CS330 Group 28 Assignment 3

Aryan Vora(200204) **S Pradeep(200826)** **Tejas Ramakrishnan(201050)**

Yash Gupta(201144)

1 Part 1: Condition variable implementation

Structure of condition variable was in file condvar.h as cond_t.

This structure contains only sleeplock lk which is used for holding sleeplock lk.

1.1 COND_WAIT

Implemented in condvar.c as cond_wait() which simply takes pointer to condition variable as cond_t*cv and sleeplock as *lock as arguments and call condsleep system call.

Function-: Condwait keeps waiting for the particular signal and goes to sleep.

1.2 COND_SIGNAL

Implemented in condvar.c as cond_signal() which simply takes pointer to condition variable as cond_t*cv as argument and call wakeupone system call.

Function-: It simply wakes up the first process it finds waiting on the condition variable cv.

1.3 COND_BROADCAST

Implemented in condvar.c as cond_broadcast() which simply takes pointer to condition variable as cond_t*cv as argument and call wakeup system call.

Function-: It wakes all the process that are currently waiting for the signal cv on channel.

1.4 CONDSLEEP

Implemented in proc.c as system call, which takes a condition variable cv and sleeplock lock as arguments and change the the state of current running process to SLEEPING by first holding lock for current process and releasing sleeplock and finally call sched(), which on correct implementation never returns.

Function-: It is similar to sleep syscall, but with extra power to sleep on a particular condition.

1.5 WAKEUPONE

Implemented in proc.c as system call, which takes chan as argument and iterate over process table and check each process whether it is sleeping on chan or not, and if it is, then wake that process and return without checking for any other process.

Function-: Its function is to wake up first encountered process in process table which is sleeping on chan without checking for other processes. This is the main difference between wakeup and wakeupone syscall.

2 Part 2: Semaphore implementation

Structure of semaphore was in file semaphore.h.

This structure contains sleeplock lk, condition variable cv, and value v which dictates what is the current value of semaphore and for what condition it was made for.

2.1 SEM_INIT

Implemented in semaphore.c which takes a semaphore structure and value as arguments. Value is used for the initialization of semaphore which serves the real purpose of semaphore's power, help to behave differently in different scenarios.

Function-: Used for the initialization of semaphore.

2.2 SEM_WAIT

Implemented in semaphore.c which takes a semaphore structure as argument. It first holds the semaphore lock and calls condsleep (with condition as that of semaphore and lock of semaphore itself) if the value of semaphore is 0, and finally decrements its value by 1 and releases lock.

Function-: Its function is to wait for the condition.

2.3 SEM_POST

Implemented in semaphore.c which takes a semaphore structure as argument. It first hold the semaphore lock so as to increment the value of semaphore by 1 and signals all waiting semaphore on condition cv by calling cond_signal.

Function-: Its function is signals all the waiting semaphore on condition cv.

3 Part 3: System calls for Testing

3.1 BARRIER_ALLOC

Implemented in proc.c as system call. It didn't take any argument, but simply iterate over barrier array to check whether there is some free barrier and if found then allocate it by assigning barrier.free=1, and barrier.count=0 (as no threads), and finally calls initsleeplock to initialize sleeplock for barrier_lock, print_lock, and barrier_cond.lk

Function-: The barrier_alloc system call will find a free barrier from the barrier array and return its id to the user program if there exist one, and if not then simply return -1, indicating none of the barrier are free.

3.2 BARRIER

Implemented in proc.c as system call. It takes barrier instance number (bin), barrier array id(id), and number of processes(n) as arguments and prints the respective required lines using print_lock(defined only for printing). It increments barrier.count by 1(indicating number of procecces currently waiting at barrier) and if this is equal to barrier instance no. then wakeup all waiting process(Broadcast) else just wait for this condition.

Function-: this system call implements the barrier using condition variables. It takes three arguments: barrier instance number, barrier array id, and number of processes.

3.3 BARRIER_FREE

Implemented in proc.c as system call. It simply takes id as an argument and just free barrier with id=id.

Function-: Used to free barrier with id=id, i.e. creating space in barrier array.

3.4 BUFFER_COND_INIT

Implemented in proc.c as system call. It initialises the buffer_elem structure defined in bounded-Buffer.h file.

Function-: It initialises the sleep locks, conditional variables and also the full and x variables in the buffer.

3.5 COND_PRODUCE

Implemented in proc.c. It takes as argument the produced value and it waits till the buffer at that index is not full and calls condsleep on deleted condvar to wait till some value gets consumed in the buffer. It also calls condsignal on inserted condvar to wake up any consumers waiting for a value to be inserted.
Function-: Function of this syscall is to produce a product and then persistently add it in the buffer so as there is no race condition between any producer.

3.6 COND_CONSUME

Implemented in proc.c, it takes no argument. It waits on condvar inserted if the buffer is not full so that some producer inserts product. It reads the buffer and prints the value once it is woken up. Additionally, sends signal on condvar deleted after setting the buffer to not full.
Function-: Function of this syscall is to consume a product from the buffer persistently and shifting the head pointer precisely so that no race condition appears.

3.7 BUFFER_SEM_INIT

Implemented in proc.c as system call. It initialises the sem_buffer_elem structure defined in bounded-Buffer.h file.
Function-: It initialises the sleep lock and the full and x variables in the buffer. Additionally it initialises the global semaphores empty, full, pro, con in proc.c

3.8 SEM_PRODUCE

Implemented in proc.c as system call. It only takes x(value to be produced) as argument and check for the condition whether the buffer has space using semaphore sem_wait(&empty) if not then wait for this condition to be true else again check for the condition that no other producer is producing a product(i.e. no other producer is also in critical section), if not then add the produced quantity(x) in the buffer and increment next pointer(for accessing next index in buffer for some other producer) and finally signals all the producer which are waiting for this condition.

Function-: Function of this syscall is to produce a product and then persistently add it in the buffer so as there is no race condition between any producer.

3.9 SEM_CONSUME

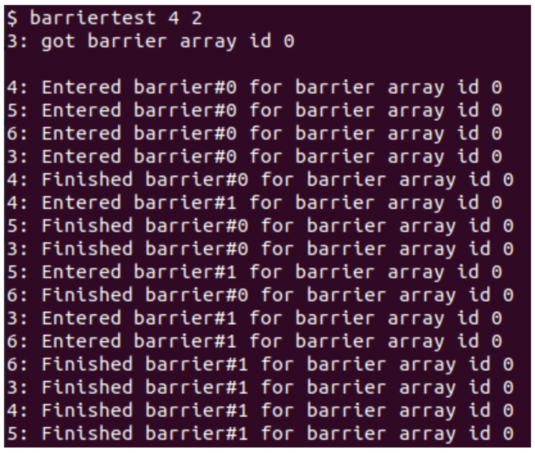
Implemented in proc.c as system call. It didn't take any argument but consume the ones that are already present there in the buffer. It first check for the condition that there is some product to consume in buffer using semaphore sem_wait(&full), and if yes then again check for the condition that there are no more consumer in the critical section consuming any product, and after both condition satisfied it simply takes out a product from buffer and shift the nextc pointer for some other consumer to consume, and then finally signals all the consumer to begin consuming.

Function-: Function of this syscall is to consume a product from the buffer persistently and shifting the nextc pointer precisely so that no race condition appears.

4 Part 4: Observations

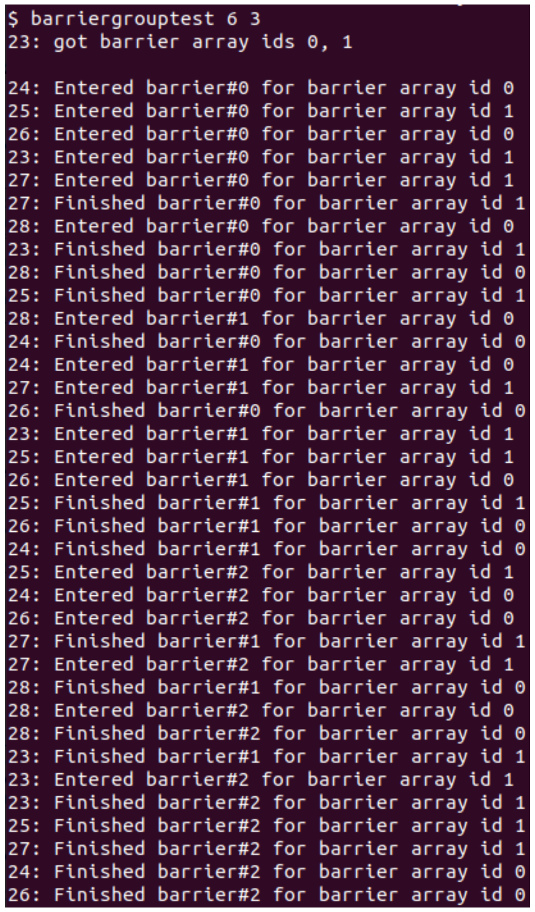
4.1 Barrier Testing

BARRIERTEST and BARRIERGROUPTEST user calls were implemented using the mentioned 3 system calls(BARRIER_ALLOC, BARRIER, BARRIER_FREE). Sample outputs of the user calls is as follows:



```
$ barriertest 4 2
3: got barrier array id 0

4: Entered barrier#0 for barrier array id 0
5: Entered barrier#0 for barrier array id 0
6: Entered barrier#0 for barrier array id 0
3: Entered barrier#0 for barrier array id 0
4: Finished barrier#0 for barrier array id 0
4: Entered barrier#1 for barrier array id 0
5: Finished barrier#0 for barrier array id 0
3: Finished barrier#0 for barrier array id 0
5: Entered barrier#1 for barrier array id 0
6: Finished barrier#0 for barrier array id 0
3: Entered barrier#1 for barrier array id 0
6: Entered barrier#1 for barrier array id 0
6: Finished barrier#1 for barrier array id 0
3: Finished barrier#1 for barrier array id 0
4: Finished barrier#1 for barrier array id 0
5: Finished barrier#1 for barrier array id 0
```

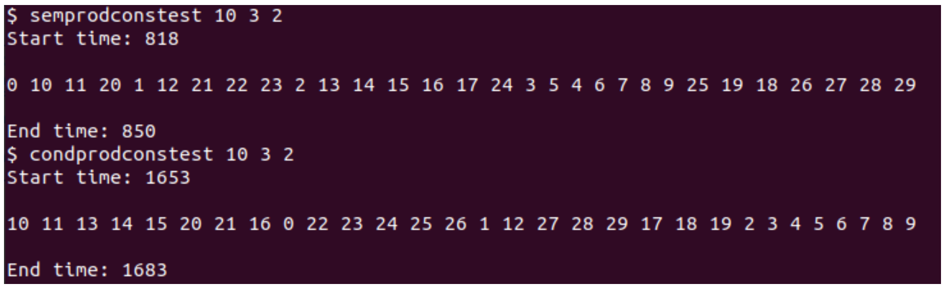


```
$ barriergrouptest 6 3
23: got barrier array ids 0, 1

24: Entered barrier#0 for barrier array id 0
25: Entered barrier#0 for barrier array id 1
26: Entered barrier#0 for barrier array id 0
23: Entered barrier#0 for barrier array id 1
27: Entered barrier#0 for barrier array id 1
27: Finished barrier#0 for barrier array id 1
28: Entered barrier#0 for barrier array id 0
23: Finished barrier#0 for barrier array id 1
28: Finished barrier#0 for barrier array id 0
25: Finished barrier#0 for barrier array id 1
28: Entered barrier#1 for barrier array id 0
24: Finished barrier#0 for barrier array id 0
24: Entered barrier#1 for barrier array id 0
27: Entered barrier#1 for barrier array id 1
26: Finished barrier#0 for barrier array id 0
23: Entered barrier#1 for barrier array id 1
25: Entered barrier#1 for barrier array id 1
26: Entered barrier#1 for barrier array id 0
25: Finished barrier#1 for barrier array id 1
26: Finished barrier#1 for barrier array id 0
24: Finished barrier#1 for barrier array id 0
25: Entered barrier#2 for barrier array id 1
24: Entered barrier#2 for barrier array id 0
26: Entered barrier#2 for barrier array id 0
27: Finished barrier#1 for barrier array id 1
27: Entered barrier#2 for barrier array id 1
28: Finished barrier#1 for barrier array id 0
28: Entered barrier#2 for barrier array id 0
28: Finished barrier#2 for barrier array id 0
23: Finished barrier#1 for barrier array id 1
23: Entered barrier#2 for barrier array id 1
23: Finished barrier#2 for barrier array id 1
25: Finished barrier#2 for barrier array id 1
27: Finished barrier#2 for barrier array id 1
24: Finished barrier#2 for barrier array id 0
26: Finished barrier#2 for barrier array id 0
```

4.2 Multiple Consumer-Producer Testing

CONDPRODCONSTEST user call is implemented using the mentioned 3 system calls(BUFFER_COND_INIT, COND_PRODUCE, COND_CONSUME), while SEMPROD-CONSTEST user call is implemented using the 3 mentioned system calls(BUFFER_SEM_INIT, SEM_PRODUCE, SEM_CONSUME) and both these system calls simulate multiple consumers and producers using a bounded buffer of size 20. Sample outputs of the user calls is as follows:



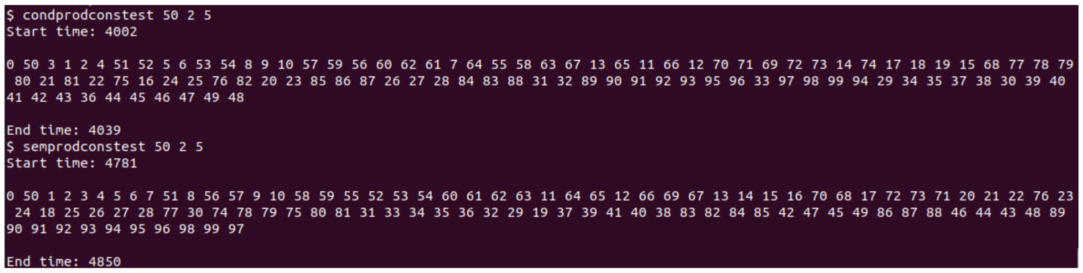
```
$ semprodconstest 10 3 2
Start time: 818

0 10 11 20 1 12 21 22 23 2 13 14 15 16 17 24 3 5 4 6 7 8 9 25 19 18 26 27 28 29

End time: 850
$ condprodconstest 10 3 2
Start time: 1653

10 11 13 14 15 20 21 16 0 22 23 24 25 26 1 12 27 28 29 17 18 19 2 3 4 5 6 7 8 9

End time: 1683
```



```
$ condprodconstest 50 2 5
Start time: 4002

0 50 3 1 2 4 51 52 5 6 53 54 8 9 10 57 59 56 60 62 61 7 64 55 58 63 67 13 65 11 66 12 70 71 69 72 73 14 74 17 18 19 15 68 77 78 79
 80 21 81 22 75 16 24 25 76 82 20 23 85 86 87 26 27 28 84 83 88 31 32 89 90 91 92 93 95 96 33 97 98 99 94 29 34 35 37 38 30 39 40
41 42 43 36 44 45 46 47 49 48

End time: 4039
$ semprodconstest 50 2 5
Start time: 4781

0 50 1 2 3 4 5 6 7 51 8 56 57 9 10 58 59 55 52 53 54 60 61 62 63 11 64 65 12 66 69 67 13 14 15 16 70 68 17 72 73 71 20 21 22 76 23
 24 18 25 26 27 28 77 30 74 78 79 75 80 81 31 33 34 35 36 32 29 19 37 39 41 40 38 83 82 84 85 42 47 45 49 86 87 88 46 44 43 48 89
90 91 92 93 94 95 96 98 99 97

End time: 4850
```

4.3 Comparison between Semaphores and Condition Variables for multiple consumers and producers:

# Items per producer	# Producers	# Consumers	Time taken(SEM)	Time taken(COND)
10	3	2	32	30
20	3	2	37	24
50	2	5	69	37
50	10	5	84	42
20	5	2	13	8

We see that the condition variable implementation of multiple consumers and producers is faster because there is lesser concurrency in the semaphore implementation compared to that of condition variables.