Contents

Chapter 1

Threads

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 UNIX systems, thread and process characteristics are grouped into a single entity called a process. Sometimes, threads are called "lightweight processes".

Note: This introduction is adapted from the AIX 4.3 Programmer's Manual.

1.1 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.

1.1.1 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)

1.1.2 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.

1.1.3 The Initial Thread

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

Note: At the end of this chapter is a glossary of thread specific terms

1.2 Implementation of Threads in ROOT

The **TThread** class has been developed to provide a platform independent interface to threads for ROOT.

1.2.1 Installation

For the time being, it is still necessary to compile a threaded version of ROOT to enable some very special treatments of the canvas operations. We hope that this will become the default later.

To compile ROOT, just do (for example on a debian Linux):

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

This configures and builds ROOT using `/usr/lib/libpthread.so` as the `Pthread` library, and defines `R__THREAD`.

This enables the thread specific treatment of ***gPad***, and creates `$ROOTSYS/lib/libThread.so`.

Note: The parameter linuxdeb2 has to be replaced with the appropriate ROOT keyword for your platform.

1.2.2 Classes

TThread class implements threads . The platform dependent implementation is in the **TThreadImp** class and its descendant classes (e.g. **TPosixThread**).

TMutex class implements `mutex` locks. A mutex is a mutually exclusive lock. The platform dependent implementation is in the **TMutexImp** class and its descendant classes (e.g. **TPosixMutex**)

TCondition class implements a condition variable. Use a condition variable to signal threads. The platform dependent implementation is in the **TConditionImp** and **TPosixCondition** classes .

TSemaphore class implements a counting semaphore. Use a semaphore to synchronize threads. The platform dependent implementation is in the **TMutexImp** and **TConditionImp** classes.

1.2.3 TThread for Pedestrians

To run a thread in ROOT, follow these steps:

1. Initialization

Add these lines to your `rootlogon.C`:

```
{
   ...
   // The next line may be unnecessary on some platforms
   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.

2. Coding

Define a function (e.g. **void* UserFun(void* UserArgs)**) that should run as a thread. The code for the examples is at the web site of the authors (Jörn Adamczewski, Marc Hemberger). After downloading the code from this site, you can follow the example below:

http://www-linux.gsi.de/~go4/HOWTOthreads/howtothreadsbody.html

3. Loading

Start an interactive ROOT session. Load the shared library:

```
root[] gSystem->Load("mhs3.so");              // or
root[] gSystem->Load("CalcPiThread.so");
```

4. Creating

Create a thread instance (see also example **RunMhs3.CorRunPi.C**) with:

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

When called from the interpreter, this gives the name "**UserFun**" to the thread. This name can be used to retrieve the thread later. However, when called from compiled code, this method does not give any name to the thread. So give a name to the thread in compiled use:

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

You can pass arguments to the thread function using the **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**.

5. Running

```
root[] th->Run();
root[] TThread::Ps(); // like UNIX ps c.ommand;
```

With the **mhs3** example, you should be able to see a canvas with two pads on it. Both pads keep histograms updated and filled by three different threads. With the **CalcPi** example, you should be able to see two threads calculating Pi with the given number of intervals as precision.

1.2.4 TThread in More Details

Cling is not thread safe yet, and it will block the execution of the threads until it has finished executing.

1.2.4.1 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 happens at the same time, the results are undetermined. To avoid this problem, the user has to synchronize these actions with:

```
TThread::Lock()    // Locking the following part of code
...                // Create an object, etc...
TThread::UnLock()  // Unlocking
```

The code between `Lock()` and `UnLock()` will be performed uninterrupted. No other threads can perform actions or access objects/collections while it is being executed. The methods `TThread::Lock()`and **`TThread::UnLock()`** internally use a global `TMutex` instance for locking.

The user may also define their own **`TMutex`** `MyMutex` instance and may locally protect their asynchronous actions by calling `MyMutex.Lock()` and `MyMutex.UnLock()`.

1.2.4.2 Synchronous Actions: TCondition

To synchronize the actions of different threads you can use the **`TCondition`** class, which provides a signaling mechanism. The **`TCondition`** instance must be accessible by all threads that need to use it, i.e. it should be a global object (or a member of the class which owns the threaded methods, see below). To create a **`TCondition`** object, a **`TMutex`** instance is required for the Wait and `TimedWait` locking methods. One can pass the address of an external mutex to the **`TCondition`** constructor:

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

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

```
TCondition MyCondition(0);
```

You can now use the following methods of synchronization:

- `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.
- If several threads wait for the signal from the same **`TCondition`** `MyCondition`, at `MyCondition.Signal()` only one thread will react; to activate a further thread another `MyCondition.Signal()` is required, etc.
- If several threads wait for the signal from the same **`TCondition`** `MyCondition`, at `MyCondition.Broadcast()` all threads waiting for `MyCondition` are activated at once.

In some tests of `MyCondition` using an internal mutex, `Broadcast()` activated only one thread (probably depending whether `MyCondition` had been signaled before).

- `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"; for example:

```
Ulong_t now,then,delta;                  // seconds
TDatime myTime;                          // root daytime class
myTime.Set();                            // myTime set to "now"
now=myTime.Convert();                    // to seconds since 1970
```

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

1.2.4.3 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. The actual result depends strongly on which version of `Xlib` has been 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**, this object is actually created in the main thread; this should be transparent to the user. Actions on the canvas are controlled via a function, which returns a pointer to either thread specific data (TSD) or the main thread pointer. This mechanism works currently only for ***gPad****,`gDirectory`*, ***gFile*** and will be implemented soon for other global Objects as e.g. ***gVirtualX***.

1.2.4.4 Canceling a TThread

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

1.2.4.5 Deferred

Set by `TThread::SetCancelDeferred()` (default): When the user knows safe places in their code where a thread can be canceled without risk for the rest of the system, they can define these points by invoking **`TThread::CancelPoint()`**. Then, if a thread is canceled, the cancellation 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 the `TCondition::Wait()`, **`TCondition`**`::TimedWait()`, `TThread::Join()`.

1.2.4.6 Asynchronous

Set by `TThread::SetCancelAsynchronous()`: If the user is sure that their application is cancel safe, they 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` call:

```
th->Kill();
```

To cancel by thread name:

```
TThread::Kill(name);
```

To cancel a thread by ID:

```
TThread::Kill(tid);
```

To cancel a thread and delete `th` when cancel finished:

```
th->Delete();
```

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 taken. To define clean up functions use:

```
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.

1.2.4.7 Finishing thread

When a thread returns from a user function the thread is finished. It also can be finished by `TThread::Exit()`. Then, in case of `thread`-detached mode, the thread vanishes completely. By default, on finishing **`TThread`** executes the most recent cleanup function (`CleanUpPop(1)` is called automatically once).

1.3 Advanced TThread: Launching a Method in a Thread

Consider a class `Myclass` with a member function that shall be launched as a thread.

```
void* Myclass::Thread0((void* arg)
```

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`and should be initialized to 0 in the constructor. The **`TThread`** constructor is called as when we used a plain C function above, except for the following two differences.

First, the member function Thread0 requires an explicit cast to `(void(*) (void *))`. This may cause an annoying but harmless compiler warning:

```
Myclass.cxx:98: warning:
              converting from "void (Myclass::*)(void *)"to "void *" )
```

Strictly speaking, `Thread0` must be a static member function to be called from a thread. Some compilers, for example `gcc` version 2.95.2, may not allow the `(void(*) (void*))s` cast and just stop if `Thread0` is not static. On the other hand, if `Thread0` is static, no compiler warnings are generated at all. Because the `'this'` pointer is passed in `'arg'` in the call to `Thread0(void *arg)`, you have access to the instance of the class even if `Thread0` is static. Using the `'this'` pointer, non static members can still be read and written from `Thread0`, as long as you have provided Getter and Setter methods for these members. For example:

```
Bool_t state = arg->GetRunStatus();
arg->SetRunStatus(state);
```

Second, 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++ member functions internally expect 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 member function of a class. Thus, you have to pass this information by hand, if you want to access all members of the `Myclass` instance from the Thread0 function.

Note: Method Thread0 cannot be a virtual member function, since the cast of `Thread0` to `void(*)` in the **`TThread`** constructor may raise problems with C++ virtual function table. However, Thread0 may call another virtual member function virtual void `Myclass::Func0()` which then can be overridden in a derived class of `Myclass`. (See example `TMhs3`).

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 `TMhs3:` 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 8.1 (`mhs3.h, mhs3.cxx`)within a class. The `Makefile` of this example builds the shared libraries `libTThreadframe.so` and `libTMhs3.so`. These are either loaded or executed by the ROOT script `TMhs3demo.C,` or are linked against an executable: `TMhs3run.cxx`.

1.3.1 Known Problems

Parts of the ROOT framework, like the interpreter, are not yet thread-safe. Therefore, you should use this package with caution. If you restrict your threads to distinct and 'simple' duties, you will able to benefit from their use. The **`TThread`** class is available on all platforms, which provide a POSIX compliant thread implementation. On Linux, Xavier Leroy's Linux Threads implementation is widely used, but the **`TThread`** implementation should be usable on all platforms that provide `pthread`.

Linux Xlib on SMP machines is not yet thread-safe. This may cause crashes during threaded graphics operations; this problem is independent of ROOT.

Object instantiation: there is no implicit locking mechanism for memory allocation and global ROOT lists. The user has to explicitly protect their code when using them.

1.4 The Signals of ROOT

The list of default signals handled by ROOT is:

```
kSigChildkSigPipe
kSigBuskSigAlarm
kSigSegmentationViolationkSigUrgent
kSigIllegalInstructionkSigFloatingException
kSigSystemkSigWindowChanged
```

The signals ***kSigFloatingException****,*`kSigSegmentationViolation`*, **kSigIllegalInstruction*, and*`kSigBus`* cause the printing of the ****** Break ****** message and make a long jump back to the ROOT prompt. No other custom **`TSignalHandler`** can be added to these signals.

The ***kSigAlarm* signal handles asynchronous timers. The***`kSigWindowChanged`* signal handles the resizing of the terminal window. The other signals have no other behavior then that to call any registered **`TSignalHandler`*****::Notify()*** .

When building in interactive application the use of the **`TRint`** object handles the `kSigInterrupt` signal. It causes the printing of the message: ****** Break *** keyboard interrupt*** and makes a long jump back to the ROOT command prompt. If no **`TRint`** object is created, there will be no `kSigInterrupt` handling. All signals can be reset to their default

UNIX behavior via the call of `TSytem::ResetSignal()`. All signals can be ignored via `TSytem::IgnoreSignal()`. The `TSytem::IgnoreInterrupt()` is a method to toggle the handling of the interrupt signal. Typically it is called to prevent a `SIGINT` to interrupt some important call (like writing to a ROOT file).

If `TRint` is used and the default ROOT interrupt handler is not desired, you should use `GetSignalHandler()` of `TApplication` to get the interrupt handler and to remove it by `RemoveSignalHandler()`of `TSystem` .

1.5 Glossary

The following glossary is adapted from the description of the Rogue Wave `Threads.h++` package.

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.

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

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 the same thread, in the case of recursion, or different threads, in the case of concurrency.

`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.

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.

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, the section is not reentrant.

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.

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-reader, 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.

Use a **`condition variable`** 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:

- 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.

- 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.

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.

A **`multiprocessor`** is a hardware system with multiple processors or multiple, simultaneous execution units.

- Examples can be found at http://www-linux.gsi.de/~go4/HOWTOthreads/howtothreadsbody.html (the thread authors' web site - Jörn Adamczewski and Marc Hemberger).