README

- For **default 5-philosopher cycle** drinking problem, enter parameters like: **./p -s 20**
- For **configuration specified** drinking problem, enter parameters like: **./p -s 20 -** , and input the name of configuration file.

Running screenshot is as follows:
default 5-philosopher cycle drinking problem:

```
[zji@cycle1 hw2]$ ./p -s 2
Default Drinking Problem Begin.
philosopher 5 thinking
philosopher 5 drinking
philosopher 4 thinking
philosopher 1 thinking
philosopher 3 thinking
philosopher 2 thinking
philosopher 4 drinking
philosopher 1 drinking
philosopher 3 drinking
philosopher 1 thinking
philosopher 4 thinking
philosopher 5 thinking
philosopher 5 drinking
philosopher 2 drinking
philosopher 3 thinking
philosopher 4 drinking
philosopher 1 drinking
philosopher 2 thinking
philosopher 3 drinking
philosopher 2 drinking
Default Drinking Problem Done.
```

configuration specified drinking problem:

```
[zji@cycle1 hw2]$ ./p -s 2 -
Please enter the filename.
configuration.txt
Drinking Problem Begin.
philosopher 1 thinking
philosopher 1 drinking
philosopher 3 thinking
philosopher 3 drinking
philosopher 5 thinking
philosopher 4 thinking
philosopher 5 drinking
philosopher 1 thinking
philosopher 2 thinking
philosopher 3 thinking
philosopher 4 drinking
philosopher 1 drinking
philosopher 3 drinking
philosopher 5 thinking
philosopher 2 drinking
philosopher 5 drinking
philosopher 4 thinking
philosopher 4 drinking
philosopher 2 thinking
philosopher 2 drinking
Drinking Done.
```

Introduction

The problem of the dining philosophers is first proposed by Edsger Dijkstra. Chandy and Misra proposed a paradigm, the drinking philosophers problem, which is a generalization of the classical dining philosophers problem.

The description of drinking problem is detailed in the Chandy and Misra' paper, hence I won't put more words about it. It is necessary to make it more compact and concrete before we implement it in our program.

Specifically, philosophers (i.e. threads) are placed at the vertices of a finite undirected graph G with one philosopher at each vertex. A philosopher has two states: thinking and drinking. Associated with each edge in G is a bottle. A thirsty philosopher can only drink from bottles associated with his adjacent edges. On holding all needed bottles, a philosopher starts drinking in finite time. On entering the drinking state a philosopher remains that state for a finite period, after which he continues to think. A philosopher may be in the thinking state for an random period of time. I adopt the state of fork in dining problem, the bottle in drinking problem has one property called dirty. A bottle is dirty when it is drunk by someone and it will be clean when someone gets it and before drinking.

Implementation

The algorithm is the following:

1. Initially, for every pair of philosophers contending for a shared bottle, create a bottle and assign it to the philosopher with lower ID. Each bottle can be dirty or not. Initially, all bottles are dirty.
2. A philosopher finishes one drinking session only after he drinks from the bottles associated with all his adjacent edges.
3. When a philosopher with a bottle receives a request message, he keeps the bottle if it is clean, but gives it up when it is dirty. If a philosopher sends the bottle over, he cleans the bottle before doing so.
4. After a philosopher is done drinking, all his bottles get dirty. If another philosopher had previously requested one of the bottles, the philosopher that has just finished drinking cleans the fork and sends it.

In this implementation, philosophers, i.e. threads, need to communicate with each other to request the bottles. In order to implement this, we must use std::condition_variable, which is a synchronization primitive that enables the blocking of one or more threads until another thread notifies it. The following class, syn_controller, has a condition variable and a mutex and provides two methods: one that waits for the condition variable and blocking the calling thread, the other notifies the condition variable, unblocking all the threads that are waiting for a signal.

```
class syn_controller {
    std::mutex mutex;
    std::condition_variable g_state_flag;

public:
    void wait() {
        std::unique_lock<std::mutex> lock(mutex);
        g_state_flag.wait(lock);
    }

    void notify_all() {
        g_state_flag.notify_all();
    }
};
```

The bottle is a class with an identifier, an owner, a flag to indicate whether it is dirty or not, and a syn_controller which enables owners to request used bottles. There are two methods here:

- request(). It enables a philosopher to request the bottle. On one hand, if the philosopher who requests the bottle is not the owner of the bottle, he will first check whether the bottle is dirty or not. If the bottle is dirty, it stands for that the bottle is just used by someone else, so philosopher that request this bottle will set it to clean and the bottle's owner will changed to current philosopher who is using it. If the bottle is clean, the philosopher has to wait for it to become dirty and will be blocked. On the other hand, if the request philosopher is exactly the owner of the bottle, he will also change it to clean because a philosopher will only finish a drinking session when he

drinks from all the bottles he shares with his neighbors, hence if we do nothing in this case that the request philosopher is the owner of the bottle, it is possible that the bottle will be used by another philosopher who is requesting it and sees it is dirty correctly.

- finish_drinking() enables the philosopher who has finished drinking and notifies other philosopher that is waiting for this bottle.

```
class bottle {
    int id;
    int owner;
    bool dirty;
    std::mutex mutex;
    syn_controller signal;
public:
    bottle(int const bottle_id, int const owner_id) :
            id(bottle_id), owner(owner_id), dirty(true) {

    }

    void request(int const ownerId) {

        while (owner != ownerId) {
            if (dirty) {
                std::lock_guard<std::mutex> lk(mutex);
                dirty = false;
                this->owner = ownerId;
            } else {
                signal.wait();
            }
        }
        dirty = false;
    }

    void finish_drinking() {
        std::lock_guard<std::mutex> lk(mutex);
        dirty = true;
        signal.notify_all();
    }
};
```

The philosopher class has an identifier, the drinking session he wants to finish, the bottles he can request, and a syn_controller which is responsible for launching all threads at the same time.

The drink() function simulates the philosopher's drinking: a philosopher will send request message to all the bottles he can ask for. Once he holds all the bottles, it's time to drink. After finishing drinking, the philosopher will call finish_drinking() to let other waiting philosophers know.

```
void drink() {
    for (auto b : bottles) {
        b->request(id);
    }

    print(" drinking");
    int t;
    random_r(rand_state, &t);
    usleep(t%1000000);
    for (auto b : bottles) {
        b->finish_drinking();
    }
}
```

Start_session() is the thread function which only starts thinking and drinking after a signal is received. A condition variable from the syn_controller is used to implement this.

```
void start_session() {
    controller.wait();
    do {
        think();
        drink();
        session_count += 1;
    } while (session_count < drinking_session);
}
```

Problems

During the implementation of drinking problem, I meet several problems. Some of them is hard to deal with and enable me to think deeper behind it. Therefore, I think it is necessary to write it down.

- Lost wakeup. The phenomenon of the lost wakeup is that the sender sends its notification before the receiver gets to its wait state. The consequence is that the notification is lost. The C++ standard describes condition variables as a simultaneous synchronisation mechanism: "The condition_variable class is a synchronisation primitive that can be used to block a thread, or multiple threads at the same time, ...". So the notification gets lost, and the receiver is waiting and waiting and ... . What I do to prevent lost wakeup is that I make the main thread to sleep for a while before notifies all philosophers threads. It works well actually.
- Memory leak. It is usually caused by unrelease of allocated memory and makes program do some undefined behaviors. At first, I create the object of philosopher as pointers but forget to free it before the program ends, which indeed make threads behave abnormally.