CSE 344 HOMEWORK 5

PCP

A PARALLEL COPY PROGRAM

Çağrı Çaycı

1901042629

OVERVIEW

Parallel Copy Program is a tool to enhance the efficiency and speed of file copying/transferring. In this project, parallel copying mechanism provided with multithread approach. To explain, each file is copied with a thread from the available threads in the thread pool. In this way, files can be copied in parallel instead of waiting for the previous task to be completed. One of the possible problems which can be encountered is synchronization. The synchronization is ensured with the solution of a well-known problem producer-consumer.

CONCEPTS

MULTITHREAD APPROACH

The program must create a thread to produce items and multiple threads to consume items.

```
pthread_create(&producer, NULL, produce, (void *)argv);

for(int i = 0; i < number_of_consumers; i++){
    pthread_create(&consumers[i], NULL, consume, NULL);
}

pthread_join(producer, NULL);

for(int i = 0; i < number_of_consumers; i++){
    pthread_join(consumers[i], NULL);
}
```

After the producer thread produces some file descriptors, consumer threads can copy them in parallel. So, the file copying operations are done in multithread manner. As can be seen in the adjacent picture, it is probable that dead locks occur. For example, if the consumer's lock function runs before the producer's lock, the program is stuck forever in between lock and unlock function in consumer. To avoid these deadlocks, condition variables are used.

```
void producer(){
    while(1){
        lock();
        produce();
        unlock();
    }
}
void consumer(){
    while(1){
        lock();
        consume();
        unlock();
    }
}
```

```
void producer(){
    while(1){
        lock();
        while(buffer == full){
            wait();
        }
        produce();
        signal()
        unlock();
    }
}
void consumer(){
    while(1){
        lock();
        while(buffer == 0){
            wait();
        }
        consume();
        signal();
        unlock();
    }
}
```

Condition variables provide the synchronization by wait and signal functions. While the producer is running, if the buffer is full, it waits until there is space in the buffer. After the consumer consumes an item, it sends a signal to the producer to tell there is at least one empty slot in the buffer. On the other hand, while consumer is running, if the buffer is empty, it waits until there is an item in the buffer. After the producer produces an item, it sends a signal to the consumer to tell there is at least one item in the buffer. Also, the producer and consumer check the buffer in while loop to avoid spurious awakes. In this way, threads are used safely.

```
pthread_mutex_lock(&mutex);

done = 1;

pthread_mutex_unlock(&mutex);

for(int i = 0; i < number_of_consumers; i++){
    pthread_cond_signal(&notEmpty);
}

pthread_exit(NULL);
```

Addition to producer-consumer problem, a done flag is added to program because the program is not finite program like producer-consumer problem. After the all files are copied in the directory, the task of the producer is done. So, it sends signal to consumers to awake them and terminate them.

HANDLING SIGNALS

```
void SIGINTHandler(){
    pthread_mutex_lock(&mutex);
    sigint = 1;
    pthread_mutex_unlock(&mutex);
    for(int i = 0; i < number_of_consumers; i++){
        pthread_join(consumers[i], NULL);
    }
    pthread_join(producer, NULL);
    free(consumers);
    free(buffer);

    pthread_cond_destroy(&notEmpty);
    pthread_cond_destroy(&notFull);

    exit(0);
}
```

As the design of the program, the threads run until all the files are copied from the source directory to destination directories. However, if the SIGINT signal is sent to the program during the execution, all threads must be closed in proper way. To provide this utility, a flag as sigint is added to the program and threads are set to end when the SIGINT signal is received.

WORKING STEPS

1) Gets the buffer size, number of consumers and the source directory and the destination directories from the user.
2) Creates a producer thread and n number of consumer threads.
3) The producer thread opens source and destination directory, creates the destination directory if it does not exist.
4) The producer thread opens a file from the source directory, creates a file with the same name in the destination directory.
5) Puts the file descriptors of these files to buffer.
6) Consumer reads the buffer.
7) Copies the file from source directory to destination directory.
8) Repeats last five steps.

TESTS

```
cagri@CSE344:~/Masaüstü/HW5$ ls
Makefile  readMe.txt  source  src.c
cagri@CSE344:~/Masaüstü/HW5$ make
gcc src.c -o pCp
cagri@CSE344:~/Masaüstü/HW5$ ./pCp 15 10 /home/cagri/Masaüstü/HW5/source /home/cagri/Masaüstü/HW5/destination
STATISTICS:
211 number of files are copied.
6 number of subdirectories are copied.
2474465825 number of bytes copied.
The process takes 47.631570 seconds.
cagri@CSE344:~/Masaüstü/HW5$ diff -r source/ destination/
cagri@CSE344:~/Masaüstü/HW5$
```

(There is no difference between folders.)

```
cagri@CSE344:~/Masaüstü/HW5$ ls
destination1  Makefile  pCp  readMe.txt  source  src.c
cagri@CSE344:~/Masaüstü/HW5$ diff -r destination1/ source/
Only in source/: Makefile
Only in source/: pCp
cagri@CSE344:~/Masaüstü/HW5$ make
gcc src.c -o pCp
cagri@CSE344:~/Masaüstü/HW5$ ./pCp 15 10 /home/cagri/Masaüstü/HW5/source /home/cagri/Masaüstü/HW5/destination1 /home/cagri/Masaüstü/HW5/destination2
STATISTICS:
8 number of files are copied.
0 number of subdirectories are copied.
154128 number of bytes copied.
The process takes 0.16533 seconds.
cagri@CSE344:~/Masaüstü/HW5$ ls
destination1  destination2  Makefile  pCp  readMe.txt  source  src.c
cagri@CSE344:~/Masaüstü/HW5$ diff -r destination1/ source/
cagri@CSE344:~/Masaüstü/HW5$ diff -r destination2/ source/
cagri@CSE344:~/Masaüstü/HW5$
```

(Copying the source file to two destination files which one of them has already some files in it.)