Report for Homework #3
Submitted by:
Pankaj Bongale (pbongal@ncsu.edu)
Advait Javadekar (ajavade@ncsu.edu)

1. test_and_set.S
Implementation: First we push the ebp onto stack and then record current SP into ebp. Next we disable interrupts using cli instruction. This is needed since the question requires us to implement test_and_set as an atomic function. This is followed by pushing the flags and registers on the stack.
Now we read *ptr into ebx and read new_value into eax. xchg instruction is used to exchange values. Now eax has the new value that is to be returned. This is temporarily pushed on stack before restoring the registers and then written back into eax.
Finally ebp is restored and the interrupts are enabled before returning

2. Spinlock
struct: The struct has only a flag called taken.
sl_init: sets taken to zero.
sl_lock: waits in while for test_and_set to return 1.
sl_unlock: sets taken to zero.

3. BWF lock
<u>struct:</u>
taken flag, guard flag, and a thread wait list named twlist
<u>bwf_init:</u>
taken and guard are set to zero. twlist is initialized as newqueue.
<u>bwf_lock:</u>
First guard is acquired. Then if taken is zero, then taken is made one and guard is reset.
Else the current process is entered into twlist and put to wait. Then yield is called to reschedule. Upon return from other processes, the current process acquires the lock.
<u>bwf_unlock:</u>
First guard is acquired. Then taken is set to zero. Next if twlist has processes waiting on the lock, then the first process is dequeued and resumed.

4. Activelock
<u>proctable new entries:</u>
locks array – to store all locks acquired by a process, index – count of all the locks acquired so far, waitlock – lock on which process is waiting
<u>struct:</u>
taken flag, guard flag, and a thread wait list named twlist
<u>al_init:</u>
taken and guard are set to zero. twlist is initialized as newqueue.
<u>al_lock:</u>
First guard is acquired. Then if taken is zero, then taken is set and proctab entry of locks array is updated.

If taken is one, then waitlock entry of proctab is updated to the current lock. Next is deadlock detection. All processes and their lock arrays are looped through. If any process is seen to hold the current lock, then that process is checked if waiting on some other lock. If yes, then the looping is continued till the process holding the lock is not waiting on any lock. But during this loop, if a process appears twice, then that is the deadlock case.
Finally the current process is added to twlist and changed to WAIT, followed by a yield(). When the lock is finally released by other processes and this current process is resumed, it marks the taken flag as 1 and updates the locks array for the current process.
al_unlock:
First guard is acquired. Then taken is set to zero and locks array is decremented. Next if twlist has processes waiting on the lock, then the first process is dequeued and resumed.
al_trylock:
First guard is acquired. Next if taken is zero, then taken is made 1 and locks array of the proctab is updated with the current lock. Then the function returns 1.
If the taken is already 1, then the function returns 0.

5. Priority Inheritance:
proctable new entries:
locks array – to store all locks acquired by a process, index – count of all the locks acquired so far, orig_prio – to save the original priority of the process, prio_for_locks array to store the priority of the process with highest priority currently waiting on the lock.
waitlock – lock on which process is waiting
struct:
pi_lock flag, pi_guard flag, and a thread wait list named pi_blist
pi_init:
pi_lock and pi_guard are set to zero. pi_blist is initialized as newqueue.
pi_lock:
First pi_guard is acquired. Then if pi_lock is zero, then pi_lock is set and proctab entry of locks array and prio_for_locks array is updated.
If pi_lock is one, then waitlock entry of proctab is updated to the current lock. Next, we try and see if any transitivity case needs to be taken care of i.e. process 1 waiting on lock 1 but lock 1 held by process 2 which is waiting on lock 3 held by process 3; then if process 1 priority is higher than priority of process 3, process 3 should by transitivity acquire the priority of process 1. All processes and their lock arrays are looped through and priorities of processes currently holding the locks are updated accordingly.
If there is a process that is already waiting on some lock and we find a dependency of another process waiting on this lock then the priority of this waiting process is updated if it is lesser and it is re-inserted on the waiting queue.
Finally the current process is added to pi_blist according to its priority and changed to WAIT, followed by a yield(). When the lock is finally released by other processes and this current process is resumed, it marks the pi_lock flag as 1 and updates the locks array for the current process.
pi_unlock:
First guard is acquired. Then pi_lock is set to zero and locks array is decremented. Next if this process is waiting on any other lock it’s priority is set according to the max priority of waiting for that lock from the prio_for_locks array OR if it isn’t waiting on any locks then it’s priority is set to it’s original priority orig_prio. If pi_blist is empty then yield(). If pi_blist has processes waiting on the lock, then the first process is dequeued and resumed.

TEST CASES:

#1 main-basic.c

serial_summation:

This function serially adds all array elements within the same process.

naïve_parallel_summation:

This function calls multiple threads called “add_nolock”. Each thread adds a subset of the array. But the threads update the global sum with every new addition without any locks.

sync_parallel_summation:

This function calls multiple threads called “add_lock”. Each such thread adds a subset of the array. In this case the threads update the global sum with each new addition only inside the locks.

Results:

This test was run for different thread numbers and different array sizes. Here are the tabulated results:

Array size	300000000	400000000	500000000
num_threads	30	40	50
serial_summation	300000000	400000000	500000000
naïve_parallel_summation	269594570	379944724	483799676
sync_parallel_summation	300000000	400000000	500000000

#2 main-perf.c

sync_parallel_summation:

This function calls multiple threads called “add_lock”. Each such thread adds a subset of the array. In this case the threads update the global sum with each new addition only inside the locks. These threads use a queue based bwf_lock.

sl_parallel_summation:

This function calls multiple threads called “add_sl_lock”. Each such thread adds a subset of the array. In this case the threads update the global sum with each new addition only inside the locks. These threads use spinlock based sl_lock.

Results:

Array size	200000000	300000000
num_threads	20	30
Execution time for Queue based lock parallel summation	5315	9058
Execution time for spinlock parallel summation	13742	32804

We can clearly see that the queue based lock bwf works faster than spinlock in both cases.
More the number of threads more the busy waiting and so execution time of spinlock increases disproportionately with respect to threads.

#3 main-deadlock.c:
There are 3 threads. The task of each thread is to add the input variable to the global sum variable.
But before the threads do additions, they try to acquire 3 locks:
Step 1: thread 1 acquired lock l1, then thread 2 acquired lock l2, then thread3 acquired lock l3.
Step 2: thread 1 tries to acquire l2, then thread 2 tries to acquire l3, then thread 3 tries to acquire l1.
Step 3: thread 1 tries to acquire l3, then thread 2 tries to acquire l1, then thread 3 tries to acquire l2.
Step 4: each thread leaves all the threads acquired.

In the first case al_lock is used to acquire locks. This results in deadlock in step 2.
In the second case al_trylock is used. This prevents any deadlocks and gives the correct results.

#4 main-pi.c

Case 1: Priority Inversion works effectively.
3 threads p1,p2,p3 add elements of an array to sum variable. They use Priority Inheritance based pi_lock pl.
Priorities p1->2, p2->4, p3->6.
They start running in order p1 then p2 then p3.
Due to priority inheritance p3 runs after p1 and p2 runs last.
Thus, higher priority process could acquire lock faster because of priority inheritance, increasing the efficiency of the system.

3 threads p4,p5,p6 add elements of an array to sum variable. They use bwf_lock l.
Priorities p4->2, p5->4, p6->6.
They start running in order p4 then p5 then p6.
After p4 releases lock, p5 gets dequeued from the blocked queue first and it runs acquires lock and runs before higher priority process p6.
Thus, higher priority process p6 was blocked by lower priority process p5 decreasing the efficiency of the system.

Results:

Process ID	Execution Time
P1	717
P2	2138
P3	***1436***
P4	709
P5	1412
P6	***2114***

Case 2: Chain of Blocking reduces efficiency of Priority inheritance:
3 threads p1,p2,p3 add elements of an array to sum variable. They use Priority Inheritance based pi_lock pl and pl2.
Priorities p1->2, p2->4, p3->6.
They start running in order p1 then p2 then p3.

p1 acquires lock pl starts running.
p2 acquires lock pl2 preempts p1 and starts running.
p3 tries to acquire locks pl and pl2 sequentially and gets blocked by lower priority processes p1 and p2 and thus has to wait for both to complete critical sections before finally executing. Therefore it's efficiency os decreased and it works similarly to the basic queue based bwf lock.

3 threads p4,p5,p6 add elements of an array to sum variable. They use bwf_lock l and l2.
Priorities p4->2, p5->4, p6->6.
They start running in order p4 then p5 then p6.
p4 acquires lock pl starts running.
p5 acquires lock pl2 preempts p4 and starts running.
p6 tries to acquire locks pl and pl2 sequentially and gets blocked by lower priority processes p4 and p5 and thus has to wait for both to complete critical sections before finally executing.

Results:

Process ID	Execution Time
P1	1426
P2	715
P3	***2136***
P4	1410
P5	704
P6	***2121***

----------------------------------EXTRA TEST CASE FOR TRANSITIVITY BEHAVIOR----------------------------------

 # main-pi-behav.c

This test case clearly shows the transitivity (behavior) of priority due to interdependency of locks.

4 processes P1->2, P2->4, P3->6, P4->8.
2 pi_locks l1 and l2.

BEGIN:
P1 starts acquires lock l1.
P2 starts acquires lock l2 then tries to acquire lock l1, gets blocked and P1 inherits priority of P2 i.e. 4.
P1 runs with priority 4.
P3 starts does not need any locks it runs and ends.
P4 starts tries to acquire lock l2, this increases priority of P2 to 8 and as P2 is waiting on lock l1 by transitivity this increases priority of P1 to 8.
P1 runs with priority 8.
P1 releases lock l1 and exits.
P2 acquires lock l1 and starts executing with priority 8.
P2 releases lock l1.
P2 releases lock l2.
P4 starts acquires lock l2 and starts executing with priority 8.
P4 releases lock l2 and exits.
P2 starts running with it's original priority 4.
P2 exits.