WEEK 6 – LAB 6 - UE15CS305 – Introduction to Operating System (IOS) Laboratory Problem Statements with Solutions

Problem 1:

Write a program to implement classical inter process communication problem (Producer Consumer) using Semaphores.

Background:

Chapter 3 : Read Shared Memory Systems and the producer consumer problem.
Chapter 4 : Read the section on pthreads and how to create threads.
Chapter Read 6.5 and 6.6 to learn about semaphores.
Refer man pages for pthread, semaphores

Implement a main program that creates two threads : producer and consumer threads which executes producer and consumer functions respectively. The producer should produce an item and update the buffer. The consumer should consume an item and update the buffer. You need to use a semaphore (sem_t) and enclose the critical sections in both producer and consumer so that only one of them can update the buffer at a time. Also, consumer should wait if buffer is empty and producer should signal when the buffer has at least one item. You can assume it is unbounded buffer so that producer does not have to wait for buffer availability. Both the producer and consumer threads can be infinite loops and each should also randomly sleep to let the other proceed. The output should be the number of items in buffer along with the consumer/producer that is updating the buffer.

The structure of your program:

```
producer thread function()
        produce an item
        update the buffer

consumer thread function()
        if buffer is empty, wait for producer
        update the buffer
        :
main function()
        initialize the semaphores
        create consumer and producer threads

// interprocess communication
// gcc ipc.c -o ipc.exe –lpthread
```

Semaphores

Since all threads run in the same address space, they all have access to the same data and variables. If two threads simultaneously attempt to update a global counter variable, it is possible for their operations to interleave in such way that the global state is not correctly modified. Although such a case may only arise only one time out of thousands, a concurrent program needs to coordinate the activities of multiple threads using something more reliable that just depending on the fact that such interference is rare. The semaphore is designed for just this purpose.

A semaphore is somewhat like an integer variable, but is special in that its operations (increment and decrement) are guaranteed to be atomic—you cannot be halfway through incrementing the semaphore and be interrupted and waylaid by another thread trying to do the same thing. That means you can increment and decrement the semaphore from multiple threads without interference. By convention, when a semaphore is zero it is "locked" or "in use". Otherwise, positive values indicate that the semaphore is available. A semaphore will never have a negative value.

Semaphores are also specifically designed to support an efficient waiting mechanism. If a thread can't proceed until some change occurs, it is undesirable for that thread to be looping and repeatedly checking the state until it changes. In this case semaphore can be used to represent the right of a thread to proceed. A non-zero value means the thread should continue, zero means to hold off. When a thread attempts to decrement a unavailable semaphore (with a zero value), it efficiently waits until another thread increments the semaphore to signal the state change that will allow it to proceed.

Semaphores are usually provided as an ADT by a machine-specific package. As with any ADT, you should only manipulate the variables through the interface routines—in this case SemaphoreWait and SemaphoreSignal below. There is no single standard thread synchronization facility, but they all look and act pretty similarly.

SemaphoreWait(Semaphore s)
If a semaphore value is positive, decrement the value otherwise suspend the thread and block on that semaphore until it becomes positive. The thread package keeps track of the threads that are blocked on a particular semaphore. Many packages guarantee FIFO/queue behavior for the unblocking of threads to avoid starvation. Alternately the threads blocked on a semaphore may be stored as a set where the thread manager is free to choose any one. In that case, a thread could theoretically starve, but it's unlikely. 2 Historically, P is a synonym for SemaphoreWait. You see, P is the first letter in the word prolagen which is of course a Dutch word formed from the words proberen (to try) and verlagen (to decrease).

SemaphoreSignal(Semaphore s)
Increment the semaphore value, potentially awakening a suspended thread that is blocked on it. If multiple threads are waiting, it is not deterministic which will be chosen. Also there is no guarantee that any suspended thread will actually begin running immediately when awakened. The awakened thread may just be marked or queued for execution and will run at some later time. V historically is a synonym for SemaphoreSignal since, of course, verhogen

means "to increase" in Dutch.

No GetValue(Semaphore s) function

One special thing to note about semaphores is that there is no "SemaphoreValue" function in the interface. You cannot look at the value directly, you can only operate on the value through the increment and decrement operations of Signal and Wait. It isn't really useful to retrieve the value of the semaphore since as you receive the return value there is no guarantee it hasn't been changed in the meantime by another thread.

Semaphore use

In client code, a SemaphoreWait call is a sort of checkpoint. If the semaphore is available (i.e. has a positive value) a thread will decrement the value and breeze right through the call to SemaphoreWait. If the semaphore is not available, then the thread will efficiently block at the point of the SemaphoreWait until the semaphore is available. A call to SemaphoreWait is usually balanced by a call to SemaphoreSignal to release the semaphore for other threads.

Binary semaphores

A binary semaphore can only be 0 or 1. Binary semaphores are most often used to implement a lock that allows only a single thread into a critical section. The semaphore is initially given the value 1 and when a thread approaches the critical region, it waits on the semaphore to decrement the value and "take out" the lock, then signals the semaphore at the end of the critical region to release the lock. Any thread arriving at the critical region while the lock is in use will block when trying to decrement the semaphore, because it is already at 0. When the thread inside the critical region exits, it signals the semaphore and brings its value back up to 1. This allows the waiting thread to now take out the lock and enter the critical section. The result is that at most one thread can enter into the critical section and only after it leaves can another enter. This sort of locking strategy is often used to serialize code that accesses a shared global variable. 3 The main issues to watch with binary semaphores is ensuring they are initialized to the proper starting state when created and making sure each thread that locks the semaphore is careful to unlock it. You also want to try to keep the critical region as small as possible— only exactly those statements that need to be serialized should be done while holding the lock. If a thread holds the lock during a lot of other operations that aren't accessing any shared data, it is unnecessary holding up all the other threads that need to acquire that lock.

General semaphores

A general semaphore can take on any non-negative value. General semaphores are used for "counting" tasks such as creating a critical region that allows a specified number of threads to enter. For example, if you want at most four threads to be able to enter a section, you could protect it with a semaphore and initialize that semaphore to four. The first four threads will be able to decrement the semaphore and enter the region, but at that point, the semaphore will be zero and any other threads will block outside the critical region until one of the current threads leaves and signals the semaphore.

You can also use a general semaphore for representing the quantity of an available resource. Let's say you need to limit the number of simultaneously open file descriptors among multiple threads. You could initialize a general semaphore to the maximum number of open

file descriptors and each thread that wants to open a file needs to wait on the semaphore first. If the max hasn't yet been reached, the semaphore will have a positive value and the thread will be able to breeze right through the wait, decrement the semaphore and thus open a file. If the max has been reached, the semaphore value will be zero and thread will block until another thread closes a file, releasing a resource, and incrementing the semaphore that allows others to proceed.

A general semaphore can also be used to implement a form of rendezvous between threads, such as when Thread2 needs to know that Thread1 is done with something before proceeding. A rendezvous semaphore is usually initialized to zero. Thread1 waits on that semaphore (and thus immediately blocks since the value starts at zero) until Thread2 signals the semaphore when ready. If you need to rendezvous among several threads, you could have Thread1 wait several times, once for each of the threads that will signal when ready. In this case, the semaphore is "counting" the number of times an action occurred.

Global variables

You will find that concurrent programs tend to use global variables, something you may have been trained to believe is evil in its purest form and thus will balk a bit at this kind of design. It is characteristic of multi-threaded programs that data is made globally visible to allow multiple threads to access it. This is appropriate for the data that is shared and worked upon by more than one thread. Global variables tend to be the easiest way to share data in a concurrent program. However, you inherit all the usual 4 downsides of globals—keeping track of who changes the data where is difficult, it leads to routines that have lots of interdependencies other than what is indicated by the parameter lists, and so on.

As an alternative, you can declare variables as local variables within a function (most usually the main function) and then pass pointers to those variables as arguments to the new threads. This avoids the global variables and all their attendant risks and gives you direct control and documentation about which routines have access to these pieces of data. The downside is longer argument lists for the functions and more complicated variable management. You also need to be very careful here—if you are going to pass a pointer to a stack variable from one thread's stack to an another, you need to be absolutely positive that the original stack frame will remain valid for the entire time the other thread is using the pointers it was given. This can be tricky! Since the main function exists for the lifetime of the entire program, its local variables aren't at risk, but be very wary when trying to do this with any other function's local variables.

IPC Semaphores implementation in C

Here is a simple example to show how the IPC semaphores work. In this example two processes (parent and child) are communicating between each other using semaphores. Each process access the same track thrice based on which process gets the access. Once the process gets the access of the track it sleeps for 5 seconds (in this section user can do some useful work instead of sleeping !) and after that it releases the track using sem_op function.

Instead of communicating between the parent and child process we can have two different processes running on the same machine also.

```
#include <stdio.h>

//ipcsemaphore.c
//To compile : gcc -o sem ipcsemaphore.c
//To run : ./sem

#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>

void die(char *msg)
{
  perror(msg);
  exit(1);
}

int main()
{
    int i,j;
    int pid;
    int semid; /* semid of semaphore set */
    key_t key ; /* key to pass to semget() */
    int semflg = IPC_CREAT | 0666; /* semflg to pass to semget() */
    int nsems = 1; /* nsems to pass to semget() */
    int nsops; /* number of operations to do */
    struct sembuf *sops = (struct sembuf *) malloc(2*sizeof(struct sembuf));
    /* ptr to operations to perform */

    /* generate key */
    if ((key = ftok("ipcsemaphore.c", 'Q')) == -1)
       die("ftok");

    /* set up semaphore */

    printf("\nsemget: Setting up semaphore: semget(%#lx, %\
                %#o)\n",key, nsems, semflg);
    if ((semid = semget(key, nsems, semflg)) == -1)
        die("semget: semget failed");


    if ((pid = fork()) < 0)
        die("fork");

    if (pid == 0)
```

```
{
   /* child */
   i = 0;


   while (i  < 3)  /* allow for 3 semaphore sets */
   {

      nsops = 2;

      /* wait for semaphore to reach zero */

      sops[0].sem_num = 0; /* We only use one track */
      sops[0].sem_op = 0; /* wait for semaphore flag to become zero */
      sops[0].sem_flg = SEM_UNDO; /* take off semaphore asynchronous  */


      sops[1].sem_num = 0;
      sops[1].sem_op = 1; /* increment semaphore -- take control of track */
      sops[1].sem_flg = SEM_UNDO | IPC_NOWAIT; /* take off semaphore */

      /* Recap the call to be made. */

      printf("\nsemop:Child  Calling semop(%d, &sops, %d) with:", semid, nsops);
      for (j = 0; j < nsops; j++)
      {
          printf("\n\tsops[%d].sem_num = %d, ", j, sops[j].sem_num);
          printf("sem_op = %d, ", sops[j].sem_op);
          printf("sem_flg = %#o\n", sops[j].sem_flg);
      }

      /* Make the semop() call and report the results. */
      if ((j = semop(semid, sops, nsops)) == -1)
      {
         perror("semop: semop failed");
      }
      else
      {
         printf("\n\nChild Process Taking Control of Track: %d/3 times\n", i+1);
         sleep(5); /* DO Nothing for 5 seconds */

         nsops = 1;

         /* wait for semaphore to reach zero */
         sops[0].sem_num = 0;
         sops[0].sem_op = -1; /* Give UP COntrol of track */
```

```
            sops[0].sem_flg = SEM_UNDO | IPC_NOWAIT; /* take off semaphore,
asynchronous  */


            if ((j = semop(semid, sops, nsops)) == -1)
            {
               perror("semop: semop failed");
            }
            else
                printf("Child Process Giving up Control of Track: %d/3 times\n", i+1);
            sleep(5); /* halt process to allow parent to catch semaphore change first */
         }
         ++i;
      }

   }
   else /* parent */
   {
      i = 0;

      while (i  < 3)   /* allow for 3 semaphore sets */
      {

         nsops = 2;

         /* wait for semaphore to reach zero */
         sops[0].sem_num = 0;
         sops[0].sem_op = 0; /* wait for semaphore flag to become zero */
         sops[0].sem_flg = SEM_UNDO; /* take off semaphore asynchronous  */


         sops[1].sem_num = 0;
         sops[1].sem_op = 1; /* increment semaphore -- take control of track */
         sops[1].sem_flg = SEM_UNDO | IPC_NOWAIT; /* take off semaphore */

         /* Recap the call to be made. */

          printf("\nsemop:Parent Calling semop(%d, &sops, %d) with:", semid, nsops);
         for (j = 0; j < nsops; j++)
         {
            printf("\n\tsops[%d].sem_num = %d, ", j, sops[j].sem_num);
            printf("sem_op = %d, ", sops[j].sem_op);
            printf("sem_flg = %#o\n", sops[j].sem_flg);
         }

         /* Make the semop() call and report the results. */
         if ((j = semop(semid, sops, nsops)) == -1)
```

```
        {
           perror("semop: semop failed");
        }
        else
        {
           printf("Parent Process Taking Control of Track: %d/3 times\n", i+1);
           sleep(5); /* Sleep for 5 seconds */

           nsops = 1;

           /* wait for semaphore to reach zero */
           sops[0].sem_num = 0;
           sops[0].sem_op = -1; /* Give UP Control of track */
           sops[0].sem_flg = SEM_UNDO | IPC_NOWAIT; /* take off semaphore,
asynchronous  */

           if ((j = semop(semid, sops, nsops)) == -1)
           {
              perror("semop: semop failed");
           }
           else
              printf("Parent Process Giving up Control of Track: %d/3 times\n", i+1);
           sleep(5); /* halt process to allow child to catch semaphore change first */
        }
        ++i;

     }

  }
}
```

Problem 2:

Implement Producer-Consumer Problem using Pipes.

Creating ``pipelines" with the C programming language can be a bit more involved than our simple shell example. To create a simple pipe with C, we make use of the pipe() system call. It takes a single argument, which is an array of two integers, and if successful, the array will contain two new file descriptors to be used for the pipeline. After creating a pipe, the process typically spawns a new process (remember the child inherits open file descriptors).

SYSTEM CALL: pipe();

PROTOTYPE: int pipe(int fd[2]);
RETURNS: 0 on success
-1 on error: errno = EMFILE (no free descriptors)

EMFILE (system file table is full)
EFAULT (fd array is not valid)

NOTES: fd[0] is set up for reading, fd[1] is set up for writing

---

The first integer in the array (element 0) is set up and opened for reading, while the second integer (element 1) is set up and opened for writing. Visually speaking, the output of fd1 becomes the input for fd0. Once again, all data traveling through the pipe moves through the kernel.

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

main()
{
      int     fd[2];

      pipe(fd);
      .
      .
}
```

Remember that an array name in C *decays* into a pointer to its first member. Above, fd is equivalent to &fd[0]. Once we have established the pipeline, we then fork our new child process:

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

main()
{
      int     fd[2];
      pid_t   childpid;

      pipe(fd);

      if((childpid = fork()) == -1)
      {
            perror("fork");
            exit(1);
      }
      .
      .
}
```

If the parent wants to receive data from the child, it should close fd1, and the child should close fd0. If the parent wants to send data to the child, it should close fd0, and the child should close fd1. Since descriptors are shared between the parent and child, we should always be sure to close the end of pipe we aren't concerned with. On a technical note, the EOF will never be returned if the unnecessary ends of the pipe are not explicitly closed.

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

main()
{
        int     fd[2];
        pid_t   childpid;

        pipe(fd);

        if((childpid = fork()) == -1)
        {
                perror("fork");
                exit(1);
        }

        if(childpid == 0)
        {
                /* Child process closes up input side of pipe */
                close(fd[0]);
        }
        else
        {
                /* Parent process closes up output side of pipe */
                close(fd[1]);
        }
        .
        .
}
```

As mentioned previously, once the pipeline has been established, the file descriptors may be treated like descriptors to normal files.

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


/*****************************************
```

```
 *              producer                *
 ****************************************
 * Write information into the pipe whose   *
 * write-end is given by pipe_write_end.   *
 ****************************************/

void producer(FILE *pipe_write_end)
{
  int i;
  for(i = 1; i <= 100; i++) {
    fprintf(pipe_write_end, "%d ", i);
  }
  fclose(pipe_write_end);
  exit(0);
}

/****************************************
 *              consumer                *
 ****************************************
 * Read information from the pipe whose    *
 * read-end is given by pipe_read_end,     *
 * and copy it to the standard output.     *
 ****************************************/

void consumer(FILE *pipe_read_end)
{
  int n,k;

  while(1) {
    int n = fscanf(pipe_read_end, "%d", &k);
    if(n == 1) printf("consumer: got %d\n", k);
    else break;
  }
  fclose(pipe_read_end);
  exit(0);
}

/****************************************
 *              main                    *
 ****************************************
 * Build the producer and consumer and     *
 * wait for them to finish.                *
 ****************************************/

int main()
{
  pid_t producer_id, consumer_id;
```

```
  int pd[2];
  FILE *pipe_write_end, *pipe_read_end;

  /*-----------------*
   * Build the pipe. *
   *-----------------*/

  pipe(pd);
  pipe_read_end  = fdopen(pd[0], "r");
  pipe_write_end = fdopen(pd[1], "w");

  /*--------------------*
   * Fork the producer. *
   *--------------------*/

  producer_id = fork();
  if(producer_id == 0) {
    fclose(pipe_read_end);
    producer(pipe_write_end);
  }

  /*--------------------*
   * Fork the consumer. *
   *--------------------*/

  consumer_id = fork();
  if(consumer_id == 0) {
    fclose(pipe_write_end);
    consumer(pipe_read_end);
  }

  /*-------------------------*
   * Wait for both to finish. *
   *-------------------------*/

  fclose(pipe_read_end);
  fclose(pipe_write_end);
  wait(NULL);
  wait(NULL);

  return 0;
}
```