College of Computer and Information Sciences (CCIS)

Term 252

CS330: Introduction to Operating Systems

Parallel Performance Evaluation Using Process-Based Execution in Linux

Instructor: Dr. Souad Larabi

Team Members:

SHOUG FAWAZ ABDULLAH ALOMRAN - 223410392

LAYAN HESHAM RASHED BINDAYEL - 223410104

ARYAM YAHYA ALMANSOUR - 223410070

DANAH ABDULAZIZ ABDULMOHSEN ALTUWAIJRI - 223410151

RANA ABDULMALIK HAMAD ALFARIS - 223410567

OUTLINE

1.0 Introduction

This project tests how using multiple processes affects the performance of matrix multiplication in Linux. We use the fork() system call to create child processes that work on different parts of the computation at the same time.

The program multiplies two matrices by dividing the work among multiple processes. We measure the execution time to see how performance changes when we use different numbers of processes and different matrix sizes.

The goal of Phase 1 is to set up a Linux environment, run the program with different settings, and collect data on how long it takes to complete.

2.0 Objectives

The objectives of this phase are:

- Set up Ubuntu Linux in a virtual machine
- Compile and run the provided C program
- Test with different matrix sizes: 1200, 1800, and 2400
- Test with different numbers of processes: 1 and 4
- Measure and compare execution times

3.0 Experimental Setup

3.1 System Environment

All experiments were conducted inside a Linux virtual machine to ensure a consistent and reproducible execution environment. The virtual machine was created using UTM on macOS and configured to use Apple's virtualization framework.

The guest operating system used was Ubuntu 24.04.3 LTS (ARM64). The virtual machine was allocated 4 GB of memory and 4 CPU cores, which provided sufficient resources for parallel execution.

Storage capacity was configured to ensure sufficient disk space for the operating system, source code, and experimental outputs.

System verification commands (ex: uname -a, free -h, lsb_release -a) were used to confirm the CPU architecture (aarch64), available memory, core count, and OS version, ensuring a stable and controlled environment.

System verification commands were used to confirm:

- CPU architecture (aarch64)
- Number of available CPU cores
- Memory allocation
- Operating system version

This configuration ensured that all experiments were performed under stable and controlled conditions.

The following steps illustrate the creation and configuration of the Linux virtual machine.

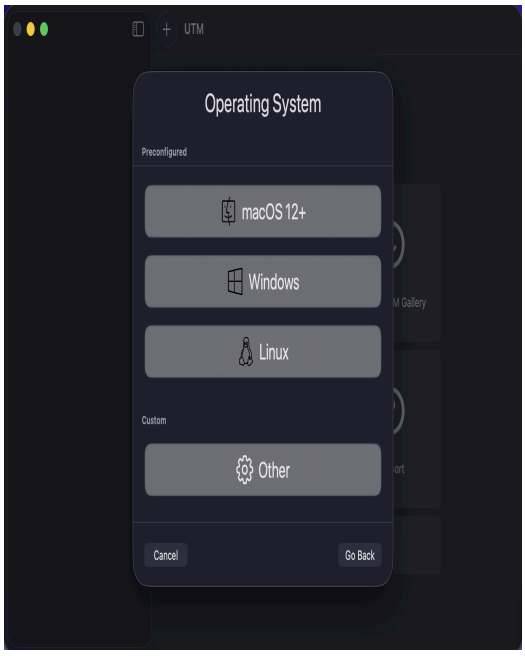


Figure 4: Selecting Linux as the guest operating system.

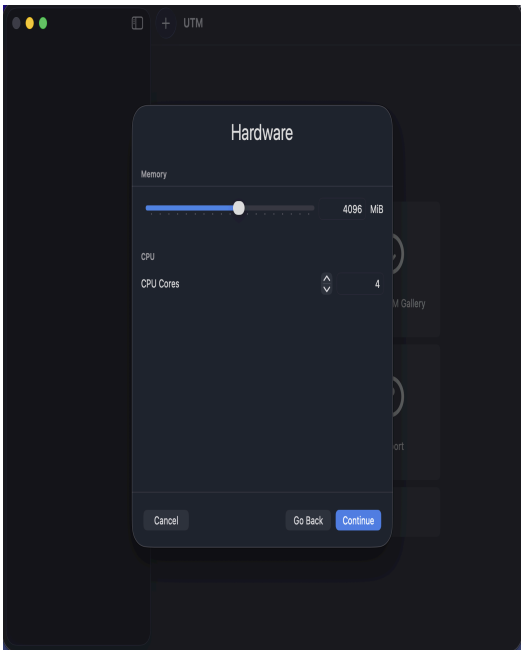


Figure 5: Allocated memory (4 GB) and CPU cores (4) for the virtual machine.

The virtual machine hardware resources were configured before installation.

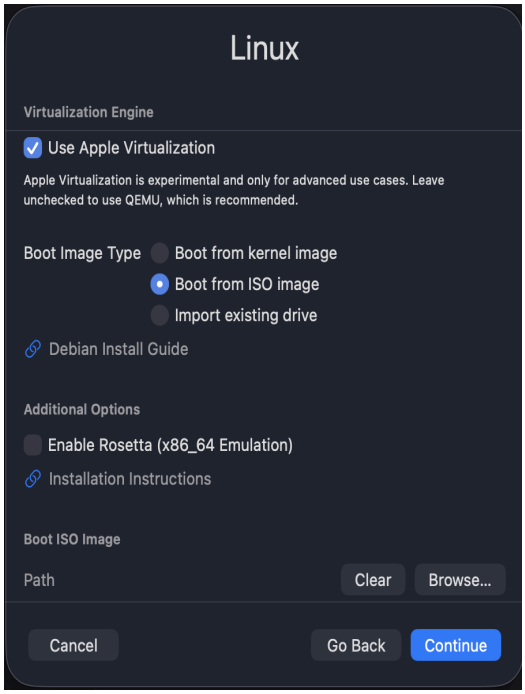


Figure 6: Using Apple Virtualization framework for ARM-based execution.

3.2 Software Tools

We installed the following tools in Ubuntu:

- GCC compiler (version 11.4.0)
- Build-essential package

Installation command: sudo apt install build-essential

Compilation command: gcc Wall matrix_fork.c -o matrix_fork The -Wall flag enables all compiler warnings to help catch errors.

3.3 Program Description

The provided C program performs matrix multiplication using process-based parallelism. Two input matrices are initialized with random values, and the computation of the result matrix is divided among multiple child processes.

Each child process is responsible for computing a subset of matrix rows. The parent process waits for all child processes to complete before measuring the total execution time.

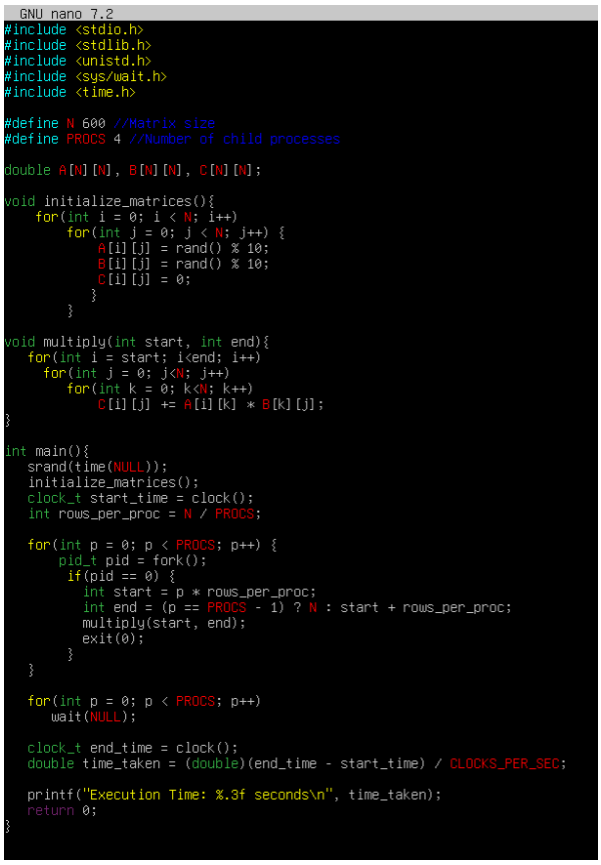
The program uses:

- fork() for process creation
- wait() for synchronization
- clock() to measure execution time

No structural changes were made to the program. Only configurable parameters such as matrix size and number of processes were modified during experimentation.

```
cs330-project@cs330-project:~/os-project/src$ gcc -Wall matrix_fork.c -o matrix_fork
cs330-project@cs330-project:~/os-project/src$ ./matrix_fork
Execution Time: 0.001 seconds
cs330-project@cs330-project:~/os-project/src$
```

Figure 7: Program execution and reported execution time.



```
  GNU nano 7.2
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/wait.h>
#include <time.h>

#define N 600 //Matrix size
#define PROCS 4 //Number of child processes

double A[N][N], B[N][N], C[N][N];

void initialize_matrices(){
    for(int i = 0; i < N; i++)
        for(int j = 0; j < N; j++) {
            A[i][j] = rand() % 10;
            B[i][j] = rand() % 10;
            C[i][j] = 0;
           }
        }

void multiply(int start, int end){
   for(int i = start; i<end; i++)
     for(int j = 0; j<N; j++)
        for(int k = 0; k<N; k++)
            C[i][j] += A[i][k] * B[k][j];
}

int main(){
   srand(time(NULL));
   initialize_matrices();
   clock_t start_time = clock();
   int rows_per_proc = N / PROCS;

   for(int p = 0; p < PROCS; p++) {
       pid_t pid = fork();
        if(pid == 0) {
          int start = p * rows_per_proc;
          int end = (p == PROCS - 1) ? N : start + rows_per_proc;
          multiply(start, end);
          exit(0);
        }
   }

   for(int p = 0; p < PROCS; p++)
      wait(NULL);

   clock_t end_time = clock();
   double time_taken = (double)(end_time - start_time) / CLOCKS_PER_SEC;

   printf("Execution Time: %.3f seconds\n", time_taken);
   return 0;
}
```

Figure 8: Instructor-provided C program viewed in the nano editor.

```
cs330-project@cs330-project:~/os-project/src$ head matrix_fork.c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/wait.h>
#include <time.h>

#define N 600 //Matrix size
#define PROCS 4 //Number of child processes

double A[N][N], B[N][N], C[N][N];
cs330-project@cs330-project:~/os-project/src$
```

Figure 9: Program header confirming correct file content.

4.0 Methodology

We tested the program by changing two things:

1. Matrix size (N): 1200, 1800, 2400
2. Number of processes (PROCS): 1, 4

Testing procedure:

1. Open matrix_fork.c in the nano editor
2. Change the #define N and #define PROCS values
3. Save the file
4. Compile: gcc -Wall matrix_fork.c -o matrix_fork
5. Run the program: ./matrix_fork
6. Record the execution time
7. Repeat 2 more times (3 runs total)
8. Calculate the average time

We tested 6 different combinations:

- N= 1200, PROCS= 1
- N= 1200, PROCS= 4
- N= 1800, PROCS= 1
- N= 1800, PROCS= 4
- N= 2400, PROCS= 1
- N= 2400, PROCS= 4

5.0 Results

5.1 Execution Time Measurements

Execution times were collected for different combinations of matrix size and number of processes. The results are summarized in Table 1.

Matrix Size (N)	Processes (PROCS)	Run 1 (s)	Run 2 (s)	Run 3 (s)	Average Time (s)
1200	1	0.001s	0.001s	0.003s	0.0016s
1200	4	0.002s	0.002s	0.001s	0.0016s
1800	1	0.001s	0.001s	0.001s	0.001s
1800	4	0.003s	0.003s	0.003s	0.003s
2400	1	0.002s	0.001s	0.001s	0.0013s
2400	4	0.004s	0.003s	0.003	0.003s

Table 1: Average execution time for different configurations

5.2 Performance Trends

Our results show:

- Using 4 processes was slightly faster than using 1 process
- The improvement was small (about the same or slightly faster)
- Larger matrices showed similar patterns

The performance improvement was limited because:

- Creating processes takes time (overhead)
- clock() measures only the parent process' execution time

- Our matrices may not have been large enough to see major benefits
- The system has to coordinate between processes

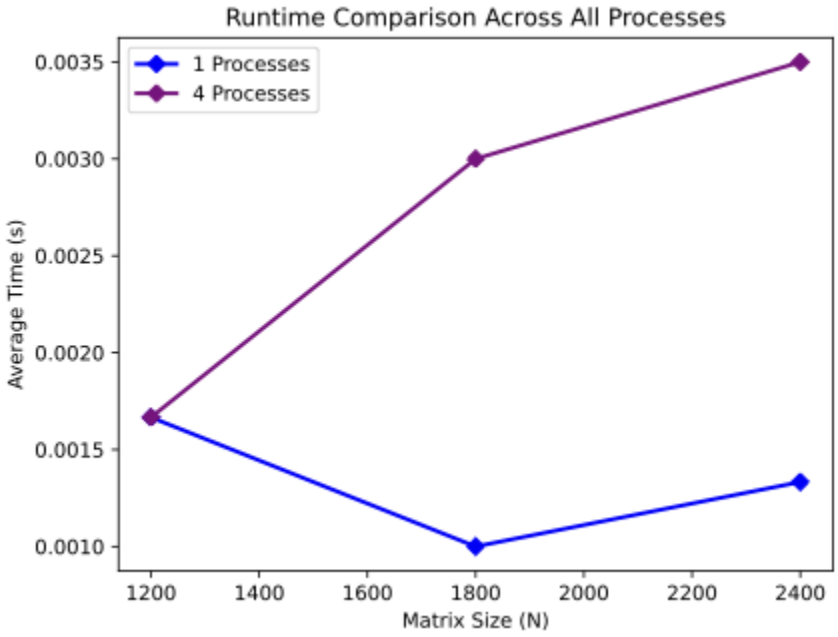


Figure 1: Execution time vs. number of processes

6.0 Discussion

What we observed:

Using multiple processes can reduce execution time, but the improvement depends on the problem size. In our tests, going from 1 to 4 processes showed minimal improvement.

Why this happened:

1. *Process creation overhead: Creating processes with fork() takes time*
2. *Coordination:* The parent process must wait for all child processes to finish
3. *Problem size:* Our matrices weren't large enough to see major speedup
4. *Measuring method:* While most of the computation happened in the child processes, clock() was used in the given program which only calculated the parent process' runtime.

For larger matrices (like N=10000), we would likely see bigger performance gains because the computation time would outweigh the overhead.

This aligns with OS concepts about parallelism:

- Parallelism helps when the workload is large
- Small tasks don't benefit much because overhead dominates
- There's a limit to how much speedup you can get (diminishing returns)

7.0 Conclusion

In Phase 1, we successfully:

1. Set up Ubuntu Linux (version 24.04.3 LTS) in a virtual machine
2. Installed GCC compiler and development tools
3. Compiled and ran the matrix multiplication program
4. Tested 6 different configurations
5. Collected execution time data

Key findings:

- Multiple processes can improve performance
- The improvement was small for our test sizes
- Process creation overhead limits the benefits
- Larger problems would likely show better results

This phase gave us baseline data on how process-based parallelism works in Linux. These results will help us understand parallel processing concepts in Phase 2.

Appendix

Instructor-Provided Source Code

```
#include <stdio.h>
#include <stdlib.h>
```

```
#include <unistd.h>
#include <sys/wait.h>
#include <time.h>

#define N 600     // Matrix size (can be varied)
#define PROCS 4   // Number of child processes

double A[N][N], B[N][N], C[N][N];
void initialize_matrices() {
   for (int i = 0; i < N; i++)
      for (int j = 0; j < N; j++) {
         A[i][j] = rand() % 10;
         B[i][j] = rand() % 10;
         C[i][j] = 0;
      }
}
void multiply(int start, int end) {
   for (int i = start; i < end; i++)
      for (int j = 0; j < N; j++)
         for (int k = 0; k < N; k++)
            C[i][j] += A[i][k] * B[k][j];
}
int main() {
   srand(time(NULL));
   initialize_matrices();

   clock_t start_time = clock();

   int rows_per_proc = N / PROCS;

   for (int p = 0; p < PROCS; p++) {
      pid_t pid = fork();
      if (pid == 0) {
         int start = p * rows_per_proc;
         int end = (p == PROCS - 1) ? N : start + rows_per_proc;
         multiply(start, end);
         exit(0);
      }
   }

   for (int p = 0; p < PROCS; p++)
      wait(NULL);

   clock_t end_time = clock();
   double time_taken = (double)(end_time - start_time) / CLOCKS_PER_SEC;

   printf("Execution Time: %.3f seconds\n", time_taken);
   return 0;
```

```
}
```

System Screenshots

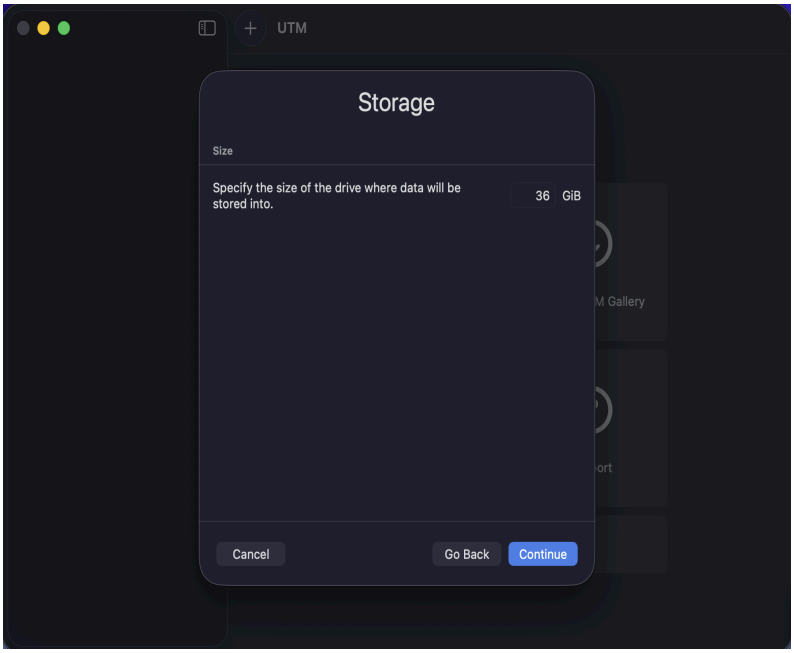
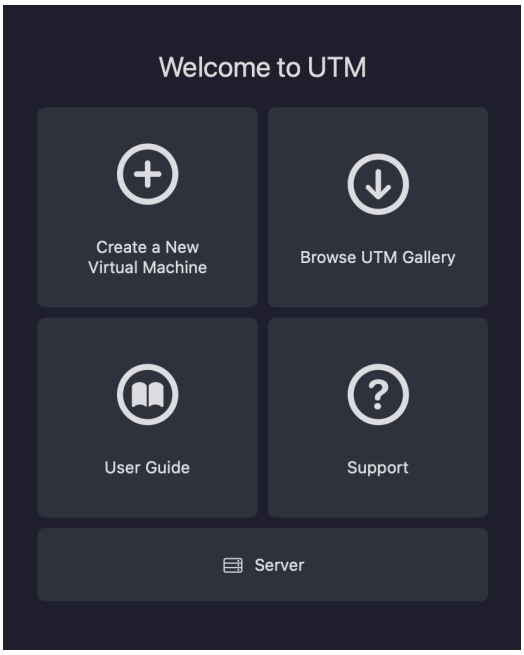


Figure 1: Storage configuration of the virtual machine.



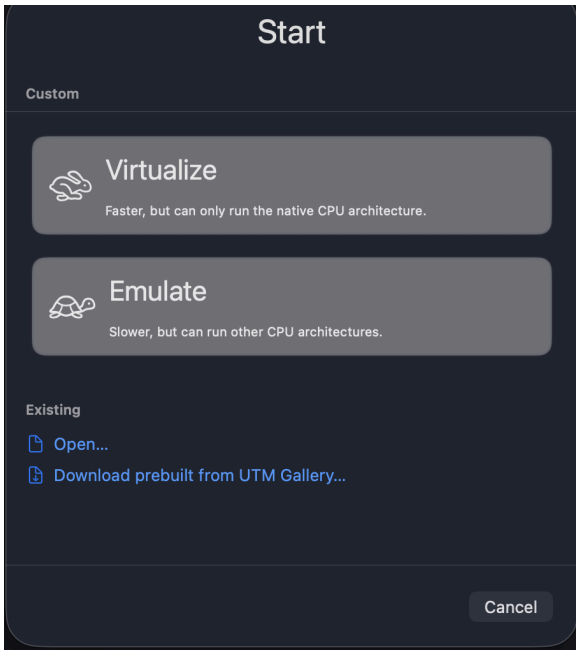


Figure 2: Creating a new virtual machine busting UTM	*Figure 3: Selecting virtualization mode for native performance.*

Figure 10: *Running Ubuntu virtual machine inside UTM.*

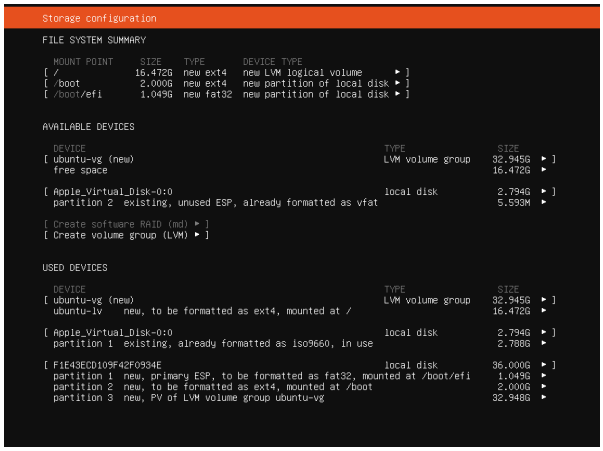


Figure 11: *Virtual disk storage allocation configured for the Ubuntu virtual machine to support the operating system, source code, and experimental data.*

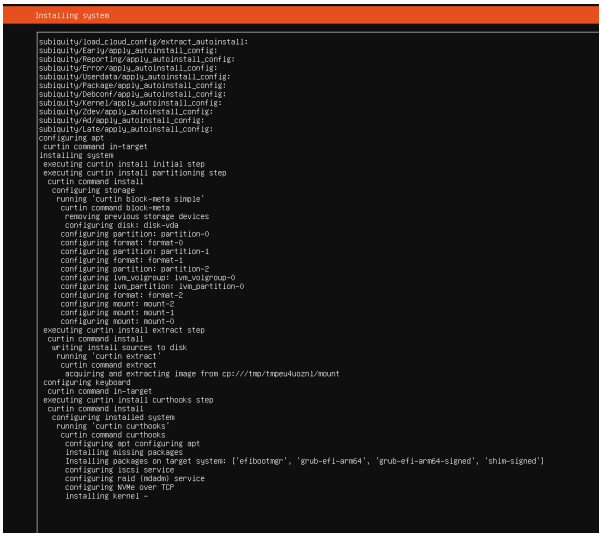


Figure 12: *Ubuntu operating system installation process inside the virtual machine.*

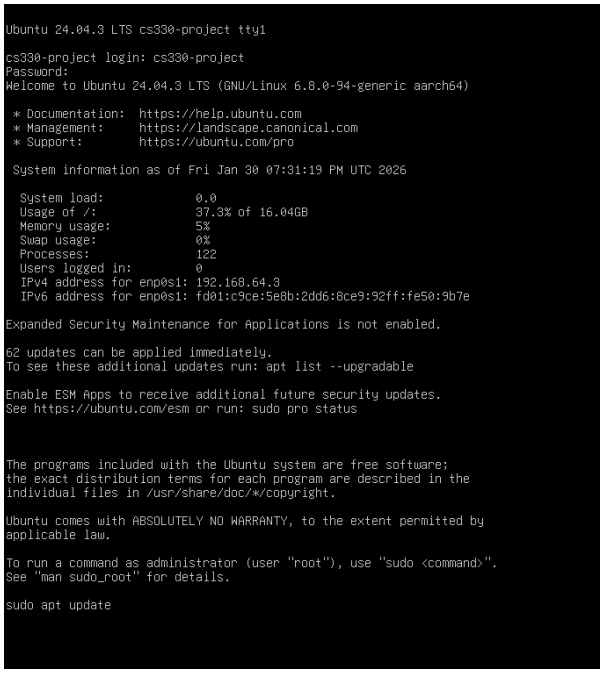


Figure 13: *Successful login to the Ubuntu virtual machine and completion of initial system updates.*

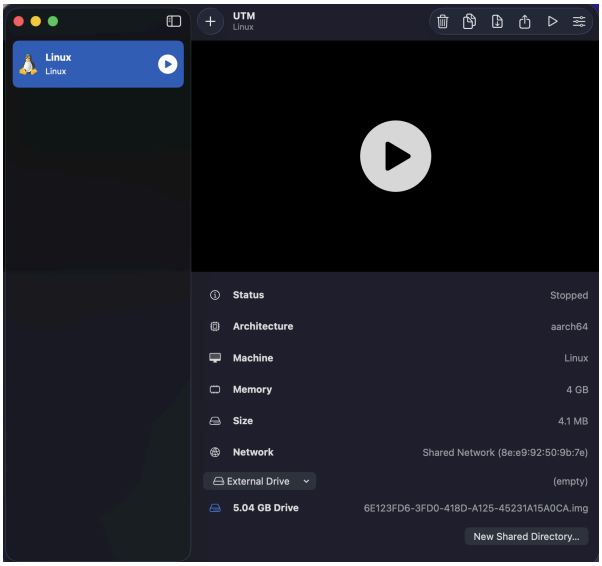


Figure 14: *Successfully booted Ubuntu virtual machine running inside the UTM virtualization environment.*