W4 Lab Task 1

Created by	Alexis Ryu
Created time	@August 22, 2025 10:19 AM
Last edited by	Alexis Ryu
Last updated time	@August 22, 2025 10:36 AM

Task 1

Overview

- Serial program to calculate primes in range
- Runtime: 2.4313 seconds (10,000,00)

is_prime

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

int is_prime(int num) {

    int limit = (int) sqrt(num);
    for (int i = 3; i <= limit; i += 2) {
        if (num % i == 0) return 0; // if i divides num, num must not be prime
    }
    return 1;
}
```

1. Cast the square root of num (double) to an int.
2. Iterate for all odd numbers for 3 to sqrt(n) - these are the only numbers we need to check to see if prime.
3. Return 1 (true) if num is prime, else return 0 (false)

main function

```
int main() {
int n;
printf("Enter n: ");
scanf("%d", &n); // store our scan into n

struct timespec start, end;
clock_gettime(CLOCK_MONOTONIC, &start); // store the time into start

if (n < 2) return 0;
if (n == 2) return printf("2");
```

- Get user input
- Store the start time
- Input validation and edge cases

```
FILE *fp = NULL; // avoid seg fault
if (n > 100000) { // write to file if n is large
    fp = fopen("primes1.txt", "w");
    if (!fp) {
        perror("File open failed");
        return EXIT_FAILURE;
    }
}

if (fp)
```

```
        fprintf(fp, "%d\n", 2);
    else
        printf("%d ", 2);
```

- Create a file for large value of n
- Either write to file or print to console for edge case of 2 (the only even prime number)

```
    for (int i = 3; i < n; i += 2) {
        if (is_prime(i)) {
            if (fp)
                fprintf(fp, "%d\n", i);
            else
                printf("%d ", i);
        }
    }
```

- Only check odd numbers,
- Print number to console or write to file, depending on where a file was created

```
    if (fp) fclose(fp); //close writing once we are done

    clock_gettime(CLOCK_MONOTONIC, &end); // store clock time into end
    double elapsed = (end.tv_sec - start.tv_sec) + (end.tv_nsec - start.tv_nsec)
/ 1e9;
    printf("\nTime taken: %.4f seconds\n", elapsed); //0.1247 seconds on deskt
op

    return 0;
}
```

- Wrap up

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

//2.4313 seconds

int is_prime(int num) {
    /*
    1. Cast the square root of num (double) to an int.
    2. Iterate for all odd numbers for 3 to sqrt(n) - these are the only numbers w
e need to check to see if prime.
    3. Return 1 (true) if num is prime, else return 0 (false)
    */
    int limit = (int) sqrt(num);
```

```
    for (int i = 3; i <= limit; i += 2) {
        if (num % i == 0) return 0; // if i divides num, num must not be prime
    }
    return 1;
}

int main() {
    int n;
    printf("Enter n: ");
    scanf("%d", &n); // store our scan into n

    struct timespec start, end;
    clock_gettime(CLOCK_MONOTONIC, &start); // store the time into start

    if (n < 2) return 0;
    if (n == 2) return printf("2");

    FILE *fp = NULL; // avoid seg fault
    if (n > 100000) { // write to file if n is large
        fp = fopen("primes1.txt", "w");
        if (!fp) {
            perror("File open failed");
            return EXIT_FAILURE;
        }
    }

    if (fp)
        fprintf(fp, "%d\n", 2);
    else
        printf("%d ", 2);


    for (int i = 3; i < n; i += 2) { // only check odd numbers
        if (is_prime(i)) {
            if (fp)
                fprintf(fp, "%d\n", i);
```

```
        else
            printf("%d ", i);
    }
}

if (fp) fclose(fp); //close writing once we are done

clock_gettime(CLOCK_MONOTONIC, &end); // store clock time into end
double elapsed = (end.tv_sec - start.tv_sec) + (end.tv_nsec - start.tv_nsec)
/ 1e9;
printf("\nTime taken: %.4f seconds\n", elapsed); //0.1247 seconds on deskt
op

return 0;
}
```

W4 Lab Task 2

Created by	Marcus Miccelli
Created time	@August 22, 2025 10:21 AM
Last edited by	Marcus Miccelli
Last updated time	@August 22, 2025 11:19 AM

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <pthread.h>
#include <time.h>

#define MAX_THREADS 16 //prevents creating too many by accident
//0.7926 seconds (6 threads), 2.1650 seconds (1 thread)

typedef struct {
    int start, end;   // range [start, end) assigned to this thread
    int *primes;      // array of primes found in this range
    int count;        // number of primes found
} ThreadData;
```

```
int is_prime(int num) {
    int limit = (int) sqrt(num);
    for (int i = 3; i <= limit; i += 2) {
        if (num % i == 0) return 0;
    }
```

```
    return 1;
}
```

```
void* find_primes(void* arg) {
    /*
    . is used when you have a struct variable directly.
    → is used when you have a pointer to a struct.

    (*data).primes same as data→primes
    */
    ThreadData* data = (ThreadData*) arg;   // unpack thread data - if you tried using arg directly, the compiler wouldn't know what fields exist, because it just sees a void*
    data→count = 0;

    // Allocate enough space for worst-case (every number is prime in range).
    data→primes = malloc((data→end - data→start) * sizeof(int));

    if (data→start == 2) {
        data→primes[data→count++] = 2;
    }
    if (data→start % 2 == 0) {
        data→start += 1;
    }

    for (int i = data→start; i < data→end; i+=2) {
        if (is_prime(i)) {
            data→primes[data→count++] = i;  // store prime
        }
    }
    return NULL;
}

int main() {
    int n, num_threads;
```

```
    printf("Enter n: ");
    scanf("%d", &n);

    printf("Enter number of threads (<= %d): ", MAX_THREADS);
    scanf("%d", &num_threads);

    if (num_threads > MAX_THREADS) num_threads = MAX_THREADS;

    pthread_t threads[MAX_THREADS]; //array of thread identifiers
    ThreadData thread_data[MAX_THREADS]; //array of structs (one struct per t
hread)

    struct timespec start, end;
    clock_gettime(CLOCK_MONOTONIC, &start); // store the time into start

    if (n < 2) return 0;
    if (n == 2) return printf("2");

    int chunk = n / num_threads; // how many numbers per thread (static sched
uling)
    for (int i = 0; i < num_threads; i++) {
        thread_data[i].start = (i == 0) ? 2 : i * chunk;
        thread_data[i].end = (i == num_threads - 1) ? n : (i + 1) * chunk; // last thre
ad covers any remainder for n
        pthread_create(&threads[i], NULL, find_primes, &thread_data[i]); // starts
theead, runs find_primes(), passing thread_data[i]
    }

    for (int i = 0; i < num_threads; i++) {
        pthread_join(threads[i], NULL); //waits for each thread to finish and ensur
es results are ready before merging
    }

    FILE *fp = NULL; // Avoid seg fault
    if (n > 100000) { // Write to file if n is large
        fp = fopen("primes2.txt", "w");
```

```
        if (!fp) {
            perror("File open failed");
            return EXIT_FAILURE;
        }
    }

    // Since ranges are already ordered, concatenation preserves ascending or
der
    for (int i = 0; i < num_threads; i++) {
        for (int j = 0; j < thread_data[i].count; j++) {
            if (fp)
                fprintf(fp, "%d\n", thread_data[i].primes[j]);
            else
                printf("%d\n", thread_data[i].primes[j]);

        }
        free(thread_data[i].primes); // free memory allocated by each thread
    }
    if (fp) fclose(fp);

    clock_gettime(CLOCK_MONOTONIC, &end); // store clock time into end
    double elapsed = (end.tv_sec - start.tv_sec) + (end.tv_nsec - start.tv_nsec)
/ 1e9;
    printf("\nTime taken: %.4f seconds\n", elapsed);

    return 0;
}
```

- What is the speed-up? Is it reasonable? Please explain.
  - 2.4313 → 0.7926. Not the theoretical 6x but still substantial.
- How would the speed-up change when you increase and decrease the number of
  threads? Why?
  - Increase relative to threads general

- How do you distribute the tasks to the threads? Is it a good approach? Please explain.
    - Static scheduling.

W4 Lab task 3

Created by	B Ben Leahy
Created time	@August 22, 2025 10:18 AM
Last edited by	B Ben Leahy
Last updated time	@August 22, 2025 11:36 AM

Task 3

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <time.h>
#include <omp.h>

// 0.6375 seconds

// check primality
int is_prime(int num)
/*Returns 1 if the number is a prime, and 0 if the number is not a prime*/
{
    int limit = (int)sqrt(num);
    for (int i = 3; i <= limit; i += 2)
    {
        if (num % i == 0)
            return 0;
    }
    return 1;
}
```

```
int main()
{
    int n;
    printf("Enter number: ");
    scanf("%d", &n);

    int largeN = 1000;

    // Start clock
    struct timespec start, end;
    clock_gettime(CLOCK_MONOTONIC, &start); // store the time into start

    // allocate n bytes worth of memory to mark numbers as prime or not prime.
Starts at 1
    int *primes = malloc((n + 1) * sizeof(int));

    // Handle small cases to allow us to skip even numbers
    if (n < 2)
        return 0;
    primes[2] = 1; // include prime 2 so we can skip all other evens

// parallel loop: mark primes
#pragma omp parallel for schedule(dynamic)
    for (int i = 3; i <= n; i += 2)
    {
        if (is_prime(i))
        {
            primes[i] = 1;
        }
        else
        {
            primes[i] = 0;
        }
    }
    // End parallel part
```

```
    FILE *pFile = fopen("primes3.txt", "w");

    // Print prime 2
    (n < largeN) ? printf("%d\n", 2) : fprintf(pFile, "%d\n", 2); // Finish accountin
g for the only even prime

    for (int i = 3; i <= n; i += 2)
    {
        if (primes[i])
        {
            if (n < largeN)
            {
                printf("%d\n", i);
            }
            else
            {
                fprintf(pFile, "%d\n", i);
            }
        }
    }

    fclose(pFile);
    free(primes); // we have already moved the contents of each thread's prime
s into the output file/ printed to terminal

    // End clock
    clock_gettime(CLOCK_MONOTONIC, &end); // store clock time into end
    double elapsed = (end.tv_sec - start.tv_sec) + (end.tv_nsec - start.tv_nsec)
/ 1e9;
    printf("\nTime taken: %.4f seconds\n", elapsed);

    return 0;
}

// Time task 1: 2.4313 seconds
```

```
// Time task 2: 0.7926 seconds (6 threads)
// Time task 3: 0.6375 seconds (6 threads)
```

Discussion points:

- Primes function → skip even and store in array size n
- Magic numbers are BANNED
- Dynamic scheduling → task 2 is limited by the chunks, the last chunk will require a far greater number of operations because of the number of operations required to determine if each is a prime (O(root(n)). Number of threads is number of cores. Number of tasks is the number of iterations.
- malloc: at runtime not compile time. Stored in heap not stack. Reduces chance of blowing stack with large n. Heap shared between threads.
- Speed up
    - *4 from task 1. Theoretical maximum is *6. But we have overhead of creating and joining threads, and assigning each iteration of the loop (a task) to a thread.
    - 25% faster than task 2 because we are doing dynamic scheduling. This is not limited by the last chunk in task 2 being a limiting factor.
    - If we increase the number of threads past the number of cores we would expect a decrease in efficiency → we have more thread overhead and task switching. This could be worthwhile if we had API calls or something that requires waiting in our code but we don't.

Race condition: a race condition is when multiple threads attempt to modify shared data concurrently → meaning that there are different possible outcomes of the code based on which thread arrives first.

Other stuff

Timing something

We can just run

```
time ./a.out -lm
```

To get the time that it takes to run the program.

Or:

```
double elapsed_time;
clock_t start = clock();
//Put your code here.
clock_t end = clock();
elapsed_time = (double) (start - end) / CLOCKS_PER_SEC;

//or

// Other clock
double start2 = omp_get_wtime();
// End other clock
double end2 = omp_get_wtime();
printf("\nExecution time: %f seconds\n", end2 - start2);
```