Partage Cohérent de Fichiers Mappés en Mémoire

Contributors

name	email	student number
Felipe Paris Mollo Christondis	felipe.paris@etu.sorbonne-universite.fr	21305222
Victor Spehar	vgdspehar@gmail.com	21309680

Mentor

Pierre Sens - pierre.sens@lip6.fr

Introduction

Problem

Memory Mapping is an advanced feature in operating systems, it allows a file stored on the disk to be projected into the virtual memory. This will facilitate the manipulation of the file, since the contents are directly accessible as if it were in the RAM, changes or reading are done more quickly, by manipulating memory addresses.

When multiple processes access a memory mapped file, they all see the same content, but if these processes start writing to the file mapped with `MAP_SHARED`, all writes are visible, and that will likely create conflicts or inconsistencies. The goal of this project is to allow each process to write to a shared mapped memory file without modifying the common file and therefore avoiding creating conflicts that could affect other processes also using that same file.

Solution Proposed

To solve the problem mentioned above, we can implement a copy on write mechanism.

1. All processes share the same view of a file as long as they are read only.

2. When a process attempts to write to the file, the system generates a SIGSEGV signal, a manager (signal handler) of this signal is then called.
3. The handler will copy the modified memory page to a new location, it adjusts the page rights to allow both reading and writing, thanks to the `mprotect` system call.
4. The physical address in the page is updated to point to the new copied page - use of the API `pteditor`.
5. Each process keeps its changes in a dedicated log file. The original file is never modified directly.
6. At the end of operations, a merge command merges the original file with the changes saved in the log files, to create an update version of the file.

Project Setup and Structure

```
.
├── app # Contains the entry point of the program
│   └── main.c
├── docs # Include the project report
│   └── rapport_final.md
├── files # Directory to store files for processes to interact with
├── include # Header files for api.h and pteditor
│   ├── api.h
│   └── ptedit_header.h
├── logs # Store log files generated during processes execution
├── Makefile
├── merge # Store the output of files when merged with logs
├── README.md
└── src # Contains the source file api.c that implements the functions
declared in the api.h
    └── api.c

7 directories, 7 files
```

Core functions and Their implementations

```
bool start_file_write_processes();
```

This function initializes signal handlers and the PTeditor API. It attempts to create a specified number of child processes (NUMBER_OF_PROCESSES). Each child is tasked with performing file modifications through the perform_file_modifications function. If a child process fails to start (i.e., fork returns a negative value), an error is logged, and the all_success flag is set to false. If the child starts successfully but fails during file modification, it logs an error and exits with EXIT_FAILURE. If any child process creation fails, the parent terminates all successfully started children and exits the loop early. If all child processes are created and complete their tasks successfully, the parent checks their exit statuses to ensure each child exited successfully. Any failure updates all_success to false. After managing the child processes, the function performs a cleanup using the ptedit_cleanup API and returns the all_success status.

```
bool log_and_write_memory_region(char *mapped_region, off_t offset, const
char *data, size_t len, size_t region_size, char * file_name);
```

This function modifies a memory-mapped region and logs the changes. It sets up the log directory and records details such as the offset, data length, and actual data, e.g., Offset: 15, Length: 3, Data: xxx. After logging, it copies the data to the specified offset within the memory-mapped region.

```
bool merge(const char* original_file_path, const char* log_file_path);
```

The merge function creates a new version of an original file by incorporating changes detailed in a log file. It first copies the original file to another dedicated file. It then calls the apply_merge function to apply modifications based on the data in the log file.

```
bool perform_file_modifications();
```

This function attempts to write to NUMBER_OF_FILES read-only files. It first maps the files into memory with read-only access. It then calls log_and_write_memory_region, which will trigger the signal handler in the event of a write attempt on read-only memory.

```
void signal_handler(int sig, siginfo_t * si, void * unused);
```

The signal_handler function handles segmentation faults that occur when a process attempts to write to a read-only memory region. It captures the address where the fault occurred, copies the contents from the faulting address to a new mapped page (with write authorization), and then modifies the page table entry for the faulting address to point to the new memory page. This involves retrieving and manipulating the Page Table Entries of both the faulting and the new pages. The page table is then updated to reference the new page instead of the old read-only page.

```
void apply_merge(int to_fd, int from_fd);
```

The apply_merge function processes log information from one file descriptor (from_fd) and applies the described changes to another file (to_fd). It reads the log file line by line, extracting values for offset, length, and data using sscanf. Once the information is extracted, it adjusts the write position of the target file descriptor and writes the data to it.

Compilation and Execution

Makefile

```
# suppresses command echo - no prints to the terminal
.SILENT:

CC=gcc # compiler
CFLAGS=-I./include # tells compiler to include the include folder during
header file lookups

# Name of the executable
EXEC=psar

# Source files - find all .c files
SRC=$(wildcard src/*.c app/*.c)
OBJS=$(SRC:.c=.o)

# Default build target
all: $(EXEC)

$(EXEC): $(OBJS)
    @echo "Building $@"
    @$(CC) $(CFLAGS) $^ -o $@
    @echo "Build complete"
# clean project set up
clean:
    @echo "Cleaning up"
    @rm -f $(OBJS) $(EXEC)
    @rm -f files/*
    @rm -rf logs/*
    @rm -rf merge/*
    @echo "Clean complete"
# avoid confusion if files were named like this
.PHONY: all clean
```

To build the project, run `make`. To remove compiled files and clean the project directory, run `make clean`.

Running the program

> *Attention: Do not forget to install the pteditor tool and to load the module to your kernel.*

Once the project has been compiled, you will find the executable `psar` in the project directory.

- `./psar init`: Initializes the project environment.
- `./psar test`: Starts the file writing processes for testing. See variables `DATA_DEMO`, `WRITE_DEMO` and `WRITE_OFFSET` in `api.h` for testing different scenarios. Once this commmand finishes, see the `logs/` folder for the results.
- `./psar merge -s [source_file] -l [log_file]`: Merges a source file with its corresponding log file. Once this commmand finishes, see the `merge/` folder for the results.
- `./psar merge_all -s [source_file]`: Merges all logs associated with a source file. Once this commmand finishes, see the `merge/` folder for the results.

Conclusion

The project demonstrates how to handle concurrent file access through memory mapping by implementing a copy-on-write system. This approach ensures that the integrity of data is maintained across multiple processes, therefore solving the issue of data consistency in a shared memory environment.

Acknowledgments

This project would not have been possible without the advice and expertise of Pierre Sans, who generously took the time to answer questions and resolve doubts about the best way to implement the project. Likewise, we are grateful to the creators of the PTEditor API (https://github.com/misc0110/PTEditor), who developed this tool that facilitated the manipulation of page table entries.