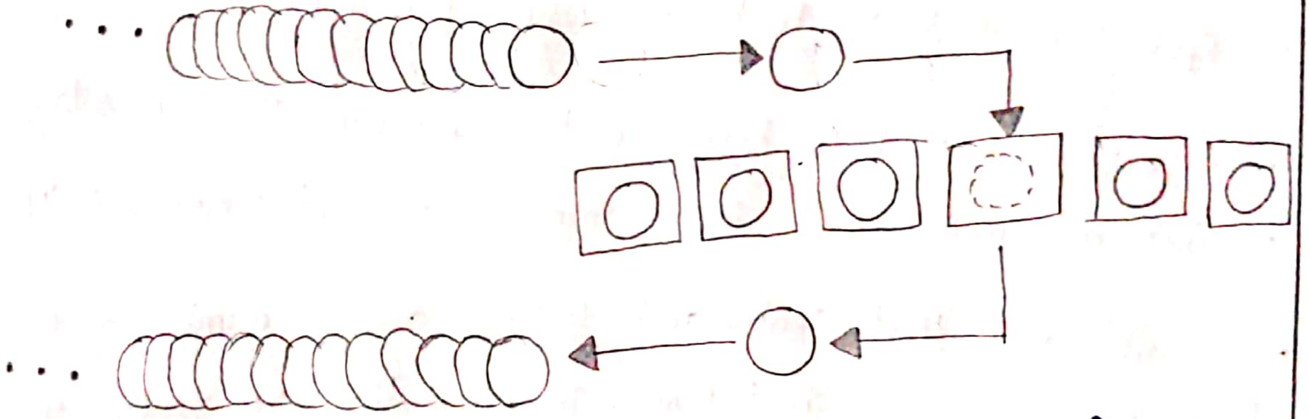
THREAD POOLING

-> what is a thread Pool ?

- A thread Pool is a collection of worker threads that effeciently Execute asynchronous callbacks on behalf of the application.
- Application can queue work items, associate work with wast. able handler, automatically queue based on a timer, and bind with I/o.



- A thread Pool maintains multiple threads waiting for tasks to be allocated for concurrent Execution by the supresing Program.

-> Advantages:

- Thread managment costs from thread evation are minimized. This leads to better response times for Processing workouts and allows for multi-threading of timer - gained workloads.

- Creating a thread Pool over Creating a new thread for each task is that thread creation & destruction overhead is restricted to the initial Creation of the Pool. which results in better Performance & better System stability.

- Possible runtime features midway through application execution due to Inability to Create threads can be avoided with simple Control to give.

→ Applications that can be benifit using thread Pools:

- An application that is highly Parllel and can dispatch a large number of small work stres asynchroniously [Ex: network I/O}

- An application that Creates and destroys a large number of threads that each sum for a short time. Thread Pooling can reduce the Complexity of thread managment.

- An application that went Perform an Exclusive unit on kernel Objects as blocks on Imcoming Events on an object. Thread Pooling can increse Performance by reducing the number of Content.

→ <u>Usage</u> <u>Scenario</u>:-

Server applications, which after launch a thread for every new request better strategy → Queue service requests from the queue, process it, and returns to the queue to get worn work.

→ Thread Pool architecture contains of :

1. worker threads that Execute the callback functions.
2. waiter threads that waits on multiple wait handler.
3. A work queue.
4. A default thread Pool for each Process.
5. A worker factory that manages the worker threads.

→ Using threads Pool functions.

Create threadPool function	
Definition	Allocates a new Pool of threads to Execute callback.

Syntax	PTP_Pool Create ThreadPool {
	PVOID reserved
	};

Parameters	reversed [This Parameter is shared & invert be null}
return value	Suceeds → returns a Pointer to a TP_Pool structures (The newly allocated thread Pool) Fails → returns NULL

Create Thread Pool wait function

Definition

create a new wait object

Syntax

```
PTP_WAIT Create ThreadPool wait [
            PTP_WAIT_callback Pfnwa,
            PVOID Pv,
            PTP_CALLBACK_ENVIRON Pebe
    };
```

Parameters

Pfnwa → callback function to call when the wait complete or timer out.

Pv → optional application defined later to pass the callback function

Pebe → A PTP → CALLBack_ENVIRON structure that defines the environment to execute in.

Succeeds → ~~S~~returns a Pointer to a TPP_WAIT structure that ~~be~~ defines wast object

fails → returns null.

close Thread Pool function

definition

closes the specified thread Pool.

Syntax

```
void closeThread Pool [
        PTP_Pool PTPP
    };
```

Parameters

PtPP → a Pointer to a TP_Pool structure that defines the thread pool that create thread Pool function return this Pointer.

return value

None

THREAD SYNCHRONIZATION:-

It is concurrent execution of two (or) more threads that share critical resources. Synchronisation of threads help in avoiding conflicts regarding critical resources. otherwise, conflicts may arise, when parallel running threads attempts to modify a common variable at the same time.

* critical section:- The region of a program that try to access shared resources and cause race condition.

* Race condition:- It typically occurs when two (or) more threads try to read, write and possibly make the decisions based on the memory they are accessing concurrently

Pseudocode:-

```
do
  {
  //entry section → wait ( ); here request are processed for entry into
                              critical section.
     *critical section*
  //exit section → signal ( ); here removes locks on critical section.
     - remaining section
  }
  while (True)
```

Solution for critical section:-

solution for a critical section Problem must satisfy following cond:-

i) Mutual exclusion:- out of group of threads, only one thread can be in its critical section at a given point of time.

ii) Progress:- If no thread is in the critical section, and if one or more thread wants to execute their critical section then only one of these threads must be allowed to get in.

iii) Bounded wait:- After a thread makes requests for getting into critical section, there is limit for how many process may get into the critical section, before threads request is granted.

widely used methods for critical section Problem:-

* Petersons solution.
* Mutex lock.
* semaphores.

SEMAPHORE:- It is a signaling mechanism, and a thread that is waiting on Semaphore, can be signaled by another thread. There are two types of semaphores.

i) Counting semaphore.

ii) Binary semaphore.

All the semaphores make use of two atomic operations i) wait and (ii) signal.

System calls in windows :-

```
(i) Create semaphore ( .
            LPsecurity - attribute ,
            lInitial count,
            lMaxium cound,
            lPName
    );
```

LPsecurity attribute :- it is a security attribute, if NULL handle can't be inherited by its child.

lInitial count :- must be greater than zero and less than or equal to Max count. Signaled state → greater than zero , Non - signaled state = 0.

lMaxium count :- max count of semaphore object.

lPName :- name of semaphore object.

```
(ii) Wait for single object (
            hHandle;
            dwMilliseconds
    );
```

hHandle :- A handle to object.

dwMilliseconds :- time - out interval in milliseconds

```
(iii) WaitforMultipleObjects (
          nCount,
          lpHandle,
          bwaitAll
          dwMilliseconds,
   );
```

nCount:- max Numbers of objects handles in array pointed to lpthreads

lpthread:- array of object handles.

bwait All:- If TRUE, returns when all objects are signaled, If FALSE, returns when any one objects is signaled.

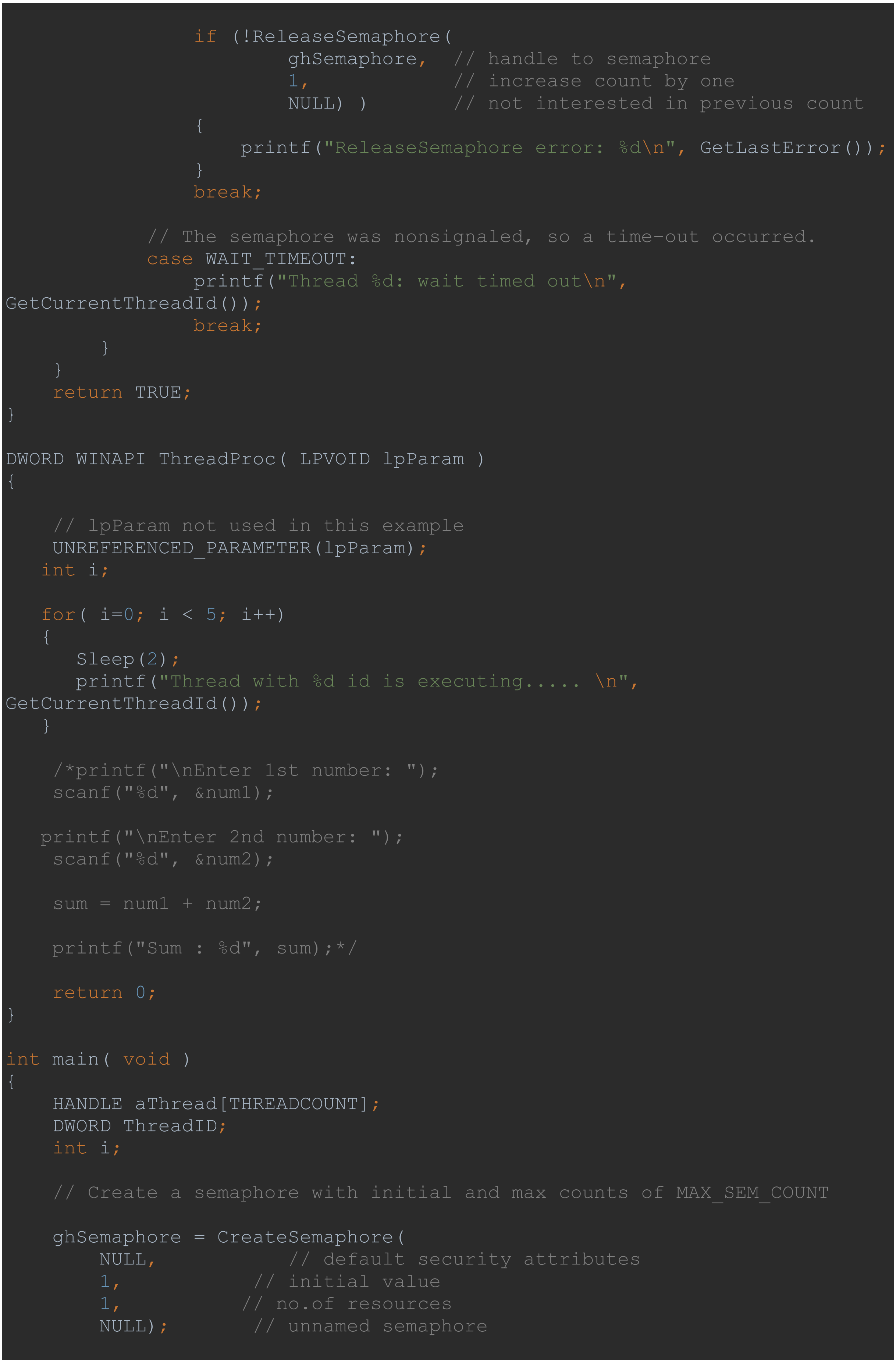
dwMillisecond:- time - out interval in Millisecond.

```
(iv) Release semaphore (
          hSemaphore,
          lRelease count,
          lP Previous count.
   );
```

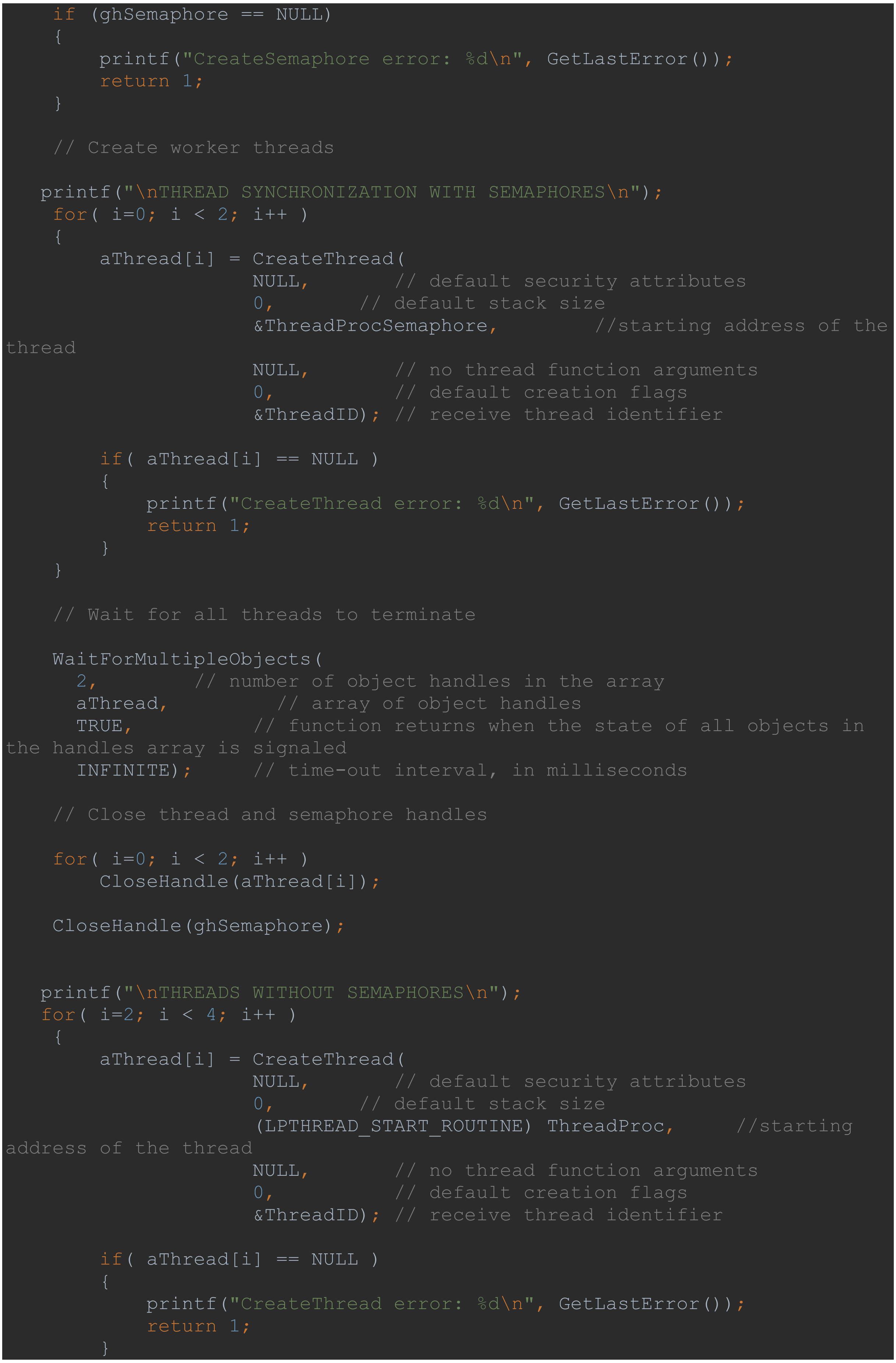
SYNCHRONIZATION

NAME	S.SHANTHAN REDDY
ROLL No.	CB.EN.U4CSE19459

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

#define MAX_SEM_COUNT 4
#define THREADCOUNT 4

HANDLE ghSemaphore;
int num1, num2, sum;

DWORD WINAPI ThreadProcSemaphore( LPVOID lpParam )
{

    // lpParam not used in this example
    UNREFERENCED_PARAMETER(lpParam);

    DWORD dwWaitResult;
    BOOL bContinue=TRUE;

    while(bContinue)
    {
        // Try to enter the semaphore gate.

        dwWaitResult = WaitForSingleObject(
            ghSemaphore,   // handle to semaphore
            INFINITE);           // zero-second time-out interval

        switch (dwWaitResult)
        {
            // The semaphore object was signaled.
            case WAIT_OBJECT_0:
                // TODO: Perform
            printf("\nThread with %d id is executing...",
GetCurrentThreadId());
            printf("\nEnter 1st number: ");
            scanf("%d", &num1);

            printf("Enter 2nd number: ");
            scanf("%d", &num2);

            sum = num1 + num2;
            printf("Sum of %d and %d : %d\n", num1, num2, sum);


            bContinue=FALSE;

                // Release the semaphore when task is finished
```

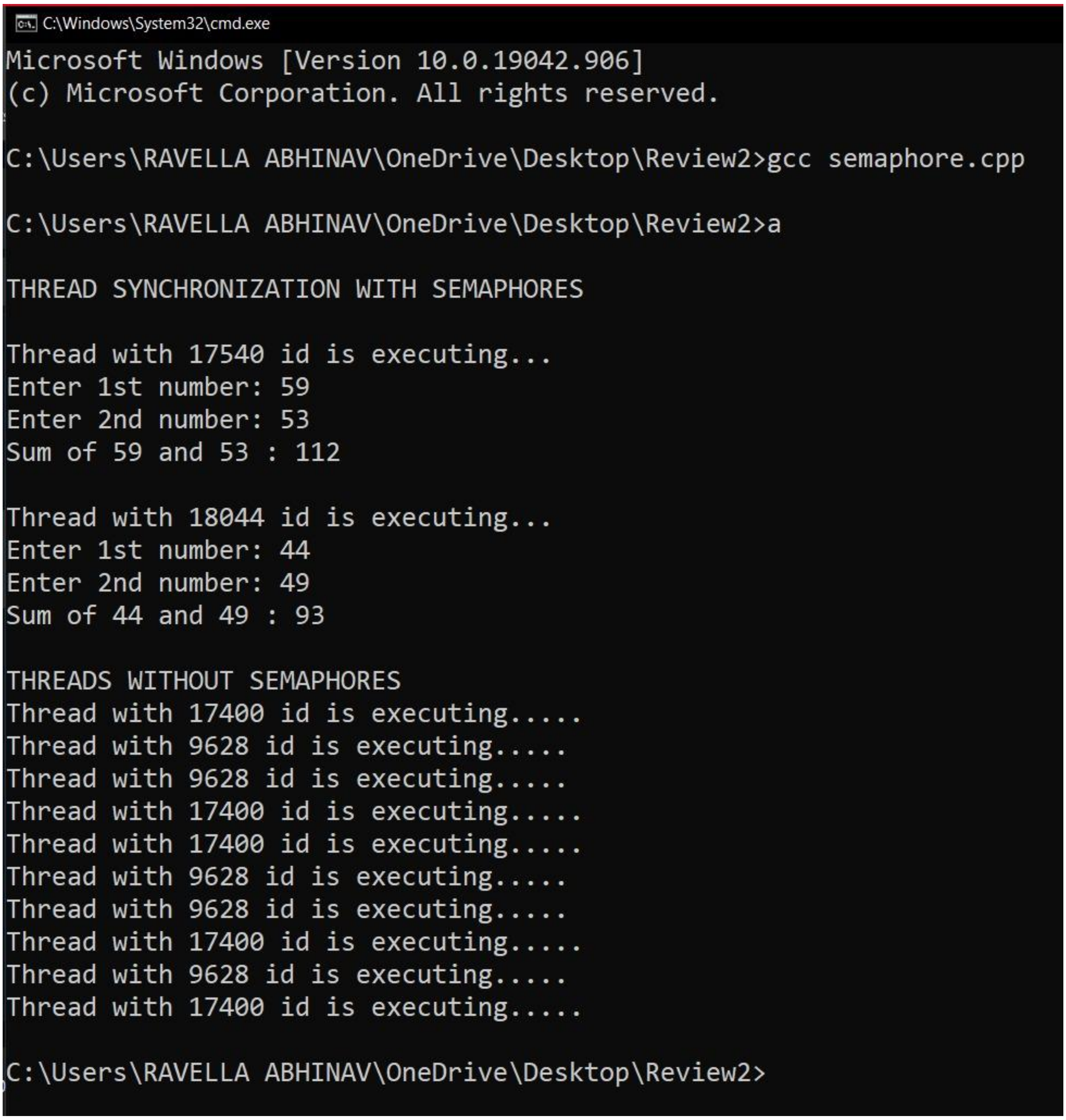


```
                if (!ReleaseSemaphore(
                        ghSemaphore,  // handle to semaphore
                        1,            // increase count by one
                        NULL) )       // not interested in previous count
                {
                    printf("ReleaseSemaphore error: %d\n", GetLastError());
                }
                break;

            // The semaphore was nonsignaled, so a time-out occurred.
            case WAIT_TIMEOUT:
                printf("Thread %d: wait timed out\n",
GetCurrentThreadId());
                break;
        }
    }
    return TRUE;
}

DWORD WINAPI ThreadProc( LPVOID lpParam )
{

    // lpParam not used in this example
    UNREFERENCED_PARAMETER(lpParam);
   int i;

   for( i=0; i < 5; i++)
   {
      Sleep(2);
      printf("Thread with %d id is executing..... \n",
GetCurrentThreadId());
   }

    /*printf("\nEnter 1st number: ");
    scanf("%d", &num1);

   printf("\nEnter 2nd number: ");
    scanf("%d", &num2);

    sum = num1 + num2;

    printf("Sum : %d", sum);*/

    return 0;
}

int main( void )
{
    HANDLE aThread[THREADCOUNT];
    DWORD ThreadID;
    int i;

    // Create a semaphore with initial and max counts of MAX_SEM_COUNT

    ghSemaphore = CreateSemaphore(
        NULL,           // default security attributes
        1,           // initial value
        1,          // no.of resources
        NULL);       // unnamed semaphore
```



```
    if (ghSemaphore == NULL)
    {
        printf("CreateSemaphore error: %d\n", GetLastError());
        return 1;
    }

    // Create worker threads

   printf("\nTHREAD SYNCHRONIZATION WITH SEMAPHORES\n");
    for( i=0; i < 2; i++ )
    {
        aThread[i] = CreateThread(
                     NULL,       // default security attributes
                     0,       // default stack size
                     &ThreadProcSemaphore,        //starting address of the
thread
                     NULL,       // no thread function arguments
                     0,          // default creation flags
                     &ThreadID); // receive thread identifier

        if( aThread[i] == NULL )
        {
            printf("CreateThread error: %d\n", GetLastError());
            return 1;
        }
    }

    // Wait for all threads to terminate

    WaitForMultipleObjects(
      2,         // number of object handles in the array
      aThread,          // array of object handles
      TRUE,           // function returns when the state of all objects in
the handles array is signaled
      INFINITE);      // time-out interval, in milliseconds

    // Close thread and semaphore handles

    for( i=0; i < 2; i++ )
        CloseHandle(aThread[i]);

    CloseHandle(ghSemaphore);


   printf("\nTHREADS WITHOUT SEMAPHORES\n");
   for( i=2; i < 4; i++ )
    {
        aThread[i] = CreateThread(
                     NULL,       // default security attributes
                     0,       // default stack size
                     (LPTHREAD_START_ROUTINE) ThreadProc,      //starting
address of the thread
                     NULL,       // no thread function arguments
                     0,          // default creation flags
                     &ThreadID); // receive thread identifier

        if( aThread[i] == NULL )
        {
            printf("CreateThread error: %d\n", GetLastError());
            return 1;
        }
    }

   WaitForSingleObject(
      aThread[2],    // handle to semaphore
        INFINITE);            // zero-second time-out interval

   WaitForSingleObject(
        aThread[3],    // handle to semaphore
        INFINITE);            // zero-second time-out interval


    return 0;
}
```

Output :



C:\Windows\System32\cmd.exe

```
Microsoft Windows [Version 10.0.19042.906]
(c) Microsoft Corporation. All rights reserved.

C:\Users\RAVELLA ABHINAV\OneDrive\Desktop\Review2>gcc semaphore.cpp

C:\Users\RAVELLA ABHINAV\OneDrive\Desktop\Review2>a

THREAD SYNCHRONIZATION WITH SEMAPHORES

Thread with 17540 id is executing...
Enter 1st number: 59
Enter 2nd number: 53
Sum of 59 and 53 : 112

Thread with 18044 id is executing...
Enter 1st number: 44
Enter 2nd number: 49
Sum of 44 and 49 : 93

THREADS WITHOUT SEMAPHORES
Thread with 17400 id is executing.....
Thread with 9628 id is executing.....
Thread with 9628 id is executing.....
Thread with 17400 id is executing.....
Thread with 17400 id is executing.....
Thread with 9628 id is executing.....
Thread with 9628 id is executing.....
Thread with 17400 id is executing.....
Thread with 9628 id is executing.....
Thread with 17400 id is executing.....

C:\Users\RAVELLA ABHINAV\OneDrive\Desktop\Review2>
```

Name:- N. Karthik Kumar Reddy

Roll. No:- CB.EN.U4CSE19444

Synchronization:-

The Process synchronization is the task of coordinating the Execution of Process in a way that no two Processes can have to the same shared data and resources.

They can also have varity of ways to coordinate multiple threads of executes.

Using critical section objects: A Critical section object Provides synchronization semilars to that provided by a mutex object, Except that a critical section can be used only by the threads of a singular Proccess. A synchronization object whose handle can be specified in one of the wait functions to coordinate the execution of multiple threads.

Enter critical section:- waits for ownership of the specified critical section object.

Leave critical section:- Releases ownership of the specified critical section object.

Initialize critical section: initialize a critical section object.

Delete critical section:- Releases all resources used by the unowned critical section object.

* In windows os, the code area requiring Exclusive access to some shared data is called as critical section.

* The structure type for working with critical section is CRITICAL_SECTION.

* Critical section in Linux os is file variable mutex pthreads _mutex_t. Before using this variable needs to be initialized write the value of the constant PTHREAD_MUTEX_INITIALIZE or call pthread_mutex_init.

Syntax:-

```
HANDLE CREATE THREAD (

        LPSECURITY_Attributes        lpthread Attributes,
         SIZE_T                       dwstack size,
        LPTHREAD_START_ROUTINE       lp·parameter Address,
        _dRU_LPØIO                   lp parameter
          PWORD                      dw creation flags,
         LP WORD                     lp ThreadId,
        );
```

lp Thread Attributes:- A pointer to a security_Attributes structure that determines weather the required returned handle can be inherited by child process.

dw stack size :- The initial size of stack in bytes.

lpstart Address :- A pointer to the application defined function to be executed by thread.

lp parameter :- A pointer to a variable to be possed to thread.

dw creation flags :- The flags to a variable to that control the creation of thread if o the thread runs immediately.

lp thread Id :- It is a thread Identifier.

Initialize critical section :- A pointer to the critical section object.

```
wait for single object ( HEANDLE    HANDLE,
                         DWORD      dw milliseconds
                       );
```

* waits until the specified object is in the signaled state are timed out.

CPU SCHEDULING

· CPU scheduling is a process that allows one process to use the cpu while the execution of another process is on hold due to unavailability of any resources like I/o etc, thereby making full use of CPU. The aim of the CPU scheduling is to make the System efficient fast and fair.

Dragonfly BSD mainly supports 2 CPU scheduling algorithms

* Light weight kernel threads (LWKT) and
* Priority Based Round Robin (PBRR)

LIGHTWEIGT KERNAL THREADS (LWKT)

The light weight kernal Threads system decouples the traditional unix notions of an execution context from a vm address space. This is similar to what many other systems such as free BSD have done.

* The LWKT has its own self-contained thread scheduler. Threads are tied to a process and can only move under special circumstances

* A Thread in LWKT Can only be preempted by an interupt thread Both fast interupts, where the interupt is handled in the current thread context, and threaded interrupts, where the LWKT scheduler switches to the interrupt thread and back when its done, are supported by the LWKT system.

Cross Process Scheduling is implemented via asynchronus inter-Processor interrupts. These messages can be batched for a given interrupt, the system Executes graceful degradation under load.

A lot of work went into seperation the LWKT Scheduler and the user Process scheduler. The LWKT thread scheduler is MP safe and uses a fast Per-CPU fixed Priority Round Robin scheme with a well defined API for communicating 3 with other Processors schedulers. The traditional BSD multilevel scheduler is implemented on top of the threads scheduler.

Advantages of LWKT :-

* Scheduler can decide to give more time for a Process which has more number of threads than Process having less number of threads.

Disadvantages of LWKT :-

* LWKT can be really slow if the time difference between Processes is very Large, (i.e) if the burst time range varies a lot then LWKT becomes slow as it gives Priority to Process with more number of threads which makes Processes with less number of threads waiting for a long Period of time.

PRiority Based Round Robin :

Priority based Round Robin (PBRR) focuses on the drawbacks of simple Round Robin architecture which gives equal Priority to all the process (process are scheduled in FCFS manner). Because of this drawback round robin architecture is not efficient with smaller cpu burst. This result increased waiting time and response time of a Process which results in the decreases in the system through put.

Priority Based Round Robin (PBRR) eliminates the defects of implementing simple Round architecture. PBRR algorithm will be Executed in two steps which will help to minimize a number of Performane parameters such as context switches, average waiting time, and Avege turnaround time.

The algorithm performs following steps:

Step 1: Allocate cpu to Every Process in RR fashion, according to the given priority, for given quantum (say k units) only for 1 time.

Step 2: After step 1, the following are done.

a) processors are managed in increasing order or their remaining cpu burst time in the ready queue. New priorities are assigned according to the remaining cpu burst of Process, the Process with shortest cpu time is assigned with highest Priority.

b) The Processes are Executed according to the New Priorities based on the remaining cpu bursts and Each Process gets the control of the cpu until they finished their Execution.

Advantages of PBRR :

- * Dramatically improves average response time.
- * Improves efficiency of context switching, average waiting and turnaround time.
- * Though every Process gets a chance to run, the higher Priority Processor will complete first, and they need not wait for a very long time.