****

Assignment 1

CSE-325

Sec-01

Submitted to:

Tanni Mittra

Senior Lecturer

Department of Computer Science & Engineering

Submitted by:

Name: Hasib Ar Rafiul Fahim

ID: 2019-1-60-036

Ans to the question no. 1

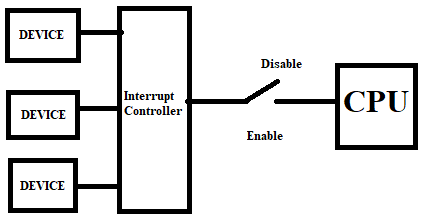
A:

Interrupt and Polling is the two different ways of stopping a CPU when an important task arises. While CPU is busy, Interrupt and Polling handles task in hand.

Interrupt works by its Interrupt handler. CPU has a wire interrupt request line and CPU checks it every time after execution of every single process. While a IO request arises to interrupt request line, CPU lets the interrupt handler to execute the important IO request while keeping less important task at waiting state. When CPU finishes processing IO requests, it gets back to waiting queue.

Polling is also a protocol which tells a CPU when a device needs its attention. Polling keeps asking the io device weather it needs CPU processing or not. CPU repeatedly checks its ports to see if any device needs to be processed or not.

My proposed process device communication model will be based on interrupt. Interrupt is more efficient than polling. Interrupt is faster than polling as polling keeps checking for requests frequently. That's why polling is slow and time consuming. Although, Interrupt is a little bit complex process then polling, it handles process perfectly unlike polling.



B:

My proposed model was base on interrupt handling. While an important process arises, the connection between interrupt controller to CPU of low priority processes get disconnected. CPU switches to IO requests or more important task. When the task is done, CPU checks the interrupt request line and if any task found that is more important, CPU stops current task and responds to interrupt by passing the control to Interrupt controller. When Interrupt controller finishes the task, it lets CPU to finish the task which were at waiting queue.

The system call or IO call arises from operating system, the normal task by software and hardware gets to waiting state and system call or IO call gets executed by Interrupt handler after the execution, operating system sends new tasks to Interrupt controller. Interrupt controller handles the task send by the operating system by priority.

Ans to the question no. 2

Merge.c code is below

#include<stdio.h>

#include<dirent.h>

#include<sys/types.h>

#include<unistd.h>

int main(){

struct dirent \*de;

DIR \*dr=opendir("./MargeFile");

de=readdir(dr);

if(dr==NULL){

printf("failed to open dir\n");

}

while(de!=NULL){

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

de=readdir(dr);

}

int fd=0;

while(de!=NULL){

printf("ino= %ld\nreclen= %ud \nName= %s\n Type= %d\n\n",de-> d\_ino,de-> d\_reclen,

de-> d\_name,de->d\_type);

de=readdir(dr);

}

closedir(dr);

char \*args[]={"/usr/bin/find","-name","Dirr.c",NULL};

execv("./Exec",args);

}

Exec.c code is below:

#include<stdio.h>

#include<unistd.h>

int main(){

char \*args[]={"./MergeFile.c"};

execv(args[0],args);

return 0;

}

Answer to the question no. 3

#include<stdio.h>

#include<unistd.h>

#include<stdlib.h>

int main(){

char \*dupr="rdfind";

char \*moveC="mv";

char \*arg1=" home/user";

char \*arg2=" -deleteduplicates true";

char \*arg3="Filename";

char arg4=".txt ../txtFolder";

char arg5=".jpg ../Pictures";

char arg6=".mp3 ../Music";

execlp(dupr, dupr,arg1,NULL);

execlp(dupr, dupr,arg2,arg1,NULL);

execlp(moveC, moveC,arg3,arg4,NULL);

execlp(moveC, moveC,arg3,arg5,NULL);

execlp(moveC, moveC,arg3,arg6,NULL);

printf(“Duplicate files removed \n All files moved to directory by category\n”);

return 0;

}



Ans to the question no. 4

A:

#include <stdio.h>;

#include <sys/types.h>;

#include<stdlib.h>

void ChildProcess(void);

void ParentProcess(void);

void main(void)

{

pid\_t pid;

pid pPid,cPid;

pid = fork();

if (pid == 0)

ChildProcess();

cPid=getpid();

else

ParentProcess();

pPid=getpid();

}

void ChildProcess(void)

{

int system (const char \*"grep check \* -lR theWord"); //this is the specific word the childProcess

//will try to find

if(system<0){

printf("Error");

}

}

void ParentProcess(void)

{

wait();

printf("%d",cPid);

return();

}

B:

I will change the code so that child process will exit before the parent is done. I have putted the parents to sleep and forced child to exit which will make the child process to zombie.

#include <stdio.h>;

#include <sys/types.h>;

#include<stdlib.h>

void ChildProcess(void);

void ParentProcess(void);

void main(void)

{

pid\_t pid;

pid pPid,cPid;

pid = fork();

if (pid == 0)

ChildProcess();

cPid=getpid();

else

ParentProcess();

pPid=getpid();

}

void ChildProcess(void)

{

int system (const char \*"grep check \* -lR theWord");

if(system<0){

printf("Error");

}

exit(0); //child is exiting before parent is done

}

void ParentProcess(void)

{

sleep(50); //Parent is sleeping

printf("%d",cPid);

return();

}

Here is the zombie process: 

Ans to the question no. 5

A: Zombie process has its very own process ID. Operating system has a specific number of process IDs. If zombie process starts to grow rapidly, all process ID will be assigned to zombie process which will prevent other process from launching. That’s why zombie process is not welcomed.

If zombie processes are already created, we must kill them to prevent its rapid growth. But zombie processes are already dead. So, we have to do certain things to stop them. First of all, we must identify the zombie process. Then we have to find the parent process. After that, we have to send SIGCHLD signal to parent process. This might stop the zombie process. If the zombie process is still running, we must kill the parent process.

B: Return statement is used to return from the current function. On the other hand, exit() stops and terminates the whole program. If we use return on a function, it may return a value and stops the function and goes to next function. But if an exit() call is made, the whole program along with all the functions stop working and exits from the program.

C: A zombie process is the process whose work is complete but still has an entry in the process table. If the parent process doesn’t invoke a wait system call, its child exits and reaped and become a zombie process.

On the other hand, and orphan process is a child process whose parent process has finished executing, but the child process remains running itself. However, orphan process can be picked by the init() system call.

D:

#include <stdlib.h>

#include <sys/types.h>

#include <unistd.h>

int main ()

{

int i;

pid\_t child\_pid;

for(i=0;i<5;i++) child\_pid = fork (); //I used loop to create multiple

//zombie process

if (child\_pid > 0) {

sleep (60);

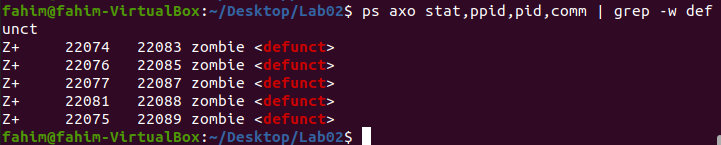
}

else {

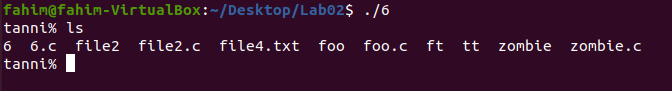
exit (0);

}

return 0;

}

Answer to the question no. 6

The only error I found in the code was, the code was missing #include<stdlib.h> which is required for exit() command. The screenshot after execution is below:

The code with explanation(in the command) is below:

#include <unistd.h>

#include<stdlib.h>

#include <sys/wait.h> //here is the libraries used in this code

#define CMD\_LEN 120 //CMD\_LEN was globally defined here

int main(int argc, char \*\*argv) { //main function

char cmd[CMD\_LEN]; //took a character array with the length of CMD\_LEN

char \*cmd\_args[2]; //took another array of characters

int n;

int child\_pid; //took some variables

while (1) { //this loop is for the program to repeat

write(1, "tanni% ", 7); //this will be written in the console

n = read(0, cmd, CMD\_LEN); //this command is to read the input given to the console

if (n == 0) break; /\* EOF reached; exit program \*/ //breaking condition

cmd[n - 1] = '\0'; /\* replace '\n' with '\0' \*/ //this is for the null value

child\_pid = fork(); //finding the childs process id

if (child\_pid == 0) { //conditions; if child\_pid =0 the console will return a

cmd\_args[0] = cmd; //result. If its 1, it will return NULL

cmd\_args[1] = NULL;

execvp(cmd, cmd\_args); //executing the result from the input given to cmd

write(1, "Command not found\n", 19); //if the command is not found, it will return this line

exit(-1); //exit condition

} waitpid(child\_pid, &n, 0); //waiting for child to end

}

return 0; //program return condition

}

The above code was something like recreating the console. Here the program will take query as input and return the result.

Answer to the question no. 7

The code for foo is done by C programming. I compiled it using gcc plugin. The code is below:

#include<stdio.h>

#include<sys/types.h>

#include<unistd.h>

#include<fcntl.h>

#include<stdlib.h>

void foo(int fd, char\* buf,int b\_size, int n, int skip){

read(fd,buf,100);

}

int main(){

int fd;

fd=open("./file2",O\_RDONLY,0777);

printf("fd is %d\n",fd);

int rdd=0;

char \*ch=(char\*) malloc(100);

foo(fd,ch,100,5,3);

printf("Rdd = %d\n",rdd);

printf("%d blocks of size %d skipping %d bits after each read are read from file",ch,rdd,fd);

}

A screenshot of the output is below:

