CL2006 - Operating Systems Spring 2024
LAB # 6 MANUAL (Common)

Please note that all labs' topics including pre-lab, in-lab and post-lab exercises are part of the theory and labsyllabus. These topics will be part of your Midterms and Final Exams of lab and theory.

Objectives:

Demonstrate System Calls Programmatically:

1. Develop an understanding of system calls by implementing various functions programmatically and explore the functionality and usage of each system call through practical coding exercises, emphasizing their role in ordinary pipes, named pipes and shared memory.

2. Implement C program producing and consuming data using ordinary and names pipes and shared memory by utilizing appropriate system calls.

Lab Tasks:

Use of system calls related to the following:
1. Ordinary Pipes
2. Named Pipes
3. Shared Memory.

Delivery of Lab contents:

Strictly following the following content delivery strategy. Ask students to take notes during the lab.

1st Hour
- Experiment 6a

2nd Hour
- Experiment 6b

3rd Hour
- Experiment 6c

Initial Document: Hamza Yousuf (January 2019). This version by: Nadeem Kafi (23/02/2024)
DEPARTMENT OF COMPUTER SCEICEN, FAST-NU, KARACHI

** ChatGPT is heavily used to make the contents of this document along with other Internet sources.

EXPERIMENT 6a

InterProcess Communication using Pipes

OBJECTIVE:

- Learn and Understand InterProcess Communication using implementation of Pipes

Pipes:

Ordinary pipes allow two processes to communicate in standard producer consumer fashion:the producer writes to one end of the pipe (the **write-end)** and the consumer reads from the other end (the **read-end). As a result, ordinary** pipes are unidirectional, allowing only one-way communication. If two-way communication is required, two pipes must be used, with each pipe sending data in a different direction.

- A pipe has a read end and a write end.
- Data written to the write end of a pipe can be read from the rear end ofthe pipe.

Creating an Ordinary Pipe:

```c
#include <sys/types.h>
#include <stdio.h>
#include <string.h>
#include <unistd.h>
#define BUFFER_SIZE 25
#define READ_END 0
#define WRITE_END 1
int main(void) {
    char write_msg[BUFFER_SIZE] = "Greetings";
    char read_msg[BUFFER_SIZE];
    int fd[2];
    pid_t pid;

    if (pipe(fd) == -1) { // create a pipe
        fprintf(stderr, "Pipe failed");
        return 1;
    }
    pid = fork();
    if (pid < 0) { /* error occurred */
        fprintf(stderr, "Fork Failed");
        return 1;
    }
    if (pid > 0) { /* parent process */
        close(fd[READ_END]);
        write(fd[WRITE_END], write_msg, strlen(write_msg) + 1);
        close(fd[WRITE_END]);
    }
    else { /* child process */
        close(fd[WRITE_END]); // close the unused end of the pipe
        read(fd[READ_END], read_msg, BUFFER_SIZE); // read from the pipe
        printf("read %s", read_msg);
        close(fd[READ_END]); // close the read end of the pipe
    }
    return 0;
}
```

In-Lab

- Compile and execute the above code. Open a separate terminal window to note the process ids using in this code.
- Modify the code such that it tasks write_msg value from user terminal.

EXPERIMENT 6b
Named Pipes

OBJECTIVE:
- Learn and execute InterProcess Communication using implementation of named pipes.

Named Pipes:

- It is an extension to the traditional pipe concept on Unix. A traditional pipe is "unnamed" and lasts only as long as the 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. Once we have created a FIFO special file in this way, any process can open it for reading or writing, in the same way as an ordinary file. However, it must be open at both ends simultaneously before you can proceed to do any input or output operations on it.
- Reading from or writing to a named pipe occurs just like traditional file reading and writing; except that the data for named pipe is never written to or read from a file inhard disk but memory.

Producer code using Named pipes:

```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <fcntl.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <string.h>

#define FIFO_FILE "/tmp/myfifo"

int main() {
    int fd;
    char buffer[BUFSIZ];
    ssize_t num_bytes;

    mkfifo(FIFO_FILE, 0666);         // Create the named pipe (FIFO)
    fd = open(FIFO_FILE, O_WRONLY); // Open the named pipe for writing (producer)
    if (fd == -1) {
        perror("open");
        exit(EXIT_FAILURE);
    }
    while (1) { // Producer loop
        printf("Producer: Enter a message (or 'exit' to quit): ");
        fgets(buffer, BUFSIZ, stdin);
        num_bytes = write(fd, buffer, strlen(buffer)); // Write input to the named pipe
        if (num_bytes == -1) {
            perror("write");
            exit(EXIT_FAILURE);
        }
        if (strncmp(buffer, "exit", 4) == 0) { // Check for exit condition
            break;
        }
    }
    close(fd);          // Close the named pipe
    unlink(FIFO_FILE); // Remove the named pipe from the file system

    return 0;
}
```

In-Lab:

1. Compile and execute the producer code above that uses named pipe. Correct errors, if any.
2. Carefully study the producer code and verify that the FIFO_FILE has been created.
3. Now, write consumer code for the above producer using the following hints:
   - Open FIFO_FILE in read only mode using O_RDONLY
   - Consumer read the data using num_bytes = read(fd, buffer, BUFSIZ);
   - Display the data sent by the producer using
     printf("Consumer: Received message: %s", buffer);

EXPERIMENT 6c

InterProcess Communication using Shared Memory

OBJECTIVE:

- Learn and Understand InterProcess Communication using implementation of Shared Memory

Shared Memory:

InterProcess Communication through shared memory is a concept where two or more processes can access the common memory. And communication is done via this shared memory where changes made by one process can be viewed by another process.

Producer process using POSIX shared memory API:

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <fcntl.h>
#include <sys/shm.h>
#include <sys/stat.h>
#include <sys/mman.h>
#include <unistd.h>

int main() {
    const int SIZE = 4096;          // the size (in bytes) of shared memory object
    const char *name = "OS";        // name of the shared memory object
    const char *message_0 = "Hello"; // strings written to shared memory
    const char *message_1 = "World!";
    int fd;     // shared memory file descriptor
    char *ptr; // pointer to shared memory obect

    fd = shm_open(name, O_CREAT | O_RDWR, 0666); // create the shared memory object
    ftruncate(fd, SIZE); // configure the size of the shared memory object
    // memory map the shared memory object
    ptr = (char *)mmap(0, SIZE, PROT_READ | PROT_WRITE, MAP_SHARED, fd, 0);
    // write to the shared memory object
    sprintf(ptr, "%s", message_0);
    ptr += strlen(message_0);
    sprintf(ptr, "%s", message_1);
    ptr += strlen(message_1);

    return 0;
}
```

```
gcc test.c -o test -lrt
```

Consumer process using POSIX shared-memory API:

```c
#include <stdio.h>
#include <stdlib.h>
#include <fcntl.h>
#include <sys/shm.h>
#include <sys/stat.h>
#include <sys/mman.h>
int main() {
    const int SIZE = 4096;
    const char *name = "OS";
    int fd;
    char *ptr;
    fd = shm_open(name, O_RDONLY, 0666);
    ptr = (char *)mmap(0, SIZE, PROT_READ | PROT_WRITE, MAP_SHARED, fd, 0);
    printf("%s", (char *)ptr); // read from the shared memory object
    shm_unlink(name);          // remove the shared memory object
    return 0;
}
```

In-lab

- Compile and execute producer code.
- Compile and execute the consumer code. It will give a run-time error. Why?
- Now modify the producer code such that the producer and the consumer code run concurrently as two processes. Now the consumer code will print the message.
- Now modify the consumer code to print the second message.

----------(X)----------