boost::future

and such

Advent of Code Day 10: Balance Bots

You come upon a factory in which many robots are zooming around handing small microchips to each other. Upon closer examination, you notice that each microchip contains a single number; each bot only proceeds when it has two microchips, and once it does, it gives each chip to a different bot or puts it in a marked "output" bin. Sometimes, bots take microchips from "input" bins, too.

Your puzzle input is the collection of instructions for the bots. Some of the instructions specify that a bot should take a specific-valued microchip from the input; the rest of the instructions indicate what a given bot should do once it has two chips, by comparing their numeric values. For example:

```
value 5 goes to bot 2
bot 2 gives low to bot 1 and high to bot 0
value 3 goes to bot 1
bot 1 gives low to output 1 and high to bot 0
bot 0 gives low to output 2 and high to output 0
value 2 goes to bot 2
```

Advent of Code Day 10: Balance Bots

You come upon a factory in which many robots are zooming around handing small microchips to each other. Upon closer examination, you notice that each microchip contains a single number; each bot only proceeds when it has two microchips, and once it does, it gives each chip to a different bot or puts it in a marked "output" bin. Sometimes, bots take microchips from "input" bins, too.

Your puzzle input is the collection of instructions for the bots. Some of the instructions specify that a bot should take a specific-valued microchip from the input; the rest of the instructions indicate what a given bot should do once it has two chips, by comparing their numeric values. For example:

```
value 2 goes to bot 2
value 5 goes to bot 2
bot 2 gives low (value 2) to bot 1 and high (value 5) to bot 0
value 3 goes to bot 1
bot 1 gives low (value 2) to output 1 and high (value 3) to bot 0
bot 0 gives low (value 3) to output 2 and high (value 5) to output 0
```

Seems like a perfect task for *futures*!

You come upon a factory in which many robots are zooming around handing small microchips to each other. Upon closer examination, you notice that each microchip contains a single number; each bot only proceeds when it has two microchips, and once it does, it gives each chip to a different bot or puts it in a marked "output" bin. Sometimes, bots take microchips from "input" bins, too.

Your puzzle input is the collection of instructions for the bots. Some of the instructions specify that a bot should take a specific-valued microchip from the input; the rest of the instructions indicate what a given bot should do once it has two chips, by comparing their numeric values. For example:

```
bot[2].take(5);
bot[2].remember_sort_rule(bot[1], bot[0]);
bot[1].take(3);
bot[1].remember_sort_rule(output[1], bot[0]);
bot[0].remember_sort_rule(output[2], output[0]);
bot[2].take(2);  // kick off the whole thing
```

"value %d goes to bot %d"

```
struct Bot {
    int id_number;
    std::atomic<int> count_received;
    promise<int> pa, pb;
    future<int> fa, fb;

    Bot(int i) : id_number(i), count_received(0) {
        fa = pa.get_future();
        fb = pb.get_future();
    }

    void take(int v) {
        (count_received++ == 0 ? pa : pb).set_value(v);
    }

    // ...

};
```

"bot %d gives low to %s and high to %s"

```
struct Bot {

    // ...

    void remember_sort_rule(BotOrOutput& lo, BotOrOutput& hi) {
        when_all(std::move(fa), std::move(fb)).then(
            [&lo, &hi, id_number = id_number](auto future_of_ab) {
                auto [future_of_a, future_of_b] = future_of_ab.get();
                int a = future_of_a.get();
                int b = future_of_b.get();
                if (a < b) {
                    lo.take(a); hi.take(b);
                } else {
                    lo.take(b); hi.take(a);
                }
            }
        );
    }
};
```

Unfortunately, `std::future` lacks `.then()`

The `.then()` method is part of the “Concurrency” technical specification, a.k.a. *ISO/IEC TS 19571:2016 Programming Languages — Technical specification for C++ extensions for concurrency*.

So are `when_all()` and `when_any()`.

Unfortunately, the `<experimental/future>` header isn’t present in libstdc++ (GCC) or libc++ (Clang).

`boost::future` has all the toys in one place!

Boost has a future with `.then()`, `when_all()`, `when_any()`, `.unwrap()`, and so on and so forth. The one awkward bit is that you have to `#define` a ton of macros in order to get all the cool parts.

```
#define BOOST_THREAD_PROVIDES_FUTURE
#define BOOST_THREAD_PROVIDES_FUTURE_CONTINUATION  // .then
#define BOOST_THREAD_PROVIDES_FUTURE_UNWRAP
#define BOOST_THREAD_PROVIDES_FUTURE_WHEN_ALL_WHEN_ANY
#include <boost/thread/future.hpp>
using boost::promise, boost::future;
```

Unfortunately, `boost::when_all` leaks threads

```
#include <unistd.h>
#include <stdlib.h>
#include <stdio.h>

void print_thread_count()
{
    int pid = ::getpid();
    char cmd[100];
    sprintf(cmd, "grep Threads /proc/%d/status", pid);
    ::system(cmd);
}
```

Here’s the proof of concept: http://goo.gl/BFGQeY

Basically, every time you call `boost::when_all`, it creates a new thread in the background `wait_for_all()`ing on the futures you passed in. After stacking up a couple hundred `when_all`s, you’ll encounter an exception with the `what()` string `"Resource temporarily unavailable"`.

Solution: write your own, non-allocating, `when_all`!

Non-allocating `when_all()`, for the record

```
using boost::promise, boost::future, std::apply, std::make_tuple, std::tuple;

auto When_All() {
    promise<tuple<>> p;
    p.set_value({});
    return p.get_future();
}

template<class F, class... Rest>
auto When_All(F f, Rest... rest) {
    return future<tuple<F, Rest...>>(
        f.then([rest = make_tuple(std::move(rest)...)](auto f) mutable {
            auto w = [](auto... as) { return When_All(std::move(as)...); };
            return apply(w, std::move(rest)).then(
                [f = std::move(f)](auto future_of_tuple_of_rest) mutable {
                    return tuple_cat(make_tuple(std::move(f)), future_of_tuple_of_rest.get());
                }
            );
        })
    );
}
```

http://goo.gl/6ELJdz

Real-world solution (for today)

Just don’t use concurrency stuff to solve AoC Day 10.

Figure out a non-concurrent, non-future-based solution.