Development report

Task Launcher

Artem Makurin

I’ve started working on the solution with selecting the type of the future application. As the requirements don’t call for a GUI of any kind, I assumed that a basic Console app would suffice, since the original goal of the program was to save the output to the file and not to display it. Additionally, when creating a project, I had to specify the version of the .NET Framework that it would target, and this part was mandatory. .NET Framework version 6.0 was selected as a result.

Next, a basic menu had to be implemented to enable the user to work with the program. The menu, conveniently, forms the “skeleton” of the program’s logic. I’ve implemented the menu using the *switch* statement. Cases of the *switch* statement are navigated by reading a number from user. Users need to confirm their inputs by pressing Enter and this might not be obvious to all of them, so the input prompts that the program gives to users should indicate that, just in case. The menu has four options, two of which display variables and update dynamically.

Text

Description automatically generated

The latter changes the *exit* variable and stops the main loop of the program from executing, effectively exiting the program. If user enters a number different from 1 through 4, they’re warned that it is not a valid input and additionally, if user enters text or symbols the wrong input format exception is shown.

*Console.Clear()* & empty *Console.ReadLine()* methods are used throughout the program to de-clutter the screen and make the information more readable. The default values for file path and polling interval are "C:\\Windows\\System32\\NotePad.exe" and 3000 milliseconds, respectively.

Pressing number 1 in the main menu opens the path selection screen. There, a user is told that they can use double backslash (\\) or forward slash (/) to input the file path and name of the task that needs to be launched. C# can understand these types of paths due to the way C# handles single backslashes: in combination with single letters they form escape sequences. This behavior, however, can be avoided by using *@”path”* which tells the compiler to treat everything between quotes literally or using two other ways of expressing the folder structure: the abovementioned double backslash and forward slash. After the input is read from user it is sent into the function *CheckPath()* as an argument. The function checks if the provided file exists and if it is of .exe extension, then returns **true** if it does. If the file is deemed valid, it is saved in the *path* variable and displayed in the main menu.

Pressing number 2 brings up the polling interval adjustment screen. By default, it is set to 3000 milliseconds. The input is read from user, first checked to be a number, and then compared to 500. The CPU Load percentage must be calculated over time, so user is limited from setting the interval lower than 500 milliseconds. When a valid interval is read it is saved.

What happens after pressing number 3 in the menu is the most important part of the program. First, the *Sleep()* method is declared to be used throughout the polling process. I want to avoid using *Thread.Sleep()* as much as possible as its purpose is different than making the process wait a certain amount of time. Afterwards, we set the *UseShellExecute* to **true** to avoid a nasty crash from occurring. As far as I know, this crash is an open issue and hasn’t yet been fixed. (<https://github.com/dotnet/runtime/issues/28005>) A new process is created and given the desired start information. Then, a method for calculating the percentage of CPU usage is defined. It works by taking two measurements of processor time and dividing the difference in use by time between measurements multiplied by the number of CPU cores in the computer. Then, two loops are started that continue to run until the process has exited. Inside the loop, the process is refreshed, the performance measurements are taken and displayed in the console for visual feedback.

Graphical user interface

Description automatically generated with low confidence

Finally, a *StreamWriter* is used to call an *AppendText()* method which creates a .csv output file of the measurements, if it hasn’t already existed. The loop is then paused to wait for the next measure, according to the saved interval.

Throughout the program, *try-catch* blocks are used for activities that can cause crashes. To prevent errors, all variables that get assigned in *try* blocks have saved default values.