CS107 Thread Package Documentation

Thread package and handout was written by Julie Zelenski

Threads

We will do our concurrent programming work using a platform-specific thread packages that run on the **elaine**s and the **pod**s. Note that unlike the earlier straight C programming assignments, you won't be able write and debug it on a Mac/PC and just port it to UNIX as the last step because the thread library only runs on the campus machines. Our simple interface is defined in **thread_107.h** which itself is built on the primitives in Solaris **thread.h**.

Editorial Comment

This is a classic standards problem: there is nothing especially difficult about defining a standard threads package. There are many, many out there, and they all do pretty much the same thing in similar by not identical ways. Sadly, without a standard interface, there is no way you can write code that is portable across all machines. This is exactly the situation standards committees are supposed to solve, but it is a politically difficult one given there is so much inertia invested in all the existing yet slightly incompatible interfaces.

The code for our package resides in the **/usr/class/cs107/other/thread_library** directory. The source files are **thread_107.c** and **thread_107.h**. The thread package has been compiled and saved as a library in the class directory. You need to include the header file **thread_107.h** to see the prototypes and must link with the **libthread_107.a** library to bring in the necessary code. You need not be concerned with the library source, although if you are interested in how some of the routines are implemented, feel free to browse through it. We will provide a **Makefile** which puts **cs107/include** in the include path and **cs107/lib** in the link path as well as linking with the necessary libraries.

```
void  InitThreadPackage(bool  traceFlag);
```

This procedure initializes the thread package. Call it once (and only once) at the beginning of your main program before you call any other functions from the thread package. The Boolean **traceFlag** controls whether debug tracing output is produced on thread exit and semaphore wait and signal activity. Turning on tracing may be useful during debugging, although the output can be prodigious.

Thread functions

```
void ThreadNew(const char *debugName, void (*func)(), int nArg, ...);
```

This procedure creates a new thread to execute the function `func`. `ThreadNew` is a variable-argument function, somewhat like `printf`, that allows you to specify different numbers and types of arguments depending on what values you need to pass to the new thread's starting function. `nArg` is the number of arguments, it should be zero if no arguments are passed. At most 8 arguments can be passed. After `nArg`, you give the arguments in order that you want passed to the starting function. Due to C's compile-time size constraints, all of the arguments must be integers or pointers (pointers can point to any type though). Like all variable-argument functions, you must be extra-careful. If you pass `3` for `nArg` yet only provide `2` arguments, the compiler's type-checking will not detect this and nasty things will happen at runtime. The `debugName` is used to associate a name with each thread so that you can more easily distinguish threads when debugging, it is also used to identify threads when tracing is on. `ThreadNew` makes its own copy of whatever you pass as the `debugName` string.

The `ThreadNew` function can be called from within threads earlier created by `ThreadNew`., as well as from main. You don't get a returned thread value from `ThreadNew`, instead it creates a new thread and adds it to the queue of threads to be scheduled. No threads will execute until `RunAllThreads` is called, at which point all threads will start being scheduled. Once `RunAllThreads` has been called, any further calls to `ThreadNew` will create threads that are immediately available to start executing. Whenever a thread gets to the end of its function, the thread exits on its own.

```
void ThreadSleep(int microSecs);
```

This function causes the currently executing thread to go into a sleep state for the specified number of microseconds. Other threads continue to run while the thread sleeps. The thread is guaranteed to not run for at least the specified interval, however, it may be longer if it wakes up and other threads are scheduled ahead of it. This function is typically used to vary the execution pattern of the program or similar some time-consuming behavior.

```
const char *ThreadName(void);
```

Returns the name of the currently executing thread as set earlier with `ThreadNew`. This can be useful when debugging.

```
void RunAllThreads(void);
```

Your main function has a higher priority than any functions created with `ThreadNew`, so no spawned threads will run until you call the `RunAllThreads` function. At that time, the priority of main becomes less than all the new threads, so it will not do anything more until all the new threads exit. The function only returns after all threads (including those started from within other threads) have finished executing.

Our package uses a modified round-robin scheduling algorithm. It cycles the queue giving each thread a time slice, but injects a bit of randomness in the determining the size of the time slice and the order threads are chosen from the queue. In truly generic concurrent programming, you cannot assume any sort of round-robin scheduling.

Semaphore functions

```
typedef   struct   SemaphoreImplementation   *Semaphore;
```

`Semaphore` is an abstract type that exported from the thread package to provide a generalized semaphore abstraction.

```
Semaphore   SemaphoreNew(const   char   *debugName,   int   initialValue);
```

Allocate and return a new semaphore variable. The semaphore is initialized to the given starting value. The `debugName` is used to associate a name with each `Semaphore` for debugging and is used in the tracing routines to identify `Semaphore`s. It makes its own copy of the string passed as the name.

```
const   char   *SemaphoreName(Semaphore   s);
```

Returns the name of the semaphore as set earlier with `SemaphoreNew`. This can be useful for debugging.

```
void   SemaphoreWait(Semaphore   s);
```

This function waits until the semaphore has a positive value and then decrements its value, indicating its availability has decreased. This function corresponds to the `P` operation for those of you who think in Dutch.

```
void   SemaphoreSignal(Semaphore   s);
```

This function increments the given semaphore, signalling its availability to other potentially waiting threads. This function corresponds to the `V` operation for you Dutch thinkers.

```
void   SemaphoreFree(Semaphore   s);
```

This function deallocates a semaphore and all its resources. You should only do this after all threads are completely done using the semaphore.

Library lock functions and MT-safeness of library functions

In addition to take care to synchronize access to your own shared data, you also need to be cautious about use of routines in the system libraries. On Solaris, the man page for each library calls has an indication of whether it is MT-safe or not. If it is listed as safe, then you can freely call it without any special synchronization. For example, the `malloc`/`realloc`/`free` functions are all MT-safe. This means you don't have to worry

about allocations from one thread clobbering those from another and you can call **malloc** from multiple threads without any extra precautions. Another important case is the **printf**/**fprintf** family of functions. On the latest version of Solaris, these functions lock access to the file descriptor during a call to make sure that two threads trying to **fprintf** to the same file won't have their output garbled together by ensuring that each **fprintf** call goes through in its entirety before the next one can begin.

Some calls in the system libraries are not re-entrant, and multiple simultaneous calls to the routines can result in unexpected responses, core dumps, or corruption of shared global data. If a function is listed as MT-unsafe in its man page, you should take precautions to ensure that multiple threads cannot call the function at the same time. Setting up a binary semaphore to use as a lock before calling such functions is usually the way to handle this. To make easier on you, we provide a pre-configured binary semaphore just for this purpose along with these functions to use it:

```
void   AcquireLibraryLock(void);
```

This function can be used before code that calls a non-MT-safe routine (such as **random**). The man page for any library function should tell you whether any particular call is MT-safe.

```
void   ReleaseLibraryLock(void);
```

After finishing a library function that was called under the library lock, you should release the lock to allow other threads access to the standard library.

```
#define   PROTECT(code)   {                                 \\
                        AcquireLibraryLock();        \\
                        code;                        \\
                        ReleaseLibraryLock();        \\
                    }
```

This macro has been defined to allow you to concisely program an unsafe library call. For example, **PROTECT(result = random();)** This ensure the **random** will go through in its entirely before any other thread (which must also correctly use the **PROTECT** macro) can attempt to use the routine, and thus avoids clashes with the function.

In the work you will do for this class, you will interact with few (or possibly no) MT-unsafe library functions. Even though issues may come up only rarely, never assume anything. Be sure to verify the safeness of all library routines used in any concurrent program and take precautions for those routines that aren't thread-friendly.

Debugging functions

Lastly, our thread library also includes two debugging routines. These can be useful to call when stopped in **gdb** and you'd like to learn what the current state of the world is.

You can also use the `gdb` function "info threads" but its information is sometimes hard to match up with our threads since **gdb** lists them by number, not name, and doesn't print any internal status information.

```
void   ListAllThreads(void);
```

Lists all current threads by debug name and current status (ready to run, blocked on some condition, etc.) Probably most useful when in a deadlock situation and need figure out who is stuck and why. When you ask the debugger to execute a function (any function, not just this one), it lets all threads run during the processing, so it is normal to expect the debug printing to be interspersed with the activity of the other threads running. Also sometimes **gdb** spews out a warning about being "unable to restore previously selected frame" when you invoke this function. You can safely ignore the message, it's harmless.

```
void   ListAllSemaphores(void);
```

Lists all semaphores by debug name and shows their current value and number of threads currently blocked on that semaphore. Like **ListAllThreads**, you may get unexpected effects during the printout due to interactions with other threads or the debugger.

One thing that students find surprising that when you are stopped in **gdb** and you try to `step`/`next` or call a function, all threads get to run, not just the one you currently investigating. It is sort of a Heisenberg-like situation: the more you snoop around inside the program to observe what is going on, the more you change the execution path and the results. There are commercial (French for expensive) debuggers that have some very nice facilities for debugging threads, but unfortunately we have just good old **gdb**, which only has minimal thread support. Sadly, there are quirks in **gdb** that sometimes make it difficult to reliably stop a concurrent program and poke around. At times, you may have to resort to more primitive debugging strategies (**printf**s and the like) when **gdb** is hindering more than helping.