Technical Report

Version 1.0

Eric Tsang, A00841554, 6D

|  |
| --- |
| Table of Contents [Table of Contents 1](#_Toc440760429)  [1 Summary 2](#_Toc440760430)  [2 Introduction 2](#_Toc440760431)  [3 Theory 2](#_Toc440760432)  [4 Experimental Apparatus 3](#_Toc440760433)  [5 Procedure 4](#_Toc440760434)  [5.1 Execution Time Efficiency 4](#_Toc440760435)  [5.2 Initialization Time Efficiency 4](#_Toc440760436)  [5.3 Memory Usage 4](#_Toc440760437)  [5.4 Heap Memory Usage 4](#_Toc440760438)  [5.5 Processor Utilization 5](#_Toc440760439)  [6 Results & Discussion 6](#_Toc440760440)  [6.1 Execution Time Efficiency 6](#_Toc440760441)  [6.2 Initialization Time Efficiency 7](#_Toc440760442)  [6.3 Memory Usage 8](#_Toc440760443)  [6.4 Heap Memory Usage 9](#_Toc440760444)  [6.4.1 Threaded Version 9](#_Toc440760445)  [6.4.2 Process Version 10](#_Toc440760446)  [6.5 Processor Utilization 12](#_Toc440760447)  [7 Conclusion 13](#_Toc440760448) |

# Summary

In Linux distributions, there are 2 ways to achieve concurrency for an application: using threads, or using processes. However, threads and processes operate and perform differently. This report discusses the strengths and weaknesses of both threads and processes.

Threads initialize faster, and take up less memory when compared to a process. In theory, using threads should have a slight positive impact on performance. In practice, this is often not the case because threads share the same heap memory with other threads, and access to the heap often needs to be synchronized with them which can be extremely time consuming.

Processes take on average 4 times longer than threads to initialize since they take up more memory. Communication between processes is often more complicated than communication between threads since processes don't naturally share the same memory like threads do and thus must communicate using IPC mechanisms. Using IPC mechanisms ensures proper and efficient synchronization between processes. When processes are executed simultaneously, they can easily out-perform threads because each process has exclusive access to their own heap memory and other resources, so access to them does not need to be subject to the time-consuming synchronization that threads are subject to.

# Introduction

This report compares the strengths and weaknesses of threads with processes on the Ubuntu 14.04 operating system in a multi-core environment. The threads and processes will be compared in terms of time efficiency, impact on memory usage and how well each takes advantage of processors as a resource in a multi-core environment.

# Theory

In the Ubuntu operating system, threads and processes are used to achieve concurrency in applications. and possibly performance boosts in multi-processor systems.

Multiple threads can exist in a single process, all of which can be executing simultaneously on the system. Implications of multi-threaded, single-process applications include:

* All threads share the same memory space, and therefore share heap memory which simplifies inter-thread-communication.
* Thread performance can be bottlenecked by thread synchronization when mutual exclusion is necessary. For example, heap operations are often mutually exclusive, like when allocating and de-allocating heap memory.
* Less overhead when sharing system resources between threads; If the process has acquired a system resource, then all threads of the process will also have access to the same system resource.
* Faster initialization times; unlike processes, threads do not have any private heap memory allocated to them. Only a stack, and thread control block; allocating multiple threads should be faster and more memory efficient than allocating the same number of processes.
* Processors spend less time switching between threads of a process because threads are smaller than processes.

Multiple processes can be executing simultaneously on the same system. They can communicate and work together through IPC (Inter Process Communication) mechanisms. Implications of multi-process applications include:

* Complex inter process communication. Each process is allocated memory that only they can access; in order for processes to communicate with one another, they must use inter process communication mechanisms to do so. This can add extra complexities to code.
* Faster heap operations. Since processes do not share heap memory, when they perform heap memory allocation or de-allocation operations, they are not hindered by synchronization.
* Slower initialization times. Processes have many more properties compared to threads, such as memory allocated for the program, heap memory, the stack, and the process control block. Because processes are larger, they take more time to initialize compared to threads.

# Experimental Apparatus

Two custom programs were created to help compare processes with threads. One uses threads, and the other uses processes. Both programs:

* Perform identical tasks. The task performed is mostly mathematically intensive and somewhat IO intensive. The purpose of this experiment is to compare the performance of threads and processes. making the tasks too IO intensive would bottleneck their performance, as IO tasks are long, non-computationally intensive tasks.
* Compute all the factors of a user specified value.
* Write progress, factors, and execution statistics to std out and a log file.
* Spawn a user specified number of workers to perform the mathematically intensive tasks.

The tools used in to measure the two programs' impact on memory usage on the system is valgrind version 3.10.1.

The System Monitor application will be used to measure how well threads or processes take advantage of CPU resources.

# Procedure

This section describes the procedures followed to carry out the various experiments used to measure the different performance aspects of threads and processes.

## Execution Time Efficiency

This procedure is used to run the program, and measure its execution time efficiency:

1. Open a Terminal. Navigate to ./source/ (relative to the project's root directory)
2. Make a clean compile of both the threaded and process version of the program.

$ make clean

$ make Threads-Main

$ make Processes-Main

1. Execute the programs with the parameters shown below.

$ Threads-Main.out 1000000 /dev/null 4

$ Processes-Main.out 1000000 /dev/null 4

1. Repeat ten times and record the execution time results outputted immediately before the program terminates.

## Initialization Time Efficiency

This procedure is used to run the program, and measure its initialization time efficiency:

1. Open a Terminal. Navigate to ./source/ (relative to the project's root directory)
2. Make a clean compile of both the threaded and process version of the program.

$ make clean

$ make Threads-Main

$ make Processes-Main

1. Execute the programs with the parameters shown below.

$ Threads-Main.out 1 /dev/null 20

$ Processes-Main.out 1 /dev/null 20

1. Repeat ten times and record the execution time results outputted immediately before the program terminates.

## Memory Usage

This procedure is used to run the program, and measure its overall time efficiency:

1. Open a Terminal. Navigate to ./source/ (relative to the project's root directory)
2. Make a clean compile of both the threaded and process version of the program.

$ make clean

$ make Threads-Main

$ make Processes-Main

1. Execute the programs with the parameters shown below.

$ ./Threads-Main.out 100000000 /dev/null 4

$ ./Processes-Main.out 100000000 /dev/null 4

1. Before the program returns, open the System Monitor, and check the Memory usage for each of the relevant processes in the Processes tab.

## Heap Memory Usage

This procedure is used to run the program, and measure its overall time efficiency:

1. Open a Terminal. Navigate to ./source/ (relative to the project's root directory)
2. Make a clean compile of both the threaded and process version of the program.

$ make clean

$ make Threads-Main

$ make Processes-Main

1. Execute the programs using valgrind with the parameters shown below. This will generate one massif.out file when running the threaded program, and 5 massif.out files for the process version.

$ valgrind --tool=massif --time-unit=B ./Threads-Main.out 1000000 /dev/null 4

$ valgrind --tool=massif --time-unit=B ./Processes-Main.out 100000 /dev/null 4

1. Run ms\_print for each of the massif.out files that were generated in the previous step, to analyze its heap usage.

$ ms\_print massif.out.*[processId]*

## Processor Utilization

This procedure is used to run the program, and measure its execution time efficiency:

1. Open a Terminal. Navigate to ./source/ (relative to the project's root directory)
2. Make a clean compile of both the threaded and process version of the program.

$ make clean

$ make Threads-Main

$ make Processes-Main

1. Execute the programs with the parameters shown below.

$ Threads-Main.out 100000000 /dev/null 4

$ Processes-Main.out 100000000 /dev/null 4

1. Observe the CPU History in the Resources tab of the System Monitor. Run the program with different values for the num workers parameter. Observe how it affects CPU usage.

# Results & Discussion

The following are results of the tests.

## Execution Time Efficiency

10 tests were performed. Their results were recorded. In general, the process version of the program performed better than the threaded version in this area of the test. The following graph illustrates the test results:

The following graph shows the average runtime of all attempts of the processes version of the program, versus the average runtime of all attempts of the threaded version of the program:

As shown above, processes execute tasks more efficiently when compared to threads. Since threads of the same process share the same address space, and heap memory, some operations must be serialized, and access to shared data structures must be mutually exclusive to avoid memory corruption. Poorly engineered mutual exclusion to shared data structures can cause significant delays during execution.

## Initialization Time Efficiency

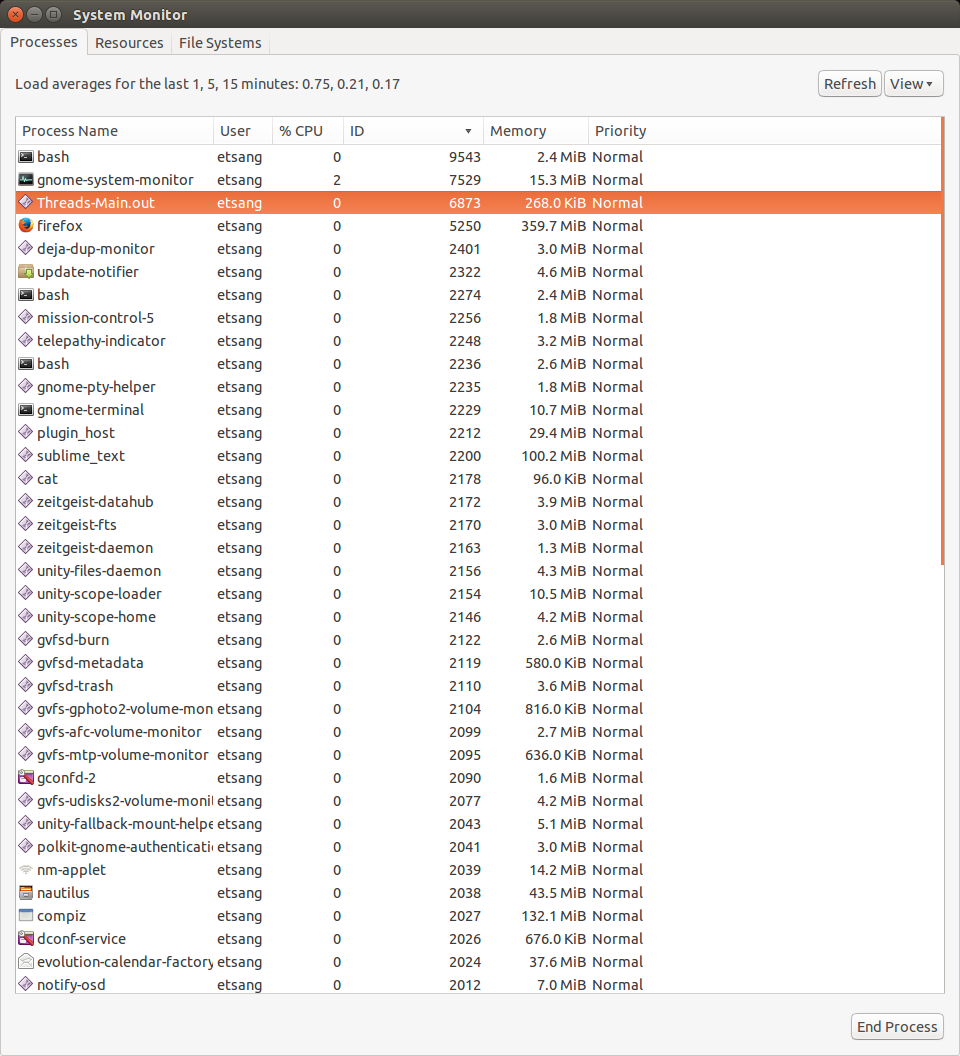
10 tests were performed. Their results were recorded. In general, the threaded version of the program performed better than the process version in this area of the test. The following graph illustrates the test results:

The following graph shows the average runtime of all attempts of the processes version of the program, versus the average runtime of all attempts of the threaded version of the program:

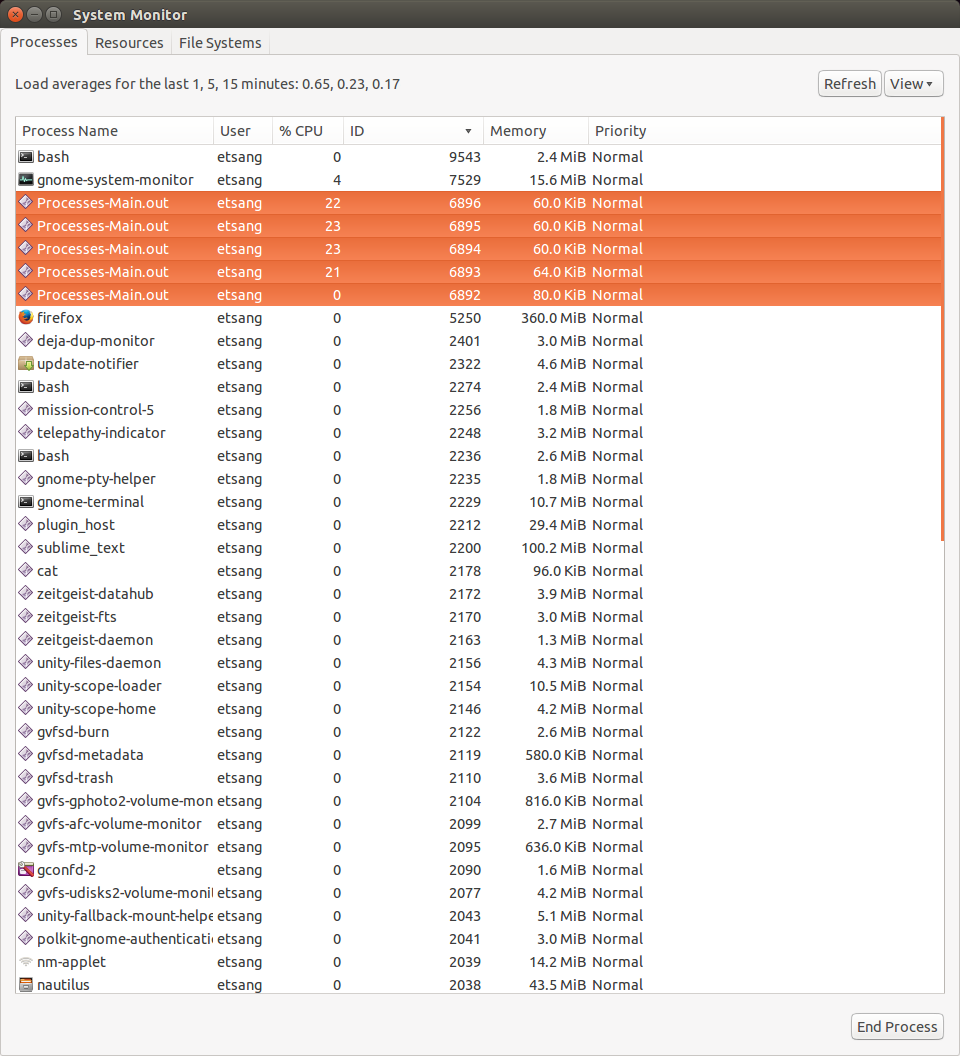
As shown above, threads performed better in this test compared to processes. Processes take 4 times longer than threads to initialize. Unlike processes that have a heap, stack, code, and a process control block, Threads only have a stack, and thread control block; they are simpler to create, and more memory efficient compared to processes, so initializing threads is faster than initializing processes.

## Memory Usage

The following is a screenshot of the relevant entry in the System Monitor of the threaded version of the application:



The following is a screenshot of the relevant entries in the System Monitor of the process version of the application:



As shown above, each individual process of the process version of the application is smaller than the process of the threaded version of the application. However, the total memory footprint of the threaded version of the application (268KB) is smaller than the total memory footprint of the process version of the application (324KB).

The memory footprint of a process is larger than a thread's. A process at minimum consists of a heap, stack, code, and process control block, while a thread is created within a process, and consists of only a stack, and thread control block which is physically smaller than a process.

## Heap Memory Usage

The following are graphs created by valgrind followed by explanations of why the graph is shaped the way it is.

### Threaded Version

--------------------------------------------------------------------------------

Command: ./Threads-Main.out 1000000 /dev/null 4

Massif arguments: --time-unit=B

ms\_print arguments: massif.out.4543

--------------------------------------------------------------------------------

KB

6.594^#

|#

|#

|# : @:: :: :::: :

|# :::: ::::: :: ::::::: @:::::@: ::: ::::::: :: :::::::::@:

|# : : : ::::: :: :: ::::::::@:: ::@: ::: ::: ::::: ::::: : :::::@::

|#::: : ::::: ::::: :: :: :::: :: @:: ::@: ::: : : ::::: ::: : : :::::@::

|#: : : ::: : ::::: :: :: :::: :: @:: ::@: ::: : : ::::: ::: : : :::::@::

|#: : : ::: : ::::: :: :: :::: :: @:: ::@: ::: : : ::::: ::: : : :::::@::

|#: : : ::: : ::::: :: :: :::: :: @:: ::@: ::: : : ::::: ::: : : :::::@::

|#: : : ::: : ::::: :: :: :::: :: @:: ::@: ::: : : ::::: ::: : : :::::@::

|#: : : ::: : ::::: :: :: :::: :: @:: ::@: ::: : : ::::: ::: : : :::::@::

|#: : : ::: : ::::: :: :: :::: :: @:: ::@: ::: : : ::::: ::: : : :::::@::

|#: : : ::: : ::::: :: :: :::: :: @:: ::@: ::: : : ::::: ::: : : :::::@::

|#: : : ::: : ::::: :: :: :::: :: @:: ::@: ::: : : ::::: ::: : : :::::@::

|#: : : ::: : ::::: :: :: :::: :: @:: ::@: ::: : : ::::: ::: : : :::::@::

|#: : : ::: : ::::: :: :: :::: :: @:: ::@: ::: : : ::::: ::: : : :::::@::

|#: : : ::: : ::::: :: :: :::: :: @:: ::@: ::: : : ::::: ::: : : :::::@::

|#: : : ::: : ::::: :: :: :::: :: @:: ::@: ::: : : ::::: ::: : : :::::@::

|#: : : ::: : ::::: :: :: :::: :: @:: ::@: ::: : : ::::: ::: : : :::::@::

0 +----------------------------------------------------------------------->MB

0 47.08

As shown above, the heap usage of the threaded version of the application is steady throughout the execution of the program.

The program was designed to create a maximum of n pending tasks. Once the number of pending tasks fell below the maximum, the main thread would spin up, and create more pending tasks until there is no more to create. This explains the steady use of heap memory throughout the application.

As the program executes, calculation results will be saved in the heap. This explains why heap memory usage is slightly higher in the latter half of the program's execution.

### Process Version

--------------------------------------------------------------------------------

Command: ./Processes-Main.out 1000000 /dev/null 4

Massif arguments: --time-unit=B

ms\_print arguments: massif.out.4628

--------------------------------------------------------------------------------

KB

4.617^ #

| @ ::::::@@::::@::#::

| : ::@:: ::::@@::: @::#: ::

| ::::: @:: ::::@@::: @::#: :::

| ::: : @:: ::::@@::: @::#: :::::

| ::: : @:: ::::@@::: @::#: :::::::

| ::: : @:: ::::@@::: @::#: :::::: :@

| ::::: : @:: ::::@@::: @::#: :::::: :@::

| : ::: : @:: ::::@@::: @::#: :::::: :@::::

| : ::: : @:: ::::@@::: @::#: :::::: :@::::::

| @@: ::: : @:: ::::@@::: @::#: :::::: :@::::::@::

| @ : ::: : @:: ::::@@::: @::#: :::::: :@::::::@:

| : :: : : : : :@ : ::: : @:: ::::@@::: @::#: :::::: :@::::::@:

| ::::::: :@:::::::::@ : ::: : @:: ::::@@::: @::#: :::::: :@::::::@: :

| :: :::: :@:::::::::@ : ::: : @:: ::::@@::: @::#: :::::: :@::::::@: :

| ::: :::: :@:::::::::@ : ::: : @:: ::::@@::: @::#: :::::: :@::::::@: :

| ::: :::: :@:::::::::@ : ::: : @:: ::::@@::: @::#: :::::: :@::::::@: ::

| ::: :::: :@:::::::::@ : ::: : @:: ::::@@::: @::#: :::::: :@::::::@: ::

|:::: :::: :@:::::::::@ : ::: : @:: ::::@@::: @::#: :::::: :@::::::@: :::

|:::: :::: :@:::::::::@ : ::: : @:: ::::@@::: @::#: :::::: :@::::::@: :::

0 +----------------------------------------------------------------------->KB

0 42.93

The above heap usage graph is of the parent process of the process version of the application.

The program is designed to have 3 phases: setup, work, then cleanup. Each of these three phases can be clearly identified in the graph above:

1. The setup phase of the program is between time zero until and the first column of @ signs. The parent process allocates shared memory, IPC, and data structures to prepare for the work phase of the program.
2. The work phase of the program is between the first column of @ signs until the column of # signs. The parent is generating tasks for child processes to execute, and is receiving calculation results from them via IPC, which increases heap memory usage.
3. The cleanup phase is between the column of # signs until program termination. All child processes of the parent have terminated. The parent prints all received results. As the results are written to stdout and the log file, they are deleted. Once all results have been deleted, the parent proceeds to delete any remaining IPC and data structures allocated on the heap.

--------------------------------------------------------------------------------

Command: ./Processes-Main.out 1000000 /dev/null 4

Massif arguments: --time-unit=B

ms\_print arguments: massif.out.4630

--------------------------------------------------------------------------------

KB

1.531^ ##

| : # :

| ::# :: ::::::: :

| ::# :::::: ::: :::::::::::::::::::::::@::::@::::::::::@::::::@::::::@::

| ::# :::::: ::: :: ::::::::::::::::::::@::::@::::::::::@::::::@::::::@::

| ::# :::::: ::: :: ::::::::::::::::::::@::::@::::::::::@::::::@::::::@::

| ::# :::::: ::: :: ::::::::::::::::::::@::::@::::::::::@::::::@::::::@::

| ::# :::::: ::: :: ::::::::::::::::::::@::::@::::::::::@::::::@::::::@::

| ::# :::::: ::: :: ::::::::::::::::::::@::::@::::::::::@::::::@::::::@::

| ::# :::::: ::: :: ::::::::::::::::::::@::::@::::::::::@::::::@::::::@::

| ::# :::::: ::: :: ::::::::::::::::::::@::::@::::::::::@::::::@::::::@::

| ::# :::::: ::: :: ::::::::::::::::::::@::::@::::::::::@::::::@::::::@::

| ::# :::::: ::: :: ::::::::::::::::::::@::::@::::::::::@::::::@::::::@::

| ::# :::::: ::: :: ::::::::::::::::::::@::::@::::::::::@::::::@::::::@::

| ::# :::::: ::: :: ::::::::::::::::::::@::::@::::::::::@::::::@::::::@::

| ::# :::::: ::: :: ::::::::::::::::::::@::::@::::::::::@::::::@::::::@::

| ::# :::::: ::: :: ::::::::::::::::::::@::::@::::::::::@::::::@::::::@::

| ::# :::::: ::: :: ::::::::::::::::::::@::::@::::::::::@::::::@::::::@::

| ::# :::::: ::: :: ::::::::::::::::::::@::::@::::::::::@::::::@::::::@::

| ::# :::::: ::: :: ::::::::::::::::::::@::::@::::::::::@::::::@::::::@::

0 +----------------------------------------------------------------------->MB

0 18.28

The above graph was generated for one of the 4 child processes of the process version of the application. All of the other graphs generated for the worker processes are very similar to the one above and thus have not been included in the report.

The heap memory usage of the worker processes is quite featureless and steady since worker processes spend most of their time in the same states and only work on one task at a time, so only one task is allocated on the heap at a time. This creates a steady heap memory usage graph.

## Processor Utilization

The graphs below compare how threads take advantage of CPU resources with how processes take advantage of CPU resources.

|  |  |  |
| --- | --- | --- |
| # of Workers | Processes | Threads |
| 1 | C:\Users\Eric Tsang\AppData\Local\Microsoft\Windows\INetCache\Content.Word\processes x 1.png |  |
| 2 |  | C:\Users\Eric Tsang\AppData\Local\Microsoft\Windows\INetCache\Content.Word\threads x 2.png |
| 4 | C:\Users\Eric Tsang\AppData\Local\Microsoft\Windows\INetCache\Content.Word\processes x 4.png | C:\Users\Eric Tsang\AppData\Local\Microsoft\Windows\INetCache\Content.Word\Threads x 4.png |

As shown above, when there are more workers, the application, regardless if it is implemented using threads or processes till utilize multiple processors to concurrently perform tasks. There is no significant difference between threads and processes in how they utilize the extra processors of a multi-core system.

# Conclusion

Based on the results from section 6.2 and 6.3, threads have a smaller memory footprint and also initialize faster than processes.

The results from sections 6.1, and 6.4 are based more on how the program was designed and implemented rather than how threads and processes perform.

The results from section 6.5 show that both processes and POSIX threads are kernel level entities, so using multiple threads or processes enables the program to make use of more CPU resources when available.

In theory, execution time efficiency of threads should be insignificantly faster than the execution efficiency of processes since they are smaller, and require less time to perform a context switch from one thread to another. However, in practice, the speed of multi-threaded applications are often impeded by other factors especially thread synchronization like needing to serialize all allocation and de-allocation operations on the heap since it is shared by all the threads of the process.