

Synchronous Threading

Adam Langley <agl@imperialviolet.org>

REVISION WED DEC 26 14:09:18 GMT 2001

1 Terms

- I. **SYNCHRONOUS THREADING:** threading where threads of execution do not run concurrently, and are not preempted
- II. **CONCURRENT THREADING:** threading where threads of execution run concurrently and are preempted by a scheduler.
- III. **USER-LAND THREADS:** threads implemented by code in the application and, about which, the kernel has no knowledge. Can be either synchronous or concurrent.

2 Advantages of Threading

In applications where transactions have little state (for example, SMTP servers) an event model will suffice where the current state of the transaction is recorded in an in memory structure which is read by handling functions.

However, in applications where the state of a transaction can be very complex and can involve multiple communications channels, implementing this concept can be error prone and confusing.

The alternative (a threading model) is to carry state implicitly in the position that the code is currently executing since this is far more intuitive.

As an example, imagine, for the moment, that 1000 cakes need to be baked. The recipe for baking a cake is well understood and can be formalised. In an event model one “master chef” would read the notes beside each cake and do whatever was needed to move it to the next stage. The chef would then update the notes and move on. At some point in the future they return to that cake and read the notes they updated last time to learn what action is now needed.

In a threading model there would be one chef for each cake and they would follow the recipe without needing to read or write notes.

Now, thinking generally about the structure of the code needed to implement each model, we can see the event model would involve code to read state, a complex system of CASE statements and code to write the state out. Converting a flow chart recipe in to the code would be a major, and error prone, translation.

However, in the threading model, a recipe flowchart has a much closer isomorphism with the structure of the code (flow chart loops can be converted into while loops and so on) – so the code is simpler and clearer.

3 Advantages of Synchronous Threading

At the risk of flogging the cooking example to death, let us now consider the differences of synchronous threading and concurrent threading.

In a concurrent threading world all the cooks move at once (we are now discarding the “master cook” model). The major problem is that cooks might step on each others toes as they work. If a cook requires the sugar, but the cook next to them is using it, then the system fails. Mutexs are required to “lock” the sugar so that only one cook at a time is using a bag of sugar.

Although this may seem silly when thinking in terms of cooks, it is the critical weakness of concurrent threading. When thinking, not in terms of bags of sugar, but complex data structures being in an inconsistent state as two threads try to update it at the same time, it becomes very serious.

In concurrent threading, mutexs are needed wherever threads interact to stop these types of problem. This not only requires much thought about where to lock, it introduces the risk of heisenbugs.

Dealing with these in turn. Locking has to carefully thought out. Having one super-lock for the whole application might be simple, but it kills performance as threads have to wait when they shouldn't have to. However, having many small locks increases complexity and the risk of deadlock (where there is a cyclic lock dependency) greatly.

Heisenbugs occur where a piece of code forgets to lock something before using it. From the jargon file:

“(from Heisenberg’s Uncertainty Principle in quantum physics) A bug that disappears or alters its behaviour when one attempts to probe or isolate it. (This usage is not even particularly fanciful; the use of a debugger sometimes alters a program’s operating environment significantly enough that buggy code, such as that which relies on the values of uninitialised memory, behaves quite differently.)”

Because the resulting bug is highly dependent on the timing of threads it creates very subtle problems which can be near impossible to track down. Often the best way to find these bugs is a code audit.

In the synchronous threading model, only one cook is acting at a time and they perform a small operation before letting someone else act.

(In uniprocessor systems, at one level, this is how the concurrent threading model also works as the kernel task switches between the threads. The difference is that an application has no control of task switching in the concurrent model and task switches happen where the kernel, not the application, wants them to.)

Since only one cook is acting at a time they can make sure the sugar is in the right place before letting someone else act. This removes the need for locking in 99% of cases and the code is far simpler and much less error prone.

It’s not a perfect world, there are some disadvantages of synchronous threading. Firstly, if a thread doesn’t yield and let other threads run the whole process locks. Mainly this is the result of a code bug (an infinite loop) and is easy to find (break the running application and see what it’s executing).

However, some operations are long and difficult to split up (public key operations spring to mind) and no other thread can run until they are complete. In reality, common public key operations are short enough not to cause problems, but something like key generation certainly is not.

Some libraries cause a similar problem. If a library function calls triggers a blocking system call (for example, `GETHOSTBYNAME(3)`) then the kernel will put the whole application to sleep (since it knows nothing of the user-land threads contained therein).

The solution to both of these problems (long atomic operations and blocking library calls) is to have some concurrent, kernel worker threads. In this model, problem operations are sent to a concurrent thread (switched by the kernel) while the user-land thread may be put to sleep and woken up when the result is ready from the worker thread.

This gives the best of both worlds – it solves the problems of synchronous threading without introducing the problems of concurrent threading since the number of required worker threads is usually very small and have a very simple interface (requiring very little locking).

4 Implementation Practicalities

The basic operation of user-land threading is characterised by the `LONGJMP(3)` and `SETJMP(3)` functions. These functions load and save the current register set into a structure. Reports suggest that these functions are broken in the Visual C++ Runtime Libraries, but are not difficult to implement with a little assembly.

The whole of user-land threading can be built from these primitives. The following is a simple code example

```
1  class Thread {
2      void *stack;
3      jmp_buf our_state;

4      Thread () {
5          stack = (void *) mmap (0, PSZ * STACK_PAGES, PROT_READ | PROT_WRITE,
6                                MAP_PRIVATE, zerofd, 0);
7          memset (stack, 0, PSZ * STACK_PAGES);
8          if (stack == MAP_FAILED)
9              abort ();
10         if (mprotect ((char *) st, PSZ, PROT_NONE) < 0)
11             abort ();
12     }

13     void new_thread () {
14         GLOBAL_thread = this;
15         jmp_buf tmp_ns_state;

16         memcpy (&tmp_ns_state, &GLOBLAL_ns_state, sizeof (jmp_buf));
17         if (setjmp (GLOBAL_ns_state)) {
18             memcpy (&GLOBAL_ns_state, &tmp_ns_state, sizeof
19                   (jmp_buf));
20             return;
21     }
```

```

22     asm ("movl %0,%%esp" : : "g" (&st[(PSZ * STACK_PAGES) - 16]));
23     start ();
24 }
25 void start () {
26     u8 a;
27     memset ( ((u8 *) &a) + 5, 0, 4);
28     try {
29         run_thread ();
30     } catch (...) {
31         ERROR ("Uncaught exception in thread");
32     }
33     end_thread (this);
34 }
35 uint
36 sleep () {
37     uint val;
38     val = setjmp (our_state);
39     if (val == 0)
40         longjmp (GLOBAL_ns_state, 1);
41     return val;
42 }
43 void
44 run (uint v) {
45     if (setjmp (GLOBAL_ns_state))
46         return;
47     longjmp (our_start, v);
48 }
49 };

```

The code above is an example, not practical code. The reader is invited to read NEUROSES.CC and NEUROSES.H for practical code.

I will try not to use the line numbers of the code too much, as they all break when I find I've missed something in the example, so keep your wits about you.

In the constructor a block of memory is allocated which will be the stack of the new thread. This is the first point of style – with userland threading you have a fixed stack and you cannot go off the end. This isn't generally a problem if you have a sensible stack size (neuroses uses an 76KiB stack, and this seems to work fine), but you may need to use MALLOC where you might have got away with ALLOCA before.

The stack memory is allocated by memory mapping “/dev/zero” because this allows a call to MPROTECT to lock the lowest page of the stack. On Intel processors the stack grows *down* in memory, items on the top of the stack have the smallest memory address. Thus locking the lowest page of the stack will trigger a GPE if the thread runs off the end of the stack (rather than start misusing random areas of memory)

The NEW_THREAD function actually sets up the thread for real, but before we can discuss it we need to understand the actions of SETJMP and LONGJMP.

SETJMP saves all the required CPU registers into a JMP_BUF structure, which is passed as its only argument. It then returns 0. LONGJMP takes two arguments, the JMP_BUF and an integer. It loads the registers from the JMP_BUF and jumps to the address saved by SETJMP, thus the call to SETJMP returns *for a second time*. The second time it returns the integer argument to LONGJMP (which may not be 0) so the caller can tell the cases apart.

We can now understand NEW_THREAD. Firstly it sets the global GLOBAL_THREAD to itself. This is needed because the THIS pointer is on the stack, and we are about to switch stacks. Next it makes a copy of the global GLOBAL_NS_STATE. This JMP_BUF is used to store the state of the scheduler so that threads can yield and jump back to the scheduler.

Now it stores the current register state in GLOBAL_NS_STATE. Let us, for the moment, ignore the case where the return value is non-zero. The assembly snippet sets the ESP register to the stack we allocated in the constructor. This is the actual switch of stack and we are now running in the thread and we call START which runs the actual code for the thread.

The thread now runs its initial code, what ever that may be, and at some point it is ready to yield back to the scheduler so it calls SLEEP.

SLEEP saves the register in the per-thread JMP_BUF called OUR_STATE. If the return value of SETJMP is 0, then it LONGJMPs to the state we saved in NEW_THREAD.

Now in `NEW_THREAD` the `SETJMP` will return 1 and we are executing in the scheduler context again (because we called `SETJMP` before switching the stack). `GLOBAL_NS_STATE` is restored and we now return.

At some point in the future the thread may be worked by calling `RUN`. The code for this should be understandable by now. It saves the current state and `LONGJMPs` to `OUR_STATE` (which will end up in `SLEEP` again).

That leaves two things to be discussed; the `MEMCPY` call in `START` and the copying of `GLOBAL_NS_STATE` in `NEW_THREAD`.

The `MEMCPY` call is simply an example of fixing up the stack. In this case GDB (the debugger) needed a `NULL` on the stack so that getting backtraces didn't have a lot of cruft at the end. You may need something similar.

The copying of `GLOBAL_NS_STATE` is needed because threads can be created by other threads, which need their own copies of it so they can return.

Fork like threading

One concept so far ignored is threading with semantics like `FORK(2)`. In this case the call to create a thread deep copies the stack and returns one value in the context of the parent and another in the context of the new thread. For example

```
1  std::string s;
2  some_code();
3  more_code();
4  if (new_thread ()) {
5      /* In the context of the child thread here */
6      s = "abc";
7  } else {
8      /* In the context of the original thread here */
9  }
```

I've ignored this for a very good reason, namely that it's a really bad idea. Although it might lead to cleaner code in some cases it is very dangerous. Firstly, if the parent thread executes first and returns from the scope where `s` was defined, then the use of `s` in the child thread is invalid and will refer to free memory. Insert all the standard problems when deep copying objects here.

Also throwing exceptions in this model creates a total mess. Avoid it.

5 Scheduler

Now that we have the basics down, most of the work goes into waking up threads in new and interesting ways.

The most useful and most basic reason to wake up a thread is because some I/O operation has completed, or is ready to be performed. We need two things for this; a way to monitor I/O events and wakeup processes, and a way for threads to mark their interest in something.

To do the first we build an event loop which takes in all the registered interests of all the threads and calls `SELECT`, `POLL` or some similar function (like `WAITFOR-MULTIPLEOBJECTS`). The event loop can then mark the right threads as runnable so that they get run when the scheduler next ticks.

In order for a thread to mark its interest it is helpful to simulate standard `READ` and `WRITE` operations with functions which perform non-blocking operations. For example, if the event loop has an array for each possible `fd`, and threads enter themselves into that array to mark their interest, a read function could be written thus:

```
1  int read (int fd, void *buf, uint count)
2  {
3      fds[fd]->reading_thread = current_thread;
4      try {
5          sleep ();
6      } catch (...) {
7          fds[fd]->reading_thread = NULL;
8          throw;
9      }
10     /* that fd is now readable */
11     fds[fd]->reading_thread = NULL;
12     return ::read (fd, buf, count);
13 }
```

That should be easy to understand, given that `::READ` references the global function `READ` (e.g. the libc one). Functions like the above for `WRITE`, `ACCEPT`, `CONNECT` and the like can be written.

Another scheduler trick is timeouts. In the threading code above the scheduler can pass an integer to the thread when waking it up (via the return value of `SETJMP`). Put a case statement in `SLEEP` and throw a `TIMEDOUT` object (or some such) when that value is passed. Now write a priority queue which can take (time, thread) tuples and a call in the event loop to a function which pops tuples off the front of the queue and wakes up threads whose time is up.

The practical upshot of which is that threads can register a timeout and have an exception thrown if it happens. This makes it easy to register timeouts for complex operations (such as crypto handshaking) which may involve many small I/O operations.

If you feel adventurous you can register a handler for `SIGSEGV` and set things up so that exceptions are thrown in the thread such as `NULLPTREREXCEPTION`. This isn't too hard to do (though I took it out of Whiterose because I don't know how to do it on Windows).

It might be advantageous to allow threads to register I/O operations, but not sleep on them. The thread could then set a number of non-blocking operations off and do a pseudo-select on them.

Generally, have a ball.

Higher level operations

It can be useful for the waking of threads to happen outside the event loop. For example, a worker thread (kernel thread to perform long atomic operations or blocking library calls) may use a pipe to communicate with a user-land thread. The UL thread may use the event loop to wait on the pipe and process the results. That thread then needs to wake others.

For this situation I have found that two primitives usually suffice; a `BLOCKPOINT` and a `WAITQ`. The block point can have one thread wait on it at a time and simply makes this thread runnable when triggered.

A wait queue can have a number of threads waiting and triggering it either wakes the thread at the front of the queue, or all the threads in the queue.

From this `LOCKS` and `LOCKHANDLES` can be made (user-land mutexes for the rare times when they are needed. Actually, a grep of the Whiterose sources shows that

the only time they are used is in the unit tests), as can SIGNALS and SLOTS. The reader is invited to look at WAITQ.CC and WAITQ.H in the Whiterose source for implementations of these.

This simply leads to an implementation of a function like GET_MESSAGE. In this case we want to wake a thread when a message with a given UniqueID comes in. Very simply, two maps will do; the first mapping UniqueIDs to MESSAGE objects and the second to BLOCKPOINT objects. When a thread listening on a connection gets a message it looks in the second map for a waiting thread. If there is one it wakes it and puts the message in the first map for the thread to process. Otherwise, it starts a new message handler (possibly because the memory of the chain has timed out and it's a stale message).

That's just an example, in reality you would want to be able for threads to wait on a (peer, UniqueID) pair and to queue messages, but it's the same concept.