190031210 Home Assignment - CO-IV P N V KRISHNA TEJA

Operating Systems Design 19CS2106A

Mode of submission: Post Handwritten scanned documents in LMS Submission date: on or before 10th November 2020

1. Write a UNIX system program to Solve Producer Consumer problem using POSIX semaphores.

Three conditions must be maintained by the code when the shared buffer is considered as a circular buffer:

1. The consumer cannot try to remove an item from the buffer when the buffer is empty.
2. The producer cannot try to place an item into the buffer when the buffer is full.
3. Shared variables may describe the current state of the buffer (indexes, counts, linked list pointers, etc.), so all buffer manipulations by the producer and consumer must be protected to avoid any race conditions.

Your solution using semaphores should demonstrates three different types of semaphores:

1. A binary semaphore named mutex protects the critical regions: inserting a data item into the buffer (for the producer) and removing a data item from the buffer (for the consumer). A binary semaphore that is used as a mutex is initialized to 1. (Obviously we could use a real mutex for this, instead of a binary semaphore.)
2. A counting semaphore named nempty counts the number of empty slots in the buffer. This semaphore is initialized to the number of slots in the buffer (NBUFF).
3. A counting semaphore named nstored counts the number of filled slots in the buffer. This semaphore is initialized to 0, since the buffer is initially empty

OSD

Home Assignment - 4

190031210
PNV Krishna Teja

1. Aim : write UNIX System program to solve producer consumer using posix semaphores

```
#include <stdio.h>
#include <semaphore.h>
#include <sys/types.h>
#include <fcntl.h>
#define BUFFER_SIZE 10
#define CONSUMER_SLEEP_SEC 3
#define PRODUCER_SLEEP_SEC 1
#define KEY 1010
typedef struct
{ int buff [BUFFER_SIZE]
  sem_t mutex, empty, full;
} MEM;
MEM *memory( )
{ key_t key = KEY;
  int shmid;
}
void init()
{ MEM *M = memory();
  sem_init (&M -> mutex, 1, 1);
  sem_init (&M -> empty, 1, BUFFER_SIZE);
  sem_init (&M -> full, 1, 0);

}
```

producer.c

```
#include "problem.h"
void producer()
{
    int i o, n;
    MEM *s - memory();
    while(1)
    {
        i++;
        sem-wait (&s->empty);
        sem-getvalue (&s -> full, &n);
        (s->buff) (n] = 1;
        printf(" [PRODUCER] placed item[%d]\n,i);

    }
}
main()
{ init();
  producer();
}
```

consumer.c :-

```
#include "problem.h"
void consumer()
{ int n;
  MEM *s - memory ())
  while (1)
  { sem-wait (&s -> full),
    sem-wait (&s -> mutex);
    sem-post (&s -> mutex);
    sem-post (&s -> empty);
    sleep (CONSUMER-SLEEP-SEC);
  }
}
```

```
main ( )
{  consumer ( );

}
```

2. Considering a system with five processes P0 through P4 and three resources types A, B, C. Resource type A has 10 instances, B has 5 instances and type C has 7 instances. Suppose at time t0 following snapshot of the system has been taken:

Process	Allocation	Max	Available
	A B C	A B C	A B C
P_0	0 1 0	7 5 3	3 3 2
P_1	2 0 0	3 2 2	
P_2	3 0 2	9 0 2	
P_3	2 1 1	2 2 2	
P_4	0 0 2	4 3 3	

i. What will be the content of the Need matrix?

2. (i) Need [i,j] Max[i,j] - Allocation (i,j)

So the content of need matrix is

	Need		
	A	B	C
P_0	7	4	3
P_1	1	2	2
P_2	6	0	0
P_3	0	1	1
P_4	4	3	1

ii. Is the system in safe state? If Yes, then what is the safe sequence?

(ii) yes, Applying safety algorithm on given system

Step-1 :- $m=5$, $n=5$

work = available

work = | 3 | 3 | 2 |

	0	1	2	3	4
finish =	false	false	false	false	false

step-2 :-

for $i=0$, $Need_0$: 7, 4, 3

finish [0] is false

7, 4, 3   3, 3, 2

Need 0 > work ×

So, P_0 must wait

But need ≤ work

for $i=0$, $Need_0 = 7,4,3$

finish [0] is false

$7,4,3 \quad 3,4,5$
$Need_0 < work$

so, P_0 must be keep in safe sequence

For $i=1$ $Need_1 = 1,2,2$

finish [1] is false.

$1,2,2 \quad 3,3,2$
$Need_1 < work$

So, P_1 must be keep in safe sequence

For $i=2$ $Need_2 = 6,0,0$

finish [2] is false

$6,0,0 \quad 5,3,2$
$Need_2 > work$

so, P_2 must wait

For $i=2$ $Need_2 = 6,0,0$

finish (2) is false

$6,0,0 \quad 7,5,5$
$Need_2 < work$

so, P_2 must be in safe sequence

For $i=3$ $Need_3 = 0,1,1$

Finish (3) is false

$0,1,1 \quad 5,3,2$
$Need_3 < work$

so, P_3 must be in safe sequence

For $i=4$ $Need_4 = 4,3,1$

finish [4] = false

$4,3,1 \quad 7,4,3$
$Need_4 < work$

so, P_4 must be in safe sequence

Step - 3

work = work + allocation (7, 4, 5 + 0, 1, 0)

work =

A	B	C
7	4	5

Finish =

0	1	2	3	4
True	false	false	True	True

work = work + allocation (3, 3, 2 + 2, 0, 0)

work =

A	B	C
5	3	2

Finish =

0	1	2	3	4
false	True	false	false	false

work = work + Allocation (7, 5, 5 + 3, 0, 2)

work =

A	B	C
10	5	7

Finish =

0	1	2	3	4
True	True	True	True	True

work =

A	B	C
7	4	3

Finish =

0	1	2	3	4
False	True	False	True	False

step - 4 : finish [i] = true for $0 \le i < n$

Hence, the system is in safe state

∴ The safe seq is P_1, P_3, P_4, P_0, P_2

iii. What will happen if process P_1 requests one additional instance of resource type A and two instances of resource type C?

(iii) $Request_1 = \begin{array}{ccc} A & B & C \\ 1, & 0, & 2 \end{array}$

First we need to decide whether req is granted or not

Step-1

$\begin{array}{ccc} 1,0,2 & & 1,2,2 \\ Req\,1 & < & Need_1 \end{array}$

Step-2

$\begin{array}{ccc} 1,0,2 & & 3,3,2 \\ Req\,1 & < & Available \end{array}$

Step-3

Available = Available − req ;

Allocation = Alloc1 + req1 ;

Need1 = Need1 − req1 ;

Process	Allocation			Need			Available		
	A	B	C	A	B	C	A	B	C
P_0	0	1	0	7	4	3	2	3	0
P_1	3	0	2	0	2	0			
P_2	3	0	2	6	0	0			
P_3	2	1	1	0	1	1			
P_4	0	0	2	4	3	1			

By applying safety algorithm we get to know that the new system is safe. so we can grant the req for process P_1

3. Write a Program using pthreads to demonstrate deadlock.

3.
```
#include < pthread.h >
# include < stdio.h >
# include < stdlib.h >
pthread_mutex_t resource1, resource2;
int test = 0;
void * proc1 ( )
{
    printf ("This is proc1 using 1");
    pthread_mutex_lock ( resource 1 );
    usleep (200);
    printf ("\n P1 trying to get is 2");
    pthread_mutex_lock ( resource 2);
    test ++;
    printf (" \n proc", got rs2!!);
    pthread_mutex_unlock ( resource 2);
    pthread_mutex_unlock ( resource 1);
   return 0;
}
void * proc2 ( )
{
   pthread_mutex_lock (resource 2);
    unsleep (200);
   printf ("\n P2 trying to get 2");
     test --;
  printf (" \n proc2 trying to get rs1!!")
  return 0;
}
```

```
int main ()
{
    pthread_t   t1, t2 ;
  pthread_mutex_init (&resource1, NULL);
  pthread_mutex_init (&resource2, NULL);
  pthread_create (&1, NULL, proc1, NULL);
  pthread_create (&2, NULL, proc2, NULL);
  pthread_join (t1, NULL);
  pthread_join (t2, NULL);

}
```

4. Solve Readers-Writers Problem using counter and 2 semaphores.

4. → No reader will be kept waiting unless a writer has the object.

→ writing is performed as soon as possible

```
semaphore mutex = 1;
semaphore db = 1;

Reader()
{  while (true) {
      down (&mutex)
      reader_count += 1;
    if (reader_count == 1)
          down (&db);
          up(& mutex);
          readdb ();
     down (&mutex);
     reader_count -= 1 ;
```