POSIX Threads, Pthreads, and Linux Thread

LinuxThreads is a Posix compliant implementation of PThreads. It terms of programming with PThreads, you'll need to include the header file *pthread.h* in your program and you'll need to link in the thread library (using the *–lpthread* option on the compiler).

There are some basic thread routines shown in the table below

Functionality	Library call, type name	Description
Thread handle	`pthread_t`	A type used to declare thread "handles." A handle is just a variable you use to refer to a thread.
Thread creation	`pthread_create(pthread_t *p, NULL, (void *) (*func)(void *), void *args)`	This library call creates a thread. You pass the address of the thread handle variable, a function the new thread will execute, and arguments for the function. Note the strange type for the third parameter: this is a pointer to a function that returns a void pointer and takes a void pointer as an argument.
Thread completion	`pthread_exit(void *r)`	Terminates the thread and returns the value pointed to by *r* to a thread that has done a pthread_join on the thread. Note a thread can also complete by simply returning from the function.
Thread joining	`pthread_join(pthread_t p, void **args)`	Allows a thread to wait for a thread to complete and receive its results.
Mutex variables	`pthread_mutex_t`	A type used to declare a mutex variable. A mutex variable can be use to provide mutual exclusion for a thread.
Mutex variable initialization	`pthread_mutex_init(pthread_mutex_t *m, const pthread_mutexattr_t *a);`	This routine makes a mutex variable *m* ready to use.
Mutex variable cleanup	`pthread_mutex_destroy(pthread_mutex_t &m)`	Cleans up a mutex variable after it is no longer needed.
Mutex locking	`pthread_mutex_lock(prthread_mutex_t &m)`	Sets the lock on a mutex variable. Any other thread trying to take this mutex lock will be blocked until the lock is released.
Mutex unlocking	`pthread_mutex_unlock(prthread_mutex_t &m)`	Releases a mutex lock, allowing a blocked

		thread to proceed.
Conditional variables	`pthread_cond_t`	A type used to declare condition variable. Condition variables allow a thread to wait for an event to occur.
Condition variable initialization	`pthread_cond_init(pthread_cond_t &c, [thread_condattr_t *a);`	Makes the condition variable ready to use.
Condition variable initialization	`pthread_cond_destroy(pthread_cond_t &c)`	Cleans up a condition variable after it is no longer needed.
Condition variable wait	`pthread_cond_wait(pthread_cond_t &c, pthread_mutex_t &m)`	This routine allows a thread to wait for an event (represented by the condition variable). The thread must previously have taken the mutex lock on m. The thread blocks on the mutex variable queue and releases the mutex lock. The thread will retake the mutex lock when it is signaled that the event has occurred.
Condition variable signaling	`pthread_cond_signal(pthread_cond_t &c)`	Signals on thread that is waiting on the condition variable *c* to unblock.
Semaphore variable	`sem_t`	type used to declare a semaphore variable
Semaphore variable initialization	`sem_init(sem_t *s, int pshared, usigned int v);`	Initiallizes the semaphore variable s to have the value v. On Linux, pshared is always 0. Indicates whether the semaphore is shared between processes.
Semaphore Wait	`sem_wait(sem_t *s)`	The P operation. Blocks if the semaphore value is zero and decrements value.
Semaphore Non-Blocking Wait	`sem_trywait(sem_t *s)`	Returns a zero is the semaphore value is not zero and error value EAGAIN if it is. Otherwise, works the same as sem_wait.
Semaphore Sginal	`sem_post(sem_t *s)`	The V operation. Increments the semaphore value and cause block thread to awaken.
Semaphore Get Value	`sem_getvalue(sem_t, *s, int *v)`	Returns the semaphore value in the variable v.
Semaphore Destroy	`sem_destroy(sem_t *s)`	Gets rid of the semaphore resources built by sem_init.