15.7 Exercise: Implementing Pipes with Shared Memory

This section develops a specification for a software pipe consisting of a semaphore set to protect access to the pipe and a shared memory segment to hold the pipe data and state information. The pipe state information includes the number of bytes of data in the pipe, the position of next byte to be read and status information. The pipe can hold at most one message of maximum size _POSIX_PIPE_BUF. Represent the pipe by the following pipe_t structure allocated in shared memory.

```
typedef struct pipe {
        int semid;              /* ID of protecting semaphore set */
        int shmid;              /* ID of the shared memory segment */
        char data[_POSIX_PIPE_BUF];     /* buffer for the pipe data */
        int data_size;          /* bytes currently in the pipe */
        void *current_start;    /* pointer to current start of data */
        int end_of_file;        /* true after pipe closed for writing */
} pipe_t;
```

A program creates and references the pipe by using a pointer to pipe_t as a handle. For simplicity, assume that only one process can read from the pipe and one process can write to the pipe. The reader must clean up the pipe when it closes the pipe. When the writer closes the pipe, it sets the end_of_file member of pipe_t so that the reader can detect end-of-file.

The semaphore set protects the pipe_t data structure during shared access by the reader and the writer. Element zero of the semaphore set controls exclusive access to data. It is initially 1. Readers and writers acquire access to the pipe by decrementing this semaphore element, and they release access by incrementing it. Element one of the semaphore set controls synchronization of writes so that data contains only one message, that is, the output of a single write operation. When this semaphore element is 1, the pipe is empty. When it is 0, the pipe has data or an end-of-file has been encountered. Initially, element one is 1. The writer decrements element one before writing any data. The reader waits until element one is 0 before reading. When it has read all the data from the pipe, the reader increments element one to indicate that the pipe is now available for writing. Write the following functions.

pipe_t *pipe_open(void);

creates a software pipe and returns a pointer of type pipe_t * to be used as a handle in the other calls. The algorithm for pipe_open is as follows.

1. Create a shared memory segment to hold a pipe_t data structure by calling shmget. Use a key of IPC_PRIVATE and owner read/write permissions.

2. Attach the segment by calling shmat. Cast the return value of shmat to a pipe_t * and assign it to a local variable p.

3. Set p->shmid to the ID of the shared memory segment returned by the shmget.

4. Set p->data_size and p->end_of_file to 0.

5. Create a semaphore set containing two elements by calling semget with IPC_PRIVATE key and owner read, write, execute permissions.

6. Initialize both semaphore elements to 1, and put the resulting semaphore ID value in p->semid.

7. If all the calls were successful, return p.

8. If an error occurs, deallocate all resources, set errno, and return a NULL pointer.

int pipe_read(pipe_t *p, char *buf, int bytes);

behaves like an ordinary blocking read function. The algorithm for pipe_read is as follows.

1. Perform semop on p->semid to atomically decrement semaphore element zero, and test semaphore element one for 0. Element zero provides mutual exclusion. Element one is only 0 if there is something in the buffer.

2. If p->data_size is greater than 0 do the following.
    a. Copy at most bytes bytes of information starting at position p->current_start of the software pipe into buf. Take into account the number of bytes in the pipe.
    b. Update the p->current_start and p->data_size members of the pipe data structure.
    c. If successful, set the return value to the number of bytes actually read.

3. Otherwise, if p->data_size is 0 and p->end_of_file is true, set the return value to 0 to indicate end-of-file.

4. Perform another semop operation to release access to the pipe. Increment element zero. If no more data is in the pipe, also increment element one unless p->end_of_file is true. Perform these operations atomically by a single semop call.

5. If an error occurs, return –1 with errno set.

int pipe_write(pipe_t *p, char *buf, int bytes);

behaves like an ordinary blocking write function. The algorithm for pipe_write is as follows.

1. Perform a semop on p->semid to atomically decrement both semaphore elements zero and one.

2. Copy at most _POSIX_PIPE_BUF bytes from buf into the pipe buffer.

3. Set p->data_size to the number of bytes actually copied, and set p->current_start to 0.

4. Perform another semop call to atomically increment semaphore element zero of the semaphore set.

5. If successful, return the number of bytes copied.

6. If an error occurs, return –1 with errno set.

int pipe_close(pipe_t *p, int how);

closes the pipe. The how parameter determines whether the pipe is closed for reading or writing. Its possible values are O_RDONLY and O_WRONLY. The algorithm for pipe_close is as follows.

1. Use the semop function to atomically decrement element zero of p->semid. If the semop fails, return –1 with errno set.

2. If how & O_WRONLY is true, do the following.
   a. Set p->end_of_file to true.
   b. Perform a semctl to set element one of p->semid to 0.
   c. Copy p->semid into a local variable, semid_temp.
   d. Perform a shmdt to detach p.
   e. Perform a semop to atomically increment element zero of semid_temp.

   If any of the semop, semctl, or shmdt calls fail, return –1 immediately with errno set.

3. If how & O_RDONLY is true, do the following.
   a. Perform a semctl to remove the semaphore p->semid. (If the writer is waiting on the semaphore set, its semop returns an error when this happens.)
   b. Copy p->shmid into a local variable, shmid_temp.
   c. Call shmdt to detach p.
   d. Call shmctl to deallocate the shared memory segment identified by shmid_temp.

   If any of the semctl, shmdt, or shmctl calls fail, return –1 immediately with errno set.

Test the software pipe by writing a main program that is similar to Program 6.4. The program creates a software pipe and then forks a child. The child reads from standard input and writes to the pipe. The parent reads what the child has written to the pipe and outputs it to standard output. When the child detects end-of-file on standard input, it closes the pipe for writing. The parent then detects end-of-file on the pipe, closes the pipe for reading (which destroys the pipe), and exits. Execute the ipcs command to check that everything was properly destroyed.