CMR
INSTITUTE OF
TECHNOLOGY

USN | | | | | | | | | | |

Internal Assesment Test - III

Sub:	UNIX SYSTEM PROGRAMMING							Code:	10CS62
Date:	25 / 05 /2017	Duration:	90 mins	Max Marks:	50	Sem:	6(A,B,C)	Branch:	CSE

Answer any **FIVE** full questions

		Marks	OBE	
			CO	RBT
1 a)	What are pipes? What are the limitations of pipes ? Explain with diagrams different ways to view a half duplex pipe.	[6]	CO6	L1
b)	Write a C program to create a pipe between a parent and its child process to send Data (“HELLO”) down the pipe.	[4]	CO6	L3
2	What are message queues? Write the declaration of the structure of the message message queue (structmsqid_ds). Explain 4 APIs used with message queues along with prototypes	[10]	CO6	L1
3. a)	Discuss with diagrams, the client server communications using FIFO.	[5]	CO6	L2
b)	What you mean by passing file descriptor? Explain.	[5]	CO6	L1

3. a)	Discuss with diagrams, the client server communications using FIFO.	[5]	CO6	L2
b)	What you mean by passing file descriptor? Explain.	[5]	CO6	L1

4.	What is a socket?Explain the four different APIs used for establishing connection between two systems using sockets	[10]	CO6	L1
5.a)	What are STREAMS pipes?Explain with diagrams different ways to view a STREAMS pipe.	[5]	CO6	L1
b)	Explain with neat diagrams ,how STREAMS PIPES can be used to implement client server model.	[5]	CO6	L4
6.a)	Explain the 3 APIs used to create and control the semaphores	[6]	CO6	L4
b)	Write a C program to demonstrate the usage of popen and pclose functions	[4]	CO6	L3

DEPARTMENT : CSE/ISE

Scheme and Solution for IAT3-May 2017

Unix System Programming (10CS62)

1a) Pipes are the oldest form of UNIX System IPC.
Two limitations.
1.they have been half duplex (i.e., data flows in only one direction).
2. Pipes can be used only between processes that have a common ancestor. Normally, a pipe is created by a process, that process calls fork , and the pipe is used between the parent and the child.
A pipe is created by calling the pipe function.

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

Returns: 0 if OK, 1 on error.
Two file descriptors are returned through the filedes argument: filedes[0] is open for reading, and filedes[1] is open for writing. The output of filedes[1] is the input for filedes[0].
Two ways to picture a half-duplex pipe are shown in Figure 15.2. The left half of the figure shows the two ends of the pipe connected in a single process. The right half of the figure emphasizes that the data in the pipe flows through the kernel.

PROGRAM: shows the code to create a pipe between a parent and its child and to send data down the pipe.

```
#include "apue.h"
int
main(void)
{
int
int
pid_t
char
n;
fd[2];
pid;
line[MAXLINE];
if (pipe(fd) < 0)
err_sys("pipe error");
if ((pid = fork()) < 0) {
err_sys("fork error");
} else if (pid > 0) {
/* parent */
close(fd[0]);
write(fd[1], "hello world\n", 12);
} else {
/* child */
close(fd[1]);
n = read(fd[0], line, MAXLINE);
write(STDOUT_FILENO, line, n);
}
exit(0);
}
```

2)
A message queue is a linked list of messages stored within the kernel and identified by a message queue identifier.
We'll call the message queue just a queue and its identifier a queue ID.
A new queue is created or an existing queue opened by msgget . New messages are added to the end of a queue by
msgsnd . Every message has a positive long integer type field, a non-negative length, and the actual data bytes
(corresponding to the length), all of which are specified to msgsnd when the message is added to a queue. Messages
are fetched from a queue by msgrcv . We don't have to fetch the messages in a first-in, first-out order. Instead, we
can fetch messages based on their type field.
Each queue has the following msqid_ds structure associated with it:

```
struct msqid_ds
{
struct ipc_perm
msgqnum_t
msglen_t
pid_t
pid_t
msg_perm;
msg_qnum;
msg_qbytes;
msg_lspid;
msg_lrpid;
/*
/*
/*
/*
/*
see Section 15.6.2 */
# of messages on queue */
max # of bytes on queue */
pid of last msgsnd() */
pid of last msgrcv() */


time_t
time_t
time_t
.
.
```

.

msg_stime;
msg_rtime;
msg_ctime;
/* last-msgsnd() time */
/* last-msgrcv() time */
/* last-change time */
};
This structure defines the current status of the queue.
The first function normally called is msgget to either open an existing queue or create a new queue.
#include <sys/msg.h>
int msgget(key_t key, int flag);
Returns: message queue ID if OK, 1 on error
When a new queue is created, the following members of the msqid_ds structure are initialized.
  The ipc_perm structure is initialized. The mode member of thisstructure is set to the corresponding
permission bits of flag.
  msg_qnum , msg_lspid , msg_lrpid , msg_stime , and msg_rtime are all set to 0.
  msg_ctime is set to the current time.
  msg_qbytes is set to the system limit.
On success, msgget returns the non-negative queue ID. This value is then used with the other three message queue
functions.
The msgctl function performs various operations on a queue.
#include <sys/msg.h>
int msgctl(int msqid, int cmd, struct msqid_ds *buf);
Returns: 0 if OK, 1 on error.
The cmd argument specifies the command to be performed on the queue specified by msqid.

ata is placed onto a message queue by calling msgsnd .
#include <sys/msg.h>
int msgsnd(int msqid, const void *ptr, size_t nbytes, int flag);
Returns: 0 if OK, 1 on error.
Each message is composed of a positive long integer type field, a non-negative length (nbytes), and the actual data
bytes (corresponding to the length). Messages are always placed at the end of the queue.
The ptr argument points to a long integer that contains the positive integer message type, and it is immediately
followed by the message data. (There is no message data if nbytes is 0.) If the largest message we send is 512 bytes,
we can define the following structure:
struct mymesg
{
long mtype;
/* positive message type */
char mtext[512]; /* message data, of length nbytes */

};
The ptr argument is then a pointer to a mymesg structure. The message type can be used by the receiver to fetch
messages in an order other than first in, first out.

Messages are retrieved from a queue by msgrcv .
#include <sys/msg.h>
ssize_t msgrcv(int msqid, void *ptr, size_t nbytes, long type, int flag);
Returns: size of data portion of message if OK, 1 on error.
The type argument lets us specify which message we want.
type == 0 The first message on the queue is returned.
type > 0 The first message on the queue whose message type equals type is returned.
type < 0 The first message on the queue whose message type is the lowest value less than or equal to the absolute value of type is returned.

3a)

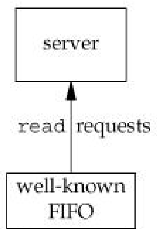
FIFO’s can be used to send data between a client and a server. If we have a server that is contacted by
numerous clients, each client can write its request to a well-known FIFO that the server creates. Since there
are multiple writers for the FIFO, the requests sent by the clients to the server need to be less than PIPE_BUF
bytes in size.

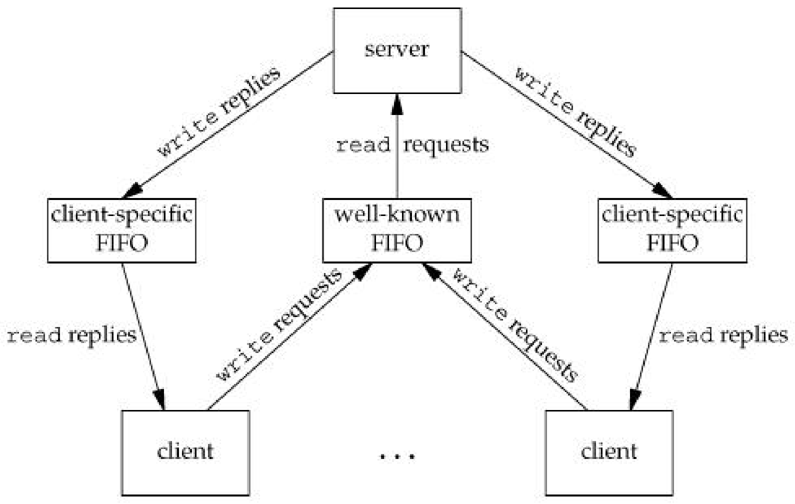
This prevents any interleaving of the client writes. The problem in using FIFOs for this type of client server
communication is how to send replies back from the server to each client.

A single FIFO can’t be used, as the clients would never know when to read their response versus responses
for other clients. One solution is for each client to send its process ID with the request. The server then
creates a unique FIFO for each client, using a pathname based on the client’sprocess ID.

For example, the server can create a FIFO with the name /vtu/ ser.XXXXX, where XXXXX is replaced with the
client’s process ID. This arrangement works, although it is impossible for the server to tell whether a client
crashes. This causes the client-specific FIFOs to be left in the file system.

The server also must catch SIGPIPE, since it’s possible for a client to send a request and terminate before
reading the response, leaving the client-specific FIFO with one writer (the server) and no reader.





4.

Sockets allow communication between two different processes on the same or different machines. To be more precise, it's a way to talk to other computers using standard Unix file descriptors. In Unix, every I/O action is done by writing or reading a file descriptor. A file descriptor is just an integer associated with an open file and it can be a network connection, a text file, a terminal, or something else.

To a programmer, a socket looks and behaves much like a low-level file descriptor. This is because commands such as read() and write() work with sockets in the same way they do with files and pipes.

isten()

NAME

listen() - listen for connections on a socket

SYNOPSIS

#include <sys/socket.h>

int listen(int sockfd, int backlog);

- ❒ sockfd is the usual socket file descriptor from the socket() system call.
- ❒ backlog is the number of connections allowed on the incoming queue.
- ❒ As an example, for the server, if you want to wait for incoming connections and handle them in some way, the steps are: first you listen(), then you accept().

- The incoming connections are going to wait in this queue until you accept() (explained later) them and this is the limit on how many can queue up.
- Again, as per usual, listen() returns -1 and sets errno on error.
- We need to call bind() before we call listen() or the kernel will have us listening on a random port.
- So if you're going to be listening for incoming connections, the sequence of system calls you'll make is something like this:

```
socket();

bind();

listen();

/* accept() goes here */
```

accept()

NAME

accept() - accept a connection on a socket

SYNOPSIS

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

#include <sys/socket.h>

int accept(int sockfd, struct sockaddr *addr, int *addrlen);
```

- sockfd is the listen()ing socket descriptor.
- addr will usually be a pointer to a local struct sockaddr_in. This is where the information about the incoming connection will go (and with it you can determine which host is calling you from which port).
- addrlen is a local integer variable that should be set to sizeof(struct sockaddr_in) before its address is passed to accept().
- accept() will not put more than that many bytes into addr. If it puts fewer in, it'll change the value of addrlen to reflect that.
- accept() returns -1 and sets errno if an error occurs.
- Basically, after listen(), a server calls accept() to wait for the next client to connect. accept() will create a new socket to be used for I/O with the new client. The server then will continue to do further accepts with the original sockfd.
- When someone try to connect() to your machine on a port that you are listen()ing on, their connection will be queued up waiting to be accepted. You call accept() and you tell it to get the pending connection.

- ❒ It'll return to you a new socket file descriptor to use for this single connection.
- ❒ Then, you will have two socket file descriptors where the original one is still listening on your port and the newly created one is finally ready to send() and recv().
- ❒ The following is a program example that demonstrates the use of the previous functions.

send()

int send(int sockfd, const void *msg, int len, int flags);

- ❒ sockfd is the socket descriptor you want to send data to (whether it's the one returned by socket() or the new one you got with accept()).
- ❒ msg is a pointer to the data you want to send.
- ❒ len is the length of that data in bytes.
- ❒ Just set flags to 0. (See the send() man page for more information concerning flags).

recv()

- ❒ The recv() call is similar in many respects:

int recv(int sockfd, void *buf, int len, unsigned int flags);

- ❒ sockfd is the socket descriptor to read from, buf is the buffer to read the information into and len is the maximum length of the buffer.
- ❒ flags can again be set to 0. See the recv() man page for flag information.
- ❒ recv() returns the number of bytes actually read into the buffer, or -1 on error (with errno set, accordingly).
- ❒ If recv() return 0, this can mean only one thing that is the remote side has closed the connection on you. A return value of 0 is recv()'s way of letting you know this has occurred.
- ❒ At this stage you can now pass data back and forth on stream sockets.
- ❒ These two functions send() and recv() are for communicating over stream sockets or connected datagram sockets.
- ❒ If you want to use regular unconnected datagram sockets (UDP), you need to use the sendto() and recvfrom().
- ❒ Or you can use more general, the normal file system functions, write() and read().

write()

NAME

write() - write to a file descriptor

SYNOPSIS

#include <unistd.h>

ssize_t write(int fd, const void *buf, size_t count);

- ❒ Writes to files, devices, sockets etc.
- ❒ Normally data is copied to a system buffer and write occurs asynchronously.
- ❒ If buffers are full, write can block.

read()

NAME

read() - read from a file descriptor

SYNOPSIS

#include <unistd.h>

ssize_t read(int fd, void *buf, size_t count);

- ❒ Reads from files, devices, sockets etc.
- ❒ If a socket has data available up to count bytes are read.
- ❒ If no data is available, the read blocks.
- ❒ If less than count bytes are available, read returns what it can without blocking.
- ❒ For UDP, data is read in whole or partial datagrams. If you read part of a datagram, the rest is discarded.

close() and **shutdown()**

NAME

close() - close a file descriptor

SYNOPSIS

#include <unistd.h>

int close(int sockfd);

- ❒ You can just use the regular UNIX file descriptor close() function:

close(sockfd);

- ❒ This will prevent any more reads and writes to the socket. Anyone attempting to read or write the socket on the remote end will receive an error.

❒ UNIX keeps a count of the number of uses for an open file or device.

❒ Close decrements the use count. If the use count reaches 0, it is closed.

❒ Just in case you want a little more control over how the socket closes, you can use the shutdown() function.

❒ It allows you to cut off communication in a certain direction, or both ways just like close() does.

❒ The prototype:

int shutdown(int sockfd, int how);

❒ sockfd is the socket file descriptor you want to shutdown, and how is one of the following:

1. 0 – Further receives are disallowed.

2. 1 – Further sends are disallowed.

3. 2 – Further sends and receives are disallowed (like close()).

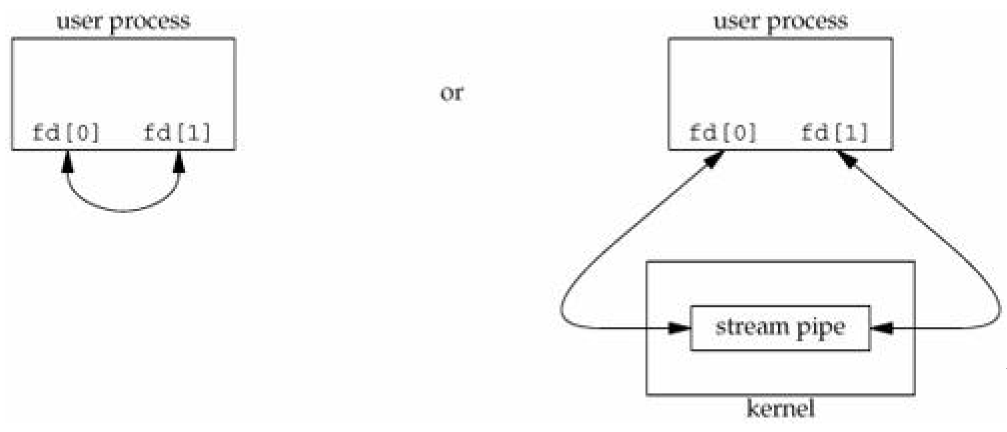
❒ shutdown() returns 0 on success, and -1 on error (with errno set accordingly).

❒ If you deign to use shutdown() on unconnected datagram sockets, it will simply make the socket unavailable for further send() and recv() calls (remember that you can use these if you connect() your datagram socket).

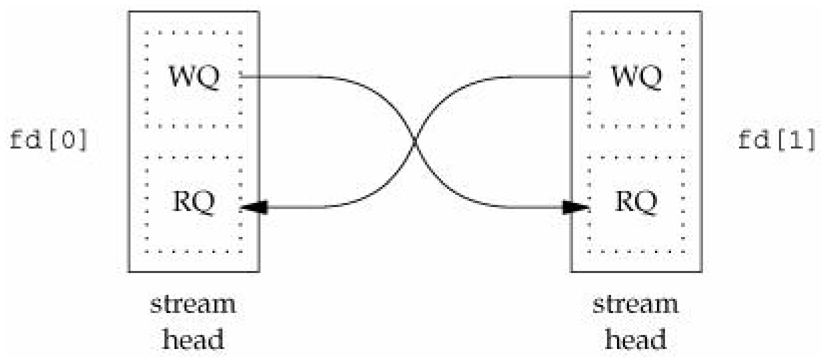
❒ It's important to note that shutdown() doesn't actually close the file descriptor, it just change its usability.

❒ To free a socket descriptor, you need to use close().

5a)
A STREAMS-based pipe ("STREAMS pipe," for short) is a bidirectional (full-duplex) pipe. To obtain bidirectional data flow between a parent and a child, only a single STREAMS pipe is required.



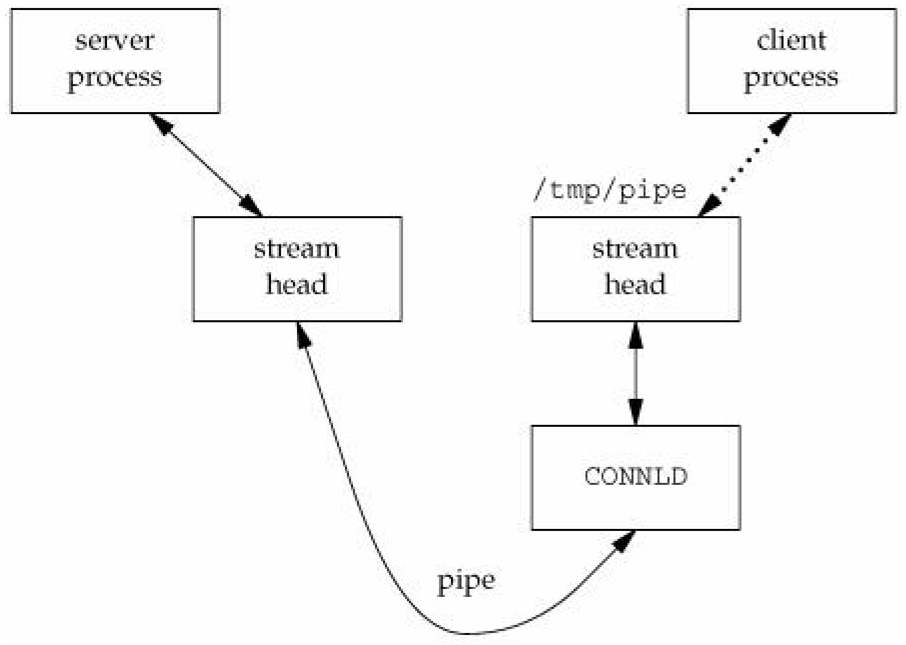
If we look inside a STREAMS pipe , we see that it is simply two stream heads, with each write queue (WQ) pointing at the other's read queue (RQ). Data written to one end of the pipe is placed in messages on the other's read queue.



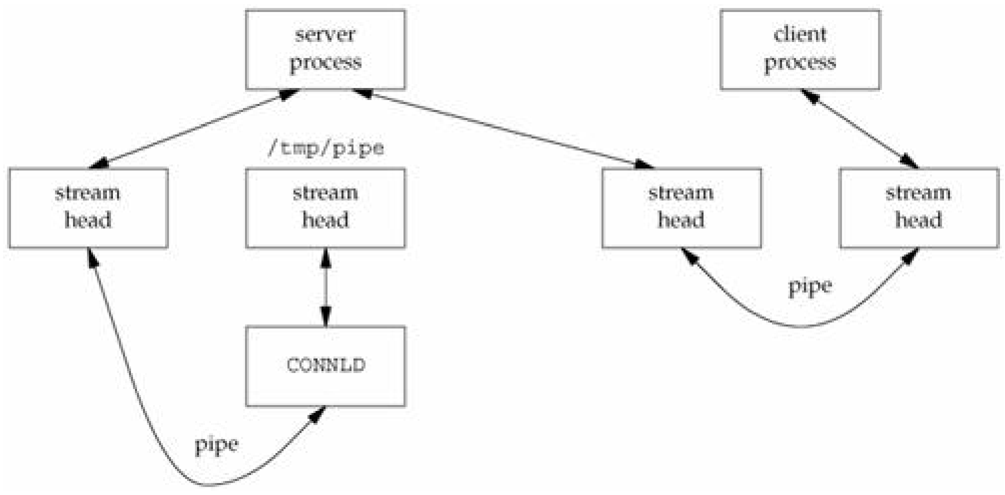
5b)

Data from one client will be interleaved with data from the other clients writing to the pipe. Even if we guarantee that the clients write less than PIPE_BUF bytes so that the writes are atomic, we have no way to write back to an individual client and guarantee that the intended client will read the message. With multiple clients reading from the same pipe, we cannot control which one will be scheduled and actually read what we send.

The connld STREAMS module solves this problem. Before attaching a STREAMS pipe to a name in the file system, a server process can push the connld module on the end of the pipe that is to be attached. This results in the configuration



the server process has attached one end of its pipe to the path /tmp/pipe. We show a dotted line to indicate a client process in the middle of opening the attached STREAMS pipe. Once the open completes, we have the configuration.



he client process never receives an open file descriptor for the end of the pipe that it opened. Instead, the operating system creates a new pipe and returns one end to the client process as the result of opening /tmp/pipe. The system sends the other end of the new pipe to the server process by passing its file descriptor over the existing (attached) pipe, resulting in a unique connection between the client process and the server process. We'll see the mechanics of passing file descriptors using STREAMS pipes

6a)
The first function to call is semget to obtain a semaphore ID.
#include <sys/sem.h>
int semget(key_t key, int nsems, int flag);
Returns: semaphore ID if OK, 1 on error
When a new set is created, the following members of the semid_ds structure are initialized.
    The ipc_perm structure is initialized. The mode member of this structure is set to the corresponding
permission bits of flag.
    sem_otime is set to 0.
    sem_ctime is set to the current time.
    sem_nsems is set to nsems.
The number of semaphores in the set is nsems. If a new set is being created (typically in the server), we must specify
nsems. If we are referencing an existing set (a client), we can specify nsems as 0.
The semctl function is the catchall for various semaphore operations.
#include <sys/sem.h>
int semctl(int semid, int semnum, int
cmd,... /* union semun arg */);
The fourth argument is optional, depending on the command requested, and if present, is of type semun , a union of
various command-specific arguments:
union semun
{
int
val;
struct semid_ds *buf;
unsigned short *array;
};
/* for SETVAL */

```
/* for IPC_STAT and IPC_SET */
/* for GETALL and SETALL *
```

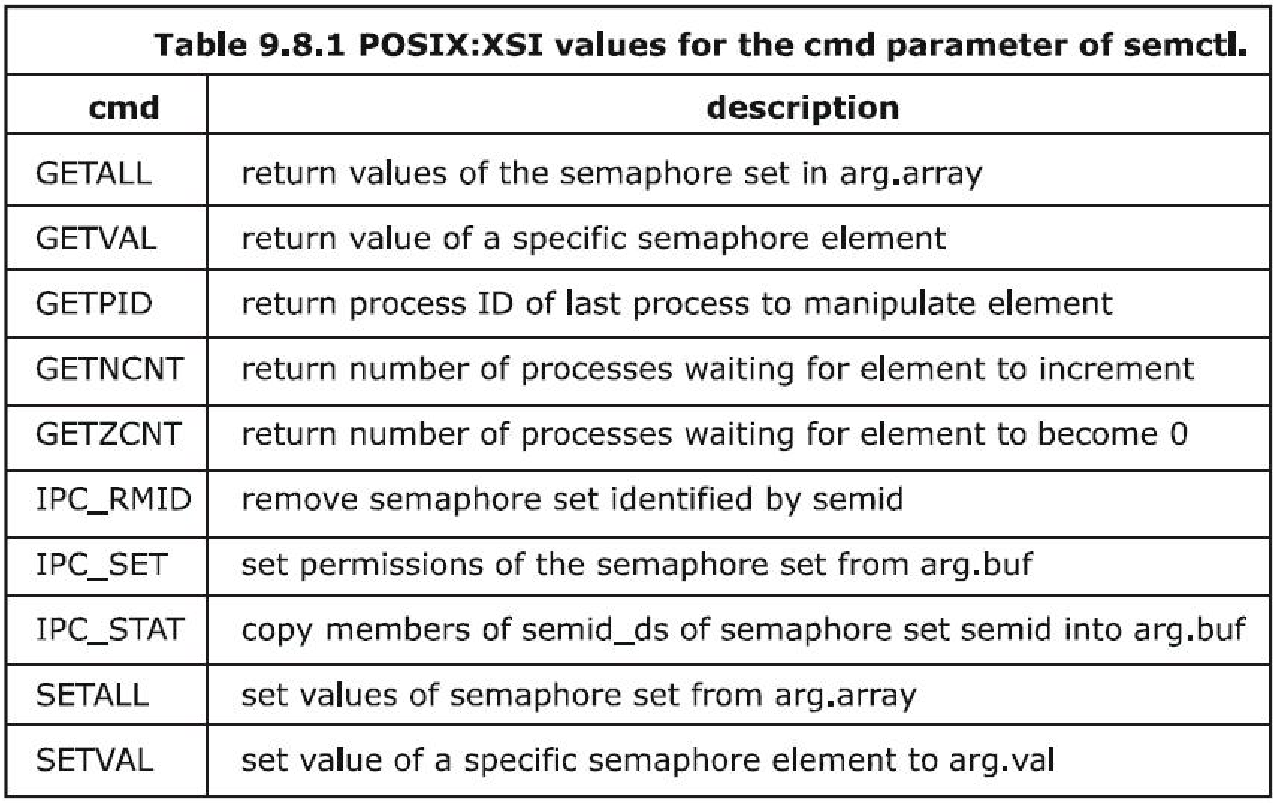


Table 9.8.1 POSIX:XSI values for the cmd parameter of semctl.

cmd	description
GETALL	return values of the semaphore set in arg.array
GETVAL	return value of a specific semaphore element
GETPID	return process ID of last process to manipulate element
GETNCNT	return number of processes waiting for element to increment
GETZCNT	return number of processes waiting for element to become 0
IPC_RMID	remove semaphore set identified by semid
IPC_SET	set permissions of the semaphore set from arg.buf
IPC_STAT	copy members of semid_ds of semaphore set semid into arg.buf
SETALL	set values of semaphore set from arg.array
SETVAL	set value of a specific semaphore element to arg.val

The cmd argument specifies one of the above ten commands to be performed on the set specified by semid.

The function semop atomically performs an array of operations on a semaphore set.

```
#include <sys/sem.h>

int semop(int semid, struct sembuf semoparray[], size_t nops);
```

Returns: 0 if OK, 1 on error.

The semoparray argument is a pointer to an array of semaphore operations, represented by sembuf structures:

```
struct sembuf {
unsigned short
short
short
};
sem_num;
sem_op;
sem_flg;
/* member # in set (0, 1, ..., nsems-1) */
/* operation (negative, 0, or positive) */
/* IPC_NOWAIT, SEM_UNDO */
```

The nops argument specifies the number of operations (elements) in the array.

The sem_op element operations are values specifying the amount by which the semaphore value is to be changed.

If sem_op is an integer greater than zero, semop adds the value to the corresponding semaphore element
value and awakens all processes that are waiting for the element to increase.
If sem_op is 0 and the semaphore element value is not 0, semop blocks the calling process (waiting for 0)
and increments the count of processes waiting for a zero value of that element.
If sem_op is a negative number, semop adds the sem_op value to the corresponding semaphore element
value provided that the result would not be negative. If the operation would make the element value negative, semop blocks the process on the event that the semaphore element value increases. If the resulting value is 0, semop wakes the processes waiting for 0.

6b)

```
#include "apue.h"
Int main(void)
{
int
n, int1, int2;
char
line[MAXLINE];
while ((n = read(STDIN_FILENO, line, MAXLINE)) > 0) {
line[n] = 0;
/* null terminate */
if (sscanf(line, "%d%d", &int1, &int2) == 2) {
sprintf(line, "%d\n", int1 + int2);
n = strlen(line);
if (write(STDOUT_FILENO, line, n) != n)
err_sys("write error");
} else {
if (write(STDOUT_FILENO, "invalid args\n", 13) != 13)
err_sys("write error");
}
}
exit(0);
}
```