	Carátula para entrega de prácticas
Facultad de Ingeniería	Laboratorio de docencia

Laboratorios de computación salas A y B

Profesor: M.I. Edgar Tista García

Asignatura: Estructura de datos y algoritmos II -1317

Grupo: 10

No. de práctica(s): 11

Integrante(s): Mendoza Hernández Carlos Emiliano

No. de lista o brigada: 26

Semestre: 2023-1

Fecha de entrega: 9 de diciembre del 2022

Observaciones:

CALIFICACIÓN: _____

Práctica 11.

Introducción a OpenMP

OBJETIVO

- El estudiante conocerá y aprenderá a utilizar algunas de las directivas de OpenMP utilizadas para realizar programas paralelos.

DESARROLLO

Sección 1. Ejercicios de la guía OpenMP

Actividad 1.

Implementación de *hola.c*

```c
#include <stdio.h>

int main()
{
    int i;
    printf("Hola Mundo\n");
    for (i = 0; i < 10; i++)
        printf("Iteracion: %d\n", i);
    printf("Adios \n");
    return 0;
}
```

Ejecución de *hola.out*

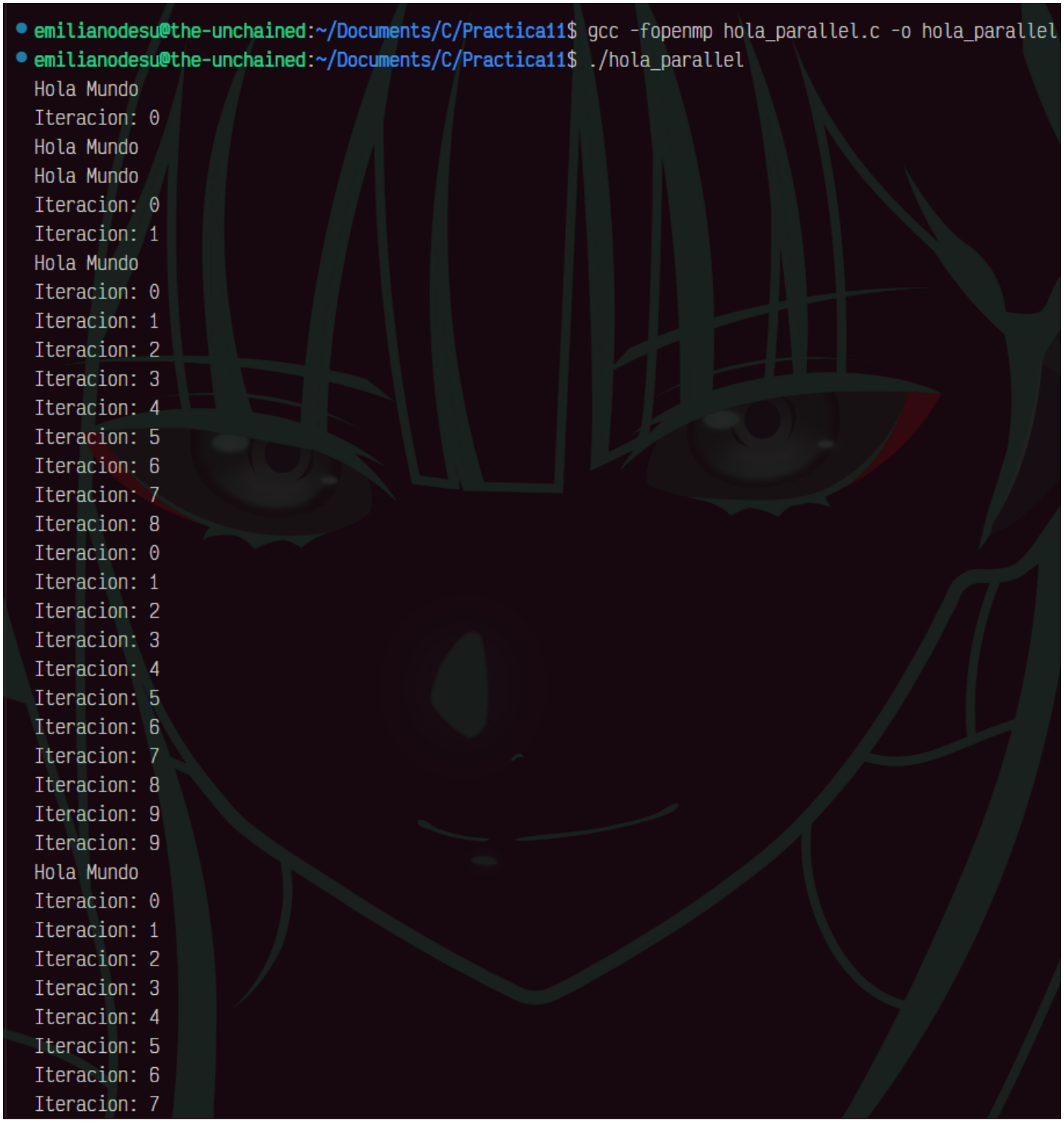
```
emilianodesu@the-unchained:~/Documents/C/Practica11$ gcc hola.c -o hola
emilianodesu@the-unchained:~/Documents/C/Practica11$ ./hola
Hola Mundo
Iteracion: 0
Iteracion: 1
Iteracion: 2
Iteracion: 3
Iteracion: 4
Iteracion: 5
Iteracion: 6
Iteracion: 7
Iteracion: 8
Iteracion: 9
Adios
emilianodesu@the-unchained:~/Documents/C/Practica11$
```

Comentarios: El código anterior es una versión serial de un programa que recorre un ciclo *for* que imprime una variable.

Implementación de *hola_parallel.c*

```c
#include <stdio.h>
int main()
{
#pragma omp parallel
    {
        int i;
        printf("Hola Mundo\n");
        for (i = 0; i < 10; i++)
            printf("Iteracion: %d\n", i);
    }
    printf("Adios \n");
    return 0;
}
```

Ejecución de *hola_parallel.out*



Comentarios: *Al compilar, usamos por primera vez la bandera -fopenmp para indicar que se agregan las directivas de openMP. La salida imprime, aparentemente en desorden, el texto de "hola mundo" y el valor de las iteraciones del ciclo for, pero no una sino varias veces (8 en total).*

¿Qué diferencia hay en la salida del programa con respecto a la secuencial?

El programa de *hola_parallel* ejecuta repetidamente las instrucciones dentro del constructor *parallel*. Podemos contar que las instrucciones se ejecutan 8 veces, pero no necesariamente se ejecutan una vez después de otra, sino que llegan a "traslaparse" las salidas. En cambio, el programa en su versión serial ejecuta una sola vez y en orden las instrucciones.

¿Por qué se obtiene esa salida?

Podemos relacionar las veces que se ejecuta el bloque de código con la cantidad de núcleos que tiene el procesador. En este caso particular, se ejecutó con un procesador de 8 núcleos, por ello se imprime 8 veces. El desorden aparente se debe a que ambos hilos se están ejecutando al mismo tiempo, y por ello no siempre aparecen las salidas en orden secuencial.

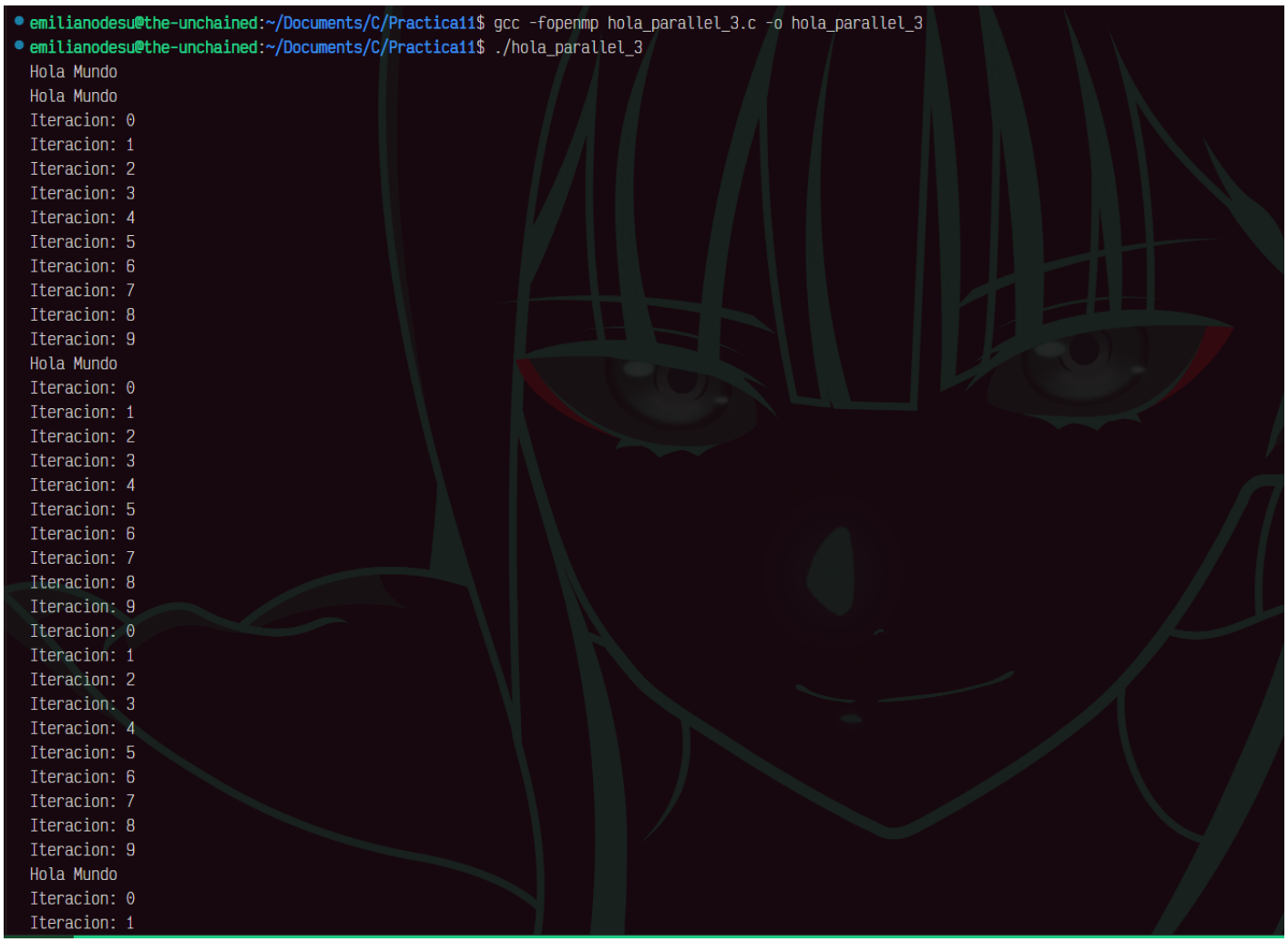
Actividad 2.

Modificación de *hola_parallel.c* agregando la cláusula *num_threads(4) (hola_parallel_3.c)*

```c
#include <stdio.h>
#include<omp.h>

int main()
{
    #pragma omp parallel num_threads(4)
    {
        int i;
        printf("Hola Mundo\n");
        for ( i = 0; i < 10; i++)
            printf("Iteracion: %d\n", i);
    }
    printf("Adios \n");
    return 0;
}
```

Ejecución de *hola_parallel_3.out*



Comentarios: Tenemos 3 maneras de cambiar el número de hilos por defecto. En este caso se optó por la opción de agregar la cláusula de *num_threads(n)*; sin embargo, esto también se puede lograr modificando la variable de ambiente *OMP_NUM_THREADS* desde la consola, o usando la función *omp_set_num_threads(n)* de la librería *omp.h*.

¿Qué sucedió en la ejecución con respecto al de la actividad 1?

La salida del programa es la ejecución del bloque dentro de la región paralela (imprime hola mundo y los valores de la variable del *for*), pero se realiza 4 veces dado que se estableció *n = 4*. De igual forma, al ejecutarse varias veces se obtuvieron distintos resultados para algunas de las ejecuciones (el desorden aparente).

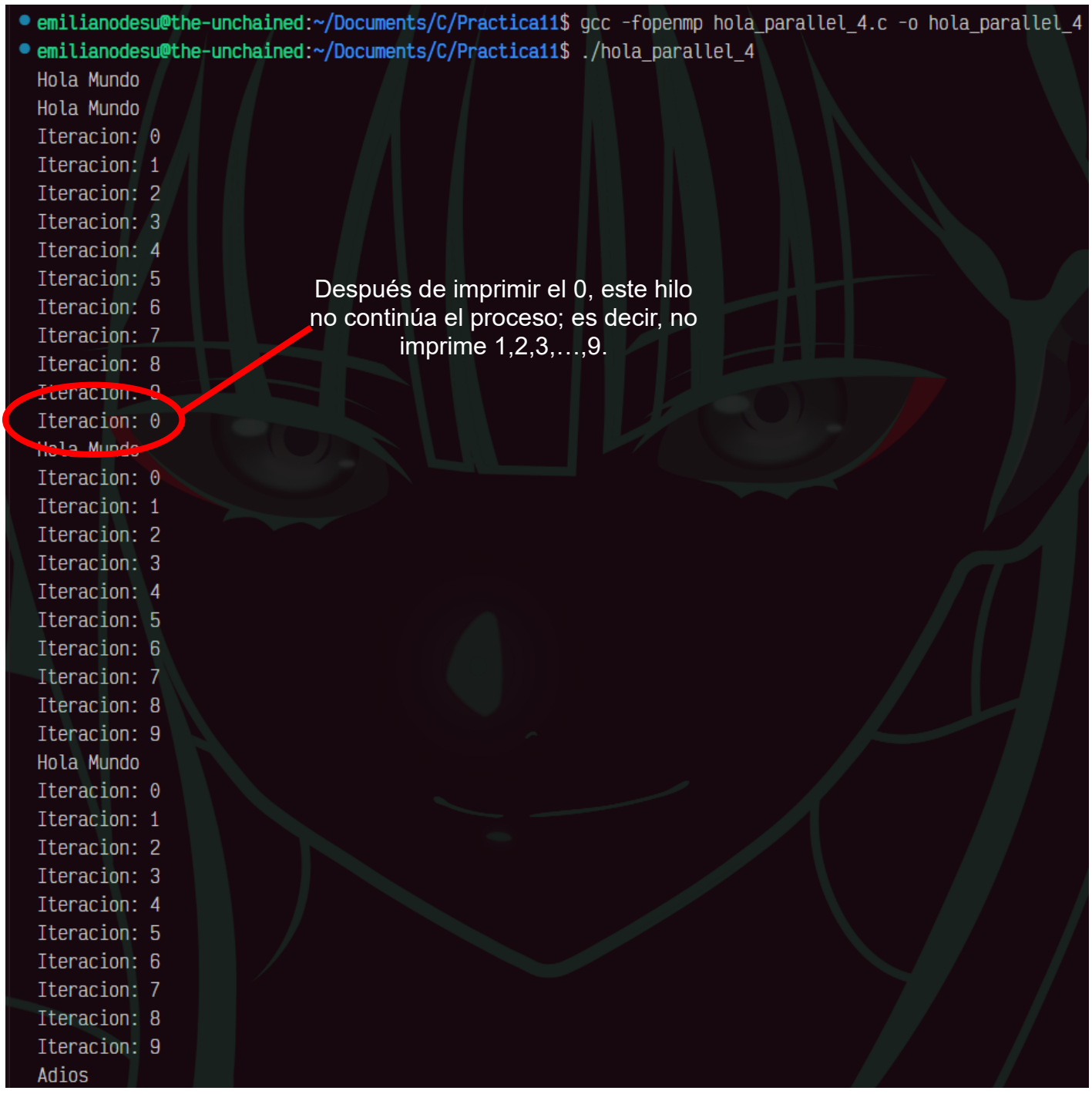
Actividad 3

Modificación de *hola_parallel_3.c* sacando *i* de la región paralela *(hola_parallel_4.c)*

```c
#include <stdio.h>
#include <omp.h>

int main()
{
    int i;
#pragma omp parallel num_threads(4)
    {
        printf("Hola Mundo\n");
        for (i = 0; i < 10; i++)
            printf("Iteracion: %d\n", i);
    }
    printf("Adios \n");
    return 0;
}
```

Ejecución de *hola_parallel_4.out*



Comentarios: Después de ejecutarlo varias veces, se observó que en algunas ocasiones se "interrumpió" el proceso de alguno (o más de uno) de los hilos.

¿Qué sucedió y por qué?

El programa no funcionó de la manera esperada; se presentaron anomalías con la variable de acceso compartido. Esto sucede porque varios hilos tuvieron acceso al recurso **sin control** alguno, intentando escribir en la misma localidad de memoria al mismo tiempo.

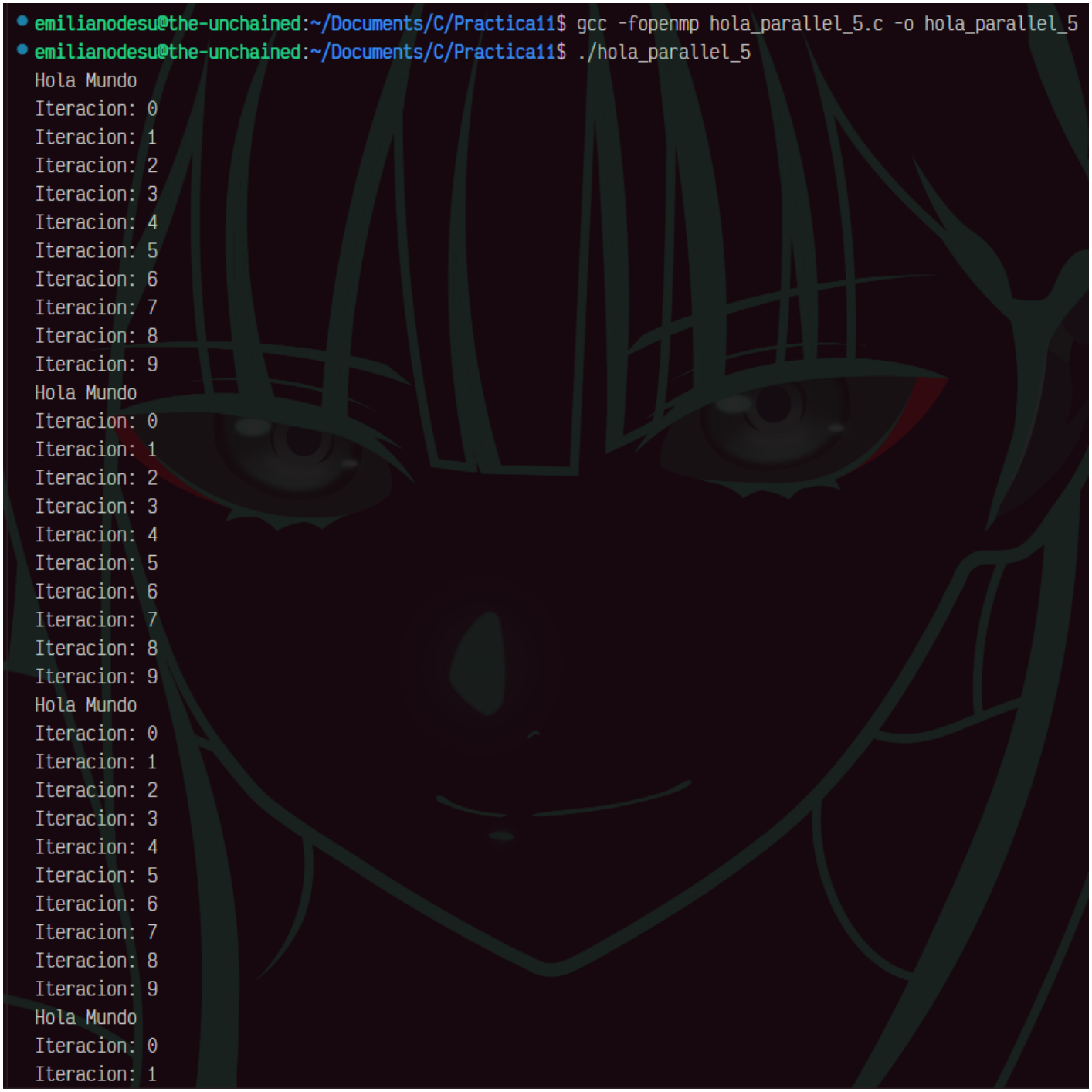
Actividad 4

Modificación de *hola_parallel_4.c* con la cláusula *private()* (*hola_parallel_5.c*)

```c
#include <stdio.h>
#include <omp.h>

int main()
{
    int i;
#pragma omp parallel num_threads(4) private(i)
    {
        printf("Hola Mundo\n");
        for (i = 0; i < 10; i++)
            printf("Iteracion: %d\n", i);
    }
    printf("Adios \n");
    return 0;
}
```

Ejecución de *hola_parallel_5.out*



Comentarios: Después de ejecutar varias veces el programa, se obtuvo una salida más consistente en cada ejecución.

¿Qué sucedió?

Se volvió a poner la variable *i* como privada. Sin embargo, la variable *i* sigue siendo declarada como de uso compartido (afuera de la región paralela), pero es “forzada” a ser privada. Se crea una copia para cada hilo y todos modifican dicha copia. De esta manera, la salida del programa vuelve a ser coherente.

Actividad 5

Implementación de *hola_parallel_6.c* para conocer el identificador de los hilos

```c
#include <stdio.h>
#include <omp.h>

int main()
{
    int tid, nth;
#pragma omp parallel num_threads(4) private(tid)
    {
        tid = omp_get_thread_num();
        nth = omp_get_num_threads();
        printf("Hola mundo desde el hilo %d de un total de %d\n", tid, nth);
    }
    printf("Adios \n");
    return 0;
}
```

Ejecución de *hola_parallel_6.out*

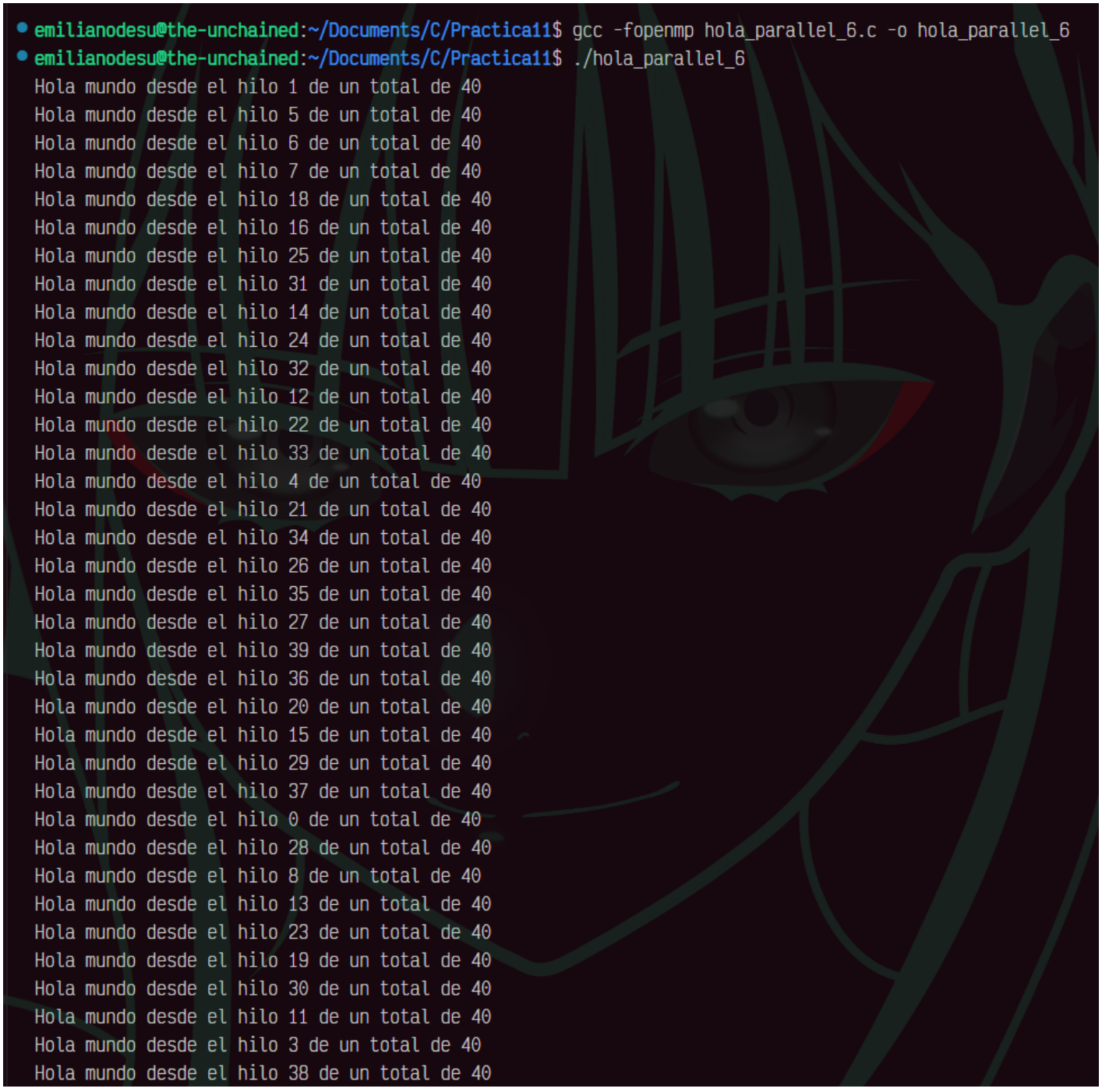
```
emilianodesu@the-unchained:~/Documents/C/Practica11$ gcc -fopenmp hola_parallel_6.c -o hola_parallel_6
emilianodesu@the-unchained:~/Documents/C/Practica11$ ./hola_parallel_6
Hola mundo desde el hilo 1 de un total de 4
Hola mundo desde el hilo 0 de un total de 4
Hola mundo desde el hilo 3 de un total de 4
Hola mundo desde el hilo 2 de un total de 4
Adios
emilianodesu@the-unchained:~/Documents/C/Practica11$
```

Comentarios: Los hilos se identifican con números enteros desde 0 hasta *n-1*.

¿Qué sucedió?

La función *omp_get_thread_num()* devuelve el identificador del hilo actual; por ello, la variable que guarda su valor debe ser especificada de uso privado. Por otro lado, usando la función *omp_get_num_threads()* devueve el número total de hilos que se ejecutan en el programa.

Ejecución de *hola_parallel_6.out* eliminando la cláusula *private()*



Comentarios: En este caso, todos los hilos escriben en la dirección de memoria de la variable *tid*, esto provoca un resultado impredecible y descontrolado.

Actividad 6

Implementación de *suma_serial.c*

```c
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#define n 10

void llenaArreglo(int *a);
void suma(int *a, int *b, int *c);

int main()
{
    int max, *a, *b, *c;
    a = (int *)malloc(sizeof(int) * n);
    b = (int *)malloc(sizeof(int) * n);
    c = (int *)malloc(sizeof(int) * n);
    llenaArreglo(a);
    llenaArreglo(b);
    suma(a, b, c);
}

void llenaArreglo(int *a)
{
    int i;
    for (i = 0; i < n; i++)
    {
        a[i] = rand() % n;
        printf("%d\t", a[i]);
    }
    printf("\n");
}

void suma(int *A, int *B, int *C)
{
    int i;
    for (int i = 0; i < n; i++)
    {
        C[i] = A[i] + B[i];
        printf("%d\t", C[i]);
    }
}
```

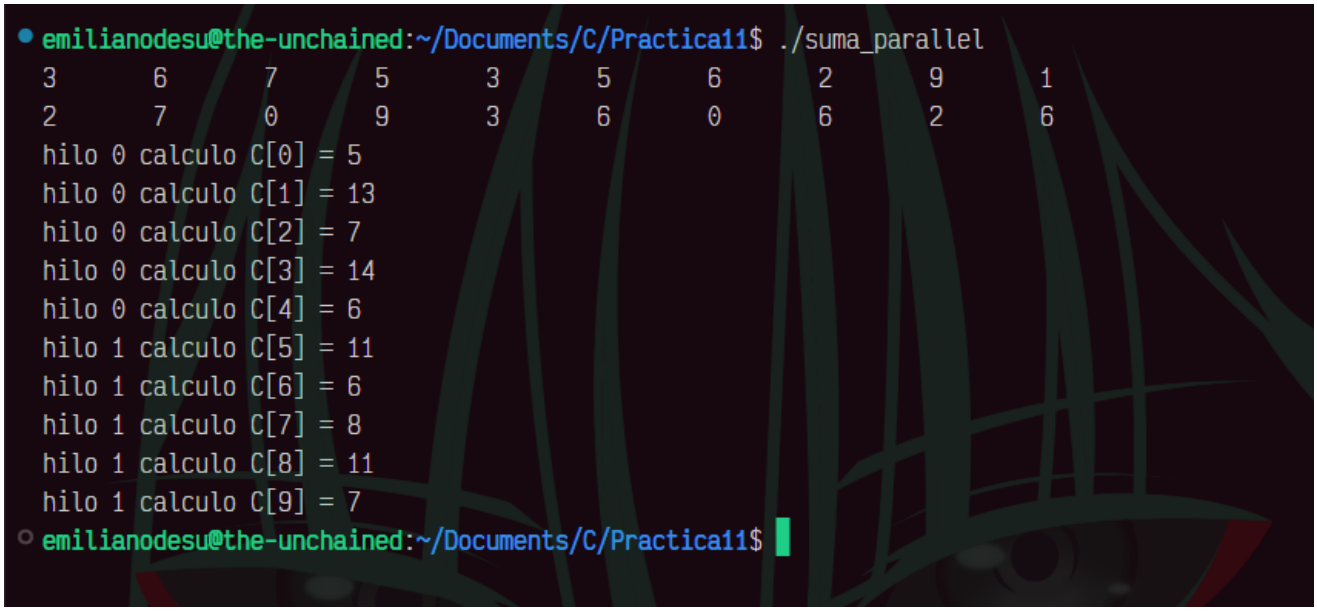
Ejecución de *suma_serial.out*

```
emilianodesu@the-unchained:~/Documents/C/Practica11$ gcc -fopenmp suma_serial.c -o suma_serial
emilianodesu@the-unchained:~/Documents/C/Practica11$ ./suma_serial
  3       6       7       5       3       5       6       2       9       1
  2       7       0       9       3       6       0       6       2       6
  5       13      7       14      6       11      6       8       11      7
emilianodesu@the-unchained:~/Documents/C/Practica11$
```

Modificación del método *suma()* en *suma_parallel.c*

```c
void suma(int *A, int *B, int *C)
{
    int i, tid, inicio, fin;
    omp_set_num_threads(2);
#pragma omp parallel private(inicio, fin, tid, i)
    {
        tid = omp_get_thread_num();
        inicio = tid * 5;
        fin = (tid + 1) * 5 - 1;
        for (i = inicio; i <= fin; i++)
        {
            C[i] = A[i] + B[i];
            printf("hilo %d calculo C[%d] = %d\n", tid, i, C[i]);
        }
    }
}
```

Ejecución de *suma_parallel.out*



Comentarios: Para la versión paralela, el hilo 0 suma la primera mitad del arreglo *A* con la primera mitad del arreglo *B*. El hilo 1 suma la segunda mitad de *A* con la segunda mitad de *B*. Para esto, cada hilo realiza las mismas instrucciones, pero utilizando índices diferentes para referirse a diferentes elementos del arreglo, entonces cada uno inicia u termina el ciclo en valores diferentes.

Actividad 7

Modificación de *hola_parallel* con el constructor *for* (*hola_parallel_7.c*)

```c
#include <stdio.h>
int main()
{
#pragma omp parallel
    {
        int i;
        printf("Hola Mundo\n");
        #pragma omp for
        for (i = 0; i < 10; i++)
            printf("Iteracion: %d\n", i);
    }
    printf("Adios \n");
    return 0;
}
```

Ejecución de *hola_parallel_7.out*

Comentarios: Usando el constructor *for*, se dividen las iteraciones entre los (8 en este caso) hilos. Por lo tanto, las iteraciones solo se hacen una vez para todo el ciclo, y no una vez por cada hilo como se tenía en *hola_parallel.c*.

Actividad 8

Modificación de *suma_parallel.c* usando el constructor *for* (*suma_parallel_2.c*)

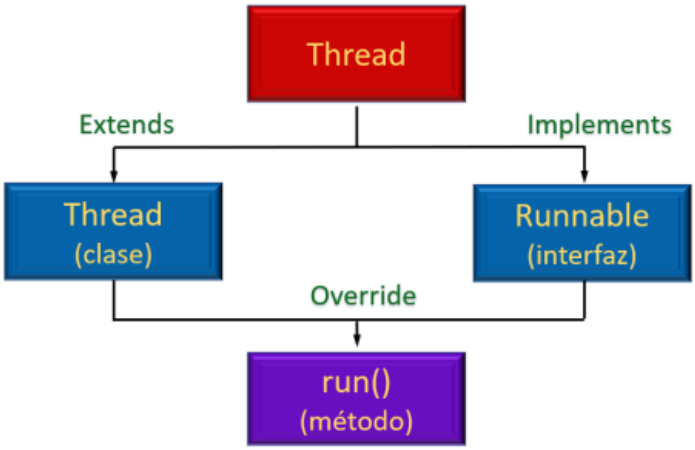
```c
void suma(int *A, int *B, int *C)
{
    int i, tid;
    #pragma omp parallel private(tid)
    {
        tid = omp_get_thread_num();
        #pragma omp for
        for (i = 0; i < n; i++)
        {
            C[i] = A[i] + B[i];
            printf("hilo %d calculo C[%d] = %d\n", tid, i, C[i]);
        }
    }
}
```

Ejecución de *suma_parallel_2.out*

```
emilianodesu@the-unchained:~/Documents/C/Practica11$ gcc -fopenmp suma_parallel_2.c -o suma_parallel_2
emilianodesu@the-unchained:~/Documents/C/Practica11$ ./suma_parallel_2
  3       6       7       5       3       5       6       2       9       1
  2       7       0       9       3       6       0       6       2       6
  hilo 6 calculo C[8] = 11
  hilo 7 calculo C[9] = 7
  hilo 1 calculo C[2] = 7
  hilo 1 calculo C[3] = 14
  hilo 2 calculo C[4] = 6
  hilo 0 calculo C[0] = 5
  hilo 0 calculo C[1] = 13
  hilo 3 calculo C[5] = 11
  hilo 5 calculo C[7] = 8
  hilo 4 calculo C[6] = 6
emilianodesu@the-unchained:~/Documents/C/Practica11$
```

Comentarios: En esta modificación se usan los hilos por defecto que se generan en la región paralela y el ciclo *for* divide las iteraciones entre los (8) hilos. Al igual que en el ejercicio 6, se realizan las mismas operaciones, pero sobre diferentes elementos del arreglo, pero para los valores asignados por el constructor *for*, permitiendo que cada hilo trabaje con valores diferentes del índice del arreglo.

Sección 2. Hilos en Java



En Java, la clase responsable de producir hilos es la clase *Thread*. Para que una clase tenga la funcionalidad de un hilo debe heredar de la clase *Thread*.

La clase *Thread* contiene el método *run()*, el cual define las acciones del hilo y, se conoce como el cuerpo del hilo. Por lo tanto, para añadir la funcionalidad deseada a cada hilo creado es necesario redefinir (sobrescribir) el método *run()*. Este método es invocado cuando se inicia el hilo con el método *start()*.

Por otra parte, existe una manera de implementar hilos en Java cuando ya no podemos heredar de la clase *Thread* (esto puede suceder porque Java maneja herencia simple, y si la clase ya hereda de otra no es posible heredar de *Thread* también) usando la interfaz *Runnable*. Esta interfaz permite producir hilos para otras clases. Para añadir la funcionalidad de hilo a una clase usando la interfaz *Runnable*, la clase debe implementar esta interfaz. Las clases que implementen la interfaz *Runnable* son proporcionadas con un método *run()* (que se debe definir e implementar en la clase) que es ejecutado por un objeto *Thread* creado.

Problema del productor – consumidor

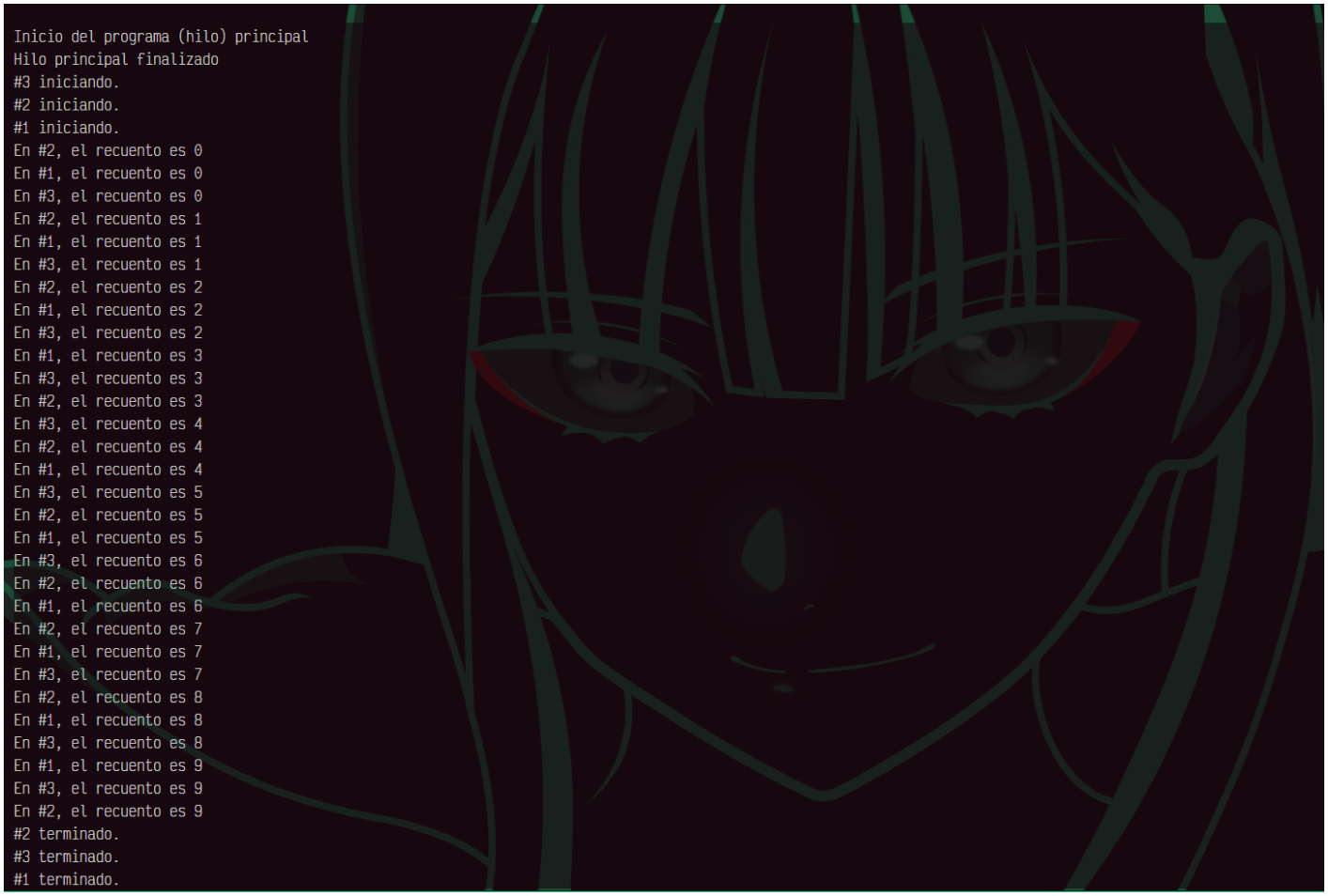
Este problema describe ejemplos de fallas de sincronización de multiprocesos. En él hay dos procesos: productor y consumidor. El productor es encargado de generar un producto, almacenarlo en un búfer y comenzar nuevamente, mientras que el consumidor toma productos del búfer que comparte con el productor. El problema se presenta en dos casos:

1. El consumidor no tiene productos que tomar del búfer.
2. El productor no tiene espacio en el búfer para añadir más productos.

Una idea de solución se presenta con el uso de semáforos:

1. Ambos procesos (productor y consumidor) se ejecutan simultáneamente.
2. En el caso del problema 1:
   - 2.1 Si el consumidor no encuentra productos en el búfer, el semáforo le indica al consumidor que deba esperar hasta que existan productos.
   - 2.2 El consumidor “duerme” mientras espera la señal del semáforo para continuar.
   - 2.3 Una vez que el productor comienza a llenar el buffer, el semáforo le indica al consumidor que puede continuar.
   - 2.4 El consumidor toma simultáneamente los productos uno a uno.
3. En el caso del problema 2:
   - 3.1 El productor llena la capacidad del búfer y no existe espacio para los demás productos.
   - 3.2 El semáforo le indica al productor que debe esperar que el consumidor se lleve los productos para seguir produciendo.
   - 3.3 El productor se pone a dormir.
   - 3.4 El consumidor se debe llevar los productos para liberar espacio en el búfer.
   - 3.5 El semáforo le indica al productor que puede continuar.

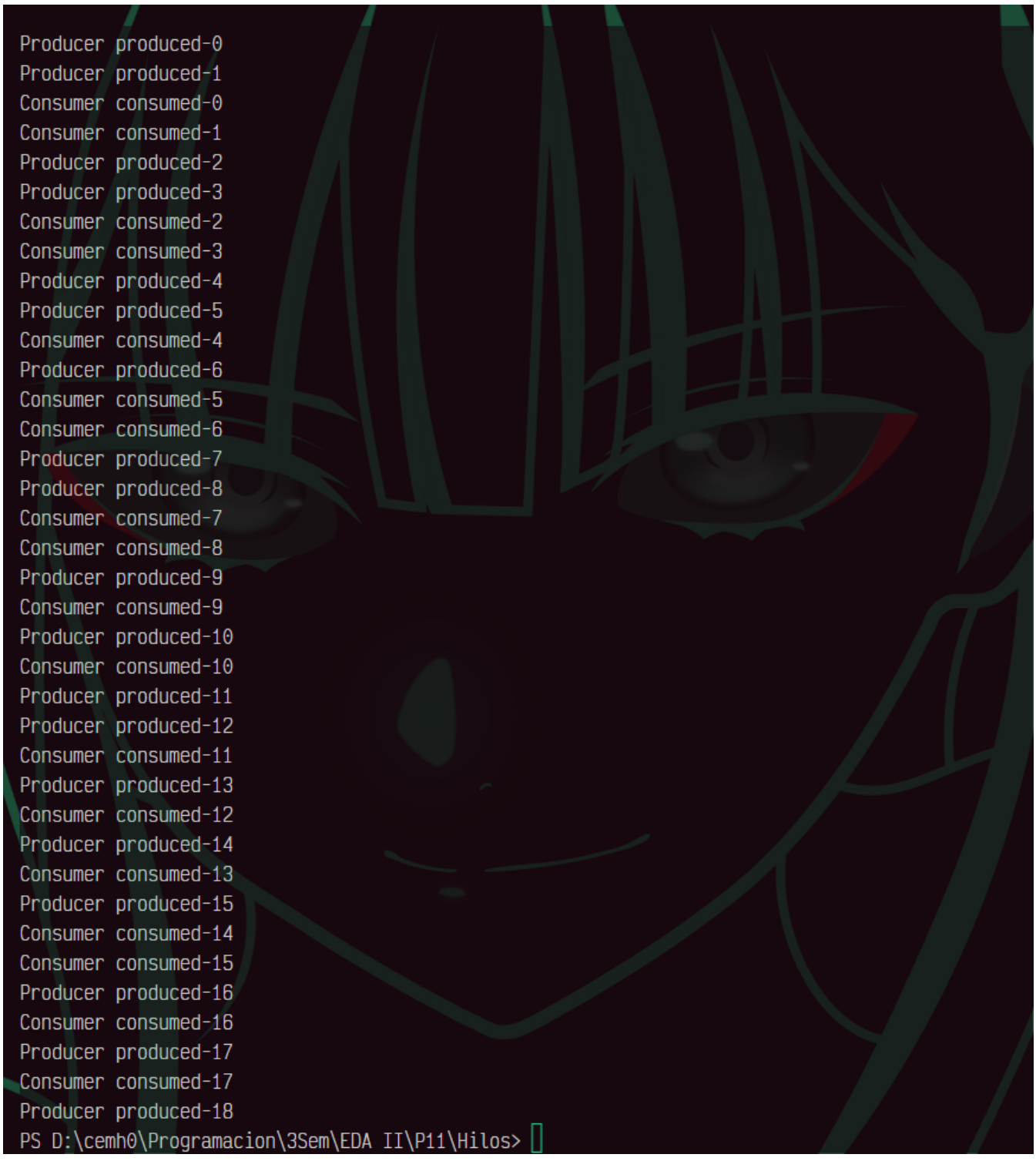
Ejecución de *Hilos1.java*



Este programa funciona con 3 hilos que ejecutan procesos concurrentes. Para cada iteración del ciclo *for*, se usa el método *sleep()* para hacer una pausa antes de ejecutar la siguiente instrucción. Como se puede ver el ciclo va de 0 a 9 y en cada iteración varía el orden de ejecución de los hilos.

Lo que más me llama la atención de este programa es que la clase *MiHilo2* implementa la interfaz *Runnable*. Para poder tener la funcionalidad de hilo se debe tener un objeto de clase *Thread* que, al inicializarse, debe incluir en sus parámetros una referencia a un objeto (t*arget*) que implementa la interfaz *Runnable.*

Ejecución de *Productor.java*



Este programa tiene una clase anidada (*PC*) que contiene una lista de 2 elementos de capacidad, y 2 métodos.

El método *produce()* tiene un bloque sincronizado (*synchronized*) lo que implica que no es posible, para dos llamadas a métodos o bloques sincronizados, intervenir en el mismo campo. Cuando se está ejecutando un bloque sincronizado, automáticamente se bloquea el recurso para los demás métodos o bloques sincronizados que se ejecuten. Una vez que el hilo ha terminado con el objeto, se libera el recurso para que otro método o bloque sincronizado pueda usarlo. Esto garantiza que los cambios en el estado de un objeto sean visibles para todos los demás hilos.

Dentro del bloque sincronizado, el hilo del productor se pone en espera si la capacidad de la lista está a tope. Para ello se usa el método *wait()*. Cuando es posible, el hilo del productor agrega un valor nuevo a la lista y notifica al consumidor (con el método *notify()*) que la lista ya no está vacía.

El método *consume()* también tiene un bloque sincronizado. En este bloque, el hilo del consumidor se pone en espera si la lista está vacía. Cuando es posible, elimina (consume) un objeto de la lista y notifica al productor que ya hay espacio en la lista.

En el método *main()* se crean dos hilos que ejecutarán los métodos de *produce()* y *consume()* simultáneamente. En la ejecución se puede notar como el productor deja de producir cuando la lista está llena, y como el consumidor deja de consumir cuando la lista está vacía.

Algo que me llamó la atención en este programa es el uso de una clase anónima que implementa la interfaz *Runnable*. Esta clase anónima es de un solo uso, el cual es implementar el método *run()* para el productor y para el consumidor, y reduce el código necesario para el programa.

CONCLUSIONES

Algunos puntos que se concluyen de la primera sección son los siguientes:

- OpenMP funciona por medio de directivas del compilador, las cuales instruyen al compilador para insertar órdenes en el código fuente y realizar una acción en particular.
- También, OpenMP tiene una librería de funciones y variables de entorno.
- El constructor más importante de OpenMP es *parallel*. Con él podemos crear regiones paralelas para ejecutar instrucciones con un número de hilos. Al finalizar sus instrucciones se tiene un solo hilo maestro.
- Para usar OpenMP es necesario analizar e identificar en que partes del programa se puede aplicar el paralelismo, y en ellas indicar de manera explícita el uso de un conjunto de hilos para resolver el problema.

Para la segunda sección se puede concluir que:

- Java implementa la programación concurrente con la clase *Thread* y la interfaz *Runnable*.
- Existen métodos o bloques sincronizados que controlan el acceso y cambios en el estado de objetos. Esto permite que los cambios en el estado de un objeto se hagan de manera segura y uno a la vez.
- Todos los objetos en Java tienen el método *wait()*, que pone en espera al hilo actual hasta ser despertado o interrumpido.
- Todos los objetos en Java tienen el método *notify()*, que despierta a un hilo que esté en espera de ser despertado.
- Implícitamente, Java hace uso de monitores, que proveen la sincronización y proporcionan un acceso controlado a los objetos.

Dicho todo lo anterior, considero que se ha cumplido con el objetivo planteado para el desarrollo de esta práctica dado que se conocieron y utilizaron las directivas más importantes de OpemMP para algoritmos paralelos; y, además, se profundizó en el uso de programación concurrente en Java.