ECE 592: OPERATING SYSTEMS DESIGN

PROJECT 3

REPORT

SAURABH LABDE UNITY ID: sslabde

SHRINATH CHERIYANA UNTIY ID: scheriy

LOCK IMPLEMENTATIONS:

1) Test and Set

We store the address of old value into edx and new value into eax. Now the xchg instruction is used to swap the value of eax and value at address edx atomically. eax now has the old value that is to be returned. The value to be returned always need to be eax register

2) Spinlock

Files Changed - lock.h, spinlock.c, prototypes.h

Structures:- sl_lock_t: It has a field called flag.

Flag – This indicates whether the lock is taken or not.

In spinlock.c, there are three functions defined sl_init(sl_lock_t *l), sl_lock(sl_lock_t *l) and sl_unlock(sl_lock_t *l).

Initialization: - sl_init(sl_lock_t *l)

This function sets the value of flag to be NOT_TAKEN (i.e. 0).

Lock - Checks the value returned by the testandset function. If this testandset returns the value as TAKEN then the process that has called sl_lock (&l) will spin inside the while loop till the lock is released. If the testandset returns NOT_TAKEN, the process that has called sl_lock(&l) will acquire the lock.

Unlock - Sets the value of the flag to be NOT_TAKEN thereby, indicating the lock has been released.

3) Lock limiting busy waiting

Files changed – lock.h, lock.c, process.h, prototypes.h

Structures: - lock_ t: It consists of taken, guard and wait queue.

flag – This indicates whether the lock is taken or not.

Guard – This flag indicates whether the guard is taken or not. The guard should be acquired if a process wishes to take the lock.

Wait queue – This is the queue_id of the queue used to hold all the processes waiting on the given lock.

Initialization: - init_lock (lock_t *l)

This function initializes the guard and the flag values to 0 and stores the queue_id obtained from the newqueue () call.

Helper Functions - park, set_park and unpark(pid)

Park – This function checks if the set to park flag in the proctab of the given process is high. If it is, it calls suspend system call on the given process. This is done to provide protection against preemption.

Set-Park – This function sets the set_park_flag in the proctab of the given process to 1. This is indicative that the process is going to go in suspend state to wait for the lock.

Unpark () – It makes the given pid ready by calling the ready system call on it.

Lock – The process uses the test and set function in order to acquire the guard. Once the guard is acquired, it can set the flag value to be 1 and thus indicate that it has taken the lock provided that the flag value was initially 0. If the flag is 1, it indicates that the lock is already taken. Thus, the process puts itself in the wait_queue of the lock and calls set_park and park respectively to suspend it. The guard to be reset in both cases for the next process.

Unlock – The test and set function is used to acquire the guard. Once the guard is acquired, the lock is released to the process at the head of the wait_queue. The guard value is reset and this process exits the lock function – holding the lock. If no process is currently waiting for the lock, the flag is simply set to 0 so that when the next process requests a lock, it gets it.

4) Active lock with deadlock detection

Files changed: lock.h, active_lock.c, process.h, prototypes.c, create.c, initialize.c , kernel.h

Structures – al_lock_t: It consists of lock_id, flag, guard, wait_queue and lock_holder.

The flag, guard and the wait-queue fields are exactly the same as in the busy wait free lock.

Lock_holder – This field is to indicate that which process is currently holding the lock.

Lock_id – This is an identifier assigned to this lock from a global incrementing counter.

Locktab – This is an array which can be indexed with the lock id and contains the holder for that particular lock.

Number_of_locks – global counter which is used for assigning the lock_id;

Proctab entries – has lock- indicates which lock the processor owns currently.

Waiting_for_lock – Indicates the lock that is blocking the current process. It defaults to 0.

Part_of_deadlock – Indicates if the given process is involved in deadlock.

Initialization – assign the lock id using the pre-incremented global counter and set all the other fields to 0 other than the wait_queue filed which stores the queue_id.

Helper_functions – park and **set_park** is same as in the previous lock. **Unpark** is modified to update the Locktab with the information of the new lock holder.

Deadlock detector – This function is for detecting the deadlock. It identifies cyclic dependencies between the processes. When it is called, it checks the lock on which the current lock holder is waiting. If it is 0, the function returns control back to the calling function. If it is not zero, it identifies the holder of this lock by looking into the Locktab. It then checks if the next lock holder is equal to the current pid and thus detecting cyclic dependencies. This function is written in a recursive fashion and thus computes for N locks detecting cyclic dependencies.

Display_deadlock – Display the processes involved in deadlock

Clear_deadlock_flags – To reset the deadlock flags if no deadlock is detected.

Lock – This function is the same as the one on the busy wait free lock with minor updates. When the lock is acquired by the process, the entries of the Locktab are updated to reflect this. On the other hand, if a process fails to acquire a lock, the deadlock detector function is called to check for deadlock. If it returns 1, deadlock is observed and thus the display deadlock function is called along with clear_deadlock_flags to enable detection of other deadlocks in the system. If it returns 0, simple the deadlock flags are cleared.

Unlock – This function is exactly the same as in the busy wait free lock. It just has one change that that Locktab entries are updated indicating the current process has released the lock. The Locktab entries for the next lock holder are updated in park.

Bool8 trylock () – In this function , the guard is acquired using the test and set instruction. Once the guard is acquired if the flag is not set, the process acquires the lock and sets the values in the Locktab to indicate it and returns TRUE. If the process fails to acquire the lock, it simply returns control to the caller by returning FALSE.

5) Lock with priority inheritance

Files changed – lock.h, pi-lock.c, prototypes.c, process.h, create.c, initialize.c, kernel.h

Structures: - pi_lock_t: it contains lock_id, flag, guard, wait_queue, lock_holder. These are the same as the ones in al_lock_t from the previous lock. The lock id is assigned from a global incrementing counter.

tab_ t: It contains pid and wq_Id. This is the format of each entry in the Locktab_pi structure.

Pid – The pid of the lockHolder.

Wq_id – The wait_queue id associated with a lock.

Locktab_pi [] – This is an array of structures which can be indexed by the lock id and stores the lock holder and the wait queue id of a lock. The size of this array is set to MAX_LOCKS which is 100.

Number_of_locks_pi is a global counter used for assigning lock_id.

Proctab Entries – has_lock – Indicates the lock currently owned by the process/

Waiting_for_lock – Indicates the lock blocking the current process.

Wait_queue_id – the id of the wait_queue in which the process is currently residing.

Initialization - assign the lock id using the pre-incremented global counter and set all the other fields to 0 other than the wait_queue filed which stores the queue_id.

Helper_functions – the **park** and **set park** functions are the same as the ones from the previous locks. The **unpark** function is modified a little. The priority release function is called at the beginning of unpark. In unpark, the Locktab entries for the new lock holder is also updated.

Priority_inheritance_protocol – This function implements the priority inheritance protocol to avoid inversion. It checks if the priority of the process requested the lock is greater than the priority of the locks' current holder. If it is, the lock_holder gets the priority of the current process. Its priority is changed, and it is re inserted in the queue it belonged to. The transitive nature of this property is considered and implemented here. This function checks for any locks which are blocking the current lock holder. If the current process has a greater priority than the blocking lock holder, the priority of

the blocking lock holder is changed to the priority of the current process and this continues till we find that no other process is blocking a lock.

Priority_release – This function is called from unpark. It checks if the original priority of the process is different from its current priority. If yes, it assigns the process the highest priority of the process which it is currently blocking.

Lock - This function is same as the one in busy wait free lock with two changes. When a process acquires a lock, the Locktab_pi entries are updated to reflect it. If a process fails to acquire the lock, the priority inheritance protocol is called to update the priorities if required.

Unlock – This function is same as the one in busy wait free locks. There is a single change that when a process releases the lock, the Locktab is updated to reflect it.

TEST CASES:

1) *main-basic.c*

In this file there are different implementations of summation of a large array of integer numbers. The description of each implementation is given below.

a) uint32 serial_summation(uint32 *array, uint32 n)

This function performs a simple serial summation of the array of integers. A loop iterates from 0 to n(size of array) – 1 and in every iteration the current element of array is added to a global sum variable. After the loop exits the sum which contains the sum of the array of integers is returned.

b) uint32 naive_parallel_summation(uint32 *array, uint32 n, uint32 num_threads)

This function spawns “num_threads” number of processes each of which execute a function called parallel_sum_naive. We maintain a global counter index. In the parallel_sum_naive function this index is checked to be less than “n”, if yes then the current array element is added to a global sum variable. So, each thread whenever scheduled adds elements of array to the global sum variable till the global index variable has reached n-1.

The possibility of getting a correct sum value with this implementation is less as depending on the scheduling there may be a case where the threads add same elements of the array to the sum or a case where certain elements of the array were skipped.

c) uint32 sync_parallel_summation(uint32 *array, uint32 n, uint32 num_threads)

This function spawns “num_threads” number of processes each of which execute a function called parallel_sum_sync. The parallel_sum_sync function is similar to the parallel_sum_naive function described in the previous case with the only difference being that the updating of the global sum variable and incrementing the pointer to the array element is protected by a lock. So, at any given instant only one thread can update the sum variable and increment the pointer to the array.

This implementation will always result in a correct implementation as the shared variables are protected by a lock.

N (array size)	***1000***	***80000***	***900000***	***1000000***	***1100000***
Number of Threads	***15***	***25***	***40***	***50***	***62***
Serial	1000	80000	900000	1000000	1100000
Naive	1000	80000	1196818	1264266	446459
Sync	1000	80000	900000	1000000	1100000

The above shows that for smaller array size all three summation implementations provide correct result but as the array size increases the naive summation gives incorrect results but the sync summation always gives correct results.

2) *main-perf.c*

In this file we have two functions called the uint32 test_spin(uint32 num_threads) and uint32 test_bwf(uint32 num_threads).

The uint32 test_spin(uint32 num_threads) function spawns “num_threads” number of processes each of which execute a function called spin_task(). When in the spin_task() function, a thread will acquire a spin lock and will enter the critical section in which the thread spins for specified number of cycles and then releases the lock.

The uint32 test_bwf(uint32 num_threads) function spawns “num_threads” number of processes each of which execute a function call bwf_task(). When in the bwf_task() function, a thread will acquire a busy wait free(bwf) lock and will enter the critical section(same as the spin_task) in which the thread spins for specified number of cycles and then releases the lock.

To measure the performance of the spin and active locks in the function main_perf we first get the time using ctr1000 and then call the function test_spin and then after this function returns, we again get the time. The difference between these two time values give a rough estimate of the time taken by the implementation which uses spin locks. We repeat the same process of recording time for the test_bwf function.

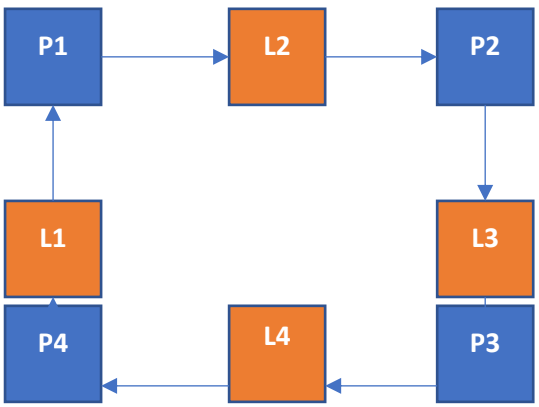
After analysing both the time results, we can see that the implementation which uses bwf lock takes less time as compared to the one which uses spin lock. Thereby concluding that bwf lock performs better than spin locks.

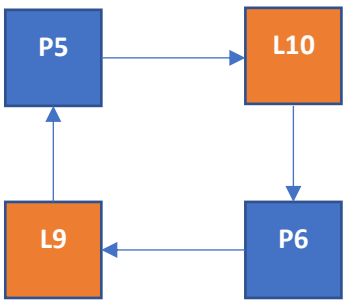
Number of Threads	***15***	***25***	***50***
Execution Time using spin lock (ms)	180	420	1700
Execution Time using busy wait free lock (ms)	30	40	90

3) *main-deadlock.c*

a) Trigger a deadlock situation.

We create two different deadlock situations, one involving 4 processes and the other involving two processes.





In the first case P1 has acquired L1 and is waiting on L2. L2 has been acquired by P2 and is waiting L3. L3 has been acquired by P3 and is waiting on L4 and finally L4 has been acquired by P4 and is waiting on L1.

In the second case P5 has acquired L9 and is waiting on L10 which has been acquired by P6 and P6 is waiting on L9.

In the file main-deadlock.c, the functions test_deadlock_1(), test_deadlock_2(), test_deadlock_3(), test_deadlock_4(), test_deadlock_5(), test_deadlock_6() are the functions executed by the processes (shown in the diagram) P1, P2, P3, P4, P5, P6 respectively. Eg. In test_deadlocK_1(), the process will acquire lock 1 using al_lock() and then try to acquire lock 2 using al_lock(). Once it has acquired lock 2 it will then first release lock 2 and then release lock 1. The other test_deadlock_* functions do the same thing only with different locks.

When the test case is run, the program outputs deadlock detected P1-P2-P3-P4 and deadlock detected P5-P6

b) Avoid deadlock by using trylock

In this part the same situation in the first case (i.e involving 4 processes) of the (a) part is created but with a modification such that now the process uses trylock instead of al_lock() to acquire the lock. So, four processes P7, P8, P9, P10 implement the function test_trylock_1(), test_trylock_2(), test_trylock_3(), test_trylock_4() respectively. Eg. In test_trylock_1(), process will try to acquire lock 5 using trylock i.e it will wait till the trylock(&l5) returns TRUE. In this case it has acquired lock and now it checks if trylock(&l6) is TRUE. If this is FALSE then it will release lock 5 and then again start by calling trylock on lock 5 and if trylock(&l6) is TRUE then it will first release lock 6 and then lock 5. The other test_trylock_* functions do the same thing only with different locks.

When this test case is run, the program outputs nothing as there is no deadlock to be detected.

4) *main-pi.c*

We a total of 6 test cases in this file that show priority inversion, priority inheritance, transitive priority inheritance, the problem blocking chain formation and also a testcase which compares the performance of busy wait free lock and pi lock.

(i) Test case showing priority inversion:

In this we create a low priority process P1 which acquires a busy wait free lock. After this a high priority process P2 is created which tries to acquire the same busy wait free lock acquired by P1. Also, an unbounded medium priority process P3 is created which does not use any locks.

The result of the test case is that P1 runs first followed by P3 and then P2. Thus, highest priority task was run last and thus, there is priority inversion.

(ii) Test case showing priority inheritance:

In this we create a low priority process P1 which acquires a lock which prevents priority inversion and then later a high priority process P2 is created which tries to acquire the same lock which prevents priority inversion.

The result of this test case is that P1 runs first and its priority has been changed to that of P2 and then P2 runs after that. This shows that P1 inherited P2's priority.

(iii) Test case showing that priority inheritance is transitive:

Locks used in this case are locks which prevent priority inversion.

We first create a low priority process P1 which acquires a lock l1. Then a medium priority process P2 is created which acquire lock l2 and tries to acquire l1. Then a high priority process P3 is created which tries to acquire lock l2. Now, we see that P3 is blocked on P2 and P2 is blocked on P1. Now in this case P1 should inherit the priority of P3 via P2.

The result of this testcase is that first we see that since P1 has acquired lock l1 and P2 is waiting on lock l1 P1 will inherit the priority of P2 and then since P2 has already acquired lock l2 and P3 is waiting on lock l2 P2 will inherit the priority of P3. Now since, P2 is still waiting on lock l1, P1 will get P3's priority via P2 and it runs first with priority of P3 followed P2 who also runs with priority of P3 finally followed by P3. This shows that priority inheritance is transitive.

(iv) Test case showing blocking chain formation with priority inversion:

We first create a low priority process P1 which acquires a busy wait free lock b1 and another process P2 with the same priority which acquires another busy wait free lock b2. Now a high priority process P3 is created tries to acquire both b1 and b2.

The result of this test case is the P3 is blocked for the duration of two critical sections, thus forming a blocking chain.

(v) Test case showing blocking chain formation with priority inheritance:

Locks used in this case are locks which prevent priority inversion.

We first create a low priority process P1 which acquires a lock p1 and another process P2 with the medium priority which acquires another busy wait free lock p2. Now a high priority process P3 is created tries to acquire both p1 and p2.

The result of this test case is the P3 is blocked for the duration of two critical sections, thus forming a blocking chain and also shows that the effectiveness of priority inheritance is limited by the formation of chain of blocking.

(vi) Test case showing that locks that prevent priority inversion performs better than busy wait free locks.

In this we create a low priority process P1 which acquires a lock which prevents priority inversion. After this a high priority process P2 is created which tries to acquire the same acquired by P1. Also, an unbounded medium priority process P3 is created which does not use any locks.

Then we create a low priority process P4 which acquires a busy wait free lock. After this a high priority process P5 is created which tries to acquire the same acquired by P1. Also, an unbounded medium priority process P6 is created which does not use any locks.

The result of this test case is that we see the execution time of high priority task that uses a lock that prevents priority inversion is less than the execution time of high priority task that uses a busy wait free lock. Thus, priority inheritance reduces the blocking time of the high priority task that uses locks that prevent priority inversion.