LibreOfficeKit & Online event delivery & scheduling

What happens; in what order?

Michael Meeks
Itinerant Idiot & Hacker
michael.meeks@collabora.com
First synchronous vs. asynchronous
What does it mean?

A **synchronous** operation blocks a process till the operation completes.

- int read(char *data, size_t buffer_size);
  
  “Do it now, while I wait!”

An **asynchronous** operation is non-blocking and only initiates the operation. The caller could discover completion by some other mechanism discussed later.

- readAsync(std::function<void
  (std::vector<char> &)> & cb);
  
  “Go away and do it, tell me when you’re done”
Problems of async:

Producers / consumers mismatch

- for (i = 0; i < 1000000; ++i)
  - writeAsync("Hello world");

How is that solved generally?

- Queueing
- Protocol knowledge:
  - eg. invalidations – we can coalesce.
  - eg. state changes – eliding intermediate state transitions.
  - eg. mouse moves (?) ...

Debuggability

- What are we waiting for?
  - Stack trace: an empty loop ...
  - kill -USR2 → expose state ...

Some problems of sync

- Endless threads – all of them blocking doing nothing:
  - Hammers the O(1) scheduler, memory cost
  - Typical thread problems ...
- ‘read’ thread, ‘write’ thread + ‘do’ thread - per socket?
LibreOfficeKit workings
LibreOffice main app ...

Nastily complicated threading
- Won’t go into this here – lots of synchronization and complexity
  - 1+ epsilon threading mostly

Threading successes
- Manageable small zones, of code-locking, targetted:
  - threading for image scaling
  - ZIP compress / de-compress
  - XML parsing
  - Calc Formula calculation

Everything else
- Uses the main-loop:
  - Idle
  - Timer
  - Scheduler ...
- One big lock: SolarMutex ...

LibreOfficeKit advantages
- headless/svpinst.cpp
  - a somewhat simpler ImplYield
- SolarMutexReleaser
Main-loop integration

What is a main-loop?

- Process - consumes events from various directions, and processes them
- Then it sleeps - in a 'poll' type call waiting for more work.
- This is Scheduler and ImplYield() in LibreOfficeKit core

Even when super-busy - 10x users

- Spend lots of time in poll

Problem:

- LibreOffice has no good async I/O abstraction and/or APIs.
- COOL - is ~100% async to improve debuggability

Unipoll - vcl::lok::isUnipoll()

- We replace VCL's headless backend with our own poll callback:

  ```cpp
  int nPollResult = pSVData->mpPollCallback(pSVData->mpPollClosure, nTimeoutMicroS);
  ```
COOL polling pieces
How does that look?

LibreOfficeKit

- Process idle handlers eg. re-layout document

Yield /poll

Coolforkit

- Kit Document
  - Event processing
  - Poll & read from incoming sockets
  - Push non-rendering events into the Kit
  - Render tiles, RLE & delta in threads & send

Yield /poll

Coolwsd

- DocumentBroker
  - Ooh events ! =)

Yield /poll

Browser

- JS Websocket
  - INCOMING: status bar change
  - INCOMING: tile / delta & re-render

Yield /poll
Queues – important ...
LOK: CallbackFlushHandler

core/desktop/source/lib/init.cxx:

- Queues core LOK events before sending them.

```c++
void CallbackFlushHandler::
    libreOfficeKitViewCallback
    (int nType,
        const OString& pPayload)
{
    CallbackData
        callbackData(pPayload);
    queue(nType, callbackData);
}
```

Problem:

- LibreOfficeKit loves to give duplicate event notifications:
  - 1st step – de-duplicate in the queue.
- LibreOfficeKit loves to emit events at unhelpful times:
  - eg. invalidations during rendering
- Potential threading issues ...

Solution:

- Queue events, de-duplicate, sanitize
- scheduleFlush()

```c++
Application::PostUserEvent(
    LINK(this,CallbackFlushHandler,
        FlushQueue));
```
Kit: KitQueue ...

online/kit/KitQueue.hpp

- Queues events from CallbackFlushHandler
- Queues incoming events from DocumentBroker
- ‘poll’ callback works on this queue
  - First → sending callbacks to the coolwsd
  - Then processing the incoming events

// a LOK compatible poll function merging the functions.
// returns the number of events signalled
int KitSocketPoll::kitPoll(int timeoutMicroS)
   → void Document::drainQueue()

Queue filling:

Kit.hpp: CallbackFlushHandler:

/// A new message from wsd for the queue
void queueMessage(const std::string &msg)
   { _queue->put(msg); } 

Solution:

- From one queue to another ...
- Similar work – de-duplication etc.
coolwsd: DocumentBroker

Queue ...

- Process incoming data from Kit fast ...
- For outgoing data:

  void enqueueSendMessage(
    const std::shared_ptr<Message>& data);

Producer /Consumer handling:
  
  Notified when space in the outgoing socket → browser

void ClientSession::writeQueuedMessages(std::size_t capacity)
{
    LOG_TRC("performing writes, up to " << capacity << " bytes");
    std::shared_ptr<Message> item;
    std::size_t wrote = 0;
    try
    {
        // Drain the queue, for efficient communication.
        while (capacity > wrote && _senderQueue.dequeue(item)…

          if (item->isBinary())
              Session::sendBinaryFrame(data.data(), size);
          else
              { Session::sendTextFrame(data.data(), size);
JS – Queueing incoming messages

online/browser/src/core/Socket.js

- Touching the DOM is -very- expensive
- We badly need to control when we mutate anything that is not internal state.
- Two queues:
  - this._delayedMessages
  - this._slurpQueue