Group 18 Submission:

Mehta Shrey Kartik - 200580
Shashwat Gupta - 200923
Priya Gole - 200727
Harshit Kumar Tiwari - 200432

Implementation

1. condition variable
   The condition variable is implemented in files condvar.h and condvar.c. condvar.h defines struct cond_t and condvar.c implements functions associated with the condition variable namely :

   - void cond_init(struct cond_t *cv); // Initialise the condition variable which is just a sleeplock initialization
   - void cond_wait(struct cond_t *cv, struct sleeplock *lock); // set the current process to wait by using the condsleep() function
   - void cond_signal(struct cond_t *cv); // signal any 1 waiting process to wakeup by using the wakeupone() function
   - void cond_broadcast(struct cond_t *cv); // signal all the waiting processes to wake up and enter the ready queue by using wakeup() function

   Several auxiliary functions were defined:

- void condsleep(struct cond_t * cv, struct sleeplock* lk) // in proc.c

  This function is essentially sleep function suited to take cond_t argument instead of void* argument of usual sleep.

- wakeupone(void* chan)
  This function is to wake up just any one process

All above functions were declared in defs.h for smooth compilation.

2. Semaphore
   Semaphores are implemented in semaphore.h and semaphore.c. the header file defines the semaphore struct while the .c files defines the following methods associated with it as has been described in the lectures.

- void sem_init (struct semaphore *s, int x) // This is to initialise the semaphore
- void sem_wait (struct semaphore *s) // this functions decrements semaphore value and blocks if the wait gets negative
- void sem_post (struct semaphore *s) // This increments semaphore value and is non-blocking. The waiting processes are can enter ready queue

All above functions were declared in defs.h for smooth compilation.

3. system calls.
   The following systemcalls were implemented to test the implementation of semaphore and condition variable.

   A. Barrier_alloc
   This function iterates over barrier array and if any barrier is found to be empty, it returns the barrier index. Proper locks were acquired while checking if a barrier id id empty or not.

   B. Barrier

   Upon correct barrier-instance, barrier-id and number of processes, the function acquires a lock of the barrier_id barrier. It then increments the count of processes waiting on the barrier and if it is equal to n, the barrier signals all the processes (cond_broadcast call) to wake up all the waking processes for the given barrier id. If the count is not n, it blocks the current process (cond_wait call). A corresponding message is displayed when the process enters and exits the barrier. Upon printing the exit message, the barrier releases its sleeplock. Proper locks were acquired while printing the messages.

   C. barrier_free

   Upon getting correct barrier_id it resets the counter of barrier and re-initialises the sleeplock and condition variable of the barrier.

   D. buffer_cond_init

   This initialises various condition variables which are used for the bounded buffer problem

   implemented using condition variables.

   E. Cond_produce

   Producer function which takes the produced item from the condprodconstest.c user

   function and adds it to the free space in the buffer. It waits (cond_wait()) until it finds

   and empty buffer

   entry and then inserts the produced item there. Proper locks were added to ensure the

   mutual exclusion of the critical sections as has been discussed in the class.

F. cond_consume

Consumer function consumes the first entry it finds filled from the buffer and if the buffer is empty, waits for the producer to put an element in the buffer using cond_wait() . Then the consumed item is printed while acquiring proper locks for printing and all the other critical sections of the function.

G. buffer_sem_init

It initialises the empty, full and mutex(this is binary lock) which are used in implementing the bounded buffer problem using semaphores.

H. sem_produce

Producer function similar to that as implemented using condition variables. The only difference being that it has been implemented using semaphores.

I. sem_consume

Consumer function with similar implementation as of condition variables with them being implemented using seamphores.

The variables related to bounded buffer were defined in sysproc.c (as well as in main.c)

Observations:
Time taken to execute condprodconstest and semprodconstest for the same inputs :

Sr. No	No. of items	Producers	Consumers	Time by Condition Variable	Time by Semaphore
1	10	5	3	0	0
2	10	3	5	1	0
3	100	10	10	2	5
4	100	15	15	5	7
5	2000	5	5	27	47

Explanation:

There is considerable difference in times from semaphore and condition variable. The condition variable are faster than semaphores.

Condition variables have a more concurrent implementation. This is because the condition variables have the producer and consumer split into different critical sections, thus allowing for more concurrency. In case of semaphores, only one producer or consumer can be there in the critical section, whereas in the condition variable case, there may be multiple producers or consumers which can update the buffer, allowing more concurrency.

Also, condition variables have less statements, so they have less time of running. The condition variables only have a wait queue, whereas the semaphores have a wait queue, a counter and a sleeplock for implementing the bounded buffer problem, and so allow lesser time for the condition variables.