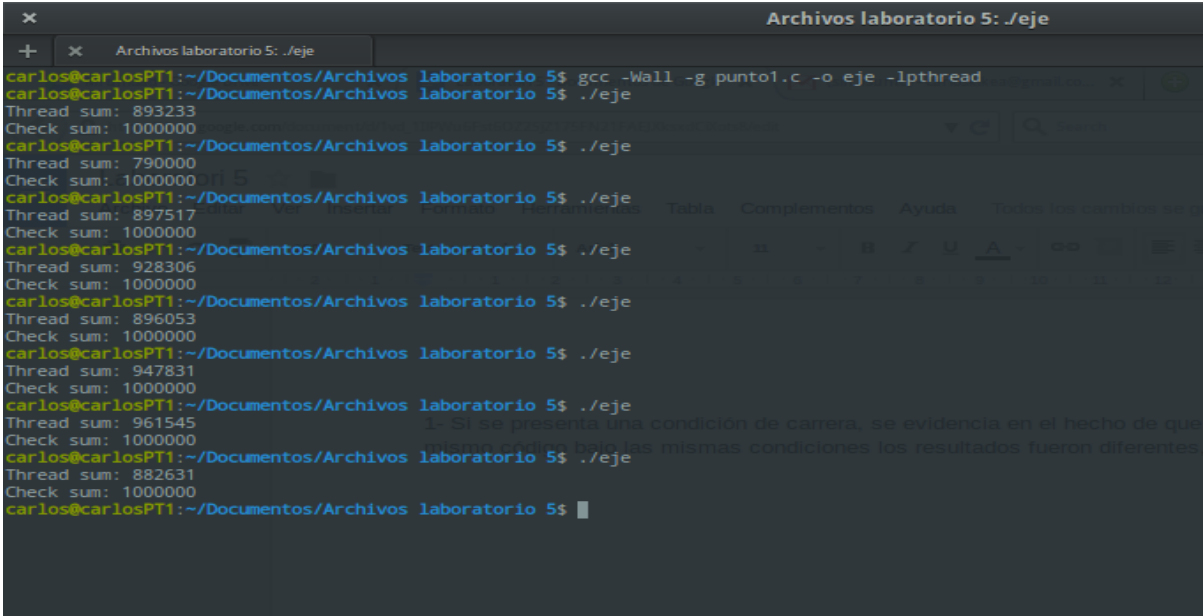


1- Si se presenta una condición de carrera, se evidencia en el hecho de que en ejecución del mismo código bajo las mismas condiciones los resultados fueron diferentes,



```
Archivos laboratorio 5: ./eje
carlos@carlosPT1:~/Documentos/Archivos laboratorio 5$ gcc -Wall -g punto1.c -o eje -lpthread
carlos@carlosPT1:~/Documentos/Archivos laboratorio 5$ ./eje
Thread sum: 893233
Check sum: 1000000
carlos@carlosPT1:~/Documentos/Archivos laboratorio 5$ ./eje
Thread sum: 790000
Check sum: 1000000
carlos@carlosPT1:~/Documentos/Archivos laboratorio 5$ ./eje
Thread sum: 897517
Check sum: 1000000
carlos@carlosPT1:~/Documentos/Archivos laboratorio 5$ ./eje
Thread sum: 928306
Check sum: 1000000
carlos@carlosPT1:~/Documentos/Archivos laboratorio 5$ ./eje
Thread sum: 896053
Check sum: 1000000
carlos@carlosPT1:~/Documentos/Archivos laboratorio 5$ ./eje
Thread sum: 947831
Check sum: 1000000
carlos@carlosPT1:~/Documentos/Archivos laboratorio 5$ ./eje
Thread sum: 961545
Check sum: 1000000
carlos@carlosPT1:~/Documentos/Archivos laboratorio 5$ ./eje
Thread sum: 882631
Check sum: 1000000
carlos@carlosPT1:~/Documentos/Archivos laboratorio 5$
```

2- La condición de carrera se presenta por la falta de sincronización entre los hilos que están realizando la misma acción, y modificando el mismo recurso compartido (variable global suma). de manera que el resultado que el resultado a la final va ser dependiente de la manera en que acceden y como llega el resultado.

3- La región crítica del sistema es la función “do_work”, que es el recurso compartido por todos los hilos, el cual a su vez es el que modifica la variable global sum. la cual debería ser modificada una vez por hilo.

4- La adición del mutex al código se presenta en el archivo punto1-1.c adjunto en la carpeta.

5- Con la implementación del mutex, podremos usar dos métodos bastante simples y de mucha ayuda a la hora de prevenir las condición de carrera, estos métodos son:

- **pthread_mutex_lock:** que me permite bloquear la porcion de codigo que esta siendo utilizado por un hilo, prohibiendo que esta sea usada por demas hilos en ejecución hasta que lo que lo tenga usado termine.
- **pthread_mutex_unlock:** permite desbloquear los recursos que estaba utilizando el hilo anterior para poder ser asignados a un nuevo hilo.

6- las funciones funcionan de la siguiente forma:

- **sem_init:** es la función que se usa para la inicialización de semáforos, esta recibe los siguientes parámetros:
 - sem_t *m: es el apuntador a una variable de tipo semáforo que se quiere inicializar.
 - int pshared: este valor indica si el semáforo será de uso compartido entre hilos.
 - unsigned int value: valor inicial del semáforo.

- **sem_wait:** es la función con la que esperamos que el semáforo acabe su ejecución se le envía como parámetro al apuntador de dicho semáforo.
- **sem_post:** incrementa el valor de un semáforo , se le envía el valor de dicho semáforo.
- **sem_destroy:** destruye un semáforo enviando el apuntador de este.

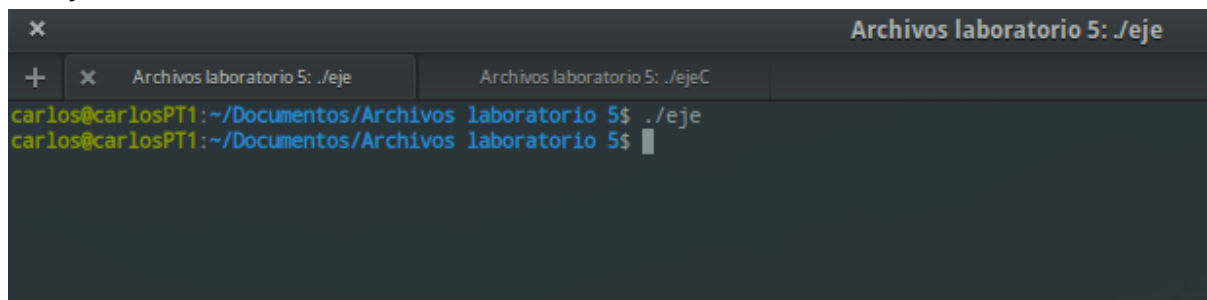
7- La diferencia radica en que el semáforo permite concurrencia en la región crítica del sistema, a través del control de hilos que pueden acceder a esta región.

8-

- **shmget:** función para obtener el identificador de memoria compartida referenciar segmentos de la misma en la cual varios procesos o hilos pueden hacer uso de esta, recibe estos tres argumentos.
 - key: referencia el nombre o código único mediante el cual vamos a identificar el segmento compartido.
 - size: indica el tamaño redondeado que tendrá el segmento de memoria compartida.
 - ipc_creat: crea un nuevo segmento, si flag no está ocupado entonces se asocia con el identificador key y verifica que el usuario tenga permisos para acceder a este segmento.
- **shmat:** función que une el proceso de memoria compartida con el proceso que llama la función, recibe estos tres argumentos:
 - shmid: hace referencia al identificador de segmento de memoria compartida.
 - *shmaddr: indica la dirección específica en el espacio de memoria compartida del proceso en el cual vamos a unir el segmento de memoria.
 - shmflag: pasa algunas configuraciones más personalizadas que deseamos realizar en el segmento.
- **shmdt:** desasocia el segmento de datos del proceso de segmentos de memoria compartida, retorna 0 en caso de éxito, de lo contrario retorna 1, reciben los mismo parámetros y es el proceso inverso a shmat.
- **shmctl:** esta función permite realizar un conjunto de operaciones de control sobre una zona de memoria compartida.

9- Se analizó y verificó el uso de los parámetros entregados a las funciones.

10- ejecución servidor:

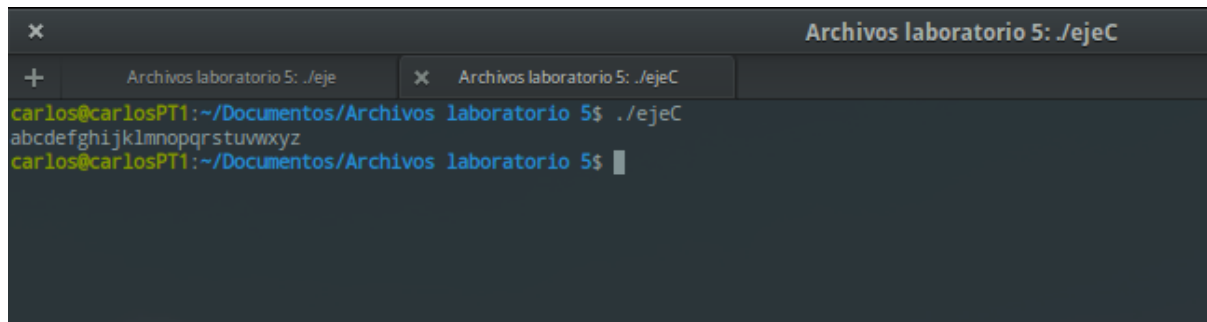


The screenshot shows a terminal window titled "Archivos laboratorio 5: ./eje". The prompt is "carlos@carlosPT1:~/Documentos/Archivos laboratorio 5\$". The user has entered the command "./eje" and the terminal is waiting for input.

```

x
+ x Archivos laboratorio 5: ./eje Archivos laboratorio 5: ./ejeC
carlos@carlosPT1:~/Documentos/Archivos laboratorio 5$ ./eje
carlos@carlosPT1:~/Documentos/Archivos laboratorio 5$
  
```

ejecución del cliente:



```
Archivos laboratorio 5: ./ejeC
+ Archivos laboratorio 5: ./eje
x Archivos laboratorio 5: ./ejeC
carlos@carlosPT1:~/Documentos/Archivos laboratorio 5$ ./ejeC
abcdefghijklmnopqrstuvwxyz
carlos@carlosPT1:~/Documentos/Archivos laboratorio 5$
```

El programa server lo que hace es crear un segmento de memoria compartida, con la key 1234, en este almacena el abecedario y lo deja a disposición hasta que el cliente lo lea, una vez el cliente lo lee este finaliza su ejecución.

El programa cliente lo que hace es acceder al segmento de memoria generado por el programa server de key 1234, para leerlo y después ser mostrado en la salida de la ejecución de este.

Lo que pasa si se invierte el orden de ejecución de los programas es que el cliente escribe lo que el programa server muestra en salida lo que la anterior ejecución del programa server había guardado en memoria compartida en la ejecución anterior y el server queda en espera de que una nueva ejecución del programa cliente lea lo que tiene en la memoria compartida para terminar la ejecución.

11- Es obligatorio para todos los procesos seguir este flujo excepto para el creador del segmento compartido, si un proceso no se desadhiere lo que pasa si se vuelve a ejecutar ese mismo proceso es que este encontrará información errada lo cual haría que los cálculos realizados fueran muy poco confiables. Pero si es el proceso creador el que no se desadhiere lo que pasa es que este espacio quedará en desuso y acaba con los resultados del sistema.

12- Se adjunta el código con nombre Punto12.c a la carpeta.

13- Se adjunta el código con nombre Punto13.c a la carpeta.

14- Se adjunta el código con nombre Punto14.c a la carpeta.