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

Internal Assessment Test 5 – FEB 2022

Sub:	Unix Programming					Sub Code:	18CS56	Branch:	CSE
Date:	07/02/2022	Duration:	90 mins	Max Marks:	50	Sem / Sec:	V/A,B&C		OBE

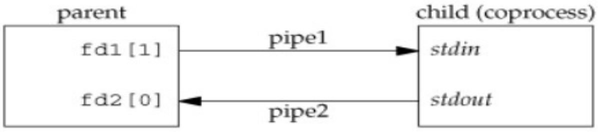
Answer any FIVE FULL Questions | MARKS | CO | RBT

1 ***Explain coprocess with an example program.*** [10] CO3 L3

Filters are programs that take plain text (either stored in a file or produced by another program) as standard input, transforms it into a meaningful format, and then returns it as standard output.

For example a filter copies standard input to standard output, converting any uppercase character to lowercase.

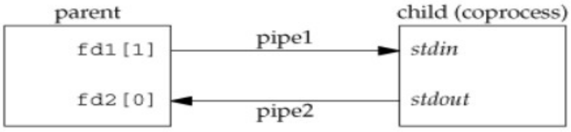
Filters are normally connected linearly in shell pipelines. The process creates two pipes: one is the standard input of the coprocess, and the other is the standard output of the coprocess.



A filter becomes a coprocess when the same program generates the filter's input and reads the filter's output.

The simple coprocess reads two numbers from its standard input, computes their sum, and writes the sum to its standard output.

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>
#define MAXLINE 80
int main(void)
{
        int n, int1, int2;
        char line[MAXLINE];
        while ((n = read(0, 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(1, line, n) != n) printf("write error");
                }
                else
                {
                        if (write(1, "invalid args\n", 13) != 13)
                        printf("write error");
```



```
                }
        }
        exit(0);
}

# include <stdio.h>
# include <stdlib.h>
# include <unistd.h>
# include <string.h>
#define MAXLINE 80

int
main(void)
{
        int n, fd1[2], fd2[2];
        pid_t pid;
        char line[MAXLINE];
        if (pipe(fd1) < 0 || pipe(fd2) < 0)
        printf("pipe error");
        if ((pid = fork()) < 0) {printf("fork error");}
        else if (pid > 0)
        {
                /* parent */
        close(fd1[0]);
        close(fd2[1]);
        while (fgets(line, MAXLINE, stdin) != NULL) {
        n = strlen(line);
        if (write(fd1[1], line, n) != n) printf("write error to pipe");
        if ((n = read(fd2[0], line, MAXLINE)) < 0) printf("read error from pipe");
        if (n == 0) {printf("child closed pipe");break;}
        line[n] = 0;
/* null terminate */
        if (fputs(line, stdout) == EOF) printf("fputs error"); }
        if (ferror(stdin)) printf("fgets error on stdin");
        exit(0);
        }
        else
        {
/* child */
        close(fd1[1]);
        close(fd2[0]);
        if (fd1[0] != 0)
        {
                if (dup2(fd1[0], 0) != 0)
                printf("dup2 error to stdin");
                //close(fd1[0]);
        }
        if (fd2[1] != 1) {
        if (dup2(fd2[1], 1) != 1)
        printf("dup2 error to stdout");
        //close(fd2[1]);
        }
        if (execl("./add2", "add2", (char *)0) < 0)printf("execl error");
        }
        exit(0);
}
```

2 ***Explain popen and pclose APIs.*** [10] CO3 L1

A common operation is to create a pipe to another process, to either read its output or send it input, the standard I/O library has provided the popen and pclose functions. These two functions handle the following tasks: creating a pipe, forking a child, closing the unused ends of the pipe, executing a shell to run the command, and waiting for the command to terminate.

The prototype of the functions are as follows:

```
FILE *popen(const char *cmdstring, const char *type);
Returns: file pointer if OK, NULL on error

int pclose(FILE *fp);
Returns: termination status of cmdstring, or -1 on error
```

The function popen does a fork and exec to execute the cmdstring, and returns a standard I/O file pointer. If type is "r", the file pointer is connected to the standard output of cmdstring. If type is "w", the file pointer is connected to the standard input of cmdstring. The pclose function closes the standard I/O stream, waits for the command to terminate, and returns the termination status of the shell.

An example program is shown below:

```
#include <stdio.h>
int main()
{
FILE *fp;
char line[130]; /* line of data from unix command*/
fp = popen("ls -l", "r"); /* Issue the command.
/* Read a line
while ( fgets( line, sizeof line, fp))
{
printf("%s", line);
}
pclose(fp);
return 0;
}
```

3a ***Differentiate between pipes and FIFOs.*** [05] CO3 L1

Pipes	Fifos
In PIPE, data transfer takes place between the child process and parent process. I	FIFO have multiple processes communicating through it, like multiple client-server application.
n PIPE, communication is among the process having a common ancestor (related process).	In FIFO, it is not necessary for the process having a common ancestor for communication (unrelated process).
PIPE is created by pipe () function.	FIFO is created by mkfifo () function.
PIPE is unidirectional.	FIFO is bi-directional. The same FIFO can be used for reading and writing.

3b ***Write a program to send data from parent process to child process using pipes.*** [05] CO3 L3

Normally, a pipe is created by a process, that process calls fork, and pipe is used between the parent and the child. A pipe is created by calling the pipe()

function. The prototype is as follows:

```
int pipe(int filedes[2]);
```

The function returns 0 on success and -1 on error.

```
Int main (void)
{
int n;
int fd[2];
pid_t pid;
char line[MAXLINE];
if (pipe(fd) < 0) printf(“Error in creating pipe\n”);
if ((pid = fork()) < 0) printf(“Error in creating process\n”);
else if (pid > 0)
{
close(fd[0]);
write(fd[1],”hello world\n”,12);
}
else
{
close(fd[1]);
n=read(fd[0], line,MAXLINE);
write(1, line, n);
}
exit(0);
}
```

4 ***Explain Sigsetjmp and Siglongjmp with example program.*** [10] CO4 L3

These functions provide Inter function goto capability

```
Int sigsetjmp (sigjmpbuf env, int save_sigmask);
Int siglongjmp (sigjmpbuf env, int ret_val);
```

Similar setjmp, except that save_sigmask which helps the calling process to save signal mask to the env.

Siglongjmp is called from a user defined signal handling functions.

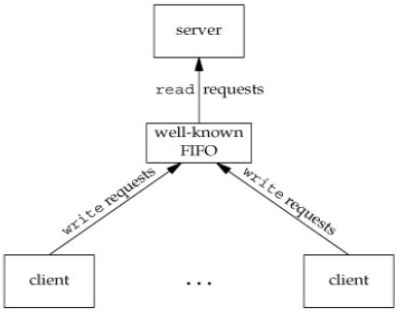
This is because a process signal mask is modified when a signal handler is called.

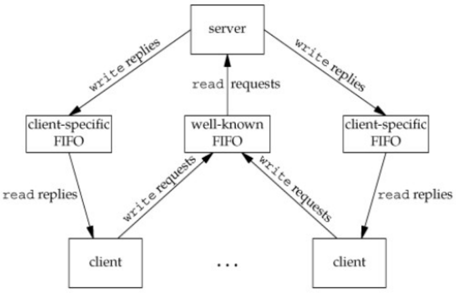
If user does not want to continue execution from a point where signal interruption occured.

Siglongjump should be called to ensure the process signal mask is restored properly when jumping out from a signal handling function.

```
# include <stdio.h>
# include <unistd.h>
# include <signal.h>
# include <setjmp.h>
# include <stdlib.h>


sigjmp_buf env;


void call_me(int sig_num)
{
        printf("Catch signal: %d\n",sig_num);
        siglongjmp(env,2);
}


int main()
{
        sigset_t sigmask;
        struct sigaction action, old_action;


        sigemptyset(&sigmask);

        if(sigaction(SIGINT,&action,&old_action)==-1)
        {
                printf("Error2\n");
        }


        if(sigsetjmp(env,1)!=0)
        {
                printf("return from signal interruption\n");
                return 0;
        }
        else
                printf("return from sigsetjmp\n");



        pause();
}
```

5 *Discuss client – server communication using FIFOs. Identify the limitations and suggest solutions.* [10] CO3 L2





FIFOs another means of inter-process communication in Unix. They are also called named pipes. Pipes can be used only between related processes when a common ancestor has created the pipe. With FIFOs, however, unrelated processes can exchange data. Another use for FIFOs is 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. The pathname of the FIFO must be known to all the clients that need to contact the server. 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 clientserver 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's process ID. The arrangement has the l

6 [10] CO4 L2

Explain in detail the basic coding rules for daemon process.

1.The first thing to do is call umask to set the file mode creation mask to 0. The file mode creation mask that's inherited could be set to deny certain permissions. If the daemon process is going to create files, it may want to set specific permissions. For example, if it specifically creates files with group-read and group-write enabled, a file mode creation mask that turns off either of these permissions would undo its efforts.

2. Call fork and have the parent exit . This does several things. The child inherits the process group ID of the parent but gets a new process ID, so we're guaranteed that the child is not a process group leader. This is a prerequisite for the call to setsid that is done next.

3. Call setsid() to create a new session. The three steps occur. The process (a) becomes a session leader of a new session, (b) becomes the process group leader of a new process group, and (c) has no controlling terminal.

4. Change the current working directory to the root directory. The current working directory inherited from the parent could be on a mounted file system. Since daemons normally exist until the system is rebooted, if the daemon stays on a mounted file system, that file system cannot be unmounted.

5. Alternatively, some daemons might change the current working directory to some specific location, where they will do all their work. For example, line printer spooling daemons often change to their spool directory.

6. Unneeded file descriptors should be closed. This prevents the daemon from holding open any descriptors that it may have inherited from its parent . We can use our open_max function or the getrlimit function to determine the highest descriptor and close all descriptors up to that value.

7. Some daemons open file descriptors 0, 1, and 2 to /dev/null so that any library routines that try to read from standard input or write to standard output or standard error will have no effect. Since the daemon is not associated with a terminal device, there is nowhere for output to be displayed; nor is there anywhere to receive input from an interactive user. Even if the daemonwas started from an interactive session, the daemon runs in the background, and the login session can terminate without affecting the daemon. If other users log in on the same terminal device, we wouldn't want output from the daemon showing up on the terminal, and the users wouldn't expect their input to be read by the daemon