How to Use the ROOT TThread Class

Jörn Adamczewski, Marc Hemberger

April 18, 2000

Contents

1 Summary

1.1 Disclaimer

The ROOT framework is not thread-safe in general. There are some known problems the core developer team of ROOT is aware of and is working on. For this reason, it should clearly be stated, that `TThread` is by no means a solution for everything. Be aware of possible problems, restrict your threads to distinct and 'simple' duties, and you will benefit from their use.

The `TThread`-class can be used on all platforms which provide a POSIX compliant thread implementation. On Linux Xavier Leroy's LinuxThreads implementation is widely used, but the `TThread` implementation should be usable on all platforms which provide pthreads.

1.2 Problems

1. **Linux Xlib on SMP machines** is not threadsafe yet, which may cause a crash during threaded graphics operations; this problem is independent of ROOT.
2. **Object instantiation:** no implicit locking mechanism for memory allocation and global root lists, the user has to take care for this on using threads.

2 Installation

For the time being (i.e. up to ROOT 2.23/xx), it is necessary to compile a threaded version of ROOT to achieve some very special treatment of the canvas operations. This will not be necessary later on anymore.

To compile ROOT, just do:

```
./configure linuxdeb2 --with-thread=/usr/lib/libpthread.so
```

This sets up the environment to use /usr/lib/libpthread.so as the Pthread library, and to switch on a define R__THREAD. This enables to have the thread specific treatment of `gPad`, and to get the `$ROOTSYS/lib/libThread.so`. Note: Parameter `linuxdeb2` has to be replaced with the appropriate ROOT keyword for your linux distribution.

```
gmake depend
```

```
gmake
```

This compiles ROOT as usual.

3 Understanding Threads

(adopted from the AIX 4.3 Programmer's Manual)

A thread is an independent flow of control that operates within the same address space as other independent flows of controls within a process. In most of UNIX systems, thread and process characteristics are grouped into a single entity called a process. Sometimes, threads are called "lightweight processes".

3.1 Threads and Processes

In traditional single-threaded process systems, a process has a set of properties. In multi-threaded systems, these properties are divided between processes and threads.

- Process Properties

  A process in a multi-threaded system is the changeable entity. It must be considered as an execution frame. It has all traditional process attributes, such as:

  - Process ID, process group ID, user ID, and group ID,
  - Environment,
  - Working directory.

  A process also provides a common address space and common system resources:

  - File descriptors
  - Signal actions
  - Shared libraries
  - Inter-process communication tools (such as message queues, pipes, semaphores, or shared memory).

- Thread Properties

  A thread is the schedulable entity. It has only those properties that are required to ensure its independent flow of control. These include the following properties:

  - Stack

- Scheduling properties (such as policy or priority)
- Set of pending and blocked signals
- Some thread-specific data (TSD).

An example of thread-specific data is the error indicator, errno. In multi-threaded systems, errno is no longer a global variable, but usually a subroutine returning a thread-specific errno value. Some other systems may provide other implementations of errno.

With respect to ROOT, a thread specific data is for example the gPad pointer, which is treated in a different way, whether it is accessed from any thread or the main thread.

Threads within a process must not be considered as a group of processes (even though in Linux each thread receives an own process id, so that it can be scheduled by the kernel scheduler). All threads share the same address space. This means that two pointers having the same value in two threads refer to the same data. Also, if any thread changes one of the shared system resources, all threads within the process are affected. For example, if a thread closes a file, the file is closed for all threads.

3.2 The Initial Thread

When a process is created, one thread is automatically created. This thread is called the initial thread or the main thread. It is the initial thread that executes the main routine in multi-threaded programs.

4 Threads related ROOT Classes

`TThread` This class implements threads. A thread is an execution environment much lighter than a process. A single process can have multiple threads. The actual work is done via the `TThreadImp` class (i. e. `TPosixThread`).

`TMutex` This class implements mutex locks. A mutex is a mutual exclusive lock. The actual work is done via the `TMutexImp` class (i. e. `TPosixMutex`)

`TCondition` This class implements a condition variable. Use a condition variable to signal threads. The actual work is done via the `TConditionImp` class (i. e. `TPosixCondition`).

`TSemaphore` This class implements a counting semaphore. Use a semaphore to synchronize threads. The actual work is done via the `TMutexImp` and `TConditionImp` classes.

5 TThread for pedestrians

5.1 TThread user function

To run a thread within ROOT, you have to follow these steps:

Initialization Add these lines to your rootlogon.C:

```
{
   printf("\nWelcome to the ROOT session\n\n");
   gSystem->Load("/usr/lib/libpthread.so");
   gSystem->Load("$ROOTSYS/lib/libThread.so");
}
```

This loads the library with the TThread class and the Pthread specific implementation file for Posix threads.

Define a function (e.g. `void* UserFun(void* UserArgs)`) that should run as a thread. See the examples *CalcPiThread.cxx* with *CalcPiThread.h* and *mhs3.cxx* with *mhs3.h*. For these functions we create (with the *Makefile.CalcPiThread* resp. *Makefile.mhs3*) a shared library to be loaded into the interactive ROOT session. How to go on with member functions will be explained in section 7.

Start your interactive ROOT session

Load the shared library with
`gSystem→Load("CalcPiThread.so");` or `gSystem→Load("mhs3.so");`

Create the thread instance (see also example *RunPi.C* or *RunMhs3.C*) by

```
TThread *th = new TThread(UserFun,UserArgs);
```

You can also give the thread a self defined name, then it might be called via

```
TThread *th = new TThread("MyThread", UserFun, UserArgs);
```

This method is appropriate to use threads in compiled code if you want to access them by name. Normally, the function name "UserFun" will be the name of the thread (this does not work if `TThread` constructor is called from compiled code).

Arguments to the thread function can be handed over via `UserArgs`-pointer. When you want to start a method of a class as a thread, you have to give the pointer to the class instance as `UserArgs` (see examples in section 7).

Run thread by `th→Run();`

Control `TThread::Ps();` - like UNIX ps command;

The *CalcPiThread*-example creates two threads, who calculate Pi to the desired precision, given by the number of intervals.

With the *mhs3*-example (example 9.2), you should be able to see a canvas with two pads on it. Both pads keep histograms updated and filled by three different threads.

6 TThread in more detail

6.1 CINT

CINT is not thread safe yet. CINT execution will block the execution of the threads untill CINT has finished.

6.2 Asynchronous Actions

Different threads can work simultaneously with the same object. Some actions can be dangerous. For example, when two threads create a histogram object, ROOT allocates memory and puts them to the same collection. If it happened to be at the same time, some clashes are possible. To avoid this, the user has to synchronize these actions:

```
TThread::Lock()    // Locking the following part of code

...                // Create an object, etc...

TThread::UnLock()  // Unlocking
```

The part of code between `Lock()` and `UnLock()` will be performed separately. No other thread can perform actions or access objects/collections at the same time. The `TThread::Lock()` and `TThread::UnLock()` methods internally use one global `TMutex` instance for locking. The user may also define his own `TMutex MyMutex` instance and may protect his asynchronous actions by calling `MyMutex.Lock()` and `MyMutex.UnLock()`.

6.3 Synchronous Actions: TCondition

To synchronize the actions of different threads the `TCondition` class can be applied which provides a signaling mechanism.

The `TCondition` instance must be accessible by all threads which want to use it, i.e. it should be a global object (or a member of the class which owns threaded methods, see section 7). To create a `TCondition` object, a `TMutex` instance for locking of `Wait` and `TimedWait` methods is required. One can pass the address of an external mutex to `TCondition` constructor:

```
TMutex MyMutex;
TCondition MyCondition(&MyMutex);
```

If `NULL` is passed, TCondition creates and uses its own internal mutex:

```
TCondition MyCondition(NULL);
```

From then on it is possible to use the following methods of synchronization:

1. `TCondition::Wait()` waits until any thread sends a signal of the same condition instance: `MyCondition.Wait()` reacts on `MyCondition.Signal()` or `MyCondition.Broadcast()`; `MyOtherCondition.Signal()` has no effect.

2. If several threads wait for the signal of the same `TCondition MyCondition`, at `MyCondition.Signal()` only one thread will react; to activate a further thread another `MyCondition.Signal()` is required, etc.

3. If several threads wait for the signal of the same `TCondition MyCondition`, at `MyCondition.Broadcast()` all threads waiting for `MyCondition` are activated at once. Remark: In some tests of `MyCondition` using an internal mutex, `Broadcast()` activated only one thread (probably depending whether `MyCondition` had been signalled before).

4. `MyCondition.TimedWait(secs,nanosecs)` waits for `MyCondition` until the *absolute* time in seconds and nanoseconds since beginning of the epoch (January, 1st,1970) is reached; to use relative timeouts "`delta`", it is required to calculate the absolute time at the beginning of waiting "`now`"; a possible coding could look like this:

   ```
   Ulong_t now,then,delta;              // seconds
   TDatime myTime;                      // root daytime class
   myTime.Set();                        // myTime set to ``now''
   now=myTime.Convert();                // to seconds since 1970
   then=now+delta;                      // absolute timeout
   wait=MyCondition.TimedWait(then,0);  // waiting
   ```

   Return value `wait` of `MyCondition.TimedWait` should be 0, if `MyCondition.Signal()` was received, and should be nonzero, if timeout was reached.

The conditions example (example 9.3) shows how three threaded functions are synchronized using `TCondition`: ROOT macro *condstart.C* starts the threads which are defined in a shared library (*conditions.cxx*, *conditions.h*).

6.4 Xlib connections

Usually Xlib is not thread safe. This means that calls to the X could fail, when it receives X-messages from different threads. At first this depends very crucially on the Xlib-version you have installed on your system. The only thing we can do here within ROOT is calling a special function `XInitThreads()` (which is part of the Xlib), which should (!) prepare the Xlib for the usage with threads.

To avoid further problems within ROOT some redefinition of the `gPad` pointer was done (that's the main reason for the recompilation). When a thread creates a `TCanvas`, actually this object is created in the main thread, and this is hidden from the user. Actions on the canvas are controlled via a function which returns a pointer to either thread specific data (TSD) or the main pointer. This mechanism works currently only for `gPad` and will also be implemented for other global Objects as e. g. `gVirtualX, gDirectory, gFile`.

6.5 Canceling a TThread

Canceling of a thread is a rather dangerous action. In `TThread` canceling is forbidden by default. But the user can change this default by calling `TThread::SetCancelOn()`. There are two cancellation modes:

Deferred - Set by `TThread::SetCancelDeferred()` (default): When the user knows safe places in his code where a thread can be canceled without interference with the rest of the system, he can define these points by invoking `TThread::CancelPoint()`. Then, if a thread is canceled, the cancelation is deferred up to the call of `TThread::CancelPoint()` and then the thread is canceled safely. There are some default cancel points for Pthreads implementation, e.g. any call of `TCondition::Wait()`, `TCondition::TimedWait()`, `TThread::Join()`.

Asynchronous - Set by `TThread::SetCancelAsynchronous()`: If the user is sure that his application is cancel safe, he could call:

```
TThread::SetCancelAsynchronous();
TThread::SetCancelOn();
// Now cancelation in any point is allowed.
...
...
//  return to default
TThread::SetCancelOff();
TThread::SetCancelDeferred();
```

To cancel a thread `TThread* th` the user can call:

th→Kill(); thread `th` is canceled

TThread::Kill(name); cancel by thread name;

TThread::Kill(tid); cancel by thread Id

th→Delete(); cancel thread and delete `th` when cancel finished.

Deleting of the thread instance by the operator `delete` is dangerous. Use `th→Delete` instead. C++ `delete` is safe only if thread is not running.

Often during the canceling some clean up actions must be made. To define clean up functions a user can call:

```
void UserCleanUp(void *arg){
// here the user cleanup is done
...
}

TThread::CleanUpPush(&UserCleanUp,arg); // push user function
                                        // into cleanup stack
                                        // ``last in, first out''

TThread::CleanUpPop(1); // pop user function out of stack
                        // and execute it,
                        // thread resumes after this call

TThread::CleanUpPop(0); // pop user function out of stack
                        // _without_ executing it
```

Note: `CleanUpPush` and `CleanUpPop` should be used as corresponding pairs like brackets; unlike Pthreads cleanup stack (which is *not* implemented here), `TThread` does not force this usage.

6.6 Finishing thread

When a thread returns from a user function the thread is finished. But it also can be finished by `TThread::Exit()`. Then, in case of Pthread "detached" mode, the thread vanishes completely.

By default, on finishing `TThread` executes the most recent cleanup function (`CleanUpPop(1)` is called automatically once).

7 Advanced: TThreads with classes

7.1 How to launch a class memberfunction as TThread

Consider a class `Myclass` with a memberfunction `void* Myclass::Thread0((void* arg)` that shall be launched as a thread. To start `Thread0` as a `TThread`, class `Myclass` may provide a method:

```
Int_t Myclass::Threadstart(){
 if(!mTh){
     mTh= new TThread("memberfunction",
                      (void(*) (void *))&Thread0,
                      (void*) this);
```

```
        mTh->Run();
        return 0;
        }
 return 1;
}
```

Here `mTh` is a `TThread*` pointer which is member of `Myclass` (should be initialized to 0 in the constructor). `TThread` constructor is called as for a plain C function (see section 5.1), except for two differences:

1. Memberfunction `Thread0` requires an explicit cast to `(void(*) (void *))` (this may cause a compiler warning like:
   Myclass.cxx:98: warning: converting from 'void (Myclass::)(void *)' to 'void *'*), which is annoying but harmless.

2. The pointer to the current instance of `Myclass`, i.e. `(void*) this`, has to be passed as first argument of the threaded function `Thread0` (C++ memberfunctions internally expect the `this` pointer as first argument to have access to class members of the same instance). Pthreads are made for simple C functions and do not know about `Thread0` being a memberfunction of a class. Thus you have to pass `this` by hand, if you want to access all members of the `Myclass` instance from the `Thread0` function.

Note: Method Thread0 must not be a virtual memberfunction, since the cast of `Thread0` to `void(*)` in TThread constructor may raise problems with C++ virtual function table. However, `Thread0` may *call* another virtual memberfunction `virtual void Myclass::Func0()` which then can be overridden in a derived class of `Myclass`. (see example 9.4).

Class `Myclass` may also provide a method to stop the running thread:

```
Int_t Myclass::Threadstop(){
 if(mTh){
        TThread::Delete(mTh);
        delete mTh;
        mTh=0;
        return 0;
 }
 return 1;
}
```

Example 9.4: Class `TThreadframe` (*TThreadframe.h*, *TThreadframe.cxx*) is a simple example of a framework class managing up to four threaded methods. Class `TMhs3` (*TMhs3.h*, *TMhs3.cxx*) inherits from this base class, showing the *mhs3* example 9.2 (*mhs3.h*, *mhs3.cxx*) within a class.

Makefile of example builds shared libraries *libTThreadframe.so* and *libTMhs3.so*. These are either loaded and executed by root macro *TMhs3demo.C*, or are linked against executable *TMhs3run.cxx*.

8 Thanks and Credits

To:

Victor Perevoztchikov

Fons Rademakers

Rene Brun

for their support in reimplementing and putting threads back to ROOT.

9 List of Example files

9.1 Example *CalcPiThread*

- Makefile.CalcPiThread
- CalcPiThread.h
- CalcPiThreadLinkDef.h
- CalcPiThread.cxx
- rootlogon.C
- RunPi.C

9.2 Example *mhs3*

- Makefile.mhs3
- mhs3.h
- mhs3LinkDef.h
- mhs3.cxx
- rootlogon.C
- RunMhs3.C

9.3 Example *conditions*

- Makefile.conditions
- conditions.h
- conditionsLinkDef.h
- conditions.cxx
- condstart.C

9.4 Example *TMhs3*

- Makefile.TMhs3
- TThreadframe.h
- TThreadframeLinkDef.h
- TThreadframe.cxx
- TMhs3.h
- TMhs3LinkDef.h
- TMhs3.cxx
- TMhs3run.cxx
- TMhs3demo.C

10 Glossary

(adopted from the description of the RogueWave Threads.h++ package)

Process A process is a program that is loaded into memory and prepared for execution. Each process has a private address space. Processes begin with a single thread.

Thread A thread of control, or more simply, a thread, is a sequence of instructions being executed in a program. A thread has a program counter and a private stack to keep track of local variables and return addresses. A multithreaded process is associated with one or more threads. Threads execute independently. All threads in a given process share the private address space of that process.

Concurrency Concurrency exists when at least two threads are in progress at the same time. A system with only a single processor can support concurrency by switching execution contexts among multiple threads.

Parallelism Parallelism arises when at least two threads are executing simultaneously. This requires a system with multiple processors. Parallelism implies concurrency, but not vice-versa.

Reentrant A function is reentrant if it will behave correctly even if a thread of execution enters the function while one or more threads are already executing within the function. These could be either the same thread, in the case of recursion, or different threads, in the case of concurrency.

Thread-specific data Thread-specific data (TSD) is also known as thread-local storage (TLS). Normally, any data that has lifetime beyond the local variables on the thread's private stack are shared among all threads within the process. Thread-specific data is a form of static or global data that is maintained on a per-thread basis. That is, each thread gets its own private copy of the data.

Synchronization Left to their own devices, threads execute independently. Synchronization is the work that must be done when there are, in fact, interdependencies that require some form of communication among threads. Synchronization tools include mutexes, semaphores, condition variables, and other variations on locking.

Critical Section A critical section is a section of code that accesses a non-sharable resource. To ensure correct code, only one thread at a time may execute in a critical section. In other words, it is a section of code which is not reentrant.

Mutex A mutex, or mutual exclusion lock, is a synchronization object with two states locked and unlocked. A mutex is usually used to ensure that only one thread at a time executes some critical section of code. Before entering a critical section a thread will attempt to lock the mutex which guards that section. If the mutex is already locked, the thread will block until the mutex is unlocked, at which time it will lock the mutex, execute the critical section, and unlock the mutex upon leaving the critical section.

Semaphore A semaphore is a synchronization mechanism that starts out initialized to some positive value. A thread may ask to wait on a semaphore in which case the thread blocks until the value of the semaphore is positive. At that time the semaphore count is decremented and the thread continues. When a thread releases semaphore, the semaphore count is incremented. Counting semaphores are useful for coordinating access to a limited pool of some resource.

Readers/Writer Lock A multiple-readers, single-writer lock is one that allows simultaneous read access by many threads while restricting write access to only one thread at a time. When any thread holds the lock for reading, other threads can also acquire the lock reading. If one thread holds the lock for writing, or is waiting to acquire the lock for writing, other threads must wait to acquire the lock for either reading or writing.

Condition Variable Use a condition variables in conjunction with a mutex lock to automatically block threads until a particular condition is true.

Multithread safe levels A possible classification scheme to describe thread-safety of libraries:

Level I All public and protected functions are reentrant. The library provides protection against multiple threads trying to modify static and global data used within a library. The developer must explicitly lock access to objects shared between threads. No other thread can write to a locked object unless it is unlocked. The developer needs to lock local objects. The spirit, if not the letter, of this definition requires the user of the library only to be familiar with the semantic content of the objects in use. Locking access to objects that are being shared due to extra-semantic details of implementation (for example, copy-on-write) should remain the responsibility of the library.

Level II All public and protected functions are reentrant. The library provides protection against multiple threads trying to modify static and global data used within the library. The preferred way of providing this protection is to use mutex locks. The library also locks an object before writing to it. The developer is not required to explicitly lock or unlock a class object (static, global or local) to perform a single operation on the object. Note that even multithread safe level II hardly relieves the user of the library from the burden of locking.

Deadlock A thread suffers from deadlock if it is blocked waiting for a condition that will never occur. Typically, this occurs when one thread needs to access a resource that is already locked by another thread, and that other thread is trying to access a resource that has already been locked by the first thread. In this situation, neither thread is able to progress; they are deadlocked.

Multiprocessor A hardware system with multiple processors or multiple, simultaneous execution units.