CSC 33200 (L) - Operating Systems – Spring 2023

Lab 4: Inter-process Communications
Date: 03/10/2023

Some of the methods through which process can communicate are:

- Pipes
- Signals
- Message Queue
- Shared Memory

Pipes:
pipe() creates a pipe, a unidirectional data channel that can be used for interprocess communication. The array *pipefd* is used to return two file descriptors referring to the ends of the pipe.

pipefd[0] refers to the read end of the pipe. *pipefd[1]* refers to the write end of the pipe. Data written to the write end of the pipe is buffered by the kernel until it is read from the read end of the pipe.

int pipe(int fd[2]) -- creates a pipe and returns two file descriptors, fd[0], fd[1].

fd[0] is opened for reading,

fd[1] for writing.

We can use fork() to copy the same pipe into the parent and child process. Then both process can exchange data via pipe.

Signals:

The signals SIGKILL and SIGSTOP cannot be caught, blocked, or ignored. Some of the signals are:

SIGHUP 1 /* hangup */	SIGINT 2 /* interrupt */
SIGQUIT 3 /* quit */	SIGILL 4 /* illegal instruction */
SIGABRT 6 /* used by abort */	SIGKILL 9 /* hard kill */
SIGALRM 14 /* alarm clock */	
SIGCONT 19 /* continue a stopped process */	
SIGCHLD 20 /* to parent on child stop or exit */	

To view the list of signals you can use “man 7 signal” or “kill -l”

Send signal to a different process: int kill(int pid, int signal) - a system call that send a signal to a process, pid. If pid is greater than zero, the signal is sent to the process whose process ID is equal to pid. If pid is 0, the signal is sent to all processes, except system processes

kill() returns 0 for a successful call, -1 otherwise and sets errno accordingly.

Send signal to a running process: int raise(int sig) sends the signal sig to the executing program. raise() actually uses kill() to send the signal to the executing program:

```
    kill(getpid(), sig);
```

Unless caught or ignored, the kill signal terminates the process.

The SIGKILL signal cannot be caught or ignored and will always terminate a process.

Singal Handling:

```
sighandler_t signal(int signum, sighandler_t handler);
```

handler can have 3 values:

SIG_DFL

-- a pointer to a system default function SID_DFL(), which will terminate the process upon receipt of sig.

SIG_IGN

-- a pointer to system ignore function SIG_IGN() which will disregard the sig action (<u>UNLESS</u> it is SIGKILL).

A function address

-- a user specified function.

The signals SIGKILL and SIGSTOP cannot be caught or ignored.

Message Queue:

Two (or more) processes can exchange information via access to a common system message queue. The ***sending*** process places via some (OS) message-passing module a message onto a queue which can be read by another process. Each message is given an identification or type so that processes can select the appropriate message

Create a queue: msgget()

Send message: msgsend()

Receive message: msgrcv()

Queue: msquid = msgget(key, msgflag)

Key can be any arbitrary value.

Msgflag is similar to mode argument in open() where we define the permissions for the msgqueue.

The key is an arbitrary value or one that can be derived from a common seed at run time. One way is with ftok() , which converts a filename to a key value that is unique within the system.

Functions that initialize or get access to messages return an ID number of type int. IPC functions that perform read, write, and control operations use this ID.

If the key argument is specified as IPC_PRIVATE, the call initializes a new instance of an IPC facility that is private to the creating process. When the IPC_CREAT flag is supplied in the flags argument appropriate to the call, the function tries to create the facility if it does not exist already.

When called with both the IPC_CREAT and IPC_EXCL flags, the function fails if the facility already exists. This can be useful when more than one process might attempt to initialize the facility. One such case might involve several server processes having access to the same facility. If they all attempt to create the facility with IPC_EXCL in effect, only the first attempt succeeds. If neither of these flags is given and the facility already exists, the functions to get access simply return the ID of the facility.

If IPC_CREAT is omitted and the facility is not already initialized, the calls fail. These control flags are combined, using logical (bitwise) OR, with the octal permission modes to form the flags argument. For example, the statement below initializes a new message queue if the queue does not exist.

```
msqid = msgget(ftok("/filename",key), (IPC_CREAT | IPC_EXCL | 0400));
```

The first argument evaluates to a key based on the string ("/filename "). The second argument evaluates to the combined permissions and control flags.

The msgctl() function alters the permissions and other characteristics of a message queue. The owner or creator of a queue can change its ownership or permissions using msgctl() Also, any process with permission to do so can use msgctl() for control operations.

The msgctl() function is prototypes as follows:

```
int msgctl(int msqid, int cmd, struct msqid_ds *buf )
```

The msqid argument must be the ID of an existing message queue. The cmd argument is one of:

IPC_STAT
-- Place information about the status of the queue in the data structure pointed to by buf. The process must have read permission for this call to succeed.

IPC_SET
-- Set the owner's user and group ID, the permissions, and the size (in number of bytes) of the message queue. A process must have the effective user ID of the owner, creator, or superuser for this call to succeed.

IPC_RMID
-- Remove the message queue specified by the msqid argument.

The following code illustrates the msgctl() function with all its various flags:

```
if (msgctl(msqid, IPC_STAT, &buf) == -1) {
        perror("msgctl: msgctl failed");
        exit(1);

}
```

Msg send and receive:

Msg data:

```
struct mymsg {
    long      mtype;    /* message type must be long */
    char mtext[MSGSZ]; /* message text of length MSGSZ */
}
```

```
int msgsnd(int msqid, const void *msgp, size_t msgsz, int msgflg);

int msgrcv(int msqid, void *msgp, size_t msgsz, long msgtyp, int msgflg);
```

Shared Memory:

A process creates a shared memory segment using shmget()|. The original owner of a shared memory segment can assign ownership to another user with shmctl(). It can also revoke this assignment. Other processes with proper permission can perform various control functions on the shared memory segment using shmctl(). Once created, a shared segment can be attached to a process address space using shmat(). It can be detached using shmdt()

The attaching process must have the appropriate permissions for shmat(). Once attached, the process can read or write to the segment, as allowed by the permission requested in the attach operation. A shared segment can be attached multiple times by the same process. A shared memory segment is described by a control structure with a unique ID that points to an area of physical memory. The identifier of the segment is called the shmid

```
int shmget(key_t key, size_t size, int shmflg);
int shmctl(int shmid, int cmd, struct shmid_ds *buf);
void *shmat(int shmid, const void *shmaddr, int shmflg);

int shmdt(const void *shmaddr);
```

shmat() returns a pointer, shmaddr, to the head of the shared segment associated with a valid shmid. shmdt() detaches the shared memory segment located at the address indicated by shmaddr

Lab 4. Marks: 30 **Submission Deadline: 03/23/2023**

Part 1 Open a file called readme.txt in the child process, read the contents and pass to the parent process using Pipe. Parent process will write to readme.txt, "Parent is writing:" and write the contents it received from the child to the readme.txt file.

Marks: 5

Part 2 Write a program where a Parent process sends 2 signals to the child process, Child process catches two signals and prints out that it received the signal. And then child process terminates.

Marks: 5

Part 3

Write code using message queue to simulate a chat program where one program can send and receive data from another program in real time. For example, if you have progA.exe and progB.exe then – you can run both program and they will behave like this:

ProgA	ProgB
Proga: Can you see my message?	From ProgA: Can you see my message?
From ProgB: Yes I can.	ProgB: Yes I can
Proga: I can also see your message.	From ProgA: I can also see your message.

And so on.. Input for both program will be from the terminal. If one program writes "Bye" – both will terminate.

Marks: 10

Part4:

Write a program where a 3 process are attached to a shared memory.

Process A, writes 5 characters {A,A,A,A,A} to the shared memory. Then waits till the 1st character is changed to B by Process B , print out the new contents of the shared memory and then terminates.

Process B changes the first character to B => {B,A,A,A,A} and Then waits till the 1st Character is changed to C by Process C, print out the new contents and then terminates.

Process C changes the first character to C => {C,A,A,A,A} and Then detaches the shared memory and terminates.

There is no parent-child relationship between the processes.

Marks: 10

Submission Instructions

• All the programs MUST be clearly indented and internally documented

• Make sure your programs compile and run without any errors

• Only include c files or txt files for submission. Do not include any executables.

• Save all your programs with meaningful names and zip into a single folder as:
Lab4_[your last name here].zip (e.g., Lab4_Xyz.zip)

• Email your code with the subject line, "**Lab4-CSC33200–Class#G-*lastname***"

• Email: sdebnath@ccny.cuny.edu