Thread Synchronization

- Mutex - A construct used to protect access to a shared bit of memory
- Think of a lock that only has one key. If you want to open the lock you must get the key. If you don’t have the key you must wait until it becomes available.
Thread Synchronization in Java

- Java threads are preemptible. Java threads may or may not be time-sliced. The programmer should not make any timing assumptions.
- Threads have priorities that can be changed (increased or decreased). An application cannot usurp resources from another application since all threads operate within one process.
- This implies that multiple threads will have race conditions (read/write conflicts based on time of access) when they run. The programmer has to resolve these conflicts.
- Example of a race condition: Account.java, TestAccount.java
- Another example of a race condition: PingPong.java
Thread Synchronization (2)

- Java has **synchronized** keyword for guaranteeing mutually exclusive access to a method or a block of code. Only one thread can be active among all synchronized methods and synchronized blocks of code in a class.

  ```java
  // Only one thread can execute the update method at a time in the class.
  synchronized void update() { //... }
  ```

  // Access to individual datum can also be synchronized.
  // The object buffer can be used in several classes, implying
  // synchronization among methods from different classes.

  ```java
  synchronized(buffer) {
      this.value = buffer.getValue();
      this.count = buffer.length();
  }
  ```

- Every Java object has an implicit monitor associated with it to implement the synchronized keyword. Inner class has a separate monitor than the containing outer class.

- Java allows **Rentrant Synchronization**, that is, a thread can reacquire a lock it already owns. For example, a synchronized method can call another synchronized method.
Synchronization Example 1

- Race conditions: `Account.java`, `TestAccount.java`
- Thread safe version using `synchronized` keyword: `RentrantAccount.java`
The `wait()` and `notify()` methods (of the `Object` class) allow a thread to give up its hold on a lock at an arbitrary point, and then wait for another thread to give it back before continuing.

Another thread must call `notify()` for the waiting thread to wakeup. If there are other threads around, then there is no guarantee that the waiting thread gets the lock next. *Starvation* is a possibility. We can use an overloaded version of `wait()` that has a timeout.

The method `notifyAll()` wakes up all waiting threads instead of just one waiting thread.
Example with `wait()`/`notify()`

class MyThing {
    synchronized void waiterMethod() {
        // do something
        // now we need to wait for the notifier to do something
        // this gives up the lock and puts the calling thread to sleep
        wait();
        // continue where we left off
    }

    synchronized void notifierMethod() {
        // do something
        // notifier the waiter that we've done it
        notify();
        // do more things
    }

    synchronized void relatedMethod() {
        // do some related stuff
    }
}
Synchronized Ping Pong using \texttt{wait()}/\texttt{notify()}

See example \texttt{threads/SynchronizedPingPong.java}
MS Windows API for Threads

In MS Windows, the system call interface is not documented. Instead the MS Windows API is documented, which helps with being able to run programs portably across multiple versions of the MS Windows operating systems.

Creating a thread gives you a *handle* that is used to refer to the actual object that represents a thread.

- **CreateThread(...).** Create a new thread and start running the start function specified in the new thread.
- **ExitThread(...), GetExitCodeThread(...), TerminateThread(...), GetCurrentThreadId(), GetCurrentThread().**
- **WaitForSingleObject(...), WaitForMultipleObjects(...).** These can be used to wait for either a process or a thread.

Get detailed information from http://msdn.microsoft.com/library/
HANDLE WINAPI CreateThread(
    LPSECURITY_ATTRIBUTES lpThreadAttributes,
    SIZE_T dwStackSize,
    LPTHREAD_START_ROUTINE lpStartAddress,
    LPVOID lpParameter,
    DWORD dwCreationFlags,
    LPDWORD lpThreadId
);

// prototype for a thread start method
DWORD WINAPI ThreadProc(
    LPVOID lpParameter
);

DWORD WINAPI GetCurrentThreadId(void);
HANDLE WINAPI GetCurrentThread(void);

VOID WINAPI ExitThread(
    DWORD dwExitCode
);

BOOL WINAPI TerminateThread(
    HANDLE hThread,
    DWORD dwExitCode
);
The `/MT` or `/MTd` flags to the compiler in Visual Studio enable multi-threaded behavior. These are turned on by default in Visual Studio 2005 onward.

Go to the project properties. In the *Property Pages* dialog box, click the *C/C++* folder. Select the *Code Generation* page. From the *Runtime Library* drop-down box, select the appropriate Multi-threaded option (it should already be the default). Click *OK*.

See the following page for details on the various multi-threading and related flags for the C/C++ compiler: http://msdn.microsoft.com/en-us/library/2kzt1wy3(v=vs.110).aspx
MS Windows API Examples

- lab/ms-windows/threads/thread-hello-world.c
- lab/ms-windows/threads/thread-scheduling.c
- lab/ms-windows/threads/thread-test.c
- and others in the ms-windows/files-processes examples folder....
Exercises

1. **Thread Dance.** Convert the program in `lab/threads/random` to use multiple threads and measure how much speedup you get relative to the number of CPUs on the system for generating the given number of random values. Also experiment with increasing the number of threads to be higher than the number of CPUs. Allow the user to specify the number of threads via a command line argument as follows:
   ```
   random <numberOfRandoms> <numThreads>
   ```

2. **The fault in our threads.** Modify the `TestList.c` driver program from your linked list project so that it creates multiple threads that run random tests on the same linked list. Also modify the driver program to accept an additional command line argument to specify the number of threads. Since all threads are sharing the same list without any protection, expect many segmentation faults due to race conditions. The purpose of this assignment is to explore race conditions in multi-threaded code.

3. **Safety in the mosh pit!** Study the `lab/threads/safe-bank-balance.c` example to see the use of Pthread mutexes. Try to implement the same idea to protect your linked list from the previous assignment. Can you get it to work safely? Can you prove that it is safe? These topics will be covered in much more depth in the Operating Systems class.

4. **Multithreaded Chat Server.** Write a multithreaded server that can chat with multiple clients simultaneously. Also write a simple client program to test the server. Use named pipes for the communication between clients and servers.