9

Exercise 58

Become familiar with `packaged_task`

We used the following code,

main.cc

```
#include <iostream>
#include <future>
#include <thread>
#include <iomanip>

using namespace std;

double lhs[4][5] =
{
    {1, 2, 3, 4, 1},
    {3, 4, 5, 7, 4},
    {2, 4, 5, 9, 3},
    {21, 8, 9, 42, 4}
};

double rhsT[6][5] =
{
    {1, 2, 3, 4, 2},
    {3, 4, 5, 7, 2},
    {2, 4, 5, 90, 3},
    {21, 8, 9, 42, 4},
    {1, 2, 3, 4, 8},
    {3, 4, 5, 7, 4}
};

enum
{
    ROWS = 4,
    COLS = 6,
    COMMON = 5,
};

future<double> fut[4][6];
```

unnecessarily global?

enums not used
global unnecessary

This uses packaged tasks allright.
But try to keep your programs a bit tidy.

```

double innerProduct(size_t row, size_t col)
{
    double sum = 0;
    for (size_t idx = 0; idx != COMMON; ++idx)
        sum += lhs[row][idx] * rhsT[col][idx];
    return sum;
}

void computeElement(size_t row, size_t col)
{
    packaged_task<double (size_t, size_t)> task(innerProduct);
    fut[row][col] = task.get_future();
    thread(move(task), row, col).detach();
}

int main()
{
    for (size_t row = 0; row != ROWS; ++row)
        for (size_t col = 0; col != COLS; ++col)
            computeElement(row, col);

    for (size_t row = 0; row != ROWS; ++row)
    {
        for (size_t col = 0; col != COLS; ++col)
        {
            try
            {
                cout << setw(5) << fut[row][col].get();
            }
            catch (exception &msg)
            {
                cout << "Exception: " << msg.what() << '\n';
            }
        }
        cout << '\n';
    }
}
```

In this case, better return nonzero, too!
(Either from the loop or later on)

1

7 = use actual pool of tasks

Exercise 59

Become familiar with `packaged_task` (2)

We used the following code,

main.ih

```
#include <iostream>
#include <future>
#include <thread>
#include <iomanip>
#include <mutex>
#include <queue>
#include "semaphore/semaphore.h"     // from ex57

using namespace std;


enum
{
    ROWS = 4,
    COLS = 6,
    COMMON = 5,

    NTHREADS = 8,
    NBUSYWORKERS = 0
};

struct RC
{
    size_t row;
    size_t col;
};

typedef packaged_task<double (RC)> PTask;

extern PTask pTask[ROWS][COLS];
extern double lhs[4][5];
extern double rhsT[6][5];
extern queue<RC> todoQueue;
```

Good :)

Now use them!

ugly

```
extern mutex queueMutex;
extern Semaphore producer;
extern Semaphore worker;


double innerProduct(RC rc);
RC getSpecs();
void client();
void produce();
```

main.cc

```
#include "main.ih"

double lhs[4][5] =
{
    {1, 2, 3, 4, 1},
    {3, 4, 5, 7, 4},
    {2, 4, 5, 9, 3},
    {21, 8, 9, 42, 4}
};

double rhsT[6][5] =
{
    {1, 2, 3, 4, 2},
    {3, 4, 5, 7, 2},
    {2, 4, 5, 90, 3},
    {21, 8, 9, 42, 4},
    {1, 2, 3, 4, 8},
    {3, 4, 5, 7, 4}
};

PTask pTask[ROWS][COLS];

queue<RC> todoQueue;
mutex queueMutex;

Semaphore producer(NTHREADS);
Semaphore worker(NBUSYWORKERS);

```

Why must all this stuff suddenly be global?

this way, all functions essentially form one big tangle.

```
int main()
{
    for (size_t idx = 0; idx != NTHREADS; ++idx)
        thread(client).detach();

    produce();

    for (size_t row = 0; row != ROWS; ++row)
    {
        for (size_t col = 0; col != COLS; ++col)
        {
            try
            {
                cout << setw(5) << pTask[row][col].get_future().get();
            }
            catch (exception const &msg)
            {
                cout << "Exception: " << msg.what() << '\n';
            }
        }
        cout << '\n';
    }
}
```

Ok: main is kept quite readable :)

return nonzero if shit happens

client.cc

```
#include "main.ih"

void client()
{
    while (true)
    {
        worker.wait();

        RC rc = getSpecs();
        if (rc.row == ROWS)
            return;

        pTask[rc.row][rc.col](rc);

```

This depends on each client() thread popping at least an RC with row == ROWS. How is that guaranteed to happen?

```
        producer.notify_all();
    }
}
```

getspecs.cc

```
#include "main.ih"

RC getSpecs()
{
    lock_guard<mutex> lg(queueMutex);
    RC ret = todoQueue.front();

    if (ret.row == ROWS)
    {
        worker.notify_all();
        return ret;
    }

    todoQueue.pop();
    return ret;
}
```

why keep the lock so long?

. . . oh, I see: row == ROWS is not popped.

innerproduct.cc

```
#include "main.ih"

double innerProduct(RC rc)
{
    double sum = 0;
    for (size_t idx = 0; idx != COMMON; ++idx)
        sum += lhs[rc.row][idx] * rhsT[rc.col][idx];
    return sum;
}
```

produce.cc

```
#include "main.ih"
```

```

void produce()
{
    for (size_t row = 0; row != ROWS; ++row)
    {
        for (size_t col = 0; col != COLS; ++col)
        {
            producer.wait();
            pTask[row][col] = PTask(innerProduct);
            {
                lock_guard<mutex> lg(queueMutex);
                todoQueue.push(RC{ row, col });
            }
            worker.notify_all();
        }
    }
    todoQueue.push(RC{ ROWS, COLS });
    worker.notify_all(); // notify threads to stop
}
```

NAE: this makes sure only N tasks
are active at a time, but there are ROW * COL
packaged tasks still.
There is no pool; there is a matrise of tasks.
Hint: a matrix of futures would be ok...

All this global stuff gives me itches. But our example is about as ugly.
So you get 1 on my assumption that you could easily replace
a matrise of tasks with a matrix of futures.

6

Exercise 60

Learn to implement a multi-threaded algorithm (2)

We used the following code,

main.cc

```
#include <iostream>
#include <algorithm>
#include <future>

using namespace std;

void quickSort(int *beg, int *end)
{
    if (end - beg <= 1)
        return;

    int lhs = *beg;
    int *mid = partition(beg + 1, end,
        [&](int arg)
        {
            return arg < lhs;
        }
    );

    swap(*beg, *(mid - 1));

    async(launch::async, quickSort, beg, mid);
    async(launch::async, quickSort, mid, end);
}

int main()
{
    int ia[] = {16, 2, 77, 40, 12071, 12, 3134, 42,
                5, 2453, 45, 3456, 35, 6, 56, 546, 2};

    size_t iaSize = 17;

    quickSort(ia, ia + iaSize);
```

← No guarantee that this finishes before the function returns.