MPEG Sorter Technical Documentation

Overview

MPEG Sorter is a Python utility designed to analyze and sort media files based on their actual content signatures rather than relying on potentially misleading file extensions. The tool specifically addresses the common issue where MP3 audio files might be incorrectly saved with an .mp4 extension or MP4 video files might have an .mp3 extension.

Implementation Details

File Structure

```
mpeg-sorter/
  mpeg_sorter.py            # Main script
  README.md                 # User documentation
  TECHNICAL.md              # Technical documentation
  tests/
      conftest.py           # Pytest configuration
      test_mpeg_sorter_pytest.py    # Pytest-compatible test script
      data/                 # Test files directory
```

Dependencies

The script uses only Python standard library modules: - `pathlib`: For cross-platform path handling - `argparse`: For command-line argument parsing - `shutil`: For file operations - `concurrent.futures`: For parallel processing - `time`: For performance benchmarking

Core Components

The main script consists of several key components:

1. **File Signature Detection**:
   - Reads binary file headers to identify actual file types
   - Maintains a dictionary of known signatures for audio and video formats
2. **File Processor**:
   - Analyzes each file's signature
   - Determines appropriate destination based on content type
   - Handles extension mismatches and renaming
3. **Directory Manager**:
   - Creates necessary subdirectories
   - Ensures proper path handling across platforms
4. **Asynchronous Processing Engine**:

- Implements parallel processing using ThreadPoolExecutor
- Dynamically allocates workers based on system capabilities
- Provides fallback sequential processing option

File Signature Detection

The script detects file types using the first 12 bytes of each file, which typically contain the "magic numbers" or signature that identifies the file format:

```
def get_file_signature(file_path):
    """Read the first 12 bytes to identify the file type based on signatures."""
    signatures = {
        # MP3 signatures
        b'\xFF\xFB': 'mp3',          # MPEG-1 Layer 3
        b'\xFF\xF3': 'mp3',          # MPEG-2 Layer 3
        b'\xFF\xF2': 'mp3',          # MPEG-2.5 Layer 3
        b'ID3': 'mp3',               # ID3 tag (common in MP3 files)

        # MP4 signatures
        b'\x00\x00\x00\x18ftyp': 'mp4',  # ISO Base Media file (MPEG-4)
        b'\x00\x00\x00\x20ftyp': 'mp4',  # ISO Base Media file (MPEG-4)
        b'\x33\x67\x70\x35': 'mp4',      # 3GP5 (mobile MP4 variant)
    }

    try:
        with open(file_path, 'rb') as f:
            # Read first 12 bytes which typically contain signature
            header = f.read(12)

            for signature, file_type in signatures.items():
                if header.startswith(signature):
                    return file_type

        return None  # Unknown file type
    except Exception as e:
        print(f"Error reading {file_path}: {e}")
        return None
```

Analysis of Strengths and Weaknesses

Strengths

1. **Content-based identification**: Uses actual file signatures rather than unreliable extensions
2. **Extension correction**: Automatically fixes mislabeled file extensions
3. **Parallel processing**: Utilizes multi-threading for efficient processing of large directories

4. **Performance optimization**: Dynamically allocates workers based on system capabilities
5. **Flexibility**: Offers both parallel and sequential processing modes
6. **Low dependencies**: Relies solely on the Python standard library
7. **Cross-platform**: Works on Windows, macOS, and Linux
8. **Unknown file handling**: Provides option to organize unrecognized files
9. **Comprehensive testing**: Includes automated tests to verify functionality

Weaknesses

1. **Limited file format support**: Currently only recognizes MP3 and MP4 formats
2. **Basic signature detection**: Uses only the first 12 bytes, which may not be sufficient for some formats
3. **No deep scanning**: Cannot detect file types with signatures deeper in the file
4. **No content validation**: Only checks headers, not whether the entire file is valid
5. **Single-pass processing**: Does not support recursive directory scanning
6. **Minimal error recovery**: Basic error handling without sophisticated recovery mechanisms

Future Improvements

1. **Enhanced format support**: Add signatures for more audio/video formats (FLAC, AAC, MKV, etc.)
2. **Recursive directory support**: Add option to process nested folders
3. **Content validation**: Implement more thorough file validation
4. **Configuration file**: Allow custom signatures and paths via config file
5. **Progress bar**: Add visual progress indication for large operations
6. **Improved error handling**: More sophisticated error recovery mechanisms
7. **File deduplication**: Option to detect and handle duplicate files
8. **Metadata extraction**: Read and use audio/video metadata for organization
9. **GUI interface**: Develop a simple graphical interface

Implementation Considerations

Parallel Processing Implementation

The script implements parallel processing using Python's ThreadPoolExecutor, which significantly improves performance when processing large directories:

```
def sort_files_async(source_folder, create_unknown_folder=True, max_workers=None):
    """
```

```
    Sort files asynchronously (multi-threaded) from source folder into video and audio subd

    Args:
        source_folder: Path to the folder containing files to sort
        create_unknown_folder: Whether to create a folder for unknown file types
        max_workers: Maximum number of worker threads (None = auto-detect based on CPU cores
    """
    # Setup code omitted for brevity

    start_time = time.time()

    with concurrent.futures.ThreadPoolExecutor(max_workers=max_workers) as executor:
        # Submit all file processing tasks to the thread pool
        future_to_file = {executor.submit(process_file, item, ...): item for item in items}

        # Process results as they complete
        for future in concurrent.futures.as_completed(future_to_file):
            # Result processing code omitted for brevity

    end_time = time.time()
    print(f"Processed {total_files} files in {end_time - start_time:.2f} seconds")
```

The code also provides a sequential processing option for comparison:

```
def sort_files_sequential(source_folder, create_unknown_folder=True):
    """
    Sort files sequentially (single-threaded) from source folder into video and audio subdi
    """
    # Similar implementation but without threading
```

Testing Framework

The project includes a comprehensive testing framework that validates the functionality:

Test Structure

The `test_mpeg_sorter_pytest.py` file contains pytest-compatible test functions:

```
def test_parallel_processing(tmp_path):
    """Test the parallel processing functionality."""
    # Test code omitted for brevity

def test_sequential_processing(tmp_path):
    """Test the sequential processing functionality."""
    # Test code omitted for brevity
```

```python
def test_unknown_file_handling(tmp_path):
    """Test handling of unknown file types."""
    # Test code omitted for brevity
```

Test Data Generation

The test framework automatically generates sample test files with appropriate signatures:

```python
def _create_test_files(self):
    """Create test files with appropriate signatures for testing."""
    # Generate MP3 file incorrectly named as MP4
    with open(self.data_dir / "audio1.mp4", "wb") as f:
        f.write(b'\xFF\xFB' + b'\x00' * 10)  # MP3 signature

    # Generate MP4 file incorrectly named as MP3
    with open(self.data_dir / "video1.mp3", "wb") as f:
        f.write(b'\x00\x00\x00\x18ftyp' + b'\x00' * 6)  # MP4 signature

    # Generate properly named files and unknown file
    # Code omitted for brevity
```

Test Cleanup

After testing, the framework automatically restores the original structure:

```python
def _cleanup(self):
    """Restore original file structure after test."""
    # Move all files back to original data directory
    # Delete subdirectories created during testing
```

Running Tests

Tests can be run using pytest:

```
pytest -xvs test_mpeg_sorter_pytest.py
```

The test suite validates: 1. Correct file identification 2. Proper directory creation 3. Accurate file movement and renaming 4. Both parallel and sequential processing modes 5. Error handling and unknown file processing