SISTEMAS OPERATIVOS I

INFORME SOBRE MAPA DE MEMORIA

Por:

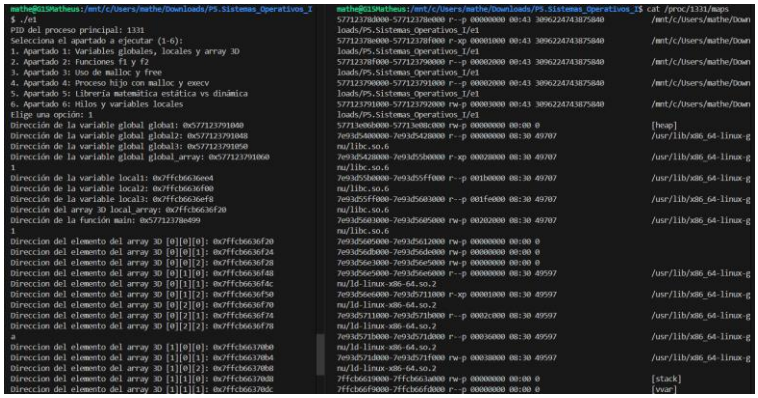
- ***Matheus Muiños Kruschewsky***
- ***Alejandro Alonso Muiños***

INTRODUCCIÓN

El objetivo de esta práctica es realizar un análisis exhaustivo del mapa de memoria de un proceso. Para conseguirlo, hemos desarrollado un programa en C que simula seis situaciones clave de gestión de memoria, como son el manejo de variables estáticas y locales, la asignación dinámica, el uso de hilos, la vinculación de librerías y la creación de procesos hijos. La metodología principal ha sido examinar el archivo /proc/PID/maps en diversos momentos de la ejecución de cada apartado. Esto nos ha permitido identificar y delimitar las regiones fundamentales y entender cómo el sistema operativo gestiona la memoria para cada proceso.

EJERCICIO 1

-Resultados:



- Al ejecutar el código y buscar el mapa de memoria del proceso creado para su ejecución, se puede ver la dirección de memoria de la función main (0x57712378e499). Cuando buscamos en el mapa de memoria el intervalo donde se encuentra esta dirección, vemos que está en el intervalo marcado como 57712378d000–57712378f000 r-xp. Este marco de permiso r-xp (lectura y ejecución, sin escritura) está asociado al fichero ejecutable del programa y se corresponde con la región de memoria conocida como el segmento TEXT. Esto confirma que el código de la función main y, en general, todas las instrucciones del programa residen en esta área fija del proceso.

- Cuando buscamos las variables globales del programa, vemos que presentan direcciones en el rango 0x577123791040–0x577123791060. Estas direcciones se sitúan dentro de una zona del mapa de memoria marcada con permisos rw-p (577123791000-577123792000). El permiso de lectura y escritura indica que se trata del segmento de datos estáticos. Esto confirma que todas las variables globales se almacenan en la región de datos globales del proceso, donde permanecen accesibles durante toda la vida de la ejecución.

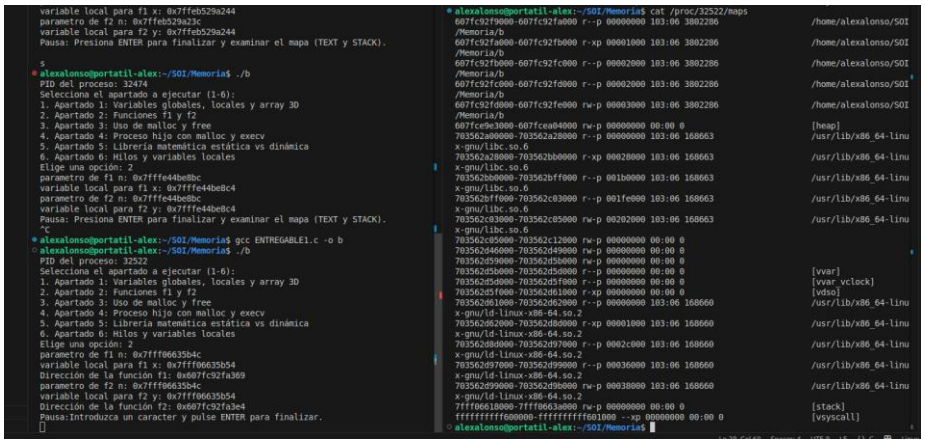
- Para las variables locales, estas presentan direcciones del tipo 0x7fffcb6636e4, 0x7fffcb6636f0 y valores similares. Al compararlas con el mapa de memoria, se verifica que caen dentro de un intervalo identificado como [stack]. Esta región corresponde al segmento de pila del proceso, utilizado para almacenar variables automáticas, parámetros de funciones y direcciones de retorno. El hecho de que las variables locales estén en esta zona confirma que su duración es automática (solo existen mientras la función está activa).

- El array tridimensional declarado como variable local también presenta una dirección de inicio dentro del rango 0x7fffcb66xxxx, coincidiendo con el mismo intervalo del mapa de memoria marcado como [stack]. Esto indica que el array 3D se reserva en la pila, junto con el resto de las variables locales. Al ser un array local, su memoria se asigna al entrar en la función y se libera automáticamente al salir de ella.

- Para estudiar el orden de almacenamiento del array tridimensional, se imprimieron las direcciones de cada uno de sus elementos. Las direcciones muestran que los elementos [i][j][k], [i][j][k+1] y [i][j][k+2] ocupan posiciones contiguas en memoria, con incrementos constantes en la dirección, mientras que los elementos [i][j+1][0] aparecen a continuación de terminar la secuencia del último índice. Este comportamiento demuestra que el índice más interno (k) es el que varía más rápido, seguido por el índice j y finalmente por i. Dicho patrón coincide con el esquema de almacenamiento row-major utilizado por el lenguaje C para matrices multidimensionales, en el cual el array 3D se representa linealmente en memoria siguiendo el orden natural de los índices desde el más interno al más externo.

EJERCICIO 2

-Resultados:



-El código de las funciones f1 y f2 forma parte del ejecutable del programa. En el mapa de memoria se observa que pertenecen a la región asociada al fichero del ejecutable con permisos r-xp. Como ya se ha dicho anteriormente, este tipo de permiso (lectura y ejecución) identifica el segmento de texto, donde se almacena el código de todas las funciones del programa.

-Durante la ejecución, el programa muestra que la dirección del parámetro n en f1 y en f2 es 0x7fff06635b4c. Al comparar esta dirección con el mapa de memoria del proceso, se comprueba que cae dentro del intervalo marcado como [stack]. Esta región corresponde al segmento de pila, utilizado para almacenar los marcos de activación de las funciones. En consecuencia, los parámetros enteros pasados a f1 y f2 se almacenan en la pila cuando dichas funciones están en ejecución.

-La dirección de las variables locales x (en f1) e y (en f2) es 0x7fff06635b54, muy próxima a la dirección del parámetro n. Estas direcciones también se encuentran dentro de la región [stack] del mapa de memoria. Esto indica que las variables locales de cada función se almacenan en la pila del proceso, compartiendo el mismo marco de activación que sus parámetros.

<u>Observación sobre la coincidencia de direcciones:</u>

Es importante destacar que el hecho de que f1 y f2 muestren exactamente las mismas direcciones para sus parámetros y sus variables locales no constituye un error, sino que es el comportamiento esperado.

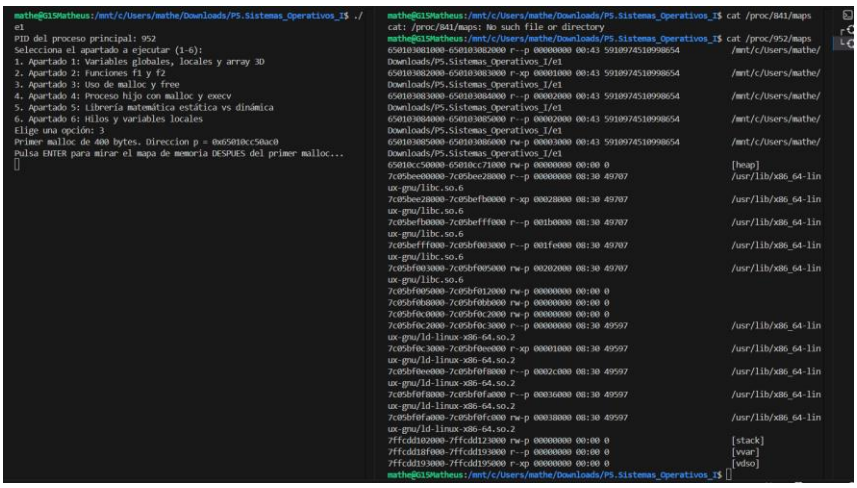
Ambas funciones se ejecutan de forma secuencial y utilizan la misma región de la pila para sus marcos de activación. Cuando f1 termina, su espacio en la pila se libera, y la llamada siguiente a f2 reutiliza exactamente las mismas posiciones de memoria. Por este motivo, las direcciones impresas para el parámetro n y para las variables locales x e y coinciden en ambas funciones. Esto confirma el funcionamiento típico de la gestión de la pila en programas no recursivos y con funciones que poseen estructuras de activación equivalentes.

Sin embargo, las funciones f1 y f2 residen en el segmento TEXT del programa. Este segmento es estático y de solo lectura (r-xp). La dirección de inicio de cada función es fija y se determina en el momento de la carga del programa, por lo que nunca se libera ni se reutiliza. A diferencia de la pila, que es un espacio de trabajo temporal, el código es una parte permanente del proceso.

EJERCICIO 3

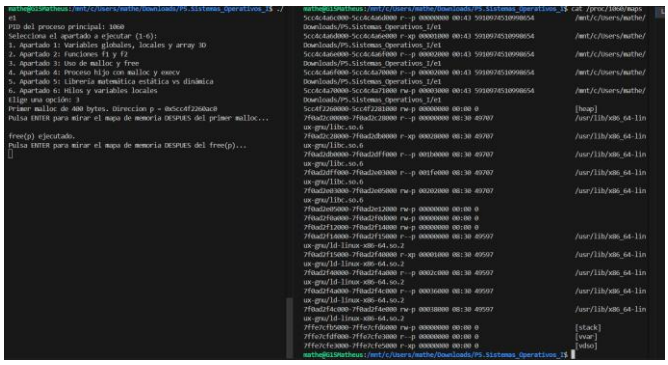
-Resultados:

-Mapa de memoria después del primer malloc (400 bytes):



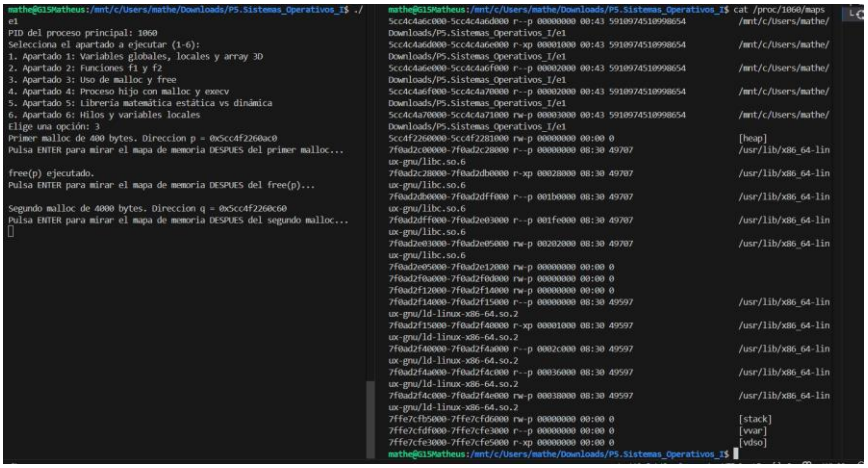
En esta captura podemos ver que el proceso tiene una región marcada como [heap], que es la zona donde el sistema guarda la memoria dinámica. El puntero p devuelto por el primer malloc tiene la dirección 0x5cc4f2260ac0, que pertenece al rango que el sistema etiqueta como heap. Esto significa que la memoria que hemos reservado (400 bytes) está efectivamente ubicada dentro de la región dinámica del proceso. Por tanto, comprobamos que malloc utiliza la heap para las reservas pequeñas y que esta región aparece claramente identificada en el mapa de memoria.

-Mapa de memoria después del free(p).



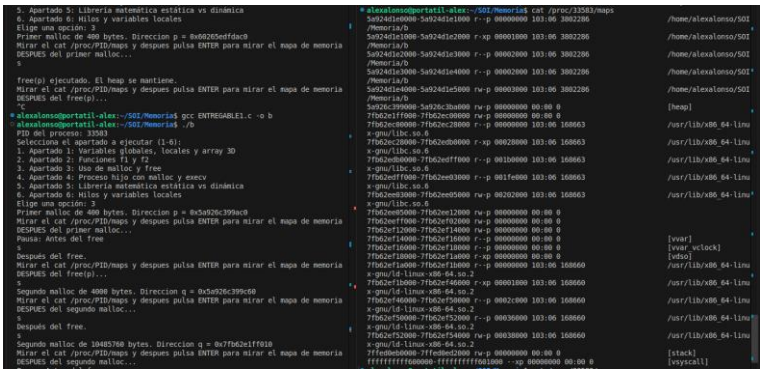
En esta captura se observa que la región [heap] sigue apareciendo exactamente igual que en la captura anterior. Aunque ya hemos llamado a free(p), el tamaño de la heap no cambia y tampoco desaparece del mapa de memoria. Esto se debe a que free no libera la zona completa de la heap en el sistema operativo, sino que simplemente marca internamente el bloque como “libre” para que el propio programa pueda reutilizarlo más adelante. Por eso, desde el punto de vista del mapa de memoria del proceso, no hay ningún cambio visible, aunque el free se haya ejecutado correctamente.

-Mapa de memoria después del segundo malloc (4000 bytes).



Después de reservar un segundo bloque de memoria más grande (4000 bytes), el puntero recibido también corresponde a una dirección que pertenece a la región [heap]. En el mapa de memoria se puede ver que la heap continúa existiendo como antes, sin desaparecer ni dividirse. Además, el tamaño de la región tampoco cambia de forma visible, lo cual indica que la heap ya disponía de suficiente espacio interno para atender esta nueva reserva.

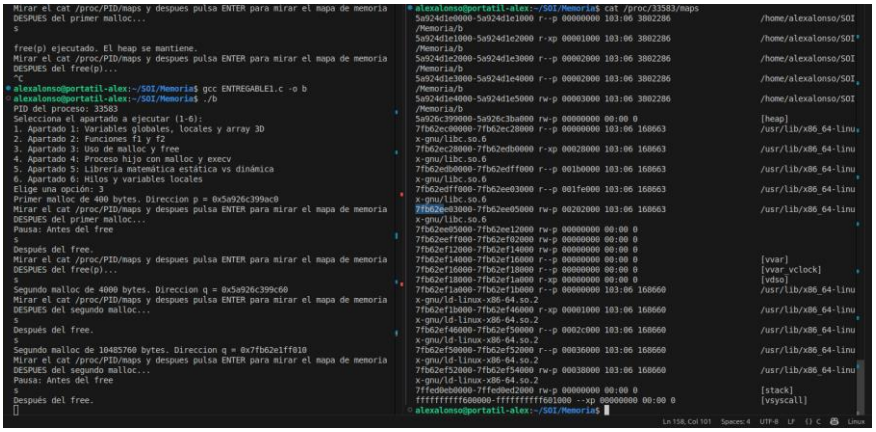
-Mapa de memoria despues del tercer malloc (10MB).



En esta captura podemos ver que la dirección devuelta por el malloc no pertenece a la región [heap] ya existente. Esto se debe a que la memoria reservada (10 MB) supera el umbral interno de asignación de la

biblioteca C, lo que obliga a utilizar la llamada al sistema mmap() en lugar del gestor del Heap tradicional. Por lo tanto, se crea una nueva región de memoria anónima en el mapa, identificada únicamente por sus permisos rw-p y el rango de direcciones.

-Mapa de memoria después de liberar los 10 MB del último malloc.



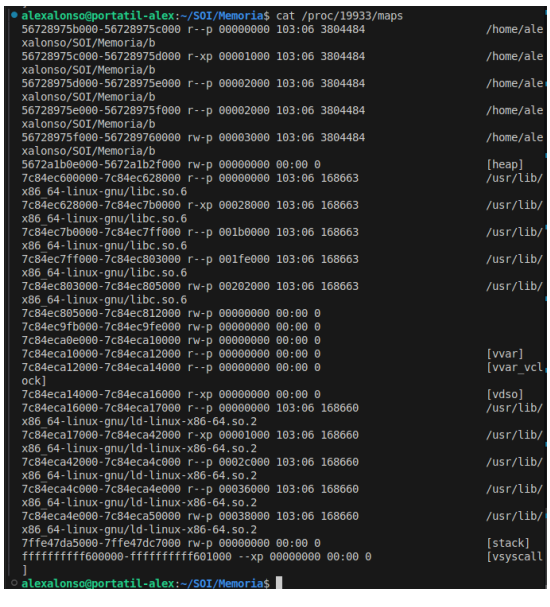
Al ejecutar free() sobre el bloque grande (10 MB), observamos que la región de memoria anónima que se había creado para esta reserva desaparece completamente del mapa de memoria. Esto sucede porque, dado que el bloque fue asignado inicialmente con la llamada al sistema mmap(), la función free() del gestor de la biblioteca C invoca a munmap(). Esta acción es la que desvincula el rango de direcciones virtuales de la memoria física y devuelve inmediatamente la RAM al sistema operativo. Por lo tanto, se confirma que las asignaciones grandes no se gestionan dentro del Heap tradicional, sino como mapeos temporales de memoria.

EJERCICIO 4

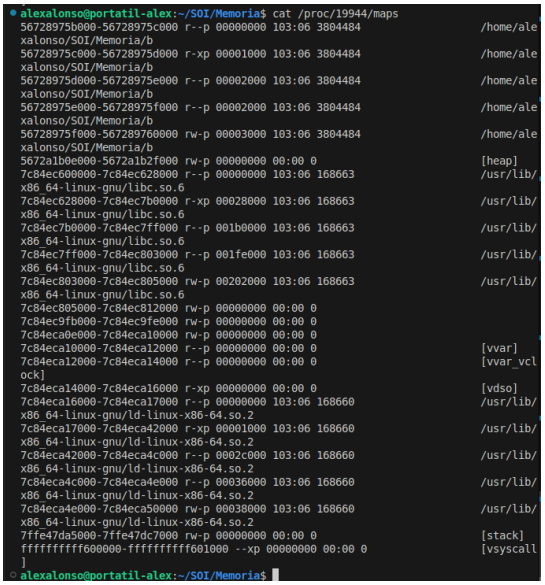
-Resultados:

-Después del fork:

Mapa del padre (PID=19933)



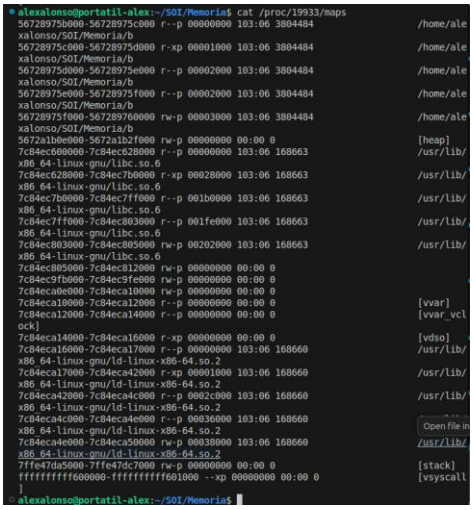
Mapa del hijo (PID=19944)



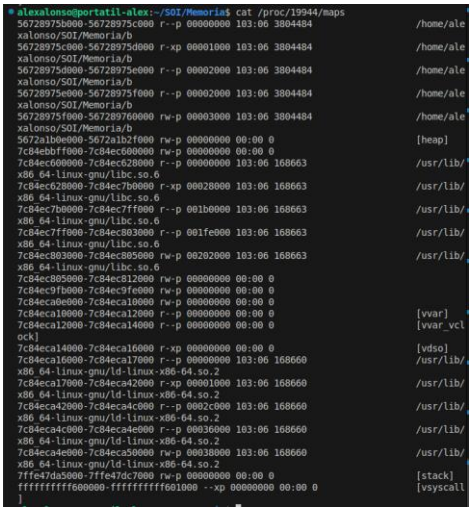
Observamos que el hijo hereda el código, las regiones de datos (variables) y el HEAP en las mismas direcciones virtuales que el padre. Esto es debido a que el fork() crea un espacio de direcciones virtualmente idéntico. Sin embargo, el sistema asigna una nueva Tabla de Páginas al hijo, y utiliza el mecanismo de Copia en Escritura , haciendo que las páginas de memoria física sean compartidas inicialmente, pero se dupliquen solo cuando el padre o el hijo intenten escribir, garantizando así un aislamiento total entre ambos procesos.

-Después del malloc en el hijo:

Mapa del padre (PID=19933)

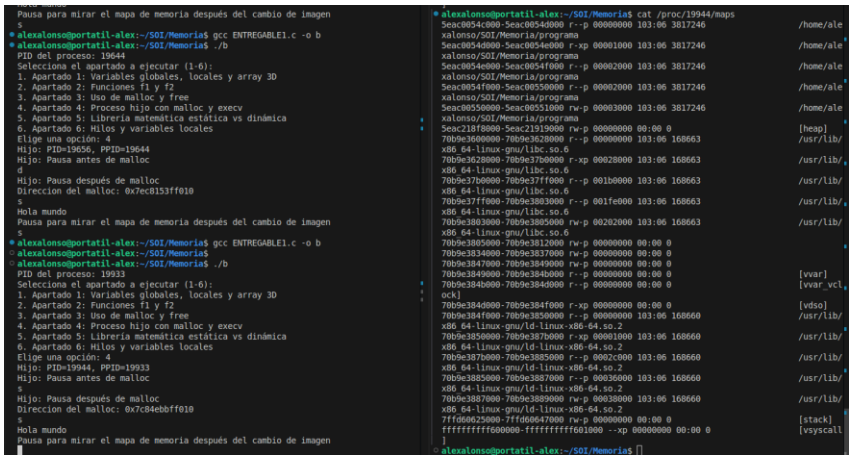


Mapa del hijo (PID=19944)



Observamos que en el mapa de memoria del padre no existe ningún cambio. Por el contrario, al ejecutar el malloc, el proceso hijo crea una nueva región de memoria para su propio *Heap* o un mapeo anónimo (si la reserva es grande). Esta nueva región es privada y única para el hijo, ya que, al necesitar un espacio de escritura, el mecanismo Copy-on-Write garantiza que la memoria física asignada al hijo sea completamente independiente de la memoria del padre.

-Después del cambio de imagen:



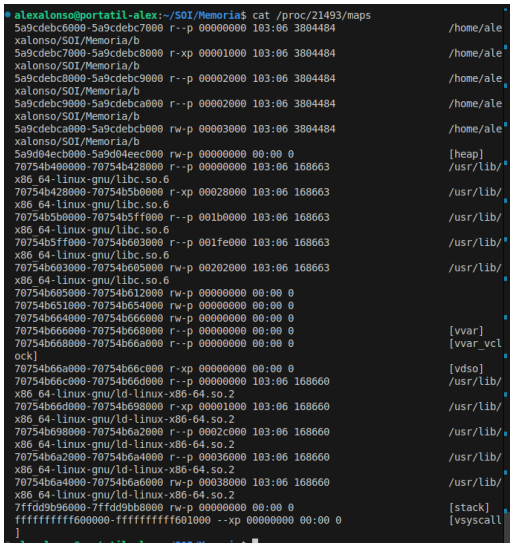
Al ejecutar execv o execlp, se produce una destrucción y reconstrucción completa del espacio de direcciones del proceso hijo, reemplazando el mapa de memoria original. Esto significa que todas las regiones previas heredadas del padre, incluyendo el código (TEXT), los datos estáticos (DATA/BSS) y cualquier memoria dinámica asignada previamente con malloc o mmap (incluso si era grande), son eliminadas. El mapa de memoria resultante solo contendrá las regiones esenciales (TEXT, DATA, STACK) del nuevo programa cargado, logrando que el hijo ejecute un código totalmente nuevo sin rastro de su predecesor.

EJERCICIO 5

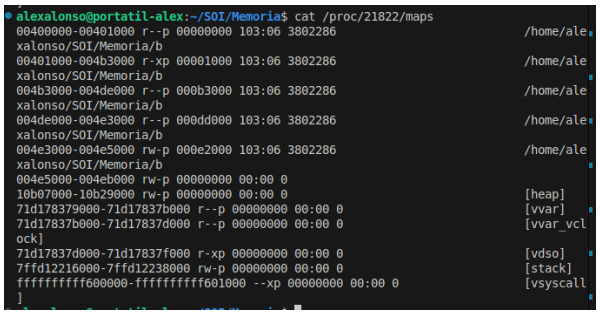
-Resultados:

-Enlazamos de forma dinámica y estática:

Mapa de memoria (-lm)



Mapa de memoria (-static –lm)

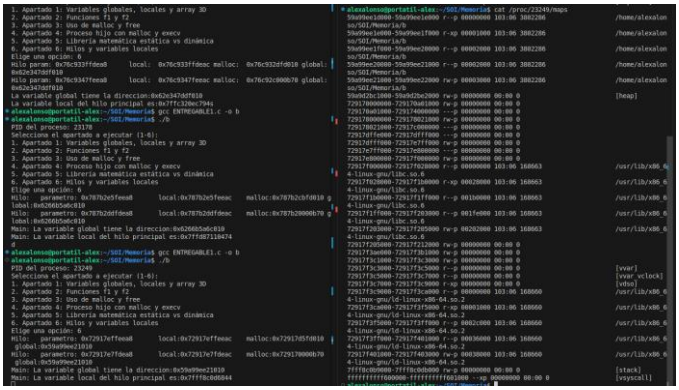


El enlace dinámico (por defecto, usando -lm) genera un mapa de memoria con múltiples entradas, ya que la librería matemática (libm.so) se carga como un módulo independiente en una región de memoria separada, visible como una línea r-xp en el mapa. Por el contrario, el enlace estático (usando -static -lm) produce un mapa significativamente más corto. En este caso, el código de la librería se fusiona directamente con el código del ejecutable principal. El resultado es que el segmento de código (r-xp) del binario estático es mucho más grande y las entradas separadas para las librerías (.so) desaparecen por completo, aumentando el tamaño de la región TEXT a costa de la flexibilidad en tiempo de ejecución.

EJERCICIO 6

-Resultados:

- Creamos los hilos, imprimimos direcciones de memoria y observamos el mapa:



- Localizamos las direcciones en el mapa y las regiones en las que se encuentran:

-Variable global:

Observamos que tanto en el main como en los dos hilos la dirección de memoria es la misma:0x573c8a9b9010.

Se ubica en la región DATA/BSS del proceso. En el mapa de memoria, esta región se identifica por tener la ruta del archivo ejecutable y permisos de rw-p (lectura y escritura). Se sitúa en direcciones bajas, justo después del código del programa. Esta memoria es compartida por todos los hilos del proceso.

-Variable local main:

Observamos que la dirección de memoria 0x7ffc316e6554 se corresponde con el stack.

La variable local del hilo principal se ubica en la pila principal del proceso. En el mapa de memoria podemos identificar esta región buscando la etiqueta [stack] y comprobando que tiene permisos de rw-p (lectura y escritura). Su dirección de memoria está en el rango más alto del espacio de direcciones virtuales, y aunque el valor de esta variable local se pasa a otros hilos, su propia ubicación en el [stack] del hilo principal permanece inalterada y aislada de las pilas de los hilos secundarios.

-Parámetros y variables locales de cada hilo:

Observamos que para cada hilo, parámetro y variable local pertenecen a la misma región de memoria.

Estas se ubican en las pilas privadas de cada hijo. Aparecen como bloques de memoria anónima que, aunque se encuentran en rangos de direcciones distintos entre sí y separadas del [stack], garantizan el aislamiento del contexto de ejecución de cada hilo. Estas regiones dinámicas y separadas se asignan con mmap, lo que explica por qué sus direcciones son muy diferentes a la pila principal y a las pilas de los otros hilos, y por qué parecen estar en la misma zona que el heap.

-Memoria dinámica creada por malloc:

Observamos que las dos direcciones de memoria creadas por el malloc pertenecen a la misma región: 77e5a15fc000-77e5a29fe000 rw-p.

Se ubican en el heap, que es una región de memoria compartida por todos los hilos del proceso, etiquetada como [heap] y con permisos rw-p. Las llamadas a malloc realizadas por el hilo principal y por los hilos secundarios asignan memoria desde este mismo espacio, resultando en que todas las asignaciones dinámicas se encuentren en el mismo rango de direcciones virtuales. Debido a que el HEAP es un recurso compartido, la biblioteca C debe implementar la gestión de malloc de forma segura para hilos para prevenir conflictos de asignación concurrentes.