**AAIT – ITSC** **4931**- **Operating Systems**

**Lab 3 – Threads Synchronization (Condition Variables)**

# Objective:

When multiple threads are running they will invariably need to communicate with each other in order synchronize their execution. One main benefit of using threads is the ease of using synchronization facilities.

Threads need to synchronize their activities to effectively interact. This includes:

* Implicit communication through the modification of shared data
* Explicit communication by informing each other of events that have occurred.

**This lab describes** the synchronization types available with threads and discusses when and how to use synchronization. There are a few possible methods of synchronizing threads:

* Mutex (Previous Lab Session)
* **Condition Variables (Todays Lab Session)**
* Semaphores (Next Lab Session)

# Condition Variables:

* Condition variables provide yet another way for threads to synchronize. While *mutexes* implement synchronization by controlling thread access to data, *condition variables* allow threads to synchronize based upon the actual value of data.
* Without condition variables, the programmer would need to have threads continually polling (possibly in a critical section), to check if the condition is met. This can be very resource consuming since the thread would be continuously busy in this activity. A condition variable is a way to achieve the same goal without polling.
* *A condition variable is always used in conjunction with a mutex lock because the shared information that they depend on must be synchronized across threads.*
* The typical sequence for using condition variables:
  + Create and initialize a condition variable
  + Create and initialize an associated mutex
* **Define a predicate variable (variable whose condition must be checked)**
* A thread does work up to the point where it needs a certain condition to occur (such as the predicate must reach a specified value). It then "waits" on a condition variable by:
  + Locking the mutex
  + While predicate is unchanged wait on condition variable
  + Unlocking the mutex
* Another thread does work which results in the waited for condition to occur (such as changing the value of the predicate). Other waiting threads are "signalled" when this occurs by:
  + Locking the mutex
  + Changing the predicate
  + Signalling on the condition variable Unlocking the mutex

# Creating / Destroying Condition Variables:

* pthread\_cond\_init ( pthread\_cond\_t condition, pthread\_condattr\_t attr)
* pthread\_cond\_destroy ( pthread\_cond\_t condition)
* pthread\_condattr\_init ( pthread\_condattr\_t attr )
* pthread\_condattr\_destroy ( pthread\_condattr\_t attr )

pthread\_cond\_init( ) creates and initializes a new condition variable object. The ID of the created condition variable is returned to the calling thread through the condition parameter.

Condition variables must be of type pthread\_cond\_t .

The optional attr object is used to set condition variable attributes. There is only one attribute defined for condition variables: process-shared, which allows the condition variable to be seen by threads in other processes. The attribute object, if used, must be of type pthread\_condattr\_t (may be specified as NULL to accept defaults). Refer to the internet for more details of pthread\_condattr\_t.

Currently, the attributes type attr is ignored in some implementation of pthreads; use NULL.

If implemented, the pthread\_condattr\_init( ) and pthread\_condattr\_destroy( ) routines are used to create and destroy condition variable attribute objects.

pthread\_cond\_destroy( ) should be used to free a condition variable that is no longer needed.

##### Waiting / Destroying Condition Variables :

* pthread\_cond\_wait ( pthread\_cond\_t condition, pthread\_mutex\_t mutex )
* pthread\_cond\_signal ( pthread\_cond\_t condition )
* pthread\_cond\_broadcast ( pthread\_cond\_t condition )

pthread\_cond\_wait( ) blocks the calling thread until the specified condition is signalled. This routine should be called while mutex is locked, and it will automatically release the mutex while it waits.

The pthread\_cond\_signal( ) routine is used to signal (or wake up) another thread which is waiting on the condition variable. It should be called after mutex is locked.

The pthread\_cond\_broadcast( ) routine should be used instead of pthread\_cond\_signal( ) if more than one thread is in a blocking wait state.

It is a logical error to call pthread\_cond\_signal( ) before calling pthread\_cond\_wait( ).

##### Example: Using Condition Variables:

#### Lab4.c:

###### /\* Code for cond-hellothread.c \*/

#include <pthread.h>

#include <stdio.h>

*/\* This is the initial thread routine \*/*

void\* compute\_thread (void\*);

*/\* This is the lock for thread synchronization \*/*

pthread\_mutex\_t my\_sync;

*/\* This is the condition variable \*/*

pthread\_cond\_t rx;

#define TRUE 1

#define FALSE 0

*/\* this is the Boolean predicate \*/*

int thread\_done = FALSE;

main(){

*/\* This is data describing the thread created \*/*

pthread\_t tid;

pthread\_attr\_t attr;

char hello[ ] = {"Hello, "};

char thread[ ] = {"thread"};

*/\* Initialize the thread attributes \*/*

pthread\_attr\_init (&attr);

*/\* Initialize the mutex (default attributes) \*/*

pthread\_mutex\_init (&my\_sync, NULL);

*/\* Initialize the condition variable (default attr) \*/*

pthread\_cond\_init (&rx, NULL);

*/\* Create another thread. ID is returned in &tid \*/*

*/\* The last parameter is passed to the thread function \*/*

pthread\_create(&tid, &attr, compute\_thread, hello);

*/\* wait until the thread does its work \*/*

pthread\_mutex\_lock(&my\_sync);

while (!thread\_done)

pthread\_cond\_wait(&rx, &my\_sync);

*/\* When we get here, the thread has been executed \*/*

printf(thread);

printf("\n");

pthread\_mutex\_unlock(&my\_sync);

exit(0);

}

*/\* The thread to be run by create\_thread \*/*

void\* compute\_thread(void\* dummy){

*/\* Lock the mutex - the cond\_wait has unlocked it \*/*

pthread\_mutex\_lock (&my\_sync);

printf(dummy);

*/\* set the predicate and signal the other thread \*/*

thread\_done = TRUE;

pthread\_cond\_signal (&rx);

pthread\_mutex\_unlock (&my\_sync);

return;

}

# Lab Exercise 1: Hello World with a Controlled Order

Now that we know how to wait for a condition to occur, you need to implement a program that creates 10 threads which say hello in a specific order: we want those with an even id to print first, and those with an odd id to print after all the evens have printed. Since we are not guaranteed that the threads will start in any given order, we must have the odd threads wait until all the even threads have printed.

Hint: pthread\_self() returns the id of the thread.

# Deliverable:

Send the file containing your answer to Lab exercise 1 to lisanu@gmail.com.