Санкт-Петербургский государственный университет

Направление ”Большие данные и распределенная цифровая платформа”

Лабораторная работа по дисциплине
Системное программирование в Linux

”Создание демона для регулярного резервного копирования данных.”

Выполнил:
Зайнуллин Мансур Альбертович
Группа: 23.Б16-пу

Руководитель:
Киямов Жасур Уткирович

Санкт-Петербург
2024

Оглавление

1 Цель работы

Цель данной работы создать демона для автоматического регулярного резервного копирования данных с одного каталога в другой.

2 Описание задачи

В данном проекте необходимо реализовать демона для автоматического регулярного резервного копирования данных. Основные задачи включают:

- **Создание демона:** Разработать системный процесс, который будет работать в фоновом режиме и выполнять резервное копирование данных. Демон должен быть интегрирован в систему таким образом, чтобы он запускался при старте операционной системы.

- **Настройка конфигурации:** Создать конфигурационный файл, в котором пользователь сможет указать исходный каталог, каталог для резервных копий и частоту резервного копирования. Это позволит пользователю гибко настраивать работу демона в соответствии с его потребностями.

- **Управление и мониторинг:** Обеспечить возможность управления демоном через командную строку. Реализовать команды для запуска (`start`), остановки (`stop`), перезапуска (`restart`) демона, а также для изменения настроек, таких как добавление (`add`) и удаление (`remove`) файлов из списка резервного копирования, установка интервала (`set_interval`) и изменение папки для резервных копий (`change_destination`).

- **Функциональность резервного копирования:** Демон должен уметь считывать конфигурацию из файла, ожидать наступления времени для следующей резервной копии, создавать резервные копии файлов из исходного каталога в каталог для резервных копий с добавлением временной метки, и журналировать выполнение операций в системный журнал.

- **Обеспечение безопасности:** Ограничить доступ к конфигурационным файлам и резервным копиям для защиты данных. Это важно для предотвращения несанкционированного доступа и обеспечения безопасности данных.

3 Теоретическая часть

3.1 Определение и роль демонов

Демон — это фоновый процесс, который выполняется в операционной системе и не требует взаимодействия с пользователем. Они запускаются при старте системы и продолжают работать до ее выключения или перезагрузки. Демоны выполняют важные системные задачи, такие как управление сетевыми соединениями, обслуживание веб-серверов и мониторинг системных ресурсов. Примеры демонов включают `cron` для планирования задач и `sshd` для управления удаленными соединениями.

3.2 Характеристики демонов

Основные характеристики демонов включают:

- **Фоновый режим:** Демоны работают в фоновом режиме, не имея интерфейса для взаимодействия с пользователем.
- **Автономность:** Они запускаются автоматически при старте системы и работают независимо от пользовательских процессов.
- **Управление через системные службы:** Демоны управляются через системные службы, такие как systemd в Linux.

3.3 Управление демонами в Ubuntu

В Ubuntu управление демонами осуществляется с помощью `systemd` — системы инициализации и управления службами. `Systemd` позволяет запускать, останавливать, перезапускать и проверять статус демонов с помощью команд `systemctl`. Для настройки автозапуска демонов используется команда `systemctl enable`.

3.4 Создание и настройка service-файлов

Для каждого демона создается service-файл, который описывает, как и когда должен запускаться демон. Эти файлы обычно находятся в директории `/etc/systemd/system/`. Service-файл содержит секции `[Unit]`, `[Service]`, и `[Install]`, которые определяют описание службы, параметры запуска и условия автозапуска соответственно.

Пример service-файла:

```
[Unit]
Description=My Custom Service
After=network.target

[Service]
Type=simple
ExecStart=/usr/bin/my_custom_service
Restart=on-failure

[Install]
WantedBy=multi-user.target
```

3.5 Мониторинг и логирование

`Systemd` включает в себя систему журналирования (`journald`), которая собирает и хранит логи служб. Логи демонов можно просматривать с помощью команды `journalctl -u <service>`, что позволяет отслеживать их работу и выявлять проблемы.

4 Описание программы

Программа представляет собой демона для автоматического регулярного резервного копирования данных. Она разработана для работы в фоновом режиме и обеспечивает надежное копирование данных из одного каталога в другой. Рассмотрим основные аспекты работы программы и ее функциональность.

4.1 Алгоритм работы программы

1. **Инициализация:** Программа начинает с инициализации необходимых директорий и файлов, таких как конфигурационные файлы, логи и контрольные суммы. Это обеспечивает сохранение состояния между запусками.

2. **Чтение конфигурации:** Функция `get_config_file(path)` загружает конфигурационный файл, который содержит параметры резервного копирования, включая исходный каталог, каталог для резервных копий и интервал резервного копирования. Если файл отсутствует или содержит некорректные данные, создается новый файл с настройками по умолчанию.

3. **Запуск демона:** Функция `start_daemon()` проверяет, не запущен ли уже демон, и, если нет, запускает новый процесс. Она также регистрирует функцию для удаления PID-файла при завершении работы демона.

4. **Основной цикл резервного копирования:** В функции `daemon_main()` реализован основной цикл работы демона. Этот цикл выполняется бесконечно с интервалом, указанным в конфигурации. В каждом цикле:

    - Загружается текущая конфигурация и список файлов для резервного копирования.

- Для каждого файла вычисляется контрольная сумма с помощью `calculate_checksum(file_path)`.
- Если контрольная сумма файла изменилась, файл копируется в каталог резервных копий с помощью `copy_file(source, destination)`.
- Обновленные контрольные суммы сохраняются в файл.

5. **Журналирование:** Программа ведет журнал своей работы, записывая информацию о выполненных операциях, ошибках и предупреждениях. Логи сохраняются в файл `/var/log/backupd/backupd.log`, что позволяет отслеживать работу демона и выявлять проблемы.

6. **Управление демоном:** Программа поддерживает управление через командную строку. Пользователь может запускать и останавливать демона, изменять настройки резервного копирования, просматривать список файлов для резервного копирования и логи, а также восстанавливать файлы из резервной копии.

7. **Завершение работы:** Функция `stop_daemon()` завершает работу демона, удаляя PID-файл и освобождая системные ресурсы. Это позволяет корректно завершить процесс и избежать конфликтов при последующих запусках.

4.2 Основные функции программы

- `get_config_file(path)`: Загружает и проверяет конфигурационный файл.
- `calculate_checksum(file_path)`: Вычисляет контрольную сумму MD5 для файла.
- `copy_file(source, destination)`: Копирует файл, сохраняя метаданные.
- `daemon_main()`: Выполняет основной цикл резервного копирования.

- `start_daemon()`: Запускает демона резервного копирования.
- `stop_daemon()`: Останавливает демона и освобождает ресурсы.

5 Листинг программы

```
#!/usr/bin/env python3

import argparse
import json
import os
import re
import time
import hashlib
import shutil
import sys
import atexit
import logging
from signal import signal, SIGTERM

# Constants for paths
# PATH_TO_CONFIG_JSON =
    ↪ os.path.join(os.path.dirname(os.path.abspath(__file__)),
    ↪ 'config.json')
# PATH_TO_CHECKSUMS_JSON =
    ↪ os.path.join(os.path.dirname(os.path.abspath(__file__)),
    ↪ 'checksums.json')

PATH_TO_CONFIG_JSON = '/etc/backupd/config.json'
PATH_TO_CHECKSUMS_JSON = '/var/lib/backupd/checksums.json'
LOG_FILE_PATH = '/var/log/backupd/backupd.log'
PID_FILE = '/var/lib/backupd/backupd.pid'

# Logging setup
logging.basicConfig(
    level=logging.INFO,
    format='%(asctime)s [%(levelname)s] %(message)s',
    handlers=[
        logging.FileHandler(LOG_FILE_PATH),
        logging.StreamHandler(sys.stdout)
    ]
)
```

```
def remove_escape_characters(path):
    """
    Removes escape characters (backslashes) from the path string.

    :param path: Path string with escape characters.
    :return: String without escape characters.
    """
    return path.replace('\\', '')


def is_valid_mac_path(path):
    """
    Checks the validity of a path in Mac OS.

    :param path: Path to check.
    :return: True if the path is valid, otherwise False.
    """
    path = remove_escape_characters(path)
    if not path.startswith('/'):
        return False
    if re.search(r'[\\:]', path):
        return False
    return True


def get_config_file(path):
    """
    Loads the configuration file and checks its contents.

    :param path: Path to the configuration file.
    :return: Contents of the configuration file.
    """
    try:
        with open(path, 'r+') as file:
            config = json.load(file)

            if 'interval' not in config or not
                ↪ isinstance(config['interval'], int) or config['interval']
                ↪ < 0:
```

```
            config['interval'] = 300
            logging.warning("Invalid 'interval' value in configuration.
                ↪ Default value of 300 set.")

        if 'backup_destination' not in config or not
            ↪ isinstance(config['backup_destination'], str) or not
            ↪ is_valid_mac_path(config['backup_destination']):
            config_folder = os.path.dirname(os.path.abspath(path))
            config['backup_destination'] = os.path.join(config_folder,
                ↪ 'backup/')
            logging.warning("Invalid 'backup_destination' value in
                ↪ configuration. Default path set.")

        if 'items_to_backup' not in config or not
            ↪ isinstance(config['items_to_backup'], list) or \
                not all(isinstance(item, str) and is_valid_mac_path(item)
                    ↪ for item in config['items_to_backup']):
            config['items_to_backup'] = []
            logging.warning("Invalid 'items_to_backup' value in
                ↪ configuration. Empty list set.")

        file.seek(0)
        json.dump(config, file, indent=4)
        file.truncate()

        logging.info(f"Configuration file '{path}' successfully loaded.")
        return config

except FileNotFoundError:
    logging.warning(f"File '{path}' not found. Creating a new file with
        ↪ default settings.")
    config = {
        "interval": 300,
        "backup_destination":
            ↪ os.path.join(os.path.dirname(os.path.abspath(path)),
            ↪ 'backup/'),
        "items_to_backup": []
    }
    with open(path, 'w+') as file:
        json.dump(config, file, indent=4)
```

```
            return config

        except json.JSONDecodeError:
            logging.error(f"File '{path}' contains invalid JSON data.")
            sys.exit(1)


def save_json_file(json_object, path_to_json_file):
    """
    Saves a JSON object to the specified file.

    :param json_object: Object to save.
    :param path_to_json_file: Path to the file.
    :return: None
    """
    try:
        with open(path_to_json_file, 'w', encoding='utf-8') as config_file:
            json.dump(json_object, config_file, ensure_ascii=False, indent=4)
        logging.info(f"Configuration successfully saved to
            ↪ '{path_to_json_file}'.")
    except Exception as e:
        logging.error(f"Error saving configuration: {e}")


def is_path_exist(path):
    """
    Checks the existence of a directory or file at the specified path.

    :param path: Path to check.
    :return: 'file' if file, 'directory' if directory, 'not exists' if not
        ↪ exists.
    """
    if os.path.exists(path):
        if os.path.isfile(path):
            return 'file'
        elif os.path.isdir(path):
            return 'directory'
    else:
        return 'not exists'
```

```
def not_included_in_other_directories(path, list_of_paths):
    """
    Checks if the path is not a subdirectory of another path in the list.

    :param path: Path to check.
    :param list_of_paths: List of paths.
    :return: True if the path is not a subdirectory of another path.
    """
    for other_path in list_of_paths:
        if path != other_path and os.path.commonpath([path, other_path]) ==
            ↪ other_path:
            return False
    return True


def get_files_from_directory(directory_path):
    """
    Recursively collects all files from a directory and its subdirectories.

    :param directory_path: Path to the directory.
    :return: List of files.
    """
    files_list = []
    for root, dirs, files in os.walk(directory_path):
        for file in files:
            files_list.append(os.path.join(root, file))
    return files_list


def get_filtered_files_list(list_of_paths):
    """
    Filters the list of paths, leaving only existing files and directories,
    excluding nested paths.

    :param list_of_paths: List of paths to filter.
    :return: Filtered list of files.
    """
    absolute_paths = [os.path.realpath(path) for path in list_of_paths]
```

```
    existing_paths = [(path, is_path_exist(path)) for path in
        ↪ absolute_paths if is_path_exist(path) != 'not exists']
    existing_paths_set = set([p[0] for p in existing_paths])
    pre_filtered_paths = []
    for path, path_type in existing_paths:
        if not_included_in_other_directories(path, existing_paths_set):
            pre_filtered_paths.append((path, path_type))
    final_paths = set()
    for path, path_type in pre_filtered_paths:
        if path_type == 'directory':
            final_paths.update(get_files_from_directory(path))
        elif path_type == 'file':
            final_paths.add(path)
    return list(final_paths)


def calculate_checksum(file_path):
    """
    Calculates the MD5 checksum for the specified file.

    :param file_path: Path to the file.
    :return: Checksum or None if the file is not found.
    """
    try:
        md5_hash = hashlib.md5()
        with open(file_path, "rb") as file:
            for chunk in iter(lambda: file.read(4096), b""):
                md5_hash.update(chunk)
        return md5_hash.hexdigest()
    except (FileNotFoundError, PermissionError) as e:
        logging.error(f"Error accessing file '{file_path}': {e}")
        return None
    except Exception as e:
        logging.error(f"Error calculating checksum for file '{file_path}':
            ↪ {e}")
        return None


def get_checksums_json(path):
    """
```

```
    Loads the checksums file.

    :param path: Path to the checksums file.
    :return: Contents of the checksums file.
    """
    try:
        with open(path, 'r+') as file:
            checksums = json.load(file)
        logging.info(f"Checksums file '{path}' successfully loaded.")
        return checksums
    except FileNotFoundError:
        logging.warning(f"Checksums file '{path}' not found. Creating a new
            ↪ file.")
        checksums = {}
        with open(path, 'w+') as file:
            json.dump(checksums, file, indent=4)
        return checksums
    except json.JSONDecodeError:
        logging.error(f"Checksums file '{path}' contains invalid JSON
            ↪ data.")
        sys.exit(1)


def copy_file(source, destination):
    """
    Copies a file from source to destination, preserving metadata.

    :param source: Path to the source file.
    :param destination: Path to the destination file.
    :return: None
    """
    try:
        if os.path.isdir(destination):
            raise IsADirectoryError(f"'{destination}' is a directory, not a
                ↪ file.")
        if os.path.exists(destination) and os.path.samefile(source,
            ↪ destination):
            raise ValueError("Source and target files are the same. Copying
                ↪ is not possible.")
        destination_folder = os.path.dirname(destination)
```

```
        if not os.path.exists(destination_folder):
            os.makedirs(destination_folder)
        shutil.copy2(source, destination)
        logging.info(f"File '{source}' successfully copied to
            ↪ '{destination}'.")
    except Exception as e:
        logging.error(f"Error copying file from '{source}' to
            ↪ '{destination}': {e}")


def daemon_main():
    """
    Main function of the daemon, performing backup.
    """
    while True:
        config = get_config_file(PATH_TO_CONFIG_JSON)
        list_of_file_paths =
            ↪ get_filtered_files_list(config['items_to_backup'])
        checksums = get_checksums_json(PATH_TO_CHECKSUMS_JSON)

        for file_path in list_of_file_paths:
            file_checksum = calculate_checksum(file_path)
            if file_checksum is None:
                continue

            backup_file_checksum = checksums.get(file_path, "0")

            if file_checksum != backup_file_checksum:
                checksums[file_path] = file_checksum
                backup_destination =
                    ↪ os.path.join(config['backup_destination'],
                    ↪ os.path.relpath(file_path, '/'))
                copy_file(file_path, backup_destination)

        save_json_file(checksums, PATH_TO_CHECKSUMS_JSON)
        logging.info("Backup cycle completed. Waiting for the next run...")
        time.sleep(config["interval"])


def start_daemon():
```

```
"""
Starts the backup daemon.
"""
# Create necessary directories if they do not exist
os.makedirs(os.path.dirname(LOG_FILE_PATH), exist_ok=True)
os.makedirs(os.path.dirname(PATH_TO_CHECKSUMS_JSON), exist_ok=True)

# pidfile = '/tmp/backupd.pid'
pidfile = PID_FILE

if os.path.exists(pidfile):
    with open(pidfile, 'r') as f:
        pid = f.read().strip()
        if pid:
            logging.error(f"Daemon already running with PID: {pid}")
            print(f"Daemon already running with PID: {pid}")
            sys.exit(1)

def remove_pidfile():
    os.remove(pidfile)
    logging.info("Daemon stopped and PID file removed.")

# try:
#     pid = os.fork()
#     if pid > 0:
#         sys.exit(0)
# except OSError as e:
#     logging.error(f"Fork error: {e}")
#     sys.exit(1)

# os.setsid()

# try:
#     pid = os.fork()
#     if pid > 0:
#         sys.exit(0)
# except OSError as e:
#     logging.error(f"Second fork error: {e}")
#     sys.exit(1)
```

```
    # sys.stdout.flush()
    # sys.stderr.flush()

    # with open('/dev/null', 'r') as f:
    #     os.dup2(f.fileno(), sys.stdin.fileno())

    # with open('/dev/null', 'a') as f:
    #     os.dup2(f.fileno(), sys.stdout.fileno())

    # with open('/dev/null', 'a') as f:
    #     os.dup2(f.fileno(), sys.stderr.fileno())

    with open(pidfile, 'w') as f:
        f.write(str(os.getpid()))

    atexit.register(remove_pidfile)
    signal(SIGTERM, lambda signum, frame: sys.exit(0))

    logging.info("Daemon successfully started.")
    daemon_main()


def stop_daemon():
    """
    Stops the backup daemon.
    """
    # pidfile = '/tmp/backupd.pid'
    pidfile = PID_FILE
    if os.path.exists(pidfile):
        with open(pidfile, 'r') as f:
            pid = int(f.read().strip())
            try:
                os.kill(pid, SIGTERM)
                logging.info(f"Daemon with PID {pid} successfully stopped.")
            except OSError as e:
                logging.error(f"Error stopping daemon with PID {pid}: {e}")
        os.remove(pidfile)
        logging.info(f"PID file {pidfile} removed.")
    else:
```

```
        logging.warning(f"PID file not found. Daemon may not have been
            ↪ running.")


def list_backup_items():
    """
    Displays the list of files and folders that are in the backup process.
    """
    config = get_config_file(PATH_TO_CONFIG_JSON)
    items = config.get('items_to_backup', [])

    if items:
        logging.info("List of files and folders for backup obtained.")
        print("List of files and folders for backup:")
        for item in items:
            print(f" - {item}")
    else:
        logging.info("List of files and folders for backup is empty.")
        print("List of files and folders for backup is empty.")


def add_backup_item(item_path):
    """
    Adds a file or folder to the backup list.

    :param item_path: Path to the file or folder.
    """
    absolute_path = os.path.realpath(item_path)
    absolute_path = remove_escape_characters(absolute_path)

    if not is_valid_mac_path(absolute_path):
        logging.error(f"Error: Path '{item_path}' is invalid.")
        print(f"Error: Path '{item_path}' is invalid.")
        return

    path_type = is_path_exist(absolute_path)
    if path_type == 'not exists':
        logging.error(f"Error: Path '{absolute_path}' does not exist.")
        print(f"Error: Path '{absolute_path}' does not exist.")
        return
```

```
    config = get_config_file(PATH_TO_CONFIG_JSON)
    items_to_backup = config.get('items_to_backup', [])

    if absolute_path in items_to_backup:
        logging.info(f"Path '{absolute_path}' is already in the backup
            ↪ list.")
        print(f"Path '{absolute_path}' is already in the backup list.")
        return

    items_to_backup.append(absolute_path)
    config['items_to_backup'] = items_to_backup
    save_json_file(config, PATH_TO_CONFIG_JSON)

    logging.info(f"Path '{absolute_path}' successfully added to the backup
        ↪ list.")
    print(f"Path '{absolute_path}' successfully added to the backup list.")


def remove_backup_item(item_path):
    """
    Removes a file or folder from the backup list.

    :param item_path: Path to the file or folder.
    """
    absolute_path = os.path.realpath(item_path)
    absolute_path = remove_escape_characters(absolute_path)

    if not is_valid_mac_path(absolute_path):
        logging.error(f"Error: Path '{item_path}' is invalid.")
        print(f"Error: Path '{item_path}' is invalid.")
        return

    path_type = is_path_exist(absolute_path)
    if path_type == 'not exists':
        logging.error(f"Error: Path '{absolute_path}' does not exist.")
        print(f"Error: Path '{absolute_path}' does not exist.")
        return

    config = get_config_file(PATH_TO_CONFIG_JSON)
```

```
    items_to_backup = config.get('items_to_backup', [])

    if absolute_path in items_to_backup:
        items_to_backup.remove(absolute_path)
        config['items_to_backup'] = items_to_backup
        save_json_file(config, PATH_TO_CONFIG_JSON)
        logging.info(f"Path '{absolute_path}' successfully removed from the
            ↪ backup list.")
        print(f"Path '{absolute_path}' successfully removed from the backup
            ↪ list.")
    else:
        logging.warning(f"Path '{absolute_path}' not found in the backup
            ↪ list.")
        print(f"Path '{absolute_path}' not found in the backup list.")


def update_sleep_interval(interval):
    """
    Changes the interval time between backups.

    :param interval: Time in seconds.
    """
    if not isinstance(interval, int) or interval <= 0:
        logging.error(f"Error: Interval '{interval}' must be a positive
            ↪ integer.")
        print(f"Error: Interval '{interval}' must be a positive integer.")
        return

    config = get_config_file(PATH_TO_CONFIG_JSON)
    config['interval'] = interval
    save_json_file(config, PATH_TO_CONFIG_JSON)

    logging.info(f"Backup interval successfully changed to {interval}
        ↪ seconds.")
    print(f"Backup interval successfully changed to {interval} seconds.")


def change_backup_destination(new_path):
    """
    Changes the folder for storing backups, moves files to the new folder,
```

```
deletes the old folder, and clears the checksums file.

:param new_path: Path to the new folder.
"""
absolute_new_path = os.path.realpath(new_path)
absolute_new_path = remove_escape_characters(absolute_new_path)

if not is_valid_mac_path(absolute_new_path):
    logging.error(f"Error: Path '{new_path}' is invalid.")
    print(f"Error: Path '{new_path}' is invalid.")
    return

if is_path_exist(absolute_new_path) != 'directory':
    try:
        os.makedirs(absolute_new_path)
        logging.info(f"New backup directory created:
            ↪ '{absolute_new_path}'.")
    except OSError as e:
        logging.error(f"Error creating directory '{absolute_new_path}':
            ↪ {e}")
        print(f"Error creating directory '{absolute_new_path}': {e}")
        return

config = get_config_file(PATH_TO_CONFIG_JSON)
old_backup_folder = config.get('backup_destination', '')

if is_path_exist(old_backup_folder) != 'directory':
    logging.error(f"Error: Old backup folder '{old_backup_folder}' does
        ↪ not exist.")
    print(f"Error: Old backup folder '{old_backup_folder}' does not
        ↪ exist.")
    return

try:
    for root, dirs, files in os.walk(old_backup_folder):
        relative_path = os.path.relpath(root, old_backup_folder)
        destination_dir = os.path.join(absolute_new_path, relative_path)

        if not os.path.exists(destination_dir):
            os.makedirs(destination_dir)
```

```
    for file in files:
        source_file = os.path.join(root, file)
        destination_file = os.path.join(destination_dir, file)
        try:
            shutil.move(source_file, destination_file)
            logging.info(f"File '{file}' moved to
                ↪ '{destination_file}'.")
        except Exception as e:
            logging.error(f"Error moving file '{source_file}': {e}")

shutil.rmtree(old_backup_folder)
logging.info(f"Old backup folder '{old_backup_folder}' successfully
    ↪ deleted.")
print(f"Old backup folder '{old_backup_folder}' successfully
    ↪ deleted.")

if os.path.exists(PATH_TO_CHECKSUMS_JSON):
    try:
        with open(PATH_TO_CHECKSUMS_JSON, 'w') as checksums_file:
            json.dump({}, checksums_file, indent=4)
        logging.info(f"Checksums file '{PATH_TO_CHECKSUMS_JSON}'
            ↪ successfully cleared.")
        print(f"Checksums file '{PATH_TO_CHECKSUMS_JSON}'
            ↪ successfully cleared.")
    except Exception as e:
        logging.error(f"Error clearing checksums file
            ↪ '{PATH_TO_CHECKSUMS_JSON}': {e}")
else:
    logging.warning(f"Checksums file '{PATH_TO_CHECKSUMS_JSON}' not
        ↪ found for clearing.")
    print(f"Checksums file '{PATH_TO_CHECKSUMS_JSON}' not found for
        ↪ clearing.")

config['backup_destination'] = absolute_new_path
save_json_file(config, PATH_TO_CONFIG_JSON)
logging.info(f"Backup folder successfully changed to
    ↪ '{absolute_new_path}'.")
print(f"Backup folder successfully changed to
    ↪ '{absolute_new_path}'.")
```

```
    except Exception as e:
        logging.error(f"Error moving files or changing backup folder: {e}")
        print(f"Error moving files or changing backup folder: {e}")


def clear_backup_folder():
    """
    Completely clears the folder where all backups are copied and resets
        ↪ the checksums file.
    """
    config = get_config_file(PATH_TO_CONFIG_JSON)
    backup_folder = config.get('backup_destination', '')

    if is_path_exist(backup_folder) != 'directory':
        logging.error(f"Error: Backup folder '{backup_folder}' does not
            ↪ exist or is not a directory.")
        print(f"Error: Backup folder '{backup_folder}' does not exist or is
            ↪ not a directory.")
        return

    try:
        for root, dirs, files in os.walk(backup_folder):
            for file in files:
                try:
                    os.remove(os.path.join(root, file))
                except Exception as e:
                    logging.error(f"Error deleting file '{file}': {e}")
            for dir in dirs:
                try:
                    shutil.rmtree(os.path.join(root, dir))
                except Exception as e:
                    logging.error(f"Error deleting directory '{dir}': {e}")

        logging.info(f"Backup folder '{backup_folder}' successfully
            ↪ cleared.")
        print(f"Backup folder '{backup_folder}' successfully cleared.")

        if os.path.exists(PATH_TO_CHECKSUMS_JSON):
            try:
```

```
                with open(PATH_TO_CHECKSUMS_JSON, 'w') as checksums_file:
                    json.dump({}, checksums_file, indent=4)
                logging.info(f"Checksums file '{PATH_TO_CHECKSUMS_JSON}'
                    ↪ successfully cleared.")
                print(f"Checksums file '{PATH_TO_CHECKSUMS_JSON}'
                    ↪ successfully cleared.")
            except Exception as e:
                logging.error(f"Error clearing checksums file
                    ↪ '{PATH_TO_CHECKSUMS_JSON}': {e}")
        else:
            logging.warning(f"Checksums file '{PATH_TO_CHECKSUMS_JSON}' not
                ↪ found for clearing.")
            print(f"Checksums file '{PATH_TO_CHECKSUMS_JSON}' not found for
                ↪ clearing.")

    except Exception as e:
        logging.error(f"Error clearing folder or checksums file: {e}")
        print(f"Error clearing folder or checksums file: {e}")


def paste_backup(target_dir):
    """
    Pastes files from the backup into the specified folder.

    :param target_dir: Path to the folder for pasting.
    :return: None
    """
    absolute_target_dir = os.path.realpath(target_dir)
    absolute_target_dir = remove_escape_characters(absolute_target_dir)

    if not is_valid_mac_path(absolute_target_dir):
        logging.error(f"Error: Path '{target_dir}' is invalid.")
        print(f"Error: Path '{target_dir}' is invalid.")
        return

    if is_path_exist(absolute_target_dir) != 'directory':
        logging.error(f"Error: Path '{absolute_target_dir}' does not exist
            ↪ or is not a directory.")
        print(f"Error: Path '{absolute_target_dir}' does not exist or is
            ↪ not a directory.")
```

```
        return

    config = get_config_file(PATH_TO_CONFIG_JSON)
    backup_folder = config.get('backup_destination', '')

    if is_path_exist(backup_folder) != 'directory':
        logging.error(f"Error: Backup folder '{backup_folder}' does not
            ↪ exist.")
        print(f"Error: Backup folder '{backup_folder}' does not exist.")
        return

    try:
        for root, dirs, files in os.walk(backup_folder):
            relative_path = os.path.relpath(root, backup_folder)
            destination_dir = os.path.join(absolute_target_dir,
                ↪ relative_path)

            if not os.path.exists(destination_dir):
                os.makedirs(destination_dir)

            for file in files:
                source_file = os.path.join(root, file)
                destination_file = os.path.join(destination_dir, file)
                shutil.copy2(source_file, destination_file)

        logging.info(f"Files successfully restored from backup to folder
            ↪ '{absolute_target_dir}'.")
        print(f"Files successfully restored from backup to folder
            ↪ '{absolute_target_dir}'.")
    except Exception as e:
        logging.error(f"Error restoring files: {e}")
        print(f"Error restoring files: {e}")


def show_logs():
    """
    Displays the last lines from the log file.
    """
    # log_file = os.path.join(os.path.dirname(os.path.abspath(__file__)),
        ↪ 'backupd.log')
```

```
    log_file = LOG_FILE_PATH

    try:
        with open(log_file, 'r') as f:
            lines = f.readlines()[-50:]
            for line in lines:
                print(line, end="")
    except FileNotFoundError:
        logging.warning(f"Log file '{log_file}' not found.")
        print(f"Log file '{log_file}' not found.")
    except Exception as e:
        logging.error(f"Error reading log file: {e}")
        print(f"Error reading log file: {e}")


def restart():
    """
    Restarts the backup daemon.
    First stops the current instance, then starts it again.
    """
    try:
        logging.info("Attempting to restart daemon...")
        print("Restarting daemon...")

        stop_daemon()
        start_daemon()

        logging.info("Daemon successfully restarted.")
        print("Daemon successfully restarted.")

    except Exception as e:
        logging.error(f"Error restarting daemon: {e}")
        print(f"Error restarting daemon: {e}")


def main():
    """
    Main function for command line processing and executing corresponding
        ↪ actions.
    """
```

```
parser = argparse.ArgumentParser(description='Utility for managing the
    ↪ backup daemon')

subparser = parser.add_subparsers(dest='command', help='Commands for
    ↪ managing the daemon')

parser_start = subparser.add_parser('start', help='Starts the backup
    ↪ daemon')

parser_stop = subparser.add_parser('stop', help='Stops the backup
    ↪ daemon')

parser_list = subparser.add_parser('list', help='Shows the list of
    ↪ files and folders for backup')

parser_add = subparser.add_parser('add', help='Adds a file or folder
    ↪ for backup')
parser_add.add_argument('path_to_file', type=str, help='Path to the
    ↪ file or folder to add')

parser_remove = subparser.add_parser('remove', help='Removes a file or
    ↪ folder from the backup list')
parser_remove.add_argument('path_to_file', type=str, help='Path to the
    ↪ file or folder to remove')

parser_set_interval = subparser.add_parser('set_interval', help='Sets
    ↪ the backup interval')
parser_set_interval.add_argument('interval', type=int, help='Interval
    ↪ in seconds between backups.')

parser_change_destination = subparser.add_parser('change_destination',
    ↪ help='Changes the folder for backups')
parser_change_destination.add_argument('path_to_folder', type=str,
    ↪ help='Path to the new folder for backups')

parser_clear_destination = subparser.add_parser('clear_destination',
    ↪ help='Completely clears the backup folder')

parser_paste = subparser.add_parser('paste', help='Pastes files from
    ↪ the backup into the specified folder')
```

```
    parser_paste.add_argument('target_dir', type=str, default=None,
        ↪ help='Path to the folder for pasting')

    parser_logs = subparser.add_parser('logs', help='Displays the daemon
        ↪ logs')

    parser_restart = subparser.add_parser('restart', help='Restarts the
        ↪ backup daemon')

    args = parser.parse_args()

    if args.command == 'start':
        start_daemon()
    elif args.command == 'stop':
        stop_daemon()
    elif args.command == 'list':
        list_backup_items()
    elif args.command == 'add':
        add_backup_item(args.path_to_file)
    elif args.command == 'remove':
        remove_backup_item(args.path_to_file)
    elif args.command == 'set_interval':
        update_sleep_interval(args.interval)
    elif args.command == 'change_destination':
        change_backup_destination(args.path_to_folder)
    elif args.command == 'clear_destination':
        clear_backup_folder()
    elif args.command == 'paste':
        paste_backup(args.target_dir)
    elif args.command == 'logs':
        show_logs()
    elif args.command == 'restart':
        restart()


if __name__ == "__main__":
    main()
```

6 Рекомендации пользователю

6.1 Инструкция по установке демона на Ubuntu 22

Эта инструкция поможет вам установить и настроить демон для автоматического резервного копирования данных на Ubuntu 22. Следуйте шагам ниже, чтобы успешно установить и запустить демон.

1. **Установка Python 3.9**:

   - Проверьте, установлен ли Python, и его версию:

     ```
     python3 --version
     ```

   - Если Python 3.9 не установлен, выполните следующие команды:

     ```
     sudo apt update
     sudo apt install python3.9
     ```

   - Установите pip, если он не установлен:

     ```
     sudo apt install python3-pip
     ```

2. **Установка Git**:

   - Проверьте, установлен ли Git:

     ```
     git --version
     ```

   - Если Git не установлен, выполните команду:

     ```
     sudo apt install git
     ```

3. **Клонирование репозитория проекта**:

   - Перейдите в домашнюю директорию:

     ```
     cd ~
     ```

   - Клонируйте репозиторий проекта:

```
git clone https://github.com/MansurYa/backup-demon.git
```

4. **Переход в папку проекта**:

   - Перейдите в директорию проекта:

```
cd backup-demon
```

5. **Установка файлов и настройка прав доступа**:

   - Создайте необходимые директории:

```
sudo mkdir -p /etc/backupd
sudo mkdir -p /var/lib/backupd
sudo mkdir -p /var/log/backupd
sudo mkdir -p /opt/backupd
```

   - Скопируйте файлы `backupd.py` и `config.json`:

```
sudo cp main.py /opt/backupd/backupd.py
sudo cp config.json /etc/backupd/config.json
```

   - Создайте системного пользователя `backupd`:

```
sudo useradd --system --no-create-home --shell
   ↪ /usr/sbin/nologin backupd
```

   - Установите правильные права доступа:

```
sudo chown backupd:backupd /opt/backupd/backupd.py
sudo chmod 750 /opt/backupd/backupd.py
sudo chown backupd:backupd /etc/backupd/config.json
sudo chmod 640 /etc/backupd/config.json
sudo chown -R backupd:backupd /var/lib/backupd
sudo chmod -R 750 /var/lib/backupd
sudo chown -R backupd:backupd /var/log/backupd
sudo chmod -R 750 /var/log/backupd
```

6. **Создание службы systemd**:

- Создайте файл службы `/etc/systemd/system/backupd.service`:

```
sudo nano /etc/systemd/system/backupd.service
```

- Вставьте в него следующий контент:

```
[Unit]
Description=Backup Daemon
After=network.target

[Service]
Type=simple
User=backupd
Group=backupd
ExecStart=/opt/backupd/backupd.py start
Restart=on-failure

[Install]
WantedBy=multi-user.target
```

- Перезагрузите конфигурацию systemd и запустите службу:

```
sudo systemctl daemon-reload
sudo systemctl enable backupd.service
sudo systemctl start backupd.service
```

- Проверьте статус службы:

```
sudo systemctl status backupd.service
```

7. **Создание команды `backupd`**:

- Сделайте скрипт `backupd.py` исполняемым:

```
sudo chmod +x /opt/backupd/backupd.py
```

- Создайте символическую ссылку для команды `backupd`:

```
sudo ln -s /opt/backupd/backupd.py /usr/local/bin/backupd
```

- Теперь вы можете вызывать команду `backupd` из любой директории.

6.2 Инструкция по эксплуатации демона

Теперь, когда демон установлен, вы можете управлять им с помощью командной строки. Вот основные команды для эксплуатации демона:

1. **Запуск демона**:
    - Чтобы запустить демон, используйте команду:

```
backupd start
```

2. **Остановка демона**:
    - Чтобы остановить демон, используйте команду:

```
backupd stop
```

3. **Просмотр списка файлов для резервного копирования**:
    - Чтобы увидеть список файлов и папок, которые находятся в процессе резервного копирования, используйте:

```
backupd list
```

4. **Добавление файла или папки для резервного копирования**:
    - Чтобы добавить файл или папку в список резервного копирования, используйте:

```
backupd add /path/to/your/file_or_directory
```

5. **Удаление файла или папки из списка резервного копирования**:
    - Чтобы удалить файл или папку из списка резервного копирования, используйте:

```
backupd remove /path/to/your/file_or_directory
```

6. **Установка интервала резервного копирования**:

- Чтобы установить интервал времени (в секундах) между резервными копиями, используйте:

```
backupd set_interval 600
```

7. **Изменение папки для резервных копий**:

   - Чтобы изменить папку для хранения резервных копий, используйте:

```
backupd change_destination /new/backup/directory
```

8. **Очистка папки для резервных копий**:

   - Чтобы полностью очистить папку, в которую копируются все резервные копии, используйте:

```
backupd clear_destination
```

9. **Восстановление файлов из резервной копии**:

   - Чтобы вставить файлы из резервной копии в указанную папку, используйте:

```
backupd paste /path/to/restore/directory
```

10. **Просмотр логов работы демона**:

    - Чтобы вывести последние строки из файла логов, используйте:

```
backupd logs
```

11. **Перезапуск демона**:

    - Чтобы перезапустить демон, используйте:

```
backupd restart
```

7 Контрольный пример

В этом разделе я продемонстрирую процесс установки и проверки работы демона резервного копирования данных на новой установке Ubuntu 22.04. В качестве объекта для резервного копирования будет использоваться каталог с тестовым файлом.

7.1 Установка и настройка Ubuntu 22.04

1. **Установка Ubuntu 22.04**: Я установил чистую версию Ubuntu 22.04 на сервер, предоставленный хостинг-провайдером. После входа в систему с помощью SSH были выполнены все шаги по установке демона, описанные в инструкции.

2. **Подготовка системы**: Я проверил, что на сервере установлены необходимые версии Python и Git для работы с репозиторием проекта.

```
root@server:~# python3 --version
Python 3.9.2
root@server:~# git --version
git version 2.25.1
```

3. **Клонирование репозитория и установка демона**: Я клонировал репозиторий проекта и выполнил шаги установки:

```
root@server:~# git clone
    ↪ https://github.com/MansurYa/backup-demon.git
Cloning into 'backup-demon'...
root@server:~# cd backup-demon
```

4. **Настройка демона**: Создал необходимые директории, установил права доступа, и настроил файл службы:

```
root@server:~/backup-demon# sudo mkdir -p /etc/backupd
    ↪ /var/lib/backupd /var/log/backupd /opt/backupd
root@server:~/backup-demon# sudo cp main.py
    ↪ /opt/backupd/backupd.py
```

```
root@server:~/backup-demon# sudo cp config.json
    ↪ /etc/backupd/config.json
root@server:~/backup-demon# sudo useradd --system
    ↪ --no-create-home --shell /usr/sbin/nologin backupd
root@server:~/backup-demon# sudo chown backupd:backupd
    ↪ /opt/backupd/backupd.py
root@server:~/backup-demon# sudo chmod 750 /opt/backupd/backupd.py
```

5. **Конфигурация и запуск службы**: Я настроил файл службы Systemd и запустил демон, чтобы убедиться, что он работает правильно.

```
root@server:~# sudo systemctl enable backupd.service
root@server:~# sudo systemctl start backupd.service
root@server:~# sudo systemctl status backupd.service
 backupd.service - Backup Daemon
      Loaded: loaded (/etc/systemd/system/backupd.service; enabled)
      Active: active (running) since ...
      Main PID: 2128 (python3)
```

Вывод показывает, что демон успешно запущен и работает в фоновом режиме.

7.2 Создание тестовой директории и проверка резервного копирования

1. **Создание тестовой директории и файла**: В корневом каталоге была создана директория `test` с тестовым файлом `test.txt`, содержащим строку ”texttest0”.

```
root@server:~# mkdir ~/test
root@server:~# echo "texttest0" > ~/test/test.txt
root@server:~# cat ~/test/test.txt
texttest0
```

2. **Добавление директории для резервного копирования**: Я добавил директорию `test` в список резервных копий и проверил, что она была успешно добавлена.

```
root@server:~# backupd add ~/test
 '/root/test'                                .
root@server:~# backupd list
     :
- /root/test
```

3. **Установка интервала резервного копирования**: Я установил интервал резервного копирования в 10 секунд, чтобы ускорить процесс тестирования.

```
root@server:~# backupd set_interval 10
     10     .
```

4. **Просмотр логов**: После запуска демона и установки интервала я проверил логи, чтобы убедиться в правильности работы демона.

```
root@server:~# backupd logs
[INFO]              .
[INFO]               '/etc/backupd/config.json'          .
[INFO]    '/root/test'                              .
[INFO]                                  10     .
```

5. **Восстановление данных из резервной копии**: Для проверки восстановления данных я создал папку `restore_dir` и воспользовался командой `paste` для копирования данных из резервной копии в нее.

```
root@server:~# mkdir ./restore_dir
root@server:~# backupd paste
   ↪ ./restore_dir
       '/root/restore_dir'.
```

Затем убедился, что данные успешно восстановлены:

```
root@server:~# ls restore_dir/test
test.txt
root@server:~# cat restore_dir/test/test.txt
texttest0
```

8 Вывод по работе

В ходе выполнения работы был успешно реализован демон для автоматического регулярного резервного копирования данных, который обеспечивает следующие ключевые функции:

1. **Автоматическое резервное копирование**: Демон регулярно выполняет резервное копирование данных с заданным интервалом, копируя файлы в указанный каталог.

2. **Гибкость настройки**: Пользователь может настраивать исходные каталоги для резервного копирования, интервал времени между копиями, а также управлять процессом через простые команды.

3. **Надежность**: Демон использует контрольные суммы для проверки изменений в файлах, что гарантирует, что только измененные файлы будут скопированы, а не все данные целиком.

4. **Логирование и мониторинг**: Работа демона журналируется, что позволяет пользователю отслеживать успешные операции, а также выявлять возможные ошибки.

5. **Управление через командную строку**: Весь процесс резервного копирования управляется через удобные команды, включая добавление файлов, настройку интервала, просмотр логов и восстановление данных.

Процесс установки и настройки был успешным, и демон функционирует корректно на сервере с Ubuntu 22.04, что подтверждается результатами тестирования: данные были успешно скопированы и восстановлены из резервной копии.