CS – 519 : OPERATING SYSTEMS - ASSIGNMENT 1

Mandanna Thekkada, Karan Desai
mkt58@rutgers.edu , desai.karan@rutgers.edu

LOCK1

Design

As inferred from the given Bench1 file we can figure out that it is a situation where different worker thread are trying to get into the critical section and this has to be controlled by the locking mechanism. We are implementing this using a Mutex sort of design in which a lock variable which co

Logic

We added a state variable called ***lock_state*** into the ***lock1*** structure. If it is"0" it suggests an unlocked state and if it is "1", it suggests that the lock is currently held by some other Thread and not available.

Lock 1 has been implemented as a simple spin lock. It uses Atomic Functions provided by GCC for the memory access. Note that there is no Scheduling done in particular by our code for the Fairness. NOTE: All the atomic functions that are used are described at the end of this report.

LOCK 2

Design

The given Bench 2 is a client/server based on Read / Write locks. The lock that is designed worked in the following manner

- Write operations are exclusive and cannot be done in parallel with Reads/Writes
- Read operations can be done in parallel with other Reads. Note that there can be a limit on how many read operations can be done together

Again we have used Atomic functions to carry out the execution. We are avoiding the possible Reader/Writer Starvation. Also there is no possibility of a Dirty Read or a Dirty Write in the execution. We have more than one "locks" for this scheme.

Logic

There are four variables we are assigning

```
rcount          //number of readers
sect_lock       //lock to gain entry to that particular section of code
action_lock     //lock to gain to read or write access to resource
rcount_lock     //lock to gain access to rcount ( Number of Readers )
```

The section lock is used to synchronize the a part of code which has a group of Atomic Functions. The Action Lock is the main lock that determines the access. The Reader Count lock is for the rcount variable which allows updates to the number of readers.

BARRIER

Design

"A barrier is a type of synchronization method. A barrier for a group of threads in the source code means any thread must stop at this point and cannot proceed until all other threads reach this barrier "

A thread count is maintained and it is updated when threads call the wait functions. After a certain limit is reached they are allowed to proceed.

Logic

We have a thread counter **thread_count** that is incremented every time the ***barrier-wait*** is called. This is compared with the **tot_threads** (total number) of threads. This comparison decides weather the barrier should let all the threads go or not. When the specific number is reached the barrier let goes of the waiting threads. This thread count has synchronized access and is controlled by the **already designed lock1**.

ATOMIC FUNCTIONS USED

1) __sync_bool_compare_and_swap

Purpose :
This function compares the value of __*compVal* with the value of the variable that __*p* points to. If they are equal, the value of __*exchVal* is stored in the address that is specified by __*p*; otherwise, no operation is performed.
A full memory barrier is created when this function is invoked.
Prototype :
bool __sync_bool_compare_and_swap (*T __*p*, *T* __*compVal*, *T* __*exchVal*, ...);**

2) __sync_lock_test_and_set

Purpose :
This function atomically assigns the value of __*v* to the variable that __*p* points to.
An acquire memory barrier is created when this function is invoked.
Prototype :
***T* __sync_lock_test_and_set (*T** __*p*, *T* __*v*, ...);**

3) __sync_fetch_and_add

Purpose :

This function atomically adds the value of __*v* to the variable that __*p* points to. The result is stored in the address that is specified by __*p*.
A full memory barrier is created when this function is invoked.
Prototype :
***T* __sync_fetch_and_add (*T** __*p*, *T* __*v*, ...);**

4) __sync_fetch_and_sub
Purpose :
This function atomically subtracts the value of __*v* from the variable that __*p* points to. The result is stored in the address that is specified by __*p*.
A full memory barrier is created when this function is invoked.
Prototype :
***T* __sync_fetch_and_sub (*T** __*p*, *T* __*v*, ...);**

Note : T is the suitable data structure

REFERENCES

[1] http://gcc.gnu.org/onlinedocs/gcc-4.1.1/gcc/Atomic-Builtins.html
[2] http://pic.dhe.ibm.com/infocenter/comphelp/v121v141/index.jsp?topic=%2Fcom.ibm.xlc121.aix.doc%2Fcompiler_ref%2Fbif_gcc_atomic_bool_comp_swap.html
[3] http://en.wikipedia.org/wiki/Non-blocking_algorithm
[4] http://en.wikipedia.org/wiki/Compare-and-swap
[5] http://locklessinc.com/articles/barriers/
[6] http://locklessinc.com/articles/locks/
[7] http://www.rfc1149.net/blog/2011/01/07/the-third-readers-writers-problem/