

XV6 Synchronization implementation

1. Condition variable.

For the implementation of condition variables we just make a simple struct with a name field in it and whenever someone has to wait on it just set the channel for that process to this struct.

a. Cond_wait

This function takes a lock and a condition variable as input and put the calling process to sleep with the condition variable as the channel and also releases the lock.

Once the process wakes up it regains the lock.

b. Cond_signal

It takes a condition variable as input and runs a for loop on the set of all processes and if it finds any process sleeping on the given condition variable it wakes it up and breaks from the loop.

c. Cond_broadcast

It runs the same as Cond_signal the only difference is that it does not break out from the loop and run over the whole set of processes.

2. Semaphores

For semaphores we have kept a sleeplock and a condition variable which is what we will sleep on and the sleeplock is used to update the value of x, which indicates whether someone is sleeping on it or not.

a. sem_wait

We just hold the lock and check the value of x if it is 0, that means we have to go to sleep so we use the condition variable to go to sleep and release the lock.

b. sem_post

It holds the lock and increases the x by one and then signal the condition variable to wake up.

c. sem_init

It sets the value of x in the semaphore and then initializes the sleep lock.

3. Condition Variable consumer producer

a. cond_buff_init

It initializes all the buffers with there full field set to 0, and initializing there sleeplocks and also it sets the head and tail to 0.

b. cond_produce

We first hold a lock to get a index where the producer will produce the value, and then we just hold the lock of the buffer and wait (by using cond_wait which will also release the buffer lock) until the buffers get empty once it is empty we put the value in it, signal if someone is waiting that the value has been put and then finally release the bufferlock.

c. cond_consume

Same as in case of cond_produce we first hold the lock and get the index from where we will consume the value. Then we will hold the buffer lock and wait for the buffer to get filled if not already and then just take out the value and make a signal that we have consumed the value and release the bufferlock.

4. Semaphore consumer producer

(We have implemented code which is given on page 63 as mentioned in the assignment although this will be slow due to lack of concurrency among the producers and consumers themselves.)

a. sem_buff_init

Initializes the producer lock (used by producers so that at 1 place only 1 producer produces), consumer lock, empty lock and full lock as in the slides.

b. sem_produce

Holds the 'empty' semaphore but if there is no place empty in the buffer then it goes to sleep. And then holds the producer semaphore to produce x in the buffer and then posts the producer and also the full lock.

c. sem_consume

Holds the 'full' semaphore but if no place is full in the buffer then it goes to sleep. and then hold the consumer semaphore to consume x in the buffer and then post the consumer and also the empty lock.

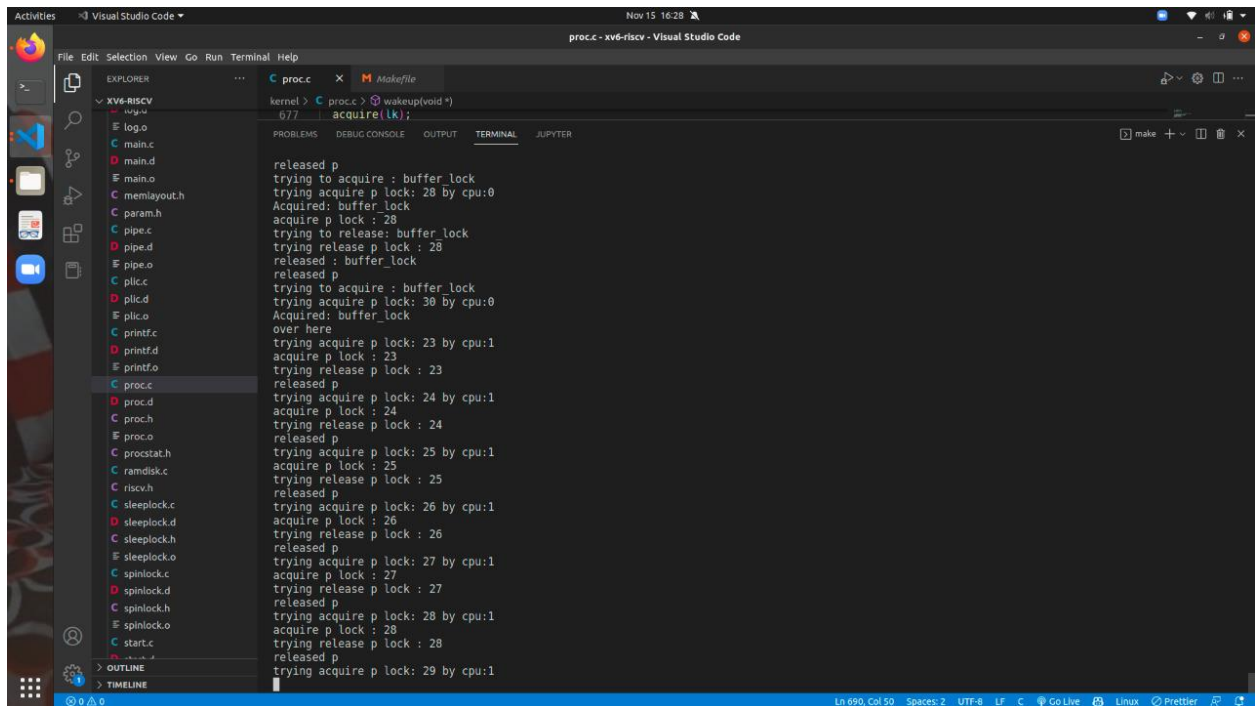
Speed Comparison for Semaphore and Condition variable

1. For small test cases both of them take more or less the same time.
2. In case of large test cases clearly the condition variable is faster due to their different algorithm implemented.

Number of item per producer	Number of producer	Number of consumer	Time taken by cond	Time Taken by semaphores
20	3	2	0	0
80	4	7	0	2
160	4	7	2	4

Notes:

1. Even after increasing the timer interval value the code is getting stuck due to **a problem in acquiring the process lock during wakeup function (please see the attached screenshot)**. Due to approaching exams, we couldn't try any further, although we would have liked to. Also we found that **tickslock also has this problem of getting stuck**.



```
kernel > C: proc.a > wakeup(void *)
077: acquire(lk);

released p
trying to acquire : buffer_lock
trying acquire p lock: 28 by cpu:0
Acquired: buffer_lock
acquire p lock : 28
trying to release: buffer_lock
trying release p lock : 28
released : buffer_lock
released p
trying to acquire : buffer_lock
trying acquire p lock: 30 by cpu:0
Acquired: buffer_lock
over here
trying acquire p lock: 23 by cpu:1
acquire p lock : 23
trying release p lock : 23
released p
trying acquire p lock: 24 by cpu:1
acquire p lock : 24
trying release p lock : 24
released p
trying acquire p lock: 25 by cpu:1
acquire p lock : 25
trying release p lock : 25
released p
trying acquire p lock: 26 by cpu:1
acquire p lock : 26
trying release p lock : 26
released p
trying acquire p lock: 27 by cpu:1
acquire p lock : 27
trying release p lock : 27
released p
trying acquire p lock: 28 by cpu:1
acquire p lock : 28
trying release p lock : 28
released p
trying acquire p lock: 29 by cpu:1
```

2. We tried to trace the problem. So we found out that due to multiple cpus, sometimes one of the cpus gets stuck while trying to acquire the spinlock of a process. But the code runs fine for a small number of forks like for 8-10 total number of forks for condprodconstest and also semaphore one.