CSE 344 HW5

REPORT

ŞEHMUS ACAR
161044085
CSE344

Additions and Enhancements to HW4

In HW4, I completed the directory copying tool (MWCp) using only condition variables. In HW5, I made the following additions and enhancements:

1. **Condition Variables:**
    - Added pthread_cond_wait(&buffer.not_full) for the manager thread to wait when the buffer is full.
    - Added pthread_cond_wait(&buffer.not_empty) for the worker threads to wait when the buffer is empty.
    - Implemented signaling of the appropriate condition variables (pthread_cond_signal(&buffer.not_empty) and pthread_cond_signal(&buffer.not_full)) when the buffer is full or empty.
2. **Barrier Usage:**
    - Added pthread_barrier_wait(&barrier) to synchronize worker threads at a certain point.

In HW4, I completed the assignment using only condition variables. In HW5, I used both condition variables and barriers

Main Program *int main(int argc, char *argv[])*

The main program checks the command line arguments to retrieve the buffer size, number of workers, source, and destination directories. It then initializes the buffer, starts the manager and worker threads, and waits for them to complete. During execution, it measures the start and end times to calculate the elapsed time and prints the statistics.

Explanation: In my implementation, the main program takes the buffer size, number of workers, source, and destination directories from the user. It initializes the buffer using the **initialize_buffer** function, creates and starts the manager and worker threads. After all threads have completed their tasks, it measures the start and end times of the program to calculate the elapsed time. Finally, it prints the statistics of the copying process and cleans up the buffer with the **free_buffer** function.

Condition Variables Usage

In my implementation, condition variables **pthread_cond_t not_full** and **pthread_cond_t not_empty** are used for synchronization between manager and worker threads. The manager thread waits with **pthread_cond_wait** if the buffer is full and signals **pthread_cond_signal** when items are added. Worker threads wait with **pthread_cond_wait** if the buffer is empty and signal **pthread_cond_signal** when items are removed. This ensures efficient coordination, preventing race conditions and busy-waiting. **Buffer Initialization** *void initialize_buffer(int size)*

This function initializes the buffer structure and allocates the necessary memory. It also initializes the **pthread_mutex** and **pthread_cond** variables.

Explanation: In my implementation, the **initialize_buffer** function sets up the buffer structure and allocates the necessary memory. It sets the **in**, **out**, and **count** values of the buffer to zero, ensuring that the buffer is ready for use. Additionally, it initializes the **pthread_mutex** and **pthread_cond** variables to manage synchronization between threads.

Manager Thread *void *manager_function(void *arg)*

The manager thread reads the source directory and enqueues file and directory names into the buffer. It skips hidden files and directories. If the buffer is full, it waits until it is emptied before adding more entries. When the manager finishes processing, it sets the **done_flag** and notifies the workers.

Explanation: In my implementation, the **manager_function** reads the source directory and enqueues file and directory names into the buffer. It skips hidden files and directories. If the buffer is full, it waits using **pthread_cond_wait** until it is emptied. The function updates the statistics based on file types (regular file, FIFO, directory). When the manager finishes processing, it sets the **done_flag** and notifies the workers using **pthread_cond_broadcast**.

Worker Thread *void *worker_function(void *arg)*

The worker thread dequeues file and directory names from the buffer and performs the copying operation. It handles different file types (regular files, FIFOs, directories) appropriately. If the buffer is empty, it waits until more items are added. It updates the statistics after each operation.

Explanation: In my implementation, the **worker_function** dequeues file and directory names from the buffer and performs the copying operation. It handles different file types: copies regular files, creates FIFO files, and recreates directories in the destination directory. If the buffer is empty, it waits using **pthread_cond_wait** until more items are added. It updates the statistics after each copying operation and checks the buffer to continue processing or determine if the process is complete. **Buffer Cleanup** *void free_buffer()*

This function cleans up the buffer structure and frees dynamically allocated memory. It also destroys the **pthread_mutex** and **pthread_cond** variables.

Explanation: In my implementation, the **free_buffer** function cleans up the buffer structure and frees dynamically allocated memory. It also destroys the **pthread_mutex** and **pthread_cond** variables to ensure proper resource management and avoid memory leaks.

Signal Handling *void signal_handler(int sig)*

The signal handler ensures the program terminates gracefully.

Explanation: In my implementation, the **signal_handler** function ensures the program terminates gracefully. It catches the **SIGINT** signal and safely shuts down the program.

The algorithm of my project follows these steps:

1. The main program checks the command line arguments and retrieves the buffer size, number of workers, source, and destination directories.
2. It initializes the buffer structure using the **initialize_buffer** function.
3. The program creates and starts the manager and worker threads.
4. The manager thread reads the source directory and enqueues file and directory names into the buffer.
5. Worker threads dequeue file and directory names from the buffer and perform the copying operation.
6. When the manager thread has enqueued all file and directory names, it sets the **done_flag** and notifies the workers.
7. Worker threads complete all copying operations, and then the program terminates.
8. The program measures the elapsed time and prints the statistics.
9. Finally, the buffer structure is cleaned up using the **free_buffer** function.

Testing and Results

Test1: Checking for memory leaks using valgrind

```
acar@acar:~/Desktop/hw4test/put_your_codes_here$  valgrind ./MWCp 10 10 ../testdir/src/libvterm ../tocopy
==3664== Memcheck, a memory error detector
==3664== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et al.
==3664== Using Valgrind-3.18.1 and LibVEX; rerun with -h for copyright info
==3664== Command: ./MWCp 10 10 ../testdir/src/libvterm ../tocopy
==3664==

--------------STATISTICS-------------------
Consumers: 10 - Buffer Size: 10
Number of Regular Files: 194
Number of FIFO Files: 0
Number of Directories: 7
TOTAL BYTES COPIED: 25009680
TOTAL TIME: 00:03.092 (min:sec.mili)
==3664==
==3664== HEAP SUMMARY:
==3664==     in use at exit: 0 bytes in 0 blocks
==3664==   total heap usage: 223 allocs, 223 frees, 274,872 bytes allocated
==3664==
==3664== All heap blocks were freed -- no leaks are possible
==3664==
==3664== For lists of detected and suppressed errors, rerun with: -s
==3664== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
acar@acar:~/Desktop/hw4test/put_your_codes_here$
```

- The valgrind report should indicate no memory leaks.
- Memory leaks: None

Test2: Buffer size 10, number of workers 4

```
acar@acar:~/Desktop/hw4test/put_your_codes_here$ ./MWCp 10 4 ../testdir/src/libvterm/src ../toCopy

--------------STATISTICS-------------------
Consumers: 4 - Buffer Size: 10
Number of Regular Files: 140
Number of FIFO Files: 0
Number of Directories: 2
TOTAL BYTES COPIED: 24873082
TOTAL TIME: 00:00.499 (min:sec.mili)
acar@acar:~/Desktop/hw4test/put_your_codes_here$
```

Test3: Buffer size 10, number of workers 10

```
acar@acar:~/Desktop/hw4test/put_your_codes_here$ ./MWCp 10 10 ../testdir ../toCopy

--------------STATISTICS--------------------
Consumers: 10 - Buffer Size: 10
Number of Regular Files: 3118
Number of FIFO Files: 0
Number of Directories: 151
TOTAL BYTES COPIED: 73520649
TOTAL TIME: 00:01.736 (min:sec.mili)
acar@acar:~/Desktop/hw4test/put_your_codes_here$
```

Test4: FIFO File Handling

```
acar@acar:~/Desktop/Sistem Ödev/HW4$ mkfifo ../testdir/src/libvterm/fifo test
acar@acar:~/Desktop/Sistem Ödev/HW4$ ./MWCp 10 4 ../testdir/src/libvterm ../toCopy
Manager: Added file2.txt to buffer, type: Regular
Manager: Added fifo test to buffer, type: FIFO
Manager: Added file1.txt to buffer, type: Regular
Worker: Copying FIFO ../testdir/src/libvterm/fifo test to ../toCopy/fifo test
Worker: Copied ../testdir/src/libvterm/fifo test to ../toCopy/fifo test, type: FIFO
Worker: Copied ../testdir/src/libvterm/file2.txt to ../toCopy/file2.txt, type: Regular
Worker: Copied ../testdir/src/libvterm/file1.txt to ../toCopy/file1.txt, type: Regular

--------------STATISTICS--------------------
Consumers: 4 - Buffer Size: 10
Number of Regular Files: 2
Number of FIFO Files: 1
Number of Directories: 0
TOTAL BYTES COPIED: 36
TOTAL TIME: 00:00.002 (min:sec.mili)
acar@acar:~/Desktop/Sistem Ödev/HW4$
```

Test5: Signal Handling (Ctrl+C)

```
acar@acar:~/Desktop/Sistem Ödev/HW4$ ./MWCp 10 4 ../testdir/src/libvterm ../toCopy
Manager: Added file2.txt to buffer, type: Regular
Manager: Added fifo test to buffer, type: FIFO
Manager: Added file1.txt to buffer, type: Regular
Worker: Copying FIFO ../testdir/src/libvterm/fifo test to ../toCopy/fifo test
mkfifo: File exists
Worker: Copied ../testdir/src/libvterm/file2.txt to ../toCopy/file2.txt, type: Regular
Worker: Copied ../testdir/src/libvterm/file1.txt to ../toCopy/file1.txt, type: Regular
Worker: Copied ../testdir/src/libvterm/fifo test to ../toCopy/fifo test, type: FIFO
^C
Terminating program...
acar@acar:~/Desktop/Sistem Ödev/HW4$
```

```c
// Signal handler to terminate the program gracefully
void signal_handler(int sig) {
    printf("\nTerminating program...\n");
    exit(0);
}
```