Text

Description automatically generated

**ASSIGNMENT-1**

Course Title: Operating System

Course Code: CSE325

Section: 01

Submitted to:

Tanni Mittra

Senior Lecturer

Department of Computer Science & Engineering

Submitted by:

Name: Adri Saha

ID: 2019-1-60-024

Date of submission: 3rd April 2020

1. a. Suppose a program infrequently invokes I/O requests. Whenever an urgent I/O request is invoked by the program then it starts to handle I/O requests frequently for moderate span of time. After executing specified number of I/O requests it starts executing large CPU bursts. While executing large CPU burst, program keeps the requested I/O calls in a queue. After finishing CPU burst it takes requests from the queue and invokes the queued I/O requests.

Now propose a processor- device communication model of your own by using the idea of two approaches such as interrupt and poling to make an efficient I/O communication for the above-mentioned process. If necessary, then draw a diagram that depicts the scenario and provides the workflow of your proposed model.

b. How does operating system involve in different activities to handle your proposed model. Describe the answer in your own way.

**Answer to the question no 1 (a)**

For the above-mentioned process, I think interrupt would be the better option. The first advantage is that the efficiency of the microcontroller in the Interrupt system is much superior to that of the Polling method. The microcontroller continuously checks whether the system is ready or not in the polling process, but the chances of data loss are higher in Polling than in Interrupt. Interrupt and Polling are two methods for dealing with events caused by devices that can occur at any time while the CPU is busy with another task.

A process execution begins with a CPU burst, followed by an I/O burst and then another CPU burst and so on. Polling and Interrupt cause the CPU to stop what it is doing and respond to a more urgent task. In this case, a program only uses I/O requests infrequently. For a reasonable period, the program manages regular urgent I/O requests. It begins executing large CPU bursts after completing a given number of I/O requests. The software holds requested I/o calls in a queue when performing massive CPU bursts. It takes requests from the queue and invokes the queued I/O requests after the CPU burst is done.

In the polling process, a microcontroller continuously checks the status of a register and executes the task based on the register's status. Polling is a method of informing the CPU that a device requires its attention. Unlike interrupts, where a computer informs the CPU that it requires processing, polling includes the CPU constantly asking the I/O device if it requires processing. Each computer connected to the CPU is tested on a regular basis by the CPU.

In most cases, the microcontroller's hardware is very slow in terms of processing speed. As a result, using the Polling method will slow down the microcontroller's speed and, as a result, results. As a result, we will need a different approach to complete the task—the Interrupt Method. The microcontroller in this method does not check the status of the unit or the status of the registers on a regular basis. Instead, the unit or register indicates that it is ready to the microcontroller. When an interrupt request line receives an I/O request, the CPU allows the interrupt handler to process the critical IO request while maintaining the less important tasks in a waiting state. The microcontroller resumes the program where it left off after executing the ISR (Interrupt Service Routine-ISR).

Diagram

Description automatically generated

Figure: Interrupt handling CPU block diagram

So, my proposed a processor is interrupt controller for device communication model to make an efficient I/O communication for the above-mentioned process. It is more reliable than polling and quicker than polling because polling constantly checks for requests. Even though interrupting is a more complicated procedure than polling, polling is slower and takes longer for these purposes. So, unlike poling, I believe interrupt handles this situation perfectly.

**Answer to the question no 1 (b)**

Interrupt handling was the basis of my proposed model. An interrupt is a signal received from external devices, usually I/O devices connected to a computer, or from a program running on the computer, that causes the operating system to pause its current operations and execute the appropriate part of the operating system. Hardware interrupts, software interrupts, and traps are the three types of interrupts. When an interrupt signal is detected in a computer, the computer either resumes the current program or switches to another program.

When an important process begins, the relation between the interrupt controller and the CPU of low-priority processes is broken. CPU moves to IO requests or a higher priority mission. When the task is completed, the CPU checks the interrupt request line for any tasks that are more critical than the current task. If any tasks are found to be more important, the CPU stops the current task and responds to the interrupt by transferring control to the interrupt controller. When the interrupt controller completes the task, it releases the CPU to complete the tasks in the waiting list.

The system call or IO call is initiated by the operating system, after which the usual task performed by software and hardware is put in a waiting state, and the system call or IO call is executed by the interrupt handler, after which the operating system sends new tasks to the interrupt controller. The interrupt controller prioritizes the tasks sent by the operating system. So, operating system involves in different activities to handle interrupt. For example:

* An interrupt handler is a piece of code found in most operating systems. If there are several interrupts waiting to be dealt, the interrupt handler prioritizes them and stores them in a queue.
* Interrupts are signals sent by external devices, usually I/O devices, to the CPU. They instruct the CPU to halt its current activities and run the required operating system component.
* An operating system in which the interrupt system is the mechanism for disclosing all changes in the states of hardware and software resources, and these changes are the events that cause these resources to be reassigned to satisfy work-load demands.

So, that is how operating system involve in different activities to handle interrupt.

1. Write a program “**MergeFile.c**” to read all txt files in a specified directory that begins with that same name and merge them all into a new txt file. You must have to returns a file descriptor of the new file.

We all know about exec system call. To understand the functionality of exec system call, write another program **Exec.c.**  The program will be replaced by the image of “**MergeFile.c**” program. The specified name of files which you have to search in the directory need to be passed to “**MergeFile.c**” file from “**Exec.c**” file. Now do the necessary actions to implement the above-mentioned scenario.

**Answer to the question no 2**

This Program is MergeFile.c which read all txt files in a specified directory that begins with that same name and merge them all into a new txt file. You must have to returns a file descriptor of the new file.

#include<stdio.h>

#include<fcntl.h>

#include<stdlib.h>

#include<dirent.h>

#include<unistd.h>

#include<string.h>

int main()

{

DIR \*dirptr=opendir(".");

creat("newFile.txt",0644);

if(dirptr==NULL)

{

printf("Error in opening the directory\n");

exit(0);

}

char \*merge="MergeFile";

struct dirent \*entry = readdir(dirptr);

while(entry!= NULL)

{

char \*fp;

fp=entry->d\_name;

int flag=strcmp(entry->d\_name,merge);

if(flag==0)

{

int fd = open(fp,O\_RDONLY,0);

if(fd==-1)

{

printf("Error opening file\n");

continue;

}

char str[1024];

read(fd, str, 1024);

close(fd);

char \*newDoc="newFile.txt";

fd = open(newDoc,O\_WRONLY | O\_CREAT,0);

int size;

if(fd==-1)

{

printf("Error opening file\n");

continue;

}

size = write(fd, str, 1024);

close(fd);

}

entry = readdir(dirptr);

}

printf("Merged successfully and created newFile.txt file\n");

closedir(dirptr);

return 0;

}Text

Description automatically generated

Figure: Output in terminal of merged file.c

This program is Exec.c which will be replaced by the image of “**MergeFile.c**” program.

#include<stdio.h>

#include<stdlib.h>

#include<unistd.h>

#include<fcntl.h>

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

execl("./MergeFile.c","extra","cmd","arg",NULL);

printf("Exec successfully and created Output.txt file\n");

return 0;

}

Text

Description automatically generated

Figure: Output in terminal of execfile.c

1. Let us consider you have a directory and it contains many sub-directories and files. First you have to find duplicate files name inside the directory and if found then remove the duplicate files. Write a program that will remove the duplicate files and categorize all files in the specified folder based on their file type. That is all .txt file in txt folder, all .jpg files in image folder etc. The argument to the program is a directory name.

**Answer to the question no 3**

#include<stdio.h>

#include<stdlib.h>

#include<unistd.h>

int main()

{

char \*duplicatemove = "rdfind";

system(duperemove);

printf("All Duplicate files removed ");

char \*new="rdfind";

char \*move="mv";

char file1="mv.txt../home/Desktop/Documents";

char file2="mv.c../home/Desktop/Documents";

char file3="mv.jpg../home/Pictures";

char file4="mv.mp4../home/Videos";

execlp(new,new,file1,NULL);

execlp(new,new,file2,file1,NULL);

execlp(move,move,file3,file4,NULL);

execlp(move,move,file3,file5,NULL);

return 0;

}

Text

Description automatically generated

Figure: Output in terminal of finding duplicate & categorized file

1. Write a program that will create a child process that will execute a specified command they will display all the files name that contains a specified word. If child process cannot execute the command, then print an error message. While child is executing, parent will wait for it, after finishing child task parent will print the child id with its returned status.

b. Now change the code of the above program in such a way that the child process will become a zombie process. This zombie process must remain in the system for at least 10 seconds. Show the zombie processes from CLI.

**Answer to the question no 4**

(a)

#include<stdio.h>

#include<stdlib.h>

#include<unistd.h>

#include<sys/types.h>

#include<sys/wait.h>

int main()

{

int pid= fork();

printf("Fork returned %d\n ",(int)pid);

if(pid==0)

{

system("grep -l");

printf("exit child\n ");

exit(50);

}

if(pid>0){

printf("I am waiting for my child to end\n");

pid\_t Cpid;

int status;

Cpid= wait(&status);

int C\_Status = WEXITSTATUS(status);

printf("Child`s work finished with status %d\n ",C\_Status);

printf("My Child pid is %d\n",(int)pid);

printf("Parent exiting\n ");

}

else{

printf("Fork Failed. Error creating child\n");

}

}Text

Description automatically generated

Figure: Output in terminal of child process 4(a)

(b)

#include<stdio.h>

#include<stdlib.h>

#include<unistd.h>

#include<sys/types.h>

#include<sys/wait.h>

#include<fcntl.h>

int main(){

int pid= fork();

printf("Fork returned %d\n",(int)pid);

if(pid==0){

printf("Zombie Process executing\n");

printf("child executed\n ");

exit(0);

}

if(pid>0){

sleep(20);

printf("I am waiting for my child to end \n");

printf(" My Child id is %d \n",(int)pid);

printf("Parent exit \n");

}

else{

printf("Fork Failed. Error creating child");

}

}

Text

Description automatically generated

Figure: Output in terminal of zombie process 4(b)

5.

* 1. Why is zombie process not welcomed? What action you need to take if zombie processes are already created.
  2. Write the differences between exit () and return statement.
  3. Write the differences between zombie and orphan process.
  4. Write similar program like problem no 4(b) that creates five zombie process.

**Answer to the question no 5**

**(a)**

**Reasons why is zombie process not welcomed**:

When a parent begins a child process and the child process ends, but the parent does not pick up the child's exit code, zombie processes are formed. The process entity must remain in place before this occurs - it consumes no energy and is dead, but it remains, thus the word "zombie."

* Zombie When their parent processes are no longer running, the existence of zombie processes can indicate an operating system bug. If there are just a few zombie processes, this is not a big deal, but when the system is under a lot of pressure, it can trigger problems.
* All resources associated with a dead process are deallocated so that they can be reused by other systems. A zombie process consumes no more memory than is needed to sustain its resource table entry, which is insignificant. If there are so many zombies, a dilemma arises. As a result, zombie processes are not welcome because they can cause bugs in the operating system.

**Action I need to take if zombie processes are already created**:

The address space of the Parent process is repeated when a process is generated in UNIX using the fork () function call. When a parent process uses the wait () method call, the parent's execution is suspended until the child is terminated. There are many ways to prevent the formation of zombies.

1. Using the wait () system call: When a parent process uses the wait () system call after creating a child, it means that it will wait for the child to complete before reaping the child's exit status.
2. The zombie process is fully removed from memory after wait () is called. This usually happens quickly, so you will not see any zombie processes on your machine.

(b)

**Differences between exit () and return statement:**

|  |  |
| --- | --- |
| **exit ()** | **return ()** |
| 1. exit () is a system call which terminates current process and not an instruction of C language. | 1. return () is a C language instruction/statement, and it returns from the current function. |
| 1. The exit () is used to exit from the program and returns the control back to the operating system. | 1. The return is used to return a value from the function. |
| 1. Syntax: exit (1); | 1. Syntax: return; |

**(c)**

**Differences between zombie and orphan process:**

|  |  |
| --- | --- |
| Zombie process | Orphan process |
| 1. A zombie process, also known as a defunct process, is one that has ended execution but still has an entry in the process table since its parent process neglected to invoke the wait () system call. | 1. 1. An orphan process is a computer process whose parent process has completed or terminated, but the child process continues to operate. |
| 1. When the child terminates it gives exit status to parent. | 1. Parent exit, Init process becomes the parent of child process. |
| 1. Meanwhile time suppose your parent is in sleep state and unable to receive any status from child. Though the child exit but the process occupies space in process table. | 1. Whenever child is terminated, process table gets deleted by os. |
| 1. If no parent waiting (did not invoke wait ()) process is a zombie | 1. If parent terminated without invoking wait, process is an orphan. |

**(d)**

#include<stdio.h>

#include<stdlib.h>

#include<unistd.h>

#include<sys/types.h>

#include<sys/wait.h>

int main(){

int pid=fork();

fork();

fork();

int state;

if(pid==0){

execl("grep","grep","-l","\"hello\"","\*",NULL);

execl("/usr/bin/grep","grep","-l","\"hello\"","\*",NULL);

exit(4);

}

else if(pid>0){

sleep(30);

wait(&state);

printf("Zombie Process created \n");

printf("Child pid is %d, return status-%d \n",pid,state);

}

else{

printf("Fork Failed. Error creating child\n");

}

return 0;

}

Text

Description automatically generated

Figure: Output in terminal of many zombie process program 5(d)

6.

Consider the following Code:

#include <unistd.h>

#include <sys/wait.h>

#define CMD\_LEN 120

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

char cmd[CMD\_LEN];

char \*cmd\_args[2];

int n;

int child\_pid;

while (1) {

write(1, "tanni% ", 7);

n = read(0, cmd, CMD\_LEN);

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

cmd[n - 1] = '\0'; /\* replace '\n' with '\0' \*/

child\_pid = fork();

if (child\_pid == 0) {

cmd\_args[0] = cmd;

cmd\_args[1] = NULL;

execvp(cmd, cmd\_args);

write(1, "Command not found\n", 19);

exit(-1);

}

waitpid(child\_pid, &n, 0);

}

return 0;

}

Now compile the above code and if you find any error then fix it. Explain the code line by line. Explain your understanding about the code in your own words.

**Answer to the question no 6**

The only error I found in the code was that #include<stdlib.h> was missing, which is needed for the exit() command.

Here, mainly the code is for executing commands in linux terminal using C program.

Now, I am explaining the code line by line,

#include<stdlib.h> //defines standard library header functions

#include <unistd.h> // defines various symbolic constants, types & functions

#include <sys/wait.h> // wait for process termination

#define CMD\_LEN 120 //define command length globally

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

char cmd[CMD\_LEN]; //define character type command array with length of CMD\_LEN

char \*cmd\_args[2]; //took another character command array size of 2

int n;

int child\_pid; //took integer variables

while (1) { //loop for continuous

write(1, "tanni% ", 7); // inside write will be shown on the terminal

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

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

cmd[n - 1] = '\0'; /\* replace '\n' with '\0' \*/ //if find space in string

child\_pid = fork(); //fork system call to find children process id

if (child\_pid == 0) { //condition

cmd\_args[0] = cmd; // in 1st position put comand

cmd\_args[1] = NULL; //next position null

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

write(1, "Command not found\n", 19); //if comand will not found return this line

exit(-1); // exit condition

}

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

}

return 0; //program returning condition

}

Text

Description automatically generated

Figure: Output in terminal of program 6

1. Write a function foo (int fd, char\* buf, int b\_size, int n, int skip) that reads to buf from file with file descriptor fd, n blocks of size b\_size each. The last argument specifies how many bytes to skip after reading each block. Return -1 if the operation is unsuccessful. Else return total number of bytes read.

**Answer to the question no 7**

#include<stdio.h>

#include<fcntl.h>

#include<string.h>

#include<dirent.h>

#include<unistd.h>

#include<stdlib.h>

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

{

int i,byts=0;

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

{

read(fd,buf,b\_size);

byts +=b\_size;

lseek(fd,skip,SEEK\_SET);

}

return byts;

}

int main()

{

char \*filename="NewFile.txt";

creat(filename,0644);

int fd = open(filename,O\_WRONLY,0);

int size\_f;

if(fd==-1)

{

printf("Efinding error in opening file\n");

exit(0);

}

printf("File Descriptior = %d\n", fd);

char string[1024];

strcpy(string,"Sample text.Read it.Output purpose.");

size\_f = write(fd, string, 1024);

close(fd);

char reader[1024];

fd = open(filename,O\_RDONLY,0);

if(fd==-1)

{

printf("finding error in opening file\n");

exit(0);

}

char finalstr[1024];

int result=foo(fd,finalstr,5,1,4);

read(fd, finalstr, 1024);

printf("in read of 30 bytes\n");

printf("%d bytes to skip each block after reading each block\n",result);

close(fd);

return 0;

}

Text

Description automatically generated

Figure: Output in terminal of finding bytes program 7