A Method For Handling Asynchronous Events In **FLTK** 1.x

Introduction

The **FLTK** graphical toolkit (versions 1.x) doesn't handle multithreading, i.e. it may cause problems and unexpected behaviour if more than one thread performs update on widgets at a time. This means limitations in cases when an asynchronous event happens outside the GUI event loop, e.g. in another thread of the program, and this should cause updates on the GUI.

This short guide describes a method to handle this kind of asynchronous trigger.

Getting into the main event loop

The main goal is to *synchronize* an outside event with the GUI event loop, i.e. somehow provide a trigger which can be acted upon in **FLTK**'s GUI event loop by the user's program. This way we can make sure that when the trigger is processed no other thread is performing updates on any widget. The explanation for this is that the GUI event loop is *single−threaded*, so only one trigger at a time is (can be) processed.

Note that I won't give you a method which would allow *simultaneous updates on a widget*. However, this is *not* a real limitation; first because **FLTK** has only one event loop thread, so even after updating widgets simultaneously one would have to wait for the event loop to finish and redraw the widgets. Second, any number asynchronous events from any threads can send a trigger *into* the GUI event loop with the mechanism described below.

The Solution

Fortunately, **FLTK** provides a quite handy group of methods for this:

```
static void Fl::add_fd(int fd, void (*cb)(int, void *),
                       void* = 0);
static void Fl::add_fd(int fd, int when, void (*cb)(int, void*),
                       void* = 0);
static void Fl::remove_fd(int);
```

This means that we can get a callback if any data is available to be read from a specified file descriptor (**fd**). (Please refer to the bundled FLTK API documentation.) The sketch of the solution is as follows:

- The program's main loop launches the other thread which will generate the trigger. The GUI's main loop is also started (see **Fl::run()**).
- When the asynchronous thread decides that it wants to notify the main event loop, it writes to an fd, which is actually a local socket, see below.
- The "other end" is another fd addressing a peer socket. This fd generates a callback *in the context of the main event loop thread*.
- The callback function reads the indication from the peer fd, then it may perform any GUI update it wants.

The Implementation

Here I list some practical issues when implementing the solution.
First, Linux/Unix has a handy way to create two sockets that are in turn connected with each other and can be used for inter−thread communication: it's

```
socketpair(PF_UNIX, type, 0, int *sv);
```

It creates UNIX (or local) anonymous sockets and connects them, see the man pages for **unix**(7) and **socketpair**(2). The type can be **SOCK_DGRAM** which is a simple and effective solution. `*sv` points to an integer array of 2 elements. The programmer can choose which of the fd−s (that is, socket) will be used by the asynchronous thread and by the main loop. It's completely indifferent, as *the sockets are indistinguishable*. In our example the other thread will write to `sv[1]` using system call **write**(2) and the main event loop will obtain the written data from `sv[0]` using **read**(2) upon receiving the callback. It's important to read the same amount of data from the socket as the amount written. If the main loop reads less than that, the callback will be generated again in the next GUI event loop(s) until all the available data has been read.

Of course, the socket pair has to be connected first and registered for callback:

```
if(socketpair(PF_UNIX, SOCK_DGRAM, 0, sv) == −1)
  {
    fprintf(stderr, "Error creating socket pair: %d\n", errno);
  }
else
  {
    Fl::add_fd(sv[0], asyncEvent_cb, data);
    // ... Create and start the async thread.
  }
```

If we use the mechanism above in the simplest way we don't want to pass any data from the asynchronous thread to the main thread, only inform it about an event. In that case the async thread can do:

```
 int lbuf;
write(sv[1], &lbuf, 1);  // Write indicator token

... whereas the callback function may look like this:

void asyncEvent_cb(int fd, void *data)
{
 // ...
  int lbuf;
  read(fd, &lbuf, 1); // Read indicator token
  // ... Perform any widget update
}
```

Of course it's possible to have several asynchronous event sources generating events towards the main event loop. In that case we have to set up the corresponding number of socket pairs, i.e. *one socket pair for each event source*.

Note that the solution is thread−safe because the socket operations are thread−safe.

Author

This guide was written by **Zoltan Toth** <*tzp@mailbox.hu*>
Any comments are welcome.