Report for CS344 Assignment-4

By:- Akshat Arun
190101007

Video Link: - https://youtu.be/svK4l9ClQvc

Synchronisation and Locking Primitives in xv6:-

No thread libraries are implemented in xv6, so no 2 user programs can access the same user space memory image so no need of locks are there and hence the user space lacks locking primitives. However there is scope of concurrency in the kernel space processes. Two processes in kernel mode in different CPU cores can access the same kernel data structures.

Spinlocks: So for the protection of shared data in kernel space the xv6 os has spinlocks implemented in it. It is a binary lock and allows only one process at a time to access the shared data while the other process' who requested for the lock are trapped in a busy waiting while loop, till the process who held the lock releases it.

Sleeplocks: Xv6 also has another lock implemented known as the sleep lock which is also binary in nature. But this lock does not disable the interrupt handler like the spinlock and is not widely used in xv6 and was implemented only 6 years back in xv6 to avoid busy waiting of the processes.

Missing features in xv6 which I plan to implement are:-

1) Counting sleep semaphores in the kernel space
2) Sleep mutex locks in the kernel space

Feature 1:-

Semaphores are yet another mechanism for support of thread or process synchronization. Linux already has spinlocks but spinlocks are generally used when the locks are required for a shorter duration of time. For longer duration Linux uses counting sleep semaphore which makes a process sleep on a channel if it fails to acquire the lock. And counting semaphore is an advantage, because if we have N resources to be shared among the processes then we can initialise the value of the semaphore to be N.

Implementation guidelines

To implement counting sleep semaphores in xv6, we start by creating 2 files semaphore.h and semaphore.c.

Semaphore.h:-

1) This file contains the definition of the semaphore struct, we will include the necessary header files like the param.h and spinlock.h.
2) The struct definition would contain an integer val which will store the current value of the semaphore.
3) It will contain a spinlock “lock”(we included spinlock.h for this) which will be used to lock the semaphore.
4) Now we would like to maintain a circular queue of sleeping channels on which the processes who failed to acquire the lock would sleep, so for that we would have an array of void* channels with size as NPROC(we included param.h for this) and also we will have 2 pointers next and end which will basically act as the head and tail of the circular queue.
5) And also we will be having spinlocks for all of the NPROC channels.

Semaphore.c:-

1) Here we will mainly have 4 methods:
    a) Sem_init - It will help us in initialising our newly declared semaphore.
    b) Sem_wait - It will make a process wait if the current value of the counting semaphore is equal to zero else it will let the process access the critical section/resource which it requested and decrement the value of the semaphore.
    c) Sem_signal - It will increase the semaphore value by one and will also wakeup the process sleeping at the head pointer of our circular queue so that it can then access the critical section.
    d) Sem_broadcast - It is a method created in order to wakeup all of the processes sleeping inside the circular queue while also signalling the semaphore at the same time. Basically broadcast to all the sleeping processes that the semaphore is now available.
2) We will see how we will be implementing each of these methods:-
   Sem_init:-
    a) It will be a void method which will have 2 arguments: the first one will be the pointer to our semaphore struct and the second parameter would be the value with which we want to initialise our semaphore.
    b) Inside the function we will set the value of our semaphore equal to the parameter’s value.

    c) Then we also need to initialise the spinlock which we are going to use. Hence we will be calling the initlock() with our semaphore's lock as the argument.
    d) Then we will iterate over the entire circular queue of channels and will set the value of each element as a null pointer and also initialise each channel's lock.
    e) And finally we will also be initialising the head and tail pointers of the queue to 0.
3) Sem_wait:-
    a) It will be a void function with a pointer to the semaphore as the only parameter.
    b) We will first acquire the semaphore lock.
    c) Then we will loop in a while loop till the value of our semaphore is 0. Because if it's zero then the current process cannot access the critical section.
    d) Inside the loop we will push the current running process in the circular queue's tail's channel and will then increment the circular queue's tail pointer by one with modulo NPROC.
    e) And as we have pushed it in the queue then we also have to make it sleep, hence we call the sleep method with the current process' channel and its lock as the argument.
    f) And outside this while loop we decrement the value of our semaphore because one more process is now accessing the resource.
    g) And finally we release the semaphore's lock.
4) Sem_signal:-
    a) It will be a void function with a pointer to the semaphore as the only parameter.
    b) We will first acquire the semaphore lock.
    c) We will firstly increment the semaphore's value because the current process is leaving the resource/critical section.
    d) And then we will have an if block which will be triggered only if the circular queue is non empty. And there we will wakeup the process sleeping at the head, change the value at the head so that now it points to null and finally increment the head pointer by one modulo NPROC.
    e) And finally we release the semaphore's lock.
5) Sem_broadcast:-
    a) It will be a void function with a pointer to the semaphore as the only parameter.
    b) We will first acquire the semaphore lock.

c) As it is similar to sem_signal, we firstly increment the semaphore's value by one.
d) And then we update the head and tail to point to 0 again.
e) Then we iterate over all of the elements of the circular queue and if it is not null then we wake up that process and change the element of the queue so that now it points to null and then we exit the for loop.
f) And finally we release the semaphore's lock.

Feature 2:-

The mutex concept is very similar to a semaphore concept. But it has some differences. The main difference between semaphore and mutex synchronization primitive is that mutex has more strict semantics. Unlike a semaphore, only one process may hold a mutex lock at one time and only the owner of a mutex may release or unlock it.
In spinlocks of xv6 while releasing the lock its only checked whether the current cpu is holding the lock or not hence if suppose P1 was running on cpu1 and has acquired a lock "L1" and then a context switch happens and then P2 starts executing on cpu1 and someone didn't write the code of P2 properly and there P2 is releasing the lock without acquiring it. Then spinlock's implementation would allow that because P2 is running on a processor which has held the lock that is cpu1 and hence would free the lock and then this would be wrong. But in mutexes we have the criteria of ownerships, i.e. a process who has held the lock can only release the lock.

Implementation guidelines

To implement sleep mutex locks in xv6, we start by creating 2 files mutex.h and mutex.c.
Mutex.h:-

1) This file contains the definition of the mutex struct, we will include the necessary header files like the param.h and spinlock.h.
2) The struct definition would firstly contain a boolean variable "locked", which will be true if the lock is acquired or false otherwise.
3) The second element of the struct would be a spinlock "lock"(for this we included the spinlock.h) which would act as the lock.
4) Because we are implementing the sleep version of mutex, hence we will require channels where the processes who failed to acquire the lock would sleep. So here also we would have an array of channels of size NPROC(for this we loaded params.h) which would act as a circular queue.
5) Two integers head and tail for the circular queue.

6) And also we will be having spinlocks for all of the NPROC channels.
7) And finally a pointer to the process who held the lock “owner”.

Mutex.c:-

1) Here we will be having 3 methods:-
    a) Mutex_init - This will be used to initialise the mutex lock.
    b) Mutex_lock - This will be used to acquire the mutex lock by the current process.
    c) Mutex_unlock - This will be used to release the mutex lock by the current process.
2) We will now see how we will be implementing these functions:-
   Mutex_init:-
    a) It will be a void method which will take a single parameter which will be the pointer to our struct mutex lock.
    b) We will initialise “locked” with false as initially the lock is not acquired.
    c) We will now initialise the actual spinlock “lock” by calling the initlock() method with our mutex lock as an argument.
    d) We will iterate and make all the channels point to null and also initialise their locks.
    e) We will set the head and tail pointers to zero.
    f) We will set the owner to point to null initially as no process has held the lock.
3) Mutex_lock:-
    a) It will be a void method which will take a single parameter which will be a pointer to the mutex lock.
    b) We will first acquire the spinlock.
    c) Then we will loop in a while loop till the boolean “locked” variable is true. Because if it's true then the current process cannot access the critical section.
    d) Inside the loop we will push the current running process in the circular queue’s tail’s channel and will then increment the circular queue’s tail pointer by one with modulo NPROC.
    e) And as we have pushed it in the queue then we also have to make it sleep, hence we call the sleep method with the current process’ channel and its lock as the argument.
    f) Now once we are out of the while loop then this means that we can acquire the lock, hence we will change the owner to point to the current process.
    g) And also we will set the locked variable to true.
    h) And finally we will release the spinlock.

4) Mutex_unlock:-
    a) It will be a void method which will take a single parameter which will be a pointer to the mutex lock.
    b) We will first acquire the spinlock.
    c) Now we will first check in an if block whether the current running process is the owner of the mutex lock.
    d) If it isn't then we will call panic() with the message: “Some other process is trying to release a mutex lock”.
    e) Else we will change “locked” to false.
    f) And then we will have an if block which will be triggered only if the circular queue is non empty. And there we will wake up the process sleeping at the head, change the value at the head so that now it points to null and finally increment the head pointer by one modulo NPROC.
    g) And finally we release the spinlock.