**Project Name:** Multiprocessing Menu

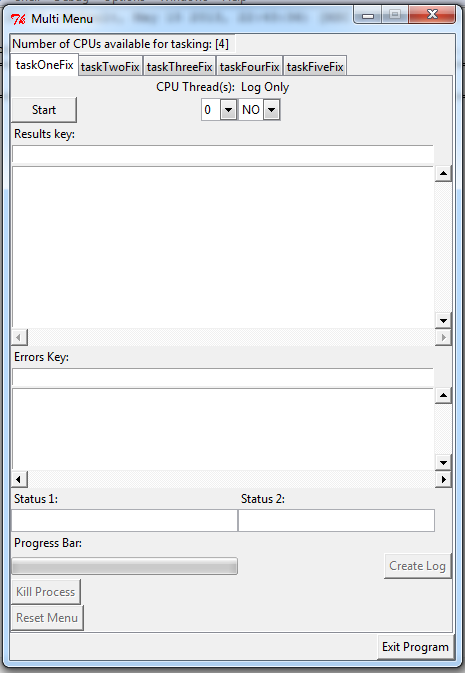
**Program:** Python 2.7.5

**Modules:** ctypes, functools, multiprocessing, numpy, os, re, sys, time, Tkinter, tkFileDialog, ttk

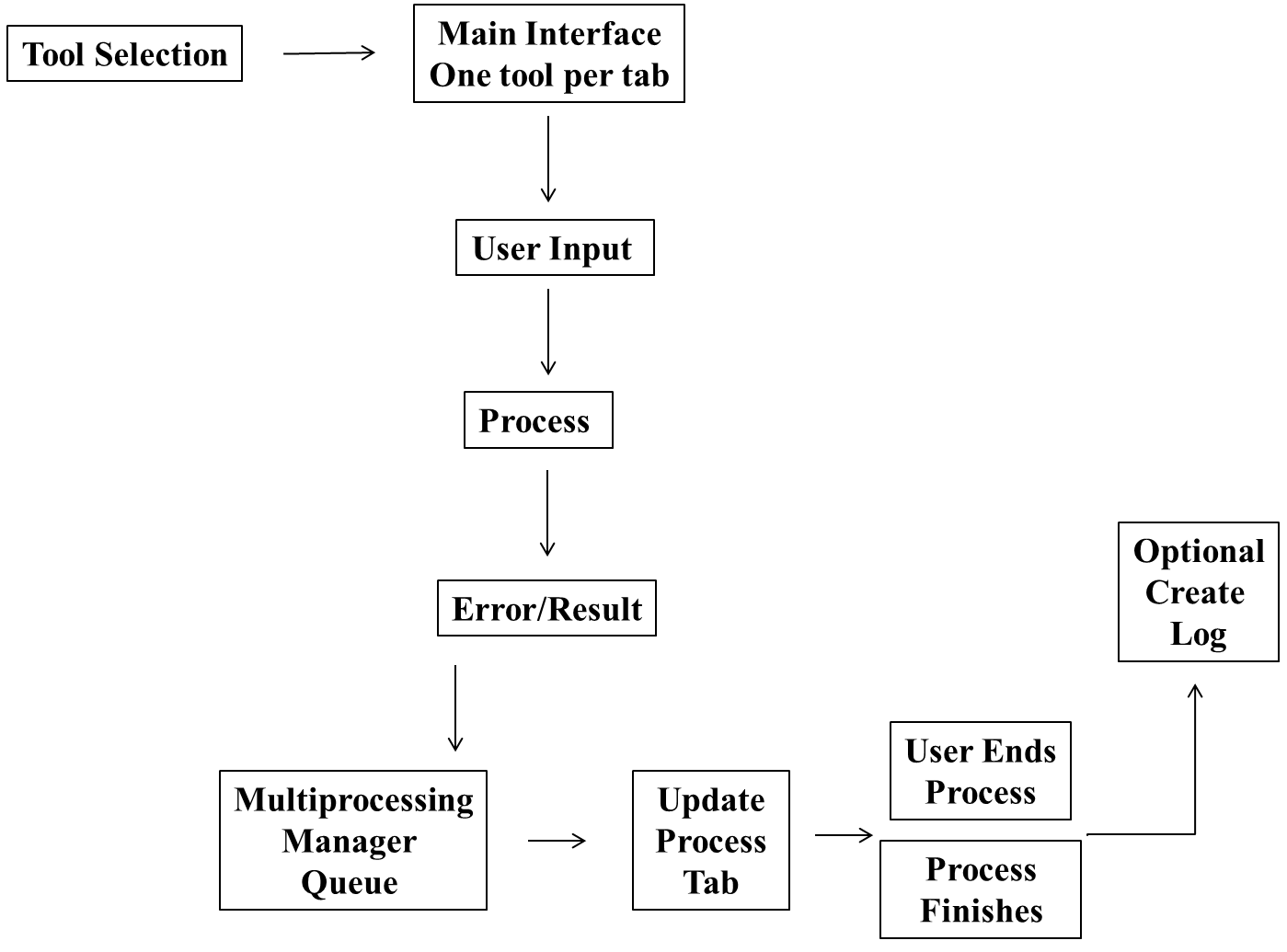
**Lines of Code:**

1. Main Interface: 994
2. Test Process: 49

**Purpose:** To create a single interface capable of handling multiple processes scaled to the number of working threads available via the CPU.



**Program Flow Diagram:**

****

**Walkthrough:**

The application begins by initiating a menu containing tools that the user can chose from. The tools are derived from an accompanying tools folder in the parent directory that contains the main multi menu script. Each tool’s naming nomenclature must contain a process number then a short name that must match the internal primary function of the process.

Expansions for a pre- and post- processing function will be defined by “pre\_” and “post\_” process name from the associated file name.

The user can select one or many tools to fill their interface. However, the number of tools that can be selected should remain limited by the tools needed at the time and having the user take into account computer resources available for the duration of the tool’s run.

After the user choses their tool or tools the main interface loads with named tabs, minus the starting number designator, for each tool chosen. Each tab contains a start button, CPU thread assignment combo box, log only combo box, results and error key list, two status entries, a progress bar, kill process button, reset button and create log button. Each tab’s element is accessed and updated via a series of get/set attribute off of the base class variable assignment (i.e. self) and the process name established through the tool’s name.

The results and errors keys are defined within the tool’s script accessible via dynamic variable attribution and retrieval.

Every menu element has the potential for this same attribution and retrieval.

Upon building of the main interface, a menu updater is initiated and refreshed every 100 milliseconds. This update also covers CPU resource usage via each tool’s CPU combo box and the parent menu’s CPU resource entry.

Currently there are five duplicate process tools that are different in name only to test the application. However, the internal structure is built to handle the input and output folder before beginning the actual process.

The structure for the main interface is currently designed to handle a pre‑process action in preparation for known tasks that are planned for integration at a later time. An example of such a task is the run of a process that establishes a list of variables for explicit use in the primary process. This concept has the potential for expansion (e.g. post-process cleanup) depending on analyst needs. The only thing that really matters upon creation of subsequent process capability is the ability to integrate the process without interfering with the host’s shell structure.

Once a tool is initiated the respective results and error list keys are loaded and once the process starts generating results the respective list boxes are updated as well.

The user has the ability to stop a currently operating tool by selecting the kill button. The process tying up the threads are terminated and the user can now reallocate CPU resources to another tool that has not started by selecting “0” for CPU resources in the recently ended tool’s menu.

Once a tool’s process is ended or finished the user can create a log of what was completed. The user selects the create log button and a popup menu appears asking what folder the analyst wants to save the log. The log’s name consists of the tool name followed by a joined time.ctime string while the contents consist of the time the process started and ended as well as a section for results and errors with respective keys for each.