**Multi-Threading and Socket Programming in C**

Students Name

Institutional Affiliation

Course

Instructors Name

Date

**Multi-Threading and Socket Programming in C**

**Task 1**

One component of a process is the thread. Multithreading is the practice of running numerous threads at once by-passing control of the central processing unit (CPU) between them quickly (called context switching). Threads are independent lines of code that may run simultaneously. Your application will essentially have to handle two tasks at once.

**Task 2**

most Python 3 implementations just provide the impression that the several threads are executing concurrently. It's easy to see threading as if your software had two (or more) separate processors, each carrying out its own work in parallel. Almost, but not quite. Even if the threads are spread over many CPUs, only one will be active at any one moment. In order to have numerous tasks running at once, you will need to use a non-standard version of Python, write part of your code in a foreign language, or use multiprocessing, which adds some cost to your program.

However, not all tasks can necessarily be accelerated by using threading due to the design of the Python implementation. This is because one Python thread can only execute at a time owing to interactions with the GIL. Generally speaking, tasks that spend a lot of time waiting for external events are excellent candidates for threading. It's possible that problems that need extensive CPU calculation but spend little time waiting for external events would not benefit from the optimizations. This holds true for Python programs when they are executed on the default Python distribution. The GIL may be released and many threads can run in parallel if they are written in C. Read the documentation for your Python distribution to learn how threads are handled. You should look at the multiprocessing module if you are using a vanilla Python installation and just using Python code to solve a CPU-bound issue. You can get design clarity benefits from threading if you use it in your program's architecture. Even while most of the examples you'll see in this guide make use of threads, that doesn't guarantee that they'll run any quicker. They are more organized and logical due to the use of threading.

Task 3

While the execution of one process has no effect on another, processes that cooperate may be impacted by those that are already running. Even while it would seem that separately operating processes would be more efficient, there are numerous cases in which taking use of people's natural tendency to work together would improve things like processing time, user friendliness, and adaptability. IPC, or inter-process communication, is a method through which different processes may exchange information and coordinate their behavior. Essentially, the interaction between these systems is a kind of cooperation enabled by their ability to exchange information with one another. Interprocess communication may occur through both of the following:

* Remembering Together
* Sending a Message

An OS may support both forms of interaction. The shared-memory and message-passing techniques will be covered first. It is up to the programmer to decide which variables the processes should share in order to facilitate communication between them utilizing shared memory. The following is an example of a possible mode of interaction based on the use of shared memory: Assume processes 1 and 2 are running in parallel and are dependent on each other for access to shared resources or data. Process1 produces data in shared memory concerning specific calculations or resource use. When process2 requires the shared information, it will look up the record in shared memory, make note of the data produced by process1, and take appropriate action. Through shared memory, processes may retrieve data for archival purposes from another process and send data to other processes with granularity and specificity.

**Task 4**

#include <stdio.h>

#include <string.h>

#include <sys/socket.h>

#include <arpa/inet.h>

int main(void)

{

int socket\_desc;

struct sockaddr\_in server\_addr;

char server\_message [2000], client\_message [2000];

// Clean buffers:

memset (server\_message,'\0’, sizeof(server\_message));

memset (client\_message,'\0’, sizeof(client\_message));

// Create socket:

socket\_desc = socket (AF\_INET, SOCK\_STREAM, 0);

if (socket\_desc < 0) {

printf ("Unable to create socket\n");

return -1;

}

printf ("Socket created successfully\n");

// Set port and IP the same as server-side:

server\_addr.sin\_family = AF\_INET;

server\_addr.sin\_port = htons(2000);

server\_addr.sin\_addr.s\_addr = inet\_addr("127.0.0.1");

// Send connection request to server:

if(connect(socket\_desc, (struct sockaddr\*)&server\_addr, sizeof(server\_addr)) < 0){

printf("Unable to connect\n");

return -1;

}

printf("Connected with server successfully\n");

// Get input from the user:

printf("Enter message: ");

gets(client\_message);

// Send the message to server:

if(send(socket\_desc, client\_message, strlen(client\_message), 0) < 0){

printf("Unable to send message\n");

return -1;

}

// Receive the server's response:

if(recv(socket\_desc, server\_message, sizeof(server\_message), 0) < 0){

printf("Error while receiving server's msg\n");

return -1;

}

printf("Server's response: %s\n",server\_message);

// Close the socket:

close(socket\_desc);

return 0;

}

**Task 5**

**Server side**

#include <stdio.h>

#include <string.h>

#include <sys/socket.h>

#include <arpa/inet.h>

int main(void)

{

int socket\_desc, client\_sock, client\_size;

struct sockaddr\_in server\_addr, client\_addr;

char server\_message[2000], client\_message[2000];

// Clean buffers:

memset(server\_message, '\0', sizeof(server\_message));

memset(client\_message, '\0', sizeof(client\_message));

// Create socket:

socket\_desc = socket(AF\_INET, SOCK\_STREAM, 0);

if(socket\_desc < 0){

printf("Error while creating socket\n");

return -1;

}

printf("Socket created successfully\n");

// Set port and IP:

server\_addr.sin\_family = AF\_INET;

server\_addr.sin\_port = htons(2000);

server\_addr.sin\_addr.s\_addr = inet\_addr("127.0.0.1");

// Bind to the set port and IP:

if(bind(socket\_desc, (struct sockaddr\*)&server\_addr, sizeof(server\_addr))<0){

printf("Couldn't bind to the port\n");

return -1;

}

printf("Done with binding\n");

// Listen for clients:

if(listen(socket\_desc, 1) < 0){

printf("Error while listening\n");

return -1;

}

printf("\nListening for incoming connections.....\n");

// Accept an incoming connection:

client\_size = sizeof(client\_addr);

client\_sock = accept(socket\_desc, (struct sockaddr\*)&client\_addr, &client\_size);

if (client\_sock < 0){

printf("Can't accept\n");

return -1;

}

printf("Client connected at IP: %s and port: %i\n", inet\_ntoa(client\_addr.sin\_addr), ntohs(client\_addr.sin\_port));

// Receive client's message:

if (recv(client\_sock, client\_message, sizeof(client\_message), 0) < 0){

printf("Couldn't receive\n");

return -1;

}

printf("Msg from client: %s\n", client\_message);

// Respond to client:

strcpy(server\_message, "This is the server's message.");

if (send(client\_sock, server\_message, strlen(server\_message), 0) < 0){

printf("Can't send\n");

return -1;

}

// Closing the socket:

close(client\_sock);

close(socket\_desc);

return 0;

}

**Task 6**

To exchange data between a server and a client, TCP sockets are used. Before any client connections can be established, the server's code must execute and initiate a listening session on the specified port. Any time a client and server are connected to the same (server) port, either may initiate communication. Once the message has been transmitted, it will be handled appropriately by the server or client.

