Task:

Please implement a program that synchronizes two folders: source and replica.

Main Task:
The program should maintain a full, identical copy of source folder at replica folder.

Condition A: Solve the test task by writing a program in one of these programming languages:
C, C++, C#, Python

Condition B: Synchronization must be one-way: after the synchronization content of the replica folder should be modified to exactly match content of the source folder;

Condition C: Synchronization should be performed periodically.

Condtion D: File creation/copying/removal operations should be logged to a file and to the console output;

Condition E: Folder paths, synchronization interval and log file path should be provided using the command line arguments;

Condition F: It is undesirable to use third-party libraries that implement folder synchronization;

Condtion G: It is allowed (and recommended) to use external libraries implementing other well-known algorithms. For example, there is no point in implementing yet another function that calculates MD5 if you need it for the task – it is perfectly acceptable to use a third-party (or built-in) library.

This is my solution in Python

Analysis:

1. Parse Command Line Arguments ( Condition E) :
    - Source folder path
    - Replica folder path
    - Synchronization interval ( in seconds)
    - Log file path
2. Logging Mechanism ( Condtion D) :
    - I will log the operations (file creation, deletion, copying) to both the console and a specified log file.
3. Synchronization Logic ( Condition B, Condtion G):
    - One-way synchronization: Ensure that any new, modified, or deleted files in the source folder are reflected in the replica folder
    - I will use file hashes (MD5) to determine if a file has changed.
4. File Operations ( Main Task) :
    - Copy files: Copy files from the source to the replica if they are different.
    - Remove files: Delete files in the replica if they do not exist in the source.
    - Create Directories: Ensure all directories in the source are present in the replica.
5. Periodic Syncronization ( Condition C):
    - The program will run in an infinite loop, performing synchronization at specified intervals.
6. Main Function:
    - Organize the entire process by integrating argument parsing, logging, and synchronization.

```
import os
import hashlib
import shutil
import time
import argparse
import logging

def compute_md5(file_path):
    """Compute MD5 hash of a file."""
    hash_md5 = hashlib.md5()
    with open(file_path, "rb") as f:
        for chunk in iter(lambda: f.read(4096), b""):
            hash_md5.update(chunk)
    return hash_md5.hexdigest()
```

""" This is the function definition. It takes one argument, file_path, which is the path to the file for which the MD5 hash will be calculated.

First command creates a new MD5 hash object using Python's hashlib module. The MD5 algorithm is widely used to generate a unique "fingerprint" (or hash) for a file, which allows us to compare files by their contents.

Second command opens the file at file_path in binary mode ("rb"). The binary mode is used because files can contain non-text data, and reading them as binary ensures compatibility with any file type (text, image, etc.).

Meaning of for-loop: Instead of reading the entire file into memory at once, which might be inefficient for large files, this reads the file in chunks of 4096 bytes (4KB).

- The iter(lambda: f.read(4096), b"") creates an iterator that reads the file chunk by chunk until it encounters an empty byte string (b""), signaling the end of the file.

Meaning the command in the for-loop:

For each chunk of the file, the hash_md5.update(chunk) method is called to update the MD5 hash with the contents of that chunk. The MD5 hash is accumulated progressively over the entire file.

Last command: After the entire file has been read and the hash updated with all chunks, the method hexdigest() returns the final MD5 hash value as a hexadecimal string, which is a unique representation of the file's contents

"""

```
def sync_folders(source, replica, logger):
    """Synchronize replica folder with source folder."""
    # Ensure all files and folders in source are present in replica
    for root, dirs, files in os.walk(source):
        # Compute relative path
        relative_path = os.path.relpath(root, source)
        replica_dir = os.path.join(replica, relative_path)

        # Ensure directory exists in replica
        if not os.path.exists(replica_dir):
            os.makedirs(replica_dir)
            logger.info(f"Created directory: {replica_dir}")
            print(f"Created directory: {replica_dir}")

        # Copy files
        for file in files:
            source_file = os.path.join(root, file)
            replica_file = os.path.join(replica_dir, file)

            if not os.path.exists(replica_file) or compute_md5(source_file) !=
compute_md5(replica_file):
                shutil.copy2(source_file, replica_file)
                logger.info(f"Copied/Updated file: {replica_file}")
                print(f"Copied/Updated file: {replica_file}")

    # Ensure all files and folders in replica match source
    for root, dirs, files in os.walk(replica):
        # Compute relative path
        relative_path = os.path.relpath(root, replica)
        source_dir = os.path.join(source, relative_path)

        # Remove directories not in source
        for dir in dirs:
            if not os.path.exists(os.path.join(source_dir, dir)):
                shutil.rmtree(os.path.join(root, dir))
                logger.info(f"Removed directory: {os.path.join(root, dir)}")
                print(f"Removed directory: {os.path.join(root, dir)}")

        # Remove files not in source
        for file in files:
            if not os.path.exists(os.path.join(source_dir, file)):
                os.remove(os.path.join(root, file))
                logger.info(f"Removed file: {os.path.join(root, file)}")
                print(f"Removed file: {os.path.join(root, file)}")
```

```
def main():
    # Argument Parsing with argparse
    parser = argparse.ArgumentParser(description="Synchronize two folders.")
    parser.add_argument("source", help="Source folder path")
    parser.add_argument("replica", help="Replica folder path")
    parser.add_argument("interval", type=int, help="Synchronization interval in seconds")
    parser.add_argument("logfile", help="Log file path")

    args = parser.parse_args()

    # Argument Assignment and Validation
    source = args.source
    replica = args.replica
    interval = args.interval

    # Check if source folder exists
    if not os.path.exists(source):
        print(f"Error: Source folder '{source}' does not exist.")
        sys.exit(1)

    # Setup logging
    logging.basicConfig(filename=args.logfile, level=logging.INFO, format="%(asctime)s - %
(message)s")
    logger = logging.getLogger()

    # Main Loop for Synchronization
    while True:
        logger.info("Starting synchronization.")
        print("Starting synchronization.")
        sync_folders(source, replica, logger)
        logger.info("Synchronization complete.")
        print("Synchronization complete.")
        time.sleep(interval)

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

Discussion: I would know, why there is no condition of termination? ( If user should terminate this program, what he should do? This condition would be also great for UML modeling….)