CS330A: Assignment 3

Jaya Meena: 200472
Kajal Deep: 200483
Kumar Arpit: 200532
Pratham Jain: 200712

- **Condition Variable:**
  - We declare the structure of condition variable in kernel/condvar.h. It only has a dummy integer variable for our implementation. We typedef it to cond_t.
  - We implement cond_wait(), cond_signal(), and cond_broadcast() which by themselves are simple, one liner functions to condsleep(), wakeupone(), and wakeup() respectively.
  - We implement wakeupone() almost identical to wakeup() except that if it finds any process sleeping on chan, it breaks out and returns.
  - Similarly, implemented condsleep() same as sleep() except that since it takes sleeplock instead of a spinlock as its argument, we use acquiresleep and releasesleep instead for its lock.
- **Barrier array:**
  - We declare struct barrier in barrier.h which contains a lock, a condition variable and an integer (denoting the number of processes which have reached barrier).
  - We declare the barrier_array of size BARRIER_ARRAY_SIZE (defined as 10 in param.h) in proc.h.
  - We initialize the n of barriers of the barrier_array to -1 and also their locks.
- **barrier_alloc():**
  - We go through the barrier array and while holding the lock, check if the barrier at an index is available or not. If it is available, we update its count to zero, release its lock, return its index else we loop.
- **barrier():**
  - We acquire the lock for the barrier id passed. We increment its count by 1. If the count becomes equal to the number of processes waiting on that barrier, we send a broadcast else we wait on it. We print that we are done and release the sleep lock on the barrier.
- **barrier_free():**
  - We acquire the lock for the barrier instance, reset its count to 0 and then release the lock.
- **Condition variable bounder buffer:**
  - We define a structure for an element of the Buffer for the condprodconstest containing x for the item value, a full flag, a sleeplock for the element, two condition variables to check for inserted and deleted each.
  - We define the array of the above structure of the size 20. We also have 3 other locks for delete, insert and print. We also have two indices, head and tail for the producers and consumers to find a buffer slot.
- **buffer_cond_init():**
  - We initialize the elements of the buffer to -1. We also initialize the locks and set the full flag to 0 for each buffer element.
  - We initialize the sleeplocks, delete, print and insert. We also set head and tail to 0.
- **cond_produce():**

- We utilize the code given in multi_prod_multi_cons.c.
- We hold the insert lock, obtain the index to produce and then increment the tail and release the lock. We wait until that slot is empty and then fill that slot with the passed argument and mark the full flag as 1. We then send inserted signal and release the lock of the buffer element.

- **cond_consume():**
  - We again utilize the code given in multi_prod_multi_cons.c.
  - We acquire the delete lock. We obtain the consume index, then increment the head and release the lock. We then wait for the buffer element to get a value and set the full flag to 0. Then we signal on the deleted condition variable, release the lock on the buffer element and print the consumed value.

- **Semaphore:**
  - We define the structure of semaphore in semaphore.h which has the integer semaphore value, a condition variable cv and a sleeplock.
  - We implement the functions sem_init(), sem_wait(), and sem_post() in semaphore.c.
  - sem_init() sets the initial value of the semaphore to the passed value, x. It also initializes the lock.
  - sem_wait() acquires the lock of the semaphore, and waits on the condition variable until the lock value becomes positive. Then, we decrement the value of semaphore, release the lock and return. Note that the value of the semaphore never becomes negative here.
  - sem_post() acquires the lock of the semaphore, increments its value, sends a signal to processes waiting on its condition variable and then releases the lock.

- **Semaphore bounded buffer:**
  - The buffer itself is an integer array of size 20. Also, we have 4 semaphores: pro, con, empty and full. And we also have two index variables nextp and nextc for the producers and consumers to find a slot.

- **buffer_sem_init():**
  - We initialize each element of the buffer to -1. Also, we initialize the semaphores full, empty, pro, and con and we set the value of nextp and nextc indices to zero.

- **sem_produce():**
  - We utilize the code given in slide 63.
  - First we wait on semaphore empty to get an empty slot in the buffer. Then we wait on the semaphore pro and then produce on the nextp index, increment it and then we post on both the semaphores.

- **sem_consume():**
  - We utilize the code given in slide 63.
  - We wait on the semaphore full and then on the semaphore con. We then consume the item on the index nextc, increment it and then post on both con and semaphores. We return and print the produced items.

- **condprodconstest vs. semprodconstest:**

  Tabulation of some execution times as observed from our code:

  (Note that for the values of 10 and 100 for number of items, the TIMER_INTERVAL was set to 100000 and for number of producers equal to 400, the TIMER_INTERVAL was set to 1000000).

Input	**semprodconstest**	**condprodconstest**
10 1 1	2	2
10 2 3	4	3
10 3 2	4	4
10 5 5	7	7
100 1 1	10	10
100 2 3	31	21
100 3 2	48	33
100 5 5	75	60
400 1 1	6	4
400 2 3	15	11
400 3 2	23	19
400 5 5	42	28

  So, we observe that semprodconstest on average takes more time (roughly 1.5 times for the non insignificant cases where the runtime is dominated by) than that taken by condprodconstest.

  For the condition variable implementations of produce and consume, we always acquire the next index, whether that was full or not. However, even if there were multiple processes waiting on the lock of the same buffer index, only one of them produce/consume. This is different from that in case of the semaphore implementation where we first check if the buffer has any element to consume or an empty slot to produce and then only get into it. This adds more overhead to the process of acquiring/releasing locks and thus our process spends more time waiting on average.