Threads

A Hands-on Approach using C

Computer Networks Lab - Assignment 5

Dr. Sujoy Saha

Assistant Professor, NIT Durgapur

Program

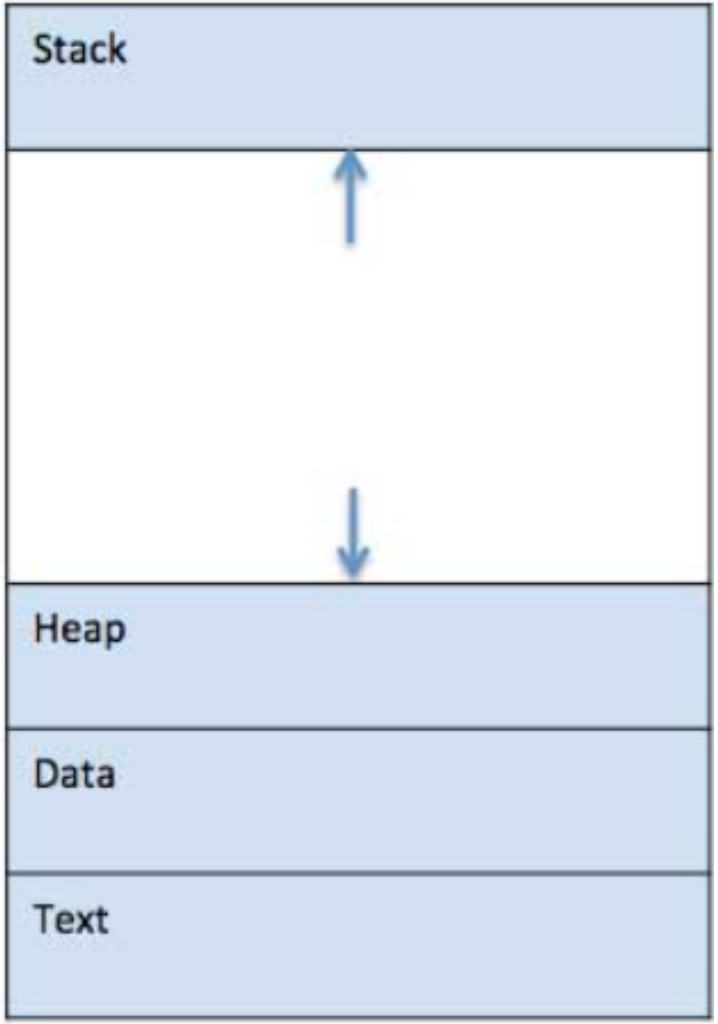
Program: Set of instructions written that a computer will execute

Process: Basic unit of work that will be implemented in the system (in order to execute the program you need some things)

Source: https://www.tutorialspoint.com/operating_system/os_processes.htm

Process

Process: Basic unit of work that will be implemented in the system (in order to execute the program you need some things)



Temporary data such as method/function parameters, return address and local variables.

Dynamically allocated memory to a process during its run time.

Current activity (program counter value); and the contents of the processor's registers.

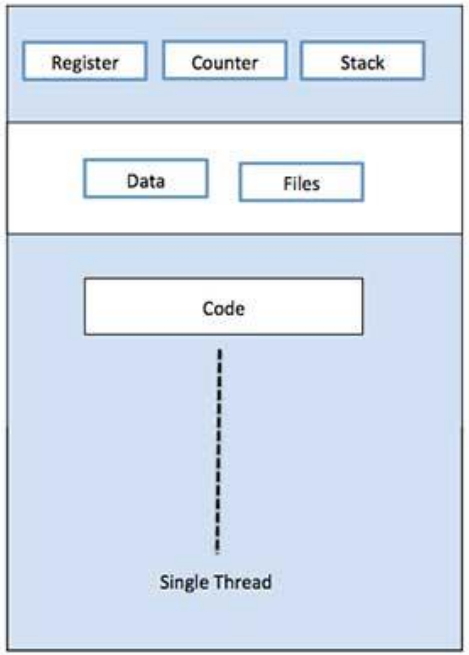
Global and static variables

Source: https://www.tutorialspoint.com/operating_system/os_processes.htm

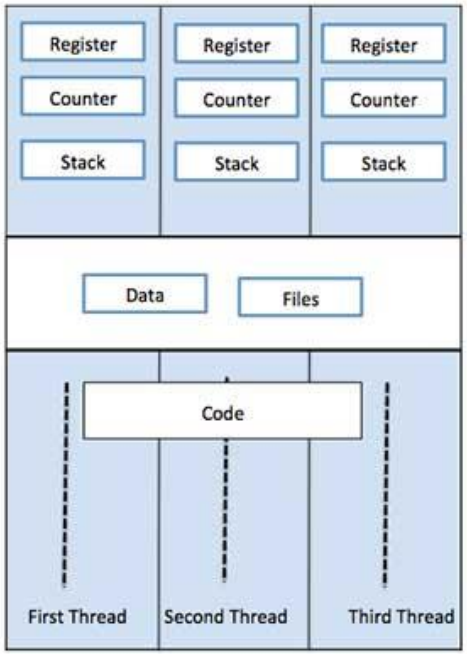
Thread

"A Thread is a Lightweight Process"

- Flow of execution through the process code
- OWN: Program Counter, System Registers, Stack
- SHARED (among Peer Threads): Code Segment, Data Segment, Open Files



Single Process P with single thread



Single Process P with three threads

Source: https://www.tutorialspoint.com/operating_system/os_multi_threading.htm

Thread

What sets multithreading apart

S.N.	Process	Thread
1	Process is heavy weight or resource intensive.	Thread is light weight, taking lesser resources than a process.
2	Process switching needs interaction with operating system.	Thread switching does not need to interact with operating system.
3	In multiple processing environments, each process executes the same code but has its own memory and file resources.	All threads can share same set of open files, child processes.
4	If one process is blocked, then no other process can execute until the first process is unblocked.	While one thread is blocked and waiting, a second thread in the same task can run.
5	Multiple processes without using threads use more resources.	Multiple threaded processes use fewer resources.
6	In multiple processes each process operates independently of the others.	One thread can read, write or change another thread's data.

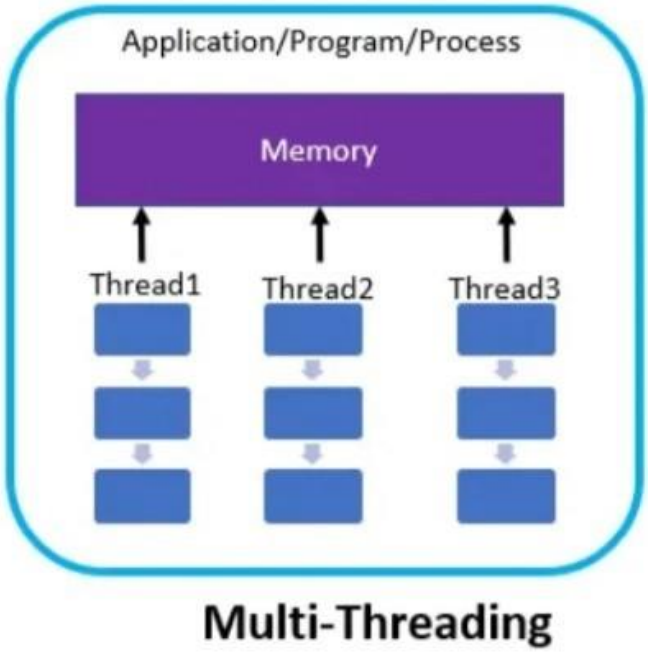
Source: ***https://www.tutorialspoint.com/ope***

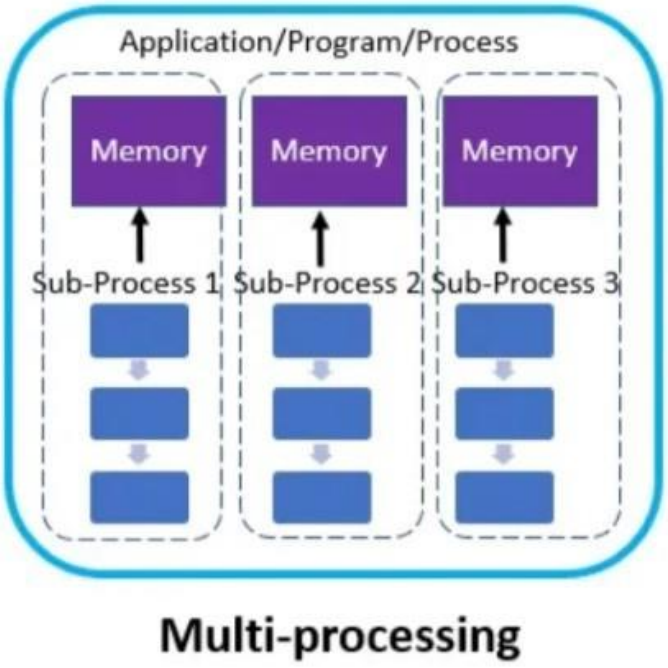
Fork

- fork() is a System call
- Used for Multi-Processing
- Makes a process duplicate itself.
- New process: Child Process
- Old process: Parent Process
- Child gets a ***copy*** of the parent's text and memory space; don't share the same memory.

Source: *https://www.geeksforgeeks.org/fork-system-call-in-operating-system/*

Thread vs Fork: Concept





Source: *https://www.linkedin.com/pulse/multiprocessing-vs-multithreading-jasser-ksouri-rsybe/*

Thread vs Fork: Applications

Multithreading: Tasks involve a lot of waiting (I/O) and require shared data. For example, Web Servers, GUIs and User Interfaces, Networked Applications and Real-time Processing.

Multiprocessing: Tasks involve heavy computations that are CPU-bound and are independent of each other. Examples: Scientific computing, video processing, database engines.

Multi-threading in C

PThreads

PThreads, or POSIX threads, provide the threading APIs an Operating System provides.
We can use the C implementation of PThread in order to implement multithreading in C.

pthread.h: Library imported to support multithreading operations

pthread_create()

To create a new thread, we use the **pthread_create()** function provided by the thread library in C.

Syntax:

pthread_create(thread, attr, routine, arg***);***

```
int pthread_create(pthread_t *thread,
   const pthread_attr_t *attr,
   void *(*start_routine)(void *),
   void *arg);
```

PThreads

pthread_create() takes 4 arguments.

- The **first argument** is a pointer ***to thread_id*** which is set by this function.
- The **second argument** specifies **attributes**. Pointer to a thread attributes object that defines thread properties. Use NULL for default attributes.
- The **third argument** is **name of function** to be executed for the thread to be created.
- The **fourth argument** is used to **pass arguments** to the function

pthread_join: Blocks the calling thread until the thread with identifier equal to the first argument terminates.

const pthread_attr_t *attr — What does it really mean?

- If you pass NULL, the thread will be created with default behavior.
- If you want to customize stack size, scheduling policy, detach state, etc., you must define and use a pthread_attr_t object.

Setting	**What it controls**
detach state	Should the thread auto-cleanup when done? (JOINABLE vs DETACHED)
stack size	How much stack memory the thread has
scheduling policy	FIFO vs Round-Robin
priority	(if real-time scheduling is enabled)
stack address	Where stack is located (rarely needed)
scope	Whether scheduling is system-wide or process-wide

What is pthread_join()?

- pthread_join() is used to wait for a thread to finish before the program continues.

- Without pthread_join(), your main program might exit before your threads are done working!

Syntax

- int pthread_join(pthread_t thread, void **retval);
  - **thread: The thread you want to wait for**
  - **retval: You can ignore it for now (just pass NULL)**

Multithreaded Sequential and Parallel Task Execution

Create a multithreaded C program using POSIX threads (pthreads) to perform three different tasks:

- **Number_Thread**: Print numbers from ***0 to 9*** with a ***2-second*** delay between each
- **Letter_Thread**: Print uppercase letters from ***'A' to 'Z'*** with a ***1.5-second*** delay between each
- **Other_Thread:**
  - When the number ***5 is printed***, spawn ***a third thread*** that prints a **message 10 times (Other_thread)**
  - ***once per second*** and during the execution of this thread *Number_Thread* may halt/pause.

Requirements:
Multithreaded Sequential and Parallel Task Execution

Use **pthread_create()** to create and run multiple threads in parallel.

Use **pthread_join()** to wait for specific threads to finish before proceeding.

Ensure that the **other_thread** runs only when number 5 is reached in the number_printer thread.

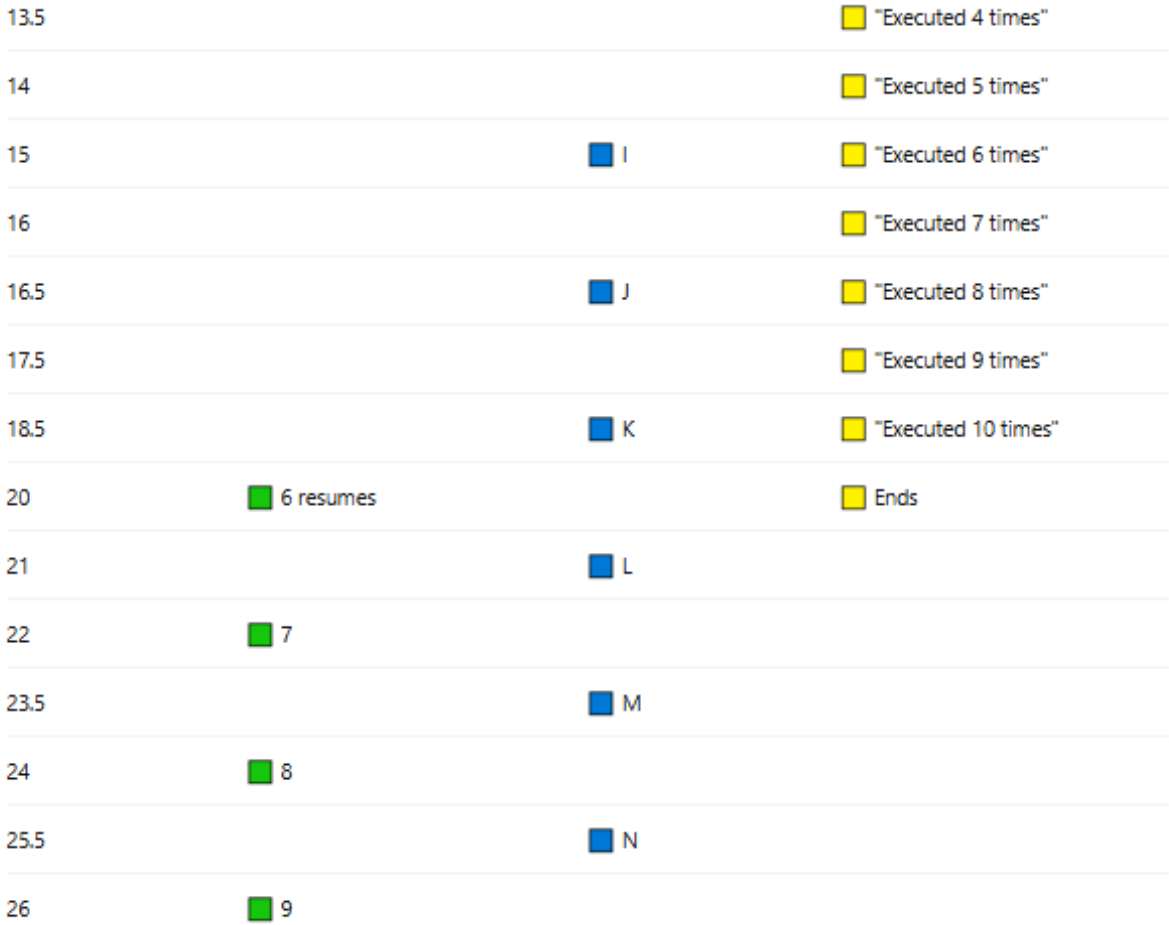
The **main()** function should wait until both the number and letter printing threads are complete.

Demonstrate thread synchronization and sequencing using join().

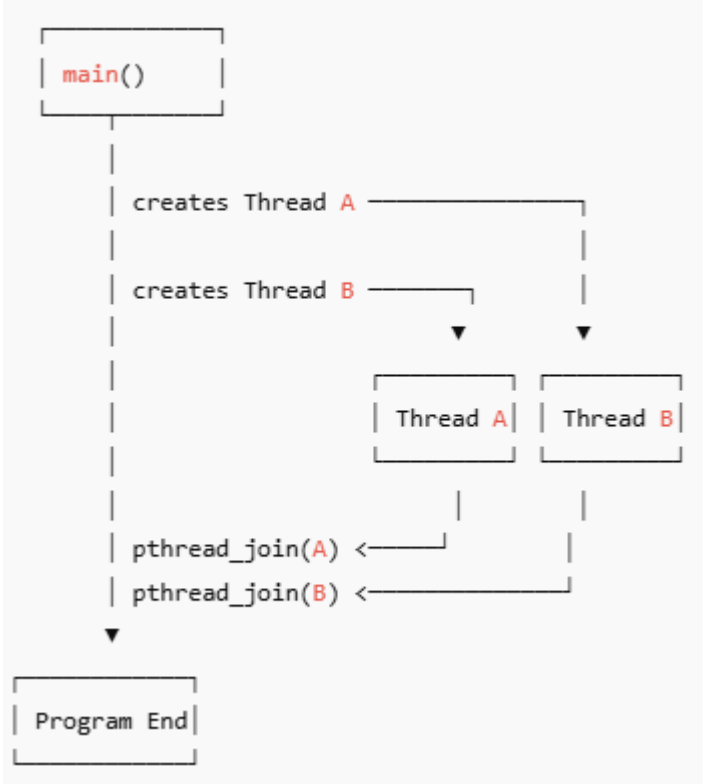
- 🟩 **Number_Thread:** prints `0–9`, every **2 seconds**
- 🟦 **Letter_Thread:** prints `A–Z`, every **1.5 seconds**
- 🟨 **Other_Thread:** spawned **after printing 5**, prints `"Other thread executed X times"` **10 times, every 1 second**
  - ➤ **Number_Thread pauses until Other_Thread finishes (** `pthread_join(`

Time (s)	🟩 Number_Thread	🟦 Letter_Thread	🟨 Other_Thread (Blocking Number_Thread)
0	🟩 0	🟦 A	
1.5		🟦 B	
2	🟩 1		
3		🟦 C	
4	🟩 2		
4.5		🟦 D	
6	🟩 3	🟦 E	
8	🟩 4		
9		🟦 F	
10	🟩 5 → Spawns 🟨 Other_Thread		🟨 Starts, blocks Number_Thread
10.5		🟦 G	🟨 "Executed 1 times"
11.5			🟨 "Executed 2 times"
12		🟦 H	🟨 "Executed 3 times"

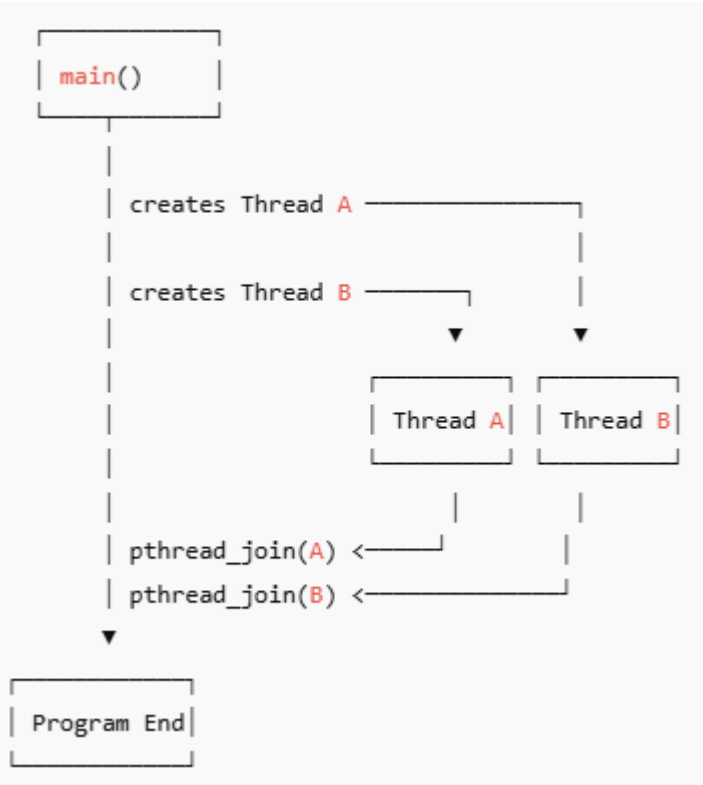
- **Number_Thread:** prints `0–9`, every **2 seconds**
- **Letter_Thread:** prints `A–Z`, every **1.5 seconds**
- **Other_Thread:** spawned **after printing 5**, prints `"Other thread executed X times"` **10 times, every 1 second**
  ➤ **Number_Thread pauses until Other_Thread finishes** (`pthread_join()`)



- main()
  - The program starts in the main() function.
- From here, you initiate the threads.
  - Creating Thread A
    - main() creates Thread A using pthread_create().
    - Thread A starts running in parallel (not sequentially).
  - Creating Thread B
    - Almost immediately, main() also creates Thread B.
- So now there are three threads running:
  - Main thread
  - Thread A
  - Thread B



- Waiting with pthread_join()
  - main() reaches ***the line pthread_join(t1, NULL);***
  - This means***: "Main will wait until Thread A is done."***
- After **Thread A finishes**, main() continues to pthread_join(***t2***, NULL);
  - Now, main waits for Thread B to finish.
- Even if *Thread B finishes earlier*, main won't move past pthread_join(t1) until Thread A is done



After All Threads Finish

- Only after both pthread_join() calls are complete, the program will proceed to the final statement:

Basic MultiThreading Code in C

Importing Header Files

```c
#include <pthread.h>//Header file for enabling multithreading
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h> //Header file for sleep(). man 3 sleep for details.
```

Reference:
https://www.geeksforgeeks.org/multithreading-in-c/

Basic MultiThreading Code in C

Defining the Basic Functions which will be called as threads

```c
void* other_thread(void* args)
{
    for(int i=1;i<=10;i++)
    {
        printf("Other thread executed %d times\n",i);
        sleep(1);
    }
}

void* number_printer(void* args)
{
    for(int i = 0;i<10;i++){
        printf("%d\n",i);
        if(i==5){
            pthread_t other;
            pthread_create(&other, NULL, other_thread, NULL);//Creates a new thread after number 5 is printed, which performs the task defined in the "other_thread" function
            pthread_join(other, NULL);//Waits for the "other" thread to end. Then the execution will continue
        }
        usleep(2000000);
    }
    return NULL;
}

void* letter_printer(void* args)
{
    for(char a = 'A';a<='Z';a++){
        printf("%c\n",a);
        usleep(1500000);
    }
    return NULL;
}
```

Reference:

https://www.geeksforgeeks.org/multithreading-in-c/

Basic MultiThreading Code in C

Defining the main function

```c
int main()
{
    pthread_t number_thread, letter_thread;//Two thread handles one for the number_printer function and the other for letter_printer function.
    printf("Before Thread\n");
    pthread_create(&number_thread, NULL, number_printer, NULL);//Creates thread that executes the number_printer function. Assigns this thread to the handle number_thread
    pthread_create(&letter_thread, NULL, letter_printer, NULL);//Creates thread that executes the letter_printer function. Assigns this thread to the handle letter_thread
    pthread_join(number_thread, NULL);//Waits till the number thread has stopped executing
    printf("Reached end of number thread");
    pthread_join(letter_thread, NULL);//Waits till the letter thread has stopped executing
    printf("Reached end of letter thread");
    printf("After Thread\n");
    exit(0);
}
```

Reference:

https://www.geeksforgeeks.org/multithreading-in-c/

Basic MultiThreading Code in C

Expected Output

```
Before Threads
0
A
B
1
C
2
D
3
E
F
4
G
5
Other thread executed 1 times
H
Other thread executed 2 times
Other thread executed 3 times
```

```
I
Other thread executed 4 times
J
Other thread executed 5 times
K
Other thread executed 6 times
Other thread executed 7 times
L
Other thread executed 8 times
Other thread executed 9 times
M
Other thread executed 10 times
N
O
6
P
7
Q
```

```
R
8
S
9
T
Reached end of number thread
U
V
W
X
Y
Z
Reached end of letter thread
After Threads
```

Reference:

https://www.geeksforgeeks.org/multithreading-in-c/

Basic MultiThreading Code in C

What if pthread_join is not there

```
arko@darxyboi:~/Devstuff/Classes$ ./thread.out
Before Thread
0
Reached end of number threadA
Reached end of letter threadAfter Thread
```

The main process continues as normal, executes all the pthread_create lines and terminates, without waiting for the other threads to finish.

Reference:
https://www.geeksforgeeks.org/multithreading-in-c/

Multithreaded Sequential and Parallel Task Execution

Create a multithreaded C program using POSIX threads (pthreads) to perform three different tasks:

- **Number_Thread**: Prints numbers from **0 to 9**, with a **2-second delay** between each print.
- **Letter_Thread**: Prints letters from **A to Z**, with a **1.5-second delay** between each print.
- **Other_Thread:**
  - Spawned only when Number_Thread reaches 5.
  - Prints a message "Other thread executed X times" for 10 times, once per second.
  - ***Runs in the background (detached***), allowing the program to continue without waiting.

Time (s)	Number_Thread (Green)	Letter_Thread (Blue)	Other_Thread (Yellow - Detached)
0	0	A	
1.5		B	
2	1		
3		C	
4	2		
4.5		D	
6	3	E	
8	4		
9		F	
10	5 → Spawns Other_Thread		Starts (detached)

Number_Thread: Prints numbers from **0 to 9**, with a **2-second delay** between each print.

Letter_Thread: Prints letters from **A to Z**, with a **1.5-second delay** between each print.

Other_Thread:

Spawned only when Number_Thread reaches 5.

Prints a message "Other thread executed X times" for 10 times, once per second.

Runs in the background (detached), allowing the program to continue without waiting.

Time (s)	Number_Thread (Green)	Letter_Thread (Blue)	Other_Thread (Yellow - Detached)
10.5		G	"Executed 1 times"
11.5			"Executed 2 times"
12	6		
12		H	"Executed 3 times"
13.5			"Executed 4 times"
14	7		
15		I	"Executed 5 times"
15.5			"Executed 6 times"
16	8		
16.5		J	"Executed 7 times"
17.5			"Executed 8 times"
18	9		
19.5			"Executed 9 times"
21			"Executed 10 times"

Number_Thread: Prints numbers from **0 to 9**, with a **2-second delay** between each print.

Letter_Thread: Prints letters from **A to Z**, with a **1.5-second delay** between each print.

Other_Thread:

Spawned only when Number_Thread reaches 5.

Prints a message "Other thread executed X times" for 10 times, once per second.

Runs in the background (detached), allowing the program to continue without waiting.

Race Condition

Full Duplex Chat Application in C

Full Duplex Chat Application in C

Importing Libraries

```c
//Standard Libraries
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

//Libraries necessary for networking / socket programming
#include <netdb.h>
#include <netinet/in.h>
#include <sys/socket.h>
#include <sys/types.h>

//Miscellaneous imports
#include <pthread.h>//Header file for enabling multithreading
#include <unistd.h> // read(), write(), close()
```

Full Duplex Chat Application in C

Defining Macros

```
//Defining macros
#define MAX 80
#define PORT 8090
#define SA struct sockaddr
```

Full Duplex Chat Application in C

Receive Function (Similar in Server as well as client)

```c
// Function designed for receiving messages from the client
void* receive_msg(void* args)
{
    int connfd = *(int*)args; // Correct argument passing
    char buff[MAX];
    int n;
    // infinite loop for chat
    while(1){
        bzero(buff, MAX);

        // read the message from client and copy it in buffer
        read(connfd, buff, sizeof(buff));
        // print buffer which contains the client contents
        printf("From client: %s", buff);

        // if the message contains "exit", exit the loop
        if (strncmp("exit", buff, 4) == 0) {
            printf("Client disconnected...\n");
            break;
        }
    }
    return NULL;
}
```

Full Duplex Chat Application in C

Send Function (Similar in Server as well as client)

```c
// Function designed for sending messages to the client
void* send_msg(void* args)
{
    int connfd = *(int*)args; // Correct argument passing
    char buff[MAX];
    int n;
    // infinite loop for chat
    for (;;) {
        bzero(buff, MAX);
        n = 0;
        // copy server message in the buffer
        while ((buff[n++] = getchar()) != '\n')
            ;

        // send the buffer to client
        write(connfd, buff, sizeof(buff));

        // if msg contains "exit" then server exits and chat ends
        if (strncmp("exit", buff, 4) == 0) {
            printf("Server Exit...\n");
            break;
        }
    }
    return NULL;
}
```

Full Duplex Chat Application in C

Main/Driver Function (Similar in Server as well as client)

```c
// Driver function
int main()
{



    //Socket Setup



    pthread_t send_thread, recv_thread;
    // Function for chatting between client and server
    pthread_create(&send_thread, NULL, send_msg, &connfd); // Create send thread
    pthread_create(&recv_thread, NULL, receive_msg, &connfd); // Create receive thread

    // Wait for both threads to finish
    pthread_join(send_thread, NULL);
    pthread_join(recv_thread, NULL);

    // After chatting close the socket
    close(sockfd);
    return 0;
}
```

Full Duplex Chat Application in C

Server Socket Setup

```c
int sockfd, connfd, len;
struct sockaddr_in servaddr, cli;

// socket create and verification
sockfd = socket(AF_INET, SOCK_STREAM, 0);
if (sockfd == -1) {
    printf("socket creation failed...\n");
    exit(0);
}
else
    printf("Socket successfully created..\n");
bzero(&servaddr, sizeof(servaddr));

// assign IP, PORT
servaddr.sin_family = AF_INET;
servaddr.sin_addr.s_addr = htonl(INADDR_ANY);
servaddr.sin_port = htons(PORT);

// Binding newly created socket to given IP and verification
if ((bind(sockfd, (SA*)&servaddr, sizeof(servaddr))) != 0) {
    printf("socket bind failed...\n");
    exit(0);
}
else
    printf("Socket successfully binded..\n");
```

```c
// Now server is ready to listen and verification
if ((listen(sockfd, 5)) != 0) {
    printf("Listen failed...\n");
    exit(0);
}
else
    printf("Server listening..\n");
len = sizeof(cli);

// Accept the data packet from client and verification
connfd = accept(sockfd, (SA*)&cli, &len);
if (connfd < 0) {
    printf("server accept failed...\n");
    exit(0);
}
else
    printf("Server accepted the client...\n");
```

Full Duplex Chat Application in C
Client Socket Setup

```c
int sockfd;
struct sockaddr_in servaddr;

// socket create and verification
sockfd = socket(AF_INET, SOCK_STREAM, 0);
if (sockfd == -1)
{
    printf("Socket creation failed...\n");
    exit(0);
}
else
    printf("Socket successfully created..\n");
bzero(&servaddr, sizeof(servaddr));

// assign IP, PORT
servaddr.sin_family = AF_INET;
servaddr.sin_addr.s_addr = inet_addr("127.0.0.1");
servaddr.sin_port = htons(PORT); // Client connects to server's RECV_PORT

// connect the client socket to server socket
if (connect(sockfd, (SA *)&servaddr, sizeof(servaddr)) != 0)
{
    printf("Connection with the server failed...\n");
    exit(0);
}
else
    printf("Connected to the server..\n");
```

SCENARIO: Multiple Clients, connecting to a single server.

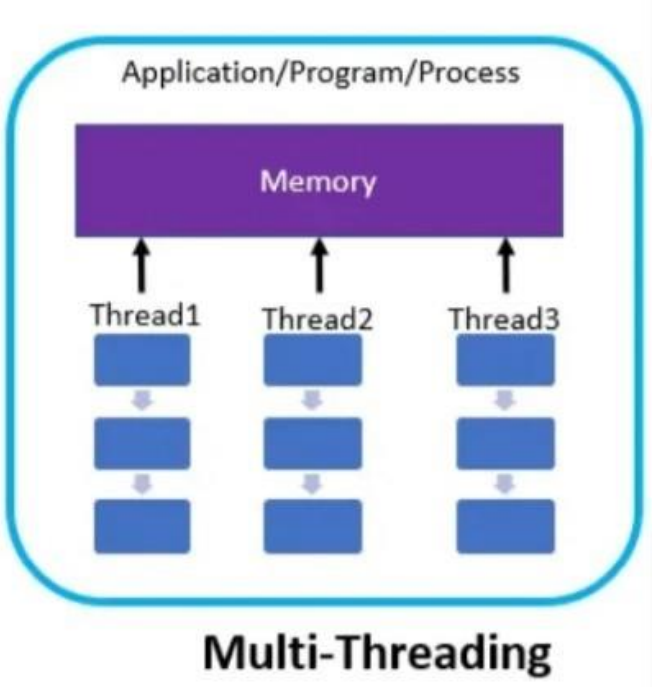
PROBLEM: The server should tend to each resource parallelly.

POTENTIAL SOLUTIONS:

A. Multithreading
B. Multiprocessing

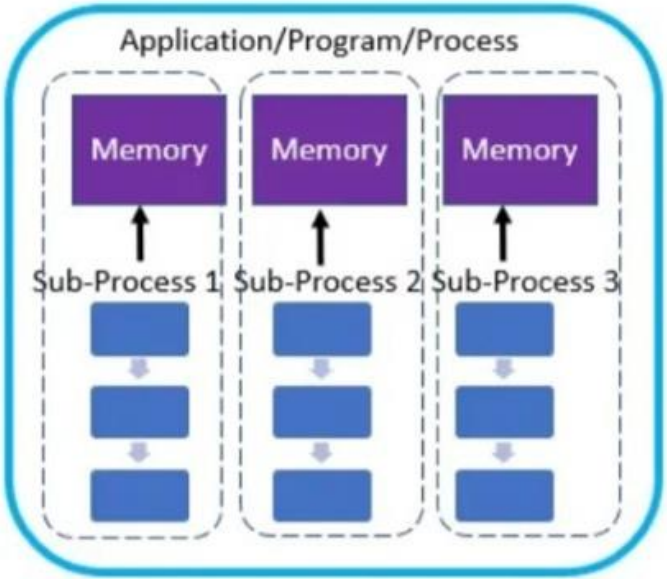
Multithreading

- Shared memory, for each thread
- Clients require separate memory because of requirement for Isolation. Not possible in Multi-Threading



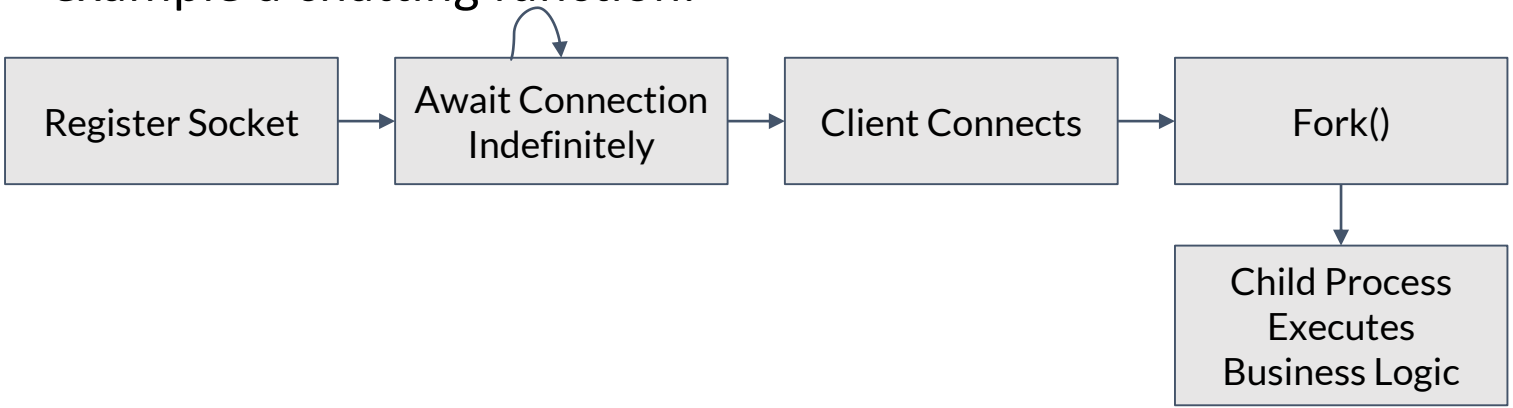
Multiprocessing

- Each process has own allocated memory.
- Data Isolation is ensured.
- Open files, too, are not shared, ensuring security.



Multi-processing

- A single socket is registered.
- Whenever a client connects with the server, fork() is called to create a child process. This process handles the business logic of the function, for example a chatting function.



Zombie Process: When a child process exits, the system keeps a record of the child process's termination status until the parent process reads (reaps) that status. This is known as ***zombie process.***

If not reaped, zombie processes accumulate, consuming system resources, causing a memory leak

- Signal is a software generated interrupt (interrupts the process). It is sent by the OS because of various reasons.

- One of the reasons is when a child process is terminated or stopped. This is denoted by SIGCHLD signal.

- In C, the signals can be handled using user defined functions. A Zombie Process cleaner could be executed on receiving the SIGCHLD signal.

```c
#include <stdio.h>
#include <netdb.h>
#include <netinet/in.h>
#include <stdlib.h>
#include <string.h>
#include <sys/socket.h>
#include <sys/types.h>
#include <unistd.h> // read(), write(), close()
#include <signal.h> // signal()
#define MAX 80
#define PORT 8080
#define SA struct sockaddr
```

```c
// Function to handle communication between client and server.
void func(int connfd)
{
    char buff[MAX];
    int n;
    // infinite loop for chat
    for (;;) {
        bzero(buff, MAX);

        // read the message from client and copy it in buffer
        read(connfd, buff, sizeof(buff));
        // print buffer which contains the client contents
        printf("From client: %s\t To client : ", buff);
        bzero(buff, MAX);
        n = 0;
        // copy server message in the buffer
        while ((buff[n++] = getchar()) != '\n')
            ;

        // and send that buffer to client
        write(connfd, buff, sizeof(buff));

        // if msg contains "exit" then server exit and chat ended.
        if (strncmp("exit", buff, 4) == 0) {
            printf("Server Exit...\n");
            break;
        }
    }
    close(connfd);
}
```

```c
// Signal handler to reap zombie processes
void sigchld_handler(int signum)
{
    // Wait for all child processes without blocking
    // Reference: https://www.ibm.com/docs/en/zos/2.4.0?topic=functions-waitpid-wait-specific-child-process-end

    /*
    waitpid() is used to wait for a specific function.
    Parameters are (pid_t pid, int *status, int options);
    pid: Determines which child process(es) to wait for. Here, pid = -1 means wait for any process
    status: Pointer to the variable where the status information of the child process is stored
    options: Different options to modify the waitpid() function behaviour. WNOHANG Returns immediately if no child has exited, instead of blocking.
    */
    while (waitpid(-1, NULL, WNOHANG) > 0)//checks if any child process has terminated. If any child has exited, the child ID is provided.
        ;//Loop is called as long as there are child processes that have exited
    //This function is used to clean up terminated functions
    /*
    When a child process exits, it turns into a zombie process.
    The system keeps a record of the child process's termination status until the parent process reads (reaps) that status using wait() or waitpid().
    If not reaped, zombie processes accumulate, consuming system resources.
    */

}
```

```c
// Driver function
int main()
{
    int sockfd, connfd, len;
    struct sockaddr_in servaddr, cli;

    // Signal handler for SIGCHLD to handle zombie processes.
    // Signal is a software generated interrupt (interrupts the process). It is sent by the OS because of various reasons.
    // SIGCHLD is generated when a child process is terminated or stopped.
    // Reference: https://www.geeksforgeeks.org/signals-c-language/
    signal(SIGCHLD, sigchld_handler);//Calls the function sigchld_handler whenever SIGCHLD interrupt is received
```

```c
// socket create and verification
sockfd = socket(AF_INET, SOCK_STREAM, 0);
if (sockfd == -1) {
    printf("socket creation failed...\n");
    exit(0);
} else
    printf("Socket successfully created..\n");
bzero(&servaddr, sizeof(servaddr));

// assign IP, PORT
servaddr.sin_family = AF_INET;
servaddr.sin_addr.s_addr = htonl(INADDR_ANY);
servaddr.sin_port = htons(PORT);

// Binding newly created socket to given IP and verification
if ((bind(sockfd, (SA*)&servaddr, sizeof(servaddr))) != 0) {
    printf("socket bind failed...\n");
    exit(0);
} else
    printf("Socket successfully binded..\n");

// Now server is ready to listen and verification
if ((listen(sockfd, 5)) != 0) {
    printf("Listen failed...\n");
    exit(0);
} else
    printf("Server listening..\n");
len = sizeof(cli);
```

```c
    // Infinite loop to keep accepting clients
    while (1) {
        // Accept the data packet from client and verification
        connfd = accept(sockfd, (SA*)&cli, &len);
        if (connfd < 0) {
            printf("server accept failed...\n");
            continue; // Don't exit, continue to wait for other clients
        } else
            printf("Server accepted a client...\n");

        // Create a child process to handle this client
        pid_t pid = fork();
        if (pid == 0) {
            // Child process
            func(connfd);  // Handle client communication
            exit(0);       // Exit child process after handling client
        }
    }

    // Close the listening socket
    close(sockfd);
    return 0;
}
```

Thank You


NATIONAL INSTITUTE OF TECHNOLOGY
DURGAPUR
उद्योगः पुरूषस्य लक्षणम्