Homework 3 - 4/25/2019

Monday, April 22, 2019 1:10 PM

Due 4/25/2019

1) Problem 4.25: Modifying the socket-based date time server to be multithreaded from Figure 3.21. MAKE SURE YOU TURN in executable .java files

Create two versions

a) Creating a new thread for each request

b) Using a Java-based thread-pool

2) Problem 4.17

3) Read and summarize following page: http://www.thegeekstuff.com/2013/11/linux-process-and-threadsLinks to an external site.

Exercise 4.17

The program shown in Figure 4.16 uses the PthreadsAPI. What would be the output from the program at LINE C and LINE P?

```
#include <pthread.h>
#include <stdio.h>

#include <types.h>

int value = 0;
void *runner(void *param); /* the thread */

int main(int argc, char *argv[])
{
pid_t pid;
pthread_t tid;
pthread_attr_t attr;

   pid = fork();

   if (pid == 0) { /* child process */
      pthread_attr_init(&attr);
      pthread_create(&tid,&attr,runner,NULL);
      pthread_join(tid,NULL);
      printf("CHILD: value = %d",value); /* LINE C */
   }
   else if (pid > 0) { /* parent process */
      wait(NULL);
      printf("PARENT: value = %d",value); /* LINE P */
   }
}

void *runner(void *param) {
   value = 5;
   pthread_exit(0);
}
```

Figure 4.16 C program for Exercise 4.17.

LINE C: CHILD: 5

LINE P: PARENT: 0

The value of the global variable $value$ is changed in the CHILD program to be 5 in the new thread

created from the $pthread\ create$ command and using the $*runner$ function. Inside the CHILD memory the $value$ would be considered 5, however; the PARENT memory global value for $value$ wouldn't be changed. The CHILD and PARENT only share memory when the memory written is explicitly made to be shared memory. Otherwise the memories are separate copies.

Exercise 4.25
Modify the socket-based date server (Figure 3.21) in Chapter 3 so that the server services each client request in a separate thread

```
import java.net.*;
import java.io.*;

public class DateServer
{
   public static void main(String[] args) {
      try {
         ServerSocket sock = new ServerSocket(6013);

         /* now listen for connections */
         while (true) {
            Socket client = sock.accept();

            PrintWriter pout = new
             PrintWriter(client.getOutputStream(), true);

            /* write the Date to the socket */
            pout.println(new java.util.Date().toString());

            /* close the socket and resume */
            /* listening for connections */
            client.close();
         }
      }
      catch (IOException ioe) {
         System.err.println(ioe);
      }
   }
}
```

Figure 3.21 Date server.

Assume client is using DataClient.java as provided.

Code for part a, new thread per request:
DateServer

```
/**
 * Created by rtdimpsey on 4/13/17.
  * Edited by Daniel Yan 4/25/2019
```

```
 */
import java.net.*;
import java.io.*;

//This multi-threaded implementation uses Java Thread class
public class DateServer extends Thread
{
	private Socket client;
	//Each thread id should be corresponding to another client
	private static int id;
	private final static int MAX_THREAD = 100;
	//Quick constructor to create a DateServer
	public DateServer(Socket client)
	{
		super();
		this.client = client;
	}
	public void run(){
		//Try catch block is needed for exception
		while(Thread.activeCount() < MAX_THREAD)
		{
			System.out.println("Running thread[" + Thread.activeCount() + "]");
			try
			{
				//Print out client as a Date object
				PrintWriter pout = new PrintWriter(client.getOutputStream(), true);
				pout.println(new java.util.Date().toString());
				Thread.sleep(1000);
			}
			catch(Exception ie)
			{
				System.err.println(ie);
			}
		}
		//Ending thread print
		System.out.println("All threads are running...for regular Threads");
  }
	public static void main(String[] args)
  {
    try
    {
			//Create a new DateServer thread
	 DateServer thread[] = new DateServer[MAX_THREAD];
			ServerSocket sock = new ServerSocket(6014);
	 //Create an accept socket
	 Socket client = sock.accept();
			for (int i = 0; i < MAX_THREAD; i++)
	 {
				//Create a new DateServer thread with id
				thread[i] = new DateServer(client);
				//Run the thread
				thread[i].start();
				}
```

```
    }
            //Catch any errant exception
    catch (IOException ie) {
                    System.err.println(ie);
            }
  }
}
```

Code for part b, thread pool usage
DateServerPool

```
/**
 * Created by rtdimpsey on 4/13/17.
 * Edited by Daniel Yan 4/25/2019
 */
import java.net.*;
import java.io.*;
//Needed ThreadPool import
import java.util.concurrent.Executors;
import java.util.concurrent.ExecutorService;


//This thread pool implementation uses the Java Runnable interface
public class DateServerPool implements Runnable
{
        private Socket client;
        //Each thread id should be corresponding to another client
        private final static int MAX_THREAD = 100;
        //Quick constructor to create a DateServerPool
        public DateServerPool(Socket client)
        {
                super();
                this.client = client;
        }
        public void run(){
                //Try catch block is needed for exception
                while(Thread.activeCount() < MAX_THREAD)
                {
                        System.out.println("Running thread[" + Thread.activeCount() + "]");
                        try
                        {
                                //Print the client out as a Date object
                                PrintWriter pout = new PrintWriter(client.getOutputStream(), true);
                                pout.println(new java.util.Date().toString());
                        }
                        catch(IOException ie)
                        {
                                System.err.println(ie);
                        }
                }
                System.out.println("All threads are running...for Executor ThreadPool");
        }
```

```
   public static void main(String[] args)
   {
                //Array needed for ExecutorService
                Runnable[] dsp = new DateServerPool[MAX_THREAD];
      try
      {
         ServerSocket sock = new ServerSocket(6014);
                        //Make a accept socket
                        Socket client = sock.accept();
                        ExecutorService threadExecutor = Executors.newFixedThreadPool(MAX_THREAD);
         //As we specified the MAX_THREAD size in array we need a for loop
                        for(int i = 0; i < MAX_THREAD; i++)
         {
            //Initialize the thread with client
                                dsp[i] = new DateServerPool(client);
            //Execute the thread
                                threadExecutor.execute(dsp[i]);
                                }
      }
                //For any odd IO exception
      catch (IOException ie) {
                        System.err.println(ie);
                }
   }
}
```

Date Client

```
/**
 * Created by rtdimpsey on 4/13/17.
 */
import java.net.*;
import java.io.*;

public class DateClient
{
   public static void main(String[] args)
   {
      try
      {
         Socket sock = new Socket("127.0.0.1",6014);
         InputStream in = sock.getInputStream();
         BufferedReader bin = new BufferedReader(new InputStreamReader(in));
         String line;
         while ((line = bin.readLine()) != null)
         {
            System.out.println(line);
         }
         sock.close();
      }
      catch (IOException ie) {}
   }
}
```

GeekStuff Summarization:

Linux Processes:

- Simply a running instance of a program, where a kernel provides system resources for the process
- Processes have priority assigned based on kernel context switches, and can be pre-empted due to higher priority processes
- Interprocess communication is used for processes to talk to another, and shared memory is used for sharing data
- Fork() creates a child and parent process. Where in the child process if a change is made, a separate copy of stack and the heap are made for the child

Linux Threads & Light Weight Processes:

- Simply a flow of execution for a process, where a process with multiple flows would be called multi-threaded process
- Linux kernel views threads as processes inside of a kernel, these are termed Light Weight Processes (LWPs)
- LWPs share the same address space and other resources comparative to normal processes
- Threads and LWPs are the same, the terminology differentiation is threads for user view and LWPs for kernel view

Linux Process States:

- Each Linux Process will contain various states
- RUNNING: Process is executing or waiting to be executed
- INTERRUPTIBLE: Process is in sleep mode and is waiting to be woken up by interruption
- UN-INTERRUPTIBLE: Process is in sleep mode but cannot be woken up by a signal
- STOPPPED: Process is stopped, may be due to a signal (SIGSTOP for example)
- TRACED: Process is being debugged
- ZOMBIE: Process is terminated but parent of process hasn't found the termination of their child
- DEAD: Process is terminated and process table entry is removed. Parent has found terminated child state