opiframe
Knowledge equals power

Opiframe Oy

Posix Threads

Posix Threads - Overview

opiframe
Knowledge equals power

- A thread of execution results from a fork of a computer program into two or more concurrently running tasks.

- The implementation of threads and processes differs from one operating system to another, but in most cases, a thread is contained inside a process.

Posix Threads - Overview

opiframe
Knowledge equals power

- Multiple threads can exist within the same process and share resources such as memory, while different processes do not share these resources.

- On a single processor, multithreading generally occurs by time-division multiplexing.

Posix Threads - Overview

opiframe
Knowledge equals power

- On a multiprocessor or multi-core system, the threads or tasks will generally run at the same time, with each processor or core running a particular thread or task.

- Threads differ from traditional multitasking operating system processes in that:

Posix Threads - Overview

opiframe
Knowledge equals power

- Processes are typically independent, while threads exist as subsets of a process

- Processes carry considerable state information, whereas multiple threads within a process share state as well as memory and other resources

- Processes have separate address spaces, whereas threads share their address space

Posix Threads - Overview

opiframe
Knowledge equals power

- Processes interact only through system-provided inter-process communication mechanisms.
- Context switching between threads in the same process is typically faster than context switching between processes.

Posix Threads - Overview

opiframe
Knowledge equals power

- Threads also happen to be extremely nimble.
- Compared to a standard **fork()**, they carry a lot less overhead.
- The kernel does not need to make a new independent copy of the process memory space, file descriptors, etc.

Posix Threads - Overview

opiframe
Knowledge equals power

- That saves a lot of CPU time, making thread creation ten to a hundred times faster than new process creation.

- Because of this, you can use a whole bunch of threads and not worry too much about the CPU and memory overhead incurred.

Posix Threads - Overview

opiframe
Knowledge equals power

- You don't have a big CPU hit the way you do with **fork()**.
- This means you can generally create threads whenever it makes sense in your program.

Posix Threads - Creation

- Posix Threads are handled by a specific library Pthread library.
- You need to add **–lpthread** option to your gcc compile command.
- This way the compiler recognises that you are infact using an extra library.

Posix Threads - Creation

opiframe
Knowledge equals power

- Threads are created with a system call
- int **pthread_create**(pthread_t *restrict thread, const pthread_attr_t *restrict attr, void *(*start_routine)(void*), void *restrict arg);

- First argument is the thread id,
- second the attributes for the thread itself,
- third is the start function that the thread operates
- and fourth is the argument given to the start function.

Posix Threads - Creation

opiframe
Knowledge equals power

- When a process creates another new process, using **fork()**, the new process is considered the child and the original process is considered the parent.

- With POSIX threads this hierarchical relationship doesn't exist.

Posix Threads - Creation

opiframe
Knowledge equals power

- While a main thread may create a new thread, and that new thread may create an additional new thread, the POSIX threads standard views all your threads as a single pool of equals.

- So the concept of waiting for a child thread to exit doesn't make sense.

Posix Threads - Creation

opiframe
Knowledge equals power

- However threads do take one slot from the kernel scheduler table like processes.
- So we need a way to remove unwanted threads when they have finished.
- Again we might be interested in the exit status of the thread.

Posix Threads - Creation

opiframe
Knowledge equals power

- For this we have joining:
- **int pthread_join(pthread_t thread, void **value_ptr);**

- This function causes your thread to block until such time that the thread specified by the first argument has finished.

- Then this function will return with the exit status of the joined thread stored at **value_ptr**.

- Note that any thread can join any other thread.

Posix Threads - Creation

opiframe
Knowledge equals power

- If we are not interested in the exit status of the thread we can specify that with:
- **int pthread_detach(pthread_t thread);**

- This function causes the thread specified to be considered detached so that we are not interested in the exit status. Kernel is free to release all resources of this thread when thread terminates.

Posix Threads - Termination

opiframe
Knowledge equals power

- A thread can terminate in three different ways:
- Exiting the outermost function (the start routine)
- Thread can call function **int pthread_exit(void *status)** where status is the return status.
- And some other thread can call **int pthread_cancel(pthread_t thread)** on a specific thread causing it to terminate.

Posix Threads – other operations

opiframe
Knowledge equals power

- There are a few basic operations with threads that need to be considered.

- If you need the thread ID of the current thread:
- **pthread_t pthread_self(void);**

- This returns the pthread_t id of the current thread.

Posix Threads – other operations

- To compare the thread identification numbers of two threads.
- **int pthread_equal(pthread_t tid1, pthread_t tid2);**

- As with other comparison functions, **pthread_equal()** returns a non-zero value when **tid1** and **tid2** are equal; otherwise, zero is returned. When either **tid1** or **tid2** is an invalid thread identification number, the result is unpredictable.

Posix Threads – other operations

opiframe
Knowledge equals power

- For dynamic initialization of environments or libraries:
- **int pthread_once(pthread_once_t *_once_control_,**
  void (*_init_routine_)(void));

- The first call to **pthread_once()** by any thread in a process, with a given **once_control**, shall call the **init_routine** with no arguments. Subsequent calls of **pthread_once()** with the same **once_control** shall not call the **init_routine**. On return from **pthread_once()**, **init_routine** shall have completed. The **once_control** parameter shall determine whether the associated initialization routine has been called.

Posix Threads – other operations

opiframe
Knowledge equals power

- For sending signals to a specific thread:
- **int pthread_kill(pthread_t *thread*, int *sig*);**
- The **pthread_kill()** function shall request that a signal be delivered to the specified thread.
- As in **kill()**, if *sig* is zero, error checking shall be performed but no signal shall actually be sent.
- Note that **pthread_kill()** only causes the signal to be handled in the context of the given thread; the signal action (termination or stopping) affects the process as a whole.

Posix Threads – Thread attributes

- Attributes are a way to specify behavior that is different from the default.

- When a thread is created with **pthread_create()** or when a synchronization variable is initialized, an attribute object can be specified.

Posix Threads – Thread attributes

opiframe
Knowledge equals power

- Attributes are specified only at thread creation time; they cannot be altered while the thread is being used.
- An attribute object is opaque, and cannot be directly modified by assignments.
- A set of functions is provided to initialize, configure, and destroy each object type.

Posix Threads – Thread attributes

opiframe
Knowledge equals power

- Attributes are specified only at thread creation time; they cannot be altered while the thread is being used.
- An attribute object is opaque, and cannot be directly modified by assignments.
- A set of functions is provided to initialize, configure, and destroy each object type.

Posix Threads – Thread attributes

opiframe
Knowledge equals power

- The suggested method for using attributes is to configure all required state specifications at one time in the early stages of program execution.

- The appropriate attribute object can then be referred to as needed.

Posix Threads – Thread attributes

opiframe
Knowledge equals power

- Using attribute objects has two primary advantages:
- It adds to code portability.
- State specification in an application is simplified.

Posix Threads – Thread attributes

- There are following attributes that can be set for each thread:
- scope (either system or process)
- detachstate (either joinable or detached)
- stackaddress
- stacksize
- whether the thread inherts the scheduling from creator

Posix Threads – Thread attributes

opiframe
Knowledge equals power

- The attribute init is done with:
- **int pthread_attr_init(pthread_attr_t *tattr);**

- The function **pthread_attr_init()** initialises a thread attributes object **attr** with the default value for all of the individual attributes used by a given implementation.

- The resulting attribute object (possibly modified by setting individual attribute values), when used by **pthread_create()**, defines the attributes of the thread created. A single attributes object can be used in multiple simultaneous calls to **pthread_create()**.

Posix Threads – Thread attributes

opiframe
Knowledge equals power

- The function **pthread_attr_destroy()** is used to remove the storage allocated during initialization.

- **int pthread_attr_destroy(pthread_attr_t *tattr);**

- An implementation may cause **pthread_attr_destroy()** to set *attr* to an implementation-dependent invalid value. The behaviour of using the attribute after it has been destroyed is undefined.

Posix Threads – Thread attributes

opiframe
Knowledge equals power

- Example of an attribute is the joinable attribute.
- It comes in two states: PTHREAD_CREATE_JOINABLE and PTHREAD_CREATE_DETACHED.
- Use function **pthread_attr_setdetachstate()** which is prototyped by:
- **int pthread_attr_setdetachstate(pthread_attr_t *tattr,int detachstate);**

Posix Threads – Mutexes

opiframe
Knowledge equals power

- Mutual exclusion (often abbreviated to mutex) algorithms are used in concurrent programming to avoid the simultaneous use of a common resource, such as a global variable, by pieces of computer code called critical sections.

- A critical section is a piece of code where a process or thread accesses a common resource.

Posix Threads – Mutexes

opiframe
Knowledge equals power

- This is how mutexes work:
- If thread "a" tries to lock a mutex while thread "b" has the same mutex locked, thread "a" goes to sleep.

- As soon as thread "b" releases the mutex, thread "a" will be able to lock the mutex.

- In other words, it will return from the function call with the mutex locked.

Posix Threads – Mutexes

opiframe
Knowledge equals power

- Initializing mutex is done with
- **int pthread_mutex_init(pthread_mutex_t *mymutex, const pthread_mutexattr_t *attr)**

- The **pthread_mutex_init()** function shall initialize the mutex referenced by *mutex* with attributes specified by *attr*. If *attr* is NULL, the default mutex attributes are used; the effect shall be the same as passing the address of a default mutex attributes object. Upon successful initialization, the state of the mutex becomes initialized and unlocked.

Posix Threads – Mutexes

opiframe
Knowledge equals power

- Locking and unlocking is handled with three functions:
- **int pthread_mutex_lock(pthread_mutex_t *_mutex_);**
  int pthread_mutex_trylock(pthread_mutex_t *_mutex_);
  int pthread_mutex_unlock(pthread_mutex_t *_mutex_);

- The mutex object referenced by *mutex* shall be locked by calling **pthread_mutex_lock().** If the mutex is already locked, the calling thread shall block until the mutex becomes available. This operation shall return with the mutex object referenced by *mutex* in the locked state with the calling thread as its owner.

Posix Threads – Mutexes

opiframe
Knowledge equals power

- The **pthread_mutex_trylock()** function shall be equivalent to **pthread_mutex_lock(),** except that if the mutex object referenced by *mutex* is currently locked (by any thread, including the current thread), the call shall return immediately.

- The **pthread_mutex_unlock()** function shall release the mutex object referenced by *mutex*. The manner in which a mutex is released is dependent upon the mutex's type attribute. If there are threads blocked on the mutex object referenced by *mutex* when **pthread_mutex_unlock()** is called, resulting in the mutex becoming available, the scheduling policy shall determine which thread shall acquire the mutex.

Posix Threads – Mutexes

opiframe
Knowledge equals power

- Let’s say we are waiting for a certain condition to fulfill in order for our thread to start its business.

- We could just lock a mutex, poll the necessary information and incase of the condition not met, unlock the mutex and redo the whole thing at later time. This is generally not a good idea.

Posix Threads – Mutexes

opiframe
Knowledge equals power

- To handle situations like this pthread library offers an object called wait condition. Using a wait condition works as follows.

- Let's consider a scenario where a thread has locked a mutex, in order to take a look at a linked list, and the list happens to be empty.

Posix Threads – Mutexes

- This particular thread can't do anything -- it's designed to remove a node from the list, and there are no nodes available.

- While still holding the mutex lock, our thread will call **pthread_cond_wait(&mycond, &mymutex).**

Posix Threads – Mutexes

opiframe
Knowledge equals power

- The first thing **pthread_cond_wait()** does is simultaneously unlock the mutex **mymutex** (so that other threads can modify the linked list) and wait on the condition **mycond**.

- Unlocking the mutex happens immediately, but waiting on the condition **mycond** is normally a blocking operation, meaning that our thread will go to sleep.

Posix Threads – Mutexes

- Let's say that another thread (call it "thread 2") locks **mymutex** and adds an item to our linked list.

- Immediately after unlocking the mutex, thread 2 calls the function **pthread_cond_broadcast(&mycond)**.

Posix Threads – Mutexes

opiframe
Knowledge equals power

- By doing so, thread 2 will cause all threads waiting on the mycond condition variable to immediately wake up.

- **pthread_cond_wait()** will perform one last operation: relock **mymutex**. Once **pthread_cond_wait()** has the lock, it will then return and allow thread 1 to continue execution.

Posix Threads – Mutexes

opiframe
Knowledge equals power

- By doing so, thread 2 will cause all threads waiting on the mycond condition variable to immediately wake up.

- **pthread_cond_wait()** will perform one last operation: relock **mymutex**. Once **pthread_cond_wait()** has the lock, it will then return and allow thread 1 to continue execution.

Posix Threads – Mutexes

opiframe
Knowledge equals power

- Initializing the condition:
- **int pthread_cond_init(pthread_cond_t *restrict *cond*, const pthread_condattr_t *restrict *attr*);**

- The **pthread_cond_init()** function shall initialize the condition variable referenced by *cond* with attributes referenced by *attr*. If *attr* is NULL, the default condition variable attributes shall be used; the effect is the same as passing the address of a default condition variable attributes object. Upon successful initialization, the state of the condition variable shall become initialized.

Posix Threads – Mutexes

opiframe
Knowledge equals power

- Waiting can be done with two functions:
- **int pthread_cond_timedwait(pthread_cond_t *restrict *cond*,**
  pthread_mutex_t *restrict *mutex*,
  const struct timespec *restrict *abstime*);
  int pthread_cond_wait(pthread_cond_t *restrict *cond*,
  pthread_mutex_t *restrict *mutex*);

- The ***pthread_cond_timedwait*()** and ***pthread_cond_wait*()** functions shall block on a condition variable. They shall be called with *mutex* locked by the calling thread or undefined behavior results.

Posix Threads – Mutexes

opiframe
Knowledge equals power

- These functions atomically release *mutex* and cause the calling thread to block on the condition variable *cond*; atomically here means "atomically with respect to access by another thread to the mutex and then the condition variable".

- That is, if another thread is able to acquire the mutex after the about-to-block thread has released it, then a subsequent call to ***pthread_cond_broadcast*()** or ***pthread_cond_signal*()** in that thread shall behave as if it were issued after the about-to-block thread has blocked.

Posix Threads – Mutexes

opiframe
Knowledge equals power

- Informing that a condition has been met:
- **int pthread_cond_broadcast(pthread_cond_t *_cond_);**
  int pthread_cond_signal(pthread_cond_t *_cond_);

- The **pthread_cond_broadcast()** function shall unblock all threads currently blocked on the specified condition variable *cond*.

- The **pthread_cond_signal()** function shall unblock at least one of the threads that are blocked on the specified condition variable *cond* (if any threads are blocked on *cond*).

Posix Threads – Mutexes

- If more than one thread is blocked on a condition variable, the scheduling policy shall determine the order in which threads are unblocked.

- When each thread unblocked as a result of a **pthread_cond_broadcast()** or **pthread_cond_signal()** returns from its call to **pthread_cond_wait()** or **pthread_cond_timedwait()**, the thread shall own the mutex with which it called **pthread_cond_wait()** or **pthread_cond_timedwait()**

Posix Threads – Thread Local Storage

opiframe
Knowledge equals power

- When data structures are shared by a group of threads and at least one thread is writing into them, synchronization between the threads is sometimes necessary to make sure that all threads see a consistent view of the shared data.

- A typical synchronized access regime for threads in this situation is for a thread to acquire a lock, read or write the shared data structures, then release the lock.

Posix Threads – Thread Local Storage

opiframe
Knowledge equals power

- All forms of locking have overhead to maintain the lock data structures, and they use atomic instructions that slow down modern processors.

- Synchronization also slows down the program, because it eliminates parallel execution inside the synchronized code, forming a serial execution bottleneck.

- Therefore, when synchronization occurs within a time-critical section of code, code performance can suffer

Posix Threads – Thread Local Storage

opiframe
Knowledge equals power

- The synchronization can be eliminated from the multithreaded, time-critical code sections if the program can be re-written to use thread-local storage instead of shared data structures.

- This is possible if the nature of the code is such that real-time ordering of the accesses to the shared data is unimportant.

- Synchronization can also be eliminated when the ordering of accesses is important, if the ordering can be safely postponed to execute during infrequent, non-time-critical sections of code.

Posix Threads – Thread Local Storage

opiframe
Knowledge equals power

- An additional advantage of using thread-local storage during time-critical portions of the program is that the data may stay live in a processor's cache longer than shared data, if the processors do not share a data cache.

- When the same address exists in the data cache of several processors and is written by one of them, it must be invalidated in the caches of all other processors, causing it to be re-fetched from memory when the other processors access it.

- But thread-local data will never be written by any other processors than the one it is local to and will therefore be more likely to remain in the cache of its processor.

Posix Threads – Thread Local Storage

opiframe
Knowledge equals power

- One must be careful about the trade-offs involved in this technique.
- The technique does not remove the need for synchronization, but only moves the synchronization from a time-critical section of the code to a non-time-critical section of the code.
- First, determine whether the original section of code containing the synchronization is actually being slowed down significantly by the synchronization.

Posix Threads – Thread Local Storage

opiframe
Knowledge equals power

- Second, determine whether the time ordering of the operations is critical to the application.
- If not, synchronization can be removed.
- If time ordering is critical, can the ordering be correctly reconstructed later?

Posix Threads – Thread Local Storage

opiframe
Knowledge equals power

- Third, verify that moving synchronization to another place in the code will not cause similar performance problems in the new location.

- One way to do this is to show that the number of synchronizations will decrease dramatically because of your work

Posix Threads – Thread Local Storage

opiframe
Knowledge equals power

- To use TLS (Thread Local Storage using Pthreads API you must do following:

- A **key** must be *created* using the **pthread_key_create().**

- Data that needs to have thread-scope(e.g file descriptor in a web server) should be *associated* with the key using **pthread_setspecific()**

Posix Threads – Thread Local Storage

opiframe
Knowledge equals power

- Associated data can be *retrieved* later using **pthread_getspecific()**
- **int pthread_key_create(pthread_key_t *key, void (*destructor)(void*));**
- The pthread_key_create() function shall create a thread-specific data key visible to all threads in the process. Key values provided by pthread_key_create() are opaque objects used to locate thread-specific data. Although the same key value may be used by different threads, the values bound to the key by pthread_setspecific() are maintained on a per-thread basis and persist for the life of the calling thread**.**

Posix Threads – Thread Local Storage

opiframe
Knowledge equals power

- Upon key creation, the value NULL shall be associated with the new key in all active threads. Upon thread creation, the value NULL shall be associated with all defined keys in the new thread.

- An optional destructor function may be associated with each key value. At thread exit, if a key value has a non-NULL destructor pointer, and the thread has a non-NULL value associated with that key, the value of the key is set to NULL, and then the function pointed to is called with the previously associated value as its sole argument. The order of destructor calls is unspecified if more than one destructor exists for a thread when it exits.

Posix Threads – Thread Local Storage

- **int pthread_setspecific(pthread_key_t key, const void *value);**
- **void *pthread_getspecific(pthread_key_t key);**

- The **pthread_setspecific**() function associates a thread-specific value with a key obtained via a previous call to **pthread_key_create**(). Different threads may bind different values to the same key. These values are typically pointers to blocks of dynamically allocated memory that have been reserved for use by the calling thread.

Posix Threads – Thread Local Storage

opiframe
Knowledge equals power

- The **pthread_getspecific**() function returns the value currently bound to the specified key on behalf of the calling thread.

- The effect of calling **pthread_setspecific**() or **pthread_getspecific**() with a key value not obtained from **pthread_key_create**() or after key has been deleted with **pthread_key_delete**() is undefined.