Name : Vikrant Suresh Tripathi
Roll No : 2103141
Subject : Operating System LAB 6 Assignment

Assignment

1) Consider the following problem: A program is to be written to print all numbers between 1 and 1000 (inclusive) that are not (evenly) divisible by either 2 or 3. This problem is to be solved using three processes (P0, P1, P2) and two one-integer buffers (B0 and B1) as follows:

a. P0 is to generate the integers from 1 to 1000, and place them in B0 one at a time. After placing 1000 in the buffer, P0 places the sentinel 0 in the buffer, and terminates.

b. P1 is to read successive integers from B0. If a value is not divisible by 2, the value is placed in B1. If the value is positive and divisible by 2, it is ignored. If the value is 0, 0 is placed in B1, and P1 terminates.

c. P2 is to read successive integers from B1. If a value is not divisible by 3, it is printed. If the value is positive and divisible by 3, it is ignored. If the value is 0, P2 terminates.

Write a program to implement P0, P1, and P2 as separate processes and B0 and B1 as separate pieces of shared memory – each the size of just one integer. Use semaphores to coordinate processing. Access to B0 should be independent of access to B1; for example, P0 could be writing into B0 whileeither P1 was writing into B1 or P2 was reading.

Output :

```
(base) vikrant@Vikrants-MacBook-Pro LAB6 % gcc Assign_Q1.c
(base) vikrant@Vikrants-MacBook-Pro LAB6 % ./a.out
1
5
7
11
13
17
19
23
25
29
31
35
37
41
43
47
49
53
55
59
61
65
67
71
73
77
79
83
85
89
91
95
97
101
103
107
109
```

.

.

.

```
887
889
893
895
899
901
905
907
911
913
917
919
923
925
929
931
935
937
941
943
947
949
953
955
959
961
965
967
971
973
977
979
983
985
989
991
995
997
Total count of numbers not divisible by 2 and 3: 333
```

Initialization: The code begins by including necessary headers and defining functions for creating and working with semaphores (sem_create, sem_init, P, and V). These functions help manage synchronization between processes.

Semaphore Creation: The sem_create function creates a set of semaphores that will be used for coordinating the processes. In this case, there are four semaphores created: one for each of the three processes (P0, P1, P2) and an additional one for synchronization purposes.

Semaphore Initialization: The sem_init function initializes each semaphore with an initial value. This step sets up the initial state of the semaphores, such as setting them to 1 for P0 and P2 to ensure they start in the correct order.

Shared Memory Allocation: Two shared memory segments, B0 and B1, are created using the mmap function. These shared memory segments act as buffers for passing data between the processes. Each shared memory segment is of integer type and is initially set to 0.

Process Creation (P0, P1, and P2):

- *P0 Process: A child process (P0) is forked, and it executes code to generate numbers from 1 to 1000. It places these numbers in B0, one at a time. After placing 1000, it puts a sentinel value (0) into B0 to indicate the end of the numbers. This process keeps repeating until all numbers are generated.*
- *P1 Process: Another child process (P1) is forked, which continuously checks B0 for new values. If a value is not divisible by 2, it places the*

value into B1. If the value is 0 (the sentinel), it places 0 into B1 and exits. This process filters out even numbers.

- *P2 Process: A third child process (P2) is forked, which continuously checks B1 for new values. If a value is not divisible by 3, it prints the value to the console. If the value is 0 (the sentinel), it prints the total count of numbers not divisible by 2 and 3 and exits. This process also counts the numbers that meet the criteria.*
-

Wait for Child Processes: The main process waits for all three child processes to finish using a loop and the wait system call. This ensures that all processes complete their tasks before proceeding.

Semaphore Cleanup: After all processes have finished, the code cleans up the semaphores using semctl to release the system resources associated with them. Total Count: P2 prints the total count of numbers that meet the criteria before it exits. This count represents the numbers between 1 and 1000 that are not divisible by 2 or 3.

In summary, this code demonstrates how to use semaphores and shared memory to coordinate multiple processes (P0, P1, and P2) to solve a problem concurrently. Each process has a specific role in generating, filtering, and counting numbers that meet the specified criteria.

2) Write a C program to implement the following game. The parent program P first creates two pipes, and then spawns two child processes C and D. One of the two pipes is meant for communications between P and C, and the other for communications between P and D. Now, a loop runs as follows. In each iteration (also called round), P first randomly chooses one of the two flags: MIN and MAX (the choice randomly varies from one iteration to another). Each of the two child processes C and D generates a random positive integer and sends that to P via its pipe. P reads the two integers; let these be c and d. If P has chosen MIN, then the child who sent the smaller of c and d gets one point. If P has chosen MAX, then the sender of the larger of c and d gets one point. If c= d, then this round is ignored.
The child process who first obtains ten points wins the game. When the game ends, P sends a user- defined signal to both C and D, and the child processes exit after handling the signal (in order to know who was the winner). After C and D exit, the parent process P exits. During each iteration of the game, P should print appropriate messages (like P's choice of the flag, the integers received from C and D, which child gets the point, the current scores of C and D) in order to let the user know how the game is going on. Name your program childs game.c .

Output

```
(base) vikrant@Vikrants-MacBook-Pro LAB6 % gcc Assign_Q2.c
(base) vikrant@Vikrants-MacBook-Pro LAB6 % ./a.out
Round number: 1
Integer received from Child 1: 65
Integer received from Child 2: 30
Parent's choice of flag: MIN
Child 2 gets a point
Updated scores: Child 1 = 0, Child 2 = 1

Round number: 2
Integer received from Child 1: 43
Integer received from Child 2: 98
Parent's choice of flag: MIN
Child 1 gets a point
Updated scores: Child 1 = 1, Child 2 = 1

Round number: 3
Integer received from Child 1: 37
Integer received from Child 2: 60
Parent's choice of flag: MIN
Child 1 gets a point
Updated scores: Child 1 = 2, Child 2 = 1

Round number: 4
Integer received from Child 1: 39
Integer received from Child 2: 43
Parent's choice of flag: MAX
Child 2 gets a point
Updated scores: Child 1 = 2, Child 2 = 2

Round number: 5
Integer received from Child 1: 27
Integer received from Child 2: 18
Parent's choice of flag: MAX
Child 1 gets a point
Updated scores: Child 1 = 3, Child 2 = 2

Round number: 6
Integer received from Child 1: 15
```

```
Child 2 gets a point
Updated scores: Child 1 = 8, Child 2 = 6

Round number: 15
Integer received from Child 1: 56
Integer received from Child 2: 68
Parent's choice of flag: MAX
Child 2 gets a point
Updated scores: Child 1 = 8, Child 2 = 7

Round number: 16
Integer received from Child 1: 77
Integer received from Child 2: 56
Parent's choice of flag: MAX
Child 1 gets a point
Updated scores: Child 1 = 9, Child 2 = 7

Round number: 17
Integer received from Child 1: 9
Integer received from Child 2: 15
Parent's choice of flag: MAX
Child 2 gets a point
Updated scores: Child 1 = 9, Child 2 = 8

Round number: 18
Integer received from Child 1: 19
Integer received from Child 2: 74
Parent's choice of flag: MIN
Child 1 gets a point
Updated scores: Child 1 = 10, Child 2 = 8

Child 1: I am the winner
Child 2: Child 1 is the winner
Exiting from Child 1
Exiting from Child 2


Exiting from Parent
(base) vikrant@Vikrants-MacBook-Pro LAB6 %
```

This code simulates a simple game involving a parent process and two child processes (Child 1 and Child 2) using inter-process communication (IPC) through pipes and signals. Here's an explanation of how the code works:

Signal Handling: The code defines a custom signal handler handleTermination to handle the termination of the game. It listens for two signals, SIGUSR1 and SIGUSR2, which indicate the winner of the game.

Child Process Creation (Child 1 and Child 2): Two child processes, Child 1 and Child 2, are created using the fork function. Each child process will perform specific tasks in the game.

Pipe Creation: Two pipes, pipe1 and pipe2, are created for communication between the parent process and Child 1 and Child 2, respectively. These pipes are used to send random numbers generated by child processes to the parent.

Child Process Logic:

- *Child 1 Logic: Child 1 generates random numbers and writes them to pipe1. It continues generating numbers until a termination signal is received (terminationFlag is set to 1 or 2). After termination, it prints a message indicating if it is the winner and exits.*
- *Child 2 Logic: Child 2 operates similarly to Child 1 but communicates through pipe2. It also generates random numbers and writes them to the pipe until a termination signal is received. It prints a message indicating if it is the winner and exits.*

Parent Process Logic:

- *Parent Logic: The parent process is responsible for coordinating the game and determining the winner. It closes the write ends of both pipes (pipe1 and pipe2) because it doesn't write to them. It also sets up a random number generator for making choices during the game.*
- *Game Rounds: The game consists of multiple rounds. In each round, the parent reads one random number from each child through the pipes. It also randomly chooses a "flag" (0 for MAX, 1 for MIN) that determines the winning condition for that round. The parent compares the numbers received from the children based on the flag and updates the scores accordingly.*
- *Winning Condition: The game continues until one of the children reaches the winning condition (WINNING_CONDITION). When this happens, the parent breaks out of the game loop.*

- *Signaling Winner: After determining the winner, the parent sends signals to both Child 1 and Child 2 to let them know the outcome of the game. If Child 1 wins, it sends SIGUSR1; if Child 2 wins, it sends SIGUSR2.*
- *Waiting for Child Processes: The parent uses waitpid to wait for both child processes to terminate and retrieve their exit statuses. After that, it prints an exit message and terminates itself.*

In summary, this code demonstrates a simple game where two child processes generate random numbers, and the parent process determines the winner based on specific rules. It utilizes pipes for communication and custom signals to announce the winner to the children.