4th Seminar

IPC

Multiple processes may be running on a machine and there is a need for these processes to communicate with each other, exchanging data or control information.
There are a few methods to accomplish this:

- Pipes
- Message Queues
- Signals
- Semaphores
- Shared Memory
- Sockets

IPC

The first two methods are used for communication between processes that exist on a single system, using byte streams.

This type of communication involves establishing a **one-way** channel between two processes. One of the processes writes bytes in the channel, the other reads the bytes, the access order being first-come first served.

Pipes

Piping is a process where the output of one process is made the input of another. We have seen examples of this from the UNIX command line using |.

`pipe()` -- Low level Piping

`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.

`pipe()` returns 0 on success, -1 on failure and sets errno accordingly.

The standard programming model is that after the pipe has been set up, two (or more) cooperative processes will be created by a fork and data will be passed using `read()` and `write()`.
Pipes opened with `pipe()` should be closed with `close(int fd)`.

Pipes

Pipe communication is **considered** to be **one-way** communication: either one process writes and the other process reads or vice-versa.
If both processes need to write and read from the pipe simultaneously, the solution is to use 2 pipes to establish two-way communication.

```
int pd[2];
pipe(pd);
if ( fork() == 0 )
  {          close(pdes[1]);
             read( pdes[0]); /* read from parent */
             .....          }
else
     {     close(pdes[0]);
             write( pdes[1]); /* write to child */
             .....          }
```

Example

```
#include <everything.h>
int main () {
    int a[] = {1,2,3,4}, pd[20];
    pipe(pd);
    if (fork()==0) {
    close(pd[0]);
    a[0]+=a[1];
    write(pd[1], &a[0],
sizeof(int));
    close(pd[1]);
    exit(0);
    }
```

```
    close(pd[1]);
    a[2]+=a[3];
    read(pd[0], &a[0], sizeof(int));
    close(pd[0]);
    wait(NULL);
    a[0]+=a[2];
    printf("Sum is %d\n", a[0]);
    return 0;
}
```

Exercise

Write a program that creates a child process; in the child read a number (0-10) and the parent try to guess if odd or even (parent should guess then read value) and print if correct or not.

```
#include <everything.h>
int main () {
    int pd[2];
    pipe(pd);
    if (fork()==0) {
    close(pd[0]);
    int number;
    printf("Give a value: ");
    scanf("%d",&number);
    write(pd[1], &number,
sizeof(int));
    close(pd[1]);
    exit(0);
    }
    close(pd[1]);
    int guess, number;
    srand(getpid());
    guess=rand()%10;
    read(pd[0],&number,sizeof(int));
    close(pd[0]);
    wait(NULL);
    if((number%2)==0&&(guess%2)==0)
    printf("Guessed even %d\n",
number);
    else
if((number%2)==1&&(guess%2)==1)
    printf("Guessed odd %d\n", number);
    else printf("Did not guess\n");
```

Dup2

A powerful capability in the shell is the ability to treat terminal input and output interchangeably with file input/output for most commands.
You can "duplicate" a file descriptor, or allocate another file descriptor that refers to the same open file as the original.
The major use of duplicating a file descriptor is to implement "redirection" of input or output: that is, to change the file or pipe that a particular file descriptor corresponds to.

- Function: `int dup (int OLD)`
  This function copies descriptor OLD to the first available descriptor number (the first number not currently open). It is equivalent to `fcntl (OLD, F_DUPFD, 0)`
- Function: `int dup2 (int OLD, int NEW)`
  This function copies the descriptor OLD to descriptor number NEW.

Dup2

If OLD is an invalid descriptor, then ‘dup2’ does nothing; it does not close NEW. Otherwise, the new duplicate of OLD replaces any previous meaning of descriptor NEW, as if NEW were closed first.
If OLD and NEW are different numbers, and OLD is a valid descriptor number, then ‘dup2’ is equivalent to:

```
close (NEW);
fcntl (OLD, F_DUPFD, NEW)
```

However, `dup2' does this atomically; there is no instant in the middle of calling `dup2' at which NEW is closed and not yet a duplicate of OLD.
Makes newfd be the copy of oldfd, closing newfd first if necessary, but note the following:

* If oldfd is not a valid file descriptor, then the call fails, and newfd is not closed.
* If oldfd is a valid file descriptor, and newfd has the same value as oldfd, then dup2() does nothing, and returns newfd.

Dup2

After a successful return from one of these system calls, the old and new file descriptors may be used interchangeably. They refer to the same open file description (see open(2)) and thus share file offset and file status flags; for example, if the file offset is modified by using lseek(2) on one of the descriptors, the offset is also changed for the other.
The two descriptors do not share file descriptor flags (the close-on-exec flag).

Example

```
main (){
    int p[2];
    pipe (p);
    if (fork () == 0) {
     dup2 (p[1], 1);
     close (p[0]);
     execlp ("who", "who", 0);
     }
    else if (fork () == 0) {
     dup2 (p[0], 0);
     close (p[1]);
     execlp ("sort", "sort",0);
     }
    else {
        close (p[0]);
        close (p[1]);
        wait (0);
        wait (0);
        }
}
```

Popen

Pipes can be used to send data to or receive data from a program being run as a **subprocess**. One way of doing this is by using a combination of `pipe' (to create the pipe), `fork' (to create the subprocess), `dup2' (to force the subprocess to use the pipe as its standard input or output channel), and `exec' (to execute the new program).

Or, you can use `popen' and `pclose'.

The advantage of using `popen' and `pclose' is that the interface is much simpler and easier to use. But it doesn't offer as much flexibility as using the low-level functions directly.

Popen

- Function: `FILE * popen (const char *COMMAND, const char *MODE)`

It executes the shell command COMMAND as a subprocess. However, instead of waiting for the command to complete, it creates a pipe to the subprocess and returns a stream that corresponds to that pipe.
If you specify a MODE argument of "r", you can read from the stream to retrieve data from the standard output channel of the subprocess. The subprocess inherits its standard input channel from the parent process.
Similarly, if you specify a MODE argument of "w", you can write to the stream to send data to the standard input channel of the subprocess. The subprocess inherits its standard output channel from the parent process.
In the event of an error `popen' returns a null pointer. This might happen if the pipe or stream cannot be created, if the subprocess cannot be forked, or if the program cannot be executed.

Popen

- Function: `int pclose (FILE *STREAM)` The `pclose' function is used to close a stream created by `popen'. It waits for the child process to terminate and returns its status value, as for the `system' function.

Example

```
#include <everything.h>
int main(){
	FILE *w2p;
	FILE *p2s;
	w2p=popen("who","r");
	p2s=popen("sort","w");
	char  line[50];
	while (fgets(line,50,w2p)){
		fprintf(p2s,"%s",line);
		}
	fclose(w2p);
	fclose(p2s);
}
```

Exercise

Sort the content of a file given as cmd line parameter using popen

```
#include <everything.h>
int main(int argc, char* argv[]){
    char cmd[100];
    FILE *p1;
    FILE *p2; //parent to sort
    strcpy(cmd,"cat ");
    strcat(cmd,argv[1]);
    p1=popen(cmd,"r");
    p2=popen("sort","w");
    char  line[50];
    while (fgets(line,50,p1)){
            fprintf(p2,"%s",line);
            }
    fclose(p1);
    fclose(p2);
}
```

Named Pipes (FIFO)

A named pipe (also known as a FIFO) is one of the IPC methods.
It is an extension to the traditional pipe concept on Unix. A traditional pipe is “unnamed” and exists within the creator process.
A named pipe, however, can last as long as the system is up, beyond the life of the process. It can be deleted if no longer used.
Usually a named pipe appears as a file, and generally processes attach to it for inter-process communication. A FIFO file is a special kind of file on the local storage which allows two or more processes to communicate with each other by reading/writing to/from this file.
A FIFO special file is entered into the filesystem by calling mkfifo() in C (or mkfifo in shell). Once created, any process can open it for reading or writing, in the same way as an ordinary file. However, it has to be open at both ends simultaneously before you can proceed to do any input or output operations on it.

FIFO

```
int mkfifo(const char *pathname, mode_t mode);
```

makes a FIFO special file with name pathname. Here mode specifies the FIFO's permissions. It is modified by the process's umask in the usual way: the permissions of the created file are (mode & ~umask).

As named pipe(FIFO) is a kind of file, we can use all the system calls associated with it i.e. open, read, write, close.

Example

```
#include <Everything.h>
int main()
{
        int fd;
        char * myfifo = "/tmp/myfifo";
        mkfifo(myfifo, 0666);
        char arr1[80], arr2[80];
        while (1)
        {
        fd = open(myfifo, O_WRONLY);
        fgets(arr2, 80, stdin);
        write(fd, arr2, strlen(arr2)+1);
        close(fd);
        fd = open(myfifo, O_RDONLY);
        read(fd, arr1, sizeof(arr1));
        printf("User2: %s\n", arr1);
        close(fd);
        }
        return 0;
}
```

```
#include <Everything.h>
int main()
{
        int fd1;
        char * myfifo = "/tmp/myfifo";
        mkfifo(myfifo, 0666);
        char str1[80], str2[80];
        while (1)
        {
        fd1 = open(myfifo,O_RDONLY);
        read(fd1, str1, 80);
        printf("User1: %s\n", str1);
        close(fd1);
        fd1 = open(myfifo,O_WRONLY);
        fgets(str2, 80, stdin);
        write(fd1, str2, strlen(str2)+1);
        close(fd1);
        }
        return 0;
}
```

SHM

Efficient means of sharing data between programs. One program creates a memory portion which other processes (if permitted) can access.
A process creates a shared memory segment using shmget()
The original owner of a shared memory segment can assign or revoke ownership to another user with shmctl(). 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.

SHM

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. The structure definition for the shared memory segment control structures and prototypes can be found in <sys/shm.h>

shmget()

used to obtain access to a shared memory segment.

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

The key argument is an access value associated with the shared memory segment ID. The size argument is the size in bytes of the requested shared memory. The shmflg argument specifies the initial access permissions and creation control flags.
When the call succeeds, it returns the shared memory segment ID. This call is also used to get the ID of an existing shared segment (from a process requesting sharing of some existing memory portion).

shmctl()

used to alter the permissions and other characteristics of a shared memory segment.

```
int shmctl(int shmid, int cmd, struct shmid_ds *buf);
```

The process must have an effective shmid of owner, creator or superuser to perform this command. The cmd argument is one of following control commands:
SHM_LOCK -- Locks the specified shared memory segment in memory.
SHM_UNLOCK -- Unlocks the shared memory segment.
IPC_STAT -- Return the status information contained in the control structure and place it in the buffer pointed to by buf.
IPC_SET -- Set the effective user and group identification and access permissions.
IPC_RMID -- Remove the shared memory segment.

shmat() and shmdt()

shmat() and shmdt() are used to attach and detach shared memory segments.

```
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

Example

```
#include <Everything.h>
#define SHMSZ   27
int main()
{
        char c,*shm, *s;
        int shmid;
        key_t key = 5678;
        if ((shmid = shmget(key, SHMSZ, IPC_CREAT | 0666)) < 0)
{
        perror("shmget");
        exit(1);
        }
        if ((shm = shmat(shmid, NULL, 0)) == (char *) -1)
{
        perror("shmat");
        exit(1);
        }
        s = shm;
        for (c = 'a'; c <= 'z'; c++)
        *s++ = c;
        *s = NULL;
        while (*shm != '*')
        sleep(1);
        return 0;
}
```

Example

```
#include <Everything.h>
#define SHMSZ   27
int main()
{
        int shmid;
        key_t key = 5678;
        char *shm, *s;
        if ((shmid = shmget(key, SHMSZ, 0666)) < 0) {
                perror("shmget");
                exit(1);
        }
        if ((shm = shmat(shmid, NULL, 0)) == (char *) -1) {
                perror("shmat");
                exit(1);
        }
        for (s = shm; *s != NULL; s++)
                putchar(*s);
        putchar('\n');
        *shm = '*';
        return(0);
}
```