Table of Contents

Chapter 1. Locking Interface

1.1. Why use it ?

The main reason in creating it was to have a single transparent interface to various locking methods. For example right now SIP Express Router uses the following locking methods, depending on their availability on the target system.

- *FAST_LOCK*

  Fast inline assembly locks, defined in `fast_lock.h`. They are currently available for x86, sparc64, strongarm (amv4l) and ppc (external untested contributed code). In general if the assembly code exists for a given architecture and the compiler knows inline assembly (for example sun cc does not) FAST_LOCK is prefered. The main advantage of using FAST_LOCK is very low memory overhead and extremely fast lock/unlock operations (like 20 times faster then SYSV semaphores on linux & 40 times on solaris). The only thing that comes close to them are pthread mutexes (which are about 3-4 times slower).

- *PHTREAD_MUTEX*

  Uses pthread_mutex_lock/unlock. They are quite fast but they work between processes only on some systems (they do not work on linux).

- *POSIX_SEM*

  Uses posix semaphores (`sem_wait`/`sem_post`). They are slower then the previous methods but still way faster then SYSV sempahores. Unfortunately they also do not work on all the systems (e.g. linux).

- *SYSV_SEM*

  This is the most portable but also the slowest locking method. Another problem is that the number of semaphores that can be alocated by a process is limited. One also has to free them before exiting.

1.2. How to use it ?

First of all you have to include `locking.h`. Then when compiling the code one or all of FAST_LOCK, USE_PTHREAD_MUTEX, USE_PTHREAD_SEM or USE_SYSV_SEM must be defined (the ser `Makefile.defs` takes care of this, you should need to change it only for new architectures or compilers). `locking.h` defines 2 new types: gen_lock_t and lock_set_t.

1.3. Simple Locks

The simple locks are simple mutexes. The type is gen_lock_t.

> **Warning**
>
> Do not make any assumptions on gen_lock_t base type, it does not have to be always an int.

1.3.1. Allocation & Initialization

The locks are allocated with: `gen_lock_t* lock_alloc()` and initialized with `gen_lock_t* lock_init(gen_lock_t* lock)`. Both functions return 0 on failure. The locks must be initialized before use. A proper alloc/init sequence looks like:

```
gen_lock_t* lock;

lock=lock_alloc();
if (lock==0) goto error;
if (lock_init(lock)==0){
   lock_dealloc(lock);
   goto error; /* could not init lock*/
}
...
```

Lock allocation can be skipped in some cases: if the lock is already in shared memory you don't need to allocate it again, you can initialize it directly, but keep in mind that the lock *MUST* be in shared memory.

Example:

```
struct s {
    int foo;
    gen_lock_t lock;
} bar;

bar=shm_malloc(sizeof struct s); /* we allocate it in the shared memory */
if (lock_init(&bar->lock)==0){
 /* error initializing the lock */
 ...
}
```

1.3.2. Destroying & Deallocating the Locks

```
void lock_destroy(gen_lock_t* lock);
void lock_dealloc(gen_lock_t* lock);
```

The `lock_destroy` function must be called first. It removes the resources associated with the lock, but it does not also free the lock shared memory part. Think of sysv **rmid**. Please don't forget to call this function, or you can leave allocated resources in some cases (e.g sysv semaphores). Be carefull to call it in your module destroy function if you use any global module locks.

Example:

```
lock_destroy(lock);
lock_dealloc(lock);
```

Of course you don't need to call `lock_dealloc` if your lock was not allocated with `lock_alloc`l.

1.3.3. Locking & Unlocking

```
void lock_get(gen_lock_t* lock);
void lock_release(gen_lock_t* lock);
```

1.4. Lock Sets

The lock sets are kind of sysv semaphore sets equivalent. The type is lock_set_t. Use them when you need a lot of mutexes. In some cases they waste less system resources than arrays of gen_lock_t (e.g. sys v semaphores).

1.4.1. Allocating & Initializing

```
lock_set_t* lock_set_alloc(int no);
lock_set_t* lock_set_init(lock_set_t* set);
```

Both functions return 0 on failure.

> **Warning**
>
> Expect the allocation function to fail for large numbers. It depends on the locking method used & the system available resources (again the sysv semaphores example).

Example:

```
lock_set_t *lock_set;

lock_set=lock_set_alloc(100);
if (lock_set==0) goto error;
if (lock_set_init(lock_set)==0){
```

```
    lock_set_dealloc(lock_set);
    goto error;
}
```

1.4.2. Destroying & Deallocating

```
void lock_set_destroy(lock_set_t* s);
void lock_set_dealloc(lock_set_t* s);
```

Again don't forget to "destroy" the locks.

1.4.3. Locking & Unlocking

```
void lock_set_get( lock_set_t* s int i );
void lock_set_release( lock_set_t* s int i );
```

Example:

```
lock_set_get(lock_set, 2);
/* do something */
lock_set_release(lock_set, 2);
```