Dear Students,

Your assignment submission is due on (June 26th, 2016).

**Assignment Statement**  
**A. Scheduling**

**i.** Simulate **TAKE** and **Central Queue** scheduling algorithms for multiprocessor scheduling as discussed in the paper:

*S. Curran and M Stumm, “A Comparison of basic CPU Scheduling Algorithms for Multiprocessor Unix”, Computer Systems, 3(4), Oct., 1990, pp. 551--579.*

Your simulation should take processes’ details (processors #, arrival time, priority, process name etc) via a text file named “input.txt” and output the resultant schedule as per required algorithms in a text file named “output.txt”.

**ii.** Simulate **Multilevel feedback FIFO/RR queue** as described below:

1. A new process is inserted at the end (tail) of the top-level FIFO queue.

2. At some stage the process reaches the head of the queue and is assigned the CPU.

3. If the process is completed within the time quantum of the given queue, it leaves the system.

4. If the process voluntarily relinquishes control of the CPU, it leaves the queuing network, and when the process becomes ready again it is inserted at the tail of the same queue which it relinquished earlier.

5. If the process uses all the quantum time, it is pre-empted and inserted at the end of the next lower level queue. This next lower level queue will have a time quantum which is more than that of the previous higher level queue.

6. This scheme will continue until the process completes or it reaches the base level queue.

7. At the base level queue the processes circulate in round robin fashion until they complete and leave the system.

8. Optionally, if a process blocks for I/O, it is 'promoted' one level, and placed at the end of the next-higher queue. This allows I/O bound processes to be favored by the scheduler and allows processes to 'escape' the base level queue.

9. For scheduling, the scheduler always start picking up processes from the head of the highest level queue. If the highest level queue has become empty, then only will the scheduler take up a process from the next lower level queue. The same policy is implemented for picking up in the subsequent lower level queues. Meanwhile, if a process comes into any of the higher level queues, it will preempt a process in the lower level queue.

10. Also, a new process is always inserted at the tail of the top level queue with the assumption that it would be a short time consuming process. Long processes will automatically sink to lower level queues based on their time consumption and interactivity level. In the multilevel feedback queue, a process is given just one chance to complete at a given queue level before it is forced down to a lower level queue.

11. You should use two input text files to implement this algorithm, namely “input.txt” and “parameters.txt”. “input.txt” should be populated using top command in Linux (i.e. using the realistic process load). “parameters.txt” should contain all modifiable parameters used by your algorithm. Some of the parameters are suggested as below:

*The number of queues, time quanta for round-robin for each queue,*

Your code should at least define following methods:

*The method used to determine when to promote a process to a higher priority queue.*

*The method used to determine when to demote a process to a lower priority queue.*

*The method used to determine which queue a process will enter when that process needs service.*

Your simulation should output the resultant schedule in a text file named “output.txt”.

**Instructor may change parameters in the input.txt file during evaluation.**

**B. Buddy Systems**

Simulate memory management scheme of Buddy systems, with an improvement that permits you to re-utilize internally fragmented blocks.

**C. Synchronization**

Please provide a solution to “shark and fish” synchronization problem. Provide a solution to this problem (described below) using semaphores.

**Problem:**

S sharks and F fish live in an ocean with only ONE sea food-point serving sea-weed having N seats. A Fish can dine at a sea-food point as long as a shark does not arrive there simultaneously. If the shark sees the fish, it must eat the fish. There are N seats at the sea-food point, S sharks and F fish. Our sharks and fish love sea-weed, however, shark does not mind eating fish as well.

Please write code to simulate sharks and fish. Create one thread for each simulated shark, and one thread for each simulated fish. Each shark thread repeatedly sleeps/blocks and then wakes up to eat from the sea-food point, by occupying a seat available there. Each fish thread behaves similarly. Each shark and fish iterates a fixed number of times before terminating. Your should synchronize the sharks and the fish to meet the following requirements:

1. Shark and fish should not eat at the same time. i.e. if a shark is occupying a seat at the sea-food point then no fish should be sitting at any seat in the sea-food point. Similarly, if a fish is sitting at a seat in the sea-food point, then no shark should be eating there. This will ensure that the fish is not consumed by the sharks.  
2. Only one fish or one shark can sit at a given seat at any particular time.

3. Neither sharks nor fish should starve. A shark or fish that wants to eat should eventually be able to eat. e.g. a solution that permanently prevents all fish from eating would be unacceptable. So would a solution in which sharks were always given priority over fish. When we actually test your solution, each simulated  
shark and fish will eat a finite number of times; however, even if the simulation were allowed to run forever, neither sharks nor fish should starve.

4. Your synchronization mechanism should not impose an upper bound on the number of seats that can be occupied simultaneously. i.e. if there are enough animals, it should be possible for them to occupy all available seats simultaneously.

Provide a file synch.c to provide solution to synchronization. Your solution should be implemented entirely in this file.

It can contain six functions, which are invoked by the "shark and fish simulation" created by you.  
sharkb4eating: called each time a shark eats, before it eats

fishb4eating: called each time a fish eats, before it eats

sharkafteating: called each time a shark eats, after it eats

fishafteating: called each time a fish eats, after it eats

sharkfishsyncinit: called only once, before the shark and fish are created

sharfishsynccleanup: called only once, after all sharks and fish have finished

**Implement your solution using functions as described above.**

Use the sharkb4eating and fishb4eating functions to make animals wait before eating, when waiting is necessary to satisfy the synchronization requirements. e.g. if a fish is eating and a shark arrives at sea-food point, the shark must be prevented from sitting in, at least, until the fish has finished, otherwise requirement.1 will be violated.  To enforce this, your implementation of sharkb4eating should cause that shark (thread) to block until it is OK for it to eat. In other words, sharkb4eating should not return until it is OK for the shark to eat, since once it returns the shark will eat.

It is up to you to decide when it is OK for a shark or fish to eat. You should answer this question when designing the synchronization mechanism that satisfies the requirements. You may use sharkfishsyncinit to create or initialize any synchronization primitives or variables that your solution requires.  
  
**Testing**  
You can launch the shark and fish simulation menu using "syncproblem" command. In its short form, it should take 4 parameters, i.e., "syncproblem 2 3 4 6"

These parameters specify the number of seats, the number of sharks, the number of fish, and the number of eat/sleep iterations each animal makes before finishing.

Thus, the command above will simulate 3 sharks and 4 fish eating using 2 seats in the food-point, with each shark and fish iterating 6 times. A longer version should allow you to specify the amount of time each animal spends eating and sleeping.

When the simulation program terminates, it should print simulation statistics:

utilization of the seats and the average waiting times for sharks and for fish.

These statistics will tell whether your synchronization mechanism satisfies the synchronization requirements. When there are equal numbers of sharks and fish, a synchronization mechanism that does not starve sharks or fish should show similar waiting times for sharks and fish. Also, for a given number of seats, an efficient solution should show increasing bowl utilization if we run a series of simulations with larger and larger numbers of animals.

**Please following the submission guidelines as provided below.**

**Submission Guidelines:**

1. Kindly provide code documentation with comments.

2. In case of multiple files or project, prepare a zip file containing all project/code files and add an additional text file named "Readme.txt" explaining the role of each file.

3. Please keep files for part A, B, and C in different folders named “partA”, “partB” and “partC” respectively.

4. Assignment can be done in a group of FOUR students. Kindly add your group members’ details in the following google-sheet:

<https://drive.google.com/open?id=1b4kDEWbOWVj41JHF7oM6E_3b_VOmr8BIk0s8nO8sLpw>

5. Submitted main folder should be compressed zip file named as "YourGroupNo\_PA.zip" (e.g. 2\_PA.zip for group#2) and should be submitted at following dropbox link: <https://www.dropbox.com/request/jiNF21AWyXeWkezfza7V>