**ECE 592 Homework 3 Report**

Name : **Karthick Narayanan Srinivasa Raghavan** Unity ID : **ksriniv**

Name : **Venkata Surya Teja Penumatcha**  Unity ID : **vpenuma**

**P1:**

**Implementation Details:**

The actual test and set implementation was achieved by first declaring the header in prototypes.h. Since it takes an unsigned pointer address and value, when testandset.S is called from a function,

1. Pushes EBP onto the stack
2. Moves the value of stack pointer to the ebp value
3. Move the first argument into the edx register
4. Moves the second argument into the eax register
5. Atomically exchanges the value using “xchngl”
6. Restores the stack pointer and returns the old value.

**Files Changed:**

testandset.S

prototypes.h

**P2:**

**Implementation Details:**

The lock structure s1\_lock\_t was defined with variable ‘flag’ inside the structure which denotes if the lock is taken or not.

Init Function :

Takes the input structure’s address and sets the flag to ‘zero’.

Lock Function:

Takes the input structure’s address and run’s testandset on it. If the lock is already taken, it waits till the lock is unlocked. If the lock is not taken, then takes the lock and returns back.

Unlock Function:

Takes the input structure’s address and resets the flag to zero.

**Files Changed:**

spinlock.c

prototypes.h

lock.h

**P3:**

**Implementation Details:**

The lock structure bwf\_lock\_t was defined with the variable flag inside the structure denotes if the lock is taken or not and variable wait which is a queue for processes waiting on the lock.

Init Function :

Takes the input structure’s address and sets the flag to ‘zero’. Initializes the queue wait for the structure.

Lock Function:

Takes the input structure’s address and run’s testandset on it. If the lock is already taken, it adds the current process to the wait queue of the structure. It also changes the process state to ‘PR\_WAIT ‘ and calls resched() to change the current state of the process. If the lock is not taken, it takes the lock and returns back.

Unlock Function:

Takes the input structure’s address and resets the flag to zero. If the queue is not empty, it dequeues the first element on the wait queue of the structure and puts into ready state.

**Files Changed:**

lock.c

prototypes.h

lock.h

queue.h

**P4:**

**Implementation Details:**

* Extended from P3
* Lock structure keeps track of which process currently holds the lock
* Every process keeps track of which process is blocking it. In order to do this, we save the pid of process that has the lock as an entry in the PCB of the process trying to acquire the lock. This should also be updated for every process in the lock queue, whenever a new process acquires the lock.
* After placing a process in lock queue, check for a loop in lock dependencies. If there is loop then deadlock exists.

**Files Changed:**

active\_lock.c

prototypes.h

lock.h

**P5:**

**Implementation Details:**

* Extended from P3
* Lock structure keeps track of which process currently holds the lock
* Every process keeps track of which process is blocking it. In order to do this, we save the pid of process that has the lock as an entry in the PCB of the process trying to acquire the lock. This should also be updated for every process in the lock queue, whenever a new process acquires the lock.
* When the priority of the current process requesting the lock is higher, we check which process is blocking the current process and update all their priorities.
* During unlock, we update the process priority to old priority (old priority is created at the beginning of process creation, old priority is always the initial assigned priority at process creation)

**Files Changed:**

pi\_lock.c

prototypes.h

lock.h

**P6:**

**Main-basic.c:**

Create an array of size 100, but used array(index%100) for indexing to simulate a large array.

Incremented a global variable sum in a for loop. In naive case without using locks when changing the sum variable. This method should result in an incorrect final sum. As predicted, the final sum value was incorrect.

In the sync case, we utilized locks to lock and unlock when changing the sum variable. This resulted in correct sum at the end.

**Main-perf.c:**

Used the functionality from main-basic.c but added starttime and endtime before and after the function call to measure performance in milliseconds.

**Main-deadlock.c:**

**Following case is simulated:** Each process acquires locks in the order displayed below.

Process 1 acquires lock 1

Process 2 acquires lock 2

Process 3 acquires lock 3

Process 1 waits for lock 2

Process 2 waits for lock 3

Process 3 waits for lock 1

Deadlock detected

The end result was as predicted.

**Main-trylock.c:**

Process 1 acquires lock 1

Process 2 acquires lock 2

Process 2 tries lock 1 and fails

Process 2 releases lock 2

Added some delay to process 2 to avoid livelock

Process 1 acquires lock 1

Process 1 tries and acquires lock 2

Process 1 releases lock 2 and 1

Process 2 acquires lock 2

Process 2 acquires lock 1

Process 2 releases lock 1 and 2

The end result was as predicted.

**Main-pi.c:**

**Following are the simulated steps:**

Process J1 created, priority 10

J1 acquires lock 1

Process j2 created, priority 12

J2 acquires lock 2

J2 waits for lock 1 and is blocked by J1

J1 inherits priority of J2

Process J3 created, priority 14

Process J4 created, priority 13

J3 waits for lock 2 and is blocked by J2 (J2 is blocked by J1), so blocked by J1

J2 inherits priority of J3 and J1 inherits new J2 priority

J1 releases lock 1

J1 priority back to priority 10

J2 acquires lock 1

J2 releases lock 1 and lock 2

J2 back to priority 12

J3 acquires lock 2

J3 release lock 2

J3 completes

J4 completes

J2 completes

J1 completes

The outcome was as expected