Synchronization primitives

Synchronization primitives enforce coordination between the threads. Here are the popular ones:

1. Mutex (Mutual Exclusion Lock)

- Only **one thread** can hold the lock at a time.
- Other threads must **wait** until it's released.
- Used when you need **exclusive access** to a resource.

Example

```
// Only one thread can enter here shared++;
std::mutex m; m.lock();
// Others can now proceed
m.unlock();
```

Let's say there's a hall full of workers, and many workers need to coordinate together to get work done, but they keep arguing and have a big heated discussion. A chicken comes along and enforces a rule:

> "Whoever has me, has the right to talk. If you do not have the me, then you cannot speak."

The chicken here makes sure that whoever wants to talk needs to wait. The person that acquired the chicken must pass that along the next worker.

The chicken here is the **mutex**.

---

2. Semaphore

A counter that controls access to a resource pool. Used for limiting concurrency (e.g., max 5 threads accessing a database).

Example

```
std::counting_semaphore<5> sem(5);
sem.acquire();    // decrease count, wait if 0
use_resource();
sem.release();    // increase count`
```

If a **mutex** is one chicken, a **semaphore** is N-number of chickens in the hall. Now, multiple people can debate simultaneously.

---

3. Spinlock

- A lock where threads **keep spinning (busy waiting)** until the lock is free.
- Faster than mutex for **short waits** because it avoids *overhead waiting time* of context switching, but wastes CPU cycles.
- Good for **low-latency critical sections**.

Example

```
std::atomic_flag lock = ATOMIC_FLAG_INIT;

void critical_section() {

  while (lock.test_and_set(std::memory_order_acquire)) {
      // Busy wait (spin)
  }

// —— critical section ——
shared++;
// ————————————————————————

lock.clear(std::memory_order_release);
}
```

Analogy

There was an old shabby hotel that wants can only afford for one customer to go in. The chicken enforces the rule that only one person to acquire the lobby of the hotel with a rotating door. If one customer saw that someone is in the lobby,

> "Hey! Get out, only one customer only. Go spin in the door in a loop while you wait."

They'll spend energy spinning around the rotating door; they wastes their time and energy for a bit to wait, but makes it up for short cardio.

That rotating door is the **spinlock**.

---

4. Condition Variable

Lets threads **sleep until a condition is met**, instead of constantly checking. Used with a mutex.

Example

One example is producer-consumer queue (consumer waits until producer signals queue is not empty).

```
std::condition_variable cv;
std::mutex m;
std::queue<int> q;  // consumer waits
std::unique_lock<std::mutex> lock(m);

cv.wait(lock, [] { return !q.empty(); }); // this checks for a bool
process(q)
```

Analogy

When workers and a customer needs to coordinate with each other, but workers keeps trying to work on an empty order - giving out unusual results.

The chicken comes and finds a cow to send a signal if the customer finishes writing their order.

> The cow knows only truths and propositions - whether something is true or is false. You shall wait for the cow to give a signal and it will lead you to light.
>
> - The Chicken.

Once the cow notices that there's tasks on the queue, it waves a flag and workers begin to scramble for it.

---

5. Barrier

- Makes a group of threads wait until **all reach the same point**.
- Useful in parallel algorithms where threads must sync at stages.

Example

```cpp
#include <barrier>
#include <iostream>

std::barrier sync_point(3); // wait for 3 threads

void worker(int id) {
  std::cout << "Thread " << id << " finished step 1\n";
  sync_point.arrive_and_wait();  // all wait here
  std::cout << "Thread " << id << " continues to step 2\n";
}
```

Analogy

The workers wants a race, but many of them keep cheating and go first ahead of others.

> "No fair! I want the chicken to signal only once everybody is ready!"
>
> - One of the workers

The workers called the chicken, and the chicken counts down as each worker signals they are ready.

The chicken blows its whistle once they counted to zero, and off the workers go.

---

6. Read-Write Lock (Shared-Exclusive Lock)

- Multiple readers can access simultaneously.
- Writers get **exclusive** access.
- Good when reads are much more frequent than writes.

Example

```
#include <shared_mutex>
#include <thread>
#include <iostream>

std::shared_mutex rwlock;
int data = 0;

void reader(int id) {
  std::shared_lock lock(rwlock); // shared (read) lock
 std::cout << "Reader " << id << " sees data = " << data << "\n";
}

void writer(int id) {
   std::unique_lock lock(rwlock); // exclusive (write) lock
   data++;
   std::cout << "Writer " << id << " updated data = " << data <<
"\n";
}
```

Analogy

Multiple stressed students in the school library are reading books about their favorite manga. The writer spots this and says:

> Oh! I really want to add these new additions and update the old editions.

However, she values the reading experience of the students. They'll get confused about the sudden changes in person, so she cannot just take away the current books they are reading ...

... So, the chicken was asked by the author / writer of those books to close the library early so she can place the changes secretly.

When the students come back to see, they were the delighted to see the changes!

---

7. Futures & Promises

- High-level primitive for synchronizing results of asynchronous tasks.
- A **promise** sets a value, a **future** waits for it.

Example

```cpp
#include <future>
#include <iostream>

string compute() {
 return "Chocolates";
}

int main() {
 std::promise<string> prom;
 std::future<string> fut = prom.get_future();

 std::thread producer([&] {
      prom.set_value(compute());
 });

 std::cout << "Waiting for result...\n";
 string result = fut.get(); // waits until producer sets it
 std::cout << "Got: " << result << "\n";

 producer.join();
}
```

Analogy

A worker says to his lover that she will have chocolates in the future. The lover remembers this happily and goes on its her days doing her work.

Once she gets stressed and needed a break, she waits for her lover to give her those **future** chocolate.

However, she gets frustrated as it takes a while. The chicken reminded her to wait patiently.

> "Trust the worker, he promises you a future after all."
>
> - The Chicken.

Then, the worker returns to the space she owns with a basket full of them. Both the worker and lover is happy.

8. Atomics and Memory Barriers

- Atomics, and its low level version named memory barriers (or fences) enforces the rule that any update within one cache is updated in the result.

Example

```
std::atomic<int> A = 1 // guranteed to be consistent across memory caches.
int B = 0 // no gurantee

increment_thread([thread1, thread2, thread3], A)
increment_thread([thread4, thread5, thread6], B)
```

Analogy

The record keepers are assigned to keep track of what the workers said so far, but they often have conflicting notes across rooms.

Then, the chicken calls a sheep for each data that the record keepers need to keep.

> "Baa! We'll keep track for you. We think similarly a herd, so we keep memorize as one big fluffy sheep!"

With the chicken, the cow, and the sheep in hand. The town hall can be united as one.

References:
multithreading - What is a mutex? - Stack Overflow