CS4414 Recitation 13
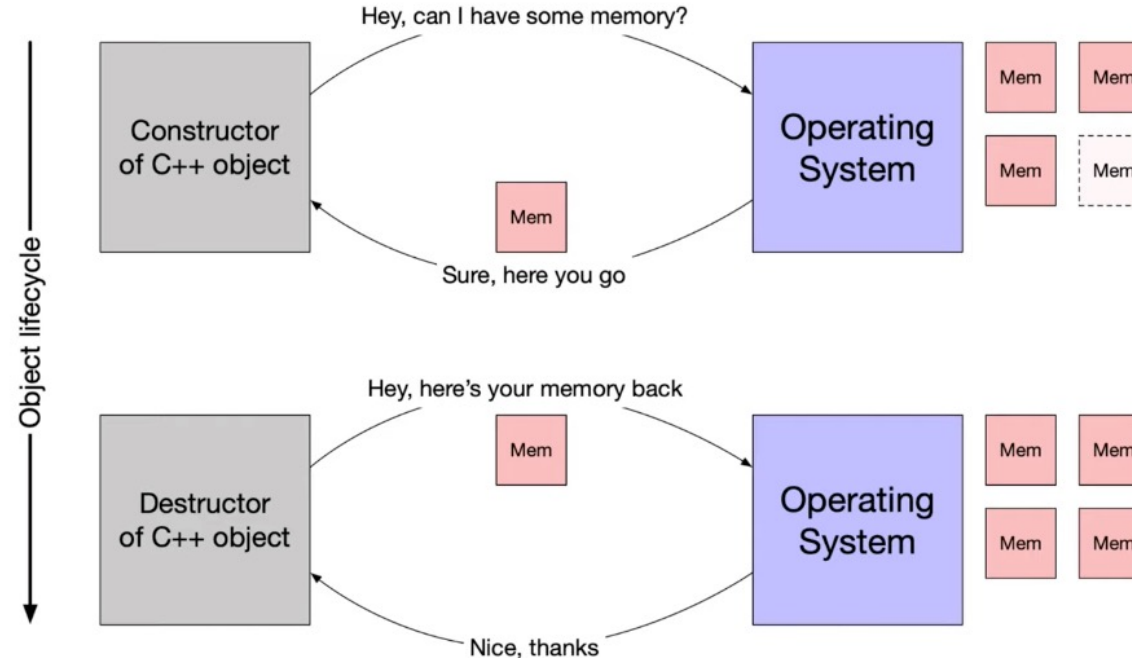## Multithreading and Synchronization III

11/19/2021

Alicia Yang

Recap

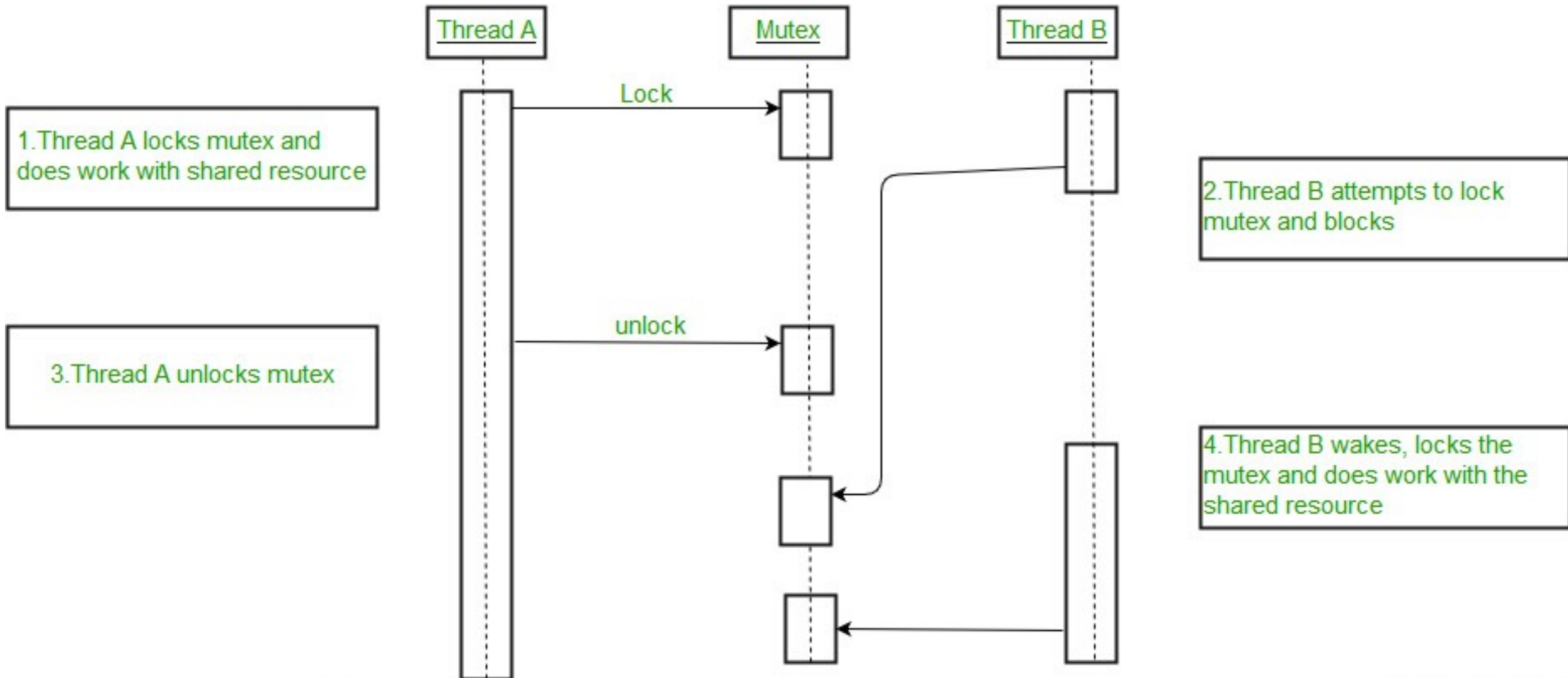# RAII (Resource Acquisition is initialization)

- Resource acquisition must succeed for initialization to succeed:

  - In RAII, holding a resource is a class invariant is tied to object lifetime: resource allocation is done during object creation, by the constructor; while resource deallocation is done during object destruction, by the destructor.

- If there are no object leaks, there are no resource leaks.

  - The resource is guaranteed to be held between when initialization finishes and finalization starts, and to be held only when the object is alive.

RAII (Resource Acquisition is initialization)

- RAII

  - When acquire resources in a constructor, also need to release them in the corresponding destructor

  - Resources:

    - Heap memory,

    - files,

    - sockets,

    - mutexes

Recap Mutex and Lock in C++

Thread A

Mutex

Thread B

Lock

1. Thread A locks mutex and does work with shared resource

2. Thread B attempts to lock mutex and blocks

unlock

3. Thread A unlocks mutex

4. Thread B wakes, locks the mutex and does work with the shared resource

Mutex and RAII locks

- std::scoped_lock

- std::unique_lock

- std::shared_lock

```
std::mutex my_mutex;
{
    std::scoped_lock<std::mutex> lck(my_mutex);
    … …
}
```

```
{
    std::unique_lock<std::mutex> lck(my_mutex);
    … …
}
```

```
{
    std::shared_lock<std::mutex> lck(my_mutex);
    … …
}
```

What is the difference between unique_lock and scoped_lock?

Multithreading

- Threads management
  - Launching threads
  - Threads completion
- Synchronization
  - Race condition
  - Atomic
  - Mutex
  - Locks
  - Condition variables
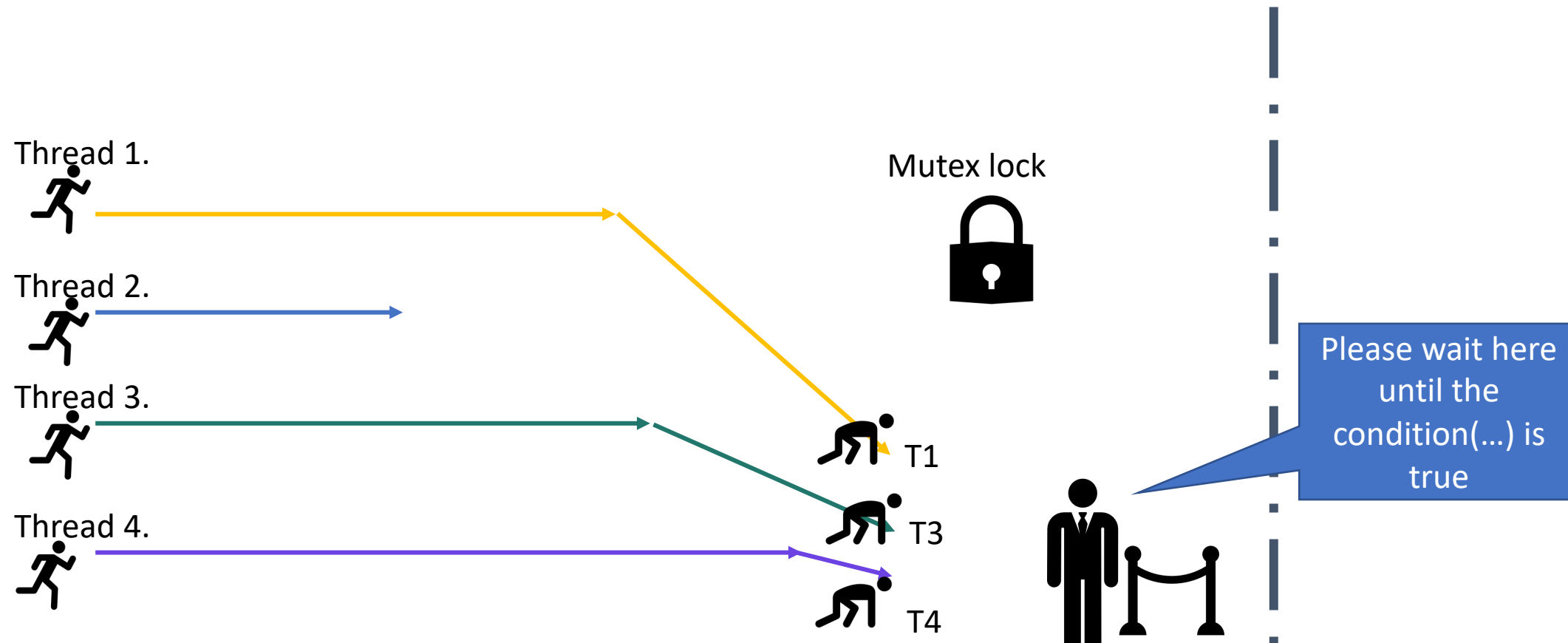 - Futures and promises(async)

Condition Variable

Suppose a thread needs to wait for some other threads to do something for it, how would you encode this into the program?

Condition Variable

- Two main purpose of condition variable
  - Notify other threads
  - Waiting for some conditions that other thread can change

Thread 1.

Thread 2.

Thread 3.

Thread 4.

Mutex lock

T1

T3

T4

Please wait here until the condition(...) is true

Condition Variable   --- lock

1. Need mutex to use condition variable

    Two roles

    - Waiting threads: first acquire the lock, then wait() if condition not satisfied

    - Notifying threads: thread make the changes that can allow other thread's wait condition to true and move on.

Condition Variable   --- wait

1. Need mutex to use condition variable

2. Condition Variable allows running threads to **wait** on some conditions and once the threads wake up

   - Atomically acquire the lock and check the condition

     - If the condition is satisfied, then it will continue the program

     - If not satisfied, it waits by releasing the lock, and goes back to waiting

Condition Variable   --- wait

- Two types of wait functions for condition variable

- Unconditional wait()   and   predicate wait()

```
{

    std::unique_lock<std::mutex> lck(mtx);
    cv.wait(lck);

    ……

}
```

```
bool pred() {return current_balance!=0;}

{

    std::unique_lock<std::mutex> lck(mtx);

    cv.wait(lck, pred);

    ……

}
```

Condition Variable   --- wait

- Two types of wait functions for condition variable

To avoid the affect of spurious wake ups, always use predicate wait() !

- Unconditional wait()   and   predicate wait()

```
{

    std::unique_lock<std::mutex> lck(mtx);
    cv.wait(lck);

    ……

}
```

```
bool pred() {return current_balance!=0;}

{

    std::unique_lock<std::mutex> lck(mtx);

    cv.wait(lck, pred);

    ……

}
```

Condition Variable   --- wait

- When a thread calls the member function wait() on a condition variable
  - The execution of the current thread (which currently has the locked's mutex) is blocked until notified.
  - When the thread is blocked, the function automatically calls unlock(), allowing other threads to acquire the lock and continue.

- The wait function performs three atomic operations:
  - The initial unlocking of mutex and simultaneous entry into the waiting state.
  - The unblocking of the waiting state.
  - The locking of mutex before returning.

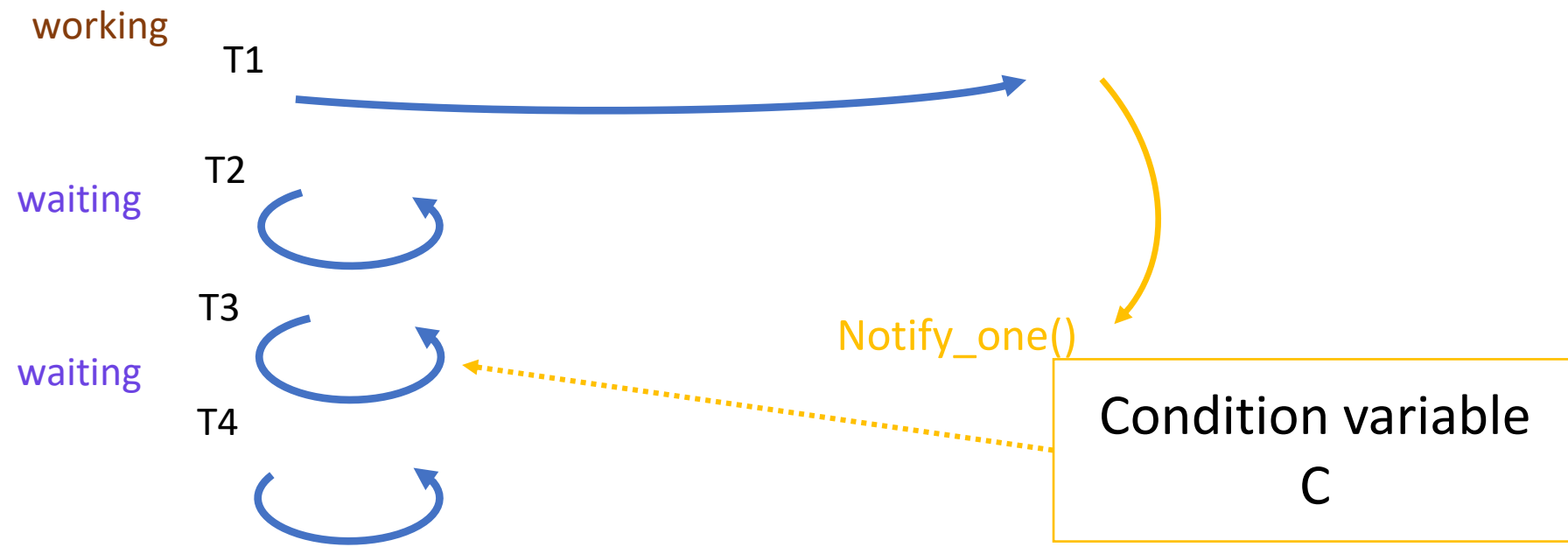# Condition Variable   --- notify

1. Need mutex to use condition variable

2. Condition Variable allows running threads to wait on some conditions

3. The waiting thread(s) is notified by working thread using:

   - notify_one();

   - notify_all();

Condition Variable

- The waiting thread is notified by working thread using:
  - notify_one():
    - Unblocks one of the threads currently waiting for this condition.
    - If no threads are waiting, the function does nothing.
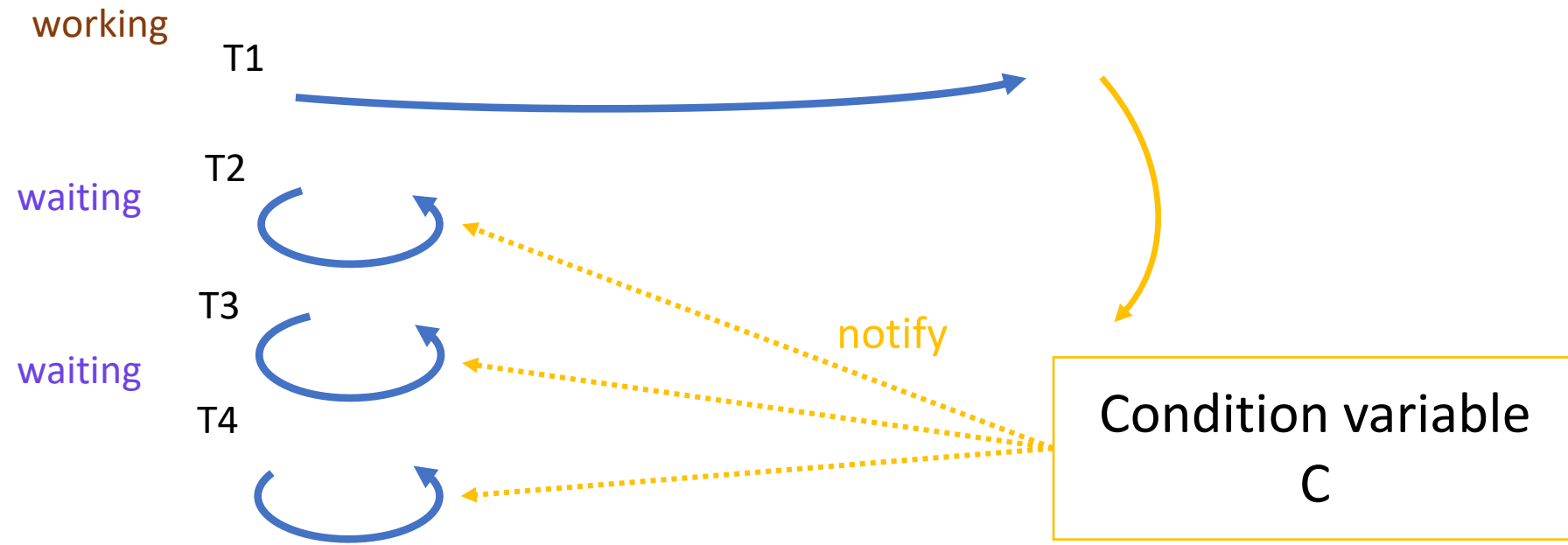   - If more than one, it is unspecified which of the threads is selected.

working

T1

waiting

T2

waiting

T3

T4

Notify_one()

Condition variable
C

Condition Variable

- The waiting thread is notified by working thread using:
  - notify_all():
    - Unblocks all threads currently waiting for this condition.

Condition Variable

1. Each thread first acquire the mutex lock

2. Then check the condition in wait()

3. Waiting thread(s) is notified by working thread

4. When thread(s) waiting at the condition variable gets notified,

   - it first try to acquire the lock of mutex

   - Check the condition, the thread will not go further until the condition is true:

     - if it is true, then go further;

     - if it is not, it will again wait for the condition variable

Demo

Promises and futures

- What are promises and futures?

- How to use them in C++?

Futures and Promises

- Why future and promise?

  - A way to pass values between threads without synchronization, such as locking a mutex.

- When to use?

  - When some operations produce results take some time, or do not need to be executed in a particular order

  - Reading or writing data:

    - Reading large files from disks

    - Web service calls over HTTP

    - Reading data from a Socket

    - Database queries

    - Responsive user interface

    - Distributed systems

  - Run a program(function) asynchronously

Promises

- Class template object: a facility to store a value or an exception that is later acquired asynchronously via a std::future object   std::promise<**T**> my_promise ;

- Promise object has an associated future object, which is automatically instantiated when a promise is created.   std::future<**T**> my_future = my_promise.get_future() ;

- The constructed future will only be valid when the promise fills in the data

- Promise object guarantees that the future object will return the result when the set_value function is called on it by the computing thread

Futures

- Class template object: provides a mechanism to access the result of asynchronous

  operations   std::future<T> my_future = ...;

- Future is a read-only object containing data

  - The data may not be available or computed in the present

  - The data is promised to be available in the future

- get() method is the main purpose of the future object

  - Calling get() will block the current thread until the data is available

  - get() will either returns a value or throws an exception.

How do futures and promises work?

1. Construct a promise object

2. Get the future object from the promise

3. Move the promise to another thread/function.

4. When the function has completed

   1. Place the return value or exception in the promise
   2. The future becomes valid or available

5. Call get() on the future object to retrieve the data

```cpp
std::promise<int> pObj;

std::future<int> fObj=pObj.get_future();

std::thread thread_A(fun,std::move(pObj))




pObj.set_value(42);


fObj.get()
```

Demo

What happens if the value cannot be produced as promised?

- The computing thread can set an exception on the promise object by calling the set_exception method.

- Then, future<T>::get() will throw the same exception

- If the computing thread forgets to set a value or an exception and the promise object is destroyed, a default exception will be stored in the future object (broken_promise)

```cpp
std::promise<int> p;
    std::future<int> f = p.get_future();
    std::thread t([&p]{
        try {
        throw std::runtime_error("Example");
        } catch(...) {
        p.set_exception(std::current_exception());
        }
    });
     … …
}
```

Why do we separate the future and promise classes?

- Encapsulate the two sets of functionalities
    - Promise: used by the function to compute the value, and store the value/exception in the future.

        --- set_value() method
    - Future: used to retrieve the value being computed

        --- get() method
- Works well when different threads have different tasks

Async

- Abstraction of calling a function in a different thread

- The async function will be executed in a separate thread. Main program does not wait for the async function to complete

- std::async automatically sets up the Future/Promise

- Return the future object right away

- At some pointer later when the function complete, the returned future will be valid

Async

```cpp
#include <iostream>
#include <future>
bool is_prime(int x)
{
    … …
    Return true;
}

int main()
{
    std::future<bool> fut = std::async(is_prime,321);

    bool ret = fut.get(); // waits for is_prime to return

    return 0;
}
```

Futures and promises

- Problem:
  - No way to notify the other thread when finished
  - Get() method is blocking
  - Non-blocking
    - Alternative 1. use wait_for(std::chrono::second(0)) on the future
    - Alternative 2. use concurrency extension in c++20

```
std::future_status status;
while (status != std::future_status::ready) {
        status =
future.wait_for(std::chrono::seconds(0));
        if (status == std::future_status::ready)
        {
            std::cout << "ready!\n";
        }
    }
```

```
auto f = std::async(std::launch::async, func);
while (!f.is_ready()) {
            // … …
}
auto result = f.get();
```

More on wordcount example

- Recap and continue with previous recitation

Where to find the resources?

- Condition Variable:

  - https://www.cplusplus.com/reference/condition_variable/condition_variable/wait/

- Future and promise:

  - https://www.cplusplus.com/reference/future/async/

  - https://en.cppreference.com/w/cpp/thread/future/wait_for