Faculty of Engineering and Technology

Electrical and Computer Engineering Department

OPERATING SYSTEMS

ENCS3390

Project 1 Report

Student Name: Hala Mohammed

Student #: 1210312

Instructor: Dr.Abdel Salam Sayyad

Section: 1

Date:3/5/2024

Abstract:

This project focuses on implementing a program to calculate the average BMI (Body Mass Index) for a dataset using various techniques: a naive approach, a multiprocessing approach, and a multithreading approach. The performance of each method is evaluated by recording the time required to compute the BMI average. This evaluation involves experimenting with different numbers of child processes and threads, followed by a comparative analysis of the methods based on the observed results.

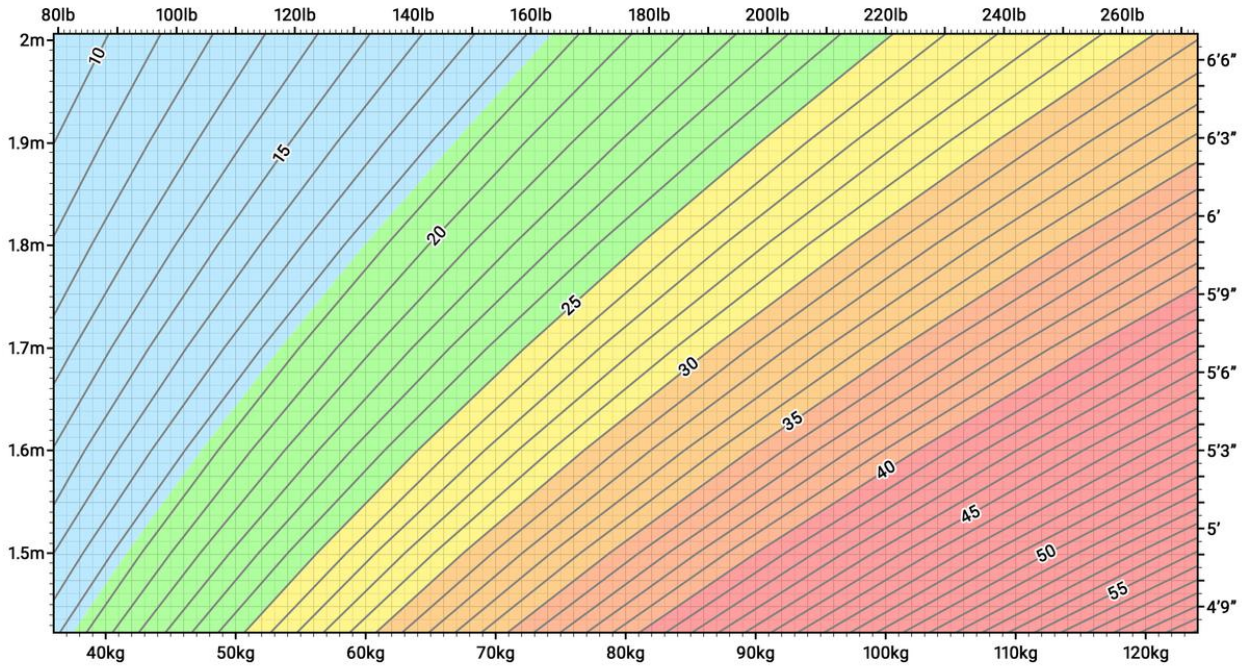
Table of Contents

Theory

Body mass index (BMI) is a measure of body fat based on height and weight, applicable to adult men and women. When dealing with a substantial amount of data, the calculation process naturally consumes a considerable amount of time.

In order to overcome this difficulty, we implemented the multiprocessing and multithreading methods, two cutting-edge techniques for programmatically calculating BMI. These techniques made a substantial contribution to cutting down on the total implementation time, which allowed the dataset to be processed effectively.

The software was created on the Linux operating system, which is recognized for its stability and performance, and was run using the C programming language, which is well-known for its effectiveness and low-level control. We were able to calculate BMI with optimal performance by utilizing these technologies.



BMI_chart

Process and Thread Management

Process

A program that is running at the moment is referred to as a process. For instance, the compiler produces binary code when we develop a program in a high-level programming language like C or C++ and then compile it. Programs can be defined as either the binary code or the original code. But the binary code doesn't become a process until it is really executed. A process is a "active" entity in this sense, whereas a program is a "passive" object. many instances are formed when we open a binary or.exe file more than once, which leads to the creation of many processes from a single program.

Put more simply, processes are just scheduled programs that are supposed to execute on the CPU after they are released from the ready state. A data structure called a Process Control Block (PCB) holds the concept of a process. Processes spawned by a parent process are known as child processes. Because it does not share memory with other processes, a process runs in isolation and takes longer to terminate. A process can be in one of the following states: suspended, terminated, waiting, ready, running, or new.

Process Management

Process management is the collection of methods and strategies that businesses use to plan, monitor, and control their business processes so they may accomplish their goals efficiently. It comprises figuring out the actions that need to be taken in order to finish a work, assessing the resources required at each stage, and choosing the best course of action.

Process management can be implemented to help firms save costs, increase customer happiness, improve operational efficiency, and comply with regulations. It comprises evaluating how well-performing current processes are working, spotting inefficiencies or bottlenecks, and making necessary adjustments to improve process flow.

Process mapping, process analysis, process improvement, process automation, and process control are just a few of the tools and methods that are included in process management. Organizations can improve productivity by doing away with inefficient procedures, streamlining their operations, and using various tools and strategies.

Inter Process Communication (IPC)

Inter-Process Communication (IPC) allows processes to coordinate and communicate with one another. Independent and cooperative processes are the two different categories of processes. Independent processes are not effected by other processes' execution, while collaborating processes could be by other executing processes' actions.

While it might seem that autonomous processes would perform better, there are a number of scenarios where cooperative processes can enhance computing efficiency, ease of use, and modularity. IPC facilitates communication and process synchronization. It serves as a conduit for coordination of activities and information exchange amongst processes. The way processes interact can be thought of as a form of cooperation. IPC facilitates communication and process synchronization. It serves as a conduit for coordination of activities and information exchange among processes. The way processes interact can be thought of as a form of cooperation. Processes can communicate with one another in two primary ways:

Shared Memory: By assigning a set amount of memory to each other, separate processes can read from and write to the same memory area

Message passing: By exchanging messages with one another, processes can coordinate their actions and share information.Direct message transmission between processes is also possible, as is message delivery via intermediary organizations.

Processes can successfully cooperate, share data, and coordinate their actions by using shared memory and message forwarding. The particular requirements and features of the associated processes determine which IPC technique is best.

Pipe

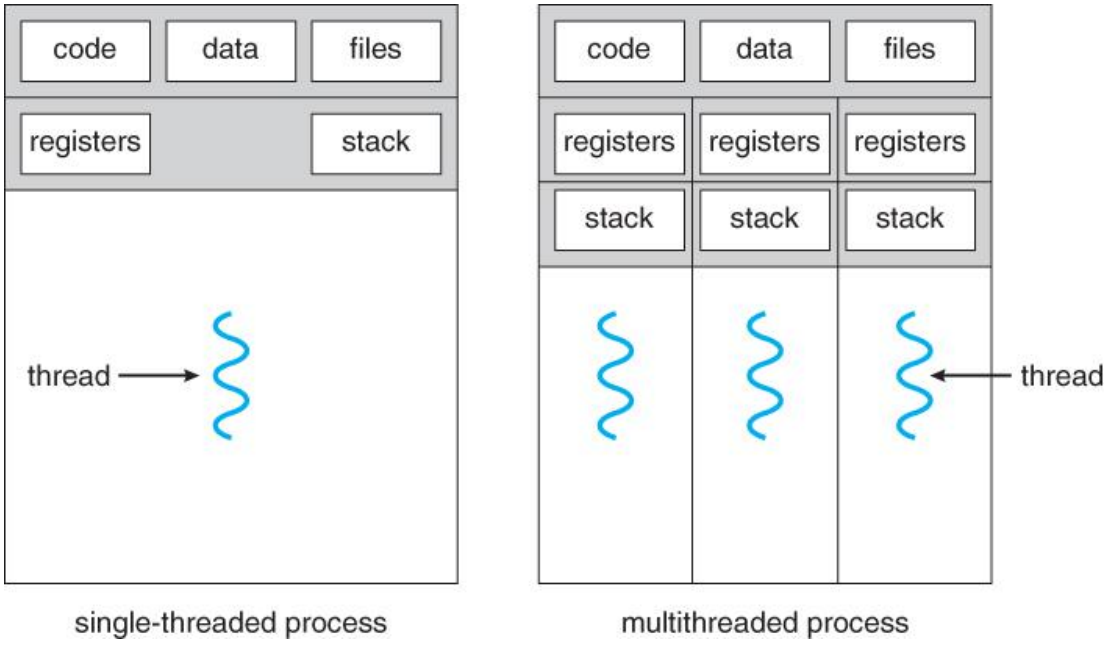
As an IPC, a pipe allows one-way data exchange between connected processes. Internally, it is controlled like a FIFO queue. The pipe system function can be used to create one, producing an input and output pipe file. Writing to the input end and reading from the output end are how processes exchange information. The size of each pipe is fixed when it is created, and if the pipe is full or the data is unavailable, processes may be suspended. In order for processes to communicate properly, the unused ends of the pipe must be closed. A pipe can only be accessed by one process at a time, guaranteeing exclusive access while communicating.

Threads

Multiple threads, or chunks of the process, can make up a process. There are three states for each thread: Blocked, Ready, and Running. Threads can be terminated faster than processes and do not operate in isolation.

Because they divide a process into many execution units, threads enable parallelism and are hence frequently referred to as lightweight processes. For example, in a web browser, a distinct thread can handle each open tab. Similar to this, programs like Microsoft Word use many threads to handle input and prepare text, among other things. There are other benefits to multithreading, which will be covered in more detail.

Operating systems use multithreading as a way to improve computer systems' responsiveness and performance. It permits the sharing of resources like the CPU, memory, and I/O devices among several threads, which are lightweight processes, all within a single process.



Threads in operating system

Procedure Discussion

The code provides three methods (naive, multithreading, and multiprocessing) for determining the average BMI from a CSV file. Every technique possesses unique benefits and compromises concerning intricacy and efficiency.

Data Structure Definition and Function Prototypes

The program's ability to work depends on the definition of important data structures (linked list nodes) and function prototypes in this section. Data from a CSV file is kept in a single linked list by the application. By establishing these structures and prototypes early on, the program improves code organization and execution and offers a clear framework for subsequent stages.

```
//define a node for a single linked list, store the data
struct Node {
    char gender[10];
    double height;
    double weight;
    struct Node *next;
};


// Function prototypes
void readTheData(struct Node **head);
double calculate_BMI(double weight, double height);
void NaiveApproach(struct Node **head);
void MultiprocessingApproach(struct Node **head, int numProcesses);
void *TheProcess(void *arg);
void MultithreadingApproach(struct Node **head, int numThreads);
void InsertAtEnd(struct Node **head, char gender[], double height, double weight);
struct Node *createNewNode(char gender[], double height, double weight);
void printList(struct Node* head);
```

Linked List Operations

Important functions for linked list operations, like insertion and printing, are defined in this section. These routines are essential to the program's creation and manipulation of linked lists. While the createNewNode function creates a new node with specified data, the InsertAtEnd function inserts nodes to the end of the list. To help visualize and validate the stored data, the printList function is utilized to show the contents of the linked list.

Reading Data from CSV File

A linked list must be created using CSV data. The `readTheData` function reads each line and pulls out important data like weight, height, and gender. By updating the linked list with this data, the application can effectively use it for subsequent analysis.

BMI Calculation

The body index calculation law is utilized by the Calculate_BMI function, which takes as inputs variables like height and weight to determine each person's BMI.

```
//calculate the Body Mass Index
double calculate_BMI(double weight, double height) {
    double heightInMeters = height / 100;
    double BMI = weight / pow(heightInMeters, 2);
    return BMI;
}
```

A Naive Approach

The Naive Approach works out the average BMI by iteratively going through every node in the linked list. It reads information from the CSV file, calculates each person's BMI, totals all of the BMI values, and divides by the total number of people. This method processes the data in a linear fashion, thus while it may seem simple, it might not be the best option for huge datasets.

Q: How I achieved the multiprocessing and multithreading requirements, i.e. The API and functions that I used.

Multithreading Approach

The linked list is split up into smaller pieces using the multithreading approach, and each piece is given its own thread for processing. This method makes use of POSIX Threads (pthread.h) to generate several threads, each in charge of carrying out a certain task simultaneously, such adding up the BMI values. This technique, which makes use of parallelism, can drastically cut down on the overall execution time, especially on systems with numerous cores. The main program waits for all threads to finish their calculations before combining the data to calculate the average BMI thanks to thread synchronization enabled by pthread_join(). In order to ensure thread safety while accessing shared global variables, such the total sum and count of BMI values, mutex locks are also used.

Multiprocessing Approach

I used the waitpid() function for synchronization and the fork() system call to create multiple processes in order to accomplish multiprocessing. The pipe() function made data flow between programs easier by facilitating communication between them. This method made use of the available CPU cores to enable the concurrent execution of code segments across many processes, improving performance and scalability. Separately, each procedure processed a segment of the linked list and added up the BMI values. Because this approach uses distinct memory areas for every activity, it can be applied to tasks that need to be isolated from other processes, like data processing in parallel. Multiprocessing considerably boosts performance by dividing the workload among several processors and carrying out tasks concurrently, resulting in the total average BMI.

Result Discussion

After running the program, we record the execution times for each method and try different numbers of processes and threads. The following table shows the readings that were recorded:

```
halamohammed@halamohammed-VirtualBox:~$ gcc OS_P1.c -o OS_P1 -pthread -lm
halamohammed@halamohammed-VirtualBox:~$ ./OS_P1
Operating System - Project#1
**********************************************************
Performing test using a naive approach:
Naive Approach: Average BMI = 37.77
Naive Approach Execution Time is: 0.021006 seconds

Performing test using a Multithreading Approach:
Using 1 thread(s):
Multithreading Approach: Average BMI = 37.77
Multithreading  Approach Execution Time: 0.000553 seconds
Using 2 thread(s):
Multithreading Approach: Average BMI = 37.82
Multithreading  Approach Execution Time: 0.000225 seconds
Using 3 thread(s):
Multithreading Approach: Average BMI = 37.87
Multithreading  Approach Execution Time: 0.000259 seconds
Using 4 thread(s):
Multithreading Approach: Average BMI = 37.93
Multithreading  Approach Execution Time: 0.000316 seconds

Performing test using a Multiprocessing Approach:
Using 1 core(s):
Multiprocessing Approach: Average BMI = 37.77
Multiprocessing Approach Execution Time: 0.003702 seconds
Using 2 core(s):
Multiprocessing Approach: Average BMI = 37.77
Multiprocessing Approach Execution Time: 0.005886 seconds
Using 3 core(s):
Multiprocessing Approach: Average BMI = 37.64
Multiprocessing Approach Execution Time: 0.004698 seconds
Using 4 core(s):
Multiprocessing Approach: Average BMI = 37.59
Multiprocessing Approach Execution Time: 0.002935 seconds
```

Naive Approach:

1. Average BMI: 37.77
2. Execution Time: 0.021006 seconds

Multithreading Approach:

1. Using 1 thread(s): Average BMI = 37.77, Execution Time = 0.000553 seconds
2. Using 2 thread(s): Average BMI = 37.82, Execution Time = 0.000225 seconds
3. Using 3 thread(s): Average BMI = 37.87, Execution Time = 0.000259 seconds
4. Using 4 thread(s): Average BMI = 37.93, Execution Time = 0.000316 seconds

Multiprocessing Approach:

1. Using 1 core(s): Average BMI = 37.77, Execution Time = 0.003702seconds
2. Using 2 core(s): Average BMI = 37.77, Execution Time = 0.005886 seconds
3. Using 3 core(s): Average BMI = 37.64, Execution Time = 0.004698 seconds
4. Using 4 core(s): Average BMI = 37.59, Execution Time = 0.002935 seconds

A table that compares the performance of the 3 approaches:

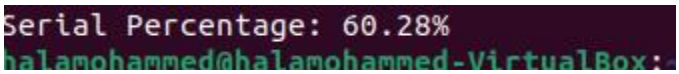
Execution Time / BMI Value	Naive Approach	Multithreading Approach	Multiprocessing Approach
Execution Time Using 1 core in Multiprocessing Approach And 1 thread in Multithreading Approach	**Average BMI: 37.77** **Execution Time: 0.021006 seconds**	**0.000553 seconds**	**0.003702 seconds**
Avg BMI Value		**37.77**	**37.77**
Execution Time Using 2 cores in Multiprocessing Approach And 2 threads in Multithreading Approach		**0.000225 seconds**	**0.005886 seconds**
Avg BMI Value		**37.82**	**37.77**
Execution Time Using 3 cores in Multiprocessing Approach And 3 threads in Multithreading Approach		**0.000259 seconds**	**0.004698 seconds**
Avg BMI Value		**37.87**	**37.64**
Execution Time Using 4 cores in Multiprocessing Approach And 4 threads in Multithreading Approach		**0.000316seconds**	**0.002935 seconds**
Avg BMI Value		**37.93**	**37.59**

Based on the results obtained and presented above, we conclude the following:

- The execution time increases as the number of processes or threads increases, indicating an increase in parallelism overhead.
- Beyond a certain point, increasing the number of processes or threads yields diminishing returns in performance gain. While an initial increase in performance is noticeable and beneficial, further additions result in diminishing improvements

An analysis according to Amdahl's law

We begin by timing the entire program in order to determine the serial percent. Next, we use the straightforward timing method to display the serial time of the program. We compute a basic arithmetic using these times: divide the time in the simple method by the total time and times by 100%. This illustrates the percentage of serial time in the program.



So the portion of the code that can be parallelized is 39.72% (1 - 60.28%)

the maximum speedup according to the available number of cores:

the maximum speedup S is given by:

$$S = 1 / (1 - p + p/N)$$

For different numbers of cores:

For 1 core: S= 1

For 2 cores: S= 1.1094

For 3 cores: S=1.0703

For 4 cores: S= 1.0524

The actual speedup values based on the result in execution times:

For 1 core: S=1

For 2 cores: S=0.629

For 3 cores: S=0.788

For 4 cores: S=1.262

Based on Amdahl's law, we find that the actual speedup values are smaller than the ones that were expected.

This could be the result of a number of functions, including software efficiency limits, competition for shared resources, or overhead generated by parallelization.

What is the proper/optimal number of child processes or threads?

- Using **two threads** for the Multithreading Approach results in the shortest execution time while keeping an acceptable average BMI
- When utilizing **4 cores** not fewer, the Multiprocessing Approach gives the lowest execution time and an almost higher average BMI.
- Choosing the optimal number of processes and threads depends on the characteristics of system and factors such as the nature of the workload, available hardware resources, and overall increased efficiency, which means it will vary from one device to another. Where at a certain point when the number of processes or threads increases, the additional gain in performance becomes gradually smaller or less significant. Therefore, it is necessary to find a balance in the number of processes or threads to improve performance without introducing unnecessary burdens or reducing the return on the effort invested in parallelization.

Conclusion

The aim of this project was to calculate BMI using various techniques, include naive approach, a multiprocessing approach, and a multithreading approach The performance of each method was assessed by measuring the time to calculate the average BMI (Body Mass Index) for a dataset across different numbers of child processes and threads. The comparison of these methods revealed insights into their respective efficiencies. The results underscore the importance of selecting an appropriate parallelization strategy, considering factors such as the number of child processes or threads, to optimize the performance of calculation.

References

[1]: https://www.geeksforgeeks.org/program-to-calculate-bmi/

[2]: https://www.geeksforgeeks.org/introduction-of-process-management/

[3]: https://www.geeksforgeeks.org/difference-between-process-and-thread/

[4]: https://www.geeksforgeeks.org/inter-process-communication-ipc/

[5]: https://www.geeksforgeeks.org/ipc-technique-pipes/

[6]: https://www.geeksforgeeks.org/thread-in-operating-system/

[7]:https://youtube.com/playlist?list=PLfqABt5AS4FkW5mOn2Tn9ZZLLDwA3kZUY&si=ItrbMaP_sZIQPk0v

[8]:https://youtube.com/playlist?list=PLfqABt5AS4FmuQf70psXrsMLEDQXNkLq2&si=JBeDN6mY7E_gZ_TF