INFORME DE LABORATORIO

(formato estudiante)

INFORMACIÓN BÁSICA					
ASIGNATURA:	Sistemas Operativos				
TÍTULO DE LA PRÁCTICA:	*Semaforos en linux*				
NÚMERO DE PRÁCTICA:	*08*	**AÑO LECTIVO:**	*2023*	**NRO. SEMESTRE:**	*05*
FECHA DE PRESENTACIÓN	*12/06/2023*	**HORA DE PRESENTACIÓN**	–		
INTEGRANTE (s): Yoset Cozco Mauri				**NOTA:**	
DOCENTE(s): *NIETO VALENCIA, RENE ALONSO*					

SOLUCIÓN Y RESULTADOS

Ejercicios resueltos:

1. Muestre los resultados, analice y describa la actividad que se muestra en el siguiente código realizado en el marco teórico.

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

int main() {
    sem_t my_semaphore;
    int rc;
    rc = sem_init(&my_semaphore, 0, 10);

    if (rc == 0) {
        printf("Semaforo iniciado correctamente.\n");
    } else {
        printf("Error al iniciar Semaforo.\n");
    }
    return 0;
}
```

```
Semaforo iniciado correctamente.


...Program finished with exit code 0
Press ENTER to exit console.
```

Se declara la variable my_semahore de tipo semaforo y tambien el entero rc.

Luego al entero rc se le asigna el valor de inicialización del semáforo, que inicializa su valor en 10. 0 para indica que no es compartido entre procesos y 10 para indicar la cantidad de recursos disponibles, se agregó las impresiones para ver el valor de rc y corroborar que el semáforo haya sido inicializado correctamente.

2. Muestre los resultados, analice y describa la actividad que se muestra en el siguiente código realizado en el marco teórico.

main.c

```c
#include <stdio.h>
#include <semaphore.h>
int main() {
    sem_t my_semaphore;
    int rc;
    rc = sem_init(&my_semaphore, 0, 10);
    if (rc == 0) {
        printf("Semaforo inicializado correctamente.\n");
    } else {
        printf("Error al inicializar el Semaforo.\n");
        return 1;
    }
    rc = sem_destroy(&my_semaphore);

    if (rc == 0) {
        printf("Semaforo destruido correctamente.\n");
    } else {
        printf("Error al destruir el Semaforoaforo.\n");
        return 1;
    }
    return 0;
}
```

Ln: 10, Col: 49

Run | Share | Command Line Arguments

```
Semaforo inicializado correctamente.
Semaforo destruido correctamente.


** Process exited - Return Code: 0 **
```

Se declara tanto el semáforo como el entero rc, nuevamente se asigna los valores a rc de la inicialización del semáforo con la condición (0 no compartido y 10 para indicar la cantidad de recursos disponibles), después de esto se realizan impresiones para ver si este proceso fue llevado de manera correcta, pasado las condiciones se aplica la función destroy que recibe como argumento nuestro

semáforo inicializado previamente, igual el valor de esta sentencia se guarda en rc, que también imprimimos para poder visualizar si se realizó de manera correcta.

4. Muestre los resultados, analice y describa la actividad que se muestra en el siguiente código realizado en el Marco teórico.

main.c

```c
#include <stdio.h>
#include <semaphore.h>
int main() {
    sem_t my_semaphore;
    int value;
    // Inicialización del semáforo con valor inicial de 10
    sem_init(&my_semaphore, 0, 10);
    // Obtener el valor inicial del semáforo
    sem_getvalue(&my_semaphore, &value);
    printf("El valor inicial del semáforo es %d\n", value);
    // Incrementar el valor del semáforo utilizando sem_post
    sem_post(&my_semaphore);
    // Obtener el valor del semáforo después de incrementarlo
    sem_getvalue(&my_semaphore, &value);
    printf("El valor del semáforo después del post es %d\n", value);
    return 0;
}
```

Ln: 18, Col: 1

Run | Share | Command Line Arguments

```
El valor inicial del semáforo es 10
El valor del semáforo después del post es 11


** Process exited - Return Code: 0 **
```

Se declaran las variables my_semaphore y value, se inicializa el semáforo con valor de 10(unidades de recurso disponible), se imprime el valor obtenido de la función getvalue que asignado el valor del semáforo a la variable value, luego con la función semo_post se incrementa el valor de my_semaphore, y nuevamente cambiamos el valor de value para el nuevo valor de semaphore. Los dos valores de my_semaphore fueron imprimidos.

Ejercicios propuestos:

1. El siguiente programa se muestra una solución al problema del Productor – Consumidor haciendo uso de los semáforos contadores vistos en este laboratorio. Observe que el productor espera a que se ingresen caracteres los cuales simulan ser los ítems que produce.

Prueba previa a corregir código:

Prueba 1: compile y ejecute el programa anterior, y observe el comportamiento y la interacción entre el proceso productor y consumidor.

main.c

```
27
28  static void *consumidor(void *arg2) {
29      int i, sale, consumir;
30      sale = 0;
31      for (i = 1; i <= SUMSIZE; i++) {
32          sem_wait(&llenos);
33          sleep(1); // el consumo se demora 1 segundo
34          sem_post(&vacios); // Esto debe ser sem_post(&vacios);
35          consumir = buffer[sale];
36          sale = (sale + 1) % BUFSIZE;
37          printf(" \t%c\tConsumidor -->\n", consumir);
38          sem_post(&vacios);
39      }
40      return NULL;
41  }
42
43  int main() {
44      pthread_t prodtid;
45      pthread_t constid;
46
47      // semáforos contadores para controlar los espacios en el almacén
48      sem_init(&llenos, 0, 0);
49      sem_init(&vacios, 0, BUFSIZE);
50
51      pthread_create(&prodtid, NULL, productor, NULL);
52      pthread_create(&constid, NULL, consumidor, NULL);
53
54      pthread_join(prodtid, NULL);
55      pthread_join(constid, NULL);
56
57      printf("terminado\n");
58
59      return 0;
60  }
61
```

Ln: 1, Col: 1

Run | Share | Command Line Arguments

```
main.c: In function 'consumidor':
main.c:33:9: warning: implicit declaration of function 'sleep' [-Wimplicit-function-declaration]
   33 |         sleep(1); // el consumo se demora 1 segundo
      |         ^~~~~

Session Killed due to Timeout.
```

Interacción:

- El productor genera caracteres y los coloca en el búfer.
- Si el búfer está lleno, el productor espera.
- El consumidor toma caracteres del búfer y los consume.
- Si el búfer está vacío, el consumidor espera.
- Ambos hilos trabajan en conjunto para producir y consumir caracteres.

Observaciones:

El código contiene un error en la función consumidor. El problema radica en que se llama dos veces seguidas a sem_wait(&llenos) dentro del bucle del consumidor. Esta duplicación de la llamada causará que el hilo del consumidor se bloquee indefinidamente en la segunda llamada a sem_wait(&llenos), ya que el productor no realiza una doble llamada a sem_post(&llenos) para liberar dos veces el semáforo.

• Prueba 2: comente la línea etiqueta como 1 en donde se crea el proceso productor y vuelva a compilar y ejecutar. Observe que como no hay proceso productor, el consumidor esta detenido por no tener nada que consumir.

Si se comenta la línea etiquetada como 1, el hilo del productor no se creará y no generará caracteres para el búfer. Como resultado, el hilo del consumidor quedará bloqueado esperando elementos en el búfer que nunca estarán disponibles. El programa se quedará en un estado bloqueado y no avanzará más allá de este punto.

• Prueba 3: ahora comente solo la línea etiqueta como 2 (creación del proceso consumidor). Observe que el productor se ejecutará hasta que se llene el almacén y luego se detendrá pues el proceso Consumidor no se ha creado.

En este escenario, el hilo del productor quedará bloqueado esperando a que haya espacio disponible en el búfer (vacios). Como no se ha creado el hilo del consumidor para liberar espacios en el búfer al consumir elementos, el hilo del productor se quedará esperando indefinidamente.

Tenga en cuenta que sólo hay un productor y un consumidor, sin embargo, es posible que se tenga más procesos productores que ejecuten la misma función, y más procesos consumidores. Por lo tanto, al existir por ejemplo varios productores se debe tener en cuenta que el acceso al almacén (arreglo buffer) se convierte en una SC, por lo tanto, se debe implementar los mecanismos necesarios para evitar resultados incorrectos y/o incoherentes

2. Utilizando exclusivamente Pthreads y semáforos contadores: La fábrica de bicicletas:

En una fábrica de bicicletas se tiene tres operarios OP1, OP2 y OP3. OP1 monta ruedas, OP2 monta el cuadro de las bicicletas, y OP3 el manillar. Un cuarto operario, llamado Montador, se encarga de tomar dos ruedas, un cuadro y un manillar, y ensamblar la bicicleta. Además, se sabe que:

- Los operarios OP2 y OP3 solo tienen espacio para almacenar N piezas de las que producen.
- El operador OP1 tiene espacio para M piezas.

Se le pide escribir el código de cada uno de estos operarios de forma que el accionar de cada uno se sincronice con los demás, para esto utilice los semáforos vistos en este laboratorio.

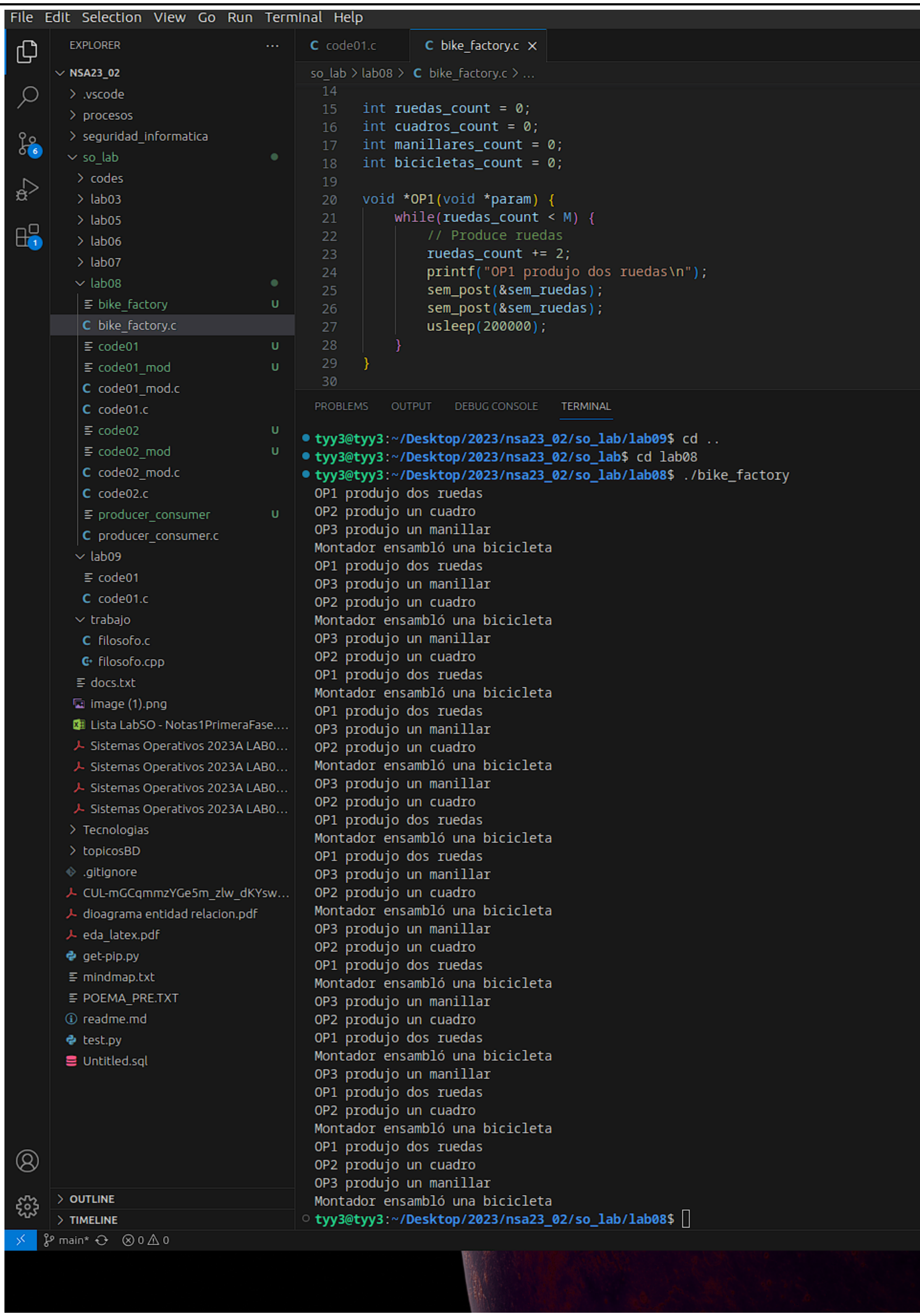
Code:

```c
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <semaphore.h>
#include <unistd.h>

#define N 10
#define M 20

sem_t sem_ruedas;
sem_t sem_cuadros;
sem_t sem_manillares;
sem_t sem_bicicletas;

int ruedas_count = 0;
int cuadros_count = 0;
int manillares_count = 0;
int bicicletas_count = 0;

void *OP1(void *param) {
    while(ruedas_count < M) {
        // Produce ruedas
        ruedas_count += 2;
        printf("OP1 produjo dos ruedas\n");
        sem_post(&sem_ruedas);
        sem_post(&sem_ruedas);
        usleep(200000);
```

```
    }
}

void *OP2(void *param) {
    while(cuadros_count < N) {
        // Produce cuadros
        cuadros_count++;
        printf("OP2 produjo un cuadro\n");
        sem_post(&sem_cuadros);
        usleep(200000);
    }
}

void *OP3(void *param) {
    while(manillares_count < N) {
        // Produce manillares
        manillares_count++;
        printf("OP3 produjo un manillar\n");
        sem_post(&sem_manillares);
        usleep(200000);
    }
}

void *Montador(void *param) {
    while(bicicletas_count < N) {
        sem_wait(&sem_ruedas);
        sem_wait(&sem_ruedas);
        sem_wait(&sem_cuadros);
        sem_wait(&sem_manillares);
        printf("Montador ensambló una bicicleta\n");
        bicicletas_count++;
        sem_post(&sem_bicicletas);
        usleep(200000);
    }
}
```

```c
int main() {
    sem_init(&sem_ruedas, 0, 0);
    sem_init(&sem_cuadros, 0, 0);
    sem_init(&sem_manillares, 0, 0);
    sem_init(&sem_bicicletas, 0, 0);

    pthread_t hilo_OP1, hilo_OP2, hilo_OP3, hilo_Montador;
    pthread_create(&hilo_OP1, NULL, OP1, NULL);
    pthread_create(&hilo_OP2, NULL, OP2, NULL);
    pthread_create(&hilo_OP3, NULL, OP3, NULL);
    pthread_create(&hilo_Montador, NULL, Montador, NULL);

    pthread_join(hilo_OP1, NULL);
    pthread_join(hilo_OP2, NULL);
    pthread_join(hilo_OP3, NULL);
    pthread_join(hilo_Montador, NULL);

    return 0;
}

// path: so_lab/lab08/bike_factory.c
// how to exec: gcc bike_factory.c -o bike_factory -lpthread
```

Pruebas:



File Edit Selection View Go Run Terminal Help
EXPLORER
NSA23_02
.vscode
procesos
seguridad_informatica
so_lab
codes
lab03
lab05
lab06
lab07
lab08
bike_factory
bike_factory.c
code01
code01_mod
code01_mod.c
code01.c
code02
code02_mod
code02_mod.c
code02.c
producer_consumer
producer_consumer.c
lab09
code01
code01.c
trabajo
filosofo.c
filosofo.cpp
docs.txt
image (1).png
Lista LabSO - Notas1PrimeraFase....
Sistemas Operativos 2023A LAB0...
Sistemas Operativos 2023A LAB0...
Sistemas Operativos 2023A LAB0...
Sistemas Operativos 2023A LAB0...
Tecnologias
topicosBD
.gitignore
CUL-mGCqmmzYGe5m_zlw_dKYsw...
dioagrama entidad relacion.pdf
eda_latex.pdf
get-pip.py
mindmap.txt
POEMA_PRE.TXT
readme.md
test.py
Untitled.sql
OUTLINE
TIMELINE
code01.c
bike_factory.c
so_lab > lab08 > bike_factory.c > ...
int ruedas_count = 0;
int cuadros_count = 0;
int manillares_count = 0;
int bicicletas_count = 0;
void *OP1(void *param) {
while(ruedas_count < M) {
// Produce ruedas
ruedas_count += 2;
printf("OP1 produjo dos ruedas\n");
sem_post(&sem_ruedas);
sem_post(&sem_ruedas);
usleep(200000);
}
}
PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL
tyy3@tyy3:~/Desktop/2023/nsa23_02/so_lab/lab09$ cd ..
tyy3@tyy3:~/Desktop/2023/nsa23_02/so_lab$ cd lab08
tyy3@tyy3:~/Desktop/2023/nsa23_02/so_lab/lab08$ ./bike_factory
OP1 produjo dos ruedas
OP2 produjo un cuadro
OP3 produjo un manillar
Montador ensambló una bicicleta
OP1 produjo dos ruedas
OP3 produjo un manillar
OP2 produjo un cuadro
Montador ensambló una bicicleta
OP3 produjo un manillar
OP2 produjo un cuadro
OP1 produjo dos ruedas
Montador ensambló una bicicleta
OP1 produjo dos ruedas
OP3 produjo un manillar
OP2 produjo un cuadro
Montador ensambló una bicicleta
OP3 produjo un manillar
OP2 produjo un cuadro
OP1 produjo dos ruedas
Montador ensambló una bicicleta
OP1 produjo dos ruedas
OP3 produjo un manillar
OP2 produjo un cuadro
Montador ensambló una bicicleta
OP3 produjo un manillar
OP2 produjo un cuadro
OP1 produjo dos ruedas
Montador ensambló una bicicleta
OP3 produjo un manillar
OP2 produjo un cuadro
OP1 produjo dos ruedas
Montador ensambló una bicicleta
OP3 produjo un manillar
OP1 produjo dos ruedas
OP2 produjo un cuadro
Montador ensambló una bicicleta
OP1 produjo dos ruedas
OP2 produjo un cuadro
OP3 produjo un manillar
Montador ensambló una bicicleta
tyy3@tyy3:~/Desktop/2023/nsa23_02/so_lab/lab08$
main*

3. Utilice PThreads para resolver el problema de la cena de los filósofos utilizando semáforos contadores.

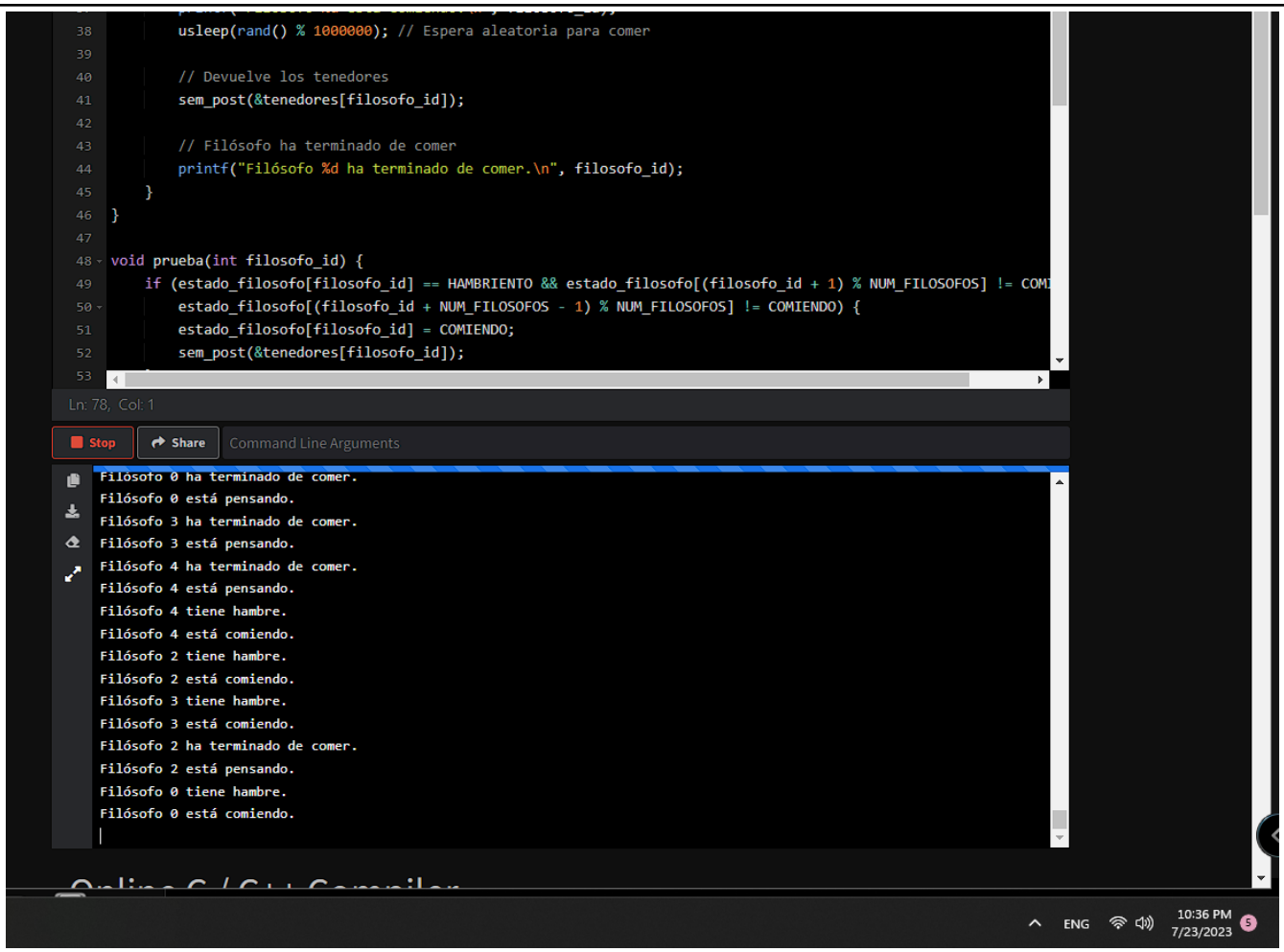
Codigo:

```c
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <semaphore.h>
#include <unistd.h>

#define NUM_FILOSOFOS 5
#define PENSANDO 0
#define HAMBRIENTO 1
#define COMIENDO 2

sem_t tenedores[NUM_FILOSOFOS];
sem_t mutex;

int estado_filosofo[NUM_FILOSOFOS];

void prueba(int filosofo_id); // Prototipo de la función prueba

void *filosofo(void *arg) {
    int filosofo_id = *(int *)arg;
    while (1) {
        // Filósofo está pensando
        printf("Filósofo %d está pensando.\n", filosofo_id);
        usleep(rand() % 1000000); // Espera aleatoria para pensar

        // Filósofo quiere comer
        sem_wait(&mutex);
        estado_filosofo[filosofo_id] = HAMBRIENTO;
        printf("Filósofo %d tiene hambre.\n", filosofo_id);
        prueba(filosofo_id); // Intenta tomar tenedores
        sem_post(&mutex);
```

```
        // Espera a que los tenedores estén disponibles
        sem_wait(&tenedores[filosofo_id]);

        // Filósofo está comiendo
        printf("Filósofo %d está comiendo.\n", filosofo_id);
        usleep(rand() % 1000000); // Espera aleatoria para comer

        // Devuelve los tenedores
        sem_post(&tenedores[filosofo_id]);

        // Filósofo ha terminado de comer
        printf("Filósofo %d ha terminado de comer.\n", filosofo_id);
    }
}

void prueba(int filosofo_id) {
                if     (estado_filosofo[filosofo_id]     ==     HAMBRIENTO     &&
estado_filosofo[(filosofo_id + 1) % NUM_FILOSOFOS] != COMIENDO &&
          estado_filosofo[(filosofo_id + NUM_FILOSOFOS - 1) % NUM_FILOSOFOS] !=
COMIENDO) {
        estado_filosofo[filosofo_id] = COMIENDO;
        sem_post(&tenedores[filosofo_id]);
    }
}

int main() {
    pthread_t filosofos[NUM_FILOSOFOS];
    int id_filosofos[NUM_FILOSOFOS];

    sem_init(&mutex, 0, 1);

    for (int i = 0; i < NUM_FILOSOFOS; i++) {
        sem_init(&tenedores[i], 0, 1);
        estado_filosofo[i] = PENSANDO;
        id_filosofos[i] = i;
```

```
    }

    for (int i = 0; i < NUM_FILOSOFOS; i++) {
                        pthread_create(&filosofos[i],   NULL,   filosofo,   (void
*)&id_filosofos[i]);
    }

    for (int i = 0; i < NUM_FILOSOFOS; i++) {
        pthread_join(filosofos[i], NULL);
    }

    return 0;
}
// path: so_lab\lab09\filosofos.c
// how to exec (linux): gcc filosofos.c -o filosofos -lpthread
```

Pruebas:



1. ¿Cuáles son las ventajas o desventajas encontradas del uso de Semáforos contadores?

Ventajas del uso de semáforos contadores:

- Proporcionan sincronización y exclusión mutua para recursos compartidos.
- Son flexibles en el número de recursos disponibles.
- Implementación eficiente en sistemas operativos modernos.
- Ayudan a prevenir bloqueos y condiciones de inanición.

Desventajas del uso de semáforos contadores:

- Pueden ser propensos a errores de programación si no se utilizan correctamente.
- Posibles situaciones de interbloqueo si no se diseñan adecuadamente.
- Dificultad para depurar problemas de sincronización.
- Mayor complejidad en el diseño y gestión para aplicaciones más complejas.

2. ¿En qué se diferencian los semáforos contadores a los semáforos vistos en el laboratorio anterior?
Al ser binarios los que usan contadores actúan como interruptores permitiendo o bloqueando el acceso a un recurso compartido y los contadores pueden tomar valores positivos para representar la cantidad de recursos disponibles para el acceso concurrente.

I. CONCLUSIONES

En el desarrollo del código y el aprendizaje sobre el uso de semáforos y sincronización en programación concurrente, he adquirido un mayor entendimiento sobre cómo manejar eficientemente recursos compartidos entre múltiples hilos o procesos. Los semáforos contadores y sin contadores son herramientas valiosas para garantizar la exclusión mutua y evitar problemas de sincronización.

El uso adecuado de semáforos contadores nos brinda flexibilidad para controlar diferentes cantidades de recursos disponibles, mientras que los semáforos sin contadores son útiles para situaciones "todo o nada". Sin embargo, es esencial tener en cuenta la complejidad y los posibles problemas de interbloqueo al implementar la sincronización.

RETROALIMENTACIÓN GENERAL

REFERENCIAS Y BIBLIOGRAFÍA

- *Abraham Silberschatz, Greg Gagne, Peter B. Galvin, Operating System Concepts, 10th Edition, 2018 John Wiley & Sons.*
- *William Stallings, Sistema Operativos, Aspectos internos y principios de diseño, 5ta Edición, 2015, Person Prentice Hall*