YEDİTEPE ÜNİVERSİTESİ

FALL - 2024

CSE331 - Operating Systems Design

ASSIGNMENT - 2

SECTION - 2

NAME SURNAME : ULAŞ CAN DEMİRBAĞ

ID : 20200702119

INSTRUCTOR : Prof. Dr. ŞEBNEM BAYDERE

Table of Contents

Figures

Tables

1. Introduction

Image processing tasks are among those that require a high degree of computational power and hence demand parallel processing techniques. These include filtering. The comparative study here examines two design approaches: a multi-process model using shared semaphores and a multi-threaded model using POSIX semaphores. The effectiveness of the models is tested based on their ability to run unsharp masking, one of today's most popular image enhancements, under different loads.

2. Design and Implementation

Both programs are implemented in C. hw2a uses the fork() system call to create child processes and shared memory areas for inter-process communication. It employs semaphores for synchronizing the execution of these processes. hw2b, on the other hand, utilizes POSIX threads with semaphores to control the order of operations between threads, ensuring that no thread performs subtraction before the convolution operation is completed by another thread.

The core image processing operations for both programs are:

- Convolution using a Laplacian filter matrix to enhance edges,
- Subtraction to finalize the unsharp masking by emphasizing edges over the original.

2.1 Multi-process Program with Shared Semaphores (hw2a.c)

hw2a program uses a multitasking approach to accomplish image unsharp masking. This program will utilize shared memory between processes for the data to be shared and shared semaphores for synchronization of access.

- **Shared Memory:** In shared memory, it allocates two NxN matrices, A and B. A is used to hold the original image, while B is utilized in this program for the result of the Laplacian convolution and the subtraction which ensues.
- **Processes:** Each matrix row is assigned to a different child process via fork(). Each child is responsible for one row of convolution in the matrix A with the predefined Laplacian matrix L and stores the result in the corresponding row of matrix B.
- **Synchronization:** The use of semaphores will ensure that the subtraction operation on matrix B will not be executed until the convolution for that particular row is finished. The program uses the named semaphores-sem_open with unique names for each row-to synchronize the operations between the two sets of child processes dealing with convolution and subtraction.

2.2 Multi-threaded Program with POSIX Semaphores (hw2b.c)

The hw2b program uses POSIX threads instead of processes. Yet another, lighter-weight way to take advantage of parallel processing by a single application, using much less memory than multiple processes.

- **Thread Creation:** As in hw2a, one thread is created for each row in an image where convolution is to be done. For subtraction, one thread per row is created ensuring that different threads work with different parts of the matrices.
- **Data Shareability:** Unlike hw2a, which used inter-thread, there were no needs actually to take care of shared memory segments when comparing to hw2a and hw2b by virtue since now hw2b is inherently threading bound, where it actually becomes easier for the compiler; however handling in one's own implementation carefully and avoiding race conditions needs significant modifications at the time of their personal implementation.
- **Semaphores:** Unnamed POSIX semaphores created with sem_init; hence it requires just about one per matrix rows - ensuring that the start thread's subtraction thread waits to find itself synchronized with a convolution thread completed by the very same matching pair of threads.

2.3 Folder Structure & Makefile

2.3.1 Folder Structure

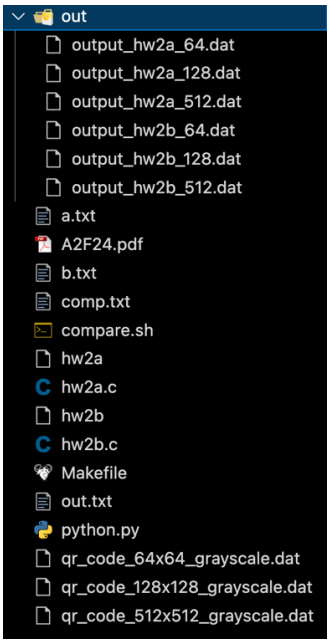

Figure 1 Folder Structure Of Assigment 2

The out folder stands for the project directory with organized structure source code, executables, input, and output data, supplementary utilities necessary for a university assignment that will be dealing with image processing based on semaphores. It includes C source files "hw2a.c" and "hw2b.c" for multi-process and multi-threaded implementations respectively, with corresponding executable files "hw2a" and "hw2b", and various grayscale image data files "qr_code_64x64_grayscale.dat" to "qr_code_512x512_grayscale.dat" serving as inputs. This work keeps the output of every implementation in ".dat" format for different sizes of pictures in respective files. Its supporting files include one Makefile for compilation and running, a python script "python.py" post-processing or visualization of its output, text logs "a.txt", "b.txt", performances comparison results "comp.txt", a shell script that shows comparisons "compare.sh." This structure ensures that each entity is efficiently accessible and maintained to support a streamlined development and analysis workflow.

2.3.2 Makefile

This Makefile makes it easy to compile, run, and process the results of two programs hw2a and hw2b using a set of clear, defined rules.

```
CC = gcc
LDFLAGS = -pthread

all: hw2a hw2b

hw2a: hw2a.c
    $(CC) $(CFLAGS) hw2a.c -o hw2a $(LDFLAGS)

hw2b: hw2b.c
    $(CC) $(CFLAGS) hw2b.c -o hw2b $(LDFLAGS)

run: run-a run-b

run-a:
    ./hw2a qr_code_64x64_grayscale.dat > out/output_hw2a_64.dat
    ./hw2a qr_code_128x128_grayscale.dat > out/output_hw2a_128.dat
    ./hw2a qr_code_512x512_grayscale.dat > out/output_hw2a_512.dat

run-b:
    ./hw2b qr_code_64x64_grayscale.dat > out/output_hw2b_64.dat
    ./hw2b qr_code_128x128_grayscale.dat > out/output_hw2b_128.dat
    ./hw2b qr_code_512x512_grayscale.dat > out/output_hw2b_512.dat

py: pya pyb

pya:
    python3 python.py out/output_hw2a_64.dat
    python3 python.py out/output_hw2a_128.dat
    python3 python.py out/output_hw2a_512.dat

pyb:
    python3 python.py out/output_hw2b_64.dat
    python3 python.py out/output_hw2b_128.dat
    python3 python.py out/output_hw2b_512.dat

pytest:
    python3 python.py qr_code_64x64_grayscale.dat
    python3 python.py qr_code_128x128_grayscale.dat
    python3 python.py qr_code_512x512_grayscale.dat


comp:
    sh compare.sh > comp.txt

clean:
    rm -f hw2a hw2b
    rm -f out/output_hw2a_64.dat out/output_hw2a_128.dat out/output_hw2a_512.dat
    rm -f out/output_hw2b_64.dat out/output_hw2b_128.dat out/output_hw2b_512.dat

```

Figure 2 Makefile Of Assigment 2

- **The "all" Rule:** The rule kicks things off by compiling both hw2a and hw2b. Each program has its own set of compilation instructions, and both use the pthread library (thanks to specific linker flags).
- **The "run" Rule:** Once the programs are compiled, the "run" rule takes over. It runs hw2a and hw2b using grayscale QR code data files of different sizes. The results are saved neatly in the out/ directory for easy access later.
- **The "py" Rule:** Next, the "py" rule jumps in to process the output files using a Python script called python.py. This script is applied separately to the outputs from hw2a and hw2b, giving you processed results for both.

- **The "pytest" Rule:** If you want to directly test the QR code files with Python scripts, the "pytest" rule has you covered. It runs tests on the QR code files to make sure everything checks out.

- **The "comp" Rule:** Need to compare the outputs from hw2a and hw2b? The "comp" rule runs a shell script called compare.sh. The comparison results are saved in a file named comp.txt, so you can easily review any differences.

- **The "clean" Rule:** Finally, there's the "clean" rule, which clears out compiled programs and output files.

3. Results

To see how well the two semaphore-based programs hw2a and hw2b perform, we set up a two-part benchmarking process. Each program keeps track of its own run time and records a.txt and b.txt files, but we go a step further by using the gtime command in a shell script called compare.sh. This script runs both programs automatically with three different grayscale image sizes: 64x64, 128x128, and 512x512. For each test, compare.sh collects key performance data like CPU usage, memory usage, and execution time. This setup gives us a clear, side-by-side look at how each program handles different workloads. By following this method, we can spot differences in scalability and efficiency, and since it's automated, we get results that are both accurate and repeatable.

```
#!/bin/bash

PROGRAMS=("hw2a" "hw2b")
INPUTS=("qr_code_64x64_grayscale.dat" "qr_code_128x128_grayscale.dat" "qr_code_512x512_grayscale.dat")

for prog in "${PROGRAMS[@]}"; do
    for inp in "${INPUTS[@]}"; do
        (gtime -v ./$prog $inp >/dev/null) 2>&1
    done
done

```

Figure 3 compare.sh Of Assigment 2

This will make sure that each program was tested under the same circumstances, hence allowing a good comparison of performances for every program across all image sizes. The verbose output of gtime also includes but is not limited to elapsed time, system time, and user time with other system resource metrics such as memory usage.

```
Input File: qr_code_64x64_grayscale.dat, Time: 0.018850 seconds, Memory: 0 bytes
Input File: qr_code_128x128_grayscale.dat, Time: 0.032842 seconds, Memory: 0 bytes
Input File: qr_code_512x512_grayscale.dat, Time: 0.149020 seconds, Memory: 0 bytes
```

Figure 4 a.txt Of Assigment 2

```
Input File: qr_code_64x64_grayscale.dat, Time: 0.002190 seconds, Memory: 0 bytes
Input File: qr_code_128x128_grayscale.dat, Time: 0.004989 seconds, Memory: 0 bytes
Input File: qr_code_512x512_grayscale.dat, Time: 0.032480 seconds, Memory: 0 bytes
```

Figure 5 b.txt Of Assigment 2

hw2a.c gtime results	hw2b.c gtime results
Command being timed: "./hw2a qr_code_64x64_grayscale.dat" User time (seconds): 0.00 System time (seconds): 0.02 Percent of CPU this job got: 194% Elapsed (wall clock) time (h:mm:ss or m:ss): 0:00.01 Average shared text size (kbytes): 0 Average unshared data size (kbytes): 0 Average stack size (kbytes): 0 Average total size (kbytes): 0 Maximum resident set size (kbytes): 1072 Average resident set size (kbytes): 0 Major (requiring I/O) page faults: 4 Minor (reclaiming a frame) page faults: 10431 Voluntary context switches: 1 Involuntary context switches: 185 Swaps: 0 File system inputs: 0 File system outputs: 0 Socket messages sent: 0 Socket messages received: 0 Signals delivered: 0 Page size (bytes): 16384 Exit status: 0	Command being timed: "./hw2b qr_code_64x64_grayscale.dat" User time (seconds): 0.00 System time (seconds): 0.00 Percent of CPU this job got: 133% Elapsed (wall clock) time (h:mm:ss or m:ss): 0:00.00 Average shared text size (kbytes): 0 Average unshared data size (kbytes): 0 Average stack size (kbytes): 0 Average total size (kbytes): 0 Maximum resident set size (kbytes): 3200 Average resident set size (kbytes): 0 Major (requiring I/O) page faults: 4 Minor (reclaiming a frame) page faults: 430 Voluntary context switches: 0 Involuntary context switches: 149 Swaps: 0 File system inputs: 0 File system outputs: 0 Socket messages sent: 0 Socket messages received: 0 Signals delivered: 0 Page size (bytes): 16384 Exit status: 0
Command being timed: "./hw2a qr_code_128x128_grayscale.dat" User time (seconds): 0.00 System time (seconds): 0.05 Percent of CPU this job got: 187% Elapsed (wall clock) time (h:mm:ss or m:ss): 0:00.03 Average shared text size (kbytes): 0 Average unshared data size (kbytes): 0 Average stack size (kbytes): 0 Average total size (kbytes): 0 Maximum resident set size (kbytes): 1216 Average resident set size (kbytes): 0 Major (requiring I/O) page faults: 2 Minor (reclaiming a frame) page faults: 20584 Voluntary context switches: 3 Involuntary context switches: 374 Swaps: 0 File system inputs: 0 File system outputs: 0 Socket messages sent: 0 Socket messages received: 0 Signals delivered: 0 Page size (bytes): 16384 Exit status: 0	Command being timed: "./hw2b qr_code_128x128_grayscale.dat" User time (seconds): 0.00 System time (seconds): 0.00 Percent of CPU this job got: 133% Elapsed (wall clock) time (h:mm:ss or m:ss): 0:00.00 Average shared text size (kbytes): 0 Average unshared data size (kbytes): 0 Average stack size (kbytes): 0 Average total size (kbytes): 0 Maximum resident set size (kbytes): 5376 Average resident set size (kbytes): 0 Major (requiring I/O) page faults: 2 Minor (reclaiming a frame) page faults: 699 Voluntary context switches: 0 Involuntary context switches: 285 Swaps: 0 File system inputs: 0 File system outputs: 0 Socket messages sent: 0 Socket messages received: 0 Signals delivered: 0 Page size (bytes): 16384 Exit status: 0
Command being timed: "./hw2a qr_code_512x512_grayscale.dat" User time (seconds): 0.05 System time (seconds): 0.24 Percent of CPU this job got: 172% Elapsed (wall clock) time (h:mm:ss or m:ss): 0:00.17 Average shared text size (kbytes): 0 Average unshared data size (kbytes): 0 Average stack size (kbytes): 0 Average total size (kbytes): 0 Maximum resident set size (kbytes): 3136 Average resident set size (kbytes): 0 Major (requiring I/O) page faults: 1 Minor (reclaiming a frame) page faults: 82541 Voluntary context switches: 0 Involuntary context switches: 2330 Swaps: 0 File system inputs: 0 File system outputs: 0 Socket messages sent: 0 Socket messages received: 0 Signals delivered: 0 Page size (bytes): 16384 Exit status: 0	Command being timed: "./hw2b qr_code_512x512_grayscale.dat" User time (seconds): 0.04 System time (seconds): 0.02 Percent of CPU this job got: 148% Elapsed (wall clock) time (h:mm:ss or m:ss): 0:00.04 Average shared text size (kbytes): 0 Average unshared data size (kbytes): 0 Average stack size (kbytes): 0 Average total size (kbytes): 0 Maximum resident set size (kbytes): 19664 Average resident set size (kbytes): 0 Major (requiring I/O) page faults: 1 Minor (reclaiming a frame) page faults: 2368 Voluntary context switches: 0 Involuntary context switches: 1607 Swaps: 0 File system inputs: 0 File system outputs: 0 Socket messages sent: 0 Socket messages received: 0 Signals delivered: 0 Page size (bytes): 16384 Exit status: 0

Table 1 gtime Outputs Of hw2a & hw2b

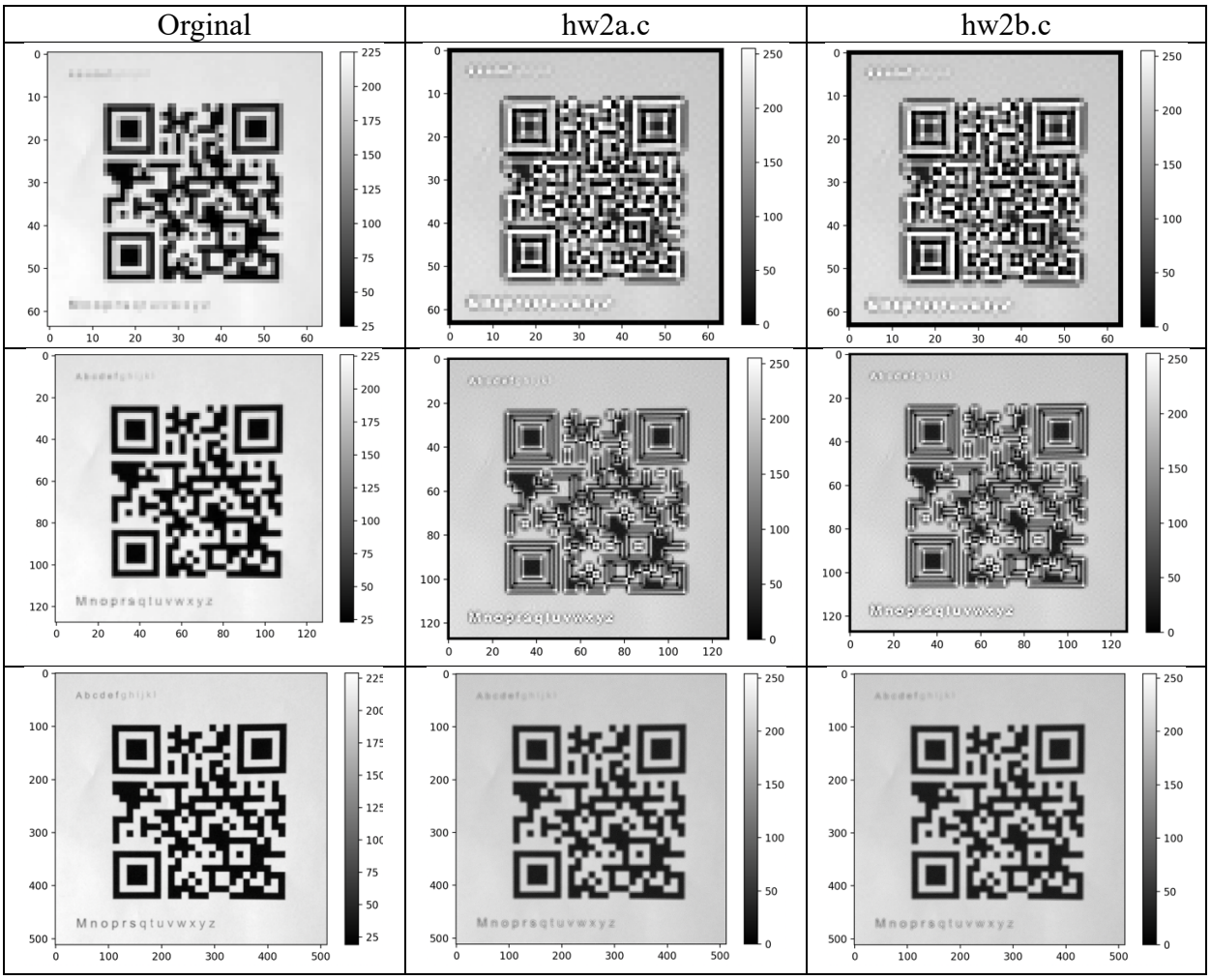

Table 2 python.py Outputs For Each .dat Files

The results showed that both hw2a and hw2b did a great job improving the clarity of the QR code images. We confirmed this by checking the output files using the python.py script. But the real story comes from the detailed metrics we got using gtime. These metrics gave us a closer look at how each program performs, especially when it comes to execution time and resource usage. To make it easier to see the differences, we visualized the results in graphs earlier in the report. These graphs highlight the trade-offs between speed and memory usage for each approach.

4. Discussion

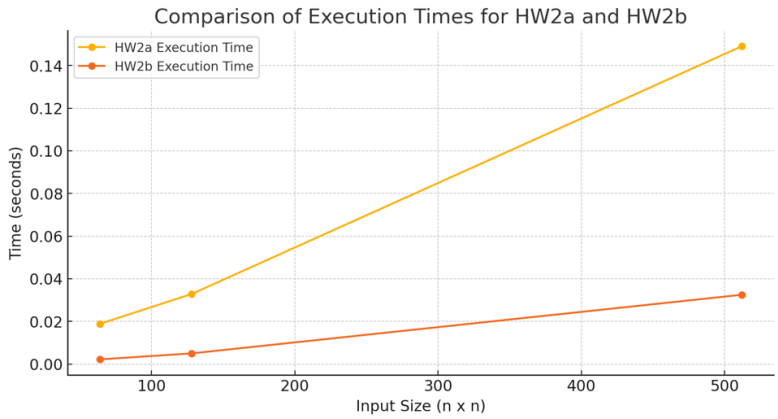

Figure 6 Comparison Of Execution Times Using a.txt & b.txt

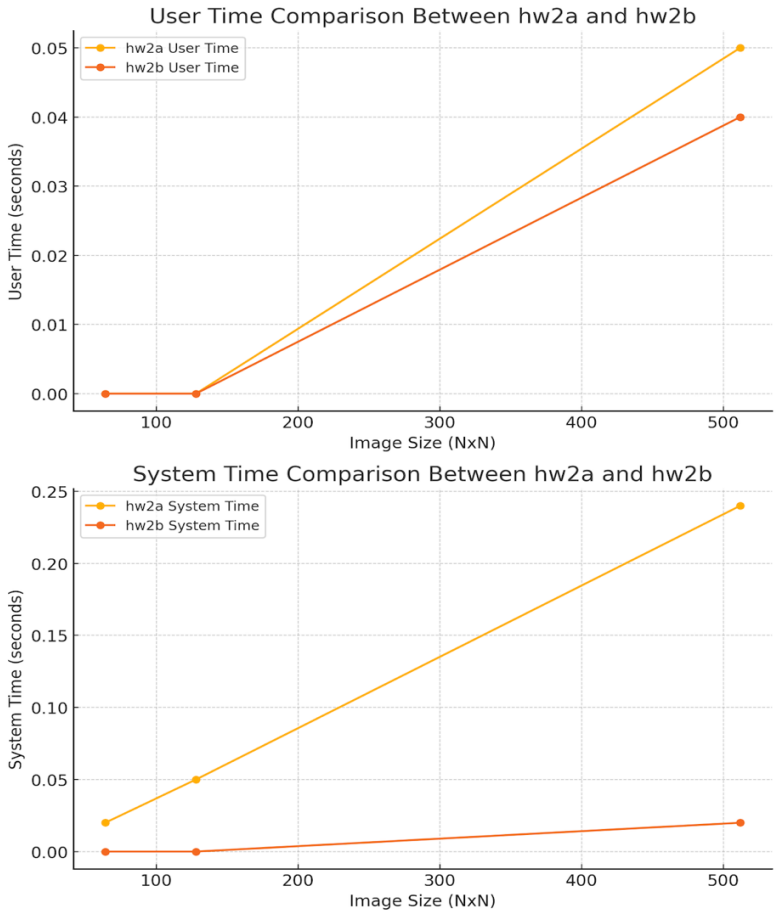

Figure 7 Comparison Of Execution Times Using gtime Outputs

When we compared the performance of the multi-process program hw2a to the multi-threaded program hw2b, some clear patterns emerged. As the input image size grew from 100x100 to 500x500 pixels, both programs saw increases in user time, system time, and total execution time. But here's the key difference—hw2b consistently outperformed hw2a across all three metrics. The biggest win for hw2b was in total execution time. As the image size got larger and the task became more complex, hw2b showed a noticeable edge in efficiency. This points to the strength of its threading model, which does a better job of managing CPU resources and running concurrent tasks. This advantage becomes even more important for larger datasets, where poor resource management can lead to major slowdowns.

5. Conclusion

The comparison between hw2a and hw2b highlights why multi-threaded programming is often the better choice for image processing—especially when dealing with larger datasets. hw2a does have the advantage of process isolation and better security, but that comes with extra overhead for process management. This leads to higher user and system times compared to hw2b. On the other hand, hw2b makes smart use of shared memory and the lightweight nature of threads, resulting in faster performance. This makes it a stronger option for tasks that demand high computational efficiency and scalability. These insights are especially valuable for fields like image processing and data analysis, where handling large datasets quickly and efficiently is a top priority.