WEEK-END ASSIGNMENT-S02
Process/Thread Synchronization
Operating Systems Workshop (CSE 3541)

Problem Statement:

Working with POSIX thread and process synchronization techniques using semaphores.

Assignment Objectives:

Students will be able to identify critical section problem and synchronize the process actions to protect critical section.

Instruction to Students (If any):

Students are required to write his/her own program/output by avoiding any kind of copy from any sources. Students may use additional pages on requirement.

Programming/ Output Based Questions:

1. Find the critical section and determine the race condition for the given pseudo code. Write program using **fork()** to realize the race condition(**Hint:** *case-1:* Execute P_1 first then P_2, *case-2:* Execute P_2 first then P_1). Here **`shrd`** is a shared variables.

```
P-1 executes:
    x = *shrd;
    x = x + 1;
  sleep(1);
  *shrd = x;
```

```
P-2 executes
    y = *shrd;
    y = y - 1;
    sleep(1);
    *shrd = y;
```

Code here

```
#include <stdio.h>
#include <unistd.h>
#include <sys/wait.h>
int shrd = 0;
void process(int inc) {
  int x = shrd;
  x += inc;
  sleep(1);
  shrd = x;
  printf("Process: shrd = %d\n", shrd);
}
int main() {
  pid_t pid1, pid2;
  pid1 = fork();
  if (pid1 == 0) process(1);
  else {
     pid2 = fork();
     if (pid2 == 0) process(-1);
     else {
        waitpid(pid1, NULL, 0);
        waitpid(pid2, NULL, 0);
        printf("Final value: %d\n", shrd);
     }
  }
  return 0;
}
```

Critical section: The lines where shrd is read and updated.

Race condition: Occurs due to unsynchronized access to shrd.

2. Race conditions are possible in many computer systems. Consider a banking system that maintains an account balance with two functions: **`deposit(amount)`** and **`withdraw(amount)`**. These two functions are passed the amount that is to be deposited or withdrawn from the bank account balance. Assume that a husband and wife share a bank account. **Concurrently**, the husband calls the **`withdraw()`** function and the wife calls **`deposit()`**. Describe how a race condition is possible and what might be done to prevent the race condition from occurring. Develop a C code to show the race condition.

Code here

```
#include <stdio.h>
#include <unistd.h>
#include <pthread.h>
int balance = 1000;
void *deposit(void *arg) {
   int amount = *(int *)arg;
   balance += amount;
   printf("Deposited %d, balance = %d\n", amount, balance);
   return NULL;
}
void *withdraw(void *arg) {
   int amount = *(int *)arg;
   if (balance >= amount) {
      balance -= amount;
      printf("Withdrew %d, balance = %d\n", amount, balance);
   } else {
      printf("Insufficient funds\n");
   }
   return NULL;
}
int main() {
   pthread_t thread1, thread2;
   int amount1 = 500, amount2 = 300;
   pthread_create(&thread1, NULL, deposit, &amount1);
   pthread_create(&thread2, NULL, withdraw, &amount2);
   pthread_join(thread1, NULL);
   pthread_join(thread2, NULL);
   printf("Final balance: %d\n", balance);
   return 0;
}
```

Race Condition:
Both deposit and withdraw functions access and modify the shared balance variable.
If they execute concurrently without synchronization, their operations might interleave, leading to incorrect results.
For example, the wife's deposit might be overwritten by the husband's withdrawal, even though the deposit happened first.

Prevention:

Synchronization: Use mechanisms like mutexes or semaphores to ensure exclusive access to the shared variable during critical sections.

3. There will be a race among the two processes in question No.-2. Find a solution to avoid the race condition using semaphore. (Hint: try using named and unnamed POSIX semaphores).

Code here

```
#include <stdio.h>
#include <unistd.h>
#include <semaphore.h>

int shrd = 0;
sem_t sem;

void process(int inc) {
  sem_wait(&sem);
  int x = shrd;
  x += inc;
  sleep(1);
  shrd = x;
  printf("Process: shrd = %d\n", shrd);
  sem_post(&sem);
}

int main() {
  sem_init(&sem, 1, 1);

  pid_t pid1 = fork();
  if (pid1 == 0) process(1);
  else {
     pid_t pid2 = fork();
     if (pid2 == 0) process(-1);
     else {
        waitpid(pid1, NULL, 0);
        waitpid(pid2, NULL, 0);
        printf("Final value: %d\n", shrd);
     }
  }

  sem_destroy(&sem);

  return 0;
}
```

Prevention from race condition:
Replace sem_init(&sem, 1, 1) with sem_t sem; sem_init(&sem, 0, 1).
Unnamed semaphores are local to a process, so sharing them across processes requires manual sharing oftheir addresses.

4. Suppose **process 1** must execute statement **a** before **process 2** executes statement **b**. The semaphore **sync** enforces the ordering in the following pseudocode, provided that **sync** is initially **0**.

```
Process 1 executes:              Process 2 executes:
   a;                               wait(&sync);
   signal(&sync);                   b;
```

Because **sync** is initially **0**, **process 2** blocks on its **wait** until **process 1** calls **signal**. Now, You develop a C code using **POSIX:SEM semaphore** to get the desired result.

Code here

```
#include <stdio.h>
#include <unistd.h>
#include <semaphore.h>

sem_t sync;

void process1() {
  printf("Process 1: a\n");
  sem_post(&sync);
}

void process2() {
  sem_wait(&sync);
  printf("Process 2: b\n");
}

int main() {
  sem_init(&sync, 0, 0);

  pid_t pid1 = fork();
  if (pid1 == 0) process1();
  else {
     process2();
     wait(NULL);
  }

  sem_destroy(&sync);

  return 0;
}
```

5. Implement the following pseudocode over the semaphores S and Q. State your answer on different initializations of S and Q.

(a) S=1, Q=1
(b) S=1, Q=0
(c) S=0, Q=1
(d) S=0, Q=0
(e) S=8, Q=0

```
Process 1 executes:  Process 2 executes:
for( ; ; ) {           for( ; ; ) {
  wait(&S);                 wait(&Q);
  a;                        b;
  signal(&Q);               signal(&S);
}                      }
```

Code here

```
#include <stdio.h>
#include <unistd.h>
#include <semaphore.h>

sem_t S, Q;

void process1() {
   for (;;) {
      sem_wait(&S);
      printf("Process 1: a\n");
      sem_post(&Q);
   }
}

void process2() {
   for (;;) {
      sem_wait(&Q);
      printf("Process 2: b\n");
      sem_post(&S);
   }
}

int main() {
   // Initialize semaphores based on scenarios

   pid_t pid1 = fork();
   if (pid1 == 0) {
      process1();
   } else {
      process2();
      wait(NULL);
   }

   return 0;
}
```

6. Implement and test what happens in the following pseudocode if semaphores S and Q are both initialized to 1?

```
process 1 executes:
   for( ; ; ) {
     wait(&Q);
     wait(&S);
     a;
     signal(&S);
     signal(&Q);
   }
```

```
process 2 executes:
   for( ; ; ) {
     wait(&S);
     wait(&Q);
     b;
     signal(&Q);
     signal(&S);
   }
```

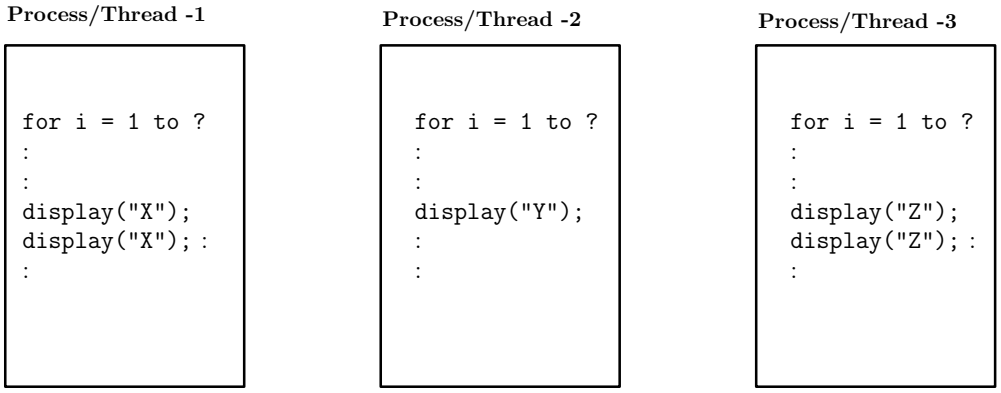
Code here

```
#include <stdio.h>
#include <unistd.h>
#include <semaphore.h>
sem_t S, Q;
void process1() {
   for (;;) {
      sem_wait(&Q);
      sem_wait(&S);
      printf("Process 1: a\n");
      sem_post(&S);
      sem_post(&Q);
   }
}

void process2() {
   for (;;) {
      sem_wait(&S);
      sem_wait(&Q);
      printf("Process 2: b\n");
      sem_post(&Q);
      sem_post(&S);
   }
}

int main() {
   sem_init(&S, 0, 1);
   sem_init(&Q, 0, 1);
   pid_t pid1 = fork();
   if (pid1 == 0) {
      process1();
   } else {
      process2();
      wait(NULL);
   }
   return 0;
}
```

7. You have three processes/ threads as given in the below diagram. Create your C code to print the sequence gievn as;
XXYZZYXXYZZYXXYZZYXXYZZ. Your are required to choose any one of the process/ thread synchromization mechanism(s).



Code here

```
#include <stdio.h>
#include <unistd.h>
#include <semaphore.h>
sem_t sem1, sem2, sem3;
void process1() {
   for (int i = 0; i < 4; i++) {
      sem_wait(&sem1);
      printf("XX");
      fflush(stdout);  // Force immediate output
      sem_post(&sem2);
   }
}
void process2() {
   for (int i = 0; i < 4; i++) {
      sem_wait(&sem2);
      printf("YZZY");
      fflush(stdout);
      sem_post(&sem3);
   }
}
void process3() {
   for (int i = 0; i < 4; i++) {
      sem_wait(&sem3);
      printf("XX");
      fflush(stdout);
      sem_post(&sem1);
   }
}
int main() {
   sem_init(&sem1, 0, 1);
   sem_init(&sem2, 0, 0);
   sem_init(&sem3, 0, 0);
   pid_t pid1 = fork();
   if (pid1 == 0) {
      process1();
   } else {
      pid_t pid2 = fork();
      if (pid2 == 0) {
         process2();
      } else {
         process3();
         wait(NULL);
         wait(NULL);
      }
   }
   sem_destroy(&sem1);
   sem_destroy(&sem2);
   sem_destroy(&sem3);
   return 0;
}
```

8. **Process ordering using semaphore**. Create 6 number of processes (1 parent + 5 children) using **`fork()`** in **`if-elseif-else`** ladder. The parent process will be waiting for the termination of all it's children and each process will display a line of text on the standard error as;

```
P1: Coronavirus
P2: WHO:
P3: COVID-19
P4: disease
P5: pandemic
```

Your program must display the message in the given order
`WHO: Coronavirus disease COVID-19 pandemic`.

Code here

```
sem_t sem1, sem2, sem3, sem4, sem5;
void process1() {
   sem_wait(&sem1);
   fprintf(stderr, "P1: Coronavirus\n");
   sem_post(&sem5);
}
void process2() {
   sem_wait(&sem2);
   fprintf(stderr, "P2: WHO:\n");
   sem_post(&sem1);
}
void process3() {
   sem_wait(&sem3);
   fprintf(stderr, "P3: COVID-19\n");
   sem_post(&sem2);
}
void process4() {
   sem_wait(&sem4);
   fprintf(stderr, "P4: disease\n");
   sem_post(&sem3);
}
void process5() {
   fprintf(stderr, "P5: pandemic\n");
   sem_post(&sem4);
}
int main() {
   sem_init(&sem1, 0, 0);
   sem_init(&sem2, 0, 0);
   sem_init(&sem3, 0, 0);
   sem_init(&sem4, 0, 0);
   sem_init(&sem5, 0, 1);
   pid_t pid1 = fork();
   if (pid1 == 0) {
      process1();
   } else {
       pid_t pid2 = fork();
      if (pid2 == 0) {
         process2();
      }
      else {
            pid_t pid3 = fork();
            if (pid3 == 0) {
               process3();
            } else {
               pid_t pid4 = fork();
               if (pid4 == 0) {
                  process4();
               } else {
                  process5();
                  wait(NULL);
                  wait(NULL);
                  wait(NULL);
                  wait(NULL);
               }
            }
         }
      }
   sem_destroy(&sem1);
   sem_destroy(&sem2);
   sem_destroy(&sem3);
   sem_destroy(&sem4);
   sem_destroy(&sem5);
   return 0;
}
```

- Write a program to give a semaphore-based solution to the producer-consumer problem using a bounded buffer scheme.

Code here

```
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <semaphore.h>

#define BUFFER_SIZE 5
int buffer[BUFFER_SIZE];
int in = 0, out = 0;
sem_t mutex, empty, full;

void* producer(void* arg) {
    int item;
    for (int i = 0; i < 10; i++) {
        item = rand() % 100;
        sem_wait(&empty);
        sem_wait(&mutex);
        buffer[in] = item;
        in = (in + 1) % BUFFER_SIZE;
        printf("Produced: %d\n", item);
        sem_post(&mutex);
        sem_post(&full);
        sleep(1);
    }
    pthread_exit(NULL);
}

void* consumer(void* arg) {
    int item;
    for (int i = 0; i < 10; i++) {
        sem_wait(&full);
        sem_wait(&mutex);
        item = buffer[out];
        out = (out + 1) % BUFFER_SIZE;
        printf("Consumed: %d\n", item);
        sem_post(&mutex);
        sem_post(&empty);
        sleep(2);
    }

    pthread_exit(NULL);
}

int main() {
    pthread_t producerThread, consumerThread;

    sem_init(&mutex, 0, 1);
    sem_init(&empty, 0, BUFFER_SIZE);
    sem_init(&full, 0, 0);

    pthread_create(&producerThread, NULL, producer, NULL);
    pthread_create(&consumerThread, NULL, consumer, NULL);

    pthread_join(producerThread, NULL);
    pthread_join(consumerThread, NULL);

    sem_destroy(&mutex);
    sem_destroy(&empty);
    sem_destroy(&full);

    return 0;
}
```

- **The Sleeping-Barber Problem.** A barbershop consists of a waiting room with n chairs and a barber room with one barber chair. If there are no customers to be served, the barber goes to sleep. If a customer enters the barbershop and all chairs are occupied, then the customer leaves the shop. If the barber is busy but chairs are available, then the customer sits in one of the free chairs. If the barber is asleep, the customer wakes up the barber. Write a program to coordinate the barber and the customers.

Code here

```
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <semaphore.h>
#include <unistd.h>

#define CHAIRS 5
sem_t customersSem, barberSem, mutex;
int waitingCustomers = 0;

void* barber(void* arg) {
   while (1) {
      sem_wait(&customersSem);
      sem_wait(&mutex);
      waitingCustomers--;
      printf("Barber is cutting hair. Waiting customers: %d\n", waitingCustomers);
      sem_post(&mutex);
      sem_post(&barberSem);
      sleep(2);
   }
}

void* customer(void* arg) {
   int customerId = *(int*)arg;
   sem_wait(&mutex);
   if (waitingCustomers < CHAIRS) {
      waitingCustomers++;
      printf("Customer %d enters the shop. Waiting customers: %d\n",
      customerId, waitingCustomers);
      sem_post(&customersSem);
      sem_post(&mutex);
      sem_wait(&barberSem);
      printf("Customer %d is getting a haircut.\n", customerId);
   } else {
      printf("Customer %d leaves the shop as there are no available chairs.\n", customerId);
      sem_post(&mutex);
   }

   pthread_exit(NULL);
}
```

Code here

```
int main() {
   pthread_t barberThread;
   pthread_t customerThreads[10];
   sem_init(&customersSem, 0, 0);
   sem_init(&barberSem, 0, 0);
   sem_init(&mutex, 0, 1);

   pthread_create(&barberThread, NULL, barber, NULL);

   int customerIds[10];
   for (int i = 0; i < 10; i++) {
      customerIds[i] = i + 1;
      pthread_create(&customerThreads[i], NULL, customer, (void*)&customerIds[i]);
      sleep(1);  // Introduce a delay between customer arrivals
   }

   pthread_join(barberThread, NULL);

   for (int i = 0; i < 10; i++) {
      pthread_join(customerThreads[i], NULL);
   }

   sem_destroy(&customersSem);
   sem_destroy(&barberSem);
   sem_destroy(&mutex);

   return 0;
}
```

- **The Cigarette-Smokers Problem**. Consider a system with three smoker processes and one agent process. Each smoker continuously rolls a cigarette and then smokes it. But to roll and smoke a cigarette, the smoker needs three ingredients: tobacco, paper and matches. One of the smoker processes has paper, another has tobacco, and the third has matches. The agent has an infinite supply of all three materials. The agent places two of the ingredients on the table. The smoker who has the remaining ingredient then makes and smokes a cigarette, signaling the agent on completion. The agent then puts out another two of the three ingredients, and the cycle repeats. Write a program to synchronize the agent and the smokers using any synchronization tool.

Code here

```
#include <stdio.h>
#include <pthread.h>
#include <semaphore.h>
#include <unistd.h>

sem_t agentSem, tobaccoSem, paperSem, matchesSem;

void* agent(void* arg) {
   while (1) {
      sem_wait(&agentSem);
      sleep(1); // Simulate agent putting ingredients on the table
      int ingredients = rand() % 3;
      switch (ingredients) {
         case 0:
            printf("Agent puts tobacco and paper on the table.\n");
            sem_post(&matchesSem);
            break;
         case 1:
            printf("Agent puts tobacco and matches on the table.\n");
            sem_post(&paperSem);
            break;
         case 2:
            printf("Agent puts paper and matches on the table.\n");
            sem_post(&tobaccoSem);
            break;
      }
   }
}

void* smoker(void* arg) {
   int* ingredient = (int*)arg;
   while (1) {
      sem_wait(ingredient);
      sem_wait(&agentSem);
      printf("Smoker with ingredient %d: Rolling and smoking a cigarette.\n", *ingredient);
      sem_post(&agentSem);
      // Simulate smoking
      sleep(2);
   }
}
```

Code here

```
int main() {
    sem_init(&agentSem, 0, 1);
    sem_init(&tobaccoSem, 0, 0);
    sem_init(&paperSem, 0, 0);
    sem_init(&matchesSem, 0, 0);

    pthread_t agentThread, smokerThread1, smokerThread2, smokerThread3;

    pthread_create(&agentThread, NULL, agent, NULL);
    pthread_create(&smokerThread1, NULL, smoker, (void*)&tobaccoSem);
    pthread_create(&smokerThread2, NULL, smoker, (void*)&paperSem);
    pthread_create(&smokerThread3, NULL, smoker, (void*)&matchesSem);

    pthread_join(agentThread, NULL);
    pthread_join(smokerThread1, NULL);
    pthread_join(smokerThread2, NULL);
    pthread_join(smokerThread3, NULL);

    sem_destroy(&agentSem);
    sem_destroy(&tobaccoSem);
    sem_destroy(&paperSem);
    sem_destroy(&matchesSem);

    return 0;
}
```