POSIX Threads Quick Guide
v 0.1.1

Adam Grossman <adamtg@metashadow.com>

Definitions

Thread safe/re-entrant
NOTE: Every re-entrant function is thread safe, but not every thread-safe function is re-entrant.

Concept	Description
thread-safe	shared data is safe. A thread-safe function must: • if it has shared resources, it is protected through locks • must not have any side effects
re-entrant	can safely be entered by multiple threads. A re-entrant function must: • not have non-constant static or global variables can be used • not return any pointers pointing to the stack • can only use data supplied by the caller to the function • not call any non re-entrant functions

Thread Detach State

Concept	Description
joinable	Joinable threads can be waited on. The call to wait on the thread will wait until the joined thread terminates. If a joinable thread is not waited on, the resources will not be relinquished.
detached	The resources of the thread are returned immediately to system and the thread can't be waited on, as it can be with a joinable thread.

Creating Threads

Thread Creation
Asynchronously creates and starts a thread.

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

Field	Purpose	Value
int (return value)	return value of call	**0:** no error **EAGAIN**: not enough resources to create thread, or thread limit (PTHREAD_THREADS_MAX) is reached **EINVAL:** *attr* is invalid **EPERM**: caller does not have permissions to create threads
**pthread_t ***tid	returns the process unique ID of the	pointer to **pthread_t** variable

	created thread.	
const pthread_attr_t **attr*	sets the various attributes for the thread	**NULL**: set defaults attributes. See section **Appendix A: Thread Attributes** for details
void *(*thread_function)(void*)	the function that will run as the thread	a function that returns a void pointer, and it's single argument is a void pointer
void * *arg*	pointer to the data that is the argument to `thread_function`	void pointer

Thread Control

Waiting for a thread to exit

`pthread_join` suspends the calling thread until the thread that is being waited on terminates. The thread that is ebing waited on must be a joinable thread, or `pthread_join` will exit with an error.

```
void pthread_join(pthread_t thread_id, void **value_ptr)
```

Field	Purpose	Value
int (return value)	return value	0: successful **EINVAL**: specified thread is not joinable **ESRCH**: invalid thread id **EDEADLK**: deadlock was detected or thread id is the thread id of the calling thread
pthread_t thread_id	id of thread to be waited on	Any thread id that is in a `JOINABLE` state
**void ****value_ptr	allows the exiting thread to pass exit data	pointer value of argument passed via the **pthread_exit**

Exit Thread

Exits the currently running thread. If there is a `pthread_join` waiting on the thread, control is returned to the threading making the `pthread_join` call. Before the thread actually exits, it pops off the functions on the cleanup stack (see **Thread Cleanup**)

```
void pthread_exit(void *value)
```

Field	Purpose	Value
**void ***value	Return value to the `pthread_join` call. This value is only used when the exiting thread is waited upon by the `pthread_join` call. If `value` points to resources within the exiting threads private resource (i.e. thread stack), the pointer will be undefined.	A pointer to any globally available memory location

Sending Signal to a Thread

Sends a signal to a specific thread

```
int pthread_kill(pthread_t  thread_id, int sig)
```

Field	Purpose	Value
int (return value)	return value	0: successful **ESRCH**: thread id is invalid **EINVAL**: invalid signal number
pthread_t thread_id	thread to send signal to	valid thread
int sig	signal to send to thread	valid signal number

Detach a running thread
If a thread is currently running, it can be set to a detach state. It will not cause the thread to terminate.

```
int pthread_detach(pthread_t thread_id)
```

Field	Purpose	Value
int (return value)	return value	0: successful **EINVAL**: thread is not a joinable thread **ESRCH**: invalid thread
pthread_t thread_id	id of thread to be detached	valid thread id

Get Thread ID
Get the currently running thread's id

```
pthread_t pthread_detach()
```

Field	Purpose	Value
pthread_t	return value	The current running thread's id. This function does not return an error value.

Thread Cancellation
Thread cancellation allows a controlled termination of threads. The cancellation request is sent to the thread, and the thread decides how to handle the request. Before the thread actually exits, it pops off the functions on the cleanup stack (see **Thread Cleanup**)

There are two states:

State	Description
Enable	Cancellation requests are sent to the threads. Any held requests will be immediately sent.
Disable	The cancellation request is held, and not actually sent to the target thread.

If the state is Enable, there are two types of cancellations

Type	Description
Deferred	Cancellation is held off until a cancellation point is reached
Asynchronous	Cancellation can happen at anytime

The default state is cancellation Enable and the default type is Deferred

Setting the cancellation states
The call which actually cancels a thread is asynchronous and does not wait for the thread to terminate.

```
int pthread_setcancelstate(int state, int *prev_state)
```

Field	Purpose	Value
`int (return value)`	return value	0: successful `EINVAL`: state is not a valid state
`int state`	the cancel state to set the thread	`PTHREAD_CANCEL_ENABLE` `PTHREAD_CANCEL_DISABLE`
`int *prev_state`	the previous cancel state of the thread	`PTHREAD_CANCEL_ENABLE` `PTHREAD_CANCEL_DISABLE`

Setting the cancellation type

```
int pthread_setcanceltype(int type, int *prev_type)
```

Field	Purpose	Value
`int (return value)`	return value	0: successful EINVAL: state is not a valid state
`int type`	the cancel type to set the thread	PTHREAD_CANCEL_DEFERRED PTHREAD_CANCEL_ASYNCHRONOUS
`int *prev_type`	the previous cancel type of the thread	PTHREAD_CANCEL_DEFERRED PTHREAD_CANCEL_ASYNCHRONOUS

Sending a cancellation to a thread
This allows a thread to request a thread to cancel. This is a asynchronous call, so it does not wait for the destination thread to actual cancel.

```
int pthread_cancel(pthread_t thread_id)
```

Field	Purpose	Value
`int (return value)`	return value	0: successful ESRCH: invalid thread
`pthread_t thread_id`	id of the thread the cancel is sent to	any valid thread id

Deferred Cancellation
When a thread is in deferred cancellation state, the thread terminates only at determined points. Certain functions are considered cancellation points [list needs to be added to an appendix]. A cancellation point can also explicit be set by calling `pthread_testcancel`.

```
void pthread_testcancel()
```

There are no return values or arguments. When the function is called, and there is a pending cancel, the thread will be canceled, otherwise it continues processing as if nothing has happened.

Asynchronous Cancellation
There is no safety checks, and the thread can be canceled at anytime.

Thread Cleanup

Each thread has a stack where the callbacks for thread cleanup are placed. When a thread exits due to cancellation or a call to `pthread_exit`, it pops off each and runs each call. The cleanup stack is not called when a thread terminates due to a call to `return`. The cleanup calls can explicitly be ran by manually popping the calls off the stack.

NOTE: **`pthread_cleanup_push`** and **`pthread_cleanup_pop`** are sometime implemented as macros, and need to be used in matching pairs in the nested level.

Push Function on the Stack

```
void pthread_cleanup_push(void (*func)(void *), void *arg)
```

Field	Purpose	Value
`void (*func)(void *)`	pointer to a callback function	pointer to a function that returns void and has a void pointer as it's only argument
`void *arg`	pointer to data which will be the argument to the callback	void pointer

Pop Function off the Stack

```
void pthread_cleanup_pop(int execute)
```

Field	Purpose	Value
`int execute`	controls whether the popped callback will be executed or not	0: pop the callback, but do not actually run it >0: pop and run the callback

Synchronization

Mutex

Allows for synchronization between threads. Only one thread can own the lock, and the other threads that are trying to obtain the lock block until the thread that owns the lock unlocks the mutex. When the lock is unlocked, and other threads are waiting on the mutec, the scheduling policy dictates which waiting thread gets the mutex.

Creating
Initializing an already initialized mutex will either produce an error, or is undefined. This behavior is system dependent.

```
int pthread_mutex_init(pthread_mutex_t *mutex, const pthread_mutexattr_t *attr)
```

Field	Purpose	Value
`int (return value)`	return value	0: successful **`EAGAIN`**: out resources (other than memory) to create another mutex **`ENOMEM`**: out of memory **`EPERM`**: insufficient privileges **`EBUSY`**: trying to reinitialize an already initialized mutex (this error is system

		dependent) **`EINVAL`**: attr is invalid
`pthread_mutex_t` *`*mutex`*	mutex that is to be initialized	pointer to `pthread_mutex_t`
`pthread_mutexattr_t` *`*attr`*	mutex attributes	**`NULL`**: use defaults see **Appending B: Mutex Attributes** for attributes details

```
pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
```

This is a macro to initialize a statically allocated mutex with default attributes.

Destroying
This will turn the mutex into an uninitialized mutex. It does not release any memory/resources. Attempting to destroy a locked mutex is undefined or returns an error. This behavior is system dependent.

```
int pthread_mutex_destroy(pthread_mutex_t *mutex)
```

Field	Purpose	Value/Description
`int` `(return value)`	return value	**0**: success **`EBUSY`**: mutex is locked or being waited on (this error is system dependent) **`EINVAL`**: mutex is invalid (this error is system dependent)
`pthread_mutex_t` *`*mutex`*	mutex that is to be initialized	pointer to `pthread_mutex_t`

Locking (Blocking)
Tries to lock the mutex. If the mutex is already locked, the call waits until the mutex is unlocked and available to the calling thread.

```
int pthread_mutex_lock(pthread_mutex_t *mutex)
```

Field	Purpose	Value/Description
`int` `(return value)`	return value	**0**: success EINVAL: if protocol type is `PTHREAD_PRIO_PROTECT` and the calling threads priority is higher then the mutex's priority ceiling EINVAL: invalid mutex EAGAIN: if type is `PTHREAD_MUTEX_RECURSIVE` and max number of recursive locks has been exceeded EDEADLK: if type is `PTHREAD_MUTEX_ERRORCHECK` and the thread already owns the lock for the mutex
`pthread_mutex_t` *`*mutex`*	mutex that is to be locked	pointer to `pthread_mutex_t`

Locking (Non-Blocking)
Does not block if it can not lock the mutex

```
int pthread_mutex_trylock(pthread_mutex_t *mutex)
```

Field	Purpose	Value/Description
int (return value)	return value	**0**: success **EINVAL**: if protocol type is PTHREAD_PRIO_PROTECT and the calling threads priority is higher then the mutex's priority ceiling EBUSY: can not be locked because it is already locked **EINVAL**: invalid mutex **EAGAIN**: if type is PTHREAD_MUTEX_RECURSIVE and max number of recursive locks has been exceeded
pthread_mutex_t *mutex	mutex that is to be locked	pointer to pthread_mutex_t

Unlock
Releases the lock and allows another thread to obtain it.

```
int pthread_mutex_unlock(pthread_mutex_t *mutex)
```

Field	Purpose	Value/Description
int (return value)	return value	**0**: success **EINVAL**: invalid mutex **EAGAIN**: if type is PTHREAD_MUTEX_RECURSIVE and max number of recursive locks has been exceeded **EAPERM**: thread does not own the mutex
pthread_mutex_t *mutex	mutex that is to be unlock	pointer to pthread_mutex_t

TBD

condition variables
threads and signals

stack management

posix keys?
posix scheduling?

Appendix A

Thread Attributes

Thread attributes are created by using different calls to set a pthread_attr_t structure. See section **Getting/Setting Attributes** for defaults attribute settings

Initializing/Destroying Thread Attributes

```
int pthread_attr_init(pthread_attr_t *attr);
```

Initializes the attribute structure, and sets all the fields to the default values. It can used in multiple calls to **pthread_create**.

Field	Purpose	Value/Description
int (return value)	return value	**0**: success **ENOMEM**: insufficient memory to initialize structure
pthread_attr_t *attr	structure to be initializes	pointer to **pthread_attr_t** structure.

The default values are: (the description of each field/values are explained under **Get/Set Attribute Functions** section) :

Description	Default Value
Detach State	PTHREAD_CREATE_JOINABLE
Scheduling Policy	SCHED_OTHER
Schedule Parameter (Priority)	0
Inherit Schedule	PTHREAD_EXPLICIT_SCHED
Scope	PTHREAD_SCOPE_SYSTEM
Stack Address	N/A
Stack Size	PTHREAD_STACK_MIN
Guard Size	1 page. (ignored if user managed stack)

```
int pthread_attr_destroy(pthread_attr_t *attr);
```

Sets the structure value to an invalid value. Does not set the pointer to an invalid value, just the values of the structure to invalid values.

Field	Purpose	Value/Description
int (return value)	return value	**0**: success **ENOMEM**: insufficient memory to initialize structure
pthread_attr_t *attr	structure to be made invalid	pointer to **pthread_attr_t** structure.

Get/Set Attribute Functions

Each attribute has it's own get/set call. Each call is pretty much the same, so only the meaning of the options will be described.

Default attributes are marked with an asterisk (*).

Detach State

```
int pthread_attr_setdetachstate(pthread_attr_t *attr, int detachstate);
```

```
int pthread_attr_getdetachstate(const pthread_attr_t *attr, int *detachstate);
```

Attribute	Description
PTHREAD_CREATE_DETACHED*	Detached thread
PTHREAD_CREATE_JOINABLE	Joinable thread

Guardsize

NOTE: Guardsize attribute is ignored for user managed thread stacks

```
int pthread_attr_setgaurdsize(pthread_attr_t *attr, size_t gaurdsize);
int pthread_attr_getgaurdsize(const pthread_attr_t *attr, size_t *gaurdsize);
```

Attribute	Description
size in bytes (DEFAULT: 1 page size*)	Guard size is the number of bytes used for to protect against stack overflows.

Schedule Parameter

```
int pthread_attr_setschedparam(pthread_attr_t *attr, struct sched_param *schedparam);
int pthread_attr_getschedparam(const pthread_attr_t *attr, const struct sched_param
*schedparam);
```

Attribute	Description
Scheduling priority parameters structure	Look up "**struct sched_param**" for structure details

Schedule Inheritance

```
int pthread_attr_setinheritsched(pthread_attr_t *attr, int schedule);
int pthread_attr_getinheritsched(const pthread_attr_t *attr, int *schedule);
```

Attribute	Description
PTHREAD_EXPLICIT_SCHED*	Get scheduling parameters from the attribute structure.
PTHREAD_INHERIT_SCHED	Inherit scheduling from creating thread. The schedule parameter attribute will be ignored.

Schedule Policy

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

Attribute	Description
SCHED_OTHER*	Implementation specific
SCHED_FIFO	First in, first out
SCHED_RR	Round Robin

Stack Address

```
int pthread_attr_setstackaddr(pthread_attr_t *attr, void *stackaddress);
int pthread_attr_getstackaddr(const pthread_attr_t *attr, void **stackaddress);
```

Attribute	Description
Memory location	Pointer to the memory location that will be used as the threads stack.

Stack Size

```
int pthread_attr_setstacksize(pthread_attr_t *attr, size_t stacksize);
int pthread_attr_setstacksize(const pthread_attr_t *attr, size_t *stacksize);
```

Attribute	Description
Stack size	minimum stack size for the thread (default: PTHREAD_STACK_MIN)

Appendix B
Mutex Attributes

Initialize Attributes Structure
The structure is initialized to default values

```
int pthread_mutexattr_init(pthread_mutexattr_t *attr)
```

Field	Purpose	Value/Description
int (return value)	return value	**0**: success **ENOMEM**: insufficient memory to initialize structure
pthread_mutexattr_t *attr	structure to be initializes	pointer to **pthread_mutexattr_t** structure.

The default values are: (the description of each field/values are explained under **Get/Set Attribute Functions** section) :

Description	Default Value
Shared	PTHREAD_PROCESS_PRIVATE
Type	PTHREAD_MUTEX_DEFAULT
Protocol	PTHREAD_PRIO_NONE
Priority Ceiling	implementation specific

Get/Set Attribute Functions
Each attribute has it's own get/set call. Each call is pretty much the same, so only the meaning of the options will be described.

Default attributes are marked with an asterisk (*).

Shared

```
int pthread_mutexattr_setpshared(pthread_mutexattr_t *attr, int shared)
int pthread_mutexattr_getpshared(const pthread_mutexattr_t *attr, int *shared)
```

Attribute	Description
PTHREAD_PROCESS_PRIVATE *	Only threads in the same process can operate an the mutex
PTHREAD_PROCESS_SHARED	Any process that has access to the mutex's memory can operate on the mutex

Type

```
int pthread_mutexattr_settype(pthread_mutexattr_t *attr, int type)
int pthread_mutexattr_gettype(const pthread_mutexattr_t *attr, int *type)
```

Attribute	Description
PTHREAD_MUTEX_DEFAULT *	Attempts to relock a locked mutex in the same thread is undefined. Unlocking the mutex by thread other than the thread that locked is undefined. Unlocking an unlocked thread is undefined.

PTHREAD_MUTEX_NORMAL	Attempts to relock a locked mutex in the same thread will create a deadlock. Unlocking the mutex by thread other than the thread that locked will return an error. Unlocking an unlocked thread will return an error.
PTHREAD_MUTEX_ERRORCHECK	Attempts to relock a locked mutex in the same thread will return an error. Unlocking the mutex by thread other than the thread that locked will return an error. Unlocking an unlocked thread will return an error.
PTHREAD_MUTEX_RECURSIVE	Allows the same thread to relock a locked thread, but it requires the same number of unlocks to completely unlock the mutex. Unlocking the mutex by thread other than the thread that locked will return an error. Unlocking an unlocked thread will return an error.

Protocol
Allows for the priority of the thread be affected when it gets the mutex

```
int pthread_mutexattr_setprotocol(pthread_mutexattr_t *attr, int protocol)
int pthread_mutexattr_getprotocol(const pthread_mutexattr_t *attr, int *protocol)
```

Attribute	**Description**
PTHREAD_PRIO_NONE*	Priorities are not affected
PTHREAD_PRIO_INHERIT	If the thread is blocking other threads that are waiting for the mutex, and those threads have a higher priority, the thread with the mutex inherits the highest priority of the blocked threads
PTHREAD_PRIO_PROTECT	If the thread holds several mutexes, the thread will get the highest priority ceiling for those mutexes (regardless of the protocol the others are set to). Priority Ceiling attribute must be set.

Priority Ceiling

```
int pthread_mutexattr_setprioceiling(pthread_mutexattr_t *attr, int
priority_ceiling)
int pthread_mutexattr_getprioceiling(const pthread_mutexattr_t *attr, int
*priority_ceiling)
```

Attribute	**Description**
integer	the minimum priority the thread will run in while the thread has the mutex. To avoid priority inversion, the priority will be set the highest priority of all threads that may lock the mutex.

Destroy
Unitializes an attribute structure

```
int pthread_mutexattr_destroy(pthread_mutexattr_t *attr)
```

Field	Purpose	Value/Description
`int` `(return value)`	return value	**0**: success **`EINVAL`**: invalid attribute
`pthread_mutexattr_t` *`*attr`*	structure to be initializes	pointer to **`pthread_mutexattr_t`** structure.

FAQ

q1: What happens to the threads when the calling thread exits?

a1: If the calling thread exits by calling **`exit`**, or exit due to an uncaught signal, the threads will also terminate, and will not pop off any cleanup routines. if the calling thread exits (even it is the the main processes itself) by calling **`pthread_exit`**, the threads will continue to run.

License

Document Changes

Date	Version	Name	Change
10/24/09	0.01	Adam Grossman <adamtg@metashadow.com>	Initial Release
11/17/09	0.05	Adam Grossman <adamtg@metashadow.com>	Added in the main thread creation/attributes functions
11/25/09	0.1	Adam Grossman <adamtg@metashadow.com>	Added in mutex Reformatted First Release
12/06/09	0.1.1	Adam Grossman <adamtg@metashadow.com>	Added in notes in the clean-up routines. Added in FAQ sections