LOCKS

Class Notes

Atomic operations:

Operations that read or change data in a single uninterrupted step are called atomic operations. Common atomic operations are:

- Test and set: Returns the current value of a memory location and replaces it with a given new value.
- Compare and swap (CAS): Compares the contents of a memory location with a given value and if found equal, it replaces with the given new value.
- Load link / store conditional instruction pair (LL/SC)
- Atomic arithmetic operations.

If two or more threads try to access a same shared data in parallel, race conditions will occur. Using atomic counter, software mutual exclusion locking protocols can be designed to protect large amounts of shared data from data races.

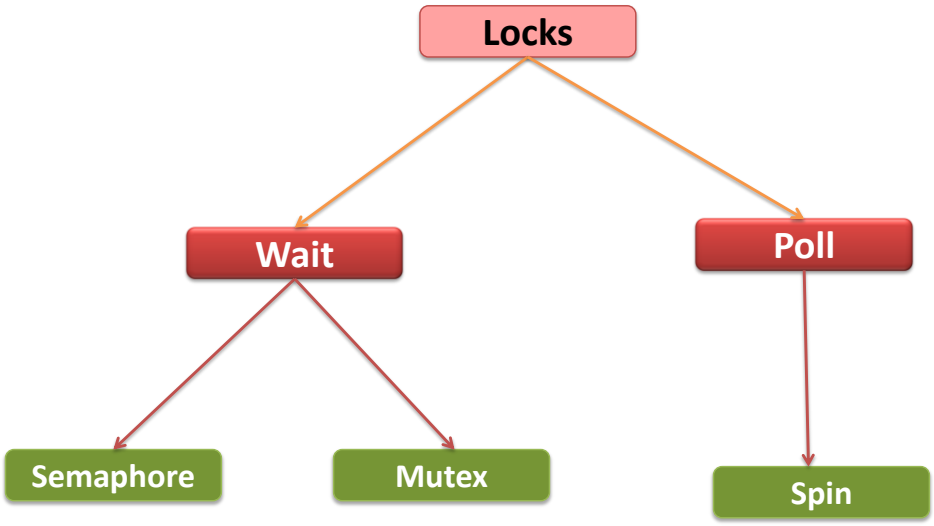
Locks

All thread libraries will provide locking and unlocking functions which can be used by the developers to synchronize threads. Locking and unlocking primitives are implemented using atomic instructions. Mainly there are two kinds of locking primitives provided by the thread libraries.

1) Wait locks
2) Poll locks.

- Wait locking interface push the caller process into wait state when lock acquisition fails. These are recommended when critical sessions are long and undeterministic.

- Poll locking interface put the caller process into a loop until lock is available. These are recommended when the critical sessions are short and deterministic.



Semaphores

Semaphores are shared counters that are incremented or decremented with atomic operations. Semaphores can be used in three different scenarios.

1) Data synchronization (mutual exclusion)
2) Resource counter
3) Wait / Event notification

Posix IPC semaphores

```
#include<semaphore.h>

sem_t *sem_open (char *name , int flag);

sem_t *sem_open (char *name , int flag, mode_t mode,  int value );
```

- **sem_wait** and **sem_post** are used to decrement and increment semaphore counter.
- **sem_wait** blocks the caller if semaphore value is 0, if it is 1 decrements to 0.
- There are three ways to get semaphore

```
int sem_trywait(sem_t*sem);

int sem_wait(sem_t*sem);

int sem_timedwait(sem_t*sem , struct timespec*abs_timeout);
```

- When semaphores are to be applied as a locking counter to enable synchronization of shared data, the following verifications must be supported
    1) Accidental release: lock and unlock operations must not be allowed to execute independently.
    2) Recursive dead lock: thread which has locked semaphore trying again to lock the same semaphore will result in recursive dead lock. This should be verified.
    3) Owner death dead lock: owner state must be trapped and any exceptions on the owner process must be reported to other processes waiting for the same semaphore.
    4) While destroying the semaphore its current state must be verified and must not be allowed to be released if it is in the active state.

Mutex

- Pthreads provide ready to use binary semaphore implementation with all required validations as a frame work called mutex.
- Pthreads mutex framework includes
    1) Mutex object (pthread_mutex_t)
    2) Mutex attribute object (pthread_mutexattr_t)
    3) A set of operations for locking, unlocking, setting and getting mutex attributes.

Sample code:

```
static int glob = 0;
int local;
static pthread_mutex_t mtx = PTHREAD_MUTEX_INITIALIZER;

pthread_mutex_lock(&mtx);
 local = glob;
 local++;
glob = local;
pthread_mutex_unlock(&mtx);
```

- Pthread mutex operation validations for recursive lock, accidental release, and owner death are disabled by default to allow lock and unlock operations to execute faster.
- Developers are advised to enable these validations by using an attribute object attached with mutex.
- Attribute instants can be disabled for production code and can be enabled for test code.

Pthread mutex attributes

Attribute object categorizes validation into following types

- Error check
- Recursive
- Robust
- Consistent

Error check mutex:

- This type enables recursive dead lock validation that returns an error when recursive lock attempt is detected.
- This type also verifies accidental release validations that do not allow a thread to unlock the mutex that is not owned by it.

Sample code:

```
pthread_mutexattr_t attr;
pthread_mutexattr_init(&attr);
pthread_mutexattr_settype(&attr,PTHREAD_MUTEX_ERRORCHECK);
pthread_mutex_init(&mutex,&attr);
```

Recursive Mutex

This type allows a recursive lock to succeed provided it is unlocked same number of times. This type also adds validation for accidental release.

Sample code:

```
 pthread_mutexattr_t attr;
pthread_mutexattr_init(&attr);
pthread_mutexattr_settype(&attr,PTHREAD_MUTEX_RECURSIVE);
pthread_mutex_init(&mutex,&attr);
```

Robust mutex:

This type of mutex adds validation for owner death dead locks. With the attribute set, a call to pthread_mutex_lock shall return EOWNER DEAD in the containing thread.

Sample code:

```
pthread_mutexattr_t attr;
pthread_mutexattr_init(&attr);
pthread_mutexattr_setrobust_np(&attr,PTHREAD_MUTEX_ROBUST_NP);
pthread_mutex_init(&mutex,&attr);
```

Consistent mutex

- When mutex lock operation returns error code EOWNERDEAD, it is responsibility of the current thread to ensure that shared data and mutual lock set back to a valid consistent state.
- To arrays ownership records of the mutex lock and to be able to set it back to unlock state, the recovery code should call,

```
Pthread_mutex _consistent_np(&mutex);
```

Reader /writer locks (rw)

- Standard mutual exclusion protocols are not recommended on shared data that is frequently read and rarely updated.
- For reader intensive applications, rw mutual exclusion lock protocols should be used.
- rw locks allows sharing the lock in between multiple reader threads and mutual exclusion in between writer threads and rw threads.
- Pthreads provides implementation of rw locks using **pthread_rwlock_t**.

Spin locks

- Spinlock interface implements poll mode locking protocols.
- Pthread spinlocks can be used to protect shared data accessed by multiple process or a set of user level threads with in a process.

Sample code :

```
pthread_spinlock_t spin ;

pthread_spin_lock(&spin);
```

Design patterns

- Shared data synchronization using appropriate locking resources does have an impact on the overall performance of the application since all major locking protocols are implemented around mutual exclusion.

The following patterns have been observed across many applications that use various types of locking resources.

1) **Giant locking**: This pattern involves using a single lock counter to protect entire shared data.
2) **Coarse –grained locking**: This method of locking involves large shared data being logically divided into independent modules that can be parallely accessed and protected with local locks.
3) **Fine-grained locking**: This implementation involves identifying smallest possible units of data that can be parallely accessed and protecting them with individual locks.

Thread synchronization

When a group of threads need to achieve a common task by breaking down execution into multiple parallel threads, we need to synchronize their execution with appropriate methods.

Condition variables are one of the thread synchronization primitives provided by pthread library.

Condition variables

- They are used between producer consumer threads for unicast or broadcast event notifications.

Usage:

```
pthread_mutex_t mtx = PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t cond = PTHREAD_COND_INITIALIZER;

/*producer notification*/

pthread_cond_broadcast(&cond);
pthread_cond_signal(&cond);

/*consumer */

pthread_cond_wait(&cond, &mtx);
```

- Above function releases the mutex lock held by the consumer and puts consumer thread into wait until condition is signaled by the producer.
- When condition signal is notified, it acquires the mutex lock and resumes execution of consumer thread.