print ID, the current buffer state and the place request, and the updated buffer state and updated place-request. (See format of output at the end of this write-up.) If the buffer has sufficient space for the place request, then the updated place request will be (0,0,0,0). Similarly, when a product worker generates a pickup request or is awakened to continue a pickup request, you need to print the current buffer state and the pickup request, and the updated buffer state and updated pickup request. If the buffer has all the requested parts for the pickup request, then the updated place-request will be (0,0,0,0). (See format of output at the end of this write-up.)

The following are examples for a place request and a pickup request.
(You need to make sure your output follows the same format.)
Note that there is a blank line between 2 requests.
There is a space after symbol ':' .

In addition, if a thread detects a deadlock (i.e., no longer possible for any Part Worked to add parts to buffer and also no longer possible for any Product Worker to pick up parts from buffer), this thread will abort the current iteration (as if this iteration is completed). In this case, "Deadlock Detected", "Aborted Iteration: x" will be printed. (See the third thread printout below.)

All workers will be treated equally. Especially, workers blocked by locks or in sleep should be fairly treated; that is, they should have equal chance to become the next lock owner.

For grading convenience, I have introduced const variables m and n in main function to represent number of Part Workers and number of Product Workers, where m>=n.

```
Part Worker ID: 8
Iteration: 2
Buffer State: (5,2,3,2)
Place Request: (2,0,2,0)
Updated Buffer State: (5,2,4,2)
Updated Place Request: (2,0,1,0)

Product Worker ID: 5
Iteration: 4
Buffer State: (2,3,4,0)
Pickup Request: (3,1,0,1)
Updated Buffer State: (0,2,4,0)
Updated Pickup Request: (1,0,0,1)

…

Part Worker ID: 3
Iteration: 4
Buffer State: (6,3,4,0)
Place Request: (2,0,0,0)
Updated Buffer State: (6,3,4,0)
Updated Pickup Request: (2,0,0,0)
Deadlock Detected
Aborted Iteration: 4
```

The following is a sample main function.

```
int main(){
	const int m = 20, n = 20; //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;

        getchar();
        getchar();
        return 0;
}
```