## Pthreads

* **POSIX Threads**, or **Pthreads**, is a POSIX standard for threads. The standard, POSIX.1c, Threads extensions (IEEE Std 1003.1c-1995), defines an API for creating and manipulating threads.
* Implementations of the API are available on many Unix-like POSIX systems such as **FreeBSD**, **NetBSD**, **GNU/Linux**,**Mac OS X** and **Solaris**, but **Microsoft Windows** implementations also exist. For example, the pthreads-w32 is available and supports a subset of the Pthread API for the Windows 32-bit platform.
* The POSIX standard has continued to evolve and undergo revisions, including the Pthreads specification. The latest version is known as IEEE Std 1003.1, 2004 Edition.
* Pthreads are defined as a set of C language programming types and procedure calls, implemented with a **pthread.h**header file. In GNU/Linux, the pthread functions are not included in the standard C library. They are in **libpthrea**, therefore, we should add **-lpthread** to link our program.

THE PTHREAD API

Pthreads API can be grouped into four:

* **Thread management:**  
  Routines that work directly on threads - creating, detaching, joining, etc. They also include functions to set/query thread attributes such as joinable, scheduling etc.
* **Mutexes:**  
  Routines that deal with synchronization, called a "mutex", which is an abbreviation for "mutual exclusion". Mutex functions provide for creating, destroying, locking and unlocking mutexes. These are supplemented by mutex attribute functions that set or modify attributes associated with mutexes.
* **Condition variables:**  
  Routines that address communications between threads that share a mutex. Based upon programmer specified conditions. This group includes functions to create, destroy, wait and signal based upon specified variable values. Functions to set/query condition variable attributes are also included.
* **Synchronization:**  
  Routines that manage read/write locks and barriers.

CREATING THREADS

* Our **main()** program is a single, default thread. All other threads must be explicitly created by the programmer.
* **pthread\_create** creates a new thread and makes it executable. This routine can be called any number of times from anywhere within our code.
* **pthread\_create (pthread\_t \*thread, pthread\_attr\_t \*attr, void \*(\*start\_routine)(void \*), void \*arg)**arguments:
  + **thread:**   
    An identifier for the new thread returned by the subroutine. This is a pointer to **pthread\_t** structure. When a thread is created, an identifier is written to the memory location to which this variable points. This identifier enables us to refer to the thread.
  + **attr:**   
    An attribute object that may be used to set thread attributes. We can specify a thread attributes object, or NULL for the default values.
  + **start\_routine:**   
    The routine that the thread will execute once it is created.

**void \*(\*start\_routine)(void \*)**

We should pass the address of a function taking a pointer to void as a parameter and the function will return a pointer to void. So, we can pass any type of single argument and return a pointer to any type.   
While using **fork()** causes execution to continue in the same location with a different return code, using a new thread explicitly provides a pointer to a function where the new thread should start executing.

**arg:**   
A single argument that may be passed to **start\_routine**. It must be passed as a **void pointer**. NULL may be used if no argument is to be passed.

* The maximum number of threads that may be created by a process is implementation dependent.
* Once created, threads are peers, and may create other threads. There is no implied hierarchy or dependency between threads.

JOIN

* **int pthread\_join (pthread\_t th, void \*\*thread\_return)**  
  The first parameter is the thread for which to wait, the identified that **pthread\_create** filled in for us. The second argument is a pointer to a pointer that itself points to the return value from the thread. This function returns zero for success and an error code on failure.
* When a thread is created, one of its attributes defines whether the thread is joinable or detached. Only threads that are created as joinable can be joined. If a thread is created as detached, it can never be joined.

A thread can execute a thread join to wait until the other thread terminates. In our case, you - the main thread - should execute a thread join waiting for your colleague - a child thread - to terminate. In general, thread join is for a parent (**P**) to join with one of its child threads (**C**). Thread join has the following activities, assuming that a parent thread **P** wants to join with one of its child threads **C**:

* When **P** executes a thread join in order to join with **C**, which is still running, **P** is suspended until **C** terminates. Once **C**terminates, **P** resumes.
* When **P** executes a thread join and **C** has already terminated, **P** continues as if no such thread join has ever executed (i.e., join has no effect).

A parent thread may join with many child threads created by the parent. Or, a parent only join with some of its child threads, and ignore other child threads. In this case, those child threads that are ignored by the parent will be terminated when the parent terminates.

* The **pthread\_join()** subroutine **blocks the calling thread** until the specified thread terminates..

## Semaphores :

A **semaphore** is a **counting** and **signaling** mechanism. We use it to allow threads access to a specified number of items. If there is a single item, then a semaphore is virtually the same as a mutex.

However, it is more commonly used in a situation where there are multiple items to be managed. Semaphores can also be used to signal between threads or processes. For example, to tell another thread that there is data present in a queue. There are two types of semaphores: named and unnamed semaphores.

The semaphores form a classic system for confining access to shared assets (e.g. capacity) in a multi-processing environment. They were formulated by Dijkstra .

A semaphore is a protected variable (or abstract data type ) which can only be accessed using the following operations:

P(s) Semaphore s; while (s == 0) ; /\* wait until s>0 \*/ s = s-1;

V(s) Semaphore s; s = s+1;

Init(s, v) Semaphore s; Int v; s = v;

P and V stand for Dutch "Proberen", to test, and "Verhogen", to increment. The value of a semaphore is the number of units of the resource which are free (if there is only one resource a "binary semaphore" with values 0 or 1 is used). The P operation busy-waits (or maybe sleeps ) until a resource is available whereupon it immediately claims one. V is the inverse, it simply makes a resource available again after the process has finished using it. Init is only used to initialise the semaphore before any requests are made. The P and V operations must be indivisible, i.e. no other process can access the semaphore during the their execution.

To avoid busy-wait ing, a semaphore may have an associated queue of processes (usually a FIFO ). If a process does a P on a semaphore which is zero the process is added to the semaphore's queue. When another process increments the semaphore by doing a V and there are tasks on the queue, one is taken off and resumed.

A semaphore is as an object with an integer value that we can manipulate with two routines sem\_wait() sem\_post() to follow the POSIX standard). Because the initial value of the semaphore determines its behavior, before calling any other routine to interact with the semaphore, we must first initialize it to some value, as this code below does:

int sem\_init(sem\_t \**sem*, int *pshared*, unsigned *value*);

If the *pshared* argument is zero, then the semaphore is shared between threads of the process; any thread in this process can use *sem* for performing [*sem\_wait*()](http://pubs.opengroup.org/onlinepubs/009695399/functions/sem_wait.html), [[TMO](javascript:open_code('TMO'))] Option Start][*sem\_timedwait*()](http://pubs.opengroup.org/onlinepubs/009695399/functions/sem_timedwait.html), Option End] [*sem\_trywait*()](http://pubs.opengroup.org/onlinepubs/009695399/functions/sem_trywait.html), [*sem\_post*()](http://pubs.opengroup.org/onlinepubs/009695399/functions/sem_post.html), and [*sem\_destroy*()](http://pubs.opengroup.org/onlinepubs/009695399/functions/sem_destroy.html) operations.

After a semaphore is initialized, we can call one of two functions to interact with it, sem\_wait() or sem\_post() [4]. The behavior of these two functions is described here: -------------------------------------------------------------------------------- int sem\_wait(sem\_t \*s) { wait until value of semaphore s is greater than 0 decrement the value of semaphore s by 1 } int sem\_post(sem\_t \*s) { increment the value of semaphore s by 1 if there are 1 or more threads waiting, wake 1 }

### The Student Routine :

**sem\_wait**(&waitingRoom) :

Wait until the value of semaphore waitingRoom is > 0

Then decrement the value of semaphore by 1

**sem\_post**(&waitingRoom);

Increment the value of semaphore by 1

if there are 1 or more threads waiting, wake any one

**sem\_post**(&studentAvailable)

Increment the value of semaphore by 1

if there are 1 or more threads waiting, wake any one

**sem\_wait**(&taReady)

wait for the value of semaphore to be >0

and then decrement the value by 1

### The TA routine :

**sem\_wait**(&studentAvailable)

Wait until the value of semaphore studentAvailable is > 0

Then decrement the value of semaphore by 1

**sem\_wait**(&waitingRoom)

Wait until the value of semaphore waitingRoom is > 0

Then decrement the value of semaphore by 1

**sem\_post**(&waitingRoom);/\*Release waiting room access\*/

Increment the value of semaphore by 1

if there are 1 or more threads waiting, wake any one

**sem\_post**(&taReady)

Increment the value of semaphore by 1

if there are 1 or more threads waiting, wake any one