11/13/13 ①

Concurrency:

This tutorial is about multi-threading in C++. Concurrency is already important in some software domains, but with the commoditisation of true multi-core CPUs it will become increasingly important for wider application development as well. Respected voices say the concurrency revolution will be more disruptive than the object oriented revolution[1], and I believe them.

As concurrency moves into the mainstream, there will be a great competitive advantage for companies and individuals able to harness the power of parallellism effectively and safely. Since it is so very difficult to debug poorly executed concurrent programs, there is also a nasty quagmire waiting for those who cannot.

Multi-threading vs. multi-processing

Submitted by pbridger on Wed, 2006-03-22 03:59.

This tutorial focusses on *multi-threading* rather than multi-processing, though most of the concepts discussed are common to all concurrency. Let's compare these two forms of concurrency. If you're not interested, feel free to skip to the next section.

Multi-threading refers to an application with multiple threads running within a process, while *multi-processing* refers to an application organised across multiple OS-level processes.

A thread is a stream of instructions within a process. Each thread has its own instruction pointer, set of registers and stack memory. The virtual address space is *process specific*, or common to all threads within a process. So, data on the heap can be readily accessed by all threads, for good or ill.

IP
Registers
Stack

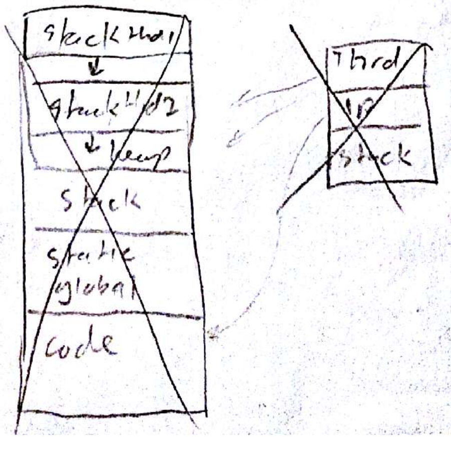
Multi-threading is a more "light weight" form of concurrency: there is less context per thread than per process. As a result thread lifetime, context switching and synchronisation costs are lower. The shared address space (noted above) means data sharing requires no extra work.

- switch fast
- shared heap
- crash?

Multi-processing has the opposite benefits. Since processes are insulated from each other by the OS, an error in one process cannot bring down another process. Contrast this with multi-threading, in which an error in one thread can bring down all the threads in the process. Further, individual processes may run as different users and have different permissions.

Read more: http://www.paulbridger.com/node/17/#ixzz2l0AQfeyH

Read more: http://www.paulbridger.com/multithreading_tutorial/#ixzz2l09hkP00



~~OS must support thread scheduling (stop / start another on a queue (ready + blocked) share main thread global static heaps + possibly main thread stack)~~

②

multi thread verses multiprocess

Limited resource

200,000 to create

100,000 to destroy

1 Meg for stack

2000-8000 for context switch

threads - shared mem (no extra work)

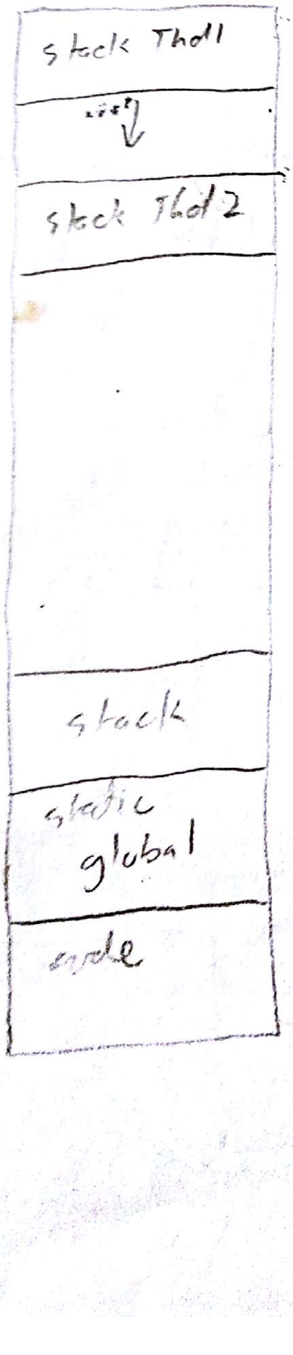
- lightweight
- own IP, registers, stack space
- lifetime, switching, sync cost lower
- crash 1 thread crash process

Process - error in one cannot bring down another

- hard to share data
- more expensive to switch

For thds dead hard to flush mem

For process might have to



Main

Thd1

Thd2

with processors time-slice (15 → 30ms on Windows)

each thread Main, Thd1 & 2

~~share~~ split it up.

OS supports scheduling (stop / start another).

Circular buffer (ready queue)

→Thd1 → Thd2 → . . . Thdn;

If a thread stops or is blocked for some reason goes to a blocked queue & stays until its ready to return ~~std::chrono::milliseconds dura(2000);~~

~~the std::this_thread::sleep_for (dura)~~

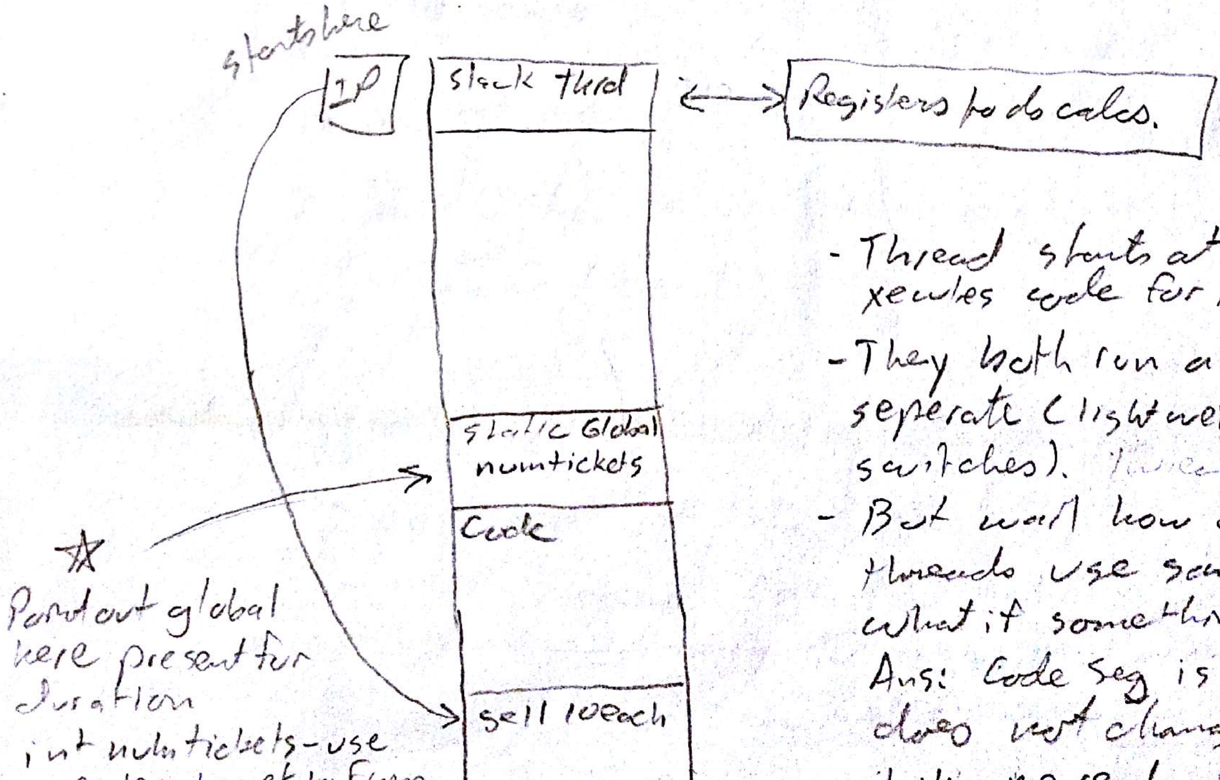
Demo 20_Thread_Ticket_Agent

① Do sequential method
not natural 1 → 2 → 3 . . .
sell 10 sell 10 sell 10

② Then concurrent with sell_10_each()
still not natural, should sell all you can.

Examine Thread launch code (line 70)

$$\texttt{std::thread(}\ \underbrace{\texttt{sell\_10\_each}}_{\text{function to start execution}}\ ,\ \underbrace{\texttt{myagents[i]}}_{\text{params in this case agent \#}}\ \texttt{)}$$



- Thread starts at sell_10_each & xecutes code for its timeslice
- They both run along entirely seperate (lightweight context switches). Twice
- But wait how can do diff threads use same codebase? what if something changes? Ans: Code Seg is read only does not change.
- Like me reading over your shoulder, it doesn't matter where either of us are, we don't affect each

- They both run merrily along
- one or the other exits
- what happens?
  - Main thread exits 1st, your thread tries to continue on reclaimed mem. crash
  - Thread exits 1st, generally no worries.
  - But which?
- Answer - you decide join the thread (wait for it to finish) before you exit.

```
thread.join();
```

syntax in code;

```
for ( auto& thread : threads)
```

auto means automatically infer type in this case

auto& just infers the type, same as

```
for ( std::vector<std::thread>::iterator
```

join it twice (crash)

next change to concurrent → sellAsManyAsPossible

pick one of the lines

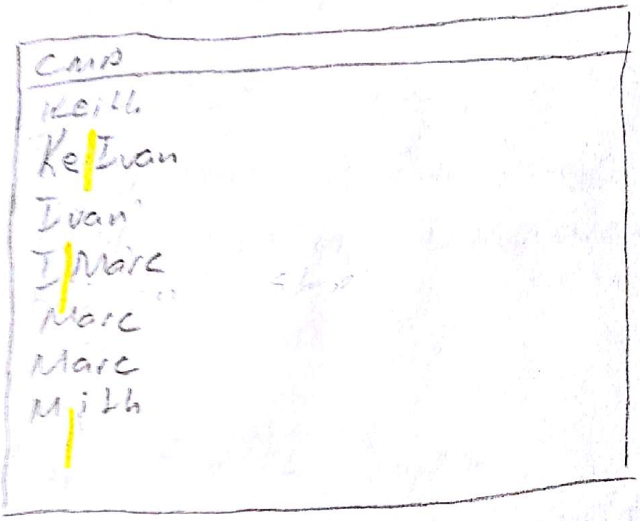
Agent, Agent 10 sold 1 ticket.
↑ preempted here, find where it started again. (after everyones turn)

am what happens

ass demo:

Main Thread: pick someone
spawn couple of new threads: pick 2 people

draw cmd window:
time slice, each write name & CRLF, & stop exactly when time is up.



```
CMD
Keith
KeIvan
Ivan
IMarc
Marc
Marc
Mith
```

Each have 20 → 30ms

Show code again point out where context switch causes problem. Show where rest of line printed This is what thread can do per (20-30ms slice)

- Other problems?

[redacted]

Yes lots, in sellAsManyAsPossible()

[Debug → Assembly (Ctrl-F11)]

--numTickets;

3 statements.

1st get copy
2nd subtract
3rd put back

move around, get -19 move back works

ed regulation

Board people

Have a token called Mutex (Pink paper)

Must have mutex to write

```
→ aquire
  write name
  give up
```

But when your time slice only you running you will always get it regardless of whether others want it or not!

The Fix

Need a traffic cop -
1 at a time others wait.

Mutual Exclusion

```
#include <mutex>

std::mutex mutex.
```

//usage

```
mutex.lock()          or try_lock() → returns false go about
   block + wait                          business & wait
   or success                            return true - got it.
mutex.unlock()
```

went over RIAA mutex as well

what if only had to read var? Is locking necessary?

[redacted]

at:

wat if you try to lock mutex twice?
can't you've already locked it, bizarrly you have to get in line again.

20_Mutex

Deadlock1();

Boom! locked up

Fix? Yep, use std::recursive_mutex()

Deadlock1Fixed_mutex_needs_same_number_...

but pain to unlock once per lock

Roll your own, 1 unlock does it all.

Deadlock1Fixed_1_unlock.....

Bigger Probs a typical deadlock,
2 synch objects, Must Aquire in same order every time or lock up

show deadlock_2_.....

Break into code Debug → Attach to process →
Debug → break all
show 2 worker threads waiting to lock

Missing? ~~good bit there are other thread libraries that are better, boost for instance~~
Need to ~~to~~ have cross process mutexes then need 'named' mutexes so each process can look for mutex by name

but see en.cppreference.com/w/cpp/thread to meager C++11 offerings