PThreads

In C & C++, one of the most common libraries for concurrency is called POSIX Threads (Or PThreads.) This provided signal and continue style monitors.

Bounded buffer using PThreads

Data Structure

A “mutex” object plays the role of the monitor. A “condition” objects provide condition queues.

```cpp
class BoundedBuffer {
private: static int const N = 10;
private: int buf[N];
private: int front, length;
private: pthread_mutex_t entryMutex;
private: pthread_cond_t bufferNotEmpty;
private: pthread_cond_t bufferNotFull;

public: BoundedBuffer() {
pthread_mutex_init( &entryMutex, NULL );
pthread_cond_init( &bufferNotEmpty, NULL );
pthread_cond_init( &bufferNotFull, NULL );
}
```

Initialization

Mutex and condition objects must be initialized. I did this in my constructor.
front = 0;
length = 0;

Destruction

public: ~BoundedBuffer() {
    pthread_mutex_destroy(&entryMutex);
    pthread_cond_destroy(&bufferNotEmpty);
    pthread_cond_destroy(&bufferNotFull);
}

Mutual exclusion

Each entry point must explicitly lock and unlock the monitor

public: void put(int value) {
    pthread_mutex_lock(&entryMutex);
    ...
    pthread_mutex_unlock(&entryMutex);
}

Waiting and signalling

public: void put(int value) {
    pthread_mutex_lock(&entryMutex);
    while(length == N)
        pthread_cond_wait(&bufferNotFull,
                           &entryMutex);
    assert(length < N);
}
buf[ (front + length) % N ] = value ;
++length ;
pthread_cond_broadcast( &bufferNotEmpty ) ;

pthread_mutex_unlock( &entryMutex ) ; }

• Because of signal and continue semantics the wait must happen in a loop.
• Note that the “wait” routine mentions the mutex, this is because conditions objects are not explicitly associated with mutex objects.
• The “broadcast” subroutine is a “signalAll”, all waiting threads are awakened.

Threads in PThreads

PThreads also provides facilities for creating threads that share shared memory.
Depending on the implementation, threads may be
• Native threads — known to and scheduled by the OS
• User level threads — not known to the OS. The user process must arrange for switching the CPU between threads.
• Nonpreemptive threads — GNU Portable Threads supports cooperative multitasking. There is no actual concurrency.
Only native threads allow you to take advantage of multiple cores.