7.1 Implementation of Locks/Mutexes

How could we implement locks? No matter how we choose to implement them, we must have some hardware support. One possibility for implementing locks on a uniprocessor machine is to disable interrupts when testing/setting locks. With interrupts disabled on a single processor machine, the processor cannot switch processes, and so we can guarantee that only the active process will have access to the shared data. Another option would be to make use of atomic operations, such as test_and_set. This type of operation (which usually corresponds to a single atomic assembly instruction) behaves as if it used the following C function, atomically:

```c
int test_and_set(int x) // let x be strictly either 0 or 1.
{    
    if (x) { return 1; } else { x=1; return 0; }
}
```

All this needs to be implemented atomically, in hardware. Using this type of atomic operation, one could implement thread_lock(l) simply as

```c
while test_and_set(l) { do nothing; } // spinlock version of thread_lock()
```

and thread_unlock(l) simply as

```c
l = 0; // we need this to be an atomic clear (or assign) instruction
```

The assembly instruction test_and_set can be made to be atomic across multiple processors. An equivalent option would be an atomic compare_and_swap assembly instruction.

These low-level hardware solutions are then built up into high-level functions, either built into the languages, or in libraries. In general, do not implement your own locking functions, but rather use functions from a tested library. Getting things right can be tricky, and your own solution is also likely to be non-portable.

Summary of this section:

- Communication between threads is done implicitly, via shared variables;
- Critical sections are regions of code that access shared variables;
- Critical sections must be protected by synchronization methods;
  - We need primitives that ensure mutual exclusion;
  - Writing “personalized” solutions to concurrency is tricky and error-prone;
  - The solution is to introduce general high-level constructs into the language, such as pthread_mutex_lock() and pthread_mutex_unlock().
7.2 Advanced Synchronization, Part 1

Synchronization serves two purposes: 1) to ensure safety for updates on shared data (e.g. to avoid races conditions), and 2) to coordinate and order actions taken by threads (e.g. handling threads which communicate intermediate results amongst one another).

One of the most important aspects of parallel programs is that all their possible interleavings must be correct. One possible way to guarantee this is to simply put one lock in the beginning of each thread; however, it is also clear that we want to use as few constraints as possible, in order to effectively exploit the available concurrency. Thus, the correct placement of locks is not always trivial.

In general, locks provide safety and correctness, while condition variable provide ordering.

7.2.1 Example

Consider the following example, of a multi-threaded program that spawns \( N \) threads; each thread performs some expensive computation, and safely adds its results to a global variable.

```c
#include <pthread.h>
#include <stdio.h>

const int N = 8; // number of threads
pthread_t threads[N];
pthread_mutex_t myLock;
int total = 0;

void * expensiveComputation (void *x) {
    int v = *((int *) x);
    delete ((int *) x);
    int res = computeNthDigitOfPi (v);
    pthread_mutex_lock (&myLock);
    total += res;
    pthread_mutex_unlock (&myLock);
    return NULL;
}

int main()
{
    pthread_mutex_init (&myLock, NULL);
    for (int i=0; i<N; i++) {
        int * ptr_i = new int (i);
        pthread_create (&threads[i], NULL, expensiveComputation, (void *)ptr_i);
    }
    for (int i=0; i<N; i++) {
        pthread_join (threads[i], NULL);
    }
    printf("total: %d\n", total);
}
```