VPS Mirror Project Implementation

CSCI 310/593

Fall 2024

Dr. Adriano Cavalcanti

Implemented by Group 13

Zachery Groenewold

Samuel Arne

Dinesh Seveti

OBJECTIVE

To create a VPS mirror synchronized with a main server, design the main server to operate with at least four threads, managing shared data within a critical session. The mirror server should seamlessly track events, data, and processes on the main server, utilizing Python for efficient threading, automation, and synchronization.

PROCEDURE
The main server will run 4 concurrent threads to handle multiple tasks. These threads will work together using shared data, which can be accessed and modified by multiple threads. Mirror server will synchronize with the main server in real-time, ensuring data and processes are replicated accurately.

Before proceeding, ensure rsync is installed on both the main and secondary servers.

```
sudo apt-get update
sudo apt-get install rsync
```

```
sudo apt update
sudo apt install python3 python3-pip -y
```

1.Having a main server running at least 4 threads

Main Server

Functionality:

- Locking: Acquires the lock to ensure that only one process can write to the log file at a time.
- Simulating Delay: Adds a small sleep (time.sleep(1)) before writing to the log file to simulate some processing time.
- Writing to the File: Appends task data to the output/log.txt file.
- Error Handling: Catches any exceptions that might occur during the file-writing process and prints an error message.
- Releasing the Lock: Releases the lock after writing is done to allow other processes to write.
- The main block runs the `execute_tasks()` function 10 times in a loop,simulating 10 rounds of task execution. Each round executes multiple tasks concurrently

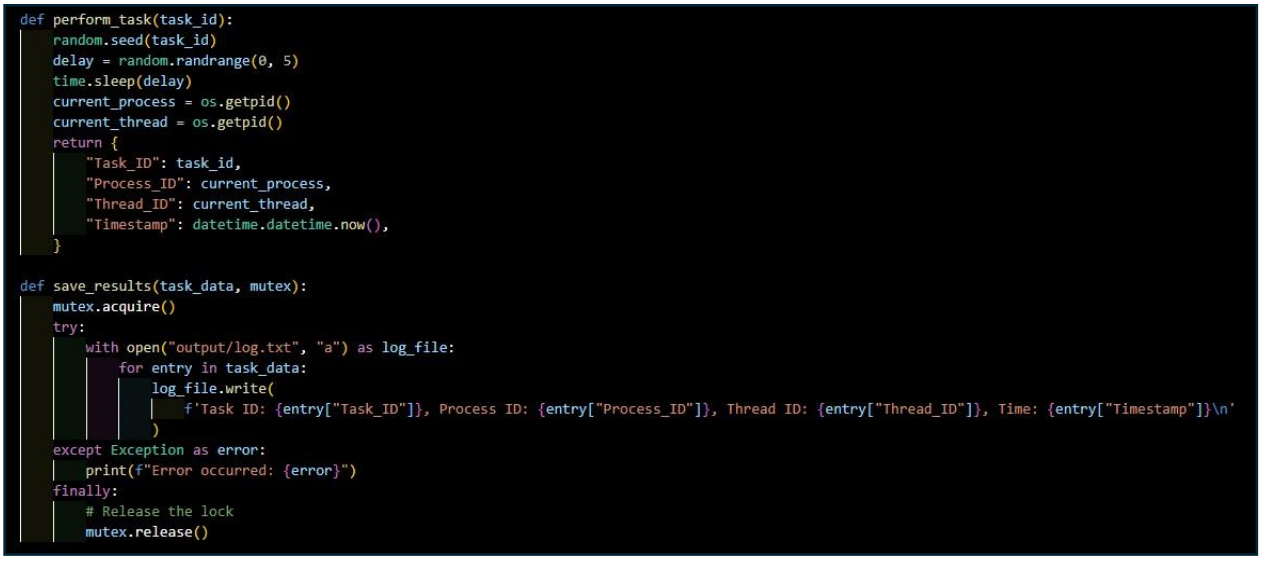
Libraries Used

```
import os
import datetime
import time
import random
from multiprocessing import Pool, Manager
```

- OS: Provides methods to interact with the operating system (e.g., getting process IDs).
- datetime: Used to get the current date and time when a task is completed.
- time: Used for simulating delays (e.g., time.sleep()).
- random: Used for generating random sleep times to simulate variability in task duration.

2. Threads should share a critical session with shared data.

Multiprocessing: Provides tools for concurrent execution, including Pool for parallel execution, Manager for shared resources, and Lock for synchronization.



```
def perform_task(task_id):
    random.seed(task_id)
    delay = random.randrange(0, 5)
    time.sleep(delay)
    current_process = os.getpid()
    current_thread = os.getpid()
    return {
        "Task_ID": task_id,
        "Process_ID": current_process,
        "Thread_ID": current_thread,
        "Timestamp": datetime.datetime.now(),
    }

def save_results(task_data, mutex):
    mutex.acquire()
    try:
        with open("output/log.txt", "a") as log_file:
            for entry in task_data:
                log_file.write(
                    f'Task ID: {entry["Task_ID"]}, Process ID: {entry["Process_ID"]}, Thread ID: {entry["Thread_ID"]}, Time: {entry["Timestamp"]}\n'
                )
    except Exception as error:
        print(f"Error occurred: {error}")
    finally:
        # Release the lock
        mutex.release()
```

- **task(index)**: This function simulates a task that is executed in parallel by multiple threads. It generates a random sleep time (between 0 and 5 seconds) based on the task index to ensure that each task runs independently. The function returns a dictionary containing the thread ID, process ID, task index, and the current timestamp, which is useful for tracking the execution of tasks.

- **update_results(data, lock)**: This function handles writing the results of the tasks to a file (output/log.txt) It accepts two arguments: data, which contains the task results, and lock, a threading lock that ensures only one thread writes to the file at a time, preventing race conditions. The function acquires the lock, writes each task result to the file, and then releases the lock to allow other threads to access the file.

- **main()**: The main function orchestrates the execution of the tasks. It uses Python's Manager() context manager to create a shared resource (the lock) and a Pool of 4 threads to process the tasks concurrently. The task function is mapped across the pool of threads, and the results are passed to the update_results function to be written to a file.

- The combination of these functions allows for efficient, concurrent task processing, while ensuring thread safety when writing results to the disk.

3. The Mirror Server should keep up with the events, data and processes running on the main server.

Secondary Server:
rsync is used to copy files from the primary server to the secondary server. A basic rsync command looks like this:

```
rsync -e "ssh -i /home/zekery360/.ssh/authorized_keys" -avz /home/zekery360/
zekery360@34.44.219.93:/home/zekery360/backup
```

Functionality:
Synchronizes data from the main server to the mirror server using rsync. A cron job is configured to automate this process periodically.

Automating Synchronization with cron
To automate the synchronization, you can schedule the **rsync** command to run at regular intervals using a **cron** job on the secondary server.

Open the cron editor:

```
crontab -e
```

Add a line to schedule the rsync job. For example:

```
*/10 * * * * rsync -avz --delete user@primary-server-ip:/path/to/source/ /path/to/destination/ >>
/var/log/rsync.log 2>&1
```

This runs the synchronization every 10 minutes. Output and errors are logged to /var/log/rsync.log. Save and exit the editor. The job will now run automatically at the specified interval.

Run the VPS_Server.py python script on the main server in a persistent manner using a process manager called system. Ensure the rsync command works and the mirror server successfully replicates the main server's data.