**BIRLA INSTITUTE OF TECHNOLOGY & SCIENCE, PILANI (RAJASTHAN)**

**IS F462 – Network Programming**

**Lab#4**

**Topic: Signals, pipes**

**Note: please use programs under *code* directory supplied with this sheet. Do not copy from this sheet.**

**Signals:**

1. **alarm()**

Consider the following program. The program demonstrates the use of alarm() as a timer.

#include <signal.h>

long n;

void sigalrm(int signo){

alarm(1);

n=n+1;

printf("%d seconds elapsed\n",n);

}

main(){

signal(SIGALRM, sigalrm);

alarm(1);

while(1) pause();

}

Q?

1. Modify the program to implement sleep(long secs). The program should sleep for the specified number of seconds using alarm() and pause() calls.
2. **Synchronization using signals**

**//sync.c**

#include <sys/types.h>

#include <unistd.h>

#include <sys/wait.h>

#include <stdio.h>

main ()

{

int i = 0, j = 0;

pid\_t ret;

int status;

ret = fork ();

if (ret == 0)

{

for (i = 0; i < 5000; i++)

printf ("Child: %d\n", i);

printf ("Child ends\n");

}

else

{

wait (&status);

printf ("Parent resumes.\n");

for (j = 0; j < 5000; j++)

printf ("Parent: %d\n", j);

}

}

Q?

1. Synchronize the printing of child and parent in such way that child and parent print the lines alternatively? (hint: use pause() and kill() calls)
2. Examine when fork () is called, does the child inherit the signal mask of parent? Also examine when you do exec() , does the new program inherit the current signal mask? [Hint: Use sigprocmask() to print the singnal mask]
3. **Signal Blocking & Unblocking**

/\*critical.c\*/

#include <signal.h>

#include <stdlib.h>

#include <stdio.h>

void err\_sys(char\* str)

{

perror(str);

exit(-1);

}

static void sig\_quit(int);

int

main(void)

{

sigset\_t newmask, oldmask, pendmask;

if (signal(SIGQUIT, sig\_quit) == SIG\_ERR)

err\_sys("can't catch SIGQUIT");

sigemptyset(&newmask);

sigaddset(&newmask, SIGQUIT);

/\* block SIGQUIT and save current signal mask \*/

if (sigprocmask(SIG\_BLOCK, &newmask, &oldmask) < 0)

err\_sys("SIG\_BLOCK error");

/\* critical section starting\*/

sleep(5); /\* SIGQUIT here will remain pending \*/

if (sigpending(&pendmask) < 0)

err\_sys("sigpending error");

if (sigismember(&pendmask, SIGQUIT))

printf("\nSIGQUIT pending\n");

/\*critical section ending\*/

/\* reset signal mask which unblocks SIGQUIT \*/

if (sigprocmask(SIG\_SETMASK, &oldmask, NULL) < 0)

err\_sys("SIG\_SETMASK error");

printf("SIGQUIT unblocked\n");

sleep(5); /\* SIGQUIT here will terminate with core file \*/

exit(0);

}

static void

sig\_quit(int signo)

{

printf("caught SIGQUIT\n");

if (signal(SIGQUIT, SIG\_DFL) == SIG\_ERR)

err\_sys("can't reset SIGQUIT");

return;

}

Q?

1. Observe the blocking and unblocking of signals within the critical section. After executing the program, press Ctrl-\ and see. If it is pressed during critical section, the signal will remain pending. If it is pressed after critical section, the signal will be directly delivered.
2. Also block SIGINT signal while in critical section?
3. In the above program only SIGQUIT signal is blocked. Block all signals being delivered while you are in critical section?
4. In the above program, only SIGQUIT signal is being check for whether it is pending? How can you check all the signals that are pending?
5. **Pipes**

//pipe.c

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#include <sys/types.h>

#include <unistd.h>

#define MSGSIZE 16

main ()

{

int i;

char \*msg = "How are you?";

char inbuff[MSGSIZE];

int p[2];

pid\_t ret;

pipe (p);

ret = fork ();

if (ret > 0)

{

i = 0;

while (i < 10)

{

write (p[1], msg, MSGSIZE);

sleep (2);

read (p[0], inbuff, MSGSIZE);

printf ("Parent: %s\n", inbuff);

i++;

}

exit(1);

}

else

{

i = 0;

while (i < 10)

{

sleep (1);

read (p[0], inbuff, MSGSIZE);

printf ("Child: %s\n", inbuff);

write (p[1], "i am fine", strlen ("i am fine"));

i++;

}

}

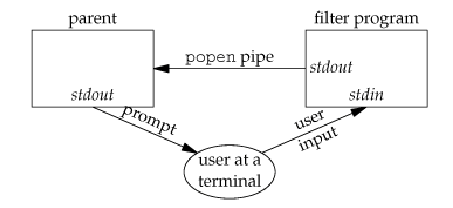
exit (0);

}

Q?

1. Check the output of the above program. Observe that using one pipe we can communicate both ways but in only one direction at a time.
2. Remove one of the sleep statements and see the output.
3. Remove both the sleep statements and see the output.
4. Try to make the above program synchronized i.e. only when the child completes its writing, parent writes data; child doesn’t write until parent completes writing.
5. **Application of Pipe**

The program needs to read only the upper case letters even though the user may enter the lowercase letters. For that the following design is considered. The filter program reads the characters from terminal and converts all of them into upper case and writes to output. Filter program is invoked by the program using popen().



/\*First compile filter program gcc filter.c –o filter\*/

/\*filter.c\*/

#include <ctype.h>

#include <stdio.h>

#include <stdlib.h>

void

err\_sys (char \*str)

{

perror (str);

exit (-1);

}

int

main (void)

{

int c;

while ((c = getchar ()) != EOF)

{

if (islower (c))

c = toupper (c);

if (putchar (c) == EOF)

err\_sys ("output error");

if (c == '\n')

fflush (stdout);

}

exit (0);

}

/\*parent.c\*/

#include <sys/wait.h>

#include <stdlib.h>

#include <stdio.h>

void err\_sys(char\* str)

{

perror(str);

exit(-1);

}

#define MAXLINE 80

int

main (void)

{

char line[MAXLINE];

FILE \*fpin;

if ((fpin = popen ("./filter", "r")) == NULL)

err\_sys ("popen error");

for (;;)

{

fputs ("prompt> ", stdout);

fflush (stdout);

if (fgets (line, MAXLINE, fpin) == NULL) /\* read from pipe \*/

break;

if (fputs (line, stdout) == EOF)

err\_sys ("fputs error to pipe");

}

if (pclose (fpin) == -1)

err\_sys ("pclose error");

putchar ('\n');

exit (0);

}

Q?

1. Observe the usage of popen. Using pipes we can filter the data that is coming from terminal into the program.
2. modify the program so that parent can accept only numbers. User may enter anything. But the program should read only the numbers.
3. **Pipelines**

Consider the following program for executing ls –l|wc –l.

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#include <sys/types.h>

#include <unistd.h>

main ()

{

int i;

int p[2];

pid\_t ret;

pipe (p);

ret = fork ();

if (ret == 0)

{

close (1);

dup (p[1]);

close (p[0]);

execlp ("ls", "ls", "-l", (char \*) 0);

}

if (ret > 0)

{

close (0);

dup (p[0]);

close (p[1]);

wait (NULL);

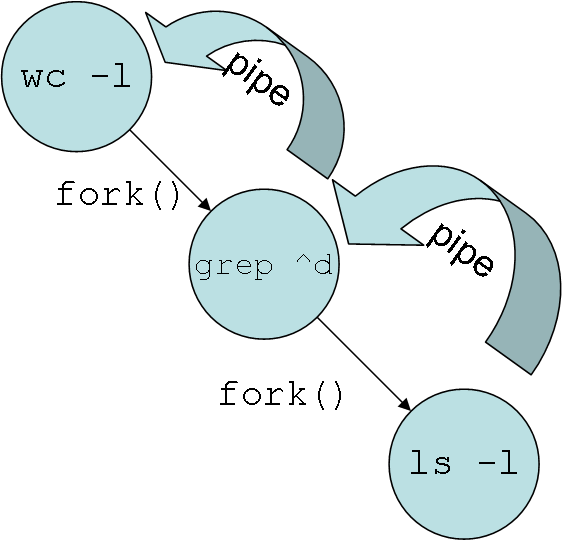
execlp ("wc", "wc", "-l", (char \*) 0);

}

}

Q?

1. Is wait required here? Why or why not?
2. Is it required to close unused ends of pipe? Suppose if you don’t close p[1] in parent will it have any effect? Find out.
3. dup2() system call can be used instead of dup(). Find out more on dup2() using man dup2 command. Dup2 is considered safer version of dup(). Can you see why is it so?
4. Modify the above program to execute ls –l| grep ^d |wc –l. The output of this should be the number of directories in the current directory.



1. **Coprocesses:**

The diagram shows the use of a child process loading an executable. Here the noticeable point is parent is supplying the input and the same parent is reading the output from the child. This requires two pipes to make the duplex communication possible. In such cases the child process is called as coprocess. The co-process can do variety of tasks like spell checking, validation, sorting, etc …



In the following example, we use a coprocess to validate emails. The emailValidate.c program is follows.

#include <stdio.h>

#include <string.h>

#define MAXSIZE 100

main()

{

char buf[MAXSIZE];int n;

while((n=read(0, buf, MAXSIZE))>0)

{

buf[n]='\0';

if(strstr(buf,"@")>0)

if(strstr(buf,".")>0)

write(1,"1\n",2);

else write(1,"-2\n",3);

else write(1,"-3\n",3);

}

}

It takes the string from stdin and writes the result to stdout. Now consider the following parent process.

main ()

{

int p1[2], p2[2];

int ret;

FILE \*fpi, \*fpo;

char line[MAXSIZE], result[MAXSIZE];

pipe (p1);

pipe (p2);

ret = fork ();

if (ret == 0)

{

close (p1[1]);

close (p2[0]);

dup2 (p1[0], 0);

dup2 (p2[1], 1);

execl ("./validateEmail", "validateEmail", (char \*) 0);

}

else

{

close (p1[0]);

close (p2[1]);

fpi = fopen ("emails.txt", "r");

fpo = fopen ("emails\_validation.txt", "w");

while (fgets (line, MAXSIZE, fpi) != NULL)

{

write (p1[1], line, strlen (line));

read (p2[0], result, MAXSIZE);

fprintf (fpo, "%s,%d\n", line, atoi (result));

}}}

The parent process takes help from the co-process validateEmail to check the emails.

Q?

* 1. Execute the above program. First compile validateEmailCoprocess.c to validateEmail executable. gcc validateEmailCoprocess.c –o validateEmail Then compile coprocess\_parent.c and run it. The emails.txt sample input is available here. Double click it to open. **.** Now parent is running, if we kill validateEmail child process by sending SIGKILL signal, find out what would happen? Modify the parent to recognize such unexpected situations and respond positively.
  2. Suppose the input is in order of millions. We want to concurrently process the validation. Modify the above program so that the there can be several coprocesses concurrently processing.
  3. Consider the following alteration in the validateEmail.c file. Compile it to validateEmail executable and run the parent.

main ()

{

char buf[MAXSIZE];

while (fgets (buf, MAXSIZE, stdin) != NULL)

{

if (strstr (buf, "@") > 0)

if (strstr (buf, ".") > 0)

printf ("1\n");

else

printf ("-2\n");

else

printf ("-3\n");

}

}

What did you notice? Why the program doesn’t work? Find out. [hint: Effect of using low-level system calls like read(), write() with standard i/o functions like printf(), scanf() etc]

**/\* End of Lab4 \*/**