Experiment Number: 14

TITLE: Producer-Consumer Problem (Using Shared memory)

OBJECTIVE:

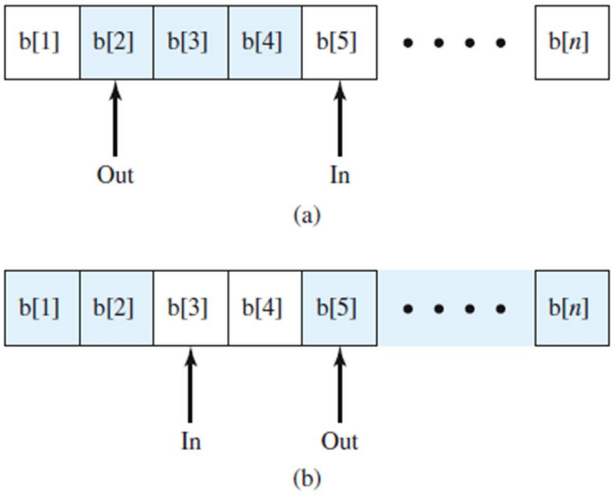
1.To study use of shared memory and semaphore in unix.
2.To study Producer-Consumer problem of operating system.

Theory:

The Producer/Consumer Problem

The general statement is this: there are one or more producers generating some type of data (records, characters) and placing these in a buffer. There is a single consumer that is taking items out of the buffer one at a time. The system is to be constrained to prevent the overlap of buffer operations. That is, only one agent (producer or consumer) may access the buffer at any one time. The problem is to make sure that the producer won't try to add data into the buffer if it's full and that the consumer won't try to remove data from an empty buffer.
To begin, let us assume that the buffer is infinite and consists of a linear array of elements. In abstract terms, we can define the producer and consumer functions as follows:

Solution to bounded/circular buffer producer-consumer problem



producer:

```
while (true) {
/* produce item v */;
b[in] = v;
in++;
}
```

consumer:

```
while (true) {
while (in <= out)
/* do nothing */;
w = b[out];
out++;
/* consume item w */;
}
```

Solution using Semaphore

```
const int sizeofbuffer = /* buffer size */;
semaphore s = 1, n = 0, e = sizeofbuffer;
void producer()
{
     while (true)
     {
          produce();
          semWait(e);
          semWait(s);
          append();
          semSignal(s);
          semSignal(n);
     }
}
void consumer()
{
     while (true)
     {
          semWait(n);
          semWait(s);
          take();
          semSignal(s);
          semSignal(e);
          consume();
     }
}
void main()
{
          parbegin (producer, consumer);
}
```

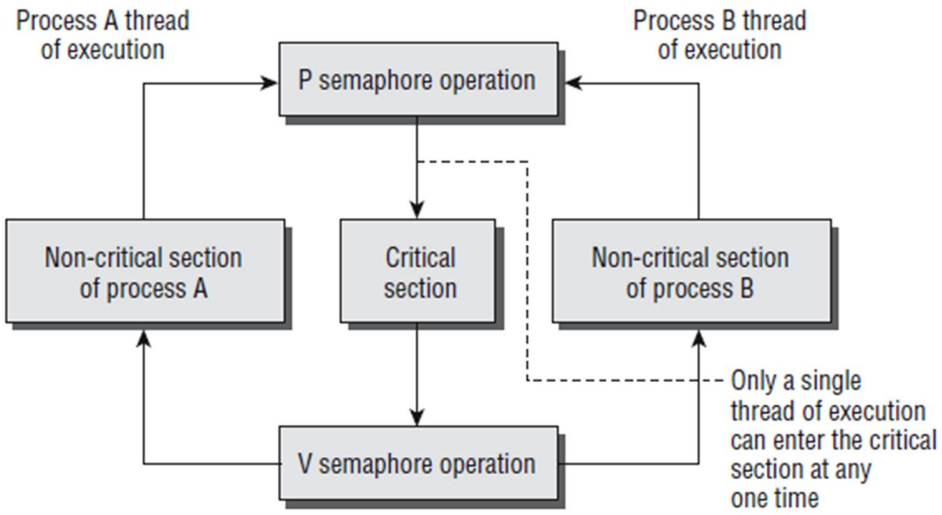
Semaphore

It is a special type of variable containing integer value used for signaling among processes. Only three operations may be performed on a semaphore, all of which are atomic: initialize, decrement, and increment. The decrement operation may result in the blocking of a process, and the increment operation may result in the unblocking of a process, also known as a counting semaphore or a general semaphore.

A semaphore may be initialized to a nonnegative integer value. The semWait operation decrements the semaphore value. If the value becomes negative, then the process executing the semWait is blocked. Otherwise, the process continues execution. The semSignal operation increments the semaphore value. If the resulting value is less than or equal to zero, then a process blocked by a semWait operation, if any, is unblocked.

Generalized use of semaphore for forcing critical section

```
semaphore sv = 1;
loop forever
{
     Wait(sv);
     critical code section;
     signal(sv);
     noncritical code section;
}
```



Linux Semaphore Facilities

```
#include <sys/sem.h>
int semctl(int sem_id, int sem_num, int command, ...);
int semget(key_t key, int num_sems, int sem_flags);
int   semop(int   sem_id,   struct   sembuf   *sem_ops,   size_t
num_sem_ops);
```

The header file sys/sem.h usually relies on two other header files, sys/types.h and sys/ipc.h. Normally they are automatically included by sys/sem.h and we do not need to explicitly add a #include for them. These functions were designed to work for arrays of semaphore values, which makes their operation significantly more complex than would have been required for a single semaphore.

semget()

The semget function creates a new semaphore or obtains the semaphore key of an existing semaphore:

```
int semget(key_t key, int num_sems, int sem_flags);
```

The first parameter, *key*, is an integral value used to allow unrelated processes to access the same semaphore. All semaphores are accessed indirectly by the program supplying a key, for which the system then generates a semaphore identifier. The semaphore key is used only with semget. All other semaphore functions use the semaphore identifier returned from semget.

There is a special semaphore key value, IPC_PRIVATE, that is intended to create a semaphore that only the creating process could access, but this rarely has any useful purpose.

The *num_sems* parameter is the number of semaphores required. This is almost always 1.

The *sem_flags* parameter is a set of flags, very much like the flags to the open function. The lower nine bits are the permissions for the semaphore, which behave like file permissions. In addition, these can be bitwise ORed with the value IPC_CREAT to create a new semaphore.

The semget function returns a positive (nonzero) value on success; this is the semaphore identifier used in the other semaphore functions. On error, it returns –1.

semop()

The function semop is used for changing the value of the semaphore:

```
int semop(int sem_id, struct sembuf *sem_ops, size_t
              num_sem_ops);
```

sem_id, is the semaphore identifier, as returned from semget. The second parameter, **sem_ops**, is a pointer to an array of structures, each of which will have at least the following members:

```
struct sembuf
{
     short sem_num;
     short sem_op;
     short sem_flg;
}
```

The first member, *sem_num*, is the semaphore number, usually 0 unless we're working with an array of semaphores.

The *sem_op* member is the value by which the semaphore should be changed. (We can change a semaphore by amounts other than 1.) In general, only two values are used, –1, which

is your 'wait' operation to wait for a semaphore to become available, and +1, which is our 'signal' operation to signal that a semaphore is now available.

The final member, *sem_flg*, is usually set to SEM_UNDO or a zero.

semctl()

The semctl function allows direct control of semaphore information:

```
int semctl(int sem_id, int sem_num, int command, ...);
```

The first parameter, sem_id, is a semaphore identifier, obtained from semget. The sem_num parameter is the semaphore number. We use this when we're working with arrays of semaphores. Usually, this is 0, the first and only semaphore. The command parameter is the action to take, and a fourth parameter, if present, is a union semun, which according to the X/OPEN specification must have at least the following members:

```
union semun
{
     int val;
     struct semid_ds *buf;
     unsigned short *array;
}
```

There are many different possible values of command allowed for semctl.

The two common values of command are:

SETVAL: Used for initializing a semaphore to a known value. The value required is passed as the val member of the union semun. This is required to set the semaphore up before it's used for the first time.

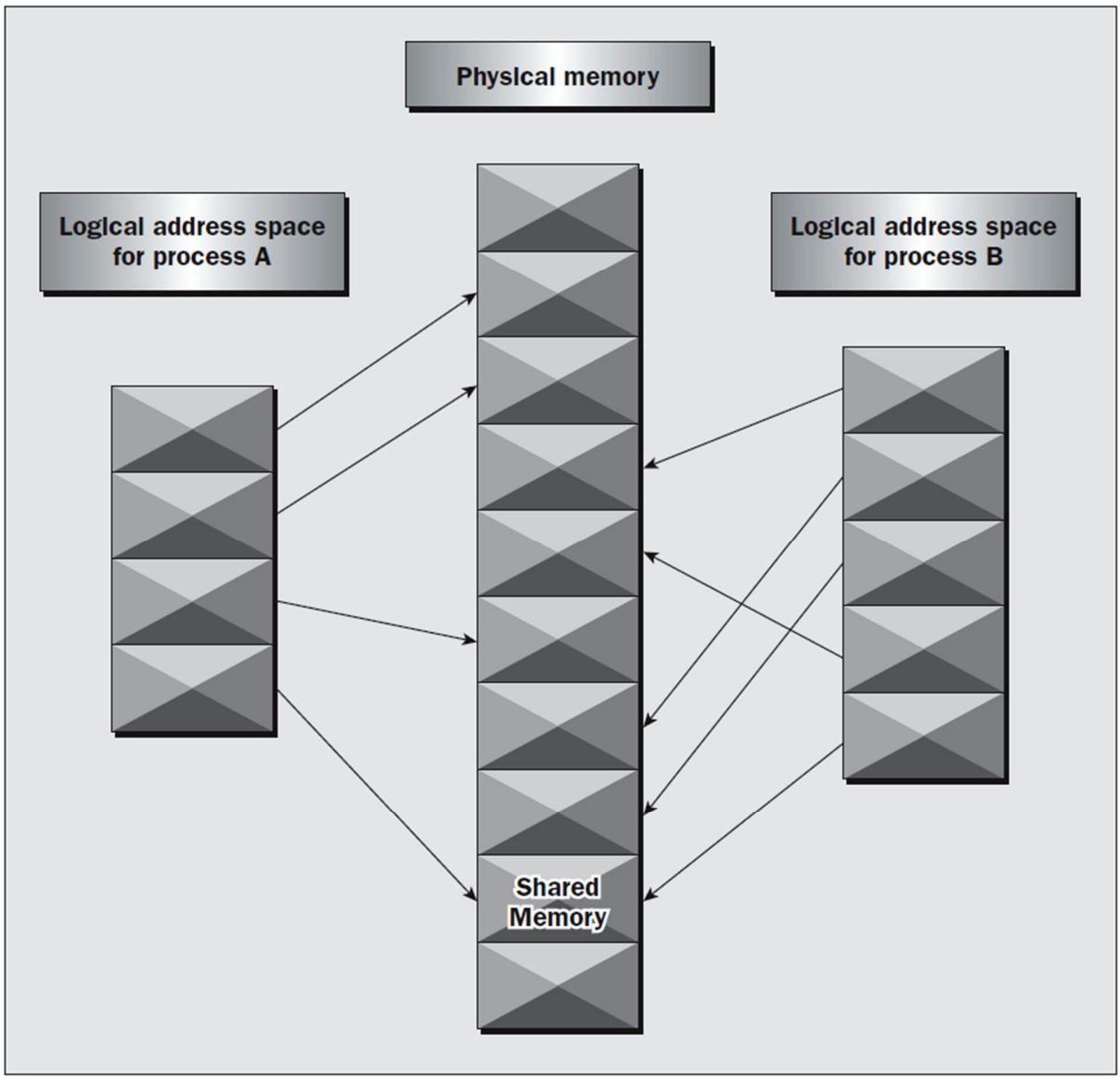
IPC_RMID: Used for deleting a semaphore identifier when it's no longer required. The semctl function returns different values depending on the command parameter. For SETVAL and IPC_RMID it returns 0 for success and –1 on error.

Shared Memory

Shared memory allows two unrelated processes to access the same logical memory. Shared memory is a very efficient way of transferring data between two running processes.

Shared memory is a special range of addresses that is created by IPC for one process and appears in the address space of that process. Other processes can then "attach" the same shared memory segment into their own address space. All processes can access the memory locations just as if the memory had been allocated by malloc. If one process writes to the shared memory, the changes immediately become visible to any other process that has access to the same shared memory. Shared memory provides an efficient way of sharing and passing data between multiple processes. By itself, shared memory doesn't provide any synchronization facilities. Because it provides no synchronization facilities, we usually need to use some other mechanism to synchronize access to the shared memory. Typically, we use shared memory to provide efficient access to large areas of memory and pass small messages to synchronize

access to that memory. There are no automatic facilities to prevent a second process from starting to read the shared memory before the first process has finished writing to it. It's the responsibility of the programmer to synchronize access. Figure below shows an illustration of how shared memory works.



The arrows show the mapping of the logical address space of each process to the physical memory available. In practice, the situation is more complex because the available memory actually consists of a mix of physical memory and memory pages that have been swapped out to disk. The functions for shared memory resemble those for semaphores:

```
#include <sys/shm.h>
void *shmat(int shm_id, const void *shm_addr, int shmflg);
int shmctl(int shm_id, int cmd, struct shmid_ds *buf);
```

```
int shmdt(const void *shm_addr);
int shmget(key_t key, size_t size, int shmflg);
```

As with semaphores, the include files sys/types.h and sys/ipc.h are normally automatically included by shm.h.

shmget()

It is used to create shared memory.

```
int shmget(key_t key, size_t size, int shmflg);
```

As with semaphores, the program provides *key*, which effectively names the shared memory segment, and the shmget function returns a shared memory identifier that is used in subsequent shared memory functions. There's a special key value, IPC_PRIVATE, that creates shared memory private to the process.

The second parameter, *size*, specifies the amount of memory required in bytes.

The third parameter, *shmflg*, consists of nine permission flags that are used in the same way as the mode flags for creating files. A special bit defined by IPC_CREAT must be bitwise ORed with the permissions to create a new shared memory segment. It's not an error to have the IPC_CREAT flag set and pass the key of an existing shared memory segment. The IPC_CREAT flag is silently ignored if it is not required.

If the shared memory is successfully created, shmget returns a nonnegative integer, the shared memory identifier. On failure, it returns –1.

shmat()

When we first create a shared memory segment, it's not accessible by any process. To enable access to the shared memory, we must attach it to the address space of a process. We do this with the shmat function:

```
void *shmat(int shm_id, const void *shm_addr, int shmflg);
```

The first parameter, *shm_id*, is the shared memory identifier returned from shmget. The second parameter, *shm_addr*, is the address at which the shared memory is to be attached to the current process. This should almost always be a null pointer, which allows the system to choose the address at which the memory appears.

The third parameter, shmflg, is a set of bitwise flags. The two possible values are SHM_R for writing and SHM_W for write access. If the shmat call is successful, it returns a pointer to the first byte of shared memory. On failure –1 is returned.

shmctl( )

it is used for controlling functions for shared memory.

```
int shmctl(int shm_id, int command, struct shmid_ds *buf);
```

The shmid_ds structure has at least the following members:

```
struct shmid_ds
{
     uid_t shm_perm.uid;
     uid_t shm_perm.gid;
     mode_t shm_perm.mode;
}
```

The first parameter, *shm_id*, is the identifier returned from shmget. The second parameter, *command*, is the action to take. It can take three values, shown in the following table.

Command	Description
IPC_STAT	Sets the data in the shmid_ds structure to reflect the values associated with the shared memory.
IPC_SET	Sets the values associated with the shared memory to those provided in the shmid_ds data structure, if the process has permission to do so.
IPC_RMID	Deletes the shared memory segment.

The third parameter, *buf*, is a pointer to the structure containing the modes and permissions for the shared memory.

References:

1. "Beginning Linux Programming" by Neil Mathew and Richard Stones, Wrox Publications.
2. "Operating System Internals and Design Implementation" by William Stallings, Pearson Education.

Tushar B Kute
(Subject Teacher)