# Slip 1

// Assignment : Write a C program to send SIGALRM signal by child process to parent process and parent process make a provision to catch the signal

//and display alarm is fired.(Use Kill, fork, signal and sleep system call)

#include <signal.h> #include <stdio.h> #include <stdlib.h> #include <unistd.h> #include <sys/wait.h>

void signalHandler(int signal) // this is signal handler for

{

if (signal == SIGALRM)

{

printf("Ding!\n"); wait(NULL);

}

}

int main(int argc, char \*argv[])

{

signal(SIGALRM, signalHandler); if (argc != 2)

{

printf("Invalid arguments\n"); return 0;

}

printf("Alarm application starting\n"); int delay;

sscanf(argv[1], "%d", &delay); // compute delay

if (fork() == 0) // start child process

{

printf("Waiting for alarm to go off\n"); sleep(delay);

kill(getppid(), SIGALRM); exit(0);

}

wait(NULL);

printf("done\n");

}

/\*

$root: gcc Slip1.c

$root: ./a.out 5

\*/

# Slip 2

// Assignment : Write a C program that catches the ctrl-c (SIGINT) signal for the first time and display the appropriate message and exits on pressing ctrl-c again

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

void sigint()

{

write(STDOUT\_FILENO, "Press Ctrl + C once again to exit",1);

signal(SIGINT, SIG\_DFL);

}

void main()

{

signal(SIGINT, sigint); while(1)

{

printf("Hello");

}

}

# Slip 3

// Assignment : Write a C program which creates a child process to run linux/ unix command or any user defined program.

//The parent process set the signal handler for death of child signal and Alarm signal.

//If a child process does not complete its execution in 5 second then parent process kills child process.

#include <stdio.h> #include <signal.h> #include <stdlib.h> #include <unistd.h> #include <sys/types.h>

// function declaration of sighup, sigint and sigquit functions

void sighup(); void sigint(); void sigquit();

// main function or driver code void main()

{

int pid;

// pid variable, which will be used later to identify the process, whether it is child process or parent process

// to get the child process if ((pid = fork()) < 0)

{

perror("fork"); exit(1);

}

if (pid == 0)

{ /\* child process, since pid equals to zero for child process \*/ signal(SIGHUP, sighup);

signal(SIGINT, sigint); signal(SIGQUIT, sigquit);

for (;;); /\* infinite loop i.e. loop for ever \*/

}

else /\* parent process\*/

{ // pid hold the process id of child process

printf("\nPARENT: sending SIGHUP\n\n"); kill(pid, SIGHUP);

sleep(3); // pause for 3 seconds

printf("\nPARENT: sending SIGINT\n\n"); kill(pid, SIGINT);

sleep(3); // pause for 3 seconds

printf("\nPARENT: Waiting for 5 Second then kill child\n\n"); printf("\nPARENT: sending SIGQUIT\n\n");

kill(pid, SIGQUIT);

sleep(5); // pause for 5 seconds

}

}

// function definition of sighup() void sighup()

{

signal(SIGHUP, sighup); /\* reset signal \*/

printf("CHILD: I have received a SIGHUP\n");

}

// function definition of sigint() void sigint()

{

signal(SIGINT, sigint); /\* reset signal \*/

printf("CHILD: I have received a SIGINT\n");

}

// function definition of sigquit() void sigquit()

{

printf("My Papa has Killed me!!!\n"); exit(0);

}

# Slip 4

// Assignment : Write a C program which creates a child process and child process catches a signal

//SIGHUP, SIGINT and SIGQUIT. The Parent process send a SIGHUP or SIGINT signal after every 3 seconds, at the end of 15 second parent send SIGQUIT signal to

//child and child terminates by displaying message “My Papa has Killed me!!!”.

#include <stdio.h> #include <signal.h> #include <stdlib.h> #include <unistd.h> #include <sys/types.h>

// function declaration of sighup, sigint and sigquit functions void sighup();

void sigint(); void sigquit();

// main function or driver code void main()

{

int pid;

// pid variable, which will be used later to identify the process, whether it is child process or parent process

// to get the child process if ((pid = fork()) < 0)

{

perror("fork"); exit(1);

}

if (pid == 0)

{ /\* child process, since pid equals to zero for child process \*/ signal(SIGHUP, sighup);

signal(SIGINT, sigint); signal(SIGQUIT, sigquit);

for (;;); /\* infinite loop i.e. loop for ever \*/

}

else /\* parent process\*/

{ // pid hold the process id of child process

printf("\nPARENT: sending SIGHUP\n\n"); kill(pid, SIGHUP);

sleep(3); // pause for 3 seconds

printf("\nPARENT: sending SIGINT\n\n"); kill(pid, SIGINT);

sleep(3); // pause for 3 seconds

printf("\nPARENT: sending SIGQUIT\n\n"); kill(pid, SIGQUIT);

sleep(3); // pause for 3 seconds

}

}

// function definition of sighup() void sighup()

{

signal(SIGHUP, sighup); /\* reset signal \*/

printf("CHILD: I have received a SIGHUP\n");

}

// function definition of sigint() void sigint()

{

signal(SIGINT, sigint); /\* reset signal \*/

printf("CHILD: I have received a SIGINT\n");

}

// function definition of sigquit() void sigquit()

{

printf("My Papa has Killed me!!!\n"); exit(0);

}

# Slip 5

//Assignment : Write a C program to create an unnamed pipe. The child process will write following three messages to pipe and parent process display it.

//Message1 = “Hello World”

//Message2 = “Hello SPPU”

//Message3 = “Linux is Funny”

#include<stdio.h> #include<unistd.h> #include<stdio.h> #include<unistd.h> int main()

{

int pipefds[2];

int returnstatus; int pid;

char writemessages[3][50]={"Hello world","Hello SPPU","Linux is Funny"}; char readmessage[50];

returnstatus = pipe(pipefds); if (returnstatus == -1)

{

printf("Unable to create pipe\n"); return 1;

}

pid = fork();

// Child process if (pid == 0)

{

printf("Child Process write Messaages\n"); printf("%s\n", writemessages[0]);

write(pipefds[1], writemessages[0], sizeof(writemessages[0])); printf("%s\n", writemessages[1]);

write(pipefds[1], writemessages[1], sizeof(writemessages[1])); printf("%s\n", writemessages[2]);

write(pipefds[1], writemessages[2], sizeof(writemessages[2]));

}

else

{ //Parent process sleep(5);

printf("Parent Process Display Messaages\n");

read(pipefds[0], readmessage, sizeof(readmessage)); printf("%s\n", readmessage);

read(pipefds[0], readmessage, sizeof(readmessage));

printf("%s\n", readmessage);

read(pipefds[0], readmessage, sizeof(readmessage)); printf("%s\n", readmessage);

}

return 0;

}

# Slip 6

// Assignment . Write a C program to create n child processes. When all n child processes

//terminates, Display total cumulative time children spent in user and kernel mode.

#include<sys/types.h> #include<sys/wait.h> #include<unistd.h> #include<time.h>

#include<sys/times.h> #include<stdio.h> #include<stdlib.h>

int main(void)

{

int i, status; //pid\_t data type is signed interger type repesenting process ID pid\_t pid; //time\_ t data type used to storeing system time value

time\_t currentTime;//times() stores the current process time in the struct tms that //that buffer points to.

struct tms cpuTime;

if((pid = fork())==-1) //start child process

{

perror("\nfork error"); exit(EXIT\_FAILURE);

}

else if(pid==0) //child process

{

time(&currentTime); // gives normal time

printf("\nChild process started at %s",ctime(&currentTime)); for(i=0;i<5;i++)

{

printf("\nCounting= %dn",i); //count for 5 seconds sleep(1);

}

time(&currentTime);

printf("\nChild process ended at %s",ctime(&currentTime)); exit(EXIT\_SUCCESS);

}

else

{ //Parent process time(&currentTime);

printf("\nParent process started at %s ",ctime(&currentTime)); if(wait(&status)== -1) //wait for child process

perror("\n wait error"); if(WIFEXITED(status))

printf("\nChild process ended normally \n");

else

printf("\nChild process did not end normally"); if(times(&cpuTime)<0) //Get process time

perror("\nTimes error"); else

{ // \_SC\_CLK\_TCK: system configuration time: seconds clock tick

printf("\nParent process user time= %fn",((double) cpuTime.tms\_utime));

printf("\nParent process system time = %fn",((double) cpuTime.tms\_stime)); printf("\nChild process user time = %fn",((double) cpuTime.tms\_cutime));

printf("\nChild process system time = %fn",((double) cpuTime.tms\_cstime));

}

time(&currentTime);

printf("\nParent process ended at %s",ctime(&currentTime)); exit(EXIT\_SUCCESS);

}

}

# Slip 7

// Assignment . Implement the following unix/linux command (use fork, pipe and exec system call) ls -l | wc -l

#include<stdio.h> #include<stdlib.h> #include<unistd.h> #include<fcntl.h> #include<sys/wait.h> #include<errno.h> void main()

{

int filedes[2];

if (pipe(filedes) == -1)

{

perror("pipe"); exit(1);

}

if(fork() == 0)

{

while ((dup2(filedes[1], STDOUT\_FILENO) == -1)) {}

char \*args[] = {"ls","-l", NULL}; int ret = execvp("ls",args);

if(ret <0)

{

printf("Program can't be executed\n");

}

exit(0);

}

close(filedes[1]);

if(fork() == 0)

{

while((dup2(filedes[0], STDIN\_FILENO) == -1)){}

char \*args[] = {"wc","-l", NULL}; int ret = execvp("wc",args);

if(ret <0)

{

printf("Program can't be executed\n");

}

exit(0);

}

char output[100];

read(filedes[0], output, 100); printf("%s", output);

close(filedes[0]); exit(0);

}

# Slip 8

// Assignment . Write a C program to Identify the type (Directory, character device, Block device,

//Regular file, FIFO or pipe, symbolic link or socket) of given file using stat() system call

#include<stdio.h> #include<stdlib.h> #include<sys/stat.h> #include<sys/types.h> #include<time.h>

#include<fcntl.h>

int main(int argc, char const \*argv[])

{

if(argc != 2){

fprintf(stderr, "usage : %s <filepath>\n", argv[0]); return 1;

}

int file = open(argv[1], O\_RDONLY); if(file < 0){

fprintf(stderr, "error opening file\n"); return 1;

}

struct stat st;

if(fstat(file, &st) < 0)

{

fprintf(stderr, "error reading file info\n"); return 1;

}

printf("File Name is %s : \n", argv[1]);

printf("File Type: ");

switch (st.st\_mode & S\_IFMT)

{

case S\_IFBLK: printf("this block device\n"); break; case S\_IFCHR: printf("this character device\n"); break; case S\_IFDIR: printf("this directory\n"); break; case S\_IFIFO: printf("this FIFO/pipe\n"); break; case S\_IFLNK: printf("this symlink\n"); break; case S\_IFREG: printf("this is regular file\n"); break; case S\_IFSOCK: printf("this socket\n"); break; default: printf("unknown?\n"); break;

}

return 0;

}

# Slip 9

// Assignment . Generate parent process to write unnamed pipe and will write into it. Also generate child process which will read from pipe

#include<stdio.h> #include<unistd.h> int main() {

int pipefds[2];

int returnstatus; int pid;

char writemessages[1][20]={"Hello"}; char readmessage[20];

returnstatus = pipe(pipefds); if (returnstatus == -1)

{

printf("Unable to create pipe\n"); return 1;

}

pid = fork();

// Child process if (pid == 0)

{

read(pipefds[0], readmessage, sizeof(readmessage));

printf("Child Process - Reading from pipe â€“ Message is %s\n", readmessage);

}

else

{ //Parent process

printf("Parent Process - Writing to pipe - Message is %s\n", writemessages[0]); write(pipefds[1], writemessages[0], sizeof(writemessages[0]));

}

return 0;

}

# Slip 10

//Assignment : Write a C program which receives file names as command line arguments and display

//those filenames in ascending order according to their sizes. I) (e.g $ a.out a.txt b.txt c.txt,

â€¦)

#include <stdio.h> #include <dirent.h> #include<string.h> #include<unistd.h> #include<time.h> #include<sys/stat.h>

#include<sys/types.h> #include<stdlib.h>

typedef struct file\_info

{

char \*name; size\_t size;

}fileinfo;

void insertionSort(fileinfo info[], int n)

{

int i, j;

fileinfo key;

for (i = 1; i < n; i++)

{

key = info[i]; j = i - 1;

while (j >= 0 && info[j].size > key.size)

{

info[j + 1] = info[j]; j = j - 1;

}

info[j + 1] = key;

}

}

void main(int argc, char \*\*argv)

{

struct stat fstat; if(argc < 3)

{

printf("no files passed\n"); exit(1);

}

int fileCount = argc -1; fileinfo info[fileCount]; int i;

printf("Display all filenames in ascending order according to their sizes.\n"); for(i =1;i<argc;i++)

{

info[i-1].name = argv[i]; stat(argv[i],&fstat);

info[i-1].size = fstat.st\_size;

}

insertionSort(info, fileCount); for(i=0;i<fileCount;i++)

{

printf("%s -> %ld\n", info[i].name, info[i].size);

}

}

# Slip 11

//Assignment : Write a C program that a string as an argument and return all the files that begins with that name in the current directory.

//For example > ./a.out foo will return all file names that begins with foo

#include<stdio.h> #include<dirent.h> #include<string.h>

int main(int argc, char\* argv[])

{

DIR \*d;

char \*position;

struct dirent \*dir; int i=0;

if(argc!=2){

printf("Provide suffiecient args");

}

else {

d = opendir("."); if (d)

{

while ((dir = readdir(d)) != NULL)

{

position=strstr(dir->d\_name,argv[1]); i=position-dir->d\_name;

if(i==0)

printf("%s\n",dir->d\_name);

}

closedir(d);

}

return(0);

}

}

# Slip 12

// Assignment : Write a C program to implement the following unix/linux command (use fork, pipe and exec system call).

//Your program should block the signal Ctrl-C and Ctrl-\ signal during the execution.

i. Ls -l | wc -l

#include<stdio.h> #include<stdlib.h> #include<unistd.h> #include<fcntl.h> #include<sys/wait.h> #include<errno.h> void main()

{

int filedes[2];

if (pipe(filedes) == -1)

{

perror("pipe"); exit(1);

}

if(fork() == 0)

{

while ((dup2(filedes[1], STDOUT\_FILENO) == -1)) {}

char \*args[] = {"ls","-l", NULL}; int ret = execvp("ls",args);

if(ret <0)

{

printf("Program can't be executed\n");

}

exit(0);

}

close(filedes[1]);

if(fork() == 0)

{

while((dup2(filedes[0], STDIN\_FILENO) == -1)){}

char \*args[] = {"wc","-l", NULL}; int ret = execvp("wc",args);

if(ret <0)

{

printf("Program can't be executed\n");

}

exit(0);

}

char output[100];

read(filedes[0], output, 100); printf("%s", output);

close(filedes[0]); exit(0);

}