CI5250 – Computing Systems

Operating Systems Assignment

Deadline: 2nd of December 2019 @ 23:59

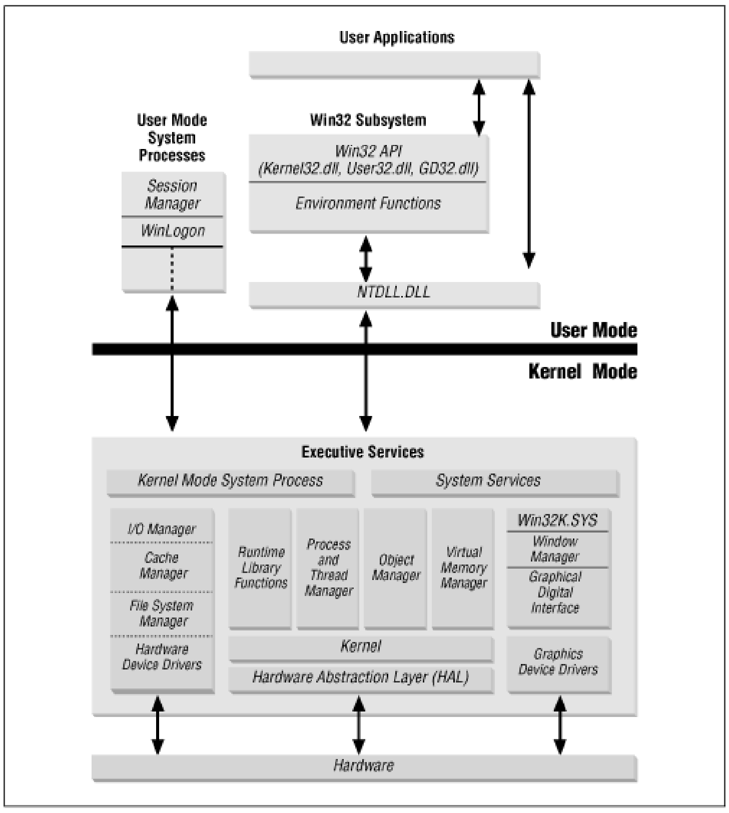
Name: Leandro Maglianella **k-number: 1833503**

Put your answers in this answer sheet which should have your name and k-number at the top. Each question should be answered on a separate page. **You must reference any external sources that you have used to derive your answers on the last page to avoid plagiarism charges! Submit the answer sheet as a WORD document or as a PDF document.**

Question 1: Windows Architecture - (10 marks)

(System diagram + ½ page of description and discussion)

The operating system is the most important software running on a computing system, it acts as an interface between the hardware and the applications. It controls and manages the processes and establishes the amount of resources (such as CPU time, memory or disk space) available to them. Windows NT is a family of operating systems designed by Microsoft since 1992, of which Windows 10 is the most recent. This operating system is supported by 32-bit and 64-bit platforms, it is multi-user, multitasking (it can perform multiple processes at the same time by sharing a core through the use of context switches) and multiprocessing (it can run more processes at the same time using more cores of the CPU). It is a graphic object-oriented operating system written in C, C++ and Assembly: it can be interacted through icons, windows, buttons, list boxes etc. and all its components, including files, resources, devices and processes are objects. The Windows operating system design is logically layered and divided into user mode and kernel mode.



The major components of user mode are user applications and subsystems, which have a client/server relationship. In fact, each subsystem supports the application programming interfaces (API) of a different operating system and allows user applications native of different operating systems to work. The main subsystems are the Win32 subsystem, the OS/2 subsystem, the Linux subsystem and the POSIX subsystem.

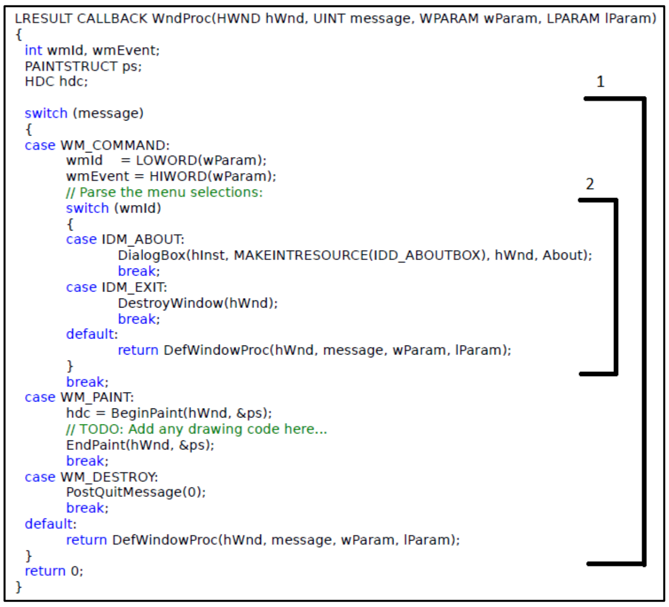
In kernel mode, executive services emulate the operating systems supported by the subsystems and, through managers and drivers, they deal with I/O, object management, security and process management. Finally, between the kernel and the hardware is the hardware abstraction layer (HAL), it provides a platform on which the kernel can function reliably by hiding the hardware differences. In this way it increases the portability and scalability of the operating system. It also deals with process and thread scheduling and handles exceptions, traps and interrupts.

There are many reasons for Windows popularity. Windows architecture was developed keeping in great importance design goals such as compatibility, reliability, usability, portability, scalability, performance and extensibility. Above all, the innovations that brought the Windows operating system to success were the presence of a graphical user interface (which provide windows, menus, buttons etc.) and the implementation of a standardized Application Programming Interface (Win32). The goal of these functionalities was to make interaction with the operating system easy for anyone, both users and developers. Taking advantage of these features, Windows had the potential to emerge early and to be adopted as operating system by many companies. In this way it subsequently expanded to the rest of the world public. Most existing software is compatible with Windows and many applications, games (i.e. Xbox and the collection of APIs DirectX) and free tools (i.e. Microsoft Office) are developed specifically for Windows.

Question 2: Windows Event Handling - (10 marks)

(½ page of description and discussion plus source code snippets)

An event-driven system is a system that works by reacting to external events. This is a basic programming technique for interactive applications in which the inputs of a user (expressed through physical and real tools) are fundamental, in fact only in this way the user is really able to interact with the software. In our case, the Windows Operating System and all the applications that are part of it are event-driven: moving the mouse pointer, pressing the mouse buttons or pressing the keyboard keys, messages are sent to the operating system, they are interpreted by it and converted into operations to be executed in a specific application. This is a main feature in the functioning of a computer and for this reason the existence of systems of this kind is obviously beneficial. Without this methodology only software that do not require input during their execution would exist, which generally are neither many nor particularly useful.

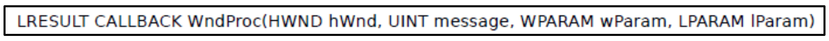


```
LRESULT CALLBACK WndProc(HWND hWnd, UINT message, WPARAM wParam, LPARAM lParam)
{
 int wmId, wmEvent;
 PAINTSTRUCT ps;
 HDC hdc;

 switch (message)
 {
 case WM_COMMAND:
        wmId    = LOWORD(wParam);
        wmEvent = HIWORD(wParam);
        // Parse the menu selections:
        switch (wmId)
        {
        case IDM_ABOUT:
                DialogBox(hInst, MAKEINTRESOURCE(IDD_ABOUTBOX), hWnd, About);
                break;
        case IDM_EXIT:
                DestroyWindow(hWnd);
                break;
        default:
                return DefWindowProc(hWnd, message, wParam, lParam);
        }
        break;
 case WM_PAINT:
        hdc = BeginPaint(hWnd, &ps);
        // TODO: Add any drawing code here...
        EndPaint(hWnd, &ps);
        break;
 case WM_DESTROY:
        PostQuitMessage(0);
        break;
 default:
        return DefWindowProc(hWnd, message, wParam, lParam);
 }
 return 0;
}
```

A Windows application is composed of the WinMain function, of a windows class declaration and of the WndProc function. Let us focus on this last component. WndProc is different in each windows application, here the programmer establishes how the application should handle the messages. Usually the action to be performed is distinguished through a switch statement.

Let us analyse the exaple code above:

WndProc is invoked with four parameters and returns a value



```
LRESULT CALLBACK WndProc(HWND hWnd, UINT message, WPARAM wParam, LPARAM lParam)
```

of type LRESULT, it is an integer value and it is the result of the message processing and consequently depends on the message. *hWnd* is a handle to the window, *message* (UINT type) is the message type to be managed, *lParam* and *wParam* are both integer values, they contain additional information and depend on the message (they can for example contain the coordinates of the mouse cursor).

The code uses two switch statements:

The first one {1} to distinguish between messages (WM_COMMAND, WM_PAINT, WM_DESTROY or default). The second one {2} is in the WM_COMMAND case and distinguishes whether the id of the selected menu item is IDM_ABOUT, IDM_EXIT or default.

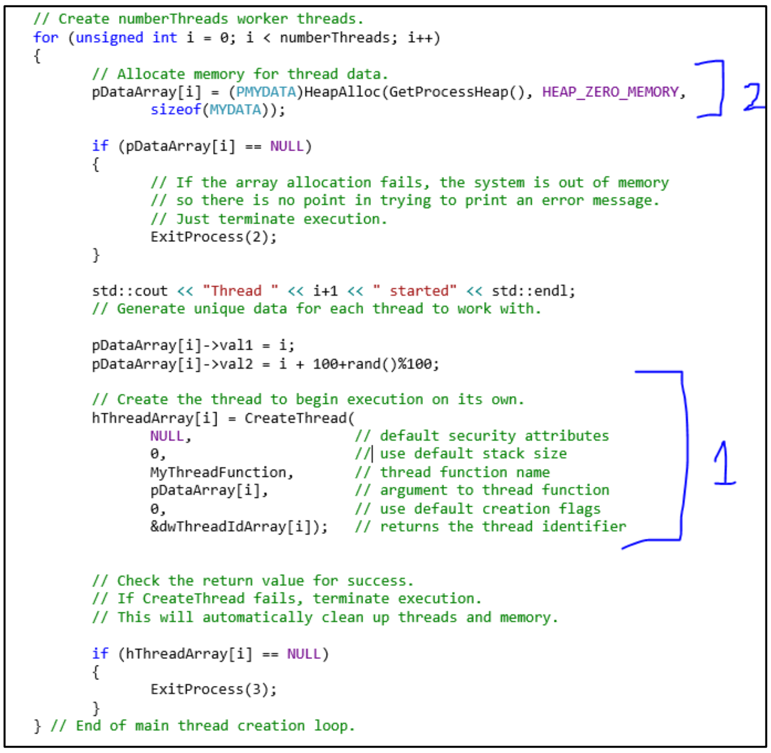
WM_COMMAND message is sent when the user command/request something to the operating system (for instance, when the user clicks on a menu button), WM_PAINT message is sent when a window must be drawn or modified (for instance if the window is resized), WM_DESTROY message is sent when a window is destroyed.

IDM_ABOUT indicates the id of the menu item "About", in this case when the menu item is selected a dialog box is created using the DialogBox function. IDM_EXIT indicates the id of the menu item "Exit", when it is selected the window is closed using the DestroyWindow function. The default case calls the DefWindowProc function which ensure that every message is processed, its action varies by message type.

Question 3: Resource Management (10 marks)

(Table+graph and ½ page of description and discussion)

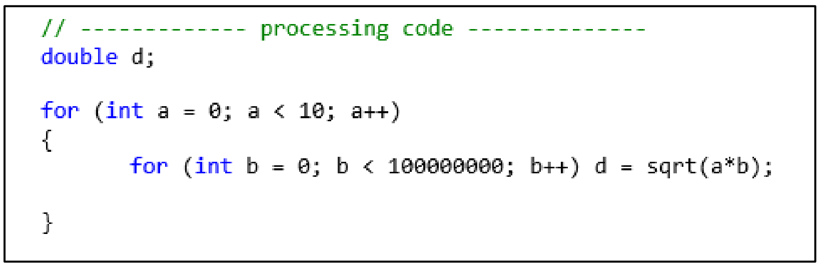
A program running in memory is a process. Each process works in its own protected environment using its own resources and it can communicate with other processes but it cannot interfere with them. The operating system manages resources for each process. At least one thread is associated with each process, which are simply a sequence of instructions. The resources of a process are shared among all the threads that compose it.



```
// Create numberThreads worker threads.
for (unsigned int i = 0; i < numberThreads; i++)
{
        // Allocate memory for thread data.
        pDataArray[i] = (PMYDATA)HeapAlloc(GetProcessHeap(), HEAP_ZERO_MEMORY,
                sizeof(MYDATA));

        if (pDataArray[i] == NULL)
        {
                // If the array allocation fails, the system is out of memory
                // so there is no point in trying to print an error message.
                // Just terminate execution.
                ExitProcess(2);
        }

        std::cout << "Thread " << i+1 << " started" << std::endl;
        // Generate unique data for each thread to work with.

        pDataArray[i]->val1 = i;
        pDataArray[i]->val2 = i + 100+rand()%100;

        // Create the thread to begin execution on its own.
        hThreadArray[i] = CreateThread(
                NULL,                   // default security attributes
                0,                      // use default stack size
                MyThreadFunction,       // thread function name
                pDataArray[i],          // argument to thread function
                0,                      // use default creation flags
                &dwThreadIdArray[i]);   // returns the thread identifier


        // Check the return value for success.
        // If CreateThread fails, terminate execution.
        // This will automatically clean up threads and memory.

        if (hThreadArray[i] == NULL)
        {
                ExitProcess(3);
        }
} // End of main thread creation loop.
```

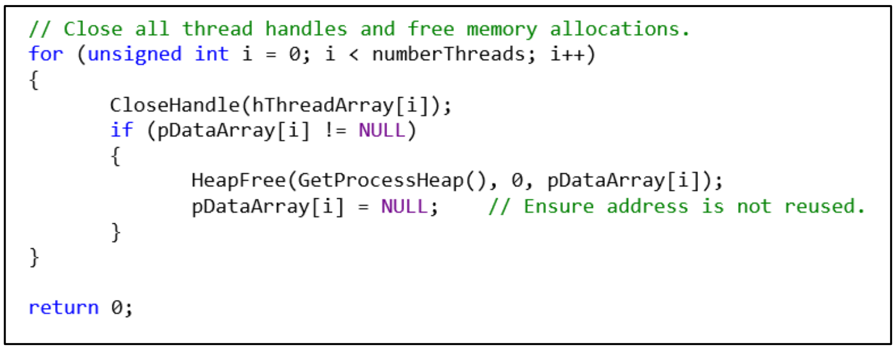
In our case (ResourcesApp.exe), the application requests processing and memory resources to the operating system through the CreateThread function {1}, which creates *i* threads. This function has six parameters, let us describe the major ones: pDataArray is an array of heap memory blocks allocated using the HeapAlloc function {2}, pDataArray[i] refers to a particular memory block reserved for a thread, MyThreadFunction is the function called by each thread and is performed in its memory block.

In this case, MyThreadFunction is used in each thread to calculate a huge amount of multiplication and square roots:



```
// ------------- processing code --------------
double d;

for (int a = 0; a < 10; a++)
{
        for (int b = 0; b < 100000000; b++) d = sqrt(a*b);

}
```

When all threads finish their execution, all thread handles are closed using the CloseHandle function and all the blocks of heap memory previously reserved for thread executions are freed:

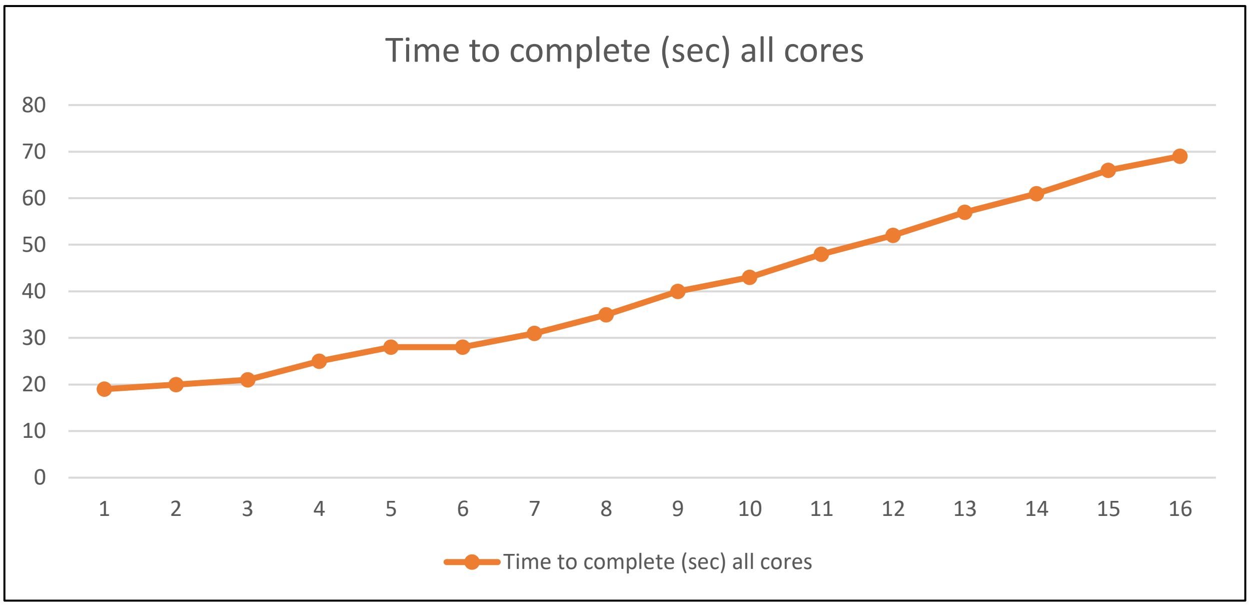


```
// Close all thread handles and free memory allocations.
for (unsigned int i = 0; i < numberThreads; i++)
{
        CloseHandle(hThreadArray[i]);
        if (pDataArray[i] != NULL)
        {
                HeapFree(GetProcessHeap(), 0, pDataArray[i]);
                pDataArray[i] = NULL;    // Ensure address is not reused.
        }
}

return 0;
```

Running the application with increasing processing threads, the following table was populated:

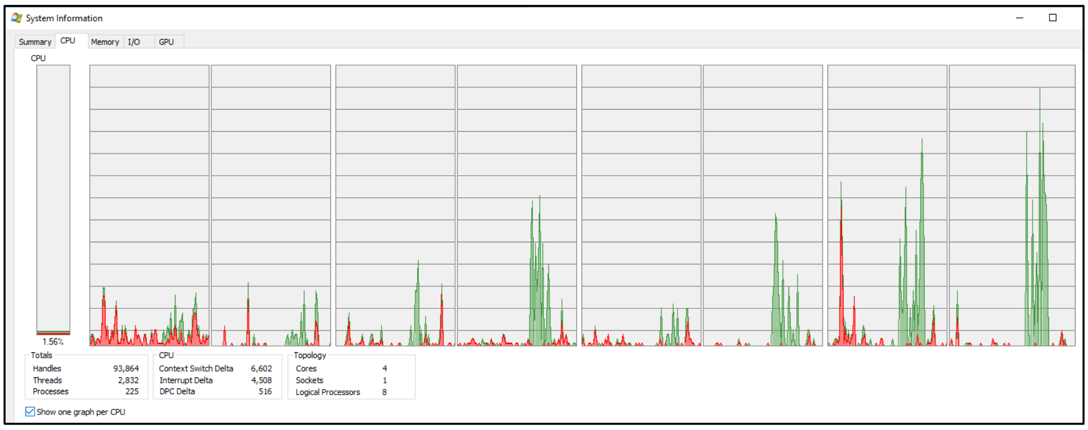
Processing threads	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
Time to complete (sec) all cores	19	20	21	25	28	28	31	35	40	43	48	52	57	61	66	69

Line graph plotted using the table data:



A CPU with 4 cores and 8 logical processors was used. A core is a processing unit, each core can simultaneously run one or more threads. In this case, thanks to hyper-threading, each core can run 2 threads (that is why we have 4 × 2 = 8 logical processors). The Windows Operating System uses the scheduler in the kernel to manage processes. This is a pre-emptive scheduler, so each process or thread is assigned a time slice during which it can use a processor. This ensures a fast, fair and efficient CPU utilisation. Looking at the graph it can be observed that the time to complete increases more quickly when more than 8 threads are executed, this happens because the CPU must perform many more context switches to run threads in parallel.
Let’s now increase the amount of threads running to analyse the increase in CPU utilisation:

1 thread, overall CPU utilisation: 14%



Totals

Handles	93,864
Threads	2,832
Processes	225

CPU

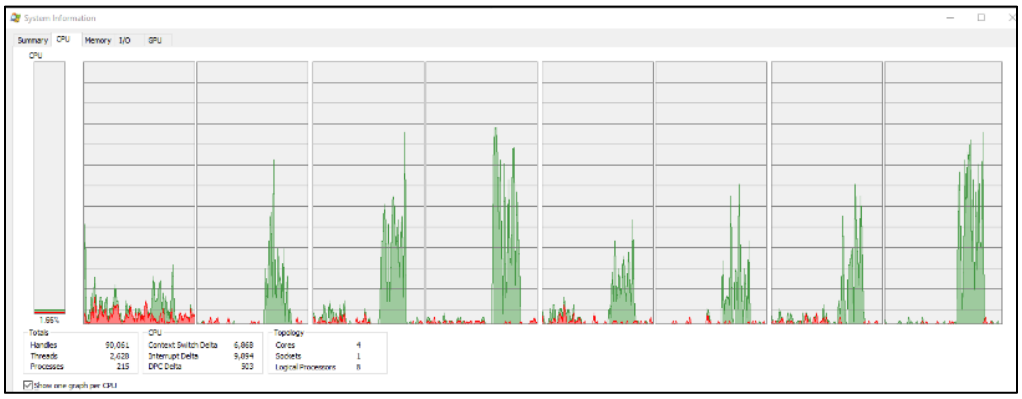
Context Switch Delta	6,602
Interrupt Delta	4,508
DPC Delta	516

Topology

Cores	4
Sockets	1
Logical Processors	8

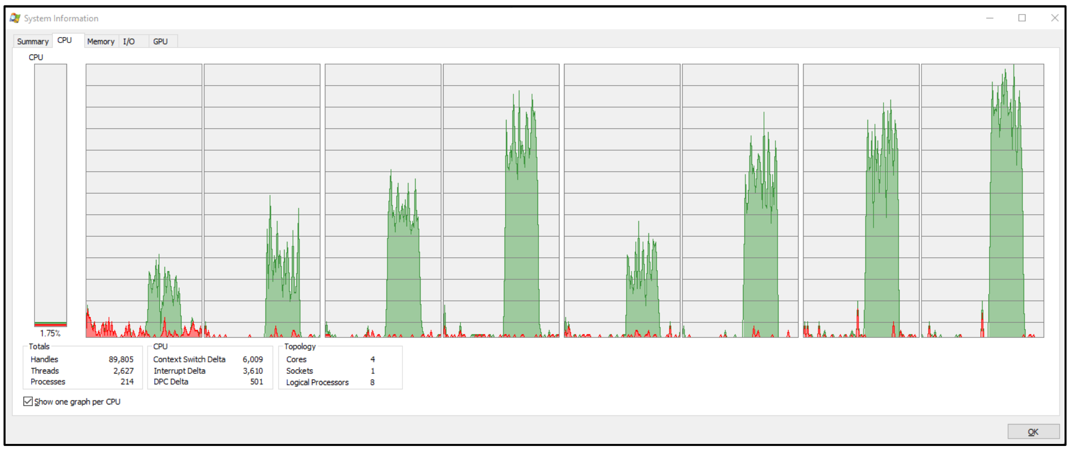
☑ Show one graph per CPU

2 threads, overall CPU utilisation: 27%



```
Totals                     CPU                              Topology
Handles         90,061     Context Switch Delta    6,868    Cores                 4
Threads          2,628     Interrupt Delta         9,094    Sockets               1
Processes          215     DPC Delta                 903    Logical Processors    8

[✓] Show one graph per CPU
```

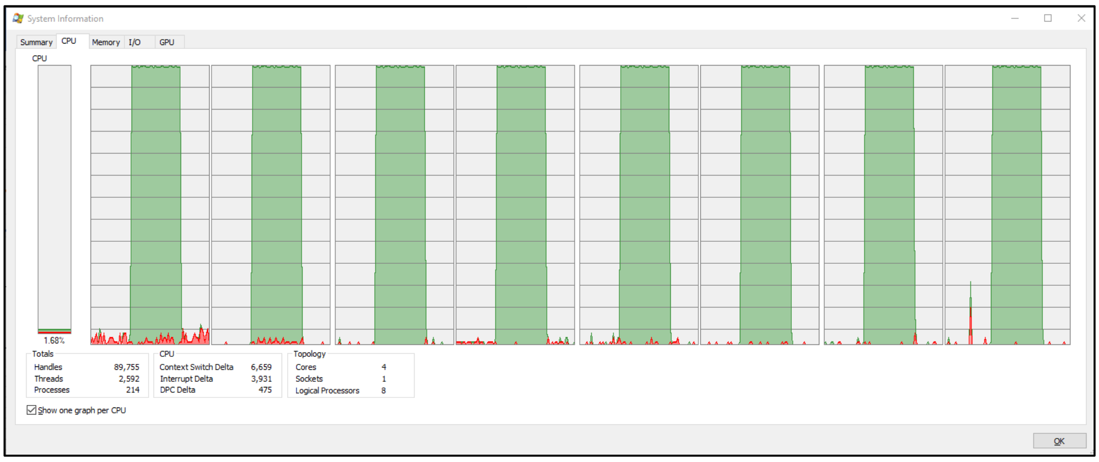
4 threads, overall CPU utilisation: 52%



```
Totals                     CPU                              Topology
Handles         89,805     Context Switch Delta    6,009    Cores                 4
Threads          2,627     Interrupt Delta         3,610    Sockets               1
Processes          214     DPC Delta                 501    Logical Processors    8

[✓] Show one graph per CPU
```

6 threads: 77%
7 threads: 89%

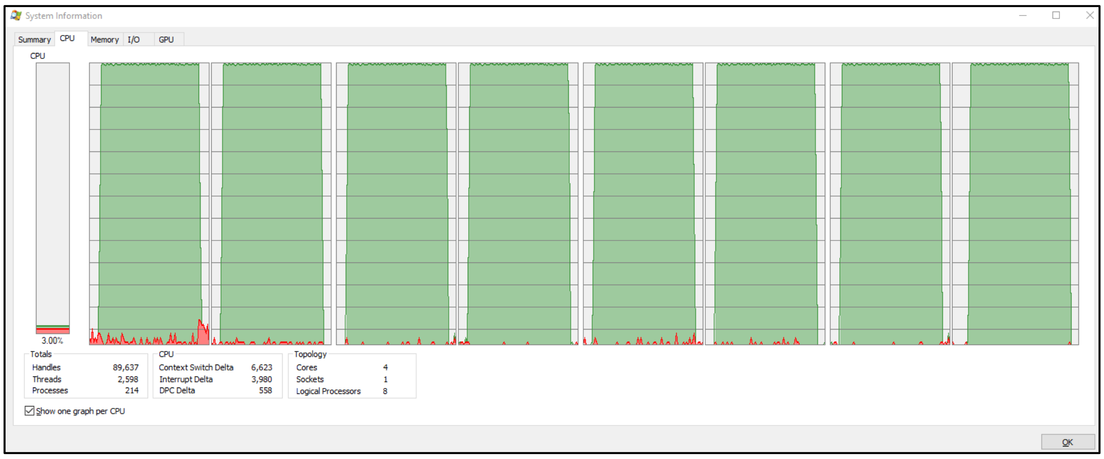
8 threads, overall CPU utilisation: 100%



```
Totals                     CPU                              Topology
Handles         89,755     Context Switch Delta    6,659    Cores                 4
Threads          2,592     Interrupt Delta         3,931    Sockets               1
Processes          214     DPC Delta                 475    Logical Processors    8

[✓] Show one graph per CPU
```

As expected, 8 threads employ 100% of the CPU since it has 8 logical processors and therefore all must be used and be busy. By running more than 8 threads, the CPU will be forced to execute more than one thread for each logical processor: this is possible thanks to the multithreading ability.

16 threads, 100%:



Totals

Handles	89,637
Threads	2,598
Processes	214

CPU

Context Switch Delta	6,623
Interrupt Delta	3,980
DPC Delta	558

Topology

Cores	4
Sockets	1
Logical Processors	8

After having carried out this analysis it is evident that the operating system is very efficient at handling the increasing processing demand: in fact, both the time to execute all the threads and the CPU utilisation increase almost linearly. This is a very interesting result, especially when more than 8 threads are executed as the CPU is forced to perform a much higher number of context switches and therefore one might expect slowdowns that are instead avoided.

References

(Reference any books, web pages, documents and lecture notes you have used)

- *Architecture of Windows NT.* (2019) Available at: https://en.wikipedia.org/wiki/Architecture_of_Windows_NT (Accessed: 20 November 2019).

- *Build desktop Windows apps using the Win32* API. (2019) Available at: https://docs.microsoft.com/en-us/windows/win32/ (Accessed: 26 November 2019).

- Helendi, A. (2018) *What Is Event-Driven Programming And Why Is It So Popular?* Available at: https://dzone.com/articles/what-is-event-driven-programming-and-why-is-it-so (Accessed: 26 November 2019).

- Lecture notes, worksheets and solutions available at: https://canvas.kingston.ac.uk/courses/15166/pages/operating-systems-topic (Accessed: 20 November 2019).