<u>POSIX Threads (pthreads)</u>

- **Pthreads** is a new standard **POSIX API** that provides library routines for multithreading on **UNIX**.

- The **POSIX API** and the **Solaris API** share identical library calls, the only difference is the name of the function call.

- The **POSIX API** uses the prefix **pthread_** before each call, **Solaris API** uses the prefix **thr_** or **lwp_** before each call.

- The functionality of the two APIs however is different in several ways. The table below ilustrates the similarities and differences between the two APIs.

POSIX API	***Similar Functionality***	***Solaris API***
Thread cancellation	Join	Continue
Scheduling policies	Key creation	Suspend
Synchronization attributes	Create, Exit, Kill	Semaphore variables
Thread attributes	Priorities, sigmask,	Concurrency setting
	Mutex variables	Reader/Writer variables
	Condition variables	Daemon threads
	Thread specific data	

- **Solaris** threads provide the following features not found in **pthreads**:

  - Reader/Writer locks (classic one server many client problem).
  - Ability to suspend and continue a given thread.
  - Ability to create daemon threads (**THR_DAEMON**).
  - Ability to set and get levels of concurrency.

- **pthreads** provide the following features not found in **Solaris**:

  - Ability to cancel a given thread.
  - Ability to set thread and synchronization attributes.
  - Scheduling policies.

<u>Creating a Thread</u>

- A new thread is created using the **pthread_create()** function. The syntax is as follows:

  #include <pthread.h>

  int pthread_create (pthread_t *thread, const pthread_attr_t *attr, void * (*start_routine) (void *), void *arg);

  - The function creates a new thread with attributes specified by **attr**, within a process. If **attr** is **NULL**, default thread attributes are used.

  - If the function is successful, the new **thread ID** is stored in the location referenced by **thread.**

  - The third argument, **start_routine,** specifies the function which the thread is to execute. This is always a function that takes **void*** as its argument and returns void*.

  - The last parameter specifies the argument to the executing function which is void.

  - If the function is successful, the function returns zero. Otherwise an error number is returned to identify the reason for not creating a new thread:

    EAGAIN: The system lacked the resources to create another thread, or creation of a new thread will result it exceeding the thread limit set by **PTHREAD_THREADS_MAX.**

    EINVAL: The value specified by **attr** is invalid.

- Another function that is often used as part of thread creation is **pthread_join().**

  #include <pthread.h>

  int pthread_join (pthread_t thread, void **value_ptr);

- This function suspends execution of the calling thread until the target function specified by **thread** terminates.

- The argument ****value_ptr** will contain the value passed to **pthread_exit**() by the terminating thread. If **pthread_exit()** is not being used than this argument is set to **NULL.**

- If the function is successful, the function returns zero. Otherwise an error number is returned to identify the reason for the failure:

  ESRCH: No thread could be found corresponding to that specified by the given **thread** ID.

  EDEADLK: A deadlock was detected or the value of thread specifies the calling thread.

<u>Attribute Objects</u>

- The pthreads API allows a programmer to create a thread in one of many **states**.
- Examples of this would be a thread that is created bound or unbound, a mutex variable that can be interprocess or intraprocess.
- Attribute objects are created and then used as arguments to creation or initialization functions to define the thread state or synchronization variables.
- The attribute objects contain all the **state information** we require to define the state of a new thread.
- The attribute object is specified when the thread is created. This is done by setting up a threads attributes structure of type **pthread_attr_t**.
- A threads attributes structure is allocated by the program and then intitialized by calling:

  int pthread_attr_init (pthread_attr_t *attr);

- The above call initializes the attributes structure's thrad attributes to the systems default values.
- Individual attributes can then be set up by calling functions that get and set up specific attributes.
- Once an object has been set up, it can be used by any **pthread_create()** call to define the state of a new thread.
- At the end of the program the attributes structure is deinitialized by calling:

  int pthread_attr_destroy (pthread_attr_t *attr);

- The state information is as follows:

Detached State:

- This state determines whether the thread will be joinable from another thread.
- When an application does not require the thread's completion status and doesn't need to know when the thread has exited, the application creates a **detached** thread.
- Detached threads cannot be the target of **pthread_join().**
- The **detachstate** attribute specifies whether the thread is to be detached and it has one of two values:

  PTHREAD_CREATE_DETACHED
  PTHREAD_CREATE_JOINABLE (default value)

- This attribute is manipulated by the following two function calls:

```
// set the value of detachstate as specified in attr
int pthread_attr_setdetachstate (pthread_attr_t *attr, int detachstate);

// get the value of detachstate
int pthread_attr_getdetachstate (pthread_attr_t *attr, int detachstatep);
```

- The following code fragment illustrates how to create a detached thread:

```
int main (void)
{
    pthread_attr_t detached_attr;
    pthread_t t;
    void *detached_func (void *);

    pthread_attr_init (&detached_attr);
    pthread_attr_setdetachstate (&detached_attr,
                                PTHREAD_CREATE_DETACHED);
    pthread_create (&t, &detached_attr, detached_func, NULL);
    ……….
    ……….
    return(0);
}
```

Scope

- This attribute determines whether the thread has a **process-wide** or **system-wide** scope.

- The attribute can take one of the two values:

  PTHREAD_SCOPE_SYSTEM **//system-wide, kernel sees the thread**

  PTHREAD_SCOPE_PROCESS **//process-wide only, kernel does not see the // thread - default value.**

- This attribute is manipulated by the following two function calls:

  // set the value of scopestate as specified in attr
  int pthread_attr_setscope (pthread_attr_t *attr, int scopestate);

  // get the value of scopestate
  int pthread_attr_getscope (pthread_attr_t *attr, int scopestatep);

Stack Address

- Specifies the base (starting) address of the stack for the thread.

- Some care must be used when using this attribute to set specific addresses simply because this attribute cannot be used later to create new threads. It requires a change of the stack base.

- The base stack can be set and retrieved using the following functions:

  pthread_attr_setstackaddr()

  pthread_attr_getstackaddr()

- The default value for a stack address is **NULL**, which means that the system will assign the stack base address.

Stack Size

- This attribute specifies the size of the stack in bytes, for a thread.
- Usually a NULL size value will allow the system to set the default stack size.
- The stack size can be set and retrieved using the following functions:

  pthread_attr_setstacksize()

  pthread_attr_getstacksize()

Scheduling Policy

- The scheduling parameters in each thread **attribute object** define how the thread is **scheduled** to run, and set the **priority** for the thread.
- The parameters used by the scheduling policy are set and retrieved via the **schedparam** attribute in the threads attribute structure.
- This attributes object consists of a **sched_param** structure with at least one defined member:

```
struct sched_param
{
        …….
        int sched_priority;
        …….
}
```

- The priority of the thread can be set and retrieved from the **schedparam** attribute by calling:

```
// set the thread priority value to sched_priority as specified in sched_param
int pthread_attr_setschedparam (pthread_attr_t *attr, const struct
sched_param *paramp);

// get the thread priority value from sched_param
int pthread_attr_getschedparam (pthread_attr_t *attr, const struct
sched_param *paramp);
```

- In the implementation model, each thread has an integer priority (**sched_priority** in the **sched_param**) that rabges between a defined **minimum** and a defined **maximum**.
- The minimum and maximum values are defined by the kernel implementation of threads.

- The implementation's minimum and maximum priority for each **scheduling policy** can be retrieved by calling:

  #include <sched.h>

  // get minimum priority for policy
  int sched_get_priority_min (int policy);

  // get maximum priority for policy
  int sched_get_priority_max (int policy);

- Where **policy** is one of the three supported policies (**SCHED_FIFO**, **SCHED_RR**, or **SCHED_OTHER**), to be discussed next.

- The threads scheduling policy is set when it is created via the **schedpolicy** attribute in the thread creation attributes (as shown earlier).

- The **schedpolicy** is set and retrieved by calling:

  // set the thread scheduling policy to the defined integer **policy**
  int pthread_attr_setschedpolicy (pthread_attr_t *attr, int policy);

  // get the thread scheduling policy
  int pthread_attr_getschedpolicy (pthread_attr_t *attr, int policyp);

- As mentioned earlier, **POSIX** defines three thread scheduling policies:

- **SCHED_FIFO**

  - Defines a simple **first in, first out** scheduler. Threads are placed on the run queue as described below.

  - When an **active** thread is **preempted**, it is placed at the head of the queue associated with its priority.

  - When a **blocked** thread becomes **runnable**, it is placed at the tail of the queue associated with its priority.

  - When an **active** thread is the target of **pthread_setschedparam()** (set parameters after the thread is running), it is placed at the tail of the queue associated with its **new priority**.

  - When an **active** thread is the target of **sched_yield()** (force running process to relinquish CPU until process gets to the head of its run queue), it is placed at the tail of the queue associated with its priority.

- **SCHED_RR**
  - Defines a **round-robin** scheduling algorithm.
  - It is similar to the **SCHED_FIFO** except that an active **SCHED_RR** thread will be automatically preempted after it has been running for a **time quantum** associated with the entire process.
  - Following the expiration of the time quantum value, the thread is put back to the tail of its run queue.
  - The value of the time quantum can be retrieved by calling:

    #include <sched.h>

    int sched_rr_get_interval (pid_t pid, struct timespec *quantump);

- **SCHED_OTHER**
  - This is the same **default** policy that is also supported by Solaris. The kernel simply assigns CPU time slices based on the thread's priority.
  - In general, **SCHED_OTHER** should be used unless there are compelling reasons for strict priority scheduling, such as in real-time applications.

Synchronization Variables

- Both POSIX and Solaris threads provide several thread **synchronizing** functions, including **Semaphores** and **Mutexes.** We will discuss the simplest of these, **Mutexes.**
- **Mutexes** provide a "**locking**" mechanism that will allow only one thread at a time to execute a section of code.
- Mutexes are allocated and defined as a static or automatic data structure of type **pthread_mutex_t** as in:

  pthread_mutex_t lock;

- They can also be dynamically allocated as in:

  pthread_mutex_t *lock;

  lock = (pthread_mutex_t *) malloc(sizeof (pthread_mutex_t));

- Once created, a mutex must be initialized as follows:

```
int pthread_mutex_init (pthread_mutex_t *lock,
        const pthread_mutexattr_t *attr);
```

- The **attr** argument points to a data structure containing attributes to be used for the mutex.

- If **attr** is **NULL**, default attributes are used by the kernel.

- Mutexes can also be statically initialized by setting them to a special value, as follows:

    pthread_mutex_t lock = PTHREAD_MUTEX_INITIALIZER;

- This allows the mutex to be initialized without additional programming overhead.

- Mutexes are destroyed by calling:

  pthread_mutex_destroy (pthread_mutex_t *lock);

- Locking and unlocking are the primary mutex operations. A mutex is locked using:

  int pthread_mutex_lock (pthread_mutex_t *lock);

- The thread that **locks** the **mutex** is considered its **owner**. Only the mutex owner can unlock the mutex.

- A mutex is unlocked using:

  int pthread_mutex_unlock (pthread_mutex_t *lock);

- The example program given is a simple but complete example of a multi-threaded program.

- You will need to compile it with the thread library as follows:

  gcc multi.c -o multi -lpthread