**Operating Systems Lab Monitor’s Task**

**Name: Ahmed Kasteer**

**Roll Number: 20F-0336**

**Section: 4D**

**Dinning Philosopher Problem**

#include <pthread.h>

#include <semaphore.h>

#include <stdio.h>

#include<unistd.h>

#define N 5

#define THINKING 2

#define HUNGRY 1

#define EATING 0

#define LEFT (phnum + 4) % N

#define RIGHT (phnum + 1) % N

int state[N];

int phil[N] = { 0, 1, 2, 3, 4 };

sem\_t mutex;

sem\_t S[N];

void test(int phnum)

{

if (state[phnum] == HUNGRY

&& state[LEFT] != EATING

&& state[RIGHT] != EATING) {

// state that eating

state[phnum] = EATING;

sleep(2);

printf("Philosopher %d takes fork %d and %d\n",

phnum + 1, LEFT + 1, phnum + 1);

printf("Philosopher %d is Eating\n", phnum + 1);

// sem\_post(&S[phnum]) has no effect

// during takefork

// used to wake up hungry philosophers

// during putfork

sem\_post(&S[phnum]);

}

}

// take up chopsticks

void take\_fork(int phnum)

{

sem\_wait(&mutex);

// state that hungry

state[phnum] = HUNGRY;

printf("Philosopher %d is Hungry\n", phnum + 1);

// eat if neighbours are not eating

test(phnum);

sem\_post(&mutex);

// if unable to eat wait to be signalled

sem\_wait(&S[phnum]);

sleep(1);

}

// put down chopsticks

void put\_fork(int phnum)

{

sem\_wait(&mutex);

// state that thinking

state[phnum] = THINKING;

printf("Philosopher %d putting fork %d and %d down\n",

phnum + 1, LEFT + 1, phnum + 1);

printf("Philosopher %d is thinking\n", phnum + 1);

test(LEFT);

test(RIGHT);

sem\_post(&mutex);

}

void\* philosopher(void\* num)

{

while (1) {

int\* i = num;

sleep(1);

take\_fork(\*i);

sleep(0);

put\_fork(\*i);

}

}

int main()

{

int i;

pthread\_t thread\_id[N];

// initialize the semaphores

sem\_init(&mutex, 0, 1);

for (i = 0; i < N; i++)

sem\_init(&S[i], 0, 0);

for (i = 0; i < N; i++) {

// create philosopher processes

pthread\_create(&thread\_id[i], NULL,

philosopher, &phil[i]);

printf("Philosopher %d is thinking\n", i + 1);

}

for (i = 0; i < N; i++)

pthread\_join(thread\_id[i], NULL);

}

Text

Description automatically generated

**Reader-Writer Problem**

#include <pthread.h>

#include <semaphore.h>

#include <stdio.h>

/\*

This program provides a possible solution for first readers writers problem using mutex and semaphore.

I have used 10 readers and 5 producers to demonstrate the solution. You can always play with these values.

\*/

sem\_t wrt;

pthread\_mutex\_t mutex;

int cnt = 1;

int numreader = 0;

void \*writer(void \*wno)

{

sem\_wait(&wrt);

cnt = cnt\*2;

printf("Writer %d modified cnt to %d\n",(\*((int \*)wno)),cnt);

sem\_post(&wrt);

}

void \*reader(void \*rno)

{

// Reader acquire the lock before modifying numreader

pthread\_mutex\_lock(&mutex);

numreader++;

if(numreader == 1) {

sem\_wait(&wrt); // If this id the first reader, then it will block the writer

}

pthread\_mutex\_unlock(&mutex);

// Reading Section

printf("Reader %d: read cnt as %d\n",\*((int \*)rno),cnt);

// Reader acquire the lock before modifying numreader

pthread\_mutex\_lock(&mutex);

numreader--;

if(numreader == 0) {

sem\_post(&wrt); // If this is the last reader, it will wake up the writer.

}

pthread\_mutex\_unlock(&mutex);

}

int main()

{

pthread\_t read[10],write[5];

pthread\_mutex\_init(&mutex, NULL);

sem\_init(&wrt,0,1);

int a[10] = {1,2,3,4,5,6,7,8,9,10}; //Just used for numbering the producer and consumer

for(int i = 0; i < 10; i++) {

pthread\_create(&read[i], NULL, (void \*)reader, (void \*)&a[i]);

}

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

pthread\_create(&write[i], NULL, (void \*)writer, (void \*)&a[i]);

}

for(int i = 0; i < 10; i++) {

pthread\_join(read[i], NULL);

}

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

pthread\_join(write[i], NULL);

}

pthread\_mutex\_destroy(&mutex);

sem\_destroy(&wrt);

return 0;

}

A screenshot of a computer

Description automatically generated with medium confidence

**Producer Consumer Problem**

// C program for the above approach

#include <stdio.h>

#include <stdlib.h>

// Initialize a mutex to 1

int mutex = 1;

// Number of full slots as 0

int full = 0;

// Number of empty slots as size

// of buffer

int empty = 10, x = 0;

// Function to produce an item and

// add it to the buffer

void producer()

{

// Decrease mutex value by 1

--mutex;

// Increase the number of full

// slots by 1

++full;

// Decrease the number of empty

// slots by 1

--empty;

// Item produced

x++;

printf("\nProducer produces"

"item %d",

x);

// Increase mutex value by 1

++mutex;

}

// Function to consume an item and

// remove it from buffer

void consumer()

{

// Decrease mutex value by 1

--mutex;

// Decrease the number of full

// slots by 1

--full;

// Increase the number of empty

// slots by 1

++empty;

printf("\nConsumer consumes "

"item %d",

x);

x--;

// Increase mutex value by 1

++mutex;

}

// Driver Code

int main()

{

int n, i;

printf("\n1. Press 1 for Producer"

"\n2. Press 2 for Consumer"

"\n3. Press 3 for Exit");

// Using '#pragma omp parallel for'

// can give wrong value due to

// synchronization issues.

// 'critical' specifies that code is

// executed by only one thread at a

// time i.e., only one thread enters

// the critical section at a given time

#pragma omp critical

for (i = 1; i > 0; i++) {

printf("\nEnter your choice:");

scanf("%d", &n);

// Switch Cases

switch (n) {

case 1:

// If mutex is 1 and empty

// is non-zero, then it is

// possible to produce

if ((mutex == 1)

&& (empty != 0)) {

producer();

}

// Otherwise, print buffer

// is full

else {

printf("Buffer is full!");

}

break;

case 2:

// If mutex is 1 and full

// is non-zero, then it is

// possible to consume

if ((mutex == 1)

&& (full != 0)) {

consumer();

}

// Otherwise, print Buffer

// is empty

else {

printf("Buffer is empty!");

}

break;

// Exit Condition

case 3:

exit(0);

break;

}

}

}

Text

Description automatically generated

**Peterson Problem**

// C program to implement Peterson’s Algorithm

// for producer-consumer problem.

#include <stdio.h>

#include <stdlib.h>

#include <unistd.h>

#include <time.h>

#include <sys/types.h>

#include <sys/ipc.h>

#include <sys/shm.h>

#include <stdbool.h>

#define \_BSD\_SOURCE

#include <sys/time.h>

#include <stdio.h>

#define BSIZE 8 // Buffer size

#define PWT 2 // Producer wait time limit

#define CWT 10 // Consumer wait time limit

#define RT 10 // Program run-time in seconds

int shmid1, shmid2, shmid3, shmid4;

key\_t k1 = 5491, k2 = 5812, k3 = 4327, k4 = 3213;

bool\* SHM1;

int\* SHM2;

int\* SHM3;

int myrand(int n) // Returns a random number between 1 and n

{

time\_t t;

srand((unsigned)time(&t));

return (rand() % n + 1);

}

int main()

{

shmid1 = shmget(k1, sizeof(bool) \* 2, IPC\_CREAT | 0660); // flag

shmid2 = shmget(k2, sizeof(int) \* 1, IPC\_CREAT | 0660); // turn

shmid3 = shmget(k3, sizeof(int) \* BSIZE, IPC\_CREAT | 0660); // buffer

shmid4 = shmget(k4, sizeof(int) \* 1, IPC\_CREAT | 0660); // time stamp

if (shmid1 < 0 || shmid2 < 0 || shmid3 < 0 || shmid4 < 0) {

perror("Main shmget error: ");

exit(1);

}

SHM3 = (int\*)shmat(shmid3, NULL, 0);

int ix = 0;

while (ix < BSIZE) // Initializing buffer

SHM3[ix++] = 0;

struct timeval t;

time\_t t1, t2;

gettimeofday(&t, NULL);

t1 = t.tv\_sec;

int\* state = (int\*)shmat(shmid4, NULL, 0);

\*state = 1;

int wait\_time;

int i = 0; // Consumer

int j = 1; // Producer

if (fork() == 0) // Producer code

{

SHM1 = (bool\*)shmat(shmid1, NULL, 0);

SHM2 = (int\*)shmat(shmid2, NULL, 0);

SHM3 = (int\*)shmat(shmid3, NULL, 0);

if (SHM1 == (bool\*)-1 || SHM2 == (int\*)-1 || SHM3 == (int\*)-1) {

perror("Producer shmat error: ");

exit(1);

}

bool\* flag = SHM1;

int\* turn = SHM2;

int\* buf = SHM3;

int index = 0;

while (\*state == 1) {

flag[j] = true;

printf("Producer is ready now.\n\n");

\*turn = i;

while (flag[i] == true && \*turn == i)

;

// Critical Section Begin

index = 0;

while (index < BSIZE) {

if (buf[index] == 0) {

int tempo = myrand(BSIZE \* 3);

printf("Job %d has been produced\n", tempo);

buf[index] = tempo;

break;

}

index++;

}

if (index == BSIZE)

printf("Buffer is full, nothing can be produced!!!\n");

printf("Buffer: ");

index = 0;

while (index < BSIZE)

printf("%d ", buf[index++]);

printf("\n");

// Critical Section End

flag[j] = false;

if (\*state == 0)

break;

wait\_time = myrand(PWT);

printf("Producer will wait for %d seconds\n\n", wait\_time);

sleep(wait\_time);

}

exit(0);

}

if (fork() == 0) // Consumer code

{

SHM1 = (bool\*)shmat(shmid1, NULL, 0);

SHM2 = (int\*)shmat(shmid2, NULL, 0);

SHM3 = (int\*)shmat(shmid3, NULL, 0);

if (SHM1 == (bool\*)-1 || SHM2 == (int\*)-1 || SHM3 == (int\*)-1) {

perror("Consumer shmat error:");

exit(1);

}

bool\* flag = SHM1;

int\* turn = SHM2;

int\* buf = SHM3;

int index = 0;

flag[i] = false;

sleep(5);

while (\*state == 1) {

flag[i] = true;

printf("Consumer is ready now.\n\n");

\*turn = j;

while (flag[j] == true && \*turn == j)

;

// Critical Section Begin

if (buf[0] != 0) {

printf("Job %d has been consumed\n", buf[0]);

buf[0] = 0;

index = 1;

while (index < BSIZE) // Shifting remaining jobs forward

{

buf[index - 1] = buf[index];

index++;

}

buf[index - 1] = 0;

} else

printf("Buffer is empty, nothing can be consumed!!!\n");

printf("Buffer: ");

index = 0;

while (index < BSIZE)

printf("%d ", buf[index++]);

printf("\n");

// Critical Section End

flag[i] = false;

if (\*state == 0)

break;

wait\_time = myrand(CWT);

printf("Consumer will sleep for %d seconds\n\n", wait\_time);

sleep(wait\_time);

}

exit(0);

}

// Parent process will now for RT seconds before causing child to terminate

while (1) {

gettimeofday(&t, NULL);

t2 = t.tv\_sec;

if (t2 - t1 > RT) // Program will exit after RT seconds

{

\*state = 0;

break;

}

}

// Waiting for both processes to exit

printf("The clock ran out.\n");

return 0;

}

Text

Description automatically generated