1.5 Pthreads

A POSIX thread is created by calling function *pthread_create()*:

```
int pthread_create(
  pthread_t* thread,                  // thread ID
  const pthread_attr_t* attr,  // thread attributes
  void* (*start)(void*),              // starting address of the function to run
  void* arg                           // an argument to be passed to the thread
);
```

The parameters for *pthread_create()* are as follows:

- *thread*: the address of a memory location that will receive an identifier assigned to the thread if creation is successful.
- *attr*: the address of a variable of type *pthread_attr_t*, which can be used to specify certain attributes of the created thread.
- *start*: the (address of the) function that the thread will execute. (This function plays the same role as the *run()* method in Java.)
- *arg*: an argument to be passed to the thread.

If *pthread_create()* (or some other Pthreads function) is successful, it returns 0; otherwise it returns an error code from the <errno.h> header file.

The program in Listing 1.4 is a C++/Pthreads version of the Java program in Listing 1.1.

- A Pthreads program must include the standard header file <*pthread.h*>
- Array *threadArray* stores the Pthreads IDs for the two threads created in *main()*. Thread IDs are of type *pthread_t*. (We are using an array to store thread IDs to be consistent with the Win32 program.)
- Each thread executes the code in function *simpleThread()*, which displays the IDs assigned by the user. The IDS are integers that are supplied as the fourth argument on the call to function *pthread_create()*. Function *pthread_create()* forwards the IDs as arguments to thread function *simpleThread()* when the threads are created.

```
#include <iostream>
#include <pthread.h>
#include <errno.h>

void PrintError(char* msg, int status, char* fileName, int lineNumber) {
        std::cout << msg << ' ' << fileName << ":" << lineNumber
            << " - " << strerror(status) << std::endl;
}

void* simpleThread (void* myID) { // myID is the fourth argument of pthread_create ()
        std::cout << "Thread " << (long) myID << " is running" << std::endl;
        return NULL;                // implicit call to pthread_exit(NULL);
}

int main() {
    pthread_t threadArray[2];           // array of thread IDs
    int status;                         // error code
    pthread_attr_t threadAttribute;     // thread attribute

    status = pthread_attr_init(&threadAttribute);      // initialize attribute object
        if (status != 0) { PrintError("pthread_attr_init failed at", status, __FILE__,
      __LINE__); exit(status);}
        // set the scheduling scope attribute
        status = pthread_attr_setscope(&threadAttribute,
      PTHREAD_SCOPE_SYSTEM);
      if (status != 0) { PrintError(…); exit(status);}

        // Create two threads and store their IDs in array threadArray
        status = pthread_create(&threadArray[0], &threadAttribute, simpleThread,
      (void*) 1L);
      if (status != 0) { PrintError(…); exit(status);}
        status = pthread_create(&threadArray[1], &threadAttribute, simpleThread,
      (void*) 2L);
      if (status != 0) { PrintError(…); exit(status);}

        status = pthread_attr_destroy(&threadAttribute); // destroy the attribute object
      if (status != 0) { PrintError(…); exit(status);}

        status = pthread_join(threadArray[0],NULL); // wait for threads to finish
      if (status != 0) { PrintError(…); exit(status);}
        status = pthread_join(threadArray[1],NULL);
       if (status != 0) { PrintError(…); exit(status);}
}
```

Listing 1.4 A simple concurrent program using C++/Pthreads.

Threads have attributes that can be set when they are created. In most cases the default attributes are sufficient.

Attributes are set by declaring and initializing an attributes object. Each attribute in the attributes object has a pair of functions for reading (get) and writing (set) its value.

In Listing 1.4, the attribute object *threadAttribute* is initialized by calling *pthread_attr_init()*:

- The scheduling scope attribute is set to *PTHREAD_SCOPE_SYSTEM* by calling *pthread_attr_setscope().*
- The address of *threadAttribute* is passed as the second argument on the call to *pthread_create()*.

The main thread must wait for the two threads it created to complete before it exits the *main()* function. It does this by calling function *pthread_join()* twice. The first argument to *pthread_join()* is the thread ID of the thread to wait on. The second argument is the address of a variable that will receive the return value of the thread.

```
void* simpleThread (void* myID) {
   std::cout << "Thread " << (long) myID << " is running" << std::endl;
   return myID; // implicit call to pthread_exit(myID);
}
```

We can use function *pthread_join()* to capture the values returned by the threads:

```
long result1, result2; // these variables will receive the return values
status = pthread_join(threadArray[0],(void**) &result1);
if (status != 0) { /* … */ }
status = pthread_join(threadArray[1],(void**) &result2);
if (status != 0) { /* … */ }
std::cout << “thread1:” << (long) result1 << “ thread2:”
          << (long) result2 << std::endl;
```

A thread usually terminates by returning from its thread function. What happens after that depends on whether the thread has been "detached":

- Threads that are terminated but not detached retain system resources that have been allocated to them. This means that the return values for undetached threads are still available and can be accessed by calling *pthread_join()*.
- Detaching a thread allows the system to reclaim the resources allocated to that thread. But a detached thread cannot be joined.

You can detach a thread anytime by calling function *pthread_detach()*.

```
status = pthread_detach(threadArray[0]);
```

Or the first thread can detach itself:

```
status = pthread_detach(pthread_self());
```

Calling *pthread_join*(threadArray[0],NULL) also detaches the first thread.

If you create a thread that definitely will not be joined, you can use an attribute object to ensure that when the thread is created it is already detached:

```
int main() {
    pthread_t threadArray[2];           // array of thread IDs
    int status;                         // error code
    pthread_attr_t threadAttribute;     // thread attribute
    status = pthread_attr_init(&threadAttribute);   // initialize the attribute object
    if (status != 0) { /*… */ }
    // set the detachstate attribute to detached
    status = pthread_attr_setdetachstate(&threadAttribute,
            PTHREAD_CREATE_DETACHED);
    if (status != 0) { /*…*/ }
    // create two threads in the detached state
    status=pthread_create(&threadArray[0],&threadAttribute,simpleThread,(void*)1L);
    if (status != 0) { /* … */ }
    status=pthread_create(&threadArray[1],&threadAttribute,simpleThread,(void*)2L);
    if (status != 0) { /* … */ }
    status = pthread_attr_destroy(&threadAttribute); // destroy the attribute object
    if (status != 0) { /*…*/ }
    // allow all threads to complete (Note: pthread_join() cannot be used here)
    pthread_exit(NULL);
}
```