Task 1: ReadAble Logs

Features

- **ReadAble logfiles -** See critical Errors or Warnings with minimal Searching
- **ExtendAble whitelist -** Create your own Filter to tweak what your logs contain
- **AdjustAble schedule -** Create your own Schedule by setting up your own Cronjob

Download

ReadAble Logs

How to Use

There is no need to install additional dependencies since the script only uses modules from the Python Standard Library and common Linux commands. It runs without command line arguments. Upon start, it announces the ensuing creation of a cronjob.

```
jan@linux:~$ python3 readable_logs.py

| ReadAble Logs |

Create a cronjob to execute this script daily on working days.

Press Enter to modify the exemplary cronjob on the last line...
```

Pressing the Enter key displays the crontab file where an exemplary cronjob is located on the last line. After this, the script is executed periodically as determined by the cronjob and the log files (system log file, simplified log file labeled by date) can be found in the home-directory.

Running the script as root or with root privilege creates the cronjob in the global crontab file and the log files are written to /home.

Dependencies

Python Standard Library:

- datetime
- re
- subprocess

Technical Summary

This script produces readable log files which are updated on a periodical basis, either in the calling users home-directory, or in /home by running it as root or with root privilege. The system log file (e.g. Ubuntu: /var/log/syslog) is copied to the calling users home-directory or /home for root. From there, it is read to memory and the simplified content is written to a file labeled by the current date. The original content of the system log file is reduced to those lines containing one of the severity keywords (EMERGENCY, ALERT, CRITICAL, ERROR, WARNING, NOTICE, INFORMATIONAL, DEBUG) in the format "<keyword>:" followed by an error message. Furthermore, all content besides the timestamp and the error message are abbreviated. To generate the simplified log files on a regular basis, the user gets the chance to approve a cronjob.

Features & Extensibility

- **Whitelist & Regex**: For rough filtering, the script checks if any of the whitelist entries are in a given line and skips it otherwise. For fine filtering within a line, the Python re module is used. The implemented patterns search for timestamps and error messages. The timestamp is assumed to be located at the start of a line as a non-empty string without intermittent whitespace. The error messages are assumed to start after a whitespace with one of the severity keywords in all caps, followed by a colon and a non-empty string running until the end of the line.

  To extend the filter in the trim_log method, one can modify the whitelist and/or the regex. While the whitelist determines if a line is considered, the regex determines what part of this line is shown. Note that when modifying the whitelist it may be necessary to modify the regex used to filter the error messages. To test the regex, one can either work in the Python command line with the re module or use a web-tool like regex101.

- **Scheduling**: The exemplary cronjob

  ```
  0        0        *        *        1-5        <path to python3> <path to script>
  ```

  is determined by the initialization of the CronJob object. Its arguments correspond to the different fields of the crontab file and can be modified.

- **Access control**: The files and permissions are specified in two lists in the main function and passed to the grant_permissions method of the User class. Internally the chmod command is used, hence both symbolic and octal notation can be used. Note that only file access is controlled and ownership is determined by the user executing the script. The implemented permission allows other users to read the system log file when it is copied by root to /home.

Access to the files can be controlled by modifying the file and permission lists.

Tasks & Design choices

Main function

After printing the name of the script, a CronJob object is created. Its arguments correspond to the different fields in the crontab file. While the path to python3 is resolved in the constructor, the script name is contained in the variable __file__. The constructor also checks if the cronjob already exists in the crontab file and sets the active attribute accordingly. In case the cronjob is not set up yet, it is added to the crontab file by means of the add_cronjob method.

Next, a LogFile object is created with the name of the file and its location. Since the location where the system log file and the modified log files shall be stored are owned by the user, a User object with attribute home is created. Then the system log file is copied to the users home-directory by means of the copy_log method which changes the location of the log file. Next, the system log file is trimmed in the trim_log method. Finally, the access permissions of files are set such that all relevant users have access to them.

```
print("\n| ReadAble Logs |\n")

cronjob = CronJob("0","0","*","*","1-5","python3",f"{__file__}")
if (not cronjob.active):
    cronjob.add_cronjob()

log_file = LogFile("syslog","/var/log")
user = User()
log_file.copy_log(user.home)
log_file.trim_log()

file_list = [f"{user.home}/syslog"]
perm_list = ["o+r"]
user.grant_permissions(file_list,perm_list)
```

CronJob class

The CronJob class has seven "private" and one public instance attributes:

- *self._m (str)*: Value of minute field in crontab
- *self._h (str)*: Value of hour field in crontab
- *self._dom (str)*: Value of day-of-month field in crontab
- *self._mon (str)*: Value of month field in crontab
- *self._dow (str)*: Value of day-of-week field in crontab
- *self._exe (str)*: Executable/program called in crontab
- *self._file (str)*: File executed in crontab
- *self.active (bool)*: Cronjob entered in crontab (active) or not (inactive)

- **Task**: Check if the cronjob already exists in the crontab file.
    - **Implementation**: To check if the cronjob already exists in the crontab file, crontab -l is executed. In case no cronjob exists, the request returns a failure which raises an CalledProcessError exception. This is handled in a try-except statement.

```
def __init__(self,m,h,dom,mon,dow,exe,file):
    """Constructs all necessary attributes for the CronJob object."""

    self._m = m
    self._h = h
    self._dom = dom
    self._mon = mon
    self._dow = dow
    cmd = f"which {exe}"
    self._exe = subprocess.check_output(cmd,shell=True,text=True) \
                        .strip()
    self._file = file

    self.active = True
    cmd = "crontab -l"
    try:
        crontab = subprocess.check_output(cmd,shell=True,text=True, \
                                        stderr=subprocess.DEVNULL)
    except subprocess.CalledProcessError:
        # if no crontab for user
        crontab = ""

    # check if cronjob for __file__ is set
    if (self._file not in crontab):
        self.active = False
```

- **Task**: Run the script daily at 00:00 o'clock from Mon-Fri.
    - **Implementation**: Inform the user about the ensuing creation of a cronjob and wait for an input by the user. Add an exemplary cronjob to the crontab file, then display the crontab file of the user.

```
def add_cronjob(self):
    """Adds cronjob to crontab."""

    print("Create a cronjob to execute this script "+ \
            "daily on working days.\n")
    input("Press Enter to modify the exemplary cronjob "+ \
            "on the last line... ")

    # add cronjob to crontab
    job = f"{self._m}\t{self._h}\t"+ \
          f"{self._dom}\t{self._mon}\t{self._dow}\t"+ \
          f"{self._exe} {self._file}"
    cmd = f"(crontab -l; echo \"{job}\") | crontab -"
    subprocess.call(cmd,shell=True)
    # allow for modification of cronjob or approval
    cmd = "crontab -e"
    subprocess.call(cmd,shell=True)
```

    - **Q**: Should I create a cronjob or is there an alternative?
      A: Since the script is supposed to run on Linux servers, using a cronjob is straightforward.
    - **Q**: Should I automatically append the cronjob to the crontab file or let the user enter it?
      A: To allow for comfortable controls for the user while maintaining their agency, the cronjob is added to the crontab file and shown to the user to be approved or modified. Note that any modification of the crontab file happens after the user is forwarded to the crontab file, hence no backup of the crontab has to be created to account for a KeyboardInterrupt.

      There exists the crontab module in Python, however it is not part of the Python standard library and would introduce additional dependencies.

LogFile class

The LogFile class has two "private" instance attributes:

- *self._name (str)*: Name of the Log file
- *self._location (str)*: Path to the Log file

```
def __init__(self,name,location):
    """Constructs necessary attributes of the LogFile object."""

    self._name = name
    self._location = location
```

- **Task**: Copy the system log file into the calling users home-directory.
    - **Implementation**: Using a system call, the system log file is copied from its location to the destination. The location is then updated with the destination. root uses /home instead of their home-directory which is implemented in the constructor of the User class.

    ```
    def copy_log(self,destination):
        """
        Copies Log to destination.

            Parameters:
                destination (str): Absolute path
        """

        cmd = f"cp {self._location}/{self._name} {destination}"
        subprocess.call(cmd,shell=True)
        self._location = destination

    ...

    def __init__(self):
        """Constructs all necessary attributes for the User object."""

        cmd = "whoami"
        self._name = subprocess.check_output(cmd,shell=True,text=True) \
                                    .strip()

        if (self._name == "root"):
            # root user: logs in accessible directory
            self.home = "/home"
        else:
            # regular user: logs in own home-directory
            self.home = f"/home/{self._name}"
    ```

    - **Q**: Are there any obstacles when copying a file from a directory if both are owned by root?
      A: There are no obstacles since the other users have reading rights.
    - **Q**: Are there any obstacles when copying a file into a users home-directory or /home?
      A: There are no obstacles when a user copies a file into their home-directory. To copy files into /home, read them or write into /home requires root privilege.
    - **Q**: Is it more sensible to use multiple system calls or to externalize commands into a bash script and invoke it by a system call?
      A: In the present case, the commands are short enough to keep them as system calls.
    - **Q**: Why does the root user operate in /home and not in its own "home-directory" /root?
      A: The script introduces redundancies and information silos when executed by separate users: Each user has a copy of the same, possibly large, system log file and their own set of filters. Running the script with the central role of root using /root as home-directory does not combat these deficiencies. Therefore the behavior when root runs the script is modified to allow for centralized log files. The code can be further extended to modify the permissions of users to access and modify the entries of the log files.
- **Task**: Read the system log file.
    - **Implementation**: Read the system log file by means of a generator, one line at a time. Account for a missing file and a lack of permissions.

```
def _read_log(self):
    """Reads from Log file, one line at a time."""

    fname = f"{self._location}/{self._name}"
    try:
        with open(fname,"r") as logfile:
            for line in logfile:
                yield line
    except FileNotFoundError:
        msg = f"The file {fname} does not exist."
        print(msg)
    except PermissionError:
        msg = f"You lack permission to read {fname}."
        print(msg)
```

- **Q**: Does the format of the system log file require any specific handling (e.g. unpacking, decompression, file conversion)?
  A: The system log file is a text file which does not require specific handling.
- **Q**: Does the system log file fit into memory when reading it?
  A: This is unclear. A generator is used to read the system log file, one line at a time.

  The current implementation could be modified to read the file in batches. This alone would not significantly alter the performance without further modifications for the following reasons:

  - The generator does not close the file between consecutive reads. Reading one line at a time or batches via a generator opens and closes the file only once.
  - The filter operates on a single line, hence to improve performance when using batchwise reads would require to modify application of the filter. One option would be to move the filter into a separate function operating on a list which can then be executed concurrently using the threading module.

  To keep the code simple, one line at a time is read.
- **Q**: Would it be expedient to store the content of the system log file in a specific data structure?
  A: Since the size of the system log file is unclear, it is read line by line. With the size of the system log file unclear, it is also unclear how many lines matching the whitelist there are. Appending the lines approved by the whitelist to a list is therefore not sensible.

  Since the overall task is to filter the file and not to display it, modifying the representation of the data by means of the __*repr*__ and __*str*__ method are not necessary.

  Each line is kept as a string to use regular expressions for filtering.

- **Task**: Modify log entries, i.e. what to filter for and how to filter.

  - **Implementation**: Set up a whitelist with expressions to search for. Start the generator used to write the modified logs to file. While looping through the system log file, each line undergoes the following examination:
    - Check if an entry of the whitelist is contained in the line.
    - No entry of the whitelist is contained in the line: Skip to the next line.
    - An entry of the whitelist is contained in the line: Obtain the timestamp and the error message behind the whitelist entry, combine both into one strng separated by a hyphen, write the modified line to file.

    After the entire system log file is handled, close the write generator.

```
def trim_log(self):
    """
    Trims Log to improve readability.

    Modify whitelist to control filter.
    """

    # whitelist of filter
    whitelist = [" EMERGENCY:"," ALERT:"," CRITICAL:"," ERROR:",\
                 " WARNING:"," NOTICE:"," INFORMATIONAL:"," DEBUG:"]

    write_gen = self._write_log()
    write_gen.send(None)

    for line_log in self._read_log():
        # check if line contains whitelist entry
        is_allowed = [(wl_entry in line_log) for wl_entry in whitelist]
        if (any(is_allowed)):
            wl_entry = whitelist[is_allowed.index(True)]
        else:
            # omit lines without whitelist entry
            continue

        # keep timestamp, messages with whitelist entry
        time_pattern = r"^(.+?)\s"
        wl_pattern = r"("+wl_entry+r".+?)$"
        timestamp = re.search(time_pattern,line_log).group(1)
        wl_message = re.search(wl_pattern,line_log).group(1).strip()
        line_mod = " - ".join([timestamp,wl_message])

        write_gen.send(line_mod)

    # close generator
    write_gen.close()
```

- **Q**: Would it be expedient to convert the content of the system log file into a specific data structure to improve filtering?
  A: Each line is kept as a string to use regular expressions for filtering. Splitting each line into a list of words to filter it word by word would introduce two obstacles: Filtering for anything but words requires additional effort and the need to parse each list would harm the performance.
- **Q**: Which method should be used to filter the file: regular expressions, string methods or manual filtering (e.g. sliding window)?
  A: To filter efficiently for arbitrary patterns, regular expressions should be used. They are applied to each line of the system log file separately.
- **Q**: Should the original data be overwritten or should a new data structure with the modified data be generated?
  A: Since strings are immutable in Python, in place modifications are not possible.

- **Task**: Write the modified data of the system log file into a file within the home-directory labeled by the current date.
  - **Implementation**: Get the current date using the date method of the datetime module. Write the data to a file named after the current date using a generator, one line at a time. Account for a lack of permissions.

```
def _write_log(self):
    """Writes trimmed Log to file YYYY-MM-dd, one line at a time."""

    fname = f"{self._location}/"+date.today().strftime("%Y-%m-%d")
    try:
        with open(fname,"w") as logfile:
            while True:
                data = (yield)
                logfile.write(data+"\n")
    except PermissionError:
        msg = f"You lack permission to create {fname}."
        print(msg)
```

- **Q**: Are there any obstacles when writing into a file within a users home-directory or /home?
  A: There are no obstacles when a user writes into a file within their home-directory. To write into files within /home requires root privilege.
- **Q**: Why is a generator used to write out the data?
  A: Since it is unknown whether the system log file fits into memory, it is also unknown whether the lines matched by the whitelist would fit into memory. Storing them in a list is thus not sensible. Since the data to be written emerges one line at a time and a generator does not introduce additional openings and closings of files, a generator is used.

Overall, reading and writing one line at a time distributes the file I/O evenly over time and avoids bottlenecks where data has to be read or written to continue execution. Reading and writing in batches is the next logical step. However to improve performance, additional concurrent execution by means of the threading module needs to be added.

- **Q**: How do I get the system date into Python? Is there a module for that purpose or do I have to perform a system call to obtain it from the Linux date command?
  A: The date command in the module datetime provides the current date in Python.

User class

The User class has one "private" and one public instance attribute:

- *self._name (str)*: Name of the user
- *self.home (str)*: Home-directory of the user

```
def __init__(self):
    """Constructs all necessary attributes for the User object."""

    cmd = "whoami"
    self._name = subprocess.check_output(cmd,shell=True,text=True) \
                        .strip()

    if (self._name == "root"):
        # root user: logs in accessible directory
        self.home = "/home"
    else:
        # regular user: logs in own home-directory
        self.home = f"/home/{self._name}"
```

- **Task**: Set the access permissions of files such that all relevant users have access to them.
  - **Implementation**: For each file in the files list, the permissions specified in the permissions list are set using the chmod command.

    ```
    @staticmethod
    def grant_permissions(files,permissions):
        """Grant permissions for files."""

        num_files = len(files)
        if (num_files == len(permissions)):
            for ii in range(num_files):
                cmd = f"chmod {permissions[ii]} {files[ii]}"
                subprocess.call(cmd,shell=True)
        else:
            raise Assertion.Error("Number of files and "+ \
                                  "permissions do not match.")
    ```

  - **Q**: Why is this a static method?
    A: A static method is owned by the class which does not use any of its attributes or methods. It is used for utility functions for which it makes sense to associate them to a class. The grant_permissions function sets for a list of files the corresponding permissions, but does not need any attributes from the class. It thus checks these boxes.

Planned Features

- **Whitelist**: Since the severities are by default not contained in the system log file under ubuntu, the rsyslog templates have to be adapted or introduced to add them to the error messages. This should not be done automatically since it could break other applications, but the documentation should point this out and add a section how to add a template.
- **Concurrency**: Read and write the log file in batches and apply the filter concurrently.
- **Presentation**: Add an option to display the logs to allow more highlighting.
- **Command line**: Add command line arguments for the user to filter the log file similar to journalctl.