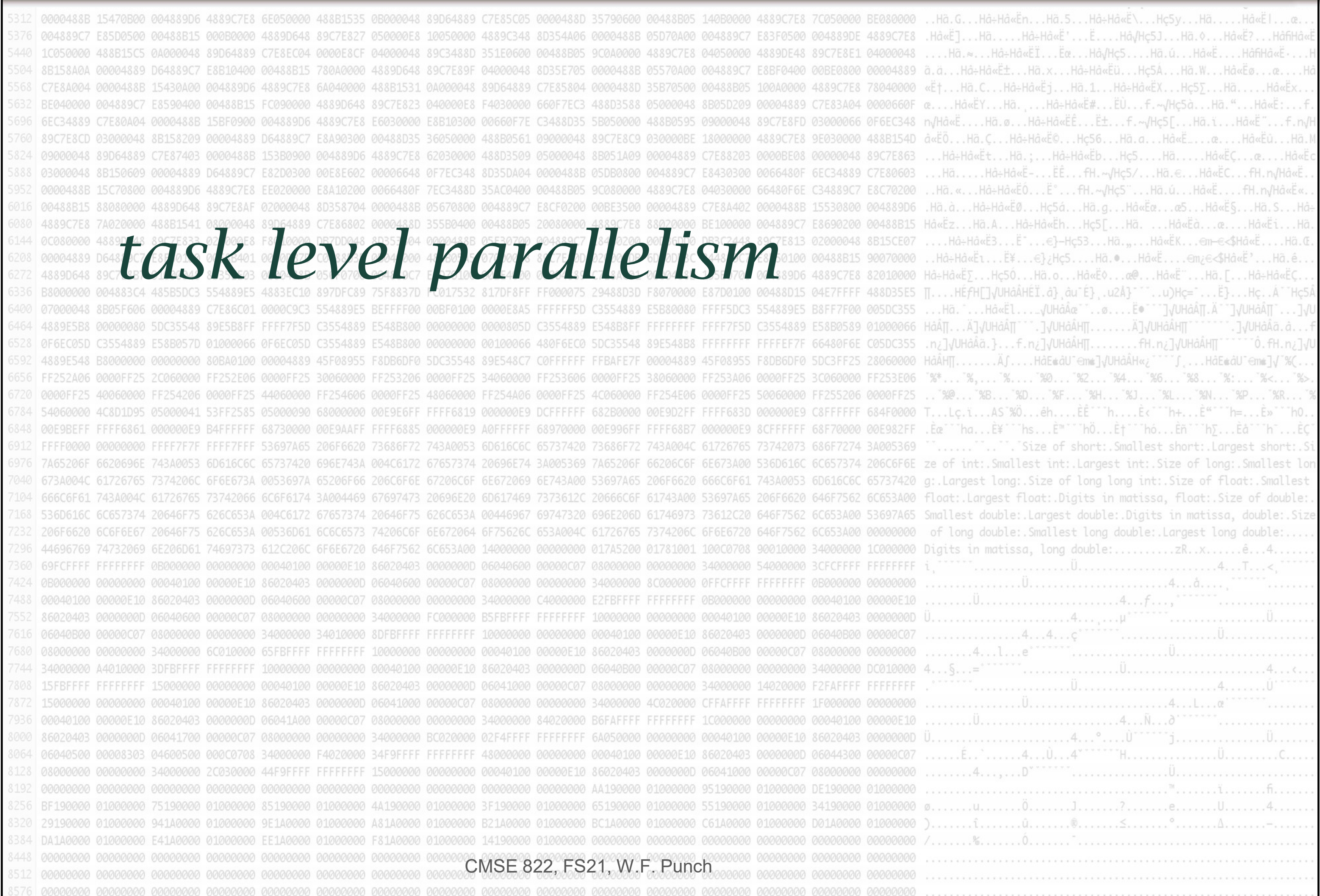


task level parallelism

async and futures

higher level interface

how to return results and handle exceptions between threads

C++11 introduces two concepts:

- async, invoke a task which can be called upon in the future to return a value
- future, a data structure to represent that future value returned by an async call

step up from thread to a task

These next elements step up (a bit) from the model of starting a thread, joining or detaching, etc.

Just start some task and allow me to get the result.

did you notice, no returns

Did you notice that there were no returns on the results of a thread function.

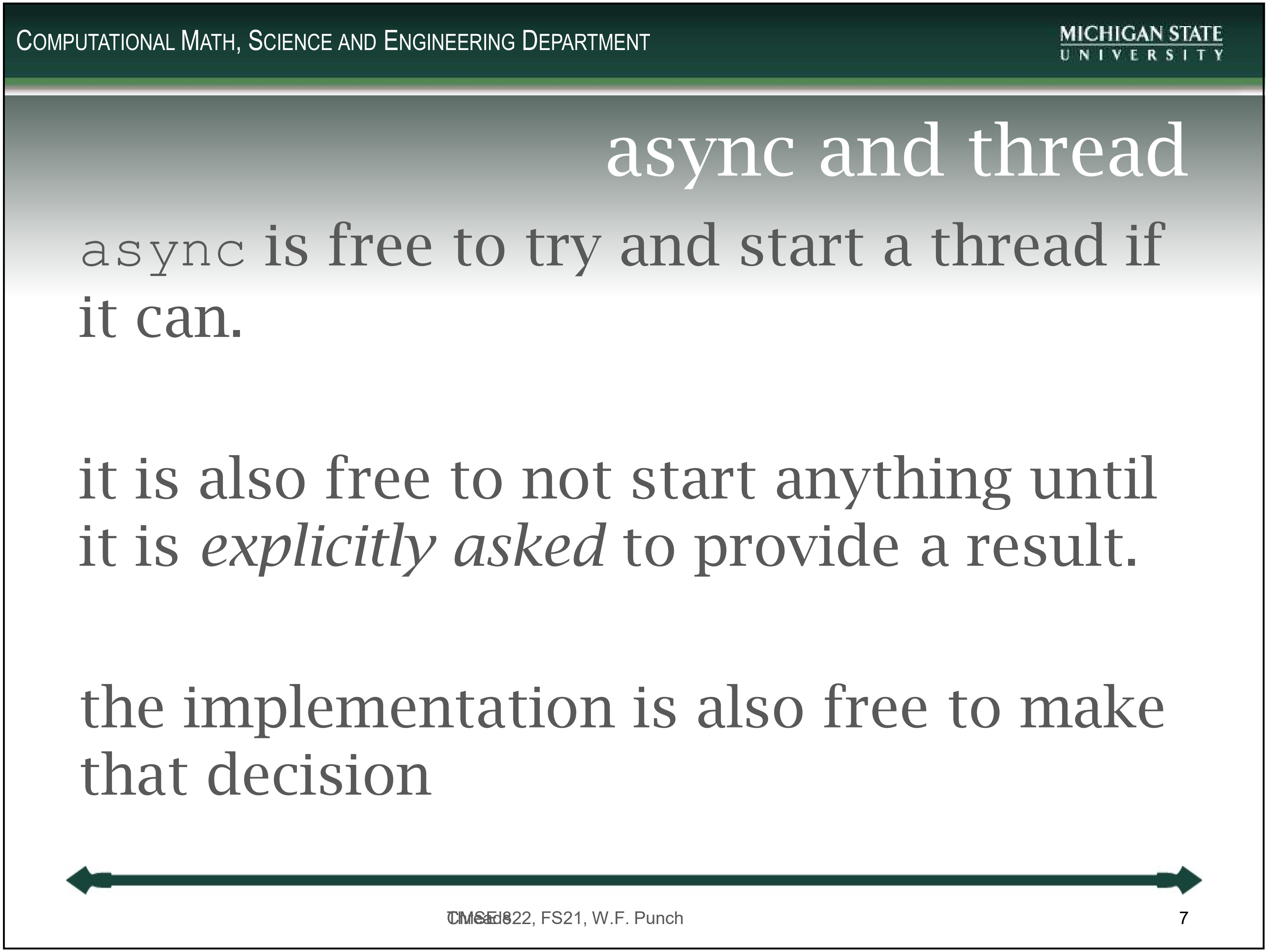
How can we connect the result of a thread to the calling program.

async

```
#include<thread>
async(callable_object)
```

potentially starts a new thread that runs `callable_object`.

potentially is a key word here



async and thread

`async` is free to try and start a thread if it can.

it is also free to not start anything until it is *explicitly asked* to provide a result.

the implementation is also free to make that decision

what?

Yes, a little hard to wrap your head around, but `async` does not have to start a thread (may not be able to start a thread either).

However, the code will run under these conditions (just serially).

force the issue

scoped enumeration can be used to resolve the issue:

- `std::launch::async`
  - Start right now!
  - throw error (at launch) if it cannot
- `std::launch::deferred`
  - wait until I have to
  - lazy evaluation
  - if it cannot start a thread, do it sequentially

a future

A ***future*** is a data structure that holds the result (whether it is actually available or not) of an async start

Yes, a future represents the *potential* answer returned

future is templated

A future is templated on the type that is expected to be returned by the async call.

Like everything else in c++, the types matter.

using a future

the `.get()` method of a future does one of two things:

- if the operation has already run, return the result
- if the operation has not yet started, run the operation (and wait) for the result
  - think join

exceptions

Nicely deals with exceptions:

- if the underlying thread throws an error which is not handled by the thread, the caller gets to handle the exception at the point of the `.get()`.

3.1

Threads

```
int doSomething (char c){
   // random-number generator (use c as seed to get different sequences)
   std::default_random_engine dre(c);
   std::uniform_int_distribution<int> id(10,1000);

   // loop to print character after a random period of time
   for (int i=0; i<10; ++i) {
      this_thread::sleep_for(chrono::milliseconds(id(dre)));
      cout.put(c).flush();
   }
   return c;
}

int func1 (){ return doSomething('.');}

int func2 (){return doSomething('+');}
```

```
int main(){
   std::cout << "starting func1() in background"
            << " and func2() in foreground:" << std::endl;

   // start func1() asynchronously (now or later or never):
   std::future<int> result1(std::async(func1));

   int result2 = func2();    // call func2() synchronously (here and now)

   // print result (wait for func1() to finish and add its result to result2
   int result = result1.get() + result2;

   std::cout << "\nresult of func1()+func2(): " << result
            << std::endl;
}
```

3.2

```
int main(){
   cout << "starting 2 tasks" << endl;
   cout << "- task1: process endless loop of memory consumption" << endl;
   cout << "- task2: wait for <return> and then for task1" << endl;

   auto f1 = async(task1);  // start task1() asynchronously (now or later or never)

   cin.get();  // read a character (like getchar())

   cout << "\nwait for the end of task1: " << endl;
   try {
      f1.get();  // wait, get exception if it ihappens
   }
   catch (const exception& e) {
      cerr << "EXCEPTION: " << e.what() << endl;
   }
}
```

```
void task1(){
   // endless insertion and memory allocation
   // - will sooner or later raise an exception
   // - BEWARE: this is bad practice
   list<int> v;
   while (true) {
      for (int i=0; i<1000000; ++i) {
         v.push_back(i);
      }
      cout.put('.').flush();
   }
}
```

only one .get()

You can only call a `.get()` once on a future.

- you have to wait at this point for the operation to complete (or throw)

you can also .wait() a future

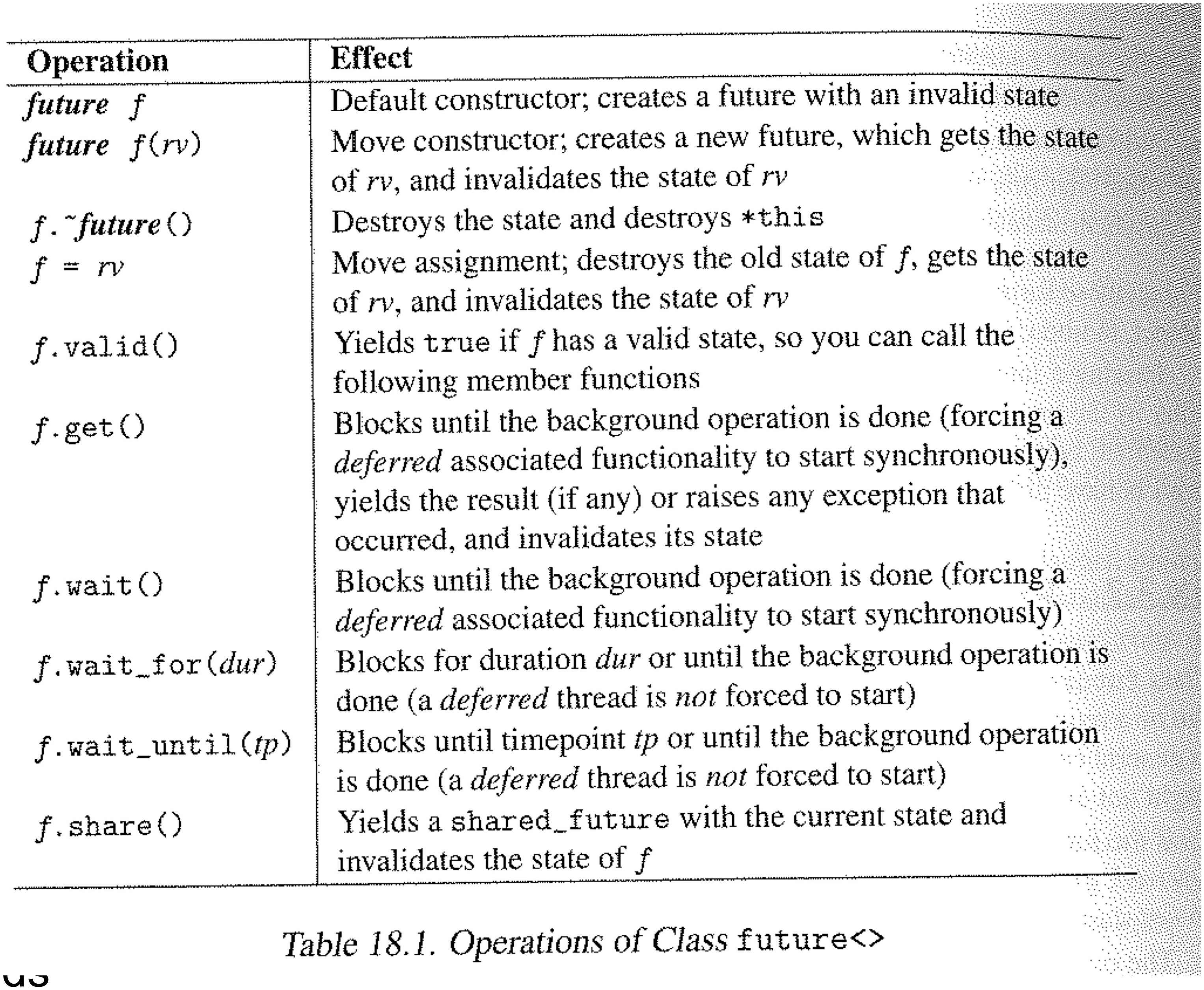
Three ways to wait

- `.wait()`, starts the thread (if it hasn't already started) and waits
- `.wait_for(duration)`
- `.wait_until(timepoint)`

timed waits return

returns a scoped enumeration

- `std::future_status::deferred` thread didn't start yet
- `std::future_status::timeout` thread is started but no result yet.
- `std::future::ready` Future result is ready, thread finished (or threw).



Operation	Effect
future f	Default constructor; creates a future with an invalid state
future `f(`*rv*`)`	Move constructor; creates a new future, which gets the state of *rv*, and invalidates the state of *rv*
`f.`*~future*`()`	Destroys the state and destroys `*this`
`f = `*rv*	Move assignment; destroys the old state of *f*, gets the state of *rv*, and invalidates the state of *rv*
`f.valid()`	Yields `true` if *f* has a valid state, so you can call the following member functions
`f.get()`	Blocks until the background operation is done (forcing a *deferred* associated functionality to start synchronously), yields the result (if any) or raises any exception that occurred, and invalidates its state
`f.wait()`	Blocks until the background operation is done (forcing a *deferred* associated functionality to start synchronously)
`f.wait_for(`*dur*`)`	Blocks for duration *dur* or until the background operation is done (a *deferred* thread is *not* forced to start)
`f.wait_until(`*tp*`)`	Blocks until timepoint *tp* or until the background operation is done (a *deferred* thread is *not* forced to start)
`f.share()`	Yields a `shared_future` with the current state and invalidates the state of *f*

Table 18.1. Operations of Class `future<>`

Threads

3.3

```
int main(){
   cout << "starting 2 operations asynchronously" << endl;
   // start two loops in the background printing characters . or +
   auto f1 = async([]{ doSomething('.'); });
   auto f2 = async([]{ doSomething('+'); });
   // if at least one of the background tasks is running
   if (f1.wait_for(chrono::seconds(0)) != future_status::deferred ||
      f2.wait_for(chrono::seconds(0)) != future_status::deferred) {
      // poll until at least one of the loops finished
      while (f1.wait_for(chrono::seconds(0)) != future_status::ready &&
           f2.wait_for(chrono::seconds(0)) != future_status::ready) {
         //...;
         this_thread::yield();  // hint to reschedule to the next thread
      }
   }
   cout.put('\n').flush();
  // wait for all loops to be finished
  // process any exception
   try {
      f1.get();
      f2.get();
   }
   catch (const exception& e) {
      cout << "\nEXCEPTION: "
           << e.what() << endl;
   }
   cout << "\ndone" << endl;
}
```

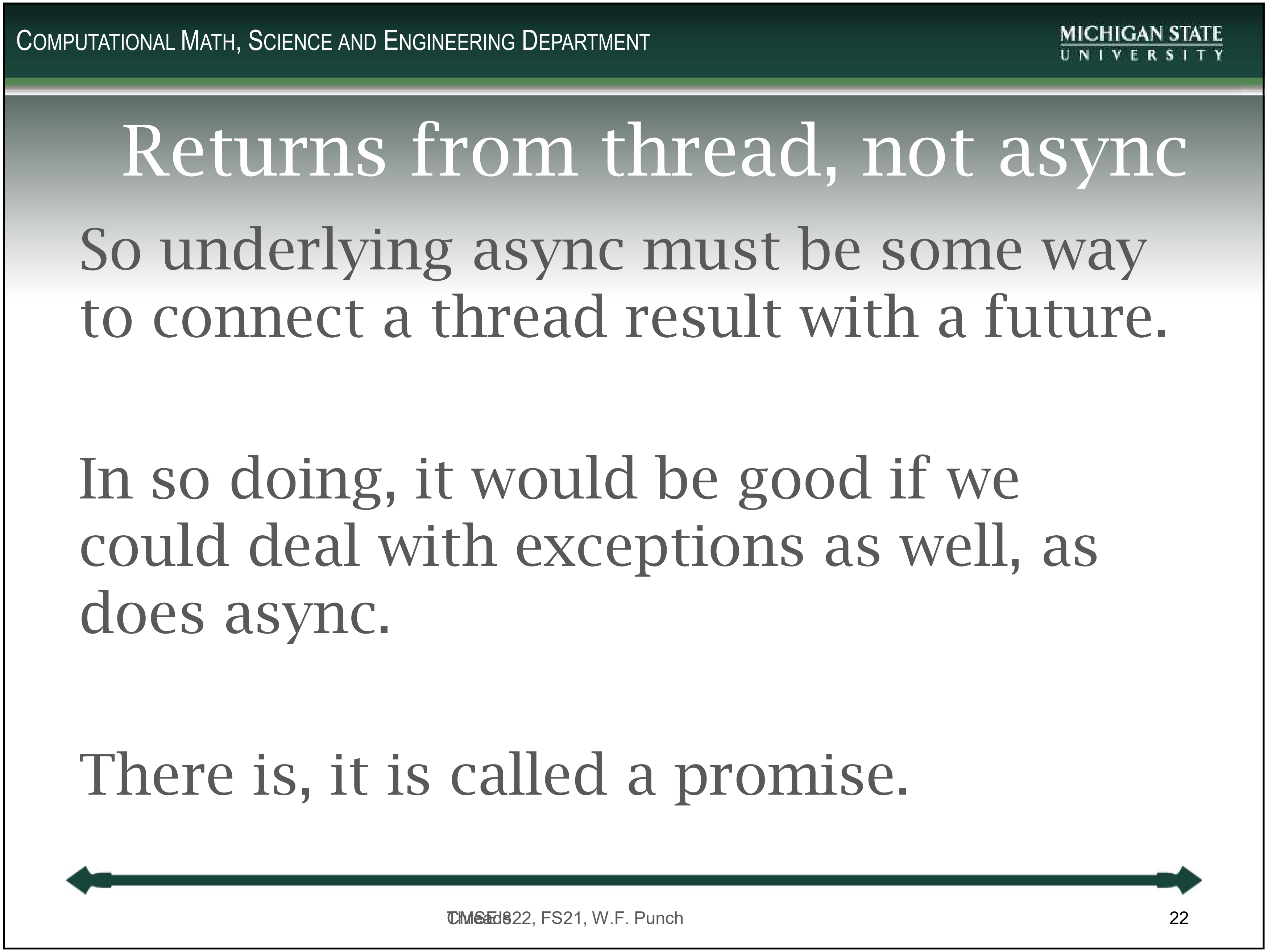
```
void doSomething (char c){
   // random-number generator
   // (use c as seed to get different sequences)
   default_random_engine dre(c);
   uniform_int_distribution<int> id(10,1000);

   // loop to print character after a random period of time
   for (int i=0; i<10; ++i) {
      this_thread::sleep_for(chrono::milliseconds(id(dre)));
      cout.put(c).flush();
   }
}
```

yield

`yield()` is a hint (not a command) to the scheduler to let `this_thread` go off a cpu and put another one on

- in a multi-cpu system this may be unnecessary so it is not a command



Returns from thread, not async

So underlying async must be some way to connect a thread result with a future.

In so doing, it would be good if we could deal with exceptions as well, as does async.

There is, it is called a promise.