Experiment Number: 14

TITLE: Producer-Consumer Problem (Using Pipes)

OBJECTIVE:

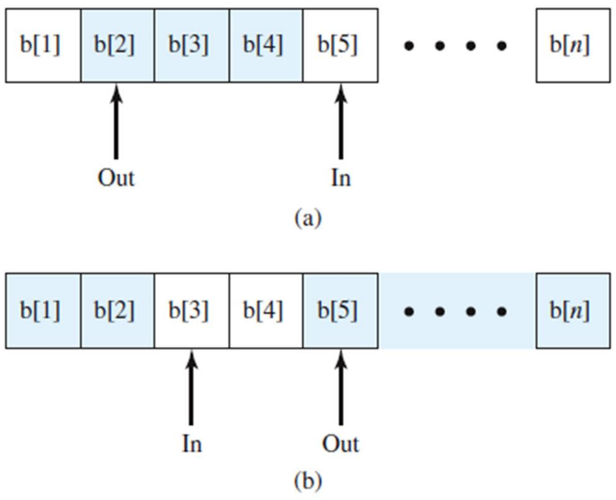
1.To study use of pipes and binary semaphore in Unix.
2.To study Producer-Consumer problem of operating system.

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 (Unbounded Buffer).

```
semaphore s = 1, n = 0;
void producer()
{
     while (true)
     {
          produce();
          semWait(s);
          append();
          semSignal(s);
          semSignal(n);
     }
}
void consumer()
{
     while (true)
     {
          semWait(n);
          semWait(s);
          take();
          semSignal(s);
          consume();
     }
}
void main()
{
          parbegin (producer, consumer);
}
```

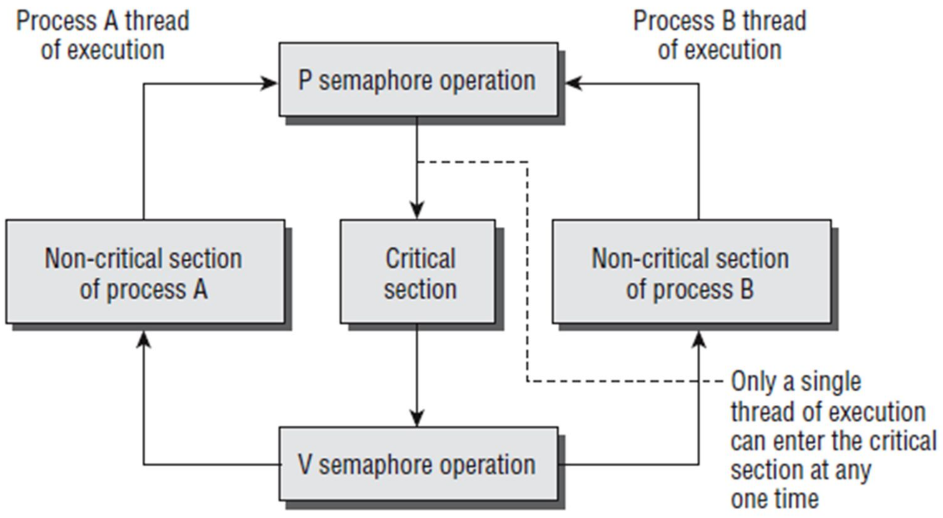
Semaphore

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;
}
```


Process A thread of execution
Process B thread of execution
P semaphore operation
Non-critical section of process A
Critical section
Non-critical section of process B
Only a single thread of execution can enter the critical section at any one time
V semaphore operation

Linux Semaphore Facilities (Binary Semaphore)

A semaphore is created with the sem_init function, which is declared as follows:

```
     #include <semaphore.h>
     int sem_init(sem_t *sem, int pshared, unsigned int value);
```

This function initializes a semaphore object pointed to by *sem*, sets its sharing option and gives it an initial integer value. The *pshared* parameter controls the type of semaphore. If the value of *pshared* is 0, the semaphore is local to the current process. Otherwise, the semaphore may be shared between processes. Here we are interested only in semaphores that are not shared between processes. At the time of writing, Linux doesn't support this sharing, and passing a nonzero value for *pshared* will cause the call to fail.

The next pair of functions controls the value of the semaphore and is declared as follows:

```
     #include <semaphore.h>
     int sem_wait(sem_t * sem);
     int sem_post(sem_t * sem);
```

These both take a pointer to the semaphore object initialized by a call to sem_init. The sem_post function atomically increases the value of the semaphore by 1. Atomically here means that if two threads simultaneously try to increase the value of a single semaphore by 1, they do not interfere with each other, as might happen if two programs read, increment, and write a value to a file at the same time. If both programs try to increase the value by 1, the semaphore will always be correctly increased in value by 2.

The sem_wait function atomically decreases the value of the semaphore by one, but always waits until the semaphore has a nonzero count first. Thus, if you call sem_wait on a semaphore with a value of 2, the thread will continue executing but the semaphore will be decreased to 1. If sem_wait is called on a semaphore with a value of 0, the function will wait until some other thread has incremented the value so that it is no longer 0. If two threads are both waiting in sem_wait for the same semaphore to become nonzero and it is incremented once by a third process, only one of the two waiting processes will get to decrement the semaphore and continue; the other will remain waiting. This atomic “test and set” ability in a single function is what makes semaphores so valuable.

The last semaphore function is sem_destroy. This function tidies up the semaphore when we have finished with it. It is declared as follows:

```
#include <semaphore.h>
int sem_destroy(sem_t * sem);
```

Again, this function takes a pointer to a semaphore and tidies up any resources that it may have. If we attempt to destroy a semaphore for which some thread is waiting, we will get an error. Like most Linux functions, these functions all return 0 on success.

The Pipe Call

The lower-level pipe( ) function provides a means of passing data between two programs, without the overhead of invoking a shell to interpret the requested command. It also gives us more control over the reading and writing of data.

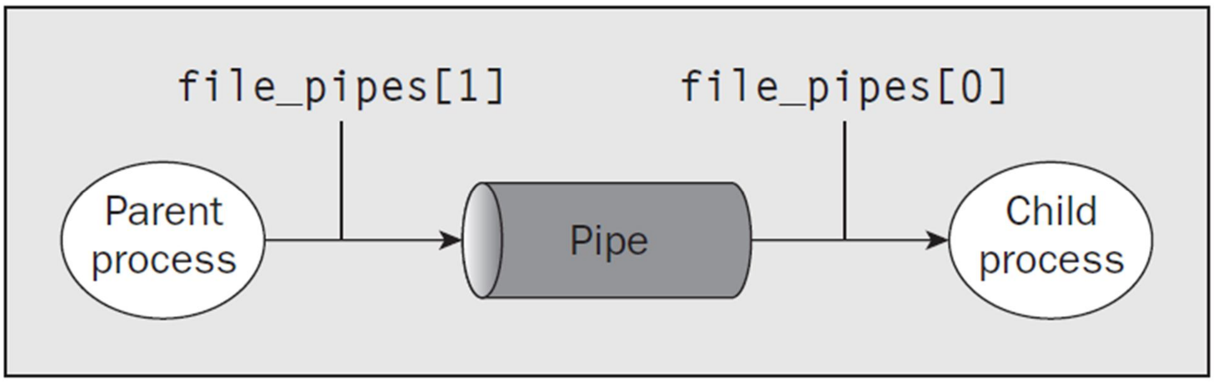
The pipe function has the following prototype:

```
#include <unistd.h>
int pipe(int file_descriptor[2]);
```

It is passed (a pointer to) an array of two integer file descriptors. It fills the array with two new file descriptors and returns a zero. On failure, it returns -1 and sets errno to indicate the reason for failure. Errors defined in the Linux manual page for pipe (in section 2 of the manual) are

- ❑ EMFILE: Too many file descriptors are in use by the process.
- ❑ ENFILE: The system file table is full.
- ❑ EFAULT: The file descriptor is not valid.

The two file descriptors returned are connected in a special way. Any data written to file_descriptor[1] can be read back from file_descriptor[0]. The data is processed in a first in, first out basis. This means that if we write the bytes 1, 2, 3 to file_descriptor[1], reading from file_descriptor[0] will produce 1, 2, 3. The illustration is given as below:



Each running program, called a process, has a number of file descriptors associated with it. These are small integers that we can use to access open files or devices. How many of these are available will vary depending on how the system has been configured. When a program starts, it usually has three of these descriptors already opened. These are:

- ❑ 0: Standard input
- ❑ 1: Standard output
- ❑ 2: Standard error

We can associate other file descriptors with files and devices by using the open system call. The file descriptors that are automatically opened, however, already allow you to create some simple programs using write.

The write( ) system call

The write system call arranges for the first *nbytes* bytes from *buf* to be written to the file associated with the file descriptor *fildes*. It returns the number of bytes actually written. This may be less than *nbytes* if there has been an error in the file descriptor or if the underlying device driver is sensitive to block size. If the function returns 0, it means no data was written; if it returns –1, there has been an error in the write call, and the error will be specified in the *errno* global variable.

```
#include <unistd.h>
size_t write(int fildes, const void *buf, size_t nbytes);
```

The read( ) system call

The read system call reads up to nbytes bytes of data from the file associated with the file descriptor fildes and places them in the data area buf. It returns the number of data bytes actually read, which may be less than the number requested. If a read call returns 0, it had nothing to read; it reached the end of the file. Again, an error on the call will cause it to return –1.

```
#include <unistd.h>
size_t read(int fildes, void *buf, size_t nbytes);
```

References:

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

Tushar B Kute
(Subject Teacher)