POSIX Threads (Pthreads)

The POSIX thread libraries are a standards based thread API for C/C++. It allows one to spawn a new concurrent process flow. It is most effective on multi-processor or multi-core systems where the process flow can be scheduled to run on another processor thus gaining speed through parallel or distributed processing. Threads require less overhead than "forking" or spawning a new process because the system does not initialize a new system virtual memory space and environment for the process. While most effective on a multiprocessor system, gains are also found on uniprocessor systems which exploit latency in I/O and other system functions which may halt process execution. (One thread may execute while another is waiting for I/O or some other system latency.) Parallel programming technologies such as MPI and PVM are used in a distributed computing environment while threads are limited to a single computer system. Context switching between threads is much faster than context switching between processes.

All threads within a process share the same address space. They share their code, data and heap segments, open file descriptors, signal and signal handlers, current working directory, and user and group ids. Each thread has its unique thread id, stack, set of registers, signal mask and priority. The purpose of using the POSIX thread library in your software is to execute software faster. This handout is designed to get you started with using the pthreads package.

Thread operations include thread creation, termination, synchronization (joins,blocking), scheduling, data management and process interaction.
Read /usr/include/pthread.h and appropriate man pages for a complete set of pthread commands available.

#include <pthread.h>

1. Thread Creation

A thread is spawned by defining a function and its arguments which will be processed in the thread. pthread_create( ) creates a thread with attributes specified by attr, starting execution at the function named start with the argument value arg. A null value of attr creates a thread with default attributes.

```
int pthread_create(pthread_t *th, pthread_attr_t *attr, void *(*start)(void *), void *arg);
```

- ## Thread Creation Attributes

A thread attributes object is a collection of values that specify how a thread is created. Changes to the attribute values of the object do not affect threads already created using the object. Attributes include size and location of the stack, detached state, and scheduling attributes.

```
int pthread_attr_init(pthread_attr_t *attr);
int pthread_attr_destroy(pthread_attr_t *attr);
int pthread_attr_setstacksize(pthread_attr_t *attr, size_t sz);
int pthread_attr_getstacksize(const pthread_attr_t *attr, size_t *sz);
int pthread_attr_setstack(pthread_attr_t *attr, void *addr, size_t stacksize);
int pthread_attr_getstack(const pthread_attr_t *attr, void **addr, size_t
                                                              *stacksize);
```

- **Thread Scheduling Attributes**

int pthread_attr_setscope(pthread_attr_t *attr, int scope);
int pthread_attr_getscope(const pthread_attr_t *attr, int *scope);

Possible values for scope are PTHREAD_SCOPE_PROCESS and PTHREAD_SCOPE_SYSTEM. POSIX.1-2001 only requires that an implementation support one of these contention scopes, but permits both to be supported. Linux supports PTHREAD_SCOPE_SYSTEM, but not PTHREAD_SCOPE_PROCESS

Attributes of a newly created thread may be inherited from the parent thread or explicitly stated. Value for inherit must be PTHREAD_INHERIT_SCHED or PTHREAD_EXPLICIT_SCHED (default).

int pthread_attr_setinheritsched(pthread_attr_t *attr, int inherit);
int pthread_attr_getinheritsched(const pthread_attr_t *attr, int *inherit);

policy can be SCHED_FIFO, SCHED_RR, or SCHED_OTHER(default, same as SCHED_RR).

int pthread_attr_setschedpolicy(pthread_attr_t *attr, int policy);
int pthread_attr_getschedpolicy(const pthread_attr_t *attr, int *policy);

2. Thread Synchronization

Pthreads provides three synchronization mechanisms: Join, Mutex, Condition variables.

- **Join**

A join is performed when one wants to wait for a thread to finish. The calling thread waits for the thread identified by th to terminate.

int pthread_join(pthread_t th, void **ret)

- ## Mutex

Mutual exclusion locks (mutexes) are used to serialize the execution of threads through critical sections of code which access shared data. A successful call for a mutex lock via pthread_mutex_lock() or pthread_mutex_trylock() will cause another thread that tries to acquire the same mutex via pthread_mutex_lock() to block until the owning thread calls pthread_mutex_unlock().

Initializes a mutex structure. A null value of attr initializes mutex with default attributes.

```
int pthread_mutex_init(pthread_mutex_t *mutex, const pthread_mutexattr_t
                                                                    *attr);
int pthread_mutex_destroy(pthread_mutex_t *mutex);
int pthread_mutex_lock(pthread_mutex_t *mutex);
int pthread_mutex_trylock(pthread_mutex_t *mutex);
int pthread_mutex_unlock(pthread_mutex_t *mutex);
```

- ## Condition Variables

Condition variables provide high performance synchronization primitives to wait for or wake up threads waiting for certain conditions to be satisfied. Functions are provided to wait on a condition variable and to wake up (signal) threads that are waiting on the condition variable. A condition variable is a variable of type pthread_cond_t and is used with the appropriate functions for waiting and later, process continuation. The condition variable mechanism allows threads to suspend execution and relinquish the processor until some condition is true. A condition variable must always be associated with a mutex to avoid a race condition created by one thread preparing to wait and another thread which may signal the condition before the first thread actually waits on it resulting in a deadlock. Any mutex can be used, there is no explicit link between the mutex and the condition variable.

```
int pthread_cond_init(pthread_cond_t *cond, pthread_condattr_t *attr);
int pthread_cond_destroy(pthread_cond_t *cond);
int pthread_cond_signal(pthread_cond_t *cond);
int pthread_cond_broadcast(pthread_cond_t *cond);
int pthread_cond_wait(pthread_cond_t *cond, pthread_mutex_t *mutex);
int pthread_cond_timedwait(pthread_cond_t *cond, pthread_mutex_t *mutex,
const struct timespec *abstime);
```

Functions pthread_cond_wait() and pthread_cond_timedwait() are used to block on a condition variable. They must be called with mutex locked by the calling thread. The functions atomically release mutex and block the calling thread on the condition variable cond. Before return, the mutex is reacquired for the calling thread. The function pthread_cond_timedwait() is the same as pthread_cond_wait() except that an error is returned if the absolute time specified by abstime passes before the waiting thread is

signalled. If a time-out occurs, pthread_cond_timedwait() still reacquires mutex before returning to the caller.

3. Thread Pitfalls

- **Race conditions:** While the code may appear on the screen in the order you wish the code to execute, threads are scheduled by the operating system and are executed at random. It cannot be assumed that threads are executed in the order they are created. They may also execute at different speeds. When threads are executing (racing to complete) they may give unexpected results (race condition). Mutexes and joins must be utilized to achieve a predictable execution order and outcome.

- **Thread safe code:** The threaded routines must call functions which are "thread safe". This means that there are no static or global variables which other threads may clobber or read assuming single threaded operation. If static or global variables are used, then mutexes must be applied or the functions must be re-written to avoid the use of these variables. In C, local variables are dynamically allocated on the stack. Therefore, any function that does not use static data or other shared resources is thread-safe. Thread-unsafe functions may be used by only one thread at a time in a program and the uniqueness of the thread must be ensured. Many non-reentrant functions return a pointer to static data. This can be avoided by returning dynamically allocated data or using caller-provided storage. An example of a non-thread safe function is strtok which is also not re-entrant. The "thread safe" version is the re-entrant version strtok_r.

- **Mutex Deadlock:** This condition occurs when a mutex is applied but then not "unlocked". This causes program execution to halt indefinitely. It can also be caused by poor application of mutexes or joins. Be careful when applying two or more mutexes to a section of code. If the first pthread_mutex_lock is applied and the second pthread_mutex_lock fails due to another thread applying a mutex, the first mutex may eventually lock all other threads from accessing data including the thread which holds the second mutex. The threads may wait indefinitely for the resource to become free causing a deadlock. It is best to test and if failure occurs, free the resources and stall before retrying.