MINOR ASSIGNMENT-10
Process/Thread Synchronization
Practical Programming with C (CSE 3544)

Publish on: 07-12-2024 **Submission on:** 21-12-2024
Course Outcome: CO_5 **Program Outcome:** PO_3 **Learning Level:** L_5

Problem Statement:

Working with POSIX thread and process synchronization techniques.

Assignment Objectives:

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

Answer the followings:

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 <stdlib.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/wait.h>

int shrd = 0;
void P1() {
   int x;
   x = shrd;
   x = x + 1;
   sleep(1);
   shrd = x;
}
void P2() {
   int y;
   y = shrd;
   y = y - 1;
   sleep(1);
   shrd = y;
}
```

```
int main() {
   pid_t pid1, pid2;

   pid1 = fork();
   if (pid1 == 0) {
      P1();
      exit(0);
   } else {
      pid2 = fork();
      if (pid2 == 0) {
         P2();
         exit(0);
      } else {
         wait(NULL);
         wait(NULL);
         printf("Final value of shrd: %d\n", shrd);
      }
   }

   return 0;
}
```

2. 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 <stdlib.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/wait.h>

int shrd = 0;

void P1() {
   int x;
   x = shrd;
   x = x + 1;
   sleep(1);
   shrd = x;
}
void P2() {
   int y;
   y = shrd;
   y = y - 1;
   sleep(1);
   shrd = y;
}
int main() {
   pid_t pid1, pid2;

   pid1 = fork();
   if (pid1 == 0) {
      P1();
      exit(0);
   } else {
      pid2 = fork();
      if (pid2 == 0) {
         P2();
         exit(0);
      } else {
         wait(NULL);
         wait(NULL);
         printf("Final value of shrd: %d\n", shrd);
      }
   }

   return 0;
}
```

3. 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

a)S=1, Q=1
Initial State: Both processes can start because S and Q are both non-zero.
Behavior:
Process 1 executes wait(&S) (decrements S to 0) and a.
Process 1 then signals Q (increments Q to 1), allowing Process 2 to proceed.
Process 2 executes wait(&Q) (decrements Q to 0) and b.
Process 2 then signals S (increments S to 1), allowing Process 1 to proceed.
Result: Both processes alternate execution without deadlock.

b)S=1, Q=0
Initial State: Process 1 can start because S is non-zero, but Process 2 cannot start because Q is zero.
Behavior:
Process 1 executes wait(&S) (decrements S to 0) and a.
Process 1 then signals Q (increments Q to 1), allowing Process 2 to proceed.
Process 2 executes wait(&Q) (decrements Q to 0) and b.
Process 2 then signals S (increments S to 1), allowing Process 1 to proceed.
Result: Both processes alternate execution without deadlock.

c)S=0, Q=1
Initial State: Process 2 can start because Q is non-zero, but Process 1 cannot start because S is zero.
Behavior:
Process 2 executes wait(&Q) (decrements Q to 0) and b.
Process 2 then signals S (increments S to 1), allowing Process 1 to proceed.
Process 1 executes wait(&S) (decrements S to 0) and a.
Process 1 then signals Q (increments Q to 1), allowing Process 2 to proceed.
Result: Both processes alternate execution without deadlock.

d)S=0, Q=0
Initial State: Both processes are blocked initially because both S and Q are zero.
Behavior:
Neither process can proceed since they are both waiting for a non-zero semaphore.
Result: Deadlock occurs immediately.

e)S=8, Q=0
Initial State: Process 1 can start because S is non-zero, but Process 2 cannot start because Q is zero.
Behavior:
Process 1 can execute 8 times consecutively because S starts at 8.
Each time Process 1 executes, it decrements S and increments Q.
After 8 iterations, S becomes 0, and Q becomes 8.
Process 2 can then execute 8 times consecutively, alternating between executing wait(&Q) (decrementing Q) and signaling S (incrementing S).

4. 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 <stdlib.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <semaphore.h>
#include <pthread.h>

sem_t S, Q;

void* process1(void* arg) {
  while (1) {
    sem_wait(&Q);
    sem_wait(&S);
    printf("Process 1 executing a\n");
    sem_post(&S);
    sem_post(&Q);
    sleep(1);
  }
}

void* process2(void* arg) {
  while (1) {
    sem_wait(&S);
    sem_wait(&Q);
    printf("Process 2 executing b\n");
    sem_post(&Q);
    sem_post(&S);
    sleep(1);
  }
}

int main() {
  pthread_t p1, p2;

  sem_init(&S, 0, 1);
  sem_init(&Q, 0, 1);

  pthread_create(&p1, NULL, process1, NULL);
  pthread_create(&p2, NULL, process2, NULL);

  pthread_join(p1, NULL);
  pthread_join(p2, NULL);

  sem_destroy(&S);
  sem_destroy(&Q);

  return 0;
}
```

5. 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 synchronization mechanism(s).

Process/Thread -1

```
for i = 1 to ?
:
:
display("X");
display("X"); :
:
```

Process/Thread -2

```
for i = 1 to ?
:
:
display("Y");
:
:
```

Process/Thread -3

```
for i = 1 to ?
:
:
display("Z");
display("Z"); :
:
```

Code here

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

sem_t semX, semY, semZ;

void* printX(void* arg) {
  for (int i = 0; i < 5; i++) {
    sem_wait(&semX);
    printf("X");
    printf("X");
    sem_post(&semY);
  }
  return NULL;
}

void* printY(void* arg) {
  for (int i = 0; i < 5; i++) {
    sem_wait(&semY);
    printf("Y");
    sem_post(&semZ); /
  }
  return NULL;
}

void* printZ(void* arg) {
  for (int i = 0; i < 5; i++) {
    sem_wait(&semZ);
    printf("Z");
    printf("Z");
    sem_post(&semX);
  }
  return NULL;
}

int main() {
  pthread_t tidX, tidY, tidZ;

  sem_init(&semX, 0, 1); // Start with X
  sem_init(&semY, 0, 0);
  sem_init(&semZ, 0, 0);

  pthread_create(&tidX, NULL, printX,
NULL);
  pthread_create(&tidY, NULL, printY,
NULL);
  pthread_create(&tidZ, NULL, printZ,
NULL);

  pthread_join(tidX, NULL);
  pthread_join(tidY, NULL);
  pthread_join(tidZ, NULL);

  sem_destroy(&semX);
  sem_destroy(&semY);
  sem_destroy(&semZ);

  printf("\n");
  return 0;
}
```

6. **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

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/wait.h>

void P1() {
   fprintf(stderr, "Coronavirus ");
}

void P2() {
   fprintf(stderr, "WHO: ");
}

void P3() {
   fprintf(stderr, "COVID-19 ");
}

void P4() {
   fprintf(stderr, "disease ");
}

void P5() {
   fprintf(stderr, "pandemic\n");
}
```

```
int main() {
   pid_t pid1, pid2, pid3, pid4, pid5;

   pid1 = fork();
   if (pid1 == 0) {
      P1();
      exit(0);
   } else {
      pid2 = fork();
      if (pid2 == 0) {
         P2();
         exit(0);
      } else {
         pid3 = fork();
         if (pid3 == 0) {
            P3();
            exit(0);
         } else {
            pid4 = fork();
            if (pid4 == 0) {
               P4();
               exit(0);
            } else {
               pid5 = fork();
               if (pid5 == 0) {
                  P5();
                  exit(0);
               }
            }
         }
      }
   }

   for (int i = 0; i < 5; i++) {
      wait(NULL);
   }

   return 0;
}
```

- ***OPTIONAL*** 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 empty;
sem_t full;
pthread_mutex_t mutex;

void* producer(void* arg) {
  int item;
  while (1) {
    item = rand() % 100;
    sem_wait(&empty);
    pthread_mutex_lock(&mutex);

    // Critical section
    buffer[in] = item;
    printf("Producer produced: %d
\n", item);
    in = (in + 1) % BUFFER_SIZE;

    pthread_mutex_unlock(&mutex);
    sem_post(&full);
    sleep(1);
  }
}
void* consumer(void* arg) {
  int item;
  while (1) {
    sem_wait(&full);
    pthread_mutex_lock(&mutex);

    // Critical section
    item = buffer[out];
    printf("Consumer consumed: %d
\n", item);
    out = (out + 1) % BUFFER_SIZE;

    pthread_mutex_unlock(&mutex);
    sem_post(&empty);
    sleep(1);
  }
}

int main() {
  pthread_t tid_producer, tid_consumer;

  sem_init(&empty, 0, BUFFER_SIZE);
  sem_init(&full, 0, 0);
  pthread_mutex_init(&mutex, NULL);

  pthread_create(&tid_producer, NULL, producer,
NULL);
  pthread_create(&tid_consumer, NULL, consumer,
NULL);

  pthread_join(tid_producer, NULL);
  pthread_join(tid_consumer, NULL);

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

  return 0;
}
```

- ***OPTIONAL* 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 waitingChairs;
sem_t barberChair;
sem_t barberSleep;
sem_t customerReady;

void* barber(void* arg) {
  while (1) {
    sem_wait(&customerReady);
    sem_wait(&waitingChairs);

    sem_post(&barberSleep);

    sem_wait(&barberChair);
    printf("Barber is cutting hair\n");
    sleep(rand() % 3 + 1);

    sem_post(&barberChair);
  }
}

void* customer(void* arg) {
  if (sem_trywait(&waitingChairs) == 0) {
    printf("Customer is waiting\n");
    sem_post(&customerReady);
    sem_wait(&barberSleep);

    sem_post(&waitingChairs);
    sem_wait(&barberChair);
    printf("Customer is getting a haircut\n");

    sem_post(&barberChair);
  } else {
    printf("No waiting chairs available. Customer is leaving.\n");
  }
}
```

Code here

```
int main() {
  pthread_t barberThread, customerThreads[10];

  sem_init(&waitingChairs, 0, CHAIRS);
  sem_init(&barberChair, 0, 1);
  sem_init(&barberSleep, 0, 0);
  sem_init(&customerReady, 0, 0);

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

  for (int i = 0; i < 10; i++) {
    sleep(rand() % 5);
    pthread_create(&customerThreads[i], NULL, customer, NULL);
  }

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

  pthread_join(barberThread, NULL);

  sem_destroy(&waitingChairs);
  sem_destroy(&barberChair);
  sem_destroy(&barberSleep);
  sem_destroy(&customerReady);

  return 0;
}
```

- ***OPTIONAL* 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 <stdlib.h>
#include <pthread.h>
#include <semaphore.h>
#include <unistd.h>

sem_t agent_sem;
sem_t tobacco_sem;
sem_t paper_sem;
sem_t matches_sem;
sem_t smoker_sem;

pthread_mutex_t table_mutex;

void* agent(void* arg) {
   while (1) {
      sem_wait(&agent_sem);
      pthread_mutex_lock(&table_mutex);

      int choice = rand() % 3;
      if (choice == 0) {
         printf("Agent puts paper and matches on the table.\n");
         sem_post(&tobacco_sem);
      } else if (choice == 1) {
         printf("Agent puts tobacco and matches on the table.\n");
         sem_post(&paper_sem);
      } else if (choice == 2) {
         printf("Agent puts tobacco and paper on the table.\n");
         sem_post(&matches_sem);
      }
      pthread_mutex_unlock(&table_mutex);
      sem_post(&smoker_sem);
      sleep(1);
   }
   return NULL;
}
```

Code here

```
void* smoker(void* arg) {
    char* item = (char*)arg;
    sem_t* my_sem;

    if (strcmp(item, "tobacco") == 0)
        my_sem = &tobacco_sem;
    else if (strcmp(item, "paper") == 0)
        my_sem = &paper_sem;
    else
        my_sem = &matches_sem;

    while (1) {
        sem_wait(my_sem);
        pthread_mutex_lock(&table_mutex);
        printf("Smoker with %s makes a cigarette.\n", item);
        pthread_mutex_unlock(&table_mutex);
        sem_post(&agent_sem);
        sleep(1);
    }
    return NULL;
}

int main() {
    srand(time(0));

    pthread_t agent_thread;
    pthread_t smokers[3];

    sem_init(&agent_sem, 0, 1);
    sem_init(&tobacco_sem, 0, 0);
    sem_init(&paper_sem, 0, 0);
    sem_init(&matches_sem, 0, 0);
    sem_init(&smoker_sem, 0, 0);

    pthread_mutex_init(&table_mutex, NULL);

    pthread_create(&agent_thread, NULL, agent, NULL);
    pthread_create(&smokers[0], NULL, smoker, "paper");
    pthread_create(&smokers[1], NULL, smoker, "tobacco");
    pthread_create(&smokers[2], NULL, smoker, "matches");

    pthread_join(agent_thread, NULL);
    for (int i = 0; i < 3; i++) {
        pthread_join(smokers[i], NULL);
    }

    sem_destroy(&agent_sem);
    sem_destroy(&tobacco_sem);
    sem_destroy(&paper_sem);
    sem_destroy(&matches_sem);
    sem_destroy(&smoker_sem);
    pthread_mutex_destroy(&table_mutex);

    return 0;
}
```