Assignment

CSE-316

Marks -30

Submission: 9th April 2020

Section-K18AW

Name-Nekkanti Chakradhar

Registration No.:11811685

Roll No:15

==========

QUESTION-2

CODE:-

```
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
int sum =0;
int num;
int numCount;
int max = -100000;
int min = 100000;
int i;
float avg;
void* avg_runner(void*arg)
{
int i; for(i = 0; i < numCount; i++)
{
scanf("%d", &num);
sum += num;
avg = sum/numCount;
}
pthread_exit(0);
void* min_runner(void*arg)
{
int i;
for(i = 0; i < numCount; i++)
{
scanf("%d", &num);
if(i == 0)
```

```
{
min = num;
}
else if (min > num) min = num;
}
pthread_exit(0);
}
void* max_runner(void* arg)
{
int i;
for(i = 0; i < numCount; i++)
{
scanf("%d", &num);
if(i ==0)
{
max =num;
}
else if(max< num)
max =num;
}
pthread_exit(0);
}

int main(int argc, char **argv)
{
printf("This program finds the maximum, minimum, and average of a series of
numbers.\n");
printf("How many numbers would you like to process?\n");
scanf("%d",&numCount);
printf("Enter the numbers\n");
int avg = atoi(argv[1]);
int min = atoi(argv[2]);
int max = atoi(argv[3]);
pthread_attr_t attr;
pthread_attr_init(&attr);
pthread_t thread1;
pthread_t thread2; pthread_t thread3;
pthread_create(&thread1, &attr, avg_runner, &avg);
pthread_create(&thread2, &attr, min_runner, &min);
pthread_create(&thread3, &attr, max_runner, &max);
pthread_join(thread1, NULL);
pthread_join(thread2, NULL);
pthread_join(thread3, NULL);
printf("The average is: %f\n", avg);
```

```
printf("The minimum is: %d\n", min);
printf("The maximum is: %d\n", max); return 0;
}
```

EXPLANATION: -

This problem provides minimum value, maximum value and average value of the given number.

#include<stdio.h>for print

we use #include<stdlib.h>

for string we use #include<pthread.h>

//for thread we use I am attempting to write a multithreaded program that calculates various statistical values fora list of numbers. This program will be passed a series of numbers on the command line and will then create three separate worker threads. One thread will determine the average, the second will find the maximum, and the third will find the minimum. This is my code. It compiles and runs, but crashes with a segmentation fault.

The variables representing the average, minimum, and maximum values will be stored globally. The worker threads will set these values, and the parent thread will output the values once the workers have exited.

This program will be passed a series of number son the 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 determine the minimum value.

Analysis of Multithreaded Algorithms:
Analysing work is simple: ignore the parallel constructs and analyse the serial algorithm. For example, we already noted previously that the work of
P-Fib(n)is $T(n)=T(n-1)+T(n-2)+\Theta(1)$,
which has the solution $T(n)=\Theta(Fn)$, the work of P-Fib(n).

analysing span requires a different approach.

Analyzing Span If a set of sub computations (or the vertices representing them) are in series, the span is the sum of the spans of the sub computations.

This is like normal sequential analysis (as was just exemplified above with the sum $T(n-1)+T(n-2)$).

If a set of sub computations (or the vertices representing them) are in parallel, the span is the maximum of the spans of the computations.

This is where analysis of multi threaded algorithms differs.

Our example, the span of the parallel recursive calls of P-Fib(n)is computed by taking the max rather than the sum:

$T_\infty(n) = \max(T_\infty(n-1), T_\infty(n-2)) + \Theta(1) = T_\infty(n-1) + \Theta(1)$. There currence $T_\infty(n) = T_\infty(n-1) + \Theta(1)$ has solution $\Theta(n)$. So the span of P-Fib(n) is $\Theta(n)$.

We can now compute the parallelism of P-Fib(n) in general (not just the specific case of n=4 that we computed earlier) by dividing its work

$\Theta(F_n)$ by its span $\Theta(n)$: $T_1(n)/T_\infty = \Theta(F_n)/\Theta(n) = \Theta(F_n/n)$

This grows dramatically, as Fn grows much faster than n. For any given number of processors P, there is consider able parallel slackness $\Theta(F_n/n)/P$.

For any P above some n there is likely to be some thing for additional processors to do. Thus there is potential for near perfect linear speed up as n g rows.

(Of course in this example it's because we chose an inefficient way to compute Fibonacci numbers, but this was only for illustration. These ideas apply to other well designed algorithms.)

Parallel Loops So far we have used spawn, but not the parallel keyword, which is used with loop constructs such as for. Here is an example.

Suppose we want to multiply an n x n matrix A=(aij) by an n -vector x =(xj). This yields an n-vector y=(yi) where:

The following algorithm m does this in parallel:

The parallel for keywords indicate that each iteration of the loop can be executed concurrently.

(Notice that the inner for loop is not parallel ; a possible point of improvement to be discussed.)

Implementing Parallel Loops It is not realistic to think that all n sub computations in these loops can be spawned immediately with no extra work.

(For some operations on some hard ware up to a constant n this may be possible ;e.g., hardware designed for matrix operations; but we are concerned with the general case.)

How might this parallel spawning be done, and how does this affect the analysis? Parallel for spawning can be accomplished by a compiler with a divide and conquer approach, itself implemented with parallelism.

The procedure shown below is called with Mat-Vec-Main-Loop(A,x,y,n,1,n). Lines 2 and 3 are the lines originally within the loop.

The computation DAG is also shown. It appears that a lot of work is being done to spawn the n leaf node computations, but the increase is not asymptotic.

The work of Mat-Vecis $T_1(n) = \Theta(n^2)$ due to the nested loops in 5-7. Since the tree is a full binary tree, the number of internal nodes is 1 fewer than then leaf nodes, so this extra work is $\Theta(n)$.

Each leaf node corresponds to one iteration of loop, and the extra work of recursive spawning can be a mortised across the work of the iterations, so that it contributes only constant work.

Concurrency platforms sometimes coarse n there cursion tree by executing several iterations in each leaf, reducing the amount of recursive spawning.

The span is increased by Θ(lgn)due to the addition of there cursion tree for Mat-VecMain-Loop, which is of height Θ(lgn).

In some cases (such as this one), this increase is washed out by other dominating factors (e.g., the span in this example is dominated by the nested loops).

Nested Parallelism Continuing with our example, the span is Θ(n) because even with full utilization of parallelism the inner for loop still requires Θ(n).

Since the work is Θ(n2) the parallelism is Θ(n2)/Θ(n)=Θ(n).

Can we improve on this? Perhaps we could make the inner for loop parallel as well? Compare the original to the revised version Mat-Vec:

Description:

Here I am giving a simple example which will demonstrate you about how to run multiple tasks using multiple threads.

In this example we will do three different tasks using three different threads. Each thread will be responsible for its own task only.

Among these three threads one will find the average number of the input numbers, one will be responsible for finding the Maximum number from the input array of numbers, and one will be responsible for finding the Minimum number from the input array of numbers.

In this example I have created three different classes which implements the Runnable interface to make a Thread class.

This is the multithreaded program that calculates various statistical values for a list or a group of numbers and this is the program which will be working when we pass a series of numbers on the command line of ubuntu interface this program helps us to create the 3 worker threads.in these 3 worker threads one worker thread is useful for the determination of the average to a given number and the other thread will give us the maximum value present, and the last thread will determine the minimum value and the variables are stored globally worker thread will helps to store the values and the parent thread will helps us to give the output whenever the worker threads gets executed

==========

QUESTION-25

Code:

```
#include<stdio.h>
#include<stdlib.h>
#include<unistd.h>
#include<sys/types.h>
#include<string.h>
#include<sys/wait.h>
intmain()
{
intfd1[2];
intfd2[2];
char input str[100];
pid tp;
if(pipe(fd1)==-1)
{
fprintf(stderr,"PipeFailed");
return1;
}
if(pipe(fd2)==-1)
{
fprintf(stderr,"PipeFailed");
return1;
}
printf("Enter a string:");

fgets (input_str, sizeof(input_str),stdin);
input_str[strlen(input_str)-1]='\0';
if((p=fork())<0)
{
fprintf(stderr,"forkFailed");
return1;
}
elseif(p>0)
{
char str[100];
close(fd1[0]);
printf("WRITEFROM:%d[%s]\n",getpid(),input_str);
write(fd1[1],input_str,strlen(input_str)+1);
close(fd1[1]);
wait(NULL);
close(fd2[1]);
read(fd2[0],str,100);
```

```
printf("READFROM :%d[%s]\n",getpid(),str);
printf("\nOutput:%s\n",str);
close(fd2[0]);
}
else
{
close(fd1[1]);
charstr[100];
read(fd1[0],str,100);
printf("READFROM :%d[%s]\n",getpid(),str);
int i=0;
while(i<strlen(str)){
charch=str[i];
intresult=(ch>64&&ch<91)?1:(ch>96&&ch<123)?2:0;
ch=(result==1)?ch+32:(result==2)?ch-32:ch;
ch=ch+32 ifit's anUpperCaseAlphabet
ch=ch-32 ifit'saLowerCaseAlphabet
ch=chifit'snotanAlphabet
str[i++]=ch;}
str[i]='\0'; //stringendswith'\0'
close(fd1[0]);

close(fd2[0]);
printf("WRITEFROM:%d[%s]\n",getpid(),str);
write(fd2[1],str,strlen(str)+1);
close(fd2[1]);
exit(0);
}
return0;
}
```

Explanation:-

Pipes were meant for communication between related processes. Can we use pipes for unrelated process communication, say, we want to execute client program from one terminal.

and the server program from another terminal? The answer is No.

Then how can we achieve unrelated processes communication,

the simple answer is Named Pipes.

Even though this works for related processes, it gives no meaning to use the named pipes for related process communication. We used one pipe for one-way communication and two pipes for bi-directional communication.

Does the same condition apply for Named Pipes?

The answer is no, we can use single named pipe that can be used for two-way communication (communication between the server and the client, plus the client and the server at the same time) as Named Pipe supports bi-directional communication.

#include <sys/types.h>

#include <sys/stat.h>

#include <fcntl.h>

#include <unistd.h>

This system call would create a special file or file system node such as ordinary file, device file, or FIFO.

The arguments to the system call are path name, mode and dev. The path name along with the attributes of mode and device information.

The path name is relative, if the directory is not specified it would be created in the current directory.

The mode specified is the mode of file which specifies the file type such as the type of file and the file mode as mentioned in the following tables.

The dev field is to specify device information such as major and minor device numbers. Two-way Communication Using Named Pipes

The communication between pipes are meant to be unidirectional. Pipes were restricted to one-way communication in general and need at least two pipes for two-way communication.

Pipes are meant for inter-related processes only. Pipes can't be used for unrelated processes communication, say, if we want to execute one process from one terminal and another process from another terminal, it is not possible with pipes.

Do we have any simple way of communicating between two processes, say unrelated processes in a simple way? The answer is YES. Named pipe is meant for communication between two or more unrelated processes and canal so have bi-directional communication.

Already, we have seen the one-directional communication between named pipes, i.e., the messages from the client to the server.

Now, let us take a look at the bi-directional communication i.e., the client sending message to the server and the server receiving the message and sending back another message to the client using the same named pipe.

Following is an example :

Step1 – Create two processes, one is fifo server two way and another one is fifo client two way.

Step2 – Server process per forms the following – Creates a named pipe
(using library function mkfifo()) with name “fifo_twoway” in /tmp directory, if not created.

Opens the named pipe for read and write purposes.

Here, created FIFO with permissions of read and write for Owner. Read for Group and no permissions for Others.

Waits infinitely for a message from the client.

If the message received from the client is not “end”, prints the message and reverses the string.

The reversed string is sent back to the client. If the message is “end”, closes the fifo and ends the process.

Step3 – Client process performs the following

Opens the named pipe for read and write purposes. Accepts string from the user.

Checks, if the user enters “end” or other than “end”.

Either way, it sends a message to the server.

However, if the string is “end”, this closes the FIFO and also ends the process.
If the message is sent as not “end”, it waits for the message (reversed string) from the client and prints there versed string.

Repeats infinitely until the user enters the string “end”.

Algorithm1:

1. create the pipe and create the process.

2. get the input in the main process and pass the output to the child process using pipe.

3. perform the operation given in the child process and print the output.

4. stop the program.

Algorithm2:

Step1 – Create a pipe.

Step2 – Send a message to the pipe.

Step3 – Retrieve the message from the pipe and write it to the standard output.

Step4 – Send another message to the pipe.

Step5 − Retrieve the message from the pipe and write it to the standard output.

Test Cases:

Step1− Create two pipes. First one is for the parent to write and child to read, say as pipe1. Second one is for the child to write and parent to read, say as pipe2.

Step2− Create a child process.

Step3− Close unwanted ends as only one end is needed foreach communication.

Step4− Close unwanted ends in the parent process, read end of pipe1 and write end of pipe2.

Step5− Close the unwanted ends in the child process, write end of pipe1 and read end of pipe2.

Step6− Perform the communication as required.

SOURCE CODE FOR 2WAY PIPE.c:-

```
#include<stdio.h>

#include<unistd.h>

 int main()

 {

int pipefds1[2], pipefds2[2];

 int returnstatus1, returnstatus2;

 int pid;

char pipe1writemessage[20] = "Hi";

char pipe2writemessage[20] = "Hello";

char readmessage[20];

 returnstatus1 = pipe(pipefds1);

if (returnstatus1 == -1)

 { printf("Unable to create pipe 1 \n"); return 1; } returnstatus2 = pipe(pipefds2);

if (returnstatus2 == -1)

 { printf("Unable to create pipe 2 \n"); return 1; } pid = fork();

if (pid != 0) // Parent process

 { close(pipefds1[0]); // Close the unwanted pipe1 read side
```

```
 close(pipefds2[1]); // Close the unwanted pipe2 write side
 printf("In Parent: Writing to pipe 1 – Message is %s\n", pipe1writemessage);
write(pipefds1[1], pipe1writemessage, sizeof(pipe1writemessage));
read(pipefds2[0], readmessage, sizeof(readmessage));
 printf("In Parent: Reading from pipe 2 – Message is %s\n", readmessage); }
else
 {
 //child process
close(pipefds1[1]); // Close the unwanted pipe1 write side
close(pipefds2[0]); // Close the unwanted pipe2 read side
read(pipefds1[0], readmessage, sizeof(readmessage));
printf("In Child: Reading from pipe 1 – Message is %s\n", readmessage);
printf("In Child: Writing to pipe 2 – Message is %s\n", pipe2writemessage);
 write(pipefds2[1], pipe2writemessage, sizeof(pipe2writemessage));
}
 return 0;
}
```

Compilation: -

```
gcc twowayspipe.c –o twowayspipe
```

Execution: -

```
In Parent: Writing to pipe 1 – Message is Hi
In Child: Reading from pipe 1 – Message is Hi
In Child: Writing to pipe 2 – Message is Hello
In Parent: Reading from pipe 2 – Message is Hello
```