**Indian Institute of Information Technology, Pune.**



**(Practical Manual)**

**2019-2020**

**Name: Devang Hemant Bhagwat**

**Subject: Operating System.**

**Department: Computer science and Engineering.**

**Batch: A2.**

**MIS No.: 111816011.**

**Professor: Dr Bhupendra Singh.**

**Lab: Computer Lab.**

****

**ASSIGNMENT 1:**

Q1. Consider The C function “printf()” on UNIX. Is it implemented by the OS, or by an application level library? What system call does printf() make internally?

Answer:

Yes, the C function printf() is executed by the OS. It executes the write function internally.

Q2. What is the total number of processes in the end of the following program? Assume there is only one process that starts running the main. Also, assume that all he system calls succeed.

* main(){

fork();

fork();

fork();

}

Answer:

The total number of processes in the end of the given program is 8. One parent with its seven descendants.

Q3. Consider the following program:

* main(){

int fd;

fd=open(outfile,O\_RDWR);

fork();

write(fd,hello,5);

exit();

}

Assume all system calls finish successfully in a uniprocessor system. Also assume that ll system calls cannot be interrupted in the middle of the execution. What will be the contents of the outfile file after all the processes have successfully exited? Explain briefly.

Answer:

In this pseudocode, The file named outfile is opened in only Read and write mode (represented by O\_RDWR) anf then fork(); is called. So there will be a child process created which will inherit the file descriptors from its parent; so, the word hello will be written two times in ‘outfile’ without any separation.

Q4. Why are **mkdir, ln**, and **rm** are implemented as separate user level programs, while cd is implemented as a built-in command?

Answer:

cd command changes the directory of the current program in the user is executing. If we use it as a user level program, then the child process will automatically copy the directory of the parent process. Hence there will be no change in the child directory and so, using the using the cd as a user level program will be useless. That is why it is used as a built in function.

Q5. In UNIX, a child process may terminate before the parent calls wait(). When the parent calls wait eventually, it still expects to read the correct exit code that the child returned. To support this functionality, UNIX does not completely removes the process till it’s parent called wait() on it.

Such processes that have completed their execution but still have an entry in the process table are called zombie processes. Usually, the presence of a zombie process in the system for a long time indicates a bug in the program. (It is a common error)

UNIX also provides the SIGCHLD signal, which is received by the parent process whenever one of the child exits. In class we discussed that the shell implements “&” functionality by not calling wait() immediately. Should the shell never call wait()? When should it call wait Answer by providing a short pseudo-code. (Hint: you may want to uses the SIGCHLD signal)

Answer:

The shell should call wait() at such a time that the zombie processes are not destroyed, then, it may resource leak or create an error. The pseudo-code is given below:

* if(pid==0){

while(SIGCHLD!=1){

wait();

}

}

else if(pid==0){

/\* execute the child process\*/

}

SIGCHLD generates if one of the processes created by the parent process has exited. Thus, when the child is a process, parent needs to wait and when the child has terminated, SIGCHLD will be true and when the child has terminated, SIGCHLD will be true and the parent need not to wait in that case.

**ASSIGNMENT 2:**

Q1. A system with two dual-core processors has four processors available for scheduling. A CPU-sensitive application is running on this system. All the input is performed at program setup, when a single file must be opened. Similarly, all output is performed just before the program terminates, when the program results must be returned in a single file between start up and termination, the program is entirely CPU-bound. Your task is to improve the performance of this application by multithreading it. The application runs on the system that uses the one-to–one threading model (Each user thread maps to a kernel thread).

* How many threads will you create to perform input and output? Explain.
* How many threads will you create for the CPU-intensive portion of the program? Explain.

Answer:

In this case, threads are going to perform both input and output operations with concurrency. There should be as many threads as there are blocking system calls, as the threads will be spent blocking. Since, we need to avoid creation of unnecessary threads; so, there should be one thread for input and one for output. Since we have the advantage of four processors, we will have four threads for the CPU intensive portion. If we use any greater than 4, the program won’t run.

Q2. Consider the following code segment:

* pid\_t pid;

pid=fork();

if(pid==0){

fork();

thread\_create( . . . . . . . );

}

fork();

* How many unique processes are created?
* How many unique threads are created?

Answer:

In this case, there are two unique threads and six unique processes.

Q3. The program shown below uses Pthreads API what will be the output from the program at LINE C and LINE P?

* #include<pthread.h>

#include<stdio.h>

int value=0;

void \*runner(void \*param); //the thread

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

pid\_t pid;

pthread\_t tid;

pthrad\_attr\_t attr;

pid=fork();

if(pid==0){

pthread\_attr\_init(&attr);

pthread\_create(&tid,&attr,runner,NULL);

pthread\_join(tid,NULL);

printf(“CHILD: value = %d\n”); //LINE C

}

else if(pid>0){

wait(NULL);

printf(“PARENT : value = %d\n”); //LINE P

}

}

void \*runner(void \*param){

value=5;

pthread\_exit(0);

}

Answer:

At line C, the output should be ‘5’ and in line P, it should be ‘0’.

Q4. Write a multithreaded program that calculates various statistical values for a list of numbers. This program will be passed a series of numbers on then command line and will then create three separate worker threads. One thread will determine the average of the numbers, the second will determine the maximum value, and the third will determine the minimum value.

Answer:

* #include<stdio.h>

#include<stdlib.h>

#include<sys/types.h>

#include<unistd.h>

#include<pthread.h>

#include<sys/wait.h>

Int m,\*a;

void \*min(void \*par){

int i,minim;

minim=a[0];

for(i=1;i<m,i++){

if(a[i]<=minim){

minim=a[i];

}

}

printf(“The minimum value is: %d\n”,minim);

}

void \*max(void \*par){

int i,ma;

ma=a[0];

for(i=1;i<m,i++){

if(a[i]<=ma){

ma=a[i];

}

}

printf(“The maximum value is: %d\n”,ma);

}

void \*avg(void \*par){

int i,sum;

float av;

for(i=1;i<m,i++){

sum+=a[i];

}

av=sum/m;

printf(“The average value is: %d\n”,av);

}

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

int i;

pid\_t pid;

pthread\_t tid[3];

pthrad\_attr\_t attr[3];

pid=fork();

if(pid==0){

for(i=0;i<3,i++){

pthread\_attr\_init(&attr[i]);

}

pthread\_create(&tid[0],&attr[0],min,NULL);

pthread\_create(&tid[1],&attr[1],max,NULL);

pthread\_create(&tid[2],&attr[2],avg,NULL);

for(i=0;i<3,i++){

pthread\_join(tid[i],NULL);

}

printf(“Child process has been executed…\n”);

}

else if(pid>0){

wait(NULL);

printf(“Parent process has been executed…\n”);

}

return(0);

}

**ASSIGNMENT 3:**

Q1. Implement the First come first serve scheduling algorithm using any programing language.

Answer:

* #include<stdio.h>

#include<stdlib.h>

struct process{

int burst\_time,wait\_time,turn\_around\_time;

};

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

struct process \*p;

int i,j,n,key;

float t\_wait\_time=0,t\_turn\_around\_time=0;

printf(“Please enter the number of processes…n”);

scanf(“%d,&n”);

p=(struct process\*)calloc(n,sizeof(struct process));

printf(“Please enter the process burst time one-by-one\n”);

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

scanf(“%d”,&p[i].burst\_time);

}

for(i=0;i<n-1;i++){

for(j=0;j<=i;i++){

p[i].wait\_time+=p[j].burst\_time;

}

t\_wait \_time+=p[i].wait\_time;

}

printf(“The average waiting time is : %d\n”, t\_wait \_time/n);

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

for(j=0;j<=i;i++){

p[i].turn\_around\_time+=p[j].burst\_time;

}

t\_ turn\_around\_time+=p[i]. turn\_around\_time;

}

printf(“The average waiting time is : %d\n”, t\_ turn\_around\_time/n);

return(0);

}

Q2. Implement the Shortest job first scheduling algorithm using any programing language.

Answer:

* #include<stdio.h>

#include<stdlib.h>

struct process{

int burst\_time,wait\_time,turn\_around\_time;

};

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

struct process \*p;

int i,j,n,key;

float t\_wait\_time=0,t\_turn\_around\_time=0;

printf(“Please enter the number of processes…n”);

scanf(“%d,&n”);

p=(struct process\*)calloc(n,sizeof(struct process));

printf(“Please enter the process burst time one-by-one\n”);

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

scanf(“%d”,&p[i].burst\_time);

}

for(i=0;i<n-1;i++){

key=p[i].burst\_time;

j=i-1;

while(j>=0 && p[j].burst\_time>=key){

p[j+1].burst\_time=p[j].burst\_time;

p[j].burst\_time=key;

j=j-1;

}

}

for(i=0;i<n-1;i++){

for(j=0;j<=i;i++){

p[i].wait\_time+=p[j].burst\_time;

}

t\_wait \_time+=p[i].wait\_time;

}

printf(“The average waiting time is : %d\n”, t\_wait \_time/n);

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

for(j=0;j<=i;i++){

p[i].turn\_around\_time+=p[j].burst\_time;

}

t\_ turn\_around\_time+=p[i]. turn\_around\_time;

}

printf(“The average waiting time is : %d\n”, t\_ turn\_around\_time/n);

return(0);

}