Threads

ECE2893

Lecture 13

Threads

1. In order to implement concurrent algorithms, such as the parallel bubble sort discussed previously, we need some way to say that we want more than one CPU executing our program.
2. This is done by creating one or more *threads*.
3. A *thread* is nothing more than a sequence of C/C++ instructions.
4. All of our programs to date in fact use a thread!
    - All C/C++ programs start out with exactly one thread.
    - The `main` function is the starting point for the thread.
5. More threads can be created, using the *pthreads* library.
6. When creating a thread, the *starting function* is specified, and any arguments for that function.
7. The details of the *pthreads* library are quite complex and tedious.
8. We will be using a simplified, and reduced functionality, version called *gthreads*.

Using *gthreads*

DISCLAIMER!

1. The *gthreads* library is not part of the standard C or C++ library.
2. It was created by the ECE2893 instructor to simplify the way that threads are created and managed.
3. The definition of the *gthreads* functions are in `gthreads.h`
4. The implementation of the *gthreads* functions are in `gthreads.cc`
5. The *gthreads* library has significantly reduced functionality versus the *pthreads* library.
6. However, *gthreads* has sufficient functionality for the ECE2893 class assignments.

Creating a Thread using *gthreads*

1. A program can ask for another thread of execution by using the `CreateThread` function defined by the *gthreads* library.
2. The first argument to `CreateThread` is required, and is the name of the starting function for the new thread.
   - The starting function can be any function already defined in the C/C++ program.
3. Following the function name argument, there can be up to four more arguments to `CreateThread`, of any type.
   - These arguments are passed by value to the thread starting function when the thread starts executing.
   - The types of the arguments *must* match the types of the arguments in the thread starting function.
   - If they don't match, a compiler error occurs.
4. See the next slide for an example.

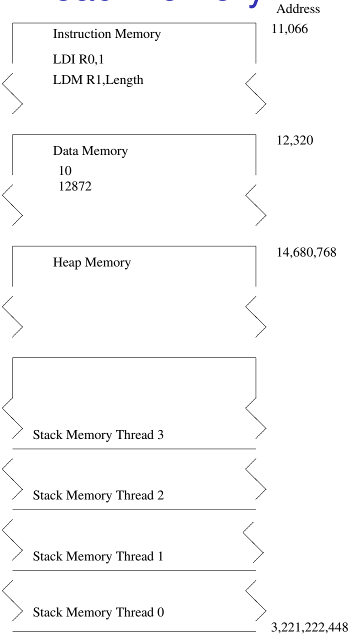
CreateThread Example

```
// Illustrate using CreateThread in gthreads
// George F. Riley, Georgia Tech, ECE2893, Spring 2011

#include "gthread.h"   // Must be included to use the gthreads library

void BubbleSort(int* d, int startingPoint, int length)
{ // This is the thread starting point
  // This is where, in this example, the sorting of array d will be done
}

const int nThreads = 4;       // Number of threads desired
const int maxSize = 512000;  // Largest sort size

int main()
{
  int d[maxSize];              // Array to be sorted
  int start = 0;               // Starting point of sub-array
  int length = (omitted);      // Length of sub-array
  for (int k = 0; k < nThreads; ++k)
    { // Create each of the four sorting threads
      CreateThread(BubbleSort, d, start, length);
      // need more code here..(omitted)
    }
  // At this point all threads are created
  // Need more code here (shown later)
}
```

Thread Memory Use

1. The previous example created 4 threads, each starting at function `BubbleSort`, and passing three parameters to `BubbleSort`.
2. The 4 threads (in this example) have access to exactly the same memory (with one exception).
    - The thread can call any function that exists in the program.
    - The thread can read or write any global variable that exists in the program.
    - However, *EACH THREAD HAS A PRIVATE STACK*
    - This means that local variables within subroutines are not shared between threads.

Thread Memory



Thread Scheduling

1. Calling `CreateThread` does *NOT* necessarily cause the `BubbleSort` function to be immediately executed!
2. The operating system is free to assign CPU's to threads in any order, and switch CPU's between the threads.
3. There can be (and often are) more threads than there are CPUs.
   - In this case, the operating system switches the CPU between the threads, just like switching between processes.
4. The obvious question now is, "How do we delete a thread?".
   - All threads stay alive until the thread starting function (`BubbleSort` in this example) exits.
   - When the thread function exits, the operating system deletes the thread and re–assigns the CPU to another thread or process.
   - *IMPORTANT.* If a threads "parent" (the thread creating the thread) exits, then all child threads are immediately terminated.
   - This means that when `main` exits (in the previous example), all of our `BubbleSort` threads terminate immediately, regardless of whether they are completed (or even started for that matter) or not.
   - We need some way for the `main` thread to wait for the completion of the child threads.

Thread Synchronization

1. In *gthreads*, there are two co–operating functions that are used to let a parent function(the `main` function in the example) wait until all threads have finished processing.
2. All threads *MUST* call `EndThread()` immediately before exiting.
   - This does several things, but primarily decrements a counter indicating how many active threads are remaining.
   - If this counter decrements to zero, the thread also *notifies* the parent function that all threads have finished.
3. After all threads are created, the parent function *MUST* call `WaitAllThreads();`
   - This call tells the operating system that the `main` thread has no useful work to do until all of the child threads have completed and called `EndThread()`.
   - No CPU is assigned to `main` until all threads have finished.
4. When `WaitAllThreads()` returns, the parent function is free do continue processing. In our case, it should perform the merging of the four sub–arrays into the final sorted array.

CreateThread Example, Better

```
// Illustrate using CreateThread in gthreads
// George F. Riley, Georgia Tech, ECE2893, Spring 2011

#include "gthread.h"   // Must be included to use the gthreads library

void BubbleSort(int* d, int startingPoint, int length)
{ // This is the thread starting point
  // This is where, in this example, the sorting of array d will be done
  EndThread(); // Call this just before exiting
}

const int nThreads = 4;      // Number of threads desired
const int maxSize = 512000;  // Largest sort size

int main()
{
  int d[maxSize];            // Array to be sorted
  int start = 0;             // Starting point of sub-array
  int length = (omitted);    // Length of sub-array
  for (int k = 0; k < nThreads; ++k)
    { // Create each of the four sorting threads
      CreateThread(BubbleSort, d, start, length);
      // need more code here..(omitted)
    }
  // At this point all threads are created
  WaitAllThreads(); // This waits until all child threads are done
  // Perform the merge procedure to merge the separate sub-arrays
}
```