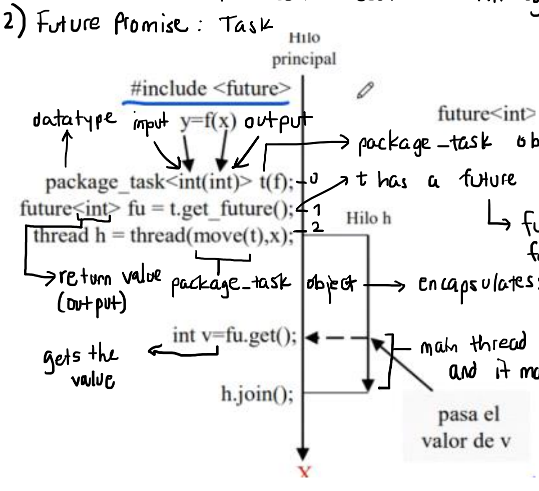
C++11 Multithread: Promises

jueves, 12 de mayo de 2022 07:03 a. m.

→ Since the thread receives only void functions, we need ways to communicate: can be through pointers, or using promises

2) Future Promise: Task



```
#include <future>
y=f(x)
package_task<int(int)> t(f);
future<int> fu = t.get_future();
thread h = thread(move(t),x);
int v=fu.get();
h.join();
```

datatype · input · output → package_task object t

0 / 1 / 2

→ t has a future
↳ future waits for the promise

→ return value (output)

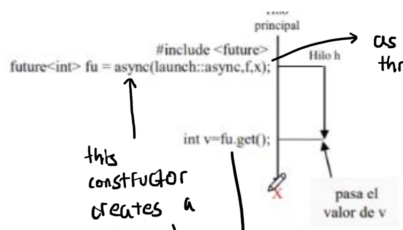
package_task object → encapsulates: function f and promise

gets the value

main thread receives and it may need to do some code, at the end it finishes all threads

package_task allows us to use functions that return something with threads

3) Future promise: Async



```
#include <future>
future<int> fu = async(launch::async,f,x);
int v=fu.get();
```

→ as soon as the object is created, the thread starts to execute its function.

this constructor creates a thread

→ thread dies since async has it

→ These 3 were ways to communicate with 'return' value threads, which is a way of synchronization, since there is a waiting process.

→ Each thread must have its own future, since it handles all return values, and these are the responsability of each thread to calculate.

```
for (future<int> &f : vt){
    cout<< f.get() <<endl;
}
```

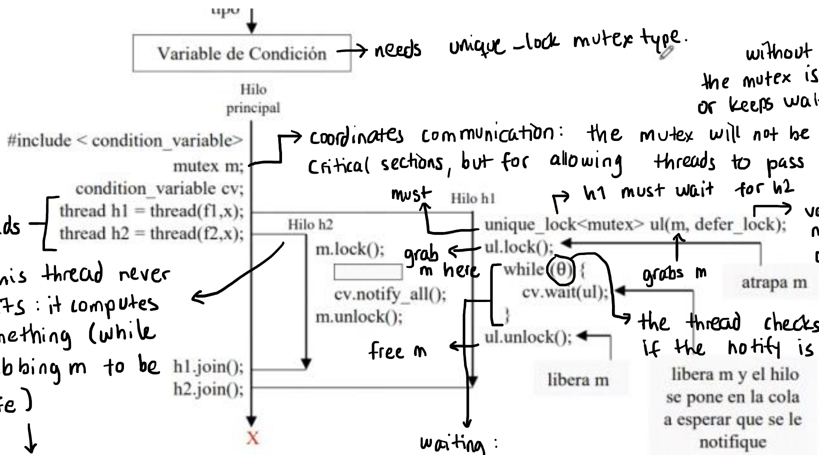
→ C++ for each loop

→ Vt is a vector of futures, in this case.

∞ - ∞ , ∞ times:

↳ many threads communicate to many others, many times

↳ we can do this using Condition Variable



```
#include < condition_variable>
mutex m;
condition_variable cv;
thread h1 = thread(f1,x);
thread h2 = thread(f2,x);
h1.join();
h2.join();

m.lock();
cv.notify_all();
m.unlock();

unique_lock<mutex> ul(m, defer_lock);
ul.lock();
while(θ){
    cv.wait(ul);
}
ul.unlock();
```

→ needs unique_lock mutex type.

→ coordinates communication: the mutex will not be for critical sections, but for allowing threads to pass values around.

2 threads

must

→ h1 must wait for h2 to do something

→ value used for not grabbing m at this moment, and thus can continue

without this, the mutex is grabbed or keeps waiting

grab m here

grabs m

free m

→ the thread checks if the notify is for that thread

this thread never waits: it computes something (while grabbing m to be safe)

↓

when it is done, it sends a msg to all threads: the θ

all waiting th awake

waiting: cv. wait(ul);

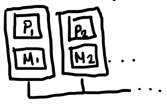
waits for one thread to respond or communicate → waits to grab the mutex

when it is done, all waiting th
it sends a msg ↗ awake
to all threads: the θ
condition decites who
answers h2 changes it

h1: global, thus to check
```
  m.lock(); → must be blocked
    while (x) {
      cv.wait(ul);
    }       ↳ thread sleeps
            and frees m.
  m.unlock();
  // thread awakes and
  frees the m ↘
              continues the
              function
```

cv. wait(ul);
waits for one thread to
respond or communicate → waits to grab the mutex
↳ this can be put in many places
↳ when this happens, the OS takes this thread out of the processor, out of ready queue and puts it to sleep until some thread responds → frees mutex for other to respond
→ θ must be a condition to verify that the thread that verifies it responds to h2
→ when the thread responds, it enters the while loop
→ at cv.wait(ul); the thread frees the mutex for other thread to grab it, since notify was not for that thread.
→ we lock and unlock θ since θ is a global variable

global variables → allows us to simulate shared memory: all threads have access
local variables → creates a distributed memory simulation, since threads only access their variables



cv. notify_all() : notifies all threads that use cv condition and that are waiting

In program: ↱ start its func execution
thread ∅ = Start
[: waits for m
[m : grabs mutex
[→m : frees mutex

↳ Thus communication happens many times, to many threads.