B. TECH - COMPUTER SCIENCE AND ENGINEERING (V SEMESTER)

CST 303Concurrent and Parallel Programming Lab

Week: 5 **Batch:2**

1. Implement solution of Critical Section problem with Semaphores (N processes).

Algorithm 6.3: Critical section with semaphores (N proc.)
binary semaphore S ← $(1, \emptyset)$
loop forever
p1: non-critical section
p2: wait(S)
p3: critical section
p4: signal(S)

2. Implement Merge-sort using Semaphores.

Algorithm 6.5: Mergesort		
integer array A binary semaphore S1 ← $(0, \emptyset)$ binary semaphore S2 ← $(0, \emptyset)$		
sort1	**sort2**	**merge**
p1: sort 1st half of A	q1: sort 2nd half of A	r1: wait(S1)
p2: signal(S1)	q2: signal(S2)	r2: wait(S2)
p3:	q3:	r3: merge halves of A

3. Implement producer-consumer problem with Semaphores (finite buffer).

Algorithm 6.8: Producer-consumer (finite buffer, semaphores)	
finite queue of dataType buffer ← empty queue semaphore notEmpty ← $(0, \emptyset)$ semaphore notFull ← $(N, \emptyset)$	
producer	**consumer**
dataType d	dataType d
loop forever	loop forever
p1: d ← produce	q1: wait(notEmpty)
p2: wait(notFull)	q2: d ← take(buffer)
p3: append(d, buffer)	q3: signal(notFull)
p4: signal(notEmpty)	q4: consume(d)

4. Implement dining philosopher's first attempt to solve critical section problem.

Algorithm 6.10: Dining philosophers (first attempt)	
semaphore array [0..4] fork ← [1,1,1,1,1]	
	loop forever
p1:	think
p2:	wait(fork[i])
p3:	wait(fork[i+1])
p4:	eat
p5:	signal(fork[i])
p6:	signal(fork[i+1])

5. Implement dining philosopher's second attempt to solve critical section problem.

Algorithm 6.11: Dining philosophers (second attempt)	
semaphore array [0..4] fork ← [1,1,1,1,1]	
semaphore room ← 4	
	loop forever
p1:	think
p2:	wait(room)
p3:	wait(fork[i])
p4:	wait(fork[i+1])
p5:	eat
p6:	signal(fork[i])
p7:	signal(fork[i+1])
p8:	signal(room)

6. Implement dining philosopher's third attempt to solve critical section problem.

Algorithm 6.12: Dining philosophers (third attempt)	
semaphore array [0..4] fork ← [1,1,1,1,1]	
philosopher 4	
	loop forever
p1:	think
p2:	wait(fork[0])
p3:	wait(fork[4])
p4:	eat
p5:	signal(fork[0])
p6:	signal(fork[4])

7. Implement critical section problem with *fetch and add*

Algorithm 3.12: Critical section problem with exchange	
integer common ← 1	
p	**q**
integer local1 ← 0	integer local2 ← 0
loop forever	loop forever
p1: non-critical section	q1: non-critical section
repeat	repeat
p2: exchange(common, local1)	q2: exchange(common, local2)
p3: until local1 = 1	q3: until local2 = 1
p4: critical section	q4: critical section
p5: exchange(common, local1)	q5: exchange(common, local2)

Replace exchange function with *fetch and add(common, local,x)*

fetch_and_add(comman, local, x)
local ←common
common ←common + x